

# DS4001-25SP-HW3: 贝叶斯网络

姓名张语捷 PB24051037

2025 年 5 月 30 日

## 1 马尔可夫链 [24%]

### 1.1 回答问题 [20%]

#### 1.1.1 手动计算 [4%]

1. TODO

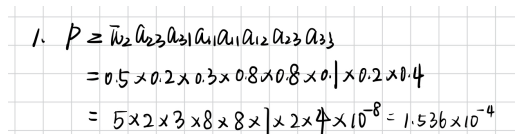

$$\begin{aligned} 1. \quad P &= \pi_{22} a_{23} a_{23} a_{11} a_{11} a_{12} a_{23} a_{23} \\ &= 0.5 \times 0.2 \times 0.3 \times 0.8 \times 0.8 \times 0.1 \times 0.2 \times 0.4 \\ &= 5 \times 2 \times 3 \times 8 \times 8 \times 1 \times 2 \times 4 \times 10^{-8} = 1.536 \times 10^{-4} \end{aligned}$$

图 1: 1.1.1.1

2. TODO

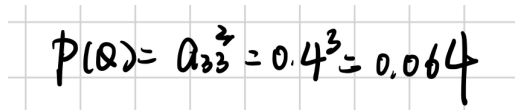
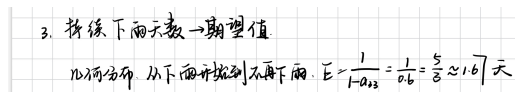

$$P(X_2) = a_{23}^2 = 0.4^2 = 0.064$$

图 2: 1.1.1.2

3. TODO



3. 计算下雨天数期望值

几何分布 从下雨开始到不再下雨  $E = \frac{1}{1-a_{22}} = \frac{1}{0.6} = \frac{5}{3} \approx 1.67$  天

图 3: 1.1.1.3

#### 1.1.2 代码填空与参数估计 [16%]

TODO: code + figure

```
1 def _forward(self, seq):
2     T = len(seq)
3     alpha = np.zeros((T, self.n_states))
```

```
4     scaling_factors = np.zeros(T)
5
6     # 初始化第一个时间步
7     alpha[0] = self.pi*self.B[:,seq[0]] ## fill in this blank with an expression ##
8     scaling_factors[0] = np.sum(alpha[0])
9     alpha[0] /= scaling_factors[0]
10
11     for t in range(1, T):
12         alpha[t] = np.dot(alpha[t-1],self.A)*self.B[:,seq[t]] ## fill in this blank with an expression ##
13         scaling_factors[t] = np.sum(alpha[t])
14         alpha[t] /= scaling_factors[t]
15
16     return alpha, scaling_factors
17
18 def _backward(self, seq, scaling_factors):
19     T = len(seq)
20     beta = np.zeros((T, self.n_states))
21
22     # 初始化最后一个时间步
23     beta[T-1] = 1.0
24     beta[T-1] /= scaling_factors[T-1]
25
26     for t in range(T-2, -1, -1):
27         beta[t] = np.dot(self.A, self.B[:, seq[t+1]]*beta[t+1]) ## fill in this blank with an expression ##
28         beta[t] /= scaling_factors[t]
29
30     return beta
```

```
● % python -u "/Users/zhangyujie/Desktop/CS/AI_P_T/hw3/USTC
-D54001-25sp/Homework/HW3/baum-welch.py"
π =
[0.248 0.252 0.5 ]
A =
[[0.    1.    0.   ]
 [0.577 0.091 0.332]
 [0.097 0.403 0.5  ]]
B =
[[0.    0.    1.   ]
 [0.    0.91 0.09]
 [1.    0.    0.   ]]
```

图 4: 1.1.2

### 1.1.3 运行结果对比 [4%]

TODO

Baum-Welch 算法:

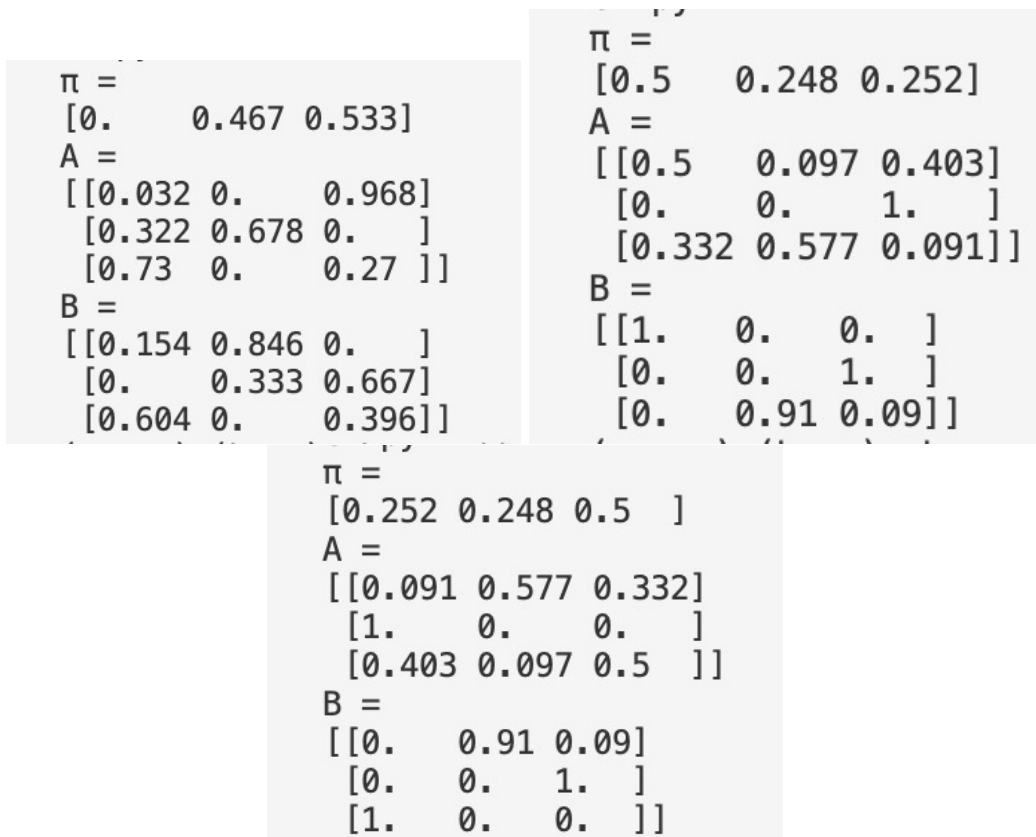


图 5: 1.1.3a

Gibbs Sampling:

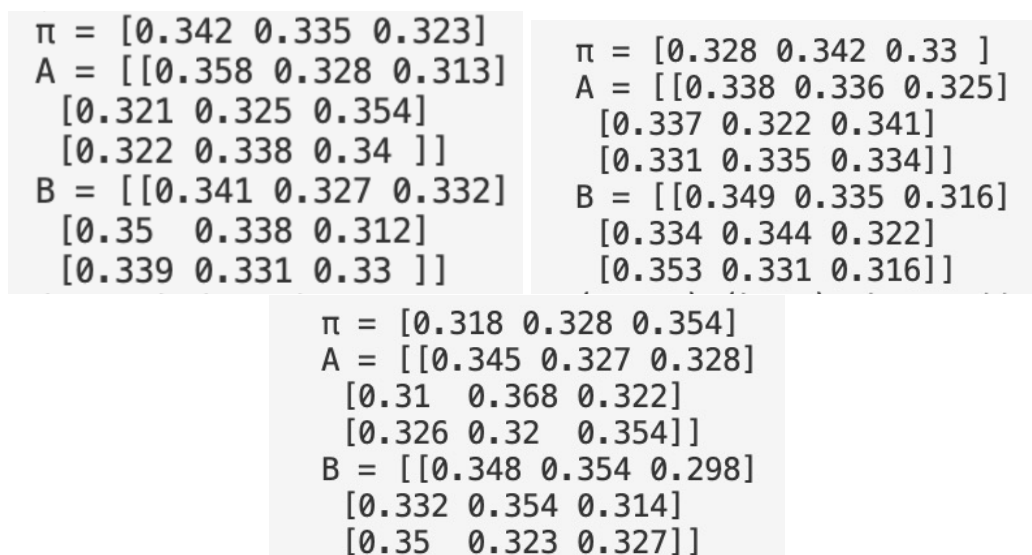


图 6: 1.1.3b

#### 1. 观察结果:

- Baum-Welch 算法: 转移矩阵中会存在接近 0 或 1 的值, 倾向于生成确定性强、非平滑、尖锐的结果, 容易收敛到局部最优, 观察三次采样结果, 可以观察到结果趋于稳定。
- Gibbs Sampling: 行列元素比较接近, 均衡平滑; 相比于 Baum-Welch 的结果, 没有出现任何接近 0 或 1 的概率, 更符合贝叶斯采样对全局分布的平均建模思路。得出的是概率分布的平均效果, 不会偏向某一结构。

#### 2. 对比结果:

- 稳定性: Baum-Welch 算法多次运行后结果基本一致, 收敛速度快, 结果稳定, 但是容易陷入局部最优; Gibbs 采样不同运行可能略有变化, 但是结果接近全局后验分布均值。
- 精度: Baum-Welch 算法, 以最大似然为目标, 只给出了单点估计结果, 最大化观测数据概率的参数; Gibbs 采样估计完整后验分布, 采用采样平均进行参数估计, 适合处理数据存在不确定性或者是小样本情况下。
- 平滑性: Baum-Welch 算法出现 0 或 1, 这是 EM 算法收敛到了局部极值的特征; Gibbs Sampling 的结果更平滑, 概率分布更均匀, 没有极端值, 减少过拟合。
- 算法机制: Baum-Welch 用 EM 算法最大似然的推断方式, 通过迭代优化参数; 通过单点估计来估计参数; 并且基于期望的确定更新方式, 容易陷入局部最优; 收敛快速。Gibbs 采样是基于马尔可夫链的采样方法, 通过多次采样得到后验分布的近似; 参数估计分布平均; 更新方式是基于采样的随机更新; 收敛慢但是能接近全局分布。
- 形象理解: Baum-Welch 像是通过梯度下降找到最陡峭的山峰, 快速到达; Gibbs 采样像是随机漫步, 在山谷中采样, 最终得到全局地图的效果。

## 2 贝叶斯网络 [64%]

### 2.1 贝叶斯网络：推理 [54%]

#### 2.1.1 精确推理 [16%]

```
1
2 def observe(self, agentX: int, agentY: int, observedDist: float) -> None:
3     # BEGIN_YOUR_CODE (our solution is 7 lines of code, but don't worry if you deviate from this)
4     for row in range(self.belief.numRows):
5         for col in range(self.belief.numCols):
6             #索引转换实际位置
7             x = util.colToX(col)
8             y = util.rowToY(row)
9
10            trueDist = math.sqrt((agentX - x) ** 2 + (agentY - y) ** 2) #计算欧氏距离
11            prob = util.pdf(trueDist, Const.SONAR_STD, observedDist) #计算发射概率
12            self.belief.setProb(row, col, self.belief.getProb(row, col) * prob) #更新belief的概率
13
14            self.belief.normalize() #归一化
15
16            # END_YOUR_CODE
17
18
19 def elapseTime(self) -> None:
20     # BEGIN_YOUR_CODE (our solution is 5 lines of code, but don't worry if you deviate from this)
21     #创建新的空信念分布
22     newBelief=util.Belief(self.belief.getNumRows(),self.belief.getNumCols(),value=0) #创建一个新的belief
23     #遍历所有 (oldTitle, newTitle)
24     #遍历新的belief的每个格子
25     for row in range(newBelief.getNumRows()):
26         for col in range(newBelief.getNumCols()):
27             newProb=0
28             #遍历旧的belief的每个格子
29             for oldRow in range(self.belief.getNumRows()):
30                 for oldCol in range(self.belief.getNumCols()):
31                     oldTile=(oldRow,oldCol)
32                     newTile=(row,col)
33                     #如果旧格子和新格子之间有转移概率
34                     if (oldTile,newTile) in self.transProb:
35                         transProb = self.transProb[(oldTile, newTile)]
36                         oldProb = self.belief.getProb(oldRow, oldCol)
37                         newProb += transProb * oldProb #累加转移概率
38                     newBelief.addProb(row, col, newProb) #更新新belief的概率
39     newBelief.normalize() #归一化
40     self.belief=newBelief #更新self.belief为新的belief
```

代码描述：

#### 1. observe 方法：

这段 observe 函数代码，用于基于观测到的距离 observedDist 更新洗哦啊车的信念分布 belief，更新每个网格位置的概率。基于贝叶斯更新方法，结合发射概率对信念分布进行修正。结合先验分布和观测数据，计算后验分布。

首先使用 `row` 和 `col` 的循环遍历每个网格位置 (`row,col`), 并将其索引转换为实际位置, 从而能够计算目标车辆与观测点之间的欧氏距离 (更新概率的其中一个关键参数)。

然后调用 `util.pdf(TrueDist, Const.SONAR_STD, observedDist)` 似然函数计算观测数据的似然概率发射概率, 得到目标车辆与观测点之间的距离的概率分布。其中使用到了高斯分布, `TrueDist` 是真实距离, `observedDist` 是观测到的距离, `Const.SONAR_STD` 是观测噪声标准差。观测概率反映当前网格位置与观测数据的匹配程度。

然后, 使用 `self.belief.setProb(row, col, self.belief.getProb(row, col) * prob)` 更新 `belief` 的概率, 将当前网格位置的概率乘以观测概率。最后调用 `self.belief.normalize()` 进行归一化处理, 确保所有网格位置的概率和为 1。

这段代码对每个网格位置均进行了更新, 确保了信念分布的全面性。

## 2. `elapsedTime` 方法:

这段 `elapsedTime` 代码的核心是通过已知的转移概率 `self.transProb` 计算每个网格位置的概率分布在时间推移后的更新结果, 从而实现基于隐马尔可夫模型 (HMM) 的状态转移过程, 结合信念分布和转移概率, 计算时间推移后的信念分布, 反映校车在网格世界中的运动。

首先创建一个新的空信念分布 `newBelief`, 用于存储时间推移后的概率分布。然后使用双重循环遍历新的 `belief` 的每个格子 (`row,col`), 对于每个位置, 初始化一个变量 `newProb` 为 0, 用于累加该格子的概率。

使用四重循环遍历旧的 `belief` 的每个格子 (`oldRow, oldCol`), 并将其索引转换为实际位置 `oldTile`。对于每个旧格子和新格子之间, 如果存在转移概率, 则获取转移概率 `self.transProb[(oldTile, newTile)]` 计算转移概率和旧格子的概率的乘积, 并累加到 `newProb` 中。

最后, 将 `newProb` 添加到 `newBelief` 中对应位置 (`row,col`) 的概率, 并进行归一化处理, 确保所有网格位置的概率和为 1。最后将 `self.belief` 更新为 `newBelief`, 完成时间推移后的信念分布更新。

同理, 代码对每个网格位置都进行了更新, 确保信念分布的完整性。

数学上的解释:

$P_{old}(i, j)$  表示旧信念分布中位置  $((i, j))$  的概率。

$T((i, j) \rightarrow (m, n))$  表示从位置  $((i, j))$  转移到位置  $((m, n))$  的转移概率。

$P_{new}(m, n)$  表示新信念分布中位置  $((m, n))$  的概率。

则,  $P_{new}(m, n) = \sum_{(i, j)} P_{old}(i, j) \cdot T((i, j) \rightarrow (m, n))$

### 2.1.2 模糊推理 [16%]

```
1
2 def updateBelief(self) -> None:
3     newBelief = util.Belief(self.belief.getNumRows(), self.belief.getNumCols(), 0)
4
5     # BEGIN_YOUR_CODE (our solution is 5 lines of code, but don't worry if you deviate from this)
6     for i,(r,c) in enumerate(self.samples): #元组迭代，遍历每个样本
7         newBelief.addProb(r,c,self.weights[i]) #将每个样本的权重添加到对应的格子
8     self.belief = newBelief #更新self.belief为新的belief
9     self.belief.normalize() #归一化
10
11
12     #raise Exception("Not implemented yet")
13     # END_YOUR_CODE
14
15 def elapseTime(self) -> None:
16     # BEGIN_YOUR_CODE (our solution is 8 lines of code, but don't worry if you deviate from this)
17     newSamples = []
18
19     for r,c in self.samples:
20         oldTitle = (r, c)
21
22         if oldTitle in self.transProbDict and self.transProbDict[oldTitle]:
23             newTitle=util.weightedRandomChoice(self.transProbDict[oldTitle]) #根据转移概率采样新的位置
24         else:
25             newTitle=oldTitle #如果没有转移概率，则保持不变
26
27         newSamples.append(newTitle) #将新的位置添加到新样本列表
28     self.samples = newSamples #更新样本位置
29     self.updateBelief() #更新belief分布
```

代码描述：

### 1. updateBelief 方法：

该段代码主要通过遍历样本 *self.samples* 和对应的权重 *self.weights* 将权重累加到信念分布对应的网格位置来实现信念分布的更新，最后通过归一化操作确保信念分布有效性。

首先创建一个新的空信念分布 *newBelief*，用于存储更新后的概率分布，初始化为 0。

然后使用循环遍历每个样本位置  $(r, c)$  和索引，对于每个样本的权重 *self.weights[i]* 累加到 *newBelief* 对应的网格位置，然后通过 *addProb* 方法进行累加。

最后将 *newBelief* 赋值给 *self.belief*，并调用 *self.belief.normalize()* 进行归一化处理，确保所有网格位置的概率和为 1。

数学解释：

$S = (r_i, c_i)$  表示样本集合，其中  $(r_i, c_i)$  是第  $i$  个样本的位置。

$w_i$  表示第  $i$  个样本的权重。

$P(r, c)$  表示信念分布中位置  $(r, c)$  的概率。则信念分布的更新公式为： $P(r, c) = \sum_{i:(r_i, c_i)=(r, c)} w_i$  这段代码通过遍历样本和累加权重实现了上述公式。

## 2. `elapsedTime` 方法:

该函数模拟目标车辆在时间推移后的动态行为, 并根据转移概率 `self.transProbDict` 为每个样本采样新的位置, 并更新样本集合 `self.samples`, 最终再调用 `updateBelief` 重新计算信念分布。

首先创建一个新的空列表 `newSamples`, 用于存储时间推移后的样本位置。

然后循环遍历当前样本集合中的每个样本, 表示当前所在的位置, 作为 `oldTitle`, 判断是否在 `self.transProbDict` 转移概率中有定义, 如果有, 则使用 `util.weightedRandomChoice(self.transProbDict[oldTitle])` 根据转移概率采样新的位置 `newTitle`, 否则保持不变, `newTitle` 设置为 `oldTitle`。

将新的位置 `newTitle` 添加到 `newSamples` 列表中。

最后将 `newSamples` 赋值给 `self.samples` 更新样本位置, 并调用 `self.updateBelief()` 更新信念分布。

样本集合 `self.samples` 和权重 `self.weights` 共同表示目标车辆位置的概率分布, 样本分布和权重反映了不确定性; 转移概率 `self.transProbDict` 本身模糊的, 表示从一个位置转移到另一个位置的可能性, 路径不确定; 此外, `util.weightedRandomChoice` 函数根据转移概率为每个样本采样新位置, 使得样本位置具有模糊性和随机性。

### 2.1.3 粒子滤波 [16%]

```
1 class ParticleFilter:
2     NUM_PARTICLES = 200
3
4     def __init__(self, numRows: int, numCols: int):
5         self.belief = util.Belief(numRows, numCols)
6
7         # Load the transition probabilities and store them in an integer-valued defaultdict.
8         # Use self.transProbDict[oldTile][newTile] to get the probability of transitioning from oldTile to newTile.
9         self.transProb = util.loadTransProb()
10        self.transProbDict = dict()
11        for (oldTile, newTile) in self.transProb:
12            if oldTile not in self.transProbDict:
13                self.transProbDict[oldTile] = collections.defaultdict(int)
14                self.transProbDict[oldTile][newTile] = self.transProb[(oldTile, newTile)]
15
16        '''
17        # Problem e: initialize the particles randomly
18        # Notes:
19        # - you need to initialize self.particles, which is a defaultdict
20        #   from grid locations to number of particles at that location
21        # - self.particles should contain |self.NUM_PARTICLES| particles randomly distributed across the grid.
22        # - after initializing particles, you must call |self.updateBelief()| to compute the initial belief distribution.
23        '''
24        # BEGIN_YOUR_CODE (our solution is 5 lines of code, but don't worry if you deviate from this)
25        self.particles = collections.defaultdict(int) # 初始化粒子 defaultdict
26        validTiles = list(self.transProbDict.keys()) # 获取所有可转移的位置
```



```
27     for _ in range(self.NUM_PARTICLES):
28         tile = random.choice(validTiles)
29         self.particles[tile] += 1 # 随机选择一个位置并增加粒子计数
30     #raise Exception("Not implemented yet")
31     # END_YOUR_CODE
32     self.updateBelief()
33
34
35     def observe(self, agentX: int, agentY: int, observedDist: float) -> None:
36         # Reweight the particles
37         # BEGIN_YOUR_CODE (our solution is 7 lines of code, but don't worry if you deviate from this)
38         weights= collections.defaultdict(float) # 创建一个新的权重分布
39         for (r,c),num in self.particles.items(): # (r,c)是行列索引, oldweights是粒子数量, self.particles[(r,c)] = num, 该
40             x= util.colToX(c)
41             y= util.rowToY(r)
42             trueDist = math.sqrt((agentX - x)**2+(agentY-y)**2) # 计算真实距离
43             prob=util.pdf(trueDist, Const.SONAR_STD, observedDist) # 计算发射概率
44             weights[(r,c)] = prob*num #更新权重,
45         # 归一化
46         totalWeight=sum(weights.values())
47         if totalWeight==0:
48             raise ValueError("All particle weight are zero.Check!")
49         for tile in weights:
50             weights[tile]/=totalWeight # 归一化, 使数量变为权重
51
52         neWeightedParticles = collections.defaultdict(int) # 创建一个新的粒子分布
53         for _ in range(self.NUM_PARTICLES):
54             # 根据权重重新采样粒子
55             sampledTile=util.weightedRandomChoice(weights) # 根据权重随机选择一个位置
56             neWeightedParticles[sampledTile] += 1
57
58         self.particles=neWeightedParticles
59         self.updateBelief() # 更新belief分布
60
61     #raise Exception("Not implemented yet")
62     # END_YOUR_CODE
63
64     # Resample the particles
65     # Now we have the reweighted (unnormalized) distribution, we can now re-sample the particles from
66     # this distribution, choosing a new grid location for each of the |self.NUM_PARTICLES| new particles.
67     newParticles = collections.defaultdict(int)
68     for _ in range(self.NUM_PARTICLES):
69         p = util.weightedRandomChoice(self.particles)
70         newParticles[p] += 1
71     self.particles = newParticles
72     self.updateBelief()
```

运行结果:

```
(base) zhangyujie@zhangyujiedeMacBook-Pro code % python drive.py -a -d -i P
articleFilter
-----
iteration 100
error: 63.31874999999995
-----
iteration 200
error: 64.05525
-----
iteration 300
error: 67.22412499999999
-----
*****
* GAME OVER *
* You Win! *
*****
closing
```

图 7: Particle

在运行结果中观察到，小车在抵达终点之前会往返，并且运行过程中有概率发生 car crash，导致小车位置不确定。在 game win 之前，出现了 iteration100 error 等问题，经过资料检索，发现其原因通常是：粒子数量不足、粒子初始化不均匀、转移概率不合理等。解决方法是增加粒子数量、调整转移概率、优化粒子初始化等。

代码描述：

1. **init** 初始化 `self.particles` 使用 `collections.defaultdict(int)`，作为一个字典表示网格位置对应的粒子数量，键为 `(row,col)`，值为该位置的粒子数量，当访问不存在的键时，自动初始化默认值为 0。

接着 `validTiles` 来获取所有可转移位置的合法位置。

根据粒子个数循环，每次随机选择一个位置，将该次循环的例子加入到该网格位置中，每次循环在该位置的粒子数量加 1。

最后（不需要自己写）更新信念分布。

## 2. **observe** 方法：

该方法用于根据观测到的距离 `observedDist` 更新粒子滤波器的信念分布。首先创建一个新的权重分布 `weights`，用于存储每个网格位置的权重。然后遍历当前粒子集合 `self.particles`，计算每个粒子位置 `(r,c)` 与观测点之间的真实距离，并使用高斯分布计算发射概率。将每个位置的权重乘以对应的粒子数量，得到新的权重分布。

首先和通过 `sollections.defaultdict(float)` 创建一个新的权重分布 `weights`，用于存储每个网格位置的权重，键为 `(row,col)`，值为该位置的权重。

然后遍历当前粒子集合 `self.particles`，将 `(r,c)` 索引通过 `util.colToC(c)` 和分别转换为当前的实际坐标，并计算该位置与观测点的欧氏距离 `trueDist`，并通过高斯分布计算发射概率 `prob`，使用 `util.pdf(trueDist, Const.SONAR_STD, observedDist)` 计算发射概率，并且更新权重，但第一个 `weights` 表示的是该位置的粒子个数。

接着计算总的粒子个数，所有权重归一化，使得上述的权重（实际上为数量）转化为新的权重，即粒子滤波中更新权重似然加权步骤。

然后创建一个新的粒子分布 *newWeightedParticles*，根据归一化后的权重重新采样粒子位置，生成新的粒子集合。使用 *util.weightedRandomChoice(weights)* 根据权重随机选择一个位置，并将该位置的粒子数量加 1。

最后将新的粒子集合赋值给 *self.particles*，并调用 *self.updateBelief()* 更新信念分布。

### 3. observe 方法：

该方法用于根据观测到的距离 *observedDist* 更新粒子滤波器的信念分布。首先创建一个新的权重分布 *weights*，用于存储每个网格位置的权重。然后遍历当前粒子集合 *self.particles*，计算每个粒子位置  $(r, c)$  与观测点之间的真实距离，并使用高斯分布计算发射概率。将每个位置的权重乘以对应的粒子数量，得到新的权重分布。

接下来，对权重进行归一化处理，确保所有权重和为 1。然后创建一个新的粒子分布 *newWeightedParticles*，根据归一化后的权重重新采样粒子位置，生成新的粒子集合。最后调用 *self.updateBelief()* 更新信念分布。

数学解释：

$P_{\text{old}}(r, c)$  表示旧信念分布中位置  $(r, c)$  的概率。

$w(r, c)$  表示位置  $(r, c)$  的权重。

则新信念分布为： $P_{\text{new}}(r, c) = P_{\text{old}}(r, c) \cdot w(r, c)$

### 4. elapseTime 方法：

该方法用于模拟时间推移后的粒子滤波器状态。首先创建一个新的空粒子集合 *newParticles*，然后根据当前粒子集合 *self.particles* 和转移概率 *self.transProbDict* 重新采样粒子位置。对于每个粒子位置  $(r, c)$ ，如果存在转移概率，则根据转移概率随机选择新的位置，否则保持不变。将新的位置添加到 *newParticles* 中。最后将 *newParticles* 赋值给 *self.particles* 并调用 *self.updateBelief()* 更新信念分布。

数学解释：

$$P_{\text{new}}(m, n) = \sum_{(i, j)} P_{\text{old}}(i, j) \cdot T((i, j) \rightarrow (m, n))$$

#### 2.1.4 思考题 [6%]

TODO 模糊推理中 *self.updateBelief()* 调用更频繁，原因如下：

1. **[LikelihoodWeighting]** 在模糊推理类中，观测 *observe* 和每次时间推移 *elapseTime* 末尾，各调用一次 *self.updateBelief* 更新信念分布。首先创建一个新的权重分布。

*LikelihoodWeighting* 中经典的模糊推理方法是似然权重采样 (*LikelihoodWeightingSampling*), 每次采样完整变量赋值, 保持观测变量固定, 对每个样本分配一个权重。所有似然值相乘形成样本的权重。

权重的更新和样本位置的更新分离, 新样本根据旧样本的位置并随机形成转移概率, 所以每次更新后都需要调用 *self.updateBelief()* 确保信念分布始终与当前样本状态一致。

2. **[ParticleFilter]** 在粒子滤波类中, 观测 *observe* 末尾, 调用一次 *self.updateBelief* 更新粒子滤波器的信念分布, 而时间推移 *elapseTime* 中不调用。

粒子滤波器是基于粒子重新采样算法通过对粒子进行加权和重新采样来近似后验分布。每次观测后, 粒子滤波器会根据观测数据更新粒子的权重, 并重新采样粒子位置。

每次时间推移, 根据转移概率更新粒子的位置, 而这个位置更新是基于每个网格位置的权重, 然后将粒子数量叠加到网格位置生成新的信念分布。因此其信念分布主要依赖于粒子的新采样结果, 而不是权重的直接叠加, 所以 *elapseTime* 中不再需要更新信念分布。

总结: 模糊推理中样本权重动态变化, 每次权重更新需要立刻反映到信念分布中, 因此在权重更新或样本位置更新后都要更新信念分布; 粒子滤波器中粒子权重变化通过重新采样重新反映到了粒子集合中, 信念分布的更新依赖于粒子集合的状态, 因此在时间推移后不需要再次更新信念分布。

## 2.2 贝叶斯网络: 学习 [10%]

TODO

第一轮:

$$P(D|A) = \pi^{id} (\theta_A^{id})^3 (1 - \theta_A^{id}) = 0.5 \times 0.6^3 \times 0.4 = 0.02592$$

$$P(D|B) = (1 - \pi)^3 (\theta_B^{id})^4 (1 - \theta_B^{id}) = 0.5 \times 0.4^4 \times 0.6 = 0.00768$$

$$\therefore \gamma_A = P(A|D) = \frac{P(D|A)}{P(D|A) + P(D|B)} = \frac{0.02592}{0.02592 + 0.00768} \approx 0.77$$

$$\gamma_B = P(B|D) = 0.23$$

假设后验概率与第一轮相同.  $N_{i,A}$  表示第  $i$  轮  $A$  为正的次数.

$$\bar{\pi}_{new} = \frac{\sum \gamma_{i,A}}{4} = \frac{4 \times 0.77}{4} = 0.77$$

$$\theta_A^{new} = \frac{\sum \gamma_{i,A} \cdot N_{i,A}}{\sum \bar{\pi}_{new} \times 5} = \frac{0.77 \times (4 + 1 + 2 + 1)}{0.77 \times 5 \times 4} = 0.45$$

$$\theta_B^{new} = \frac{\sum \gamma_{i,B} \cdot N_{i,B}}{\sum \bar{\pi}_{new} \times 5} = \frac{0.23 \times (4 + 1 + 2 + 1)}{0.23 \times 4 \times 5} = 0.45$$

因此,  $\gamma_A = 0.77$   $\gamma_B = 0.23$

$$\bar{\pi}^{new} = 0.77 \quad \theta_A^{new} = 0.45 \quad \theta_B^{new} = 0.45$$

图 8: Particle

### 3 贝叶斯深度学习 [6%]

TODO

1. **过高置信预测** 传统深度学习模型通常会对其预测结果给出过高的置信度，即使在模型不确定或输入分布与训练分布不匹配的情况下，模型仍然会给出接近 1 的置信度。这种现象在实际应用中可能导致错误决策。

解决：贝叶斯网络通过对模型参数（如权重）引入概率分布，能够量化预测的不确定性。通过对模型参数进行采样，贝叶斯深度学习可以生成多个预测结果，并计算这些结果的平均值和方差，从而提供更可靠的置信度估计。模型输出不仅是预测值，还包括预测的不确定性范围。这种不确定性可以用来判断模型是否对某些输入缺乏信心。避免过高置信预测。

2. **对分布外数据的预测能力差** 传统深度学习模型在面对分布外数据（Out-of-Distribution, OOD）时，通常会给出错误的高置信预测。

解决：贝叶斯网络通过对参数和预测结果建模不确定性，可以更好地识别分布外数据。当模型输入数据的预测不确定性较高时，可以将其标记为分布外数据，从而避免错误的高置信预测。

3. **易受对抗性操纵的影响** 传统深度学习模型容易受到对抗性样本的攻击，即通过对输入数据进行微小扰动，导致模型输出错误的高置信预测。

解决：贝叶斯网络通过对参数的不确定性建模，能够更鲁棒地应对对抗性样本。对抗性样本通常会导致模型的不确定性增加，贝叶斯网络可以利用这一特性检测并拒绝对抗性样本。

4. **现实应用** 贝叶斯序列到序列模型 *BayesianSeq2Seq* 在机器翻译任务中用于生成翻译结果，提供不确定性估计，帮助用户识别低质量翻译。

传统的基于 LSTM 或 *TransformerSeq2Seq* 模型是确定性的，即使在面对不确定性时也会给出高置信度的翻译结果，并且无法量化不确定性，且对噪声敏感，对抗样本或外部输入容易导致错误生成。

贝叶斯 Seq2Seq 模型通过对模型参数引入概率分布，把模型参数（如权重）视为随机变量，目标是学习参数的概率分布而非单一参数，在这个过程中，有几个关键的改进：

1. **参数分布建模**：贝叶斯 Seq2Seq 模型对参数进行概率建模，通常使用变分推断方法来近似后验分布。通过对参数的分布进行采样，可以生成多个翻译结果，从而量化不确定性。
2. **预测不确定性**：通过多次采样生成多样化输出，并计算置信度。
3. **正则化效果**：贝叶斯方法天然减少过拟合。

通过引入参数不确定性，显著提升了模型在这些场景的鲁棒性：需要可靠性评估的任务、面对噪声和分布外数据、生成多样化合理输出。

在 *WeightUncertaintyinNeuralNetworks* 中提出的贝叶斯反向传播(BBP)体现了 *BayesianSeq2Seq*, 如将神经网络权重表示为概率分布(高斯分布), 而非固定值, 对应的。在编码器-解码器结构中, 通过对权重采样生成多样化输出, 解决传统“确定性生成”问题。

在 *DropoutasaBayesianApproximation* 中证明了 Dropout 是贝叶斯推断的近似, 测试时保持 Dropout 激活, 通过多次前向传播采样近似后验分布, 生成多样化输出, 量化不确定性, 应用在生成文本时, 对同一输入运行多次解码, 统计输出方差作为置信度指标。

## 体验反馈 [6%]

- (a) [必做] TODO 大概花了 10 个小时, 主要是花了比较多的时间去理解精确推理和模糊推理的区别以及代码实现, 还有粒子滤波器权重更新的实现花了较多的时间, 并且花了一定的时间在想如何实现粒子滤波器的转移概率和采样。
- (b) [选做] TODO 写这次作业感觉自己写代码能力还有待提高, 有点慢了啊啊啊有时还得借助 ai。