

000 001 002 003 004 005 006 007 008 009 010 ML-AGENT: REINFORCING LLM AGENTS FOR 002 AUTONOMOUS MACHINE LEARNING ENGINEERING

005 **Anonymous authors**

006 Paper under double-blind review

009 ABSTRACT

011 The emergence of large language model (LLM)-based agents has significantly
 012 advanced the development of autonomous machine learning (ML) engineering.
 013 However, the dominant prompt-based paradigm exhibits limitations: smaller mod-
 014 els lack the capacity to learn from execution trajectories for generalization, while
 015 large proprietary models incur high computational overhead, restricting accessi-
 016 bility and scalability. Focusing on this, for the first time, we explore the paradigm
 017 of learning-based agentic ML, where an LLM agent learns through interactive
 018 experimentation on ML tasks using online reinforcement learning (RL). To realize
 019 this, we propose a novel agentic ML training framework with three key compo-
 020 nents: (1) exploration-enriched fine-tuning, which enables LLM agents to generate
 021 diverse actions for enhanced RL exploration; (2) step-wise RL, which enables
 022 training on a single action step, accelerating experience collection and improving
 023 training efficiency; (3) an agentic ML-specific reward module, which unifies varied
 024 ML feedback signals into consistent rewards for RL optimization. Leveraging
 025 this framework, we train ML-Agent, driven by a 7B-sized Qwen-2.5 LLM for
 026 autonomous ML. Despite training on only 9 ML tasks, our 7B-sized ML-Agent
 027 achieves comparable performance to agents using much larger proprietary LLMs
 028 (e.g., GPT-5) but at significantly lower computational cost, demonstrating strong
 029 performance and cross-task generalization.

030 1 INTRODUCTION

031 Machine Learning (ML) engineering is a critical yet labor-intensive process, requiring expert re-
 032 searchers to invest significant time—potentially days or even months—designing architectures, tuning
 033 parameters, and iteratively refining models through trial and error (Bergstra & Bengio, 2012). This
 034 challenge has sparked an ambitious vision of autonomous ML: building autonomous AI systems that
 035 independently orchestrate the entire ML lifecycle, from conceptual design and code implementa-
 036 tion to refinement.

037 Fortunately, the advent of LLM-based agents, equipped with capabilities of interaction (Du et al.,
 038 2023; Pang et al., 2024), coding (Hong et al., 2023; Qian et al., 2023; Hu et al., 2024) and tool-
 039 calling (Masterman et al., 2024), has propelled us significantly closer to realizing this vision (Agentic
 040 ML) (Huang et al., 2023; Chan et al., 2024). Unlike traditional automated ML with pre-defined limited
 041 search and action spaces (Tang et al., 2024; LeDell & Poirier, 2020; Jin et al., 2023), these LLM agents,
 042 when provided with instructions in natural language, can autonomously propose effective actions,
 043 generate executable codes, and iteratively improve solutions based on environmental feedback (Huang
 044 et al., 2023; Jiang et al., 2025). For example, AIDE (Jiang et al., 2025) and ML-Master (Liu et al.,
 045 2025a) both leverage LLM agents together with experimental environments to automate ML process.

046 Currently, the dominant paradigm in agentic ML relies on prompt-based design, where agents are
 047 constructed through heuristic prompt engineering. This approach offers practical advantages, as it
 048 allows rapid deployment without parameter updates or extensive retraining. However, it also exhibits
 049 notable limitations: when driven with smaller language models, such agents lack the capacity to
 050 learn from and internalize execution trajectories, causing limited generalization across diverse tasks;
 051 conversely, when implemented with large-scale proprietary models, the paradigm incurs substantial
 052 computational overhead and resource consumption, thereby restricting accessibility and undermining
 053 sustainable scalability (Belcak et al., 2025).

054 To address these limitations, we propose moving beyond the prompt-based paradigm toward a new
 055 research trajectory: learning-based agentic ML. In this paradigm, agents are no longer constrained
 056 to static prompt instructions but instead learn adaptively from task-solving trajectories via online
 057 reinforcement learning (RL). Such a formulation empowers agents to systematically explore diverse
 058 strategies, accumulate knowledge across successive runs, and progressively refine their decision-
 059 making processes (Xiong et al., 2024). Importantly, this learning-based approach endows even
 060 relatively small language models with the capacity to achieve strong generalization, while substantially
 061 reducing computational and resource demands. As a result, it opens a more accessible, efficient, and
 062 sustainable path for advancing the frontier of autonomous machine learning.

063 While being straightforward, employing online RL to train autonomous ML agents poses three
 064 key challenges. (1) *Limited exploration*: agents often propose similar actions for the same ML
 065 task across runs, leading to narrow exploration trajectories in RL (Park et al., 2024). (2) *Slow*
 066 *experience collection*: ML experiments can take minutes to hours, making online RL data gathering
 067 inefficient and thus limiting feedback-driven training samples (Chan et al., 2024). (3) *Complex*
 068 *reward design*: agentic ML involves various outcomes, such as task-specific metrics, out-of-memory
 069 failures, and compilation errors. This requires a unified reward function to reconcile varied feedback
 070 signals (Eschmann, 2021).

071 In response to these challenges, we propose a novel agentic ML training framework, the first designed
 072 to train LLM agents for autonomous ML engineering using online RL. This framework enables agents
 073 to explore diverse ML trajectories, collect rewards efficiently, and iteratively enhance their capabilities
 074 through learned experience. (1) To improve exploration diversity, we introduce *exploration-enriched*
 075 *fine-tuning*, generating a diverse action pool from fast-executable ML tasks to finetune agents for
 076 broader RL exploration. (2) To accelerate experience collection, we design a *step-wise RL paradigm*,
 077 evaluating atomic actions using expert trajectories as single-step queries, significantly boosting
 078 training efficiency. (3) To tackle reward design, we develop an *agentic ML-specific reward module*
 079 that dynamically handles errors (e.g., runtime failures) and quantifies performance via normalized,
 080 task-specific metrics (e.g., accuracy gains).

081 By leveraging our proposed agentic ML training framework, we train ML-Agent, an agent driven by a
 082 7B-sized Qwen2.5 LLM for autonomous ML. During training, our ML-Agent can efficiently explore
 083 the environment, learn from experience, and achieve continuous performance improvement through
 084 iterative exploration across various ML tasks. Surprisingly, despite its modest size and training
 085 on only **9** ML tasks, ML-Agent demonstrates strong performance and cross-task generalization,
 086 outperforming 671B-sized DeepSeek-R1 agent on 3 held-in and 10 held-out tasks across diverse data
 087 modalities and objectives. Notably, it achieves results comparable to agents using the most advanced
 088 proprietary LLMs (GPT-5) but at significantly lower computational cost.

089 In summary, our work makes the following significant contributions to the field:

- 090 • We introduce a new paradigm for autonomous ML: learning-based agentic ML, where an LLM
 091 agent learns through interactive experimentation on ML tasks via online reinforcement learning.
- 092 • We propose a novel training framework for agentic ML, which incorporates three technical
 093 designs: exploration-enriched fine-tuning, step-wise RL, and agentic ML-specific rewards.
- 094 • Extensive experiments show that despite training on only 9 ML tasks, our 7B-sized ML-Agent
 095 surpasses agents driven by much larger LLMs and even matches agents driven by proprietary
 096 LLMs (e.g., GPT-5) with much lower cost.

098 2 RELATED WORK

100 **Autonomous Machine Learning.** Autonomous machine learning aims to automate the manual and
 101 expertise-intensive aspects of machine learning, including data preprocessing, model selection and
 102 hyperparameter tuning. Autonomous machine learning has evolved from classical hyperparameter
 103 and pipeline search to agentic frameworks powered by large language models. Classical autonomous
 104 machine learning frameworks focus on automating model selection, hyperparameter optimization,
 105 and pipeline construction within a fixed search space (Tang et al., 2024; Olson & Moore, 2016; Feurer
 106 et al., 2022; Mohr et al., 2018; Erickson et al., 2020; Liu et al., 2020). For example, AutoGluon-
 107 Tabular (Erickson et al., 2020) ensembles multiple models and stackings to deliver state-of-the-
 108 art performance on tabular data with minimal user effort. These classical autonomous machine

learning works remain constrained by predefined search spaces and static configurations, lacking the adaptability and continuous learning capabilities.

LLM Agents in Autonomous Machine Learning. Recent advancements in LLMs have empowered them to autonomously generate and refine machine learning solutions, opening new possibilities in machine learning. Methods such as AutoML-GPT (Zhang et al., 2023b) and MLCopilot (Zhang et al., 2023a) prompt LLMs to automate the entire machine learning pipeline, where MLCopilot introduces past experience retrieval to help decision-making. AIDE (Jiang et al., 2025) and ML-Master (Liu et al., 2025a) focus on optimizing the ML engineering process through iterative search and refinement strategies. Other works like AutoKaggle (Li et al., 2024) and AutoML-Agent (Trirat et al., 2024) employ a multi-agent framework to address ML problems. However, these approaches are fundamentally constrained by a prompt-based paradigm. While agents may leverage past experience, their underlying models are not trained on these interaction histories. Consequently, their problem-solving strategies remain static and rely on costly advanced models. This limitation motivates our shift toward a learning-based paradigm where agents adapt and improve over time.

Reinforcement Learning for LLMs. Reinforcement learning (RL) significantly enhances the ability of LLMs, particularly in preference alignment and complex reasoning (Xu et al., 2025; Wang et al., 2024; Zheng et al., 2023). By facilitating exploration and exploitation, RL trains LLMs to adapt and improve their policy based on feedback, thus refining their performance in dynamic environments. One line of work is preference optimization (Kaufmann et al., 2023), with methods such as Reinforcement Learning from Human Feedback (RLHF)(Ouyang et al., 2022). RL is also utilized to train LLMs for complex reasoning tasks (Guo et al., 2025a; Liu et al., 2025b). Another line of research involves training LLM agents for specific tasks using RL (Zhang et al., 2025). For example, IPR (Xiong et al., 2024) and AgentQ (Putta et al., 2024) use DPO (Rafailov et al., 2023) to iteratively refine their policy. While StarPO (Wang et al., 2025) discusses the multi-turn reinforcement learning considering episode-wise reward. However, applying RL to train LLM agents for autonomous machine learning remains unexplored.

3 PROBLEM SETUP AND PRELIMINARIES

Problem Formulation. Agentic ML leverages an LLM agent to autonomously orchestrate the ML lifecycle by interacting with the experimental environment. This environment includes editable task-related code files together with an interpreter executing code and provides explicit experimental feedback (e.g., code execution results or error messages). Given an initial ML task specification (e.g., dataset description and evaluation metric), the agent begins interacting with the environment to iteratively refine its solution. At each step, the agent takes actions (e.g., add BN layers in the model architecture) and receives feedback (e.g., code execution output or error messages) from the environment. This loop continues until a step or time limit is reached. We follow the action space from prior work (Huang et al., 2023) (The details are provided in Table 4).

Agentic ML as a MDP. We format agentic ML as a Markov Decision Process (MDP) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P})$, where \mathcal{S} is the state space, \mathcal{A} the action space and \mathcal{P} the state transition dynamics. Let the environment feedback at time t be $f_t \in \mathcal{F}$, where \mathcal{F} denotes the feedback space. We employ a history-based state representation $s_t = (s_0, a_0, f_0, a_1, f_1, \dots, a_{t-1}, f_{t-1})$ to capture richer contextual information from past feedback, in which s_0 encodes the initial ML task specification and each pair (a_i, f_i) represents the agent’s action and corresponding environment feedback. The agent policy π_θ generates an action $a_t \in \mathcal{A}$ conditioned on current state s_t , forming a trajectory of interactions $\tau = (s_0, a_0, s_1, \dots, a_{n-1}, s_n)$. Note that θ is the LLM’s parameters within the agent and n is the trajectory length. The goal is to maximize the expected trajectory reward:

$$\mathcal{J}(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} [R(\tau)], \quad (1)$$

where the reward function $R(\tau)$ denotes the cumulative reward over the entire trajectory.

Challenges. Although the formulation of agentic ML is relatively straightforward, employing online RL to train LLM agents for autonomous machine learning poses several key challenges, including:

(1) Limited exploration. Agents often repeat similar actions across episodes, narrowing their exploration and limiting their ability to discover innovative ML solutions. **(2) Slow experience collection.** ML experiments can take minutes to hours, slowing down the online data collection process for RL training. **(3) Complex reward design.** Agentic ML produces varied outcomes (e.g., execution results

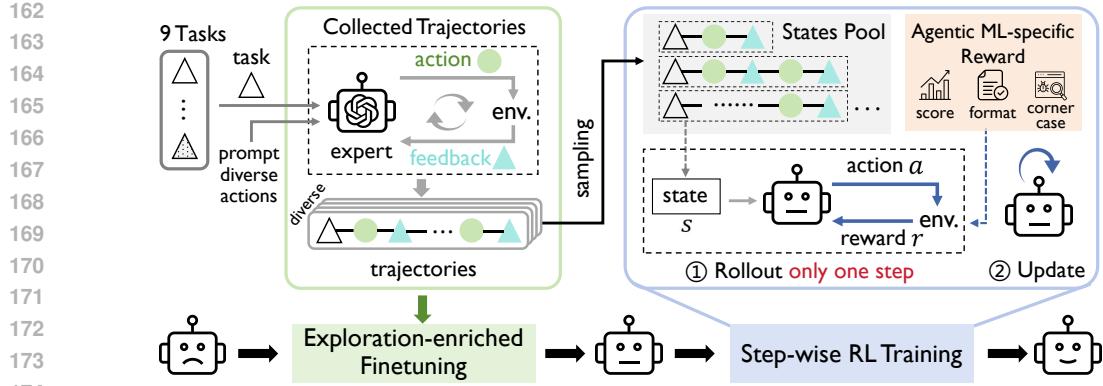


Figure 1: Overview of the agentic ML training framework, introducing (1) exploration-enriched fine-tuning for diverse action pool creation, (2) a step-wise RL paradigm for efficient experience collection using expert trajectories, and (3) an agentic ML-specific reward module for various ML feedback handling and task-specific performance evaluation.

or resource errors), making it challenging to design a unified reward function that effectively guides the agent. The subsequent section presents our agentic ML training framework designed to overcome these challenges, with the overall architecture illustrated in Figure 1.

4 AGENTIC ML TRAINING FRAMEWORK

Our agentic ML training framework is designed to train LLM agents for autonomous machine learning. As shown in Figure 1, it comprises three key steps for effective learning. First, *exploration-enriched fine-tuning* builds a diverse action pool to enhance RL exploration. Second, a *step-wise RL paradigm* uses expert trajectories as single-step queries to accelerate experience collection in RL. Third, an *agentic ML-specific reward module* handles errors and quantifies agentic ML task-specific performance. These steps sequentially enable diverse exploration, efficient training, and unified feedback, enabling agents to iteratively improve agentic ML performance across varied ML tasks.

4.1 EXPLORATION-ENRICHED FINE-TUNING

In agentic ML, limited exploration hinders autonomous machine learning workflows. Agents often repeat similar actions (e.g., small code edits) across episodes, leading to narrow exploration and preventing the discovery of innovative architectures or optimization strategies.

To address this, we introduce exploration-enriched fine-tuning with an automated data collection pipeline. It organizes ML optimization strategies into 3 semantic categories: data, model and learning. For each category, an LLM generates a large set of candidate ideas and an embedding-based diversity filter selects a compact and diverse pool. During trajectory generation, the system automatically samples 1–3 categories, shuffles their order, and draws one idea from each corresponding pool to form the initial action sequence (See Appendix B.1). An expert LLM with policy π_e then executes the full workflow on fast-executable ML tasks, producing expert trajectories $\mathcal{D} = \{\tau^{(i)}\}_{i=1}^{|\mathcal{D}|}$. We fine-tune the agent policy π_θ via supervised fine-tuning (SFT):

$$\mathcal{L}_{\text{SFT}}(\theta) = -\mathbb{E}_{\tau \sim \mathcal{D}} [\log P_{\pi_\theta}(\tau | s_0)] = -\mathbb{E}_{\tau \sim \mathcal{D}} \left[\log \prod_{t=0}^{n-1} \pi_\theta(a_t | s_t) \right] = -\mathbb{E}_{\tau \sim \mathcal{D}} \left[\sum_{t=0}^{n-1} \log \pi_\theta(a_t | s_t) \right]. \quad (2)$$

This exploration-enriched fine-tuning approach preserves action format compliance while enabling agents to learn diverse strategies, significantly broadening the exploration scope in subsequent RL.

4.2 STEP-WISE RL PARADIGM

Objective. Due to the time-consuming nature of AI experiments, directly applying RL methods (e.g., PPO) is impractical, as sampling a single trajectory during rollout takes hours. To address

this issue, we propose a step-wise RL approach that reformulates the objective function equation 1, where we sample only a single step of action during the rollout phase instead of the entire trajectory. This approach extensively reduces the computational cost of the rollout phase and makes the overall training process more efficient. Specifically, we expand equation 1 into steps according to the state distribution $d^{\pi_\theta}(s)$:

$$\mathcal{J}(\theta) = \sum_{t=0}^{n-1} \sum_{s_t \in \mathcal{S}} d^{\pi_\theta}(s_t) \left[\sum_{a_t \in \mathcal{A}} \pi_\theta(a_t | s_t) R(s_t, a_t) \right], \quad (3)$$

where $R(s_t, a_t)$ is the step-wise reward at time t , and $d^{\pi_\theta}(\cdot)$ is the state distribution at time t under policy π_θ . This distribution can be calculated recursively based on the policy π_θ and the state transition dynamics \mathcal{P} ; see Appendix A for details. The time-consuming components in equation 3 include: 1) $d^{\pi_\theta}(s_t)$, which involves multiple state transition dynamics from s_t to s_{t+1} , and 2) $R(s_t, a_t)$, where the reward is determined based on feedback from the environment (e.g., code execution platform). Since $d^{\pi_\theta}(\cdot)$ relies on π_θ , the trajectory sampling process operates repeatedly in standard RL training, making the computational cost even higher. However, using d^{π_θ} to sample state distribution is not necessary for two reasons: 1) π_θ poorly aligns with the environment format during the early stage of RL training, hindering effective state exploration; 2) Once π_θ can interact properly with the environment, the set of states it could explore tends to vary only slightly as π_θ updates. Hence, we sample the states from a states pool according to a fixed expert distribution $d^{\pi_e}(s_t)$, which forms the step-wise objective function

$$\mathcal{J}_{\text{step}}(\theta) = \sum_{s_t \in \mathcal{S}} d^{\pi_e}(s_t) \left[\sum_{a_t \in \mathcal{A}} \pi_\theta(a_t | s_t) R(s_t, a_t) \right] = \mathbb{E}_{s_t \sim d^{\pi_e}, a_t \sim \pi_\theta(\cdot | s_t)} [R(s_t, a_t)]. \quad (4)$$

This objective function $\mathcal{J}_{\text{step}}(\theta)$ reformulates multi-step trajectory RL into step-wise training. This reformulation offers two advantages: 1) The state sampling process is decoupled from the RL of the model. This allows us to directly sample states from a pre-collected set and avoids expensive online sampling during training, significantly reducing the overall training time. 2) The state sampling process is performed before RL training, rather than during the rollout phase. This enables us to perform extensive sampling from the expert distribution, making training more scalable.

Training approach. Based on the step-wise RL formulation, our goal is to maximize the expected reward $R(s_t, a_t)$ shown in equation 4 according to the state distribution d^{π_e} and π_θ . This aligns with the approach used in RLVR methods (Guo et al., 2025b), where the policy represents a token generation process and $R(s_t, a_t)$ is the outcome reward of $\pi_\theta(a_t | s_t)$. Hence, any RL training approach can be applied to this objective $\mathcal{J}_{\text{step}}(\theta)$. For our implementation, we choose PPO (Schulman et al., 2017) as the training algorithm because of its widespread use and proven effectiveness. Specifically, suppose we expand the token generating process of $\pi_\theta(a_t | s_t)$, our PPO loss function can be defined as follows:

$$\mathcal{J}_{\text{step}}^{\text{PPO}}(\theta) = \mathbb{E}_{s \sim d^e, o_{\leq i} \sim \pi_{\theta_{\text{old}}}(\cdot | s)} \left[\min \left(\frac{\pi_\theta(o_i | s, o_{<i})}{\pi_{\theta_{\text{old}}}(o_i | s, o_{<i})} \hat{A}_i, \text{clip} \left(\frac{\pi_\theta(o_i | s, o_{<i})}{\pi_{\theta_{\text{old}}}(o_i | s, o_{<i})}, 1 - \varepsilon, 1 + \varepsilon \right) \hat{A}_i \right) \right], \quad (5)$$

where o_i is the i th token of a_t and \hat{A}_i is an estimator of the advantage at the token generation step i .

4.3 AGENTIC ML-SPECIFIC REWARD

Having enabled efficient RL for agentic ML via the step-wise RL paradigm, the next crucial step is to convert the varied feedback into a unified, meaningful reward. While numerical metrics like validation accuracy or loss naturally serve as RL rewards, non-numerical feedback, such as compilation errors or out-of-memory failures, must be carefully incorporated to ensure the reward is coherent.

To address this, we propose an agentic ML-specific reward module that dynamically processes these diverse signals while quantifying performance improvements through scaled task-specific metrics. The key idea is to translate every execution outcome into a unified scalar value. Define $\mathcal{A}_{\text{valid}}$ as valid actions, $\mathcal{A}_{\text{edit}} \subset \mathcal{A}_{\text{valid}}$ as editing actions for ML code, $\mathcal{F}_{\text{error}}$ as error feedback (e.g., compilation failures), $\mathcal{F}_{\text{corner}}$ as corner cases (e.g., resource exhaustion), and $\mathcal{F}_{\text{success}}$ as successful executions. Let m_t be the task-specific metric¹ at state s_t (e.g., loss or accuracy), with m_{init} and m_{best} as the baseline and best human-achievable scores. The reward $R(s_t, a_t)$ is:

¹We follow the official Kaggle evaluation protocol which defines a scalar metric for each ML tasks.

270
271
272
273
274

$$R(s_t, a_t) = \begin{cases} -1 & \text{, if } a_t \notin \mathcal{A}_{\text{valid}} \text{ or } f_t \in \mathcal{F}_{\text{error}} \\ 0 & \text{, if } a_t \in \mathcal{A}_{\text{valid}} / \mathcal{A}_{\text{edit}} \text{ or } f_t \in \mathcal{F}_{\text{corner}} \\ \frac{m_{t+1} - m_t}{m_{\text{best}} - m_{\text{init}}} & \text{, if } a_t \in \mathcal{A}_{\text{edit}} \text{ and } f_t \in \mathcal{F}_{\text{success}}. \end{cases} \quad (6)$$

275
276
277
278
279
280
281

This reward module handles all possible agentic ML scenarios: (1) Invalid actions or errors receive -1 to penalize faulty outputs; (2) Valid non-editing actions or corner cases receive 0 as a neutral acknowledgment of legitimacy while recognizing external constraints; (3) Success edits yield a scaled metric improvement for task-driven refinement. By unifying penalties for errors, neutrality for non-editing actions, and task-driven rewards for edits, the module provides consistent, informative feedback for iterative refinement and continuous improvement across diverse ML tasks.

282

5 EXPERIMENTS

283
284
285

5.1 EXPERIMENTAL SETUPS

286
287
288
289
290
291
292
293
294
295
296

Training. For training data collection, we adopt a GPT-4o-mini-driven (OpenAI, 2024) agent scaffolded by MLAB (Huang et al., 2023). This agent interacts with the MLAGentBench (Huang et al., 2023) agentic ML environment to generate expert trajectories. We collect 10k expert trajectories across 9 ML tasks, comprising 4 tasks from MLAGentBench and 5 from MLE-bench (Chan et al., 2024), with each trajectory limited to 15 steps and 30 minutes of runtime. Additional data collection details are provided in Appendix B. For exploration-enhanced fine-tuning, we train Qwen2.5-7B (Yang et al., 2024) using these 10k expert trajectories via supervised fine-tuning (SFT). For step-wise RL, we select 10k states sampled from expert trajectories to further train the SFT model using Proximal Policy Optimization (PPO). All training is conducted on 8 A100 GPUs. The fine-tuning stage runs for 2 epochs with a learning rate of $2e - 5$, while the RL stage runs for 1 epoch with an actor learning rate of $1e - 6$ and a critic learning rate of $1e - 5$. See additional training details in Appendix C.1.

297
298
299
300
301
302
303
304

Testing. To verify the generalization ability across ML tasks of ML-Agent, we select 10 held-out tasks from MLE-bench, which are not seen during training and generally more challenging than the training tasks. Details of these tasks are provided in Appendix B.2. During testing, the MLAGentBench environment settings remain consistent with those used in training. To comprehensively assess the LLM agent’s ability in autonomous ML, we propose **Performance gain** Δ_r , the relative improvement over the initial script, defined as $\Delta_r = \beta \frac{m_{\text{avg}@8} - m_{\text{init}}}{m_{\text{init}}}$ where $m_{\text{avg}@8}$ is the mean score over 8 trajectories, m_{init} is the initial script’s score, and $\beta \in \{-1, 1\}$ adjusts for metrics (e.g. MAE, RMSE) to ensure positive Δ_r indicates improvement.

305
306
307
308
309
310
311
312

Baselines. To provide a comprehensive comparison, we evaluate ML-Agent against 3 prompted-based agentic ML methods: MLAB (Huang et al., 2023), AIDE (Jiang et al., 2025), and ML-Master (Liu et al., 2025a). All agents are tested using a diverse set of backbone LLMs, spanning small-scale open-source models (e.g., Qwen2.5-7B-Instruct (Yang et al., 2024)), medium-scale models (e.g., Qwen3-235B (Yang et al., 2025)), large-scale open-source models (e.g., DeepSeek-R1 (Guo et al., 2025b)), and state-of-the-art proprietary LLMs (Gemini-2.5-Pro (Comanici et al., 2025) and GPT-5 (OpenAI, 2025)). We keep the same time limit and number of ML code modifications for a fair comparison between agents with different scaffolds.

313
314

5.2 MAIN RESULTS

315
316
317
318

We conduct extensive experiments to evaluate the performance of ML-Agent, a learning-based LLM agent trained through our proposed framework for autonomous ML. Our results demonstrate that ML-Agent achieves strong and consistent performance across both held-in and held-out tasks, and exhibits continuous performance improvements during RL training.

319
320
321
322
323

ML-Agent achieves superior performance across both held-in and held-out tasks. We compare ML-Agent with 5 powerful LLM-based agents in 3 scaffolds across 3 held-in and 10 held-out tasks. As shown in Table 1, ML-Agent significantly outperforms other large open-source models, such as the 671B DeepSeek-R1. For closed-source GPT-5, our agent remains remarkably competitive. Notably, despite being trained on only 9 tasks, ML-Agent delivers top-tier results across all 10 held-out tasks, demonstrating strong generalization and effective learning from limited experience.

324
 325 Table 1: Comparing 7B ML-Agent with baselines across different agent frameworks driven by
 326 proprietary/open-source LLMs on 3 **held-in** tasks (included in training) and 10 **held-out** tasks
 327 (unseen during training) from MLE-bench. For each task, we report average performance gain (%)
 328 over 8 trajectories.

329 Method	330 Model	331 #Params	332 cifar10	333 house	334 feedback	335 denoising	336 leaf	337 statoil	338 whale
<i>Prompt-Based Method</i>									
331 MLAB	332 Qwen2.5-7B-Instruct	333 7B	334 1.37	335 0.23	336 1.39	337 2.10	338 2.52	339 -6.32	340 12.25
	341 Qwen3-235B	342 235B	343 57.61	344 3.01	345 6.70	346 62.60	347 -2.12	348 -16.36	349 26.68
	350 DeepSeek-R1	351 671B	352 28.96	353 3.45	354 5.53	355 8.83	356 4.85	357 0.04	358 33.44
	359 GPT-5	360 N/A	361 61.46	362 12.15	363 12.74	364 66.00	365 -45.63	366 -6.43	367 89.59
	368 Gemini-2.5-Pro	369 N/A	370 16.78	371 1.16	372 0.10	373 37.85	374 -4.38	375 -4.26	376 22.38
377 AIDE	378 Qwen2.5-7B-Instruct	379 7B	380 11.36	381 2.42	382 7.52	383 7.33	384 -4.75	385 -4.33	386 0.52
	387 Qwen3-235B	388 235B	389 -0.10	390 2.04	391 11.10	392 41.65	393 4.75	394 -2.89	395 8.26
	396 DeepSeek-R1	397 671B	398 72.55	399 5.35	400 13.07	401 33.23	402 -10.25	403 -4.54	404 30.77
	405 GPT-5	406 N/A	407 76.53	408 22.15	409 8.77	410 77.38	411 31.50	412 -9.18	413 26.42
	414 Gemini-2.5-Pro	415 N/A	416 53.59	417 11.13	418 9.44	419 62.72	420 -84.25	421 -6.08	422 56.45
423 ML-Master	424 Qwen2.5-7B-Instruct	425 7B	426 1.03	427 0.00	428 0.10	429 2.44	430 -1.38	431 -3.99	432 1.12
	433 DeepSeek-R1	434 671B	435 73.43	436 18.25	437 12.07	438 14.56	439 -14.75	440 -2.78	441 33.39
	442 GPT-5	443 N/A	444 71.64	445 22.3	446 10.54	447 10.96	448 23.88	449 -2.48	450 67.07
<i>Learning-Based Method</i>									
ML-Agent(Ours)		7B	33.80	6.77	13.47	52.38	13.87	1.41	72.89
347 Method	348 Model	349 #Params	350 learning	351 detecting	352 spooky	353 jigsaw	354 us	355 tabular	356 Avg.
<i>Prompt-Based Method</i>									
357 MLAB	358 Qwen2.5-7B-Instruct	359 7B	360 1.23	361 0.51	362 -0.46	363 -0.06	364 3.75	365 0.04	366 1.43
	367 Qwen3-235B	368 235B	369 0.30	370 1.02	371 0.80	372 0.01	373 1.96	374 -0.07	375 10.93
	376 DeepSeek-R1	377 671B	378 0.05	379 0.25	380 0.89	381 0.00	382 2.67	383 -0.13	384 6.83
	385 GPT-5	386 N/A	387 4.36	388 11.20	389 6.79	390 0.00	391 23.38	392 0.23	393 18.14
	394 Gemini-2.5-Pro	395 N/A	396 0.00	397 0.13	398 0.04	399 0.00	400 0.13	401 0.00	402 5.38
403 AIDE	404 Qwen2.5-7B-Instruct	405 7B	406 -9.78	407 -0.38	408 0.07	409 0.01	410 0.00	411 0.08	412 0.77
	413 Qwen3-235B	414 235B	415 2.37	416 0.43	417 0.96	418 -12.15	419 0.51	420 0.00	421 4.38
	422 DeepSeek-R1	423 671B	424 1.38	425 0.31	426 0.36	427 0.01	428 5.78	429 0.14	430 11.40
	431 GPT-5	432 N/A	433 4.51	434 0.13	435 4.25	436 0.14	437 29.69	438 0.11	439 20.95
	440 Gemini-2.5-pro	441 N/A	442 7.35	443 0.74	444 4.34	445 0.04	446 31.92	447 0.13	448 11.35
449 ML-Master	450 Qwen2.5-7B-Instruct	451 7B	452 1.79	453 0.26	454 -0.04	455 -0.02	456 -0.02	457 0.00	458 0.10
	459 DeepSeek-R1	460 671B	461 3.03	462 0.00	463 4.01	464 -0.04	465 29.27	466 0.22	467 13.13
	468 GPT-5	469 N/A	470 6.38	471 0.79	472 10.41	473 0.35	474 26.49	475 0.25	476 19.12
<i>Learning-Based Method</i>									
ML-Agent(Ours)		7B	1.91	1.74	1.76	0.01	12.96	0.20	16.40

364
 365
 366 **ML-Agent efficiently achieves good performance with much lower cost.** As illustrated in Figure 2,
 367 we plot the average performance gain against the average cost per trajectory for various agents. Our
 368 proposed ML-Agent (the star) is a clear outlier, positioned in the optimal top-left corner. It achieves
 369 highly competitive performance gain of over 15% while maintaining an exceptionally low cost of
 370 less than 0.01\$ per trajectory. In contrast, baseline agents like MLAB using powerful models such
 371 as GPT-5 incur costs that are more than 20 times higher for similar or even lower performance.
 372 This result highlights the significant efficiency of learning-based paradigm, proving it can produce a
 373 state-of-the-art agent without relying on expensive, large-scale models.

374 **ML-Agent achieves continuous performance improvements.** Figure 3 shows that ML-Agent
 375 demonstrates consistent performance improvement across both held-in and held-out tasks as training
 376 progresses. This highlights the effectiveness of our step-wise RL paradigm and exploration-enriched
 377 fine-tuning in enabling continuous learning from ML environmental feedback, ultimately allowing
 378 ML-Agent to outperform all baseline methods.

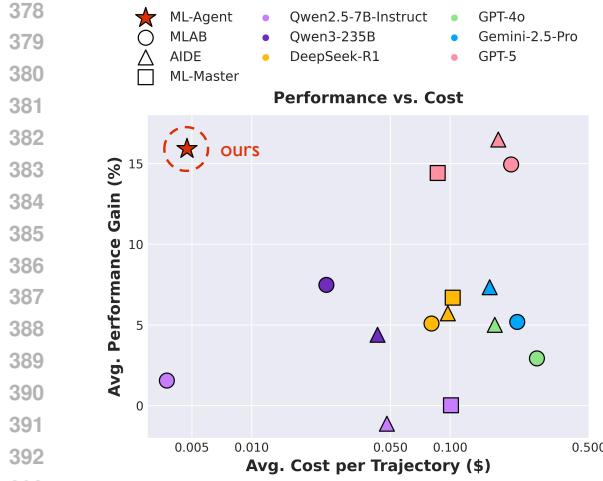


Figure 2: Comparison of average performance gain (%) vs. cost (\$) across different models and scaffolds on 10 held-out tasks. Our ML-Agent significantly outperforms other baselines with a competitive gain at a lower cost.

5.3 ANALYSIS

Exploration-enriched fine-tuning is crucial for step-wise RL training. To validate the efficacy of exploration-enriched fine-tuning in enhancing subsequent RL training, we replace our exploration-enriched fine-tuned model (ML-Agent-SFT) with Qwen2.5-7B (Qwen-7B-Base), Qwen2.5-7B-Instruct (Qwen-7B-Instruct), and DeepSeek-R1-Distill-Qwen-7B (Guo et al., 2025b)(Qwen-7B-Distill) as base models for the RL training. We evaluate the average performance gain of the resulting RL-trained agents on held-in and held-out tasks (Figure 4). The agent trained from Qwen-7B-Distill fails to generate valid actions due to distillation-induced format issues, resulting in ineffective learning. The agent trained from Qwen-7B-Base shows overall performance degradation from limited instruction-following capabilities. The agent trained from Qwen-7B-Instruct achieves +13% gains on held-in tasks but -12% on held-out tasks, indicating poor generalization. In contrast, the agent trained from our ML-Agent-SFT achieves +18% and +16% improvement on held-in and held-out tasks, respectively, with greater action diversity during autonomous ML experimentation (Figure 7). These results confirm that exploration-enriched fine-tuning promotes format-compliant, diverse actions, enhancing exploration and generalization in step-wise RL.

Effectiveness of Step-wise RL Training. To improve training efficiency and scalability, we propose a step-wise RL approach that samples single states from expert trajectories and evaluates atomic actions. To validate this, we implement an alternative episode-wise RL approach, where the policy rolls out the entire trajectory from the task description during data collecting phase in RL. Both methods are initialized from the same ML-Agent-SFT model and trained for 39 steps. We measure GPU time every 5 steps for step-wise RL and 1 step for episode-wise RL. As shown in Figure 3, step-wise RL adapts more quickly and achieves faster performance gains on both held-in and held-out tasks, while the performance of episode-wise RL improves slowly and incurs much higher time cost. These results demonstrate that step-wise RL not only improves training efficiency by avoiding expensive online rollouts, but also leads to improved performance through targeted single-step updates.

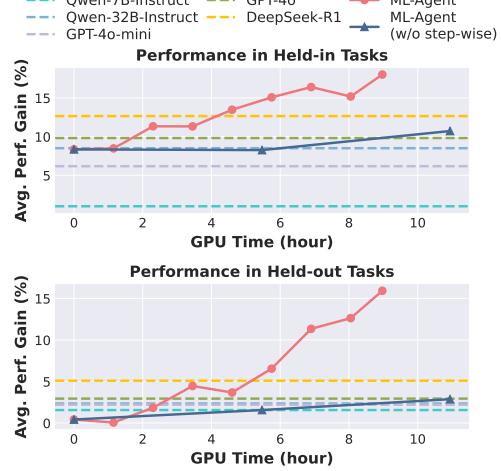


Figure 3: ML-Agent achieves continuous performance improvements; Step-wise RL (evaluated every 5 steps) is more efficient than episode-wise RL (evaluated every 1 step) on both held-in and held-out tasks.

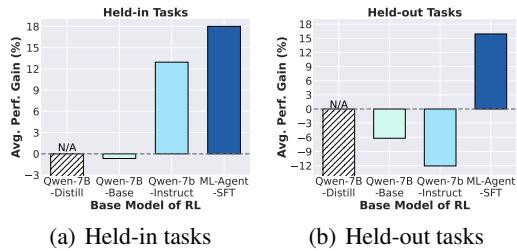


Figure 4: Exploration-enriched fine-tuning is crucial for RL training. "N/A" means the training based on the model fails to generate valid results.

432

433
434
435
Table 2: Ablation study on the ML-specific reward module, indicating the necessity of three components.
The three components are normalized performance reward ($R_{\text{perf.}}$), format reward (R_{format}),
and corner cases reward (R_{corner}). We report the average performance gain (%) for each task.

	Task			cifar10	house	feedback	leaf	detecting	us	tabular	whale
	$R_{\text{perf.}}$	R_{format}	R_{corner}								
①	✗	✓	✓	17.58	3.94	7.79	4.75	0.26	6.40	-24.96	23.24
②	✓	✗	✓	10.98	6.17	8.34	-30.25	0.03	6.27	-12.54	2.84
③	✓	✓	✗	13.56	6.64	7.67	8.50	0.58	8.67	-0.48	28.06
④	✓	✓	✓	33.80	6.77	13.47	13.87	1.74	12.96	0.20	72.89

442

443

444
445
446
447
448
449
450
451
452
453
Effectiveness of agentic-ML specific reward module. Ablation studies in Table 2 show each reward
component is essential: (1) Performance($R_{\text{perf.}}$): Replacing the scaled performance difference with
binary reward leads to noticeable performance drops. This confirms that fine-grained reward signals
are more informative for learning meaningful improvements. (2) Format(R_{format}): Removing format
constraints causes the largest degradation (e.g., -11.75% on cifar-10), emphasizing the necessity of
syntactic and semantic correctness of agent’s output format. (3) Corner cases(R_{corner}): Disabling the
neutral reward for corner cases has minimal impact due to their rarity, but improves training stability
by preventing over-penalization of non-fatal issues. In summary, each component of the reward
module plays a distinct and complementary role: $R_{\text{perf.}}$ drives performance improvement, R_{format}
ensures actions validity, and R_{corner} maintains robustness under real-world limitations. Together, they
form a coherent and comprehensive reward structure during RL training for agentic ML.

454

455

456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
Effects of task numbers in RL. We investigate the impact of
using different numbers of ML tasks (0, 3, 6, 9) during step-
wise RL training, where the "0 task" condition corresponds
to ML-Agent-SFT. We evaluate performance in terms of
average performance gain on held-in and held-out tasks. As
shown in Figure 5, performance on both task types improves
monotonically as the number of ML tasks increases during
RL training. Specifically, training with 3, 6, and 9 ML tasks
using step-wise RL lifts the average performance gain on
held-out tasks from nearly 0% to approximately 3%, 6%, and
16%, respectively. These results indicate that expanding the
diversity of ML tasks during RL not only refines the agent’s
ability on familiar tasks but also significantly improves the
agent’s ability to generalize across unseen tasks.

478

479

480

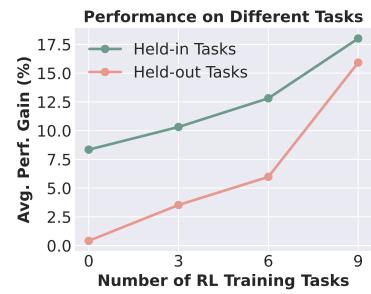
481

482

483

484

485

Case study. To provide an intuitive understanding, we
present several examples in the Appendix C.3, demonstrating
task specifications, initial code implementations, baseline
and our model’s execution trajectories. These demonstrate
that our methodology: (1) generates diverse action sequences through comprehensive reasoning,
(2) automatically initiates backtracking mechanisms when performance metrics remain unimproved
by proposed modifications, and (3) maintains operational effectiveness on novel tasks through
generalized reasoning capabilities.5
Figure 5: Effects of training task number on RL performance. While the
pure sft model shows minimal generalization, RL drives generalization.

6 CONCLUSION

486

487

488

489

490

491

492

493

494

495

496

497

498

499

500

501

502

In this paper, we propose a training framework for an LLM-based agent on autonomous machine
learning tasks. Unlike heuristic prompt-based methods, our method enables agents to learn from
task-solving experiences, iteratively refine strategies, and generalize across tasks. The framework
involved exploration-enriched fine-tuning, efficient step-wise RL training, and agentic ML-specific
reward module. Extensive experiments demonstrate that ML-Agent, powered by a 7B-parameter
LLM, surpasses agents using 671B models and achieves state-of-the-art performance on 13 tasks,
including cross-task generalization. This work advances autonomous ML engineering from rule-based
automation to dynamic, experience-driven learning, reducing reliance on human intervention.

486 REFERENCES
487

488 Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Ce-
489 line Lin, and Pavlo Molchanov. Small language models are the future of agentic ai, 2025. URL
490 <https://arxiv.org/abs/2506.02153>.

491 James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The journal*
492 *of machine learning research*, 13(1):281–305, 2012.

493

494 Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio
495 Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al. Mle-bench: Evaluating machine learning
496 agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*, 2024.

497

498 Gheorghe Comanici, Eric Bieber, Mike Schaeckermann, Ice Pasupat, Noveen Sachdeva, Inderjit
499 Dhillon, Marcel Blstein, Ori Ram, Dan Zhang, Evan Rosen, Luke Marris, Sam Petulla, Colin
500 Gaffney, Asaf Aharoni, Nathan Lintz, Tiago Cardal Pais, Henrik Jacobsson, Idan Szpektor, Nan-
501 Jiang Jiang, Krishna Haridasan, Ahmed Omran, Nikunj Saunshi, Dara Bahri, Gaurav Mishra, Eric
502 Chu, Toby Boyd, Brad Hekman, Aaron Parisi, Chaoyi Zhang, Kornraphop Kawintiranon, Tania
503 Bedrax-Weiss, Oliver Wang, Ya Xu, Ollie Purkiss, Uri Mendlovic, Ilai Deutel, Nam Nguyen, Adam
504 Langley, Flip Korn, Lucia Rossazza, Alexandre Ramé, Sagar Waghmare, Helen Miller, Nathan
505 Byrd, Ashrith Sheshan, Raia Hadsell Sangnie Bhardwaj, Pawel Janus, Tero Rissa, Dan Horgan,
506 Sharon Silver, Ayzaan Wahid, Sergey Brin, Yves Raimond, Klemen Kloboves, Cindy Wang,
507 Nitesh Bharadwaj Gundavarapu, Ilia Shumailov, Bo Wang, Mantas Pajarskas, Joe Heyward, Martin
508 Nikoltchev, Maciej Kula, Hao Zhou, Zachary Garrett, Sushant Kafle, Sercan Arik, Ankita Goel,
509 Mingyao Yang, Jiho Park, Koji Kojima, Parsa Mahmoudieh, Koray Kavukcuoglu, Grace Chen,
510 Doug Fritz, Anton Bulyenov, Sudeshna Roy, Dimitris Paparas, Hadar Shemtov, Bo-Juen Chen,
511 Robin Strudel, David Reitter, Aurko Roy, Andrey Vlasov, Changwan Ryu, Chas Leichner, Haichuan
512 Yang, Zelda Mariet, Denis Vnukov, Tim Sohn, Amy Stuart, Wei Liang, Minmin Chen, Praynaa
513 Rawlani, Christy Koh, JD Co-Reyes, Guangda Lai, Praseem Banzal, Dimitrios Vytiniotis, Jieru
514 Mei, Mu Cai, Mohammed Badawi, Corey Fry, Ale Hartman, Daniel Zheng, Eric Jia, James Keeling,
515 Annie Louis, Ying Chen, Efren Robles, Wei-Chih Hung, Howard Zhou, Nikita Saxena, Sonam
516 Goenka, Olivia Ma, Zach Fisher, Mor Hazan Taege, Emily Graves, David Steiner, Yujia Li, Sarah
517 Nguyen, Rahul Sukthankar, Joe Stanton, Ali Eslami, Gloria Shen, Berkin Akin, Alexey Guseynov,
518 Yiqian Zhou, Jean-Baptiste Alayrac, Armand Joulin, Efrat Farkash, Ashish Thapliyal, Stephen
519 Roller, Noam Shazeer, Todor Davchev, Terry Koo, Hannah Forbes-Pollard, Kartik Audhkhasi,
520 Greg Farquhar, Adi Mayray Gilady, Maggie Song, John Aslanides, Piermaria Mendolicchio,
521 Alicia Parrish, John Blitzer, Pramod Gupta, Xiaoen Ju, Xiaochen Yang, Puranjay Datta, Andrea
522 Tacchetti, Sanket Vaibhav Mehta, Gregory Dibb, Shubham Gupta, Federico Piccinini, Raia Hadsell,
523 Sujee Rajayogam, Jiepu Jiang, Patrick Griffin, Patrik Sundberg, Jamie Hayes, Alexey Frolov,
524 Tian Xie, Adam Zhang, Kingshuk Dasgupta, Uday Kalra, Lior Shani, Klaus Macherey, Tzu-
525 Kuo Huang, Liam MacDermed, Karthik Duddu, Paulo Zucchello, Zi Yang, Jessica Lo, Kai Hui,
526 Matej Kastelic, Derek Gasaway, Qijun Tan, Summer Yue, Pablo Barrio, John Wieting, Weel
527 Yang, Andrew Nystrom, Solomon Demmessie, Anselm Levskaya, Fabio Viola, Chetan Tekur,
528 Greg Billcock, George Necula, Mandar Joshi, Rylan Schaeffer, Swachhand Lokhande, Christina
529 Sorokin, Pradeep Shenoy, Mia Chen, Mark Collier, Hongji Li, Taylor Bos, Nevan Wichers,
530 Sun Jae Lee, Angéline Pouget, Santhosh Thangaraj, Kyriakos Axiotis, Phil Crone, Rachel Sterneck,
531 Nikolai Chinaev, Victoria Krakovna, Oleksandr Ferludin, Ian Gemp, Stephanie Winkler, Dan
532 Goldberg, Ivan Korotkov, Kefan Xiao, Malika Mehrotra, Sandeep Mariserla, Vihari Piratla, Terry
533 Thruk, Khiem Pham, Hongxu Ma, Alexandre Senges, Ravi Kumar, Clemens Meyer, Ellie Talius,
534 Nuo Wang Pierse, Ballie Sandhu, Horia Toma, Kuo Lin, Swaroop Nath, Tom Stone, Dorsa Sadigh,
535 Nikita Gupta, Arthur Guez, Avi Singh, Matt Thomas, Tom Duerig, Yuan Gong, Richard Tanburn,
536 Lydia Lihui Zhang, Phuong Dao, Mohamed Hammad, Sirui Xie, Shruti Rijhwani, Ben Murdoch,
537 Duhyeon Kim, Will Thompson, Heng-Tze Cheng, Daniel Sohn, Pablo Sprechmann, Qiantong
538 Xu, Srinivas Tadepalli, Peter Young, Ye Zhang, Hansa Srinivasan, Miranda Aperghis, Aditya
539 Ayyar, Hen Fitoussi, Ryan Burnell, David Madras, Mike Dusenberry, Xi Xiong, Tayo Oguntebi,
Ben Albrecht, Jörg Bornschein, Jovana Mitrović, Mason Dimarco, Bhargav Kanagal Shamanna,
Premal Shah, Eren Sezener, Shyam Upadhyay, Dave Lacey, Craig Schiff, Sebastien Baur, Sanjay
Ganapathy, Eva Schnider, Mateo Wirth, Connor Schenck, Andrey Simanovsky, Yi-Xuan Tan,
Philipp Fränken, Dennis Duan, Bharath Mankalale, Nikhil Dhawan, Kevin Sequeira, Zichuan
Wei, Shivanker Goel, Caglar Unlu, Yukun Zhu, Haitian Sun, Ananth Balashankar, Kurt Shuster,

540 Megh Umekar, Mahmoud Alnahlawi, Aäron van den Oord, Kelly Chen, Yuexiang Zhai, Zihang
 541 Dai, Kuang-Huei Lee, Eric Doi, Lukas Zilka, Rohith Vallu, Disha Shrivastava, Jason Lee, Hisham
 542 Husain, Honglei Zhuang, Vincent Cohen-Addad, Jarred Barber, James Atwood, Adam Sadovsky,
 543 Quentin Wellens, Steven Hand, Arunkumar Rajendran, Aybuke Turker, CJ Carey, Yuanzhong Xu,
 544 Hagen Soltau, Zefei Li, Xinying Song, Conglong Li, Iurii Kemaev, Sasha Brown, Andrea Burns,
 545 Viorica Patraucean, Piotr Stanczyk, Renga Aravamudhan, Mathieu Blondel, Hila Noga, Lorenzo
 546 Blanco, Will Song, Michael Isard, Mandar Sharma, Reid Hayes, Dalia El Badawy, Avery Lamp,
 547 Itay Laish, Olga Kozlova, Kelvin Chan, Sahil Singla, Srinivas Sunkara, Mayank Upadhyay, Chang
 548 Liu, Aijun Bai, Jarek Wilkiewicz, Martin Zlocha, Jeremiah Liu, Zhuowan Li, Haiguang Li, Omer
 549 Barak, Ganna Raboshchuk, Jiho Choi, Fangyu Liu, Erik Jue, Mohit Sharma, Andreea Marzoca,
 550 Robert Busa-Fekete, Anna Korsun, Andre Elisseeff, Zhe Shen, Sara Mc Carthy, Kay Lamerigts,
 551 Anahita Hosseini, Hanzhao Lin, Charlie Chen, Fan Yang, Kushal Chauhan, Mark Omernick,
 552 Dawei Jia, Karina Zainullina, Demis Hassabis, Danny Vainstein, Ehsan Amid, Xiang Zhou, Ronny
 553 Votel, Eszter Vértés, Xinjian Li, Zongwei Zhou, Angeliki Lazaridou, Brendan McMahan, Arjun
 554 Narayanan, Hubert Soyer, Sujoy Basu, Kayi Lee, Bryan Perozzi, Qin Cao, Leonard Berrada, Rahul
 555 Arya, Ke Chen, Katrina, Xu, Matthias Lochbrunner, Alex Hofer, Sahand Sharifzadeh, Renjie
 556 Wu, Sally Goldman, Pranjal Awasthi, Xuezhi Wang, Yan Wu, Claire Sha, Biao Zhang, Maciej
 557 Mikuła, Filippo Graziano, Siobhan Mcoughlin, Irene Giannoumis, Youhei Namiki, Chase Malik,
 558 Carey Radebaugh, Jamie Hall, Ramiro Leal-Cavazos, Jianmin Chen, Vikas Sindhwani, David Kao,
 559 David Greene, Jordan Griffith, Chris Welty, Ceslee Montgomery, Toshihiro Yoshino, Liangzhe
 560 Yuan, Noah Goodman, Assaf Hurwitz Michaely, Kevin Lee, KP Sawhney, Wei Chen, Zheng
 561 Zheng, Megan Shum, Nikolay Savinov, Etienne Pot, Alex Pak, Morteza Zadimoghaddam, Sijal
 562 Bhatnagar, Yoad Lewenberg, Blair Kutzman, Ji Liu, Lesley Katzen, Jeremy Selier, Josip Djolonga,
 563 Dmitry Lepikhin, Kelvin Xu, Jacky Liang, Jiewen Tan, Benoit Schillings, Muge Ersoy, Pete
 564 Blois, Bernd Bandemer, Abhimanyu Singh, Sergei Lebedev, Pankaj Joshi, Adam R. Brown, Evan
 565 Palmer, Shreya Pathak, Komal Jalan, Fedir Zubach, Shuba Lall, Randall Parker, Alok Gunjan,
 566 Sergey Rogulenko, Sumit Sanghai, Zhaoqi Leng, Zoltan Egyed, Shixin Li, Maria Ivanova, Kostas
 567 Andriopoulos, Jin Xie, Elan Rosenfeld, Auriel Wright, Ankur Sharma, Xinyang Geng, Yicheng
 568 Wang, Sam Kwei, Renke Pan, Yujing Zhang, Gabby Wang, Xi Liu, Chak Yeung, Elizabeth
 569 Cole, Aviv Rosenberg, Zhen Yang, Phil Chen, George Polovets, Pranav Nair, Rohun Saxena,
 570 Josh Smith, Shuo yiin Chang, Aroma Mahendru, Svetlana Grant, Anand Iyer, Irene Cai, Jed
 571 McGiffin, Jiaming Shen, Alanna Walton, Antonious Girgis, Oliver Woodman, Rosemary Ke, Mike
 572 Kwong, Louis Rouillard, Jinmeng Rao, Zhihao Li, Yuntao Xu, Flavien Prost, Chi Zou, Ziwei Ji,
 573 Alberto Magni, Tyler Liechty, Dan A. Calian, Deepak Ramachandran, Igor Krivokon, Hui Huang,
 574 Terry Chen, Anja Hauth, Anastasija Ilić, Weijuan Xi, Hyeontaek Lim, Vlad-Doru Ion, Pooya
 575 Moradi, Metin Toksoz-Exley, Kalesha Bullard, Miltos Allamanis, Xiaomeng Yang, Sophie Wang,
 576 Zhi Hong, Anita Gergely, Cheng Li, Bhavishya Mittal, Vitaly Kovalev, Victor Ungureanu, Jane
 577 Labanowski, Jan Wassenberg, Nicolas Lacasse, Geoffrey Cideron, Petar Dević, Annie Marsden,
 578 Lynn Nguyen, Michael Fink, Yin Zhong, Tatsuya Kiyono, Desi Ivanov, Sally Ma, Max Bain,
 579 Kiran Yalasangi, Jennifer She, Anastasia Petrushkina, Mayank Lunayach, Carla Bromberg, Sarah
 580 Hodkinson, Vilobh Meshram, Daniel Vlasic, Austin Kyker, Steve Xu, Jeff Stanway, Zuguang Yang,
 581 Kai Zhao, Matthew Tung, Seth Odoom, Yasuhisa Fujii, Justin Gilmer, Eunyoung Kim, Felix Halim,
 582 Quoc Le, Bernd Bohnet, Seliem El-Sayed, Behnam Neyshabur, Malcolm Reynolds, Dean Reich,
 583 Yang Xu, Erica Moreira, Anuj Sharma, Zeyu Liu, Mohammad Javad Hosseini, Naina Raisinghani,
 584 Yi Su, Ni Lao, Daniel Formoso, Marco Gelmi, Almog Gueta, Tapomay Dey, Elena Gribovskaya,
 585 Domagoj Ćevid, Sidharth Mudgal, Garrett Bingham, Jianling Wang, Anurag Kumar, Alex Cullum,
 586 Feng Han, Konstantinos Bousmalis, Diego Cedillo, Grace Chu, Vladimir Magay, Paul Michel,
 587 Ester Hlavnova, Daniele Calandriello, Setareh Ariafar, Kaisheng Yao, Vikash Sehwag, Arpi Vezer,
 588 Agustin Dal Lago, Zhenkai Zhu, Paul Kishan Rubenstein, Allen Porter, Anirudh Baddepudi, Oriana
 589 Riva, Mihai Dorin Istin, Chih-Kuan Yeh, Zhi Li, Andrew Howard, Nilpa Jha, Jeremy Chen, Raoul
 590 de Liedekerke, Zafarali Ahmed, Mikel Rodriguez, Tanuj Bhatia, Bangju Wang, Ali Elqursh, David
 591 Klinghoffer, Peter Chen, Pushmeet Kohli, Te I, Weiyang Zhang, Zack Nado, Jilin Chen, Maxwell
 592 Chen, George Zhang, Ayush Singh, Adam Hillier, Federico Lebron, Yiqing Tao, Ting Liu, Gabriel
 593 Dulac-Arnold, Jingwei Zhang, Shashi Narayan, Buhuang Liu, Orhan Firat, Abhishek Bhowmick,
 Bingyuan Liu, Hao Zhang, Zizhao Zhang, Georges Rotival, Nathan Howard, Anu Sinha, Alexander
 Grushetsky, Benjamin Beyret, Keerthana Gopalakrishnan, James Zhao, Kyle He, Szabolcs Payrits,
 Zaid Nabulsi, Zhaoyi Zhang, Weijie Chen, Edward Lee, Nova Fallen, Sreenivas Gollapudi, Aurick
 Zhou, Filip Pavetić, Thomas Köppe, Shiyu Huang, Rama Pasumarthi, Nick Fernando, Felix
 Fischer, Daria Ćurko, Yang Gao, James Svensson, Austin Stone, Haroon Qureshi, Abhishek

594 Sinha, Apoorv Kulshreshtha, Martin Matysiak, Jieming Mao, Carl Saroufim, Aleksandra Faust,
 595 Qingnan Duan, Gil Fidel, Kaan Katircioglu, Raphaël Lopez Kaufman, Dhruv Shah, Weize Kong,
 596 Abhishek Bapna, Gellért Weisz, Emma Dunleavy, Praneet Dutta, Tianqi Liu, Rahma Chaabouni,
 597 Carolina Parada, Marcus Wu, Alexandra Belias, Alessandro Bissacco, Stanislav Fort, Li Xiao,
 598 Fantine Huot, Chris Knutson, Yochai Blau, Gang Li, Jennifer Prendki, Juliette Love, Yinlam
 599 Chow, Pichi Charoenpanit, Hidetoshi Shimokawa, Vincent Coriou, Karol Gregor, Tomas Izo, Arjun
 600 Akula, Mario Pinto, Chris Hahn, Dominik Paulus, Jiaxian Guo, Neha Sharma, Cho-Jui Hsieh,
 601 Adaeze Chukwuka, Kazuma Hashimoto, Nathalie Rauschmayr, Ling Wu, Christof Angermueller,
 602 Yulong Wang, Sebastian Gerlach, Michael Pliskin, Daniil Mirylenka, Min Ma, Lexi Baugher,
 603 Bryan Gale, Shaan Bijwadia, Nemanja Rakićević, David Wood, Jane Park, Chung-Ching Chang,
 604 Babi Seal, Chris Tar, Kacper Krasowiak, Yiwen Song, Georgi Stephanov, Gary Wang, Marcello
 605 Maggioni, Stein Xudong Lin, Felix Wu, Shachi Paul, Zixuan Jiang, Shubham Agrawal, Bilal Piot,
 606 Alex Feng, Cheolmin Kim, Tulsee Doshi, Jonathan Lai, Chuqiao, Xu, Sharad Vikram, Ciprian
 607 Chelba, Sebastian Krause, Vincent Zhuang, Jack Rae, Timo Denk, Adrian Collister, Lotte Weerts,
 608 Xianghong Luo, Yifeng Lu, Håvard Garnes, Nitish Gupta, Terry Spitz, Avinatan Hassidim, Lihao
 609 Liang, Izhak Shafran, Peter Humphreys, Kenny Vassigh, Phil Wallis, Virat Shejwalkar, Nicolas
 610 Perez-Nieves, Rachel Hornung, Melissa Tan, Beka Westberg, Andy Ly, Richard Zhang, Brian
 611 Farris, Jongbin Park, Alec Kosik, Zeynep Cankara, Andrii Maksai, Yunhan Xu, Albin Cassirer,
 612 Sergi Caelles, Abbas Abdolmaleki, Mencher Chiang, Alex Fabrikant, Shravya Shetty, Luheng
 613 He, Mai Giménez, Hadi Hashemi, Sheena Panthaplackel, Yana Kulizhskaya, Salil Deshmukh,
 614 Daniele Pighin, Robin Alazard, Disha Jindal, Seb Noury, Pradeep Kumar S, Siyang Qin, Xerxes
 615 Dotiwalla, Stephen Spencer, Mohammad Babaeizadeh, Blake JianHang Chen, Vaibhav Mehta,
 616 Jennie Lees, Andrew Leach, Penporn Koanantakool, Ilia Akolzin, Ramona Comanescu, Junwhan
 617 Ahn, Alexey Svyatkovskiy, Basil Mustafa, David D'Ambrosio, Shiva Mohan Reddy Garlapati,
 618 Pascal Lamblin, Alekh Agarwal, Shuang Song, Pier Giuseppe Sessa, Pauline Coquinot, John
 619 Maggs, Hussain Masoom, Divya Pitta, Yaqing Wang, Patrick Morris-Suzuki, Billy Porter, Johnson
 620 Jia, Jeffrey Dudek, Raghavender R, Cosmin Paduraru, Alan Ansell, Tolga Bolukbasi, Tony Lu,
 621 Ramya Ganeshan, Zi Wang, Henry Griffiths, Rodrigo Benenson, Yifan He, James Swirhun, George
 622 Papamakarios, Aditya Chawla, Kuntal Sengupta, Yan Wang, Vedrana Milutinovic, Igor Mordatch,
 623 Zhipeng Jia, Jamie Smith, Will Ng, Shitij Nigam, Matt Young, Eugen Vušak, Blake Hechtman,
 624 Sheela Goenka, Avital Zipori, Kareem Ayoub, Ashok Popat, Trilok Acharya, Luo Yu, Dawn
 625 Bloxwich, Hugo Song, Paul Roit, Haiqiong Li, Aviel Boag, Nigamaa Nayakanti, Bilva Chandra,
 626 Tianli Ding, Aahil Mehta, Cath Hope, Jiageng Zhang, Idan Heimlich Shtacher, Kartikeya Badola,
 627 Ryo Nakashima, Andrei Sozanschi, Iulia Comşa, Ante Žužul, Emily Caveness, Julian Odell,
 628 Matthew Watson, Dario de Cesare, Phillip Lippe, Derek Lockhart, Siddharth Verma, Huizhong
 629 Chen, Sean Sun, Lin Zhuo, Aditya Shah, Prakhar Gupta, Alex Muzio, Ning Niu, Amir Zait,
 630 Abhinav Singh, Meenu Gaba, Fan Ye, Prajit Ramachandran, Mohammad Saleh, Raluca Ada Popa,
 631 Ayush Dubey, Frederick Liu, Sara Javanmardi, Mark Epstein, Ross Hemsley, Richard Green,
 632 Nishant Ranka, Eden Cohen, Chuyuan Kelly Fu, Sanjay Ghemawat, Jed Borovik, James Martens,
 633 Anthony Chen, Pranav Shyam, André Susano Pinto, Ming-Hsuan Yang, Alexandru Tifrea, David
 634 Du, Boqing Gong, Ayushi Agarwal, Seungyeon Kim, Christian Frank, Saloni Shah, Xiaodan Song,
 635 Zhiwei Deng, Ales Mikhalap, Kleopatra Chatziprimou, Timothy Chung, Toni Creswell, Susan
 636 Zhang, Yennie Jun, Carl Lebsack, Will Truong, Slavica Andačić, Itay Yona, Marco Fornoni, Rong
 637 Rong, Serge Toropov, Afzal Shama Soudagar, Andrew Audibert, Salah Zaiem, Zaheer Abbas,
 638 Andrei Rusu, Sahitya Potluri, Shitao Weng, Anastasios Kementsietsidis, Anton Tsitsulin, Daiyi
 639 Peng, Natalie Ha, Sanil Jain, Tejas Latkar, Simeon Ivanov, Cory McLean, Anirudh GP, Rajesh
 640 Venkataraman, Canoe Liu, Dilip Krishnan, Joel D'sa, Roey Yogev, Paul Collins, Benjamin Lee,
 641 Lewis Ho, Carl Doersch, Gal Yona, Shawn Gao, Felipe Tiengo Ferreira, Adnan Ozturk, Hannah
 642 Muckenhirk, Ce Zheng, Gargi Balasubramaniam, Mudit Bansal, George van den Driessche, Sivan
 643 Eiger, Salem Haykal, Vedant Misra, Abhimanyu Goyal, Danilo Martins, Gary Leung, Jonas
 644 Valfridsson, Four Flynn, Will Bishop, Chenxi Pang, Yoni Halpern, Honglin Yu, Lawrence Moore,
 645 Yuvein, Zhu, Sridhar Thiagarajan, Yoel Drori, Zhisheng Xiao, Lucio Dery, Rolf Jagerman, Jing
 646 Lu, Eric Ge, Vaibhav Aggarwal, Arjun Khare, Vinh Tran, Oded Elyada, Ferran Alet, James Rubin,
 647 Ian Chou, David Tian, Libin Bai, Lawrence Chan, Lukasz Lew, Karolis Misiunas, Taylan Bilal,
 Aniket Ray, Sindhu Raghuram, Alex Castro-Ros, Viral Carpenter, CJ Zheng, Michael Kilgore,
 Josef Broder, Emily Xue, Praveen Kallakuri, Dheeru Dua, Nancy Yuen, Steve Chien, John Schultz,
 Saurabh Agrawal, Reut Tsarfaty, Jingcao Hu, Ajay Kannan, Dror Marcus, Nisarg Kothari, Baochen
 Sun, Ben Horn, Matko Bošnjak, Ferjad Naeem, Dean Hirsch, Lewis Chiang, Boya Fang, Jie Han,
 Qifei Wang, Ben Hora, Antoine He, Mario Lučić, Beer Changpinyo, Anshuman Tripathi, John

648 Youssef, Chester Kwak, Philippe Schlattner, Cat Graves, Rémi Leblond, Wenjun Zeng, Anders
 649 Andreassen, Gabriel Rasskin, Yue Song, Eddie Cao, Junhyuk Oh, Matt Hoffman, Wojtek Skut,
 650 Yichi Zhang, Jon Stritar, Xingyu Cai, Saarthak Khanna, Kathie Wang, Shriya Sharma, Christian
 651 Reisswig, Younghoon Jun, Aman Prasad, Tatiana Sholokhova, Preeti Singh, Adi Gerzi Rosenthal,
 652 Anian Ruoss, Françoise Beaufays, Sean Kirmani, Dongkai Chen, Johan Schalkwyk, Jonathan
 653 Herzig, Been Kim, Josh Jacob, Damien Vincent, Adrian N Reyes, Ivana Balazevic, Léonard
 654 Hussenot, Jon Schneider, Parker Barnes, Luis Castro, Spandana Raj Babbula, Simon Green,
 655 Serkan Cabi, Nico Duduta, Danny Driess, Rich Galt, Noam Velan, Junjie Wang, Hongyang Jiao,
 656 Matthew Mauger, Du Phan, Miteyan Patel, Vlado Galić, Jerry Chang, Eyal Marcus, Matt Harvey,
 657 Julian Salazar, Elahe Dabir, Suraj Satishkumar Sheth, Amol Mandhane, Hanie Sedghi, Jeremiah
 658 Willcock, Amir Zandieh, Shruthi Prabhakara, Aida Amini, Antoine Miech, Victor Stone, Massimo
 659 Nicosia, Paul Niemczyk, Ying Xiao, Lucy Kim, Sławek Kwasiborski, Vikas Verma, Ada Maksutaj
 660 Oflazer, Christoph Hirnschall, Peter Sung, Lu Liu, Richard Everett, Michiel Bakker, Ágoston
 661 Weisz, Yufei Wang, Vivek Sampathkumar, Uri Shaham, Bibo Xu, Yasemin Altun, Mingqiu Wang,
 662 Takaaki Saeki, Guanjie Chen, Emanuel Taropa, Shanthal Vasanth, Sophia Austin, Lu Huang,
 663 Goran Petrovic, Qingyun Dou, Daniel Golovin, Grigory Rozhdestvenskiy, Allie Culp, Will Wu,
 664 Motoki Sano, Divya Jain, Julia Proskurnia, Sébastien Cevey, Alejandro Cruzado Ruiz, Piyush
 665 Patil, Mahdi Mirzazadeh, Eric Ni, Javier Snaider, Lijie Fan, Alexandre Fréchette, AJ Piergiovanni,
 666 Shariq Iqbal, Kenton Lee, Claudio Fantacci, Jinwei Xing, Lisa Wang, Alex Irpan, David Raposo,
 667 Yi Luan, Zhuoyuan Chen, Harish Ganapathy, Kevin Hui, Jiazhong Nie, Isabelle Guyon, Heming
 668 Ge, Roopali Vij, Hui Zheng, Dayeong Lee, Alfonso Castaño, Khuslen Baatarsukh, Gabriel
 669 Ibagon, Alexandra Chronopoulou, Nicholas FitzGerald, Shashank Viswanadha, Safeen Huda,
 670 Rivka Moroshko, Georgi Stoyanov, Prateek Kolhar, Alain Vaucher, Ishaan Watts, Adhi Kuncoro,
 671 Henryk Michalewski, Satish Kambala, Bat-Orgil Batsaikhan, Alek Andreev, Irina Jurenka, Maigo
 672 Le, Qihang Chen, Wael Al Jishi, Sarah Chakera, Zhe Chen, Aditya Kini, Vikas Yadav, Aditya
 673 Siddhant, Ilia Labzovsky, Balaji Lakshminarayanan, Carrie Grimes Bostock, Pankil Botadra,
 674 Ankesh Anand, Colton Bishop, Sam Conway-Rahman, Mohit Agarwal, Yani Donchev, Achintya
 675 Singhal, Félix de Chaumont Quiry, Natalia Ponomareva, Nishant Agrawal, Bin Ni, Kalpesh
 676 Krishna, Masha Samsikova, John Karro, Yilun Du, Tamara von Glehn, Caden Lu, Christopher A.
 677 Choquette-Choo, Zhen Qin, Tingnan Zhang, Sicheng Li, Divya Tyam, Swaroop Mishra, Wing
 678 Lowe, Colin Ji, Weiyi Wang, Manaal Faruqui, Ambrose Slone, Valentin Dalibard, Arunachalam
 679 Narayanaswamy, John Lambert, Pierre-Antoine Manzagol, Dan Karliner, Andrew Bolt, Ivan
 680 Lobov, Aditya Kusupati, Chang Ye, Xuan Yang, Heiga Zen, Nelson George, Mukul Bhutani,
 681 Olivier Lacombe, Robert Riachi, Gagan Bansal, Rachel Soh, Yue Gao, Yang Yu, Adams Yu,
 682 Emily Nottage, Tania Rojas-Esponda, James Noraky, Manish Gupta, Ragha Kotikalapudi, Jichuan
 683 Chang, Sanja Deur, Dan Graur, Alex Mossin, Erin Farnese, Ricardo Figueira, Alexandre Moufarek,
 684 Austin Huang, Patrik Zochbauer, Ben Ingram, Tongzhou Chen, Zelin Wu, Adrià Puigdomènec,
 685 Leland Rechis, Da Yu, Sri Gayatri Sundara Padmanabhan, Rui Zhu, Chu ling Ko, Andrea Banino,
 686 Samira Daruki, Aarush Selvan, Dhruba Bhawar, Daniel Hernandez Diaz, Chen Su, Salvatore
 687 Scellato, Jennifer Brennan, Woohyun Han, Grace Chung, Priyanka Agrawal, Urvashi Khandelwal,
 688 Khe Chai Sim, Morgane Lustman, Sam Ritter, Kelvin Guu, Jiawei Xia, Prateek Jain, Emma Wang,
 689 Tyrone Hill, Mirko Rossini, Marija Kostelac, Tautvydas Misiunas, Amit Sabne, Kyuyeun Kim,
 690 Ahmet Iscen, Congchao Wang, José Leal, Ashwin Sreevatsa, Utku Evcı, Manfred Warmuth, Saket
 691 Joshi, Daniel Suo, James Lottes, Garrett Honke, Brendan Jou, Stefani Karp, Jieru Hu, Himanshu
 692 Sahni, Adrien Ali Taïga, William Kong, Samrat Ghosh, Renshen Wang, Jay Pavagadhi, Natalie
 693 Axelsson, Nikolai Grigorev, Patrick Siegler, Rebecca Lin, Guohui Wang, Emilio Parisotto, Sharath
 694 Maddineni, Krishan Subudhi, Eyal Ben-David, Elena Poichernina, Orgad Keller, Thi Avrahami,
 695 Zhe Yuan, Pulkit Mehta, Jialu Liu, Sherry Yang, Wendy Kan, Katherine Lee, Tom Funkhouser,
 696 Derek Cheng, Hongzhi Shi, Archit Sharma, Joe Kelley, Matan Eyal, Yury Malkov, Corentin Tallec,
 697 Yuval Bahat, Shen Yan, Xintian, Wu, David Lindner, Chengda Wu, Avi Caciularu, Xiyang Luo,
 698 Rodolphe Jenatton, Tim Zaman, Yingying Bi, Ilya Kornakov, Ganesh Mallya, Daisuke Ikeda, Itay
 699 Karo, Anima Singh, Colin Evans, Praneeth Netrapalli, Vincent Nallatamby, Isaac Tian, Yannis
 700 Assael, Vikas Raunak, Victor Carbune, Ioana Bica, Lior Madmoni, Dee Cattle, Snchit Grover,
 701 Krishna Somandepalli, Sid Lall, Amelio Vázquez-Reina, Riccardo Patana, Jiaqi Mu, Pranav Talluri,
 Mohammad Hossein Bateni, AJ Maschinot, Bing Wang, Muye Zhu, Arjun Pillai, Tobias Weyand,
 Maggie Tran, Rajeev Aggarwal, RJ Skerry-Ryan, Jun Xu, Mike Burrows, Xiaoyue Pan, Edouard
 Yvinec, Di Lu, Zhiying Zhang, Duc Dung Nguyen, Hairong Mu, Gabriel Barcik, Helen Ran,
 Lauren Beltrone, Krzysztof Choromanski, Dia Kharrat, Samuel Albanie, Sean Purser-haskell,
 David Bieber, Carrie Zhang, Jing Wang, Tom Hudson, Zhiyuan Zhang, Han Fu, Johannes Mauerer,
 Mohammad Hossein Bateni, AJ Maschinot, Bing Wang, Muye Zhu, Arjun Pillai, Tobias Weyand,

702 Shuang Liu, Oscar Akerlund, Fred Bertsch, Vittal Premachandran, Alicia Jin, Vincent Roulet,
 703 Peter de Boursac, Shubham Mittal, Ndaba Ndebele, Georgi Karadzhov, Sahra Ghalebikesabi,
 704 Ricky Liang, Allen Wu, Yale Cong, Nimesh Ghelani, Sumeet Singh, Bahar Fatemi, Warren, Chen,
 705 Charles Kwong, Alexey Kolganov, Steve Li, Richard Song, Chenkai Kuang, Sobhan Miryoosefi,
 706 Dale Webster, James Wendt, Arkadiusz Socala, Guolong Su, Artur Mendonça, Abhinav Gupta,
 707 Xiaowei Li, Tomy Tsai, Qiong, Hu, Kai Kang, Angie Chen, Sertan Girgin, Yongqin Xian, Andrew
 708 Lee, Nolan Ramsden, Leslie Baker, Madeleine Clare Elish, Varvara Krayvanova, Rishabh Joshi,
 709 Jiri Simsa, Yao-Yuan Yang, Piotr Ambroszczyk, Dipankar Ghosh, Arjun Kar, Yuan Shangguan,
 710 Yumeya Yamamori, Yaroslav Akulov, Andy Brock, Haotian Tang, Siddharth Vashishtha, Rich
 711 Munoz, Andreas Steiner, Kalyan Andra, Daniel Eppens, Qixuan Feng, Hayato Kobayashi, Sasha
 712 Goldshtain, Mona El Mahdy, Xin Wang, Jilei, Wang, Richard Killam, Tom Kwiatkowski, Kavya
 713 Koppaparu, Serena Zhan, Chao Jia, Alexei Bendebury, Sheryl Luo, Adrià Recasens, Timothy
 714 Knight, Jing Chen, Mohak Patel, YaGuang Li, Ben Withbroe, Dean Weesner, Kush Bhatia, Jie
 715 Ren, Danielle Eisenbud, Ebrahim Songhori, Yanhua Sun, Travis Choma, Tasos Kementsietsidis,
 716 Lucas Manning, Brian Roark, Wael Farhan, Jie Feng, Susheel Tatineni, James Cobon-Kerr, Yunjie
 717 Li, Lisa Anne Hendricks, Isaac Noble, Chris Breaux, Nate Kushman, Liqian Peng, Fuzhao Xue,
 718 Taylor Tobin, Jamie Rogers, Josh Lipschultz, Chris Alberti, Alexey Vlaskin, Mostafa Dehghani,
 719 Roshan Sharma, Tris Warkentin, Chen-Yu Lee, Benigno Uria, Da-Cheng Juan, Angad Chandorkar,
 720 Hila Sheftel, Ruibo Liu, Elnaz Davoodi, Borja De Balle Pigem, Kedar Dhamdhere, David Ross,
 721 Jonathan Hoech, Mahdis Mahdieh, Li Liu, Qiuja Li, Liam McCafferty, Chenxi Liu, Markus
 722 Mircea, Yunting Song, Omkar Savant, Alaa Saade, Colin Cherry, Vincent Hellendoorn, Siddharth
 723 Goyal, Paul Pucciarelli, David Vilar Torres, Zohar Yahav, Hyo Lee, Lars Lowe Sjoesund, Christo
 724 Kirov, Bo Chang, Deepanway Ghoshal, Lu Li, Gilles Baechler, Sébastien Pereira, Tara Sainath,
 725 Anudhyan Boral, Dominik Grewe, Afiep Halumi, Nguyen Minh Phu, Tianxiao Shen, Marco Tilio
 726 Ribeiro, Dhriti Varma, Alex Kaskasoli, Vlad Feinberg, Navneet Potti, Jarrod Kahn, Matheus
 727 Wisniewski, Shakir Mohamed, Arnar Mar Hrafnkelsson, Bobak Shahriari, Jean-Baptiste Lespiau,
 728 Lisa Patel, Legg Yeung, Tom Paine, Lantao Mei, Alex Ramirez, Rakesh Shivanna, Li Zhong, Josh
 729 Woodward, Guilherme Tubone, Samira Khan, Heng Chen, Elizabeth Nielsen, Catalin Ionescu,
 730 Utsav Prabhu, Mingcen Gao, Qingze Wang, Sean Augenstein, Neesha Subramaniam, Jason Chang,
 731 Fotis Iliopoulos, Jiaming Luo, Myriam Khan, Weicheng Kuo, Denis Teplyashin, Florence Perot,
 732 Logan Kilpatrick, Amir Globerson, Hongkun Yu, Anfal Siddiqui, Nick Sukhanov, Arun Kandoor,
 733 Umang Gupta, Marco Andreetto, Moran Ambar, Donnie Kim, Paweł Wesołowski, Sarah Perrin,
 734 Ben Limonchik, Wei Fan, Jim Stephan, Ian Stewart-Binks, Ryan Kappedal, Tong He, Sarah Cogan,
 735 Romina Datta, Tong Zhou, Jiayu Ye, Leandro Kieliger, Ana Ramalho, Kyle Kastner, Fabian
 736 Mentzer, Wei-Jen Ko, Arun Suggala, Tianhao Zhou, Shiraz Butt, Hana Strejček, Lior Belenki,
 737 Subhashini Venugopalan, Mingyang Ling, Evgenii Eltyshev, Yunxiao Deng, Geza Kovacs, Mukund
 738 Raghavachari, Hanjun Dai, Tal Schuster, Steven Schwarcz, Richard Nguyen, Arthur Nguyen, Gavin
 739 Buttimore, Shrestha Basu Mallick, Sudeep Gandhe, Seth Benjamin, Michal Jastrzebski, Le Yan,
 740 Sugato Basu, Chris Apps, Isabel Edkins, James Allingham, Immanuel Odisho, Tomas Kociský,
 741 Jewel Zhao, Linting Xue, Apoorv Reddy, Chrysovalantis Anastasiou, Aviel Atias, Sam Redmond,
 742 Kieran Milan, Nicolas Heess, Herman Schmit, Allan Dafoe, Daniel Andor, Tynan Gangwani,
 743 Anca Dragan, Sheng Zhang, Ashyana Kachra, Gang Wu, Siyang Xue, Kevin Aydin, Siqi Liu,
 744 Yuxiang Zhou, Mahan Malahi, Austin Wu, Siddharth Gopal, Candice Schumann, Peter Stys,
 745 Alek Wang, Mirek Olšák, Dangyi Liu, Christian Schallhart, Yiran Mao, Demetra Brady, Hao
 746 Xu, Tomas Mery, Chawin Sitawarin, Siva Velusamy, Tom Cobley, Alex Zhai, Christian Walder,
 747 Nitzan Katz, Ganesh Jawahar, Chinmay Kulkarni, Antoine Yang, Adam Paszke, Yinan Wang,
 748 Bogdan Damoc, Zalán Borsos, Ray Smith, Jinning Li, Mansi Gupta, Andrei Kapishnikov, Sushant
 749 Prakash, Florian Luisier, Rishabh Agarwal, Will Grathwohl, Kuangyuan Chen, Kehang Han,
 750 Nikhil Mehta, Andrew Over, Shekoofeh Azizi, Lei Meng, Niccolò Dal Santo, Kelvin Zheng, Jane
 751 Shapiro, Igor Petrovski, Jeffrey Hui, Amin Ghafouri, Jasper Snoek, James Qin, Mandy Jordan,
 752 Caitlin Sikora, Jonathan Malmaud, Yuheng Kuang, Aga Świertlik, Ruoxin Sang, Chongyang Shi,
 753 Leon Li, Andrew Rosenberg, Shubin Zhao, Andy Crawford, Jan-Thorsten Peter, Yun Lei, Xavier
 754 Garcia, Long Le, Todd Wang, Julien Amelot, Dave Orr, Praneeth Kacham, Dana Alon, Gladys
 755 Tyen, Abhinav Arora, James Lyon, Alex Kurakin, Mimi Ly, Theo Guidroz, Zhipeng Yan, Rina
 Panigrahy, Pingmei Xu, Thais Kagohara, Yong Cheng, Eric Noland, Jinhyuk Lee, Jonathan Lee,
 Cathy Yip, Maria Wang, Efrat Nehorai, Alexander Bykovsky, Zhihao Shan, Ankit Bhagatwala,
 Chaochao Yan, Jie Tan, Guillermo Garrido, Dan Ethier, Nate Hurley, Grace Vesom, Xu Chen,
 Siyuan Qiao, Abhishek Nayyar, Julian Walker, Paramjit Sandhu, Mihaela Rosca, Danny Swisher,
 Mikhail Dektiarev, Josh Dillon, George-Cristian Muraru, Manuel Tragut, Artiom Myaskovsky,

756 David Reid, Marko Velic, Owen Xiao, Jasmine George, Mark Brand, Jing Li, Wenhao Yu, Shane
 757 Gu, Xiang Deng, François-Xavier Aubet, Soheil Hassas Yeganeh, Fred Alcober, Celine Smith,
 758 Trevor Cohn, Kay McKinney, Michael Tschanne, Ramesh Sampath, Gwoon Cheon, Liangchen
 759 Luo, Luyang Liu, Jordi Orbay, Hui Peng, Gabriela Botea, Xiaofan Zhang, Charles Yoon, Cesar
 760 Magalhaes, Paweł Stradomski, Ian Mackinnon, Steven Hemingray, Kumaran Venkatesan, Rhys
 761 May, Jaeyoun Kim, Alex Druinsky, Jingchen Ye, Zheng Xu, Terry Huang, Jad Al Abdallah, Adil
 762 Dostmohamed, Rachana Fellinger, Tsendsuren Munkhdalai, Akanksha Maurya, Peter Garst, Yin
 763 Zhang, Maxim Krikun, Simon Bucher, Aditya Srikanth Veerubhotla, Yixin Liu, Sheng Li, Nishesh
 764 Gupta, Jakub Adamek, Hanwen Chen, Bennett Orlando, Aleksandr Zaks, Joost van Amersfoort,
 765 Josh Camp, Hui Wan, HyunJeong Choe, Zhichun Wu, Kate Olszewska, Weiren Yu, Archita Vadali,
 766 Martin Scholz, Daniel De Freitas, Jason Lin, Amy Hua, Xin Liu, Frank Ding, Yichao Zhou, Boone
 767 Severson, Katerina Tsihlas, Samuel Yang, Tammo Spalink, Varun Yerram, Helena Pankov, Rory
 768 Blevins, Ben Vargas, Sarthak Jauhari, Matt Miecnikowski, Ming Zhang, Sandeep Kumar, Clement
 769 Farabet, Charline Le Lan, Sebastian Flennerhag, Yonatan Bitton, Ada Ma, Arthur Bražinskas,
 770 Eli Collins, Niharika Ahuja, Sneha Kudugunta, Anna Bortsova, Minh Giang, Wanzheng Zhu,
 771 Ed Chi, Scott Lundberg, Alexey Stern, Subha Puttagunta, Jing Xiong, Xiao Wu, Yash Pande,
 772 Amit Jhindal, Daniel Murphy, Jon Clark, Marc Brockschmidt, Maxine Deines, Kevin R. McKee,
 773 Dan Bahir, Jiajun Shen, Minh Truong, Daniel McDuff, Andrea Gesmundo, Edouard Rosseel,
 774 Bowen Liang, Ken Caluwaerts, Jessica Hamrick, Joseph Kready, Mary Cassin, Rishikesh Ingale,
 775 Li Lao, Scott Pollom, Yifan Ding, Wei He, Lizzeth Bellot, Joana Iljazi, Ramya Sree Boppana,
 776 Shan Han, Tara Thompson, Amr Khalifa, Anna Bulanova, Blagoj Mitrevski, Bo Pang, Emma
 777 Cooney, Tian Shi, Rey Coaguila, Tamar Yakar, Marc'aurelio Ranzato, Nikola Momchev, Chris
 778 Rawles, Zachary Charles, Young Maeng, Yuan Zhang, Rishabh Bansal, Xiaokai Zhao, Brian
 779 Albert, Yuan Yuan, Sudheendra Vijayanarasimhan, Roy Hirsch, Vinay Ramasesh, Kiran Vodrahalli,
 780 Xingyu Wang, Arushi Gupta, DJ Strouse, Jianmo Ni, Roma Patel, Gabe Taubman, Zhouyuan
 781 Huo, Dero Gharibian, Marianne Monteiro, Hoi Lam, Shobha Vasudevan, Aditi Chaudhary, Isabela
 782 Albuquerque, Kilol Gupta, Sebastian Riedel, Chaitra Hegde, Avraham Ruderman, András György,
 783 Marcus Wainwright, Ashwin Chaugule, Burcu Karagol Ayan, Tomer Levinboim, Sam Shleifer,
 784 Yogesh Kalley, Vahab Mirrokni, Abhishek Rao, Prabakar Radhakrishnan, Jay Hartford, Jialin
 785 Wu, Zhenhai Zhu, Francesco Bertolini, Hao Xiong, Nicolas Serrano, Hamish Tomlinson, Myle
 786 Ott, Yifan Chang, Mark Graham, Jian Li, Marco Liang, Xiangzhu Long, Sebastian Borgeaud,
 787 Yanif Ahmad, Alex Grills, Diana Mincu, Martin Izzard, Yuan Liu, Jinyu Xie, Louis O'Bryan,
 788 Sameera Ponda, Simon Tong, Michelle Liu, Dan Malkin, Khalid Salama, Yuankai Chen, Rohan
 789 Anil, Anand Rao, Rigel Swavely, Misha Bilenko, Nina Anderson, Tat Tan, Jing Xie, Xing Wu,
 790 Lijun Yu, Oriol Vinyals, Andrey Ryabtsev, Rumen Dangovski, Kate Baumli, Daniel Keysers,
 791 Christian Wright, Zoe Ashwood, Betty Chan, Artem Shtefan, Yaohui Guo, Ankur Bapna, Radu
 792 Soricut, Steven Pecht, Sabela Ramos, Rui Wang, Jiahao Cai, Trieu Trinh, Paul Barham, Linda
 793 Friso, Eli Stickgold, Xiangzhuo Ding, Siamak Shakeri, Diego Ardila, Eleftheria Briakou, Phil
 794 Culliton, Adam Raveret, Jingyu Cui, David Saxton, Subhrajit Roy, Javad Azizi, Pengcheng Yin,
 795 Lucia Loher, Andrew Bunner, Min Choi, Faruk Ahmed, Eric Li, Yin Li, Shengyang Dai, Michael
 796 Elabd, Sriram Ganapathy, Shivani Agrawal, Yiqing Hua, Paige Kunkle, Sujeevan Rajayogam, Arun
 797 Ahuja, Arthur Conmy, Alex Vasiloff, Parker Beak, Christopher Yew, Jayaram Mudigonda, Bartek
 798 Wydrowski, Jon Blanton, Zhengdong Wang, Yann Dauphin, Zhuo Xu, Martin Polacek, Xi Chen,
 799 Hexiang Hu, Pauline Sho, Markus Kunesch, Mehdi Hafezi Manshadi, Eliza Rutherford, Bo Li,
 800 Sissie Hsiao, Iain Barr, Alex Tudor, Matija Kecman, Arsha Nagrani, Vladimir Pchelin, Martin
 801 Sundermeyer, Aishwarya P S, Abhijit Karmarkar, Yi Gao, Grishma Chole, Olivier Bachem, Isabel
 802 Gao, Arturo BC, Matt Dibb, Mauro Verzetti, Felix Hernandez-Campos, Yana Lunts, Matthew
 803 Johnson, Julia Di Trapani, Raphael Koster, Idan Brusilovsky, Binbin Xiong, Megha Mohabey, Han
 804 Ke, Joe Zou, Tea Sabolić, Víctor Campos, John Palowitch, Alex Morris, Linhai Qiu, Pranavaraj
 805 Ponnuramu, Fangtao Li, Vivek Sharma, Kiranbir Sodhia, Kaan Tekelioglu, Aleksandr Chuklin,
 806 Madhavi Yenugula, Erika Gemzer, Theofilos Strinopoulos, Sam El-Husseini, Huiyu Wang, Yan
 807 Zhong, Edouard Leurent, Paul Natsev, Weijun Wang, Dre Mahaarachchi, Tao Zhu, Songyou Peng,
 808 Sami Alabed, Cheng-Chun Lee, Anthony Brohan, Arthur Szlam, GS Oh, Anton Kovsharov, Jenny
 809 Lee, Renee Wong, Megan Barnes, Gregory Thornton, Felix Gimeno, Omer Levy, Martin Sevenich,
 Melvin Johnson, Jonathan Mallinson, Robert Dadashi, Ziyue Wang, Qingchun Ren, Preethi Lahoti,
 Arka Dhar, Josh Feldman, Dan Zheng, Thatcher Ulrich, Liviu Panait, Michiel Blokzijl, Cip
 Baetu, Josip Matak, Jitendra Harlalka, Maulik Shah, Tal Marian, Daniel von Dincklage, Cosmo
 Du, Ruy Ley-Wild, Bethanie Brownfield, Max Schumacher, Yury Stuken, Shadi Noghabi, Sonal
 Gupta, Xiaoqi Ren, Eric Malmi, Felix Weissenberger, Blanca Huergo, Maria Bauza, Thomas

810 Lampe, Arthur Douillard, Mojtaba Seyedhosseini, Roy Frostig, Zoubin Ghahramani, Kelvin
 811 Nguyen, Kashyap Krishnakumar, Chengxi Ye, Rahul Gupta, Alireza Nazari, Robert Geirhos, Pete
 812 Shaw, Ahmed Eleryan, Dima Damen, Jennimaria Palomaki, Ted Xiao, Qiyin Wu, Quan Yuan,
 813 Phoenix Meadowlark, Matthew Bilotti, Raymond Lin, Mukund Sridhar, Yannick Schroecker,
 814 Da-Woon Chung, Jincheng Luo, Trevor Strohman, Tianlin Liu, Anne Zheng, Jesse Emond, Wei
 815 Wang, Andrew Lampinen, Toshiyuki Fukuzawa, Folawiyo Campbell-Ajala, Monica Roy, James
 816 Lee-Thorp, Lily Wang, Iftekhar Naim, Tony, Nguy ên, Guy Bensky, Aditya Gupta, Dominika
 817 Rogozińska, Justin Fu, Thanumalayan Sankaranarayana Pillai, Petar Veličković, Shahar Drath,
 818 Philipp Neubeck, Vaibhav Tulsyan, Arseniy Klimovskiy, Don Metzler, Sage Stevens, Angel
 819 Yeh, Junwei Yuan, Tianhe Yu, Kelvin Zhang, Alec Go, Vincent Tsang, Ying Xu, Andy Wan,
 820 Isaac Galatzer-Levy, Sam Sobell, Abodunrinwa Toki, Elizabeth Salesky, Wenlei Zhou, Diego
 821 Antognini, Sholto Douglas, Shimu Wu, Adam Lelkes, Frank Kim, Paul Cavallaro, Ana Salazar,
 822 Yuchi Liu, James Besley, Tiziana Refice, Yiling Jia, Zhang Li, Michal Sokolik, Arvind Kannan,
 823 Jon Simon, Jo Chick, Avia Aharon, Meet Gandhi, Mayank Daswani, Keyvan Amiri, Vighnesh
 824 Birodkar, Abe Ittycheriah, Peter Grabowski, Oscar Chang, Charles Sutton, Zhixin, Lai, Umesh
 825 Telang, Susie Sargsyan, Tao Jiang, Raphael Hoffmann, Nicole Brichtova, Matteo Hessel, Jonathan
 826 Halcrow, Sammy Jerome, Geoff Brown, Alex Tomala, Elena Buchatskaya, Dian Yu, Sachit
 827 Menon, Pol Moreno, Yuguo Liao, Vicky Zayats, Luming Tang, SQ Mah, Ashish Shenoy, Alex
 828 Siegman, Majid Hadian, Okwan Kwon, Tao Tu, Nima Khajehnouri, Ryan Foley, Parisa Haghani,
 829 Zhongru Wu, Vaishakh Keshava, Khyati Gupta, Tony Bruguer, Rui Yao, Danny Karmon, Luisa
 830 Zintgraf, Zhicheng Wang, Enrique Piqueras, Junehyuk Jung, Jenny Brennan, Diego Machado,
 831 Marissa Giustina, MH Tessler, Kamyu Lee, Qiao Zhang, Joss Moore, Kaspar Daugaard, Alexander
 832 Frömmgen, Jennifer Beattie, Fred Zhang, Daniel Kasenberg, Ty Geri, Danfeng Qin, Gaurav Singh
 833 Tomar, Tom Ouyang, Tianli Yu, Luowei Zhou, Rajiv Mathews, Andy Davis, Yaoyiran Li, Jai
 834 Gupta, Damion Yates, Linda Deng, Elizabeth Kemp, Ga-Young Joung, Sergei Vassilvitskii, Mandy
 835 Guo, Pallavi LV, Dave Dopson, Sami Lachgar, Lara McConaughey, Himadri Choudhury, Dragos
 836 Dena, Aaron Cohen, Joshua Ainslie, Sergey Levi, Parthasarathy Gopavarapu, Polina Zablotskaia,
 837 Hugo Vallet, Sanaz Bahargam, Xiaodan Tang, Nenad Tomasev, Ethan Dyer, Daniel Balle, Hongrae
 838 Lee, William Bono, Jorge Gonzalez Mendez, Vadim Zubov, Shentao Yang, Ivor Rendulic, Yanyan
 839 Zheng, Andrew Hogue, Golan Pundak, Ralph Leith, Avishkar Bhoopchand, Michael Han, Mislav
 840 Žanić, Tom Schaul, Manolis Delakakis, Tejas Iyer, Guanyu Wang, Harman Singh, Abdelrahman
 841 Abdelhamed, Tara Thomas, Siddhartha Brahma, Hilal Dib, Naveen Kumar, Wenxuan Zhou, Liang
 842 Bai, Pushkar Mishra, Jiao Sun, Valentin Anklin, Roykrong Sukkerd, Lauren Agubuzu, Anton
 843 Briukhov, Anmol Gulati, Maximilian Sieb, Fabio Pardo, Sara Nasso, Junquan Chen, Kexin Zhu,
 844 Tiberiu Sosea, Alex Goldin, Keith Rush, Spurthi Amba Hombaiah, Andreas Noever, Allan Zhou,
 845 Sam Haves, Mary Phuong, Jake Ades, Yi ting Chen, Lin Yang, Joseph Pagadora, Stan Bileschi,
 846 Victor Cotruta, Rachel Saputro, Arijit Pramanik, Sean Ammirati, Dan Garrette, Kevin Villela, Tim
 847 Blyth, Canfer Akbulut, Neha Jha, Alban Rustemi, Arissa Wongpanich, Chirag Nagpal, Yonghui
 848 Wu, Morgane Rivière, Sergey Kishchenko, Pranesh Srinivasan, Alice Chen, Animesh Sinha, Trang
 849 Pham, Bill Jia, Tom Hennigan, Anton Bakalov, Nithya Attaluri, Drew Garmon, Daniel Rodriguez,
 850 Dawid Wegner, Wenhao Jia, Evan Senter, Noah Fiedel, Denis Petek, Yuchuan Liu, Cassidy Hardin,
 851 Harshal Tushar Lehri, Joao Carreira, Sara Smoot, Marcel Prasetya, Nami Akazawa, Anca Stefanou,br/>
 852 Chia-Hua Ho, Anelia Angelova, Kate Lin, Min Kim, Charles Chen, Marcin Sieniek, Alice Li,
 853 Tongfei Guo, Sorin Baltateanu, Pouya Tafti, Michael Wunder, Nadav Olmert, Divyansh Shukla,
 854 Jingwei Shen, Neel Kovelamudi, Balaji Venkatraman, Seth Neel, Romal Thoppilan, Jerome Connor,
 855 Frederik Benzing, Axel Stjerngren, Golnaz Ghiasi, Alex Polozov, Joshua Howland, Theophane
 856 Weber, Justin Chiu, Ganesh Poomal Girirajan, Andreas Terzis, Pidong Wang, Fangda Li, Yoav Ben
 857 Shalom, Dinesh Tewari, Matthew Denton, Roee Aharoni, Norbert Kalb, Heri Zhao, Junlin Zhang,
 858 Angelos Filos, Matthew Rahtz, Lalit Jain, Connie Fan, Vitor Rodrigues, Ruth Wang, Richard
 859 Shin, Jacob Austin, Roman Ring, Mariella Sanchez-Vargas, Mehadi Hassen, Ido Kessler, Uri Alon,
 860 Gufeng Zhang, Wenhua Chen, Yenai Ma, Xiance Si, Le Hou, Azalia Mirhoseini, Marc Wilson,
 861 Geoff Bacon, Becca Roelofs, Lei Shu, Gautam Vasudevan, Jonas Adler, Artur Dwornik, Tayfun
 862 Terzi, Matt Lawlor, Harry Askham, Mike Bernico, Xuanyi Dong, Chris Hidey, Kevin Kilgour,
 863 Gaël Liu, Surya Bhupatiraju, Luke Leonhard, Siqi Zuo, Partha Talukdar, Qing Wei, Aliaksei
 Severyn, Vít Listík, Jong Lee, Aditya Tripathi, SK Park, Yossi Matias, Hao Liu, Alex Ruiz, Rajesh
 Jayaram, Jackson Tolins, Pierre Marcenac, Yiming Wang, Bryan Seybold, Henry Prior, Deepak
 Sharma, Jack Weber, Mikhail Sirotenko, Yunhsuan Sung, Dayou Du, Ellie Pavlick, Stefan Zinke,
 Markus Freitag, Max Dylla, Montse Gonzalez Arenas, Natan Potikha, Omer Goldman, Connie
 Tao, Rachita Chhaparia, Maria Voitovich, Pawan Dogra, Andrija Ražnatović, Zak Tsai, Chong

864 You, Oleaser Johnson, George Tucker, Chenjie Gu, Jae Yoo, Maryam Majzoubi, Valentin Gabeur,
 865 Bahram Raad, Rocky Rhodes, Kashyap Kolipaka, Heidi Howard, Geta Sampemane, Benny Li,
 866 Chulayuth Asawaroengchai, Duy Nguyen, Chiyuan Zhang, Timothee Cour, Xinxin Yu, Zhao Fu,
 867 Joe Jiang, Po-Sen Huang, Gabriela Surita, Iñaki Iturrate, Yael Karov, Michael Collins, Martin
 868 Baeuml, Fabian Fuchs, Shilpa Shetty, Swaroop Ramaswamy, Sayna Ebrahimi, Qiuchen Guo,
 869 Jeremy Shar, Gabe Barth-Maron, Sravanti Addepalli, Bryan Richter, Chin-Yi Cheng, Eugénie
 870 Rives, Fei Zheng, Johannes Griesser, Nishanth Dikkala, Yoel Zeldes, Ilkin Safarli, Dipanjan Das,
 871 Himanshu Srivastava, Sadh MNM Khan, Xin Li, Aditya Pandey, Larisa Markeeva, Dan Belov, Qiqi
 872 Yan, Mikołaj Rybiński, Tao Chen, Megha Nawhal, Michael Quinn, Vineetha Govindaraj, Sarah
 873 York, Reed Roberts, Roopal Garg, Namrata Godbole, Jake Abernethy, Anil Das, Lam Nguyen
 874 Thiet, Jonathan Tompson, John Nham, Neera Vats, Ben Caine, Wesley Helmholz, Francesco
 875 Pongetti, Yeongil Ko, James An, Clara Huiyi Hu, Yu-Cheng Ling, Julia Pawar, Robert Leland,
 876 Keisuke Kinoshita, Waleed Khawaja, Marco Selvi, Eugene Ie, Danila Sinopalnikov, Lev Proleev,
 877 Nilesh Tripuraneni, Michele Bevilacqua, Seungji Lee, Clayton Sanford, Dan Suh, Dustin Tran,
 878 Jeff Dean, Simon Baumgartner, Jens Heitkaemper, Sagar Gubbi, Kristina Toutanova, Yichong Xu,
 879 Chandu Thekkath, Keran Rong, Palak Jain, Annie Xie, Yan Virin, Yang Li, Lubo Litchev, Richard
 880 Powell, Tarun Bharti, Adam Kraft, Nan Hua, Marissa Ikonomidis, Ayal Hitron, Sanjiv Kumar,
 881 Loic Matthey, Sophie Bridgers, Lauren Lax, Ishaan Malhi, Ondrej Skopek, Ashish Gupta, Jiawei
 882 Cao, Michelle Rasquinha, Siim Pöder, Wojciech Stokowiec, Nicholas Roth, Guowang Li, Michaël
 883 Sander, Joshua Kessinger, Vihan Jain, Edward Loper, Wonpyo Park, Michal Yarom, Lijun Cheng,
 884 Guru Guruganesh, Kanishka Rao, Yan Li, Catarina Barros, Mikhail Sushkov, Chun-Sung Ferng,
 885 Rohin Shah, Ophir Aharoni, Ravin Kumar, Tim McConnell, Peiran Li, Chen Wang, Fernando
 886 Pereira, Craig Swanson, Fayaz Jamil, Yan Xiong, Anitha Vijayakumar, Prakash Shroff, Kedar
 887 Soparkar, Jindong Gu, Livio Baldini Soares, Eric Wang, Kushal Majmundar, Aurora Wei, Kai
 888 Bailey, Nora Kassner, Chizu Kawamoto, Goran Žužić, Victor Gomes, Abhirut Gupta, Michael
 889 Guzman, Ishita Dasgupta, Xinyi Bai, Zhufeng Pan, Francesco Piccinno, Hadas Natalie Vogel,
 890 Octavio Ponce, Adrian Hutter, Paul Chang, Pan-Pan Jiang, Ionel Gog, Vlad Ionescu, James
 891 Manyika, Fabian Pedregosa, Harry Ragan, Zach Behrman, Ryan Mullins, Coline Devin, Aroonalok
 892 Pyne, Swapnil Gawde, Martin Chadwick, Yiming Gu, Sasan Tavakkol, Andy Twigg, Naman
 893 Goyal, Ndidi Elue, Anna Goldie, Srinivasan Venkatachary, Hongliang Fei, Ziqiang Feng, Marvin
 894 Ritter, Isabel Leal, Sudeep Dasari, Pei Sun, Alif Raditya Rochman, Brendan O'Donoghue, Yuchen
 895 Liu, Jim Sproch, Kai Chen, Natalie Clay, Slav Petrov, Sailesh Sidhwani, Ioana Mihailescu, Alex
 896 Panagopoulos, AJ Piergiovanni, Yunfei Bai, George Powell, Deep Karkhanis, Trevor Yacovone,
 897 Petr Mitrichev, Joe Kovac, Dave Uthus, Amir Yazdanbakhsh, David Amos, Steven Zheng, Bing
 898 Zhang, Jin Miao, Bhuvana Ramabhadran, Soroush Radpour, Shantanu Thakoor, Josh Newlan, Oran
 899 Lang, Orion Jankowski, Shikhar Bharadwaj, Jean-Michel Sarr, Shereen Ashraf, Sneha Mondal, Jun
 900 Yan, Ankit Singh Rawat, Sarmishta Velury, Greg Kochanski, Tom Eccles, Franz Och, Abhanshu
 901 Sharma, Ethan Mahintorabi, Alex Gurney, Carrie Muir, Vered Cohen, Saksham Thakur, Adam
 902 Bloniarz, Asier Mujika, Alexander Pritzel, Paul Caron, Altaf Rahman, Fiona Lang, Yasumasa Onoe,
 903 Petar Sirkovic, Jay Hoover, Ying Jian, Pablo Duque, Arun Narayanan, David Soergel, Alex Haig,
 904 Loren Maggiore, Shyamal Buch, Josef Dean, Ilya Figotin, Igor Karpov, Shaleen Gupta, Denny
 905 Zhou, Muhan Huang, Ashwin Vaswani, Christopher Semturs, Kaushik Shivakumar, Yu Watanabe,
 906 Vinodh Kumar Rajendran, Eva Lu, Yanhan Hou, Wenting Ye, Shikhar Vashishth, Nana Nti, Vytenis
 907 Sakenas, Darren Ni, Doug DeCarlo, Michael Bendersky, Sumit Bagri, Nacho Cano, Elijah Peake,
 908 Simon Tokumine, Varun Godbole, Carlos Guía, Tanya Lando, Vittorio Selo, Seher Ellis, Danny
 909 Tarlow, Daniel Gillick, Alessandro Epasto, Siddhartha Reddy Jonnalagadda, Meng Wei, Meiyang
 910 Xie, Ankur Taly, Michela Paganini, Mukund Sundararajan, Daniel Toyama, Ting Yu, Dessie
 911 Petrova, Aneesh Pappu, Rohan Agrawal, Senaka Buthpitiya, Justin Frye, Thomas Buschmann,
 912 Remi Crocker, Marco Tagliasacchi, Mengchao Wang, Da Huang, Sagi Perel, Brian Wieder, Hideto
 913 Kazawa, Weiyue Wang, Jeremy Cole, Himanshu Gupta, Ben Golan, Seojin Bang, Nitish Kulkarni,
 914 Ken Franko, Casper Liu, Doug Reid, Sid Dalmia, Jay Whang, Kevin Cen, Prasha Sundaram, Johan
 915 Ferret, Berivan Isik, Lucian Ionita, Guan Sun, Anna Shekhawat, Muqthar Mohammad, Philip
 916 Pham, Ronny Huang, Karthik Raman, Xingyi Zhou, Ross Mcilroy, Austin Myers, Sheng Peng,
 917 Jacob Scott, Paul Covington, Sofia Erell, Pratik Joshi, João Gabriel Oliveira, Natasha Noy, Tajwar
 Nasir, Jake Walker, Vera Axelrod, Tim Dozat, Pu Han, Chun-Te Chu, Eugene Weinstein, Anand
 Shukla, Shreyas Chandrakaladharan, Petra Poklukar, Bonnie Li, Ye Jin, Prem Erubetidine, Steven
 Hansen, Avigail Dabush, Alon Jacovi, Samrat Phatale, Chen Zhu, Steven Baker, Mo Shomrat, Yang
 Xiao, Jean Pouget-Abadie, Mingyang Zhang, Fanny Wei, Yang Song, Helen King, Yiling Huang,
 918 Yun Zhu, Ruoxi Sun, Juliana Vicente Franco, Chu-Cheng Lin, Sho Arora, Hui, Li, Vivian Xia,

918 Luke Vilnis, Mariano Schain, Kaiz Alarakyia, Laurel Prince, Aaron Phillips, Caleb Habtegebriel,
 919 Luyao Xu, Huan Gui, Santiago Ontanon, Lora Aroyo, Karan Gill, Peggy Lu, Yash Katariya,
 920 Dhruv Madeka, Shankar Krishnan, Shubha Srinivas Raghvendra, James Freedman, Yi Tay, Gaurav
 921 Menghani, Peter Choy, Nishita Shetty, Dan Abolafia, Doron Kukliansky, Edward Chou, Jared
 922 Lichtarge, Ken Burke, Ben Coleman, Dee Guo, Larry Jin, Indro Bhattacharya, Victoria Langston,
 923 Yiming Li, Suyog Kotecha, Alex Yakubovich, Xinyun Chen, Petre Petrov, Tolly Powell, Yanzhang
 924 He, Corbin Quick, Kanav Garg, Dawsen Hwang, Yang Lu, Srinadh Bhojanapalli, Kristian Kjems,
 925 Ramin Mehran, Aaron Archer, Hado van Hasselt, Ashwin Balakrishna, JK Kearns, Meiqi Guo,
 926 Jason Riesa, Mikita Sazanovich, Xu Gao, Chris Sauer, Chengrun Yang, XiangHai Sheng, Thomas
 927 Jimma, Wouter Van Gansbeke, Vitaly Nikolaev, Wei Wei, Katie Millican, Ruizhe Zhao, Justin
 928 Snyder, Levent Bolelli, Maura O'Brien, Shawn Xu, Fei Xia, Wentao Yuan, Arvind Neelakantan,
 929 David Barker, Sachin Yadav, Hannah Kirkwood, Farooq Ahmad, Joel Wee, Jordan Grimstad, Boyu
 930 Wang, Matthew Wiethoff, Shane Settle, Miaosen Wang, Charles Blundell, Jingjing Chen, Chris
 931 Duvarney, Grace Hu, Olaf Ronneberger, Alex Lee, Yuanzhen Li, Abhishek Chakladar, Alena
 932 Butryna, Georgios Evangelopoulos, Guillaume Desjardins, Jonni Kanerva, Henry Wang, Averi
 933 Nowak, Nick Li, Alyssa Loo, Art Khurshudov, Laurent El Shafey, Nagabhushan Baddi, Karel Lenc,
 934 Yasaman Razeghi, Tom Lieber, Amer Sinha, Xiao Ma, Yao Su, James Huang, Asahi Ushio, Hanna
 935 Klimczak-Plucińska, Kareem Mohamed, JD Chen, Simon Osindero, Stav Ginzburg, Lampros
 936 Lamprou, Vasilisa Bashlovkina, Duc-Hieu Tran, Ali Khodaei, Ankit Anand, Yixian Di, Ramy
 937 Eskander, Manish Reddy Vuyyuru, Jasmine Liu, Aishwarya Kamath, Roman Goldenberg, Mathias
 938 Bellaiche, Juliette Pluto, Bill Rosgen, Hassan Mansoor, William Wong, Suhas Ganesh, Eric Bailey,
 939 Scott Baird, Dan Deutsch, Jinoo Baek, Xuhui Jia, Chansoo Lee, Abe Friesen, Nathaniel Braun, Kate
 940 Lee, Amayika Panda, Steven M. Hernandez, Duncan Williams, Jianqiao Liu, Ethan Liang, Arnaud
 941 Autef, Emily Pitler, Deepali Jain, Phoebe Kirk, Oskar Bunyan, Jaume Sanchez Elias, Tongxin Yin,
 942 Machel Reid, Aedan Pope, Nikita Putikhin, Bidisha Samanta, Sergio Guadarrama, Dahun Kim,
 943 Simon Rowe, Marcella Valentine, Geng Yan, Alex Salcianu, David Silver, Gan Song, Richa Singh,
 944 Shuai Ye, Hannah DeBalsi, Majd Al Merey, Eran Ofek, Albert Webson, Shibli Mourad, Ashwin
 945 Kakarla, Silvio Lattanzi, Nick Roy, Evgeny Sluzhaev, Christina Butterfield, Alessio Tonioni,
 946 Nathan Waters, Sudhindra Kopalle, Jason Chase, James Cohan, Girish Ramchandra Rao, Robert
 947 Berry, Michael Voznesensky, Shuguang Hu, Kristen Chiafullo, Sharat Chikkerur, George Scrivener,
 948 Ivy Zheng, Jeremy Wiesner, Wolfgang Macherey, Timothy Lillicrap, Fei Liu, Brian Walker, David
 949 Welling, Elinor Davies, Yangsibo Huang, Lijie Ren, Nir Shabat, Alessandro Agostini, Mariko
 950 Iinuma, Dustin Zelle, Rohit Sathyanarayana, Andrea D'olimpio, Morgan Redshaw, Matt Ginsberg,
 951 Ashwin Murthy, Mark Geller, Tatiana Matejovicova, Ayan Chakrabarti, Ryan Julian, Christine
 952 Chan, Qiong Hu, Daniel Jarrett, Manu Agarwal, Jeshwanth Challagundla, Tao Li, Sandeep Tata,
 953 Wen Ding, Maya Meng, Zhuyun Dai, Giulia Vezzani, Shefali Garg, Jannis Bulian, Mary Jasarevic,
 954 Honglong Cai, Harish Rajamani, Adam Santoro, Florian Hartmann, Chen Liang, Bartek Perz,
 955 Apoorv Jindal, Fan Bu, Sungyong Seo, Ryan Poplin, Adrian Goedeckemeyer, Badih Ghazi, Nikhil
 956 Khadke, Leon Liu, Kevin Mather, Mingda Zhang, Ali Shah, Alex Chen, Jinliang Wei, Keshav
 957 Shivam, Yuan Cao, Donghyun Cho, Angelo Scorzafra, Michael Moffitt, Clara Barbu, Ivan
 958 Jurin, Ming-Wei Chang, Hongbin Liu, Hao Zheng, Shachi Dave, Christine Kaeser-Chen, Xiaobin
 959 Yu, Alvin Abdagic, Lucas Gonzalez, Yanping Huang, Peilin Zhong, Cordelia Schmid, Bryce
 960 Petri, Alex Wertheim, Jifan Zhu, Hoang Nguyen, Kaiyang Ji, Yanqi Zhou, Tao Zhou, Fangxiaoyu
 961 Feng, Regev Cohen, David Rim, Shubham Milind Phal, Petko Georgiev, Ariel Brand, Yue Ma,
 962 Wei Li, Somit Gupta, Chao Wang, Pavel Dubov, Jean Tarbouriech, Kingshuk Majumder, Huijian
 963 Li, Norman Rink, Apurv Suman, Yang Guo, Yinghao Sun, Arun Nair, Xiaowei Xu, Mohamed
 964 Elhawaty, Rodrigo Cabrera, Guangxing Han, Julian Eisenschlos, Junwen Bai, Yuqi Li, Yamini
 965 Bansal, Thibault Sellam, Mina Khan, Hung Nguyen, Justin Mao-Jones, Nikos Parotsidis, Jake
 966 Marcus, Cindy Fan, Roland Zimmermann, Yony Kochinski, Laura Graesser, Feryal Behbahani,
 967 Alvaro Caceres, Michael Riley, Patrick Kane, Sandra Lefdal, Rob Willoughby, Paul Vicol, Lun
 968 Wang, Shujian Zhang, Ashleah Gill, Yu Liang, Gautam Prasad, Soroosh Mariooryad, Mehran
 969 Kazemi, Zifeng Wang, Kritika Muralidharan, Paul Voigtlaender, Jeffrey Zhao, Huanjie Zhou,
 970 Nina D'Souza, Aditi Mavalankar, Séb Arnold, Nick Young, Obaid Sarvana, Chace Lee, Milad
 971 Nasr, Tingting Zou, Seokhwan Kim, Lukas Haas, Kaushal Patel, Neslihan Bulut, David Parkinson,
 Courtney Biles, Dmitry Kalashnikov, Chi Ming To, Aviral Kumar, Jessica Austin, Alex Greve,
 Lei Zhang, Megha Goel, Yeqing Li, Sergey Yaroshenko, Max Chang, Abhishek Jindal, Geoff
 Clark, Hagai Taitelbaum, Dale Johnson, Ofir Roval, Jeongwoo Ko, Anhad Mohananey, Christian
 Schuler, Shenil Dodhia, Ruichao Li, Kazuki Osawa, Claire Cui, Peng Xu, Rushin Shah, Tao Huang,
 Ela Gruzecka, Nathan Clement, Mudit Verma, Olcan Sercinoglu, Hai Qian, Viral Shah, Masa

972 Yamaguchi, Abhinit Modi, Takahiro Kosakai, Thomas Strohmann, Junhao Zeng, Beliz Gunel, Jun
 973 Qian, Austin Tarango, Krzysztof Jastrzębski, Robert David, Jyn Shan, Parker Schuh, Kunal Lad,
 974 Willi Gierke, Mukundan Madhavan, Xinyi Chen, Mark Kurzeja, Rebeca Santamaria-Fernandez,
 975 Dawn Chen, Alexandra Cordell, Yuri Chervonyi, Frankie Garcia, Nithish Kannen, Vincent Perot,
 976 Nan Ding, Shlomi Cohen-Ganor, Victor Lavrenko, Junru Wu, Georgie Evans, Cicero Nogueira dos
 977 Santos, Madhavi Sewak, Ashley Brown, Andrew Hard, Joan Puigcerver, Zeyu Zheng, Yizhong
 978 Liang, Evgeny Gladchenko, Reeve Ingle, Uri First, Pierre Sermanet, Charlotte Magister, Mihajlo
 979 Velimirović, Sashank Reddi, Susanna Ricco, Eirikur Agustsson, Hartwig Adam, Nir Levine, David
 980 Gaddy, Dan Holtmann-Rice, Xuanhui Wang, Ashutosh Sathe, Abhijit Guha Roy, Blaž Bratanič,
 981 Alen Carin, Harsh Mehta, Silvano Bonacina, Nicola De Cao, Mara Finkelstein, Verena Rieser,
 982 Xinyi Wu, Florent Altché, Dylan Scardinaro, Li Li, Nino Vieillard, Nikhil Sethi, Garrett Tanzer,
 983 Zhi Xing, Shibo Wang, Parul Bhatia, Gui Citovsky, Thomas Anthony, Sharon Lin, Tianze Shi,
 984 Shoshana Jakobovits, Gena Gibson, Raj Apte, Lisa Lee, Mingqing Chen, Arunkumar Byravan,
 985 Petros Maniatis, Kellie Webster, Andrew Dai, Pu-Chin Chen, Jiaqi Pan, Asya Fadeeva, Zach
 986 Gleicher, Thang Luong, and Niket Kumar Bhumihar. Gemini 2.5: Pushing the frontier with
 987 advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025.
 988 URL <https://arxiv.org/abs/2507.06261>.

989 Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving
 990 factuality and reasoning in language models through multiagent debate. In *Forty-first International
 991 Conference on Machine Learning*, 2023.

992 Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander
 993 Smola. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint
 994 arXiv:2003.06505*, 2020.

995 Jonas Eschmann. Reward function design in reinforcement learning. *Reinforcement learning
 996 algorithms: Analysis and Applications*, pp. 25–33, 2021.

997 Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. Auto-
 998 sklearn 2.0: Hands-free automl via meta-learning. *arXiv 2020*. *arXiv preprint arXiv:2007.04074*,
 999 2022.

1000 Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu,
 1001 Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms
 1002 via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025a.

1003 Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu,
 1004 Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms
 1005 via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025b.

1006 Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang,
 1007 Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent
 1008 collaborative framework. *arXiv preprint arXiv:2308.00352*, 3(4):6, 2023.

1009 Yue Hu, Yuzhu Cai, Yixin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and
 1010 Siheng Chen. Self-evolving multi-agent collaboration networks for software development. *arXiv
 1011 preprint arXiv:2410.16946*, 2024.

1012 Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. Mlagentbench: Evaluating language agents
 1013 on machine learning experimentation. *arXiv preprint arXiv:2310.03302*, 2023.

1014 Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixin Xu, Ian Kaplan, Deniss Jacenko, and
 1015 Yuxiang Wu. Aide: Ai-driven exploration in the space of code, 2025. URL <https://arxiv.org/abs/2502.13138>.

1016 Haifeng Jin, François Fleuret, Qingquan Song, and Xia Hu. Autokeras: An automl library for deep
 1017 learning. *Journal of machine Learning research*, 24(6):1–6, 2023.

1018 Timo Kaufmann, Paul Weng, Viktor Bengs, and Eyke Hüllermeier. A survey of reinforcement
 1019 learning from human feedback. *arXiv preprint arXiv:2312.14925*, 10, 2023.

1026 Erin LeDell and Sebastien Poirier. H2o automl: Scalable automatic machine learning. In *Proceedings*
 1027 of the AutoML Workshop at ICML, volume 2020, pp. 24, 2020.

1028

1029 Ziming Li, Qianbo Zang, David Ma, Jiawei Guo, Tuney Zheng, Minghao Liu, Xinyao Niu, Yue Wang,
 1030 Jian Yang, Jiaheng Liu, et al. Autokaggle: A multi-agent framework for autonomous data science
 1031 competitions. *arXiv preprint arXiv:2410.20424*, 2024.

1032 Sijia Liu, Parikshit Ram, Deepak Vijaykeerthy, Djallel Bouneffouf, Gregory Bramble, Horst Samu-
 1033 lowitz, Dakuo Wang, Andrew Conn, and Alexander Gray. An admm based framework for automl
 1034 pipeline configuration. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34,
 1035 pp. 4892–4899, 2020.

1036

1037 Zexi Liu, Yuzhu Cai, Xinyu Zhu, Yujie Zheng, Runkun Chen, Ying Wen, Yanfeng Wang, Weinan
 1038 E, and Siheng Chen. MI-master: Towards ai-for-ai via integration of exploration and reasoning.
 1039 *arXiv preprint arXiv:2506.16499*, 2025a.

1040

1041 Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min
 1042 Lin. Understanding r1-zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*,
 1043 2025b.

1044

1045 Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher
 1046 Potts. Learning word vectors for sentiment analysis. In Dekang Lin, Yuji Matsumoto, and Rada
 1047 Mihalcea (eds.), *Proceedings of the 49th Annual Meeting of the Association for Computational
 1048 Linguistics: Human Language Technologies*, pp. 142–150, Portland, Oregon, USA, June 2011.
 1049 Association for Computational Linguistics. URL <https://aclanthology.org/P11-1015/>.

1050

1051 Tula Masterman, Sandi Besen, Mason Sawtell, and Alex Chao. The landscape of emerging ai agent
 1052 architectures for reasoning, planning, and tool calling: A survey. *arXiv preprint arXiv:2404.11584*,
 1053 2024.

1054

1055 Felix Mohr, Marcel Wever, and Eyke Hüllermeier. MI-plan: Automated machine learning via
 1056 hierarchical planning. *Machine Learning*, 107:1495–1515, 2018.

1057

1058 Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating
 1059 machine learning. In *Workshop on automatic machine learning*, pp. 66–74. PMLR, 2016.

1060

1061 OpenAI. Gpt-4o mini: Advancing cost-efficient intelligence. <https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/>, 2024. Accessed: 2025-01-23.

1062

1063 OpenAI. Introducing gpt 5. <https://openai.com/index/introducing-gpt-5/>, 2025. Accessed:
 1064 2025-09-25.

1065

1066 Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong
 1067 Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow
 1068 instructions with human feedback. *Advances in neural information processing systems*, 35:27730–
 1069 27744, 2022.

1070

1071 Xianghe Pang, Shuo Tang, Rui Ye, Yuxin Xiong, Bolun Zhang, Yanfeng Wang, and Siheng Chen.
 1072 Self-alignment of large language models via monopolylogue-based social scene simulation. *arXiv
 1073 preprint arXiv:2402.05699*, 2024.

1074

1075 Peter S Park, Philipp Schoenegger, and Chongyang Zhu. Diminished diversity-of-thought in a
 1076 standard large language model. *Behavior Research Methods*, 56(6):5754–5770, 2024.

1077

1078 Pranav Putta, Edmund Mills, Naman Garg, Sumeet Motwani, Chelsea Finn, Divyansh Garg, and
 1079 Rafael Rafailov. Agent q: Advanced reasoning and learning for autonomous ai agents. *arXiv
 1079 preprint arXiv:2408.07199*, 2024.

1080

1081 Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen,
 1082 Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. *arXiv
 1083 preprint arXiv:2307.07924*, 2023.

1080 Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea
 1081 Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances*
 1082 *in Neural Information Processing Systems*, 36:53728–53741, 2023.

1083 John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy
 1084 optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

1085 Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng,
 1086 Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint*
 1087 *arXiv: 2409.19256*, 2024.

1088 Zhiqiang Tang, Haoyang Fang, Su Zhou, Taojiannan Yang, Zihan Zhong, Tony Hu, Katrin Kirchhoff,
 1089 and George Karypis. Autogluon-multimodal (automm): Supercharging multimodal automl with
 1090 foundation models. *arXiv preprint arXiv:2404.16233*, 2024.

1091 Patara Trirat, Wonyong Jeong, and Sung Ju Hwang. Automl-agent: A multi-agent llm framework for
 1092 full-pipeline automl. *arXiv preprint arXiv:2410.02958*, 2024.

1093 Shuhe Wang, Shengyu Zhang, Jie Zhang, Runyi Hu, Xiaoya Li, Tianwei Zhang, Jiwei Li, Fei Wu,
 1094 Guoyin Wang, and Eduard Hovy. Reinforcement learning enhanced llms: A survey. *arXiv preprint*
 1095 *arXiv:2412.10400*, 2024.

1096 Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Kefan Yu,
 1097 Minh Nhat Nguyen, Licheng Liu, Eli Gottlieb, et al. Ragen: Understanding self-evolution in llm
 1098 agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025.

1099 Weimin Xiong, Yifan Song, Xiutian Zhao, Wenhao Wu, Xun Wang, Ke Wang, Cheng Li, Wei Peng,
 1100 and Sujian Li. Watch every step! llm agent learning via iterative step-level process refinement. In
 1101 *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp.
 1102 1556–1572, 2024.

1103 Fengli Xu, Qianyue Hao, Zefang Zong, Jingwei Wang, Yunke Zhang, Jingyi Wang, Xiaochong Lan,
 1104 Jiahui Gong, Tianjian Ouyang, Fanjin Meng, et al. Towards large reasoning models: A survey of
 1105 reinforced reasoning with large language models. *arXiv preprint arXiv:2501.09686*, 2025.

1106 An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li,
 1107 Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint*
 1108 *arXiv:2412.15115*, 2024.

1109 An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang
 1110 Gao, Chengan Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu,
 1111 Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin
 1112 Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang,
 1113 Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui
 1114 Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang
 1115 Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger
 1116 Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan
 1117 Qiu. Qwen3 technical report, 2025. URL <https://arxiv.org/abs/2505.09388>.

1118 Guibin Zhang, Hejia Geng, Xiaohang Yu, Zhenfei Yin, Zaibin Zhang, Zelin Tan, Heng Zhou,
 1119 Zhongzhi Li, Xiangyuan Xue, Yijiang Li, et al. The landscape of agentic reinforcement learning
 1120 for llms: A survey. *arXiv preprint arXiv:2509.02547*, 2025.

1121 Lei Zhang, Yuge Zhang, Kan Ren, Dongsheng Li, and Yuqing Yang. Mlcopilot: Unleashing the power
 1122 of large language models in solving machine learning tasks. *arXiv preprint arXiv:2304.14979*,
 1123 2023a.

1124 Shujian Zhang, Chengyue Gong, Lemeng Wu, Xingchao Liu, and Mingyuan Zhou. Automl-gpt:
 1125 Automatic machine learning with gpt. *arXiv preprint arXiv:2305.02499*, 2023b.

1126 Rui Zheng, Shihan Dou, Songyang Gao, Yuan Hua, Wei Shen, Binghai Wang, Yan Liu, Senjie Jin,
 1127 Qin Liu, Yuhao Zhou, et al. Secrets of rlhf in large language models part i: Ppo. *arXiv preprint*
 1128 *arXiv:2307.04964*, 2023.

1134 Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyuan Luo, Zhangchi Feng, and
1135 Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Pro-*
1136 *ceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3:*
1137 *System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics.
1138 URL <http://arxiv.org/abs/2403.13372>.

1139

1140

1141

1142

1143

1144

1145

1146

1147

1148

1149

1150

1151

1152

1153

1154

1155

1156

1157

1158

1159

1160

1161

1162

1163

1164

1165

1166

1167

1168

1169

1170

1171

1172

1173

1174

1175

1176

1177

1178

1179

1180

1181

1182

1183

1184

1185

1186

1187

1188 A PROBLEM FORMULATION
11891190 **Reformulation from equation 1 to equation 3.** Starting from equation 1, suppose the initial state
1191 distribution is $d^{\pi_\theta}(s_0)$, the state transition probability is $p_\pi(s_{t+1}|s_t, a_t)$, then we have
1192

1193
$$\mathcal{P}_{\pi_\theta}(\tau) = d^{\pi_\theta}(s_0) \prod_{t=0}^{n-1} p(s_{t+1}|s_t, a_t) \pi_\theta(a_t|s_t). \quad (7)$$

1194

1195 Hence the reformulation is:
1196

1197
$$\begin{aligned} \mathcal{J}(\theta) &= \mathbb{E}_{\pi_\theta}[R(\tau)] \\ &= \sum_{\tau} \mathcal{P}_{\pi_\theta}(\tau) R(\tau) \\ &= \sum_{s_0, a_0, \dots, s_n} \left(d^{\pi_\theta}(s_0) \prod_{t=0}^{n-1} p(s_{t+1}|s_t, a_t) \pi_\theta(a_t|s_t) \right) \left(\sum_{t=0}^n R(s_t, a_t) \right) \\ &= \sum_{t=0}^{n-1} \sum_{s_0, a_0, \dots, s_n} \left(d^{\pi_\theta}(s_0) \prod_{k=0}^{n-1} p(s_{k+1}|s_k, a_k) \pi_\theta(a_k|s_k) \right) R(s_t, a_t) \\ &= \sum_{t=0}^{n-1} \sum_{s_t, a_t} \left(\sum_{s_0, a_0, \dots, s_{t-1}, a_{t-1}} d^{\pi_\theta}(s_0) \prod_{k=0}^{t-1} p(s_{k+1}|s_k, a_k) \pi_\theta(a_k|s_k) \right) \pi_\theta(a_t|s_t) R(s_t, a_t) \end{aligned} \quad (8)$$

1200
1201
1202
1203
1204
1205
1206
1207
1208

1209 However, we can define the state distribution $d^{\pi_\theta}(s_t)$ as the probability agent visits state s_t at time t .
1210 Then according to this definition, this probability can be written as:
1211

1212
$$d^{\pi_\theta}(s_t) = \sum_{s_0, a_0, \dots, s_{t-1}, a_{t-1}} d^{\pi_\theta}(s_0) \prod_{k=0}^{t-1} p(s_{k+1}|s_k, a_k) \pi_\theta(a_k|s_k). \quad (9)$$

1213
1214

1215 Then we have
1216

1217
$$\begin{aligned} \mathcal{J}(\theta) &= \sum_{t=0}^{n-1} \sum_{s_t, a_t} d^{\pi_\theta}(s_t) \pi_\theta(a_t|s_t) R(a_t, s_t) \\ &= \sum_{t=0}^{n-1} \sum_{s_t \in \mathcal{S}} d^{\pi_\theta}(s_t) \sum_{a_t \in \mathcal{A}} \pi_\theta(a_t|s_t) R(a_t, s_t) \\ &= \mathcal{J}_{\text{step}}(\theta) \end{aligned} \quad (10)$$

1218
1219
1220
1221
1222

1223 B MACHINE LEARNING TASKS AND DATA COLLECTION PROCESS
12241225 B.1 DATA COLLECTING PIPELINE FOR EXPLORATION-ENRICHED FINE-TUNING
12261227 We construct diverse action pools along three semantic axes—**Data**, **Model**, and **Learning**—to
1228 support structured exploration. For each axis, we prompt a frozen LLM (GPT-4o-mini) to generate a
1229 large set of candidate actions (e.g., “Add MixUp augmentation”, “Switch to AdamW optimizer”). To
1230 promote diversity, we embed all candidates using a sentence transformer and apply farthest-point
1231 sampling (FPS) to select a compact, representative subset. The resulting pools $\mathcal{P}_{\text{Data}}$, $\mathcal{P}_{\text{Model}}$, and
1232 $\mathcal{P}_{\text{Learning}}$ are fixed during training.1233 During data collection, we form exploration-enriched prompts by randomly selecting 1–3 axes,
1234 shuffling their order, and drawing one action from each corresponding pool. These actions are
1235 concatenated into an initial instruction for the expert agent, which then interacts with a fast-executable
1236 ML environment (e.g., small-scale tabular or vision tasks) to produce a full trajectory. The complete
1237 pipeline is summarized in Algorithm 1.1238 B.2 DETAILS OF MACHINE LEARNING TASKS
12391240 The machine learning tasks utilized in our paper are all from MLAgentBench or MLE-bench.
1241 Table 3 shows all 9 training tasks and 10 testing tasks. The 9 training tasks contain 4 tasks from

1242 **Algorithm 1** Exploration-Enriched Trajectory Generation

1243 **Require:** Semantic axes $\mathcal{X} = \{\text{Data, Model, Learning}\}$,

1244 Set of fast-executable tasks \mathcal{N} , each with base description p_n^{task}

1245 **Ensure:** Dataset of expert trajectories \mathcal{D}

1246 1: **#Phase 1: Build diverse action pools via FPS**

1247 2: **for** each axis $X \in \mathcal{X}$ **do**

1248 3: Generate M candidate actions \mathcal{C}_X using LLM prompting

1249 4: $\mathcal{P}_X \leftarrow \text{FARTHESTPOINTSAMPLING}(\mathcal{C}_X, K)$ {Select K diverse actions}

1250 5: **end for**

1251 6: **#Phase 2: Generate trajectories**

1252 7: $\mathcal{D} \leftarrow \emptyset$

1253 8: **for** each task $n \in \mathcal{N}$ **do**

1254 9: Sample $k \sim \text{Uniform}\{1, 2, 3\}$

1255 10: Sample k distinct axes $\{X_1, \dots, X_k\} \subset \mathcal{X}$

1256 11: Sample $a_i \sim \text{Uniform}(\mathcal{P}_{X_i})$ for $i = 1, \dots, k$

1257 12: Form prompt: $p_n \leftarrow p_n^{\text{task}}.\text{format}(a_1, \dots, a_k)$

1258 13: Run expert LLM (GPT-4o-mini) on task n with prompt p_n

1259 14: Record trajectory τ

1260 15: $\mathcal{D} \leftarrow \mathcal{D} \cup \{\tau\}$

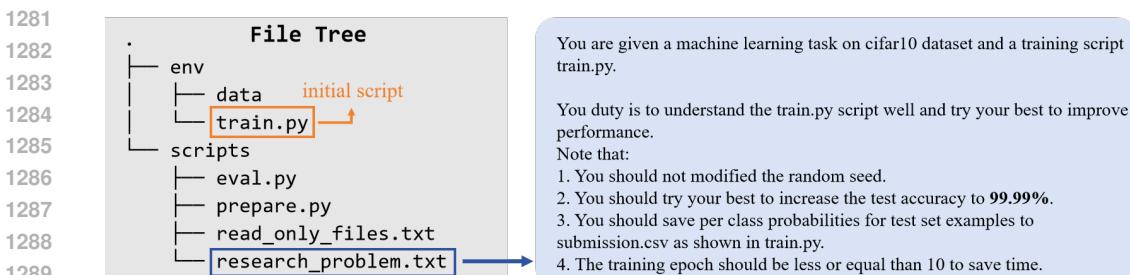
1261 16: **end for**

1262 17: **return** \mathcal{D}

1263
1264 MLAgentBench and 5 from MLE-bench (Chan et al., 2024); while the 10 testing tasks are all from
1265 MLE-bench.

1266 The selection strategy of training tasks aims to enhance data collection efficiency. Specifically,
1267 we select relatively simpler machine learning tasks (e.g. tasks labeled with low complexity in
1268 MLE-bench) for training. These training tasks typically involve smaller datasets, which enable
1269 faster iterations. For testing, we select relatively more complex tasks to evaluate the generalization
1270 capability. In addition, the training tasks and test tasks span three machine learning data types (image,
1271 text and tabular) and two general task categories (regression and classification).

1272 Specifically, Each task consists of the following components: (1) training, validation, and test
1273 data; (2) an initial bug-free script, "train.py", generated by GPT-4o-mini; (3) an evaluation script,
1274 "eval.py", which is used to calculate the test score from the submitted results; (4) a problem
1275 description file, "research_problem.txt"; and (5) a "prepare.py" script to download the data if
1276 necessary. An example file structure and related problem descriptions are shown in Figure 6. To
1277 ensure clarity regarding the task details and training objectives, we have refined some initial prompts
1278 from MLAGentBench by incorporating specific targets, such as "try your best to increase the test
1279 accuracy to 99.99%" (see in the right box in Figure 6). The format for the initial prompt, including
1280 the tool and format prompts, follows actions defined by MLAGentBench (see Table 5).



1291 Figure 6: Task file structure and prompt about the machine learning problem of cifar-10 task, for
1292 instance.

1293
1294
1295

1296
 1297
 1298
 1299
 1300 Table 3: All training and testing tasks used in our experiments. MLA and MLE stand for MLAGent-
 1301 Bench (Huang et al., 2023) and MLE-bench (Chan et al., 2024) respectively.

Task Name	Data Type	Task Type	Metric	Source
Training				
cifar-10	Image	Classification	Acc. (%) \uparrow	MLA
aerial-cactus-identification	Image	Classification	AUC \uparrow	MLE
dogs-vs-cats-redux-kernels-edition	Image	Classification	Logloss \downarrow	MLE
plant-pathology-2020-fgvc7	Image	Classification	AUC \uparrow	MLE
home-data-for-ml-course	Tabular	Regression	MAE \downarrow	MLA
spaceship-titanic	Tabular	Regression	Acc. (%) \uparrow	MLA
nomad2018-predict-transparent-conductors	Tabular	Regression	RMSLE \downarrow	MLE
feedback-prize-english-language-learning	Text	Classification	MCRMSE \downarrow	MLA
ogbn-arxiv (Maas et al., 2011)	Graph	Classification	Acc. (%) \uparrow	MLA
Testing				
denoising-dirty-documents	Image	Generation	RMSE \downarrow	MLE
leaf-classification	Image	Classification	Logloss \downarrow	MLE
statoil-iceberg-classifier-challenge	Image	Classification	Logloss \downarrow	MLE
whale-categorization-playground	Image	Classification	MAP@5 \uparrow	MLE
learning-agency-lab-automated-essay-scoring-2	Text	Regression	QWK \uparrow	MLE
detecting-insults-in-social-commentary	Text	Classification	Acc. (%) \uparrow	MLE
spooky-author-identification	Text	Classification	Logloss \downarrow	MLE
jigsaw-toxic-comment-classification-challenge	Text	Classification	AUC \uparrow	MLE
us-patent-phrase-to-phrase-matching	Tabular	Regression	PCC \uparrow	MLE
tabular-playground-series-dec-2021	Tabular	Regression	Acc. (%) \uparrow	MLE

1325
 1326
 1327
 1328
 1329
 1330
 1331 Table 4: Actions in MLAGentBench (Huang et al., 2023), where each action has a name, input and
 1332 output. Most of the actions are primitive actions that include file system operations and python script
 1333 execution. The last two are compound actions that is composed of multiple primitive actions and LM
 1334 calls.

Action Name	Input	Observation
List Files	directory (e.g. ..)	list of files in the directory
Copy File	Source (e.g. train.py), destination (e.g. train_copy.py)	A success or error message
Inspect Script Lines	file name, start line number, end line number	the file content between start and end line numbers
Execute Script	file name (e.g. train.py)	Any output from the execution
Final Answer	None	None
Understand File	file name, a query (e.g. the model architecture)	retrieved content from the file relevant to the query
Edit Script	file name, edit instruction (e.g. change epoch to 20), save file name	The diff of the edited file based on the instruction

1350
 1351
 1352
 1353
 1354
 1355
 1356
 1357
 1358
 1359
 1360
 1361
 1362
 1363
 1364
 1365
 1366
 1367
 1368
 1369
 1370
 1371

Table 5: Initial prompt template for agents on autonomous machine learning.

1372
 1373
 1374
 1375
 1376
 1377
 1378
 1379
 1380
 1381
 1382
 1383
 1384
 1385
 1386
 1387
 1388
 1389
 1390
 1391
 1392
 1393
 1394
 1395
 1396
 1397
 1398
 1399
 1400
 1401
 1402
 1403

You are a helpful research assistant. You have access to the following tools:
{tools_prompt}
 Research Problem: **{research_problem}**
 Always respond in this format exactly:
{format_prompt}
 Observation:
 ""
 the result of the action
 ""


```
1458         "script_name": [a valid python script name with relative path to current
1459                         directory if needed],
1460         "start_line_number": [a valid line number],
1461         "end_line_number": [a valid line number]
1462     }
1463     Observation: [The observation will be the content of the script between
1464                   start_line_number and end_line_number . If the script does not exist, the
1465                   observation will be an error message.]
1466     ...
1467
1468 - Edit Script (AI):
1469     Use this to do a relatively large but cohesive edit over a python script. Instead of
1470         editing the script directly, you should describe the edit instruction so that
1471         another AI can help you do this.
1472     Usage:
1473     ...
1474     Action: Edit Script (AI)
1475     Action Input: {
1476         "script_name": [a valid python script name with relative path to current
1477                         directory if needed. An empty script will be created if it does not exist
1478                         .],
1479         "edit_instruction": [a detailed step by step description on how to edit it.],
1480         "save_name": [a valid file name with relative path to current directory if
1481                         needed]
1482     }
1483     Observation: [The observation will be the edited content of the script. If the
1484                   script does not exist, the observation will be an error message. You should
1485                   always double check whether the edit is correct.]
1486     ...
1487
1488
```

Table 6: Response format requirement (`{format_prompt}`) in the initial prompt.

1483 Reflection: What does the observation mean? If there is an error, what caused the error and
1484 how to debug?
1485 Research Plan and Status: The full high-level research plan, with current status and confirmed
1486 results of each step briefly annotated. It must only include progress that has been made
1487 by previous steps. If there is any update, enclose the new update text in double asterisks
1488 ****like this****. If there is no update, just copy the previous step Research Plan and Status.
1489 The high-level plan from the previous step should be fully retained, unless it is intentionally
1490 revised.
1491 Fact Check: List all objective statements in the updates to Research Plan and Status one by
1492 one and point out whether it is guessed versus directly confirmed by the previous observation
1493 directly above. Performance numbers can only be confirmed by running the code and
1494 observing the output.
1495 Thought: What you are currently doing, what actions to perform and why
1496 Action: The action to take, should be one of the names of the tools
Action Input: The input to the action as a valid JSON string

B.3 DETAILS OF DATA COLLECTION

1502 In this paper, we use the MLAGentBench (Huang et al., 2023) environment to collect training
1503 trajectories across 9 machine learning tasks. The environment needs an LLM-based agent to take
1504 actions and send feedback to the agent. This will iterate for certain steps. We employ GPT-4o-
1505 mini (OpenAI, 2024) as the LLM-based agent to generate thinking and action following Table B.2.
1506 This agent interacts with the environment, while Qwen2.5-Coder-32B-Instruct (Yang et al., 2024)
1507 powers the coder agent, which is responsible for writing code and understanding files within the
1508 environment.

1509 Each trajectory comprises a multi-turn conversation between the agent and the environment. For each
 1510 trajectory, we set the maximum number of steps as 15 and the time limit as 30 minutes to control
 1511 the length and duration of interactions. Finally, we generated 10k trajectories on 9 tasks. These
 trajectories are utilized both in SFT training and PPO training.

1512 Since each task in the MLAgentBench environment requires an initial script, tasks sourced from
 1513 MLE-bench do not have a natural initial script. To address this, we generate simple, bug-free initial
 1514 scripts for those tasks using GPT-4o-mini to meet the environment's requirements.

1515 To diversify the trajectories we collect for SFT training, we curate an initial idea pool of at least
 1516 100 diverse ideas which may potentially improve the performance of our initial script. We calculate
 1517 the embedding distance of each idea in initial idea pool and filter out the top 10 initial ideas whose
 1518 average embedding distance is farthest to others. These ideas form a defined idea pool, which guides
 1519 the first step of each trajectory. For the generation of each trajectory, we randomly select 1 to 3 idea
 1520 combinations from this idea pool and prioritize their implementation in the initial step by including
 1521 the relevant instructions in the file `research_problem.txt` (see Figure 6). Table 7 show the prompt
 1522 we use and Table 8 shows an example of defined idea pool for the first step.
 1523

1524 Table 7: The prompt we use to generate the data-preprocessing idea pool.
 1525

1526 You are given a machine learning task and an initial script on the task.
 1527

1528 The machine learning task description is:
 1529 {task_description}
 1530

1531 The initial script is:
 1532 {initial_script}
 1533

1534 You should give {number_to_generate} advice that may potentially improve the
 1535 metric performance(e.g. accuracy) of the script on this machine learning task. Your advice
 1536 can only be related to data preprocessing.

1537 The advice in your answer should strictly follow the following format(one advice should be
 1538 in a line), note that [advice] flag should be mentioned only once in your answer:
 1539 [advice]

1540 YOUR ADVICE HERE
 1541 ...
 1542

1543 Table 8: An example of the first step action space(after filtering) when collecting training trajectories.
 1544

1545 Tune the momentum parameter in the optimizer for better convergence.
 1546 Use early stopping to terminate training when the test accuracy starts decreasing.
 1547 Experiment with focal loss to deal with imbalanced data if classes are not evenly distributed.
 1548 Regularize model weights with L1 or L2 regularization.
 1549 Implement feature visualization to understand what features are being learned.
 1550 Use a higher resolution for input images, if feasible, to capture more details.
 1551 Increase the complexity of the neural network by adding more convolutional layers.
 1552 Explore semi-supervised learning methods to leverage unlabeled data for training improve-
 1553 ments.
 1554 Normalize the data further by scaling the input images to a range of [0, 1].
 1555 Experiment with different batch sizes to see if a smaller or larger batch size affects
 1556 performance.
 1557

1560 C EXPERIMENTAL DETAILS

1561 C.1 DETAILS OF EXPERIMENTAL SET-UP

1562 **Training details.** We implement our supervised fine-tuning (SFT) and proximal policy optimization
 1563 (PPO) training using 8 A100s. For the SFT, the code base is LLama-Factory (Zheng et al., 2024),

1566 where we fully fine-tune the qwen2.5-7b model for 2 epochs with batch size 64 and learning rate
 1567 $2e - 5$. For the PPO, the code base is VeRL (Sheng et al., 2024). The PPO training setup involves the
 1568 following hyperparameters and configurations: the training batch size is set to 256, and the number of
 1569 epochs is 1. Additionally, the learning rate of actor and critic is set as $1e - 6$ and $1e - 5$, respectively,
 1570 and the coefficient of KL is 0.001.

1571 **Baseline details.** We show the specific versions of baselines in Table 9.
 1572

1573

1574

Table 9: Model Version and Identifier Mapping

Model Name	Version
GPT-4o-mini	GPT-4o-mini-2024-07-18
GPT-4o	GPT-4o-2024-08-06
Qwen-7B-Base	Qwen2.5-7B
Qwen-7B-Instruct	Qwen2.5-7B-Instruct
Qwen-32B-Instruct	Qwen2.5-32B-Instruct

1581

1582

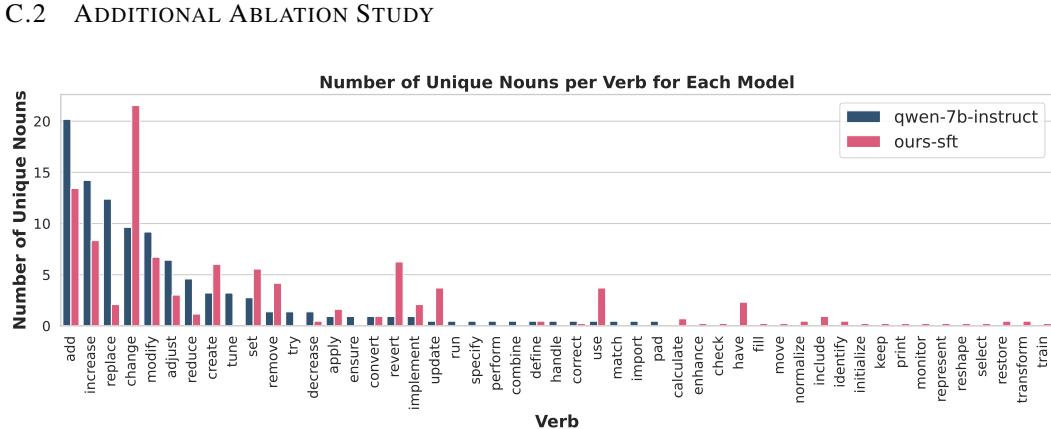
1583

C.2 ADDITIONAL ABLATION STUDY

1584

1585

1586



1599

1600

1601

Figure 7: Unique noun counts per verb across 100 randomly sampled edit instructions, comparing the Qwen2.5-7B-Instruct model (blue) with the ML-Agent-SFT model (red).

1602

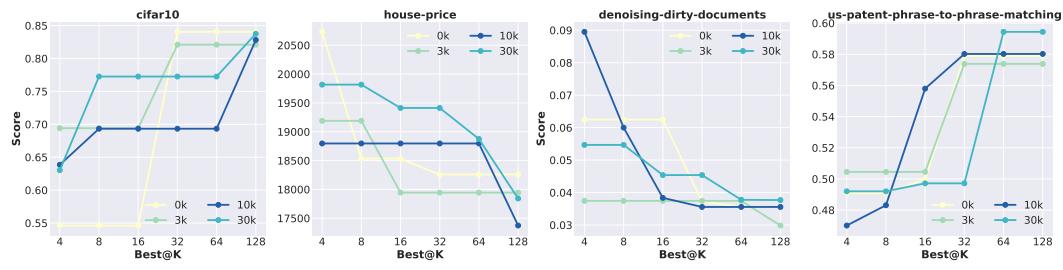
1603

1604

Diversity. Figure 7 compares the number of unique nouns associated with each editing verb in two models: Qwen2.5-7B-Instruct and ours-sft (ML-Agent-SFT). To generate these counts, we randomly sampled 100 edit_instruction sentences from the recorded expert trajectories. Then, we utilize an open-source NLP toolkit SpaCy to obtain the verb and noun for each edit_instruction sentence. Results show that after supervised fine-tuning with expert’s trajectories, the model can output a broader variety of actions, evidenced by the higher counts of unique nouns per verb.

1608

1609



1618

1619

Figure 8: Different number of training samples in RL, starting from our sft model.

1620

Effects of training data size in RL.

1621

1622 Here, we investigate how varying the number of training data samples (0k, 3k, 10k, 30k) affects the
 1623 performance in RL. The 0k scenario represents ML-Agent-SFT model. For every model, we run 128
 1624 trajectories for each task and evaluate the **best@K**, where K ranges over [4, 8, 16, 32, 64, 128], as
 1625 shown in Figure 8. In particular, for the two held-out tasks (second row), training with ppo (3k, 6k,
 1626 and 9k) improves performance faster than 0k as the number of sampled trajectories increases.

1627

1628 **Is thought helpful?** In Table 10, we compare the performance of models with and without the
 1629 requirement of thought before taking action across 13 tasks. The thought process includes several key
 1630 components such as "Reflection," "Research Plan and Status," "Fact Check," "Thought," "Action,"
 1631 and "Action Input.". In contrast, the model without thought only requires "Action" and "Action
 1632 Input." Note that the supervised fine-tuning data is also consistent with the key requirement. The
 1633 models with thought generally exhibit higher improved performance on both held-in and held-out
 1634 tasks. For instance, on the held-in cifar-10, the model with thought reaches 33.80% performance gain
 1635 compared to 13.03% when thought is not required. This trend continues across the held-out tasks,
 1636 where the model with thought shows higher accuracy and lower loss, demonstrating the importance
 1637 of incorporating a thoughtful reflection and planning phase for Autonomous machine learning via
 1638 RL.

1639

1640 Table 10: Performance comparison of reinforcement learning models with and without the require-
 1641 ment of thought prior to action. Average performance gains (%) are shown for both held-in and
 1642 held-out tasks, highlighting improvements in various tasks when thought is incorporated.

Thought?	Held-in tasks			Held-out tasks								
	cifar-10	house.	feedback	denoising.	leaf.	statoil.	learning.	detecting.	spooky.	jigsaw.	us.	tabular.
✗	13.03	5.68	9.88	28.66	2.50	-0.03	1.27	0.64	-3.40	0.00	7.15	-0.02
✓	33.80	6.77	13.47	52.38	13.87	1.41	1.91	1.74	1.76	0.01	12.96	0.20

1643

C.3 CASE STUDY

1644

1645 In this section, we will present more detailed case study on trajectories generated by ML-Agent(ppo)
 1646 on some test tasks such as denoising-dirty-documents. In Appendix C.3.1, we show the task
 1647 description for denoising-dirty-documents. In Appendix C.3.2, we show the initial script for
 1648 denoising-dirty-documents. In Appendix C.3.3, we show partial trajectory generated by ML-
 1649 Agent. We give an analysis in Appendix C.3.4.

1650

C.3.1 TASK DESCRIPTION FOR denoising-dirty-documents**Task description for denoising-dirty-documents**

1651 You are given a machine learning task on "denoising-dirty-documents" dataset. The dataset descriptions are given below:

1652 # Description

1653 [Optical Character Recognition](http://en.wikipedia.org/wiki/Optical_character_recognition) (OCR) is the process of getting type or
 1654 handwritten documents into a digitized format. If you've read a classic novel on a digital reading device or had your doctor
 1655 pull up old healthcare records via the hospital computer system, you've probably benefited from OCR.

1656 OCR makes previously static content editable, searchable, and much easier to share. But, a lot of documents eager for digitization
 1657 are being held back. Coffee stains, faded sun spots, dog-eared pages, and lot of wrinkles are keeping some printed documents
 1658 offline and in the past.

1659 This competition challenges you to give these documents a machine learning makeover. Given a dataset of images of scanned text that
 1660 has seen better days, you're challenged to remove the noise. Improving the ease of document enhancement will help us get that
 1661 rare mathematics book on our e-reader before the next beach vacation.

1662 We've kicked off the fun with a few [handy scripts to get you started on the dataset](<https://www.kaggle.com/c/denoising-dirty-documents/scripts>).

1663 # Evaluation

1664 Submissions are evaluated on the [root mean squared error](<https://www.kaggle.com/wiki/RootMeanSquaredError>) between the cleaned
 1665 pixel intensities and the actual grayscale pixel intensities.

1666 # Submission File

1667 Form the submission file by melting each images into a set of pixels, assigning each pixel an id of image_row_col (e.g. 1_2_1 is
 1668 image 1, row 2, column 1). Intensity values range from 0 (black) to 1 (white). The file should contain a header and have the
 1669 following format:

```

1674
1675     """
1676     id,value1_1_1,
1677     1_2_1,1
1678     1_3_1,1
1679     etc.
1680     """
1681
1682     ## Dataset Description
1683
1684     You are provided two sets of images, train and test. These images contain various styles of text, to which synthetic noise has been
1685     added to simulate real-world, messy artifacts. The training set includes the test without the noise (train_cleaned). You must
1686     create an algorithm to clean the images in the test set.

```

1683 C.3.2 INITIAL SCRIPT FOR denoising-dirty-documents

1684 Initial script for denoising-dirty-documents

```

1685
1686     import os
1687     import numpy as np
1688     import pandas as pd
1689     from PIL import Image
1690     import glob
1691     from sklearn.model_selection import train_test_split
1692     import torch
1693     from torch.utils.data import Dataset, DataLoader
1694     from torchvision import transforms
1695     import torch.nn as nn
1696     import torch.optim as optim
1697     import torch.nn.functional as F
1698     import math
1699
1700     # Set device
1701     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
1702
1703     # Define dataset
1704     class DenoisingDataset(Dataset):
1705         def __init__(self, noisy_images, clean_images=None, transform=None):
1706             self.noisy_images = noisy_images
1707             self.clean_images = clean_images
1708             self.transform = transform
1709
1710         def __len__(self):
1711             return len(self.noisy_images)
1712
1713         def __getitem__(self, idx):
1714             noisy_image = Image.open(self.noisy_images[idx]).convert("L")
1715             if self.transform:
1716                 noisy_image = self.transform(noisy_image)
1717             if self.clean_images is not None:
1718                 clean_image = Image.open(self.clean_images[idx]).convert("L")
1719                 if self.transform:
1720                     clean_image = self.transform(clean_image)
1721                 return noisy_image, clean_image
1722             else:
1723                 return noisy_image
1724
1725         # Custom collate function to pad images to the same size
1726         def collate_fn(batch):
1727             if len(batch[0]) == 2:
1728                 imgs, targets = zip(*batch)
1729             else:
1730                 imgs = batch
1731                 targets = None
1732
1733             # Compute necessary heights and widths after padding to next multiple of 8
1734             heights = []
1735             widths = []
1736             for img in imgs:
1737                 c, h, w = img.shape
1738                 new_h = ((h - 1) // 8 + 1) * 8
1739                 new_w = ((w - 1) // 8 + 1) * 8
1740                 heights.append(new_h)
1741                 widths.append(new_w)
1742
1743             max_h = max(heights)
1744             max_w = max(widths)
1745
1746             padded_imgs = []
1747             if targets is not None:
1748                 padded_targets = []
1749
1750             for i, img in enumerate(imgs):
1751                 c, h, w = img.shape
1752                 pad_h = max_h - h
1753                 pad_w = max_w - w
1754                 padding = (0, pad_w, 0, pad_h) # left, right, top, bottom
1755                 padded_img = F.pad(img, padding)
1756                 padded_imgs.append(padded_img)
1757                 if targets is not None:
1758                     target = targets[i]

```

```

1728
1729     padded_target = F.pad(target, padding)
1730     padded_targets.append(padded_target)
1731
1732     imgs_tensor = torch.stack(padded_imgs, dim=0)
1733     if targets is not None:
1734         targets_tensor = torch.stack(padded_targets, dim=0)
1735         return imgs_tensor, targets_tensor
1736     else:
1737         return imgs_tensor
1738
1739
1740
1741     # Load image paths
1742     noisy_images = sorted(glob.glob("./train/*.png"))
1743     clean_images = sorted(glob.glob("./train_cleaned/*.png"))
1744
1745     # Split into train and validation sets
1746     train_noisy, val_noisy, train_clean, val_clean = train_test_split(
1747         noisy_images, clean_images, test_size=0.2, random_state=42
1748     )
1749
1750     # Define transforms
1751     transform = transforms.Compose(
1752         [
1753             transforms.ToTensor(),
1754         ]
1755     )
1756
1757     # Create datasets and dataloaders
1758     train_dataset = DenoisingDataset(train_noisy, train_clean, transform)
1759     val_dataset = DenoisingDataset(val_noisy, val_clean, transform)
1760
1761     train_loader = DataLoader(
1762         train_dataset, batch_size=4, shuffle=True, collate_fn=collate_fn
1763     )
1764     val_loader = DataLoader(val_dataset, batch_size=4, shuffle=False, collate_fn=collate_fn)
1765
1766
1767     # Define the model (modified U-Net with cropping)
1768     class UNet(nn.Module):
1769         def __init__(self):
1770             super(UNet, self).__init__()
1771             # Encoder
1772             self.enc1 = nn.Sequential(
1773                 nn.Conv2d(1, 64, 3, padding=1),
1774                 nn.ReLU(),
1775                 nn.Conv2d(64, 64, 3, padding=1),
1776                 nn.ReLU(),
1777             )
1778             self.pool1 = nn.MaxPool2d(2, 2)
1779
1780             self.enc2 = nn.Sequential(
1781                 nn.Conv2d(64, 128, 3, padding=1),
1782                 nn.ReLU(),
1783                 nn.Conv2d(128, 128, 3, padding=1),
1784                 nn.ReLU(),
1785             )
1786             self.pool2 = nn.MaxPool2d(2, 2)
1787
1788             self.enc3 = nn.Sequential(
1789                 nn.Conv2d(128, 256, 3, padding=1),
1790                 nn.ReLU(),
1791                 nn.Conv2d(256, 256, 3, padding=1),
1792                 nn.ReLU(),
1793             )
1794             self.pool3 = nn.MaxPool2d(2, 2)
1795
1796             # Bottleneck
1797             self.bottleneck = nn.Sequential(
1798                 nn.Conv2d(256, 512, 3, padding=1),
1799                 nn.ReLU(),
1800                 nn.Conv2d(512, 512, 3, padding=1),
1801                 nn.ReLU(),
1802             )
1803
1804             # Decoder
1805             self.up1 = nn.ConvTranspose2d(512, 256, 2, stride=2)
1806             self.dec1 = nn.Sequential(
1807                 nn.Conv2d(512, 256, 3, padding=1),
1808                 nn.ReLU(),
1809                 nn.Conv2d(256, 256, 3, padding=1),
1810                 nn.ReLU(),
1811             )
1812
1813             self.up2 = nn.ConvTranspose2d(256, 128, 2, stride=2)
1814             self.dec2 = nn.Sequential(
1815                 nn.Conv2d(256, 128, 3, padding=1),
1816                 nn.ReLU(),
1817                 nn.Conv2d(128, 128, 3, padding=1),
1818                 nn.ReLU(),
1819             )
1820
1821             self.up3 = nn.ConvTranspose2d(128, 64, 2, stride=2)
1822             self.dec3 = nn.Sequential(
1823                 nn.Conv2d(128, 64, 3, padding=1),
1824                 nn.ReLU(),
1825                 nn.Conv2d(64, 64, 3, padding=1),
1826                 nn.ReLU(),
1827             )
1828
1829             self.up4 = nn.ConvTranspose2d(64, 32, 2, stride=2)
1830             self.dec4 = nn.Sequential(
1831                 nn.Conv2d(64, 32, 3, padding=1),
1832                 nn.ReLU(),
1833                 nn.Conv2d(32, 32, 3, padding=1),
1834                 nn.ReLU(),
1835             )
1836
1837             self.up5 = nn.ConvTranspose2d(32, 16, 2, stride=2)
1838             self.dec5 = nn.Sequential(
1839                 nn.Conv2d(32, 16, 3, padding=1),
1840                 nn.ReLU(),
1841                 nn.Conv2d(16, 16, 3, padding=1),
1842                 nn.ReLU(),
1843             )
1844
1845             self.up6 = nn.ConvTranspose2d(16, 8, 2, stride=2)
1846             self.dec6 = nn.Sequential(
1847                 nn.Conv2d(16, 8, 3, padding=1),
1848                 nn.ReLU(),
1849                 nn.Conv2d(8, 8, 3, padding=1),
1850                 nn.ReLU(),
1851             )
1852
1853             self.up7 = nn.ConvTranspose2d(8, 4, 2, stride=2)
1854             self.dec7 = nn.Sequential(
1855                 nn.Conv2d(8, 4, 3, padding=1),
1856                 nn.ReLU(),
1857                 nn.Conv2d(4, 4, 3, padding=1),
1858                 nn.ReLU(),
1859             )
1860
1861             self.up8 = nn.ConvTranspose2d(4, 2, 2, stride=2)
1862             self.dec8 = nn.Sequential(
1863                 nn.Conv2d(4, 2, 3, padding=1),
1864                 nn.ReLU(),
1865                 nn.Conv2d(2, 2, 3, padding=1),
1866                 nn.ReLU(),
1867             )
1868
1869             self.up9 = nn.ConvTranspose2d(2, 1, 2, stride=2)
1870             self.dec9 = nn.Sequential(
1871                 nn.Conv2d(2, 1, 3, padding=1),
1872                 nn.ReLU(),
1873                 nn.Conv2d(1, 1, 3, padding=1),
1874                 nn.ReLU(),
1875             )
1876
1877             self.up10 = nn.ConvTranspose2d(1, 1, 2, stride=2)
1878             self.dec10 = nn.Sequential(
1879                 nn.Conv2d(1, 1, 3, padding=1),
1880                 nn.ReLU(),
1881                 nn.Conv2d(1, 1, 3, padding=1),
1882                 nn.ReLU(),
1883             )
1884
1885             self.up11 = nn.ConvTranspose2d(1, 1, 2, stride=2)
1886             self.dec11 = nn.Sequential(
1887                 nn.Conv2d(1, 1, 3, padding=1),
1888                 nn.ReLU(),
1889                 nn.Conv2d(1, 1, 3, padding=1),
1890                 nn.ReLU(),
1891             )
1892
1893             self.up12 = nn.ConvTranspose2d(1, 1, 2, stride=2)
1894             self.dec12 = nn.Sequential(
1895                 nn.Conv2d(1, 1, 3, padding=1),
1896                 nn.ReLU(),
1897                 nn.Conv2d(1, 1, 3, padding=1),
1898                 nn.ReLU(),
1899             )
1900
1901             self.up13 = nn.ConvTranspose2d(1, 1, 2, stride=2)
1902             self.dec13 = nn.Sequential(
1903                 nn.Conv2d(1, 1, 3, padding=1),
1904                 nn.ReLU(),
1905                 nn.Conv2d(1, 1, 3, padding=1),
1906                 nn.ReLU(),
1907             )
1908
1909             self.up14 = nn.ConvTranspose2d(1, 1, 2, stride=2)
1910             self.dec14 = nn.Sequential(
1911                 nn.Conv2d(1, 1, 3, padding=1),
1912                 nn.ReLU(),
1913                 nn.Conv2d(1, 1, 3, padding=1),
1914                 nn.ReLU(),
1915             )
1916
1917             self.up15 = nn.ConvTranspose2d(1, 1, 2, stride=2)
1918             self.dec15 = nn.Sequential(
1919                 nn.Conv2d(1, 1, 3, padding=1),
1920                 nn.ReLU(),
1921                 nn.Conv2d(1, 1, 3, padding=1),
1922                 nn.ReLU(),
1923             )
1924
1925             self.up16 = nn.ConvTranspose2d(1, 1, 2, stride=2)
1926             self.dec16 = nn.Sequential(
1927                 nn.Conv2d(1, 1, 3, padding=1),
1928                 nn.ReLU(),
1929                 nn.Conv2d(1, 1, 3, padding=1),
1930                 nn.ReLU(),
1931             )
1932
1933             self.up17 = nn.ConvTranspose2d(1, 1, 2, stride=2)
1934             self.dec17 = nn.Sequential(
1935                 nn.Conv2d(1, 1, 3, padding=1),
1936                 nn.ReLU(),
1937                 nn.Conv2d(1, 1, 3, padding=1),
1938                 nn.ReLU(),
1939             )
1940
1941             self.up18 = nn.ConvTranspose2d(1, 1, 2, stride=2)
1942             self.dec18 = nn.Sequential(
1943                 nn.Conv2d(1, 1, 3, padding=1),
1944                 nn.ReLU(),
1945                 nn.Conv2d(1, 1, 3, padding=1),
1946                 nn.ReLU(),
1947             )
1948
1949             self.up19 = nn.ConvTranspose2d(1, 1, 2, stride=2)
1950             self.dec19 = nn.Sequential(
1951                 nn.Conv2d(1, 1, 3, padding=1),
1952                 nn.ReLU(),
1953                 nn.Conv2d(1, 1, 3, padding=1),
1954                 nn.ReLU(),
1955             )
1956
1957             self.up20 = nn.ConvTranspose2d(1, 1, 2, stride=2)
1958             self.dec20 = nn.Sequential(
1959                 nn.Conv2d(1, 1, 3, padding=1),
1960                 nn.ReLU(),
1961                 nn.Conv2d(1, 1, 3, padding=1),
1962                 nn.ReLU(),
1963             )
1964
1965             self.up21 = nn.ConvTranspose2d(1, 1, 2, stride=2)
1966             self.dec21 = nn.Sequential(
1967                 nn.Conv2d(1, 1, 3, padding=1),
1968                 nn.ReLU(),
1969                 nn.Conv2d(1, 1, 3, padding=1),
1970                 nn.ReLU(),
1971             )
1972
1973             self.up22 = nn.ConvTranspose2d(1, 1, 2, stride=2)
1974             self.dec22 = nn.Sequential(
1975                 nn.Conv2d(1, 1, 3, padding=1),
1976                 nn.ReLU(),
1977                 nn.Conv2d(1, 1, 3, padding=1),
1978                 nn.ReLU(),
1979             )
1980
1981             self.up23 = nn.ConvTranspose2d(1, 1, 2, stride=2)
1982             self.dec23 = nn.Sequential(
1983                 nn.Conv2d(1, 1, 3, padding=1),
1984                 nn.ReLU(),
1985                 nn.Conv2d(1, 1, 3, padding=1),
1986                 nn.ReLU(),
1987             )
1988
1989             self.up24 = nn.ConvTranspose2d(1, 1, 2, stride=2)
1990             self.dec24 = nn.Sequential(
1991                 nn.Conv2d(1, 1, 3, padding=1),
1992                 nn.ReLU(),
1993                 nn.Conv2d(1, 1, 3, padding=1),
1994                 nn.ReLU(),
1995             )
1996
1997             self.up25 = nn.ConvTranspose2d(1, 1, 2, stride=2)
1998             self.dec25 = nn.Sequential(
1999                 nn.Conv2d(1, 1, 3, padding=1),
2000                 nn.ReLU(),
2001                 nn.Conv2d(1, 1, 3, padding=1),
2002                 nn.ReLU(),
2003             )
2004
2005             self.up26 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2006             self.dec26 = nn.Sequential(
2007                 nn.Conv2d(1, 1, 3, padding=1),
2008                 nn.ReLU(),
2009                 nn.Conv2d(1, 1, 3, padding=1),
2010                 nn.ReLU(),
2011             )
2012
2013             self.up27 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2014             self.dec27 = nn.Sequential(
2015                 nn.Conv2d(1, 1, 3, padding=1),
2016                 nn.ReLU(),
2017                 nn.Conv2d(1, 1, 3, padding=1),
2018                 nn.ReLU(),
2019             )
2020
2021             self.up28 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2022             self.dec28 = nn.Sequential(
2023                 nn.Conv2d(1, 1, 3, padding=1),
2024                 nn.ReLU(),
2025                 nn.Conv2d(1, 1, 3, padding=1),
2026                 nn.ReLU(),
2027             )
2028
2029             self.up29 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2030             self.dec29 = nn.Sequential(
2031                 nn.Conv2d(1, 1, 3, padding=1),
2032                 nn.ReLU(),
2033                 nn.Conv2d(1, 1, 3, padding=1),
2034                 nn.ReLU(),
2035             )
2036
2037             self.up30 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2038             self.dec30 = nn.Sequential(
2039                 nn.Conv2d(1, 1, 3, padding=1),
2040                 nn.ReLU(),
2041                 nn.Conv2d(1, 1, 3, padding=1),
2042                 nn.ReLU(),
2043             )
2044
2045             self.up31 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2046             self.dec31 = nn.Sequential(
2047                 nn.Conv2d(1, 1, 3, padding=1),
2048                 nn.ReLU(),
2049                 nn.Conv2d(1, 1, 3, padding=1),
2050                 nn.ReLU(),
2051             )
2052
2053             self.up32 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2054             self.dec32 = nn.Sequential(
2055                 nn.Conv2d(1, 1, 3, padding=1),
2056                 nn.ReLU(),
2057                 nn.Conv2d(1, 1, 3, padding=1),
2058                 nn.ReLU(),
2059             )
2060
2061             self.up33 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2062             self.dec33 = nn.Sequential(
2063                 nn.Conv2d(1, 1, 3, padding=1),
2064                 nn.ReLU(),
2065                 nn.Conv2d(1, 1, 3, padding=1),
2066                 nn.ReLU(),
2067             )
2068
2069             self.up34 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2070             self.dec34 = nn.Sequential(
2071                 nn.Conv2d(1, 1, 3, padding=1),
2072                 nn.ReLU(),
2073                 nn.Conv2d(1, 1, 3, padding=1),
2074                 nn.ReLU(),
2075             )
2076
2077             self.up35 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2078             self.dec35 = nn.Sequential(
2079                 nn.Conv2d(1, 1, 3, padding=1),
2080                 nn.ReLU(),
2081                 nn.Conv2d(1, 1, 3, padding=1),
2082                 nn.ReLU(),
2083             )
2084
2085             self.up36 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2086             self.dec36 = nn.Sequential(
2087                 nn.Conv2d(1, 1, 3, padding=1),
2088                 nn.ReLU(),
2089                 nn.Conv2d(1, 1, 3, padding=1),
2090                 nn.ReLU(),
2091             )
2092
2093             self.up37 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2094             self.dec37 = nn.Sequential(
2095                 nn.Conv2d(1, 1, 3, padding=1),
2096                 nn.ReLU(),
2097                 nn.Conv2d(1, 1, 3, padding=1),
2098                 nn.ReLU(),
2099             )
2100
2101             self.up38 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2102             self.dec38 = nn.Sequential(
2103                 nn.Conv2d(1, 1, 3, padding=1),
2104                 nn.ReLU(),
2105                 nn.Conv2d(1, 1, 3, padding=1),
2106                 nn.ReLU(),
2107             )
2108
2109             self.up39 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2110             self.dec39 = nn.Sequential(
2111                 nn.Conv2d(1, 1, 3, padding=1),
2112                 nn.ReLU(),
2113                 nn.Conv2d(1, 1, 3, padding=1),
2114                 nn.ReLU(),
2115             )
2116
2117             self.up40 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2118             self.dec40 = nn.Sequential(
2119                 nn.Conv2d(1, 1, 3, padding=1),
2120                 nn.ReLU(),
2121                 nn.Conv2d(1, 1, 3, padding=1),
2122                 nn.ReLU(),
2123             )
2124
2125             self.up41 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2126             self.dec41 = nn.Sequential(
2127                 nn.Conv2d(1, 1, 3, padding=1),
2128                 nn.ReLU(),
2129                 nn.Conv2d(1, 1, 3, padding=1),
2130                 nn.ReLU(),
2131             )
2132
2133             self.up42 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2134             self.dec42 = nn.Sequential(
2135                 nn.Conv2d(1, 1, 3, padding=1),
2136                 nn.ReLU(),
2137                 nn.Conv2d(1, 1, 3, padding=1),
2138                 nn.ReLU(),
2139             )
2140
2141             self.up43 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2142             self.dec43 = nn.Sequential(
2143                 nn.Conv2d(1, 1, 3, padding=1),
2144                 nn.ReLU(),
2145                 nn.Conv2d(1, 1, 3, padding=1),
2146                 nn.ReLU(),
2147             )
2148
2149             self.up44 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2150             self.dec44 = nn.Sequential(
2151                 nn.Conv2d(1, 1, 3, padding=1),
2152                 nn.ReLU(),
2153                 nn.Conv2d(1, 1, 3, padding=1),
2154                 nn.ReLU(),
2155             )
2156
2157             self.up45 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2158             self.dec45 = nn.Sequential(
2159                 nn.Conv2d(1, 1, 3, padding=1),
2160                 nn.ReLU(),
2161                 nn.Conv2d(1, 1, 3, padding=1),
2162                 nn.ReLU(),
2163             )
2164
2165             self.up46 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2166             self.dec46 = nn.Sequential(
2167                 nn.Conv2d(1, 1, 3, padding=1),
2168                 nn.ReLU(),
2169                 nn.Conv2d(1, 1, 3, padding=1),
2170                 nn.ReLU(),
2171             )
2172
2173             self.up47 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2174             self.dec47 = nn.Sequential(
2175                 nn.Conv2d(1, 1, 3, padding=1),
2176                 nn.ReLU(),
2177                 nn.Conv2d(1, 1, 3, padding=1),
2178                 nn.ReLU(),
2179             )
2180
2181             self.up48 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2182             self.dec48 = nn.Sequential(
2183                 nn.Conv2d(1, 1, 3, padding=1),
2184                 nn.ReLU(),
2185                 nn.Conv2d(1, 1, 3, padding=1),
2186                 nn.ReLU(),
2187             )
2188
2189             self.up49 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2190             self.dec49 = nn.Sequential(
2191                 nn.Conv2d(1, 1, 3, padding=1),
2192                 nn.ReLU(),
2193                 nn.Conv2d(1, 1, 3, padding=1),
2194                 nn.ReLU(),
2195             )
2196
2197             self.up50 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2198             self.dec50 = nn.Sequential(
2199                 nn.Conv2d(1, 1, 3, padding=1),
2200                 nn.ReLU(),
2201                 nn.Conv2d(1, 1, 3, padding=1),
2202                 nn.ReLU(),
2203             )
2204
2205             self.up51 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2206             self.dec51 = nn.Sequential(
2207                 nn.Conv2d(1, 1, 3, padding=1),
2208                 nn.ReLU(),
2209                 nn.Conv2d(1, 1, 3, padding=1),
2210                 nn.ReLU(),
2211             )
2212
2213             self.up52 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2214             self.dec52 = nn.Sequential(
2215                 nn.Conv2d(1, 1, 3, padding=1),
2216                 nn.ReLU(),
2217                 nn.Conv2d(1, 1, 3, padding=1),
2218                 nn.ReLU(),
2219             )
2220
2221             self.up53 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2222             self.dec53 = nn.Sequential(
2223                 nn.Conv2d(1, 1, 3, padding=1),
2224                 nn.ReLU(),
2225                 nn.Conv2d(1, 1, 3, padding=1),
2226                 nn.ReLU(),
2227             )
2228
2229             self.up54 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2230             self.dec54 = nn.Sequential(
2231                 nn.Conv2d(1, 1, 3, padding=1),
2232                 nn.ReLU(),
2233                 nn.Conv2d(1, 1, 3, padding=1),
2234                 nn.ReLU(),
2235             )
2236
2237             self.up55 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2238             self.dec55 = nn.Sequential(
2239                 nn.Conv2d(1, 1, 3, padding=1),
2240                 nn.ReLU(),
2241                 nn.Conv2d(1, 1, 3, padding=1),
2242                 nn.ReLU(),
2243             )
2244
2245             self.up56 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2246             self.dec56 = nn.Sequential(
2247                 nn.Conv2d(1, 1, 3, padding=1),
2248                 nn.ReLU(),
2249                 nn.Conv2d(1, 1, 3, padding=1),
2250                 nn.ReLU(),
2251             )
2252
2253             self.up57 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2254             self.dec57 = nn.Sequential(
2255                 nn.Conv2d(1, 1, 3, padding=1),
2256                 nn.ReLU(),
2257                 nn.Conv2d(1, 1, 3, padding=1),
2258                 nn.ReLU(),
2259             )
2260
2261             self.up58 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2262             self.dec58 = nn.Sequential(
2263                 nn.Conv2d(1, 1, 3, padding=1),
2264                 nn.ReLU(),
2265                 nn.Conv2d(1, 1, 3, padding=1),
2266                 nn.ReLU(),
2267             )
2268
2269             self.up59 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2270             self.dec59 = nn.Sequential(
2271                 nn.Conv2d(1, 1, 3, padding=1),
2272                 nn.ReLU(),
2273                 nn.Conv2d(1, 1, 3, padding=1),
2274                 nn.ReLU(),
2275             )
2276
2277             self.up60 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2278             self.dec60 = nn.Sequential(
2279                 nn.Conv2d(1, 1, 3, padding=1),
2280                 nn.ReLU(),
2281                 nn.Conv2d(1, 1, 3, padding=1),
2282                 nn.ReLU(),
2283             )
2284
2285             self.up61 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2286             self.dec61 = nn.Sequential(
2287                 nn.Conv2d(1, 1, 3, padding=1),
2288                 nn.ReLU(),
2289                 nn.Conv2d(1, 1, 3, padding=1),
2290                 nn.ReLU(),
2291             )
2292
2293             self.up62 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2294             self.dec62 = nn.Sequential(
2295                 nn.Conv2d(1, 1, 3, padding=1),
2296                 nn.ReLU(),
2297                 nn.Conv2d(1, 1, 3, padding=1),
2298                 nn.ReLU(),
2299             )
2300
2301             self.up63 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2302             self.dec63 = nn.Sequential(
2303                 nn.Conv2d(1, 1, 3, padding=1),
2304                 nn.ReLU(),
2305                 nn.Conv2d(1, 1, 3, padding=1),
2306                 nn.ReLU(),
2307             )
2308
2309             self.up64 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2310             self.dec64 = nn.Sequential(
2311                 nn.Conv2d(1, 1, 3, padding=1),
2312                 nn.ReLU(),
2313                 nn.Conv2d(1, 1, 3, padding=1),
2314                 nn.ReLU(),
2315             )
2316
2317             self.up65 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2318             self.dec65 = nn.Sequential(
2319                 nn.Conv2d(1, 1, 3, padding=1),
2320                 nn.ReLU(),
2321                 nn.Conv2d(1, 1, 3, padding=1),
2322                 nn.ReLU(),
2323             )
2324
2325             self.up66 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2326             self.dec66 = nn.Sequential(
2327                 nn.Conv2d(1, 1, 3, padding=1),
2328                 nn.ReLU(),
2329                 nn.Conv2d(1, 1, 3, padding=1),
2330                 nn.ReLU(),
2331             )
2332
2333             self.up67 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2334             self.dec67 = nn.Sequential(
2335                 nn.Conv2d(1, 1, 3, padding=1),
2336                 nn.ReLU(),
2337                 nn.Conv2d(1, 1, 3, padding=1),
2338                 nn.ReLU(),
2339             )
2340
2341             self.up68 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2342             self.dec68 = nn.Sequential(
2343                 nn.Conv2d(1, 1, 3, padding=1),
2344                 nn.ReLU(),
2345                 nn.Conv2d(1, 1, 3, padding=1),
2346                 nn.ReLU(),
2347             )
2348
2349             self.up69 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2350             self.dec69 = nn.Sequential(
2351                 nn.Conv2d(1, 1, 3, padding=1),
2352                 nn.ReLU(),
2353                 nn.Conv2d(1, 1, 3, padding=1),
2354                 nn.ReLU(),
2355             )
2356
2357             self.up70 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2358             self.dec70 = nn.Sequential(
2359                 nn.Conv2d(1, 1, 3, padding=1),
2360                 nn.ReLU(),
2361                 nn.Conv2d(1, 1, 3, padding=1),
2362                 nn.ReLU(),
2363             )
2364
2365             self.up71 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2366             self.dec71 = nn.Sequential(
2367                 nn.Conv2d(1, 1, 3, padding=1),
2368                 nn.ReLU(),
2369                 nn.Conv2d(1, 1, 3, padding=1),
2370                 nn.ReLU(),
2371             )
2372
2373             self.up72 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2374             self.dec72 = nn.Sequential(
2375                 nn.Conv2d(1, 1, 3, padding=1),
2376                 nn.ReLU(),
2377                 nn.Conv2d(1, 1, 3, padding=1),
2378                 nn.ReLU(),
2379             )
2380
2381             self.up73 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2382             self.dec73 = nn.Sequential(
2383                 nn.Conv2d(1, 1, 3, padding=1),
2384                 nn.ReLU(),
2385                 nn.Conv2d(1, 1, 3, padding=1),
2386                 nn.ReLU(),
2387             )
2388
2389             self.up74 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2390             self.dec74 = nn.Sequential(
2391                 nn.Conv2d(1, 1, 3, padding=1),
2392                 nn.ReLU(),
2393                 nn.Conv2d(1, 1, 3, padding=1),
2394                 nn.ReLU(),
2395             )
2396
2397             self.up75 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2398             self.dec75 = nn.Sequential(
2399                 nn.Conv2d(1, 1, 3, padding=1),
2400                 nn.ReLU(),
2401                 nn.Conv2d(1, 1, 3, padding=1),
2402                 nn.ReLU(),
2403             )
2404
2405             self.up76 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2406             self.dec76 = nn.Sequential(
2407                 nn.Conv2d(1, 1, 3, padding=1),
2408                 nn.ReLU(),
2409                 nn.Conv2d(1, 1, 3, padding=1),
2410                 nn.ReLU(),
2411             )
2412
2413             self.up77 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2414             self.dec77 = nn.Sequential(
2415                 nn.Conv2d(1, 1, 3, padding=1),
2416                 nn.ReLU(),
2417                 nn.Conv2d(1, 1, 3, padding=1),
2418                 nn.ReLU(),
2419             )
2420
2421             self.up78 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2422             self.dec78 = nn.Sequential(
2423                 nn.Conv2d(1, 1, 3, padding=1),
2424                 nn.ReLU(),
2425                 nn.Conv2d(1, 1, 3, padding=1),
2426                 nn.ReLU(),
2427             )
2428
2429             self.up79 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2430             self.dec79 = nn.Sequential(
2431                 nn.Conv2d(1, 1, 3, padding=1),
2432                 nn.ReLU(),
2433                 nn.Conv2d(1, 1, 3, padding=1),
2434                 nn.ReLU(),
2435             )
2436
2437             self.up80 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2438             self.dec80 = nn.Sequential(
2439                 nn.Conv2d(1, 1, 3, padding=1),
2440                 nn.ReLU(),
2441                 nn.Conv2d(1, 1, 3, padding=1),
2442                 nn.ReLU(),
2443             )
2444
2445             self.up81 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2446             self.dec81 = nn.Sequential(
2447                 nn.Conv2d(1, 1, 3, padding=1),
2448                 nn.ReLU(),
2449                 nn.Conv2d(1, 1, 3, padding=1),
2450                 nn.ReLU(),
2451             )
2452
2453             self.up82 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2454             self.dec82 = nn.Sequential(
2455                 nn.Conv2d(1, 1, 3, padding=1),
2456                 nn.ReLU(),
2457                 nn.Conv2d(1, 1, 3, padding=1),
2458                 nn.ReLU(),
2459             )
2460
2461             self.up83 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2462             self.dec83 = nn.Sequential(
2463                 nn.Conv2d(1, 1, 3, padding=1),
2464                 nn.ReLU(),
2465                 nn.Conv2d(1, 1, 3, padding=1),
2466                 nn.ReLU(),
2467             )
2468
2469             self.up84 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2470             self.dec84 = nn.Sequential(
2471                 nn.Conv2d(1, 1, 3, padding=1),
2472                 nn.ReLU(),
2473                 nn.Conv2d(1, 1, 3, padding=1),
2474                 nn.ReLU(),
2475             )
2476
2477             self.up85 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2478             self.dec85 = nn.Sequential(
2479                 nn.Conv2d(1, 1, 3, padding=1),
2480                 nn.ReLU(),
2481                 nn.Conv2d(1, 1, 3, padding=1),
2482                 nn.ReLU(),
2483             )
2484
2485             self.up86 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2486             self.dec86 = nn.Sequential(
2487                 nn.Conv2d(1, 1, 3, padding=1),
2488                 nn.ReLU(),
2489                 nn.Conv2d(1, 1, 3, padding=1),
2490                 nn.ReLU(),
2491             )
2492
2493             self.up87 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2494             self.dec87 = nn.Sequential(
2495                 nn.Conv2d(1, 1, 3, padding=1),
2496                 nn.ReLU(),
2497                 nn.Conv2d(1, 1, 3, padding=1),
2498                 nn.ReLU(),
2499             )
2500
2501             self.up88 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2502             self.dec88 = nn.Sequential(
2503                 nn.Conv2d(1, 1, 3, padding=1),
2504                 nn.ReLU(),
2505                 nn.Conv2d(1, 1, 3, padding=1),
2506                 nn.ReLU(),
2507             )
2508
2509             self.up89 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2510             self.dec89 = nn.Sequential(
2511                 nn.Conv2d(1, 1, 3, padding=1),
2512                 nn.ReLU(),
2513                 nn.Conv2d(1, 1, 3, padding=1),
2514                 nn.ReLU(),
2515             )
2516
2517             self.up90 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2518             self.dec90 = nn.Sequential(
2519                 nn.Conv2d(1, 1, 3, padding=1),
2520                 nn.ReLU(),
2521                 nn.Conv2d(1, 1, 3, padding=1),
2522                 nn.ReLU(),
2523             )
2524
2525             self.up91 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2526             self.dec91 = nn.Sequential(
2527                 nn.Conv2d(1, 1, 3, padding=1),
2528                 nn.ReLU(),
2529                 nn.Conv2d(1, 1, 3, padding=1),
2530                 nn.ReLU(),
2531             )
2532
2533             self.up92 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2534             self.dec92 = nn.Sequential(
2535                 nn.Conv2d(1, 1, 3, padding=1),
2536                 nn.ReLU(),
2537                 nn.Conv2d(1, 1, 3, padding=1),
2538                 nn.ReLU(),
2539             )
2540
2541             self.up93 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2542             self.dec93 = nn.Sequential(
2543                 nn.Conv2d(1, 1, 3, padding=1),
2544                 nn.ReLU(),
2545                 nn.Conv2d(1, 1, 3, padding=1),
2546                 nn.ReLU(),
2547             )
2548
2549             self.up94 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2550             self.dec94 = nn.Sequential(
2551                 nn.Conv2d(1, 1, 3, padding=1),
2552                 nn.ReLU(),
2553                 nn.Conv2d(1, 1, 3, padding=1),
2554                 nn.ReLU(),
2555             )
2556
2557             self.up95 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2558             self.dec95 = nn.Sequential(
2559                 nn.Conv2d(1, 1, 3, padding=1),
2560                 nn.ReLU(),
2561                 nn.Conv2d(1, 1, 3, padding=1),
2562                 nn.ReLU(),
2563             )
2564
2565             self.up96 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2566             self.dec96 = nn.Sequential(
2567                 nn.Conv2d(1, 1, 3, padding=1),
2568                 nn.ReLU(),
2569                 nn.Conv2d(1, 1, 3, padding=1),
2570                 nn.ReLU(),
2571             )
2572
2573             self.up97 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2574             self.dec97 = nn.Sequential(
2575                 nn.Conv2d(1, 1, 3, padding=1),
2576                 nn.ReLU(),
2577                 nn.Conv2d(1, 1, 3, padding=1),
2578                 nn.ReLU(),
2579             )
2580
2581             self.up98 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2582             self.dec98 = nn.Sequential(
2583                 nn.Conv2d(1, 1, 3, padding=1),
2584                 nn.ReLU(),
2585                 nn.Conv2d(1, 1, 3, padding=1),
2586                 nn.ReLU(),
2587             )
2588
2589             self.up99 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2590             self.dec99 = nn.Sequential(
2591                 nn.Conv2d(1, 1, 3, padding=1),
2592                 nn.ReLU(),
2593                 nn.Conv2d(1, 1, 3, padding=1),
2594                 nn.ReLU(),
2595             )
2596
2597             self.up100 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2598             self.dec100 = nn.Sequential(
2599                 nn.Conv2d(1, 1, 3, padding=1),
2600                 nn.ReLU(),
2601                 nn.Conv2d(1, 1, 3, padding=1),
2602                 nn.ReLU(),
2603             )
2604
2605             self.up101 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2606             self.dec101 = nn.Sequential(
2607                 nn.Conv2d(1, 1, 3, padding=1),
2608                 nn.ReLU(),
2609                 nn.Conv2d(1, 1, 3, padding=1),
2610                 nn.ReLU(),
2611             )
2612
2613             self.up102 = nn.ConvTranspose2d(1, 1, 2, stride=2)
2614             self.dec102 = nn.Sequential(
2615                 nn.Conv2d(1, 1, 3, padding=1),
2616                 nn.ReLU(),
2617
```

```

1782
1783     self.dec1 = nn.Sequential(
1784         nn.Conv2d(128, 64, 3, padding=1),
1785         nn.ReLU(),
1786         nn.Conv2d(64, 64, 3, padding=1),
1787         nn.ReLU(),
1788     )
1789     self.conv_last = nn.Conv2d(64, 1, 1)
1790
1791     def center_crop(self, layer, target_h, target_w):
1792         _, _, h, w = layer.size()
1793         diff_y = h - target_h
1794         diff_x = w - target_w
1795         cropped = layer[
1796             :,
1797             :,
1798             diff_y // 2 : diff_y // 2 + target_h,
1799             diff_x // 2 : diff_x // 2 + target_w,
1800         ]
1801         return cropped
1802
1803     def forward(self, x):
1804         # Encoder
1805         enc1 = self.enc1(x)
1806         pool1 = self.pool1(enc1)
1807
1808         enc2 = self.enc2(pool1)
1809         pool2 = self.pool2(enc2)
1810
1811         enc3 = self.enc3(pool2)
1812         pool3 = self.pool3(enc3)
1813
1814         # Bottleneck
1815         bottleneck = self.bottleneck(pool3)
1816
1817         # Decoder
1818         up3 = self.up3(bottleneck)
1819         enc3_cropped = self.center_crop(enc3, up3.size(2), up3.size(3))
1820         cat3 = torch.cat([up3, enc3_cropped], dim=1)
1821         dec3 = self.dec3(cat3)
1822
1823         up2 = self.up2(dec3)
1824         enc2_cropped = self.center_crop(enc2, up2.size(2), up2.size(3))
1825         cat2 = torch.cat([up2, enc2_cropped], dim=1)
1826         dec2 = self.dec2(cat2)
1827
1828         up1 = self.up1(dec2)
1829         enc1_cropped = self.center_crop(enc1, up1.size(2), up1.size(3))
1830         cat1 = torch.cat([up1, enc1_cropped], dim=1)
1831         dec1 = self.dec1(cat1)
1832
1833         out = self.conv_last(dec1)
1834         out = torch.sigmoid(out)
1835         return out
1836
1837     # Instantiate model, loss function, optimizer
1838     model = UNet().to(device)
1839     criterion = nn.MSELoss()
1840     optimizer = optim.Adam(model.parameters(), lr=1e-4)
1841
1842     # Training loop
1843     num_epochs = 5
1844     for epoch in range(num_epochs):
1845         model.train()
1846         running_loss = 0.0
1847         for inputs, targets in train_loader:
1848             inputs = inputs.to(device)
1849             targets = targets.to(device)
1850
1851             optimizer.zero_grad()
1852             outputs = model(inputs)
1853             loss = criterion(outputs, targets)
1854             loss.backward()
1855             optimizer.step()
1856
1857             running_loss += loss.item() * inputs.size(0)
1858     epoch_loss = running_loss / len(train_loader.dataset)
1859     print(f"Epoch {epoch+1}/{num_epochs}, Training Loss: {epoch_loss:.6f}")
1860
1861     # Validation
1862     model.eval()
1863     val_loss = 0.0
1864     with torch.no_grad():
1865         for inputs, targets in val_loader:
1866             inputs = inputs.to(device)
1867             targets = targets.to(device)
1868
1869             outputs = model(inputs)
1870             loss = criterion(outputs, targets)
1871             val_loss += loss.item() * inputs.size(0)
1872     val_loss /= len(val_loader.dataset)

```

```

1836     print(f"Epoch {epoch+1}/{num_epochs}, Validation Loss: {val_loss:.6f}")
1837
1838 # Compute RMSE on validation set
1839 def compute_rmse(model, loader):
1840     model.eval()
1841     mse = 0.0
1842     num_pixels = 0
1843     with torch.no_grad():
1844         for inputs, targets in loader:
1845             inputs = inputs.to(device)
1846             targets = targets.to(device)
1847             outputs = model(inputs)
1848             mse += F.mse_loss(outputs, targets, reduction="sum").item()
1849             num_pixels += targets.numel()
1850     rmse = math.sqrt(mse / num_pixels)
1851     return rmse
1852
1853 rmse = compute_rmse(model, val_loader)
1854 print(f"Validation RMSE: {rmse}")
1855
1856 # Predict on test images
1857 model.eval()
1858 test_images = sorted(glob.glob("./test/*.png"))
1859 ids = []
1860 vals = []
1861 transform = transforms.Compose([transforms.ToTensor()])
1862 for img_path in test_images:
1863     img = Image.open(img_path).convert("L")
1864     img_id = os.path.basename(img_path).split(".")[0]
1865     input_tensor = transform(img).unsqueeze(0).to(device)
1866     # Pad the input tensor to next multiple of 8
1867     _, _, h, w = input_tensor.size()
1868     new_h = ((h - 1) // 8 + 1) * 8
1869     new_w = ((w - 1) // 8 + 1) * 8
1870     pad_h = new_h - h
1871     pad_w = new_w - w
1872     padding = (0, pad_w, 0, pad_h) # left, right, top, bottom
1873     input_tensor = F.pad(input_tensor, padding)
1874     with torch.no_grad():
1875         output = model(input_tensor)
1876         # Unpad the output tensor
1877         output = output[:, :, :h, :w]
1878         output = output.squeeze().cpu().numpy()
1879         # Ensure values are between 0 and 1
1880         output = np.clip(output, 0, 1)
1881         h, w = output.shape
1882         for i in range(h):
1883             for j in range(w):
1884                 ids.append(f"{img_id}_{i+1}_{j+1}")
1885                 vals.append(output[i, j])
1886
1887 # Create submission dataframe
1888 submission = pd.DataFrame({"id": ids, "value": vals})
1889
1890 # Ensure the submission matches the sample submission format
1891 sample_submission = pd.read_csv("./sampleSubmission.csv")
1892 submission = submission.merge(sample_submission[["id"]], on="id", how="right")
1893 submission = submission.fillna(0)
1894 submission = submission[["id", "value"]]
1895
1896 # Save submission
1897 submission.to_csv("./submission.csv", index=False)
1898 print("Submission file saved to ./submission.csv")
1899
1900 # Print validation RMSE
1901 print(f"Final Validation RMSE: {rmse}")

```

C.3.3 ML-AGENT TRAJECTORY ON denoising-dirty-documents

Partial trajectory of ML-Agent on denoising-dirty-documents

```

=====
Step 7 =====
Reflection: The modified training script executed successfully, and the validation RMSE has improved to approximately 0.0838. This is a slight improvement compared to the previous RMSE of 0.0884. However, it still does not meet the target of reducing the RMSE below 0.01. The training loss decreased consistently over the epochs, indicating that the model is learning effectively. Further modifications are necessary to achieve the desired performance.
Research Plan and Status: 1. Understand the 'train.py' script: Completed. The model architecture, training process, hyperparameters, and evaluation methods have been identified.
2. Identify potential modifications: Completed. The first modification (changing the learning rate) has been implemented and evaluated. The second modification (increasing the number of epochs) has been implemented and evaluated. The third modification (changing the optimizer) has been implemented and evaluated.
3. Implement modifications: The optimizer has been modified in 'train_modified_optimizer.py', resulting in a validation RMSE of approximately 0.0838.
4. Iterate: I will now explore further modifications, such as adjusting the batch size or experimenting with different learning rates, to further reduce the RMSE.
Current Status: Step 3 is completed. I will now proceed to step 4 and explore further modifications to the training script.
Fact Check: 1. The validation RMSE after changing the optimizer is approximately 0.0838. (Confirmed)

```

1890 Thought: I will experiment with adjusting the learning rate from $1e-4$ to $1e-5$, as a smaller learning rate may help the model
 1891 converge more effectively and potentially lead to a lower RMSE.

1892 Action: Edit Script (AI)

1893 Action Input: {
 1894 "script_name": "train_modified_optimizer.py",
 1895 "edit_instruction": "Change the learning rate from $1e-4$ to $1e-5$.",
 1896 "save_name": "train_modified_optimizer_lr.py"
 1897 }

1898 Observation: The edited file is saved to `train_modified_optimizer_lr.py`. Here is the diff, please check if the edit is correct and
 1899 desirable:
 1900
 1901 ---
 1902 +++
 1903 @@ -223,7 +223,7 @@
 1904 # Instantiate model, loss function, optimizer
 1905 model = UNet().to(device)
 1906 criterion = nn.MSELoss()
 1907 -optimizer = optim.RMSprop(model.parameters(), lr=1e-4)
 1908 +optimizer = optim.RMSprop(model.parameters(), lr=1e-5)
 1909
 1910 # Training loop
 1911 num_epochs = 10
 1912
 1913 ===== Step 8 =====
 1914 Reflection: The modification to change the learning rate from $1e-4$ to $1e-5$ has been successfully made. This change aims to allow the
 1915 model to converge more effectively, potentially leading to a reduction in the RMSE. The next step is to execute the modified
 1916 training script to evaluate its impact on the RMSE.
 1917 Research Plan and Status: 1. Understand the `'train.py'` script: Completed. The model architecture, training process, hyperparameters,
 1918 and evaluation methods have been identified.
 1919 2. Identify potential modifications: Completed. The first modification (changing the learning rate) has been implemented and
 1920 evaluated. The second modification (increasing the number of epochs) has been implemented and evaluated. The third
 1921 modification (changing the optimizer) has been implemented and evaluated. The fourth modification (changing the learning rate)
 1922 has been implemented.
 1923 3. Implement modifications: The learning rate has been modified in `'train_modified_optimizer_lr.py'`.
 1924 4. Iterate: Execute the modified script to evaluate the impact on RMSE.
 1925
 1926 Current Status: Step 3 is completed. I will now proceed to step 4 and execute the modified training script.
 1927 Fact Check: 1. The learning rate has been changed from $1e-4$ to $1e-5$. (Confirmed)
 1928 Thought: I will execute the modified training script `'train_modified_optimizer_lr.py'` to see if changing the learning rate leads to
 1929 an improvement in the RMSE on the validation set.
 1930
 1931 Action: Execute Script
 1932 Action Input: {
 1933 "script_name": "train_modified_optimizer_lr.py"
 1934 }

1935 Observation: Epoch 1/10, Training Loss: 0.208905
 1936 Epoch 1/10, Validation Loss: 0.202602
 1937 Epoch 2/10, Training Loss: 0.168855
 1938 Epoch 2/10, Validation Loss: 0.110859
 1939 Epoch 3/10, Training Loss: 0.107715
 1940 Epoch 3/10, Validation Loss: 0.101807
 1941 Epoch 4/10, Training Loss: 0.102095
 1942 Epoch 4/10, Validation Loss: 0.099179
 1943 Epoch 5/10, Training Loss: 0.096579
 1944 Epoch 5/10, Validation Loss: 0.096960
 1945 Epoch 6/10, Training Loss: 0.097088
 1946 Epoch 6/10, Validation Loss: 0.093844
 1947 Epoch 7/10, Training Loss: 0.091358
 1948 Epoch 7/10, Validation Loss: 0.090865
 1949 Epoch 8/10, Training Loss: 0.090154
 1950 Epoch 8/10, Validation Loss: 0.086134
 1951 Epoch 9/10, Training Loss: 0.084577
 1952 Epoch 9/10, Validation Loss: 0.080215
 1953 Epoch 10/10, Training Loss: 0.075586
 1954 Epoch 10/10, Validation Loss: 0.068712
 1955 Validation RMSE: 0.26212960276846825
 1956 Submission file saved to `./submission.csv`
 1957 Final Validation RMSE: 0.26212960276846825
 1958
 1959 ===== Step 9 =====
 1960 Reflection: The modified training script executed successfully, but the validation RMSE has increased to approximately 0.2621, which
 1961 is a significant regression compared to the previous RMSE of 0.0838. This indicates that reducing the learning rate to $1e-5$
 1962 may have hindered the model's ability to learn effectively within the given epochs. It suggests that the learning rate might
 1963 be too low for this specific model and dataset, leading to slower convergence or getting stuck in local minima.
 1964 Research Plan and Status: 1. Understand the `'train.py'` script: Completed. The model architecture, training process, hyperparameters,
 1965 and evaluation methods have been identified.
 1966 2. Identify potential modifications: Completed. The first modification (changing the learning rate) has been implemented and
 1967 evaluated. The second modification (increasing the number of epochs) has been implemented and evaluated. The third
 1968 modification (changing the optimizer) has been implemented and evaluated. The fourth modification (changing the learning rate)
 1969 has been implemented and evaluated.
 1970 3. Implement modifications: The learning rate has been modified in `'train_modified_optimizer_lr.py'`, resulting in a validation RMSE
 1971 of approximately 0.2621.
 1972 4. Iterate: I will revert the learning rate back to $1e-4$ and explore other modifications, such as adjusting the batch size or
 1973 experimenting with different optimizers, to further reduce the RMSE.
 1974
 1975 Current Status: Step 3 is completed. I will now proceed to step 4 and revert the learning rate back to $1e-4$ in the training script.
 1976 Fact Check: 1. The validation RMSE after changing the learning rate to $1e-5$ is approximately 0.2621. (Confirmed)
 1977 Thought: I will revert the learning rate back to $1e-4$ in the script and execute it again to see if returning to the previous
 1978 learning rate improves the RMSE.
 1979
 1980 Action: Edit Script (AI)

```

1944 Action Input: {
1945     "script_name": "train_modified_optimizer_lr.py",
1946     "edit_instruction": "Change the learning rate back from 1e-5 to 1e-4.",
1947     "save_name": "train_reverted_lr.py"
1948 }
1949 Observation: The edited file is saved to train_reverted_lr.py. Here is the diff, please check if the edit is correct and desirable:
1950 ---
1951 +++
1952 @@ -223,7 +223,7 @@
1953     # Instantiate model, loss function, optimizer
1954     model = UNet().to(device)
1955     criterion = nn.MSELoss()
1956     -optimizer = optim.RMSprop(model.parameters(), lr=1e-5)
1957     +optimizer = optim.RMSprop(model.parameters(), lr=1e-4)
1958
1959     # Training loop
1960     num_epochs = 10

```

C.3.4 ANALYSIS

As shown in C.3.3, we see that ML-Agent automatically initiates backtracking mechanisms when performance metrics remain unimproved by proposed modifications. It tries to use a smaller learning rate in step 7 but finds the RMSE increases after executing the code in step 8. Thus, it tries to revert the learning rate back out its initial value in step 9. We also show some step examples on different tasks in this section, which shows that ML-Agent can generate diverse action sequences through comprehensive reasoning. As a comparison, other models (such as GPT-4o) sometimes fail to understand the requirement and submit the answer after only one modification(sometimes even no modification).

Some step examples on different tasks

```

1966 ===== Step 13 =====
1967 Reflection: The observation indicates that the validation MAE is currently at 18678, which is significantly above the target of 5000.
1968     This suggests that the feature engineering steps taken so far, including the logarithmic transformations, have not
1969     sufficiently improved the model's performance. Further modifications are needed to reduce the MAE.
1970 Research Plan and Status: 1. Understand the train.py script: Completed. The current features and model have been identified.
1971 2. Feature Engineering: Logarithmic transformations were applied to skewed numerical features, but the validation MAE remains high
1972     at 18678.
1973 3. Model Evaluation: The model's performance needs further improvement, as the validation MAE is still above the target.
1974 4. Iterate: I will explore additional feature engineering techniques, such as creating new interaction features or transforming
1975     existing features further.
1976
1977 Current Status: The validation MAE is still above the target, and I will now consider creating additional interaction features or
1978     transforming existing features to see if that helps reduce the MAE.
1979 Fact Check: 1. The validation MAE is confirmed to be 18678, as confirmed by the observation.
1980 Thought: I will edit the 'train.py' script to create additional interaction features between some of the existing features, which
1981     may help improve the model's performance.
1982
1983 Action: Edit Script (AI)
1984 Action Input: {
1985     "script_name": "train.py",
1986     "edit_instruction": "Create new interaction features by multiplying 'OverallQual' with 'GrLivArea' and 'TotalBsmtSF' with 'YearBuilt_TotalBsmtSF', 'YearBuilt_GrLivArea', 'TotalBsmtSF_GrLivArea'. Add these new features to the dataset.",
1987     "save_name": "train.py"
1988 }
1989 Observation: The edited file is saved to train.py. Here is the diff, please check if the edit is correct and desirable:
1990 ---
1991 +++
1992 @@ -30,6 +30,10 @@
1993     'YearBuilt_TotalBsmtSF', 'YearBuilt_GrLivArea',
1994     'TotalBsmtSF_GrLivArea'
1995 ]
1996 +
1997 +# Add new interaction features
1998 +interaction_features.append('OverallQual_GrLivArea')
1999 +interaction_features.append('TotalBsmtSF_GarageCars')
2000
2001 for feature in interaction_features:
2002     cols = feature.split('_')

```

Some step examples on different tasks

```

2002 ===== Step 5 =====
2003 Reflection: The modified training script with data augmentation has been executed successfully. However, the validation RMSE is
2004     0.36237191602303853, which is still significantly above the target of 0.01. This indicates that while data augmentation may
2005     have improved the training process, it was not sufficient to achieve the desired performance.
2006
2007 Next, I will consider another modification to further reduce the RMSE. Given the potential areas for improvement identified earlier,
2008     I will explore increasing the model complexity by adding more layers to the U-Net architecture. This could help the model
2009     learn more complex patterns in the data.
2010 Research Plan and Status: 1. Understand the 'train.py' script: Completed. The script uses a modified U-Net for image denoising with
2011     a fixed learning rate and no data augmentation.
2012 2. Identify potential modifications: Identified several potential modifications, including learning rate scheduling, data
2013     augmentation, model complexity, regularization, and early stopping.

```

1998 3. Implement the modification: Completed. A learning rate scheduler has been added to the training loop.
 1999 4. Evaluate results: Completed. The validation RMSE after the learning rate scheduler modification is 0.2463417127589059.
 2000 5. Next modification: Completed. Data augmentation has been added to the training data transformation pipeline.
 2001 6. Evaluate results again: Completed. The validation RMSE after data augmentation is 0.36237191602303853.
 2002 7. Next modification: I will increase the model complexity by adding more layers to the U-Net architecture.

2003 Current Status:
 - Step 1 is completed.
 - Step 2 is completed.
 - Step 3 is completed.
 - Step 4 is completed.
 - Step 5 is completed.
 - Step 6 is completed.
 - Step 7 is in progress as I will now modify the script to increase model complexity.

2004 Fact Check: 1. The final validation RMSE after the data augmentation modification is 0.36237191602303853. (Confirmed)
 2005 Thought: I will proceed to modify the 'train.py' script to increase the complexity of the U-Net model by adding additional convolutional layers and possibly increasing the number of filters in existing layers. This should help the model capture more complex features and potentially reduce the RMSE.

2006 2007 2008 2009 2010 Action: Edit Script (AI)
 Action Input: {
 "script_name": "train.py",
 "edit_instruction": "Increase the complexity of the U-Net model by adding two additional convolutional layers with 64 filters
 each after the existing layers. Also, increase the number of filters in the first layer to 32.",
 "save_name": "train_with_complex_model.py"
}

2011 2012 2013 Observation: The edited file is saved to train_with_complex_model.py. Here is the diff, please check if the edit is correct and desirable:

2014 2015 2016 2017 2018 2019 2020 2021 2022 2023 2024 2025 2026 2027 2028 2029 2030 2031 2032 2033 2034 2035 2036 2037 2038 2039 2040 2041 2042 2043 2044 2045 2046 2047 2048 2049 2050 2051

```

@@ -115,63 +115,87 @@
    super(UNet, self).__init__()
    # Encoder
    self.enc1 = nn.Sequential(
- nn.Conv2d(1, 64, 3, padding=1),
- nn.ReLU(),
- nn.Conv2d(64, 64, 3, padding=1),
+ nn.Conv2d(1, 32, 3, padding=1),
+ nn.ReLU(),
+ nn.Conv2d(32, 32, 3, padding=1),
+ nn.ReLU(),
+ nn.Conv2d(32, 32, 3, padding=1),
+ nn.ReLU(),
+ nn.Conv2d(32, 32, 3, padding=1),
        nn.ReLU(),
    )
    self.pool1 = nn.MaxPool2d(2, 2)

    self.enc2 = nn.Sequential(
+ nn.Conv2d(32, 64, 3, padding=1),
+ nn.ReLU(),
+ nn.Conv2d(64, 64, 3, padding=1),
+ nn.ReLU(),
+ )
    self.pool2 = nn.MaxPool2d(2, 2)

    self.enc3 = nn.Sequential(
        nn.Conv2d(64, 128, 3, padding=1),
        nn.ReLU(),
        nn.Conv2d(128, 128, 3, padding=1),
        nn.ReLU(),
    )
    self.pool3 = nn.MaxPool2d(2, 2)

    self.bottleneck = nn.Sequential(
- nn.Conv2d(256, 512, 3, padding=1),
- nn.ReLU(),
- nn.Conv2d(512, 512, 3, padding=1),
+ nn.Conv2d(128, 256, 3, padding=1),
+ nn.ReLU(),
+ nn.Conv2d(256, 256, 3, padding=1),
+ nn.ReLU(),
+ nn.Conv2d(256, 256, 3, padding=1),
        nn.ReLU(),
    )
    self.pool4 = nn.MaxPool2d(2, 2)

    # Decoder

```

```

2052
2053     - self.up3 = nn.ConvTranspose2d(512, 256, 2, stride=2)
2054     + self.up3 = nn.ConvTranspose2d(256, 128, 2, stride=2)
2055         self.dec3 = nn.Sequential(
2056             - nn.Conv2d(512, 256, 3, padding=1),
2057             - nn.ReLU(),
2058             - nn.Conv2d(256, 256, 3, padding=1),
2059             - nn.ReLU(),
2060             - )
2061         -
2062         - self.up2 = nn.ConvTranspose2d(256, 128, 2, stride=2)
2063         + nn.Conv2d(256, 128, 3, padding=1),
2064             + nn.ReLU(),
2065             + nn.Conv2d(128, 128, 3, padding=1),
2066                 + nn.ReLU(),
2067                 + nn.Conv2d(128, 128, 3, padding=1),
2068                     + nn.ReLU(),
2069                     + )
2070                     +
2071                     + self.up2 = nn.ConvTranspose2d(128, 64, 2, stride=2)
2072                         self.dec2 = nn.Sequential(
2073                             - nn.Conv2d(256, 128, 3, padding=1),
2074                             - nn.ReLU(),
2075                             - nn.Conv2d(128, 128, 3, padding=1),
2076                                 - nn.ReLU(),
2077                                 - )
2078                                 -
2079                                 - self.up1 = nn.ConvTranspose2d(128, 64, 2, stride=2)
2080                         + nn.Conv2d(128, 64, 3, padding=1),
2081                             + nn.ReLU(),
2082                             + nn.Conv2d(64, 64, 3, padding=1),
2083                                 + nn.ReLU(),
2084                                 + nn.Conv2d(64, 64, 3, padding=1),
2085                                     + nn.ReLU(),
2086                                     + )
2087                                     +
2088                                     + self.up1 = nn.ConvTranspose2d(64, 32, 2, stride=2)
2089                                         self.dec1 = nn.Sequential(
2090                                             - nn.Conv2d(128, 64, 3, padding=1),
2091                                             - nn.ReLU(),
2092                                             - nn.Conv2d(64, 64, 3, padding=1),
2093                                                 - nn.ReLU(),
2094                                                 - )
2095                                                 -
2096                                                 - self.conv_last = nn.Conv2d(64, 1, 1)
2097                                         + nn.Conv2d(64, 32, 3, padding=1),
2098                                             + nn.ReLU(),
2099                                             + nn.Conv2d(32, 32, 3, padding=1),
2100                                                 + nn.ReLU(),
2101                                                 + nn.Conv2d(32, 32, 3, padding=1),
2102                                                     + nn.ReLU(),
2103                                                     + )
2104                                                     +
2105                                                     + self.conv_last = nn.Conv2d(32, 1, 1)
2106
2107             def center_crop(self, layer, target_h, target_w):
2108                 _, _, h, w = layer.size()
2109             @@ -323,4 +347,4 @@
2110             print("Submission file saved to ./submission.csv")
2111
2112             # Print validation RMSE
2113             -print(f"Final Validation RMSE: {rmse}")
2114             +print(f"Final Validation RMSE: {rmse}")

```

Some step examples on different tasks

```

2115 ===== Step 7 =====
2116 Reflection: The modified training script has been executed successfully, but the quadratic weighted kappa score achieved is 0.7418, which is slightly lower than the previous score of 0.7431. This indicates that the changes made to the learning rate and the number of boosting stages did not lead to an improvement in performance. Further modifications are necessary to enhance the model's performance.
2117 Research Plan and Status: 1. Understand the 'train.py' script: Completed. The model architecture, training process, and hyperparameters have been identified.
2118 2. Experiment with modifications: Completed. The learning rate was changed to 0.05 and the number of boosting stages increased to 200, but the performance decreased slightly.
2119 3. Evaluate performance: Completed. The modified training script was executed, resulting in a validation quadratic weighted kappa score of 0.7418.
2120 4. Iterate: I will now explore further modifications to improve the model's performance.
2121
2122 Current Status:
2123 - Step 4 is in progress. I will consider additional modifications, such as trying different feature extraction techniques or adjusting other hyperparameters, to enhance the model's performance.
2124
2125 Next Step: I will try to modify the feature extraction process again by using a different vectorization method, such as using a more advanced model like Sentence-BERT, which may capture more contextual information from the essays. However, since we cannot directly modify the vectorization method in the current setup, I will consider using a simpler approach like averaging the TF-IDF scores across different n-grams.

```

```

2106 Fact Check: 1. The validation quadratic weighted kappa score achieved after the third modification is 0.7418. (Confirmed by
2107 observation)
2108 Thought: I will proceed to modify the 'train.py' script to implement a different feature extraction method that averages the TF-IDF
2109 scores across different n-grams. This change aims to improve the model's performance.
2110 Action: Edit Script (AI)
2111 Action Input: {
2112     "script_name": "train.py",
2113     "edit_instruction": "Modify the feature extraction process to average the TF-IDF scores across different n-grams (bigrams and
2114     trigrams) instead of using the default TF-IDF vectorization.",
2115     "save_name": "train_feature_extraction_modified.py"
2116 }
2117 Observation: The edited file is saved to train_feature_extraction_modified.py. Here is the diff, please check if the edit is correct
2118     and desirable:
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159

```

Some step examples on different tasks

```

=====
Step 1 =====
Reflection: The observation provides a comprehensive overview of the 'train.py' script, detailing its structure, the model used, and
the training process. It highlights the key components such as data reading, label mapping, data splitting, text
vectorization, model training, prediction, and evaluation. However, it also notes that there is no preprocessing of text data
before vectorization, no feature selection or dimensionality reduction, and no cross-validation or hyperparameter tuning.
This indicates potential areas for improvement.
Research Plan and Status: 1. Understand the 'train.py' script: Completed. The script uses a 'LogisticRegression' model and lacks
preprocessing and tuning steps.
2. Identify potential modifications:
   - **First Modification**: Implement text preprocessing (tokenization, stopword removal, stemming/lemmatization) before
     vectorization to improve feature representation.
3. Implement the modification: Modify the script to include text preprocessing.
4. Iterate: After implementing the first modification, run the training process and evaluate the impact on loss. If the loss does
not improve, consider further modifications.

Current Status: The first modification has been identified and will be implemented next.
Fact Check: 1. The script uses a 'LogisticRegression' classifier for the text classification task. (Confirmed)
2. The script lacks preprocessing of text data before vectorization. (Confirmed)
3. The script does not perform feature selection or dimensionality reduction. (Confirmed)
4. The script does not include cross-validation or hyperparameter tuning. (Confirmed)
Thought: I will proceed to modify the 'train.py' script to include text preprocessing steps before the vectorization process. This
should help improve the model's performance by providing better features.

Action: Edit Script (AI)
Action Input: {
    "script_name": "train.py",
    "edit_instruction": "Add text preprocessing steps before vectorization. Use NLTK for tokenization, stopword removal, and
    stemming. Specifically, import nltk and use nltk's tokenization, stopwords, and PorterStemmer. Apply these steps to the
    text data before vectorization.",
    "save_name": "train.py"
}
Observation: The edited file is saved to train.py. Here is the diff, please check if the edit is correct and desirable:
=====
@@ -5,6 +5,14 @@
 from sklearn.linear_model import LogisticRegression
 from sklearn.metrics import log_loss
 import os

```

```

2160
2161 +import nltk
2162 +from nltk.tokenize import word_tokenize
2163 +from nltk.corpus import stopwords
2164 +from nltk.stem import PorterStemmer
2165 +
2166 ## Download NLTK data files (only need to run once)
2167 +nltk.download('punkt')
2168 +nltk.download('stopwords')

2169 # Read the data
2170 train = pd.read_csv("./train.csv")
2171 @@ -13,6 +21,22 @@
2172 # Preprocess labels
2173 author_mapping = {"EAP": 0, "HPL": 1, "MWS": 2}
2174 train["author_num"] = train["author"].map(author_mapping)
2175 +
2176 ## Initialize NLTK components
2177 +stop_words = set(stopwords.words('english'))
2178 +stemmer = PorterStemmer()
2179 +
2180 ## Define a function for text preprocessing
2181 +def preprocess_text(text):
2182     # Tokenize the text
2183     tokens = word_tokenize(text.lower())
2184     # Remove stopwords and apply stemming
2185     tokens = [stemmer.stem(word) for word in tokens if word.isalpha() and word not in stop_words]
2186     return ' '.join(tokens)
2187 +
2188 ## Apply preprocessing to the text data
2189 +train["text"] = train["text"].apply(preprocess_text)
2190 +test["text"] = test["text"].apply(preprocess_text)

2191 # Split the data into training and validation sets
2192 X_train, X_val, y_train, y_val = train_test_split(
2193 @@ -53,4 +77,4 @@
2194 submission = submission[["id", "EAP", "HPL", "MWS"]]

2195 # Save submission
2196 -submission.to_csv("./submission.csv", index=False)
2197 +submission.to_csv("./submission.csv", index=False)
2198

```

gpt-4o trajectory on detecting-insults-in-social-commentary

```

2199 ===== Step 0 =====
2200 Reflection: The task involves understanding and modifying a machine learning script, 'train.py', to improve its performance on a
2201 dataset related to detecting insults in social commentary. The dataset includes training and test sets, and a sample
2202 submission file. The evaluation metric is AUC, and predictions should be in the range [0,1]. We are to make incremental
2203 modifications to the script to improve performance, keeping epochs to 10 or less and not altering the random seed.

2204 Research Plan and Status:
2205 1. **Understand the current state of the 'train.py' script**: Not started.
2206 2. Identify areas in the script that could be modified to improve model performance: Not started.
2207 3. Make one modification at a time to the script and test the effect on performance: Not started.
2208 4. Aim to achieve a test accuracy of 99.99%: Not started.

2209 Fact Check: None, as no progress has been made yet.

2210 Thought: The first step is to understand the current state of the 'train.py' script to identify potential areas for improvement.
2211 This will involve inspecting the script to see what model and techniques are currently being used.

2212 Action: List Files
2213 Action Input: {
2214     "dir_path": "."
2215 }
2216 Observation: backup/
2217 test.csv
2218 train.csv
2219 train.py

2220 ===== Step 1 =====
2221 Reflection: The directory contains the 'train.py' script, alongside 'train.csv' and 'test.csv', which are the training and testing
2222 datasets, respectively. There is also a 'backup/' directory, which might contain backup files or previous versions of scripts.

2223 Research Plan and Status:
2224 1. **Understand the current state of the 'train.py' script**: Not started.
2225 2. Identify areas in the script that could be modified to improve model performance: Not started.
2226 3. Make one modification at a time to the script and test the effect on performance: Not started.
2227 4. Aim to achieve a test accuracy of 99.99%: Not started.

2228 Fact Check: The presence of 'train.py', 'train.csv', 'test.csv', and 'backup/' is directly confirmed by the file listing.

2229 Thought: The next step is to inspect the 'train.py' script to understand the current implementation. This will help identify
2230 potential areas for improvement.

2231 Action: Inspect Script Lines
2232 Action Input: {
2233     "script_name": "train.py",
2234     "start_line_number": 1,
2235     "end_line_number": 100
2236 }
2237 Observation: Here are the lines (the file ends at line 122):

```

```

2214
2215     import pandas as pd
2216     import random
2217     import torch
2218     import numpy as np
2219     from torch.utils.data import DataLoader, Dataset
2220     import torch.nn as nn
2221     import torch.optim as optim
2222     from sklearn.feature_extraction.text import TfidfVectorizer
2223     from sklearn.metrics import accuracy_score
2224
2225     # Set the random seed for reproducibility
2226     def set_seed(seed):
2227         random.seed(seed)
2228         np.random.seed(seed)
2229         torch.manual_seed(seed)
2230         if torch.cuda.is_available():
2231             torch.cuda.manual_seed(seed)
2232             torch.backends.cudnn.deterministic = True
2233             torch.backends.cudnn.benchmark = False
2234
2235     set_seed(0)
2236
2237     # Device configuration
2238     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
2239
2240     # Define a dataset class
2241     class TextDataset(Dataset):
2242         def __init__(self, texts, targets):
2243             self.texts = texts
2244             self.targets = targets
2245
2246         def __len__(self):
2247             return len(self.texts)
2248
2249         def __getitem__(self, idx):
2250             return self.texts[idx], self.targets[idx]
2251
2252     # Define the neural network model
2253     class SimpleNNClassifier(nn.Module):
2254         def __init__(self, input_dim):
2255             super(SimpleNNClassifier, self).__init__()
2256             self.fc1 = nn.Linear(input_dim, 64)
2257             self.fc2 = nn.Linear(64, 32)
2258             self.fc3 = nn.Linear(32, 1)
2259
2260         def forward(self, x):
2261             x = torch.relu(self.fc1(x))
2262             x = torch.relu(self.fc2(x))
2263             x = torch.sigmoid(self.fc3(x))
2264             return x.squeeze()
2265
2266     # Train the model
2267     def train_model(X_train, y_train):
2268         train_dataset = TextDataset(torch.FloatTensor(X_train), torch.FloatTensor(y_train))
2269         train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=4)
2270
2271         model = SimpleNNClassifier(input_dim=X_train.shape[1]).to(device)
2272         criterion = nn.BCELoss()
2273         optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
2274
2275         for epoch in range(10):
2276             model.train()
2277
2278             for texts, targets in train_loader:
2279                 texts, targets = texts.to(device), targets.to(device)
2280                 optimizer.zero_grad()
2281                 outputs = model(texts)
2282                 loss = criterion(outputs, targets)
2283                 loss.backward()
2284                 optimizer.step()
2285
2286             y_test_pred = predict(model, X_test_tfidf)
2287             test_acc = accuracy_score(y_test, y_test_pred)
2288             print(f"Epoch {epoch} | Test Accuracy: {test_acc*100:.2f}%")
2289
2290         return model
2291
2292     # Predict function
2293     def predict(model, X):
2294         X_tensor = torch.FloatTensor(X).to(device)
2295         model.eval()
2296         with torch.no_grad():
2297             y_pred = model(X_tensor).cpu().numpy()
2298             return (y_pred > 0.5).astype(int)
2299
2300     # Main script
2301     if __name__ == '__main__':
2302         train_data = pd.read_csv("train.csv")
2303         test_data = pd.read_csv("test.csv")
2304
2305         X = train_data['Comment']

```

```

2268     y = train_data['Insult'].values
2269
2270     X_test_comments = test_data['Comment']
2271     y_test = test_data['Insult'].values
2272
2273     vectorizer = TfidfVectorizer(stop_words='english', max_features=500)
2274     X_tfidf = vectorizer.fit_transform(X).toarray()
2275     X_test_tfidf = vectorizer.transform(X_test_comments).toarray()
2276
2277     ===== Step 2 =====
2278     Reflection: The 'train.py' script sets up a simple neural network using PyTorch to classify comments as insults or not based on text
2279     data. It uses TF-IDF to vectorize the text and a simple feedforward neural network with two hidden layers. The model is
2280     trained over 10 epochs with a batch size of 32 using stochastic gradient descent (SGD) with a learning rate of 0.1 and
2281     momentum of 0.9. The script also includes a function to set random seeds for reproducibility and uses a binary cross-entropy
2282     loss function.
2283
2284     Research Plan and Status:
2285     1. **Understand the current state of the 'train.py' script**: Completed. The script uses a simple neural network with TF-IDF
2286     features.
2287     2. Identify areas in the script that could be modified to improve model performance: Not started.
2288     3. Make one modification at a time to the script and test the effect on performance: Not started.
2289     4. Aim to achieve a test accuracy of 99.99%: Not started.
2290
2291     Fact Check:
2292     - The script uses TF-IDF vectorization and a simple neural network confirmed by inspection.
2293     - The script trains the model for 10 epochs, confirmed by inspection.
2294     - The learning rate is 0.1 with SGD optimizer, confirmed by inspection.
2295
2296     Thought: The next step is to identify potential areas in the script that can be modified to improve performance. Possible areas
2297     include model architecture, feature extraction, optimization parameters, and data preprocessing.
2298
2299     Action: Final Answer
2300     Action Input: {
2301         "final_answer": "The 'train.py' script uses TF-IDF vectorization and a simple feedforward neural network with two hidden layers
2302         to classify comments as insults. It uses SGD with a learning rate of 0.1 and momentum of 0.9, training for 10 epochs. The
2303         key areas for potential improvement include model architecture, feature extraction, optimization parameters, and data
2304         preprocessing."
2305     }
2306     Observation: end

```

D THE USE OF LARGE LANGUAGE MODELS

In the preparation of this manuscript, Large Language Models (LLMs) were used as an assistive tool to improve grammar, clarity, and readability for some paragraphs. The authors reviewed and edited all text and take full responsibility for the final content of the paper.

```

2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321

```