Meta-learning through Deep Reinforcement Learning

Ruiqi Zhong
Columbia University
rz2383@columbia.edu

Haoyan Min
Columbia University
hm2689@columbia.edu

Yilan Ji
Columbia University
yj2425@columbia.edu

Abstract

In response to the growing demand for data analysts and machine learning experts, in this project we aim to train a neural network that can automatically generate machine learning pipeline consisting of a small set of basic sklearn primitives. The objective is to train a model/algorithm that can use the minimum amount of time to find the model combined with its hyper-parameter that can minimize the cross validation loss. We employ reinforcement learning to let the model learn what is the best strategy to generate the pipeline. By the end of this semester, we achieved observable result that can improve our neural network's first attempted sklearn primitive model on the data set.

1. Previous Works, Introduction and Problem Formulation

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?

Here we define the problem more formally. 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 such a model that approximately minimizes the cross validation loss in the shortest time.

Previous works have employed Bayesian optimization techniques (with or without labeled meta-data) and achieved respectable performance[4][5][6]. Bayesian optimization views this problem as optimizing a black box objective function, which is very expensive to query. It 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 takes into account both exploration (sampling from areas of high uncertainty) and exploitation (sampling areas likely to offer improvement over the current best observation).).

However, Bayesian optimization does not make full use of the information returned by the black box. Upon each query, not only is the classifier cross validation loss available, but also the training set loss/accuracy and the training time. Imagine that you are a data analysts and are performing a logistic regression on a data set. The program feed back the results telling you that the cross validation accuracy is 0.8. Will you increase or decrease the regularization? You are probably not sure. But if the program also tell you that the training set accuracy is 0.999, then you probably know that the next promising to try is to increase the regularization, as the machine learning algorithm is overfitting the training set. In this project, beyond the simple bayesian assumption of the objective function itself, we attempt to discover more known and unknown assumptions and relations between different model performance statistics and meta-features.

Recently we also see a large wave of work using reinforcement learning and metalearning to find a good network architecture and have achieved respectable success[10][11][12] in the field of image classification. Most of them try to fit a single or a few very large data sets. On the contrary, however, in this project we are working on a wide range of data sets that have drastically different properties. Though one can view a distribution of different datasets is essentially one data set, here we focus more on capturing the relation between performance of different primitive models rather than memorizing and fitting one particular data set; also, the resulting best model is INTERPRETABLE: the best model we find is just one of the best sklearn primitives, instead of a complicated neural network architecture that even the best researchers cannot understand why the found architecture works well.

2. Reinforcement Learning Formulation and Model Architecture

2.1. Motivation

We see many technical breakthroughs in recent years achieved by reinforcement learning. In a nutshell, the model will explore in the simulated environment and adapt itself to maximize the reward in the long run. On the other hand, the progress of combined algorithm and hyper-parameter search can be perfectly simulated, and this motivates us to use reinforcement learning to train our network.

2.2. Reinforcement Formulation

Suppose that we want to maximize the cv test accuracy of the best model within T steps. Let the data set be D (including the feature values and the labels), r (reward) cv test accuracy increase over the best past attempted model; o the meta-features of D (e.g. number of dimensions, proportion of missing entries, skewness, sparseness, etc) and all the past attempted models and their performance; s the current state of the searching algorithm; a_t a probability distribution of the attempted model at time t: for example, one ϵ greedy exploration is a probability distribution over the choices, with ϵ probability of choosing uniformly and $1-\epsilon$ probability of choosing the best model. With r, s, a, o being defined, we can perform $Q-learning$.

2.3. Model Architecture - Q network

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. We use long short term memory to model the algorithm search procedure. During evaluation time, the input to the first hidden unit h_0 is the meta-features of the data-set, and it outputs a vector s_0 representing the state of the search algorithm given the observations o, meta-features. s_0 is concatenated with the action input a_0 to form (s, a) layer, which is followed by dense layers and output the predicted value $Q(s, a)$. During the evaluation phase, $g \sim uniform(0, 1), a_0 = \pi(s_0) = argmax_a Q(s, a)$ if $g > \epsilon$, otherwise randomly choose a model in the prior action distribution $P(a)$. A model choice m_0 is sampled from the model distribution a_0 and then its performance statistics o_0 is calculated. At the next time step, the meta-features o, past model choice m_0 and the performance statistics o_0 is input to the next hidden state h_1. Then this process is repeated over and over again. See fig.1 for a vivid demonstration.

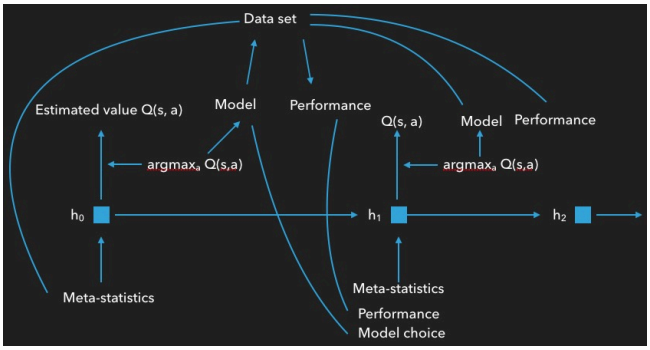

Figure 1. Q learning

2.4. Model Architecture - Policy Network

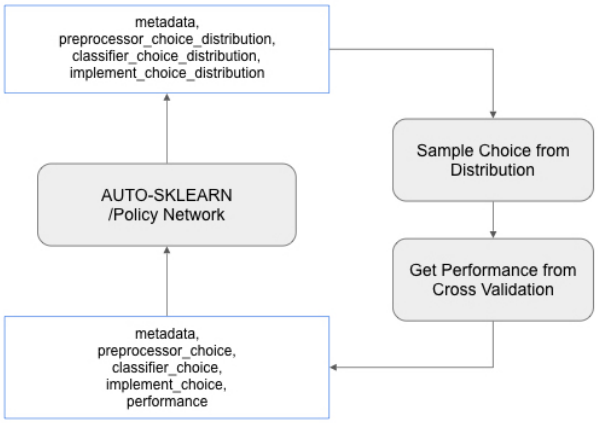

Figure 2. Mimicry the behavior of Auto-sklearn

It is interesting to test whether the model can start from zero domain knowledge and learn to search the model in-

telligently from scratch. However, it is also worth using existing algorithms as starting points and use reinforcement learning to improve it. Here we train a policy network $\pi(s)$ that uses the behavior of auto-sklearn as ground truth. Similar to the Q network, it is modeled by a long short term memory network, takes meta-features o, the last model choice m_{t-1} and its performance o_{t-1} to the hidden state of h_t. Different fromt the Q network, instead of outputting a vector representing the state s_t, it directly outputs a probability distribution of model choices a_t. During training, the loss is defined by a psuedo log-likelihood function $L = \log(P(a_t, m_t^*))$, where m_t^* is the ground truth given by auto-sklearn. During evaluation phase, the model chosen m_t is sampled from a_t, which is a probability distribution. See fig.2 for a vivid demonstration.

3. Comparing and Evaluating CASH Algorithms

Given two *CASH* algorithms $T^{(1)}, T^{(2)} \in \mathbb{T}$ a family of algorithms that output the sequence of machine learning models $\mathbb{A}$ and hyper-parameters λ, we compare them in the following ways: on the set of data set $\mathbb{D}$ and a fixed positive integer t, after trying t models, what is the best performance of the (minimum in terms of cross validation loss) model. Since the LSTM 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.

4. Methods, Technical Details and Implementations

Our github is: https://github.com/PB12203006/Meta-Learning-with-Deep-Reinforcement-Learning .

4.1. Data set

In this problem we only consider binary classification problem and we evaluate a model by cross validation accuracy. Since there are not many real-world data set to learn from, we use simulated data sets. Each of the data set is of small or medium size, containing 100 to 1,000 data points and 5 to 1,000 dimensions. To simulate real world situations, each data point incorporates randomness, each basis has a certain probability of undergoing a certain transform (e.g. inverse, exponential, log, square, etc). We also add noise to the observations and perform linear transformation to inflate its dimension so that PCA in many cases would be useful. In each data set D, a certain feature entry has p_D probability of being missing/nan, so that imputation becomes useful. The detail of the distribution of the data set can be seen in Generate_Data_Set.py in the github.

4.2. Encoding Scheme

Each model is encoded to a 17 dimensional vector, including imputation strategy, classifier choice, pre processor choice and their respective hyper-parameters (see detail in Appendix A). Each hyper parameter is encoded as real value, and each categorical choice is encoded as one-hot. Meta-features are encoded as a 29(round2,3)/38(round1, 9 more in red) dimension vector and the comprehensive list can be seen in Appendix B. Model performance o_t is a four dimensional vector corresponding to test/train loss/accuracy (Appendix D). At time 0, the previous model choice and model performance are padded with 0. The actions, which are probability distributions of model choices, are also encoded in 17 dimension. (see detail in Appendix C)

4.3. Network architecture

Here we use a small size of Long Short Term Memory neural network to model our algorithm: only 8 hidden states and layers with 5 dense units in order to prevent over-fitting issues (round 1/2). After the report meeting we agreed to enlarge the dimension of hidden states and reduce the time steps to see whether there are any improvement in the performance. The newest result will be shown after the discussion section.

4.4. Training of Reinforcement Learning

While drawing an action a from he action prior $P(a)$, the model choice, hyper-parameter, imputation strategy and prepossessing steps are independent. The detail of the action prior can be seen in generate_random.py on our github. Since $argmax_a Q(s, a)$ itself is a neural network and finding its global maximum is almost impossible, 100 points are sampled from the prior randomly, fed into the network, and the action chosen is defined by one of the 100 actions that would maximize the Q.

To speed the entire process up, instead of doing 3-cross fold validation, we are only performing a 0.75-0.25 train test split and training-testing for once. The performance becomes more unstable; however, this amount of random noise is not likely to influence much the evaluation process. For each iteration, chosen models are run 3 steps on 4096 data sets (generated on the fly). After we generate the data in the evaluation phase, we train the LSTM on the CPU; since each sequence is short and there are not tons of data point, for each iteration we train the network 2 epochs with the data just generated in the evaluation iteration.

4.5. Implementation

All code is in Python 3.4. We also used Keras 2.0.9 to implement the neural network. The evaluation stage is implemented as multi-process using 32 CPUs and each stage of evaluation (attempting 3 models on 4096 data sets) take

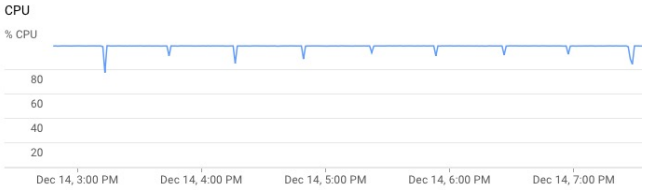

Figure 3. CPU usage

about 40 minutes. Fig. 3 demonstrates the CPU usage of our reinforcement learning algorithm: the plateau (using 100% CPUs) corresponds to the evaluation phase while the valleys correspond to the training phase. As demonstrated in the graph, evaluation phase takes the most time, since for each of the 3 * 4096 data point it is training one classifier on simulated data set, while each training process it is only training a LSTM 2 epochs on 4096 sequences of length 3. In this project, we did not implement the asynchronous reinforcement learning[13] algorithm for simplicity and robustness.

5. Experiments and Results

We conducted two rounds of reinforcement learning. During the first round, during the evaluation phase, we attempt 10 models on each of the 100 datasets, 3-fold cross validation and the reward r_t is defined by the $max(0, accuracy_t$ - past_best_accuracy). During the second round, we attempt 3 models on each of the 4096 data sets, 0.75-0.25 train-test split and the reward r_t is defined by the $accuracy_t$ - past_best_accuracy). We also trained a policy network; however, due to time constraint, we do not have time to incorporate it into our reinforcement learning framework.

5.1. Round 1

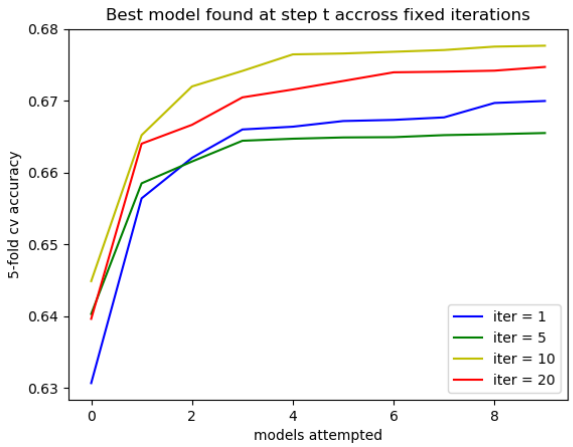

Figure 4. Round1 policy performance, fixed iteration

Fig4. is the graph shown during the presentation. For each of the curve, it represents the policy performance at a given iteration, the x axis represents the number of models attempted, y axis represents the test accuracy of the best model attempted model until time x. As we might observe, though the policy performance is not monotonously increasing, they are all significantly and consistently better than the random starting point. However, this graph turns out to be deceptive. As we investigate it later, the improvement seems only to be noise (see Fig 5): For each of the curve,

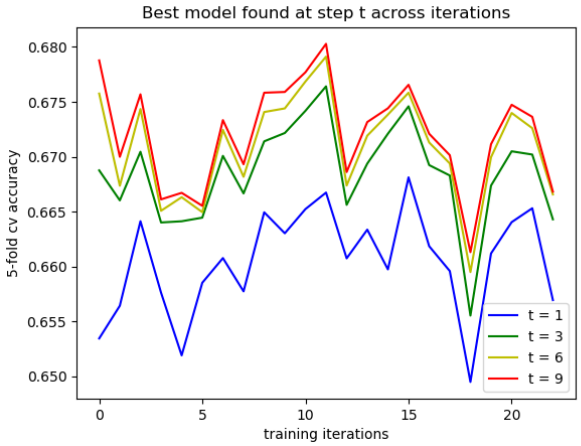

Figure 5. Round1 policy performance, fixed steps

it represents the policy performance after attempting a fixed number of models, and the x axis represents the change of the policy's performance on its i^{th} attempted model against iterations. As we can see in the graph, the changes are not improvements; rather, they are pure noises.
We performed error analysis and considered that the policy evaluation is too noisy. Therefore, we increased the number of data sets being evaluated (increasing from 100 to 4096). In compensate for this change, we only attempt 3 models for each data set and we only perform a train-test 0.75, 0.25 split rather than a 3-fold cross validation. Below are the results for the second round.

5.2. Round 2

We plot the same graphs as in round1. Again, the first graph (Fig 6) seems nice, but still turns out to be deceptive. Now we look at the other graph Again, the changes are pure noises (Fig 7).

5.3. Policy Network

We also trained a policy network that mimics the performance of auto-sklearn. Below is the evaluation for our policy network: The black line corresponds to the randomized search baseline. In predicting the model choice of auto-sklearn, it is hard to design a loss function: since each com-

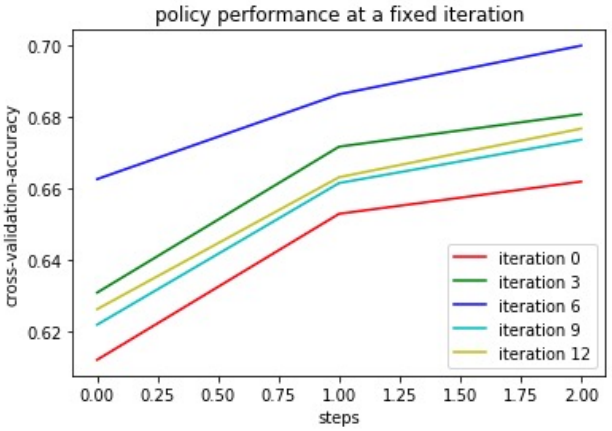

Figure 6. Round2 policy performance, fixed iteration

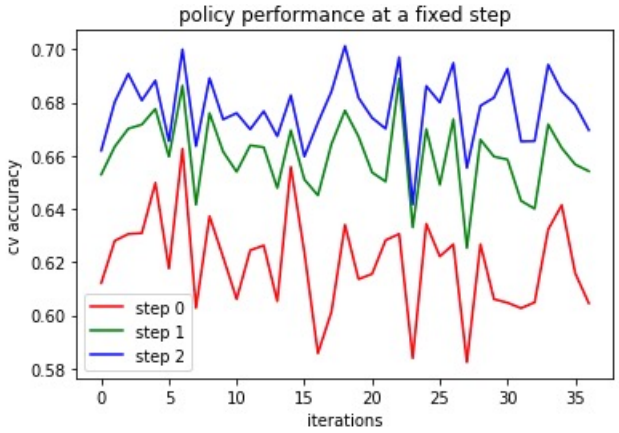

Figure 7. Round2 policy performance, fixed steps

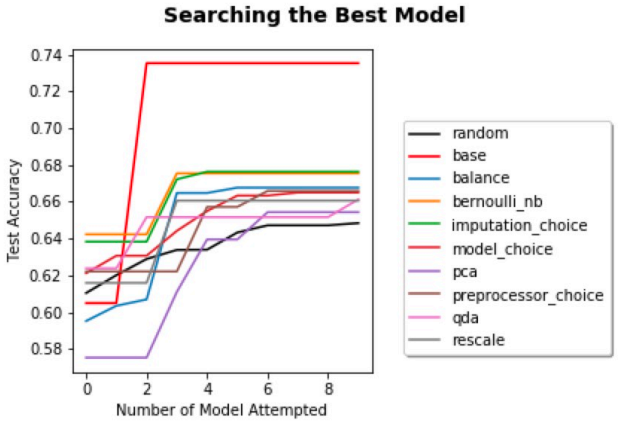

Figure 8. Performance of policy network

ponent (e.g. pre-processor, imputation, hyper-parameter) has a loss function, how to weight them. Here we explored different weight (different curves in Fig 8). We increase the loss weight for each component compared to the "base" and the resulting performance in the graph. As we can see, a balanced loss function would result in the best performance. As we can see in the graph, it is performing better than any of the policies we trained in round1 and round2. Therefore, it is worth exploring whether incorporating it into our reinforcement learning framework will improve the performance.

6. Discussion

To summarize, we built a novel reinforcement learning framework different from previous works (as pointed out in the introduction section). However, the experiments show that the reinforcement learning does not improve the performance. Since it is a large framework and most parts are newly invented by us, many of them can go wrong. Here is the error analysis and possible ways to improve it.

6.1. Deeper and larger network

In this project we tried a small shallow LSTM network to prevent over-fit. However, given that there are only 16 hidden units for LSTM and 10 units for each dense layer and our encoding scheme is simple, it might be the fact that the network is not expressive enough to learn the better policy.

6.2. Design of reward function

As is true for most reinforcement learning problem, it is hard to design a good reward and set a reasonable γ. In our project, the reward defined as a function of the current test accuracy and the best past test accuracy. However, since each data set has different learning difficulties (for example, some data sets' gold standard only has 0.7 accuracy; the rest of the error is caused by noise), the expected accuracy is different and thus giving out different reward. It might worth exploring more sophisticated reward function, for example as in [9]. In this paper, they used reinforcement learning to train a better beam search algorithm; in the beam search stage for the encoder decoder stage for neural machine translation, the reward is defined by "how well the policy network can differentiate the top and the second top choice" rather than a function of the final BLEU score. In fact, they did try using a function of predicted BLEU score as a reward for the reinforcement learning; however, it did not lead to significant improvements. It turns out that the new sophisticated reward function did lead to a significant improvement. More details can be seen in the paper.

6.3. Retrospectively

In this project, we spent the first 2 weeks figuring out the architecture and the reinforcement learning problem formulation, and the next 2 weeks coding things up and running experiments (and spending all the Google cloud credits). Nevertheless, we now realize that the full-fleshed architecture is a novel large framework. We are inventing many

parts from scratch; therefore, when the model fails, we do not know which part it has gone wrong. In round 2 we can conclude that the inability of our algorithm to learn is not because we are not evaluating enough data points and the policy evaluation is too noisy (3 * 4096 in round 2, while the std is less than 0.1, indicating that any changes in test accuracy larger than 0.005 (¿ 3std) is a significant change).The part that is not working might include: 1) design of reward function 2) reinforcement problem formulation 3) data set distribution 4) action prior 5) maximizing strategy in obtaining $argmax_a Q(s, a)$. Retrospectively, **if we had more time**, here is an example alternative proposal. Stage 1: try learning on a one single data set rather than a distribution of data set, and observe whether the designed reward function would gradually lead the policy to find the best model during its first time. Stage 2: limit our model choice to only quadratic discriminant analysis and only include one hyper-parameter, which is the regularization, and observe whether the our policy would learn a better policy than randomized search: e.g. increase the regularization while observing overfits. Stage X: introduce imputation strategy, pre processor and other classifiers one by one. Nevertheless, it is not feasible for this project, since each stage would take time to debug, explore and analyze, and for each stage we will need to implement a different encoding scheme, which is extremely time-consuming. Hence, this alternative proposal might be suitable for a dedicated full time phd student.

7. Newest Result (after the report meeting)

We change the hidden states of LSTM to be 128 (notation consistent with Keras implementation), dense layers each containing 128 units, and only attempt one model on the dataset for each data set after seeing its metafeatures. With this setting, we expect to see an improvement. After

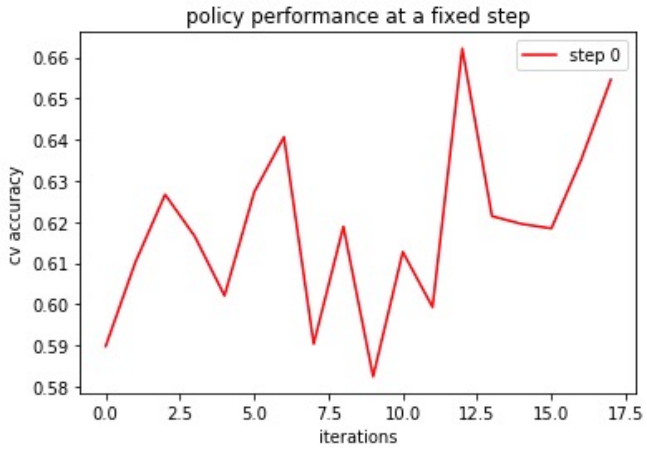

Figure 9. Performance of policy network

we enlarged the dimension of the neural network to a reasonable size, we do see a significant improvement.

8. Conclusion

In this project we designed a novel and interpretable (in terms of the best model found on a data set) architecture and reinforcement learning for the problem of meta-learning. In contrast to more traditional approaches such as Bayesian optimization, it leverages more information (e.g. train loss). At this stage, it successfully learned which model to attempt at the first step and it is worth exploring further potentials of our architecture.

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.

[9] D. He, H. Lu, Y. Xia, T. Qin, L. Wang, and T. Liu. Decoding with value networks for neural machine translation. In 31st Annual Conference on Neural Information Processing Systems (NIPS), 2017.

[10] arXiv:1712.00559

[11] Zoph, Barret and Le, Quoc V. Neural architecture search with reinforcement learning. In ICLR, 2017.

[12] Zoph, Barret, Vasudevan, Vijay, Shlens, Jonathon, and Le, Quoc V. Learning transferable architectures for scalable image recognition. CoRR, abs/1707.07012, 2017.

[13] Mnih, Volodymyr, et al. "Asynchronous methods for deep reinforcement learning." International Conference on Machine Learning. 2016.

Appendix A : MODEL CHOICE

#	description	range	one hot
0	balancing:strategy	{0,1}	none / weighting
1	imputation:strategy	{0,1}	not mean / mean
2	imputation:strategy	{0,1}	not median / median
3	imputation:strategy	{0,1}	not most_frequent/ most_frequent
4	rescaling:__choice__	{0,1}	not none/none
5	rescaling:__choice__	{0,1}	not minmax/minmax
6	rescaling:__choice__	{0,1}	not normalize/normalize
7	rescaling:__choice__	{0,1}	not standardize/ standardize
8	preprocessor:__choice__	{0,1}	no_preprocessing/PCA
9	classifier:__choice__	{0,1}	bernoulli_nb/qda
10	one_hot_encoding:use_minimum_fraction	{0,1}	True/False
11	one_hot_encoding:minimum_fraction	[0.0001, 0.5]	
12	preprocessor:pca:keep_variance	[0.5, 0.9999]	
13	preprocessor:pca:whiten	{0,1}	True/False
14	classifier:bernoulli_nb	[0.01, 100.0]	
15	classifier:bernoulli_nb:fit_prior	{0,1}	True/False
16	classifier:qda:reg_param	[0.0, 1.0]	

Appendix B: METADATA

#	description
0	ClassEntropy
1	SymbolsSum
2	SymbolsSTD
3	SymbolsMean
4	SymbolsMax
5	SymbolsMin
6	ClassProbabilitySTD
7	ClassProbabilityMean
8	ClassProbabilityMax
9	ClassProbabilityMin
10	InverseDatasetRatio
11	DatasetRatio
12	RatioNominalToNumerical
13	RatioNumericalToNominal
14	NumberOfCategoricalFeatures
15	NumberOfNumericFeatures
16	NumberOfMissingValues
17	NumberOfFeaturesWithMissingValues
18	NumberOfInstancesWithMissingValues
19	NumberOfFeatures
20	NumberOfClasses
21	NumberOfInstances
22	LogInverseDatasetRatio
23	LogDatasetRatio
24	PercentageOfMissingValues
25	PercentageOfFeaturesWithMissingValues
26	PercentageOfInstancesWithMissingValues
27	LogNumberOfFeatures
28	LogNumberOfInstances

29	LandmarkRandomNodeLearner
30	SkewnessSTD
31	SkewnessMean
32	SkewnessMax
33	SkewnessMin
34	KurtosisSTD
35	KurtosisMean
36	KurtosisMax
37	KurtosisMin

Appendix C : MODEL ACTION

#	description	range	probability distribution
0	balancing:strategy	[0,1]	none / weighting
1	imputation:strategy	[0,1]	not mean / mean
2	imputation:strategy	[0,1]	not median / median
3	imputation:strategy	[0,1]	not most_frequent/ most_frequent
4	rescaling:__choice__	[0,1]	not none/none
5	rescaling:__choice__	[0,1]	not minmax/minmax
6	rescaling:__choice__	[0,1]	not normalize/normalize
7	rescaling:__choice__	[0,1]	not standardize/ standardize
8	preprocessor:__choice__	[0,1]	no_preprocessing/PCA
9	classifier:__choice__	[0,1]	bernoulli_nb/qda
10	one_hot_encoding:use_minimum_fraction	[0,1]	True/False
11	one_hot_encoding:minimum_fraction	[0.0001, 0.5]	
12	preprocessor:pca:keep_variance	[0.5, 0.9999]	
13	preprocessor:pca:whiten	[0,1]	True/False
14	classifier:bernoulli_nb	[0.01, 100.0]	
15	classifier:bernoulli_nb:fit_prior	[0,1]	True/False
16	classifier:qda:reg_param	[0.0, 1.0]	

Appendix D : PERFORMANCE

#	description
0	train_accuracy_score
1	test_accuracy_score
2	train_log_loss
3	test_log_loss