



SRECON 2026 AMERICAS · MAR 26

Closing the Observability Gap in AI Training

with **TrainCheck** — Beyond Loss and Accuracy

Yuxuan Jiang
University of Michigan

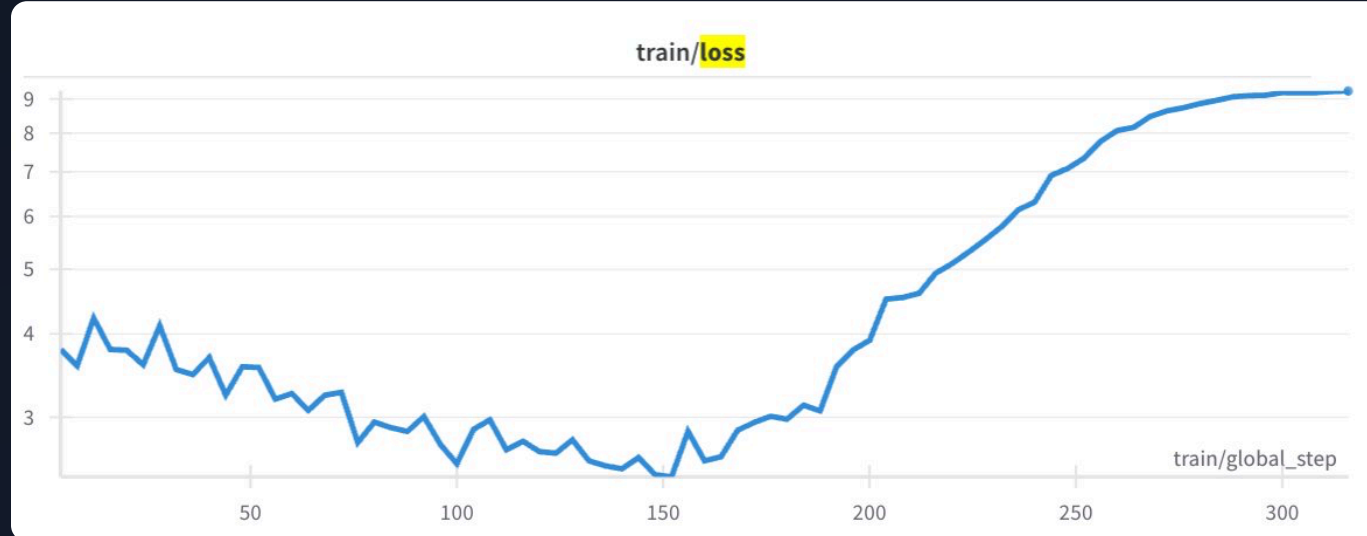
Ryan Huang
University of Michigan

Agenda

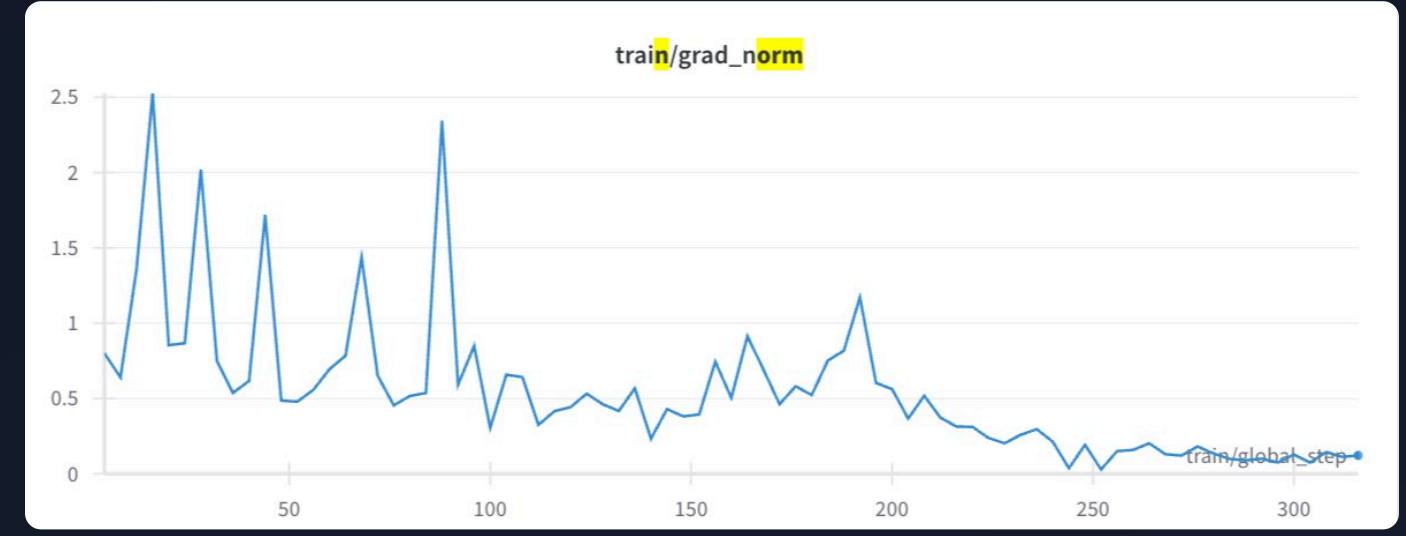
- Silent failures in ML training
- Why this is a monitoring problem
- TrainCheck: adding correctness signals
- Demo: catching issues early
- Real examples from production
- Key takeaways

Loss down. GPUs busy. No errors.

A real bug by @jxmnop. DDP training with HuggingFace. 35+ replies, hundreds of likes on X.



✓ First ~150 steps: loss decreasing → ✗ Then: loss climbs back up



✓ Grad norms: reasonable throughout No spike, no signal

So people start turning knobs.

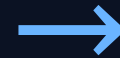
Gradient clipping. Optimizer changes. Learning rate tweaks.

THE QUESTION

"Why does my model get worse over time without the grad norm noticeably increasing?"



Loss climbs past step 150. No standard metric explains why.



THE REPLIES

- Try gradient clipping
- Switch the optimizer
- Increase the learning rate
- Check the data pipeline
- Maybe it's a batchnorm issue?

The root cause was there from **step 1**.

A missing gradient sync hook caused each GPU to train its own model. Loss still went down initially.

✓ WHAT SHOULD HAPPEN

GPU 0

Compute gradients

GPU 1

Compute gradients

GPU 2

Compute gradients

AllReduce → average gradients → one shared model



✗ WHAT ACTUALLY HAPPENED

GPU 0

Updated its **own** model

GPU 1

Updated its **own** model

GPU 2

Updated its **own** model

Loss still looks good — model learned to **cheat**

In DDP, we expect **weights to be consistent across GPUs**. If we had monitored that from step 1, we would have caught this immediately.

KEY OBSERVATION

That's not diagnosis.
That's guessing.

By the time you are here, you've already lost hours.

Late

Frustrating

Expensive

This keeps happening.

Training failures that look healthy are a well-recognized pattern across the community.



Mayank Mishra

@MayankMish98

"We identified an issue with the Mamba-2 initialization in HuggingFace and FlashLinearAttention (dt_bias being incorrectly initialized)... The difference is substantial. Mamba-2 seems to be quite sensitive to the initialization."

❤️ 747 💬 17 ↻ 99 👁 368K



BLOOM Developer

BigScience / 176B model

"We had no time to do much software validation because we were rushed to train... as soon as it looked OK, we had to start training."

384 A100s · 3.5 months ·
\$millions



Andrej Karpathy ✓

@karpathy

"Beautiful technical debugging detective longread that starts with a suspicious loss curve and ends all the way in the Objective-C++ depths of PyTorch MPS backend of addcmul_ that silently fails on non-contiguous output tensors."

❤️ 4.2K 💬 201 ↻ 313 👁 605K

This is a monitoring problem.

We lack the right signals. The ones we have are too high-level to catch these bugs early.

WHY EXISTING SIGNALS FAIL

- Noisy: loss fluctuates naturally, masking real issues
- Non-deterministic: hard to distinguish a bug from normal variance
- End-to-end: loss is the final aggregate of thousands of components
- Late: by the time loss diverges, the root cause may be thousands of steps old

THE CONSEQUENCE

- No symptoms until it is too late
- Signal is unclear: is this a real issue or normal noise?
- Detection and debugging are conflated into one painful loop

We need fine-grained, deterministic correctness signals.

We are missing correctness signals.

WHAT WE ALREADY MONITOR

- ✓ Training / validation loss
- ✓ GPU utilization & throughput
- ✓ Crash logs and exceptions
- ✓ Step time / iteration speed

WHAT WE DO NOT MONITOR

- ✗ Whether gradients are actually updating all parameters
- ✗ Whether all GPUs are in sync
- ✗ Whether the training loop is following expected patterns
- ✗ Whether *training itself is correct*

"We measure *activity*. We don't measure *correctness*."

Many silent bugs have precise, actionable root causes at the [API behavior](#) and [event sequence](#) level. That is where we need to monitor.

Capture correctness issues **early**.

Apply SRE discipline to training: define expected behavior, monitor it, alert on violations.

SRE STANDARD FOR PRODUCTION SYSTEMS

- Define expected behavior as invariants
- Monitor continuously against those invariants
- Alert immediately on violations
- Capture enough signal for fast root cause analysis

TRAINING TODAY HAS NONE OF THIS

- No correctness invariants
- No continuous correctness monitoring
- No fine-grained alerts
- **Yet training runs cost millions and last months**

Make correctness expectations explicit and checkable.

Every bug we showed has a simple expectation that, if monitored, would have caught it immediately.

BUG	EXPECTATION
DDP sync missing	Weights should be consistent across GPUs after each step
Frozen encoder (MPS)	optimizer.step() should update all tracked parameters
BLOOM TP divergence	LayerNorm weights equal across TP ranks
Mamba-2 init bug	Initialization values match expected distribution

Turn each expectation into a runtime check evaluated at every step

Violation = immediate signal, before hours of wasted compute



TrainCheck

Turn silent failures into signals.

- **Encodes** expected training behavior as checkable invariants
- **Validates** invariants at runtime, step by step, against the live trace
- **Surfaces** violations immediately with step, stage, and trace context

HOW IT WORKS

Training step → Check invariants → Alert on violations

Invariants are learned from healthy runs, not written by hand.
Preconditions ensure they only fire when relevant.

EXAMPLE INVARIANTS

```
// optimizer.step must follow loss.backward
APISequence(loss.backward, optimizer.step)

// weights must be equal across TP ranks (LayerNorm only)
Consistent(Parameter.data, Parameter.data)
  where UNEQUAL(TP_RANK) ^ EQUAL(name)

// zero_grad must reset gradients to None
EventContain(optim.zero_grad,
  VarChange(Parameter.grad, Any, None))

// optimizer.step must update all tracked parameters
APIContain(optimizer.step, Parameter.data)

// output dtype must match mixed-precision policy
ConsistentOutput(torch.matmul, attr.dtype)
```

What TrainCheck captures.

Structured, fine-grained execution traces. Not just loss scalars.

INSTRUMENTATION

- **Monkey-patching** for API calls
torch, nn, optim, lr_scheduler
- **Subclass / Proxy** for model tracking
Track parameter state changes across steps

OPTIMIZATIONS

- **Selective instrumentation** (only APIs in active invariants)
- **Sampling + warm-up** policies
- **Tensor hashing** to avoid expensive full comparisons

API TRACE (OPTIMIZER STEP, MNIST)

```
{ "func_call_id": "568b...", "thread_id": ..., "process_id": ...,  
  "meta_vars": { "step": 133, "stage": "training" },  
  "type": "function_call (pre)",  
  "function": "torch.optim.adadelta.adadelta",  
  "proxy_obj_names": [ "model.conv1.weight", "Parameter" ],  
  [ "model.conv1.bias", "Parameter" ], ... 6 more ... ],  
  "kwargs": { "lr": 0.7, "rho": 0.9, "eps": 1e-06, ... } }
```

VARIABLE TRACE (PARAMETER STATE CHANGE)

```
{ "process_id": ..., "thread_id": ...,  
  "meta_vars": { "step": 179, "stage": "training" },  
  "var_name": "model.conv1.weight",  
  "var_type": "torch.nn.Parameter",  
  "dump_loc": "torch.optim.adadelta.adadelta",  
  "attributes": {  
    "data": 2184268332262243433,  
    "dtype": "torch.float32",  
    "grad": -4966851652872791426,  
    "shape": [32, 1, 3, 3], "is_cuda": true, ... },  
  "type": "state_change" }
```

Real traces from MNIST. Each step produces structured events for every API call and parameter change.

How TrainCheck infers invariants.

Three steps: generate hypotheses, validate, deduce preconditions.

EXAMPLE: BLOOM PARAMETER DIVERGENCE

```
// simplified trace for torch.nn.Parameter
① name="layernorm.weight" TP_RANK=0 data=411977 tp=false
② name="layernorm.weight" TP_RANK=1 data=411977 tp=false
③ name="dense_h_to_4h.bias" TP_RANK=1 data=650462 tp=true
```

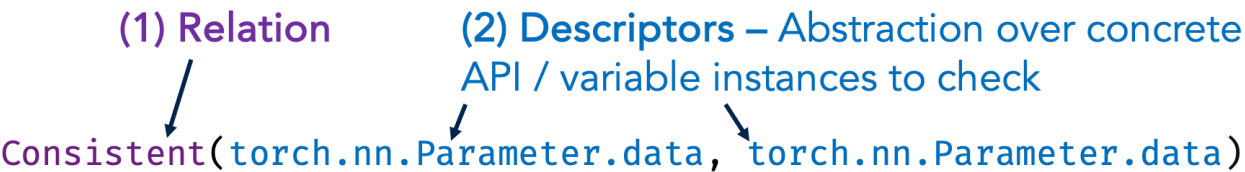
- 1 **Generate hypothesis**
Consistent(Parameter.data, Parameter.data)
- 2 **Validate against all traces**
Passing: ①② (same data). Failing: ①③ ②③ (different data).
- 3 **Deduce precondition**
UNEQUAL(TP_RANK) && EQUAL(step) && EQUAL(name) && CONSTANT(tp, false)

Invariant building blocks.

Each invariant has three parts: a relation, descriptors, and a precondition.

Invariant Representation

"The weights of certain layers should stay consistent across tensor parallelism (TP) ranks."



```
&& UNEQUAL(meta_vars.TP_RANK)
&& EQUAL(meta_vars.step)
&& EQUAL(name)
&& CONSTANT(attr.tensor_model_parallel, false)
```

!! Only applicable to LayerNorm (<1% of parameters).

(3) ★ Precondition (Context)

RELATION TYPES

Consistent(Va, Vb)	Va and Vb should have the same values
EventContain(Ea, Eb)	Eb must happen during the execution of Ea
APISequence(la, lb)	la and lb must occur in this order
APIArg(la, distinct)	Argument consistency or distinction across calls
APIOutput(la, bound)	Output must meet attribute constraints

Each relation is a **template** the inference engine matches against the trace. Hypotheses are generated proactively, then validated and pruned.

Preconditions make invariants **precise**: this consistency check only applies to <1% of parameters (LayerNorm, non-TP).

Preconditions: from patterns to accurate checks.

DL correctness is context-dependent. Statistical likelihood alone is not enough.

WHY PRECONDITIONS MATTER

- "Optimizer should update all parameters"
...except frozen layers, or when grad is None
- "Weights consistent across GPUs"
...only for non-tensor-parallel parameters
- "Dropout active during training"
...only when model.training is True

✓ Accepted

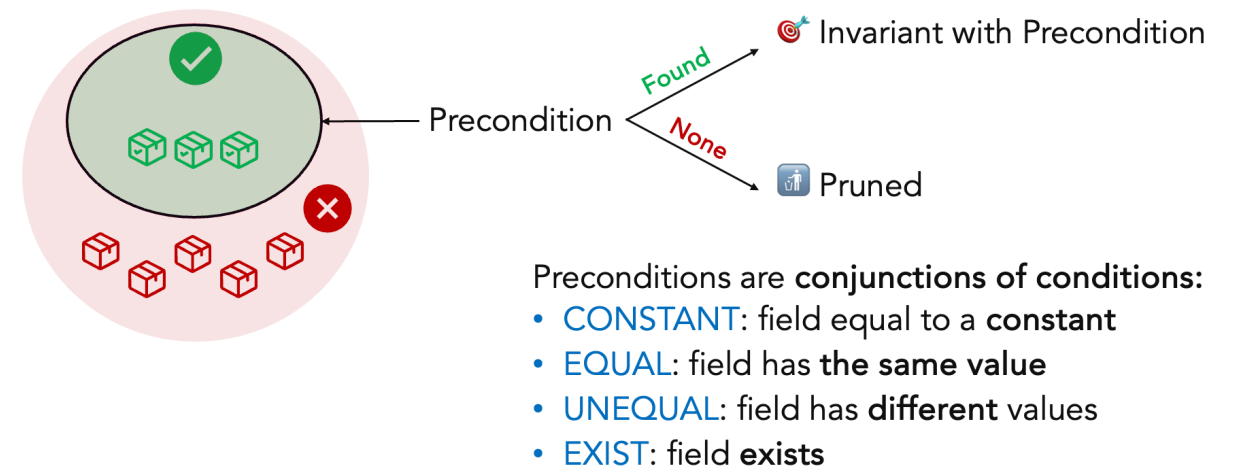
Bloom consistency invariant
1:38 pass/fail ratio
Valid precondition found

✗ Pruned

Consistent(is_cuda, requires_grad)
Holds 99% of the time
No meaningful precondition

In practice, 3 to 5 pipelines are enough to cover all bugs in our evaluation.

For every hypothesis, infer a precondition based on passing/failing examples:



PRECONDITION OPERATORS

CONSTANT field = value

EQUAL same value

UNEQUAL different

EXIST field present

Learn once, check everywhere.

DL correctness is context-dependent. So are our invariants.

KEY INSIGHT

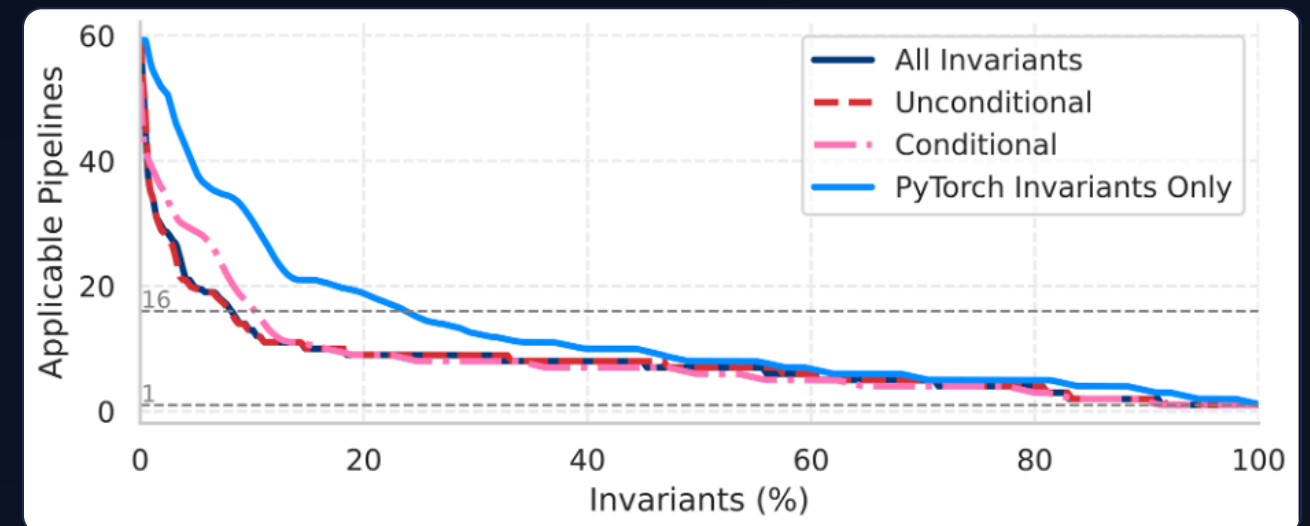
- **Correctness rules depend on context**
e.g. TP consistency only applies to non-sharded layers
- **TrainCheck infers preconditions automatically**
Invariants only fire when their context matches
- **Conditional invariants transfer better**
They encode the *when*, not just the *what*

100%

of tested silent errors covered
by 3 example pipelines

23%

of invariants transfer across
different training tasks



Applicable pipelines per invariant percentile

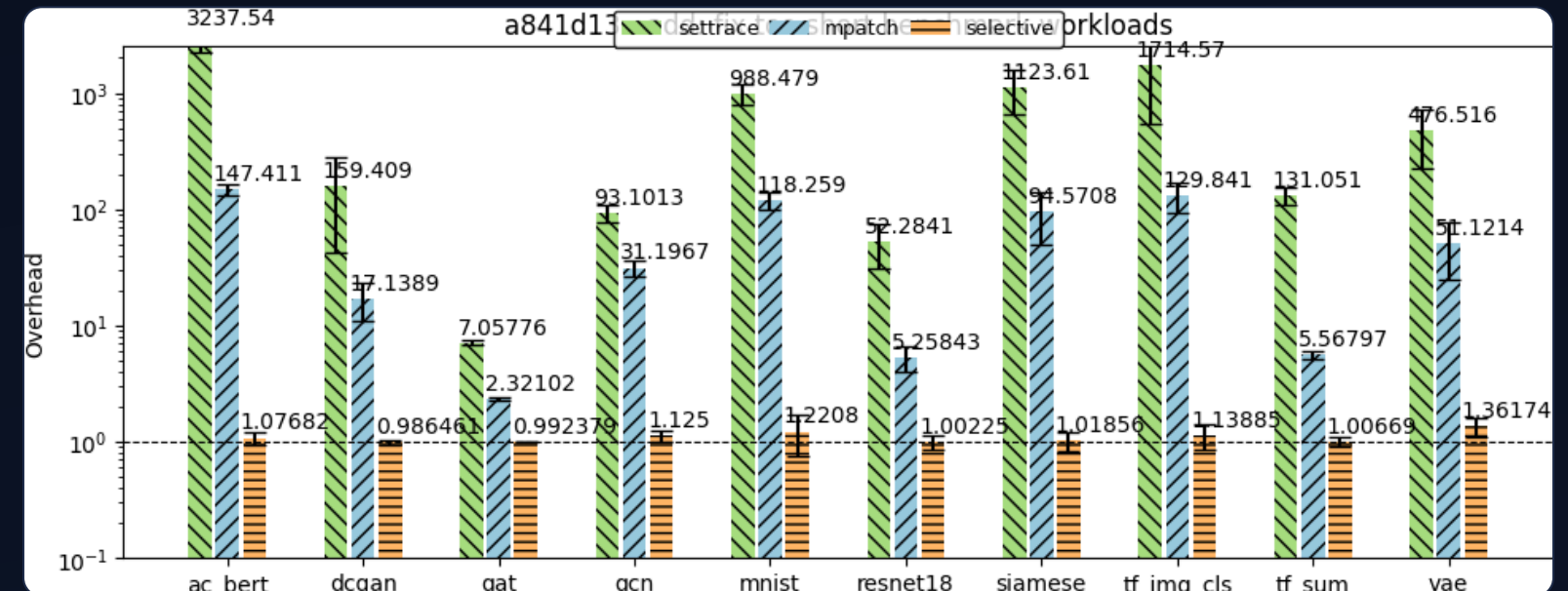
Low overhead.

Selective instrumentation keeps checking cost near zero.

~1x Near-baseline overhead on all tested workloads with selective instrumentation

Full trace collection is heavier — but only used during invariant *inference*, not checking.

- Warm-up period — skip first N steps
- Step sampling — check every Kth step
- Selective instrumentation — only APIs in active invariants



Per-iteration overhead — orange (selective) stays near 1x

Catch bugs as early as one iteration.

18 out of 20 real-world silent errors — detected within a single training step.

BUG	DETECTED AT	INVARIANT
BLOOM weight divergence	iter 3	ConsistencyRelation
FSDP stagnant training	iter 1	APIContainRelation
Frozen encoder (MPS)	iter 1	APIContainRelation
Wrong training mode	iter 1	FunctionLeadRelation
6 new bugs in DeepSpeed	iter 1	Various

WHAT THIS MEANS

- No more waiting for loss to diverge
- No more guessing at root causes
- Pinpoints exact root cause in 10/18 cases
- Close to root cause in remaining 8

Baselines (loss monitoring, PyTea+NeuRI)
detect only 3/20

Precise and effective.

Low false positive rate. High detection rate. Actionable results.

<2%

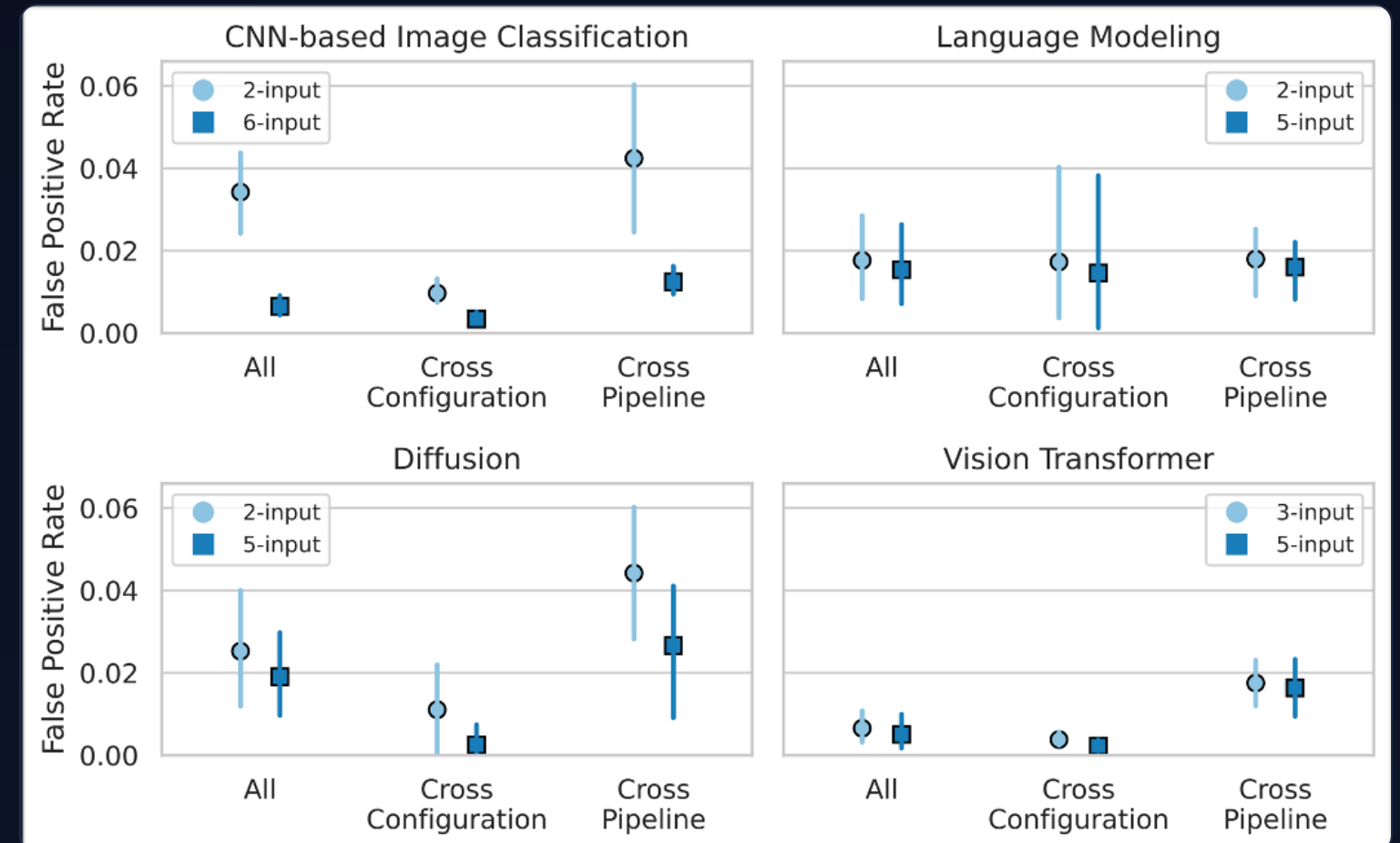
False positive rate
across 63 representative pipelines

Cross-configuration and cross-pipeline. Consistently low with 5 representative input pipelines.

18/20

Real-world silent errors
detected within 1 iteration

Pinpoints exact root cause in 10 cases, close to root cause in 8 more.
6 new bugs exposed in DeepSpeed and Transformers.

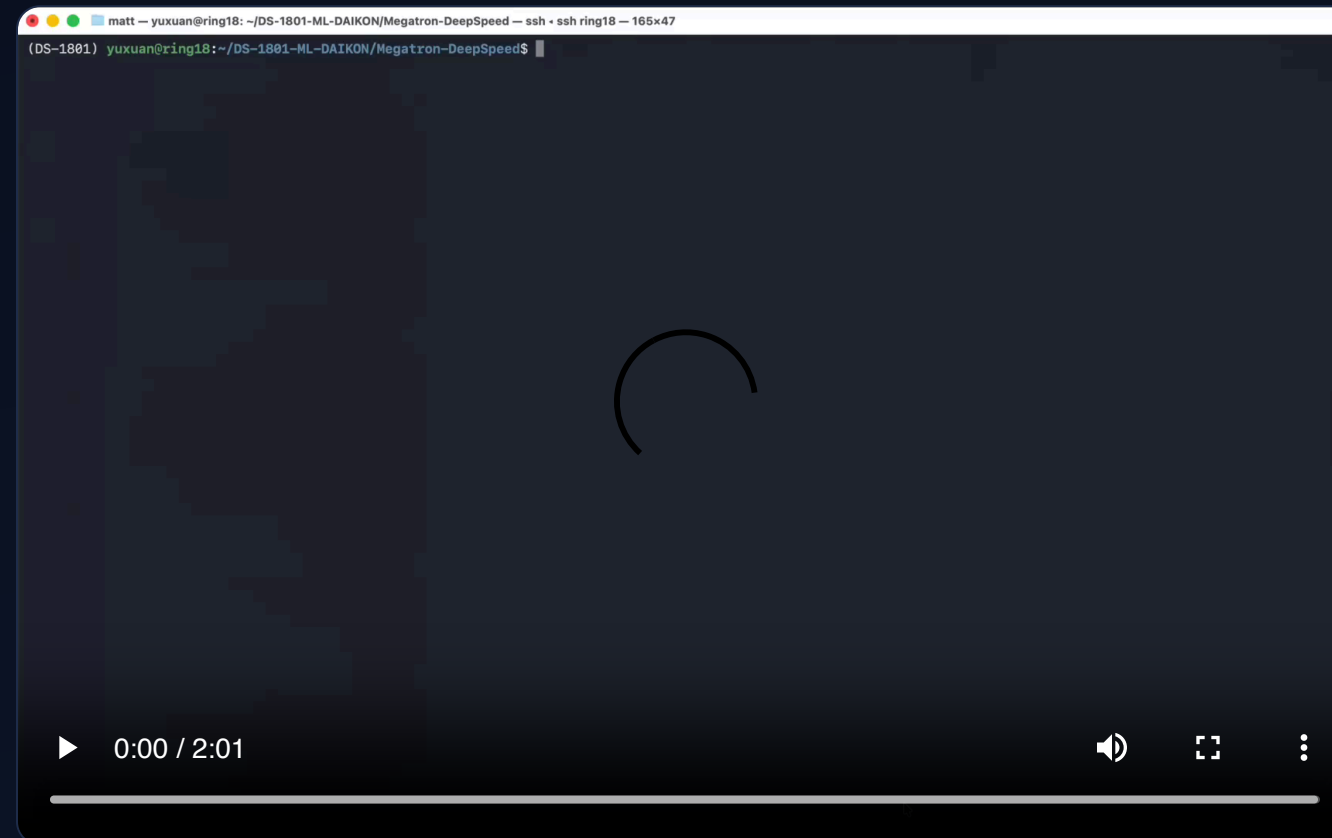


FP rate by task type — consistently under 2% with 5-input pipelines

DEMO

Demo: catching issues early.

Using the BLOOM parameter divergence bug. Violation appears within the first few steps.



Show: violation appears at step 1—5 vs baseline (no signal for 200+ steps)

Where we are and where we are going.

What we support today

- PyTorch training pipelines (single-GPU, DDP, FSDP, Megatron)
- 6 invariant types: Consistency, APIContain, FunctionLead, FunctionCover, Threshold, DistinctArgument
- Online + offline checking with live dashboard
- W&B and MLflow integration with per-step violation tracking
- `pip install traincheck`

What's next

- LLM-specific training patterns
- Deeper distributed training invariants (TP, PP, ZeRO)
- Alerting pipelines (PagerDuty, Slack)
- JAX / Flax backend support
- Community invariant library

Real bugs that TrainCheck catches.

Two real-world training failures in detail.

CASE 1

BLOOM-176B Divergence

Gradient clipping bug in Megatron-LM causes silent weight divergence across TP ranks

384 A100 GPUs · 3.5 months · BF16 optimizer

CASE 2

Frozen Encoder

PyTorch MPS backend silently skips Adam updates on non-contiguous tensors

Apple Silicon · Sparse autoencoder · Referenced by Karpathy

BUG DESCRIPTION



Model weights silently diverged across GPUs.

BLOOM-176B · 384 A100s · 3.5 months of training · loss looked fine throughout.

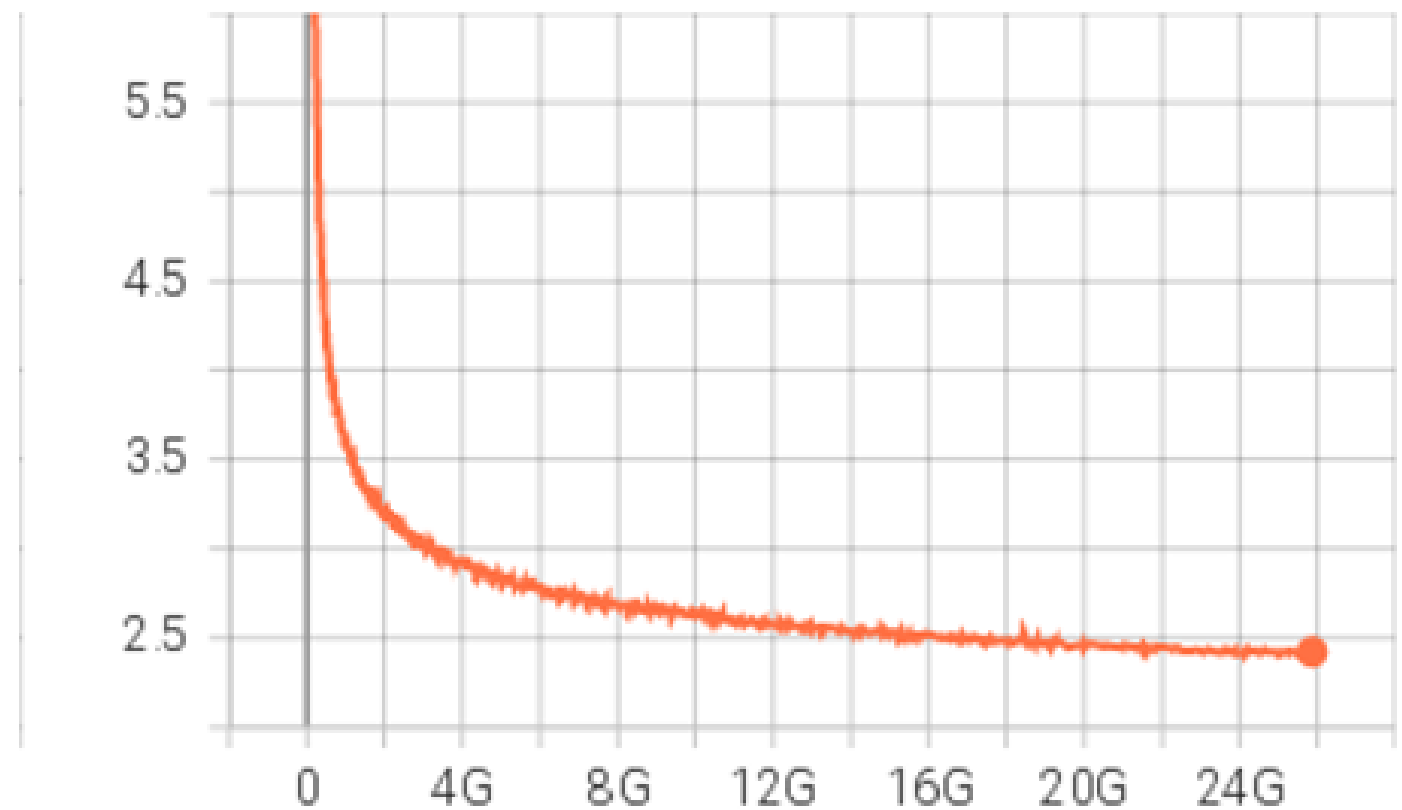
WHAT HAPPENED

- Software bug in Megatron-LM's gradient clipping code
- De-duplication logic misplaced: only first TP worker had correct gradient clipping
- BF16 optimizer — model weights silently diverged across tensor-parallel ranks
- Produced invalid checkpoints — discovered late in training

384 A100 GPUs · 3.5 months · Invalid checkpoints from silent weight divergence

lm-loss-training/lm loss vs tokens
tag: lm-loss-training/lm loss vs tokens

176B-ml



WHY IT WAS HARD TO DETECT

Everything was fine — until it wasn't.

The corruption was silent. Loss was healthy for 39,999 steps.

WHAT YOU SAW

- Loss steadily decreasing for 40k steps
- All GPUs reporting normal utilization
- No OOM, no exceptions, no crash
- Loss steadily decreasing across all training tokens

THE ACTUAL ROOT CAUSE

```

@torch.no_grad()
def get_grads_for_norm(self, for_clipping=False):
    grads = []
    tensor_mp_rank = bwc_tensor_model_parallel_rank(mpu)
    for i, group in enumerate(self.bf16_groups):
        for j, lp in enumerate(group):
            if not for_clipping:
                if hasattr(lp, PIPE_REPLICATED) and lp.
                    continue

            if not (tensor_mp_rank == 0 or is_model_parallel_rank_0):
                continue # YUXUAN: as compared to the original code, this line is moved out by one indentation

            if not self.fp32_groups_hasattr(lp, FP32_GROUPS_HASATTR):
                continue

            grads.append(self.fp32_grads[i * self.tensor_model_parallel_world_size + j])

    return grads

```

1. for clipping = False →
collect gradients to compute norm
(de-duplication needed)

2. for clipping = True →
collect gradients to be clipped (all
gradients needs to be clipped)

The de-duplication logic is misplaced to
for clipping == True

De-duplication logic placed under for_clipping==True instead of
for_clipping==False

By the time you see the spike, you need to roll back to a checkpoint 5,000+ steps earlier.

HOW TRAINCHECK CATCHES IT

Parameter consistency invariant fires at step 1.

RELEVANT INVARIANT TYPE

CONSISTENCYRELATION

Consistent(Parameter.data, Parameter.data)

precond: UNEQUAL(TP_RANK) $\wedge$ EQUAL(step) $\wedge$ EQUAL(name)

ALSO DETECTABLE VIA

- **APICONTAINRELATION** — gradient clipping should be applied across all TP ranks
- Only non-tensor-parallel parameters (e.g., LayerNorm) need the consistency check

// TrainCheck output — early in training

[STEP 12] VIOLATION

Invariant: ConsistencyRelation

Rule: Consistent(Parameter.data, Parameter.data)

Precond: UNEQUAL(TP_RANK) $\wedge$ EQUAL(step) $\wedge$ EQUAL(name)

Layer: transformer.h.0.ln_1.weight

TP rank 0 vs 1: values differ (should be identical)

↳ Detectable from step 1 — long before any visible loss symptom

Detectable from step 1. The divergence visible in the loss curve is the *consequence*, not the first signal.

BUG DESCRIPTION

The encoder trained fine. Except it didn't train at all.

A silent bug in PyTorch's MPS backend caused Adam to silently skip encoder weight updates.

THE SETUP

- Training a sparse autoencoder on Apple Silicon (M3)
- Encoder weight initialized as transposed copy of decoder: `encoder.weight = decoder.weight.T.clone()`
- This creates a non-contiguous memory layout

Adam's `addcmul_` operation silently failed on non-contiguous tensors in the MPS backend.

`exp_avg_sq` remained zero. Weights never updated. No error. No warning.

TIMELINE OF CONFUSION

- 1 **Loss decreasing**
Decoder weights updating normally — overall loss still improves
- 2 **Model appears to train**
Gradients flow correctly to encoder — just not applied
- 3 **Manual weight norm inspection reveals the truth**
Hours later — encoder weights identical to initialization

WHY IT WAS HARD TO DETECT

Gradients were correct. Loss was decreasing. Everything looked fine.

STANDARD SIGNALS SAID

- Gradients computed for encoder ✓
- No NaN or Inf values ✓
- Loss decreasing ✓
- No exceptions ✓

≠

REALITY WAS

- Encoder weights frozen at init ✗
- Adam's second moment (`exp_avg_sq`) = 0 ✗
- `Optimizer.step()` called but silently a no-op for encoder ✗
- Decoder compensating for the frozen encoder

Eventually loss plateaus and you know something is wrong. But diagnosing it means manually inspecting parameter updates across the entire model to find which weights stopped moving.

Invariant fires at **step 1**.

RELEVANT INVARIANT TYPE

APICONTAINRELATION

"optimizer.step()" should result in a non-trivial update to *encoder.weight*" — inferred from healthy run where encoder updated every step

ALSO FLAGGED BY

- **CONSISTENCYRELATION** — weight value unchanged across consecutive steps
- Optimizer state (exp_avg_sq) tracks as zero — violates post-first-step baseline

```
// TrainCheck output — step 1
```

[STEP 1] VIOLATION

Invariant: APIContainRelation

Rule: encoder.weight changes after optimizer.step()

Expected: **weight delta > 0**

Observed: **encoder.weight unchanged (delta = 0.0)**

Stage: train

↳ Alert at step 1 — before any training time is wasted

Zero debugging time wasted. The violation is at step 1, with the exact parameter name.

Takeaways

- **Training can fail silently** — loss curves and GPU utilization don't tell the full story
- **Debugging is too late** — by the time you are debugging, you've already lost compute
- **Correctness signals are missing** — we monitor activity, not whether training is correct
- **Monitoring is the right abstraction** — invariant-based checks, applied at runtime, catch failures early

CLOSING

Training is becoming production infrastructure.

It deserves **SRE-style correctness monitoring**.

The same discipline we apply to production services — invariants, continuous checking, immediate alerts — applies to training too.

GET STARTED

Add signals before you need debugging.



github.com/OrderLab/TrainCheck

Install

```
pip install traincheck
```

Source

github.com/OrderLab/TrainCheck

Docs & Blog

orderlab.io/TrainCheck

Questions? traincheck@umich.edu · orderlab.io