|  |
| --- |
| 1. 实验目的及要求   利用学习过的深度学习强化学习知识设计一个游戏迷宫游戏 |
| 1. 实验原理与内容   当应用Q-learning算法进行迷宫实验时，可以创建各种不同类型的迷宫结构，包括不同大小的迷宫、不同数量和位置的障碍物以及不同起点和终点的位置。这样可以让测试强化学习算法在各种条件下的表现，并且验证算法对于复杂环境的适应能力。  通过在这些不同环境下运行Q-learning算法，可以观察到智能体是如何通过与环境的交互来逐步学习最优路径。实验不仅可以帮助理解Q-learning算法的工作原理，还可以为提供一个直观的展示，展示了智能体是如何在面对复杂环境时提高其决策能力的过程。  除了Q-learning算法之外，还可以尝试其他强化学习算法，比如Deep Q Network（DQN）、Double DQN等，来比较它们在迷宫实验中的表现。这种对比分析有助于更全面地了解不同算法的优缺点，以及在不同环境下的适用性。  总的来说，通过在生成的迷宫环境中应用Q-learning等强化学习算法，可以深入研究智能体在复杂环境中的学习能力，探索算法的性能表现，并且为解决现实世界中类似问题提供启示。 |
| 1. 实验软硬件环境   PC，pytorch |
| 四、实验过程（实验步骤、记录、数据、分析）  1.首先是https://github.com/QikaiXu/Robot-Maze-Solving到上下载号对应的智能体走迷宫的所有项目代码。  2.创建对应的虚拟环境，并安装好运行代码所需的包，如cv2等。  3.解读代码  其中有三个类的定义：Maze类、Robot类和Runner类。  **其一Robot类定义：**  **初始化\_\_init\_\_**方法**:**  接收迷宫(maze)、学习率(alpha)、折扣因子(gamma)和初始探索率(epsilon0)作为参数。  设置迷宫、有效动作、状态、动作等属性。  初始化学习参数，如学习率、折扣因子、探索率(epsilon)和迭代次数(t)。  创建一个空的Q表(Qtable)，用于存储每个状态-动作对的Q值。  调用reset方法初始化智能体的状态。  **重置 (**reset**方法)**:  调用sense\_state方法（该方法在给出的代码段中未定义，但预计会返回智能体的当前状态）来设置智能体的状态。  调用create\_Qtable\_line方法（该方法在给出的代码段中未定义，但预计会在Q表中为当前状态创建新的行或更新其值）。  **设置状态 (**set\_status**方法)**:  允许设置智能体是否处于学习模式(learning)或测试模式(testing)。  在测试模式下，智能体不会进行随机选择（这是TODO 1的内容）。  **update\_parameter**方法**:**  当智能体处于测试模式时，将探索率(epsilon)设置为0，以确保智能体不会进行随机选择（这是TODO 1的实现）。当智能体处于学习模式时，更新迭代次数(t)和探索率(epsilon)。这里使用了epsilon的衰减策略，即随着迭代次数的增加，epsilon逐渐减小，以减少随机选择动作的可能性（这是TODO 2的实现）。 **sense\_state 方法** 这个方法用于感知智能体的当前状态。它调用了迷宫对象（self.maze）的sense\_robot方法来获取当前状态，并返回这个状态。这个状态通常表示智能体在迷宫中的位置或配置。 **create\_Qtable\_line 方法** 这个方法用于在Q表中为给定的状态（state）创建条目。如果Q表中已经存在这个状态的条目，则不会进行任何更改。否则，它会为所有有效的动作（self.valid\_actions）在Q表中创建条目，并将它们的Q值初始化为0.0。这里使用了setdefault方法来避免重复创建相同的条目。 **choose\_action 方法** 这个方法用于根据给定的规则选择动作。它首先定义了一个内部函数is\_random\_exploration，用于确定是否进行随机探索。  如果智能体处于学习模式（self.learning为True），则根据is\_random\_exploration的结果来选择动作：如果进行随机探索（即is\_random\_exploration返回True），则随机选择一个有效动作。否则，选择当前状态下Q值最高的动作。如果智能体处于测试模式（self.testing为True），则总是选择当前状态下Q值最高的动作。如果智能体既不在学习模式也不在测试模式，则随机选择一个有效动作。 **update\_Qtable 方法** 这个方法用于根据给定的规则更新Q表。当智能体处于学习模式（self.learning为True）时，它使用Q-learning的更新规则来更新Q值。具体地，它根据当前状态（self.state）、采取的动作（action）、奖励（r）、下一个状态（next\_state）、学习率（self.alpha）和折扣因子（self.gamma）来计算新的Q值。 **update 方法** 这个方法描述了智能体在每个时间步中应执行的操作。无论智能体是在训练还是测试模式下，它都会被调用。该方法返回当前采取的动作和相应的奖励。  self.state = self.sense\_state()：智能体感知其当前状态。  self.create\_Qtable\_line(self.state)：为当前状态在Q表中创建条目（如果尚不存在）。  action = self.choose\_action()：根据当前状态选择动作。  reward = self.maze.move\_robot(action)：智能体执行选择的动作，并从迷宫环境中接收奖励。  next\_state = self.sense\_state()：智能体感知其下一个状态。  self.create\_Qtable\_line(next\_state)：为下一个状态在Q表中创建条目（如果尚不存在）。  如果智能体处于学习模式且不是测试模式（self.learning为True且self.testing为False），则执行以下操作：  self.update\_Qtable(reward, action, next\_state)：使用Q-learning的更新规则更新Q表。  self.update\_parameter()：更新其他可能的参数（尽管这个方法在提供的代码段中未定义，但可能用于更新如epsilon（探索率）等参数）。  返回当前采取的动作和奖励。  总的来说，Robot类的定义即是算法的定义，使用Qlearning算法即创建Q表，通过动作选择来收集奖励，从而更新Q表，最后又通过Q表来实现对下次动作的预测。Robot类是一个封装了Q-learning算法逻辑的实体，它代表了一个能够学习如何执行最优动作的智能体（agent）。Robot类负责与环境进行交互，并根据环境给出的奖励来调整其策略，从而最大化未来获得的奖励。  微信截图_20240610204526  **第二就是Maze类它是自动生成迷宫的代码：**  通过传入两个参数来实现对迷宫所有整体信息的创建，包括迷宫结构、陷阱和目的地的位置。其中maze\_size是必传参数，它将定义迷宫的大小，通过传入长和宽的数值大小来实现对不同大小迷宫的创建。其中也有对当前机器位置的绘制以及每一步智能体移动后的位置创建。其次就是传入陷阱的智能体图片，从而绘制出有图片的迷宫，使迷宫呈现得更加具体。  Maze_code  run_maze  第三类Runner则是记录和存储对应训练后的参数以及绘制对应智能体位置和视频的生成  ****初始化方法 (**\_\_init\_\_**)****:  当创建一个Runner对象时，需要传入一个robot对象和一个maze对象。这两个对象可能分别代表了智能体和其所在的环境。  **run\_training**方法****:  这是Runner类的主要方法，用于运行训练过程。  参数:  training\_epoch: 训练的总轮数。  training\_per\_epoch: 每轮训练中的训练次数。  display\_direction: 一个布尔值，指示是否显示动作方向（可能用于可视化）。  在方法内部，定义了一个嵌套的train\_logger\_before\_act函数，该函数在每次行动之前记录训练信息。  **train\_logger\_before\_act**函数****:  这个函数在每个训练步骤（即每次智能体行动）之前被调用，用于记录当前的状态和参数。  记录的内容包括：  当前训练轮数和步骤的索引（(e, i)）。  是否成功（初始化为False）。  当前智能体的状态（通过sense\_state方法获得）。  当前状态对应的Q表的一个拷贝（这样可以在不改变原始Q表的情况下记录它）。  当前的学习率、探索率和折扣因子。  迷宫中智能体的位置（通过maze.robot.copy()获得，假设maze对象有一个robot属性且它有一个可拷贝的方法）。  如果display\_direction为True，则还会记录整个Q表的拷贝（可能是为了后续的可视化）。  **train\_logger\_after\_act**函数定义****：  这个函数是在训练循环中定义的，但它是作为一个独立的函数来使用的。  它接收四个参数：e（epoch编号）、i（当前epoch中的迭代次数）、action（智能体执行的动作）和reward（智能体获得的奖励）。  函数的功能是更新一个名为self.train\_robot\_record的字典，该字典存储了每个(e, i)对应的action和reward。  **训练循环**：  外层循环遍历training\_epoch次，即总共进行training\_epoch轮训练。  在每一轮训练中，accumulated\_reward（累积奖励）和run\_time（运行时间或迭代次数）被初始化为0。  内层循环遍历training\_per\_epoch次，即每一轮训练包含training\_per\_epoch次迭代。  **训练步骤**：  在每次迭代中，首先调用train\_logger\_before\_act(e, i)（这个函数在代码片段中没有给出，可能是记录当前状态或之前的信息）。  调用self.robot.update()，使智能体更新其状态，并返回执行的动作action和获得的奖励reward。  使用train\_logger\_after\_act(e, i, action, reward)记录这次迭代的信息。  更新run\_time和accumulated\_reward。  **到达目标位置的处理**：  如果智能体到达目标位置（self.maze.robot['loc'] == self.maze.destination），则执行以下操作：  将i加1（但这里有一个潜在问题，因为这将导致跳过下一次迭代）。  调用train\_logger\_before\_act(e,i)（可能是为了记录到达目标位置的信息，但这样做可能会覆盖之前迭代的信息）。  在self.train\_robot\_record[(e,i)]中添加'success': True表示智能体成功到达目标。使用break跳出内层循环。 **run\_testing 方法：** 这个方法主要用于在给定数量的测试迭代中评估智能体的性能。  **初始化统计信息**：  创建一个空的字典 self.test\_robot\_statics，并初始化三个列表来存储测试成功次数、累积奖励和测试时间。  **设置智能体状态**：  调用 self.robot.set\_status(learning=False, testing=True) 将智能体的状态设置为测试模式，禁用学习。  **计算测试迭代次数**：  testing\_per\_epoch 被重新设置为迷宫高度的平方的85%，可能是为了测试迷宫中大约85%的区域。  **测试循环**：  在循环中，智能体执行更新操作，并获取奖励。  如果智能体到达目标位置，循环会提前结束。  **更新统计信息**：  根据智能体是否到达目标位置，将成功次数添加到 self.test\_robot\_statics['success'] 中。将累积奖励和测试时间分别添加到 self.test\_robot\_statics['reward'] 和 self.test\_robot\_statics['times']中。 draw\_header 方法 这个方法用于生成一个视频或图像的头部。  **设置Logo尺寸**：  默认的Logo尺寸是200x50像素（高度为50像素，但这里只给出了宽度）。  **创建Logo图像**：  创建一个白色的条带作为Logo的底部背景。  将Logo图像（self.logo\_img）放在白色条带上方。  在Logo上方再添加一个白色条带。  **计算头部形状**：  头部宽度基于基础图像的宽度和Logo的尺寸计算。  头部高度是Logo的高度加上一些额外的空间。  **创建头部图像**：  创建一个白色的背景图像，其形状由 header\_shape 变量定义。  将Logo图像和白色背景图像水平堆叠，形成头部图像。  **返回头部图像**：  返回生成的头部图像。 **draw\_robot 方法：** 这个方法用于在基础图像上绘制智能体。  **初始化**：  复制基础图像 base\_image 到 img 以避免修改原始图像。  从 parameters 字典中获取智能体的位置 maze\_loc。  设置网格大小 grid\_size 为 100，这决定了图像中每个网格的大小。  **绘制智能体**：  从智能体的位置 robot['loc'] 提取行 r 和列 c。  在图像 img 的对应网格上叠加智能体的图像 self.robot\_img[robot['dir']]，其中 robot['dir'] 可能是表示智能体方向的键。  **可选：绘制方向箭头**：  如果 self.display\_direction 为 True，则绘制表示 Q 表中最大 Q 值的箭头。  遍历 parameters['Qtable'] 中的每个状态，找到具有最大 Q 值的动作方向。  将箭头图像 self.arrow\_img[direction] 与原始图像混合，并将结果存储回 img。  **返回结果**：  返回更新后的图像 img。 write\_on\_header 方法 这个方法用于在视频或图像的头部上写入文本。  **初始化**：  从 header\_img 创建一个 PIL 图像对象 header。  创建一个 PIL 绘图对象 draw 用于在 header 上绘制文本。  **准备文本**：  定义多行文本，包括训练周期、当前状态、Q 值等。  根据是否成功完成任务，决定是否显示与动作和奖励相关的文本以及学习参数（如 epsilon、alpha、gamma）。  **绘制文本**：  使用 draw.text() 方法在 header 图像上绘制文本。  设置文本的位置、颜色、字体等属性。  **返回结果**：  将 PIL 图像 header 转换回 NumPy 数组并返回。 **generate\_movie 方法：**这个方法用于生成一个视频，其中包含了训练过程中智能体的不同状态。 **初始化**：  获取迷宫的基础图像 base\_image。  调用 self.draw\_header 方法（注意这里应该是 draw\_on\_header 的一个笔误）来绘制视频的头部 header\_img。  **定义**ensemble\_image**函数**：  这个内部函数接收一个训练ID train\_id，并返回一个元组，其中第一个元素是训练ID，第二个元素是一个通过垂直堆叠头部和当前迷宫图像得到的图像。  使用 self.write\_on\_header 写入头部信息，使用 self.draw\_robot 绘制智能体。  **确定视频参数**：  调用 ensemble\_image 使用第一个训练ID (0, 0) 来确定图像的高度、宽度和通道数。  使用 cv2.VideoWriter 创建一个视频写入器，设置视频编码器为 "XVID"，帧率为 10.0，并将视频尺寸减半以适应可能的输出分辨率。  **写入视频帧**：  使用 tqdm 创建一个进度条来显示视频生成的进度。  遍历 self.train\_robot\_record 中的所有训练ID。  对于每个训练ID，调用 ensemble\_image 获取图像，并使用 cv2.resize 将其尺寸减半以适应视频写入器的参数。  使用 writer.write 将图像帧写入视频文件。  **释放资源**：  调用 writer.release() 释放视频写入器资源。 plot\_results 方法 这个方法用于绘制训练结果的图表。  **设置绘图窗口**：  使用 plt.figure 创建一个大小为 12x4 的绘图窗口。  **绘制成功次数图**：  在第一个子图中，使用 plt.subplot(131) 绘制成功次数的累计图。  使用 np.cumsum(self.train\_robot\_statics['success']) 计算成功次数的累计值。  **绘制累积奖励图**：  在第二个子图中，使用 plt.subplot(132) 绘制累积奖励的图表。  直接使用 self.train\_robot\_statics['reward'] 作为奖励值。  **绘制每轮运行时间图**：  在第三个子图中，使用 plt.subplot(133) 绘制每轮运行时间的图表。  使用 np.array(self.train\_robot\_statics['times']) 将时间值转换为 NumPy 数组。  **显示图表**：  调用 plt.show() 显示图表。  **改进方面：**  首先是Maze类，由于原项目代码它是**保持随机生成陷阱位置和迷宫结构**其传入参数只有两个，一个就迷宫大小向量，另一个就是是否有传入已有的迷宫结构**。但在测试时，发现即使有传入迷宫文件，其迷宫内部结构还是变化，只有大小不变；并且其对应的陷阱位置也在不断变化。**所以我修改两个文件参数传递，其中一个文件内保存的是迷宫的完整信息包括内部结构，其二就是陷阱的位置信息。首先就是通过判断是否有两个文件的传入来决定是否要随机生成迷宫结构、陷阱位置和终点位置。第二就是在生成陷阱位置的时候它是一个内部参数，只需调用，无法保存，**所以通过将陷阱位置转换成数组，从而保存陷阱的位置信息。第三，新增一个函数判断迷宫是否在范围内，以保证迷宫的正确不会越过边境。**之所以为了修改上面这些就是为了在能使用相同迷宫模型结构时，来对比不同算法通过迷宫时的速度，即保持单一变量原则。  第二就是Robot类，在原先的项目代码当中使用的是**Q-learning算法**，而改进后使用的是**Double DQN算法**。而改进Double DQN算法，Double DQN算法通过引入两个Q值函数来缓解高估计问题。这两个网络具有相同的结构，但参数更新方式不同。在于分离了选择动作和评估动作价值的过程。具体来说，使用在线Q网络来选择在下一个状态中的最大Q值对应的动作，而使用目标Q网络来评估这个所选动作的Q值。  最后就是Runner类，由于Robot类的改进，其对应代码也需要做出相应的修改首先就是记录的将是target\_Q table,而不是Q table。并且在绘制对应参数时也应该是target\_Q table，还有就是target\_Q table是一个键值对不能像Q table直接显示出来，要先转化成字典然后读取键值对，并且因为在Double DQN算法当中，目标策略网络才是用于预测行为价值的，而行为策略网络是用于预测下一个动作。 |
| 五、实验结果分析  实验结果改进：  在通过不断的调整参数，如：alpha学习率、gamma折扣回报率、epsilon探索率和update\_freq更新频率  Maze_result  首先是10\*10迷宫结构并保持迷宫结构相同的情况下的对比：  maze_10-10    原始参数Q-learning：  maze_Qlearning  原始参数Double DQN：  maze_DoubleDQN（减少训练次数）2  可以对比出在修改成Double DQN算法后其累计的奖励更快收敛更快达到峰值，并且其找到目的地的时间更快了而且也表现的更加稳定（即每个epoch的运行时间）。即可以看出Double DQN 算法能在一定程度上缓解高估问题，由自举产生的高估和最大化导致的高估。  maze_reward_modify通过增大触碰陷阱扣除的奖励和到达目的地获得的奖励（原始触碰陷阱-30；到达目的地50），也能使模型更快的收敛，但也不是越大越好。  通过改进学习率和折扣率超参数  原始：  Qlearning_onset  alpha=0.1，gamma=0.95  Qlearning_0.1_0.95  alpha=0.3，gamma=0.99  Qlearning_0.3_0.99  alpha=0.4，gamma=0.95  Qlearning_0.4_0.95  alpha=0.6，gamma=0.85  Qlearning_0.6_0.85  alpha=0.55，gamma=0.95  Qleaing_0.55_0.95  alpha=0.5，gamma=0.9  Qlearning_0.55_0.9  alpha=0.5，gamma=0.85  Qlearning_0.55_0.85  Epsilon=0.3  Qlearning_epsilon0_0.3  Epsilon=0.4  Qlearning_epsilon0_0.4  Epsilon=0.6  Qlearning_epsilon0_0.6  Epsilon=0.7  Qlearning_epsilon0_0.7  通过以上结论图片对比，可以看出相应的alpha=0.55，gamma=0.85，Epsilon=0.6时，其收敛的更快更稳定，震荡线段较少（即不稳定性）。  通过对Q learning的调整参数也同时适用于Double DQN算法，但Double DQN算法还有update\_freq这个参数。通过以下对比也能找到最好的数值区间。  update\_freq=5  DDQN5  update\_freq=15  DDQN15  update\_freq=20  DDQN20  update\_freq=40  update\_freqDDQN40=70  DoubleDQN_15-15  可以从上图结果中对比得出，其对应****update\_freq=15时，效果最好。****  ****过于频繁的更新（update\_freq过大）****：  ****过度拟合****：频繁更新可能导致模型在每次训练时都对新的少量数据做出调整，这可能会使模型过于适应噪声或短期变化，从而降低泛化能力。  ****资源浪费****：如果数据流非常频繁，频繁的训练可能消耗过多的计算资源，特别是在在线学习或实时环境中，可能会导致系统响应变慢。  ****效率降低****：过多的训练次数可能增加通信延迟，特别是在分布式环境下，频繁同步模型状态会减慢整体流程。  ****更新频率过小（update\_freq过小）****：  ****信息滞后****：如果模型更新太慢，模型可能无法快速捕获数据的新趋势，导致决策过时，尤其是在动态变化的数据环境中。  ****错过重要模式****：如果数据集中存在周期性变化，长时间不更新可能会错过捕捉这些模式的机会。  ****欠拟合****：如果模型过于保守，不经常更新，可能会导致欠拟合，即模型未能充分利用所有可用的信息进行学习。 |