Automated Machine Learning
(AutoML)

M. Lindauer F. Hutter

University of Freiburg

AutoML.org

Lectures 11:
Beyond AutoML: Algorithm Configuration and Control

Where are we? The big picture

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

$\rightarrow$ Beyond AutoML: algorithm configuration and control

- Project announcement and closing

Learning Goals

After this lecture, you will be able to . . .

- define the algorithm configuration problem
- discuss differences between HPO and algorithm configuration
- explain the components of SMAC to combine Bayesian Optimization across instances
- define and give examples for the algorithm control problem
- list some further topics related AutoML and algorithm configuration

Further Material

There is one recent tutorials on algorithm configuration:

- ICML'19: Frank Hutter and Kevin Leyton-Brown on "Algorithm configuration: learning in the space of algorithm designs" `https://www.facebook.com/icml.imls/videos/vb.118896271958230/2044426569187107/`

Lecture Overview

1. Algorithm Configuration
2. SMAC: BO for Algorithm Configuration
3. Algorithm Control
4. Other Related Topics

Generalization of HPO

- hyperparameter optimization (HPO) is not limited to ML

Generalization of HPO

- hyperparameter optimization (HPO) is not limited to ML
- in fact, you can optimize the performance of any algorithm by means of HPO if
  1. the algorithm at hand has parameters that influence its performance
  2. you care about the empirical performance of an algorithm

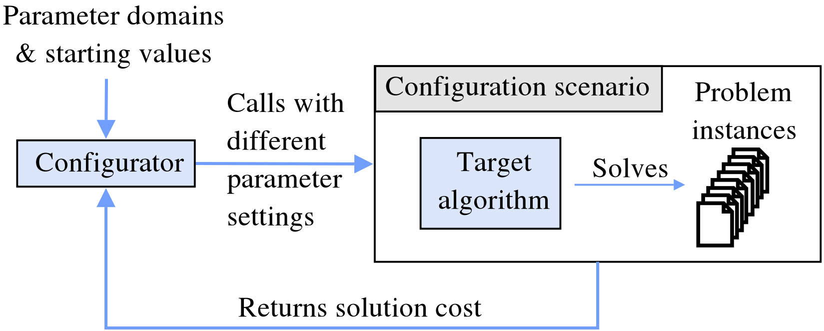
Generalization of HPO

- hyperparameter optimization (HPO) is not limited to ML
- in fact, you can optimize the performance of any algorithm by means of HPO if
  1. the algorithm at hand has parameters that influence its performance
  2. you care about the empirical performance of an algorithm
- a limitation of HPO is that we assume that we care only about a single task (i.e., dataset or input to the algorithm)

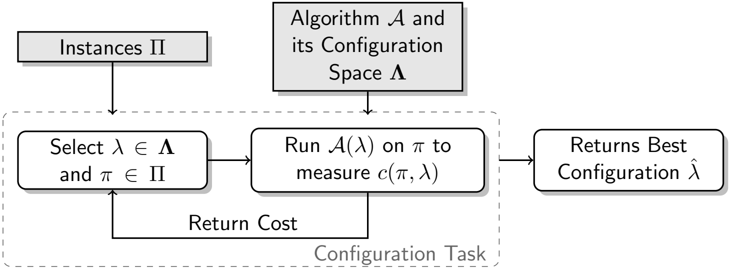
⇝ Can we find an algorithm's configuration that performs well and robustly across a set of tasks?
  - An hyperparameter configuration for a set of datasets
  - A parameter configuration of a SAT solver for a set of SAT instances
  - A parameter configuration of a AI planning solver for a set of planning problems
  - ...

⇝ Algorithm configuration

Algorithm Configuration Visualized



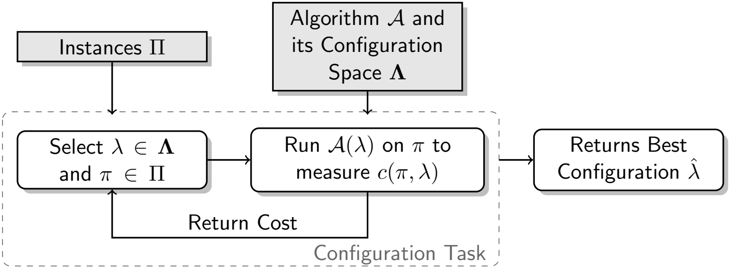
Algorithm Configuration – in More Detail



Definition

Given a parameterized algorithm $\mathcal{A}$ with possible parameter settings $\mathbf{\Lambda}$,

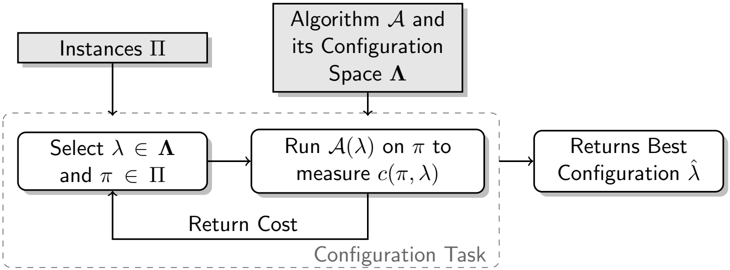
Algorithm Configuration – in More Detail



Definition

Given a parameterized algorithm $\mathcal{A}$ with possible parameter settings $\mathbf{\Lambda}$, a set of training problem instances Π,

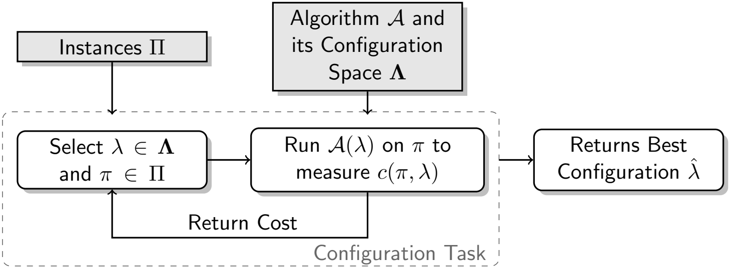
Algorithm Configuration – in More Detail



Definition

Given a parameterized algorithm $\mathcal{A}$ with possible parameter settings $\mathbf{\Lambda}$, a set of training problem instances Π, and a cost metric $c : \mathbf{\Lambda} \times \Pi \to \mathbb{R}$,

Algorithm Configuration – in More Detail



Definition

Given a parameterized algorithm $\mathcal{A}$ with possible parameter settings $\mathbf{\Lambda}$, a set of training problem instances Π, and a cost metric $c : \mathbf{\Lambda} \times \Pi \to \mathbb{R}$, the algorithm configuration problem is to find a parameter configuration $\lambda^* \in \mathbf{\Lambda}$ that minimizes c across the instances in Π.

Algorithm Configuration – Full Formal Definition

Definition

An instance of the algorithm configuration problem is a 5-tuple $(\mathcal{A}, \mathbf{\Lambda}, \mathcal{D}, \kappa, c)$ where:

- $\mathcal{A}$ is a parameterized algorithm;
- $\mathbf{\Lambda}$ is the parameter configuration space of $\mathcal{A}$;
- $\mathcal{D}$ is a distribution over problem instances with domain Π;

Algorithm Configuration – Full Formal Definition

Definition

An instance of the algorithm configuration problem is a 5-tuple $(\mathcal{A}, \mathbf{\Lambda}, \mathcal{D}, \kappa, c)$ where:

- $\mathcal{A}$ is a parameterized algorithm;
- $\mathbf{\Lambda}$ is the parameter configuration space of $\mathcal{A}$;
- $\mathcal{D}$ is a distribution over problem instances with domain Π;
- $\kappa < \infty$ is a cutoff time, after which each run of $\mathcal{A}$ will be terminated if still running

Algorithm Configuration – Full Formal Definition

Definition

An instance of the algorithm configuration problem is a 5-tuple $(\mathcal{A}, \mathbf{\Lambda}, \mathcal{D}, \kappa, c)$ where:

- $\mathcal{A}$ is a parameterized algorithm;
- $\mathbf{\Lambda}$ is the parameter configuration space of $\mathcal{A}$;
- $\mathcal{D}$ is a distribution over problem instances with domain Π;
- $\kappa < \infty$ is a cutoff time, after which each run of $\mathcal{A}$ will be terminated if still running
- $c : \mathbf{\Lambda} \times \Pi \rightarrow \mathbb{R}$ is a function that measures the observed cost of running $\mathcal{A}(\lambda)$ on an instance $\pi \in \Pi$ with cutoff time κ

Algorithm Configuration – Full Formal Definition

Definition

An instance of the algorithm configuration problem is a 5-tuple $(\mathcal{A}, \mathbf{\Lambda}, \mathcal{D}, \kappa, c)$ where:

- $\mathcal{A}$ is a parameterized algorithm;
- $\mathbf{\Lambda}$ is the parameter configuration space of $\mathcal{A}$;
- $\mathcal{D}$ is a distribution over problem instances with domain Π;
- $\kappa < \infty$ is a cutoff time, after which each run of $\mathcal{A}$ will be terminated if still running
- $c : \mathbf{\Lambda} \times \Pi \rightarrow \mathbb{R}$ is a function that measures the observed cost of running $\mathcal{A}(\lambda)$ on an instance $\pi \in \Pi$ with cutoff time κ

The cost of a candidate solution $\lambda \in \mathbf{\Lambda}$ is $f(\lambda) = \mathbb{E}_{\pi \sim \mathcal{D}}(c(\lambda, \pi))$.
The goal is to find $\lambda^* \in \arg\min_{\lambda \in \mathbf{\Lambda}} f(\lambda)$.

Distribution of Instances

We usually have a finite number of instances from a given application

- We want to do well on that type of instances
- Future instances of this type should be solved well

Distribution of Instances

We usually have a finite number of instances from a given application

- We want to do well on that type of instances
- Future instances of this type should be solved well

Like in machine learning

- We split the instances into a training set and a test set
- We configure algorithms on the training instances
- We only use the test instances afterwards
  - $\rightarrow$ unbiased estimate of generalization performance for unseen instances

Challenges of Algorithm Configuration

- Structured high-dimensional parameter space
  - categorical vs. continuous parameters
  - conditionals between parameters

Challenges of Algorithm Configuration

- Structured high-dimensional parameter space
  - categorical vs. continuous parameters
  - conditionals between parameters
- Stochastic optimization
  - Randomized algorithms: optimization across various seeds
  - Distribution of benchmark instances (often wide range of hardness)
  - Subsumes so-called *multi-armed bandit problem*

Challenges of Algorithm Configuration

- Structured high-dimensional parameter space
  - categorical vs. continuous parameters
  - conditionals between parameters
- Stochastic optimization
  - Randomized algorithms: optimization across various seeds
  - Distribution of benchmark instances (often wide range of hardness)
  - Subsumes so-called *multi-armed bandit problem*
- Generalization across instances
  - apply algorithm configuration to homogeneous instance sets
  - Instance sets can also be heterogeneous,
    i.e., no single configuration performs well on all instances

  $\rightsquigarrow$ combination of algorithm configuration and selection

Challenges of Algorithm Configuration

- Structured high-dimensional parameter space
  - categorical vs. continuous parameters
  - conditionals between parameters
- Stochastic optimization
  - Randomized algorithms: optimization across various seeds
  - Distribution of benchmark instances (often wide range of hardness)
  - Subsumes so-called *multi-armed bandit problem*
- Generalization across instances
  - apply algorithm configuration to homogeneous instance sets
  - Instance sets can also be heterogeneous,
    i.e., no single configuration performs well on all instances

  $\rightsquigarrow$ combination of algorithm configuration and selection

$\rightsquigarrow$ Hyperparameter optimization is a subproblem of algorithm configuration

Lecture Overview

State-of-the-art Algorithm Configuration

SMAC: Sequential Model-based Algorithm Configuration =

- + Bayesian Optimization (instead of local search)
- + aggressive racing
- + adaptive capping (for optimizing runtime)

SMAC: Overview

Algorithm 1: SMAC

Input : instance set Π, Algorithm $\mathcal{A}$ with configuration space $\mathbf{\Lambda}$, Initial configuration λ_0, performance metric c, Configuration budget b

Output: best incumbent configuration $\hat{\lambda}$

run history H $\leftarrow$ initial design based on λ_0; // `H =` $(\lambda, \pi, c(\pi, \lambda))_i$

while b *remains* **do**

SMAC: Overview

Algorithm 2: SMAC

Input : instance set Π, Algorithm $\mathcal{A}$ with configuration space $\mathbf{\Lambda}$, Initial configuration λ_0, performance metric c, Configuration budget b

Output: best incumbent configuration $\hat{\lambda}$

run history H $\leftarrow$ initial design based on λ_0; // `H =` $(\lambda, \pi, c(\pi, \lambda))_i$

while b *remains* **do**

  $\hat{f} \leftarrow$ train empirical performance model based on run history H;

Algorithm 3: SMAC

Input : instance set Π, Algorithm $\mathcal{A}$ with configuration space $\mathbf{\Lambda}$, Initial configuration λ_0, performance metric c, Configuration budget b

Output: best incumbent configuration $\hat{\lambda}$

run history H $\leftarrow$ initial design based on λ_0; // `H =` $(\lambda, \pi, c(\pi, \lambda))_i$

while b *remains* **do**

  $\hat{f} \leftarrow$ train empirical performance model based on run history H;

  $\mathbf{\Lambda}_{challengers} \leftarrow$ select configurations based on $\hat{f}$;

Algorithm 4: SMAC

Input : instance set Π, Algorithm $\mathcal{A}$ with configuration space $\mathbf{\Lambda}$, Initial configuration λ_0, performance metric c, Configuration budget b

Output: best incumbent configuration $\hat{\lambda}$

run history H $\leftarrow$ initial design based on λ_0; `// H =` $(\lambda, \pi, c(\pi, \lambda))_i$

while b *remains* **do**

  $\hat{f} \leftarrow$ train empirical performance model based on run history H;

  $\mathbf{\Lambda}_{challengers} \leftarrow$ select configurations based on $\hat{f}$;

  $\hat{\lambda}$, H $\leftarrow$ intensify($\mathbf{\Lambda}_{challengers}$, $\hat{\lambda}$); `// racing and capping`

Algorithm 5: SMAC

Input : instance set Π, Algorithm $\mathcal{A}$ with configuration space $\mathbf{\Lambda}$, Initial configuration λ_0, performance metric c, Configuration budget b

Output: best incumbent configuration $\hat{\lambda}$

run history H $\leftarrow$ initial design based on λ_0; `// H =` $(\lambda, \pi, c(\pi, \lambda))_i$

while b *remains* **do**

  $\hat{f} \leftarrow$ train empirical performance model based on run history H;

  $\mathbf{\Lambda}_{challengers} \leftarrow$ select configurations based on $\hat{f}$;

  $\hat{\lambda}$, H $\leftarrow$ intensify($\mathbf{\Lambda}_{challengers}$, $\hat{\lambda}$); `// racing and capping`

return $\hat{\lambda}$

Comparisons on N instances

- Basic(N) uses a pretty basic comparison: $\mathit{better}_N(\lambda', \lambda)$:
  - Compare λ' and λ based on N instances

Comparisons on N instances

- Basic(N) uses a pretty basic comparison: *better*$_N(\lambda', \lambda)$:
  - Compare λ' and λ based on N instances
  - How does this relate to cross-validation?

Comparisons on N instances

- Basic(N) uses a pretty basic comparison: *better*$_N(\lambda', \lambda)$:
  - Compare λ' and λ based on N instances
  - How does this relate to cross-validation?
- Problem: How to set N? Problems of large N? Small N?

Comparisons on N instances

- Basic(N) uses a pretty basic comparison: *better*$_N(\lambda', \lambda)$:
  - Compare λ' and λ based on N instances
  - How does this relate to cross-validation?
- Problem: How to set N? Problems of large N? Small N?
  - Problem of large N: evaluations are slow
  - Problem of small N: overfitting to a small set of instances
  - $\leadsto$ Tradeoff: Choose N of moderate size

Comparisons on N instances

Question: Which N instances should we use?

1. N different instances for each configuration
2. The same set of N instances for the entire run

Comparisons on N instances

Question: Which N instances should we use?

1. N different instances for each configuration
2. The same set of N instances for the entire run

Answer: the same N instances, so that we compare apples with apples

(but: using the same instances can also yield overtuning)

If we sampled different instances for each configuration:

- Some configurations would randomly get easier instances
- Those configurations would look better than they really are

Comparisons on N instances

Question: For randomized algorithms, how should we set the seeds?

1. Sample a new seed for each algorithm run
2. Fix the seeds together with the instances

Comparisons on N instances

Question: For randomized algorithms, how should we set the seeds?

1. Sample a new seed for each algorithm run
2. Fix the seeds together with the instances

Answer: just like for instances, fix them to compare apples to apples

Comparisons on N instances

Question: For randomized algorithms, how should we set the seeds?

1. Sample a new seed for each algorithm run
2. Fix the seeds together with the instances

Answer: just like for instances, fix them to compare apples to apples

In summary, for each run of Basic(N):
pick N (instance, seed) pairs and use them for evaluating each λ.

Comparisons on N instances

Question: For randomized algorithms, how should we set the seeds?

1. Sample a new seed for each algorithm run
2. Fix the seeds together with the instances

Answer: just like for instances, fix them to compare apples to apples

In summary, for each run of Basic(N):
pick N (instance, seed) pairs and use them for evaluating each λ.
(Different Basic(N) runs can use different instances and seeds.)

The concept of overtuning

Very related to overfitting in machine learning

- Performance improves on the training set
- Performance does not improve on the test set, and may even degrade

The concept of overtuning

Very related to overfitting in machine learning

- Performance improves on the training set
- Performance does not improve on the test set, and may even degrade

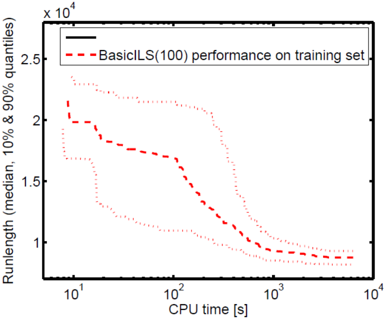
More pronounced for more heterogeneous benchmark sets

- But it even happens for very homogeneous sets
- Indeed, one can even overfit on a single instance, to the seeds used for training

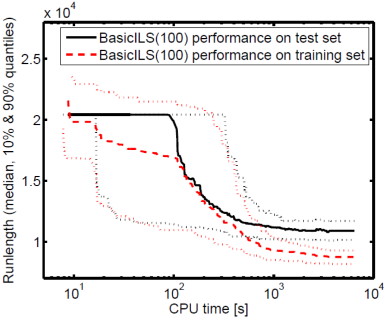
Overtuning Visualized

- Example: minimizing SLS solver runlengths for a single SAT instance
- Training cost, e.g., with N=100:
  average runlengths across 100 runs with different seeds
- Test cost of $\hat{\lambda}$ here based on 1000 new seeds

- Example: minimizing SLS solver runlengths for a single SAT instance
- Training cost, e.g., with N=100:
  average runlengths across 100 runs with different seeds
- Test cost of $\hat{\lambda}$ here based on 1000 new seeds



- Example: minimizing SLS solver runlengths for a single SAT instance
- Training cost, e.g., with N=100:
  average runlengths across 100 runs with different seeds
- Test cost of $\hat{\lambda}$ here based on 1000 new seeds

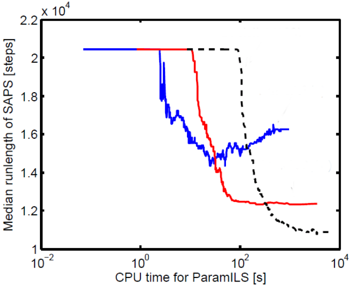


Basic(N) Test Results with Various N

- Example: minimizing SLS solver runlengths for a single SAT instance
- Training cost, e.g., with N=?:
  average runlengths across N runs with different seeds
- Test cost of $\hat{\lambda}$ here based on 1000 new seeds

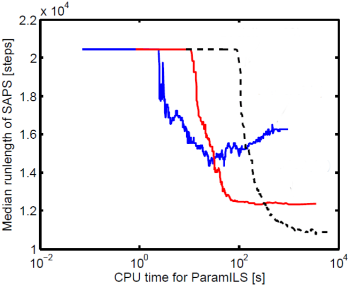
Basic(N) Test Results with Various N

- Example: minimizing SLS solver runlengths for a single SAT instance
- Training cost, e.g., with N=?:
  average runlengths across N runs with different seeds
- Test cost of $\hat{\lambda}$ here based on 1000 new seeds



Basic(N) Test Results with Various N

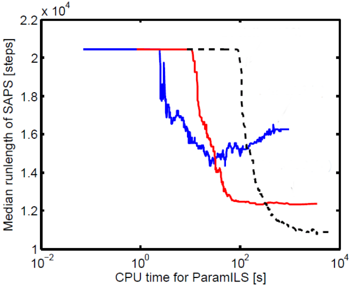
- Example: minimizing SLS solver runlengths for a single SAT instance
- Training cost, e.g., with N=?:
  average runlengths across N runs with different seeds
- Test cost of $\hat{\lambda}$ here based on 1000 new seeds



Which of these results corresponds to $N = 1$, $N = 10$, and $N = 100$?

Basic(N) Test Results with Various N

- Example: minimizing SLS solver runlengths for a single SAT instance
- Training cost, e.g., with N=?:
  average runlengths across N runs with different seeds
- Test cost of $\hat{\lambda}$ here based on 1000 new seeds

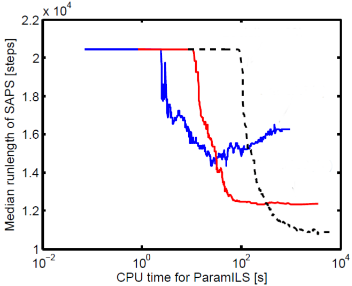


Which of these results corresponds to $N = 1$, $N = 10$, and $N = 100$?

1. N=1: blue, N=10: red, N=100 dashed black
2. N=1: dashed black, N=10: red, N=100 blue

Basic(N) Test Results with Various N

- Example: minimizing SLS solver runlengths for a single SAT instance
- Training cost, e.g., with N=?:
  average runlengths across N runs with different seeds
- Test cost of $\hat{\lambda}$ here based on 1000 new seeds



Which of these results corresponds to $N = 1$, $N = 10$, and $N = 100$?

1. N=1: blue, N=10: red, N=100 dashed black
2. N=1: dashed black, N=10: red, N=100 blue

Correct Answer: 1

Aggressive Racing (inspired by FocusedILS)

Intuition: get the best of both worlds

- Perform more runs for good configurations
  - to avoid overtuning
- Quickly reject poor configurations
  - to make progress more quickly

Aggressive Racing (inspired by FocusedILS)

Intuition: get the best of both worlds

- Perform more runs for good configurations
  - to avoid overtuning
- Quickly reject poor configurations
  - to make progress more quickly

Definition: $N(\lambda)$ and $c_N(\lambda)$

$N(\lambda)$ denotes the number of runs executed for λ so far.
$\hat{c}_N(\lambda)$ denotes the cost estimate of λ based on N runs.

Aggressive Racing (inspired by FocusedILS)

Intuition: get the best of both worlds

- Perform more runs for good configurations
  - to avoid overtuning
- Quickly reject poor configurations
  - to make progress more quickly

Definition: $N(\lambda)$ and $c_N(\lambda)$

$N(\lambda)$ denotes the number of runs executed for λ so far.
$\hat{c}_N(\lambda)$ denotes the cost estimate of λ based on N runs.

In the beginning: $N(\lambda) = 0$ for every configuration λ

Aggressive Racing

Definition: domination

λ_1 dominates λ_2 if

- $N(\lambda_1) \geq N(\lambda_2)$ and
- $\hat{c}_{N(\lambda_2)}(\lambda_1) \leq \hat{c}_{N(\lambda_2)}(\lambda_2)$.

I.e.: we have at least as many runs for λ_1 and its cost is at least as low.

Aggressive Racing

Definition: domination

λ_1 dominates λ_2 if

- $N(\lambda_1) \geq N(\lambda_2)$ and
- $\hat{c}_{N(\lambda_2)}(\lambda_1) \leq \hat{c}_{N(\lambda_2)}(\lambda_2)$.

I.e.: we have at least as many runs for λ_1 and its cost is at least as low.

better(λ', λ^*) in a nutshell

- λ^* is the current configuration to beat (incumbent)

Aggressive Racing

Definition: domination

λ_1 dominates λ_2 if

- $N(\lambda_1) \geq N(\lambda_2)$ and
- $\hat{c}_{N(\lambda_2)}(\lambda_1) \leq \hat{c}_{N(\lambda_2)}(\lambda_2)$.

I.e.: we have at least as many runs for λ_1 and its cost is at least as low.

better(λ', λ^*) in a nutshell

- λ^* is the current configuration to beat (incumbent)
- Perform runs of λ' until either
  - λ^* dominates λ' $\rightsquigarrow$ reject λ', or
  - λ' dominates λ^* $\rightsquigarrow$ change current configuration $(\lambda^* \leftarrow \lambda')$

Aggressive Racing

Definition: domination

λ_1 dominates λ_2 if

- $N(\lambda_1) \geq N(\lambda_2)$ and
- $\hat{c}_{N(\lambda_2)}(\lambda_1) \leq \hat{c}_{N(\lambda_2)}(\lambda_2)$.

I.e.: we have at least as many runs for λ_1 and its cost is at least as low.

better(λ', λ^*) in a nutshell

- λ^* is the current configuration to beat (incumbent)
- Perform runs of λ' until either
  - λ^* dominates λ' $\rightsquigarrow$ reject λ', or
  - λ' dominates λ^* $\rightsquigarrow$ change current configuration $(\lambda^* \leftarrow \lambda')$
- Over time: perform extra runs of λ^* to gain more confidence in it

Toy Example

- Let λ^* be the incumbent (evaluated on π_1, π_2, π_3)
- We'll look at challengers λ' and λ''

	π_1	π_2	π_3
λ^*	3	2	10

Toy Example

- Let λ^* be the incumbent (evaluated on π_1, π_2, π_3)
- We'll look at challengers λ' and λ''

	π_1	π_2	π_3
λ^*	3	2	10
λ'			

Toy Example

- Let λ^* be the incumbent (evaluated on π_1, π_2, π_3)
- We'll look at challengers λ' and λ''

	π_1	π_2	π_3
λ^*	3	2	10
λ'	2		

Toy Example

- Let λ^* be the incumbent (evaluated on π_1, π_2, π_3)
- We'll look at challengers λ' and λ''

	π_1	π_2	π_3
λ^*	3	2	10
λ'	2	10	

Toy Example

- Let λ^* be the incumbent (evaluated on π_1, π_2, π_3)
- We'll look at challengers λ' and λ''

	π_1	π_2	π_3
λ^*	3	2	10
λ'	2	10	
	$\rightarrow$ reject, since $\hat{c}_2(\lambda') = 6 > \hat{c}_2(\lambda^*) = 2.5$		

Toy Example

- Let λ^* be the incumbent (evaluated on π_1, π_2, π_3)
- We'll look at challengers λ' and λ''

	π_1	π_2	π_3
λ^*	3	2	10
λ'	2	10	
		$\rightarrow$ reject, since $\hat{c}_2(\lambda') = 6 > \hat{c}_2(\lambda^*) = 2.5$	
λ''	3		

Toy Example

- Let λ^* be the incumbent (evaluated on π_1, π_2, π_3)
- We'll look at challengers λ' and λ''

	π_1	π_2	π_3
λ^*	3	2	10
λ'	2	10	
		$\rightarrow$ reject, since $\hat{c}_2(\lambda') = 6 > \hat{c}_2(\lambda^*) = 2.5$	
λ''	3	1	

Toy Example

- Let λ^* be the incumbent (evaluated on π_1, π_2, π_3)
- We'll look at challengers λ' and λ''

	π_1	π_2	π_3
λ^*	3	2	10
λ'	2	10	
		$\rightarrow$ reject, since $\hat{c}_2(\lambda') = 6 > \hat{c}_2(\lambda^*) = 2.5$	
λ''	3	1	5

Toy Example

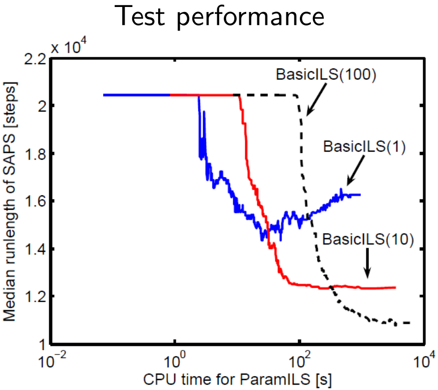
- Let λ^* be the incumbent (evaluated on π_1, π_2, π_3)
- We'll look at challengers λ' and λ''

	π_1	π_2	π_3
λ^*	3	2	10
λ'	2	10	
	$\rightarrow$ reject, since $\hat{c}_2(\lambda') = 6 > \hat{c}_2(\lambda^*) = 2.5$		
λ''	3	1	5

- new incumbent: $\lambda^* \leftarrow \lambda''$
- Perform an additional run for new λ^* to increase confidence over time

Racing achieves the best of both worlds

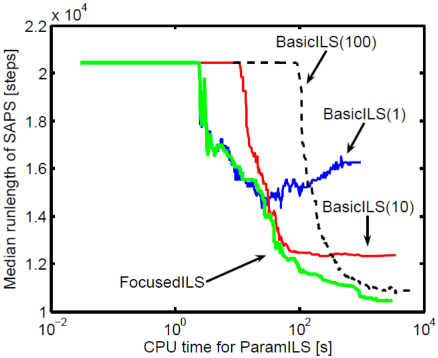
Aggressive racing (aka FocusedILS): Fast progress and no overtuning



Racing achieves the best of both worlds

Aggressive racing (aka FocusedILS): Fast progress and no overtuning

Test performance



Adaptive capping

- Assumptions
  - optimization of runtime
  - each configuration run has a time limit (e.g., 300 sec)

Adaptive capping

- Assumptions
  - optimization of runtime
  - each configuration run has a time limit (e.g., 300 sec)
- E.g., λ^* needed 1 sec to solve π_1
  - Do we need to run λ' for 300 sec?
  - Terminate evaluation of λ' once guaranteed to be worse than λ^*

Adaptive capping

- Assumptions
  - optimization of runtime
  - each configuration run has a time limit (e.g., 300 sec)
- E.g., λ^* needed 1 sec to solve π_1
  - Do we need to run λ' for 300 sec?
  - Terminate evaluation of λ' once guaranteed to be worse than λ^*

$\leadsto$ To compare against λ^* based on N runs,
we can terminate evaluation of λ' after time $\sum_{i=1}^{N} c(\lambda^*, \pi_i)$

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances (using $\hat{c}_3$)

	π_1	π_2
λ^*	4	2

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances (using $\hat{c}_3$)

	π_1	π_2
λ^*	4	2
Without adaptive capping		
λ'		

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances (using $\hat{c}_3$)

	π_1	π_2
λ^*	4	2
Without adaptive capping		
λ'	3	

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances (using $\hat{c}_3$)

	π_1	π_2
λ^*	4	2
Without adaptive capping		
λ'	3	300

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances (using $\hat{c}_3$)

	π_1	π_2
λ^*	4	2
Without adaptive capping		
λ'	3	300
		$\rightarrow$ reject λ' (cost: 303)

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances (using $\hat{c}_3$)

	π_1	π_2
λ^*	4	2
Without adaptive capping		
λ'	3	300
		$\rightarrow$ reject λ' (cost: 303)
With adaptive capping		
λ'		

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances (using $\hat{c}_3$)

	π_1	π_2
λ^*	4	2
Without adaptive capping		
λ'	3	300
		$\rightarrow$ reject λ' (cost: 303)
With adaptive capping		
λ'	3	

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances (using $\hat{c}_3$)

	π_1	π_2
λ^*	4	2
Without adaptive capping		
λ'	3	300
		$\rightarrow$ reject λ' (cost: 303)
With adaptive capping		
λ'	3	300

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances (using $\hat{c}_3$)

	π_1	π_2
λ^*	4	2
Without adaptive capping		
λ'	3	300
		$\rightarrow$ reject λ' (cost: 303)
With adaptive capping		
λ'	3	300
	$\rightarrow$ cut off after $\kappa = 4$ seconds, reject λ' (cost: 7)	

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances (using $\hat{c}_3$)

	π_1	π_2
λ^*	4	2
Without adaptive capping		
λ'	3	300
		$\rightarrow$ reject λ' (cost: 303)
With adaptive capping		
λ'	3	300
	$\rightarrow$ cut off after $\kappa = 4$ seconds, reject λ' (cost: 7)	

Note: To combine adaptive capping with BO, we need to impute the censored observations caused by adaptive capping.

SMAC: Overview

Algorithm 6: SMAC

Input : instance set Π, Algorithm $\mathcal{A}$ with configuration space $\mathbf{\Lambda}$, Initial configuration λ_0, performance metric c, Configuration budget b

Output: best incumbent configuration $\hat{\lambda}$

run history H $\leftarrow$ initial design based on λ_0; `// H =` $(\lambda, \pi, c(\pi, \lambda))_i$

while b *remains* **do**

- $\hat{f} \leftarrow$ train empirical performance model based on run history H;
- $\mathbf{\Lambda}_{challengers} \leftarrow$ select configurations based on $\hat{f}$; `// BO with EI`
- $\hat{\lambda}$, H $\leftarrow$ intensify($\mathbf{\Lambda}_{challengers}$, $\hat{\lambda}$); `// racing and capping`

return $\hat{\lambda}$

Lecture Overview

Dynamic Heuristics

- Many heuristics in algorithms are dynamic and adaptive
  1. the algorithm's behavior changes over time
  2. the algorithm's behavior changes based on internal statistics

- these heuristics might control other parameters of the algorithms

Dynamic Heuristics

- Many heuristics in algorithms are dynamic and adaptive
  1. the algorithm's behavior changes over time
  2. the algorithm's behavior changes based on internal statistics
- these heuristics might control other parameters of the algorithms
- example: learning rate schedules for training DNNs
  1. exponential decaying learning rate: based on number of iterations, learning rate decreases

Dynamic Heuristics

- Many heuristics in algorithms are dynamic and adaptive
  1. the algorithm's behavior changes over time
  2. the algorithm's behavior changes based on internal statistics
- these heuristics might control other parameters of the algorithms
- example: learning rate schedules for training DNNs
  1. exponential decaying learning rate: based on number of iterations, learning rate decreases
  2. Reduce learning rate on plateaus: if the learning stagnates for some time, the learning rate is decreased by a factor

Dynamic Heuristics

- Many heuristics in algorithms are dynamic and adaptive
  1. the algorithm's behavior changes over time
  2. the algorithm's behavior changes based on internal statistics
- these heuristics might control other parameters of the algorithms
- example: learning rate schedules for training DNNs
  1. exponential decaying learning rate: based on number of iterations, learning rate decreases
  2. Reduce learning rate on plateaus: if the learning stagnates for some time, the learning rate is decreased by a factor
- What examples for dynamic heuristics can you think of?

Dynamic Heuristics

- Many heuristics in algorithms are dynamic and adaptive
  1. the algorithm's behavior changes over time
  2. the algorithm's behavior changes based on internal statistics
- these heuristics might control other parameters of the algorithms
- example: learning rate schedules for training DNNs
  1. exponential decaying learning rate: based on number of iterations, learning rate decreases
  2. Reduce learning rate on plateaus: if the learning stagnates for some time, the learning rate is decreased by a factor
- What examples for dynamic heuristics can you think of?
- other examples: restart probability of search, mutation rate of evolutionary algorithms, . . .

Parametrization of Learning Rate Schedules

- How would we parameterize learning rate schedules?
  1. exponential decaying learning rate:
     - initial learning rate
     - minimal learning rate
     - multiplicative factor

Parametrization of Learning Rate Schedules

- How would we parameterize learning rate schedules?
  1. exponential decaying learning rate:
     - initial learning rate
     - minimal learning rate
     - multiplicative factor
  2. Reduce learning rate on plateaus:
     - patience (in number of epochs)
     - patience threshold
     - decreasing factor
     - cool-down break (in number of epochs)

Parametrization of Learning Rate Schedules

- How would we parameterize learning rate schedules?
  1. exponential decaying learning rate:
     - initial learning rate
     - minimal learning rate
     - multiplicative factor
  2. Reduce learning rate on plateaus:
     - patience (in number of epochs)
     - patience threshold
     - decreasing factor
     - cool-down break (in number of epochs)

$\rightsquigarrow$ Many parameters only to control a single parameter (learning rate)

Parametrization of Learning Rate Schedules

- How would we parameterize learning rate schedules?
  1. exponential decaying learning rate:
     - initial learning rate
     - minimal learning rate
     - multiplicative factor
  2. Reduce learning rate on plateaus:
     - patience (in number of epochs)
     - patience threshold
     - decreasing factor
     - cool-down break (in number of epochs)

$\rightsquigarrow$ Many parameters only to control a single parameter (learning rate)

- Still not guaranteed that optimal setting of learning rate schedules will lead to optimal learning rate behavior
  - Learning rate schedules are only heuristics

Algorithm Control [Biedenkapp et al. 2019]

- Goal: control a (set of) hyperparameter(s) during the run
- Problem can be defined as an MDP $\mathcal{M} := (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$

State Space $\mathcal{S}$ At each time step t, internal state s_t of the target algorihtm being controlled.

Action space $\mathcal{A}$ Given a state s_t, the controller has to decide how to change the value $v \in \mathcal{A}_h$ of a hyperparameter h.

Transition Function $\mathcal{T}$ dynamics of the algorithm at hand transitioning from s_t to s_{t+1} by applying action a_t with probability $t(s_t, a_t, s_{t+1})$

Reward $\mathcal{R}$ Either sparse reward at the end of the algorithm run or intermediate run quality estimate

Algorithm Control [Biedenkapp et al. 2019]

- Goal: control a (set of) hyperparameter(s) during the run
- Problem can be defined as an MDP $\mathcal{M} \coloneqq (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$

State Space $\mathcal{S}$ At each time step t, internal state s_t of the target algorihtm being controlled.

Action space $\mathcal{A}$ Given a state s_t, the controller has to decide how to change the value $v \in \mathcal{A}_h$ of a hyperparameter h.

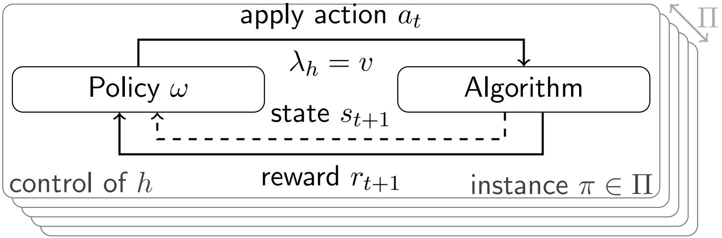
Transition Function $\mathcal{T}$ dynamics of the algorithm at hand transitioning from s_t to s_{t+1} by applying action a_t with probability $t(s_t, a_t, s_{t+1})$

Reward $\mathcal{R}$ Either sparse reward at the end of the algorithm run or intermediate run quality estimate

$$\omega^*(s) \in \arg\max_{a \in \mathcal{A}} \mathcal{R}(s, a) + \mathcal{Q}_{\omega^*}(s, a)$$

$$\mathcal{Q}_\omega(s, a) = \mathbb{E}_\omega \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a \right]$$

Algorithm Control across Instances [Biedenkapp et al. 2019]



Following the same arguments as for algorithm configuration, we want to learn a robust policy across instances $\pi \in \Pi$

Algorithm Control across Instances [Biedenkapp et al. 2019]

- instances can be modeled as context of Markov decision process ($\rightsquigarrow$ contextual MDP)

Algorithm Control across Instances [Biedenkapp et al. 2019]

- instances can be modeled as context of Markov decision process ($\rightsquigarrow$ contextual MDP)
- homogeneous instances
  - instances are similar to each other and a good policy would perform well on all instances
  - instances provides some noise in the policy optimization

  $\rightsquigarrow$ more robust policies which generalize better to new instances

Algorithm Control across Instances [Biedenkapp et al. 2019]

- instances can be modeled as context of Markov decision process ($\rightsquigarrow$ contextual MDP)
- homogeneous instances
  - instances are similar to each other and a good policy would perform well on all instances
  - instances provides some noise in the policy optimization

  $\rightsquigarrow$ more robust policies which generalize better to new instances
- heterogeneous instances
  - instances have different characteristics s.t. the policy has to be adapted to the instance at hand

  $\rightsquigarrow$ we might have to characterize the instances by using instance features (aka meta features)

Lecture Overview

Performance Predictions

- Observation: We collect a lot of data of the performance of an algorithm given its hyperparameter configuration and an instance

Performance Predictions

- Observation: We collect a lot of data of the performance of an algorithm given its hyperparameter configuration and an instance
- Question: Can we accurately predict the performance on new configurations and instances?

Performance Predictions

- Observation: We collect a lot of data of the performance of an algorithm given its hyperparameter configuration and an instance
- Question: Can we accurately predict the performance on new configurations and instances?
- Idea I (RF): For complex spaces (with continuous and categorical hyperparameter), we can use random forests [Hutter et al. '14]

Performance Predictions

- Observation: We collect a lot of data of the performance of an algorithm given its hyperparameter configuration and an instance
- Question: Can we accurately predict the performance on new configurations and instances?
- Idea I (RF): For complex spaces (with continuous and categorical hyperparameter), we can use random forests [Hutter et al. '14]
- Idea II (DNN): If we have randomized algorithms and we are interested in the algorithm's performance distribution, DNNs perform better [Eggensperger et al. '18]

Hyperparameter Importance

- Observation: Often only a few hyperparameters are important to improve the performance of an algorithm

Hyperparameter Importance

- Observation: Often only a few hyperparameters are important to improve the performance of an algorithm
- Question: How can we identify these important hyperparameters?

Hyperparameter Importance

- Observation: Often only a few hyperparameters are important to improve the performance of an algorithm
- Question: How can we identify these important hyperparameters?
- Idea I (Local Importance) If we have an optimized configuration, how much would the performance change if we only change one hyperparameter of it.

Hyperparameter Importance

- Observation: Often only a few hyperparameters are important to improve the performance of an algorithm
- Question: How can we identify these important hyperparameters?
- Idea I (Local Importance) If we have an optimized configuration, how much would the performance change if we only change one hyperparameter of it.
- Idea II (fANOVA): How much of the performance variance is explained by single hyperparameter marginalized across all other settings

Hyperparameter Importance

- Observation: Often only a few hyperparameters are important to improve the performance of an algorithm
- Question: How can we identify these important hyperparameters?
- Idea I (Local Importance) If we have an optimized configuration, how much would the performance change if we only change one hyperparameter of it.
- Idea II (fANOVA): How much of the performance variance is explained by single hyperparameter marginalized across all other settings
- Idea II (Ablation): If we compare two configurations (e.g., default and incumbent), we flip with a greedy strategy the hyperparameters step by step such that we can order them

Hyperparameter Importance

- Observation: Often only a few hyperparameters are important to improve the performance of an algorithm
- Question: How can we identify these important hyperparameters?
- Idea I (Local Importance) If we have an optimized configuration, how much would the performance change if we only change one hyperparameter of it.
- Idea II (fANOVA): How much of the performance variance is explained by single hyperparameter marginalized across all other settings
- Idea II (Ablation): If we compare two configurations (e.g., default and incumbent), we flip with a greedy strategy the hyperparameters step by step such that we can order them

$\rightsquigarrow$ CAVE for analyzing AutoML experiments:
`https://github.com/automl/CAVE`

Benchmark Libraries

- Observation: Results in papers are often not comparable because authors use different benchmarks

Benchmark Libraries

- Observation: Results in papers are often not comparable because authors use different benchmarks
- Question: Can we design standardized benchmarks libraries?

Benchmark Libraries

- Observation: Results in papers are often not comparable because authors use different benchmarks
- Question: Can we design standardized benchmarks libraries?
- Benchmark I (HPOlib): Benchmark library with different HPO benchmarks

Benchmark Libraries

- Observation: Results in papers are often not comparable because authors use different benchmarks
- Question: Can we design standardized benchmarks libraries?
- Benchmark I (HPOlib): Benchmark library with different HPO benchmarks
- Benchmark II (NASBench 101): Tabular benchmark for neural architecture search

Benchmark Libraries

- Observation: Results in papers are often not comparable because authors use different benchmarks
- Question: Can we design standardized benchmarks libraries?
- Benchmark I (HPOlib): Benchmark library with different HPO benchmarks
- Benchmark II (NASBench 101): Tabular benchmark for neural architecture search
- Benchmark III (AClib) Benchmarks and cheap-to-run surrogate benchmarks for algorithm configuration

Learning Goals

After this lecture, you are able to . . .

- define the algorithm configuration problem
- discuss differences between HPO and algorithm configuration
- explain the components of SMAC to combine Bayesian Optimization across instances
- define and give examples for the algorithm control problem
- list some further topics related AutoML and algorithm configuration

Literature [These are links]

- SMAC [Sequential Model-Based Optimization for General Algorithm Configuration]
- [Pitfalls and Best Practices in Algorithm Configuration]
- [Towards White-box Benchmarks for Algorithm Control]
- [CAVE: Configuration Assessment, Visualization and Evaluation]