Reinforcement Learning from Data Meta-features to Automate Machine Learning Pipeline

Van-Duc Le[1]

***Abstract*— Machine Learning (ML) pipeline is a sequence of steps to solve a ML problem from data pre-processing, features transformation to model selection, hyper-parameters optimizing, and get the desired outcomes (e.g. the accuracy). Recently, the tasks of model selection and hyper-parameter tuning have been automated by using some methods such as Auto Machine Learning (AutoML) or Neural Architecture Search (NAS). Nevertheless, the mentioned methods mostly focus on finding a good model for a ML dataset and assume that the data was already processed before entering the model. In reality, the pre-processing and features transformation steps are important tasks to make a good ML model. Clearly, these steps depend on the characteristics of the input data such as the data meta-features (e.g. number of features, types of features, the features distribution, to name a few). In this project, I will try to automate the whole ML pipeline by learning from data meta-features using Reinforcement Learning (RL). Specifically, I will use a Policy Gradient method with the policy is represented by a Deep Neural Network (DNN) such as a 2-layer Fully Connected NN or a Recurrent Neural Network (e.g. LSTM). I train the policy gradient by the REINFORCE algorithm with nearly 100 real-world ML datasets and test with other 15 datasets. Although the results are quite limited because of the not-yet-optimized policy models, they show that RL can be used to automate the whole ML pipeline for real-life ML problems by learning the data meta-features. This result could be applied to planning problems that try to forecast a new plan based on the historical business data. I also point out the reasons for current poor results and some future improvements in the Discussion section.**

I. INTRODUCTION

A Machine Learning (ML) problem is using some types of ML algorithms such as linear regression, logistic regression, Naive Bayes, random forest, neural networks,... to train from existing data (called training set) and get predictions with new unseen data (called test set). The whole process to solve a ML problem includes some steps, that is called a ML pipeline. For example, a typical ML pipeline includes data pre-processing (e.g. data re-scaling, data balancing, or one-hot encoding), features transformation (e.g. Principal Component Analysis), model selection (e.g. choose an appropriate ML algorithms) and model parameters optimizing to reach to a good outcome (e.g. high accuracy on the test set).

Recently, there is a growing demand for ML systems that can be used efficiently by non-experts in ML. In the basic ML algorithms, some of the famous open-source solutions are Auto-WEKA [1] and Auto-Sklearn [2]. In the Deep Learning (DL) area, a sub-set of ML field, Neural Architecture Search (NAS) [4] based methods are the most successful solutions. At their core, every automatic ML or DL method needs to solve the problem of deciding which ML (DL) algorithm (or model) to use on a given dataset. They are often formalized as Combined Algorithm Selection and Hyper-parameter optimization (CASH) problems that try to select one ML algorithm or model and then find the best combinations of its hyper-parameters by using Bayesian optimization [3]. While these methods are quite successful in finding a good ML model for a testing data, they often consider the input data is perfect or leave the users the task of preparing a clean input data for the model selection. However, in real-world ML problems, most of the time for a ML process is doing the data preparation such as data pre-processing and features transformation. As a result, the whole pipeline of a ML problem is not automated.

In fact, all the steps of a ML pipeline will depend on the characteristics of its input data. In Auto-Sklearn paper [2], the authors have proposed to use the meta-learning to learn about the performance of ML algorithms across datasets. They have defined a set of meta-features, e.g. the properties of the dataset that could be computed efficiently and that help to determine which algorithm to use on a new dataset. In this project, I will use these defined meta-features to represent for the data core features and to learn the whole pipeline process, e.g. select the appropriate steps of data pre-processing, features transformation, and model selection. The data meta-features information and the steps for a ML pipeline can be seen from figure 1.

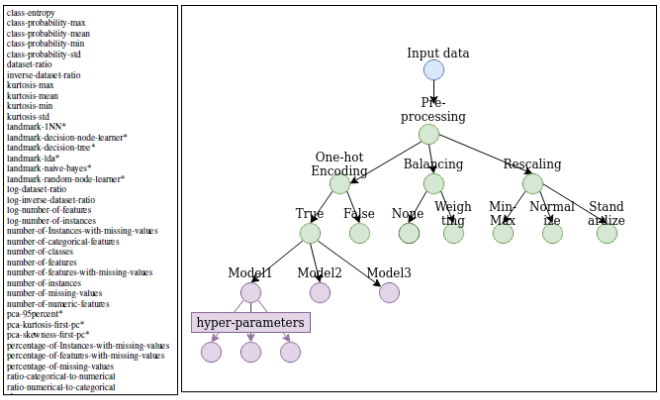

Fig. 1. The data meta-features information and a ML pipeline structure

To do this, I will use Reinforcement Learning (RL), an efficient method to learn how to act at each step of a ML pipeline. Inspired from NAS paper [4], a Policy Gradient

[1]Van-Duc Le is with Department of Electrical and Computer Engineering, College of Engineering, Seoul National University, 08826, Republic of Korea levanduc@snu.ac.kr

method is used to find the optimal policy and the action of each pipeline step. Policy Gradient method is a type of RL techniques that target on optimizing parameterized policies with respect to the expected return. Specially, to find the optimal steps of a ML pipeline, we ask our RL agent to maximize its expected reward R, represented by cost function $J(\theta)$:

$$J(\theta) = E_{P(a1:T;\theta)}[R]$$

R is the accuracy when we run our predicted ML pipeline on the test set. R is used as a reward signal to train the RL agent. $a1:T$ are the list of actions at each step to make a ML pipeline. $P(a1:T;\theta)$ is the policy network with parameters θ that outputs a probability to sample a ML pipeline. The entire architecture that uses a RL agent to learn from data meta-features and predict a ML pipeline is showed in figure 2.

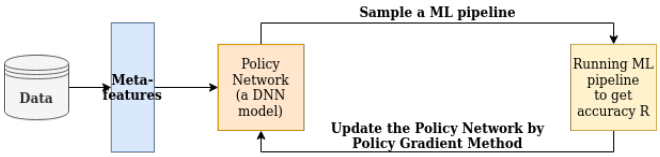

Fig. 2. An architecture of a RL agent to learn from data meta-features to get a ML pipeline

Since the reward signal R is non-differentiable, we need to use a policy gradient method to iteratively update parameters θ. In this paper, I also use the REINFORCE rule [5] as in the NAS paper [4]:

$$\nabla\theta J(\theta) = \sum_{t=1}^{T} E_{P(a1:T;\theta)}[\nabla_\theta log P(a_t|a_{(t-1):1};\theta)R]$$

An empirical approximate of the above computation is:

$$\frac{1}{m}\sum_{k=1}^{m}\sum_{t=1}^{T} \nabla_\theta log P(a_t|a_{(t-1):1};\theta)R_k$$

Where m is the batch size and T is the number of steps that our RL agent has to predict to sample a full ML pipeline. The accuracy when running a k^th sampled ML pipeline on the test set is R_k.

In the next section, I will introduce in detail the Problem definition as a Markov Decision Process (MDP). Then, the Policy Networks will be presented with 2 types of Deep Neural Network (DNN) are defined. Lastly is my Experiments with training on nearly 100 real datasets and testing with other 15 datasets. I finished my report with the Conclusions and Discussions.

II. PROBLEM DEFINITION

A. Markov Decision Process (MDP)

For this problem, a MDP is defined as below:

$A = \{1, ...N\}$ = set of actions at each step of a ML pipeline. For example, at the step $Pre-processing$, there are some actions such as $none, minmax, normalize$. You can check table III in the Appendix for more information.

$S = \{1, ...N\}$ = set of states of a ML problem after executing each step in a pipeline. S_1 = initial state is the meta-features of the data. After executing each step, the data meta-features have changed because the data has been processed by some algorithms (e.g. normalize, PCA). Then we will have a new state after each step of a ML pipeline.

R = only 1 reward after executing the whole ML pipeline. Unlike in a typical RL problem such as in an Atari game where we have the reward at each state, in this problem we need to go through all the states and get an unique reward at the last step.

$P(a) = [P_{ij}(a)]$ = probability to move from state i to state j given an action a at step i.

B. Reinforcement learning Environment

In this part, I will explain how an environment is constructed for this problem of automated ML pipeline using RL algorithms. As you know, in a common RL problem such as learning to control a game, the RL environment includes the game's screens, game's controls (e.g. go left, go right, jump, fire,...). There are some libraries existed that model the environment for our easy in RL problem development such as OpenAI Gym [6]. Our RL environment for this problem is very different than a game's environment but I have tried to create an Environment class that gives convenience for developers to learn a ML pipeline from a dataset.

Our Environment class has 3 main functions that are similar to a Gym environment as illustrated in Table I. An Input function is to register which datasets we will use for our ML problem (similar to the environment's name in Gym). A Data State function returns the initial state of a dataset (similar to the $reset$ function in Gym). And a Run Pipeline function similar to the $step$ function in Gym that executes the ML pipeline on the data and return new state and the reward.

TABLE I
COMPARE OUR ENVIRONMENT CLASS AND OPENAI GYM ENVIRONMENT

Gym environment	Our environment class
env = gym.make('CartPole-v0')	env = Env('dataset')
state = env.reset()	state = env.data_state()
state, reward, _ = env.step(action)	state, reward, _ = env.run_actions(action)

Following are the details of our Environment class.

The Input function accepts a $dataset\_path$ parameter that contains the list of dataset files in CSV format. A $load\_data$ function will load the specific dataset to our Environment with 3 information: the data features (X), the label (y) and the features type ($feat\_type$) of the data (e.g. Numerical or Categorical). After loading completely, the data will be splitted into training data and testing data by the ratio $80:20$.

The Data State function tries to return the initial state of each dataset as meta-features. It uses the Auto-Sklearn [2]

meta-features computation function. The output is a vector of 29 meta-features of float type. One important problem with meta-features is that we need to normalize the meta-features' values to the same scale to be used as the input of the Policy Networks. Inspired from the de facto standard in the ImageNet pre-trained models that normalize the input image by the ImageNet datasets' mean and standard, we also compute the min, max from all of the ML datasets and use them to $0-1$ normalize the meta-features' values. Please check table IV in the Appendix for more information on the data meta-features.

The Run Pipeline function gets the input is a ML pipeline that is sampled from the output of Policy Networks. This function uses a modified of the Auto-Sklearn API to execute the sampled ML pipeline along with the hyper-parameters optimization (based on Bayesian optimization algorithm [3]) on the dataset to get the final testing accuracy. This accuracy is then returned as the reward for the RL process. Because the whole ML pipeline has already executed, we have finished 1 episode of the RL training process.

III. POLICY NETWORKS

Inspired from NAS paper [4], I also use REINFORCE algorithm from Williams (1992) [5] to train our policy networks. This section will firstly introduce the cost function used to optimize policy networks and then presents the policy networks as Deep Neural Networks (DNN) models.

A. Cost function

The cost function tries to maximize the reward as the accuracy after running the whole ML pipeline. A ML pipeline includes the sampled actions that are the output of our Policy Networks. Our Policy Networks will maximize the probabilities to output actions that construct a ML pipeline with highest accuracy on the test set. Therefore, the cost function will maximize the multiplication between log of the action value function, that is the predicted value action, with the reward. Moreover, I add a $L2$ regularization loss to the cost function to make it less suffer with overfit problem. Following are the equations to denote the cost function used in this problem.

$$action\_pred\_value = policy\_value * masked\_action$$
$$action\_loss = -log(action\_pred\_value) * r$$
$$l2\_loss = l2\_loss(trainable\_variables)$$
$$cost = loss + 0.002 * l2\_loss$$

B. Policy Networks as DNN models

This part will show my attempt to represent the policy networks as DNN models. For this project, I have tried with 2 options. The first is a Fully Connected (FC) model which is quite common in representing model in RL problems. The second is taking from idea in NAS paper [4] that uses a Recurrent Neural Network (RNN) to represent the controller. I will use a Long Short-Term Memory (LSTM) model to encode the policy networks that output ML pipeline steps. Although 2 models are quite different, I found that they give similar results on my experimental datasets.

Firstly, I try with a 2-layer FC model with each layer includes 32 hidden units. After the data state (meta-features) running through this 2-layer network, it will output the probability for each set of actions at each step in the ML pipeline. For example, in the re-scaling step, we will have 6 types of actions (including no-rescaling action) to select and will output the probability to choose one of 6 actions. Similarly, the feature-transformation step needs to output 1 of 13 actions and the classifier (model selection) step needs to output 1 in 15 actions. Figure 3 illustrates this 2-layer FC model and its output.

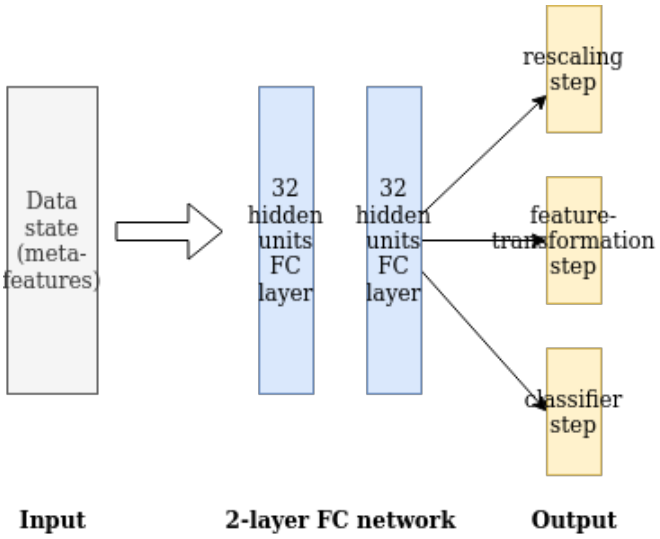

Fig. 3. A 2-layer FC model that outputs actions at each ML pipeline step from a data state

Second is a LSTM model that compute the output from initial state as data meta-features. As you know, a LSTM model will have an input, a hidden state and an output. Usually, the initial state of a LSTM cell will be initialized randomly at the first time and then it will be updated with the hidden state of previous time step. However, in this problem, to learn from the data meta-features, I will set the initial state of the LSTM cell to the encoded meta-features (by a neural network layer) and then follow the normal LSTM learning. I also apply a sequence-to-sequence architecture for LSTM model to generate the action of each ML pipeline step consequently. Figure 4 shows my proposed LSTM model with the hidden units are 64.

IV. EXPERIMENTS

In this section, I will present the datasets that I use for training and testing as well as the evaluation of my experimental results. I have published my source-code for training and testing at the address: https://github.com/vanduc103/RL2020.

A. Data sets

To do experiments with real-world datasets, I collect the training datasets from OpenML website [7], a place for many real ML datasets uploaded by users. There are more

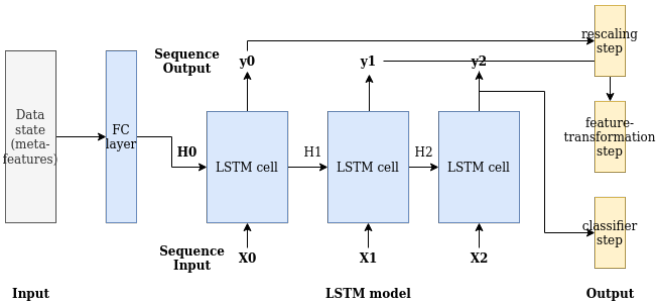

Fig. 4. A seq-to-seq LSTM model that get data state as initial state and output actions sequence for the ML pipeline

than 3000 datasets in OpenML (as the time of June 2020) that target both classification and regression tasks. In this project, I only focus on large-enough datasets that have at least 1000 rows of data. And I also limit the experiments with classification datasets. In addition, for simplicity, I will not use datasets that have missing values. I believe that this assumption will not break the generality of my approach because we can easily add 1 step of fixing missing records in the ML pipeline. However, I do my experiments with datasets have both numeric or categorical data types.

In total, I have collected nearly 100 ML datasets for training in a variety of real-world problems such as in finance, health, science, to name a few. For testing datasets, I use the same testing datasets as in Auto-Sklearn paper [2] that includes 15 datasets such as amazon, car, cifar, mnist, wine quality white, and so on. All the datasets have the CSV format with 1 column as the label and remaining columns are data features.

B. Training

I execute the experimental training with 2 Policy Networks models as mentioned above. Each training process uses Adam optimizer with the learning rate is fixed at the start and then gradually decreases when the training runs. I tried with some initial learning rate such as 0.1 and 0.01. All experiments run 100 epochs in total, each epoch trains with the training datasets with batch size is 2. I save the network checkpoint whenever the loss is decreasing. The action at each pipeline step is epsilon-sampled with the epsilon is scaled down to a final epsilon during the training.

Each time, when the policy networks output a ML pipeline (sampled a list of actions for each step of the ML pipeline), I invoke the Auto-Sklearn [2] hyper-parameters optimization API with the total assigned running time is 30 seconds. It means the Bayesian optimizer has a maximum 30 seconds to find the most suitable set of hyper-parameters for the ML dataset. With the batch size is 2 and nearly 100 training datasets, the training time for an epoch is 25 minutes and for total 100 epochs is approximate 42 hours.

C. Evaluation

For the evaluation part, the input to Environment class is a list of testing datasets. I restore the saved checkpoint and run learned policy networks on testing datasets to get a predicted ML pipeline for each dataset. Then each ML pipeline is run to get the accuracy. The final output of the evaluation part is the average of these accuracy.

To compare our approach, I run the original Auto-Sklearn API with the testing datasets. The experiments results are showed in Table II.

TABLE II
AVERAGE ACCURACY ON TESTING DATASETS OF MY APPROACH AND AUTO-SKLEARN API

Our approach	Auto-Sklearn API
0.26	0.72

From the results above, our approach gives not good results compare to the original Auto-Sklearn API. I will discuss these results further in the Discussions section.

V. CONCLUSIONS AND DISCUSSIONS

A. Conclusions

To conclude, this project has proved that we can use a Reinforcement Learning method to learn from data meta-features to automate the whole ML pipeline. In addition, to make it easier for developers to apply RL in this type of problem, I have created an Environment class that has similar functions to an OpenAI Gym environment which is a common lib in RL research.

B. Discussions

As showed in the Experiments Evaluation section, my results are not very good compared to existing Auto-Sklearn library. I think following are some reasons to this limitation:

- We only use the data meta-features (or the environment's state) once, at the start of an episode to learn all the steps of the pipeline. The reason for this approach is we only get the reward after executing the whole pipeline on the test set. Instead, the data meta-features will change after each step of the pipeline because we have applied some transformation to the data. In the future, I will try to find how to define the reward after each step of the pipeline to learn from the state of each step.
- The exploration/exploitation problem. Maybe the epsilon greedy algorithm for actions sampling is not sufficient for this task. And the policy optimization only gets local optimum. I think I need to add more constraints to get a better exploration, e.g. add an entropy regularization to encourage the policy networks to reach to global optimal point.
- I do not think the selection of Policy Gradient algorithms (e.g. REINFORCE algorithm) will much affect to the performance of this system. I think the policy network representation (the DNN model) and the data input (data meta-features) have the most impact to the final performance. Also, I need to find more constraints to guide the policy learning, such as a better exploration/exploitation trade-off.

- One thing I could try in the future is to apply an initial pre-training approach by pre-training the policy networks with the historical running of training datasets from Auto-Sklearn API.

APPENDIX

Table III presents the list of possible actions at each step of a ML pipeline that are used in my Environment class.

Table IV shows the meta-features definition of a dataset (29 meta-features). I also have computed the min, max and mean value among all training and testing datasets.

TABLE III
LIST OF ACTIONS IN EACH STEP OF A ML PIPELINE

Step	List of actions
Pre-processing	none minmax normalize quantile_transformer robust_scaler standardize
Features Transformation	no_preprocessing extra_trees_preproc_for_classification fast_ica feature_agglomeration kernel_pca kitchen_sinks liblinear_svc_preprocessor nystroem_sampler pca polynomial random_trees_embedding select_percentile_classification select_rates
Classifier	adaboost bernoulli_nb decision_tree extra_trees gaussian_nb gradient_boosting k_nearest_neighbors lda liblinear_svc libsvm_svc multinomial_nb passive_aggressive qda random_forest sgd

REFERENCES

[1] C. Thornton, F. Hutter, H. Hoos, and K. Leyton-Brown. Auto-WEKA: combined selection and hyperpa-rameter optimization of classification algorithms. In Proc. of KDD'13, pages 847–855, 2013.

[2] Matthias Feurer et al., Efficient and Robust Automated Machine Learning, Twenty-ninth Conference on Neural Information Processing Systems, 2015.

[3] M. Feurer, J. Springenberg, and F. Hutter. Initializing Bayesian hyper-parameter optimization via meta-learning. In Proc. of AAAI'15, pages 1128–1135, 2015.

[4] Zoph, Barret; Le, Quoc V. , Neural Architecture Search with Reinforcement Learning, arXiv:1611.01578.

[5] Ronald J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, In Machine Learning, 1992.

[6] OpenAI Gym, website: https://gym.openai.com/docs/.

[7] OpenML.org, website: https://www.openml.org/.

TABLE IV

THE META-FEATURES DEFINITION OF A DATASET

Meta-feature	**min**	**max**	**mean**
ClassEntropy	0.03853718088787195	5.634725237579686	1.2833706866488395
SymbolsSum	0.0	287	7.169642857142857
SymbolsSTD	0	3.0952509611333126	0.11628641612175684
SymbolsMean	0	6.666666666666667	0.46413053712160857
SymbolsMax	0	12	0.7053571428571429
SymbolsMin	0	4	0.32142857142857145
ClassProbabilitySTD	0.0	0.49588477366255146	0.1477539836312912
ClassProbabilityMean	0.02	0.5	0.40916001657073103
ClassProbabilityMax	0.025714285714285714	0.9958847736625515	0.5713088860091884
ClassProbabilityMin	0.00013793103448275863	0.5	0.26446024662686446
InverseDatasetRatio	0.105	4833.333333333333	549.0306747222126
DatasetRatio	0.00020689655172413793	9.523809523809524	0.6832655137201668
RatioNominalToNumerical	0.0	3.5	0.07844387755102042
RatioNumericalToNominal	0.0	7.0	0.10442830978545266
NumberOfCategoricalFeatures	0	60	1.9464285714285714
NumberOfNumericFeatures	0	10935	1091.875
NumberOfMissingValues	0.0	0.0	0.0
NumberOfFeaturesWithMissingValues	0.0	0.0	0.0
NumberOfInstancesWithMissingValues	0.0	0.0	0.0
NumberOfFeatures	1.0	10935.0	1093.8214285714287
NumberOfClasses	2.0	50.0	4.035714285714286
NumberOfInstances	700.0	70000.0	7262.517857142857
LogInverseDatasetRatio	-2.2537949288246137	8.483291639740557	4.4724832256630895
LogDatasetRatio	-8.483291639740557	2.2537949288246137	-4.4724832256630895
PercentageOfMissingValues	0.0	0.0	0.0
PercentageOfFeaturesWithMissingValues	0.0	0.0	0.0
PercentageOfInstancesWithMissingValues	0.0	0.0	0.0
LogNumberOfFeatures	0.0	9.299723933110869	3.6941254838551267
LogNumberOfInstances	6.551080335043404	11.156250521031495	8.16660870951822