Dynamic Reward Shaping in Reinforcement Learning

Somanshu Singla
CSE
UCSD
A59023795

Eashan Gupta
ECE
UCSD
A59022933

***Abstract*—In RL, the reward function aligns the agent to the true goal intended in the problem. In real life, the rewards are sparse, as the agent receives the reward only on successful completion of tasks. In practice, people use 'dense' reward functions to train their models because agents need more supervision than just sparse rewards to learn meaningfully. We propose a framework to dynamically update the dense reward function to the sparse reward function during training using LiveTune [6]. Our proposed method leads to performance improvement across a wide variety of RL algorithms. Finally, we also show that this method is highly dependent on the episode during which the dense to sparse transition is made and provide an intuition on how to tune this parameter.**

I. Introduction

Reinforcement Learning tasks are modelled as Markov Decision Processes, which can be specified by a set of States, Actions and Rewards. RL algorithms try to learn a policy which will help the agent to decide an optimal action given a state. The optimality of a state is typically controlled by these rewards or reward functions formally.

Reward function is the way you can give signals to your models to identify the optimal actions and hence reward functions and their design is a very crucial aspect in any RL task. In the real world, these reward functions are often 'sparse' i.e. you earn a reward when you win a game or complete a task and no reward otherwise. In practice, learning from these real-world 'sparse' reward functions is very difficult, so, they are typically replaced by 'dense' reward functions. Dense reward functions provide more fine-grained rewards to the RL algorithms, making learning feasible and efficient. Experts often spend a lot of time and try out multiple of these reward functions before choosing one.

The task that we intend to tackle in this paper is what happens if we dynamically change the reward function while the RL algorithm is running. We plan to answer the following questions in our paper:

1) Is dynamic reward shaping a desirable property?
2) How hard is it to implement in practice?

More specifically, we intend to explore the outcome of changing the 'dense' expert design reward functions to the sparse real-world reward function. The motivation for this task stems from the fact that these dense reward functions used in training don't represent the real-world picture correctly, so we intend to see the effect of using this dense reward as a proxy when initiating the training and then switch to the real world sparse reward function to optimize for the 'ideal' goal.

In this proper, we will provide evidence that this switching is a desired property that can boost your model's performance and some easy ways to implement this in practice.

II. Related Work

Reward design has been widely studied and explored in the literature. The effects on AI safety and accidents have been explored due to the poor design of reward functions [1]. The idea of sanity checks and ensuring that your reward function will have the intended learning effects has also been explored [2]. The idea of overfitting reward functions to RL algorithms and the pitfalls of doing so has also been explored [3]. There have been works in which people have explored the ideas of dynamic rewards, but that has been to either improve the sample efficiency, run-time or accuracy of training [4] [5].

Overall, a lot of work has gone into the careful design of the 'dense' reward functions and dynamic reward functions but as per our knowledge research on updating the reward function from dense to sparse is very limited. Hence, we chose this idea to explore in our paper.

III. Problem Formulation

A. RL Domain

The Domain we will be working on within our paper is called the Hungry Thirsty domain.

In this domain food and water are placed at random locations in the grid and the agent also starts from a random location. Some movements are blocked by walls (depicted by the red thick lines in the figure). This domain has a fixed time horizon of 200 steps.

At each time step the agent can choose one of the six actions: move in a cardinal direction, eat or drink. The agent's goal is to avoid hunger for as many time steps as possible but the agent can only eat if it is in the same cell as food and is not thirsty. On each time step, the agent stochastically becomes thirsty with 0.1 probability, and only becomes not thirsty if it drinks while being in the same cell with water.

B. Agent State

For this domain, the agent state is defined as (i, j, H, T), where (i, j) is the location of the agent in our grid and H, T

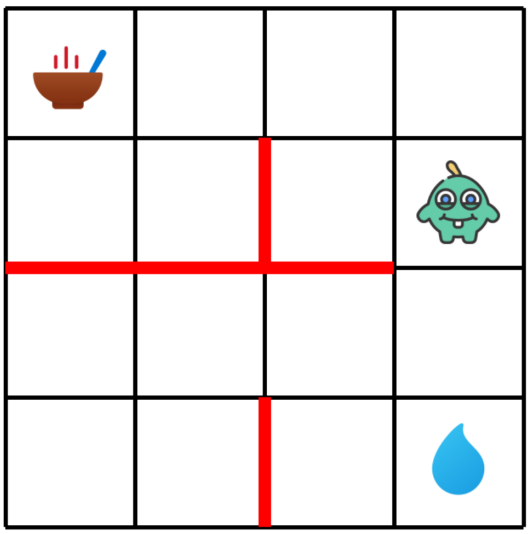

Fig. 1. Example of a 4x4 Hungry Thirsty setup [3]

are boolean variables indicating whether the agent is Hungry and Thirsty respectively.

For a 4x4 grid, the number of states comes out to be 64.

C. Performance Metric

For evaluation, we define a performance metric called "Not Hungry Count Per Episode". As mentioned previously the time horizon in this problem is fixed to 200 steps per episode. So, for each episode, we can calculate the Total Not Hungry Count.

$$Total\ Not\ Hungry\ Count = \sum_{t=1}^{200} I(H_t = 0) \qquad (1)$$

Here, I is an indicator function checking whether the value of the Hungry Variable is 0 at time step t. The higher this performance metric is the better the performance of the agent.

D. Reward Function Family

For our problem the reward function is defined as follows:

$$r(H \wedge T) = a$$
$$r(H \wedge \neg T) = b$$
$$r(\neg H \wedge T) = c$$
$$r(\neg H \wedge \neg T) = d$$

Here, $(a, b, c, d) \in R$ are the user-defined rewards for various states of the agent. Note that, the reward doesn't depend on the location of the agent.

1) Dense and Sparse Reward Functions: For all our experiments the dense and reward function values are defined as follows:

Reward Type	a	b	c	d
Sparse	0.0	0.0	1.0	1.0
Dense	-0.05	-0.01	1.0	0.5

E. Reward Function Transition

In our experiments, we use a step function kind of approach to make the transition from one reward function to another. We use one kind of reward function till a training episode and then switch to the other type after that epoch.

The epoch where this change is made is called *transition epoch* and we make this transition only once during training.

F. LiveTune

We leverage LiveTune [6] to enable dynamic updates on the reward functions.

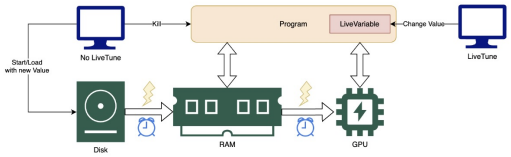

Fig. 2. Overview of LiveTune Framework [6]

The Livetune framework introduces 'Live Variables' which can be updated using very simple shell commands by communicating through a port.

IV. Experiments

In this section, we provide a brief overview of the RL algorithms that we used in our project. We then explain the experimental design used in this report.

A. PPO

Proximal Policy Optimization or PPO is a popular policy gradient method. Policy gradient methods in reinforcement learning involve optimizing parameterized policies through gradient descent with the aim of maximizing the expected return, which represents the long-term cumulative reward. A neural network is typically used to model the Policy. PPO is an on-policy algorithm where the agent sticks to the policy unlike Q-Learning.

B. DDQN

A double deep Q network or DDQN is an RL algorithm that uses two neural networks, a target network and an online network. The target network is used to predict the Q value of the action done by the agent at state S, whereas the online network is used to predict the action that the agent at state S should take to maximise the Q value. In DDQN, the target network is a deep copy of the online network that is updated intermittently, whereas the online network is updated after each timestep. For DDQN we define the Q value as,

$$Q^*(s_t, a_t) = r_t + \gamma[Q'(s_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(s_{t+1}, a))]$$

We then update our model by minimising the term

$$(Q^*(s_t, a_t) - Q'(s_t, a_t))^2$$

Here $gamma$ is the discount factor.

C. Q-learning

In Q-learning, we model the policy as a Q table that has the dimensions of MxN, where M is the number of possible states of the agent and N is the number of all possible actions that an agent can perform. Value at (i,j) position in the Q table is equal to the Q value that we get when the robot is at state s_i and performs the actions a_j. In the hungry thirst domain, we have 64 possible states for the agent (16 position values x 4 hungry-thirsty states), and 6 possible actions.

During training, we use the epsilon-greedy approach to update our Q matrix. In the epsilon-greedy algorithm, we select the best possible action given the state of the agent with a probability $1-\epsilon$ and select random action with the probability ϵ, where ϵ is a hyperparameter. This algorithm helps the model to explore more and improve training.

D. Experiment Design

We study the effect of reward function on the training of all three algorithms. For all three algorithms, we run three experiments.

- *Experiment-1*: We use the dense reward function throughout the model training.
- *Experiment-2*: We use the sparse reward function throughout the model.
- *Experiment-3*: We start our algorithm training with a dense reward function and we switch to a sparse reward function at a predefined transition point.
- *Experiment-4*: For Q learning we also experiment with starting our training with the sparse reward function and then switching to the dense reward function at a predefined transition point.

E. Hyperparameters

We trained the DDQN and the PPO algorithm for 5000 episodes, whereas the Q learning model was trained for 10000 episodes. Since we wanted to see the effect of the reward function on training, we kept other hyperparameters like ϵ in Q-learning, γ also known as the discount factor constant. The table below lists some of the hyperparameters used in this experiment.

Hyperparameter	PPO	Q-Learning	DDQN
Number of Episodes	5000	10000	5000
Timestep/Episode	200	200	200
Discount Factor (γ)	0.99	0.99	0.99
ϵ	-	0.15	-

V. Overall Results

In this section, we tabulate the results and discuss the findings of our experiments. In this report, we experimented with three common RL algorithms, PPO, DDQN and Q learning. We report the performance of the algorithms in terms of the average number of time steps for which our agent in the hunger-thirsty domain is not hungry. The average is taken over the last 100 episodes of the model training.

We first discuss the results of the PPO and DDQN algorithms followed by the observations of the Q-learning algorithm.

A. DDQN & PPO

The table below gives the average non-hungry count of our agent for the three experiments.

Algorithm	Experiment 1 Dense	Experiment 2 Sparse	Experiment 3 Dense to Sparse
PPO	50.4	26.79	94.12
DDQN	79.43	39.68	80.39

Overall, the findings in the results can be found below:

- Training from sparse function from start to end gives poor results when compared to training from dense function from start to end as seen in 3 and 5.
- The results from experiment-3 show that the dense to sparse transition can be tuned to give better results when compared to just sparse or just dense training as seen in 4 and 6.
- The transition episode affects the results a lot as seen in 4 and 6.
- Making the transition early on during training typically yielded better results for both PPO and DDQN.

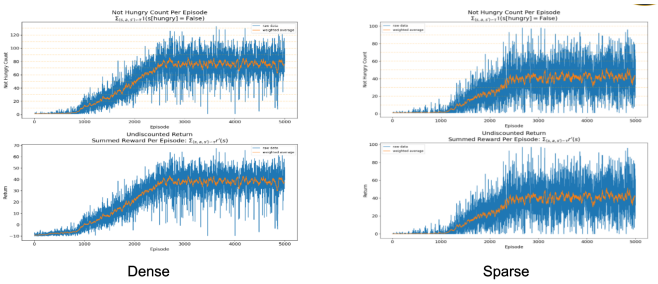

Fig. 3. DDQN Reward for Experiment 1 and Experiment 2

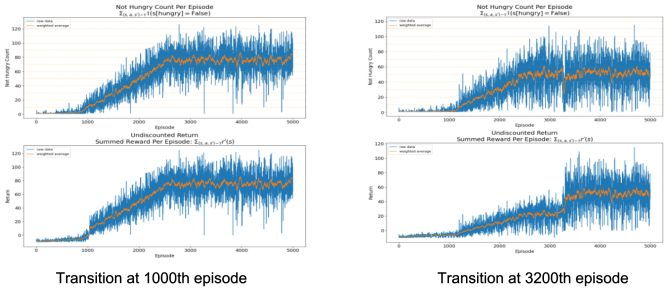

Fig. 4. DDQN Reward for Experiment 3

We believe that for PPO and DDQN transition at earlier episodes improves the results because the agents can explore more during the starting phase of the training. While training with such a sparse reward function it is necessary to explore more because only then there can be meaningful updates to model parameters.

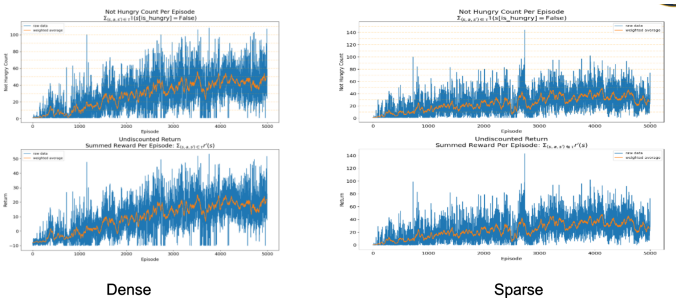

Fig. 5. PPO Reward for Experiment 1 and Experiment 2

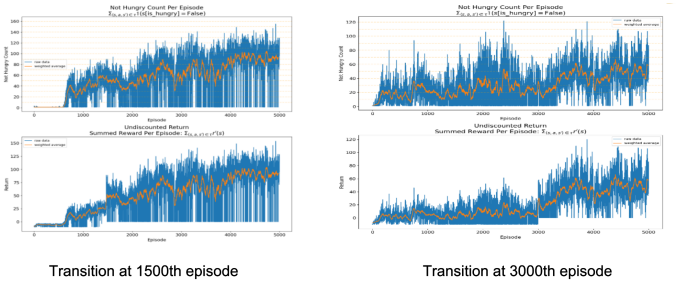

Fig. 6. PPO Reward for Experiment 3

B. Q Learning

The table below gives the average non-hungry count of our agent for the three experiments.

Exp. 1 Dense	Exp. 2 Sparse	Exp. 3 Dense to Sparse	Exp. 4 Sparse to Dense
19.92	3.12	19.78	20.04

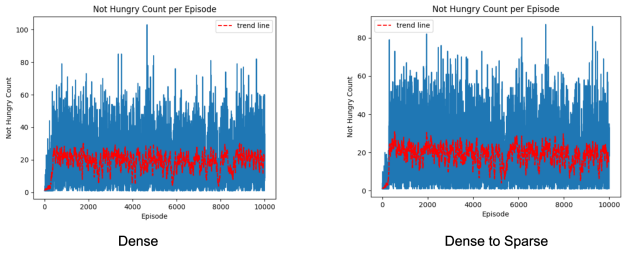

Fig. 7. Q Learning Reward for Experiment 1 and Experiment 3

For Q-Learning the major observation is that dense to sparse transition doesn't have any noticeable effect on the model performance, we believe that this is due to the fact that Q-Learning is a less explorative algorithm when compared with PPO, DDQN and isn't able to meaningfully learn from a sparse reward function.

VI. Sparse Reward Learning: Exploration Hypothesis

We hypothesize that for any RL algorithm to learn meaningfully from sparse reward function, it must be sufficiently explorative. We believe that is the reason for the failure of dense to sparse transition in Q-learning.

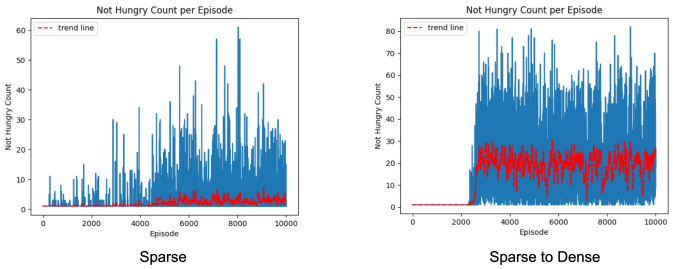

Fig. 8. Q Learning Reward for Experiment 2 and Experiment 4

Some possible solutions for enabling the learning of an RL algorithm in sparse reward settings:

- Increase the exploration of algorithm after transition by increasing ϵ if your algorithm is ϵ greedy.
- Increasing the number of training episodes post-transition.

VII. Contribution and Conclusions

In this section, we will summarize the overall findings of the paper and the contributions we have made:

- We find some merit in upgrading the dense reward function to a sparse reward function.
- For our domain we propose a step function style transition that seemed to work well.
- The algorithms are very sensitive to the 'transition point' and choosing a transition point early in the training leads to better results.
- LiveTune is useful when the user is exploring these kinds of choices while training, as it saves time because you need not run the entire training from scratch.

Overall, we propose a step-function-based transition from dense to sparse function while RL training which can boost the performance of the algorithm. We develop a framework to do this transition efficiently using Livetune.

VIII. Limitations and Futurework

In this section, we list some limitations in our current work and future work that can be done to explore this problem further:

- Our work explores a very simple domain and further research for more complex domains is warranted to further test the usability of this approach.
- Automated algorithms can be explored for tuning the transition point.
- Effect of reward function transition can be explored, where instead of a single step we use a continuous transition.
- The selection of transition points can be studied by doing transitions on various points and measuring the success metric for each transition point.

REFERENCES

[1] Amodei, D.; Olah, C.; Steinhardt, J.; Christiano, P.; Schul- man, J.; and Man´e, D. 2016. Concrete problems in AI safety. arXiv preprint arXiv:1606.06565.

[2] Knox, W. B.; Allievi, A.; Banzhaf, H.; Schmitt, F.; and Stone, P. 2021. Reward (mis) design for autonomous driv- ing. arXiv preprint arXiv:2104.13906.

[3] Booth, S., Knox, W. B., Shah, J., Niekum, S., Stone, P., Allievi, A. (2023). The Perils of Trial-and-Error Reward Design: Misdesign through Overfitting and Invalid Task Specifications. Proceedings of the AAAI Conference on Artificial Intelligence, 37(5), 5920-5929. https://doi.org/10.1609/aaai.v37i5.25733

[4] Tenorio-Gonzalez, A.C., Morales, E.F., Villaseñor-Pineda, L. (2010). Dynamic Reward Shaping: Training a Robot by Voice. In: Kuri-Morales, A., Simari, G.R. (eds) Advances in Artificial Intelligence – IBERAMIA 2010. IBERAMIA 2010. Lecture Notes in Computer Science(), vol 6433. Springer, Berlin, Heidelberg.

[5] Jiang, Y., Bharadwaj, S., Wu, B., Shah, R., Topcu, U., Stone, P. (2021). Temporal-Logic-Based Reward Shaping for Continuing Reinforcement Learning Tasks. Proceedings of the AAAI Conference on Artificial Intelligence, 35(9), 7995-8003. https://doi.org/10.1609/aaai.v35i9.16975

[6] Shabgahi S. and Sheybani N. and Tabrizi A.and Koushanfar F. (2023). LiveTune: Dynamic Parameter Tuning for Training Deep Neural Networks. arXiv:2311.17279.