hyperpath: DEEP LEARNING PROGRAMS AS DIRECTED HYPERGRAPHS

Anonymous authors
Paper under double-blind review

ABSTRACT

Research code in machine learning is written twice. The first version is the idea, usually a diagram of values and the relationships between them. The second decides the loops over time, what to keep and for how long, how to batch, and where to compile. That second version takes the effort and carries the bugs, and when a language model writes it, someone still has to read it and check that it is right. We present `hyperpath`, a Python library in which a program is a directed (possibly cyclic) hypergraph. Each edge relates a set of values through an ordinary Python function and says nothing about when it runs. The user states which values they hold and which they want. The library compiles the one (acyclic) derivation the graph admits, naming what it could not reach where there is none and declining to choose where there are two. Users write time as a local relationship, `x[t]` and `x[t+1]`, and the library derives each call's loop over time, how long it keeps each value and how much of the sequence runs at once, leaving the user to declare a loop's direction and the order of an update. We evaluate by reproduction, writing GPT-2 124M against llm.c's PyTorch trainer, ResNet-50 against torchvision's ImageNet recipe, DDPM against its CIFAR-10 reference and PPO and DQN against CleanRL. We re-run every reference on the same hardware, and each graph reproduces its reference's result at comparable cost. We then write nineteen questions and variants of three of these models and of a VAE, a GAN and nanoGPT. The diffusion model's reference writes its loop over the timesteps by hand four times, for sampling, for DDIM, for the images predicted along the chain and for the likelihood, where its graph writes no loop and derives all four.

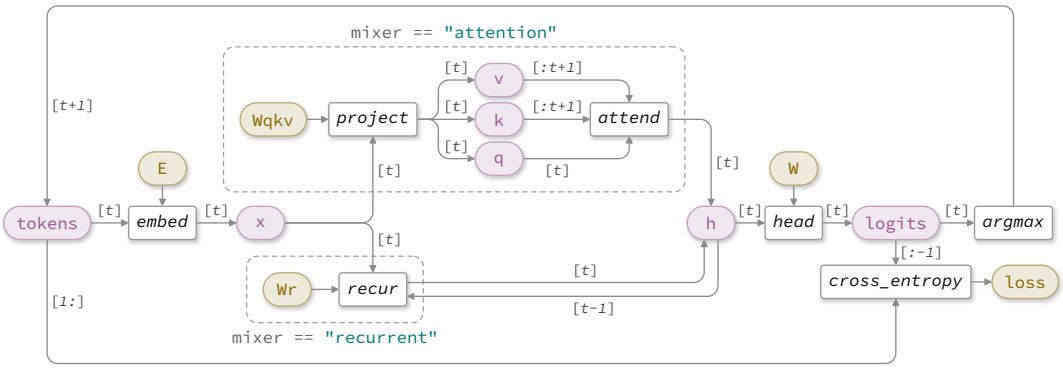
1 INTRODUCTION

Research code in machine learning is written twice. The first version is the idea, usually a diagram: a rollout, a value estimate, an advantage, a loss, and arrows between them. The second version implements it. Someone writes the loops over time, decides what to keep and for how long, batches the data and draws a compiler boundary. That second version absorbs much of the effort and the bugs, and a published result depends on it. In deep reinforcement learning, choices that never reach the pseudocode change results widely (Henderson et al., 2018) and account for most of PPO's gain over TRPO (Engstrom et al., 2020). The NeurIPS reproducibility programme lists under-specified models and training among the reasons results fail to reproduce (Pineau et al., 2021).

The second version also fixes one route through the model. A function `train_step(batch)` computes the loss and nothing else, so evaluating, resuming from a cached rollout or logging one intermediate each needs another entry point that restates which parts to compute. Nothing checks that those entry points agree. Array frameworks supply the operations and the compiler (Paszke et al., 2019; Bradbury et al., 2018; Ansel et al., 2024), and leave this composition to every project, which writes it by hand and throws it away with the project.

Now that models write code, cheap generation changes who types the second version and leaves open who has to trust it. When Google migrated code with a language model, the speed at which engineers could review the changes became the bottleneck (Nikolov et al., 2025), and people shown flawed code by an assistant often accept it and judge it secure (Perry et al., 2023; Serafini et al., 2025). When a language model writes an implementation in seconds, the reviewer's unit of work becomes the diff. This paper measures the size of that diff and does not study the review itself:

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.




```
g = hp.Graph(backend="torch")
mixer = g.condition()

E, W, Wqkv, Wr     = g.atom(role="input")  # weights
tokens             = g.series(recurrence="transition")
h                  = g.series(recurrence="cumulative")
x, q, k, v, logits = g.series()
loss               = g.atom()

x[t]        << embed(E, tokens[t])
with g.when(mixer == "attention"):
    (q[t], k[t], v[t]) << project(Wqkv, x[t])
    h[t]    << attend(q[t], k[:t+1], v[:t+1])
with g.when(mixer == "recurrent"):
    h[t]    << recur(Wr, h[t-1].or_else(0.0), x[t])
logits[t]   << head(W, h[t])
loss        << cross_entropy(logits[:-1], tokens[1:])
tokens[t+1] << argmax(logits[t])
```

Figure 1: The running example: seven edges and no control flow, drawn (above) and written (below). Each vertex takes its name from the variable it is assigned to. `x[t]` is this step's value, `k[:t+1]` the prefix through it, `h[t-1]` the step before, `tokens[t+1]` the next step's token, and `tokens[1:]` the finished sequence from its second token on, which the loss scores the logits against. A box stands for an edge, and a dashed box for the edges one condition guards. Every listing in this paper is printed by a script, and this one first checks its results against hand-written code. Appendix A prints the seven functions and how each is declared.

reviewers miss more defects in larger changes, both defects planted in an experiment (Baum et al., 2019) and security defects that went unnoticed in a large open-source project's reviews (Paul et al., 2021). Giving the language model of Figure 1 a recurrent mixer in place of attention costs the graph more code than it costs nanoGPT (Karpathy, 2022). We give nanoGPT a module calling PyTorch's GRU, while the graph declares the recurrence's edges beside attention's under a condition. Generating with the new mixer then costs the graph one call and no new edge, because the edge that continues the sequence never mentions a mixer. nanoGPT instead needs a new sampling loop to generate incrementally, threading the recurrent state out of every block and back in.

One part of that second version is already derived. Reverse-mode automatic differentiation (Baydin et al., 2018) took the backward pass away from the person writing the model. Few people write backward passes now. We argue that the route through a model, and its schedule over time, can move the same way.

We present `hyperpath`, a Python library in which a program is a directed hypergraph (Gallo et al., 1993). Each edge relates a set of input values to a set of output values through an ordinary Python function, and *no edge says when it runs*. The user states which values they hold and which they want, and the library compiles the one derivation the graph admits, or refuses and names what it could not reach or the choice it declines to make. The library is named for the *hyperpath*, graph theory's term for the edges needed to compute one value from given ones (Gallo et al., 1993; Ausiello & Laura, 2017). From edges that state one step of a relationship (`x[t]`, `x[t+1]`) it derives the schedule. Figure 1 is the graph we carry through the paper.

Contributions.

1. **Route** (Section 2). The user declares the composition and the library derives the route, answering with one plan or a named refusal. We designed it never to choose silently between two derivations (Appendix B). A question names the values it wants and binds those it holds, so a new question needs no new entry point. A variant sits beside the original, and each run picks one by assuming a condition.
2. **Schedule** (Section 3). From time written as a local relationship, the library derives each question's loop, what each series keeps and a *lifting* of per-step bodies into whole-series calls, where a user's function declares itself elementwise. An edge states one step and runs over the whole sequence wherever it can, and the loop carries a recurrent state from step to step, or, where a graph writes attention's keys and values as series, keeps them, by itself. A watch logs any value at the steps its read names.
3. **Implementation** (Section 4). The library generates plain Python, prints it and each plan on request, and hands one function to an existing tracing compiler such as `torch.compile`.
4. **Evaluation** (Section 5). We reproduce published models against standard reference implementations, reaching their results at comparable cost once planned, and measure what a new question or variant of a model costs to write. Over nineteen of them the graph writes 2,754 tokens where PyTorch writes 3,632 (PyTorch is shorter on five), and none is a loop over time or a hook into a model.

2 THE ROUTE: EDGES RELATE SETS OF VALUES

In `hyperpath`, a model is a directed hypergraph. Vertices hold *values*, and a hyperedge names a set of input vertices, a set of output vertices and a Python callable relating them. In Figure 1 we declare a language model whose mixer is causal attention or a recurrence, put each mixer's edges in a `g.when` block on the condition `mixer`, and let a question pick one with `assuming=`.

PyTorch's autograd records a computational graph as the code runs (Paszke et al., 2019), and in TensorFlow 1 the user builds one before a session runs it (Abadi et al., 2016). Each tensor there has one producing operation, so the route to any value is fixed. In both, the user writes the loop over time, which PyTorch records as the operations of every iteration and a TensorFlow 1 graph holds as a scan or while-loop construct. In our graph a value may have several producing edges and an edge may read a value at another step, so a question picks a route and the library derives its loop. The program we derive runs as plain Python, so autograd records it and a compiler traces it.

2.1 A QUESTION PICKS A ROUTE

We call a request a *question*: the values the user holds and the values they want. The library searches backward for a *derivation*, a set of edges and an order over them that reaches every target from the sources, acyclic even where the graph is not, and generates a Python program from it. Two questions of Figure 1 differ in how much of `tokens` the caller holds. Training holds the whole sequence and asks for the loss:

```
training_loss = g.derive({E: table, W: proj, Wqkv: qkv_proj, tokens: batch},
                         outputs=loss, assuming=[mixer("attention")])
```

A value the caller binds is never derived, so training never runs *argmax*. Generation binds a prompt of *t0* tokens as the bootstrap `tokens[:t0]` and asks for `tokens` itself:

```
generated = g.derive({E: table, W: proj, Wr: rec_proj, tokens[:t0]: prompt},
                     outputs=tokens, horizon=H, assuming=[mixer("recurrent")])
```

Neither call names an entry point, and no edge distinguishes the two. Each call declares everything it is handed, the weights its mixer reads included. Generation's program embeds the prompt first, and *head* and *argmax* start at the prompt's last token, as their $(t \geq t0\text{-}1)$ says:

```
Program from {E, W, Wr, tokens[:t0]} to tokens  (H = horizon)
├── x[:t0] = embed(E, tokens[:t0])        (vectorized)
└── ↻ t over 0:H
    ├── x[t] = embed(E, tokens[t])        (t ≥ t0)
    ├── h[t] = recur(Wr, h[t-1].or_else(0.0), x[t])
    ├── logits[t] = head(W, h[t])         (t ≥ t0-1)
    └── tokens[t+1] = argmax(logits[t])   (t ≥ t0-1)
Assumptions: mixer == 'recurrent'
Excluded: h[t] ← attend(q[t], k[:t+1], v[:t+1]) (when mixer == 'attention')
```

The assumption excludes the attention edges, as its last line shows for *attend*, and the four remaining edges run in one loop, which Section 3 derives. `(vectorized)` marks an edge run as one call over the whole sequence (Section 3.3).

2.2 One plan per question

Exactly one derivation compiles, which Appendix B checks over a corpus rather than proves. Zero derivations raise before any user code runs. Asking for the loss without choosing a mixer (no `assuming=`) raises `UnderivableError`, which names `h` as the value the search could not reach and the two conditions that excluded both of its edges. Two or more derivations raise as well. Declaring both mixers with no condition on either makes training ambiguous, and the refusal prints one tree that branches where the two programs part:

```
AmbiguityError: Multiple valid programs from {E, W, Wqkv, Wr, tokens} to loss exist.
├── x = embed(E, tokens[:H])  (vectorized)
├── one of:
│   ├── Alternative 1:
│   │   ├── (q, k, v) = project(Wqkv, x[:H])  (vectorized)
│   │   └── h = attend(q[:H], k[:H], v[:H])   (vectorized, time_axis=-2)
│   └── Alternative 2:
│       └── ↻ t over 0:H
│           └── h[t] = recur(Wr, h[t-1].or_else(0.0), x[t])
├── logits = head(W, h[:H])  (vectorized)
└── loss = cross_entropy(logits[:-1], tokens[1:])
```

Each alternative, with the steps around the branch, is a whole training program. Breaking the tie by declaration order would make a derived value depend, invisibly, on the order somebody typed the relationships in, so we ask the user to make the choice part of the model.

An edge declared inside a `when` block joins a derivation only when the call assumes its condition. `provided(x)` makes the same choice on whether the caller supplied `x`. A new question of a model therefore costs edges, and a `when` block where two answers coexist, which Section 5 measures.

3 The schedule: time as a local relationship

We call a value indexed by step a *series*, and write `x[t]` for its value at step `t`. In Figure 1, `recurrence="transition"` lets the edge of `tokens` write step `t+1` from reads at `t`, and `recurrence="cumulative"` lets the edge of `h` write step `t` from its own step `t-1`. We claim that the schedule follows from the steps each edge reads and writes: which series exist and over which steps, how much of each to keep, how each sits in memory, which edges run once over a whole series, and where a value may be dropped. None of the five appears in the seven edges, and all five move when the question moves.

3.1 Loops and storage

Training and generation, each under either mixer, give four programs. A program runs as a sequence of what we call *blocks*, a loop over the steps or a group of edges that run once each. The two training programs are the alternatives of the `AmbiguityError` in Section 2.2, and only the recurrence's has a loop, over `h` alone, between two blocks that run once. Generation loops over every edge in a single block, because *argmax* writes the token that the next step embeds. At full detail a plan also prints what each series costs, and generation under attention prints this table:

```
series   runs      holds    stores   read in   kept   builder
tokens   [t0-1:]   all      all      1         yes    stepwise
h        [t0-1:]   1 step   --       1         no     stepwise
x        [t0:]     1 step   --       1         no     stepwise
q        [t0:]     1 step   --       1         no     stepwise
k        [t0:]     all      all      1         no     packed
v        [t0:]     all      all      1         no     packed
logits   [t0-1:]   1 step   --       1         no     stepwise
```

`runs` names the steps the loop computes, `holds` counts those it keeps at once, `stores` says what is left when it ends, and `builder` whether it keeps the steps one by one (`stepwise`) or writes them into one array (`packed`). Over the prompt (the steps before *t0*), x, q, k and v depend only on tokens the caller bound, so the plan computes those steps ahead of the loop in one call per edge, as LLM inference prefills a prompt. The loop runs those four from *t0*, and h and logits from the prompt's last step, whose logits pick the first new token. Under the recurrence the table loses the rows of q, k and v, only x runs ahead of the loop, and h runs at every step, since each state reads the one before.

The two generation programs differ in what the loop keeps. Under the recurrence each step reads the h before the one it writes, so the plan holds two steps of h and one of x and drops the rest. Under attention each step reads every key and value up to it, so the plan keeps all of k and v and one step of x and q. It packs k and v into one array each, since every step reads a prefix of them, which a slice serves without a copy. Nobody wrote this key-value cache, its prefill or the discard. The models Section 5 measures call their references' attention modules on the residual stream's prefix instead, so their plans keep that prefix and recompute keys and values at every step, as the references do.

3.2 Backward loops

The language model runs forward. The advantage that PPO estimates from a rollout runs backward, and we write the generalized advantage estimator (Schulman et al., 2016) as two edges:

```
td[t]  << reward[t+1] + gamma * value[t+1] * (1 - done[t+1]) - value[t]
adv[t] << td[t] + gamma * lam * (1 - done[t+1]) * adv[t+1]
```

Each step of adv reads the next one, so we declare adv with `direction="backward"`, and td with it so that both start from zero at *tT*, where the rollout ends. From these edges and the rollout's (Appendix A), the library derives a plan that runs the environment forward, evaluates the critic and every residual td in one call each, and then runs ⟲ *t over :tT backward* from the step before *tT*, holding nothing but adv. CleanRL (Huang et al., 2022) writes the same estimate as a reversed `for` loop with a special case for its last step, which the graph writes as the two zeroing edges. The PPO experiment of Section 5 runs these edges, renamed, and its advantages equal that loop's bit for bit.

3.3 Lifting

An edge *lifts* when its per-step body can run once over whole series, each read such as x[*t*] replaced by the series x. It is a candidate when it writes all its outputs at *t* or all at *t*+1, from reads such as x[*t*], x[*t*-1] and x[:*t*+1] and the step index *t*. An edge that reads its own output lifts only where its function registers a body carrying that state (Appendix A), since no whole-series read expresses a value depending on its own earlier steps. So td lifts and adv stays in a backward loop.

The body must also mean the same thing over an added axis. Arithmetic lifts automatically. A user function opts in by declaring itself elementwise, so that the *same* body runs over whole arrays, or by registering a whole-series implementation beside its per-step one. *attend* registers one that folds the running prefixes of k and v. *recur* registers none, so its edge alone scans and every edge around it lifts. Section 7 names the declarations these rest on. Opt-in is per backend, so the same graph scans on plain Python storage and lifts on an array backend. A lifted edge checks its shapes on every call and raises on a mismatch instead of broadcasting into a wrong result.

3.4 Batches and training steps

A step's value is whatever the functions on its edges accept, so a batch widens each step's value and adds no loop. In the GPT-2 experiment step *t* holds the token at position *t* of every sequence in

the batch. We store the batch as llm.c loads it, one $B \times T$ array, by declaring time its second axis (`Backend("torch", axis=1)`), and each lifted edge runs once over it, as the reference's forward pass does. Appendix A declares the graph of Figure 1 over such a batch and prints its plan. Nothing ties a step to a position in a sequence. In the ResNet-50 experiment each step takes one micro-batch, which a series declared `supply="stream"` pulls from the loader as the loop reaches it.

We write the optimizer's update of the weights as an edge too. Its target holds no value (`role="effect"`), and a call asks for it with `run=[fit]` beside the values it wants. Written as `fit[7::8] << sgd_step(...)`, the edge steps the optimizer on every eighth micro-batch and so accumulates the gradients of eight, which takes the recipe's eight GPUs in turn on one. The library treats an edge's function as computing a value and nothing more, so a plan may run it over a whole series at once or share one call between two reads. We mark the optimizer step `side_effect=True` to rule both out. No value records that the next forward pass reads the weights the step changed, so the DDPM, DQN, VAE and GAN graphs declare that order with `after=`, and ResNet's keeps it by running step by step (`lifting="never"`). Left out, a lifted forward pass can run before the updates it should see, with nothing raised. Passed to a call in `watches=[...]`, a *watch* such as `g.watch(loss[::16])` writes every sixteenth step's loss while the call runs, and a call without it compiles as if the watch had never been declared. Appendix A lists the rest of the vocabulary and the experiments using it.

4 Implementation

4.1 The planner

The planner searches backward from the targets. For each value it needs, it chooses one of the edges that produce it, then does the same for that edge's inputs, until it reaches values the caller holds. It also groups the chosen edges into blocks that run in sequence, each a loop over the steps or a group of edges that run once. That grouping makes the search expensive, because one derivation can be cut into blocks in many ways that compute the same thing, and the number of such cuts grows exponentially with the number of independent outputs.

We therefore collapse these alternatives. At six places where the search would branch only into different cuts of one plan, it follows only the cuts that put the most work into each loop. A dropped branch might have led to the only derivation, or to a second one that makes the question ambiguous, so two checks back every collapse. If the collapsed search finds nothing, we search again in full. If it finds an answer, a second check looks for a rival derivation the collapse could have hidden. It tries every way of arranging the found derivation's edges into loops, and where a value has several producing edges, it searches again with each in turn. Appendix B defines when two cuts are the same plan, lists the six places and what each must guarantee, and says how far the checks reach.

4.2 Generated code

Each edge becomes several statements of a generated Python function (Figure 11, Appendix C), and `program.source()` prints the function as it runs, so a traceback points into code the user can read. `Program.kernel()` composes a plan's functions into one function of its inputs, its *kernel*, which we hand to `torch.compile` (Ansel et al., 2024). Such a compiler records the tensor operations a Python function performs, compiles that record, and installs runtime checks, *guards*, that decide on each call whether the record still applies. We generate source text and not closures, because the compiler guards each constant a closure captures and none written as a literal in the source. We count its *guard lines*, the lines of the guard tree it prints per artifact, because the count is stable across runs and names the code that caused it. Appendix C gives the counts under PyTorch 2.13.

5 Evaluation

We evaluate by reproduction. We write five published models as graphs and train each beside the hand-written implementation it reproduces: GPT-2 124M (Radford et al., 2019) beside llm.c (Karpathy, 2024), ResNet-50 (He et al., 2016) beside torchvision's ImageNet recipe (TorchVision maintainers and contributors, 2016), a DDPM (Ho et al., 2020) beside its CIFAR-10 reference (Chen, 2022), and PPO (Schulman et al., 2017) and DQN (Mnih et al., 2015) beside CleanRL (Huang et al.,

Table 1: Fidelity (C1) and cost (C2) of each reproduction. C2 divides the graph's wall-clock by the reference's, at the reference's settings, so a value below 1 means the graph is faster. In parentheses, we divide by the same arithmetic written by hand in PyTorch, on PPO's three classic-control cells.

Model	C1 metric	Reference	Graph	C2
DDPM, CIFAR-10	FID over 50,000 images	3.2248	3.2359	1.00
GPT-2 124M, FineWeb 10B	Validation loss at 19,552 steps	3.2772	3.2773	1.00
ResNet-50, ImageNet	Top-1 after 90 epochs	75.606	75.492	0.99
PPO, eight cells	IQM return, ten seeds (HalfCheetah 30)	Each difference contains 0		0.67–0.86 (1.03–1.04)
DQN, three cells	IQM return over ten seeds	Each difference contains 0		0.98–0.99

2022). Each graph reproduces its reference's result at comparable cost (Table 1). We pretrained GPT-2 on 10 billion tokens of FineWeb (Penedo et al., 2024) three times beside llm.c's own trainer, at the one seed it sets, and each pair ends less than 0.001 apart in validation loss, the graph lower in two of them, and llm.c's own three runs land up to 0.0015 apart. Our evaluator and llm.c's give each of the six checkpoints the same loss to every digit they record. On HellaSwag (Zellers et al., 2019) the graph's three checkpoints score 0.301 on average and llm.c's 0.302.

5.1 Claims and protocol

We state three claims, and fixed which each experiment serves when its first result was recorded, not when we drew the tables. *C1, fidelity.* A model written as a graph reproduces the result of the same model written by hand, re-run on our hardware, with the same returns or loss. *C2, cost.* The abstraction is approximately free in wall-clock and throughput, and in peak memory where an experiment records it, against a canonical reference on the same hardware. We run each reference as its authors ship it, and the graph in the same mode and precision, compiled only where the reference compiles itself. Where we also hand the graph a compiler the reference does not use, we compare that run only with the graph's uncompiled one, so we never credit the library with the compiler's speed. *C3, the size of a change.* A variant, or a new question of one model, costs edges rather than a second entry point, measured as a diff against the reference and never asserted.

What we argue is *equivalence*, and an underpowered comparison finds no difference whether or not one exists, so we report by three rules. First, equivalence rests on short runs: each reproduction trains a short run both ways from the same initialisation and random draws and compares them. Twelve steps of a small DDPM with its three checkpoints, two epochs of ResNet-50 on a slice of ImageNet, five float32 steps of GPT-2, 3,000 DQN steps that fit 199 times and sync the target network three times, and five PPO iterations each end with the graph's weights equal to the reference's to the last bit. For PPO we compare against a copy of CleanRL's loop that takes the graph's random draws and rounds as the graph does (Appendix D). The long runs then check that nothing drifts over a whole training: we report each reinforcement learning cell as an interquartile mean (IQM) over at least ten seeds with a bootstrap interval (Agarwal et al., 2021), and an interval on the difference between the two implementations. In all eleven cells the 95% interval on the difference contains zero. The other reproductions train one to three times per implementation, so we report the gap and how many runs it rests on. Second, we re-run every reference on our hardware and framework version with the same seeds. A published curve checks the re-run and never replaces it as the comparator, since a parity claim read off someone else's infrastructure is a claim about their cluster. Third, we vendor every reference byte for byte at a pinned upstream commit, with a digest per file, and keep every adaptation in our experiment code, since a parity claim against a reference edited in place is about a file nobody else has.

5.2 Fidelity and cost

The DDPM row of Table 1 averages four scores per implementation, from two trainings for the 2,040 epochs the reference's CIFAR-10 recipe specifies, each scored over 50,000 images at two sample seeds. The reference's own two trainings land further apart than the graph lands from either, and in both pairs the two loss curves stay within 1.3×10^{-4} of each other at every epoch and coincide at none (Appendix D gives the trainings' cost). The ResNet-50 row rests on one run of

each implementation. For the weights its recipe names, torchvision reports 76.130 top-1; our re-run of the recipe on one GPU lands 0.52 under that, and the graph a further 0.11 under it.

ResNet-50's graph takes 0.99 of the recipe's time for an epoch, at a 3.02 GiB peak against the recipe's 2.96, and a compiler takes that to 0.83 of its own eager time, or to 0.53 when it also records the step as a CUDA graph, which raises its peak memory from 3.02 GiB to 5.08. DDPM's graph peaks at 7.45 GiB against the reference's 7.00, holding three more copies of the network's weights (Appendix D). On PPO's three classic-control cells we also timed the graph's arithmetic written by hand in PyTorch, and the graph takes 1.03 to 1.04 of its time. So the graph's lead there comes from how the file does the work, about half of it from how the file builds its policy's distribution (Appendix D). DQN's graph comes within 2% on its three, where a compiler buys PPO a further 11 to 28% and costs DQN 5 to 6%. Compiling sits outside these clocks, and planning takes every graph under 0.3 s.

5.3 The size of a change

DDPM, GPT-2 and PPO serve C3, with three *demonstrations*, a VAE (Kingma & Welling, 2014), a GAN (Goodfellow et al., 2014) and nanoGPT-GRU, which swaps nanoGPT's Transformer attention (Vaswani et al., 2017) for a GRU (Cho et al., 2014), each written against a PyTorch reference. Each demonstration's outputs, and its weights after several training steps, equal its reference's bit for bit (Appendix D). We do not train the demonstrations to convergence, so they report no result for C1. We count both sides in Python tokens without comments, docstrings or type annotations, as a diff that includes any new call the question needs. Over nineteen questions the graph writes 2,754 tokens and PyTorch 3,632. We chose fifteen of the questions and wrote every PyTorch answer but DDPM's three (Appendix D.8). Where the graph answers a question with no new edge, as it does for five of them, it writes a quarter to four fifths less than plain PyTorch; where it adds edges, it writes from under half to over one and a half times as much. We asked the diffusion model six questions. Its reference answers three with methods of its own. For a strip of the model's predicted images, which the reference answers by rewriting its sampling loop, the graph writes 72 tokens against 248. For DDIM sampling it writes 408 against 647, and for the variational bound 435 against 450. The reference's methods for the bound write its loop over the timesteps once more, and two of their statements raise as written (Appendix D). For the other three we wrote functions of 112, 148 and 139 tokens that each repeat the sampling loop, and the graph 100, 160 and 208.

`train_gpt2.py` answers none of GPT-2's four questions, and we wrote a function for each, the first three reaching inside `GPT.forward` with hooks, as plain PyTorch does (Appendix E names the libraries built for such questions). The graph writes 108 tokens against 154 for the logit lens (nostalgebraist, 2020), 211 against 244 for the Jacobian lens (Gurnee et al., 2026), 131 against 177 for activation patching, which binds the activation it patches, and 46 against 100 for the in-context score. With NNsight (Fiotto-Kaufman et al., 2025) in place of hooks, the first three take 117, 186 and 119 tokens, and the graph writes less than NNsight only for the logit lens.

CleanRL writes PPO for continuous actions as a second file, a copy of `ppo.py` with, among other changes, a Gaussian policy in place of the categorical one. The graph keeps both policies under a condition, so we edited `ppo.py` to keep both too, in 279 tokens across seven places. The graph writes 167, the Gaussian policy's edges and the head holding the same mean network and log standard deviation.

Swapping the language model's attention for a GRU costs the graph 273 tokens against nanoGPT's 176. Generating with the new mixer costs the graph one call of 45 tokens, and nanoGPT a 240-token loop we wrote to carry the state from token to token. The graph writes less for each of the VAE's three questions, at most 70 tokens where PyTorch writes 58 to 154. Of the GAN's three, it writes less only for the minimax loss, 42 tokens against 49. For the other two it writes 79 and 96 against 61 and 88, where PyTorch adds a few lines to its loop and the graph declares each loss as edges under a condition. The GAN's losses and the VAE's new estimator train through the unchanged call.

6 Related work

Theano (Bergstra et al., 2010) and TensorFlow 1 (Abadi et al., 2016) are the closest ancestors, and a compiled `theano.function` or a `session.run(fetches)` asks a question of a standing graph other

questions reuse. With one producing operation per tensor, the route there is unique, so nothing is searched for and no question is ambiguous. Feeding a tensor through `feed_dict` replaced its value as binding one does here. Build tools from `make` (Feldman, 1979) to Snakemake (Mölder et al., 2021) find an order from declared dependencies between files, and Snakemake refuses a file its rules produce ambiguously, as Section 2 refuses a value. Halide (Ragan-Kelley et al., 2013) infers each stage's buffer from the region its consumers read, as Section 3 derives what a series keeps, though its user or autotuner still writes where each stage runs; it and the tensor compilers after it take the program's composition as given, and we hand our derivation to one of them. Appendix E treats them at length, with Lustre, Lucid, Dyna, JAX's `vmap` and `scan`, TorchRL and interpretability tooling.

7 Limitations

Adoption and maturity. `hyperpath` has no external users and nothing published uses it, so we cannot argue its significance from uptake, and we built the evaluation to need none. The library remains under active development, we still fix defects, the planner's among them, and we prove nothing about the planner or the lift. Beyond the parts the body shows, Appendix A lists 40 more, which a token count does not measure. It marks those no experiment uses, and shows the four largest (a loop that stops on a condition, a window, steps picked by an index and a choice between routes), each in a figure of its own, checked against code written by hand.

Planning and ambiguity. The backward search is worst-case exponential. Its collapses plan a synthetic 128-edge chain in 24 ms and a 64-recurrence graph in 31 ms on an AMD EPYC 9655, and without them the evaluation's DQN graph would take 8.4 s to plan instead of 12 ms. Where a collapse loses a plan, a second, complete search recovers it, which some graphs need and none of our reproductions does. We pay planning once per derivation shape, and a graph of more than about 120 recurrences exceeds CPython's default recursion limit. Refusing ambiguity leaves the user to pick between two interchangeable derivations, which we regard as the correct default and still a cost.

Lifting and scope. Lifting never derives a parallel scan (Martin & Cundy, 2018) for a linear recurrence such as `adv`, though a rule may supply one. Four declarations can change a result without a word when they are wrong or missing: an `after=` ordering an update no value carries, a `side_effect=True` keeping a plan from sharing or broadcasting a call, a function declared elementwise or given a whole-series body, and a value a scope keeps after an optimizer changes in place the weights it was computed from. Edges no value orders run in the order they were declared, and a plan refuses two with side effects that nothing orders. We ran each measured graph's short check again with `lifting="never"`, which runs every edge over a series one step at a time. Where training lifts, the two runs agree within 10^{-5} of each output's largest value or bit for bit, and the comparison catches two errors we planted (Appendix D.8). A user can check their own graph the same way, by deriving it a second time with `lifting="never"` and comparing the two results. Where no edge lifts, both derivations are the same program, so the comparison cannot fail. A call derives the loop over one step index only. A second, such as GPT-2's training steps, is a Python loop around the calls, and a question that binds one call's result into the next takes several calls. A backward scan runs only over a series stored whole, and we plan for one process, leaving multi-device work and differentiation to the backend. The shape checks cost 1 to 6% of eager wall-clock on the CPU (Appendix C), and a loop run step by step spends 0.15 to 0.20 ms of host time a step, which nanoGPT-GRU's generation does not hide, at 1.06 to 1.08 times a hand-written loop (Appendix D).

8 Conclusion

Research code is written twice, and this paper argues that the second version shrinks to a few declarations: the direction of a loop, the order of an update, and the calls a training run makes. Each question asked of a `hyperpath` program compiles to its own program, with the route and the loops over time both derived. Standard models written this way reproduce their hand-written results at comparable cost. A change to a model costs edges and the functions and calls they need. Over nineteen questions, none of them a loop over time, the graph writes 2,754 tokens where PyTorch writes 3,632. A language model can now write the second version in seconds, and somebody still has to read what it wrote.

REPRODUCIBILITY STATEMENT

The library, the examples, the evaluation harness and the recorded runs are submitted as anonymised supplementary material. `hyperpath` is a pure-Python library of roughly 128k lines and 24,000 tests, and requires only Python 3.14, with NumPy (Harris et al., 2020) and PyTorch (Paszke et al., 2019) as optional extras. Every experiment records its environment, the commit it ran at, its seed list and its raw per-run rows, so every number in Section 5 is read from a checked-in corpus file. Comparators are vendored at pinned upstream commits with per-file digests that are re-verified by the test suite.

AI USE STATEMENT

We used generative AI tools throughout this work, and this statement says where they helped and how we checked their output. We worked with Claude Code, an agentic coding assistant, running Anthropic's Claude models.

We designed the model the library rests on and wrote its first implementation without AI assistance. From mid-2026 the assistant wrote most of the code, in the library and in the evaluation harness, including every answer Section 5.3 counts, on both sides, but the three that DDPM's reference already had. We made the design decisions, often by choosing among alternatives the assistant laid out. The assistant also helped design the evaluation protocol, launched the experiments and drafted first readings of their results.

Changes to the library land only after they pass a linter, a static type checker, the test suite in two environments and a documentation build, and continuous integration repeats these checks. The evaluation vendors each reference implementation unmodified at a pinned upstream commit, and the test suite re-verifies the vendored files. Each number the evaluation reports is read from a checked-in run file.

The assistant drafted most of the paper's text, including the abstract, and proposed its title and keywords. We revised the drafts and decided what the paper claims. The assistant also found and formatted references, and we checked each bibliography entry against its published version.

We did not use generative AI to generate data, and the paper states no mathematical claims or proofs. We have reviewed all AI-assisted work and take responsibility for the full content of this paper, including text, code, claims and results produced with the aid of generative AI.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pp. 265–283. USENIX Association, 2016.

Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron Courville, and Marc G. Bellemare. Deep reinforcement learning at the edge of the statistical precipice. In *Advances in Neural Information Processing Systems*, volume 34, pp. 29304–29320. Curran Associates, Inc., 2021.

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster machine learning through dynamic Python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*, pp. 929–947. ACM, 2024. doi: 10.1145/3620665.3640366.

E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, 1977. doi: 10.1145/359636.359715.

Giorgio Ausiello and Luigi Laura. Directed hypergraphs: Introduction and fundamental algorithms—a survey. *Theoretical Computer Science*, 658:293–306, 2017. doi: 10.1016/j.tcs.2016.03.016.

Tobias Baum, Kurt Schneider, and Alberto Bacchelli. Associating working memory capacity and code change ordering with code review performance. *Empirical Software Engineering*, 24(4):1762–1798, 2019. doi: 10.1007/s10664-018-9676-8.

Atılım Güneş Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A CPU and GPU math compiler in Python. In *Proceedings of the 9th Python in Science Conference (SciPy 2010)*, pp. 18–24, 2010. doi: 10.25080/Majora-92bf1922-003.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research*, 20(28):1–6, 2019.

Albert Bou, Matteo Bettini, Sebastian Dittert, Vikash Kumar, Shagun Sodhani, Xiaomeng Yang, Gianni De Fabritiis, and Vincent Moens. TorchRL: A data-driven decision-making library for PyTorch. In *International Conference on Learning Representations*, pp. 1778–1811, 2024.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Yash Katariya, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: Composable transformations of Python+NumPy programs, 2018. URL https://github.com/jax-ml/jax.

Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1):1–32, 2017. doi: 10.18637/jss.v076.i01.

Tianqi Chen. PyTorch implementation of denoising diffusion probabilistic models, 2022. URL https://github.com/tqch/ddpm-torch.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, pp. 578–594. USENIX Association, 2018.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pp. 1724–1734. Association for Computational Linguistics, 2014. doi: 10.3115/v1/D14-1179.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255. IEEE, 2009. doi: 10.1109/CVPR.2009.5206848.

Jason Eisner and Nathaniel W. Filardo. Dyna: Extending Datalog for modern AI. In *Datalog Reloaded*, volume 6702 of *Lecture Notes in Computer Science*, pp. 181–220. Springer, 2011. doi: 10.1007/978-3-642-24206-9_11.

Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation matters in deep RL: A case study on PPO and TRPO. In *International Conference on Learning Representations*, 2020.

Stuart I. Feldman. Make — a program for maintaining computer programs. *Software: Practice and Experience*, 9(4):255–265, 1979. doi: 10.1002/spe.4380090402.

Jaden Fiotto-Kaufman, Alexander Loftus, Eric Todd, Jannik Brinkmann, Koyena Pal, Dmitrii Troitskii, Michael Ripa, Adam Belfki, Can Rager, Caden Juang, Aaron Mueller, Samuel Marks, Arnab Sen Sharma, Francesca Lucchetti, Nikhil Prakash, Carla Brodley, Arjun Guha, Jonathan Bell, Byron Wallace, and David Bau. NNsight and NDIF: Democratizing access to open-weight foundation model internals. In *International Conference on Learning Representations*, pp. 92337–92370, 2025.

Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42(2–3):177–201, 1993. doi: 10.1016/0166-218X(93)90045-P.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

Wes Gurnee, Nicholas Sofroniew, Adam Pearce, Mateusz Piotrowski, Isaac Kauvar, Runjin Chen, Anna Soligo, Paul Bogdan, Euan Ong, Rowan Wang, T. Ben Thompson, David Abrahams, Subhash Kantamneni, Emmanuel Ameisen, Joshua Batson, and Jack Lindsey. Verbalizable representations form a global workspace in language models. *Transformer Circuits Thread*, 2026. URL https://transformer-circuits.pub/2026/workspace/.

Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991. doi: 10.1109/5.97300.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. doi: 10.1038/s41586-020-2649-2.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778. IEEE, 2016. doi: 10.1109/CVPR.2016.90.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, pp. 3207–3214, 2018. doi: 10.1609/aaai.v32i1.11694.

Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 556–571. ACM, 2017. doi: 10.1145/3062341.3062354.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In *Advances in Neural Information Processing Systems*, volume 33, pp. 6840–6851. Curran Associates, Inc., 2020.

Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G. M. Araújo. CleanRL: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022.

Andrej Karpathy. nanoGPT, 2022. URL https://github.com/karpathy/nanoGPT.

Andrej Karpathy. llm.c: LLM training in simple, raw C/CUDA, 2024. URL https://github.com/karpathy/llm.c.

Diederik P. Kingma and Max Welling. Auto-encoding variational Bayes. In *International Conference on Learning Representations*, 2014.

Alex Krizhevsky. Learning multiple layers of features from tiny images, 2009. URL https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14. IEEE, 2021. doi: 10.1109/CGO51591.2021.9370308.

Eric Martin and Chris Cundy. Parallelizing linear recurrent neural nets over sequence length. In *International Conference on Learning Representations*, 2018.

Lars Mescheder, Andreas Geiger, and Sebastian Nowozin. Which training methods for GANs do actually converge? In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 3481–3490. PMLR, 2018.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. doi: 10.1038/nature14236.

Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. Build systems à la carte. *Proceedings of the ACM on Programming Languages*, 2(ICFP):79:1–79:29, 2018. doi: 10.1145/3236774.

Felix Mölder, Kim Philipp Jablonski, Brice Letcher, Michael B. Hall, Christopher H. Tomkins-Tinch, Vanessa Sochat, Jan Forster, Soohyun Lee, Sven O. Twardziok, Alexander Kanitz, Andreas Wilm, Manuel Holtgrewe, Sven Rahmann, Sven Nahnsen, and Johannes Köster. Sustainable data analysis with Snakemake. *F1000Research*, 10:33, 2021. doi: 10.12688/f1000research.29032.2.

Neel Nanda and Joseph Bloom. TransformerLens, 2022. URL https://github.com/TransformerLensOrg/TransformerLens.

Stoyan Nikolov, Daniele Codecasa, Anna Sjövall, Maxim Tabachnyk, Satish Chandra, Siddharth Taneja, and Celal Ziftci. How is Google using AI for internal code migrations? In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 481–492. IEEE, 2025. doi: 10.1109/ICSE-SEIP66354.2025.00048.

nostalgebraist. Interpreting GPT: The logit lens, 2020. URL https://www.lesswrong.com/posts/AcKRB8wDpdaN6v6ru/interpreting-gpt-the-logit-lens.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. Getting to the point: Index sets and parallelism-preserving autodiff for pointful array programming. *Proceedings of the ACM on Programming Languages*, 5(ICFP):88:1–88:29, 2021. doi: 10.1145/3473593.

Rajshakhar Paul, Asif Kamal Turzo, and Amiangshu Bosu. Why security defects go unnoticed during code reviews? A case-control study of the Chromium OS project. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 1373–1385. IEEE, 2021. doi: 10.1109/ICSE43902.2021.00124.

Guilherme Penedo, Hynek Kydlíček, Loubna Ben Allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The FineWeb datasets: Decanting the web for the finest text data at scale. In *Advances in Neural Information Processing Systems*, volume 37, pp. 30811–30849. Curran Associates, Inc., 2024. doi: 10.52202/079017-0970.

Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure code with AI assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*, pp. 2785–2799. ACM, 2023. doi: 10.1145/3576915.3623157.

Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d'Alché Buc, Emily Fox, and Hugo Larochelle. Improving reproducibility in machine learning research (a report from the NeurIPS 2019 reproducibility program). *Journal of Machine Learning Research*, 22(164):1–20, 2021.

PyTorch contributors. PyTorch examples, 2017. URL https://github.com/pytorch/examples.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In *International Conference on Learning Representations*, 2016.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019. URL https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*, pp. 519–530. ACM, 2013. doi: 10.1145/2491956.2462176.

Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference (SciPy 2015)*, pp. 126–132, 2015. doi: 10.25080/Majora-7b98e3ed-013.

Alex Rogozhnikov. Einops: Clear and reliable tensor manipulations with Einstein-like notation. In *International Conference on Learning Representations*, 2022.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, 115 (3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training GANs. In *Advances in Neural Information Processing Systems*, volume 29, pp. 2234–2242. Curran Associates, Inc., 2016.

John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations*, 2016.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

Raphael Serafini, Asli Yardim, and Alena Naiakshina. Exploring the impact of intervention methods on developers' security behavior in a manipulated ChatGPT study. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*, pp. 520:1–520:26. ACM, 2025. doi: 10.1145/3706598.3713989.

Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, pp. 10–19. ACM, 2019. doi: 10.1145/3315508.3329973.

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033. IEEE, 2012. doi: 10.1109/IROS.2012.6386109.

TorchVision maintainers and contributors. TorchVision: PyTorch's computer vision library, 2016. URL https://github.com/pytorch/vision.

Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A standard interface for reinforcement learning environments. In *Advances in Neural Information Processing Systems*, volume 38, pp. 163114–163129. Curran Associates, Inc., 2025. doi: 10.52202/085713-4916.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

XLA and TensorFlow teams. XLA – TensorFlow, compiled, 2017. URL https://developers.googleblog.com/xla-tensorflow-compiled/.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4791–4800. Association for Computational Linguistics, 2019. doi: 10.18653/v1/P19-1472.

A Examples and vocabulary

The body introduces the vocabulary as its running examples need it, and this appendix writes the examples out in full. Figure 2 prints the seven functions of Figure 1 and how each is declared. Figure 3 writes the rollout of Section 3 with the advantage estimated from it, one loop running forward and one backward in a single graph, and prints the plan the library derives for the advantage. Figure 4 prints the batch of Section 3.4.

Table 2 lists everything we write a graph in beyond Python's own operators, indexing and attribute access on what an edge reads, grouped by what each part declares, and names the experiments of Section 5 whose graphs use each part. Of the parts no experiment uses, we write out the four largest: a loop that stops on a condition (Figure 5), a window over the last few steps (Figure 6), steps picked by an index (Figure 7), and a choice between two edges made by the call (Figure 8). The table lists the smaller parts no experiment uses without a figure of their own, and Figure 2 declares one of them, `time_axis=`.

Two modules ship operations we would otherwise write as rules ourselves. `hyperpath.array` holds 75 array operations as rules, such as *`array.mean`* and *`array.zeros_like`*, declared so that an edge calling one lifts. *`array.mean`*`(x[t], 0)` runs once over the whole series, where the method call `x[t].mean(0)` declares nothing and runs step by step. `hyperpath.distributions` holds a distribution as a value, built by a family's constructor such as *`dist.normal`*`(mean, std)`. Rules carry its verbs (*`dist.sample`*, *`dist.log_prob`*, *`dist.kl_divergence`*), so a draw with it is an edge that reads a random stream as an input. Figure 8 uses both. PPO's graph takes its policy's categorical from this module, where CleanRL builds a `torch.distributions.Categorical` at every step, and Section 5.2 attributes about half of PPO's lead to that difference.

A script prints each figure here from its own source, as it prints every listing in this paper, and first checks each graph against code written by hand. The library prints each plan, and we cut nothing from it. The rollout's advantages equal CleanRL's reversed loop to the last bit. The batch's scores, the window's means and the picked steps equal loops over the same steps, the stopped iteration returns the iterates of the `while` loop that writes it, and each route returns its own formula. Where a check compares one call over a whole array with a loop, it allows a difference of 10^{-12}, since the two add the same numbers in a different order.

```python
@hp.rule.vectorized("torch", axes="any")
def embed(table, ids):
    return table[ids]

@hp.rule.vectorized("torch", axes="any", outputs=3)
def project(weights, values):
    return (values @ weights).chunk(3, dim=-1)

@hp.rule(time_axis=-2)
def attend(query, key, value):
    return F.scaled_dot_product_attention(query.unsqueeze(-2), key, value).squeeze(-2)

@attend.vectorized("torch", folds="history")
def attend_whole(query, key, value):
    # Row t is what the body above answers from key[:t+1] and value[:t+1].
    return F.scaled_dot_product_attention(query, key, value, is_causal=True)

@hp.rule
def recur(weights, state, current):
    return torch.tanh(state + current @ weights)

@hp.rule.vectorized("torch", axes="any")
def head(projection, hidden):
    return hidden @ projection

@hp.rule
def argmax(scores):
    return scores.argmax(-1)

@hp.rule
def cross_entropy(scores, answers):
    return F.cross_entropy(scores.flatten(0, -2), answers.flatten())
```

Figure 2: The seven functions of Figure 1 in PyTorch, printed from the script that prints that figure. We declare *embed*, *project* and *head* by a single body over whole series, which also computes one step as a series one step long. *attend* declares the axis it reads time on, the second-to-last as PyTorch's attention reads it, and the library moves time there from whichever axis the storage keeps it on. Its second body, `attend_whole`, answers the reads `k[:t+1]` and `v[:t+1]` of every step at once. *recur* registers none, so its edge loops and every edge around it lifts. A body declaring `carries=` would run it as one call too (Table 2), as nanoGPT-GRU's does, and we leave it out so that the training programs of Section 3 show a loop. *cross_entropy* reads two finished series and *argmax* runs inside generation's loop, so neither needs one. Each function takes one sequence or a batch of them (Figure 4). The library does not check that a second body computes what the first does (Section 7).

```
g = hp.Graph(backend="numpy")
env = g.atom(role="input")
gamma, lam = g.atom(role="input")

# The rollout runs forward: each step writes the next one.
obs, reward, done = g.series(recurrence="transition")
action, value = g.series()
(obs[:t0], reward[:t0], done[:t0]) << env_start(env)
(obs[t+1], reward[t+1], done[t+1]) << env_step(env, action[t])
action[t] << act(obs[t])
value[t]  << critic(obs[t])

# The advantage runs backward: each step reads the next one.
td, adv = g.series(recurrence="cumulative", direction="backward")
td[tT:]  << array.zeros_like(value[-1:])
adv[tT:] << array.zeros_like(value[-1:])
td[t]  << reward[t+1] + gamma * value[t+1] * (1 - done[t+1]) - value[t]
adv[t] << td[t] + gamma * lam * (1 - done[t+1]) * adv[t+1]

# The advantages of one rollout of 25 steps, at CleanRL's two coefficients.
advantages = g.derive({env: walk, gamma: 0.99, lam: 0.95}, outputs=adv, horizon=25)
```

```
Program from {env, gamma, lam} to adv  (H = horizon)
├── 1. (obs[:t0], reward[:t0], done[:t0]) = env_start(env)
├── 2. ⟳ t over t0-1:H
│      ├── action[t] = act(obs[t])
│      └── (obs[t+1], reward[t+1], done[t+1]) = env_step(env, action[t])
├── 3. value = critic(obs[:H])
↪  (broadcast)
│      td[tT:] = array.zeros_like(value[-1:])
│      adv[tT:] = array.zeros_like(value[-1:])
│      td = reward[1:tT+1] + gamma * value[1:tT+1] * (1 - done[1:tT+1]) - value[:tT]
↪  ++ td[tT:]     (broadcast)
└── 4. ⟳ t over :tT backward
       └── adv[t] = td[t] + gamma * lam * (1 - done[t+1]) * adv[t+1]
```

Figure 3: A rollout over a toy environment and critic, the advantage estimated from it in the PPO experiment's own edges, and the plan derived for its call. The rules `env_start`, `env_step`, `act` and `critic` are defined in the script that prints this figure. In the rollout's edge, `[t+1]` stands on the *left*, so each step writes the next and the loop runs forward. In the advantage's edge, it stands on the *right*, so each step reads the next and the loop runs backward. The two series declare the direction, and the library places the loops. Between them the critic and every temporal-difference residual run as one call over the whole rollout each. The edges into `td[tT:]` and `adv[tT:]` zero both series where the rollout ends.

```
# Every series of this graph is stored with time on its second axis.
g = hp.Graph(backend=Backend("torch", axis=1))
# ... Figure 1's declarations and edges, unchanged

# Four sequences of 12 tokens, whose logits come back 4 x 12 x 50.
batch = torch.randint(0, 50, (4, 12))
scores = g.derive({E: table, W: proj, Wqkv: qkv_proj, tokens: batch},
                  outputs=logits, assuming=[mixer("attention")])
```

```
Program from {E, W, Wqkv, tokens} to logits  (H = horizon, storage: axis=1)
├── x = embed(E, tokens[:H])           (vectorized)
├── (q, k, v) = project(Wqkv, x[:H])   (vectorized)
├── h = attend(q[:H], k[:H], v[:H])    (vectorized, time_axis=-2)
└── logits = head(W, h[:H])            (vectorized)
Assumptions: mixer == 'attention'
Excluded: h[t] ← recur(Wr, h[t-1].or_else(0.0), x[t]) (when mixer == 'recurrent')
```

Figure 4: The graph of Figure 1 over a batch of sequences, stored $B \times T$ with time second, and the plan derived for `logits`. We change only the storage the graph is built on, and each edge still states one step. The library keeps time on the second axis of every series, runs each lifted edge once over the whole array, and hands *attend* time second-to-last, as it declares, so the prefixes `k[:t+1]` and `v[:t+1]` run within each sequence. The plan's header names the axis, and `tokens[:H]` reads every sequence up to the horizon. The script holds every sequence's logits to a loop written by hand over that sequence alone.

```
g = hp.Graph()
a = g.atom(role="input")

# The next iterate is the mean of this one and a / root[t].
root = g.series(recurrence="transition")
root[t+1] << (root[t] + a / root[t]) / 2

# Iterate from 1 until the square is within 1e-12 of a, for at most 100 steps.
roots = g.derive({a: value, root[:t0]: [1.0]}, outputs=root,
                 horizon=100, until=abs(root[t] * root[t] - a) < 1e-12)
```

```
Program from {a, root[:t0]} to root  (H = horizon)
└── ↻ t over 0:H until stop[t]
    └── root[t+1] = (root[t] + a / root[t]) / 2       (t ≥ t0-1)
        stop[t] = abs(root[t] * root[t] - a) < 1e-12  (loop stop)
```

Figure 5: Newton's iteration for the square root of `a`, one iterate per step, and the plan derived for the call. `horizon=100` bounds the loop, and `until=` stops it at the first step whose iterate squares to within 10^{-12} of `a`. The plan computes that condition at every step as a value `stop[t]` and ends the loop where it holds. For each of five values the call returns the same iterates as the equivalent `while` loop, and as many of them.

```
g = hp.Graph(backend="numpy")
noise = g.series(role="input")
pos = g.series(recurrence="transition")
recent = g.series()

# A walk, and at every step the mean of its last four positions.
pos[t+1]  << pos[t] + noise[t]
recent[t] << array.mean(pos[t-3:t+1], 0)

means = g.derive({noise: draws, pos[:t0]: [0.0]}, outputs=recent)
```

```
Program from {noise, pos[:t0]} to recent  (H = horizon)
└── ↻ t over 0:H
    └── pos[t+1] = pos[t] + noise[t]  (t ≥ t0-1)
        recent[t] = array.mean(pos[t-3:t+1], 0)
series   runs         holds     stores   read in   kept   builder
pos      [t0-1:]      5 steps   --       1         no     packed
recent   every step   all       all      --        yes    stepwise
```

Figure 6: A walk and the mean of its last four positions, and the plan at full detail, with the table of what each series costs that Section 3.1 explains. `pos[t-3:t+1]` names the four positions up to this step, and fewer before the fourth step. The loop holds five steps of `pos` at once: the four the mean reads, and `pos[t+1]`, which the same step writes. It keeps none of them past the call, since the call asks for `recent` alone.

```
# A rollout of several environments, stored (environments, steps, ...).
g = hp.Graph(backend=Backend("numpy", axis=1))
obs = g.series(role="input")

# The minibatch takes the steps named in picks, from every environment.
picks = g.atom(role="input")
minibatch = g.atom()
minibatch << hp.take_steps(obs, picks)

taken = g.derive({obs: rollout, picks: indices}, outputs=minibatch)
```

```
Program from {obs, picks} to minibatch  (storage: axis=1)
└── minibatch = take_steps(obs, picks)
```

Figure 7: A minibatch taken from a rollout of several environments stored with time second, and the plan derived for it. *take_steps* picks along the axis the storage declares time on, which the plan's header names, so `minibatch` holds the chosen steps of every environment, where a subscript on the array would pick environments.

```
g = hp.Graph(backend="numpy")
mean, std, z = g.atom(role="input")
posterior, prior, log_ratio, kl = g.atom()
posterior << dist.normal(mean, std)
prior     << dist.normal(array.zeros_like(mean), 1.0)
log_ratio << dist.log_prob(posterior, z) - dist.log_prob(prior, z)

# Two edges derive kl: in closed form, and averaged over the samples z.
kl << dist.kl_divergence(posterior, prior)
kl << array.mean(log_ratio, 0)
closed, sampled = g.producers(kl)

# Each call picks a route: one disables the sampled edge, the other requires it.
given = {mean: m, std: s, z: samples}
exact = g.derive(given, outputs=kl, disabled=[sampled])
estimate = g.derive(given, outputs=kl, required=[sampled])
```

```
Program from {mean, std, z (unread)} to kl
├── posterior = distributions.normal(mean, std)
│   prior = distributions.normal(array.zeros_like(mean), 1.0)
└── kl = distributions.kl_divergence(posterior, prior)
Excluded: kl ← array.mean(log_ratio, 0)
```

```
Program from {mean, std, z} to kl
├── posterior = distributions.normal(mean, std)
│   prior = distributions.normal(array.zeros_like(mean), 1.0)
├── log_ratio = distributions.log_prob(posterior, z) - distributions.log_prob(prior, z)
└── kl = array.mean(log_ratio, 0)  (required)
```

Figure 8: Two edges derive `kl`, the KL divergence of a Gaussian from a standard normal: one in closed form, one averaged over samples `z`. Declared together they make any question of `kl` ambiguous, and each call settles it: `disabled=` removes an edge from its search and `required=` puts one in its plan. Below, the plan of each call. The first lists the disabled edge and never reads `z`, and the second marks the edge it requires.

Table 2: The vocabulary a graph is written in, and the experiments whose graphs use each part. The body writes 24 of its 64 parts, marked †, and only the appendices write the other 40. Of those, the four largest that no experiment uses are written out in Figures 5, 6, 7 and 8. *All* names the eight graphs of Table 4. A printed plan uses the same notation for time: ⟳ *t over 0:H* is a loop whose step *t* runs from 0 to the step before the horizon `H`, ⟳ *t over :tT backward* one that runs from the step before *tT* down to 0, and a row marked (*t* ≥ *t0-1*) runs on fewer steps than its loop. Whether its stream is bound or global, a draw takes the stream's next numbers in the order the plan runs its edges, which is fixed for a graph and a question, as a loop's draws follow its code.

Written	Meaning	Used by
Values		
`E = g.atom(), x = g.series()`†	A value, and a value at every step, each named after the variable it is assigned to	All
`recurrence="transition"`†	Each step writes the next one, `x[t+1]`	GPT-2, PPO, DQN, nanoGPT-GRU
`recurrence="cumulative"`†	The series takes a bootstrap, and its own edge may read its earlier steps; a backward series needs it	DDPM, PPO, nanoGPT-GRU
`direction="backward"`†	The loop runs from the last step to the first	DDPM, PPO
`role="input"`†	The caller binds it, and no edge derives it	All

Table 2, continued

Written	Meaning	Used by
`role="effect"`†	It holds no value, and a call runs its edge by naming it in `run=`	All
`role="output"`	The caller may read it and no other edge may, so its own edge may read its earlier steps	None
`lifetime="keep"`, `"transient"`	Whether a value outlives the call that computed it	ResNet, DDPM
`supply="stream"`†	The caller hands an iterator, and each step pulls one item	ResNet, DDPM, VAE, GAN
`lifting="never"`†	Every edge on the series runs step by step, holding only the steps its reads need where a lift holds every step at once; set on a graph, a series or a call	ResNet, DDPM, DQN, VAE
`lifting="required"`	A call raises where an edge runs step by step for a reason a declaration could remove	None
`combine="sum"`, `x[t] += f(...)`	The value adds up its edges, each written `+=` or `-=`, and one under a condition joins only where a call pins it	PPO, VAE, GAN
Edges		
`@hp.rule`	An ordinary Python function as the operation of an edge	All
`hp.call(torch.optim.SGD, ...)`	A function nobody declared, called in place	ResNet, DDPM, DQN, VAE, GAN
`x[t].via(f(y[t]), after=fit[t-1])`†	The same edge with keywords; `after=` and `before=` order two edges no value connects	DDPM, DQN, VAE, GAN
`@hp.rule(broadcast=True)`	The function works elementwise, so one body serves a step and a whole series	GPT-2, DDPM, PPO, nanoGPT-GRU
`broadcast="elementwise"`, `removes=-1`	What the body does to its arguments' shapes: broadcasts them together, or removes an axis	DDPM, PPO
`@f.vectorized("torch", axes=1)`	A second body, over whole series	GPT-2, nanoGPT-GRU
`@hp.rule.vectorized("torch")`	A rule declared by its body over whole series alone, which also computes one step as a series one step long	nanoGPT-GRU, Figure 2
`folds={"prefix": "history"}`	The second body answers a running read, such as `x[:t+1]`, at every step at once	GPT-2, nanoGPT-GRU
`@hp.rule(time_axis=-2)`	Where the rule's bodies read time in a run of steps; the library moves it there from the axis the storage keeps it on	Figure 2
`carries=("hidden",)`	The second body carries a recurrence's state itself, so an edge reading its own past runs as one call	nanoGPT-GRU
`@hp.rule(outputs=6)`	How many values the function returns, so that a call can be unpacked	nanoGPT-GRU
`f(x[t]).returns(like=y[t])`	What one step of a call's result looks like, where no step runs to show it	nanoGPT-GRU
`@hp.dispatch`	One rule, with a body registered per backend	None
`@hp.rule(env=True)`	The body is also handed the time axis and the backend it runs over	None
`@hp.rule(side_effect=True)`†	The body acts on the world, so the plan never saves a call by hoisting, sharing or broadcasting it	All

Table 2, continued

Written	Meaning	Used by
`@hp.rule(traced=False)`	The body stays out of what a compiler traces	ResNet
`(images[t], labels[t]) << batch[t]`†	Several values from one call	All but GAN
Reads		
`x[:t0]` or `x.initial`, `x[tT:]` or `x.terminal`†	The bootstrap, the steps a series starts from, given before its own edge writes the rest: the first `t0` steps of a forward series, or the steps from `tT` to the end of a backward one	GPT-2, PPO, DQN, nanoGPT-GRU
`x[t-1].or_else(0.0)`†	A read before the first step, answered by a default	DDPM, nanoGPT-GRU
`x[t-3:t+1]`	A window of the last four steps, shorter at the start, as a slice clips at an array's ends. `x[t-1]`, in contrast, has no step to clip to at the first step, so it raises unless `.or_else` gives it a value	Figure 6
`x[-1]`, `x[1:]`, `x`†	A finished series, or part of it, read after its loop	Figure 1, GPT-2, ResNet, DDPM, PPO, nanoGPT-GRU
`x[::16]`†	Every sixteenth step, as an output or a watch	ResNet, DDPM
`hp.take_steps(obs, picks)`	The steps an index names, where the call computes or binds the index	Figure 7
`batch[t][0]`	A second bracket picks an element of one step's value	VAE, GAN
`hp.time_axis(term)`	The axis time runs along in a whole series, wherever the storage put it	DDPM
Targets that cover some of the steps		
`fit[7::8] << sgd_step(...)`†	An effect edge that runs on every eighth step	ResNet, DDPM, DQN
`v[:1] << f(...)`, `v[1:] << g(...)`	Two edges write one series, each over its own steps	DDPM
Modes		
`g.when(training)`, `g.when(~training)`, `g.condition()`, `assuming=[...]`†	Edges that join a derivation only when the call pins the condition, or its negation	All but DQN
`hp.provided(mask)`†	The same choice, made by whether the caller binds `mask`	DDPM
Questions		
`g.derive(inputs, outputs=..., run=[...])`†	What the caller holds, what it wants, and which effects run	All
`horizon=H`†	How many steps the loop runs	GPT-2, DDPM, PPO, DQN, nanoGPT-GRU
`until=done[t]`	The condition that stops the loop early. It is one boolean a step, so a loop over a batch of sequences stops for all of them at once	Figure 5
`disabled=[edge]`, `required=[edge]`	Edges this call may not use, or must	Figure 8
`g.scope(...)`, `scope.bind(...)`	Keeps what one call binds or names for the calls after it	ResNet, PPO
`scope.rebind(epoch_index, epoch)`	Replaces a value the scope holds, and drops what was derived from it. An optimizer changing a weight in place replaces nothing, so it drops nothing	ResNet
`name="rollout"`	A label for the call's program, where several are shown	ResNet, PPO, DQN, VAE, GAN, nanoGPT-GRU

Table 2, continued

Written	Meaning	Used by
`g.watch(loss[::16], watches=[...])`†	Reports a value to a writer at the steps its read names, while the call runs	ResNet, DDPM
`g.compile(...)`, `program.kernel()`†	The program without running it, and one function of it	Section 4
Compiling		
`compiler=...`	A compiler, which compiles the call's program	GPT-2, ResNet, DDPM, PPO, DQN
`TorchCompiler(cudagraph=True), static=True`	Records the compiled program into a CUDA graph, and holds each input at an address that does not move	PPO, in a mode no run here takes; in the compiler's own mode, ResNet
`unroll=32`	How many steps of a loop to trace into what the compiler sees	None
The graph as a value		
`g["score_fake"]`	Looks up a vertex by its name	DDPM, PPO, VAE, GAN
`g.producers(kl)`	The edges deriving a vertex, in the order they were declared	DDPM, PPO, VAE, GAN, nanoGPT-GRU, Figure 8
Storage		
`backend="numpy", "torch", "python"`†	Which arrays hold a series and which body of a rule runs, on a graph or per series	All; per series, all but GPT-2 and nanoGPT-GRU
`Backend("torch", axis=1)`†	Time runs along the stored array's second axis	GPT-2, DDPM, nanoGPT-GRU
`Backend("torch", device="cuda")`	Where a value the library creates lands when nothing it reads says	All but PPO and DQN
`transfer="store"`	A value bound on another device is copied to the one its series declares, where the default refuses it	GPT-2, ResNet, VAE, GAN, nanoGPT-GRU
`Backend("torch", record=True)`	A series of records, held as one array per field	PPO, VAE
`array.mean``(step_loss)`	`hyperpath.array`, array operations as rules declared to lift	All but GPT-2 and nanoGPT-GRU
`dist.sample``(`*`dist.normal`*`(m, s), rng)`	`hyperpath.distributions`, a distribution as a value and a draw as an edge	DDPM, PPO, DQN, VAE
`g.backend.rng(0)`, `shared_rng()`	A random stream the call binds like any input, its own or the one the process shares	DDPM, PPO, DQN, VAE
`seed_all(1)`	Seeds every backend's global generator at once	All

B THE PLANNER'S SEARCH

Section 4 outlines the planner's search. It picks one edge for each value it needs, groups the picked edges into blocks, and at six places follows only some of the ways of cutting them into blocks. This appendix defines when two cuts are the same plan and lists the six places and what each must guarantee. It then describes the two checks that back the planner's answer, so that the answer does not rest on any one place's own argument.

Terms. Figure 9 follows one question through the terms below.

Term	Meaning
Derivation	The edges the search picks, one producing edge for each value it needs, down to the values the caller holds
Block	A loop over the steps, running forward or backward, or a *one-shot* block, whose edges run once each
Candidate	A derivation's edges cut into a sequence of blocks, one result of the search
Normal form	The sequence of blocks a candidate reaches under the rewrites below
Plan	The candidates that share one normal form, each of them one *cut* of the plan, as Section 4 calls it
Rival	A second plan for the same question, which makes the question ambiguous
Branch	One option the search can follow where it has a choice to make
Site	One of six places where the search would branch only into different cuts of one plan
Collapse	Following some of a site's branches and dropping the rest
Representative	A branch a site keeps, standing for the branches it drops
Guard	A condition a site checks before it collapses

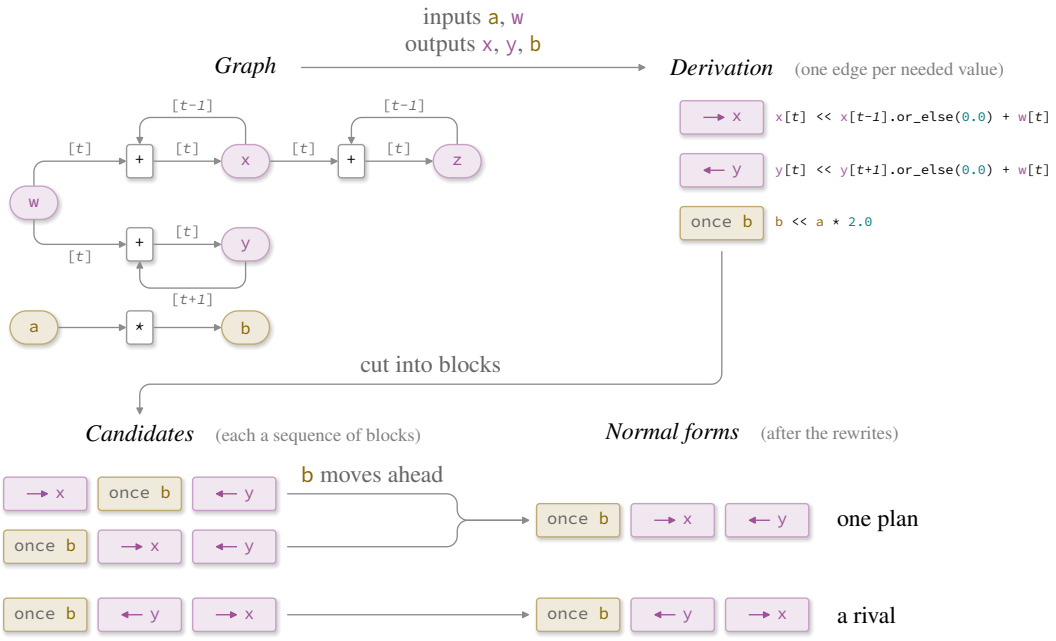


Figure 9: One question from its graph to its plans, a loop drawn with the direction it runs in. The graph computes x by a forward loop and y by a backward loop, each from the input w, z by a forward loop from x, and b once from the input a. With inputs a and w and outputs x, y and b, the search picks the three edges that produce the outputs and leaves z's out. The derivation draws each edge as the block that would run it alone, beside the edge as the graph declares it. The first two candidates are two cuts of one plan, since the first reaches the second once b moves ahead of the loop before it. The third runs the loops in the other order, and no rewrite changes it. It is a rival, so the planner refuses the question as ambiguous until `after=` or `before=` orders the loops.

The normal form. We rewrite a candidate's sequence of blocks in three ways.

1. Two adjacent blocks of one kind merge, two loops only where they run the same way and neither needs a series the other writes to be complete.
2. The part of a one-shot block that does not depend on the loop before it moves ahead of that loop.
3. A loop moves ahead of a one-shot block it does not depend on and merges with the loop before that block. It moves only where the two loops can merge.

We apply these rewrites over and over, in one fixed order, until none of them applies, and call the sequence reached the candidate's normal form. A question has as many plans as its candidates have normal forms, and the planner answers one and refuses more as ambiguous. The rewriting always stops, since each rewrite lowers the number of loops, or keeps it and moves one-shot work ahead of a loop, or changes neither and lowers the number of blocks. We do not prove that every order of the rewrites reaches the same normal form. If the order mattered, a candidate and one of its own rewrites could reach two normal forms, and the planner would count them as two plans and refuse the question as ambiguous. In Figure 9, the second rewrite moves `b` ahead of the loop before it.

Sites and obligations. Table 3 names the six sites A to F by the choice each one makes. At sites A to C we keep the branch that puts the most work into the loop being built, as Figure 10 shows for site C. At E and F we keep the one that computes a value in the block being built, and at D one of two orders that reach the same states. Sites A and B also keep the branches that run each series in the earliest loop that can run it. Dropping a branch is safe only where nothing is lost by it, so we hold every site to three obligations.

1. *Existence.* Every plan a dropped branch leads to is also reached through a kept branch, as another cut of the same plan. In Figure 10, the dropped branch leads to `x` and `z` in two loops, and the kept branch reaches the same plan with both in one.
2. *Survival.* No later step of the search discards the kept branch before it yields its plan.
3. *Rivals.* A site never drops a branch that leads to a rival, since the planner would then answer with one plan where the question has two.

Every site meets existence through a guard, and Table 3 gives what each does for the other two.

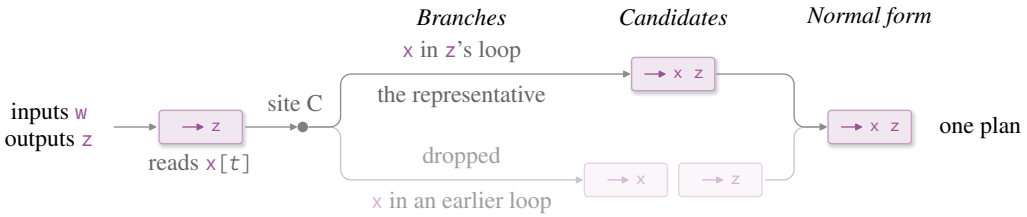


Figure 10: A site at work, on the graph of Figure 9. With input `w` and output `z`, the search picks `z`'s edge and starts a loop for it, which reads `x` one step at a time. At site C, `x` can run inside that loop or in an earlier one. The site collapses the two branches, keeping the one that puts the most work into each loop, its representative, and dropping the other. Its guard for existence checks that an earlier loop for `x` would merge with `z`'s, so the dropped branch reaches the same plan by another cut.

Table 3: The six collapse sites and how each meets survival and rivals. *Guard* means the site checks a condition that ensures the obligation, and *Partial* that it covers only some cases (at site C, that the kept loop is found, but not that it survives into a plan). A dash leaves it to the check below.

Site	Choice	Survival	Rivals
A	Which series asked of a loop it computes, and which the work after it derives	Partial	–
B	Which series the work after a loop needs that the loop computes itself	Partial	–
C	Leaving a value to an earlier loop	Partial	Guard
D	Whether a loop first adds an edge for a missing output or settles a missing input	Guard	Guard
E	Leaving a value to an earlier one-shot block	Guard	Guard
F	The same, for the one-shot block just ahead of a loop	Guard	Guard

The fallback and the check. Sites A to C meet survival only in part, and A and B do not argue rivals at all, so we let no answer rest on a guard alone. The search notes whether any collapse dropped a branch. Where none did its answer stands, and where one did, what follows depends on whether it found an answer.

A search that found nothing runs again without collapsing, and we call this second, complete search the *fallback*. A plan lost to a collapse then costs a second search instead of being reported as underivable. The fallback covers existence and survival where the lost plan was the only one. It costs far more than the collapsed search (Section 7).

Where the collapsed search found an answer, a second *check* runs behind it and looks for a rival the collapses may have hidden. It covers the rival obligation, and existence and survival where a lost plan had a rival. A rival can differ from the answer in two ways. It can run the same edges arranged into loops another way, as the third candidate of Figure 9 does by running its two loops in the other order. The check finds every such rival by laying out every arrangement of each derivation's edges into loops. We count it as a rival even where both arrangements compute the same values, as in Figure 9, since two arrangements can keep different values in memory, and the user picks one with `after=` or `before=`. Or a rival can run different edges, which it can only do where some value has two producing edges and it uses the other one. For every such value the check fixes one of its producers and searches again, once per combination. With every producer fixed only one derivation remains, so each search finds it, and laying out its loops finds every plan of that combination. Where one of them finds nothing, the complete search runs for that combination.

Testing the check. We test the check instead of proving it. We plan 2,428 test graphs twice, once collapsed, with the fallback and the check behind it, and once with the complete search, and the two give the same answer for every graph. Both share every step but the collapses, so a difference could only come from a collapse. We have not compared the complete search itself with a brute-force enumeration of plans. Three generators build 2,425 of the graphs, the first two from random seeds. The first (900 graphs) puts a value computed from a whole series between a forward and a backward loop, among a random number of other forward loops, and in half of its graphs gives a looped series a second producing edge. From the 450 without a second producer it builds 1,113 more, one for each forward series, each with a condition on that series that would stop the run early. No run meets the condition, since the test needs only a plan that can compute it. The second (400 graphs) builds one to three loops in either direction and values computed from them, a value's second producing edge sometimes sitting under a condition or carrying an `after=`, and adds an effect, a watch or an assumed condition to some graphs. It stores values as plain Python values or NumPy arrays. The third builds a fixed set of 12 graphs, in which a series can run in either of two blocks that a read of a whole series separates. We wrote the last three graphs by hand, each with a value that two edges produce inside one loop. None of the experiments' graphs is among them. The 18 derivations the reproductions of Section 5 ask for have no rival under the complete search either. Every change that can alter the planner's answers runs against these graphs before it lands.

C GENERATED CODE

Figure 11 is the loop body the library generates for the recurrence of Figure 1, under the generation question whose program Section 2 prints. We cut nothing from it but the error decoration its caption names.

```
# h[t] <- recur(Wr, h[t-1].or_else(0.0), x[t])
_v_edge3 = 1
_v_arg21 = _v_atom_Wr
if t < 1:
    if not _v_placement_device:
        _v_placement_device.append(_g_TorchSeriesOps_working_device([*_v_bound_values,
        ↪  _g_latest_reference(_v_builder_x)]))
    _v_h5 = 0.0
else:
    _v_h5 = _v_builder_container_h[t - 1 - _v_builder_h._pointer]
_v_arg22 = _v_h5
_v_arg23 = _v_builder_container_x[t - _v_builder_x._pointer]
_v_recur3 = _g_recur(_v_arg21, _v_arg22, _v_arg23)
_v_produced10 = _v_recur3
_v_h6 = _v_produced10
if not isinstance(_v_produced10, _g_Tensor) and _v_produced10.__class__ in _g_tuple2:
    if not _v_placement_device:
        _v_placement_device.append(_g_TorchSeriesOps_working_device([*_v_bound_values,
        ↪  _g_latest_reference(_v_builder_x)]))
    _v_h6 = _g_TorchPrimitives.fabricate_element(_v_produced10, None, _v_placement_device[0])
if _v_builder_h._append_direct:
    _v_element10 = _v_h6
else:
    if isinstance(_v_h6, _g_Tensor):
        _v_element10 = _v_h6
    else:
        _v_element10 = _g_TorchPrimitives.element_of(_v_h6, what='item')
    _v_ref10 = _v_builder_h._element_ref
    if _v_ref10 is None:
        _v_builder_h._element_ref = _v_element10
    else:
        if _v_element10.dtype is not _v_ref10.dtype:
            _v_element10 = _g_TorchPrimitives.cast_element(_v_element10, _v_ref10)
    if _v_builder_h._append_direct is None:
        _v_builder_h._append_direct = _v_element10 is _v_h6
_v_builder_container_h.append(_v_element10)
_v_builder_h._pointer += 1
if _v_probed_h:
    _v_coordinator._capture_index('h', t, _v_builder_h)
```

Figure 11: Generated source for one edge, with the per-operation error decoration removed by parsing; every line below the comment is one the library generated for this edge. The plan resolved each read at plan time to a container and an index, `.or_else` became an inline boundary test, and the append carries the series builder's own bookkeeping: an element reference, a dtype cast against it as writing into an array makes, and a pointer. The two lines we wrap find, once per call, the device on which the library places a value it creates itself, such as the fill. The last test is false in this call: it captures a step for a later read the plan could not place, which this question makes none of. The library mints readable names, `_v_` for a local and `_g_` for a global, and makes them unique across the whole namespace, which allows one generated function to be spliced into another.

Section 4 and Section 7 summarise two of the three measurements from the evaluation corpus (`experiments/kernel_cost`, claim C2), and unrolling's is reported here alone. We measure each against a comparand a reader can inspect, never against an asserted baseline. We take every guard count and compile time here from `torch.compile` in PyTorch 2.13.

The lift's shape checks. The unconditional shape checks around a lift cost a number of guard lines *fixed per compiled artifact*, and none per edge. We measured them against a copy of the library with those checks textually removed. They cost 2 guard lines on a pointwise lift, 7 on a rule declared elementwise, and 0 on element indexing, whose checks sit in backend code and so emit nothing into the traced graph. The counts come out identical at one lifted edge and at sixteen, and on two machines. Uncompiled, the same checks cost a small share of wall-clock that takes care to measure. On four dedicated cores of an AMD EPYC 9655, three seeds per backend agree to within

2 percentage points on every cell we measured. The checks cost 1 to 3% for one lifted edge or sixteen chained ones, and 5 to 6% over sixteen when each edge is a rule declared elementwise, with NumPy above PyTorch throughout. The checked and the unchecked library are two source trees and cannot share a process, so the spread within one run often contains zero, and these figures rest on the agreement across seeds.

The kernel. The kernel likewise costs a fixed number of guard lines and a small slope. We compare it with the same computation written out by hand and handed to the compiler directly. Against that, it costs 70 lines fixed and 6 per added edge on a chain of lifted edges, 83 and 6 on independent unrolled recurrences, and 90 and 12 on a shape with lifted edges either side of a recurrence.

Unrolling. In the compiled runs, only the loop estimating PPO's advantages is traced whole, since nothing in it stops a tracer and its number of steps is known when it is compiled. Every other loop calls a compiled region at every step, except PPO's loop over its environments, whose steps do too little to repay the call. Unrolling costs compile time where the checks and the kernel cost guard lines, so a budget on the steps traced is declared, and without one a loop whose number of steps is known is traced whole. Tracing a scan whole grows the compile with the horizon, from 8 s at $H = 64$ to 33 s at $H = 1024$ on an AMD EPYC 9655 with a cold compiler cache per measurement. The same scan split at a budget of 32 steps compiles in under 8 s at every horizon in that range. Each call then pays for the split according to how much the full unroll was winning. The traced prefix covers cap/H of the steps and the remainder runs behind the scan boundary, so on shapes whose full unroll wins a large factor the split call costs 10 to 25 times the unsplit one. A cap therefore pays off when a program compiles more often than it runs, and not otherwise.

D THE EXPERIMENTS

Table 4 lists the experiments of Section 5, each one model on one suite, with the claims it serves and the implementation it is compared against.

Table 4: The experiments, each with the claims it serves (C1, fidelity; C2, cost; C3, the size of a change), the implementation it is compared against, and the hardware and precision the graph and its reference both run on.

Model	Suite	C1	C2	C3	Compared against	Hardware
DDPM (Ho et al., 2020)	CIFAR-10 (Krizhevsky, 2009)	✓	✓	✓	`ddpm-torch` (Chen, 2022)	One H200, float32
GPT-2 124M (Radford et al., 2019)	FineWeb, HellaSwag (Penedo et al., 2024; Zellers et al., 2019)	✓	✓	✓	`llm.c`'s `train_gpt2.py` (Karpathy, 2024)	One H100, bfloat16
ResNet-50 (He et al., 2016)	ImageNet (Deng et al., 2009; Russakovsky et al., 2015)	✓	✓		torchvision's reference recipe (TorchVision maintainers and contributors, 2016)	One H100, float32
PPO (Schulman et al., 2017)	Classic control and MuJoCo (Towers et al., 2025; Todorov et al., 2012)	✓	✓	✓	CleanRL (Huang et al., 2022)	Two CPU cores, float32
DQN (Mnih et al., 2015)	Classic control (Towers et al., 2025)	✓	✓		CleanRL (Huang et al., 2022)	Two CPU cores, float32
VAE, GAN	None; synthetic batches			✓	PyTorch's examples (PyTorch contributors, 2017)	The CPU, float32
nanoGPT-GRU	None; synthetic batches		✓	✓	nanoGPT; for C2, its generation against a sampling loop we wrote (Karpathy, 2022)	The CPU, float32; for C2, one A100

For DDPM, Ho et al. (2020) report 3.17, which we quote as context alone, since FID depends on the feature network, the sample count and the reference statistics. DDPM's two step medians over 795,600 steps differ by 0.03 ms, and the two runs finish within half a percent of each other. The graph writes the weights' moving average as a new value at every step, where the reference updates one in place, and three copies stay alive through a step: the starting copy, which the training code keeps, and the previous step's average and the last checkpoint's, which the loop the library generates keeps until the same computation runs again, where a hand-written loop would already have freed them. The library already frees a value once nothing will read it again in the code that runs once per call, but not yet inside a loop over the steps; doing so there would release those two copies, which we have not measured. One-step tensors and the allocator's rounding make up the rest. A compiler takes its step to 0.77 of its own eager time, after an 80.1 s compile. In two of GPT-2's three pairs of full runs, the graph and llm.c ran side by side on one node, and over those the graph matches llm.c's throughput to 0.2%, each pair within 3%, at the same 29.0 GiB peak. Where a reference cannot run in our environment, we run it twice over the same seeds, unmodified at its own pinned versions for C1 and adapted for C2. ResNet-50's recipe trains on eight GPUs, so we run it once, on one, adapted from outside its files: each step accumulates eight micro-batches of 32, and batch norm's running statistics come from each step's first micro-batch, as the recipe's data-parallel broadcast takes them from one GPU; before any run, a check holds this form to data-parallel training itself. GPT-2's compiled runs spend 176 to 180 s on the compile and their first step together, and PPO's and DQN's 14 to 115 s on the compile alone; ResNet-50's runs record no compile time.

Table 5 gives each reinforcement learning cell of Table 1: each implementation's interquartile mean return with its 95% bootstrap interval, and the difference, graph minus CleanRL, with an interval of its own. HalfCheetah's runs end in one of two modes, which ten seeds are too few to compare, so we ran twenty more, a number fixed in advance; over thirty, each implementation reaches the higher mode in five. On MountainCar neither implementation ever solves the task.

Table 5: The reinforcement learning cells of Table 1, over ten seeds each and thirty on HalfCheetah: the interquartile mean (IQM) return of each implementation and of their difference, graph minus CleanRL, each with its 95% bootstrap interval.

		Graph		CleanRL		Graph − CleanRL	
Algorithm	**Environment**	IQM	95% interval	IQM	95% interval	IQM	95% interval
PPO	CartPole	488.09	[478.42, 494.28]	488.84	[482.23, 494.75]	−0.74	[−11.98, +8.77]
	Acrobot	−84.15	[−84.98, −83.71]	−84.04	[−84.52, −83.68]	−0.11	[−1.01, +0.55]
	MountainCar	−200.00	[−200.00, −200.00]	−200.00	[−200.00, −200.00]	0.00	[0.00, 0.00]
	HalfCheetah	1552.04	[1413.91, 1977.70]	1482.04	[1448.16, 1733.89]	+70.00	[−216.36, +487.47]
	Walker2d	3272.04	[2862.06, 3605.25]	3064.60	[2263.25, 3538.83]	+207.44	[−416.15, +1063.62]
	Hopper	2235.35	[1942.76, 2469.92]	2150.79	[1981.06, 2313.94]	+84.56	[−255.32, +381.77]
	InvertedPendulum	916.94	[885.46, 944.94]	929.64	[896.63, 964.20]	−12.70	[−59.68, +30.16]
	Humanoid	663.78	[630.80, 706.62]	650.41	[620.54, 676.35]	+13.37	[−27.96, +65.15]
DQN	CartPole	497.86	[493.20, 500.00]	496.05	[467.31, 499.82]	+1.81	[−4.04, +30.42]
	Acrobot	−89.04	[−90.04, −88.51]	−90.43	[−91.72, −88.57]	+1.39	[−0.74, +2.73]
	MountainCar	−197.71	[−199.99, −185.93]	−197.71	[−199.99, −185.93]	0.00	[−11.66, +11.81]

On PPO's three classic-control cells the graph's lead comes from how `ppo.py` does the work (Section 5.2). About half of it comes from the policy, which the file builds as a `torch.distributions.Categorical` at every step and minibatch, where the graph uses `hyperpath.distributions` (Appendix A). With only that swapped into the file, the graph takes 0.81 to 0.84 of its time, where it takes 0.67 to 0.72 of the unswapped file's. A smaller share comes from the critic, which the graph writes per step and the library lifts to one call per rollout.

The subsections below print the code behind Section 5.3, in the order that section takes them: the questions we ask of the diffusion model and of GPT-2, the variant of PPO for continuous actions, and the three demonstrations of the last two rows of Table 4. Each subsection starts on a page of its own, with a table of its questions, and ends with the declaration of the graph they are asked of. Each question opens on what it asks; a rule labelled PyTorch then carries how PyTorch answers it and the code that does, and one labelled Graph what the graph does and its code. A grey line opens each listing and names its file, and every line carries its number in that file, so a listing skips over a docstring, or over a comment other than the dividers a declaration is written in parts by, and still says where each line sits. A line an edit removes is marked `-` beside its number, and the code either side writes for a question is marked `+`. A line an edit keeps, printed for context or moved under a new `if` or `with`, keeps its number in the reference or the declaration, and is set grey and marked `=`. The tables count Python tokens. Appendix D.8 says how we chose the questions, and how we print, count and check each side.

Appendix D.7 prints the graph of DQN, one of the reproductions of Section 5, and has no table, since we ask DQN no question. Its replay buffer keeps past transitions and its target network keeps past weights, so how long DQN keeps each value is part of its algorithm.

D.1 DDPM

We compare against `ddpm-torch` (Chen, 2022), a PyTorch port of the code of Ho et al. (2020), and ask its CIFAR-10 model six questions. The reference answers the first three with methods of its own. We answer the other three with functions we wrote around its `q_sample` and `p_sample_step`. The graph's questions call the `q_sample` its own training step writes, and read its reverse step's edges, so neither side counts its model's code. The graph's t counts optimizer steps in its training step and diffusion timesteps in its schedule and reverse step. The two never meet in one derivation, which has one step index. Training binds the schedule's two tables whole and reads them at each image's drawn timestep. Before it records a row, the experiment holds each of the graph's six answers to the reference's method or to our function, on the CPU, and each agrees to the last bit.

	Tokens		
Question	**PyTorch**	**Graph**	**The graph's edges**
1. Progressive generation	248	72	No new edge
2. Negative log-likelihood	450	435	8 added, 1 guarded
3. DDIM	647	408	9 added, 9 guarded
4. Sampling from a noised image	112	100	2 added
5. Interpolation	148	160	3 added
6. Inpainting	139	208	2 added, 1 guarded
Total	1,744	1,383	

1. Progressive generation. Show the image the model predicts at regular steps along the reverse chain, the strip of predicted images Ho et al. (2020) show.

PyTorch

The reference answers it with `p_sample_progressive`, which writes `p_sample`'s loop out a second time to keep each prediction.

```
    diffusion.py: p_sample_progressive, p_sample's loop written out a second time
176 @torch.inference_mode()
177 def p_sample_progressive(
178         self, denoise_fn, shape, device=torch.device("cpu"), noise=None, pred_freq=10, seed=None):
179     B = (shape or noise.shape)[0]
180     t = torch.empty(B, dtype=torch.int64, device=device)
181     rng = None
182     if seed is not None:
183         rng = torch.Generator(device).manual_seed(seed)
184     if noise is None:
185         x_t = torch.empty(shape, device=device).normal_(generator=rng)
186     else:
187         x_t = noise.to(device)
188     L = self.timesteps // pred_freq
189     preds = torch.zeros((L, B) + shape[1:], dtype=torch.float32)
190     idx = L
191     for ti in range(self.timesteps - 1, -1, -1):
192         t.fill_(ti)
193         x_t, pred = self.p_sample_step(
194             denoise_fn, x_t, t, return_pred=True, generator=rng)
195         if (ti + 1) % pred_freq == 0:
196             idx -= 1
197             preds[idx] = pred.cpu()
198     return x_t.cpu(), preds

    selfcheck.py: the call, in strip_matches_upstream
374 return diffusion.p_sample_progressive(network, shape, pred_freq=every)
```

Graph

The graph answers it with its 29 edges and no new one, and it is asked through the call below.

```
    ddpm_hyperpath.py: Inference.progressive
594 return g.derive({**self.given(model), model.x_top: torch.randn(shape, device=self.device)},
595                 outputs=(model.x[0], model.pred_x0[every - 1::every]), horizon=self.timesteps,
596                 assuming=[model.mode("sample")])
```

2. Negative log-likelihood. Bound each image's negative log-likelihood in bits per dimension, the variational bound Ho et al. (2020) report.

PyTorch

The reference answers it with 3 methods, `calc_all_bpd` and the two terms it sums, though two of their statements raise as written: `calc_all_bpd` takes the batch's shape where its size is meant, and `_prior_bpd` hands `normal_kl` two floats where it takes tensors. The check runs the three with those two statements repaired.

```
    diffusion.py: _loss_term_bpd, one timestep's term: the decoder's likelihood and the divergence
203 def _loss_term_bpd(self, denoise_fn, x_0, x_t, t, clip_denoised, return_pred):
207     true_mean, _, true_logvar = self.q_posterior_mean_var(x_0=x_0, x_t=x_t, t=t)
208     model_mean, _, model_logvar, pred_x_0 = self.p_mean_var(
209         denoise_fn, x_t=x_t, t=t, clip_denoised=clip_denoised, return_pred=True)
210     kl = normal_kl(true_mean, true_logvar, model_mean, model_logvar)
211     kl = flat_mean(kl) / math.log(2.)
212     decoder_nll = discretized_gaussian_loglik(x_0, model_mean, log_scale=0.5 * model_logvar).neg()
213     decoder_nll = flat_mean(decoder_nll) / math.log(2.)
214     output = torch.where(t.to(kl.device) > 0, kl, decoder_nll)
215     return (output, pred_x_0) if return_pred else output

    diffusion.py: _prior_bpd, the last timestep's marginal against a standard normal
245 def _prior_bpd(self, x_0):
246     B, T = len(x_0), self.timesteps
247     T_mean, _, T_logvar = self.q_mean_var(
248         x_0=x_0, t=(T - 1) * torch.ones((B, ), dtype=torch.int64))
249     kl_prior = normal_kl(T_mean, T_logvar, mean2=0., logvar2=0.)
250     return flat_mean(kl_prior) / math.log(2.)

    diffusion.py: calc_all_bpd, a third loop over the timesteps, noising the images afresh at each timestep
252 def calc_all_bpd(self, denoise_fn, x_0, clip_denoised=True):
253     B, T = x_0.shape, self.timesteps
254     t = torch.empty([B, ], dtype=torch.int64)
255     losses = torch.zeros([B, T], dtype=torch.float32)
256     mses = torch.zeros([B, T], dtype=torch.float32)
257
258     for ti in range(T - 1, -1, -1):
259         t.fill_(ti)
260         x_t = self.q_sample(x_0, t=t)
261         loss, pred_x_0 = self._loss_term_bpd(
262             denoise_fn, x_0, x_t=x_t, t=t, clip_denoised=clip_denoised, return_pred=True)
263         losses[:, ti] = loss
264         mses[:, ti] = flat_mean((pred_x_0 - x_0).pow(2))
265
266     prior_bpd = self._prior_bpd(x_0)
267     total_bpd = torch.sum(losses, dim=1) + prior_bpd
268     return total_bpd, losses, prior_bpd, mses

    selfcheck.py: the call, in likelihood_matches_upstream
435 return diffusion.calc_all_bpd(network, images)
```

Graph

The graph adds eight edges to its 29 and moves one of those under a condition, in the code below, and it is asked through the call after it.

```
    ddpm_hyperpath.py: normal_kl
126 + normal_kl = hp.rule(reference_normal_kl, broadcast="elementwise")
    ddpm_hyperpath.py: discretized_gaussian_loglik
129 + discretized_gaussian_loglik = hp.rule(reference_discretized_gaussian_loglik, broadcast="elementwise")
    ddpm_hyperpath.py: in_bits
134 + def in_bits(nats):
136 +     return array.mean(nats, axis=(1, 2, 3)) / math.log(2.0)
    ddpm_hyperpath.py: DDPM.with_likelihood
365 + image = g.atom(role="input")
366 + level_noise, true_mean = g.series(recurrence="cumulative", direction="backward")
367 + term, mse = g.series(
368 +     recurrence="cumulative", direction="backward", backend=Backend("torch", device=config.device,
    ↪  axis=1))
369 + prior_bpd, total_bpd = g.atom()
370 +
371 + with g.when(mode == "sample"):
338 =     noisy[t] << x[t + 1].or_else(x_top)
373 + with g.when(mode == "likelihood"):
375 +     level_noise[t].via(dist.sample(dist.standard_normal(like=image), rng), lifting="never")
376 +     noisy[t] << q_sample(image, t, level_noise[t])
377 +
378 + true_mean[t] << posterior_mean_coef1[t] * image + posterior_mean_coef2[t] * noisy[t]
379 + term[:1] << in_bits(-discretized_gaussian_loglik(image, posterior_mean[t], 0.5 * fixed_model_logvar[t]))
380 + term[1:] << in_bits(normal_kl(true_mean[t], array.astype(array.log(posterior_var[t]), "float32"),
381 +                               posterior_mean[t], fixed_model_logvar[t]))
382 + mse[t] << array.mean((pred_x0[t] - image) ** 2, axis=(1, 2, 3))
383 +
386 + zero = array.zeros(())
387 + prior_bpd << in_bits(normal_kl(array.sqrt(alphas_bar[-1]) * image,
388 +                                array.astype(array.log(1 - alphas_bar[-1]), "float32"), zero, zero))
389 + total_bpd << array.sum(term, axis=hp.time_axis(term)) + prior_bpd

    ddpm_hyperpath.py: Inference.likelihood
606 return g.derive({**self.given(model), model.image: images},
607                 outputs=(model.total_bpd, model.term, model.prior_bpd, model.mse),
608                 horizon=self.timesteps, assuming=[model.mode("likelihood")])
```

3. DDIM. Sample by DDIM over a subsequence of the timesteps, with eta scaling the variance.

PyTorch

The reference answers it with `ddim.py`'s `DDIM`, in 2 places: a subclass of the model that computes every constant of the schedule again over the subsequence, and a fourth loop over the timesteps.

```
   ddim.py: DDIM.__init__, every constant of the schedule computed again over the subsequence
48 def __init__(self, betas, model_mean_type, model_var_type, loss_type, eta, subsequence):
49     super().__init__(betas, model_mean_type, model_var_type, loss_type)
50     self.eta = eta
51     self.subsequence = subsequence
52
53     eta2 = eta ** 2
54     try:
55         assert not (eta2 != 1. and model_var_type != "fixed-small"), \
56             "Cannot use DDIM (eta < 1) with var type other than `fixed-small`"
57     except AssertionError:
59         self.model_var_type = "fixed-small"
60
61     self.alphas_bar = self.alphas_bar[subsequence]
62     self.alphas_bar_prev = torch.cat([torch.ones(1, dtype=torch.float64), self.alphas_bar[:-1]], dim=0)
63     self.alphas = self.alphas_bar / self.alphas_bar_prev
64     self.betas = 1. - self.alphas
65     self.sqrt_alphas_bar_prev = torch.sqrt(self.alphas_bar_prev)
66
69     self.sqrt_alphas_bar = torch.sqrt(self.alphas_bar)
70     self.sqrt_one_minus_alphas_bar = torch.sqrt(1. - self.alphas_bar)
71
72     self.posterior_var = self.betas * (1. - self.alphas_bar_prev) / (1. - self.alphas_bar) * eta2
73     self.posterior_logvar_clipped = torch.log(torch.cat([
74         self.posterior_var[[1]], self.posterior_var[1:]]).clip(min=1e-20))
75
77     self.sqrt_recip_alphas_bar = torch.sqrt(1. / self.alphas_bar)
78     self.sqrt_recip_m1_alphas_bar = torch.sqrt(1. / self.alphas_bar - 1.)
79
81     self.posterior_mean_coef2 = torch.sqrt(
82         1 - self.alphas_bar - eta2 * self.betas
83     ) * torch.sqrt(1 - self.alphas_bar_prev) / (1. - self.alphas_bar)
84     self.posterior_mean_coef1 = self.sqrt_alphas_bar_prev * \
85                                 (1. - torch.sqrt(self.alphas) * self.posterior_mean_coef2)
86
88     self.fixed_model_var, self.fixed_model_logvar = {
89         "fixed-large": (
90             self.betas, torch.log(torch.cat([self.posterior_var[[1]],
  ↪  self.betas[1:]]).clip(min=1e-20))),
91         "fixed-small": (self.posterior_var, self.posterior_logvar_clipped)
92     }[self.model_var_type]
93
94     self.subsequence = torch.as_tensor(subsequence)

   ddim.py: DDIM.p_sample, a fourth loop over the timesteps, over the subsequence
96 @torch.inference_mode()
97 def p_sample(self, denoise_fn, shape, device=torch.device("cpu"), noise=None, seed=None):
98     S = len(self.subsequence)
99     B, *_ = shape
100    subsequence = self.subsequence.to(device)
101    _denoise_fn = lambda x, t: denoise_fn(x, subsequence.gather(0, t))
102    t = torch.empty((B, ), dtype=torch.int64, device=device)
103    rng = None
104    if seed is not None:
105        rng = torch.Generator(device).manual_seed(seed)
106    if noise is None:
107        x_t = torch.empty(shape, device=device).normal_(generator=rng)
108    else:
109        x_t = noise.to(device)
110    for ti in range(S - 1, -1, -1):
111        t.fill_(ti)
112        x_t = self.p_sample_step(_denoise_fn, x_t, t, generator=rng)
113    return x_t

   selfcheck.py: the call, in ddim_matches_upstream
464 reference = DDIM(diffusion.betas, "eps", "fixed-small", "mse", eta=eta, subsequence=subsequence)
465 return reference.p_sample(network, shape=shape, noise=torch.randn(shape))
```

Graph

The graph adds nine edges to its 29 and moves nine of those under a condition, in the code below, and it is asked through the two derivations after it.

```
    ddpm_hyperpath.py: DDPM.with_ddim
261 = previous_alphas_bar = alphas_bar[t - 1].or_else(1.0)
412 + sampler = g.condition()
413 + eta, whole_alphas_bar = g.atom(role="input")
414 + timestep = g.series(role="input")
415 +
416 + with g.when(sampler == "ddpm"):
255 =     betas << array.linspace(config.beta_start, config.beta_end, config.timesteps)
256 =     alphas[t] << 1 - betas[t]
257 =     alphas_bar[t] << array.prod(alphas[:t + 1])
263 =     posterior_var[t] << betas[t] * (1.0 - previous_alphas_bar) / (1.0 - alphas_bar[t])
264 =     posterior_mean_coef1[t] << betas[t] * array.sqrt(previous_alphas_bar) / (1.0 - alphas_bar[t])
265 =     posterior_mean_coef2[t] << (array.sqrt(alphas[t]) * (1.0 - previous_alphas_bar)
266 =                                / (1.0 - alphas_bar[t]))
269 =     fixed_model_logvar[:1] << array.astype(array.log(posterior_var[t + 1]), "float32")
270 =     fixed_model_logvar[1:] << array.astype(array.log(betas[t]), "float32")
332 =     eps[t] << active_network(noisy[t], array.full(noisy[t].shape[:1], t))
427 +
428 + with g.when(sampler == "ddim"):
429 +     alphas_bar[t] << whole_alphas_bar[timestep[t]]
430 +     alphas[t] << alphas_bar[t] / previous_alphas_bar
431 +     betas[t] << 1 - alphas[t]
432 +     posterior_var[t] << betas[t] * (1.0 - previous_alphas_bar) / (1.0 - alphas_bar[t]) * eta ** 2
433 +
434 +     posterior_mean_coef2[t] << (array.sqrt(1 - alphas_bar[t] - eta ** 2 * betas[t])
435 +                                 * array.sqrt(1 - previous_alphas_bar) / (1.0 - alphas_bar[t]))
436 +     posterior_mean_coef1[t] << (array.sqrt(previous_alphas_bar)
437 +                                 * (1.0 - array.sqrt(alphas[t]) * posterior_mean_coef2[t]))
438 +
439 +     fixed_model_logvar[:1] << array.astype(array.log(array.clip(posterior_var[t + 1], min=1e-20)),
440 +                                            "float32")
441 +     fixed_model_logvar[1:] << array.astype(array.log(array.clip(posterior_var[t], min=1e-20)),
442 +                                            "float32")
443 +     eps[t] << active_network(noisy[t], array.full(noisy[t].shape[:1], timestep[t]))

    ddpm_hyperpath.py: Inference.ddim
617 whole_alphas_bar = g.derive({}, outputs=model.alphas_bar, horizon=self.timesteps,
618                               assuming=[model.sampler("ddpm")])
619 return g.derive({**self.given(model), model.whole_alphas_bar: whole_alphas_bar,
620                  model.timestep: subsequence.to(self.device), model.eta: eta,
621                  model.x_top: torch.randn(shape, device=self.device)},
622                 outputs=model.x[0], horizon=len(subsequence),
623                 assuming=[model.mode("sample"), model.sampler("ddim")])
```

4. Sampling from a noised image. Denoise an image noised part-way up the chain, starting the reverse chain from that step.

PyTorch

PyTorch needs a function of its own, which noises the image with `q_sample` and writes `p_sample`'s loop out once more over the shorter range.

```
    new function: from_noised
20 + def from_noised(diffusion, network, images, start):
22 +     t = torch.full((images.shape[0],), start - 1, dtype=torch.int64, device=images.device)
23 +     x_t = diffusion.q_sample(images, t, noise=torch.randn_like(images))
24 +     for ti in range(start - 1, -1, -1):
25 +         t.fill_(ti)
26 +         x_t = diffusion.p_sample_step(network, x_t, t)
27 +     return x_t

    selfcheck.py: the call, in from_noised_matches_by_hand
486 return solutions.from_noised(diffusion, network, images, start)
```

Graph

The graph adds two edges to its 29, in the code below, and it is asked through the call after it.

```
    ddpm_hyperpath.py: DDPM.with_noised_start
454 + source = g.atom(role="input")
455 + source_noise = g.atom()
456 + source_noise.via(dist.sample(dist.standard_normal(like=source), rng), after=g["active_network"])
457 + x_top << q_sample(source, -1, source_noise)

    ddpm_hyperpath.py: Inference.from_noised
629 return g.derive({**self.given(model), model.source: images},
630                 outputs=model.x[0], horizon=start, assuming=[model.mode("sample")])
```

5. Interpolation. Interpolate between two images by noising both part-way, mixing them and denoising the mixture, as Ho et al. (2020) do.

PyTorch

PyTorch needs a function of its own, which noises both images, mixes them, and writes `p_sample`'s loop out once more over the shorter range.

```
    new function: interpolate
 30 + def interpolate(diffusion, network, source, other, mix, start):
 33 +     t = torch.full((source.shape[0],), start - 1, dtype=torch.int64, device=source.device)
 34 +     x_t = ((1 - mix) * diffusion.q_sample(source, t, noise=torch.randn_like(source))
 35 +            + mix * diffusion.q_sample(other, t, noise=torch.randn_like(other)))
 36 +     for ti in range(start - 1, -1, -1):
 37 +         t.fill_(ti)
 38 +         x_t = diffusion.p_sample_step(network, x_t, t)
 39 +     return x_t

    selfcheck.py: the call, in interpolation_matches_by_hand
503 return solutions.interpolate(diffusion, network, source, other, mix, start)
```

Graph

The graph adds three edges to its 29, in the code below, and it is asked through the call after it.

```
    ddpm_hyperpath.py: DDPM.with_interpolation
467 + source, other, mix = g.atom(role="input")
468 + source_noise, other_noise = g.atom()
469 + source_noise.via(dist.sample(dist.standard_normal(like=source), rng), after=g["active_network"])
470 + other_noise.via(dist.sample(dist.standard_normal(like=other), rng), after=source_noise)
471 + x_top << (1 - mix) * q_sample(source, -1, source_noise) + mix * q_sample(other, -1, other_noise)

    ddpm_hyperpath.py: Inference.interpolate
636 return g.derive({**self.given(model), model.source: source, model.other: other, model.mix: mix},
637                 outputs=model.x[0], horizon=start, assuming=[model.mode("sample")])
```

6. Inpainting. Fill in the pixels a mask leaves out of an image, putting the known pixels back after every step of the reverse chain.

PyTorch

PyTorch needs a function of its own, which writes `p_sample`'s loop out once more, noising the known pixels to each step and putting them back after it.

```
    new function: inpaint
 42 + def inpaint(diffusion, network, source, mask):
 44 +     t = torch.empty((source.shape[0],), dtype=torch.int64, device=source.device)
 45 +     x_t = torch.randn_like(source)
 46 +     for ti in range(diffusion.timesteps - 1, -1, -1):
 47 +         t.fill_(ti)
 48 +         noise = torch.randn_like(source)
 49 +         known = source if ti == 0 else diffusion.q_sample(source, t - 1, noise=noise)
 50 +         x_t = mask * known + (1 - mask) * diffusion.p_sample_step(network, x_t, t)
 51 +     return x_t

    selfcheck.py: the call, in inpainting_matches_by_hand
524 return solutions.inpaint(diffusion, network, source, mask)
```

Graph

The graph adds two edges to its 29 and moves one of those under a condition, in the code below, and it is asked through the call after it.

```
    ddpm_hyperpath.py: DDPM.with_inpainting
261 = previous_alphas_bar = alphas_bar[t - 1].or_else(1.0)
487 + source, mask = g.atom(role="input")
488 + known_noise = g.series(recurrence="cumulative", direction="backward")
489 + known_noise[t].via(dist.sample(dist.standard_normal(like=source), rng), lifting="never")
490 + with g.when(~hp.provided(mask)):
339 =     x[t] << reverse_draw(posterior_mean[t], fixed_model_logvar[t], rng)
492 + with g.when(hp.provided(mask)):
493 +     drawn = reverse_draw(posterior_mean[t], fixed_model_logvar[t], rng)
494 +     known = array.sqrt(previous_alphas_bar) * source + array.sqrt(1.0 - previous_alphas_bar) *
    ↪  known_noise[t]
495 +     x[t] << mask * known + (1 - mask) * drawn

    ddpm_hyperpath.py: Inference.inpaint
643 return g.derive({**self.given(model), model.source: source, model.mask: mask,
644                 model.x_top: torch.randn(source.shape, device=self.device)},
645                outputs=model.x[0], horizon=self.timesteps, assuming=[model.mode("sample")])
```

Declaration

```
    ddpm_hyperpath.py: DDPM.declare, the model every question is asked of
234 g = hp.Graph(backend=Backend("torch", device=config.device))
235 mode = g.condition()
236 network = g.atom(role="input")
237 active_network = g.atom()
238
239 with g.when(mode == "train"):
240     active_network << in_mode(network, train=True)
241 with g.when(mode != "train"):
242     active_network << in_mode(network, train=False)
243
244 # --- the noise schedule, one value per timestep
251 betas = g.series(backend=Backend("torch", device=config.device, dtype=torch.float64))
252 alphas, alphas_bar, posterior_var = g.series()
253 posterior_mean_coef1, posterior_mean_coef2, fixed_model_logvar = g.series()
254
255 betas << array.linspace(config.beta_start, config.beta_end, config.timesteps)
256 alphas[t] << 1 - betas[t]
257 alphas_bar[t] << array.prod(alphas[:t + 1])
258
261 previous_alphas_bar = alphas_bar[t - 1].or_else(1.0)
262
263 posterior_var[t] << betas[t] * (1.0 - previous_alphas_bar) / (1.0 - alphas_bar[t])
264 posterior_mean_coef1[t] << betas[t] * array.sqrt(previous_alphas_bar) / (1.0 - alphas_bar[t])
265 posterior_mean_coef2[t] << (array.sqrt(alphas[t]) * (1.0 - previous_alphas_bar)
266                             / (1.0 - alphas_bar[t]))
267
269 fixed_model_logvar[:1] << array.astype(array.log(posterior_var[t + 1]), "float32")
270 fixed_model_logvar[1:] << array.astype(array.log(betas[t]), "float32")
271
```

```
# --- a training step
generator, checkpoint_path = g.atom(role="input")

optimizer, lr_scheduler = g.atom()
parameters, initial = g.atom()
sqrt_alphas_bar, sqrt_one_minus_alphas_bar = g.atom()
with g.when(mode == "init"):
    optimizer << hp.call(torch.optim.Adam, network.parameters(), lr=config.lr,
                         betas=(config.beta1, config.beta2))
    lr_scheduler << warmup_over(optimizer, config.warmup)
    parameters << hp.call(trainable, network)
    initial << hp.call(cloned, network)

sqrt_alphas_bar << array.reshape(array.astype(array.sqrt(alphas_bar), "float32"), (-1, 1, 1, 1))
sqrt_one_minus_alphas_bar << array.reshape(array.astype(array.sqrt(1.0 - alphas_bar), "float32"),
                                           (-1, 1, 1, 1))

def q_sample(x_0, step, noise):
    return sqrt_alphas_bar[step] * x_0 + sqrt_one_minus_alphas_bar[step] * noise

batch = g.series(role="input", supply="stream")
steps, noise, noised, predicted, loss, stepped, averaged = g.series()
shadow = g.series(recurrence="cumulative", backend="python", lifetime="transient")
checkpointed = g.series(role="effect")

steps[t].via(dist.sample(dist.expand(dist.integers(0, config.timesteps), config.batch_size), generator),
             lifting="never")
noise[t].via(dist.sample(dist.standard_normal(like=batch[t]), generator), after=steps[t])
noised[t] << q_sample(batch[t], steps[t], noise[t])

predicted[t].via(active_network(noised[t], steps[t]), after=stepped[t - 1])
loss[t] << array.mean(array.mean((noise[t] - predicted[t]) ** 2, axis=(1, 2, 3)))
stepped[t] << optimize(network, optimizer, lr_scheduler, loss[t], config.grad_norm)
(shadow[t], averaged[t]) << averaged_weights(shadow[t - 1].or_else(initial), parameters,
                                             config.ema_decay, t, stepped[t])

every = config.checkpoint_every
checkpointed[every - 1::every] << save(checkpoint_path, network, optimizer, shadow[t],
                                       config.ema_decay, lr_scheduler, t, config.steps_per_epoch)

# --- one reverse step, from the image at the top down the chain
rng = g.atom(role="input")
x_top = g.atom()
x, noisy, eps, pred_x0, posterior_mean = g.series(recurrence="cumulative", direction="backward")

eps[t] << active_network(noisy[t], array.full(noisy[t].shape[:1], t))
pred_x0[t] << array.clip(array.sqrt(1.0 / alphas_bar[t]) * noisy[t]
                         - array.sqrt(1.0 / alphas_bar[t] - 1.0) * eps[t], -1.0, 1.0)
posterior_mean[t] << posterior_mean_coef1[t] * pred_x0[t] + posterior_mean_coef2[t] * noisy[t]

noisy[t] << x[t + 1].or_else(x_top)
x[t] << reverse_draw(posterior_mean[t], fixed_model_logvar[t], rng)
```

D.2 GPT-2

We compare against `train_gpt2.py` from llm.c (Karpathy, 2024) and ask GPT-2 124M four questions interpretability research asks of it. The file answers none of them, and `GPT.forward` keeps no intermediate, so our functions reach inside it with forward hooks. The graph's declaration writes each token's loss, `nll`, because our evaluation scores HellaSwag from it, and the in-context score reads that value with no new edge. Before it records a row, the experiment asks each question both ways of a freshly initialised model on the CPU. The two lenses and the in-context score agree to the last bit. Activation patching agrees to within 9.5×10^{-7}: past the last block's attention the graph computes the last position alone where `GPT.forward` computes every position, and a float32 product rounds by how many rows it is handed. Asked only for the logits at the last position, the graph computes everything after the last block's attention for that one position, where llm.c narrows only its output layer, with an `else:` branch written into `GPT.forward`. We also answer with NNsight (Fiotto-Kaufman et al., 2025) every question in this appendix whose PyTorch answer needs a hook, and only these first three need one. The GAN's feature matching reaches the discriminator's features by cutting its `nn.Sequential`, and the VAE's questions call the model's own `encode` and `decode`. Each NNsight answer follows the hook answer under a rule of its own and counts in the table's NNsight column, and the experiment holds it to the graph's answer as it holds the hook answer. The hook, NNsight and graph answers all fit the Jacobian lens with the same 236 tokens, which the table counts once, in its first column, and the graph's fitting edge calls. The graph trains in two kinds of call, which the experiment's driver makes in order: one per micro-batch asks for `share`, whose edge runs that micro-batch's backward pass, and one after the last asks for `norm` and runs `fit`. So the clip follows every backward pass although no edge orders the two.

	Tokens				
Question	**All three**	**PyTorch**	**NNsight**	**Graph**	**The graph's edges**
1. Logit lens	–	154	117	108	13 added
2. Jacobian lens	236	244	186	211	15 added
3. Activation patching	–	177	119	131	No new edge
4. In-context score	–	100	–	46	No new edge
Total	236	675		496	

1. Logit lens. Read every residual stream out through the logit lens of nostalgebraist (2020), the final norm and the output projection, and score it against the next token.

PyTorch

PyTorch needs a function of its own, which catches each block's input with a forward hook, since `GPT.forward` overwrites its `x` block by block.

```
    new function: logit_lens
21 + def logit_lens(model, tokens, targets):
23 +     streams = []
24 +     consumers = [*model.transformer.h, model.transformer.ln_f]
25 +     hooks = [module.register_forward_pre_hook(lambda _module, args: streams.append(args[0]))
26 +              for module in consumers]
27 +     try:
28 +         model(tokens)
29 +     finally:
30 +         for hook in hooks:
31 +             hook.remove()
32 +     losses = []
33 +     for stream in streams:
34 +         logits = model.lm_head(model.transformer.ln_f(stream))
35 +         losses.append(F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1),
36 +                                       ignore_index=-1))
37 +     return tuple(losses)

    questions.py: the call, in same_answers
109 lenses = solutions.logit_lens(model, tokens, targets)
```

NNsight

PyTorch with NNsight needs a function of its own, which reads each block's input and the final norm's inside a trace of `GPT.forward`, unchanged.

```
    new function: logit_lens
 29 + def logit_lens(gpt, tokens, targets):
 35 +     with gpt.trace(tokens):
 36 +         losses = nnsight.save([])
 37 +         for module in [*gpt.transformer.h, gpt.transformer.ln_f]:
 38 +             logits = gpt.lm_head(gpt.transformer.ln_f(module.input))
 39 +             losses.append(F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1),
 40 +                                           ignore_index=-1))
 41 +     return tuple(losses)

    nnsight_questions.py: the call, in same_answers
 60 lenses = nnsight_solutions.logit_lens(gpt, tokens, targets)
```

Graph

The graph adds thirteen edges to its 37, in the code below, and it is asked through the call after it.

```
    gpt2_hyperpath.py: GPT2.with_logit_lens
304 + lens = tuple(self.graph.atom(f"lens{index}") for index in range(len(self.stream)))
305 + for loss_there, residual in zip(lens, self.stream, strict=True):
306 +     loss_there << lm_loss(pointwise(self.lm_head, pointwise(self.ln_f, residual)), self.targets)

    gpt2_hyperpath.py: Inference.logit_lens
 468 return g.derive({**self.given, gpt2.x: tokens, gpt2.targets: targets}, outputs=gpt2.lens,
 469                 assuming=self.evaluating)
```

2. Jacobian lens. Read every residual stream out through the Jacobian lens, or J-lens, of Gurnee et al. (2026), which carries each stream into the last by a Jacobian averaged over a corpus of prompts and reads it out as the logit lens does. Sparse nonnegative combinations of the lens's vectors make up what that paper calls the J-space.

All three

All three answers use this code. It is written where PyTorch's answer has it, counted in the first column above, and NNsight's answer and the graph's edges call it.

```
    new function: averaged_jacobians, the fit every answer uses
 40 + def averaged_jacobians(*streams, sinks=16, rows=64):
 49 +     final = streams[-1]
 50 +     width = final.size(-1)
 51 +     counted = torch.zeros(final.size(1), dtype=final.dtype, device=final.device)
 52 +     counted[sinks:-1] = 1.0
 53 +     jacobians = final.new_zeros(len(streams), width, width)
 54 +     identity = torch.eye(width, dtype=final.dtype, device=final.device)
 55 +     for first in range(0, width, rows):
 56 +         features = identity[first:first + rows, None, None, :]
 57 +         cotangent = (features * counted[None, None, :, None]).expand(-1, final.size(0), -1, -1)
 58 +         gradients = torch.autograd.grad(final, streams, grad_outputs=cotangent,
 59 +                                         retain_graph=True, is_grads_batched=True)
 60 +         for jacobian, gradient in zip(jacobians, gradients, strict=True):
 61 +             jacobian[first:first + rows] = gradient[:, :, sinks:-1].mean(dim=(1, 2))
 62 +     return jacobians
```

PyTorch

PyTorch needs a function of its own beside the fit every answer uses, which catches every stream with a forward hook while autograd records and catches them again to read them out. Evaluating `GPT.forward` records no gradient, and it keeps no intermediate to differentiate.

new function: jacobian_lens

```
+ def jacobian_lens(model, prompts, tokens, targets):
+     consumers = [*model.transformer.h, model.transformer.ln_f]
+     fitting = []
+     hooks = [module.register_forward_pre_hook(lambda _module, args: fitting.append(args[0]))
+              for module in consumers]
+     try:
+         model.eval()
+         with torch.enable_grad():
+             model(prompts)
+     finally:
+         for hook in hooks:
+             hook.remove()
+     jacobians = averaged_jacobians(*fitting)
+     streams = []
+     hooks = [module.register_forward_pre_hook(lambda _module, args: streams.append(args[0]))
+              for module in consumers]
+     try:
+         model(tokens)
+     finally:
+         for hook in hooks:
+             hook.remove()
+     losses = []
+     for stream, jacobian in zip(streams, jacobians, strict=True):
+         logits = model.lm_head(model.transformer.ln_f(stream @ jacobian.mT))
+         flat = logits.view(-1, logits.size(-1))
+         losses.append(F.cross_entropy(flat, targets.view(-1), ignore_index=-1))
+     return tuple(losses)
```

questions.py: the call, in same_answers

```
jacobian_lenses = solutions.jacobian_lens(model, prompts, tokens, targets)
```

NNsight

PyTorch with NNsight needs a function of its own beside the fit printed above, which all three answers use. It reads every stream inside a trace while autograd records, and again inside a second trace to carry each by its Jacobian.

new function: jacobian_lens

```
+ def jacobian_lens(gpt, prompts, tokens, targets):
+     consumers = [*gpt.transformer.h, gpt.transformer.ln_f]
+     gpt.eval()
+     with torch.enable_grad(), gpt.trace(prompts):
+         fitting = nnsight.save([module.input for module in consumers])
+     jacobians = solutions.averaged_jacobians(*fitting)
+     with gpt.trace(tokens):
+         losses = nnsight.save([])
+         for module, jacobian in zip(consumers, jacobians, strict=True):
+             logits = gpt.lm_head(gpt.transformer.ln_f(module.input @ jacobian.mT))
+             flat = logits.view(-1, logits.size(-1))
+             losses.append(F.cross_entropy(flat, targets.view(-1), ignore_index=-1))
+     return tuple(losses)
```

nnsight_questions.py: the call, in same_answers

```
jacobian_lenses = nnsight_solutions.jacobian_lens(gpt, prompts, tokens, targets)
```

Graph

The graph adds fifteen edges to its 37, in the code below, and it is asked through the two derivations after it.

```
    gpt2_hyperpath.py: averaged_jacobians
194 + averaged_jacobians = hp.rule(solutions.averaged_jacobians)
    gpt2_hyperpath.py: GPT2.with_jacobian_lens
317 + differentiating = g.condition()
318 + with g.when(differentiating):
319 +     self.active_model << in_mode(self.model, train=False, grad=True)
320 + jacobians = g.atom()
321 + jacobians << averaged_jacobians(*self.stream)
322 + jlens = tuple(g.atom(f"jlens{index}") for index in range(len(self.stream)))
323 + for index, residual in enumerate(self.stream):
324 +     carried = pointwise(self.ln_f, residual @ jacobians[index].mT)
325 +     jlens[index] << lm_loss(pointwise(self.lm_head, carried), self.targets)

    gpt2_hyperpath.py: Inference.jacobian_lens
479 jacobians = g.derive({**self.given, gpt2.x: prompts}, outputs=gpt2.jacobians,
480                       assuming=[gpt2.differentiating])
481 return g.derive({**self.given, gpt2.jacobians: jacobians, gpt2.x: tokens, gpt2.targets: targets},
482                 outputs=gpt2.jlens, assuming=self.evaluating)
```

3. Activation patching. Patch one layer's residual stream as causal tracing does, restoring it at one position from a clean run into a corrupted one and reading the last position's logits.

PyTorch

PyTorch needs a function of its own, which catches the clean stream with one forward hook and splices it into the corrupted run with another, since `GPT.forward` has no way in between two blocks.

```
    new function: patched
100 + def patched(model, clean, corrupted, layer, position):
102 +     consumer = (model.transformer.h[layer] if layer < len(model.transformer.h)
103 +                 else model.transformer.ln_f)
104 +     kept = []
105 +     hook = consumer.register_forward_pre_hook(lambda _module, args: kept.append(args[0]))
106 +     try:
107 +         model(clean)
108 +     finally:
109 +         hook.remove()
110 +
111 +     def restore(_module, args):
112 +         stream = args[0].clone()
113 +         stream[:, position] = kept[0][:, position]
114 +         return (stream, *args[1:])
115 +
116 +     hook = consumer.register_forward_pre_hook(restore)
117 +     try:
118 +         logits, _ = model(corrupted)
119 +     finally:
120 +         hook.remove()
121 +     return logits[:, -1, :]

    questions.py: the call, in same_answers
116 logits = solutions.patched(model, tokens, corrupted, layer, position)
```

NNsight

PyTorch with NNsight needs a function of its own, which reads the clean stream inside one trace and writes it into the corrupted run inside another.

```
    new function: patched
 66 + def patched(gpt, clean, corrupted, layer, position):
 73 +     consumer = gpt.transformer.h[layer] if layer < len(gpt.transformer.h) else gpt.transformer.ln_f
 74 +     with gpt.trace(clean):
 75 +         kept = consumer.input[:, position].save()
 76 +     with gpt.trace(corrupted):
 77 +         consumer.input[:, position] = kept
 78 +         logits = gpt.output[0][:, -1, :].save()
 79 +     return logits

    nnsight_questions.py: the call, in same_answers
 67 logits = nnsight_solutions.patched(gpt, tokens, corrupted, layer, position)
```

Graph

The graph answers it with its 37 edges and no new one, and it is asked through the three derivations below.

```
    gpt2_hyperpath.py: Inference.patched
495 clean_stream = g.derive({**self.given, gpt2.x: clean}, outputs=gpt2.stream[layer],
496                         assuming=self.evaluating)
497 restored = g.derive({**self.given, gpt2.x: corrupted}, outputs=gpt2.stream[layer],
498                     assuming=self.evaluating).clone()
499 restored[:, position] = clean_stream[:, position]
500 return g.derive({**self.given, gpt2.stream[layer]: restored}, outputs=gpt2.logits[-1],
501                 horizon=corrupted.size(1), assuming=self.evaluating)
```

4. In-context score. Score in-context learning as each sequence's loss on a late token minus its loss on an early one.

PyTorch

PyTorch needs a function of its own, which recomputes each position's loss from the logits, since `GPT.forward` averages the loss over every position.

```
    new function: in_context
124 + def in_context(model, tokens, targets, late, early):
126 +     logits, _ = model(tokens, targets)
127 +     losses = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1), ignore_index=-1,
128 +                              reduction="none").view(targets.shape)
129 +     return losses[:, late] - losses[:, early]

    questions.py: the call, in same_answers
118 scores = solutions.in_context(model, tokens, targets, late, early)
```

Graph

The graph answers it with its 37 edges and no new one, and it is asked through the call below.

```
    gpt2_hyperpath.py: Inference.in_context
507 return g.derive({**self.given, gpt2.x: tokens, gpt2.targets: targets},
508                 outputs=gpt2.nll[late] - gpt2.nll[early], assuming=self.evaluating)
```

Declaration

gpt2_hyperpath.py: GPT2.declare, the model every question is asked of

```
g = hp.Graph(backend=Backend("torch", device=str(device), axis=1), transfer="store")
model = g.atom(role="input")
step_index = g.atom(role="input")
active_model, optimizer = g.atom()
wte, wpe, ln_f, lm_head = g.atom()
layers = tuple(g.atom(f"layer{index}") for index in range(n_layer))
x = g.series(recurrence="transition")
targets = g.series(role="input")
stream = [g.series(f"r{index}") for index in range(n_layer + 1)]
attended = [g.series(f"a{index}") for index in range(n_layer)]
logits, nll = g.series()
loss, share, norm, rate = g.atom()
fit = g.atom(role="effect")
init, training = g.condition()

(wte, wpe, ln_f, lm_head, *layers) << parts(active_model)
stream[0][t] << embed(wte, wpe, x[t], t)
for index, layer in enumerate(layers):
    attended[index][t] << stream[index][t] + attend(layer, stream[index][: t + 1])
    stream[index + 1][t] << (attended[index][t]
                             + pointwise(layer.mlp, pointwise(layer.ln_2, attended[index][t])))
logits[t] << pointwise(lm_head, pointwise(ln_f, stream[-1][t]))
loss << lm_loss(logits, targets)
nll[t] << token_nll(logits[t], targets[t])
x[t + 1] << greedy(logits[t])

with g.when(init):
    optimizer << configured(model, config.weight_decay, config.learning_rate, config.zero_stage,
                            str(device))

with g.when(training):
    active_model << in_mode(model, train=True, grad=True)
    share << accumulate(loss, config.grad_accum_steps)
    norm  << clipped(active_model, config.grad_clip)
    rate  << warmup_cosine(step_index, config.learning_rate, config.warmup_iters,
                           config.num_iterations, config.learning_rate_decay_frac)
    fit   << optimize(optimizer, rate, norm)

with g.when(~training):
    active_model << in_mode(model, train=False, grad=False)
```

D.3 PPO

We compare against CleanRL (Huang et al., 2022), whose `ppo_continuous_action.py` is its `ppo.py` copied with a Gaussian policy in place of the categorical one, normalising wrappers and its own defaults. The graph keeps both policies and a run picks one, so we edited `ppo.py` to keep both as well, choosing one from the environment's action space, and print the edit as a diff of `ppo.py`. We leave out the environment wrappers the edit adds, since the graph builds its own outside its edges. Before it records the row, the experiment runs each of the graph's two derivations beside a copy of its CleanRL file's loop on the CPU for five iterations of rollout and update, and every tensor agrees to the last bit. The copy takes the graph's random draws, which the file takes from global generators. It also holds its learning rate as a tensor and takes the library's entropies, as the graph does. For the Gaussian policy it takes the library's log-probabilities too, and runs the critic once over the whole rollout. Each of these four choices moves only the last bits of a result. It then runs the edited file for three iterations beside each of CleanRL's, and every value the two scripts both log agrees, except the steps per second.

	Tokens		
Question	**PyTorch**	**Graph**	**The graph's edges**
1. Continuous actions	279	167	2 added, 2 guarded; same calls

1. Continuous actions. Train a Gaussian policy over continuous actions, CleanRL's `ppo_continuous_action.py`, beside the categorical policy `ppo.py` trains.

PyTorch

CleanRL answers it with a second file, `ppo.py` copied with its categorical policy replaced. To keep that policy beside the new one, as the graph does, PyTorch changes `ppo.py` in 7 places, the policy following the environment's action space: a mean network and one log standard deviation per action beside upstream's actor, a normal over them beside its categorical, their log-probability and entropy summed over the action axis, and the import, the assert and the update's call that name the kind of action. We leave out `make_env`'s wrappers for continuous environments, which the graph keeps outside its edges too.

```
  ppo.py, top-level code after line 13: Normal imported beside Categorical
+ from torch.distributions.normal import Normal
  ppo.py, Agent.__init__ after line 102: whether the action space is continuous, read off the environments
+         self.continuous = isinstance(envs.single_action_space, gym.spaces.Box)
  ppo.py, Agent.__init__: a mean network as wide as the action space and one log standard deviation per
  ↪ action, beside upstream's actor
+         if self.continuous:
+             self.actor_mean = nn.Sequential(
+                 layer_init(nn.Linear(np.array(envs.single_observation_space.shape).prod(), 64)),
+                 nn.Tanh(),
+                 layer_init(nn.Linear(64, 64)),
+                 nn.Tanh(),
+                 layer_init(nn.Linear(64, np.prod(envs.single_action_space.shape)), std=0.01),
+             )
+             self.actor_logstd = nn.Parameter(torch.zeros(1, np.prod(envs.single_action_space.shape)))
+         else:
110 =            self.actor = nn.Sequential(
111 =                layer_init(nn.Linear(np.array(envs.single_observation_space.shape).prod(), 64)),
112 =                nn.Tanh(),
113 =                layer_init(nn.Linear(64, 64)),
114 =                nn.Tanh(),
115 =                layer_init(nn.Linear(64, envs.single_action_space.n), std=0.01),
116 =            )
  ppo.py, Agent.get_action_and_value: a normal from the mean and the exponentiated deviations, beside
  ↪ upstream's categorical
+         if self.continuous:
+             action_mean = self.actor_mean(x)
+             action_logstd = self.actor_logstd.expand_as(action_mean)
+             action_std = torch.exp(action_logstd)
+             probs = Normal(action_mean, action_std)
+         else:
122 =            logits = self.actor(x)
123 =            probs = Categorical(logits=logits)
  ppo.py, Agent.get_action_and_value after line 125: its log-probability and entropy summed over the
  ↪ action axis
+         if self.continuous:
+             return action, probs.log_prob(action).sum(1), probs.entropy().sum(1), self.critic(x)
  ppo.py, top-level code: either kind of action space asserted
165 -    assert isinstance(envs.single_action_space, gym.spaces.Discrete), "only discrete action space is
  ↪ supported"
+    assert isinstance(envs.single_action_space, (gym.spaces.Discrete, gym.spaces.Box)), "only discrete
  ↪ and continuous action spaces are supported"
  ppo.py, top-level code: the update's actions without the .long() a categorical takes
250 -                _, newlogprob, entropy, newvalue = agent.get_action_and_value(b_obs[mb_inds],
  ↪ b_actions.long()[mb_inds])
+                mb_actions = b_actions[mb_inds] if agent.continuous else b_actions.long()[mb_inds]
+                _, newlogprob, entropy, newvalue = agent.get_action_and_value(b_obs[mb_inds],
  ↪ mb_actions)
```

Graph

The graph adds two edges to its 41 and moves two of those under a condition, in the code below, and it trains through the same calls as before, pinning `action_space` to `"continuous"`.

```
    ppo_hyperpath.py: GaussianHead, the mean network and log deviation
136 + class GaussianHead(nn.Module):
146 +
147 +     def __init__(self, mean, actions):
149 +         super().__init__()
150 +         self.mean = mean
151 +         self.log_deviation = nn.Parameter(torch.zeros(1, actions))
    ppo_hyperpath.py: gaussian_networks, networks' actor and critic, the actor wrapped in the head
172 + def gaussian_networks(config, observations, actions):
177 +     actor, critic = networks(config, observations, actions)
178 +     return GaussianHead(actor, actions), critic
    ppo_hyperpath.py: PPO.with_continuous_actions
369 + action_space = g.condition()
370 + with g.when(action_space == "discrete"):
265 =     policy[t] << dist.categorical(logits=array.stop_gradient(actor(observation[t])))
323 =     scored << dist.categorical(logits=actor(minibatch_observation))
373 + with g.when(action_space == "continuous"):
374 +     policy[t] << dist.normal(array.stop_gradient(actor.mean(observation[t])),
375 +                              log_scale=array.stop_gradient(actor.log_deviation), event_dims=1)
376 +     scored << dist.normal(actor.mean(minibatch_observation), log_scale=actor.log_deviation,
    ↪  event_dims=1)
```

Declaration

```
    ppo_hyperpath.py: PPO.declare, ppo.py's model, the variant's base
242 g = hp.Graph(backend="torch")
243
244 envs, iteration_index, generator = g.atom(role="input")
245 start_observation, start_done = g.atom(role="input")
246 actor, critic, optimizer = g.atom(role="input")
247 minibatch_picks = g.atom(role="input")
248
249 # --- the rollout: T + 1 observations, T transitions
250 observation, reward, done = g.series(recurrence="transition")
251 policy = g.series(backend=Backend("torch", record=True))
252 action, logp = g.series()
253
258 observation[:t0] << start_observation
259 reward[:t0] << array.zeros_like(start_done)
260 done[:t0] << start_done
261 (observation[t + 1], reward[t + 1], done[t + 1]) << env_step(envs, action[t])
262
265 policy[t] << dist.categorical(logits=array.stop_gradient(actor(observation[t])))
266 action[t] << dist.sample(policy[t], generator)
267 logp[t] << dist.log_prob(policy[t], action[t])
268
269 # --- what each action was worth
270 value, returns = g.series()
271 delta, advantage = g.series(recurrence="cumulative", direction="backward")
272
274 value[t] << array.stop_gradient(state_values(critic, observation[t]))
275
279 delta[tT:] << array.zeros_like(value[-1:])
280 advantage[tT:] << array.zeros_like(value[-1:])
281 delta[t] << reward[t + 1] + config.gamma * value[t + 1] * (1 - done[t + 1]) - value[t]
282 advantage[t] << delta[t] + config.gamma * config.gae_lambda * (1 - done[t + 1]) * advantage[t + 1]
283 returns[t] << advantage[t] + value[t]
284
```

```
# --- the flatten, and one minibatch of it
shuffled_order, minibatch_indices = g.atom()
shuffled_order << shuffle(config.batch_size, generator)
minibatch_indices << shuffled_order[minibatch_picks]

batch_observation, batch_action, batch_logp = g.atom()
batch_advantage, batch_returns, batch_value = g.atom()
batch_observation << array.reshape(observation[:-1], (-1, observation_width))
batch_action << array.reshape(action[:-1], (-1,) if action_width is None else (-1, action_width))
batch_logp << array.reshape(logp[:-1], (-1,))
batch_advantage << array.reshape(advantage[:-1], (-1,))
batch_returns << array.reshape(returns[:-1], (-1,))
batch_value << array.reshape(value[:-1], (-1,))

minibatch_observation, minibatch_action, minibatch_logp = g.atom()
minibatch_advantage, minibatch_returns, minibatch_value = g.atom()
minibatch_observation << batch_observation[minibatch_indices]
minibatch_action << batch_action[minibatch_indices]
minibatch_logp << batch_logp[minibatch_indices]
minibatch_advantage << batch_advantage[minibatch_indices]
minibatch_returns << batch_returns[minibatch_indices]
minibatch_value << batch_value[minibatch_indices]

# --- the update, over one minibatch
normalized, scored, ratio, clipped = g.atom()
new_value, clipped_value = g.atom()
policy_loss, value_loss, entropy = g.atom()
loss = g.atom(combine="sum")
learning_rate = g.atom()

normalized << ((minibatch_advantage - array.mean(minibatch_advantage))
               / (array.std(minibatch_advantage, correction=1) + 1e-8))

scored << dist.categorical(logits=actor(minibatch_observation))
ratio << array.exp(dist.log_prob(scored, minibatch_action) - minibatch_logp)
clipped << array.clip(ratio, 1 - config.clip_coef, 1 + config.clip_coef)
policy_loss << -array.mean(array.minimum(ratio * normalized, clipped * normalized))

new_value << state_values(critic, minibatch_observation)
clipped_value << minibatch_value + array.clip(new_value - minibatch_value, -config.clip_coef,
                                              config.clip_coef)
value_loss << 0.5 * array.mean(array.maximum((new_value - minibatch_returns) ** 2,
                                             (clipped_value - minibatch_returns) ** 2))

entropy << array.mean(dist.entropy(scored))
loss += policy_loss
loss -= config.ent_coef * entropy
loss += config.vf_coef * value_loss
learning_rate << annealed(config.learning_rate, iteration_index, config.num_iterations)

fit = g.atom(role="effect")
fit << optimize(optimizer, loss, learning_rate, config.max_grad_norm)
```

D.4 NANOGPT-GRU

We compare against nanoGPT's `model.py` (Karpathy, 2022), a GPT-style transformer, and swap its attention for a GRU (Cho et al., 2014). Figure 1 is this model cut to one layer, with attention's keys and values written as series of their own, a tanh step in place of the GRU, and no residual stream. nanoGPT gains a module that calls PyTorch's GRU and an option that picks it. The graph adds GRU weights to each block, writes the recurrence as a rule that calls the GRU over the whole sequence, which a step runs over the one position it reads, and moves attention under a condition. It binds a prompt of any length as the bootstrap `tokens[:t0]`. Before it records the rows, the experiment trains both models from the same weights and generates from prompts of one and three tokens under each mixer. The graph's weights after training equal nanoGPT's to the last bit, and so do its logits against nanoGPT's modules run one position at a time, after one call over the prompt under attention. Under the GRU its tokens also equal those of the loop we print.

We also time that generation on one A100 GPU, eagerly and in float32. At 1, 64 and 1,024 prompts at once, five processes of one seed each continue one-token prompts to 32, 64, 128 and 256 tokens, the length of the model's context. In all twelve cells the graph takes 1.06 to 1.08 times as long as the loop we print. nanoGPT's `GPT.generate`, unchanged, runs the whole prefix again for every token, and takes 1.5 to 4.4 times the loop's time continuing one prompt, 1.8 to 9.4 continuing 64 and 9.4 to 71 continuing 1,024. The graph's excess stays at 0.15 to 0.20 ms a step at every batch size, so the graph pays it once a step on the host. There it places the values each step stores and appends them to their series, and it hands the GRU's rule each step as a sequence one position long. At this model's size the loop's own step takes 2.3 to 2.4 ms whether it continues one prompt or 1,024, so the host issuing its operations paces the step, and the graph's host work adds to it in full. We time each first call apart, and the graph's takes 180 to 245 ms longer than its later ones, planning included.

	Tokens		
Question	**PyTorch**	**Graph**	**The graph's edges**
1. The mixer	176	273	12 added, 6 guarded; same call
2. Generating with the new mixer	240	45	No new edge
Total	416	318	

1. The mixer. Let every block mix by a GRU in place of its causal self-attention, leaving the rest of the model as it is.

PyTorch

PyTorch changes `model.py` in 3 places: a new module, `GruMixer`, with `CausalSelfAttention`'s interface, a branch in `Block` building the mixer the configuration names, and a `mixer` option in `GPTConfig`.

```
  model.py, GruMixer after line 76: new, a recurrent mixer with CausalSelfAttention's interface
+ class GruMixer(nn.Module):
+
+     def __init__(self, config):
+         super().__init__()
+         self.gru = nn.GRU(config.n_embd, config.n_embd, batch_first=True)
+         self.c_proj = nn.Linear(config.n_embd, config.n_embd, bias=config.bias)
+         self.resid_dropout = nn.Dropout(config.dropout)
+
+     def forward(self, x):
+         return self.project(self.recur(x))
+
+     def recur(self, x, state=None):
+         hidden, _final = self.gru(x, None if state is None else state.unsqueeze(0))
+         return hidden
+
+     def project(self, hidden):
+         return self.resid_dropout(self.c_proj(hidden))
  model.py, Block.__init__: the mixer the configuration names, attention otherwise
99 -         self.attn = CausalSelfAttention(config)
   +         self.attn = GruMixer(config) if config.mixer == "gru" else CausalSelfAttention(config)
  model.py, GPTConfig after line 116: an option naming the mixer
+     mixer = "attention"
```

Graph

The graph adds twelve edges to its 21 and moves six of those under a condition, in the code below, and it trains through the same call as before, pinning `mixer` to `"gru"`.

```
    language_model_hyperpath.py: recur
115 + @hp.rule.vectorized("torch", axes=1, carries=("hidden",))
116 + def recur(block, hidden, current):
128 +     return block.attn.gru(block.ln_1(current), hidden[:, -1].unsqueeze(0))[0]
    language_model_hyperpath.py: LanguageModel.with_gru
312 + mixer = g.condition()
313 + hidden = tuple(g.series(f"hidden{index}", recurrence="cumulative") for index in range(config.n_layer))
265 = for index in range(config.n_layer):
266 =     layer = layers[index]
316 +     with g.when(mixer == "attention"):
268 =         mixed[index][t] << stream[index][t] + attend(layer, stream[index][: t + 1])
318 +     with g.when(mixer == "gru"):
319 +         hidden[index][t] << recur(layer, hidden[index][t - 1].or_else(0.0), stream[index][t]
320 +                                   ).returns(like=stream[index][t])
321 +         mixed[index][t] << stream[index][t] + pointwise(layer.attn.proj, hidden[index][t])
    language_model_hyperpath.py: with_gru_weights
325 + def with_gru_weights(model, config):
332 +     for block in model.transformer.h:
333 +         block.attn = nn.ModuleDict({"gru": nn.GRU(config.n_embd, config.n_embd, batch_first=True),
334 +                                     "proj": nn.Sequential(nn.Linear(config.n_embd, config.n_embd),
335 +                                                           nn.Dropout(config.dropout))})
336 +     return model
```

2. Generating with the new mixer. Continue a prompt token by token with the GRU model, carrying its state from one token to the next.

PyTorch

PyTorch needs a loop of its own that steps each block's modules itself to carry its state, since `GPT.generate` re-runs the model over the whole prefix for every token.

```
    new function: generate
 38 + @torch.no_grad()
 39 + def generate(model, idx, max_new_tokens, temperature=1.0):
 41 +     layers = model.transformer
 42 +     states = [None] * len(layers.h)
 43 +     for position in range(idx.size(1) + max_new_tokens - 1):
 44 +         x = layers.drop(layers.wte(idx[:, position]) + layers.wpe.weight[position])
 45 +         for index, block in enumerate(layers.h):
 46 +             gru = block.attn
 47 +             states[index] = gru.recur(block.ln_1(x).unsqueeze(1), states[index])[:, 0]
 48 +             x = x + gru.project(states[index])
 49 +             x = x + block.mlp(block.ln_2(x))
 50 +         if position >= idx.size(1) - 1:
 51 +             logits = model.lm_head(layers.ln_f(x))
 52 +             probs = F.softmax(logits / temperature, dim=-1)
 53 +             idx = torch.cat((idx, torch.multinomial(probs, num_samples=1)), dim=1)
 54 +     return idx

    selfcheck.py: the call, in swap_generating_needs_no_new_entry_point
1024 solved = solutions.generate(model, prompt, NEW_TOKENS, cell_config.temperature)
```

Graph

The graph answers it with its 33 edges and no new one, and it is asked through the call below, pinning `mixer` to `"gru"`.

```
    language_model_hyperpath.py: Inference.generate
469 return g.derive({**self.given, graph.tokens[:t0]: prompt}, outputs=graph.tokens,
470                 horizon=prompt.size(1) + new_tokens, assuming=self.evaluating)
```

Declaration

language_model_hyperpath.py: LanguageModel.declare, nanoGPT's model with attention in every block

```
g = hp.Graph(backend=Backend("torch", device=device, axis=1), transfer="store")

model = g.atom(role="input")
learning_rate = g.atom(role="input")
active, optimizer, loss = g.atom()
fit = g.atom(role="effect")
targets = g.series(role="input")
tokens = g.series(recurrence="transition")
logits = g.series()
mode, init = g.condition()

wte, wpe, drop, ln_f, lm_head, layers = g.atom()
(wte, wpe, drop, ln_f, lm_head, layers) << parts(active)

stream = tuple(g.series(f"stream{index}") for index in range(config.n_layer + 1))
mixed = tuple(g.series(f"mixed{index}") for index in range(config.n_layer))

stream[0][t] << embed(wte, wpe, drop, tokens[t], t)
for index in range(config.n_layer):
    layer = layers[index]
    mixed[index][t] << stream[index][t] + attend(layer, stream[index][: t + 1])
    stream[index + 1][t] << (mixed[index][t]
                             + pointwise(layer.mlp, pointwise(layer.ln_2, mixed[index][t])))

logits[t] << pointwise(lm_head, pointwise(ln_f, stream[-1][t]))
loss << cross_entropy(logits, targets)

tokens[t + 1] << pick_next(logits[t], config.temperature)

with g.when(mode == "train"):
    active << in_mode(model, train=True)
    fit << optimize(optimizer, active, loss, learning_rate, config.grad_clip)
with g.when(mode == "eval"):
    active << in_mode(model, train=False)

with g.when(init):
    optimizer << model.configure_optimizers(config.weight_decay, config.learning_rate,
                                            (config.beta1, config.beta2), device.type)
```

D.5 VAE

We compare against `vae/main.py` from `pytorch/examples` (PyTorch contributors, 2017), which trains a variational autoencoder (Kingma & Welling, 2014) on MNIST. We train both only on synthetic batches of MNIST's shape.

	Tokens		
Question	**PyTorch**	**Graph**	**The graph's edges**
1. Mean reconstruction	58	55	1 added
2. Test set terms	108	48	No new edge
3. Generic estimator	154	70	1 added, 1 guarded; same call
Total	320	173	

1. Mean reconstruction. Reconstruct each image from its posterior mean instead of a draw, so that the answer is deterministic.

PyTorch

PyTorch needs a function of its own, which repeats the encode and decode of `VAE.forward` without its draw.

```
    new function: mean_reconstruction
27 + def mean_reconstruction(model, data):
29 +     model.eval()
30 +     with torch.no_grad():
31 +         mu, _logvar = model.encode(data.view(-1, 784))
32 +         return model.decode(mu)

    selfcheck.py: the call, in vae_mean_reconstruction_is_the_decoder_over_the_mean
273 expected = solutions.mean_reconstruction(model, _images(loader))
```

Graph

The graph adds one edge to its 14, in the code below, and it is asked through the call after it.

```
    vae_hyperpath.py: Vae.with_mean_reconstruction
182 + mean_recon = g.series()
183 + mean_recon[t] << active.decode(mean[t])

    vae_hyperpath.py: Inference.mean_reconstruction
298 return g.derive({**self.given, vae.batches: loader}, outputs=vae.mean_recon,
299                 assuming=self.test, lifting="never")
```

2. Test set terms. Report the test set's reconstruction term and KL divergence beside its loss.

PyTorch

PyTorch changes `main.py` in five places: `loss_function` returns the two terms beside their sum, so both of its callers take the new return value apart, the training loop included, and `test` sums, averages and prints each term.

```
    main.py, loss_function: the two terms, handed back beside their sum
 89 - return BCE + KLD
    + return BCE + KLD, BCE, KLD
    main.py, train: the new return value taken apart, though training reads only the sum
 99 -     loss = loss_function(recon_batch, data, mu, logvar)
    +     loss, _, _ = loss_function(recon_batch, data, mu, logvar)
    main.py, test: two more running sums
115 - test_loss = 0
    + test_loss = test_bce = test_kld = 0
    main.py, test: the return value taken apart, and each term summed
120 -         test_loss += loss_function(recon_batch, data, mu, logvar).item()
    +         loss, bce, kld = loss_function(recon_batch, data, mu, logvar)
    +         test_loss += loss.item()
    +         test_bce += bce.item()
    +         test_kld += kld.item()
    main.py, test: each term averaged and printed beside the loss
129 - print('====> Test set loss: {:.4f}'.format(test_loss))
    + test_bce /= len(test_loader.dataset)
    + test_kld /= len(test_loader.dataset)
    + print(f"====> Test set loss: {test_loss:.4f} (reconstruction {test_bce:.4f}, KL {test_kld:.4f})")
```

Graph

The graph answers it with its 14 edges and no new one, and it is asked through the call below.

```
    vae_hyperpath.py: Inference.evaluate_with_terms
305 return g.derive({**self.given, vae.batches: loader}, outputs=(vae.loss, vae.bce, vae.kld),
306                 name="test", assuming=self.test, lifting="never")
```

3. Generic estimator. Train with the generic estimator of Kingma & Welling (2014), which scores the drawn latent under the posterior and the prior, beside the closed-form KL.

PyTorch

PyTorch changes `main.py` in two places, a new option in the parser and `train` branching on it. At the generic estimator it repeats `VAE.forward` to reach the latent it scores, and otherwise it calls `VAE.forward` and `loss_function` as before.

```
    main.py, the parser after line 21: the option naming the estimator
  + parser.add_argument("--estimator", default="closed_form", help="closed_form | generic")
    main.py, train: a branch drawing the latent and scoring it under the posterior and the prior
  +         if args.estimator == "generic":
  +             flat = data.view(-1, 784)
  +             mu, logvar = model.encode(flat)
  +             std = torch.exp(0.5 * logvar)
  +             z = mu + torch.randn_like(std) * std
  +             bce = F.binary_cross_entropy(model.decode(z), flat, reduction="sum")
  +             log_q = -0.5 * ((z - mu) / std) ** 2 - torch.log(std) - math.log(2 * math.pi) / 2
  +             log_p = -0.5 * z ** 2 - math.log(2 * math.pi) / 2
  +             loss = bce + torch.sum(log_q - log_p)
  +         else:
98 =             recon_batch, mu, logvar = model(data)
99 =             loss = loss_function(recon_batch, data, mu, logvar)
```

Graph

The graph adds one edge to its 14 and moves one of those under a condition, in the code below, and it trains through the same call as before, pinning `estimator` to `"generic"`.

```
    vae_hyperpath.py: Vae.with_generic_estimator
200 + estimator = g.condition()
201 + with g.when(estimator == "closed_form"):
161 =     kld[t] << -0.5 * array.sum(1 + logvar[t] - mean[t] ** 2 - array.exp(logvar[t]))
203 + with g.when(estimator == "generic"):
204 +     kld[t] << array.sum(dist.log_prob(posterior[t], z[t])
205 +                         - dist.log_prob(dist.normal(0.0, 1.0), z[t]))
```

Declaration

vae_hyperpath.py: Vae.declare, the model every question is asked of

```
g = hp.Graph(backend=Backend("torch", device=device), transfer="store")

model, rng = g.atom(role="input")
optimizer, active = g.atom()
batches = g.series(role="input", supply="stream", backend="python")
images, flat, mean, logvar = g.series()
z, recon, bce, kld = g.series()
loss = g.series(combine="sum")
posterior = g.series(backend=Backend("torch", device=device, record=True))
fit = g.series(role="effect")
mode, init = g.condition()

images[t]    << batches[t][0]
flat[t]      << images[t].view(-1, config.input_dim)
hp.tuple(mean[t], logvar[t]).via(active.encode(flat[t]), after=fit[t - 1])
posterior[t] << dist.normal(mean[t], array.exp(0.5 * logvar[t]))
z[t]         << dist.rsample(posterior[t], generator=rng)
recon[t]     << active.decode(z[t])
bce[t]       << hp.call(F.binary_cross_entropy, recon[t], flat[t], reduction="sum")
kld[t]       << -0.5 * array.sum(1 + logvar[t] - mean[t] ** 2 - array.exp(logvar[t]))
loss[t]      += bce[t]
loss[t]      += kld[t]

with g.when(mode == "train"):
    active << in_mode(model, train=True)
    fit[t] << optimize(optimizer, loss[t])

with g.when(mode == "test"):
    active << in_mode(model, train=False)

with g.when(init):
    optimizer << hp.call(torch.optim.Adam, model.parameters(), lr=config.learning_rate)
```

D.6 GAN

We compare against `dcgan/main.py` from `pytorch/examples` (PyTorch contributors, 2017), a DCGAN (Radford et al., 2016) whose two networks train with alternating optimizers. Each question trains the model in a way the file reaches only by editing its training loop. We edit the loop as its author would to keep every loss, with a branch on a new option. In the graph each generator loss moves the reference's under a condition and sits beside it, and the penalty joins the discriminator's loss as one more term, `loss_d[t] += ...`, each under a condition a run pins. Both then keep every loss. PyTorch puts a new one into the loop, beside the reference's lines, and the graph into edges of its own, leaving its training call unchanged. The penalty scores the real batch a second time, and we order that pass after the one over the generated batch (`after=score_fake[t]`). The order the passes run in decides the order in which autograd adds up each weight's gradient, and the edited file runs them in that order. The penalty's weight is a hyperparameter neither side counts: the file reads it from its options and the graph from its configuration.

	Tokens		
Question	**PyTorch**	**Graph**	**The graph's edges**
1. Minimax loss	49	42	1 added, 1 guarded; same call
2. Feature matching	61	79	1 added, 1 guarded; same call
3. Gradient penalty	88	96	2 added; same call
Total	198	217	

1. Minimax loss. Train the generator by the minimax objective of Goodfellow et al. (2014), beside the non-saturating loss the reference trains it by.

PyTorch

PyTorch changes `main.py` in three places, a new option in the parser and two in the training loop branching on it, which score the generated batch against the fake label and negate its error.

```
   main.py, the parser after line 35: the option naming the generator's loss
+  parser.add_argument("--generator-loss", default="non_saturating", help="non_saturating | minimax")
   main.py, the training loop: the generated batch scored against the fake label
-          label.fill_(real_label)
+          minimax = opt.generator_loss == "minimax"
+          label.fill_(fake_label if minimax else real_label)
   main.py, the training loop: the error negated
-          errG = criterion(output, label)
+          errG = -criterion(output, label) if minimax else criterion(output, label)
```

Graph

The graph adds one edge to its 16 and moves one of those under a condition, in the code below, and it trains through the same call as before, pinning `generator_loss` to `"minimax"`.

```
   dcgan_hyperpath.py: Dcgan.generator_losses
+ generator_loss = g.condition()
+ with g.when(generator_loss == "non_saturating"):
=      loss_g[t] << error(score_for_g[t], REAL_LABEL)
   dcgan_hyperpath.py: Dcgan.with_minimax_loss
+ with g.when(generator_loss == "minimax"):
+     loss_g[t] << -error(score_for_g[t], FAKE_LABEL)
```

2. Feature matching. Train the generator by the feature matching of Salimans et al. (2016), which matches the discriminator's mean features at an intermediate layer, the last hidden one in their released code, over a real and a generated batch.

PyTorch

PyTorch changes `main.py` in two places, a new option in the parser and a branch on it around the training loop's error, which cuts `Discriminator.main` before its last layer to reach the features `Discriminator.forward` never hands back.

```
    main.py, the parser after line 35: the option naming the generator's loss
+ parser.add_argument("--generator-loss", default="non_saturating", help="non_saturating |
  ↪  feature_matching")
    main.py, the training loop: a branch taking the distance between the two batches' mean features
+         if opt.generator_loss == "feature_matching":
+             features = netD.main[:-2]
+             errG = nn.functional.mse_loss(features(fake).mean(0), features(real_cpu).mean(0))
+         else:
256 =             errG = criterion(output, label)
```

Graph

The graph adds one edge to its 16 and moves one of those under a condition, in the code below, and it trains through the same call as before, pinning `generator_loss` to `"feature_matching"`.

```
    dcgan_hyperpath.py: Dcgan.generator_losses
245 + generator_loss = g.condition()
246 + with g.when(generator_loss == "non_saturating"):
219 =     loss_g[t] << error(score_for_g[t], REAL_LABEL)
    dcgan_hyperpath.py: Dcgan.with_feature_matching
274 + features = updated_discriminator[t].main[:-2]
275 + with g.when(generator_loss == "feature_matching"):
276 +     loss_g[t] << hp.call(F.mse_loss, features(fake[t]).mean(0), features(real[t]).mean(0))
```

3. Gradient penalty. Train the discriminator with the R1 gradient penalty of Mescheder et al. (2018), the squared norm of its gradient with respect to each real image.

PyTorch

PyTorch changes `main.py` in two places, a new option in the parser and a branch on it in the training loop, which scores the real batch again as a tensor a gradient can be taken with respect to and adds the penalty to the generated batch's error.

```
    main.py, the parser after line 35: the option naming the penalty
+ parser.add_argument("--penalty", default="none", help="none | gradient")
    main.py, the training loop after line 244: a branch adding the penalty to the generated batch's error
+         if opt.penalty == "gradient":
+             real_input = real_cpu.detach().requires_grad_()
+             (gradient,) = torch.autograd.grad(netD(real_input).sum(), real_input, create_graph=True)
+             errD_fake = errD_fake + opt.penalty_weight / 2 * (gradient ** 2).sum((1, 2, 3)).mean()
```

Graph

The graph adds two edges to its 16, in the code below, and it trains through the same call as before, pinning `penalty` to `"gradient"`.

```
    dcgan_hyperpath.py: Dcgan.with_gradient_penalty
294 + penalty = g.condition()
295 + gradient = g.series()
296 + with g.when(penalty == "gradient"):
297 +     gradient[t].via(array.grad(array.sum(discriminator(real[t])), real[t]), after=score_fake[t])
298 +     loss_d[t] += config.penalty_weight / 2 * array.mean(array.sum(gradient[t] ** 2, (1, 2, 3)))
```

Declaration

```
    dcgan_hyperpath.py: Dcgan.declare, the model every question is asked of
g = hp.Graph(backend=Backend("torch", device=device), transfer="store")

net_g, net_d = g.atom(role="input")
optimizer_g, optimizer_d = g.atom()
generator, discriminator = g.atom()
batches = g.series(role="input", supply="stream", backend="python")
real, noise, fake, fake_detached = g.series()
score_real, score_fake, score_for_g = g.series()
loss_d = g.series(combine="sum")
loss_g = g.series()
updated_discriminator = g.series(backend="python")
fit = g.series(role="effect")
init = g.condition()

generator << in_mode(net_g, train=True)
discriminator.via(in_mode(net_d, train=True), after=generator)

real[t]  << batches[t][0]
noise[t] << latent(real[t], config.latent_dim)
fake[t].via(generator(noise[t]), after=fit[t - 1])

fake_detached[t] << array.stop_gradient(fake[t])
score_fake[t].via(discriminator(fake_detached[t]), after=updated_discriminator[t - 1])
score_real[t].via(discriminator(real[t]), after=updated_discriminator[t - 1])
loss_d[t] += error(score_real[t], REAL_LABEL)
loss_d[t] += error(score_fake[t], FAKE_LABEL)

updated_discriminator[t] << optimize(optimizer_d, discriminator, loss_d[t])

score_for_g[t] << updated_discriminator[t](fake[t])
loss_g[t] << error(score_for_g[t], REAL_LABEL)
fit[t] << optimize(optimizer_g, generator, loss_g[t])

with g.when(init):
    optimizer_g << hp.call(torch.optim.Adam, net_g.parameters(),
                           lr=config.learning_rate, betas=(config.beta1, 0.999))
    optimizer_d << hp.call(torch.optim.Adam, net_d.parameters(),
                           lr=config.learning_rate, betas=(config.beta1, 0.999))
```

D.7 DQN

Section 5 trains this graph beside CleanRL's `dqn.py` (Huang et al., 2022), and we print the part of `DQN.declare` that declares it. A run derives it once over every environment step, and each setting `dqn.py` reads from `args` the graph reads from `config` under the same name. The fit and the target network's sync are effects, valueless edges a run asks to happen. The graph writes each only at the steps `dqn.py` runs it, as a slice of the steps: `fit[first_fit::config.train_frequency]` and `sync[first_sync::config.target_network_frequency]`. The lines above the listing compute `first_fit` and `first_sync` as the first steps past `config.learning_starts` at which `dqn.py` fits and syncs. Both edges write weights in place, and no value carries those writes to the edges that read the weights, so each reader declares what it follows. The act follows the previous step's fit (`after=fit[t - 1]`), the forward pass through the target network follows the previous step's sync (`after=sync[t - 1]`), and the sync follows its own step's fit (`after=fit`).

We hand the graph the replay buffer as an input, as we hand it both networks, and the buffer holds its rows itself. A `remember` edge writes each step's transition into it and hands it back as `stored[t]`, and the five gathers of a batch read `stored[t]` at the rows in `drawn[t]`. Those come from `dist.integers(0, config.buffer_size)`, the uniform over the buffer's rows, drawn `config.batch_size` at a time as `dqn.py`'s `np.random.randint` draws them once its buffer is full, which it is by the first fit at CleanRL's settings, and the graph refuses settings under which it is not. So the buffer decides which transitions it keeps and for how long, as a ring of `config.buffer_size` rows like `dqn.py`'s, and the plan keeps none of them. Reading `stored[t]` also puts each gather after the step's store, as `dqn.py` samples after its `rb.add`. Section 5's check runs this graph beside a copy of `dqn.py`'s loop, and at the end both networks, every row of the buffer and the state of each random stream equal the copy's to the last bit.

```
    dqn_hyperpath.py: DQN.declare, dqn.py's loop
g = hp.Graph(backend="torch")

envs, coins, picks, start_observation = g.atom(role="input")
q_network, target_network, optimizer, replay = g.atom(role="input")
fit, sync = g.series(role="effect")

# --- acting and storing
observation = g.series(recurrence="transition", backend="numpy")
landed, reward, terminated, action = g.series(backend="numpy")
stored = g.series(backend="python")
epsilon = g.series(backend="python")

observation[:t0] << start_observation
slope = (config.end_e - config.start_e) / (config.exploration_fraction * config.total_timesteps)
epsilon[t] << hp.call(max, slope * t + config.start_e, config.end_e)
action[t].via(act(q_network, observation[t], epsilon[t], coins, envs), after=fit[t - 1])
(observation[t + 1], landed[t], reward[t], terminated[t]) << env_step(envs, action[t])
stored[t] << remember(replay, observation[t], action[t], reward[t], landed[t], terminated[t])

# --- one batch, at upstream's fitting steps
drawn = g.series(backend="numpy", lifting="never")
drawn[t].via(dist.sample(dist.expand(dist.integers(0, config.buffer_size), config.batch_size), picks),
             before=action)

batch_observation, batch_action, batch_reward, batch_next, batch_done = g.series()
batch_observation[t] << stored[t].observations[drawn[t]]
batch_action[t] << stored[t].actions[drawn[t]]
batch_reward[t] << stored[t].rewards[drawn[t]]
batch_next[t] << stored[t].next_observations[drawn[t]]
batch_done[t] << stored[t].dones[drawn[t]]

target_max, td_target, old_val, loss = g.series()
target_max[t].via(array.stop_gradient(array.max(target_network(batch_next[t]), axis=1)),
                  after=sync[t - 1])
td_target[t] << batch_reward[t] + config.gamma * target_max[t] * (1 - batch_done[t])
old_val[t] << q_network(batch_observation[t]).gather(1, batch_action[t]).squeeze()
loss[t] << hp.call(functional.mse_loss, td_target[t], old_val[t])

fit[first_fit::config.train_frequency] << optimize(optimizer, loss[t])
sync[first_sync::config.target_network_frequency].via(
    soft_update(q_network, target_network, config.tau), after=fit)
```

D.8 PROTOCOL

Four of the nineteen questions above are the references' own: three of DDPM's and PPO's continuous-action variant. We chose the rest as questions the model's code was not written to answer, the case C3 is about: GPT-2's four from interpretability research, and the other three of DDPM's, the VAE's three, the GAN's three and nanoGPT-GRU's two as uses each reference reaches only through new code or an edited training loop. We left out variants PyTorch writes in a line, a least-squares GAN loss and spectral norm among them. A variant that adds parameters, such as multi-token prediction's extra heads, is easier as an `nn.Module`, whose new submodules reach the optimizer by themselves. We settled the set while counting: of the questions first asked, we dropped those both sides answer with methods they already have (the VAE's encode, decode and sample), two that measured a mistake rather than a question (the GAN's severing and ordering), and two that came out about even (the VAE's importance-weighted bound and sampling from the GAN).

For every question, we print what each side writes to answer it. In PyTorch that is the reference's own code where it already answers the question, and otherwise code we wrote. We answer a question about how the model trains, or what it is built from, by editing the reference's code to branch on a new option, so that the file keeps its own beside the new one as the graph does, and print the edit as a diff against it; we answer any other with a new function. In the graph it is the code that declares the question's edges, every function of its file that code calls and the model's own code does not, and the call that asks it, which we leave out where a question changes only the value a training call pins.

We count both sides in Python tokens, leaving out comments, docstrings and type annotations, which only the code we wrote carries. A count of lines would change with how each statement is wrapped, and a count of statements with how many values one statement declares, and we wrote many of the answers ourselves. Where the reference already answers a question, we count the tokens its authors wrote for it. Each side's count includes the call that asks the question, except where an edit trains through the call its reference already makes, and each table lists the graph's edges beside its count. We print no type annotation, as we count none.

We declare each graph as its reference is written, with no condition or value that only a question uses. A question that replaces one of the graph's edges takes the edge out and declares it again under a condition beside its own, and we print its code as an edit of the declaration. The tables call these edges guarded. We count every change as a diff, so a line that an edit only moves under a new `if` or `with` costs nothing, and the new `if` or `with` does.

We typeset each block from the file the experiment wrote beside its corpus row and change nothing in it but the indentation and the annotations, closing up a signature they had wrapped, and leave out what the legend says a listing skips, so the counts in the text and the code here come from one record.

Before it records anything, each experiment checks that its graph computes what its reference computes, and holds each PyTorch answer we print to the graph's answer to the same question. On the CPU in float32, each demonstration's outputs, and its weights after several training steps, equal the reference's to the last bit; for the recurrent model that reference is nanoGPT's `model.py` edited to build a GRU in each block, which we wrote. An edit runs a second time at the option value that keeps the reference's loss, held to its weights.

Section 7 runs each measured graph's short check again with `lifting="never"`. Where training lifts, the two runs agree within 10^{-5} of each output's largest value on nanoGPT, nanoGPT-GRU and GPT-2, and bit for bit on DDPM. We ran GPT-2's check at 64 positions, because a backward pass over 1,024 run step by step does not fit in memory. The comparison catches two errors we planted. Attention whose whole-series body drops its causal mask makes the two runs disagree. Without the VAE's `after=`, which makes each batch wait for the update before it, the lifted run moves every batch's forward pass ahead of the first update, and PyTorch fails. It cannot stack the epoch's short last batch with the others, and with equal batches the second update backpropagates through a graph the first has freed. The library itself raises nothing about the order. Run step by step, the same graph still matches its reference. The training loops of DQN, the VAE and the GAN lift nothing. PPO's rollout plans step by step only once a `before=` says which of two loops computes its log-probabilities, and the two runs then agree bit for bit.

E FURTHER RELATED WORK

Letting a tool find the order from declared dependencies is an old idea at the granularity of tasks, from `make` (Feldman, 1979) and the build systems later work systematised (Mokhov et al., 2018) to Snakemake (Mölder et al., 2021) and Dask (Rocklin, 2015), which schedule files and coarse units of work with no notion of a step. Snakemake refuses to build a file its rules produce ambiguously until the user ranks the rules or constrains their wildcards (Mölder et al., 2021), the refusal of Section 2 at the grain of a file. Synchronous dataflow languages bring the idea to time. A Lustre program defines each stream by one equation, in any order, reads a stream's value at the previous instant with `pre`, and compiles into one loop over instants (Halbwachs et al., 1991). With one equation per stream there is no route to search, and the program's `pre` reads fix what it remembers, leaving its compiler only how to store it.

Lucid (Ashcroft & Wadge, 1977) wrote iteration as equations over the successive values of a variable, and Lustre's designers took it as their starting point. They describe Lustre as Lucid restricted to a model in which a new value can only be appended to the end of a sequence, where Lucid lets one be placed anywhere in it, and then restricted further until memory and reaction time were bounded (Halbwachs et al., 1991). We take the other side of that trade. An edge may read a whole prefix, or the next step of a series declared to run backward, and the library derives for each question what its loop keeps.

Halide (Ragan-Kelley et al., 2013), TVM (Chen et al., 2018), Triton (Tillet et al., 2019), XLA (XLA and TensorFlow teams, 2017) and PyTorch 2 (Ansel et al., 2024) separate what is computed from how it is scheduled, over loop nests and memory inside a kernel or a traced region, and MLIR (Lattner et al., 2021) is an infrastructure for building such compilers. Halide infers each stage's buffer from the region its consumers read, and folds it into a rolling window when a loop walks it in order, as Section 3 does for a series. There the user or an autotuner writes where each stage is computed and stored. Dex (Paszke et al., 2021) and Futhark (Henriksen et al., 2017) recover scheduling freedom from purity within an expression. Each of these compilers takes the program's composition as given, and we hand our derivation to one of them.

JAX's `vmap` and `scan` (Bradbury et al., 2018) work a layer below the planner. `vmap` turns a function written for one example into one over a batch by transforming each operation it performs, and `scan` compiles a loop as one operation; neither decides which computation needs a loop, which way it runs or what it keeps, which is what the planner derives. Each could run part of a plan. A transform could lift a rule in place of a declaration, where the backend has one and the body allows it: NumPy has none, and `torch.func.vmap` refuses a module that updates its own buffers as it runs, such as batch normalisation in training mode. A scan could run a loop in place of tracing it whole or calling a compiled region at every step (Appendix C).

Probabilistic programming systems such as Pyro (Bingham et al., 2019) and Stan (Carpenter et al., 2017) turn a declared model into an inference procedure, the same move as ours specialised to one target. einops (Rogozhnikov, 2022) contributes no new mathematics and makes an intent declarable and a class of shape error catchable where it is written, and we argue the same about composition and time.

Dyna (Eisner & Filardo, 2011) generalises Datalog to equations over items that carry values, and leaves the order of computation, indexing and storage to an execution model, as a database engine chooses how to run a query. Its authors present that model's strategies as an open research programme, asking whether a generic engine can find the algorithms and data structures an expert would pick by hand. `hyperpath` takes up a narrower form of the question for one shape of program, and derives the loop over time, what each series keeps and which edges run once over a whole series. The two also differ where several rules define one value. Dyna aggregates their contributions, summing them or taking a minimum, and its `=` aggregator reports an error when more than one contributes. A `hyperpath` graph declares the choice per value. By default several edges producing one value are alternatives, the planner derives the value along whichever one a question's inputs reach, and a question that leaves more than one of them open raises before anything runs. A value declared `combine="sum"` instead adds up its edges, each written `+=` or `-=`, and an edge under a condition joins the sum only where a call pins it. A question then adds a term to a loss with one line, as the GAN's gradient penalty does in Appendix D. A sum gives the planner nothing to pick between, so it refuses a question that cannot derive every term instead of adding up only the ones it can.

TorchRL (Bou et al., 2024) wraps each module to read and write named entries of a TensorDict, a dictionary of tensors that share their leading dimensions, and chains the wrapped modules in the order the user lists them, as `nn.Sequential` does. A module there names the values it reads and writes, as an edge does here, and a chain can be cut down to the modules that depend on one entry. The order of the chain stays the user's. Components written for the purpose run the loops over time: a collector runs the policy against the environment, a value estimator such as GAE (Schulman et al., 2016) runs back over the rollout, and a trainer or the user's own loop runs the updates.

For the questions we ask of GPT-2, TransformerLens (Nanda & Bloom, 2022) exposes every internal activation of the language models it loads, to cache or to replace as the model runs, and NNsight (Fiotto-Kaufman et al., 2025) wraps the modules of an unmodified PyTorch model and runs the reads and edits a user writes inside a tracing context in step with the model's forward pass. Both make cheap the kind of question C3 asks of GPT-2, one the model's own code was not written to answer, wherever the question reads or replaces an activation. We write our PyTorch answers with forward hooks because C3 compares the graph with plain PyTorch on every model, and answering GPT-2's four questions with one of these libraries would make those four a different comparison. We report NNsight's answers to the three that need a hook beside them, as a second baseline, in Section 5.3 and Appendix D.