

transformerXL__PPO__JAX

This repository provides a JAX implementation of TransformerXL with PPO in a RL setup following : “Stabilizing Transformers for Reinforcement Learning” from Parisotto et al. (<https://arxiv.org/abs/1910.06764>).

The code uses the PureJaxRL template for PPO and copied some of the code from Huggingface transformerXL transferring it to JAX. We also took inspiration from the pytorch code in <https://github.com/MarcoMeter/episodic-transformer-memory-ppo>, which has some simplification of gradient propagation and positional encoding compared to transformerXL as it is described in the original paper (<https://arxiv.org/abs/1901.02860>).

The training handles Gymnax environment.

We also tested it on Craftax, on which it beat the baseline presented in the paper (<https://arxiv.org/abs/2402.16801>) including PPO-RNN, training with unsupervised environment design and intrinsic motivation. Notably we reach the 3rd level (the sewer) and obtain several advanced advancements, which was not achieved by the methods presented in the paper. See Craftax Results for more informations.

The training of a 5M transformer on craftax for 1e9 steps (with 1024 environments) takes about 6h30 on a single A100.

Installation

```
git clone git@github.com:Reytuag/transformerXL_PPO_JAX.git
cd transformerXL_PPO_JAX
pip install -r requirements.txt
```

:warning: By default, this will install the cpu version of JAX. You can install the GPU version of JAX following <https://jax.readthedocs.io/en/latest/installation.html>.

Training

You can edit the training config in `train_PPO_trXL.py` (or `train_PPOTrXL_pmap.py` if you want to go multi GPU) including the name of the environment. (you can put any gymtax environment name, or “craftax” which will use the CraftaxSymbolic env)

To launch the training:

```
python3 train_PPO_trXL.py
```

Or if you go multi GPU.(it will use all your GPU)

```
python3 train_PPO_trXL_pmap.py
```



Figure 1: enter_sewerb

Results on Craftax

Without much parameter search, with a budget of $1e9$ timesteps, the normalized return (% max) achieve 18.3% compared to 15.3% for PPO-RNN according to the craftax paper. (with one seed visiting the sewer). The config used can be found as the default config in `train_PPO_trXL_pmap.py`. The results can be found in `results_craftax` (and can be loaded with `jnp.load(str(seed)+"_metrics.npy",allow_pickle=True).item())` as well as the trained parameters.

Here are the achievements success rates across training for $1e9$ steps. Notably “enter the gnomish mine” is much higher than what is reported in the craftax paper, even PPO-RNN trained on $10e9$ steps so 10 times more ends up not visiting the gnomish mines while one seed luckily visit the level after the gnomish mine: the sewer.

With a budget of $4e9$ timesteps, the normalized return is 20.6 %. Visiting the 3rd floor (the sewer) a decent amount of time and achieve several advanced achievements. Both visiting the 3rd floor and reaching any advanced achievement were not reached by any of the baseline in the craftax paper even PPO-RNN with $10e9$ interactions with the environment.

Here are the achievements success rates across training for $4e9$ steps:

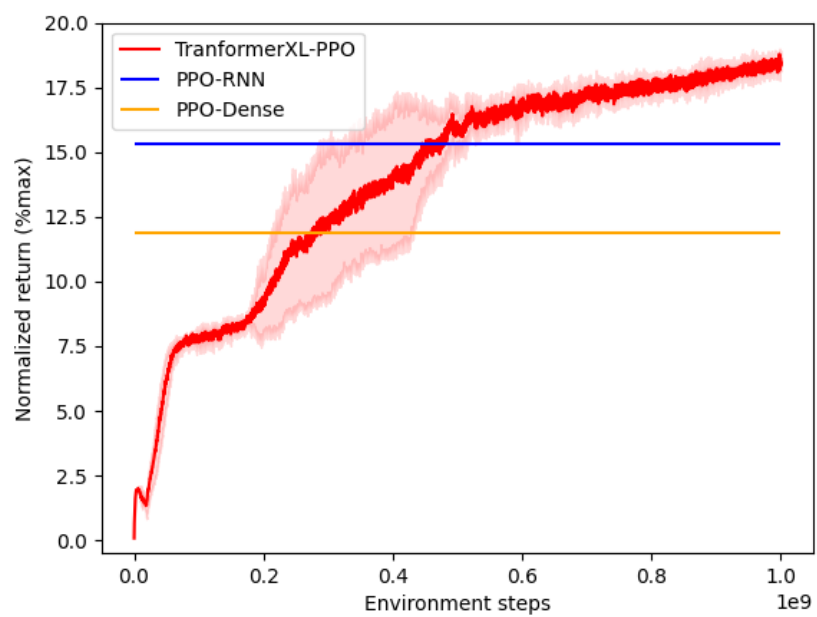


Figure 2: craftax_training_transfoXL_PPO

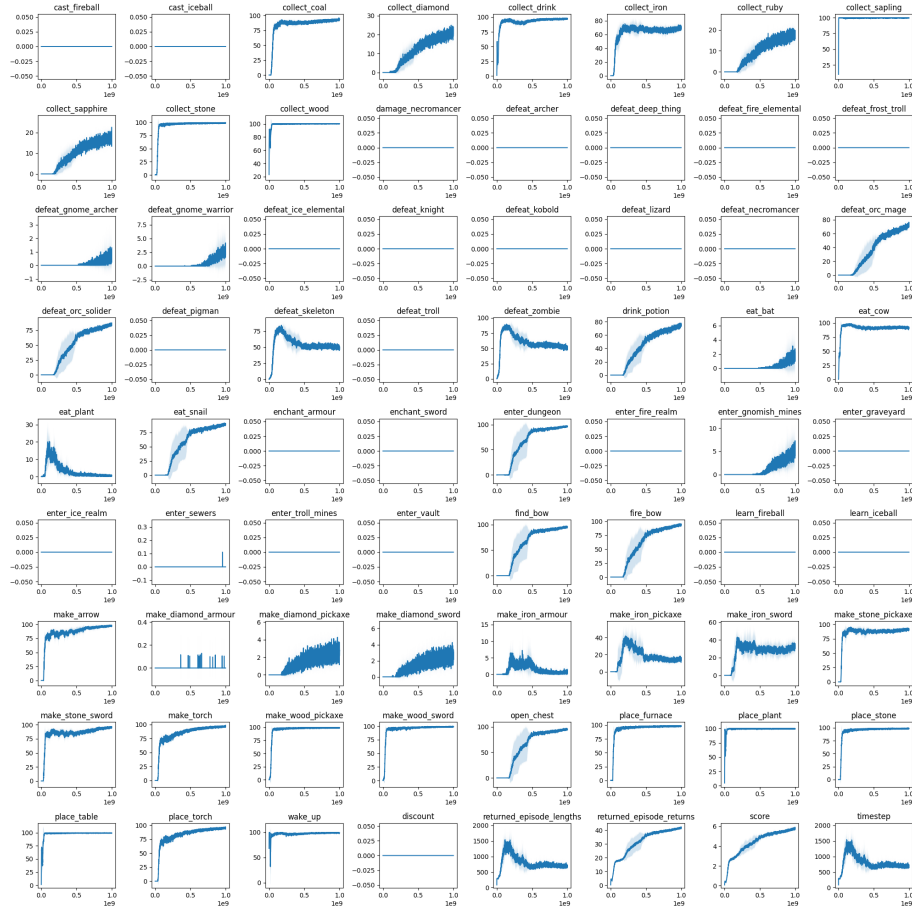


Figure 3: craftax_achievements_1e9steps

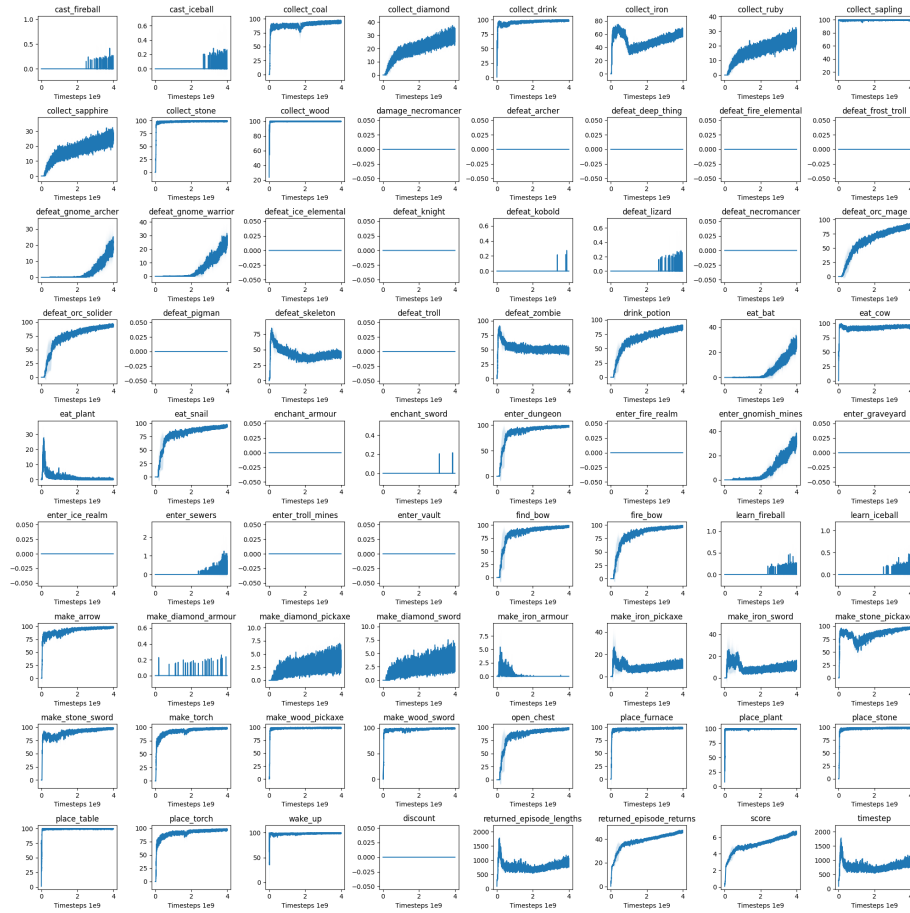


Figure 4: craftax_achievements_4e9steps

However training for 8e9 steps did not lead to significant improvement. Though we did not conducted much hyperparameters tuning.

Config parameters :

- `TOTAL_TIMESTEPS` : Total number of environment steps during training.
- `NUM_STEPS` : Number of steps between updates. The training data for each update will thus contain `NUM_STEPS*NUM_ENVS` total steps.
- `WINDOW_MEM` : Size of the memory. (the last step will attend to `WINDOW_MEM` previous steps in addition to itself)
- `WINDOW_GRAD` : At training, size of the context window where transformer embeddings are computed again (not using cached one) and thus where gradient flow maximally. (in the base transformerXL paper <https://arxiv.org/abs/1901.02860> , `WINDOW_GRAD=WINDOW_MEM`, but in “Human-Timescale Adaptation in an Open-Ended Task Space” (<https://arxiv.org/abs/2301.07608>), they seem to use different values (a memory size of 300, and a “rollout context window” so a `WINDOW_GRAD` of 80).

:warning: `WINDOW_GRAD` must divide `NUM_STEPS`. * `num_layers` : Number of transformer layers * `EMBED_SIZE` : Size of the embeddings in the transformer * `num_heads` : Number of attention heads at each transformer layer * `qkv_features` : Size of the query,key,value vectors will be `qkv_features//num_heads`

For continuous action space, you can follow PurejaxRL example and replace the categorical distrib in the actor network with “`pi = distrax.MultivariateNormalDiag(actor_mean, jnp.exp(actor_logtstd))`”

Related Works

- Gymnax: <https://github.com/RobertTLange/gymnax>
- Craftax: <https://github.com/MichaelTMatthews/Craftax>
- Xland-Minigrid: <https://github.com/corl-team/xland-minigrid>
- PureJaxRL: <https://github.com/luchris429/purejaxrl>
- JaxMARL: <https://github.com/FLAIROx/JaxMARL>
- Jumanji: <https://github.com/instadeepai/jumanji>
- Evojax: <https://github.com/google/evojax>
- Evosax: <https://github.com/RobertTLange/evosax>
- Brax: <https://github.com/google/brax>

Next steps

- Train it on XLand-MiniGrid (<https://github.com/corl-team/xland-minigrid>) to test it on an open-ended environment in a meta-RL fashion.

- Add an implementation of Muesli (<https://arxiv.org/abs/2104.06159>) with transformerXL as in “Human-Timescale Adaptation in an Open-Ended Task Space” (<https://arxiv.org/abs/2301.07608>)