Intro:
We're exploring whether reinforcement learning at test time can reliably specialize an LLM for "hard to solve, easy to verify" optimization problems, tuning pretrained LLM models at test time.

Motivation:
Instead of asking an LLM to judge an LLM, we evaluate the model on problems where correctness is strictly verifiable by a deterministic procedure. In other words, the model generates a candidate solution, and then a program or mathematical check tells us unambiguously whether it's valid and how good it is.

There are lots of natural examples in this category: optimizing kernel speed where you can literally benchmark runtime, discovering faster matrix multiplication variants where you can verify correctness and count operations, and even certain Erdős-style combinatorics problems where a construction can be checked automatically. This verifiable setup is what makes an iterative improvement loop possible, because the feedback signal is reliable."

Background:
Google DeepMind recently introduced **AlphaEvolve**, a system aimed at **autonomous discovery** in math and computing. The core idea is that it treats the LLM like a creative engine, but it does not trust it to decide what is correct. Instead, AlphaEvolve repeatedly proposes **small edits** to an existing solution or codebase, runs an **automated evaluator** to verify and score the result, and then uses an evolutionary style loop to keep refining the best candidates over many rounds

Those examples motivate our project: we want the same verifiable discovery setup, and we ask whether we can push it further by adding gradient based learning during the search loop.

Circle packing: introduction
Circle packing is our main 'strictly verifiable' benchmark. The task is: place **26 circles inside a unit square** so that they **do not overlap** and **every circle stays fully within the boundaries**. The objective is to **maximize the sum of all radii**, so higher sum means a denser packing

Idea: Add in reinforcement learning during test time

This is the overall AlphaGrad loop: generate a candidate packing, verify it with deterministic geometric checks, score it by sum of radii if valid or zero otherwise, then use those rewards to update a LoRA adapter. We repeat while keeping a buffer of best solutions so we combine search with learning.

TTT-Discover:

The LLM **doesn't directly print 26 circle coordinates and radii**. Instead, it **writes Python code** that *computes* a packing. We then **execute that code locally**, collect the resulting centers

and radii, and run the same deterministic verifier to check boundary and non overlap constraints.

The reason we do this is that it changes the nature of the search. The LLM is effectively proposing an **algorithmic construction**, not just a single candidate solution. That's more powerful than searching directly in coordinate space, because coordinate space is huge and unstructured, and small changes can easily break validity.

By searching over programs, we get several benefits. First, we get **structure and reuse**: the code can encode symmetry, templates, or heuristics that generalize across attempts. Second, it becomes **easier to enforce constraints programmatically**, because the code can bake in rules like clipping to boundaries or maintaining minimum distances. And third, we're searching a **richer space** overall—programs can generate families of solutions and refinement steps, instead of just one fixed point.

Initial results:
At Step 0, the model is pretty unreliable: only about 21% of outputs are in the correct format and about 20% actually pass the verifier. That means most of our compute early on is wasted on invalid candidates.

As training progresses, two things improve in parallel. First, format compliance increases dramatically, ending at 96.8% by Step 4. Second, the correctness rate—meaning candidates that are valid under the geometric constraints—rises from 19.9% to 61.5%, so validity roughly triples. This is important because higher validity means we get a denser and more informative reward signal for learning.

Then there's the quality of the best solutions. The best score—sum of radii—steadily improves, and by Step 4 we reach 2.6360, which matches the known benchmark target for CP26. So we're not just producing more valid outputs, we're actually finding state-of-the-art constructions under this evaluation.

Finally, the average performance among valid samples also rises: the mean score increases from about 2.23 to 2.55, which is around a 14–15% improvement. So overall, the trajectory looks like an implicit curriculum: the model first learns to output in the right structure, then becomes more consistently valid, and then concentrates probability mass on higher-quality packings.

Further steps:

"Here's what we do next, and we're focusing specifically on **CP26** because it's fast to evaluate and already shows a strong learning signal.

First, we need to **compare to a baseline model** so we can make a causal claim. Right now, our improvements could be coming from the PUCT buffer alone, not from the LoRA updates. So the next run is a frozen baseline: same prompts, same sampling, same buffer logic, same verifier — but **no weight updates**. Then we compare step-by-step curves like validity rate, best score, and how quickly we reach the benchmark target. If the frozen run matches our learning run, the story is mostly search. If it lags, that's evidence the RL updates are actually helping.

Second, we're choosing **CP26 as the main optimization target** going forward. It gives clean deterministic rewards, rapid iteration, and we've already hit the known target, so improvements are now about efficiency and reliability — doing it faster and with fewer wasted samples.

Then we have two concrete levers.

One is **token efficiency**. Early steps waste a lot of tokens because outputs are malformed or include unnecessary explanation. We can constrain the generation format to be minimal — basically 'return the array of centers and radii' with no extra text — and add stricter parsing rules. The goal is to reduce wasted compute and increase the number of valid attempts per budget, especially in the first couple of steps.

The second lever is **reward shaping**. Right now it's very sparse: valid solutions get a reward, invalid get zero. That's clean, but it means the model gets almost no learning signal from near-misses. We can instead assign partial credit to invalid packings based on how close they are — for example, penalize overlap magnitude and boundary violations. That gives a smoother gradient early on and should raise validity faster, which then helps the system reach high-quality packings sooner.