The State of Sparse Training in Deep Reinforcement Learning

Laura Graesser [*] [1] [2] **Utku Evci** [*] [2] **Erich Elsen** [3] **Pablo Samuel Castro** [2]

Abstract

The use of sparse neural networks has seen rapid growth in recent years, particularly in computer vision. Their appeal stems largely from the reduced number of parameters required to train and store, as well as in an increase in learning efficiency. Somewhat surprisingly, there have been very few efforts exploring their use in Deep Reinforcement Learning (DRL). In this work we perform a systematic investigation into applying a number of existing sparse training techniques on a variety of DRL agents and environments. Our results corroborate the findings from sparse training in the computer vision domain – *sparse networks perform better than dense networks for the same parameter count* – in the DRL domain. We provide detailed analyses on how the various components in DRL are affected by the use of sparse networks and conclude by suggesting promising avenues for improving the effectiveness of sparse training methods, as well as for advancing their use in DRL[1].

1. Introduction

Deep neural networks are typically organized as a stack of layers. Each layer consists of multiple neurons, where each neuron is connected to all neurons in the next layer; this is often referred to as a *dense* network. Alternatively, each neuron can be wired to a subset of the neurons in the next layer, resulting in a sparse, and smaller, network. Such sparse neural networks have been shown to match the performance of their dense counterparts while requiring only 10%-to-20% of the connections in most cases (Han et al., 2015; Gale et al., 2019; Blalock et al., 2020) providing significant memory, storage, and latency gains.

Deep networks have become a mainstay of scalable reinforcement learning (RL), key to recent successes such as playing – at superhuman levels – Atari games (Mnih et al., 2015), Go (Silver et al., 2016), Dota 2 (Berner et al., 2019) and as well as controlling complex dynamical systems such as stratospheric balloons (Bellemare et al., 2020) and plasma in real-time (Degrave et al., 2022). Despite their importance, most deep reinforcement learning (DRL) research focuses on improving the *algorithmic* aspect of DRL, and less on the *architecture* aspect. Sparse networks in particular have received very little attention, likely due to the belief that network over-parameterization helps with learning. However, recent work suggests that RL agents may suffer from *implicit under-parameterization* when training deep networks with gradient descent (Kumar et al., 2021), suggesting that the network's expressivity is in fact underused. In addition to this, Nikishin et al. (2022) suggests deep RL agents may have a tendency to overfit to early training data. Given this, one might expect there is substantial opportunity to compress RL agents. Further, sparse networks might benefit DRL by reducing the cost of training or aid running them in latency-constrained settings such as controlling plasma (Degrave et al., 2022).

One limitation of current research on training sparse neural networks is that it almost solely focuses on image classification benchmarks (Blalock et al., 2020; Hoefler et al., 2021) creating the risk of over-fitting to a specific domain. Do advances observed in computer vision (CV) transfer to DRL? A few recent works (Sokar et al., 2021; Arnob et al., 2021) attempt to address this by applying individual sparse training algorithms to DRL agents. However, it is still unknown if the key observation made in CV, that *sparse models perform better than dense ones for the same parameter count*, transfers to DRL.

In this work we focus on answering that question and systematically explore the effectiveness of different sparse learning algorithms in the online DRL setting. In order to achieve this, we benchmark four different sparse training algorithms using value-based (DQN (Mnih et al., 2015)) and actor-critic, (SAC (Haarnoja et al., 2018) and PPO (Schulman et al., 2017)) agents. Our results also include a broad analysis of various components that play a role in the train-

[*]Equal contribution [1]Robotics at Google [2]Google Research, Canada [3]Adept. Correspondence to: Laura Graesser <laura-graesser@google.com>, Utku Evci <evcu@google.com>, Pablo Samuel Castro <psc@google.com>.

Proceedings of the 39^{th} International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022. Copyright 2022 by the author(s).

[1] Code for reproducing our results can be found at github.com/google-research/rigl/tree/master/rigl/rl

ing of these sparse networks: sparsity distribution strategies, weight decay, layer initialization, signal-to-noise ratio for gradients, as well as batch size, topology update strategy and frequency. We summarize our main findings below:

- In almost all cases, sparse neural networks perform better than their dense counterparts for a given parameter count, demonstrating their potential for DRL.
- It is possible to train up to 80 - 90% sparse networks with minimal loss in performance compared to the standard dense networks.
- Pruning often obtains the best results, and dynamic sparse training improves over static sparse training significantly. However gradient based growth (Evci et al., 2020) seems to have a limited effect on performance. We argue this is due to low signal-to-noise ratio in gradients.
- The distribution of parameters among the actor and critic networks, as well as among different layers, impact training greatly. We observe that the best performance is obtained by allocating the majority of parameters to the critic network and using Erdos Renyi Kernel (ERK) sparsity distributions.
- We observe robust performance over various hyperparameter variations. Somewhat surprisingly, when adding noise to the observations, sparse methods achieve better robustness in most cases.

2. Background

2.1. Sparse training

Removing connections from neural networks was suggested at least as early as Mozer & Smolensky (1989), which coined the name "Skeleton" networks for what we today call sparse networks. Techniques for finding sparse neural networks can be grouped under two main categories.
(1) ◆ **Dense-to-sparse training** approaches (Han et al., 2016; Molchanov et al., 2017; Wortsman et al., 2019; Kusupati et al., 2020; Peste et al., 2021) start with a dense neural network and gradually reduce the network size by pruning its weights. This approach often achieves state-of-the-art performance amongst sparse networks, however it requires the same (or more) computation as training a large dense network. An alternative to pruning is **(2)** ▸◂● **sparse training** (Mocanu et al., 2018). This family of methods sparsifies the network at initialization and maintains this sparsity throughout training, thus reducing the training cost proportional to the sparsity of the network. However, training sparse neural networks from scratch is known to be difficult, leading to sub-optimal solutions (Frankle & Carbin, 2019; Liu et al., 2019; Evci et al., 2019). DRL training is notoriously resource hungry, hence we focus on the second family of methods (i.e. sparse training) in this work. There are various approaches to sparse training. One line of work (Lee et al., 2019; Wang et al., 2020; Tanaka et al., 2020), attempts to prune a dense network *immediately* on iteration 0. The resulting networks are used as an initialization for sparse training and kept fixed throughout. These techniques have been shown to have marginal gains over random pruning (Frankle et al., 2020), especially when used in modern training pipelines. Furthermore they may not generalize well in the RL setting as the non-stationarity of the data make it less clear that any decision made at iteration 0 will remain optimal throughout training.

Another line of work starts with randomly initialized sparse neural networks (both weights and masks) and focuses on improving sparse training by changing the sparse connectivity among neurons (Mocanu et al., 2018; Bellec et al., 2018) throughout the optimization. Known as Dynamic sparse training (DST), such approaches have been shown to match pruning results, making it possible to train sparse networks efficiently without sacrificing performance (Dettmers & Zettlemoyer, 2019; Evci et al., 2020).

In this work we benchmark one dense-to-sparse and three sparse training methods, which we briefly describe below:

◆ **Pruning (Zhu & Gupta, 2018):** uses a simple procedure to slowly make a dense network sparse over the course of one training run using weight magnitudes. We start pruning the network from 20% of the training steps and stop when we reach 80%, keeping the final sparse network fixed for the remaining of the training. This simple pruning algorithm is shown to exceed or match more complex pruning algorithms (Gale et al., 2019). Despite the fact it requires the same order of magnitude resources as training a dense network, we included this method since it serves as an upper bound on the sparse training performance.

▸ **Static:** prunes a given dense network randomly at initialization and the resulting sparse network is trained with a fixed structure. This is an important baseline to show the effectiveness of DST algorithms explained below.

◂ **Sparse Evolutionary Training (SET) (Mocanu et al., 2018):** Similar to *Static*, SET starts training with a random sparse network. During training, a portion of the connections are changed every N steps (the *update interval*) by replacing the lowest magnitude connections with new random ones. The fraction (*drop fraction*) of updated weights are decayed over the course of training to help the network converge to a minima. We use cosine decay as proposed by Dettmers & Zettlemoyer (2019).

● **Rigged Lottery (RigL) (Evci et al., 2020):** is the same as SET, except the new connections are activated using the gradient signal (highest magnitude) instead of at random.

This criteria has been shown to improve results significantly in image classification and with enough training iterations matches or exceed accuracies obtained by pruning.

2.2. Reinforcement learning

Reinforcement learning (RL) aims to design learning algorithms for solving sequential decision-making problems. Typically these are framed as an *agent* interacting with an *environment* at discrete time-steps by making action choices from a set of possible agent states; the environment in turn responds to the action selection by (possibly) changing the agent's state and/or providing a numerical *reward* (or cost); the agent's objective is to find a *policy* mapping states to actions so as to maximize (minimize) the sum of rewards (costs). This is formalized as a Markov decision process (Puterman, 1994) defined as a tuple $\langle \mathcal{X}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma\rangle$, where $\mathcal{X}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{P} : \mathcal{X} \times \mathcal{A} \rightarrow \Delta(\mathcal{X})$ defines the transition dynamics[2], $\mathcal{R} : \mathcal{X} \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward function, and $\gamma \in [0, 1)$ is a discount factor. A policy $\pi : \mathcal{X} \rightarrow \Delta(\mathcal{A})$ formalizes an agent's behaviour and induces a value function $V^\pi : \mathcal{X} \rightarrow \mathbb{R}$ defined via the well-known Bellman recurrence:

$$V^\pi(x) := \mathbb{E}_{a\sim\pi(x)}\left[\mathcal{R}(x,a) + \gamma\mathbb{E}_{x'\sim\mathcal{P}(x,a)}V^\pi(x')\right] \quad (1)$$

It is convenient to define state-action value functions $Q^\pi : \mathcal{X}\times\mathcal{A} \rightarrow \mathbb{R}$ as: $Q^\pi(x,a) := \mathcal{R}(x,a)+\gamma\mathbb{E}_{x'\sim\mathcal{P}(x,a)}V^\pi(x')$.

The goal of an RL agent is to find a policy $\pi^* := \max_\pi V^\pi$ (which is guaranteed to exist); for notational convenience we denote $V^* := V^{\pi^*}$ and $Q^* := Q^{\pi^*}$. In online RL the agent achieves this by iteratively improving an initial policy π_0: $\{\pi_0, \pi_1, \cdots, \pi_t, \cdots\}$ and using these intermediate policies to collect new experience from the environment in the form of *transitions* (x, a, r, x'), where $a \sim \pi_t(x)$, $r = \mathcal{R}(x, a)$, and $x' \sim \mathcal{P}(x, a)$. These transitions constitute the *dataset* the agent uses to improve its policies. In other words, the learning proocess is a type of *closed feedback loop*: an agent's policy directly affects the data gathered from the environment, which in turn directly affects how the agent updates its policy.

When $\mathcal{X}$ is very large, it is impractical to store V^π and Q^π in a table, so a function approximator $V_\theta \approx V^\pi$ (where θ are the approximator's parameters) is employed instead. This function approximator is usually one or more deep networks, and this type of RL is known as deep RL (DRL). DRL algorithms can be broadly categorized into two groups:

Value-based: The function Q^π is approximated by a deep network Q_θ. The policy is directly induced from the value estimate via $\pi_t(x) = \arg\max_{a\in\mathcal{A}} Q_{\theta_t}(x,a)$[3]. The parameters θ are trained using a temporal difference loss (based on Equation 1) from transitions sampled from $\mathcal{D}$:

$$\mathcal{L}(\theta) = \mathbb{E}_{(x,a,r,x')\sim\mathcal{D}}\left[Q_\theta(x,a) - (r + \gamma\max_{a'\in\mathcal{A}} Q_{\bar{\theta}}(x',a'))\right] \quad (2)$$

Here, $\bar{\theta}$ s a copy of θ that is infrequently synced with θ for more stable training (Mnih et al., 2015). These methods are typically employed for *discrete control* environments, where there is a finite (and relatively small) set of actions (e.g. Atari games (Bellemare et al., 2013)).

Policy-gradient: In contrast to value-based methods where the policy is implicitly improved by virtue of improving Q_θ, policy-gradient methods maintain and directly improve upon a policy π_ψ parameterized by ψ. These methods typically still make use of a value estimate Q_θ as part of their learning process, and are thus often referred to as actor-critic methods (where π_ψ is the actor and Q_θ the critic). Two potential advantages of these methods is that they can be more forgiving of errors in the Q_θ estimates, and they can handle continuous action spaces (for instance, by having $\pi_\psi(x)$ output mean and variance parameters from which actions may be sampled). These methods are typically employed for *continuous control* environments, where the action space is continuous (e.g. MuJoCo (Todorov et al., 2012)).

3. Experimental setup

DRL algorithms We investigate both value-based and policy-gradient methods. We chose DQN (Mnih et al., 2015) as the value-based algorithm, as it is the algorithm that first spurred the field of DRL, and has thus been extensively studied and extended. We chose two actor-critic algorithms for our investigations: an *on-policy* algorithm (PPO (Schulman et al., 2017)) and an *off-policy* one (SAC (Haarnoja et al., 2018)) ; both are generally considered to be state-of-the-art for many domains.

Environments For discrete-control we focus on three classic control environments (CartPole, Acrobot, and MountainCar) as well as 15 games from the ALE Atari suite (Bellemare et al., 2013) (see subsection A.4 for game selection details). For continuous-control we use five environments of varying difficulty from the MuJoCo suite (Todorov et al., 2012) (HalfCheetah, Hopper, Walker2d, Ant, and Humanoid). Rewards obtained by DRL algorithms have notoriously high variance (Agarwal et al., 2021). Therefore we repeat each experiment with at least 10 different seeds

[2] $\Delta(X)$ denotes the set of probability distributions over a finite set X.

[3] Although there are other mechanisms for defining a policy, such as using a softmax, and there are exploration strategies to consider, we present only the argmax setup for simplicity, as the other variants are mostly orthogonal to our analyses.

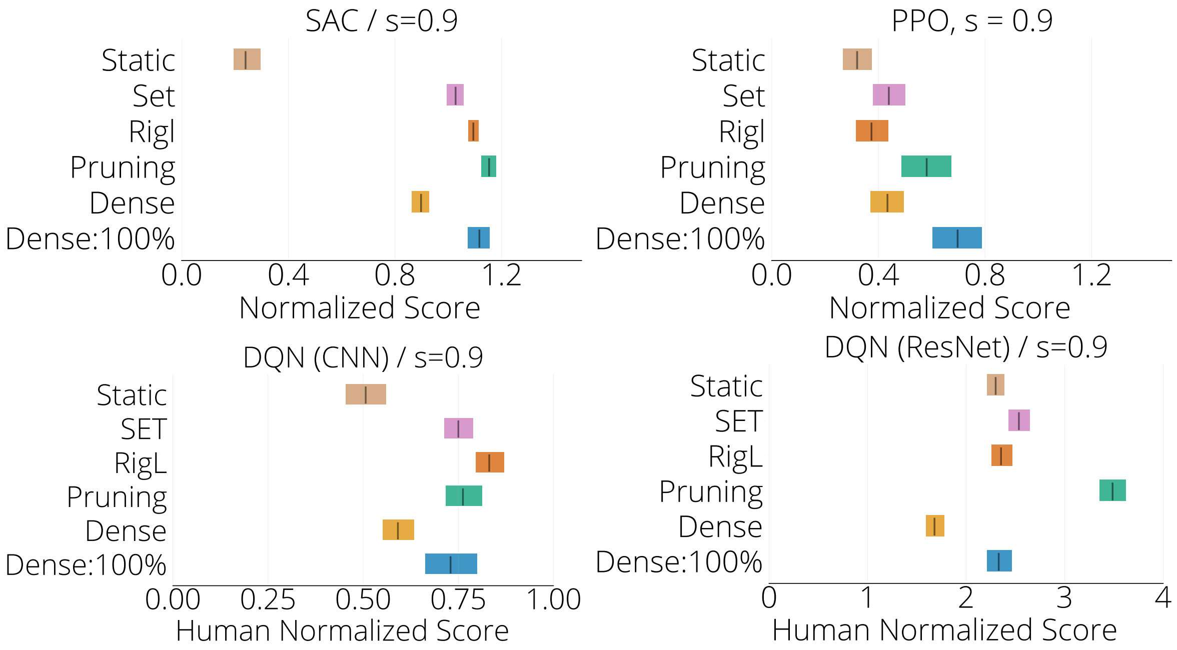

Figure 1. IQM plots for networks at 90% sparsity for various architecture and algorithm combinations. SAC and PPO are averaged over 5 MuJoCo environments, whereas DQN is averaged over 15 Atari environments. Results at different sparsities can be found at Appendix E. ”Dense: 100%” corresponds to the standard dense model. Atari scores were normalized using human performance per game. MuJoCo scores were normalized using the average returns obtained by the Dense: 100% SAC agent per game.

and report the average reward obtained over the last 10% of evaluations. We also provide 95% confidence intervals in all plots. See subsection A.1 for additional details.

Training For each sparse training algorithm considered (◆ Pruning , ▶ Static, ● RigL, and ◀ SET) we train policies ranging between 50% to 99% sparsity. To ensure a fair comparison between algorithms, we performed a hyper parameter sweep for each algorithm separately. The exception is DQN experiments on Atari for which it was too computationally expensive to do a full hyper-parameter sweep and we used values found in previous experiments instead. Sparse results in these environments may therefore be conservative compared to the well tuned dense baseline.

In addition to training the standard dense networks used in the literature, we also train smaller dense networks by scaling down layer widths to approximately match the parameter counts of the sparse networks, thereby providing a ”parameter-equivalent” dense baseline. We share details of the hyper parameter sweeps and hyper parameters used for each algorithm in subsection A.3.

Code Our code is built upon the TF-Agents (Guadarrama et al., 2018), Dopamine (Castro et al., 2018), and RigL (Evci et al., 2020) codebases. We use rliable (Agarwal et al., 2021) to calculate the interquartile mean (IQM) and plot the results. The IQM is calculated by discarding the bottom and top 25% of normalized scores aggregated from multiple runs and environments, then calculating the mean (reported with 95% confidence intervals) over the remaining 50% runs. Code for reproducing our results can be found at github.com/google-research/rigl/tree/master/rigl/rl

4. The State of Sparse Networks in Deep RL

We begin by presenting the outcome of our analyses in Figure 1 and Figure 2 for DQN (Atari), PPO and SAC (MuJoCo). Figure 1 presents the IQM at 90% sparsity, whilst in Figure 2 we evaluate final performance relative to the number of parameters. We share results for classic control, 2 additional MuJoCo and 12 additional Atari environments in Appendix B. Three main conclusions emerge; **(1)** In most cases performance obtained by sparse networks significantly exceeds that of their dense counterparts with a comparable number of parameters. Critically, in more difficult environments requiring larger networks (e.g. Humanoid, Atari), sparse networks can be obtained with efficient sparse training methods. **(2)** It is possible to train sparse networks with up to 80-90% fewer parameters and without loss in performance compared to the standard dense model. **(3)** Gradient based growing (i.e. RigL) seems to have limited impact on the performance of sparse networks. Next, we discuss each of these points in detail.

Sparse networks perform better. Inline with previous observations made in speech (Kalchbrenner et al., 2018), natu-

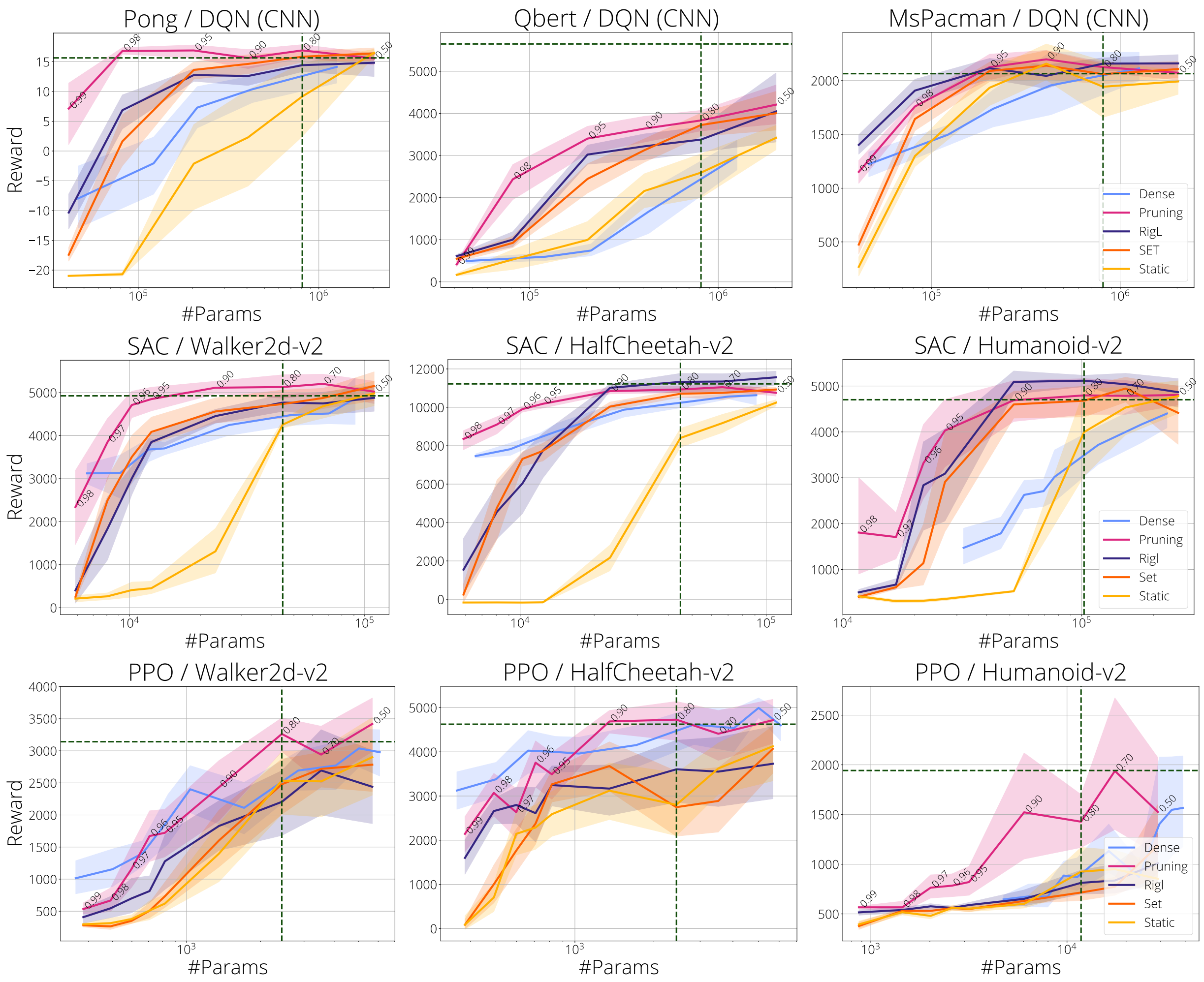

Figure 2. Comparison of the final reward relative to parameter count for the various methods considered: (row-1) DQN on Atari (CNN) (row-2) SAC on MuJoCo, (row-3) PPO on MuJoCo. We consider sparsities from 50% to 95% (annotated on the pruning curve) for sparse training methods and pruning. Parameter count for networks with 80% sparsity and the reward obtained by the dense baseline are highlighted with vertical and horizontal lines. Shaded areas represent 95% confidence intervals. See Appendix B for results on additional environments.

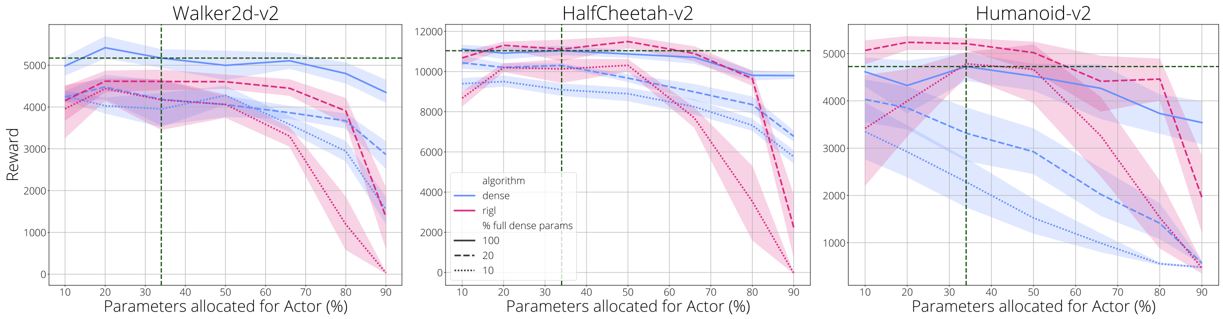

Figure 3. Evaluating how varying the actor-critic parameter ratio affects performance for a given parameter budget on different environment with policies trained using SAC. % of parameters allocated to the actor network is reported on the x axis. In SAC the parameter count of both critic networks is summed to give the overall critic parameter count. The vertical line corresponds the the standard parameter split and the horizontal line to the full dense training reward.

ral language modelling (Li et al., 2020) and computer vision (Evci et al., 2020), in almost all environments, sparse networks found by pruning achieve significantly higher rewards than the dense baseline. However training these sparse networks from scratch (*static*) performs poorly. DST algorithms (RigL and SET) improve over *static* significantly, however often fall short of matching the pruning performance.

Critically, we observe that for more difficult environment requiring larger networks such as Humanoid, MsPacman, Qbert and Pong, sparse networks found by efficient DST algorithms exceed the performance of the dense baseline.

How sparse? Next we asked how much sparsity is possible without loss in performance relative to that of the standard dense model (denoted by Dense:100% in Figure 1 and by the horizontal lines in Figure 2). We find that on average DST algorithms maintain performance up to 90% sparsity using SAC (Figure 1 top left) or DQN (Figure 1 bottom row), after which performance drops.

However performance is variable. For example, DST algorithms maintain performance especially well in MsPacman and Humanoid. Whereas in Qbert none of the methods are able to match the performance of the standard dense model at any of the examined levels of sparsity.

In the Atari environments, training a ResNet (He et al., 2015) following the architecture from Espeholt et al. (2018) instead of the standard CNN alone provided about 3x improvement in IQM scores. We were also surprised to see that pruning at 90% sparsity exceeds the performance of the standard ResNet model.

These observations indicate that while sparse training can bring very significant efficiency gains in some environments, it is not a guaranteed benefit. Unlike supervised learning, expected gains likely depend on both task and network, and merits further inquiry.

RigL and SET: For most sparsities (50% - 95%) we observe little difference between these two sparse training algorithms. At very high sparsities, RigL may outperform SET. The difference can be large (e.g. MsPacman), but is more often moderate (e.g. Pong) or negligible (e.g. Humanoid, Qbert) with overlapping confidence intervals. This suggests that the gradient signal used by RigL may be less informative in the DRL setting compared to image classification, where it obtains state-of-the-art performance and consistently outperforms SET. Understanding this phenomenon could be a promising direction for improving sparse training methods for DRL.

Perhaps unsurprisingly, the clarity of the differences between sparse and dense training is affected by the stability of the underlying RL algorithm. Our results using SAC, designed for stability, were the clearest, as were the DQN results. In contrast, our results using PPO which has much higher variance, were less stark. For this reason, we used SAC and DQN when studying the different aspects of sparse agents in the rest of this work.

5. Where should sparsity be distributed?

When searching for efficient network architectures for DRL it is natural to ask where sparsity is best allocated. To that end, we consider both how to distribute parameters between network types and as well as within them.

Actor or Critic? Although in value-based agents such as DQN there is a single network, in actor-critic methods such as PPO and SAC there are at least two: an actor and a critic network. It is believed that the underlying functions these networks approximate (e.g. a value function vs. a policy) may have significantly different levels of complexity, and this complexity likely varies across environments. Actor and critic typically have near-identical network architectures. However, for a given parameter budget it is not clear that this is the best strategy, as the complexity of the functions being approximated may vary significantly. We thus seek to understand how performance changes as the parameter ratio between the actor and critic is varied for a given parameter budget. In Figure 3 we assess three parameter budgets: 100%, 20% and 10% of the standard dense parameter count, and two training regimes, dense and sparse. Given the observed similarity in performance between RigL and SET in Figure 2, we selected one method, RigL, for this analysis.

We observe that assigning a low proportion of parameters to the critic (10 - 20%) incurs a high performance cost across all regimes. When parameters are more scarce, in 20% and 10% of standard dense settings, performance degradation is highest. This effect is not symmetric. Reducing the actor parameters to just 10% rarely affects performance compared to the default actor-critic split of 34:66 (vertical line).

Interestingly the default split appears well tuned, achieving the best performance in most settings. However in the more challenging Humanoid environment we see that for smaller dense networks, reducing the actor parameters to just 10% yields the best performance. Sparse networks follow a similar trend, but we notice that they appear to be more sensitive to the parameter ratio, especially at higher sparsities.

Overall this suggests that the value function is the more complex function to approximate in these settings, benefiting from the lion's share of parameters. It also suggests that tuning the parameter ratio may improve performance. Furthermore, FLOPs at evaluation time is determined only by the actor network. Since the actor appears to be easier to compress, this suggests large potential FLOPs savings for real-time usage of these agents. Finally, this approach

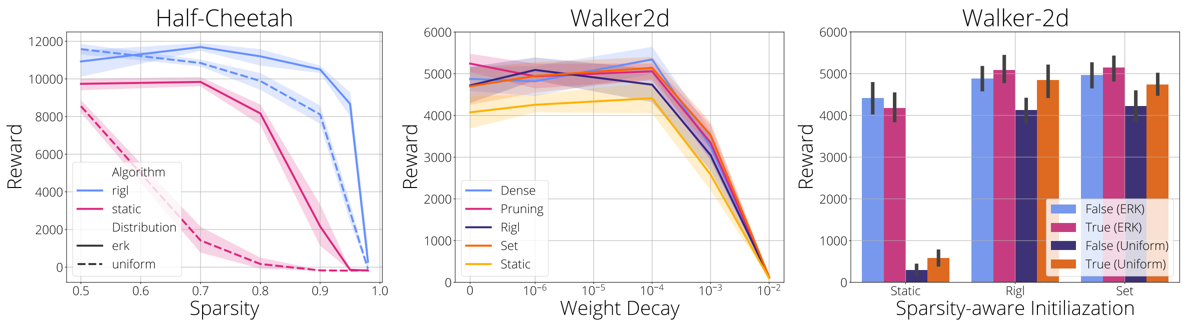

Figure 4. Sensitivity analysis on policies trained with SAC: (left) uniform vs ERK sparsity distributions, (center) weight decay, (right) sparsity-aware vs dense initialization. We use 80% (ERK) sparse networks in all plots unless noted otherwise. Plots for the remaining hyper-parameters are shared in Appendix D.

could be used to better understand the relative complexity of policies and values functions across different environments.

Within network sparsity In Figure 4 (left) we turn our attention to the question of distributing parameters within networks and compare two strategies; uniform and ERK (Evci et al., 2020). Given a target sparsity of, say 90%, uniform achieves this by making each layer 90% sparse; ERK distributes them proportional to the sum of its dimension, which has the effect of making large layers relatively more sparse than the smaller ones. Due to weight sharing in convolutional layers, ERK sparsity distribution doubles the FLOPs required at a given sparsity (Evci et al., 2020), which we also found to be the case with the convolutional networks used by DQN in the Atari environments (see subsection A.2 for further discussion). On the other hand, ERK has no effect on the FLOPs count of fully connected networks used in MuJoCo environments. Our results show that ERK significantly improves performance over uniform sparsity and thus we use ERK distribution in all of our experiments and share results with uniform distribution in Appendix C.

We hypothesize the advantage of ERK is because it leaves input and output layers relatively more dense, since they typically have few incoming our outgoing connections, and this enables the network to make better use of (a) the observation and (b) the learned representations at the highest layers in the network. It is interesting to observe that maintaining a dense output layer is one of the key design decisions made by Sokar et al. (2021) for their proposed algorithm.

6. Sensitivity analysis

In this section we assess the sensitivity of some key hyper-parameters for sparse training. Plots and commentary for the remaining hyper-parameters (drop fraction, topology update interval, and batch size) are shared in Appendix D.

Weight decay In Figure 4 (center) we evaluate the effect of weight decay and find that a small amount of weight decay is beneficial for pruning, RigL, and SET. This is to be expected since network topology choices are made based on weight magnitude, although we do note that the improvements are quite minor. Surprisingly weight decay seems to help dense even though it is not often used in DRL.

Findings: We recommend using small weight decay.

Sparsity-aware initialization In Figure 4 (right) we evaluate the effect of adjusting layer weight initialization based on a layer's sparsity on static, RigL and SET. A common approach to initialization is to scale a weight's initialization inversely by the square root of the number of incoming connections. Consequently, when we drop incoming connections, the initialization distribution should be scaled proportionately to the number of incoming connections (Evci et al., 2022). Figure 4 (right) shows that this sparsity-aware initialization consistently improves performance when using uniform distribution over layer sparsities. However the difference disappears when using ERK for RigL and SET and may even harm performance for static.

Findings: Performance is not sensitive to sparsity-aware initialization when using ERK and helps when using uniform layer sparsity. For RigL and SET we recommend always using sparsity-aware weight initialization (since it never appears to harm performance) but for static this may depend on layer sparsity.

7. Signal-to-noise ratio in DRL environments

Variance reduction is key to training deep models and often achieved through using momentum based optimizers (Schmidt et al., 2011; Kingma & Ba, 2015a). However when new connections are grown such averages are not available, therefore noise in the gradients can provide misleading signals. In Figure 5 we share the signal-to-noise ratio (SNR) for the Classic control and MuJoCo environments over the course of training. SNR is calculated as $\frac{|\mu|}{\sigma}$ where μ is the mean and σ is the standard deviation of gradients over a mini-batch. A low SNR means the signal

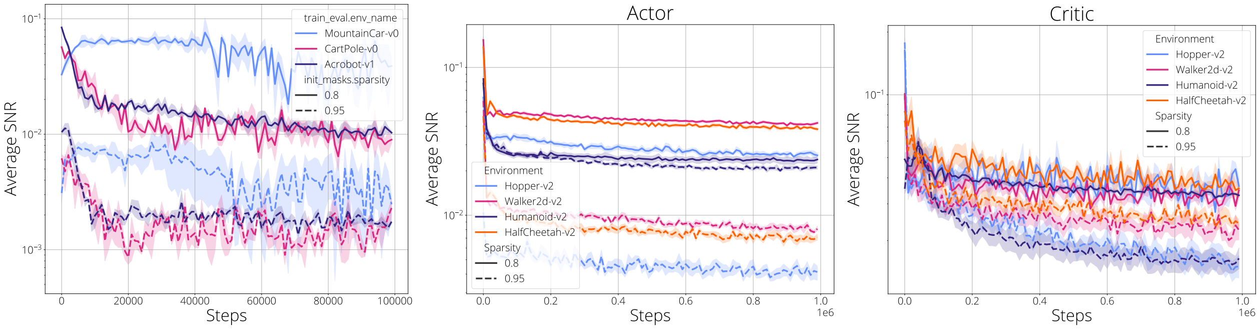

Figure 5. Signal-to-noise (SNR) ratio comparison of different gradient based DRL algorithms. We calculate SNR for every parameter in corresponding network (including the inactive/pruned weights) and report the mean SNR value. (left) DQN networks when training on classic control environments. SAC actor (center) and critic (right) networks during the training on MuJoCo environments.

is dominated by the variance and thus the mean (the signal) is uninformative. We calculate SNR for all parameters separately and report the mean. Mini-batch gradients can have average SNR values as low as 0.01 starting early in training. Higher sparsities seem to cause lower SNR values. Similarly, actor networks have lower SNR.

Findings: We find the average SNR for gradients to decrease with sparsity, potentially explaining the difficulty of using gradient based growing criteria in sparse training.

8. Are sparse networks robust to noise?

Sparse neural networks can improve results on primary metrics such accuracy and rewards, yet they might have some unexpected behaviours in other aspects (Hooker et al., 2020). In Figure 6 we assess the effect of adding increasing amounts of noise to the observations and measuring their effect on a trained policy. Noise was sampled $\sim \mathcal{N}(0, \sigma), \sigma \in [0, 1, ..., 30]$, quantized to an integer, and added to each observation's pixel values ($\in [0, 255]$) before normalization. Noise was sampled independently per pixel. We look at three data regimes; 100%, 50% and 10% of the standard dense model parameter count and compare dense and sparse training (RigL and SET). We made an effort to select policies with comparable performance for all the methods, chosen from the set of all policies trained during this work.

We observe that (1) smaller models are generally more robust to high noise than larger models, (2) sparse models are more robust to high noise than dense models on average, and (3) in most cases there are minimal differences when the noise is low.

We can see that in the very low data regime (10% full parameter count) policies trained using RigL are more robust to high noise compared with their dense counterparts, a fact observed across every environment. In the moderate data regime (50% full parameter count) the ordering is more mixed. In Qbert the dense model is most robust but the picture is reversed for Pong and McPacman. Finally, SET appears less robust to high noise than RigL, although we note this is not the case for Pong at 50% density.

Although a preliminary analysis, it does suggest that sparse training can produce networks that are more robust to observational noise, even when experienced post-training.

9. Related work

Sparse training Though research on pruning has a relatively long story by deep learning standards (Mozer & Smolensky, 1989; Sietsma & Dow, 1988; Han et al., 2015; Molchanov et al., 2017; Louizos et al., 2017), training sparse networks from scratch has only recently gained popularity. Goyal et al. (2017), Frankle & Carbin (2019), and Evci et al. (2019) showed that training sparse networks from a random initialization is difficult compared to dense neural networks. Despite this, various approaches have been recently proposed to improve sparse training, most notably lottery tickets (Frankle et al., 2019; Zhou et al., 2019), dynamic sparse training (Dettmers & Zettlemoyer, 2019; Mostafa & Wang, 2019; Mocanu et al., 2018; Bellec et al., 2018; Evci et al., 2020; Liu et al., 2021) and one-shot pruning (Lee et al., 2019; Wang et al., 2020; Tanaka et al., 2020; Liu & Zenke, 2020). Solutions that focus on initialization alone have been shown to be ineffective for contemporary models (Evci et al., 2022; Frankle et al., 2020), possibly due to the catapult mechanism observed early in training (Lewkowycz et al., 2020). For an in-depth survey on the topic, please see Hoefler et al. (2021).

Sparse networks in RL Livne & Cohen (2020) used pruning as an intermediary step to guide the width of dense neural networks for DQN and A2C agents. Most of the work investigating the use of sparse sparse training in RL are in the context of the lottery ticket hypothesis; Morcos et al. (2019) studied the existence of lucky sparse initializations using pruning and late-rewinding; Hasani et al. (2020) proposed an interesting approach by repurposing the sparse

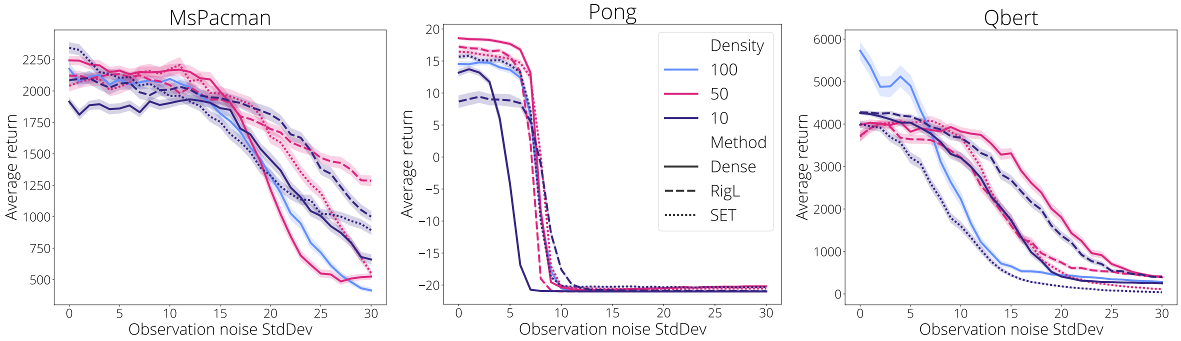

Figure 6. Robustness to observation noise. We test the robustness of networks trained using sparse and dense methods (denoted by line style) by adding Gaussian noise to the observations. We examine three parameter regimes, 100% (blue), 50% (pink) and 10% (purple) of the standard dense model parameter count. All policies were trained using DQN.

circuitry of the C. elegans soil-worm for RL tasks; Vischer et al. (2021) observed that the success of such initializations dependens heavily on selecting correct features for the input data, and not on any general qualities of the different initializations. Lee et al. (2021) proposed the use of block-circulant masks during early steps of training to improve the efficiency of pruning on TD3 agents, while Arnob et al. (2021) applied one-shot pruning algorithms in an offline-RL setting. Perhaps the work closest to ours is the algorithm proposed by Sokar et al. (2021), where authors applied the SET algorithm for end-to-end training of sparse networks in two actor-critic algorithms (TD3 and SAC). By a carefully chosen topology update schedule and dynamic architecture design, the proposed algorithm was able to match the dense network with a sparsity of around 50%.

Novel architectures in DRL A number of works have focused on evolving network architectures for RL policies. Nadizar et al. (2021) applied pruning together with evolution algorithms. Whiteson & Stone (2006) combined NEAT (Stanley & Miikkulainen, 2002) with Q-learning (Watkins, 1989) to evolve better learners, Gaier & Ha (2019) evolved strong architectural priors, resulting in networks that could solve tasks with a single randomly initialized shared weight, whilst Tang et al. (2020) evolve compact self-attention architectures as a form of indirect network encoding. Zambaldi et al. (2019) similarly explored self-attention enabling agents to perform relational reasoning and achieve state-of-the-art performance on the majority of StarCraft II mini-games. Another line of research seeks to improve the stability (Parisotto et al., 2020) and efficiency (Parisotto & Salakhutdinov, 2021) of transformers applied to DRL, whilst Shah & Kumar (2021) explore the utility of using features extracted from a pre-trained Resnet in the standard DRL pipeline. Consistent with our observations in this work, Ha & Schmidhuber (2018) showed it is possible to train very compact controllers (i.e. actors) albeit in a the context of model-based instead of the model-free RL setting considered here.

10. Discussion and Conclusion

In this work we sought to understand the state of sparse training for DRL by applying pruning, static, SET and RigL to DQN, PPO, and SAC agents trained on a variety of environments. We found sparse training methods to be a drop-in alternative for their dense counterparts providing better results for the same parameter count. From a practical standpoint we made recommendations regarding hyper-parameter settings and showed that non-uniform sparse initialization combined with tuning actor:critic parameter ratios improves performance.

We hope this work establishes a useful foundation for future research into sparse DRL algorithms and highlights a number of interesting research questions. In contrast to the computer vision domain, we observe that RigL fails to match pruning results. Low SNR in high sparsity regimes offers a clue but more work is needed to understand this phenomena. Our results in section 8 also suggest that sparse networks may aid in generalization and robustness to observational noise; this is an active area of interest and research in the DRL community, so a more thorough understanding could result in important algorithmic advances.

Acknowledgements

We thank Fabian Pedregosa, Rishabh Agarwal, and Adrien Ali Taïga for their helpful feedback on the manuscript, Oscar Ramirez for his help with on the TF-Agents codebase, and Trevor Gale and Sara Hooker for inspiring the title of this work. We also thank Brain Montreal RL team for their useful feedback on an early version of this work. Finally, we thank Bram Grooten for pointing out Degrave et al. (2022) and their contribution to the motivation for this work.

References

Agarwal, R., Schwarzer, M., Castro, P. S., Courville, A., and Bellemare, M. G. Deep reinforcement learning at the edge of the statistical precipice. *Advances in Neural Information Processing Systems*, 2021.

Arnob, S. Y., Ohib, R., Plis, S., and Precup, D. Single-shot pruning for offline reinforcement learning. *ArXiv*, abs/2112.15579, 2021.

Bellec, G., Kappel, D., Maass, W., and Legenstein, R. A. Deep rewiring: Training very sparse deep networks. In *International Conference on Learning Representations*, 2018.

Bellemare, M., Candido, S., Castro, P., Gong, J., Machado, M., Moitra, S., Ponda, S., and Wang, Z. Autonomous navigation of stratospheric balloons using reinforcement learning. *Nature*, 588:77–82, 12 2020. doi: 10.1038/s41586-020-2939-8.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The Arcade Learning Environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, June 2013.

Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., de Oliveira Pinto, H. P., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F., and Zhang, S. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019. URL http://arxiv.org/abs/1912.06680.

Blalock, D., Ortiz, J. J. G., Frankle, J., and Guttag, J. What is the state of neural network pruning? *ArXiv*, 2020. URL https://arxiv.org/abs/2003.03033.

Castro, P. S., Moitra, S., Gelada, C., Kumar, S., and Bellemare, M. G. Dopamine: A research framework for deep reinforcement learning. *CoRR*, abs/1812.06110, 2018. URL http://arxiv.org/abs/1812.06110.

Degrave, J., Felici, F., Buchli, J., Neunert, M., Tracey, B., Carpanese, F., Ewalds, T., Hafner, R., Abdolmaleki, A., de las Casas, D., Donner, C., Fritz, L., Galperti, C., Huber, A., Keeling, J., Tsimpoukelli, M., Kay, J., Merle, A., Moret, J.-M., Noury, S., Pesamosca, F., Pfau, D., Sauter, O., Sommariva, C., Coda, S., Duval, B., Fasoli, A., Kohli, P., Kavukcuoglu, K., Hassabis, D., and Riedmiller, M. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 2022.

Dettmers, T. and Zettlemoyer, L. Sparse networks from scratch: Faster training without losing performance. *ArXiv*, 2019. URL http://arxiv.org/abs/1907.04840.

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S., and Kavukcuoglu, K. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. *CoRR*, 2018.

Evci, U., Pedregosa, F., Gomez, A. N., and Elsen, E. The difficulty of training sparse neural networks. *ArXiv*, 2019. URL http://arxiv.org/abs/1906.10732.

Evci, U., Gale, T., Menick, J., Castro, P. S., and Elsen, E. Rigging the lottery: Making all tickets winners. In *Proceedings of Machine Learning and Systems 2020*, 2020.

Evci, U., Ioannou, Y. A., Keskin, C., and Dauphin, Y. Gradient flow in sparse neural networks and how lottery tickets win. In *AAAI Conference on Artificial Intelligence*, 2022.

Frankle, J. and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *7th International Conference on Learning Representations (ICLR)*, 2019.

Frankle, J., Dziugaite, G. K., Roy, D. M., and Carbin, M. Stabilizing the lottery ticket hypothesis. *ArXiv*, 2019. URL https://arxiv.org/abs/1903.01611.

Frankle, J., Dziugaite, G. K., Roy, D. M., and Carbin, M. Pruning neural networks at initialization: Why are we missing the mark? *ArXiv*, 2020. URL https://arxiv.org/abs/2009.08576.

Gaier, A. and Ha, D. Weight agnostic neural networks. 2019. URL https://weightagnostic.github.io. https://weightagnostic.github.io.

Gale, T., Elsen, E., and Hooker, S. The state of sparsity in deep neural networks. *ArXiv*, 2019. URL http://arxiv.org/abs/1902.09574.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Guadarrama, S., Korattikara, A., Ramirez, O., Castro, P., Holly, E., Fishman, S., Wang, K., Gonina, E., Wu, N., Kokiopoulou, E., Sbaiz, L., Smith, J., Bartók, G., Berent, J., Harris, C., Vanhoucke, V., and Brevdo, E. TF-Agents: A library for reinforcement learning in tensorflow. https://github.com/tensorflow/agents, 2018. URL https://github.com/tensorflow/agents. [Online; accessed 25-June-2019].

Ha, D. and Schmidhuber, J. Recurrent world models facilitate policy evolution. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/file/2de5d16682c3c35007e4e92982f1a2ba-Paper.pdf.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning (ICML)*, 2018.

Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, 2015.

Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., and Dally, W. J. EIE: Efficient Inference Engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

Hasani, R., Lechner, M., Amini, A., Rus, D., and Grosu, R. A natural lottery ticket winner: Reinforcement learning with ordinary neural circuits. In *Proceedings of the 37th International Conference on Machine Learning*. PMLR, 2020. URL https://proceedings.mlr.press/v119/hasani20a.html.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *CoRR*, 2015.

Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N., and Peste, A. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *ArXiv*, abs/2102.00554, 2021.

Hooker, S., Courville, A. C., Clark, G., Dauphin, Y., and Frome, A. What do compressed deep neural networks forget. *arXiv: Learning*, 2020.

Kalchbrenner, N., Elsen, E., Simonyan, K., Noury, S., Casagrande, N., Lockhart, E., Stimberg, F., Oord, A., Dieleman, S., and Kavukcuoglu, K. Efficient neural audio synthesis. In *International Conference on Machine Learning (ICML)*, 2018.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015a.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In Bengio, Y. and LeCun, Y. (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015b. URL http://arxiv.org/abs/1412.6980.

Kumar, A., Agarwal, R., Ghosh, D., and Levine, S. Implicit under-parameterization inhibits data-efficient deep reinforcement learning. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=O9bnihsFfXU.

Kusupati, A., Ramanujan, V., Somani, R., Wortsman, M., Jain, P., Kakade, S., and Farhadi, A. Soft threshold weight reparameterization for learnable sparsity. In *Proceedings of the International Conference on Machine Learning*, 2020.

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Lee, J., Kim, S., Kim, S., Jo, W., and Yoo, H.-J. Gst: Group-sparse training for accelerating deep reinforcement learning. *ArXiv*, abs/2101.09650, 2021.

Lee, N., Ajanthan, T., and Torr, P. H. S. SNIP: Single-shot Network Pruning based on Connection Sensitivity. In *International Conference on Learning Representations (ICLR), 2019*, 2019.

Lewkowycz, A., Bahri, Y., Dyer, E., Sohl-Dickstein, J., and Gur-Ari, G. The large learning rate phase of deep learning: the catapult mechanism. *Arxiv*, 2020. URL https://arxiv.org/pdf/2003.02218.

Li, Z., Wallace, E., Shen, S., Lin, K., Keutzer, K., Klein, D., and Gonzalez, J. Train large, then compress: Rethinking model size for efficient training and inference of transformers. *ArXiv*, abs/2002.11794, 2020.

Liu, S., Yin, L., Mocanu, D. C., and Pechenizkiy, M. Do we actually need dense over-parameterization? in-time over-parameterization in sparse training. In *ICML*, 2021.

Liu, T. and Zenke, F. Finding trainable sparse networks through neural tangent transfer. In *ICML*, 2020.

Liu, Z., Sun, M., Zhou, T., Huang, G., and Darrell, T. Rethinking the value of network pruning. In *International Conference on Learning Representations*, 2019.

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

Louizos, C., Ullrich, K., and Welling, M. Bayesian compression for deep learning. In *Advances in Neural Information Processing Systems*, 2017.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533, 2015.

Mocanu, D. C., Mocanu, E., Stone, P., Nguyen, P. H., Gibescu, M., and Liotta, A. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature Communications*, 2018.

Molchanov, D., Ashukha, A., and Vetrov, D. P. Variational Dropout Sparsifies Deep Neural Networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, 2017.

Morcos, A., Yu, H., Paganini, M., and Tian, Y. One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers. In *Advances in Neural Information Processing Systems*, 2019.

Mostafa, H. and Wang, X. Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, 2019. URL http://proceedings.mlr.press/v97/mostafa19a.html.

Mozer, M. C. and Smolensky, P. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In *Advances in Neural Information Processing Systems 1*, 1989.

Nadizar, G., Medvet, E., Pellegrino, F. A., Zullich, M., and Nichele, S. On the effects of pruning on evolved neural controllers for soft robots. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2021.

Nikishin, E., Schwarzer, M., D'Oro, P., Bacon, P.-L., and Courville, A. The primacy bias in deep reinforcement learning. In *Proceedings of the Thirty-ninth International Conference on Machine Learning (ICML'22)*, 2022.

Parisotto, E. and Salakhutdinov, R. Efficient transformers in reinforcement learning using actor-learner distillation. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=uR9LaO_QxF.

Parisotto, E., Song, F., Rae, J., Pascanu, R., Gulcehre, C., Jayakumar, S., Jaderberg, M., Kaufman, R. L., Clark, A., Noury, S., Botvinick, M., Heess, N., and Hadsell, R. Stabilizing transformers for reinforcement learning. In III, H. D. and Singh, A. (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 7487–7498. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/parisotto20a.html.

Peste, A., Iofinova, E., Vladu, A., and Alistarh, D. Ac/dc: Alternating compressed/decompressed training of deep neural networks. *ArXiv*, abs/2106.12379, 2021.

Puterman, M. L. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994. ISBN 0471619779.

Schmidt, M., Roux, N. L., and Bach, F. Convergence rates of inexact proximal-gradient methods for convex optimization. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, 2011.

Schulman, J., Moritz, P., Levine, S., Jordan, M. I., and Abbeel, P. High-dimensional continuous control using generalized advantage estimation. In Bengio, Y. and LeCun, Y. (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1506.02438.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

Shah, R. M. and Kumar, V. Rrl: Resnet as representation for reinforcement learning. In Meila, M. and Zhang, T. (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 9465–9476. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/shah21a.html.

Sietsma, J. and Dow, R. J. Neural net pruning-why and how. In *IEEE International Conference on Neural Networks*, 1988. doi: 10.1109/ICNN.1988.23864.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

Sokar, G., Mocanu, E., Mocanu, D. C., Pechenizkiy, M., and Stone, P. Dynamic sparse training for deep reinforcement learning. *ArXiv*, abs/2106.04217, 2021.

Stanley, K. O. and Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evol. Comput.*, 10(2):99–127, jun 2002. ISSN 1063-6560. doi: 10.1162/106365602320169811. URL https://doi.org/10.1162/106365602320169811.

Tanaka, H., Kunin, D., Yamins, D. L. K., and Ganguli, S. Pruning neural networks without any data by iteratively conserving synaptic flow. *ArXiv*, 2020. URL https://arxiv.org/abs/2006.05467.

Tang, Y., Nguyen, D., and Ha, D. Neuroevolution of self-interpretable agents. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, pp. 414–424, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371285. doi: 10.1145/3377930.3389847. URL https://doi.org/10.1145/3377930.3389847.

Todorov, E., Erez, T., and Tassa, Y. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, 2012. doi: 10.1109/IROS.2012.6386109.

Vischer, M. A., Lange, R., and Sprekeler, H. On lottery tickets and minimal task representations in deep reinforcement learning. *ArXiv*, abs/2105.01648, 2021.

Wang, C., Zhang, G., and Grosse, R. Picking winning tickets before training by preserving gradient flow. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SkgsACVKPH.

Watkins, C. J. C. H. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989. URL http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

Whiteson, S. and Stone, P. Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research*, 7(31):877–917, 2006. URL http://jmlr.org/papers/v7/whiteson06a.html.

Wortsman, M., Farhadi, A., and Rastegari, M. Discovering neural wirings. In *Advances in Neural Information Processing Systems*, 2019.

Zambaldi, V., Raposo, D., Santoro, A., Bapst, V., Li, Y., Babuschkin, I., Tuyls, K., Reichert, D., Lillicrap, T., Lockhart, E., Shanahan, M., Langston, V., Pascanu, R., Botvinick, M., Vinyals, O., and Battaglia, P. Deep reinforcement learning with relational inductive biases. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=HkxaFoC9KQ.

Zhou, H., Lan, J., Liu, R., and Yosinski, J. Deconstructing lottery tickets: Zeros, signs, and the supermask. In *Advances in Neural Information Processing Systems*, 2019.

Zhu, M. and Gupta, S. To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression. In *International Conference on Learning Representations Workshop*, 2018.

A. Experimental Details

A.1. Training schedule

During training all agents are allowed M environment transitions, with policies being evaluated for K episodes / steps every N environment frames, where the values vary per suite and shared below. Atari experiments use a frame skip of 4, following (Mnih et al., 2015) thus 1 environment step = 4 environment frames.

Environment	M	N	K
K = episodes			
Classic control	$100,000$	$2,000$	20
MujoCo (SAC)	$1,000,000$	$10,000$	30
MujoCo (PPO)	$1,000,000$	$2,000$	20
K = environment steps			
Atari Suite	$40,000,000$	$1,000,000$	$125,000$

A.2. FLOPs behaviour of Sparse ERK networks in DRL

As reported in Evci et al. (2020), using ERK sparsity distribution often doubles the FLOPs needed for sparse models compared using the uniform sparsity distribution. This is due to the parameter sharing in convolutional layers. The spatial dimensions, kernel size and the stride of a convolutional layer affects how many times each weight is used during the convolution which in turn determines the contribution of each weight towards the total FLOPs count. In modern CNNs, the spatial dimensions of the feature maps often decreases monotonically towards the output of the network, making the contribution of the connections in later layers to the total FLOPs count smaller. Furthermore, the size of the layers typically increases towards the output and thus ERK removes a larger proportion of the connections from these later layers compared to uniform. Consequently a network sparsified using the ERK distribution will have a larger FLOPs count compared to one sparsified using a uniform distribution.

Due to the lack of parameter sharing fully connected layers used in MuJoCo and classic control experiments, sparse networks with ERK have same amount of FLOPs as the uniform. Networks used in Atari experiments, however, uses convolutional networks and thus ERK doubles the FLOPs required compared to uniform. FLOPs scaling of sparse networks with ERK distributions used for Pong game can be found in Figure 7. Atari games have differently sized action spaces (Pong has 6 actions for example), which affects the number of neurons in the last layer. However since the last layer is very small and fully connected, it should have a very little effect on the results provided here.

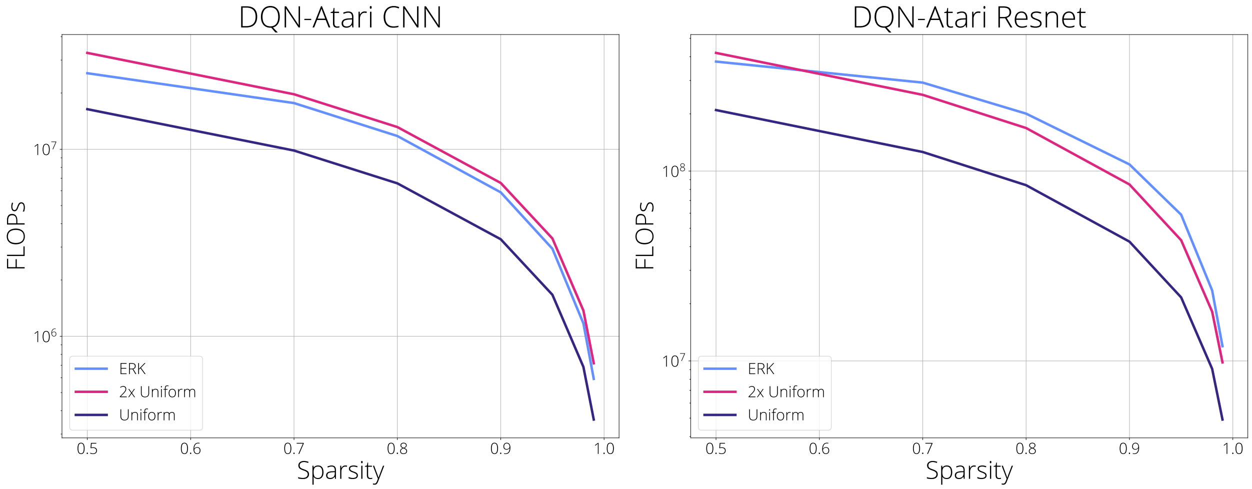

Figure 7. FLOPs scaling of different sparsity distributions on (left) Nature CNN and (right) Impala ResNet architectures used in MsPacman environment.

A.3. Hyper-parameter sweep

We perform a grid search over different hyper parameters used in ■ *Dense*, ◆ *Prune*, ▶ *Static*, ◀ *SET* and ● *RigL* algorithms. Unless otherwise noted, we use hyper-parameters used in regular dense training. When pruning, we start pruning around 20% of training steps and stop when 80% of training is completed following the findings of Gale et al. (2019). We use same default hyper-parameters for SET and RigL. Fort both algorithms we start updating the mask at initialization and decay the drop fraction over the course of the training using a cosine schedule, similar to pruning stopping the updates when 80% of training is completed.

We search over the following parameters:

1. **Weight decay (■ ◆ ▶ ◀ ●):** Searched over the grid [0, 1e-6, 1e-4, 1e-3].
2. **Update Interval (◆ ◀ ●):** refers to how often models are pruned or sparse topology is updated. Searched over the grid [100, 250, 500, 1000, 5000].
3. **Drop Fraction (◆ ◀ ●):** refers to the maximum percentage of parameters that are dropped and added when network topology is updated. This maximum value is decayed during training according to a cosine decay schedule. Searched over the grid [0.0,0.1,0.2,0.3,0.5].
4. **Sparsity-aware initialization (▶ ◀ ●):** refers to whether sparse models are initialized with scaled initialization or not.

We repeat the hyper-parameter search for each DRL algorithm using the Acrobot (for DQN) and Walker2D (for PPO and SAC) environments. Best hyper-parameters found in these environments are then used when training in other similar environments (i.e. classic control for DQN and MuJoCO for PPO and SAC). See Table 1, Table 2, and Table 3 for the best hyper parameters found in each setting.

Table 1. DQN best hyper-parameters for Classic Control from sweep

Algorithm	Weight Decay	Update Interval	Drop Fraction	Sparsity-aware init
Dense	$1 \cdot 10^{-6}$	-	-	-
Pruning	$1 \cdot 10^{-6}$	1,000	-	True
Static	$1 \cdot 10^{-6}$	-	-	True
RigL	$1 \cdot 10^{-6}$	1,000	0.5	True
SET	$1 \cdot 10^{-6}$	1,000	0.5	True

Table 2. SAC best hyper-parameters from sweep

Algorithm	Weight Decay	Update Interval	Drop Fraction	Sparsity-aware init
Dense	$1 \cdot 10^{-4}$	-	-	-
Pruning	$1 \cdot 10^{-4}$	1,000	-	True
Static	$1 \cdot 10^{-4}$	-	-	False
RigL	$1 \cdot 10^{-6}$	1,000	0.5	True
SET	$1 \cdot 10^{-4}$	250	0.3	True

Table 3. PPO best hyper-parameters from sweep

Algorithm	Weight Decay	Update Interval	Drop Fraction	Sparsity-aware init
Dense	$1 \cdot 10^{-6}$	-	-	-
Pruning	$1 \cdot 10^{-6}$	500	-	False
Static	0	-	-	True
RigL	$1 \cdot 10^{-4}$	250	0.3	True
SET	$1 \cdot 10^{-6}$	100	0	True

Atari hyper-parameters Due to computational constraints we did not search over hyper-parameters for the Atari environments, except for a small grid-search to tune the dense ResNet. The CNN architecture from Mnih et al. (2015) has been used in many prior works thus was already well tuned. The ResNet hyper-parameter sweep for the original dense model is detailed below:

1. **Weight decay:** Searched over the grid [0, 1e-6, 1e-5, 1e-4].
2. **Learning rate:** Searched over the grid [1e-4, 2.5e-4, 1e-3, 2.5e-3].

The final hyper-parameters we used for the Atari environments are shown in in Table 4 for the CNN and Table 5 for the ResNet.

Table 4. DQN (CNN) Atari hyper-parameters

Algorithm	Weight Decay	Learning Rate	Update Interval	Drop Fraction	Sparsity-aware init
Dense	0	$2.5 \cdot 10^{-4}$	-	-	-
Pruning	0	$2.5 \cdot 10^{-4}$	5,000	-	False
Static	0	$2.5 \cdot 10^{-4}$	-	-	True
RigL	0	$2.5 \cdot 10^{-4}$	5,000	0.3	True
SET	0	$2.5 \cdot 10^{-4}$	5,000	0.3	True

Table 5. DQN (ResNet) Atari hyper-parameters

Algorithm	Weight Decay	Learning Rate	Update Interval	Drop Fraction	Sparsity-aware init
Dense	$1 \cdot 10^{-5}$	$1 \cdot 10^{-4}$	-	-	-
Pruning	$1 \cdot 10^{-5}$	$1 \cdot 10^{-4}$	5,000	-	False
Static	$1 \cdot 10^{-5}$	$1 \cdot 10^{-4}$	-	-	True
RigL	$1 \cdot 10^{-5}$	$1 \cdot 10^{-4}$	5,000	0.3	True
SET	$1 \cdot 10^{-5}$	$1 \cdot 10^{-4}$	5,000	0.3	True

Remaining hyper-parameters Next, we include details of the DRL hyper-parameters used in all training settings for DQN (Table 6 and Table 7), SAC (Table 8), and PPO (Table 8).

Table 6. DQN Hyperparameters/ Table format from Haarnoja et al. (2018).

Parameter	Value
Shared	
optimizer	Adam (Kingma & Ba, 2015b)
discount (γ)	0.99
nonlinearity	ReLU
target smoothing coefficient (τ)	1.0
gradient steps per training step	1
exploration policy	epsilon greedy
epsilon decay period (env steps)	$2.5 \cdot 10^4$
Classic Control	
replay buffer size	10^5
learning rate	$1 \cdot 10^{-3}$
initial collect steps	1,000
target update interval	100
reward scale factor	1.0
gradient steps every k env steps, k =	1
final epsilon	0.1
eval epsilon	0.1
number of samples per minibatch	128
network type	MLP
number of hidden dense layers	2
number of hidden units per layer	512
Atari	
replay buffer size	10^6
initial collect steps	20,000
target update interval	8000
reward scale factor	1.0
gradient steps every k env steps, k =	4
final epsilon	0.01
eval epsilon	0.001
number of samples per minibatch	32
network type	CNN or ResNet

Table 7. DQN Atari: CNN and ResNet Architectures

Parameter	Value
CNN	
learning rate	$2.5 \cdot 10^{-4}$
Adam optimizer, epsilon	$1 \cdot 10^{-8}$
CNN Architecture	
number of hidden CNN layers	3
number of hidden dense layers	1
number of hidden units per dense layer	512
CNN params per layer (filters, kernel, stride)	
layer 1 (filters, kernel, stride)	32, 8, 4
layer 2 (filters, kernel, stride)	64, 4, 2
layer 3 (filters, kernel, stride)	64, 3, 1
ResNet	
learning rate	$1 \cdot 10^{-4}$
Adam optimizer, epsilon	$3.125 \cdot 10^{-4}$
ResNet Architecture	
number of stacks	3
number of hidden dense layers	1
number of hidden units per dense layer	512
use batch norm	False
ResNet stack layers	
num CNN layers	1
num max pooling layers	1
num residual-CNN layers	2
ResNet params per layer (filters, kernel, stride)	
stack 1 (filters, kernel, stride)	32, 3, 1
stack 2 (filters, kernel, stride)	64, 3, 1
stack 3 (filters, kernel, stride)	64, 3, 1

Table 8. SAC Hyperparameters.

Parameter	Value
optimizer	Adam (Kingma & Ba, 2015b)
learning rate	$3 \cdot 10^{-4}$
discount (γ)	0.99
replay buffer size	10^6
number of hidden layers (all networks)	2
number of hidden units per layer	256
number of samples per minibatch	256
nonlinearity	ReLU
target smoothing coefficient (τ)	0.005
target update interval	1
train every k env steps, k =	1
gradient steps per training step =	1
Hopper, Walker, Humanoid	
initial collect steps	1,000
HalfCheetah, Ant	
initial collect steps	10,000

Table 9. PPO Hyperparameters.

Parameter	Value
optimizer	Adam (Kingma & Ba, 2015b)
learning rate	$3 \cdot 10^{-4}$
discount (γ)	0.99
shared / separate networks	separate
number of hidden layers (all networks)	2
number of hidden units per layer	64
collect sequence length (batch size)	2048
minibatch size	64
num epochs	10
importance ratio clipping	0.2
use GAE (Schulman et al., 2016)	True
λ (GAE)	0.95
entropy regularization	0
value loss coeff	0.5
gradient clipping	0.5

A.4. Atari Game Selection

Our original three games (MsPacman, Pong, Qbert) were selected to have varying levels of difficulty as measured by DQN's human normalized score in Mnih et al. (2015), Figure 3. To this we added 12 games (Assault, Asterix, BeamRider, Boxing, Breakout, CrazyClimber, DemonAttack, Enduro, FishingDerby, SpaceInvaders, Tutankham, VideoPinball) selected to be roughly evenly distributed amongst the games ranked by DQN's human normalized score in Mnih et al. (2015) with a lower cut off of approximately 100% of human performance.

B. Sparse Scaling Plots in Other Environments

Here we share results on additional environments, Acrobot, CartPole, MountainCar, Hopper, and Ant. Figure 8 compares final reward relative to parameter count using DQN. ERK sparsity distribution was used in the top row whilst uniform was used in the bottom row. Figure 9 presents results on the two remaining MuJoCo environment, Hopper and Ant with SAC (top row) and PPO (bottom row). In Figures 10 and 11 we show sparsity scaling plots for 15 Atari games using the standard CNN and ResNet respectively.

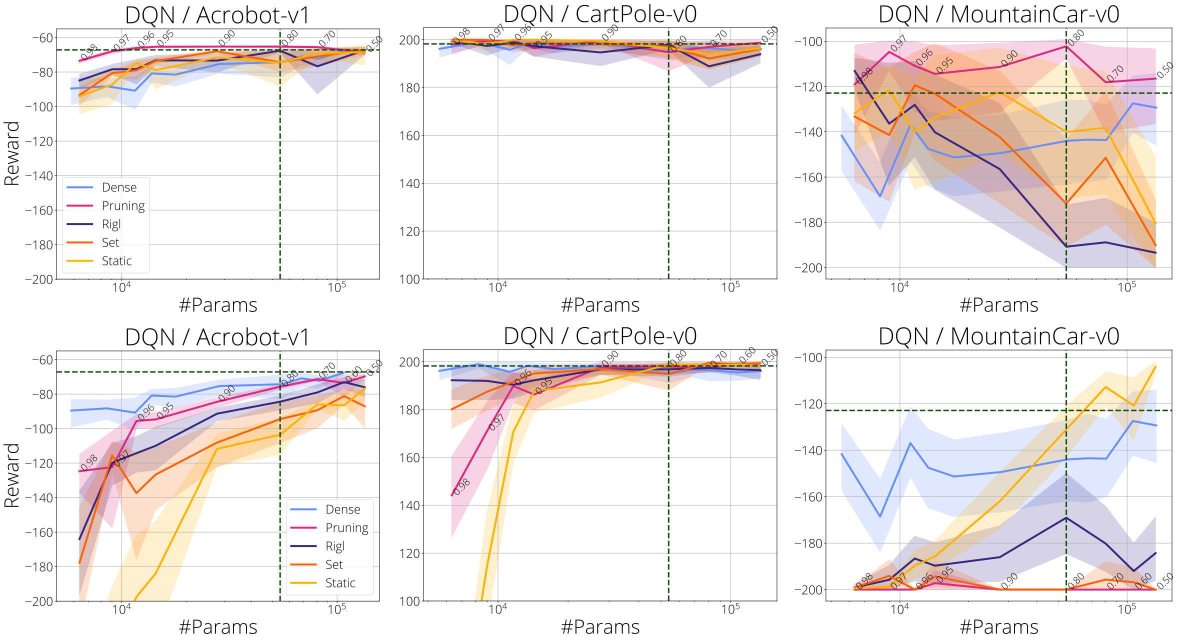

Figure 8. DQN in the Classic Control environments with ERK network sparsity distribution (top) and uniform network sparsity (bottom).

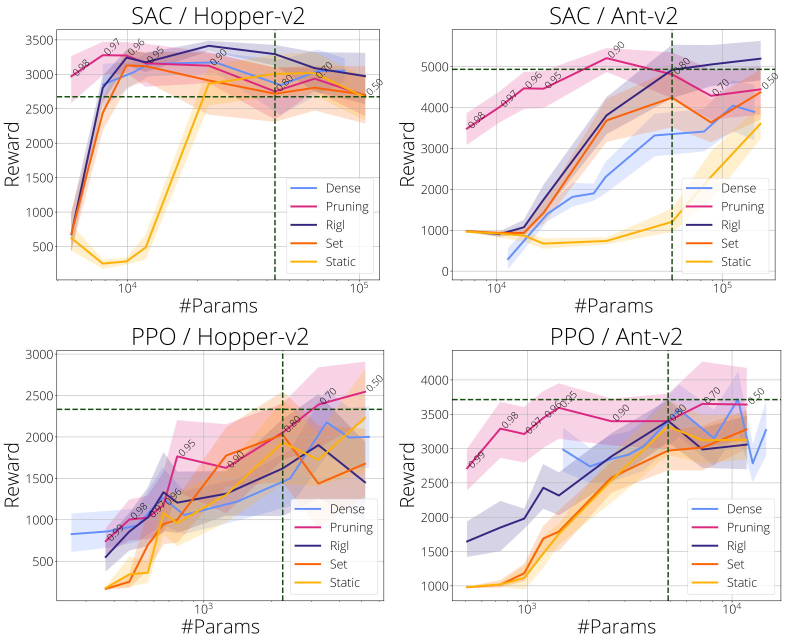

Figure 9. Additional MuJoCO environments (Hopper and Ant) for SAC and PPO algorithms. Networks are initialized with ERK network sparsity distribution.

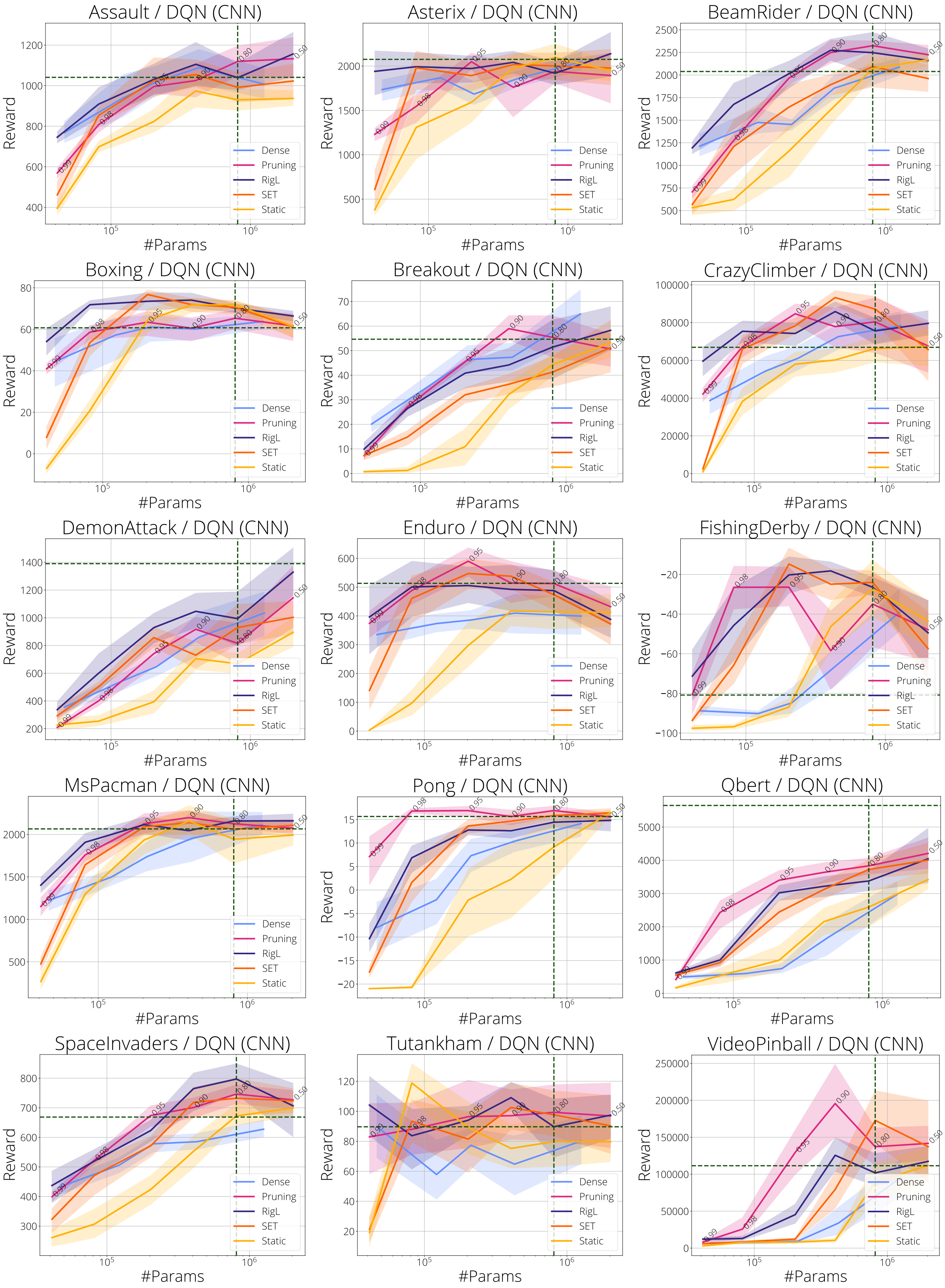

Figure 10. CNN: Sparsity plots per game.

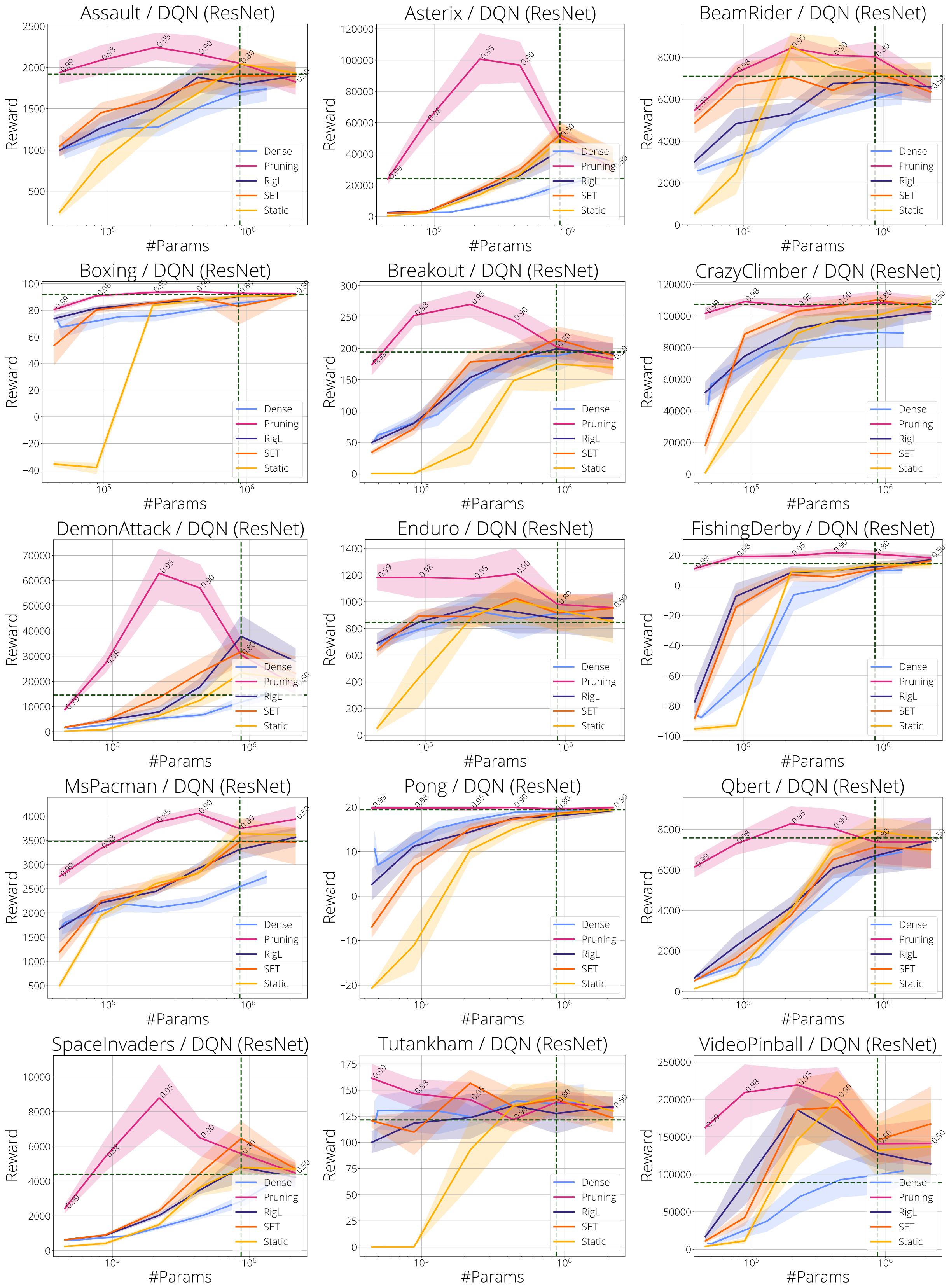

Figure 11. ResNet: Sparsity plots per game.

C. Additional Results with Uniform Sparsity Distribution

In Figure 12 we repeat the experiments presented in Figure 2, however this time using a uniform network sparsity distribution at initialization. These plots provide further evidence as the the benefit of using ERK over uniform to distribute network sparsity.

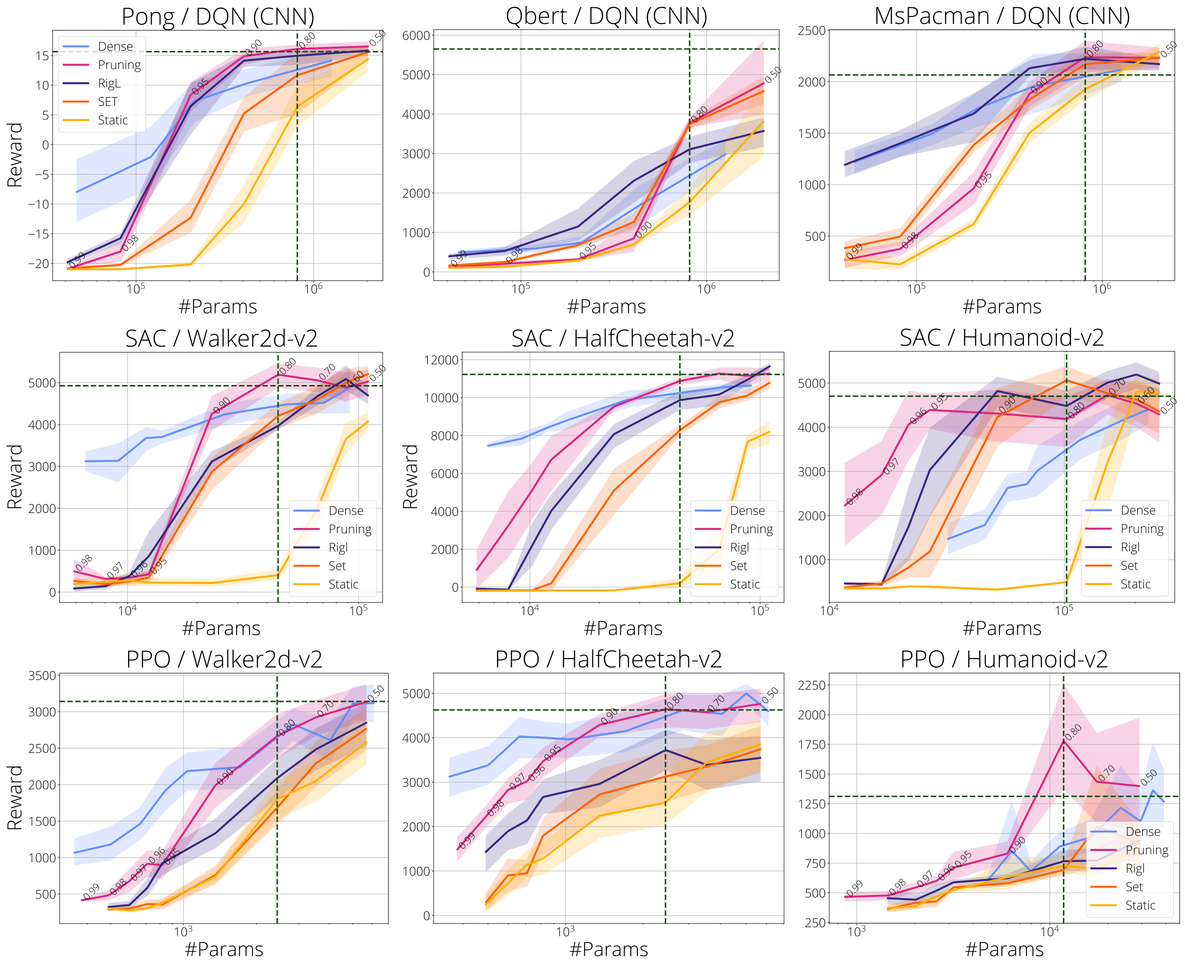

Figure 12. Uniform network sparsity initialization. Comparison of final reward relative to parameter count for the various methods considered: (row-1) DQN on Atari (row-2) SAC on MuJoCo (row-3) PPO on MuJoCo. We consider sparsities from 50% to 95% for sparse training methods and pruning. Parameter count for networks with 80% sparsity and the reward obtained by the dense baseline are highlighted with vertical and horizontal lines. Shaded areas represent 95% confidence intervals.

D. Additional Hyper-parameter Sensitivity Plots

Here we share the remaining plots for our analysis on the sensitivity of sparse training algorithms to various hyper-parameters. Policies area trained using SAC. We note that different architectures, environments and training algorithms might show different curves, which we omit due to high cost of running such analysis.

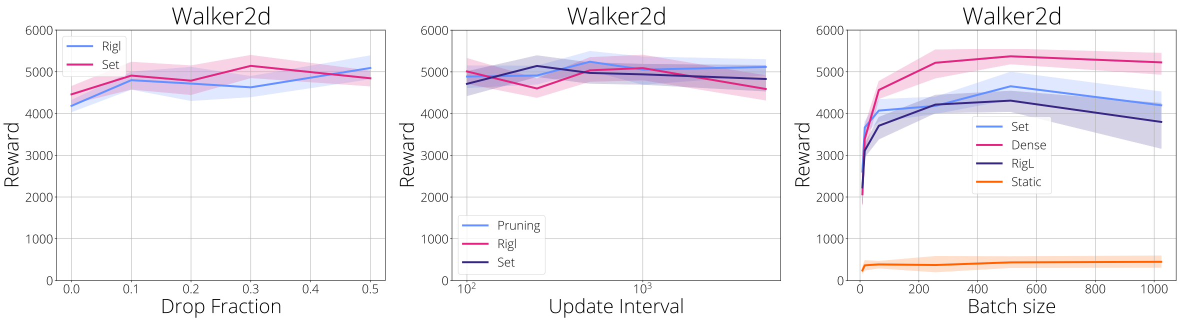

Figure 13. Sensitivity analysis: (left) Drop fraction: Comparing different drop fractions for SET and RigL at 80% sparsity. (center) Comparing the effect of topology update interval on pruning, static, and RigL. (right) The effect of batch size.

Drop fraction In Figure 13 (left) we evaluate the effect of drop fraction and observe that drop fractions > 0 yield a small performance improvement over drop fraction $= 0$, which is equivalent to static sparse training. This indicates that changing the network topology during training helps performance. Surprisingly training does not appear particularly sensitive to the drop fraction chosen, with values from 10 - 50% yielding approximately equivalent performance. It is possible that this is because the environment is relatively easy, thus leading to little separation between different settings. The large improvement that RigL and SET give over static (drop fraction = 0) in the Humanoid and Atari environments is one reason to suspect this.

Findings: Drop fractions of 10 - 50% worked well for RigL, whilst a drop fraction of 30% worked best for SET. 30% therefore appears to be a reasonable default for dynamic sparse training. However, the uniformity of performance merits investigation on a harder environment in which more separation between drop fractions may be observed.

Topology update interval In Figure 13 (center) we evaluate the effect of topology update interval on pruning, RigL and SET and find that training is not particularly sensitive to it.

Findings: Updating the network every 1000 environment steps appears to be a reasonable default.

Batch size In Figure 13 (right) we evaluate the effect of batch size. Batch size is critical for obtaining a good estimate for the gradients during training for all methods, however for RigL it is also used when selecting new connections. We observe performance degradation for all methods when smaller batch sizes are used, with no particular additional effect on RigL.

Findings: Sparse networks seems to have similar sensitivity to the batch size as the dense networks.

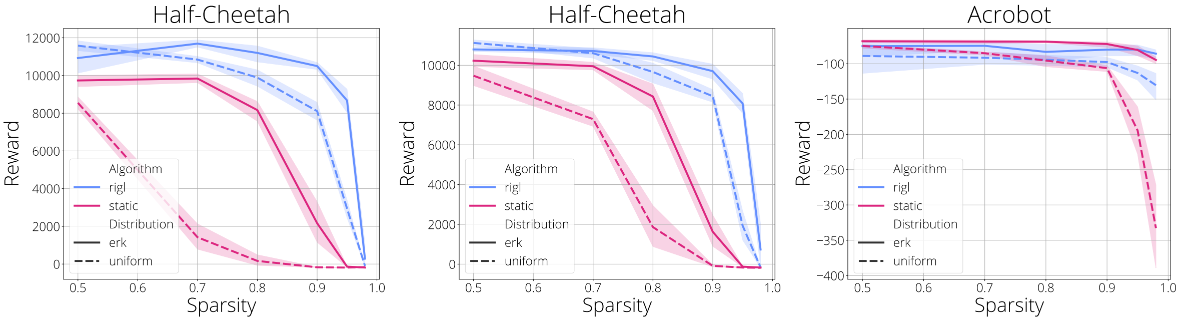

Figure 14. Evaluating non uniform sparsities on HalfCheetah using SAC when all layers pruned (left) and last layer is kept dense (center). On the right we prune all layers of the DQN MLP network on the Acrobot task.

Within network sparsity In Figure 14 we share additional analysis on within network sparsity for SAC and also on DQN in the Acrobot environment.

E. Additional Interquartile Mean (IQM) Plots

Mujoco SAC In Figure 15 we present the interquartile mean (IQM) calculated over five Mujoco environments for SAC at four different sparsities, 50%, 90%, 95% and 99%, or the networks with the equivalent number of parameters in the case of Dense training. Note that for 99% sparsity the IQM is only calculated over three Mujoco environments.

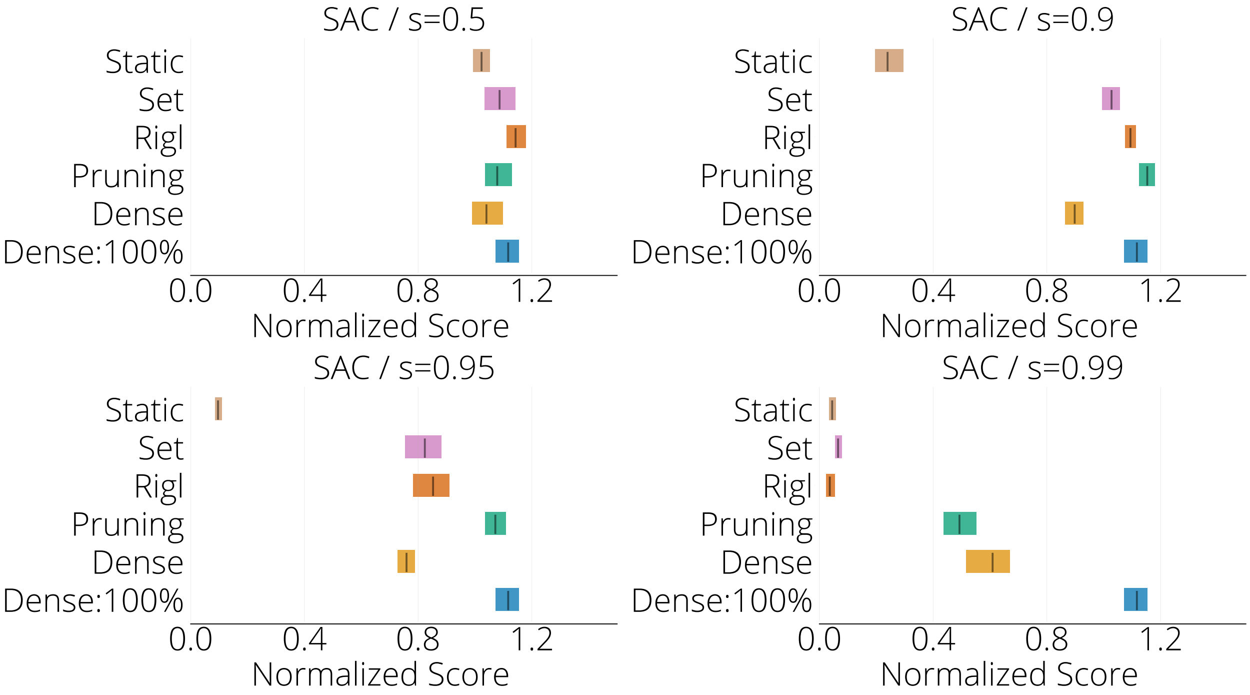

Figure 15. SAC: IQM plots calculated over five Mujoco games (Ant, HalfCheetah, Hopper, Humanoid, Walker2d), with 10 seeds per game. Except for sparsity = 0.99 which only includes results from HalfCheetah, Hopper, and Walker2d, each with 10 seeds.

Mujoco PPO In Figure 16 we present the interquartile mean (IQM) calculated over five Mujoco environments for PPO at four different sparsities, 50%, 90%, 95% and 99%, or the networks with the equivalent number of parameters in the case of Dense training. Note that for 95% sparsity the IQM is calculated over four environments and for 99% sparsity the IQM is only calculated over three Mujoco environments.

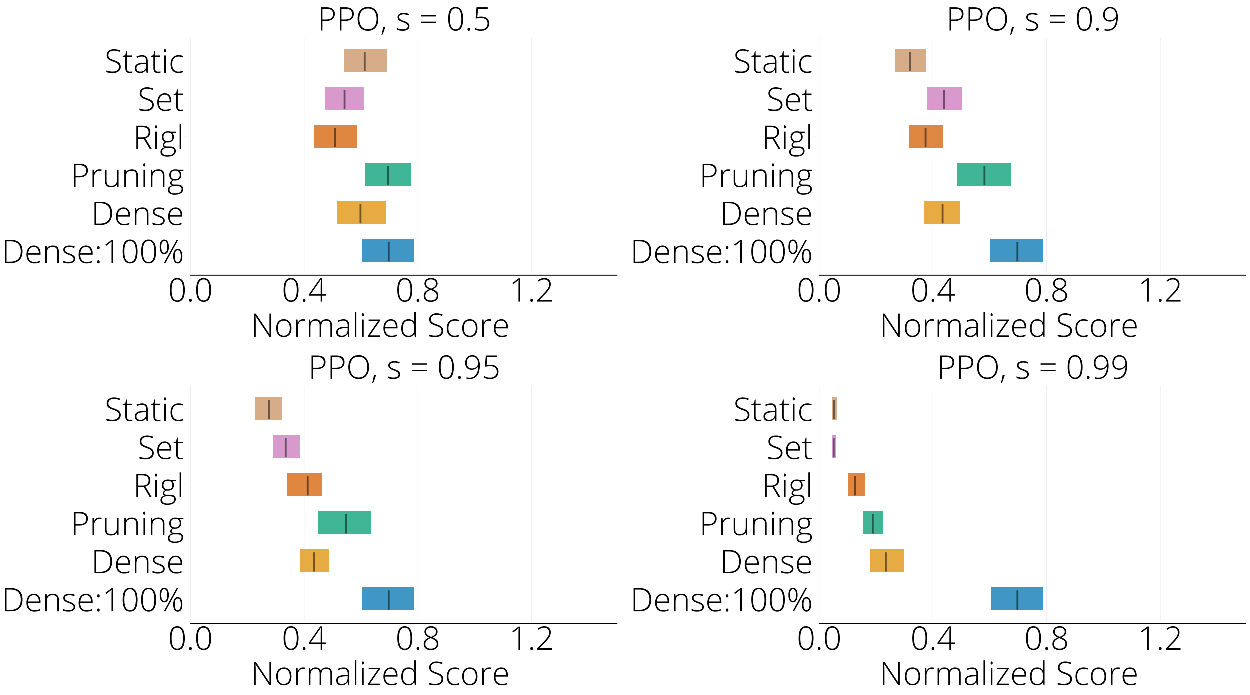

Figure 16. SAC: IQM plots calculated over multiple Mujoco games with 9 seeds per game. 50 - 90% sparsity include five games (Ant, HalfCheetah, Hopper, Humanoid, Walker2d), 95% sparsity includes Ant, HalfCheetah, Hopper, Walker2d, and 99% sparsity HalfCheetah, Hopper, Walker2d.

Atari DQN Figure 17 presents IQM plots calculated over 15 Atari games for the standard CNN network architecture and Figure 18 presents IQM plots calculated over the same set of games for a ResNet architecture with an approximately equivalent number of parameters as the standard CNN (≈4M).

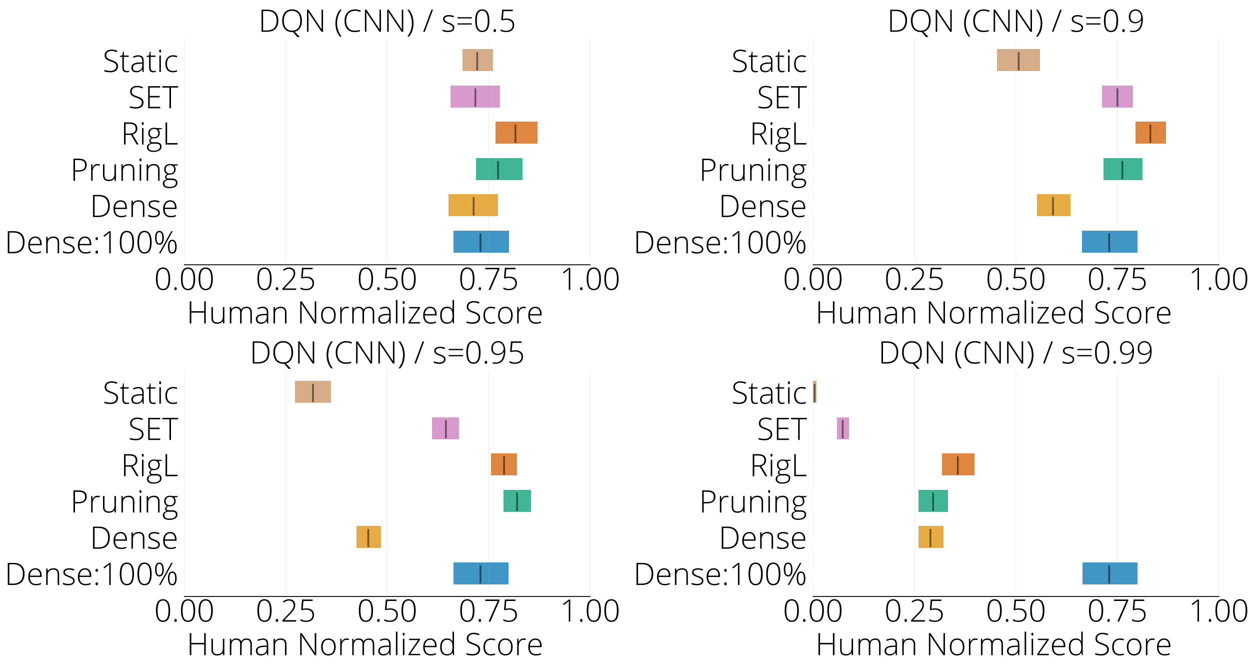

Figure 17. CNN: IQM plots calculated over 15 Atari Games, with 9 seeds per game.

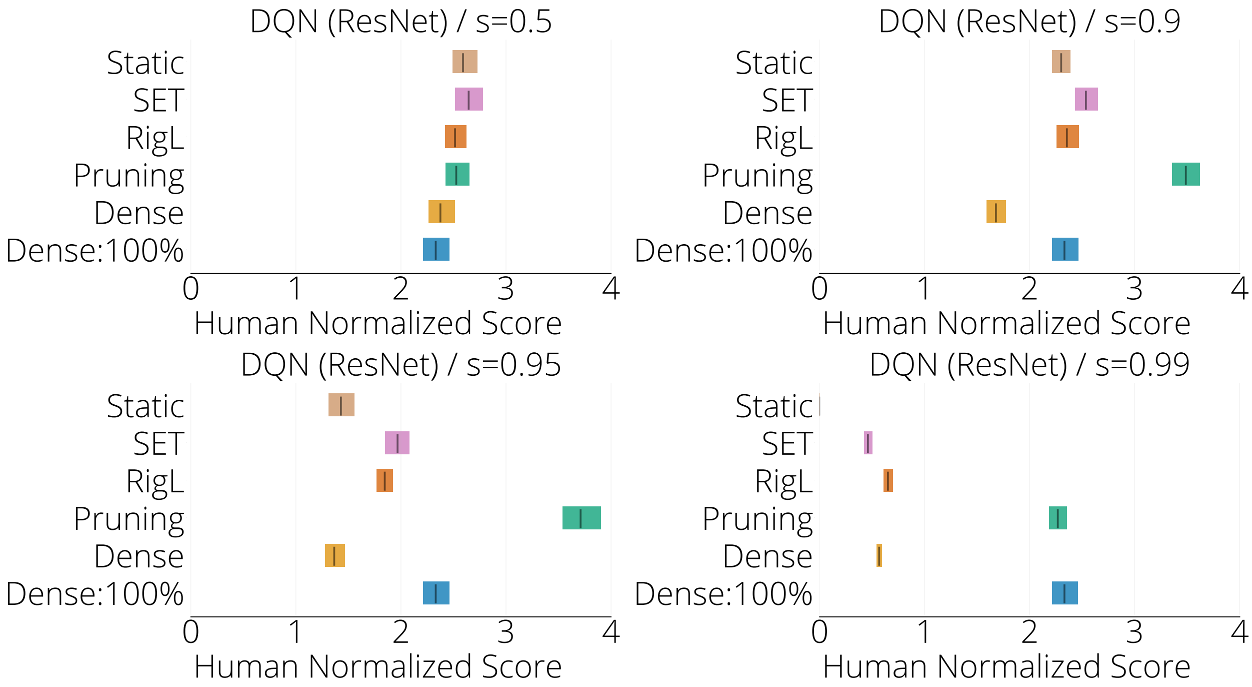

Figure 18. ResNet: IQM plots calculated over 15 Atari Games, with 9 seeds per game.