ML for Operations
Pitfalls, Dead Ends, and Hope

STEVEN ROSS AND TODD UNDERWOOD

Steven Ross is a Technical Lead in site reliability engineering for Google in Pittsburgh, and has worked on machine learning at Google since Pittsburgh Pattern Recognition was acquired by Google in 2011. Before that he worked as a Software Engineer for Dart Communications, Fishtail Design Automation, and then Pittsburgh Pattern Recognition until Google acquired it. Steven has a BS from Carnegie Mellon University (1999) and an MS in electrical and computer engineering from Northwestern University (2000). He is interested in mass-producing machine learning models. stross@google.com

Todd Underwood is a lead Machine Learning for Site Reliability Engineering Director at Google and is a Site Lead for Google's Pittsburgh office. ML SRE teams build and scale internal and external ML services and are critical to almost every product area at Google. Todd was in charge of operations, security, and peering for Renesys's Internet intelligence services that is now part of Oracle's cloud service. He also did research for some early social products that Renesys worked on. Before that Todd was Chief Technology Officer of Oso Grande, an independent Internet service provider (AS2901) in New Mexico. Todd has a BA in philosophy from Columbia University and a MS in computer science from the University of New Mexico. He is interested in how to make computers and people work much, much better together. tmu@goggle.com

Machine learning (ML) is often proposed as the solution to automate this unpleasant work. Many believe that ML will provide near-magical solutions to these problems. This article is for developers and systems engineers with production responsibilities who are lured by the siren song of magical operations that ML seems to sing. Assuming no prior detailed expertise in ML, we provide an overview of how ML works and doesn't, production considerations with using it, and an assessment of considerations for using ML to solve various operations problems.

Even in an age of cloud services, maintaining applications in production is full of hard and tedious work. This is unrewarding labor, or toil, that we collectively would like to automate. The worst of this toil is manual, repetitive, tactical, devoid of enduring value, and scales linearly as a service grows. Think of work such as manually building/testing/deploying binaries, configuring memory limits, and responding to false-positive pages. This toil takes time from activities that are more interesting and produce more enduring value, but it exists because it takes just enough human judgment that it is difficult to find simple, workable heuristics to replace those humans.

We will list a number of ideas that appear plausible but, in fact, are not workable.

What Is ML?

Machine learning is the study of algorithms that learn from data. More specifically, ML is the study of algorithms that enable computer systems to solve some specific problem or perform some task by learning from known examples of data. Using ML requires training a model on data where each element in the data has variables of interest (features) specified for it. This training creates a model that can later be used to make inferences about new data. The generated model is a mathematical function, which determines the predicted value(s) ("dependent variable(s)") based on some input values ("independent variables"). How well the model's inferences fit the historical data is the objective function, generally a function of the difference between predictions and correct inferences for supervised models. In an iterative algorithm, the model parameters are adjusted incrementally on every iteration such that they (hopefully) decrease the objective function.

Main Types of ML

In order to understand how we'll apply ML, it is useful to understand the main types of ML and how they are generally used. Here are broad categories:

Supervised Learning

A supervised learning system is presented with example inputs and their desired outputs labeled by someone or a piece of software that knows the correct answer. The goal is to learn a mapping from inputs to outputs that also works well on new inputs. Supervised learning is the most popular form of ML in production. It generally works well if your data consist of a large volume (millions to trillions) of correctly labeled training examples. It can be effective with many fewer examples, depending on the specific application, but it most commonly does well with lots of input data.

Think of identifying fruit in an image. Given a set of pictures that either contain apples or oranges, humans do an amazing job of picking out the right label ("apple" or "orange") for the right object. But doing this without ML is actually quite challenging because the heuristics are not at all easy. Color won't work since some apples are green and some oranges are green. Shape won't work because it's hard to project at various angles, and some apples are exceedingly round. We could try to figure out the skin/texture but some oranges are smooth and some apples are bumpy.

With ML we simply train a model on a few hundred (or a few thousand) pictures labeled "orange" or "apple." The model builds up a set of combinations of features that predict whether the picture has an apple or an orange in it.

Unsupervised Learning

The goal of unsupervised learning is to cluster pieces of data by some degree of "similarity" without making any particular opinion about what they are, i.e., what label applies to each cluster. So unsupervised learning draws inferences without labels, such as classifying patterns in the data.

One easy-to-understand use case is fraud detection. Unsupervised learning on a set of transactions can identify small clusters of outliers, where some combination of features (card-not-present, account creation time, amount, merchant, expense category, location, time of day) is unusual in some way.

Unsupervised learning is particularly useful as part of a broader strategy of ML, as we'll see below. In particular, in the example above, clustering outlier transactions isn't useful unless we do something with that information.

Semi-Supervised Learning

The goal of semi-supervised learning is to discover characteristics of a data set when only a subset of the data is labeled. Human raters are generally very expensive and slow, so semi-supervised learning tries to use a hybrid of human-labeled data and automatically "guessed" labels based on those human labels. Heuristics are used to generate assumed labels for the data that isn't labeled, based on its relationship to the data that is labeled.

Semi-supervised learning is often used in conjunction with unsupervised learning and supervised learning to generate better results from less effort.

Reinforcement Learning

In reinforcement learning (RL), software is configured to take actions in an environment or a simulation of an environment in order to accomplish some goal or cumulative set of values. The software is often competing with another system (which may be a prior copy of itself or might be a human) without externally provided labeled training data, following the rules.

Google's DeepMind division is well known for using RL to solve various real-world problems. Famously, this has included playing (and winning) against humans in the strategy game Go [1] as well as the video game StarCraft [2]. But it has also included such practical and important work as optimizing datacenter power utilization [3].

ML for Operations: Why Is It Hard?

Given that ML facilitates clustering, categorization, and actions on data, it is enormously appealing as a system to automate operational tasks. ML offers the promise of replacing the human judgment still used in decisions, such as whether a particular new deployment works well enough to continue the roll-out, and whether a given alert is a false positive or foreshadowing a real outage. Several factors make this more difficult than one might think.

ML produces models that encode information by interpreting features in a fashion that is often difficult to explain and debug (especially with deep neural networks, a powerful ML technique). Errors in the training data, bugs in the ML algorithm implementation, or mismatches between the encoding of data between training and inference will often cause serious errors in the resulting predictions that are hard to debug. Below we summarize some common issues.

ML Makes Errors

ML is probabilistic in nature, so it will not always be right. It can classify cats as dogs or even blueberry muffins [4] as dogs a small fraction of the time, especially if the data being analyzed is significantly different from any specific training example. Of course, humans make errors as well, but we are often better able to predict, tolerate, and understand the types of errors that humans make. Systems need to be designed so such occasional gross errors will be tolerable, which sometimes requires sanity tests on the result (especially for numerical predictions).

Large Problem Spaces Require Lots of Training Data

The more possible combinations of feature values that a model needs to deal with, the more training data it requires to be accurate. In other words, where many factors could contribute to a particular labeling or clustering decision, more data is required. But in large feature spaces, there may be a large difference between examples being analyzed and the closest training data, leading to error caused by trying to generalize over a large space. This is one of the most serious issues with using ML in operations, as it is often hard to find sufficient correctly labeled training data, and there are often many relevant variables/features.

Specifically, the problem space of production engineering or operations is much messier than the space of fruit categorization.

In practice, it turns out to be quite difficult to get experts to categorize outages, SLO violations, and causes in a mutually consistent manner. Getting good labels is going to be quite difficult.

Training Data Is Biased Relative to Inference Demand

The data you use to train your model may be too different from the data you're trying to cluster or categorize. If your training data only cover a particular subset of all things the model might need to infer over, all the other slices it wasn't trained on will see higher errors because of their divergence from the training data. Additionally, if the statistical distribution of classifications in the training data differs from the statistical distribution in the real world, the model will make skewed predictions, thinking that things that are more common in the training set are more common in the real world than they really are. For example, if the training data had 10 million dogs and 1000 cats, and dogs and cats are equally likely in the inference examples, it will tend to infer the presence of a dog more often than it should.

Lack of Explainability

Many of the best performing ML systems make judgments that are opaque to their users. In other words, it is often difficult or impossible to know why, in human intelligible terms, an ML model made a particular decision with respect to an example. In some problem domains, this is absolutely not a difficulty. For example, if you have a large number of false positive alerts for a production system and you're simply trying to reduce that, it's not generally a concern to know that an ML model will use unexpected combinations of features to decide which alerts are real. For this specific application, as long as the model is accurate, it is useful. But models with high accuracy due purely to correlation rather than causation do not support decision making. In other situations aspects of provable fairness and lack of bias are critical. Finally, sometimes customers or users are simply uncomfortable with systems that make decisions that cannot be explained to them.

Potential Applications of ML to Operations

Given all of these challenges, it will be useful to examine several potential applications of ML to operations problems and consider which of these is feasible or even possible.

Monitoring

For complex systems, the first problem of production maintenance is deciding which of many thousands of variables to monitor. Candidates might include RAM use by process, latency for particular operations, request rate from end users, timestamp of most recent build, storage usage by customer, number, and type of connections to other microservices, and so on. The possibilities of exactly what to monitor seem unbounded.

Systems and software engineering sometimes suggest using ML to identify the most relevant variables to monitor. The objective would be to correlate particular data series with the kinds of events that we are most interested in predicting—for example, outages, slowness, capacity shortfalls, or other problems.

In order to understand why this is a difficult problem, let us consider how to build an ML model to solve it. In order to use ML to create a dashboard that highlights the best metrics to see any current problems with your system, the best approach will be to treat the problem as a supervised multiclass prediction problem. To address that problem we will need:

- A class to predict for every metric of interest
- Labels for all classes that were helpful for millions of production events of concern
- Training and periodic retraining of your model as you fix bugs and create new ones with failure types shifting over time
- Periodic (potentially on page load) inferring with the model over which metrics should be shown to the user.

There are other complexities, but the biggest issue here is that you need millions of labeled training examples of production events of concern. Without millions of properly categorized examples, simple heuristics, for example that operators select the metrics that appear to be the most relevant, are likely to be as or more effective and at a fraction of the cost to develop and maintain. Simple heuristics also have several advantages over ML, as previously mentioned. We hope you don't have millions of serious problematic events to your production infrastructure to train over. However, if your infrastructure is of a scale and complexity that you think that you will, eventually, have an appropriate amount of data for this kind of application, you should begin accumulating and structuring that data now.

Alerting

Most production systems have some kind of manually configured but automated alerting system. The objective of these systems is to alert a human if and only if there is something wrong with the system that cannot be automatically mitigated by the system itself.

The general idea of an ML-centric approach to alerting is that once you have determined which time series of data are worth *monitoring* (see above) it might be possible to automatically and semi-continuously correlate values and combinations of these. To accomplish this we can start with every alert that we have or could easily have and create a class for each.

We then need to create positive and negative labels. Positive labels are applied to the alerts that were both useful and predictive of some serious problem in the system that actually required human intervention. Negative labels are the opposite: either not useful or not predictive of required human intervention. We need to label many events, those where something bad was happening and those where everything was fine, and continuously add new training examples. To scope the effort, we estimate that we will need at least tens of thousands of positive examples and probably even more (millions, most likely) of negative examples in order to have a pure-ML solution that is able to differentiate real problems from noise more effectively than a simple heuristic. We are not discussing potential hybrid heuristic + ML solutions here since, in many practical setups, this will lead to increased complexity from integrating two systems that need to be kept in sync for the intended outcome, which is unlikely to be worth the extra effort.

Even if we had all these labels (and they're correct) and a good model, which we know to be difficult from the monitoring case above, the on-call will still need to know where to look for the problem. While we may be able to correlate anomalous metrics with a confident alerting signal, covering the majority of alert explanations this way would not be enough. For as long as the fraction of "unexplainable" alerts is perceived by alert recipients as high, the explainability problem makes adoption cumbersome at best. This is the problem of explainability.

Canarying/Validation

Pushing new software to production frequently or continuously as soon as it is effectively tested poses risks that new software will sometimes be broken in ways the tests won't catch. The standard mitigation for this is to use a canary process that gradually rolls out to production combined with monitoring for problems and a rapid rollback if problems are detected. The problem is that monitoring is incomplete, so occasionally bad pushes slip through the canary process unnoticed and cause serious issues.

For this reason, production engineers often suggest using ML to automatically detect bad pushes and alert and/or roll them back.

This is a specialized version of the alerting problem; you need positive labels and negative labels, labeling successful pushes with positive labels and broken pushes with negative labels. Much like with alerting, you will probably need thousands of examples of bad pushes and hundreds of thousands of examples of good pushes to differentiate real problems from noise better than a simple heuristic. The main factor that makes canarying a little less hard than general alerting is that you have a strong signal of a high-risk event when your canary starts (as opposed to continuous monitoring for general alerting) and an obvious mitigation step (roll back), but you still need a large number of correctly labeled examples to do better than heuristics. Note that if you have a bad push that you didn't notice in your labeling, because it was rolled back too fast or got blocked by something else and improperly labeled as a good push, it will mess up your data and confuse your ML model.

False-positive canary failures will halt your release (which is usually a preferable outcome to an outage). To maintain release velocity, these need to be kept to a minimum, but that will lower the sensitivity of your model.

Root Cause Analysis

Outages are difficult to troubleshoot because there are a huge number of possible root causes. Experienced engineers tend to be much faster than inexperienced engineers, showing that there is some knowledge that can be learned.

Production engineers would like to use ML to identify the most likely causes and surface information about them in an ordered fashion to the people debugging problems so that they can concentrate on what is likely. This would require classifying the set of most likely causes, and then labeling and training over enough data to rank this list of causes appropriately.

Because you need a fixed list of classes to train over for this problem, if a new type of problem shows up your model won't be able to predict it until it has trained over enough new examples. If you have a case that isn't on your list, then people may spend excessive time looking through the examples recommended by the model even though they're irrelevant. To minimize this risk, you might want to add lots of classes to handle every different possibility you can think of, but this makes the training problem harder as you need more properly labeled training data for every class of problem you want the model to be able to predict. To be able to differentiate between a list of a hundred causes, you'll probably need tens of thousands of properly labeled training examples. It will be difficult to label these examples with the correct root cause(s) without a huge number of incidents, and there is a strong risk that some of the manually determined root cause labels will be incorrect due to the complexity, making the model inaccurate. An additional complexity is that events (potential causes) sequenced in one order may be harmless (capacity taken down for updates after load has dropped), but sequenced in another order may cause a serious outage (capacity taken down for updates during peak load), and the importance of this sequencing may confuse the ML model.

A manually assembled dashboard with a list of the top *N* most common root causes, how to determine them (some of which might be automated heuristics), and related monitoring will probably be more helpful than an ML model for root cause analysis in most production systems today.

Path Forward

We do not recommend that most organizations use machine learning to manage production operations at this point in the maturity of software services and ML itself. Most systems are not large enough and would do better to focus their engineering effort and compute resources on more straightforward means of improving production operations or expanding the business by improving the product itself. Unless all of your monitoring is well curated, alerting is carefully tuned, new code releases thoroughly tested, and rollouts carefully and correctly canaried, there is no need to expend the effort on ML.

However, in the future as production deployments scale, data collection becomes easier, and ML pipelines are increasingly automated, ML will definitely be useful to a larger fraction of system operators. Here are some ways to get ready:

1. Collect your data. Figure out what data you think you might use to run production infrastructure and collect it.
2. Curate those data. Make sure that the data are part of a system that separates and, where possible, labels the data.
3. Begin to experiment with ML. Identify models that might make sense and, with the full understanding that they will not reach production any time soon, begin the process of prototyping.

Conclusion

While ML is promising for many applications, it is difficult to apply to operations today because it makes errors, it requires a large amount of high-quality training data that is hard to obtain and label correctly, and it's hard to explain the reasons behind its decisions. We've identified some areas where people commonly think ML can help in operations and what makes it difficult to use in those applications. We recommend using standard tools to improve operations first before moving forward with ML, and we suggest collecting and curating your training data as the first step to take before using ML in operations.

References

[1] https://deepmind.com/research/case-studies/alphago-the-story-so-far.

[2] https://www.seas.upenn.edu/~cis520/papers/RL_for_starcraft.pdf.

[3] https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42542.pdf.

[4] https://www.topbots.com/chihuahua-muffin-searching-best-computer-vision-api/.