Using RNN and Reinforcement Learning to generate Machine Learning Pipeline

Ruiqi Zhong
Columbia University
rz2383@columbia.edu

Haoyan Min
Columbia University
hm2689@columbia.edu

Yilan Ji
Columbia University
yj2425@columbia.edu

Abstract

In this project we aim to train a neural network that can generate machine learning pipeline consisting of a small set of basic sklearn primitives. We first train an RNN based on the behavior of auto-sklearn on a set of simulated data set; then we use reinforcement learning to let it explore better alternatives in searching for a good machine learning model. We hope that by the end of our project, the trained program can at least beat the randomized hyper parameter search and possibly bayesian optimization. As was agreed in the meeting, we spent most of our time designing our network architecture and reinforcement learning algorithm in this milestone, rather than trying to code things up and get preliminary results.

1. Introduction

In recent years the combination of reinforcement learning and deep learning has led to significant technological breakthroughs. More profoundly, programs starting with zero knowledge can learn from simulated environments and achieve super-human performance[1][2][3]. On the other hand, big data and machine learning are more widely employed by the industry and there is an increasing demand for data analysts; nevertheless, many of their works are routine (e.g. parameter tuning). Can we train a meta-learning program that can automatically and efficiently find an optimal machine learning model?
Previous works have employed Bayesian optimization techniques (with or without labeled meta-data) and achieved respectable performance[4][5][6] Bayesian optimization employs the Bayesian technique of setting a prior over the objective function and combining it with evidence to get a posterior function. This permits a utility-based selection of the next observation to make on the objective function, which must take into account both exploration (sampling from areas of high uncertainty) and exploitation (sampling areas likely to offer improvement over the current best observation).). However, they fail to take into account other important meta-data such as time required to train a model, training accuracy, etc while searching for the optimal model. In this project, we attempt to take these factors into account and use reinforcement learning and RNN to generate a meta-learning program composed of scikit-learn primitives that can quickly and efficiently search for a near optimal model m^* on data set D. (see more rigid definition of an "optimal model" "CASH problem" here[7]). Such a program will not only validate experiences and techniques employed by data analysts, but also help us discover new ones.

2. Problem Formulation and Evaluation Method

2.1. CASH

The task of "automatically finding the best machine learning model" can be formulated as a search/optimization problem of *CASH Combined Algorithm Selection and Hyperparameter optimization*[7]:
Let $\mathcal{A} = \{A^{(1)}, ..., A^{(R)}\}$ be a set of algorithms, and let hyperparameters of each algorithm $A^{(j)}$ has domain $\Lambda^{(j)}$. Further, let $D_{train} = \{(x_1, y_1), ..., (x_n, y_n)\}$ be a training set which is split into K cross-validation folds $\{D^{(1)}_{valid}, ..., D^{(K)}_{valid}\}$ and $\{D^{(1)}_{train}, ..., D^{(K)}_{train}\}$ such that $D^{(i)}_{train} = D_{train} \backslash D^{(i)}_{valid}$ for $i = 1, ..., K$. Finally, let $\mathcal{L}(A^{(j)}_{\lambda}, D^{(i)}_{train}, D^{(i)}_{valid})$ denote the loss that algorithm $A^{(j)}$ achieves on $D^{(i)}_{valid}$ when trained on $D^{(i)}_{train}$ with hyperparameter λ. Then the Combined Algorithm Selection and Hyperparameter optimization (CASH) problem is to find the joint algorithm and hyperparameter setting that minimizes this loss:

$$A^*, \lambda_* \in \underset{A^{(j)} \in \mathcal{A}, \lambda \in \Lambda^{(j)}}{\operatorname{argmin}} \frac{1}{K} \sum_{i=1}^{K} \mathcal{L}(A^{(j)}_{\lambda}, D^{(i)}_{train}, D^{(i)}_{valid})$$

We want to find the model that can achieve the best cross validation score in a limited amount of time.

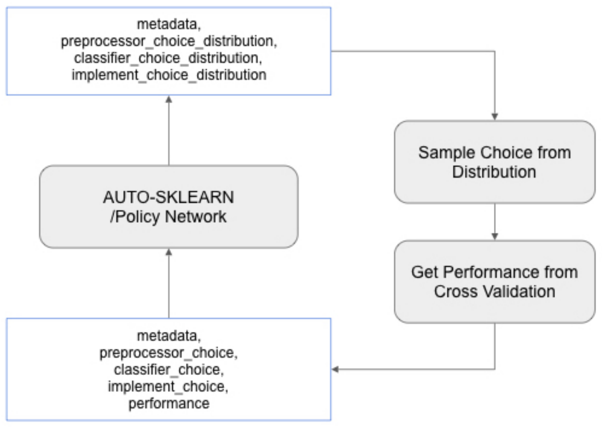
2.2. Evaluation method

Given two *CASH* algorithms $T^{(1)}, T^{(2)} \in \mathbb{T}$ (a family of algorithms that decides the sequence of $A \in \mathcal{A}$), we compare them in the following ways: on the set of data set $\mathbb{D}$, which percentage of time does the best model found by $T^{(1)}$ have higher cross validation score than $T^{(2)}$ within a given amount of time; here the ”time” can be the number of models explored or the actual run-time of the search algorithm. Since the RNN we train will represent a search algorithm $T \in \mathbb{T}$, and so are randomized search, bayesian optimization and auto-sklearn, we can use the method above to compare them. By the end of the project, we hope that the search algorithm represented by a trained RNN will at least outperform randomized search, hopefully bayesian optimization and ideally auto-sklearn, on the simualted data set, and idealy real world data set.

2.3. Our Limit

In this project we only generate machine learning code using sklearn as primitives. Also due to time limit, we only 1) limit ourselves to two classification algorithm - quadratic discriminant analysis and bernoulli naive bayes - and two preprocessing steps - PCA and no preprocessing step 2) consider binary classification problems 3) train our algorithm on a simulated data set with balanced class labels.

3. Method



metadata, preprocessor_choice_distribution, classifier_choice_distribution, implement_choice_distribution
AUTO-SKLEARN /Policy Network
Sample Choice from Distribution
Get Performance from Cross Validation
metadata, preprocessor_choice, classifier_choice, implement_choice, performance

Figure 1. Mimicry the behavior of Auto-sklearn

High Level Idea: Due to limit of computation power, we cannot start our neural network (which represents the search algorithm, same for the rest of this report) as Alpha Go Zero[1]. Therefore, we need to first train it to mimic the behavior of auto-sklearn, and then use reinforcement learning to let the neural network explore a better alternative search algorithm.

3.1. What does "pipeline" mean?

Here we do not distinguish pre-processing steps, feature processing and actual machine learning's optimizing the loss function process. We treat them as one: for each time step, we choose prepossessing steps, feature processing stpes and machine learning algorithm jointly, instead of deciding which type of step (pre-process or optimization) to take and its sub-choice (qda or berounlli naive bayes) at each time step, we output one choice as a whole that includes all its components.

3.2. Simulated Data Set

See the generating process on our github (link). Each of the data set has [100, 1000] data points with data points-dimensions ratio between [0, 0.5]. The coefficients β are generated either by uniform distribution, Laplace distribution or Gaussian distribution. The generation of X and label Y are as follows: $X_{hidden} \sim \mathbb{N}(0,1)$, $X_{basis} \sim f(X), X_{observed} \sim X_{hidden} + \mathbb{N}(0, 0.1)$, $y_{label} \sim Bernoulli(sigmoid(\beta X_{basis}))$,
where $X_{observed}$ and y_{label} are the dataset we obtain, f a function that with some probability transform a basis e.g. from x_1 to x_1^2. Each entry also has a probability to become ”not a number”, thus making imputation necessary.

3.3. Obtain the Behavior of Auto-Sklearn

To ”mimic” the behavior of auto-sklearn, we need to feed it with data sets and observe its interaction with the environment. Since for each data set, auto-sklearn will only generate 1 sequence of model choices, the number of datasets (¡150) on OpenML is not enough to train a neural network. Therefore, we generate a set of simulated data set (altogether 1250, explained below) and record the interaction history between auto-sklearn and each of the data set. We consider the first 20 models attempted by auto-sklearn and their performance on the data set.

3.4. RNN architecture - policy network

We use an RNN to mimic the behavior of auto-sklearn and learn a network that can depict the next move of auto-sklearn. (See fig 1 for detail) At each time 0, the RNN will take as input $x^{metadata}$; the RNN will output a distribution over all the model choices $y_0^{imputation}, y_0^{pre-process}, y_0^{classifier}$ (y_0 for brief). Then the environment will sample a deterministic model choice $x_1^{imputation}, x_1^{pre-process}, x_1^{classifier}$ from this distribution, run it on the data set through cross validation and obtain its actual performance $x_1^{performance}$. At time t, the RNN takes in input x_t, the last model choice and its performance and metadata, and output the distribution of model prediction y_t. We use cross entropy for each categorical choice (e.g. which imputation to use) and mean squared error of the log of hyper-parameters.

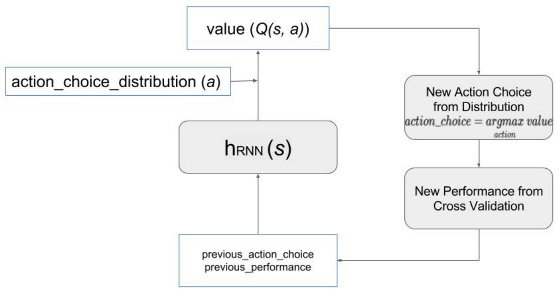

value (Q(s, a))
action_choice_distribution (a)
New Action Choice from Distribution
action_choice = argmax value
action
hRNN (s)
New Performance from Cross Validation
previous_action_choice
previous_performance

Figure 2. Q learning

3.5. RRN architecture - Q learning

The task of Q learning is to evaluate $Q(s, a)$, where s is the current state, a the action taken, and Q the value of the action a at state s. Here s corresponds to the inner state h of RNN, action a corresponds to the choice distribution output by the network. Here we define the reward r to be the accuracy percentage increase over the best model attempted in the past minus constant times the number of more steps used. For example, if mean imputation was used before, using mean imputation again will not decrease the reward; the reason is straightforward: data analysts will store the preprocessed data rather than running them again. During the phase of action evaluation, at each time step we choose the a_t (or y_t in the previous section) that maximizes $Q(s, a)$, sample x_t from the model distribution a_t and get its performance on the data set. Then h_{t+1} is updated given h_t and x_t. Hence, a_t will not directly influence how the inner state h updates, which only depends on metadata, model choice and model performance. During the phase of model training, we simply train an RNN that can predict the value of the state.
Though $Q(s, a)$ alone seems to suffice in this problem, we need the aforementioned RNN policy network to take advantage of learning the behavior of auto-sklearn: we observe from auto-sklearn only the sampled model choice instead of a model distribution; therefore, we need to learn the actual underlying distribution in order to train the Q learning network.

3.6. Implementations

The operation system we use is Ubuntu 14.04 on Google Cloud Platform, with 16 CPU, 60G RAM and 1000G storage. We write all our code in Python3, and the RNN is implemented in Keras. We obtain the behavior of auto-sklearn through the log it outputs and reproduce the performance of each model tried by auto-sklearn, since it does not provide API that allow the users to know the performance of each model attempted.

4. Preliminary Result

We have decided to settle down the entire architecture and algorithm first before getting things running and get some preliminary result. However, we did implement some preliminary functions. See our code on github page: https://github.com/PB12203006/Meta-Learning-with-Deep-Reinforcement-Learning

References

[1] Silver, David, et al. ”Mastering the game of go without human knowledge.” Nature 550.7676 (2017): 354-359.

[2] Andrew, Alex M. ”REINFORCEMENT LEARNING: AN INTRODUCTION by Richard S. Sutton and Andrew G. Barto, Adaptive Computation and Machine Learning series, MIT Press (Bradford Book), Cambridge, Mass., 1998, xviii+ 322 pp, ISBN 0-262-19398-1,(hardback, 31.95).-.” Robotica 17.2 (1999): 229-235.

[3] Hessel, Matteo, et al. ”Rainbow: Combining Improvements in Deep Reinforcement Learning.” arXiv preprint arXiv: 1710.02298 (2017).

[4] Feurer, Matthias, et al. ”Efficient and robust automated machine learning.” Advances in Neural Information Processing Systems. 2015.

[5] MLA Brochu, Eric, Vlad M. Cora, and Nando De Freitas. ”A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning.” arXiv preprint arXiv:1012.2599 (2010).

[6] Swersky, Kevin, Jasper Snoek, and Ryan P. Adams. ”Multi-task bayesian optimization.” Advances in neural information processing systems. 2013.

[7] C. Thornton, F. Hutter, H. Hoos, and K. Leyton-Brown. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In Proc. of KDD13, pages 847855, 2013.

[8] Vanschoren, Joaquin, et al. ”OpenML: networked science in machine learning.” ACM SIGKDD Explorations Newsletter 15.2 (2014): 49-60.