Overton: A Data System for Monitoring and Improving Machine-Learned Products

Christopher Ré   Feng Niu   Pallavi Gudipati   Charles Srisuwananukorn
Apple   Apple   Apple   Apple

September 13, 2019

Abstract

We describe a system called Overton, whose main design goal is to support engineers in building, monitoring, and improving production machine learning systems. Key challenges engineers face are monitoring fine-grained quality, diagnosing errors in sophisticated applications, and handling contradictory or incomplete supervision data. Overton automates the life cycle of model construction, deployment, and monitoring by providing a set of novel high-level, declarative abstractions. Overton's vision is to shift developers to these higher-level tasks instead of lower-level machine learning tasks. In fact, using Overton, engineers can build deep-learning-based applications without writing any code in frameworks like TensorFlow. For over a year, Overton has been used in production to support multiple applications in both near-real-time applications and back-of-house processing. In that time, Overton-based applications have answered billions of queries in multiple languages and processed trillions of records reducing errors $1.7 - 2.9\times$ versus production systems.

1   Introduction

In the life cycle of many production machine-learning applications, maintaining and improving deployed models is the dominant factor in their total cost and effectiveness–much greater than the cost of *de novo* model construction. Yet, there is little tooling for model life-cycle support. For such applications, a key task for supporting engineers is to improve and maintain the quality in the face of changes to the input distribution and new production features. This work describes a new style of data management system called Overton that provides abstractions to support the model life cycle by helping build models, manage supervision, and monitor application quality.[1]

Overton is used in both near-real-time and backend production applications. However, for concreteness, our running example is a product that answers factoid queries, such as *"how tall is the president of the united states?"* In our experience, the engineers who maintain such machine learning products face several challenges on which they spend the bulk of their time.

- **Fine-grained Quality Monitoring** While overall improvements to quality scores are important, often the week-to-week battle is improving fine-grained quality for important subsets of the input data. An individual subset may be rare but are nonetheless important, e.g., 0.1% of queries may correspond to a product feature that appears in an advertisement and so has an outsized importance. Traditional machine learning approaches effectively optimize

---

[1]The name Overton is a nod to the *Overton Window*, a concept from political theory that describes the set of acceptable ideas in public discourse. A corollary of this belief is that if one wishes to move the "center" of current discourse, one must advocate for a radical approaches outside the current window. Here, our radical approach was to focus only on programming by supervision and to prevent ML engineers from using ML toolkits like TensorFlow or manually selecting deep learning architectures.

for aggregate quality. As hundreds of such subsets are common in production applications, this presents data management and modeling challenges. An ideal system would monitor these subsets and provide tools to improve these subsets while maintaining overall quality.

- **Support for Multi-component Pipelines** Even simple machine learning products comprise myriad individual tasks. Answering even a simple factoid query, such as *"how tall is the president of the united states?"* requires tackling many tasks including (1) find the named entities ('united states', and 'president'), (2) find the database ids for named entities, (3) find the intent of the question, e.g., the height of the topic entity, (4) determine the topic entity, e.g., neither president nor united states, but the person Donald J. Trump, who is not explicitly mentioned, and (5) decide the appropriate UI to render it on a particular device. Any of these tasks can go wrong. Traditionally, systems are constructed as pipelines, and so determining which task is the culprit is challenging.

- **Updating Supervision** When new features are created or quality bugs are identified, engineers provide additional supervision. Traditionally, supervision is provided by annotators (of varying skill levels), but increasingly *programmatic supervision* is the dominant form of supervision [12, 23], which includes labeling, data augmentation, and creating synthetic data. For both privacy and cost reasons, many applications are constructed using programmatic supervision as a primary source. An ideal system can accept supervision at multiple granularities and resolve conflicting supervision for those tasks.

There are other desiderata for such a system, but the commodity machine learning stack has evolved to support them: building deployment models, hyperparameter tuning, and simple model search are now well supported by commodity packages including TensorFlow, containers, and (private or public) cloud infrastructure.[2] By combining these new systems, Overton is able to automate many of the traditional modeling choices, including deep learning architecture, its hyperparameters, and even which embeddings are used.

Overton provides the engineer with abstractions that allow them to build, maintain, and monitor their application by manipulating data files–not custom code. Inspired by relational systems, supervision (data) is managed separately from the model (schema). Akin to traditional *logical independence*, Overton's schema provides *model independence*: serving code does not change even when inputs, parameters, or resources of the model change. The schema changes very infrequently–many production services have not updated their schema in over a year.

Overton takes as input a *schema* whose design goal is to support rich applications from modeling to automatic deployment. In more detail, the *schema* has two elements: (1) *data payloads* similar to a relational schema, which describe the input data, and (2) *model tasks*, which describe the tasks that need to be accomplished. The schema defines the input, output, and coarse-grained data flow of a deep learning model. Informally, *the schema defines what the model computes but not how the model computes it*: Overton does not prescribe architectural details of the underlying model (e.g., Overton is free to embed sentences using an LSTM or a Transformer) or hyperparameters, like hidden state size. Additionally, sources of supervision are described as data–not in the schema–so they are free to rapidly evolve.

As shown in Figure 1, given a schema and a data file, Overton is responsible to instantiate and train a model, combine supervision, select the model's hyperparameters, and produce a production-ready binary. Overton compiles the schema into a (parameterized) TensorFlow or PyTorch program, and performs an architecture and hyperparameter search. A benefit of this compilation approach is that Overton can use standard toolkits to monitor training (TensorBoard equivalents) and to meet service-level agreements (Profilers). The models and metadata are written to an S3-like data store that is accessible from the production infrastructure. This has enabled model retraining and deployment to be nearly automatic, allowing teams to ship products more quickly.

In retrospect, the following three choices of Overton were the most important in meeting the above challenges.

(1) Code-free Deep Learning In Overton-based systems, engineers focus exclusively on fine-grained monitoring of their application quality and improving supervision–not tweaking deep learning models. An Overton engineer does

---

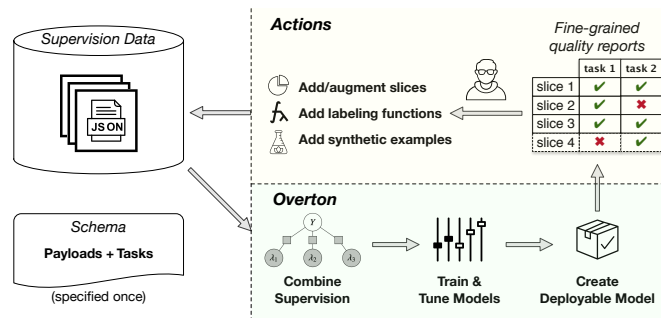[2]Overton builds on systems like Turi [1].

Figure 1: Schema and supervision data are input to Overton, which outputs a deployable model. Engineers monitor and improve the model via supervision data.

not write any deep learning code in frameworks like TensorFlow. To support application quality improvement, we use a technique, called *model slicing* [3]. The main idea is to allow the developer to identify fine-grained subsets of the input that are important to the product, e.g., queries about nutrition or queries that require sophisticated disambiguation. The system uses developer-defined slices as a guide to increase representation capacity. Using this recently developed technique led to state-of-the-art results on natural language benchmarks including GLUE and SuperGLUE [31].[3]

(2) Multitask Learning Overton was built to natively support multitask learning [2,24,26] so that all model tasks are concurrently predicted. A key benefit is that Overton can accept supervision at whatever granularity (for whatever task) is available. Overton models often perform ancillary tasks like part-of-speech tagging or typing. Intuitively, if a representation has captured the semantics of a query, then it should reliably perform these ancillary tasks. Typically, ancillary tasks are also chosen either to be inexpensive to supervise. Ancillary task also allow developers to gain confidence in the model's predictions and have proved to be helpful for aids for debugging errors.
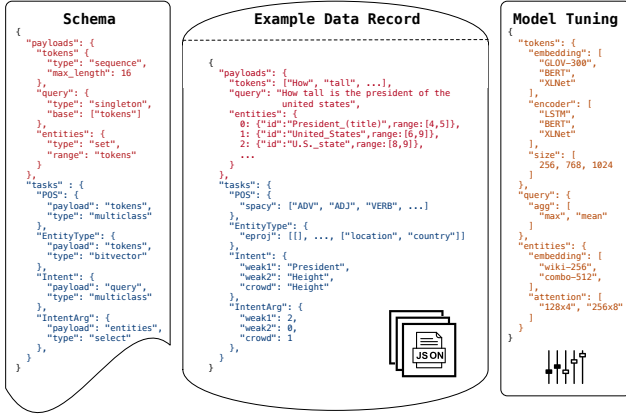
(3) Weak Supervision Applications have access to supervision of varying quality and combining this contradictory and incomplete supervision is a major challenge. Overton uses techniques from Snorkel [23] and Google's Snorkel DryBell [12], which have studied how to combine supervision in theory and in software. Here, we describe two novel observations from building production applications: (1) we describe the shift to applications which are constructed almost *entirely* with weakly supervised data due to cost, privacy, and cold-start issues, and (2) we observe that weak supervision may obviate the need for popular methods like transfer learning from massive pretrained models, e.g., BERT [8]–on some production workloads, which suggests that a deeper trade-off study may be illuminating.

In summary, Overton represents a first-of-its kind machine-learning lifecycle management system that has a focus on monitoring and improving application quality. A key idea is to separate the model and data, which is enabled by a code-free approach to deep learning. Overton repurposes ideas from the database community and the machine learning community to help engineers in supporting the lifecycle of machine learning toolkits. This design is informed and refined from use in production systems for over a year in multiple machine-learned products.
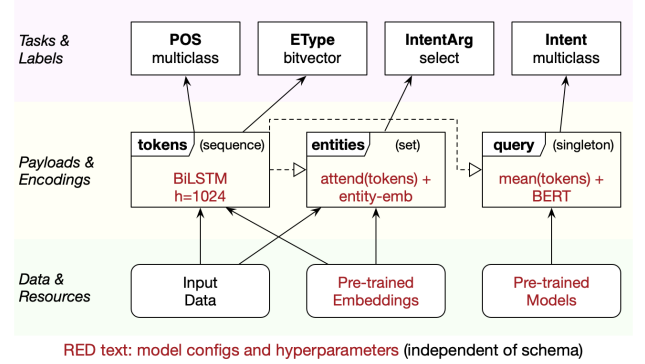
2 An Overview of Overton

To describe the components of Overton, we continue our running example of a factoid answering product. Given the textual version of a query, e.g., *"how tall is the president of the united states"*, the goal of the system is to appropriately render the answer to the query. The main job of an engineer is to measure and improve the quality of the system across many queries, and a key capability Overton needs to support is to measure the quality in several fine-grained ways. This

---

[3]A blog post introduction to slicing is here `https://snorkel.org/superglue.html`.

Schema
```
{
  "payloads": {
    "tokens": {
      "type": "sequence",
      "max_length": 16
    },
    "query": {
      "type": "singleton",
      "base": ["tokens"]
    },
    "entities": {
      "type": "set",
      "range": "tokens"
    }
  },
  "tasks" : {
    "POS": {
      "payload": "tokens",
      "type": "multiclass"
    },
    "EntityType": {
      "payload": "tokens",
      "type": "bitvector"
    },
    "Intent": {
      "payload": "query",
      "type": "multiclass"
    },
    "IntentArg": {
      "payload": "entities",
      "type": "select"
    }
  }
}
```

Example Data Record
```
{
  "payloads": {
    "tokens": ["How", "tall", ...],
    "query": "How tall is the president of the
               united states",
    "entities": {
      0: {"id":"President_(title)",range:[4,5]},
      1: {"id":"United_States",range:[6,9]},
      2: {"id":"U.S._state",range:[8,9]},
      ...
    }
  },
  "tasks": {
    "POS": {
      "spacy": ["ADV", "ADJ", "VERB", ...]
    },
    "EntityType": {
      "eproj": [[], ..., ["location", "country"]]
    },
    "Intent": {
      "weak1": "President",
      "weak2": "Height",
      "crowd": "Height"
    },
    "IntentArg": {
      "weak1": 2,
      "weak2": 0,
      "crowd": 1
    }
  }
}
```

Model Tuning
```
{
  "tokens": {
    "embedding": [
      "GLOV-300",
      "BERT",
      "XLNet"
    ],
    "encoder": [
      "LSTM",
      "BERT",
      "XLNet"
    ],
    "size": [
      256, 768, 1024
    ]
  },
  "query": {
    "agg": [
      "max", "mean"
    ]
  },
  "entities": {
    "embedding": [
      "wiki-256",
      "combo-512",
    ],
    "attention": [
      "128x4", "256x8"
    ]
  }
}
```

(a) An example schema, data file, and tuning specification for our running example. Colors group logical sections.

Tasks & Labels: POS multiclass | EType bitvector | IntentArg select | Intent multiclass

Payloads & Encodings: tokens (sequence) BiLSTM h=1024 | entities (set) attend(tokens) + entity-emb | query (singleton) mean(tokens) + BERT

Data & Resources: Input Data | Pre-trained Embeddings | Pre-trained Models

RED text: model configs and hyperparameters (independent of schema)

(b) A deep architecture that might be selected by Overton. The red components are selected by Overton via model search; the black boxes and connections are defined by the schema.

Figure 2: The inputs to Overton and a schematic view of a compiled model.

quality is measured within Overton by evaluation on curated test sets, which are fastidiously maintained and improved by annotators and engineers. An engineer may be responsible for improving performance on a specific subset of the data, which they would like to monitor and improve.

There are two inputs to Overton (Figure 2a): The schema (Section 2.1), which specifies the tasks, and a data file, which is the primary way an engineer refines quality (Section 2.2). Overton then compiles these inputs into a multitask deep model (Figure 2b). We describe an engineer's interaction with Overton (Section 2.3) and discuss design decisions (Section 2.4).

2.1 Overton's Schema

An Overton schema has two components: the *tasks*, which capture the tasks the model needs to accomplish, and *payloads*, which represent sources of data, such as tokens or entity embeddings. Every example in the data file conforms to this schema. Overton uses a schema both as a guide to compile a TensorFlow model and to describe its output for downstream use.[4] Although Overton supports more types of tasks, we focus on classification tasks for simplicity. An example schema and its corresponding data file are shown in Figure 2a. The schema file also provides schema information in a traditional database sense: it is used to define a memory-mapped row-store for example.[5]

A key design decision is that the schema does not contain information about hyperparameters like hidden state sizes. This enables *model independence*: the same schema is used in many downstream applications and even across different languages. Indeed, the same schema is shared in multiple locales and applications, only the supervision differs.

Payloads Conceptually, Overton embeds raw data into a payload, which is then used as input to a task or to another payload. Overton supports payloads that are singletons (e.g., a query), sequences (e.g. a query tokenized into words or characters), and sets (e.g., a set of candidate entities). Overton's responsibility is to embed these payloads into tensors of the correct size, e.g., a query is embedded to some dimension d, while a sentence may be embedded into an array of

---

[4]Overton also supports PyTorch and CoreML backends. For brevity, we describe only TensorFlow.

[5]Since all elements of an example are needed together, a row store has obvious IO benefits over column-store-like solutions.

size $m \times d$ for some length m. The mapping from inputs can be learned from scratch, pretrained, or fine-tuned; this allows Overton to incorporate information from a variety of different sources in a uniform way.

Payloads may refer directly to a data field in a record for input, e.g., a field 'tokens' contains a tokenized version of the query. Payloads may also refer to the contents of another payload. For example, a query payload may aggregate the representation of all tokens in the query. A second example is that an entity payload may refer to its corresponding span of text, e.g., the "united states of america" entity points to the span "united states" in the query. Payloads may aggregate several sources of information by referring to a combination of source data and other payloads. The payloads simply indicate dataflow, Overton learns the semantics of these references.[6]

Tasks Continuing our running example in Figure 2b, we see four tasks that refer to three different payloads. For each payload type, Overton defines a multiclass and a bitvector classification task. In our example, we have a multiclass model for the INTENT task: it assigns one label for each query payload, e.g., the query is about "height". In contrast, in the ENTITYTYPE task, fine-grained types for each token are not modeled as exclusive, e.g., location and country are not exclusive. Thus, the ENTITYTYPE task takes the token payloads as input, and emits a bitvector for each token as output. Overton also supports a task of selecting one out of a set, e.g., INTENTARG selects one of the candidate entities. This information allows Overton to compile the inference code and the loss functions for each task and to build a *serving signature*, which contains detailed information of the types and can be consumed by model serving infrastructure. At the level of TensorFlow, Overton takes the embedding of the payload as input, and builds an output prediction and loss function of the appropriate type.

The schema is changed infrequently, and many engineers who use Overton simply select an existing schema. Applications are customized by providing supervision in a data file that conforms to the schema, described next.

2.2 Weak Supervision and Slices

The second main input to Overton is the data file. It is specified as (conceptually) a single file: the file is meant to be engineer readable and queryable (say using jq), and each line is a single JSON record. For readability, we have pretty-printed a data record in Figure 2a. Each payload is described in the file (but may be null).

The supervision is described under each task, e.g., there are three (conflicting) sources for the INTENT task. A task requires labels at the appropriate granularity (singleton, sequence, or set) and type (multiclass or bitvector). The labels are tagged by the source that produced them: these labels may be incomplete and even contradictory. Overton models the sources of these labels, which may come human annotators, or from engineer-defined heuristics such as data augmentation or heuristic labelers. Overton learns the accuracy of these sources using ideas from the Snorkel project [23]. In particular, it estimates the accuracy of these sources and then uses these accuracies to compute a probability that each training point is correct [29]. Overton incorporates this information into the loss function for a task; this also allows Overton to automatically handle common issues like rebalancing classes. *(handwritten)* ② Reinforcement? in ML?

Monitoring For monitoring, Overton allows engineers to provide user-defined *tags* that are associated with individual data points. The system additionally defines default tags including *train*, *test*, *dev* to define the portion of the data that should be used for training, testing, and development. Engineers are free to define their own subsets of data via tags, e.g,. the date supervision was introduced, or by what method. Overton allows report per-tag monitoring, such as the accuracy, precision and recall, or confusion matrices, as appropriate. These tags are stored in a format that is compatible with Pandas. As a result, engineers can load these tags and the underlying examples into other downstream analysis tools for further analytics.

---

[6] By default, combination is done with multi-headed attention. The method of aggregation is not specified and Overton is free to change it.

Slicing   In addition to tags, Overton defines a mechanism called *slicing*, that allows monitoring but also adds representational capacity to the model. An engineer defines a slice by tagging a subset of the data and indicating that this tag is also a slice. Engineers typically define slices that consist of a subset that is particular relevant for their job. For example, they may define a slice because it contains related content, e.g., *"nutrition-related queries"* or because the subset has an interesting product feature, e.g., *"queries with complex disambiguation"*. The engineer interacts with Overton by identifying these slices, and providing supervision for examples in those slices.[7] Overton reports the accuracy conditioned on an example being in the slice. The main job of the engineer is to diagnose what kind of supervision would improve a slice, and refine the labels in that slice by correcting labels or adding in new labels.

A slice also indicates to Overton that it should increase its representation capacity (slightly) to learn a *"per slice"* representation for a task.[8] In this sense, a slice is akin to defining a "micro-task" that performs the task just on the subset defined by the slice. Intuitively, this slice should be able to better predict as the data in a slice typically has less variability than the overall data. At inference time, Overton makes only one prediction per task, and so the first challenge is that Overton needs to combine these overlapping slice-specific predictions into a single prediction. A second challenge is that slices heuristically (and so imperfectly) define subsets of data. To improve the coverage of these slices, Overton learns a representation of when one is *"in the slice"* which allows a slice to generalize to new examples. Per-slice performance is often valuable to an engineer, even if it does not improve the overall quality, since their job is to improve and monitor a particular slice. A production system improved its performance on a slice of complex but rare disambiguations by over 50 points of F1 using the same training data.

2.3   A Day in the Life of an Overton Engineer

To help the reader understand the process of an engineer, we describe two common use cases: improving an existing feature, and the cold-start case. Overton's key ideas are changing where developers spend their time in this process.

Improving an Existing Feature   A first common use case is that an engineer wants to improve the performance of an existing feature in their application. The developer iteratively examines logs of the existing application. To support this use case, there are downstream tools that allow one to quickly define and iterate on subsets of data. Engineers may identify areas of the data that require more supervision from annotators, conflicting information in the existing training set, or the need to create new examples through weak supervision or data augmentation. Over time, systems have grown on top of Overton that support each of these operations with a more convenient UI. An engineer using Overton may simply work entirely in these UIs.

Cold-start Use Case   A second common use case is the cold-start use case. In this case, a developer wants to launch a new product feature. Here, there is no existing data, and they may need to develop synthetic data. In both cases, the identification and creation of the subset is done by tools outside of Overton. These subsets become the aforementioned slices, and the different mechanisms are identified as different sources. Overton supports this process by allowing engineers to tag the lineage of these newly created queries, measure their quality in a fine-grained way, and merge data sources of different quality.

In previous iterations, engineers would modify loss functions by hand or create new separate models for each case. Overton engineers spend no time on these activities.

---

[7]Downstream systems have been developed to manage the slicing and programmatic supervision from the UI perspective that are managed by independent teams.

[8]We only describe its impact on systems architecture, the machine learning details are described in Chen et al. [3].

2.4 Major Design Decisions and Lessons

We briefly cover some of the design decisions in Overton.

Design for Weakly Supervised Code As described, weakly supervised machine learning is often the dominant source of supervision in many machine learning products. Overton uses ideas from Snorkel [23] and Google's Snorkel Drybell [12] to model the quality of the supervision. The design is simple: lineage is tracked for each source of information. There are production systems with *no* traditional supervised training data (but they do have such data for validation). This is important in privacy-conscious applications.

Modeling to Deployment In many production teams, a deployment team is distinct from the modeling team, and the deployment team tunes models for production. However, we noticed quality regressions as deployment teams have an incomplete view of the potential modeling tradeoffs. Thus, Overton was built to construct a deployable production model. The runtime performance of the model is potentially suboptimal, but it is well within production SLAs. By encompassing more of the process, Overton has allowed faster model turn-around times.

Use Standard Tools for the ML Workflow Overton compiles the schema into (many versions of) TensorFlow, CoreML, or PyTorch. Whenever possible, Overton uses a standard toolchain. Using standard tools, Overton supports distributed training, hyperparameter tuning, and building servable models. One unanticipated benefit of having both backends was that different resources are often available more conveniently on different platforms. For example, to experiment with pretrained models, the Huggingface repository [14] allows quick experimentation–but only in PyTorch. The TensorFlow production tools are unmatched. The PyTorch execution mode also allows REPL and in-Jupyter-notebook debugging, which engineers use to repurpose elements, e.g., query similarity features. Even if a team uses a single runtime, different runtime services will inevitably use different versions of that runtime, and Overton insulates the modeling teams from the underlying changes in production serving infrastructure.

Model Independence and Zero-code Deep Learning A major design choice at the outset of the project was that domain engineers should not be forced to write traditional deep learning modeling code. Two years ago, this was a contentious decision as the zeitgeist was that new models were frequently published, and this choice would hamstring the developers. However, as the pace of new model building blocks has slowed, domain engineers no longer feel the need to fine-tune individual components at the level of TensorFlow. Ludwig[9] has taken this approach and garnered adoption. Although developed separately, Overton's schema looks very similar to Ludwig's programs and from conversations with the developers, shared similar motivations. Ludwig, however, focused on the one-off model building process not the management of the model lifecycle. Overton itself only supports text processing, but we are prototyping image, video, and multimodal applications.

Engineers are Comfortable with Automatic Hyperparameter Tuning Hyperparameter tuning is conceptually important as it allows Overton to avoid specifying parameters in the schema for the model builder. Engineers are comfortable with automatic tuning, and first versions of all Overton systems are tuned using standard approaches. Of course, engineers may override the search: Overton is used to produce servable models, and so due to SLAs, production models often pin certain key parameters to avoid tail performance regressions.

---

[9]https://uber.github.io/ludwig/

Resourcing	Error Reduction	Amount of Weak Supervision
High	65% (2.9×)	80%
Medium	82% (5.6×)	96%
Medium	72% (3.6×)	98%
Low	40% (1.7×)	99%

Figure 3: For products at various resource levels, percentage (and factor) fewer errors of Overton system makes compared to previous system, and the percentage of weak supervision of all supervision.

Make it easy to manage ancillary data products Overton is also used to produce back-end data products (e.g., updated word or multitask embeddings) and multiple versions of the same model. Inspired by HuggingFace [14], Overton tries to make it easy to drop in new pretrained embeddings as they arrive: they are simply loaded as payloads. Teams use multiple models to train a "large" and a "small" model on the same data. The large model is often used to populate caches and do error analysis, while the small model must meet SLA requirements. Overton makes it easy to keep these two models synchronized. Additionally, some data products can be expensive to produce (on the order of ten days), which means they are refreshed less frequently than the overall product. Overton does not have support for model versioning, which is likely a design oversight.

3 Evaluation

We elaborate on three items: (1) we describe how Overton improves production systems; (2) we report on the use of weak supervision in these systems; and (3) we discuss our experience with pretraining.[10]

Overton Usage Overton has powered industry-grade systems for more than a year. Figure 3 shows the end-to-end reduction in error of these systems: a high-resource system with tens of engineers, a large budget, and large existing training sets, and three other products with smaller teams. Overton enables a small team to perform the same duties that would traditionally be done by several, larger teams. Here, multitask learning is critical: the combined system reduces error and improves product turn-around times. Systems that Overton models replace are typically deep models and heuristics that are challenging to maintain, in our estimation because there is no model independence.

Usage of Weak Supervision Weak supervision is the dominant form of supervision in all applications. Even annotator labels (when used) are filtered and altered by privacy and programmatic quality control steps. Note that *validation* is still done manually, but this requires orders of magnitude less data than training.
 Figure 4a shows the impact of weak supervision on quality versus weak supervision scale. We downsample the training data and measure the test quality (F1 and accuracy) on 3 representative tasks: singleton, sequence, and set.[11] For each task, we use the 1x data's model as the baseline and plot the relative quality as a percentage of the baseline; e.g., if the baseline F1 is 0.8 and the subject F1 is 0.9, the relative quality is $0.9/0.8 = 1.125$. In Figure 4a, we see that increasing the amount of supervision consistently results in improved quality across all tasks. Going from 30K examples or so (1x) to 1M examples (32x) leads to a 12%+ bump in two tasks and a 5% bump in one task.

Pre-trained Models and Weak Supervision A major trend in the NLP community is to pre-train a large and complex language model using raw text and then fine-tune it for specific tasks [8]. One can easily integrate such pre-

---

[10]Due to sensitivity around production systems, we report relative quality numbers and obfuscate some tasks.

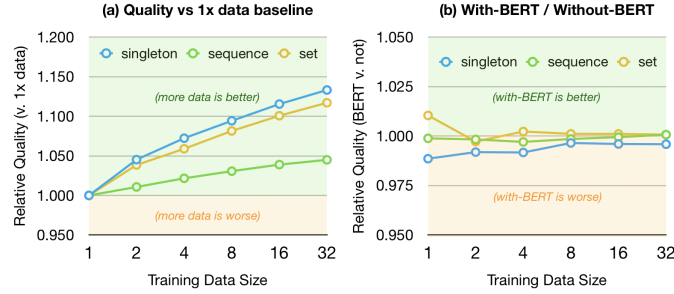[11]We obfuscate tasks using their underlying payload type.

Figure 4: Relative quality changes as data set size scales.

trained models in Overton, and we were excited by our early results. Of course, at some point, training data related to the task is more important than massive pretraining. We wondered how weak supervision and pretrained models would interact. Practically, these pretrained models like BERT take large amounts of memory and are much slower than standard word embeddings. Nevertheless, motivated by such models' stellar performance on several recent NLP benchmarks such as GLUE [31], we evaluate their impact on production tasks that are weakly supervised. For each of the aforementioned training set sizes, we train two models: **without-BERT**: production model with standard word embeddings but without BERT, and **with-BERT**: production model with fine tuning on the "BERT-Large, Uncased" pretrained model [8].

For each training set, we calculate the relative test quality change (percentage change in F1 or accuracy) of **with-BERT** over **without-BERT**. In Figure 4b, almost all percentage changes are within a narrow 2% band of no-change (i.e., 100%). This suggests that sometimes pre-trained language models have a limited impact on downstream tasks–when weak supervision is used. Pretrained models do have higher quality at smaller training dataset sizes–the Set task here shows an improvement at small scale, but this advantage vanishes at larger (weak) training set sizes in these workloads. This highlights a potentially interesting set of tradeoffs among weak supervision, pretraining, and the complexity of models.

4 Related Work

Overton builds on work in model life-cycle management, weak supervision, software for ML, and zero-code deep learning.

Model Management A host of recent data systems help manage the model process, including MLFlow[12], which helps with the model lifecycle and reporting [11], ModelDB [30], and more. Please see excellent tutorials such as Kumar et al. [16]. However, these systems are complementary and do not focus on Overton's three design points: fine-grained monitoring, diagnosing the workflow of updating supervision, and the production programming lifecycle. This paper reports on some key lessons learned from productionizing related ideas.

Weak Supervision A myriad of weak supervision techniques have been used over the last few decades of machine learning, notably external knowledge bases [4, 19, 27, 33], heuristic patterns [13, 20], feature annotations [18, 32], and noisy crowd labels [6, 15]. Data augmentation is another major source of training data. One promising approach is to learn augmentation policies, first described in Ratner et al. [21], which can further automate this process. Google's AutoAugment [5] used learned augmentation policies to set new state-of-the-art performance results in a variety of

---

[12]https://mlflow.org

domains, which has been a tremendously exciting direction. The goal of systems like Snorkel is to unify and extend these techniques to create and manipulate training data.[13] These have recently garnered usage at major companies, notably Snorkel DryBell at Google [12]. Overton is inspired by this work and takes the next natural step toward supervision management.

Software Productivity for ML Software The last few years have seen an unbelievable amount of change in the machine learning software landscape. TensorFlow, PyTorch, CoreML and MXNet have changed the way people write machine learning code to build models. Increasingly, there is a trend toward higher level interfaces. The pioneering work on higher level domain specific languages like Keras[14] began in this direction. Popular libraries like Fast.ai, which created a set of libraries and training materials, have dramatically improved engineer productivity.[15] These resources have made it easier to build models but equally important to train model developers. Enabled in part by this trend, Overton takes a different stance: *model development is in some cases not the key to product success.* Given a fixed budget of time to run a long-lived ML model, Overton is based on the idea that success or failure depends on engineers being able to iterate quickly and maintain the supervision–not change the model. Paraphrasing the classical relational database management mantra, Overton focuses on what the user wants–not how to get it.

Zero-code Deep Learning The ideas above led naturally to what we now recognize as *zero-code deep learning*, a term we borrow from Ludwig. It is directly related to previous work on multitask learning as a key building block of software development [22] and inspired by Software 2.0 ideas articulated by Karpathy.[16] The world of software engineering for machine learning is fascinating and nascent. In this spirit, Uber's Ludwig shares a great deal with Overton's design. Ludwig is very sophisticated and has supported complex tasks on vision and others. These methods were controversial two years ago, but seem to be gaining acceptance among production engineers. For us, these ideas began as an extension of joint inference and learning in DeepDive [25].

Network Architecture Search Zero-code deep learning in Overton is enabled by some amount of architecture search. It should be noted that Ludwig made a different choice: no search is required, and so zero-code deep learning does not depend on search. The area of Neural Architecture Search (NAS) [10] is booming: the goal of this area is to perform search (typically reinforcement learning but also increasingly random search [17]). This has led to exciting architectures like EfficientNet [28].[17] This is a tremendously exciting area with regular workshops at all major machine learning conferences. Overton is inspired by this area. On a technical level, the search used in Overton is a coarser-grained search than what is typically done in NAS. In particular, Overton searches over relatively limited large blocks, e.g., should we use an LSTM or CNN, not at a fine-grained level of connections. In preliminary experiments, NAS methods seemed to have diminishing returns and be quite expensive. More sophisticated search could only improve Overton, and we are excited to continue to apply advances in this area to Overton. Speed of developer iteration and the ability to ship production models seems was a higher priority than exploring fine details of architecture in Overton.

Statistical Relational Learning Overton's use of a relational schema to abstract statistical reasoning is inspired by Statistical Relational Learning (SRL), such as Markov Logic [9]. DeepDive [25], which is based on Markov Logic, allows one to wrap deep learning as relational predicates, which could then be composed. This inspired Overton's design of compositional payloads. In the terminology of SRL [7], Overton takes a knowledge compilation approach (Overton does

---

[13] http://snorkel.org
[14] http://keras.io
[15] http://fast.ai
[16] https://medium.com/@karpathy/software-2-0-a64152b37c35
[17] It is worth noting that data augmentation plays an important role in this architecture.

not have a distinct querying phase). Supporting more complex, application-level constraints seems ideally suited to an SRL approach, and is future work for Overton.

5 Conclusion and Future Work

This paper presented Overton, a system to help engineers manage the lifecycle of production machine learning systems. A key idea is to use a schema to separate the model from the supervision data, which allows developers to focus on supervision as their primary interaction method. A major direction of on-going work are the systems that build on Overton to aid in managing data augmentation, programmatic supervision, and collaboration.

References

[1] P. Agrawal, R. Arya, A. Bindal, S. Bhatia, A. Gagneja, J. Godlewski, Y. Low, T. Muss, M. M. Paliwal, S. Raman, V. Shah, B. Shen, L. Sugden, K. Zhao, and M. Wu. Data platform for machine learning. In *SIGMOD*, 2019.

[2] R. Caruana. Multitask learning: A knowledge-based source of inductive bias, 1993.

[3] V. S. Chen, S. Wu, A. Ratner, Z. Wang, and C. Ré. Slice-based Learning: A Programming Model for Residual Learning in Critical Data Slices. In *NeurIPS*, 2019.

[4] M. Craven and J. Kumlien. Constructing biological knowledge bases by extracting information from text sources, 1999.

[5] E. D. Cubuk, B. Zoph, D. Mané, V. Vasudevan, and Q. V. Le. Autoaugment: Learning augmentation policies from data. *CoRR*, abs/1805.09501, 2018.

[6] A. P. Dawid and A. M. Skene. Maximum likelihood estimation of observer error-rates using the em algorithm. *Applied statistics*, pages 20–28, 1979.

[7] G. V. den Broeck and D. Suciu. Query processing on probabilistic data: A survey. *Foundations and Trends in Databases*, 7(3-4):197–341, 2017.

[8] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, pages 4171–4186, 2019.

[9] P. M. Domingos and D. Lowd. Unifying logical and statistical AI with markov logic. *Commun. ACM*, 62(7):74–83, 2019.

[10] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *J. Mach. Learn. Res.*, 20:55:1–55:21, 2019.

[11] M. Z. et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41(4):39–45, 2018.

[12] S. H. B. et al. Snorkel drybell: A case study in deploying weak supervision at industrial scale. In *SIGMOD*, 2019.

[13] S. Gupta and C. D. Manning. Improved pattern learning for bootstrapped entity extraction., 2014.

[14] HuggingFace. *Hugging Face Repository*, 2019.

[15] D. R. Karger, S. Oh, and D. Shah. Iterative learning for reliable crowdsourcing systems, 2011.

[16] A. Kumar, M. Boehm, and J. Yang. Data management in machine learning: Challenges, techniques, and systems. In *SIGMOD*, pages 1717–1722, 2017.

[17] L. Li and A. Talwalkar. Random search and reproducibility for neural architecture search. In *Proceedings of the Thirty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI 2019, Tel Aviv, Israel, July 22-25, 2019*, page 129, 2019.

[18] G. S. Mann and A. McCallum. Generalized expectation criteria for semi-supervised learning with weakly labeled data. *JMLR*, 11(Feb):955–984, 2010.

[19] M. Mintz, S. Bills, R. Snow, and D. Jurafsky. Distant supervision for relation extraction without labeled data, 2009.

[20] A. Ratner, S. Bach, H. Ehrenberg, J. Fries, S. Wu, and C. Ré. Snorkel: Rapid training data creation with weak supervision, 2018.

[21] A. J. Ratner, H. R. Ehrenberg, Z. Hussain, J. Dunnmon, and C. Ré. Learning to compose domain-specific transformations for data augmentation. In *NIPS*, pages 3236–3246, 2017.

[22] A. J. Ratner, B. Hancock, and C. Ré. The role of massively multi-task and weak supervision in software 2.0. In *CIDR*, 2019.

[23] A. J. Ratner, C. D. Sa, S. Wu, D. Selsam, and C. Ré. Data programming: Creating large training sets, quickly. In *NIPS*, pages 3567–3575, 2016.

[24] S. Ruder. An overview of multi-task learning in deep neural networks. *CoRR*, abs/1706.05098, 2017.

[25] J. Shin, S. Wu, F. Wang, C. D. Sa, C. Zhang, and C. Ré. Incremental knowledge base construction using deepdive. *PVLDB*, 8(11):1310–1321, 2015.

[26] A. Søgaard and Y. Goldberg. Deep multi-task learning with low level tasks supervised at lower layers, 2016.

[27] S. Takamatsu, I. Sato, and H. Nakagawa. Reducing wrong labels in distant supervision for relation extraction, 2012.

[28] M. Tan and Q. V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, pages 6105–6114, 2019.

[29] P. Varma, F. Sala, A. He, A. Ratner, and C. Ré. Learning dependency structures for weak supervision models. In *ICML*, pages 6418–6427, 2019.

[30] M. Vartak and S. Madden. MODELDB: opportunities and challenges in managing machine learning models. *IEEE Data Eng. Bull.*, 41(4):16–25, 2018.

[31] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *BlackboxNLP@EMNLP 2018*, pages 353–355, 2018.

[32] O. F. Zaidan and J. Eisner. Modeling annotators: A generative approach to learning from annotator rationales, 2008.

[33] C. Zhang, C. Ré, M. Cafarella, C. De Sa, A. Ratner, J. Shin, F. Wang, and S. Wu. DeepDive: Declarative knowledge base construction. *Commun. ACM*, 60(5):93–102, 2017.