# Simple Grid World task for Reinforcement Learning

Alena Guseva

***Abstract* — In this project been studding a different way of solving small grid world example. Grid World helps easier understand the Reinforcement Learning framework, algorithms and challenges involved in solving RL tasks. Planning, control and performance evaluation on a smaller task as Grid World gave extra confidence in their correctness before running a lengthier training procedure for bigger tasks in the future. Goal of this project is to gain deeper understanding of Reinforcement Learning by studying and developing algorithm which will be used in the future bigger projects.**

**Keywords: artificial intelligence, computer games, grid world, reinforcement learning, search path, machine learning.**

## I. Introduction

Reinforcement learning is an area of machine learning concerned with how software agents should to take actions in an environment in order to maximize some concept of cumulative reward. Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning.

Grid world is one of the simplest RL examples, we define a game where a robot learns to find an optimal path in a room to a target while avoiding traps. It is a little gym where one can practice a different algorithm and use different knowledge gain during class to improve it understanding and gain confidence in using such. For small problems it is also possible to compare learnt policies with the optimal one by evaluating the number of differences.

## II. Methodology

Program language Python with Numpy libraries been chosen for realization this problem. Final code includes two classes: Class Environment and Class Agent. Class Environment have information about dynamic of the grid – which action of agent allowed on any squares of the grid, how environment will act at any moment and what reward agent will receive. Class Agent has information about agents learning parameters and policy.
NumPy is main package for computing with Python, which contains valuable linear algebra, Fourier transform, and random number capabilities, powerful N-dimensional array object and many other useful things.

A.A. Guseva. Master student of Electric and Computer Engineering Department, Rowan University. Glassboro, NJ 08028 USA (e-mail: gusevaa2@students.rowan.edu)

### *2.1. The agent-environment model*

An RL problem can be specified formally by introducing the agent-environment framework, conceptually shown on Figure 2 from the book, where the agent is a decision maker controlled by an RL algorithm and everything it does not have absolute control over or only interacts with is considered to be part of the environment. The environment provides the agent with its state and an immediate reward.

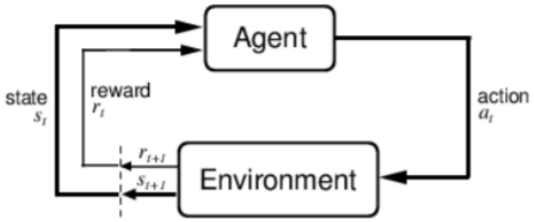


Fig. 1. The agent-environment interaction

*2.1.1 State space*. At each time step *t*, the environment provides an agent with its current state, $s_t \in S$, where $S$ is a set of possible states.

For this project 5x5 grid been applied (see **fig 2**) which response to 25 squares of possible position of agent. Different obstacles have been introduced for this grid (wall, block, teleport, sad_terminal, happy_terminal and later star.

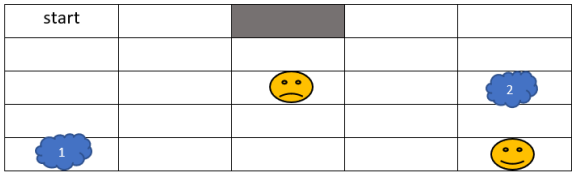


Fig. 2. Grin World space

*2.1.2 Action space*. Similarly, to the state space, at each time step *t* the agent chooses an action $a_t \in A$, where A is a set of actions available at state $S_t$.

- agent can move (up, down, left, right).
- agent cannot go outside of grid
- agent moves reliable 80% of the time, 20% will move randomly
- On grey – nothing happens, just bounce back.

- Cloud – "take a chance" - 50% - will teleport from 1 to 2, 50% will stay at the same place

*2.1.3 Reward.* The main part of the Reinforcement Learning it is a reward signal, numerical feedback from environment to agent.

- each step (-0.1)
- ☺ (+10)
- ☹ (-10)
- ★ (0.2)

*2.1.4 Returns.* Total discounted reward from time step *t*.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where γ is a discount factor. Small value of γ puts more meaning on immediate rewards whereas γ = 1 makes the agent far-sighted.

*2.1.5 Policy.* Agent behavior from state to action at each particular time-step *t* which depends on probability distribution $\pi_t(s, a)$. Where *s* – state and *a* – action.

### *2.2. Markov decision processes (MDP)*

Provide a mathematical framework for decision-making characterized by a set of states. Every state has several actions (from which the decision maker must choose) and transitions to a new state. New state is only dependent on the current state and independent of all previous states and outcomes are partly random and partly under the control of a decision maker. MDPs are useful for studying a wide range of optimization problems solved via dynamic programming and reinforcement learning.

A Markov Decision Process (MDP) is a 5-tuple (S, A, P, R, γ) where S is a set of states A is a set of actions P(S,A,S') is the transition model R(S) is the reward function s' is the next state

*2.2.1. Prediction problem.* Evaluate the future - given the policy. In this project for prediction proposes been using two policies – fixed and random. Random policy states that we have equal probability to take any available action at every state. In Fix policy at each state, the action is fixed. For our example fixed policy illustrated on Fig.3.

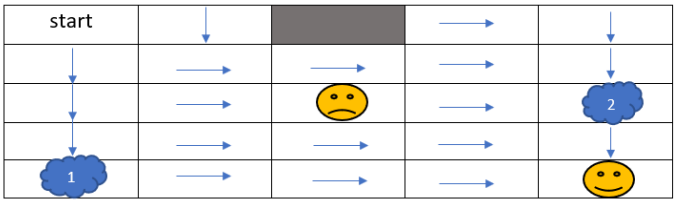


Fig. 3. Fixed policy

*Policy evaluation* – computes the value function for policy by using Bellman equations. In our Grid World problem dynamic of environment is known that is means that iterative solution methods are most suitable for this past. Each iteration of iterative policy evaluation *backs up* the value of every state once to produce the new approximate value function $V_{k+1}$. All the backups based on all possible next states not on a sample next state. Bellow complete algorithm for iterative policy evaluation which being used for the project.

Input $\pi$, the policy to be evaluated
Initialize $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx V^\pi$

Initially, every value set to zero. Then over all states in loop we are applying the Bellman equation to compute the value at each state. The loop finishes when all the values converge toward a stable value (which mean the difference between the new computed value and the previous one is very small).

*2.2.2 Control problem.* Policy improvement – finding the best policy that maximizes future reward. For each state we have a finite set of actions (Up, Down, Left, Right), we check each one of them until we get best value.

$$\forall a \in A, s \in S \quad Q^*(s,a) = \max_\pi Q^\pi(s,a)$$

### *2.3. Computing the optimal Q function*

To find an optimal solution we can use two ways. One of them depends on state value function, V(s), and another on state-action value function Q(s,a). If using V(s)could be an appropriate solution for a smaller task however for the larger state space it becomes computational expensive and infeasible. Several methods depends on task, to find an optimal Q function exists.

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s'} P(s,a,s') \max_{a'} Q^\pi(s',a')$$

Where P(s,a,s') is the probability of a transition from s to s' by taking action a, and $Q^\pi$(s,a) is stored for any combination of state and action.

*2.3.1. Dynamic programming.*

DP uses iterative scheme to calculate the solution Q to the Bellman optimality equation

$$Q^\pi_{(k+1)} \rightarrow R + \gamma P^\pi Q^\pi_{(k)}$$

*Policy iteration and Value iteration.* In Policy Iteration - Randomly select a policy and find value function corresponding to it then find a new (improved) policy based on the previous value function, and so on this will lead to optimal policy. Policy evaluation -> Policy improvement. In Value Iteration - Randomly select a value function, then find a new (improved) value function in an iterative process, until reaching the optimal value function, then derive optimal policy from that optimal value function. Optimal value function -> Optimal policy.

*2.3.2. Monte Carlo methods.*

MC method can be useful in situations where model of the environment are known with transition probabilities P(s,a,s'). Our task with a small grid is a good example of it. MC methods is a class of algorithms that rely on repeated random sampling

to obtain numerical results. To find an optimal solution we calculating a sample mean from number of episodes N:

$$\bar{V}^{\pi}(s) = \frac{1}{N}\sum_{i=1}^{N} G_{i,\ s}$$

where total reward finding recursively for the number of episodes. $G_t = R_{t+1} + \gamma G_{t+1}$

First visit MC average returns only for first time s is visited in an episode. Every visit MC average return for every time s is visited in an episode

**First-visit MC policy evaluation (returns $V \approx v_\pi$)**

Initialize:
$\pi \leftarrow$ policy to be evaluated
$V \leftarrow$ an arbitrary state-value function
$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:
Generate an episode using $\pi$
For each state $s$ appearing in the episode:
$G \leftarrow$ return following the first occurrence of $s$
Append $G$ to $Returns(s)$
$V(s) \leftarrow$ average($Returns(s)$)

### *2.3.3. Temporal Differences algorithms*

TD algorithms combine good aspects of both DP and Monte Carlo methods. They update their estimates based in part on other learned estimates without waiting for them to converge as in Policy Iteration method. TD algorithm learn directly from interacting with the environment without knowing model of it.

One of the examples for TD learning algorithm is Q-Learning. This method is off-policy algorithm which mean we don't need to know policy or model of the environment from beginning and going to find an optimal state-action value function by approximation.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right]$$

In other words, at each step it tries to decrease the temporal difference between the current state-action estimate $Q(s_t, a_t)$ and its stochastic expected value based on the immediate reward and next state's estimated value.

**Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$**

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
Initialize $S$
Loop for each step of episode:
Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
Take action $A$, observe $R$, $S'$
$Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
$S \leftarrow S'$
until $S$ is terminal

## III. RESULTS

Different experiments been used during work on this project. Changes in the Grid, dynamic, behavior, in different parameters and values of learning rate, discount, etc. Some of them will be added bellow.

```
Value function
+-------+-------+-------+-------+-------+
| -2.15 | -2.40 | +0.00 | -1.74 | -1.35 |
+-------+-------+-------+-------+-------+
| -2.38 | -3.18 | -5.04 | -2.51 | -1.26 |
+-------+-------+-------+-------+-------+
| -2.59 | -4.33 | +0.00 | -3.13 | -0.34 |
+-------+-------+-------+-------+-------+
| -1.92 | -2.38 | -3.09 | +0.06 | +3.25 |
+-------+-------+-------+-------+-------+
| -1.42 | -1.24 | -0.32 | +3.26 | +0.00 |
+-------+-------+-------+-------+-------+
```

Value function for random policy (equal probability for all allowed actions)

```
Value function
+-------+-------+-------+-------+-------+
| +4.78 | +5.31 | +0.00 | +0.00 | +7.29 |
+-------+-------+-------+-------+-------+
| +5.31 | +5.90 | +6.56 | +7.29 | +8.10 |
+-------+-------+-------+-------+-------+
| +5.90 | -10.00 | +0.00 | +8.10 | +9.00 |
+-------+-------+-------+-------+-------+
| +6.56 | +7.29 | +8.10 | +9.00 | +10.00 |
+-------+-------+-------+-------+-------+
| +7.29 | +8.10 | +9.00 | +10.00 | +0.00 |
+-------+-------+-------+-------+-------+
```

Value function for fixed policy from Fig.3.

```
Value function
+-------+-------+-------+-------+-------+
| +0.56 | +0.89 | +0.00 | +2.55 | +3.05 |
+-------+-------+-------+-------+-------+
| +3.67 | +3.70 | +4.70 | +6.63 | +7.27 |
+-------+-------+-------+-------+-------+
| +4.31 | +4.10 | +0.00 | +7.54 | +8.42 |
+-------+-------+-------+-------+-------+
| +5.11 | +6.09 | +7.22 | +8.50 | +9.73 |
+-------+-------+-------+-------+-------+
| +5.71 | +6.79 | +8.05 | +9.51 | +0.00 |
+-------+-------+-------+-------+-------+
```

Value function if agent move reliable 80% of the time and 20% agent goes right from desire. Also, negative reward been added. In this case we can see that agent will avoid passing "sad tile" from it's left.

```
[episode 80/500] eps = 0.358 -> iter = 10, rew = 9.1
[episode 90/500] eps = 0.324 -> iter = 8, rew = 9.3
[episode 100/500] eps = 0.293 -> iter = 8, rew = 9.3
[episode 110/500] eps = 0.265 -> iter = 10, rew = 9.1
[episode 120/500] eps = 0.240 -> iter = 6, rew = 9.5
[episode 130/500] eps = 0.217 -> iter = 10, rew = 9.1
[episode 140/500] eps = 0.196 -> iter = 4, rew = -0.4
[episode 150/500] eps = 0.177 -> iter = 8, rew = 9.3
[episode 160/500] eps = 0.160 -> iter = 8, rew = 9.3
[episode 170/500] eps = 0.145 -> iter = 6, rew = 9.5
[episode 180/500] eps = 0.131 -> iter = 10, rew = 9.1
[episode 190/500] eps = 0.119 -> iter = 8, rew = 9.3
[episode 200/500] eps = 0.107 -> iter = 8, rew = 9.3
[episode 210/500] eps = 0.097 -> iter = 8, rew = 9.3
[episode 220/500] eps = 0.088 -> iter = 6, rew = 9.5
[episode 230/500] eps = 0.079 -> iter = 10, rew = 9.1
[episode 240/500] eps = 0.072 -> iter = 6, rew = 9.5
[episode 250/500] eps = 0.065 -> iter = 6, rew = 9.5
[episode 260/500] eps = 0.059 -> iter = 6, rew = 9.5
[episode 270/500] eps = 0.053 -> iter = 8, rew = 9.3
```

Q-learning algorithm trying different combinations of moves to get best result.

```
[episode 320/500] eps = 0.032 -> iter = 6, rew = 9.5
[episode 330/500] eps = 0.029 -> iter = 6, rew = 9.5
[episode 340/500] eps = 0.026 -> iter = 6, rew = 9.5
[episode 350/500] eps = 0.024 -> iter = 6, rew = 9.5
[episode 360/500] eps = 0.021 -> iter = 8, rew = 9.3
[episode 370/500] eps = 0.019 -> iter = 8, rew = 9.3
[episode 380/500] eps = 0.018 -> iter = 6, rew = 9.5
[episode 390/500] eps = 0.016 -> iter = 8, rew = 9.3
[episode 400/500] eps = 0.014 -> iter = 6, rew = 9.5
[episode 410/500] eps = 0.013 -> iter = 6, rew = 9.5
[episode 420/500] eps = 0.012 -> iter = 6, rew = 9.5
[episode 430/500] eps = 0.011 -> iter = 8, rew = 9.3
[episode 440/500] eps = 0.010 -> iter = 8, rew = 9.3
[episode 450/500] eps = 0.010 -> iter = 6, rew = 9.5
[episode 460/500] eps = 0.010 -> iter = 6, rew = 9.5
[episode 470/500] eps = 0.010 -> iter = 6, rew = 9.5
[episode 480/500] eps = 0.010 -> iter = 6, rew = 9.5
[episode 490/500] eps = 0.010 -> iter = 8, rew = 9.3
[episode 500/500] eps = 0.010 -> iter = 6, rew = 9.5
```

Agent choosing to through “teleport” even with 50% chance that it will work.

```
[[2 2 0 0 2]
 [2 2 3 2 2]
 [2 2 0 2 2]
 [2 2 1 2 2]
 [1 1 1 1 0]]

 action['up'] = 0
 action['right'] = 1
 action['down'] = 2
 action['left'] = 3
```

Optimal policy without extra reward “star” with teleport

```
[episode 460/500] eps = 0.010 -> iter = 8, rew = 9.3
[episode 470/500] eps = 0.010 -> iter = 6, rew = 9.5
[episode 480/500] eps = 0.010 -> iter = 8, rew = 9.3
[episode 490/500] eps = 0.010 -> iter = 8, rew = 9.3
[episode 500/500] eps = 0.010 -> iter = 10, rew = 9.1

Greedy policy(y, x):
[[1 2 0 2 3]
 [1 2 1 1 2]
 [1 2 0 1 2]
 [1 1 1 1 2]
 [1 1 1 1 0]]
```

Optimal policy without teleport

```
[episode 460/500] eps = 0.010 -> iter = 6, rew = 9.5
[episode 470/500] eps = 0.010 -> iter = 6, rew = 9.5
[episode 480/500] eps = 0.010 -> iter = 6, rew = 9.5
[episode 490/500] eps = 0.010 -> iter = 10, rew = 9.7
[episode 500/500] eps = 0.010 -> iter = 8, rew = 9.6

Greedy policy(y, x):
[[2 2 0 2 2]
 [2 2 1 1 2]
 [2 2 0 1 2]
 [2 1 2 1 2]
 [1 1 1 1 0]]
```

Optimal policy with extra reward at [4,1] = 0,2. Still choose to go to teleport even with 50% chance that it will move agent and get less reward.

```
[episode 440/500] eps = 0.010 -> iter = 27054, rew = 65558.0
[episode 450/500] eps = 0.010 -> iter = 259180, rew = 628452.8
[episode 460/500] eps = 0.010 -> iter = 57142, rew = 138554.5
[episode 470/500] eps = 0.010 -> iter = 71798, rew = 174018.0
[episode 480/500] eps = 0.010 -> iter = 43450, rew = 105325.3
[episode 490/500] eps = 0.010 -> iter = 15804, rew = 38230.2
[episode 500/500] eps = 0.010 -> iter = 43770, rew = 106124.6

Greedy policy(y, x):
[[1 2 0 2 2]
 [1 2 3 2 2]
 [1 2 0 2 2]
 [1 2 2 2 3]
 [1 0 3 3 0]]
```

Add extra reward 3. Agent will try to do as many iterations to hit reward as possible. Only reason it is break loop that agent walks as desire only 80% of the time.

Interesting to see that if we are going to add an extra reward on top of “sad” (probability of passing that tile is only 80% and 20% it will fall into “sad”). If extra reward [0.1, 0.5] agent will ignore that policy, but if extra reward [0.6, 1] agent will choose to risk and step on that tile.

```
[episode 430/500] eps = 0.011 -> iter = 8, rew = 10.0
[episode 440/500] eps = 0.010 -> iter = 8, rew = 10.0
[episode 450/500] eps = 0.010 -> iter = 8, rew = 10.0
[episode 460/500] eps = 0.010 -> iter = 8, rew = 10.0
[episode 470/500] eps = 0.010 -> iter = 8, rew = 10.0
[episode 480/500] eps = 0.010 -> iter = 8, rew = 10.0
[episode 490/500] eps = 0.010 -> iter = 8, rew = 10.0
[episode 500/500] eps = 0.010 -> iter = 8, rew = 10.0

Greedy policy(y, x):
[[1 2 0 2 2]
 [1 1 1 1 2]
 [1 0 0 0 2]
 [0 0 1 0 2]
 [2 1 0 1 0]]
```

Optimal policy with extra reward 0.6 at [1,2] (on top of “sad”)

## IV. Discussion and Conclusions

During this project was created a unique Grid World with an agent. Agent been train on 500 iteration using Q-learning algorithm. To make sure agent will perform it best, it been tested on all different combinations of rewards, obstacles, dynamics etc.

One of the imperfections in code I notice when extra reward more than 1, then agent trying to make as many loops as possible to collect a bigger reward. After been added a

punishment (-100) if agent will try to make more than 50 steps in 1 game, but it did not stop agent from doing loops. See below.

```
-100 reward
[episode 500/500] eps = 0.010 -> iter = 50, rew = -56.7

Greedy policy(y, x):
[[1 2 0 0 2]
 [1 2 3 2 2]
 [2 2 0 1 2]
 [1 2 3 1 2]
 [1 0 3 1 0]]

 action['up'] = 0
 action['right'] = 1
 action['down'] = 2
 action['left'] = 3
```

Another interesting moment, as it been mentioned earlier. With reward more than 0.5 agent will try to take risk walking on top of reward -10, but if we will increase that punishment to -100, agent will change his policy back on teleport and assume that that extra reward does not worth the risk.

For the feature improvement of Grid World project one can add a different method of training an agent, by using different algorithm. Another option, obstacles can me moved over time (for example wall can be moved every 5 time-steps). Another improvement could be adding a second agent way that they will compete to get best reward. All of those improvement can be a topic for the future study.

## REFERENCES

[1] Sutton R. S., Barto A. G. Reinforcement Learning: An Introduction. The MIT Press Cambridge, Massachusetts London, England. http://webdocs.cs.ualberta.ca/~sutton/book/ebook/the-book.html

[2] Алгоритмы поиска пути. Режим доступа: http://pmg.org.ru/ai/stout.htm#listing1

[3] M. L. Puterman, Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons, 2009, vol. 414.

[4] Wikipedia
https://en.wikipedia.org/wiki/Markov_decision_process
https://en.wikipedia.org/wiki/Reinforcement_learning