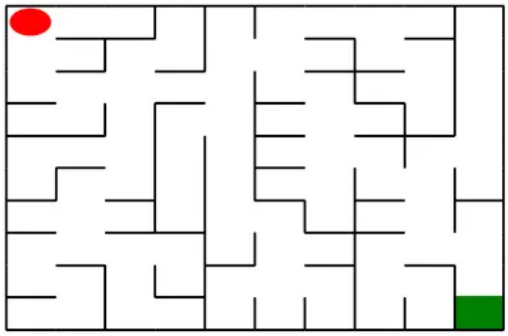
Mo Lab2 Report - Robot in Maze

万晨阳 3210105327

实验内容介绍

实验背景

在本实验中，要求分别使用基础搜索算法和 Deep QLearning 算法，完成机器人自动走迷宫。



如上图所示，左上角的红色椭圆既是起点也是机器人的初始位置，右下角的绿色方块是出口。

游戏规则为：从起点开始，通过错综复杂的迷宫，到达目标点(出口)。

- 在任一位置可执行动作包括：向上走 `'u'`、向右走 `'r'`、向下走 `'d'`、向左走 `'l'`。
- 执行不同的动作后，根据不同的情况会获得不同的奖励，具体而言，有以下几种情况。
  - 撞墙
  - 走到出口
  - 其余情况
- 需要您分别实现**基于基础搜索算法**和 **Deep QLearning 算法**的机器人，使机器人自动走到迷宫的出口。

实验要求

- 使用 Python 语言。
- 使用基础搜索算法完成机器人走迷宫。
- 使用 Deep QLearning 算法完成机器人走迷宫。
- 算法部分需要自己实现，不能使用现成的包、工具或者接口。

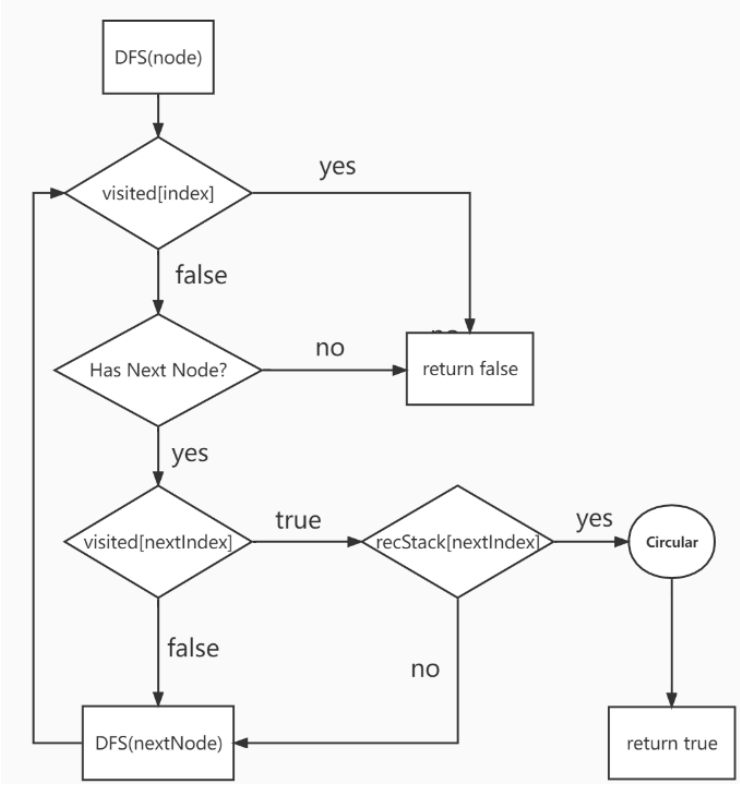
实验环境

使用 Python 实现基础算法的实现， 使用PyTorch框架实现 Deep QLearning 算法。

算法实现与改进

基于基础搜索算法的任务实现

我选择使用**深度优先搜索算法**实现基于基础搜索算法的机器人走迷宫任务。广度优先搜索基于队列实现，而深度优先搜索基于栈实现。算法的基本流程如下：



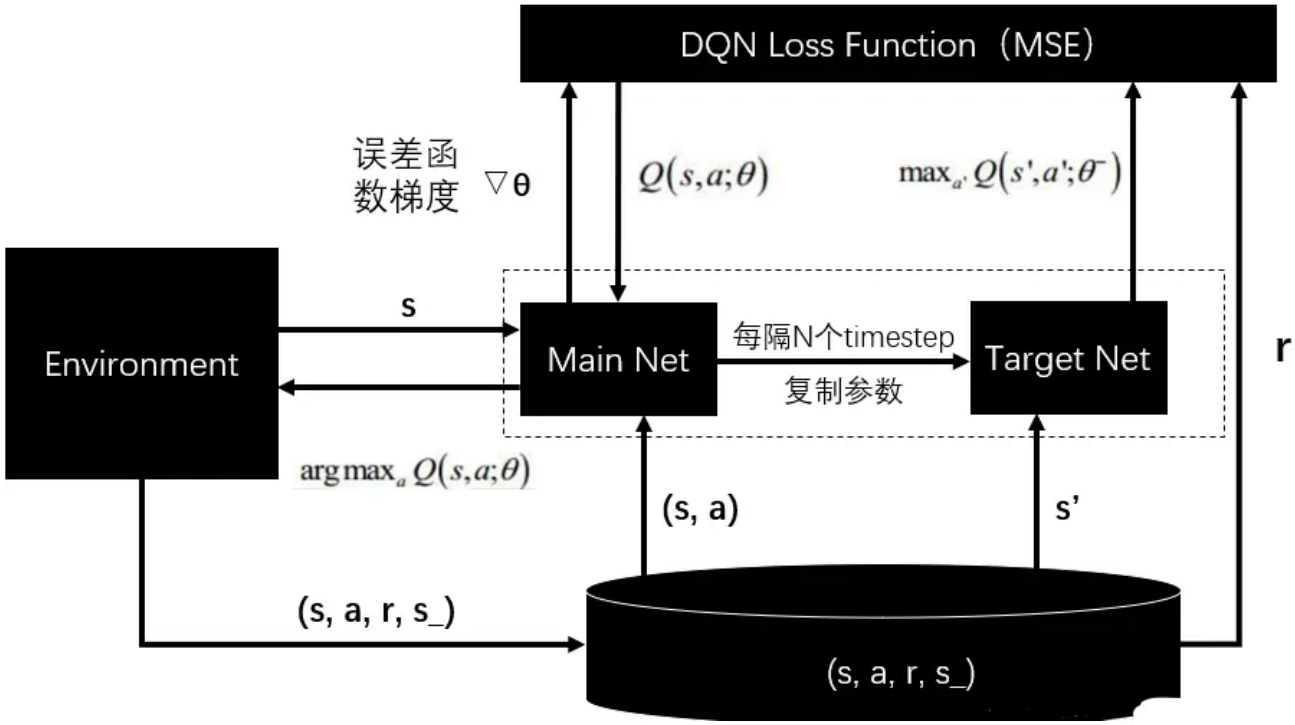
代码如下：

```
def my_search(maze):

    move_map = {
        'u': (-1, 0),  # up
        'r': (0, +1),  # right
        'd': (+1, 0),  # down
        'l': (0, -1),  # left
    }

    # define the search tree
    class SearchTree(object):
        def __init__(self, loc=(), action='', parent=None):
            self.loc = loc # the location of the node
            self.to_this_action = action # the action to this node
            self.parent = parent # the parent of the node
            self.children = [] # the children of the node

        def add_child(self, child):
            self.children.append(child)

        def is_leaf(self):
            return len(self.children) == 0

    # define the expand function
    def expand(maze, is_visit_m, node):
        can_move = maze.can_move_actions(node.loc)
        for a in can_move:
            new_loc = tuple(node.loc[i] + move_map[a][i] for i in range(2))
            if not is_visit_m[new_loc]:
                child = SearchTree(loc=new_loc, action=a, parent=node)
                node.add_child(child)

    # define the back_propagation function
    def back_propagation(node):
        path = []
        while node.parent is not None:
```

```
            path.insert(0, node.to_this_action) # insert the action to the path if the node is not the root
            node = node.parent
        return path # return the path with back prop

    # DFS
    start = maze.sense_robot()
    root = SearchTree(loc=start)
    queue = [root] # the queue of nodes
    h, w, _ = maze.maze_data.shape
    is_visit_m = np.zeros((h, w), dtype=int) # the matrix of visited
    path = []

    while True:
        current_node = queue.pop() # pop the last node in the queue
        is_visit_m[current_node.loc] = 1 # mark the current node as visited

        # if the current node is the destination, then break
        if current_node.loc == maze.destination:
            path = back_propagation(current_node)
            break

        # if the current node is leaf, then expand it
        if current_node.is_leaf():
            expand(maze, is_visit_m, current_node)
        # add the children of current node to the queue
        for child in current_node.children:
            queue.append(child)

    return path
```

基于Deep QLearning算法的任务实现

算法原理



在提供的代码中提供了基于简单的 DQN Robot 实现，其中依靠简单的两层全连接神经网络决策动作。 #### 模型优化

题中给出了普通的Qlearning算法，通过更新Q表实现学习，但是当状态维度过大，Q表的维度就会膨胀，所以我们需要DQN，不再实现状态和Q值的一一对应，而是构造$f:s\rightarrow a$。我们通过设定Q和Q_{update}的迭代关系，让f的取值逐渐收敛到这个迭代关系。迭代关系如下：

$$Q(s_t,a)_{update} = (1-\alpha) \times Q(s_t,a) + \alpha \times (R_{t+1} + \gamma \times \max_a Q(a, s_{t+1}))$$

等式左边由Q Network直接计算，等式右边由Q的复制网络计算，一定轮次之后我们更新右边网络，这样可以保证不会存在边计算边更新网络的情况，之后我们把这两边作差求MSE，得到loss函数。

下面介绍训练过程中的对几个比较重要的参数的调优工作。

1. reward
   最初的reward中，对于撞墙（hit wall）行为的惩罚太严重导致某些路径被过地抛弃；同时对到达终点（destination）的reward值，我认为应该随着地图大小变化而不应该为一固定值；对于正常行动的reward，我选择将惩罚略微增大，使得机器人倾向于找到最短路径。修改后的reward list如下：

```
        maze.set_reward(reward={
        "hit_wall": 5.0,
        "destination": -maze.maze_size ** 2.0,
        "default": 1.0})
```

2. 折扣因子 γ

> 折扣因子，它影响未来奖励的权重。常见值在 0.9 到 0.99 之间

对于折扣因子，我们通过遍历搜索0.9到0.99进行参数寻优。基于模型收敛速度进行评价。最后确定为0.9。

3. 探索率 ϵ 以及探索衰减率 ϵ_{decay}

> 探索率用于控制智能体在学习过程中的探索（随机选择动作）和利用（根据 Q 函数选择动作）之间的平衡。探索的衰减率，每次训练迭代后，探索率都会乘以该衰减率。衰减率的选择取决于具体任务和学习速度。较小的衰减率（如 0.995）意味着探索率减小得较慢，这有助于在复杂任务中进行更多的探索；较大的衰减率（如 0.99 或更大）意味着更快地减少探索，可能在简单任务中更有效。

对于本任务，探索率设置为0.5相对合理，但是因为搜索的任务相对比较简单，所以存在训练后期过程中探索率衰减速度过慢导致训练步浪费、无法有效探索的情况。所以我们选择适当降低衰减率以契合本任务情景。

4. 学习率

> 学习率是优化器在更新神经网络权重时使用的步长。较小的学习率（如 0.0001 或 0.001）意味着更新权重的速度较慢，可能需要更多的迭代次数；较大的学习率（如 0.01 或更大）可能导致权重更新过快，从而影响收敛性能。

对于学习率，我们通过遍历搜索进行参数寻优。基于模型收敛速度进行评价。最后确定为1e-1。

5. QNetwork 网络结构调整

由于输入输出维度较低（任务相对简单），而网络中每一层的参数数量相对较多且都是线性层，我的想法是可以选择增加网络层数而降低每层的参数量，同时添加非线性激活层。这样总的参数数目变化相对不大，而能够更有效地发挥激活函数的作用。最终QNetwork结构如下：

```
class QNetwork(nn.Module, ABC):
    """Actor (Policy) Model."""

    def __init__(self, state_size: int, action_size: int, seed: int):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
        """
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.net = nn.Sequential(
```

```
            nn.Linear(state_size, 128),
            nn.ReLU(False),
            nn.Linear(128, 256),
            nn.ReLU(False),
            nn.Linear(256, 128),
            nn.ReLU(False),
            nn.Linear(128, action_size)  # the final fc_net
        )

    def forward(self, state):
        """Build a network that maps state -> action values."""
        return self.net(state)
```

具体实现

在基于PyTorch框架的DQN机器人基础上进行改进

```
class Robot(QRobot):

    valid_action = ['u', 'r', 'd', 'l']

    ''' QLearning parameters'''
    epsilon0 = 0.5
    gamma = 0.9
    EveryUpdate = 1  # the interval of target model's updating

    """some parameters of neural network"""
    target_model = None
    eval_model = None
    batch_size = 32

    learning_rate = 1e-1
    TAU = 1e-3
    step = 1

    """setting the device to train network"""
    device = torch.device(
        "cuda:0") if torch.cuda.is_available() else torch.device("cpu")


    def __init__(self, maze):
        super(Robot, self).__init__(maze)
        maze.set_reward(reward={
            "hit_wall": 10.,
            "destination": -maze.maze_size ** 2,
            "default": 1.})

        self.maze = maze
        self.maze_size = maze.maze_size

        """build network"""
        self.target_model = None
        self.eval_model = None
        self._build_network()

        """create the memory to store data"""
        max_size = max(self.maze_size ** 2 * 3, 1e4)
        self.memory = ReplayDataSet(max_size=max_size)
        self.memory.build_full_view(maze=maze)
        self.loss_list = self.train()

    def train(self):
```

```
        loss_list = []
        batch_size = len(self.memory)

        while True:
            loss = self._learn(batch=batch_size)
            loss_list.append(loss)
            self.reset()
            for _ in range(self.maze.maze_size ** 2 - 1):
                a, r = self.train_update()
                if r == self.maze.reward["destination"]:
                    return loss_list

    def _build_network(self):
        seed = 0
        random.seed(seed)

        """build target model"""
        self.target_model = QNetwork(
            state_size=2, action_size=4, seed=seed).to(self.device)

        """build eval model"""
        self.eval_model = QNetwork(
            state_size=2, action_size=4, seed=seed).to(self.device)

        """build the optimizer"""
        self.optimizer = optim.Adam(
            self.eval_model.parameters(), lr=self.learning_rate)

    def target_replace_op(self):
        """ replace the whole parameters"""
        self.target_model.load_state_dict(self.eval_model.state_dict())

    def _choose_action(self, state):
        state = np.array(state)
        state = torch.from_numpy(state).float().to(self.device)
        if random.random() < self.epsilon:
            action = random.choice(self.valid_action)
        else:
            self.eval_model.eval()
            with torch.no_grad():
                # use target model choose action
                q_next = self.eval_model(state).cpu().data.numpy()
            self.eval_model.train()
            action = self.valid_action[np.argmin(q_next).item()]
        return action


    def _learn(self, batch: int = 16):
        if len(self.memory) < batch:
            # print("the memory data is not enough")
            return
        state, action_index, reward, next_state, is_terminal = self.memory.random_sample(batch)

        """ convert the data to tensor type"""
        state = torch.from_numpy(state).float().to(self.device)
        action_index = torch.from_numpy(action_index).long().to(self.device)
        reward = torch.from_numpy(reward).float().to(self.device)
        next_state = torch.from_numpy(next_state).float().to(self.device)
        is_terminal = torch.from_numpy(is_terminal).int().to(self.device)

        self.eval_model.train()
        self.target_model.eval()
```

```
        """Get max predicted Q values (for next states) from target model"""
        Q_targets_next = self.target_model(
            next_state).detach().min(1)[0].unsqueeze(1)

        """Compute Q targets for current states"""
        Q_targets = reward + self.gamma * Q_targets_next * \
            (torch.ones_like(is_terminal) - is_terminal)

        """Get expected Q values from local model"""
        self.optimizer.zero_grad()
        Q_expected = self.eval_model(state).gather(dim=1, index=action_index)

        """Compute loss"""
        loss = F.mse_loss(Q_expected, Q_targets)
        loss_item = loss.item()

        """ Minimize the loss"""
        loss.backward()
        self.optimizer.step()

        """copy the weights of eval_model to the target_model"""
        self.target_replace_op()
        return loss_item

    def train_update(self):
        state = self.sense_state()
        action = self._choose_action(state)
        reward = self.maze.move_robot(action)
        next_state = self.sense_state()

        is_terminal = 1 if next_state == self.maze.destination or next_state == state else 0
        self.memory.add(state, self.valid_action.index(
            action), reward, next_state, is_terminal)

        if self.step % self.EveryUpdate == 0:
            self._learn(batch=32)

        """---update the step and epsilon---"""
        self.step += 1
        self.epsilon = max(0.01, self.epsilon * 0.99)
        return action, reward

    def test_update(self):
        state = np.array(self.sense_state(), dtype=np.int16)
        state = torch.from_numpy(state).float().to(self.device)

        self.eval_model.eval()
        with torch.no_grad():
            q_value = self.eval_model(state).cpu().data.numpy()
        action = self.valid_action[np.argmin(q_value).item()]
        reward = self.maze.move_robot(action)
        return action, reward
```

训练结果

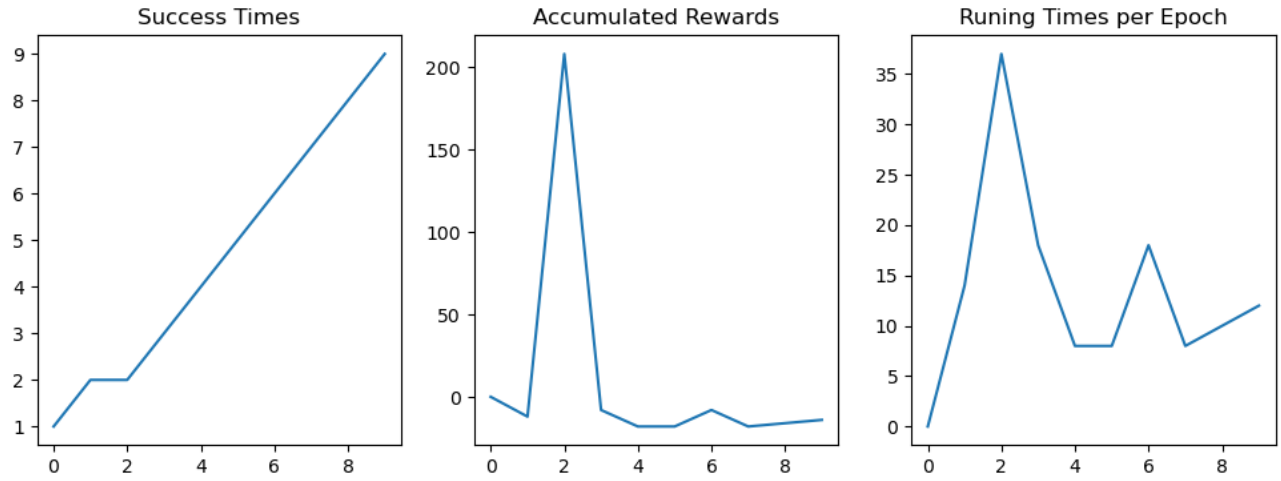
我们随机生成迷宫对模型进行测试。epoch大小和maze_size大小见下面结果说明。

```
epoch = ...  # 训练轮数
maze_size = ...  # 迷宫size
training_per_epoch=int(maze_size * maze_size * 1.5)

g = Maze(maze_size=maze_size)
```

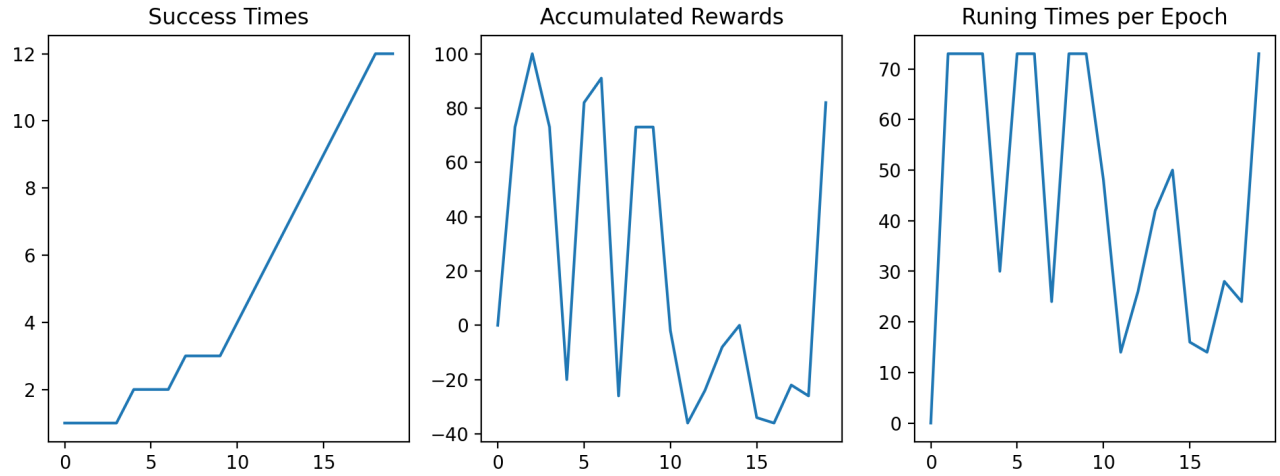
```
r = Robot(g)
runner = Runner(r)
runner.run_training(epoch, training_per_epoch)
runner.generate_gif(filename="results/dqn_size10.gif")
runner.plot_results()
```

以下为训练过程的结果展示：

- epoch = 10, maze_size = 5



- epoch = 20, maze_size = 7



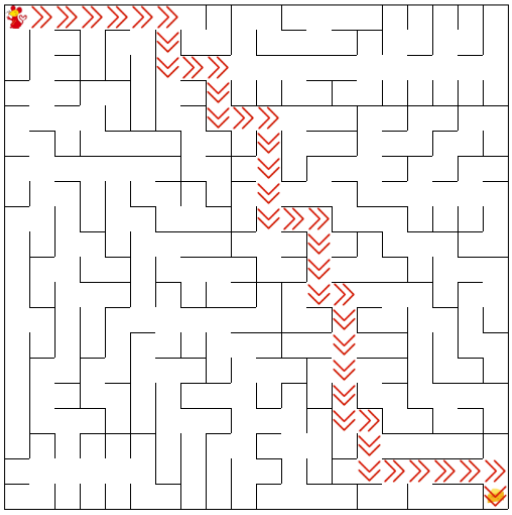
可以看到随着训练次数的增长，reward值虽然存在一些波动，但大致上收敛较快。说明我们的模型效果是较好的。

测试结果

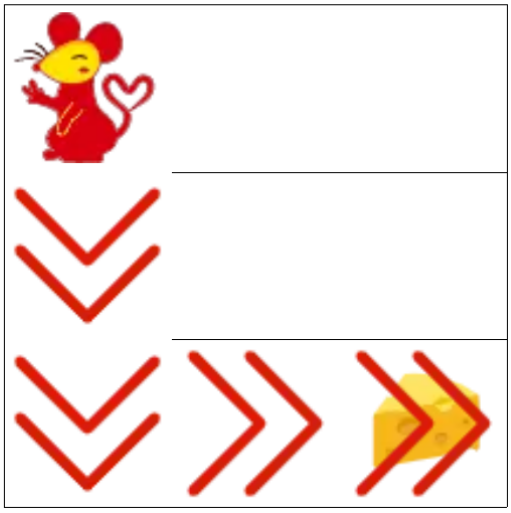
在平台上进行提交，测试结果如下：

测试点	状态	时长	结果
测试基础搜索算法	✓	2s	恭喜, 完成了迷宫
测试强化学习算法(初级)	✓	1s	恭喜, 完成了迷宫
测试强化学习算法(中级)	✓	3s	恭喜, 完成了迷宫
测试强化学习算法(高级)	✓	602s	恭喜, 完成了迷宫

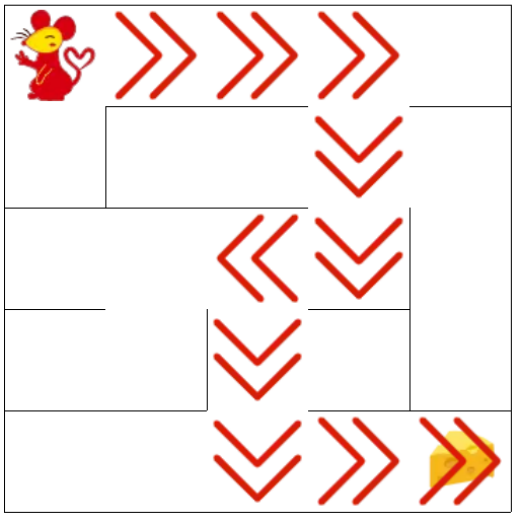
基础搜索算法 (Victory)



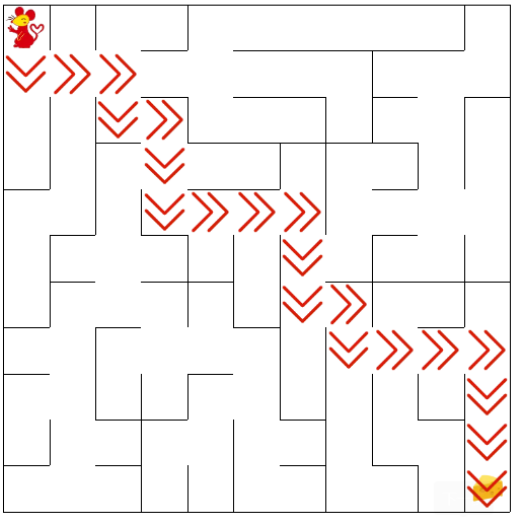
强化学习level3 (Victory)



强化学习level5 (Victory)



强化学习level11 (Victory)



收获总结

在进行模型训练时，我注意到学习率、探索策略和神经网络结构的选择对算法的性能有着重要影响。通过反复调整这些参数，我逐步改进了模型的性能，提高了机器人在迷宫中找到出口的效率。通过这个项目，我更深刻地理解了深度强化学习在解决复杂任务中的潜力。DQN算法的应用使得机器人能够通过不断的试错来学习最优策略，而不需要显式地指定规则。基于当前项目的经验，我计划进一步研究和实践更先进的深度强化学习算法，以适应更复杂的环境。此外，我也希望探索如何将所学到的知识应用于其他领域，例如自动驾驶、机器人导航等。