中山大学数据科学与计算机学院

人工智能本科生实验报告

（2018-2019）学年秋季学期

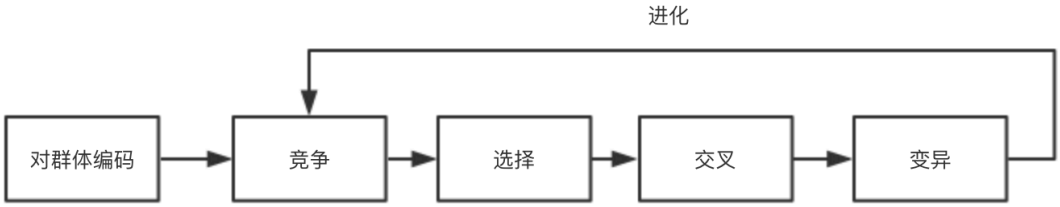
教学班级	16级计科二班	专业（方向）	计算机科学与技术
学号	16337334	姓名	周启恒

期末项目

（1）原理及流程图

遗传算法：

- 定义：模拟达尔文生物进化论的自然选择和遗传学机理的生物进化过程的计算模型，是一种通过模拟自然进化过程搜索最优解的方法。简而言之，个体之间通过不断的竞争，选择、交叉、变异，进化出一组更好的基因。
- 步骤：
  - 将一个群体的所有个体进行编码，这里的个体为权值表
  - 个体之间进行竞争（利用权值表，使用alpha-beta剪枝进行对弈），评估个体的适应度（对局的结果）
  - 根据适应度选择部分个体
  - 根据概率选择个体进行基因交叉、基因变异
  - 进化到下一代，重复步骤二
- 评估适应度的方法：
  - 适应度初始化为0，随机选择竞争、记录竞争结果，按结果对基因组进行排序
- 交叉：
  - 在适应度高的个体之间进行基因交叉
- 变异：
  - 对适应度低的个体进行基因变异，随机变异或者根据一定的算法变异。
- 流程图：



DQN：

- 增强学习的概念：一个agent在执行一个task的时候，需要观察当前状态而作出动作，从而取得最大化的利益。
- DQN原理分析：
  - Task执行过程中包含了一系列的动作**Action,**观察**Observation**还有反馈值**Reward**。
  - **回报**：代表当前状态所对应的回报$G_i = R_i + \lambda R_{i+1} + \lambda^2 R_{i+2} + \ldots$， 概念上认为当前的reward最重要，时间越久远对回报的影响越小。
  - **价值函数value function**：即回报的期望。
    - 最终的决策过程：通过估计价值函数，我们可以知道每个状态的优劣，从而进行决策。
  - **估计价值函数**的方法：
    - Bellman方程：计算回报的期望，最终推出了一个结论，价值函数是可以通过迭代计算的。

$$\begin{aligned} v(s) &= E(G_t | S_t = s) \\ &= E(R_{t+1} + \lambda R_{t+2} + \lambda^2 R_{t+3} + \ldots | S_t = s) \\ &= E(R_{t+1} + \lambda v(S_{t+1}) \mid S_t = s) \end{aligned}$$

  - **动作价值函数**：
    - 如果知道了每个动作对应的价值，即可选择动作价值最高的动作执行。
    - 定义一个在特定策略下的动作价值函数：

$$Q^{\pi}(s, a) = E[r_{t+1} + \lambda r_{t+1} + \ldots | s, a]$$

      - π为当前的策略，该公式表示的是当前动作执行之后得到的reward与之后的动作产生的reward的和的期望。
  - **最优价值函数**：$Q^* = E_{s'}[r + \lambda max_{a'}(Q^*(s', a'))|s, a]$
  - **价值迭代：**根据bellman方程进行迭代$v_{k+1} = E(R_{t+1} + \lambda v_k(S_{t+1}) \mid S_t = s)$
  - **Q-learning**：根据价值迭代的方法更新Q值，
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \lambda max_a(S_{t+1}, a) - Q(S_t, A_t)$
  - **DQN：**
    - 由于在许多问题中，任务的状态数量过多，而Q-learning算法需要存Q值矩阵，会出现维度灾难。Q-learning故不适用于大规模的问题，这里引入DQN的概念。

- 由于无法存储Q值，巧妙的采用一个函数来近似Q value function，即 $Q(s,a) \approx f(s,a)$
- 结合**神经网络**，将Q值转换为一个网络，其损失函数为

$$L = E[\ (r + \alpha max_{a'}\ (s', a', w)\ - Q(s, a, w)\ )^2]$$

- 训练方法：
  - 带有存储池的DQN，设定一个固定大小的存储池，将新的状态放置进去，每次学习的时候从其中随机抽取batch_size个状态进行学习。
-

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

（2）关键代码（带注释）

遗传算法

选择

- 输入：基因集合
- 随机抽取一定数量的个体进行竞争，记录对局结果，给每一个基因存储一个得分
- 最后根据得分对基因集合进行排序，以达到选择的目的

```
//选择出适应度高的个体
void select(vector<vector<int> >& gene_list){
    int index1, index2;
    //记录对弈得分
    vector<int> tmp_score_list(gene_list.size(), 0);
    //竞争20（个体总数）次
    for (int i = 0; i < gene_list.size(); i++){
        //每次随机选择2个个体，先后手各对弈一局
        index1 = random_gene(0, gene_list.size());
        do{
            index2 = random_gene(0, gene_list.size());
        }while(index1 == index2);

        int tmp_score = 0;
```

```
            //alpha-beta，记录先后手对局结果
            tmp_score += fight(gene_list[index1], gene_list[index2]);
            tmp_score -= fight(gene_list[index2], gene_list[index1]);

            //记录双方得分
            tmp_score_list[index1] += tmp_score;
            tmp_score_list[index2] -= tmp_score;
        }
        //根据得分将基因表排序
        for (int i = 0; i < gene_list.size(); i ++){
            for (int j = 0; j < gene_list.size() - i - 1; j ++){
                if(tmp_score_list[j] > tmp_score_list[j + 1]){
                    swap(tmp_score_list[j], tmp_score_list[j+1]);
                    swap(gene_list[j], gene_list[j+1]);
                }
            }
        }
        return;
    }
```

交叉

- 从得分较高的部分集合中随机抽取两个基因
- 比较两个基因的得分排位关系
- 代替的概率为 $\frac{size/2+abs(index_1-index_2)}{size}$
    - 比如size = 20, index1 = 10, index2 = 15, 用基因2代替基因1的概率为15/20 = 0.75

```
//交叉
void crossover(vector<vector<int> >& gene_list){
    int index1, index2;
    for (int i = gene_list.size() / 2; i < gene_list.size(); i++){
        /*从得分排名一半之前选择两个基因进行交叉*/
        index1 = random_gene(gene_list.size() / 2, gene_list.size());
        do{
            index2 = random_gene(gene_list.size() / 2, gene_list.size());
        }while(index1 == index2);
        //对于选择的两个基因，比较两者的得分，利用得分调整交叉替换的概率。
        for (int j = 0; j < N * N / 4; j++){
            int tmp = rand() % gene_list.size();
            int half = gene_list.size() / 2;
            //比较得分来确定交叉策略
            //index是排好序的，越大的证明得分越高，排序越高被选中的概率越高
            if(index1 < index2)
                gene_list[index1][j] = (tmp >  half + index2 - index1 ?
gene_list[index1][j]: gene_list[index2][j]);
            else
                gene_list[index2][j] = (tmp >  half + index1 - index2 ?
gene_list[index1][j]: gene_list[index2][j]);
```

```
        }
    }
    return;
}
```

变异

- 对得分较低的基因进行变异，两种方式
  - 用得分较高的进行替代
  - 按照一定的随机计算公式进行变异

```
//变异
void mutation(vector<vector<int> >& gene_list){
    int size = gene_list.size();
    int half = gene_list.size() / 2;
    for (int i = 0; i < half; i++){
        /*将排序前一半的基因进行变异*/
        int index = rand() % half + half;
        for (int j = 0; j < N * N / 4; j++){
            int tmp = rand() % size;
            //替换的概率为 10 + i2 - i1 / 20
            gene_list[i][j] = gene_list[index][j];
            if (tmp <= half + index - i)
                gene_list[i][j] += (rand() % 20 - 10);
            //判断是否越界
            gene_list[i][j] = gene_list[i][j] <= 0 ? 1 : gene_list[i][j];
            gene_list[i][j] = gene_list[i][j] >= 50 ? 49 : gene_list[i][j];
        }
    }
}
```

遗传算法训练

- 确定训练次数
- 每次按顺序调用几个过程，选择、交叉、变异

```
//遗传算法
void GA(vector<vector<int> > &gene_list, int epoch){
    init_gene(gene_list);
    int i = 0;
    while (i ++ <= epoch){
        select(gene_list);
        crossover(gene_list);
        mutation(gene_list);
        //输出epoch
        cout << "Epoch " << i << '/' << epoch << ' ' << endl;
        //输出基因集合
    }
}
```

DQN算法

state类：

- 当前的状态
- 下一个状态
- 收益
- 动作

```
class my_state:
    def __init__(self, state, next_state=None, reward=None, action=None):
        self.state = state
        self.nextState = next_state
        self.reward = reward
        self.action = action
```

DQN网络：

- 定义参数

```
class DeepQNetwork:
    def __init__(self,
                 n_actions=64,
                 reward_decay=0.9,
                 learning_rate=0.01,
                 batch_size=32,
                 upgrade_target_inter=100,
                 memory_size=200,
                 e_greedy=0.9):
        #定义参数
        self.learning_rate = learning_rate
        self.n_actions = n_actions
        self.gamma = reward_decay
        #epsilon可变化
```

```
        self.epsilon = e_greedy
        #随机梯度下降的
        self.batch_size = batch_size

        #设定学习次数
        self.learn_count = 0
        self.upgrade_target_inter = upgrade_target_inter
        self.build_network()

        self.memory = []
        self.memory_index = 0
        self.memory_size = memory_size

        #定义目标网络和估计网络的参数
        t_params = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
scope='target_net')
        e_params = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
scope='evaluate_net')

        with tf.variable_scope('hard_replacement'):
            self.target_replace_op = [tf.assign(t, e) for t, e in
zip(t_params, e_params)]

        #初始化sess
        self.saver = tf.train.Saver()
        self.sess = tf.Session()
        self.sess.run(tf.global_variables_initializer())
```

创建网络：

- 包括两个网络：均为简单的双层网络，输出值为：$q = w_2(w_1X + b_1) + b_2$
    - evaluate网络

```
'''evaluate_net'''
with tf.variable_scope('evaluate_net'):
'''重新决定网络结构'''
w1_e = tf.Variable(tf.truncated_normal(shape=[N*N, 100], mean=1,
stddev=0.1))
b1_e = tf.Variable(tf.constant(0.1, shape=[100]))
hidden_e = tf.nn.relu(tf.matmul(tf.reshape(self.state, shape=
[-1,N*N]), w1_e) + b1_e)

w2_e = tf.Variable(tf.truncated_normal(shape=[100, self.n_actions],
mean=1, stddev=0.1))
b2_e = tf.Variable(tf.constant(0.1, shape=[self.n_actions]))
self.q_eval = tf.matmul(hidden_e, w2_e) + b2_e
```

    - target网络

```
'''target_net'''
with tf.variable_scope('target_net'):
w1_t = tf.Variable(tf.truncated_normal(shape=[N*N, 100], mean=1,
stddev=0.1))
b1_t = tf.Variable(tf.constant(0.1, shape=[100]))
hidden_t = tf.nn.relu(tf.matmul(tf.reshape(self.state, shape=
[-1,N*N]), w1_t) + b1_t)

w2_t = tf.Variable(tf.truncated_normal(shape=[100, self.n_actions],
mean=1, stddev=0.1))
b2_t = tf.Variable(tf.constant(0.1, shape=[self.n_actions]))
self.q_next = tf.matmul(hidden_t, w2_t) + b2_t
```

- 计算损失函数：$loss = (q_{eval} - q_{target})^2$

```
def build_network(self):
        #定义自己的网络
        #默认一次输入一组数据
        self.state = tf.placeholder(tf.float32, shape=[N, N],
name='currentState')
        self.nextState = tf.placeholder(tf.float32, shape=[N, N],
name='nextState')
        self.reward = tf.placeholder(tf.float32, shape=[1], name='reward')
        self.action = tf.placeholder(tf.int32, shape=[1], name='action')

        '''evaluate_net'''
        '''target_net'''

        #计算'q_target'
        with tf.variable_scope('q_target'):
            q_target = self.reward + self.gamma *
tf.reduce_max(self.q_next, axis=1, name='Qmax_s_')    # shape=(None, )
            self.q_target = tf.stop_gradient(q_target)
        #计算'q_evaluate'
        with tf.variable_scope('q_eval'):
            a_indices = tf.stack([tf.range(tf.shape(self.action)[0],
dtype=tf.int32), self.action], axis=1)
            self.q_eval_wrt_a = tf.gather_nd(params=self.q_eval,
indices=a_indices)    # shape=(None, )
        #计算loss
        with tf.variable_scope('loss'):
            self.loss = tf.reduce_mean(tf.squared_difference(self.q_target,
self.q_eval_wrt_a, name='TD_error'))
        with tf.variable_scope('train'):
            self.train_op =
tf.train.RMSPropOptimizer(self.learning_rate).minimize(self.loss)
```

存储状态：

- 覆盖存储状态，用于之后的学习。

```
def store_transition(self, state, nextState, reward, action):
        print("memory:",len(self.memory))
        #memory库已满，从头开始覆盖
        if len(self.memory) == self.memory_size:
            self.memory[self.memory_index] = my_state(state, nextState,
reward, action)
            self.memory_index = self.memory_index + 1 if self.memory_index
+ 1 < self.memory_size else 0
        #向memory库中添加
        else:
            self.memory.append(my_state(state, nextState, reward, action))
```

选择动作：

- 设置一个ϵ，每次取一个随机数k
  - 若$k > \epsilon$，随机选择一个动作
  - 否则按照输入网络，算出q值序列，之后选择q值最高的动作

```
def choose_action(self, observation):
        if np.random.uniform() < self.epsilon:
            actions_value = self.sess.run(self.q_eval, feed_dict=
{self.state: observation})
        else:
            #随机选择
            actions_value = []
        return actions_value
```

学习：

- 利用evaluate网络更新target网络
- 随机梯度下降更新evaluate网络的参数

```
def learn(self):
        #更新target网络
        if self.learn_count % self.upgrade_target_inter:
            self.sess.run(self.target_replace_op)
            print('target_params_replaced')

        #随机梯度下降
        for i in range(self.batch_size):
            #从现有的memory中选择一条
            sample_index = np.random.choice(len(self.memory))
            tmp_memory = self.memory[sample_index]
            train_result, loss = self.sess.run(
                [self.train_op, self.loss],
                feed_dict={
```

```
                    self.state: tmp_memory.state,
                    self.nextState: tmp_memory.nextState,
                    self.reward: [tmp_memory.reward],
                    self.action: [tmp_memory.action]
                }
            )
        self.learn_count += 1
```

训练：

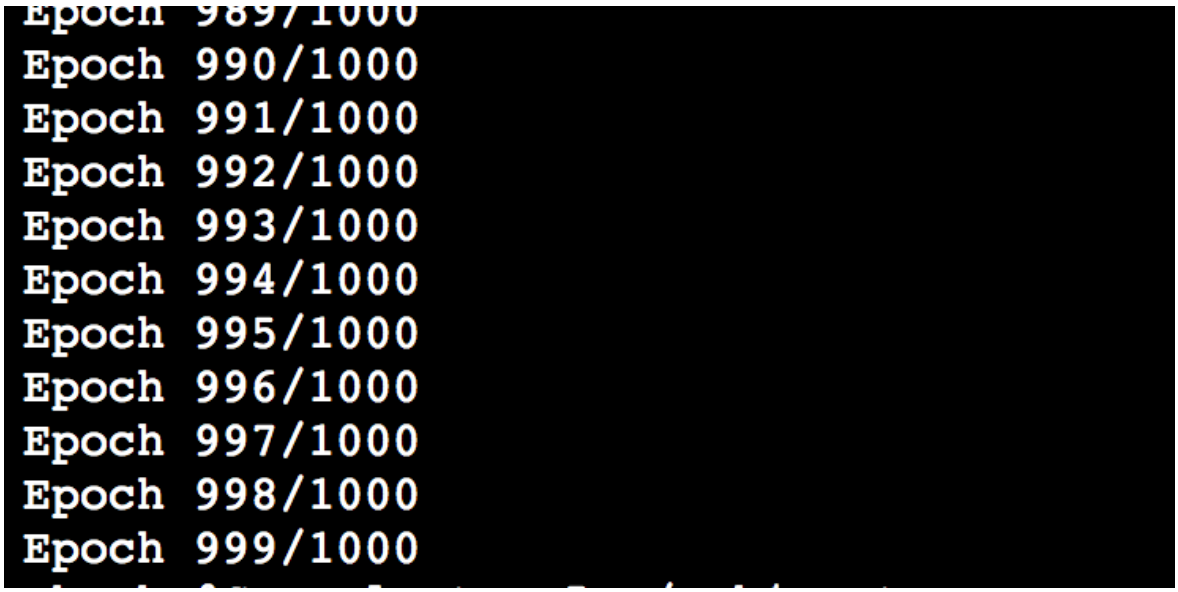
- 初始化两个DQN网络
- 每一轮进行对弈，间隔若干轮学习一次
- 保存模型参数

```
def trainDQN(epoch):
    AI1 = DeepQNetwork()
    AI2 = DeepQNetwork()
    for episode in range(epoch):
        start_DQN_game(AI1, AI2)
        if episode % 5 == 0:
            AI1.learn()
            AI2.learn()
        print("episode:", episode)
    AI1.save_model()
    AI2.save_model()
```

（3）实验结果展示

遗传算法结果：

- 训练过程：



- 最终的权值表结果：
    - 权值表使用棋盘的四分之一，与其他三块对称。

- 可以看到，利用alpha-beta进行对弈的训练，将最终权值表的稳定点的权值大幅提高，将危险子的权值大幅降低，且个体之间的特征十分相似。根据最终的权值表，判断训练的效果较好。

```
//仅展示种群中的三个个体
//0
38 13 37 1
5 4 7 5
48 4 1 8
19 11 1 1
//1
28 23 31 27
9 9 9 1
44 10 4 10
20 1 15 1
//2
46 42 29 20
10 4 17 1
44 1 1 1
11 17 12 1
//3
41 23 41 13
11 10 1 1
49 1 1 1
1 4 8 1
...
```

DQN训练：

- loss变化不明显，无法收敛，没有完成良好的训练。
- 初次接触tensoflow，对其框架、用法不熟悉，没有深入的了解调参的方法。
- 由于训练成本太高，未进行多次调参。

（4）评测指标

- rank中未使用DQN和遗传算法的训练结果。
- 本地测试DQN，与随机数的差异不大。
- 猜测遗传算法训练出来的权值矩阵效果较好，未经过测试。