Sparse Reward Propagation for Deep Reinforcement Learning

A Project Report for the end-term evaluation of the course ID3801 (OELP)

Bachelor of Technology

by

Raswanth Murugan, Alisetti Sai Vamsi
111801056, 111801002

Under the supervision of
Dr. Chandra Shekar Laskminarayan

Department of Computer Science and Engineering
Kerala, India - 678557
April 2021

Contents

1 Introduction 3

2 Related Background 4
2.1 Basic Introduction to Reinforcement learning 4
2.2 Reward Shaping 6
2.3 Graph Convolutional Networks 6
2.4 Latent Feature Representation 8

3 Solution - Sparse Reward Propagation 8

4 Implementation 9
4.1 Implementation and Training of GCN 9
4.2 Actor Critic Algorithm 10
4.3 Hyper Parameters 12
4.4 Approximation of the MDP as a graph and Proto Value Functions 12
4.5 CodeBase 14

5 Results 14
5.1 Reward Propagation & Scalability 14
5.2 Performance 15

6 Future Directions 17

7 Contributions 18

8 Conclusion 19

9 Acknowledgement 19

1 Introduction

Reinforcement learning (RL) is a computational approach to learning from interaction. In spite of the extensive research being conducted in the area in the last 2-3 decades, RL's potential as a learning paradigm had not been leveraged until recently. The advances in deep-learning allowed RL algorithms to use neural networks to represent information obtained through interaction with the environment more efficiently, giving rise to Deep Reinforcement Learning. The success stories of Alpha-Go and Alpha-zero are some examples of the same.

In RL, the "control task" is represented by a reward function. The reward function (mapping from state space to real numbers) is a measure of how good it is to be in a state and the objective of the RL system (agent) is to learn to behave in way that maximizes the cumulative reward obtained. Once the reward function is found, the optimal policy can be determined. However, in a lot of RL tasks, the agent will only receive the reward when it reaches the goal. In fact the rate of converge of RL algorithms depend on the how well or poorly the reward function is designed. Designing a good reward function requires expert knowledge in the domain of application and varies in difficulty based on the control task.

Consider the task where we want a robotic arm to autonomously learn to make a specific kind of knot on a rope based on pictures of the knot. This problem can be well posed in the RL framework however it is challenging to solve. The challenge here is mainly associated with designing a good reward function for the task. Recall that the reward function is a map from state space to real numbers (In this particular task the state space is set of all different configurations of rope and the goal state is the knot). From this we can see that the state space (number configurations of a rope) is very large and the most intuitive reward function is assigning the same value to every non-goal state and higher value to a goal state as described in Eq.1.

$$R(s) = \begin{cases} c & \forall s \neq s* \\ c + 100 & s \ = s* \end{cases} \tag{1}$$

This kind of a reward function is easy to design however is not a good reward function. Rewards act as signals the agent uses to infer the best actions and hence if the robotic arm used Eq.1 then the robotic arm would get same reward on making a loop (a crucial intermediate step to making the knot) and just sliding the rope (an example of an unnecessary step toward making the knot). This kind of reward system is the same as telling a student he/she is correct only if they manage to get the full correct answer and providing no feedback in every other case (wrong, partially correct,almost correct etc.). These kind of RL tasks where the reward function is not informative for the learning are called sparse reward problems.

Recent work in RL and Graph Neural Networks (GNN) suggests that it should be possi-

ble to learn a latent representation of each state and incorporate reward shaping theory [1] with GNN [2] to generate potential rewards of every single state so that they can solve the sparse reward problems. We use a combination of GNNs or the laplacian [3] method and reinforcement learning methods [4],[5] to provide a solution that not only addresses the sparse reward problem but also reduces effort required to carefully design a good reward shaping function.

2 Related Background

In this section we will give a brief overview of the concepts employed in our solution.

2.1 Basic Introduction to Reinforcement learning

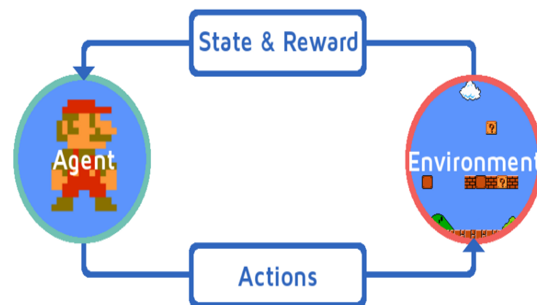


Figure 1: The Agent - Environment interaction in RL

In this section we introduce basic formalisms in Reinforcement learning in the context of our experimental setup. This is by no means an extensive introduction [6],[7],[8] are great sources for a more comprehensive introduction to Reinforcement learning.

The loop in figure Fig.1 is present in almost all RL tasks. In the case of the task of learning to make a knot the agent is the robotic arm and the environment is the rope. In our work we run our experiments in a 2D-gridworld as show in Fig.2.

RL is used to perform control tasks and RL problem can be posed as a Markov Decision Process (MDP) which is defined by a 5-tuple (S,A,P,R, γ). We will define each element of this tuple and also explain the corresponding quantities in our 2D- gridworld setup. The task in our experiments is to learn the **shortest path** from any state to the goal state (the grid marked 25 in Fig.2) .

- **S** (State Space) - We are dealing with the grid world and so each state inside the $n \times m$ grid world is a state and belongs to the set S. The coordinates (i,j), $\forall i \in [n], j \in [m]$ represent all the states in S. In Fig.2, $n = m = 5$.
- **A** (Agent) - at each state the agent can more $\uparrow$, $\downarrow$, $\leftarrow$ or $\rightarrow$.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Figure 2: The gridworld environment

- **P** (transition probability matrix) - P defines mechanism of the transition. An entry in this matrix is of the form $\mathrm{P}(s'|s,a)$ or the probability of going to state s' by doing action a in state s. So in the case of the grid world the transition matrix looks like
  - $\mathrm{P}((x-1,y)|(x,y),\uparrow) = 1$
  - $\mathrm{P}((x,y+1)|(x,y),\leftarrow) = 1$
  - $\mathrm{P}((x+1,y)|(x,y),\downarrow) = 1$
  - $\mathrm{P}((x,y-1)|(x,y),\rightarrow) = 1$

  However, if an action a takes the agent out of the gridworld, the agent lands back on the current state or $\mathrm{P}((x,y)|(x,y),a) = 1$.

- **R** (Reward Function) - It is a function $S \times A \rightarrow \mathbb{R}$. Our objective is to start from (1,1) and reach (n,m) through the shortest path. In this case we choose to work with 5×5 gridworld. The general aim of RL agents is to maximize total rewards $(\sum_{t=0}^{T} R_t(s_t, a_t))$ and since our goal is for the agent to learn the shortest path, we design the reward in such a way that maximizing the total reward is equivalent to the shortest path taken.

$$R((x,y),a) = \begin{cases} -1 & \forall a \; and \; (x,y) \neq (n,m) \\ 0 & \{(n,m,.)\} \end{cases} \tag{2}$$

- γ - *Discount factor.* γ is a constant that $\in [0,1]$ that determines how much the agent looks forward into the future i.e it is a factor that informs the agent to pick between immediate rewards vs future rewards. Setting $\gamma = 0$, makes the agent choose immediate rewards over future rewards. And setting $\gamma = 1$ makes the agent choose future rewards over immediate rewards.

The next set of important terminologies are not essential to describe the MDP, but are essential for the agent to learn and solve the MDP.

- π - *Policy* is a function from $S \times A \mapsto [0, 1]$. It tells the agent with what probability the agent should choose an action given its state. It is probability distribution over the action space for a given state (or) $\Sigma\pi(.|s) = 1$. An optimal policy helps the agent to maximize the total reward and it is this quantity that we hope to learn at the end of the learning process. The optimal policy in our experiment should direct the agent to go the the goal state by taking the shortest path.
- **Q** - *Action Value function* is a function $S \times A \mapsto \mathbb{R}$. It tells us how good it is to be at a particular state and do a particular action. $Q(s, a) = \mathbb{E}\left[\sum_{t=0}^{T} R_t \mid s_0 = s, a_0 = a\right]$. This is an intrinsic record maintained by the agent to keep track of its experiences.

It must be kept in mind that R and P are used to define the MDP however, during experiments agent does not have direct access to R or P and is expected to learn about them through interactions with the environment. Also it is easy see the Eq.2 and Eq.1 and fundamentally the same sparse reward function.

2.2 Reward Shaping

The reward shaping is a method to address the sparse reward problem in RL systems. It modifies the sparse reward function by adding a potential-shaping function that makes the rewards less sparse.The method is as follows:

$$R^{'}(s_t, a_t, s_{t+1}) = R(s_t, a_t) + F(s_t, s_{t+1}) \tag{3}$$

Here $F(s_t, s_{t+1})$ is the shaping function which can encode expert knowledge. RL comes from a lot of inspiration derived from neurological pathways in animals and an important aspect innate in animals that helps with learning is curiosity. [9] explores such analogies between the shaping function and curiosity in animals further motivating the choice of such functions. As mentioned earlier reward function is key to learning the optimal policy so by adding the shaping function will the optimal policy change? As it turns out it this possible. [10] showed a way by which (a necessary and sufficient condition) the shaping can be done without changing the optimal policy. This condition is given by

$$F(s_t, s_{t+1}) = \gamma\phi(s_{t+1}) - \phi(s_t) \tag{4}$$

Here ϕ is a scalar potential function $\phi : S \to \mathbb{R}$, which can be any arbitrary function defined on states. γ is a constant between $[0, 1]$ and we will use $\gamma = 1$ for our experiments. As mentioned earlier, this function encodes expert knowledge and hence researchers ([10] defines it distance between current position to goal state) have explored and used different definitions according to the problem at had. In our work we differ by using a an end-to-end approach (using a neural network) to determine the potential functions as opposed to previous approaches that require human designers.

2.3 Graph Convolutional Networks

Graph Convolutional Networks (GCN) belong to the family of neural networks called Graph Neural Networks(GNNs) which operate on graph structured data. GNNs employ

a message exchange mechanism between vertices to perform different tasks like classification, regression etc. hence they are also called message passing networks. GCNs are infact first-order approximations of spectral graph convolutions giving and hence are more explainable neural networks as opposed to more deep learning models today. Among the class of GNNs, GCN was chosen based for the experiments on its recent success in semi-supervised learning for classifying nodes on a graph. semi-supervised refers finding the labels of unlabelled nodes in a graph while only knowing the true labels of a few nodes. The GCN is able to perform such a task because it uses message passing to propagate the information about the labelled nodes to other nodes.

The mathematical framework behind the message passing framework in a GCN uses the graph laplacian. This graph laplacian also functions similar to a discrete case diffusion operator and hence does a good job in diffusing information among nodes where each node requires information about the other nodes to arrive at a result.

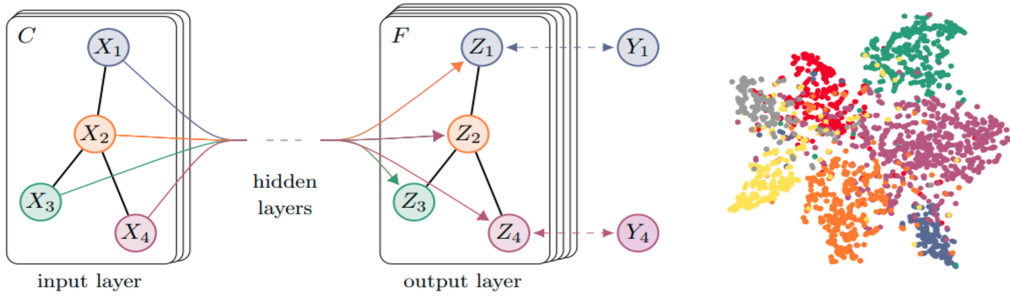


Figure 3: The figures are taken from [2]. The figure on the right shows dots that represent the nodes of a graph and each different colour is the label for each node. Here at the start only the colours of a few nodes are known and the GCN helps in accurately guess the colours of the other node (colours refer to labels). This result is used as the basis to the solution to sparse reward propagation

$$\Phi(X) = \sigma(\hat{T}\, ReLU(\hat{T} X W^{(0)}) W^{(1)}) \tag{5}$$

where $W(i)$ corresponds to the weight matrix of layer i and X is the input matrix with shape $N^{nodes} \times M^{features}$ which we will look in greater in the next section. The matrix $\hat{T}$ is a transition matrix defined as $\hat{T} = D^{-1/2}\tilde{A}D^{-1/2}$ and σ is a non-linearity (in our case a softmax layer) , where A is the adjacency matrix with added self-connections and D is the degree matrix, that is $D_{xx} = \sum_y A_{xy}$. The GCN passes messages across vertices in the graph and so if we manage to pass the information about the reward between the states, then we can use Φ as a potential function that will address the sparse reward problem by passing information from goal state to other non-rewarding states.

2.4 Latent Feature Representation

The matrix X of shape $N^{nodes} \times M^{features}$ goes as input in Eq.5. This is know as the feature matrix and plays a crucial role while training the GCN and turns out to be non-trivial quantity to obtain. Traditionally X is hand engineered from knowledge about the domain however this is challenging for the following reasons.

1. Cannot be generalised for different graph structures
2. Time consuming
3. No theoretical guarantees

To overcome these challenges we use Proto Value Functions (PVF) derived in [11], [12]. PVF is derived from the *Graph Laplacian*. Simply put they are eigen vectors of the graph laplacian. First we convert the MDP into a graph (the vertices are states and the edges are between two states if a transition is possible). Then we use the *Adjacency Matrix* and the *Degree Matrix* to compute the eigen vectors associated with this MDP and stack them row-wise forming the feature matrix.

$$L = D - A \,(or)\, L^{sym} = I - D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \tag{6}$$

L is the graph laplacian and L^{sym} is the normalized graph laplacian and is symmetric in nature. It is on this matrix (L or L^{sym}) whose eigen decomposition gives us the feature matrix X which later becomes the input to the GCN (L^{sym} is usually preferred as it is symmetric and hence more easily decomposable). [12] shows that such eigen vectors form basis vectors of all possible value functions for a given vector and capture the MDP symmetries and topology effectively. Hence we now have a method that is theoretically sound, can generalise to different graphs and does not require hand engineering. Eigen decomposition is not a computationally trivial task and there have been methods that produce features. These methods trade off between the computational expense and richer representation of the MDP. Some examples of such methods include neural networks (latent representation learning networks) and random walk based approaches (DeepWalk). Hence depending on the task one can choose appropriately.

3 Solution - Sparse Reward Propagation

Firstly we begin by representing the MDP (2D gridworld) as a graph or more specifically as an adjacency matrix, A. Using this A and D (degree matrix obtained from A) we compute the feature matrix X using eigen decomposition of the graph laplacian or any other methods of choice in section 2.4. This matrix is then used as input to the Graph Convolutional Network using equation Eq.5.

Once the GCN is trained, the output function Φ is augmented with the sparse reward function as mentioned in section 2.2. Once this augmentation is performed we then use the new reward function for training a standard RL algorithm (Actor critic in our case)

and learn an optimal policy.

In the proceeding sections we will look at implementation details followed by results comparing the augmented reward case with the the naive sparse reward case and show a substantial improvement in learning rates using the same algorithms.

Algorithm 1: Sparse Reward Propagation using GCN

Input: $A, X, \alpha, \beta, w, \theta, s_{goal}$

`// α, β step sizes`

Output: Q^*_w, π^*_θ

`// Q and π are parametrized by w and θ`

1 Train the GCN using A and X
2 $\Phi(X) = \sigma(\hat{T}\ ReLU(\hat{T}XW^{(0)})W^{(1)})$
3 **for** $Episodes = 1, 2, 3...$ **do**
4 $\quad s_0 \leftarrow s, a_0 \sim \pi_\theta(.|s_0)$
5 $\quad$ **while** $s_t \neq s_{goal}$ **do**
6 $\quad\quad s_{t+1} \leftarrow (s_t, a_t)$
7 $\quad\quad a_{t+1} \sim \pi_\theta(.|s_{t+1})$
8 $\quad\quad \phi(s_{t+1}) \leftarrow \Phi(X[s_{t+1}, :])[0]$
9 $\quad\quad \phi(s_t) \leftarrow \Phi(X[s_t, :])[0]$
10 $\quad\quad R^{'}(s_t, a_t, s_{t+1}) \leftarrow R(s_t, a_t) + \phi(s_{t+1}) - \phi(s_t)$
11 $\quad\quad \delta^\pi_t \leftarrow R^{'}(s_t, a_t, s_{t+1}) + Q^\pi_w(s_{t+1}, a_{t+1}) - Q^\pi_w(s_t, a_t)$
12 $\quad\quad w \leftarrow w + \alpha\delta^\pi_t \frac{\partial Q^\pi_w}{\partial w}$
13 $\quad\quad \theta \leftarrow \theta + \beta\nabla \log \pi_\theta (a_t \mid s_t)\, \delta^\pi_t$
14 $\quad\quad s_t \leftarrow s_{t+1}$
15 $\quad\quad a_t \leftarrow a_{t+1}$

---

4 Implementation

4.1 Implementation and Training of GCN

In this section we provide details on the step 1 of the algorithm described in the previous section.

In our experiment we use a 2 layered GCN network as described in equation Eq.5. The output vector Φ is of size $N \times 2$, where N is the number of vertices in the graph (MDP). Φ can be thought of as a matrix that represents the probability of a state (each vertex is a state) being a non - rewarding state (1st column) and rewarding state (2nd column) respectively. Hence it can be used a used for reward shaping like this $\phi(s_{t+1}) = \Phi(X[s, :])[0]$.

To train the GCN we collected a set of trajectories by making the agent act more ran-

domly (every action in any state is equally likely). For each such trajectory we use the true labels of the state and the end state (non-rewarding state - 0, rewarding state - 1) and use Cross Entropy Loss on these labels to train the GCN.

Let the i represent the ith step in a chosen trajectory and let $i = 0$ be start and $i = T$ be the end of a trajectory. s_i is the ith state in the trajectory and r_i is the corresponding label. The data of a single trajectory for the GCN looks like $\{(s_0, r_0), (s_T, r_T)\}$ where

$$r_i = \begin{cases} 1 & s_i \; is \; goal \; state \\ 0 & otherwise \end{cases}$$

Next we move to implementing the GCN Model. We have chosen to use PyTorch do our experiments. As mentioned earlier we construct a 2-layered GCN with an output size of $N \times 2$. We use a non linear activation function like ReLU. The following is the code of the of the same in PyTorch.

So far we have assumed that the information about the entire MDP i.e the entire adjacency matrix is given to us apriori. However, for practical purposes storing and performing the eigen decomposition (for feature extraction) of the sparse adjaceny martix of the entire graph is computationally expensive and it may be the case that we do not know the MDP fully beforehand. So constructing the graph incrementally as the agent explores the MDP is more easier, computationally less expensive and prevents the GCN from overfitting.

4.2 Actor Critic Algorithm

Policy Gradient Methods introduce several policy - based reinforcement learning algorithms out of which actor critic is one. Policy Gradient Methods tend to perform better in Model Free environments compared to the value based methods as it can easily learn stochastic policies and is quite efficient in MDP's with continuous state spaces. In policy gradient methods the policy $\pi(a|s)$ is parametrized i.e $(\pi_\theta(a|s))$ where a is action and s is the state.

$$\pi_\theta(a|s) = P[a|s, \theta] \tag{7}$$

The goal of policy gradient methods is to find the best θ given $\pi_\theta(a|s)$. To evaluate the quality of the policy we have an objective function $J(\theta)$ which essentially tells us how good the policy is compared to other policies based on the parameter θ. For starters the most generally used objective function is

$$J(\theta) = V_{\pi_\theta}(s) \tag{8}$$

where the $V_{\pi_\theta}(s)$ denotes the value obtained upon following the policy π_θ. Now the goal is to find the θ that maximizes $J(\theta)$. This can be done using gradient ascent. This update step is where the method gets its name.

$$\theta = \theta + \alpha \frac{\partial J(\theta)}{\partial \theta} \tag{9}$$

where α is the step size. Now according to the policy gradient theorem we have,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log(\pi_\theta(a|s))\delta] \tag{10}$$

where,

$$\delta = R(s,a) + \gamma Q^\pi_w(s', a') - Q^\pi_w(s,a) \tag{11}$$

Using the above equations our policy gradient step becomes,

$$\theta = \theta + \alpha \nabla_\theta log(\pi_\theta(a|s))\delta \tag{12}$$

We lose the expectation since we follow an iterative algorithm and will be taking averages instead. The Q values in this equation also needs to evaluated on the current policy which needs a separate linear approximator or a neural network. These Q values gives us a measure of the quality of the policy.

$$Q^\pi_w(s,a) = w^T.\phi(s,a) \tag{13}$$

In our case we used a linear approximator for estimating Q values and in the above equation the vector w denotes the weights of the linear approximator and $\phi(s,a)$ is the feature vector corresponding to the state action pair (s,a). We used mean squared error loss $J(w)$ and one step temporal difference (TD(0)) approximation for the true Q-value. Upon using gradient descent the step sizes will be of the following form,

$Q^\pi(s,a) = R(s,a) + \gamma Q^\pi_w(s', a')$ - *One Step TD approximation*
$Q^\pi_w(s,a) = w^T.\phi(s,a)$
$\delta = R(s,a) + \gamma Q^\pi_w(s', a') - Q^\pi_w(s,a)$ - *TD Error*
$\nabla_w(Q^\pi_w(s,a)) = \phi(s,a)$

$J(w) = \mathbb{E}[(Q^\pi(s,a) - Q^\pi_w(s,a))^2]$ - *Loss Function*
$\Delta w = \frac{-1}{2}\beta \nabla_w J(w)$
$\Delta w = \beta(Q^\pi(s,a) - Q^\pi_w(s,a))\nabla_w(Q^\pi_w(s,a))$
$\Delta w = \beta(Q^\pi(s,a) - Q^\pi_w(s,a))\nabla_w(Q^\pi_w(s,a))$
$\Delta w = \beta\delta\phi(s,a)$

In essence the critic updates Q-value function weights, and the actor updates policy parameters θ guided by the Q values from the critic.

Algorithm 2: Q Value Actor Critic

1 Initialise s, θ, w
2 Sample action a $\sim \pi_\theta$
3 **for** $Episodes = 1, 2, 3...$ **do**
4   **while** $s \neq s_{goal}$ **do**
5         $s', r \leftarrow (s, a)$
6         $a' \sim \pi_t heta(.|s')$
7         $\delta \leftarrow r + \gamma Q_w^{\pi_\theta}(s', a')$ - $Q_w^{\pi_\theta}(s, a)$
8         $\theta \leftarrow \alpha \nabla_\theta log(\pi_\theta(a|s)) Q_w(s, a)$
9         $w \leftarrow w + \beta \delta \phi(s, a)$
10         $a \leftarrow a'$
11         $s \leftarrow s'$

4.3 Hyper Parameters

In this section we provide details on the choice of Hyperparameters. Choosing the correct set of Hyperparameters plays a crucial role in how effectively we can train any Deep learning algorithms. We ran our experiments on GoogleColab and used the following list of Hyperparameters for the same.

α	0.01
β	0.005
$Episodes$	2000
GCN_{lr}	0.01
GCN_{Epochs}	40
GCN_{hidden}	16
γ	1

GCN_{lr} is the learning rate or step size for training the GCN over GCN_{Epochs} number of epochs. The W matrix in the equation (3) is of shape $M^{features} \times GCN_{hidden}$. The $\gamma = 1$ as mentioned earlier.

4.4 Approximation of the MDP as a graph and Proto Value Functions

We are constructing the graph according to the grid world environment. Each node in the graph represents a state in the MDP, and the edges represent the transitions from one state to another upon a specific action. Let the graph be $G = (V, E)$ where V is the set of nodes and E is the set of edges. Equation for calculating the graph laplacian:

$$L_{norm} = I - D^{-1/2} A D^{-1/2} \tag{14}$$

where A is the adjacency matrix of the graph, and D is the diagonal matrix where each diagonal element is the sum of the corresponding row in the adjacency matrix, I is the

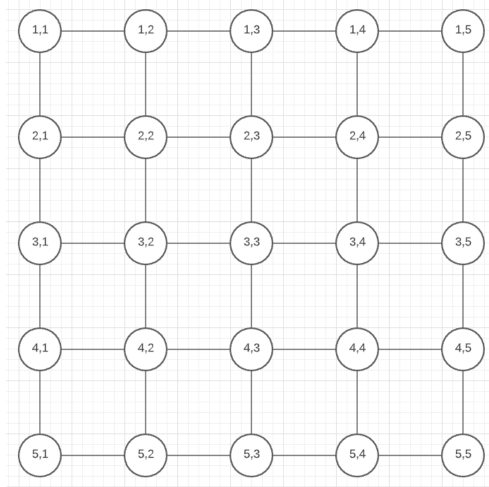


Figure 4: A 5x5 grid environment in a graph structure

identity matrix. The eigen vectors of L_{norm} form the features of the MDP which is also in this case the graph.

```
def shortest_dist(n, m, goal_states):
    # Adjacency Matrix
    adj = np.zeros((n * m, n * m))
    D = np.zeros((n * m, n * m))

    for i in range(n):
        for j in range(m):
            currcod = i * m + j
            north, east, south, west = m * (i - 1) + j, m * (i) + j +
                1, m * (i + 1) + j, m * (i) + j - 1

            if i - 1 >= 0:
                adj[currcod, north] = 1
            if j + 1 <= m - 1:
                adj[currcod, east] = 1
            if i + 1 <= n - 1:
                adj[currcod, south] = 1
            if j - 1 >= 0:
                adj[currcod, west] = 1

            D[currcod, currcod] = sum(adj[currcod, :])

    D_hat = la.fractional_matrix_power(D, -0.5)
    L_norm = np.identity(n * m) - np.dot(D_hat, adj).dot(D_hat)
    eigvals, features = la.eig(L_norm)
```

4.5 CodeBase

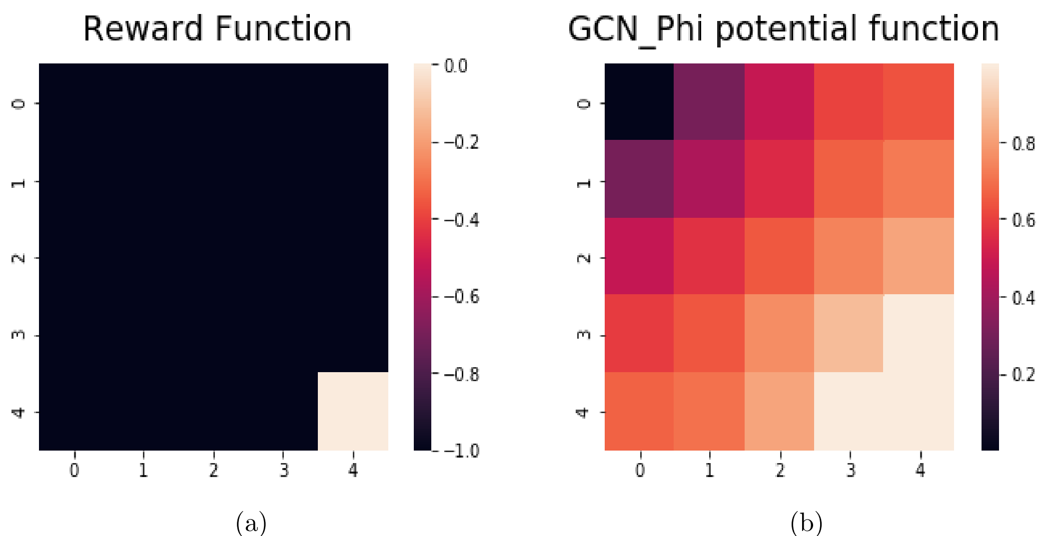
The entire code base can be found in these github repositories each having our own style of implementation of the same:

https://github.com/RaswanthMurugan20/Gridworld_PhiGCN
https://github.com/Vamsi995/Reward-Propagation

5 Results

5.1 Reward Propagation & Scalability

In section 2.1 when we described the experimental setup we also defined the reward function R for our experiments Eq.2. This reward function as mentioned earlier is a sparse reward function. First we show that the Φ function learnt from the GCN does in fact propagate the information about the reward from the goal state to the other states, for example see Fig.5. One might also notice that the plot of the Φ function looks like diffusion of some quantity from the goal state. We think this is due to the graph laplacian that works like a discretized diffusion operator.

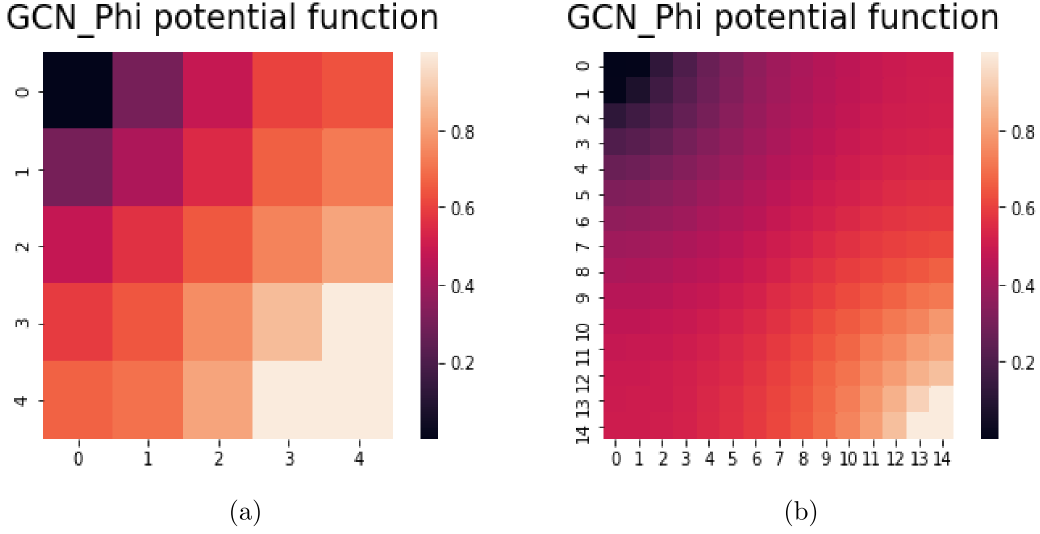


(a) (b)

Figure 5: (a) is a plot of the unaugmented reward function R in Eq.2. (b) is a plot of the potential function learnt from the GCN. It can be clearly seen that the phi function learnt is a result of propagating the reward function from the goal state to all other states. The information about the reward waiting at the goal state has been propagated to the states.

The other important aspect of using GCN for reward shaping was to automate the process of creating a potential function that scales. This is shown to be true in Fig.6.

Therefore, we can use the GCN method to extend to larger MDPs.



(a) (b)

Figure 6: (a) is potential function for 5x5 grid, (b) is a potential function for 15x15 grid

So far the MDP setup we have discussed assumes the reward is at one corner of the grid which is the goal state. But this need not be the case always. Depending on the problem we may the reward at a different state or even have multiple rewards in different states. It turns out the GCN learns Φ in different cases with minimal changes during the training process. The results are shown in Fig.7.

Therefore we conclude that the Φ function from the GCN manages to propagate information from rewarding states to non-rewarding states and is able to scale and work for different sparse reward functions.

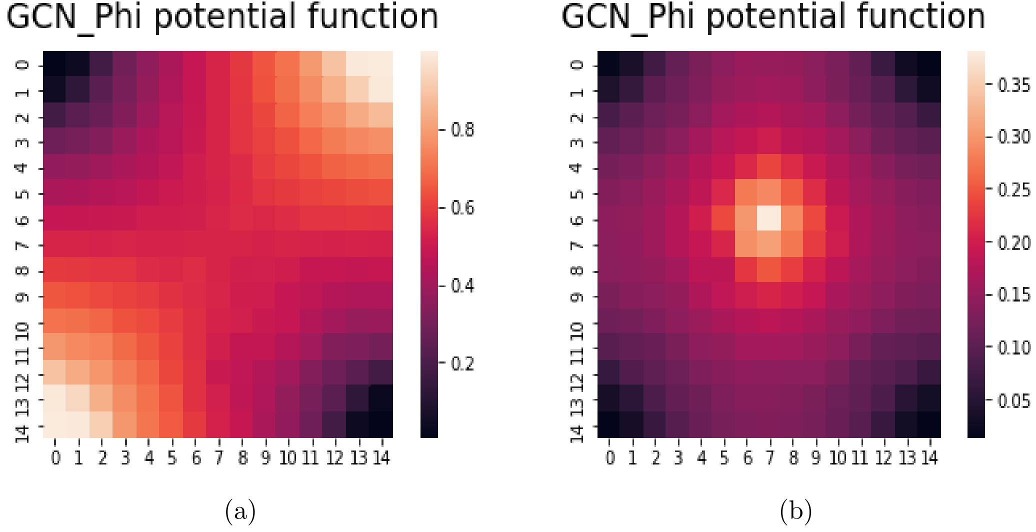
5.2 Performance

The final question we need to ask now is whether if the reward shaping using such a Φ fairs better when compared to the unaugmented sparse reward case. For this purpose we first define two metrics over which we perform the comparison.

The first metric is called **Regret**, denoted by ρ. It is total sum of the difference between the optimal reward and the obtained reward over all the iterations of the algorithm. Formally regret is defined as

$$\rho\left(T, s_{\text{start}}\right) := TR^{*} - \sum_{t=1}^{T} R_{total}^{t} \tag{15}$$

Here T stands for the total number of episodes (or) epochs (or) the number of times the



(a) (b)

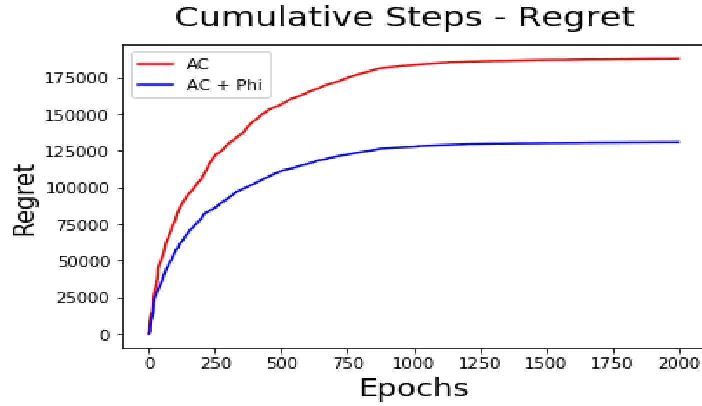
Figure 7: (a) the MDP has 2 goals states at opposite corners of the grid and the Φ function has managed to propagate the reward as expected,(b) the MDP has a goal state at the center of the grid and Φ propagates from the center to the other states.

agents is made to go from s_{start} which is the start state to s_{goal} which is the goal state. R^* is the maximum reward the agent can possibly get starting from s_{start} and R^t_{total} is the reward obtained at the end of the tth episode during training. It is easy to see that ρ is always a non-negative quantity because the $R^* \geq R^t_{total} \forall t$ by definition. So if the agent is behaving optimally then ρ is 0 (or) the agent has no regrets.

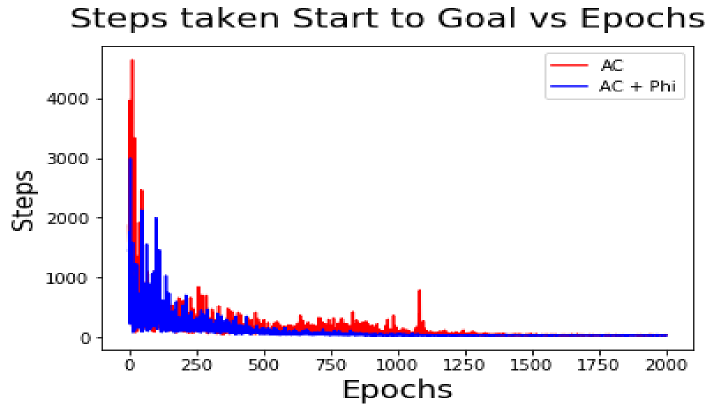
Now we compare the regret plots between the shaped reward case and the unshaped reward case. The RL algorithm of choice is the Actor-Critic algorithm. Actor-Critic is chosen in both cases to make comparison possible. The result is shown in Fig.8(a). The blue line (Actor critic with shaped rewards) flattens faster than the red line (Actor Critic with sparse rewards) and flattening implies the obtained rewards R^t_{total} is equal to the maximum possible reward $R*$ or in other works flattening happens when the agent begins to start behaving optimally. This means agent learns the optimal behaviour faster when the sparse rewards are augmented with the potential function learnt from the GCN.

The other metric we will use to compare the two cases is the number of steps taken to reach the goal from the start state s_{start} (In these experiments we fix the start state to the top left corner of the grid and the goal at the lower right corner). The results can be seen in Fig.8(b). It can be seen that the blue points reach the minimum steps faster than the red points. This is consistent with out inference from the regret plot.

We can therefore conclude by using a GCN for reward shaping in sparse reward systems,



(a)



(b)

Figure 8: The red line regret plot using unaugmented rewards while the blue line is using the rewards shaped by Φ from GCN.(a) is a regret vs epochs plot.(b) is a total number of steps vs epochs plot.

the RL algorithms performs faster. This method can scale and does provide a way to perform reward shaping in an end-to-end fashion.

6 Future Directions

Let us take a closer look on image Fig.9. Here the red square is the goal state and the blue square is the agent. The goal is once again to find the shortest path to the goal state with sparse rewards. Now this problem has been tackled in this report for the case of a single grid. The 4 - connected grid worlds can also be solved using the algorithm in the report however, the adjacency matrix for this case is highly sparse and it becomes even more harder to compute features for each state.

One possible way to tackle this problem would be to exploit the symmetries in such a graph using unsupervised graph clustering algorithms and perform feature matrix computation and generate potential function for each of the 4 grid worlds instead of operating on the whole MDP as a single unit. This way we can exploit the symmetries in the MDP to reduce the computation significantly.

So far we have primarily shown most of the work on gridworld where converting the MDP to a graph can be done trivially but this is not the case in many other MDPs, take for instance the robotic arm example which we talked about in the introduction. Trying to see if these results hold even in these cases will make this approach more generic and therefore more useful.

The above mentioned problems are interesting future directions we would like to explore.

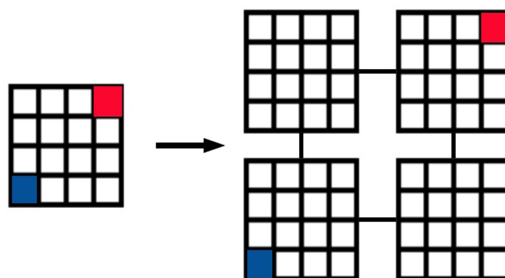


Figure 9: Blue - Agent, Red - Target

7 Contributions

1. Literature Survey to understand the various aspects required to put the algorithm together such as Reward Shaping Theory, Graph Convolution Networks, Latent Feature Extraction. [Raswanth and Vamsi]
2. Approximation of the MDP as a graph to calculate the graph laplacian. [Raswanth]
3. Implementing Graph Convolution Network. [Vamsi]
4. Training the Graph Convolution Network. [Raswanth]
5. Using the state probability functions obtained from the trained GCN for potential based reward shaping. [Vamsi]
6. Incorporate the potential functions into a policy iteration algorithm to solve an optimization problem on approximated MDP, first using tabular representations - [Vamsi] and then using then function approximators - [Raswanth].

7. Running the algorithm on grid-world with different reward functions to see different optimal behaviors. [Vamsi and Raswanth]

8. Comparing the performance with existing benchmark algorithms such as the actor-critic algorithms. [Vamsi] Optimizing the algorithmic hyperparameters through experimentation. This includes reading and implementing various latent feature representation methods like Proto-Value functions and DeepWalk.[Raswanth]

9. Insights into the future line of work on the proposed method [Vamsi and Raswanth]

8 Conclusion

Graph Convolutional Networks provide an end-to-end method to perform reward shaping in sparse reward Reinforcement learning systems. They leverage the message passing mechanism to pass the message about the rewards to other state thereby making the learning easier. Some open questions remain that point to newer directions for research.

9 Acknowledgement

We would like to sincerely thank the Open Ended Lab Project Committee for giving this opportunity in spite of the pandemic. We would also like to thank Dr.Chandra Shekhar for his time and insightful inputs that made this project possible.

References

[1] Patrick Mannion et al. "Policy Invariance under Reward Transformations for Multi-Objective Reinforcement Learning". In: *Neurocomputing* 263 (Nov. 2017), pp. 60–73. DOI: 10.1016/j.neucom.2017.05.090.

[2] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *Proceedings of the 5th International Conference on Learning Representations* (ICLR). ICLR '17. Palais des Congrès Neptune, Toulon, France, 2017. URL: https://openreview.net/forum?id=SJU4ayYgl.

[3] Sridhar Mahadevan. "Proto-Value Functions: Developmental Reinforcement Learning". In: *Proceedings of the 22nd International Conference on Machine Learning*. ICML '05. Bonn, Germany: Association for Computing Machinery, 2005, 553â560. ISBN: 1595931805. DOI: 10.1145/1102351.1102421. URL: https://doi.org/10.1145/1102351.1102421.

[4] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: (2013). cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013. URL: http://arxiv.org/abs/1312.5602.

[5] Vijay Konda and John Tsitsiklis. "Actor-Critic Algorithms". In: *SIAM Journal on Control and Optimization*. MIT Press, 2000, pp. 1008–1014.

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[7] David Silver. *Introduction to Reinforcement Learning with David Silver*. DeepMind,Youtube. 2015. URL: https://youtube.com/playlist?list=PLqYmG7hTraZBiG_XpjnPrSNw-1XQaM_gB.

[8] Emma Brunskill. *CS234 : Reinforcement Learning — Winter 2019*. Stanford,Youtube. 2019. URL: https://youtube.com/playlist?list=PLoROMvodv4rOSOPzutgyCTapiGlY2Nd8u.

[9] Pierre-Yves Oudeyer and Frederic Kaplan. “What is Intrinsic Motivation? A Typology of Computational Approaches”. In: *Frontiers in neurorobotics* 1 (Feb. 2007), p. 6. DOI: 10.3389/neuro.12.006.2007.

[10] Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. “Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping”. In: *Proceedings of the Sixteenth International Conference on Machine Learning*. ICML ’99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, 278â287. ISBN: 1558606122.

[11] Sridhar Mahadevan. “Proto-Value Functions: Developmental Reinforcement Learning”. In: *Proceedings of the 22nd International Conference on Machine Learning*. ICML ’05. Bonn, Germany: Association for Computing Machinery, 2005, 553â560. ISBN: 1595931805. DOI: 10.1145/1102351.1102421. URL: https://doi.org/10.1145/1102351.1102421.

[12] Sridhar Mahadevan and Mauro Maggioni. “Proto-value Functions: A Laplacian Framework for Learning Representation and Control in Markov Decision Processes”. In: *Journal of Machine Learning Research* 8.74 (2007), pp. 2169–2231. URL: http://jmlr.org/papers/v8/mahadevan07a.html.