Contents

1 Chapter 10: Pixels, No Cheating -- Visual SAC on FetchReachDense

Week 10 Goal: Quantify the cost of learning from raw pixels instead of privileged state, and recover part of the gap with data augmentation (DrQ).

1.1 Bridge: From Robustness Analysis to Visual Observations

In Chapter 7, we injected observation noise into trained policies and measured how quickly success degraded. That analysis assumed the policy operated on low-dimensional state vec-

tors -- the 10-float Fetch observation that encodes end-effector position, velocity, and gripper state. A real robot does not have access to this vector. It has cameras.

This chapter asks: **what happens when we replace the privileged state vector with raw pixel images?** We train SAC on FetchReachDense in three configurations -- state, pixels, and pixels with DrQ augmentation -- and measure the sample-efficiency gap quantitatively. The goal is not just to "make it work from pixels" but to understand *why* pixels are harder and *what* helps.

This chapter is self-contained and can be read directly after Chapter 7 -- no concepts beyond those introduced in Chapters 0-7 are needed.

1.2 WHY: The Pixel Observation Problem

1.2.1 10.1 Why Pixels Matter

Every Fetch experiment so far has used the `observation` vector: a 10D float array containing the end-effector's Cartesian position (3), velocity (3), gripper width (1), and related quantities. This is **privileged information** -- a real robot does not know its end-effector position to millimeter accuracy without a calibrated motion capture system.

Cameras are the natural sensor for robotic manipulation, but replacing a 10D float vector with an 84x84x3 RGB image (21,168 values) creates several compounding difficulties that interact in ways worth understanding individually:

1. **Dimensionality.** The observation grows from 10 to 21,168 -- a 2,100x increase. The policy network must extract spatial features before it can reason about actions.
2. **Partial observability.** A single image does not encode velocity. The policy must infer motion from pixel differences across frames (or learn to act without velocity information).
3. **Rendering cost.** Each `env.step()` now requires a MuJoCo render call. For Fetch at 480x480 (the default), this is ~230,000 pixels per frame, per environment. Rendering becomes the training bottleneck -- not the neural network.
4. **Q-function overfitting.** With high-dimensional image inputs, the Q-network can memorize pixel-level details of specific transitions rather than learning generalizable value estimates. This is the problem DrQ addresses.

1.2.2 10.2 Observation Design Choices

When wrapping a goal-conditioned environment for pixel observations, we face a design decision: **which goal information (if any) do we expose alongside the image?**

Our `PixelObservationWrapper` supports three modes:

`goal_mode`	Observation keys	What the policy sees
`"none"`	`{"pixels"}`	Image only -- must infer both current sta
`"desired"`	`{"pixels", "desired_goal"}`	Image + 3D goal position (privileged goa
`"both"`	`{"pixels", "achieved_goal", "desired_goal"}`	Image + both goal vectors

We default to `goal_mode="none"` -- the hardest setting, where the policy must extract everything from the image -- since this is the most realistic scenario: a camera sees the scene, and the goal (a target position) is indicated visually (e.g., by a marker in the scene).

Non-example: `goal_mode="both"` with pixel observations is not "learning from pixels" in any meaningful sense -- the policy has direct access to the 3D coordinates it needs to solve the task. We include it as an experimental control, not as a recommended configuration.

1.2.3 10.3 The Sample-Efficiency Gap

The central quantity this chapter measures is the cost of replacing state vectors with pixel observations, expressed as a ratio:

Definition (Sample-Efficiency Ratio). Given two agents trained on the same task with different observation types, the sample-efficiency ratio is:

$$\rho = \frac{N_{\text{pixel}}}{N_{\text{state}}}$$

where N_{pixel} and N_{state} are the number of environment steps each agent needs to reach a target success rate (e.g., 90%). A ratio $\rho = 4$ means pixel training needs 4x more samples.

In our experiments below, we measure this ratio on FetchReachDense and find that augmentation (DrQ) substantially reduces it.

1.3 HOW: Three Approaches

1.3.1 10.4 Approach 1: State-Based SAC (Baseline)

This is the SAC configuration from Chapter 3: `MultiInputPolicy` on the dictionary observation with flat state vectors, where SB3 routes the observation, achieved_goal, and desired_goal through its `CombinedExtractor`, which flattens and concatenates them into a single vector for the actor and critic MLPs. This configuration serves as our performance ceiling -- the best we can do with privileged information.

1.3.2 10.5 Approach 2: Pixel SAC (No Augmentation)

We wrap FetchReachDense with `PixelObservationWrapper` (from `scripts/labs/pixel_wrapper.py`), replacing the flat observation with an 84x84 RGB image. SB3's `MultiInputPolicy` detects the image space via `is_image_space()` and automatically routes the `"pixels"` key through a NatureCNN encoder (Mnih et al., 2015), so no manual feature extractor configuration is needed.

The rendering pipeline proceeds as follows:

```
env.render() -> 480x480 HWC uint8
             -> PIL resize to 84x84 (bilinear)
             -> transpose to CHW
             -> PixelObservationWrapper stores as obs["pixels"]
```

SB3 then normalizes uint8 [0, 255] to float32 [0, 1] internally.

1.3.3 10.6 Approach 3: Pixel SAC + DrQ Augmentation

DrQ (Kostrikov et al., 2020) is a single, clean idea: **augment pixel observations at replay buffer sample time** with random spatial shifts, so that each time a transition is replayed, the Q-network sees a slightly different crop of the image. This prevents the Q-function from

memorizing pixel-level details of specific transitions, which is the dominant failure mode for pixel-based RL without regularization.

Definition (Random Shift Augmentation). Given an image x of size (H, W) and pad size p:

1. Pad x by p pixels on all sides using replicate padding, producing a $(H + 2p, W + 2p)$ image.
2. Randomly crop back to (H, W).

For our setup ($H = W = 84$, $p = 4$), this creates shifts of up to ± 4 pixels ($\sim 5\%$ of the image). Replicate padding avoids black borders that would create artificial edge features.

The entire implementation is a replay buffer wrapper -- requiring no changes to the loss function, network architecture, or training loop -- which is what makes DrQ elegant: the algorithmic innovation reduces to a single clean abstraction that slots into any existing off-policy pipeline.

1.4 BUILD IT: Pixel Observation Pipeline (10.7)

This section walks through the pedagogical implementations in `scripts/labs/pixel_wrapper.py`, `scripts/labs/visual_encoder.py`, and `scripts/labs/image_augmentation.py`.

1.4.1 10.7.1 Rendering and Resizing

The core rendering function maps MuJoCo's high-resolution output to the 84x84 images our CNN expects:

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

When MuJoCo renders natively at 84x84 (via `gym.make(..., width=84, height=84)`), the function detects the matching size and skips PIL entirely -- this is the fast path used by `--fast` mode (Section 10.11).

1.4.2 10.7.2 Pixel Observation Wrapper

The wrapper replaces the flat state observation with a rendered image, while optionally preserving goal vectors:

```
--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`

1.4.3 10.7.3 NatureCNN Encoder

The NatureCNN architecture from Mnih et al. (2015) maps 84x84x3 images to a 512-dimensional feature vector through three convolutional layers of decreasing kernel size followed by a fully connected projection:

```
Conv2d(3, 32, 8x8, stride=4)  ->  ReLU      84 -> 20
Conv2d(32, 64, 4x4, stride=2) ->  ReLU      20 -> 9
Conv2d(64, 64, 3x3, stride=1) ->  ReLU       9 -> 7
Flatten                        ->  Linear(3136, 512)  ->  ReLU
```

```
--8<-- "scripts/labs/visual_encoder.py:nature_cnn"
```

The dummy forward pass trick (`torch.zeros(1, C, H, W)` through the conv layers) computes the flatten dimension automatically, which avoids hard-coding 3136 and makes the encoder work for image sizes other than 84x84.

!!! lab "Checkpoint" Verify encoder shapes and parameter count: `python python scripts/labs/visual_encoder.py --verify` Expected: `[ALL PASS] Visual encoder verified`

1.4.4 10.7.4 DrQ: Random Shift Augmentation

The augmentation function pads and randomly crops pixel observations:

```
--8<-- "scripts/labs/image_augmentation.py:random_shift_aug"
```

Key design choice: replicate padding extends border pixels outward. Zero padding would create black borders at the crop edges, which the Q-network could learn to exploit as an artificial feature.

1.4.5 10.7.5 DrQ Replay Buffer

The replay buffer wrapper applies augmentation at sample time -- the only change needed to add DrQ to an existing SAC pipeline:

```
--8<-- "scripts/labs/image_augmentation.py:drq_replay_buffer"
```

This is the entire DrQ integration: the buffer stores un-augmented observations (so that they can be replayed with different augmentations each time) and applies `aug_fn` to the pixel key when sampling, while goals, actions, and rewards pass through unchanged.

!!! lab "Checkpoint" Verify augmentation preserves shapes and only affects pixels: `python python scripts/labs/image_augmentation.py --verify` Expected: `[ALL PASS] Image augmentation verified`

1.4.6 10.7.6 Pixel Replay Buffer (uint8 Storage)

For the from-scratch implementation (`visual_encoder.py --demo`), we use a custom replay buffer that stores images as uint8 for 4x memory savings:

??? lab "PixelReplayBuffer (click to expand)" `python --8<-- "scripts/labs/pixel_wrapper.py:pixel_rep`

Standard SB3 replay buffers store observations as float32, which for 84x84x3 images means 84,672 bytes per image. Storing as uint8 instead costs only 21,168 bytes -- a 4x savings that, with 200K transitions (obs + next_obs), amounts to ~25 GB of memory saved. The conversion to float32 [0, 1] happens at sample time, when the batch is small (typically 256), so the cost of the type cast is negligible.

1.5 WHAT: Experiments and Expected Results (Run It)

1.5.1 10.8 Quick Start

The full pipeline trains all three variants and produces a comparison table:

```
bash docker/dev.sh python scripts/ch10_visual_reach.py all --seed 0
```

This runs seven steps in sequence (train-state, eval-state, train-pixel, eval-pixel, train-pixel-drq, eval-drq, compare), with a total wall time of approximately 10-12 hours on a DGX with GPU.

For faster iteration, use `--fast` mode (Section 10.11), which reduces pixel rendering overhead and parallelizes environment execution:

```
bash docker/dev.sh python scripts/ch10_visual_reach.py all --seed 0 --fast
```

1.5.2 10.9 Experiment 1: State-Based SAC (Baseline)

```
bash docker/dev.sh python scripts/ch10_visual_reach.py train-state --seed 0
bash docker/dev.sh python scripts/ch10_visual_reach.py eval --ckpt checkpoints/sac_state_F
```

FetchReachDense-v4 -- State SAC (500K steps):

Metric	Value
Success rate	100.0%
Return (mean)	-0.730 +/- 0.245
Final distance (mean)	0.0116
Training time	21.3 min (1280s)
FPS	391

State SAC should converge to ~100% success rate within 200-300K steps, which establishes the performance ceiling against which we measure the pixel agents.

1.5.3 10.10 Experiment 2: Pixel SAC (No Augmentation)

```
bash docker/dev.sh python scripts/ch10_visual_reach.py train-pixel --seed 0
bash docker/dev.sh python scripts/ch10_visual_reach.py eval --ckpt checkpoints/sac_pixel_F
```

FetchReachDense-v4 -- Pixel SAC (2M steps):

Metric	Value
Success rate	98.0%
Return (mean)	-1.022 +/- 0.503
Final distance (mean)	0.0156
Training time	4.0 hrs (14,362s)
FPS	139

Interpretation: Pixel SAC needs significantly more samples than state SAC because the NatureCNN must first learn to extract spatial features (end-effector position, target marker) from raw pixels before the RL algorithm can learn a useful policy. The Q-function also tends to overfit to pixel-level details -- watch for `critic_loss` that decreases early but climbs later, which is the signature of this failure mode.

1.5.4 10.10.1 Experiment 3: Pixel SAC + DrQ

```
bash docker/dev.sh python scripts/ch10_visual_reach.py train-pixel-drq --seed 0
bash docker/dev.sh python scripts/ch10_visual_reach.py eval --ckpt checkpoints/sac_drq_Fet
```

FetchReachDense-v4 -- Pixel+DrQ SAC (2M steps):

Metric	Value
Success rate	100.0%
Return (mean)	-0.889 +/- 0.516
Final distance (mean)	0.0104
Training time	4.5 hrs (16,211s)
FPS	123

Interpretation: DrQ improves pixel SAC's sample efficiency by regularizing the Q-function through random shift augmentation, which prevents overfitting to pixel-level details and partially closes the performance gap to state SAC.

1.5.5 10.10.2 Three-Way Comparison

```
bash docker/dev.sh python scripts/ch10_visual_reach.py compare
```

FetchReachDense-v4 -- State vs Pixel vs Pixel+DrQ:

Metric	State	Pixel	Pixel+DrQ
Training steps	500,000	2,000,000	2,000,000
Success rate	100.0%	98.0%	100.0%
Return (mean)	-0.730 +/- 0.245	-1.022 +/- 0.503	-0.889 +/- 0.516
Final distance	0.0116	0.0156	0.0104
FPS	391	139	123

Pixel SAC requires 4x more training steps (2M vs 500K) and runs at ~2.8x lower throughput (139 vs 391 FPS). DrQ closes 100% of the success rate gap (98% -> 100%) and actually achieves the lowest final distance of all three agents (0.0104 vs 0.0116 for state). The return gap closes by ~46%: pixel SAC's return of -1.022 improves to -0.889 with DrQ, compared to state SAC's -0.730.

1.5.6 10.10.3 The ”Too Easy” Problem

FetchReachDense is a single-phase task -- move the end-effector to a target -- so all three agents converge to near-perfect success (98-100%), and the interesting signal is **convergence speed**, not final performance. We saw this same pattern in Chapter 4 with sparse Reach, where the task was solved before algorithmic differences became dramatic.

This makes Reach a useful sanity check (”does pixel SAC work at all?”), but it understates the real cost of learning from pixels. For a sharper test, we need a task where the agent must interact with an object in the scene.

1.5.7 10.10.4 Extension: Why Push Needs More Than Dense Rewards

FetchPushDense-v4 asks the robot to **push a block** to a target position on the table. The dense reward is $R = -\|\text{object\_pos} - \text{target\_pos}\|$.

At first glance, this should be a straightforward extension of Reach: same pipeline, different environment, continuous reward signal. But when we train state SAC on FetchPushDense with the same hyperparameters from Reach, something surprising happens:

FetchPushDense-v4 -- State SAC (1M steps, ent_coef=”auto”, gamma=0.95):

Metric	Value
Success rate	2%
Return (mean)	-9.69 +/- 3.75
Final distance (mean)	0.194
Training time	44.8 min (2689s)
FPS	372

Only **2%** success rate from privileged state vectors. Compare this to 100% on Reach at 500K steps. The same hyperparameters that solve Reach in minutes fail completely on Push.

The "deceptively dense" reward. FetchPushDense's reward is continuous but not uniformly informative. In FetchReachDense, where $R = -\|\text{gripper\_pos} - \text{target\_pos}\|$, every gripper movement changes the reward, so the gradient is always informative. In FetchPushDense, by contrast, $R = -\|\text{object\_pos} - \text{target\_pos}\|$ -- the reward only changes when the *object* moves, which means the agent can move freely before contact without affecting the reward at all.

This asymmetry creates a two-phase reward landscape:

1. **Pre-contact (steps 1-15):** Reward is CONSTANT. The agent receives no signal that approaching the block is useful. Random exploration must discover contact.
2. **Post-contact (steps 15+):** Reward responds to actions. Now the agent can learn to push in the right direction.

FetchPushDense is therefore a **sparse exploration problem disguised as a dense reward** -- the "dense" label refers to the reward's continuity *given contact*, not its informational content across the full episode. This is exactly the problem HER (Chapter 4) solves, since by relabeling goals to where the block already is, even accidental non-pushing becomes a successful episode.

Connection to the "charging a phone" aspiration. FetchPickAndPlace compounds this difficulty because the agent must not only contact the object but also close the gripper, lift, transport, and place -- each phase layering a new exploration challenge on top of the previous one. Without HER (or curriculum learning, or reward shaping), the probability of randomly discovering the full sequence from pixels is vanishingly small.

This finding confirms the curriculum's arc. In Ch3, SAC solved dense Reach out of the box; in Ch4, HER made sparse rewards tractable for Reach and Push; and here in Ch10, we see that pixel observations add cost but do not block learning on easy tasks. The natural next step is visual HER for Push and PickAndPlace, where both the observation and exploration problems are hard simultaneously.

1.6 10.11 Making It Fast: Rendering Is the Bottleneck

The pixel training pipeline above takes ~8.5 hours per arm (pixel + DrQ = ~17 hours total), which raises a natural question: where does that time go, and how much of it can we eliminate?

1.6.1 10.11.1 Profiling the Pipeline

In RL with physics simulators, the **GPU sits idle** most of the time because the bottleneck is CPU-bound: MuJoCo runs contact dynamics, then renders the scene to an image, then Python (PIL) resizes it -- all sequentially on the CPU -- while the neural network forward/backward pass on the GPU completes in microseconds.

Per-step cost breakdown for pixel training without `--fast`:

Component	Time per step	Notes
MuJoCo simulation	~0.5 ms	CPU-bound physics
Render at 480x480	~4 ms	230,400 pixels
PIL resize to 84x84	~0.3 ms	Python/C interop
Neural network update	~0.1 ms	GPU, batch=256

Rendering dominates -- but 480x480 does not *sound* large, so why does it matter? The answer lies in what we might call **the inner-loop multiplier effect**: any per-step cost, no matter how small it looks in isolation, gets multiplied by millions of steps, so the right question is not "how long does one render take?" but "how long do 2 million renders take?"

Resolution	Pixels	Render time (per frame)	Over 2M steps	Data per frame
480 x 480	230,400	~4 ms	**2.2 hours**	691 KB
84 x 84	7,056	~0.5 ms	**17 minutes**	21 KB
Savings	**33x fewer**	**~3.5 ms**	**~2 hours**	**670 KB**

That is roughly a quarter of the total 8.5-hour training time spent drawing pixels we were about to discard via PIL resize. Adding the resize itself (~0.3 ms/frame x 2M = another 10 minutes) and the Python-to-C data copy of 691 KB per frame, the "small" rendering overhead accounts for over 2.5 hours of wall time.

The general principle here is to **optimize by elimination (don't do the work) rather than acceleration (do the work faster).** Rendering 33x fewer pixels is not "faster rendering" -- it is "less rendering," since the default 480x480 pipeline generates data, ships it through two libraries, and then throws away 97% of it. The fast path avoids generating that data in the first place.

1.6.2 10.11.2 Three Optimizations

The `--fast` flag bundles three independent optimizations that compound multiplicatively, since each addresses a different part of the pipeline:

Factor	Default	`--fast`	Effect
Pixels rendered per frame	230,400 (480x480)	7,056 (84x84)	**33x** fewer pixels
PIL resize overhead	~0.3 ms/frame	**skipped**	eliminated entirely
Env parallelism	4 sequential (DummyVec)	12 parallel (SubprocVec)	**3x** workers
Gradient steps per env step	1	3	**3x** learning per sample

Optimization 1: Native Resolution Rendering. Instead of rendering at 480x480 and resizing, we tell MuJoCo to render directly at 84x84:

```
env = gym.make("FetchReachDense-v4", render_mode="rgb_array", width=84, height=84)
```

This renders 33x fewer pixels (7,056 vs 230,400) and skips PIL entirely, since the `render_and_resize` function detects the matching size and takes the fast path automatically.

Optimization 2: SubprocVecEnv. `DummyVecEnv` (the default) runs all environments sequentially in the main process, whereas `SubprocVecEnv` runs each environment in a separate process, parallelizing MuJoCo rendering across CPU cores:

```
from stable_baselines3.common.vec_env import SubprocVecEnv
env = make_vec_env(make_pixel_env, n_envs=12, vec_env_cls=SubprocVecEnv)
```

With 12 workers, we get ~3x throughput on a multi-core DGX.

Optimization 3: More Gradient Steps. When data arrives faster (thanks to more envs and faster rendering), we can afford more gradient updates per environment step -- `gradient_steps=3` means 3 SAC updates per step, which increases learning per sample without stale-data concerns. This is the **replay ratio** concept: faster data collection enables a higher ratio of learning to experience collection.

1.6.3 10.11.3 Using `--fast`

```
# Fast mode with default settings (n_envs=12, gradient_steps=3, native render)
bash docker/dev.sh python scripts/ch10_visual_reach.py train-pixel --seed 0 --fast

# Override specific values while keeping other fast defaults
bash docker/dev.sh python scripts/ch10_visual_reach.py train-pixel --seed 0 --fast --pixel
```

`--fast` sets: `native_render=True`, `use_subproc=True`, `pixel_n_envs=12`, `gradient_steps=3`. Explicit CLI arguments override fast defaults.

What stays the same: The algorithm, architecture, hyperparameters, and final performance are all unchanged -- `--fast` only optimizes the engineering of data collection, which means a model trained with `--fast` should be statistically equivalent to one trained without it.

1.6.4 10.11.4 Speedup Results

Setting	FPS	Wall time (2M steps)	Notes
Default (4 env, DummyVec, PIL resize)	~65	~8.5 hrs	Pedagogical pipeline
`--fast` (12 env, SubprocVec, native 84x84)	139	4.0 hrs	gradient_steps=3

The `--fast` pipeline achieved 139 FPS and completed 2M steps in 4.0 hours -- a ~2.1x speedup over the estimated ~65 FPS / ~8.5 hrs default pipeline. The fast-mode metadata is recorded in the checkpoint's `.meta.json` file for reproducibility, so that any results can be traced back to the exact configuration used:

```
{
  "fast_mode": true,
  "native_render": true,
  "gradient_steps": 3,
  "use_subproc": true,
  "n_envs": 12
}
```

1.7 Summary

1.7.1 Key Findings

1. **State SAC solves FetchReachDense** in 500K steps (100.0% success rate, return -0.730 +/- 0.245, 21.3 min wall time).
2. **Pixel SAC is harder** -- 98.0% success rate at 2M steps (4x more samples), with return -1.022 +/- 0.503 and final distance 0.0156 (vs 0.0116 for state).

3. **DrQ augmentation closes the gap** -- 100.0% success rate (matching state), return -0.889 +/- 0.516, final distance 0.0104 (actually *lower* than state's 0.0116). Random shift regularization eliminates Q-function overfitting.
4. **Rendering is the bottleneck** -- `--fast` mode achieved 139 FPS (vs ~65 default), completing 2M pixel steps in 4.0 hrs instead of ~8.5 hrs (~2.1x speedup).

1.7.2 Concepts Introduced in This Chapter

Concept	Definition
Pixel observation wrapper	Replaces flat state vector with rendered 84x84 RGB images
Goal mode (none/desired/both)	Controls which goal vectors are exposed alongside pixels
NatureCNN	CNN encoder from Mnih et al. (2015): three conv layers + FC to 512D fe
Sample-efficiency ratio	$\rho = N_{\text{pixel}}/N_{\text{state}}$ -- how many more samples pixels need
DrQ (random shift augmentation)	Pad-and-crop pixel augmentation at replay buffer sample time
Native resolution rendering	`gym.make(..., width=84, height=84)` to skip PIL resize
SubprocVecEnv	Parallel environment execution across CPU cores
Replay ratio / gradient steps	Number of SAC gradient updates per environment step

1.7.3 Files Generated

File	Purpose
`scripts/labs/pixel_wrapper.py`	PixelObservationWrapper, render_and_resize, PixelReplayBuffer
`scripts/labs/visual_encoder.py`	NatureCNN, VisualGoalEncoder, VisualGaussianPolicy, VisualTwin
`scripts/labs/image_augmentation.py`	RandomShiftAug, DrQDictReplayBuffer
`scripts/ch10_visual_reach.py`	Chapter script: train-state, train-pixel, train-pixel-drq, eval, comp

1.7.4 Artifacts

Artifact	Location
State SAC checkpoint	`checkpoints/sac_state_FetchReachDense-v4_seed{N}.zip`
Pixel SAC checkpoint	`checkpoints/sac_pixel_FetchReachDense-v4_seed{N}.zip`
Pixel+DrQ checkpoint	`checkpoints/sac_drq_FetchReachDense-v4_seed{N}.zip`
Training metadata	`checkpoints/sac_{mode}_FetchReachDense-v4_seed{N}.meta.json`
State evaluation	`results/ch10_state_eval.json`
Pixel evaluation	`results/ch10_pixel_eval.json`
DrQ evaluation	`results/ch10_drq_eval.json`
Comparison report	`results/ch10_comparison.json`

1.7.5 What Comes Next

This chapter showed the cost of removing privileged state information: pixel observations are harder, slower, and prone to Q-function overfitting. DrQ helps, but a gap remains -- both in return and in wall-clock time.

Why FetchReachDense is "too easy." We saw this pattern before in Chapter 4: FetchReach is a single-phase task (move the end-effector to a point in space), so both state and pixel agents converge to near-perfect success, and the interesting signal is convergence *speed*, not final performance. The sample- efficiency ratio ρ is measurable but modest.

For a more dramatic demonstration of the pixel penalty, we want a task that requires **object interaction** -- where the agent must learn to manipulate something in the scene, not just move through free space. Two natural candidates from the Fetch suite:

Task	What the robot does	Difficulty from pixels
FetchPushDense	Push a block to a goal position	Medium -- requires learning object dynamics
FetchPickAndPlaceDense	Grasp, lift, and place an object	Hard -- multi-phase (approach, grasp, lift, tra

FetchPushDense from pixels works with our existing pipeline: dense rewards, no HER, SAC with pixel observations. The robot must discover that pushing the block reduces the reward -- something that requires understanding scene geometry from images alone, not just proprioception. We expect the pixel-state gap to be significantly larger than on Reach.

The "charging a phone" problem. FetchPickAndPlace is the manipulation task closest to a practical application -- pick up an object and place it at a target location, much like a robot arm picking up a phone and placing it on a charging pad -- which makes it fetch-and-place in its core.

But PickAndPlace from pure pixels exposes a fundamental tension between observation modality and exploration strategy.

Dense rewards alone may not be enough. The agent must discover the full grasp-lift-transport-place sequence, and even with continuous distance feedback, the probability of randomly stumbling into a successful grasp (closing the gripper around the object at exactly the right position and orientation) is extremely low. Without the relabeling trick, this is a severe exploration problem. The natural solution is HER (Chapter 4), which transforms failures into successes by asking "what goal *would* this trajectory have achieved?" -- but HER's relabeling requires `achieved_goal` and `desired_goal` vectors, the 3D positions that our pixel-only wrapper (`goal_mode="none"`) deliberately strips from the observation.

The middle ground is visual HER. Our pixel wrapper supports `goal_mode="both"`, which provides pixel observations alongside goal vectors, so that the policy sees pixels through NatureCNN while HER uses the goal vectors internally for relabeling. The policy does not "cheat" on state information (it still learns spatial features from images), but it does receive the target position as a privileged vector. This is the approach used by Nair et al. (2018) for visual goal-conditioned RL.

Visual HER -- training from pixels with sparse rewards and goal relabeling -- is the natural next step beyond this chapter. It combines the pixel observation pipeline from Ch10 with the HER machinery from Ch4, creating agents that can solve multi-phase manipulation tasks like PickAndPlace from camera images. We leave this as the direction for future work.

1.7.6 References

- Mnih, V. et al. (2015). "Human-level control through deep reinforcement learning." *Nature*, 518(7540), 529-533.
- Kostrikov, I. et al. (2020). "Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels." arXiv:2004.13649, Section 3.1.
- Haarnoja, T. et al. (2018). "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor." arXiv:1801.01290.
- Nair, A. et al. (2018). "Visual Reinforcement Learning with Imagined Goals." NeurIPS 2018. arXiv:1807.04742.