UNDERSTANDING, STABILIZING AND ACCELERATING HIGH-RATIO N:M SPARSE TRAINING IN DEEP REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Row-wise $N{:}M$ sparsity has emerged as a promising approach to accelerate matrix-matrix multiplications. However, the behavior of this type of sparsity remains under-explored in deep reinforcement learning (RL). Although magnitude-based $N{:}M$ sparsity implementations perform well at lower sparsity levels, they fail to maintain performance as sparsity increases. In this paper, we systematically investigate, stabilize, and accelerate high-ratio row-wise $N{:}M$ sparse training in continuous-control RL. Under static 1:8 sparsity, randomly sampled initial mask topology has little predictive power for final return, whereas lower Fraction of Active Units (FAU) in the dense critic generally accompanies smaller sparse-dense return gaps. Under dynamic sparsity, we identify a three-stage actor saturation collapse and a mask-update shock in the critics, and counter them with scale-tuned initialization and transient replay-ratio increases. With increased width and a higher replay ratio, stabilized 1:8 agents achieve higher mean returns than same-width dense agents in 13 of 18 environment-buffer configurations. We further introduce time-multiplexed $N{:}M$ training, which co-trains adjacent discrete patterns through shared parameters while keeping each subnetwork deployable at inference. On Walker2d, both subnetworks of a 3.5:8 schedule, trained with 12.5% fewer active weights than fixed 4:8, exceed independently trained 3:8 and 4:8 networks in mean return. Finally, our custom CPU sparse kernels provide 2.17-3.95$\times$ speedups for the forward pass of 1:8 layers and, combined with an approximate active-weight-only Adam optimizer, 1.34-1.83$\times$ end-to-end training speedups over an MKL-backed dense-kernel reference.

Keywords: Semi-Structured Sparsity, Dynamic Sparse Training, Continuous Control, Efficient Reinforcement Learning, Training Acceleration

1 INTRODUCTION

Semi-structured $N{:}M$ sparsity bridges the flexibility of fine-grained unstructured connectivity and the execution efficiency of structured sparsity. An $N{:}M$ sparse matrix retains at most N non-zero weights per contiguous block of M. Skipping pruned weights offers an idealized M/N reduction in matrix-multiplication arithmetic.

While $N{:}M$ sparse training has demonstrated strong results in supervised learning, its behavior in deep reinforcement learning (RL) remains less understood. RNM-TD3 (Vrce et al., 2026) introduced row-wise sparsity across TD3 networks, with periodic mask updates based on within-block magnitude ranking. Although moderate patterns such as $2{:}4$ and $1{:}4$ match or exceed dense baselines on several MuJoCo continuous-control tasks, highly sparse regimes such as $1{:}8$ exhibit instability and performance degradation in more complex environments.

Motivated by these limitations, we ask two questions:

Q1. *Why does high-ratio $N{:}M$ sparse training fail in reinforcement learning?*

Q2. *Can it be made both performant and faster in wall-clock time?*

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Our contributions follow a progression from diagnosis to stabilization, scaling, flexible sparsity and, finally, measured wall-clock acceleration:

- **Diagnosis of the failure modes.** When the masks from the best and worst of 100 random static 1:8 runs are retrained, they reach similar mean returns, indicating that, for randomly sampled row-wise masks, the initial topology has little predictive power for final return. Instead, we identify (i) a *capacity bottleneck*, generally more severe in environments whose dense critic has a higher Fraction of Active Units (FAU); (ii) a three-stage *actor saturation collapse*, in which growing activations in the last hidden layer saturate the actor's output, suppressing gradient propagation until mask updates stall; and (iii) *mask-update shock*, in which a single mask update abruptly increases the critic's TD error and the disagreement between the twin critics (Sections 3 and 4; Appendix C).
- **Targeted stabilization.** Reducing the He initialization scale mitigates saturation: averaged over six environments and four mask-update periods, the mean dense-normalized return rises from 0.589 at the standard scale to 0.685 at a $3\times$ smaller one, and drops to 0.255 at a $3\times$ larger one. A transient replay-ratio increase after each mask update targets mask-update shock, improving mean return in four of six environments (by up to 34% on Hopper; Section 4.1).
- **Joint width-sparsity scaling.** To relieve the capacity bottleneck while keeping the $1{:}8$ pattern, we scale width instead of density. Widening static $1{:}8$ networks from 256 to 724 neurons improves returns in all six environments and surpasses the dense-256 baseline in four, at comparable or lower parameter counts (-30.5% to $+1.3\%$; Section 3.2). At width 512 and replay ratio 2, dynamically sparse agents achieve higher mean returns than width- and replay-matched dense baselines in 13 of 18 environment-buffer comparisons (Section 4.2).
- **Flexible sparsity via time-multiplexing.** Because capacity demands vary across environments while discrete $N{:}M$ patterns allow only coarse density steps, we alternate between adjacent patterns (e.g., $3{:}8$ and $4{:}8$) with shared parameters and use the duty cycle to set an intermediate average training density, while every training step keeps a strict row-wise $N{:}M$ structure. The constituent subnetworks are co-trained through shared parameters and remain individually deployable: on Walker2d, both subnetworks of a 3.5:8 schedule, which trains with 12.5% fewer active weights than fixed 4:8, achieve higher mean returns than independently trained 3:8 and 4:8 networks (Section 5).
- **Measured wall-clock acceleration.** To verify that the arithmetic savings of $N{:}M$ sparsity translate into real training time, we implement CPU kernels that skip pruned weights by exploiting the row-wise block structure and integrate them into dynamic $1{:}8$ TD3. Across six environments and complete 1M-step training runs, this yields 1.34-1.83$\times$ end-to-end speedups over an MKL-backed dense-kernel reference, without dedicated sparse hardware (Section 6).

2 Related Work

Policy distillation transfers behavior from a pretrained policy to a smaller student (Rusu et al., 2016; Hinton et al., 2015), while pruning-based compression removes parameters and fine-tunes the remaining sparse network (Han et al., 2015; Livne & Cohen, 2020). Although these approaches expose redundancy in the original agent, they do not remove the cost of its initial training. Motivated by the Lottery Ticket Hypothesis (Frankle & Carbin, 2019), RL studies show that identified sparse subnetworks can be retrained to attain returns comparable to dense agents (Yu et al., 2020; Vischer et al., 2022). Dynamic Sparse Training (DST) instead treats connectivity as part of the optimization process. Representative methods periodically prune low-magnitude connections and regrow them randomly or using gradient information (Mocanu et al., 2018; Evci et al., 2020). In RL, sparse-training approaches improve trade-offs between compression and return (Tan et al., 2023; Sokar et al., 2022; Arnob et al., 2024), demonstrating the potential of adapting connectivity as the policy and replay buffer evolve.

In continual and non-stationary RL, networks can gradually lose their ability to adapt to changing data distributions. This phenomenon, known as *loss of plasticity*, has been documented in prior

work (Dohare et al., 2024; Abbas et al., 2023) and is sensitive to the replay ratio (Ma et al., 2024). We use the terms replay ratio (RR) and update-to-data ratio (UTD) interchangeably. A related line of research examines how networks use their available capacity. In particular, dormant neurons exhibit low normalized activation and can remain inactive for extended periods during training (Sokar et al., 2023).

Several methods address these capacity and plasticity challenges. ReDo recycles dormant neurons (Sokar et al., 2023), while Neuroplastic Expansion combines network growth with an experience review stabilizer (Liu et al., 2025). Our work complements these approaches by using dense-network FAU as a diagnostic tool to investigate how capacity requirements vary across environments under high sparsity. Further discussion of dormancy, replay ratio and FAU is provided in Appendix G.

Sparse RL methods such as DS-TD3 and RLx2 use unstructured connectivity to improve the trade-off between policy performance and parameter count (Sokar et al., 2022; Tan et al., 2023). However, irregular memory access can limit the translation of parameter savings into proportional wall-clock gains (Mishra et al., 2021). Structured and semi-structured methods instead exploit regularity to enable efficient execution (Wen et al., 2016; Zhou et al., 2021). Accelerator support for patterns such as 2:4 illustrates the practical potential of $N{:}M$ sparsity (Mishra et al., 2021). RNM-TD3 (Vrce et al., 2026) introduced row-wise $N{:}M$ sparsity to TD3. Building on that framework, we investigate high-sparsity trainability, capacity requirements and stabilization, use time-multiplexing between adjacent $N{:}M$ patterns to obtain intermediate training densities, and evaluate end-to-end training acceleration.

3 STATIC N:M SPARSITY

To address Q1, we begin with static masks: fixing the topology throughout training isolates the roles of initial topology and network capacity from the mask-update dynamics studied in Section 4. Our results point to network capacity, rather than the randomly sampled initial topology, as the main limitation of static 1:8 performance.

3.1 INITIAL TOPOLOGY

For each of the six environments, we trained 100 static 1:8 agents with randomly sampled masks held fixed throughout training. Figure 1 overlays their final-return distributions (blue) with the dense baselines (orange). To assess whether the initial topology explains the observed variation, we selected the masks from the highest and lowest return sparse runs. We then retrained each donor mask over 40 independent seeds with otherwise identical settings.

Table 1 shows that reusing the best and worst donor masks yields similar mean returns within each environment. On Walker2d, donor returns ranged from approximately 360 to 5300, yet retraining their masks produced means of 2120.2 and 2121.6, respectively. These results suggest that, within the evaluated distribution of randomly sampled row-wise masks, initial topology is not a strong determinant of final performance. This does not exclude benefits from explicitly optimized or structured mask selection, such as the iterative magnitude pruning studied by Vischer et al. (2022).

3.2 SCALING NETWORK SIZE IN STATIC SPARSITY

Since the initial topology does not explain the return deficit of static 1:8 agents, we investigate whether they lack capacity. In our Walker2d static-sparsity sweep, mean-return degradation becomes apparent at 3:8 and increases at 2:8 and 1:8, suggesting a limitation in *network capacity*. Full seed-wise return distributions are reported in Appendix B (Figure 7). To investigate environment-dependent sensitivity to pruning, also observed in prior work (Yu et al., 2020), we use the Fraction of Active Units (FAU) as an empirical diagnostic of capacity utilization. FAU is the fraction of positive hidden-layer ReLU activations across neurons and samples in a replay-buffer mini-batch. Figure 2(a) compares dense actor and critic FAU at widths 256 and 724.

At width 256, lower dense-critic FAU generally accompanies higher 1:8 sparse-to-dense return ratios, shown by the blue circles in Figure 2(b). Humanoid and Ant have the lowest critic FAU (28.2% and 34.9%) and retain 86.4% and 72.3% of dense returns, respectively. Conversely, Hopper has the highest critic FAU (50.4%) and the lowest sparse-to-dense return ratio (51.7%). Across the

Table 1: Static 1:8 returns (mean and SD) for random and reused best/worst donor topologies; initialization scale 0.5.

Environment	Topology	Runs	Mean	SD
Ant	Random	100	3199.9	686.33
	Best	40	3398.2	829.59
	Worst	40	3351.1	706.98
HalfCheetah	Random	100	6859.5	510.50
	Best	40	6960.4	583.47
	Worst	40	6951.7	552.59
Hopper	Random	100	1583.1	1021.2
	Best	40	1817.0	971.89
	Worst	40	1811.4	1032.2
Humanoid	Random	100	4285.5	1151.0
	Best	40	4258.4	1018.6
	Worst	40	4131.8	1224.7
Swimmer	Random	100	52.878	10.879
	Best	40	51.933	6.9840
	Worst	40	53.777	13.010
Walker2d	Random	100	2121.3	872.90
	Best	40	2121.6	1010.4
	Worst	40	2120.2	882.47

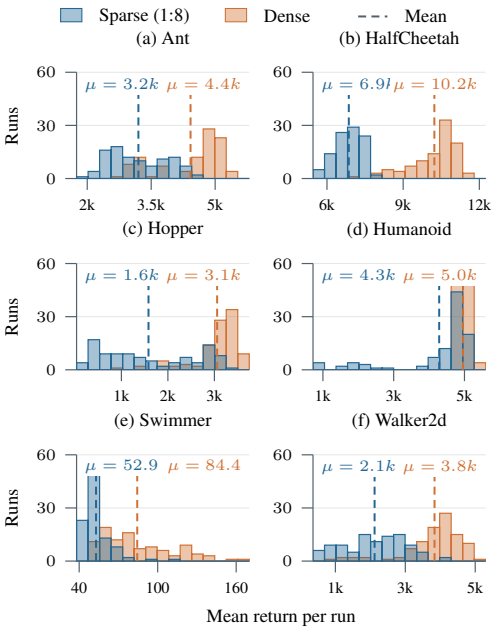

Figure 1: Return distributions ($n = 100$). Blue: static 1:8; orange: dense.

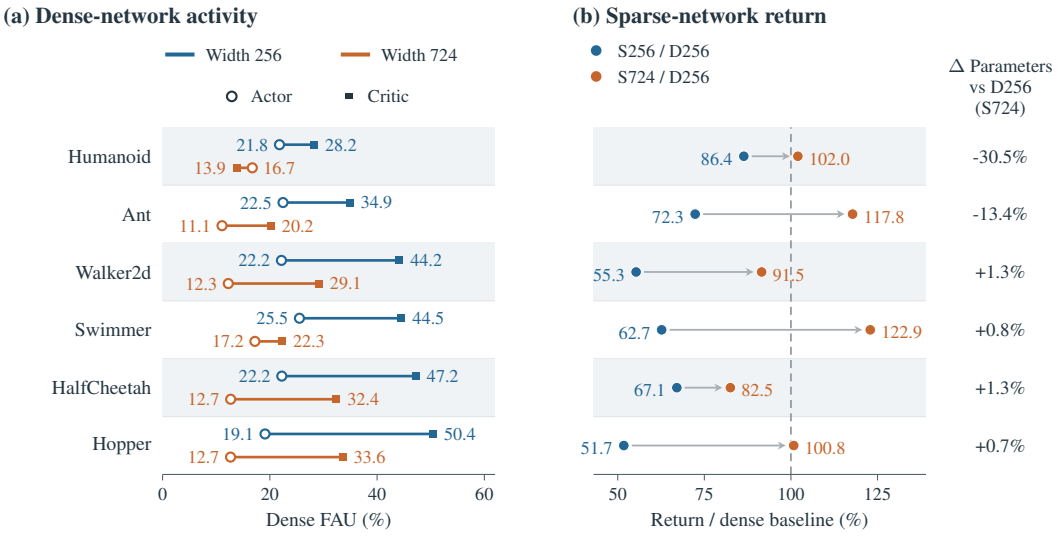

Figure 2: Dense-network activity and sparse-network performance. **(a)** Dense actor and critic FAU at widths 256 and 724. **(b)** Mean-return ratios S256/D256 and S724/D256 (circles). Arrows compare sparse widths against D256; the dashed line marks return parity (100%). The rightmost column reports the parameter-count change of S724 relative to D256. S and D denote static 1:8 sparse and dense networks; numbers indicate hidden-layer width.

six environments, dense-critic FAU and the 1:8 sparse-to-dense return ratio exhibit a negative rank correlation (Spearman $\rho = -0.77$, exact two-sided permutation $p = 0.10$), whereas dense-actor FAU shows no such association ($\rho = 0.20$). Given the small number of environments, we interpret this result as evidence of association between lower dense critic FAU and smaller performance gap between sparse and dense agents.

Increasing width from 256 to 724 is accompanied by lower actor and critic FAU in the dense networks (Figure 2(a)) and higher mean returns in the static 1:8 sparse networks (Figure 2(b)) in all six environments. The arrows compare the two sparse widths against the same dense-256 baseline. The wider sparse networks exceed this baseline in four of six environments, with parameter-count

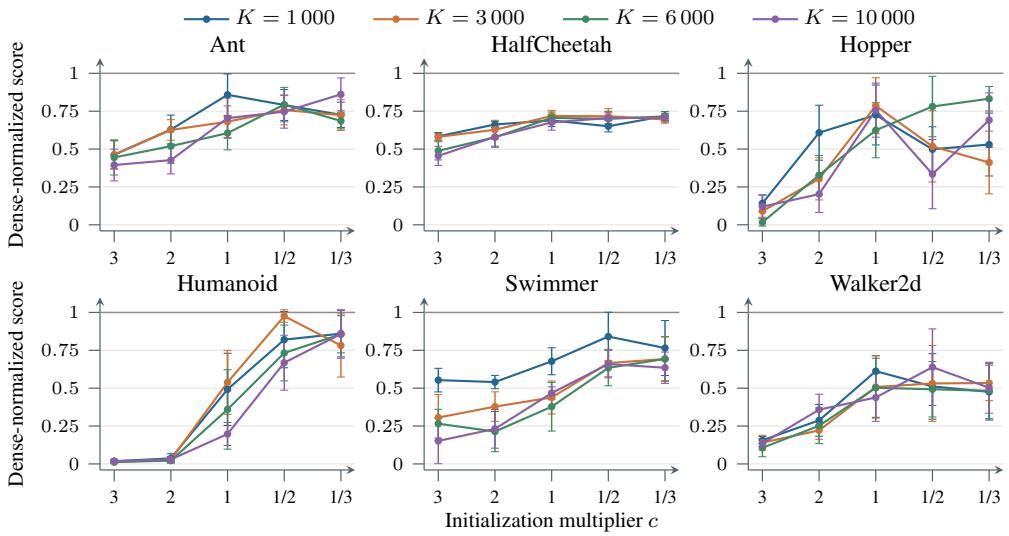

Figure 3: Dense-normalized final-window reward for dynamic 1:8 TD3. Panels are environments, curves are mask-update periods, and each point averages 11 seeds. Bars are approximate 95% t intervals conditional on the plotted dense normalizer.

differences ranging from -30.5% to $+1.3\%$ relative to dense-256. These results motivate considering width and sparsity jointly: increasing width may alleviate representational constraints under high sparsity. This is consistent with Ma et al. (2025), who find that one-shot random unstructured sparsity allows scaling beyond dense counterparts. Here we observe a similar effect under row-wise N:M constraints.

Finally, critic FAU exceeds actor FAU in all six environments at width 256. Together with the scaling results, this asymmetry motivates investigating whether the critic benefits from greater capacity allocation than the actor. This interpretation is qualitatively consistent with findings that smaller actors can preserve performance (Mysore et al., 2021) and that critic scaling can offer greater benefits than actor scaling in the evaluated settings (Lee et al., 2025; Nauman et al., 2024b).

4 Dynamic N:M Sparsity

Section 3 points to limited capacity, rather than the initial topology, as the main limitation of static 1:8 agents. Static masks, however, cannot adapt connectivity as the policy and the replay data change. Dynamic row-wise N:M sparsity instead refreshes masks at predefined intervals. Each refresh selects the N largest-magnitude weights within every block of M. This allows connectivity to evolve while preserving the sparsity structure, but requires the agent to adapt to structural changes alongside ongoing parameter updates. We identify two failure modes under dynamic sparsity, which complete our answer to Q1.

Actor saturation. Early in training, saturated actor neurons can produce nearly identical actions across different states. The method in Vrce et al. (2026) mitigates this issue through random mask resampling when Sparse Architecture Divergence (SAD) (Zhou et al., 2021) falls below a critical threshold. However, this intervention provides a mechanism for recovery rather than preventing saturation.

Mask-update shock. Avoiding early actor saturation does not by itself ensure dense-baseline performance. Repeated mask changes can perturb network outputs (Appendix C) and disrupt learning, leaving training at low returns even without complete policy collapse.

4.1 Stabilizing training

To analyze these failure modes, we monitor the actor with three diagnostics that define three severity stages; they motivated initialization-scale tuning but are not used to trigger interventions. *Potential instability*: the RMS of the final hidden-layer activations grows, signaling saturation pressure at the

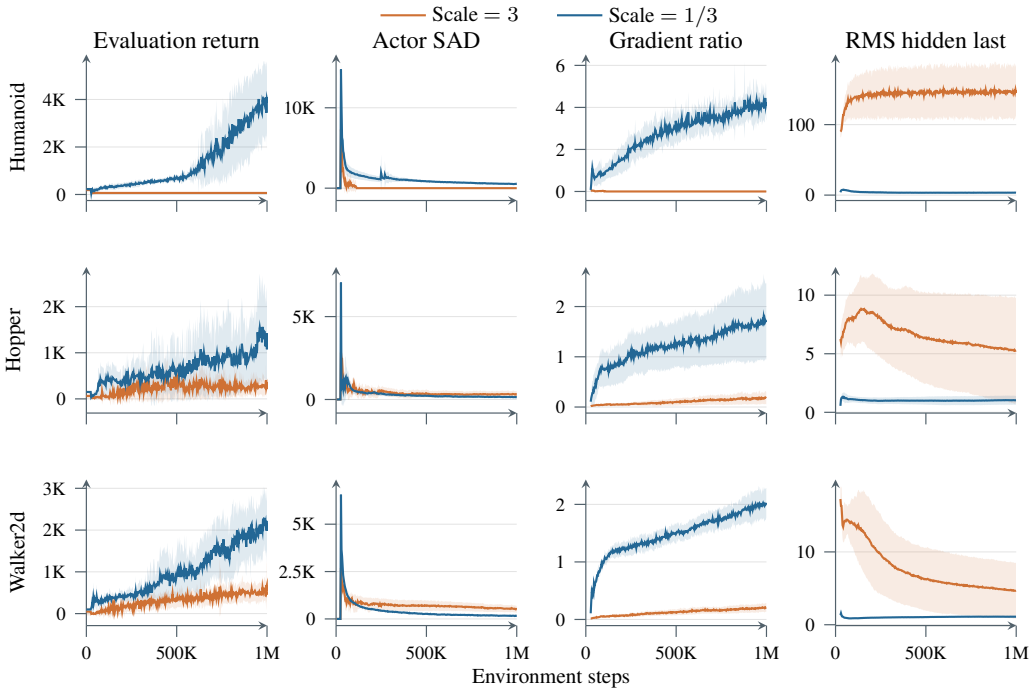

Figure 4: Evaluation return and actor diagnostics under dynamic 1:8 sparsity over 1M environment steps, with mask updates every 3,000 steps. Rows show Humanoid, Hopper and Walker2d; columns show return, SAD, gradient ratio and final hidden-layer RMS. Orange and blue denote initialization scales 3 and $1/3$, respectively. Curves show means over 11 seeds; shaded bands indicate ± 1 standard deviation.

$\tanh$ output. *Probable collapse*: the ratio of gradient norms between the actor's final and initial layers falls, indicating impaired gradient propagation. *Policy collapse*: saturated output neurons yield a nearly state-independent action mapping, parameter updates become negligible, within-block magnitude rankings stop changing and mask updates reselect the same connections ($\mathrm{SAD} = 0$). Mask stagnation is thus a consequence of collapse rather than its cause, and the soft reset of RNM-TD3 acts only at this terminal stage.

Actor saturation can be substantially mitigated by reducing the initialization scale. Figure 4 compares He (Kaiming) initialization scaled by 3 and $1/3$ in Humanoid, Hopper and Walker2d. On Humanoid, at scale 3, the RMS metric increases rapidly, immediately driving the gradient ratio to zero and causing rapid SAD collapse, coinciding with stalled learning. At scale $1/3$, activations remain smaller, the gradient ratio increases and SAD decays more slowly as returns improve. Hopper and Walker2d show similar activation and gradient trends, supporting the role of initialization scale in mitigating saturation. Consistent with these diagnostics, Figure 3 shows that smaller initialization multipliers generally improve normalized returns across the evaluated environments. However, the best-performing scale factor and mask-update period remain environment-dependent. Averaging equally over the 24 environment-period combinations yields normalized scores of 0.255, 0.362, 0.589, 0.674 and 0.685 for $c = 3$, $c = 2$, $c = 1$, $c = 1/2$ and $c = 1/3$, respectively (Appendix A.3, Table 5).

While periodic mask updates are intended to explore potentially superior network topologies, they can temporarily destabilize training. In particular, newly activated connections after a mask change introduce abrupt structural changes that may cause sudden shifts in the actor's outputs and fluctuations in the critic's Q-value estimates. To circumvent this transient instability, we temporarily increase the replay ratio (RR) immediately after each mask update. Following each mask update, we set $RR_{\text{transient}} = 8$ for the first 10% of the subsequent interval, then restore $RR = 1$. These additional updates are intended to help the actor and critic adapt to newly activated connections and reduce transient disturbances to policy outputs and value estimates.

Table 2 reports positive mean-return changes in four environments, little change on Ant and a decrease on Swimmer. With initialization scale 0.5 and environment-specific mask-update periods,

Table 2: Final returns of dynamic 1:8 agents (width 256, initialization scale 0.5, 1M buffer; mean over 25 runs) without a transient replay-ratio increase (RR = 1) and with the replay ratio raised after each mask update to 4 for the first 5% or to 8 for the first 10% of the mask-update interval. Dense (%) is the RR = 8 mean as a percentage of the dense baseline. Mask-update periods are selected per environment from the sweep in Figure 3 (1000 for Swimmer, 3000 for Humanoid, 6000 for Ant, Hopper and HalfCheetah, 10000 for Walker2d).

	Dynamic 1:8				
Environment	RR = 1	RR = 4, 5%	RR = 8, 10%	Dense	Dense (%)
Ant	3605	3307	3602	4422	81.46
HalfCheetah	7519	7482	8009	10228	78.30
Hopper	1960	2422	2631	3059	86.01
Humanoid	4153	4159	4751	4960	95.79
Swimmer	71.10	70.95	69.25	84.38	82.07
Walker2d	2221	2315	2428	3836	63.32

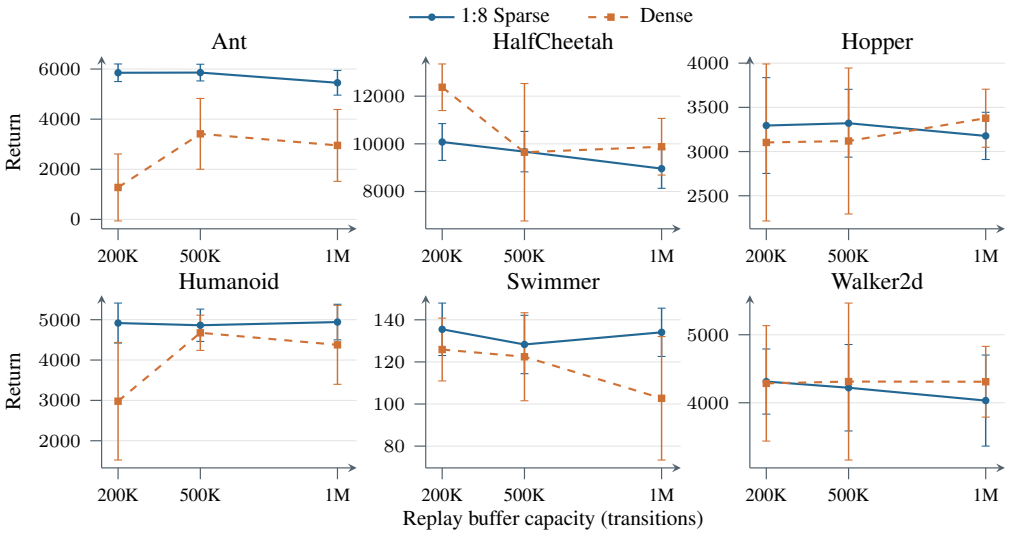

Figure 5: Final returns of dynamically sparse 1:8 and dense TD3 agents with hidden-layer width 512 and RR = 2. Points and error bars show means ± 1 standard deviation across 20 seeds, with 2,000 evaluation episodes at the end of the training per seed.

the resulting sparse agents attain 63.3%-95.8% of their dense baselines, with Humanoid closest to parity and Walker2d furthest.

4.2 Scaling Network Size under Dynamic Sparsity

With both failure modes mitigated, width-256 1:8 agents reach 63.3-95.8% of their dense returns (Table 2). Section 3.2 suggests that the remaining gap may reflect limited capacity, so we apply the same remedy under dynamic sparsity, scaling width rather than density, and combine it with a higher replay ratio and smaller replay buffers.

Figure 5 compares 1:8 sparse and dense TD3 networks across three replay buffer capacities (200K, 500K and 1M), with the hidden-layer width fixed at 512 and the replay ratio of 2. Sparse agents achieve higher mean returns than their same-width dense counterparts on Ant, Humanoid and Swimmer at every tested buffer capacity. On HalfCheetah, Hopper and Walker2d, the outcome depends on buffer capacity. Overall, sparse agents achieve higher mean returns in 13 of the 18 comparisons. These results indicate that increased network width with a higher replay ratio can substantially mitigate the representational constraints imposed by 1:8 sparsity and allow dynamically sparse agents to match or exceed same-width dense agents.

5 TIME-MULTIPLEXED N:M TRAINING

Because N is an integer, N:M sparsity offers only M density levels, spaced $1/M$ apart; for $M = 8$, moving from 1:8 to 2:8 doubles the number of active weights. To reach intermediate densities without giving up the N:M structure, we time-multiplex two adjacent patterns, N:M and $(N{+}1)$:M, that share parameters. We switch patterns only at mask updates, so every training step uses a strict row-wise N:M mask, while the duty cycle, the fraction of training steps spent in the denser pattern, sets the average number of active weights. We denote a schedule by its effective ratio: for example, 2.5:8 denotes equal numbers of training steps at 2:8 and 3:8. The effective ratio describes the training schedule, not a deployable pattern. At inference, either constituent subnetwork can be selected to trade return against computational cost.

The principle of alternating between weight-sharing sparse subnetworks also underlies alternating sparse training (AST; Yang et al., 2022), which trains subnetworks spanning a wide range of sparsity levels to obtain several deployable models from one training run. AST switches subnetworks at every mini-batch and applies gradient correction to reduce interference between them. Our schedule does neither: we alternate between two adjacent patterns with the same M only at mask-update boundaries, and the duty cycle sets the average training density. We therefore compare time-multiplexed schedules against fixed-pattern training at matched and higher density.

Tables 3 and 4 summarize the Walker2d sparsity sweep and the higher-replay-ratio experiments across six environments, respectively.

Table 3: Discrete and time-multiplexed sparsity on Walker2d. Results are averaged over 50 independent runs. RR = 1; mask-update period: 5,000 steps.

Sparsity	Mean			SD
	Lower	Upper	Final	
4:8	—	—	3645.3	926.39
3.5:8	3772.0	3858.1	3815.0	738.43
3:8	—	—	3440.5	946.85
2.5:8	3075.2	3426.7	3251.0	937.71
2:8	—	—	2999.4	1013.2
1.5:8	1592.4	2622.2	2107.3	928.06
1:8	—	—	1754.3	981.03

Table 4: Non-discrete 1.5:8 sparsity across environments. RR = 4; mask-update period: 6,000 steps; 25–27 seeds.

Environment	Mean			SD
	Lower	Upper	Final	
Ant	3774.6	4269.0	4021.8	800.02
HalfCheetah	9065.6	9512.9	9289.3	794.73
Hopper	2853.1	2979.1	2916.1	886.05
Humanoid	5078.0	5081.1	5079.6	176.07
Swimmer	30.422	32.309	31.366	20.747
Walker2d	3322.2	3719.2	3520.7	825.00

Note. Both experiments use 256 units per hidden layer. For time-multiplexed schedules, Lower and Upper denote the constituent discrete-pattern means (1:8 and 2:8 in the right-hand table). Final is their combined evaluation mean, which characterizes the training schedule rather than a single deployable model, and SD is reported for Final.

Because deployment uses one of the constituent subnetworks, Table 3 reports them separately. On Walker2d at replay ratio 1, both subnetworks of the 3.5:8 schedule achieve higher mean returns than independently trained networks at the same deployment sparsity (3772.0 vs. 3440.5 for 3:8 and 3858.1 vs. 3645.3 for 4:8). Importantly, time multiplexing is not merely an averaging mechanism: the two discrete subnetworks are co-trained through shared parameters. Under 2.5:8, the 2:8 subnetwork (3075.2) exceeds fixed 2:8 training (2999.4), and the 3:8 subnetwork (3426.7) comes within 0.4% of fixed 3:8 training while the schedule reduces the nominal arithmetic cost of the sparse layers by 16.7% relative to fixed 3:8 training.The benefit is not uniform: under 1.5:8 training, both subnetworks underperform their fixed-sparsity counterparts. We hypothesize that the limited representational capacity of the 1:8 subnetwork contributes to this degradation. Additional results for HalfCheetah, Hopper and Ant are reported in Appendix F (Table 10).

5.1 TIME-MULTIPLEXED SPARSITY UNDER A HIGHER REPLAY RATIO

Motivated by the earlier gains from transient replay-ratio increases after mask updates, we evaluate 1.5:8 training with a constant replay ratio of 4 (Table 4). On Walker2d, the deployable 1:8 and 2:8 subnetworks reach 3322.2 and 3719.2, i.e., 76.2% and 85.3% of the replay-ratio-matched dense baseline (4358). Higher replay ratios are also associated with lower final actor and critic FAU in the

dense baselines across all evaluated environments (Appendix G, Table 11), complementing the earlier association between lower dense-critic FAU and smaller sparse-dense return gaps.

6 Acceleration with row-wise $N{:}M$ sparsity

We now turn to the speed part of Q2. The row-wise $1{:}8$ sparsity used in this paper reduces the number of multiply-accumulate (MAC) operations in the corresponding matrix multiplications by a factor of eight when operations involving pruned weights are skipped. However, this arithmetic reduction does not imply a proportional reduction in execution time. To assess practical acceleration without requiring dedicated sparse hardware, we implement CPU software-defined kernels for row-wise $N{:}M$ sparse matrix multiplications. The kernels exploit the regular block structure by accessing active weights through within-block indices rather than traversing all M positions. For the $1{:}M$ patterns evaluated here, each block's mask can be represented by a single index using $\lceil \log_2 M \rceil$ bits.

Layer-level measurements (Appendix D: Tables 7 and 8 and Figure 9) show forward-pass speedups of 2.17-3.95$\times$ for 1:8 and 3.77-8.07$\times$ for 1:16 sparsity, and full-layer speedups (forward pass and both gradients, excluding optimizer updates) of 1.14-4.12$\times$ and 1.45-5.29$\times$, respectively.

We integrate the kernels into dynamically sparse TD3 and compare against a reference that evaluates the masked weights $W \odot M$ with dense PyTorch linear layers backed by Intel MKL (Appendix D). The accelerated implementation also uses an approximate active-weight-only Adam that, after a short full-update phase following each mask refresh, updates only active weights. Both use one thread per process, and wall-clock time covers the entire environment-interaction loop (Appendix E). Across six environments and nine seeds per implementation, this yields speedups of 1.34-$1.83\times$, reducing training time by 25.2-45.5% (Table 9).

7 Conclusions

We investigated why high-ratio row-wise $N{:}M$ sparse training fails in deep RL and how it can be made both performant and faster. Under static 1:8 sparsity, randomly sampled topology has little predictive power for final return, whereas capacity matters. Widening can alleviate return deficits while keeping parameter counts close to or below those of the narrower dense baseline. Dense-critic FAU offers a descriptive diagnostic perspective of environment-dependent sensitivity to sparsity. Under dynamic sparsity, scale-tuned initialization mitigates actor saturation, transient replay-ratio increases improve mean returns in several environments, and wider 1:8 agents with a higher replay ratio achieve higher mean returns than same-width dense agents in most evaluated configurations. Time-multiplexed training co-trains individually deployable $N{:}M$ subnetworks, and our CPU implementation achieves 1.34-1.83$\times$ end-to-end speedups over an MKL-backed dense-kernel reference without dedicated sparse hardware. Overall, our results suggest that high-ratio $N{:}M$ sparsity in RL is a co-design problem rather than a choice of sparsity ratio. Sparse agents become both performant and fast when capacity, initialization, topology adaptation, replay and execution kernels are designed together. Overall, our results suggest that high-ratio $N{:}M$ sparsity in RL is a co-design problem rather than a choice of pruning ratio: sparse agents become both performant and fast when capacity, initialization, topology adaptation, replay and execution kernels are designed together.

Reproducibility Statement

Appendix A lists the network architecture, sparsity configuration, hyperparameters and evaluation protocol used in all experiments, and Appendices D and E describe the sparse kernels, hardware, software environment and timing protocol. Mask-update periods, replay-ratio schedules and numbers of seeds are given in the captions of the corresponding tables and figures. The source code, including the sparse CPU kernels and the training scripts, is provided in the supplementary material.

AI USE STATEMENT

In this work, we used generative AI tools (Claude and ChatGPT) to edit the paper for readability, terminology and translation, to provide feedback on the interpretation of results and the presentation of claims, to assist with implementing methods, and to check statistics derived from our experiment logs and reported results. We have not used generative AI tools for generating synthetic datasets, developing theoretical models or conceptual frameworks, proposing or refining hypotheses, designing experiments, cleaning and reformatting datasets, and formulating mathematical claims, mathematical proofs and qualitative data analysis are not applicable to this work. The authors reviewed all AI-assisted content, and take full responsibility for the paper.

REFERENCES

Zaheer Abbas, Rosie Zhao, Joseph Modayil, Adam White, and Marlos C. Machado. Loss of plasticity in continual deep reinforcement learning. In *Proceedings of the 2nd Conference on Lifelong Learning Agents*, volume 232 of *Proceedings of Machine Learning Research*, pp. 620–636. PMLR, 2023.

Samin Yeasar Arnob, Riyasat Ohib, Sergey Plis, Amy Zhang, Alessandro Sordoni, and Doina Precup. Efficient reinforcement learning by discovering neural pathways. *Advances in Neural Information Processing Systems*, 37:18660–18694, 2024.

Shibhansh Dohare, J Fernando Hernandez-Garcia, Qingfeng Lan, Parash Rahman, A Rupam Mahmood, and Richard S Sutton. Loss of plasticity in deep continual learning. *Nature*, 632(8026): 768–774, 2024.

Pierluca D'Oro, Max Schwarzer, Evgenii Nikishin, Pierre-Luc Bacon, Marc G Bellemare, and Aaron Courville. Sample-efficient reinforcement learning by breaking the replay ratio barrier. In *International Conference on Learning Representations*, 2023.

Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. In *International conference on machine learning*, pp. 2943–2952. PMLR, 2020.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2019.

Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1587–1596. PMLR, 2018.

Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Hojoon Lee, Dongyoon Hwang, Donghu Kim, Hyunseung Kim, Jun Jet Tai, Kaushik Subramanian, Peter R. Wurman, Jaegul Choo, Peter Stone, and Takuma Seno. SimBa: Simplicity bias for scaling up parameters in deep reinforcement learning. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=jXLiDKsuDo.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *International Conference on Learning Representations*, 2016.

Jiashun Liu, Johan Obando-Ceron, Aaron Courville, and Ling Pan. Neuroplastic expansion in deep reinforcement learning. In *International Conference on Learning Representations*, 2025.

Dor Livne and Kobi Cohen. Pops: Policy pruning and shrinking for deep reinforcement learning. *IEEE Journal of Selected Topics in Signal Processing*, 14(4):789–801, 2020.

Guozheng Ma, Lu Li, Sen Zhang, Zixuan Liu, Zhen Wang, Yixin Chen, Li Shen, Xueqian Wang, and Dacheng Tao. Revisiting plasticity in visual reinforcement learning: Data, modules and training stages. In *International Conference on Learning Representations*, 2024.

Guozheng Ma, Lu Li, Zilin Wang, Li Shen, Pierre-Luc Bacon, and Dacheng Tao. Network sparsity unlocks the scaling potential of deep reinforcement learning. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 42009–42029, 2025.

Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378*, 2021.

Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications*, 9(1):2383, 2018.

Siddharth Mysore, Bassel El Mabsout, Renato Mancuso, and Kate Saenko. Honey, i shrunk the actor: A case study on preserving performance with smaller actors in actor-critic RL. In *2021 IEEE Conference on Games (CoG)*, 2021. URL https://arxiv.org/abs/2102.11893.

Michal Nauman, Michal Bortkiewicz, Piotr Milos, Tomasz Trzcinski, Mateusz Ostaszewski, and Marek Cygan. Overestimation, overfitting, and plasticity in actor-critic: the bitter lesson of reinforcement learning. In *International Conference on Machine Learning*, 2024a.

Michal Nauman, Mateusz Ostaszewski, Krzysztof Jankowski, Piotr Miłoś, and Marek Cygan. Bigger, regularized, optimistic: Scaling for compute and sample efficient continuous control. In *Advances in Neural Information Processing Systems*, volume 37, 2024b. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/cd3b5d2ed967e906af24b33d6a356cac-Abstract-Conference.html.

Andrei A. Rusu, Sergio Gomez Colmenarejo, Caglar Gulcehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy distillation. In *International Conference on Learning Representations*, 2016.

Ghada Sokar, Elena Mocanu, Decebal Constantin Mocanu, Mykola Pechenizkiy, and Peter Stone. Dynamic sparse training for deep reinforcement learning. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence*, pp. 3437–3443, 2022. doi: 10.24963/ijcai.2022/477.

Ghada Sokar, Rishabh Agarwal, Pablo Samuel Castro, and Utku Evci. The dormant neuron phenomenon in deep reinforcement learning. In *International Conference on Machine Learning*, pp. 32145–32168. PMLR, 2023.

Yiqin Tan, Pihe Hu, Ling Pan, Jiatai Huang, and Longbo Huang. RLx2: Training a sparse deep reinforcement learning model from scratch. In *International Conference on Learning Representations*, 2023.

Marc Aurel Vischer, Robert Tjarko Lange, and Henning Sprekeler. On lottery tickets and minimal task representations in deep reinforcement learning. In *International Conference on Learning Representations*, 2022.

Isam Vrce, Andreas Kassler, and Gökçe Aydos. Rnm-td3: N:m semi-structured sparse reinforcement learning from scratch. In *Proceedings of the 18th International Conference on Agents and Artificial Intelligence - Volume 3: ICAART*, pp. 2105–2114. INSTICC, SciTePress, 2026. ISBN 978-989-758-796-2. doi: 10.5220/0014489400004052.

Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, 29, 2016.

Guowei Xu, Ruijie Zheng, Yongyuan Liang, Xiyao Wang, Zhecheng Yuan, Tianying Ji, Yu Luo, Xiaoyu Liu, Jiaxin Yuan, Pu Hua, Shuzhen Li, Yanjie Ze, Hal Daumé, III, Furong Huang, and Huazhe Xu. Drm: Mastering visual reinforcement learning through dormant ratio minimization. In *International Conference on Learning Representations*, 2024.

Li Yang, Jian Meng, Jae-sun Seo, and Deliang Fan. Get more at once: Alternating sparse training with gradient correction. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Haonan Yu, Sergey Edunov, Yuandong Tian, and Ari S. Morcos. Playing the lottery with rewards and multiple languages: Lottery tickets in RL and NLP. In *International Conference on Learning Representations*, 2020.

Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. Learning N: M fine-grained structured sparse neural networks from scratch. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=K9bw7vqp_s.

A Network Architecture and Sparsity Configuration

Unless otherwise stated, all experiments utilize the Twin Delayed Deep Deterministic Policy Gradient (TD3) (Fujimoto et al., 2018) algorithm. Both the actor and the critic networks are modeled as Multi-Layer Perceptrons (MLPs) comprising two hidden layers with 256 neurons each, utilizing ReLU activation functions. The actor's final output layer applies a `tanh` activation scaled to the environment's valid action bounds, while the critic follows the standard TD3 twin-Q design, processing the concatenated state-action vector through two independent Q-networks.

For sparse configurations, we apply an $N{:}M$ sparsity pattern row-wise to the linear weight matrices of the hidden layers. In our primary highly-sparse setups, the first two hidden layers strictly enforce a 1:8 sparsity pattern, while the final output heads of both the actor and the critic remain dense. For static sparsity baselines, compliant random masks are sampled at initialization and kept frozen throughout training. For dynamic $N{:}M$ sparsity runs, masks are periodically updated by selecting the largest-magnitude weights within each $N{:}M$ block.

A.1 Initialization Strategy and Tanh Saturation

The hidden layers of all networks are initialized using a He/Kaiming Uniform distribution, where the weights are sampled from the interval $\left[-\sqrt{6/\text{fan}_{\text{in}}}, \sqrt{6/\text{fan}_{\text{in}}}\right]$, with fan_{in} denoting the number of input units to the layer. To rigorously evaluate the impact of the initialization magnitude on training stability and mask evolution, we introduce a scaling factor. The base uniform bounds are multiplied by factors of 3, 2, 1, 0.5 and 0.33 across different experimental runs.

Following the initialization used for low-dimensional observations in DDPG (Lillicrap et al., 2016), we sample the weights and biases of the actor's final layer independently of the hidden-layer scaling from a uniform distribution $\mathcal{U}[-0.003, 0.003]$.

A.2 Initialization-scale sensitivity under static sparsity

Initialization scale affects static 1:8 performance, with environment-dependent preferences (Figure 6). The scale factor multiplies the standard He (Kaiming) initialization. Building on the input-layer scaling analysis of Vischer et al. (2022), we examine this factor under static row-wise 1:8 sparsity. Larger scales benefit Swimmer and Hopper, whereas Humanoid performs better without scaling up. We hypothesize that these differences partly reflect actor-output saturation. Larger initial weights can push the actor's `tanh` output into saturation, attenuating gradients and hindering learning. This may be particularly detrimental when many action dimensions must be coordinated, making smaller scales beneficial.

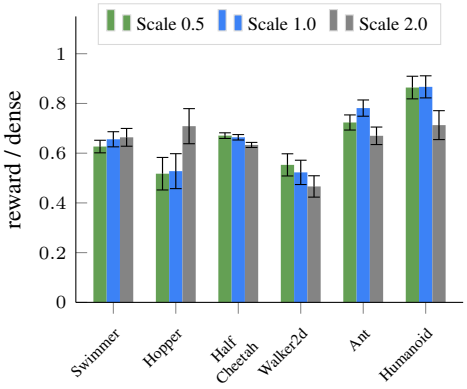

Figure 6: Normalized mean returns of static 1:8 sparse networks across different weight initialization scale factors (0.5, 1.0, and 2.0). All rewards are normalized against the dense baseline (1.0). Error bars show 95% confidence intervals.

A.3 Initialization-scale sensitivity under dynamic sparsity

Table 5 summarizes the results in Figure 3. For each environment and initialization scale, we average the dense-normalized mean returns over mask-update periods $K \in \{1000, 3000, 6000, 10000\}$. The overall mean assigns equal weight to all 24 environment-period combinations. Reduced multipliers perform well in aggregate, although the preferred scale factor remains environment-dependent.

A.4 Training Dynamics and Evaluation

Each agent is trained for a total of 10^6 environment interactions. To populate the replay buffer with diverse transitions, the first $25\,000$ steps are collected using a purely random policy. Following this

Table 5: Mean dense-normalized final-window return across the four mask-update periods at each initialization scale. The final row averages all 24 environment-period combinations equally.

Environment	$c = 3$	$c = 2$	$c = 1$	$c = 1/2$	$c = 1/3$
Ant	0.441	0.550	0.712	0.772	0.749
HalfCheetah	0.527	0.612	0.697	0.693	0.709
Hopper	0.092	0.361	0.723	0.533	0.616
Humanoid	0.015	0.029	0.397	0.799	0.839
Swimmer	0.319	0.341	0.490	0.700	0.696
Walker2d	0.135	0.280	0.515	0.544	0.499
Overall mean	0.255	0.362	0.589	0.674	0.685

warm-up phase, the actor interacts with the environment using Gaussian exploration noise $\sigma = 0.1$, which is clipped to the valid action range.

We utilize a replay-buffer minibatch size of 256 and optimize the networks using Adam with a learning rate of 3×10^{-4} for both the actor and the critic. The discount factor is set to $\gamma = 0.99$ and the target networks are updated via Polyak averaging with $\tau = 0.005$. We apply target-policy smoothing noise of $\sigma = 0.2$, clipped at $c = 0.5$ and delay the actor network updates, performing one actor update for every two critic updates.

Table 6: Summary of TD3 hyperparameters.

Hyperparameter	Value
Algorithm	TD3
Optimizer	Adam
Learning rate (Actor & Critic)	3×10^{-4}
Discount factor (γ)	0.99
Target update rate (τ)	0.005
Batch size	256
Architecture (Hidden layers)	[256, 256]
Activation function	ReLU
Actor output activation	Tanh
Total training steps	1 000 000
Random warm-up steps	25 000
Exploration noise	$\mathcal{N}(0, 0.1)$ (clipped to bounds)
Target policy smoothing noise	$\mathcal{N}(0, 0.2)$ (clipped to ± 0.5)
Policy update delay	2
Dynamic mask update period	3 000 steps
Actor output layer initialization	$\mathcal{U}[-0.003, 0.003]$

B Static Sparsity Sweep on Walker2d

We compare static 1:8, 2:8, 3:8, 4:8 and 6:8 sparsity with a dense baseline on Walker2d. Each configuration is evaluated over 100 independent seeds, except 2:8, which uses 150. Figure 7 shows the final-return distributions.

C Immediate Effect of a Mask Update

This appendix isolates the immediate effect of a mask update on the critics from the learning that follows it, providing direct evidence for the mask-update shock described in Section 4. We use the dynamic 1:8 runs at initialization scales 3 and $1/3$, with masks refreshed every 3,000 environment steps and 11 seeds per configuration. Learning starts after 25,000 random warm-up steps, so the first mask update after learning begins occurs at environment step 27,000.

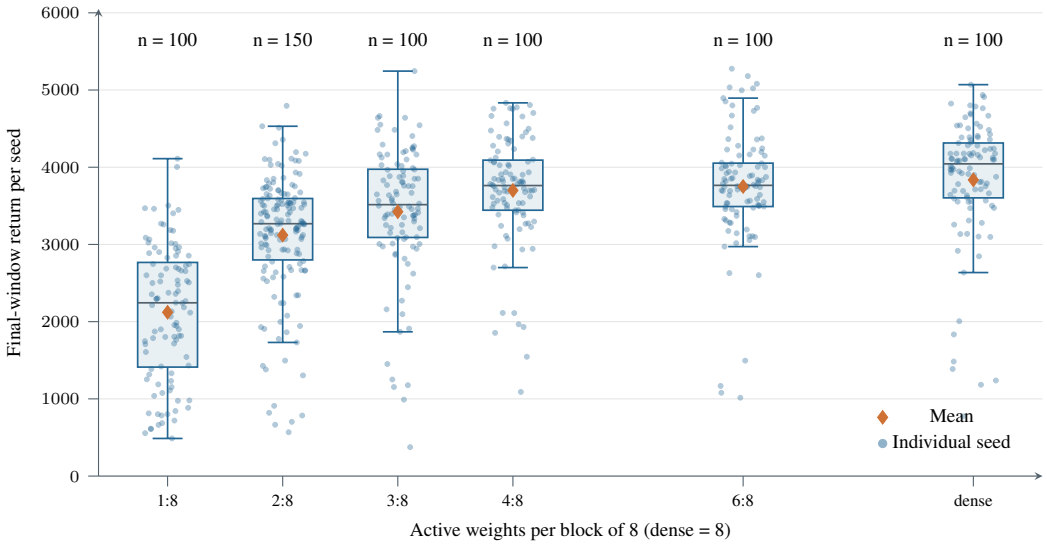

Figure 7: Final-return distributions for Walker2d under static $N{:}M$ sparsity and the dense baseline. Each point is one seed's mean return over the final 20 evaluations (steps 990,500–1,000,000). All seed observations are shown. orange diamonds mark means. The 2:8 configuration uses 150 seeds and each remaining configuration uses 100.

Measurement protocol. Immediately before the update, we sample one minibatch from the replay buffer and record two quantities: the mean absolute TD residual based on the minimum of the two online critics, $|\min\{Q_1(s,a), Q_2(s,a)\} - y|$, averaged over the minibatch, where y is the TD3 target computed with the target networks and target-policy smoothing; and the mean absolute disagreement between the twin critics, $|Q_1(s,\pi(s)) - Q_2(s,\pi(s))|$. We then refresh the masks of the online and target networks, recompute y and measure both quantities again on the same minibatch with the same target-smoothing noise. No optimizer step occurs between the two measurements, so their difference reflects the mask change alone. The disagreement is evaluated at the same states, but $\pi(s)$ may differ before and after the update because the actor's mask also changes.

Results. The update increases both quantities in all three environments at both initialization scales (Figure 8). At scale $1/3$, which mitigates actor saturation (Section 4.1), the mean absolute TD error rises from 1.41 to 5.19 on Humanoid, from 0.317 to 0.460 on Hopper and from 0.418 to 0.919 on Walker2d. The twin-critic disagreement rises from 0.311 to 4.61, from 0.0551 to 0.502 and from 0.104 to 1.32, respectively, a 9-15-fold increase. At scale 3, both quantities are already larger before the update, and after it they exceed their scale-$1/3$ counterparts by more than an order of magnitude. A smaller initialization scale thus reduces the shock but does not remove it; the transient replay-ratio increase in Section 4.1 targets this residual shock.

D Layer-level sparse kernel acceleration

We benchmark bias-free linear layers with square $H \times H$ weight matrices, using $H \in \{256, 512, 1024, 2048, 4096\}$ and batch sizes $B \in \{16, 32, 64, 256, 512, 1024\}$. The dense and sparse kernel paths operate on the same masked weight matrices: the dense path evaluates the multiplications using dense computation, whereas the sparse path skips pruned positions. Tables 7 and 8 report results for row-wise 1:8 and 1:16 sparsity, respectively.

Forward-pass timings measure sparse-weight-dense-input multiplication with pregathered sparse weights, including activation packing and output allocation. Full-layer timings cover one bias-free forward pass and both the input and weight gradients, including fresh weight gathering, layout copies and the full-shaped weight-gradient output. Optimizer updates, loss computation and mask construction are excluded.

All layer-level benchmarks use FP32 arithmetic on a dual-socket system with AMD EPYC 75F3 32-core processors, with the process pinned to logical CPU 0. The software environment comprises

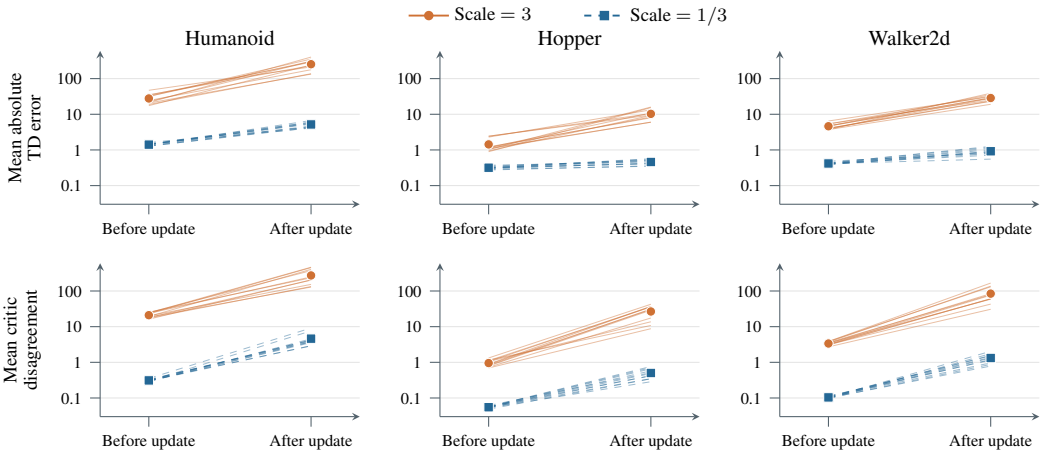

Figure 8: Critic diagnostics immediately before and after the first mask update after learning begins (environment step 27,000) under dynamic 1:8. The two measurements in each pair use the same minibatch and target-smoothing noise, with no optimizer step in between. All panels share the same logarithmic y-axis, so equal vertical changes correspond to equal multiplicative changes.

Python 3.13.2 and PyTorch 2.12.1+cu130, with execution on the CPU. The configured BLAS backend is Intel oneAPI MKL 2024.2 (build `20240605`; `BLAS_INFO=mkl`). PyTorch intra-op and inter-op thread counts are both set to one, as are `OMP_NUM_THREADS`, `MKL_NUM_THREADS`, and `OPENBLAS_NUM_THREADS`. The sparse kernels are implemented as a C++ extension compiled with GCC 11.4 using `-O3 -mavx2 -mfma`. Measurements use seed 91 and comprise 31 repetitions of 20 calls after 20 warm-up calls; reported timings are medians. Speedup is the ratio of dense to sparse execution time for the corresponding timing scope. Figure 9 visualizes the speedups reported in Tables 7 and 8. For reference, the nominal single-core FP32 peak of the AMD EPYC 75F3 (Zen 3) at its specified 4.0 GHz maximum boost is 128 GFLOP/s The dense forward GEMM reaches 87-92% of this bound for the tested $B \geq 256$.

Table 7: Layer-level CPU timings and speedups for row-wise 1:8 sparsity. Forward timings use pregathered weights. Full-layer speedups include fresh weight gathering and both gradients, but exclude optimizer updates.

		Forward time (ms)		Speedup	
H	B	Dense	Sparse	Forward	Full
256	16	0.027	0.012	2.332×	1.144×
256	32	0.045	0.013	3.454×	1.615×
256	64	0.081	0.024	3.362×	1.718×
256	256	0.301	0.090	3.356×	2.025×
256	512	0.600	0.177	3.385×	2.069×
256	1024	1.195	0.353	3.384×	1.903×
512	16	0.106	0.043	2.500×	1.592×
512	32	0.173	0.046	3.802×	2.502×
512	64	0.319	0.090	3.540×	2.412×
512	256	1.170	0.355	3.294×	2.286×
512	512	2.314	0.707	3.271×	2.152×
512	1024	4.619	1.419	3.255×	2.060×
1024	16	0.425	0.176	2.420×	1.978×
1024	32	0.721	0.182	3.951×	3.298×
1024	64	1.278	0.364	3.512×	2.615×
1024	256	4.650	1.442	3.224×	2.257×
1024	512	9.096	2.870	3.169×	2.170×
1024	1024	18.207	5.783	3.148×	2.073×
2048	16	1.737	0.689	2.522×	1.883×
2048	32	2.859	0.837	3.416×	2.965×
2048	64	5.100	1.692	3.015×	2.327×
2048	256	18.465	6.533	2.826×	2.144×
2048	512	36.785	13.483	2.728×	2.081×
2048	1024	73.757	26.547	2.778×	2.088×
4096	16	9.040	2.970	3.044×	3.779×
4096	32	13.369	4.202	3.182×	4.120×
4096	64	22.221	8.349	2.661×	3.452×
4096	256	75.771	33.495	2.262×	2.456×
4096	512	147.492	67.282	2.192×	2.268×
4096	1024	294.867	135.723	2.173×	2.144×

Table 8: Layer-level CPU timings and speedups for row-wise 1:16 sparsity. Forward timings use pregathered weights. Full-layer speedups include fresh weight gathering and both gradients, but exclude optimizer updates.

		Forward time (ms)		Speedup	
H	B	Dense	Sparse	Forward	Full
256	16	0.027	0.007	3.765×	1.451×
256	32	0.044	0.008	5.662×	1.944×
256	64	0.081	0.014	5.979×	2.206×
256	256	0.300	0.049	6.161×	2.969×
256	512	0.596	0.096	6.201×	3.177×
256	1024	1.192	0.190	6.266×	2.932×
512	16	0.104	0.023	4.618×	2.578×
512	32	0.175	0.024	7.399×	3.499×
512	64	0.318	0.046	6.951×	3.920×
512	256	1.166	0.176	6.616×	3.928×
512	512	2.318	0.349	6.635×	3.889×
512	1024	4.619	0.704	6.563×	3.637×
1024	16	0.426	0.085	4.988×	3.409×
1024	32	0.728	0.090	8.072×	5.114×
1024	64	1.284	0.178	7.207×	4.124×
1024	256	4.609	0.703	6.560×	4.177×
1024	512	9.137	1.410	6.478×	4.009×
1024	1024	18.201	2.835	6.421×	3.801×
2048	16	1.718	0.337	5.103×	3.106×
2048	32	2.844	0.404	7.047×	4.312×
2048	64	5.087	0.868	5.859×	3.735×
2048	256	18.493	3.216	5.750×	3.620×
2048	512	36.798	6.497	5.664×	3.811×
2048	1024	73.496	12.807	5.739×	3.895×
4096	16	8.990	1.526	5.890×	4.922×
4096	32	13.412	2.011	6.671×	5.288×
4096	64	22.250	4.019	5.536×	4.737×
4096	256	75.807	15.806	4.796×	4.108×
4096	512	147.818	32.089	4.607×	4.021×
4096	1024	294.698	65.060	4.530×	3.856×

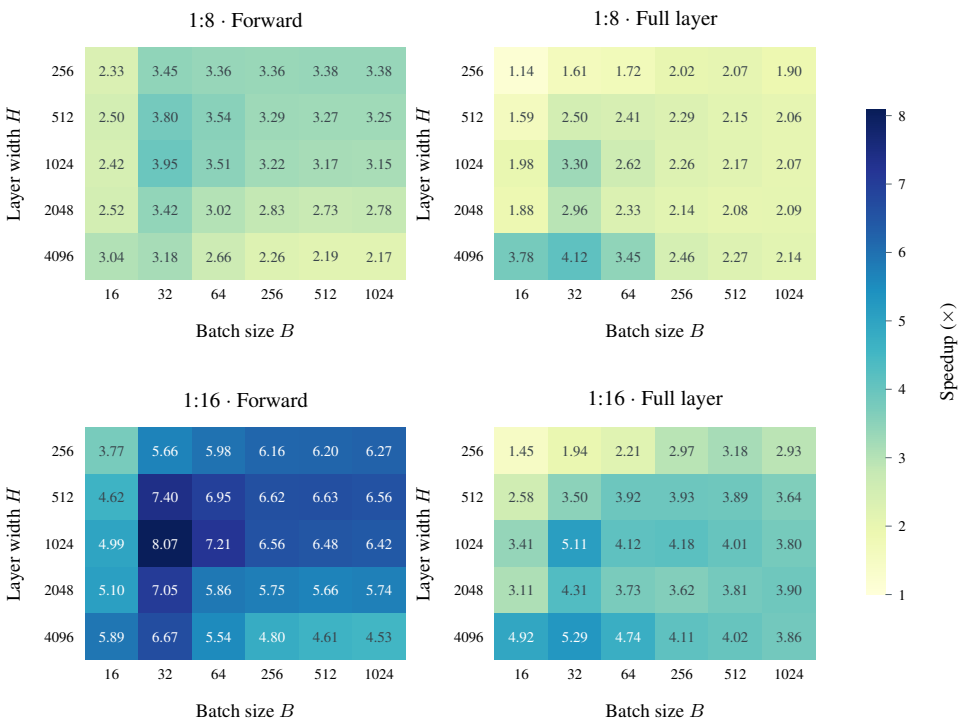

Figure 9: Layer-level CPU speedups for row-wise 1:8 and 1:16 sparsity across layer widths H and batch sizes B, using a shared color scale. Cells show the speedups reported in Tables 7 and 8, rounded to two decimals. Forward timings use pregathered weights and include activation packing and output allocation. Full-layer timings include a bias-free forward pass, both gradients, fresh weight gathering, layout copies, and a full-shaped weight-gradient output; optimizer updates, loss computation, and mask construction are excluded. Measurements use FP32 on one pinned CPU core and report medians over 31 repetitions of 20 calls after 20 warm-up calls.

E END-TO-END TRAINING ACCELERATION

We evaluate end-to-end acceleration on dynamically sparse $1{:}8$ TD3 agents with a replay buffer capacity of 200K transitions. The reference implementation evaluates the masked weights $W \odot M$ with Intel oneAPI MKL SGEMM and uses standard Adam. The accelerated implementation combines the sparse forward and backward kernels described in Appendix D with compiled mask construction and approximate active-only Adam: after 48 full optimizer updates at initialization and after each mask refresh, updates are restricted to active weights, while inactive weights and their moments are frozen. Both implementations use TD3 with dynamic row-wise $1{:}8$ sparsity, a hidden-layer width of 256, a minibatch size of 256 and one PyTorch CPU thread per process. Each run comprises one million environment steps and each configuration is evaluated over nine seeds. The actor and critic masks are refreshed synchronously every K environment steps. The base replay ratio is 1; following each mask refresh, it increases to 4 for the next $L = 0.1K$ environment steps before returning to 1. This schedule lies between the two transient schedules of Table 2 (RR = 4 for 5% and RR = 8 for 10% of the interval). Both implementations use the same mask-update and replay-ratio schedules, with environment-specific periods $K = 1000$ (Swimmer), 3000 (Humanoid), 6000 (Ant, Hopper and HalfCheetah) and 10000 (Walker2d). End-to-end wall-clock time is measured from immediately before the main environment-interaction loop until completion of its final iteration. It includes environment interaction, replay-buffer operations, forward and backward passes, optimizer and mask updates, periodic evaluation, console logging and best-evaluation checkpoint writes, and excludes initialization and the separate final evaluation. The nine seeds of each implementation ran concurrently as single-threaded processes, and the two implementations were launched in separate batches. Table 9 reports speedups of 1.34-$1.83\times$, corresponding to training-time reductions of 25.2-45.5%, together with the returns of both implementations over the same runs.

F ADDITIONAL TIME-MULTIPLEXED TRAINING RESULTS

Table 10 compares fixed and time-multiplexed sparsity configurations on HalfCheetah, Hopper and Ant at a replay ratio of 1 and a mask update period of 5,000 environment steps. Results report mean evaluation returns and standard deviations across 25 seeds. Across all three environments, $3.5{:}8$ achieves a higher mean return than the fixed $4{:}8$ configuration, whereas $1.5{:}8$ remains below the fixed $2{:}8$ configuration.

G REPLAY RATIO, DORMANT NEURONS AND SPARSE-NETWORK PERFORMANCE

Previous work links neuron inactivity to reduced network expressivity and loss of plasticity in deep reinforcement learning (Sokar et al., 2023; Abbas et al., 2023). Several methods address these challenges through different mechanisms: ReDo recycles dormant neurons (Sokar et al., 2023), Neuroplastic Expansion combines network growth with dormant-neuron pruning (Liu et al., 2025) and DrM uses dormant-ratio-guided perturbations and exploration-exploitation control (Xu et al., 2024). Higher replay ratios can exacerbate neuron dormancy (Sokar et al., 2023), while prior studies demonstrate that resets and regularization can help agents benefit from additional updates (D'Oro et al., 2023; Nauman et al., 2024a).

Table 11 reports the final actor and critic FAU of the dense baselines at replay ratios of 1, 2 and 4. Across all six environments, both actor and critic exhibit lower final FAU as the replay ratio increases. For example, increasing the replay ratio from 1 to 4 reduces critic FAU from 37.51% to 26.73% on Ant and from 43.80% to 38.44% on Walker2d.

Our static-sparsity results provide a complementary perspective: lower FAU in the dense critic generally coincides with sparse-network returns closer to the dense baseline (Figure 2). Dense-critic FAU may therefore help characterize differences in sensitivity to sparsity, but the present results do not establish a universal FAU threshold for return parity or a causal explanation for the gains observed under a higher replay ratio.

Table 9: End-to-end training of dynamic 1:8 TD3 (width 256, 200K buffer; K as in Table 2, $L = 0.1K$) with the MKL dense-kernel reference (standard Adam) and the accelerated implementation (sparse kernels, active-only Adam). Times and final returns are mean $\pm$ sample SD over nine seeds per implementation; each final return averages 2,000 evaluation episodes. Speedup is the ratio of mean times, with a nominal 95% percentile-bootstrap confidence interval (200,000 independent-arm resamples).

	Training time (h)				Return	
Environment	Reference	Accelerated	Speedup [95% CI]	Time red.	Reference	Accelerated
Ant	7.29 ± 0.12	4.55 ± 0.23	$1.60\times$ [1.56, 1.66]	37.6%	4301 ± 790	4755 ± 676
HalfCheetah	6.39 ± 0.11	3.99 ± 0.34	$1.60\times$ [1.52, 1.69]	37.5%	8749 ± 728	8304 ± 911
Hopper	6.15 ± 0.18	3.83 ± 0.32	$1.61\times$ [1.52, 1.69]	37.7%	2201 ± 1371	2683 ± 821
Humanoid	10.01 ± 0.11	5.46 ± 0.22	$1.83\times$ [1.79, 1.89]	45.5%	4565 ± 493	4513 ± 971
Swimmer	6.34 ± 0.11	4.00 ± 0.20	$1.59\times$ [1.54, 1.64]	37.0%	96.5 ± 39.4	105.3 ± 36.7
Walker2d	5.67 ± 0.14	4.24 ± 0.22	$1.34\times$ [1.30, 1.39]	25.2%	3641 ± 815	3341 ± 842

Table 10: Comparison of fixed and time-multiplexed $N{:}M$ training at replay ratio 1 and mask-update period 5,000. Results report mean return and SD across 25 seeds

Sparsity	HalfCheetah		Hopper		Ant	
	Mean	SD	Mean	SD	Mean	SD
4:8	9524.9	1208.5	2795.1	687.73	4528.1	913.37
3.5:8	9677.5	910.58	2893.6	493.21	4610.8	615.54
3:8	9149.4	1279.2	2850.1	747.78	4645.0	605.17
2.5:8	9282.9	760.08	2911.3	471.07	4299.7	771.92
2:8	8676.1	949.66	2500.0	865.52	4117.9	796.98
1.5:8	7466.8	818.51	2084.5	722.02	3060.6	609.70
1:8	7130.3	771.67	2031.3	1024.6	3335.7	802.25

Note: For time-multiplexed sparsity configurations, the final 2,000 evaluation episodes are proportionally divided between the two constituent discrete patterns.

Table 11: Final actor/critic FAU by environment for the dense baseline with varying replay ratios.

Environment	RR = 1		RR = 2		RR = 4	
	Actor	Critic	Actor	Critic	Actor	Critic
Ant	22.52%	37.51%	21.88%	32.82%	19.84%	26.73%
HalfCheetah	23.35%	45.65%	20.25%	44.56%	20.14%	41.96%
Hopper	22.41%	50.34%	18.36%	44.99%	17.41%	39.35%
Humanoid	23.35%	28.48%	21.92%	26.99%	20.01%	26.73%
Swimmer	27.05%	46.12%	23.97%	40.69%	15.26%	33.67%
Walker2d	22.25%	43.80%	22.04%	39.09%	20.05%	38.44%

H COMPARISON AGAINST SOTA UNSTRUCTURED SPARSITY METHODS

Table 12: Comparison of mean returns for 1:8 target sparsity (87.5%). The dense baseline is provided for reference along with static sparsity, DS-TD3, and RLx2. Realized sparsities for our methods are set identically to static sparsity.

Mean ± SD

Environment	Dense baseline	Ours 1:8	Static 1:8	DS-TD3	RLx2
Ant	4423 ± 792	3602 ± 706	3200 ± 686	3698 ± 846	3656 ± 838
HalfCheetah	10229 ± 933	8009 ± 570	6860 ± 511	8318 ± 828	8174 ± 556
Hopper	3059 ± 511	2631 ± 709	1583 ± 1021	3190 ± 195	2870 ± 654
Humanoid	4960 ± 143	4751 ± 879	4286 ± 1151	4972 ± 147	3109 ± 1289
Swimmer	84.37 ± 27.06	69.25 ± 28.77	52.878 ± 10.879	46.46 ± 2.00	78.80 ± 26.99
Walker2d	3836 ± 828	2428 ± 1099	2121 ± 873	3230 ± 1132	3122 ± 768

Realized sparsity (%)

Environment	Ours 1:8		Static 1:8		DS-TD3		RLx2
	Actor	Critic	Actor	Critic	Actor	Critic	Actor/Critic
Ant	85.60	87.26	85.60	87.26	85.34	87.19	87.5
HalfCheetah	85.62	87.18	85.62	87.18	85.23	87.05	87.5
Hopper	86.53	87.18	86.53	87.18	86.26	87.04	87.5
Humanoid	85.10	87.36	85.10	87.36	84.89	87.28	87.5
Swimmer	86.84	87.17	86.84	87.17	86.83	87.25	87.5
Walker2d	85.62	87.18	85.62	87.18	85.23	87.05	87.5

To evaluate the effectiveness of the proposed stabilization mechanisms, we compare our agents against the original static 1:8 sparsity architecture and several established baselines. Specifically, we benchmark against the following approaches:

- **Dense Baseline:** A standard TD3 agent utilizing dense neural networks, serving as the reference.
- **static 1:8 sparsity:** Statically determined N:M sparse mask at the beginning of the training.
- **DS-TD3** (Sokar et al., 2022): An unstructured dynamic sparsity method that utilizes random regrowth for network connections.
- **RLx2** (Tan et al., 2023): An unstructured dynamic sparsity method that prunes weights based on the smallest magnitude and regrows connections based on gradient magnitude.

The comparative results for a highly sparse regime (target sparsity of 1:8, or 87.5%) are summarized in Table 12. Incorporating our stabilization measures (scale-tuned initialization and transient replay-ratio increases) improves upon the static $N{:}M$ baseline in all six environments. Our approach comes within 4% of both unstructured dynamic sparsity methods on Ant and HalfCheetah and lies between them on Humanoid and Swimmer, but remains below both on Hopper and Walker2d. Unlike DS-TD3 and RLx2, however, its $N{:}M$ structure enables the wall-clock acceleration of Section 6, whereas unstructured sparsity is harder to translate into such gains (Mishra et al., 2021).