A. Ethics Statement

Deep learning models have grown extremely large over the last few years, and their computational demands keep on increasing. This translates into substantial energy consumption which can have environmental consequences. By performing model training and inference in low precision, it becomes possible to reduce energy expenditure. Deep reinforcement learning methods have many practical applications and can quickly transform into products such as domestic robots and self-driving cars. Upon commercialization, low-precision RL could enable substantial energy savings. RL has the potential for military and malicious applications, but also positive societal outcomes such as scientific discovery and autonomous vehicles. We do not perceive that our work differs significantly from other RL work in this regard.

B. Experimental details

The hyper-parameters for SAC, following (Yarats & Kostrikov, 2020), are given in Table 4. We also use the network architecture of (Yarats & Kostrikov, 2020): both the actor and critic networks have hidden depth 2 and hidden dimensions 1024. The actor outputs $\log \sigma$ for the actions, which is coerced to lie in $[-5, 2]$ via a $\tanh$ non-linearity. The hyperparameters of our methods are listed in table 5. For dynamic loss scaling, we follow the strategy of PyTorch amp (Carilli, 2020). The scale is initialized at some large value $init\_grad\_scale$. After each backward pass, we inspect the gradients. If there are non-finite values, we decrease the loss scale by a factor of 2. If we observe no such issues for $inc\_grad\_scale\_freq$ consecutive epochs, we increase the scale by a factor of 2 and reset this counter. To choose a proper K in eq. (2), one needs to consider $M_{\max}$, the largest number that can be represented in the numerical format. To avoid overflow of $\log(1 + \exp(x))$ in the backwards pass in a naive implementation, one should exchange it for a linear function for $x \approx \log M_{\max}$. Larger limits are possible depending on how the soft-plus function is implemented, we take 10 as it is a round number and works well in practice. The Kahan-momentum updates are made to a scaled buffer to avoid underflow as τ can be small, we use a scale of $1e4$.

Table 4. Hyper-parameters used for SAC, following (Yarats & Kostrikov, 2020).

Parameter	Value
γ	0.99
T_0	0.1
τ	0.005
α_{adam}	1e-4
ϵ_{adam}	1e-8
$\beta_1$adam	0.9
$\beta_2$adam	0.999
batch size	1024
target update freq	2
seed steps	5000
log σ bounds	[-5, 2]
actor update frequency	1

Table 5. Hyper-parameters for our methods.

Parameter	Value
$init\_grad\_scale$	1e4
$inc\_grad\_scale\_freq$	1e4
K	1e1
Kahan-momentum scale	1e4

C. Proof of Statement 1

hAdam. We prove this by induction on the statement $w_t = \sqrt{v_t}$. It holds at $t = 0$ as both buffers are initialized to zero, thus we only need to prove the induction step. Assuming $w_t = \sqrt{v_t}$ we have

$$w_{t+1} = \text{hypot}(\sqrt{\beta} w_t, \sqrt{(1-\beta)} g_{t+1})$$

$$= \text{hypot}(\sqrt{\beta}\sqrt{v_t}, \sqrt{(1-\beta)} g_{t+1})$$

$$= \sqrt{\beta v_t + (1-\beta) g_{t+1}^2} = \sqrt{v_{t+1}}$$

For the Adam/hAdam updates we then have $\frac{m}{\sqrt{v}+\epsilon} = \frac{m}{w+\epsilon}$, and thus the updates are identical.

Compound loss-scaling. For any $\gamma > 0$ we have $\frac{\gamma m}{\gamma(w+\epsilon)} = \frac{m}{w+\epsilon}$. Thus, as long as ϵ is scaled by γ, this modification changes nothing in infinite precision.

Normal-fix. In infinite precision we have $\frac{(x-\mu)^2}{\sigma^2} = \left(\frac{x-\mu}{\sigma}\right)^2$. Thus, the normal fix will not change anything.

Kahan-momentum. In infinite precision where arithmetic is commutative and associative, Kahan summation reduces to normal summation.

Kahan-gradients. In infinite precision where arithmetic is commutative and associative, Kahan summation reduces to normal summation. ■

D. Infrastructure

Experiments were conducted with PyTorch 1.7.0 on Nvidia Tesla V100 GPUs using CUDA 10.2 and CUDNN 7.6.0.5, except for the performance measurements where we also

consider CUDA 11.0 and CUDNN 8.0.0.5. See Appendix H for details on the performance measurements.

E. Additional Experiments

We here present some additional supporting experiments. In Figure 9 we show the original ablation experiments with results broken down by task. We see that all tasks require several of the proposed methods to work well, but also variation between tasks. Specifically, it seems like some tasks require fewer methods to work well – suggesting that these are more robust to numerical issues. To further verify that our proposed methods contribute individually to the performance, we perform an ablation experiment where we remove one component from the final agent which uses all other techniques. The results, averaged over seeds and environments, are shown in Figure 7. We see that all proposed methods are needed to reach satisfactory performance across all games.

We also compare our proposed method against the baselines from the main text with some hyperparameter modifications. Specifically, we consider 1) using the default amp settings for the loss scaler schedule – an initial scale of 2^{16} and a growth interval of 2000. We refer to this modification as amp in the figures. We also consider 2) to increase ϵ in Adam by a factor of 10 to stabilize training. We refer to this modification as eps in the figures. Results are shown in Figure 8. None of these methods improve the training substantially.

In the main text, we compare against training RL from pixels with weight standardization applied to the linear layer. In Figure 10 we compare our method (which uses weight standardization) against an fp32 baseline which does not. Again, the results are close.

F. Random Parameters

Most RL algorithms requires hyperparameter tuning to successfully adapt to new environments. In the main text, we simply use the default hyperparameters from (Yarats & Kostrikov, 2020). To demonstrate the parameter stability of our method, we now consider random hyperparameters. We generate five sets of random hyperparameters, randomizing learning rate α, discount γ, initial temperature T_0, batch size, critic update speed τ, action bound as per Table 6, keeping other parameters from (Yarats & Kostrikov, 2020) (see Table 4). The average scores are given in Table 7, and our method performs close to fp32 across the different hyperparameter groups, thus showing that our method is stable across hyper parameters.

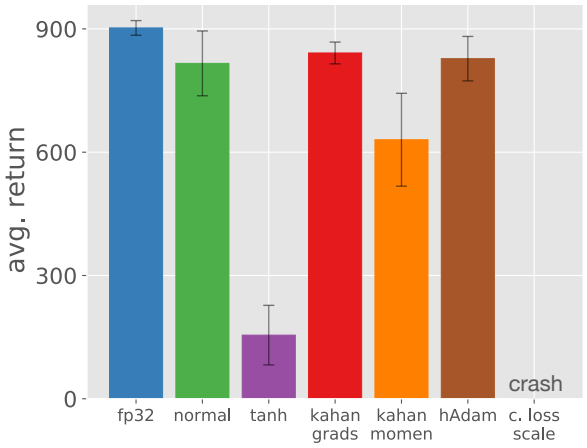


Figure 7. We remove one of our proposed method from the complete agent (which uses the other 5 methods) and then train the agent in fp16. Results are averaged over seeds and environments. Removing any single method decreases the performance, suggesting that all proposed methods are needed to reach the final performance.

G. Details on RL from Pixels

The hyper-parameters we use for SAC from pixels follow (Kostrikov et al., 2020). They are largely the same as for SAC from states, the differences are listed in Table 9. We use 100 as the scale of the Kahan momentum and add a ϵ ($1\mathrm{e}{-4}$) to the outputted σ of the actor. This prevents underflow which now can happen as (Kostrikov et al., 2020) uses a larger range for the stds. Otherwise, we use hyper-parameters from Table 5 for our methods. Following (Hafner et al., 2019; Kostrikov et al., 2020) we use action repeat for the tasks as per Table 8. Images are resized to 84-by-84 RGB images, we then use frame stacking of three to get input shapes of (9, 84, 84). We follow (Kostrikov et al., 2020) and apply image augmentations to the input. Specifically, we use random cropping with padding by 4 and input both an augmented batch of size 512 and an original batch of 512 to the critic network, resulting in a total batch size of 1024. The actor and α uses a batch size of 512 for their updates.

For the CNN encoder, we use four convolutional layers with ReLu non-linearity between them. All convolutional layers have spatial extent 3-by-3, the first layer uses a stride of 2, the others use stride 1. By default, all convolutional layers have 32 filters. The feature map from the convolutions is fed into a linear layer which outputs a 50-dimensional vector which is fed into a layer-normalization layer (Ba et al., 2016). We occasionally observed the internal variance calculations of the layer normalization overflowing. To remedy this, we apply weight standardization (Qiao et al., 2019) to the linear layer and further down-scale output larger than 10 to 10,

Table 6. Random hyperparameters obtained. Learning rate is obtained from a log uniform distribution over $[1e{-}5, 1e{-}3]$, $\min \log \sigma$ over a uniform distrubution over $[-7, -3]$, τ over a uniform distribution over $[0.0025, 1e{-}2]$, T_0 over a log uniform distribution over $[1e{-}2, 1e{-}1]$ and batch size from the discrete distribution $\{512, 2024, 2048\}$.

	γ	learning rate	$\min \log \sigma$	τ	T_0	batch size
params 1	0.921	0.000669	-5.37	0.00331	0.0797	512
params 2	0.983	0.0000751	-5.64	0.00289	0.0137	2048
params 3	0.979	0.0000751	-6.99	0.00577	0.0122	2048
params 4	0.982	0.00097	-4.07	0.00327	0.0111	2048
params 5	0.948	0.0000263	-6.92	0.00933	0.14	1024

Table 7. Average scores for random hyperparameters. Scores are averaged across 3 seeds and 6 environments. Parameters are generated per Table 6. The scores are similar for fp32 and our fp16 agent, demonstrating that our methods can match fp32 results robustly across parameters.

avg. reward	params 1	params 2	params 3	params 4	params 5
fp32	767 ± 11	877 ± 14	872 ± 12	887 ± 60	732 ± 50
fp16 (ours)	778 ± 27	869 ± 33	862 ± 29	880 ± 54	709 ± 36

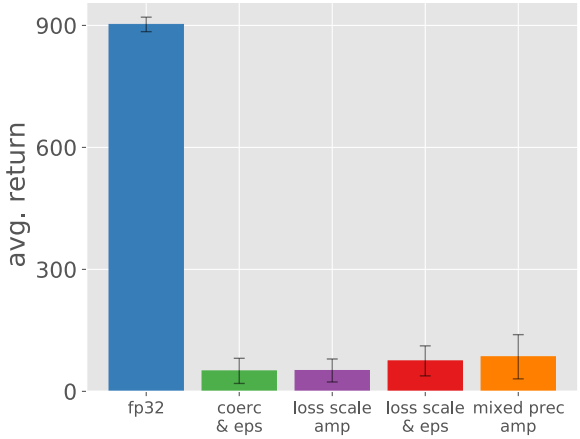


Figure 8. The performance of our methods compared to a few additional baselines. We specifically consider 1) standard loss scaling, but using the amp default settings for the dynamic loss scaling, referred to as amp; and 2) increasing the value of ϵ in Adam by a factor of 10 to increase the numerical stability, referred to as eps. None of these methods work well.

task	action repeat
Cartpole Swingup	8
Reacher Easy	4
Cheetah Run	4
Finger Spin	2
Ball In Cup Catch	4
Walker Walk	2

Table 8. The action repeat hyper-parameter for each task, values come from (Hafner et al., 2019).

Table 9. Hyper-parameters used for SAC from pixels, following (Kostrikov et al., 2020). We only list hyperparameters that differ from those given in table 4.

Parameter	Value
τ	0.01
α_{adam}	1e-3
seed steps	1000
actor update frequency	2
log σ bounds	[-10, 2]

this avoids overflow in the layer-norm calculations. Since layer-norm is invariant under rescaling the input and adding a constant term, these modifications will not change layer-norm in infinite precision. This strategy could likely be implemented in the layer-norm CUDA kernel, but since it is not open-source we defer such investigations to future work. Please consult our code for the PyTorch implementation which is based upon the codebase of (Kostrikov et al., 2020).

H. Details on Performance Measurements

Memory consumption and throughput are measured and averaged over 500 iterations, with 500 iterations as a warm start. We include momentum updates for the target network in these measurements. Time is measured with CUDA events, which are provided natively in PyTorch. Memory is simply measured as the maximum CUDA memory allocated during training, which again is provided natively in PyTorch.

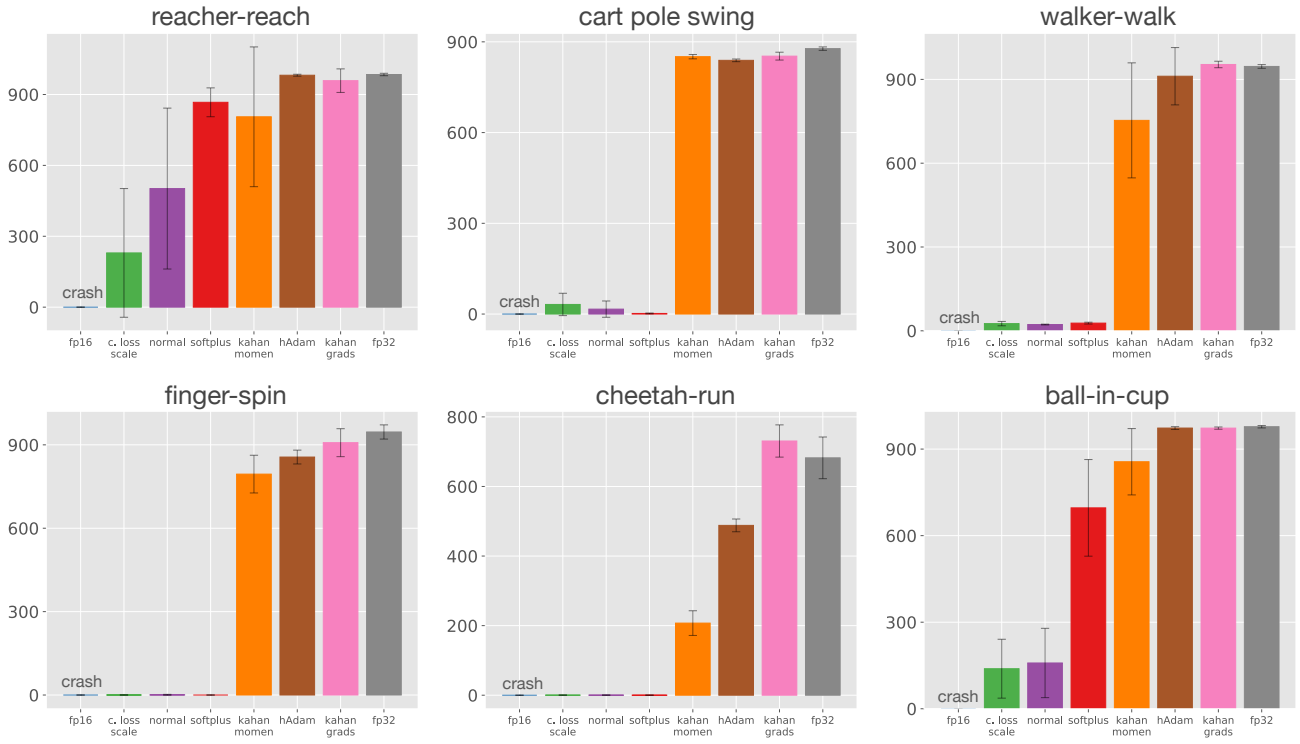


Figure 9. Performance of fp16 training as we add our proposed methods one by one. Scores are broken down by individual tasks. For all tasks, many proposed methods are needed to reach satisfactory performances. However, the games differ in how many are needed, suggesting that some tasks might be more numerically robust. This figure follows Figure 3, but for individual tasks.

We observed non-deterministic CUDNN sometimes settling for sub-optimal GPU kernels. To provide the memory and compute measurements closest to optimal performance, we report the best numbers obtained when setting CUDNN to deterministic or non-deterministic and when using 1) CUDA 11.0 and CUDNN 8.0.0.5 or 2) CUDA 10.2 and CUDNN 7.6.0.5.

In Table 10 we show the performance of SAC from states and observe that the gains become very large as the computational demands grow. Performance differences for the largest models might be related to cache issues rather than the absolute improvement in speed for individual floating-point operations. For the smallest models, where updates take less than 20 milliseconds, the overhead of our methods is larger than the gains from using half-precision numbers. Again, the reason for this is likely that such small workloads do not saturate available parallel resources on the V100, for less performant hardware we would expect a larger difference for small configurations. In Table 11 we instead show the memory footprint for SAC from states. The improvement is relatively constant but scales somewhat differently with the batch size of model capacity. This is natural as the memory footprint of Kahan summation scales with model size. We have observed that moving the entire network to half-precision, without any further modifications, can result in memory savings slightly under 50%. Thus, it seems like the CUDA kernels or the tensor caching policy is not identical for fp32 and fp16. If they were, we would likely see slightly larger memory improvements.

Table 10. Time (milliseconds) for processing one minibatch for SAC from states as a function of network width and batch size (bsize), measured for the Cheetah task. For the smallest model, where a update takes less than 20 milliseconds, the overhead of our fp16 method is larger than the gains. However, as the computational demands increase the benefits grow very large.

width	1024	1024	4096	4096
bsize	1024	4096	1024	4096
fp32	16.63	17.94	58.22	202.38
fp16 (ours)	17.38	16.99	20.58	45.64
improvement	0.96	1.06	2.83	4.43

I. Comparison of Learned Models with Different Numerical Precision

We now investigate how differences in precision influence the learned neural network models. In Figure 11, we show

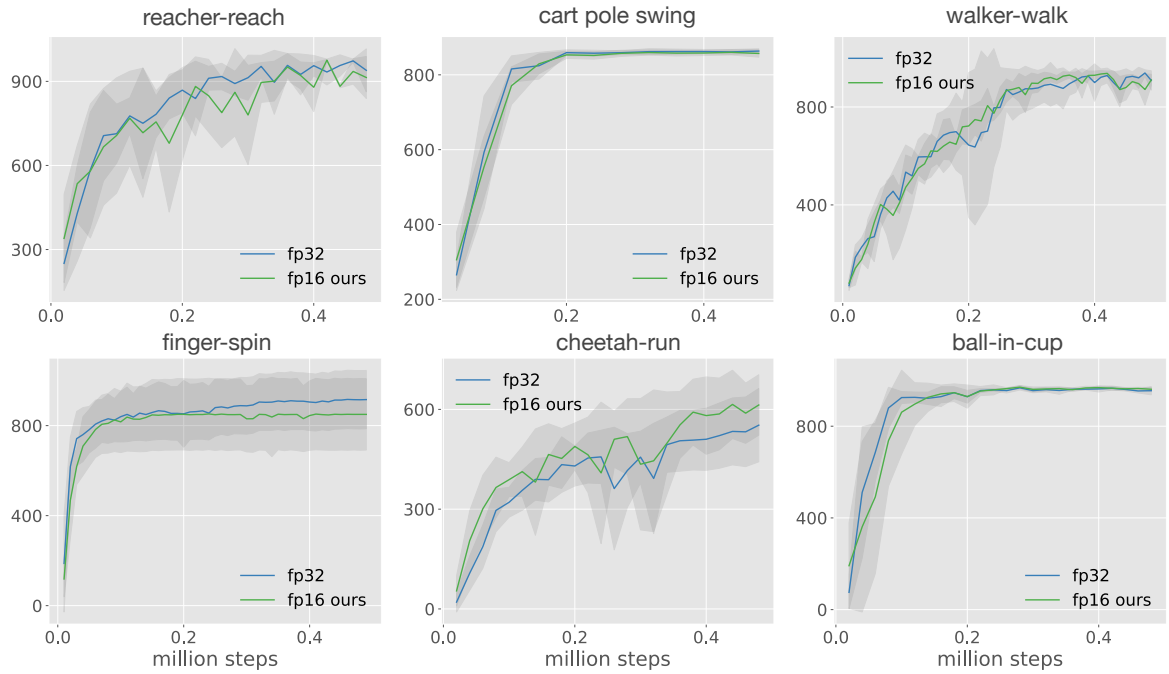
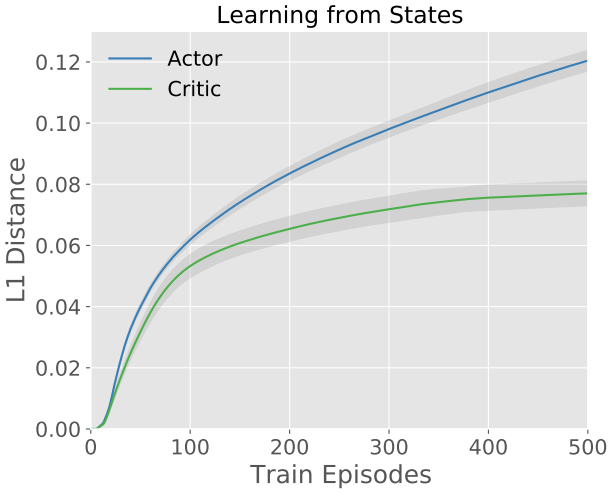


Figure 10. Learning curves for SAC trained from pixels comparing standard fp32 training and training in fp16 with our modifications. The fp32 baseline here does not use weight standardization. Average performance is again close.

Table 11. Memory (MB) consumed for SAC from states as a function of network width and batch size (bsize), measured for the Cheetah task. The memory benefits are relatively constant across computational demands, but scale slightly differently with batch size and model capacity. This is natural as Kahan summation requires storing tensors the size of the model weights.

width	1024	1024	4096	4096
bsize	1024	4096	1024	4096
fp32	128	320	1265	1973
fp16 (ours)	77	185	826	1163
improvement	1.67	1.73	1.53	1.7

the L1 distance between the model weights learned with different precision – full fp32 precision and half fp16 precision. In Figure 12, we show their difference in terms of the predicted Q value on the same state. Models trained with different precision differ in weights, and the difference grows with training. The Q value difference increases in the beginning but will eventually converge, although not to 0. The convolutional model (trained on images) has a larger difference as well as variance compared with the linear model (trained on states) in terms of both model weight and predicted Q values.



(a)

(b)

Figure 11. Average L1 distance between the actor/critic weights during training. We compare in 2 settings: (a) *learning from states* and (b) *learning from images*. We trained 3 pairs of SAC agents, each pair with the same random seed, and report the L1 distance between learned weights averaged over 3 pairs.

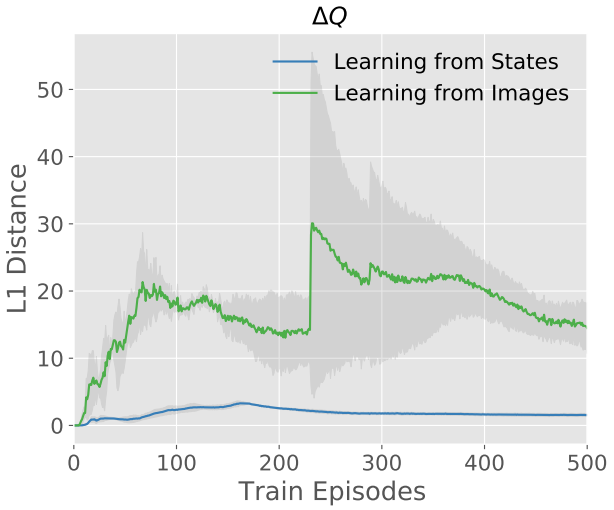


Figure 12. Difference between the Q value prediction from learned models. We trained 3 pairs of SAC agents, each pair with a same random seed, and report absolute value of difference between each pair's predicted Q values averaged on $2,000$ states encountered during training.