A Relationship to the Laplace Transform

We often think of RL in terms of discrete timesteps, but the sum in Equation 11 could be replaced with an integral for continuous-time domains, where $\tilde{\boldsymbol{x}}$ is a function of time:

$$\boldsymbol{S}(n) = \int_0^n \tilde{\boldsymbol{x}}(\tau) \odot \exp\left(-\tau \boldsymbol{\alpha}\right) \exp\left(-\tau i \boldsymbol{\omega}\right)^\top d\tau. \tag{23}$$

If we look at just a single dimension of $\tilde{\boldsymbol{x}}$, we have

$$\boldsymbol{S}(n) = \int_0^n \tilde{x}(\tau) \exp\left(-\tau(\alpha + i\boldsymbol{\omega})\right) d\tau, \tag{24}$$

which is equivalent to the Laplace Transform L of a function $\tilde{x}(\tau)$. One way to interpret the connection between our aggregator and the Laplace transform is that the aggregator transforms the memory task into a pole-finding task in the S-plane, a well-known problem in control theory. When we train FFM for an RL task, we are attempting to find the poles and zeros that minimize the actor/critic loss for a batch of sequences.

B Weight Initialization

We find that our model is sensitive to initialization of the $\boldsymbol{\alpha}, \boldsymbol{\omega}$ values. We compute the upper limit for $\boldsymbol{\alpha}$ such that memory will retain $\beta = 0.01 = 1\%$ of its value after some elapsed number of timesteps we call t_e. This can be computed mathematically via

$$\frac{\log \beta}{t_e}. \tag{25}$$

The lower limit is set based on the maximum value a double precision float can represent, minus a small ϵ

$$\frac{\log 1.79 \times 10^{308}}{t_e} - \epsilon. \tag{26}$$

Note that if needed, we can choose a smaller lower bound, but the input must be chunked into sequences of length t_e and run in minibatches. Since we can compute minibatches recurrently, the gradient spans across all minibatches rather than being truncated like a quadratic transformer. Nonetheless, we did not need to do this in any of our experiments. We set $\boldsymbol{\alpha}$ to be linearly spaced between the computed limits

$$\boldsymbol{\alpha} = [\alpha_1, \dots \alpha_j \dots \alpha_n]^\top \tag{27}$$

$$\alpha_j = \frac{j}{m} \frac{\log \beta}{t_e} + (1 - \frac{j}{m}) \left(\frac{\log 1.79 \times 10^{308}}{t_e} - \epsilon \right) \tag{28}$$

We initialize the $\boldsymbol{\omega}$ terms using a similar approach, with linearly spaced denominators between one and t_e

$$\boldsymbol{\omega} = 2\pi / [\omega_1, \omega_2, \dots, \omega_n]^\top \tag{29}$$

$$\omega_j = \frac{j}{c} + (1 - \frac{j}{c}) t_e \tag{30}$$

All other parameters are initialized using the default Pytorch initialization.

C Computing States in Parallel with Linear Space Complexity

Here, we show how the recurrent formulation can be computed in parallel. A naive implementation would use $O(n^2)$ space, but using a trick, we can accomplish this in $O(n)$ space.

Assume we have already computed hidden state $\boldsymbol{S}_{k-1}$ for some sequence. Our job is now to compute the next $n-k+1$ recurrent states $\boldsymbol{S}_k, \boldsymbol{S}_{k+1}, \dots, \boldsymbol{S}_n$ in parallel. We can rewrite Equation 11 in matrix form:

$$\boldsymbol{S}_{k:n} = \begin{bmatrix} \boldsymbol{S}_k \\ \boldsymbol{S}_{k+1} \\ \boldsymbol{S}_{k+2} \\ \vdots \\ \boldsymbol{S}_n \end{bmatrix} = \begin{bmatrix} \boldsymbol{\gamma}^0 \odot (\boldsymbol{x}_k \mathbf{1}_c^\top) + \boldsymbol{\gamma}^1 \odot \boldsymbol{S}_{k-1} \\ \boldsymbol{\gamma}^0 \odot (\boldsymbol{x}_{k+1} \mathbf{1}_c^\top) + \boldsymbol{\gamma}^1 \odot (\boldsymbol{x}_k \mathbf{1}_c^\top) + \boldsymbol{\gamma}^2 \odot \boldsymbol{S}_{k-1} \\ \boldsymbol{\gamma}^0 \odot (\boldsymbol{x}_{k+2} \mathbf{1}_c^\top) + \boldsymbol{\gamma}^1 \odot (\boldsymbol{x}_{k+1} \mathbf{1}_c^\top) + \boldsymbol{\gamma}^2 \odot (\boldsymbol{x}_k \mathbf{1}_c^\top) + \boldsymbol{\gamma}^3 \odot \boldsymbol{S}_{k-1} \\ \vdots \\ \left(\sum_{j=k}^n \boldsymbol{\gamma}^j \odot (\boldsymbol{x}_{k+j} \mathbf{1}_c^\top)\right) + \boldsymbol{\gamma}^{n+1} \odot \boldsymbol{S}_{k-1} \end{bmatrix} \tag{31}$$

We can write the closed form for the pth row of the matrix, where $0 \geq p \geq n-k$

$$\boldsymbol{S}_{k+p} = \left(\sum_{j=0}^{p} \boldsymbol{\gamma}^{p-j} \odot (\boldsymbol{x}_{k+j} \mathbf{1}_c^\top) \right) + \boldsymbol{\gamma}^{p+1} \odot \boldsymbol{S}_{k-1} \tag{32}$$

Unfortunately, it appears we will need to materialize $\frac{(n-k+1)^2}{2}$ terms:

$$\boldsymbol{\gamma}^0 \odot (\boldsymbol{x}_k \mathbf{1}_c^\top) \tag{33}$$
$$\boldsymbol{\gamma}^1 \odot (\boldsymbol{x}_k \mathbf{1}_c^\top), \boldsymbol{\gamma}^0 \odot (\boldsymbol{x}_{k+1} \mathbf{1}_c^\top) \tag{34}$$
$$\vdots \tag{35}$$
$$\boldsymbol{\gamma}^{n-k} \odot (\boldsymbol{x}_k \mathbf{1}_c^\top), \boldsymbol{\gamma}^{n-k+1} \odot (\boldsymbol{x}_{k+1} \mathbf{1}_c^\top), \ldots \boldsymbol{\gamma}^0 \odot (\boldsymbol{x}_n \mathbf{1}_c^\top) \tag{36}$$

However, we can factor out $\boldsymbol{\gamma}^{-p}$ from Equation 32 via a combination of the distributive property and the product of exponentials

$$\boldsymbol{S}_{k+p} = \gamma^p \odot \left(\sum_{j=0}^{p} \boldsymbol{\gamma}^{-j} \odot (\boldsymbol{x}_{k+j} \mathbf{1}_c^\top) \right) + \boldsymbol{\gamma}^{p+1} \odot \boldsymbol{S}_{k-1} \tag{37}$$

Now, each $\boldsymbol{\gamma}^j$ is associated with a single $\boldsymbol{x}_j$, requiring just $n-k+1$ terms:

$$\boldsymbol{\gamma}^0 \odot (\boldsymbol{x}_k \mathbf{1}_c^\top) \tag{38}$$
$$\boldsymbol{\gamma}^0 \odot (\boldsymbol{x}_k \mathbf{1}_c^\top), \boldsymbol{\gamma}^1 \odot (\boldsymbol{x}_{k+1} \mathbf{1}_c^\top) \tag{39}$$
$$\vdots \tag{40}$$
$$\boldsymbol{\gamma}^0 \odot (\boldsymbol{x}_k \mathbf{1}_c^\top), \boldsymbol{\gamma}^1 \odot (\boldsymbol{x}_{k+1} \mathbf{1}_c^\top), \ldots, \boldsymbol{\gamma}^n \odot (\boldsymbol{x}_n \mathbf{1}_c^\top) \tag{41}$$

We can represent each of these rows as a slice of a single $n-k+1$ length tensor, for a space complexity of $O(n-k)$ or for $k=1$, $O(n)$.

Finally, we want to swap the exponent signs because it provides better precision when working with floating point numbers. Computing small values, then big values is more numerically stable than the other way around. We want to compute the inner sum using small numbers ($\boldsymbol{\gamma}^+$ results in small numbers while $\boldsymbol{\gamma}^-$ produces big numbers; $n-k$ is positive and $k-n$ is negative). We can factor $\boldsymbol{\gamma}^{k-n}$ out of the sum and rewrite Equation 37 as

$$\boldsymbol{S}_{k+p} = \gamma^{k-n+p} \odot \left(\sum_{j=0}^{p} \boldsymbol{\gamma}^{n-k-j} \odot (\boldsymbol{x}_{k+j} \mathbf{1}_c^\top) \right) + \boldsymbol{\gamma}^{p+1} \odot \boldsymbol{S}_{k-1} \tag{42}$$

If we let $t = n-k$, this is equivalent to Equation 13:

$$\boldsymbol{S}_{k+p} = \boldsymbol{\gamma}^{p+1} \odot \boldsymbol{S}_{k-1} + \boldsymbol{\gamma}^{p-t} \odot \sum_{j=0}^{p} \boldsymbol{\gamma}^{t-j} \odot (\boldsymbol{x}_{k+j} \mathbf{1}_c^\top), \quad \boldsymbol{S}_{k+p} \in \mathbb{C}^{m \times c}. \tag{43}$$

We also need to compute $2(n-k+1)$ gamma terms: $\boldsymbol{\gamma}^{k-n}, \ldots \boldsymbol{\gamma}^{n-k+1}$, resulting in linear space complexity $O(n)$ for a sequence of length n.

D Benchmark Details

We utilize the episodic reward and normalized episodic reward metrics. We record episodic rewards as a single mean for each epoch, and report the maximum reward over a trial.

We do not modify the hyperparameters for PPO, SAC, or TD3 from the original benchmark papers. We direct the reader to [Morad et al., 2023] for PPO hyperparameters and [Ni et al., 2022] for SAC and TD3 hyperparameters.

D.1 Hardware and Efficiency Information

We trained on a server with a Xeon CPU running Torch version 1.13 with CUDA version 11.7, with consistent access to two 2080Ti GPUs. Wandb reports that we used roughly 161 GPU days of compute to produce the POMDP-Baselines results, 10 GPU days for the POPGym results, and 45 GPU days for the FFM ablation using POPGym. This results in a total of 216 GPU days. Since we ran four jobs per GPU, this corresponds to 54 days of wall-clock time for all experiments.

D.2 PPO and POPGym Baselines

Morad et al. [2023] compares models along the recurrent state size, with a recurrent state size of 256. For fairness, We let $m = 32$ and $c = 4$, which results in a complex recurrent state of 128, which can be represented as a 256 dimensional real vector. We initialize $\boldsymbol{\alpha}, \boldsymbol{\omega}$ following Appendix B for $t_e = 1024, \beta = 0.01$. We run version 0.0.2 of POPGym, and compare FFM numerically with the MMER score from the paper in Table 3.

D.3 SAC, TD3, and POMDP-Baselines

[Ni et al., 2022] does not compare along the recurrent state size, but rather the hidden size, while utilizing various hidden sizes h. In other words, the LSTM recurrent size is twice that of the GRU. We let $c = h/32$ and $m = h/c$, so that $mc = h$. This produces an equivalent FFM configuration to the POPGym baseline when $h = 128$, with equivalent recurrent size in bytes to the LSTM (or equivalent in dimensions to the GRU). The paper truncates episodes into segments of length 32 or 64 depending on the task, so we let $t_e = 128$ to ensure that information can persist between segments. Thanks to the determinism provided in the paper, readers should be able to reproduce our exact results using the random seeds $0, 1, 2, 3, 4$. We utilize separate memory modules for the actor and critic, as done in the paper. We base our experiments off of the most recent commit at the time of writing, 4d9cbf1.

Model	MMER
MLP	0.067
PosMLP	0.064
FWP	0.112
FART	0.138
S4D	-0.180
TCN	0.233
Fr.Stack	0.190
LMU	0.229
IndRNN	0.259
Elman	0.249
GRU	0.349
LSTM	0.255
DNC	0.065
FFM	**0.400**

Table 3: MMER score comparison from the POPGym paper

E Benchmark Lineplots

We provide cumulative max reward lineplots in Figure 9, Figure 7, and Figure 8 for all the experiments we ran.

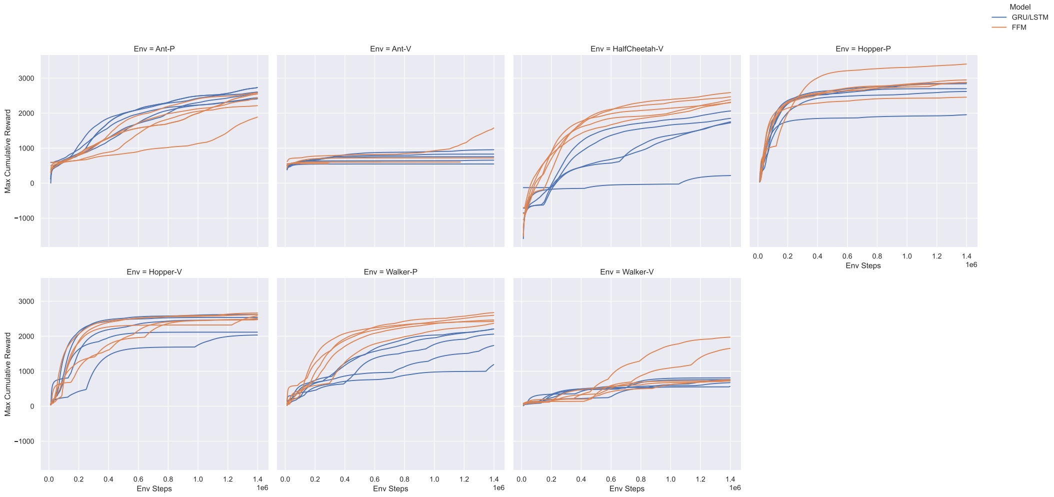

Figure 7: SAC lineplots for POMDP-Baselines, where each trial is plotted separately.

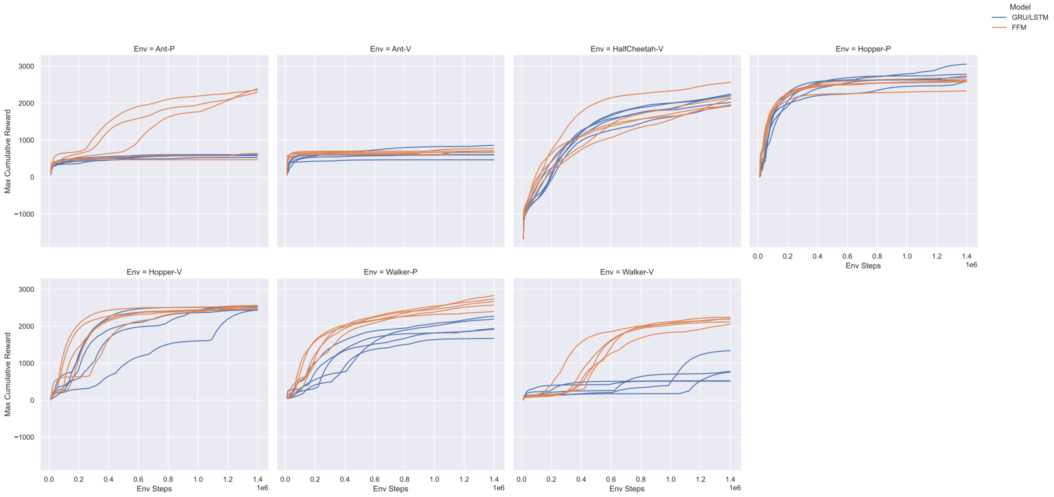

Figure 8: TD3 lineplots for POMDP-Baselines, where each trial is plotted separately.

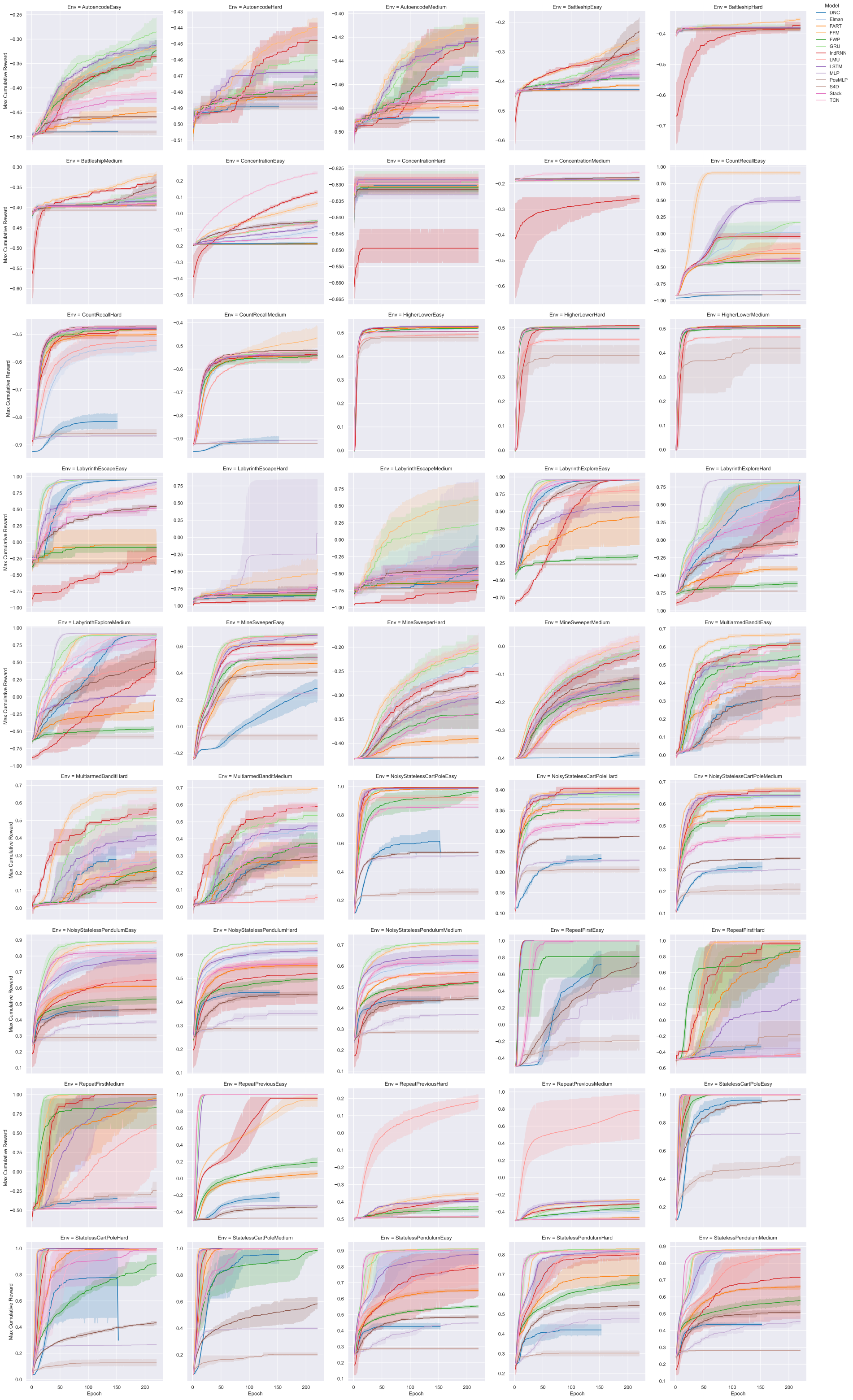

Figure 9: Lineplots for POPGym, where the shaded region represents the 95% bootstrapped confidence interval.

F POPGym Comparisons by Model

We provide Figure 10, Figure 11, Figure 12 showing the relative FFM return compared to the other 12 POPGym models, including temporal convolution, linear transformers, and more.

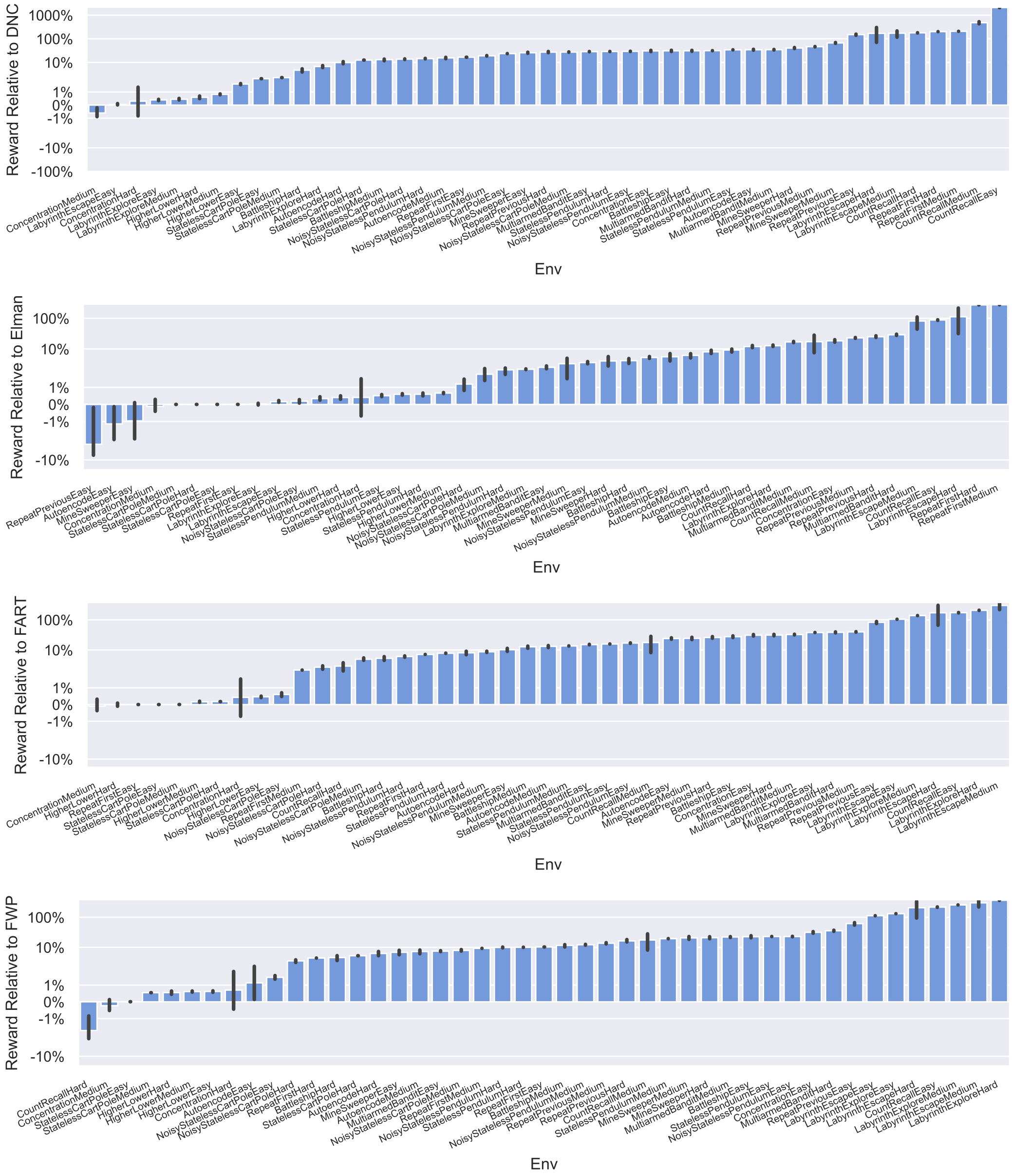

Figure 10: Relative POPGym returns compared to FFM.

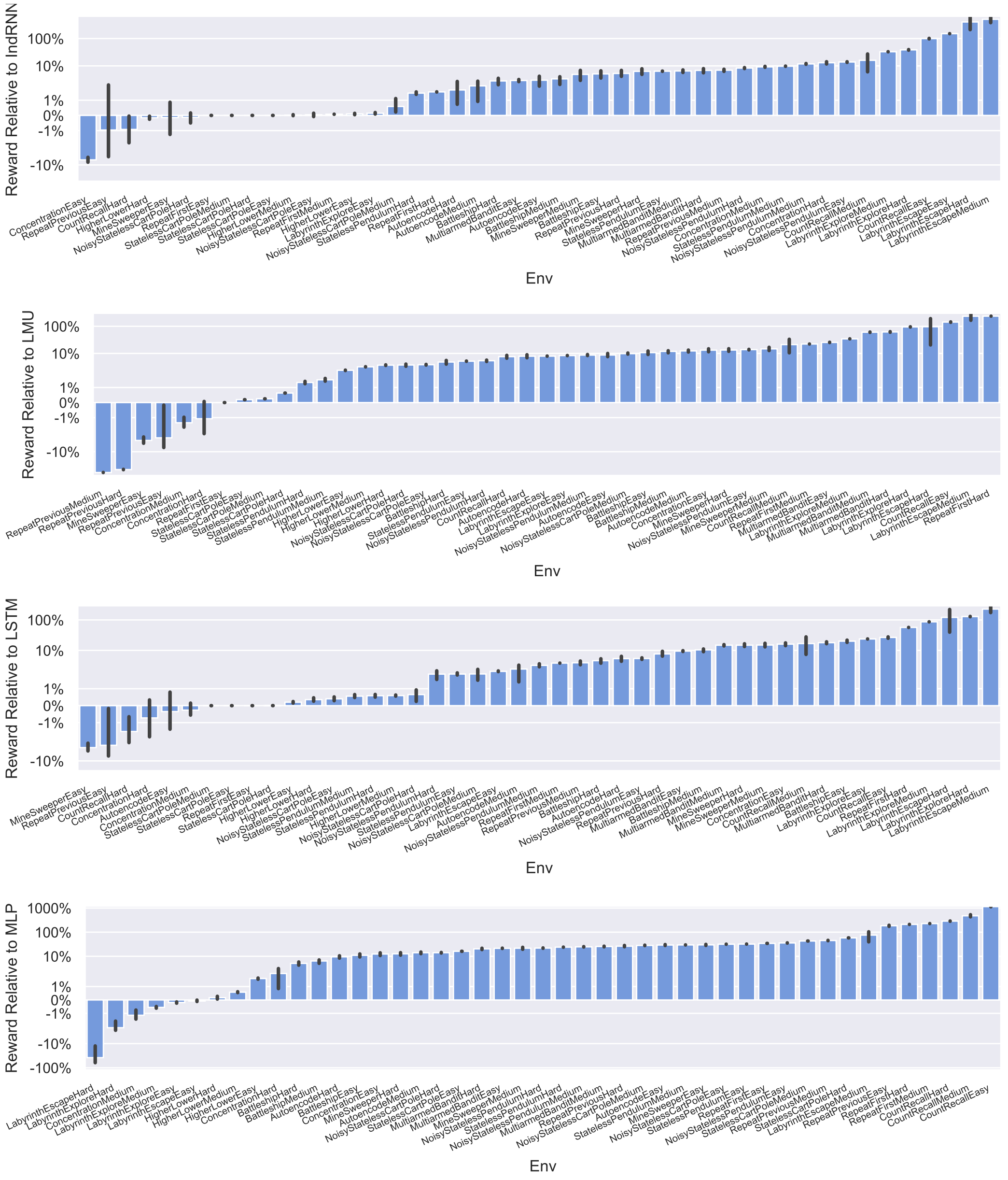

Figure 11: Relative POPGym returns compared to FFM.

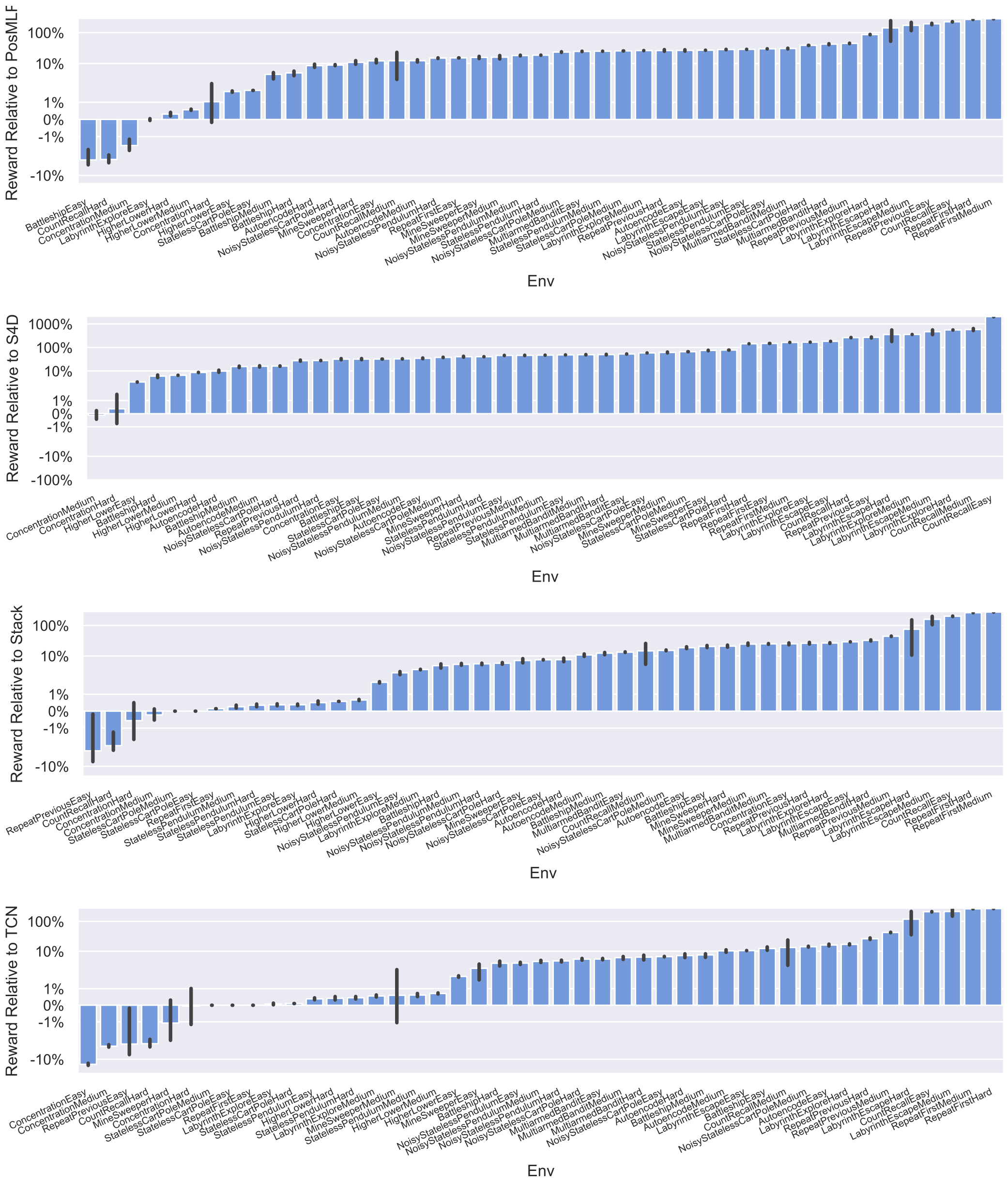

Figure 12: Relative POPGym returns compared to FFM.

G Efficiency Statistic Details

This section explains how we computed the bar plots showing efficiency statistics. We construct the POPGym models and perform one PPO training epoch. The train time metric corresponds to the time spent doing forward and backward passes over the data. One epoch corresponds to the epoch defined in POPGym: 30 minibatches of 65,336 transitions each, with an episode length of 1024, a hidden dimension of 128, and a recurrent dimension of 256. We do this 10 times and compute the mean and confidence interval. Torch GRU, LSTM, and Elman networks have specialized CUDA kernels, making them artificially faster than LMU and IndRNN which are written in Torch and require the use of for loops. We utilize the pure python implementation of these models, wrapping them in a for loop instead of utilizing their hand-designed CUDA kernels. We consider this a fair comparison since this makes the GRU, Elman, and LSTM networks still run slightly faster than IndRNN and LMU (Torch-native RNNs). FFM is also written in Torch and does not have access to specialized CUDA kernels. CUDA programmers more skilled than us could implement a custom FFM kernel that would see a speedup similar to the GRU/LSTM/Elman kernels.

To compute inference latency, we turn off Autograd and run inference for 1024 timesteps, computing the time for each forward pass. We do this 10 times and compute the mean and 95% confidence interval.

To compute memory usage, we utilize Torch's GPU memory tools and record the maximum memory usage at any point during the training statistic. Memory usage is constant between trials.

For reward, we take the mean reward over all environments, split by model and trial. We then report the mean and 95% confidence interval over trials and models.