Accelerating Rule-matching Systems with Learned Rankers

Zhao Lucis Li, ***University of Science and Technology China;***
Chieh-Jan Mike Liang and Wei Bai, ***Microsoft Research;***
Qiming Zheng, ***Shanghai Jiao Tong University;*** **Yongqiang Xiong,** ***Microsoft Research;***
Guangzhong Sun, ***University of Science and Technology China***

https://www.usenix.org/conference/atc19/presentation/li-zhao

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

Accelerating Rule-matching Systems with Learned Rankers

Zhao Lucis Li[⋆‡] Chieh-Jan Mike Liang[‡] Wei Bai[‡] Qiming Zheng[†‡]
Yongqiang Xiong[‡] Guangzhong Sun[⋆]
[⋆]*University of Science and Technology of China* [‡]*Microsoft Research* [†]*Shanghai Jiao Tong University*

Abstract

Infusing machine learning (ML) and deep learning (DL) into modern systems has driven a paradigm shift towards learning-augmented system design. This paper proposes the learned ranker as a system building block, and demonstrates its potential by using rule-matching systems as a concrete scenario. Specifically, checking rules can be time-consuming, especially complex regular expression (regex) conditions. The learned ranker prioritizes rules based on their likelihood of matching a given input. If the matching rule is successfully prioritized as a top candidate, the system effectively achieves early termination. We integrated the learned rule ranker as a component of popular regex matching engines: PCRE, PCRE-JIT, and RE2. Empirical results show that the rule ranker achieves a top-5 classification accuracy at least 96.16%, and reduces the rule-matching system latency by up to 78.81% on a 8-core CPU.

1 Introduction

Machine learning (ML) and deep learning (DL) bring new possibilities to modern system designs [9, 15, 17, 23, 24], which traditionally rely on human-written heuristics. Under the *learning-augmented design*, system logic is implemented with both heuristics and ML/DL to better address performance bottlenecks. Such design has the following advantages. First, compared to heuristics, ML/DL has been shown to excel in learning complex data patterns to enable classification, regression and prediction. Second, while traditional heuristics are designed to be general purpose and computation-efficient, systems such as web services can have a workload that is highly dynamic and scenario-specific. Adapting to workload characteristics can enable highly optimized algorithmic operations and system components.

This work was done when Zhao Lucis Li and Qiming Zheng were interns at Microsoft Research. Chieh-Jan Mike Liang is the corresponding author.

However, formulating ML/DL tasks into system building blocks is non-trivial. Given that ML/DL is stochastic in nature, inference uncertainties should not impact the system correctness. And, inference should not impose a significant resource overhead on the end-to-end system performance. Recently, the industry has had success in using learning-driven space exploration as a system building block, for scenarios such as system configuration tuning [9, 17, 23]. Building on this success, this paper explores the potential and feasibility of another building block for learning-augmented systems – the *learned ranker*.

Particularly, we use rule-matching systems as a concrete scenario for learned rankers. One common task of rule-matching systems is to match the given input to *one* rule in the ruleset as fast as possible, and the performance bottlenecks come from two observations. For rulesets that do not impose a mandatory ordering on rule checking, the naïve practice of sequentially going through rules can result in processing many unnecessary rules. The problem exacerbates when we consider that rules can have non-trivial conditions written in regular expressions (regex). Since regex matching engines typically rely on either deterministic finite automaton (DFA) or non-deterministic finite automaton (NFA), its overhead largely depends on the length and complexity of regex patterns and inputs. In the worst case, the backtracking problem can result in $O(2^n)$ time complexity for an input string of size n [13], rather than the expected $O(n)$.

To reduce the rule matching latency, common optimization techniques include string-matching pre-filters [7, 12], just-in-time compilation [4], and specialized hardware-based regex acceleration [20, 25]. The learned ranker enables a different but complementary technique – after the string-matching pre-filter removes unlikely inputs, it performs per-input rule prioritization for the regex matching engine, based on the likelihood of a given input to match each rule. Conceptually, if the matching rule can be prioritized as one of the top candidates, the rule-matching system effectively achieves early-termination, thus minimizing unnecessary rule checking.

Designing learning-augmented systems with the learned

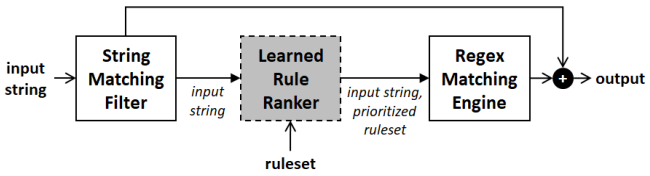


Figure 1: A learning-augmented design of rule-matching systems. Complementing existing acceleration techniques, we introduce a learned rule ranker to dynamically prioritize rules for each input. The goal is to minimize unnecessary rule processing and achieve early-termination.

ranker as a building block should go beyond simply selecting the most accurate ML/DL model – although the ranker helps to reduce the computation load for other system components, it is crucial to balance the trade off between inference accuracy and cost, with respect to the end-to-end system performance. To evaluate the benefits of the learned ranker as a building block for learning-augmented systems, we have integrated it into popular regex matching engines: PCRE [3], PCRE-JIT [4], and RE2 [5]. We benchmark with two publicly available rulesets: ModSecurity CRS [2] and Snort [6]. Empirical results show that the learning-augmented design reduces the rule-matching system latency by as much as 78.81%; in particular, the learned rule ranker can achieve a top-5 ranking accuracy at least 96.16%, and this reduces the average number of per-input regex matching invocations by as much as 98.54%.

2 System Overview

Figure 1 illustrates the learning-augmented design of rule-matching systems. The rule ranker exploits the fact that many rulesets do not fix a mandatory ordering of rules, and it dynamically re-orders rules according to how likely they would match the given input. If the ranker successfully prioritizes the matching rule among the top N candidates, then the regex matching engine can effectively early-terminate after checking at most N rules. The figure also illustrates that the learned rule ranker can complement many existing optimization solutions. First, there is a string-matching pre-filter that first removes inputs unlikely to match any rule [7, 12]. Second, there are efforts on reducing the regex matching engine latency, e.g., PCRE's just-in-time compilation [4] and hardware-based regex acceleration [20, 25].

The rule ranker can take different realizations and ML/DL models. While ranking accuracy is a primary consideration in designing the ranker, achieving high accuracy typically comes with the cost of computation overhead and latency. This trade-off is crucial, as each input incurs the inference cost. Therefore, it is possible that a ranker does not speed up the overall system performance – in the context of rule-matching systems, these worst cases happen when a given input triggers a large amount of rule processing. Possible reasons include ***(1)*** the string-matching pre-filter fails to first remove unlikely inputs, or ***(2)*** the ranker fails to optimally prioritize rules.

2.1 Strawman Solutions for Rule Ranker

Static and heuristics-based solutions. If the overall system workload exhibits a long tail in the rule hit distribution (e.g., some rules account for a majority of matches), then both static and heuristics-based solutions can be effective. In particular, for cases where system workloads are assumed to rarely exhibit temporal dynamics, system operators can sort rules by statistically counting the number of rule hits in historical logs. Otherwise, heuristics such as least recently used (LRU) and least frequently used (LFU) can be used to improve the adaptability to sporadically temporal dynamics.

While both strawman solutions are simple, they can be sub-optimal due to the following reasons. First, while inputs are independent, these solutions rank rules based on the historical hit distribution, rather than any features of the current input. Second, they are suitable only for scenarios with known or long-tailed rule hit distributions. For cases and systems where different inputs can match different rules, LRU and LFU might not work well if the principle of locality does not hold.

Classification-based solutions. The rule-matching problem can be formulated as a multi-class classification problem in the machine learning domain. Specifically, assuming each rule is one class, we aim to predictively classify an input and rank rules by the likelihood score of each class. One widely-used non-DL classification technique is the logistic regression (LR). While being used traditionally for single-class classification, LR can be extended for multi-class classification through the one-vs-rest strategy. Unlike linear regression and support vector machine (SVM), LR is able to output probabilistic values, rather than binary answers. Probabilistic values are useful in comparing the relative likelihood of rules in a ruleset.

As a strawman solution, LR can be sub-optimal due to the following reasons. First, since the one-vs-rest-strategy [11] requires one model for each rule, a ruleset with r rules would result in r LR models. In addition to the training cost, each input effectively forces inferences over all r models. Second, since LR commonly targets linearly separable datasets, it is inadequate to model the space of matching inputs for rules of complicated regex conditions. §5 compares LR with DL-based solutions.

3 Learned Rule Ranker

Recent advances from deep learning communities have driven the availability of off-the-shelf DL models such as the popular

fully connected Deep Neural Networks (DNN) and Recurrent Neural Networks (RNN). DL models have the following advantages in the context of realizing learned rankers for modern systems. First, DL models can model complex non-linear datasets (e.g., rules with complicated conditions in our case). Second, DL models have hyper-parameters (e.g., number of hidden layers and neurons) that can easily be tuned to optimize the trade off between model accuracy and inference latency. Third, although DL models have been known to require a large amount of training data, rule-matching system inputs can be randomly generated and cheaply labeled.

Deploying DL models for learned rule ranker involves the following considerations and customizations.

Model selection. Given that inputs are strings, we consider the use of both DNN models (for their simplicity) and RNN models (for their ability to handle an arbitrary length of texts). As §5 shows, DNN typically has a lower inference latency, and RNN typically has a higher accuracy in prioritizing the matching rule among the top-N candidates. However, we argue that model selection goes beyond simply selecting the most accurate model configuration. Being a system building block, the learned ranker design must consider how the inference accuracy and costs would impact the end-to-end system performance. We illustrate this consideration with Figure 1 – if the regex matching engine is fast, having a relatively inaccurate rule ranker might be a reasonable design, especially if the overhead of checking one unnecessary rule is lower than the inference overhead of more accurate rankers. At the same time, a relatively inaccurate rule ranker might hurt the overall system performance, especially if the reduction in the amount of unnecessary rule checking does not adequately compensate the inference overhead.

Model inputs and outputs. The input layer of a neural network takes in a vector of real numbers. Since inputs in our case are a string of characters, they go through the process of word embedding to convert individual characters into 8-bit numbers in ASCII encoding. Furthermore, we note that DNN needs to take the entire input string at once, which forces the DNN input layer size to be at least as large as the input string. While the maximum input string length needs to be decided beforehand, system operators usually have statistics on the typical system workload. If an input string is shorter than the maximum length, we pad "0" at the end of the vectorized input. On the other hand, since RNN can take the input string in chunks, it can handle inputs of arbitrary length.

The output layer has a set of neurons where each neuron corresponds to a particular rule. Each neuron outputs a number between 0 and 1 representing the classification probability. We use these outputs to rank rules.

Input Generator. In addition to real-world traces of rule-matching system inputs, ranker training can happen with artificially generated matching/unmatching inputs. One advantage that the input generator offers is the large quantity of training data necessary for training DL models. To generate training inputs for a rule, our input generator runs Xeger [21] and Exrex [22], which are popular Python libraries for generating random strings from a given regex. Then, we randomly repeatedly choose S random characters from each of these generated inputs, and replace them with random characters. These mutated strings are then classified as either the `matching` and `unmatching`, by running the regex matching engine. The value of S is a crucial parameter – a larger S produces a nearly random unmatching string, and a smaller S changes only a few characters to simulate "near-miss" cases in the real world.

Training. With training inputs collected in the real world or generated by the input generator, we follow the popular training method of backward propagation with gradient descent. Since training data are labeled, the training is effectively a supervised learning. We use batch training, and each batch contains one input for each rule and one unmatching input. In addition, we train the DL model with 1,000 epochs, we use ReLu as the activation function at hidden layers and we use softmax at the output layer for outputting classification probabilities.

4 Implementation

We implement our learned ranker in Python 3.6 and TensorFlow 1.10.0. Our current implementation consists of ∼1,400 lines of Python code. In order to optimize the performance of TensorFlow on a CPU, we recompile the library with SSE 4.2, AVX and FMA instructions. We also enable just-in-time compilation for TensorFlow graphs.

To expose the target rule-matching system for the purpose of labelling inputs, we write a client stub. The client stub receives input strings from our input generator, and calls the target system's API. The communication between the input generator and the client stub happens over HTTP with messages in the JSON format.

5 Evaluation

Our major results include – ***(1)*** a learned rule ranker can reduce the average number of per-input regex matching invocations by as much as 98.54%, with a top-5 ranking accuracy of at least 96.16%. ***(2)*** Factoring in the rule ranker inference overhead, the learning-augmented design reduces the rule matching latency by as much as 78.81%. ***(3)*** We demonstrate that the ranking model design should consider a global optimization strategy, as having the most accurate model does not necessarily benefit the end-to-end system performance.

5.1 Methodology

Rulesets. While a learned rule ranker is not limited to security-related scenarios, two popular rulesets that are publicly available are ModSecurity CRS v3.0 [2] and Snort v3.0 [6]. The former is a web application firewall module, and the latter is a network-based intrusion detection system. We are interested in rules with complicated regular expressions with meta-characters, rather than simple string matching – our RS_{CRS} ruleset consists of 69 regex rules ranging from 20 to 3447 characters, and RS_{Snort} ruleset consists of 196 regex rules ranging from 21 to 243 characters. We note that rules can have matching criteria on multiple input fields, e.g., `ARGS`, `ARGS_NAMES`, `REQUEST_COOKIES`, and `REQUEST_COOKIES_NAMES` in CRS, and `HTTP_header`, `HTTP_uri`, and `HTTP_method` in Snort.

Workload datasets. Our experiments are based on following rule-matching system workloads: ***(1)*** the public ECML data set, WL_{ECML} [1], and ***(2)*** artificial data sets generated from the CRS and Snort rulesets, WL_{CRS} and WL_{Snort}. The former is primarily used as testing dataset. The latter can drive training and testing, by separately generating multiple sets of inputs.

For the artificial data sets, an input generator (c.f. §3) outputs random matching and unmatching strings, with respect to the given regex pattern. Unmatching strings allow us to test rules that require only some of the specified fields to match. The tool can generate a balanced workload to simulate the worst case where all rules are likely to be hit.

Testbed environment. We run popular rule-matching engines including PCRE [3], PCRE-JIT [4], and RE2 [5]. PCRE is the most widely used open-source regex matching engine, and the JIT optimization minimizes unnecessary parsing of the internal bytecode representation, especially the matching engine can contain many unused code branches from `if` and `switch` statements. RE2 is a fast and thread-friendly regex matching engine. We use Python and carry out experiments on a Ubuntu-based Azure VM with access to 8 cores of Intel Xeon E5-2673 running at 2.4 GHz and 3 GB of RAM.

5.2 Rule Ranking Accuracy

The primary goal of the learned rule ranker is to minimize unnecessary regex rule matching, and the system performance gain depends on its effectiveness in correctly prioritizing the matching rule as a top candidate. We quantify the effectiveness by the top-*N* accuracy, or the probability that the rule ranker successfully prioritizes the matching rule to be one of the first *N* rules to check. This subsection evaluates the different factors of the top-*N* accuracy.

Impacts of model selection. One factor that can impact the rule ranker's top-*N* accuracy is the learning model, and we empirically evaluate rule rankers implemented by DNN models (with two hidden layers of 128, 256, and 512 neurons), RNN models (with input chunk size of 32, 64, and 128 characters), and logistic regression (LR). We train each model with 100,000 inputs from WL_{CRS} and WL_{Snort}. Table 1a and 1b show empirical measurements for WL_{CRS} and WL_{Snort}, respectively. We make the following observations. RNN and LR have the highest and lowest top-*N* accuracies, respectively. While DNN (512) exhibits a 2.19% lower top-5 accuracy than RNN (32), it is $\sim 2.24\times$ faster. This trade-off suggests that simply using top-*N* accuracies as the selection metric might not benefit the entire system, and we further discuss how the trade-off between top-*N* accuracies and inference costs impacts the end-to-end rule matching throughput in §5.3.

Next, we look at inputs where the rule ranker fails to properly prioritize rules. Since these inputs require the regex matching engine to process more rules, they have a higher rule matching latency. Figure 2 illustrates the distribution of the predicted ranking for matching rules in the case of

Model	Top-1	Top-3	Top-5	Latency (μs)
DNN(128)	81.85%	94.39%	97.05%	11.65
DNN(256)	83.37%	95.18%	97.45%	14.68
DNN(512)	83.72%	95.61%	97.52%	21.44
RNN(128)	89.44%	97.51%	98.82%	33.43
RNN(64)	92.98%	98.55%	99.30%	39.88
RNN(32)	95.02%	99.23%	99.71%	48.02
LR	67.69%	82.89%	88.18%	48.25

(a) RS_{CRS}

Model	Top-1	Top-3	Top-5	Latency (μs)
DNN(128)	80.08%	93.34%	96.16%	11.75
DNN(256)	83.14%	94.69%	97.27%	15.42
DNN(512)	84.45%	95.34%	97.41%	22.44
RNN(128)	85.59%	96.88%	98.26%	41.62
RNN(64)	91.33%	98.19%	99.18%	46.21
RNN(32)	94.45%	99.22%	99.63%	56.21
LR	83.83%	93.29%	95.65%	93.19

(b) RS_{Snort}

Table 1: One factor that impacts rule ranking accuracy is the learning model selection: DNN (with two hidden layers of 128, 256, and 512 neurons), RNN (with input chunk size of 32, 64, and 128 characters), and logistic regression (LR). Results illustrate the trade off between top-*N* accuracies and ranking latency.

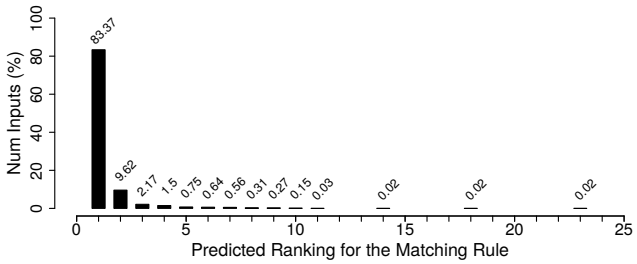
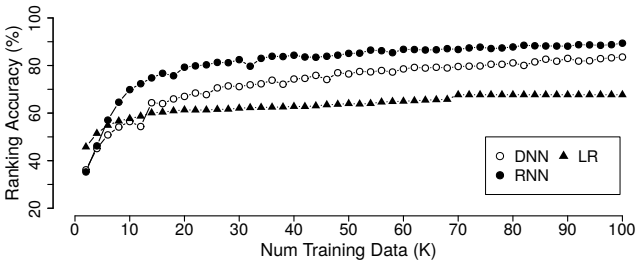
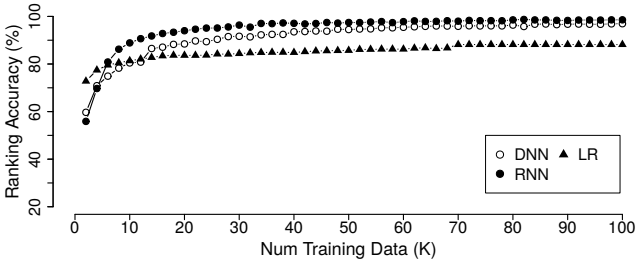


Figure 2: Distribution of the predicted ranking for matching rules. For 83.37% of inputs, the rule ranker is able to prioritize the matching rule as the first candidate.



(a) Top-1 accuracy



(b) Top-5 accuracy

Figure 3: Increase in ranking accuracy in terms of training data size, in the case of RS_{CRS}.

DNN(256). For 83.37% of inputs, the rule ranker is able to prioritize the matching rule as the first candidate. For 2.55% of inputs, it fails to prioritize the matching rule as a top-5 candidate. Interestingly, most of these inputs are relatively short, and the excessive padding might cause the ranker to infer those inputs incorrectly.

Impacts of training dataset size. Another factor that can impact the rule ranker's top-*N* accuracy is the amount of training data. Figure 3 shows how the accuracy increases for different DL/ML models in the case of RS_{CRS}, and we evaluate the accuracy by testing 1,000 randomly generated inputs (after each training iteration with 2,000 generated inputs). We note that our models generally start to converge after being trained with ∼90,000 inputs.

Impacts from imbalanced workload distributions. We acknowledge that, if the system operator has a complete prior knowledge of the workload distributions, it is possible to hard-code a static rule checking order. One case where this is particularly useful is the long-tailed rule hit distribution. In other words, the workload is imbalanced such that a majority of inputs match only a subset of the ruleset. Compared to WL_{CRS} and WL_{Snort}, the `ECML` dataset is relatively imbalanced. And, empirical results show that static order can significantly reduce the number of rules that the regex matching engine needs to process for WL_{ECML}.

However, given that the learned rule ranker is able to prioritize rules with each input's features, it can actually achieve a larger reduction – compared to static ordering, the DNN-based rule ranker reduces by 88.70% and 56.04% for RS_{CRS} and RS_{Snort} in the case of WL_{ECML}, respectively.

Regex engine	Without rule ranker	With rule ranker	Reduction
PCRE	1878.79 μsec	404.36 μsec	78.47%
PCRE-JIT	773.82 μsec	185.65 μsec	78.81%
RE2	206.01 μsec	55.15 μsec	73.22%

Table 2: Latency for matching one input with DNN(256)-based ranking model, on the RS_{CRS} ruleset.

Ruleset	No rule ranker	Rule ranker	Reduction
RS_{CRS}	22.38	1.68	92.49%
RS_{Snort}	91.56	1.34	98.54%

Table 3: Average number of regex rules that the regex matching engine needs to process for each input.

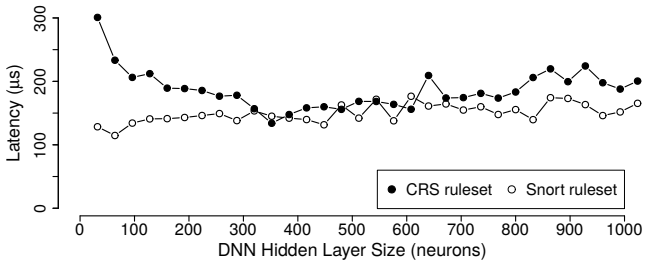
5.3 Rule Matching Latency Reduction

We next evaluate the rule matching latency reduction from the learned rule ranker, and we integrate the learned rule ranker into the PCRE, PCRE-JIT, and RE2 engines.

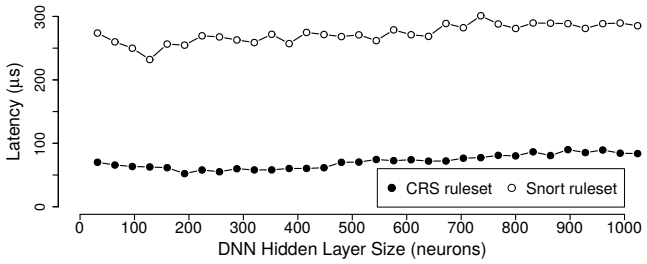
Table 2 shows the latency reduction for processing one input with different regex matching engines, on the RS_{CRS} ruleset. Each experiment runs 10,000 different inputs to avoid measurement noise. By fixing the ranking model, we observe that the reduction varies with different regex matching engines – the reduction ranges from 73.22% to 78.47%. This reduction in latency is correlated with the reduction in the number of rules that a regex matching engine needs to process. Table 3 shows that, for RS_{CRS}, the average number of regex matching invocations is 1.68, which is a 92.49% reduction from 22.38. The reduction in the case of RS_{Snort} is 98.54%. We note that, since both WL_{CRS} and WL_{Snort} are fairly balanced workloads (i.e., the number of matching inputs for each rule is roughly the same), per-input rule prioritization is key to reduction. And, the learned ranker exhibits a higher gain in cases where the overhead of processing an unnecessary rule is higher.

Finally, we highlight that the design of learning-augmented systems should consider the trade off between the learning's interference costs and the system's global performance gain. Figure 4 illustrates results from a DNN-based ruler ranker. Although having larger hidden layers improves DNN model accuracy, it does not necessarily benefit the end-to-end matching latency. This is due to the fact that, not only does an accurate model reduce unnecessary rule checking, but it also imposes higher inference costs to the end-to-end system performance. Unfortunately, the optimal model configuration depends on the execution environment and system configurations – in our case, the selection of regex matching engine and ruleset. For instance, Figure 4 shows that the optimal DNN hidden layer size is ∼192 neurons for RE2 with RS_{CRS} ruleset, ∼128 neurons for RE2 with RS_{Snort} ruleset, ∼256 neurons for PCRE-JIT with RS_{CRS} ruleset, and ∼64 neurons for PCRE with RS_{Snort} ruleset.

Discussion. We note that certain ruleset characteristics are also factors that potentially impact the overall rule matching



(a) PCRE-JIT rule-matching engine



(b) RE2 rule-matching engine

Figure 4: This figure illustrates that, although having larger hidden layers improves DNN model accuracy, it does not necessarily benefit the end-to-end matching latency. The reason behind this observation is the correlation between model accuracy and inference costs.

performance. One prominent example is the number of rules in the ruleset. Specifically, as the ruleset becomes larger, successfully prioritizing the matching rule means more rules can be skipped. On the other hand, a larger ruleset requires more complicated ML/DL models, which can impose additional overhead on the system performance. Given the lack of tools to automatically generate rulesets of different characteristics, our experiments in this section are based on two public rulesets, RS_{CRS} and RS_{Snort}. We leave the evaluation of ruleset characteristics as future work.

6 Related Work

Accelerating rule matching. The realization of regular expression patterns as finite state automata was first proposed by Kleene et al. [14] in the 1950s, and NFA and DFA have been widely used to formalize the description of regex patterns. Since most rule-based systems (such as traffic detection, data retrieving, and even DNA sequence matching) rely on regular expression patterns for condition matching, there have been efforts in speeding up regex pattern matching.

Some efforts focus on software optimizations for finite state automaton. PCRE-JIT [4] uses the just-in-time (JIT) library to minimize unnecessary parsing of the internal bytecode representation. Recently, Choi et al. proposed DFC [12], a memory-efficient and cache-friendly data structure that minimizes CPU stalls to maximize instruction-level parallelism. Kumar et al. [16] proposed a representation of regular expression patterns called Delayed Input DFA (D^2FA), which reduces the space requirement as compared to DFA. Further efforts build upon D^2FA [10, 18], and compress states and transitions. Finally, some efforts leverage hardware capabilities to speed up automaton – Mitra et al. [20] and Yuan et al. [26] explored the use of FPGAs and GPUs, respectively.

Being a system building block, the learned rule ranker should complement many existing optimizations.

Learning-augmented systems. Auto-tuning system parameters is a popular scenario for learning-augmented systems. OtterTune [8] is a database optimization tool. It uses a combination of supervised and unsupervised machine learning methods to reduce the parameter dimension, characterize observed workloads, and recommend configurations. CherryPick [9] demonstrates the potential of using Bayesian optimization and Gaussian process in predicting the best-performing cloud configuration for a given machine learning computation workload. Metis [17] addresses challenges that systems introduce to hinder the tuning robustness.

Furthermore, the networking community [24] has been applying ML and DL techniques to traffic prediction, traffic classification, resource management, network adaption, etc. To improve the QoS metric of video streaming, Mao et al. [19] used reinforcement learning in the rate adapting mechanism to continuously and adaptively adjust the streaming bit rate.

Finally, Kraska et al. [15] proposed the learned index to replace common indexes in databases. The learned index formulates the problem of database indexes as a DL predictive problem. It offers similar semantic guarantees, and a significant improvement in speed and memory efficiency.

Building on the success of these efforts, we explore whether the learned ranker can be a building block of learning-augmented systems.

7 Conclusion

This paper explores the potential and feasibility of learned ranker as a building block for learning-augmented systems. Particularly, we use rule-matching systems as a concrete scenario. To evaluate the benefits of the learned ranker, we have integrated it into popular regex matching engines. As future work, we plan to study challenges in training learned rankers, and apply learned rankers to other system scenarios.

Acknowledgments

We thank anonymous reviewers and our shepherd, Dr. Julia Lawall, for their constructive feedback and suggestions. This work is partly supported by the Youth Innovation Promotion Association of CAS and the National Natural Science Foundation of China (No.61772485 and No.61432016).

References

[1] ECML/PKDD 2007 Discovery Challenge - Analyzing Web Traffic. http://www.lirmm.fr/pkdd2007-challenge.

[2] ModSecurity - Open Source Web Application Firewall. https://modsecurity.org.

[3] PCRE - Perl Compatible Regular Expressions. http://www.pcre.org.

[4] PCRE JIT. http://www.pcre.org/original/doc/html/pcrejit.html.

[5] RE2. https://github.com/google/re2.

[6] Snort. https://www.snort.org.

[7] Hyperscan: Turbo Boosting Regular Expression Matching for Network Security Applications. In *NSDI (Operational Systems Track)*. USENIX, 2019.

[8] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD*. ACM, 2017.

[9] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *NSDI*. USENIX, 2017.

[10] Michela Becchi and Patrick Crowley. An Improved Algorithm to Accelerate Regular Expression Evaluation. In *ANCS*. ACM, 2007.

[11] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[12] Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, Kyoungsoo Park, and Dongsu Han. DFC: Accelerating String Pattern Matching for Network Applications. In *NSDI*, 2016.

[13] Russ Cox. Regular Expression Matching Can Be Simple And Fast (But Is Slow in Java, Perl, PHP, Python, Ruby, ...). http://swtch.com/~rsc/regexp/regexp1.html, 2007.

[14] Stephen Cole Kleene. Representation of Events in Nerve Nets and Finite Automata. Technical report, United States Air Force, 1951.

[15] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *SIGMOD*. ACM, 2018.

[16] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In *ACM SIGCOMM Computer Communication Review*. ACM, 2006.

[17] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. Metis: Robustly Optimizing Tail Latencies of Cloud Systems. In *ATC*. USENIX, 2018.

[18] Alex X Liu and Eric Torng. An Overlay Automata Approach to Regular Expression Matching. In *INFOCOM*. IEEE, 2014.

[19] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *SIGCOMM*. ACM, 2017.

[20] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling PCRE to FPGA for Accelerating Snort IDS. In *ANCS*. ACM, 2007.

[21] Colm O'Connor. Xeger. https://pypi.org/project/xeger/, 2018.

[22] Adam Tauber. Exrex. https://pypi.org/project/exrex/, 2018.

[23] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytic. In *NSDI*. USENIX, 2016.

[24] Mowei Wang, Yong Cui, Xin Wang, Shihan Xiao, and Junchen Jiang. Machine Learning for Networking: Workflow, Advances and Opportunities. *IEEE Network*, 2018.

[25] Norio Yamagaki, Reetinder Sidhu, and Satoshi Kamiya. High-speed Regular Expression Matching Engine Using Multi-character NFA. In *FPL*. IEEE, 2008.

[26] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. GPU-based NFA Implementation for Memory-efficient High-speed Regular Expression Matching. In *PPoPP*, 2012.