Dartmouth College

COSC 89.34/189 · AI Agents

Problem Set 1: Introduction to Agents

Instructor: Nikhil Singh

Welcome

This first problem set is designed to introduce you to the conceptual and computational foundations of AI Agent design and engineering. We will use software "copilot" style agents as a reference application throughout.

Part I: The Copilot's Dilemma

We will begin our journey with a simple kind of AI Agent, something like GitHub Copilot; i.e. this is an interactive AI assistant that may optionally surface a suggestion to a human user while the user is engaged in a task. Showing a suggestion can save time if it is accepted, but can also impose cognitive and latency costs. The assistant must decide, in real-time and under uncertainty, whether a suggestion should be generated and/or displayed. This first problem develops a formal decision-theoretic treatment of this setting from first principles.

Problem Setup and Notation

At a decision point t, the assistant observes:

- Context features $x_t \in \mathbb{R}^d$, available before generating any suggestion.
- If a suggestion is generated, suggestion features $s_t \in \mathbb{R}^k$.

There is an unobserved latent user state $\phi_t \in \{1, \ldots, K\}$. If a suggestion is shown, the user either accepts it ($A_t = 1$) or rejects it ($A_t = 0$). We treat A_t as the (potential) accept/reject outcome *if the generated suggestion were shown.* In data, A_t is observed only when the suggestion is actually shown. Let

$$p_t^\star = \mathbb{P}(A_t = 1 \mid x_t, s_t, \phi_t) \tag{1}$$

denote the true acceptance probability. The assistant does not observe ϕ_t and instead learns predictors

$$\hat{p}^{(1)}(x_t) = \mathbb{P}(A_t = 1 \mid x_t), \qquad \hat{p}^{(2)}(x_t, s_t) = \mathbb{P}(A_t = 1 \mid x_t, s_t). \tag{2}$$

We model the interaction in terms of time. All listed costs are nonnegative scalars:

- c_{verify} is the time to read and evaluate a shown suggestion
- c_{edit} is the additional time if a shown suggestion is accepted
- c_{write} is the time to complete the task manually after rejecting a shown suggestion
- c_{noshow} is the time to complete the task manually if no suggestion is shown
- τ is the latency cost incurred when a suggestion is generated

Expected time and value of showing a suggestion

Problem 1. Assume a suggestion is generated and shown and has acceptance probability $p \in [0, 1]$. Write an expression for the expected total time $\mathbb{E}[T \mid \text{show}]$.

Problem 2. Define the utility of showing a suggestion as the *time saved relative to not showing* one. Express this quantity and simplify.

Problem 3. Assume $c_{\text{noshow}} = c_{\text{write}}$ and $c_{\text{write}} > c_{\text{edit}}$. Derive the acceptance-probability threshold p^* above which showing a suggestion *reduces* expected time.

Acceptance probability as a sufficient statistic

Problem 4. Suppose for a moment that the assistant knows the true acceptance probability $p_t^\star$ and that the costs $(c_{\text{verify}}, c_{\text{edit}}, c_{\text{write}}, c_{\text{noshow}}, \tau)$ are fixed constants. Show that, under the time-based objective above, no additional information about (x_t, s_t) is required to decide whether to show the suggestion.

Problem 5. Explain why, in practice, the assistant relies on $\hat{p}^{(2)}(x_t, s_t)$ instead of $p_t^\star$, and state the implicit assumption required for optimality.

Two-stage conditional display

Generating a suggestion can itself be costly. We probably don't want to generate a suggestion every chance we get (for example, every time the programmer pauses typing) before deciding whether to show it. When we are able to make an accurate display decision before generating a suggestion, we should do so.

We will therefore consider a two-stage policy. Let $m(x, s) \in \{0, 1\}$ denote the final show decision *when a suggestion s exists.* We define

$$m(x, s) = r(x)\, m_1(x) + (1 - r(x))\, m_2(x, s) \tag{3}$$

- $m_1(x) = \mathbb{1}\{\hat{p}^{(1)}(x) \geq t_1\}$ is a coarse, context-only display rule
- $m_2(x, s) = \mathbb{1}\{\hat{p}^{(2)}(x, s) \geq t_2\}$ is a refined rule using the suggestion
- $r(x) \in \{0, 1\}$ determines whether we *skip generation* and use the coarse decision rule

Problem 6. Suppose $r(x) = 1$ when the Bernoulli entropy of the stage-1 prediction

$$H(\hat{p}^{(1)}(x)) = -\hat{p}^{(1)}(x) \log \hat{p}^{(1)}(x) - (1 - \hat{p}^{(1)}(x)) \log(1 - \hat{p}^{(1)}(x)) \tag{4}$$

is $\leq t_r$ (use natural logs). How should we interpret this?

Problem 7. Suppose thresholds (t_1, t_2, t_r) are chosen to maximize

$$J = \lambda \mathbb{E}[1 - m(x, s)] + (1 - \lambda)\mathbb{E}[r(x)] \tag{5}$$

$$s.t.\ \mathbb{P}(A = 0 \mid m(x, s) = 0) \geq \beta \tag{6}$$

where the probability/expectations are with respect to the (logged) joint distribution over (x, s, A) for generated suggestions, and A is the accept/reject outcome if shown. Explain the point of this constraint.

Optimizing which suggestion we should show

Up to this point, we have studied the decision of *whether* to display a suggestion. We now turn to a related but distinct question: *which* suggestion should be displayed when multiple candidate suggestions are available.

Consider a task with a unique correct solution represented as a sequence of tokens

$$y^* = (y_1, y_2, \ldots, y_L), \tag{7}$$

where $L \in \mathbb{N}$ is the solution length. At a given decision point, the assistant may choose to display one suggestion from a finite candidate set. We consider the following structured candidate set:

$$\mathcal{S} = \{s_\ell : \ell = 1, \ldots, L\}, \quad s_\ell \equiv (y_1, \ldots, y_\ell) \tag{8}$$

Each candidate suggestion is thus a prefix of the correct solution with prefix length ℓ.

The acceptance preference model. For each candidate suggestion s_ℓ, define the probability that the user accepts the suggestion if it is shown:

$$R(\ell) \equiv \mathbb{P}(A = 1 \mid s_\ell). \tag{9}$$

Acceptance reflects a *revealed preference* for using the suggestion rather than writing manually.

We model $R(\ell)$ as the product of two effects:

- A *usefulness* effect, i.e. very brief suggestions may not be worth the interruption
- A *verification burden* effect, i.e. longer suggestions are harder to read and verify

Concretely, we will assume:

$$R(\ell) = \left(1 - e^{-\gamma \ell}\right) q^\ell, \qquad \gamma > 0,\ q \in (0, 1). \tag{10}$$

The first term penalizes suggestions that are too brief to be useful; the second captures the increasing verification cost with length. Under this model, $R(\ell)$ is generally unimodal.

Utility of acceptance. If suggestion s_ℓ is accepted, it replaces the need for the user to manually write ℓ tokens. We model the time saved upon acceptance as

$$U(\ell) = \alpha \ell^\rho \tag{11}$$

for some constant $\alpha > 0$ and $\rho \in (0, 1)$. If the suggestion is rejected, no time is saved. We compare two objectives for selecting which suggestion to show.

Acceptance-only objective. A system trained purely on acceptance feedback selects

$$\ell_{\text{acc}} \in \operatorname*{arg\,max}_{\ell \in \{1,\ldots,L\}} R(\ell) \tag{12}$$

Time-based objective. Under the time-saving objective, the expected time saved by showing s_ℓ is

$$\mathbb{E}[\text{time saved} \mid s_\ell] = R(\ell)\, U(\ell) = \alpha \ell^\rho R(\ell) \tag{13}$$

and the optimal choice is

$$\ell_{\text{time}} \in \operatorname*{arg\,max}_{\ell \in \{1,\ldots,L\}} \ell\, R(\ell) \tag{14}$$

Problem 8. Under the acceptance-only objective, characterize ℓ_{acc} in terms of the shape of $R(\ell)$.

Problem 9. Give a necessary and sufficient condition, stated in terms of $R(\ell)$, ρ, and ℓ_{acc}, under which $\ell_{\text{time}} > \ell_{\text{acc}}$.

Problem 10. Explain why optimizing for acceptance probability alone constitutes a form of proxy optimization failure in this setting.

Part II: Whither Suggestions?

Part I treated the suggestion policy as an abstract decision rule. What are these suggestions, and where do they come from?

Part II makes the other half of the copilot concrete. In it, we will produce a model that might generate such candidate suggestions, and a training loop that improves that generator using rewards derived from automatic code evaluation.

First, it's worth ruling out simpler alternatives. One such alternative is supervised learning on reference solutions. Assume we start with a language model, this corresponds to supervised fine-tuning (SFT). Why not just do this? If we have a dataset of solutions, then surely this is the simpler way?

Problem 11. You are given a distribution over problems $P \sim \mathcal{D}$. Each problem has a set of correct solutions $\mathcal{Y}(P)$. A supervised dataset provides exactly one reference solution $Y^{\text{ref}}(P) \in \mathcal{Y}(P)$ per problem.

Let $\pi_\theta(y \mid P)$ be the model's conditional distribution over solutions (token sequences).

(a) Write the SFT objective (negative log-likelihood) for this dataset.

(b) When might minimizing this objective reduce probability mass on other correct solutions?

Problem 12. Suppose a problem comes with a test suite of size n; a candidate solution passes k tests. Consider two reward choices:

$$R_{\text{binary}} = \mathbb{1}\{k = n\}, \qquad R_{\text{fraction}} = k/n \tag{15}$$

(a) What learning signal does R_{fraction} contain that R_{binary} does not?

(b) Give at least one reason that we might prefer R_{binary}.

The simplest code generator

We now make the suggestions from Part I concrete. A suggestion is a completion sampled from a code model under a fixed prompt format and stopping condition.

Problem 13. Write pseudocode for a function

$$\texttt{complete}(P) \to \text{string} \tag{16}$$

that samples a solution from a language model in text (make sure to reason about *all* of the steps involved). You must specify at least one stop condition for the sampling that is not "max tokens."

Problem 14. In practice, different base models may require different prompt "renderers" (chat templates).

(a) What can go wrong if you use the wrong renderer for a given model?

(b) Give at least one symptom you might observe in practice.

Evaluating code

We need a procedure that maps a model completion to a scalar *reward* in such a way that is fully automated, robust, and safe.

Problem 15. A model response may include explanations plus one or more *code fences* (strings demarcating code blocks).

(a) Define a deterministic extraction rule that maps raw text to a single Python program string or returns $\varnothing$ if extraction fails.

(b) Why should extraction be treated as a first-class part of the environment, rather than an incidental preprocessing step?

Problem 16. Explain why `exec()` on the host machine is usually not acceptable for evaluating LLM code in a training loop. Give two distinct failure modes, one accidental and one adversarial.

Problem 17. Consider a shaped reward

$$r = r_{\text{format}} + r_{\text{correct}} \tag{17}$$

where $r_{\text{format}} \in \{-0.1, 0\}$ and $r_{\text{correct}} \in \{0, 1\}$.

(a) Give an argument for why r_{format} should be able to be negative rather than zero.

(b) Give an argument *against* including r_{format} at all.

Policy gradients for LLMs

Now we optimize π_θ with respect to expected reward. Note: the "action" here is a long sequence of discrete tokens.

Problem 18. Let $y = (y_1, \ldots, y_T)$ be a sampled completion and $R(y)$ the scalar reward after evaluation. Show that the policy gradient can be written as

$$\nabla_\theta \mathbb{E}_{y \sim \pi_\theta(\cdot|P)}[R(y)] = \mathbb{E}\left[R(y) \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(y_t \mid P, y_{<t})\right]. \tag{18}$$

Problem 19. Let $b(P)$ be a baseline that depends on the difficulty of the problem but not on sampled y. Define advantage $A(y) = R(y) - b(P)$.

(a) Show that replacing $R(y)$ by $A(y)$ in Problem 18 leaves the expected gradient unchanged.

(b) Give one reason this replacement helps in practice.

Variance reduction via grouping

In code RL, rewards are sparse and high-variance. A common trick is to sample multiple solutions per prompt and normalize within that group.

Problem 20. For a fixed problem P, sample G completions $\{y^{(g)}\}_{g=1}^{G}$ with rewards $\{R_g\}$. Define

$$\bar{R} = \frac{1}{G} \sum_{g=1}^{G} R_g, \qquad A_g = R_g - \bar{R} \tag{19}$$

(a) Show that $\sum_{g=1}^{G} A_g = 0$.

(b) Interpret this as a statement about what information the update uses.

Problem 21. Under the same setup, suppose all rewards in a group are equal i.e. $R_1 = \cdots = R_G$.

(a) What are the advantages $\{A_g\}$?

(b) What does this imply about the learning signal for that problem under our current estimator?

Problem 22. You will now implement an RL post-training pipeline using Tinker[1]. The goal is to train a code-generating language model to solve programming problems by optimizing (mostly) a binary correctness signal from automated test execution.

We have provided a library of tools to help you interact efficiently with Tinker. You will provide the main RL training and evaluation loop, using *Qwen 3 4B Instruct* as the initial checkpoint, group relative policy optimization (GRPO) as the algorithm.

First, a few preliminaries:

(a) **Tinker** is an API for post-training language models, developed by *Thinking Machines*. The basic principle is to allow you to write your training code locally, execute it locally, and have the actual model sampling and training run remotely. You may find the **Tinker Cookbook**[2] useful as a reference (but, try not to just directly port code from there — we'll check).

(b) **LoRA** (Low-Rank Adaptation) is a parameter-efficient fine-tuning method that freezes the pretrained model weights and injects trainable low-rank decomposition matrices into each layer. Instead of updating all parameters $W \in \mathbb{R}^{d\times k}$, LoRA learns $W' = W + BA$ where $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$ with rank $r \ll \min(d, k)$. This often reduces the number of trainable parameters substantially, but maintains reasonable performance. For this assignment, you will use LoRA with **rank 32**.

(c) **GRPO** is a policy gradient method designed for language model fine-tuning. Unlike PPO (Proximal Policy Optimization), which requires a separate value network, GRPO uses group-based advantage estimation, i.e.:

- Sample K completions for each prompt (the "group")
- Compute the mean reward across the group: $\bar{r} = \frac{1}{K}\sum_{i=1}^{K} r_i$
- Use group-relative advantages: $A_i = r_i - \bar{r}$
- Update policy with importance sampling correction. This corrects for the fact that you sampled outputs from an old policy but need gradients for your current policy, so you reweight by how much more/less likely the current policy would have generated that same output (Tinker will use the provided `logprobs` for this; see docs for more[3]).

Similar to the DeepCoder[4] recipe, we *will not* require you to use entropy/KL terms.

You will train on the **DeepCoder-Preview** dataset, which contains competitive programming problems from multiple sources (TACO Verified, PrimeIntellect SYNTHETIC-1, and LiveCodeBench). Each problem includes a natural language description, test cases, optional starter code, and some metadata. The model must generate Python code from the description that passes all test cases. The reward function is:

$$r = \alpha \cdot r_{\text{format}} + r_{\text{correct}} \tag{20}$$

where:

- $r_{\text{format}} \in \{0, -1\}$: whether output is properly formatted

[1] thinkingmachines.ai/tinker
[2] github.com/thinking-machines-lab/tinker-cookbook
[3] tinker-docs.thinkingmachines.ai/losses
[4] together.ai/blog/deepcoder

- $r_{\text{correct}} \in \{0, 1\}$: whether code passes all test cases
- $\alpha = 0.1$: format coefficient (controls penalty scale for format errors)

You are given `train.py`, which contains the basic configuration and dataset loading. Your task is to complete the training pipeline by implementing the main training loop. This will also include the evaluation loop, checkpointing, and many other aspects of setup. The default configuration in the file is the one you should use (unless you would like to experiment).

Your implementation should be able to, at minimum:

- Train for the specified number of batches (or the full dataset if `max_steps = -1`)
- Log training and evaluation metrics at regular intervals
- Save checkpoints periodically and at completion
- Handle sampling and environment execution efficiently (consider using `async` here)
- Skip degenerate groups (where all rewards are identical)

Test your implementation by running a few steps (we recommend at least 20), and ideally produce evidence consistent with a modest improvement on the test set before vs. after training. However, we know this is under high variance and a rather small training and evaluation budget, so we will primarily look for evidence your pipeline is wired correctly. Submit a fully reproducible `train.py` script only, along with a `requirements.txt` file if you add any additional dependencies. It is your responsibility to make sure we can run your code.

Part III: Plan of Action

Modern coding agents go beyond the copilot role, often proceeding from sparse natural language specifications to full implementations. In this assignment, you will build *SimpleCoder*, a CLI coding agent that can help a user write code, navigate codebases, and complete software engineering tasks. This agent will combine several concepts from the modern AI agent stack: tool use, RAG, context management, and task planning. This is similar to Claude Code, but of course is greatly simplified and will be implemented in Python (though, if you choose, you are certainly welcome to implement in Rust or otherwise and add a Python frontend to it). By the end of this part, you should have a working coding agent that you will use to build a demo project.

Note: you are *not* actually expected to use the model you post-trained in part III, given the logistical difficulties serving this at scale. You may use any LLM API you choose (including Dartmouth Chat APIs, or Gemini for which we will make some credits available).

You are provided with:

- `simplecoder/main.py` contains the CLI entrypoint (complete, do not modify)
- `pyproject.toml` contains the package configuration (complete, do not modify)

You need to implement, at minimum:

- `simplecoder/agent.py` for main agent logic. Use a ReAct loop.
- `simplecoder/tools.py` for tool functions and schemas. At minimum, you need tools to list, read, search, write, and edit source files.
- `simplecoder/rag.py` for RAG for code search. When working on large codebases, we cannot fit all code into the context window. RAG (Retrieval-Augmented Generation) is the traditional solution to this. Conventionally, we would slice the content into chunks and retrieve relevant components via text or embedding search. We ask that you use abstract syntax trees (ASTs) to chunk code in a more principled way (e.g. into functions, classes, etc.) and implement embedding-based RAG on top of these representations.
- `simplecoder/context.py` for context management and compacting. LLMs have token limits and long conversations may eventually exceed their context window, despite our efforts to contain codebases themselves via RAG. We ask you to implement a toolkit for managing this: track context use, and summarize conversation history when exceeding a certain point (you can choose to set this heuristically rather than being model specific, for the course of this assignment). You should allow for keeping the last k (configurable number) messages intact.
- `simplecoder/planner.py` for task planning and decomposition. Should take a task description and return a set of subtasks, with a procedure for managing their incremental completion towards the goal.
- `simplecoder/permissions.py` for managing task- and session-level permissions from the user for reading, writing files, etc.

Try to engineer the agent so that the interaction feels responsive and there is sufficient feedback for a user to judge progress.

Test your implementation by screen-recording a session where you vibe-code with it to build you a simple project (for example, a simple Python-based text-adventure game). Write a README.md explaining and justifying your specific approach to solving each of the above problems (max. 1 paragraph per listed module). Submit a fully reproducible package + your example project that we can install and immediately use. It is your responsibility to make sure we can run your code.