A Taxonomy of ML for Systems Problems

Martin Maas
Google Research, Brain Team

***Abstract*—Machine learning has the potential to significantly improve systems, but only under certain conditions. We describe a taxonomy to help identify whether or not machine learning should be applied to particular systems problems, and which approaches are most promising. We believe that this taxonomy can help practitioners and researchers decide how to most effectively use machine learning in their systems, and provide the community with a framework and vocabulary to discuss different approaches for applying machine learning in systems.**

Machine learning (ML) has transformed many research areas, from image recognition to natural language processing. ML has also had a significant impact on computer systems and inspired the development of new systems for designing and training ML models (e.g., TensorFlow), as well as new hardware (e.g., TPUs).

In contrast to such *Systems for ML* research, *ML for Systems* is only now seeing more attention. While ML has long been used in areas such as branch prediction, recent work has shown promising results in caching, compilers, and cluster scheduling. These advances indicate that ML could hold the key to improving many areas in computer systems. However, these successes hide the fact that ML does not always lead to the immediate wins that its popularity promises. Applying ML to systems does not always outperform highly tuned non-ML solutions, and even if ML improves a particular metric, its resource cost does not always justify the improvement.

This article makes the case that while ML has the potential to improve systems, it does so only in certain cases. Furthermore, different ML techniques are suitable for different problems. We therefore categorize systems problems and develop a taxonomy for identifying whether ML can be applied, and what strategies might be suitable. We also provide a bibliography[1] that matches existing work to this taxonomy. We believe that our approach can help practitioners and researchers decide how to most effectively use ML in their systems and provide the research community with a framework to discuss ML for Systems strategies.

Digital Object Identifier 10.1109/MM.2020.3012883
Date of publication 30 July 2020; date of current version 1 September 2020.

Published by the IEEE Computer Society

BACKGROUND

"Systems" is a broad term. To ground discussion in a common terminology, we therefore focus on "system policies." Given a software or hardware component that makes decisions related to the execution of computer programs, a system policy describes how these decisions are made. Compiler passes, branch predictors and memory allocators are all examples of system policies. When we talk about ML for Systems, we therefore mean "using ML in the implementation of a system policy." Specifically, we focus on supervised and reinforcement learning (RL); other techniques such as learning-to-learn, transfer learning, or representation learning have applications in systems as well,[2] but are out of scope for this taxonomy.

System policies typically fall into four categories that oftentimes correspond to the degree to which a system has been optimized.

- *An ad hoc policy* based on assumptions at the time of development. Consider an inlining policy in a compiler: An ad-hoc heuristic could inline all functions with less than 10 instructions.
- *An empirically tuned policy* that has been optimized for a set of benchmarks. This is the type of policy often published in research papers. For example, such a policy could consider the call graph and apply carefully crafted inlining rules, chosen to optimize performance across a set of benchmarks (e.g., SPEC).
- *A data-driven* policy that optimizes towards a specific target. In contrast to an empirically-tuned policy that uses benchmarks as a proxy for the real target, this policy is tuned to the target itself (e.g., feedback-directed inlining).
- *An adaptive data-driven policy* that does not make the same decision for the same target every time, but adapts online in response to its own decisions. An example is trace-based JIT compilers that re-evaluate and revise inlining decisions over time.

ML is often defined as the ability of a program to learn from experience. By this definition, data-driven policies are a form of ML, although potentially a rudimentary one. Most data-driven policies collect databases of examples and learn from them, either by exploring a search space (e.g., autotuners) or by building a lookup table and using it in future executions (e.g., profile-guided optimization). However, we typically start explicitly calling this approach ML only when we use tools from the ML literature, such as support vector machines, decision trees, or neural networks.

What sets ML approaches apart from lookup tables is that they can potentially *generalize* to unseen cases. An early example is neural branch predictors. Recently, there has been an explosion of such techniques, ranging from learning compiler optimizations[3] to cluster scheduling.[4] Note that a complicated ML technique is not a necessary requirement to generalize. For example, work on index structures has shown that while it is possible to learn a key distribution using neural networks, a similar goal can be achieved by fitting splines to its cumulative distribution function.[5]

This article makes the case that while ML has the potential to improve systems, it does so only in certain cases. Furthermore, different ML techniques are suitable for different problems. We therefore categorize systems problems and develop a taxonomy for identifying whether ML can be applied, and what strategies might be suitable.

Note that highly tuned and complex heuristics are similar to data-driven policies. For example, compilers have cost models to accurately predict performance and generalize to unseen programs, but only recently has this problem been revisited using modern ML techniques such as neural networks.[6] A corollary is that if a system has been well-tuned, it can be difficult to improve the baseline with a learned policy. One interpretation is that engineers have used a "real-life version of gradient descent" to move the system to a local optimum, not unlike what ML would do. This shows that applying ML to systems does not represent a fundamental departure from systems research, but only provides a new set of tools.

This aspect is often obscured in the discussion of ML for Systems, in part due to the

popularity of end-to-end learning. Modern ML techniques can learn very complex behavior, and it is therefore possible to train models that learn complex policies end-to-end. We have seen this approach in areas ranging from RL for caches, to end-to-end cluster scheduling, to RL for compilers. While these approaches often work, they are not always data efficient, consume large amounts of resources, and sometimes do not conclusively outperform strong baselines. Many problems have a known structure that can be captured in a handwritten heuristic. However, end-to-end learning has to learn this structure from scratch and may re-learn known facts, at the cost of maximizing performance on the otherwise intractable part.

We therefore argue that effectively applying ML to systems requires identifying which part of a systems policy requires ML, and developing specific ML techniques for this part. This is supported by the fact that many recent successes of ML for Systems have focused on specific subproblems rather than end-to-end learning (e.g., learning-based index structures, learned cost models[6]). Note that these learning techniques were used in areas that were already data-driven. As such, there was already an interface for the learning technique to fit into the conventional portion of the system, as well as strong baselines. Meanwhile, when learning replaces an end-to-end heuristic, it can be hard to attribute which gains are due to shifting to a data-driven approach versus ML. The resulting tradeoff space can be difficult to reason about, in part because the vocabulary to discuss the way ML is used is often missing.

CONSTRAINTS AND TRADEOFFS

When considering whether to apply ML, several tradeoffs need to be considered: The problem needs to be well-suited to ML, the deployment constraints need to allow for ML, and suitable data needs to be available. We now discuss these considerations in turn.

Problem Suitability

- *Target:* System policies have different optimization metrics (e.g., resource utilization or the worst case latency). ML can help for metrics that are difficult to reason about, while metrics that can be optimized analytically might be addressed without ML.
- *Data-driven baselines:* If the main benefit of ML is to make a heuristic data-driven, simple data-driven methods should be tried first. In some cases, this is ML (e.g., because the input features are too complex), but in others, a lookup table might yield most of the benefits. For example, table-based branch predictors are competitive with ML approaches.
- *High-dimensional input space:* If the number of possible inputs is small, a lookup table can be used to memorize all predictions instead of using ML for generalization.

Deployment Constraints

- *Latency:* ML for Systems differs from areas such as NLP in its ultralow latency requirements. OSs and runtimes often need to make decisions in micro/nanoseconds, branch predictors in cycles. In contrast, even small neural networks often take hundreds of microseconds and GBRTs take tens of microseconds. More complex models can take tens of milliseconds. Before applying ML, it is therefore necessary to identify the latency requirements—offline policies (e.g., in compilers) are often more latency-tolerant than online policies (e.g., in schedulers).
- *Space/time overheads:* Even if prediction latency is low, models that run often can consume significant resources (CPU/DRAM). In our case study, we use a model for memory allocation, with 250 000 allocations/second. The model takes $>100\ \mu$s; running it at every allocation is thus infeasible. To use ML in such a setting, formulating the problem so that prediction results can be cached is crucial.
- *Custom hardware:* Scenarios where latencies are large (e.g., compilers) can use GPUs/TPUs, and hardware policies (e.g., branch predictors, prefetchers) can have custom implementations. Other models typically run on the CPU, which can be inefficient for neural networks. Compiling and inlining the model directly into the code can help.

- *Risk/robustness/interpretability*: Models sometimes mispredict and systems need to adapt. The specific use case dictates the robustness requirements and risk, and many non-ML systems policies are not 100% robust themselves. However, many ML models are opaque, which makes problems more difficult to track down when they occur. In high-risk scenarios, it can therefore be required to rely on interpretable approaches, even if they yield lower accuracy (e.g., decision trees).

Data Availability

- *Privacy/Security:* It needs to be ensured that the data used for learning does not expose sensitive data (e.g., encryption keys).
- *Offline Versus Online Learning:* Some policies learn online (e.g., branch predictors) while others train offline. For the latter, ML needs to be integrated into the development and QA cycle (e.g., is the model deployed with a new binary or updated dynamically?). Meanwhile, online training can be challenging for expensive models that require accelerators to train.
- *Distribution Shifts:* Models take time to train and require quality control. ML therefore introduces challenges in cases where the output distribution of the policy shifts quickly. Such cases may require an online approach.

Applying ML requires trading off these priorities. For example, it is sometimes more important to be robust than achieve perfect accuracy. Even if ML achieves better accuracy on a given task, it may therefore not be suitable.

TAXONOMY OF ML FOR SYSTEMS

To give researchers and practitioners a framework to reason about ML for Systems, we divide ML for Systems problems into five categories that we believe capture most problems (they can overlap). We start by classifying existing work.

- *Anomaly detection:* Detect when a system does not behave as expected (e.g., system failures, security incidents, interference, performance regressions[7]).
- *Forecasting:* Predict future behavior of a system. This includes program speculation (e.g., prefetching and branch prediction), object lifetime prediction,[8] cardinality estimation in databases, and system or network resource demand forecasting.
- *Extrapolation:* Given a policy for a known subset of inputs, extend it to unseen inputs. This can include classification tasks in schedulers (e.g., whether a program is scale-up or scale-out[4]), cost models in compilers, or selecting configuration parameters.
- *Discovery:* Generate a new/better policy. This includes policies that could not be handwritten, either because the rules are too counterintuitive for a human to come up with (e.g., caching policies identified through learning) or because they are based on a large amount of data. Examples include custom inlining heuristics based on performance profiles or data-specific index structures for databases.
- *Optimization:* Exploring a potentially large space to find a good or optimum solution (e.g., ML for hardware design, autotuners).

The full bibliography is available online.[1] Note that all of these use cases can and are being solved without ML. We argue that this list represents a hierarchy of how hard it is to achieve improvements with ML, from easiest to hardest.

Anomaly Detection

Detecting faults in mechanical and control systems is one of the traditional uses of AI. Anomaly detection is an attractive target for ML because it is often data-driven already: Many anomaly detectors start from a set of examples and cluster them to detect outliers. Modern ML adds new tools, such as autoencoders, where the reconstruction error measures anomaly.[7]

Baselines: Simple data-driven baselines should be tried first (e.g., clustering). If a baseline is used that is not data-driven, the main lift from ML might be the result of switching from a heuristic to a data-driven approach.

Strategies: The first step is to select input features and model their statistical properties. Before using complex techniques, it is worth trying clustering approaches that show a difference between normal and anomalous examples. Complex techniques can be appropriate when the features cannot be readily encoded for clustering (e.g., if they represent graphs), or if it is unknown which of a large set of features to use. In such cases, complex neural networks such as graph neural networks or recurrent neural networks can be appropriate, either used as an embedding, or in an autoencoder setting.

Forecasting

Most computer systems use some form of forecasting, either implicitly or explicitly. Any form of hardware speculation relies on forecasting (e.g., prefetching), and schedulers adjust resources based on predicting future resource usage from current/past usage.

Baselines: Because most systems use forecasting, it is important to determine whether these baselines are data-driven or heuristic-based. If they are not data-driven, prior to creating a complex ML-based approach, a simpler data-driven baseline should be tried. In many cases, the simplest baseline is a lookup table that maps input features to predictions (e.g., branch predictors or inline caches).

Strategies: ML models can provide benefits over table-based forecasting when generalization is required. For example, recent work[2] that learned from application-level features in storage shows that a lookup table approach degrades over time as previously unseen features appear. In such cases, different models can be applied, from decision trees to neural networks.

Extrapolation

Most extrapolation in systems is heuristic-driven and often performs classification. For example, a scheduler might compare counters against hard-coded thresholds.

Baselines: Extrapolation should start from a data-driven baseline. However, as previously observed, highly tuned heuristics are similar to data-driven policies and may therefore be appropriate baselines as well.

Strategies: Extrapolation strategies are often problem-specific. Collaborative filtering[4] has been used (e.g., for workload classification), but supervised learning also works for many areas, such as predictions based on stack traces[8] or learning memory access patterns.[9]

Discovery

Discovery is about designing previously unknown policies, such as new caching strategies.[10] There are two variants: Discovering a new general policy that is intended to be universally used, and learning a specialized policy for a particular set of workloads (i.e., data-driven).

Baselines: Since the goal of discovery is to find a new algorithm, it should be evaluated in the same way as existing approaches of the same (data or nondata-driven) type.

Strategies: A common strategy is RL, where a policy is learned by exploring different decisions in a simulator or real environment and updating the policy based on a reward. While most problems can be framed as RL, it is in practice harder to train a model using it. Simpler approaches are available: One approach is to use an expensive method to solve instances of the problem offline (e.g., SAT/ILP solvers) and use them as training inputs for imitation learning.[3] Another alternative is to design several parameterized subpolicies and learn a policy that picks the best one (i.e., a bandit-based approach).

Optimization

In some cases, ML can be used to solve a static optimization problem (e.g., a complex scheduling problem). In contrast to discovery, this approach is not learning a general policy to solve new problem instances, but is using ML to explore the search space of a specific instance.

Baselines: There are many well-known optimization techniques, including genetic algorithms and simulated annealing, some of which have been shown to work in the same areas as ML (e.g., playing video games[11]). While ML techniques can be used for optimization (e.g., gradient descent is a form of optimization), this alone

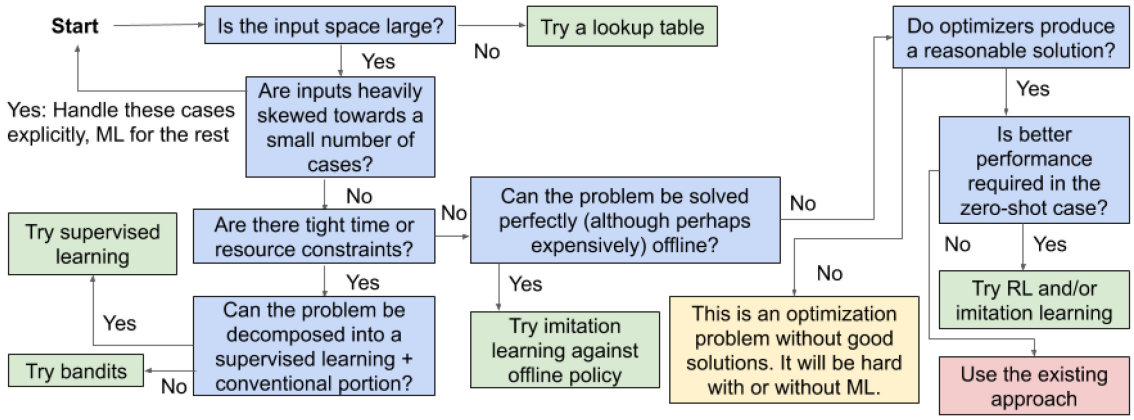

Figure 1. How to decide which ML approach to use.

does not constitute learning and is not necessarily better than alternatives.

When applying ML to an optimization problem, it is important to identify whether the goal is to learn a policy that transfers to new problem instances (i.e., discovery) or whether it is to solve a standalone instance. While discovery problems need to be evaluated based on their zero-shot performance for a previously unseen example, optimization problems need to be evaluated against other optimization techniques, such as genetic algorithms. Both baseline and ML approach need to be evaluated with the same amount of resources.

Strategies: RL has been successfully used in optimization problems, by learning a policy that selects the next design decision, combined with a value function that estimates the quality of a particular choice. The policy function and value function can potentially be reused when solving a new optimization problem, leading to transferability. It is, however, possible that these functions overfit to a particular optimization example, so transferability cannot be taken for granted.

For design spaces that are low-dimensional, Bayesian optimization frameworks can work well. It is also important to not dismiss alternative optimization strategies in favor of ML, particularly if transferability is not required.

CHOOSING AN ML STRATEGY

We now discuss how to determine whether a problem is suitable for ML. While major improvements from ML have been shown in all five areas, the further we go from "anomaly detection" to "optimization," the more a model has to learn about how the system works, and the more data/examples are required.

The first step is to check whether the input features are predictive of the output. For low-dimensional prediction problems, this can be done with a lookup table baseline that stores a prediction for every possible input. For higher dimensional problems, replaying a run with an oracle (e.g., in a simulator) is an alternative. This indicates the headroom.

The next decision is the scope of the learned system policy. Almost every system could be framed as an end-to-end RL problem. However, such a model needs to not only learn the statistical properties of the data but also everything about its environment. It can therefore be advantageous to separate the prediction problem from the rest of the system, to limit the complexity of the function that needs to be learned. As shown in the next section, it is sometimes possible to decompose an end-to-end problem into a supervised learning portion and a (traditional) algorithmic portion that consumes these predictions. Alternatively, the latter (reduced) problem may be solved with ML itself (e.g., RL).

Once the learning problem has been defined, an ML technique needs to be chosen. We frame our recommendations as a decision diagram (see Figure 1). For each presented ML type, different learning strategies are available. For example, supervised learning could use neural networks or decision trees. One important factor

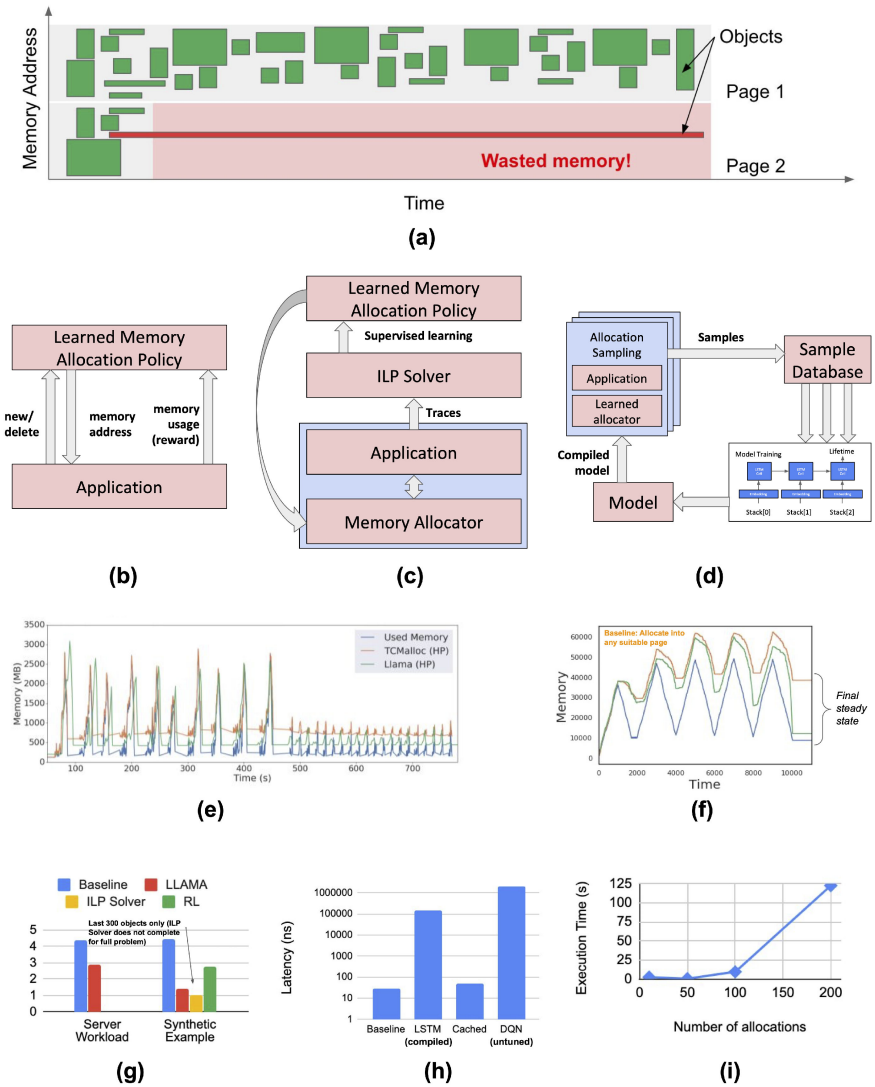

Figure 2. ML for memory allocation.[8] (a) Visualizing memory fragmentation. (b) Reinforcement learning approach. (c) Imitation learning approach. (d) Decomposed approach using supervised learning and a new type of memory allocator. (e) C++ server workload memory reduction (running against synthetic requests).[8] (f) Synthetic memory trace. (g) Final steady-state fragmentation. (h) Allocation/inference latency. (i) ILP solver scalability.

is how to deploy a model within a system, based on resource constraints. Deployment ranges from compiling a model directly into code to running the model offline (e.g., at compile time). The uniqueness of systems problems and their constraints may necessitate new ML techniques.

CASE STUDY

We demonstrate how the previous insights apply to recent work on ML for memory allocation.[8] The goal of this work was to reduce fragmentation in C++ workloads with huge (2 MiB) pages and varying memory footprints. Since C++ cannot move objects, long-lived objects can prevent entire pages from being released to the OS [see Figure 2(a)], causing fragmentation. To solve this problem, a memory allocator needs to reason about object lifetimes and group objects with similar lifetimes together.

This requires the memory allocator to predict future behavior of the application, which suggests that this could be a target for ML. The

allocator represents the following *system policy*: Given a sequence of allocation requests (each with a size and an unknown lifetime), the allocator needs to place objects in virtual address space such that the number of live 2 MiB pages (i.e., pages containing at least one object) is minimized. At the time of allocation, we know the current stack trace and the size of the allocation. As such, we need to solve a *forecasting* problem.

Our baseline is TCMalloc, which organizes objects by size but ignores lifetime. Figure 2(e) shows used memory and actual memory footprint for running a server workload against synthetic inputs. The baseline incurred over 2× fragmentation (*footprint*/*used*) on average, over 4× at low memory usage. Our goal was to reduce this fragmentation. Since the trace is large (110M allocations), we also generated a synthetic driver and baseline that replicates similar behavior with 5000 allocations [see Figure 2(f)].

Reinforcement Learning

This looks like a perfect setup for RL, with a sequence of decisions (where to place each object) and a reward function (memory fragmentation). We therefore started collecting traces and built a simulator to replay these traces, which would allow an RL policy to learn a good allocation strategy. However, several constraints make RL challenging for this scenario. First, the state space is large and complex, as there are 2^{64} addresses. Furthermore, the number of allocations is large (millions/min), and rewards are sparse, creating credit assignment challenges.

We implemented a naive DQN model that observes the state of all hugepages and picks an allocation target for our synthetic trace. This simple model outperformed the baseline, but needs to run every allocation and takes 2 ms (TCMalloc's fast path takes 8.3 ns). Even if optimizations improved this latency by 1000×, this approach would thus be impractical.

Imitation Learning

Given full allocation traces of a program, the problem can be solved offline. It reduces to a two-dimensional packing problem, which can be solved retroactively using an ILP solver. Given such a solution, we could train a policy against it. This yields low fragmentation [see Figure 2(g)], but the ILP approach does not scale [see Figure 2(i)], as applications have >10M allocations/minute (the solver did not even solve the full synthetic trace). Furthermore, just like with RL, we would need a DQN that learns from these solutions, and running such a model every allocation is impractical.

Decomposed Supervised Learning

We tackle these challenges by breaking up the prediction problem. Instead of learning object placement end-to-end, we decomposed it into a supervised learning portion that predicts (potentially incorrectly) the lifetime of an object based on its stack trace. We built a new memory allocation algorithm (LLAMA) that relies on these predictions and can detect when past predictions were incorrect. Using this approach, we can avoid running the model at every allocation, because results can be cached (we use a hash table and identify stack traces based on a cheap fingerprinting mechanism).

Following the rules laid out in this article, we validated our approach by using it with a lifetime oracle in the simulator. We then replaced the oracle with a lookup table, but found that this table did not generalize across different versions of an application. We then decided to use an LSTM neural network that we compiled directly into the CPU code to maximize performance (a similar approach could be used for a DQN).

This model is easier to train than RL, because training is supervised. This also means that training does not require full traces from entire runs (like RL or imitation learning). We can instead sample individual allocations. Even so, running the LSTM every allocation takes 144 μs and is therefore impractical [see Figure 2(h)]. However, since predictions now depend only on the current stack trace and no state, we can cache predictions, bringing the predictions down to ≈20 ns. LLAMA reduces steady-state fragmentation by 43% in Figure 2(e) (up to 78% for other workloads[8]).

Lessons Learned

This approach shows that while it is often possible to apply end-to-end learning to an ML for Systems problem, it is not always the best approach. The successful solution required us to apply the rules laid out in this article: We moved from an

empirically-tuned policy (i.e., past memory allocators) to a *data-driven policy*. Since a *lookup table* alone was not sufficient, we applied ML to learn an embedding of stack traces, using supervised learning. The other parts of the problem (selecting how to allocate memory based on the prediction) were solved with conventional heuristics that tolerate mispredictions.

CONCLUSION

ML for Systems is an emerging area. To maximize its potential, ML needs to be used in the most effective way and evaluated against suitable baselines. Meanwhile, systems-specific requirements such as low latency and large input/output spaces necessitate systems-specific innovation on the ML side. Our goal is to establish a framework and vocabulary to discuss these alternatives and tradeoffs. We see this article and the accompanying bibliography[1] as a contribution toward this discussion and believe that these points are going to be increasingly important as we are seeing more ML for Systems work.

ML for Systems is an emerging area. To maximize its potential, ML needs to be used in the most effective way and evaluated against suitable baselines.

ACKNOWLEDGMENTS

The author would like to thank A. Klimovic, A. Goldie, A. Mirhoseini, C. Raffel, H. Lim, J. Laudon, M. Phothilimthana, M. Abadi, R. Singh, R. Frostig, and S. Roy for their feedback.

REFERENCES

1. Supplementary material/bibliography, 2020. [Online]. Available: https://github.com/google-research/ml-for-systems-taxonomy
2. G. Zhou and M. Maas, "Multi-task learning for storage systems," in *Proc. ML Syst. Workshop*, 2019. [Online]. Available: http://mlforsystems.org/assets/papers/neurips2019/multi_task_zhou_2019.pdf
3. C. Mendis, C. Yang, Y. Pu, S. Amarasinghe, and M. Carbi, "Compiler auto-vectorization with imitation learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 14625–14635.
4. C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2014, pp. 127–144.
5. T. Neumann, "The case for B-tree index structures," 2017. [Online]. Available: http://databasearchitects.blogspot.com/2017/12/the-case-for-b-tree-index-structures.html
6. C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin. "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 4505–4515.
7. M. Alam, J. Gottschlich, N. Tatbul, J. S. Turek, T. Mattson, and A. Muzahid, "A zero-positive learning approach for diagnosing software performance regressions," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 11627–11639.
8. M. Maas, D. G. Andersen, M. Isard, M. M. Javanmard, K. S. McKinley, and C. Raffel, "Learning-based memory allocation for C++ server workloads," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2020, pp. 541–556.
9. M. Hashemi *et al.*, "Learning memory access patterns," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 1919–1928.
10. D. S. Berger, "Towards lightweight and robust machine learning for CDN caching," in *Proc. 17th ACM Workshop Hot Topics Netw.*, 2018, pp. 134–140.
11. F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," 2017, *arXiv:1712.06567*.

Martin Maas is a Research Scientist at Google. His research interests span language runtimes, computer architecture, systems, and machine learning. Maas received the Ph.D. degree in computer science from UC Berkeley. Contact him at mmaas@google.com.