Ludwig-Maximilians-Universität München
Institut für Statistik

Master thesis

ML-ReinBo: Machine learning pipeline search with reinforcement learning and Bayesian optimization

Author Jiali Lin

Supervisor Prof. Dr. Bernd Bischl, Xudong Sun

Date March 17, 2019

Abstract

Machine learning pipeline typically consists of several sequential stages such as data pre-processing, feature engineering and core model selection with a large number of alternative operations in each stage. To design a machine learning pipeline for a given data set, not only the operations at each stage have to be carefully selected with expertise, but also the hyper-parameters for some operations (especially for core models) need to be tuned because of their large impact on predictive performance. Automated machine learning (AutoML) aims to automatically design machine learning pipelines without human intervention, which especially benefits people with limited experience in machine learning.

However, machine learning pipeline design is characterized by conditional dependency, which means that the hyper-parameters are conditional on operations at each stage and become relevant only when the corresponding operations are selected. This is also called Combined Algorithm Selection and Hyper-parameter optimization (CASH) problem in Auto-WEKA's AutoML approach.

This thesis is proposed to solve this hierarchical hyper-parameter space problem by combining reinforcement learning with Bayesian optimization, where the stages in machine learning pipeline are modeled as a sequential decision process to be learned by reinforcement learning agent and the conditional (discrete or continuous) hyper-parameters are tuned by Bayesian optimization, which is dubbed as **ML-ReinBo**. Furthermore, some empirical experiments are also performed to compare ML-ReinBo with the state of art AutoML systems like Auto-sklearn and TPOT, as well as popular optimization methods such as Random Search, Irace and TPE. The results show that ML-ReinBo performs favorably in most cases.

Contents

1 Introduction

As subset of artificial intelligence, machine learning has achieved great success in a wide range of application areas and has changed our life incredibly, which has largely increased the demand for machine learning systems in recent years.

However, for a given data set, to get better predictive performance, we often have to carefully design the machine learning pipeline (Figure 1) by selecting appropriate pre-processing method, engineering features and choosing classification/regression model, as well as configuring a good set of conditional hyper-parameters, which requires a lot of expert knowledge and is especially unfriendly to non-expert users. Manually exploring the pipelines costs a lot of work and generally does not lead to models with satisfactory accuracy.

In stead of making decisions among such a large amount of potential operations and hyper-parameters by ourselves, automated machine learning (AutoML) helps to automatically generate a data analysis pipeline with optimized hyper-parameters, which provides opportunity to users with limited machine learning expertise to train high-quality models and also makes machine learning more applicable in a variety of real-world problems.

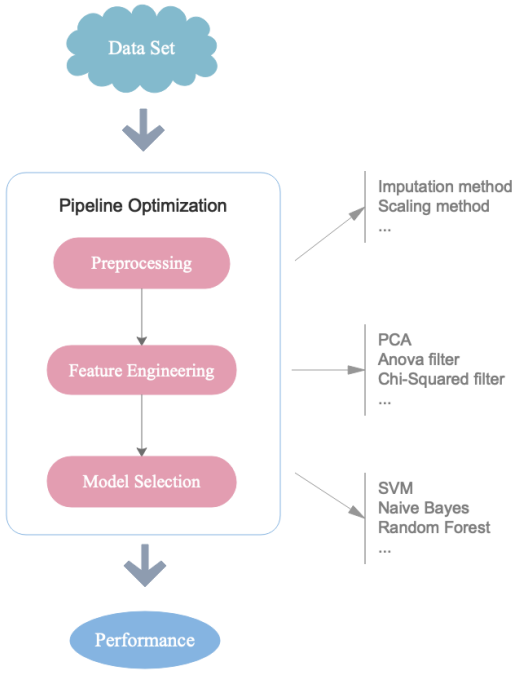

Figure 1: Machine learning pipeline optimization

The growing number of machine learning models and hyper-parameters in recent years makes it totally impossible to evaluate all potential pipelines. An AutoML system is proposed to search in the machine learning pipeline space and find a well-performing architecture for a given data set. To build an efficient AutoML system is particularly difficult for two main reasons:

- Machine learning pipeline evaluation is typically very expensive in terms of computational time. During the search process, an efficient AutoML system need to make a trade-off between exploration (of unseen situations) and exploitation (of current knowledge) and make the best of computational resources to find better pipeline configurations.

- An AutoML system optimizes pipeline stages and the hyper-parameters simultaneously. However, it is remarkable that the hyper-parameters are conditional on the selected operations at each stage. More specifically, only when Support Vector Machine (SVM) is chosen as core model, it makes sense to specify the hyper-parameters C and σ, and once the model *naive Bayes* is selected, C and σ become irrelevant while *laplace* becomes valid now. This hierarchical structure of hyper-parameters makes the optimization of machine learning pipeline much more complicated, which is also called Combined Algorithm Selection and Hyper-parameter optimization (CASH) problem in Auto-WEKA's AutoML approach[17].

Some state of art AutoML systems like Auto-sklearn[13] and Auto-WEKA[17] use sequential model-based algorithm configuration (SMAC)[15] method to solve CASH problem while Tree-based Pipeline Optimization Tool (TPOT)[24] optimizes machine learning pipelines with genetic programming (GP)[2] algorithm. However, this thesis will address this problem by combining reinforcement learning with Bayesian optimization, efficiently to design machine learning pipeline and tune conditional hyper-parameters, which is dubbed as **ML-ReinBo** here.

We describe the main structure of this thesis now. Some previous work related to optimization algorithms and popular AutoML systems would be reviewed in Section 2. Section 3 includes the methodology of reinforcement learning and Bayesian optimization involved in our proposed ML-ReinBo algorithm. The most important part of this thesis is the introduction of ML-ReinBo algorithm and it will be described in detail in Section 4, explaining how the states and actions of reinforcement learning algorithm are defined in our case and in which way Bayesian optimization is integrated in a reinforcement learning process. Section 5 will introduce the baselines we set to be compared with ML-ReinBo and the specific empirical experiment settings and results will be presented in Section 6.

2 Related work

2.1 Optimization algorithms

As mentioned above, machine learning pipeline optimization is restricted to resources because of its time-consuming property. Some optimization algorithms aim to determine the best configuration in a limited amount of time by reasonably allocating resources to candidate configurations, which can be classified as budget-based optimization algorithms and the most representative one would be racing algorithm.

F-Race[6] is a racing algorithm for selecting the best configuration of parameterized algorithms based on statistical approaches. The essential idea is to evaluate a given finite set of candidates iteratively on a stream of instances. After each iteration, some candidate configurations, which have performed significantly worse than others regarding to Friedman test with post-hoc analysis for pairs, will be eliminated and only surviving ones will be evaluated for further iterations. As the evaluation proceeds, this approach focuses more and more on the most promising candidate configurations.

An important part in F-Race is how to define the set of candidate configurations of the first step. One way is iterated F-Race, which constructs a probability model for a candidate solution. At each iteration, a sample of candidates are evaluated to update the probability model to bias the next sampling towards the better candidate solutions until some termination criterion is satisfied. The **Irace** package[19] implements the iterated racing procedure, which is an extension of the Iterated F-race procedure.

Hyperband[18] is a relatively new method for tuning hyper-parameters of algorithms. In this case, hyper-parameter optimization is formulated as a pure-exploration non-stochastic infinite-armed bandit problem. The Successive Halving algorithm uniformly allocates predefined resource like iterations, data samples or features to a set of hyper-parameter configurations, throws out the half configurations with worst performance and repeats until one remains. However, for a fixed budget, it is not clear whether we should consider many configurations with a small average training time or a small number of configurations with a long average training time. Hence, Hyperband performs the Successive Halving algorithm with several possible values of number of configurations at the cost of more work than just running Successive Halving for a single value. By doing so, Hyperband is able to exploit situations in which adaptive allocation works well, while protecting itself in situations where more conservative allocations are required.

Another popular category of optimization algorithms is **Bayesian optimization**, also known as sequential model-based optimization (SMBO) for

general algorithm configuration[15], which is a powerful method for solving such black-box optimization problems. It iterates between fitting surrogate models and using them to make choices about which configurations to investigate.

Tree-structured Parzen Estimator Approach (TPE)[5] is a special instantiation of SMBO, but conventional SMBO could not deal with the conditional hyper-parameters. Instead of using Gaussian Process as a surrogate, TPE uses Parzen Window to construct two non-parametric density estimators on top of a tree like hyper-parameter set. Expected improvement induced from lower and upper quantiles density estimators is used to select new candidate proposal from points generated through ancestral sampling.

BOHB[12] is a combination of Bayesian optimization and Hyperband. It relies on Hyperband to determine how many configurations to evaluate, but it replaces random selection of configurations at the beginning of each Hyperband iteration by a model-based search. Once the desired number of configuration for the iteration is reached, the standard Successive Halving procedure is carried out.

2.2 AutoML systems

Auto-Sklearn[13] is an AutoML system based on the Python machine learning library *scikit-learn*[25], and **Auto-Weka**[17] is an AutoML system based on Java library *weka*[14]. Both of them use **sequential model-based algorithm configuration** (SMAC)[15] to solve CASH problem. SMAC is also a sophisticated instantiation of the general SMBO framework. Auto-Weka and Auto-Sklearn transform the conditional hierarchical hyper-parameter space into a flat structure, where SMAC handles conditional hyper-parameters by instantiating inactive conditional parameters to default values and uses random forest to focus on active hyper-parameters[15]. A potential drawback for this method is that the surrogate model needs to learn upon a high dimensional hyper-parameter space, which accordingly might need a large sample of observations to be sufficiently trained, while in practice, running machine learning algorithm is usually very expensive. Additionally, it is not clear what effects the default values will play since the conventional random forest training algorithm is intact in SMAC.

Another popular AutoML system is **Tree-based Pipeline Optimization Tool** (TPOT)[24], which is also built on top of *scikit-learn*. TPOT automatically designs and optimizes machine learning pipelines with genetic programming (GP)[2] algorithm. The machine learning operators are used as genetic programming primitives, which will be combined by tree-based pipelines and the GP algorithm is used to evolve such tree-based pipelines

until the "best" pipeline is found. Similar methods also include Recipe[10]. However, this family of method does not scale well[23].

A new approach to AutoML system is **ML Plan**[23], which uses a Monte Carlo Tree Search alike algorithm to search for pipelines and also configures the pipeline with hyper-parameters. Task is expanded based on best-first search, where the score is estimated by a randomized depth first search by randomly trying different sub-tree possibilities on a Hierarchical Task Network. To ensure exploration, ML-Plan gives equal possibility to the starting node in a Hierarchical Task Network and then use a best-first strategy for searching at the lower part of the network. Potential drawback for this method is that the hyper-parameter space is discretized, which might essentially loses good candidates in continuous spaces since large continuous hyper-parameter spaces would be essentially hard to discretize.

2.3 Reinforcement learning based optimization

The idea of ML-ReinBo is firstly inspired by **MetaQNN**[1], which uses Q learning to search convolutional neural network architectures.

The learning agent in MetaQNN is trained to sequentially choose CNN layers using Q-learning with an ϵ-greedy exploration strategy and the goal is to maximize the cross-validation accuracy. Explicitly, each state is defined as a tuple of all relevant layer parameters: *layer_type, layer_depth, filter_depth, filter_size, stride, image_size, fc_size, terminate*. The parameter *layer_type* can be selected from convolution (C), pooling (P), fully connencted (FC), global average pooling (GAP) or softmax (SM).

In **Neural Architecture Search** (NAS) framework[32], instead of using Q learning, the authors use Recurrent Neural Network as the reinforcement learning policy approximator to generate variable strings to represent various neural architecture forms. The reward function is designed to be the validation performance of the constructed network. The reinforcement learning policy is trained with gradient descent algorithm, specificially REINFORCE. The architecture elements being searched are very similar to MetaQNN.

Inspired from MetaQNN and NAS, ML-ReinBo assumes the machine learning pipeline to be optimized could be represented by a variable length string, but could use both Deep Q Learning and Tabular Q Learning. More importantly, the elements of the neural architecture are mostly factor variables like *layer_type* or discretized elements like *filter_size*, while in our case, ML-ReinBo deals heavily with continuous hyper-parameters (C and σ for Radial Basis Function kernel SVM). To jointly optimize the discrete pipeline

choice and associated continuous hyper-parameters, we embed Bayesian optimization inside the reinforcement learning agent.

A most recent work [31] also combines pipeline search and hyper-parameter optimization in a reinforcement learning process, however, they use R learning for the reinforcement learning part and most importantly, the hyper-parameters are randomly sampled during the reinforcement learning process. An extra stage is needed to sweep the hyper-parameters using hyper-parameter optimization technique.

3 Methodology

3.1 Reinforcement learning

There are three important subfields of machine learning: supervised learning, unsupervised learning and reinforcement learning. In supervised learning, it requires the algorithm to generalize from labeled training data to unseen situations while an unsupervised learning algorithm is to detect hidden structure of input data without labeled responses such as Clustering. As one area of machine learning, reinforcement learning is different from supervised learning and unsupervised learning. Instead, it helps us to solve sequential decision problems and focuses on maximizing the reward or performance, which involves finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge)[16][27].

Reinforcement learning algorithm is popular in game applications like Atari[20], but it could also be generalized to solve some other tasks. Typically, we set up a reinforcement learning task with an agent and an environment. The agent can interact with the dynamic environment by taking actions (A_i) at different states (S_i) and receiving feedback or rewards (R_i) from the environment.

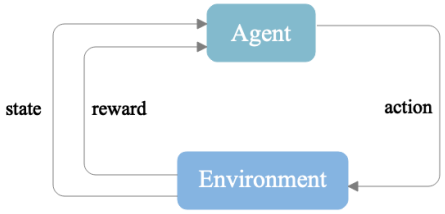

Figure 2: Agent interacts with environment in Reinforcement Learning.

The environment is commonly a Markov Decision Process (MDP)[29], where the future is independent of the past given the present and as a result,

the next state and reward only depend on the current state and action. A rule/policy π is used to guide behaviours of the agent in the environment, which describes the probabilities of taking each action at each state. Since the agent is reward-motivated, it is supposed to continually learn how to select better actions to maximize reward and find the optimal policy.

3.1.1 Tabular Q Learning

A reinforcement learning episode always starts from an initialized state S_0. At each state $S_t \in \mathcal{S}$, the available actions space is denoted as $\mathcal{A}_{S_t}$. Suppose an action A_t of $\mathcal{A}_{S_t}$ is taken at S_t, the state would transition to another state S_{t+1} and the agent receives a reward R_t from the environment. But for a sequential decision problem, our goal is to maximize cumulative future rewards instead of the immediate reward. Hence, following a given policy π, the state value function is defined as

$$V_\pi(s) = \mathbb{E}_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ...|S_t = s] \tag{1}$$

where R_t is the immediate reward and we use $\gamma \in [0, 1]$ as a discount factor for future rewards since the immediate reward is always preferable than delayed reward. Analogously, taking action a at state s and then following the given policy π, the action value function can be defined as

$$Q_\pi(s, a) = \mathbb{E}_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ...|S_t = s, A_t = a] \tag{2}$$

According to the Bellman Equation[11], the value function can also be written as

$$V_\pi(s) = \mathbb{E}_\pi[R_t + \gamma V_\pi(S_{t+1})|S_t = s] \tag{3}$$
$$Q_\pi(s, a) = \mathbb{E}_\pi[R_t + \gamma Q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \tag{4}$$

The optimal value function is the maximum value function over all possible policies:

$$V^*(s) = \max_\pi V_\pi(s) \;\; and \;\; Q^*(s, a) = \max_\pi Q_\pi(s, a), \;\; \forall s \in \mathcal{S} \;\; a \in \mathcal{A} \tag{5}$$

Hence, if we find $V^*(s)$ or $Q^*(s, a)$, the problem is solved since the optimal policy π^* could be easily derived by choosing action with the maximum action value at each state, namely a greedy policy.

Dynamic programming can be applied to find the optimal value function and optimal policy π^* in MDP by iteratively updating value function and policy according to Equation (3), but it requires model of the environment,

which means that the state transitions and rewards given the action are fully known in advance. However, most problems are not in this case. As in our case, the reward is unknown until the machine learning pipeline is completely configured with hyper-parameters and trained to return the cross validation accuracy.

Another solution is Temporal-difference (TD) learning, which instead samples (S_t, A_t, R_t, S_{t+1}) from the environment and performs updates based on the current estimates[28]. In each step, the action value function is updated by

$$Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha\delta_t \tag{6}$$

$$\delta_t = R_t + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \tag{7}$$

where α is the learning rate and δ_t is TD error, which is the difference between the previous estimate of action value $Q(S_t, A_t)$ and the newly computed target value $R_t + \gamma Q(S_{t+1}, A_{t+1})$ based on Equation (4).

Q-Learning[30] is one of the TD algorithms. Tabular Q Learning updates the estimates in a Q-table by taking the maximum value of all available actions at the next step, thus,

$$\delta_t = R_t + \gamma max_{a\in\mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t) \tag{8}$$

according to the Bellman Equation again. Therefore, in Q-Learning we update the action value function in each step by

$$Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha(R_t + \gamma max_{a\in\mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t)), \tag{9}$$

During the sample process, Q-Learning selects the action of highest value at each state with an ϵ-greedy strategy, which makes a trade-off between exploration and exploitation. We do not want to always choose the action with highest value because the agent need to learn more about the environment. Therefore, at each state, the agent picks an uniformly random action with probability ϵ and chooses the best action according to current policy π with probability of $1 - \epsilon$. The ϵ is set relatively high at the beginning of learning process for enough exploration and then reduced to exploit the knowledge and select the best action. And iteratively updating each pair of state and action for enough times will yield correct Q values. The complete Q-Learning algorithm is shown in Algorithm 1[1].

[1]In practice, we can also use experience replay in Q-Learning to circumvent the highly correlated samples.

Algorithm 1 Q-learning

Initialize $Q(s, a)$ arbitrarily
for each episode **do**
  Initialize s
  for each step of episode **do**
    Choose a from $\mathcal{A}_s$ using policy derived from Q-table (with ϵ-greedy)
    Take action a, observe r, s'
    $Q(s, a) \longleftarrow Q(s, a) + \alpha[r + \gamma max_{a'} Q(s', a') - Q(s, a)]$
    $s \longleftarrow s'$
  end for
end for

3.1.2 Deep Q Network (DQN)

The Tabular Q Learning is implemented by creating a Q-table and updating each pair of state and action value estimate as mentioned above. However, this is not scalable, since it would not be efficient at all in case of a huge state space. Deep Q Network[21] creates a neural network to approximate the Q values instead.

Algorithm 2 Deep Q Learning

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode = 1, M **do**
  Initialize s_0
  for t = 1, T **do**
    Select a random action a_t with probability ϵ
    otherwise choose $a_t = argmax_a Q(s_t, a; \theta)$
    Take action a_t, observe r_t and s_{t+1}, store (s_t, a_t, r_t, s_{t+1}) in D
    Sample random mini-batch (s_j, a_j, r_j, s_{j+1}) from D
    Set $y_j = r_j$ if episode terminates at step $j + 1$
    otherwise $y_j = r_j + \gamma max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-)$
    Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ with respect to network parameters θ
    Every C steps reset $\hat{Q} = Q$
  end for
end for

DQN takes the current state S_t as input and output a vector of Q-values for each possible action at the given state through a neural network. It

stores all experience of actions, state transitions and rewards (S_t, A_t, R_t, S_{t+1}) in memory D, but during the learning, it only uses a random sample (or mini-batches) from D to update neural networks, which is called Experience Replay. This increases the learning speed and also reduces correlations in the observation sequence[21]. The complete algorithm of DQN is presented in Algorithm 2.

3.2 Bayesian optimization

Bayesian optimization is a sequential design strategy for global optimization of black-box functions[22], which is a type of Model-based Optimization (MBO). It aims to find the global optimum of an objective function $f(x)$, which doesn't require derivatives and it instead uses a cheap probabilistic surrogate model, a prior distribution, to estimate the objective function and then decides, via a learning criterion (acquisition function), where the next query point should be. Bayesian optimization is specially useful when the objective function has no closed-form expression or the evaluation of $f(x)$ is extremely expensive.

Two major choices that must be made in Bayesian optimization are the surrogate model and acquisition function. The commonly used surrogate model is Gaussian Process (GP)[26], which is defined by the property that any finite set of N points $\{x_n \in \mathcal{X}\}_{n=1}^{N}$ induces a multivariate Gaussian distribution on $\mathbb{R}^N$. A Gaussian Process is fully specified by a mean function $m(\cdot)$ and a covariance function $k(\cdot, \cdot)$. For a finite set of $x_1, \cdots, x_t \in \mathcal{X}$, the corresponding random variables $f(x_1), \cdots, f(x_t)$ have the distribution:

$$\begin{bmatrix} f(x_1) \\ \vdots \\ f(x_t) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(x_1) \\ \vdots \\ m(x_t) \end{bmatrix}, \begin{bmatrix} k(x_1, x_1) & \cdots & k(x_1, x_t) \\ \vdots & \ddots & \vdots \\ k(x_t, x_1) & \cdots & k(x_t, x_t) \end{bmatrix} \right) \tag{10}$$

with

$$\begin{aligned} m(x) &= E[x] \\ k(x, x') &= E[(x - m(x))(x' - m(x'))] \end{aligned}$$

where $x, x' \in \mathcal{X}$.

$k(\cdot, \cdot)$ is also called kernel function and the choice of $k(\cdot, \cdot)$ is crucial for Gaussian Process. A very popular choice is squared exponential kernel function:

$$k_{SE}(x, x') = exp(-\frac{1}{2l^2}\|x - x'\|^2) \tag{11}$$

where $l > 0$ is a length scale parameter.

Based on Gaussian Process we can predict the distribution of the next point x_{t+1}, given the previous observations $(x_1, y_1), ..., (x_t, y_t)$. Suppose $m(x) = \mathbf{0}$ and $\mathbf{y}_{1:t} = [y_1, \cdots, y_t]$, then based on Gaussian process property we have the joint distribution:

$$\begin{bmatrix} \mathbf{y}_{1:t} \\ y_{t+1} \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \mathbf{K} & \mathbf{k} \\ \mathbf{k}^T & k(x_{t+1}, x_{t+1}) \end{bmatrix}\right) \tag{12}$$

where

$$\mathbf{K} = \begin{bmatrix} k(x_1, x_1) & \cdots & k(x_1, x_t) \\ \vdots & \ddots & \vdots \\ k(x_t, x_1) & \cdots & k(x_t, x_t) \end{bmatrix}$$

$$\mathbf{k} = [k(x_{t+1}, x_1) \quad k(x_{t+1}, x_2) \quad \cdots \quad k(x_{t+1}, x_t)]$$

And the predictive distribution can be generated as

$$P(y_{t+1}|(x_1, y_1), \cdots, (x_t, y_t), x_{t+1}) = \mathcal{N}(\mu_t(x_{t+1}), \sigma_t^2(x_{t+1})) \tag{13}$$

where

$$\begin{aligned} \mu_t(x_{t+1}) &= \mathbf{k}^T \mathbf{K}^{-1} \mathbf{y}_{1:t} \\ \sigma_t^2(x_{t+1}) &= k(x_{t+1}, x_{t+1}) - \mathbf{k}^T \mathbf{K}^{-1} \mathbf{k} \end{aligned}$$

To select the next point, on one side, we should exploit the current knowledge and take a point with highest probability to be the new maximum to accelerate convergence. On the other side, we should also explore larger regions, which might be sub-optimal currently but the maximum of $f(x)$ may be located there with high probability[3], in order to avoid falling into a local optimum. Therefore, Bayesian optimization applies an acquisition function, e.g. Expected Improvement, which makes a trade-off between high mean value (exploitation) and high variance (exploration). The Expected Improvement acquisition function is given by:

$$\begin{aligned} \alpha_{EI} &= \mathbb{E}[y_{t+1} - y^*] P(y_{t+1} > y^*) \\ &= \Phi(Z)(\mu_t(x_{t+1}) - y^*) + \phi(Z)\sigma_t(x_{t+1}) \end{aligned}$$

where y^* is the current best value, $Z = \frac{\mu(x_{t+1}) - y^*}{\sigma_t(x_{t+1})}$, Φ and ϕ are the cumulative distribution function and probabilistic density function of the standard normal distribution respectively. And the next point x_{t+1} to be selected is supposed to have the highest expected improvement.

4 ML-ReinBo

A machine learning pipeline potentially consists of several stages and there are various available operations at each stage to be selected. This process could be formulated as a task of searching the pipeline space with a reinforcement learning agent to maximize the final model accuracy.

For example, a supervised classification machine learning task could be solved in a 3-stage process including data pre-processing, feature engineering and classification model selection, where at each stage there are several available operations (for instance, Principal Component Analysis could be an operation at the stage of feature engineering and Support Vector Machine could be a potential choice at stage of classification model selection). A reinforcement learning agent is supposed to learn how to select the best operation at each stage to maximize the accuracy of final model for a given data set.

ML-ReinBo models the pipeline composition process as a reinforcement learning episode, where a particular operation at stage i is treated as an action a_i, taken upon a particular state s_i, which is supposed to include previous information of actions taken from starting state s_0 until the current stage. $\mathcal{A}_{s_i}$ denotes all potential operations/actions at state s_i. Suppose the machine learning pipeline has K stages, then one complete episode consists of a chain of states $s_0, s_1, ..., s_K$ and the pipeline search space could be denoted by $\prod_{i=1}^{K} \mathcal{A}_{s_i}$, where $\prod$ denotes the Cartesian Product. Accordingly, the pipeline search task then becomes to find an optimal policy π to decide which action from $\mathcal{A}_{s_i}$ to select at a particular state s_i, or specifically in Q-learning, to estimate the state-action values in Q-table.

At end of each episode, a machine learning pipeline is generated by the reinforcement learning agent with current policy π and ϵ-greedy strategy, which is, however, an incomplete pipeline. Aside from the pipeline operation itself, a set of conditional hyper-parameters Φ_{a_i} for operation/action a_i at stage i also need to be specified before we can train the model. For instance, Φ_{a_i} could be hyper-parameter *rank* of PCA for keeping the number of top features or hyper-parameters C and σ for classification model SVM. Thus, a complete pipeline search space would be $\prod_{i=1}^{K} \mathcal{A}(S_i; \Phi(A_i))$, where $\Phi(A_i)$ denotes the conditional hyper-parameter space at stage i, which means that the conditional hyper-parameters could be involved in the model training phase only when the relevant operation/action has been selected during the reinforcement learning phase. During the composition process, a current uncomplete pipeline can not be evaluated until the operations (especially the final learner) are configured with hyper-parameters and trained on the data. Therefore, Bayesian optimization is applied here to search for the

hyper-parameters and evaluate configured machine learning pipelines.

Running MBO at end of each episode for large number of iterations could help to find optimal conditional hyper-parameters, which is however quite expensive, since machine learning pipeline evaluation is typically time-consuming as mentioned in Section 1. Instead, ML-ReinBo would only run MBO with a small budget at end of each episode and then store the hyper-parameters and corresponding cross-validation accuracy (or loss values) for the generated pipeline in a MBO surrogate dictionary (memory dictionary). Then if the pipeline is sampled again by reinforcement learning agent, the surrogate model could be retrieved from the dictionary to facilitate further searching using MBO. In that case, ML-ReinBo will allocate more resources to better pipelines as reinforcement learning agent focus more and more on pipelines with higher accuracy, and avoids wasting time on unpromising pipelines. The Figure 3 shows the main process of ML-ReinBo.

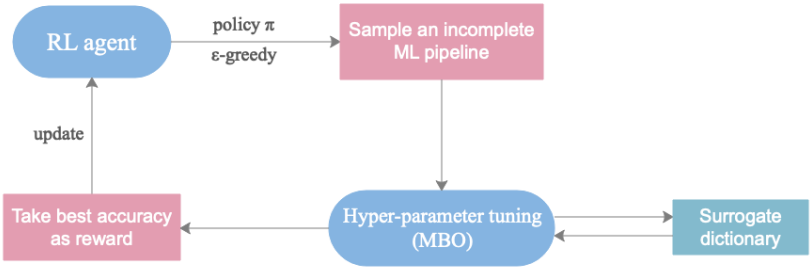

Figure 3: Main process of ML-ReinBo

4.1 Algorithm

The main procedure of ML-ReinBo algorithm is described by Algorithm 3. At the starting point, the policy π is initialized arbitrarily and a MBO surrogate dictionary is prepared for storing model information. At the beginning, reinforcement learning agent performs with a high ϵ, so that different pipelines could be tried out randomly to ensure enough exploration, and it decreases as time goes by to accelerate convergence. Each sampled pipelines at end of each episode is trained by MBO_PROBE (Algorithm 4), where MBO searches the hyper-parameter space for a few steps, which depends on the dimension of the conditional hyper-parameters. Specifically, MBO would run further $n^{probe} * length(hyperparameters)$ iterations based on MBO surrogate memory, or $(n^{init}+n^{probe})*length(hyperparameters)$ for a new generated pipeline. And as a result, hyper-parameter space with higher dimension could get more sampling budgets (iterations) of MBO after each episode. And then the cur-

rent best performance (or least loss value) found by MBO will be taken as reward to guide the reinforcement learning agent towards a better policy.

Algorithm 3 ML-ReinBo

Require: dataset $\mathcal{D}$, operations and Hyper-parameters
  Initialize Policy π
  Initialize MBO Surrogate Dictionary $\mathcal{R} \leftarrow \emptyset$
  while Budget not reached **do**
    Roll-out a **unconfigured** pipeline $\prod a_i$ according to policy π
    Get hyper-parameters set for the ground pipeline $\Lambda(\prod a_i) = \prod_i \Phi_{a_i}$
    Reward $R \leftarrow$ MBO_PROBE$(\prod a_i, \Lambda, \mathcal{R})$
    Update Policy π with reinforcement learning algorithm
  end while

Algorithm 4 MBO_PROBE$(\prod a_i, \Lambda, \mathcal{R})$

Require:
  if $\mathcal{R}\{\prod_i a_i\} = \emptyset$ **then**
    generate initial design of size n^{init} for surrogate model;
    for j in $1 : n^{init}$ **do**
      evaluate each design by initiating the pipeline with corresponding hyper-parameters.
    end for
    initialize $\mathcal{R}\{\prod_i a_i\}$
  end if
  for j in $1 : n^{probe}$ **do**
    propose new point according to surrogate model $\mathcal{R}\{\prod a_i\}$
    evaluate new point
  end for
  return $y \leftarrow$ best accuracy until now

4.2 Design of state space

For a reinforcement learning algorithm, the most crucial part is to encode the state space. Two reinforcement learning variants will be introduced in this thesis with different state encoding styles, i.e. the Tabular Q learning and the Deep Q learning. For both methods, we use the implementation in rlR[2]. A simple example of potential operations for a 3-stage supervised classification task is listed in the following Table.

[2]https://smilesun.github.io/rlR/

Table 1: Available operations/actions at each stage

Stage		operation/Action		
1	Pre-process	Scale	Certer	NA
2	Feature filter	PCA	FilterAnova	NA
3	Classifier	KNN	Xgboost	SVM

An operation of "NA" here is used to indicate that no operation would be taken in the corresponding stage to add more flexibility to the pipeline space.

4.2.1 State space design with Tabular Q Learning

Each state in the Tabular Q learning is defined as a string of variable length. At the beginning of each episode it is initialized with $S_0 =$ "s". At a state of S_i, if action $A_i \in \mathcal{A}_{S_i}$ is taken according to current policy π, it transitions to state $S_{i+1} = S_i.append(A_i)$, thus, the state string is always extended with actions in sequence and the transition between states is deterministic. Therefore, state S_i always contains all historical information from start to i-th stage. For the example in Table 1, if the available actions at each stage are denoted as 1 to 3, then a state string of "$s213$" indicates a trajectory of "Center" - "PCA" - "SVM".

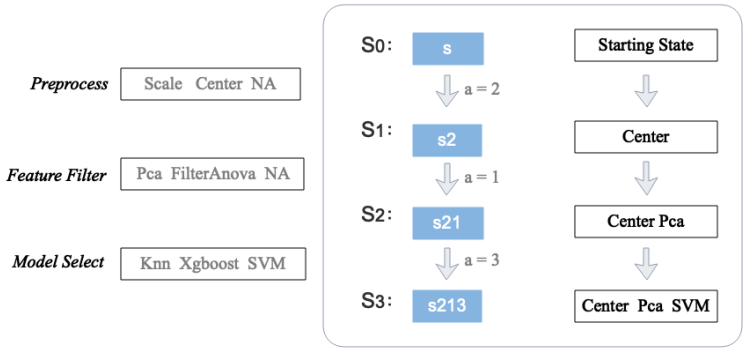

Figure 4: An example of state transition in Tabular Q Learning.

It is notable that we have 3 available operations for each stage here, but actually the number of potential actions at each state in Tabular Q Learning has not to be the same.

4.2.2 State space design with DQN

In the common DQN framework, unlike in Tabular Q learning, each state must be encoded as numerics of same dimension and the number of actions at each state has also to be the same, since DQN uses a neural network to proximate Q values and the output layer has a prefixed dimension.

Each state in our case could be encoded as a flat vector of all relevant operations. Starting from $S_0 = (0, 0, 0.., 0)$, each element corresponds to one specific operation. In case of the example above with 3 stages and 3 operations for each stage, each state has a dimension of 9 (3*3). If A_i is selected at state S_i, the corresponding element for the operation/action will be changed from 0 to 1, i.e. S_{i+1}: $S_i[A_i] = 1$. An example of state transitions in DQN is shown in Figure 5.

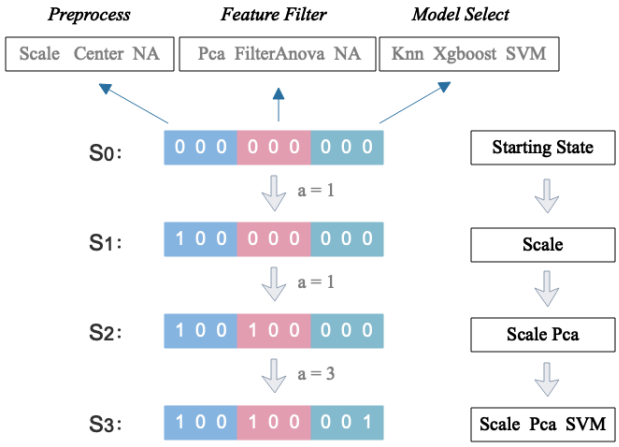

Figure 5: An example of state transition in DQN.

It would never happen that one operation has been selected more than once because the agent always steps in a direct and acylic way and operations in each stage are totally different. Hence, all previous information can be also tracked from state S_i. Another two possible encoding styles of states in DQN could be found in Appendix.

5 Baselines

To evaluate the general empirical behavior of the ML-ReinBo algorithm objectively, we performed some popular optimization algorithms and the state of art AutoML systems in the empirical experiments as baselines.

5.1 Random search

A straight-forward method for machine learning pipeline search is Random Search through ancestral sampling, which is most widely used strategy for hyper-parameter optimization. It is a natural baseline against which to judge one model-based or more complex optimization algorithm. In this thesis, the ancestral sampling of pipeline components and hyper-parameters is implemented by the R package *parabox*[3] , which provides functionality to specify a graph-structured configuration space and the conditional ancestral sampling on the conditional space.

5.2 Tree-structured Parzen Estimator Approach (TPE)

The second comparison algorithm is TPE[5], which is a sequential model-based optimization (SMBO) approach and also a powerful method for solving such black-box optimization problem. Unlike the common Gaussian-process based approach, TPE uses Parzen Windows to construct two non-parametric densities on top of a tree like hyper-parameter set, which enables it to solve problems with hierarchical space. TPE is implemented by Python library *Hyperopt*[4] for optimizing discrete and conditional dimensions.

5.3 Irace in pipeline search

Irace[19], an extension of Iterated F-race, searches the configuration space effectively with some probability model $p(X)$ defined over the parameter space X. Since Irace allows conditional hyper-parameter inputs, the machine learning pipeline space can be easily set up according to the API of Irace. Specifically, each configuration in Irace corresponds to a machine learning pipeline candidate and the instances in this case could be the data set with different *hold-out* (or *cross-validation*) resampling strategies. Thus, Irace will evaluate each machine learning pipeline with average performance on many different *hold-out* (or *cross-validation*) datasets, throw significantly worse candidates through Friedman-test and focus more and more on the most promising pipelines.

5.4 Auto-sklearn

Auto-sklearn[13] is an automated machine learning toolkit to automatically find a high-quality machine learning pipeline, which is built upon the *scikit-learn* machine learning library. Auto-sklearn extends the idea of Auto-

[3]https://github.com/smilesun/parabox

WEKA[17], uses meta-learning to speed up the optimization process and builds ensemble of models. But the core algorithm of Auto-sklearn is still SMAC[15], which is proposed to be compared with ML-ReinBo in this thesis.

5.5 Tree-Based Pipeline Optimization Tool (TPOT)

Another AutoML system to be compared with ML-ReinBo is TPOT[24], one of the very first AutoML methods, which is also built on top of *scikit-learn*. It is based on genetic programming algorithm and considers pipelines with multiple pre-processing steps as well as multiple ways to ensemble or stack the algorithms. It could construct machine learning pipelines of arbitrary length using *scikit-learn* algorithms and xgboost. TPOT provides several configuration space possibilities. For example, the default configuration of TPOT in document searches over a broad range of pre-processors, feature constructors, feature selectors, models, and parameters. Some of these operators are complex and may take a long time to run. However, the light configuration of TPOT uses a built-in configuration with only fast models and pre-processors.

6 Empirical experiments

6.1 Search space

ML-ReinBo is implemented in R in this thesis mainly with packages *rlR*[4], *mlr*[8], *mlrMBO*[9] and *mlrCPO*[5], where *rlR* provides APIs for the reinforcement learning algorithms, *mlr* as well as its extensions *mlrMBO* and *mlrCPO* conducts hyper-parameters tuning process and trains machine learning models. However, ML-ReinBo algorithm could actually be implemented with any machine learning library.

Notably, it is almost impossible to set up a perfectly fair comparison framework for the baseline methods above, since different machine learning libraries include different pre-processing operators and algorithms and even the implementation of the same algorithm could also vary across different libraries.

Concerning the potential impact of different machine learning libraries and different operations, Random Search through ancestral sampling, Irace and TPE also take *mlr* and *mlrCPO* as backends to evaluate machine learning models and search the same machine learning pipeline space as ML-

[4]https://smilesun.github.io/rlR/

[5]https://github.com/mlr-org/mlrCPO

ReinBo does. In the empirical experiments, we set up a 3-stage of machine learning pipeline process for supervised classification tasks including preprocessing, feature filtering and classifier selection. At each stage, 5 potential operations/actions are provided, where the operations are constructed by *mlrCPO* (listed in Table 2) and could be integrated with *mlr* to perform machine learning pipeline evaluations.

Table 2: Operations/Actions in Experiments

	Stage	Operation/Action				
1	Preprocess	cpoScale	cpoScale(center=FALSE)	cpoScale(scale=FALSE)	cpoSpatialSign*	NA
2	Feature Filter	cpoPca	cpoFilterKruskal	cpoFilterAnova	cpoFilterUnivariate	NA
3	Classifier	classif.ksvm	classif.xgboost	classif.ranger	classif.naiveBayes	classif.kknn

* Normalize values row-wise

The relevant conditional hyper-parameters for the operations are listed in Table 3. This pipeline pool (operations and hyper-parameters in Table 2 and 3) is used for ML-ReinBo, Irace pipeline search, TPE, and Random Search through ancestral sampling in the empirical experiments.

Table 3: List of Hyper-parameters

Operation	Parameter	Type	Range
Anova,Kruskal,Univariate	perc	numeric	(0.1,1)
Pca	rank	integer	(p/10,p)
kknn	k	integer	(1,20)
ksvm	C	numeric	$(2^{-15},2^{15})$
ksvm	sigma	numeric	$(2^{-15},2^{15})$
ranger	mtry	integer	(p/10,p/1.5)
ranger	sample.fraction	numeric	(0.1,1)
xgboost	eta	numeric	(0.001,0.3)
xgboost	max_depth	integer	(1,15)
xgboost	subsample	numeric	(0.5,1)
xgboost	colsample_bytree	numeric	(0.5,1)
xgboost	min_child_weight	numeric	(0,50)
naiveBayes	laplace	numeric	(0.01,100)

However, the search space are different for Auto-sklearn and TPOT since they are based on Python library *scikit-learn*. The default configuration space of TPOT contains a lot of operators while the light version of TPOT provides only fast models and pre-processors. The light TPOT is therefore less time-consuming but it could probably lead to lower accuracy in consequence. For this reason, we compare ML-ReinBo with both TPOT with the default configuration and TPOT with light configuration, and we call them TPOT and TPOT_light respectively in case of any confusion. As for Autosklearn, we use the default search space and it has wrapped a total of 15 classifiers,

14 feature pre-processing methods and 4 data pre-processing methods from *scikit-learn*. Furthermore, we do not use meta-learning in Auto-sklearn and also disable the ensemble process in comparison with ML-ReinBo.

6.2 Benchmarking datasets

A set of standard benchmarking datasets of high quality are collected from OpenML-CC18[6] and OpenML100 repository[7], which are rather well-curated from many thousands and have diverse numbers of classes, features, observations, as well as various ratios of the minority and majority class size. The information for these datasets are listed in the following table.

Table 4: Datasets from the OpenML repository. Information of each dataset listed here: the OpenML task_id and name, the number of classes (nClass), features (nFeat) and observations (nObs), as well as the ratio of the minority and majority class sizes (rMinMaj)

id	name	nClass	nFeat	nObs	rMinMaj	omlcc18	oml100
14	mfeat-fourier	10	77	2000	1.00	TRUE	TRUE
23	cmc	3	10	1473	0.53	TRUE	TRUE
37	diabetes	2	9	768	0.54	TRUE	TRUE
53	vehicle	4	19	846	0.91	TRUE	TRUE
3917	kc1	2	22	2109	0.18	TRUE	TRUE
9946	wdbc	2	31	569	0.59	TRUE	TRUE
9952	phoneme	2	6	5404	0.42	TRUE	TRUE
9978	ozone-level-8hr	2	73	2534	0.07	TRUE	TRUE
125921	LED-display-domain-7digit	10	8	500	0.65	FALSE	TRUE
146817	steel-plates-fault	7	28	1941	0.08	TRUE	FALSE
146820	wilt	2	6	4839	0.06	TRUE	FALSE

6.3 Experiment settings

Considering different performances of hardware like various types of CPU, Memory configurations and so on, instead of using running time as budget, the number of pipeline configurations to be evaluated during the optimization process is used in the experiments here as budget for each algorithm. Specifically, each algorithm is assigned with resources of 1000 times of a 5-fold cross-validation (*CV5*) resampling process to search in the machine learning pipeline space for a given data set.

Accordingly, for a given task, each algorithm could evaluate (at most) 1000 pipeline configurations with *CV5* resampling strategy on the data set and then take the pipeline with the least value of mean miss-classification

[6]https://www.openml.org/s/99

error (mmce) as the best model. However, Irace trains each pipeline on many different instances due to its racing property while other algorithms evaluate each candidate configuration with *CV5* only once. As a result, we found Irace could only search much less pipelines within the budget of 1000 times of *CV5* in the previous experiments and performed obviously worse than others. For this reason, we give Irace a budget of 5000 (1000*5) times of *hold-out* resampling strategy now, thus, each instance in Irace is defined as a different *hold-out* dataset, which allows Irace to search more pipelines and the budget is kind of equivalent to 1000 times of *CV5* for other algorithms. And Irace will take the pipeline with the least average mmce across different instances as the best model.

It should be aware that there are two types of errors, thus, the optimization error we get during the optimization process to find the best model, and another more reliable error we should consider – test error.

Since the optimization process has potential overfitting behavior, a nested cross-validation (NCV) strategy is used to evaluate the generalization capability of algorithms. The original data set D is partitioned into D^{opt} and D^{test} for calculating optimization error and test error. Specifically, D^{opt} and D^{lock} are created by a *CV5* resampling strategy, which means that the dataset D is partitioned into 5 subsets of equal size and in the b-th iteration, the b-th subset is used as D^{test}, while the union of the remaining parts forms D^{opt}. The split of D^{opt} and D^{test} corresponds to the outer loop of NCV and we always have $D = D^{opt} \bigcup D^{test}$ and $D^{opt} \bigcap D^{test} = \emptyset$.

In each iteration, thus, in the inner loop of NCV, D^{opt} is used for algorithm to find the best pipeline within the budget of 1000 times of *CV5* while D^{test}, which has never been seen during the optimization process, is used to test the best pipeline. The performance of the algorithm on the data set will be finally calculated by the average value of 5 testing errors on D^{test}.

Besides, as the search space we defined in Section 6.1 is relatively small, we would only apply ML-ReinBo with Tabular Q Learning in the empirical experiments.

6.4 Experiment results

Optimization process

At first, an overview of the optimization process is provided in Figure 6, where ML-ReinBo is compared with TPE and Random Search through ancestral sampling. These 3 algorithms searched the same machine learning pipeline space defined in Section 6.1 and each algorithm was given a budget constraint of 1000 times of *CV5* on each data set to find the best configuration with

least mmce during the optimization process.

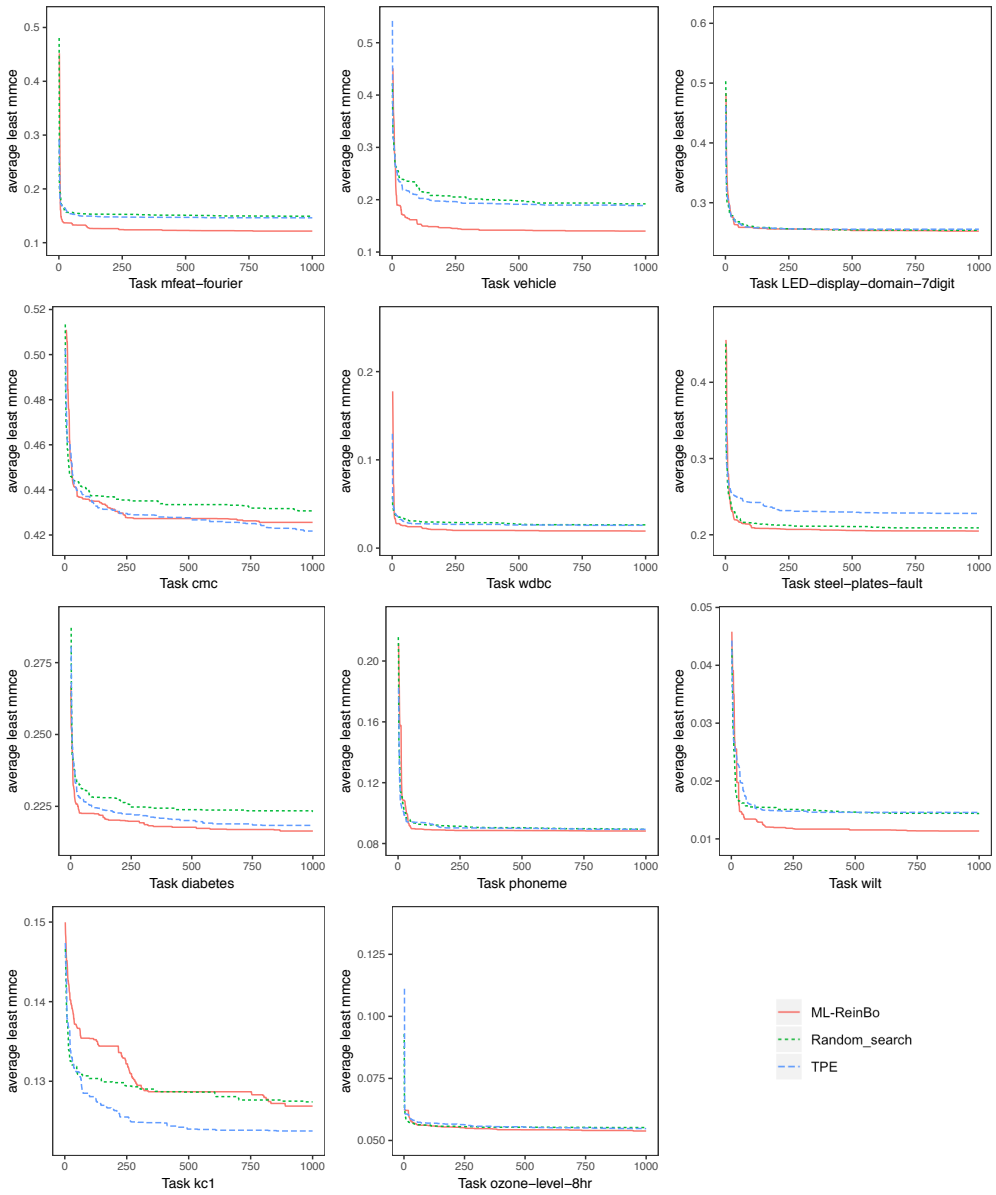

Figure 6: Average optimization error across 10 trials. In each trial, ML-ReinBo, Random Search and TPE search the same pipeline space and each algorithm is allocated with the same budget of 1000 times of *CV5* resampling process. The average best performance (least mmce) achieved by each algorithm across 10 trials is plotted here against budget.

We performed 10 runs for each algorithm on each data set and plot the

average optimization errors across the 10 independent trials against budget in Figure 6. Irace is excluded here because it uses different resampling strategy (*hold-out*) while Auto-sklearn and TPOT are not shown here due to the different search spaces.

As can be seen from Figure 6, in most cases, explicitly for 9 of the 11 benchmarking datasets/tasks we collected from OpenML repository (except for *cmc* and *kc*1), ML-ReinBo found a good configuration faster than TPE and Random Search through ancestral sampling during the optimization process, and the average least mmce ML-ReinBo achived within the budget constraint is always lower or almost equal to the other two algorithms.

Validation/Test errors

To evaluate how well the best machine learning pipeline selected by each algorithm during the optimization process could generalize to unseen data, the nested cross-validation (NCV) strategy is used to test the configuration on the validation data set D^{test}. D^{test} and D^{opt} are split by $CV5$ and in each iteration, each algorithm is given a total budget of 1000 times of $CV5$ on D^{opt} to search for the best configuration as mentioned in Section 6.3. And the aggregated mmce (mean miss-classification error) value across 5 iterations is taken as performance measure of algorithms. This NCV process is repeated 10 times for each algorithm on each task/data set with different seeds and the mean value across these 10 runs are shown in Table 5. Meanwhile, the distribution of the explicit 10 aggregated values for each algorithm are displayed in Figure 7.

Table 5: Average performance (mmce) of algorithms across 10 trials. 10 independent runs are performed for all algorithms on each data set and in each run the aggregated mmce based on NCV is taken as performance measure for each algorithm. Each value in this table is the mean value of the aggregated mmce values across 10 trials and the bold-faced values indicate that the algorithm does not perform significantly worse than the best algorithm on the corresponding task based on Mann-Whitney U test.

name	**ML-ReinBo**	**Irace**	**Random_search**	**TPE**	**Auto-sklearn**	**TPOT_light**	**TPOT**
mfeat-fourier	0.1269	**0.1225**	0.1581	0.1542	0.1412	0.1490	0.1451
cmc	**0.4488**	**0.4453**	**0.4500**	**0.4485**	0.4470	**0.4506**	**0.4457**
diabetes	**0.2426**	**0.2431**	**0.2455**	**0.2436**	**0.2483**	**0.2413**	0.2452
vehicle	**0.1580**	**0.1611**	0.2020	0.2117	0.1679	0.2057	0.1784
kc1	**0.1358**	**0.1339**	**0.1353**	**0.1351**	0.1421	0.1438	0.1380
wdbc	**0.0274**	**0.0299**	0.0341	0.0348	**0.0299**	**0.0264**	0.0353
phoneme	**0.0911**	**0.0900**	**0.0912**	**0.0920**	**0.0902**	0.1016	**0.0893**
ozone-level-8hr	**0.0591**	**0.0585**	0.0598	0.0601	**0.0588**	0.0603	**0.0577**
LED-display-domain-7digit	0.2678	0.2685	0.2696	0.2702	0.4738	**0.2622**	**0.2608**
steel-plates-fault	0.2128	0.2112	0.2146	0.2330	0.2041	0.2601	**0.1985**
wilt	**0.0126**	**0.0131**	0.0161	0.0159	0.0132	0.0164	0.0141

A Mann-Whitney U test can test whether a randomly selected value from one sample will be less than or greater than a randomly selected value from a second sample, which is used here to test whether an algorithm performs significantly worse than another algorithm based on the 10 aggregated mmce values for the individual task.

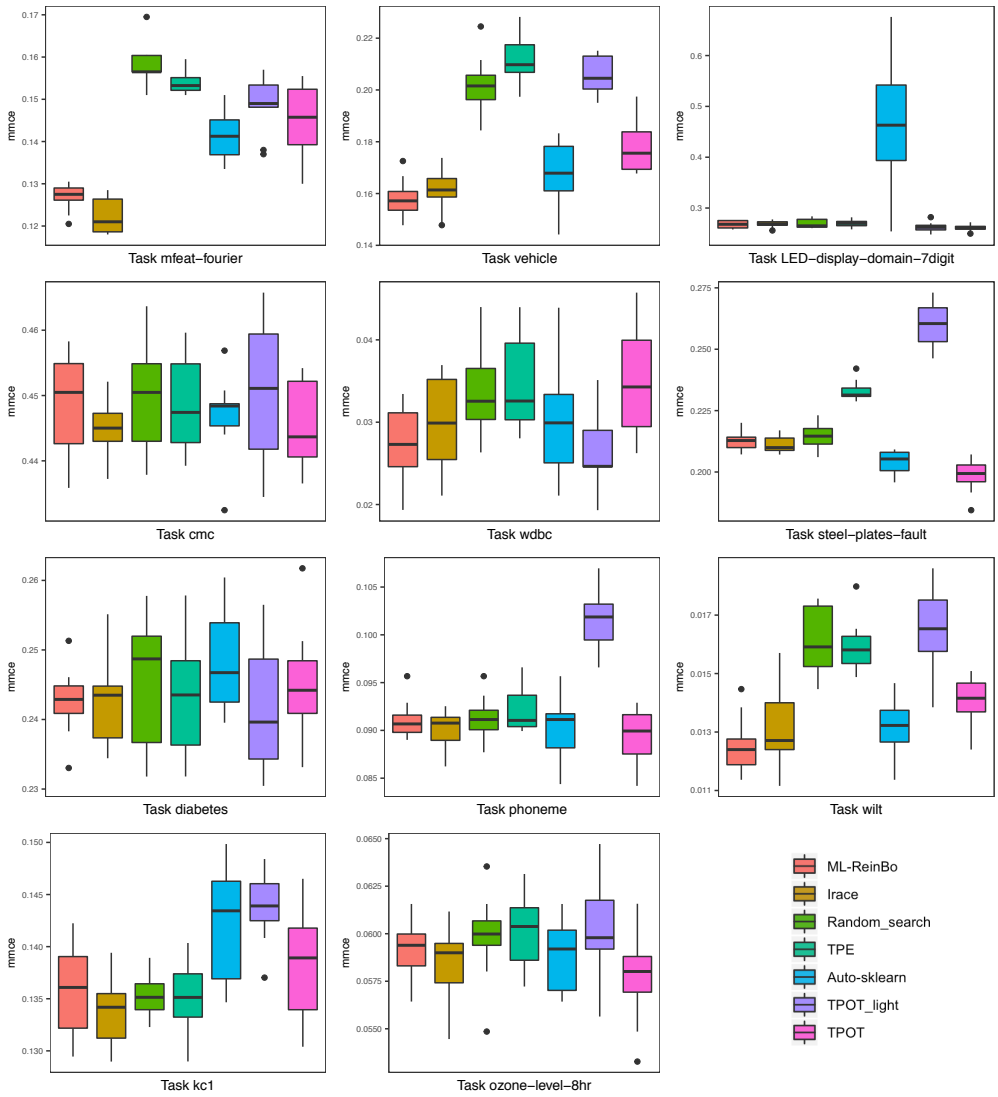

Figure 7: Boxplots for performance of algorithms in 10 trials of NCV resampling strategy. For each algorithm, the distribution of the 10 performance values based on *NCV* for each task is plotted here underlying mean values in Table 5.

The bold-faced values in Table 5 indicate that the algorithm does not

perform significantly worse than the best algorithm on the corresponding task/data set. In addition, the test result between ML-ReinBo and other algorithms is shown in Table 6.

As shown in Table 5 and Figure 7, overall, Irace and ML-ReinBo empirically perform the best in comparison with other algorithms, where Irace has the best performance (statistically not worse than the best one) for 9 of 11 tasks and ML-ReinBo performs almost equally to Irace for all tasks except for *mfeat-fourier*. At the same time, both Irace and ML-ReinBo have relatively shorter box range than other algorithms in most cases, which indicates that they have more stable behaviours. On 8 of the 11 datasets, ML-ReinBo has even shorter box range than Irace. And we also found that within the 5000 of *hold-out* budget, the number of configurations that Irace has evaluated varies from 499 to 930.

Table 6: Comparison between ML-ReinBo and other algorithms on 11 benchmarking datasets based on Mann-Whitney U test. Each pair of algorithms are tested with the 10 values from boxplot in Figure 7 on each dataset to check whether they significantly win or lose to each other with con (significance level $\alpha = 0.05$).

		Irace	**Random_search**	**TPE**	**Auto-sklearn**	**TPOT_light**	**TPOT**
ML-ReinBo	win	0	4	5	5	6	4
	tie	10	7	6	5	5	5
	lose	1	0	0	1	0	2

TPOT is among the best on 5 of 11 datasets and not surprisingly, it has performed considerably better than TPOT_light in the empirical experiments since TPOT_light has smaller search space with only fast models and preprocessors. However, according to the Mann-Whitney U test, even TPOT with the default configuration space has only won ML-ReinBo on 2 datasets and lost on 4 datasets.

Auto-sklearn without meta-learning and ensemble method performs the best on 4 of 11 datasets. It is better than Random Search and TPE in most cases but compared to ML-ReinBo, it has only won once and behaved worse than ML-ReinBo on 5 of 11 datasets. And it performs extremely bad on the task *LED-display-domain-7digit* as plotted in Figure 7.

As mentioned in Optimization process, ML-ReinBo found a good machine learning pipeline faster than Random Search through ancestral sampling and TPE. In terms of the validation errors, ML-ReinBo also performed much more better than them, where ML-ReinBo has significantly won Random Search on 4 of 11 tasks, won TPE on 5 of 11 tasks and for other tasks they produced almost equally good results.

Computing time

The budget constraint in this thesis is the number of configuration pipelines to be evaluated regarding the resampling strategy. Comparing the algorithms in this way instead of computing time has ignored the cost of complexity of each algorithm, thus, how each algorithm works to select configuration pipelines during the optimization process. For example, the time to update Q-table in reinforcement learning and the Bayesian optimization method in ML-ReinBo, the Friedman test and the update of model to select configurations in each iteration in Irace. The following Figure shows the average time that each algorithm took to implement one NCV process per task in the empirical experiments.

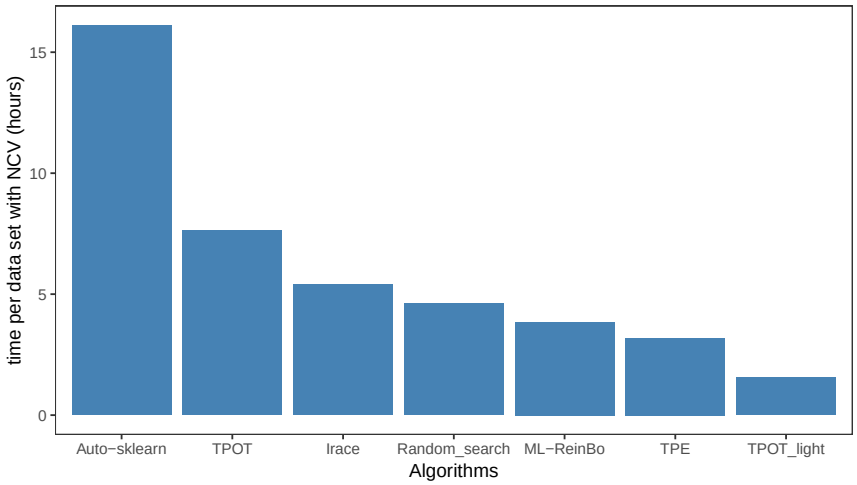

Figure 8: Sorted barplot of the running time for each algorithm in average. The value reported here corresponds to the average running time of each algorithm per data set based on the NCV strategy.

From the figure, it can be seen that Auto-sklearn is the most time-consuming algorithm in the empirical experiments. It has taken more than twice of the time that the second most time-consuming algorithm TPOT costed in average. TPOT has performed better than TPOT_light at the cost of much more computing time since it includes more complex pre-processors and models. It is not surprising that it could take less time for ML-ReinBo and TPE than Random Search since the model-based algorithms focus more and more on the promising candidate configurations, which sometimes might be less complex, while Random Search through ancestral sampling samples pipelines totally randomly, it could take more or less time than model-based

algorithms. Furthermore, ML-ReinBo spent slightly less time than Irace in average.

Selected models

Besides comparing the performance of optimization algorithms and AutoML systems, it is also worth looking into the best machine learning pipelines selected by each algorithm during the optimization process, which might guide us to better configure the search space for an AutoML system.

Since ML-ReinBo and Irace have performed the best among all algorithms in the empirical experiments and this thesis is more interested in the operations defined in Table 2, the frequency of each operation included in the best models selected by ML-ReinBo and Irace is listed in the following table.

Table 7: Frequency of each operator selected by ML-ReinBo and Irace in the empirical experiments.

Preprocess	**ML-ReinBo**	**Irace**	**Filter**	**ML-ReinBo**	**Irace**	**Classifier**	**ML-ReinBo**	**Irace**
Scale	252	126	Anova	146	82	ksvm	252	212
Scale(scale=FALSE)	271	125	Kruskal	176	124	ranger	263	284
Scale(center=FALSE)	9	102	Univariate	85	29	xgboost	14	2
SpatialSign	7	47	PCA	87	91	kknn	17	25
NA	11	150	NA	56	224	naiveBayes	4	27

Compared to the models selected by ML-ReinBo, Irace has chosen more models without pre-processing or feature engineering. *SpatialSign*, which normalize values row-wise, seems not preferred in pre-processing stage. In feature filtering stage, *Kruskal* is selected relatively frequently. As for classifiers, ML-ReinBo and Irace have very similar results that *ksvm* and *ranger* performed the best in most cases.

7 Summary and future work

Inspired by MetaQNN[1], which optimizes the neural network pipeline by reinforcement learning, we proposed a new AutoML algorithm ML-ReinBo to design machine learning pipelines with reinforcement learning.

By embedding Bayesian optimization into reinforcement learning, ML-ReinBo is able to address such problem with hierarchical hyper-parameter space, where the reinforcement learning agent learns to optimize the pipeline composition and Bayesian optimization takes care of the hyper-parameters conditional on the pipeline operations. Considering the evaluation of machine learning pipeline is typically expensive, similar to Hyperband, Irace

and some other AutoML systems, ML-ReinBo is basically to reasonably allocate resources to different configurations to balance the trade-off between exploration and exploitation. Thus, the reinforcement learning policy guides the agent to select a particular configuration to exploit more hyper-parameter configurations by Bayesian optimization but also gives it chances to explore unseen pipelines by ϵ-greedy strategy. And as time goes on, by decreasing the possibility of exploring new or sub-optimal pipelines ML-ReinBo focuses more and more on the most promising configuration pipelines.

We have performed some experiments to evaluate the general empirical behaviours of ML-ReinBo and set up some baselines with as fair as possible training budget. The results show that ML-ReinBo found a good configuration faster than Random Search through ancestral sampling and TPE in most cases, and also performed considerably better than them regarding the test accuracy. Compared to Auto-sklearn without meta-learning and ensemble method, TPOT and TPOT with light configuration space, ML-ReinBo also performed more favorably and stably. In addition, we set up Irace to search machine learning pipeline space and considering its racing property, we give it 1000*5 *hold-out* instances instead of 1000 times of *CV5* (for other algorithms) to give it more flexibility. The empirical experiments implied that Irace and ML-ReinBo produced almost equally good results on the benchmarking datasets.

Compared to other AutoML systems, ML-ReinBo has an advantage of interpretability because of the use of reinforcement learning algorithm. In case of using Tabular Q learning, there is a Q table including state-action values $Q(s, a)$, which could suggest which operation to choose at the next stage given the current particular state. The larger the value, the more preferable the operation, and the values are iteratively updated by reinforcement learning algorithm during the optimization process.

For the two ML-ReinBo variants, we recommend using ReinBo-Table for better sample efficiency. However, ReinBo-DQN could deal with larger and continuous state space, which could be particularly useful if we want to combine meta features of the datasets into the reinforcement learning state space.

For future work, it would be interesting to include meta learning into ML-ReinBo, which does not only learn how to construct a pipeline and configure it for a given dataset, but also learn how to generalize the learned policy to a wide range of datasets by learning jointly on the meta features. Additionally, it would be nice to see how ReinBo performs on jointly optimizing neural architecture and hyper-parameters like learning rate and momentum, etc. Another possibility could be to optimize the machine learning pipeline operations with conditional hyper-parameters only with reinforce-

ment learning, which requires reinforcement learning to deal with conditional and continuous parameters.

Acknowledgement

I would like to express my very great appreciation to Prof. Dr. Bernd Bischl and Xudong Sun, my supervisors, who have provided me with very valuable and constructive suggestions during the planning and development of this thesis. I am also particularly grateful for Xudong Sun's patient guidance and help in the technical problems during this work.

I would also like to thank Janek Thomas for his valuable suggestions, Martin Binder for his help with the mlrCPO settings and Florian Pfisterer for his help with Auto-sklearn setup in R.

Furthermore, there would probably be a submitted paper about ML-ReinBo to some conference later, which is written by Xudong Sun, Jiali Lin and Bernd Bischl. The basic concept, algorithm and experiments will be similar to this thesis.

References

[1] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *International Conference on Learning Representations*, 2017.

[2] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic programming: an introduction*, volume 1. Morgan Kaufmann San Francisco, 1998.

[3] Juan Cruz Barsce, Jorge A Palombarini, and Ernesto C Martínez. Towards autonomous reinforcement learning: Automatic setting of hyperparameters using bayesian optimization. In *2017 XLIII Latin American Computer Conference (CLEI)*, pages 1–9. IEEE, 2017.

[4] James Bergstra, Dan Yamins, and David D Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in Science Conference*, pages 13–20. Citeseer, 2013.

[5] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[6] Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle. F-race and iterated f-race: An overview. In *Experimental methods for the analysis of optimization algorithms*, pages 311–336. Springer, 2010.

[7] Bernd Bischl, Giuseppe Casalicchio, Matthias Feurer, Frank Hutter, Michel Lang, Rafael G Mantovani, Jan N van Rijn, and Joaquin Vanschoren. Openml benchmarking suites and the openml100. *arXiv preprint arXiv:1708.03731*, 2017.

[8] Bernd Bischl, Michel Lang, Lars Kotthoff, Julia Schiffner, Jakob Richter, Erich Studerus, Giuseppe Casalicchio, and Zachary M. Jones. mlr: Machine learning in r. *Journal of Machine Learning Research*, 17(170):1–5, 2016.

[9] Bernd Bischl, Jakob Richter, Jakob Bossek, Daniel Horn, Janek Thomas, and Michel Lang. *mlrMBO: A Modular Framework for Model-Based Optimization of Expensive Black-Box Functions*, 2017.

[10] Alex GC de Sá, Walter José GS Pinto, Luiz Otavio VB Oliveira, and Gisele L Pappa. Recipe: a grammar-based framework for automatically

evolving classification pipelines. In *European Conference on Genetic Programming*, pages 246–261. Springer, 2017.

[11] Avinash K Dixit, John JF Sherrerd, et al. *Optimization in economic theory*. Oxford University Press on Demand, 1990.

[12] Stefan Falkner, Aaron Klein, and Frank Hutter. Practical hyperparameter optimization for deep learning. 2018.

[13] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[14] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[15] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.

[16] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[17] Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *The Journal of Machine Learning Research*, 18(1):826–830, 2017.

[18] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[19] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[22] Jonas Mockus. *Bayesian approach to global optimization: theory and applications*, volume 37. Springer Science & Business Media, 2012.

[23] Felix Mohr, Marcel Wever, and Eyke Hüllermeier. Ml-plan: Automated machine learning via hierarchical planning. *Machine Learning*, 107(8-10):1495–1515, 2018.

[24] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on Automatic Machine Learning*, pages 66–74, 2016.

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

[26] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer School on Machine Learning*, pages 63–71. Springer, 2003.

[27] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[28] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[29] Martijn van Otterlo and Marco Wiering. Reinforcement learning and markov decision processes. In *Reinforcement Learning*, pages 3–42. Springer, 2012.

[30] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[31] Fangkai Yang, Steven Gustafson, Alexander Elkholy, Daoming Lyu, and Bo Liu. Program search for machine learning pipelines leveraging symbolic planning and reinforcement learning. In *Genetic Programming Theory and Practice XVI*, pages 209–231. Springer, 2019.

[32] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

A Potential state encoding methods in DQN

If we have a lot of available actions at each stage, the encoding method mentioned in Section 4.2.2 (Figure 5) will not be efficient since the dimension of the state will grow largely. Another state encoding method displayed by Figure 9 is inspired by MetaQNN. Instead of stacking all operations in a flat vector of large dimension, we could use a separate one-hot encoding vector to present the stage/depth of the current state and another one-hot encoding vector to indicate which action has be selected in length of available actions in one stage. Accordingly, the dimension could be largely reduced. But one drawback might be that each state only contains information of the action selected in the last step.

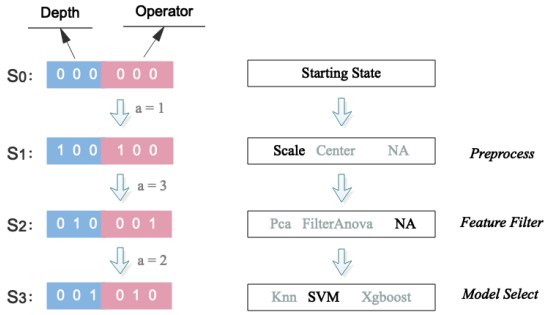

Figure 9: State transitions in DQN (2)

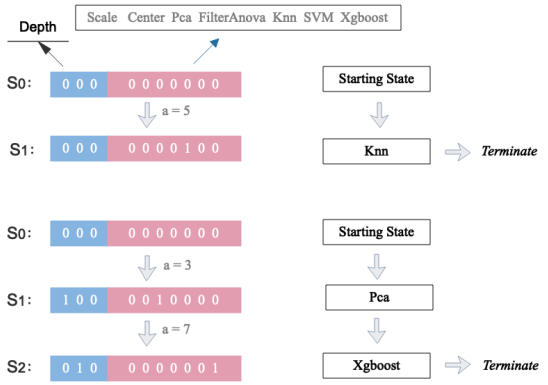

Figure 10: State transitions in DQN (3)

The state encoding method in Figure 10 is a more flexible method. It could take arbitrary operation at each step and if a classifier is selected, the reinforcement learning episode will terminate. In this case, we take machine

learning as a totally black box and do not care whether it must firstly have a pre-processor and then filter the features or whether it has applied 2 kinds of feature filters for one given data set, which seems uncommon but might make sense in some special cases. (Like TPOT, it could construct machine learning pipelines of arbitrary length.)