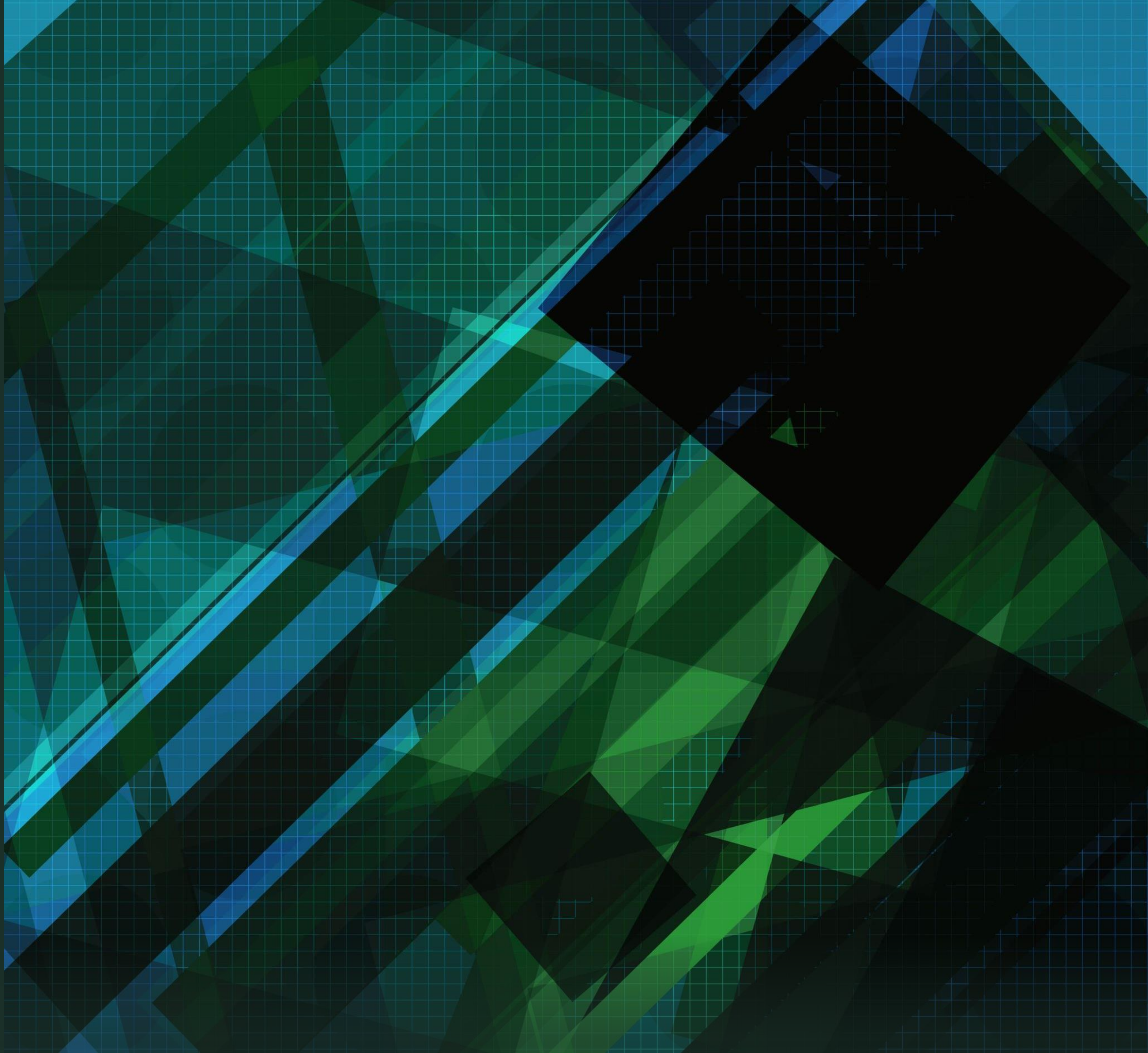
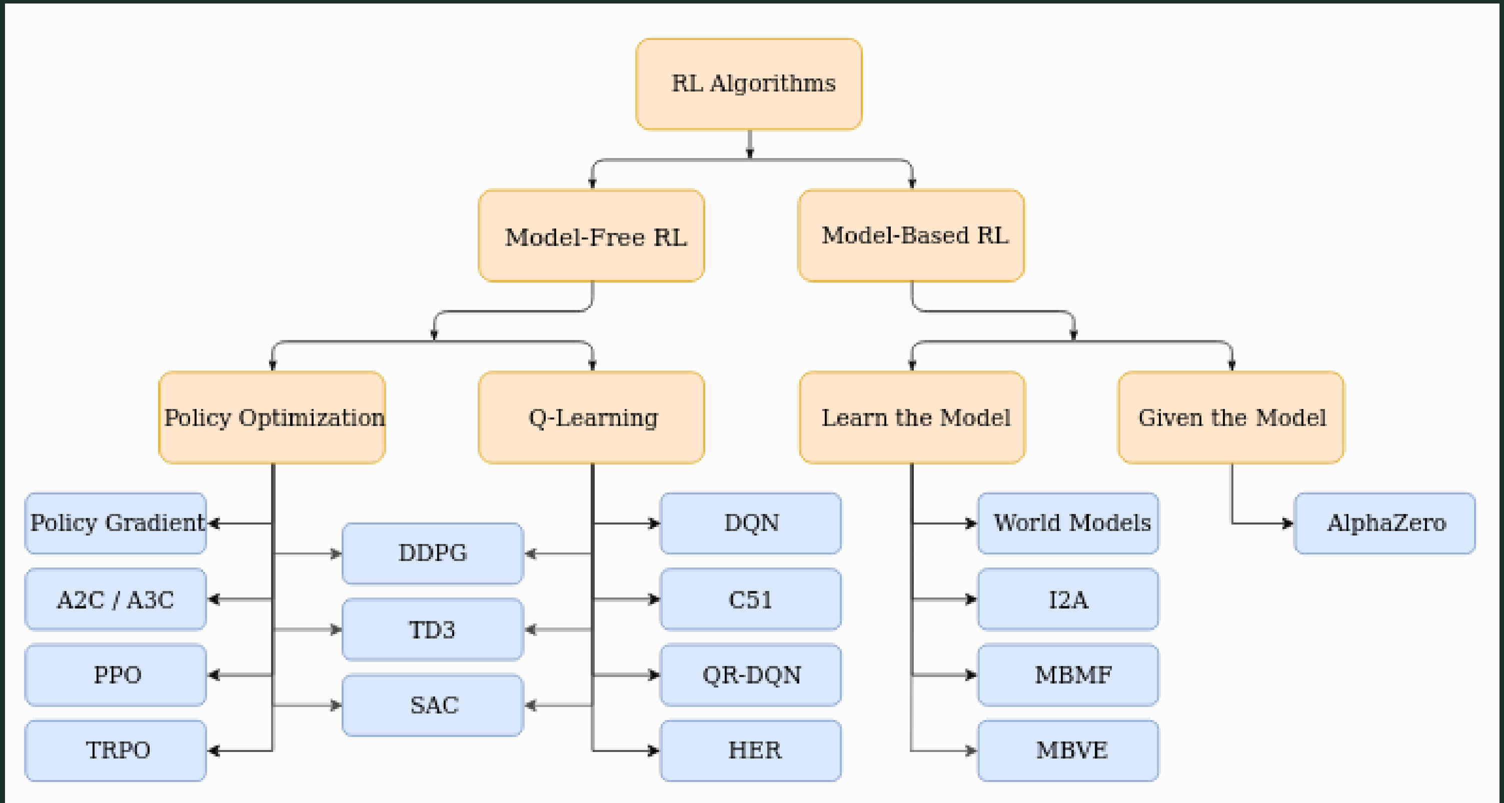
RLGraph + PPG



Model-free RL

- Policy network:
  select the action to take
- Value network:
  predict the expected reward in current state



Source: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html. MIT-licensed, 2018.

Policy optimization

REINFORCE (Sutton et al.) (1999) → A2C/A3C (Mnih et al.) (2016) → TRPO (Schulman et al.) (2015) → PPO (Schulman et al.) (2017) → PPG (Cobbe et al.) (2020)

REINFORCE (1999)

- AKA Vanilla Policy Gradient
- Notable success: 2013 Atari
- Standard gradient ascent
- But small parameter changes might still harm performance
- A. Karpathy implemented it in 130 lines in numpy

Algorithm 1 Vanilla Policy Gradient Algorithm

Input: initial policy parameters θ_0, initial value function parameters ϕ_0

for $k = 0, 1, 2, ...$ **do**

Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

Compute rewards-to-go $\hat{R}_t$.

Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$

or via another gradient ascent algorithm like Adam.

Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2,$$

typically via some gradient descent algorithm.

end for

Source: https://spinningup.openai.com/en/latest/algorithms/vpg.html

PPO (2017)

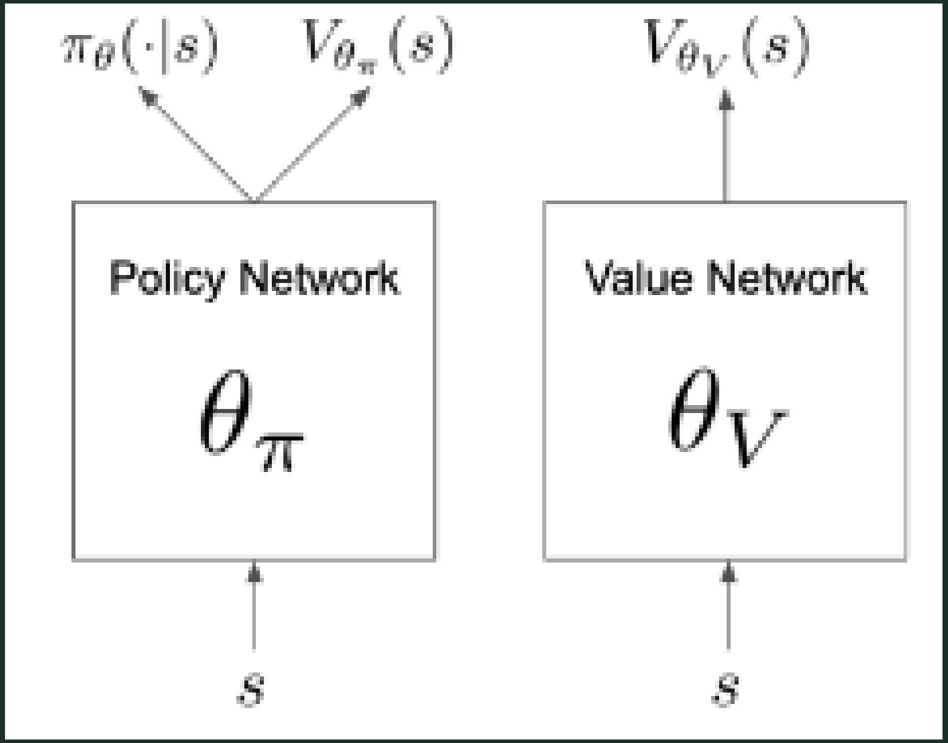
- Idea: disincentivise large changes in one step of policy improvement.
- One network approximates the policy and the value function (but this is not key to PPO)
- Updates are standard. In each step, loss looks at the old and new probability
- If the actual update is too big, loss treats it as if it changed only by ε*100%. Typically, ε = 0.2

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right)$$

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0. \end{cases}$$

PPG (2020)

- Policy network:
  - Same as that used in PPO
  - Optimizing the *clipped surrogate function* with an entropy bonus
  - Has two heads: a policy head and an auxilliary value head
  - Parameters shared
- Value network:
  - Predicts the value given a state
  - Parameters distinct



Source: PPG paper

Training of PPG

- Policy phase:
  - Estimate advantage function (GAE)
  - Optimize the policy for clipped loss
  - Optimize the value network w/ MSE
- Auxiliary phase:
  - Auxiliary loss: MSE on target values.
  - Joint loss = auxiliary loss + behavioural cloning (keeps policy).
  - Optimize these two losses.

Algorithm 1 PPG

```
for phase = 1, 2, ... do
    Initialize empty buffer B
    for iteration = 1, 2, ..., N_π do                          ▷ Policy Phase
        Perform rollouts under current policy π
        Compute value function target V̂_t^targ for each state s_t
        for epoch = 1, 2, ..., E_π do                          ▷ Policy Epochs
            Optimize L^clip + β_S S[π] wrt θ_π
        for epoch = 1, 2, ..., E_V do                          ▷ Value Epochs
            Optimize L^value wrt θ_V
        Add all (s_t, V̂_t^targ) to B
    Compute and store current policy π_θold(·|s_t) for all states s_t in B
    for epoch = 1, 2, ..., E_aux do                            ▷ Auxiliary Phase
        Optimize L^joint wrt θ_π, on all data in B
        Optimize L^value wrt θ_V, on all data in B
```

$$L^{value} = \hat{\mathbb{E}}_t\left[\frac{1}{2}(V_{\theta_V}(s_t) - \hat{V}_t^{\text{targ}})^2\right] \quad L^{aux} = \frac{1}{2}\cdot\hat{\mathbb{E}}_t\left[(V_{\theta_\pi}(s_t) - \hat{V}_t^{\text{targ}})^2\right] \quad L^{joint} = L^{aux} + \beta_{clone}\cdot\hat{\mathbb{E}}_t\left[KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]\right]$$

Results

- Tested on Procgen (which is an improved variation of Atari)
- Converges faster and learns better than PPO

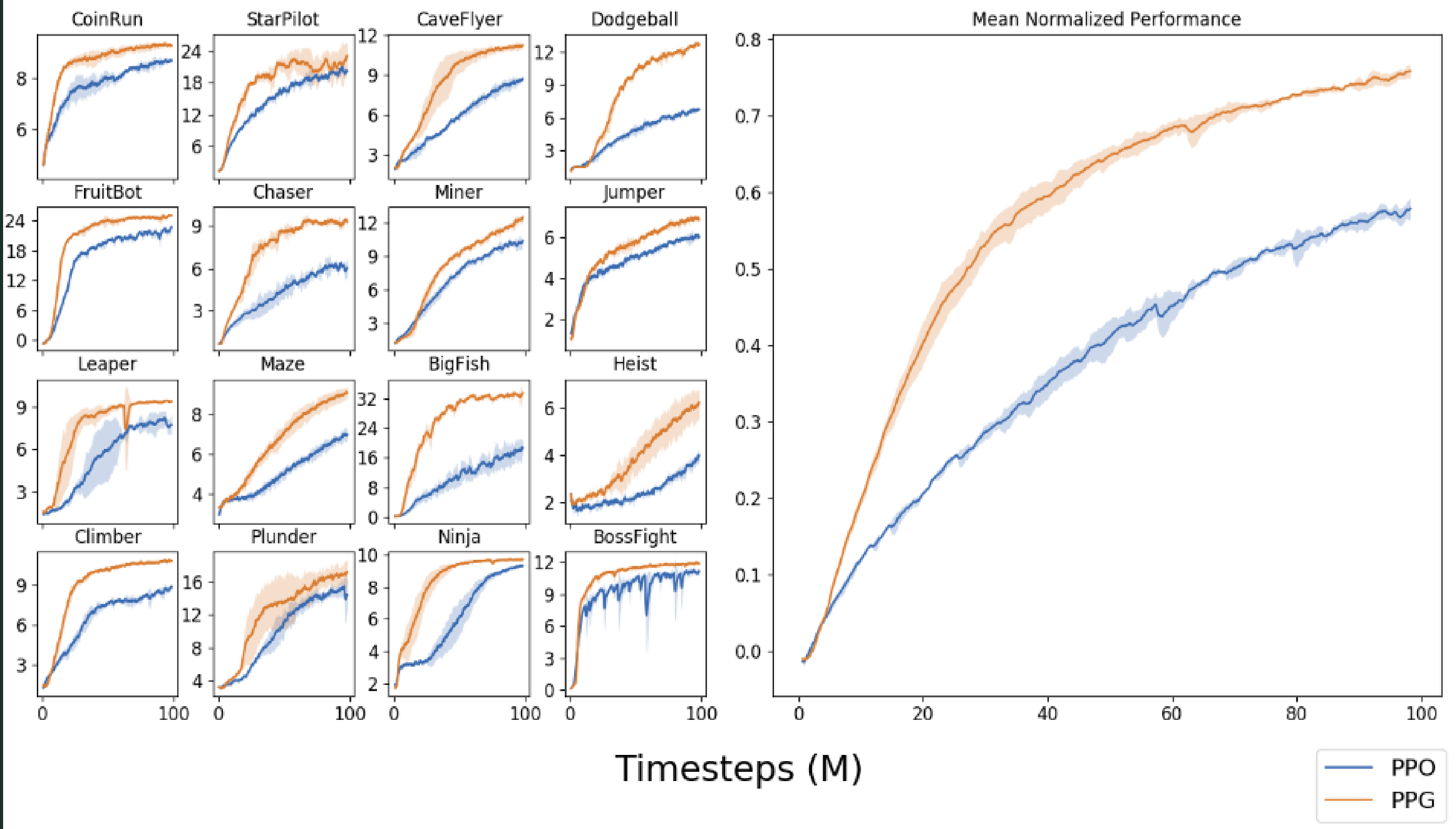
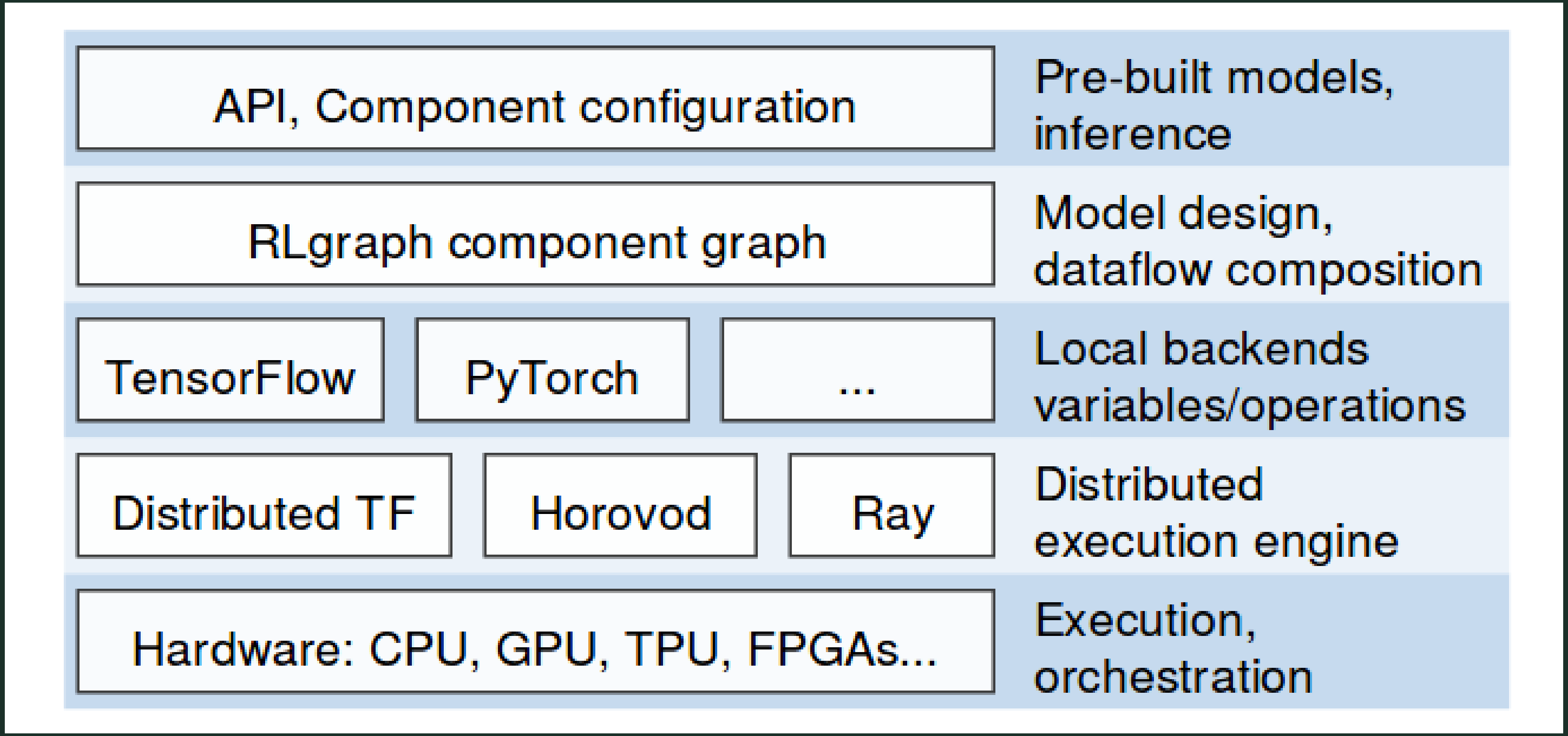


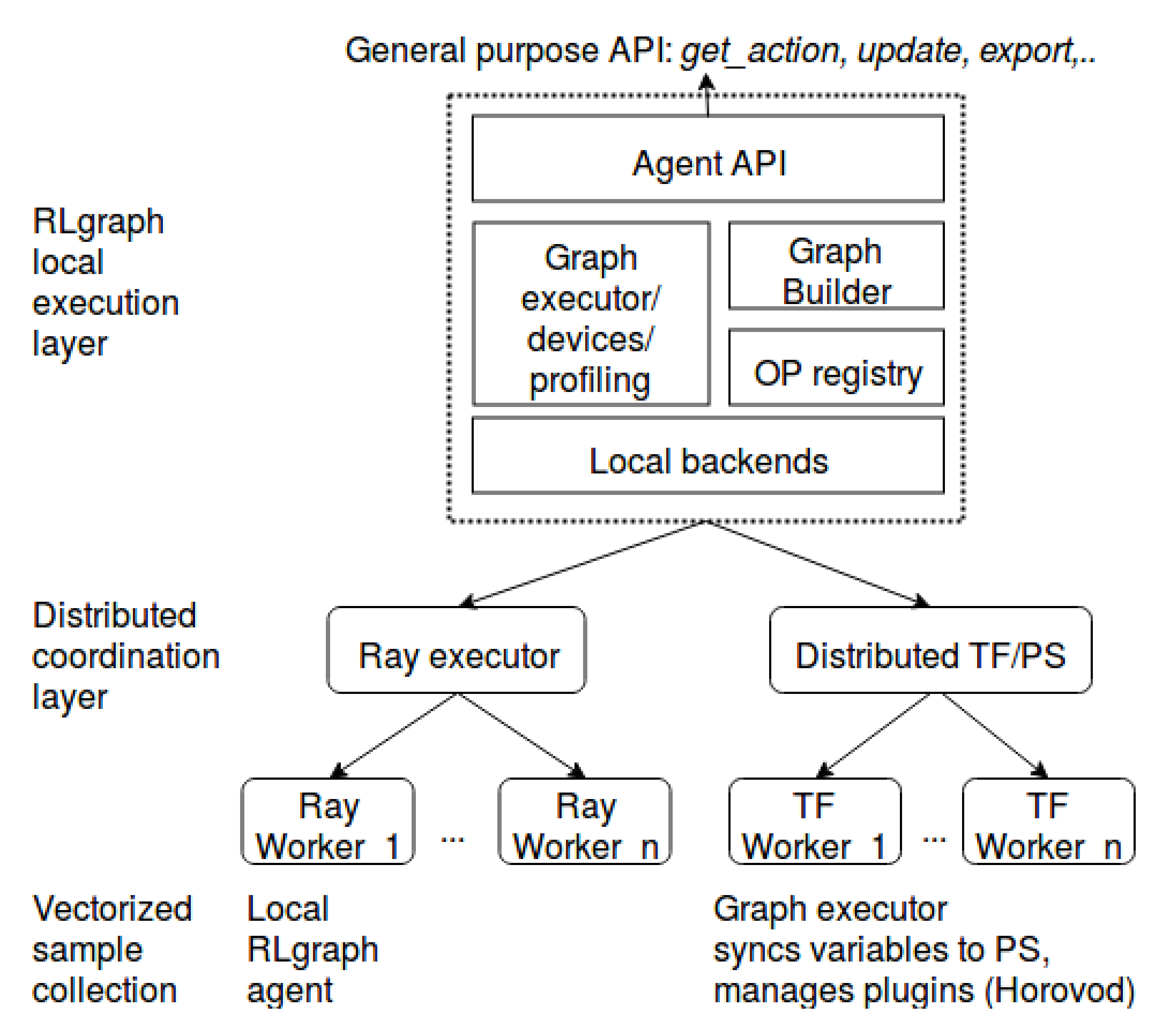
Figure 2: Sample efficiency of PPG compared to a PPO baseline

RLGraph



Source: https://rlgraph.github.io/rlgraph/2019/01/04/introducing-rlgraph.html

RLGraph



Source: https://rlgraph.github.io/rlgraph/2019/01/04/introducing-rlgraph.html

Goals

- Reimplement the PPG in RLGraph
- Benchmark performance of PPG in RLGraph on Atari and Gym scenarios