9 Pixels, No Cheating: From State Vectors to Camera Images

Vlad Prytula

2026-02-28

Contents

9 Pixels, No Cheating: From State Vectors to Camera Images

This chapter covers:

- Building a complete visual observation pipeline from scratch -- pixel wrappers, CNN encoders, spatial feature extraction, and memory-efficient replay buffers -- that turns 84x84 camera images into features a policy can act on
- Understanding why NatureCNN fails on manipulation tasks (stride-4 destroys 5-pixel objects) and implementing ManipulationCNN + SpatialSoftmax as the fix
- Discovering why architecture alone is not enough: SB3's default gradient routing starves the encoder of learning signal, and the DrQ-v2 pattern (encoder in the critic's optimizer) is the 15 lines that make the difference between 5% and 95%
- Reading the three-phase loss signature in pixel RL -- declining losses with flat success means the critic is memorizing failure, not learning structure; rising losses during the hockey-stick means real value learning has begun
- Running a 5-step progressive investigation that arrives at a pixel Push agent achieving 95%+ success, with each failure teaching a transferable debugging principle

Through Chapters 3-8, you built SAC from scratch, added HER for sparse rewards, solved Push at 89% from state vectors, tested robustness under noise, and learned to tune hyperparameters systematically. You have a working, validated, robust manipulation pipeline -- but it assumes the agent directly observes state vectors: joint positions, object coordinates, goal positions.

Real robots do not observe state vectors. They see pixels from cameras. Can the pipeline you built survive the switch from 25 numbers to 84x84 images? The answer is: not without significant changes to the encoder, the gradient routing, and the training budget. This chapter adds a visual observation pipeline (pixel wrappers, CNN encoders, spatial feature extraction), the gradient routing that makes the encoder actually learn (the critic-encoder pattern from DrQ-v2), and the diagnostic skills to read pixel RL training curves. You will discover these through a 5-step investigation: try the obvious thing, watch it fail, diagnose why, fix one component at a time, and arrive at a working solution. This chapter achieves 95% Push from pixels, matching state-based performance at the cost of 2-4x more training steps. Chapter 10 will explore how pre-trained encoders and world models can reduce this overhead.

9.1 WHY: The pixel penalty

What changes

State-based Push uses a 25-dimensional vector: 3D gripper position, 3D object position, 3D relative position, 2D gripper state, 3D object rotation, and various velocities. Every number is precise to millimeter accuracy, calibrated, and cheap to process -- a two-layer MLP (256 x 256 parameters) handles it in microseconds, and the GPU sits idle at 5-10% utilization because the bottleneck is CPU-side MuJoCo simulation.

Pixel Push replaces most of that with a camera image: 84 x 84 x 3 = 21,168 values per frame, stacked 4 deep for temporal information, producing 84 x 84 x 12 = 84,672 values per observation. These values are raw pixel intensities, uncalibrated, and expensive to process through a convolutional neural network. GPU utilization rises to 40-60% because the CNN processes batches of pixel observations for both `obs` and `next_obs` on every training step. Training throughput drops from roughly 600 fps (state-based) to 30-50 fps (pixel-based). Figure 9.1 shows what the policy actually sees.

An 84x84 pixel observation from FetchPush-v4 showing the gripper (approximately 4 pixels wide), the puck (approximately 5 pixels wide), and the goal marker on the table surface

Figure 9.1: FetchPush-v4 at 84x84 resolution -- what the pixel agent sees. The puck is roughly 5 pixels wide and the gripper roughly 4 pixels wide. The spatial relationship between them -- the one thing the policy needs to act on -- occupies a tiny fraction of the image. (Generated by `bash docker/dev.sh python scripts/ch09_pixel_push.py render-frame --seed 0`.)

Four compounding challenges

Every pixel RL challenge from earlier chapters compounds when we add object interaction:

1. Tiny objects. The puck in FetchPush is roughly 5 pixels wide at 84x84 resolution, and the gripper is roughly 4 pixels, so the spatial relationship between them -- the signal the policy needs to act on -- occupies a tiny fraction of the image.

2. Sparse rewards plus pixels. FetchReachDense gave continuous distance feedback, meaning every arm movement changed the reward. 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.

3. Two learning problems at once. The agent must simultaneously learn visual representations (CNN: pixels to spatial features) and a control policy (actor-critic: spatial features to 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 for both.

4. Contact dynamics from images. Pushing requires understanding what happens after contact -- does the puck move in the right direction, and how far? This temporal reasoning must be inferred from sequences of pixel observations. Frame stacking (4 frames, concatenated along the channel dimension) provides a weak velocity signal, since pixel differences between consecutive frames imply motion direction and speed. But the motion signal is subtle at 84x84: a puck moving 1 cm per timestep shifts by roughly 1 pixel. Without frame stacking, the environment becomes a POMDP (partially observable MDP), because a single static image cannot distinguish ”puck moving left” from ”puck moving right.”

The observation design space

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. This mirrors how real robotic systems operate (joint encoders plus cameras, never cameras alone), and we call it the **sensor separation principle**.

Visual HER: two kinds

”Visual HER” can mean two very different architectures:

Approach	Policy sees	HER relabels	Com
Our approach (`goal_mode="both"`)	Pixels + goal vectors	3D goal vectors (swap)	Mod
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`). Relabeling is a swap of two 3D vectors. No image-space goal representation is 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. We are testing whether a CNN can learn spatial features good enough for manipulation, not whether it can learn a visual goal representation. Full visual HER (Nair et al., 2018) tackles the harder problem of specifying goals as images, requiring a VAE to map images to a latent goal space where relabeling is meaningful. That is a worthwhile problem, but it adds an entire representation learning subsystem on top of what is already a challenging pixel RL task. We scope this chapter to the first approach: prove the CNN can learn to control, using vector goals as scaffolding.

Hadamard check

Before investing 40+ hours of GPU time per seed (at 30-50 fps, 5M steps takes 28-46 hours), we check our three practical questions. **Can this be solved?** Yes -- we have the 89% state-based result as existence proof, and the information is visually present in the image (you can see the puck and gripper), so the question reduces to whether a CNN can extract it. **Is the solution reliable?** We will need 3 seeds to find out, since one seed at 95% could be lucky. **Is the solution stable?** Prior experience with pixel RL says no -- small changes in hyperparameters, encoder architecture, or gradient routing can mean the difference between 95% and 5%. Henderson et al. (2018) showed that even state-based RL exhibits high variance across seeds and implementations, and adding a CNN encoder multiplies the sensitivity surface. This chapter maps that sensitivity by systematically varying components and observing their impact.

9.2 Build It: The visual observation pipeline

We now build the components that convert raw camera frames into training data. The visual pipeline has three parts: a rendering function that captures camera images, a wrapper that integrates them into Gymnasium's observation structure with frame stacking and proprioception, and a replay buffer that stores pixel transitions without exhausting system memory. Each component is individually testable, so we verify shapes and types before connecting them.

9.2.1 Rendering and resizing

The rendering function captures a MuJoCo camera frame and converts it to a CHW uint8 tensor that PyTorch and SB3 expect. MuJoCo's `render()` returns an HWC array (height x width x channels) -- the image format used by NumPy and PIL -- but PyTorch and SB3 expect CHW format (channels first), so we transpose. This transpose is a zero-cost memory reinterpretation, not a data copy.

MuJoCo renders at its default resolution (480x480 for Fetch), and we resize to 84x84 via PIL bilinear interpolation. When you create the environment with `gym.make("FetchPush-v4", render_mode="rgb_array", width=84, height=84)`, MuJoCo renders directly at 84x84 and the resize step is skipped entirely -- a worthwhile optimization since rendering happens every step.

Listing 9.1: render_and_resize -- camera capture to CHW tensor

```
def render_and_resize(
    env: gym.Env,
    image_size: tuple[int, int] = (84, 84),
) -> np.ndarray:
    """Render MuJoCo scene -> CHW uint8 array (3, H, W)."""
    frame = env.render()               # HWC uint8 (480x480x3)
    assert frame is not None, "render_mode must be 'rgb_array'"

    # Fast path: already at target size (native rendering)
    if frame.shape[:2] == image_size:
        return frame.transpose(2, 0, 1).copy()

    # Resize via PIL bilinear interpolation
    from PIL import Image
    pil_img = Image.fromarray(frame)
    pil_img = pil_img.resize(
        (image_size[1], image_size[0]), Image.Resampling.BILINEAR
    )
    return np.array(pil_img, dtype=np.uint8).transpose(2, 0, 1)
```

The `.copy()` on the fast path matters -- MuJoCo may reuse the internal render buffer on the next call, so we need the array to own its data.

> **Checkpoint:** The output should be `(3, 84, 84)` with dtype `uint8` and values in `[0, 255]`. If the image is all black, check that `render_mode="rgb_array"` was passed to `gym.make()`.

9.2.2 PixelObservationWrapper

The wrapper replaces Gymnasium's flat `"observation"` key with a `"pixels"` key, optionally stacks frames for temporal information, and passes through proprioception and goal vectors.

Listing 9.2: PixelObservationWrapper -- core observation method

```
class PixelObservationWrapper(gym.ObservationWrapper):
    def observation(self, observation):
        self.last_raw_obs = observation
        pixels = render_and_resize(self.env, self._image_size)

        if self._frame_stack > 1:
            pixels = self._get_stacked_pixels(pixels)

        out = {"pixels": pixels}
        if self._proprio_indices is not None:
            out["proprioception"] = (
                observation["observation"][self._proprio_indices]
            )
        if self._goal_mode in {"desired", "both"}:
            out["desired_goal"] = observation["desired_goal"]
        if self._goal_mode == "both":
            out["achieved_goal"] = observation["achieved_goal"]
        return out
```

Frame stacking concatenates the last N frames along the channel dimension: 4 frames of RGB produce `(12, 84, 84)`. On reset, the deque is filled with copies of the first frame so the stack is always complete -- this avoids a cold-start artifact where the first few observations would otherwise contain stale frames from a previous episode.

The `last_raw_obs` attribute stores the unwrapped Gymnasium observation for debugging. When you need to check the ground-truth object position or goal distance, `wrapper.last_raw_obs["achieved_goal"]` gives you the 3D coordinates without parsing the pixel observation.

> **Checkpoint:** Run `bash docker/dev.sh python scripts/labs/pixel_wrapper.py --verify`. Expected: `[ALL PASS] Pixel wrapper verified.` Verify the output observation dict has keys `pixels` (12, 84, 84), `proprioception` (10,), `achieved_goal` (3,), `desired_goal` (3,).

9.2.3 Pixel replay buffer with uint8 storage

Pixel observations are expensive to store. Each frame is `12 x 84 x 84 = 84,672` bytes as uint8. Each transition stores both obs and next_obs, so:

$$\text{bytes per transition} = 84{,}672 \times 2 = 169{,}344 \approx 165 \text{ KB}$$

> **Sidebar: The memory wall.** A 500,000-transition pixel buffer costs $500{,}000 \times 169{,}344 \approx 80$ GB for pixel arrays alone. Adding proprioception, goals, actions, and rewards brings the total to roughly 85 GB. On our DGX with 119 GB RAM, this leaves about 34 GB for the OS, CUDA, and the model -- tight but feasible. A 1M buffer would need 170 GB and is physically impossible on this machine. For comparison, a 500K state-based buffer uses about 250 MB. Pixels cost 340x more per transition.

What if you don't have 120 GB? Smaller buffers work, at the cost of delayed or weaker convergence. The buffer retains early diverse exploration that HER relabels into learning signal; when old episodes are overwritten before their value propagates through the Bellman equation, the hockey-stick ignites later or stalls at a lower plateau.

Buffer size	Buffer RAM	Total needed	Expected effect
500K	~80 GB	~85-90 GB	Full performance; hockey-stick at ~2.2M steps
300K	~48 GB	~53-58 GB	Hockey-stick may shift to ~3M; final success 90%+
200K	~32 GB	~37-42 GB	Slower convergence; may need 5M+ steps for 90%

Buffer size	Buffer RAM	Total needed	Expected effect
100K	~16 GB	~21-24 GB	May plateau at 70-85%; consider `--full-state` first

Readers with 64 GB should use `--buffer-size 300000`. Readers with 32 GB should use `--buffer-size 100000`. Below 32 GB, pixel Push training is experimental -- verify the pipeline with `--full-state` first, then try pixels with `--buffer-size 50000`.

The key insight is to store pixels as uint8 (1 byte per value) and convert to float32 only at sample time. The conversion cost is negligible for a 256-sample batch (256 x 84,672 x 4 bytes = 87 MB for obs + next_obs) but would quadruple memory if applied to the full buffer. SB3's `DictReplayBuffer` already stores pixels as uint8 when the observation space dtype is `np.uint8`, so our wrapper defines the pixel space with `dtype=np.uint8` to ensure this behavior. One caveat: SB3's `DictReplayBuffer` does NOT support `optimize_memory_usage=True` (which would avoid storing next_obs separately for a 2x savings), raising `ValueError` if you try, so we live with the full obs + next_obs cost.

Listing 9.3: PixelReplayBuffer -- uint8 storage, float32 sampling

```python
class PixelReplayBuffer:
    def __init__(self, img_shape, goal_dim, act_dim, capacity):
        # uint8 storage: 1 byte/pixel (not 4 bytes for float32)
        self.pixels = np.zeros((capacity, *img_shape), dtype=np.uint8)
        self.next_pixels = np.zeros((capacity, *img_shape), dtype=np.uint8)
        # Goals, actions, rewards are small -- float32 is fine
        self.goals = np.zeros((capacity, goal_dim), dtype=np.float32)
        self.achieved = np.zeros((capacity, goal_dim), dtype=np.float32)
        self.actions = np.zeros((capacity, act_dim), dtype=np.float32)
        self.rewards = np.zeros((capacity, 1), dtype=np.float32)
        self.dones = np.zeros((capacity, 1), dtype=np.float32)
        self.capacity, self.size, self.pos = capacity, 0, 0

    def add(self, obs_pixels, next_pixels, achieved, goal, action, reward, done):
        self.pixels[self.pos] = obs_pixels          # store uint8 directly
        self.next_pixels[self.pos] = next_pixels
        self.goals[self.pos] = goal
        self.achieved[self.pos] = achieved
        self.actions[self.pos] = action
        self.rewards[self.pos] = reward
        self.dones[self.pos] = done
        self.pos = (self.pos + 1) % self.capacity
        self.size = min(self.size + 1, self.capacity)

    def sample(self, batch_size):
        idx = np.random.randint(0, self.size, size=batch_size)
        return {
            # Convert to float32 [0, 1] at sample time only
            "pixels": self.pixels[idx].astype(np.float32) / 255.0,
            "next_pixels": self.next_pixels[idx].astype(np.float32) / 255.0,
            "achieved_goal": self.achieved[idx],
            "desired_goal": self.goals[idx],
            "actions": self.actions[idx],
            "rewards": self.rewards[idx],
            "dones": self.dones[idx],
        }
```

This is the complete from-scratch implementation. The core insight is in the dtype asymmetry: `__init__` allocates `uint8` for pixels (1 byte each), while `sample` converts to `float32` (4 bytes each). The conversion cost is negligible for a 256-sample batch but would quadruple memory if applied to the full 500K-transition buffer. SB3's `DictReplayBuffer` implements this same pattern automatically when the observation space dtype is `np.uint8`. In Run It, we use SB3's production buffer for its integration with vectorized environments and HER -- but the storage principle is identical to what you see here.

> **Checkpoint:** Create a buffer with `capacity=100`, add 50 transitions, sample a batch of 16. The internal `.pixels` array should have dtype `uint8`; the sampled batch should have dtype `float32` with values in `[0, 1]`.

9.3 Build It: Encoder architecture

The encoder is where pixel observations become features a policy can act on. We build two encoders -- NatureCNN (the ”wrong” one, to understand why it fails) and ManipulationCNN (the ”right” one) -- then add SpatialSoftmax to extract spatial coordinates and ManipulationExtractor to wire everything together for SB3.

9.3.1 NatureCNN -- the ”wrong” encoder

NatureCNN (Mnih et al., 2015) is the default visual encoder in SB3 and was the workhorse of the Atari RL era. Its architecture uses large kernels with aggressive downsampling:

```
NatureCNN spatial progression (84x84 input):
  Layer 1: Conv2d(C, 32, 8x8, stride=4)   84 -> 20   (4x reduction!)
  Layer 2: Conv2d(32, 64, 4x4, stride=2)   20 ->  9
  Layer 3: Conv2d(64, 64, 3x3, stride=1)    9 ->  7
  Flatten -> Linear(3136, 512)
```

Listing 9.4: NatureCNN -- Atari-era encoder (from `scripts/labs/visual_encoder.py:nature`

```python
class NatureCNN(nn.Module):
    def __init__(self, in_channels=3, features_dim=512):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Flatten(),
        )
        with torch.no_grad():
            n_flat = self.conv(torch.zeros(1, in_channels, 84, 84)).shape[1]
        self.fc = nn.Sequential(
            nn.Linear(n_flat, features_dim), nn.ReLU()
        )
```

The problem is in the first layer. The puck is roughly 5 pixels wide, so after stride-4 it becomes roughly 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 wide and decisions are coarse (”go left” vs ”go right”). Manipulation requires millimeter-precision spatial reasoning about objects that are 3-5 pixels wide, making the architecture fundamentally mismatched.

In our experiments, NatureCNN achieves 5-8% success on FetchPush across 2M+ training steps -- indistinguishable from the random-policy baseline. Since the algorithm (SAC + HER) is proven to work from state at 89%, the encoder is the bottleneck.

9.3.2 ManipulationCNN -- the ”right” encoder

The fix is gentle downsampling: 3x3 kernels throughout, stride 2 only in layers 1 and 4, padding to preserve resolution where possible. This follows the DrQ-v2 encoder design (Yarats et al., 2021).

```
ManipulationCNN spatial progression (84x84 input):
  Layer 1: Conv2d(C, 32, 3x3, stride=2, pad=1)   84 -> 42   (2x, not 4x)
  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 roughly 3 pixels -- still wide enough to carry spatial information. After all four layers, the puck occupies roughly 1-2 pixels on the 21x21 feature map. The spatial relationship between gripper and puck is preserved throughout the network, and a subsequent SpatialSoftmax layer (Section 9.3.3) can extract precise coordinates from these activations.

Listing 9.5: ManipulationCNN -- gentle downsampling for small objects

```
class ManipulationCNN(nn.Module):
    def __init__(self, in_channels=12, num_filters=32):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, num_filters, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(num_filters, num_filters, 3, stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(num_filters, num_filters, 3, stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(num_filters, num_filters, 3, stride=2, padding=1),
            nn.ReLU(),
        )

    def forward(self, pixels):
        """(B, C, 84, 84) float32 [0,1] -> (B, 32, 21, 21) feature map."""
        return self.conv(pixels)
```

Figure 9.2 illustrates the difference side by side.

Side-by-side comparison of NatureCNN and ManipulationCNN spatial progression. NatureCNN: 84 to 20 to 9 to 7 pixels (puck goes from 5px to 1px). ManipulationCNN: 84 to 42 to 42 to 42 to 21 pixels (puck goes from 5px to 3px)

Figure 9.2: NatureCNN vs ManipulationCNN spatial progression. A 5-pixel puck becomes roughly 1 pixel after NatureCNN's stride-4 first layer (left), but survives as roughly 3 pixels through ManipulationCNN's stride-2 downsampling (right). The spatial relationship between gripper and puck -- the critical signal for Push -- is preserved in ManipulationCNN and destroyed in NatureCNN. (Generated by `bash docker/dev.sh python scripts/labs/manipulation_encoder.py --compare-nature`.)

> **Checkpoint:** Feed a `(4, 12, 84, 84)` tensor through ManipulationCNN. Output shape should be `(4, 32, 21, 21)`, all values finite.

9.3.3 SpatialSoftmax -- "where" not "what"

ManipulationCNN produces a `(B, 32, 21, 21)` feature map -- 32 channels, each 21x21 pixels. The standard approach would flatten this into a 14,112-dimensional vector, but for manipulation we do not need to know what the image looks like; we need to know WHERE things are.

SpatialSoftmax (Levine et al., 2016) extracts the expected (x, y) coordinate of each channel's activation peak. For C channels, the output is $2C$ values in $[-1, 1]$ -- spatial coordinates, not pixel values.

The operation per channel with height H and width W:

1. Softmax over all $H \times W$ spatial positions: $\alpha_{h,w} = \text{softmax}(f_{h,w}/\tau)$ where τ is a learnable temperature
2. Expected x-coordinate: $\bar{x} = \sum_{h,w} \alpha_{h,w} \cdot \text{pos}_x(w)$, where $\text{pos}_x \in [-1, 1]$
3. Expected y-coordinate: $\bar{y} = \sum_{h,w} \alpha_{h,w} \cdot \text{pos}_y(h)$

The temperature τ is learnable: a high temperature produces uniform attention early in training (when the network is unsure where to look), while a low temperature produces peaked attention once the policy converges (focusing on precise object locations).

Listing 9.6: SpatialSoftmax -- expected (x, y) coordinates per channel

```
class SpatialSoftmax(nn.Module):
    def __init__(self, height, width, num_channels, temperature=1.0):
        super().__init__()
        self.temperature = nn.Parameter(torch.ones(1) * temperature)
        pos_x = torch.linspace(-1.0, 1.0, width)
        pos_y = torch.linspace(-1.0, 1.0, height)
        self.register_buffer("pos_x", pos_x.reshape(1, 1, -1))
        self.register_buffer("pos_y", pos_y.reshape(1, 1, -1))

    def forward(self, features):
        B, C, H, W = features.shape
        attn = F.softmax(
            features.reshape(B, C, -1) / self.temperature, dim=-1
        ).reshape(B, C, H, W)
        exp_x = (attn.sum(dim=2) * self.pos_x).sum(dim=-1)
        exp_y = (attn.sum(dim=3) * self.pos_y).sum(dim=-1)
        return torch.cat([exp_x, exp_y], dim=-1)  # (B, 2C)
```

With 32 channels, the output is 64 values: 32 x-coordinates and 32 y-coordinates, each in $[-1, 1]$. This is a powerful inductive bias for manipulation -- the policy needs spatial coordinates (where is the puck? where is the gripper?), not a compressed image representation.

The LayerNorm and Tanh layers that follow SpatialSoftmax in the full pipeline (see Listing 9.7) normalize the coordinate values to a bounded range. This stabilizes gradients early in training when the SpatialSoftmax output might be noisy, and ensures the spatial features have comparable scale to the proprioception and goal vectors they are concatenated with.

> **Checkpoint:** Feed `(4, 32, 21, 21)` random features through SpatialSoftmax. Output shape should be `(4, 64)`. Values should be in `[-1, 1]`. Place a strong activation at position `(0, 0)` (top-left corner) and verify the output coordinates are near `(-1, -1)`.

9.3.4 ManipulationExtractor -- SB3-compatible wiring

SB3 needs a `BaseFeaturesExtractor` subclass that takes a dict observation space and produces a flat feature vector. ManipulationExtractor routes each key to the appropriate sub-encoder: image keys (detected by is_image_space) go through ManipulationCNN + SpatialSoftmax + LayerNorm + Tanh, while vector keys (proprioception, goals) pass through unchanged.

Listing 9.7: ManipulationExtractor -- routing dict observations

```
class ManipulationExtractor(BaseFeaturesExtractor):
    def __init__(self, observation_space, spatial_softmax=True,
                 num_filters=32, flat_features_dim=50):
```

```
        super().__init__(observation_space, features_dim=1)
        extractors, total = {}, 0
        for key, subspace in observation_space.spaces.items():
            if is_image_space(subspace):
                cnn = ManipulationCNN(subspace.shape[0], num_filters)
                with torch.no_grad():
                    feat = cnn(torch.zeros(1, *subspace.shape) / 255.0)
                    _, C, H, W = feat.shape
                if spatial_softmax:
                    extractors[key] = nn.Sequential(
                        cnn, SpatialSoftmax(H, W, C),
                        nn.LayerNorm(2 * C), nn.Tanh(),
                    )
                    total += 2 * C  # 64 for 32 filters
                else:
                    # Flatten + learned projection (DrQ-v2 trunk pattern)
                    flat_size = C * H * W
                    extractors[key] = nn.Sequential(
                        cnn, nn.Flatten(),
                        nn.Linear(flat_size, flat_features_dim),
                        nn.LayerNorm(flat_features_dim), nn.Tanh(),
                    )
                    total += flat_features_dim
            else:
                extractors[key] = nn.Flatten()
                total += gym.spaces.utils.flatdim(subspace)
        self.extractors = nn.ModuleDict(extractors)
        self._features_dim = total
```

For FetchPush with `spatial_softmax=True`, proprioception, and `goal_mode="both"`:

Component	Dimension
Pixels -> CNN -> SpatialSoftmax -> LN -> Tanh	64
Proprioception passthrough	10
achieved_goal passthrough	3
desired_goal passthrough	3
Total features_dim	**80**

9.3.5 Proprioception passthrough and sensor separation

The ManipulationExtractor does not force the CNN to learn about the robot's own body. Joint positions, velocities, and gripper width come from the `"proprioception"` key as a 10D vector that passes through unchanged, so the CNN only sees pixels and learns about the world (object positions, obstacles), not the self.

This is the sensor separation principle in practice: cameras observe the environment while joint encoders observe the robot. Mixing these signals in the CNN wastes network capacity on a problem that cheaper sensors already solve with perfect accuracy. In real robotic systems, proprioception comes from encoders sampling at kHz rates with sub-millimeter resolution -- no camera can compete with that for self-state measurement -- so the CNN should focus on what cameras are uniquely good at: perceiving the world beyond the robot's own body.

> **Checkpoint:** Run `bash docker/dev.sh python scripts/labs/manipulation_encoder.py --verify`. Expected: `[ALL PASS] Manipulation encoder verified.` Key checks: `features_dim = 80` (64 spatial + 10 proprio + 3 ag + 3 dg), SpatialSoftmax coordinates in $[-1, 1]$, ManipulationCNN output `(B, 32, 21, 21)`.

9.4 Build It: Data augmentation

DrQ (Kostrikov et al., 2020) regularizes the Q-function by augmenting pixel observations at sample time from the replay buffer. Each replay of a transition sees a different random crop, creating implicit regularization against Q-function overfitting to pixel-level details.

We build it here because it is a standard technique for pixel RL, used in DrQ, DrQ-v2, CURL, and many other visual RL methods. Whether it helps for manipulation Push with SpatialSoftmax is an empirical question we will answer in Section 9.7 -- the answer will surprise you. Building it now lets us run the controlled comparison later.

9.4.1 DrQ random shift

The augmentation is a pad-and-crop: pad the image by k pixels on all four sides using replicate padding (border pixels extended outward), then randomly crop back to the original size. With $k = 4$ on an 84x84 image, shifts of up to roughly 5% of the image size are possible.

Listing 9.8: RandomShiftAug -- pad-and-crop augmentation

```python
class RandomShiftAug(nn.Module):
    def __init__(self, pad: int = 4):
        super().__init__()
        self.pad = pad

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        B, C, H, W = x.shape
        padded = F.pad(x, (self.pad,)*4, mode="replicate")
        crop_h = torch.randint(0, 2*self.pad + 1, (B,), device=x.device)
        crop_w = torch.randint(0, 2*self.pad + 1, (B,), device=x.device)
        # Index into padded tensor (each image gets independent shift)
        h_idx = torch.arange(H, device=x.device)[None] + crop_h[:, None]
        w_idx = torch.arange(W, device=x.device)[None] + crop_w[:, None]
        b_idx = torch.arange(B, device=x.device)[:, None, None, None]
        c_idx = torch.arange(C, device=x.device)[None, :, None, None]
        return padded[
            b_idx.expand(B,C,H,W), c_idx.expand(B,C,H,W),
            h_idx[:,None,:,None].expand(B,C,H,W),
            w_idx[:,None,None,:].expand(B,C,H,W),
        ]
```

Replicate padding is important because zero padding would create artificial dark borders that the CNN might learn to exploit as a position signal. Replicate padding instead extends the border pixels outward, maintaining the visual appearance of the scene edges, and the shift magnitudes are small enough (± 4 pixels on 84x84, roughly $\pm 5\%$) that the augmented images remain plausible views of the same scene.

> **Checkpoint:** Augment a batch of 8 images twice with the same input. The two outputs should differ (random shifts are independent). Augmenting a constant-valued image should return the same constant (replicate padding of a constant is the same constant).

9.4.2 DrQ replay buffer

The augmentation happens at sample time, not store time. This means each replay of the same transition produces a different view -- the source of DrQ's regularization effect.

Listing 9.9: DrQDictReplayBuffer -- augment at sample time

```python
class DrQDictReplayBuffer(DictReplayBuffer):
    def __init__(self, *args, aug_fn=None, image_key="pixels", **kwargs):
        super().__init__(*args, **kwargs)
        self.aug_fn = aug_fn
        self.image_key = image_key

    def _get_samples(self, batch_inds, env=None):
        samples = super()._get_samples(batch_inds, env)
        if self.aug_fn is not None:
            aug_obs = dict(samples.observations)
            aug_next = dict(samples.next_observations)
            aug_obs[self.image_key] = self.aug_fn(
                samples.observations[self.image_key])
            aug_next[self.image_key] = self.aug_fn(
                samples.next_observations[self.image_key])
            return DictReplayBufferSamples(
                observations=aug_obs, actions=samples.actions,
                next_observations=aug_next,
                dones=samples.dones, rewards=samples.rewards)
        return samples
```

Notice that obs and next_obs are augmented independently with different random shifts. This is by design in DrQ -- each sees a different view. We will revisit whether this is a good idea for SpatialSoftmax in Section 9.7.

9.4.3 HER + DrQ composed buffer

When using HER with pixel observations, we need goal relabeling AND augmentation. The `HerDrQDictReplayBuffer` composes both: HER first produces its merged batch (real + relabeled transitions with recomputed rewards), then we apply pixel augmentation to the result.

Listing 9.10: HerDrQDictReplayBuffer -- HER relabeling then DrQ augmentation

```python
class HerDrQDictReplayBuffer(HerReplayBuffer):
    def __init__(self, *args, aug_fn=None, image_key="pixels", **kwargs):
        super().__init__(*args, **kwargs)
        self.aug_fn = aug_fn
        self.image_key = image_key

    def sample(self, batch_size, env=None):
        samples = super().sample(batch_size, env)  # HER relabeling
        if self.aug_fn and self.image_key in samples.observations:
            aug_obs = dict(samples.observations)
            aug_next = dict(samples.next_observations)
            aug_obs[self.image_key] = self.aug_fn(
                samples.observations[self.image_key])
            aug_next[self.image_key] = self.aug_fn(
                samples.next_observations[self.image_key])
            return DictReplayBufferSamples(
                observations=aug_obs, actions=samples.actions,
                next_observations=aug_next,
                dones=samples.dones, rewards=samples.rewards)
        return samples
```

We override `sample()` rather than `_get_samples()` because HER's internal flow calls `_get_real_samples()` + `_get_virtual_samples()` and merges them without going through `_get_samples()`. Goals are never augmented -- only pixel observations change -- which preserves the HER invariant: `compute_reward(achieved_goal, desired_goal)` must be consistent with the reward stored in the transition, and augmenting goal vectors would break this invariant.

> **Checkpoint:** Run `bash docker/dev.sh python scripts/labs/image_augmentation.py --verify`. Expected: `[ALL PASS] Image augmentation verified.` Key checks: augmented images have same shape as input, two augmentations of the same input differ, goals and rewards are unchanged by augmentation.

9.5 Build It: Gradient routing

This is the most subtle component in the pipeline -- and the one that makes the difference between 5% and 95% success. The code changes are small (about 15 lines of overrides), but the impact is decisive.

9.5.1 The problem: SB3's default puts encoder in the actor optimizer

When you create SB3's SAC with `share_features_extractor=True` (the default for shared encoders), SB3 puts the encoder parameters in the **actor's** optimizer. During critic training, SB3 wraps the encoder forward pass in `set_grad_enabled(False)` -- gradients do not flow through the encoder during TD loss computation.

This means the encoder learns ONLY from the actor's policy gradient. Early in training the policy is random -- it pushes in random directions and succeeds roughly 5% of the time by luck -- so the actor loss gradient is essentially noise: with a near-uniform random policy, the gradient signal says "all directions are equally bad," which gives the encoder nothing to learn from. The encoder therefore remains at its random initialization, the critic cannot distinguish states (since they all look the same through a random encoder), and training never bootstraps. In our experiments, this default configuration produces 5-8% success, flat, for 2M+ steps -- regardless of whether you use NatureCNN or ManipulationCNN.

9.5.2 DrQ-v2 pattern: encoder in the critic optimizer

DrQ-v2 (Yarats et al., 2021) does the opposite: it places the encoder in the critic's optimizer, so the critic's TD loss directly asks "does this visual feature help predict future value?" -- a rich, stable learning signal. Meanwhile, the actor receives detached features, which prevents noisy policy gradients (especially at low success rates) from corrupting the encoder's representation.

The key insight is that the critic provides value-based supervision ("this state leads to high/low returns"), which is exactly what the encoder needs to learn useful spatial features. Even when success rate is near zero, the critic's TD error still provides signal: "this state where the gripper is near the puck has slightly higher Q than this state where the gripper is far away." That signal is weak, but it is directional -- it consistently pushes the encoder toward features that discriminate states by spatial proximity to goals. The actor's policy gradient, by contrast, is noisy and uninformative until the encoder already represents something useful -- a chicken-and-egg problem that the critic breaks.

The implementation requires one `forward()` override, one `.detach()` call, and an optimizer rewiring.

9.5.3 CriticEncoderCritic and CriticEncoderActor

Two small class overrides implement the DrQ-v2 pattern:

Listing 9.11: CriticEncoderCritic -- enable gradients through shared encoder

```python
class CriticEncoderCritic(ContinuousCritic):
    def forward(self, obs, actions):
        # CHANGED: always enable gradients through features_extractor
        # (SB3 default wraps this in set_grad_enabled(False))
        features = self.extract_features(obs, self.features_extractor)
        qvalue_input = th.cat([features, actions], dim=1)
        return tuple(q_net(qvalue_input) for q_net in self.q_networks)
```

Listing 9.12: CriticEncoderActor -- detach features before policy MLP

```
class CriticEncoderActor(Actor):
    def get_action_dist_params(self, obs):
        features = self.extract_features(obs, self.features_extractor)
        features = features.detach()  # CHANGED: stop encoder gradient
        latent_pi = self.latent_pi(features)
        mean_actions = self.mu(latent_pi)
        log_std = self.log_std(latent_pi)
        log_std = th.clamp(log_std, LOG_STD_MIN, LOG_STD_MAX)
        return mean_actions, log_std, {}
```

The first override removes the gradient gate so that the TD loss updates the encoder, while the second adds .detach() so the policy loss does NOT update the encoder. Together, these two changes route all encoder learning through the critic.

9.5.4 DrQv2SACPolicy -- wiring the optimizers

The policy subclass ties everything together: one shared encoder, encoder parameters in the critic optimizer, encoder parameters excluded from the actor optimizer.

Listing 9.13: DrQv2SACPolicy -- encoder in critic optimizer

```
class DrQv2SACPolicy(SACPolicy):
    def _build(self, lr_schedule):
        self.actor = self.make_actor()       # CriticEncoderActor
        self.critic = self.make_critic(      # CriticEncoderCritic
            features_extractor=self.actor.features_extractor  # shared
        )
        # REVERSED from SB3 default:
        # encoder in CRITIC optimizer, excluded from actor optimizer
        encoder_ids = {id(p) for p in
                       self.actor.features_extractor.parameters()}
        actor_params = [p for p in self.actor.parameters()
                        if id(p) not in encoder_ids]
        self.actor.optimizer = self.optimizer_class(
            actor_params, lr=lr_schedule(1), **self.optimizer_kwargs)
        self.critic.optimizer = self.optimizer_class(
            list(self.critic.parameters()),  # includes shared encoder
            lr=lr_schedule(1), **self.optimizer_kwargs)
        # Target critic gets its own encoder copy (not shared)
        self.critic_target = self.make_critic(features_extractor=None)
        self.critic_target.load_state_dict(self.critic.state_dict())
```

The identity-based filtering (id(p) not in encoder_ids) avoids fragility if parameter naming conventions change across SB3 versions, since we match on Python object identity rather than parameter name strings. The target critic gets its own separate encoder that is updated via Polyak averaging ($\tau = 0.005$), as in standard SAC -- this is important because the target encoder must NOT be the same object as the online encoder, or Polyak averaging would be a no-op.

It is worth tracing the gradient flow carefully through this shared-encoder setup:

1. **Forward pass:** Computing the actor loss requires calling the critic's Q-networks: $L_{\text{actor}} = \alpha \log \pi(a|s) - Q(s, a)$. The critic uses the shared encoder to compute features for $Q(s, a)$.
2. **Backward pass:** When PyTorch runs `actor_loss.backward()`, it computes gradients for *every* parameter on the computational graph -- including the shared encoder, because the encoder sits between the pixels and the Q-value.
3. **Why no update happens:** PyTorch's `optimizer.step()` only updates parameters that are in `optimizer.param_groups`, and the actor optimizer does not contain encoder parameters (we filtered them out). So even though `.backward()` writes gradients into the encoder's `.grad` tensors, `actor_optimizer.step()` ignores them entirely, and the encoder is only updated when `critic_optimizer.step()` runs.
4. **The `CriticEncoderActor` detach:** Our actor wrapper detaches features before the policy MLP as a safety measure, preventing encoder gradients from being *computed* during the actor loss backward pass. This is a belt-and-suspenders approach: the optimizer filtering (step 3) is sufficient on its own, but the detach avoids wasting compute on gradients that would be ignored anyway.

The net effect: the encoder learns from Bellman error (critic loss) only, not from the actor's policy gradient. This is the DrQ-v2 design -- the encoder should learn *state features* from TD error, not learn to *fool the critic* via the actor.

> **Checkpoint:** Run `bash docker/dev.sh python scripts/labs/drqv2_sac_policy.py --verify`. Expected: `[ALL PASS] DrQ-v2 SAC policy verified.` Key checks:
>
> - 0 encoder params in actor optimizer, all encoder params in critic optimizer
> - After `critic.backward()`: encoder params have non-zero gradients
> - After `actor.backward()`: encoder params have NO gradients (features detached)
> - Actor and critic share the same encoder instance (`is` check passes)
> - Target critic has its own encoder (separate instance)
> - Save/load round-trip: predictions match within 1e-5 tolerance

9.6 Bridge: From scratch to SB3

You have now built 13 components across five lab files: a pixel wrapper, a replay buffer, two CNN encoders, SpatialSoftmax, an SB3-compatible feature extractor, DrQ augmentation, two DrQ replay buffers, and three gradient routing overrides. These are the same components that SB3 uses when you launch a pixel training run, so we can verify the full pipeline with three commands:

```bash
bash docker/dev.sh python scripts/labs/pixel_wrapper.py --verify
bash docker/dev.sh python scripts/labs/manipulation_encoder.py --verify
bash docker/dev.sh python scripts/labs/image_augmentation.py --verify
bash docker/dev.sh python scripts/labs/drqv2_sac_policy.py --verify
```

All four should print `[ALL PASS]`. Then run the bridging proof to verify that our from-scratch components produce identical results to what SB3 uses internally:

```bash
bash docker/dev.sh python scripts/labs/drqv2_sac_policy.py --bridge
```

This feeds the same pixel observation through our ManipulationExtractor and SB3's features_extractor, comparing output tensors. Expected: `max_diff < 1e-6` -- the implementations are numerically identical.

Next, run the gradient probe to see how SB3's default compares to our override:

```bash
bash docker/dev.sh python scripts/labs/drqv2_sac_policy.py --probe
```

This prints the encoder parameter membership in the actor and critic optimizers for both SB3's default `SACPolicy` and our `DrQv2SACPolicy`, so you should see encoder parameters in the actor optimizer only (SB3 default) versus encoder parameters in the critic optimizer only (our override).

Here is how the Build It components map to what you see in TensorBoard during training:

TensorBoard metric	Build It component	What it measures
`train/critic_loss`	CriticEncoderCritic (Listing 9.11)	TD error through sha
`train/actor_loss`	CriticEncoderActor (Listing 9.12)	Policy loss on detach
`train/ent_coef`	SAC automatic tuning (Ch3)	Entropy temperature
`rollout/success_rate`	Full pipeline: wrapper + encoder + routing + HER	End-to-end task perf

Before moving to the full investigation, you can see the pipeline produce actual (short) learning curves on CPU:

```bash
bash docker/dev.sh python scripts/ch09_pixel_push.py demo --seed 0
```

This runs roughly 10 minutes on CPU with a small buffer and fewer steps. It will not reach the hockey-stick -- the demo is too short for that -- but you will see the critic loss decline during Phase 1, confirming that the pipeline is wired correctly and the encoder receives gradients.

What SB3 adds on top of our components: vectorized environment rollout (`SubprocVecEnv` for parallel pixel rendering), optimized replay sampling with proper episode boundary handling, TensorBoard logging, and checkpoint management. These are engineering concerns, not learning concerns -- the math is what you built.

9.7 Run It: The five-step investigation

This is where the components meet reality. We run five experiments, each adding one piece, and watch the success rate. The first two fail, the third succeeds (eventually), and the fourth shows that a standard technique (DrQ) actually hurts. Each failure teaches a transferable debugging principle.

Step 0: NatureCNN baseline (pixels are not drop-in)

Start with the obvious approach: take the working SAC + HER pipeline from Chapter 5 and replace the 25D state vector with 84x84 pixels. Use SB3's default NatureCNN encoder.

```bash
bash docker/dev.sh python scripts/ch09_pixel_push.py train \
  --seed 0 --total-steps 2000000
```

Result: 5-8% success, flat, for 2M+ steps -- indistinguishable from a random policy.

The algorithm is proven to work at 89% from state vectors, and the only change is the observation modality, so something about the visual processing pipeline is fundamentally wrong.

Look at the spatial progression from Section 9.3.1: NatureCNN's stride-4 first layer crushes an 84x84 image down to 20x20. The puck, which starts at roughly 5 pixels wide, becomes roughly 1 pixel. The gripper-puck spatial relationship -- the entire signal the policy needs -- is destroyed in the first convolutional layer.

Principle: Pixels are not a drop-in replacement for state vectors. The encoder architecture must match the task's spatial requirements.

Step 1: Architecture fix (ManipCNN + SpatialSoftmax + proprioception)

Replace NatureCNN with the components from Sections 9.3.2-9.3.5: ManipulationCNN (gentle 3x3 stride-2 downsampling), SpatialSoftmax (extract ”where” coordinates, not ”what” features), and proprioception passthrough (10D robot state alongside pixel features). Same pipeline, better encoder -- the CNN can now represent precise spatial relationships between 5-pixel objects.

Result: Still 5-8% flat. Architecture is necessary but not sufficient.

This is the more subtle failure. The encoder CAN represent the right information -- ManipulationCNN preserves the puck at roughly 3 pixels through all four layers, and SpatialSoftmax extracts precise (x, y) coordinates from those activations -- but CAN represent and DOES represent are different things. The network has the capacity; the question is whether it is learning.

Checking where the encoder's gradients come from reveals the problem. In SB3's default configuration, the encoder sits in the actor optimizer, and the critic disables gradients through the encoder during TD updates, so the encoder learns only from the actor's policy gradient -- which is noisy and weak early in training because the policy is near-random. A random policy generates the gradient signal ”all directions are equally bad,” which gives the encoder nothing useful to learn from.

Principle: Architecture determines what a network CAN represent. Training determines what it DOES represent. We fixed the capacity; now we need to fix the learning signal.

Step 2: Gradient routing fix (critic-encoder) -- the breakthrough

Add the 15 lines from Section 9.5: `CriticEncoderCritic`, `CriticEncoderActor`, and `DrQv2SACPolicy`. Move the encoder into the critic optimizer. Detach features before the actor.

```
---------------------------------------------------------
EXPERIMENT CARD: SAC + HER + Pixel Pipeline on FetchPush-v4
---------------------------------------------------------
Algorithm:    SAC + HER (critic-encoder gradient routing,
              ManipulationCNN + SpatialSoftmax, no DrQ)
Environment:  FetchPush-v4 (pixel observations, sparse reward)
Fast path:    5,000,000 steps, seed 0
Time:         ~40 hours (GPU); not feasible on CPU for full run
              (Build It --verify: < 2 min CPU; --demo: ~10 min CPU)

Run command (fast path):
  bash docker/dev.sh 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

Checkpoint track (skip training):
  checkpoints/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed0.zip

Expected artifacts:
  checkpoints/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed0.zip
  checkpoints/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed0.meta.json
  results/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed0_eval.json
  runs/ch09_manip_noDrQ_criticEnc/FetchPush-v4/seed0/

Success criteria (fast path):
  success_rate >= 0.90 (at 5M steps)
  hockey-stick inflection visible in TensorBoard by ~2.5M steps
  critic_loss non-monotonic trajectory (decline -> rise -> decline)

Full multi-seed results: see REPRODUCE IT at end of chapter.
---------------------------------------------------------
```

Result: Flat at 6% for 2M steps -- you might think this failed too, but it did not. At 2.2M steps a slow upward trend appears; by 2.5M, success hits 25-34%; by 3.5M it crosses 70%; and by 4.4M it reaches 95%.

This is the **hockey-stick learning curve**: a long flat phase where the encoder is learning spatial structure from the critic's TD signal, followed by a rapid climb once the representation becomes good enough for the policy to exploit. The flat phase is not failure -- it is the representation learning overhead that pixel RL imposes. Section 9.8 explains the three mechanisms behind this curve.

The difference between Step 1 and Step 2 is 15 lines of gradient routing code -- the architecture is identical. The only change is WHERE the encoder's learning signal comes from: the critic's TD loss (rich, stable, directional even at low success rates) versus the actor's policy gradient (noisy, uninformative when the policy is near-random). This is the decisive choice of the chapter.

Principle: Where gradients flow matters as much as what architecture you use. The encoder needs value-based supervision from the critic, not noisy policy gradients from the actor.

Step 3: Reading the training curve (the patience tax)

Before adding more components, we pause to understand what just happened. Step 2's curve is not a smooth ascent -- it has three distinct phases, each with a characteristic loss signature. Learning to read this signature helps you decide when to be patient and when to intervene. Section 9.8 unpacks the three-phase loss signature in detail.

The headline: pixel RL needs 2-4x the training budget of state-based RL. State-based Push reached 89% at 2M steps, while pixel Push reached 95% at 4.4M steps -- a 2.2x overhead. If your stop rules are calibrated from state-based experience (”kill the run if no progress at 2M steps”), it helps to recalibrate for pixel RL, because otherwise runs get killed during Phase 1 before the hockey-stick has a chance to appear.

Principle: The representation learning phase is an unavoidable overhead. Rising losses during the hockey-stick are good news, not bad. Always read loss curves alongside success rate.

Step 4: DrQ ablation -- augmentation versus representation

You might wonder: we never added DrQ data augmentation, and DrQ is the standard technique for pixel RL (Kostrikov et al., 2020). Would adding it help?

Run Step 2's exact configuration with one change -- DrQ augmentation enabled (omit the `--no-drq` flag) -- while keeping all other hyperparameters (buffer size, HER strength, encoder, gradient routing) identical to Step 2:

```bash
bash docker/dev.sh python scripts/ch09_pixel_push.py train \
  --seed 0 --critic-encoder --buffer-size 500000 \
  --her-n-sampled-goal 8 --total-steps 2000000
```

Result: 3% success, flat at 1.54M steps. DrQ makes things worse. (By comparison, Step 2 showed a clear upward trend by 1.4M steps with the same hyperparameters minus DrQ.)

Here is why. DrQ shifts images by +/-4 pixels via pad-and-crop, which after the CNN (84 -> 21 spatial resolution) becomes +/-1 pixel on the 21x21 feature map. SpatialSoftmax converts this to +/-0.10 in the [-1, 1] coordinate space, and -- critically -- DrQ augments obs and next_obs independently with different random shifts, doubling the noise in Bellman targets.

The gripper-puck distance signal in SpatialSoftmax coordinates is roughly 0.25-0.50 units, so with +/-0.10 noise on both obs and next_obs (independent), the noise-to-signal ratio reaches 40-80%. The TD target becomes unreliable, and the critic cannot learn the value structure it needs to train the encoder.

Principle: Data augmentation is not universally good. Choose your representation, then choose your augmentation. SpatialSoftmax extracts precise spatial coordinates; DrQ injects spatial noise. They are fundamentally incompatible.

Investigation summary

Figure 9.3 summarizes all five experimental steps.

Five-step investigation summary showing success rate for each configuration: Step 0 NatureCNN at 5%, Step 1 ManipCNN+SS at 5%, Step 2 plus critic-encoder at 95% after hockey-stick, Step 3 interprets the training curve, Step 4 plus DrQ at 3%

Figure 9.3: The five-step investigation. Steps 0 and 1 fail because the architecture is wrong (stride-4) and gradient routing is wrong (encoder in actor optimizer). Step 2 succeeds after the representation learning phase completes. Step 3 interprets the training curve. Step 4 shows that DrQ augmentation is harmful when combined with SpatialSoftmax. The breakthrough is gradient routing (Step 2), not architecture (Step 1) or augmentation (Step 4). (Generated by `bash docker/dev.sh python scripts/ch09_pixel_push.py plot-investigation`.)

Step	What changed	Success rate	Lesson
Full-state control	None (25D vectors)	89% at 2M	Pipeline validat
0: NatureCNN	Pixels with default CNN	5% flat	Stride-4 destro
1: ManipCNN + SS + proprio	Better encoder	5% flat	Architecture is
2: + critic-encoder	Gradient routing fix	95% at 4.4M	WHERE gradier
3: Training curve	Interpret Step 2's loss signature	(see Section 9.8)	Rising losses =
4: + DrQ	Data augmentation	3% flat	Augmentation

9.8 Reading the training curve

The Step 2 training curve is not a smooth ascent. It has three distinct phases, each with a characteristic loss signature. Learning to read this signature helps you tell the difference between a genuinely stuck run and one that just needs more time (”the critic loss is declining, the representation is warming up -- give it another 2M steps”).

The three-phase loss signature

Figure 9.4 shows the three phases annotated on the actual training curve from Step 2.

Three-phase loss signature showing success rate, critic loss, and actor loss over training steps, with Phase 1 (flat success, declining critic loss), Phase 2 (rising success, rising losses), and Phase 3 (saturating success, declining losses) annotated

Figure 9.4: The three-phase loss signature from Step 2. Phase 1: the critic memorizes uniform failure. Phase 2: real value learning begins -- losses rise because the problem becomes harder for the critic. Phase 3: the value function converges to an accurate model. (Generated from TensorBoard logs at `runs/ch09_manip_noDrQ_criticEnc/FetchPush-v4/seed0/`.)

Phase	Steps	success_rate	critic_loss	actor_loss	What
1: Flat	0-2.2M	3-7%	Declining to 0.07	~0	Critic
2: Hockey-stick	2.2-3.5M	10-90%	Rising to 0.3+	Rising to 1.0+	Encod
3: Convergence	3.5M+	95%+	Declining to 0.15-0.35	Negative (-0.2 to -0.7)	Value

The counterintuitive lesson: **rising losses during Phase 2 are good news.** In supervised learning, rising loss means the model is getting worse. In sparse RL, rising critic loss means the critic has moved past the trivial solution (predict constant $Q = -18.5$ for all states) and is now trying to learn which states actually lead to success. That is a harder prediction problem, so the loss rises -- but it is a productive rise. The success rate climbing alongside the loss confirms this.

Phase 1's declining loss with flat success is the diagnostic red flag: it looks healthy on a loss plot, but the critic is not learning value structure -- it is memorizing the fact that all trajectories fail. If you only watch loss curves without checking success rate, Phase 1 looks indistinguishable from Phase 3, which is why we always read loss curves alongside success rate. The loss value alone is ambiguous.

The actor loss going negative in Phase 3 is a SAC-specific convergence signal. The actor loss is $L_{\text{actor}} = \alpha \log \pi(a|s) - Q(s, a)$. When Q is high (near 0, meaning "success is likely"), the Q-value term dominates the entropy penalty $\alpha \log \pi$, making the total loss negative. This means the policy is confidently selecting high-value actions -- a sign of convergence, not divergence.

Sidebar: Why the hockey-stick?

The flat phase followed by rapid improvement is not accidental. Three mechanisms interact to produce it.

1. Value propagation bottleneck. The critic learns Q-values through Bellman backups: $Q(s_t, a_t|g) \leftarrow r_t + \gamma \max_{a'} Q(s_{t+1}, a'|g)$. Each backup propagates value information one step outward from states where the agent has succeeded. HER seeds this process by relabeling nearby goals, but the "value wavefront" still grows one backup step at a time. Laidlaw et al. (2024) formalize this as the effective horizon k^* -- the minimum number of backup steps before greedy actions become near-optimal. Sample complexity is exponential in k^*, creating a sharp threshold between "not enough training" and "enough."

2. Geometric phase transition. Test goals are sampled uniformly over the table surface. The agent's ”competence region” -- the set of goals it can push to -- starts small and grows as the value wavefront expands. In 2D, the overlap between a growing competence region and the fixed test goal distribution scales quadratically with the competence radius. When the radius is small, almost no test goals are reachable. Once it crosses a critical threshold, many become reachable at once. This creates the sharp inflection from 5% to 30%+.

3. Positive feedback loop. Once real test goals are reached, successful episodes provide higher-quality training signal than HER's relabeled goals (no distribution mismatch from goal substitution). Better signal drives the policy to explore further, reaching more goals, generating more signal. This converts the linear wavefront expansion into super-linear growth -- the steep part of the hockey-stick.

The full picture is assembled from three independent theoretical results (Laidlaw et al., 2024 on effective horizon; Wang & Isola, 2022 on quasimetric Q-functions; Huang et al., 2025 on difficulty spectrum dynamics), not derived from a single theorem. No closed-form expression for the inflection point exists. The practical lesson: if your HER training curve is flat at 1M steps but critic loss is declining, the wavefront may not have reached the test goal distribution yet. Our Step 2 went from 6% at 2M to 95% at 4.4M.

The patience tax

Pixel RL needs 2-4x the training budget of state-based RL. State-based Push reached 89% at 2M steps, while pixel Push reached 95% at 4.4M steps -- a 2.2x overhead. This overhead is the representation learning phase: the encoder must learn useful spatial features before the policy can exploit them, and there is no shortcut.

We find it helpful to recalibrate stop rules for pixel RL. State-based intuitions (”no progress at 2M means it is broken”) can lead to terminating runs during Phase 1, before the hockey-stick has a chance to emerge.

Tip: For pixel RL with sparse rewards, set your initial training budget to at least 4x the state-based budget. Monitor critic loss: if it is declining during the flat phase, the representation is warming up. If critic loss is flat AND success is flat for 3M+ steps, something is likely wrong -- check the ”What Can Go Wrong” table.

9.9 What Can Go Wrong

Pixel RL has more failure modes than state-based RL because the visual pipeline adds a large surface area for bugs. This table covers the failures we encountered and the ones readers are most likely to hit.

Symptom	Likely cause
Flat at 5% after 3M+ steps	Missing `--criti`
OOM during training	Pixel replay buff
Flat at 5% WITH `--critic-encoder` after 2M steps	Normal pre-hock
`critic_loss` declining monotonically for 3M+ steps, success still flat	Critic memorizin
DrQ + SpatialSoftmax stuck at 3%	Augmentation-r
Very slow FPS (< 15 fps)	Multiple Docker
TensorBoard shows mixed/confusing curves	Log contaminat
`--probe` shows encoder in actor optimizer	Using SB3's def
Hockey-stick at 2.2M but convergence stalls at 50-60%	Buffer too small
`RuntimeError: Unable to sample before end of first episode` on resume	`learning_star`

The first row is the most common failure mode. We watched it happen multiple times during development. Everything looks correct -- ManipulationCNN, SpatialSoftmax, proprioception, HER, large buffer -- but the encoder learns nothing because it is in the wrong optimizer. One flag fixes it.

9.10 Summary

This chapter added a visual observation pipeline to the SAC + HER stack from Chapters 4-5 and achieved 95%+ success on FetchPush from raw 84x84 pixels -- matching the state-based performance at the cost of 2.2x more training steps.

What you built (13 components across 5 lab files):

- `render_and_resize`: camera capture to CHW tensor
- `PixelObservationWrapper`: frame stacking, proprioception passthrough, goal modes
- `PixelReplayBuffer`: uint8 storage, float32 at sample time
- `NatureCNN` (the "wrong" encoder): stride-4 destroys 5-pixel objects
- `ManipulationCNN` (the "right" encoder): 3x3 stride-2 preserves spatial information
- `SpatialSoftmax`: extracts "where" coordinates, not "what" features
- `ManipulationExtractor`: routes dict observations for SB3 compatibility
- `RandomShiftAug`: DrQ pad-and-crop augmentation
- `DrQDictReplayBuffer` and `HerDrQDictReplayBuffer`: augment at sample time
- `CriticEncoderCritic` and `CriticEncoderActor`: gradient routing overrides
- `DrQv2SACPolicy`: encoder in critic optimizer, detached actor features

Three transferable principles:

1. **Architecture must match the task.** NatureCNN's stride-4 was designed for Atari sprites, not 5-pixel manipulation objects, so ManipulationCNN's stride-2 with SpatialSoftmax is needed to preserve spatial precision.
2. **Gradient routing is decisive.** The encoder needs the critic's value-based supervision, not the actor's noisy policy gradient -- 15 lines of code and the difference between 5% and 95%.

3. **Augmentation must match representation.** DrQ's random shift corrupts SpatialSoftmax's precise coordinates, so choosing the representation first and then selecting compatible augmentation is essential.

The compositional insight: This chapter did not invent a new algorithm; it composed SAC (Ch4) + HER (Ch5) + a visual pipeline (this chapter) and discovered that the critical missing piece was not a new loss function or a clever trick, but the routing of gradients through the encoder. The components from earlier chapters transferred directly -- the innovation was in wiring them correctly for pixel observations.

Looking ahead: We achieved 95% from pixels, but it took 4.4M steps and roughly 40 hours of GPU time per seed, because the representation learning phase -- the long flat period before the hockey-stick -- is an unavoidable overhead when learning visual features from scratch. Chapter 10 explores whether pre-trained visual encoders and world models can reduce this tax, and what happens when the gap between simulation and reality becomes the bottleneck.

```
----------------------------------------------------------
REPRODUCE IT
----------------------------------------------------------
The results and pretrained checkpoints in this chapter
come from these runs:

  for seed in 0 1 2; do
    bash docker/dev.sh python scripts/ch09_pixel_push.py train \
      --seed $seed --critic-encoder --no-drq --buffer-size 500000 \
      --her-n-sampled-goal 8 --total-steps 5000000 \
      --checkpoint-freq 500000
  done

Hardware:     NVIDIA GPU with >= 60 GB system RAM
              (tested on DGX; any modern GPU works, times will vary)
Time:         ~40 hours per seed at ~30 fps (GPU; not feasible on
              CPU -- use the checkpoint track instead)
Seeds:        0, 1, 2

Artifacts produced:
  checkpoints/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed{0,1,2}.zip
 checkpoints/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed{0,1,2}.meta.json
  results/ch09_manip_noDrQ_criticEnc_FetchPush-v4_seed{0,1,2}_eval.json
  runs/ch09_manip_noDrQ_criticEnc/FetchPush-v4/seed{0,1,2}/

Results summary (what we got -- seed 0; seeds 1-2 pending):
  success_rate: 0.95  (seed 0, 100 episodes)
  hockey_stick_onset: ~2.2M steps
  90%_success: ~3.5M steps
  training_time: ~40h per seed
  Multi-seed variance will be reported after seeds 1 and 2 complete.
  We expect similar results but cannot claim +/- bounds from one seed.

Comparison runs (failure baselines for the investigation):
  # Step 0: NatureCNN baseline
  bash docker/dev.sh python scripts/ch09_pixel_push.py train \
    --seed 0 --total-steps 2000000

  # Step 4: DrQ ablation (shows augmentation-representation conflict)
  bash docker/dev.sh python scripts/ch09_pixel_push.py train \
    --seed 0 --critic-encoder --buffer-size 500000 \
    --her-n-sampled-goal 8 --total-steps 2000000

If your numbers differ by more than ~10% (hockey-stick not visible
by 3M, or final success below 80%), check the "What Can Go Wrong"
section above.

The pretrained checkpoints are available in the book's
companion repository for readers using the checkpoint track.
----------------------------------------------------------
```

Exercises

1. (Verify) Run the full-state control baseline.

Confirm that the Ch9 script wiring is correct independently of pixel processing:

```bash
bash docker/dev.sh python scripts/ch09_pixel_push.py train \
  --seed 0 --full-state --total-steps 2000000
```

Expected: success_rate >= 0.85 at 2M steps, matching Ch5's state Push results. If this fails, the problem is in the script wiring, not the visual pipeline. Always validate the non-pixel path first.

2. (Tweak) HER relabeling strength.

Change `--her-n-sampled-goal` from 8 to 4 (the Ch5 default). Run for 3M steps with the winning config (`--critic-encoder --no-drq --buffer-size 500000`). Does the hockey-stick onset shift later? What about the final success rate?

Expected: later inflection (more steps needed to build the value wavefront with fewer relabeled transitions). Final success may be similar but convergence is slower. Quantify: at what step does success first exceed 20% with HER-4 vs HER-8?

3. (Explore) SpatialSoftmax ablation.

Run with `--no-spatial-softmax` (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 a necessary component or a helpful inductive bias?

Expected: likely still learns (ManipulationCNN + critic-encoder may be sufficient), but possibly slower or with lower final success. The result tells you whether ”where not what” is critical or supplementary.

4. (Challenge) Test DrQ with flat features.

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?

Expected: if DrQ helps with flat features, the conclusion is ”DrQ and SpatialSoftmax are separately good ideas but fundamentally incompatible.” If DrQ still hurts, the conclusion is ”DrQ's random shift is harmful for manipulation regardless of feature type.”

5. (Challenge) Measure gradient magnitudes.

Read the `verify_gradient_flow()` function in `scripts/labs/drqv2_sac_policy.py`. Modify it to measure the actual gradient magnitude (L2 norm) flowing through the encoder during critic versus actor training. How much larger is the critic gradient? This quantifies the argument from Section 9.5 that the critic provides a richer learning signal than the actor.

Expected: critic gradient magnitude should be substantially larger than actor gradient (before the detach). Report the ratio as a concrete number -- this is the quantitative evidence for why critic-encoder routing matters.