Supplementary Material

A. Experimental Details

A.1. Performance analysis

In this section, we provide details of the experimental setup used in our performance benchmarks. One of the goals of our experiments was to compare the performance of different asynchronous RL algorithms "apples to apples", i.e. where all the details that influence throughput are exactly the same for all methods we compare. This includes hardware configuration, simulated environments and their settings (e.g. observation resolution), model size and architecture, and the number of environment instances sampled in parallel.

A.1.1. Hardware configuration

We focused on commodity hardware often used for deep learning experimentation. Systems #1 and #2 were used for performance benchmarks. System #3 is similar to System #2, except with four accelerators instead of one. We used System #3 for our large-scale experiments with self-play and population-based training. See Table A.1 for details.

	System #1	System #2	System #3
Processor	Intel Core i9-7900X	2 x Intel Xeon Gold 6154	2 x Intel Xeon Gold 6154
Base frequency	3.30 GHz	3.00 GHz	3.00 GHz
Physical cores	10	36	36
Logical cores	20	72	72
RAM	128 GB DDR4	256 GB DDR4	256 GB DDR4
GPUs	1 x NVidia GTX 1080Ti	1 x NVidia RTX 2080Ti	4 x NVidia RTX 2080Ti
GPU memory	11GB GDDR5X	11GB GDDR6	11GB GDDR6
OS	Ubuntu 18.04 64-bit	Ubuntu 18.04 64-bit	Ubuntu 18.04 64-bit
GPU drivers	NVidia 440.44	NVidia 418.40	NVidia 418.40

Table A.1. Hardware setups used for profiling and performance measurements (Systems #1 and #2) and for large-scale experiments with self-play and PBT (System #3).

A.1.2. Environments

We used three reinforcement learning domains for benchmarking: Atari, VizDoom, and DeepMind Lab. For Atari we simply chose Breakout with 4-framestack, although other environments exhibit almost identical throughput. The VizDoom scenario we selected is a simplified version of *Battle* with a single discrete action head and the input space including only the pixel observations (no auxiliary game info). Most of the frameworks we tested do not support complex action and observation spaces, so this simplification allowed us to use the exact same version of the environment for all of the evaluated algorithms without major code modifications.

We chose *rooms_collect_good_objects_train* from DMLab-30 as our benchmark environment for DeepMind Lab. This environment is also referred to as *seekavoid_arena_01* in prior work (Espeholt et al., 2018). Just like the VizDoom scenario, this environment has pixel-based observations and a simple discrete action space.

In DeepMind Lab some environment states can be significantly harder to render, and therefore the simulation time depends on the behavior of the agent, e.g. as the agent learns to explore the environment the simulation can slow down or speed up as the distribution of visited states changes. To eliminate this potential source of variance in throughput we ignore the action distribution provided by the policy and sample actions randomly instead in our performance measurements for DMLab.

This way we can measure only the throughput, disentangled from the learning performance. Note that using the random policy for acting does not change the amount of computation done by the algorithm. We collect and process the experience in the exact same way, only the actions sampled from the policy are replaced by random actions on the actors.

VizDoom environments are rendered with native resolution of $160 \times 120 \times 3$ which is downsampled to $128 \times 72 \times 3$. For DMLab the observation resolution is $96 \times 72 \times 3$. For VizDoom and DMLab we used 4-frameskip and no framestacking. Atari frames are rendered in $210 \times 160 \times 3$ and downsampled to 84×84 greyscale images. For Atari we used 4-frameskip and 4-framestack in all measurements, although higher overall throughput can be achieved without frame stacking. Following (Espeholt et al., 2018) and (Espeholt et al., 2019) we report the throughput of all algorithms measured in *environment frames* per second, i.e. a number of simulated environment transitions, or, in our case, $4\times$ the number of samples processed by the learner per second.

A.1.3. MODEL ARCHITECTURES

In all our performance benchmarks we used the same convolutional neural network to parameterize the actor and the critic, which is similar to model architectures used in prior work (Mnih et al., 2016; Espeholt et al., 2018). In our implementation the 3-layer convolutional head is followed by a fully-connected layer, an LSTM core, and another pair of fully-connected layers to output the action distribution and the baseline. This architecture is referred to as *simplified* (see Figure A.1), in contrast to the *full* architecture used in *Battle*, *Deathmatch*, and *Duel* experiments, that contains additional observation and action spaces. We used the *simplified* architecture to benchmark throughput in Atari, VizDoom, and DMLab.

Note that in our large-scale VizDoom experiments with the *full* model we chose to use GRU RNN cells (Cho et al., 2014) instead of LSTM (Hochreiter & Schmidhuber, 1997). Empirically we find that GRU cells exhibit similar sample efficiency to LSTM cells and require slightly less computation.

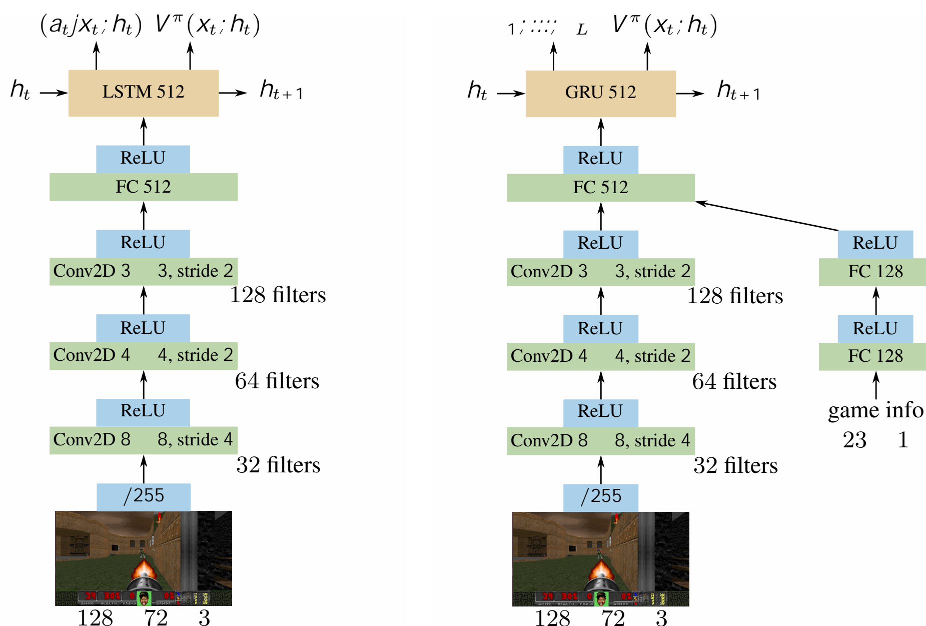

Figure A.1. Neural network architectures used in VizDoom experiments. **Left:** *simplified* architecture used for performance measurements and standard VizDoom environments. **Right:** *full* architecture with additional low-dimensional game information input (health, armor, ammunition, etc.) and L independent action heads.

A.1.4. BENCHMARKING RESULTS

We provide benchmarking results in the tabular form (see Table A.2). Data points are omitted for configurations that could not be initialized due to lack of resources, such as memory, simultaneously open file descriptors, or active parallel threads. Since Sample Factory allocates a very minimal amount of resources per environment instance, we were able to test configurations running as many as 3000 environments on a single machine for VizDoom and Atari, although increasing

number of environments further provides diminishing returns.

Table A.3 shows performance figures for SampleFactory in some additional scenarios. As mentioned in the main paper, using GPU for rendering DeepMind Lab environments can improve performance, especially on systems with fewer CPU cores (e.g. System #1).

Finally, we show performance figures for population-based training scenarios. Here we use 4 GPUs to accelerate learners and policy workers associated with up to 12 agents trained in parallel. Performance figures show that there is a very small penalty for increasing the population size, despite the fact that the amount of communication required grows significantly (e.g. rollout workers have to send observations to many different policy workers associated with different agents). The measurements in the table show the performance only for single-player environments. Multiplayer environments that involve actual network communication between individual game instances are significantly slower, up to 2-3 times, depending on the number of communicating instances. Significant performance gains are possible through replacing network communication between game instances with faster local mechanism, although this could require significant modifications to the VizDoom engine and lies beyond the scope of this project.

System 1 (10xCPU, 1xGPU)

	Atari 84x84x4						VizDoom 128x72 RGB						DmLab 96x72 RGB					
# of envs sampled:	20	40	80	160	320	640	20	40	80	160	320	640	20	40	80	160	320	640
DeepMind IMPALA	6350	6470	6709	6880	-	-	6615	6776	7041	6669	-	-	6179	5943	6133	6448	-	-
SeedRL IMPALA	11347	15734	20715	24906	26149	-	11443	14537	19705	22059	22733	-	6747	10293	11262	11191	10604	-
RLLib IMPALA	10808	13596	17744	20236	21192	18232	10676	12556	12472	13444	11500	11868	7736	9224	9948	11644	11516	-
rlpyt PPO	13312	17764	21772	27240	31408	35272	16268	23688	26448	31660	38908	41940	**9028**	10852	11376	11560	12280	12400
SampleFactory APPO	**17544**	**25307**	**35287**	**42113**	**46169**	**48016**	**16985**	**24809**	**37300**	**47913**	**55772**	**59525**	8183	**11792**	**12903**	**13040**	**13869**	**14746**

System 2 (36xCPU, 1xGPU)

	Atari 84x84x4						VizDoom 128x72 RGB						DmLab 96x72 RGB					
# of envs sampled:	72	144	288	576	1152	1728	72	144	288	576	1152	1728	72	144	288	576	1152	1728
DeepMind IMPALA	9661	8826	8602	-	-	-	10708	10043	9990	-	-	-	8782	8622	8491	-	-	-
SeedRL IMPALA	25400	33425	39500	39726	-	-	23395	29591	34428	-	-	-	22814	30354	32149	34773	-	-
RLLib IMPALA	19148	20960	20440	19328	19360	22440	11471	11361	12144	11974	12098	12391	12536	13084	13932	-	-	-
rlpyt PPO	24520	33544	39920	53112	63984	68880	37848	40040	57792	68644	71080	73544	22700	24140	29180	29424	32652	32948
SampleFactory APPO	**37061**	**59610**	**81247**	**95555**	**120355**	**135893**	**38955**	**61223**	**79857**	**103658**	**131571**	**146551**	**26421**	**37088**	**41781**	**42149**	**41383**	**41784**

Table A.2. Throughput of asynchronous RL methods measured in environment frames per second (samples per second 4).

Hardware	Training scenario	Rollout workers	Total number of envs	Throughput, env. frames/sec
System # 1	DMLab with GPU rendering	20	160	17952
System # 1	DMLab with GPU rendering	20	320	18243
System # 3	VizDoom *Battle* PBT, *full* model, 4 agents	72	2304	153602
System # 3	VizDoom *Battle* PBT, *full* model, 8 agents	72	2304	154081
System # 3	VizDoom *Battle* PBT, *full* model, 12 agents	72	2304	146443

Table A.3. Performance of Sample Factory in additional training scenarios.

A.2. DMLab-30 experiment

In this section we share our findings related to multi-task training on DMLab-30. Overall, we largely follow the same training procedure as the original IMPALA implementation, e.g. we used the exact same model based on ResNet backbone. We found however that seemingly subtle implementation details can significantly influence the learning performance.

One of the key choices when training on a multi-task benchmark like DMLab-30 with an asynchronous RL algorithm is whether to give different tasks the same amount of samples, or the same amount of compute. We follow (Espeholt et al., 2018) and employ the second strategy. Just like the original implementation of IMPALA, we spawn an equal number of workers for every task (in our case 90 workers on a 36-core system, 3 workers per task) and let the OS schedule these processes. Note that this gives somewhat unfair advantage to tasks which render faster, since with the same amount of CPU time more samples can be generated for the faster environments. Sample Factory supports both training regimes, but we

decided to go with the IMPALA strategy to ensure fair comparison of scores. Also, the throughput is higher in this mode. The authors argue that this implementation detail (distribution of compute resources across tasks) should be stated explicitly whenever different multi-task algorithms are compared.

The only significant difference compared to the original IMPALA setup is the chosen action space. We decided to use a slightly different discretization of the game inputs introduced in (Hessel et al., 2019), since it makes the action space closer to the one available to humans, e.g. it allows the agent to turn and move forward within the same frame. The increased number of actions, however, makes exploration harder, and we see a drop in performance in some of the levels where exploration is key. Figure A.2 shows the full breakdown of the agent's performance on individual tasks.

Finally, we noticed that one of the most significant factors affecting the throughput is the level generation at the episode boundary. To make the DMLab-30 benchmark more accessible we release a dataset of pre-generated levels, as well as the environment wrapper that makes it easy to use the dataset with any RL algorithm implementation. This wrapper builds on top of the already existing DMLab level cache. Without relying on the random seed provided by the environment, it will load the levels from the dataset until all of them are used in the training session, after which new levels will be generated and added to the cache. Follow the link below to find instructions on how to download the dataset and use it with Sample Factory: `https://github.com/alex-petrenko/sample-factory#dmlab-level-cache`.

A.3. VizDoom experiments

We used *full* neural network architecture (as shown on Figure A.1) to train our final VizDoom agents. All advanced VizDoom environments we used (*Battle, Battle2, Deathmatch, Duel*) included an additional observation space with game information in numerical form. We only used information available to a human player through in-game UI. This includes: health and armor, current score, number of players in a match, selected weapon index, possession of different types of weapons, and amount of ammunition available for each weapon. We do not use previous rewards as a policy input, because the reward function can be based on hidden in-game information (e.g. damage dealt) and thus may give the agent an unfair advantage at test time.

Table A.4 describes the action space used in VizDoom experiments. We decompose the set of possible actions into seven independent action distributions, which allows the agent to combine multiple actions within the same frame, e.g. run forward, strafe, and attack at the same time. The action space for horizontal aim is technically continuous although in this work we discretize it with 1.25 step, which empirically leads to faster learning.

Action head	**Number of actions**	**Comment**
Moving	3	no-action / forward / backward
Strafing	3	no-action / left / right
Attacking	2	no-action / attack
Sprinting	2	no-action / sprint
Object interaction	2	no-action / interact
Weapon selection	8	no-action / select weapon slot 1..7
Horizontal aim	21	no-action / turning between 12.5 and 12.5 in 1.25 steps
Total number of possible actions	12096	

Table A.4. Action space used in VizDoom multi-agent experiments.

Reward function for *Battle* and *Battle2* is based on the game score ($+1$ for killing a monster) plus a small additional reward for collecting health and ammo packs. In *Deathmatch* and *Duel* we extended the reward function to include penalties for dying, as well as additional rewards for picking up new types of weapons and dealing damage to opponents. Finally, we penalize the agent for switching the weapons too often, which accelerates the training in early stages.

The basic hyperparameters of all our experiments are presented in Table A.5. We deviate from these parameters only in *Deathmatch* and *Duel* experiments where we used action repeat (frameskip) of two consecutive frames instead of four. Consequently, we adjusted the discount factor to 0.995 to account for this change. We observe that in these environments repeating actions fewer times led to better final performance of the agents.

In all hardware setups we used the number of rollout workers equal to the number of CPU cores. This allows us to use CPU affinity setting for processes to minimize the amount of context switching and accelerate sampling. The number of environments per core that enables the highest throughput lies between 2^4 and 2^5 for VizDoom. Note that for systems with

large number of CPU cores a larger batch size might be required to reduce the policy lag. In all our experiments the policy lag was on average between 5 and 10 SGD steps, which results in stable training. Tensorboard summaries were used to monitor the policy lag during training.

Learning rate	$10\ ^{4}$
Action repeat (frameskip)	$2/4$
Framestack	No
Discount	$0.995/0.99$
Optimizer	Adam (Kingma & Ba, 2015)
Optimizer settings	$_1 = 0.9,\ \ _2 = 0.999,\ \ = 10\ ^{6}$
Gradient norm clipping	4.0
Rollout length T	32
Batch size, samples	2048
Number of training epochs	1
V-trace parameters	$\bar{} = \bar{c} = 1$
PPO clipping range	$[1.1\ ^{1}, 1.1]$
Entropy coefficient	0.003
Critic loss coefficient	0.5

Table A.5. Hyperparameters for VizDoom experiments.

A.3.1. POPULATION-BASED TRAINING

In our VizDoom population-based training experiments we used System #3 to train a population of 8 agents in parallel. The full configuration of Sample Factory in this setup includes 72 rollout workers (one worker per logical core), 32 environment instances per rollout worker, 8 policy workers, and 8 learners (one for every policy involved). We deployed 2 learners and 2 policy workers on each available GPU with more GPU memory to spare.

Every $5M$ frames during training we randomly mutate hyperparameters and reward shaping weights of the bottom 70% of the population. The mutation rate is 15% for each hyperparameter. In our experiments we mutated learning rate, entropy loss coefficient, Adam $_1$, and individual reward shaping coefficients by increasing or decreasing these parameters by a factor of 1.2. Additionally, every $5M$ frames we replace the policy weights for the worst 30% of agents with weights of the policy randomly sampled from the best 30%. In *Duel* experiment we introduce an additional threshold that prevents the weights exchange mechanism if policies are relatively close in performance (the difference in win rate is less than 0.35), which helps to increase the diversity of the population.

B. Additional performance considerations

Seemingly small details can make a big difference in the performance of an asynchronous system. We found that tuning CPU core affinity and priority for various components of the system can give us a substantial performance gain. In Sample Factory we recommend setting the number of rollout workers to the number of logical CPU cores. In this case we can use processor affinity to run these worker processes on individual cores, preventing a lot of unnecessary context switching. We also found that in most configurations it helps to deprioritize rollout workers and let policy workers and learners to be scheduled as soon as there is any work available. This helps saturate the rollout workers with actions and increases the overall performance. Sample Factory comes with a default set of priorities/affinities that will work well for many training configurations.

In the highest throughput configurations batching of trajectories into minibatches and transferring them to the GPU can also become a bottleneck. Similar to (Espeholt et al., 2018) and (Espeholt et al., 2019) we implement this preprocessing step in the background thread on the learner, eliminating this particular performance issue.

B.1. FIFO queues

Sample Factory generally avoids explicit data transfer between system components, instead these components exchange addresses in shared memory buffers. Perhaps rather surprisingly, we found that at frame rates above 10^5 FPS even communicating these addresses can be difficult. In fact, at this speed the standard Python's *multiprocessing.Queue* tends to

occupy a significant portion of CPU time.

To solve this issue we implemented our own version of the IPC FIFO queue in C++, based on a circular buffer and POSIX mutexes. This custom implementation is a drop-in replacement for the standard *multiprocessing.Queue* and it allows for 20-30 times faster message exchange in many producers - few consumers configuration, also achieving lower latency. The URL below contains installation instructions and detailed performance measurements:

https://github.com/alex-petrenko/faster-fifo

References

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*, 2014.

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

Espeholt, L., Marinier, R., Stanczyk, P., Wang, K., and Michalski, M. SEED RL: Scalable and efficient deep-rl with accelerated central inference. *CoRR*, abs/1910.06591, 2019.

Hessel, M., Soyer, H., Espeholt, L., Czarnecki, W., Schmitt, S., and van Hasselt, H. Multi-task deep reinforcement learning with popart. In *AAAI*, 2019.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 1997.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.

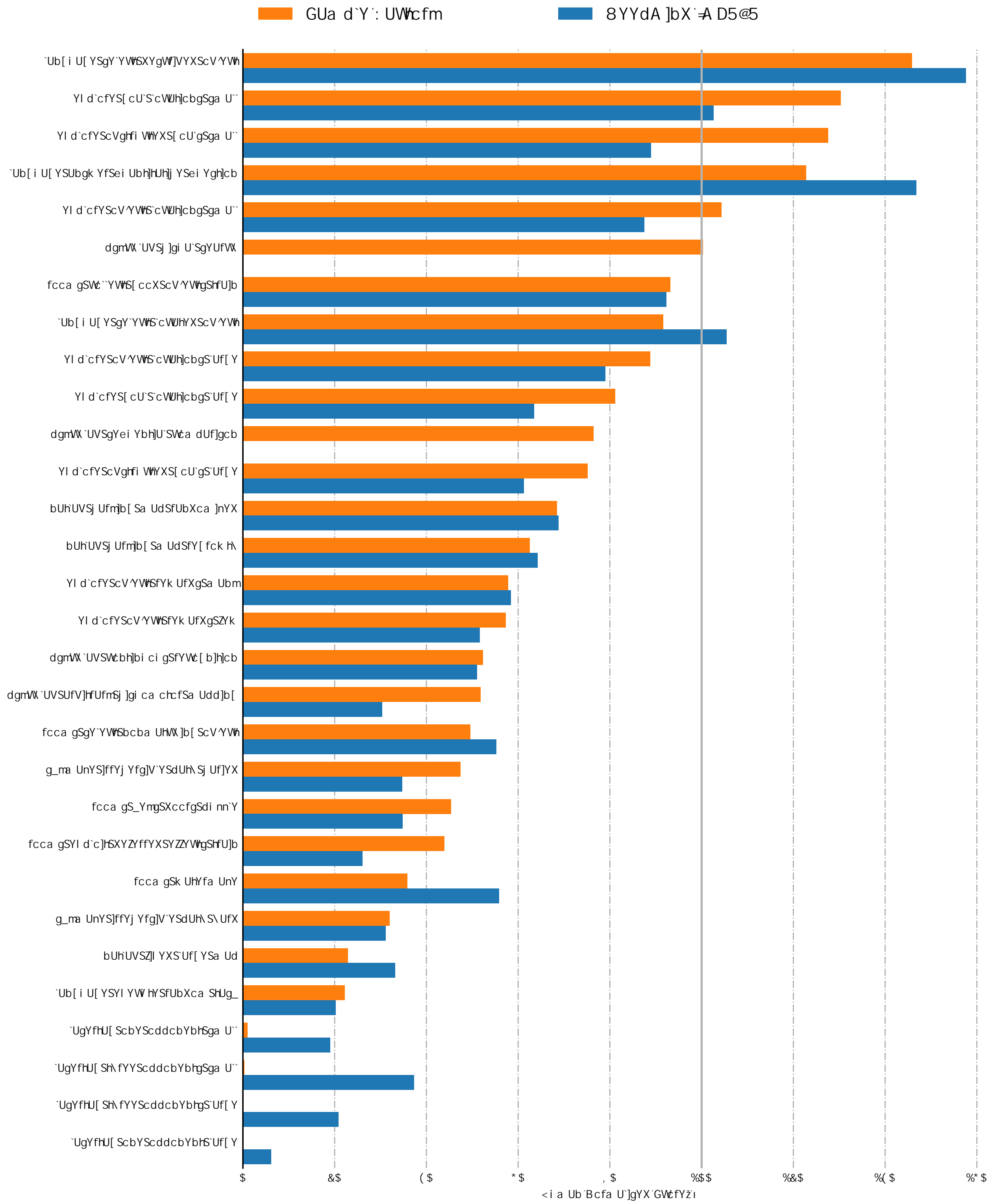

Figure A.2. Final human-normalized training scores for individual DMLab-30 environments.