

目录

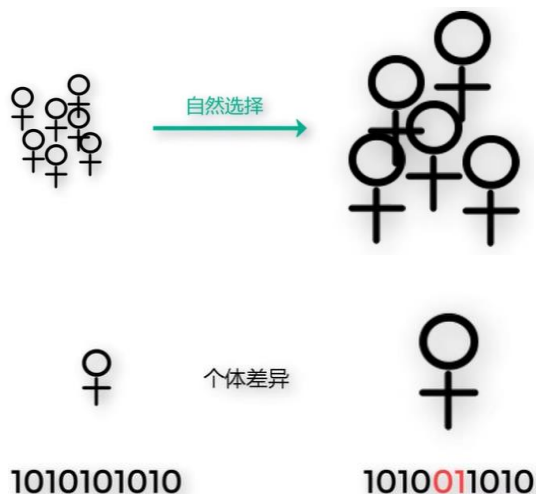
1 遗传算法的学习与实践	3
1.1 遗传算法基本思想	3
1.2 遗传算法中的基本概念	5
1.3 遗传算法的基本流程	5
1.4 遗传算法解决的问题	6
1.5 旅行商问题（TSP）	6
1.6 旅行商问题（TSP）的代码分析	6
1.6.1 config.py 文件代码	7
1.6.2 ga.py 文件代码	8
1.6.3 main.py 文件代码	12
1.6.4 运行结果	13
2 CNN 的学习与实践	14
2.1 画 CNN 网络矩阵图和流程图	14
2.2 用 pytorch 尝试实现以上 CNN 网络	16
2.3 利用以上网络训练和测试 cifar10 数据集	17
2.4 部分训练过程以及最后的结果曲线图	21
3 针对原始 CNN 网络的改进	23
3.1 加入残差结构	23
3.2 尝试调节部分超参数以及加入 dropout	24
3.3 尝试将遗传算法整合到 CNN 网络中	26
4 针对 GA-CNN 网络的改进	29

4.1 对照组在 cifar10 数据集上的效果	29
4.2 实验组在 cifar10 数据集上的效果	30
4.3 二者在 cifar10 数据集上的精度与耗用时间对比	31
4.4 二者在 STL10 数据集上的精度对比	33
4.5 尝试选用不同优化器与不同 ratio 在 cifar10 上的对比	33
5 目标检测网络 retinanet 的学习	34
5.1 retinanet 解决的问题	34
5.2 retinanet 原理	34
5.2.1 RetinaNet 网络结构	34
5.2.2 正负样本划分	36
5.2.3 Facol Loss	37
5.3 retinanet 优劣	39
6 retinanet 项目	39
6.1 目前所构建的 Retinanet 项目使用流程	40
6.1.1 数据集需要转换成的格式说明	40
6.1.2 训练	42
6.1.3 测试	42
7 动态调度网络	43
7.1 动态调度网络的思想	43
7.2 目前动态调度网络项目使用说明	43
7.3 目前的动态调度网络项目训练效果	44

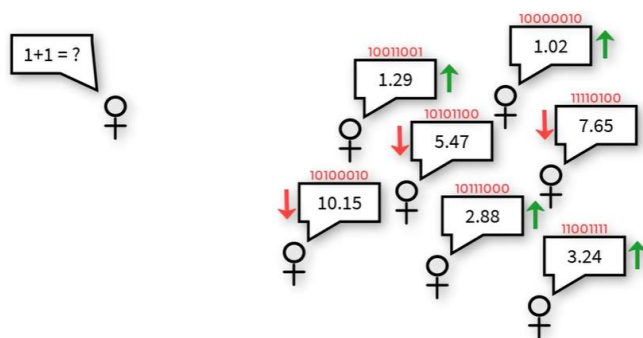
1 遗传算法的学习与实践

1.1 遗传算法基本思想

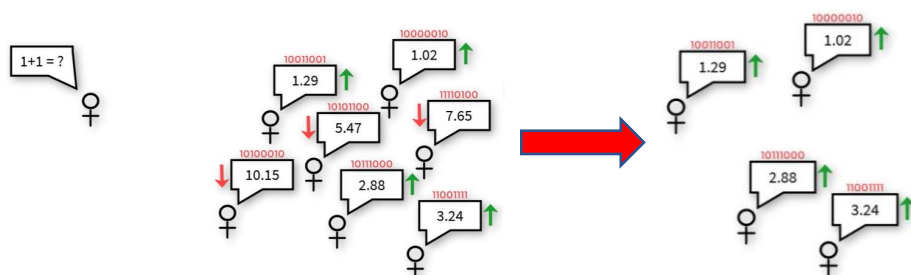
遗传算法主要用于模拟自然界中的种群繁衍的行为，经过自然选择，自然界中的种群一代比一代更加适应环境。



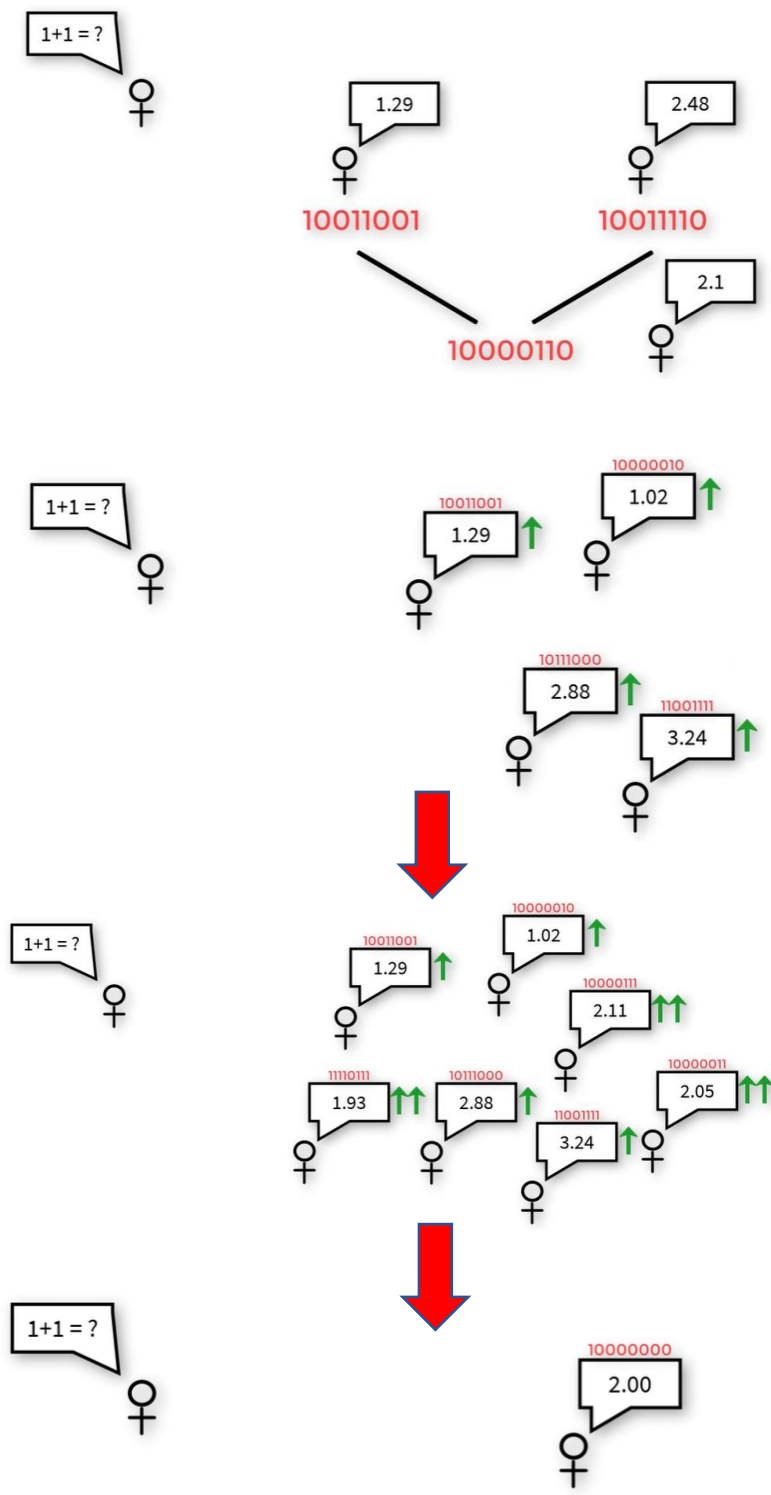
而种群中的个体差异就是基因的差异，遗传算法的思想就是模拟基因的概念，然后将待解决问题的答案进行编码：



而待解决问题的答案（可行解）可能开始时有多个，将每个答案进行编码，在这些答案中既有好的答案也有差的答案，分别对应种群中优秀的个体和差的个体，而好的答案更加接近于最终结果。遗传算法就是将全部的答案（可行解）进行评价，保留好的而剔除差的，相当于自然界中自然选择的过程：



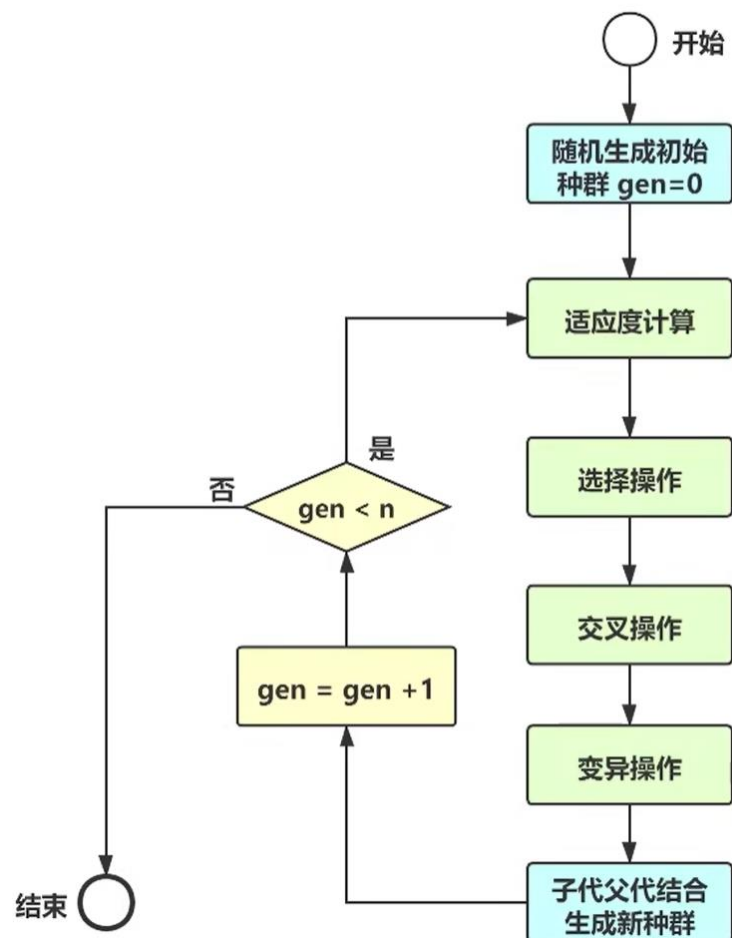
在选择的过程中，编码可以去模拟物种繁衍时基因交叉、变异的过程，从而能够去寻找更加合适的答案，不断迭代，最终能够得到相对最优的解即为最终的答案。



1.2 遗传算法中的基本概念

- ①适应度：衡量个体（可行解）的优劣程度
- ②编码：将可行解抽象成基因的过程
- ③解码：将基因还原成可行解的过程

1.3 遗传算法的基本流程



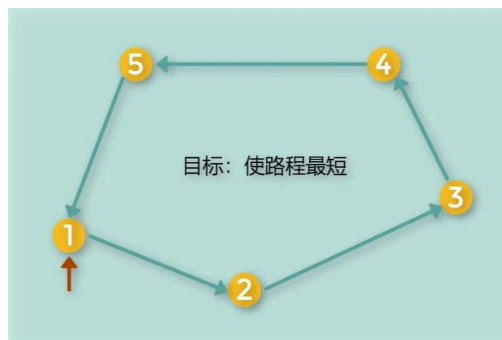
首先，随机生成初始种群并且计算适应度，然后根据适应度对种群中的个体进行选择，优胜劣汰，随后就像种群繁殖一样，将个体的基因两两交叉、以一定概率基因还会发生变异，从而去产生新的个体，然后将新产生的个体和老的个体聚合作为下一代新种群，从而不断地进行迭代循环，直到满足我们设定的迭代停止条件，遗传算法结束，最终筛选出相对最优的个体。

1.4 遗传算法解决的问题

遗传算法实际上就像物种的进化一样，是一个不断尝试的过程，所以遗传算法通常用于解决无法得出最优解的 NP 难问题，例如旅行商问题（TSP）。

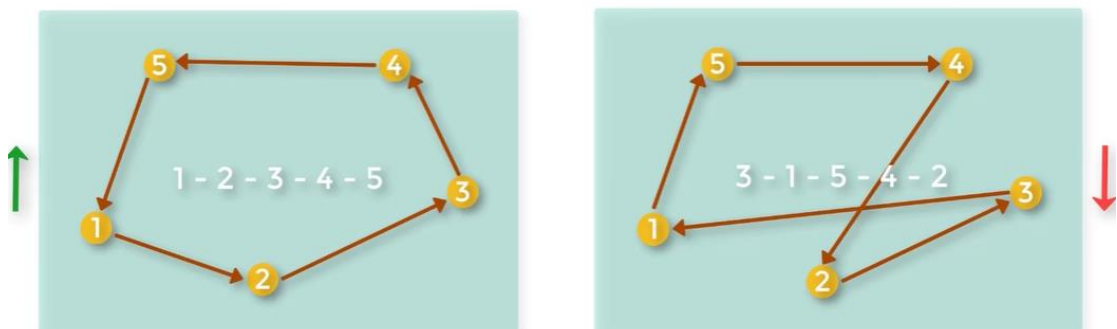
1.5 旅行商问题（TSP）

假设地图上有 5 个城市，一位旅行商从其中的一个城市（假设是城市 1）出发，经过其他的各个城市各一次，最终回到出发的初始城市（城市 1）。我们需要找到一条最优路径，使得旅行商走的路程最短。



而遗传算法有两个关键点：其一是如何将答案编码成基因，其二是如何定义个体的适应度。

在旅行商问题（TSP）中，将旅行路线编码成基因，将路线的长度即路程作为适应度（路程越短则适应度越大，个体越优秀，越可能在选择的过程中被保留下来）。



1.6 旅行商问题（TSP）的代码分析

参考代码：<https://github.com/zifeiyu0531/ga-tsp>

代码包含三个文件：config.py、ga.py、main.py。config.py 文件中包含一些参数的配置（城市数量、坐标维度、个体数、迭代轮数、变异概率），ga.py 文件中是遗传算法的实现，main.py 文件中是程序执行的入口。

1.6.1 config.py 文件代码

```
1.  # -*- coding: utf-8 -*-
2.  import argparse
3.
4.  parser = argparse.ArgumentParser(description='Configuration file')
5.  arg_lists = []
6.
7.
8.  def add_argument_group(name):
9.      arg = parser.add_argument_group(name)
10.     arg_lists.append(arg)
11.     return arg
12.
13.
14.  # Data
15.  data_arg = add_argument_group('Data')
16.  data_arg.add_argument('--city_num', type=int, default=15, help='city num') # 城市数量
17.  data_arg.add_argument('--pos_dimension', type=int, default=2, help='city num') # 坐标维度
18.  data_arg.add_argument('--individual_num', type=int, default=60, help='individual num') # 个体数
19.  data_arg.add_argument('--gen_num', type=int, default=400, help='generation num') # 迭代轮数
20.  data_arg.add_argument('--mutate_prob', type=float, default=0.25, help='probability of mutate') # 变异概率
21.
22.
23.  def get_config():
24.      config, unparsed = parser.parse_known_args()
25.      return config
26.
27.
28.  def print_config():
29.      config = get_config()
30.      print('\n')
31.      print('Data Config:')
32.      print('* city num:', config.city_num)
33.      print('* individual num:', config.individual_num)
34.      print('* generation num:', config.gen_num)
35.      print('* probability of mutate:', config.cross_prob)
```

在该文件中主要定义了包括城市数量、坐标维度、个体数、迭代轮数、变异概率在内的参数配置，以便后续使用。

1.6.2 ga.py 文件代码

```
1. import config as conf
2. import random
3.
4. city_dist_mat = None
5. config = conf.get_config()
6. # 各项参数
7. gene_len = config.city_num
8. individual_num = config.individual_num
9. gen_num = config.gen_num
10. mutate_prob = config.mutate_prob
11.
12.
13. def copy_list(old_arr: [int]):
14.     new_arr = []
15.     for element in old_arr:
16.         new_arr.append(element)
17.     return new_arr
18.
19.
20. # 个体类
21. class Individual:
22.     def __init__(self, genes=None):
23.         # 随机生成序列
24.         if genes is None:
25.             genes = [i for i in range(gene_len)]
26.             random.shuffle(genes)
27.             self.genes = genes
28.             self.fitness = self.evaluate_fitness()
29.
30.     def evaluate_fitness(self):
31.         # 计算个体适应度
32.         fitness = 0.0
33.         for i in range(gene_len - 1):
34.             # 起始城市和目标城市
35.             from_idx = self.genes[i]
36.             to_idx = self.genes[i + 1]
37.             fitness += city_dist_mat[from_idx, to_idx]
38.         # 连接首尾
39.         fitness += city_dist_mat[self.genes[-1], self.genes[0]]
40.         return fitness
41.
42.
```



```

43. class Ga:
44.     def __init__(self, input_):
45.         global city_dist_mat
46.         city_dist_mat = input_
47.         self.best = None # 每一代的最佳个体
48.         self.individual_list = [] # 每一代的个体列表
49.         self.result_list = [] # 每一代对应的解
50.         self.fitness_list = [] # 每一代对应的适应度
51.
52.     def cross(self):
53.         new_gen = []
54.         random.shuffle(self.individual_list)
55.         for i in range(0, individual_num - 1, 2):
56.             # 父代基因
57.             genes1 = copy_list(self.individual_list[i].genes)
58.             genes2 = copy_list(self.individual_list[i + 1].genes)
59.             index1 = random.randint(0, gene_len - 2)
60.             index2 = random.randint(index1, gene_len - 1)
61.             pos1_recorder = {value: idx for idx, value in enumerate(genes1)}
62.             pos2_recorder = {value: idx for idx, value in enumerate(genes2)}
63.             # 交叉
64.             for j in range(index1, index2):
65.                 value1, value2 = genes1[j], genes2[j]
66.                 pos1, pos2 = pos1_recorder[value2], pos2_recorder[value1]
67.                 genes1[j], genes1[pos1] = genes1[pos1], genes1[j]
68.                 genes2[j], genes2[pos2] = genes2[pos2], genes2[j]
69.                 pos1_recorder[value1], pos1_recorder[value2] = pos1, j
70.                 pos2_recorder[value1], pos2_recorder[value2] = j, pos2
71.             new_gen.append(Individual(genes1))
72.             new_gen.append(Individual(genes2))
73.         return new_gen
74.
75.     def mutate(self, new_gen):
76.         for individual in new_gen:
77.             if random.random() < mutate_prob:
78.                 # 翻转切片
79.                 old_genes = copy_list(individual.genes)
80.                 index1 = random.randint(0, gene_len - 2)
81.                 index2 = random.randint(index1, gene_len - 1)
82.                 genes_mutate = old_genes[index1:index2]
83.                 genes_mutate.reverse()
84.                 individual.genes = old_genes[:index1] + genes_mutate + old_genes[index2:]
85.             # 两代合并
86.             self.individual_list += new_gen

```

```

87.
88.     def select(self):
89.         # 锦标赛
90.         group_num = 10 # 小组数
91.         group_size = 10 # 每小组人数
92.         group_winner = individual_num // group_num # 每小组获胜人数
93.         winners = [] # 锦标赛结果
94.         for i in range(group_num):
95.             group = []
96.             for j in range(group_size):
97.                 # 随机组成小组
98.                 player = random.choice(self.individual_list)
99.                 player = Individual(player.genes)
100.                group.append(player)
101.                group = Ga.rank(group)
102.                # 取出获胜者
103.                winners += group[:group_winner]
104.            self.individual_list = winners
105.
106.     @staticmethod
107.     def rank(group):
108.         # 冒泡排序
109.         for i in range(1, len(group)):
110.             for j in range(0, len(group) - i):
111.                 if group[j].fitness > group[j + 1].fitness:
112.                     group[j], group[j + 1] = group[j + 1], group[j]
113.         return group
114.
115.     def next_gen(self):
116.         # 交叉
117.         new_gen = self.cross()
118.         # 变异
119.         self.mutate(new_gen)
120.         # 选择
121.         self.select()
122.         # 获得这一代的结果
123.         for individual in self.individual_list:
124.             if individual.fitness < self.best.fitness:
125.                 self.best = individual
126.
127.     def train(self):
128.         # 初代种群
129.         self.individual_list = [Individual() for _ in range(individual_num)]
130.         self.best = self.individual_list[0]

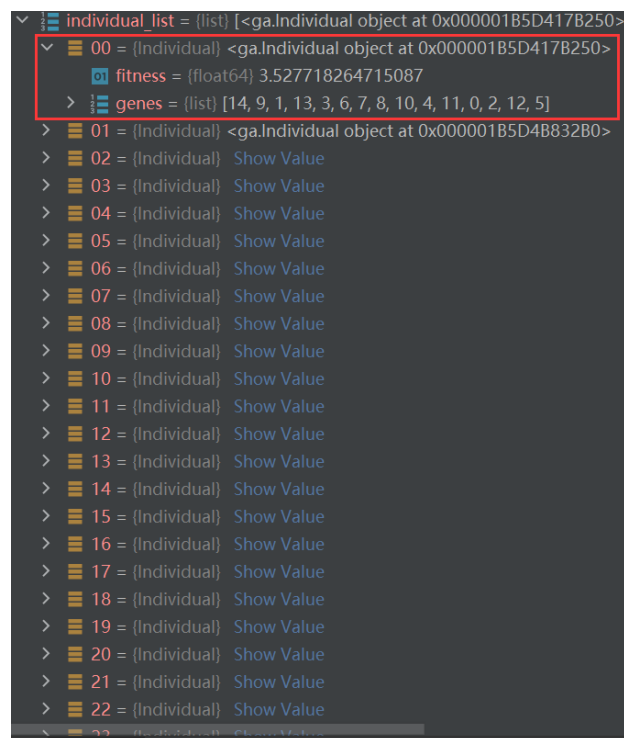
```

```

131.     # 迭代
132.     for i in range(gen_num):
133.         self.next_gen()
134.         # 连接首尾
135.         result = copy_list(self.best.genes)
136.         result.append(result[0])
137.         self.result_list.append(result)
138.         self.fitness_list.append(self.best.fitness)
139.     return self.result_list, self.fitness_list

```

首先生成初代种群，在 config.py 文件中设定种群中个体数为 60，利用循环创建 60 个个体类 Individual 的对象，并且将这 60 个初代种群的个体保存到 individual_list 数组中，同时对每个个体利用 evaluate_fitness() 函数去计算其适应度。初代种群的 individual_list 数组如下图所示：（每个元素为一个个体，每个个体都有基因和适应度两个属性）



然后进行 400 次迭代（该参数在 config.py 中事先设定），在每次迭代中，首先调用 next_gen() 函数进行交叉、变异、选择的操作，并且获得这一代最优的个体。交叉是在种群中采用循环方式两两选择个体，在两个个体的基因上选择一段切片序列，然后将两个个体的切片序列进行交叉互换得到新的个体，将新的个体 append 到 new_gen 数组中；变异是当随机数小于实现设定的变异率后进行翻转切片的操作；选择的方式采用锦标赛选择：设置 10 个小组，每个小组个体为 10，每个小组的获胜人数为种群个体数（60）整除小组数（10），即十进六，将锦标赛结果保存到 winners 数组中，源代码采用一个双层循环来进行具体的实现。经过交叉、变异、选择之后，获得这一代的结果（在种群的所有个体中进行遍历，选择适应度/路程最小的一个个体作为最优的个体 best）。400 次迭代结束后得到最终的结果。

1.6.3 main.py 文件代码

```
1. import numpy as np
2. import config as conf
3. from ga import Ga
4. import matplotlib.pyplot as plt
5.
6. config = conf.get_config()
7.
8.
9. def build_dist_mat(input_list):
10.     n = config.city_num
11.     dist_mat = np.zeros([n, n])
12.     for i in range(n):
13.         for j in range(i + 1, n):
14.             d = input_list[i, :] - input_list[j, :]
15.             # 计算点积
16.             dist_mat[i, j] = np.dot(d, d)
17.             dist_mat[j, i] = dist_mat[i, j]
18.     return dist_mat
19.
20.
21. # 城市坐标
22. city_pos_list = np.random.rand(config.city_num, config.pos_dimension)
23. # 城市距离矩阵
24. city_dist_mat = build_dist_mat(city_pos_list)
25.
26. print(city_pos_list)
27. print(city_dist_mat)
28.
29. # 遗传算法运行
30. ga = Ga(city_dist_mat)
31. result_list, fitness_list = ga.train()
32. result = result_list[-1]
33. result_pos_list = city_pos_list[result, :]
34.
35. # 绘图
36. # 解决中文显示问题
37. plt.rcParams['font.sans-serif'] = ['KaiTi'] # 指定默认字体
38. plt.rcParams['axes.unicode_minus'] = False # 解决保存图像是负号'-'显示为方块的问题
39.
40. fig = plt.figure()
41. plt.plot(result_pos_list[:, 0], result_pos_list[:, 1], 'o-r')
42. plt.title(u"路线")
```

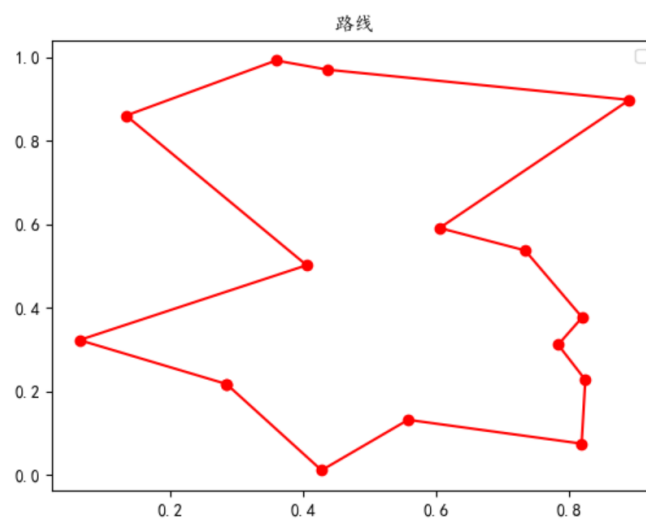
```

43. plt.legend()
44. fig.show()
45.
46. fig = plt.figure()
47. plt.plot(fitness_list)
48. plt.title(u"适应度曲线")
49. plt.legend()
50. fig.show()

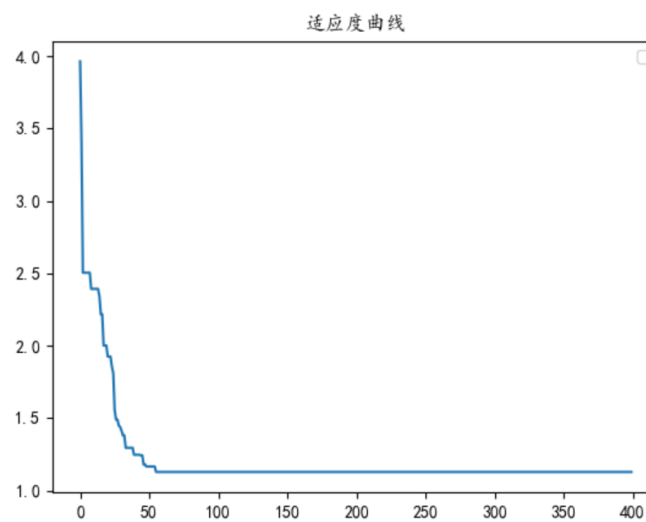
```

1.6.4 运行结果

最终的最优路线：



适应度曲线：



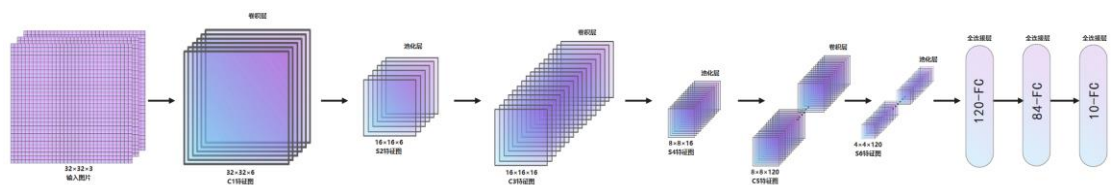
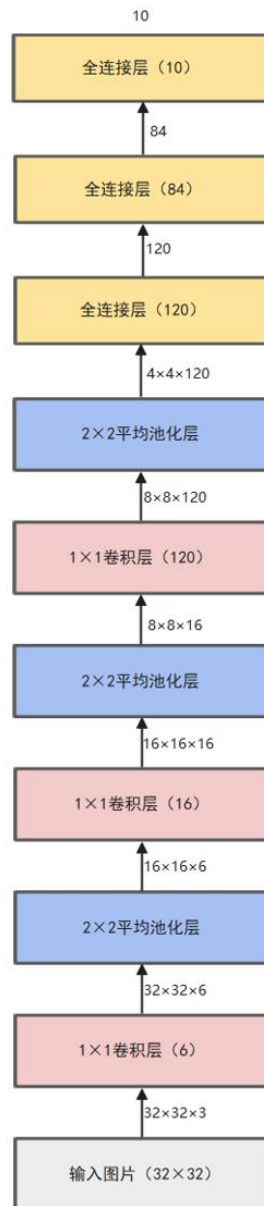
2 CNN 的学习与实践

2.1 画 CNN 网络矩阵图和流程图

根据该 CNN 的代码：

```
#进行网络的初始化
def initialization(self):
    for i in range(self.pop_size):
        model = Sequential()
        if self.dataset == 'cifar10':
            model.add(Conv2D(6, (1, 1), activation='relu', use_bias=False, input_shape=(32, 32, 3)))
            model.add(AveragePooling2D((2, 2)))
            model.add(Conv2D(16, (1, 1), activation='relu', use_bias=False))
            model.add(AveragePooling2D((2, 2)))
            model.add(Conv2D(120, (1, 1), activation='relu', use_bias=False))
            model.add(AveragePooling2D((2, 2)))
            model.add(Flatten())
            model.add(Dense(120, activation='relu', use_bias=False))
            model.add(Dense(84, activation='relu', use_bias=False))
            model.add(Dense(10, activation='softmax', use_bias=False))
```

画出该 CNN 网络结构的流程图和矩阵图：



在以上所示的 CNN 网络代码中，存在三个卷积层，三个平均池化层，三个全连接层。在定义卷积层时没有指定步长和填充，则默认为 1 和 0，池化层的 kernel 大小为 2x2 则其步长为 2。所以当输入图片的 shape 是 32x32x3 时，经过卷积层后不改变图片的分辨率大小只增加了通道数，经过平均池化层后不改变通道数而图片的分辨率减半，据此可以画出流程图和矩阵图。

如果记， W_{in} 为输入图片的宽， H_{in} 为输入图片的高，kernel 大小为 $w \times h$ ，步长为 s ，填

充为 p , W_{out} 为输出图片的宽, H_{out} 为输出图片的高, 则有以下关系:

$$W_{out} = 1 + \frac{W_{in} + 2p - w}{s} \quad (1)$$

$$H_{out} = 1 + \frac{H_{in} + 2p - h}{s} \quad (2)$$

2.2 用 pytorch 尝试实现以上 CNN 网络

代码如下:

```
1. import torch
2. from torch import nn
3. import torch.nn.functional as F
4. from torchsummary import summary
5.
6. # 模型构建
7. class Model(nn.Module):
8.
9.     def __init__(self):
10.         super(Model, self).__init__()
11.         self.conv1 = nn.Conv2d(in_channels=3,out_channels=6,kernel_size=(1,1))
12.         self.conv2 = nn.Conv2d(in_channels=6,out_channels=16,kernel_size=(1,1))
13.         self.conv3 = nn.Conv2d(in_channels=16,out_channels=120,kernel_size=(1,1))
14.         self.flatten = nn.Flatten()
15.         self.fc1 = nn.Linear(in_features=120*4*4,out_features=120)
16.         self.fc2 = nn.Linear(in_features=120,out_features=84)
17.         self.fc3 = nn.Linear(in_features=84, out_features=10)
18.
19.     def forward(self,x):
20.         x = F.avg_pool2d(F.relu(self.conv1(x)),2)
21.         x = F.avg_pool2d(F.relu(self.conv2(x)),2)
22.         x = F.avg_pool2d(F.relu(self.conv3(x)), 2)
23.         x = self.flatten(x)
24.         x = F.relu(self.fc1(x))
25.         x = F.relu(self.fc2(x))
26.         x = self.fc3(x)
27.         return F.softmax(input=x,dim=1)
28. device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
29. model = Model().to(device)
30.
31. print(summary(model,input_size=(3,32,32)))
```

在打印网络结构时, 采用第三方库 torchsummary 将构建的 CNN 网络相关信息成功打印如下:


```
D:\DL\Anaconda3\envs\pyt_1\python.exe D:/DL/PycharmWorkspace/pythonProject/CNN-test01/test01.py
-----
Layer (type)           Output Shape          Param #
-----
Conv2d-1               [-1, 6, 32, 32]       18
Conv2d-2               [-1, 16, 16, 16]      96
Conv2d-3               [-1, 120, 8, 8]       1,920
Flatten-4              [-1, 1920]            0
Linear-5               [-1, 120]             230,400
Linear-6               [-1, 84]              10,080
Linear-7               [-1, 10]              840
-----
Total params: 243,354
Trainable params: 243,354
Non-trainable params: 0
-----
Input size (MB): 0.01
Forward/backward pass size (MB): 0.15
Params size (MB): 0.93
Estimated Total Size (MB): 1.09
-----
None

Process finished with exit code 0
```

上图表中的 Layer 列的值依次对应的 CNN 网络各层名称，Output Shape 列的值为各层输出的形状（最前面的-1 表示批量大小，当不指定批量大小时默认该值为-1），Param 列的值为各层的参数量。

网络构建成功后，考虑尝试采用 cifar10 数据集进行训练和测试。

2.3 利用以上网络训练和测试 cifar10 数据集

设置相关超参数：

1. # 参数设置
2. batch = 256 #批量大小
3. lr = 3e-3 #学习率
4. epoch = 40 #训练的轮数

下载 cifar10 数据集并且查看数据集大小，代码如下：

1. #下载 cifar10 数据集
2. trian_data = torchvision.datasets.CIFAR10(root='D:\DL\PycharmWorkspace\pythonProject\CNN-test01',train=True,transform=torchvision.transforms.ToTensor(),
3. download=True)
4. test_data = torchvision.datasets.CIFAR10(root='D:\DL\PycharmWorkspace\pythonProject\CNN-test01',train=False,transform=torchvision.transforms.ToTensor(),
5. download=True)
6. #查看数据集大小
7. trian_data_size = len(trian_data)
8. test_data_size = len(test_data)
9. print('训练集大小: {}'.format(trian_data_size))

```
10. print('测试集大小{}'.format(test_data_size))
```

加载数据集:

```
1. #加载数据集
2. trian_dataloader = DataLoader(trian_data,batch_size=batch) #加载训练集
3. test_dataloader = DataLoader(test_data,batch_size=batch) #加载测试集
```

定义损失函数和参数优化器:

```
1. #定义交叉熵损失函数
2. loss_fn = nn.CrossEntropyLoss().cuda()
3. #定义优化器
4. optimizer = torch.optim.SGD(model.parameters(),lr=lr)
```

最后编写训练和测试的代码, 并且画出最后训练集和测试集上的准确率曲线:

```
1. #训练和测试
2. epoch_list = []
3. train_accuracy_list = []
4. test_accuracy_list = []
5.
6. for i in range(epoch):
7.     print('-----第{}轮训练开始-----'.format(i+1))
8.     model.train()
9.     total_accuracy = 0 #整体准确率
10.    for data in trian_dataloader:
11.        imgs,targets = data
12.        imgs = imgs.cuda()
13.        targets = targets.cuda()
14.        outputs = model(imgs)
15.        accuracy = (outputs.argmax(1) == targets).sum()
16.        total_accuracy += accuracy
17.        loss = loss_fn(outputs,targets)
18.        #优化器优化模型
19.        optimizer.zero_grad() #优化器梯度清零
20.        loss.backward() #反向传播
21.        optimizer.step() #优化器进行优化
22.        print('train_accuracy:{}'.format(total_accuracy/trian_data_size))
23.        epoch_list.append(i)
24.        train_accuracy_list.append((total_accuracy/trian_data_size).cpu().detach().numpy())
25.
26.
27. #测试
28. model.eval()
29. total_accuracy = 0 #整体准确率
30. with torch.no_grad(): #网络模型没有梯度, 不需要梯度优化
31.     for data in test_dataloader:
32.         imgs,targets = data
33.         imgs = imgs.cuda()
```

```

34.     targets = targets.cuda()
35.     outputs = model(imgs)
36.     accuracy = (outputs.argmax(1)==targets).sum()
37.     total_accuracy+=accuracy
38.     print('test_accuracy:{ }'.format(total_accuracy/test_data_size))
39.     test_accuracy_list.append((total_accuracy/test_data_size).cpu().detach().numpy())
40.
41.
42. # 绘图
43. plt.plot(epoch_list,train_accuracy_list,'-',epoch_list,test_accuracy_list,'-')
44. plt.title('accuracy')
45. plt.legend(['train_accuracy','test_accuracy'])
46. plt.show()

```

完整代码如下：

```

1.  import torch
2.  from torch import nn
3.  import torch.nn.functional as F
4.  from torch.utils.data import DataLoader
5.  import torchvision
6.  import matplotlib.pyplot as plt
7.  from torchsummary import summary
8.  import os
9.  os.environ['KMP_DUPLICATE_LIB_OK']='TRUE'
10.
11. # 参数设置
12. batch = 256 #批量大小
13. lr = 0.9 #学习率
14. epoch = 10 #训练的轮数
15.
16. # 模型构建
17. class Model(nn.Module):
18.
19.     def __init__(self):
20.         super(Model, self).__init__()
21.         self.conv1 = nn.Conv2d(in_channels=3,out_channels=6,kernel_size=(1,1))
22.         self.conv2 = nn.Conv2d(in_channels=6,out_channels=16,kernel_size=(1,1))
23.         self.conv3 = nn.Conv2d(in_channels=16,out_channels=120,kernel_size=(1,1))
24.         self.flatten = nn.Flatten()
25.         self.fc1 = nn.Linear(in_features=120*4*4,out_features=120)
26.         self.fc2 = nn.Linear(in_features=120,out_features=84)
27.         self.fc3 = nn.Linear(in_features=84, out_features=10)
28.
29.     def forward(self,x):
30.         x = F.avg_pool2d(F.relu(self.conv1(x)),2)

```

```

31.     x = F.avg_pool2d(F.relu(self.conv2(x)),2)
32.     x = F.avg_pool2d(F.relu(self.conv3(x)), 2)
33.     x = self.flatten(x)
34.     x = F.relu(self.fc1(x))
35.     x = F.relu(self.fc2(x))
36.     x = self.fc3(x)
37.     return F.softmax(input=x,dim=1)
38. device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
39. model = Model().to(device)
40. #print(summary(model,input_size=(3,32,32)))
41.
42. #下载 cifar10 数据集
43. trian_data = torchvision.datasets.CIFAR10(root='D:\DL\PycharmWorkspace\pythonProject\C
    NN-test01',train=True,transform=torchvision.transforms.ToTensor(),
44.                                         download=True)
45. test_data = torchvision.datasets.CIFAR10(root='D:\DL\PycharmWorkspace\pythonProject\CN
    N-test01',train=False,transform=torchvision.transforms.ToTensor(),
46.                                         download=True)
47. #查看数据集大小
48. trian_data_size = len(trian_data)
49. test_data_size = len(test_data)
50. print('训练集大小: {}'.format(trian_data_size))
51. print('测试集大小{}'.format(test_data_size))
52.
53. #加载数据集
54. trian_dataloader = DataLoader(trian_data,batch_size=batch) #加载训练集
55. test_dataloader = DataLoader(test_data,batch_size=batch) #加载测试集
56.
57. #定义交叉熵损失函数
58. loss_fn = nn.CrossEntropyLoss().cuda()
59.
60. #定义优化器
61. optimizer = torch.optim.SGD(model.parameters(),lr=lr)
62.
63. #训练和测试
64. epoch_list = []
65. train_accuracy_list = []
66. test_accuracy_list = []
67.
68. for i in range(epoch):
69.     print('-----第{}轮训练开始-----'.format(i+1))
70.     model.train()
71.     total_accuracy = 0 #整体准确率
72.     for data in trian_dataloader:

```

```

73.     imgs,targets = data
74.     imgs = imgs.cuda()
75.     targets = targets.cuda()
76.     outputs = model(imgs)
77.     accuracy = (outputs.argmax(1) == targets).sum()
78.     total_accuracy += accuracy
79.     loss = loss_fn(outputs,targets)
80.     #优化器优化模型
81.     optimizer.zero_grad() #优化器梯度清零
82.     loss.backward() #反向传播
83.     optimizer.step() #优化器进行优化
84.     print('train_accuracy:{}'.format(total_accuracy/trian_data_size))
85.     epoch_list.append(i)
86.     train_accuracy_list.append((total_accuracy/trian_data_size).cpu().detach().numpy())
87.
88.
89.     #测试
90.     model.eval()
91.     total_accuracy = 0 #整体准确率
92.     with torch.no_grad(): #网络模型没有梯度，不需要梯度优化
93.         for data in test_dataloader:
94.             imgs,targets = data
95.             imgs = imgs.cuda()
96.             targets = targets.cuda()
97.             outputs = model(imgs)
98.             accuracy = (outputs.argmax(1)==targets).sum()
99.             total_accuracy+=accuracy
100.         print('test_accuracy:{}'.format(total_accuracy/test_data_size))
101.         test_accuracy_list.append((total_accuracy/test_data_size).cpu().detach().numpy())
102.
103.
104. # 绘图
105. plt.plot(epoch_list,train_accuracy_list,'.',epoch_list,test_accuracy_list,'-')
106. plt.title('accuracy')
107. plt.legend(['train_accuracy','test_accuracy'])
108. plt.show()

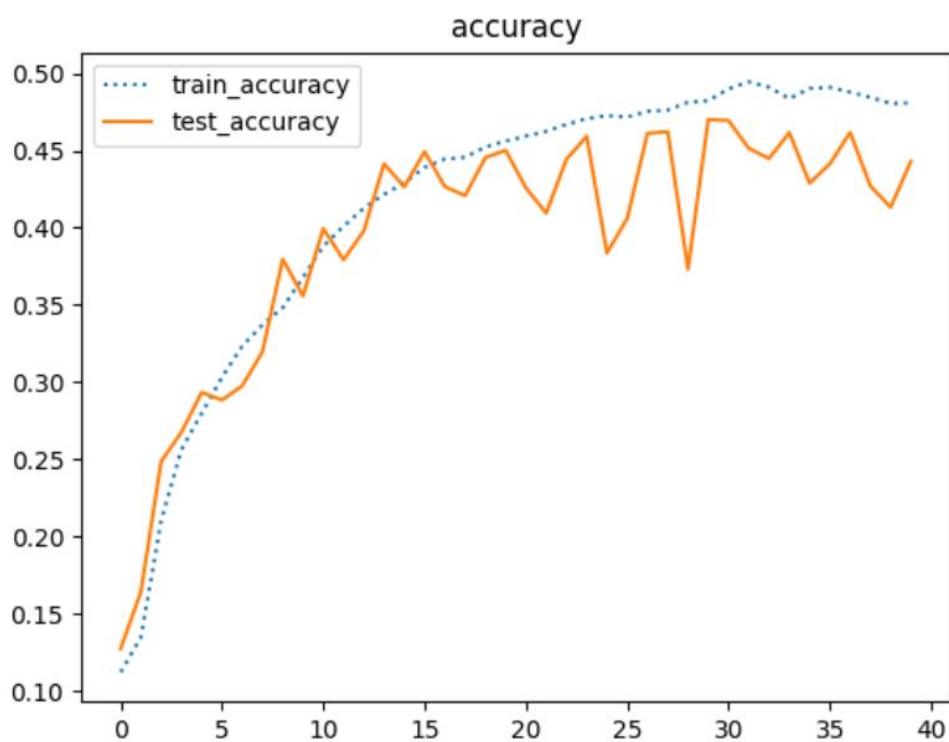
```

2.4 部分训练过程以及最后的结果曲线图

```

-----第21轮训练开始-----
train_accuracy:0.4592999815940857
test_accuracy:0.42579999566078186
-----第22轮训练开始-----
train_accuracy:0.46243998408317566
test_accuracy:0.40950000286102295
-----第23轮训练开始-----
train_accuracy:0.46691998839378357
test_accuracy:0.44429999589920044
-----第24轮训练开始-----
train_accuracy:0.47061997652053833
test_accuracy:0.459199994802475
-----第25轮训练开始-----

```

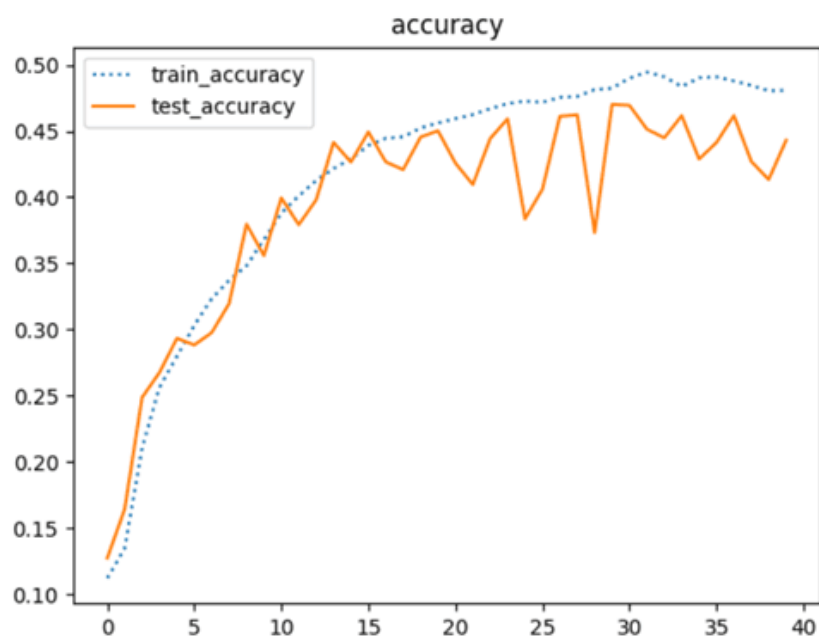


根据训练和测试的准确率曲线可以看出，目前准确率不是很高，并且出现了一定程度的过拟合现象，接下来考虑改进网络结构并且增加一些图像数据增广的方法。

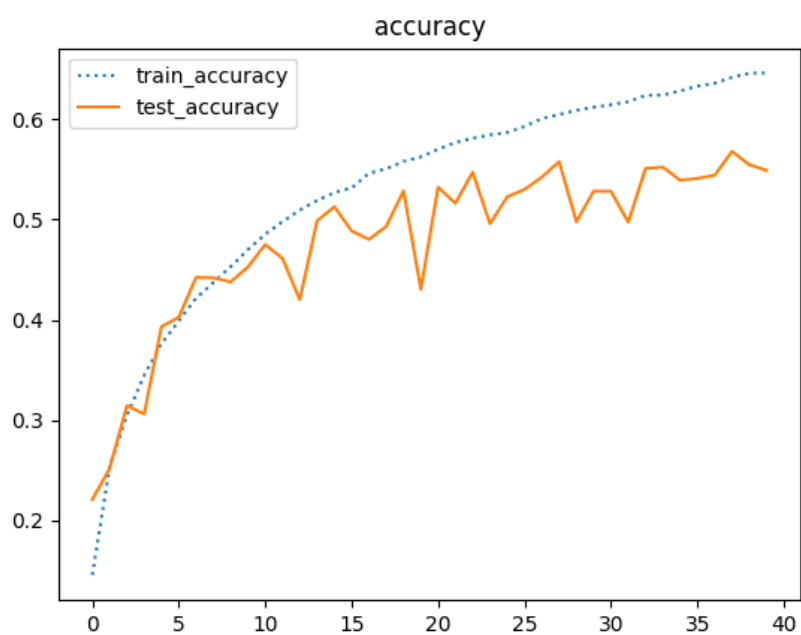
3 针对原始 CNN 网络的改进

3.1 加入残差结构

改进前设定学习率为 0.9, batch_size 为 256, epoch 为 40, 未加残差结构, 在 cifar10 数据集上的训练结果如下图所示:

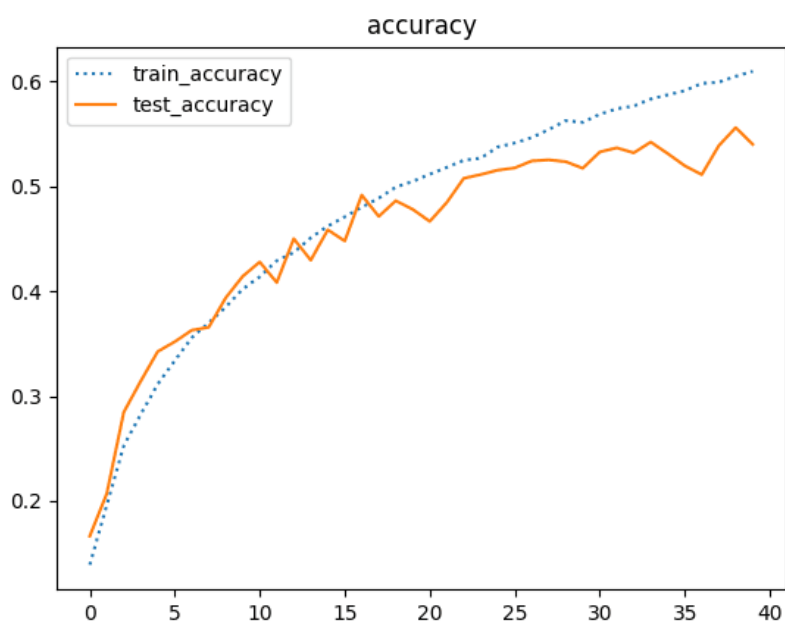


在原始 CNN 网络结构的基础上加入残差结构后, 设定学习率为 0.9, batch_size 为 256, epoch 为 40, 依然用 cifar10 数据集。准确率与之前相比有所提升但测试集准确率曲线存在较大波动, 并且出现一定的过拟合现象, 如下图所示:



3.2 尝试调节部分超参数以及加入 dropout

进一步尝试将 `batch_size` 调大为 512，其余超参数设置不变，效果如下：



可以看到与之前相比测试集准确率曲线变得平滑一些，过拟合现象得到一定的缓解。

之后，进一步加入 dropout 层(`nn.Dropout2d(p=0.3)`):

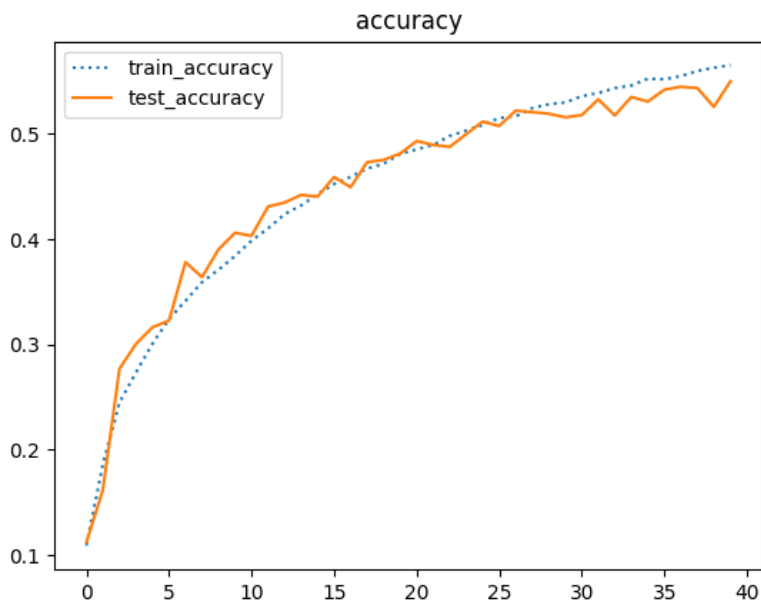
1. `class Model(nn.Module):`
- 2.


```

3.     def __init__(self):
4.         super(Model, self).__init__()
5.         self.conv1 = nn.Conv2d(in_channels=3,out_channels=6,kernel_size=(1,1),bias=False)
6.         self.bn1 = nn.BatchNorm2d(6)
7.         self.conv2 = nn.Conv2d(in_channels=6,out_channels=16,kernel_size=(1,1),bias=False)
8.         self.bn2 = nn.BatchNorm2d(16)
9.         self.conv3 = nn.Conv2d(in_channels=16,out_channels=120,kernel_size=(1,1),bias=False)
10.
11.        self.bn3 = nn.BatchNorm2d(120)
12.        self.downsample = nn.Conv2d(in_channels=3,out_channels=120,kernel_size=(4,4),stride=
13.            4,bias=False)
14.        self.flatten = nn.Flatten()
15.        self.fc1 = nn.Linear(in_features=120*4*4,out_features=120,bias=False)
16.        self.fc2 = nn.Linear(in_features=120,out_features=84,bias=False)
17.        self.fc3 = nn.Linear(in_features=84, out_features=10,bias=False)
18.        self.dropout = nn.Dropout2d(p=0.3)
19.
20.    def forward(self,x):
21.        residual = x.clone() # 3x32x32
22.        x = self.conv1(x) # 6x32x32
23.        x = self.bn1(x)
24.        x = F.relu(x) # 6x32x32
25.        x = F.avg_pool2d(x,2) # 6x16x16
26.        x = self.dropout(x)
27.        x = self.conv2(x) # 16x16x16
28.        x = self.bn2(x)
29.        x = F.relu(x)
30.        x = F.avg_pool2d(x,2) # 16x8x8
31.        x = self.dropout(x)
32.        x = self.conv3(x) # 120x8x8
33.        x = self.bn3(x)
34.        residual = self.downsample(residual)
35.        x += residual
36.        x = F.relu(x)
37.        x = F.avg_pool2d(x,2)
38.        x = self.dropout(x)
39.        x = self.flatten(x)
40.        x = F.relu(self.fc1(x))
41.        x = F.relu(self.fc2(x))
42.        x = self.fc3(x)
43.        return F.softmax(input=x,dim=1)

```

效果如下：



可以看到与之前相比测试集准确率曲线变得更加平滑并且进一步缓解了过拟合现象。

3.3 尝试将遗传算法整合到 CNN 网络中

相关超参数设置：

种群个体数=4,

染色体上基因交换率=0.5,

个体变异率=0.5,

个体的染色体上的基因变异率=0.1,

最大算法迭代次数 $r=10$ (超过则算法停止)

适应度阈值=0.8 (超过阈值则算法停止)

到下一代中的上一代精英个数=2,

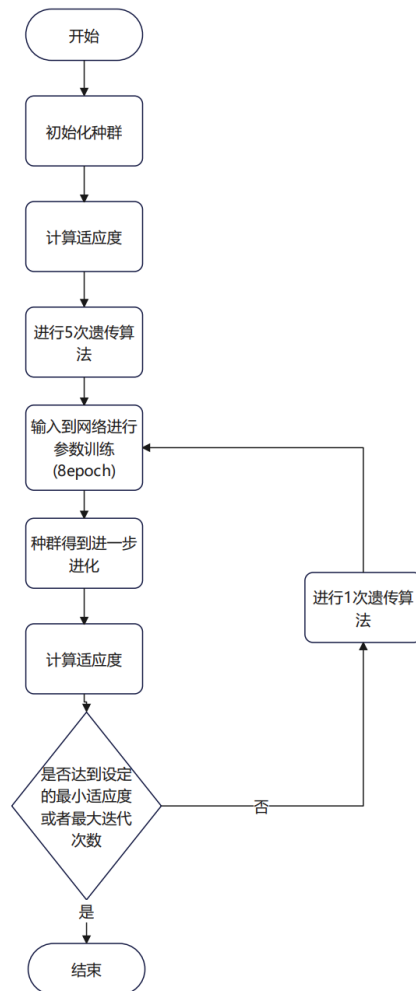
交配池大小=2,

批量大小=512,

学习率=0.9,

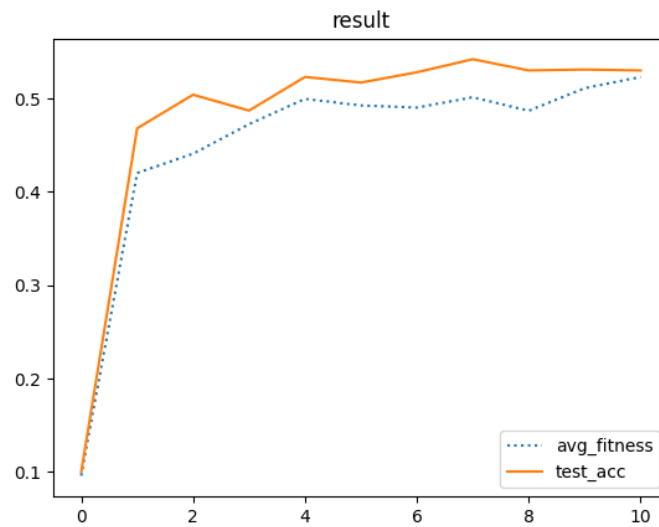
网络训练轮数=8

整体算法流程如下图所示：

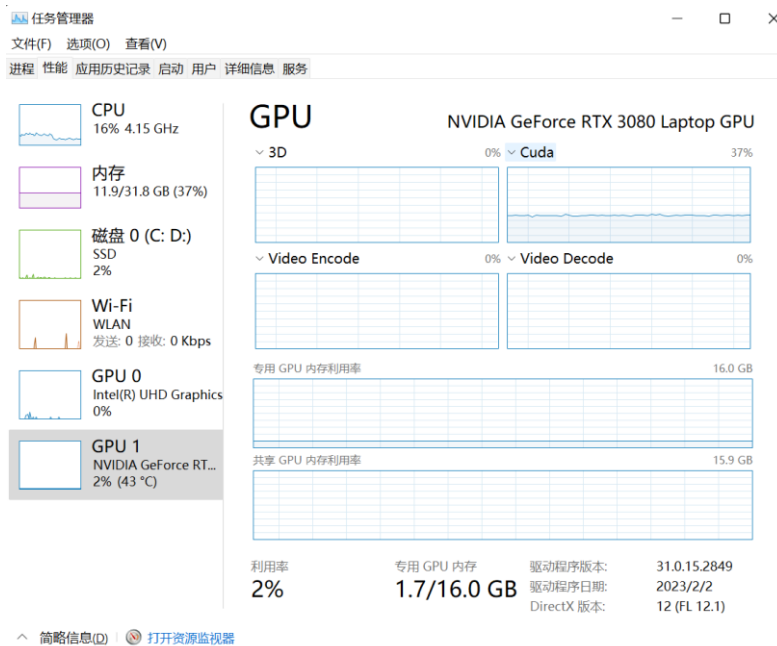


首先初始化种群个体数为 4，经过 5 次遗传算法（交叉、变异、选择）后，到神经网络中训练，计算新种群的适应度，之后再经过 1 次遗传算法，再到神经网络中训练，如此交替执行迭代 10 次。计算适应度时选择测试集的 1000 个数据进行测试，训练网络时用全部训练集数据以 512 的 batch 小批量训练 8 个 epoch。

结果如下：



算法执行过程中资源占用情况如下所示：



内存在执行程序后大概占用提升了 3-4G 因为采用 GPU 训练，CPU 占用率较低，显存占用大约为 1.7G

程序运行过程控制台部分输出以及耗用的时间如下图所示：

```
Iter: 8
Best_fit: {'pop': 1, 'test_accuracy': 0.53}, avg_fitness: 0.4868
Elites: [0, 1]
Pairs: [[1, 1], [1, 1]]
Cross over finished.
Replacement finished.
Mutation(2) finished.

Iter: 9
Best_fit: {'pop': 0, 'test_accuracy': 0.531}, avg_fitness: 0.5110
Elites: [1, 0]
Pairs: [[0, 0], [0, 0]]
Cross over finished.
Replacement finished.

Iter: 10
Best_fit: {'pop': 3, 'test_accuracy': 0.53}, avg_fitness: 0.5230
Elites: [2, 3]
Pairs: [[1, 1], [0, 0]]
Cross over finished.
Replacement finished.
Mutation(3) finished.
Maximum iterations(10) reached.
程序执行经过22.38分钟
```

4 针对 GA-CNN 网络的改进

将遗传算法整合到 CNN 网络后, 将 CNN 网络进一步换为 resnet50 网络。从而以 resnet50 网络作为对照组, GA-resnet50 网络作为实验组, 进一步对实验组进行尝试改进。

针对实验组首先主要的改进有三个方面:

1. 调节部分遗传算法的超参数
2. 将 CNN 网络的学习率调整为学习率衰减的形式
3. 将遗传算法中交叉和变异的作用范围进行调整, 分别作用于网络的一半

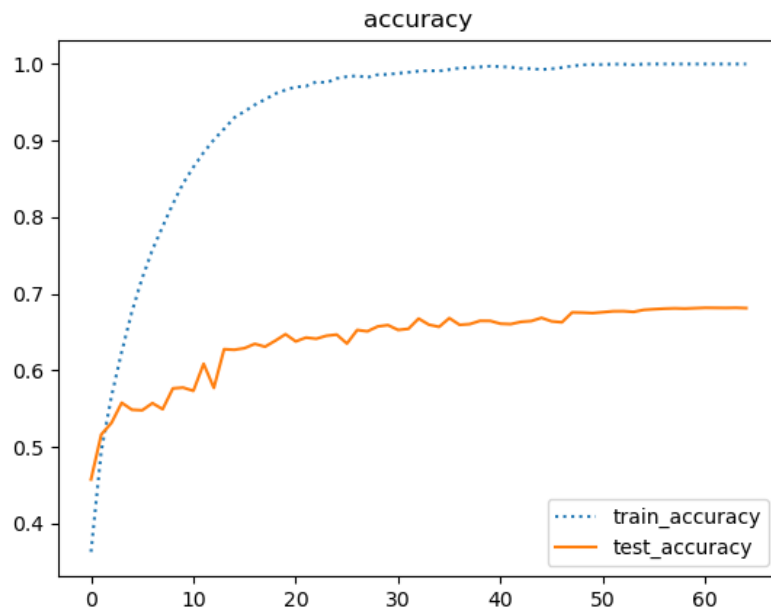
4.1 对照组在 cifar10 数据集上的效果

参数设置:

Lr=0.001

Epoch=65

Batch_size=64



整个过程用时为：28.62 分钟

4.2 实验组在 cifar10 数据集上的效果

参数设置如下：

```
_init_pop_size=20,  
_pop_size=4,  
_p_crossover=0.2,  
_p_mutation=0.2,  
_r_mutation=0.08,  
_max_iter=8,  
_min_fitness=0.7,  
init_elite_num=5,  
_elite_num=1,  
init_mating_pool_size=8,  
_mating_pool_size=4,  
_batch_size=64,  
lr=0.001,  
epoch=2
```

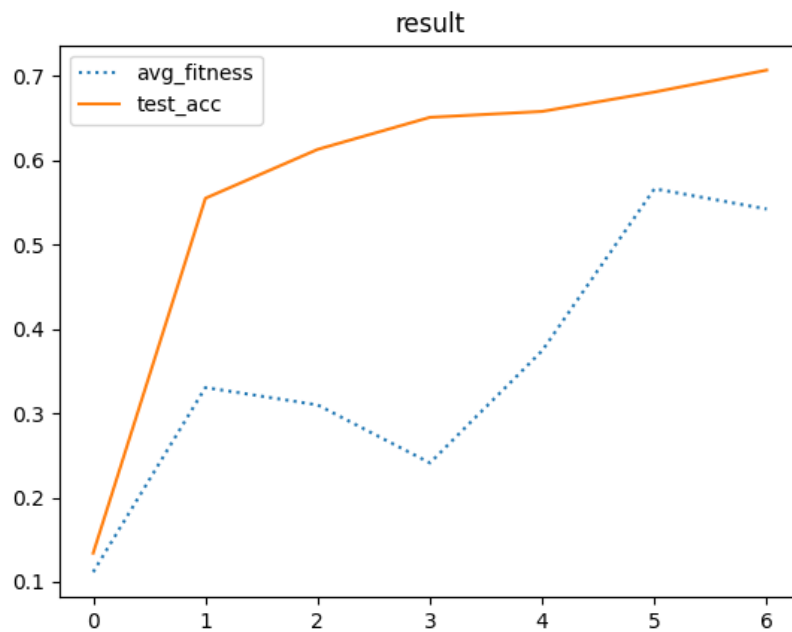
初始化阶段：

初始化设置种群个体数为 20，经过 2 次 GA 后，选择其中的 4 个较好个体组成新种群（该新种群的个体数为 4），

交替训练阶段：

然后这个新种群进入网络进行训练，再经过 1 次 GA，如此反复交替，直到达到设定条件。

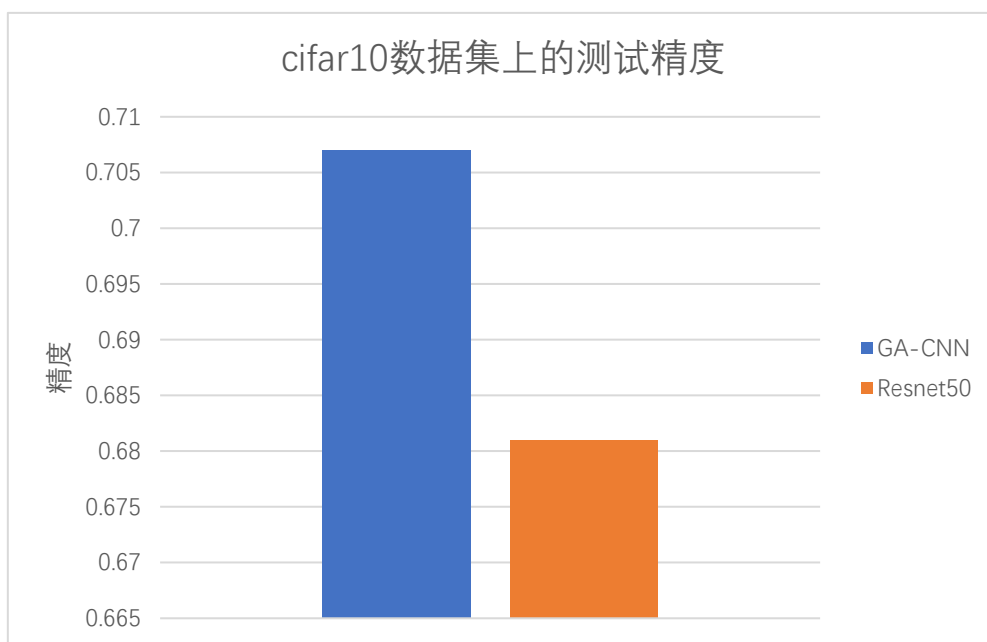
初始化后，种群个体数为 4，每个个体在网络中训练轮数 epoch 为 2，整个算法最大执行次数_max_iter 为 8。在第 6 次迭代时，精度已达到算法设定的最小精度 0.7，算法停止。



整个过程用时为：19.55 分钟

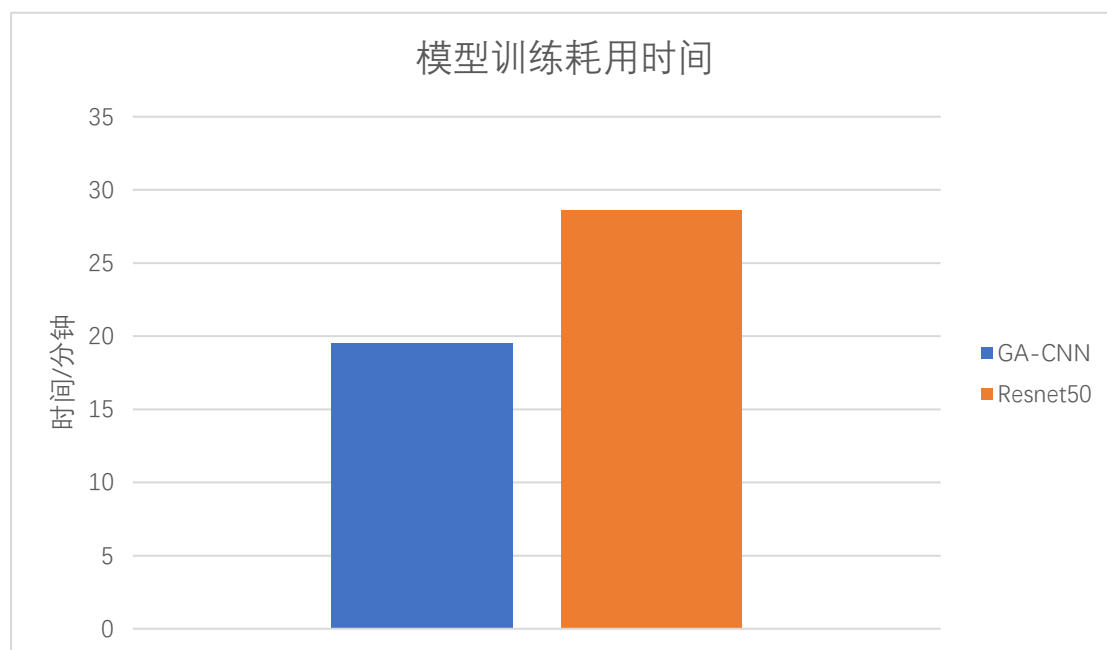
4.3 二者在 cifar10 数据集上的精度与耗用时间对比

以上两种方法所得模型在 cifar10 数据集上的测试集精度，如下图所示：



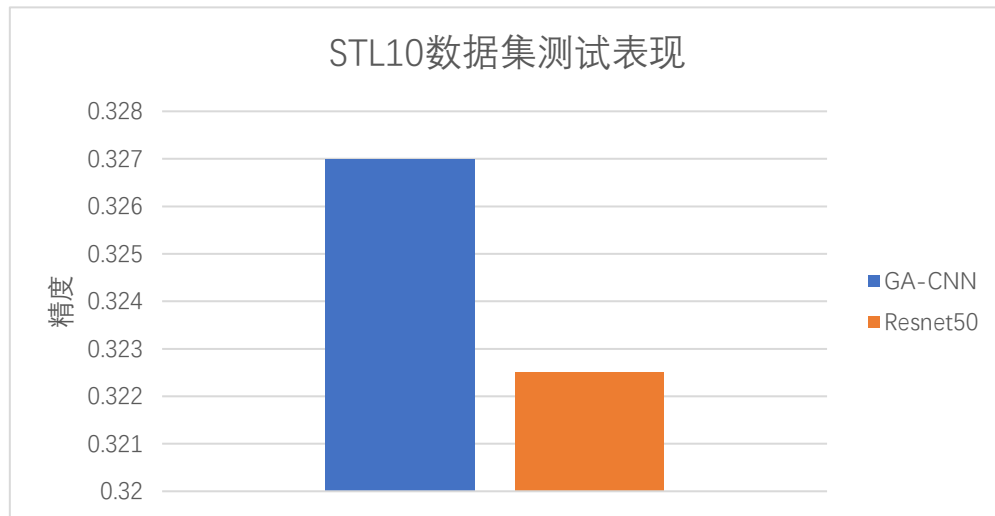
上图结果表明, 在 cifar10 数据集上 GA-CNN 算法所得模型精度相比于 Resnet50 模型精度大约提升了 3.9%

以上两种模型在 cifar10 数据集上训练时所耗用的时间, 如下图所示:



上图结果表明, GA-CNN 算法迭代所耗用时间相比于 Resnet50 训练所耗用时间, 大约减少了 31.69%

4.4 二者在 STL10 数据集上的精度对比



上图结果表明，两种方法所得到的模型在 STL10 数据集上测试：GA-CNN 精度比 Resnet50 精度提升大约 1.4%

4.5 尝试选用不同优化器与不同 ratio 在 cifar10 上的对比

注：
ratio为一超参数，衡量交叉和变异作用于染色体范围的比率， $\text{ratio} = \text{交叉范围} / \text{染色体总长度}$ ； $1 - \text{ratio} = \text{变异范围} / \text{染色体总长度}$
上周的最好结果选用sgd优化器，ratio为0.5，因为上周算法停止的最小适应度设置为0.7，故没有跑满8轮，8轮估计时间为26.06分钟
选用不同优化器以及不同ratio所得到的结果如下：

对照组：

优化器	最终收敛精度	精度收敛到0.7耗时	65epoch总共用时
SGD	0.686	精度未达到0.7	28.62分钟
AdamW	0.736	26.02分钟	42.28分钟

实验组：

优化器	ratio	精度	精度达0.7耗时	达到最高精度耗时	八轮总共用时
SGD	0.3	0.682	精度未达到0.7	28.24分钟(第八次迭代0.682)	28.24分钟
SGD	0.4	0.720	17.64分钟(第五次迭代0.709)	24.69分钟(第七次迭代0.720)	28.22分钟
SGD	0.5	0.707	19.55分钟(第六次迭代0.707)	19.55分钟(第六次迭代0.707)	26.06分钟(估计) *上周最佳
SGD	0.6	0.679	精度未达到0.7	28.64分钟(第八次迭代0.679)	28.64分钟
AdamW	0.3	0.738	19.05分钟(第四次迭代0.704)	28.58分钟(第六次迭代0.738)	38.11分钟
AdamW	0.4	0.733	28.31分钟(第六次迭代0.700)	33.03分钟(第七次迭代0.733)	37.75分钟
AdamW	0.5	0.775	23.84分钟(第六次迭代0.731)	31.79分钟(第八次迭代0.775)	31.79分钟
AdamW	0.6	0.761	25.42分钟(第五次迭代0.717)	40.67分钟(第八次迭代0.761)	40.67分钟

5 目标检测网络 retinanet 的学习

5.1 retinanet 解决的问题

retinanet 出现之前，目标检测领域普遍以 yolo 系列、SSD 算法为代表的 one-stage 算法准确率不如以 Faster RCNN 为代表的 two-stage 算法。出现这个问题的原因主要有两个：

其一：算法的差异所导致

two-stage 算法，先采用 RPN 网络(区域生成网络)筛选、提取出一系列的 Region Proposal (区域候选框)，而这些区域候选框都是很有可能包含待检测物体的；然后在第一阶段的基础上再微调进行细粒度的物体检测，自然精度会更高一些，但是相对来说速度会比较慢。

One-stage 算法，则是直接在网络中提取特征来预测待检测物体的类别和位置，速度会快一些，但精度自然会比较低。

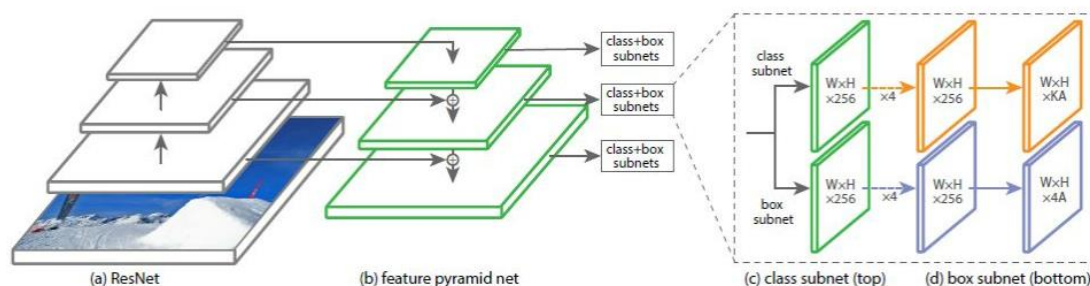
其二：样本不均衡

Faster RCNN 明确了训练时正负样本比 1:3，而 one-stage 算法是极度不均衡的 (1:1000 都是有可能的)。并且 one-stage 算法因为是一步到位，但是在这个过程中可能会出现大量的简单样本（背景），复杂样本（待检测物体）只占很小一部分，这些简单样本是我们不关注的区域，但在 loss 计算过程中被这些大量的简单样本所主导，造成最终精度的降低。

而 RetinaNet 的出现，在一定程度上改善了这个问题，让 one-stage 的方法具备了更好的准确率。

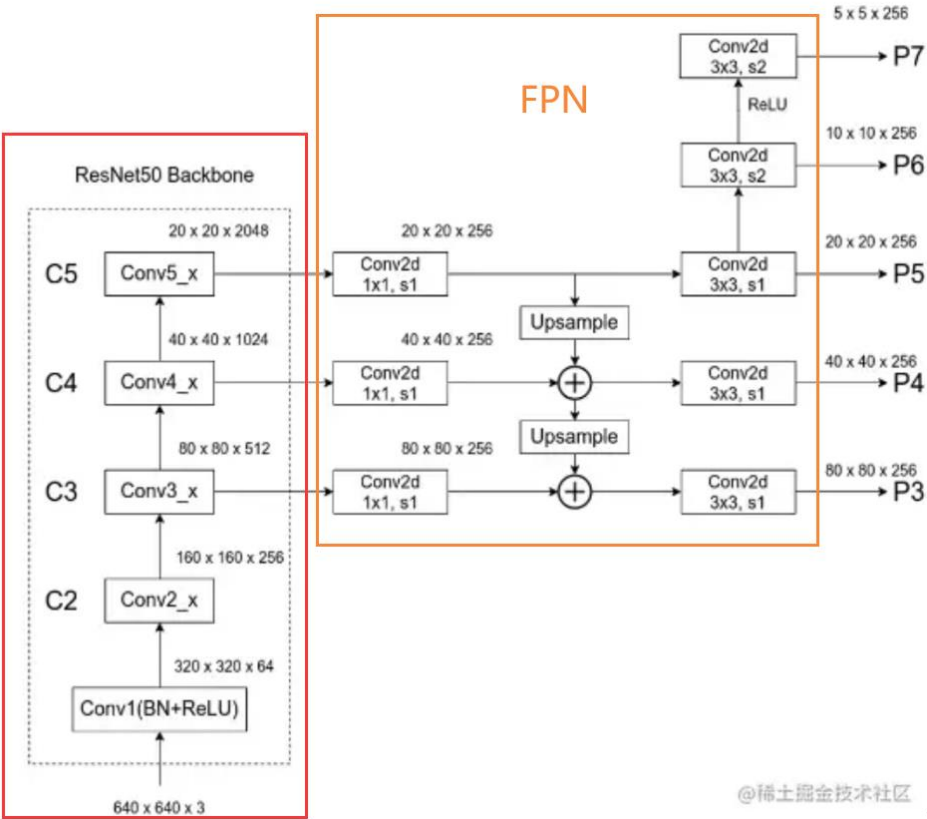
5.2 retinanet 原理

5.2.1 RetinaNet 网络结构



retinanet 属于一种 one-stage 算法，它的特征提取网络选择了 resnet 网络以提取多尺度的特征，特征融合选择了 FPN（特征金字塔网络），经过特征金字塔网络处理后，输出不同尺度的特征图，对每一种尺度的特征图，构建用于分类和框回归的子网络，从而最终得到待检测目标的位置和类别。分类网络输出的特征图尺寸为 (W,H,K_A) ，其中 W 、 H 为特征图宽

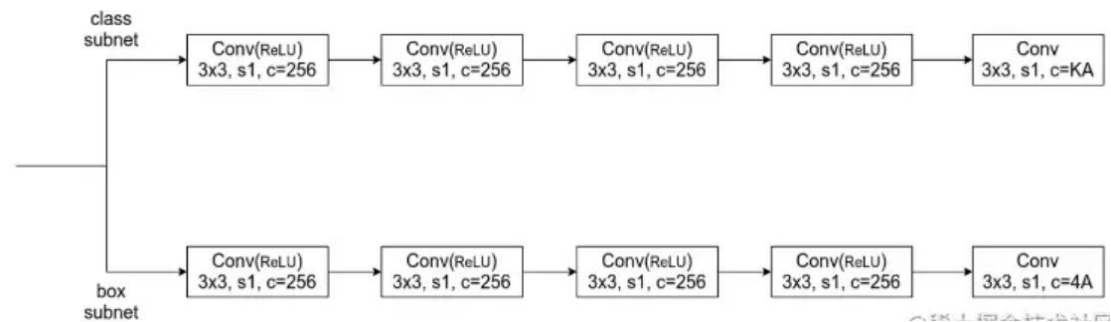
高，KA 为特征图通道，存放 A 个 anchor 各自的类别信息（K 为类别数）。



retinanet 设计改造了 FPN，考虑到 P2 的特征图尺寸是非常大的，非常消耗空间和计算量，FPN 中的 P2 被删掉了，同时在 P5 的基础上往上再延伸两层，进行两次下采样得到 P6 和 P7 层。P3 到 P7 对应了不同的尺度特征，P3 更适合小目标的检测，而 P7 更适合大目标的检测，作者根据尺度特点设计了相应的 anchor 机制，每个尺度上的每个特征图点对应 3 个不同大小和 3 个不同比例的 anchor，共 9 个，如下表所示：

Scale	Ratio
32 $\{2^0, 2^{1/3}, 2^{2/3}\}$	$\{1:2, 1:1, 2:1\}$
64 $\{2^0, 2^{1/3}, 2^{2/3}\}$	$\{1:2, 1:1, 2:1\}$
128 $\{2^0, 2^{1/3}, 2^{2/3}\}$	$\{1:2, 1:1, 2:1\}$
256 $\{2^0, 2^{1/3}, 2^{2/3}\}$	$\{1:2, 1:1, 2:1\}$
512 $\{2^0, 2^{1/3}, 2^{2/3}\}$	$\{1:2, 1:1, 2:1\}$

之后每个尺度特征后都会接一个检测头子网络，包括 class subnet 和 box subnet，负责分类和框预测，这里五个特征图后的检测头子网络都是权值共享的。具体的 class subnet 和 box subnet 结构如下图所示：

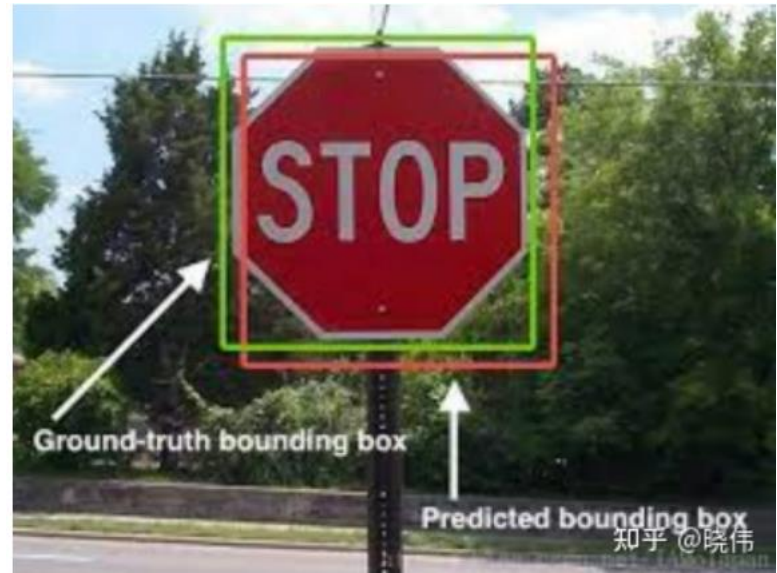


class subnet 子网络最后输出维度为 KA，其中 K 为要分类的类别数量（不包括背景），A 为 anchor 的数量 9。box subnet 输出维度为 4A，其中 4 表示中心点坐标和长宽，A 为 anchor 数量 9。这样就得到了最终的类别和框位置预测。

5.2.2 正负样本划分

IOU：用来衡量目标检测任务中，预测结果的位置信息的准确程度。

在目标检测中，我们需要从给定的图片里，推测出这张图片里有哪样（或者是哪几样）东西，并且推测这样（或者这几样）东西在图片中的具体位置。比如，在下面这张图片里，我们推测出图片当中有个 STOP 标识，并且给出了它的推测位置（红色的方框）。



但是，图片中绿色的方框才是 STOP 标识真正的位置。两个方框所在的位置存在着一定的偏差。我们通常使用 IoU (Intersection over Union)这个指标来衡量上面提到的偏差的大小。

$$IoU = \frac{\text{物体实际区域与推测区域重合的面积}}{\text{两个区域整体所占的面积}}$$

回到 retinanet，作者将以下作为正负样本的衡量标准：

- $\text{IOU} \geq 0.5$, 正样本
- $\text{IOU} < 0.4$, 负样本
- $\text{IOU} \in [0.4, 0.5)$, 舍弃

5.2.3 Focal Loss

前面提到的, one-stage 算法精度不如 two-stage 算法的样本不均衡原因——正负样本不均衡和难易样本不均衡, 就是在 Focal Loss 部分进行解决的。首先是传统的交叉熵损失函数:

$$CE(p, y) = -[y \log(p) + (1 - y) \log(1 - p)]$$

可以改写成这种形式:

$$CE(p, y) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1 - p) & \text{otherwise.} \end{cases}$$

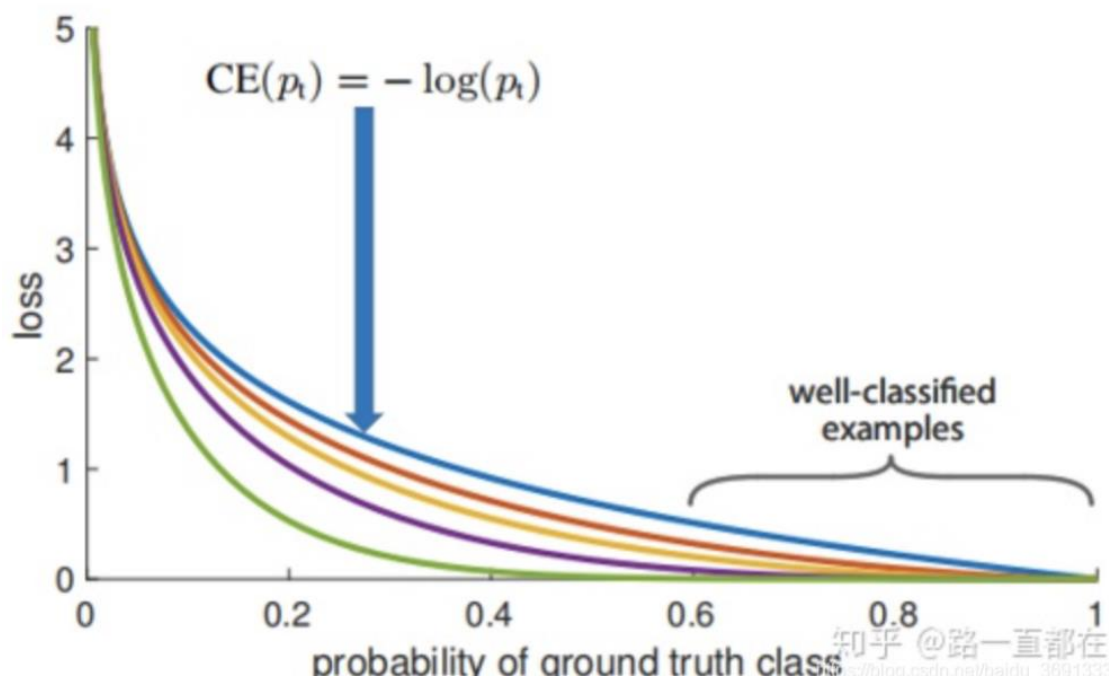
我们定义 p_t :

$$p_t = \begin{cases} p & \text{if } y = 1 \\ 1 - p & \text{otherwise,} \end{cases}$$

那么就有

$$CE(p, y) = CE(p_t) = -\log(p_t)。$$

而采用交叉熵损失函数在训练过程中会出现如下问题:



在分类很好的情况下，损失函数仍然有较大的 loss 值，而分类很好的情况大多是由于简单样本（背景）导致的，大量简单样本的 loss 相加就会主导整个模型训练过程，造成最终精度的下降。

因此，在 loss 函数的基础上加上权重，将 loss 函数改成平衡交叉熵损失函数，而该权重可以设置为样本数量的倒数，复杂样本（待检测物体）数量较少则权重更大：

$$CE(p_t) = -\alpha_t \log(p_t).$$

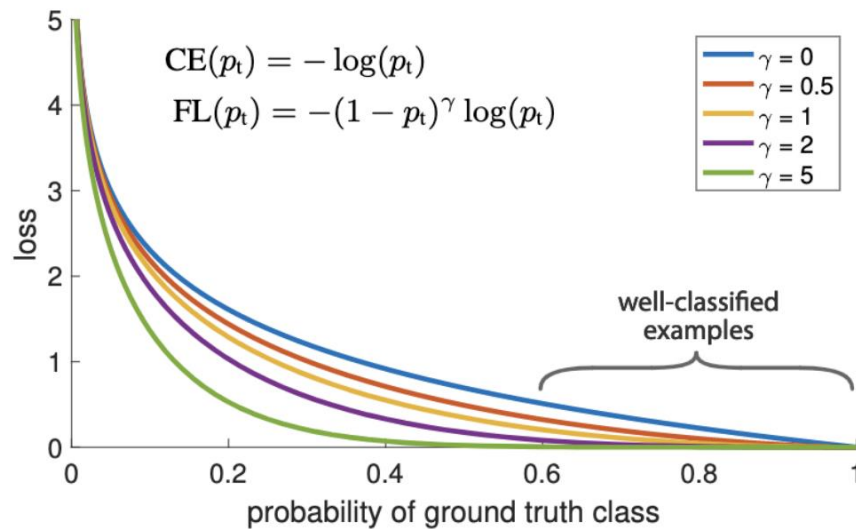
在此基础上引入权重 $(1 - p_t)^\gamma$, 其中 γ 是一个超参数，最终的 Facol Loss 如下：

$$FL(p_t) = -\alpha_t (1 - p_t)^\gamma \log(p_t).$$

展开之后如下形式：

$$FL(p) = \begin{cases} -\alpha(1-p)^\gamma \log(p) & \text{if } y = 1 \\ -(1-\alpha)p^\gamma \log(1-p) & \text{otherwise,} \end{cases}$$

采用 Facol Loss 与采用交叉熵损失函数的 loss 值对比图像如下所示：



Focal loss 是一个动态缩放的交叉熵损失，通过动态缩放因子，可以动态降低训练过程中易区分样本的权重，从而将 loss 函数的重心快速聚焦在那些复杂样本上。

5.3 retinanet 优劣

优点：

- 1.使得 one-stage 算法的准确率得到提升。
- 2.引入了 focal loss，降低了样本不均衡带来的影响。
- 3.在分辨率降低时，RetinaNet 具备非常优越的推理性能。

缺点：

- 1.RetinaNet 引入的 focal loss 易受噪声干扰，对图像标注的准确性要求非常高，一旦有标错的样本，就会被 focal loss 当做困难样本，干扰样本对 loss 贡献很大，从而影响学习效果。

6 retinanet 项目

注：目前的这个 retinanet 项目已经可以实现最终输出 [图片名称,IOU 值] 的工作。在 pytorch-retinanet\output 文件夹中的 output.txt 文件内有更详细的说明。

6.1 目前所构建的 Retinanet 项目使用流程

6.1.1 数据集需要转换成的格式说明

- (1)数据集中所有的 jpg 格式图片放在 pytorch-retinanet\data\images 文件夹内
- (2)创建 class_list.csv 文件，文件中所写的内容格式如下：

Class mapping format

The class name to ID mapping file should contain one mapping per line. Each line should use the following format:

```
class_name,id
```

Indexing for classes starts at 0. Do not include a background class as it is implicit.

For example:

```
cow,0  
cat,1  
bird,2
```

可参考目前项目中的 class_list.csv 文件进行修改

- (3)创建 train.csv 文件，文件中所写的内容格式如下：

	A	B	C	D	
1	20151127_115133.jpg				
2	20151127_115151.jpg				
3	20151127_115229.jpg				
4	20151127_115424.jpg				
5	20151127_120002.jpg				
6	20151127_120156.jpg				
7	20151127_120208.jpg				

包含训练集所有图片的名称

- (4)创建 val.csv 文件，文件中所写的内容格式与上述 2.1 的(3)相同，不同点是该文件中包含测试集所有图片的名称

- (5)创建 train_annot.csv 文件，文件中所写的内容格式如下：

Annotations format

The CSV file with annotations should contain one annotation per line. Images with multiple bounding boxes should use one row per bounding box. Note that indexing for pixel values starts at 0. The expected format of each line is:

```
path/to/image.jpg,x1,y1,x2,y2,class_name
```

Some images may not contain any labeled objects. To add these images to the dataset as negative examples, add an annotation where `x1`, `y1`, `x2`, `y2` and `class_name` are all empty:

```
path/to/image.jpg,,,,,
```

A full example:

```
/data/imgs/img_001.jpg,837,346,981,456,cow  
/data/imgs/img_002.jpg,215,312,279,391,cat  
/data/imgs/img_002.jpg,22,5,69,64,bird  
/data/imgs/img_003.jpg,,,,,
```

This defines a dataset with 3 images. `img_001.jpg` contains a cow. `img_002.jpg` contains a cat and a bird. `img_003.jpg` contains no interesting objects/animals.

第一列：训练集每张图片存放路径

第二列~第五列：图片上相应类别的标注框的左上角和右下角坐标(坐标原点为图片左上角，坐标单位为像素)

第六列：类别名称

可参考目前项目中的 train_annots.csv 文件进行修改

(6)创建 val_annots.csv 文件，文件中所写的内容格式和上述 2.1 的(5)相同，不同点是该文件中描述的是测试集，可参考目前项目中的 val_annots.csv 文件进行修改

(7)在 val 文件夹下存放所有 txt 文件，每个 txt 文件其实对应一张图片。代表要将哪些图片最后输出为[图片名称,IOU 值]的形式。每个 txt 文件中包含有对应图片上的所有类别名称以及类别对应的标注框信息(左上，右上，右下，左下角坐标)。

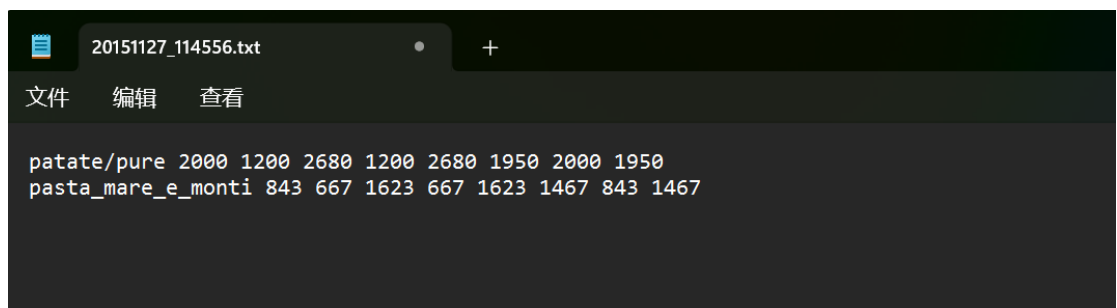
注：在实际构建的时候这些 txt 文件可以只写对应图片上的所有类别名称即可，类别对应的标注框信息有无均可(仅限 6.1.1 的(7)部分可以这样操作)

例如：



名称	修改日期	类型	大小
20151127_114556.txt	2023/3/3 17:30	文本文档	1 KB
20151127_114946.txt	2023/3/3 17:30	文本文档	1 KB

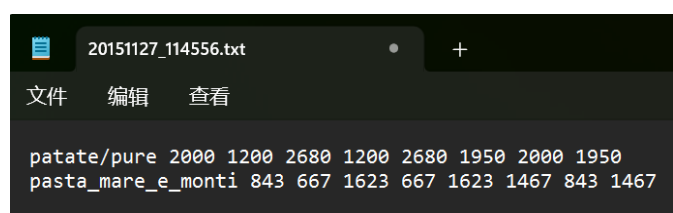
比如说 20151127_114556.txt 文件，其实对应了数据集中的 20151127_114556.jpg 图片。打开 20151127_114556.txt 文件可以看到



```
patate/pure 2000 1200 2680 1200 2680 1950 2000 1950
pasta_mare_e_monti 843 667 1623 667 1623 1467 843 1467
```

这代表了 20151127_114556.jpg 图片上有两个类别，以及这两个类别对应的标注框信息。在实际构建的过程中，这里的 txt 文件中的内容格式如下两种均可：

第一种：



```
patate/pure 2000 1200 2680 1200 2680 1950 2000 1950
pasta_mare_e_monti 843 667 1623 667 1623 1467 843 1467
```

第二种:



6.1.2 训练

2.1 完成后, 可直接运行 train.py 文件进行训练, 目前训练所用的超参数设置如下(未调参):

batchsize=1 (bz=1 时 16G 内存会占用大约 98%)

epoch=5,

学习率 1e-5,

优化器为 Adam

depth=50(表示此时特征提取网络采用 resnet50, 可选值有 18, 34, 50, 101, 152)

训练结束后会保存一个训练好的模型文件 model_final.pt

6.1.3 测试

2.2 完成后, 可直接运行 csv_validation.py 文件, 模型默认加载 model_final.pt, IOU 阈值默认为 0.5。该部分最终输出 [图片名称,IOU 值] 的具体过程代码在 csv_validation.py 文件中有详细的注释说明:

```
print(“正在评测中,.....”)\n\n# 一个字典:{类别标号:(在测试集中该类检测的平均IOU值,在测试集中包含有该类的标注框个数), ..... }\naverage_iou = csv_eval.evaluate(dataset_val, retinanet_iou_threshold=float(parser.iou_threshold))\n#average_iou = {0: (0.9073717518680259, 144.0), 1: (0.8542296415959562, 53.0), 2: (0.811257967417113, 44.0), 3: (0.866853157500508, 40.0), 4: (0.8217168\n\n# 转换,输出为txt保存。txt文件每一行为[图片名称,图片平均检测精度]\nfilePath = “./val” # 包含测试集相关txt文件的文件路径\nfileList = os.listdir(filePath) # 该列表中包含所有测试集的txt文件\ndict1 = {} # {图片名称:[标注框信息1],[标注框信息2]], ..... ,}\nfor file in fileList:\n    list = []\n    f = open(os.path.join(filePath, file))\n    for line in f.readlines():\n        line = line.strip(“\\n”)\n        line = line.split()\n        list.append(line)\n    f.close()\n    dict1[file.replace(“txt”, “jpg”)] = list # list是一个二维列表[[标注框信息1],[标注框信息2], ... ,]\n\nfor jpg_name in dict1: # dict1 = {图片名称:[标注框信息1],[标注框信息2]], ..... ,}\n    jpg_average_precision = 0 # 用于计算每一张图片的平均检测精度(以该图片上包含的所有类别的IOU均值来衡量)\n    for class_list in dict1[jpg_name]: # class_list = [类别名称,x1,y1,x2,y2,x3,y3,x4,y4] 其实在计算jpg_average_precision时只用类别名称即可\n        index = dataset_val.name_to_label[class_list[0]]\n        jpg_average_precision += average_iou[index][0] # 计算当前图片的平均检测精度\n\n    with open(“./output/output.txt_”+a) as fw:\n        # 将当前图片名称和当前图片平均检测精度组合成[图片名称,图片平均检测精度]的形式写入txt文件\n        fw.write(“{},{},{}\\n”.format(jpg_name,jpg_average_precision/len(dict1[jpg_name])))\n\nprint(“[图片名称,图片平均检测精度]已写入txt文件!”)
```

最后会在 pytorch-retinanet\\output 文件夹中生成一个以当前时间命名的 txt 文件, 文件内即保存了所有 [图片名称,IOU 值] 信息。

7 动态调度网络

7.1 动态调度网络的思想

动态调度网络实际上是一个简单的 CNN 网络，这里以 LeNet 作为动态调度网络。动态调度网络的输入是一张图片，输出是目标检测网络的标号。所以，动态调度网络决定了输入的图片进入哪个目标检测网络进行处理。

另外，根据以上目标检测网络 retinanet 项目的说明，最终可以得到所有的[图片名称,IOU 值] 信息，这些信息都保存到了 txt 文件中。以此，多个目标检测网络，可以得到多个目标检测网络最终输出的 txt 文件，则一张图片会对应多个 IOU 值，该图片为动态调度网络的输入，取该张图片最大的 IOU 值所对应的目标检测网络标号最为标签。以此构造动态调度网络的数据集，进而对动态调度网络进行训练。

7.2 目前动态调度网络项目使用说明

项目存放在 PreNet 文件夹内。动态调度网络为 LeNet，在 PreNet/Model.py 文件内。目前的目标检测网络有 yolo5、FasterRCNN、retinanet，在目前的动态调度网络项目中，存放三种目标检测网络输出文件的目录为：PreNet/yolo5_new_label、PreNet/ fasterrcnn_output.txt、PreNet/retinanet_output.txt。该项目的运行需要结合上述第 6 章的 retinanet 项目，具体的使用流程如下：

第一步，在 PreNet/image/train 文件夹内创建三个文件夹 fasterrcnn、retinanet、yolo5 对应目前的三个目标检测网络。

第二步，运行 process01.py 文件对不同目标检测网络输出的文件进行处理，并且将 pytorch-retinanet\data\images 文件夹中的对应图片复制到 PreNet\images\train 的对应文件夹中从而制作动态调度网络的数据集。

第三步，运行 process02.py 文件进行 K 折交叉验证前的数据处理工作。

第四步，运行 main.py 文件从而对动态调度网络进行训练和验证。在该过程中会保存每折交叉验证中最好的模型，并且最终会生成 output.txt 文件以记录结果数据。

7.3 目前的动态调度网络项目训练效果

fold	train_loss	best_train_acc	best_test_acc
fold1:	1.0157211981713772	0.5702479338842975	0.5333333333333333
fold2:	1.010720755904913	0.5082644628099173	0.5333333333333333
fold3:	0.9974193871021271	0.5	0.5166666666666667
fold4:	1.0329743772745132	0.5289256198347108	0.5333333333333333
fold5:	1.048311773687601	0.45867768595041325	0.6833333333333333
=====			
avg in k fold:			
=====			
	train_loss	train_acc	test_acc
	1.0210294984281063	0.5132231404958677	0.5599999999999999

从目前的训练结果来看效果一般，可能原因有两个：

1. 数据问题，目前不同标签文件夹下的图片数据差异性过于小，导致动态调度网络学习效果较差。
2. 三个目标检测网络输出的 IOU 值计算方式目前是不同的，可能也会对动态调度网络的训练产生一定的影响。