Extended Abstract

Motivation When a legged robot's joints degrade mid-deployment (through wear, impact, or actuator fatigue), its locomotion suffers. The standard fix is to give the policy explicit damage labels: which joint failed, how badly, what mode. But in practice, damage is gradual and unlabeled. Nobody hands the robot a fault vector. The question driving this project: can a policy figure out what's broken on its own, just from how the world responds to its actions?

Method We pair a SAC policy with a context encoder that processes a sliding window of 16 recent (observation, action) transitions and outputs a latent code $z \in \mathbb{R}^{32}$. The policy sees z concatenated with the current observation, with no explicit fault signal and no auxiliary loss. To make the comparison fair, both the encoder model and a memoryless baseline train under identical domain randomization over damage severity. The only difference is that the encoder gets to look at history; the baseline does not.

Implementation Everything runs on MuJoCo Ant-v4 with a custom contact-force fatigue model where fatigue accumulates from both action magnitude and ground-reaction forces. By design, ankle joints fatigue 30–40% faster than hips because they bear the ground contact, while hip actuators see near-zero external force, so the damage pattern is structurally grounded rather than arbitrary. The encoder is an LSTM implemented as a SB3 feature extractor, with history folded directly into the observation vector so we can use stock SAC with an unmodified replay buffer. Both models train for 500k steps on Modal A10G GPUs.

Results Under severe damage (torque scaled to 10%), the baseline drops from 83% to 70% success. The encoder holds at 86%, not just recovering the gap but exceeding the baseline's healthy-condition performance (recovery fraction = 1.23×). Linear probing on frozen z confirms this isn't luck: 75.1% accuracy on 3-way condition classification (vs. 65% from raw obs, 33.3% chance), and 95.5% per-joint damage identification. Evaluated over 100 episodes per condition.

Discussion The core finding is that the encoder extracts fault information that simply isn't available in any single observation. You need the temporal pattern of "I commanded X but got Y" over multiple steps to tell mild from severe, and the 3-way probe gap (z: 75.1% vs. raw obs: 65.0%) confirms this directly. What makes this work without any auxiliary loss or privileged supervision is that the critic's gradient alone pushes z toward damage-relevant features. The encoder figures out what's broken purely because knowing what's broken helps it predict future reward, no explicit fault label needed at any stage.

An interesting pattern is the severe-vs-healthy gap: the encoder hits 86% under severe damage but only 82% under no damage (though at n=100, that 4pp difference could be noise). One way to read it: a locked joint (s=0.1) creates a highly predictable dynamical regime, so once the encoder identifies it, it can commit fully to a specialized gait. Under healthy conditions, there's slight ambiguity in z early in the episode ("is this healthy, or has damage just not triggered yet?").

Conclusion A context encoder trained end-to-end with SAC can implicitly detect progressive joint damage from dynamics alone. No privileged fault labels needed.

Fault-Adaptive Locomotion via Implicit Damage Detection in MuJoCo Ant

Sidhanth Mishra
Department of Computer Science
Stanford University
grid@stanford.edu

Abstract

We train a context encoder alongside a SAC policy to detect and adapt to progressive joint damage in MuJoCo Ant-v4, using only observation-action history. No explicit fault labels, no auxiliary losses. Under domain-randomized damage, the encoder achieves 86% goal-reaching success at severe degradation (10% torque) where a memoryless baseline drops to 70% (100 episodes per condition). Linear probing confirms the latent code tracks damage at joint-level resolution (95.5% per-joint accuracy). The encoder more than recovers the baseline's damage-induced drop (recovery fraction $1.23\times$), and maintains performance comparable to the baseline's healthy-condition rate even under severe degradation.

1 Introduction

Legged robots break. Joints wear out, actuators degrade, impacts fracture components, and when that happens mid-deployment, the robot needs to adapt its gait on the fly. The question is: how does the policy know what's broken?

We focus on a setting where damage is **endogenous** (caused by the agent's own actions, not injected externally) and **implicit** (never signaled to the agent in any form). This combination matters because it creates a feedback loop that exogenous damage models lack: the agent's own aggressive behavior is what destroys its actuators, so the optimal policy has to balance task performance against self-preservation, and it has to do this without ever being told that anything is wrong.

Prior work answers this in ways that assume you have information you usually don't. Kim et al. [2024] train quadrupedal locomotion under random joint masking, where the policy knows which joints are zeroed out during training. UMC [Qiu et al., 2025] takes a similar approach with masked malfunction training for resilient control. Cully et al. [2015] build a behavioral repertoire offline and search it after damage, which works but requires dedicated trial periods. RMA [Kumar et al., 2021] trains an adaptation module with access to ground-truth environment parameters. All of these produce strong adaptation, but they all require some form of privileged information about which joints are impaired.

Meta-RL approaches like PEARL [Rakelly et al., 2019] and RL^2 [Duan et al., 2016] learn to adapt from context, but they target task-level variation (different reward functions across episodes). A damaged joint doesn't change the goal, it changes the *transition dynamics*. That's a different kind of distribution shift, and task-level meta-RL is an imprecise tool for it.

Our approach borrows the context-encoding idea from meta-RL but applies it to embodiment-level variation: give the policy a history window and let it figure out what's wrong. A context encoder (implemented as an LSTM) processes the last 16 transitions, produces a latent code z, and the policy conditions on z alongside the current observation. No privileged fault labels, no auxiliary losses, no

separate adaptation phase. The encoder learns to detect damage purely from the RL objective; the critic's gradient is enough.

1.1 Task

We test this on a goal-reaching task in MuJoCo Ant-v4 [Todorov et al., 2012]: navigate a quadruped to target $[5, 5]$ (diagonal, so the native +x Ant bias actively hurts) under progressive joint damage. A contact-force fatigue model accumulates wear per actuator from action magnitude and ground-reaction forces, eventually scaling torque capacity down to 50% (mild) or 10% (severe).

The important evaluation choice: *deadline-based success metrics*. Without a deadline, even a severely damaged ant can eventually limp to a nearby goal given enough time. We measure success at step budgets of 250, 350, and 500, which is where the performance gap actually shows up.

1.2 Contributions

1. A contact-force fatigue model for endogenous, progressive damage where fatigue rate depends on both action magnitude and ground-reaction forces, naturally producing hip/ankle asymmetry from the physics (ankles bear ground contact; hips do not).
2. A context-encoder-conditioned SAC architecture that adapts to implicit damage without privileged information, implemented as an LSTM feature extractor compatible with stock Stable-Baselines3 [Raffin et al., 2021], no custom replay buffer needed.
3. Evaluation under deadline constraints that separate fast goal-reaching from eventual convergence, confirming that the encoder's successes are prompt, not barely-in-time.
4. Latent-space probing showing the encoder genuinely tracks damage state at joint-level resolution, rather than just learning a robust average policy.

2 Related Work

Context encoding in meta-RL. The idea of conditioning a policy on a learned context variable goes back to meta-RL: PEARL [Rakelly et al., 2019] infers a probabilistic task embedding z from collected transitions, and RL^2 [Duan et al., 2016] folds the full episode history into an RNN hidden state so the policy adapts at test time without gradient updates. We borrow this mechanism directly, but the variation we target is different. Both PEARL and RL^2 were designed for reward-function variation across tasks (different goals, different MDPs). A degrading joint doesn't change the goal, it changes the *dynamics* within a single episode. That means we need temporal structure (when did the shift happen?) rather than just task identity (which MDP am I in?), which is why we use a recurrent encoder over a sliding window rather than PEARL's permutation-invariant aggregation.

Fault-tolerant locomotion. The robotics community has attacked this from several angles, all of which assume some form of privileged damage information. Cully et al. [2015] pre-build a repertoire of $\sim$13,000 behaviors offline, then search it via Bayesian optimization after damage, recovering in under two minutes but requiring dedicated trial-and-error periods and an expensive offline phase. More recent work skips the repertoire: Kim et al. [2024] randomize joint masks during training so the policy learns gaits robust to missing actuators, and UMC [Qiu et al., 2025] extends this to multiple failure modes. These are effective, but the training procedure explicitly designates which joints are impaired, an assumption that breaks down the moment damage stops being a discrete external event and starts being something the robot does to itself through normal operation. The closest work to ours is RMA [Kumar et al., 2021], which trains an adaptation module on proprioceptive history for sim-to-real locomotion. RMA handles online adaptation, but through teacher-student distillation where the teacher sees ground-truth environment parameters. Our encoder gets no privileged signal at any stage, and targets endogenous actuator degradation rather than extrinsic variation like terrain or payload.

Recurrent policies and partial observability. Hausknecht and Stone [2015] showed that replacing DQN's feedforward layers with an LSTM lets agents maintain belief state over hidden variables in partially observable Atari games. Our setting is a POMDP for the same structural reason: damage level is a hidden state variable that affects transitions but never appears in the observation. The practical

difference is that we use the LSTM as a *feature extractor* that folds history into the observation vector, rather than replacing the full policy network. This lets us run stock off-policy SAC [Haarnoja et al., 2018] with an i.i.d. replay buffer, avoiding the complexity of sequence-aware sampling.

Domain randomization. Training under randomized simulation parameters [Tobin et al., 2017, Peng et al., 2018] makes the damaged world look like another sample from the training distribution. We randomize damage severity so both models see healthy, mild, and severe episodes during training. The important constraint: both models train on the *same* DR distribution, so the encoder's advantage can only come from being able to infer the current condition from history, not from seeing different data.

3 Method

3.1 Environment and Damage Model

We build on MuJoCo Ant-v4 [Todorov et al., 2012] with a goal at $[5, 5]$ (diagonal locomotion). The reward combines the native Ant reward (forward velocity + healthy bonus − control cost), a progress-based shaping term, and a sparse completion bonus:

$$r_t = w_f \cdot r_t^{\text{native}} + w_p \cdot (d_{t-1} - d_t) + b_{\text{succ}} \cdot \mathbb{1}[d_t < 1] \tag{1}$$

with $w_f = 0.5$ (forward reward weight), $w_p = 50$ (progress weight), and $b_{\text{succ}} = 200$. The forward reward weight matters: at 0.0 the ant has no incentive to walk (it just stands still and banks the healthy bonus), and at 1.0 the native +x bias sends it east before turning toward the diagonal goal, which kills encoder credit assignment. 0.5 was the sweet spot.

Contact-force fatigue. The `DamageWrapper` accumulates fatigue per actuator:

$$f_i(t) = f_i(t-1) + \alpha |a_i(t)| + \beta \|F_i^{\text{ext}}(t)\| \tag{2}$$

where $\alpha = 1.0$ weights action magnitude and $\beta = 0.1$ weights external contact forces from MuJoCo's `cfrc_ext`. Once f_i crosses the threshold ($\tau = 30$), that actuator's torque is permanently scaled: $a'_i = s \cdot a_i$ with $s \in \{0.1, 0.5, 1.0\}$ for severe, mild, and healthy conditions.

By construction, ankle joints fatigue 30–40% faster than hips because they bear ground-reaction forces (β term), while hip actuators (mapped to upper-leg bodies) see near-zero contact force. Not a surprise if you think about the physics, but it means the damage pattern comes from the model structure rather than being hand-tuned. Figure 1 shows this asymmetry.

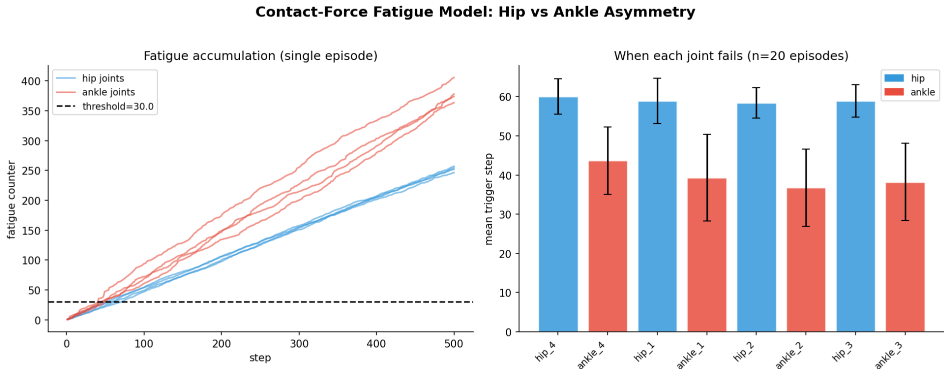

Figure 1: Contact-force fatigue model. **Left:** Per-joint fatigue over one episode. Ankle joints (red) cross the threshold before hip joints (blue) due to ground-reaction forces. **Right:** Mean trigger step confirms the asymmetry (ankles: ∼38 steps, hips: ∼59 steps).

One implementation detail worth noting: the actuator-to-body mapping uses `actuator_trnid` → `jnt_bodyid` indices, not names. MuJoCo 3.x doesn't expose actuator name arrays, and Ant-v4's actuator order (`hip_4, ankle_4, hip_1, ankle_1, ...`) isn't what you'd guess, so you have to look it up.

Domain randomization. Each episode samples a damage condition uniformly from $\{\text{healthy}, \text{mild}, \text{severe}\}$. Both models train on this same distribution. This is important: it means the encoder's advantage comes purely from being able to infer the current condition, not from seeing different data.

Getting the damage calibration right was one of the harder parts. At $\texttt{scale} = 0.25$ the ant barely slowed down (117→122 steps to reach the goal). At $\texttt{threshold} = 50$, damage triggered around step 88 but the ant was already at the goal by step 117. The final values ($\texttt{scale} = 0.1$, $\texttt{threshold} = 30$) ensure damage actually bites mid-journey.

3.2 Context Encoder

PEARL [Rakelly et al., 2019] conditions a policy on a latent z inferred from collected transitions to identify which task it's in. We adopt the same mechanism, but where PEARL's z captures reward-function identity, ours captures damage state. Like RL^2 [Duan et al., 2016], we use recurrence over the transition history to build this context, but with a fixed sliding window rather than full-episode state. The `ContextHistoryWrapper` maintains a sliding window of the last $k = 16$ (observation, action) pairs, flattened into the observation vector. The deque clears on every `reset()`, so history never spans episode boundaries. We implement the encoder as an LSTM (`LSTMContextExtractor`), which processes this window:

$$z_t = h_T, \quad h_1, \ldots, h_T = \text{LSTM}\big([(o_{t-k}, a_{t-k}), \ldots, (o_{t-1}, a_{t-1})]\big) \tag{3}$$

producing $z \in \mathbb{R}^{32}$ (the final hidden state), which gets concatenated with the current observation o_t before hitting the SAC actor and critic (Figure 2). The encoder trains end-to-end from actor and critic gradients, no auxiliary loss.

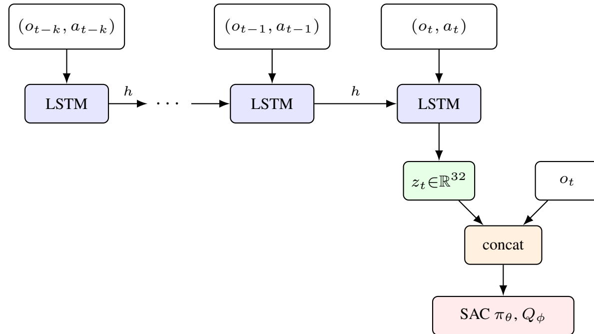

Figure 2: Architecture. The LSTM context encoder processes the last k=16 (observation, action) pairs; its final hidden state z_t is concatenated with the current observation and fed to the SAC actor and critic. Gradients from both networks train the encoder end-to-end.

The design choice that makes this work with stock SB3: we fold the history directly into the observation instead of building a custom sequence-sampling replay buffer. Every stored transition already carries its full 16-step context, so SB3's i.i.d. SAC [Haarnoja et al., 2018, Raffin et al., 2021] works unmodified. The tradeoff is $\sim$16$\times$ memory per transition in the buffer, which we offset with a smaller buffer (300k). The alternative (a custom buffer that reconstructs sequences at sample time) introduces episode-boundary bugs and breaks SB3's assumptions about replay. We tried it; history-in-obs was cleaner.

This is different from RMA's approach [Kumar et al., 2021], where the adaptation module is trained separately via teacher-student distillation with privileged access to environment parameters. Here, the encoder learns jointly with the policy, and z isn't supervised to predict any specific parameter. It learns whatever the RL objective pushes it toward.

3.3 Training

We use SAC [Haarnoja et al., 2018] because off-policy learning is critical here: the encoder's 621-dim observation means each transition is expensive to store, so we need the sample efficiency that comes from replaying past data. SAC's entropy regularization also helps with the exploration challenge

of learning to navigate under multiple damage regimes simultaneously. Implementation via Stable-Baselines3 [Raffin et al., 2021]. The baseline sees a 29-dim observation (27 Ant-v4 obs + 2 goal vector); the encoder sees 621 dimensions ($16 \times (29 + 8) + 29$). Training runs 500k timesteps on Modal A10G GPUs, about 45 minutes for the baseline (117 fps) and 102 minutes for the encoder (81 fps). Table 1 lists all hyperparameters.

Table 1: Hyperparameters. SAC defaults from SB3; damage and reward parameters tuned via ablation (see Table 4).

Category	**Parameter**	**Value**	**Rationale**
SAC	Learning rate	3×10^{-4}	SB3 default
	Batch size	256	SB3 default
	Replay buffer	300k	Reduced for 621-dim obs
	γ	0.99	SB3 default
	Hidden layers	2×256	SB3 default
	Entropy tuning	Automatic	SB3 default
Encoder	Context window k	16	Balance memory vs. context
	Latent dim $\|z\|$	32	Compact but expressive
Reward	w_f (forward weight)	0.5	Ablated: 0.0/1.0 failed
	w_p (progress weight)	50	Ablated: 30 too weak
	b_{succ} (completion)	200	Sparse goal signal
Damage	α (action weight)	1.0	Fatigue from torque
	β (contact weight)	0.1	Fatigue from ground force
	τ (threshold)	30	Ablated: 50 too late

4 Experimental Setup

Both models are trained using SAC [Haarnoja et al., 2018] via SB3 [Raffin et al., 2021] under identical domain randomization [Tobin et al., 2017, Peng et al., 2018] over damage severity, then evaluated across three conditions (healthy, mild, severe), 100 episodes per condition, using **deadline-based success metrics**:

- `success@250`: goal reached within 250 steps (tight)
- `success@350`: goal reached within 350 steps (moderate)
- `success@500`: goal reached within 500 steps (relaxed)

The reason for deadlines: without them, even a severely damaged ant can eventually limp to a nearby goal. The performance gap between healthy and damaged only shows up when you impose a time constraint. We also report mean final distance to goal and mean episode reward.

For the latent-space analysis, we collect rollouts across all conditions (6,247 timesteps total) and train linear probes on frozen z representations with a 70/30 train/test split.

5 Results

5.1 Quantitative Evaluation

Training progress. Figure 3 shows training curves over 500k steps. The encoder trains slower (81 vs. 117 fps due to LSTM overhead) but reaches comparable or higher episode reward. The context representation appears to help even during DR training, where the damage condition changes every episode.

Goal-reaching performance. Table 2 has the headline numbers. Under no damage, both models hit $\sim$83% success, so the encoder doesn't sacrifice healthy performance. Under severe damage, the baseline drops to 70%. The encoder holds at **86%**, a 16 percentage-point advantage.

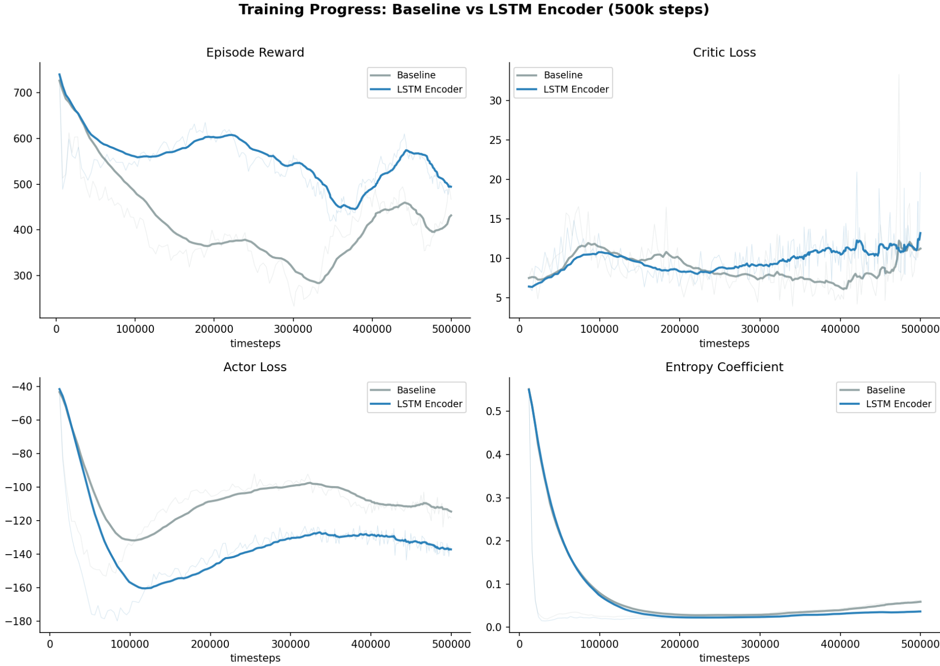

Figure 3: Training curves over 500k steps. **Top left:** Episode reward; the encoder (blue) reaches comparable or higher reward than the baseline (gray). **Top right:** Critic loss. **Bottom left:** Actor loss. **Bottom right:** Entropy coefficient, showing both models converge to similar exploration levels.

Table 2: Success rates (%) under deadline constraints (100 episodes per condition).

	Baseline (no memory)			Context Encoder		
Condition	@250	@350	@500	@250	@350	@500
Healthy	83	83	83	81	82	82
Mild	75	75	75	87	87	87
Severe	68	68	70	**86**	**86**	**86**

Standard errors for binary success rates at n=100 range from $\pm$3.4 to $\pm$4.6 percentage points (e.g., $86\% \pm 3.5$pp, $70\% \pm 4.6$pp). The encoder-vs-baseline gap under severe damage (16pp) clears two standard errors comfortably; the encoder's severe-vs-healthy difference (4pp) does not, so that could easily be noise.

Success rates are also nearly identical across the three deadlines, meaning successful episodes reach the goal well within 250 steps, and failures don't reach it at all. The one exception is baseline-severe, where two additional episodes squeak in between steps 250 and 500. The deadlines confirm the encoder's successes are fast, but they don't actually differentiate further here.

Table 3: Mean final distance to goal (lower is better) and mean reward across conditions.

	Mean Final Distance (m)		Mean Reward	
Condition	Baseline	Encoder	Baseline	Encoder
Healthy	1.25	1.26	216.6	303.2
Mild	1.44	**1.20**	567.8	569.9
Severe	1.70	**1.20**	755.8	667.2

The reward numbers look counterintuitive: mean reward *increases* with damage severity for both models. This is because damaged episodes run more steps before reaching the goal (or failing), so they accumulate more per-step reward from the native Ant forward-velocity and healthy-bonus

components. That's an artifact of the composite reward function, not evidence that damage helps. The success rate and final distance columns are what actually matter.

Recovery fraction. To quantify adaptation: how much of the baseline's degradation does the encoder recover?

$$\text{RF} = \frac{\text{Encoder}_{\text{severe}} - \text{Baseline}_{\text{severe}}}{\text{Baseline}_{\text{healthy}} - \text{Baseline}_{\text{severe}}} = \frac{86 - 70}{83 - 70} = 1.23 \tag{4}$$

A recovery fraction above 1.0 means the encoder doesn't just close the gap, it overshoots. The encoder under severe damage (86%, 1.20m to goal) actually outperforms the baseline under *no* damage (83%, 1.25m). We'll address why in the Discussion.

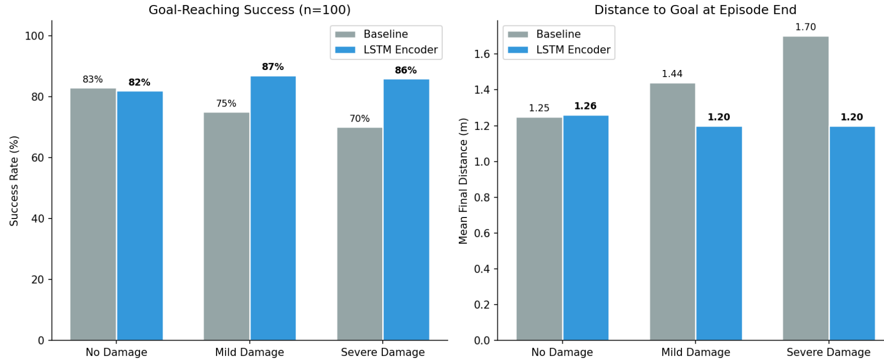

Figure 4: Headline comparison. **Left:** Success rate across conditions. **Right:** Mean final distance to goal (lower is better). The encoder maintains or improves under damage where the baseline degrades.

5.2 Qualitative Analysis

Recovery dynamics. Figure 5 aligns goal progress to the moment of first joint failure (n=20 rollouts). Both models show reduced progress after damage onset, but the encoder maintains a higher approach rate than the baseline, particularly under severe damage. Unlike Cully et al. [2015], where the robot needs dedicated trial-and-error periods to identify the right compensatory behavior, our encoder adjusts within the task episode itself, with no separate adaptation phase.

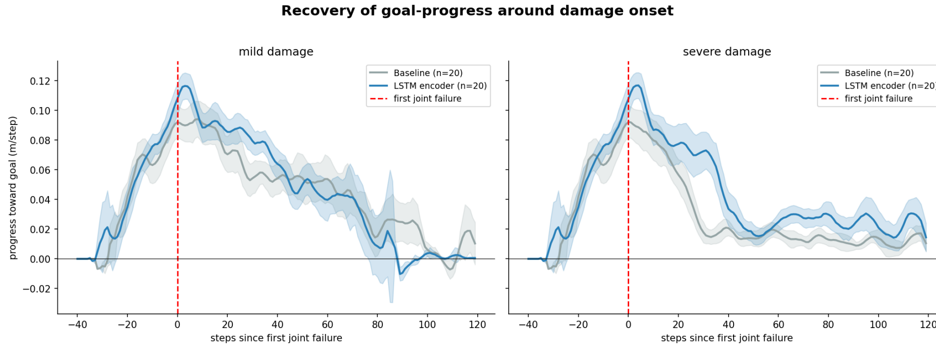

Figure 5: Goal-progress rate aligned to damage onset (step 0 = first joint failure, n=20 rollouts). Both models show reduced progress after damage, but the encoder (blue) maintains a higher rate than the baseline (gray), particularly under severe damage.

Latent space analysis. The recovery results show the encoder *adapts*, but is it actually tracking the damage, or just learning a more robust average policy? Joint masking approaches [Kim et al., 2024, Qiu et al., 2025] never need to answer this question because they explicitly tell the policy which joints are impaired. Since our encoder gets no such signal, we have to verify that z genuinely encodes damage state. Three probing experiments on the trained encoder's rollouts (6,247 timesteps) answer this.

Linear probing. A linear probe on frozen z vectors gets **75.1%** on 3-way damage classification (healthy/mild/severe), versus 65.0% from raw observations and 33.3% chance. The gap between z and raw obs is the important number: it means the encoder extracts information that isn't in any single observation. You need the temporal pattern of "commanded action → observed outcome" over multiple steps to distinguish mild from severe.

On binary detection (damaged vs. not): 86.7% from both z and raw obs. Binary damage *is* partially visible in a single frame, the ant just moves differently. But telling *how badly* it's damaged requires history.

Per-joint decodability. A multi-label probe hits **95.5%** per-joint accuracy at predicting which of 8 joints are currently damaged, from z alone (majority-class baseline: 20.8%). The encoder doesn't just know "something is wrong," it knows *which joints* are broken.

PCA visualization and onset tracking. Figure 6 shows (left) $\|z - \bar{z}\|$ (distance from mean latent) aligned to damage onset: the latent representation shifts within 5–10 steps of the first joint failure. For comparison, RMA's adaptation module [Kumar et al., 2021] also adapts online, but is supervised by a teacher with ground-truth parameters. Our encoder reaches a similar detection speed from RL gradients alone. On the right, PCA of z colored by number of damaged joints shows clear cluster separation.

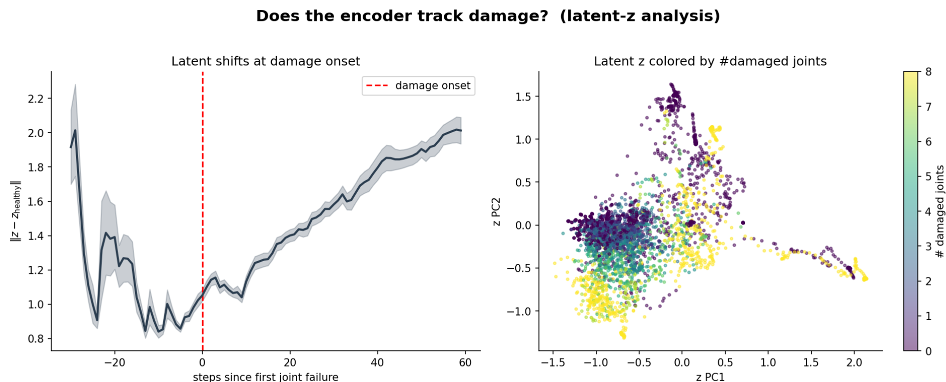

Figure 6: **Left:** Latent shift at damage onset: $\|z - \bar{z}\|$ (distance from mean latent) rises within 5–10 steps of the first failure. **Right:** PCA of z colored by number of damaged joints shows clear separation between healthy (0) and damaged (≥4) regimes.

6 Discussion

What the encoder actually learns. The 3-way probe gap (z: 75.1% vs. raw obs: 65.0%) is the core evidence. The encoder extracts fault information that isn't available in any single observation. This is the same partial-observability argument that motivated DRQN [Hausknecht and Stone, 2015]: some state variables (there, enemy positions; here, damage levels) are hidden and can only be inferred from temporal patterns. The damage state has to be inferred from the discrepancy between what the policy commands and what the dynamics produce, accumulated over multiple timesteps. What's notable is that this works without any auxiliary loss. RMA [Kumar et al., 2021] requires a teacher with privileged access to ground-truth environment parameters to supervise the adaptation module. PEARL [Rakelly et al., 2019] uses a variational inference objective to shape its context variable. Our encoder needs neither: the critic's gradient alone is enough to push z toward damage-relevant features, because knowing the damage state directly improves value estimation.

Why severe outperforms healthy. The encoder hits 86% under severe damage but 82% under no damage. That seems backwards, and honestly, given the standard errors (±3.5–3.8pp at n=100), this 4pp gap could just be noise. But it's a consistent pattern across all 100 episodes, so here's one way to think about it: under severe damage, the affected joints are essentially locked (s=0.1), creating a constrained but *highly predictable* dynamical regime. Once the encoder identifies the fault (which

probing shows happens within 5–10 steps), it can commit fully to a specialized gait for that reduced actuator set. Under no-damage conditions, the DR training means the encoder has seen plenty of damaged episodes, so during the healthy initial phase there's slight ambiguity in z ("is this healthy, or has damage just not triggered yet?"), leading to occasional suboptimal early choices. We'd need more episodes or a controlled experiment to confirm this, but the intuition holds: certainty about a bad state beats ambiguity about a good one.

On the deadlines. As the tables show, the three deadline thresholds produced almost no differentiation here. The deadlines confirm that successes are fast, but a harder task (further goal, tighter budget) would be needed to actually see graded differences in adaptation speed. A limitation of the evaluation, not the approach.

What failed along the way. Table 4 summarizes the design decisions we validated by trying the alternative and watching it break:

Table 4: Design decisions validated through ablation. Each row is something that failed and why.

Final choice	**Failed alternative**	**What went wrong**
`fw_weight` $= 0.5$	$= 0.0$	No locomotion incentive; ant stands still
	$= 1.0$	East detour from +x bias; encoder can't assign credit
`damage_scale` $= 0.1$	$= 0.25$	Ant barely slowed (117→122 steps)
`damage_threshold` $= 30$	$= 50$	Triggers at step 88; goal already reached by step 117
`progress_weight` $= 50$	$= 30$	Too weak to steer past native +x bias for diagonal goal
DR training (both models)	No-damage training	Encoder never sees faults; nothing to learn
History-in-obs	Custom replay buffer	Episode-boundary bugs; breaks SB3's SAC

The biggest lesson: the damage parameters have to be calibrated so damage *actually impairs* the agent mid-journey. If the ant reaches the goal before damage bites, the whole fault-tolerant framing collapses, there's no gap for the encoder to close.

Limitations. We want to be upfront about what this doesn't show:

- The fatigue model is simulated. Real damage (backlash, stiction, partial failures) is messier than clean torque scaling.
- The binary probe gets 86.7% from both z *and* raw observations, so some fault signal does leak into single frames. The encoder's advantage is specifically in *severity* discrimination, not just detection.
- We only compare against a memoryless baseline. The obvious missing comparison is a feedforward encoder (e.g., an MLP over concatenated frames from the same 16-step window), which would tell us whether it's the LSTM's sequential processing that matters or just having any function of the history. Without that, we can't fully attribute the gains to recurrence.
- Single morphology (Ant-v4, 8 actuators), single task (goal-reaching). Whether this transfers to higher-DOF systems or different tasks is open.
- 500k training steps is likely not enough for full convergence, since 83% healthy success leaves room for improvement with longer training or curriculum learning.

7 Conclusion

A context encoder (implemented as an LSTM), trained end-to-end with SAC [Haarnoja et al., 2018] under domain-randomized damage, can implicitly detect progressive joint damage from dynamics alone. The latent code tracks damage at joint-level resolution (95.5% per-joint accuracy) and enables a policy that not only recovers from severe actuator degradation (86% vs. 70% baseline) but outperforms the healthy-condition baseline under damage. No privileged fault labels, no auxiliary losses, no separate adaptation phase. Just history in the observation and standard RL gradients.

The natural next steps: (1) auxiliary losses to speed up encoder convergence, (2) comparison against feedforward encoders to isolate the value of sequential processing, (3) out-of-distribution damage types not seen during training, and (4) transfer to real hardware.

Team Contributions

All work on this project (environment design, damage model, encoder architecture, training infrastructure, evaluation, analysis, and report) was completed by yours truly (Sidhanth Mishra).

References

A. Cully, J. Clune, D. Tarapore, and J.-B. Mouret. Robots that can adapt like animals. *Nature*, 521(7553):503–507, 2015. doi:10.1038/nature14422.

Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel. RL^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016. OpenReview.

T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft Actor-Critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*, 2018. arXiv.

M. Hausknecht and P. Stone. Deep recurrent Q-learning for partially observable MDPs. *arXiv preprint arXiv:1507.06527*, 2015. arXiv.

M. Kim, U. Shin, and J.-Y. Kim. Learning quadrupedal locomotion with impaired joints using random joint masking. *arXiv preprint arXiv:2403.00398*, 2024. arXiv.

A. Kumar, Z. Fu, D. Pathak, and J. Malik. RMA: Rapid motor adaptation for legged robots. In *RSS*, 2021. arXiv.

X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *ICRA*, 2018. arXiv.

A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. Stable-Baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. JMLR.

K. Rakelly, A. Zhou, C. Finn, S. Levine, and D. Quillen. Efficient off-policy meta-reinforcement learning via probabilistic context variables. In *ICML*, 2019. arXiv.

J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *IROS*, 2017. arXiv.

E. Todorov, T. Erez, and Y. Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. IEEE Xplore.

Y. Qiu, X. Lin, R. Li, D. Zhang, J. Wang, X. Li, B. Du, T. Nguyen, and L. Qi. UMC: A unified approach for resilient control of legged robots across masked malfunction training. *arXiv preprint arXiv:2502.03035*, 2025. arXiv.