天津大学

# 人工智能基础实验报告


| | |
|---|---|
| 学　院： | 智能与计算学部 |
| 专　业： | 计算机科学与技术 |
| 姓　名： | 陆子毅 |
| 学　号： | 3022206045 |
| 年　级： | 2022 级 |
| 班　级： | 4 班 |

**实验日期：2024 年 06 月 11 日**

# 实验一 产生式动物识别系统

## 一、 实验内容

实现动物识别系统——识别虎、金钱豹、斑马、长颈鹿、鸵鸟、企鹅、信天翁等七种动物的产生式系统。能够根据输入的特征来判断是哪一种动物。

**规则库：**

$r_1$: IF 该动物有毛发 THEN 该动物是哺乳动物
$r_2$: IF 该动物有奶 THEN 该动物是哺乳动物
$r_3$: IF 该动物有羽毛 THEN 该动物是鸟
$r_4$: IF 该动物会飞 AND 会下蛋 THEN 该动物是鸟
$r_5$: IF 该动物吃肉 THEN 该动物是食肉动物
$r_6$: IF 该动物有犬齿 AND 有爪 AND 眼盯前方 THEN 该动物是食肉动物
$r_7$: IF 该动物是哺乳动物 AND 有蹄 THEN 该动物是有蹄类动物
$r_8$: IF 该动物是哺乳动物 AND 是反刍动物 THEN 该动物是有蹄类动物
$r_9$: IF 该动物是哺乳动物 AND 是食肉动物 AND 是黄褐色 AND 身上有暗斑点 THEN 该动物是金钱豹
$r_{10}$: IF 该动物是哺乳动物 AND 是食肉动物 AND 是黄褐色 AND 身上有黑色条纹 THEN 该动物是虎
$r_{11}$: IF 该动物是有蹄类动物 AND 有长脖子 AND 有长腿 AND 身上有暗斑点 THEN 该动物是长颈鹿
$r_{12}$: IF 该动物有蹄类动物 AND 身上有黑色条纹 THEN 该动物是斑马
$r_{13}$: IF 该动物是鸟 AND 有长脖子 AND 有长腿 AND 不会飞 AND 有黑白二色 THEN 该动物是鸵鸟
$r_{14}$: IF 该动物是鸟 AND 会游泳 AND 不会飞 AND 有黑白二色 THEN 该动物是企鹅
$r_{15}$: IF 该动物是鸟 AND 善飞 THEN 该动物是信天翁

设已知初始事实存放在综合数据库中：该动物身上有：暗斑点，长脖子，长腿，奶，蹄，哺乳动物，

有蹄类动物推理机构的工作过程 ：

（1）从规则库中取出 r1，检查其前提是否可与综合数据库中的已知事实匹配。匹配失败则 r1 不能被用于推理。然后取 r2 进行同样的工作。匹配成功则 r2 被执行。

（2）分别用 r3，r4，r5，r6 综合数据库中的已知事实进行匹配，均不成功。r7 匹配成功，执行 r7。

（3）r11 匹配成功，并推出 “该动物是长颈鹿”

## 二、 实验要求

1. 实现产生式动物识别系统；
2. 打印出中间的推理过程以及最后的结果
3. 按要求书写实验报告。

## 三、 代码实现

```
// 产生式动物识别系统

#include <iostream>
#include <cstring>
using namespace std;
string fea[25] = {"", "有毛发", "有奶", "有羽毛", "会飞", "会下蛋", "吃肉", "有犬齿", "有爪", "眼盯前方", "有蹄",
"嚼反刍", "黄褐色", "身上有暗斑点", "身上有黑色条纹", "有长脖子", "有长腿", "不会飞", "会游泳", "有黑白二色", "善飞", "
哺乳动物", "鸟", "食肉动物", "蹄类动物"};

string judge(int inpfea[])
{
    string answer;
    cout << "推理过程：" << endl;
    if (inpfea[1])
    {
        inpfea[21] = 1;
        cout << "特征 1：有毛发" << endl;
        cout << "推理结果：属于哺乳动物" << endl;
    }
    if (inpfea[2])
    {
        inpfea[21] = 1;
        cout << "特征 2：有奶" << endl;
        cout << "推理结果：属于哺乳动物" << endl;
    }
    if (inpfea[3])
    {
        inpfea[22] = 1;
        cout << "特征 3：有羽毛" << endl;
```

```
        cout << "推理结果：属于鸟类" << endl;
    }
    if (inpfea[4] && inpfea[5])
    {
        inpfea[22] = 1;
        cout << "特征 4：会飞" << endl;
        cout << "特征 5：会下蛋" << endl;
        cout << "推理结果：属于鸟类" << endl;
    }
    if (inpfea[6])
    {
        inpfea[23] = 1;
        cout << "特征 6：吃肉" << endl;
        cout << "推理结果：属于食肉动物" << endl;
    }
    if (inpfea[7] && inpfea[8] && inpfea[9])
    {
        inpfea[23] = 1;
        cout << "特征 7：有犬齿" << endl;
        cout << "特征 8：有爪" << endl;
        cout << "特征 9：眼盯前方" << endl;
        cout << "推理结果：属于食肉动物" << endl;
    }
    if (inpfea[21] && inpfea[10])
    {
        inpfea[24] = 1;
        cout << "特征 10：有蹄" << endl;
        cout << "推理结果：属于蹄类动物" << endl;
    }
    if (inpfea[21] && inpfea[11])
    {
        inpfea[24] = 1;
        cout << "特征 11：嚼反刍" << endl;
        cout << "推理结果：属于蹄类动物" << endl;
    }

    if (inpfea[21] && inpfea[23] && inpfea[12] && inpfea[13])
    {
        answer = "金钱豹";
        cout << "特征 12：黄褐色" << endl;
        cout << "特征 13：身上有暗斑点" << endl;
        cout << "推理结果：属于金钱豹" << endl;
    }
    else if (inpfea[21] && inpfea[23] && inpfea[12] && inpfea[14])
    {
        answer = "虎";
        cout << "特征 12：黄褐色" << endl;
        cout << "特征 14：身上有黑色条纹" << endl;
        cout << "推理结果：属于虎" << endl;
    }
    else if (inpfea[24] && inpfea[15] && inpfea[16] && inpfea[13])
    {
        answer = "长颈鹿";
        cout << "特征 15：有长脖子" << endl;
        cout << "特征 16：有长腿" << endl;
        cout << "特征 13：身上有暗斑点" << endl;
        cout << "推理结果：属于长颈鹿" << endl;
    }
    else if (inpfea[24] && inpfea[15] && inpfea[16] && inpfea[14])
    {
        answer = "斑马";
        cout << "特征 15：有长脖子" << endl;
        cout << "特征 16：有长腿" << endl;
        cout << "特征 14：身上有黑色条纹" << endl;
        cout << "推理结果：属于斑马" << endl;
    }
    else if (inpfea[22] && inpfea[15] && inpfea[16] && inpfea[17] && inpfea[19])
    {
        answer = "鸵鸟";
        cout << "特征 15：有长脖子" << endl;
        cout << "特征 16：有长腿" << endl;
        cout << "特征 17：不会飞" << endl;
        cout << "特征 19：会游泳" << endl;
        cout << "推理结果：属于鸵鸟" << endl;
    }
    else if (inpfea[22] && inpfea[18] && inpfea[17] && inpfea[19])
    {
```

```
        answer = "企鹅";
        cout << "特征 18：有黑白二色" << endl;
        cout << "特征 17：不会飞" << endl;
        cout << "特征 19：会游泳" << endl;
        cout << "推理结果：属于企鹅" << endl;
    }
    else if (inpfea[22] && inpfea[20])
    {
        answer = "信天翁";
        cout << "特征 20：善飞" << endl;
        cout << "推理结果：属于信天翁" << endl;
    }
    else
    {
        answer = "error";
        cout << "推理结果：无法识别该动物" << endl;
    }

    return answer;
}

int main()
{
    string s;
    cout << "动物特征如下：\n";
    for (int i = 1; i <= 24; i++)
    {
        cout << i << "." << fea[i] << "\t";
        if (i % 4 == 0)
            cout << endl;
    }
    cout << "------------------------------------------------------------\n 请输入数字选择动物的特征，结尾处用
end：" << endl;

    int inpfea[25] = {0};
    while (cin >> s && s != "end")
    {
        inpfea[stoi(s)] = 1;
    }
    // 输出 inpfea


    string answer;
    answer = judge(inpfea);
    if (answer != "error")
        cout << "success:该动物名称为："
             << answer << endl;
    else
        cout << "error:无法识别该动物\n";

    cout << "程序成功退出！\n";
    return 0;
}
```

## 四、 实验结果

```
o lab1 && "/workspaces/Introduction-to-Artificial-Intelligence/Project/Lab1/"lab1
动物特征如下：
1.有毛发          2.有奶    3.有羽毛          4.会飞
5.会下蛋          6.吃肉    7.有犬齿          8.有爪
9.眼盯前方        10.有蹄   11.嚼反刍         12.黄褐色
13.身上有暗斑点   14.身上有黑色条纹         15.有长脖子       16.有长腿
17.不会飞         18.会游泳         19.有黑白二色     20.善飞
21.哺乳动物       22.鸟     23.食肉动物       24.蹄类动物
------------------------------------------------------------
请输入数字选择动物的特征，结尾处用end:
21 23 12 13 end
推理过程：
特征12：黄褐色
特征13：身上有暗斑点
推理结果：属于金钱豹
success:该动物名称为：金钱豹
程序成功退出！
```

# 实验二 启发式搜索

## 一、 实验内容

### A 算法求解八数码

状态：描述 8 个棋子和空位在棋盘的 9 个方格上的分布情况。其中，任何状态都可以被指定为初始状态。

操作符：产生 4 个行动，即上下左右移动

目标测试：用来检测状态是否能匹配上给定的目标状态。

路径费用函数：每一步的费用为 1，因此整个路径的费用是路径中的步数。

问题描述：给定任意一个初始状态，要求找到一种搜索策略，用尽可能少的步数得到上图的目标状态。

- OPEN表保存所有已生成而未考察的节点
- CLOSED表中记录已访问过的节点。

1. 将起始点加入open表
2. 当open表不为空时：
3. 寻找open表中f值最小的点current
4. 如果current是终止点，则找到结果，程序结束。
5. 否则，open表移出current，对current表中的每一个临近点：
6. 若它不可走或在close表中，略过
7. 若它不在open表中，加入。
8. 若它在open表中，计算g值，若g值更小，替换其父节点为current，更新它的g值。
9. . 若open表为空，则路径不存在。

A 算法特点在于对估价函数 f 的定义上。对于一般的启发式图搜索，总是选择估价函数 f 值最小的节点作为扩展节点。

估价函数：$f(n)=g(n)+h(n)$

- $g(n)$为初始状态到状态 n 是已付出的实际代价；
- $h(n)$是从状态 n 到目标状态的最优路径的估计代价，而搜索的启发式信息主要由$h(n)$决定。

## 二、 实验要求

1. 实现 A 算法；
2. 统计达到目标状态是走的路径长度，并可按要求展示中间结果
3. 按要求书写实验报告。

## 三、 代码实现

```cpp
#include <iostream>
#include <queue>
#include <map>
using namespace std;
struct state
{
    int data[3][3];
    int g = 0, h = 0;
    int i, j; // 表示空格的位置
    bool operator>(const state &x) const
    {
        return g + h >= x.g + x.h;
    }
    string path;
};

map<long long, bool> m; // 标记 open 表中是否存在该情况
```

```cpp
priority_queue<state, vector<state>, greater<state>> q; // 用优先队列实现 open 表

int result[3][3] = {1, 2, 3, 8, 0, 4, 7, 6, 5}; // 题目指定的目标状态
int xx[4] = {1, -1, 0, 0}, yy[4] = {0, 0, 1, -1};

int hn(state x) // 求出从当前状态到目标状态的估计代价，返回 -1 表示该状态存在过
{
    int h = 0;
    long long vis = 0;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            h += (x.data[i][j] != result[i][j]);
            vis = vis * 10 + x.data[i][j];
        }
    }
    if (m[vis])
        return -1; // 标记 vis 中是否存在这个情况，也就是探测到的情况
    else
        m[vis] = 1;
    return h;
}

bool check(int result[3][3], int data[3][3]) // 检查输入是否合法
{
    int cnt = 0;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (data[i][j] != 0)
                cnt++;
        }
    }
    if (cnt != 8)
    {
        printf("输入不合法\n");
        exit(0);
    }
    int vis[10] = {0};
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (data[i][j] < 0 || data[i][j] > 8 || vis[data[i][j]])
            {
                printf("输入不合法\n");
                exit(0);
            }
            vis[data[i][j]] = 1;
        }
    }
    return true;
}

int main()
{
    printf("请输入 9 个数（用空格分开，用 0 代表空格）：\n");
    state a;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            cin >> a.data[i][j];
            if (a.data[i][j] == 0) // 用 0 表示空格
            {
                a.i = i;
                a.j = j;
            }
        }
    }

    if (check(result, a.data))
    {
        printf("输入合法\n");
    } // 检查输入是否合法
```

```
    else
        return 0;

    a.h = hn(a); // 计算估计代价

    q.push(a);

    while (!q.empty())
    {
        state x = q.top();
        q.pop();
        printf("+ - + - + - +\n");
        for (int i = 0; i < 3; i++)
        {
            printf("|");
            for (int j = 0; j < 3; j++)
            {
                printf(" %d |", x.data[i][j]);
            }
            printf("\n");
            printf("+ - + - + - +\n");
        }
        printf("\n");

        if (x.h == 0) // 如果 f（n）为 0，表明到达了目标状态
        {
            printf("空格移动路径为：%s\n", x.path.c_str());
            printf("最少移动次数为%d次\n", x.g + x.h);
            break;
        }
        for (int i = 0; i < 4; i++)
        {
            if (x.i + xx[i] < 0 || x.i + xx[i] > 2 || x.j + yy[i] < 0 || x.j + yy[i] > 2)
                continue;
            state y = x;
            swap(y.data[x.i][x.j], y.data[x.i + xx[i]][x.j + yy[i]]);
            y.g++;        // 每一步移动的代价是 1
            y.h = hn(y); // 计算估计代价
            if (y.h == -1)
                continue; // vis 中已经存在这个情况
            y.i = x.i + xx[i];
            y.j = x.j + yy[i];
            y.path += (i == 0 ? "下 " : (i == 1 ? "上 " : (i == 2 ? "右 " : "左 ")));
            q.push(y);
        }
    }
    return 0;
}
```

# 四、 实验结果

```
@luziyi ➜/workspaces/Introduction-to-Artificial-Intelligence/Project/Lab2 (main) $ cd "/workspaces/Introduction-to-Artificial-Intelligence/Project/Lab2/" && g++ lab2.cpp -o lab2 && "/workspaces/Introduction-to-Artificial-Intelligence/Project/Lab2/"lab2
请输入9个数（用空格分开，用0代表空格）：
2 1 3
7 6 4
0 8 5
```

```
+ - + - + - +
| 1 | 2 | 3 |
+ - + - + - +
| 0 | 8 | 4 |
+ - + - + - +
| 7 | 6 | 5 |
+ - + - + - +

+ - + - + - +
| 1 | 2 | 3 |
+ - + - + - +
| 8 | 0 | 4 |
+ - + - + - +
| 7 | 6 | 5 |
+ - + - + - +

空格移动路径为：右 上 上 右 下 下 左 左 上 右 下 右 上 上 左 左 下 右
最少移动次数为18次
```

# 实验三 LeNet-5 手写数字识别

## 一、 实验内容

LeNet-5 模型是 Yann LeCun 教授 1998 年在论文 Gradient-Based Learning Applied to Document Recognition 中提出的，它是第一个成功应用于数字识别问题的卷积神经网络。在 MNIST 数据集上，LeNet-5 模型可以达到大约 99.2%的正确率。LeNet-5 模型总共有 7 层，下图展示了 LeNet-5 模型的架构：

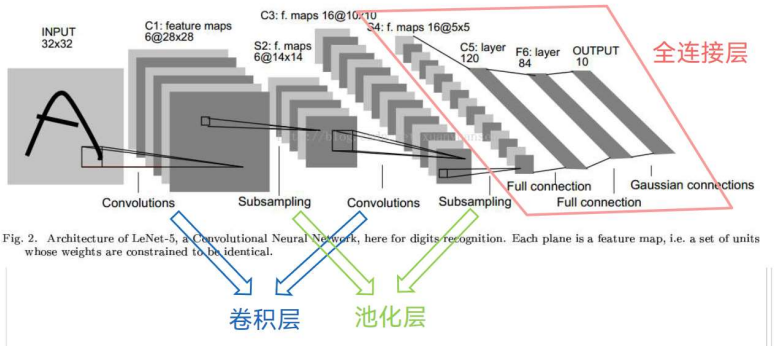



Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

LeNet-5 共有 7 层（不包含输入），每层都包含可训练参数。输入图像大小为 32*32，比 MNIST 数据集的图片要大一些，这么做的原因是希望潜在的明显特征如笔画断点或角能够出现在最高层特征检测子感受野（receptive field）的中心。因此在训练整个网络之前，需要对 28*28 的图像加上 paddings（即周围填充 0）。

C1 层：该层是一个卷积层。使用 6 个大小为 5*5 的卷积核，步长为 1，对输入层进行卷积运算，特征图尺寸为 32-5+1=28，因此产生 6 个大小为 28*28 的特征图。这么做够防止原图像输入的信息掉到卷积核边界之外。

S2 层：该层是一个池化层（pooling，也称为下采样层）。这里采用 max_pool（最大池化），池化的 size 定为 2*2，经池化后得到 6 个 14*14 的特征图，作为下一层神经元的输入。

C3 层：该层仍为一个卷积层，我们选用大小为 5*5 的 16 种不同的卷积核。这里需要注意：C3 中的每个特征图，都是 S2 中的所有 6 个或其中几个特征图进行加权组合得到的。输出为 16 个 10*10 的特征图。

S4 层：该层仍为一个池化层，size 为 2*2，仍采用 max_pool。最后输出 16 个 5*5 的特征图，神经元个数也减少至 16*5*5=400。

F5 层：该层我们继续用 5*5 的卷积核对 S4 层的输出进行卷积，卷积核数量增加至 120。这样 C5 层的输出图片大小为 5-5+1=1。最终输出 120 个 1*1 的特征图。这里实际上是与 S4 全连接了，但仍将其标为卷积层，原因是如果 LeNet-5 的输入图片尺寸变大，其他保持不变，那该层特征图的维数也会大于 1*1。

F6 层：该层与 C5 层全连接，输出 84 张特征图。

输出层：该层与 F6 层全连接，输出长度为 10 的张量，代表所抽取的特征属于哪个类别。（例如[0,0,0,1,0,0,0,0,0,0]的张量，1 在 index=3 的位置，故该张量代表的图片属于第三类）

## 二、 实验要求

- 完成基于 LeNet-5 的手写数字识别系统。
- MINIST 手写数字数据集下载：http://yann.lecun.com/exdb/mnist/
- 提交代码和实验报告。

## 三、 代码实现

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import torch.nn.functional as F

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# 定义名为 LeNet5 的类，该类继承自 nn.Module
class LeNet5(nn.Module):
    def __init__(self, num_classes):
        super(LeNet5, self).__init__()
        # C1 层：该层是一个卷积层。使用 6 个大小为 5*5 的卷积核，步长为 1，对输入层进行卷积运算，特征图尺寸为 32-5+1=28，因此产生 6 个大小为 28*28 的特征图。这么做够防止原图像输入的信息掉到卷积核边界之外。
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=0),    # 卷积
            # 第一个参数 1，表示输入图像的通道数。在这个例子中，输入图像是灰度图像，所以通道数为 1。
            # 第二个参数 6，表示卷积层的输出通道数，也就是卷积核的数量。6 个卷积核。
            # kernel_size=5 定义了卷积核的大小，这里是 5x5 的卷积核。
            # stride=1 定义了卷积核移动的步长，这里每次移动 1 个像素点。
            # padding=0 定义了在输入图像周围填充 0 的层数，这里没有填充。
            nn.BatchNorm2d(6),      # 批归一化
            nn.ReLU(),)



        # S2 层：该层是一个池化层（pooling，也称为下采样层）。这里采用 max_pool（最大池化），池化的 size 定为 2*2，经池化后得到 6 个 14*14 的特征图，作为下一层神经元的输入。
        self.subsampel1 = nn.MaxPool2d(kernel_size = 2, stride = 2)     # 最大池化   使用 max 提取特征




        # C3 层：该层仍为一个卷积层，我们选用大小为 5*5 的 16 种不同的卷积核。这里需要注意：C3 中的每个特征图，都是 S2 中的所有 6 个或其中几个特征图进行加权组合得到的。输出为 16 个 10*10 的特征图。
        self.layer2 = nn.Sequential(
            nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0),
            nn.BatchNorm2d(16),
            nn.ReLU(),)



        # S4 层：该层仍为一个池化层，size 为 2*2，仍采用 max_pool。最后输出 16 个 5*5 的特征图，神经元个数也减少至 16*5*5=400。
        self.subsampel2 = nn.MaxPool2d(kernel_size = 2, stride = 2)


        # F5 层：该层我们继续用 5*5 的卷积核对 S4 层的输出进行卷积，卷积核数量增加至 120。这样 C5 层的输出图片大小为 5-5+1=1。最终输出 120 个 1*1 的特征图。这里实际上是与 S4 全连接了，但仍将其标为卷积层，原因是如果 LeNet-5 的输入图片尺寸变大，其他保持不变，那该层特征图的维数也会大于 1*1。
        self.L1 = nn.Linear(400, 120)
        self.relu = nn.ReLU()

        # F6 层：该层与 C5 层全连接，输出 84 张特征图。
        self.L2 = nn.Linear(120, 84)
        self.relu1 = nn.ReLU()

        # 输出层：该层与 F6 层全连接，输出长度为 10 的张量，代表所抽取的特征属于哪个类别。（例如 [0,0,0,1,0,0,0,0,0,0]的张量，1 在 index=3 的位置，故该张量代表的图片属于第三类）
        self.L3 = nn.Linear(84, num_classes)

    # 前向传播
    def forward(self, x):
        out = self.layer1(x)
        out = self.subsampel1(out)
        out = self.layer2(out)
        out = self.subsampel2(out)

        out = out.reshape(out.size(0), -1) # 将上一步输出的 16 个 5×5 特征图中的 400 个像素展平成一维向量，以便下一步全连接

        # 全连接
        out = self.L1(out)
        out = self.relu(out)
        out = self.L2(out)
```

```
        out = self.relu1(out)
        out = self.L3(out)


        # 找到概率最大的元素的索引
        max_index = torch.argmax(out, dim=1)
        # 创建一个新的张量，其中只有这个索引位置的元素为 1，其他元素都为 0
        out1 = F.one_hot(max_index, num_classes=10)
        # print(out1)
        return out

# 加载训练集
train_dataset = torchvision.datasets.MNIST(root = './data', # 数据集保存路径
                                           train = True,    # 是否为训练集
                                           # 数据预处理
                                           transform = transforms.Compose([
                                                  transforms.Resize((32,32)),
                                                  transforms.ToTensor(),
                                                  transforms.Normalize(mean = (0.1307,),
                                                                       std = (0.3081,))]),
                                           download = True) #是否下载
# transforms.Resize((32,32)) 将每个图像调整为 32x32 像素。
# transforms.ToTensor() 将图像数据转换为 PyTorch 张量。
# transforms.Normalize(mean = (0.1307,), std = (0.3081,)) 对图像数据进行标准化，这里的均值和标准差是根据 MNIST 数据集的特性设定的。
# 从图像的每个通道中减去对应的均值。
# 然后将结果除以对应的标准差。
# 这样处理后，每个通道的数据都会变成均值为 0，标准差为 1 的分布，这有助于神经网络的训练。

# 加载测试集
test_dataset = torchvision.datasets.MNIST(root = './data',
                                          train = False,
                                          transform = transforms.Compose([
                                                  transforms.Resize((32,32)),
                                                  transforms.ToTensor(),
                                                  transforms.Normalize(mean = (0.1325,),
                                                                       std = (0.3105,))]),
                                          download=True)




batch_size = 64
# 定义了一个变量 batch_size，并将其值设置为 128。在机器学习和深度学习中，batch_size 通常用于指定在一次迭代中用于更新模型权重的样本数量。
# 选择合适的 batch_size 是优化模型性能的关键。如果 batch_size 太小，模型可能会在训练过程中遇到噪声，导致权重更新不稳定。如果 batch_size 太大，模型可能会需要更多的内存，并且可能会导致模型在训练过程中过早地收敛到一个局部最优解，而不是全局最优解。

# 加载训练数据
train_loader = torch.utils.data.DataLoader(dataset = train_dataset, batch_size = batch_size, shuffle =
True)
# 加载测试数据
test_loader = torch.utils.data.DataLoader(dataset = test_dataset, batch_size = batch_size, shuffle =
False)

num_classes = 10 # 0-9 数字，共 10 个类别

model = LeNet5(num_classes).to(device)  # 实例化模型，并将其移动到设备上

cost = nn.CrossEntropyLoss()    # 定义损失函数 交叉熵损失 度量模型的预测概率分布与真实分布之间的差距

learning_rate = 0.001 # 学习率
# 如果学习率设置得过大，可能会导致模型在训练过程中震荡不定，难以收敛。如果学习率设置得过小，模型训练的速度可能会非常慢，需要更多的时间才能收敛。
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate) # 定义优化器 用于更新权重

# torch.optim.Adam 是一个实现了 Adam 优化算法的类。Adam 优化算法是一种自适应学习率的优化算法，它结合了 RMSProp 算法和 Momentum 算法的优点。
# model.parameters()是一个函数，它返回模型中所有的可训练参数。这些参数是需要优化的对象，因此我们将它们传递给优化器。
# lr=learning_rate 设置了优化器的学习率。学习率是一个超参数，它决定了模型参数更新的速度。如果学习率太高，模型可能会在最优解附近震荡而无法收敛；如果学习率太低，模型收敛的速度可能会非常慢。
# 创建一个 Adam 优化器，用于管理和更新模型的参数，以便在训练过程中改进模型的性能。

total_step = len(train_loader)

# 设置一共训练几轮（epoch）
```

```
num_epochs = 10



# 外部循环用于遍历轮次
for epoch in range(num_epochs):
    # 内部循环用于遍历每轮中的所有批次
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device) # 将加载的图像和标签移动到设备上
        labels = labels.to(device)

        # 前向传播
        outputs = model(images)   # 通过模型进行前向传播，得到模型的预测结果 outputs
        loss = cost(outputs, labels)    # 计算模型预测与真实标签之间的损失

        # 反向传播和优化
        optimizer.zero_grad()   # 清零梯度，以便在下一次反向传播中不累积之前的梯度
        loss.backward()     # 进行反向传播，计算梯度
        optimizer.step()    # 根据梯度更新（优化）模型参数

        # 定期输出训练信息
        # 在每经过一定数量的批次后，输出当前训练轮次、总周轮数、当前批次、总批次数和损失值
        if (i+1) % 300 == 0:
            print('训练轮次 [{:2d}/{:2d}], 批次 [{}/{}], 损失值: {:.4f}'.format(epoch+1, num_epochs, i+1,
total_step, loss.item()))




# 测试数据集
with torch.no_grad():   # 指示 PyTorch 在接下来的代码块中不要计算梯度
    # 初始化计数器
    correct = 0     # 正确分类的样本数
    total = 0       # 总样本数

    # 遍历测试数据集的每个批次
    for images, labels in test_loader:
        # 将加载的图像和标签移动到设备上
        images = images.to(device)
        labels = labels.to(device)

        # 模型预测
        outputs = model(images)

        # 计算准确率
        # 从模型输出中获取每个样本预测的类别
        _, predicted = torch.max(outputs.data, 1)
        # 累积总样本数
        total += labels.size(0)
        # 累积正确分类的样本数
        correct += (predicted == labels).sum().item()

    # 输出准确率，正确的 / 总的
    print('测试总数: {} '.format(total))
    print('测试准确率: {} %'.format(100 * correct / total))
```

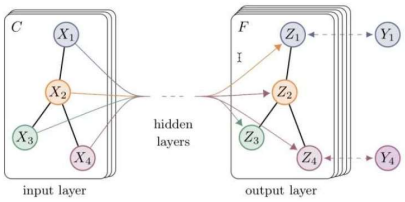
# 四、 实验结果

```
训练轮次 [ 2/10], 批次 [300/938], 损失值: 0.0504
训练轮次 [ 2/10], 批次 [600/938], 损失值: 0.1021
训练轮次 [ 2/10], 批次 [900/938], 损失值: 0.0243
训练轮次 [ 3/10], 批次 [300/938], 损失值: 0.0125
训练轮次 [ 3/10], 批次 [600/938], 损失值: 0.0737
训练轮次 [ 3/10], 批次 [900/938], 损失值: 0.0030
训练轮次 [ 4/10], 批次 [300/938], 损失值: 0.1125
训练轮次 [ 4/10], 批次 [600/938], 损失值: 0.1504
训练轮次 [ 4/10], 批次 [900/938], 损失值: 0.0179
训练轮次 [ 5/10], 批次 [300/938], 损失值: 0.0689
训练轮次 [ 5/10], 批次 [600/938], 损失值: 0.0016
训练轮次 [ 5/10], 批次 [900/938], 损失值: 0.0221
训练轮次 [ 6/10], 批次 [300/938], 损失值: 0.0112
训练轮次 [ 6/10], 批次 [600/938], 损失值: 0.0057
训练轮次 [ 6/10], 批次 [900/938], 损失值: 0.0008
训练轮次 [ 7/10], 批次 [300/938], 损失值: 0.0038
训练轮次 [ 7/10], 批次 [600/938], 损失值: 0.0185
训练轮次 [ 7/10], 批次 [900/938], 损失值: 0.0138
训练轮次 [ 8/10], 批次 [300/938], 损失值: 0.0054
训练轮次 [ 8/10], 批次 [600/938], 损失值: 0.0098
训练轮次 [ 8/10], 批次 [900/938], 损失值: 0.0505
训练轮次 [ 9/10], 批次 [300/938], 损失值: 0.0253
训练轮次 [ 9/10], 批次 [600/938], 损失值: 0.0002
训练轮次 [ 9/10], 批次 [900/938], 损失值: 0.0704
训练轮次 [10/10], 批次 [300/938], 损失值: 0.0052
训练轮次 [10/10], 批次 [600/938], 损失值: 0.0012
训练轮次 [10/10], 批次 [900/938], 损失值: 0.0009
测试总数: 10000
测试准确率: 97.99 %
```

# 实验四 GCN

## 一、 实验内容

图卷积神经网络：结合图中的节点属性和链接结构，通过图卷积神经网络生成低维的节点表示。该节点表示可以用于各种下游任务中，如节点分类、链接预测、图分类等。




(a) Graph Convolutional Network

符号说明：

- 图$G=(V,E)$，其中V表示有N个节点的节点集，E表示有M条边的边集。
- $X \in \mathbb{R}^{N\times f}$表示的是节点的属性矩阵，f 表示的是每个节点属性的维度
- $A=\{a_{ij}\} \in \mathbb{R}^{N\times N}$表示的是图的邻接矩阵 (可以通过E构建邻接矩阵A)其中$a_{ij}$表示的是节点$v_i$和$v_j$的链接关系，如果节点$v_i$和$v_j$存在边那么$a_{ij}=1$否则$a_{ij}=0$
- $D \in \mathbb{R}^{N\times N}$表示的是对角的度矩阵，其中$d_{ii}=\sum_{j\in V} a_{ij}$

**GCN 的基本组成——图卷积层**

$$H^{l}=\sigma\left(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}\,H^{l-1}\,W^{l}\right)$$

其中，$H^{l-1}$，$H^{l}$分别代表上一层卷积层输出的节点表示和当前层卷积层输出的节点表示；$\tilde{A}=A+I$表示带有自环的邻接矩阵，$\tilde{D}$为$\tilde{A}$对应的度矩阵，$W^{l}$代表当前层中的权重矩阵(也是 GCN 需要学习到的参数)；$\sigma()$表示的是激活函数，一般选取ReLU()

**下游任务——节点分类**

针对节点分类的下游任务，需要根据所学到的节点表示对该节点的类别进行预测。一般情况下，采用softmax()进行节点类别的预测，即：

$$\hat{Y}=softmax(H^{(L)})$$

其中，$\hat{Y} \in \mathbb{R}^{N\times C}$表示预测的节点标签，C 表示节点的类别总数

**性能评估——交叉熵**

我们采用交叉熵函数来评估节点类别预测的准确率，即：

$$L=-Y log\hat{Y}$$

其中，$Y,\hat{Y}$分别表示真实标签和预测标签。

## 二、 实验要求

1. 搭建图卷积网络 GCN；
2. 基于 Cora 数据集对 GCN 的性能进行预测
3. 按要求书写实验报告。

## 三、 代码实现

```python
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import torch.nn.functional as F

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# 定义名为 LeNet5 的类，该类继承自 nn.Module
class LeNet5(nn.Module):
    def __init__(self, num_classes):
```

```
        super(LeNet5, self).__init__()
        # C1层：该层是一个卷积层。使用 6 个大小为 5*5 的卷积核，步长为 1，对输入层进行卷积运算，特征图尺寸为 32-5+1=28，因此产生 6 个大小为 28*28 的特征图。这么做够防止原图像输入的信息掉到卷积核边界之外。
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=0),    # 卷积
            # 第一个参数 1，表示输入图像的通道数。在这个例子中，输入图像是灰度图像，所以通道数为 1。
            # 第二个参数 6，表示卷积层的输出通道数，也就是卷积核的数量。6 个卷积核。
            # kernel_size=5 定义了卷积核的大小，这里是 5x5 的卷积核。
            # stride=1 定义了卷积核移动的步长，这里每次移动 1 个像素点。
            # padding=0 定义了在输入图像周围填充 0 的层数，这里没有填充。
            nn.BatchNorm2d(6),      # 批归一化
            nn.ReLU(),)



        # S2层：该层是一个池化层（pooling，也称为下采样层）。这里采用 max_pool（最大池化），池化的 size 定为 2*2，经池化后得到 6 个 14*14 的特征图，作为下一层神经元的输入。
        self.subsampel1 = nn.MaxPool2d(kernel_size = 2, stride = 2)     # 最大池化   使用 max 提取特征




        # C3层：该层仍为一个卷积层，我们选用大小为 5*5 的 16 种不同的卷积核。这里需要注意：C3 中的每个特征图，都是 S2 中的所有 6 个或其中几个特征图进行加权组合得到的。输出为 16 个 10*10 的特征图。
        self.layer2 = nn.Sequential(
            nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0),
            nn.BatchNorm2d(16),
            nn.ReLU(),)



        # S4层：该层仍为一个池化层，size 为 2*2，仍采用 max_pool。最后输出 16 个 5*5 的特征图，神经元个数也减少至 16*5*5=400。
        self.subsampel2 = nn.MaxPool2d(kernel_size = 2, stride = 2)


        # F5层：该层我们继续用 5*5 的卷积核对 S4 层的输出进行卷积，卷积核数量增加至 120。这样 C5 层的输出图片大小为 5-5+1=1。最终输出 120 个 1*1 的特征图。这里实际上是与 S4 全连接了，但仍将其标为卷积层，原因是如果 LeNet-5 的输入图片尺寸变大，其他保持不变，那该层特征图的维数也会大于 1*1。
        self.L1 = nn.Linear(400, 120)
        self.relu = nn.ReLU()

        # F6层：该层与 C5 层全连接，输出 84 张特征图。
        self.L2 = nn.Linear(120, 84)
        self.relu1 = nn.ReLU()

        # 输出层：该层与 F6 层全连接，输出长度为 10 的张量，代表所抽取的特征属于哪个类别。（例如 [0,0,0,1,0,0,0,0,0,0]的张量，1 在 index=3 的位置，故该张量代表的图片属于第三类）
        self.L3 = nn.Linear(84, num_classes)

    # 前向传播
    def forward(self, x):
        out = self.layer1(x)
        out = self.subsampel1(out)
        out = self.layer2(out)
        out = self.subsampel2(out)

        out = out.reshape(out.size(0), -1) # 将上一步输出的 16 个 5x5 特征图中的 400 个像素展平成一维向量，以便下一步全连接

        # 全连接
        out = self.L1(out)
        out = self.relu(out)
        out = self.L2(out)
        out = self.relu1(out)
        out = self.L3(out)


        # 找到概率最大的元素的索引
        max_index = torch.argmax(out, dim=1)
        # 创建一个新的张量，其中只有这个索引位置的元素为 1，其他元素都为 0
        out1 = F.one_hot(max_index, num_classes=10)
        # print(out1)
        return out

# 加载训练集
train_dataset = torchvision.datasets.MNIST(root = './data', # 数据集保存路径
```

```
                                        train = True,    # 是否为训练集
                                        # 数据预处理
                                        transform = transforms.Compose([
                                               transforms.Resize((32,32)),
                                               transforms.ToTensor(),
                                               transforms.Normalize(mean = (0.1307,),
                                                                    std = (0.3081,))]),
                                        download = True) #是否下载
# transforms.Resize((32,32)) 将每个图像调整为 32x32 像素。
# transforms.ToTensor() 将图像数据转换为 PyTorch 张量。
# transforms.Normalize(mean = (0.1307,), std = (0.3081,)) 对图像数据进行标准化，这里的均值和标准差是根据 MNIST 数据集的特性设定的。
# 从图像的每个通道中减去对应的均值。
# 然后将结果除以对应的标准差。
# 这样处理后，每个通道的数据都会变成均值为 0，标准差为 1 的分布，这有助于神经网络的训练。

# 加载测试集
test_dataset = torchvision.datasets.MNIST(root = './data',
                                        train = False,
                                        transform = transforms.Compose([
                                               transforms.Resize((32,32)),
                                               transforms.ToTensor(),
                                               transforms.Normalize(mean = (0.1325,),
                                                                    std = (0.3105,))]),
                                        download=True)




batch_size = 64
# 定义了一个变量 batch_size，并将其值设置为 128。在机器学习和深度学习中，batch_size 通常用于指定在一次迭代中用于更新模型权重的样本数量。
# 选择合适的 batch_size 是优化模型性能的关键。如果 batch_size 太小，模型可能会在训练过程中遇到噪声，导致权重更新不稳定。如果 batch_size 太大，模型可能会需要更多的内存，并且可能会导致模型在训练过程中过早地收敛到一个局部最优解，而不是全局最优解。

# 加载训练数据
train_loader = torch.utils.data.DataLoader(dataset = train_dataset, batch_size = batch_size, shuffle =
True)
# 加载测试数据
test_loader = torch.utils.data.DataLoader(dataset = test_dataset, batch_size = batch_size, shuffle =
False)

num_classes = 10 # 0-9 数字，共 10 个类别

model = LeNet5(num_classes).to(device)  # 实例化模型，并将其移动到设备上

cost = nn.CrossEntropyLoss()    # 定义损失函数 交叉熵损失 度量模型的预测概率分布与真实分布之间的差距

learning_rate = 0.001 # 学习率
# 如果学习率设置得过大，可能会导致模型在训练过程中震荡不定，难以收敛。如果学习率设置得过小，模型训练的速度可能会非常慢，需要更多的时间才能收敛。
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate) # 定义优化器 用于更新权重

# torch.optim.Adam 是一个实现了 Adam 优化算法的类。Adam 优化算法是一种自适应学习率的优化算法，它结合了 RMSProp 算法和 Momentum 算法的优点。
# model.parameters()是一个函数，它返回模型中所有的可训练参数。这些参数是需要优化的对象，因此我们将它们传递给优化器。
# lr=learning_rate 设置了优化器的学习率。学习率是一个超参数，它决定了模型参数更新的速度。如果学习率太高，模型可能会在最优解附近震荡而无法收敛；如果学习率太低，模型收敛的速度可能会非常慢。
# 创建一个 Adam 优化器，用于管理和更新模型的参数，以便在训练过程中改进模型的性能。

total_step = len(train_loader)

# 设置一共训练几轮（epoch）
num_epochs = 10



# 外部循环用于遍历轮次
for epoch in range(num_epochs):
    # 内部循环用于遍历每轮中的所有批次
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device) # 将加载的图像和标签移动到设备上
        labels = labels.to(device)

        # 前向传播
        outputs = model(images)    # 通过模型进行前向传播，得到模型的预测结果 outputs
```

```
        loss = cost(outputs, labels)    # 计算模型预测与真实标签之间的损失

        # 反向传播和优化
        optimizer.zero_grad()    # 清零梯度，以便在下一次反向传播中不累积之前的梯度
        loss.backward()     # 进行反向传播，计算梯度
        optimizer.step()    # 根据梯度更新（优化）模型参数

        # 定期输出训练信息
        # 在每经过一定数量的批次后，输出当前训练轮次、总周轮数、当前批次、总批次数和损失值
        if (i+1) % 300 == 0:
            print('训练轮次 [{:2d}/{:2d}], 批次 [{}/{}], 损失值: {:.4f}'.format(epoch+1, num_epochs, i+1,
total_step, loss.item()))




# 测试数据集
with torch.no_grad():   # 指示 PyTorch 在接下来的代码块中不要计算梯度
    # 初始化计数器
    correct = 0     # 正确分类的样本数
    total = 0       # 总样本数

    # 遍历测试数据集的每个批次
    for images, labels in test_loader:
        # 将加载的图像和标签移动到设备上
        images = images.to(device)
        labels = labels.to(device)

        # 模型预测
        outputs = model(images)

        # 计算准确率
        # 从模型输出中获取每个样本预测的类别
        _, predicted = torch.max(outputs.data, 1)
        # 累积总样本数
        total += labels.size(0)
        # 累积正确分类的样本数
        correct += (predicted == labels).sum().item()

    # 输出准确率，正确的 / 总的
    print('测试总数: {} '.format(total))
    print('测试准确率: {} %'.format(100 * correct / total))
```

# 四、 实验结果

```
Artificial-Intelligence/Project/Lab4/lab4.py"
Cora dataset:
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.x
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.tx
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.allx
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.y
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.ty
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.ally
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.graph
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.test.index
Processing...
Done!
Data(x=[2708, 1433], edge_index=[2, 10556], y=[2708], train_mask=[2708], val_mask=[2708], test_mask=[2708]) 1433 7
Epoch 000 loss 2.1690
Epoch 001 loss 0.9734
Epoch 002 loss 0.5915
Epoch 003 loss 0.4042
Epoch 004 loss 0.2783
 Epoch 190 loss 0.0015
 Epoch 191 loss 0.0067
 Epoch 192 loss 0.0017
 Epoch 193 loss 0.0015
 Epoch 194 loss 0.0017
 Epoch 195 loss 0.0007
 Epoch 196 loss 0.0012
 Epoch 197 loss 0.0007
 Epoch 198 loss 0.0010
 Epoch 199 loss 0.0008
 GCN Accuracy: 0.7400
```