Meta-Learning as a Markov Decision Process

Lisheng Sun-Hosoya

To cite this version:

Lisheng Sun-Hosoya. Meta-Learning as a Markov Decision Process. Machine Learning [cs.LG]. Université Paris Saclay (COmUE), 2019. English. NNT : 2019SACLS588 . tel-02422144v2

université
PARIS-SACLAY

UNIVERSITÉ PARIS SUD
Comprendre le monde, construire l'avenir

NNT : 2019SACLS588

Thèse de doctorat

Meta-Learning as a Markov Decision Process

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université Paris-Sud

Ecole doctorale n°580 Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat: Mathématiques et Informatique

Thèse présentée et soutenue à Orsay, le 19 Décembre 2019, par

LISHENG SUN-HOSOYA

Composition du Jury :

M. Nicolas Thiéry Professeur, Laboratoire de Recherche en Informatique, Université Paris-Sud, France	Président
Mme Cécile Capponi Maîtresse de Conférences HDR, Université d'Aix-Marseille, France	Rapporteur
M. Daniel Silver Professeur, Acadia University, Canada	Rapporteur
M. Hugo Jair Escalante Professeur, Instituto Nacional de Astrofísica, Óptica y Electrónica, Mexique	Examinateur
M. Joaquin Vanschoren Professeur, Eindhoven University of Technology, Pays-Bas	Examinateur
Mme Isabelle Guyon Professeure, UPSud/INRIA Université Paris-Saclay, France	Directrice de thèse
Mme Michèle Sebag Dir. Recherche, CNRS Université Paris-Saclay, France	Co-encadrante de thèse

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Lisheng Sun-Hosoya
January 2020

Acknowledgements

This work was performed at the Laroratoire de Recherche en Informatique at Université Paris Sud (Paris-Saclay), under the direction of Isabelle Guyon, and Michèle Sebag and with the kind advice of Amy Zhang (McGill University and Facebook AI Research). Many people helped me throughout my thesis work. I am particularly grateful to Nathan Grinsztajn who worked very hard as a master student intern under my direction.

Special thanks: The work presented in Chapter 6 was realized in collaboration with Nathan Grinsztajn from Ecole polytechnique, Lozere, France; and Amy Zhang from Facebook AI Research, Montreal, Canada.

Summary

Machine Learning (ML) has enjoyed huge successes in recent years and an ever-growing number of real-world applications rely on it. However, designing promising algorithms for a specific problem still requires huge human effort. Automated Machine Learning (**AutoML**) aims at taking the human out of the loop and develop machines that generate / recommend good algorithms for a given ML task. AutoML is usually treated as a algorithm / hyper-parameter selection problem, existing approaches include Bayesian optimization (e.g. [52, 109, 35, 106]), evolutionary algorithms (e.g. [91]) as well as reinforcement learning (e.g. [123, 5]). Among them, auto-sklearn [35] which incorporates meta-learning techniques in their search initialization[19], ranks consistently well in AutoML challenges [43, 102, 57]. This observation oriented my research to the **Meta-Learning** domain, leading to my recent paper [101] where active learning and collaborative filtering [99] are used to assign as quickly as possible a good algorithm to a new dataset, based on a meta-learning performance matrix *S*, i.e. a matrix of scores of algorithms on given datasets or tasks. This direction led me to develop a novel framework based on **Markov Decision Processes** (MDP) and reinforcement learning (RL).

After a general introduction (Chapter 1), my thesis work starts with an in-depth analysis of the results of the AutoML challenge (Chapter 2), for which I present systematic experiments, published as a book chapter [57]. The methods that particularly drew my attention in this analysis include wrapper methods from the Bayesian Optimization family, built around the toolkit scikit-learn [86], such as auto-sklearn and Freeze-Thaw [106, 73]. Such principled hyper-parameter selection methods were interesting to compare with (1) heuristic search (e.g. [107]) and (2) methods that avoided model/hyper-parameter selection altogether or reduced its need to the greatest possible extent, such as Selective Naive Bayes (SNB [18] and gradient boosting [112]. This analysis oriented my work towards meta-learning, leading me first to propose a formulation of AutoML as a recommendation problem, following the seminal work of [6, 7, 75], and ultimately to formulate a novel conceptualisation of the problem as a MDP (Chapter 3).

In the MDP setting, the problem is brought back to filling up, as quickly and efficiently as possible, a meta-learning matrix *S*, in which lines correspond to ML

tasks and columns to ML algorithms. A matrix element $S(i, j)$ is the performance of algorithm j applied to task i. Searching efficiently for the best values in S allows us to identify quickly algorithms best suited to given tasks. In the proposed MDP/RL setting, the goal of AutoML is to develop an "agent" to search for the best algorithm for each task by placing sequential "queries" *i.e.* training and testing an algorithm on a given task to gain the knowledge of a new element $S(i, j)$. The system's **state** consists of the values of $S(i, j)$ that have been revealed through this query process, each **action** consisting in placing a new query. In terms of **reward**, each query costs the agent in computational time, but this negative reward is compensated by a positive reward for eventually discovering a better performing algorithm.

In Chapter 4 the classical hyper-parameter optimization framework (HyperOpt[13]) is first reviewed, and in particular the so-called CASH problem (Combined Algorithm Selection and Hyperparameter optimization) [110]. No meta-learning is involved at this stage; this corresponds to filling one line of matrix S, independently of others. In Chapter 5 a first meta-learning approach is introduced along the lines of our paper **ActivMetaL** [101] that combines active learning and collaborative filtering techniques to predict the missing values in S. This allows us finding quicker than with other baseline methods the best algorithm in each line of S [75, 101]. In this case, we are using some lines of the matrix (performances of algorithms on known datasets) as "training data" to help fill out new lines (predict performances on new datasets), but the policy implemented by our "agent" is, in some sense, "hard-coded". Our latest research applies RL to the MDP problem we defined to learn an efficient policy to explore S. We call this approach **REVEAL** and propose an analogy with a series of toy games to help visualize agents' strategies to reveal information progressively, e.g. masked areas of images to be classified [122], or ship positions in a battleship game. This line of research is developed in Chapter 6.

The main results of my PhD project are:

- **HP / model selection**: I have explored the Freeze-Thaw method and optimized the algorithm to enter the first AutoML challenge, achieving 3rd place in the final round (Chapter 3 in [57]).

- **ActivMetaL**: I have designed a new algorithm for active meta-learning (ActivMetaL) and compared it with other baseline methods on real-world and artifical data. This study demonstrated that ActiveMetaL is generally able to discover the best algorithm faster than baseline methods.

- **REVEAL**: I developed a new conceptualization of meta-learning as a Markov Decision Process and put it into the more general framework of REVEAL games.

> With a master student intern, I developed agents that learn (with reinforcement learning) to predict the next best algorithm to be tried. To develop this agent, we used surrogate toy tasks of REVEAL games. We then applied our methods to AutoML problems.

The work presented in my thesis is empirical in nature. Several real world meta-datasets were used in this research, each of which corresponds to one score matrix S, including the AutoML challenge result matrix, which I compiled by applying all methods of the challenge top ranking participants and other baseline methods to all datasets of various challenge rounds, OpenML datasets, Statlog Dataset in UCI database, and Cause-Effect pairs challenge datasets. Artificial and semi-artificial meta-datasets are also used in my work: one was constructed from a matrix factorization in order to guarantee a certain spectrum of singular values for S; another was constructed by applying systematically various regression algorithms to simple univariate regression problems from the cause-effect pair challenge dataset [1]. The results indicate that RL is a viable approach to this problem, although much work remains to be done to optimize algorithms to make them scale to larger meta-learning problems.

[1] See book in preparation https://sites.google.com/a/chalearn.org/causality/experimental-design.

Résumé en français

L'apprentissage automatique (ML) a connu d'énormes succès ces dernières années et repose sur un nombre toujours croissant d'applications réelles. Cependant, la conception d'algorithmes prometteurs pour un problème spécifique nécessite toujours un effort humain considérable. L'apprentissage automatisé (**AutoML**) a pour objectif de sortir l'homme de la boucle. AutoML est généralement traité comme un problème de sélection d'algorithme / hyper-paramètre. Les approches existantes incluent l'optimisation Bayésienne, les algorithmes évolutionnistes et l'apprentissage par renforcement. Parmi eux, auto-sklearn, qui intègre des techniques de meta-learning à l'initialisation de la recherche, occupe toujours une place de choix dans les challenges AutoML. Cette observation a orienté mes recherches vers le domaine du **Meta-learning**. Cette orientation m'a amené à développer un nouveau cadre basé sur les **Processus de Décision Markovien** (MDP) et l'apprentissage par renforcement (RL).

Après une introduction générale (Chapter 1), mon travail de thèse commence par une analyse approfondie des résultats du challenge AutoML (Chapter 2), pour lequel je présente des expériences systématiques, publiées sous forme de chapitre de livre [57]. Les méthodes qui ont particulièrement attiré mon attention dans cette analyse incluent les méthodes d'encapsulation de la famille Bayesian Optimization, construites autour de la boîte à outils scikit-learn [86], telles que l'auto-apprentissage et Freeze-Thaw [106, 73]. De telles méthodes de sélection d'hyper-paramètres fondées sur des principes étaient intéressantes à comparer avec (1) la recherche heuristique (par exemple [107]) et (2) les méthodes qui évitaient complètement la sélection de modèle / hyper-paramètre ou réduisaient le plus possible son besoin, comme Selective Naive Bayes (SNB [18] et gradient boosting [112]. Cette analyse a orienté mon travail vers le méta-apprentissage, m'amenant d'abord à proposer une formulation d'AutoML comme problème de recommandation, à la suite du travail séminal de [6, 7, 75], et pour finalement formuler une nouvelle conceptualisation du problème en tant que MDP (Chapter 3).

Dans le cadre MDP, le problème est ramené au remplissage, aussi rapidement et efficacement que possible, d'une matrice de meta-learning S, dans laquelle les lignes correspondent aux tâches ML et les colonnes aux algorithmes ML. Un élément matriciel $S(i,j)$ est la performance de l'algorithme j appliqué à la tâche i. La recherche efficace des meilleures valeurs en S nous permet d'identifier rapidement les algorithmes les mieux adaptés à des tâches données. Dans le cadre MDP / RL proposé, l'objectif d'AutoML est de développer un «agent» pour rechercher le meilleur algorithme pour chaque tâche en plaçant des «requêtes» séquentielles, c'est-à-dire en entrainant et en testant un algorithme sur une tâche donnée pour acquérir la connaissance d'un nouvel élément $S(i,j)$. L'**état** du système est constitué des valeurs de $S(i,j)$ qui ont été révélées par ce processus de requête, chaque **action** consistant à placer une nouvelle requête. En termes de **récompense**, chaque requête coûte à l'agent un temps de calcul, mais cette récompense négative est compensée par une récompense positive pour éventuellement découvrir un algorithme plus performant.

Dans le chapitre 4, le cadre classique d'optimisation d'hyper-paramètres (HyperOpt [13]) est d'abord examiné, et en particulier le problème dit CASH (Combined Algorithm Selection and Hyperparameter optimization) [110]. Aucun meta-learning n'est impliqué à ce stade; cela correspond au remplissage d'une ligne de matrice S, indépendamment des autres. Dans le chapitre 5, une première approche de meta-learning est introduite le long des lignes de notre article **ActivMetaL** [101] qui combine l'apprentissage actif et les techniques de filtrage collaboratif pour prédire les valeurs manquantes en S. Cela nous permet de trouver plus rapidement qu'avec d'autres méthodes de base le meilleur algorithme dans chaque ligne de S [75, 101]. Dans ce cas, nous utilisons certaines lignes de la matrice (performances d'algorithmes sur des ensembles de données connus) comme des «données d'entraînement» pour aider à remplir de nouvelles lignes (prédire les performances sur de nouveaux ensembles de données), mais la politique mise en œuvre par notre «agent» est, dans un certain sens, «codée en dur». Nos dernières recherches appliquent RL au problème MDP que nous avons défini pour apprendre une politique efficace pour explorer S. Nous appelons cette approche **REVEAL** et proposons une analogie avec une série de jeux pour aider à visualiser les stratégies des agents pour révéler progressivement des informations, par ex. zones masquées d'images à classer [122] ou positions de navire dans un jeu de cuirassé. Cette ligne de recherche est développée au chapitre 6.

Les principaux résultats de mon projet de thèse sont:

- **Sélection HP / modèle**: j'ai exploré la méthode Freeze-Thaw et optimisé l'algorithme pour entrer dans le premier challenge AutoML, obtenant la 3ème place du tour final (chapitre 3).

- **ActivMetaL**: j'ai conçu un nouvel algorithme pour le meta-learning actif (ActivMetaL) et l'ai comparé à d'autres méthodes de base sur des données réelles et artificielles. Cette étude a démontré qu'ActiveMetaL est généralement capable de découvrir le meilleur algorithme plus rapidement que les méthodes de base.

- **REVEAL**: j'ai développé une nouvelle conceptualisation du meta-learning en tant que processus de décision Markovien et je l'ai intégrée dans le cadre plus général des jeux REVEAL. Avec un stagiaire en master, j'ai développé des agents qui apprennent (avec l'apprentissage par renforcement) à prédire le meilleur algorithme à essayer.

Le travail présenté dans ma thèse est de nature empirique. Plusieurs méta-données du monde réel ont été utilisées dans cette recherche. Des méta-données artificielles et semi-artificielles sont également utilisées dans mon travail. Les résultats indiquent que RL est une approche viable de ce problème, bien qu'il reste encore beaucoup à faire pour optimiser les algorithmes et les faire passer à l'échelle aux problèmes de méta-apprentissage plus vastes.

Table of contents

List of figures

List of tables

Chapter 1

Background and Motivation

1.1 Introduction

Until about ten years ago, machine learning (ML) was a discipline little known to the public. Large internet corporations accumulating massive amounts of data such as Google, Facebook, Microsoft and Amazon have popularized the use of ML and data science competitions have engaged a new generation of young scientists in this wake. Today, governments and corporations keep identifying new applications of ML and, with the increased availability of open data, everyone seems to be in need of a learning machine. Unfortunately, the success of ML applications in many domains relies heavily on the skills of data scientists who design model architectures and/or select a good set of hyper-parameters. My PhD thesis aims at advancing the field of Automated Machine Learning (AutoML) to develop a methodology to automate the design of learning machines as far as possible. Specifically, in the context of supervised learning feature-based representations only, I am working on producing black-box machine learning algorithm that can process “any data", in “any time", with “any resource". This means selecting an appropriate model with its hyper-parameter values, and outputs the desired type of predictions, taking into account available computational resources and a computational time budget. Borrowing from the vocabulary of Reinforcement Learning (RL), my goal is to design an **AutoML agent** whose objective is to quickly discover the best suited algorithm to solve a given task.

Although the acronym AutoML was coined recently by Frank Hutter and collaborators, the problems of automated machine learning, model selection, and hyper-parameter optimization have been studied for several decades in the machine learning community. The following brief overview borrows material from our paper [46].

1.2 Scope of AutoML

The overall AutoML problem covers a wide range of difficulties, which cannot be addressed all at once in a single challenge. To name only a few: data “ingestion" and formatting, pre-processing and feature/representation learning, detection and handling of skewed/biased data, inhomogeneous, drifting, multimodal, or multi-view data (hinging on transfer learning), matching algorithms to problems (which may include supervised, unsupervised, or reinforcement learning, or other settings), acquisition of new data (active learning, query learning, reinforcement learning, causal experimentation), management of large volumes of data including the creation of appropriately sized and stratified training, validation, and test sets, selection of algorithms that satisfy arbitrary resource constraints at training and run time, the ability to generate and reuse workflows, and generating explicative reports.

Therefore, restricting the scope of our thesis work was of great importance to ensure that intermediate milestones of immediate practical interest are reached, as further discussed in the rest of this chapter.

1.2.1 The AutoML setting

For the purpose of this chapter, a predictive model (or model for short) has the form $y = f(\boldsymbol{x}) = f(\boldsymbol{x};\boldsymbol{\alpha})$ with a set of parameters $\boldsymbol{\alpha} = [\alpha_0, \alpha_1, \alpha_2, ..., \alpha_n]$ trainable with a learning algorithm (trainer). The trained model (predictor) $y = f(\boldsymbol{x})$ produced by the trainer is evaluated by an objective function $J(f)$, used to assess the model performance on test data.

A model hypothesis space is defined by a vector $\boldsymbol{\theta} = [\theta_1, \theta_2, ..., \theta_n]$ of hyper-parameters, which may include both categorical variables corresponding to switching between alternative models and other modeling choices such as preprocessing parameters, type of kernel in a kernel method, number of units and layers in a neural network, or training algorithm regularization parameters [94]. Some authors refer to this problem as *full model selection* [31, 100], others as the CASH problem (Combined Algorithm Selection and Hyperparameter optimization) [108].

We will then denote hyper-models as

$$y = f(\boldsymbol{x};\boldsymbol{\theta}) = f(\boldsymbol{x};\boldsymbol{\alpha}(\boldsymbol{\theta}),\boldsymbol{\theta}), \tag{1.1}$$

where the model parameter vector $\boldsymbol{\alpha}$ is an implicit function of the hyper-parameter vector $\boldsymbol{\theta}$ obtained by using a trainer for a fixed value of $\boldsymbol{\theta}$, and training data composed of input-output pairs $(\boldsymbol{x}_i, y_i)$.

The goal of AutoML is to devise algorithms capable of searching efficiently for the best hyper-parameters $\boldsymbol{\theta}$.

1.3 Computational and statistical aspects of AutoML

As an **optimization problem**, model selection is a bi-level optimization program [24, 26, 10]; there is a lower objective to train the parameters $\boldsymbol{\alpha}$ of the model (*e.g.* the weights of a neural network), and an upper objective to train the hyper-parameters $\boldsymbol{\theta}$ (*e.g.* the number of layers and units per layer), both optimized simultaneously. A related problem, not addressed in this thesis, is that of building optimal ensembles of models in which "base learners" vote to make the final decision [20, 39, 22]. Practically, computational resources are always limited: for each task there is always an upper bound on execution time and memory available, and a given number of available CPUs and/or GPUs. This places a constraint on the AutoML agents to produce a solution in a given time, and hence to optimize the model search from a computational point of view.

As a **statistics problem**, model selection is a problem of multiple testing in which error bars on performance prediction degrade with the number of models/hyper-parameters tried or, more generally, model complexity [117]. Two common pitfalls should be avoided: over-fitting and under-fitting. By over-fitting we mean selecting a too complex model that performs well on training data but performs poorly on unseen data, *i.e.* models that do not generalize. By under-fitting we mean selecting a too simple model, which does not capture the complexity of the data, and hence performs poorly both on training and test data. The initial goal in this thesis was to jointly address the problem of over-fitting/under-fitting and the problem of efficient search for an optimal solution, as stated in [59]. But, while until recently, a key aspect of AutoML was to avoid over-fitting, with the advent of "big data" and progress made in learning theory, the emphasis has now shifted to the computational aspect of the problem: that of finding quickly the best performing model and principally avoiding under-fitting.

This was confirmed in our analysis of the first AutoML challenge summarized in Chapter 2: we found that the computational contraints have been far more challenging to participants than the problem of overfitting. The main contributions of the winners have been to devise novel efficient search techniques with cutting edge optimization methods. Practically, if very large datasets are available, it is sufficient for model selection to use a single training/validation set split and rely on the validation set performances to rank algorithms, the additional refinement of performing cross-validation or using bootstrap estimators [49] providing only a minor advantage, given the additional computational cost.[1] Likewise, in several recent competitions [46], we observed that the ranking of algorithms obtained using a (large) validation set, providing immediate feed-back on the "public" leaderboard, was usually similar to that obtained with another (large) independent test set concealed to the participants (on a "private" leaderboard).

Based on these observations, in the remainder of my thesis, we make the simplifying assumption that performances on the test set are the "gold standard" to determine whether one algorithm is better than another.

1.4 Overview of hyper-parameter selection

Everyone who has modeled data has had to face some common modeling choices: scaling, normalization, missing value imputation, variable coding (for categorical variables), variable discretization, degree of nonlinearity and model architecture, among others. ML has managed to reduce the number of hyper-parameters and produce *black-boxes* to perform tasks such as classification and regression [49, 28]. Still, any real-world problem requires at least some preparation of the data before it can be fitted into an "automatic" method, hence requiring some modeling choices. Much progress has been made on *end-to-end solutions* for more complex tasks such as text, image, video, and speech processing, using deep-learning methods [9]. However, even these methods have many modeling choices and hyper-parameters. As part of our initial thesis work, we surveyed the literature on hyper-parameter selection.

To simplify the discussion that follows, we lump all parameters describing models and hyper-parameters in a single vector $\boldsymbol{\theta}$, but more elaborate structures, such as trees or graphs can be used to define the hyper-parameter space [111].

Most practitioners use heuristics such as grid search or uniform sampling to sample $\boldsymbol{\theta}$ space, and use k-fold cross-validation as the upper-level objective [27]. In this framework, the optimization of $\boldsymbol{\theta}$ is not performed sequentially [11]. All the parameters are sampled along a regular scheme, usually in linear or log scale. This leads to a number of possibilities that exponentially increases with the dimension of $\boldsymbol{\theta}$. There is a lack of principled guidelines to determine the number of grid points and the value of k in k-fold cross-validation, and there is no guidance for regularizing the upper-level objective. Although progress has been made in experimental design to reduce the risk of over-fitting [58, 67], in particular by splitting data in a principled way [98],

[1] The most popular such method is k-fold cross-validation. It consists of splitting the dataset into k folds; $(k-1)$ folds are used for training and the remaining fold is used for testing; eventually, the average of the test scores obtained on the k folds is reported.

to our knowledge, no one has addressed the problem of optimally splitting data. However, as noted before, the problems of optimally splitting data and regularizing the upper-level objective are alleviated when big data are available.

When hyper-parameters are continuous, instead of discretizing them (as done with grid search), efforts have been made to optimize directly continuous hyper-parameters with bi-level optimization methods, using either the k-fold cross-validation estimator [10, 79] or the leave-one-out estimator as the upper-level objective. The leave-one-out estimator may be efficiently computed, in closed form, as a by-product of training only one predictor on all the training examples (*e.g.* virtual-leave-one-out [45]). The method was improved by adding a regularization term [23]. Gradient descent has been used to accelerate the search, by making a local quadratic approximation of [61]. In some cases, the full upper-level objective function can be computed from a few key examples [48, 85]. Other approaches minimize an *approximation* or an *upper bound* of the leave-one-out error, instead of its exact form [83, 116]. Nevertheless, these methods are still limited to specific models and continuous hyper-parameters.

The so-called CASH problem (Combined Algorithm Selection and Hyperparameter optimization) was coined in 2013 by Thornton and collaborators [110]. An early attempt at solving the CASH problem in 2002 was the *pattern search* method that uses k-fold cross-validation. It explores the hyper-parameter space by steps of the same magnitude and, when no change in any parameter further decreases the objective function, the step size is halved and the process repeated until the steps are deemed sufficiently small [78]. In 2009, [31] addressed the CASH problem using Particle Swarm Optimization, which optimizes a problem by having a population of candidate solutions (particles), and moving these particles around the hyper-parameter space using the particle's position and velocity.

While regularizing the second level of inference is a recent addition to the frequentist ML community, it has been an intrinsic part of Bayesian modeling via the notion of hyper-prior. Some methods of multi-level optimization combine importance sampling and Monte-Carlo Markov Chains [3]. The field of Bayesian hyper-parameter optimization has rapidly developed and yielded promising results, in particular by using Gaussian processes to model generalization performance [96, 105]. But Tree-structured Parzen Estimator (TPE) approaches modeling $P(\boldsymbol{x}|y)$ and $P(y)$ rather than modeling $P(y|\boldsymbol{x})$ directly [14, 13] have been found to outperform GP-based Bayesian optimization for structured optimization problems with many hyperparameters including discrete ones [30]. The central idea of these methods is to fit the upper-level objective to a smooth function in an attempt to reduce variance and to estimate the variance in regions of the hyper-parameter space that are under-sampled to guide the search towards regions of high variance. These methods are inspirational and some of the ideas can be adopted in the frequentist setting. For instance, the random-forest-based SMAC algorithm [53], which has helped speed up both local search and tree search algorithms by orders of magnitude on certain instance distributions, has also been found to be very effective for the hyper-parameter optimization of machine learning algorithms, scaling better to high dimensions and discrete input dimensions than other algorithms [30].

We also notice that Bayesian optimization methods often combine with other techniques such as meta-learning and ensemble methods [35] in order to gain advantage in some challenge settings with time budget limit [44]. Some of these methods consider jointly the two-level optimization and take time cost as a critical guidance for hyper-parameter search [106, 63].

Besides Bayesian optimization, several other families of approaches exist in the literature and have gained much attention with the recent rise of deep learning. Ideas borrowed from *reinforcement learning* have recently been used to construct optimal neural network architectures [123, 5]. These approaches formulate the hyper-parameter optimization problem in a reinforcement learning flavor with, for example, states being the actual hyper-parameter setting (*e.g.* network architecture), actions being added or deleting a module (*e.g.* a CNN layer or a pooling layer), and reward being the validation accuracy. They can then apply off-the-shelf reinforcement learning algorithms (*e.g.* RENFORCE, Q-learning, Monte-Carlo Tree Search) to solve the problem. Other architecture search methods use *evolutionary* algorithms [91, 4]. These approaches consider a set (population) of hyper-parameter settings (individuals), modify (mutate and reproduce) and eliminate unpromising settings according to their cross-validation score (fitness). After several generations, the global quality of the population increases. One important common point of reinforcement learning and evolutionary algorithms is that they both deal with the exploration-exploitation trade-off. Despite the impressive results, these approaches require huge amount of computational resources and some (especially evolutionary algorithms) are hard to scale. [88] recently proposed weight sharing among child models to largely speed up [123] while achieving comparable results.

Finally, splitting the problem of parameter fitting into two levels can be extended to multiple levels, at the expense of extra complexity—*i.e.* need for a hierarchy of data splits to perform multiple or nested cross-validation [29], insufficient data to train and validate at the different levels, and increase of the computational load.

In conducting this survey and analyzing the results of the AutoML challenge (Chapter 2), we arrived at the conclusion that, while still an active area of research, hyper-parameter selection has arrived at a good level of maturity, and is no longer the bottleneck of AutoML. In contrast, meta-learning remains a very uncharted territory, with very promising initial results. Hence we focused most of our effort in this thesis on meta-learning.

1.5 Filter, wrappers, and embedded methods

Borrowing from the conventional classification of feature selection methods [64, 16, 45], model search strategies can be categorized into filters, wrappers, and embedded methods. **Filters** are methods for narrowing down the model space, without training the learner. Such methods include preprocessing, feature construction, kernel design, architecture design, choice of prior or regularizers, choice of noise model, and filter methods for feature selection. Although some filters use training data, many incorporate human prior knowledge of the task or knowledge compiled from previous tasks. Recently, [8] proposed to apply collaborative filtering methods to model search. **Wrapper methods** consider learners as a black-box capable of learning from examples and making predictions once trained. They operate with a search algorithm in the hyper-parameter space (grid search or stochastic search) and an evaluation function assessing the trained learner's performance (cross-validation error or Bayesian evidence). **Embedded methods** are similar to wrappers, but they exploit the knowledge of the learning machine algorithm to make the search more efficient. For instance, some embedded methods compute the leave-one-out solution in a closed form, without leaving anything out, *i.e.* by performing a single model training on all the training data (*e.g.* [45]). Other embedded methods jointly optimize parameters and hyper-parameters [61, 80, 79].

The focus of this thesis is on filter and wrapper methods. In Chapter 2, in the context of the work I performed on the AutoML challenge, I present methods of hyper-parameter selection, which are typically wrapper methods. In Chapters 5 and 6, I turn to meta-learning, whose objective is to produce filters (methods to select algorithms without actually training and testing them).

1.6 Summary of my thesis objectives

The original focus of the AutoML community was on over-fitting avoidance. But with the emergence of "big data" the current emphasis is on model search efficiency. Novel effective approaches have been proposed in the academic literature that have become wide spread among practitioners because they are both theoretically well founded and practically efficient, *e.g.* Bayesian optimization (BO). Such approaches build a posterior $p(model|data)$ by applying candidate models to the input data and use this posterior distribution to guide the search (e.g. [54], [106]). Complementary to such approaches, Meta Learning develops a set of meta-features capturing the nature of data, which are then used to infer the model performance based on past experiences on similar data, without actually training the model (e.g. [82]). Meta learning has been used to initialize BO search [35]).

The original formulation of the problem of AutoML is limited to search for the best model/hyper-parameters for a single dataset/task. In this thesis, my ambitions are to tackle a Lifelong Machine Learning problem, which, 1) given a dataset, can suggest promising models; 2) given available models, can suggest suitable datasets to learn to reinforce its problem solving capacity. To that end, I developed Meta Learning techniques to perform these tasks using first active learning (see Chapter 5), then Reinforcement Learning (RL) to learn an optimized model search policy (see Chapter 6).

Chapter 2

Empirical setting

In this chapter, we first give a brief account of the main findings of our post-hoc analysis of the first AutoML challenge (2015-2016) [46]. This competition challenged the participants to submit code that solve classification and regression problems from fixed-length feature representations, without any human intervention. The challenge setting and the datasets have been used extensively thoughout my thesis work and the lessons learned from the challenge have influenced my research directions towards meta-learning.

The Code for the challenge analysis is at
https://github.com/LishengSun/AutoML-2016-simulations-and-analyses.

2.1 Lessons learned from the AutoML challenge

The objective of the AutoML challenge series (http://automl.chalearn.org) is to push research towards creating “universal learning machines” capable of learning and making predictions without human intervention. This means that the participants must deliver code, which is blind tested on datasets never released before. The first AutoML challenge (to which I participated) was limited to:

- **Supervised learning** problems (classification and regression).
- **Feature vector** representations.
- **Homogeneous datasets** (same distribution in the training, validation, and test set).
- **Medium size datasets** of less than 200 MBytes.
- **Limited computer resources** with an execution times of less than 20 minutes per dataset on an 8 core x86_64 machine with 56 GB RAM.

Within this constrained setting, the testbed was composed of 30 datasets from a wide variety of application domains (medical diagnosis, speech recognition, credit rating, prediction of drug toxicity/efficacy, classification of text, prediction of customer satisfaction, object recognition, protein structure prediction, action recognition in video data) and ranged across different types of complexity (class imbalance, sparsity, missing values, categorical variables). In this limited framework, there remain many modeling choices.

Many robust learning machines with a reduced number of hyper-parameters have emerged in the recent years in an effort to produce *perfect black-boxes* to perform tasks such as classification and regression [49, 28]. But the availability of toolboxes rich in such models, *e.g.* Weka [47] or scikit-learn [87], has not eliminated modeling choices. Similarly to AI, which has endeavored to pass the Turing test, ML has undertaken the task of beating the “no free lunch theorem”, stating that no ML algorithm can be superior to all others on every task. Tools like AUTO-SKLEARN [37, 35, 38][1], a wrapper around the scikit-learn library built by the winners of the AutoML challenge have made big strides towards that goal.

Statistical complexity *vs.* computational complexity

The dilemma of model selection is to avoid “searching too hard” (and falling in the trap of over-fitting) and “not searching hard enough” (and falling in the trap of under-fitting). One often refers to as **statistical complexity** all ailments related to the “curse of dimensionality” or solving ill-posed problems, in which not enough training data is available to ensure good generalization. Another notion of complexity, complementing the first one, is **computational complexity**: exploring exhaustively (or very intensively) a very large model space by evaluating “all” (or very many) models is generally infeasible because of the combinatorial nature of the problem of probing simultaneouly several hyper-parameters. Success in the AutoML challenge depends on addessing both types of complexity.

Best practices for model selection converge towards the Ockham’s razor principle, which prescribes limiting model complexity to the minimum necessary to explain the data, or *shave off unnecessary parameters*. This has been grounded in theory over the past few years in such frameworks as regularization, Bayesian priors, Minimum Description Length, Structural Risk Minimization (SRM), and the bias/variance trade-off [93, 41, 49, 28, 117]. With modern learning machines designed to regularize and prevent overfitting, good practitioners usually do not over-train their models. Indeed, in the challenge, our analyses revealed no over-fitting of models. Two means of com-

[1] https://automl.github.io/auto-sklearn/stable/

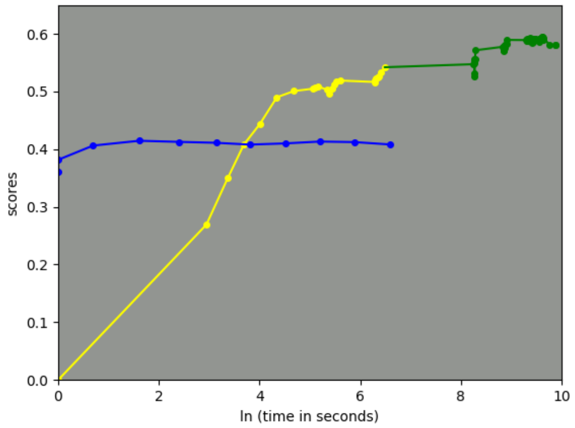


Fig. 2.1 **Learning Curve of 'aad_freiburg' (yellow) and 'abhishek' (blue) for the evita dataset.**
Abhishek starts with a better solution but performs a less efficient exploration. In contrast, *aad_freiburg* starts lower but ends up with a better solution. In green: the 'aad_freiburg' learning curve keeps improving beyond the time limit imposed in the challenge (20 min).

batting over-fitting are pervasive: using cost functions **penalizing model complexity** and **ensembling**. The latter has been adopted by all winners, see Section 2.1.

The most pressing problem in today's AutoML research is therefore that of underfitting, which cannot be solved by brute-force search if computational resources are limited. In the next section, we review what "clever search" entails in today's state-of-the art.

Heuristic search *vs.* Bayesian optimization

Model search (including hyper-parameter search, jointly referred to as HP search) involves two necessary components: (1) a means of estimating the performance of the model on future data (estimator), and (2) a strategy for exploring the search space (policy).

For problem 1, there is presently a large consensus for the choice of estimators. Obviously, when vast amounts of training data are available, reserving a single subset of the training data for validation is simple and efficient. Otherwise, common practice is to use some form of cross-validation (CV). **K-fold CV** and its variants are a favorite, in front of **bootstrapping** (*e.g.* bagging used in Random Forests). For K, most people use K=10, although there is no clear theoretical foundation for this choice. It has been known for decades that a special kind of CV estimator, the leave-one-out estimator, can be efficiently approximated by training a single model (*e.g.* virtual-leave-one-out [45]). Yet such methods are not applicable to all algorithms and require dedicated code, so they are not popular amongst practitionners, which prefer using plain CV in a wrapper setting. Likewise, techniques of bilevel optimization, optimizing simultaneously parameters and hyper-parameters (*e.g.* [10, 79]) have not gained in popularity for the same reason.

Problem 2 is presently the main focus in AutoML research: develop efficient search policies. The 'aad_freiburg' team (who developped *AUTO-SKLEARN* [37, 35, 38] and dominated both the 2015-2016 AutoML challenge and its 2018 sequel) used a method inspired by **Bayesian optimization** [30]. The key idea is to guide the search with a "cheap" evaluation of models. CV is thus used to evaluate only a few candidate points in HP space. A predictor of model performance in HP space is built with a form of active learning. Random Forest (RF) regressors lend themselves particularly well to this exercise and have superseded Gaussian processes [Hutter et al.]. One reason is that they are based on decision trees, which are hierarchical in nature, thus making it easy to map a hierarchy of hyper-parameters. Another reason is that, as an ensemble method, RF yields also an estimator of the variance of the predictions. Armed with an estimation of the expectation and the variance of the model to be evaluated, Bayesian optimization methods estimate the expected benefit of effectively training and testing a new model. Typically, this is a function expressing the exploration/exploitation tradeoff, *i.e.* you want to explore regions of high variance (where your predictions are least confident) but not waste too much time exploring if there are low hanging fruits (models with good performances candidates for winning).

Another form of Bayesian optimization, which was very strong in the first AutoML challenge, is "freeze-thaw" (introduced by J. Lloyd in the first phases whose code was overtaken by S. Sun, placing 3rd in the final phase) [73]. Other top ranking participants used various forms of **heuristic search** with performances that ended up nearly as good. It is difficult to tell apart at this stage the influence of various factors in the success of methods. Initialization played an important role. We show a typical example of learning curve in Figure 2.1. Given enough time, the Bayesian optimization method ('aad_freiburg') ends up with better performance, but Abhishek has a better initialization.

The strongest contender to such "clever search" methods is plain **grid search** applied to models having only very few hyper-parameters. Grid search applied to gradient tree boosting is a typical illustration of such approach, which has been very successful in challenges, since at least 2006 [74]. The Intel team produced very good results with such methods in the AutoML challenge.

Finally, **search free methods** are worth mentionning. Marc Boullé applied the Selective Naive Bayes (SNB) [17, 18] extending the Naive Bayes method for classification and regression. His software developed by Orange Labs and in use in production, was used in the challenge with minimal adaptation to make it compliant to the challenge settings. Without any further tuning, it returned a solution with honorable results, within the time limit of the challenge. It is therefore a very strong baseline.

Meta-learning

Meta-learning aims at defining some general principles over different datasets[2]. The 'aad_freiburg' team investigated meta-learning applied to the initialization of Bayesian search. Specifically, they considered 140 datasets from `openml.org`[114] (a platform which allows to systematically run algorithms on datasets) and they defined meta-features of datasets, including simple statistics characterizing input and output space, and the performance of a few landmark algorithms such as one nearest neighbor (1NN) and decision tree. They also ran AUTO-SKLEARN on these datasets and recorded the best performing algorithm for each dataset. Given a new dataset, and considering its neighbors in terms of meta-features, they can then initialize the HP search for the new dataset with the algorithms performing best on its neighbors, resulting in significant improvements compared to random initialization of the HP search.

To further understand the success of this method, we investigated which meta-features are most predictive of the best performing models. To that end we performed the following experiment using scikit-learn. We excluded *landmark models* from the set of meta-features and built a linear discriminant classifier (LDA) to predict which of four *basic models* would perform best on the 30 datasets of the challenge, thus defining a 4-class classification problem. Basic models included Naive Bayes (NB), Stochastic Gradient Descent linear model (SGD-linear), K=Nearest neighbor (KNN), and Random Forest (RF), with default hyper-parameter settings. The results shown in Figure 2.1 reveal three clusters in the space of the two first LDA components. Further, the features that contribute most to the first two LDA components are the fraction of missing values and features characterizing the distribution of target values.[3]

[2] Meta-learning differs from transfer learning, which is concerned with transferring models/knowledge among tasks. Transfer learning can take various forms depending on the type of information that overlaps between tasks [84], *i.e.* similarity of input space distribution and/or similarity of outputs/labels. In the AutoML challenge framework however, the diversity among the application domains and types of learning difficulties hinders transfer learning, that will not be considered further in the paper. Actually, the help of using transfer learning in such competition is an open question. We have seen in other past challenges lending themselves to transfer learning that most (if not all) the participants did not do any transfer learning, even through on the long run transfer learning proved to be useful. The main problem may be that challenges are time constrained and transfer learning pays off only if you do it right and have enough time.

[3] Given the small size of available datasets (only 30 datasets in total), the LDA is trained on all datasets, and the model has no generality.

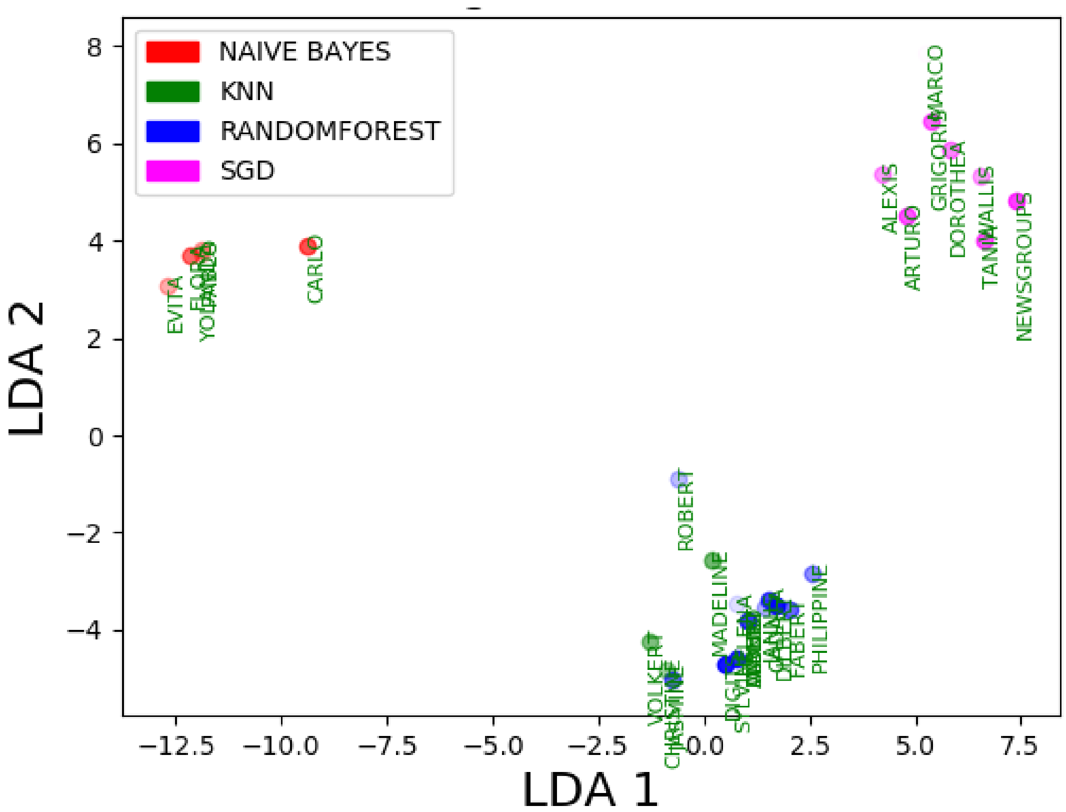


Fig. 2.2 **Linear Discriminant Analysis.** (a)We trained LDA using (X=meta features, except landmarks; y=which model won of four basic models (NB, SGD-linear, KNN, RF). The models were trained with default hyper parameters. In the space of the two first LDA components, each point represents one dataset. The colors code for the winning basic models. The color transparency reflects the scores of the corresponding winning model (better is darker).

Although the ‘aad_freiburg’ team showed in their 2015 paper [35] that this initialization faired better than random initialization, there is still room for improvement. Indeed, learning curves of the first AutoML (of which an example is shown in Figure 2.1) have revealed that other competitors had far better initializations.

For the 2018 edition of the AutoML challenge, the ‘aad_freiburg’ team introduced a novel strategy: *Portfolio Successive Halving*-AUTO-SKLEARN. As a form of meta-learning, they created a fixed portfolio of machine learning pipelines using over 400 datasets. At each period, the less successful pipelines (as estimated from the Bayesian optimization model), are discarded along the so-called *Bayesian Optimization HyperBand* (BO-HB) [32, 34].

Towards AutoML: engineering *vs.* principles

Overall, we can ask ourselves whether the AutoML challenge helped pushing “the science of AutoML” and whether some design guidelines have emerged and whether the “no free lunch” theorem has been beaten.

One thing is sure: ensembling always helps[4], whether you choose a homogeneous ensemble like Random Forests or a heterogeneous ensemble, built *e.g.* with the method of [22]. Other design choices regarding meta-learning and search are still evolving. However, what drives most progress in the field is the emergence of simple new concepts that researchers can share and re-implement to reproduce results. In that respect, Bayesian optimization has been helpful.

To reproduce the results of the challenge, there is one good news and one bad news. The good news is that all the code is open-sourced[5]. The bad news is that if you want to write your own code based on *e.g.* scikit-learn and the principles outlined in this paper, it will be a significant amount of engineering. First, most methods of scikit-learn will die on you for many datasets (out-of-memory or out-of-time). Second, there are a lot of tricks of the trade to perform meta-learning and get Bayesian optimisation to work.

So it may be far easier to create your own code from scratch and create your own “universal approximator” with a handful of hyper-parameters tunable with grid search. But be careful, many other people have tried; scikit-learn is full of such models. Figure 2.3 shows the performance of “pure models” *vs.* the performances of the challenge winners. They are lagging behind. It is not so easy to beat the “no free lunch” theorem.

Other engineering aspects play an important role. In the third round of the challenge when large sparse datasets were introduced, the vast majority of methods failed blind

[4] This phenomenon has also been observed heavily in Kaggle competitions[Kag]

[5] http://automl.chalearn.org

testing by either running out of time or out of memory. Ironically, the winners won thanks to proper exception handling (they returned random results rather than failing). During the "tweakathon" phase that followed blind testing the participants had an opportunity to fix their code. This revealed that the tasks of round 3 were not particularly difficult, once engineering problems were dealt with.

You may also be tempted to go beyond Bayesian optimization in the line of research pursuing heterogeneous model search, by performing better meta-learning or even by learning policies with reinforcement learning. Some ideas along these lines have been proposed in the literature [123, 5] and for the first time in 2018, one of the top participants (wiWangl) used Q-Learning to learn machine learning pipelines.

You may be tempted to use neural networks or deep learning. Unfortunately, for such time constrained challenge, even with GPUs (we provided GPUs in round 4 of the first AutoML challenge), they are not among the best performing methods.

In conclusion, it is fair to say that the winners provided a well engineered solution satisfying the constraint of the challenge in terms of time budget and robustness to algorithm failures, but for any new proposed task, manually selected and fined tuned algorithms may still perform better.

Discussion: challenge and benchmark design

The diversity of the 30 datasets of our first AutoML challenge was both a feature and a curse: it allowed us to test the robustness of software across a variety of situations, but made meta-learning difficult (datasets being different with respect to meta-features). Likewise, we attached different metrics (loss functions) to each dataset. This contributed to making the tasks more realistic and more difficult, but also made meta-learning harder. Consequently external datasets must be used if meta-learning is to be explored for the AutoML challenge tasks. As previously mentioned, this was the strategy adopted by the AAD Freiburg team, which used the OpenML data for meta training. They used over 400 datasets for meta-learning in last challenge edition.

With respect to task design, we learned that the devil is in the details. Challenge participants generally solve exactly the task proposed by the organizers, to the point that their solution may not be adaptable to seemingly similar scenarios. In the case of the AutoML challenge, we pondered whether the metric of the challenge should be the area under the learning curve (plotting performance as a function of time) or one point on the learning curve (the performance obtained after a fixed maximum computational time elapsed). We ended up favoring the second solution for practical reasons. Examining after the challenge the learning curves of some participants, it is quite clear that the two problems are radically different, particularly with respect

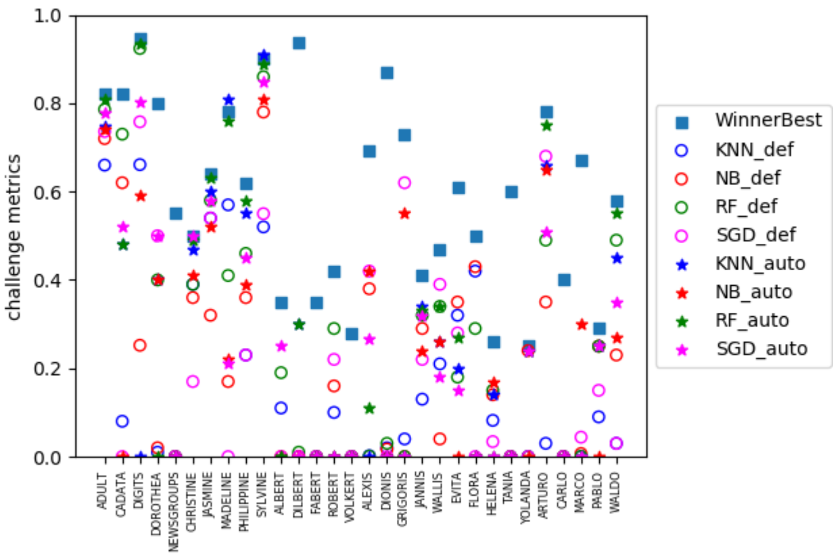


Fig. 2.3 **Comparison of methods (2015-2016 challenge) including basic methods (-def suffix), basic methods with optimised HP (-auto suffix), and challenge winners.** Winners in general win over basic methods, even with optimized HPs. There is no basic method that dominates all others. Though RF-auto (Random Forest with optimised HP) is very strong, it is often outperformed by other methods and sometimes by RF-def (Random Forest with default HP). Thus, under the tight computational constraints of the challenge, optimizing HP does not always pay. For KNN though, time permitting, optimizing HP generally helps by a long shot. Interestingly, KNN wins, even over the challenge winners, on some datasets.

to strategies mitigating “exploration” and “exploitation”. This prompted us to think about the differences between “fixed time” learning (the participants know in advance the time limit and are judged only on the solution delivered at the end of that time) and “any-time” learning (the participants can be stopped at any time and asked to return a solution). Both scenarios are useful: the first one is practical when models must be delivered continuously at a rapid pace, e.g. for marketing applications; the second one is practical in environments when computational resources are unreliable and interruption may be expected (e.g. people working remotely via an unreliable connection). This will influence the design of future challenges.

Also regarding task design, both AutoML challenges differ in the sequence of difficulties tackled in each round. In the 2015/2016 challenge, round 0 introduced five datasets representing a sample of all types of data and difficulties (types of targets, sparse data or not, missing data or not, categorical variables of not, more examples than features or not). Then each round ramped up difficulty introducing each time 5 new datasets. But in fact the datasets of round 0 were relatively easy. Then during each round, the code of the participants was blind tested on data that were one notch harder than in the previous round. Hence transfer was quite hard. In the 2018 challenge, we had only 2 phases, each with 5 datasets of similar difficulty and the datasets of the first phase were each matched with one corresponding dataset on a similar task in the second phase. As a result, transfer was made simpler.

Concerning the starting kit and baseline methods, we provided code that ended up being the basis of the solution of the majority of participants (with notable exceptions from industry such as Intel and Orange who used their own "in house" packages). Thus, we can question whether the software provided biased the approaches taken. Indeed, all participants used some form of ensemble learning, similarly to the strategy used in the starting kit. However, it can be argued that this is a "natural" strategy for this problem. But, in general, the question of providing enough starting material to the participants without biasing the challenge in a particular direction remains a delicate issue.

From the point of view of challenge protocol design, we learned that it is difficult to keep teams focused for an extended period of time and go through many challenge phases. We attained a large number of participants (over 600) over the whole course of the AutoML challenge, which lasted over a year (2015/2016) and was punctuated by several events (such as hackathons). However, few teams participated to all challenge rounds and despite our efforts to foster collaboration, the general spirit was competitive. It may be preferable to organize yearly events punctuated by workshops. This is a natural way of balancing competition an cooperation since workshops are a place of exchange where participants get rewarded by the recognition they gain via the system of scientific publications. As a confirmation of this conjecture, the second instance of the AutoML challenge (2017/2018) lasting only 4 months attracted nearly 300 participants.

One important novely of our challenge design was "code submission". Having the code of the participants executed on the same platform under rigorously similar conditions is a great step towards fairness and reproducibility, as well as ensuring the viability of solutions from the computational point of view. We have imposed to the winners to release their code under an open source licence to win their prizes. This was good enough an incentive to obtain several publicly available software as the "product" of the challenges we organized. In our second challenge (AutoML 2018), we have made use of dockers. Distributing the docker makes it possible for anyone downloading the code of the participants to reproduce easily the results without stubling upon installation problems due to inconsistencies in computer environments and libraries. Still the hardware may be different and we find that, in post-challenge evaluations, changing computer may yield significant differences in results. Hopefully, with the generalization of use of cloud computing that is becoming more affordable, this will become less of an issue.

2.2 Definition of Meta-Learning problem

As stated earlier, the analysis of the AutoML challenge results drive us to the conclution that, hyper-parameter selection, while still being an active area of research, reaches a good level of maturity. The next bottleneck of AutoML will be meta-learning, which we are going to define in this section.

Meta-learning is learning across datasets, which aims at finding a good algorithm on a new dataset, given the past experiences. To define the meta-learning problem properly, we first define its learning source, the meta-dataset:

Definition 1 (Meta-dataset)**.** *A meta-dataset S is a collection of Machine Learning datasets with algorithms applied on them. It can be expressed as a $(n \times m)$ matrix, where row i (for $i = 1, \dots, n$) corresponds to dataset D_i, and column j (for $j = 1, \dots, m$) corresponds to algorithm A_j, and the element $S(i, j)$ represents the performance score of A_j applied on D_i.*

Then, the meta-learning problem is defined as:

Definition 2 (Meta-Learning Problem)**.** *Given a meta-dataset S of dimension $((n-1) \times m)$ and a new dataset D_n, the meta-learning problem is to find the best algorithm j_* for D_n that satisfies:*

$$j_* = \operatorname*{argmax}_{j=1,\dots,m} S(n, j)$$

The next section introduces the meta-datasets we use for experiments presented in this thesis, then Chapter 5 and 6 propose two different approaches to solve the meta-learning problem.

2.3 Design of benchmarking meta-learning datasets

The AutoML challenge described in the previous section kick-started our research on meta-learning by providing a wealth of datasets and performances of many algorithms on the tasks defined on such datasets. This yielded our first meta-learning dataset, which is a meta-dataset S containing all algorithm performances (datasets in lines and algorithms in columns). However, in meta-learning, each "example" being a task (or dataset), it is difficult to gather enough examples to get a good benchmark. Our

Table 2.1 **Statistics of benchmark meta-datasets used**. #Datasets=number of datasets, #Algo=number of algorithms, Rank=rank of the performance matrix.

	Artificial	StatLog	OpenML	2015/2016 AutoML	CEP	MNIST-patch
#Dataset	50	22	76	30	8608	70,000
#Algo	20	24	292	17	163	49
Rank	20	22	76	17	118	947
Metric	None	Error rate	Accuracy	BAC or R^2	Error rate	Averaged pixel values
Pre-processing	None	Take square root	None	Scores for aborted algo. set to 0	Normal. on X (i.e. features), Errors for aborted algo. set to 1	Neighbor. average within a square window
Source	Generated by authors	StatLog Dataset in UCI database	Alors [75] website	2015/2016 AutoML	Generated by the authors from Causality chal-lenge	Generated by the authors from MNIST dataset

solution to that problem has been to create a collection of meta-learning datasets from several sources.

In this section, we describe all meta-learning datasets we have generated / gathered for the study of meta-learning in Chapters 3, 5, 6. Table 2.1 summarizes the statistics our six resulting meta-learning datasets, which we now describe in further detail.

2.3.1 Real-world meta-datasets: AutoML, StatLog, OpenML

This section describes three real-world meta-datasets: AutoML from the previously introduced AutoML challenge, OpenML from the OpenML platform and used in [75, 101], as well as StatLog from the UCI database.

AutoML meta-dataset

The main product of our post-challenge studies of the first AutoML challenge 2015/2016, from the point of view of the long term research, is the meta-learning dataset (meta-dataset) that we generated, consisting of a meta-dataset matrix S grouping the performance of 17 learning algorithms applied on all 30 datasets of the challenge. This is an essential resource we use to develop meta-learning algorithms addressing the AutoML problem as a MDP in later Chapters 3, 5 and 6.

We visualized the data in Figure 2.4. We will show a similar visualization for other meta-datasets described in the remainder of the chapter. From top to bottom:

TOP - Two-way hierarchical clustering. This allows us to see that there is some structure in data, which can potentially be exploited by meta-learning algorithms, although there is not a very marked "block" structure, which would indicate that some subsets of algorithms are more suitable to solve some subsets of tasks.

MIDDLE - Spectrum of singular values. This shows that, even though the matrix is full rank, some singular values [6] are more prominent and "explain" a large fraction of the variance in data, confirming the potential for meta-learning algorithms.

BOTTOM - Top ranking algorithms. We indicate with a red dot the best algorithm. This shows that the winning method is not always the same and therefore that an automatic method that could predict which algorithm performs best would be useful.

- **Datasets of the AutoML challenge (lines of S_{AutoML})** We describe now in detail the datasets of the 2015/2016 AutoML challenge, which we used not only to create our fist benchmark meta-learning dataset, but also to generate the plots mentioned in section 2.1 and other post-challenge studies [46].

  The organizers of the challenge gathered a pool of 70 datasets during the summer 2014 with the help of numerous collaborators, and ended up selecting 30 datasets for the 2015/2016 challenge (see Table 2.2), chosen to illustrate a wide variety of domains of applications: biology and medicine, ecology, energy and sustainability management, image, text, audio, speech, video and other sensor data processing, Internet social media management and advertising, market analysis and financial prediction. They preprocessed data to obtain feature representations (i.e., each example consists of a fixed number of numerical coefficients).

[6] The singular values are from Singular Value Decomposition (SVD) [42]. SVD is a matrix factorization method that factorizes a $(m \times n)$ matrix M into three matrices: $M = U\Sigma V^T$, where U and V are unity matrices with dimension $(m \times m)$ and $(n \times n)$ and called left and right singular matrix of M respectively, Σ is a diagonal $(m \times n)$ matrix with non-negative values on the diagonal which are called singular values of M, and express the 'proportion' of information contained in M. Keeping the k (with $k < n$) greatest singular values and replacing the rest with 0 to form Σ', then recompose $\hat{S} = U\Sigma' V^T$ gives the provably best k-rank approximation of S.

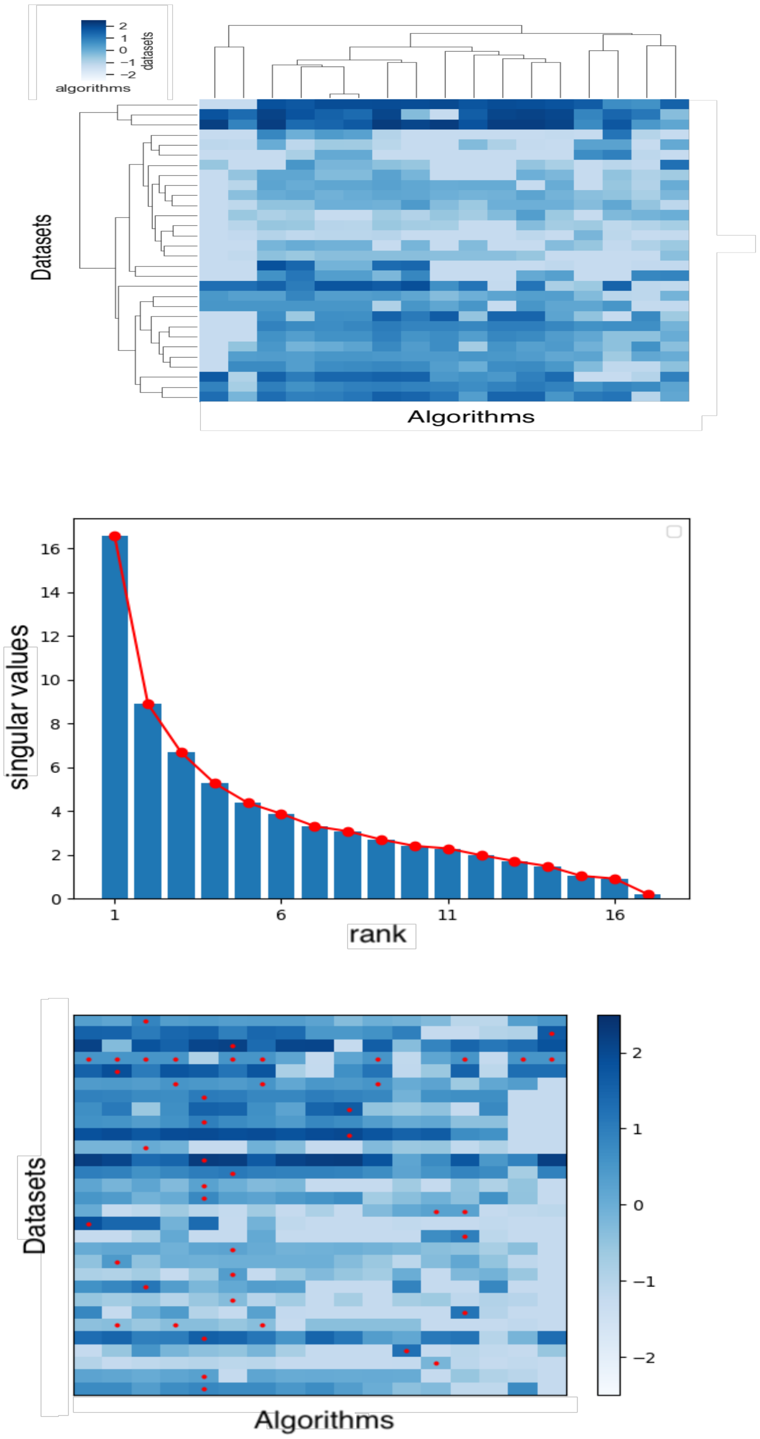


Fig. 2.4 **AutoML meta-dataset:** S_{AutoML} after global normalization (*i.e.* we subtracted the global mean and divided by the global standard deviation). TOP: Two-way hierarchical clustering based on normalized S_{AutoML}, in which both datasets (rows) and algorithms (columns) are arranged so that similar items are grouped together. MIDDLE: Spectrum of singular values of normalized S_{AutoML}. BOTTOM: Top ranking algorithms of normalized S_{AutoML}, where only algorithms (columns) are arranged based on their medians over datasets (from highest to lowest); then, for each dataset, the algorithms with maximum score values are marked with a red dot.

Text, speech, and video processing tasks were included in the challenge, but not in their native variable length representations.

- **Algorithms applied to the AutoML challenge (columns of S_{AutoML})**

  Following the AutoML challenge, we designed a challenge "re-match" called "Beat auto-sklearn"[7], whose purpose was to stimulate the community to try to beat the winners of the first challenge. This encouraged the *AAD Freiburg* team, creator of the winning entry *auto-sklearn* (a wrapper around the well-known machine learning scikit-learn library [87]), to exhibit their programmatic interface to scikit-learn (a.k.a. sklearn). In this way, it became easy to decouple their hyper-parameter optimization (hyperopt) algorithm from its interface to scikit-learn. This allowed us, in particular, to test other hyper-parameter optimization strategies (using the same model space) or to optimize the hyper-parameters of single models with the auto-sklearn hyperopt engine.

  We applied various types of learning algorithms to the 30 AutoML datasets to obtain the performance on held-out test sets. These algorithms pertain to two families:

  - **Winner solutions:** The code of winning entries (in at least some of the challenge rounds) was run systematically on all 30 datasets [8], because some participants entered late or abandoned early, we did not have performances for all methods on all datasets.
  - **Basic scikit-learn learning algorithms:** Many participants used the scikit-learn (sklearn) package, including the winning group AAD Freiburg, which produced the auto-sklearn software. We used the auto-sklearn API to conduct post-challenge systematic studies of the effectiveness of hyper-parameter optimization. We compared the performances obtained with default hyper-parameter settings in scikit-learn and with hyper-parameters optimized with auto-sklearn[9], both within the time budgets as imposed during the challenge, for four "representative" basic methods: k-nearest neighbors (KNN), naive Bayes (NB), Random Forest (RF), and a linear model trained with stochastic gradient descent (SGD-linear[10]).

  The metrics used to measure the performance of algorithms are BAC (for classification tasks) / R2 (for regression tasks), they produce values between 0 and

[7] The challenge is no longer running, but we still have the Codalab worksheet at https://worksheets.codalab.org/worksheets/0x107449520d9c48aaaceef240d557ba88.

[8] The public winner codes can be found at http://automl.chalearn.org/.

[9] We use scikit-klearn 0.16.1 to mimic the challenge environment, and auto-sklearn 0.4.0.

[10] We set the loss of SGD to be 'log' in scikit-learn for these experiments.

1 (one is best). The missing values (performance of algorithms aborted due to execution time constraints) were replaced by 0. This first meta-learning matrix S_{AutoML} took weeks of human effort to generate, especially adapting and debugging scikit-learn implementation to meet different tasks. The experiments were run on a x86_64 machine with 8 CPU. The time limit allocated to each task was the same as in the challenge. The Code and results for these post-challenge studies are at Github. We have also built a docker image lisesun / codalab_automl2016 for reproducibility purposes.

- **Difficulties encountered with the AutoML challenge meta-dataset**: The foremost difficulty we faced when generating these meta-learning matrices is the failure of algorithms when applying them on large and/or sparse datasets, due to the limitation of computational resources. This resulted in many missing values in the performance matrix S_{AutoML}, which, for practical purposes, are replaced by 0. We started investigating this issue by comparing various KNN implementations. Specifically, we compared the sklearn-KNN implementation to that of FLANN [81]. on the AutoML 2015-2016 Round 0 datasets. The results in Figure 2.5 [11] show that FLANN is generally faster than scikit-learn.

  One interesting way in which algorithms can be evaluated is to monitor their *learning curve* (performance progress as a function of time) rather than just their final performance at the end of a fixed time budget. The Area Under the Learning curve is indeed the metric used in the more recent AutoDL challenge http://autodl.chalearn.org, in which we made contributions at the design level. We attempted to generate learning curves for all datasets and all learning algorithms studied in the first AutoML challenge. Some algorithms lend themselves well to generating learning curves. For example, ensemble methods such as Random Forests (RF) or boosted ensembles of trees (Gradient boosting) allow us to increase progressively the number of trees. Also, algorithms based on stochastic gradient descent keep cycling over all training examples and can be interrupted and re-started easily to generate learning curves. But, for many algorithms (*e.g.* KNN or SVM) one needs to wait until the model is fully trained to get predictions and no further improvement is gained afterwards. This leads to learning curves that are just step-functions. Conceivably, such algorithms could be re-written to generate smooth learning curves, but we did not have time to undertake such endeavor.

  As a result of these difficulties, we focused on the "fixed time" learning setting in which algorithms must be executed (for training and testing) within a fixed

[11] This experiment is run on a x86_64 machine with 8 CPU.

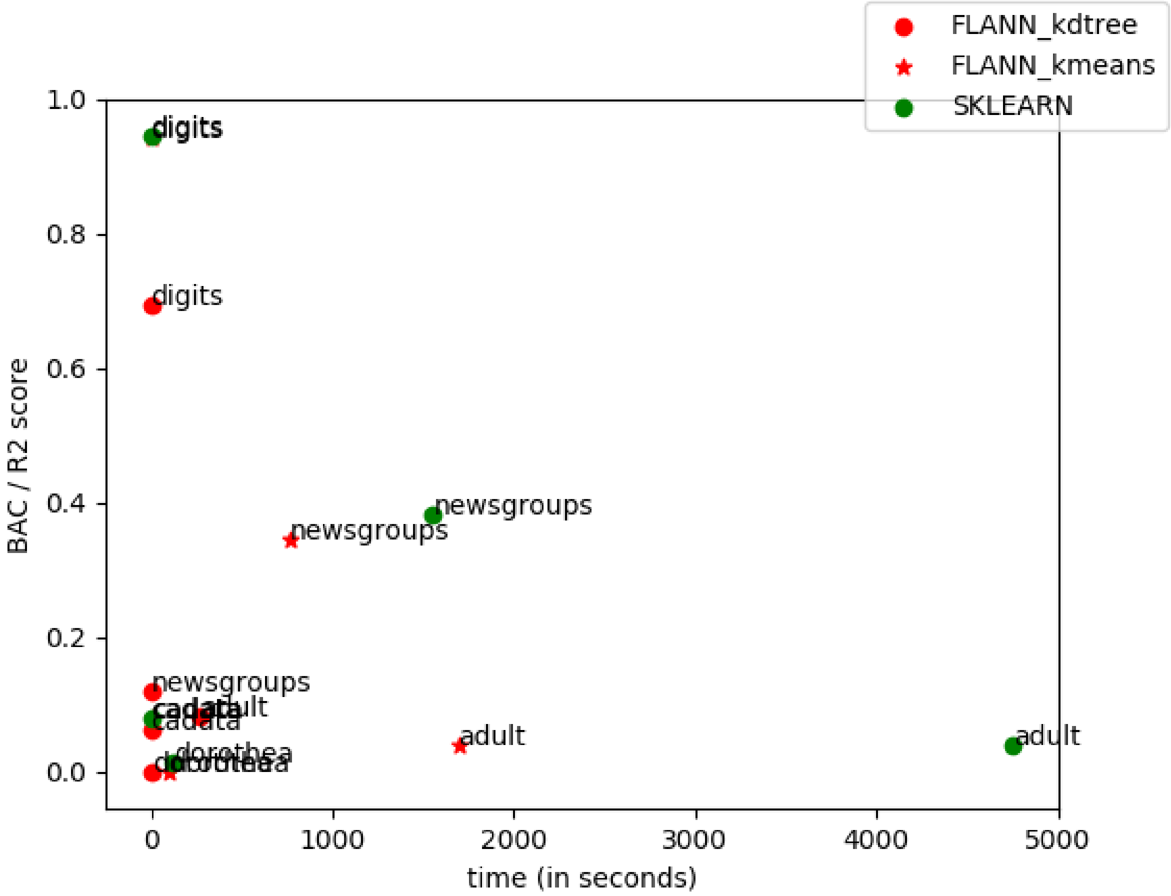


Fig. 2.5 **Round 0 datasets by sklearn-KNN and FLANN-KNN.** X axis is the total time of fitting plus predicting. Y axis is the performance expressed using BAC / R2 score for classification / regression tasks. k=5 in both cases. The KNN algorithm used to compute the nearest neighbors in sklearn is set to 'auto', i.e. sklearn can decide the most appropriate algorithm among {ball-tree, kd-tree, brute force} based on the dataset. We can see that FLANN is generally faster than sklearn, especially datasets with many features like newsgroups or many samples like adult, but performance sometimes degrades.

time budget, as opposed to the "any-time" learning setting in which algorithms can be stopped at any time. In the former case, the metric of success is the test set performance at the end of the time budget whereas in the latter case it is the area under the learning curve.

- **Meta-data for the AutoML challenge:** To perform the meta-learning experiments in Section 2.1, we have recomputed, for each of the 30 datasets, the meta-features implemented in [37, 35, 38][12]. Here is the full list of used meta-features, including the landmarks excluded in experiments in Section 2.1, but used in our later Reinforcement learning experiments in Section 2.3.2 and Section 6.3.1.
  - ClassProbabilityMin $= min_{i=1...n}(p(Class_i)) = min_{i=1...n}(\frac{NumberOfInstances\_Class_i}{TotleNumberOfInstances})$
  - ClassProbabilityMax $= max_{i=1...n}(p(Class_i)) = max_{i=1...n}(\frac{NumberOfInstances\_Class_i}{TotleNumberOfInstances})$
  - ClassEntropy $= mean(-\sum_{i=1}^{n} p(Class_i)ln(p(Class_i)))$ where $p(Class_i)$ is the probability of having an instance of Class_i
  - ClassOccurences = number of examples for each class
  - ClassProbabilityMean = mean($\frac{ClassOcurrences}{NumberOfClasses}$)
  - ClassProbabilitySTD = std($\frac{ClassOcurrences}{NumberOfClasses}$)
  - DatasetRatio $= \frac{NumberOfFeatures}{NumberOfInstances}$
  - InverseDatasetRatio $= \frac{NumberOfInstances}{NumberOfFeatures}$
  - LogInverseDatasetRatio $= \log(DatasetRatio)$
  - Landmark[Some_Model]: accuracy of [Some_Model] applied on dataset.
  - LandmarkDecisionNodeLearner & LandmarkRandomNodeLearner: Both are decision tree with max_depth=1. 'DecisionNode' considers all features when looking for best split, and 'RandomNode' considers only 1 feature, where comes the term 'random'.
  - Skewnesses: Skewness of each numerical features. Skewness measures the symmetry of a distribution. A skewness value > 0 means that there is more weight in the left tail of the distribution. Computed by scipy.stats.skew.
  - SkewnessMax / SkewnessMin / SkewnessMean / SkewnessSTD: max / min / mean / std over skewness of all features.

[12] Kurtosis, Skewness, KurtosisPCA and SkewnessPCA are intermediate metafeatures used to calculate some other metafeatures

- NumSymbols: Sizes of categorical features: for each categorical feature, compute its size (number of values in the category).
- SymbolsMax / SymbolsMin / SymbolsMean / SymbolsSTD / SymbolsSum = max / min / mean / std / sum over NumSymbols
- NumberOfCategoricalFeatures: Number of categorical features.
- NumberOfNumericFeatures: Number of numerical features
- RatioNumericalToNominal = $\frac{NumberOfNumericFeatures}{NumberOfCategoricalFeatures}$
- RatioNominalToNumerical = $\frac{NumberOfCategoricalFeatures}{NumberOfNumericFeatures}$
- Kurtosis = Fourth central moment divided by the square of the variance = $\frac{E[(x_i-E[x_i])^4]}{[E[(x_i-E[x_i])^4]]^2}$ where x_i is the i-th feature. Computed using scipy.stats.kurtosis.
- KurtosisMax / KurtosisMin / KurtosisMean / KurtosisSTD = max / min / mean / std of kurtosis over all features
- PCAKurtosis: Transform data by PCA, then compute the kurtosis
- NumberOfInstances = Number of examples
- NumberOfFeatures = Number of features
- NumberOfClasses = Number of classes
- LogNumberOfFeatures = $\log(NumberOfFeatures)$
- LogNumberOfInstances = $\log(NumberOfInstances)$
- MissingValues: Boolean matrix of dim (NumberOfInstances , NumberOfFeatures), indicating if an element of is a missing value.
- NumberOfMissingValues: Total number of missing value
- NumberOfInstancesWithMissingValues: Number of examples containing missing values.
- NumberOfFeaturesWithMissingValues: Number of features containing missing values.
- PCA: PCA decomposition of data.
- PCAFractionOfComponentsFor95PercentVariance: Fraction of PCA components explaining 95% of variance of the data.
- PCAKurtosisFirstPC: Kurtosis of the first PCA component.
- PCASkewnessFirstPC: Skewness of the first PCA component.

Rnd	DATASET	Task	Metric	Time	C	Cbal	Sparse	Miss	Cat	Irr	Pte	Pva	Ptr	N	Ptr/N
0	1 ADULT 0	multilabel	F1	300	3	1	0.16	0.011	1	0.5	9768	4884	34190	24	1424.58
0	2 CADATA 1	regression	R2	200	0	NaN	0	0	0	0.5	10640	5000	5000	16	312.5
0	3 DIGITS 2	multiclass	BAC	300	10	1	0.42	0	0	0.5	35000	20000	15000	1568	9.57
0	4 DOROTHEA 3	binary	AUC	100	2	0.46	0.99	0	0	0.5	800	350	800	100000	0.01
0	5 NEWSGROUPS 4	multiclass	PAC	300	20	1	1	0	0	0	3755	1877	13142	61188	0.21
1	1 CHRISTINE 5	binary	BAC	1200	2	1	0.071	0	0	0.5	2084	834	5418	1636	3.31
1	2 JASMINE 6	binary	BAC	1200	2	1	0.78	0	0	0.5	1756	526	2984	144	20.72
1	3 MADELINE 7	binary	BAC	1200	2	1	1.2e-06	0	0	0.92	3240	1080	3140	259	12.12
1	4 PHILIPPINE 8	binary	BAC	1200	2	1	0.0012	0	0	0.5	4664	1166	5832	308	18.94
1	5 SYLVINE 9	binary	BAC	1200	2	1	0.01	0	0	0.5	10244	5124	5124	20	256.2
2	1 ALBERT 10	binary	F1	1200	2	1	0.049	0.14	1	0.5	51048	25526	425240	78	5451.79
2	2 DILBERT 11	multiclass	PAC	1200	5	1	0	0	0	0.16	9720	4860	10000	2000	5
2	3 FABERT 12	multiclass	PAC	1200	7	0.96	0.99	0	0	0.5	2354	1177	8237	800	10.3
2	4 ROBERT 13	multiclass	BAC	1200	10	1	0.01	0	0	0	5000	2000	10000	7200	1.39
2	5 VOLKERT 14	multiclass	PAC	1200	10	0.89	0.34	0	0	0	7000	3500	58310	180	323.94
3	1 ALEXIS 15	multilabel	AUC	1200	18	0.92	0.98	0	0	0	15569	7784	54491	5000	10.9
3	2 DIONIS 16	multiclass	BAC	1200	355	1	0.11	0	0	0	12000	6000	416188	60	6936.47
3	3 GRIGORIS 17	multilabel	AUC	1200	91	0.87	1	0	0	0	9920	6486	45400	301561	0.15
3	4 JANNIS 18	multiclass	BAC	1200	4	0.8	7.3e-05	0	0	0.5	9851	4926	83733	54	1550.61
3	5 WALLIS 19	multiclass	AUC	1200	11	0.91	1	0	0	0	8196	4098	10000	193731	0.05
4	1 EVITA 20	binary	AUC	1200	2	0.21	0.91	0	0	0.46	14000	8000	20000	3000	6.67
4	2 FLORA 21	regression	ABS	1200	0	NaN	0.99	0	0	0.25	2000	2000	15000	200000	0.08
4	3 HELENA 22	multiclass	BAC	1200	100	0.9	6e-05	0	0	0	18628	9314	65196	27	2414.67
4	4 TANIA 23	multilabel	PAC	1200	95	0.79	1	0	0	0	44635	22514	157599	47236	3.34
4	5 YOLANDA 24	regression	R2	1200	0	NaN	1e-07	0	0	0.1	30000	30000	400000	100	4000
5	1 ARTURO 25	multiclass	F1	1200	20	1	0.82	0	0	0.5	2733	1366	9565	400	23.91
5	2 CARLO 26	binary	PAC	1200	2	0.097	0.0027	0	0	0.5	10000	10000	50000	1070	46.73
5	3 MARCO 27	multilabel	AUC	1200	24	0.76	0.99	0	0	0	20482	20482	163860	15299	10.71
5	4 PABLO 28	regression	ABS	1200	0	NaN	0.11	0	0	0.5	23565	23565	188524	120	1571.03
5	5 WALDO 29	multiclass	BAC	1200	4	1	0.029	0	1	0.5	2430	2430	19439	270	72

Table 2.2 **Datasets of the 2015/2016 AutoML challenge.** C=number of classes. Cbal=class balance. Sparse=sparsity. Miss=fraction of missing values. Cat=categorical variables. Irr=fraction of irrelevant variables. Pte, Pva, Ptr=number of examples of the test, validation, and training sets, respectively. N=number of features. Ptr/N=aspect ratio of the dataset. The number in red next to dataset names represents the numbering in the figure on AutoML meta-dataset 2.4.

StatLog meta-dataset

The StatLog [62] meta-dataset is a public dataset from the UCI database (http://archive.ics.uci.edu/ml/datasets/StatLog+(australian+credit+approval). We took the square root of the performances to equalize the distribution of scores (avoid a very long distribution tail). Figure 2.6 shows the visualization of the StatLog meta-dataset. We see the hierarchical clustering (TOP) is homogeneous both in datasets and algorithms. It's challenging to find sub-sets of algorithms / datasets. The median-arranged matrix (BOTTOM) also shows no specific algorithms outperform on multiple datasets.

OpenML meta-dataset

This meta-dataset is a subset of the OpenML platform [115] (https://www.openml.org/) and used in [75]. Figure 2.7 shows the visualization of the OpenML meta-dataset. We see clear block structures in the hierarchical clustering (TOP) on both dataset and algorithm directions, suggesting that sub-sets of datasets and algorithms can be found, this is coherent with the median-arranged matrix (BOTTOM) where a large number of algorithms are good at multiple datasets, and many datasets have more than one best algorithms. In this sense, the OpenML meta-dataset, compared to others, is more suitable for meta-learning.

2.3.2 Synthetic and toy meta-datasets: Matrix factorization, CEP, MNIST patches

This section describes three synthetic meta-datasets: Artificial constructed from a matrix factorization, CEP generated from the Cause-Effect Pairs challenge, and MNIST patches from the MNIST dataset.

Artificial meta-dataset

The Artificial meta-dataset is constructed from a matrix factorization to create a simple benchmark we understand well, which allows us to easily vary the problem difficulty. [13] More precisely, the artificial matrix $S_Artificial$ is obtained as a product of three matrices $U\Sigma V$, U and V being orthogonal matrices and Σ a diagonal matrix of "singular values", whose spectrum is chosen to be exponentially decreasing, with $\Sigma_{ii} = exp(-\beta i)$, $\beta = 100$ in our experiments. Figure 2.8 shows the visualization of the Artificial meta-dataset.

[13] The Code for generating the artificial meta-dataset is at https://github.com/LishengSun/ActiveMetaLearn/blob/master/src/utils/make_artificial_matrix.py.

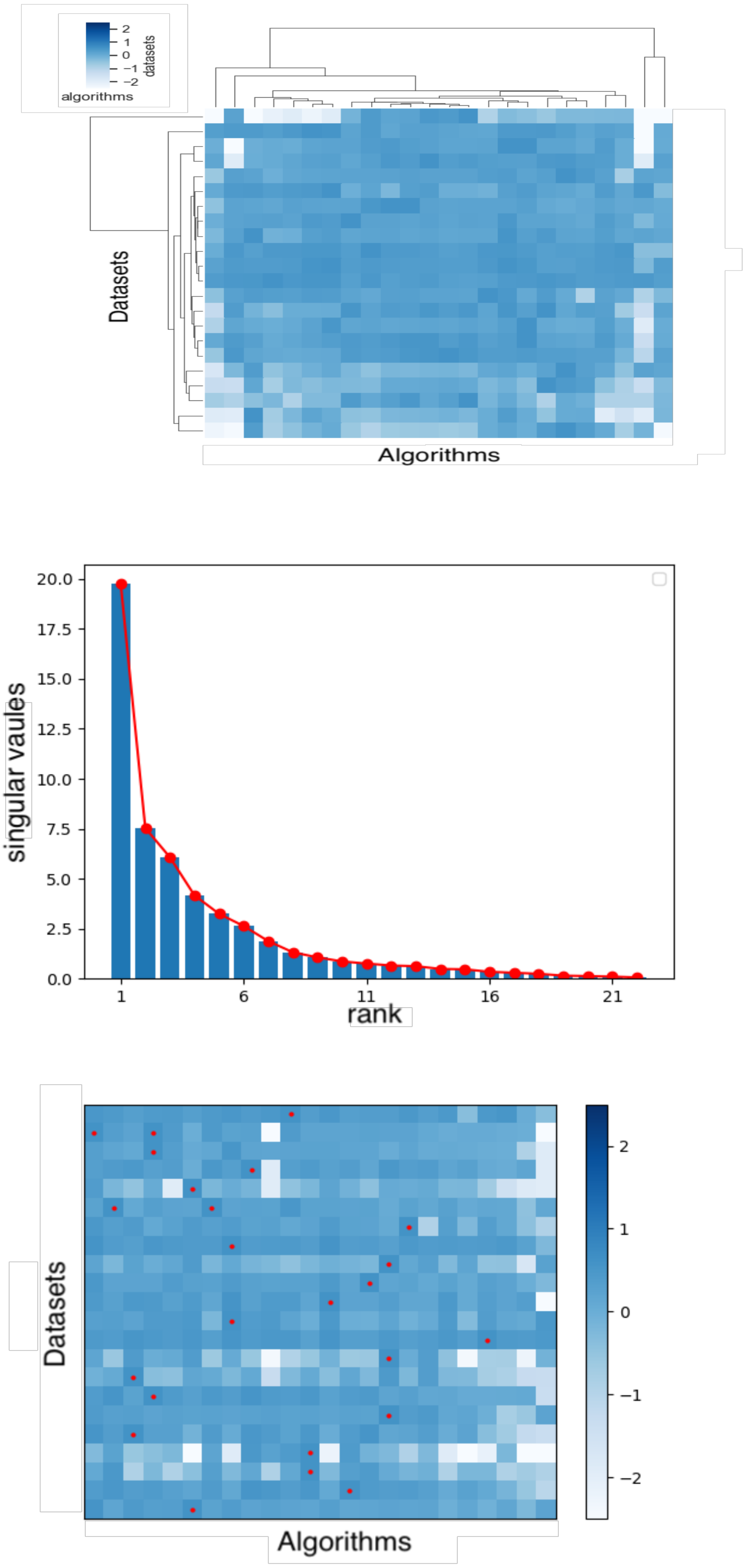


Fig. 2.6 **StatLog meta-dataset:** $S_{StatLog}$ after global normalization. TOP: Two-way hierarchical clustering based on normalized $S_{StatLog}$. MIDDLE: Spectrum of singular values of normalized $S_{StatLog}$. BOTTOM: Top ranking algorithms of normalized $S_{StatLog}$.

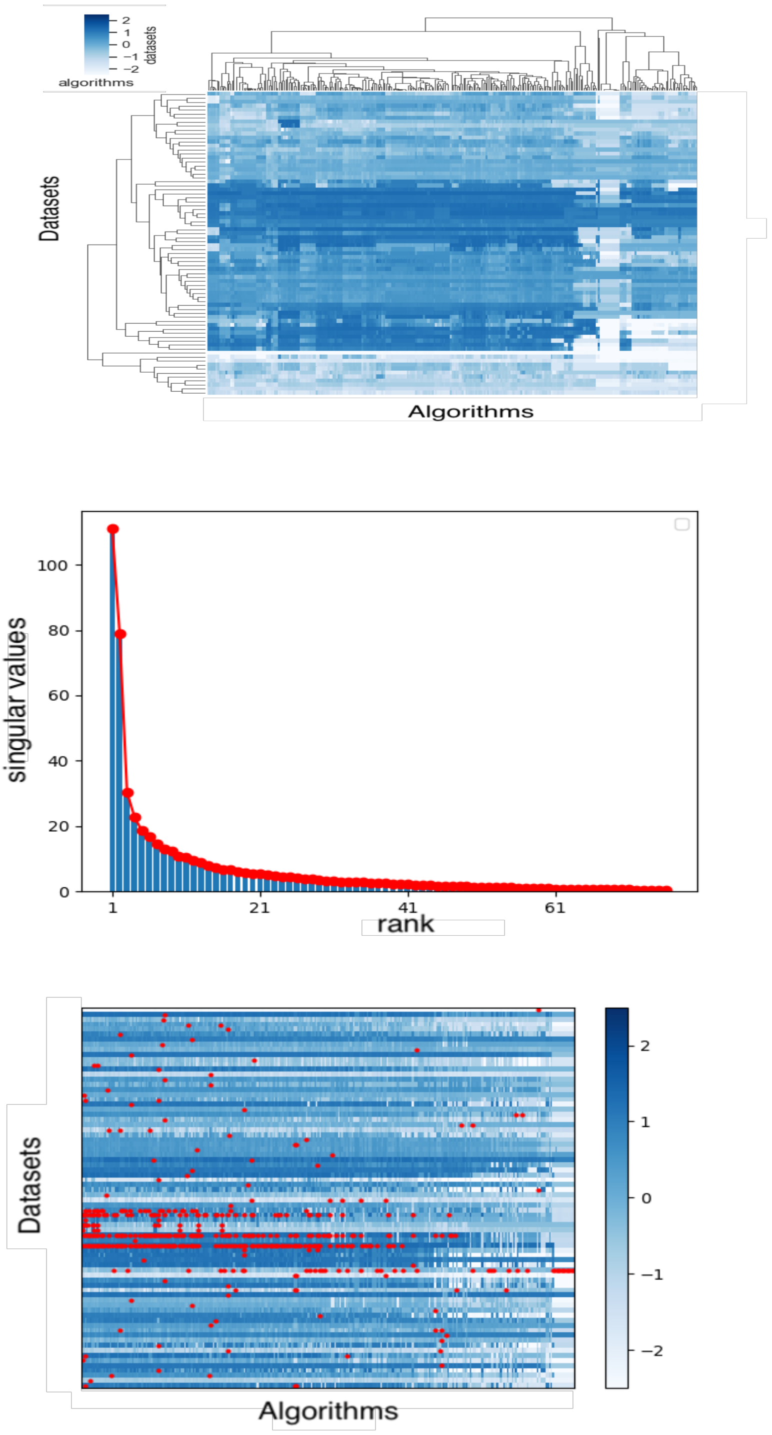


Fig. 2.7 **OpenML meta-dataset:** S_{OpenML} after global normalization. TOP: Tow-way hierarchical clustering based on normalized S_{OpenML}. MIDDLE: Spectrum of singular values of normalized S_{OpenML} . BOTTOM: Top ranking algorithms of normalized S_{OpenML}.

Compared to other real-world meta-datasets (e.g. Figure 2.4), we notice that a block structure is more apparent in the hierarchical clustering (TOP), indicating that it should be easier to find subsets of algorithms, which are better on subsets of tasks. However, surprisingly, the BOTTOM picture indicates that only a few algorithms (e.g. algorithm #3 and #13, representing 2 algorithm families that are good at different subsets of tasks.) seem to be better at a large number of tasks. This is because only the absolute maximal values are red-marked in each dataset, while multiple algorithms have performance just slightly lower than the maximum (e.g. algorithm #2 compared to #3, and algorithm #7 compared to #13 for many tasks).

CEP meta-dataset

The CEP meta-dataset was constructed using data from the cause-effect pair challenge (https://www.kdnuggets.com/2013/04/chalearn-cause-effect-pairs-challenge.html) without making any use of underlying causal relationship. The meat-dataset includes samples of pairs of variables (CE pairs), continuous, binary, or categorical. We used a subset of these pairs to create univariate "machine learning tasks": classification or regression problems using a SINGLE input variable. Thus each little task/dataset in our meta problem is one of the CE pairs. Our experiments thus far have been limited to one continuous input variable and one categorical output variable (classification problem), ignoring the causal direction. The present CEP meta-dataset consists of 8608 classification datasets, 163 algorithms and 16 meta-features. The performance of algorithms are measured by BAC. The evaluation time of each (dataset, algorithm) pair is also recorded, giving an extra time matrix S_{CEP_time}. Figure 2.9 shows the visualization both on performance and time matrix. We see that block structures are more pronounced in datasets in the performance matrix, but in algorithms when looking at time matrix. But the overall structure is rather homogeneous, high similarity presents both in datasets and algorithms. However, thanks to a large number of datasets, the CEP meta-dataset provides a suitable learning source for a simple meta-learning problem.

- **Algorithms applied to the CEP datasets**: To ensure the similarity in algorithms so that meta-learning is possible, 163 basic learning algorithms were selected from diverse families: "KNN", "DecisionTree", "RandomForest", "GradientBoostingTree", "AdaBoost", "lSVM", "kSVM", "Logit", "Perceptron", "GaussianNaiveBayes", "MultiLayerPerceptron", "ExtraTrees". All algorithms used are from scikit-learn. For each hyper-parameter in these algorithms, some values are sampled in an ad-hoc fashion, then each possible combination of

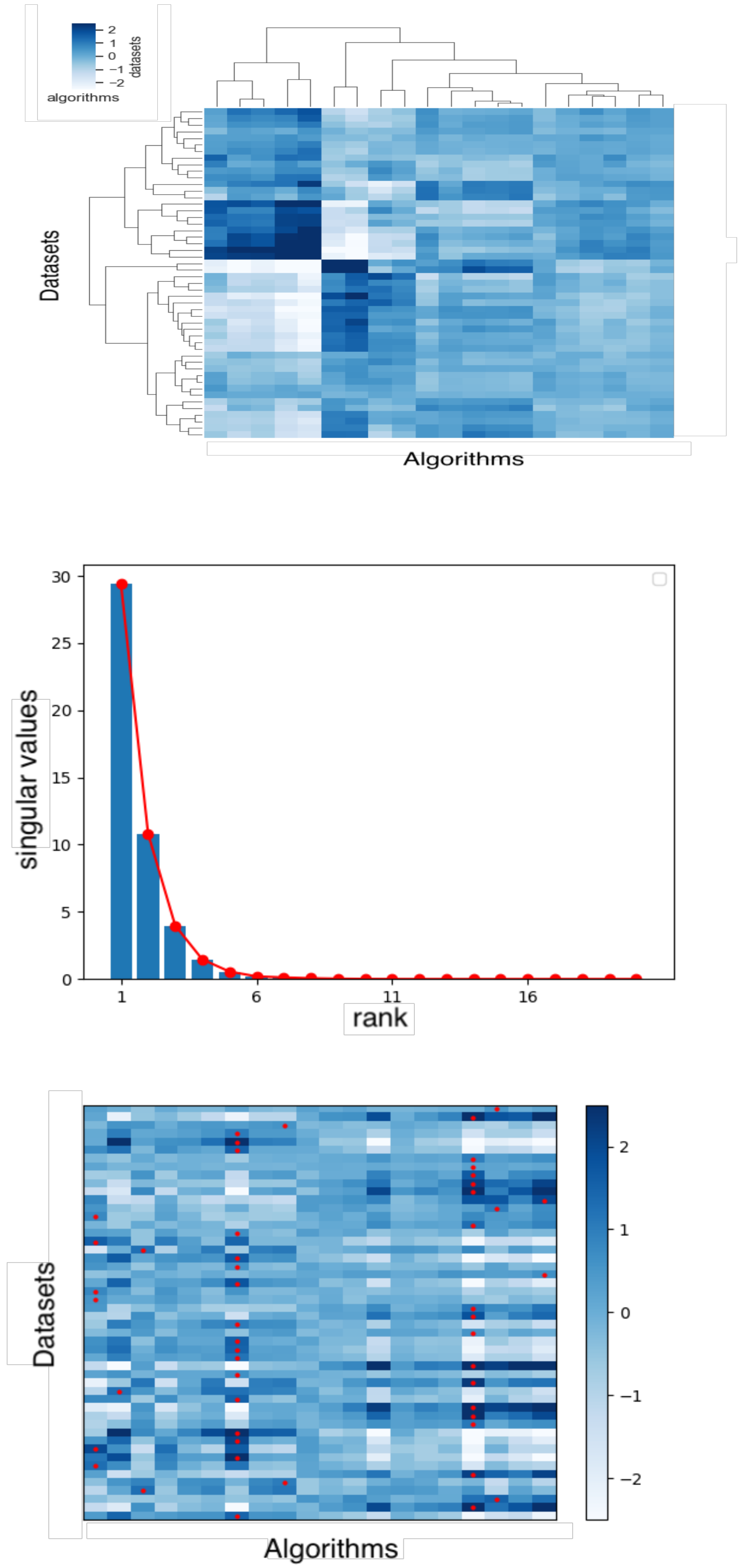


Fig. 2.8 **Artificial meta-dataset:** $S_{Artificial}$ after global normalization. TOP: Two-way hierarchical clustering based on normalized $S_{Artificial}$. MIDDLE: Spectrum of singular values of normalized $S_{Artificial}$. BOTTOM: Top ranking algorithms of normalized $S_{Artificial}$.

these hyper-parameters form one algorithm. Failures of simulation are marked as 0 (worst BAC) in the matrix of performance and 1 second in the time matrix.

- **Meta-features of CEP datasets** : 16 Meta-features are also computed for each dataset in CEP to help tackle the cold start problem in meta-learning when using RL in Chapter 6: 'NumberOfInstances', 'LogNumberOfInstances', 'NumberOfClasses', 'ClassProbabilityMin', 'ClassProbabilityMax', 'ClassProbabilityMean', 'ClassProbabilitySTD', 'Kurtosisses', 'Skewnesses', 'ClassEntropy', 'LandmarkLDA', 'LandmarkNaiveBayes', 'LandmarkDecisionTree', 'LandmarkDecisionNodeLearner', 'LandmarkRandomNodeLearner', 'Landmark1NN'. The definition of these meta-features is given in Section 2.3.1.

MNIST-patch meta-dataset

In Chapter 6, we will use the MNIST dataset [68] to create an exploration game, which is a toy example before addressing the more difficult meta-learning problem. This section will first describe this meta-dataset(for more information, please refer to Section 6.2.2). The original MNIST dataset is a handwritten digit database largely used to train various image related machine learning systems. It consists of a training set ($60,000$ images) and a test set ($10,000$ images). The digit classes range from 0 to 9.

The MNIST dataset first caught our attention when it was presented in the Natural Image game in [122] where it was used to create a game environment in which an RL agent is trained to efficiently reveal a masked MNIST image and predict its label. We then drew the link between a MNIST image and a machine learning dataset, and realized that if an agent can be trained to navigate in the pixel space, under additional assumptions, it should be able to train it to navigate in the algorithm performance space, *i.e.* our meta dataset matrices S. We then created a meta-dataset where each row is a flatten MNIST image. However, if we simply flatten the MNIST image to a $32 \times 32 = 1024$ vector [14], the searching space will be too large for the agent to learn efficiently, additionally, there are many equal pixel values in a image, making this space very redundant. We thus use a square window of size 5 to average the neighboring pixels, this is interpreted as a patch brightness, The notion of patch reduces the size of the vector to 49. This $(70000, 49)$ [15] matrix is our final MNIST-patch meta-dataset, it is similar to other meta-datasets we present so far, where an image is equivalent to a

[14] One original MNIST image is of size 28×28. We used the version of [122] where images are transformed to size of 32×32 for algorithm compatibility.

[15] 60,000 images from train set and 10,000 from test set, each of them has 49 patches and therefore 49 brightness values.

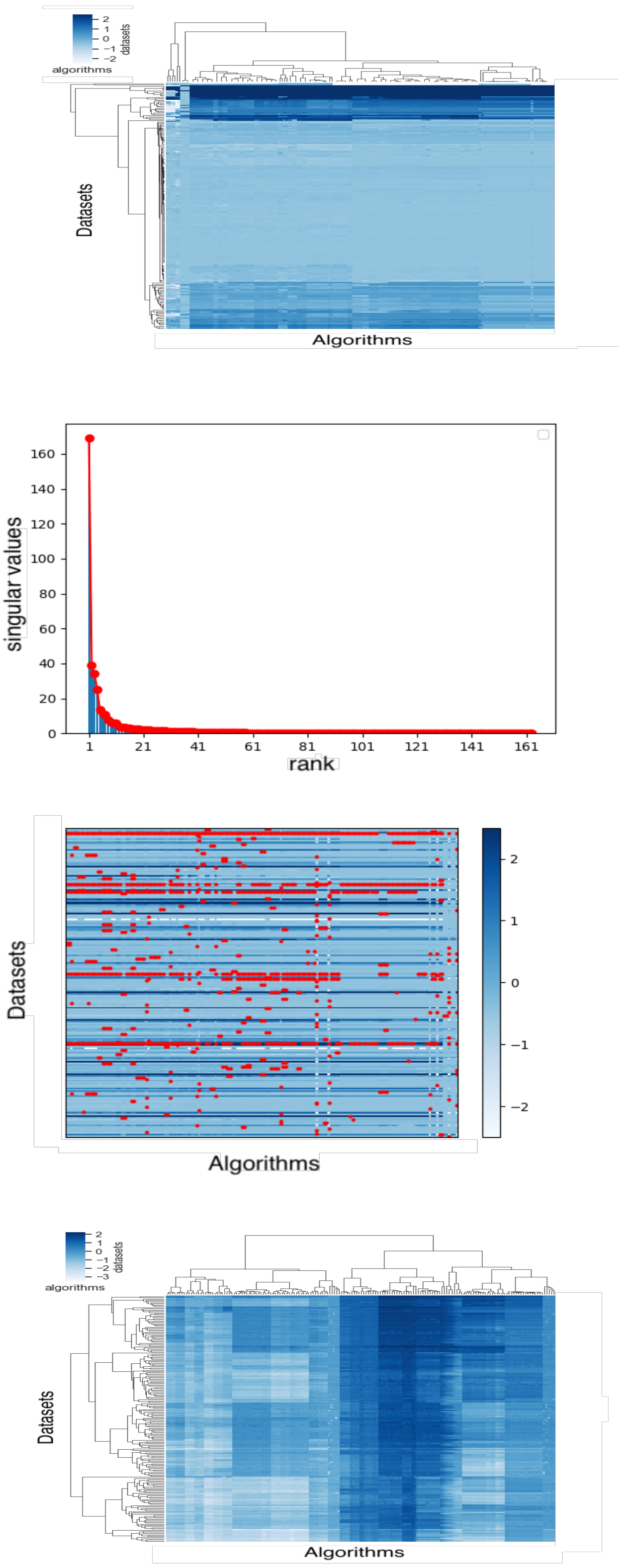


Fig. 2.9 **200 randomly sampled datasets from CEP meta-dataset, which contains 2 matrices of same dimension: S_{CEP_perf} for algorithm performances and S_{CEP_time} for algorithm computational times.** 1st panel: Tow-way hierarchical clustering based on normalized S_{CEP_perf}. 2nd panel: Spectrum of singular values of normalized S_{CEP_perf} . 3rd panel: Top ranking algorithms in normalized S_{CEP_perf}. 4th panel: Tow-way hierarchical clustering based on S_{CEP_time} after taking natural logarithm.

dataset, and a patch brightness value is equivalent to an algorithm performance value. Figure 2.10 shows a sub-sample of 200 images randomly (without replacement) drawn from the whole MNIST-patch meta-data train set. A clear block structure presents in columns (*i.e.* the direction of patch), this is not surprising because most MNIST images have digit in the center of the image, which corresponds to a common bright area. However, there is no clear block structure in rows (*i.e.* the direction of images), meaning that the images are rather i.i.d. This observation shows that training an agent to find brightest patches across different digits is possible [16].

2.4 Summary of experimental setting chapter

This chapter presented the experimental setting used in the remainder of this thesis. The analysis of the AutoML challenge (2015-2016), which was summarized at the beginning of the chapter, was a pivotal element in outlining the importance of meta-learning. The AutoML challenge datasets were used to create a novel meta-learning dataset. Several other meta-learning datasets were added to create a benchmark of meta-learning, including datasets borrowed from the literature (StatLog and OpenML), and synthetic/toy datasets (simple matrix factorization, CEP, MNIST patches).

Thus, the main contribution of this chapter is a collection meta-datasets, which are used in this thesis and are made available (at https://github.com/LishengSun/REVEAL/upload/master/env/meta_learning_env/meta_learning_matrices) to the research community to further explore the meta-learning problem. As my attention moved from standard Hyper-parameter selection to meta-learning, these meta-datasets will serve as meta-learning sources and will be extensively used in Chapter 5 and 6, where the framework of meta-learning as a Markov Decision Process (MDP) is introduced.

[16] It is possible because of the way in which people hand-write: they usually put more pressure at the beginning of writing when the pen hits the paper, which results in the brightest patches. This is a common fact across different digits

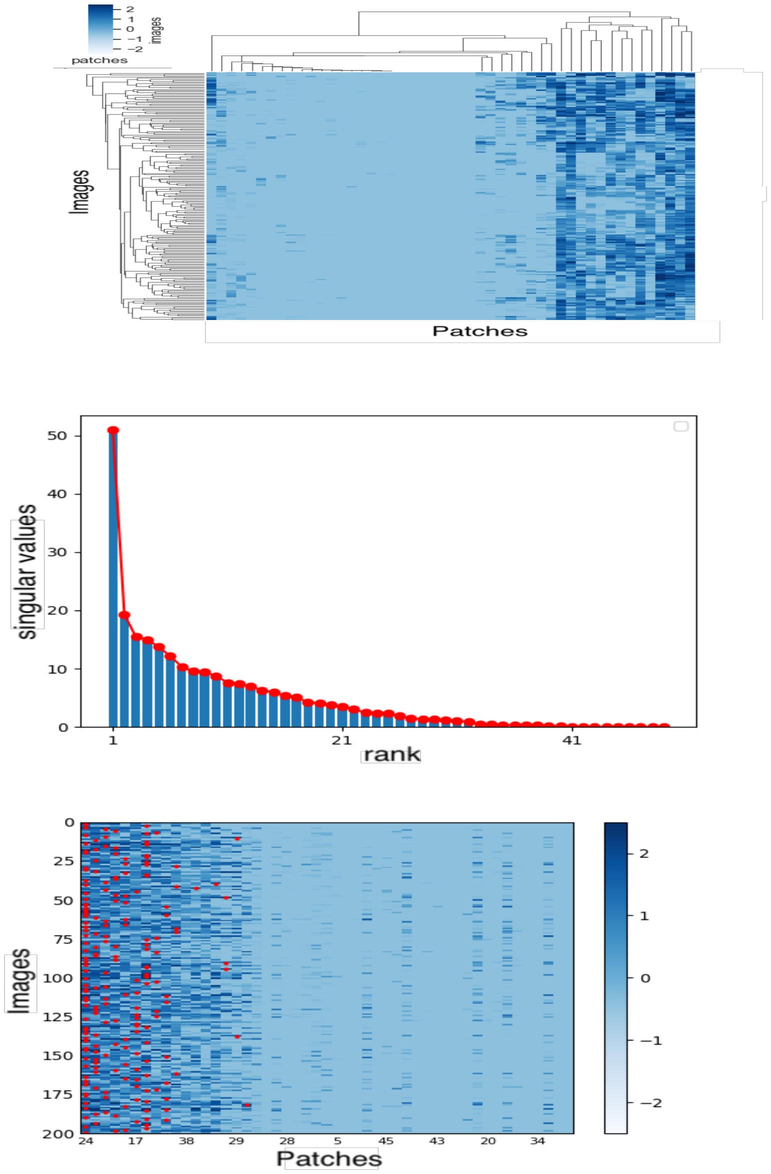


Fig. 2.10 **200 random sampled images from MNIST-patch meta-dataset**: This form a 200 × 49 matrix $S_{MNIST_{patch}}$, in which one row is one flatten MNIST image, one column is one patch, the value is the brightness. TOP: Two-way hierarchical clustering based on $S_{MNIST_{patch}}$. MIDDLE: Spectrum of singular values of $S_{MNIST_{patch}}$. BOTTOM: Top ranking patches of $S_{MNIST_{patch}}$.

Chapter 3

Mathematical statement of the AutoML problem as a MDP

In this chapter, we formulate the problem of algorithm (or model) selection, which is at the hart of the AutoML problem, as a Markov Decision Process (MDP). This will provide us with a formal framework to address the problem of hyper-parameter selection and meta-learning in a principled way. In a MDP, the system under study is a Markov process, hence has a finite number of states and the next state is determined (via state transition probabilities) from (a finite number of) recent past states and actions. In our problem of interest, **states** encode our knowledge of the performance of algorithms on given tasks (the performance of machine learning methods on given datasets); **actions** encode our next move in "information space", *i.e.* which algorithm (learning machine) we decide to run on a particular dataset to *reveal* it performance (AutoML MDPs actually belong to a larger family of problems, which we baptized REVEAL games). Our **goal** is to uncover as far as possible the best performing algorithm(s). To that end, we will define objective functions or "rewards" to guide our meta-learning algorithms. For example, an immediate reward (that can be used by reinforcement Learning algorithms) could be the improvement in algorithm performance.

3.1 Notations and problem setting

A Markov Decision Process (MDP) is a sequential decision making process. It is described by a 4-tuple (state s_t, action a_t, transition probability $p(s_{t+1}|s_t, a_t)$, reward r_{t+1}). In each time step, the process is in some state s_t, the decision maker, also called the **agent**, has access to information on s_t [1] that allows it to decide its action a_t, the process responds with a transition to a next state s_{t+1} drawn from a model $p(s_{t+1}|s_t,a_t)$ and a reward r_{t+1} to inform the agent about the quality of its action. The goal of the agent is to maximize the reward accumulated over time. Meta-learning can be formulated as a MDP, in the sense that the agent (human or any AutoML system), placed in some state (a partially filled row in the meta-dataset matrix *S*), is moving in that row (*i.e.* the algorithm performance space) to find the position of the highest value (*i.e.* the algorithm with highest performance). At each time step, the action of the agent is to choose one algorithm to evaluate. After each evaluation, the agent gains more information (one more value in matrix S): this is the next state. The improvement in evaluation score of the algorithm could play the role of reward. After seeing enough datasets (*i.e.* many rows, one at a time), the agent builds a prior knowledge on the algorithm space across datasets, this allows it to find more efficiently the best algorithm for a new dataset drawn from the same mother distribution, which is the underlying probability distribution from which the datasets (*i.e.* the rows in *S*) are sampled.

To address the problem of finding an optimal policy (a good agent in our MDP setting), it is helpful to view solutions to AutoML problems as algorithm recommender Systems (RS), seeking the algorithm best suited to a given dataset [99]. Two approaches exist: (1) Sequential Optimization that searches for the maximal performance on ONE particular dataset (thus one row in *S*). Examples include hyper-parameter optimization (Chapter 4) and Collaborative Filtering (Chapters 5); (2) Reinforcement Learning (RL) that searches for a policy that leads to the maximal performance on ANY task (Chapter 6). The advantage of RL compared to Sequential Optimization is the learned policy can be applied on new datasets.

Compared to standard MDPs, the AutoML MDP has two specific features: (1) it has finite time horizon, i.e. we limit the number of decision making steps in each episode; (2) an action only gains more information about the state, it has no effect on the data generating process. We have identified a whole family of games that share these same specifications, which we call **REVEAL** games. Section 3.2 gives the definition and examples of REVEAL games, which allowed us to test various RL algorithms and visualize their behavior. Section 3.3 explains how AutoML can be formulated as a REVEAL game.

[1] If the agent has access to partial information about the state s_t, the process is a Partially Observable Markov Decision Process (POMDP). In this chapter, we assume that the agent has access to full information about s_t and restrict the discussion to a MDP framework.

3.2 A game of REVEAL

To project the above formulation into a realistic setting, let's imagine a REVEAL game, in which the game board is filled with face-down information cards. A game player picks a card and flips it to reveal the hidden information to earn a reward. The overall goal is to accumulate rewards as quickly as possible. We now give a more formal definition of a REVEAL game.

3.2.1 REVEAL definition

Definition 3 (REVEAL game). *A REVEAL game is a MDP, fully defined by a 4-element tuple: $(S,A,p(S_{t+1}|S_t,a_t),R\|\Delta)$, where $\Delta = \{\beta,\tau,a_\bullet\}$ is the set of game parameters:*

- **State:** *S is a matrix of dimension $M \times D \times N$. It contains M channels, each of them is a $D \times N$ matrix encoding one type of information, e.g. D is for datasets and N for algorithms. The game starts at s_0 which is the empty S. s_t is the partially revealed matrix at time t defined by a triplet*

$$s_t = \begin{cases} \{i_k, j_k, S(i_k, j_k)\} \quad for\ k = 1,\dots,t-1 \quad if\ t < \tau\ and\ a_t \neq a_\bullet \\ s_\bullet \quad if\ t = \tau\ or\ a_t = a_\bullet \end{cases} \tag{3.1}$$

  *where (i_k, j_k) is revealed at time k with action a_k. The game terminates when the time t exceeds the **time budget** τ or when the action a_t hits the **goal** $a_\bullet$. The set of states is thus $\mathscr{S} = \{s_0, s_t(t = 1,\dots,\tau-1), s_\bullet\}$*

- **Action:** *The set of actions at time t is noted $\mathscr{A}_t$: .*

$$\mathscr{A}_t = \begin{cases} \mathscr{P} \quad if\ t < \tau\ and\ s_t \neq s_\bullet \\ \emptyset \quad otherwise \end{cases} \tag{3.2}$$

  - *When $t < \tau$ or when the goal is not achieved (i.e.$s_t \neq s_\bullet$), the agent can choose to reveal a position in S, this is noted by $\mathscr{P}$. It can be (but not necessary) the union of set of row indices $\mathscr{I}$ and column indices $\mathscr{J}$, i.e. $\mathscr{P} = \mathscr{I} \times \mathscr{J}$. In this case, $a_t = (i_t, j_t)$.*
  - *The game terminates when $t = \tau$ or when the goal is achieved, the action space is empty.*

- *$p(s_{t+1}|s_t,a_t)$ is the* **transition probability**. *If S is assumed pre-computed, the transition is deterministic given s_t and a_t :*

$$s_{t+1} = \begin{cases} s_t \bigcup \{i_t, j_t, S(i_t, j_t)\} & \text{if } t < \tau \text{ and } a_t \neq a_\bullet \\ s_\bullet \quad \text{otherwise} \end{cases} \tag{3.3}$$

- *R is the reward signal computed using the* **reward function** *$r(t)$, which contains a positive term $r_+(t) : \mathscr{S} \to \mathbb{R}$ representing how good to be in the current state s_t, and a negative term $r_-(t) : \mathscr{S} \to \mathbb{R}$ representing how much the agent actually pays for being in that state. β controls this* **performance-cost trade-off**.

$$r(t) = r_+(t) - \beta r_-(t) \quad \text{with} \;\; \beta > 0 \tag{3.4}$$

3.2.2 REVEAL game examples

To gain a better understanding of class of MDP problems that we called REVEAL games and thus guide our design of meta-learning algorithms, we created a suite of toy REVEAL games. In this section we describe a few. We parameterize each game using our previously described notation.

Battleship: REVEAL($\beta = 0, a_\bullet =$ ship locations)

[2] One of the simplest example is the famous game of “battleship”: “ships” are placed on a grid, hidden to the player (agent). The player must blindly aim at the position of ships to try to sink them. At each try, the player gains the information whether a ship (partially) occupies the targeted location, *i.e.* whether the player touched or sank a ship. The game is usually played with 2 players, each providing a puzzle to the other. Formally:
S = the game board. $\mathscr{A}$ = set of all locations on the game board. Goal = uncovering ship locations to destroy them faster than the opponent. If the game is not put under any time limit specified by the game designer, the time limit is bounded by the board game.

[2] $\beta = 0$ if we don’t “charge” the agent for revealing any position, it can be changed freely to suit the need of the game designer.

Reveal a MNIST image to predict its digit: REVEAL($\beta = 1, \tau = 100, a_\bullet =$ correct digit label)

This game is proposed in [122], the agent initially placed in the center of a fully masked MNIST digit image should learn to reveal that image progressively to predict its label within 100 time steps. The state is a 2-channel matrix S of dimension $2 \times 32 \times 32$ including the under being revealed 32×32 image and another 32×32 image encoding the agent's current location. The actions are close moves related to the current position: $\mathscr{P} = \{Left, Right, Up, Down\}$. After each move, the agent predicts the label, if the prediction is correct, the game board transits to $s_\bullet$ and the game terminates. [3] Since the game is also put under a time limit $\tau = 100$ steps , it terminates automatically when $t = \tau$ even the prediction is always wrong. Each action costs $r_- = -0.01$, only correct prediction is rewarded positively with $r_+ = 1$.

Reveal a MNIST image progressively to find the brightest patches: REVEAL($\beta = 0.5, \tau = 20, a_\bullet =$ positions of the brightest patches)

We have modified the setting of the previous game to make the problem more alike to AutoML. Now the goal of the agent is to reveal the image to find the brightest patches. States are still the $(2, 32, 32)$ matrix, action space is now all possible positions in the image but we don't require the agent to predict the label anymore. The game terminates automatically when one of the brightest patches is found or when the time budget exceeds τ, thus, $\mathscr{P} = \mathscr{I} \times \mathscr{J}$ during the game. $\tau = 20$ steps; The rewards is also modified to combine the performance and cost: $r_+(t) = mean(S[0, a_t])$, $\beta = 0.5$, $r_-(t) = -0.05$. In Chapter 6, we use RL to train the agent that successfully learns to navigate the the brightest patches in an unseen MNIST image. Figure 6.6 shows this agent performing one test episode.

3.3 AutoML as a REVEAL game

The AutoML problem itself can be viewed as a REVEAL game: given a sparse matrix S of dimension $M \times D \times N$ where $S_{m,i,j}$ is the value of channel m of algorithm j applied on dataset i, reveal more of its values by running more algorithms on datasets. Different channels may contain information such as: performance, computational cost, value revealed-or-not, etc. The missing values in S correspond to the pairs (algorithm, dataset) that are not yet revealed, *i.e.* the algorithm was not run yet on the particular dataset. The purpose of the agent is to find the best algorithm for a dataset of interest

[3] This game has two actions (move, predict) where the second action 'predict' is performed by a pre-trained classifier.

as quickly as possible. This setting assumes that we have a finite number of algorithms and datasets. It is similar to the “brightest patch” MNIST REVEAL game, where MNIST images play the role of one row in the meta-learning dataset (*i.e.* images are “flattened” as a line vector of the S matrix).

3.3.1 1D AutoML: REVEAL($\beta = 0.1, \tau = 20, a_\bullet =$ position of the best algorithm)

There are 2 possible settings in this problem: 1D and 2D. The 1D setting is similar to the segment game in 6.2.1. The agent is given 1 dataset (i.e. 1 row) at a time, its goal is to reveal and find the best algorithm for this dataset. In this setting:

- States: the matrix S is $M \times 1 \times N$. It contains 1 dataset and N algorithms [4], it can have $M > 1$ channels: performance channel, revealed-or-not channel (that is 1 for positions the agent has revealed and 0 otherwise), computational time channel, etc. Hence, if we note the index of the dataset of interest as d, $S_{:,d,j}, j \in (1,\ldots,N)$ is a M-dimensional vector. $M = 2$ in our experiments in Chapter 6 containing performance and computational time. All values in $S_{:,d,j} = NAN$ if the position j is not revealed yet, and channel values otherwise. In the real-world, the matrix S can be infinite because it grows with more and more datasets/algorithms added to the agent’s experience, *i.e.* the action space can be enormous compared to a small time budget τ. In this case, the agent might not be able to achieve the goal $a_\bullet$ within τ, the game termination thus is wholly determined by the condition $t = \tau$, and the agent is rewarded according to the best algorithm found within τ.

- Actions: $\mathscr{P} = \mathscr{J} = \{0,\ldots,N-1\}$ = any position in the row. $a_t = j_t$

- Reward is the combination of performance (computed as the difference of performance of current action S_{0,d,j_t} [5] and the best performance in the action history) and cost (computed as the computational time required to evaluate the chosen algorithm S_{1,d,j_t}):

$$\begin{cases} r_t = r_+(t) - \beta r_-(t) \\ r_+(t) = |S(d,j_t)_{perf} - max_{k=0}^{t-1}(S(d,j_k)_{perf})| \\ r_-(t) = S(d,j_t)_{time} \end{cases} \tag{3.5}$$

[4] An algorithm can be a learning machine, a meta feature, or other information about the dataset

[5] Assume the 0_{th} channel is the algorithm performance and 1_{th} channel is the algorithm computational time.

3.3.2 2D AutoML

Similar setting as in 1D, except now the agent is given more than one rows at a time. The purpose of this 2D setting is to mimic a challenge environment, in which the agent should solve multiple datasets within a resource budget τ, and the final reward is defined upon the average performance over all datasets. The agent thus needs to learn how to schedule its resource across datasets.

- States: the matrix S is $M \times d \times N$. It contains d datasets and N algorithms.
- Actions: $\mathscr{P} = \mathscr{I} \times \mathscr{J} = \{0,\dots,d-1\} \times \{0,\dots,N-1\}$ = any position in the $(d \times N)$ matrix;
- Reward: we use the same reward shaping as in 1D setting, but instead of considering per dataset, we take the average at each time steps.

3.4 Evaluation procedure in REVEAL games

How do we evaluate the performance of a REVEAL agent? There are mainly two ways:

- **Online learning**: No train/test split. Each new game is first treated as a test instance and solved by the agent trained with previous games. During the test time, the agent keeps adapting its parameters. Hence, the test instance is also a training instance. In the case of meta-learning, a game is equivalent to a dataset, i.e., a row in the matrix S.
- **Off-line learning**: There is a clear train/test split, *i.e.* we maintain a train set from which we sample games for the agent to solve during its training time. Then, once the training terminates, the parameters of the agent are frozen, and the testing starts, the agent is asked to solve games sample from the test set. The reward, in this case, is the average reward over the test set. For the meta-learning problems shown in Chapters 5 and 6, for large meta-datasets such as CEP, we use 70% of rows for training and the remaining 30% of rows for testing; for smaller meta-datasets (AutoML, OpenML, StatLog, Artificial), we perform 10 fold cross-validation by testing on one fold at a time.

3.5 Summary on the MDP setting

In this chapter, the AutoML meta-learning problem was formulated as a Markov Decision Process (MDP). This formulation allows us to develop different policies to solve the AutoML problem in later chapters: Hyper-parameter selection where no meta-learning is performed to find a best algorithm for a particular dataset (Chapter 4), Collaborative filtering where a hard-coded policy is used to find the best algorithm in a greedy fashion (Chapter 5) and Reinforcement learning where an agent is trained to learn a policy that is useful for new datasets. (Chapter 6)

Clearly, this simplified formulation has limitations that we will discuss in the last chapter, including that of a discrete finite state space. We also did not impose any particular constraints on action space, which could implement some prior knowledge on hyper-parameter space. We will also discuss in the last chapter possible extensions that we did not develop in this thesis, including representing the problem as a POMDP (Partially Observable Markov Decision Process), exploring continuous state-action spaces (for example continuous hyper-parameters), and revealing matrix S not only line-wise (give an particular dataset), but column-wise (task selection) of bi-bidirectionally (finding the best overall match of algorithms to tasks).

Chapter 4

Algorithm selection and Hyperparameter Optimization: No meta-learning at all

As claimed in Chapter 3, the goal of AutoML is to build an agent to learn a policy $p(a|s)$ in a MDP: the action a can be seen as choosing an algorithm to use, when in the state s. The state s encodes all information exposed to the agent so far, making the whole process a MDP. How to build such agent? There are 3 possibilities: (1) Let the agent learn from trial and error (Chapter 6); (2) Treat unknown algorithm performances as missing values and hard code the policy to choose the algorithm with maximal estimated value (Chapter 5). Both (1) and (2) make use of knowledge gained from past experiences, while (3) Hyperparameter selection looks solely at the task at hand, and assigns appropriate algorithm to it through the optimization of a metric function $\mathscr{L}$ that encodes the task preference (e.g. a loss function that measures the performance of the algorithm applied on the dataset under consideration, or a customized scoring function that trades off between the performance and the computational cost of the algorithm).

4.1 Hyperparameter selection as a CASH problem

No algorithm is best for all tasks. The best choice depends on the task (its data distribution $\mathscr{D}$, the metric $\mathscr{L}$ it uses to measure the performance) and the fact that whether the algorithm is well tuned. Auto-WEKA [110] first proposed Combined Algorithm Selection and Hyperparameter optimization (CASH) to formulate this dependence jointly:

Definition 4. *CASH Given a set of algorithm $\mathscr{A} = \{A^{(1)}, \ldots, A^{(k)}\}$ with associated hyperparameter spaces $\Lambda^{(1)}, \ldots, \Lambda^{(k)}$, we define the combined algorithm selection and hyperparameter optimization problem (CASH) as computing*

$$A^*_{\lambda^*} \in argmin_{A^{(j)} \in \mathscr{A}, \lambda \in \Lambda^{(j)}} \frac{1}{K} \sum_{i=1}^{K} \mathscr{L}(A^{(j)}, \mathscr{D}^{(i)}_{train}, \mathscr{D}^{(i)}_{valid})$$

where $\mathscr{L}(A^{(j)}, \mathscr{D}^{(i)}_{train}, \mathscr{D}^{(i)}_{valid})$ is the metric value achieved by A trained on $\mathscr{D}^{(i)}_{train}$ and evaluated on $\mathscr{D}^{(i)}_{valid})$

This way, the agent policy is hard coded as:

$$p(a|s) = \begin{cases} 1 - \varepsilon & \text{if } a = A^*_{\lambda_*} \\ \varepsilon & \text{otherwise} \end{cases}$$

where s encodes the current task, and ε trades off exploration - exploitation.

CASH setting is a particular way of formulating the Hyper-parameter optimization problem, but others are possible (e.g. having a single validation set or even using the training error if we have really "big data").

4.1.1 How to solve CASH?

The first solution family is BlackBox Optimization: without knowledge about the real function $\mathscr{L}$, we can only query algorithm points $A^{(j)}_{\lambda}$ and observe its the value of $\mathscr{L}(A^{(j)}_{\lambda})$. Then we can choose building a model of $\mathscr{L}$ (model-based) or not (model-free). The examples of model-free optimization include grid search (exhaustive search over hyperparameter space $\Lambda^{(j)}$)); random search [12] (starting from a random position λ_j in $\Lambda^{(j)}$), sample a neighbor position λ'_j from the hypersphere of a given radius surrounding λ_j, update the searching center to the new position if $\mathscr{L}(A^{(j')}_{\lambda}) > \mathscr{L}(A^{(j)}_{\lambda})$. Because the metric function $\mathscr{L}$ has usually a low effective dimension (i.e. some dimensions of $\mathscr{D}$ are more relevant to $\mathscr{L}$ than other), random search is proven to be more efficient than grid search in practice, especially when the dimension of $\mathscr{D}$ grows); population-based optimization (e.g. evolutionary algorithm [91], particle swarm optimization [31], which maintain an algorithm population and perform local mutation, recombination on these candidate to achieve better algorithm generations).

The standard approach of model-based optimization is Bayesian Optimization (BO) [21]. BO combines the Baye's Theorem and an acquisition function for efficient sampling in the search space $\Lambda^{(j)}$ to optimize an objective function $\mathscr{L}$. The Bayes' theorem

$$p(\mathscr{L}|\{A_\lambda^{(j)},\mathscr{L}(A_\lambda^{(j)})\}) \propto p(\{A_\lambda^{(j')},\mathscr{L}(A_\lambda^{(j')})\}|\mathscr{L})p(\mathscr{L})$$

allows the optimization to start from a prior $p(\mathscr{L})$ encoding our belief about the unknown objective function. After some observations (or queries) $\{A_\lambda^{(j)},\mathscr{L}(A_\lambda^{(j)})\}$ we can update the likelihood probability $p(\{A_\lambda^{(j)},\mathscr{L}(A_\lambda^{(j)})\}|\mathscr{L})$, and the product of both allows us to refine progressively the posterior probability which is our model of $\mathscr{L}$ given observations. In practice, $p(\mathscr{L})$ is usually approximated using Gaussian process (GP) [90][1]. Random Forest can also be used in the place of GP for its good performance on high-dimensional and discrete data [55, 109, 35]. At each iteration, an acquisition function is used to decide the next point $A_\lambda^{(j)}(t+1)$ to sample while automatically trades off the exploration and exploitation, some examples include expected improvement [77] and entropy search [50].

Instead of directly modeling the posterior $p(\mathscr{L}|\{A_\lambda^{(j)},\mathscr{L}(A_\lambda^{(j)})\})$ like in GP, Tree-structured Parzen Estimator Approach (TPE) [15] proposed to model the likelihood $p(\{A_\lambda^{(j')},\mathscr{L}(A_\lambda^{(j')})\}|\mathscr{L})$ using 2 non-parametric densities $l(A)$ and $g(A)$, where $l(A)$ is built using all observations $\{A_\lambda^{(j')}$ such that $\mathscr{L}(A_\lambda^{(j')})\} < L^*$ and $g(A)$ using remaining observations. Expected improvement is then used as acquisition function for next-point sampling. TPE is the sub-routine in AutoML toolkit Hyperopt-sklearn [65].

As BlackBox optimization receives algorithm performance as feedback, it can suffer when the performance evaluation gets expensive. This brings out the second family: Multi-Fidelity optimization, which trades off between the performance approximation and the evaluation complexity. One example is HyperBand [69] which starts the evaluation with randomly sampled HP configurations and successively halve the less promising ones. BOHB [33] replaces the random sampling in HyperBand with BO.

4.2 Freeze-Thaw: My exploration on HP selection

My first experience in AutoML was the AutoML0 challenge in 2015-2016, at that time AutoML and HP selection were equivalent to me. My final submission to the challenge was a re-implementation of the Freeze-Thaw Ensemble Construction algorithm (FTEC). FTEC was first implemented by a top participant ‘jlr44’ [73] where an ensembling was added on-the-fly on top of the original Freeze-Thaw algorithm [106] to profit from the data mining experiences.

[1] GP is a distribution over functions, it is fully defined by a mean function $m(A)$ and a kernel $k(A,A')$ that expresses the covariance between 2 points A and A'.

Freeze-Thaw Bayesian Optimization (BO) suggests to repeatedly stop less promising algorithms and resume / start promising ones so that limited computational resource can be concentrated on training good candidates. This is done by building 2 models: (1) model of HP from which new configurations $A_{\lambda}^{(j)}$ are sampled, this is an infinite mixture of exponential decays Gaussian Process (GP); (2) learning curve model for each sampled configuration, this is a smooth GP over time. BO with entropy search as acquisition function is then used to decide which algorithm to explore further. These ingredients together make Freeze-Thaw an any-time interruptible algorithm suitable for AutoML challenge. FTEC implements FT in the scikit-learn searching space, and uses cross-validation to evaluate the candidates. FTEC furthermore implements a stacking ensembling and memory management (e.g. quick predictions generated by pilot algorithms trained on reduced size data as a backup for algorithm failures) to reinforce the algorithm's power during the challenge. I then improved the FTEC by replacing all detected failures with a heuristic search to make it more task-flexible, hence less failures in the challenge environment 4.1.

4.3 Summary of the chapter on HP selection

In this chapter, the first level of AutoML was explored: no meta-learning at all, *i.e.* solving the "model/algorithm selection" problem for individual datasets independently, focusing only on hyper-parameter (HP) selection. The approach taken was that of Freeze-Thaw, which includes core techniques commonly used in HP selection, such as Bayesian optimization and learning curve estimation. Even pure HP selection solutions achieve good results in many applications, including the AutoML challenge, their limitation lying in the absence of meta-learning. This greatly limits the AutoML system's ability to learn from its own past, *i.e.* across datasets. Therefore, in the follwing chapters, we will start exploring meta-learning.

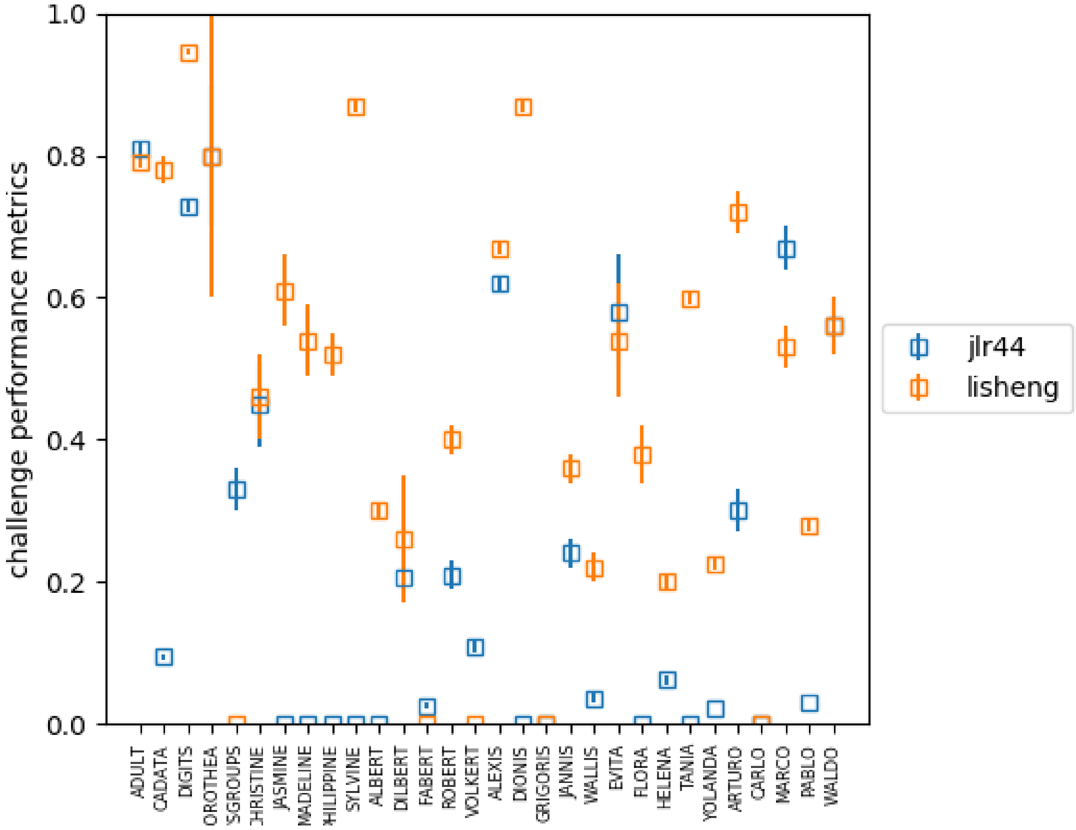


Fig. 4.1 The original FTEC implemented by ‘jlr44’ has been repaired after the challenge, and compared with my adapted version, which avoids a large number of failures (with performance 0). The simulations are rerun in a challenge-like environment with the same time budgets. X-axis are challenge dataset names. On y-axis the performances are computed using the specific metric associated to each dataset used in the challenge, error bars are computed through boostraping.

Chapter 5

Active Meta-learning

In Chapter 3, we cast the AutoML problem as a MDP, fully represented by the tuple $(S, A, P_{S_{t+1}|S_t,a_t}, R)$ where state S is the Meta-learning matrix, A is the set of actions (usually the set of positions in S available for the agent to reveal), P is the transition probability from S_t to S_{t+1} when a_t is taken and R is the reward function.

One way to solve this AutoML MDP is to use Reinforecment learning, where an agent is trained to **learn a good policy** to explore S efficiently, i.e. achieving the region of maximum values within as few steps as possible. This is our next main contribution detailed in Chapter 6.

In this chapter, we explore meta-learning with policies that are NOT learned. Since the central problem is deciding which position in S to explore in the next step, the policy can be **hard-coded** with the help of an estimator (or surrogate score) $\hat{S}$ of missing values in S. In this case, the agent chooses the element in $\hat{S}$ satisfying some criterion (*e.g.* the one with highest predicted value or highest expected improvement in a Bayesian optimization context). The surrogate score is refined over time when more and more elements of matrix S are revealed. We call this approach **active Meta-learning**.

The code for experiments in this Chapter is at https://github.com/LishengSun/ActiveMetaLearn. [1]

5.1 Active Meta-learning as a Recommendation problem

Recommender systems belong to a subclass of information filtering systems, which seek to predict the "rating" or "preference" that a user would give to an item. They are

[1] To run the code, CofiRank must be installed. We recommend using the Docker [Doc] image we built for this purpose. The repository also includes a Jupyter-notebook to demonstrate the experiments.

widely used by platforms like Amazon, Ebays, etc. to propose products to purchase. This problem can be represented by a sparse rating matrix S where $S_{i,j}$ is the rating if user i has rated item j, otherwise $S_{i,j}$ is a missing value. One common design of recommendation system is Collaborative Filtering (CF), which is based on the assumption that similar users would like similar items. This can be translated to the fact that the rating matrix S (of size (D,N) where D is the number of users and N the number of items) can be approximated as product of 2 matrices $S \sim UV'$ (U of size (D,d), V of size (N,d) with $d \leqslant min(D,N)$) under the assumption that S is of low rank d. A common approach to solve this approximation is to take it as a regression problem and let UV' estimate directly the ratings in S (e.g. Maximum Margin Matrix Factorization [97, 92] minimizes the trade-off between the complexity of UV' and the hinge loss between UV' and S computed on known values in S, the complexity of UV' actually plays the role of regularizer to ensure the generalization to unknown values). However, the authors of CofiRank [120] argue that approximating the ratings is actually a harder problem because ratings are biased by individual users, and that one should instead estimate the ranking (of the ratings). In this context, CofiRank propose to build a matrix F that maximize the ranking measure NDCG between S and F. CofiRank is used as a subroutine in our algorithm ACTIVMETAL, section 5.2.1 gives more detail about CofiRank.

Meta-learning can be thought of an algorithm recommender system in that a dataset likes more a particular algorithms because the latter performs better on it. In this sense, the rating matrix S becomes our Meta-learning matrix (of size (D,N) where D is the number of datasets and N the number of algorithms [2]). CF techniques can then be used to estimate the missing performances (MMMF) or their rakings (CofiRank), and the one with highest estimated value will be recommended.

Alors [75] applies CofiRank to build such algorithm recommender. However, there is still a challenge in algorithm recommender system: the cold start problem, i.e. how to deal with a brand new dataset (a complete empty line appended to S). This requires the dimension augmentation from D to $D+1$ in S. Alors uses dataset meta-features X of size (d',D) to learn a mapping from X to U, since the new task has its meta-features of size $(d',1)$, the learned mapping brings it to $(1,d)$ which is then appended to U and the dimension augmentation is complete, standard CF techniques such as CofiRank can then be applied. There are other approaches to deal with the cold start, in our

[2] According to Chapter 3, the dimension of S is $M \times D \times N$ where M is the number of channels. Here when we treat the problem as a recommender system, we consider only $M = 1$ channel which is the performance as the sole learning source. We note $S_{d,j}$ the value of position (d,j) in this 1-channel matrix S.

ACTIVMETAL [101], we augment the dimension by random selection or selecting the column with highest median over old tasks.

Probabilistic Matrix Factorization for AutoML [40] proposes a similar approach than ACTIVMETAL, but models the performance using a Gaussian Process [90]: $p(S|U,V,\sigma^2) = \prod_{n=1}^{N} \mathcal{N}(S_n|U_nV', \sigma^2\mathbb{I})$, the GP is refined by optimizing the log likelihood on known entries, and Bayesian optimization is used to recommend the next algorithm to evaluate.

5.2 The ACTIVMETAL algorithm

Our paper [101] proposes to treat the AutoML problem as an algorithm recommender system and use collaborative ranking technique (CofiRank[120]) to find as quickly as possible the best algorithm for a new task. Section 5.2.1 give a brief summary on CofiRank, section 5.2.2 introduces the algorithm ACTIVMETAL and its application to real world problems.

5.2.1 CofiRank: the subroutine for recommendation

Given the sparse rating matrix $S \in \mathbb{R}^{(N,D)}$ of N users and D items, CofiRank proposes to build a full low-rank matrix $F \in \mathbb{R}^{(N,D)}$, $F \sim UV'$ in such a way that ranking entries in F is equivalent to ranking entries in S. The performance of F is measured by Normalized Discounted Cumulative Gain (NDCG [118]):

$$R(F,S) = \sum_{i=1}^{N} NDCG@k(\Pi^i, S^i) \tag{5.1}$$

where $\Pi^i = argsort(-F^i)$ is the sorted row (i.e. user) i of matrix F.

$$DCG@k(s,\pi) = \sum_{i=1}^{k} \frac{2^{s_{\pi_i}} - 1}{log(i+2)} \tag{5.2}$$

where s_{π_i} is the vector s (thus a certain row in S) after the permution π. DCG is designed to focus on the top k ranked items via the denominator, k is the truncation value. DCG is then normalized to be bouneded in $[0,1]$, 1 when $\Pi = \Pi_*$:

$$DCG@k(s,\pi) = \frac{DCG@k(\Pi^i, S^i)}{DCG@k(\Pi^i_s, S^i)} \tag{5.3}$$

Given a F, we can have its performance measure $R(F, Y_{train})$, but we want to maximize $R(F, Y_{test})$, this is done by restricting the complexity of F (detailed later).

But $R(F,Y)$ is non-convex, to solve the maximization, we will instead minimize a convex upper bound of $-NDCG(\pi,s)$ for each individual user and sum them up. This convex upper bound is defined as:

$$l(f,s) := \max_{\pi}[\Delta(\pi,s) + < c, f_{\pi} - f >] \geq \Delta(\pi,s_{*}) \tag{5.4}$$

where $\Delta(\pi,s) := 1 - NDCG(\pi,s)$ brings NDCG to a regret loss. $< c, f_{\pi} - f >$ is the inner product between a decreasing sequence c and f permutated by π, this maps the rating f to a real value such that maximizing the latter yields $argsort(f)$ (Polya-Littlewood-Hardy inequality).

Now, maximizaing R becomes minimizing $l(f,s)$ over all users i:

$$L(F,S) = \sum_{i=1}^{N} l(F^{i}, S^{i}) \tag{5.5}$$

Replacing F by UV' and adding a regularizer $\Omega[F] := \frac{1}{2} min_{U,V}[trUU' + trVV']$, we get the final optimization problem:

$$\min_{U,V} L(UV', S_{train}) + \frac{1}{2}[U,V[trUU' + trVV'] \tag{5.6}$$

U (size (N,d)) and V' (size (d,D)) are respectively the user and item matrix that captures the user / item specifics. The rank d in CofiRank is rather computational concerns, $d = 10$ or 100 in their rating matrix experiments. The problem 5.6 is solved by minimizing alternatively U and V. Because the term L is expensive to minimize, they use bundle methods for a quicker converge rate.

5.2.2 ACTIVMETAL

In our paper ACTIVMETAL [101], we define the **active meta-learning problem** in a **collaborative filtering recommender** setting as follows:
GIVEN:

- An ensemble of **datasets** (or tasks) $\mathscr{D}$ of elements d (not necessarily finite);
- A finite ensemble of n **algorithms** (or machine learning models) $\mathscr{A}$ of elements $a_j, j = 1, \cdots, N$;
- A **scoring program** $\mathscr{S}(d,a)$ calculating the performance (score) of algorithm a on dataset d (*e.g.* by cross-validation). Without loss of generality we will assume that **the larger** $\mathscr{S}(d,a)$**, the better**. The evaluation of $\mathscr{S}(d,a)$ can be

computationally expensive, hence we want to limit the number of times S is invoked.

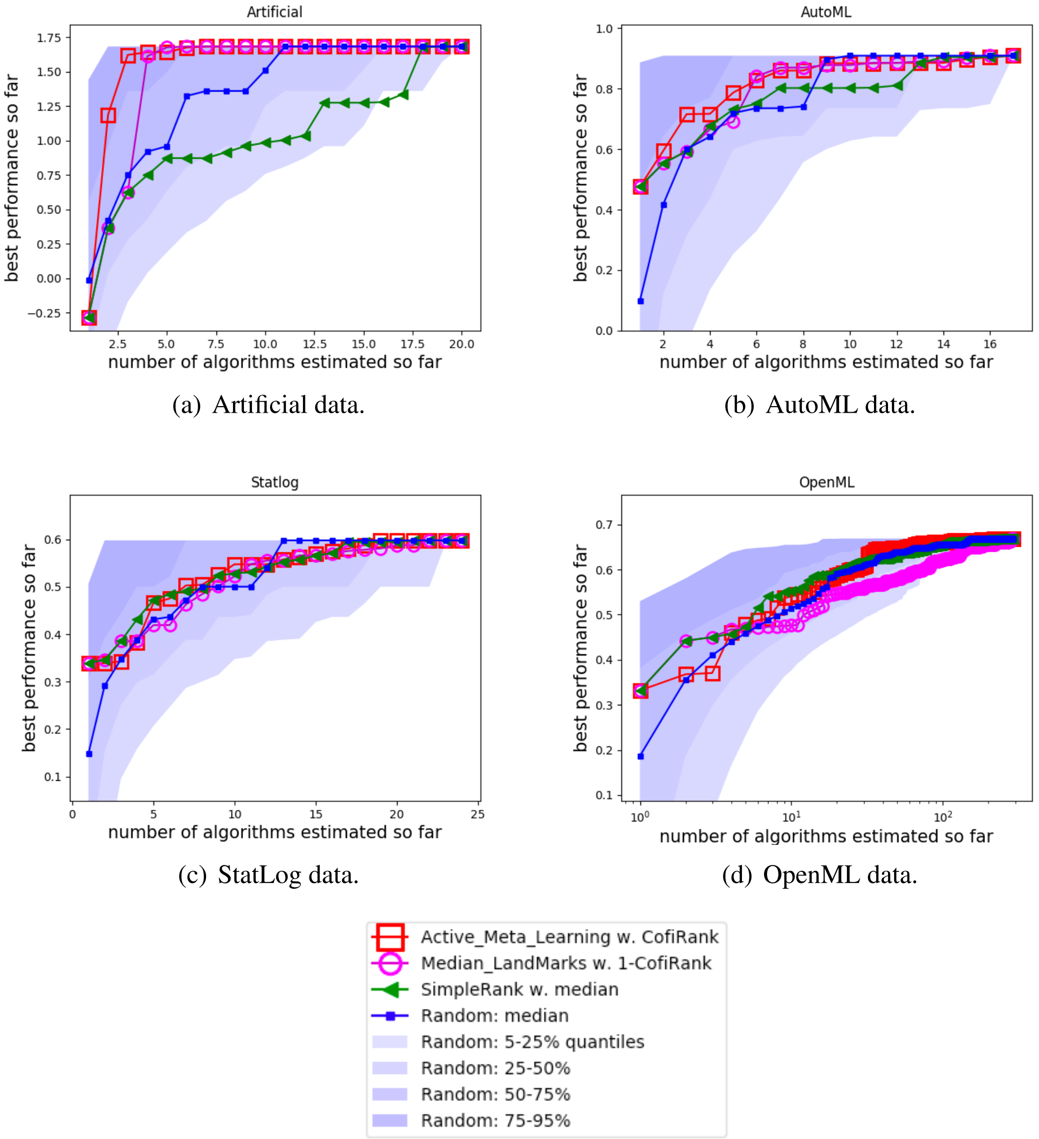
- A **training matrix** S, consisting of p lines (corresponding to example datasets $d_i, i = 1, \cdots p$ drawn from $\mathscr{D}$) and n columns (corresponding to all algorithms in $\mathscr{A}$), whose elements are calculated as $\mathrm{S}_{ij} = \mathscr{S}(d_i, a_j)$, but may contain missing values (denoted as NaN).
- A **new test dataset** $d_t \in \mathscr{D}$, NOT part of training matrix S. This setting can easily be generalized to test matrices with more than one line.

GOAL: Find "as quickly as possible" $j_* = argmax_j(S_{d_t,a_j})$. For the purpose of this paper "as quickly as possible" shall mean by evaluating as few values of $S_{d_t,a_j}, j = 1, \cdots, n$ as possible. This can be broken down to 2 processes: (1) InitializationScheme(S) which chooses the first algorithm to evaluate on the new dataset d_t; (2) SelectNext$(S, \mathbf{t})$ which select the next algorithm to evalute at time **t**, this is where CofiRank can be applied. Algorithm 1 gives the general algorithm, in which different strategies can be plugged into those two above processes. In the paper we compared ActiveMetaLearningCofiRank (initialization = best median, selection according to ranking returned by CofiRank at each time step) with 3 baselines Random (random initialization and selection), SimpleRankMedian (initialization and selection according to the ranking of median over all old datasets), MedianLandmarks1CofiRank (initialization = best median, selection according to ranking returned by CofiRank at **t**). We applied our methods on 1 Artificial dataset and 3 real-world datasets, the result is shown in Figure 5.1. For more details please refer to the full paper [101].

Algorithm 1 ActivMetaL

```
procedure ActivMetaL($\mathscr{A}$, $\mathscr{S}$, S, $d_t$, $n_{max}$)
    $n \leftarrow size(\mathrm{S}, 2)$                ▷ Number of algorithms to be evaluated on $d_t$
    $\mathbf{t} \leftarrow$ NaNvector$(n)$ ▷ Algorithm scores on $d_t$ are initialized w. missing values
    $j_+ \leftarrow$InitializationScheme(S)    ▷ Initial algorithm $a_{j+} \in \mathscr{A}$ is selected
    while $n < n_{max}$ do
        $\mathbf{t}[j_+] \leftarrow \mathscr{S}(d_t, a_{j+})$ ▷ Complete t w. one more prediction score of $a_{j+}$ on $d_t$
        $j_+ =$ SelectNext(S, t)
        $n \leftarrow$length(notNaN(t))              ▷ number of algorithms evaluated on $d_t$
    return $j_+$
```

ActivMetaL differs from other traditional algorithm selection techniques by the facts: (1) it is purely meta-learning, the only learning source is the performance of



(a) Artificial data.

(b) AutoML data.

(c) StatLog data.

(d) OpenML data.

Fig. 5.1 **Meta-learning curves.** We show results of 4 methods on 4 meta-learning datasets, using the leave-one-dataset-out estimator. The learning curves represent performance of the best model trained/tested do far, as a function of the number of models tried. The curves have been averaged over all datasets held-out. The method *Active Meta-learning w. CofiRank* (red curve) generally dominates other methods. It always performs at least as well as the median of random model selection (blue curve), a hard-to-beat benchmark. The more computationally economical *Median Landmark w. 1 CofiRank* consisting in training/testing only 3 models (Landmarks) to rank methods using only 1 call to CofiRank (pink curve) generally performs well, except on OpenML data for which it would be most interesting to use it, since this is the largest Meta-learning datasets. Thus active learning cannot easily be replaced by the use of Landmarks, lest more work is put into Landmark selection. The method *SimpleRank w. median* that ranks algorithm with their median performance (green curve) is surprisingly a strong contender to *Active Meta-learning w. CofiRank* for the StatLog and OpenML datasets, which are cases in which algorithms perform similarly on all datasets.

algorithms on past tasks; (2) it is active learning, it queries S_{d_t,a_j} on the fly to improve its estimations; (3) it is flexible, other matrix factorization technique can be used in place of CofiRank. Based on these properties, it will serve as a baseline in our work on treating Meta-learning as a REVEAL game (Chapter 6).

5.2.3 Comparison of algorithms for some single dataset

In this section, we compare different meta-learning algorithms on some single dataset for each of the meta-datasets. The observations are consistent with that in the average curves 5.1. For the Artificial meta-dataset (Fig. 5.2), which is generated to favor the low-rank propriety, not surprisingly, the two active meta-learning algorithms (red and violet curves) outperform other algorithms for most of the time. For the AutoML meta-dataset (Fig. 5.3), *Active Meta-learning w. CofiRank* does generally well, especially in the first steps; 'AutoML test_0' illustrates a special case where *SimpleRank w. median* gives a good approximation to the true ranking, and thus outperforms other comparator approaches. In the case of OpenML and StatLog (Fig. 5.4 and Fig. 5.5) it is seen that one iteration of Cofirank (violet curve) yields good results, this is interpreted for StatLog because the meta-dataset lacks diversity, so exploration is not that much needed.

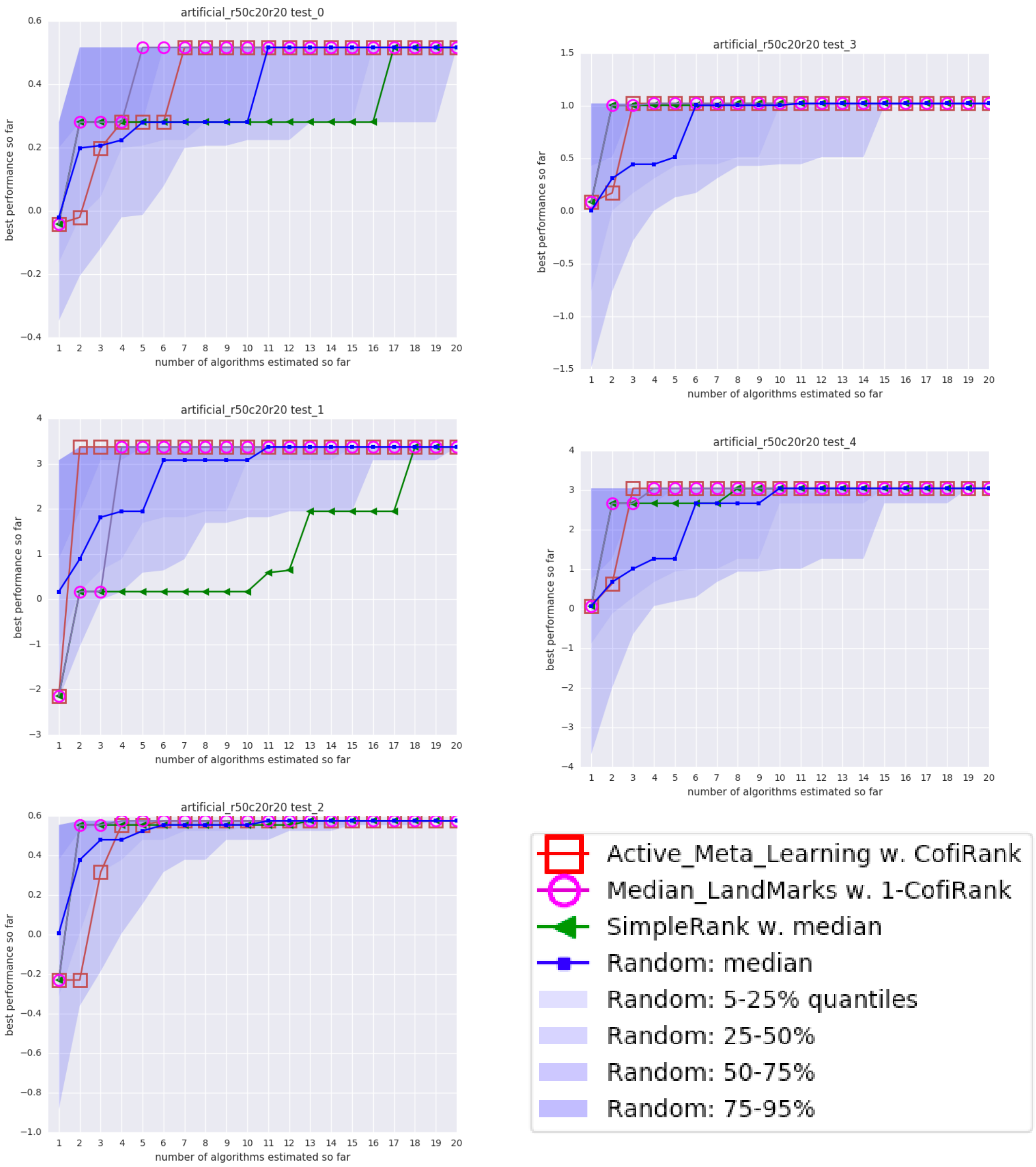


Fig. 5.2 Artificial DATA: the comparison of Meta-learning algorithms for 5 single test datasets are shown. The random curves are median over 1000 runs, the shading area are 5%, 25%, 75% and 95% quantiles.

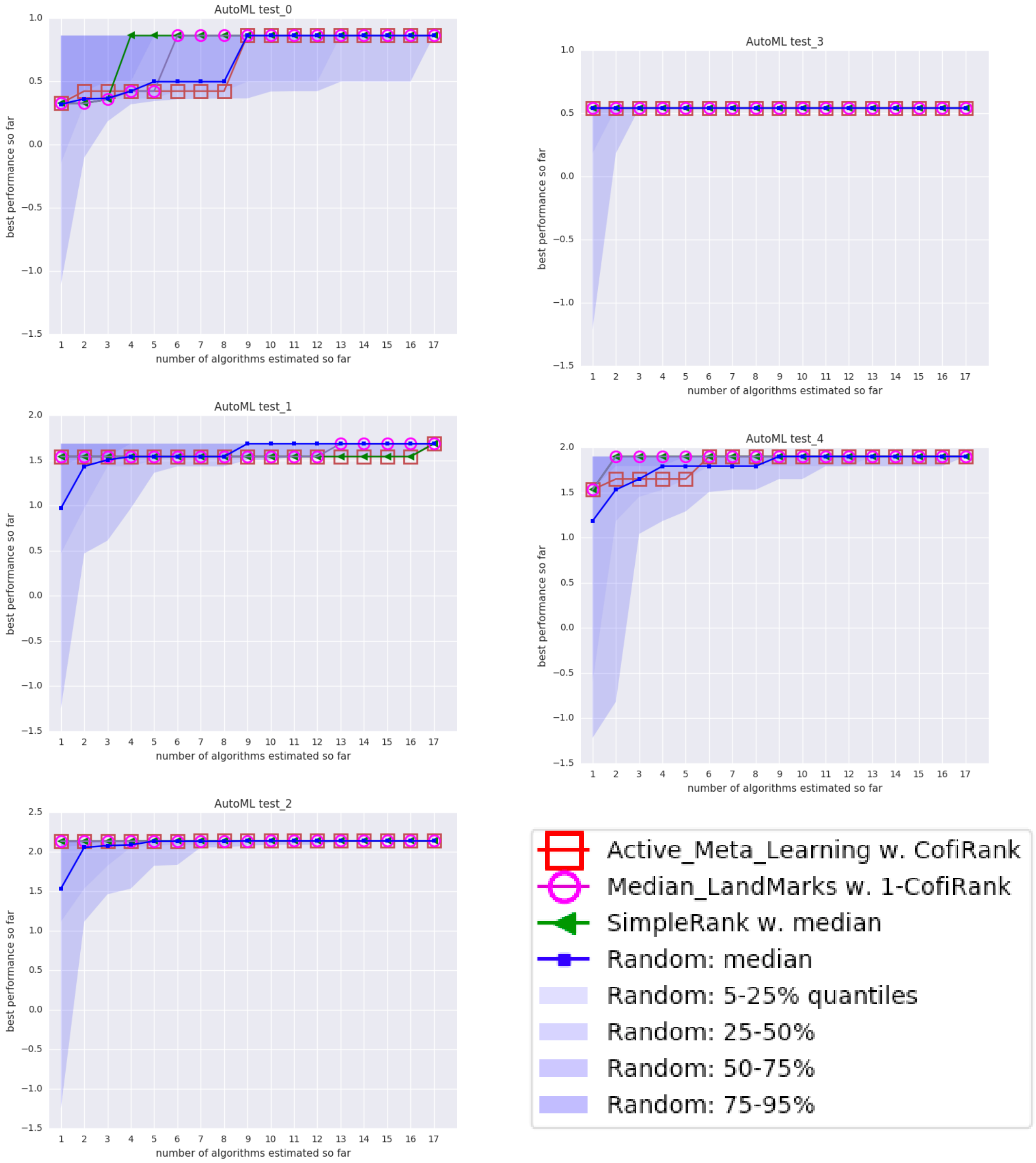


Fig. 5.3 AutoML DATA: the comparison of Meta-learning algorithms for 5 single test datasets are showed. The random curves are median over 1000 runs, the shading area are 5%, 25%, 75% and 95% quantiles.

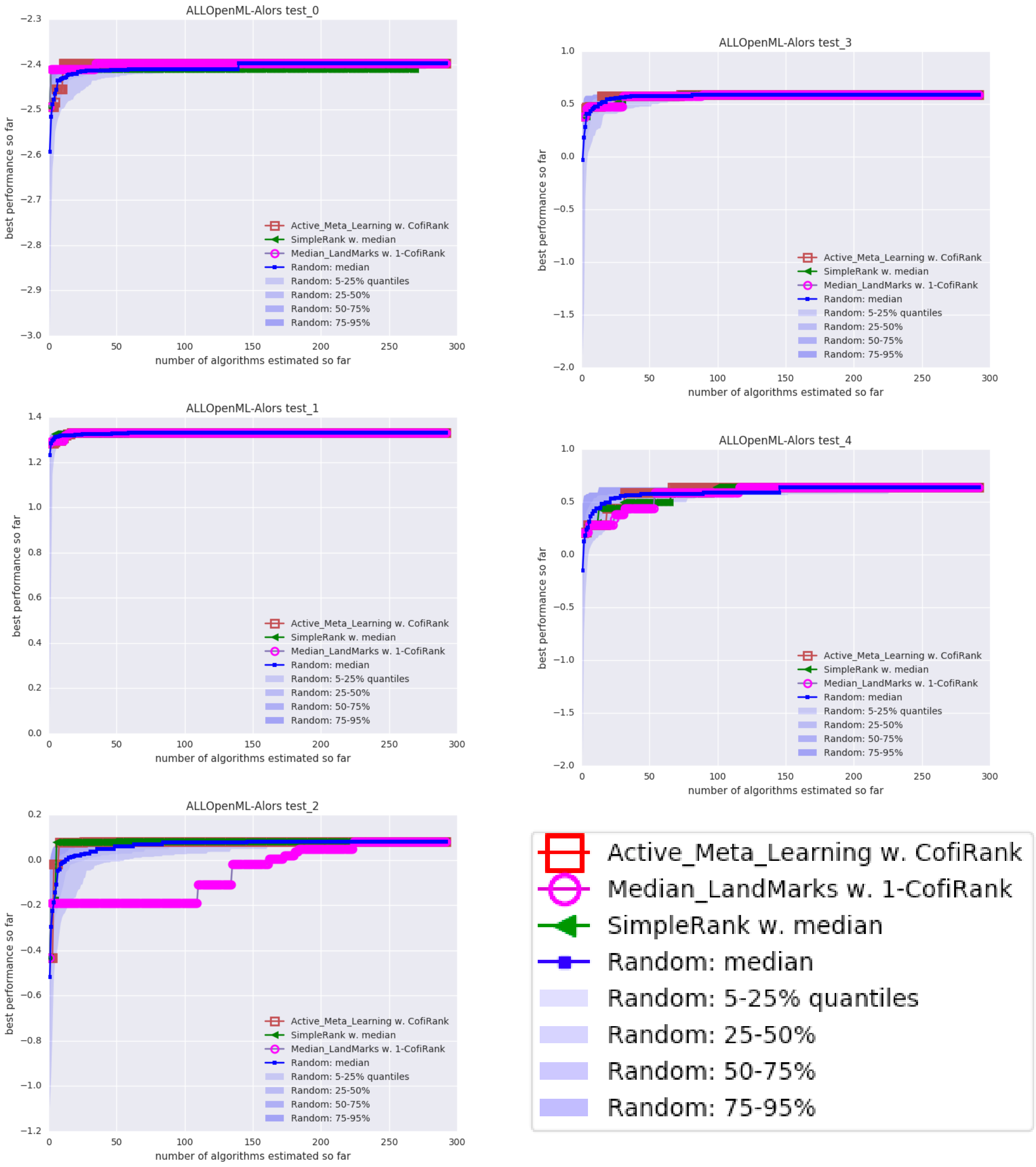


Fig. 5.4 OpenML DATA: the comparison of Meta-learning algorithms for 5 single test datasets are showed. The random curves are median over 1000 runs, the shading area are 5%, 25%, 75% and 95% quantiles.

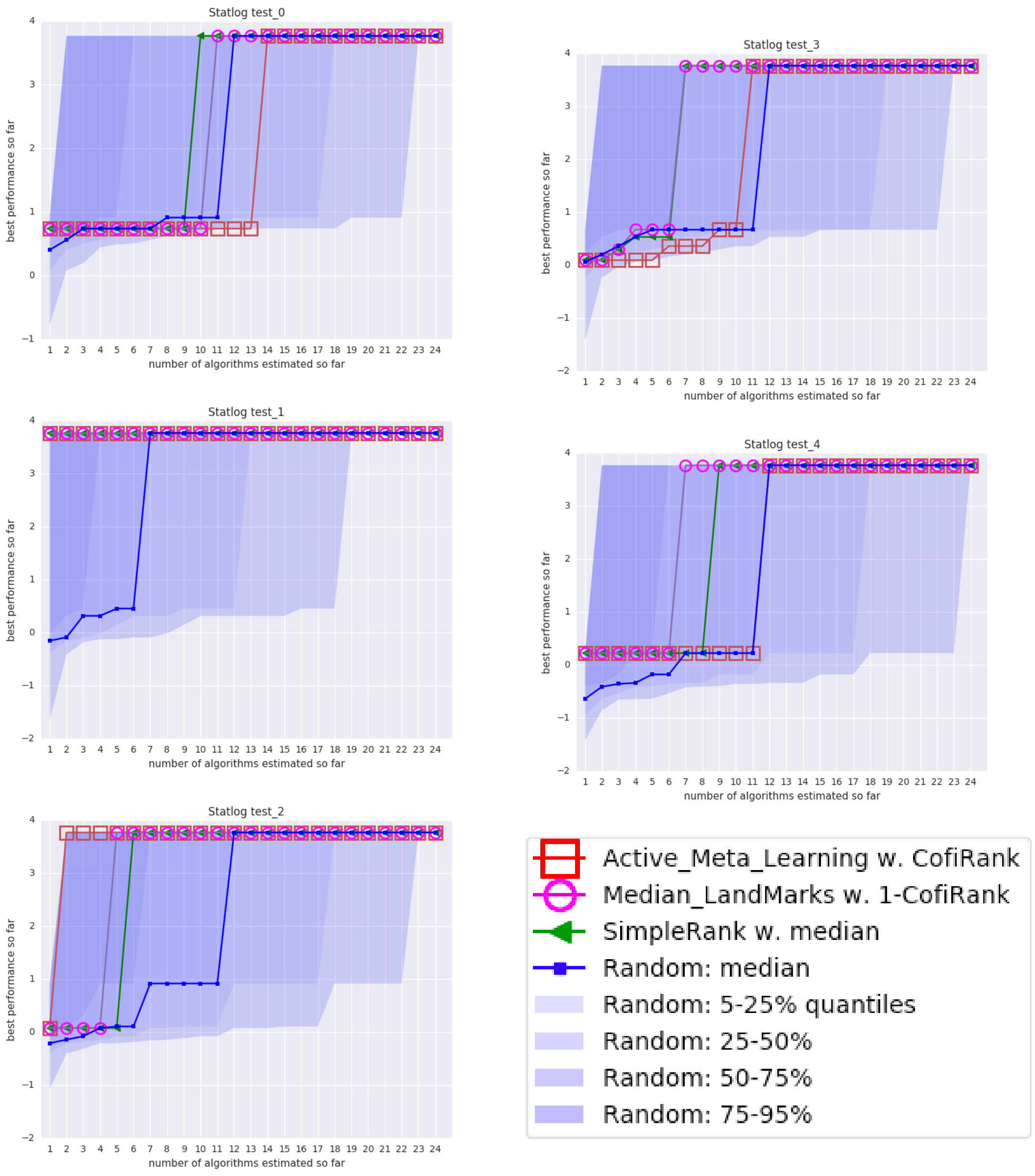


Fig. 5.5 StatLog data: the comparison of Meta-learning algorithms for 4 single test datasets are showed. The random curves are median over 1000 runs, the shading area are 5%, 25%, 75% and 95% quantiles.

5.2.4 Comparison with SVD-based algorithms

We have tested other SVD-based ranking methods: Instead of using median (over performances of all datasets in S) to obtain an algorithm ranking, we can also use SVD decomposition and rank the algorithms according to their projections on S's column space, i.e. $S \sim UV'$, $S_{:,k} \cdot U_{:,0}$ is the projection of k^{th} algorithm of S on column space U's first component. The higher this value is, the better k's ranking is. .

- SVD-landmarks w. 1 CofiRank: Run only once CofiRank, which is warm-started by the first SVD-ranked algorithm, to obtain a single ranking according to which the selection is performed.
- Simple rank w. SVD / median: No CofiRank is used in this case. The single ranking is given by SVD median.

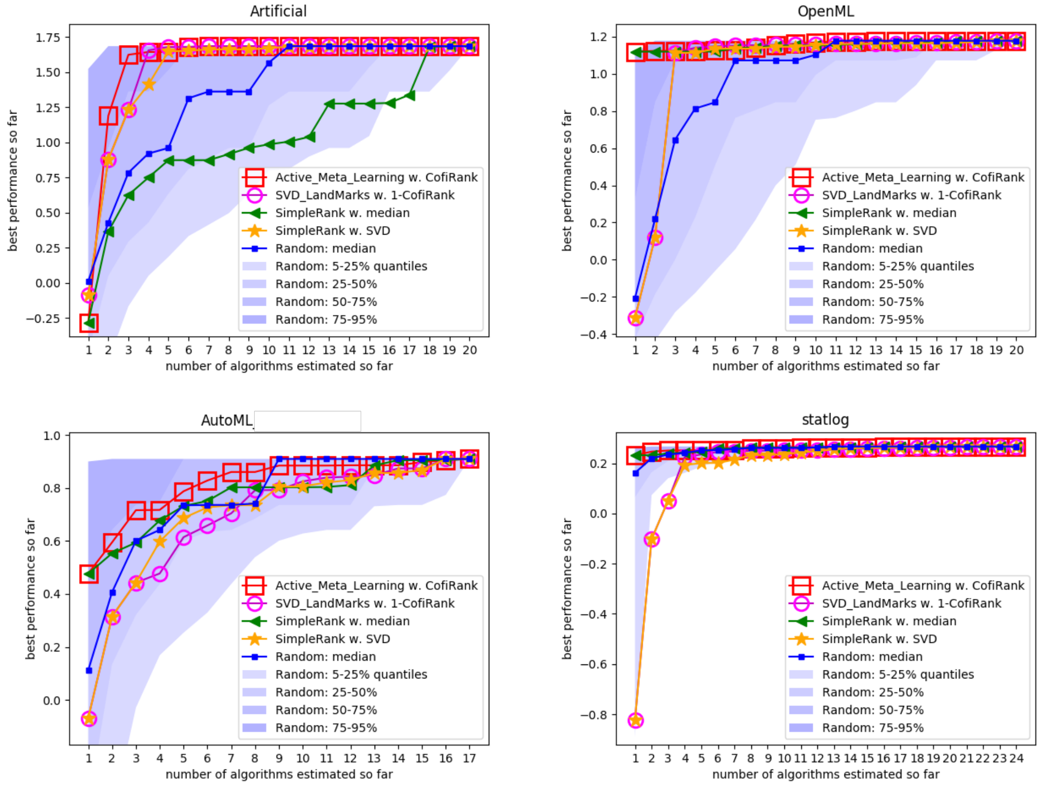


Fig. 5.6 **Meta-learning curves with SVD-based ranking.** We show results of 5 methods on 3 meta-learning datasets, using the leave-one-dataset-out estimator. The learning curves represent performance of the best model trained/tested do far, as a function of the number of models tried. The curves have been averaged over all datasets held-out. The conclusion remain unchanged as in 5.1: the method *Active Meta-learning w. CofiRank* (red curve), warm-started with the best medien algorithm, generally dominates other methods. The new added method *SimpleRank w. SVD* performs competitively well on Artificial and OpenML (which catches up quickly after a low initialization) where the low-rank assumption of SVD is better ensured (see visualization on singular values of these meta data in Section 2.3).

5.3 Summary of the chapter on ACTIVMETAL

In this chapter, we proposed a first meta-learning solution to the AutoML MDP called ACTIVMETAL. The objective of ACTIVMETAL is to find the best algorithm for a particular new dataset by evaluating as few algorithms as possible. This is done by a greedy policy that chooses the algorithm with the highest estimated performance. Even though ACTIVMETAL achieves promising results in the investigated real-world meta-datasets, its limitation lies in the fact that each selection step requires a new matrix factorization. ACTIVMETAL thus becomes computationally costly when the meta-dataset is large. Therefore, in the next chapter, we will start exploring the RL approach that learns a policy across datasets and can be used directly by any new dataset.

Chapter 6

RL solutions to meta-learning

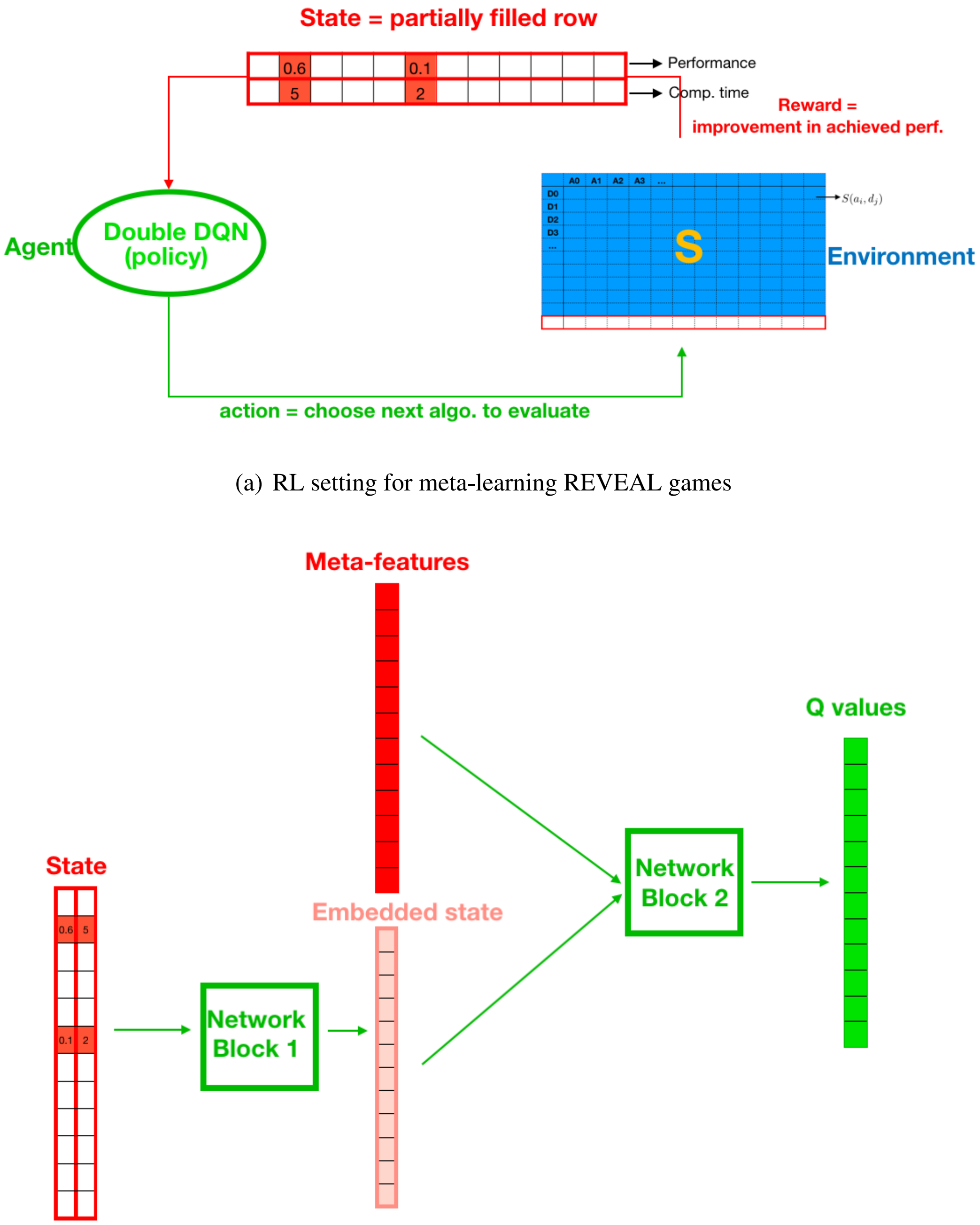
In this chapter, we experiment with novel techniques of Reinforcement Learning (RL) in an attempt to push the frontiers of meta-learning, which was previously defined as a MDP, or more precisely, a REVEAL game in Chapter 3. The central idea is to learn a policy from past tasks/datasets, then apply it to search for the best algorithm solving a new task. We apply "Deep" Reinforcement learning techniques, especially Double Deep Q-Networks (Double DQN). After an introduction to RL in Section 6.1, we introduce two toy REVEAL games to illustrate our problem-solving methodology using Double DQN (Section 6.2). We then present experimental results on meta-learning as a REVEAL game, and compare them with various baselines, including ACTIVMETAL, the method we introduced in Chapter 5.

The Code for this chapter is found at https://github.com/LishengSun/REVEAL.

6.1 Overview of Reinforcement Learning (RL)

We briefly review a few concepts of Reinforcement Learning, which are necessary to understand our approach (see *e.g.* [103] for a more complete treatment). In supervised learning, the "correct answer" (also called ground truth or target value) **y** is provided to the learning machine for each training input **x**, such that the learning machine (or agent) can learn to produce correct desired outputs. This setting is typical of classification or regression problems. In contrast, in Reinforcement Learning (RL), the agent is given a reward instead of the correct answer and must learn by trial and error to produce the correct answer, without ever being told that answer. This setting is typical of control problems, games, and Markov Decision Processes (MDPs).

See Figure 6.1 for an overview of our system.



(a) RL setting for meta-learning REVEAL games

(b) Deep Q-Learning Network (DQN)

Fig. 6.1 **System overview**: In this chapter, we consider that the RL agent's aim is to learn a policy to solve a MDP problem, more specifically a REVEAL game. Thus, what is usually referred to as "the environment" or the "world" is the "board" of our REVEAL game (a row of the meta-data matrix in the particular case of meta-learning). (a) Matrix S contains pre-computed meta-learning knowledge (the meta-dataset), which plays the role of our environment. In a given game episode, the environment "decides" on which row of matrix S to work on (the hidden "game board"). The agent takes successive actions to reveal algorithm performances and computational times on the particular dataset (values in the two-dimentional matrix row). The actions are decided by the agent's policy learned using "double" Deep Q-Learning (DQN), see text in Sec.6.1.3. (b) In our DQN architecture, the input to the neural network is the state, a two-dimensional vector representing the current partially revealed scores and computational time. An embedding of the state is then learned via the first network block to which meta-features are appended. A second block network is then used to estimate the action values, which is the output.

6.1.1 Notations and definitions

We consider RL solutions to Markov Decision Processes. RL algorithms' goal is to find a policy that maximizes an expected (discounted) reward. This is usually achieved by estimating the value of a state under that policy, or more conveniently the value of a state-action pair. We recall such definitions and the Bellman equations providing conditions for finding an optimal policy.

MDP

In what follows, we assume a finite MDP, defined as follows:

- $\mathscr{S}$: set of states $s \in \mathscr{S}$
- $\mathscr{A}$: set of actions $a \in \mathscr{A}$
- r: rewards
- $\pi(a|s) : \mathscr{S} \to \mathscr{A}$: a policy specifying the probability of taking action a in state s
- $p(s', r|s, a)$: transition probability, *i.e.* model of environment

The goal of an agent is to learn or propose a policy that maximizes the expected return G_t, an accumulation of future rewards. This expected return is defined differently depending on the *episodic* or *continuous* nature of the task. If the decision process $\{s_0, a_0, r_1, \dots, s_t, a_t, r_{t+1}, \dots, \}$ terminates naturally at $t = T$ with a transition to a terminal state $s_\bullet$, the task is called *episodic*, with episode length T. The expected return G_t is then defined as:

$$G_t = \sum_{k=t}^{T} r_k \tag{6.1}$$

In contrast, if the decision process never ends ($T \to \infty$), the task is a continuing task. In that case, the expected return G_t is defined as:

$$G_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1} \ , \tag{6.2}$$

γ being called the discounting factor ($0 < \gamma < 1$). This allows us to define G_t as a moving average and avoids dealing with an ever growing quantity. In what follows adopt this definition even for episodic tasks, granted that, if T is finite ($0 < \gamma \leq 1$).

State value $v_\pi(s)$

The value of a state s under some policy π is the total reward the agents will accumulate by starting from s and following π thereafter. Therefore, by definition:

$$v_\pi(s) = E_\pi[G_t|s_t = s] \tag{6.3}$$

One simple policy would be to move to state s' with largest $v_\pi(s')$. It can be shown that:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')], \forall s \in S \tag{6.4}$$

Eq. 6.4 is called the Bellman equation for v_π. This recursive equation relies on the sometimes unknown quantity $p(s',r|s,a)$). It is conceptually useful to derive algorithms to compute exactly or approximate $v_\pi(s')$.

State-action value $q_\pi(s,a)$

Another useful quantity is $q_\pi(s,a)$, which evaluates an action in a particular state:

$$q_\pi(s,a) = E_\pi[G_t|s_t = s, a_t = a] \tag{6.5}$$

This yields the Bellman equation for $q_\pi(s,a)$:

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,\pi(s))[r + \gamma q_\pi(s',\pi(s))], \forall s \in S \tag{6.6}$$

Note that $v_\pi(s)$ is a weighted sum of $q_\pi(s,a)$ over all possible actions.

A policy can then be defined *e.g.* by taking the maximum q value over all allowed actions. Similarly as before, this new Bellman equation is conceptually useful to derive algorithms to compute exactly or approximate $q_\pi(s,a)$.

Optimal policy and Bellman optimal equations

Solving a RL task means searching a policy that achieves a best reward over the long run. Such a policy is called optimal policy π^*, it must satisfy

$$v_{\pi^*}(s) = \max_\pi v_\pi(s), \forall s \in \mathscr{S} \tag{6.7}$$

The Bellman optimal equations are:

$$v_{\pi^*}(s) = \max_{a \in \mathscr{A}(s)} \sum_{s',r} p(s',r|s,a)[r + \gamma v^*_\pi(s')] \tag{6.8}$$

$$\pi^*(s) = arg \max_{a \in \mathscr{A}(s)} \sum_{s',r} p(s',r|s,a)[r + \gamma v^*_\pi(s')] \tag{6.9}$$

$$q_{\pi^*}(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{b \in \mathscr{A}(s)} q_{\pi^*}(s',b)] \tag{6.10}$$

$$\pi^*(s) = arg \max_{a \in \mathscr{A}(s)} q_{\pi^*}(s',a)] \tag{6.11}$$

$$v_{\pi^*}(s) = \max_{a \in \mathscr{A}(s)} q_{\pi^*}(s,a) \tag{6.12}$$

They express the fact that the value of s under the optimal policy must equal the expected value of the best action in s. Once the value of the optimal policy is computed, the optimal policy can be recovered by always choosing the action with the highest value. The importance of Bellman equations in RL lies in the fact that they allow us to compute the state value (e.g. s) from the value of other states (e.g. s'). This offers the possibility for iterative approaches for calculating the value for each action, the so-called $q-$value, used in **Q-learning** methods, which we use in the thesis.

6.1.2 Tabular RL algorithms

In this section we present a first taxonomy of RL algorithms concerning small discrete action spaces, for which the $q-$values of state-action pairs can be exhaustively estimated.

Such methods often iterate two steps policy evaluation and policy improvement. How we do these two things depends on the RL task. In general, RL algorithms can be classified based on these facts:

- Is the environment model known? Yes → Model-based (Dynamic Programming (DP)); No → Model-free (Monte Carlo (MC) and Temporal Difference (TD))
- Is a new estimation based on previous estimations? Yes → Bootstrapping (DP and TD); No → No bootstrapping (MC)

To put things in context, we briefly review these various approaches, although we are mostly concerned with function approximation methods developed in the next section (Section 6.1.3).

Model-based: Dynamic Programming (DP)

Dynamic Programming (DP): DP computes the optimal policy given the model through bootstrapping. There exists two approaches:

1. **Policy Iteration = Policy evaluation + Policy improvement:**

    - **Policy evaluation:** Using Bellman equation as a update rule:

    $$v_{k+1}(s) = \sum_{a} \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_k(s')] \tag{6.13}$$

    will converge to the true value $v_\pi(s)$ when the number of iterations is large enough. Note that v_{k+1} is estimated on the basis of previous estimations for its successor states $v_k(s')$. This is called bootstrapping.

    - Policy improvement: Make a new policy π' that is better than the current one π, by acting greedily with respect to the value function of the current policy:

    $$\pi'(s) = arg\max_{a} q_\pi(s,a) = arg\max_{a} \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')] \tag{6.14}$$

    - **Policy Iteration:** Once a policy π 's value v_π is estimated, we can use it to improve π and yield an better policy π'. This process is called policy iteration, and will converge to the optimal policy π^* with its value v_{π^*}.

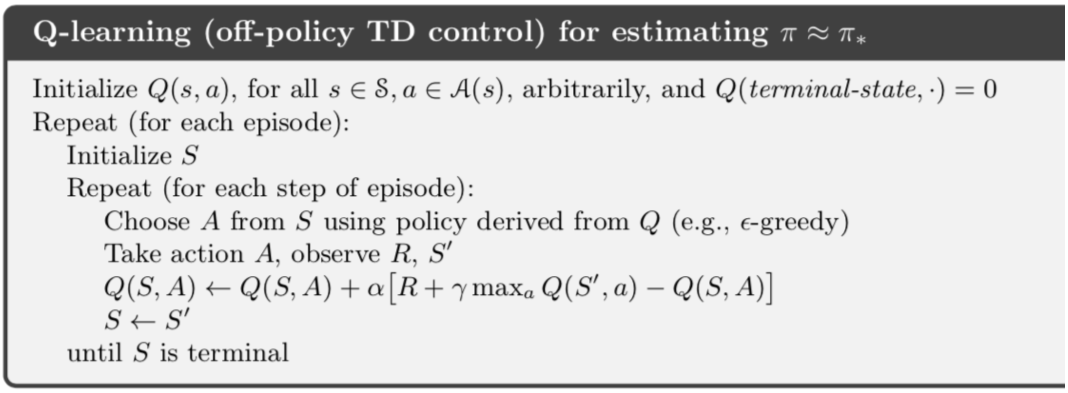
2. **Value Iteration = extreme case of Policy Iteration:** In policy iteration, the policy improvement occurs only after the current policy evaluation converges. Actually, we don't need to wait to the exact evaluation convergence. One extreme case is when policy evaluation is stopped after just one sweep, this is called value iteration.

Model-free

We introduce two principled algorithms in model-free RL:

- **Monte Carlo (MC)**: When we don't have any model of the environment, one possible workaround is to learn from experiences. Suppose we want to estimate $v_\pi(s)$, we can generate a set of episodes passing through s by following π, then average over all returns we get from these episodes. When the number of such episodes is large enough, the average actually approaches the true value of $v_\pi(s)$. This is what Monte Carlo methods do. Policy evaluation and policy improvement in MC is done only after one episode ends (episode-by-episode basis).

- **Temporal Difference (TD) and Q-learning**: Temporal difference (TD) learning is a class of model-free RL methods which learn by bootstrapping from the



```
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
   Initialize $S$
   Repeat (for each step of episode):
      Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
      Take action $A$, observe $R$, $S'$
      $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
      $S \leftarrow S'$
   until $S$ is terminal
```

Fig. 6.2 Q learning algorithm. Image source: [104].

current value estimates. Similar to MC, TD methods also learn from experience samples, but they update estimates based on other previously learned estimates, without waiting for the termination of episode. In the TD family, one popular algorithm is **Q-learning** [119]. This is a off-policy TD algorithm. The term 'off-policy' means in Q-learning, the optimal value q_* is directly approximated, independently of the actual policy, with an exploration policy (e.g. ε-greedy, *i.e.* random action is chosen with probability of ε, and the action $arg\max_a q(s,a)$ is chosen with probability of $1-\varepsilon$). Once the q_* is learned, the optimal policy can be easily recovered via Bellman equations. The Q-learning algorithm is given in 6.2. The algorithms we use to solve the REVEAL problems in this chapter will be a function approximation version of Q-learning.

6.1.3 Function approximation methods for larger action spaces

DQN, Double DQN

In the previous section, we introduced RL algorithms in "tabular situations" for which $q-$values can be estimated separately for each state-action. However, In our cases of interest, the action space $\mathscr{A}$ is too large to learn each action value in each state separately (e.g. $|\mathscr{A}| = 49$ in MNIST-patch, $|\mathscr{A}| = 163$ in meta-learning with CEP.). Instead, we use a parametric function to approximate those values. In particular, in Q-learning the state-action values $q(s,a)$ are parameterized as $\hat{q}(s,a;\theta_t)$ and updated using fitted Q-iteration:

$$\theta_{t+1} = \theta_t + \alpha(Y_t^q - \hat{q}(s_t,a_t;\theta_t))\nabla_{\theta_t}\hat{q}(s_t,a_t;\theta_t)) \tag{6.15}$$

where the target is similar to that in the tabular case:

$$Y_t^q = r_{t+1} + \gamma \max_a \hat{q}(s_{t+1}, a; \theta_t) \tag{6.16}$$

The algorithm minimizes the loss $L(\theta) = (\hat{q} - Y_t^q)^2$ via gradient descent.

A deep Q network (DQN) uses a neural network as the Q value approximator. The parameters θ in this case correspond to the weights in the network. A milestone in RL is to use DQN to successfully solve a series of Atari games [76]. In this work, in order to stabilize the learning, they developed a method called "experience replay", a biologically inspired mechanism where the agent keeps track of the outputs of previous actions $\{s_t, a_t, s_{t+1}, r_{t+1}\}$, which are then sampled to train the network.

In our cases of interest, the action space is large. Due to the "max" operator in the Bellman equation that drives the agent to choose the action with highest estimated Q value, even with a ε-greedy, it is known that DQN suffers from the maximization bias, where it tends to overestimate the value functions. The maximization bias leads to divergences and unstable behaviour. We then use a Double DQN which was proposed to correct this drawback (DDQN [113]). The DDQN corrects this bias by instantiating two separate identical DQN networks and randomly swapping between them at each optimization update, one being used as the target, and one being used as the policy. The neural network architecture we used to implement the DDQN for meta-learning is shown in Fig. 6.1.

Other RL algorithms

There exist many other RL algorithms. Among them: REINFORCE [121], Natural Policy Gradient [60], Trust Region Policy Optimization (TRPO [95]) are policy gradient methods, where the policy π is parameterized and learned directly and the agent maximizes the rewards by taking actions with high rewards more likely; Actor-Critic methods [66] combine policy gradient and value approximation by splitting the model into two parts: one actor to choose actions based on a state, and one critic to estimate the Q values of the action; Deep Deterministic Policy Gradient (DDPG [71]) is an actor-critic approach for continuous actions; etc. These are possible algorithms that we are going to try in future work to solve the meta-learning problems. In this work we focus on DDQN because it is reportedly easiest to deploy in applications, other methods requiring elaborate hyper-parameter tuning.

6.2 Toy REVEAL examples

In Chapter 3, various REVEAL games were introduced: the 1D meta-learning problem; revealing a bi-color segment to find its first black element; revealing a MNIST image to find its brightest patches (3.2.2); etc. In this section, we are going to present RL solutions to some of them. We begin with toy examples having similarity with the our meta-learning problem of interest, then move the to 1D meta-learning problem.

6.2.1 Reveal a bi-color segment to find its first black element

This toy game was created to serve as a simplified version of the 1D meta-learning problem.

Game setting

The agent initially faces to a fully masked 1D bi-color segment of length l, in which elements are white until some random position, after which all elements are black.[1] The position of the first black element (transition from black to white) is sampled randomly (Top panel of Figure 6.3). The goal of the agent is to reveal and predict the first black element. The state is the partially revealed segment. At each time step, the agent can choose to reveal an element in the segment. After each time step, the agent predicts the position of the first black element, the episode ends with a reward of 1 if the prediction is right, otherwise the agent fails after l steps. Before the termination of the episode, the agent receives a negative reward of 0.1 for revealing at each time steps; this is to help the agent learn to shorten the episode.

Double DQN architecture and Results

The RL agent has a double DQN architecture. It consists of two identical DQNs as shown in Fig. 6.1.The bottom panel of Figure 6.3 shows the learning curve of the RL agent compared with the well-known optimal solution which is binary search. We see the agent learns to perform the optimal search after $\sim$ 22000 episodes of training. Succeeding in this toy segment game gives us hope to tackle the more complex 1D meta-learning problem. This small example allowed us to perform a sanity check of our algorithm implementation.

[1] In an analogy with meta-learning, white is supposed to represent bad algorithms and black good ones.

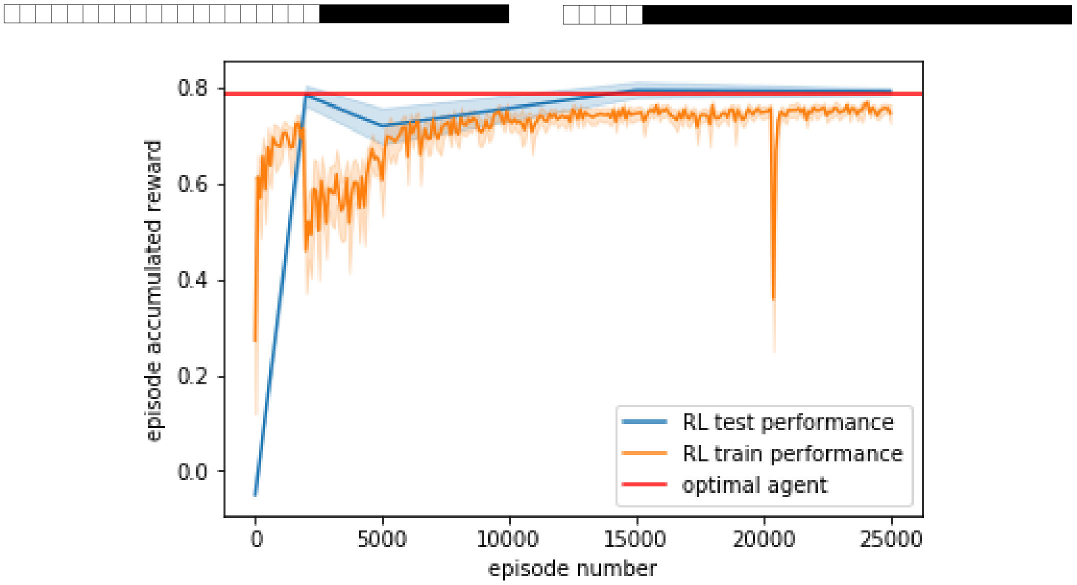


Fig. 6.3 **‘Segment’ REVEAL game**. We show results of the game in which the board is one-dimensional and finite. The goal is to find the position of the transition of black and white. This game has an optimal non-RL agent (no training at all), which performs iterative Dichotomic search. TOP: Two instances of the ‘segment’ REVEAL game. BOTTOM: graph of accumulated reward over the time of an episode as a function of the episode number, we consider 25,000 episodes. Test performance is average over 15 test datasets. At some point the blue line is going above the red line because the optimal agent is optimal only when averaging over all possible datasets, here we tested only on 15 datasets. During the RL training, we do ε-greedy for exploration, which explains the occasional drops in reward. Test performances are more stable (no exploration).

6.2.2 Revealing digits progressively to find their brightest patches

We designed a REVEAL game analogous to our meta-learning problem, which uses real-world data and provides intuitive visual feed-back. This game was inspired by the “Natural Image” games proposed in [122] and described in Section 3.2.2. The goal of the original game was modified from predicting the class of a handwritten digit by uncovering progressively its pixels to searching for its brightest patches in as few steps as possible. This is to make the whole setting more similar to a 1D meta-learning problem: one (flattened) image is equivalent to one dataset, and searching for the brightest patches is equivalent to searching for the best algorithm.

Game setting

We use the MNIST handwritten digit database [68] to create the REVEAL game environment where the agent initially faces a fully masked image. This masked image can be any class of MNIST, i.e. from 0 to 9. The state consists of the partially revealed 32×32 image and its already revealed positions. At each time step, the agent can choose any position in the image to reveal a square window w of the image. The goal of the agent is to find one of the 3 [2] brightest patches in the image, where each patch corresponds to a window-sized square. The brightness of a patch is computed as the average pixel value in the patch window. The episode ends when the goal is achieved and the agent receives a reward of 1, otherwise it fails after a maximum $\frac{32\times32}{w}$ steps. During the game, the immediate reward of each step is the combination of step cost [3] and the improvement on the best brightness found so far.

RL CNN agent *vs.* upper-baseline supervised CNN agent

We have developed 2 agents to play the game: a **RL agent**, which learns a policy from trial and errors, and a supervised CNN agent (**CCN upper-baseline**), which learns to imitate a hard-coded policy based on the full image (not available for learning to the RL agent).

- **RL agent:** we use a Double DQN architecture to train the RL agent. As introduced before, the Double DQN method includes two networks with identical architecture shown in Figure 6.4, which are randomly swapped at each optimization update to serve as target and policy network. The policy network is used to generate actions. The architecture contains a CNN identical to that used in the CNN supervised upper-baseline, followed by an extra linear layer before generating the Q values. The input to the RL agent is the state, i.e. already visited positions plus the partially revealed image. The output of the RL agent is the action that has highest estimated Q value (this is the case at test time; during training, we use an ε-greedy policy forcing the agent to act randomly with probability ε and follow the DQN policy otherwise, where ε decays with the number of training episodes. This allows the agent to explore more at the beginning of learning. Once training is done, the CNN weights are "frozen" for the testing and one of the two CNNs is used to predict action values ($\varepsilon = 0$).
- **Supervised upper-baseline CNN agent:** This agent uses a CNN (contains 3 convolutional layers and ReLU nonlinearities) to predict the full image from a partially revealed image. That is, during training, the true full image is given as target at each time step for the agent to learn. In the first few actions, since only a

[2] The reason of having 3 targets instead of 1 is that sometimes there are more than one patches with the same brightness in the image.

[3] the step cost is $\frac{-1}{l_{ep}}$, where l_{ep} is the length of one episode, which, in our setting, equals the maximum number of steps $\frac{32\times32}{w}$.

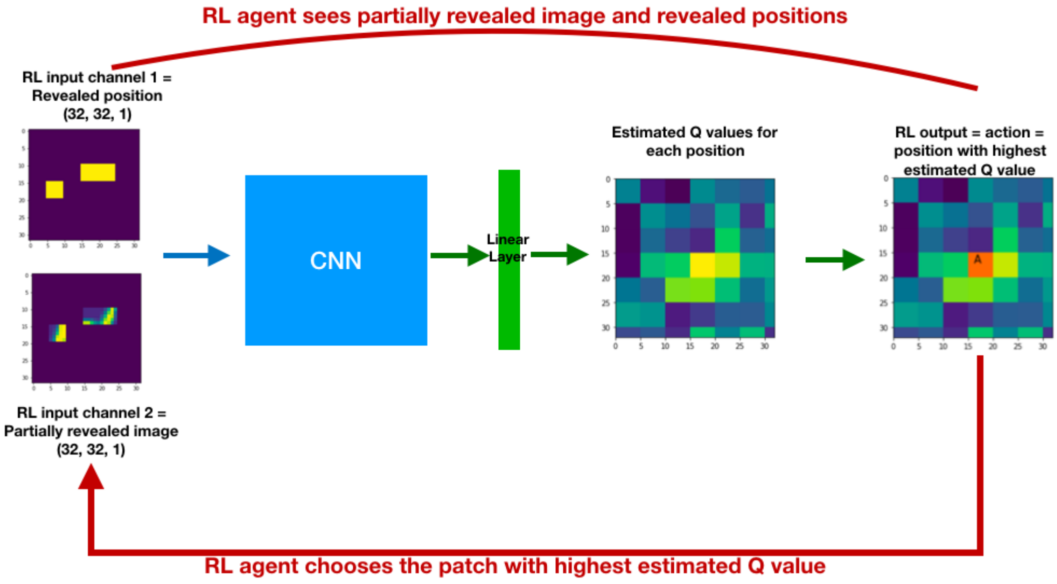

RL agent sees partially revealed image and revealed positions
RL input channel 1 = Revealed position (32, 32, 1)
RL input channel 2 = Partially revealed image (32, 32, 1)
CNN
Linear Layer
Estimated Q values for each position
RL output = action = position with highest estimated Q value
A
RL agent chooses the patch with highest estimated Q value

Fig. 6.4 **RL DQN architecture used in MNIST brightest patch game**: In the experiments we use a "double DQN architecture", which includes two CNNs used in alternance during training, to stabilize the learning procedure. The CNN gets the partially revealed image as input and uses one of the two CNNs to estimate the Q-value of each action. At test time, either CNNs can be used to select the most promising action (deterministic policy). During training, we use an ε-greedy policy: one CNN is used to select the action; the parameters of the other are updated using the Q-learning algorithm.

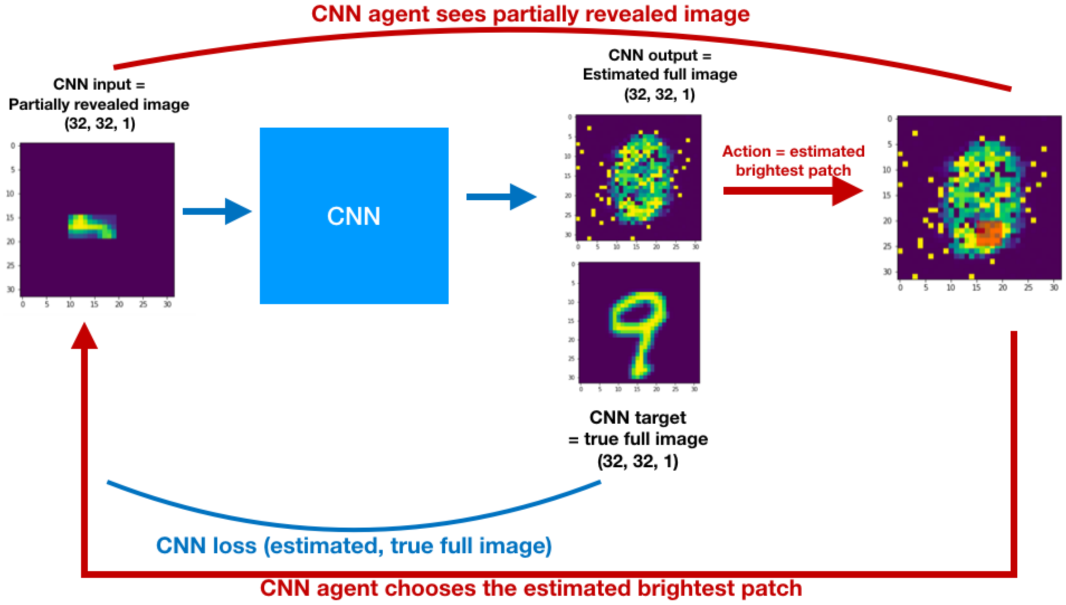

CNN agent sees partially revealed image
CNN input = Partially revealed image (32, 32, 1)
CNN
CNN output = Estimated full image (32, 32, 1)
Action = estimated brightest patch
CNN target = true full image (32, 32, 1)
CNN loss (estimated, true full image)
CNN agent chooses the estimated brightest patch

Fig. 6.5 **Supervised learning upper-baseline CNN architecture, used MNIST brightest patch game**: The agent gets the partially revealed image as input, uses a CNN to learn to predict the full image, and selects the estimated brightest patch that has not been revealed yet as its action.

few bright pixels will be visible, predicting the full image is very difficult, hence, the CNN agent is expected to predict a ‘mixture’ of all images, which is bright at the center and dark everywhere else (as shown in Figure 6.5). The agent then chooses the patch with highest predicted brightness that has not been revealed yet. During testing, the CNN is frozen, the agent then uses this pre-trained CNN to estimate the full image and acts accordingly.

These two agents have some significant differences: (1) Full *vs.* Partial learning information: the RL agent never sees the underlying full image, it learns upon partial information, however, the CNN agent during the training is given the full image to learn, which allows it to build a strong ‘prior’ on how an ‘average’ image will be like, this is its first advantage; (2) Repeating actions: The CNN agent is hard-coded to choose actions only among those that have not been revealed yet [4]; the RL agent, in contrast, should learn through the reward not to repeat actions (because repeating actions will cause negative reward and make the episode longer). This is the second advantage of the CNN agent. These two properties make the CNN agent a very difficult upper-baseline to beat.

Even though the CNN is a very difficult baseline to beat, it is not impossible to beat. Indeed, its objective is to predict the full digit with a mean-square-error loss, not focusing particularly on the next action, whose goal is to uncover the brightest patch. In contrast, the RL agent gets reinforced to do well on that specific task.

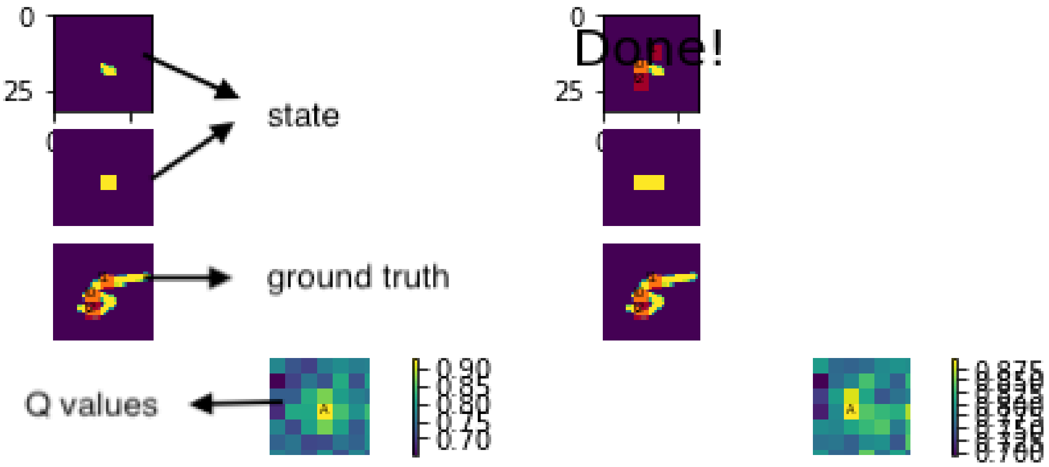
Results

The purpose of the experiments in to show that RL can approach or exceed the performances of the supervised upper-baseline CNN and show that the RL agent can learn difficult things like not visiting several times the same patch.

We first show two runs of our algorithms for illustrative purposes. In Figures 6.6 and 6.7 we show respectively the RL agent and the baseline CNN agent acting on a same test image after being trained for 2000 episodes. Two observations can be made:

- **RL agent learns not to re-visit patches:** While the CNN agent was programmed not to repeat itself (in a hard-coded manner), the RL agent had to LEARN not to revisit several times the same patch. Thus it is impressive that it terminates the game in only 2 steps, compared to CNN baseline agent which takes 5 steps.

[4] this is because the general prior that the CNN is expected to learn won’t change dramatically from one step to another, which means the action with the highest predicted brightness will stay the same for many steps, and the CNN will always choose the same action if repeated actions are allowed



(a) RL agent: $t = 1$ (b) RL agent: $t = 2$

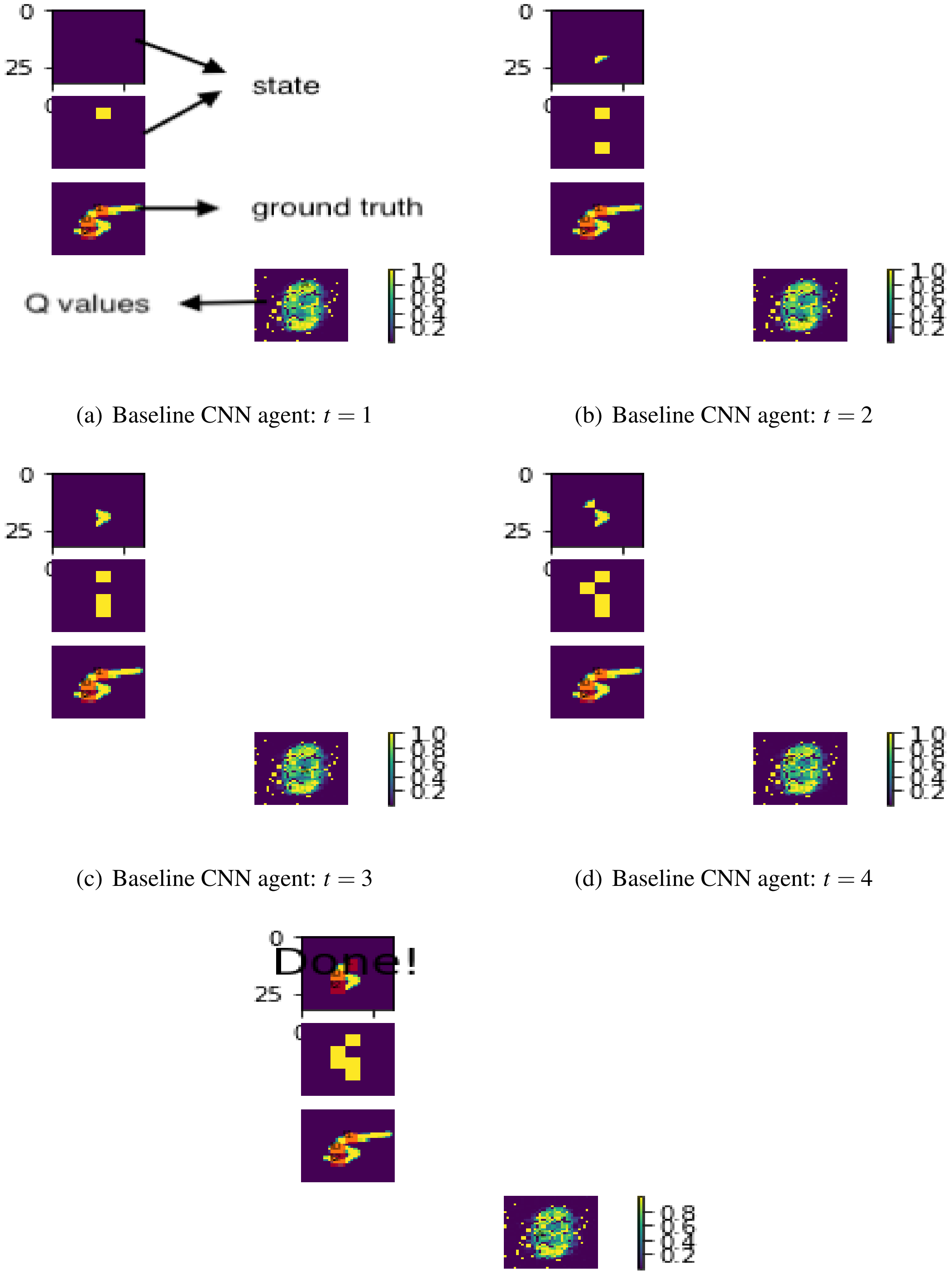
Fig. 6.6 One test episode of MNIST brightest patch game realised by a **RL agent that can repeat actions**.The agent takes 2 steps to terminate the episode.

- **RL agent is agile:** From the Q value images, we see that the RL agent changes a lot its “opinion” on what the next best action should be, conditionally on the state. In contrast, the CNN baseline agent keeps mostly the same Q value image.

These observations can be further evaluated quantitatively. In Figure 6.8, we show learning curves for the MNIST brightest patch problem. The metric of success is episode duration (the shorter the better; an episode ends when the brightest spot is found). On the x axis, each episode correspond to one image. On the y axis, the episode duration is the number of steps the agent takes to terminate the episode, so lower is better. The learning curves are averaged over 3 training runs. The shaded areas represent 68% and 95% confidence interval.

In Figure 6.8.a we show the training performance. Compared to the CNN baseline agent (green curve), we see that RL agents perform less well especially at the beginning (red curve). This is due to two factors: (1) the CNN baseline agent is programmed not to repeat actions; when this prohibition of repeating actions is imposed to the RL agent (violet curve), the performance improves, compared to the red curve. (2) RL agents perform exploration at the beginning of learning, due to the ε-greedy policy; as ε decays towards the end of the learning curve, the performance difference between RL and CNN baseline agent decreases.

Figure 6.8.b represents the “test time”. The agents are frozen each 100 training episodes and tested on 1000 images sampled from the separate test set. The learning curves shown are first averaged over those 1000 test images at each frozen point,



(a) Baseline CNN agent: $t = 1$

(b) Baseline CNN agent: $t = 2$

(c) Baseline CNN agent: $t = 3$

(d) Baseline CNN agent: $t = 4$

(e) Baseline CNN agent: $t = 5$

Fig. 6.7 Same test episode of MNIST brightest patch game realised by a **Baseline CNN agent**. The agent takes 5 steps to terminate the episode.

then averaged over 3 training runs. All agents perform a deterministic policy (*i.e.* the exploration is turned off for RL agents: $\varepsilon = 0$) during testing, this explains the better performance of RL agents at the beginning of the episodes, compared to that during the training time. The RL agrent is still allowed to repeat actions (red curve), while the CNN baseline isn't (green curve). Even in this case, we see the RL agent, after only 1000 of training episodes, achieves the same performance as the CNN baseline. If we prohibit the RL agent from repeating actions (violet curve), it beats completely the CNN agent from the very beginning.

represents two different behaviors: averaging over very short successful episodes and very long failed episodes for the RL agent (red curve), and averaging over longer but more stable episodes for the baseline CNN agent (green curve).

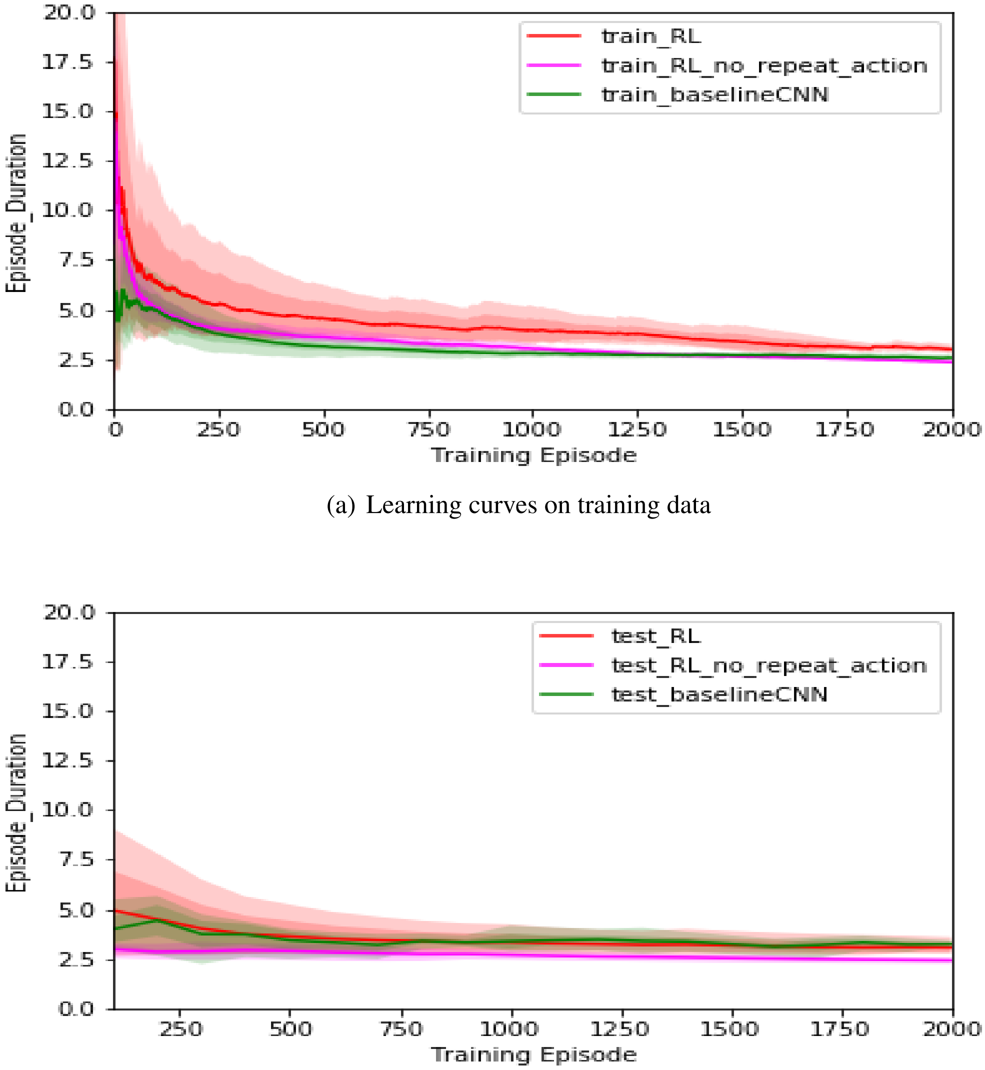
6.3 RL setting for 1D Meta-learning problem

In this section, we come back to the 1D meta-leaning setting, which is the focus of interest of this thesis. We recall that the objective of this problem is to find the algorithm with the highest performance score for ONE particular dataset (never seen before), i.e. the highest value in a new row of matrix S. We use meta-datasets: CEP, AutoML, OpenML, Statlog and Artificial introduced in Section 2.3 to benchmark RL against other methods. We treat the problem essentially as a collaborative filtering problem, *i.e.* predicting missing values in S from values already known. However, to help the agent tackle the cold-start problem (an empty row at the beginning of each episode), we also explore adding meta-features, as described in Section 2.3.2 by appending 16 meta-features to the learned features of the inner layer of our neural network, as shown in 6.1. Another particularity of our setting is to freeze our agents at test time and evaluate the policy on new "test datasets". Finally, our reward function allows us to monitor the tradeoff between effectiveness of the policy to find best performing algorithms and computational efficiency. All these various aspects are investigated in the experiments that we describe after providing some useful details on the experimental setting.

6.3.1 Experimental setting

RL framework and reward function

Figure 6.1.a illustrates the RL meta-learning system. If we translate the 1D meta-leaning setting to a RL framework, we have:



(a) Learning curves on training data

(b) Learning curves on test data

Fig. 6.8 **MNIST-patch Learning curves**. The episode duration is plotted as a function of the number of training episodes. (a) Episode duration for training datasets. (b) Episode duration for test datasets.

Environment is the entire meta-dataset matrix S (with datasets in row and algorithms in columns), not directly accessible to the agent, and a chosen row index d indicating on which dataset the agent is working. This selected row is our hidden "game board" and it is associated to a game episode.

Observation / State is the already revealed information of the current dataset $S(d,:)$, that is $S(d, revealed)$ if *revealed* is the list of indices of the already revealed algorithm performances.

Action Space includes all algorithm indices (positions in the current dataset/game board), *e.g.* for a dataset with 10 algorithms, the action space is $j \in \{1,2,3,\dots,10\}$.

Reward is attributed to each action as r_t to help the agent to refine its policy, it combines the improvement on performance score $|S(d,j_t)_{perf} - max_{k=0}^{t-1}(S(d,j_k)_{perf})|$ and cost $S(d,j_t)_{time}$ traded off by a hyper-parameter β:

$$r_t = |S(d,j_t)_{perf} - \max_{k=0}^{t-1}(S(d,j_k)_{perf})| - \beta S(d,j_t)_{time} \tag{6.17}$$

An interesting aspect of the game is to see how the agent's behavior changes with respect to the **score-time tradeoff** governed by hyper-parameter β in Eq. 6.17. β expresses the preference of the game designer and has a real-world meaning: when β is large, the evaluation time is more precious, and the agent will be penalized more for choosing a slow algorithm no matter how good its performance score is; when β is small, the time matters less, and the focus concentrates on score; when β is 0, only the score is considered.

Train-test split

Splitting data and measuring performances is performed in a classical machine learning way, at the meta-dataset level. We proceed in either one of two ways: single train/test split or 10-fold cross-validation. For a large (meta-)dataset like the CEP meta-dataset, we do a single train/test split. For other smaller (meta-)datasets, we do 10-fold cross-validation.

For ease of explanation, let us focus on a single train/test split. For instance, the whole CEP meta-dataset is split into a train set (70%) and a test set (30%). At training time, for each episode, we sample one dataset from the CEP training set; this is a 2x163 matrix including score and computational time for 163 algorithms. The initial row is empty, in each time step, the agent can move to all positions in the row, which reveals the score and computational time of the algorithm in that position. After each time step, the agent receives a reward that combines the improvement on the best score found so far, and the action computational cost (Eq. 6.17). The episode length is truncated at 20 steps (although its maximum value in principle could be as large as the total number of values to be revealed). Once the training phase is terminated, the agent's parameters are frozen to enter into the test phase in which the policy is used without further adaptation. New datasets not used during training are sampled from the test set, one at a time. The final best performance score after 20 steps is averaged over all test datasets to obtain the test score.

The agent's performance is evaluated by learning curves, *i.e.* performance as a function of number of training episodes, in two ways:

- **Training performance:** the accumulated reward during the current training episode (over 20 steps), averaged over 3 runs.

- **Test performance:** The final best performance score after 20 steps on a test dataset, average over all test datasets.

In what follows, we show only the test performance.

RL algorithm under investigation

We use a double DQN architecture as introduced in Section 6.1. The input to the network is the state, a two-dimensional vector representing the current partially revealed scores and computational time. An embedding of the state is then learned via the first network block, when meta-features are used, they are appended to the learned features. A second block network is then used to estimate the Q values.

Baseline algorithms

We compare the RL agent against two baselines:

- **Random search:** This is a stochastic method which makes random decisions every time when choosing an algorithm. Each selection is repeated 1000 times (100 times for the CEP meta-dataset for computational concerns), the median and quantiles are computed and shown in the results figures.

- **ActivMetaL:** (Chapter 5): This is a deterministic baseline. It chooses the best estimated performance each time. The ACTIVMETAL baseline is considered only when the meta-dataset is small (*i.e.* AutoML, OpenML, StatLog, Artificial) and when the computational time is ignored in the game (*i.e.* $\beta = 0$).[5]

[5] This is because the original ACTIVMETAL: (1) only uses the algorithm performance as its single learning source, and its final performance is computed based solely on that, it does not take into account computational time; (2) requires matrix factorization for each selection step, it becomes computationally non affordable when the meta-matrix is large.

6.3.2 Results on CEP

In this section, we first show experimental results for the CEP meta-dataset. In this meta-dataset, each "task" is a univariate multi-class classification problem, *i.e.* the target variable must be predicted from a single variable. Hence, it was easy for us to produce a large meta-dataset, each learning problem (training and testing a single variable classifier) being fast to evaluate. This allowed us to generate a meta-dataset lending itself to benchmarking RL algorithms in favorable conditions, since RL agents are known to be "data hungry".

There are a few experimental settings specific of the CEP dataset:

- Because the meta-dataset is large, we do not need to resort to cross-valiation and perform a single train/test split: 70% of the datasets for training and 30% for testing, resulting in 6025 datasets in the (meta-)training set and 2583 datasets in the (meta-)test set.
- We fix the episode length to be 20 *i.e.* each episode consists of 20 algorithm selection steps. This number 20 is small compared to the total number of algorithms 163. The agents thus need to learn to search efficiently.
- Rather than representing score as "best algorithm performance so far", in an effort to spread out algorithm that are near ties, we sometimes use "**reverse algorithm rank**" as score/immediate reward. The ranks are computed according to the ground truth: the best algorithm has rank 162 (we have reversed the ranking, so higher is better, because we have totally 163 algorithms and we count from 0). This is thus a way to represent the performance score of the algorithms.

Ability to find high-performance algorithms in a short episode

In this experiment, we turn off the performance-time tradeoff ($\beta = 0$) and compare the performance of the final best algorithm found by RL agents (with and without meta-features) with that found by the random search baseline. Figure 6.9 shows the result averaged over the whole test set. We see that both RL agents, which are deterministic, work at least as well as the random search baseline. This is a very positive result, because the random agent, being repeating 100 times, actually explore the action space much more than the RL agents, who explore only a small portion (20 steps / 163 possible choices). The RL agents furthermore have better searching initialization compared to the random average. Surprisingly the RL agent without meta-features achieves the best final performance. This remains to be further investigated, but could be fortuitous, given the error bars.

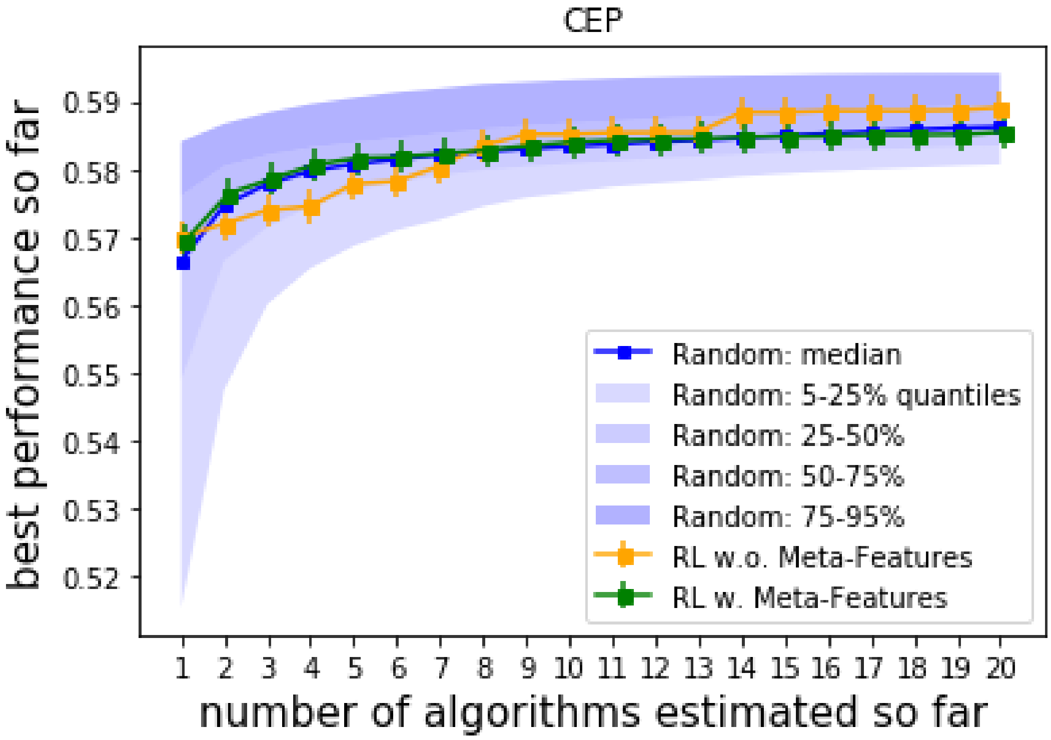


Fig. 6.9 **CEP meta-dataset: RL vs. Random search** on the problem of finding high-performance algorithms at test time a.k.a 1D meta-learning. The RL agents (with and without meta-features) show performances that match random search. The RL agent without meta-features achieves the best final performance. One single train/test split is used. The performance error bars of RL agents tested on 2583 test datasets are the standard errors. The quantiles for random search (no training) are computed for each test dataset using 100 repeats, then averaged over all datasets.

Score *vs.* time tradeoff

How does the RL agent change its behavior according to *performance score* vs. *computational time* tradeoff parameter β? To answer this question, we have trained 3 RL agents for 20000 episodes (each episode corresponds to one dataset randomly sampled from the (meta-)training set) without meta-features acts, in the environment with different β values $\{0.1, 0.05, 0.01\}$.

The importance of computational time increases with β (Equation 6.17). Figure 6.10 shows how the ranking of the best algorithm found so far varies with the accumulated evaluation time in an averaged test episode.

Because the fixed episode length 20 is less than the total number of algorithms 163, the best rank the agents found can be far from 162. Higher is better for these reversed ranks. The ranks and time are averaged over 100 test datasets. We see that, the agents with larger β have much less accumulated evaluation time, but the best algorithm found within this small time interval is also worse, which is actually a good behavior because in this case, time is more precious, finding a 'good enough' algorithm in a short period of evaluation time is actually what is appreciated. The agents with smaller β (e.g. orange), however, have learned to choose algorithms that require more evaluation time but give better ranking, which is also an expected behavior, because now the time consumed is penalized less in the reward.

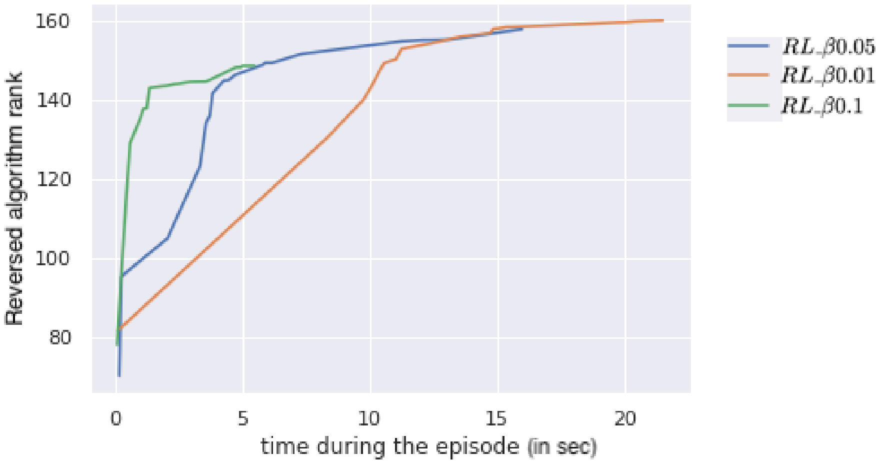


Fig. 6.10 **CEP meta-dataset: Performance - time trade-off** on the problem of finding high-performance algorithms at test time a.k.a 1D meta-learning. On the x axis we show accumulated algorithm time during a test episode. On the y axis we use "reverse algorithm rank" as performance measure (instead of algorithm best performance so far); this emphasizes near ties, but higher values are better, as before. The curves are averaged over only 100 test datasets. The RL agents show a good trade-off ability: when time is precious (large value of β), time-efficient but less score-effective algorithms are chosen (green curve); otherwise, slow algorithms with better score are chosen (orange curve). The standard error for the rank of the final algorithm is less than 1.5.

Does RL need a lot of training examples?

We used the CEP dataset to allow us to benchmark data hungry RL algorithms without being limited by small-size meta-datasets. However, a legitimate question is: how many training (meta-)samples are needed to obtain good performance. To investigate this, we have varied the size of the (meta-)training set from which we sample the datasets for training the agent, to see how this affect the agents' performance at test time. Figure 6.11 shows the results: the test performance improves in with the size of training set, up to 500 training datasets. After that, the test performance remains almost the same (actually slightly decreases for reasons that we have not fully elucidated so far and might have to do with our RL algorithm implementation). Hence, at least on this task, RL learning is possible with a relatively modest number of training samples, which raises hope that meta-learning with RL is feasible. We will now investigate it on small but more realistic (meta-)datasets.

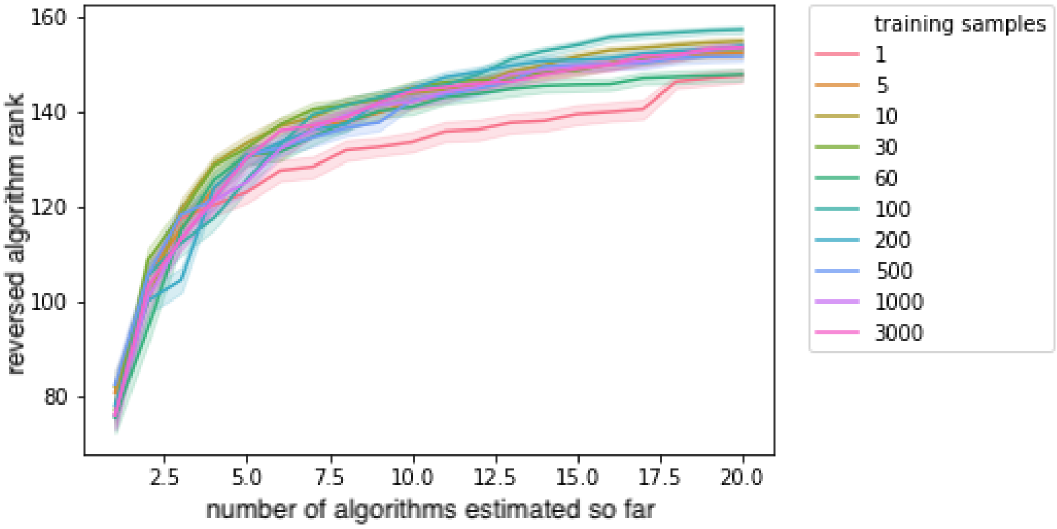


Fig. 6.11 **CEP meta-dataset: Performance as a function of number of training examples**.Performance on the y axis is reverse rank of best algorithm found so far on test datasets. x axis is the number of algorithm tested so far in an episode. We show influence of training set size on learning curves. We see that test performance improves when more training datasets are used up to about 500 datasets.

6.3.3 Results on other meta-datasets

In this section, we benchmark RL on real-world meta-datasets against various agents. This allows us to investigate the behaviour of RL in more realistic cases, but, unfortunately, with (meta-)datasets, which are small and do not have algorithm computational time records. One advantage of having small datasets is that it allows us to make comparisons against ACTIVMETAL (an effective but slow algorithm that we proposed in Chapter 5).

Experimental setting

The RL architecture is the same as in the CEP experiments. We report all results as learning curves, in which on the x axis we have the number of actions performed so far by the agent (number of algorithms tried on the given dataset) and on the y axis the best algorithm performance so far.

The differences in experimental setting compared to previous experiments on CEP data are as follows:

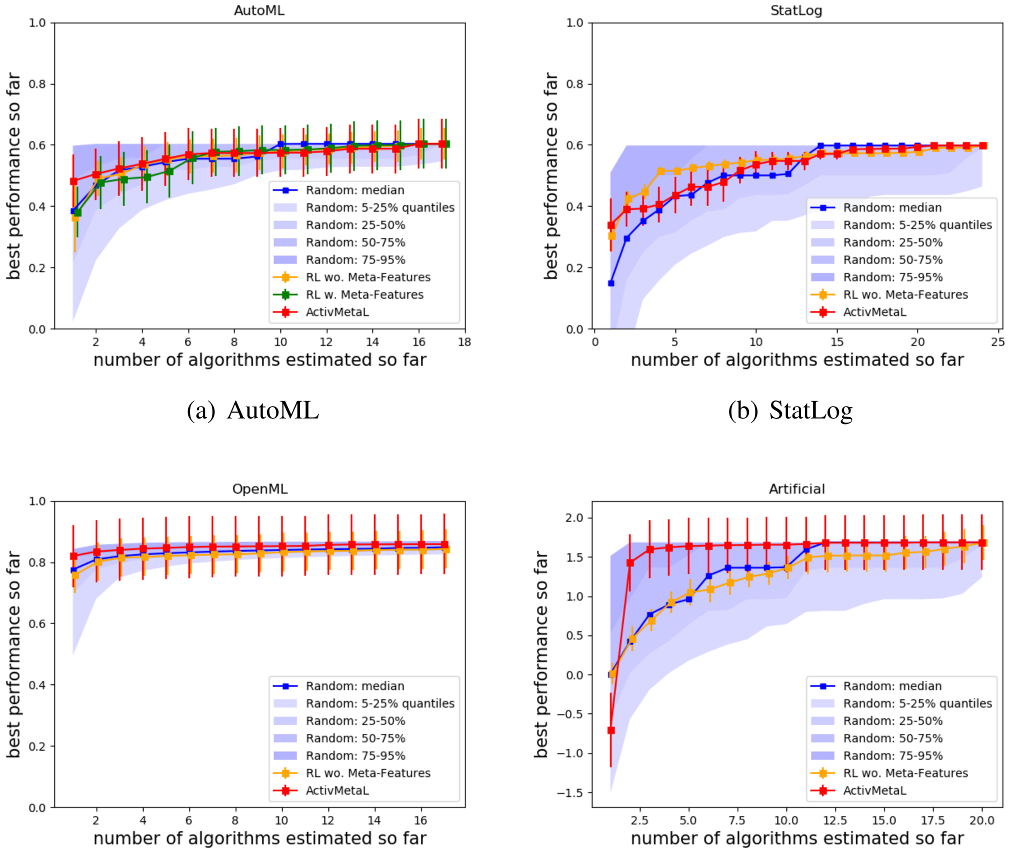
- **Baselines:** RL agents will be compared with two baselines: one deterministic (ACTIVMETAL), and one stochastic (Random search).
- **No efficacy/efficiency tradeoff.** We focus on algorithm performance score ($\beta = 0$ for all experiments in this section, individual algorithm computational time not available).
- **Cross-validation.** Since real-world meta-datasets are small, we evaluate our agents using 10 fold cross-validation at the (meta-)dataset level, instead of the 70/30% (meta-)train/test split used in the CEP meta-dataset.
- **More repeats on random search.** Random search agents are averaged over 1000 repeats (instead of 100 in the CEP meta-dataset). As before, random search repeats allow us to compute quantile learning curves for each test dataset; they are then averaged over all test datasets.
- **Test performances only.** The performances reported in this section are those on test datasets (in each fold).
- **Error bars.** Error bars are standard deviations of cross-validation performances computed as: $\sigma(j) = \frac{1}{10}\sqrt{\sum_{i=0}^{9}(\mu^{(j)} - \mu_i^{(j)})^2}$ for j^{th} point on the learning curve, where $\mu^{(j)}$ is the mean over all test datasets for j^{th} point, and $\mu_i^{(j)}$ is the mean over datasets in the i^{th} fold for j^{th} point.

Note that the performance scores of matrix S do not need to be between 0 and 1 nor be normalize in any particular way.

Results

The learning curve results are presented in Figure 6.12. The results confirm previous previous experiments on CEP data, which showed that RL performs at least as well as does Random Search on average. Since RL policies are deterministic (selection of the next algorithm with the best Q-value), they do not suffer as much variance as Random Search, which is a distinctive advantage.

In these experiments, we can use ACTIVMETAL as baseline (since the datasets are of smaller size than CEP). This allows to make comparisons against a strong algorithm, and we are pleased to see that RL fares well, except on the Artificial dataset. ACTIVMETAL strongly outperforms RL on Artificial data. This can be exaplined by the fact that Artificial data was constructed from a matrix factorization, hence they are well-suited to be solved with an algorithm like ACTIVMETAL based on matrix factorization.



(a) AutoML

(b) StatLog

(c) OpenML

(d) Artificial

Fig. 6.12 **RL vs. ActivMetaL and Random search** on the 1D meta-learning problem of finding high-performance algorithms on test datasets. The RL agents (with or without meta-features) show performances that match or exceed random search. On real-world datasets, RL fares well, and, given the error bars does as well or better than ActivMetaL. The error bars are 10-fold cross-validation standard deviation (see text). The points have been slighly translated for ease of visualization, so they do not overlap.

6.3.4 Computational considerations

Compared to ActivMetaL, the main interest of using RL to solve the meta-learning MDP lies in its computational advantage at test time. In ActivMetaL, at test time, each selection step requires one re-factorization of the entire matrix S, which costs $\mathcal{O}(DN^2)$ if $D > N$ ($\mathcal{O}(ND^2)$ otherwise, D is the number of datasets and N the number of algorithms (lines and columns of S). In contrast, RL training delivers a policy, which can be applied to a new dataset at test time, without further training. Thus, one selection step requires only one forward pass through the neural network, which costs $\mathcal{O}(N+N_h)$, where N_h is the number of hidden units. When the meta-dataset gets

Table 6.1 **Computational effort comparison between** ACTIVMETAL **and REVEAL**. D =number of datasets in S, N =number of algorithms in S. $N_{episode}$ =number of training episodes with RL. $N_{actions}$ =number of possible actions. TrCostNN = (Tfwpass+Tbwpass)$\times(N+N_h)$, where Tfwpass = time of a forward calculation per weight, *i.e.* one multiply-add; Tbwpass = time to update a weight in backprop, *i.e.* compute gradient, per weight; N and N_h are respectively number of input weights (it equals the number of algorithms) and number of hidden units. TeCostNN = Tfwpass $\times(N+N_h)$

Method	Dim. of S	Training effort	Test time effort
ACTIVMETAL	$D \times N$	None	Factorize matrix S $\mathscr{O}(DN^2)$ if $D > N$; $\mathscr{O}(ND^2)$otherwise
RL	$D\times$ N	$N_{episode} \times N_{actions}\times$TrCostNN	TeCostNN =Tfwpass$\times(N+N_h) = \mathscr{O}(N+N_h)$

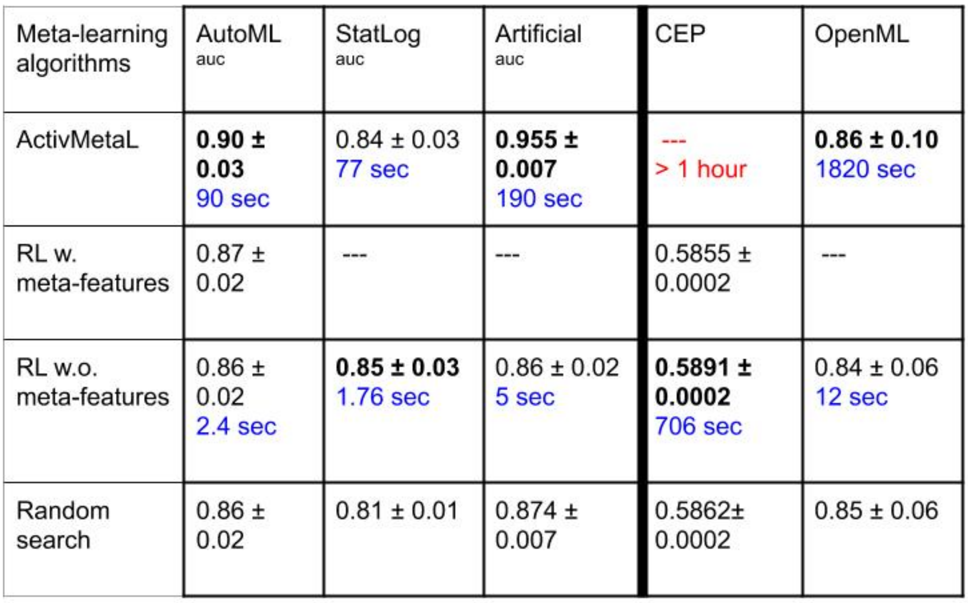
larger, the advantage of using RL becomes more significant. Table 6.1 summarizes the orders of magnitude for the computational efforts of either method.

Figure 6.13 summarizes the performance in test time of all learning agents we have investigated so far: ACTIVMETAL and RL which are both deterministic and Random search which is stochastic. The performance is computed using two metrics: (1) Area under the learning curve (AUC) for AutoML, StatLog, Artificial for which the episode length equals the total number of algorithms, hence, without repeating actions, all agents achieves the same final algorithm performance score, in this case, agents with higher AUC have more efficient searching strategies; (2) Final algorithm performance score for CEP and OpenML for which the episode length is smaller than the total number of algorithms, hence, agents achieving higher final algorithm performance at the end of the episode are more efficient in searching. The table also shows in blue the corresponding computational time [6] for each result for ACTIVMETAL and RL. We remain that this is the meta-learning time at test time. As analyzed in previous sections, ACTIVMETAL outperforms more often than RL (and Random search) in performance with small margins, while RL is significantly faster.

6.4 Summary of the RL chapter

In this section, we have explored solving meta-learning problems with RL techniques to learn a policy. Our preliminary experimental results demonstrate that: (1) with either collaborative filtering (ACTIVMETAL) or RL we have matched the performances that Random Search achieved *on average*; this is an important achievement because

[6] Recorded using a 2 core x86_64 machine.



Meta-learning algorithms	AutoML auc	StatLog auc	Artificial auc	CEP	OpenML
ActivMetaL	**0.90 ± 0.03** 90 sec	0.84 ± 0.03 77 sec	**0.955 ± 0.007** 190 sec	--- > 1 hour	**0.86 ± 0.10** 1820 sec
RL w. meta-features	0.87 ± 0.02	---	---	0.5855 ± 0.0002	---
RL w.o. meta-features	0.86 ± 0.02 2.4 sec	**0.85 ± 0.03** 1.76 sec	0.86 ± 0.02 5 sec	**0.5891 ± 0.0002** 706 sec	0.84 ± 0.06 12 sec
Random search	0.86 ± 0.02	0.81 ± 0.01	0.874 ± 0.007	0.5862± 0.0002	0.85 ± 0.06

Fig. 6.13 **Performance and computational effort comparison between** ACTIVMETAL **and RL for all 1D meta-learning datasets in test time**. The performance is measured by the Area under the learning curve when marked ‘auc’, and by the final algorithm performance achieved otherwise. The computational time is marked in blue.

Random Search has a huge variance (it performs sometimes well but sometimes very bad) while our *deterministic* policies perform well consistently; (2) We are happy that we could match with RL the performances of the strong collaborative filtering baseline and we are hoping that with some more efforts we can exceed it; (3) Even if we do not exceed it, the RL policy is still interesting because it is computationally advantageous at test time since it does not require matrix factorization at every time step.

These promising results encourage us to go even further in the future and explore, for example, 2D meta-learning, that is the simultaneous selection of datasets and algorithms. Another avenue of research would be to explore continuous hyper-parameter spaces. We will give some more details about such research directions in the following chapter.

Chapter 7

Discussion and further work

This chapter puts in perspective the work, which was performed in this thesis. First we place our work in the Liu-Xu framework that was recently proposed [72]. Then we indicate further directions of research that would be a logical continuation of our effort and that we hope other students or researchers be interested in pursuing.

7.1 The Liu-Xu $\alpha - \beta - \gamma$ framework

Liu and Xu introduced a novel generic mathematical formulation of AutoML, resting on formal definitions of hyper-parameter optimization (HPO) and meta-learning [72]. They decomposed various algorithms and show that HPO methods such as most works on Neural Architecture Search [124, 89] don't really address the AutoML problem more than "classical" machine learning algorithms, while meta-learning does. Table 7.1 summarizes the involved classes of algorithms.

In all cases, one assumes that an "autonomous agent" is delivered to solve an AutoML problem. Briefly, the levels consist of:

- **α-level** imposing that the "autonomous agent" consist of a **pre-trained model, solely equipped with a "test" method** making predictions on new examples, hence, implicitly, training data and unlabeled test data (in the supervised learning setting) are provided outside of the scope of evaluation to the algorithm designer;

- **β-level** imposing that the "autonomous agent" include **training and test methods**, hence both training and unlabeled test data are solely available to the autonomous agent, NOT to the algorithm designer;

- **γ-level** imposing that the "autonomous agent" include also a **meta-learn method** to be used for training it to deliver a **β-level** algorithm. Thus while in α and β-level settings the agent is trained and tested independently on all tasks, in the

Table 7.1 **Supervised learning illustration of the three-level formulation.** An algorithm's level is entirely determined by its type of *input* and *output*. For a given task, finding a good α-level algorithm is the ultimate goal. γ-level algorithms exploit data from *all past experience*, in the form of a "meta-dataset", to allow us to select a better β-level algorithm, which in turn exploits the dataset of a given task to produce an α-level algorithm by training.

Level	Input	Output	Examples	Encoded by
α-level	sample or example (e.g. an image)	prediction of label (e.g. 'dog' or 'cat')	heuristically hard-coded classifier or already trained classifier	parameters, hyper-parameters (if any) and meta-parameters (if any)
β-level	task/dataset (e.g. MNIST, CIFAR-10)	α-level algorithm	learning algorithms (e.g. SVM, CNN); HPO algorithms (e.g. grid search cross-validation, SMAC [56], NAS [124])	hyper-parameters and meta-parameters (if any)
γ-level	meta-dataset (e.g. OpenML [115])	β-level algorithm	meta-learning algorithms (e.g. meta-learning part in Auto-sklearn [36]); **algorithms from this thesis.**	meta-parameters

γ-level setting, the autonomous agent can learn from past tasks to perform better on next tasks (**meta-learning**).

One of the insights drawn from the Liu-Xu framework is that hyperparameter optimization (HPO) methods (*e.g.* Neural Architecture Search [123, 89]) do not really address fully the AutoML problem more than "classical" machine learning algorithms, in the sense that they remain at the β-level (no meta-learning). In this thesis, we clearly addressed the γ-level.

7.2 Generalization to other meta-learning settings

The main focus of this thesis has been on the scenario of model/algorithm selection, in which we assume that a *new task* arrives, and, capitalizing on prior experience of solving similar tasks, an "autonomous agent" finds the best algorithm. This is the most common setting considered in meta-learning.

However, other settings are possible. Considering the meta-dataset matrix S, with datasets (or tasks) in lines and models (or algorithms) in columns, one may be interested in the *dual problem* of selecting *tasks* that an algorithm is most capable of solving (the equivalent of transposing matrix S). This problem would be of interest to an algorithm developer for instance. More generally, one might be interested in navigating in the S matrix in two dimensions: given a matrix partially filled with values, fill it rapidly (starting with the most promising scores).

Thus, even though we only addressed one particular setting, the algorithms we developed lend themselves to solving a variety of problems, some of which are listed below:

1. **Model (or algorithm) selection:** This is the setting that we focused on. This is the problem of a machine learning consultant who gets periodically new tasks (datasets) and wants sot solve them as quickly as possible to make good money, *i.e.* learn from solving past tasks to quickly solve new ones. It is the problem we have called "1D meta-learning" setting: the "game board" has no particular structure and we represent it as a vector (though the position of algorithms in the vector does not inform on algorithm resemblance).

2. **Dataset (or task) selection:** The role of the datasets and algorithms are swapped. This is still a "1D meta-learning" setting, but this is the *dual problem* of the first one. It is the problem faced by startups developing good ML toolkits. Every so often, new algorithms appear in the literature, which may they want to add to their toolkit. To that end they have a set of benchmark datasets on which they evaluate algorithms systematically. They want to know on which problems a new algorithm performs well are good as quickly as possible to evaluate whether it is worth adding to the toolkit. Technically this problem is similar to the first one.

3. **Algorithm selection for multiple datasets (simultaneously):** As in the AutoML challenge (Chapter 2), the *autonomous agent* can be given a set of several datasets, corresponding *e.g.* to related but different tasks, and a fixed time budget. Its final ranking then depends on the average over all datasets of the its chosen algorithm for each dataset. Thus, this is a multi-objective problem. The agent must allocate the right amount of time to each dataset *e.g.* depending on how hard they are. In this setting, the "game board" includes multiple datasets. In order to learn, the agent should experience many game boards of this type extracted from a meta-dataset organized in blocks of tasks sharing some similarities. The meta-datasets we have used in this thesis do not lend themselves particularly

well to this setting, but it should be of practical interest, and thus it would be worthwhile generating a suitable benchmark for this problem.

4. **Simultaneous model and dataset selection:** This setting is more far-fetched. It concerns the search for the best pairs $(task, algorithm)$ by exploring matrix S in two dimensions. As mentioned, if one task/dataset is associated to a data science challenge, this problem would concerns a challenge participant eager to maximize its prize wins. This may have some practical value in the real world too as data science service providers may also be interested in maximizing their profit in a similar way. Constructing a benchmark from a full matrix S is easy as it simply amounts to randomly occluding part of the values in the matrix and try to fill in missing data as fast as possible, prioritizing best scores.

5. **Learning curve climbing:** Here we consider another case of 2D meta-learning, but aiming only as model selection. As an additional dimension, we add the (discretized) progression of algorithm performance as a function of training time steps (epochs). The problem is back to simply finding the best model/algorithm for a new task/dataset. But the "game board" becomes two-dimensional, with algorithm in one dimension and training time in the other. This setting is similar to that of Freeze-Thaw (Chapter 4.2), in which we were interested in finding the optimal time scheduler. The autonomous agent is expected to learn to mitigate exploration and exploitation of learning curves. Freeze Thaw does this through Gaussian processes that samples candidate algorithms and estimate the future learning curves for the sampled candidates and Bayesian optimization that decides in which candidate to invest time. The policy is hard-coded in Freeze-Thaw. Can we could use RL to learn the policy. It would be worth generating a benchmark dataset including learning curves for each algorithm instead of just final performances.

6. **Continuous state and/or action spaces:** Another extension would be to consider continuous *state spaces*, e.g. training the agent to navigate in a continuous HP space. Additionally, the *action space* can be made continuous too, including for example the possibility for the agent to decide how much to "pay" for partially revealing an element. Price to *reveal* may be one way to implement continuous learning curves and/or taking into account "real" cost of evaluating an algorithm or a given dataset in terms of computational and human resources. The implications on RL algorithms of such changes are likely to be more important that the previous ones we considered because of the traditional separation between discrete optimization and continuous optimization techniques.

7.3 Relating MDP, POMDP, REVEAL, and bandits

In this thesis,the AutoML problem has been viewed as a sequential decision making process and formulated it as a MDP, with the introduction of REVEAL games. The definition of REVEAL has been tailored to be as simple as possible and amenable to off-the-shelf RL algorithms for first explorations. However, as outlined in the definition of a REVEAL game, the actions of the agent affect only the amount of information available to the agent NOT the data generating process. This hints at the fact that a REVEAL game might be better described by a POMDP than a regular MDP.

In a Partially Observable MDP (POMDP), the state s_t is hidden from the agent; what is available is the observation o_t which is function of s_t (Figure 7.1). A POMDP is defined using a 7-tuple (state s_t, action a_{t+1}, state transition probability $p(s_{t+1}|s_t, a_{t+1})$, reward r_{t+1}), observable information o_t, conditional observation probability $p(o_t|s_t, a_t)$, and conditional reward probability $p(r_{t+1}|s_t, a_t, o_t, s_{t+1}, a_{t+1}, o_{t+1})$. Importantly, the reward is “second order Markov”[1].

A POMDP is convenient represented as a Directed Acyclic Graph (DAG), see Figure 7.1. The graph is a faithful representation of Markovian properties as long as each node represents a memory-less state variable or vector. This holds for all nodes in the graph except A: A represents an “ autonomous agent” performing actions, which may have an internal memory, even a long-term one like a recurrent neural network.

We can map the AutoML problem to a POMDP in one of several ways. We describe here what is closest to our MDP “board game” setting for the 1-D meta-learning “game”:

- **State:** In a given episode, the state is constant and equal to the ground truth of (pre-computed) scores of algorithms on datasets, *i.e.* one row i in matrix S. For all t, $s_t = S(i,:)$.
- **Action:** Same as in MDP setting, an index j of the next algorithm whose performance on the dataset j under consideration must be revealed.
- **State transition probability:** Deterministic. $p(s_{t+1}|s_t, a_t) = 1$
with $s_{t+1} = s_t = S(i,:)$.
- **Reward:** Same as in MDP:
$r_t = |S(i, j_t)_{perf} - \max_{k=0}^{t-1}(S(i, j_k)_{perf})| - \beta S(d, j_t)_{time}$
Reward calculation deterministic.
- **Observable information:** $o_t = S(i, j)$.

[1] https://artint.info/html/ArtInt_230.html

- **Conditional observation probability:** Deterministic. $p(o_t|s_t,a_t) = 1$.
- **Conditional rewards probability:** Deterministic. $p(r_{t+1}|s_t,a_t,o_t,s_{t+1},a_{t+1},o_{t+1}) = 1$

This setting results in a REVEAL game (Figure 7.2). Unlike a standard POMDP in which the actions DIRECTLY influence the next state, in a REVEAL-POMDP, the action only affects the amount of information revealed. Note that, in this variant, the agent must remember which index values j it already visited if it does not want to get multiple times the same information. Other ways of coding state, action, etc. are possible.

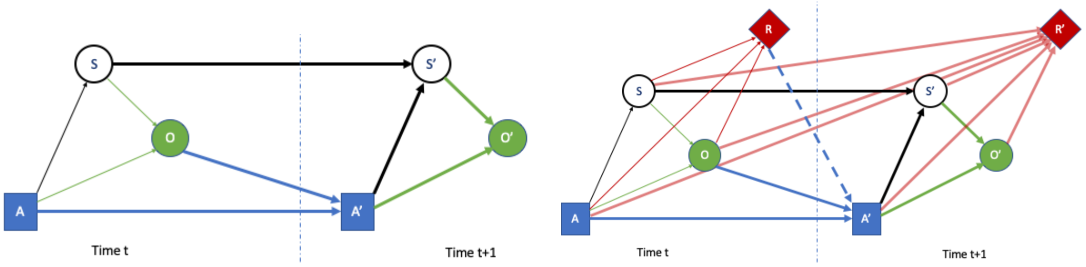
This formulation makes is easy to compare REVEAL games with multi-armed bandits, which have been used to formulate the AutoML problem [70]. The authors proposed that finding the best algorithm is analogous to finding the best arm of the bandit, given particular given dataset. The only factor that influences the agent's action is the reward it receives from trying different arms, the state information is not used.

Contextual bandits can be seen as an extension of multi-armed bandit by adding the information of the state (called 'context'). Formulating meta-learning as contextual bandit is possible because the agent can learn to act based on different contexts (datasets).

In Figure 7.3 we show the DAGs for both bandits with drift and contextual bandits. It is clear that they are NOT identical to REVEAL games. Hence treating the AutoML meta-learning problem as multi-armed bandit is a simplification, which may be harmful.

7.4 Conclusion

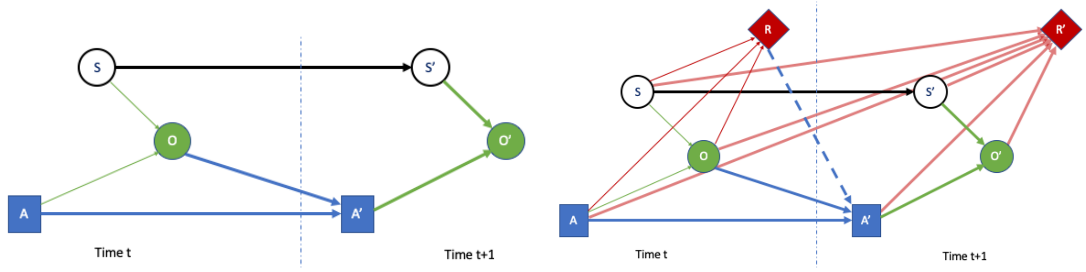
This work has been an exploration of putting the meta-learning problem in the framework of Markov Decision processes. Although such a framework may appear to be reductionist, as it makes the simplifying assumption of a search space that is discrete and finite, it is a practical setting often faced in applications since many algorithms have default hyper-parameter values that work well in practice, or have an internal optimizer of hyper-parameters. Another way of bringing back the problem to a discrete search space is obviously to discretize hyper-parameters. Hence remains the discrete choice of the best algorithm for a given task. This can be cast as a recommendation problem. Since finding the best algorithm is usually an iterative trial and error problem, a natural extension is to turn to active learning, hence our effort to develop an active meta-learning algorithm ACTIVMETAL, based on the collaborative filtering algorithm CofiRank as a sub-routine. This method has turned out to be very powerful and delivered results beating our baselines. To go beyond this approach and avoid the computational burden of inverting a matrix at every recommendation step, we developed an approach based on Deep Reinforcement Learning, which learns a policy. Although RL does not beat ACTIVMETAL at this stage, we view it as our most interesting and promising avenue of research for several reasons: learned policies are very computationally efficient, adaptive, and lend themselves to strategic planning. On the flip side, learning policies with little data is difficult and meta-learning datasets are usually small in size. This is a challenge to advance RL algorithms and make them tackle better “small data”. In parallel, RL is being applied in other ways to some aspects of AutoML than meta-learning, including neural architecture [124] search and data augmentation [25]. This prefigures possibilities of a unified RL approach to various AutoML problems and multi-level meta-learning strategies. We also discussed extensions of our formulation to 2D search (finding simultaneously the best algorithm and the best dataset) and to POMDP (Partially Observable Markov Decision Processes). The latter connection outlines that REVEAL games are distinct but related to contextual bandits and form a particular family of MDP problems. This could build bridges to other fields and open the door to solving a broader class of problems using a similar setting. On another note, we borrowed data from a causal challenge without touching upon causal modeling. However, extending our approaches to selecting causal models would be a natural and legitimate extension. Finally, this thesis work started being motivated by participating in a challenge and analyzing its results. One possible next step would be to organize a meta-learning challenge. We hope to collaborate further on a project of Automated Deep Learning (AutoDL) to which we have contributed and help put together a life-long-learning challenge with the 100+ datasets of the AutoDL challenge series. The availability of the Jean Zay super-computer (1000 GPUs) makes it possible to build an unprecedented meta-learning dataset.



(a) without reward

(b) with reward

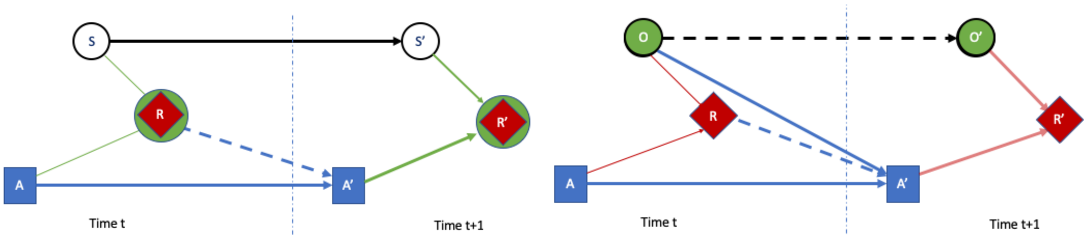
Fig. 7.1 **POMDP:** A Partially Observable Markov Decision Process (POMDP) differs from a regular MDP in that state (S) in visible to agent (A) only through (partial) observation (O): $A' \perp S \mid (O,A)$ Thus if $S = O$, POMDP=MDP . The reward may depend from current and previous variable values (here we consider second order dependence). The agent may not be Markovian *i.e.* , as opposed to the other nodes, it is not necessarily memory-less.



(a) without reward

(b) with reward

Fig. 7.2 **REVEAL:** The difference with a regular POMDP is the missing link between action (A) and state (S), hence in e REVEAL game, the agent cannot influence the data generating process S, **its action only affect the information it receives**. REVEAL games resemble "bandits with drift" and "contextual bandits", but critically differ in the second order Markov dependence of the reward: in a REVEAL game, the reward depends on the **information gain**.



(a) Bandit with drift

(b) Contextual bandit

Fig. 7.3 **Bandits**: The difference between *bandits with drift* and *contextual bandits* is the information available to the agent (performing actions). In the *bandit with drift*, the hidden state (S), responsible for the reward distribution drift, is invisible to the agent (the reward is the observable state $R = O$). The key Markov property is $R' \perp R \mid (A', S')$. In the *contextual bandit*, the reward distribution change is influenced by an observable state ($S = O$) called "context". The key Markov property is $R' \perp R \mid (A', O')$. A REVEAL game does NOT have such Markov properties. In the case where $R = S = O$ the model is a simple bandit and simply $R' \perp R \mid A'$ (not shown).

References

[Doc] Docker. https://www.docker.com/.

[Kag] Kaggle. https://www.kaggle.com/. Accessed: 2018-06-07.

[3] Andrieu, C., Freitas, N. D., and Doucet, A. (1999). Sequential MCMC for Bayesian model selection. In *IEEE Signal Processing Workshop on Higher-Order Statistics*, pages 130–134.

[4] Assunção, F., Lourenço, N., Machado, P., and Ribeiro, B. (2018). Denser: Deep evolutionary network structured representation. *arXiv preprint arXiv:1801.01563*.

[5] Baker, B., Gupta, O., Naik, N., and Raskar, R. (2016). Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*.

[6] Bardenet, R., Brendel, M., Kégl, B., and Sebag, M. (2013a). Collaborative hyperparameter tuning. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*, volume 28, pages 199–207.

[7] Bardenet, R., Brendel, M., Kégl, B., and Sebag, M. (2013b). Collaborative hyperparameter tuning. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*, pages 199–207.

[8] Bardenet, R., Brendel, M., Kégl, B., and Sebag, M. (2013c). Collaborative hyperparameter tuning. In *30th International Conference on Machine Learning*, volume 28, pages 199–207. JMLR Workshop and Conference Proceedings.

[9] Bengio, Y., Courville, A., and Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828.

[10] Bennett, K. P., Kunapuli, G., and Jing Hu, J.-S. P. (2008). Bilevel optimization and machine learning. In *Computational Intelligence: Research Frontiers*, volume 5050 of *Lecture Notes in Computer Science*, pages 25–47. Springer.

[11] Bergstra, J. and Bengio, Y. (2012a). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305.

[12] Bergstra, J. and Bengio, Y. (2012b). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305.

[13] Bergstra, J., Yamins, D., and Cox, D. D. (2013). Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *30th International Conference on Machine Learning*, volume 28, pages 115–123.

[14] Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. (2011a). Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, pages 2546–2554.

[15] Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. (2011b). Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554.

[16] Blum, A. L. and Langley, P. (1997). Selection of relevant features and examples in machine learning. *Artificial Intelligence*, 97(1-2):273–324.

[17] Boullé, M. (2007). Compression-based averaging of selective naive bayes classifiers. *Journal of Machine Learning Research*, 8:1659–1685.

[18] Boullé, M. (2009). A parameter-free classification method for large scale learning. *Journal of Machine Learning Research*, 10:1367–1385.

[19] Brazdil, P., Carrier, C. G., Soares, C., and Vilalta, R. (2008). *Metalearning: applications to data mining*. Springer.

[20] Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

[21] Brochu, E., Cora, V. M., and De Freitas, N. (2010). A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*.

[22] Caruana, R., Niculescu-Mizil, A., Crew, G., and Ksikes, A. (2004). Ensemble selection from libraries of models. In *21st International Conference on Machine Learning*, pages 18–. ACM.

[23] Cawley, G. C. and Talbot, N. L. C. (2007). Preventing over-fitting during model selection via Bayesian regularisation of the hyper-parameters. *Journal of Machine Learning Research*, 8:841–861.

[24] Colson, B., Marcotte, P., and Savard, G. (2007). An overview of bilevel programming. *Annals of Operations Research*, 153:235–256.

[25] Cubuk, E. D., Zoph, B., Mane, D., Vasudevan, V., and Le, Q. V. (2018). Autoaugment: Learning augmentation policies from data. *arXiv preprint arXiv:1805.09501*.

[26] Dempe, S. (2002). *Foundations of bilevel programming*. Kluwer Academic Publishers.

[27] Dietterich, T. G. (1998). Approximate statistical test for comparing supervised classification learning algorithms. *Neural Computation*, 10(7):1895–1923.

[28] Duda, R. O., Hart, P. E., and Stork, D. G. (2001). *Pattern Classification*. Wiley, 2nd edition.

[29] Efron, B. (1983). Estimating the error rate of a prediction rule: Improvement on cross-validation. *Journal of the American Statistical Association*, 78(382):316–331.

[30] Eggensperger, K., Feurer, M., Hutter, F., Bergstra, J., Snoek, J., Hoos, H., and Leyton-Brown, K. (2013). Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*.

[31] Escalante, H. J., Montes, M., and Sucar, L. E. (2009). Particle swarm model selection. *Journal of Machine Learning Research*, 10:405–440.

[32] Falkner, S., Klein, A., and Hutter, F. (2017). Combining hyperband and bayesian optimization. In *BayesOpt 2017 NIPS Workshop on Bayesian Optimization*.

[33] Falkner, S., Klein, A., and Hutter, F. (2018a). Bohb: Robust and efficient hyperparameter optimization at scale. *arXiv preprint arXiv:1807.01774*.

[34] Falkner, S., Klein, A., and Hutter, F. (2018b). Practical hyperparameter optimization. In *International Conference on Learning Representations 2018 Workshop track*.

[35] Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., and Hutter, F. (2015a). Efficient and robust automated machine learning. In *Proceedings of the Neural Information Processing Systems*, pages 2962–2970.

[36] Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., and Hutter, F. (2015b). Efficient and Robust Automated Machine Learning. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, pages 2962–2970. Curran Associates, Inc.

[37] Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., and Hutter, F. (2015c). Methods for improving bayesian optimization for automl. In *Proceedings of the International Conference on Machine Learning 2015, Workshop on Automatic Machine Learning*.

[38] Feurer, M., Springenberg, J., and Hutter, F. (2015d). Initializing bayesian hyperparameter optimization via meta-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1128–1135.

[39] Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232.

[40] Fusi, N., Sheth, R., and Elibol, M. (2018). Probabilistic matrix factorization for automated machine learning. In *Advances in Neural Information Processing Systems*, pages 3352–3361.

[41] Geman, S., Bienenstock, E., and Doursat, R. (1992). Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58.

[42] Golub, G. H. and Reinsch, C. (1971). Singular value decomposition and least squares solutions. In *Linear Algebra*, pages 134–151. Springer.

[43] Guyon, I., Bennett, K., Cawley, G., Escalante, H. J., Escalera, S., Ho, T. K., Macia, N., Ray, B., Saeed, M., Statnikov, A., et al. (2015a). Design of the 2015 chalearn automl challenge. In *Neural Networks (IJCNN), 2015 International Joint Conference on*, pages 1–8. IEEE.

[44] Guyon, I., Bennett, K., Cawley, G., Escalante, H. J., Escalera, S., Ho, T. K., Macià, N., Ray, B., Saeed, M., Statnikov, A., and Viegas, E. (2015b). AutoML challenge 2015: Design and first results. In *Proc. of AutoML 2015@ICML.*

[45] Guyon, I., Gunn, S., Nikravesh, M., and Zadeh, L., editors (2006). *Feature extraction, foundations and applications*. Studies in Fuzziness and Soft Computing. Physica-Verlag, Springer.

[46] Guyon et al., I. (2017, to appear). Analysis of the AutoML challenge series 2015-2018. In Hutter et al., F., editor, *Automatic Machine Learning*. Springer series in Challanges in Machine Learning.

[47] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The Weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18.

[48] Hastie, T., Rosset, S., Tibshirani, R., and Zhu, J. (2004). The entire regularization path for the support vector machine. *Journal of Machine Learning Research*, 5:1391–1415.

[49] Hastie, T., Tibshirani, R., and Friedman, J. (2001). *The elements of statistical learning: Data mining, inference, and prediction*. Springer, 2nd edition.

[50] Hennig, P. and Schuler, C. J. (2012). Entropy search for information-efficient global optimization. *Journal of Machine Learning Research*, 13(Jun):1809–1837.

[Hutter et al.] Hutter, F., Hoos, H., Murphy, K., and Ramage, S. Sequential Model-based Algorithm Configuration (SMAC). http://www.cs.ubc.ca/labs/beta/Projects/SMAC/.

[52] Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011a). Sequential model-based optimization for general algorithm configuration. In *Proc. of LION-5*, page 507–523.

[53] Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011b). Sequential model-based optimization for general algorithm configuration. In *Proceedings of the conference on Learning and Intelligent OptimizatioN (LION 5).*

[54] Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011c). Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer.

[55] Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011d). Sequential model-based optimization for general algorithm configuration. In Coello, C. A. C., editor, *the 5th Learning and Intelligent Optimization Conference (LION)*, volume 6683 of *LNCS*, pages 507–523. Springer.

[56] Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011e). Sequential Model-based Optimization for General Algorithm Configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, pages 507–523, Berlin, Heidelberg. Springer-Verlag. event-place: Rome, Italy.

[57] Hutter, F., Kotthoff, L., and Vanschoren, J. (2019). Automatic machine learning: methods, systems, challenges. *Challenges in Machine Learning*.

[58] Ioannidis, J. P. A. (2005). Why most published research findings are false. *PLoS Medicine*, 2(8):e124.

[59] Jordan, M. I. (2013). On statistics, computation and scalability. *Bernoulli*, 19(4):1378–1390.

[60] Kakade, S. M. (2002). A natural policy gradient. In *Advances in neural information processing systems*, pages 1531–1538.

[61] Keerthi, S. S., Sindhwani, V., and Chapelle, O. (2007). An efficient method for gradient-based adaptation of hyperparameters in SVM models. In *Advances in Neural Information Processing Systems*.

[62] King, R. D., Feng, C., and Sutherland, A. (1995). Statlog: comparison of classification algorithms on large real-world problems. *Applied Artificial Intelligence an International Journal*, 9(3):289–333.

[63] Klein, A., Falkner, S., Bartels, S., Hennig, P., and Hutter, F. (2017). Fast bayesian hyperparameter optimization on large datasets. In *Electronic Journal of Statistics*, volume 11.

[64] Kohavi, R. and John, G. H. (1997). Wrappers for feature selection. *Artificial Intelligence*, 97(1-2):273–324.

[65] Komer, B., Bergstra, J., and Eliasmith, C. (2014). Hyperopt-sklearn: automatic hyperparameter configuration for scikit-learn. In *ICML workshop on AutoML*, pages 2825–2830. Citeseer.

[66] Konda, V. R. and Tsitsiklis, J. N. (2000). Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014.

[67] Langford, J. (2005). Clever methods of overfitting. Blog post at http://hunch.net/?p=22.

[68] LeCun, Y. and Cortes, C. (2010). MNIST handwritten digit database.

[69] Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. (2016a). Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*.

[70] Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. (2016b). Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *arXiv:1603.06560 [cs, stat]*. arXiv: 1603.06560.

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

[72] Liu, Z., Guyon, I., Junior, J. J., Madadi, M., Escalera, S., Pavao, A., Escalante, H. J., Tu, W.-W., Xu, Z., and Treguer, S. (2019). AutoCV Challenge Design and Baseline Results. In *CAp 2019 - Conférence sur l'Apprentissage Automatique*, Toulouse, France.

[73] Lloyd, J. (2016). Freeze Thaw Ensemble Construction. https://github.com/jamesrobertlloyd/automl-phase-2.

[74] Lutz, R. W. (2006). Logitboost with trees applied to the WCCI 2006 performance prediction challenge datasets. In *Proc. IJCNN06*, pages 2966–2969, Vancouver, Canada. INNS/IEEE.

[75] Mısır, M. and Sebag, M. (2017). Alors: An algorithm recommender system. *Artificial Intelligence*, 244:291–314.

[76] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[77] Mockus, J., Tiesis, V., and Zilinskas, A. (1978). The application of bayesian methods for seeking the extremum. *Towards global optimization*, 2(117-129):2.

[78] Momma, M. and Bennett, K. P. (2002). A pattern search method for model selection of support vector regression. In *In Proceedings of the SIAM International Conference on Data Mining*. SIAM.

[79] Moore, G., Bergeron, C., and Bennett, K. P. (2011). Model selection for primal SVM. *Machine Learning*, 85(1-2).

[80] Moore, G. M., Bergeron, C., and Bennett, K. P. (2009). Nonsmooth bilevel programming for hyperparameter selection. In *IEEE International Conference on Data Mining Workshops*, pages 374–381.

[81] Muja, M. and Lowe, D. (2013). Fast library for approximate nearest neighbors (flann). *git://github. com/mariusmuja/flann. git. url: http://www. cs. ubc. ca/research/flann*.

[82] Muñoz, M. A., Villanova, L., Baatar, D., and Smith-Miles, K. (2018). Instance spaces for machine learning classification. *Machine Learning*, 107(1):109–147.

[83] Opper, M. and Winther, O. (2000). *Gaussian processes and SVM: Mean field results and leave-one-out*, pages 43–65. MIT. Massachusetts Institute of Technology Press (MIT Press) Available on Google Books.

[84] Pan, S. J. and Yang, Q. (2010). A survey on transfer learning. *IEEE Transactions on Knoweledge and Data Engineering*, 22(10):1345–1359.

[85] Park, M. Y. and Hastie, T. (2007). L1-regularization path algorithm for generalized linear models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 69(4):659–677.

[86] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Matthieu, B., Perrot, M., and Duchesnay, E. (2011a). Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830.

[87] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011b). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[88] Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. (2018a). Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*.

[89] Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. (2018b). Efficient Neural Architecture Search via Parameter Sharing. *arXiv:1802.03268 [cs, stat]*. arXiv: 1802.03268.

[90] Rasmussen, C. E. (2004). Gaussian processes in machine learning. In *Advanced lectures on machine learning*, pages 63–71. Springer.

[91] Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q. V., and Kurakin, A. (2017). Large-scale evolution of image classifiers. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2902–2911. JMLR. org.

[92] Rennie, J. and Srebro, N. (2005). Fast maximum margin matrix factorization for collaborative prediction. In *Proceedings of the 22nd international conference on Machine learning*, pages 713–719. ACM.

[93] Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, 14(5):465–471.

[94] Schölkopf, B. and Smola, A. J. (2001). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond.* MIT Press.

[95] Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015). Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897.

[96] Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems 25*, pages 2951–2959.

[97] Srebro, N., Rennie, J., and Jaakkola, T. (2005). Maximum-margin matrix factorization. *Advances in neural information processing systems*, 17(5):1329–1336.

[98] Statnikov, A., Wang, L., and Aliferis, C. F. (2008). A comprehensive comparison of random forests and support vector machines for microarray-based cancer classification. *BMC Bioinformatics*, 9(1).

[99] Stern, D., Herbrich, R., Graepel, T., Samulowitz, H., Pulina, L., and Tacchella, A. (2010). Collaborative expert portfolio management. In *AAAI*, pages 179–184.

[100] Sun, Q., Pfahringer, B., and Mayo, M. (2012). Full model selection in the space of data mining operators. In *Genetic and Evolutionary Computation Conference*, pages 1503–1504.

[101] Sun-Hosoya, L., Guyon, I., and Sebag, M. (2018a). Activmetal: Algorithm recommendation with active meta learning. In *IAL 2018 workshop, ECML PKDD*.

[102] Sun-Hosoya, L., Guyon, I., and Sebag, M. (2018b). Lessons learned from the automl challenge. In *Conférence sur l'Apprentissage Automatique 2018*.

[103] Sutton, R. S. and Barto, A. G. (2018a). *Reinforcement learning: an introduction*. Adaptive computation and machine learning series. The MIT Press, Cambridge, Massachusetts, second edition edition.

[104] Sutton, R. S. and Barto, A. G. (2018b). *Reinforcement learning: An introduction*. MIT press.

[105] Swersky, K., Snoek, J., and Adams, R. P. (2013). Multi-task Bayesian optimization. In *Advances in Neural Information Processing Systems 26*, pages 2004–2012.

[106] Swersky, K., Snoek, J., and Adams, R. P. (2014). Freeze-thaw bayesian optimization. *arXiv preprint arXiv:1406.3896*.

[107] Thakur, A. and Krohn-Grimberghe, A. (2015). Autocompete: A framework for machine learning competitions. In *AutoML Workshop, International Conference on Machine Learning 2015*.

[108] Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2012). Auto-weka: Automated selection and hyper-parameter optimization of classification algorithms. *CoRR*, abs/1208.3719.

[109] Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2013a). Auto-weka: combined selection and hyperparameter optimization of classification algorithms. In Dhillon, I. S., Koren, Y., Ghani, R., Senator, T. E., Bradley, P., Parekh, R., He, J., Grossman, R. L., and Uthurusamy, R., editors, *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, (KDD)*, pages 847–855.

[110] Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2013b). Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 847–855. ACM.

[111] Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2013c). Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 847–855. ACM.

[112] Tuv, E., Borisov, A., Runger, G., and Torkkola, K. (2009). Feature selection with ensembles, artificial variables, and redundancy elimination. *Journal of Machine Learning Research*, 10:1341–1366.

[113] Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*.

[114] Vanschoren, J., van Rijn, J. N., Bischl, B., and Torgo, L. (2013). Openml: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60.

[115] Vanschoren, J., van Rijn, J. N., Bischl, B., and Torgo, L. (2014). OpenML: networked science in machine learning. *ACM SIGKDD Explorations Newsletter*, 15(2):49–60. arXiv: 1407.7722.

[116] Vapnik, V. and Chapelle, O. (2000). Bounds on error expectation for support vector machines. *Neural computation*, 12(9):2013–2036.

[117] Vapnik, V. N. (1998). *Statistical learning theory*. Wiley.

[118] Voorhees, E. M. (2001). Overview of the trec 2001 question answering track. In *TREC*, pages 42–51.

[119] Watkins, C. J. C. H. (1989). Learning from delayed rewards.

[120] Weimer, M., Karatzoglou, A., Le, Q., and Smola, A. (2007). CofiRank-maximum margin matrix factorization for collaborative ranking. In *Proceedings of the 21st Annual Conference on Neural Information Processing Systems (NIPS)*, pages 222–230.

[121] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256.

[122] Zhang, A., Ballas, N., and Pineau, J. (2018). A dissection of overfitting and generalization in continuous reinforcement learning. *arXiv preprint arXiv:1806.07937*.

[123] Zoph, B. and Le, Q. V. (2016a). Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.

[124] Zoph, B. and Le, Q. V. (2016b). Neural Architecture Search with Reinforcement Learning. *arXiv:1611.01578 [cs]*. arXiv: 1611.01578.

ÉCOLE DOCTORALE
Sciences et technologies
de l'information
et de la communication (STIC)

Titre: Meta-Learning en tant que Processus de Décision Markovien

Mots clés: apprentissage automatisé, meta-learing, processus de décision markovien

Résumé: L'apprentissage automatique (ML) a connu d'énormes succès ces dernières années et repose sur un nombre toujours croissant d'applications réelles. Cependant, la conception d'algorithmes prometteurs pour un problème spécifique nécessite toujours un effort humain considérable. L'apprentissage automatisé (AutoML) a pour objectif de sortir l'homme de la boucle. AutoML est généralement traité comme un problème de sélection d'algorithme / hyper-paramètre. Les approches existantes incluent l'optimisation Bayésienne, les algorithmes évolutionnistes et l'apprentissage par renforcement. Parmi eux, auto-sklearn, qui intègre des techniques de meta-learning à l'initialisation de la recherche, occupe toujours une place de choix dans les challenges AutoML. Cette observation a orienté mes recherches vers le domaine du meta-learning. Cette orientation m'a amené à développer un nouveau cadre basé sur les processus de décision Markovien (MDP) et l'apprentissage par renforcement (RL).

Les principaux résultats de cette thèse sont: (1) Sélection HP / modèle: La méthode Freeze-Thaw a été exploré pour entrer dans le premier challenge AutoML, obtenant la 3ème place du tour final. (2) ACTIVMETAL: un nouvel algorithme pour le meta-learning actif (ACTIVMETAL) a été conçu, proposant une solution glouton au problème meta-learning. (3) REVEAL: Une nouvelle conceptualisation du meta-learning en tant que processus de décision Markovien a été dévelopée et intégrée dans le cadre plus général des jeux REVEAL. Les solutions basée sur l'apprentissage par renforcement ont été proposées. Le travail présenté dans cette thèse est de nature empirique. Plusieurs méta-données du monde réel ont été utilisées dans cette recherche. Des méta-données artificielles et semi-artificielles sont également utilisées dans mon travail. Les résultats indiquent que RL est une approche viable du problème de meta-learning, bien qu'il reste encore beaucoup à faire pour optimiser les algorithmes et les faire passer à l'échelle aux problèmes de meta-learning plus vastes.

Title: Meta-Learning as a Markov Decision Process

Keywords: AutoML, meta-learning, Markov Decision Process

Abstract: Machine Learning (ML) has enjoyed huge successes in recent years and an ever-growing number of real-world applications rely on it. However, designing promising algorithms for a specific problem still requires huge human effort. Automated Machine Learning (AutoML) aims at taking the human out of the loop and develop machines that generate / recommend good algorithms for a given ML tasks. AutoML is usually treated as a algorithm / hyper-parameter selection problems, existing approaches include Bayesian optimization, evolutionary algorithms as well as reinforcement learning. Among them, auto-sklearn which incorporates meta-learning techniques in their search initialization, ranks consistently well in AutoML challenges. This observation oriented my research to the Meta-Learning domain, leading to my recent paper where active learning and collaborative filtering are used to assign as quickly as possible a good algorithm to a new dataset, based on a meta-learning performance matrix S, i.e. a matrix of scores of algorithms on given datasets or tasks. This direction led me to develop a novel framework based on Markov Decision Processes (MDP) and reinforcement learning (RL).

The main results of this thesis are: (1) HP / model selection: I have explored the Freeze-Thaw method to enter the first AutoML challenge, achieving 3rd place in the final round ; (2) ACTIVMETAL: I have designed a new algorithm for active meta-learning (ACTIVMETAL) which offers a greedy solution to the meta-learning problem; (3) REVEAL: I developed a new conceptualization of meta-learning as a Markov Decision Process and put it into the more general framework of REVEAL games. Agents are developped to learn (with reinforcement learning) to predict the next best algorithm to be tried.

The work presented in my thesis is empirical in nature. Several real world meta-datasets were used in this research. Artificial and semi-artificial meta-datasets are also generated and used in this work. The results indicate that RL is a viable approach to the meta-learning problem, although much work remains to be done to optimize algorithms to make them scale to larger meta-learning problems.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France