Declarative MACHINE LEARNING SYSTEMS

THE FUTURE OF MACHINE LEARNING WILL DEPEND ON IT BEING IN THE HANDS OF THE REST OF US.

PIERO MOLINO AND CHRISTOPHER RÉ

In the past 20 years ML (machine learning) has progressively moved from an academic endeavor to a pervasive technology adopted in almost every aspect of computing. ML-powered products are now embedded in every aspect of our digital lives: from recommendations of what to watch, to divining our search intent, to powering virtual assistants in consumer and enterprise settings. Moreover, recent successes in applying ML in natural sciences have revealed that ML can be used to tackle some of the hardest real-world problems that humanity faces today.[19]

For these reasons ML has become central to the strategy of tech companies and has gathered even more attention from academia than ever before. The journey that led to the current ML-centric computing world was hastened by several factors, including hardware improvements that enabled massively parallel processing, data-infrastructure improvements that resulted in the

storage and consumption of massive data sets needed to train most ML models, and algorithmic improvements that allowed for better performance and scaling.

Despite these successes, these examples of ML adoption are only the tip of the iceberg. Right now, the people training and using ML models are typically experienced developers with years of study working within large organizations, but the next wave of ML systems should allow a substantially larger number of people, potentially without any coding skills, to perform the same tasks. These new ML systems will not require users to fully understand all the details of how models are trained and used for obtaining predictions—a substantial barrier to entry—but will provide them a more abstract interface that is less demanding and more familiar. Declarative interfaces are well-suited for this goal, by hiding complexity and favoring separation of interest, and ultimately leading to increased productivity.

We worked on such abstract interfaces by developing two declarative ML systems—Overton[16] and Ludwig[13]—that require users to declare only their data schema (names and types of inputs) and tasks rather than having to write low-level ML code. The goal of this article is to describe how ML systems are currently structured, to highlight which factors are important for ML project success and which ones will determine wider ML adoption, the issues current ML systems are facing, and how the systems we developed address them. Finally, the article describes what can be learned from the trajectory of development of ML and systems throughout the years and what the next generation of ML systems will look like.

Software engineering meets ML

A factor not appreciated enough in the successes of ML is an improved understanding of the process of producing real-world ML applications and how different it is from traditional software development. Building a working ML application requires a new set of abstractions and components, well characterized by David Sculley et al.,[18] who also identified how idiosyncratic aspects of ML projects may lead to a substantial increase in technical debt (i.e., the cost of reworking a solution that was obtained by cutting corners rather than following software engineering principles). These bespoke aspects of ML development are opposed to software engineering practices, with the main ones responsible being the amount of uncertainty at every step, which leads to a more service-oriented development process.[1]

Despite the bespoke aspects of each individual ML project, researchers first and industry later distilled common patterns that abstract the most mechanical parts of building ML projects in a set of tools, systems, and platforms. Consider, for example, how the availability of projects such as scikit-learn, TensorFlow, PyTorch, and many others allowed for wide ML adoption and quick improvement of models through more standardized processes: Where before implementing a ML model required years of work for highly skilled ML researchers, now the same can be accomplished in a few lines of code that most developers would be able to write. In a recent paper Sara Hooker argues that availability of accelerator hardware determines the success of ML algorithms potentially more than their intrinsic merits.[8] We agree with

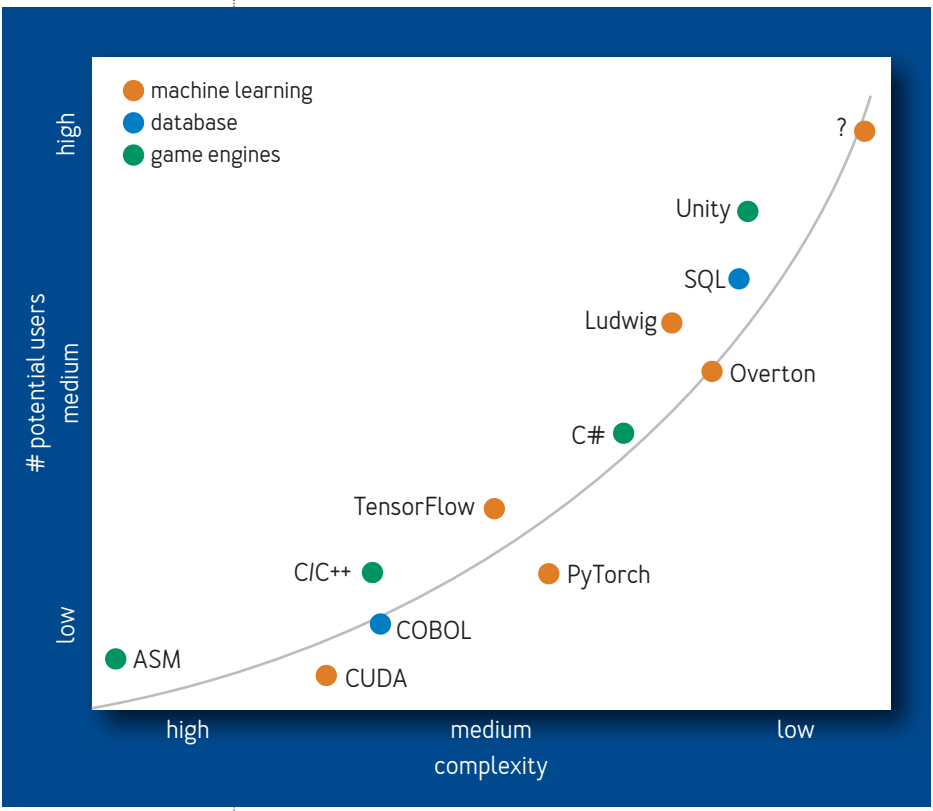
that assessment and add that availability of easy-to-use software packages tailored to ML algorithms has been at least as important for their success and adoption, if not more so.

The coming wave of ML Systems

Generations of work on compiler, database, and operating systems may inspire new foundational questions about how to build the next generation of ML-powered systems that will allow people without ML expertise to train models and obtain predictions through more abstract interfaces. One of the many lessons learned from those systems throughout the history of computing is that substantial increases in adoption always come with separation of interests and hiding of complexity, as shown in figure 1, an approximate depiction of the relationship between the complexity of learning and using a software tool (languages, libraries, or entire products) and the number of users potentially capable of using it, across different fields of computing.

As a compiler hides the complexity of low-level machine code behind the facade of a higher level, more human-readable language, and as a database management system hides the complexity of data storage, indexing, and retrieval behind the facade of a declarative query language, so should the future of ML systems steer toward hiding complexity and exposing simpler abstractions, likely in a declarative way. The separation of interests implied by such a shift will allow highly skilled ML developers and researchers to work on improving the underlying models and infrastructure in a way that is similar to how compiler

FIGURE 1: **COMPLEXITY OF LEARNING VERSUS USAGE OF SOFTWARE TOOLS**



maintainers and database developers improve their systems today, while allowing a wider audience to use ML technologies by interfacing with them at a higher level of abstraction. This is much like a programmer writing code in a simple language without knowing how it compiles in machine code or a data analyst writing SQL queries without knowing the data structures used in the database

indices or how a query planner works. These analogies suggest that declarative interfaces are good candidates for the next wave of ML systems, with the hiding of complexity and separation of interest being the keys to bringing ML to noncoders.

Following is an overview of the ML development life cycle and the current state of ML platforms, together with some challenges and desiderata for ML systems. This article describes some initial attempts at building new declarative abstractions that we worked on first-hand that address those challenges. These declarative abstractions proved useful for making ML more accessible to end users by avoiding the need to write low-level error-prone ML code. Finally, the article presents the lessons learned from these attempts and speculates on what may lay ahead.

MACHINE LEARNING SYSTEMS

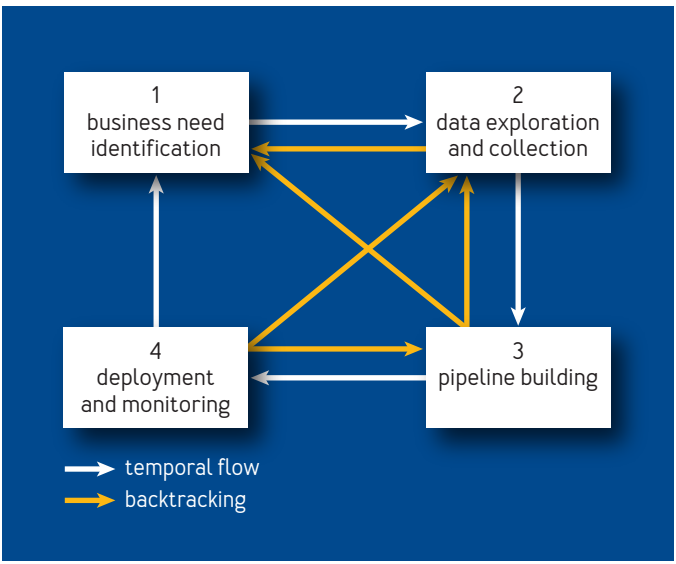
Machine Learning development life cycle

Many descriptions of the development life cycle of machine-learning projects have been proposed, but the one adopted in figure 2 is a simple coarse-grained view composed of four high-level steps:

1. Business need identification
2. Data exploration and collection
3. Pipeline building
4. Deployment and monitoring

Each of these steps is composed of many substeps, and the whole process can be seen as a loop, with information gathered from the deployment and monitoring of an ML system helping to identify the business needs for the next cycle. Despite the sequential nature of the process, each

FIGURE 2: **LIFECYCLE OF A ML PROJECT**



step's outcome is highly uncertain, and negative outcomes of each step may send the process back to previous steps. For example, data exploration may reveal that the available data does not contain enough signal to address the identified business need, or pipeline building may reveal that, given the available data, no model can reach a high enough performance to address the business need, and thus new data should be collected.

Because of the uncertainty that is intrinsic in ML processes, writing code for ML projects often leads to idiosyncratic practices: There is little to no code reuse and no logical/physical separation between abstractions, with even minor decisions taken early in the process impacting

every aspect of the downstream code. This is opposed to what happens, for example, in database systems, where abstractions are usually well defined: In a database system, changes in how you store your data don't change your application code or how you write you queries, while in ML projects, changes in the size or distribution of your data end up changing your application code, making the code difficult to reuse.

Machine-Learning platforms and AutoML

Each step of the ML process is supported by a set of tools and modules, with the potential advantage of making these complex systems more manageable and understandable.

Unfortunately, the lack of well-defined standard interfaces between these tools and modules limits the benefits of a modularized approach and makes architecture design choices difficult to change. The consequence is that these systems suffer from a cascade of compounding effects if errors or changes happen at any stage of the process, particularly early on (i.e., when a new version of a tokenizer is released with a bug), and all the following pieces (embedding layers, pretrained models, prediction modules) start to return wrong results. The data-frame abstraction is so far the only widely used contact point between components, but it could be problematic because of wide incompatibility among different implementations.

To address the issue, more end-to-end platforms are being built—mostly as monolithic internal tools in big companies—but that often comes at the cost of a bottleneck at either the organizational or technological

level. ML platform teams may become gatekeepers for ML progress throughout the organization (i.e., when a research team devises a new algorithm that does not fit the mold of what the platform already supports, which makes putting the new algorithm into production extremely difficult). The ML platform can become a crystallization of outdated practices in the ever-changing ML landscape.

At the same time, AutoML systems promise to automate the human decision making involved in some parts of the process (in particular, in the pipeline building), and, through techniques such as hyperparameter optimization[10] and architecture search,[3] abstract the modeling part of the ML process.

AutoML is a promising direction, although research efforts are often centered around optimizing single steps of the pipeline (in particular, finding the model architecture) rather than optimizing the whole pipeline, and the costs of finding a marginally better solution than a commonly accepted one may end up outweighing the gains. In some instances this worsens the reusability issue and contrasts with recent findings showing how architecture may actually not be the most impactful aspect of a model, as opposed to its size, at least for autoregressive models trained on big enough data sets.[7] Despite this, the automation that AutoML brings is positive in general, as it allows developers to focus on what matters most and automate away more mundane and repetitive parts of the development process, thus reducing the number of decisions they have to make.

The intent of these platforms and AutoML systems to encapsulate best practices and simplify parts of the ML

process is much appreciated, but there could be a better, less monolithic way to think about ML platforms that may enable the advantages of these platforms, while drastically reducing their issues, and that can incorporate the advantages of AutoML at the same time.

Challenges and desiderata

Our experiences in developing both research and industrial ML projects and platforms led us to identify a set of challenges common to most of them, as well as some desired solutions, which influenced us to develop declarative ML systems.

- Challenge 1: *Exponential decision explosion.* Building an ML system involves many decisions, all of which need to be correct, with compounding errors at each stage.
- Desideratum 1: *Good defaults and automation.* The number of decisions should be reduced by nudging developers toward reasonable defaults and a repeatable automated process that makes those decisions (hyperparameter optimization, for example).
- Challenge 2: *New Model-itis.* ML production teams try to build a new model and fail at improving performance for lack of understanding the quality and failure modes of previous models.
- Desideratum 2: *Standardization and focus on quality.* Low-added-value parts of the ML process should be automated with standardized evaluation and data processing and automated model building and comparison, shifting the attention from writing low-level ML code to monitoring quality and improving supervision,

as well as shifting the attention from monodimensional performance-based model leaderboards toward holistic evaluation.

- Challenge 3: *Organizational chasms.* There are gaps between teams working in pipelines that make it hard to share code and ideas (i.e., when entity disambiguation and intent classification teams are different in a virtual assistant project and don't share the codebase, which leads to replication and technical debt).
- Desideratum 3: *Common interfaces.* Reusability can be increased by coming up with standard interfaces that favor modularity and interchangeability of implementations.
- Challenge 4: *Scarcity of expertise.* Not many developers, even in large companies, can write low-level ML code.
- Desideratum 4: *Higher-level abstractions.* Developers should not have to set hyperparameters manually or implement their custom model code unless truly necessary, as it accounts for just a tiny fraction of the project life cycle, and differences are usually tiny.
- Challenge 5: *Slow process.* The development of ML projects in some organizations can take months or years to reach a desired quality because of the many iterations required.
- Desideratum 5: *Rapid iteration.* The quality of ML projects improves by incorporating what has been learned from each iteration, so the faster each iteration is, the higher quality can be achieved in the same amount of time. The combination of automation and higher-level abstractions can improve the speed of iteration and in turn help improve quality.

- Challenge 6: *Many diverse stakeholders.* Many stakeholders are involved in the success of an ML project, with different skill sets and interests, but only a tiny fraction of them has the capability to work hands-on with the system.
- Desideratum 6: *Separation of interests.* Enforcing a separation of interests with multiple user views would make an ML system accessible to more people in the stack, allowing developers to focus on delivering value and improving the project outcome, and consumers to tap into the created value more easily.

DECLARATIVE ML SYSTEMS

A declarative ML system could fulfill the promise of addressing the above-mentioned challenges by implementing most of the desiderata. The term may be overloaded in the vast literature of ML models and systems, so here the definition of declarative ML systems is restricted to those systems that impose a separation between what an ML system should do and how it actually does it. The what part can be declared with a configuration that, depending on its complexity and compositionality, can be seen as a declarative language and can include information about the task to be solved by the ML system and the schema of the data it should be trained on. This can be considered a low-/no-/zero-code approach, as the declarative configuration is not an imperative language where the how is specified, so a user of a declarative ML system does not need to know how to implement an ML model or pipeline, just as someone who writes a SQL query doesn't need to know about database indices and query

planning. The declarative configuration is translated/compiled into a trainable ML pipeline that respects the provided schema, and the trained pipeline can then be used for obtaining predictions.

Many declarative ML approaches have been proposed over the years, most of which use either logic or probability theory or both as their main declarative interface. Some examples of such approaches include probabilistic graphical models[9] and their extensions to relational data such as probabilistic relational models[5,12] and Markov logic networks[2] or purely logical representations such as Prolog and Datalog. In these models, domain knowledge can be specified as dependencies between variables (and relations) representing the structure of the model and their strengths as free parameters. Both the free parameters and the structure can also be learned from data. These approaches are declarative in that they separate out the specification semantics from the inference algorithm.

Performing inference on such models, however, is in general difficult, and scalability becomes a major challenge. Approaches such as Tuffy,[14] DeepDive,[21] and others have been introduced to address the issue. Nevertheless, by separating inference from representation, these models do a good job of allowing declaration of multitask and highly joint models, but are often outperformed by more powerful feature-driven engines (e.g., deep-learning-based approaches). These declarative ML models are distinguished from systems based on their scope; the latter focus on defining declaratively an entire production ML pipeline.

Other potential higher-level abstractions hide the complexity of parts of the ML pipeline, and they have their own merits, but we do not consider them declarative ML systems. Examples of such other abstractions could be libraries that allow users (ML developers) to write simpler ML code by removing the burden of having to write neural network layer implementations (as Keras does) or having to write a for loop that is distributable and parallelizable (as PyTorch Lightning does). Other abstractions such as Caffe allow writing deep neural networks by declaring the layers of its architecture, but they do it at a level of granularity close to an imperative language. Finally, abstractions such as Thinc provide a robust configuration system for parametrizing models, but also require writing ML code that becomes parametrizable by the configuration system, thus not separating the *what* from the *how*.

Data first

Integrating data-mining and ML tools has been the focus of several major research and industrial efforts since at least the '90s. For example, Oracle's Data Miner, which shipped in 2001, featured high-level SQL-style syntax to use models and supported models defined externally in Java or via the PMML (Predictive Model Markup Language) standard. These models were effectively syntax around user-defined functions to perform filtering or inference. At the time, ML models were purpose-built using specialized solvers that required heavy use of linear algebra packages (e.g., L-BFGS was one of the most popular for ML models).

The ML community, however, began to realize that an extremely simple, classical algorithm called SGD

(stochastic gradient descent), or incremental gradient methods, could be used to train many important ML models. The Bismarck project[4] showed that SGD could piggyback on existing data-processing primitives that were already widely available in database systems (compare with SciDB, which rethought the entire database in terms of linear algebra).

In turn, integrating gradient descent and its variants allowed the database management system to manage training. This led to a new breed of systems that integrated training and inference. They provided SQL syntax extensions to train models in a declarative way to manage training and deployment inside the database. Examples of such systems are Bismarck, MADlib[6] (which was integrated in Impala, Oracle, Greenplum, etc.), and MLlib.[11] The SQL extensions proposed in Bismarck and MADlib are still popular, as variants of this approach are integrated in the modeling language of Google's BigQuery[17] or within modern open-source systems such as SQLFlow.[20]

The datacentric viewpoint has the advantage of making models usable from within the same environment where the data lives, avoiding potentially complicated data pipelines. One issue that emerges is that, by exposing model training as a primitive in SQL, users did not have fine-grained control of the modeling process. For some classes of models this became a substantial challenge, as the pain of piping the data to models (which these systems decreased substantially) was outweighed by the pain of performing featurization and tuning the model. As a result, many models lived outside the database.

Models first

After successes of deep-learning models in computer vision, speech recognition, and NLP (natural language processing), the focus of both research and industry shifted toward a model-first approach, where the training process was more complicated and became the main focus. A wrong implementation of backpropagation and differentiation would influence the performance of an ML project more than data preprocessing, and efficient computation of deep-learning algorithms on accelerated hardware such as GPUs transformed models that were too slow to train into the standard solution for certain ML problems, specifically perceptual ones. In practice, having an efficient wrapper of GPGPU (general-purpose GPU) libraries was more valuable than a generic data-preprocessing pipeline. Libraries such as TensorFlow and PyTorch focused on abstracting the intricacies of low-level C code for tensor computation.

The availability of these libraries allowed for simpler model building, so researchers and practitioners started sharing their models and adapting others' models to their goals. This process of transferring (pieces of) a pretrained model and tuning them on a new task started with word embeddings but was later adopted in computer vision, and now is made easier by libraries such as Hugging Face's Transformers.

Overton, built internally at Apple, and Ludwig, an open-source system at Uber, are both model-first and focus on modern deep-learning models, but they also retrieve some features of the data-first approach (specifically, the

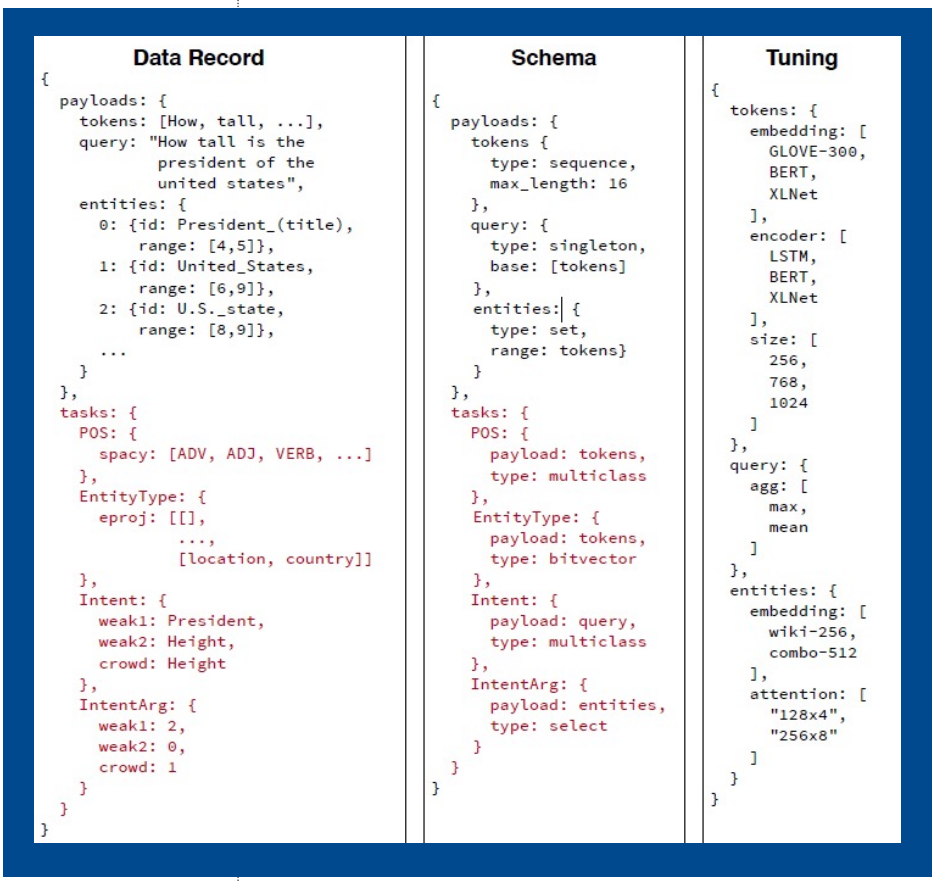
declarative nature) by adding separation of interest, and they are both capable of using transfer learning.

Overton in a nutshell

Overton[16] spawned from the same observations expressed at the beginning of this article: Commodity tools changed the landscape of ML to the point that tools capable of moving developers up the stack can be built, allowing users to focus on quality and quantity of supervision. Overton is designed to make sure that people do not need to write new models for production applications in search, information extraction, question answering, named entity disambiguation, and other tasks, while making it easy to evaluate models and improve performance by ingesting additional relevant data to get quality results on end-deployed models.

Inspired by relational databases, a user would declare a schema that describes the incoming data source called *payload*. In addition, a user would also describe a high-level data flow among the tasks, with optionally multiple sources of (weak) supervision, as shown in figure 3, which is an example of an Overton application to a complex NLP task. On the left is an example data record of a piece of text, with its payload (inputs, query, tokenization, and candidate entities) and tasks (output, parts of speech, entity type, intent, and intent arguments); in the middle is the Overton schema, detailing both payloads for the input and tasks for the output, with their respective types and parameters; on the right is a tuning specification that details the coarse-grained architecture options from which Overton will choose and compare for each payload.

FIGURE 3: **EXAMPLE OVERTON APPLICATION OF A COMPLEX NLP TASK**



The system is able to use this barebones information to compile trainable models (including data preprocessing and symbol mappings); combine supervision using data-programming techniques;[15] compile a model in TensorFlow, PyTorch, or Core ML; produce performance reports; and

finally export a deployable model in Core ML, TensorFlow, or ONNX (Open Neural Network Exchange).

A key technical idea is that many subproblems such as architecture search or hyperparameter optimization could be done with simple methods, such as coarse-grained architecture search (only classes of architectures are chosen, not all their internal hyperparameters) or very simple grid search. A user could override some of these decisions, but custom options are not heavily optimized in runtime. Other features include multitask learning, data slicing, and the use of pretrained models.

The role of Overton users becomes monitoring performance, improving supervision quality by adding new examples and providing new forms of supervision; they don't need to write models in low-level ML code. Overton is responsible for massive gains in quality (40 to 82 percent error reduction) in search and question-answering applications at Apple; as a consequence, the footprint of the engineering team is substantially reduced, and no one is writing low-level ML code.

Ludwig in a nutshell

Ludwig[13] is a system that allows its users to build end-to-end deep-learning pipelines through a declarative configuration, train them, and use them for obtaining predictions. The pipelines include data preprocessing that transforms raw data into tensors, model-architecture building, training loop, prediction, postprocessing of data, and evaluation of pipelines. Ludwig also includes a visualization module for model-performance analysis and comparison, and a declarative hyperparameter-

optimization module.

One key idea of Ludwig is that it abstracts both the data schema and tasks as data-type feature interfaces so that users need to define only a list of input and output features, both with their names and data types. This allows for modularity and extensibility: the same text-preprocessing code and the same text-encoding architecture code are reused every time a model that includes text features is instantiated, while, for example, the same multilabel classification code for prediction and evaluation is adopted every time a set feature is specified as an output.

This flexibility and abstraction are made possible because Ludwig is opinionated about the structure of the deep-learning models it builds, following the ECD (encoder-combiner-decoder) architecture introduced by Molino et al.,[13] which allows for easily defining multimodal and multitask models, depending on the data type of both the input and output available in the training data. The ECD architecture also defines precise interfaces, which greatly improve code reuse and extensibility: By imposing the dimensions of the input and output tensors of an image encoder, for example, the architecture allows for many interchangeable implementations of image encoding (i.e., a convolutional neural network stack, a stack of residual blocks, or a stack of transformer layers), and choosing which one to use in the configuration requires changing just one string parameter.

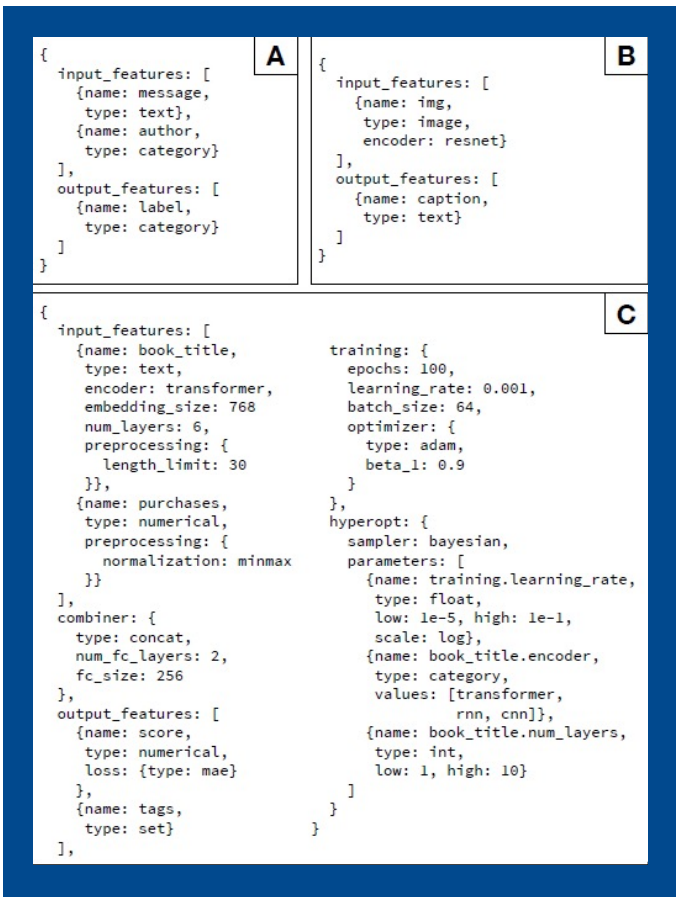
What makes Ludwig general is that, depending on the combination of types of input and output declared in the configuration, the specific model instantiated from

the ECD architecture solves a different task: Input text and output category will make Ludwig compile a text classification architecture, while an image input and a text output will result in an image-captioning system, and both image and text inputs with a text output will result in a visual question-answering model. Moreover, basic Ludwig configurations are easy to write and hide most of the complexity of building a deep-learning model, but at the same time they allow the user to specify all details of the architecture, training loop, and preprocessing if they so desire.

Figure 4 shows three examples of Ludwig configurations: (A) a simple text classifier that includes additional structured information about the author of the classified message; (B) image-captioning example; (C) a detailed configuration for a model that, given the title and sales figures of a book, predicts its user score and tags. A and B show simple configurations, while C shows the degree of control of each encoder, combiner, and decoder, together with training and preprocessing parameters, while also highlighting how Ludwig supports hyperparameter optimization of every possible configuration parameter. The declarative hyperopt section shown in figure 4(C) makes it possible to automate architectural, training, and preprocessing decisions.

In the end, Ludwig is both a modular and an end-to-end system: The internals are highly modular for allowing Ludwig developers to add options, improve the existing ones, and reuse code, but from the perspective of the Ludwig user, it's entirely end to end (including processing, training, hyperopt, and evaluation).

FIGURE 4: **THREE EXAMPLES OF LUDWIG CONFIGURATIONS**



Similarities and differences

Both Overton and Ludwig, despite being developed entirely independently of each other, converged on similar design decisions—in particular, on the adoption of declarative

configurations that include (albeit with a different syntax) both the input data schema and a notion of the tasks models should solve in Overton and the analogous notion of input and output features in Ludwig. Both systems have a notion of types associated with the data, which inform parts of the pipelines they build.

Where the two systems differ is in some assumptions, some capabilities, and their focus. Overton is more concerned with being able to compile its models in various formats—in particular, for deployments—while Ludwig has only one productionization route. Overton also allows for a more explicit way to define data-related aspects such as weak supervision and data slicing. Ludwig, on the other hand, covers a wider breadth of use cases by virtue of the compositionality of the ECD architecture, where different combinations of input and output can define different ML tasks.

Despite the differences, both systems address some of the challenges highlighted previously in this article. Both systems nudge developers toward making fewer decisions by automating part of the life cycle (desideratum 1) and toward reusing models already available to them and analyzing them thoroughly by providing both standard implementation of architectures and evaluations that can also be combined in a more holistic way (desideratum 2).

The interfaces and the use of data types and associated higher-level abstractions (desideratum 4) in both systems favor code reuse (desideratum 3) and address the expertise scarcity. Declarative configurations increase the speed of model iteration (desideratum 5), as developers just need to change details in the declaration instead of

rewriting code with cascade effects. Both systems also partially provide separation of interests (desideratum 6): They separate between the system developers adding new models, types, and features and the users using the declarative interface.

WHAT IS NEXT?

The adoption of both Overton and Ludwig in real-world scenarios by tech companies suggests that they are actually solving at least some of the concrete problems those companies face. There is substantially more value to be untapped by combining their strengths with the tighter integration with data of the data-first era of declarative ML systems. This new wave of recent deep-learning work has shown that with relatively simple building blocks and AutoML, fine control of the training and tuning process may no longer be necessary, thus solving the main pain point that data-first approaches did not address and opening the door for a convergence toward new, higher-level systems that seamlessly integrate model training, inference, and data.

In this regard, lessons can be learned from computing history, by observing the process that led to the emergence of general systems that replaced bespoke solutions:

- *Number of users.* Even higher-level abstractions are needed for ML not only to become more widely adopted, but also to be developed, trained, improved, and used by people without any coding skills. To draw another analogy with database systems, we are still in the COBOL era of ML; just as SQL allowed a substantially larger amount of

people to write database application code, the same will happen for ML.

- *Explicit user roles.* Not everyone interacting with a future ML system will be trained in ML, statistics, or even computer science. Just as databases evolved to the point that there's a stark separation between database developers implementing faster algorithms, database admins managing instances installation and configuration, database users writing application code, and final users obtaining fast answers to their requests, this role separation is expected to emerge in ML systems.
- *Performance optimizations.* More abstract systems tend to make compromises either in terms of expressiveness or performance. Ludwig achieving state of the art and Overton replacing production systems suggest that may be a false tradeoff already. The history of compilers suggests a similar pattern: Over time optimized compilers could often beat hand-tuned machine-code kernels, although the complexity of the task may have suggested otherwise initially. Developments in this direction will lead to bespoke solutions that will likely be limited to highly specific tasks in the fat part of the (growing) long tail of ML tasks within an organization, where even minor improvements are valuable, similar to the mission-critical use cases where today one may want to write assembly code.
- *Symbiotic relationship between systems and libraries.* There will likely be more ML libraries in the future, and they will co-exist with and help improve ML systems in a virtuous cycle. In the history of computing this has happened over and over; a recent example is the emergence of full-text indexing libraries such as Apache

Lucene filling the feature gap that most DBMSes had at the time, with Lucene being used in bespoke applications first, and later being used as the foundation for complete search systems such as Elasticsearch and Apache Solr, and finally being integrated in DBMSes such as OrientDB, GraphDB, and others. Some challenges are still open for declarative ML systems: They will have to demonstrate they are robust with respect to future changes in machine learning coming from research, supporting diverse training regimens, and showing that the types of tasks they can represent encompass a large fraction of practical uses. The jury is still out on this.

Technologies change the world when they can be harnessed by more people than those who can build them, so we believe the future of machine learning and its impact on everyone's life ultimately depends on the effort of putting it in the hands of the rest of us.

Acknowledgments

The authors want to thank Antonio Vergari, Karan Goel, Sahaana Suri, Chip Huyen, Dan Fu, Arun Kumar, and Michael Cafarella for insightful comments and suggestions.

References

1. Casado, M., Bornstein, M. 2020. The new business of AI (and how it's different from traditional software). Andreessen Horowitz; https://a16z.com/2020/02/16/the-new-business-of-ai-and-how-its-different-from-traditional-software/.
2. Domingos, P. M. 2004. Real-world learning with Markov logic networks. In *Proceedings of the 15th European*

Conference on Machine Learning 17; https://dl.acm.org/doi/10.1007/978-3-540-30115-8_4.

3. Elsken, T., Metzen, J. H., Hutter, F. 2019. Neural architecture search: a survey. *Journal of Machine Learning Research* 20, 1-21; https://www.jmlr.org/papers/volume20/18-598/18-598.pdf.
4. Feng, X., Kumar, A., Recht, B., Ré, C. 2012. Towards a unified architecture for in-RDBMS analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data,* 325-336; https://dl.acm.org/doi/10.1145/2213836.2213874.
5. Friedman, N., Getoor, L., Koller, D., Pfeffer, A. 1999. Learning probabilistic relational models. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, 1300-1307; https://dl.acm.org/doi/10.5555/1624312.1624404.
6. Hellerstein, J. M., Ré, C., Schoppmann, F., Wang, D. Z., Fratkin, E., Gorajek, A., Ng, K. S., Welton, C., Feng, X., Li, K., Kumar, A. 2012. The MADlib analytics library: or MAD skills, the SQL. In *Proceedings of the Very Large Data Base Endowment* 5(12), 1700-1711; https://dl.acm.org/doi/10.14778/2367502.2367510.
7. Henighan, T., Kaplan, J., Katz, M., Chen, M., Hesse, C., Jackson, J., Jun, H., Brown, T. B., Dhariwal, P., Gray, S., Hallacy, C., Mann, B., Radford, A., Ramesh, A., Ryder, N., Ziegler, D. M., Schulman, J., Amodei, D., McCandlish, S. 2020. Scaling laws for autoregressive generative modeling. arXiv; https://arxiv.org/abs/2010.14701.
8. Hooker, S. 2020. The hardware lottery. arXiv; https://arxiv.org/abs/2009.06489.
9. Koller, D., Friedman, N. 2009. *Probabilistic Graphical*

Models: Principles and Techniques. Adaptive Computation and Machine Learning series. MIT Press; https://mitpress.mit.edu/books/probabilistic-graphical-models.

10. Li, L., Jamieson, K. G., DeSalvo, G., Rostamizadeh, A., Talwalkar, A. 2017. Hyperband: a novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research* 18 (1), 6765-6816; https://dl.acm.org/doi/abs/10.5555/3122009.3242042.
11. Meng, X., Bradley, J. K., Yavuz, B., Sparks, E. R., Venkataraman, S., Liu, D., Freeman, J., Tsai, D. B., Amde, M., Owen, S., Xin, D., Xin, R., Franklin, M. J., Zadeh, R., Zaharia, M., Talwalkar, A. 2016. MLlib: machine learning in Apache Spark. *Journal of Machine Learning Research* 17(1), 1235-1241; https://dl.acm.org/doi/10.5555/2946645.2946679.
12. Milch, B., Marthi, B., Russell, S. J., Sontag, D. A., Ong, D. L., Kolobov, A. 2005. BLOG: probabilistic models with unknown objects. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence,* Edinburgh, Scotland, ed. L. P. Kaelbling and A. Saffiotti, 1352-1359. Professional Book Center; https://researchr.org/publication/MilchMRSOK05.
13. Molino, P., Yaroslav Dudin, Y., Miryala, S. S. 2019. Ludwig: a type-based declarative deep learning toolbox. arXiv; https://arxiv.org/abs/1909.07930.
14. Niu, F., Ré, C., Doan, A., Shavlik, J. W. 2011. Tuffy: Scaling up statistical inference in Markov logic networks using an RDBMS. In *Proceedings of the Very Large Data Base Endowment* 4(6), 373-384; https://dl.acm.org/doi/10.14778/1978665.1978669.

15. Ratner, A. J., De Sa, C., Wu, S., Selsam, D., Ré, C. 2016. Data programming: creating large training sets, quickly. In *Proceedings of the 30th International Conference on Neural Information Processing Systems,* 3574-3582; https://dl.acm.org/doi/10.5555/3157382.3157497.
16. Ré, C. 2020. Overton: a data system for monitoring and improving machine-learned products. In 10th Annual Conference on Innovative Data Systems Research; http://cidrdb.org/cidr2020/papers/p33-re-cidr20.pdf.
17. Sato. K. 2012. An inside look at Google BigQuery. Google White Paper; https://cloud.google.com/files/BigQueryTechnicalWP.pdf.
18. Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., Dennison, D. 2015. Hidden technical debt in machine learning systems. In *Proceedings of the 28th International Conference on Neural Information Processing Systems* 2, 2503-2511; https://dl.acm.org/doi/10.5555/2969442.2969519.
19. Senior, A. W., Evans, R., Jumper, J., Kirkpatrick, J., Sifre, L., Green, T., Qin, C., Žídek, A., Nelson, A. W. R., Bridgland, A., Penedones, H., Petersen, S., Simonyan, K., Crossan, S., Kohli, P., Jones, D. T., Silver, D., Kavukcuoglu, K., Hassabis, D. 2020. Improved protein structure prediction using potentials from deep learning. *Nature* 577, 706-710; https://www.nature.com/articles/s41586-019-1923-7.
20. Wang, Y., Yang, Y., Zhu, W., Wu,Y., Yan, X., Liu, Y., Wang, Y., Xie, L., Gao, Z., Zhu, W., Chen, X., Yan, W., Tang, M., Tang, Y. 2020. SQLflow: a bridge between SQL and machine learning. arXiv; https://arxiv.org/abs/2001.06846.
21. Zhang, C., Ré, C., Cafarella, M. J., Shin, J., Wang, F.,

Wu, S. 2017. DeepDive: declarative knowledge base construction. *Communications of the* ACM 60(5), 93-102; https://dl.acm.org/doi/10.1145/3060586.

Piero Molino *is a Staff Research Scientist at Stanford University working on Machine Learning systems and algorithms. Piero completed a PhD on Question Answering at the University of Bari, Italy. Worked for Yahoo Labs in Barcelona on learning to rank, IBM Watson in New York on natural language processing with deep learning and then joined Geometric Intelligence, where he worked on grounded language understanding. After Uber acquired Geometric Intelligence, he became one of the founding members of Uber AI Labs. At Uber he worked on research topics including Dialogue Systems, Language Generation, Graph Representation Learning, Computer Vision, Reinforcement Learning and Meta Learning. He also worked on several deployed systems like COTA, an ML and NLP model for Customer Support, Dialogue Systems for driver hands free dispatch, on the Uber Eats Recommender System with graph learning, and on collusion detection. He is the author of Ludwig, a Linux-Foundation-backed opensource declarative deep learning toolbox.*

Christopher (Chris) Ré *is an associate professor in the Department of Computer Science at Stanford University. He is in the Stanford AI Lab and is affiliated with the Statistical Machine Learning Group. His recent work is to understand how software and hardware systems will change as a result of machine learning along with a continuing, petulant drive*

to work on math problems. Research from his group has been incorporated into scientific and humanitarian efforts, such as the fight against human trafficking, along with widely used products from technology and enterprise companies including Google Ads, GMail, YouTube, and Apple. He has cofounded four companies based on his research into machine learning systems, SambaNova and Snorkel, along with two companies that are now part of Apple, Lattice (DeepDive) in 2017 and Inductiv (HoloClean) in 2020. His research contributions have spanned database theory, database systems, and machine learning. His work has won best paper or test-of-time awards at the premier venues in each area. He still can't believe he won the MacArthur Foundation Fellowship.