### 5. 模糊逻辑（FL）+ 粒子群优化（PSO）组合模型案例题目

**题目：城市交叉口交通信号动态配时优化问题**

* **问题背景**：某城市主干道与次干道交叉口的早晚高峰拥堵严重，早高峰（7:30-9:00）平均等待时长达 15 分钟，晚高峰（17:30-19:00）因行人、非机动车混行导致通行效率下降 40%。当前信号配时为固定方案（如东西向绿灯 40 秒、南北向 30 秒），无法适应动态交通流。
* **问题描述**：需设计动态配时模型，根据实时交通流调整绿灯时长，目标包括：① 最小化车辆平均等待时间；② 最小化行人过街等待时间（≤60 秒）；③ 避免同一方向连续红灯超过 90 秒。输入信息含模糊性（如 “车流量较大”“行人较多”），需处理非机动车干扰等不确定性因素。
* **数据情况**：提供连续 7 天的交通流数据，包括：各方向每 5 分钟的车流量（辆）、行人过街数量（人）、非机动车流量（辆），以及不同配时方案下的实际等待时间记录。数据中含部分缺失值（如雨天传感器故障），需通过模糊处理方法兼容。

### 5. 模糊逻辑（FL）+ 粒子群优化（PSO）求解城市交叉口交通信号配时优化代码

|  |
| --- |
| import numpy as np  import random  import matplotlib.pyplot as plt  from sklearn.model\_selection import train\_test\_split  # 数据初始化与预处理  np.random.seed(42)  # 生成7天的交通流数据（每5分钟一条记录）  n\_samples = 7 \* 24 \* 12 # 7天×24小时×12个5分钟  # 输入特征：东西向车流量、南北向车流量、东西向行人数、南北向行人数  X = np.random.randint(5, 100, size=(n\_samples, 4)) # 流量范围5-100  # 输出标签：东西向绿灯时长、南北向绿灯时长（20-60秒）  y = np.zeros((n\_samples, 2))  for i in range(n\_samples):  # 简单规则生成标签（实际应用中可用历史数据）  y[i, 0] = 20 + X[i, 0] \* 0.4 + X[i, 2] \* 0.2 # 东西向绿灯时长  y[i, 1] = 20 + X[i, 1] \* 0.4 + X[i, 3] \* 0.2 # 南北向绿灯时长  y = np.clip(y, 20, 60) # 限制在20-60秒  # 划分训练集和测试集  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2)  # 模糊逻辑模块  class FuzzyLogic:  def \_\_init\_\_(self, n\_inputs=4, n\_outputs=2):  self.n\_inputs = n\_inputs  self.n\_outputs = n\_outputs  # 初始化隶属度函数参数（三角形：中心值和半宽度）  # 输入隶属度：少、中、多  self.input\_centers = np.array([[20, 50, 80]] \* n\_inputs) # 每个输入的3个中心值  self.input\_widths = np.array([[15, 15, 15]] \* n\_inputs) # 每个输入的3个半宽度  # 输出隶属度：短、中、长  self.output\_centers = np.array([[30, 40, 50]] \* n\_outputs) # 每个输出的3个中心值  # 三角形隶属度函数  def membership(self, x, centers, widths):  mu = np.zeros((len(x), len(centers)))  for i in range(len(centers)):  mu[:, i] = np.maximum(0, 1 - np.abs(x - centers[i]) / widths[i])  return mu  # 模糊推理  def infer(self, X):  n\_samples = X.shape[0]  outputs = np.zeros((n\_samples, self.n\_outputs))    for s in range(n\_samples):  # 计算输入隶属度  input\_mu = []  for i in range(self.n\_inputs):  mu = self.membership(X[s, i], self.input\_centers[i], self.input\_widths[i])  input\_mu.append(mu) # 每个输入的3个隶属度    # 组合所有规则（3^4=81条规则）  rule\_strengths = []  rule\_outputs = []  for a in range(3): # 输入1的隶属度索引  for b in range(3): # 输入2的隶属度索引  for c in range(3): # 输入3的隶属度索引  for d in range(3): # 输入4的隶属度索引  # 规则强度：取输入隶属度的最小值  strength = min(input\_mu[0][a], input\_mu[1][b],  input\_mu[2][c], input\_mu[3][d])  rule\_strengths.append(strength)  # 规则输出：取输出中心值（简单规则：输入隶属度索引平均值）  out = []  for o in range(self.n\_outputs):  out\_idx = int((a + b + c + d) / 4) # 简化规则  out.append(self.output\_centers[o, out\_idx])  rule\_outputs.append(out)    # 加权平均计算最终输出  total\_strength = sum(rule\_strengths)  if total\_strength == 0:  outputs[s] = np.mean(self.output\_centers, axis=1) # 默认值  else:  for o in range(self.n\_outputs):  weighted\_sum = sum([rs \* ro[o] for rs, ro in zip(rule\_strengths, rule\_outputs)])  outputs[s, o] = weighted\_sum / total\_strength    return np.clip(outputs, 20, 60) # 确保在有效范围内  # 将参数转换为粒子（用于PSO优化）  def params\_to\_particle(self):  particle = []  # 输入中心值  particle.extend(self.input\_centers.flatten())  # 输入半宽度  particle.extend(self.input\_widths.flatten())  # 输出中心值  particle.extend(self.output\_centers.flatten())  return np.array(particle)  # 从粒子更新参数  def particle\_to\_params(self, particle):  ptr = 0  # 输入中心值  self.input\_centers = particle[ptr:ptr+self.n\_inputs\*3].reshape(self.n\_inputs, 3)  ptr += self.n\_inputs\*3  # 输入半宽度  self.input\_widths = particle[ptr:ptr+self.n\_inputs\*3].reshape(self.n\_inputs, 3)  ptr += self.n\_inputs\*3  # 输出中心值  self.output\_centers = particle[ptr:ptr+self.n\_outputs\*3].reshape(self.n\_outputs, 3)  # PSO优化模块  class PSOOptimizer:  def \_\_init\_\_(self, fuzzy\_system, X\_train, y\_train, n\_particles=30, max\_iter=50):  self.fuzzy = fuzzy\_system  self.X = X\_train  self.y = y\_train  self.n\_particles = n\_particles  self.max\_iter = max\_iter  # 粒子维度：输入中心(4×3) + 输入宽度(4×3) + 输出中心(2×3) = 4×3×2 + 2×3 = 30  self.dim = 4\*3 + 4\*3 + 2\*3    # 初始化粒子和速度  self.global\_best\_particle = None  self.global\_best\_score = float('inf')  self.initialize\_particles()  # 初始化粒子  def initialize\_particles(self):  # 参数范围  input\_center\_min, input\_center\_max = 10, 90  input\_width\_min, input\_width\_max = 5, 25  output\_center\_min, output\_center\_max = 20, 60    self.particles = []  self.velocities = []  self.pbest\_particles = []  self.pbest\_scores = []    for \_ in range(self.n\_particles):  # 随机初始化粒子  particle = []  # 输入中心值  particle.extend(np.random.uniform(input\_center\_min, input\_center\_max, 4\*3))  # 输入半宽度  particle.extend(np.random.uniform(input\_width\_min, input\_width\_max, 4\*3))  # 输出中心值  particle.extend(np.random.uniform(output\_center\_min, output\_center\_max, 2\*3))  particle = np.array(particle)    # 初始化速度  velocity = np.random.uniform(-1, 1, self.dim)    # 计算初始适应度  self.fuzzy.particle\_to\_params(particle)  y\_pred = self.fuzzy.infer(self.X)  score = self.fitness\_score(y\_pred, self.y)    self.particles.append(particle)  self.velocities.append(velocity)  self.pbest\_particles.append(particle.copy())  self.pbest\_scores.append(score)    # 更新全局最优  if score < self.global\_best\_score:  self.global\_best\_score = score  self.global\_best\_particle = particle.copy()  # 适应度函数（最小化预测误差和等待时间）  def fitness\_score(self, y\_pred, y\_true):  # 预测误差（MSE）  mse = np.mean((y\_pred - y\_true)\*\*2)    # 惩罚项：行人等待时间过长（>60秒）  pedestrian\_wait = np.maximum(0, (y\_pred[:,0] + y\_pred[:,1]) - 60)  ped\_penalty = np.mean(pedestrian\_wait) \* 5    # 惩罚项：同一方向连续红灯过长（>90秒）  red\_light = 90 - y\_pred # 假设周期为90秒  red\_penalty = np.mean(np.maximum(0, red\_light)) \* 3    return mse + ped\_penalty + red\_penalty  # 执行PSO优化  def optimize(self):  w = 0.8 # 惯性权重  c1, c2 = 0.5, 0.5 # 学习因子    for iter in range(self.max\_iter):  for i in range(self.n\_particles):  # 更新速度  r1, r2 = np.random.rand(self.dim), np.random.rand(self.dim)  cognitive = c1 \* r1 \* (self.pbest\_particles[i] - self.particles[i])  social = c2 \* r2 \* (self.global\_best\_particle - self.particles[i])  self.velocities[i] = w \* self.velocities[i] + cognitive + social    # 更新位置  self.particles[i] += self.velocities[i]    # 约束处理（确保参数在合理范围）  self.particles[i][:12] = np.clip(self.particles[i][:12], 10, 90) # 输入中心  self.particles[i][12:24] = np.clip(self.particles[i][12:24], 5, 25) # 输入宽度  self.particles[i][24:] = np.clip(self.particles[i][24:], 20, 60) # 输出中心    # 计算适应度  self.fuzzy.particle\_to\_params(self.particles[i])  y\_pred = self.fuzzy.infer(self.X)  score = self.fitness\_score(y\_pred, self.y)    # 更新个体最优  if score < self.pbest\_scores[i]:  self.pbest\_particles[i] = self.particles[i].copy()  self.pbest\_scores[i] = score    # 更新全局最优  if score < self.global\_best\_score:  self.global\_best\_score = score  self.global\_best\_particle = self.particles[i].copy()    if iter % 5 == 0:  print(f"迭代 {iter}, 最优适应度: {self.global\_best\_score:.4f}")    # 应用最优参数  self.fuzzy.particle\_to\_params(self.global\_best\_particle)  return self.fuzzy  # 主函数  if \_\_name\_\_ == "\_\_main\_\_":  # 初始化模糊逻辑系统  fuzzy = FuzzyLogic()    # 用PSO优化模糊系统参数  pso = PSOOptimizer(fuzzy, X\_train, y\_train)  optimized\_fuzzy = pso.optimize()    # 测试优化效果  y\_pred = optimized\_fuzzy.infer(X\_test)  test\_mse = np.mean((y\_pred - y\_test)\*\* 2)  print(f"测试集MSE: {test\_mse:.4f}")    # 对比优化前后的结果（随机选10个样本）  sample\_idx = np.random.choice(len(X\_test), 10)  X\_sample = X\_test[sample\_idx]  y\_true\_sample = y\_test[sample\_idx]    # 优化前的模糊系统  original\_fuzzy = FuzzyLogic()  y\_pred\_original = original\_fuzzy.infer(X\_sample)  # 优化后的模糊系统  y\_pred\_optimized = optimized\_fuzzy.infer(X\_sample)    # 可视化对比  plt.figure(figsize=(12, 6))  x\_ticks = np.arange(10)    # 东西向绿灯时长对比  plt.subplot(1, 2, 1)  plt.bar(x\_ticks - 0.2, y\_true\_sample[:,0], 0.4, label='实际值')  plt.bar(x\_ticks + 0.2, y\_pred\_original[:,0], 0.4, label='优化前预测')  plt.bar(x\_ticks, y\_pred\_optimized[:,0], 0.2, label='优化后预测', color='green')  plt.title('东西向绿灯时长对比')  plt.legend()    # 南北向绿灯时长对比  plt.subplot(1, 2, 2)  plt.bar(x\_ticks - 0.2, y\_true\_sample[:,1], 0.4, label='实际值')  plt.bar(x\_ticks + 0.2, y\_pred\_original[:,1], 0.4, label='优化前预测')  plt.bar(x\_ticks, y\_pred\_optimized[:,1], 0.2, label='优化后预测', color='green')  plt.title('南北向绿灯时长对比')  plt.legend()    plt.tight\_layout()  plt.show() |