Google

How ML Breaks

Fifteen years of ML production pipeline outages and insight

dannyp@google.com, tmu@google.com

How *(one big specific)* ML *(system actually)* ~~*Breaks*~~ Broke

dannyp@google.com, tmu@google.com

Quick Intro

Agenda

- Quick Intro
- Motivation: Understand how (at least one) ML system breaks so that we can run them more reliably
- Background/Methodology: 10+ years of detailed post-mortem writeups for one of the larger ML systems at Google.
- Failure Taxonomy
- Results
- Discussion & Recommendations

Basic Questions

- **Who are we?**
  - Daniel: 10+ years of building and productionizing large ML systems at Google.
  - Todd: 11+ years building multi-tenant ML systems at Google. Leads ML SRE for Google.
- **Who are you?**
  - Builders, operators, clients of ML pipelines/systems who want them to work better.
- **Why are we here?**
  - To understand and take advantage of failures. Never let a good outage (or 10 years worth of good outages) go to waste.

Motivation

For ML to Matter it has to Work

ML systems break. All the time.

- We cannot deploy models that don't finish training, or that train on bad data, etc.
- People who run pipelines, especially large pipelines, especially continuously (or periodically) retrained pipelines, know that failures are common
- Until ML pipelines reliably produce high quality models most organizations won't rely on them.

Failure is the Best Way to Understand Failure

Standard principle of reliability engineering

- Failures are a gift: they are a natural experiment of what, at least once, broke and usually why.
- In aggregate, these failures teach us what failures are most common.
- Understanding failures can help us avoid, mitigate or resolve those failures.

Hypothesis

Many (most?) ML Failures Probably Have Nothing To Do With ML

- Impression: most ML failures aren’t actually ML failures.
- Boring, commonplace failures are more difficult to notice and more difficult to take seriously (c.f. Semmelweiz and hand washing)
- Evidence could confirm this and make it more actionable. Or could contradict and point towards suitable ML-centric reliability work.

Background/Methodology

The System Under Study

One of the largest, oldest ML systems at Google

- Google has been using ML to optimize some of our larger ranking and selection systems for a long time.
- One of these systems is both particularly old (15+ years although having gone through multiple redesigns) and well documented (full postmortems with metadata available for the last 10 years).
- O(thousands) of models training concurrently of O(100B) parameters in size.
- Trains periodically O(1hr) to update model with newly arrived data.
- Serving system is global and new models are continuously synced to serving.

Google

The Dataset

We maintain a dataset of all postmortems in the company

- Google has a database of most of the postmortems written back to the earliest days of the company, indexed and searchable.
- We Searched for outages including the name of two largest components of the system.
- Identified 96 postmortems over the last ~10 years.
- Postmortem metadata include a root cause analysis and impact level. These were manually categorized into 19 categories (more on the taxonomy of failure in a minute).

Methodology

- Causes were categorized into one of 19 categories
  - The most common category caused 15 of the 96 outages analyzed
  - The least common category caused a single outage.
- Cause categories were further grouped along two axes and ranked on a 5-point scale
  - ML vs. not-really-ML
  - Distributed vs. Single System
- Categorizations were based purely on description of the outage cause.

An Example:

One amusing (in retrospect) failure

- Data arriving from multiple sources was joined prior to being sent to training. The joining implicitly provided positive labels for the data.
- The rate of new data and the total amount of processing both grew over time until some of the joining was delayed.
- Downstream training trained on unjoined (and therefore unlabeled) data as if it were all negatively labeled.
- Hijinks ensues. Fixing this kind of chronic resource planning problem is challenging.

Another Example:

A boring but totally common failure:

- Data input processing pipeline joins data from a structured data source.
- In order to save resources in the original location we copied the structured data source to a new location
- The pipeline lacked permissions to read the data source in the new location, failing to join the contents of that data source.
- The entire pipeline lost the ability to process new data.

Failure Taxonomy

Categories of Failure

Nineteen ways of thinking about how things break

- Process orchestration issues
- Overloaded backends
- Temporary failure to join with expected data
- CPU failures
- Cache invalidation bugs
- Changes to the distribution of examples that we are generating inference on
- Config changes pushed out of order
- Suboptimal data structure used
- Challenges assigning work between clusters
- Example training strategy resulted in unexpected ordering
- ML hyperparameters adjusted on the fly
- Configuration change not properly canaried or validated
- Client made incorrect assumption about model providing inference
- Inference takes too long
- Incorrect assert() in code
- Labels weren't available/mostly correct at the time the model wished to visit the example
- Embeddings interpreted in the wrong embedding-space
- QA/Test jobs incorrectly communicating with prod backends
- Failed to provision necessary resources (bandwidth, ram, CPU)

ML vs. Not-ML

Would this kind of failure have happened in a non-ML pipeline?

ML:

- Changes to the distribution of examples.
- Problems with selection and processing of training data: either sampling wrong, re-visiting the same data, skipping data, etc.
- Hyperparameters
- Mismatch in embedding interpretation
- Training on mislabeled data

Not ML:

- Dependency failure (other than data)
- Deployment failure (out of order, wrong target, wrong binaries, etc.)
- CPU failures
- Inefficient data structure

Distributed Vs. Not

How much of this is a distributed systems problem?

Distributed:

- System orchestration: which processes to run where
- Data joined between two systems fails (e.g.: missing foreign key)
- Some resource (e.g. CPU) is unavailable in the quantities we need
- Changes pushed in an unsafe order

Less Distributed

- CPU oddities (probabilistically distributed: only happening at huge scales)
- Human driven change not tested before being applied to production environment

Not Distributed:

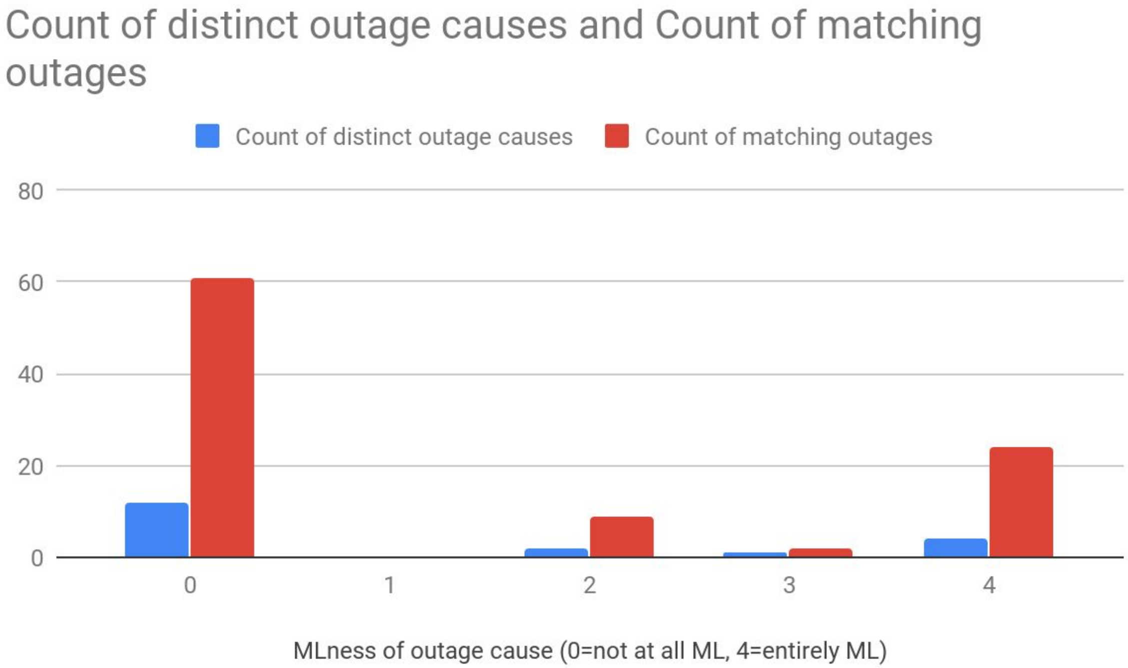
- Failed ASSERT: invariant is not invariant
- Bad data structures

Results

Finding #1: Most outages and their causes aren't ML

Failures are not characteristic of ML

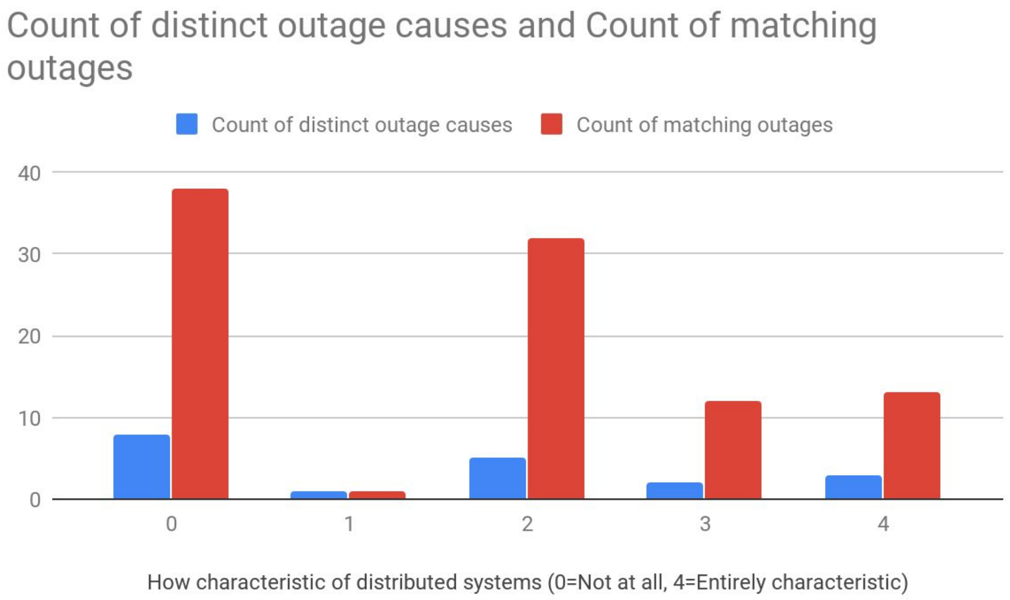
- The plurality of our distinct causes, accounting for the majority of our outages, were rated as not at all characteristic of ML.
- A subset of our outages were rated as being purely characteristic of ML. The middle ground is rare.



Finding #2: A plurality of outages have a distributed systems component

More outages are explainable as having a distributed systems component

- While about 40% of our postmortems had root causes rated as not at all characteristic of distributed systems, the majority (60%) at least partially resembled problems that are characteristic of distributed systems.



Discussion/Recommendations

Understand Failures to Avoid Them

Our system is not your system

- Systems break for the reasons they break, not for other reasons.
- Understanding how your pipelines break can direct investment into fixes
- Track outages and regressions carefully. Write postmortems and store them somewhere
- Categorize outages by severity, impact, duration and root cause category
- Review root causes regularly (yearly) for patterns.

Architect Resilient Pipelines

Build a team to build a system

- Staff with distributed data processing skills may be more important than staff with ML-specific experience.
- Testing probably matters more than ML sophistication
- Tactically: basic distributed systems hygiene and testing may have the best pay off:
    - Monitor pipeline throughput, completion rates, histograms
    - Carefully track versions of data, models and binaries
    - Pay close attention to capacity and utilization

Thank You

Image by Gundula Vogel from Pixabay

Google