Diego Toribio
Professor Curro
ECE471 - Reinforcement Learning
Final Project

Methods

Our implementation is built on top of a basic Proximal Policy Optimization (PPO) framework, originally designed to train agents in the game of Checkers. In this section, we describe the additional features and modifications integrated into the baseline PPO algorithm.

- Orthogonal Initialization of Neural Network Parameters
  - We apply orthogonal initialization to all weight layers and set biases to zero. The policy and value network layers are initialized such that weights are orthogonal and final output layers (policy and value heads) have appropriately scaled weights. Orthogonal initialization helps improve the flow of gradients and stabilizes learning, especially early in training.
  - Implemented in the layer_init function, applied to both actor and critic networks.
- Adam Optimizer with Custom Epsilon Parameter
  - PPO commonly relies on the Adam optimizer, but we explicitly set the epsilon parameter (e.g., eps=1e-5) to ensure numerical stability and reduce gradient noise. While not a major visible change, this subtle adjustment can impact the reliability of updates.
  - In the optimizer creation step for training the agent.
- Generalized Advantage Estimation (GAE)
  - We use GAE to estimate advantages more smoothly compared to simple N-step returns. GAE balances bias and variance in advantage estimates, stabilizing training and speeding convergence. It makes the training process less sensitive to long-horizon credit assignment issues.
  - Implemented In the trainer function, when computing advantages and returns after each rollout.
- Mini-Batch Updates per PPO Epoch
  - Instead of using the entire batch of collected data at once, the data is shuffled and split into smaller mini-batches. This stabilizes updates, reduces memory overhead, and prevents catastrophic updates to the policy. This approach follows standard PPO practice and is known to yield more stable training dynamics.
  - Implemented within each PPO update loop, the data is divided into mini-batches before computing gradients.
- Normalization of Advantages
  - Before updating the policy network, we normalize the estimated advantages. Normalization keeps the scale of advantages well-conditioned and consistent across training iterations, ensuring that the policy update step size remains appropriate. This improves numerical stability and can speed up training.

    - Implemented Inside the update loop, right before computing the loss.
- Clipped Surrogate Objective
    - The clipped objective is the core of PPO. We ensure that the policy update does not deviate too far from the old policy by clipping the probability ratio. This helps avoid destructive policy updates and contributes to PPO's well-known stability and good performance.
    - Implemented in the PPO loss calculation, when comparing the new policy probability ratio with a clipped threshold.
- Value Function Loss Clipping
    - Similar to the policy clipping, the value predictions are also clipped to avoid large, destabilizing updates. Although research suggests this may not always improve performance, we maintain value clipping to remain consistent with the standard PPO implementation details.
    - Implemented in the loss function, when computing the value loss term.

- Combined Objective with Entropy Bonus
    - The total PPO loss includes policy loss, value loss, and an entropy term. The entropy bonus encourages exploration by preventing the policy from collapsing to deterministic behavior too early. This ensures the agent continues to try potentially beneficial actions.
    - Implemented in the final loss calculation, adding entropy to encourage exploration.
- Global Gradient Clipping
    - We clip gradients by their global norm to prevent exploding gradients. This safeguards against instability and ensures the optimization steps remain controlled.
    - After computing gradients but before the optimizer step.
- Logging and Debugging Variables
    - We track metrics such as policy loss, value loss, entropy, and KL divergence at every update. This provides insight into training progress and potential issues. Debugging metrics help us understand policy behavior changes over time and serve as useful diagnostics for ourselves and our supervisor.
    - Metrics are logged after each update via both TensorBoard and Weights & Biases.
- Distinct Actor and Critic Networks (Separate MLPs)
    - Initially, we tested architectures where the policy and value functions share parameters. We now often employ separate MLP networks for policy and value estimation to avoid interference between the two objectives. While sharing parameters can sometimes be efficient, separate networks often improve stability and performance in more complex tasks.
    - Implemented within the Agent class definition, we provide separate heads or even separate networks entirely depending on the chosen configuration.

Results

Cart Pole

To evaluate the Proximal Policy Optimization (PPO) implementation, we selected the CartPole-v1 environment as a benchmark. Achieving an episodic return of 400 or higher in this environment is widely regarded as an indicator of a well-functioning PPO algorithm. Based on the provided visualizations and our experimentation, we confirmed that the PPO implementation performs as expected. The episodic return chart ("charts/episodic_return") demonstrates a steep increase in return values during the initial training phase, followed by stabilization near the environment's maximum return. This progression highlights the implementation’s ability to efficiently learn an optimal policy for CartPole.

The entropy loss ("entropy_loss") decreases steadily throughout training, reflecting a reduction in exploration as the policy becomes more deterministic. This behavior aligns with standard PPO dynamics, where the agent initially explores broadly before refining its actions to maximize rewards. Such a trend is essential for effective learning and convergence.
The value loss ("value_loss") shows a gradual downward trend with occasional spikes, which are expected due to the variance in value function updates in reinforcement learning. These spikes do not hinder the overall minimization of prediction error, as indicated by the steady decline over time. Similarly, the policy loss ("policy_loss") remains relatively stable with minor fluctuations, illustrating the algorithm’s ability to balance policy updates while adhering to clipping constraints.

The total loss ("total_loss"), which incorporates policy, value, and entropy losses, decreases consistently over the course of training. This overall reduction signifies successful optimization of the PPO objective. Most notably, the episodic return consistently surpasses the 400 threshold, serving as a primary validation of the implementation’s correctness. Coupled with trends in the other metrics, these results confirm that the PPO implementation is working as intended.

With these observations, we conclude that the PPO implementation for CartPole-v1 is both correct and efficient, meeting the expected performance standards. The results validate the efficacy of the selected hyperparameters, as outlined in the provided config.yaml file. Having established the implementation’s performance on CartPole, we now turn to evaluating its application to the Tic-Tac-Toe environment.

Tic-Tac-Toe

Checkpoint	Agent Goes First	Agent Wins	Agent Loses	Draws
Random Agent	True	597	268	135
Easy	True	680	225	95
	False	290	636	74
Medium	True	775	178	47
	False	439	543	18

The performance of the Tic Tac Toe agent improves significantly with training as evidenced by the results across the "Easy" and "Medium" checkpoints. When comparing the agent's performance against a random opponent, notable trends emerge. Initially, the random agents produce a moderate number of draws, reflecting the inherently stochastic and untrained nature of their strategies.

At the "Easy" checkpoint, the agent begins to demonstrate a better understanding of the game. When the agent goes first, it achieves a higher win rate and reduced losses compared to the random baseline, while draws decrease, suggesting that the agent has learned to exploit its advantage of the first move. However, when the agent does not go first, its performance declines, with a significant increase in losses and a reduction in draws, indicating the agent's difficulty in countering strategies initiated by the opponent.

The "Medium" checkpoint demonstrates further improvement. The agent's win rate when going first increases markedly, while losses and draws drop further, reflecting a more sophisticated exploitation of its positional advantage. When the agent goes second, the win rate improves over the "Easy" checkpoint, and the number of draws diminishes, but losses remain relatively high. This suggests that while the agent has developed stronger defensive and offensive strategies, it still struggles to fully neutralize the opponent's initial move advantage.

These trends highlight the progression of learning through self-play, with the agent effectively leveraging its positional advantage when going first and gradually improving its counter strategies when going second. Further training on larger checkpoints may provide insights into whether these trends continue or plateau.

Checkers

Evaluation Scenario	Agent Wins	Random Agent Wins	Draws	Avg. Moves Per Game
Random Agent vs. Random Agent	502	498	0	83.20
Trained Agent (1000 games) vs. Random Agent	554	446	0	162.75
Trained Agent (2000 games) Random Agent	548	435	0	162.30
Trained Agent (5000 games, Agent Goes First)	565	450	0	161.03
Trained Agent (5000 games, Random Goes First)	550	450	0	159.89

The Checkers environment illustrates a noticeable progression in the agent's performance, demonstrated by its increasing win rates and the gradual convergence in the number of moves per game. When comparing two random agents, the results are nearly even, with an average game lasting 83 moves. This baseline provides a reference point to evaluate the strategies learned by the agent during training.

After 1000 training games, the agent's win rate improves to 55.40%, and the average number of moves per game increases to 162.75. This suggests that the agent is learning to extend gameplay by making more calculated decisions, potentially minimizing premature losses and demonstrating an emerging strategic awareness.

By 2000 games, the win rate remains stable at 54.80%, while the average moves per game slightly decrease to 162.30. This subtle reduction may indicate that the agent is refining its strategy, becoming more efficient in closing out games without unnecessary delays.
At 5000 games, the agent's performance shows further improvement, particularly when it plays first, achieving a win rate of 56.50% and reducing the average moves per game to 161.03. When the random agent plays first, the trained agent still maintains a competitive win rate of 55%, with the average moves converging to 159.89. These results highlight the agent’s increasing ability to play efficiently while retaining a competitive edge, regardless of the turn order.

Although the agent has not reached professional-level performance, as evidenced by the move count stabilizing around 160, qualitative observations of rendered games confirm that the agent is employing strategic gameplay rather than relying on random actions. This indicates that the agent has successfully internalized key aspects of the Checkers environment. Future efforts could focus on optimizing hyperparameters or experimenting with alternative training methodologies to further enhance its performance and strategic depth.

Notebooks with Code

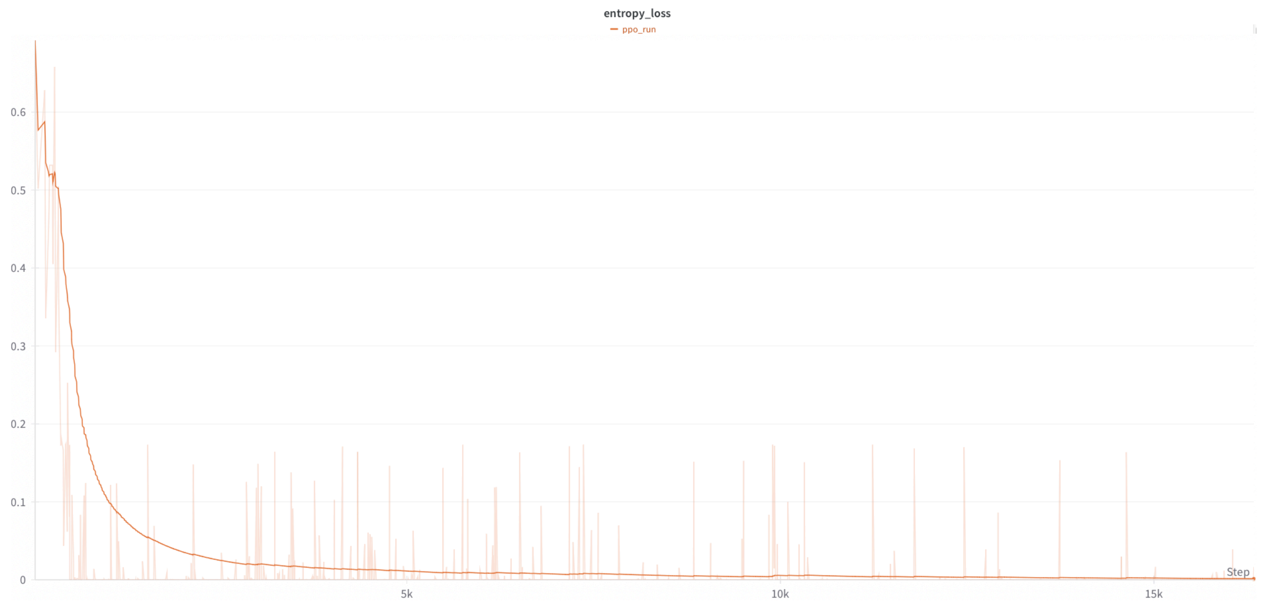
PPO Self-play Tic-Tac-Toe: Tic-Tac-Toe

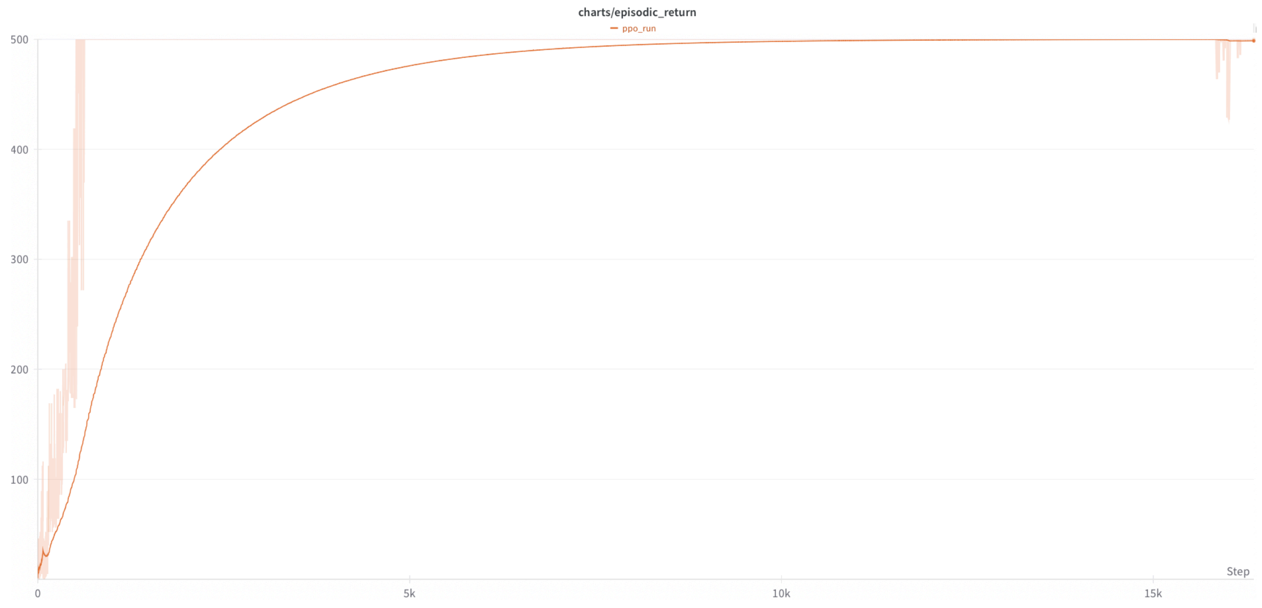
PPO Self-play Checkers: Checkers

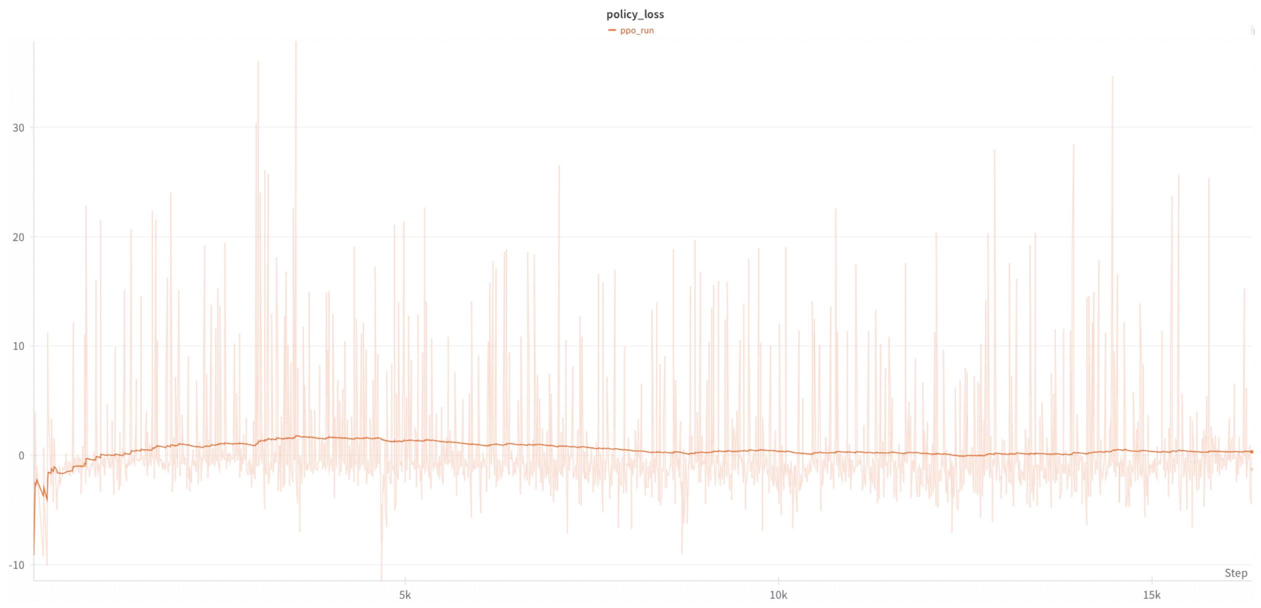
Appendix

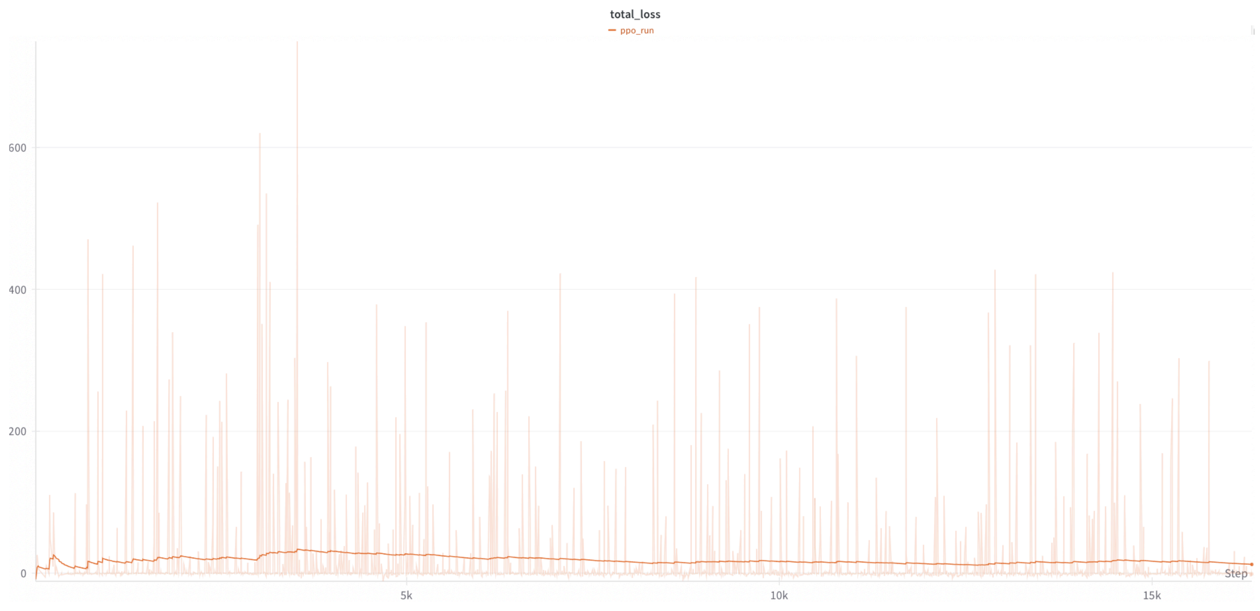
Dump of all results.

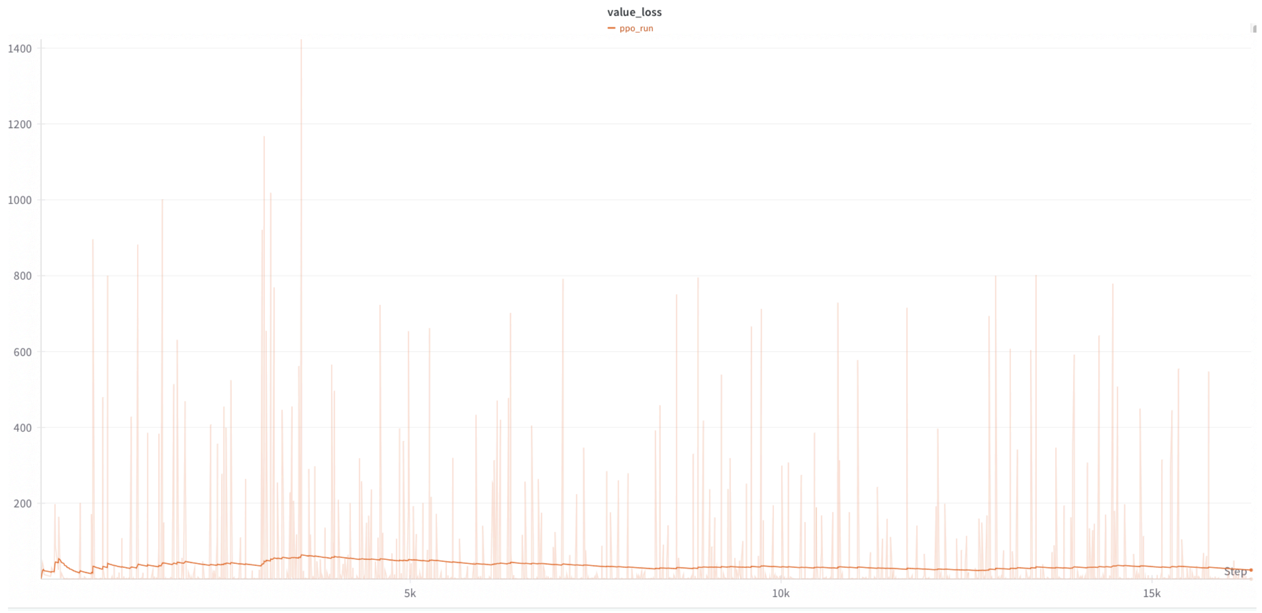
Cartpole





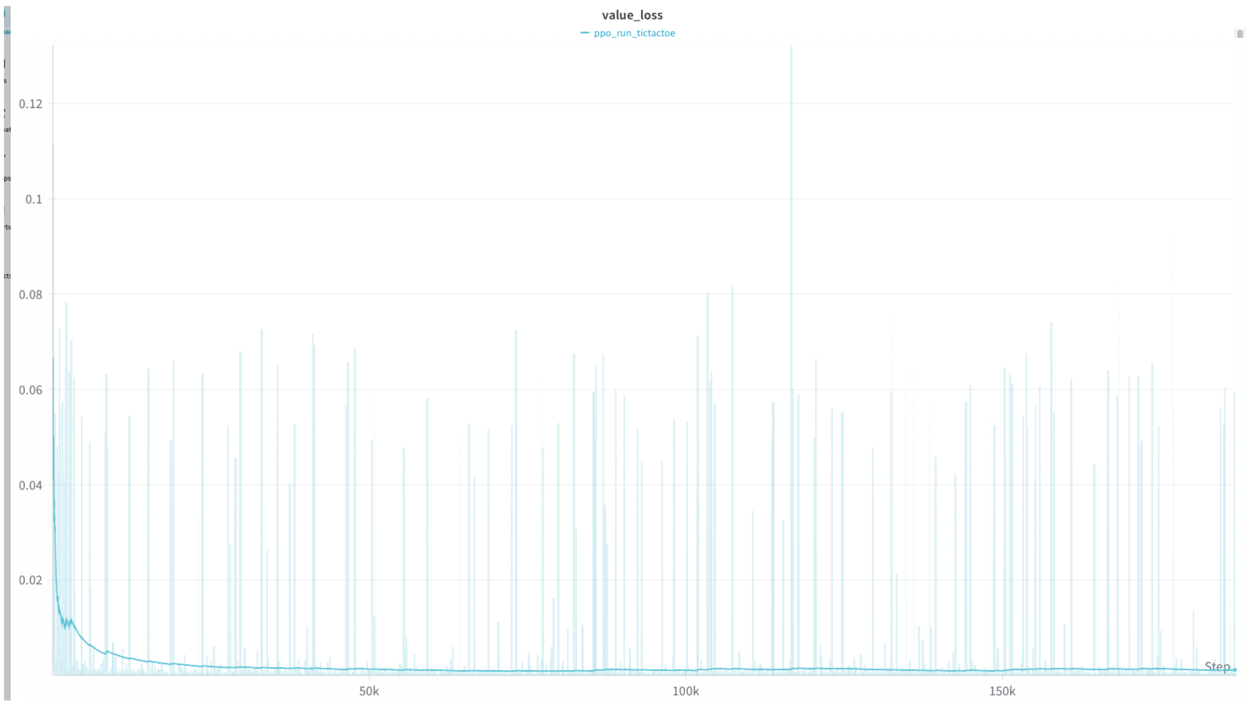
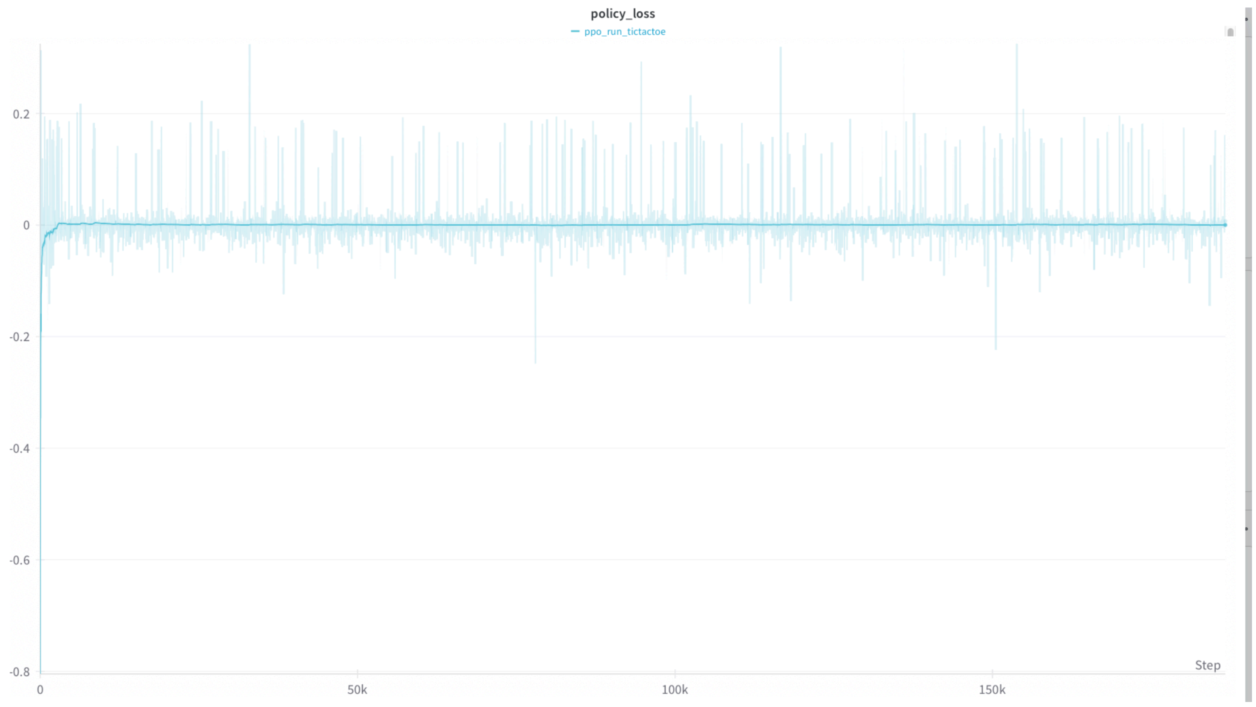

policy_loss
ppo_run
30
20
10
0
-10
Step
5k
10k
15k

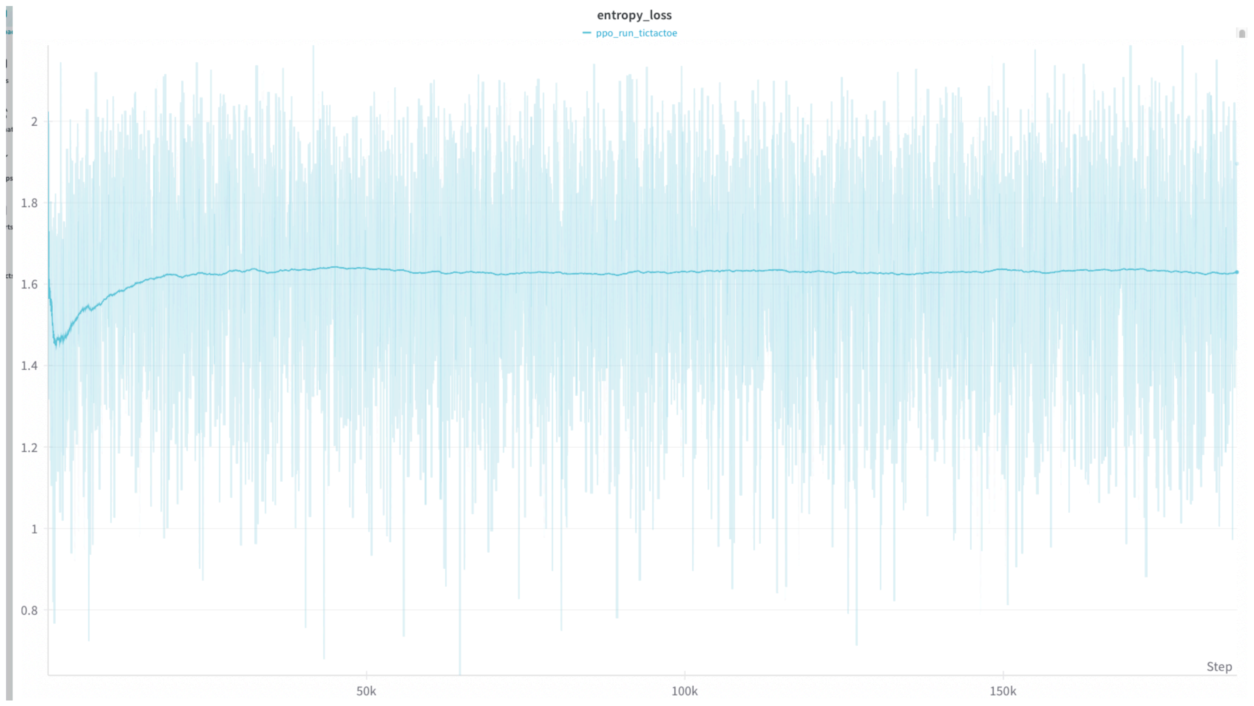


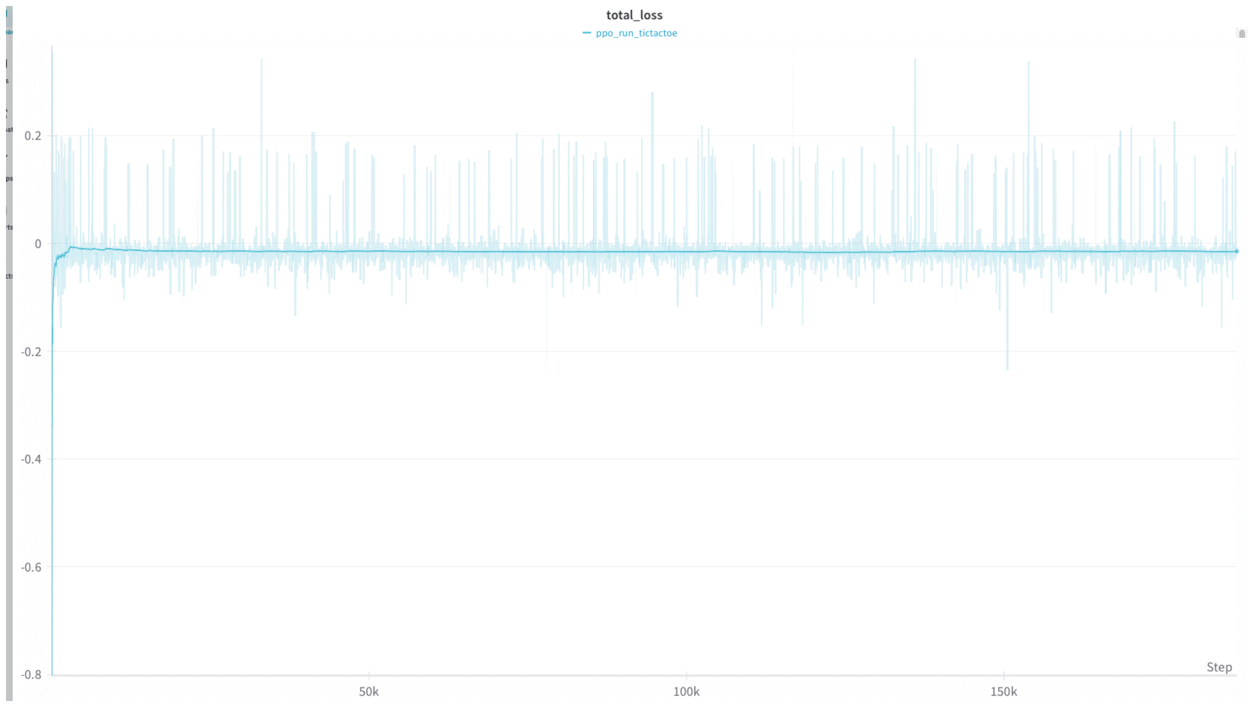


Tic Tac Toe

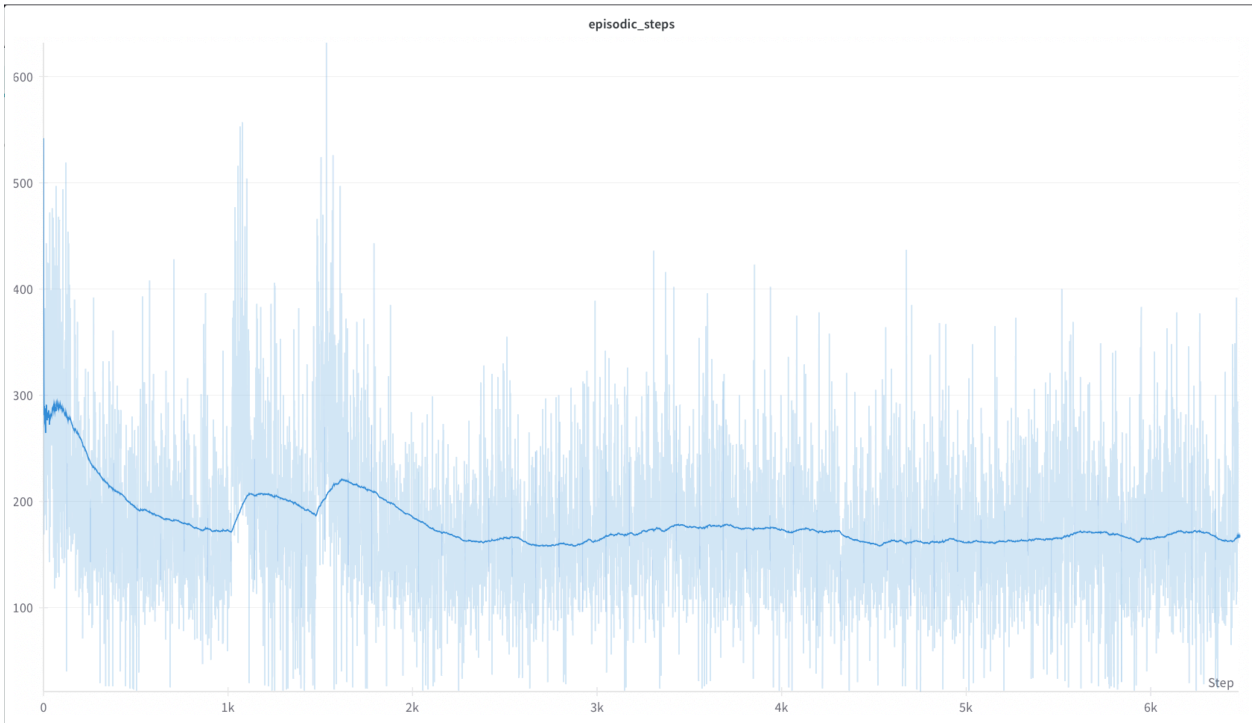
1000


value_loss
ppo_run_tictactoe
0.12
0.1
0.08
0.06
0.04
0.02
50k
100k
150k
Step

policy_loss
ppo_run_tictactoe
0.2
0
-0.2
-0.4
-0.6
-0.8
0
50k
100k
150k
Step


entropy_loss
ppo_run_tictactoe
2
1.8
1.6
1.4
1.2
1
0.8
50k
100k
150k
Step


total_loss
ppo_run_tictactoe
0.2
0
-0.2
-0.4
-0.6
-0.8
50k
100k
150k
Step

Checkers


episodic_steps
1k
2k
3k
4k
5k
6k
Step