– GridWorld –

Installation:

From Zip file	From Github

`unzip GridWorld-master.zip`
`cd GridWorld-master`
`pip install -r Requirements.txt`
`python setup.py install`
	`git clone https://github.com/prasenjit52282/GridWorld.git`
	`cd GridWorld`
	`pip install -r Requirements.txt`
	`python setup.py install`

Tested Software/Hardware Setup:

The project is implemented with **Python 3.9.x** along with numpy and pygame library to construct the GridWorld class. All library requirements are listed in the “Requirements.txt” file of the codebase. The algorithms are trained on a **MacBook M1 Air** (with 8GB primary memory running MacOS v13.4 with base-kernel version: 22.5.0).

Classical RL:

1. Create a **Gridworld environment** with holes and unsafe states.
2. **Implement** the Policy Evaluation, Policy Iteration, Value Iteration, Safe Monte-Carlo (MC) Learning, Safe Temporal-Difference (TD, **SARSA chosen**) Learning algorithms.
3. **Compare the results from the above algorithms**, and give insights why the results are better or worse for one algorithm compared to others.

Deep RL:

1. Create a **Gridworld environment** with holes and unsafe states.You have to create a large **discrete** (**40x40**) **gridworld** or a continuous Gridworld. Instead of being a single cell, each hole should be a **hole region**. The goal will also be a **goal region**.
2. Implement **DQN, NPG, TRPO and PPO** algorithms.
3. Compare the results from the above algorithms, and give insights why the results are better or worse for one algorithm compared to others.
4. Design a **visualization tool** for rendering how the algorithm controls the robot. The rendering function should preferably be part of the Environment Class. This will help demonstrate the agent's actions.

CRL: Objective 1:

After installing the tool, we can import and create a gridworld environment as follow:

```
from gridworld import GridWorld

world=\
    """
    wwwwwwwwwwwwwwwww
    wa          o    w
    w     o          w
    www       wwwwwwww
    w       o wg     w
    wwwww     ww   www
    w  o             w
    w         wwwwwwww
    w            o   w
    wwwwwwwwwwwwwwwww
    """

env=GridWorld(world,slip=0.2,log=False,max_episode_step=1000)
```

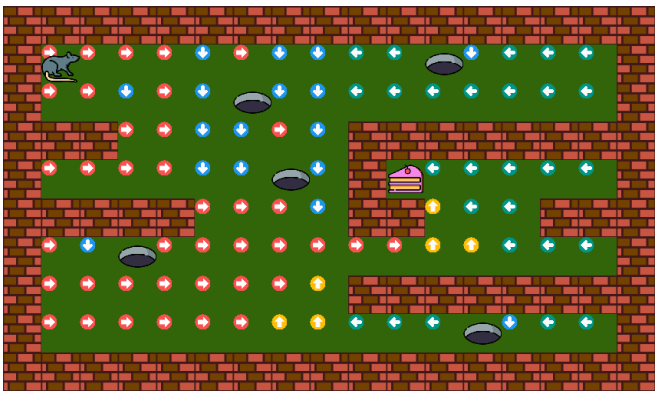


Fig. 1: Creating environment

Here “a” means Agent, “g” means Goal, “w” means Wall, “o” means Hole. The environment builds the transition probability matrix (P_sas) and the reward matrix (R_sa) from the defined string pattern to test Model-based algorithms. Moreover, for model-free algorithms, the tool provides an openAI-gym like interface to interact with the environment and explore.

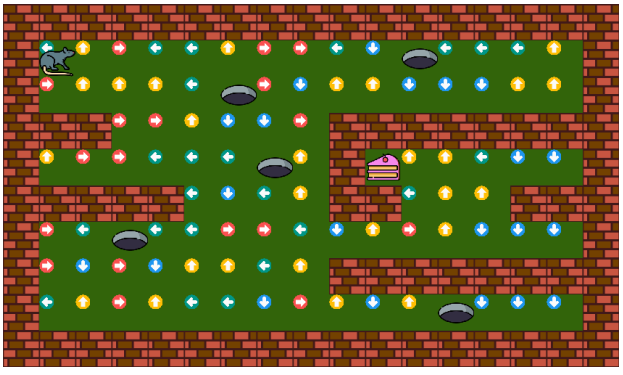
CRL: Objective 2:
“examples” folder in the codebase contains the desired Reinforcement Learning algorithms that are tested on this gridworld tool. Run the algorithms as follow:

- **Policy Evaluation**
  - `python example/policy_eval.py`
- **Policy Iteration**
  - `python example/policy_itr.py`
- **Value Iteration**
  - `python example/value_itr.py`
- **Safe Monte-Carlo**
  - `python example/safe_mc.py`
- **Safe SARSA**
  - `python example/safe_sarsa.py`

The underlined gridworld environment object is defined in "examples/gridenv.py", and the logs of each algorithm are stored in the "logs" folder of the codebase.

Definition> Safety
Unsafe probability “H_sa[s,a]” of taking an action “a” from state “s” is defined as the probability of falling in a hole at (1/(1-gamma_risk)th future state. For safe Monti-Carlo gamma_risk=1.0 (end of episode) and for safe SARSA gamma_risk=0.66 (3 future states).

CRL: Objective 3:
Note: The actions mappings are {0:'right',1:'down',2:'left',3:'up'}



Selected policy

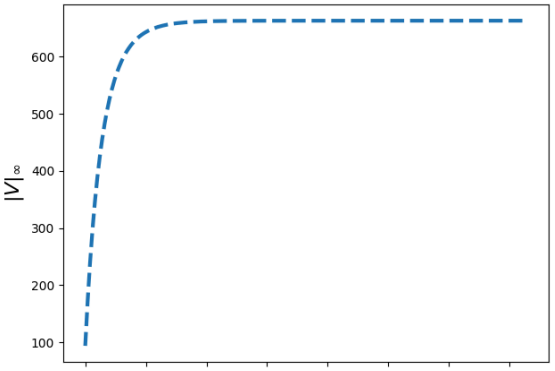


Fig. 2: Policy Evaluation

i) Policy Evaluation: A random policy is selected as shown in Fig 2. The Policy Evaluation algorithm converged in **183 iterations** with epsilon 1e-07 ($|V_{n+1}-V_n|$<epsilon).

ii) Policy Iteration: Policy Iteration converged in **20 iterations** (Pi_{n+1}=Pi_n) with final policy as Pi_*= [0 0 0 0 1 0 1 1 2 2 2 1 2 2 2 0 0 1 0 1 1 1 1 2 2 2 2 2 2 0 0 1 1 0 1 0 0 0 0 1 1 1 1 1 2 2 2 2 2 0 0 0 1 3 2 2 0 1 0 0 0 0 0 0 0 0 3 3 2 2 2 0 0 0 0 0 0 0 3 0 0 0 0 0 0 3 3 2 2 2 2 1 2 2] as shown in Fig 3.

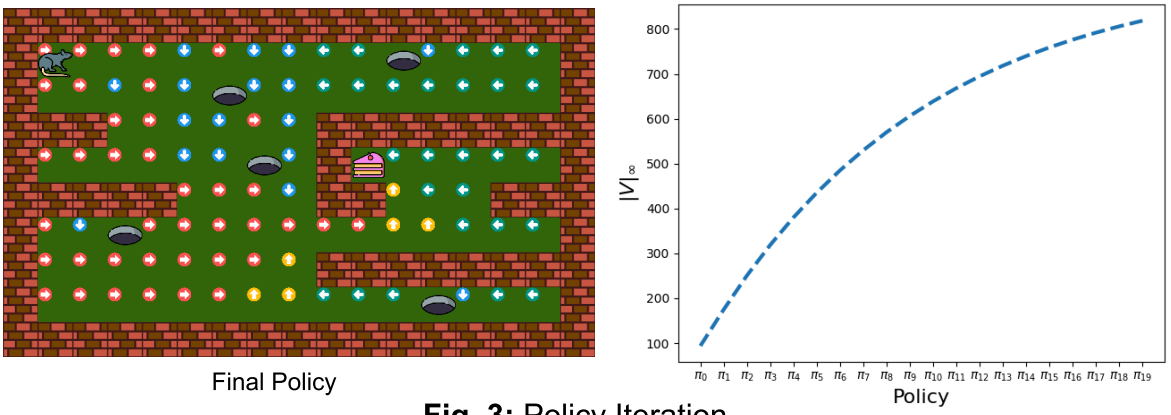


Fig. 3: Policy Iteration

iii) Value Iteration: Value Iteration converged to **epsilon-optimal** ($|V_*-V_n|$<epsilon) solution in **159 iterations** with eps 0.0001 Pi_*= [0 0 0 0 0 0 1 1 2 2 2 1 2 2 2 0 0 1 0 1 1 1 1 2 2 2 2 2 2 2 0 0 1 1 0 1 0 0 0 0 1 1 1 1 1 2 2 2 2 2 0 0 0 1 3 2 2 0 1 0 0 0 0 0 0 0 0 3 3 2 2 2 0 0 0 0 0 0 0 3 0 0 0 0 0 0 3 3 2 2 2 2 1 2 2] as shown in Fig 4.

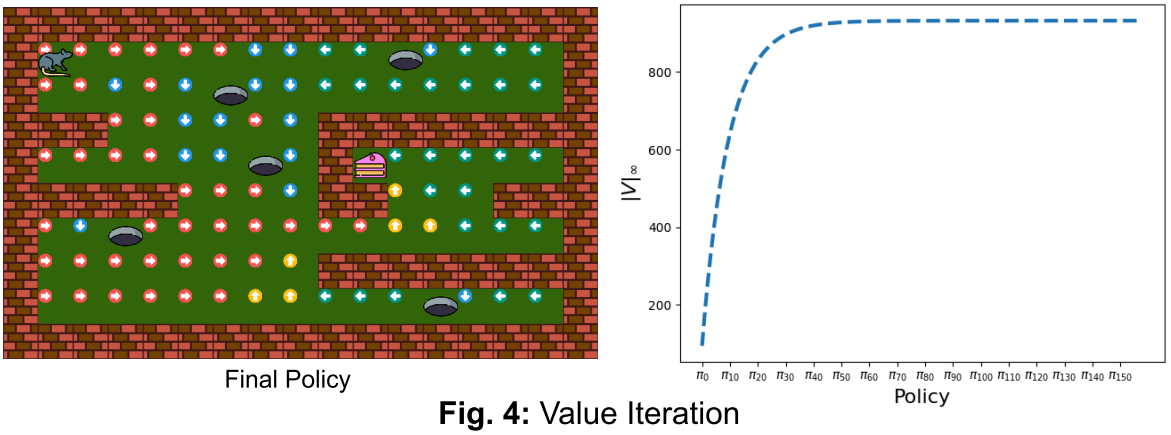


Fig. 4: Value Iteration

Policy Iteration v/s Value Iteration:

Both of the algorithms provide an all state optimal policy, however **Value iteration results in a superior policy** as shown in Fig. 3 and Fig 4. The infinity norm of value is greater in case of Value iteration. However, Value iteration takes **159 iterations** to converge as compared to Policy iteration which converges in only **20 iterations**. *This Policy Iteration is much more efficient but produces a slightly suboptimal policy than Value iteration.*

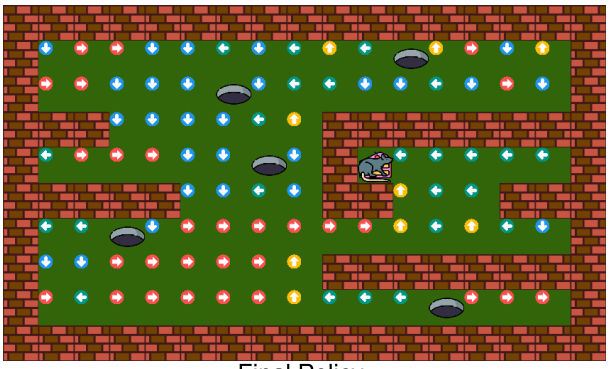
iv) Safe Monte-Carlo Control:

Hyperparametes:

gamma=0.99, alpha=1e-4 (learning rate), beta=0.999 (1-beta is learning rate for H_sa), **unsafe_prob=0.1**, episodes=100000, steady_explore_episode=30000

Epsilon is linearly decayed from 1 to 0.01 in first 30000 episodes then stays constant.

Pi= [1, 0, 0, 1, 1, 2, 1, 2, 3, 2, 0, 3, 0, 1, 3, 0, 0, 1, 1, 1, 0, 1, 2, 2, 1, 1, 2, 1, 0, 1, 1, 1, 1, 1, 2, 3, 2, 0, 0, 0, 1, 1, 0, 1, 0, 2, 2, 2, 2, 2, 1, 1, 2, 1, 3, 2, 2, 2, 2, 0, 1, 0, 0, 0, 0, 0, 0, 3, 2, 3, 2, 1, 1, 1, 0, 0, 0, 0, 0, 3, 0, 2, 0, 0, 0, 0, 0, 3, 2, 2, 2, 0, 0, 0, 0] as shown in Fig 5.



Final Policy

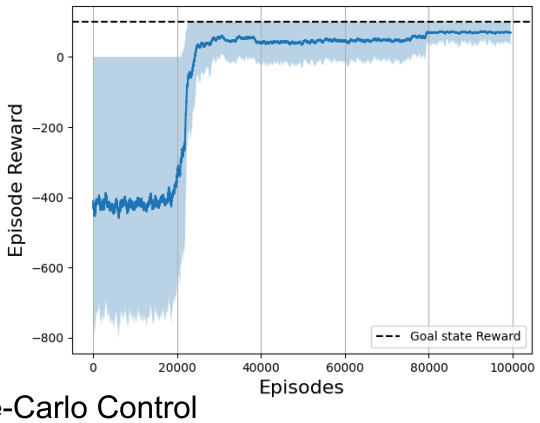


Fig. 5: Safe Monte-Carlo Control

Test runs:
Episode 0 Total reward 79
Episode 1 Total reward 76
Episode 2 Total reward 69
Episode 3 Total reward 73
Episode 4 Total reward 74
Episode 5 Total reward 79
Episode 6 Total reward 82
Episode 7 Total reward 72
Episode 8 Total reward 78
Episode 9 Total reward 71

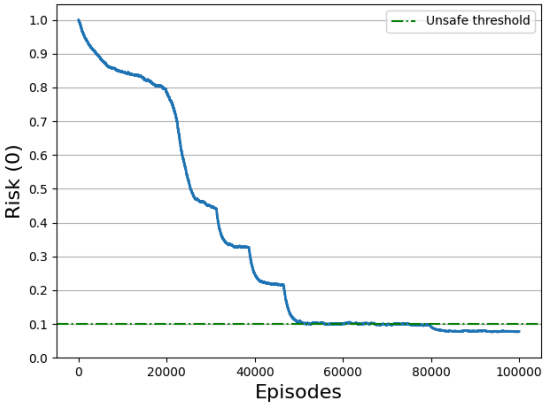


Fig. 6: Risk of falling from Start State (0)

v) Temporal Difference Learning (SARSA):

Hyperparametes:

gamma=0.99, gamma_risk=0.66 (3 future states), alpha=1e-3 (learning rate), **unsafe_prob=0.1**, train_steps=2000000, steady_explore_step=100000

Epsilon is linearly decayed from 1 to 0.01 in first 100000 interactions then stays constant.

Pi= [0, 0, 1, 1, 2, 0, 0, 3, 3, 2, 0, 3, 3, 2, 2, 0, 0, 1, 1, 2, 0, 0, 3, 1, 2, 2, 1, 1, 0, 0, 0, 1, 1, 2, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 2, 2, 2, 2, 2, 2, 1, 1, 0, 1, 3, 2, 2, 3, 1, 0, 0, 0, 0, 0, 0, 0, 0, 3, 2, 2, 2, 3, 3, 1, 0, 0, 3, 3, 3, 3, 2, 2, 2, 3, 3, 3, 3, 3, 2, 3, 2, 0, 0, 0, 0] as shown in Fig 8.

Test runs:
Episode 0 Total reward 81
Episode 1 Total reward 79
Episode 2 Total reward 80
Episode 3 Total reward 80
Episode 4 Total reward 77
Episode 5 Total reward 81
Episode 6 Total reward 74
Episode 7 Total reward 81
Episode 8 Total reward 77
Episode 9 Total reward 74

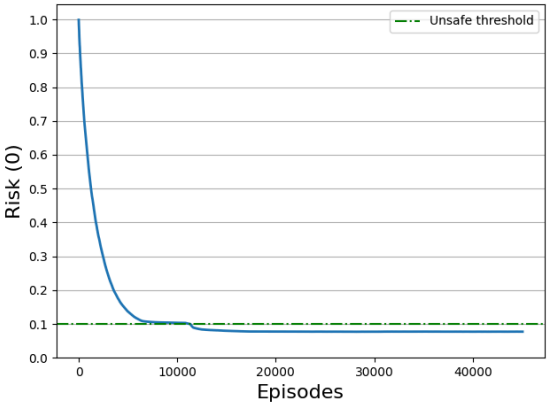


Fig. 7: Risk of falling from Start State (0)

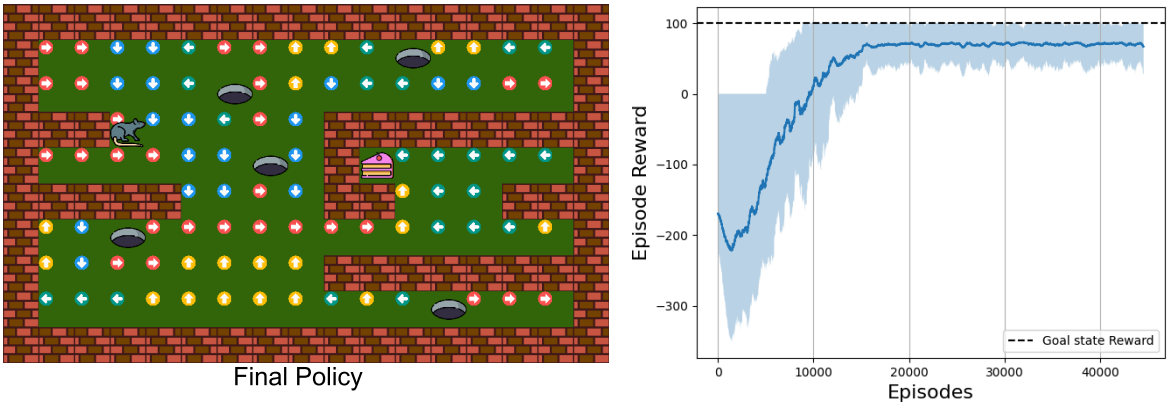


Fig. 8: SARSA (TD learning)

Monte-Carlo v/s SARSA

Both the algorithms give optimal Q values only for the states that occur frequently in the optimal path towards the goal. While satisfying the safety constraints **(10% falling rate at max)** as shown in Fig 6 and Fig 7**,** the SARSA **(20000 episodes)** algorithm converges quicker as it bootstraps the Q values thus faces less variance and stabilises earlier than Monte-Carlo **(80000 episodes)**. Thus **SARSA is efficient and performs similar** to Monte-Carlo in terms of game playing in the gridworld environment. Observing the final policies, we can say that both the algorithms come up with a path to *reach the goal as fast as possible while staying away from the holes*.

Model-based v/s Model-free

Despite that the model-based algorithms (Policy iteration, Value iteration) give the optimal policy in very few iterations as compared to model-free algorithms (Monte-Carlo, SARSA), model-based algorithms require the complete knowledge of the underline environment (P_sas, R_sa) to come up with a solution. However, most of the time we do not have the complete model of the environment. This is why model-free algorithms are more applicable in the real-world.

DRL: Objective 1:

After installing the tool, we can import and create a discrete gridworld with a string pattern shown in **examples/library/gridenv.py**

“a” means Agent, “g” means Goal, “w” means Wall, “o” means Hole. The goal and hole/trap **region can be represented with a blob** of the character in the string pattern as shown in the above file.

To make an environment compatible with DRL, pass isDRL as True in the GridWorld constructor.

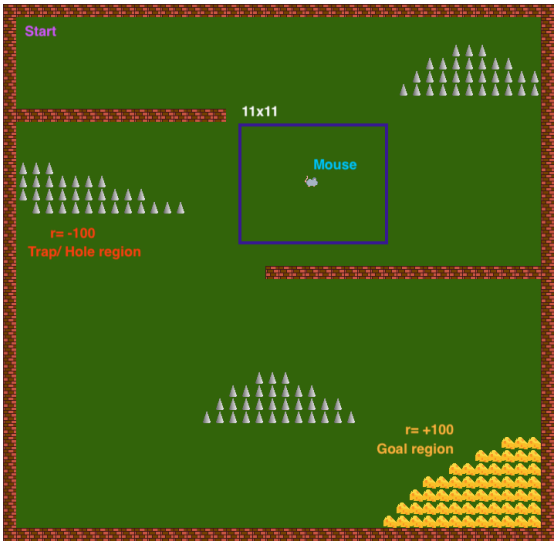


Fig. 1: 40x40 environment

DRL: Objective 2:

“examples” folder in the codebase contains the desired Reinforcement Learning algorithms that are tested on this gridworld tool. Run the algorithms as follow:

- **Deep Q Networks**
  - `python examples/dqn.py`
- **Natural Policy Gradient**
  - `python examples/npg.py`
- **Trust Region Policy Gradient**
  - `python examples/trpo.py`
- **Proximal Policy Gradient**
  - `python examples/ppo.py`

`Testing on saved model:`

```
python examples/[algo_name].py --init_from_exp [ALGO_NAME] --test --render
# example: python examples/ppo.py --init_from_exp PPO --test --render
```

The underlined gridworld environment object is defined in "**examples/library/gridenv.py**", and the logs of each algorithm are stored in the "logs" folder of the codebase.

DRL: Objective 3:

Note: The actions mappings are {0:'right',1:'down',2:'left',3:'up'}, I have used the ractGridWorld wrapper implemented in the GridWorld package. Thus, one action is repeated (repeat=**4**) time, and the corresponding state observations are stacked together to make the effective state representation (shape: 11x11x4 = **484**)

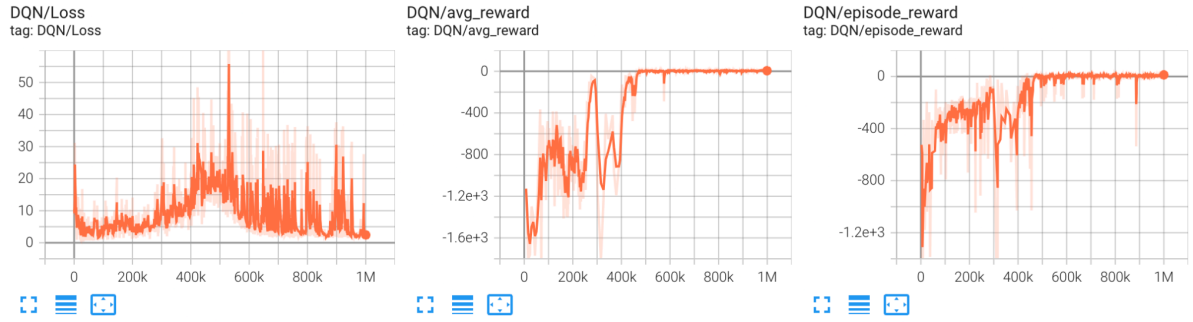


Fig. 2: DQN performance

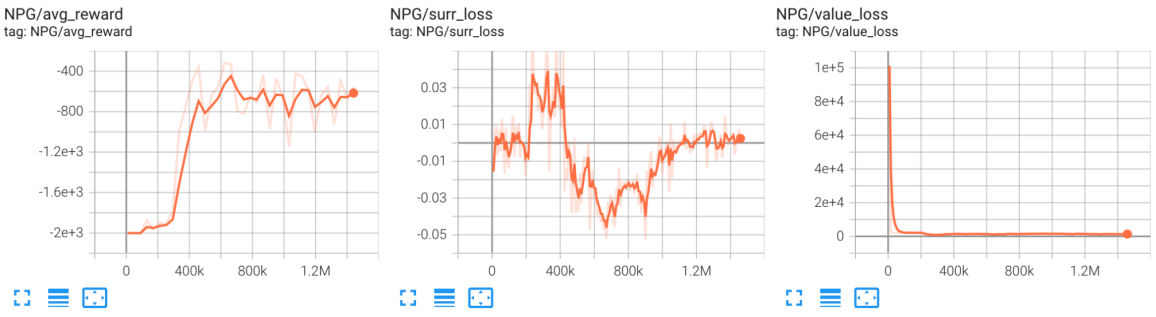


Fig. 3: NPG Performance

i) DQN: As shown in Figure 2, DQN converged with maximal performance among the above DRL algorithms (**avg reward of -1.2 for 10 runs**) in **1M steps**. It solves the environment and reaches the goal region. (See recorded video)

ii) NPG: As shown in Figure 3, NPG converged with minimal performance among the above DRL algorithms (**avg reward of -314.2 for 10 runs**) in more than **1M steps**. It does not solve the environment and falls into a trap/ hole region. (See recorded video)

iii) TRPO: As shown in Figure 4, TRPO converged with moderate performance among the above DRL algorithms (**avg reward of -67.6 for 10 runs**) in more than **1M steps**. It solves the environment. (See recorded video)

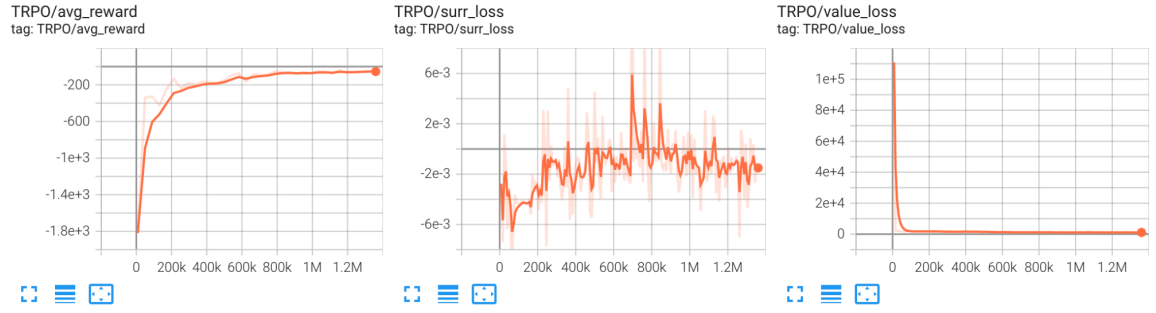


Fig. 4: TRPO Performance

iv) PPO: As shown in Figure 5, PPO converged with close to DQN level (best) performance among the above DRL algorithms (**avg reward of -11.6 for 10 runs**) in only **500k steps**. It solves the environment. (See recorded video)

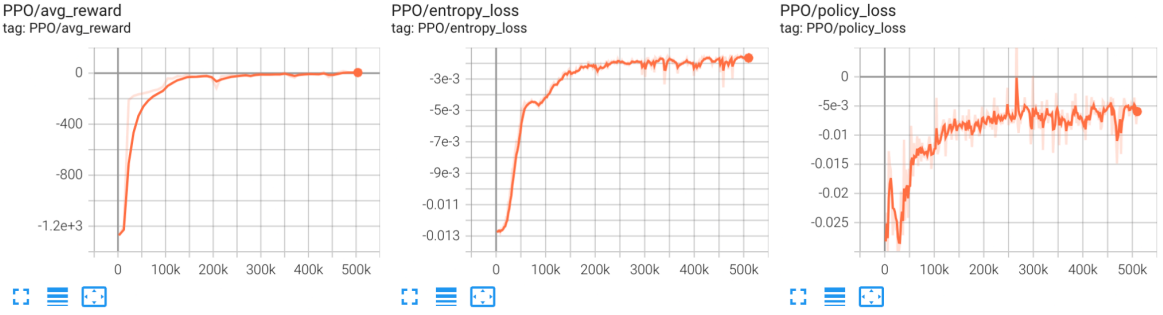


Fig. 5: PPO Performance

DQN vs PPO

Both the algorithms solved the given gridworld with almost similar performance. However, **DQN solves for a global optimality** due to complete off-policy training. Whereas, PPO is constrained with the collected data in each iteration from the current policy.
Hence **PPO solves for a local optimality** and might never reach DQN level performance. In contrast to DQN, **PPO is very fast in convergence** and has better convergence guarantee statistically. Thus, **PPO is more practical** and widely adapted in real-world RL implementations. *Note that in our experiments, DQN took **45 mins** to converge where PPO took only **10 mins**.*

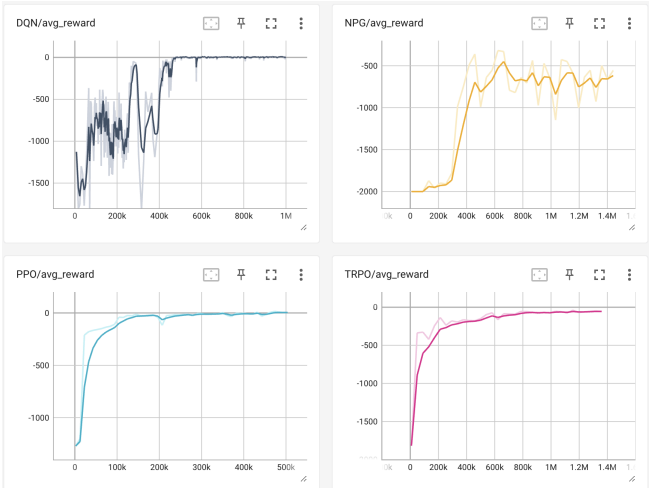


Fig. 6: Comparison

NPG vs TRPO vs PPO

NPG does not solve the environment due to **high approximation error in the Hessian Matrix** of KL divergence and direct update. Whereas, TRPO improved upon the updates with a **linesearch** for reducing noisy updates from Hassian and **importance sampling** in the policy gradient. However, due to second order differential over all network parameters, *both the algorithms are infeasible for larger networks or will take too much resources to train.*

PPO constrains the KL divergence with **importance ratio clipping** and achieves similar convergence properties with first order differentiation. Thus, among three algorithms, *PPO is the most advanced* and *NPG is the first to introduce the idea of trust region updates. Note that in our experiments, NPG and TRPO took **21 mins** to converge where PPO only took **10 mins**.*

DRL: Objective 4:

While testing the policy, we can just uncomment the “env.render()” line to have a visualisation of the agent interacting with the environment.

```
report=""
pi_report=f"Pi= {pi}\n\nTest runs:"
report+=pi_report+"\n"
print(pi_report)
for e in range(10):
    done=False
    total_reward=0
    s=env.reset()
    while not done:
        a=pi[s]
        #env.render()
        s_,r,done,info=env.step(a)
        total_reward+=r
        s=s_
    epi_report=f"Episode {e} Total reward {total_reward}"
    report+=epi_report+"\n"
    print(epi_report)
env.close()
```

Renders the environment

References

1. Richard S. Sutton and Andrew G. Barto; Reinforcement Learning: An Introduction; 2nd Edition, MIT Press, 2020.
2. Csaba Szepesvári; Algorithms of Reinforcement Learning; Synthesis Lectures on Artificial Intelligence and Machine Learning, vol. 4, no. 1, 2010.

3. Silver, D., Graves, A., Antonoglou, I., Riedmiller, M., Mnih, V., Wierstra, D. and Kavukcuoglu, K., 2013. Playing atari with deep reinforcement learning. DeepMind Lab. arXiv, 1312.
4. Kakade, S.M., 2001. A natural policy gradient. Advances in neural information processing systems, 14.
5. Schulman, J., Levine, S., Abbeel, P., Jordan, M. and Moritz, P., 2015, June. Trust region policy optimization. In International conference on machine learning (pp. 1889-1897). PMLR.
6. Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.