Training-Control-as-Code: Towards a declarative solution to control training

Padmanabha V. Seshadri, Harikrishnan Balagopal, Mehant Kammakomati, Ashok Pon Kumar, Dushyant Behl
IBM Research India,
seshapad@in.ibm.com, harikrishnan.balagopal@ibm.com, mehant.kammakomati2@ibm.com,
ashokponkumar@in.ibm.com, dushyantbehl@in.ibm.com

***Abstract*—*Training-as-a-service* platforms facilitate users to deploy pre-configured Generative AI training jobs as batch workloads. The immutability of configuration offers minimal flexibility to dynamically adapt to training progress. Existing approaches invariably involve manually monitoring training progress on a dashboard, and the *stop-reconfigure-restart* of training does not scale well with number of experiments. Relying on pre-configuration, wastes computational resources and makes debugging of training jobs difficult. We address this gap through our *training-control-as-code* paradigm, which allows users to run user-defined code to analyze the training state and intervene to flag anomalies and save resource wastage. Our framework `TrAC` offers a declarative interface to allow for declaring desired control and for reusing it at scale. Using real-world open-source data and models we provide estimates on the savings in time and resource due to `TrAC`. We also provide demo video: https://youtu.be/RmhBfFjd1oA and code: https://github.com/foundation-model-stack/fms-hf-tuning/blob/main/examples/trainercontroller_configs/Readme.md**

***Index Terms*—Generative AI, Training, Declarative framework**

I. Introduction

The advent of Generative Artificial Intelligence (GenAI) has triggered the growth of training-as-a-service [1], [2] business model, wherein training infrastructure and stack is offered to users and enterprises to train models to suit their requirements. These training experiments could have a large vertical scale (a single experiment [3] with very large data and infrastructure requirements) or horizontal scale (many experiments spawned to tune hyper-parameters, explore data space [4], or cater to different use-cases).

Training (For a background tutorial, please refer to [5]) GenAI models could be an iterative task wherein anomalies [6] could occur within the training loop (e.g. oscillating loss), data (E.g noisy data), or infrastructure (e.g. transient GPU failures). Letting a training run with these anomalies could waste resources and also increase the time to debug the training state, data and configuration. Other avenues for wastage include over-checkpointing when the training has not progressed much, or when the training is allowed to continue when the loss has already converged.

Real-word use-case Figure 1 illustrates a training anomaly reported in OPT175B model training logbook [6]. The logbook reports (page 25 and 91) that the job owner monitored the *grad-norm* and *training loss perplexity* manually using *Tensorboard* dashboard [7] and took remedial actions (stop the training in the first case, and log the anomaly in the second). This training experiment logbook spanned 3 months with several job owners taking turns to monitor the experiment and enact remedial action. `TrAC` is designed to automate such manual processes so as to reduce debugging time and resource wastage. Figure 3 illustrates a `TrAC` configuration which automates this use-case.

Tools similar to *Tensorboard* observe training processes [8], [9] manually do not scale well with volume of experiments. Besides, these frameworks do not allow for customized user-defined analysis of observed state and detection of anomalies, and customized on-the-fly intervention. There are also some rudimentary solutions like early-stopping feature [10] which allow for intervention (by stopping the training). However, this feature is limited to only stopping the training, and hard-wired for one metric. It lacks the capability to generalize and customize across the plethora of scenarios mentioned above.

In this paper, we propose *Training-control-as-code* (`TrAC`) as a novel concept to offer a generalized approach to automate training control, so as to reduce the inefficiencies due to manual intervention or static pre-configuration in training-as-a-service platforms. Model developers running several ablation experiments, LLMOps engineers involved in continuous model improvement could plug-in `TrAC` as part of their training stack to automate training control. Our contributions are as follows:

1) *Customizable open-sourced framework ([11])*: `TrAC` supports a declarative approach to analyze training process state and trigger custom intervention. We also discuss two out-of-the-box features in `TrAC`, which facilitate (1) Analyzing multi-time-scale training signals for early-stopping and checkpointing (2) Multi-granularity data signal inspection (e.g across batches, at a single batch level, samples within a batch and token level).
2) *Evaluation on real-world data and models*: We demonstrate the efficacy of `TrAC` (adopting rule-driven early-stopping and relevant checkpointing) as compared to the pre-configured (number of epochs: 4, checkpointing frequency: 0.25 epoch), uncontrolled baseline approach on training runs involving 4 real world models and 5 datasets. Compared to the uncontrolled baseline, `TrAC` leverages user-declared abstract checkpointing conditions to reduce storage wastage (using only half of the original storage), user-declared end-state configuration to detect if the training state has reached its goal and

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00384

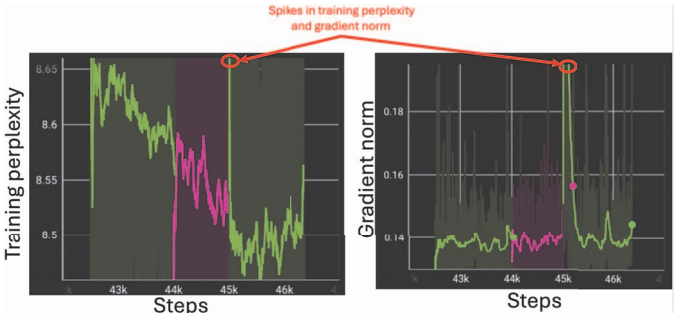


Fig. 1: Training perplexity and gradient norm spike during a OPT175B training **run 12.44** of OPT175B training [12], marked as green line.

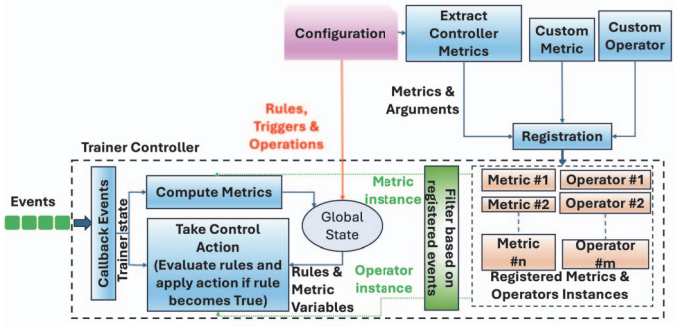


Fig. 2: TrAC architecture.

stop training appropriately leading to 9x less GPU time.

Prior work The background work for training control is summarized in Table I. *Early-stopping* [10] forms a large body of work, focusing on when to stop a training based on the behavior of training and its progress. Berta et al [13] proposes training stoppage to minimize refinement error (ability to classify effectively between classes), Yuan et al [14] proposes a metric based on second-order differential of the loss computed for a training instance or observe overfitting behavior [15] with respect to mislabeled data, to shorten the training. These approaches do not provide a declarative interface to hook into different layers of the training stack, accumulate and process training state at multiple time-scales and data granularities. TrAC provides this capability even for non-early-stopping scenarios, while also subsuming existing methods.

There are also training frameworks like TENPLEX [16] and Alibaba HPN [17] which offer a elastic migration of workloads across GPUs. TrAC is again complimentary to such platforms and could work with these platforms by providing insight into the training workloads (e.g. if all jobs using a node in a GPU cluster slow down, then TrAC could potentially flag the node).

II. Architecture

Challenges To reduce the dynamics of a training process to code, the *stack-to-user* and *user-to-stack* communication needs to be abstracted into symbolic code. To convert *stack-to-user* communication to code, the training state needs to be aggregated from multiple layers and time-scales (sub-step, step, epoch) of the training stack (training, data and infrastructure signals) and presented to the user symbolically so that he

```
1  controller_metrics:
2    - name: training_loss_window
3      class: HistoryBasedMetric
4      arguments:
5        window_size: 5
6  controllers:
7    - name: stop_on_training_loss_not_decreasing
8      triggers:
9        - on_log
10     rule: training_loss_window["training_loss"]["loss"][0] <
       training_loss_window["training_loss"]["loss"][-1] and
       training_loss_window["training_loss"]["grad_norm"][0] <
       training_loss_window["training_loss"]["grad_norm"][-1] and len
       (training_loss_window["training_loss"]["loss"]) ==
       training_loss_window["window_size"]
11     operations:
12       - hfcontrols.should_training_stop
```

Controller metric which maintains a history of last five samples of training loss

User-defined rule

Event that triggers the rule to be evaluated

Action to be performed if rule is TRUE

Fig. 3: TrAC configuration to stop training if the training loss and grad-norm are non-decreasing. This configuration captures the OPT175B training anomaly from Figure 1.

can define his own processing logic on it. This would require components of trainer controller to hook onto the events across the training stack and channel it to the user efficiently. In the reverse direction, i.e *user-to-stack* communication, the user should be able to define his custom intervention symbolically which can then be enacted in different parts of the training stack to realize the desired effect. E.g a evaluate-test-checkpoint-stop operation should first signal an evaluation to be triggered, and based on some user-defined criteria defined on evaluation metric, the stack has to then checkpoint the state, and then stop the training in that order.

We address the above challenges through a declarative interface to simplify user's effort to define the state to be observed, analysis to be performed, and define rules representing the conditions for action, and modular plugin framework to analyze the training state spread out across multiple layers and even time-scales (e.g. sub-steps, pre-step, post-step, epochs etc) and intervene in a user-defined, ordered and composite manner (e.g stopping a training, checkpoint and stop a training, expose some internal state in the log around an anomalous event, increase rate of checkpointing etc).

Internals Fig 2 illustrates the architecture of TrAC. The framework consumes a user-declared configuration which contains four elements: **1)** Declaration of custom computation on the state of the training process. We call this a *controller-metric*. The controller metric accumulates information from various layers of the stack during various training events and stores it in an representation exposed to the user-rule. The representation could be raw-data or numerical summary of the data. **2)** Declaration of the operation to be performed when a certain condition occurs. A compositive operation could be broken down into unitary operations chained in sequence. **3)** Rules which contain the *controller-metrics* and define the condition at which the operation (some examples are listed here [18]) to be performed, and finally **4)** Trigger events which cause a rule to be validated. An illustration is shown in Figure 3.

Implementation TrAC is implemented using the callback framework (in this particular instance, it is the HuggingFace trainer callback [18]), which provides event hooks to capture training process events, e.g before or end of a step or

Framework	Training state observation	User-defined training state analysis	User-defined intervention
TrAC	✓	✓	✓
Early stopping techniques	limited	×	×
Dashboards like Tensorboard, Aim and Wandb	✓	limited	×
Training and data signal agnostic frameworks like TENPLEX/HPN	×	×	×

TABLE I: Comparison of TrAC with relevant framework. **Training state** is a broader term which encompasses the training signals, infrastructure condition and data handling state.

epoch(*complete pass of the training data*). On occurrence of an event, every *controller-metric* registered to it, are executed on the global state of the training. The result is stored in a global cache, to be then used by rules referring to the output generated by the metrics. We leverage a *restricted python syntax* offered by *simple eval* [19] to safely execute rules. The rules associated with the event are validated, if any of them evaluate to true, then the corresponding *operation* is executed.

We also modified the HuggingFace trainer callback [18] to surface individual batch data and computed loss at the sample level at the end of a forward-pass of the model (*step*), which is otherwise not supported natively.

Out-of-box features We provide following *controller-metrics* which cater to multi-scale analysis with the TrAC:

1) **Data batch inspector**: This metric provides a two-level analysis of a data batch. First, batches are thresholded by training loss. The filtered (bad batch) is analyzed for sample-leve spikes by using the logits to recompute loss.
2) **History-based training signal analyzer**: This metric maintains multi-time-scale moving window of training signal samples (training/validation loss or even evaluation scores) which can then be used within a rule along with aggregate functions (e.g min, max, avg etc) or conditionals as illustrated in the sample config [20].

User interaction with TrAC In the simplest form, the user just needs to define a configuration as shown in Figure 3 selecting an out-of-box *controller-metric*, trigger event and operation. Only the rule (For instance, a rule to check if the training loss is less than 1.0, could be executed at the end of every step) can be user-defined. Once defined, the user passes this configuration YAML as a file in the training arguments list for the training stack.

If the user wants to define a custom *controller-metrics*, he could simply extend a *MetricHandler* class [11] to create a custom metric. This requires overriding a validation method which ensures that all the inputs to the metric are in the right format, and computation method which has the analysis logic to handle the inputs. This custom metric can be registered with TrAC using the class name to ensure plug-and-play.

Similarly, user can define a custom operation by overriding a *Operation* class to introduce custom operations which can contain signals to various layers of the stack to perform actions.

III. Evaluation

Our evaluation compares the performance a suite of parallel training workloads controlled with TrAC and without any

```yaml
controller_metrics:
  - name: eval_loss_window
    class: HistoryBasedMetric
    arguments:
      window_size: 10
controllers:
  - name: save_when_eval_drop_15
    triggers:
      - on_step_end
    rule: len(eval_loss_window["metrics"]["eval_loss"]) > 1 and eval_loss_window["metrics"]["eval_loss"]
    [-1] <= 0.85 * eval_loss_window["metrics"]["eval_loss"][-2]
    patience:
      patience_threshold: 1
    operations:
      - hfcontrols.should_save
  - name: checkpoint_when_eval_conseq_10_steps_no_change
    triggers:
      - on_step_end
    rule: len(eval_loss_window["metrics"]["eval_loss"]) > 9 and 0.9 * eval_loss_window["metrics"]
    ["eval_loss"][-1] <= sum(eval_loss_window["metrics"]["eval_loss"])/len(eval_loss_window["metrics"]
    ["eval_loss"]) <= 1.1 * eval_loss_window["metrics"]["eval_loss"][-1]
    patience:
      patience_threshold: 10
    operations:
      - hfcontrols.should_save
  - name: stop_when_eval_conseq_10_steps_no_change
    triggers:
      - on_step_end
    rule: len(eval_loss_window["metrics"]["eval_loss"]) > 9 and 0.9 * eval_loss_window["metrics"]
    ["eval_loss"][-1] <= sum(eval_loss_window["metrics"]["eval_loss"])/len(eval_loss_window["metrics"]
    ["eval_loss"]) <= 1.1 * eval_loss_window["metrics"]["eval_loss"][-1]
    patience:
      patience_threshold: 20
    operations:
      - hfcontrols.should_training_stop
```

Fig. 4: TrAC configuration used for the experiments

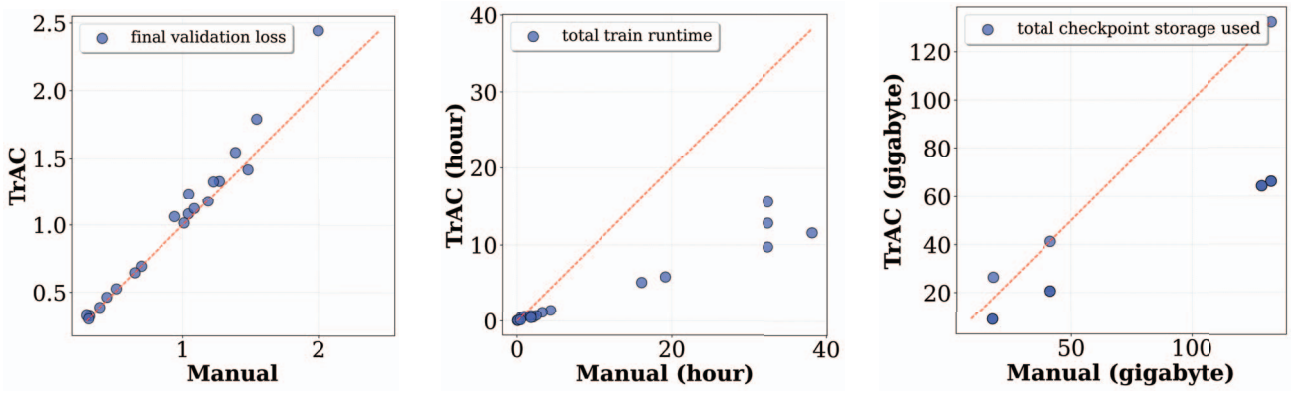
control (pre-configured manually), run on $4\times A100$ 80GB GPUs [21].

Configuration. The controller-metric used in the configuration (Figure 4) is *history-based* (Section II), i.e. the validation loss upto last 50 steps is retained for computation. The configuration defines three rules. **1)** ensures that a checkpoint is created when the validation loss drops at least by 15%. **(2)** Checkpoint when current step validation loss is close to the average validation loss for last 10 steps. **(3)** stops the training when the current step validation loss is close to the average validation loss for last 20 steps. All the rules are evaluated at the end of every step.

Workload and datasets We use *Granite* and *Llama* family models for our experiments and mix having smaller *granite-3.1-2b-base* [22] and *TinyLlama_v1.1b* [23] and larger *granite-3.1-8b-base* [24] and *Llama-3.1-8B* [25] models in our experimentation. We use publicly-available datasets with varied characteristics like, multiple choice questions (*allenai/ai2_arc* [26]), multi domain instruction-following question answering (*tatsu-lab/alpaca* [27]), translation (*Helsinki-NLP/opus_books - en-it subset* [28] and *Helsinki-NLP/opus_books - en-nl subset* [28]), and code tasks (*codeparrot: java-program-level subset* [29]).

In total, we conduct 20 experiments encompassing all combinations of aforementioned models and datasets, demonstrating TrAC efficacy in various training use cases.

Metrics We select training quality using *validation loss* computed using separate validation set in each experiment.



(a) Validation loss. (b) Training time. (c) Storage size of checkpoint.

Fig. 5: Performance with (marked **TrAC**) and without TrAC (marked **Manual**).

The system performance of the training is measured by the *training time* and *storage space* occupied by the checkpoints.

Results Figure 5a illustrates the comparative task performance of the training experiments with respect to validation loss metric. In 90% of the experiments, TrAC results in a validation loss that is within 15% of the loss observed with manual method.

Figure 5b shows that TrAC achieves a 9x reduction in training time, while staying within 10% of the validation loss (manual method) in 70% of the experiments. These gains could potentially translate to substantial cost savings for the users. For instance, with Runpod GPU pricing [2], the total GPU cost of running all 20 experiments without TrAC and with our GPU configuration could be about 3600 USD, while using TrAC reduces the cost to 324 USD. Such cost savings can vary across the service providers.

Figure 5c shows that the storage consumed by checkpointing with TrAC on an average less by 50%.

IV. Future Work

TrAC provides a declarative and extensible framework which can generalize the control of a training process at scale. Compared to an uncontrolled, pre-configured training run, TrAC only 50% less storage and avoids the unnecessary training cyles reducing training time by 9x by helping users define anomalies and stop training. TrAC opens some interesting research directions, such as: **1)** Intelligent pre-fetching and just-in-time allocation of resources and data based on the state of the training, **2)** Planning reactive and dynamic data patching to address in-training data anomalies, **3)** Debugging and patching of model parameters [30] based on discovered in-training anomalies. We expect TrAC approach to bring in a new perspective in managing training processes at scale and trigger interest in academic and industry communities.

References

[1] "Amazon SageMaker Model Training." https://aws.amazon.com/sagemaker-ai/train/.

[2] "Runpod A100 GPU pricing." https://www.runpod.io/gpu/a100-pcie.

[3] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[4] "What is model tuning?." https://www.ibm.com/think/topics/model-tuning.

[5] "Fine-tuning tutorial." https://www.kaggle.com/code/ysthehurricane/step-by-step-huggingface-model-fine-tune-guide.

[6] "OPT Model Log Books." https://github.com/facebookresearch/metaseq/blob/main/projects/OPT/chronicles/OPT_Baselines_Logbook.pdf.

[7] "Tensorboard." https://www.tensorflow.org/tensorboard.

[8] "AIMStack." https://aimstack.io/.

[9] "Wandb." https://wandb.ai/site/.

[10] "Hugging Face Early Stopping Callback." https://hf.co/docs/transformers/en/main_classes/callback#transformers.EarlyStoppingCallback.

[11] "TrAC code." https://github.com/foundation-model-stack/fms-hf-tuning/blob/main/examples/trainercontroller_configs/Readme.md.

[12] "OPT175B Log book." https://github.com/facebookresearch/metaseq/blob/main/projects/OPT/chronicles/OPT175B_Logbook.pdf.

[13] E. Berta, D. Holzmüller, M. I. Jordan, and F. Bach, "Rethinking early stopping: Refine, then calibrate," *arXiv preprint arXiv:2501.19195*, 2025.

[14] S. Yuan, R. Lin, L. Feng, B. Han, and T. Liu, "Instance-dependent early stopping," *arXiv preprint arXiv:2502.07547*, 2025.

[15] S. Yuan, L. Feng, and T. Liu, "Early stopping against label noise without validation data," *arXiv preprint arXiv:2502.07551*, 2025.

[16] M. Wagenländer, G. Li, B. Zhao, L. Mai, and P. Pietzuch, "Tenplex: Dynamic parallelism for deep learning using parallelizable tensor collections," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pp. 195–210, 2024.

[17] K. Qian, Y. Xi, J. Cao, J. Gao, Y. Xu, Y. Guan, B. Fu, X. Shi, F. Zhu, R. Miao, *et al.*, "Alibaba hpn: A data center network for large language model training," in *Proceedings of the ACM SIGCOMM 2024 Conference*, pp. 691–706, 2024.

[18] "Trainer callback." https://hf.co/docs/transformers/en/main_classes/callback#transformers.TrainerCallback.

[19] "Simple eval." https://pypi.org/project/simpleeval/.

[20] "TrainCoCo config file used in the evaluation." https://gist.github.com/anonymouspaperauthors/a84114364dc47b8161ff46e19b69ac84.

[21] "A100 80GB GPU." https://www.nvidia.com/en-in/data-center/a100/.

[22] "granite-3.1-2b-base model card." https://hf.co/ibm-granite/granite-3.1-2b-base.

[23] "TinyLlama/TinyLlama_v1.1 model card." https://hf.co/TinyLlama/TinyLlama_v1.1.

[24] "granite-3.1-8b-base model card." https://hf.co/ibm-granite/granite-3.1-8b-base.

[25] "meta-llama/Llama-3.1-8B." https://hf.co/meta-llama/Llama-3.1-8B.

[26] "allenai/ai2_arc data card." https://hf.co/datasets/allenai/ai2_arc.

[27] "tatsu-lab/alpaca data card." https://hf.co/datasets/tatsu-lab/alpaca.

[28] "Helsinki-NLP/opus_books." https://hf.co/datasets/Helsinki-NLP/opus_books.

[29] "codeparrot/xlcost-text-to-code data card." https://hf.co/datasets/codeparrot/xlcost-text-to-code.

[30] M. Geva, A. Caciularu, G. Dar, P. Roit, S. Sadde, M. Shlain, B. Tamir, and Y. Goldberg, "Lm-debugger: An interactive tool for inspection and intervention in transformer-based language models," pp. 12–21, 01 2022.