**图片包含 画

描述已自动生成**

**人工智能基础实验报告**

徽标, 公司名称

描述已自动生成

|  |  |
| --- | --- |
| **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** |

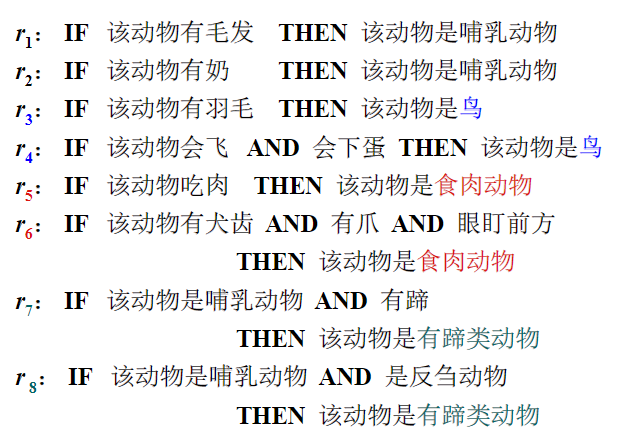
**实验日期：2024年06月11日**

**实验一 产生式动物识别系统**

1. **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：**

 文本

描述已自动生成

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”

1. **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。

1. **代码实现**
2. #include <iostream>
3. #include <cstring>
4. using namespace std;
5. string fea[25] = {"", "有毛发", "有奶", "有羽毛", "会飞", "会下蛋", "吃肉", "有犬齿", "有爪", "眼町前方", "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"};
6. string judge(int inpfea[])
7. {
8. string answer;
9. cout << "推理过程：" << endl;
10. if (inpfea[1])
11. {
12. inpfea[21] = 1;
13. cout << "特征1：有毛发" << endl;
14. cout << "推理结果：属于哺乳动物" << endl;
15. }
16. if (inpfea[2])
17. {
18. inpfea[21] = 1;
19. cout << "特征2：有奶" << endl;
20. cout << "推理结果：属于哺乳动物" << endl;
21. }
22. if (inpfea[3])
23. {
24. inpfea[22] = 1;
25. cout << "特征3：有羽毛" << endl;
26. cout << "推理结果：属于鸟类" << endl;
27. }
28. if (inpfea[4] && inpfea[5])
29. {
30. inpfea[22] = 1;
31. cout << "特征4：会飞" << endl;
32. cout << "特征5：会下蛋" << endl;
33. cout << "推理结果：属于鸟类" << endl;
34. }
35. if (inpfea[6])
36. {
37. inpfea[23] = 1;
38. cout << "特征6：吃肉" << endl;
39. cout << "推理结果：属于食肉动物" << endl;
40. }
41. if (inpfea[7] && inpfea[8] && inpfea[9])
42. {
43. inpfea[23] = 1;
44. cout << "特征7：有犬齿" << endl;
45. cout << "特征8：有爪" << endl;
46. cout << "特征9：眼町前方" << endl;
47. cout << "推理结果：属于食肉动物" << endl;
48. }
49. if (inpfea[21] && inpfea[10])
50. {
51. inpfea[24] = 1;
52. cout << "特征10：有蹄" << endl;
53. cout << "推理结果：属于蹄类动物" << endl;
54. }
55. if (inpfea[21] && inpfea[11])
56. {
57. inpfea[24] = 1;
58. cout << "特征11：嚼反刍" << endl;
59. cout << "推理结果：属于蹄类动物" << endl;
60. }
61. if (inpfea[21] && inpfea[23] && inpfea[12] && inpfea[13])
62. {
63. answer = "金钱豹";
64. cout << "特征12：黄褐色" << endl;
65. cout << "特征13：身上有暗斑点" << endl;
66. cout << "推理结果：属于金钱豹" << endl;
67. }
68. else if (inpfea[21] && inpfea[23] && inpfea[12] && inpfea[14])
69. {
70. answer = "虎";
71. cout << "特征12：黄褐色" << endl;
72. cout << "特征14：身上有黑色条纹" << endl;
73. cout << "推理结果：属于虎" << endl;
74. }
75. else if (inpfea[24] && inpfea[15] && inpfea[16] && inpfea[13])
76. {
77. answer = "长颈鹿";
78. cout << "特征15：有长脖子" << endl;
79. cout << "特征16：有长腿" << endl;
80. cout << "特征13：身上有暗斑点" << endl;
81. cout << "推理结果：属于长颈鹿" << endl;
82. }
83. else if (inpfea[24] && inpfea[15] && inpfea[16] && inpfea[14])
84. {
85. answer = "斑马";
86. cout << "特征15：有长脖子" << endl;
87. cout << "特征16：有长腿" << endl;
88. cout << "特征14：身上有黑色条纹" << endl;
89. cout << "推理结果：属于斑马" << endl;
90. }
91. else if (inpfea[22] && inpfea[15] && inpfea[16] && inpfea[17] && inpfea[19])
92. {
93. answer = "鸵鸟";
94. cout << "特征15：有长脖子" << endl;
95. cout << "特征16：有长腿" << endl;
96. cout << "特征17：不会飞" << endl;
97. cout << "特征19：会游泳" << endl;
98. cout << "推理结果：属于鸵鸟" << endl;
99. }
100. else if (inpfea[22] && inpfea[18] && inpfea[17] && inpfea[19])
101. {
102. answer = "企鹅";
103. cout << "特征18：有黑白二色" << endl;
104. cout << "特征17：不会飞" << endl;
105. cout << "特征19：会游泳" << endl;
106. cout << "推理结果：属于企鹅" << endl;
107. }
108. else if (inpfea[22] && inpfea[20])
109. {
110. answer = "信天翁";
111. cout << "特征20：善飞" << endl;
112. cout << "推理结果：属于信天翁" << endl;
113. }
114. else
115. {
116. answer = "error";
117. cout << "推理结果：无法识别该动物" << endl;
118. }
119. return answer;
120. }
121. int main()
122. {
123. string s;
124. cout << "动物特征如下：\n";
125. for (int i = 1; i <= 24; i++)
126. {
127. cout << i << "." << fea[i] << "\t";
128. if (i % 4 == 0)
129. cout << endl;
130. }
131. cout << "----------------------------------------------------------\n请输入数字选择动物的特征，结尾处用end：" << endl;
132. int inpfea[25] = {0};
133. while (cin >> s && s != "end")
134. {
135. inpfea[stoi(s)] = 1;
136. }
137. *// 输出inpfea*
138. string answer;
139. answer = judge(inpfea);
140. if (answer != "error")
141. cout << "success:该动物名称为："
142. << answer << endl;
143. else
144. cout << "error:无法识别该动物\n";
145. cout << "程序成功退出!\n";
146. return 0;
147. }
148. **实验结果**

文本

描述已自动生成

**实验二 启发式搜索**

1. **实验内容**

**A算法求解八数码**

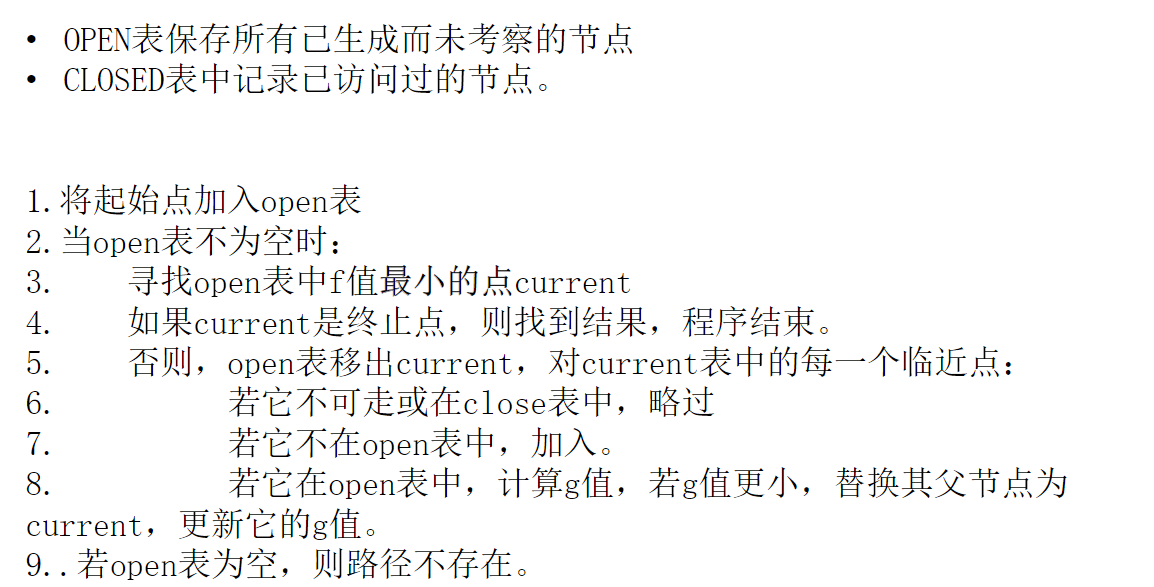
状态：描述8个棋子和空位在棋盘的9个方格上的分布情况。其中，任何状态都可以被指定为初始状态。

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。



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：

* 为初始状态到状态n是已付出的实际代价；
* 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决定。

1. **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。

1. **代码实现**
2. #include <iostream>
3. #include <queue>
4. #include <map>
5. using namespace std;
6. struct state
7. {
8. int data[3][3];
9. int g = 0, h = 0;
10. int i, j; *// 表示空格的位置*
11. bool operator>(const state &x) const
12. {
13. return g + h >= x.g + x.h;
14. }
15. string path;
16. };
17. map<long long, bool> m; *// 标记open表中是否存在该情况*
18. priority\_queue<state, vector<state>, greater<state>> q; *// 用优先队列实现open表*
19. int result[3][3] = {1, 2, 3, 8, 0, 4, 7, 6, 5}; *// 题目指定的目标状态*
20. int xx[4] = {1, -1, 0, 0}, yy[4] = {0, 0, 1, -1};
21. 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*
22. {
23. int h = 0;
24. long long vis = 0;
25. for (int i = 0; i < 3; i++)
26. {
27. for (int j = 0; j < 3; j++)
28. {
29. h += (x.data[i][j] != result[i][j]);
30. vis = vis \* 10 + x.data[i][j];
31. }
32. }
33. if (m[vis])
34. return -1; *// 标记vis中是否存在这个情况，也就是探测到的情况*
35. else
36. m[vis] = 1;
37. return h;
38. }
39. bool check(int result[3][3], int data[3][3]) *// 检查输入是否合法*
40. {
41. int cnt = 0;
42. for (int i = 0; i < 3; i++)
43. {
44. for (int j = 0; j < 3; j++)
45. {
46. if (data[i][j] != 0)
47. cnt++;
48. }
49. }
50. if (cnt != 8)
51. {
52. printf("输入不合法\n");
53. exit(0);
54. }
55. int vis[10] = {0};
56. for (int i = 0; i < 3; i++)
57. {
58. for (int j = 0; j < 3; j++)
59. {
60. if (data[i][j] < 0 || data[i][j] > 8 || vis[data[i][j]])
61. {
62. printf("输入不合法\n");
63. exit(0);
64. }
65. vis[data[i][j]] = 1;
66. }
67. }
68. return true;
69. }
70. int main()
71. {
72. printf("请输入9个数（用空格分开，用0代表空格）：\n");
73. state a;
74. for (int i = 0; i < 3; i++)
75. {
76. for (int j = 0; j < 3; j++)
77. {
78. cin >> a.data[i][j];
79. if (a.data[i][j] == 0) *// 用0表示空格*
80. {
81. a.i = i;
82. a.j = j;
83. }
84. }
85. }
86. if (check(result, a.data))
87. {
88. printf("输入合法\n");
89. } *// 检查输入是否合法*
90. else
91. return 0;
92. a.h = hn(a); *// 计算估计代价*
93. q.push(a);
94. while (!q.empty())
95. {
96. state x = q.top();
97. q.pop();
98. printf("+ - + - + - +\n");
99. for (int i = 0; i < 3; i++)
100. {
101. printf("|");
102. for (int j = 0; j < 3; j++)
103. {
104. printf(" %d |", x.data[i][j]);
105. }
106. printf("\n");
107. printf("+ - + - + - +\n");
108. }
109. printf("\n");
110. if (x.h == 0) *// 如果f（n）为0，表明到达了目标状态*
111. {
112. printf("空格移动路径为：%s\n", x.path.c\_str());
113. printf("最少移动次数为%d次\n", x.g + x.h);
114. break;
115. }
116. for (int i = 0; i < 4; i++)
117. {
118. if (x.i + xx[i] < 0 || x.i + xx[i] > 2 || x.j + yy[i] < 0 || x.j + yy[i] > 2)
119. continue;
120. state y = x;
121. swap(y.data[x.i][x.j], y.data[x.i + xx[i]][x.j + yy[i]]);
122. y.g++;       *// 每一步移动的代价是1*
123. y.h = hn(y); *// 计算估计代价*
124. if (y.h == -1)
125. continue; *// vis中已经存在这个情况*
126. y.i = x.i + xx[i];
127. y.j = x.j + yy[i];
128. y.path += (i == 0 ? "下 " : (i == 1 ? "上 " : (i == 2 ? "右 " : "左 ")));
129. q.push(y);
130. }
131. }
132. return 0;
133. }
134. **实验结果**

图片包含 图形用户界面

描述已自动生成

图形用户界面

低可信度描述已自动生成

**实验三 LeNet-5 手写数字识别**

1. **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：

图示

描述已自动生成

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）

1. **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。

1. **代码实现**
2. import torch
3. import torch.nn as nn
4. import torchvision
5. import torchvision.transforms as transforms
6. import torch.nn.functional as F
7. device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')
8. *# 定义名为 LeNet5 的类，该类继承自 nn.Module*
9. class LeNet5(nn.Module):
10. def \_\_init\_\_(self, num\_classes):
11. super(LeNet5, self).\_\_init\_\_()
12. *# 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。*
13. self.layer1 = nn.Sequential(
14. nn.Conv2d(1, 6, kernel\_size=5, stride=1, padding=0),    *# 卷积*
15. *# 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。*
16. *# 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。*
17. *# kernel\_size=5定义了卷积核的大小，这里是5x5的卷积核。*
18. *# stride=1定义了卷积核移动的步长，这里每次移动1个像素点。*
19. *# padding=0定义了在输入图像周围填充0的层数，这里没有填充。*
20. nn.BatchNorm2d(6),      *# 批归一化*
21. nn.ReLU(),)


25. *# 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。*
26. self.subsampel1 = nn.MaxPool2d(kernel\_size = 2, stride = 2)     *# 最大池化  使用max提取特征*



31. *# 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。*
32. self.layer2 = nn.Sequential(
33. nn.Conv2d(6, 16, kernel\_size=5, stride=1, padding=0),
34. nn.BatchNorm2d(16),
35. nn.ReLU(),)


39. *# 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。*
40. self.subsampel2 = nn.MaxPool2d(kernel\_size = 2, stride = 2)

43. *# 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。*
44. self.L1 = nn.Linear(400, 120)
45. self.relu = nn.ReLU()
47. *# F6层：该层与C5层全连接，输出84张特征图。*
48. self.L2 = nn.Linear(120, 84)
49. self.relu1 = nn.ReLU()
51. *# 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）*
52. self.L3 = nn.Linear(84, num\_classes)
54. *# 前向传播*
55. def forward(self, x):
56. out = self.layer1(x)
57. out = self.subsampel1(out)
58. out = self.layer2(out)
59. out = self.subsampel2(out)
61. 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*
63. *# 全连接*
64. out = self.L1(out)
65. out = self.relu(out)
66. out = self.L2(out)
67. out = self.relu1(out)
68. out = self.L3(out)

71. *# 找到概率最大的元素的索引*
72. max\_index = torch.argmax(out, dim=1)
73. *# 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*
74. out1 = F.one\_hot(max\_index, num\_classes=10)
75. *# print(out1)*
76. return out
77. *# 加载训练集*
78. train\_dataset = torchvision.datasets.MNIST(root = './data', *# 数据集保存路径*
79. train = True,    *# 是否为训练集*
80. *# 数据预处理*
81. transform = transforms.Compose([
82. transforms.Resize((32,32)),
83. transforms.ToTensor(),
84. transforms.Normalize(mean = (0.1307,),
85. std = (0.3081,))]),
86. download = True) *#是否下载*
87. *# transforms.Resize((32,32)) 将每个图像调整为32x32像素。*
88. *# transforms.ToTensor() 将图像数据转换为PyTorch张量。*
89. *# 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。*
90. *# 从图像的每个通道中减去对应的均值。*
91. *# 然后将结果除以对应的标准差。*
92. *# 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。*
94. *# 加载测试集*
95. test\_dataset = torchvision.datasets.MNIST(root = './data',
96. train = False,
97. transform = transforms.Compose([
98. transforms.Resize((32,32)),
99. transforms.ToTensor(),
100. transforms.Normalize(mean = (0.1325,),
101. std = (0.3105,))]),
102. download=True)
103. batch\_size = 64
104. *# 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。*
105. *# 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。*
106. *# 加载训练数据*
107. train\_loader = torch.utils.data.DataLoader(dataset = train\_dataset, batch\_size = batch\_size, shuffle = True)
108. *# 加载测试数据*
109. test\_loader = torch.utils.data.DataLoader(dataset = test\_dataset, batch\_size = batch\_size, shuffle = False)
110. num\_classes = 10 *# 0-9数字，共10个类别*
111. model = LeNet5(num\_classes).to(device)  *# 实例化模型，并将其移动到设备上*
112. 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*
113. learning\_rate = 0.001 *# 学习率*
114. *# 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。*
115. optimizer = torch.optim.Adam(model.parameters(), lr=learning\_rate) *# 定义优化器 用于更新权重*
116. *# 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。*
117. *# 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。*
118. *# 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。*
119. *# 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。*
120. total\_step = len(train\_loader)
121. *# 设置一共训练几轮（epoch）*
122. num\_epochs = 10
123. *# 外部循环用于遍历轮次*
124. for epoch in range(num\_epochs):
125. *# 内部循环用于遍历每轮中的所有批次*
126. for i, (images, labels) in enumerate(train\_loader):
127. images = images.to(device) *# 将加载的图像和标签移动到设备上*
128. labels = labels.to(device)
130. *# 前向传播*
131. outputs = model(images)   *# 通过模型进行前向传播，得到模型的预测结果 outputs*
132. loss = cost(outputs, labels)    *# 计算模型预测与真实标签之间的损失*
134. *# 反向传播和优化*
135. 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*
136. loss.backward()     *# 进行反向传播，计算梯度*
137. optimizer.step()    *# 根据梯度更新（优化）模型参数*
139. *# 定期输出训练信息*
140. *# 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*
141. if (i+1) % 300 == 0:
142. print('训练轮次 [{:2d}/{:2d}], 批次 [{}/{}], 损失值: {:.4f}'.format(epoch+1, num\_epochs, i+1, total\_step, loss.item()))
143. *# 测试数据集*
144. with torch.no\_grad():   *# 指示 PyTorch 在接下来的代码块中不要计算梯度*
145. *# 初始化计数器*
146. correct = 0     *# 正确分类的样本数*
147. total = 0       *# 总样本数*
149. *# 遍历测试数据集的每个批次*
150. for images, labels in test\_loader:
151. *# 将加载的图像和标签移动到设备上*
152. images = images.to(device)
153. labels = labels.to(device)
155. *# 模型预测*
156. outputs = model(images)
158. *# 计算准确率*
159. *# 从模型输出中获取每个样本预测的类别*
160. \_, predicted = torch.max(outputs.data, 1)
161. *# 累积总样本数*
162. total += labels.size(0)
163. *# 累积正确分类的样本数*
164. correct += (predicted == labels).sum().item()
166. *# 输出准确率，正确的 / 总的*
167. print('测试总数: {} '.format(total))
168. print('测试准确率: {} %'.format(100 \* correct / total))
170. **实验结果**

图形用户界面, 文本

描述已自动生成

**实验四 GCN**

1. **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。

图示

描述已自动生成

符号说明：

* + 图，其中表示有个节点的节点集，表示有条边的边集。
  + 表示的是节点的属性矩阵， 表示的是每个节点属性的维度
  + 表示的是图的邻接矩阵 (可以通过构建邻接矩阵)其中表示的是节点和的链接关系，如果节点和存在边那么否则
  + 表示的是对角的度矩阵，其中

**GCN的基本组成——图卷积层**

其中，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；表示带有自环的邻接矩阵，为对应的度矩阵，代表当前层中的权重矩阵(也是GCN需要学习到的参数)；表示的是激活函数，一般选取

**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，采用进行节点类别的预测，即：

其中*，*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：

其中，分别表示真实标签和预测标签。

1. **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。

1. **代码实现**
2. import torch
3. from torch\_geometric.datasets import Planetoid
4. import torch.nn.functional as F
5. from torch\_geometric.nn import GCNConv
6. class GCN(torch.nn.Module): *# 定义GCN模型*
7. def \_\_init\_\_(self, num\_node\_features, num\_classes): *# 初始化函数*
8. super(GCN, self).\_\_init\_\_() *# 调用父类的初始化函数*
9. hidden\_channels = 32 *# 隐藏层特征数为32*
10. *# 隐藏层特征数是指在神经网络中隐藏层的神经元数量。隐藏层特征数的选择对神经网络的性能和表达能力有很大影响。隐藏层特征数的主要作用是决定神经网络的复杂度和表示能力。较少的隐藏层特征数可能导致神经网络无法捕捉到输入数据中的复杂模式和关系，从而导致欠拟合。而较多的隐藏层特征数可能导致神经网络过度拟合训练数据，无法泛化到新的未见过的数据。*
11. self.conv1 = GCNConv(num\_node\_features, hidden\_channels) *# 输入特征数为num\_node\_features，输出特征数为hidden\_channels*
12. self.conv2 = GCNConv(hidden\_channels, num\_classes) *# 输入特征数为hidden\_channels，输出特征数为num\_classes*
13. self.norm = torch.nn.BatchNorm1d(hidden\_channels) *# 一维批标准化层，输入特征数为hidden\_channels*
14. def forward(self, data): *# 前向传播函数*
15. x, edge\_index = data.x, data.edge\_index *# 获取输入特征x和边索引edge\_index*
16. x = self.conv1(x, edge\_index) *# 第一层GCN 通过GCNConv层对输入特征x和边索引edge\_index进行卷积操作*
17. x = self.norm(x) *# 批标准化 通过BatchNorm1d层对输出特征x进行批标准化操作*
18. x = F.relu(x) *# ReLU激活函数 通过ReLU激活函数对输出特征x进行非线性变换*
19. x = F.dropout(x, training=self.training) *# dropout层 通过Dropout层对输出特征x进行随机失活操作*
20. x = self.conv2(x, edge\_index) *# 第二层GCN*
21. return x
22. def load\_data(name): *#加载数据*
23. dataset = Planetoid(root='./' + name + '/', name=name)
24. \_device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')
25. data = dataset[0].to(\_device)
26. num\_node\_features = dataset.num\_node\_features
27. return data, num\_node\_features, dataset.num\_classes
28. *#定义训练函数*
29. def train(model, data, device):  *# 模型，数据，设备*
30. optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight\_decay=1e-4)  *# Adam优化器 学习率0.01  权重衰减1e-4 根据定义的优化算法和损失函数，自动调整模型的参数，以最小化损失函数的值。在深度学习中，优化器的目标是通过反向传播算法来计算梯度，并使用梯度下降的方法来更新模型的参数。model.parameters()表示要优化的模型的参数，lr表示学习率，即每次更新参数时的步长。另外，weight\_decay参数用于控制L2正则化项的权重衰减。*
31. loss\_function = torch.nn.CrossEntropyLoss().to(device)  *# 交叉熵损失函数计算损失值 将softmax函数和负对数似然损失（negative log likelihood loss）结合在一起。在计算损失时，它首先对模型的原始输出（也就是logits）应用softmax函数，将输出转换为概率分布，然后计算真实标签和预测概率分布之间的负对数似然损失*
32. model.train() *# 训练模式*
33. for epoch in range(200): *# 迭代200次*
34. out = model(data)  *# 前向传播*
35. optimizer.zero\_grad()  *# 梯度清零*
36. loss = loss\_function(out[data.train\_mask], data.y[data.train\_mask])  *#  计算损失值*
37. loss.backward()  *# 反向传播*
38. optimizer.step()  *# 更新参数*
39. print('Epoch {:03d} loss {:.4f}'.format(epoch, loss.item()))
40. *# 测试函数*
41. def test(model, data):
42. model.eval()  *# 测试模式*
43. \_, pred = model(data).max(dim=1) *# 使用模型对数据进行预测，并返回预测结果中的最大值。具体来说，model(data)表示将数据data输入模型进行预测，返回的结果是一个张量。然后，.max(dim=1)表示在结果张量的第一个维度上取最大值，并返回最大值和对应的索引。在这里，我们使用下划线\_来表示我们不关心最大值的具体值，只关心对应的索引值，所以将最大值赋值给了pred变量*
44. correct = int(pred[data.test\_mask].eq(data.y[data.test\_mask]).sum().item())  *#获取预测正确的数量*
45. acc = correct / int(data.test\_mask.sum())
46. print('GCN Accuracy: {:.4f}'.format(acc))
47. def main():
48. print('Cora' + ' dataset:')
49. data, num\_node\_features, num\_classes = load\_data('Cora')
50. print(data, num\_node\_features, num\_classes)
51. \_device = 'cpu'
52. device = torch.device(\_device)
53. model = GCN(num\_node\_features, num\_classes).to('cpu')
54. train(model, data, device)
55. test(model, data)
56. if \_\_name\_\_ == '\_\_main\_\_':
57. main()
59. **实验结果**

文本

描述已自动生成

文本

描述已自动生成