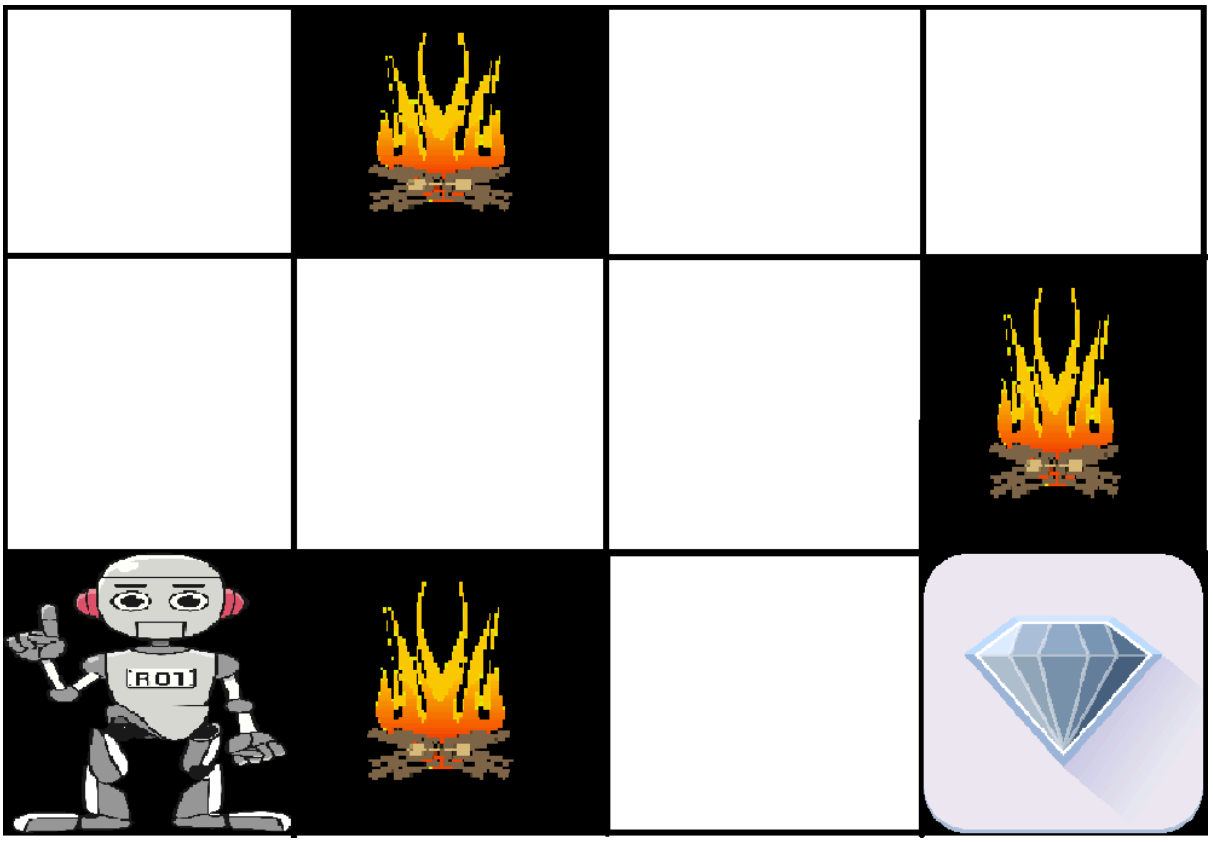
Reinforcement Learning: Reinforcement learning is an area of Machine Learning. It is about taking suitable action to maximize reward in a particular situation. In reinforcement learning, there is no answer but the reinforcement agent decides what to do to perform the given task. Without a training dataset, it is bound to learn from its experience. The data is accumulated from machine learning systems that use a trial-and-error method.



The above image shows the robot, diamond, and fire. The robot's goal is to get the diamond reward and avoid the hurdles that are fired. The robot learns by trying all the possible paths and then choosing the path that rewards him with the least hurdles. Each right step will give the robot a reward and each wrong step will subtract the reward of the robot. The total reward will be calculated when it reaches the final reward which is the diamond.

Elements of Reinforcement Learning

Reinforcement learning elements are as follows:

1. Policy
2. Reward function
3. Value function
4. Model of the environment

Policy: Policy defines the learning agent's behavior for a given period. It is a mapping from perceived states of the environment to actions to be taken when in those states.

Reward function: The reward function is used to define a goal in a reinforcement learning problem. A reward function is a function that provides a numerical score based on the state of the environment

Value function: Value functions specify what is good in the long run. The value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state.

Model of the environment: Models are used for planning.

```
import gym
import numpy as np


# Create the CartPole-v1 environment
env = gym.make('CartPole-v1')


# Parameters
alpha = 0.1  # Learning rate
gamma = 0.99  # Discount factor
```

```
epsilon = 0.1  # Exploration rate
num_episodes = 1000

# Discretize the state space
num_buckets = (1, 1, 6, 3)  # Cart position, cart velocity, pole angle, pole angular velocity
buckets = [
    np.linspace(env.observation_space.low[i], env.observation_space.high[i], num=num_buckets[i])
    for i in range(len(num_buckets))
]

# Initialize Q-table
action_size = env.action_space.n
q_table = np.zeros(num_buckets + (action_size,))

# Function to discretize state
def discretize_state(state):
    discrete_state = []
    for i in range(len(state)):
        discrete_state.append(np.digitize(state[i], buckets[i]) - 1)
    return tuple(discrete_state)

# Training loop
for episode in range(num_episodes):
    state = env.reset()
    state = discretize_state(state)
    done = False

    while not done:
```

```
        # Choose action using epsilon-greedy policy
        if np.random.rand() < epsilon:
            action = env.action_space.sample()  # Explore
        else:
            action = np.argmax(q_table[state])  # Exploit

        # Take action and observe the next state and reward
        next_state, reward, done, _ = env.step(action)
        next_state = discretize_state(next_state)

        # Update Q-table using Bellman equation
        q_table[state + (action,)] = (1 - alpha) * q_table[state +
(action,)] + \
            alpha * (reward + gamma * np.max(q_table[next_state]))

        state = next_state

    # Print progress every 100 episodes
    if (episode + 1) % 100 == 0:
        print(f"Episode {episode + 1}/{num_episodes}")

# Testing the trained agent
state = env.reset()
state = discretize_state(state)
done = False
while not done:
    action = np.argmax(q_table[state])
    next_state, _, done, _ = env.step(action)
    state = discretize_state(next_state)
```

```
    env.render()



# Close the environment

env.close()
```

Output:

```
Episode 100/1000

Episode 200/1000

Episode 300/1000

Episode 400/1000

Episode 500/1000

Episode 600/1000

Episode 700/1000

Episode 800/1000

Episode 900/1000

Episode 1000/1000
```

Alpha, Gamma, and Epsilon Parameters:

- Alpha (α): It controls the learning rate, which determines to what extent new information overrides old information in the Q-values. A higher alpha means that new information has more weight in updating the Q-values.
- Gamma (γ): It represents the discount factor, which determines the importance of future rewards. A gamma value closer to 1 gives more weight to future rewards, encouraging the agent to consider long-term consequences.
- Epsilon (ε): It is the exploration rate, which determines the probability of the agent taking a random action (exploration) instead of exploiting its current knowledge (exploitation). It balances exploration and exploitation in the learning process
- The Bellman equation is a fundamental equation in reinforcement learning that expresses the value of a state-action pair in terms of the immediate reward and the value of the next state. It describes how the value function (Q-value in this case) can be updated iteratively based on the observed rewards and transitions.

In Q-learning, the Bellman equation is used to update the Q-values according to the formula:

Q(s,a) =(1-alpha)* Q(s,a) + alpha * (r + gamma * max a’ Q(s’,a’))

S - current state

a - action taken

r - reward received

S’ - next state

Alpha - learning rate

Gamma - discount factor

Why Gym library used? In the context of reinforcement learning, the term might refer to the `Gym` library developed by OpenAI. This library provides a collection of test problems, called environments, that you can use to develop and compare reinforcement learning algorithms. The Gym library is widely used for research and experimentation in the field of reinforcement learning.

Difference between exploitation and exploration

For eg there is a cheese and a rat with some traps in an environment

Here, exploitation is that a rat can eat only some by exploitation (less cheese) because it does not do more action less reward

But, in exploration with the traps the rat can able to eat more cheese regardless of dangers (action with more rewards)

There are 11 Pytorch domains

1. Torchaudio
2. Torchtune
3. Torchvision
4. Torcharrow
5. Torchdata
6. Torchrec
7. Torchserv
8. Pytorch on XLA devices

9. Torchrl
10. Torchdict
11. Torchtext

```
pip install torchrl
```

Environments in TorchRL have two crucial methods: `reset()`, which initiates an episode, and `step()`, which executes an action selected by the actor. In TorchRL, environment methods read and write `TensorDict` instances. Essentially, `TensorDict` is a generic key-based data carrier for tensors. The benefit of using TensorDict over plain tensors is that it enables us to handle simple and complex data structures interchangeably.

```
reset = env.reset()

print(reset)
```

o/p:

```
TensorDict(

    fields={

        done: Tensor(shape=torch.Size([1]), device=cpu,
dtype=torch.bool, is_shared=False),

        observation: Tensor(shape=torch.Size([3]), device=cpu,
dtype=torch.float32, is_shared=False),

        terminated: Tensor(shape=torch.Size([1]), device=cpu,
dtype=torch.bool, is_shared=False),

        truncated: Tensor(shape=torch.Size([1]), device=cpu,
dtype=torch.bool, is_shared=False)},

    batch_size=torch.Size([]),

    device=cpu,

    is_shared=False)
```

```
reset_with_action = env.rand_action(reset)

print(reset_with_action)
```

```
TensorDict(

    fields={

        action: Tensor(shape=torch.Size([1]), device=cpu,
dtype=torch.float32, is_shared=False),

        done: Tensor(shape=torch.Size([1]), device=cpu,
dtype=torch.bool, is_shared=False),

        observation: Tensor(shape=torch.Size([3]), device=cpu,
dtype=torch.float32, is_shared=False),
```

```
        terminated: Tensor(shape=torch.Size([1]), device=cpu,
dtype=torch.bool, is_shared=False),

        truncated: Tensor(shape=torch.Size([1]), device=cpu,
dtype=torch.bool, is_shared=False)},

    batch_size=torch.Size([]),

    device=cpu,

    is_shared=False)
```

This tensordict has the same structure as the one obtained from `EnvBase()` with an additional `"action"` entry. You can access the action easily, like you would do with a regular dictionary:

```
print(reset_with_action["action"])
```

```
tensor([1.3759])
```

We now need to pass this action tp the environment. We'll be passing the entire tensordict to the `step` method, since there might be more than one tensor to be read in more advanced cases like Multi-Agent RL or stateless environments:

```
stepped_data = env.step(reset_with_action)
```

```
print(stepped_data)
```

```
If the data action is processing
```

```
from torchrl.envs import step_mdp
```

Checks whether the observation, reward, termination and truncation is done or not

```
data = step_mdp(stepped_data)
```

```
print(data)
```

Data to process markov decision process

Step_mdp function used for calculating the state value or action value estimates, processing the reward and observed data.