DeepMind RL

Human-level Atari 200x faster

Steven Kapturowski[1], **Víctor Campos**[*1], **Ray Jiang**[*1], **Nemanja Rakićević**[1], **Hado van Hasselt**[1], **Charles Blundell**[1] **and Adrià Puigdomènech Badia**[1]
[1]DeepMind, [*]Equal contribution

The task of building general agents that perform well over a wide range of tasks has been an important goal in reinforcement learning since its inception. The problem has been subject of research of a large body of work, with performance frequently measured by observing scores over the wide range of environments contained in the Atari 57 benchmark. Agent57 was the first agent to surpass the human benchmark on all 57 games, but this came at the cost of poor data-efficiency, requiring nearly 80 billion frames of experience to achieve. Taking Agent57 as a starting point, we employ a diverse set of strategies to achieve a 200-fold reduction of experience needed to outperform the human baseline. We investigate a range of instabilities and bottlenecks we encountered while reducing the data regime, and propose effective solutions to build a more robust and efficient agent. We also demonstrate competitive performance with high-performing methods such as Muesli and MuZero. The four key components to our approach are (1) an approximate trust region method which enables stable bootstrapping from the online network, (2) a normalisation scheme for the loss and priorities which improves robustness when learning a set of value functions with a wide range of scales, (3) an improved architecture employing techniques from NFNets in order to leverage deeper networks without the need for normalization layers, and (4) a policy distillation method which serves to smooth out the instantaneous greedy policy over time.

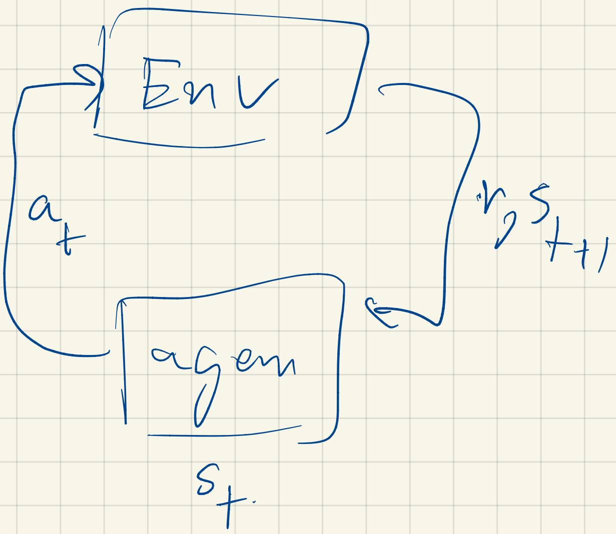
1. Introduction

To develop generally capable agents, the question of how to evaluate them is paramount. The Arcade Learning Environment (ALE) (Bellemare et al., 2013) was introduced as a benchmark to evaluate agents on an diverse set of tasks which are interesting to humans, and developed externally to the Reinforcement Learning (RL) community. As a result, several games exhibit reward structures which are highly adversarial to many popular algorithms. Mean and median human normalized scores (HNS) (Mnih et al., 2015) over all games in the ALE have become standard metrics for evaluating deep RL agents. Recent progress has allowed state-of-the-art algorithms to greatly exceed average human-level performance on a large fraction of the games (Espeholt et al., 2018; Schrittwieser et al., 2020; Van Hasselt et al., 2016). However, it has been argued that mean or median HNS might not be well suited to assess generality because they tend to ignore the tails of the distribution (Badia et al., 2019). Indeed, most state-of-the-art algorithms achieve very high scores by performing very well on most games, but completely fail to learn on a small number of them.

Agent57 (Badia et al., 2020) was the first algorithm to obtain above human-average scores on all 57 Atari games. However, such generality came at the cost of data efficiency; requiring tens of billions of environment interactions to achieve above average-human performance in some games, reaching a figure of 78 billion frames before beating the human benchmark in all games. Data efficiency remains a desirable property for agents to possess, as many real-world challenges are data-limited by time and cost constraints (Dulac-Arnold et al., 2019). In this work, our aim is to develop an agent that is as general as Agent57 but that requires only a fraction of the environment interactions to achieve the same result.

RL ⇒ ทำเพื่อ การประหยัด Resource

sample



RL > Human

ในบางงาน

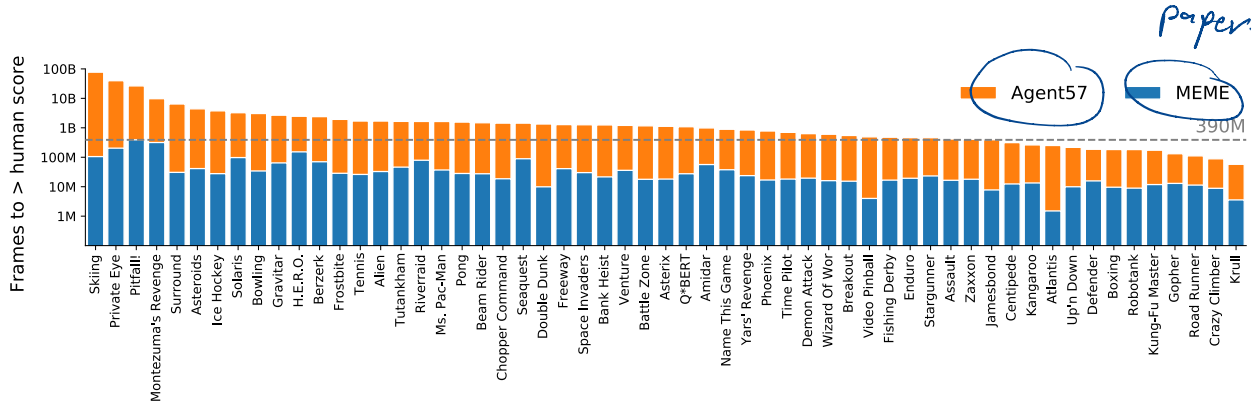


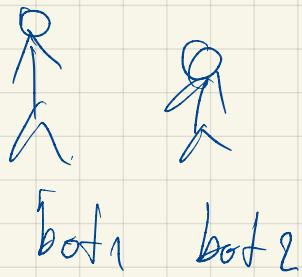
Figure 1 | Number of environment frames required by agents to outperform the human baseline on each game (in log-scale). Lower is better. On average, MEME achieves above human scores using 63× fewer environment interactions than Agent57. The smallest improvement is 9× (`road_runner`), the maximum is 721× (`skiing`), and the median across the suite is 35×. We observe small variance across seeds (c.f. Figure 8).

There exist two main trends in the literature when it comes to measuring improvements in the learning capabilities of agents. One approach consists in measuring performance after a limited budget of interactions with the environment. While this type of evaluation has led to important progress (Espeholt et al., 2018; Hessel et al., 2021; van Hasselt et al., 2019), it tends to disregard problems which are considered too hard to be solved within the allowed budget (Kaiser et al., 2019). On the other hand, one can aim to achieve a target end-performance with as few interactions as possible (Schmitt et al., 2020; Silver et al., 2017, 2018). Since our goal is to show that our new agent is as general as Agent57, while being more data efficient, we focus on the latter approach.

Our contributions can be summarized as follows. Building off Agent57, we carefully examine bottlenecks which slow down learning and address instabilities that arise when these bottlenecks are removed. We propose a novel agent that we call MEME, for *MEME is an Efficient Memory-based Exploration agent*, which introduces solutions to enable taking advantage of three approaches that would otherwise lead to instabilities: training the value functions of the whole family of policies from Agent57 in parallel, on all policies' transitions (instead of just the behaviour policy transitions), bootstrapping from the online network, and using high replay ratios. These solutions include carefully normalising value functions with differing scales, as well as replacing the Retrace (Munos et al., 2016) update target with a soft variant of Watkins' Q(λ) (Watkins and Dayan, 1992) that enables faster signal propagation by performing less aggressive trace-cutting, and introducing a trust-region for value updates. Moreover, we explore several recent advances in deep learning and determine which of them are beneficial for non-stationary problems like the ones considered in this work. Finally, we examine approaches to robustify performance by introducing a policy distillation mechanism that learns a policy head based on the actions obtained from the value network without being sensitive to value magnitudes. Our agent outperforms the human baseline across all 57 Atari games in 390M frames, using two orders of magnitude fewer interactions with the environment than Agent57 as shown in Figure 1.

2. Related work

Large scale distributed agents have exhibited compelling results in recent years. Actor-critic (Espeholt et al., 2018; Song et al., 2019) as well as value-based agents (Horgan et al., 2018; Kapturowski et al., 2018) demonstrated strong performance in a wide-range of environments, including the Atari


bot1
bot2

57 benchmark. Moreover, approaches such as evolutionary strategies (Salimans et al., 2017) and large scale genetic algorithms (Such et al., 2017) presented alternative learning algorithms that achieve competitive results on Atari. Finally, search-augmented distributed agents (Hessel et al., 2021; Schrittwieser et al., 2020) also hold high performance across many different tasks, and concretely they hold the highest mean and median human normalized scores over the 57 Atari games. However, all these methods show the same failure mode: they perform poorly in hard exploration games, such as *Pitfall!*, and *Montezuma's Revenge*. In contrast, Agent57 (Badia et al., 2020) surpassed the human benchmark on all 57 games, showing better general performance. Go-Explore (Ecoffet et al., 2021) similarly achieved such general performance, by relying on coarse-grained state representations via a downscaling function that is highly specific to Atari.

Learning as much as possible from previous experience is key for data-efficiency. Since it is often desirable for approximate methods to make small updates to the policy (Kakade and Langford, 2002; Schulman et al., 2015), approaches have been proposed for enabling multiple learning steps over the same batch of experience in policy gradient methods to avoid collecting new transitions for every learning step (Schulman et al., 2017). This decoupling between collecting experience and learning occurs naturally in off-policy learning agents with experience replay (Lin, 1992; Mnih et al., 2015) and Fitted Q Iteration methods (Ernst et al., 2005; Riedmiller, 2005). Multiple approaches for making more efficient use of a replay buffer have been proposed, including prioritized sampling of transitions (Schaul et al., 2015b), sharing experience across populations of agents (Schmitt et al., 2020), learning multiple policies in parallel from a single stream of experience (Riedmiller et al., 2018), or reanalyzing old trajectories with the most recent version of a learned model to generate new targets in model-based settings (Schrittwieser et al., 2020, 2021) or to re-evaluate goals (Andrychowicz et al., 2017).

The ATARI100k benchmark (Kaiser et al., 2019) was introduced to observe progress in improving the data efficiency of reinforcement learning agents, by evaluating game scores after 100k agent steps (400k frames). Work on this benchmark has focused on leveraging the use of models (Kaiser et al., 2019; Long et al., 2022; Ye et al., 2021), unsupervised learning (Hansen et al., 2019; Liu and Abbeel, 2021; Schwarzer et al., 2020; Srinivas et al., 2020), or greater use of replay data (Kielak, 2020; van Hasselt et al., 2019) or augmentations (Kostrikov et al., 2020; Schwarzer et al., 2020). While we consider this to be an important line of research, this tight budget produces an incentive to focus on a subset of games where exploration is easier, and it is unclear some games can be solved from scratch with such a small data budget. Such a setting is likely to prevent any meaningful learning on hard-exploration games, which is in contrast with the goal of our work.

3. Background: Agent57

Our work builds on top of Agent57, which combines three main ideas: (*i*) a distributed deep RL framework based on Recurrent Replay Distributed DQN (R2D2) (Kapturowski et al., 2018), (*ii*) exploration with a family of policies and the Never Give Up (NGU) intrinsic reward (Badia et al., 2019), and (*iii*) a meta-controller that dynamically adjusts the discount factor and balances exploration and exploitation throughout the course of training, by selecting from a family of policies. Below, we give a general introduction to the problem setting and some of the relevant components of Agent57.

Problem definition. We consider the problem of discounted infinite-horizon RL in Markov Decision Processes (MDP) (Puterman, 1994). The goal is to find a policy π that maximises the expected sum of future discounted rewards, $\mathbb{E}_\pi[\sum_{t\geq 0}\gamma^t r_t]$, where $\gamma \in [0,1)$ is the discount factor, $r_t = r(x_t, a_t)$ is the reward at time t, x_t is the state at time t, and $a_t \sim \pi(a|x_t)$ is the action generated by following some policy π. In the off-policy learning setting, data generated by a behavior policy μ is used to

learn about the target policy π. This can be achieved by employing a variant of Q-learning (Watkins and Dayan, 1992) to estimate the action-value function, $Q^\pi(x,a) = \mathbb{E}_\pi[\sum_{t\geq 0}\gamma^t r_t | x_t = x, a_t = a]$. The estimated action-value function can then be used to derive a new policy $\pi(a|x)$ using the ϵ-greedy operator $\mathcal{G}_\epsilon$ (Sutton and Barto, 2018). [1] This new policy can then be used as target policy for another iteration, repeating the process. Agent57 uses a deep neural network with parameters θ to estimate action-value functions, $Q^\pi(x,a;\theta)$ [2] , trained on return estimates G_t derived with Retrace from sequences of off-policy data (Munos et al., 2016). In order to stabilize learning, a target network is used for bootstrapping the return estimates using double Q-learning (Van Hasselt et al., 2016); the parameters of this target network, θ_T, are periodically copied from the online network parameters θ (Mnih et al., 2015).

Distributed RL framework. Agent57 is a distributed deep RL agent based on R2D2 that decouples acting from learning. Multiple actors interact with independent copies of the environment and feed trajectories to a central replay buffer. A separate learning process obtains trajectories from this buffer using prioritized sampling and updates the neural network parameters to predict the action-value function at each state. Actors obtain parameters from the learner periodically. See Appendix D for more details.

Exploration with NGU. Agent57 uses the Never Give Up (NGU) intrinsic reward to encourage exploration. It aims at learning a family of $N = 32$ policies which maximize different weightings of the extrinsic reward given by the environment (r^e_t) and the intrinsic reward (r^i_t), $r_{j,t} = r^e_t + \beta_j r^i_t$ ($\beta_j \in \mathbb{R}^+$, $j \in \{0,\dots,N-1\}$). The value of β_j controls the degree of exploration, with higher values encouraging more exploratory behaviors, and each policy in the family is optimized with a different discount factor γ_j. The Universal Value Function Approximators (UVFA) framework (Schaul et al., 2015a) is employed to efficiently learn $Q^{\pi^j}(x,a;\theta) = \mathbb{E}_{\pi^j}[\sum_{t\geq 0}\gamma^t_j r_{j,t} | x_t = x, a_t = a]$ (we use a shorthand notation $Q^j(x,a;\theta)$) for $j \in \{0,\dots,N-1\}$ using a single set of shared parameters θ. The policy $\pi^j(a|x)$ can then be derived using the ϵ-greedy operator as $\mathcal{G}_\epsilon Q^j(x,a;\theta)$. We refer the reader to Appendix G for more details.

Meta-controller. Agent57 introduces an adaptive meta-controller that decides which policies from the family of N policies to use for collecting experience based on their episodic returns. This naturally creates a curriculum over β_j and γ_j by adapting their value throughout training. This optimization process is formulated as a non-stationary bandit problem. A detailed description about the meta-controller implementation is provided in Appendix E.

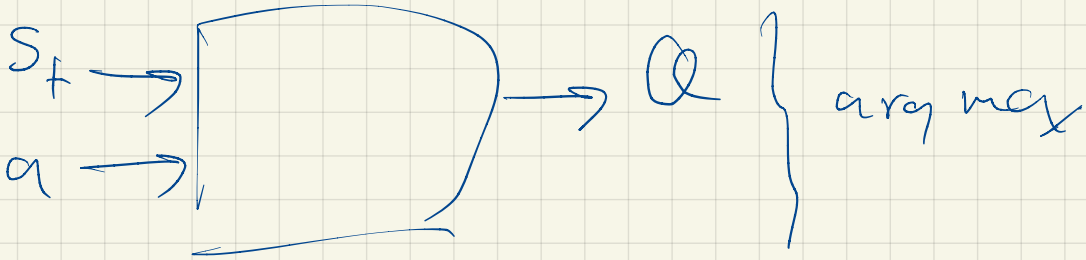
Q-function separation. The architecture of the Q-function in Agent57 is implemented as two separate networks in order to split the intrinsic and extrinsic components. The network parameters of $Q^j(x,a;\theta_e)$ and $Q^j(x,a;\theta_i)$ are separate and independently optimized with r^e_j and r^i_j, respectively. The main motivation behind this decomposition is to prevent the gradients of the decomposed intrinsic and extrinsic value function heads from interfering with each other. This can be beneficial in environments where the intrinsic reward is poorly aligned with the task's extrinsic reward.

4. MEME: Improving the data efficiency of Agent57

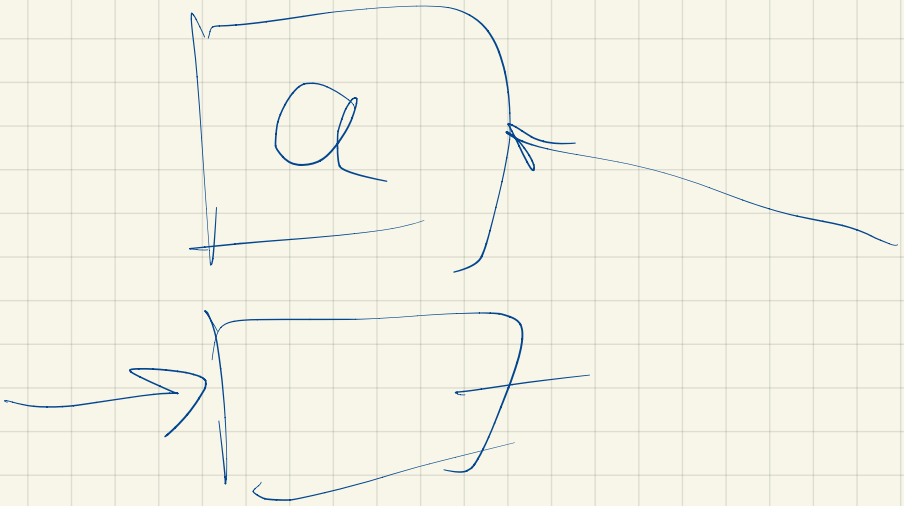
This section describes the main algorithmic contributions of the MEME agent, aimed at improving the data-efficiency of Agent57. These contributions aim to achieve faster propagation of learning signals related to rare events (**A**), stabilize learning under differing value scales (**B**), improve the neural network architecture (**C**), and make updates more robust under a rapidly-changing policy (**D**). For

[1] We also use $\mathcal{G} := \mathcal{G}_0$ to denote the pure greedy operator.

[2] For convenience, we occasionally omit (x,a) or θ from $Q(x,a;\theta)$, $\pi(x,a;\theta)$ when there is no danger of confusion.



Q → learning on-policy



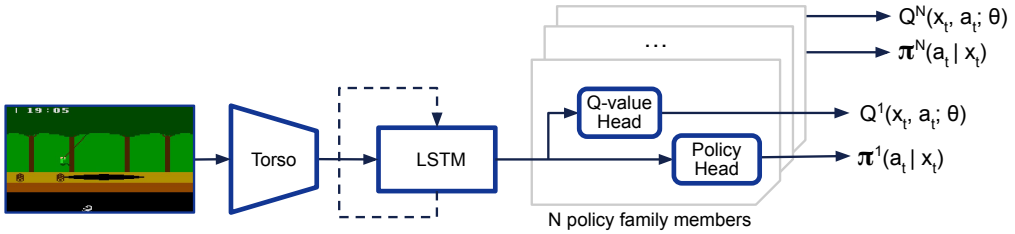


Figure 2 | MEME agent network architecture. The output of the LSTM block is passed to each of the N members of the family of policies, depicted as a light-grey box. Each policy consists of an Q-value and policy heads. The Q-value head is similar as in Agent57 paper, while the policy head is introduced for acting and target computation, and trained via policy distillation.

clarity of exposition, we label methods according to the type of limitation they address.

A1 Bootstrapping with online network. Target networks are frequently used in conjunction with value-based agents due to their stabilizing effect when learning from off-policy data (Mnih et al., 2015; Van Hasselt et al., 2016). This design choice places a fundamental restriction on how quickly changes in the Q-function are able to propagate. This issue can be mitigated to some extent by simply updating the target network more frequently, but the result is typically a less stable agent. To accelerate signal propagation while maintaining stability, we use online network bootstrapping, and we stabilise the learning by introducing an approximate trust region for value updates that allows us to filter which samples contribute to the loss. The trust region masks out the loss at any timestep for which *both* of the following conditions hold:

$$|Q^j(x_t, a_t; \theta) - Q^j(x_t, a_t; \theta_T)| > \alpha\sigma_j \tag{1}$$

$$\text{sgn}(Q^j(x_t, a_t; \theta) - Q^j(x_t, a_t; \theta_T)) \neq \text{sgn}(Q^j(x_t, a_t; \theta) - G_t) \tag{2}$$

where α is a fixed hyperparameter, G_t denotes the return estimate, θ and θ_T denote the online and target parameters respectively, and σ_j is the standard deviation of the TD-errors (a more precise description of which we defer until **B1**). Intuitively, we only mask if *the current value of the online network is outside of the trust region* (Equation 1) *and the sign of the TD-error points away from the trust region* (Equation 2), as depicted in Figure 3 in red. We note that a very similar trust region scheme is used for the value-function in most Proximal Policy Optimization (PPO) implementations (Schulman et al., 2017), though not described in the original paper. In contrast, the PPO version instead uses a constant threshold, and thus is not able to adapt to differing scales of value functions.

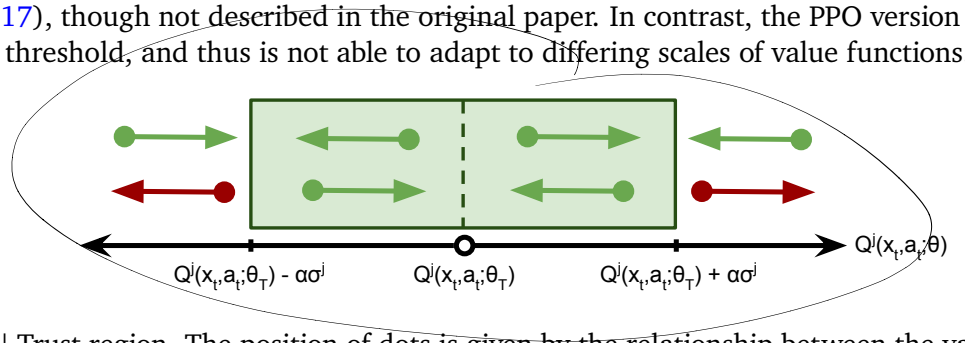


Figure 3 | Trust region. The position of dots is given by the relationship between the values predicted by the online network, $Q^j(x_t, a_t; \theta)$, and the values predicted by the target network, $Q^j(x_t, a_t; \theta_T)$ (Equation 1 and left hand side of Equation 2), the box represents the trust region bounds defined in Equation 1, and the direction of the arrow is given by the right hand side of Equation 2. Green-colored transitions are used in the loss computation, whereas red ones are masked out.

A2 Target computation with tolerance. Agent57 uses Retrace (Munos et al., 2016) to compute return estimates from off-policy data, but we observed that it tends to cut traces too aggressively when

using ϵ-greedy policies thus slowing down the propagation of information into the value function. Preliminary experiments showed that data-efficiency was improved in many dense-reward tasks when replacing Retrace with Peng's Q(λ) (Peng and Williams, 1994), but its lack of off-policy corrections tends to result in degraded performance as data becomes more off-policy (e.g. by increasing the expected number of times that a sequence is sampled from replay, or by sharing data across a family of policies). This motivates us to propose an alternative return estimator, which we derive from Q(λ) (Watkins and Dayan, 1992):

$$G_t = \max_a Q(x_t, a) + \sum_{k \geq 0} \left(\prod_{i=0}^{k-1} \lambda_i \right) \gamma^k (r_{t+k} + \gamma \max_a Q(x_{t+k+1}, a) - \max_a Q(x_{t+k}, a)) \tag{3}$$

where $\prod_{i=0}^{k-1} \lambda_i \in [0, 1]$ effectively controls how much information from the future is used in the return estimation and is generally used as a trace cutting coefficient to perform off-policy correction. Peng's Q(λ) does not perform any kind of off-policy correction and sets $\lambda_i = \lambda$, whereas Watkins' Q(λ) (Watkins and Dayan, 1992) aggressively cuts traces whenever it encounters an off-policy action by using $\lambda_i = \lambda \mathbb{1}_{a_i \in \text{argmax}_a Q(x_i, a)}$, where $\mathbb{1}$ denotes the indicator function. We propose to use a softer trace cutting mechanism by adding a fixed tolerance parameter κ and taking the expectation of trace coefficients under π:

$$\lambda_i = \lambda \mathbb{E}_{a \sim \pi(a|x_t)} \left[\mathbb{1}_{[Q(x_t, a_t; \theta) \geq Q(x_t, a; \theta) - \kappa |Q(x_i, a; \theta)|]} \right] \tag{4}$$

Finally, we replace all occurrences of the max operator in Equation 3 with the expectation under π. The resulting return estimator, which we denote *Soft Watkins Q(λ)*, leads to more transitions being used and increased sample efficiency. Note that Watkins Q(λ) is recovered when setting $\kappa = 0$ and $\pi = \mathcal{G}(Q)$.

B1 Loss and priority normalization. As we learn a family of Q-functions which vary over a wide range of discount factors and intrinsic reward scales, we expect that the Q-functions will vary considerably in scale. This may cause the larger-scale Q-values to dominate learning and destabilize learning of smaller Q-values. This is a particular concern in environments with very small extrinsic reward scales. To counteract this effect we introduce a normalization scheme on the TD-errors similar to that used in Schaul et al. (2021). Specifically, we compute a running estimate of the standard deviation of TD-errors of the online network $\sigma_j^{\text{running}}$ as well as a batch standard deviation σ_j^{batch}, and compute $\sigma_j = \max(\sigma_j^{\text{running}}, \sigma_j^{\text{batch}}, \epsilon)$, where ϵ acts as small threshold to avoid amplification of noise past a specified scale, which we fix to 0.01 in all our experiments. We then divide the TD-errors by σ_j when computing both the loss and priorities. As opposed to Schaul et al. (2021) we compute the running statistics on the learner, and make use of importance sampling to correct the sampling distribution.

B2 Cross-mixture training. Agent57 only trains the policy j which was used to collect a given trajectory, but it is natural to ask whether data-efficiency and robustness may be improved by training all policies at once. We propose a training loss L according to the following weighting scheme between the behavior policy loss and the mean over all losses:

$$L = \eta L_{j_\mu} + \frac{1 - \eta}{N} \sum_{j=0}^{N-1} L_j \tag{5}$$

where L_j denotes the Q-learning loss for policy j, and j_μ denotes the index for the behavior policy selected by the meta-controller for the sampled trajectory. We find that an intermediate value for the mixture parameter of $\eta = 0.5$ tends to work well. To achieve better compute-efficiency we choose to deviate from the original UVFA architecture which fed a 1-hot encoding of the policy index to the LSTM, aand instead modify the Q-value heads to output N sets of Q-values, one for each of the

members in the family of policies introduced in Section 3. Therefore, in the end we output values for all combinations of actions and policies (see Figure 2). We note that in this setting, there is also less deviation in the recurrent states when learning across different mixture policies.

C1 Normalizer-free torso network. Normalization layers are a common feature of ResNet architectures, and which are known to aid in training of very deep networks, but preliminary investigation revealed that several commonly used normalization layers are in fact detrimental to performance in our setting. Instead, we employ a variation of the NFNet architecture (Brock et al., 2021) for our policy torso network, which combines a variance-scaling strategy with scaled weight standardization and careful initialization to achieve state-of-the-art performance on ImageNet without the need for normalization layers. We adopt their use of stochastic depth (Huang et al., 2016) at training-time but omit the application of ordinary dropout to fully-connected layers as we did not observe any benefit from this form of regularization. Some care is required when using stochastic depth in conjunction with multi-step returns, as resampling of the stochastic depth mask at each timestep injects additional noise into the bootstrap values, resulting in a higher-variance return estimator. As such, we employ a *temporally-consistent* stochastic depth mask which remains fixed over the length of each training trajectory.

C2 Shared torso with combined loss. Agent57 decomposes the combined Q-function into intrinsic and extrinsic components, Q_e and Q_i, which are represented by separate networks. Such a decomposition prevents the gradients of the decomposed value functions from interfering with each other. This interference may occur in environments where the intrinsic reward is poorly aligned with the task objective, as defined by the extrinsic reward. However, the choice to use separate separate networks comes at an expensive computational cost, and potentially limits sample-efficiency since generic low-level features cannot be shared. To alleviate these issues, we introduce a shared torso for the two Q-functions while retaining separate heads.

While the form of the decomposition in Agent57 was chosen so as to ensure convergence to the optimal value-function $Q^\star$ in the tabular setting, this does not generally hold under function approximation. Comparing the combined and decomposed losses we observe a mismatch in the gradients due to the absence of cross-terms $Q_i(\theta)\frac{\partial Q_e(\theta)}{\partial\theta}$ and $Q_e(\theta)\frac{\partial Q_i(\theta)}{\partial\theta}$ in the decomposed loss:

$$\underbrace{\frac{\partial}{\partial\theta}\left[\tfrac{1}{2}(Q(\theta)-G)^2\right]}_{\text{combined loss}} \neq \underbrace{\frac{\partial}{\partial\theta}\left[\tfrac{1}{2}(Q_e(\theta)-G_e)^2+\tfrac{1}{2}(\beta Q_i(\theta)-\beta G_i)^2\right]}_{\text{decomposed loss}} \tag{6}$$

$$\left[Q_e(\theta)+\beta Q_i(\theta)-G\right]\frac{\partial}{\partial\theta}\left[Q_e(\theta)+\beta Q_i(\theta)\right] \neq \left[Q_e(\theta)-G_e\right]\frac{\partial Q_e(\theta)}{\partial\theta}+\beta^2\left[Q_i(\theta)-G_i\right]\frac{\partial Q_i(\theta)}{\partial\theta} \tag{7}$$

Since we use a behavior policy induced by the total Q-function $Q = Q_e + \beta Q_i$ rather than the individual components, theory would suggest to use the combined loss instead. In addition, from a practical implementation perspective, this switch to the combined loss greatly simplifies the design choices involved in our proposed trust region method described in **A1**. The penalty paid for this choice is that the decomposition of the value function into extrinsic and intrinsic components no longer carries a strict semantic meaning. Nevertheless we do still retain an implicit inductive bias induced by multiplication of Q_i with the intrinsic reward weight β^j.

D Robustifying behavior via policy distillation. Schaul et al. (2022) describe the effect of policy churn, whereby the greedy action of value-based RL algorithms may change frequently over consecutive parameter updates. This can have a deleterious effect on off-policy correction methods: traces will be cut more aggressively than with a stochastic policy, and bootstrap values will change frequently which can result in a higher variance return estimator. In addition, our choice of training with temporally-consistent stochastic depth masks can be interpreted as learning an implicit ensemble

of Q-functions; thus it is natural to ask whether we may see additional benefit from leveraging the policy induced by this ensemble.

We propose to train an explicit policy head π_{dist} (see Figure 2) via policy distillation to match the ϵ-greedy policy induced by the Q-function. In expectation over multiple gradient steps this should help to smooth out the policy over time, as well as over the ensemble, while being much faster to evaluate than the individual members of the ensemble. Similarly to the trust-region described in **A1**, we mask the policy distillation loss at any timestep where a KL constraint C_{KL} is violated:

$$L_\pi = -\sum_{a,t} \mathcal{G}_\epsilon\big(Q(x_t, a; \theta)\big) \log \pi_{\text{dist}}(a|x_t; \theta) \quad \forall t \ s.t. \ KL\big(\pi_{\text{dist}}(a|x_t; \theta_T))||\pi_{\text{dist}}(a|x_t; \theta)\big) \leq C_{\text{KL}} \tag{8}$$

We use a fixed value of $\epsilon = 10^{-4}$ to act as a simple regularizer to prevent the policy logits from growing too large. We then use $\pi'_{\text{dist}} = \text{softmax}(\frac{\log \pi_{\text{dist}}}{\tau})$ as the target policy in our *Soft Watkins Q(λ)* return estimator, where τ is a fixed temperature parameter. We tested values of τ in the range $[0, 1]$ and found that any choice of τ in this range yields improvement compared to not using the distilled policy, but values closer to 1 tend to exhibit greater stability, while those closer to 0 tend to learn more efficiently. We settled on an intermediate $\tau = 0.25$ to help balance between these two effects. We found that sampling from π'_{dist} at behavior-time was less effective in some sparse reward environments, and instead opt to have the agent act according to $\mathcal{G}_\epsilon(\pi_{\text{dist}})$.

5. Experiments

Methods proposed in Section 4 aim at improving the data-efficiency of Agent57, but such efficiency gains must not come at the cost of end performance. For this reason, we train our agents with a budget of 1B environment frames[3]. This budget allows us to validate that the asymptotic performance is maintained, i.e. the agent converges and is stable when improving data-efficiency. Hyperparameters have been tuned over a subset of eight games, encompassing games with different reward density, scale and requiring credit assignment over different time horizons: *Frostbite*, *H.E.R.O.*, *Montezuma's Revenge*, *Pitfall!*, *Skiing*, *Solaris*, *Surround*, *Tennis*. In particular, due to the improvements in stability, we find it beneficial to use a higher samples per insert ratio (SPI) than the published values of Agent57 or R2D2. For all our experiments SPI is fixed to 6. An ablation on the effect of different SPI values can be found on Appendix K. An exhaustive description of the hyperparameters used is provided in Appendix A, and the network architecture in Appendix B. We report results averaged over 6 seeds on all games for our best agent, and over three seeds for ablation experiments. We also show similar results with *sticky actions* (Machado et al., 2018) in Appendix J.

5.1. Summary of results

In this section, we show that the proposed agent is able to outperform the human benchmark in all 57 Atari games, in only 390M frames. This is a significant speed-up compared to Agent57 which requires 78B frames to achieve this benchmark. Figure 1 gives further details per game, about the required number of frames to reach the human baseline, and shows that hard exploration games such as *Private Eye*, *Pitfall!* and *Montezuma's Revenge* pose the hardest challenge for both agents, being among the last ones in which the human baseline is surpassed. This can be explained by the fact that in these games the agent might need a very large number of episodes before it is able to generate useful experience from which to learn. Notably, our agent is able to reduce the number of interactions required to outperform the human baseline in each of the 57 Atari games, by 63× on average. In addition to the improved sample-efficiency, our agent shows competitive performance

[3] This corresponds to 250M agent-environment interactions due to the standard action repeat of 4 in the Atari benchmark.

Table 1 | Number of games above human, capped mean, mean and median human normalized scores for the 57 Atari games. Similarly to Badia et al. (2020), we first compute the final score per game by averaging results from all random seeds, and then aggregate the scores of all 57 games. We sample three out of the six seeds per game without replacement and report the average as well as 95% confidence intervals over 10,000 repetitions of this process. Metrics for previous methods are computed using the mean final score per game reported in their respective publications: Agent57 (Badia et al., 2020), MuZero 20B (Schrittwieser et al., 2020), MuZero 200M (Schrittwieser et al., 2021), Muesli (Hessel et al., 2021).

	200M frames			> 200M frames		
Statistic	**MEME**	**Muesli**	**MuZero**	**MEME**	**Agent57**	**MuZero**
Env frames	200M	200M	200M	1B	90B	20B
Number of games > human	**54**$_{(53,55)}$	52	49	**57**$_{(57,57)}$	**57**	51
Capped mean	**98**$_{(97,98)}$	92	89	**100**$_{(100,100)}$	**100**	87
Mean	**3305**$_{(3163,3446)}$	2523	2856	4087$_{(3723,4445)}$	4766	**4998**
Median	848$_{(829,895)}$	**1077**	1006	1185$_{(1085,1325)}$	1933	**2041**
25th percentile	**282**$_{(269,303)}$	269	153	**478**$_{(429,515)}$	387	276
5th percentile	**100**$_{(89,108)}$	15	28	**119**$_{(119,120)}$	116	0

compared to the state-of-the-art methods shown in Table 1, while being trained for only 1B frames. When comparing our results with other state-of-the-art agents such as MuZero (Schrittwieser et al., 2020) or Muesli (Hessel et al., 2021), we observe a similar pattern to that reported by Badia et al. (2020): they achieve very high scores in most games, as denoted by their high mean and median scores, but struggle to learn completely on a few of them (Agarwal et al., 2021). It is important to note that our agent achieves the human benchmark in all games, as demonstrated by its higher scores on the lower percentiles.

5.2. Ablations

We analyze the contribution of all the methods introduced in Section 4 through ablation experiments on the same subset of eight games. We compare methods based on the area under the score curve in order to capture end performance and data efficiency under a single metric[4]. The magnitude of this quantity varies across games, so we normalize it by its maximum value per game in order to obtain a score between 0 and 1, and report results on the eight ablation games in Figure 4. Appendix I includes full learning curves for all ablation experiments as well as ablations of other design choices (e.g. the number of times each sample is replayed on average, optimizer hyperparameters).

Results in Figure 4 demonstrate that all proposed methods play a role in the performance and stability of the agent. Disabling the trust region and using the target network for bootstrapping (A1) produces the most important performance drop among all ablations, likely due to the target network being slower at propagating information into the targets. Interestingly, we have observed that the trust region is beneficial even when using target networks for bootstrapping (c.f. Figure 23 in Appendix L), which suggests that the trust region may produce an additional stabilizing effect beyond what target networks alone can provide. Besides having a similar stabilizing effect, policy distillation (D) also speeds up convergence on some games and has less tendency to converge to local optima on some others. The Soft Watkins' Q(λ) loss (A2) boosts data efficiency especially in games with sparse rewards and requiring long-term credit assignment, and we have empirically verified that it uses longer traces than other losses (c.f. Figure 5 in Appendix C). Cross-mixture training (B2) and the

[4]We first compute scores at 10,000 equally spaced points in [0, 1B] frames by applying piecewise linear interpolation to the scores logged during training. After averaging the scores at each of these points over all random seeds seeds, the trapezoidal rule is used to compute the area under the averaged curve.

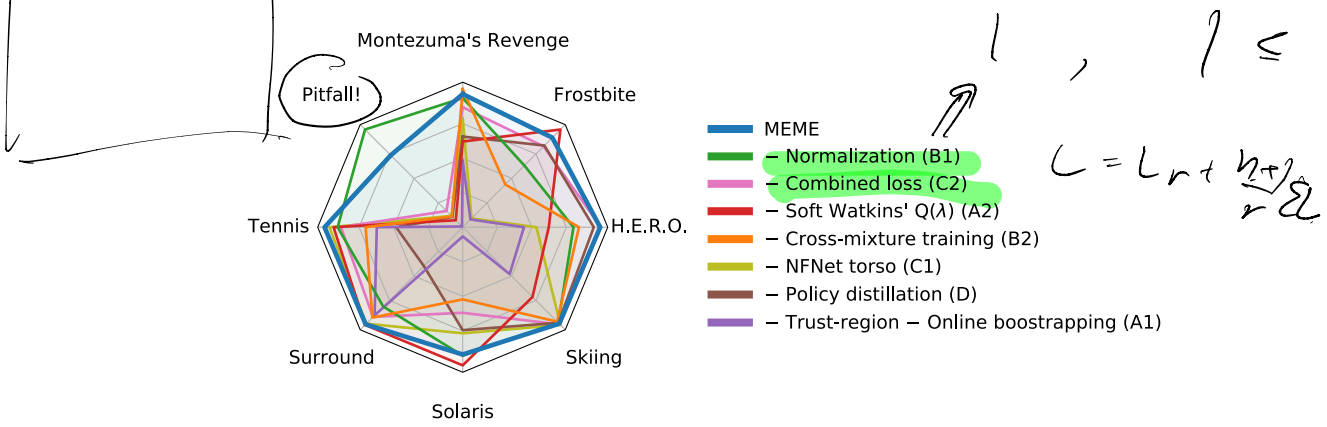


Figure 4 | Results of ablating individual components. For each experiment, we first average the score over three seeds up to 1B frames and then compute the area under the score curve as it captures not only final performance but also the amount of interaction required to achieve it. Since absolute values vary greatly across games, we report relative quantities by dividing by the maximum value obtained in each game.

combined loss (C2) tend to provide efficiency gains across most games. Finally, while we observe overall gains in performance from using normalization of TD errors (B1), the effect is less pronounced than that of other improvements. We hypothesize that the normalization has a high degree of overlap with other regularizing enhancements, such as including the trust region.

6. Discussion

We present a novel agent that outperforms the human-level baseline in a sample-efficient manner on all 57 Atari games. To achieve this, the agent employs a set of improvements that address the issues apparent in the previous state-of-the art methods, both in sample-efficiency and stability. These improvements include (1) TD-error normalisation scheme for the loss and priorities, (2) online network bootstrapping with approximate trust region, (3) improved, deeper and normalizer-free architecture, and (4) policy distillation from values. Our agent outperforms the human baseline across all 57 Atari games in 390M frames, using two orders of magnitude fewer interactions with the environment than Agent57, which leads to a 63× speed-up on average. We ran ablation experiments to evaluate the contribution of each improvement. Introducing online network bootstrapping with a trust-region has the most impact on the performance overall, while certain games require multiple different improvements to maintain stability and performance.

Although our agent achieves above average-human performance on all 57 Atari games within 390M frames with the same set of hyperparameters, the agent is separately trained on each game. An interesting research direction to pursue would be to devise a training scheme such that the agent with the same set of weights can achieve similar performance and sample-efficiency as the proposed agent on all games. Furthermore, the improvements that we propose do not necessarily only apply to Agent57, and further study could analyze the impact of these components in other state-of-the-art agents. We also expect that the generality of the agent could be expanded further. While this work focuses on Atari due to its wide range of tasks and reward structures, future research is required to analyze the ability of the agent to tackle other important challenges, such as more complex observation spaces (e.g. 3D navigation, multi-modal inputs), complex action spaces, or longer-term credit assignment. All such improvements would lead MEME towards a path of achieving greater generality.

References

R. Agarwal, M. Schwarzer, P. S. Castro, A. C. Courville, and M. Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *NeurIPS*, 2021.

M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. Pieter Abbeel, and W. Zaremba. Hindsight experience replay. *NeurIPS*, 2017.

P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 2002.

A. P. Badia, P. Sprechmann, A. Vitvitskyi, D. Guo, B. Piot, S. Kapturowski, O. Tieleman, M. Arjovsky, A. Pritzel, A. Bolt, et al. Never give up: Learning directed exploration strategies. In *ICLR*, 2019.

A. P. Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, Z. D. Guo, and C. Blundell. Agent57: Outperforming the atari human benchmark. In *ICML*, 2020.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *JAIR*, 2013.

A. Brock, S. De, S. L. Smith, and K. Simonyan. High-performance large-scale image recognition without normalization. In *ICML*, 2021.

Y. Burda, H. Edwards, A. Storkey, and O. Klimov. Exploration by random network distillation. In *ICLR*, 2019.

G. Dulac-Arnold, D. Mankowitz, and T. Hester. Challenges of real-world reinforcement learning. In *ICML*, 2019.

A. Ecoffet, J. Huizinga, J. Lehman, K. O. Stanley, and J. Clune. First return, then explore. *Nature*, 2021.

D. Ernst, P. Geurts, and L. Wehenkel. Tree-based batch mode reinforcement learning. *JMLR*, 2005.

L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *ICML*, 2018.

A. Garivier and E. Moulines. On upper-confidence bound policies for non-stationary bandit problems. *arXiv preprint arXiv:0805.3415*, 2008.

A. Garivier and E. Moulines. On upper-confidence bound policies for switching bandit problems. In *ALT*, 2011.

S. Hansen, W. Dabney, A. Barreto, D. Warde-Farley, T. Van de Wiele, and V. Mnih. Fast task inference with variational intrinsic successor features. In *ICLR*, 2019.

M. Hessel, I. Danihelka, F. Viola, A. Guez, S. Schmitt, L. Sifre, T. Weber, D. Silver, and H. Van Hasselt. Muesli: Combining improvements in policy optimization. In *ICML*, 2021.

D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. Van Hasselt, and D. Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger. Deep networks with stochastic depth. In *ECCV*, 2016.

Ł. Kaiser, M. Babaeizadeh, P. Miłos, B. Osiński, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, et al. Model based reinforcement learning for atari. In *ICLR*, 2019.

S. Kakade and J. Langford. Approximately optimal approximate reinforcement learning. In *ICML*, 2002.

S. Kapturowski, G. Ostrovski, J. Quan, R. Munos, and W. Dabney. Recurrent experience replay in distributed reinforcement learning. In *ICLR*, 2018.

K. P. Kielak. Do recent advancements in model-based deep reinforcement learning really improve data efficiency?, 2020. URL https://openreview.net/forum?id=Bke9u1HFwB.

I. Kostrikov, D. Yarats, and R. Fergus. Image augmentation is all you need: Regularizing deep reinforcement learning from pixels. *arXiv preprint arXiv:2004.13649*, 2020.

L.-J. Lin. *Reinforcement learning for robots using neural networks*. Carnegie Mellon University, 1992.

H. Liu and P. Abbeel. Behavior from the void: Unsupervised active pre-training. In *NeurIPS*, 2021.

A. Long, A. Blair, and H. van Hoof. Fast and data efficient reinforcement learning from pixels via non-parametric value approximation. *arXiv preprint arXiv:2203.03078*, 2022.

M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *JAIR*, 2018.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

R. Munos, T. Stepleton, A. Harutyunyan, and M. Bellemare. Safe and efficient off-policy reinforcement learning. In *NeurIPS*, 2016.

I. Osband, C. Blundell, A. Pritzel, and B. Van Roy. Deep exploration via bootstrapped dqn. *NeurIPS*, 2016.

J. Peng and R. J. Williams. Incremental multi-step q-learning. In *ICML*. 1994.

A. Piche, V. Thomas, J. Marino, G. M. Marconi, C. J. Pal, and M. E. Khan. Beyond target networks: Improving deep -learning with functional regularization. *arXiv*, 2021.

M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., 1994.

S. Qiao, H. Wang, C. Liu, W. Shen, and A. Yuille. Micro-batch training with batch-channel normalization and weight standardization. *arXiv preprint arXiv:1903.10520*, 2019.

M. Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *ECML*, 2005.

M. Riedmiller, R. Hafner, T. Lampe, M. Neunert, J. Degrave, T. Wiele, V. Mnih, N. Heess, and J. T. Springenberg. Learning by playing solving sparse reward tasks from scratch. In *ICML*, 2018.

T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

T. Schaul, D. Horgan, K. Gregor, and D. Silver. Universal value function approximators. In *ICML*, 2015a.

T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015b.

T. Schaul, G. Ostrovski, I. Kemaev, and D. Borsa. Return-based scaling: Yet another normalisation trick for deep rl. *arXiv preprint arXiv:2105.05347*, 2021.

T. Schaul, A. Barreto, J. Quan, and G. Ostrovski. The phenomenon of policy churn. *arXiv preprint arXiv:2206.00730*, 2022.

S. Schmitt, M. Hessel, and K. Simonyan. Off-policy actor-critic with shared experience replay. In *ICML*, 2020.

J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 2020.

J. Schrittwieser, T. Hubert, A. Mandhane, M. Barekatain, I. Antonoglou, and D. Silver. Online and offline reinforcement learning by planning with a learned model. In *NeurIPS*, 2021.

J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *ICML*, 2015.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

M. Schwarzer, A. Anand, R. Goel, R. D. Hjelm, A. Courville, and P. Bachman. Data-efficient reinforcement learning with self-predictive representations. *arXiv preprint arXiv:2007.05929*, 2020.

D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 2017.

D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 2018.

H. F. Song, A. Abdolmaleki, J. T. Springenberg, A. Clark, H. Soyer, J. W. Rae, S. Noury, A. Ahuja, S. Liu, D. Tirumala, et al. V-mpo: On-policy maximum a posteriori policy optimization for discrete and continuous control. *arXiv preprint arXiv:1909.12238*, 2019.

A. Srinivas, M. Laskin, and P. Abbeel. Curl: Contrastive unsupervised representations for reinforcement learning. *arXiv preprint arXiv:2004.04136*, 2020.

F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *AAAI*, 2016.

H. P. van Hasselt, M. Hessel, and J. Aslanides. When to use parametric models in reinforcement learning? In *NeurIPS*, 2019.

C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 1992.

W. Ye, S. Liu, T. Kurutach, P. Abbeel, and Y. Gao. Mastering atari games with limited data. In *NeurIPS*, 2021.

A. Hyper-parameters

Table 2 | Agent Hyper-parameters.

Parameter	Value
Num Mixtures	16
Bandit β	1.0
Bandit ϵ	0.5
Bandit γ	0.999
Max Discount	0.9997
Min Discount	0.97
Replay Period	80
Burn-in	0
Trace Length	160
AP Embedding Size	32
RND Scale	0.5
RND Clip Threshold	5.0
IM Reward Scale β_{IM}	0.1
β_{std}	2.0
Max KL c_{KL}	0.5
Cross-Mixture η	0.5
π'_{dist} Softmax Temperature τ	0.25
Soft Watkins-Q(λ) Threshold κ	0.01
λ	0.95
Residual Drop Rate	0.25
Eval Parameter Discount η_{eval}	0.995
Priority Exponent	0.6
Importance Sampling Exponent	0.4
Max Priority Weight	0.9
Replay Ratio	6.0
Replay Capacity	2×10^5 trajectories
Value function rescaling	$\text{sgn}(x)\left(\sqrt{x^2+1}-1\right) + 0.001\,x$
Batch Size	64
Adam β_1	0.9
Adam β_2	0.999
Adam ϵ	10^{-8}
RL Adam Learning Rate	3×10^{-4}
AP Adam Learning Rate	6×10^{-4}
RND Adam Learning Rate	6×10^{-4}
RL Weight Decay	0.05
AP Weight Decay	0.05
RND Weight Decay	0.0

B. Network Architecture

B.1. Torso

We use a modified version of the NFNet architecture (Brock et al., 2021). We use a simplified stem, consisting of a single 7×7 convolution with stride 4. We also forgo bottleneck residual blocks in favor of 2 layers of 3×3 convolutions, followed by a Squeeze-Excite block.

In addition, we make some minor modifications to the downsampling blocks. Specifically, we apply an activation prior to the average pooling and multiply the output by the stride in order to maintain the variance of the activations. This is then followed by a 3×3 convolution.

All convolutions use a Scaled Weight Standardization scheme (Qiao et al., 2019). The block section parameters are as follows:

Table 3 | Environment Hyper-parameters.

Parameter	Value
Input Shape	210×160
Grayscaling	True
Action Repeat	4
Num Stacked Frames	1
Pooled Frames	2
Max Episode Length	108000 frames (30 minutes game time)
Life Loss Signal	Not used

- num blocks: (2, 3, 4, 4)
- num channels: (64, 128, 128, 64)
- strides: (1, 2, 2, 2)

B.2. Non-image Features

We also feed the following features into the network:

- previous action (encoded as a size-32 embedding)
- previous extrinsic reward
- previous intrinsic reward
- previous RND component of intrinsic reward
- previous Episodic component of intrinsic reward
- previous Action Prediction embedding

These are fed into a single linear layer of size 256 and activation and then concatenated with the output of the torso and input into the recurrent core.

B.3. Recurrent Core

The recurrent core is composed of single LSTM with hidden size 1024. The output is then concatenated together with the core input and fed into the Action-Value and Policy heads.

B.4. Action-Value and Policy Heads

We utilize separate dueling heads for the intrinsic and extrinsic components of the value function, and a simple MLP for the policy. All heads use two hidden layers of size 1024 and output size num_actions $\times$ num_mixtures.

C. Target Computation and Trace Coefficients

As motivated in Section 4 A2, we introduce Soft Watkins Q(λ) as a trade-off between aggressive trace cutting used within Retrace and Watkins Q(λ), and the lack of off-policy correction in Peng's Q(λ). We investigate this hypothesis by observing the average trace coefficients for each of the methods in Figure 5. The lower values of the trace coefficient λ lead to more aggressive trace cutting as soon as the data is off-policy. Conversely, the λ value of 1 signifies no trace cutting whatsoever. As expected, Retrace's trace coefficient is the lowest, followed by Watkins Q(λ). The proposed Soft Watkins Q(λ) has a parameter κ that allows us to control the permissiveness of the trace-cutting which in turn

affects the final trace coefficient. We find that with the optimal value of $\kappa = 0.01$ we achieve the best performance, and this moves the average trace coefficient value towards being more permissive.

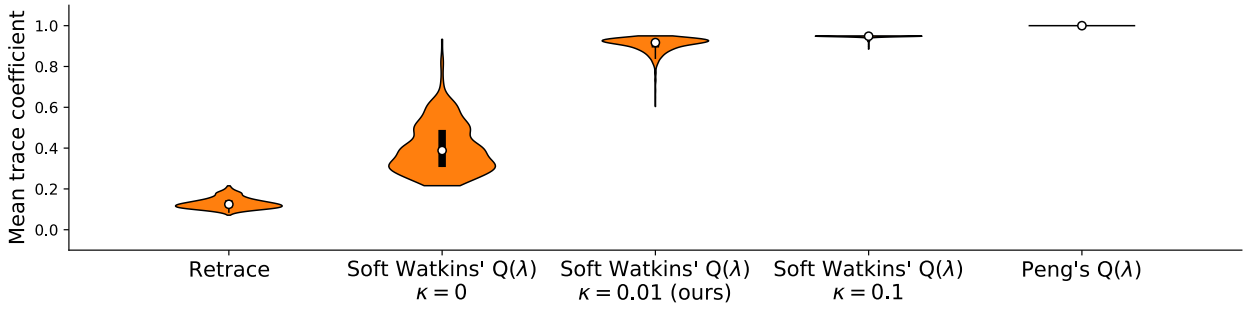


Figure 5 | Average trace coefficients for each method on the set of ablation games. For each method, we average the trace length for transitions generated between 200M and 250M frames, as we observe that their values tend to stabilize after an initial transient period. Each violin plot is thus generated from `num_seeds` × `num_games` × `num_mixtures` data points.

D. Distributed setting

All experiments are run using a distributed setting. The experiment consists of the actors, learner, bandit and evaluators, as well as a replay buffer.

The actors and evaluators are the two types of workers that draw samples from the environment. Since actors collect experience with non-greedy policies, we follow the common practice in this type of agent and report scores from separate evaluator processes that continually execute the greedy policy and whose experience is not added to the replay buffer (Kapturowski et al., 2018). Therefore, only the actor workers write to the replay buffer, while the evaluation workers are used purely for reporting the performance. The evaluation scheme differs from R2D2 (Kapturowski et al., 2018) in that a separate set of eval parameters are maintained, which are computed as an EMA of the online network with $\eta_{\text{eval}} = 0.995$; and these eval parameters are continually updated throughout each episode. We observed that the use of these eval parameters provided a consistent performance boost across almost all environments, but we continue to use the online network for the actors in order to obtain more varied experience.

In the replay buffer, we store fixed-length sequences of $(r_t^e, r_t^{\text{NGU}}, x_t, a_t)$ tuples. These sequences never cross episode boundaries. For Atari, we apply the standard DQN pre-processing, as used in R2D2. The replay buffer is split into 8 shards, to improve robustness due to computational constraints, with each shard maintaining an independent prioritisation of the entries. We use prioritized experience replay with the same prioritization scheme proposed by Kapturowski et al. (2018) which used a weighted mixture of max and mean TD-errors over the sequence. Each of the actor workers writes to a specific shard which is consistent throughout training.

Given a single batch of trajectories we unroll both online and target networks on the same sequence of states to generate value estimates. These estimates are used to execute the learner update step, which updates the model weights used by the actors, and the exponential moving average (EMA) of the weights used by the evaluator models, as this yields best performance which we report.

Acting in the environment is accelerated by sending observations from actors and evaluators to a shared server that runs batched inference. The remote inference server allows multiple clients such as actor and evaluator workers to connect to it, and executes their inputs as a batch on the

corresponding inference models. The actor and evaluator inference model parameters are queried periodically from the learner. Also, the recurrent state is persisted on the inference server so that the actor does need to communicate it. However, the episodic memory lookup required to compute the intrinsic reward is performed locally on actors to reduce the communication overhead.

At the beginning of each episode, parameters β and γ are queried from the bandit worker, i.e. meta-controller. The parameters are selected from a set of coefficients $\{(\beta_j, \gamma_j)\}_{j=0}^{N-1}$ with $N = 16$, which correspond to the N-heads of the network. The actors query optimal (β, γ) tuples, while the evaluators query the tuple corresponding to the greedy action. After each actor episode, the bandit stats are updated based on the episode rewards by updating the distribution over actions, according to Discounted UCB (Garivier and Moulines, 2011).

The following subsections describe how actors, evaluators, and learner are run in each stage.

Learner

- Sample a sequence of extrinsic rewards r_t^e, intrinsic rewards r_t^{NGU}, observations x_t and actions a_t, from the replay buffer.
- Use Q-network to learn from $(r_t^e, r_t^{\text{NGU}}, x, a)$ with our modified version of Watkins' Q(λ) (Watkins and Dayan, 1992) using the same procedure as in R2D2.
- Compute the actor model weights and EMA for the evaluator model weights.
- Use the sampled sequences to train the action prediction network in NGU.
- Use the sampled sequences to train the predictor of RND.

Actor

- Query optimal bandit action (β, γ).
- Obtain x_t from the environment.
- Obtain r_t^{NGU} and a_t from the inference model.
- Insert x_t, a_t, r_t^{NGU} and r_t^e in the replay buffer.
- Step on the environment with a_t.

Evaluator

- Query greedy bandit action (β, γ).
- Obtain x_t from the environment.
- Obtain r_t^{NGU} and a_t from the inference model.
- Step on the environment with a_t.

Bandit

- Periodically checkpoints bandit action values.
- Queried for optimal action by actors.
- Queried for greedy action by evaluators.
- Updates the stats when actors pass the episode rewards for a certain action.

E. Bandit Implementation

We utilize a centralized bandit worker as a meta-controller to select between a family of policies generated by tuples of intrinsic reward weight and discount factor (β_i, γ_i), parameterized as:

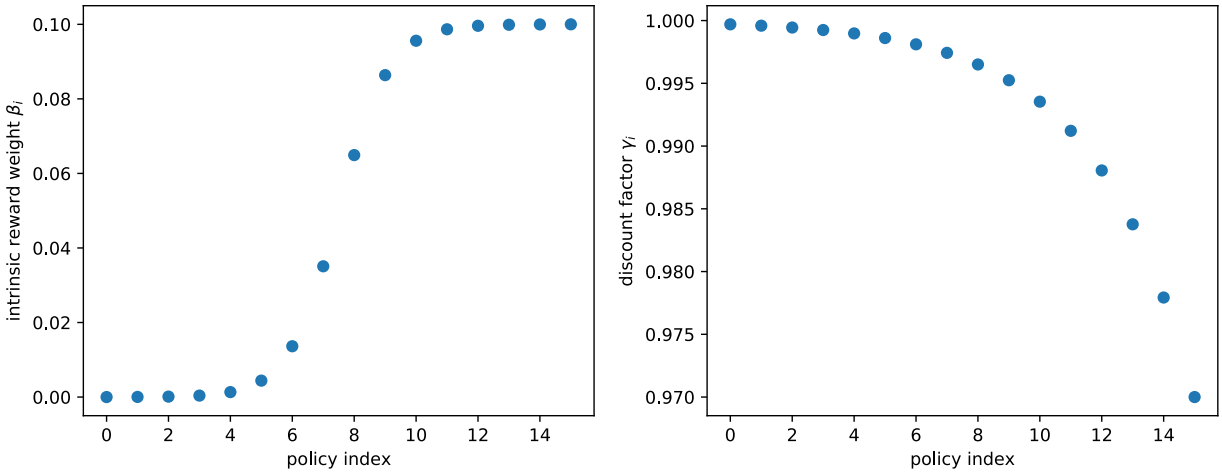


Figure 6 | β and γ for each of the 16 policies.

$$\beta_i = \begin{cases} 0 & \text{if } i = 0 \\ \beta_{\text{IM}} & \text{if } i = N-1 \\ \beta_{\text{IM}}\sigma(8\frac{2i-(N-2)}{N-2}) & \text{otherwise} \end{cases} \tag{9}$$

$$\gamma_i = 1 - \exp\left(\frac{N-1-i}{N-1}\log(1-\gamma_{\max}) + \frac{i}{N-1}\log(1-\gamma_{\max})\right) \tag{10}$$

where N is the number of policies, i is the policy index, σ is the sigmoid function, β_{IM} is the maximum intrinsic reward weight, and $\gamma_{\max}$ and γ_{int} are the maximum and minimum discount factors, respectively.

At the beginning of each episode an actor will sample a policy index with which to act for the duration of the episode. At the end of which, the actor will update the bandit with the obtained extrinsic return for that policy. We use a discounted variant of UCB-Tuned bandit algorithm (Garivier and Moulines (2008) and Auer et al. (2002)). In practice the bandit hyper-parameters did not seem to be very important. We hypothesize that the use of cross-mixture training may reduce the sensitivity of the agent to these parameters, though we have not explored this relationship thoroughly.

F. Compute Resources

For the experiments we used the TPUv4, with the $2 \times 2 \times 1$ topology used for the learner. Acting is accelerated by sending observations from actors to a shared server that runs batched inference using a $1 \times 1 \times 1$ TPUv4, which is used for inference within the actor and evaluation workers.

On average, the learner performs 3.8 updates per second. The rate at which environment frames are written to the replay buffer by the actors is approximately 12970 frames per second.

Each experiment consists of 64 actors with 2 threads, each of them acting with their own independent instance of the environment. The collected experience is stored in the replay buffer split in 8 shards, each with independent prioritisation. This accumulated experience is used by a single learner worker, while the performance is evaluated on 5 evaluator workers.

G. Intrinsic rewards

G.1. Random Network Distillation

The Random Network Distillation (RND) intrinsic reward (Burda et al., 2019) is computed by introducing a random, untrained convolutional network $g : \mathcal{X} \to \mathbb{R}^d$, and training a network $\hat{g} : \mathcal{X} \to \mathbb{R}^d$ to predict the outputs of g on all the observations that are seen during training by minimizing the prediction error $\text{err}_{\text{RND}}(x_t) = ||\hat{g}(x_t; \theta) - g(x_t)||^2$ with respect to θ. The intuition is that the prediction error will be large on states that have been visited less frequently by the agent. The dimensionality of the random embedding, d, is a hyperparameter of the algorithm.

The RND intrinsic reward is obtained by normalising the prediction error. In this work, we use a slightly different normalization from that reported in Burda et al. (2019). The RND reward at time t is given by

$$r_t^{\text{RND}} = \frac{\text{err}_{\text{RND}}(x_t)}{\sigma_e} \tag{11}$$

where σ_e is the running standard deviation of $\text{err}_{\text{RND}}(x_t)$.

G.2. Never Give Up

The NGU intrinsic reward modulates an episodic intrinsic reward, r_t^{episodic}, with a life long signal α_t:

$$r_t^{\text{NGU}} = r_t^{\text{episodic}} \cdot \min\left\{\max\left\{\alpha_t, 1\right\}, L\right\}, \tag{12}$$

where L is a fixed maximum reward scaling. The life-long novelty signal is computed using RND with the normalisation:

$$\alpha_t = \frac{\text{err}_{\text{RND}}(x_t) - \mu_e}{\sigma_e} \tag{13}$$

where $\text{err}_{\text{RND}}(x_t)$ is the prediction error described in Appendix G.1, and μ_e and σ_e are its running mean and standard deviation, respectively. The episodic intrinsic reward at time t is computed according to formula:

$$r_t^{\text{episodic}} = \frac{1}{\sqrt{\sum_{f(x_i) \in N_k} K(f(x_t), f(x_i))} + c} \tag{14}$$

where N_k is the set containing the k-nearest neighbors of $f(x_t)$ in M, c is a constant and $K : \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}^+$ is a kernel function satisfying $K(x, x) = 1$ (which can be thought of as approximating pseudo-counts Badia et al. (2019)). Algorithm 1 shows a detailed description of how the episodic intrinsic reward is computed. Below we describe the different components used in Algorithm 1:

- M: episodic memory containing at time t the previous embeddings $\{f(x_0), f(x_1), \ldots, f(x_{t-1})\}$. This memory starts empty at each episode
- k: number of nearest neighbours
- $N_k = \{f(x_i)\}_{i=1}^k$: set of k-nearest neighbours of $f(x_t)$ in the memory M; we call $N_k[i] = f(x_i) \in N_k$ for ease of notation
- K: kernel defined as $K(x, y) = \frac{\epsilon}{\frac{d^2(x,y)}{d_m^2} + \epsilon}$ where ϵ is a small constant, d is the Euclidean distance and d_m^2 is a running average of the squared Euclidean distance of the k-nearest neighbors
- c: pseudo-counts constant
- ξ: cluster distance
- s_m: maximum similarity
- $f(x)$: action prediction network output for observation x as in Badia et al. (2020).

Algorithm 1: Computation of the episodic intrinsic reward at time t: r_t^{episodic}.

Input : M; k; $f(x_t)$; c; ϵ; ξ; x_m; d_m^2
Output : r_t^{episodic}

Compute the k-nearest neighbours of $f(x_t)$ in M and store them in a list N_k
Create a list of floats d_k of size k
/* The list d_k will contain the distances between the embedding $f(x_t)$ and its neighbours N_k. */
for $i \in \{1, \ldots, k\}$ **do**
  $d_k[i] \leftarrow d^2(f(x_t), N_k[i])$
end
Update the moving average d_m^2 with the list of distances d_k
/* Normalize the distances d_k with the updated moving average d_m^2. */
$d_n \leftarrow \frac{d_k}{d_m^2}$
/* Cluster the normalized distances d_n i.e. they become 0 if too small and 0_k is a list of k zeros. */
$d_n \leftarrow \max(d_n - \xi, 0_k)$
/* Compute the Kernel values between the embedding $f(x_t)$ and its neighbours N_k. */
$K_v \leftarrow \frac{\epsilon}{d_n + \epsilon}$
/* Compute the similarity between the embedding $f(s_t)$ and its neighbours N_k. */
$s \leftarrow \sqrt{\sum_{i=1}^{k} K_v[i]} + c$
/* Compute the episodic intrinsic reward at time t: r_t^i. */
if $x > x_m$ **then**
  $r_t^{\text{episodic}} \leftarrow 0$
else
  $r_t^{\text{episodic}} \leftarrow 1/s$

H. Other Things we Tried

H.1. Functional Regularization in place of Trust Region

Concurrently with the development of our trust region method we experimented with using an explicit L2 regularization term in the loss, acting on the difference between the online and target networks, similar to (Piche et al., 2021). Prior to implementation of our normalization scheme we found that this method stabilized learning early on in training but was prone to eventually becoming unstable if run for long enough. With normalization this instability did not occur, but sample efficiency was worse compared to the trust region in most instances we observed.

H.2. Approximate Thompson Sampling

We considered using an approximate Thompson Sampling scheme scheme similar to Bootstrapped DQN (Osband et al., 2016) whereby the stochastic depth mask was fixed for some period of time at inference (such as once per episode, or every 100 timesteps). We observed some marginal benefit in certain games, but in our view this difference was not enough to justify the added complexity of implementation. We hypothesize that the added exploration this provides is not significant when a strong intrinsic reward is already present in the learning algorithm, but it may have a larger effect if this is not the case.

H.3. Mixture of Online and Replay Data

We considered using a mixture of online and replay data as was done in the Muesli agent (Hessel et al., 2021). This was beneficial for overall stability, but it also degraded performance in the harder exploration games such as Pitfall. We were not able to find an easy remedy for this so we did not investigate further in this direction.

H.4. Estimating $Q_e(\theta) + \beta Q_i(\theta)$

Agent57 uses two neural networks with completely independent weights to estimate Q_e and Q_i. As mentioned in the work, this provides the network with more robustness to the different scales and variance that r_e and r_i have for many tasks.

MEME changes the separation of networks, whereby the Q_e and Q_i are still estimated separately, but they share a common torso and recurrent core. However, since many of the components we introduce are geared toward improving stability, even this separation may no longer be necessary. To analyze this we run an experiment where the agent network has a single head that estimates $Q_e(\theta) + \beta Q_i(\theta)$. Note that in this case we still estimate N sets of Q-values. Indeed, as results of Figure 7 show, we observe similar results as our proposed method. This indicates that the inductive bias that was introduced in maintaining separate heads for intrinsic and extrinsic Q-values is no longer important.

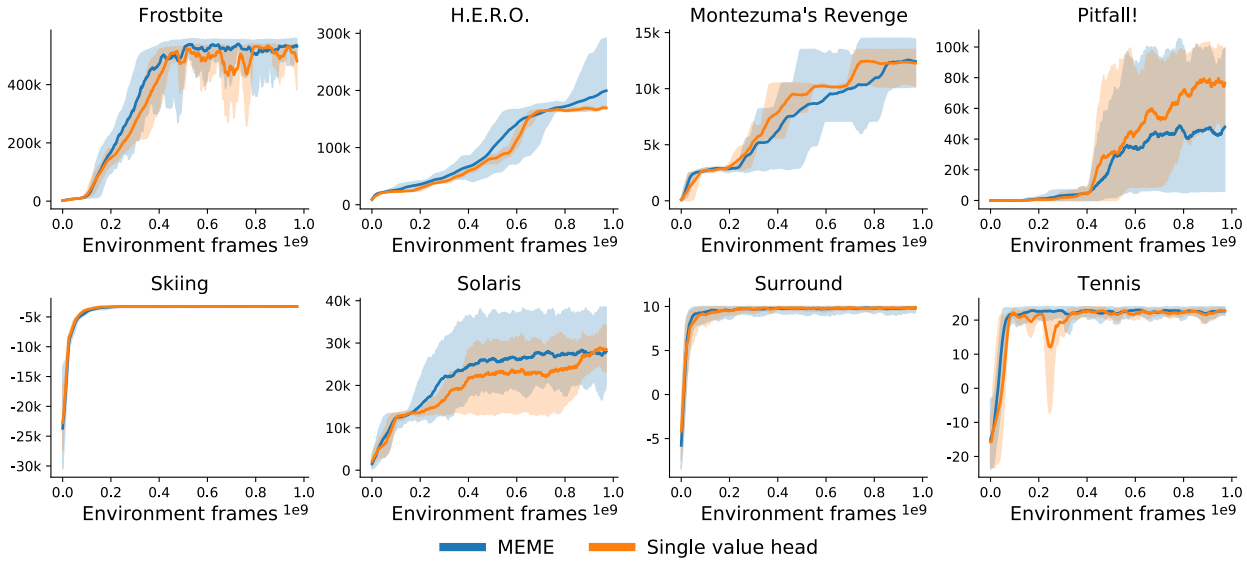


Figure 7 | Results of estimating the total loss.

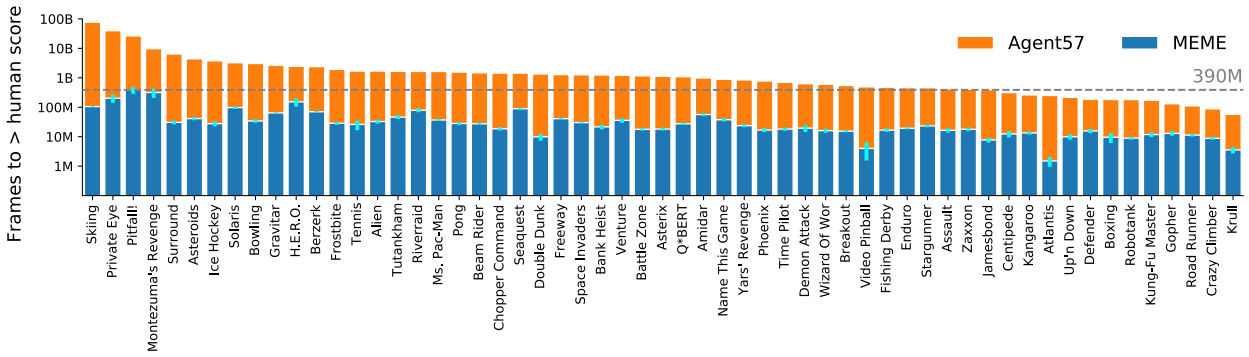


Figure 8 | Number of environment frames required by agents to outperform the human baseline on each game (in log-scale). Lower is better. Error bars represent the standard deviation over seeds for each game. On average, MEME achieves above human scores using 63× fewer environment interactions than Agent57. The smallest improvement is 9× (`road_runner`), the maximum is 721× (`skiing`), and the median across the suite is 35×.

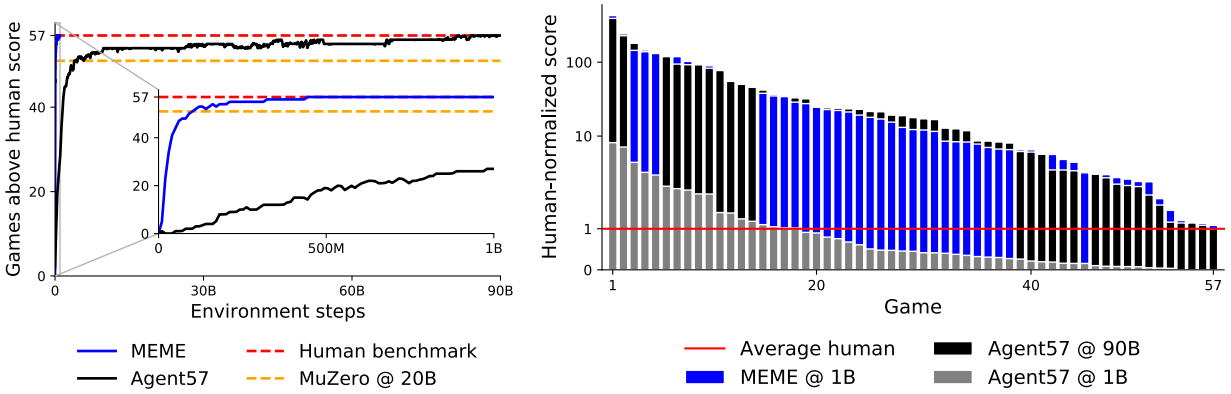


Figure 9 | Comparison with Agent57. *Left:* number of games with scores above the human benchmark. *Right:* human-normalized scores per game at different interaction budgets, sorted from highest to lowest. Our agent outperforms the human benchmark in 390M frames, two orders of magnitude faster than Agent57, and achieves similar end scores while reducing the training budget from 90B to 1B frames.

I. Additional results

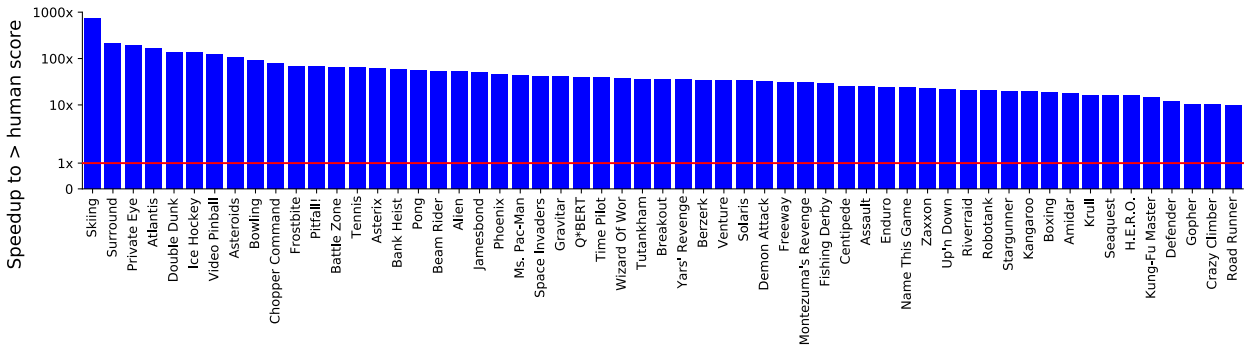


Figure 10 | Speedup in reaching above human performance for the first time, computed as the ratio between the black and blue bars in Figure 1.

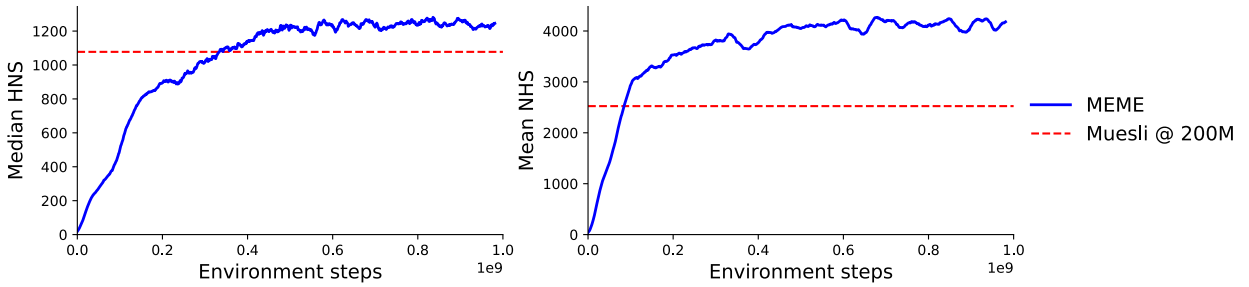


Figure 11 | Median and mean scores over the course of training.

J. Sticky actions results

This section reproduces the main results of the paper, but enabling *sticky actions* (Machado et al., 2018) during both training and evaluation. Since our agent does not exploit the determinism in the original Atari benchmark, it is still able to outperform the human baseline with *sticky actions* enabled. We observe a slight decrease in mean scores, which we attribute to label corruption in the action prediction loss used to learn the controllable representations used in the episodic reward computation: due to the implementation of sticky actions proposed by Machado et al. (2018) the agent actions are ignored in a fraction of the timesteps. This phenomenon is aggravated by the frame stacking used in the standard Atari pre-processing, as the action being executed in the environment can vary within each stack of frames. We hypothesize that the gap between the two versions of the environment would be much smaller with a different implementation of *sticky actions* that did not corrupt the action labels used by the representation learning module.

All reported results are the average over five different random seeds.

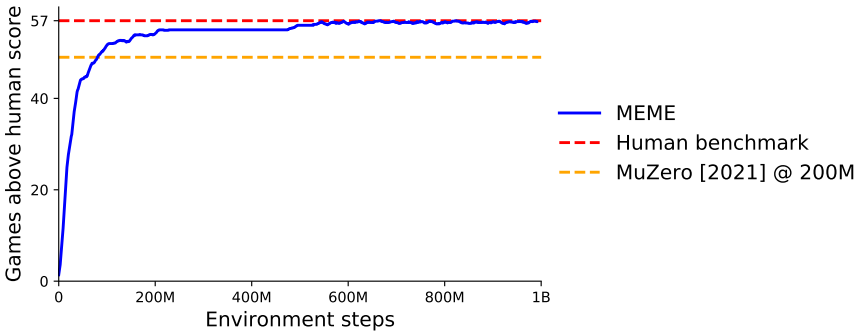


Figure 12 | Number of games with scores above the human benchmark when training and evaluating with *sticky actions* (Machado et al., 2018).

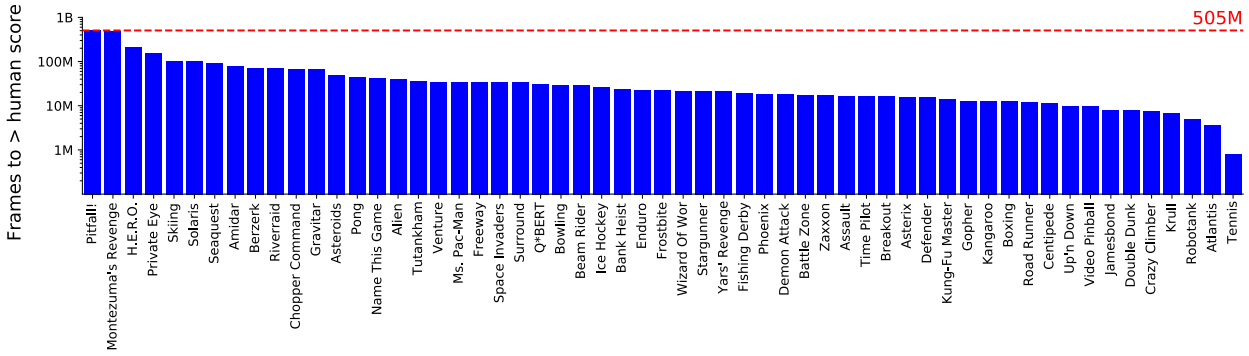


Figure 13 | Number of environment frames required by agents to outperform the human baseline on each game when training and evaluating with *sticky actions* (Machado et al., 2018). The human baseline is outperformed after 505M frames.

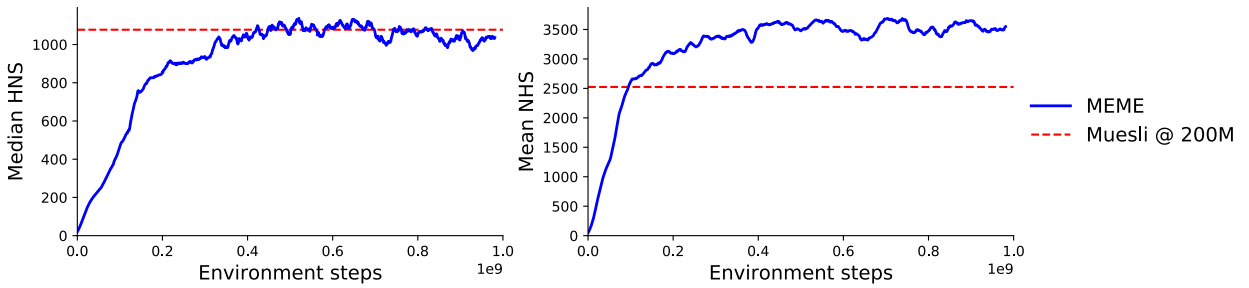


Figure 14 | Median and mean scores over the course of training when training and evaluating with *sticky actions* (Machado et al., 2018).

Table 4 | Number of games above human, capped mean, mean and median human normalized scores for the 57 Atari games when training and evaluating with *sticky actions* (Machado et al., 2018). Metrics for previous methods are computed using the final score per game reported in their respective publications: MuZero (Schrittwieser et al., 2021), Muesli (Hessel et al., 2021).

	200M frames			> 200M frames
Statistic	**MEME**	**Muesli**	**MuZero**	**MEME**
Env frames	200M	200M	200M	1B
Number of games > human	**54**	52	49	**57**
Capped mean	**97.51**	92.52	89.78	**100.0**
Mean	**2967.52**	2523.99	2856.24	**3462.93**
Median	830.57	**1077.47**	1006.4	**1074.25**
25th percentile	**299.65**	269.25	153.1	**402.56**
5th percentile	**103.86**	15.91	28.76	**118.78**

K. Effect of Samples per Insert Ratio

Results of the ablation on the amount of replay that the learner performs per sequence of experience that the actors produce can be seen in Figure 15. We can see that, while a samples per insert ratio (SPI) of 10 still provides moderate boosts in data efficiency in games such as `hero`, `montezuma_revenge`, and `pitfall`, it is not as pronounced as the increase that is seen from SPI of 3 to 6. This implies that with an SPI of 10 we obtain a much worse return in terms of wall-clock time as we replay more frequently.

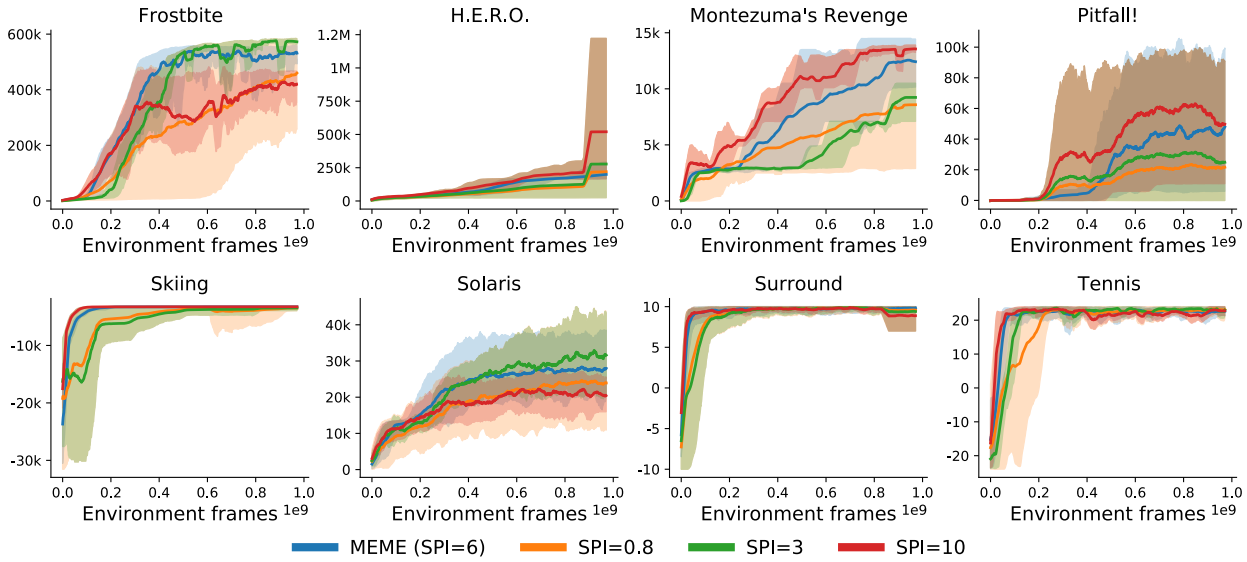


Figure 15 | Sweep over sample per insert ratios on the full ablation set.

L. Full ablations

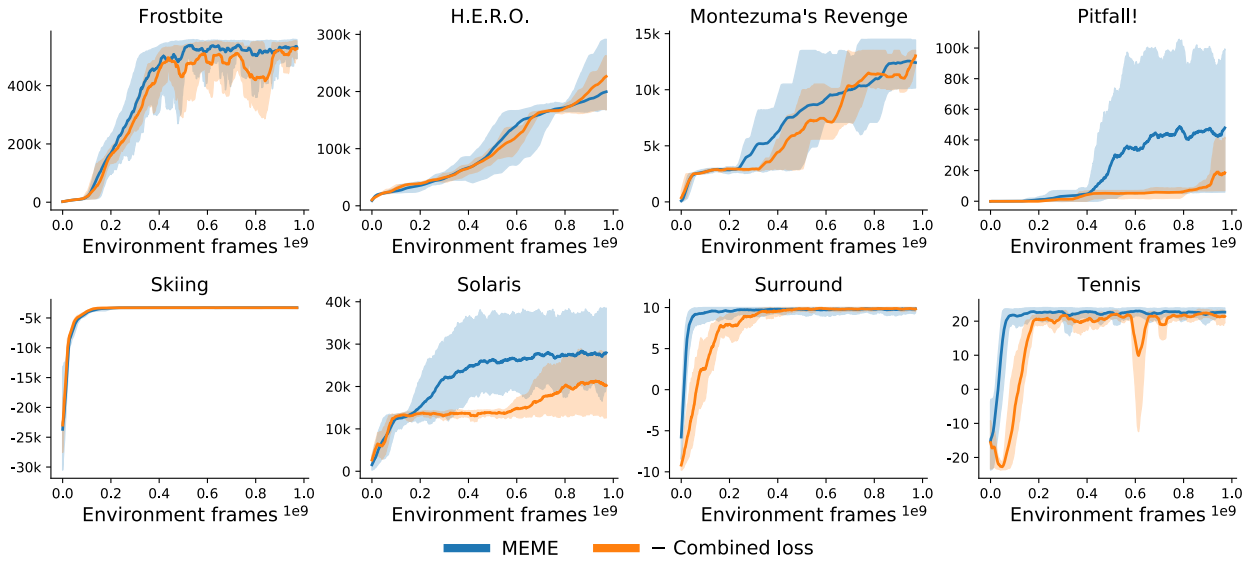


Figure 16 | Comparison between our combined loss and the separate losses used by Agent57 on the full ablation set.

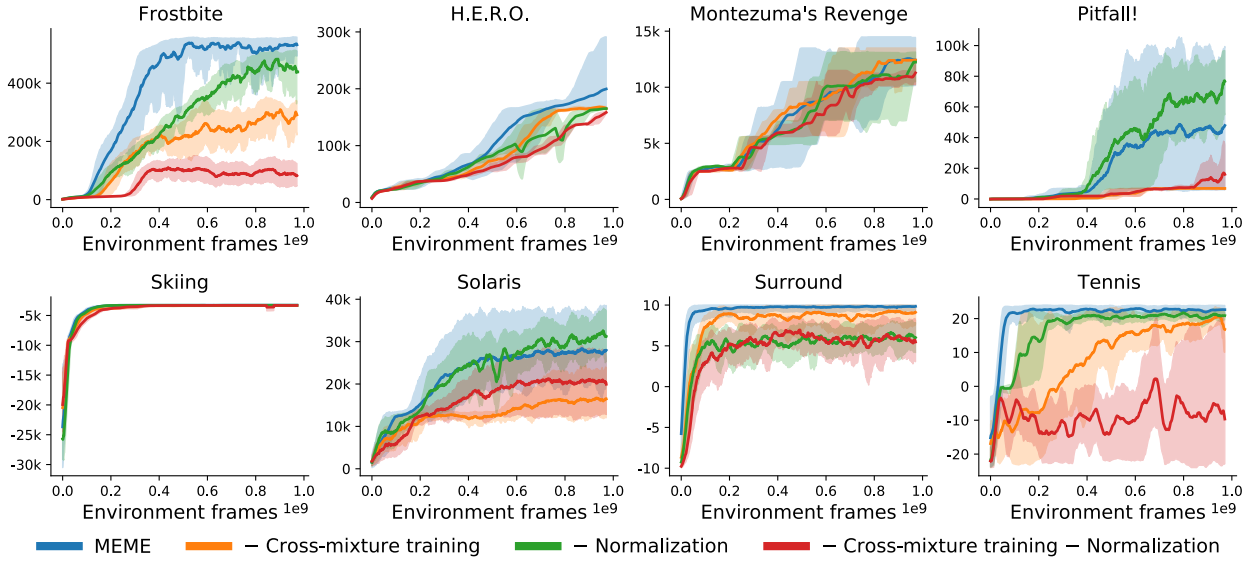


Figure 17 | Results without cross-mixture training and TD normalization on the full ablation set.

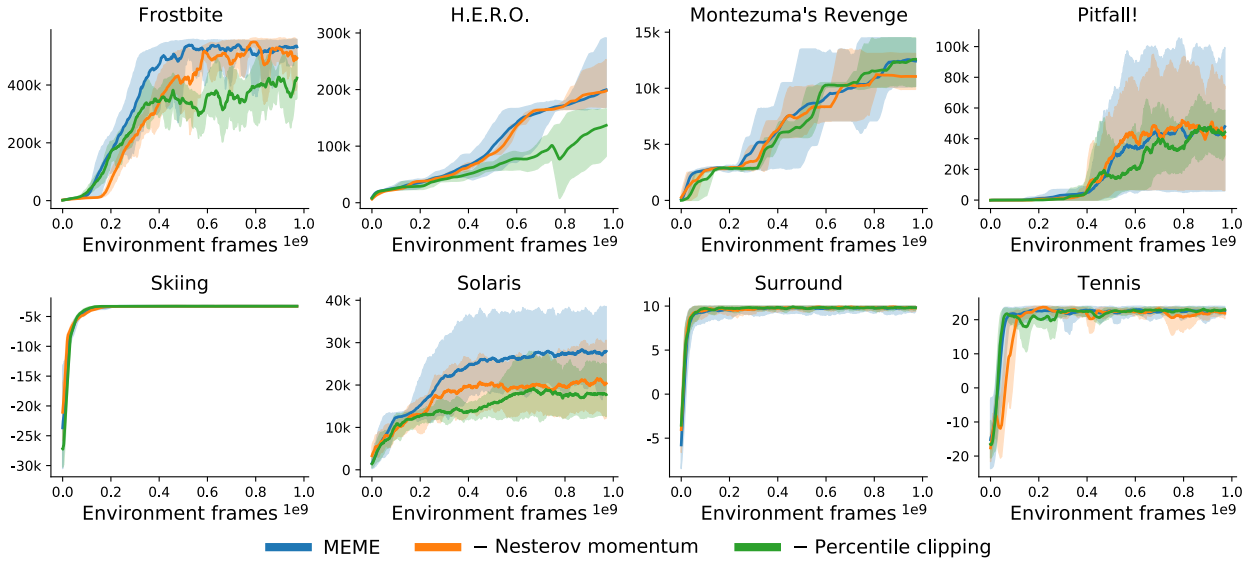


Figure 18 | Ablation experiments over different optimizer features.

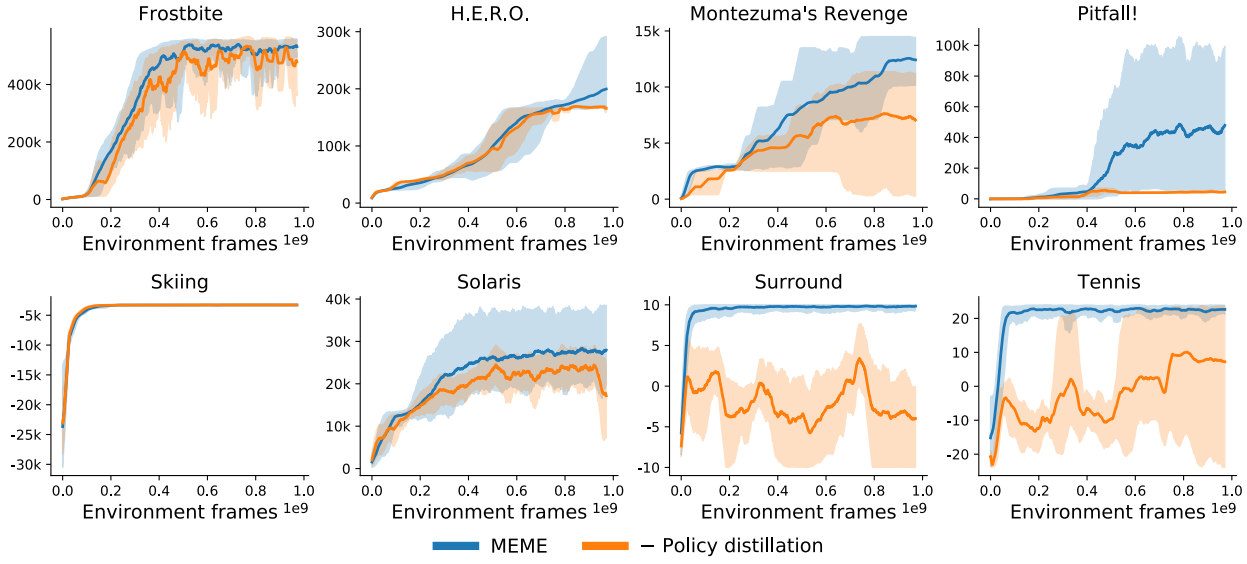


Figure 19 | Results without policy distillation on the full ablation set.

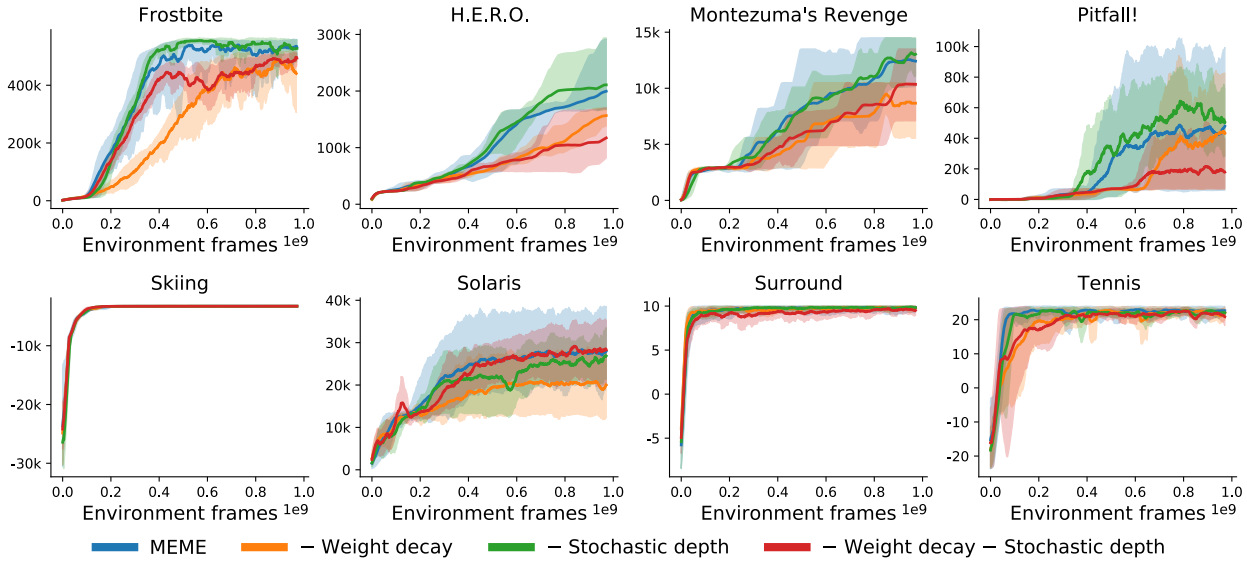


Figure 20 | Results for agents with different amounts of regularization on the full ablation set.

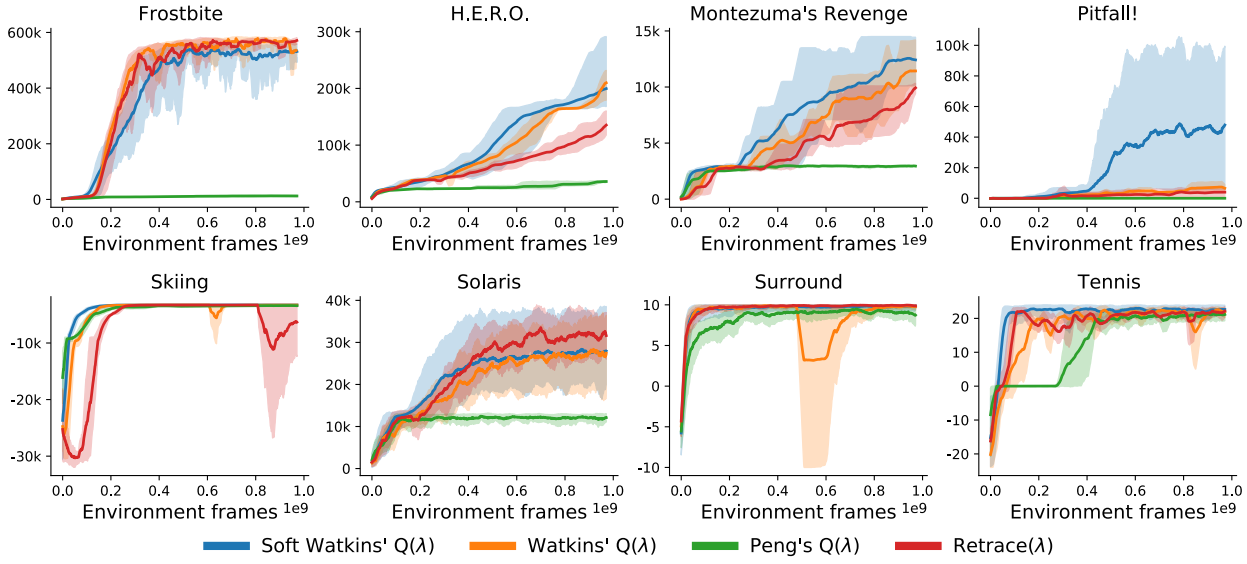


Figure 21 | Results with different learning targets on the full ablation set.

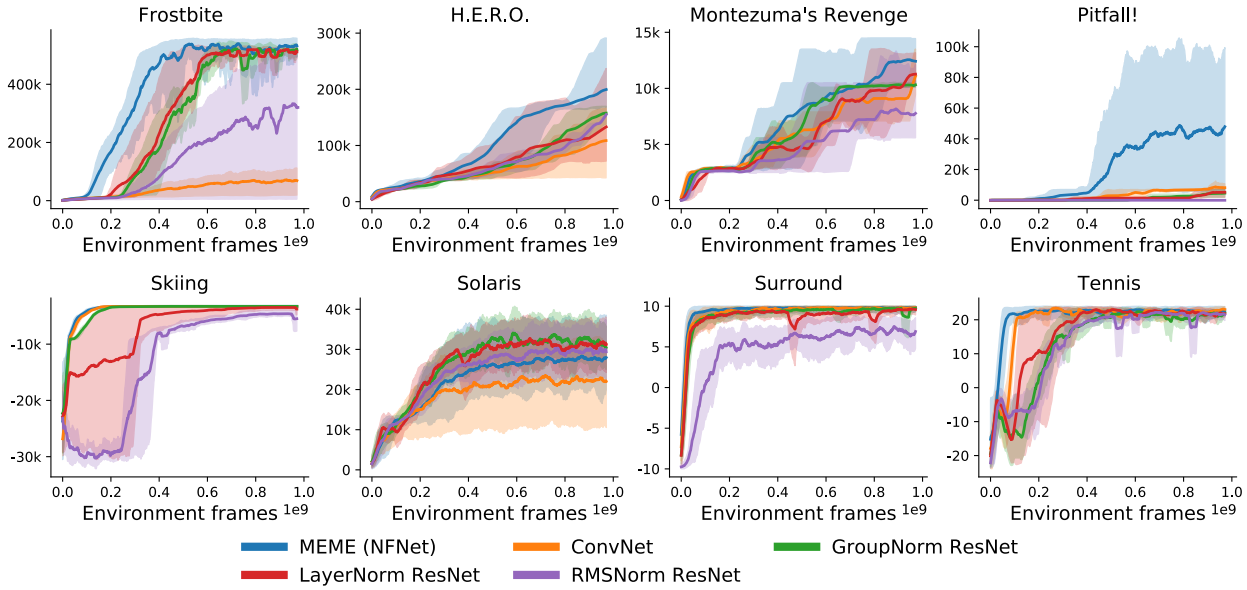


Figure 22 | Results for agents with different torsos on the full ablation set.

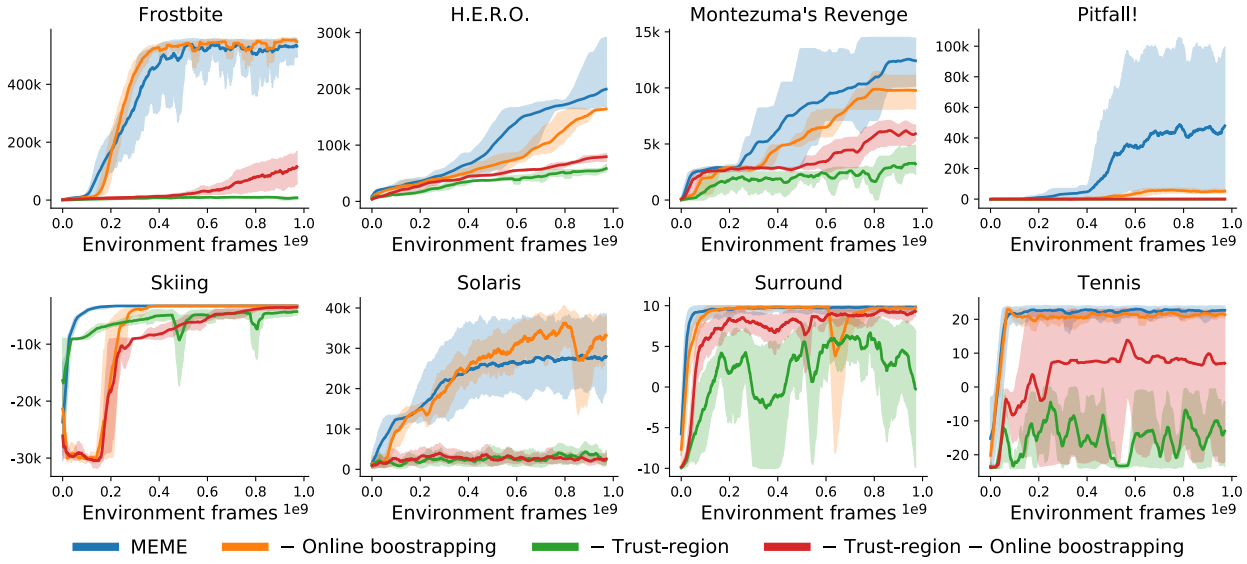


Figure 23 | Results for agents without online bootstrapping and trust-region on the full ablation set.

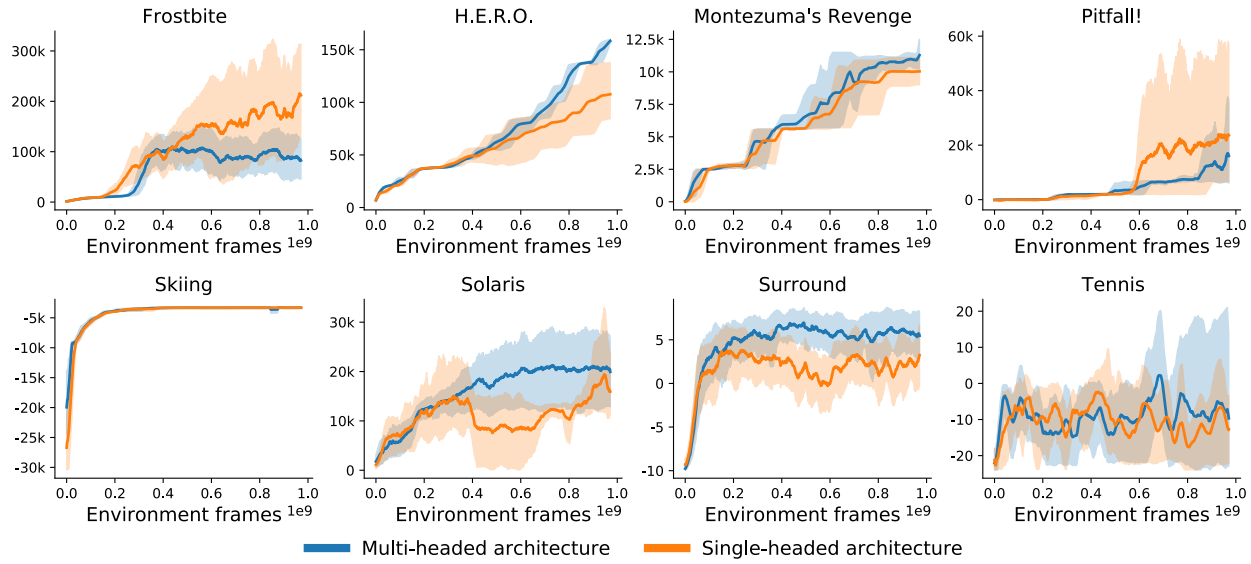


Figure 24 | Comparison of head architectures. The single-headed architecture is similar to the one in Agent57, where the network is conditioned on a one-hot encoding of the mixture id. All experiments are run without cross-mixture training and TD normalization for fairness. These results demonstrate that our multi-headed architecture, introduced to enable efficient computation of Q values for all mixtures in parallel, does not degrade performance.

M. Scores per game

Table 5 | Scores per game.

Game	Agent57 @ 1B	Agent57 @ 90B	MEME @ 200M	MEME @ 1B
Alien	3094.46 ± 1682.49	**252889.92 ± 31085.48**	41048.78 ± 8702.03	83683.43 ± 16688.58
Amidar	329.99 ± 124.14	**28671.19 ± 1046.52**	7363.47 ± 1033.44	14368.90 ± 2775.86
Assault	1183.25 ± 482.14	**49198.37 ± 9469.92**	33266.67 ± 6143.71	46635.86 ± 14846.53
Asterix	2777.67 ± 697.39	763476.87 ± 105395.98	**861780.67 ± 97588.31**	769803.92 ± 143061.91
Asteroids	2422.64 ± 412.96	105058.07 ± 45380.23	217586.98 ± 17304.43	**364492.07 ± 13982.31**
Atlantis	44844.67 ± 15838.85	1508119.97 ± 24913.10	1535634.17 ± 20700.45	**1669226.33 ± 3906.17**
Bank Heist	364.26 ± 156.87	17274.04 ± 12145.09	15563.35 ± 14565.74	**87792.55 ± 104611.67**
Battle Zone	19065.86 ± 999.82	**868540.57 ± 26523.78**	733206.67 ± 84295.20	776770.00 ± 19734.15
Beam Rider	2056.68 ± 131.88	**283845.01 ± 11289.64**	68534.71 ± 4443.34	51870.20 ± 2906.10
Berzerk	614.26 ± 84.32	31565.12 ± 34278.20	7003.12 ± 2592.61	**38838.35 ± 14783.99**
Bowling	34.00 ± 4.19	240.33 ± 18.96	**261.83 ± 2.30**	261.74 ± 8.42
Boxing	97.92 ± 1.08	**99.93 ± 0.03**	99.77 ± 0.17	99.85 ± 0.14
Breakout	62.09 ± 34.67	696.94 ± 63.20	747.62 ± 64.06	**831.08 ± 6.18**
Centipede	14411.24 ± 1780.60	**348288.74 ± 11957.45**	112609.74 ± 42701.73	245892.18 ± 39060.78
Chopper Command	3535.40 ± 1031.08	**959747.29 ± 10225.76**	842327.17 ± 168089.18	912225.00 ± 112906.65
Crazy Climber	85166.13 ± 9121.39	**456653.80 ± 13192.11**	295413.67 ± 5974.67	339274.67 ± 14818.41
Defender	33613.13 ± 6856.21	**666433.14 ± 17493.30**	518605.50 ± 18011.92	543979.50 ± 7639.27
Demon Attack	1786.19 ± 911.24	140474.21 ± 2931.07	139349.75 ± 1927.91	**142176.58 ± 1223.59**
Double Dunk	−21.76 ± 0.52	23.64 ± 0.06	23.60 ± 0.06	**23.70 ± 0.44**
Enduro	437.34 ± 97.19	2349.03 ± 7.46	2338.62 ± 38.96	**2360.64 ± 3.19**
Fishing Derby	−43.36 ± 19.65	**83.42 ± 4.11**	67.19 ± 5.30	77.05 ± 3.97
Freeway	22.46 ± 0.49	32.13 ± 0.71	33.82 ± 0.14	**33.97 ± 0.02**
Frostbite	980.15 ± 581.61	507775.65 ± 35925.95	136691.77 ± 35672.33	**526239.50 ± 18289.50**
Gopher	9760.20 ± 1790.84	98786.14 ± 4600.57	117557.53 ± 3264.72	**119457.53 ± 4077.33**
Gravitar	666.43 ± 304.08	18180.26 ± 627.48	13049.67 ± 272.66	**20875.00 ± 844.41**
H.E.R.O.	8850.63 ± 1313.70	102145.96 ± 50561.27	33872.29 ± 6917.14	**199880.60 ± 44074.56**
Ice Hockey	−14.87 ± 1.71	**62.33 ± 5.77**	26.07 ± 4.48	47.22 ± 4.41
Jamesbond	534.53 ± 408.19	107266.11 ± 15584.58	**137333.17 ± 21939.77**	117009.92 ± 55411.15
Kangaroo	3178.86 ± 996.26	**18505.37 ± 5016.41**	15863.50 ± 675.45	17311.17 ± 419.17
Krull	9179.71 ± 1222.12	**194179.21 ± 21451.96**	157943.83 ± 26699.67	155915.32 ± 43127.45
Kung-Fu Master	31613.73 ± 8854.35	192616.60 ± 9019.89	364755.65 ± 387274.47	**476539.53 ± 518479.85**
Montezuma's Revenge	200.43 ± 192.50	8666.10 ± 2928.92	2863.00 ± 117.94	**12437.00 ± 1648.44**
Ms. Pac-Man	3348.12 ± 630.07	**57402.56 ± 3077.27**	22853.12 ± 2843.55	29747.91 ± 2472.33
Name This Game	4463.97 ± 1747.50	**48644.27 ± 2390.83**	31369.93 ± 2637.53	40077.73 ± 2274.25
Phoenix	7359.04 ± 5542.07	**858909.13 ± 37669.01**	602393.53 ± 43967.79	849969.25 ± 43573.52
Pitfall!	274.33 ± 413.21	13655.05 ± 5288.29	574.32 ± 811.43	**46734.79 ± 30468.85**
Pong	−15.02 ± 1.96	**20.29 ± 0.65**	17.91 ± 6.61	19.31 ± 2.42
Private Eye	5727.66 ± 1619.10	79347.98 ± 29315.82	64145.31 ± 21106.93	**100798.90 ± 1.07**
Q*BERT	3806.99 ± 1654.33	**437607.43 ± 111087.57**	96189.83 ± 17377.23	238453.50 ± 272386.91
Riverraid	6077.47 ± 1810.98	56276.56 ± 6593.69	40266.92 ± 4087.60	**90333.12 ± 4694.40**
Road Runner	25303.07 ± 4360.56	168665.40 ± 40390.00	**447833.33 ± 128698.32**	399511.83 ± 111036.59
Robotank	13.67 ± 2.55	**116.93 ± 10.64**	87.79 ± 5.85	114.46 ± 3.71
Seaquest	2146.63 ± 1574.51	**999063.77 ± 1160.16**	577162.47 ± 56947.06	960181.39 ± 25453.79
Skiing	−25261.49 ± 1193.77	−4289.49 ± 628.37	−3401.56 ± 185.93	**−3273.43 ± 4.67**
Solaris	2968.25 ± 1470.62	**39844.08 ± 6788.17**	13514.80 ± 1231.17	28175.53 ± 4859.26
Space Invaders	640.13 ± 185.45	35150.40 ± 3388.53	33214.80 ± 5372.10	**57828.45 ± 7551.63**
Stargunner	11214.14 ± 4667.13	**796115.29 ± 73384.04**	221215.33 ± 13974.19	264286.33 ± 10019.21
Surround	−8.57 ± 0.69	8.83 ± 0.58	9.64 ± 0.17	**9.82 ± 0.05**
Tennis	−18.34 ± 2.41	**23.40 ± 0.15**	23.18 ± 0.53	22.79 ± 0.65
Time Pilot	3561.51 ± 1114.00	382111.86 ± 17388.79	169812.33 ± 37012.23	**404751.67 ± 17305.23**
Tutankham	106.68 ± 13.87	**2012.54 ± 2853.44**	402.16 ± 22.73	1030.27 ± 11.88
Up'n Down	15986.44 ± 2213.66	**614068.80 ± 32336.64**	472283.82 ± 23901.66	524631.00 ± 20108.60
Venture	477.71 ± 251.24	2544.90 ± 403.53	2261.17 ± 66.39	**2859.83 ± 195.14**
Video Pinball	18042.46 ± 2773.10	**885718.05 ± 54583.24**	778530.78 ± 79425.86	617640.95 ± 127005.48
Wizard Of Wor	3402.48 ± 1210.12	**134441.09 ± 8913.57**	67072.67 ± 13768.12	71942.00 ± 6552.86
Yars' Revenge	26310.18 ± 6442.63	**976142.42 ± 3219.52**	654338.02 ± 100597.12	633867.66 ± 128824.41
Zaxxon	7323.83 ± 1819.10	**195043.97 ± 18131.20**	79120.00 ± 9783.55	77942.17 ± 6614.61

Table 6 | Scores per game when training and evaluating with *sticky actions* (Machado et al., 2018).

Game	MEME @ 200M	MEME @ 1B	Go-Explore
Alien	48076.48 ± 10310.65	**68634.82 ± 15653.10**	
Amidar	7280.27 ± 808.09	**20776.93 ± 4859.39**	
Assault	27838.75 ± 4337.41	**31708.64 ± 14199.48**	
Asterix	**843493.60 ± 126291.56**	729820.40 ± 82360.83	
Asteroids	212460.60 ± 5585.36	**335137.50 ± 32384.14**	
Atlantis	1462275.60 ± 144898.14	**1622960.80 ± 1958.79**	
Bank Heist	6448.48 ± 3066.16	**45019.92 ± 8611.39**	
Battle Zone	756298.00 ± 71092.41	**763666.00 ± 53978.21**	
Beam Rider	**54395.06 ± 8299.92**	38049.60 ± 3714.79	
Berzerk	16265.88 ± 10497.83	**45729.94 ± 13228.29**	197376 (@10B)
Bowling	**264.73 ± 0.89**	212.70 ± 65.93	260 (@10B)
Boxing	99.77 ± 0.10	**99.86 ± 0.11**	
Breakout	**521.32 ± 49.04**	475.87 ± 53.73	
Centipede	55068.43 ± 6370.11	63792.64 ± 24203.69	**1422628** (@10B)
Chopper Command	170450.00 ± 318026.00	**181573.80 ± 342149.46**	
Crazy Climber	272227.40 ± 24884.40	**291033.20 ± 5966.79**	
Defender	521330.30 ± 10194.48	**561521.30 ± 7955.85**	
Demon Attack	130545.44 ± 9343.72	**142393.12 ± 1119.30**	
Double Dunk	23.12 ± 0.58	**23.78 ± 0.09**	
Enduro	2339.31 ± 13.75	**2352.94 ± 14.33**	
Fishing Derby	68.12 ± 4.92	**79.65 ± 2.65**	
Freeway	33.88 ± 0.08	33.92 ± 0.02	**34** (@10B)
Frostbite	137638.50 ± 38943.68	**498640.46 ± 38753.40**	
Gopher	**105836.08 ± 13458.22**	96034.76 ± 17422.13	
Gravitar	12864.10 ± 260.14	**19489.40 ± 825.38**	7588 (@10B)
H.E.R.O.	27998.94 ± 5920.99	**175258.87 ± 15772.55**	
Ice Hockey	38.14 ± 9.23	**52.51 ± 10.04**	
Jamesbond	**144864.80 ± 8709.86**	118539.20 ± 47454.71	
Kangaroo	15559.00 ± 1494.83	**16951.60 ± 275.12**	
Krull	**127340.42 ± 27604.33**	93638.02 ± 28863.77	
Kung-Fu Master	182036.20 ± 4195.48	**208166.80 ± 2714.94**	
Montezuma's Revenge	2890.40 ± 42.06	9429.20 ± 1485.32	**43791** (@10B)
Ms. Pac-Man	25158.68 ± 1786.43	**27054.73 ± 152.14**	
Name This Game	29029.20 ± 2237.35	**34471.30 ± 2135.99**	
Phoenix	440354.08 ± 70760.72	**788107.78 ± 33424.51**	
Pitfall!	235.90 ± 493.77	**7820.94 ± 16815.61**	6954 (@10B)
Pong	19.38 ± 0.58	**20.71 ± 0.08**	
Private Eye	90596.05 ± 19942.76	**100775.10 ± 15.57**	95756 (@10B)
Q*BERT	137998.15 ± 86896.43	**328686.85 ± 257052.72**	
Riverraid	36680.36 ± 1287.15	**67631.40 ± 4517.53**	
Road Runner	515838.00 ± 153908.19	**543316.20 ± 64169.67**	
Robotank	93.47 ± 4.18	**114.60 ± 4.61**	
Seaquest	474164.86 ± 62059.73	**744392.88 ± 41259.26**	
Skiing	−3339.21 ± 14.59	**−3305.77 ± 8.09**	−3660 (@10B)
Solaris	13124.24 ± 657.30	**28386.28 ± 2381.29**	19671 (@20B)
Space Invaders	26243.09 ± 6053.57	**52254.64 ± 4421.24**	
Stargunner	173677.60 ± 19678.82	**190235.40 ± 6141.42**	
Surround	9.60 ± 0.24	**9.66 ± 0.23**	
Tennis	**22.65 ± 0.73**	22.61 ± 0.53	
Time Pilot	159728.20 ± 39442.90	**354559.80 ± 22172.76**	
Tutankham	383.85 ± 61.43	**924.47 ± 130.59**	
Up'n Down	478535.54 ± 15969.44	**528786.12 ± 5200.79**	
Venture	2318.20 ± 56.39	**2583.40 ± 175.95**	2281 (@10B)
Video Pinball	750858.98 ± 115759.28	**759284.69 ± 37920.13**	
Wizard Of Wor	65005.80 ± 7034.75	**66627.00 ± 9196.92**	
Yars' Revenge	**654251.90 ± 121823.94**	556157.86 ± 147800.84	
Zaxxon	**85322.00 ± 12413.86**	69809.20 ± 2229.49	

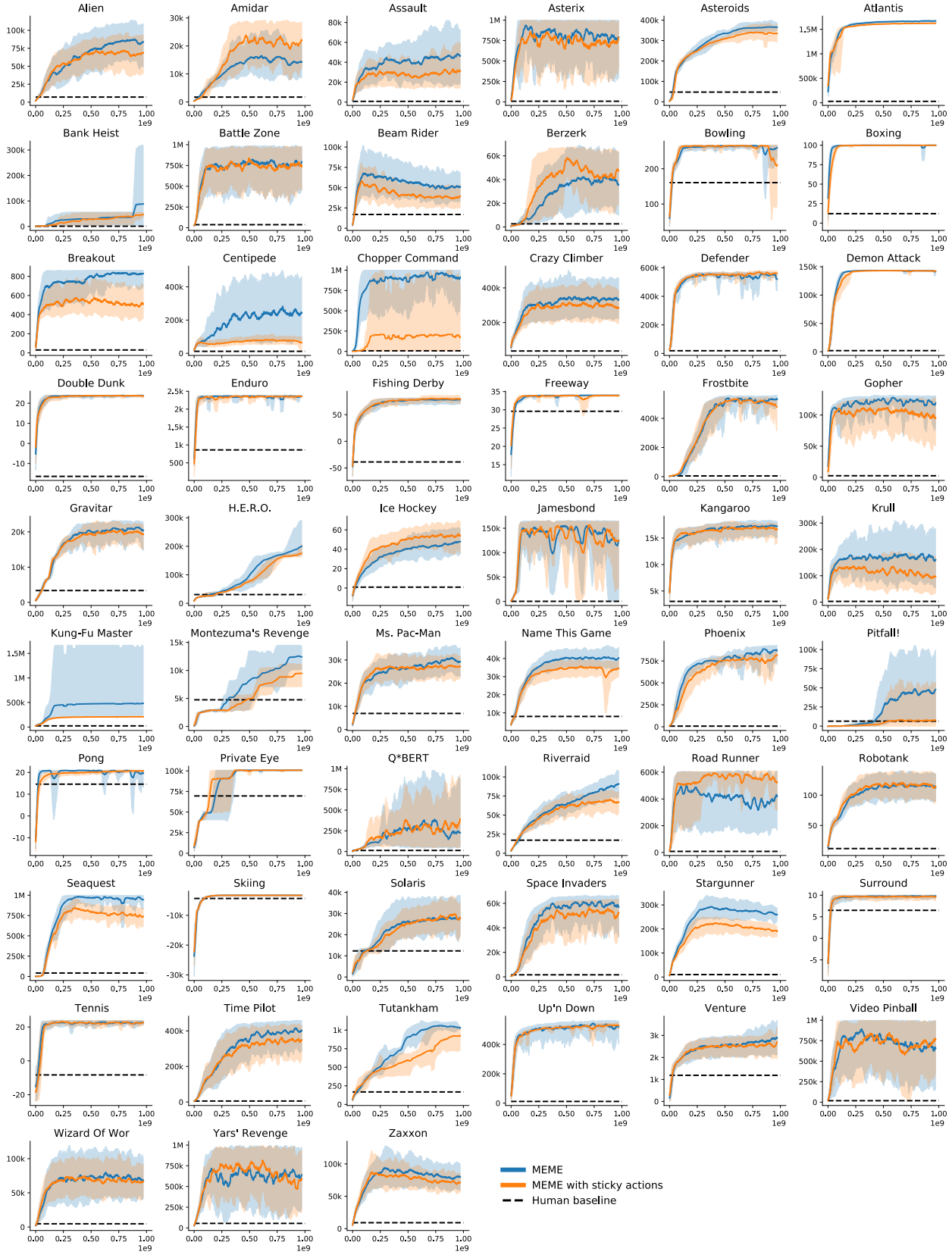


Figure 25 | Score per game as a function of the number of environment frames, both with and without *sticky actions* (Machado et al., 2018). Shading shows maximum and minimum over 6 runs, while dark lines indicate the mean.