Contents

1 Chapter 9: Cracking Pixel Push -- Visual HER on FetchPush

Week 9 Goal: Solve FetchPush from raw pixels with sparse rewards, reaching 90%+ success rate. Along the way, discover why pixel-based manipulation is fundamentally harder than pixel-based reaching, and learn to debug visual RL training failures systematically.

1.1 Bridge: From State-Based Push to Visual Push

In Chapter 4, we combined SAC with HER and solved FetchPush from state at 89% success in 2M steps. The algorithm works. The question is: **can we do it from pixels?**

Chapter 10 showed that pixel observations add a sample-efficiency tax on FetchReachDense -- NatureCNN needed 4x more steps than state SAC, and DrQ augmentation partially closed the gap. But Reach is a single-phase task: move the end-effector to a target in free space. No object interaction, no contact dynamics, no spatial reasoning about small objects.

Push is qualitatively different: the robot must locate a puck (~5 pixels wide at 84x84), approach it, make contact, and push it to a goal position -- all from pixel observations, with sparse binary rewards. This chapter discovers that solving pixel Push requires three innovations beyond what worked for pixel Reach: a manipulation-appropriate CNN encoder, correct gradient routing from critic to encoder, and the patience to let the representation learning phase complete. The chapter is structured as a progressive debugging journey in which we start with the obvious approach (drop in pixels), watch it fail, diagnose why, fix one thing at a time, and arrive at a working solution -- so that the reader feels they are discovering the solution rather than being handed a recipe.

1.2 WHY: The Pixel Push Problem

1.2.1 9.1 Why Push from Pixels Is Harder Than Reach

Every pixel RL challenge from Chapter 10 compounds when we add object interaction, and the compounding is worse than additive.

The first difficulty is **object scale**: the puck in FetchPush is ~5 pixels wide at 84x84 resolution, the gripper is ~3-4 pixels, and the spatial relationship between them -- the ONE thing the policy needs to act on -- occupies a tiny fraction of the image. This is compounded by **sparse rewards in pixel space**: FetchReachDense gave continuous distance feedback so that every arm movement changed the reward, but FetchPush with sparse rewards ($R = 0$ on success, $R = -1$ otherwise) means the CNN must learn useful spatial features with almost no reward signal. HER helps by relabeling goals, but the CNN still needs to extract spatial coordinates from pixels before HER's relabeled rewards become useful.

These two difficulties feed into a deeper structural problem: the agent must solve **two learning problems simultaneously** -- learning visual representations (CNN: pixels -> spatial features) and learning a control policy (actor-critic: spatial features -> push actions). In state-based Push, the first problem does not exist because the observation IS the spatial features; adding pixels means the agent must solve representation learning AND policy learning from scratch, using the same sparse reward signal. Finally, pushing requires understanding **contact dynamics from images** -- does the puck move in the right direction after contact? How far? This temporal reasoning must be inferred from sequences of pixel observations, and while frame stacking (4 frames) helps, the motion signal is subtle at 84x84.

1.2.2 9.2 Visual HER: Two Different Things

"Visual HER" can mean two very different architectures:

Approach	Policy sees	HER relabels	Complexity
Our approach (`goal_mode="both"`)	Pixels + goal vectors	3D goal vectors (trivial swap)	Moderate
Full visual HER (Nair et al. 2018, RIG)	Pixels + goal *images*	Goal images (requires VAE)	High

We use the first approach: the policy sees pixel observations, but HER operates on 3D goal vectors (`achieved_goal`, `desired_goal`), which means relabeling is trivial -- swap the 3D desired_goal with the 3D achieved_goal, with no image-space goal representation needed. This creates an intentional **information asymmetry**: the policy must learn to CONTROL from pixels, but it does not need to learn to SPECIFY GOALS from pixels. In other words, we are testing whether a CNN can learn spatial features good enough for manipulation, not whether it can learn a visual goal representation -- a meaningful middle ground between full-state and full-visual RL.

1.2.3 9.3 The Observation Structure

Our `PixelObservationWrapper` with `goal_mode="both"` and proprioception produces:

```
{
  "pixels":         (12, 84, 84)  uint8   # 4-frame stack x 3 RGB channels
  "proprioception": (10,)        float64  # grip_pos, gripper_state, velocities
  "achieved_goal":  (3,)          float64  # object position (for HER)
  "desired_goal":   (3,)          float64  # target position (for HER)
}
```

The CNN processes only the `pixels` key. Proprioception, achieved_goal, and desired_goal are concatenated as flat vectors alongside the CNN features:

```
pixels (12, 84, 84)     -> CNN -> spatial features (64D)
proprioception (10D)     -> passthrough
achieved_goal (3D)       -> passthrough
desired_goal (3D)        -> passthrough
                            Total: 80D feature vector
```

Why proprioception? The CNN should only learn about the WORLD -- where the puck is, where obstacles are -- because the robot's own state (joint positions, velocities, gripper width) comes from direct sensors with millimeter precision at microsecond latency. Forcing the CNN to also learn ”where is my arm?” wastes capacity on a problem that cheaper sensors already solve, and this mirrors how real robotic systems operate: joint encoders + cameras, never cameras alone.

1.3 HOW: The 5-Step Debugging Journey

1.3.1 9.4 Step 0 -- The Baseline Gap

Experiment: Take the Ch4 pipeline (SAC + HER + sparse Push) and replace the 25D state observation with 84x84 pixels. Use SB3's default NatureCNN encoder.

```
# NatureCNN baseline (default SB3 encoder)
docker run --rm \
  -e MUJOCO_GL=egl -e PYOPENGL_PLATFORM=egl -e PYTHONUNBUFFERED=1 \
  -e HOME=/tmp -e XDG_CACHE_HOME=/tmp/.cache -e TORCH_HOME=/tmp/.cache/torch \
  -e TORCHINDUCTOR_CACHE_DIR=/tmp/.cache/torch_inductor \
  -e MPLCONFIGDIR=/tmp/.cache/matplotlib -e USER=user -e LOGNAME=user \
  -v "$PWD:/workspace" -w /workspace --gpus all --ipc=host \
  robotics-rl:latest \
  bash -c 'source .venv/bin/activate && python scripts/ch09_pixel_push.py train \
    --seed 0 --total-steps 2000000'
```

Result: Flat at 5-8% success for 2M+ steps. Total failure.

Agent	Steps	Success rate
State + HER (Ch4)	2M	89%
Pixel + HER (NatureCNN)	2M	5-8%

The algorithm is proven on state, and the pixel observation pipeline is proven on Reach (Chapter 10), yet combining pixels + sparse rewards + object interaction fails completely.

The question this raises: WHY? What about the visual processing pipeline is wrong for manipulation?

One concept taught: Pixels are not a drop-in replacement for state on manipulation tasks. The visual encoder that works for Atari and for Reach does not work for Push.

1.3.2 9.5 Step 1 -- The Right Eyes (Encoder Architecture)

Diagnosis: NatureCNN (Mnih et al. 2015) uses an 8x8 kernel with stride 4 in the first convolutional layer. On 84x84 input:

```
NatureCNN spatial progression:
  Layer 1: Conv2d(12, 32, 8x8, stride=4)  84 -> 20   (aggressive!)
  Layer 2: Conv2d(32, 64, 4x4, stride=2)  20 ->  9
  Layer 3: Conv2d(64, 64, 3x3, stride=1)   9 ->  7
  Flatten -> Linear(3136, 512)
```

The puck is ~5 pixels wide, so after stride-4 it becomes ~1 pixel, which means the spatial relationship between gripper and puck -- the signal the policy needs -- is destroyed in the FIRST layer. NatureCNN was designed for Atari, where game sprites are 10-30 pixels and decisions are coarse (”go left” vs ”go right”), but manipulation requires millimeter-precision spatial reasoning about objects that are 3-5 pixels wide.

Fix: Three changes, all motivated by the same principle -- preserve spatial information through the encoder pipeline.

1. ManipulationCNN (3x3 kernels, gentle downsampling):

```
ManipulationCNN spatial progression:
  Layer 1: Conv2d(12, 32, 3x3, stride=2, pad=1)  84 -> 42
  Layer 2: Conv2d(32, 32, 3x3, stride=1, pad=1)  42 -> 42
  Layer 3: Conv2d(32, 32, 3x3, stride=1, pad=1)  42 -> 42
  Layer 4: Conv2d(32, 32, 3x3, stride=2, pad=1)  42 -> 21
  Output: (B, 32, 21, 21) feature map
```

A 5-pixel puck survives layer 1 as ~3 pixels (vs ~1 pixel with NatureCNN). The spatial relationship is preserved. This architecture follows DrQ-v2 (Yarats et al. 2022).

```
--8<-- "scripts/labs/manipulation_encoder.py:manipulation_cnn"
```

2. SpatialSoftmax (explicit coordinate extraction):

Instead of flattening the 21x21x32 feature map (14,112 values), SpatialSoftmax extracts the expected (x, y) coordinate of each feature channel's peak activation, producing 2 x 32 = 64 values that represent ”where things are” in the image rather than ”what the image looks like.” This is a powerful inductive bias for manipulation, since the policy needs SPATIAL COORDINATES (where is the puck? where is the gripper?), not a compressed image representation.

```
--8<-- "scripts/labs/manipulation_encoder.py:spatial_softmax"
```

3. Proprioception passthrough (sensor separation):

Robot self-state (grip position, gripper width, velocities) comes from joint encoders, not the camera. Passing 10D proprioception alongside pixel features means the CNN only needs to learn about the WORLD.

```
--8<-- "scripts/labs/manipulation_encoder.py:manipulation_extractor"
```

Experiment: Run with all three fixes:

```
docker run --rm \
  -e MUJOCO_GL=egl -e PYOPENGL_PLATFORM=egl -e PYTHONUNBUFFERED=1 \
  -e HOME=/tmp -e XDG_CACHE_HOME=/tmp/.cache -e TORCH_HOME=/tmp/.cache/torch \
  -e TORCHINDUCTOR_CACHE_DIR=/tmp/.cache/torch_inductor \
  -e MPLCONFIGDIR=/tmp/.cache/matplotlib -e USER=user -e LOGNAME=user \
  -v "$PWD:/workspace" -w /workspace --gpus all --ipc=host \
  robotics-rl:latest \
  bash -c 'source .venv/bin/activate && python scripts/ch09_pixel_push.py train \
    --seed 0 --total-steps 2000000'
```

Result: Still flat at 5-8%. Better architecture is NECESSARY but NOT SUFFICIENT.

Agent	Steps	Success rate
NatureCNN + pixels	2M	5-8%
ManipCNN + SpatialSoftmax + proprio	2M	5-8%

The question this raises: The encoder can now REPRESENT the right spatial information, but is it LEARNING to? Where do encoder gradients come from?

One concept taught: Architecture determines what a network CAN represent, while training determines what it DOES represent. We fixed the capacity; now we need to fix the learning signal.

1.3.3 9.6 Step 2 -- The Right Gradient Flow (Critic-Encoder Routing)

Diagnosis: In SB3's default SAC policy with a shared encoder, the encoder parameters live in the **actor's optimizer**, and during critic training SB3 calls `set_grad_enabled(False)` on the shared encoder -- which means gradients do not flow through the encoder during TD loss computation, so the encoder only learns from **policy gradients** coming from the actor loss. Early in training the policy is random, which makes the actor loss gradient essentially noise, so the encoder receives no useful learning signal.

DrQ-v2 (Yarats et al. 2022) does the opposite: place the encoder in the **critic's optimizer**, where the critic's TD loss directly asks ”does this visual feature help predict future value?” This is a much richer learning signal than the noisy policy gradient, and the actor receives **detached features** so that no encoder gradient flows through the actor loss.

Fix: Override SB3's gradient routing with `DrQv2SACPolicy`. The override has three parts: `CriticEncoderCritic` always enables gradients through the shared encoder during the critic forward pass (removing the `set_grad_enabled(False)` gate), `CriticEncoderActor` detaches features before passing them to the policy MLP (so the encoder does not receive actor loss gradients), and the optimizer wiring places encoder parameters in the critic optimizer while excluding them from the actor optimizer.

```
--8<-- "scripts/labs/drqv2_sac_policy.py:critic_encoder_critic"
```

```
--8<-- "scripts/labs/drqv2_sac_policy.py:critic_encoder_actor"
```

```
--8<-- "scripts/labs/drqv2_sac_policy.py:drqv2_sac_policy"
```

Experiment: Run with the correct architecture + correct gradient routing + larger buffer + more HER relabeling:

```
docker run --rm \
  -e MUJOCO_GL=egl -e PYOPENGL_PLATFORM=egl -e PYTHONUNBUFFERED=1 \
  -e HOME=/tmp -e XDG_CACHE_HOME=/tmp/.cache -e TORCH_HOME=/tmp/.cache/torch \
  -e TORCHINDUCTOR_CACHE_DIR=/tmp/.cache/torch_inductor \
  -e MPLCONFIGDIR=/tmp/.cache/matplotlib -e USER=user -e LOGNAME=user \
  -v "$PWD:/workspace" -w /workspace --gpus all --ipc=host \
  robotics-rl:latest \
  bash -c 'source .venv/bin/activate && python scripts/ch09_pixel_push.py train \
    --seed 0 --critic-encoder --no-drq --buffer-size 500000 \
    --her-n-sampled-goal 8 --total-steps 5000000 --checkpoint-freq 500000'
```

The key flags here are `--critic-encoder` (placing the encoder in the critic optimizer, following the DrQ-v2 pattern), `--no-drq` (disabling DrQ augmentation, for reasons we will explain in Step 4), `--buffer-size 500000` (a larger buffer to retain early exploration data), `--her-n-sampled-goal 8` (more HER relabeling, yielding 88% reward=0 transitions vs 80% at the default 4), and `--checkpoint-freq 500000` (periodic checkpointing every 500K steps to guard against OOM or process kills).

Result: The training curve tells a story in three acts:

Steps	Success rate	What is happening
0 - 2.0M	0% -> 13%	Flat. Encoder warming up. Looks like failure.
2.0M - 2.5M	13% -> 45%	Hockey-stick begins! Exponential climb.
2.5M - 3.5M	45% -> 96%	Positive feedback loop. Rapid convergence.
3.5M - 4.4M	95-98%	Saturation. Agent reliably solves Push from pixels.

Our data (seed 0, from TensorBoard):

Steps	Success rate	Reward	Actor loss	Critic loss
500K	4%	-48.6	~0	0.09 (declining)
1.0M	8%	-47.2	~0	0.07 (declining)
2.0M	13%	-45.0	~0	0.04 (declining)
2.2M	13%	-44.0	0.8	0.25 (RISING)
2.5M	45%	-36.1	1.6	0.46 (RISING)
3.0M	85%	-19.3	1.1	0.35
3.5M	96%	-15.7	0.02	0.45
4.0M	95%	-14.3	-0.5	0.25 (declining)
4.4M	95%	-13.2	-0.8	0.20 (declining)

The hockey-stick emerges at ~2.2M steps. Once it starts, the agent goes from 13% to 95% in about 1.5M steps.

One concept taught: WHERE gradients flow matters as much as WHAT architecture you use, because the encoder needs the critic's value signal (TD loss) -- not the actor's policy signal (which is noisy early on) -- to learn useful visual representations.

1.3.4 9.7 Step 3 -- Understanding the Training Curve (The Patience Tax)

This is not a separate experiment -- it is the INTERPRETATION of Step 2's training curve. The chapter pauses here to explain what the loss curves reveal.

The Three-Phase Loss Signature

In SAC + HER with sparse rewards, losses go through three distinct regimes, and the pattern is deeply counterintuitive: in supervised learning, training progress means loss goes DOWN, but in sparse-reward RL losses first decline (bad), then RISE (good), then decline again (convergence).

Phase 1: Failure Regime (0 - 2.2M steps)

```
success_rate: Flat 3-8%
critic_loss:  Declining to 0.04-0.07
actor_loss:   Near 0
```

The critic's job is trivially easy: with the agent almost always failing, Q converges to a uniform ~-18.5 everywhere (the sum of $\gamma^t \cdot (-1)$ for 50 steps with $\gamma = 0.95$), so TD error is tiny because the Q-landscape is uniformly wrong -- predicting failure everywhere and being "right" because the agent always fails. The actor, in turn, receives no useful gradient. SAC's actor loss is approximately $\alpha \log \pi(a|s) - Q(s,a)$, and when Q is the same for all actions the gradient is zero, which means actor loss near 0 signals that "the critic sees no difference between actions."

A declining critic loss with flat success is a RED flag, not a green one.

Phase 2: Hockey-Stick Regime (2.2M - 3.5M steps)

```
success_rate: Rising 13% -> 96%
critic_loss:  RISING to 0.3-0.8
actor_loss:   RISING to 0.5-1.6
```

Now some trajectories succeed ($R = 0$ on success steps) and others fail ($R = -1$), so the Q-landscape becomes heterogeneous: $Q(s,a) \approx 0$ for good state-action pairs, $Q(s,a) \approx -18.5$ for bad ones. The critic must now distinguish between states where the puck is reachable and states where it is not, which is genuinely HARDER -- hence the higher loss.

Rising losses during the hockey-stick are a GREEN flag.

The positive feedback loop activates here: the critic learns value structure, which gives the actor real gradients, which produces a better policy, which generates more successes, which creates more diverse Bellman targets, which helps the critic learn finer structure. HER amplifies this cycle because each success relabels N future transitions with $R = 0$, seeding new value wavefronts through the goal space. This self-amplifying loop is why the curve goes exponential.

Phase 3: Convergence Regime (3.5M+ steps)

```
success_rate: Saturating 95-98%
critic_loss:  Declining to 0.15-0.35
actor_loss:   Turning NEGATIVE (-0.2 to -0.8)
```

The critic's value function is now mostly accurate: most trajectories succeed ($Q \approx 0$ for most states), so the Q-landscape becomes uniform again -- but uniform SUCCESS rather than uniform failure. Declining critic loss here is a GREEN flag, indicating that the value function has converged to the true Q. Actor loss turning negative means $\alpha \log \pi < Q(s,a)$, where the Q-values dominate the entropy penalty -- the SAC-specific signal that the policy has converged.

Summary table for monitoring pixel RL:

Metric	Phase 1: Failure	Phase 2: Hockey-Stick	Phase 3: Convergence
success_rate	Flat 3-8%	Rising 10% -> 90%+	Saturating 95%+
critic_loss	Declining < 0.1 (RED)	Rising 0.3-0.8 (GREEN)	Declining 0.15-0.35 (GREEN)
actor_loss	Near 0 (no signal)	Rising 0.5-1.6 (real signal)	Negative (converged)

How to distinguish Phase 1 declining from Phase 3 declining: The answer is always to check success rate alongside the loss. In Phase 1, critic_loss declines while success remains flat; in Phase 3, critic_loss declines while success is high. The lesson is to never read loss curves in isolation.

The Training Budget Rule

Pixel RL needs 2-4x the training budget of state RL for the same task:

Agent	Hockey-stick onset	40%+ success	Ratio vs state
Full-state (25D, Ch4 baseline)	~1.2M	~1.8M	1.0x
Pixel (84x84, this chapter)	~2.2M	~2.5M	~1.8x

This matches the literature: Yarats et al. (2022, DrQ-v2) report that pixel agents on DMControl tasks typically need 2-3x the frames of state agents to reach equivalent performance. The overhead exists because the pixel agent must LEARN spatial representations before the value function becomes meaningful, so the flat first ~2M steps look like failure but are actually the encoder warm-up phase -- a necessary overhead, not a sign that something is broken.

The stop-rule trap: A stop rule calibrated for state RL ("abort if <10% at 1M steps") becomes a self-fulfilling prophecy for pixel RL, since you terminate the run precisely because the encoder has not yet learned, which ensures you never see it learn.

The practical stopping rule: Conversely, once the agent has converged, there is no reason to keep training. In our experiments, success rate plateaued at 95-96% around 3.5M steps, and running to 5M burned ~15 extra hours of compute for no meaningful improvement. We find the following heuristic useful: do not stop before 3M steps (since the hockey-stick may not have completed), but stop when success rate has been stable above 90% for 500K+ steps, provided that critic_loss is declining in Phase 3 rather than Phase 1 (check success rate to disambiguate). In our seed 0 run, 4M steps was more than sufficient, and the 3.5M checkpoint at 95% would also have been a defensible stopping point.

One concept taught: How to READ training curves in pixel RL. Rising losses are the hockey-stick signal, while declining losses with flat success are the failure signal -- the opposite of supervised learning intuition. And once convergence is clear, stopping early saves compute better spent on additional seeds.

1.3.5 9.8 Step 4 -- The DrQ Surprise (Augmentation vs Representation)

Setup: The reader might wonder: "We did not use DrQ in Step 2. DrQ is the standard data augmentation for pixel RL (Chapter 10). Would adding it help?"

Experiment: Compare Step 2's result (no DrQ, 95% at ~4M) with the SAME config plus DrQ augmentation:

```
# Same config as Step 2 but WITH DrQ (remove --no-drq)
docker run --rm \
  -e MUJOCO_GL=egl -e PYOPENGL_PLATFORM=egl -e PYTHONUNBUFFERED=1 \
  -e HOME=/tmp -e XDG_CACHE_HOME=/tmp/.cache -e TORCH_HOME=/tmp/.cache/torch \
  -e TORCHINDUCTOR_CACHE_DIR=/tmp/.cache/torch_inductor \
  -e MPLCONFIGDIR=/tmp/.cache/matplotlib -e USER=user -e LOGNAME=user \
  -v "$PWD:/workspace" -w /workspace --gpus all --ipc=host \
  robotics-rl:latest \
  bash -c 'source .venv/bin/activate && python scripts/ch09_pixel_push.py train \
    --seed 0 --critic-encoder --buffer-size 500000 \
    --her-n-sampled-goal 8 --total-steps 2000000'
```

Result: DrQ + SpatialSoftmax = flat at 3-6% at 1.5M steps. DrQ HURTS.

Config	Success @ 1.5M	Success @ 4M
No DrQ + SpatialSoftmax (Step 2)	6% (pre-hockey-stick)	95%
DrQ + SpatialSoftmax	3-6%	-- (terminated)

Diagnosis: DrQ shifts images by ± 4 pixels randomly, which after the CNN (84 -> 21 spatial) becomes ± 1 pixel on the feature map. Since SpatialSoftmax converts feature map positions to coordinates in $[-1, 1]$, a ± 1 pixel shift at 21x21 resolution becomes ± 0.10 in coordinate space. Crucially, obs and next_obs are augmented INDEPENDENTLY with different random shifts, which DOUBLES the noise in Bellman targets:

$$y = r + \gamma Q(s', a') \quad \text{where } s' = \text{aug}(s'_{\text{raw}})$$

The gripper-puck distance signal in SpatialSoftmax coordinates is only ~0.25-0.50, so with ± 0.10 independent noise on both obs and next_obs, the noise-to-signal ratio reaches 40-80% -- too noisy for the critic to learn stable value structure.

Why DrQ works on Reach but not on Push: On Reach, the NatureCNN flattens the feature map into a 512D vector, and DrQ's ± 4 pixel shift creates augmented views that regularize the Q-function against pixel-level overfitting. With SpatialSoftmax, by contrast, the representation is already low-dimensional coordinates, which means there is no pixel-level overfitting to regularize and the noise directly corrupts the spatial signal that the critic depends on.

One concept taught: Data augmentation is not universally good -- you must match your augmentation to your representation, since SpatialSoftmax extracts precise spatial coordinates while DrQ injects spatial noise, making them fundamentally incompatible.

Generalizable principle: ”Choose your representation, then choose your augmentation.”

1.4 BUILD IT: Key Components

1.4.1 9.9 Rendering and Pixel Wrapping

The pixel observation wrapper from Chapter 10 is reused with one addition -- proprioception passthrough:

```
--8<-- "scripts/labs/pixel_wrapper.py:render_and_resize"
```

```
--8<-- "scripts/labs/pixel_wrapper.py:pixel_obs_wrapper"
```

!!! lab ”Checkpoint” Verify the wrapper produces correct observation spaces: `python python scripts/labs/pixel_wrapper.py --verify` Expected: `[ALL PASS] Pixel wrapper verified`

1.4.2 9.10 ManipulationCNN + SpatialSoftmax

The encoder pipeline chains three components: ManipulationCNN produces a feature map, SpatialSoftmax extracts spatial coordinates from that map, and ManipulationExtractor routes image and vector keys to the appropriate sub-encoders.

Verification:

```
python scripts/labs/manipulation_encoder.py --verify
```

Expected output confirms that SpatialSoftmax coordinates fall in the $[-1, 1]$ range, the ManipulationCNN output shape is `(B, 32, 21, 21)`, and the ManipulationExtractor produces `features_dim = 80` (64 spatial + 10 proprio + 3 ag + 3 dg).

1.4.3 9.11 DrQ-v2 Gradient Routing

The gradient routing override that makes the encoder learn from critic loss:

Verification:

```
python scripts/labs/drqv2_sac_policy.py --verify
```

Expected output confirms that the encoder parameters appear in the critic optimizer (but not the actor optimizer), that running `critic.backward()` produces encoder gradients while `actor.backward()` does not (because features are detached), that the target critic has its own encoder copy rather than sharing the online encoder, and that save/load preserves the gradient routing configuration.

1.5 WHAT: Run It

1.5.1 9.12 Quick Start

The recommended configuration for pixel Push:

```
# Full training (seed 0, ~40 hours)
docker run --rm \
  -e MUJOCO_GL=egl -e PYOPENGL_PLATFORM=egl -e PYTHONUNBUFFERED=1 \
  -e HOME=/tmp -e XDG_CACHE_HOME=/tmp/.cache -e TORCH_HOME=/tmp/.cache/torch \
  -e TORCHINDUCTOR_CACHE_DIR=/tmp/.cache/torch_inductor \
  -e MPLCONFIGDIR=/tmp/.cache/matplotlib -e USER=user -e LOGNAME=user \
  -v "$PWD:/workspace" -w /workspace --gpus all --ipc=host \
  robotics-rl:latest \
  bash -c 'source .venv/bin/activate && python scripts/ch09_pixel_push.py train \
    --seed 0 --critic-encoder --no-drq --buffer-size 500000 \
    --her-n-sampled-goal 8 --total-steps 5000000 --checkpoint-freq 500000'
```

Configuration summary:

Parameter	Value	Why
`--critic-encoder`	Enabled	DrQ-v2 gradient routing (Step 2)
`--no-drq`	DrQ disabled	SpatialSoftmax incompatibility (Step 4)
`--buffer-size 500000`	500K transitions	~40 GB RAM; retains early exploration
`--her-n-sampled-goal 8`	HER-8	88% reward=0 transitions; stronger critic signal
`--total-steps 5000000`	5M steps	Conservative budget; can stop at 3.5-4M if converged
`--checkpoint-freq 500000`	Every 500K steps	Prevents losing training to OOM/kill

Default architecture (ManipulationCNN + SpatialSoftmax + proprioception) is used unless overridden.

1.5.2 9.13 Monitoring

TensorBoard:

```
tensorboard --logdir runs --bind_all
```

Key metrics to watch:

- `rollout/success_rate`: the primary metric
- `train/critic_loss`: watch for the three-phase signature (Section 9.7)

- `train/actor_loss`: rising = good (Phase 2), negative = converged (Phase 3)
- `time/fps`: expect 25-35 fps for pixel training with 4 envs

Milestones (from our seed 0 data):

Steps	Expected success	If you see this instead
500K	4-8% (flat)	Normal. Encoder warming up.
1.0M	6-10% (flat)	Normal. Still in Phase 1.
2.0M	10-15% (slight upward trend)	If flat at 5%: check `--critic-encoder` is set
2.5M	30-60% (hockey-stick)	If still flat: extend to 4M before aborting
3.0M	85-93%	If below 50%: something is likely misconfigured
3.5M	95%+ (converged)	Safe to stop here

When to stop: You do not need to run all 5M steps, since once the success rate has been stable above 90% for 500K+ steps and critic_loss is declining (Phase 3), the agent has converged. In our experience, 3.5-4M steps is sufficient, and running past convergence wastes compute that is better spent on additional seeds.

1.5.2.1 Memory: The Pixel Buffer Is the Binding Constraint

For pixel RL, RAM -- not GPU speed -- is usually the limiting resource, because the replay buffer stores every transition's `obs` and `next_obs` as pixel arrays that dominate memory.

Per-transition breakdown. Each transition stores two observations (obs and next_obs), each containing a 4-frame stack of 84x84 RGB images:

```
Per observation:  12 x 84 x 84 x 1 byte (uint8) = 84,672 bytes (~83 KB)
Per transition:   2 x 84,672 = 169,344 bytes (~165 KB)
                  + proprioception, goals, actions, rewards (~200 bytes)
                  Total: ~170 KB per transition
```

For comparison, a state-based transition (25D float64 obs + goals + actions) uses ~200 bytes. **Pixel transitions are ~850x larger.**

Buffer size to RAM table:

`--buffer-size`	Buffer RAM	Total with model + envs	Recommended for
500,000	~80 GB	~85-90 GB	120+ GB machines (DGX)
300,000	~48 GB	~53-58 GB	64 GB machines
200,000	~32 GB	~37-42 GB	48 GB machines
100,000	~16 GB	~21-24 GB	32 GB machines

Per-tier practical advice:

- **120+ GB RAM (DGX, large workstations):** Use the recommended `--buffer-size 500000`, which retains enough early exploration for the hockey-stick to emerge on schedule (~2.2M steps).
- **64 GB RAM:** Use `--buffer-size 300000`. The hockey-stick onset may shift later (perhaps ~3M steps instead of ~2.2M) because the buffer forgets early exploration sooner, though the final success rate should still reach 90%+.
- **32 GB RAM:** Use `--buffer-size 100000`. The agent may plateau at 70-85% and need 5-6M steps to reach 90%+, since the smaller buffer means the agent ”forgets” early diverse experience faster, weakening the HER relabeling signal.

- **16 GB RAM:** Experimental. Consider running the state-only variant (`--full-state`) first to verify the pipeline, then attempt pixels with `--buffer-size 50000` -- though success is not guaranteed, because the buffer may be too small for the hockey-stick to emerge.

Why buffer size matters for HER. HER relabels transitions using goals from *future* transitions in the same episode, so when the buffer is small relative to training length, old episodes are overwritten before their relabeled variants propagate value through the Bellman equation. This delays or weakens the hockey-stick phase transition, because the positive feedback loop (more successes -> better value estimates -> more successes) needs a critical mass of diverse experience in the buffer to ignite.

Note: SB3's `DictReplayBuffer` does not support `optimize_memory_usage` (which halves memory by storing `next_obs` implicitly). The RAM figures above are what you will actually see.

1.5.3 9.14 Resume from Checkpoint

If training is interrupted (OOM, process kill), resume from the most recent periodic checkpoint:

```
# Find the latest checkpoint
ls -t checkpoints/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed0_*_steps.zip | head -1

# Resume (--total-steps = target total, not remaining)
docker run --rm \
  -e MUJOCO_GL=egl -e PYOPENGL_PLATFORM=egl -e PYTHONUNBUFFERED=1 \
  -e HOME=/tmp -e XDG_CACHE_HOME=/tmp/.cache -e TORCH_HOME=/tmp/.cache/torch \
  -e TORCHINDUCTOR_CACHE_DIR=/tmp/.cache/torch_inductor \
  -e MPLCONFIGDIR=/tmp/.cache/matplotlib -e USER=user -e LOGNAME=user \
  -v "$PWD:/workspace" -w /workspace --gpus all --ipc=host \
  robotics-rl:latest \
  bash -c 'source .venv/bin/activate && python scripts/ch09_pixel_push.py train \
    --seed 0 --critic-encoder --no-drq --buffer-size 500000 \
    --her-n-sampled-goal 8 --total-steps 5000000 --checkpoint-freq 500000 \
    --resume checkpoints/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed0_2500000_steps.zip'
```

What resumes: Network weights and optimizer state resume from the checkpoint, but the replay buffer does NOT resume (since it is too large to serialize for pixel buffers) and instead refills from the trained policy's behavior within ~1K steps.

What `--total-steps` means on resume: This is the TARGET total, not the remaining steps -- so if the checkpoint is at 2.5M and `--total-steps` is 5M, the script trains for 2.5M more steps.

1.5.4 9.15 Evaluation

```
docker run --rm \
  -e MUJOCO_GL=egl -e PYOPENGL_PLATFORM=egl -e PYTHONUNBUFFERED=1 \
  -e HOME=/tmp -e XDG_CACHE_HOME=/tmp/.cache -e TORCH_HOME=/tmp/.cache/torch \
  -e TORCHINDUCTOR_CACHE_DIR=/tmp/.cache/torch_inductor \
  -e MPLCONFIGDIR=/tmp/.cache/matplotlib -e USER=user -e LOGNAME=user \
  -v "$PWD:/workspace" -w /workspace --gpus all --ipc=host \
  robotics-rl:latest \
  bash -c 'source .venv/bin/activate && python scripts/ch09_pixel_push.py eval \
    --ckpt checkpoints/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed0.zip \
    --critic-encoder --n-eval-episodes 100'
```

1.5.5 9.16 Full-State Control

To verify the Ch9 pipeline is wired correctly independently of pixel processing:

```
docker run --rm \
  -e MUJOCO_GL=egl -e PYOPENGL_PLATFORM=egl -e PYTHONUNBUFFERED=1 \
  -e HOME=/tmp -e XDG_CACHE_HOME=/tmp/.cache -e TORCH_HOME=/tmp/.cache/torch \
  -e TORCHINDUCTOR_CACHE_DIR=/tmp/.cache/torch_inductor \
  -e MPLCONFIGDIR=/tmp/.cache/matplotlib -e USER=user -e LOGNAME=user \
  -v "$PWD:/workspace" -w /workspace --gpus all --ipc=host \
  robotics-rl:latest \
  bash -c 'source .venv/bin/activate && python scripts/ch09_pixel_push.py train \
    --seed 0 --full-state --total-steps 2000000'
```

Expected: 85-90% success at 2M steps (matching Ch4 results), and if this fails the problem is in the script wiring rather than the visual pipeline.

1.6 What Can Go Wrong

Symptom	Likely cause
Flat at 5% after 3M+ steps	Missing `--critic-enco`
OOM during training	Buffer too large for ava
`RuntimeError: Unable to sample before end of first episode` (on resume)	Learning starts not shift
Flat at 5% WITH `--critic-encoder` after 2M	Pre-hockey-stick phase;
`critic_loss` declining monotonically, success flat	Phase 1 failure regime (
Full-state control at 0%	Missing object dynamic
DrQ + SpatialSoftmax at 3%	Augmentation-represen
Very slow FPS (<15)	Rendering at 480x480 v

1.7 Summary

1.7.1 Key Findings

1. **NatureCNN fails on manipulation** (5% flat): stride-4 first layer destroys small objects. ManipulationCNN (3x3, stride-2) preserves spatial information.
2. **Architecture is necessary but not sufficient** (still 5% with ManipCNN): The encoder can represent spatial information but is not learning to extract it.
3. **Critic-encoder gradient routing is the key** (95% at 4.4M): Placing the encoder in the critic's optimizer provides the TD loss as a learning signal. The encoder learns "which visual features predict future value?"
4. **The hockey-stick takes patience** (2.2M steps before lift-off): Pixel RL needs 2-4x the training budget of state RL. The flat first ~2M steps are encoder warm-up, not failure.
5. **DrQ + SpatialSoftmax is harmful** (3% flat): Random shift augmentation injects ± 0.10 coordinate noise into a signal that spans $\pm 0.25 - 0.50$. Augmentation must match representation.

1.7.2 The Algorithmic Recipe

Component	Choice	Why
CNN encoder	ManipulationCNN (3x3, stride 2)	Preserves small objects (puck, gripper)
Feature head	SpatialSoftmax (2C coordinates)	Inductive bias for spatial reasoning
Proprioception	10D robot state alongside pixels	CNN learns world, not self
Gradient routing	Critic-encoder (DrQ-v2)	TD loss teaches encoder what matters
Augmentation	None (no DrQ)	SpatialSoftmax + DrQ = incompatible
HER	n_sampled_goal=8	88% relabeled transitions for critic signal
Buffer	500K transitions	Retains early exploration; fits in ~40 GB
Training budget	3.5-4M steps (2-4x state)	Stop when converged; 5M is conservative upper bounc

1.7.3 Concepts Introduced

Concept	Definition
ManipulationCNN	3x3-kernel CNN with gentle downsampling (stride 2 in layers 1 and 4 only)
SpatialSoftmax	Converts feature maps to expected (x, y) coordinates per channel
Proprioception passthrough	Robot self-state via sensors, not camera; CNN learns world only
Critic-encoder gradient routing	Encoder in critic optimizer; actor receives detached features
Hockey-stick learning curve	Geometric phase transition in sparse-reward pixel RL
Three-phase loss signature	Failure (declining) -> hockey-stick (rising) -> convergence (declining)
Training budget multiplier	Pixel RL needs 2-4x the steps of state RL for the same task

1.7.4 Files

File	Purpose
`scripts/ch09_pixel_push.py`	Training, evaluation, comparison CLI
`scripts/labs/manipulation_encoder.py`	ManipulationCNN, SpatialSoftmax, ManipulationExtractor
`scripts/labs/drqv2_sac_policy.py`	DrQv2SACPolicy (critic-encoder gradient routing)
`scripts/labs/pixel_wrapper.py`	PixelObservationWrapper (reused from Ch10)

1.7.5 Artifacts

Artifact	Location
Pixel Push checkpoint	`checkpoints/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed{N}.zip`
Training metadata	`checkpoints/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed{N}.meta.json`
Periodic checkpoints	`checkpoints/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed{N}_{steps}_st`
Evaluation results	`results/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed{N}_eval.json`
TensorBoard logs	`runs/ch09_manip_noDrQ_criticEnc/{env}/seed{N}/`

1.8 Reproduce It

The results in this chapter come from these runs:

```
# Seed 0 (the data shown in sections 9.6-9.8)
python scripts/ch09_pixel_push.py train \
  --seed 0 --critic-encoder --no-drq --buffer-size 500000 \
  --her-n-sampled-goal 8 --total-steps 5000000 --checkpoint-freq 500000

# Seeds 1-2 (for multi-seed reproducibility)
python scripts/ch09_pixel_push.py train \
  --seed 1 --critic-encoder --no-drq --buffer-size 500000 \
  --her-n-sampled-goal 8 --total-steps 5000000 --checkpoint-freq 500000

python scripts/ch09_pixel_push.py train \
  --seed 2 --critic-encoder --no-drq --buffer-size 500000 \
  --her-n-sampled-goal 8 --total-steps 5000000 --checkpoint-freq 500000
```

Hardware: NVIDIA DGX (any modern GPU works; times vary). **Time:** ~25-30 hours per seed at ~30 fps (to 4M steps; 5M is the conservative upper bound, but convergence typically occurs by 3.5-4M). **RAM:** ~40-50 GB per run (500K pixel buffer + model + envs).

Practical tip: Monitor `rollout/success_rate` in TensorBoard, and once it has been stable above 90% for 500K+ steps while `train/critic_loss` is declining (Phase 3, not Phase 1 -- check success rate to tell them apart), you can stop early and use the most recent periodic checkpoint, since running past convergence wastes compute better spent on additional seeds.

Results (seed 0, stopped at 4M steps):

Metric	Value
Success rate (at 4M)	96%
Hockey-stick onset	~2.2M steps
90%+ success	~3.0M steps
Training time (to 4M)	~25 hours
FPS	~28-30

Checkpoints saved (seed 0):

Steps	Success rate	File
2,500,000	~60%	`..._seed0_2500000_steps.zip`
3,000,000	~92%	`..._seed0_3000000_steps.zip`
3,500,000	~95%	`..._seed0_3500000_steps.zip`
4,000,000	96%	`..._seed0_4000000_steps.zip`

Multi-seed results: Seeds 1 and 2 are queued. The chapter will be updated with 3-seed statistics when complete.

If your numbers differ significantly (hockey-stick not visible by 3M, or final success below 80%), check the ”What Can Go Wrong” section above.

1.9 Exercises

1. **(Verify)** Run the full-state control (`--full-state --total-steps 2000000`) and confirm it reaches ~85-90% success. This validates the SAC+HER pipeline independently of pixel processing.

2. **(Tweak)** Change `--her-n-sampled-goal` from 8 to 4 (the default). Does the hockey-stick onset shift later? What about final success rate?
3. **(Explore)** Run with `--no-spatial-softmax` (uses flatten + linear instead of SpatialSoftmax). Does the agent still learn? The flatten pathway produces 50D features instead of 64D spatial coordinates. What does this tell you about whether SpatialSoftmax is necessary or just helpful?
4. **(Challenge)** Run WITH DrQ but WITHOUT SpatialSoftmax (`--no-spatial-softmax`, without `--no-drq`). DrQ was designed for flat CNN features, not spatial coordinates. Does removing SpatialSoftmax make DrQ useful again?
5. **(Challenge)** Read the `verify_gradient_flow()` function in `scripts/labs/drqv2_sac_policy.py`. Modify it to measure the actual gradient MAGNITUDE flowing through the encoder during critic vs actor training. How much larger is the critic gradient? (This quantifies the argument in Section 9.6.)

1.9.1 References

- Mnih, V. et al. (2015). ”Human-level control through deep reinforcement learning.” *Nature*, 518(7540), 529-533.
- Yarats, D. et al. (2022). ”Mastering Visual Continuous Control: Improved Data-Augmented Reinforcement Learning.” arXiv:2107.09645.
- Levine, S. et al. (2016). ”End-to-End Training of Deep Visuomotor Policies.” JMLR 17(39):1-40. arXiv:1504.00702.
- Kostrikov, I. et al. (2020). ”Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels.” arXiv:2004.13649.
- Andrychowicz, M. et al. (2017). ”Hindsight Experience Replay.” arXiv:1707.01495.
- Nair, A. et al. (2018). ”Visual Reinforcement Learning with Imagined Goals.” NeurIPS 2018. arXiv:1807.04742.
- Haarnoja, T. et al. (2018). ”Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor.” arXiv:1801.01290.