

Learning distributed training through PS-Env

A practical account of what we built, measured, broke, fixed, and learned.

Evidence snapshot: 14 September 2026, approximately 04:10 Qatar. The overnight campaign was still running at this snapshot. Figures below describe retained or directly inspected results, not promised final outcomes. Read this in order the first time; use the glossary and code map afterward.

- Learning distributed training through PS-Env
 - 1. What we achieved—and what remains unsolved
 - 2. The journey through the project
 - 3. Vocabulary: the units that matter
 - 4. How a single PS-Env learner works
 - The actual tensor shapes in our code
 - 5. Sharing across levels versus sharing across GPUs
 - 6. Exactly how our distributed update works
 - 7. How we checked correctness
 - 8. Reading the scaling result correctly
 - 9. The separate transformer experiments
 - The small two-node pilot: more GPUs were slower
 - The expanded campaign
 - GPU reference: what we used, what we did not
 - The machines behind the GPU counts
 - Capacity, bandwidth and compute answer different questions
 - Why A10G was a useful choice here
 - 10. Why we deliberately caused out-of-memory errors
 - Shapes of the separate transformer benchmark
 - 11. Demonstrations and the gameplay learning experiments
 - Four names that describe different parts of training
 - Comparison 1: what each term commits us to
 - Comparison 2: what happens during training and gameplay
 - Comparison 3: benefits, limits and convincing evidence
 - The concrete recipes we compared in code
 - Worked example: an obstacle near the end of a level
 - How to describe a model precisely in the presentation
 - 12. Reward design: what the results actually tell us

- [13. Infrastructure is part of the experiment](#)
- [14. Failures and the lessons they gave us](#)
- [15. How to evaluate the next experiment](#)
- [16. Exercises using our own project](#)
- [17. Code and evidence map](#)
- [18. Where open robotics models fit](#)
 - [Representative open projects](#)
 - [Why we cannot just load one into Crash Bandicoot](#)
- [19. In-context learning: change the examples, not the weights](#)
 - [What changes for distributed training?](#)
- [20. Connecting this to Edward Johns and Vitalis Vosylius](#)
 - [A concrete training detail from Instant Policy](#)
 - [What to emphasize in the conversation with Ed](#)
- [21. Small code examples and a practical next experiment](#)
 - [Example A: context can change an output without an optimizer](#)
 - [Example B: training an imitation model with DDP](#)
 - [Example C: an action output still needs the correct interpretation](#)
 - [A next experiment that connects the ideas](#)

1. What we achieved—and what remains unsolved

We built a real-game reinforcement-learning system for the first five levels of Crash Bandicoot: Warped, a shared model that can act in all five levels, demonstration-based training, evaluation and films, cloud infrastructure, and two distinct distributed-training experiment tracks.

The strongest completed real-game systems result is **3.32× higher end-to-end training throughput with four GPUs**, compared with one, at matching global experience and optimizer-update budgets. Two GPUs achieved approximately **1.86×**. Both results repeated closely across two seeds.

This does **not** establish improved playing ability. All six models in that scaling comparison scored **0/20 evaluation completions**. Separate demonstration-trained models complete Orient Express from the demonstrated starting state, but small changes in starting timing still break that success. The longer overnight campaign had not produced an additional level completion at this snapshot.

The project therefore teaches three separate questions:

Question	Appropriate evidence
Can the system train faster?	Throughput at a controlled workload, repeated measurements

Question	Appropriate evidence
Can the system fit a larger workload?	Memory peaks and tested capacity boundaries
Does the trained policy perform better?	Completion, meaningful progress and robustness evaluations

A successful answer to one does not imply a successful answer to the others.

2. The journey through the project

We first extended PS-Env beyond one platforming level. The first five levels involve platforming, swimming, tiger riding, a chase, and jet-ski control. That required level-specific actions, memory interpretation, starting states and completion checks—not merely five names in a menu.

We added named presets, local/cloud execution, checkpoints, model loading, recorded gameplay and a hosted console. We then explored one shared visual model with separate action outputs for each level, alongside independent level-specific training.

Poor returns and repeated failures led us to inspect reward components, build scripted searches for candidate demonstrations, replay those candidates, and train policies to imitate their actions. We compared demonstration initialization, continued demonstration replay, learning from scratch, and practice from route sections.

A separate transformer project explored distributed systems in a more conventional neural-network workload: single-device training, DDP, FSDP, DeepSpeed ZeRO, NCCL communication, Megatron tensor parallelism, memory limits, tuning, profiling and recovery. We then returned those systems lessons to real PS-Env training using synchronous gradient sharing.

Finally, we expanded the real-game comparison to one, two and four GPUs with fixed update counts, and launched longer reward experiments and additional demonstration searches overnight.

The UI and presentation are different deliverables. At this snapshot the live experiments page contained the new six-run scaling comparison; the Machenta slide deck still contained the older 1.64x pilot result. A live dashboard updating does not automatically update a static presentation.

3. Vocabulary: the units that matter

Keep these three counters separate

A decision collects experience. An episode groups one attempt. An update changes the model. 200,000 decisions does not mean 200,000 training updates.

Term	Meaning in this project
Frame	One emulated game frame
Observation	The processed images the policy receives
Decision	One selected action, applied across eight game frames in these runs
Action	A level-specific button combination or macro
Transition	Observation, action, reward, next observation and terminal information
Episode	One attempt, ending in completion, game over, or an imposed step limit
Reward	A numerical feedback signal generated from game events and configured shaping
Return	The sum of rewards over an episode
Policy	The rule, represented by a neural network, that selects actions
Replay buffer	Stored past experience sampled again for learning
Batch	A set of replay samples used to calculate an update
Loss	The training objective whose gradient changes the network
Gradient	The derivative of the loss with respect to model parameters
Optimizer update	One application of an update rule such as Adam
Checkpoint	Saved model/training state; its exact contents determine what can be resumed
Rank	One process participating in distributed training
World size	The number of participating distributed processes
Local rank	A process's device index within its machine

Term	Meaning in this project
Seed	A setting controlling pseudorandom choices; it does not make different parallel executions identical

The policy receives four stacked 84×84 grayscale observations. Stacking helps it infer movement from changes between images. The stack contains successive sampled observations, not necessarily four adjacent game frames.

A decision is not necessarily a fresh greedy network choice: exploration can select or repeat an action. The frame-skip setting remains eight frames per decision, with level-specific button-tap behavior where required.

Example: 200,000 decisions correspond to about 1.6 million emulated action frames, excluding reset and evaluation overhead. With four workers, those are 200,000 decisions **in total**, not 200,000 per worker.

Episode return and training loss are also different. A decreasing DQN loss means its current sampled predictions fit its targets more closely. It does not establish that the targets encode desirable behavior, or that the policy can finish a level.

4. How a single PS-Env learner works

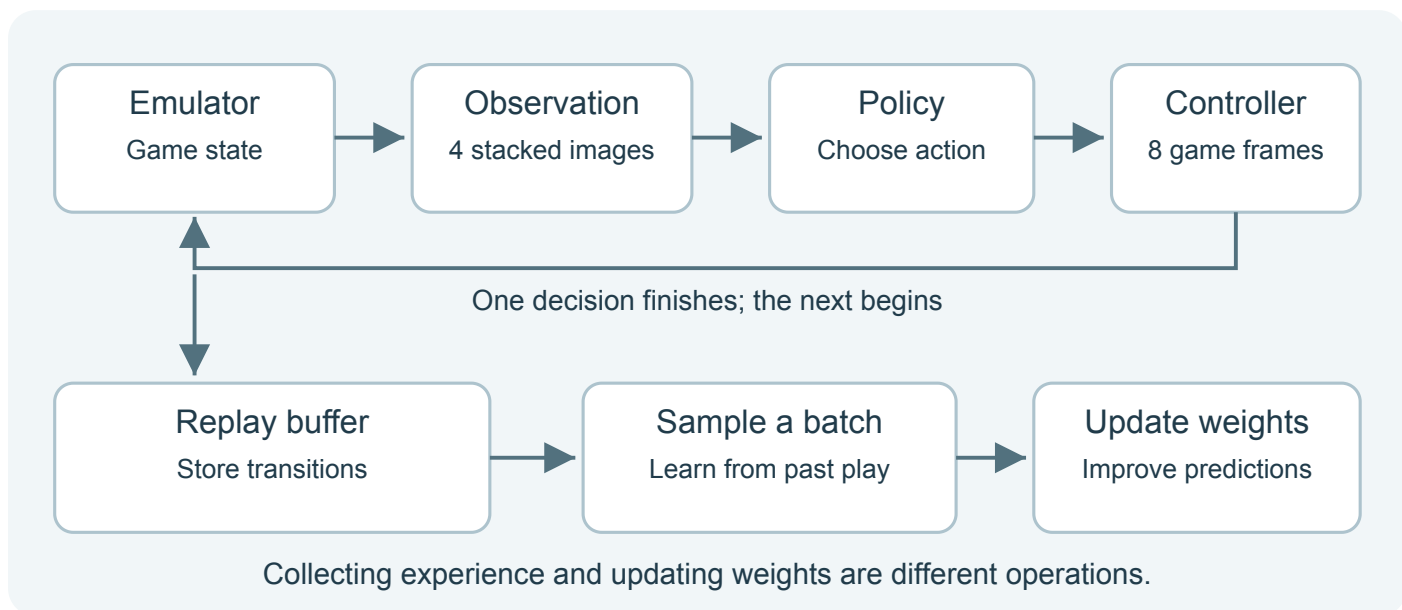


Figure 1. Two connected loops: playing creates data; learning uses that data to change the policy.

The model is a convolutional DQN. It predicts a value for each available action: how much future discounted reward taking that action is expected to produce. During greedy evaluation it chooses the

highest-valued action. During training an epsilon-greedy exploration mechanism sometimes chooses exploratory actions instead.

Double DQN separates choosing a bootstrap action from evaluating that action using the target network. A delayed target network helps stabilize the moving learning target. In simplified one-step notation:

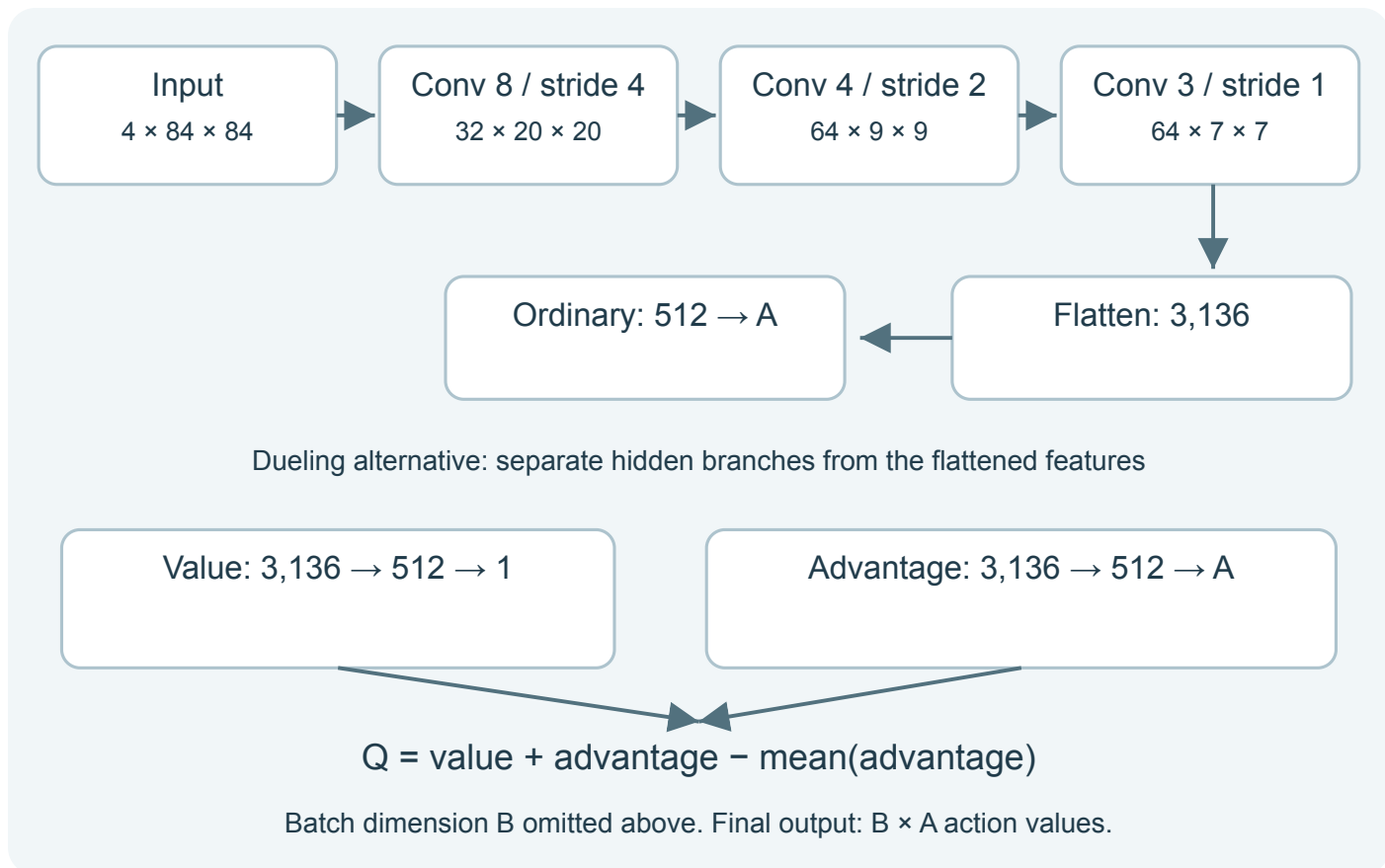
```
chosen_next_action = argmax Q_online(next_observation, action)
target = reward + gamma * Q_target(next_observation, chosen_next_action)
```

For a true terminal transition, there is no future-value term. Our implementation also supports three-step returns: it combines several consecutive rewards and bootstraps after that sequence. Episode and level boundaries must not be crossed accidentally. Artificial time limits need careful treatment because truncating an attempt is different from the game reaching a terminal state.

The implementation uses Adam, Huber loss, gradient clipping and target-network synchronization. These ingredients stabilize optimization; none guarantees successful exploration or an appropriate reward objective.

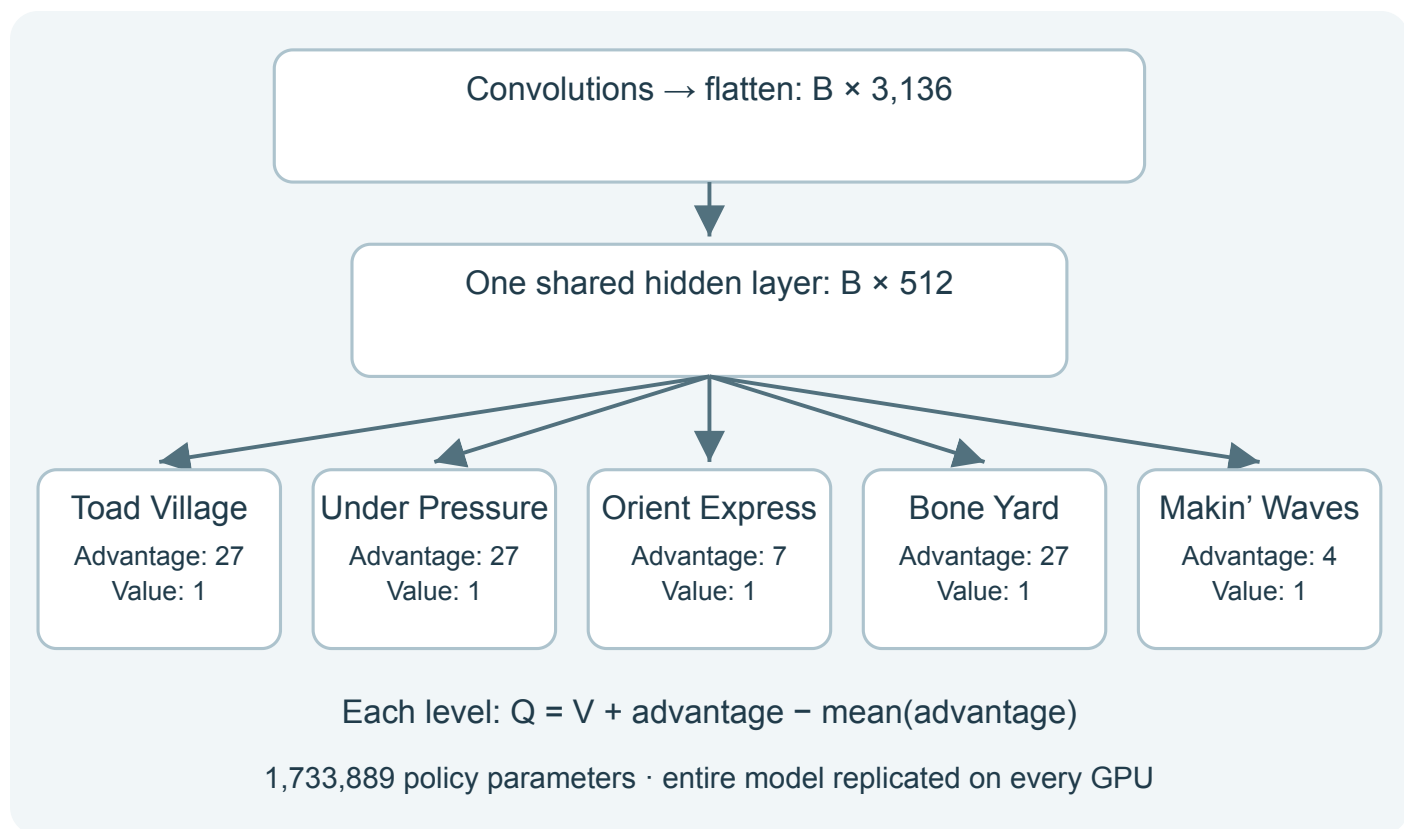
The actual tensor shapes in our code

`B` is the number of observations processed together, `A` is the number of discrete actions, and dimensions below are **batch × channels × height × width**. One grayscale image has one channel; stacking four images produces four channels. The convolutional network has no language input and is not a transformer.



Operation in <code>agent/model.py</code>	Output shape	Meaning
Input; convert uint8 to float and divide by 255	[B, 4, 84, 84]	Four sampled grayscale observations
Conv: 32 filters, 8x8 kernel, stride 4; ReLU	[B, 32, 20, 20]	Learn 32 local feature maps
Conv: 64 filters, 4x4 kernel, stride 2; ReLU	[B, 64, 9, 9]	Combine larger visual patterns
Conv: 64 filters, 3x3 kernel, stride 1; ReLU	[B, 64, 7, 7]	Last spatial representation
Flatten	[B, 3136]	$64 \times 7 \times 7$ numbers per observation
Ordinary single-level head	[B, 512] $\rightarrow$ [B, A]	One hidden layer followed by action values
Dueling single-level value branch	[B, 512] $\rightarrow$ [B, 1]	Predict state value
Dueling single-level advantage branch	[B, 512] $\rightarrow$ [B, A]	Predict relative action advantages
Dueling combination	[B, A]	$Q = v + \text{advantage} - \text{mean}(\text{advantage})$

The convolutions have no padding. For example, the first spatial dimension is $\text{floor}((84 - 8) / 4) + 1 = 20$. ReLU keeps its input shape. "512 units" describes a hidden vector, not 512 actions or 512 game states. The two single-level dueling branches have **separate** 3136 $\rightarrow$ 512 weight matrices.



The five-level `sharedDQN` reuses the same convolutional shapes, then has **one shared** 3136→512 layer. Each level has a small 512→A advantage head and a 512→1 value head. The caller supplies the level identifier to route the head; it is not concatenated as a language input. A forward call for one level produces that level's $[B, A]$ output, not a single concatenation of all five action spaces.

Network / controller	Output actions	Trainable policy parameters
Original ordinary NatureDQN, legacy controller	13	1,690,797
Single-level dueling, Toad Village / Under Pressure / Bone Yard, <code>level_v2</code>	27 each	3,304,636 each
Single-level dueling, Orient Express, <code>level_v2</code>	7	3,294,376
Single-level dueling, Makin' Waves, <code>level_v2</code>	4	3,292,837
Five-level shared dueling, <code>level_v2</code>	27 / 27 / 7 / 27 / 4	1,733,889 total

These counts were computed from instantiated code, include biases, and count **one policy**, excluding its delayed target copy, optimizer state, gradients, replay and activations. Historical checkpoints can use different flags: inspect their saved `config` rather than assuming every Toad Village model uses 27 actions. The shared model is smaller than a single-level dueling model because it shares the large hidden layer rather than maintaining two separate 3136→512 branches.

In DQN the target network has the same shape as the online policy, but is updated by copying weights rather than by its own optimizer. Double DQN concerns target construction; dueling concerns the output branches. They are independent choices. With 1/2/4 GPUs in our PS-Env benchmark, **every GPU receives the complete policy and target network**; adding GPUs does not add layers or split this model into pieces.

```
from agent.model import NatureDQN
import torch
net = NatureDQN(num_actions=7, dueling=True)
x = torch.zeros(2, 4, 84, 84, dtype=torch.uint8)
print(net(x).shape) # torch.Size([2, 7])
print(sum(p.numel() for p in net.parameters())) # 3294376
```

5. Sharing across levels versus sharing across GPUs

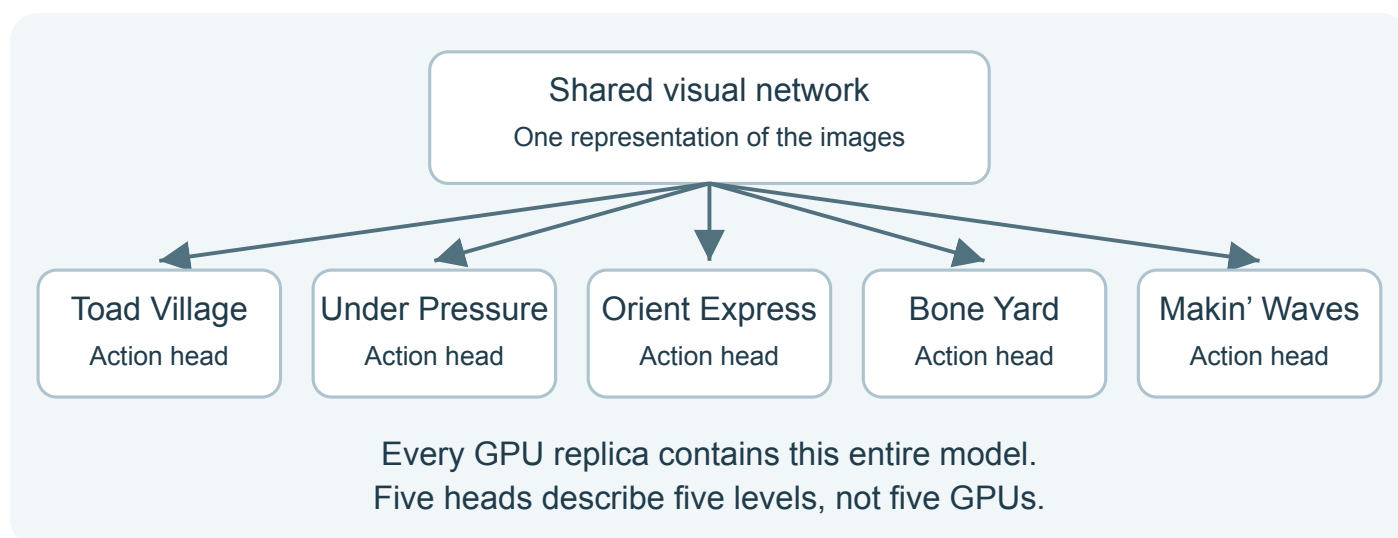


Figure 2. This entire five-head network is copied onto every GPU in a distributed run.

These are independent design choices.

Across levels: our shared model has one visual feature extractor and a shared hidden representation, followed by five level-specific action heads. An action suitable for swimming need not have the same meaning as an action suitable for tiger riding. Balanced replay gives all five levels a contribution to learning.

Across GPUs: synchronous data parallelism places a complete copy of that entire shared model on every GPU. Different workers gather different experience, but synchronize the model updates.

Therefore, four GPUs do not mean four heads, and five levels do not require five GPUs. Every distributed replica contains all five heads. Each worker assigns its next episode to a level with relatively little collected experience.

There is also **parallel experimentation**: four GPUs can run four unrelated trials with different rewards or methods. Those models never average gradients. That is how the overnight gameplay ablations are scheduled. It is different from the one-/two-/four-GPU scaling trials, which run one distributed experiment at a time.

6. Exactly how our distributed update works

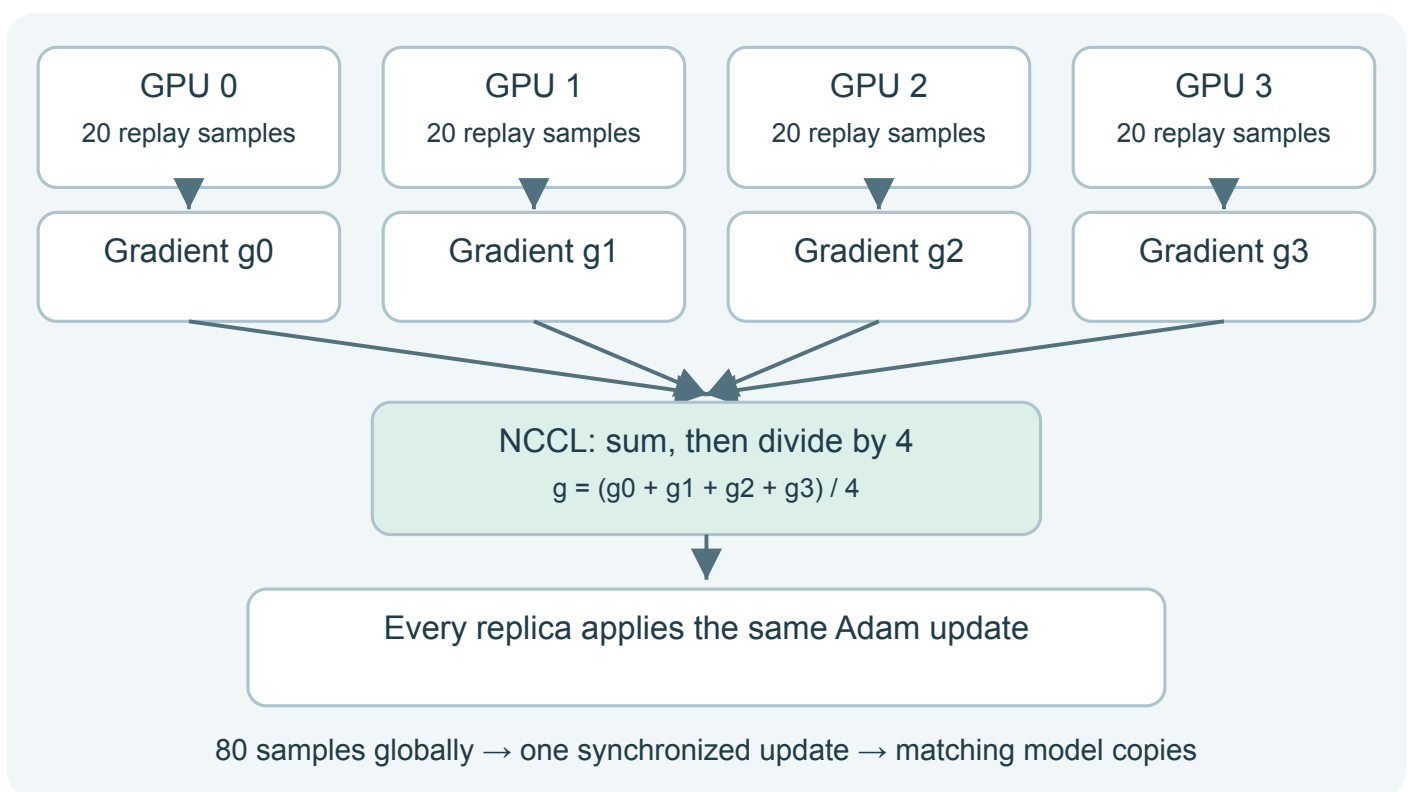


Figure 3. The batch stays at 80 globally. The arrows entering NCCL carry gradients, not raw gameplay.

Our PS-Env implementation uses `torchrun` to launch one process per GPU. PyTorch initializes the process group; CUDA workers communicate through NCCL. CPU validation uses Gloo.

Each rank owns:

- A complete online network, target network and optimizer.
- A local native emulator and independent exploration randomness.
- Replay buffers partitioned by level.

Initial model parameters are broadcast from rank zero. During training, each rank gathers transitions locally. At the configured update boundary, all ranks check whether their replay partitions are ready.

They then sample equal contributions from the five levels.

For the overnight comparison, the global batch is always 80:

GPUs/ranks	Samples per rank	Samples per level per rank	Total across five levels and all ranks
1	80	16	80
2	40	8	80
4	20	4	80

Each rank calculates its local loss and gradients. The implementation flattens the gradients into one tensor, sums it across ranks with `all_reduce`, divides by the number of ranks, copies the averaged gradients back, clips them, and performs Adam's update.

If the four gradients for one parameter were 2, 4, 6 and 8, the averaged gradient would be 5. Adam then uses that gradient together with its optimizer state; the parameter change is not necessarily just minus five times the learning rate.

Equal local sample counts matter. Averaging worker means is equivalent to averaging all samples only when workers have appropriate equal weights. Unequal batches require weighted aggregation. We validate exact batch divisibility instead of silently dropping samples.

What travels between GPUs? Primarily the gradients, plus synchronization/control information and occasional statistics. Raw replay experience stays local. This is not a central experience database, and replay is not shared simply because the model weights are synchronized.

This implementation uses explicit gradient averaging. It follows the synchronous data-parallel principle but does **not** wrap the PS-Env model in `PyTorch DistributedDataParallel`. The production DDP wrapper offers mechanisms such as gradient bucketing and communication/computation overlap that our straightforward flattened implementation does not claim to reproduce. See [PyTorch DDP](#) and [distributed communication](#).

7. How we checked correctness

Matching replicas is a correctness check

Identical weights show that the workers stayed synchronized. They do not prove that the policy learned useful behavior.

We checked three different things, each answering a different question:

1. **Reference-gradient agreement:** on identical global data, averaged gradients matched a reference learner within approximately 1.27×10^{-7} maximum absolute difference.
2. **Replica agreement:** all distributed replicas ended with identical policy hashes. This detects divergence between workers; it does not prove the objective is correct.
3. **Real-game plumbing:** short runs exercised actual emulators, updates, checkpoints, evaluations and films. Short smoke-test episodes are not evidence of mastery.

The latest two-GPU smoke collected 800 global decisions and performed exactly 11 scheduled updates, with identical replica weights. A separate five-level demonstration/reward smoke completed 2,000 decisions, 40 updates and 20 evaluation conditions.

Testing the inner algorithm alone was insufficient: a later command-line validation bug prevented all six search jobs from starting even though the underlying search function had passed a direct smoke. We fixed the CLI and added tests with and without a prefix. Test the actual entry point that automation launches.

8. Reading the scaling result correctly

Worked example: why 3.32x?

Four GPUs collected about 202 decisions per second, versus 61 on one GPU. $202 \div 61 \approx 3.32\times$ speedup. Dividing by four gives about 83% scaling efficiency—not 83% GPU utilization.

The improved comparison ran 200,000 global decisions per trial, a global batch of 80 and a global replay capacity of 40,000. Updates started at decision 50,000, inclusive, and occurred every 20 global decisions:

```
number of updates = (200,000 - 50,000) / 20 + 1 = 7,501
```

If any replay partition was unready when this fixed schedule began, the trial failed rather than silently changing the update count. All six trials passed.

GPUs	Seed 42 decisions/s	Seed 43 decisions/s	Approximate speedup	Scaling efficiency
1	60.95	60.87	1x	100%
2	113.47	112.81	1.86x	93%

GPUs	Seed 42 decisions/s	Seed 43 decisions/s	Approximate speedup	Scaling efficiency
4	202.20	202.64	3.32×	83%

Speedup is throughput divided by the one-GPU throughput. Scaling efficiency is speedup divided by GPU count. It describes how much of ideal linear scaling we achieved; it is not GPU utilization.

The workload took approximately 55, 29 and 16½ minutes respectively, before subsequent evaluation. These are end-to-end training measurements including experience collection and training-loop overhead, not isolated neural-network kernel timings. The one-/two-/four-GPU runs used the same four-GPU host with different numbers of active training devices.

The result is encouraging because it repeats across seeds at matching update counts. It is limited because two seeds are few, trajectories still differ, and fixed-start evaluations do not establish broad generalization. All six scored 0/20 completions.

The earlier pilot measured 51.54 versus 84.33 decisions/s, or 1.64×. However, it performed 20,801 versus 18,764 updates. Different replay readiness changed the amount of learning work. That pilot motivated the explicit fixed-update schedule. Do not present the change from 1.64× to 1.86× as an isolated optimization gain: the benchmark protocol also changed.

9. The separate transformer experiments

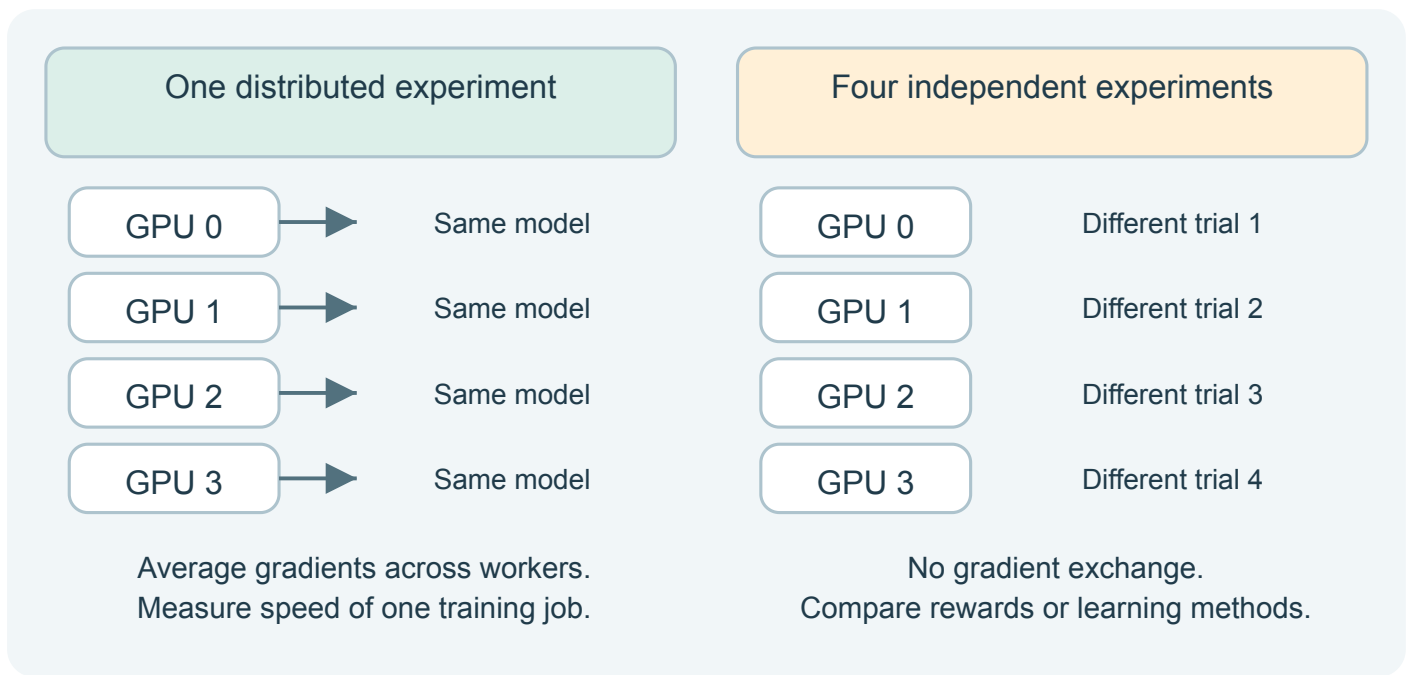


Figure 4. Both arrangements use four GPUs, but they answer different experimental questions.

The transformer track trained a small synthetic next-token prediction task. A token is a sequence element; tokens/s is not interchangeable with game decisions/s. The task was deliberately simple, not a claim of language-model quality.

Technique	What is distributed	Main reason to use it
DDP	Different batches; complete model copies	Increase training throughput when communication is affordable
FSDP	Model parameters, gradients and optimizer state are sharded	Fit larger model state across devices
DeepSpeed ZeRO-1	Optimizer state	Reduce optimizer-state duplication
ZeRO-2	Optimizer state and gradients	Save more training memory
ZeRO-3	Optimizer state, gradients and parameters	Further reduce replicated model state
Tensor parallelism	Computation within model layers	Split operations too large or costly for one device
Pipeline parallelism	Different layer groups	Distribute model depth; not used in our Megatron experiment
Activation checkpointing	Some activations are discarded and recomputed	Trade additional compute for lower activation memory

FSDP and ZeRO are different implementations with related sharding goals. NCCL is the communication library they may use, not another sharding strategy. See [PyTorch FSDP](#), [DeepSpeed's ZeRO tutorial](#) and [PyTorch's FSDP/activation-memory discussion](#).

We used genuine Megatron Core tensor parallelism, with pipeline size one. Its model and kernels differed from TinyGPT, so its throughput is not an apples-to-apples framework speed comparison.

The small two-node pilot: more GPUs were slower

Fourteen GPU trials completed on two separate A10G instances using private TCP networking. For the same small FP32 TinyGPT workload:

Trainer	GPUs	Median tokens/s
Single device	1	106,132
DDP	2	29,660

Trainer	GPUs	Median tokens/s
FSDP2	2	17,203
DeepSpeed ZeRO-2	2	19,677

These measurements are consistent with communication overhead outweighing useful parallel work for a small model. They do not prove a universal framework ranking. FSDP had the lowest measured allocation peak in this pilot. DeepSpeed's default communication buckets were large relative to the tiny model, so its memory peak was not a tuned comparison.

A separate 16 MiB NCCL reduction test verified its numerical output and measured about 26.03 ms mean latency, with 0.644 GB/s effective algorithm bandwidth. That is not physical-link bandwidth. A small Megatron run completed at 21,847 tokens/s, but used a different architecture and only one repetition.

The expanded campaign

The retained final phase summaries contain **139 completed trials, 18 OOM outcomes and 3 other failures**:

Phase	Complete	OOM	Other failures
Smoke	7	0	0
Scaling	87	3	0
Tuning	15	0	0
Profiling	5	0	0
Memory capacity	18	6	0
Expanded Megatron	0	9	0
Recovery	7	0	3

Thus the smaller Megatron pilot succeeded, but the expanded Megatron configurations did not fit. Those statements are compatible and should remain separate.

Actual accelerator work here was on **A10G**, including a four-GPU g5.12xlarge host. Requested H100 capacity did not turn into an H100 run. We do not claim hands-on H100, H200 or B200 results from these experiments.

GPU reference: what we used, what we did not

A GPU name describes an accelerator. An AWS instance name describes the whole rented machine: GPUs, CPUs, host memory and networking. The two names are not interchangeable.

Hardware	Project status	Memory per GPU	What that means for our work
NVIDIA A10G	Used for real-game training and transformer benchmarks	24 GB nominal	The hardware behind our measured throughput, memory and correctness results
NVIDIA H100 SXM	Requested, but no completed run ; capacity attempts did not produce an H100 experiment	80 GB HBM3	More room for model state and activations; unmeasured in this project
NVIDIA H200 SXM	Not used	141 GB HBM3e	More memory capacity and bandwidth than the listed H100 variant; no project benchmark
NVIDIA B200 SXM	Not used	180 GB HBM3e	A larger-memory accelerator platform; no project benchmark

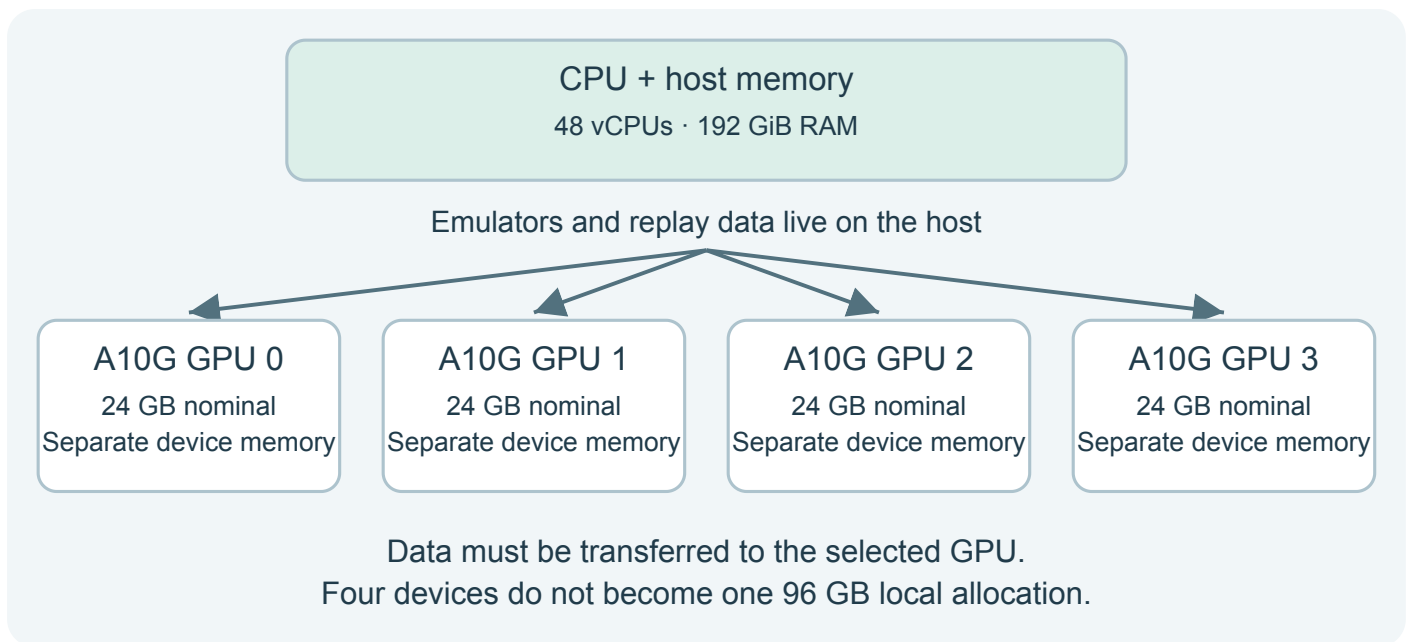
These are specific product variants. For example, H100 variants do not all have identical memory specifications. The H100/H200/B200 figures above refer to NVIDIA's **HGX SXM** comparison, not every cloud offering bearing those names. NVIDIA lists their respective device-memory bandwidths as **3.35, 4.8 and up to 8 TB/s**. These are manufacturer specifications, not rates measured by our code. Sources: [AWS G5/A10G specifications](#) and [NVIDIA HGX hardware specifications](#).

The machines behind the GPU counts

Machine arrangement we used	GPUs available	Host CPU resources	Host RAM	Role
One AWS g5.xlarge	1 × A10G	4 vCPUs	16 GiB	Individual workers; two separate instances formed the early two-node benchmark
One AWS g5.12xlarge	4 × A10G	48 vCPUs	192 GiB	Expanded experiments and the real-game 1/2/4-GPU comparison

A vCPU is an AWS scheduling unit, not a claim of one dedicated physical core. Instance resources are documented in [AWS's accelerated-instance specification table](#). The one-GPU PS-Env scaling baseline

used one active GPU **inside the four-GPU host**. It was not a comparison against the smaller g5.xlarge's CPU and memory resources.



The four-GPU host has four separate device-memory spaces. Sharding changes how software places data across them; it does not turn them into one ordinary local allocation.

4 × 24 GB does not mean one 96 GB GPU

A tensor allocated on GPU 0 must fit in GPU 0's available memory. Using the aggregate capacity requires software to partition the workload. A replicated DQN still stores a complete model on each device.

Host RAM holds CPU-side data, including our emulator state and replay arrays. **GPU memory (VRAM)** holds device tensors and neural-network workspaces. Extra host RAM does not automatically fix CUDA out-of-memory errors. Explicit offloading can move selected state to the CPU, but introduces data movement; our PS-Env learner does not claim an offloading implementation.

GB and GiB differ. GB normally denotes billions of bytes; GiB denotes powers of two. AWS markets A10G as 24 GB, while its technical instance table reports roughly 22 GiB per device. Unit conventions, reservations and runtime availability matter. Use the actual runtime memory report and measured allocation headroom when sizing a job, rather than treating the product label as guaranteed allocatable space.

Capacity, bandwidth and compute answer different questions

Hardware property	Plain-language question	Possible symptom when it is limiting
Memory capacity	How much can remain on the device at once?	Out-of-memory failure
Device-memory bandwidth	How quickly can the GPU read and write its own memory?	Operations spend substantial time moving local data
Compute throughput	How much arithmetic can the device execute?	Expensive matrix operations dominate the step
GPU-to-GPU connectivity	How quickly can workers exchange gradients or model shards?	Synchronization or parameter transfer dominates
CPU and input throughput	How quickly can the rest of the system supply useful work?	GPUs wait while emulators, preprocessing or transfers run

HBM means **high-bandwidth memory**. Its bandwidth figure describes access to memory attached to a GPU; it is not internet speed or NCCL collective throughput. NVLink/NVSwitch describe inter-GPU connectivity in applicable systems. The cited NVIDIA HGX systems use these links; using NCCL on our A10G instances does not mean we measured an HGX/NVSwitch fabric. Our early two-node pilot communicated over private TCP networking.

Why A10G was a useful choice here

We could provision it and verify the actual software stack, whereas the H100 capacity attempts did not yield a runnable experiment. It supplied enough GPU memory for the PS-Env network and enough devices to study one-/two-/four-worker synchronization. The four-GPU machine also provided CPU and host-memory resources for emulators and replay.

For the larger transformer configurations, A10G's memory boundary was useful evidence: it let us observe when replication failed, when sharding helped, and when activation checkpointing substantially increased the tested batch size. That does not mean A10G is universally the cheapest or fastest choice.

A larger accelerator would make sense as a **new experiment** when a representative workload is demonstrably memory-, compute-, or communication-limited. It would not automatically improve the reward function, extend an incomplete demonstration, or teach recovery from unfamiliar game states. We have no measured H100/H200/B200 speedup to substitute into our A10G results.

A sensible next hardware comparison would keep the model and learning protocol fixed, measure end-to-end throughput and quality, and record CPU resources, interconnect topology, GPU memory peaks, precision and software versions. GPU price alone would be insufficient: the relevant practical quantity is the cost and elapsed time to reach the desired evaluated result.

10. Why we deliberately caused out-of-memory errors

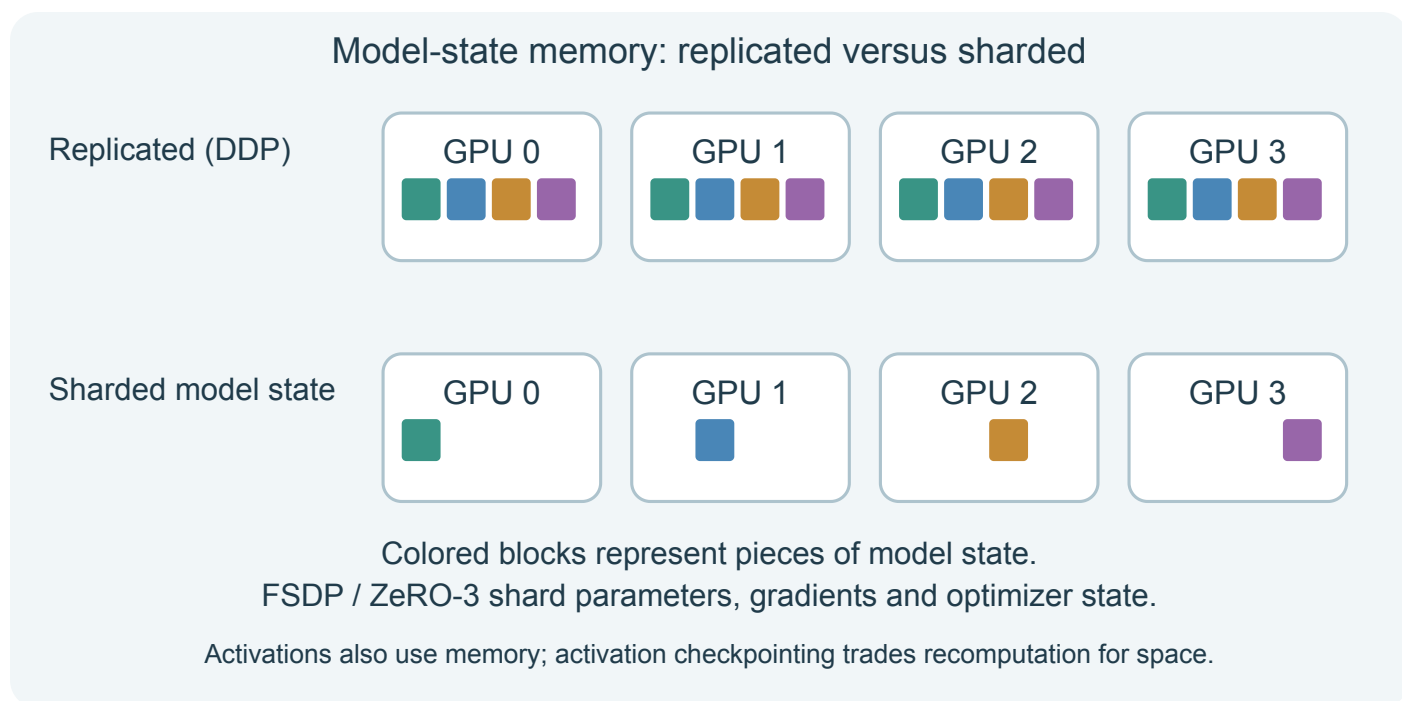


Figure 5. This is a conceptual picture of model-state storage. Activations and temporary buffers also consume memory.

GPU memory holds several things: model parameters, gradients, optimizer state, intermediate activations, communication buffers and temporary allocations. Sharding model state reduces some of these categories. It does not automatically eliminate activation memory.

In a controlled capacity probe we increase batch size until the workload no longer fits. A recorded OOM is an expected boundary observation, provided the runner captures it and proceeds safely. An unexpected OOM in a normal scheduled run is still a failure. We must not relabel every failure as deliberate.

For the tested larger BF16 transformer at sequence length 2,048, the retained capacity results were:

Configuration	Largest tested fitting global batch	Next tested batch: OOM
Single GPU	8	16
DDP, 4 GPUs	16	32
FSDP / ZeRO-2 / ZeRO-3, 4 GPUs	32	64
FSDP + activation checkpointing, 4 GPUs	256	512

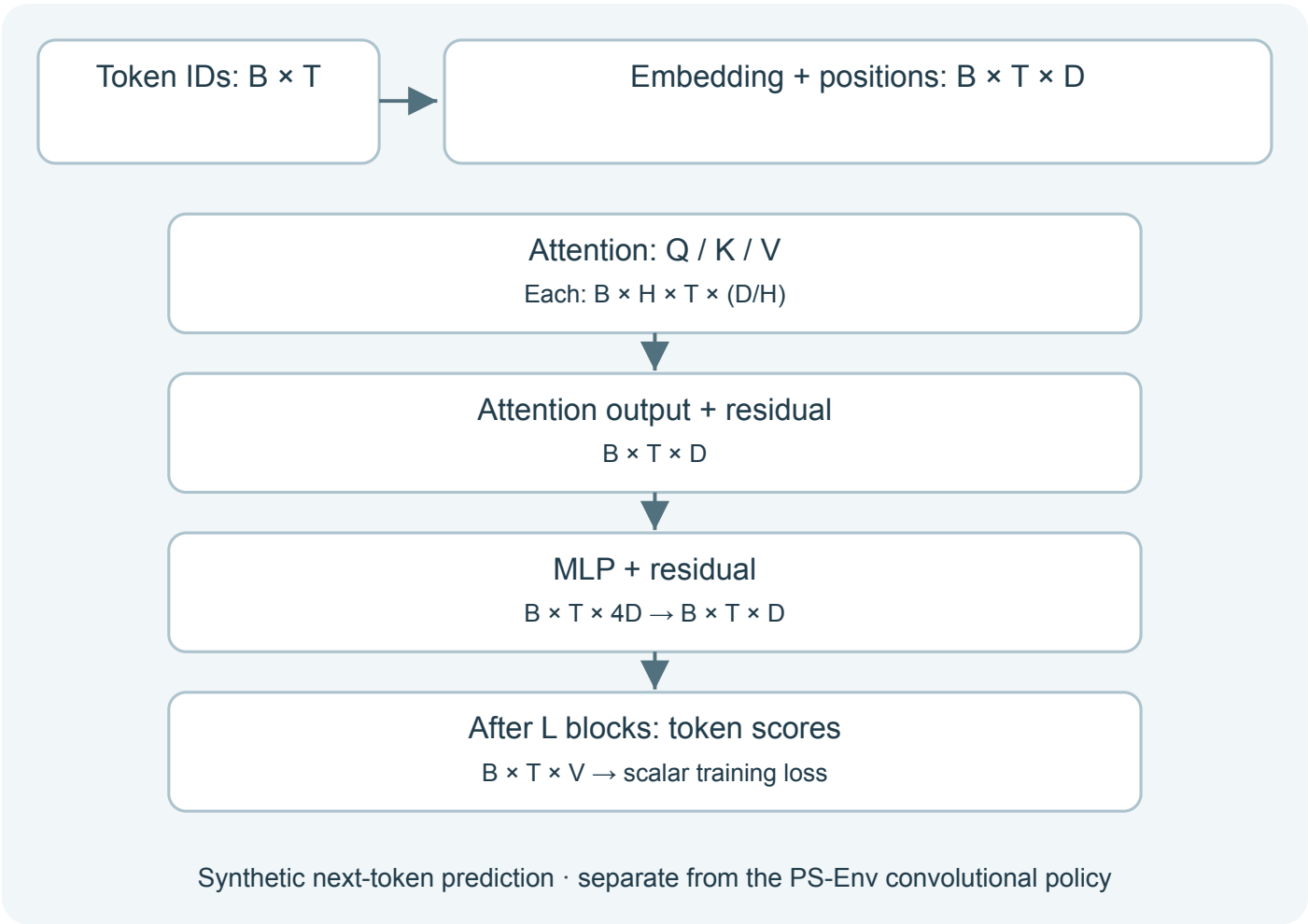
These are sampled boundaries for one model/configuration, not exact mathematical maxima or promises for other workloads. Capacity probes were short; fitting does not establish long-run

convergence. Activation checkpointing's large improvement shows why identifying the category consuming memory matters.

When reading memory statistics, distinguish allocated tensor memory, allocator-reserved memory and process/device usage reported by `nvidia-smi`. They measure different things. Likewise, high GPU utilization does not by itself prove high useful throughput, and low utilization does not by itself identify the bottleneck.

Shapes of the separate transformer benchmark

The transformer benchmarks train `bench/model.py::TinyGPT` on a synthetic next-token task, not gameplay images. Here τ is sequence length, D hidden width, H attention heads and V vocabulary size. These **attention heads** are not our five game-level output heads.



Stage	Shape	Operation
Token IDs	$[B, T]$	Integer indices
Token embedding + position embedding	$[B, T, D]$	Look up learned vectors and add them

Stage	Shape	Operation
LayerNorm → QKV projection	$[B, T, 3D]$	Produce query, key and value vectors
Reshape each Q / K / V	$[B, H, T, D/H]$	Split the feature dimension across attention heads
Causal attention + output projection + residual	$[B, T, D]$	Mix information from earlier tokens
LayerNorm → MLP → residual	$[B, T, 4D] \rightarrow [B, T, D]$	Expand features, GELU, contract features
Repeat transformer block L times; final LayerNorm	$[B, T, D]$	Preserve the outer dimensions
Vocabulary projection	$[B, T, V]$	Predict next-token scores; cross-entropy produces a scalar loss

Campaign preset	Blocks L	Width D	Attention heads H	MLP width	Sequence T	Vocabulary V
Small	4	256	8	1,024	128	4,096
Medium	12	768	12	3,072	512	4,096
Large	16	1,536	16	6,144	1,024	4,096
Large memory probes	16	1,536	16	6,144	2,048	4,096

These are the expanded campaign presets in `bench/campaign.py`; smoke tests and the early Megatron pilot used smaller overrides. Read each result's configuration before comparing. The conceptual attention matrix is $[B, H, T, T]$, but fused scaled-dot-product attention need not materialize that entire matrix. Doubling sequence length therefore does not justify assuming every measured memory category quadruples.

DDP replicates this architecture. FSDP/ZeRO change storage and communication of model state, without replacing the logical layers. Megatron tensor parallelism partitions computation inside transformer layers; it uses a separate implementation and requires its own equivalence checks. None of these synthetic transformer runs turns TinyGPT into the PS-Env policy.

11. Demonstrations and the gameplay learning experiments

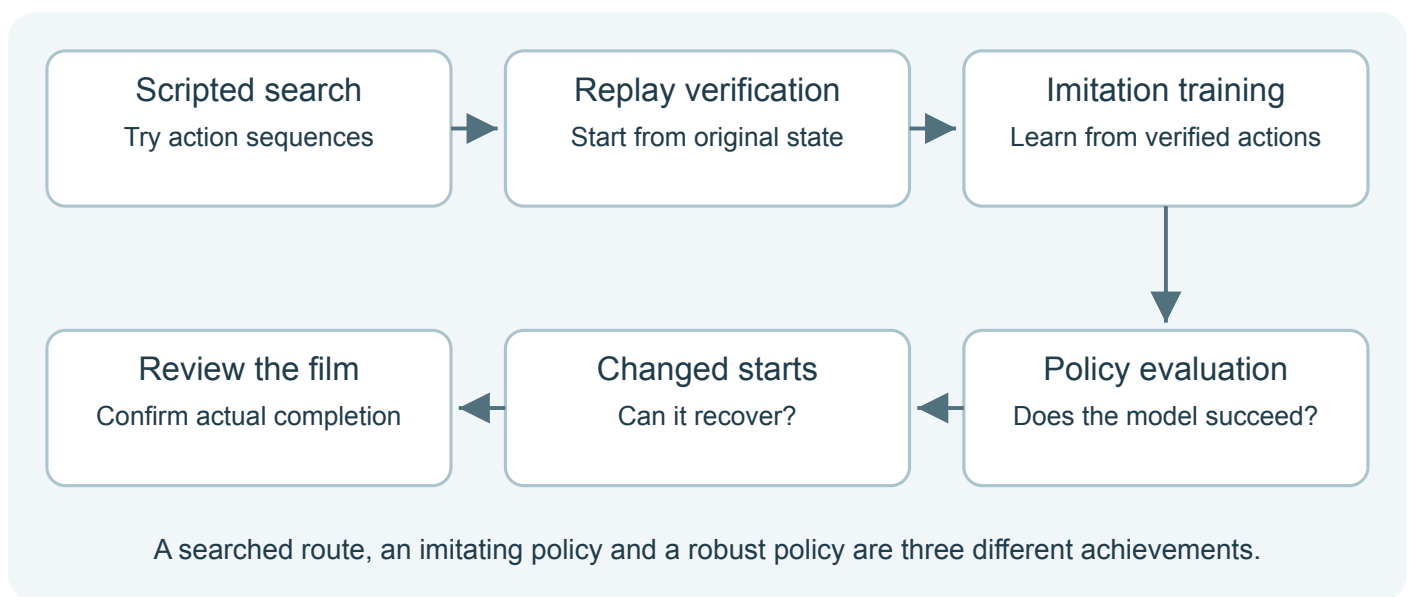


Figure 6. A successful scripted route is an input to learning; it is not itself a learned-policy win.

Our demonstrations were produced by **scripted action-sequence search**, not by claiming a human expert had played them. Search uses emulator save states to try alternatives, rejects undesirable branches and keeps candidate paths according to heuristics such as survival or spatial coverage. A beam of 12 retains up to 12 candidate branches at each search layer; it does not mean the search tested only 12 sequences overall.

This privileged save-state search is different from a deployed policy acting once through the game. Its output is a candidate action trace. We bind that trace to a starting-state hash and replay it through the actual environment. A warp transition alone is insufficient to claim completion.

Four names that describe different parts of training

These are overlapping descriptions, not four mutually exclusive neural-network architectures. The same image-to-action network can be behavior-cloned, trained with reinforcement learning, and practiced using a reverse curriculum.

Term	What it describes	Training signal / data	What we did in PS-Env	What it does not establish
Demonstration-trained	A broad description of a model's training history	Recorded teacher behavior; the exact objective must be specified	Replayed searched button sequences into observation/action examples; some policies then continued with RL and demo replay	It does not mean a human taught it, that the demonstration finished the level, or that the policy is robust

Term	What it describes	Training signal / data	What we did in PS-Env	What it does not establish
Imitation learning	A family of methods that learn behavior from demonstrations	Examples of behavior; methods differ in how they use them	Our supervised imitation stage used behavioral cloning	It is not the name of a particular layer layout; not every imitation method is BC
Behavioral cloning (BC)	A specific imitation-learning method	Supervised prediction of the teacher's action from its observation	600 updates in the audited learning trials; cross-entropy over discrete action scores	Matching recorded actions does not teach recovery from every unfamiliar state
Reverse curriculum	A schedule for choosing training start states	Practice near a goal, then move starts backward as success improves; RL can use rewards without an action-copying loss	Historical Toad Village training used this; newer section drills used verified intermediate states and local section goals	Finishing from near the exit or completing a section is not a full-start level win

BC is a subset of imitation learning. “Demonstration-trained” is a useful provenance label, but name the objective too: **BC only**, **BC followed by RL**, or **RL with continued demonstration replay**. Reverse curriculum can be combined with any suitable learning objective; it does not require copying teacher actions. Our section-practice implementation is related to reverse curriculum but a local section goal is not always the true level exit.

References: [behavioral cloning documentation](#), [imitation-learning methods](#), and [Salimans and Chen's demonstration-based reverse-start method](#). Implementation: `tools/learning_trial.py`, `agent/curriculum.py`, and `tools/train_ride_sections.py`.

```
# Our BC objective: demonstrated actions are class labels.
action_scores = policy(demo_images)          # [B, A]
loss = torch.nn.functional.cross_entropy(action_scores, demo_actions) # scalar; labels [B]
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

The scores are used as classification logits during BC and as Q estimates during subsequent DQN learning. BC alone does not calibrate them as expected future rewards.

Training recipe $\neq$ network architecture $\neq$ GPU count

BC describes a loss, reverse curriculum describes resets, and 1/2/4 GPUs describe execution. Changing any of these can leave the neural-network shapes exactly the same.

Comparison 1: what each term commits us to

The important distinction is **data source versus method family versus learning objective versus practice schedule**. A label such as “demonstration-trained model” leaves several implementation choices unspecified. The following tables make those choices explicit; “depends on the method” is a real distinction, not a missing definition.

Question	Demonstration-trained	Imitation learning	Behavioral cloning (BC)	Reverse curriculum
What kind of label is it?	Description of training history / data provenance	Family of learning methods	Specific supervised learning objective	Schedule for choosing episode starts
Main question it answers	Where did some of the learning data come from?	Are we learning behavior from examples?	How do we turn demonstrated actions into weight updates?	Which part of the task should the learner practice next?
Core idea	Use recorded behavior to help train a policy	Learn useful behavior from demonstrations	Predict the teacher's action from its observation	Learn easier goal-adjacent starts, then move farther from the goal
Does the name specify a network?	No	No	No; it needs a suitable observation-to-action predictor	No; it can reuse the same policy
Does the name specify a loss?	No: BC, RL and combinations are possible	No single loss for the whole family	Yes in our discrete-action implementation: cross-entropy	No; our curriculum policies learn using RL and optional demo losses
Required data in our implementation	Replayed traces; richer data when continuing RL	Observation/action pairs for our BC stage	Stacked images and matching action IDs	Ordered emulator snapshots and an appropriate success signal

Question	Demonstration-trained	Imitation learning	Behavioral cloning (BC)	Reverse curriculum
Must demonstrations be human?	No; ours were generated by search	No; the demonstrator can be a controller, search or another policy	No; labels can come from any accepted teacher	No; snapshots can come from a searched route
Must the demonstration finish the task?	No; ours include partial routes	Not universally, but data scope limits what it teaches	No; it can fit a partial route without learning the missing ending	A true exit-based curriculum needs useful starts toward the exit; section curricula can instead target local goals
Are reward values required?	Depends on the objective	Depends on the particular method	Not for the supervised BC loss	Our RL version needs reward feedback; the schedule also needs a success criterion
Are teacher action labels required?	Depends on how demonstrations are used	Depends on the method; not every method consumes action labels	Yes for our ordinary BC implementation	No for the start-state schedule itself

Comparison 2: what happens during training and gameplay

Question	Demonstration-trained	Imitation learning	Behavioral cloning (BC)	Reverse curriculum
Does training need new gameplay?	Not necessarily; BC-only can use stored data, continued RL collects more	Depends on the method	Our initial BC stage uses stored examples, with no new gameplay in its update loop	Yes in our implementation: play from the selected starts
What actually changes?	Whatever the chosen method trains; the label alone does not specify it	Usually policy parameters during training; details vary	Network weights change to favor demonstrated action choices	The reset distribution changes as the frontier moves; the chosen learner also updates weights
Where do episodes start?	Unspecified by the label	Unspecified by the family	The offline BC update loop has no episode reset; evaluation still needs a defined start	Near the current frontier, with a configured share of full-start attempts

Question	Demonstration-trained	Imitation learning	Behavioral cloning (BC)	Reverse curriculum
What causes advancement ?	No universal advancement rule	No universal advancement rule	More optimizer updates; no automatic "move to an earlier state" rule	A window of successful frontier attempts triggers an earlier start
How does it handle exploration?	Depends on whether RL follows the demonstrations	Depends on the method	No action exploration is needed to fit the fixed dataset	Shortens the difficult discovery problem by placing the learner closer to a reachable goal
What is fed into our policy at deployment?	Four stacked images for our demonstration-trained DQN	Four stacked images in our implementation	Four stacked images; no teacher action labels	Four stacked images; full-start evaluation does not restore helpful intermediate states
Are demonstrations supplied during inference?	Not for our policies; the label alone does not promise context conditioning	Method-dependent; our BC policies do not take demonstration context	No in our implementation: their influence is stored in the weights	No action trace is supplied to our evaluated policy
Are weights updated during evaluation?	No in our evaluations	No in our evaluations	No	No; evaluating a curriculum-trained policy is still a fixed-policy test
Can these be combined?	Yes: BC followed by RL still has demonstration-based training history	Yes: an imitation method can initialize a later RL learner	Yes: BC can precede RL, demo replay and curriculum practice	Yes: practice scheduling is separate from the policy loss
Does adding GPUs change the meaning?	No	No	No; distributed BC would average gradients from supervised batches	No; distributed workers would also need a deliberate start-sampling and curriculum-coordination rule

Comparison 3: benefits, limits and convincing evidence

Question	Demonstration-trained	Imitation learning	Behavioral cloning (BC)	Reverse curriculum
Main benefit	Useful prior experience can reduce blind discovery	Behavior examples supply information a sparse reward may not provide	A direct way to initialize useful action choices from verified examples	Makes long tasks easier to approach by practicing shorter remaining segments
Where can cost hide?	Collecting, searching, replaying and validating the demonstrations	Teacher access, data collection and any interactive training loop	Preparing images/action labels and covering enough situations; count this as well as optimizer time	Generating/restoring reliable snapshots, measuring success and replaying solved sections
Characteristic failure	"Trained on a winner" is mistaken for "the policy wins reliably"	The learned behavior fails outside the example distribution	Small action errors lead to unfamiliar observations, where further errors accumulate	Section success fails to compose into a full route, or earlier skills are forgotten
What if the examples are incomplete?	The model may learn only the demonstrated portion	Coverage remains a limitation	There is no supervised action target for an unrecorded ending	A local section curriculum may improve local control without ever teaching the true finish
Can it surpass the teacher?	Possibly with appropriate subsequent learning; not guaranteed	Method-dependent; not guaranteed	It can generalize beyond examples, but ordinary BC provides no explicit reward-based objective to discover a better route	It can learn different actions with RL; snapshots do not require copying the original actions
What does low training loss prove?	Nothing without knowing the loss	Fit to that method's objective, not automatic task success	Better fit to the sampled action labels; not reliable full-start completion	The reset schedule has no neural-network loss of its own; the learner's loss must be interpreted separately
What should be evaluated?	The actual saved policy, separately from the teacher trace	Policy success and recovery on conditions outside training	Held-out prediction plus closed-loop gameplay, especially deviations from the route	Full-start completion as well as frontier/section success and retention of solved sections

Question	Demonstration-trained	Imitation learning	Behavioral cloning (BC)	Reverse curriculum
What is a misleading metric?	Number of successful teacher traces presented as policy wins	Training objective presented as success rate	Action-label accuracy alone presented as a robust playing policy	Many near-exit wins presented as many full-level wins
What did our retained snapshot show?	The three audited routes included one complete route and two partial routes	BC was the implemented supervised imitation method	Orient Express completed the demonstrated start but failed timing-shift checks; later selected policies retained the initial BC checkpoint	Historical Toad training used reverse starts; newer section wins were recorded separately from full-start wins
What must a result label include?	Teacher source, data scope, objective and subsequent RL	Specific method, teacher/data source and evaluation protocol	Update budget, input/controller settings and evaluated checkpoint	Start-state construction, frontier rule, full-start share and definition of success

The “retained snapshot” row refers to the evidence dated in this guide. It is not a live claim about the newer level 1 campaign. That campaign uses the historical 13-action ordinary DQN and full-start training, without reconstructing its earlier curriculum or success buffer.

The concrete recipes we compared in code

These rows are actual experimental combinations. The four broad terms above are not four experiment arms.

Recipe	Initialization	What data changes the weights afterward?	Episode starts	Why include it?
Scratch RL (<i>scratch</i>)	Random network weights	New gameplay transitions, sampled through replay; TD loss	Full starts, with configured timing changes where enabled	Measure what RL discovers without demonstration initialization
BC only (the saved <i>initial</i> checkpoint)	600 supervised updates in the audited campaign	No subsequent updates when evaluating that checkpoint	Offline BC has no reset loop; evaluate from defined full starts	Measure what imitation alone accomplished

Recipe	Initialization	What data changes the weights afterward?	Episode starts	Why include it?
BC → RL (warm)	The BC-initialized policy	New gameplay replay; demonstrations are not continually added to its RL batches	Full starts, with configured timing changes where enabled	Test whether RL improves or damages the initialized behavior
BC → RL + demo replay (replay)	The BC-initialized policy	New transitions plus retained demo transitions; TD loss and a demo-action margin term	Full starts, with configured timing changes where enabled	Test whether continued teacher examples help retain useful behavior
BC → RL + demo replay + section practice (curriculum)	The BC-initialized policy	The same mixture, while practicing verified route sections	Intermediate snapshots plus a share of full starts	Test whether shorter practice segments help learn control; count section success separately

In `tools/learning_trial.py`, a normal per-task replay batch for a task with enough demonstration data uses **16 new-replay samples + 16 demo samples**. The combined batch pays the TD objective; flagged demo samples also pay the large-margin action objective. BC pretraining instead uses **32 demonstration examples per task** and cross-entropy. These are separate losses and training phases. The configured demo margin is 0.8; it is not an 80% demo sampling share.

For the section arm in this tool, the curriculum uses a five-result window, an 80% success threshold and a 50% full-start share. Historical Toad Village recipes have different settings; do not transfer these numbers to every run labelled "curriculum." A current frontier can advance on section completion in this arm, so its success signal differs from an exit-based curriculum.

Worked example: an obstacle near the end of a level

Imagine a verified route has 100 decisions, and decision 90 is just before a difficult jump. These numbers are illustrative, not a new measured PS-Env result.

Approach	What the learner experiences	What one successful result would mean
BC only	Train on the 100 recorded image/action pairs: at the observation before the jump, predict the demonstrated jump action	The policy can imitate that route if closed-loop evaluation confirms it; action accuracy alone is insufficient
RL from scratch	Start at decision 0's game state and discover useful behavior through rewards	A finish establishes that evaluated attempt succeeded without demo initialization

Approach	What the learner experiences	What one successful result would mean
Reverse curriculum with RL	Restore the state near decision 90, learn the remaining segment, then move the start earlier after sufficient success	A win from decision 90 proves the suffix was completed; it does not count as playing decisions 0–89
BC + reverse curriculum	Initialize by copying the route, then practice progressively earlier starts with RL	Combines a useful initial policy with easier practice; still requires full-start and recovery evaluations
BC + recovery examples	Add examples showing how to act after landing off-center or missing a timing cue	Tests whether broader teacher coverage improves recovery; this is a proposed improvement, not a claimed result

A minimal sketch of the difference is:

```
# Behavioral cloning: observations and teacher actions define a supervised loss.
scores = policy(demo_images)                # [B, A]
bc_loss = cross_entropy(scores, teacher_actions)
update_weights(bc_loss)

# Reverse curriculum: choose where an RL episode begins.
start = curriculum.pick()                   # None = full start; index = snapshot
observation = env.reset(start=start)
trajectory = play_episode(policy, observation)
update_from_rl_replay(trajectory)
curriculum.report(start, trajectory.success)
```

This is pseudocode: `play_episode`, `update_weights`, `trajectory.success` and `update_from_rl_replay` stand for the real implementations. It demonstrates why BC and reverse curriculum can coexist: one determines a loss, the other determines a reset. For section practice, success must be explicitly identified as section success rather than silently treated as level completion.

How to describe a model precisely in the presentation

Vague label	More informative description
"A demonstration-trained model"	"A seven-action dueling DQN initialized with 600 BC updates on a verified Orient Express completion trace; evaluated separately from the trace"
"The imitation model passed"	"The saved BC policy completed the demonstrated full-start condition; timing-shift conditions failed at the retained snapshot"

Vague label	More informative description
"The curriculum model won many times"	"The policy completed a specified number of frontier/section attempts; full-start outcomes are reported separately"
"The four-GPU model is better"	"Four replicas execute one synchronized learner faster; policy quality is determined by the common evaluation protocol"

The audited demonstration set used in the current experiments contains:

Level	Recorded teacher decisions	Scope
Under Pressure	480	Partial route
Orient Express	849	Completion route
Makin Waves	1,050	Partial route

We convert replayed observations and actions into imitation data. Behavior cloning trains the network to reproduce those action choices. Continued demonstration replay mixes demonstration transitions with newly collected experience and adds a demonstration action-margin objective. Section practice starts from verified intermediate snapshots; a section success is not a full-start win.

We compared scratch, demonstration initialization, initialization plus continued replay, and replay plus section practice. In an early snapshot all 11 finished demonstration-based trials retained their pre-RL checkpoint under our completion/milestone selector. This means no later checkpoint beat that selection score; ties do not prove every later policy was identical or worse in every respect.

The failure under tiny timing shifts is an important generalization result. Learning the demonstrated sequence of situations is easier than learning robust recovery when the game deviates from that sequence.

12. Reward design: what the results actually tell us

Reward is a proxy for the goal

The goal is completing the level robustly. A reward can increase because the agent explores more cells or collects items, while completion stays at zero. Always read reward alongside meaningful progress and outcomes.

The early level-one recipe included a particular direction/progress assumption and large death penalties. Applying it unchanged to swimming or curved routes could produce inappropriate guidance. Later presets disabled direction-specific shaping and introduced capped exploration bonuses. Exploration helped provide feedback, but exploring new cells does not necessarily mean progressing toward an exit.

One clear mismatch under the previous reward setup was Under Pressure: a policy reaching 8/8 reference landmarks earned about -3.5, while a policy reaching only 1/8 and dying earned about +6.2. Makin Waves could earn more than 200 with little reference-route progress because exploration contributed heavily.

The overnight reward package reduced exploration, reduced the per-step cost, strengthened life/game-over penalties, and added +5 for reaching each ordered verified landmark once per episode. Completion detection remained unchanged. Demonstration rewards were recomputed through the same calculator, with every resulting observation checked against the stored data.

This changes several coefficients together. It tests a reward package, not the isolated effect of one coefficient. Within each new paired comparison the other settings match; comparisons against older campaigns also involve changes to learning rate, exploration and starting-state variation.

At this snapshot, greedy checkpoint evaluation showed:

Level	At 100k decisions	Later checkpoint	Interpretation
Under Pressure	Reward 26.1; 5 landmarks	Reward 41.6 at 400k; 8 landmarks	Recovers the known partial route, no completion
Makin Waves	Reward 92.1; 17 landmarks	Reward 86.2 at 400k; 17 landmarks	No sustained return gain or completion
Orient Express	Reward 568.1; completes	Same return and completion at 500k	Preserves demonstrated success; shifts still fail

A route reaching 8/8 or 17/17 is at the end of our **reference**, not necessarily the level. That metric then saturates and cannot distinguish progress beyond the recorded route. Likewise, a level without calibrated landmarks should show “not calibrated,” not a fabricated zero-percent completion metric.

More experience can help, but it can also reinforce a poor objective or fail to discover a narrow successful trajectory. We have evidence that reward ranking is imperfect and robustness is weak. We have not proved that reward is the sole cause, or that the current revision solves it.

13. Infrastructure is part of the experiment

A checkpoint is only as complete as its contents

Saving model weights and optimizer state does not preserve replay buffers, the emulator state or every random-number generator. Our distributed warm start is not exact recovery.

The stack includes Terraform-managed workers, a pinned Python/CUDA environment, native emulator builds, private game assets, restricted SSH access, systemd campaign services, checkpointing, a private S3 artifact destination, a separate collector and the hosted console.

On the GPU workers, presigned upload URLs grant access to specific artifact objects rather than placing broad AWS credentials inside training jobs. The console collects metrics and selected models independently of the operator's laptop. Secrets and private assets are not exposed through the metrics API.

We publish checkpoints by atomic rename so readers do not mistake a partially written file for a usable model. Campaign summaries distinguish queued, running, completed, failed, interrupted and unrun work. Missing evaluation is not zero completions.

A checkpoint containing network and optimizer state still may not be an exact restart. Our PS-Env distributed checkpoint does not preserve replay contents and active emulator state. Loading it starts from learned parameters with new experience collection; it is a warm start. Exact recovery requires identifying and restoring every relevant source of state, including randomness and sampling position.

The overnight controller has a finite runtime, per-job limits, periodic uploads and a separate host shutdown timer. Instance-initiated shutdown terminates the GPU instance. At the snapshot it was still running, with hard termination scheduled for 09:01 Qatar. That is a schedule, not evidence that termination has already occurred.

14. Failures and the lessons they gave us

Failure or misleading result	What changed / what we learned
More GPUs slowed the tiny two-node transformer task	Measure communication relative to useful work; adding devices is not automatically beneficial
Optional dependencies upgraded the CUDA/PyTorch stack unexpectedly	Pin compatible versions and retain runtime/package records

Failure or misleading result	What changed / what we learned
Small numerical discrepancies upset a CPU weight comparison	Investigate the computation rather than silently widening tolerances
Service inactivity during restart was mistaken for completion	Require completion evidence or sustained failure, not one transient state
Evaluation lengths varied across ranks	Close the training process group and use atomic evaluation-completion files rather than holding a collective open through long evaluations
Local disk exhaustion broke collection and cleanup logging	Cleanup must remain possible even when logging or artifact collection fails; original evidence was recovered and checkpoints verified
The first real-game scaling pair had unequal updates	Add a fixed global update schedule and validate actual counts
Shared daytime trials hit a 90-minute limit	Budget for all levels and evaluation/film time; preserve partial checkpoints and mark interruptions honestly
All six overnight searches failed at CLI validation	Correct optional-prefix validation, test the real entry point and relaunch; these were not unsuccessful game searches
Three transformer recovery trials failed	Report recovery separately; a training run finishing does not prove interruption recovery works
Reward increased without a completion	Inspect reward components, route progress and films rather than optimizing the dashboard number
Presentation lagged the live results	Treat static slides as versioned evidence snapshots and refresh them deliberately

The three expanded recovery failures were DDP with four GPUs and ZeRO-3 with two/four GPUs. Seven other recovery trials completed. It would be inaccurate to summarize this as universally successful checkpoint recovery.

15. How to evaluate the next experiment

Compare the same amount of work

Keep the global batch, experience budget and optimizer-update count fixed when measuring scaling. Giving each GPU the original full batch would increase the total batch and change the experiment.

Start with one question and the evidence that could change your mind.

For **systems scaling**, control model, global batch, total decisions/tokens, update count, precision, replay capacity, checkpoint frequency and timing scope. Repeat seeds or measurements, reverse run order where useful, and preserve failures. Report speedup and efficiency alongside quality.

For **learning quality**, compare against both scratch and demonstration-only baselines. Keep rewards fixed when comparing training methods, or keep training settings fixed when comparing rewards. Evaluate unseen starting conditions and review films. A deterministic repeat from the same state is not a new independent robustness trial.

For **cost**, distinguish active-GPU efficiency from the cloud bill. These one-/two-/four-GPU PS-Env measurements all ran on the same four-GPU instance: using one GPU did not reduce that host's hourly price. A 3.32× speedup is not automatically a 3.32× bill reduction. End-to-end cost includes idle time, setup, evaluation, storage and artifact transfer. This document is not an audited AWS invoice.

For **memory**, identify whether parameters, optimizer state, activations or temporary buffers are limiting the workload before choosing sharding, checkpointing or a larger accelerator.

16. Exercises using our own project

1. **Calculate global batch.** Four GPUs each sample four transitions per level across five levels. How many samples contribute to one update? Answer: 80.
2. **Calculate update count.** Start at 50,000 inclusive, stop at 200,000 inclusive, update every 20. Answer: 7,501. If you omit "inclusive," you miss one update.
3. **Calculate efficiency.** Four GPUs give 3.32× speedup. Answer: $3.32/4=83\%$ scaling efficiency, not 83% GPU utilization.
4. **Spot a misleading claim.** "Four GPUs got 0/20, one GPU got 0/20, so the models are equally good." Unsupported: this test only failed to observe a completion in either; other behavior and untested conditions may differ.
5. **Inspect synchronization.** Read `average_gradients` and find where summing becomes averaging. Predict what forgetting the division would do to gradient magnitude.
6. **Inspect restart semantics.** List what `core.save` preserves and what the emulator/replay loop keeps only in memory. Explain why loading weights is not an exact continuation.
7. **Design a reward ablation.** Change only the exploration coefficient while holding data, starts, seeds and budgets fixed. Decide beforehand which gameplay metrics would count as improvement.
8. **Design a robustness check.** Choose starting perturbations excluded from training. Explain why replaying the exact winning start twenty times is inadequate.

Do not begin by launching every framework at once. First reproduce a small correct single-device loop, then identical-data gradient checks, then two-device execution, then a controlled scaling measurement. Memory-sharding and tensor-parallel experiments become easier to understand once that baseline is solid.

17. Code and evidence map

Paths in this table are relative to the corresponding repository. Generated run files are generally ignored by Git; retain collected evidence separately.

PS-Env file	Read it to understand
<code>agent/model.py</code>	Convolutional models, shared features and level-specific heads
<code>agent/dqn.py</code>	Action selection, loss, optimizer and checkpoint state
<code>agent/replay.py</code>	Replay samples and n-step transitions
<code>agent/distributed.py</code>	Rank setup, local collection, gradient averaging, scheduling and evaluation
<code>agent/progress.py</code>	Ordered landmarks and diagnostic metrics
<code>agent/route_reward.py</code>	Experimental reward package and route bonus
<code>tools/learning_trial.py</code>	BC, replay, reward relabelling, checkpoint selection and evaluations
<code>tools/search_level_demo.py</code>	Candidate action-sequence search and optional verified prefix
<code>tools/replay_ride.py</code>	Independent trace replay and completion evidence
<code>deploy/distributed/overnight.py</code>	Scaling trials followed by longer gameplay jobs
<code>deploy/distributed/overnight_search.py</code>	Additional bounded CPU search
<code>tools/collect_learning.py</code>	Private artifact collection onto the console
<code>console/learning.py</code>	Read-only experiment API and artifact allowlist
<code>docs/OVERNIGHT_EXPERIMENTS.md</code>	Exact overnight protocol, limits and operational notes

In the separate `distributed-training-benchmarks` repository, begin with `README.md`, `evidence/RESULTS.md`, `bench/`, and `presentation/evidence/expanded-*.json`. The phase summaries

support the expanded outcome counts and the per-trial memory files support the tested capacity boundaries. The PS-Env six-run record is `overnight-distributed.json` in the collected learning artifacts.

The source of project-specific numbers is our retained experiment evidence, not the framework documentation. The linked primary documentation explains the techniques; it does not validate our particular results.

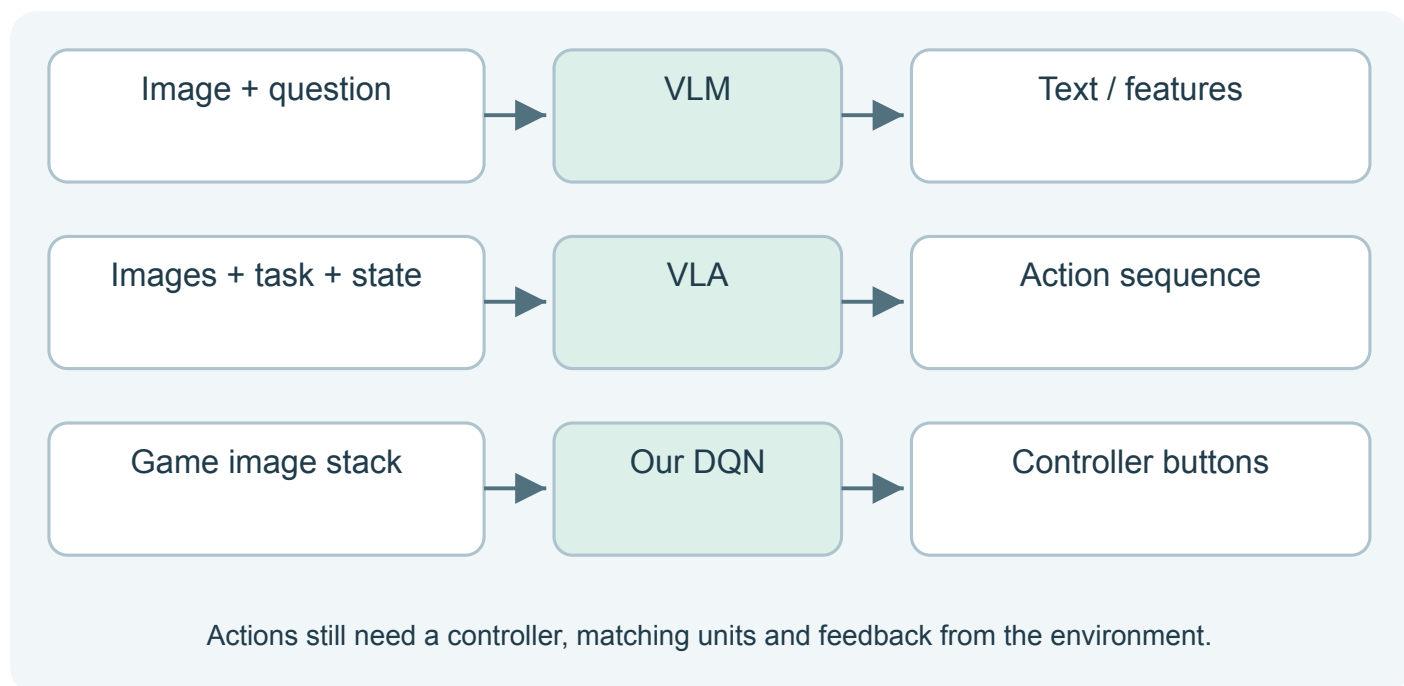
A defensible description of the work: "I built and evaluated a multi-level reinforcement-learning system, implemented synchronous data-parallel training with PyTorch/NCCL, measured 3.32× end-to-end throughput on four A10G GPUs at controlled update counts, and compared replicated, sharded and tensor-parallel training in a separate transformer benchmark. I also investigated memory limits, recovery failures, reward misalignment and demonstration generalization. Faster execution is established; robust multi-level gameplay remains unfinished."

18. Where open robotics models fit

Research extension checked on 14 September 2026. None of the external models in this chapter was trained or deployed by our PS-Env experiments. These are examples for understanding the field, not a ranking of every available release.

A **vision-language model (VLM)** combines visual input with language. It might identify objects, answer a question about an image, or describe a goal. Understanding an instruction is different from producing correctly scaled robot controls.

A **vision-language-action model (VLA)** conditions action prediction on visual information and language, often with robot state as an additional input. Its output may be discrete action tokens or a continuous sequence of movements. A VLA is a kind of policy. A robot policy does not have to be a VLA: our DQN has no language input, and geometric demonstration-conditioned policies need not contain an LLM.



Model roles are distinct from training methods. A VLA may use imitation training, while distributed training determines how its parameter updates are computed across devices.

Category	Typical inputs and outputs	Where it fits
Vision encoder	Image → visual features	Supplies representations to another model; not a complete controller
VLM	Images + text → language or multimodal features	Perception, interpretation, planning, or a backbone for an action model
VLA	Images + instruction + optional robot state → actions	A language-conditioned robot policy
Diffusion / flow policy	Observations and conditioning → action sequence	Generates continuous actions through learned refinement; language is optional
Demonstration-conditioned policy	Current observation + example trajectories → action	Uses examples as task context if trained/designed to do so
Low-level controller	Target pose, joint values or velocities → actuator commands	Executes the policy output under the robot's conventions and constraints
Simulator / environment	Action → next observation and outcome	Supplies interaction data and evaluation; PS-Env fills this role for the game

“Diffusion” here usually describes generating actions, not generating pictures. **Action chunking** means predicting several future actions together. The controller can execute part of the chunk, observe again, and replan. Longer chunks can amortize inference cost but may delay reaction to unexpected events. This is a control-design tradeoff, not the same as increasing the training batch.

Representative open projects

Project	What it is	Why it is useful to understand
Open VLA	A 7B-parameter VLA; the original model builds on a pretrained VLM and learns robot action tokens	Shows how visual-language representations can be adapted to action prediction. The original release reports 970k robot trajectories. Paper , code
Smol VLA	A compact 450M-parameter VLA with an action expert, released through the LeRobot ecosystem	A smaller reference for studying action prediction and execution latency. Authors' introduction , model card
π_0 / open pi	Physical Intelligence's open robotics model package; π_0 uses flow-based action generation, while π_0 -FAST uses autoregressive action tokenization	Demonstrates different action-generation designs within the VLA family. The repository also includes $\pi_{0.5}$. Official repository
Octo	A generalist transformer diffusion policy supporting language or goal-image conditioning; initial Small/Base versions are 27M/93M parameters	Demonstrates multi-robot data, flexible observations and adaptation to action spaces without requiring a large autoregressive VLM as the policy backbone. Project , code
Instant Policy	Vosylus and Johns's graph-diffusion policy conditioned on demonstrations	A direct reference for in-context imitation rather than a standard language-first VLA. ICLR paper , code

These sizes describe the cited releases, not a claim about every newer checkpoint. **LeRobot** is a robotics software/data ecosystem, rather than the name of a single policy. **Open X-Embodiment** denotes a robot-dataset collection used by projects such as OpenVLA and Octo; it is not a model or an optimizer.

Open code, open weights and open training data are separate

Check the specific release. OpenVLA's code is MIT-licensed, while its original Llama-2-derived weights carry inherited model-license conditions. An available checkpoint does not imply that every training trajectory is public or that every component shares one license.

The OpenVLA licensing distinction is documented in its [official README](#). For other projects, use their model cards, repository licenses and upstream model terms rather than assuming the word “open” resolves every component.

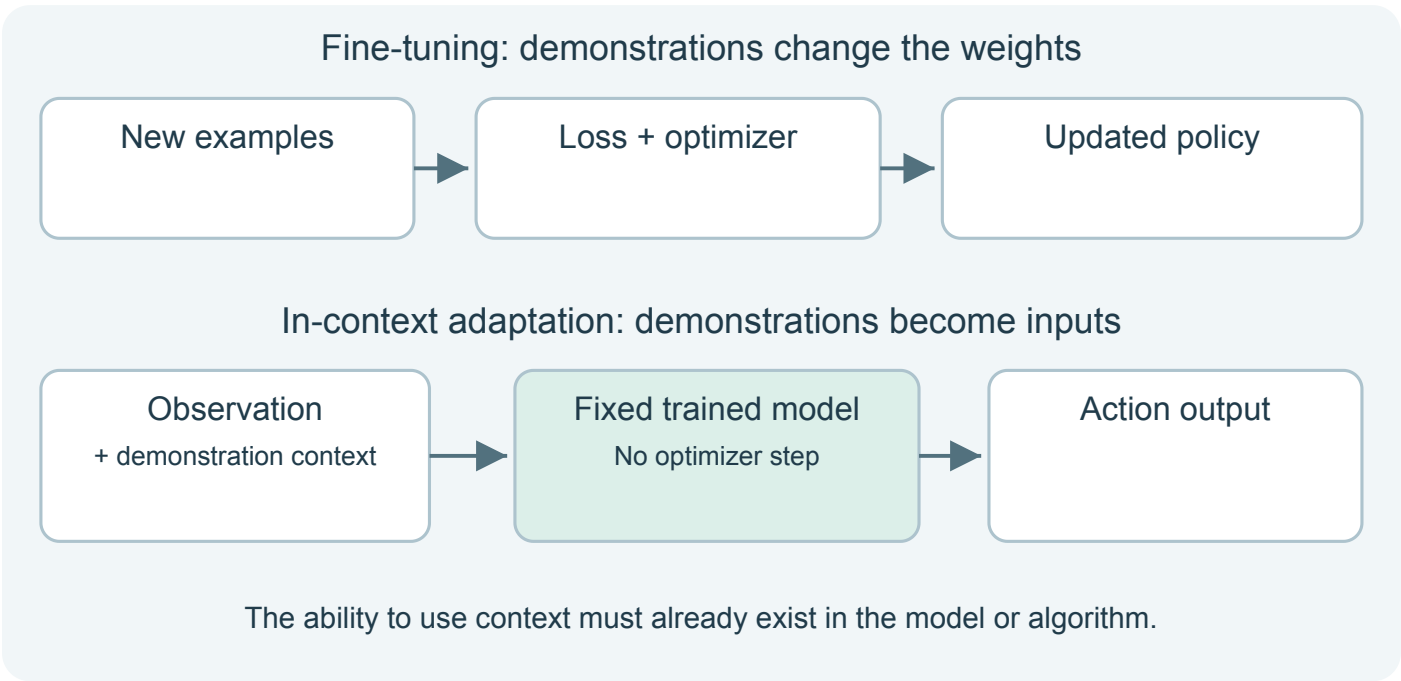
Why we cannot just load one into Crash Bandicoot

A robot model may expect multiple RGB cameras, joint positions and continuous end-effector actions. PS-Env provides game images and discrete controller actions. The visual distribution, action meaning, timing and training objective differ.

Adapting a representation is plausible; directly sending robot-action outputs to a game controller is not a meaningful integration. We would need a defined observation adapter, an action head or action mapping, task data, and a held-out evaluation. A pretrained model's success on robot manipulation does not establish competence in our game.

19. In-context learning: change the examples, not the weights

Fine-tuning uses new task data to update model parameters. **In-context learning (ICL)** changes behavior by providing examples in the model's input while keeping its parameters fixed during that adaptation. In robotics, the examples may be demonstrations containing observations and actions, rather than text question-and-answer pairs.



A model can require substantial pretraining to acquire this ability. "No updates for the new task" does not mean "no training ever."

Approach	What happens with the new demonstrations?	Weights change for the new task?
Our behavior cloning	Demonstration observations/actions become supervised training samples	Yes

Approach	What happens with the new demonstrations?	Weights change for the new task?
Our continued demonstration replay	Demonstration and online transitions contribute to optimizer updates	Yes
Fine-tuning with adapters / LoRA	A smaller trainable parameter set is optimized	Yes, even if the backbone stays frozen
In-context imitation	Demonstrations are part of inference-time conditioning	No, in the fixed-weight adaptation setting
Simple retrieval/replay	Retrieve a relevant recorded behavior, possibly align it, and replay	Not necessarily; this is not automatically learned ICL

In notation, fine-tuning changes the parameters: $\theta \rightarrow \theta_{\text{task}}$, then predicts `policy(observation;  $\theta_{\text{task}}$ )`. In-context adaptation instead predicts `policy(observation, demonstrations;  $\theta$ )` with the same θ .

For a model to benefit, it must have an appropriate context interface and the learned ability or algorithmic structure to use it. Appending arbitrary demonstrations to an ordinary DQN does not create ICL. Similarly, an instruction-following VLA does not automatically support few-shot trajectory context just because it understands text.

One way to train that ability is to construct task episodes: provide some demonstrations as context and ask the model to predict actions in another trajectory from the same task. Test on held-out tasks afterward. The context examples and prediction targets must be separated, and evaluation must not feed the answers it is supposed to measure back into the context.

What changes for distributed training?

Distributed pretraining and fixed-weight in-context adaptation can coexist:

1. During pretraining, workers receive different context/query task episodes and average gradients to train a common model.
2. At deployment, the trained model receives a new context and predicts actions without an optimizer update.
3. Serving multiple inference requests or robot streams in parallel is not the same operation as distributed training.

Demonstration context consumes input-processing time and memory. Longer context can increase attention or graph-processing cost. Reusing cached context representations may help when the architecture supports it, but we must measure latency and prediction quality. A larger context is not an unlimited free substitute for training.

Our PS-Env demonstration policies are not in-context policies

We used demonstrations to change network weights. During gameplay the DQN receives image stacks, not a new demonstration context. Calling it “demonstration-trained” is accurate; claiming Instant Policy-style task adaptation would require a different experiment.

20. Connecting this to Edward Johns and Vitalis Vosylius

Edward Johns's lab describes research in robot learning, including imitation and efficient adaptation. For this guide, the strongest connections are demonstrated capabilities in the papers below. I did **not** establish a dedicated DDP/FSDP/NCCL-method publication by him in the material reviewed. Do not turn “scalable robot learning” into an unsupported claim that he developed a particular distributed-training framework. These academic sources also do not establish Machenta's private model architecture. [Lab research index](#), [biography](#).

Publication	Research idea	Connection to our experiments—our interpretation
Instant Policy , Vitalis Vosylius and Edward Johns; ICLR 2025	Graph-diffusion in-context imitation from one or two demonstrations	Compare task adaptation without new updates against our BC/replay adaptation. Paper
Keypoint Action Tokens Enable In-Context Imitation Learning in Robotics , Norman Di Palo and Edward Johns; RSS 2024	Represents observations/actions using 3D keypoints for LLM-based in-context imitation	Representation can make examples useful; natural-language prompting is not the only way to condition a policy. Project
Learning a Thousand Tasks in a Day , Kamil Dreczkowski, Pietro Vitiello, Vitalis Vosylius and Edward Johns; Science Robotics 2025	Separates alignment and interaction, studying efficient teaching across many tasks	Our policies' failure after small deviations motivates measuring recovery and alignment, rather than only replaying a favorable start. Project
MILES: Making Imitation Learning Easy with Self-Supervision , Georgios Papagiannis and Edward Johns; CoRL 2024	Collects self-supervised augmentation trajectories around a demonstration	We could test recovery examples around failures instead of repeating only the original successful trace. Project
Auto-λ: Disentangling Dynamic Task Relationships , Shikun Liu, Stephen James, Andrew J. Davison and Edward Johns; TMLR 2022	Learns task weights using a meta-learning formulation	Our five-level loss weighting is a testable choice; equal weighting is not proof of optimal transfer. Project

Publication	Research idea	Connection to our experiments—our interpretation
Prismer: A Vision-Language Model with Multi-Task Experts , Shikun Liu and collaborators including Edward Johns; TMLR 2024	Combines pretrained experts in a data-/parameter-efficient VLM	A direct VLM connection: useful representations and selected trainable components matter alongside raw hardware scale. Project
Back to Reality for Imitation Learning , Edward Johns; CoRL 2021 blue-sky oral track	Argues for evaluating human time efficiency in robot learning	Include setup, demonstration collection, resets and evaluation time when explaining what our speedup is worth. Essay/project

A concrete training detail from Instant Policy

The paper's Appendix E reports **2.5 million optimization steps, approximately five days on a single RTX 3080 Ti**, followed by a 50k-step learning-rate cooldown, using float16 and compilation. Its scaling study compares **69M, 117M and 178M** model variants and training durations. These are the authors' results, not our runs. Parallel generation of pseudo-demonstrations supplies training data; that is distinct from reporting a multi-GPU gradient-synchronization benchmark. [Published paper, Sections 4–5 and Appendix E](#).

The useful comparison is therefore between three kinds of scaling: generating more relevant data, training a larger or longer-trained model, and executing a fixed training workload faster. We measured the third directly; our gameplay results show why the first two and the structure of the task still matter.

What to emphasize in the conversation with Ed

A relevant explanation would be: “We measured a repeatable systems speedup, but the stronger behavior came from demonstrations, and small timing shifts exposed weak recovery. That made us separate throughput from adaptation quality and ask whether demonstration-conditioned representations or recovery-focused data would be more useful than simply increasing steps.”

Useful questions grounded in these publications include:

- When should we prefer retrieval/alignment, fine-tuning or in-context imitation for a new task?
- Which demonstration variations teach recovery rather than just repeat the nominal behavior?
- How should we measure total human time to a usable policy?
- In a shared policy, how do we detect one task harming another?
- When is generating better task/context episodes more valuable than accelerating gradient updates?

These are proposed discussion questions, not claims about Machenta's internal technology or Ed's personal preferences.

21. Small code examples and a practical next experiment

Example A: context can change an output without an optimizer

This executable example uses nearest-example attention over simple vectors. It isolates **conditioning**; it is not a learned robot policy, a VLA, or an implementation of Instant Policy. Its scalar outputs are illustrative values, not commands for hardware.

```
import torch

def action_from_context(query, demo_observations, demo_actions):
    distances = torch.cdist(query, demo_observations).square()
    attention = torch.softmax(-distances / 0.1, dim=-1)
    return attention @ demo_actions

observation = torch.tensor([[1.0, 0.0]])
examples = torch.tensor([[1.0, 0.0], [0.0, 1.0]])
context_a = torch.tensor([[0.2], [-0.2]])
context_b = torch.tensor([[0.8], [-0.2]])

with torch.inference_mode():
    print(action_from_context(observation, examples, context_a)) # ~0.2
    print(action_from_context(observation, examples, context_b)) # ~0.8
```

The observation and computation are unchanged; the demonstration input changes. A learned context-conditioned policy would learn a much richer interpretation of observations, task relationships and action geometry. This toy does not demonstrate that capability.

The runnable version, with output checks, is `tools/learning_examples/robotics_context.py`. It was executed locally. No external robotics checkpoint was downloaded to run it.

Example B: training an imitation model with DDP

This is a **training-loop sketch**, not a standalone launch script or a VLA-specific API. Assume a CUDA process group, one local device per rank, a policy on that device, equal valid sample counts per rank, batch tensors on that device, and a loader that partitions examples across workers. The policy here directly predicts normalized continuous actions.

```

import torch
from torch.nn.parallel import DistributedDataParallel as DDP
import torch.nn.functional as F

# policy and train_loader are application-specific; setup precedes this snippet.
ddp_policy = DDP(policy, device_ids=[local_rank])
optimizer = torch.optim.AdamW(ddp_policy.parameters(), lr=1e-5)

for batch in train_loader:
    optimizer.zero_grad(set_to_none=True)
    predicted_actions = ddp_policy(batch["observation"], batch["context"])
    loss = F.mse_loss(predicted_actions, batch["target_actions"])
    loss.backward() # DDP synchronizes gradients during backward.
    optimizer.step()

```

For discrete actions, a classification loss may be appropriate; diffusion or flow policies use their own noise/velocity prediction objectives. Mixed embodiments also need action-space handling, masks and normalization. This sketch shows where gradient sharing fits, not a universal robotics loss function. PyTorch documents the synchronization behavior in [DDP](#).

At adaptation time an ICL-capable model instead follows the shape `action = policy(observation, new_context)` inside inference mode, with **no** `backward()` or `optimizer.step`. `eval()` alone does not disable gradient tracking; it selects evaluation behavior for applicable layers.

Example C: an action output still needs the correct interpretation

OpenVLA's original interface includes `predict_action` and a dataset-specific `unnorm_key`. The following illustrates the inference call after loading the compatible processor/model and constructing its image-plus-instruction inputs:

```

# Interface pattern from the original OpenVLA README, not executed here.
# "bridge_orig" is meaningful for the matching dataset/robot convention.
with torch.inference_mode():
    action = vla.predict_action(
        **inputs, unnorm_key="bridge_orig", do_sample=False
    )

```

The normalization key is not cosmetic. Output coordinates, units, action horizon and robot conventions must match the intended environment. This is why a robot checkpoint does not plug directly into PS-

Env. Use the release's [official setup and inference example](#); this fragment intentionally does not claim a complete installation or execution.

A next experiment that connects the ideas

A useful proposed comparison would keep evaluation tasks fixed and compare demonstration-only BC, BC plus recovery examples, and a genuinely context-conditioned policy. Train the latter on separated context/query trajectories across many task variations; evaluate it on withheld variations without adapting weights.

Measure completion, failure recovery, number of demonstrations, adaptation time, inference latency and total operator time. Only afterward compare one-/two-/four-GPU **pretraining** throughput at matching global work. Otherwise we risk mixing a change in learning method with a change in execution speed again.

For PS-Env, a first step is building a richer collection of task variations and recovery examples. One successful fixed-start trace and two partial routes are insufficient evidence that a general in-context policy will learn robust adaptation. The diagrams and code explain a direction we could test; they do not upgrade the current DQN into a robotics foundation model.