



人工智能导论实验报告

学 院： 信息工程学院

指导教师： 任晨珊

班 级： 计算机科学与技术 2 班

学生姓名： 赵浚博

学 号： 19140179

日期： 2022 年 1 月 10 日

摘 要

本实验报告包括人工智能课程的全部 7 次实验，分别为 BFS/DFS 解决八数码难题 (8-puzzle problem)、启发式搜索 (A*) 解决八数码难题 8-puzzle problem)、alpha- β 博弈树解决井字棋 (tic-tac-toe)、遗传算法 (Genetic Algorithm) 解决多项式最大值、蚁群算法 (colony algorithm) 解决旅行商 (TSP) 问题，以及利用 pytorch 构建卷积神经网络 (CNN) 识别手写数字。本文涉及到的代码主要目的是理解掌握相关算法的流程，程序运行中大多展示了算法的各个步骤，并没有进一步的优化算法效率。

本文所有源代码已上传至 Github <https://www.github.com/BestJob2000/some-easy-AI-modules>

Abstract

This report includes all 7 experiments in the ai course, BFS/DFS solves the 8-puzzle problem, heuristic search solves the 8-puzzle problem, alpha- β game tree solves the Tic-Tac-TOE and Genetic algorithm, respectively In addition, pyTorch is used to construct convolutional neural network (CNN) to recognize handwritten numbers. The main purpose of the code involved in this paper is to understand the flow of the relevant algorithm. Most of the procedures in the running of the program show the various steps of the algorithm, and there is no further optimization of the efficiency of the algorithm.

All the source code has been uploaded to Github
<https://www.github.com/BestJob2000/some-easy-AI-modules>

目 录

实验一 盲目式搜索解决八数码难题.....	1
实验目的	1
八数码难题简介	1
实验原理	1
1. 八数码状态表示及操作	1
2. 深度优先搜索 (DFS)	1
3. 广度优先搜索 (BFS)	2
程序实现流程	2
算法主要模块	2
深度优先搜索	2
广度优先搜索	3
判断模块	4
显示模块	5
实验感想	5
实验二 启发式搜索解决八数码难题.....	5
算法介绍及主要模块	6
启发式搜索 (A*)	6
评估函数	7
Open 表/查询队列	7
算法主要流程	8
实验感想	8
实验三 博弈树 α - β 剪枝解决井字棋	8
实验目的	8
实验原理	8
井字棋的状态空间表示方法	8
博弈树	8
alpha-beta 剪枝	9
算法主要模块	10
检测结束模块	10
评估模块	10
博弈树剪枝模块	11

实验感想	12
实验四 遗传算法	12
实验目的	12
实验原理	12
1、遗传算法的基本思想	12
2、二进制编码	13
3、适应度函数	13
4、选择	13
5、交叉	14
6、变异	14
7、保留最佳个体	14
8、停机准则	14
算法实现流程	15
算法主要模块	16
解码	16
评估	16
初始化	16
选择	17
交叉	17
变异	18
实验结果及感想	18
实验五 蚁群算法	19
实验目的	19
实验原理	19
算法实现流程	22
算法主要模块	22
计算距离	22
期望矩阵	23
蚁群算法	23
迭代最短路径和距离	23
更新信息素矩阵	24
实验结果及感想	24
实验六 基于 MNIST 数据集的手写数字识别	25
实验目的	25
实验原理	25
算法实现流程	25

训练环境及实验结果	27
实验环境配置:	27
模型参数配置:	27
训练结果:	28
手动绘制图像训练结果	29
实验感想	29
七、实验总结	29

实验一 盲目式搜索解决八数码难题

实验目的

使用盲目式搜索：宽度优先搜索、深度优先搜索，来解决八数码难题。

八数码难题简介

八方块移动游戏要求从一个含 8 个数字（用 1~8 表示）的方块以及一个空格方块（用 0 表示）的 3x3 矩阵的起始状态开始，不断移动该空格方块以使其和相邻的方块互换，直至达到所定义的目标状态。空格方块在中间位置时有上、下、左、右 4 个方向可移动，在四个角落上有 2 个方向可移动，在其他位置上有 3 个方向可移动。例如，假设一个 3x3 矩阵的初始状态为：

```
8 0 3
2 1 4
7 6 5
```

目标状态为：

```
1 2 3
8 0 4
7 6 5
```

实验原理

1. 八数码状态表示及操作

八数码的目前状态用一个 5x5 的数组表示周围多余部分设为 0，以方便后续的移动操作。八数码除了保存当前的棋盘状态，还应保存当前空格位置，即 0 点的 x y 坐标。以及当前步数 t。将这三个部分封装为一个结构体 node。

判断是否成功使用当前棋盘与目标棋盘进行比较，完全符合则达到目标状态。

移动空格位置需要判断移动是否合法，只有移动的位置非 0 才可以移动，移动后修改棋盘及空格的 xy 坐标。

2. 深度优先搜索 (DFS)

深度优先搜索属于图算法的一种，是一个针对图和树的遍历算法，英文缩写为 DFS 即 Depth First Search。深度优先搜索是图论中的经典算法，利用深度优先搜索算法可以产生目标图的相应拓扑排序表，利用拓扑排序表可以方便的解决很多相关的图论问题，如最大路径

问题等等。一般用堆数据结构来辅助实现 DFS 算法。其过程简要来说是对每一个可能的分支路径深入到不能再深入为止，而且每个节点只能访问一次。

3. 广度优先搜索 (BFS)

广度优先搜索（也称宽度优先搜索，缩写 BFS，以下采用广度来描述）是连通图的一种遍历算法这一算法也是很多重要的图的算法的原型。Dijkstra 单源最短路径算法和 Prim 最小生成树算法都采用了和宽度优先搜索类似的思想。其别名又叫 BFS，属于一种盲目搜寻法，目的是系统地展开并检查图中的所有节点，以找寻结果。换句话说，它并不考虑结果的可能位置，彻底地搜索整张图，直到找到结果为止。基本过程，BFS 是从根节点开始，沿着树(图)的宽度遍历树(图)的节点。如果所有节点均被访问，则算法中止。一般用队列数据结构来辅助实现 BFS 算法。

程序实现流程

- ① 从键盘读入当前棋局
- ② 广度优先搜索创建队列/深度优先搜索递归 自动生成栈
- ③ 进行递推/递归 并与目标进行比较
- ④ 达到目标退出/达到最大次数退出
- ⑤ 倒推路径 并显示

算法主要模块

深度优先搜索

```
bool dfs(node move)
{
    if(move.t>5) return false;
    if(check(move)==true)
    {
        dfs_steps=move.t;
        putin=move.t;
        for(int j=1;j<=3;j++){
            for(int k=1;k<=3;k++){
                dfs_ans[putin].map[j][k]=move.map[j][k];
            }
        }
        putin--;
        return true;
    }
    node newmove;
```

```

for(int i=0;i<=3;i++)      //不同分支
{
    int nx=move.wi+dx[i];
    int ny=move.wj+dy[i];
    if(nx>=1 && nx<=3 && ny>=1 && ny<=3) //交换点位置合规
    {
        for(int j=1;j<=3;j++)
        {
            for(int k=1;k<=3;k++)
            {
                newmove.map[j][k]=move.map[j][k];
            }
        }
        newmove.map[move.wi][move.wj]=move.map[nx][ny];
        newmove.map[nx][ny]=0;          //空格位置
        newmove.t=move.t+1;             //步数
        newmove.wi=nx;                  // 新的 x、y 位置
        newmove.wj=ny;
    }
    if (dfs(newmove)==true){
        for(int j=1;j<=3;j++){
            for(int k=1;k<=3;k++){
                dfs_ans[putin].map[j][k]=move.map[j][k];
            }
        }
        putin--;
        return true;
    }
}
return false;
}

```

广度优先搜索

```

void bfs()
{
    int head=1,tail=2;
    while(head<tail)
    {
        if(check(bfs_queue[head])==true)
        {
            bfs_steps=bfs_queue[head].t;

```



```

        printf("BFS steps %d",bfs_queue[head].t);
        int b=head;
        bfs_ans[bfs_steps]=head;
        for(int m=bfs_steps-1;m>=0;m--){
            bfs_ans[m]=bfs_back[b];
            b=bfs_back[b];
        }
        return ;
    }
    for(int i=0;i<=3;i++)
    {
        int nx=bfs_queue[head].wi+dx[i];
        int ny=bfs_queue[head].wj+dy[i];
        if(nx>=1 && nx<=3 && ny>=1 && ny<=3) //交换点位置合规
        {
            for(int j=1;j<=3;j++)
            {
                for(int k=1;k<=3;k++)
                {
                    bfs_queue[tail].map[j][k]=bfs_queue[head].map[j][k];
                    bfs_back[tail]=head;
                }
            }

            bfs_queue[tail].map[bfs_queue[head].wi][bfs_queue[head].wj]=bfs_queue[head].map[nx][ny];
            bfs_queue[tail].map[nx][ny]=0;           //空格位置
            bfs_queue[tail].t=bfs_queue[head].t+1;     //步数+1
            bfs_queue[tail].wi=nx;                     // 新的 x、y 位置
            bfs_queue[tail].wj=ny;
            tail++;
        }
    }
    head++;
}
}

```

判断模块

```

bool check(node move){
    for(int i=1;i<=3;i++)
    {
        for(int j=1;j<=3;j++)

```

```
{
    if(move.map[i][j]!=target[i][j])
    {
        return false;
    }
}
return true;
}
```

显示模块

```
printf("\ndfs steps %d",dfs_steps);
for(int i=0;i<=dfs_steps;i++){           //show steps
    printf("\nstep %d\n",i);
    for(int j=1;j<=3;j++){
        for(int k=1;k<=3;k++){
            printf("%d ",dfs_ans[i].map[j][k]);
        }
        printf("\n");
    }
    printf("\n\n");
}
```

实验感想

DFS 与 BFS 都是数据结构课程中讲授过的知识,但是当时只是对理论有了初步的了解,在这次实验中,真正动手去实现一个具体的问题时,还是遇到了很多的困难。好在都在老师和同学们的帮助下解决了。在真正实现了 DFS 与 BFS 搜索后,对 DFS BFS 的概念有了更加深刻地认识。

实验二 启发式搜索解决八数码难题

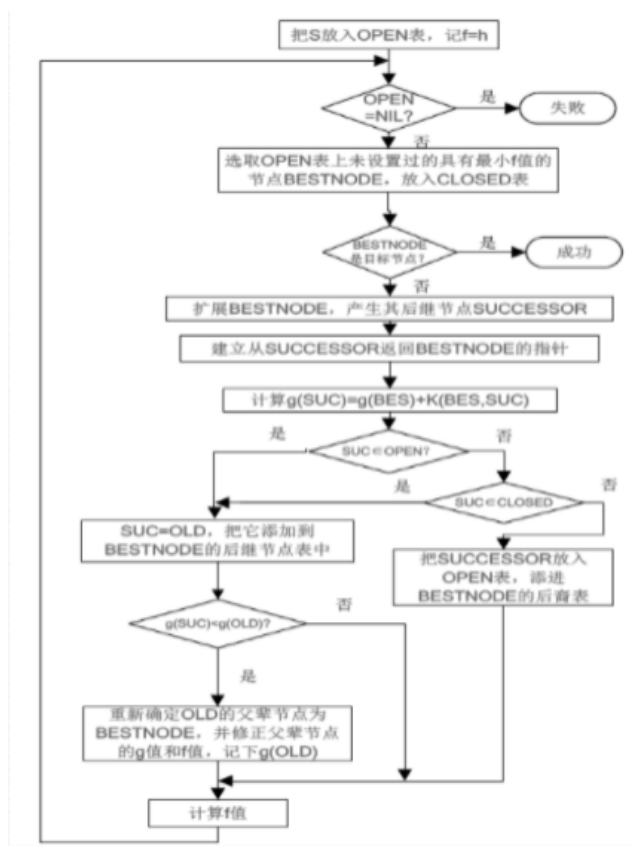
实验目的与原理与实验一相同,将算法改为启发式搜索 (A*)

算法介绍及主要模块

启发式搜索 (A*)

启发式策略可以通过指导搜索向最有希望的方向前进, 降低了复杂性。通过删除某些状态及其延伸, [启发式算法](#)可以消除组合爆炸, 并得到令人能接受的解(通常并不一定是最佳解)。

然而, 启发式策略是极易出错的。在解决问题的过程中启发仅仅是下一步将要采取措施的一个猜想, 常常根据经验和直觉来判断。由于启发式搜索只有有限的信息(比如当前状态的描述), 要想预测进一步搜索过程中状态空间的具体行为则很难。一个启发式搜索可能得到一个次最佳解, 也可能一无所获。这是启发式搜索固有的局限性。这种局限性不可能由所谓更好的启发式策略或更有效的搜索算法来消除。一般说来, 启发信息越强, [扩展](#)的无用节点就越少。引入强的启发信息, 有可能大大降低搜索工作量, 但不能保证找到最小耗散值的解路径(最佳路径)。因此, 在实际应用中, 最好能引入降低搜索工作量的启发信息而不牺牲找到最佳路径的保证。



评估函数

$$f(x) = g(x) + h(x)$$

本次八数码难题使用了两种评估函数

评估函数 1 在位的数字数目：

```
int h1(int move[5][5]){
    int result=0;
    for(int i=1;i<=3;i++){for(int j=1;j<=3;j++){
        if(move[i][j]!=target[i][j]) result++;
    }}
    return result;
}
```

评估函数 2 每个数字与其目标位置的距离和：

```
int h2(int move[5][5]){
    int result=0;
    for(int i=1;i<=3;i++){for(int j=1;j<=3;j++){
        if(move[i][j]!=target[i][j]) {
            switch(move[i][j]){
                case 1: result+=abs(i-1)+abs(j-1);break;
                case 2: result+=abs(i-1)+abs(j-2);break;
                case 3: result+=abs(i-1)+abs(j-3);break;
                case 4: result+=abs(i-2)+abs(j-1);break;
                case 5: result+=abs(i-2)+abs(j-3);break;
                case 6: result+=abs(i-3)+abs(j-1);break;
                case 7: result+=abs(i-3)+abs(j-2);break;
                case 8: result+=abs(i-3)+abs(j-3);break;
            }
        }
    }}
    return result;
}
```

Open 表/查询队列

将当前节点所有的可能下一步状态及其评估值放入 open 表中，并将最大的放在表首。

算法主要流程

启发式搜索与 BFS 的整体思路相似，都是使用数组模拟的队列来充当 open 表。区别在于启发式搜索要选取评估函数值最高的进行。但将整个 open 表排序是不明智的，因为后续的一些节点可能根本不会被用到。所以我选择每次选取节点前将评估值最高的放到 open 表首，这样满足了启发式搜索的目标，又降低了程序的复杂度。

实验感想

通过这次实验，我完整的写出了启发式搜索的全过程。在实验的过程中遇到了很多困难，大多数是由于对启发式搜索的流程不熟悉，以及相关排序等操作流程设计的不够周全。在完整的写出了启发式搜索后，对启发式搜索，评估函数的设定都有了更深入的了解。

实验三 博弈树 α - β 剪枝解决井字棋

实验目的

利用 alpha-beta 剪枝的博弈树解决一字棋博弈问题。

实验原理

井字棋的状态空间表示方法

3*3 的棋盘利用 3*3 的二维数组

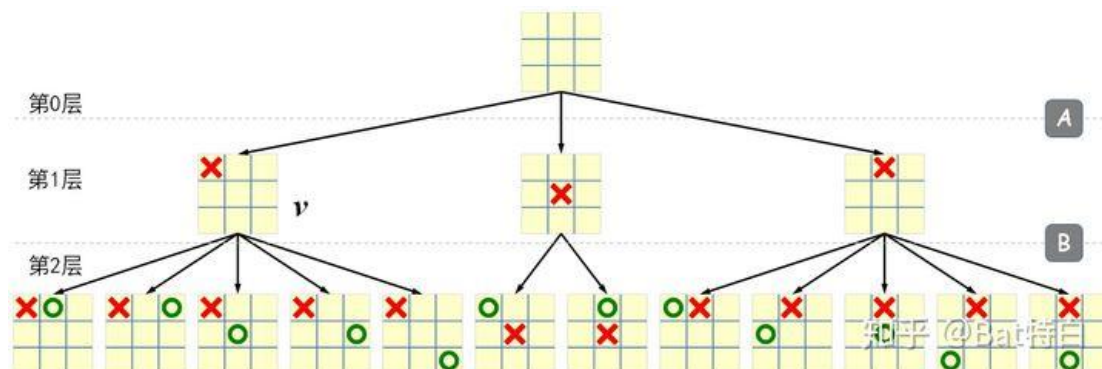
博弈树

棋类游戏中所有策略的集合可以由一棵巨大的根树来表示，树中的每个顶点都对应于棋盘的一个布局（configuration）。所谓“棋盘的一个布局”，是在游戏过程中通过标记棋盘上的小方格得到的。

在此树中，根对应于所有小方格都为空的布局。继而，游戏者 A 行棋——她有 9 个选择。因此，根有 9 个孩子顶点，每个孩子顶点对应于在 9 个小方格之一内写下符号 \times 。考虑根的一个孩子顶点 v （棋盘上恰有一个符号 \times ）。在 v 处，轮到游戏者 B 行棋。B 有 8 个选择，因此 v 将有 8 个孩子顶点（参看下图，为简洁起见，忽略掉了对称的情况）。

不断画下去可以看到这棵树至多有 9 层，其中偶数层（根所在的最高层记做 0 层）对应着轮到游戏者 A 行棋，而奇数层对应着轮到游戏者 B 行棋。有三个同类符号位于同一直线上的顶点对应于有一个游戏者获胜的情况，这样的顶点是该根树的叶子。类似的，所有九个

小方格都被标记的节点也是叶子节点（它可能对应于无人获胜的情况，如上图 9.1 所示）。这就是该游戏的博弈树，也称作对策树或游戏树。



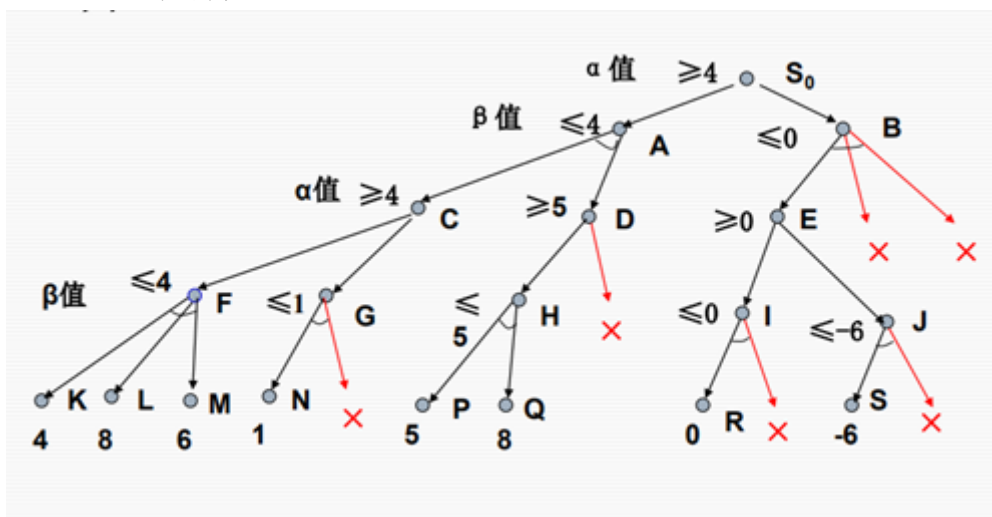
alpha-beta 剪枝

搜索策略：K一步博弈；深度优先；每次扩展一个节点；一边扩展一边评估

对于一个与节点来说，它取当前子节点中的最小倒推值作为它倒推值的上界，称此为 β 值（ $\beta \leq \text{最小值}$ ）
 对于一个或节点来说，它取当前子节点中的最大倒推值作为它倒推值的下界，称此为 α 值（ $\alpha \geq \text{最大值}$ ）

一字棋

- α : Max 节点估价值的下界
- β : Min 节点估价值的上界



算法主要模块

检测结束模块

```
def check(board):
    for i in range(0,3):
        if board[i][0]==board[i][1] and board[i][1]==board[i][2]:    #line
            if board[i][0]==1:
                return 1
            if board[i][0]==2:
                return 2
        if board[0][i]==board[1][i] and board[1][i]==board[2][i]:    #row
            if board[0][i]==1:
                return 1
            if board[0][i]==2:
                return 2
        if board[0][0]==board[1][1] and board[1][1]==board[2][2]:    #dia
            if board[1][1]==1:
                return 1
            if board[1][1]==2:
                return 2
        if board[0][2]==board[1][1] and board[1][1]==board[2][0]:    #anti-dia
            if board[1][1]==1:
                return 1
            if board[1][1]==2:
                return 2
    if isfull(board):
        return 3    #draw
    return 0
```

评估模块

```
def eval(board):
    if check(board)==2:
        return 99
    if check(board)==1:
        return -99
    if check(board)==3:
        return 0
    pc_holds=0
    human_holds=0
    for i in range(0,3):
        tmp_lines=[]    #lines
        tmp_rows=[]    #rows
```

```

tmp_lines.append(board[i][0])
tmp_lines.append(board[i][1])
tmp_lines.append(board[i][2])
tmp_rows.append(board[0][i])
tmp_rows.append(board[1][i])
tmp_rows.append(board[2][i])
if (1 not in tmp_lines) and (2 in tmp_lines):
    pc_holds+=1
elif (2 not in tmp_lines) and (1 in tmp_lines):
    human_holds+=1
if (1 not in tmp_rows) and (2 in tmp_rows):
    pc_holds+=1
elif (2 not in tmp_rows) and (1 in tmp_rows):
    human_holds+=1
tmp_dia=[board[0][0],board[1][1],board[2][2]]
tmp_antidia=[board[0][2],board[1][1],board[2][0]]
if (1 not in tmp_dia) and (2 in tmp_dia):
    pc_holds+=1
elif (2 not in tmp_dia) and (1 in tmp_dia):
    human_holds+=1
if (1 not in tmp_antidia) and (2 in tmp_dia):
    pc_holds+=1
elif (2 not in tmp_antidia) and (1 in tmp_dia):
    human_holds+=1
return (pc_holds-human_holds)*10

```

博弈树剪枝模块

```

def minMax(board,player,layer,av,alpha,beta):
    if check(board)!=0:
        return eval(board)
    if player==1:      #human moves
        for i in av:
            av_1=copy.deepcopy(av)
            av_1.remove(i)
            tmp=minMax(move(board,i,1),2,layer+1,av_1,alpha,beta)
            if tmp<beta:
                beta=tmp
            if beta<=alpha:
                return alpha
        return beta

```



```

if player==2:      #computer moves
    for i in av:
        av_1=copy.deepcopy(av)
        av_1.remove(i)
        tmp=minMax(move(board,i,2),1,layer+1,av_1,alpha,beta)
        if tmp>alpha:
            alpha=tmp
        if alpha>=beta:
            return beta
    return alpha

```

实验感想

α - β 剪枝规则之前也是在算法课程中学习过，但是并没有真正的写出一个完整的包含剪枝的博弈树程序。通过这次实验，真正的完成了一个博弈树的构建，使得我更深刻的了解了 α - β 剪枝规则，及实现的各个细节。在实验过程中遇到很多困难，通过参考老师的资料和网上的一些代码，终于将这个实验完成，受益匪浅。

实验四 遗传算法

实验目的

用遗传算法求解例题 2 优化问题。设计种群大小、染色体长度、交叉变异概率等。

➤ 例2：已知函数 $y = f(x_1, x_2) = \frac{1}{x_1^2 + x_2^2 + 1}$ ，其中
 $-5 \leq x_1, x_2 \leq 5$ ，用遗传算法求解 y 的最大值。

实验原理

1、遗传算法的基本思想

遗传算法是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。遗传算法是从代表问题可能潜在的解集的一个种群开始的，而一个种群则由经过基因编码的一定数目的个体组成，每个个体实际上是染色体带有特征的实体。在一开始需要实

现从表现型到基因型的映射即编码工作。由于仿照基因编码的工作很复杂，我们往往进行简化，如二进制编码。初代种群产生之后，按照适者生存和优胜劣汰的原理，逐代演化产生出越来越好的近似解，在每一代，根据问题域中个体的适应度大小选择个体，并借助于自然遗传学的遗传算子进行组合交叉和变异，产生出代表新的解集的种群。这个过程将导致种群像自然进化一样的后生代种群比前代更加适应于环境，末代种群中的最优个体经过解码，可以作为问题近似最优解。

2、二进制编码

它所构成的个体基因是一个由0或1组成的编码串。假设解的取值范围为： $[X_{min}, X_{max}]$ ，某一个个体编码是 $b_1 b_{i-1} b_i b_{i+1} \dots b_L$ ，则对应的解码公式为：

$$x = x_{min} + \delta \left(\sum_{i=1}^L b_i 2^{i-1} \right)$$

串长 L 取决于求解精度：

$$\delta = \frac{x_{max} - x_{min}}{2^{L-1}}$$

在本实验中，由于函数是二元的，所以一个个体 (x_1, x_2) 需要两个二进制编码分别代表 x_1 和 x_2 。

3、适应度函数

用来评估个体优劣的标准，适应度值越大个体越优。将目标函数映射成适应度函数的方法：(1) 若目标函数为最大化问题，则 $Fit(f(x)) = f(x)$ 。(2) 若目标函数为最小化问题，则 $Fit(f(x)) = 1/f(x)$ 。 x 即为二进制编码解码后的数值。在本实验中，是要求解函数的最大值所以适应度函数为

$$Fit(x_1, x_2) = \frac{1}{x_1^2 + x_2^2 + 1}$$

4、选择

本实验使用了轮盘赌选择，按个体的选择概率产生一个轮盘，轮盘每个区的

角度与个体的选择概率成比例。产生一个随机数，它落入转盘的哪个区域就选择相应的个体交叉选择概率即个体的适应度除以总体适应度的总和。

5、交叉

(1)交叉操作选中的两个父代个体交换某些基因位形成子代个体的过程。(2)交叉概率 P_c 在种群中，个体被选择出进行交叉的概率。在进行交叉操作时，可以先产生一个随机数，比较该随机数与交叉概率，如果该数小于交叉概率，则进行交叉。(3)交叉方式:本实验使用了单点交叉，它是先在两个父代个体的编码串中随机设定一个交叉点，然后对这两个父代个体交叉点后面部分的基因进行交换，生成子代中的两个新的个体。

6、变异

(1)变异操作编码按小概率扰动产生的变化，类似于基因的突变。(2)变异概率 P_m 控制算法中变异操作的使用频率。(3)变异方式本实验使用单点变异。首先产生一个随机数，比较该随机数与变异概率，如果该数小于变异概率，则进行变异。在变异时，在下标范围内产生一个随机数，作为变异的基因位，将 0 变 1 或将 1 变 0。

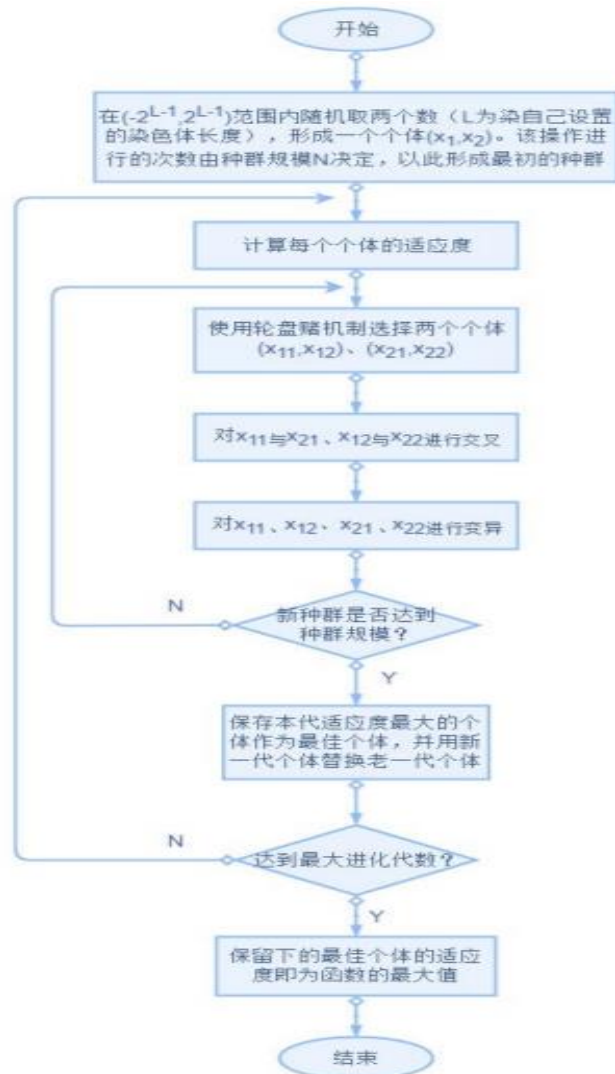
7、保留最佳个体

在使用新一代种群中替换老一代种群之前，保存老一代种群中适应度最高的个体，使得最终能收敛到全局最优解。

8、停机准则

本实验通过设置最大进化代数来实现停机，当进化代数达到最大进化代数时，则停止进化。

算法实现流程



算法主要模块

解码

```
def decode(l):  
    x=0  
    y=0  
    for i in range(10):  
        x+=l[i]*(2**(9-i))  
    for i in range(10,20):  
        y+=l[i]*(2**(19-i))  
    x=round(-5+x/100,2)  
    y=round(-5+y/100,2)  
    return x,y
```

评估

```
def eval(a,b):  
    return round(float(1/(a**2+b**2+1)),6)
```

初始化

```
#Init  
population=[[ ] for i in range(group_size)]  
for i in range(group_size):  
    for j in range(20):  
        population[i].append(random.randint(0,1))  
  
gennum=0
```

选择

```

#Select
all_eval=[]
eval_sum=0
new_population=[]
for i in population:
    a,b=decode(i)
    all_eval.append(eval(a,b))
    eval_sum+=eval(a,b)
for i in range(group_size):          #计算累加概率
    all_eval[i]=all_eval[i]/eval_sum
    if i>0:
        all_eval[i]=all_eval[i-1]+all_eval[i]
for i in range(group_size):          #进行轮盘赌
    r=random.random()
    for j in range(group_size):
        if r<all_eval[j]:
            new_population.append(copy.deepcopy(population[j]))
            break
population=copy.deepcopy(new_population)
print("\nAfter selection")
all_eval=[]
for i in population:
    a,b=decode(i)
    print(a,b,eval(a,b))
    all_eval.append(eval(a,b))
print(max(all_eval),min(all_eval))

```

交叉

```

#交叉 交叉概率0.6
if random.random()>0.4:
    for i in range(1,group_size,2):
        r=random.randint(0,19)
        population[i-1][r],population[i][r]=population[i][r],population[i-1][r]
    print("\nAfter intersection")
    all_eval=[]
    for i in population:
        a,b=decode(i)
        print(a,b,eval(a,b))
        all_eval.append(eval(a,b))
    print(max(all_eval),min(all_eval))

```

变异

```

#变异 概率0.4
if random.random()>0.6:
    for j in range(int(group_size/10)):
        ra=random.randint(0,group_size-1)
        rb=random.randint(0,19)
        population[ra][rb]=1-population[ra][rb]
        ra=random.randint(0,group_size-1)
        rb=random.randint(0,19)
        population[ra][rb]=1-population[ra][rb]
    print("\nAfter 变异")
    all_eval=[]
    for i in population:
        a,b=decode(i)
        print(a,b,eval(a,b))
        all_eval.append(eval(a,b))
    print(max(all_eval),min(all_eval))

```

实验结果及感想

```

0.3 -0.35 0.824742
0.12 -0.39 0.857265
0.12 -0.23 0.936944
0.3 -0.35 0.824742
0.3 -0.35 0.824742
0.14 -0.39 0.853461
0.936944 0.041941

Finish
0.12 -0.23 0.936944
PS C:\Users\MSI-Job>

```

这次的实验网络上并没有相关的程序可以参考,于是按照遗传算法的理论一步一步的去实现相关模块,并实验。在初期,由于参数设定的问题,经常会陷入局部最优解的。于是进一步增加了变异的概率,增加了选择的范围,使得程序最终能达到相对较好的性能。这次实验使得我对遗传算法有了更加完整的认知,对相关参数的设计与结果的产生有了深入的了解。

实验五 蚁群算法

实验目的

用蚁群算法求解下列 TSP 问题。设计蚂蚁的数量 m 、城市的数量 n 、重要程度因子 α 和 β ，信息素挥发度等。

31 个城市坐标如下：

$x=[1304, 3639, 4177, 3712, 3488, 3326, 3238, 4196, 4312, 4386, 3007, 2562, 2788, 2381, 1332, 3715, 3918, 4061, 3780, 3676, 4029, 4263, 3429, 3507, 3394, 3439, 2935, 3140, 2545, 2778, 2370]$;
 $y=[2312, 1315, 2244, 1399, 1535, 1556, 1229, 1004, 790, 570, 1970, 1756, 1491, 1676, 695, 1678, 2179, 2370, 2212, 2578, 2838, 2931, 1908, 2367, 2643, 3201, 3240, 3550, 2357, 2826, 2975]$

实验原理

蚁群算法是以旅行商问题作为应用实例提出的。旅行商问题，即 TSP 问题 (Traveling Salesman Problem)，又译为旅行推销员问题、货郎担问题，是数学领域中的著名问题之一。假设有一个旅行商人要拜访 n 个城市，他必须选择要走的路径，路径的限制是每个城市只能拜访一次，而且最后要回到原来出发的城市。可选的路径方案有很多，而 TSP 问题的目标是希望选出所有路径之中路程最短的路径方案。

□ 路径构建

- 伪随机比例选择规则

$$P_k(i, j) = \begin{cases} \frac{[\tau(i, j)]^\alpha [\eta(i, j)]^\beta}{\sum_{u \in J_k(i)} [\tau(i, u)]^\alpha [\eta(i, u)]^\beta} & \text{if } j \in J_k(i) \\ 0 & \text{其他} \end{cases}$$

- 公式描述了第 k 只蚂蚁位于城市 i 时选择城市 j 作为下一个访问城市的概率。对于每只蚂蚁 k ，路径记忆向量 R^k 按照访问顺序记录了所有 k 已经经过的城市序号（禁忌表）。 $J_k(i)$ 表示从城市 i 可以直接到达的、且又不在蚂蚁访问过的城市序列 R^k 中的城市集合。 $\eta(i, j)$ 是一个启发式信息，通常由 $\eta(i, j) = 1/d_{ij}$ 直接计算。 $\tau(i, j)$ 表示边 (i, j) 上的信息素量。

- 长度越短、信息素浓度越大的边被蚂蚁选择的概率越大。 α 和 β 是两个预先设置的参数，用来控制信息素浓度与启发式信息的重要程度。当 $\alpha=0$ 时，算法演变成随机贪心算法，即距离城市 i 最近的城市被选中的概率最大。当 $\beta=0$ 时，蚂蚁完全只根据信息素浓度确定路径，算法将快速收敛，这样构建出的路径往往与实际最优路径有着较大的差异，算法的性能不够好。

□ 信息素更新

1. 在算法初始化时，问题空间中所有边上的信息素都被初始化为 τ_0
2. 算法迭代每一轮，问题空间中的所有路径上的信息素都会发生挥发，我们为所有边上的信息素乘一个小于1的常数 ρ 。信息素挥发在自然界是信息素本身固有的特征，在算法中能够帮助避免信息素的无限积累，也使得算法可以快速丢弃之前构建过的较差的路径
3. 蚂蚁根据自己构建的路径长度在它们本轮经过的边上释放信息素。蚂蚁构建的路径越短、释放的信息素就越多。一条边被蚂蚁爬过的次数越多、它所获得的信息素也越多

□ 信息素更新

$$\tau(i, j) = (1 - \rho) \cdot \tau(i, j) + \sum_{k=1}^m \Delta\tau_k(i, j)$$



M. Dorigo给出 $\Delta\tau_k(i, j)$ 的三种不同模型

- m 是蚂蚁个数， ρ 是信息素的挥发率，规定 $0 < \rho \leq 1$ 。 $\Delta\tau_k(i, j)$ 是第 k 只蚂蚁在它经过的边上释放的信息素量

1. 蚂蚁圈系统 (Ant-cycle System)

单只蚂蚁所访问路径上的信息素浓度更新规则为：

$$\Delta\tau_k(i,j) = \begin{cases} \frac{Q}{L_k} & \text{若第}k\text{只蚂蚁在本次循环中从}x\text{到}y \\ 0 & \text{否则} \end{cases}$$

其中：

$\tau(i,j)$ 为当前路径上的信息素

$\Delta\tau_k(i,j)$ 第 k 只蚂蚁留在路径 (x,y) 上的信息素的增量

Q 为常数

L_k 为优化问题的目标函数值，表示第 k 只蚂蚁在本次循环中所走路径的长度，是 R^k 中所有边的长度和

信息素启发因子 α

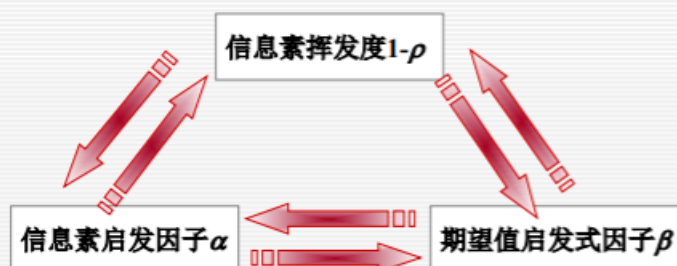
- 反映了蚁群在路径搜索中随机性因素作用的强度；
- α 值越大，蚂蚁选择以前走过的路径的可能性越大，搜索的随机性减弱；
- 当 α 过大时会使蚁群的搜索过早陷于局部最优。

期望值启发式因子 β

- 反映了蚁群在路径搜索中先验性、确定性因素作用的强度；
- β 值越大，蚂蚁在某个局部点上选择局部最短路径的可能性越大；
- 虽然搜索的收敛速度得以加快，但蚁群在最优路径的搜索过程中随机性减弱，易于陷入局部最优。

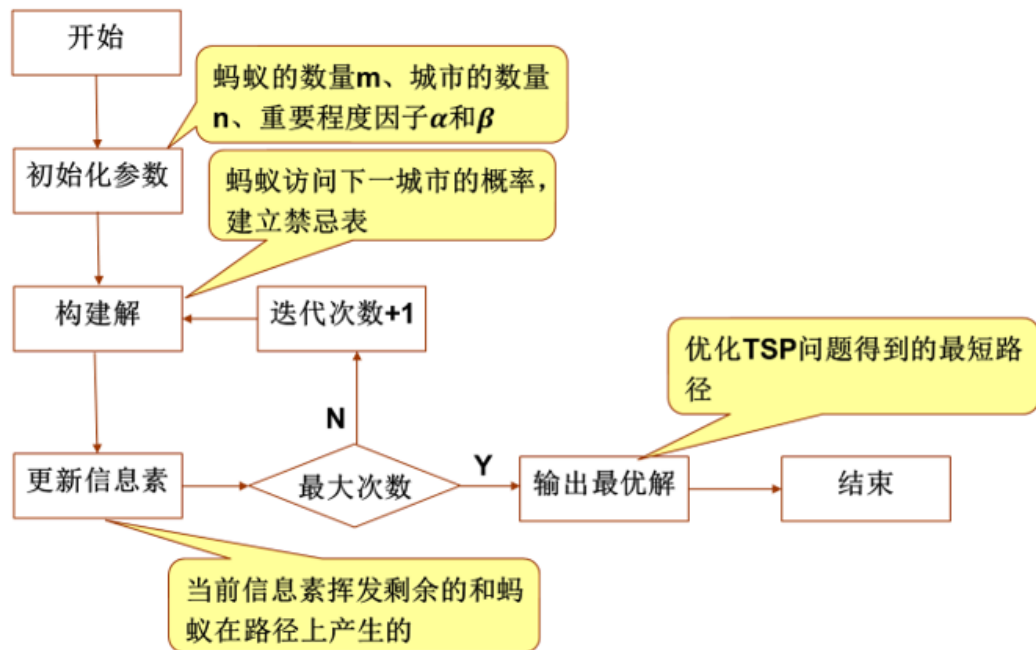
信息素挥发度 $1-\rho$

- 当要处理的问题规模比较大时，会使那些从来未被搜索到的路径(可行解)上的信息量减小到接近于0，因而降低了算法的全局搜索能力；
- 而且当 $1-\rho$ 过大时，以前搜索过的路径被再次选择的可能性过大，也会影响到算法的随机性能和全局搜索能力；
- 反之，通过减小信息素挥发度 $1-\rho$ 虽然可以提高算法的随机性能和全局搜索能力，但又会使算法的收敛速度降低。



275

算法实现流程



算法主要模块

计算距离

```

#对称矩阵，两个城市之间的距离
def distance_p2p_mat():
    dis_mat=[]
    for i in range(num_city):
        dis_mat_each=[]
        for j in range(num_city):
            dis=math.sqrt(pow(location[i][0]-location[j][0],2)+pow(location[i][1]-location[j][1],2))
            dis_mat_each.append(dis)
        dis_mat.append(dis_mat_each)
    return dis_mat

#计算所有路径对应的距离
def cal_newpath(dis_mat,path_new):
    dis_list=[]
    for each in path_new:
        dis=0
        for j in range(num_city-1):
            dis=dis_mat[each[j]][each[j+1]]+dis
        dis=dis_mat[each[num_city-1]][each[0]]+dis#回家
        dis_list.append(dis)
    return dis_list
  
```

期望矩阵


```
#期望矩阵
e_mat_init=1.0/(dis_mat+np.diag([10000]*num_city))#加对角阵是因为除数不能是0
diag=np.diag([1.0/10000]*num_city)
e_mat=e_mat_init-diag#还是把对角元素变成0
#初始化每条边的信息素浓度，全1矩阵
pheromone_mat=np.ones((num_city,num_city))
#初始化每只蚂蚁路径，都从0城市出发
path_mat=np.zeros((num_ant,num_city)).astype(int)
```

蚁群算法

```
while count_iter < iter_max:
    for ant in range(num_ant):
        visit=0#都从0城市出发
        unvisit_list=list(range(1,num_city))#未访问的城市
        for j in range(1,num_city):
            #轮盘法选择下一个城市
            trans_list=[]
            tran_sum=0
            trans=0
            for k in range(len(unvisit_list)): (variable) visit: int
                trans +=np.power(pheromone_mat[visit][unvisit_list[k]],alpha)*np.power(e_mat[visit][unvisit_list[k]],beta)
                trans_list.append(trans)
                tran_sum =trans
            rand=random.uniform(0,tran_sum)#产生随机数

            for t in range(len(trans_list)):
                if(rand <= trans_list[t]):
                    visit_next=unvisit_list[t]
                    break
                else:
                    continue
            path_mat[ant,j]=visit_next#填路径矩阵
            unvisit_list.remove(visit_next)#更新
            visit=visit_next#更新

#所有蚂蚁的路径表填满之后，算每只蚂蚁的总距离
dis_allant_list=cal_newpath(dis_mat,path_mat)
```



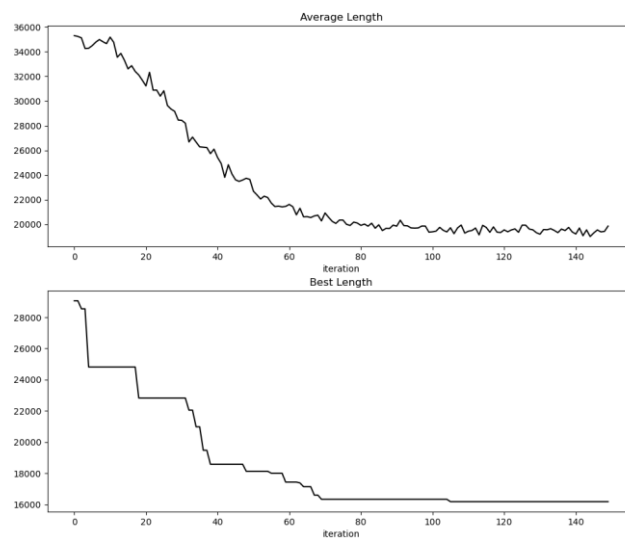
迭代最短路径和距离

```
#每次迭代更新最短距离和最短路径
if count_iter == 0:
    dis_new=min(dis_allant_list)
    path_new=path_mat[dis_allant_list.index(dis_new)].copy()
else:
    if min(dis_allant_list) < dis_new:
        dis_new=min(dis_allant_list)
        path_new=path_mat[dis_allant_list.index(dis_new)].copy()
```

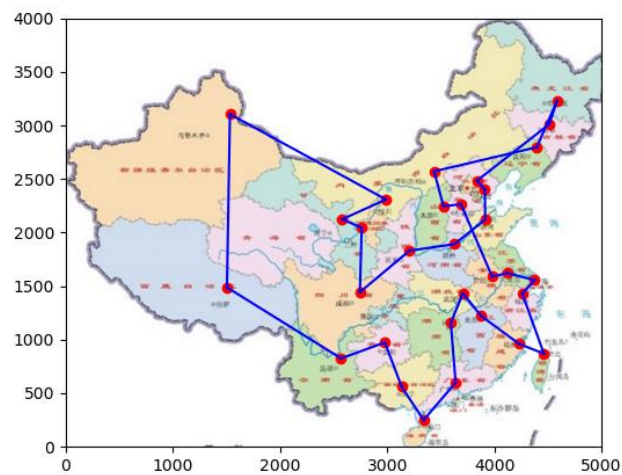
更新信息素矩阵

```
# 更新信息素矩阵
pheromone_change=np.zeros((num_city,num_city))
for i in range(num_ant):
    for j in range(num_city-1):
        pheromone_change[path_mat[i,j]][path_mat[i,j+1]] += Q/dis_mat[path_mat[i,j]][path_mat[i,j+1]]
        pheromone_change[path_mat[i,num_city-1]][path_mat[i,0]] += Q/dis_mat[path_mat[i,num_city-1]][path_mat[i,0]]
    pheromone_mat=(1-info)*pheromone_mat+pheromone_change
count_iter += 1 #迭代计数+1, 进入下一次
```

实验结果及感想



显示结果在地图上:



实验感想: 通过这次实验, 更深刻的了解了蚁群算法, 并将其用代码实现。在实验过程中遇到很多困难, 通过参考老师的资料和网上的一些代码, 终于将这个实验完成, 受益匪浅。

实验六 基于 MNIST 数据集的手写数字识别

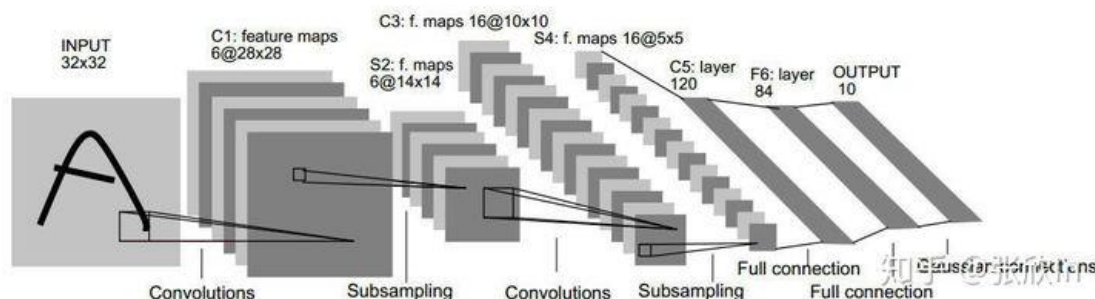
实验目的

利用 pytorch 建立神经网络，掌握神经网络训练的基本操作。更深一步的了解深度学习与神经网络。

实验原理

卷积神经网络：

本次实验是利用卷积神经网络训练识别手写数字图像，基本流程如下图：



数据集：

MNIST 数据集来自美国国家标准与技术研究所, National Institute of Standards and Technology (NIST). 训练集 (training set) 由来自 250 个不同人手写的数字构成, 其中 50% 是高中学生, 50% 来自人口普查局 (the Census Bureau) 的工作人员. 测试集(test set) 也是同样比例的手写数字数据。

框架：

使用pytorch的处理图像视频的 torchvision 工具集直接下载 MNIST 的训练和测试图片, torchvision 包含了一些常用的数据集、模型和转换函数等等, 比如图片分类、语义切分、目标识别、实例分割、关键点检测、视频分类等工具。

算法实现流程

模块一：下载数据集、训练集


```
#下载训练集
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('data', train=True, download=True,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3081,))
                   ])),
    batch_size=BATCH_SIZE, shuffle=True)

#下载测试集
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('data', train=False, transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])),
    batch_size=BATCH_SIZE, shuffle=True)
```

模块二：定义卷积神经网络

```
#定义卷积神经网络
class ConvNet(nn.Module):
    def __init__(self):
        super().__init__()
        # batch*1*28*28 (每次会送入batch个样本, 输入通道数1(黑白图像), 图像分辨率是28x28)
        # 下面的卷积层Conv2d的第一个参数指输入通道数, 第二个参数指输出通道数, 第三个参数指卷积核的大小
        self.conv1 = nn.Conv2d(1, 10, 5) # 输入通道数1, 输出通道数10, 核的大小5
        self.conv2 = nn.Conv2d(10, 20, 3) # 输入通道数10, 输出通道数20, 核的大小3
        # 下面的全连接层Linear的第一个参数指输入通道数, 第二个参数指输出通道数
        self.fc1 = nn.Linear(20*10*10, 500) # 输入通道数是2000, 输出通道数是500
        self.fc2 = nn.Linear(500, 10) # 输入通道数是500, 输出通道数是10, 即10分类

    def forward(self, x):
        in_size = x.size(0) # 在本例中in_size=512, 也就是BATCH_SIZE的值。输入的x可以看成是512*1*28*28的张量。
        out = self.conv1(x) # batch*1*28*28 -> batch*10*24*24 (28x28的图像经过一次核为5x5的卷积, 输出变为24x24)
        out = F.relu(out) # batch*10*24*24 (激活函数ReLU不改变形状)
        out = F.max_pool2d(out, 2, 2) # batch*10*24*24 -> batch*10*12*12 (2*2的池化层会减半)
        out = self.conv2(out) # batch*10*12*12 -> batch*20*10*10 (再卷积一次, 核的大小是3)
        out = F.relu(out) # batch*20*10*10
        out = out.view(in_size, -1) # batch*20*10*10 -> batch*2000 (out的第二维是-1, 说明是自动推算, 本例中第二维是20*10*10)
        out = self.fc1(out) # batch*2000 -> batch*500
        out = F.relu(out) # batch*500
        out = self.fc2(out) # batch*500 -> batch*10
        out = F.log_softmax(out, dim=1) # 计算log(softmax(x))
        return out
```

模块三：训练数据集

```
#训练
def train(model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if (batch_idx+1)%30 == 0:
            print('Train Epoch: {} [{}/{}] ({:.0f}%) \t Loss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

模块四：测试

```

#测试
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item() # 将一批的损失相加
            pred = output.max(1, keepdim=True)[1] # 找到概率最大的下标
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%) \n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))

```

训练环境及实验结果

实验环境配置：

参数	值
CPU	2.6GHz 六核 Intel Core i7
GPU	N/A
Python	3.6.2
Anaconda	2.2.1
Pytorch	1.2.0
Torchvision	0.4.0
操作系统	Windows 10 家庭版

模型参数配置：

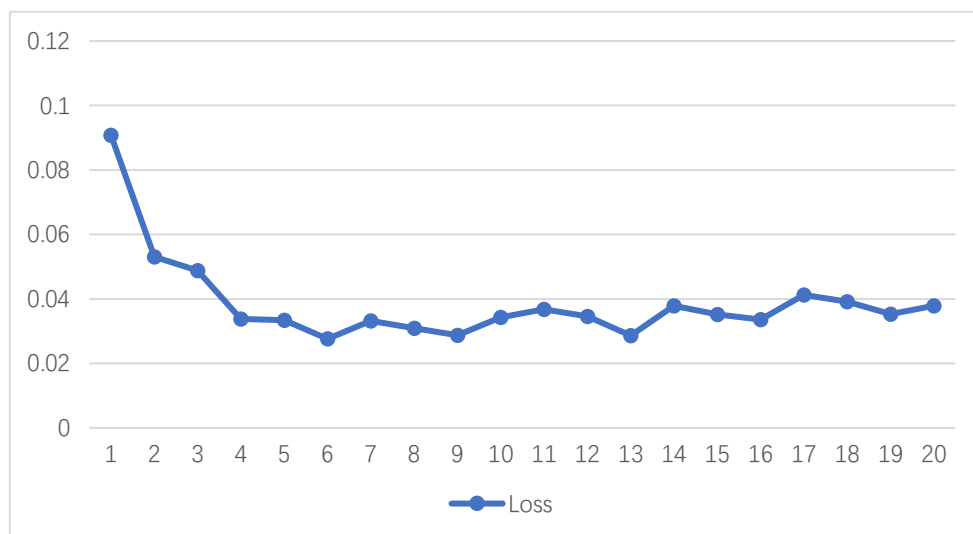
参数名	数值	含义
Conv1	5*5	卷积层 1
Conv2	3*3	卷积层 2
Epoches	20	单次训练迭代次数
Batch_size	512	输入批量大小

Pool	2*2	池化层
------	-----	-----

训练结果：

Loss 随 epoches 变化曲线

Epoches	Loss
1	0.0908
2	0.0531
3	0.0488
4	0.0338
5	0.0334
6	0.0276
7	0.0332
8	0.0309
9	0.0287
10	0.0343
11	0.0368
12	0.0346
13	0.0286
14	0.0379
15	0.0352
16	0.0336
17	0.0412
18	0.0392
19	0.0353
20	0.0379



测试集数据训练结果

Test set: Average loss: 0.0379, Accuracy: 9905/10000 (99%)

手动绘制图像训练结果

手绘了几个数字进行测试



1.jpg



2.jpg



3.jpg

识别结果如下

```
(torch) c:\Users\MSI-Job\Desktop\Coding\Introduction to AI\CNN>python test.py
[[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]]
1
(torch) c:\Users\MSI-Job\Desktop\Coding\Introduction to AI\CNN>python test.py
[[0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]]
2
(torch) c:\Users\MSI-Job\Desktop\Coding\Introduction to AI\CNN>python test.py
[[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]]
3
```

实验感想

通过这次的实验，对 pytorch 的相关操作及基于 pytorch 构建、训练、测试神经网络有了初步的认识。自己尝试了神经网络完整的训练过程，对神经网络的相关特性有了更加清楚的认识，也对之前学习过的卷积神经网络的相关理论知识有了更深入的理解。感谢老师布置了这次实验，使得我能有所收获。

七、实验总结

本学期的人工智能导论实验课是最使我受益匪浅的课程之一。这门课让我实实在在的实现了许多算法。也许这些算法我之前就学习过，但我可能就不会进一步的探索。但有了这门实验课，我真的动手，一行代码一行代码的实现了各个算法，也找到了很多之前没注意到的细节与之前没理解透彻的理论。很感谢任晨珊老师为了提供了这样的机会去将相关算法学习的更加透彻，也会在未来的学习中不断增加自己的知识！