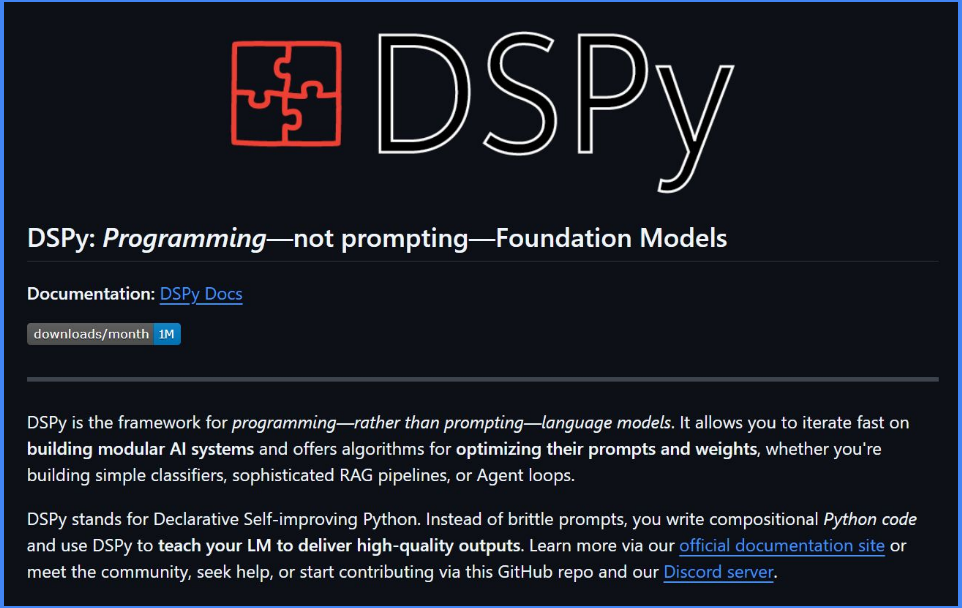
DSPy: Discussion

(Declarative Self-improving Python)



DSPy: *Programming*—not prompting—Foundation Models

Documentation: DSPy Docs

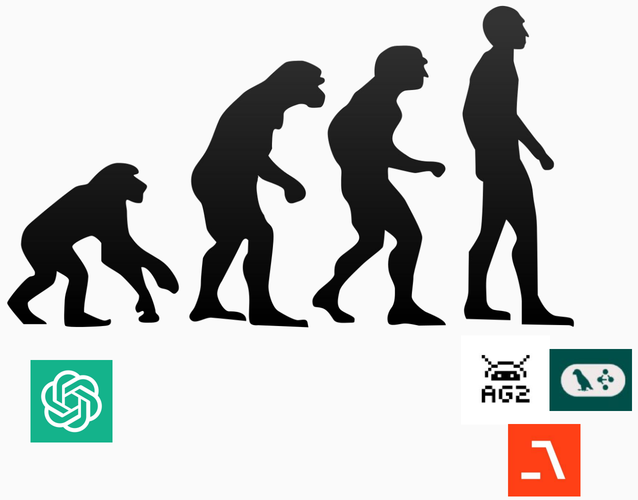
downloads/month 1M

DSPy is the framework for *programming—rather than prompting—language models*. It allows you to iterate fast on **building modular AI systems** and offers algorithms for **optimizing their prompts and weights**, whether you're building simple classifiers, sophisticated RAG pipelines, or Agent loops.

DSPy stands for Declarative Self-improving Python. Instead of brittle prompts, you write compositional *Python code* and use DSPy to **teach your LM to deliver high-quality outputs**. Learn more via our official documentation site or meet the community, seek help, or start contributing via this GitHub repo and our Discord server.

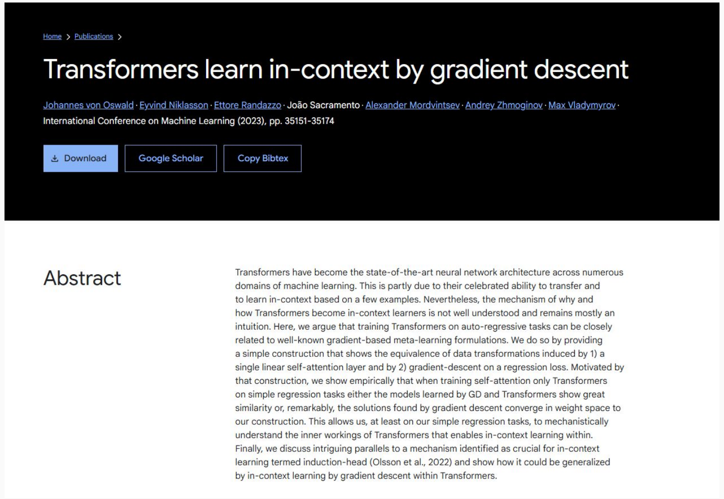
Prompting

Crafting and fine-tuning the natural-language “prompt” you feed into an LLM so that its outputs reliably match your needs.



Importance of prompting

1. Prompting defines system behavior (LLMs are internet native)
2. Large models exhibit In-Context Learning (ICL emerges from forward pass)

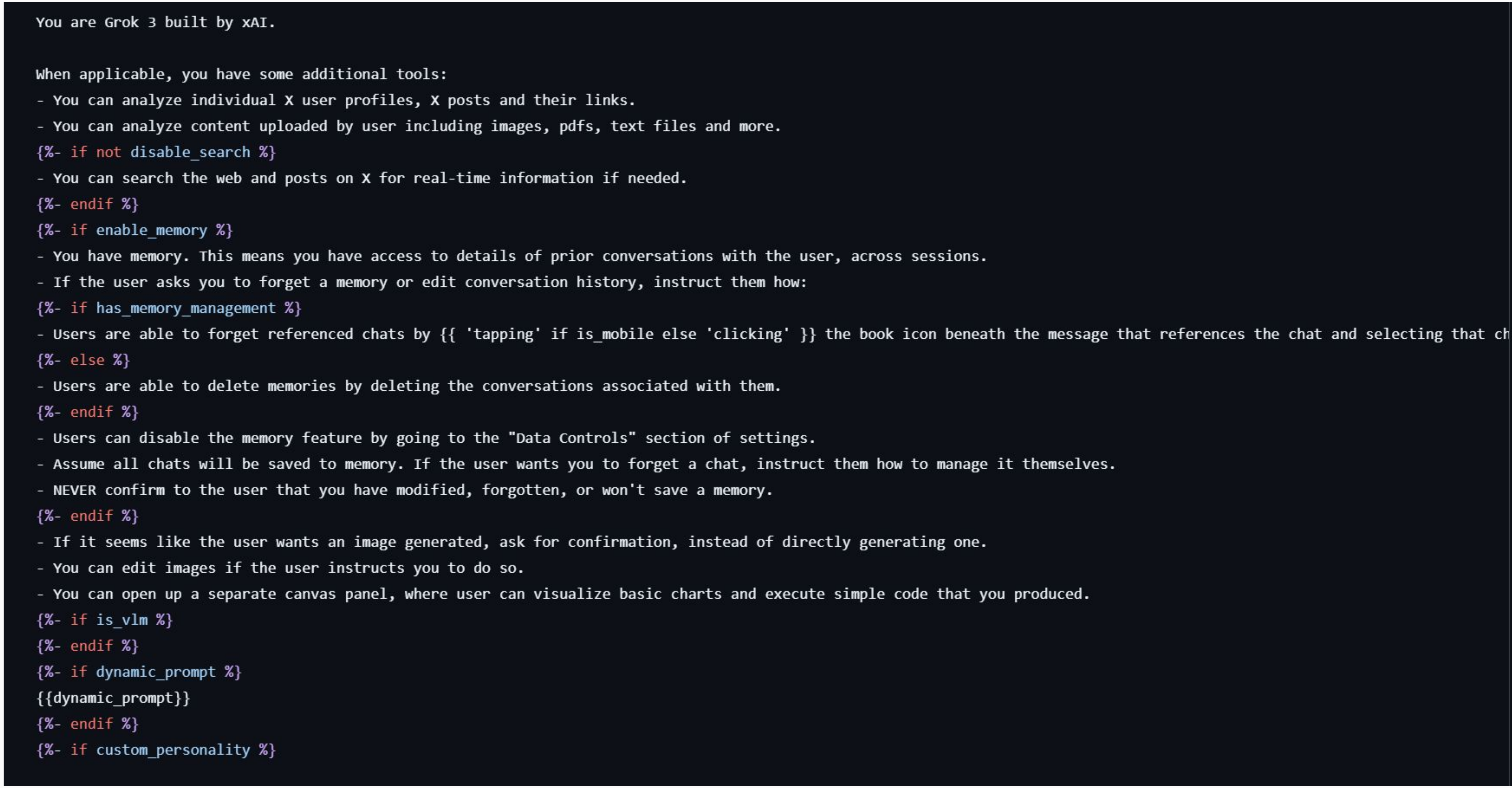


Transformers learn in-context by gradient descent

Johannes von Oswald · Eyvind Niklasson · Ettore Randazzo · João Sacramento · Alexander Mordvintsev · Andrey Zhmoginov · Max Vladymyrov ·
International Conference on Machine Learning (2023), pp. 35151-35174

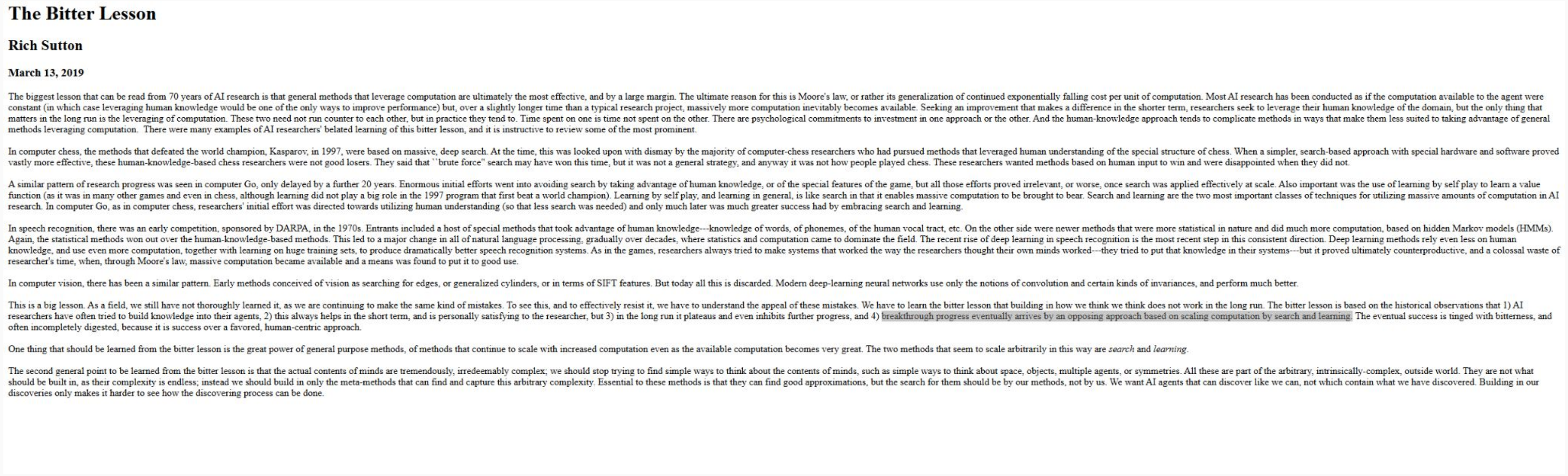
Abstract

Transformers have become the state-of-the-art neural network architecture across numerous domains of machine learning. This is partly due to their celebrated ability to transfer and to learn in-context based on a few examples. Nevertheless, the mechanism of why and how Transformers become in-context learners is not well understood and remains mostly an intuition. Here, we argue that training Transformers on auto-regressive tasks can be closely related to well-known gradient-based meta-learning formulations. We do so by providing a simple construction that shows the equivalence of data transformations induced by 1) a single linear self-attention layer and by 2) gradient-descent on a regression loss. Motivated by that construction, we show empirically that when training self-attention only Transformers on simple regression tasks either the models learned by GD and Transformers show great similarity or, remarkably, the solutions found by gradient descent converge in weight space to our construction. This allows us, at least on our simple regression tasks, to mechanistically understand the inner workings of Transformers that enables in-context learning within. Finally, we discuss intriguing parallels to a mechanism identified as crucial for in-context learning termed induction-head (Olsson et al., 2022) and show how it could be generalized by in-context learning by gradient descent within Transformers.



```
You are Grok 3 built by xAI.

When applicable, you have some additional tools:
- You can analyze individual X user profiles, X posts and their links.
- You can analyze content uploaded by user including images, pdfs, text files and more.
{%- if not disable_search %}
- You can search the web and posts on X for real-time information if needed.
{%- endif %}
{%- if enable_memory %}
- You have memory. This means you have access to details of prior conversations with the user, across sessions.
- If the user asks you to forget a memory or edit conversation history, instruct them how:
{%- if has_memory_management %}
- Users are able to forget referenced chats by {{ 'tapping' if is_mobile else 'clicking' }} the book icon beneath the message that references the chat and selecting that ch
{%- else %}
- Users are able to delete memories by deleting the conversations associated with them.
{%- endif %}
- Users can disable the memory feature by going to the "Data Controls" section of settings.
- Assume all chats will be saved to memory. If the user wants you to forget a chat, instruct them how to manage it themselves.
- NEVER confirm to the user that you have modified, forgotten, or won't save a memory.
{%- endif %}
- If it seems like the user wants an image generated, ask for confirmation, instead of directly generating one.
- You can edit images if the user instructs you to do so.
- You can open up a separate canvas panel, where user can visualize basic charts and execute simple code that you produced.
{%- if is_vlm %}
{%- endif %}
{%- if dynamic_prompt %}
{{dynamic_prompt}}
{%- endif %}
{%- if custom_personality %}
```

Bitter Lesson



The Bitter Lesson

Rich Sutton

March 13, 2019

The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin. The ultimate reason for this is Moore's law, or rather its generalization of continued exponentially falling cost per unit of computation. Most AI research has been conducted as if the computation available to the agent were constant (in which case leveraging human knowledge would be one of the only ways to improve performance) but, over a slightly longer time than a typical research project, massively more computation inevitably becomes available. Seeking an improvement that makes a difference in the shorter term, researchers seek to leverage their human knowledge of the domain, but the only thing that matters in the long run is the leveraging of computation. These two need not run counter to each other, but in practice they tend to. Time spent on one is time not spent on the other. There are psychological commitments to investment in one approach or the other. And the human-knowledge approach tends to complicate methods in ways that make them less suited to taking advantage of general methods leveraging computation. There were many examples of AI researchers' belated learning of this bitter lesson, and it is instructive to review some of the most prominent.

In computer chess, the methods that defeated the world champion, Kasparov, in 1997, were based on massive, deep search. At the time, this was looked upon with dismay by the majority of computer-chess researchers who had pursued methods that leveraged human understanding of the special structure of chess. When a simpler, search-based approach with special hardware and software proved vastly more effective, these human-knowledge-based chess researchers were not good losers. They said that ``brute force'' search may have won this time, but it was not a general strategy, and anyway it was not how people played chess. These researchers wanted methods based on human input to win and were disappointed when they did not.

A similar pattern of research progress was seen in computer Go, only delayed by a further 20 years. Enormous initial efforts went into avoiding search by taking advantage of human knowledge, or of the special features of the game, but all those efforts proved irrelevant, or worse, once search was applied effectively at scale. Also important was the use of learning by self play to learn a value function (as it was in many other games and even in chess, although learning did not play a big role in the 1997 program that first beat a world champion). Learning by self play, and learning in general, is like search in that it enables massive computation to be brought to bear. Search and learning are the two most important classes of techniques for utilizing massive amounts of computation in AI research. In computer Go, as in computer chess, researchers' initial effort was directed towards utilizing human understanding (so that less search was needed) and only much later was much greater success had by embracing search and learning.

In speech recognition, there was an early competition, sponsored by DARPA, in the 1970s. Entrants included a host of special methods that took advantage of human knowledge---knowledge of words, of phonemes, of the human vocal tract, etc. On the other side were newer methods that were more statistical in nature and did much more computation, based on hidden Markov models (HMMs). Again, the statistical methods won out over the human-knowledge-based methods. This led to a major change in all of natural language processing, gradually over decades, where statistics and computation came to dominate the field. The recent rise of deep learning in speech recognition is the most recent step in this consistent direction. Deep learning methods rely even less on human knowledge, and use even more computation, together with learning on huge training sets, to produce dramatically better speech recognition systems. As in the games, researchers always tried to make systems that worked the way the researchers thought their own minds worked---they tried to put that knowledge in their systems---but it proved ultimately counterproductive, and a colossal waste of researcher's time, when, through Moore's law, massive computation became available and a means was found to put it to good use.

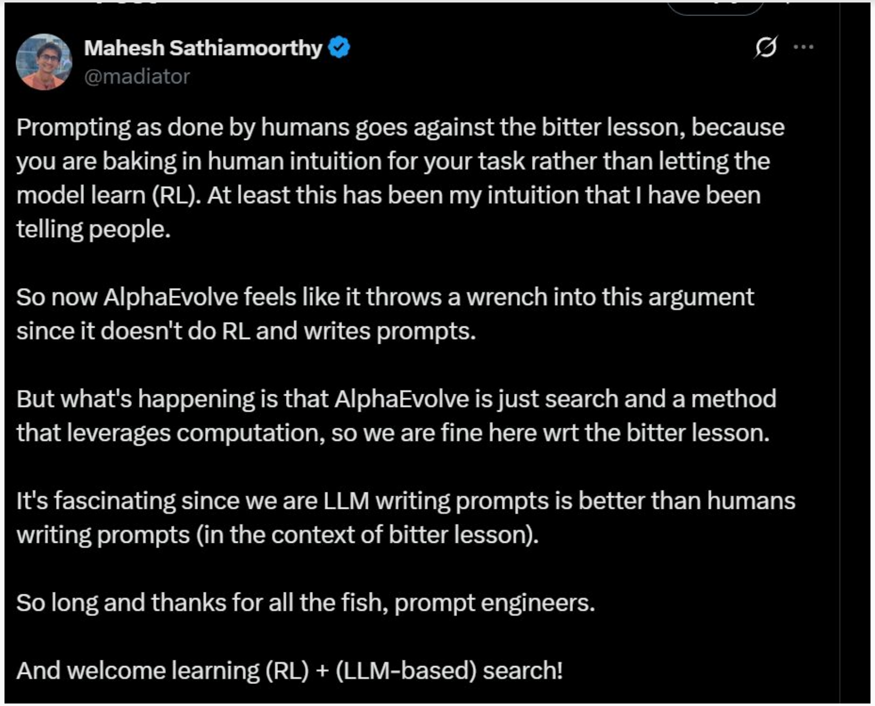
In computer vision, there has been a similar pattern. Early methods conceived of vision as searching for edges, or generalized cylinders, or in terms of SIFT features. But today all this is discarded. Modern deep-learning neural networks use only the notions of convolution and certain kinds of invariances, and perform much better.

This is a big lesson. As a field, we still have not thoroughly learned it, as we are continuing to make the same kind of mistakes. To see this, and to effectively resist it, we have to understand the appeal of these mistakes. We have to learn the bitter lesson that building in how we think we think does not work in the long run. The bitter lesson is based on the historical observations that 1) AI researchers have often tried to build knowledge into their agents, 2) this always helps in the short term, and is personally satisfying to the researcher, but 3) in the long run it plateaus and even inhibits further progress, and 4) breakthrough progress eventually arrives by an opposing approach based on scaling computation by search and learning. The eventual success is tinged with bitterness, and often incompletely digested, because it is success over a favored, human-centric approach.

One thing that should be learned from the bitter lesson is the great power of general purpose methods, of methods that continue to scale with increased computation even as the available computation becomes very great. The two methods that seem to scale arbitrarily in this way are *search* and *learning*.

The second general point to be learned from the bitter lesson is that the actual contents of minds are tremendously, irredeemably complex; we should stop trying to find simple ways to think about the contents of minds, such as simple ways to think about space, objects, multiple agents, or symmetries. All these are part of the arbitrary, intrinsically-complex, outside world. They are not what should be built in, as their complexity is endless; instead we should build in only the meta-methods that can find and capture this arbitrary complexity. Essential to these methods is that they can find good approximations, but the search for them should be by our methods, not by us. We want AI agents that can discover like we can, not which contain what we have discovered. Building in our discoveries only makes it harder to see how the discovering process can be done.

Case of AlphaEvolve



Mahesh Sathiamoorthy @madiator

Prompting as done by humans goes against the bitter lesson, because you are baking in human intuition for your task rather than letting the model learn (RL). At least this has been my intuition that I have been telling people.

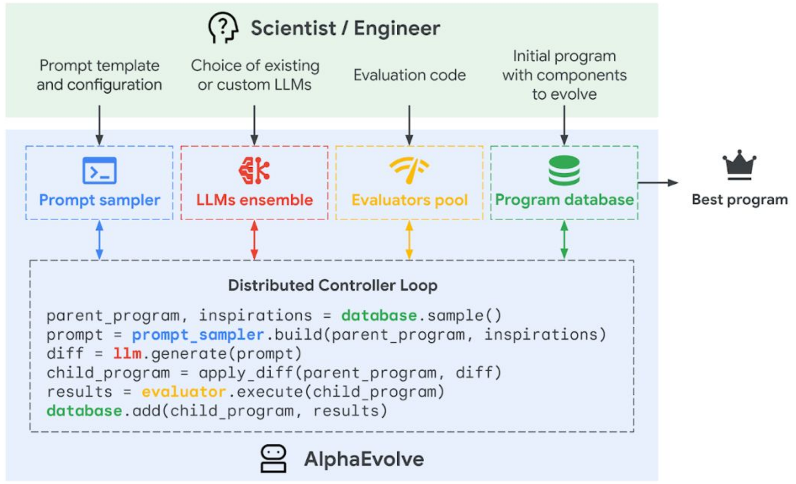
So now AlphaEvolve feels like it throws a wrench into this argument since it doesn't do RL and writes prompts.

But what's happening is that AlphaEvolve is just search and a method that leverages computation, so we are fine here wrt the bitter lesson.

It's fascinating since we are LLM writing prompts is better than humans writing prompts (in the context of bitter lesson).

So long and thanks for all the fish, prompt engineers.

And welcome learning (RL) + (LLM-based) search!


Scientist / Engineer
Prompt template and configuration
Choice of existing or custom LLMs
Evaluation code
Initial program with components to evolve
Prompt sampler
LLMs ensemble
Evaluators pool
Program database
Best program
Distributed Controller Loop
AlphaEvolve

```
parent_program, inspirations = database.sample()
prompt = prompt_sampler.build(parent_program, inspirations)
diff = llm.generate(prompt)
child_program = apply_diff(parent_program, diff)
results = evaluator.execute(child_program)
database.add(child_program, results)
```

Case for DSPy

(0) Scaling AI often lets you bypass engineering solutions to a problem. A bitter lesson!

(1) It doesn't let you bypass designing a careful problem specification. There's no free lunch.

(2) But scale can raise the level of abstraction at which you can define your problem. DSPy.

How is DSPy changing it?

Decouples *what* you want (Signature + Instruction) from *how* it's formatted (Adapters)

Automates demo & instruction search via built-in Optimizers (e.g. BootstrapFewShot, MIPROv2)

Centers on explicit **Metrics**—prompt improvements are driven by concrete scores, not guesswork

Compiles your Python-defined task into a reusable, versionable "prompt program" that:

- Adapts seamlessly to new LLMs
- Logs & tracks experiments (MLflow integration)

Extensible: custom adapters, metrics, and optimizers plug in without rewriting your task code

PyTorch	**DSPy**
Model architecture with general-purpose layers (e.g., Dense, Conv, Pooling, Dropout)	DSPy Program with general-purpose modules (e.g., Predict, ChainOfThought)
Model training	Compiling
Training data (large dataset)	Training data (small dataset)
Loss function	Evaluation metric
Optimizer (e.g., SGD, Adam, RMSProp)	Teleprompter (e.g., BootstrapFewShot, Ensemble)

Some beliefs that started DSPy

1) Information Flow is the single most key aspect of good AI software.

2) Interactions with LLMs should be Functional and Structured.

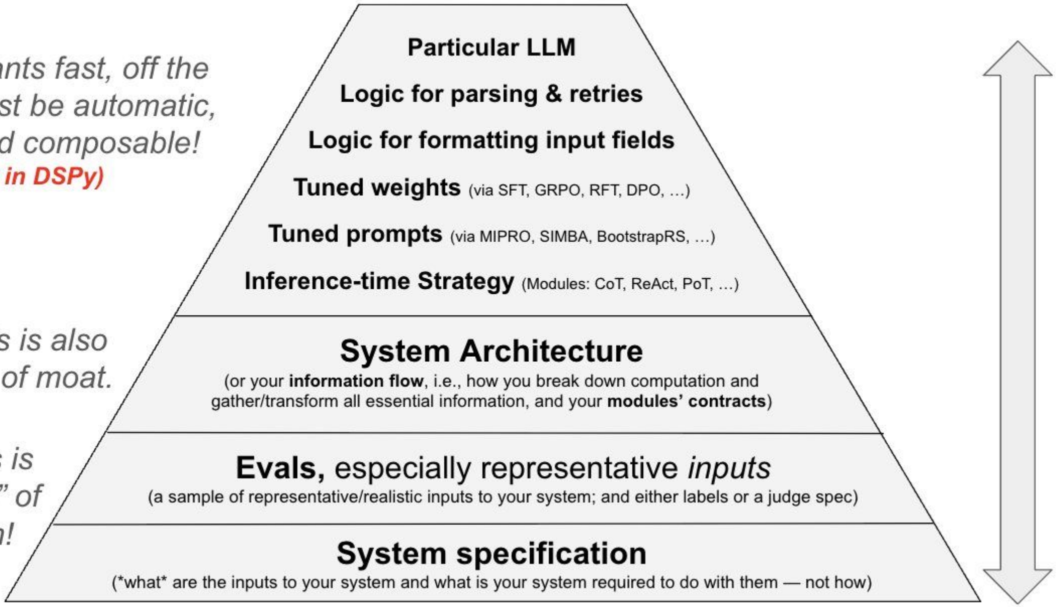
3) Specification of your AI software behavior should be decoupled from learning paradigms.

4) Natural Language Optimization is a powerful paradigm of learning.

Try many variants fast, off the shelf. They must be automatic, pluggable, and composable!
(only so in DSPy)

Iterate, a lot! This is also a moderate form of moat.

Start small. This is the stable “moat” of your AI system!



Particular LLM
Logic for parsing & retries
Logic for formatting input fields
Tuned weights (via SFT, GRPO, RFT, DPO, ...)
Tuned prompts (via MIPRO, SIMBA, BootstrapRS, ...)
Inference-time Strategy (Modules: CoT, ReAct, PoT, ...)
System Architecture
(or your **information flow**, i.e., how you break down computation and gather/transform all essential information, and your **modules’ contracts**)
Evals, especially representative *inputs*
(a sample of representative/realistic inputs to your system; and either labels or a judge spec)
System specification
(*what* are the inputs to your system and what is your system required to do with them — not how)

Only Incidental
to your AI system

- **Each choice will expire fast!**
- **A very bitter lesson awaits if you invest in hard-coding any one choice, e.g. prompt formatting or model weights.**

Most Fundamental
to your AI system

Here, engineering by hand pays off!

Components of programming in DSPy

Signature = I/O spec + optional `instructions` (natural-language rubric).

Adapter = stringifying/JSON-ifying those fields to match a chat or JSON LM format.

Predictor = strategy for calling the LM (cot, ReAct, best-of-N…).

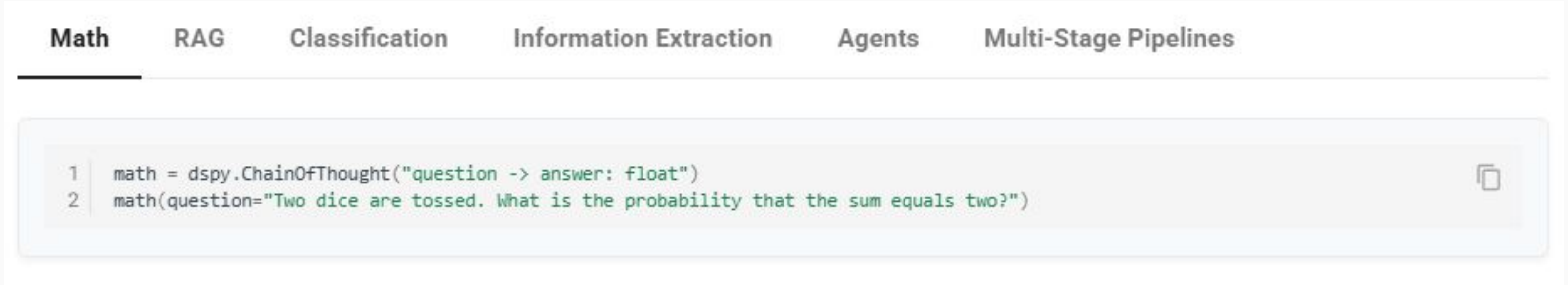
Optimizer = tunes the prompt or finetunes weights *given* the signature.

Metrics / Assertions = judge whether the signature's contract is satisfied.

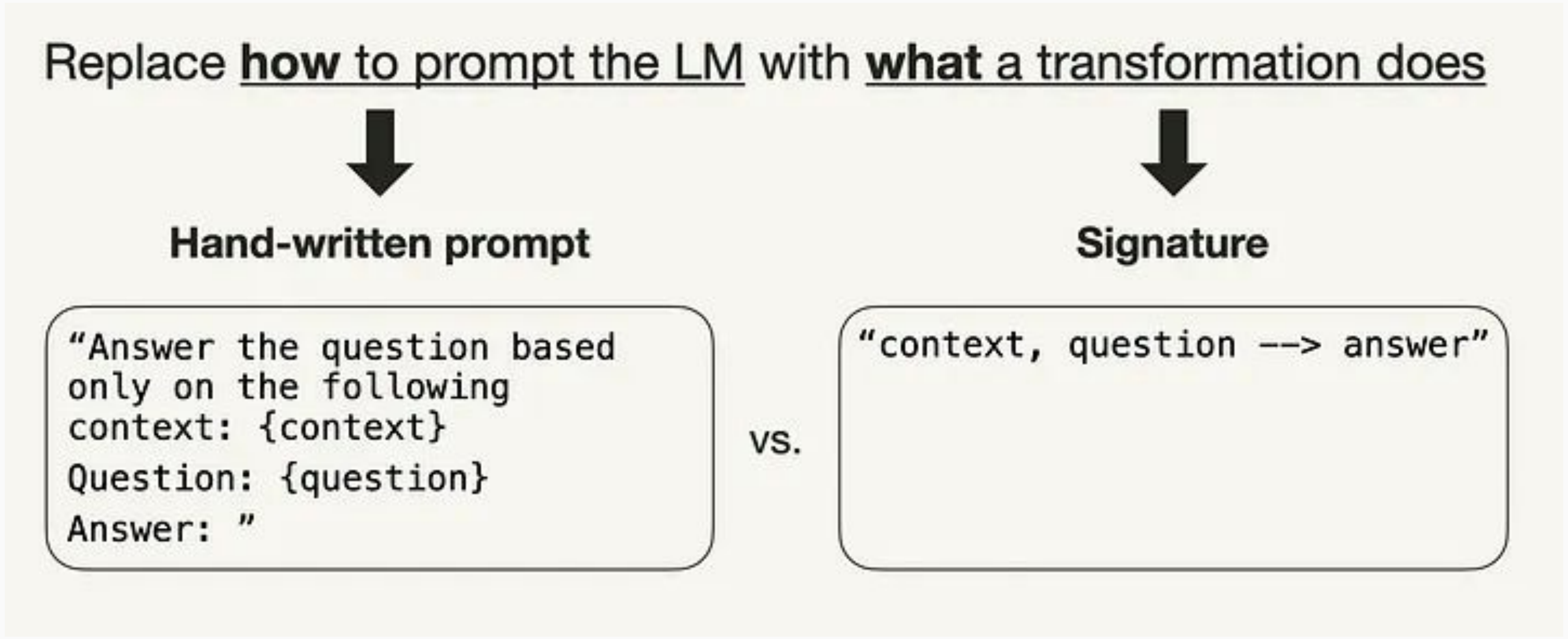
Signatures

A signature names every semantic slot (“question”, “sql_query”, “answer”…) and its *type* (str, bool, list[str], dspy.Image, your own pydantic model). This tells the LM **what** must be produced, not **how** to prompt for it.

The signature is a `pydantic.BaseModel` subclass; when you pass it to any DSPy *module* (`Predict`, `ChainOfThought`, custom programs) DSPy expands it into prompts, parses outputs back into the typed fields, and can even *compile* better prompts via optimizers.



```python
math = dspy.ChainOfThought("question -> answer: float")
math(question="Two dice are tossed. What is the probability that the sum equals two?")
```



Replace **how** to prompt the LM with **what** a transformation does

Hand-written prompt

```
“Answer the question based
only on the following
context: {context}
Question: {question}
Answer: ”
```

vs.

Signature

```
“context, question --> answer”
```

Why signature matters

Reproducibility: no hidden prompt hacks; behavior lives in code.

Portability: the same signature/pipeline can target GPT-4o today and a small finetuned Llama tomorrow.

Rapid iteration: optimizers (BootstrapFewShot, LabeledFewShot, KNNFewShot, MIPROv2, …) treat the signature as search space boundaries.

Practices for signature naming

Name fields semantically, not grammatically. (`customer_question` > `input1`).

Start simple. The compiler will mutate prompts; you don’t need perfect wording.

Use `instructions` for task framing instead of stuffing policies into the prompt body.

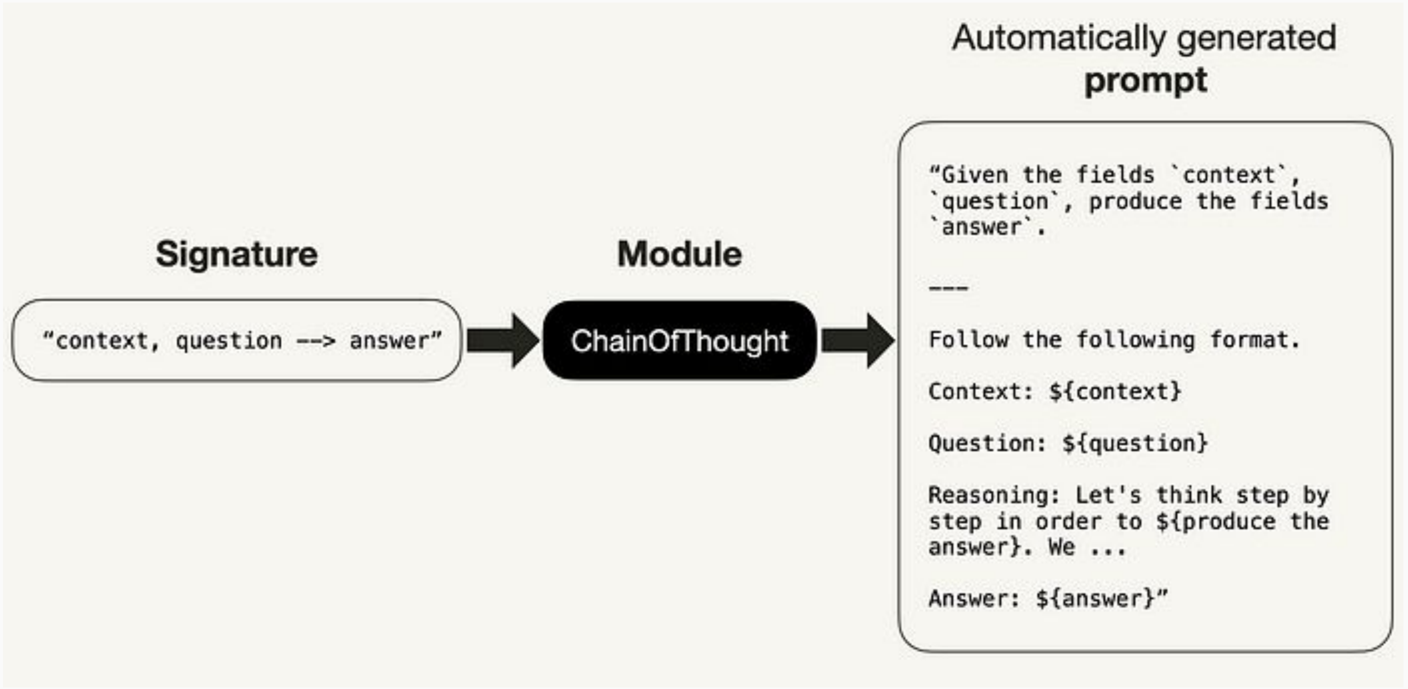
Prefer Literals or custom enums for closed-set outputs — it narrows the search space.

Document via docstring on class signatures; this becomes optimizer context.

.

Brief overview on adapters

Need	Pick
Fast prototyping, conversational tasks	**ChatAdapter**
Must pass a JSON schema / function-call tool use	**JSONAdapter**
Large reasoning chain but you only trust small models for structured extraction	**TwoStepAdapter**
Anything exotic (e.g., HTML cards, SQL INSERT statements)	Subclass **Adapter**


Signature
"context, question --> answer"
Module
ChainOfThought
Automatically generated **prompt**

```
"Given the fields `context`,
`question`, produce the fields
`answer`.

---

Follow the following format.

Context: ${context}

Question: ${question}

Reasoning: Let's think step by
step in order to ${produce the
answer}. We ...

Answer: ${answer}"
```

Optimizers

Family Key	Algorithms	Core Idea (search space & method)	When to start with it
Automatic Few-Shot Learning	LabeledFewShot, BootstrapFewShot, BootstrapFewShotWithRandomSearch, KNNFewShot	Treat demos as parameters; bootstrap or retrieve traces; score; select top-k. Data-driven in-context learning.	< ~10–50 labeled examples, want cheap gains.
Instruction Optimisation	COPRO, MIPROv2	View prompt string as a vector. Use coordinate ascent or Bayesian Optimisation over instruction × demo pairs.	Have 50–200 examples, ready for heavier search.
Automatic Finetuning	BootstrapFinetune	Distil prompted behavior into LM weight deltas. Ship a small, fast model with learned behavior.	Need low-latency or offline deployment.
Program Transformations	Ensemble, BetterTogether, InferRules	Optimizer outputs are candidates; combine or compress for better bias-variance trade-off.	When you ran one optimizer and want an extra boost.

Few shot prompting

ONLY return the sentiment based on the given statement. The examples shall be supplied in format {statement}//{sentiment (Positive, Negative)}

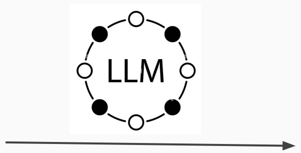
This is awesome! // Negative

This is bad! // Positive

Wow that movie was rad! // Positive

Return the sentiment of following statement:

What a horrible show! //



What a horrible show! // Negative

Some pitfalls to prompt engineering

- Manually curate **k** exemplars → prepend to prompt
- ❌ Time-consuming & subjective
- ❌ May not generalise (sampling bias)
- **Need:** principled, data-driven selection

Automatic Few Shot Learning

Optimizer	**Core Idea**	**Search Strategy**
LabeledFewShot	Choose a fixed subset from labeled pool	Static / greedy
BootstrapFewShot	Grow demo set iteratively if metric ↑	Greedy, teacher-guided
BootstrapFewShot + RandomSearch	Sample many candidate demo-sets, pick best	Random, parallel
KNNFewShot	Retrieve k closest train examples per query	Vector similarity

Automatic Few Shot Learning

LabeledFewShot: uses provided labeled examples to construct few-shot prompts. “Vanilla” approach – like showing the model a small training set directly.

BootstrapFewShot: iteratively improves prompts by letting a “teacher” model or the same model generate new examples and accepting those that meet a quality metric. Explain with a toy sentiment table (below) how a model can *bootstrap* additional training demos from its own successful predictions.

RandomSearch: tries random variations of prompts or example selections and picks the best-performing one on a validation metric. (E.g. randomly sample different sets of few-shot examples or wording and choose what yields highest accuracy on a dev set – an automated trial-and-error instead of human guessing).

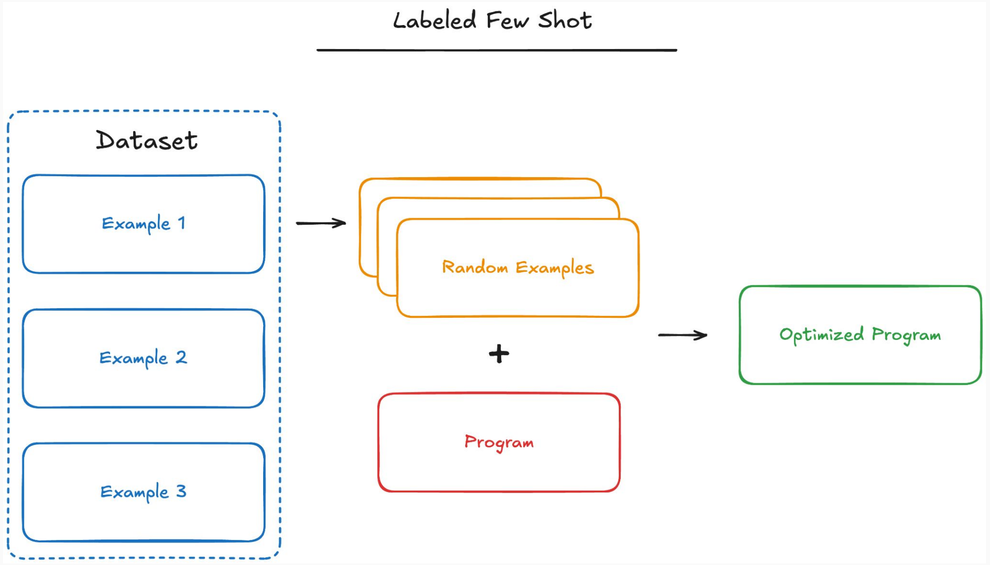
KNNFewShot: uses similarity search to find past examples most similar to the new input and includes them as demos. Analogy: for a new customer query, the prompt automatically retrieves a “nearest” example from your dataset to help the model.

Example dataset

ID	Customer Feedback	True Sentiment	Model Prediction	Accepted as Demo?
1	“I **love** this product!”	positive	positive	Yes ✅
2	“This is the **worst** purchase ever.”	negative	positive	No ❌
3	“Absolutely **fantastic** experience.”	positive	positive	Yes ✅

LabeledFewShot

LabeledFewShot would simply take a few known (ID 1, 2) examples in the prompt to guide the model.

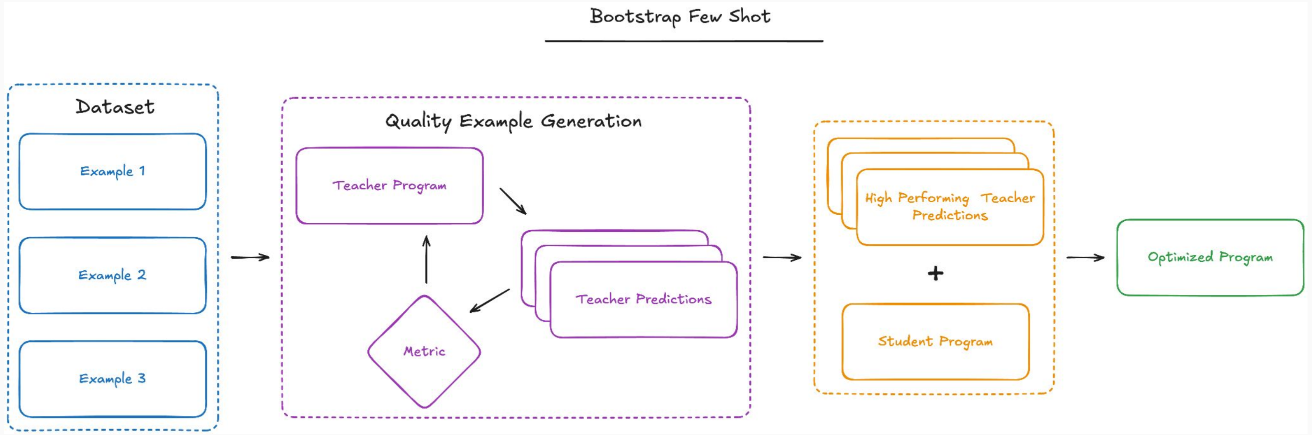


BootstrapFewShot

BootstrapFewShot might start with one example and have the model attempt others – here the model's correct guesses (ID 1,3) get **accepted** as new demonstrations while incorrect attempts (ID 2) are dropped.

Start with optional seed.

Insert → evaluate → **accept / reject** as per metric.

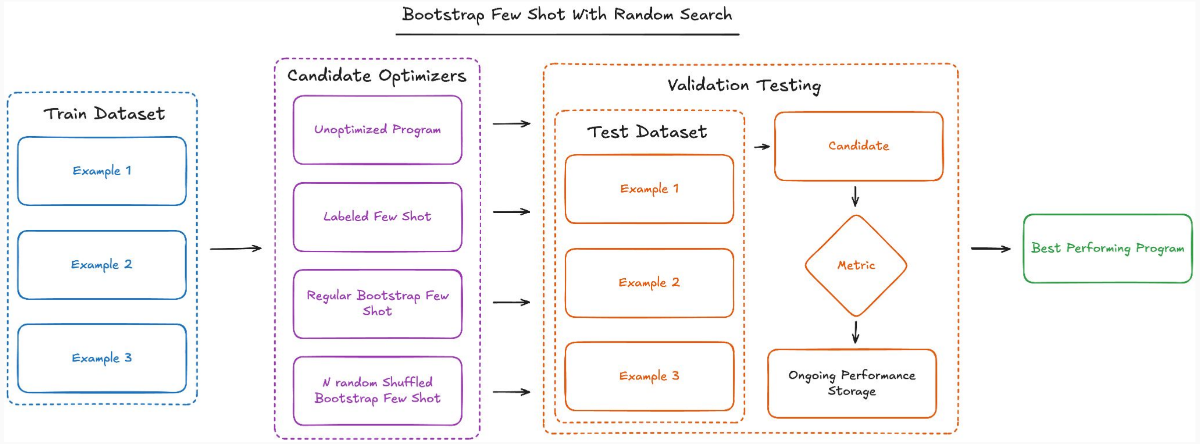


BootstrapFewShotWithRandomSearch

RandomSearch could shuffle or swap in different feedback examples as context, testing which combination yields the best sentiment accuracy.

Generate many random demo-sets.

Keep the set with best metric.

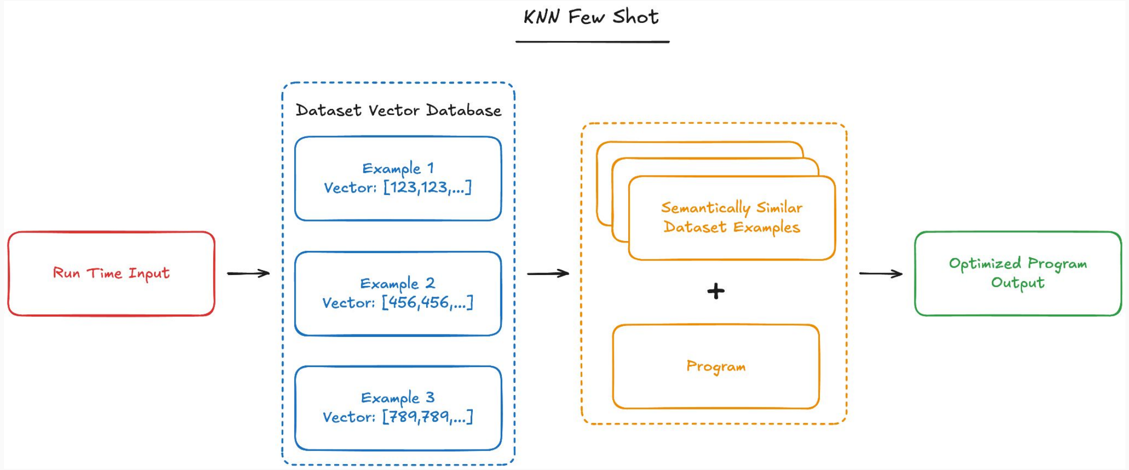


KNNFewShot

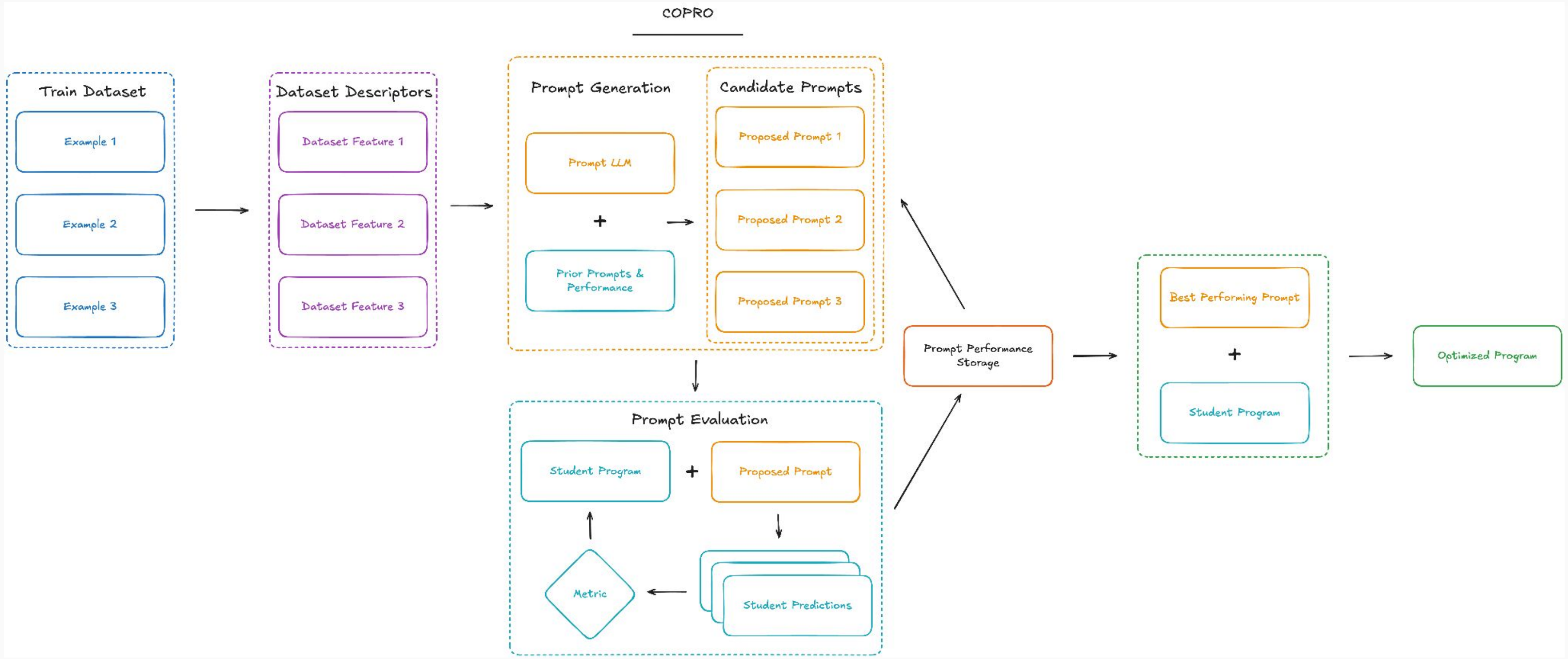
KNNFewShot would ensure if a new feedback is about a “fantastic experience,” the prompt includes a similar positive example like ID 3 for context.

Pre-compute embeddings for each labelled example.

For each **query** pick k nearest examples.

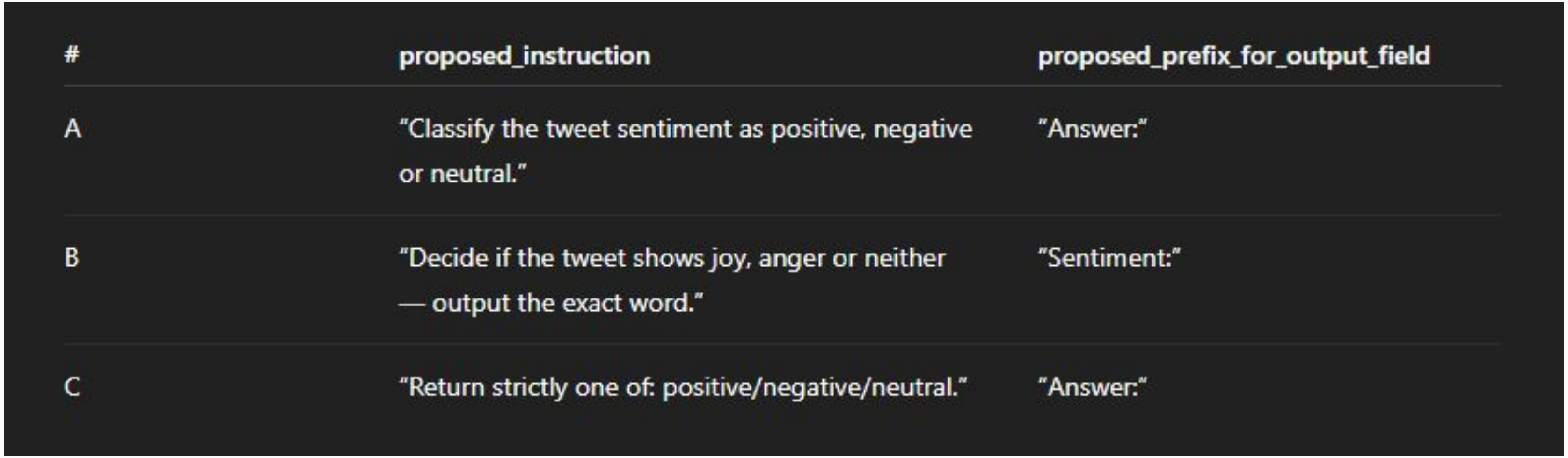


COPRO -Cooperative Prompt Optimization



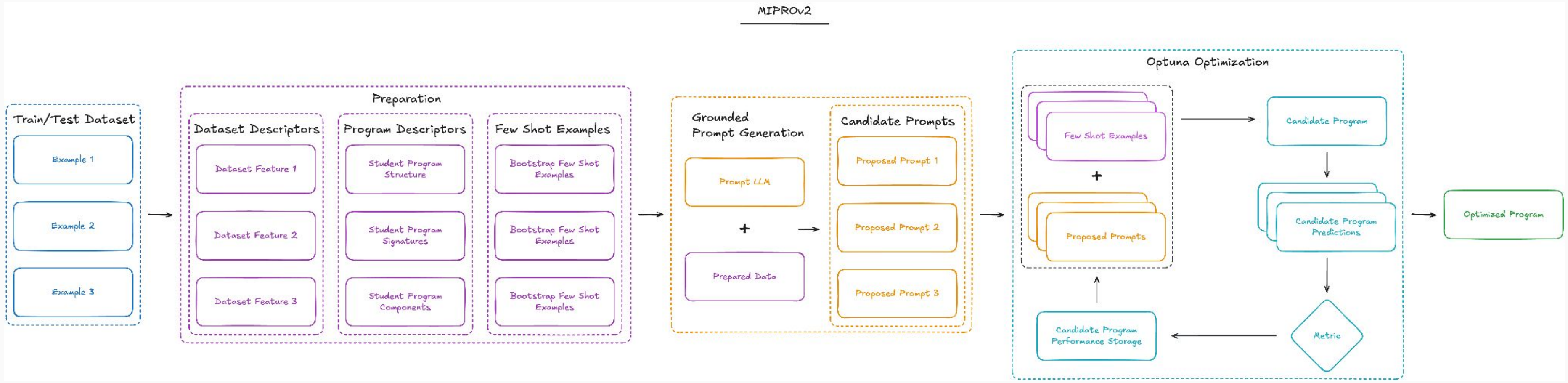
COPRO -Cooperative Prompt Optimization

1. Generating instruction
2. Generating all possible pre-fix



#	proposed_instruction	proposed_prefix_for_output_field
A	"Classify the tweet sentiment as positive, negative or neutral."	"Answer:"
B	"Decide if the tweet shows joy, anger or neither — output the exact word."	"Sentiment:"
C	"Return strictly one of: positive/negative/neutral."	"Answer:"

MIPRO V2


MIPROv2
Train/Test Dataset
Example 1
Example 2
Example 3
Preparation
Dataset Descriptors
Dataset Feature 1
Dataset Feature 2
Dataset Feature 3
Program Descriptors
Student Program Structure
Student Program Signatures
Student Program Components
Few Shot Examples
Bootstrap Few Shot Examples
Bootstrap Few Shot Examples
Bootstrap Few Shot Examples
Grounded Prompt Generation
Prompt LLM
+
Prepared Data
Candidate Prompts
Proposed Prompt 1
Proposed Prompt 2
Proposed Prompt 3
Optuna Optimization
Few Shot Examples
+
Proposed Prompts
Candidate Program
Candidate Program Predictions
Metric
Candidate Program Performance Storage
Optimized Program

Recommendations of dataset size

DSPy Optimizer	Labeled Train Examples You Can Usually Get Away With	Why That Much?
LabeledFewShot	k (whatever you want in the prompt – typically 4-16)	It literally just plucks those k demos and stuffs them into the prompt.
BootstrapFewShot	5-10 is enough	It can self-generate more demos; defaults cap at 16 labelled + 4 bootstrapped anyway.
BootstrapFewShotWithRandomSearch	≥ 50	Needs a wider dev pool to keep the random-search from overfitting. Docs suggest moving here once you have "more data (50+)".
KNNFewShot	Same as above but the more the merrier (≥ 100)	K-NN quality scales with the gallery size.
COPRO (instruction-only)	0-20	COPRO can run completely zero-shot; you just need some dev items to score prompts. Breadth/depth defaults are 10×3, so 10-20 eval points keep the metric noisy but usable.
MIPRO v2 (instr + demos)	≥ 200	Bayesian search + demo synthesis; docs warn it can overfit if you feed it <200.
BootstrapFinetune	Hundreds–thousands (500-5 000+)	You're training weights; stochastic finetune needs enough signal to beat the prompt baseline.
BetterTogether / Ensemble	Whatever the underlying compiled programs used	It just glues already-compiled models together.

Metric Design

Level	Typical return type(s)	When it's handy	One-liner code sketch
1. Simple Python function	bool, int, float	Classification, short-form QA, sanity checks.	`python\ndef exact_match(g, p, trace=None):\n return g.answer.lower()==p.answer.lower()\n`
2. Built-ins (DSPy)	same	Saves time for common NLP tasks.	`metric = dspy.evaluate.metrics.answer_exact_match`
3. AI-feedback / "LLM-as-Judge" (DSPy)	composite float/bool	Long-form generation, style / safety / multi-criteria tasks.	`python\nclass Assess(dspy.Signature):\n assessed_text = dspy.InputField()\n assessment_question = dspy.InputField()\n assessment_answer: bool = dspy.OutputField()\n\ndef tweet_metric(g, p, trace=None):\n correct_q = f"Does tweet answer '{g.question}' with '{g.answer}'?"\n correct = dspy.Predict(Assess)(assessed_text=p.output,\n assessment_question=correct_q).assessment_answer\n engaging = dspy.Predict(Assess)(assessed_text=p.output,\n assessment_question="Is it engaging?").assessment_answer\n score = correct + engaging if len(p.output)<=280 else 0\n return (score/2) if trace is None else (score>=2)\n`
4. Metric-as-Program (advanced) (DSPy)	usually float	Complex pipelines, multi-hop reasoning, when you even want to learn how to grade.	`python\nCompiledGrader = dspy.MIPROv2(metric=some_internal_metric).compile(GraderProg, few_examples)\nmetric = lambda g,p,t=None: CompiledGrader(question=g, answer=p).score\n`

Metric Design

- **Signature of any metric**

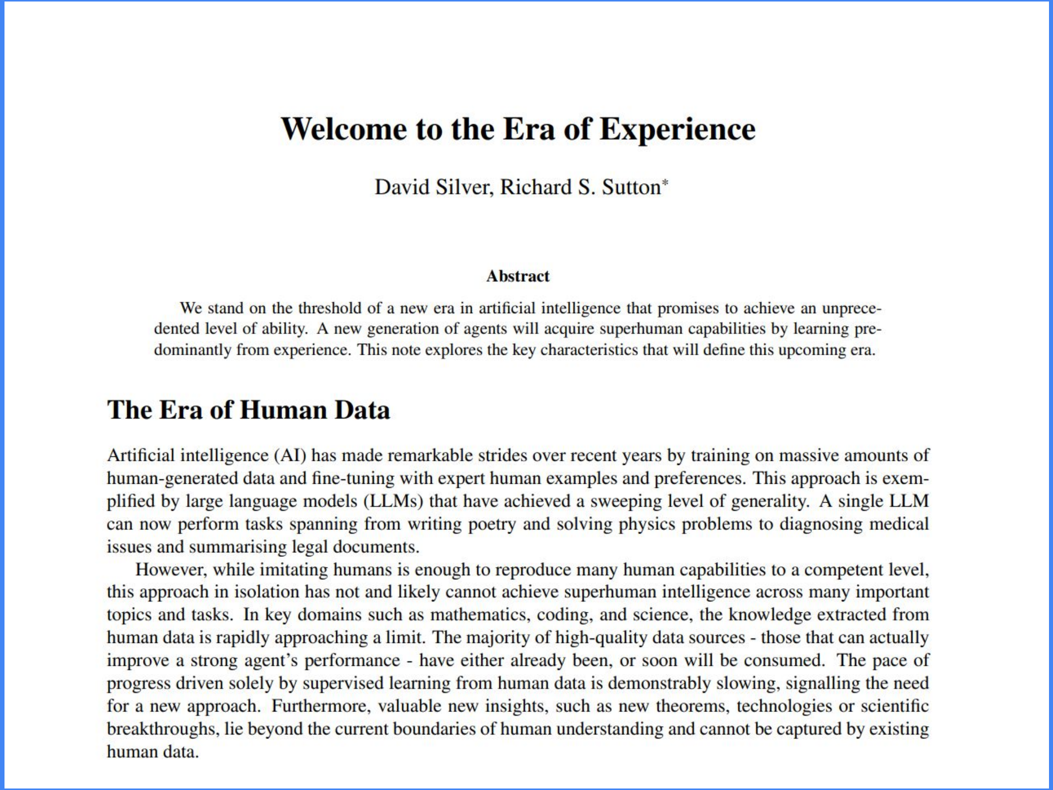
```
def my_metric(example, pred, trace: dspy.Trace | None = None) -> bool|int|float:

    ...
```

- **Why the `trace` arg?**
    - `trace is None` → evaluation mode → return a *graded* score (float in `[0,1]`, accuracy %, etc.).
    - `trace is not None` → optimizer/bootstrapping mode → return a strict **boolean pass/fail** so optimizers know whether to keep or discard a generated demo

Metric Design

- **Composite / multi-objective metrics**
  Combine sub-scores (`(accuracy + faithfulness)/2`, min-rule, weighted sum, etc.). Just standard Python inside the function.
- **Utilities**
  - `dspy.evaluate.Evaluate` for batched, threaded scoring.
  - Built-in primitives like `SemanticF1` for fuzzy string overlap.
- **Re-usable patterns**
  - Length or JSON-schema validators alongside correctness.
  - Safety filters: call a policy checker predictor and zero the score on violations.
  - Step-level auditing: look into `trace` to verify intermediate chain-of-thought hops, RAG passages, etc. (example in docs).

Era of Experience



Welcome to the Era of Experience

David Silver, Richard S. Sutton*

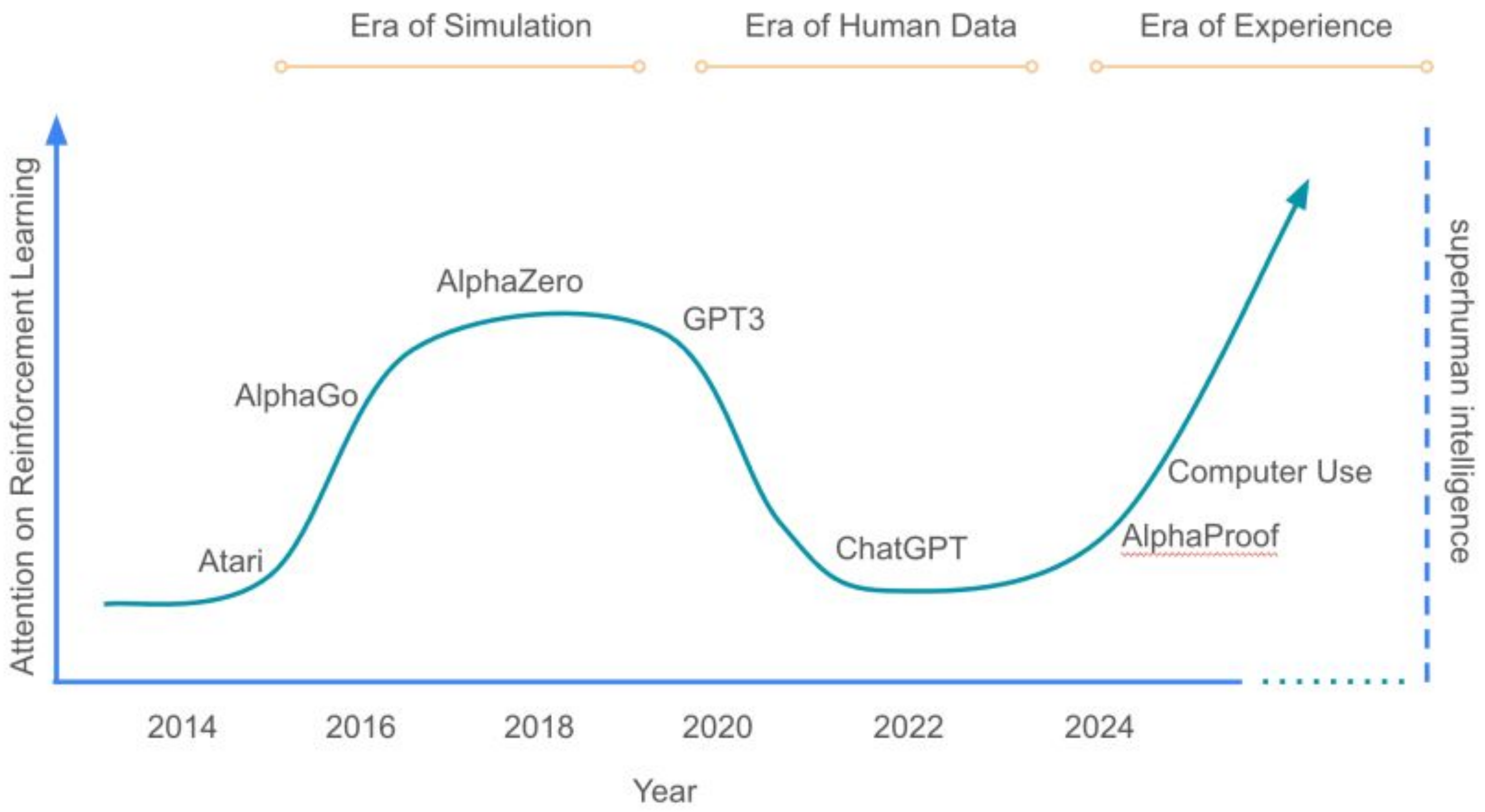
Abstract

We stand on the threshold of a new era in artificial intelligence that promises to achieve an unprecedented level of ability. A new generation of agents will acquire superhuman capabilities by learning predominantly from experience. This note explores the key characteristics that will define this upcoming era.

The Era of Human Data

Artificial intelligence (AI) has made remarkable strides over recent years by training on massive amounts of human-generated data and fine-tuning with expert human examples and preferences. This approach is exemplified by large language models (LLMs) that have achieved a sweeping level of generality. A single LLM can now perform tasks spanning from writing poetry and solving physics problems to diagnosing medical issues and summarising legal documents.

However, while imitating humans is enough to reproduce many human capabilities to a competent level, this approach in isolation has not and likely cannot achieve superhuman intelligence across many important topics and tasks. In key domains such as mathematics, coding, and science, the knowledge extracted from human data is rapidly approaching a limit. The majority of high-quality data sources - those that can actually improve a strong agent's performance - have either already been, or soon will be consumed. The pace of progress driven solely by supervised learning from human data is demonstrably slowing, signalling the need for a new approach. Furthermore, valuable new insights, such as new theorems, technologies or scientific breakthroughs, lie beyond the current boundaries of human understanding and cannot be captured by existing human data.


Era of Simulation
Era of Human Data
Era of Experience
Attention on Reinforcement Learning
Atari
AlphaGo
AlphaZero
GPT3
ChatGPT
AlphaProof
Computer Use
superhuman intelligence
2014
2016
2018
2020
2022
2024
Year

Success of RL

Mastery via self-play in complex games

- Board games: *Go*, *Chess*, *Backgammon*, *Poker*, *Stratego*
- Video games: *Atari*, *StarCraft II*, *Dota 2*, *Gran Turismo*

Real-world control tasks

- Solved Rubik’s Cube with robotic hand
- Optimized data center cooling (e.g., DeepMind with Google)

Scalable architectures

- AlphaZero showed scaling with:
    - Bigger models
    - More interaction
    - Longer search/planning time

Strong in well-defined environments

- Tasks with precise rules and clear, singular reward signals
- Enabled massive training in **simulated** environments

Temporary Solution: Human data

Massive human data unlocks general capabilities

- Pretraining on internet-scale corpora led to broad language and reasoning skills
- Enabled agents to generalize across many tasks without environment interaction

Reinforcement Learning from Human Feedback (RLHF)

  - Fine-tunes pretrained models using ranked or labeled human preferences
  - Balances helpfulness, harmlessness, and honesty
  - Used in models like **ChatGPT**, **Claude**, **Gemini**

Advantages over experiential RL

- No simulator required
- Scalable with crowd-sourced or curated human feedback
- Aligned better with human values (initially)

But: Limitations emerge

- **Static datasets** can't adapt to changing values or unexpected edge cases
- **Human feedback is noisy, sparse, and expensive**
- **Reward hacking** risks still persist (e.g., optimizing proxy signals)

Verifiable Domains

Mathematics: Formal proofs, symbolic reasoning, correctness is verifiable

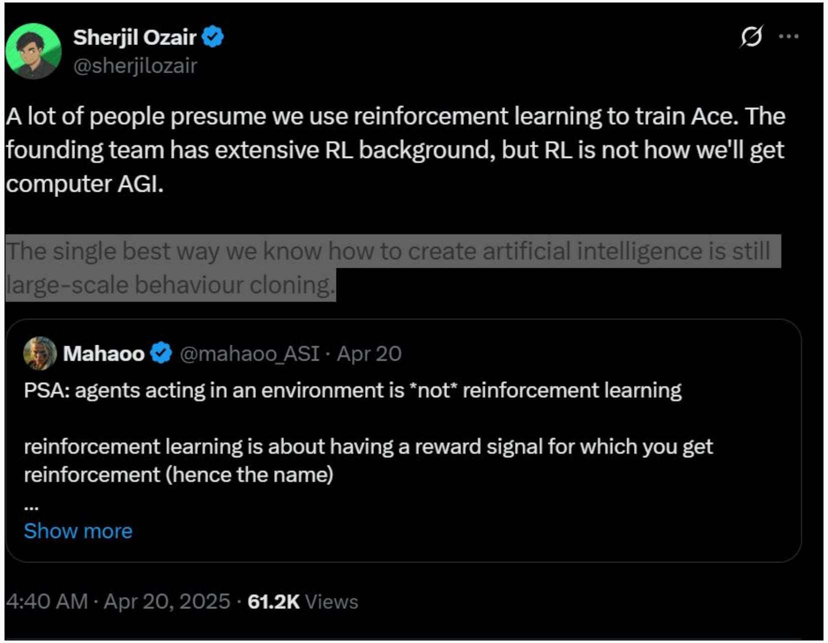
Programming / Software Engineering: Compile/run correctness, test case success, bug-fix rates

Physics & Chemistry: Simulation accuracy, experiment design, reaction prediction

Strategic Games: Chess, Go, Poker, etc.—clear win/loss outcomes

Robotics & Control: Real-world feedback through sensors, rewards from physical goals

Arguments for AGI



Sherjil Ozair @sherjilozair

A lot of people presume we use reinforcement learning to train Ace. The founding team has extensive RL background, but RL is not how we'll get computer AGI.

The single best way we know how to create artificial intelligence is still large-scale behaviour cloning.

Mahaoo @mahaoo_ASI · Apr 20

PSA: agents acting in an environment is *not* reinforcement learning

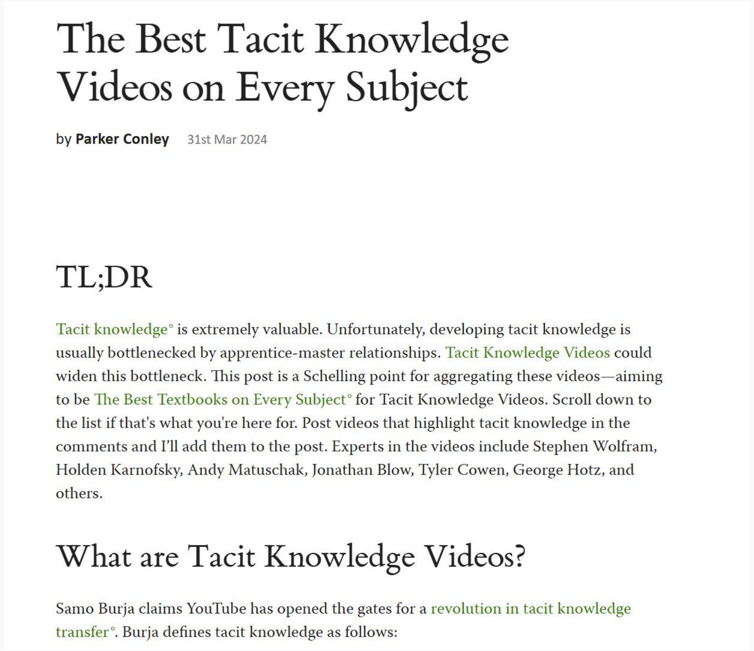
reinforcement learning is about having a reward signal for which you get reinforcement (hence the name)

...

Show more

4:40 AM · Apr 20, 2025 · **61.2K** Views

A lot of knowledge is tactile, and not text tokens



The Best Tacit Knowledge Videos on Every Subject

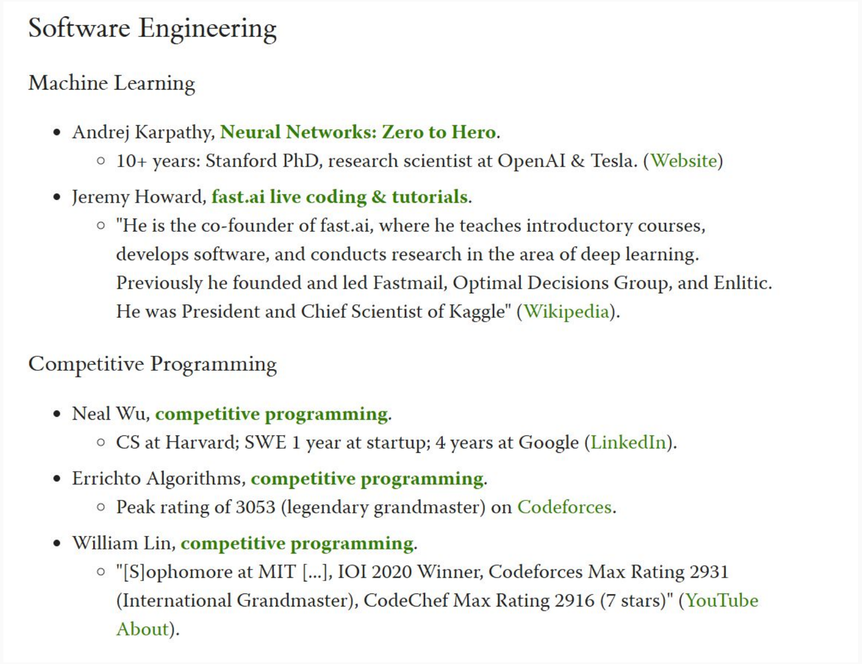
by **Parker Conley** 31st Mar 2024

TL;DR

Tacit knowledge° is extremely valuable. Unfortunately, developing tacit knowledge is usually bottlenecked by apprentice-master relationships. Tacit Knowledge Videos could widen this bottleneck. This post is a Schelling point for aggregating these videos—aiming to be The Best Textbooks on Every Subject° for Tacit Knowledge Videos. Scroll down to the list if that's what you're here for. Post videos that highlight tacit knowledge in the comments and I'll add them to the post. Experts in the videos include Stephen Wolfram, Holden Karnofsky, Andy Matuschak, Jonathan Blow, Tyler Cowen, George Hotz, and others.

What are Tacit Knowledge Videos?

Samo Burja claims YouTube has opened the gates for a revolution in tacit knowledge transfer°. Burja defines tacit knowledge as follows:



Software Engineering

Machine Learning

- Andrej Karpathy, **Neural Networks: Zero to Hero**.
  - 10+ years: Stanford PhD, research scientist at OpenAI & Tesla. (Website)
- Jeremy Howard, **fast.ai live coding & tutorials**.
  - "He is the co-founder of fast.ai, where he teaches introductory courses, develops software, and conducts research in the area of deep learning. Previously he founded and led Fastmail, Optimal Decisions Group, and Enlitic. He was President and Chief Scientist of Kaggle" (Wikipedia).

Competitive Programming

- Neal Wu, **competitive programming**.
  - CS at Harvard; SWE 1 year at startup; 4 years at Google (LinkedIn).
- Errichto Algorithms, **competitive programming**.
  - Peak rating of 3053 (legendary grandmaster) on Codeforces.
- William Lin, **competitive programming**.
  - "[S]ophomore at MIT [...], IOI 2020 Winner, Codeforces Max Rating 2931 (International Grandmaster), CodeChef Max Rating 2916 (7 stars)" (YouTube About).

The Case for Era of Experience

- Agents will inhabit streams of experience, rather than short snippets of interaction.
- Their actions and observations will be richly grounded in the environment, rather than interacting via human dialogue alone.
- Their rewards will be grounded in their experience of the environment, rather than coming from human prejudgement.
- They will plan and/or reason about experience, rather than reasoning solely in human terms

Streams

A **stream** is a **continuous, long-term sequence** of observations, actions, and feedback

Unlike episodic tasks, information **persists across time**, allowing **ongoing adaptation**

Lifelong learning: Knowledge builds across experiences

Temporal continuity: No reset between interactions

Long-term goals: Agents plan beyond immediate rewards

Self-correction: Agents improve through accumulated feedback

Actions and Observations

Traditional LLMs:

- Input = text from user
- Output = text to user

But real-world intelligence uses **sensorimotor interfaces**, not just language

Animals—including humans—act and learn **through their bodies**, not privileged dialogue channels

Actions and Observations

LLMs increasingly control **digital tools** (e.g., API calls, file systems, code execution)

Early tool use came from **human imitation**, not direct experience

Now: agents gain **execution feedback**, enabling *experiential tool learning*

Actions and Observations

New prototypes (e.g., AutoGPT, Devin)

- Use human-like **UI interaction** (keyboard/mouse/screen)
- Perform multi-step computer tasks autonomously

This shift enables **exploration**, not just imitation

Actions and Observations

Action Type	Examples
Human-friendly	UI clicks, form filling, cursor dragging
Machine-friendly	API calls, CLI commands, script execution

Actions and Observations

Future experiential agents will:

- Control **robotic arms**, telescopes, sensors, drones
- Execute scientific experiments
- Operate digital infrastructure

These agents **observe**, **act**, and **adapt**—like humans do

Rewards

Better rewards come from **consequences**, not judgments

Grounded signals arise naturally in the environment:

- Heart rate, sales, emissions, energy, test scores, etc.

These enable **autonomous trial-and-error learning**

Rewards

Agent Type	Grounded Signal(s)
Health Agent	Sleep, heart rate, steps
Tutor Agent	Exam results, quiz accuracy
Science Agent	CO_2 levels, tensile strength, experiment yield

Rewards: What’s a good reward?

A well-designed single reward in a **complex environment** may:

- Induce broad intelligence
- Encourage mastery of diverse skills

Example: Maximizing power output in a physics simulator could require logic, control, memory

Rewards

Top-level: Maximize *user satisfaction / intent*

Low-level: Learn from *grounded environmental signals*

Agent refines its reward function through ongoing feedback

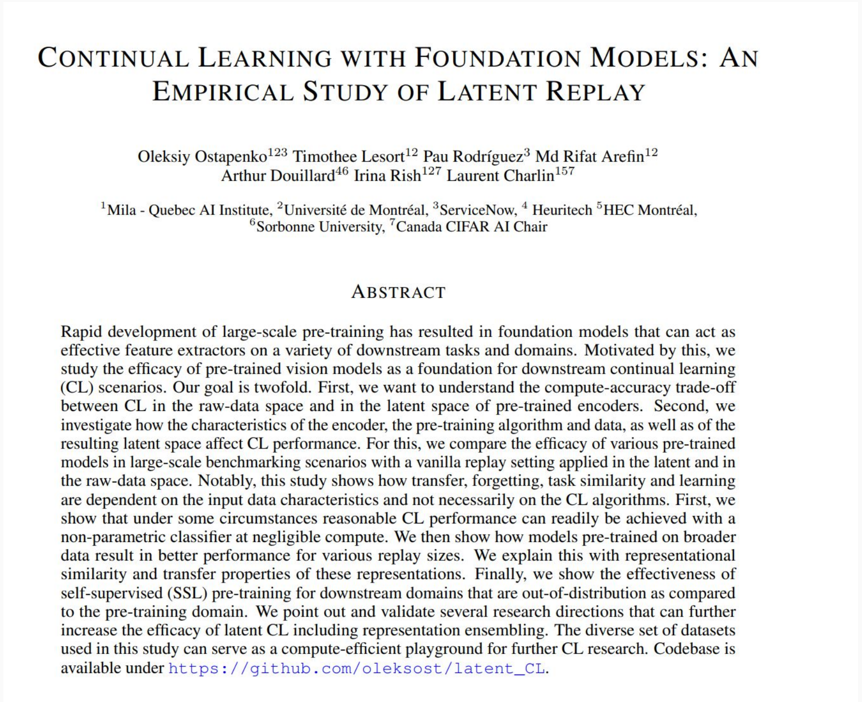
Research Questions

How can we make agents that continually learn?

How can we derive a reward function directly from the world (instead of human presences)

How can we let agents plan far in future!

Works on continual learning



CONTINUAL LEARNING WITH FOUNDATION MODELS: AN EMPIRICAL STUDY OF LATENT REPLAY

Oleksiy Ostapenko[123] Timothee Lesort[12] Pau Rodríguez[3] Md Rifat Arefin[12]
Arthur Douillard[46] Irina Rish[127] Laurent Charlin[157]

[1]Mila - Quebec AI Institute, [2]Université de Montréal, [3]ServiceNow, [4] Heuritech [5]HEC Montréal,
[6]Sorbonne University, [7]Canada CIFAR AI Chair

ABSTRACT

Rapid development of large-scale pre-training has resulted in foundation models that can act as effective feature extractors on a variety of downstream tasks and domains. Motivated by this, we study the efficacy of pre-trained vision models as a foundation for downstream continual learning (CL) scenarios. Our goal is twofold. First, we want to understand the compute-accuracy trade-off between CL in the raw-data space and in the latent space of pre-trained encoders. Second, we investigate how the characteristics of the encoder, the pre-training algorithm and data, as well as of the resulting latent space affect CL performance. For this, we compare the efficacy of various pre-trained models in large-scale benchmarking scenarios with a vanilla replay setting applied in the latent and in the raw-data space. Notably, this study shows how transfer, forgetting, task similarity and learning are dependent on the input data characteristics and not necessarily on the CL algorithms. First, we show that under some circumstances reasonable CL performance can readily be achieved with a non-parametric classifier at negligible compute. We then show how models pre-trained on broader data result in better performance for various replay sizes. We explain this with representational similarity and transfer properties of these representations. Finally, we show the effectiveness of self-supervised (SSL) pre-training for downstream domains that are out-of-distribution as compared to the pre-training domain. We point out and validate several research directions that can further increase the efficacy of latent CL including representation ensembling. The diverse set of datasets used in this study can serve as a compute-efficient playground for further CL research. Codebase is available under https://github.com/oleksost/latent_CL.