作业3实验报告

231240002余孟凡 231240002@smail.nju.edu.cn
南京大学计算机科学与技术系, 南京 210093

摘要

此任务的框架代码非常简单，整个任务以设计游戏策略，选取训练方法和改善特征提取方法为主。

关键词: 随机森林方法, KNN算法, CatBoost算法, MLP网络, LSTM方法

1 准备工作

阅读框架代码，做一些小的改善：

1.1 自动读取

一个一个打文件夹目录真的是太烦人了，我们还是写一个函数，自动查找当天生成的训练数据吧。

```python
def load_game_records(date_prefix='2024-11-16'):  1 个用法

    data_list = glob.glob(os.path.join('logs', f'game_records_lvl0_{date_prefix}*'))
    all_data = []

    for data_path in data_list:
        data_file = os.path.join(data_path, 'data.pkl')
        try:
            with open(data_file, 'rb') as f:
                all_data.extend(pickle.load(f))
            print(f"加载 {data_file} 成功")
        except Exception as e:
            print(f"加载 {data_file} 时出错: {e}")

    return all_data
```

图 1: 自动匹配

可是在尝试运行之后，发现完全没有任何信息被记录下来，尝试打印每一行的信息，发现是这种脚本的输入输出流不支持中文，那我们就把调试信息改成英文的吧。

1.2 调试输出的缺失

整个main函数只有一句模型训练完成，其他别的啥我都不知道，这不太行，我们必须获得模型的更多信息，才能更好的改善模型：因此我们有必要划分训练集和测试集，并对一些关键数据进行打印输出。

图 2: 调试打印和划分

1.3 模型基本内容可视化

使用 DataFrame 更方便地处理和查看数据，结合 describe() 和 info() 方法，能够清楚地了解数据的基本信息和统计特征。

```python
# 将数据转换为 DataFrame，便于后续处理
feature_names = [f'feature_{i}' for i in range(X.shape[1])]
df = pd.DataFrame(X, columns=feature_names)
df['action'] = y

# 数据检查
print("\n数据集基本信息:")
print(df.info())
print(df.describe())

# 检查缺失值
if df.isnull().sum().any():
    df = df.dropna()
    print("发现并移除了缺失值")
```

图 3: 数据集可视化

1.3.1 数据的分类

我们将数据分为训练集和测试集：训练集8测试集2，至于为什么不要验证集，先一步一步来呗。

1.3.2 打印关键数据

我们取总样本数，特征维度，类型分布进行打印，并在最后输出模型准确率。

1.4 更多的基础设施

由于我们需要对模型的指标进行评测，而每一次运行的结果具有随机性，因此我们应该写一个自动化的脚本，使其能够不断运行程序并记录最后结果：

```python
def run_and_log():  1 个用法  新 *
    log_file_path = 'game_results_RandomForest.log'  # 日志文件的路径

    while True:
        # 执行原程序并捕获输出
        process = subprocess.Popen(
            args: ['python', 'test.py'],
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
            text=True
        )

        # 等待程序完成并获取输出
        stdout, stderr = process.communicate()

        # 检查是否有错误输出
        if stderr:
            print(f"错误: {stderr}")

            # 查找最后一行输出
        last_line = ''
        for line in stdout.splitlines():
            if "信息:" in line:  # 找到包含"信息:"的行
                last_line = line

                # 将最后一行写入日志文件
        with open(log_file_path, 'a') as log_file:
            log_file.write(last_line + '\n')

        print(f"记录到日志: {last_line}")

        # 等待一段时间后再执行
        time.sleep(5)
```

图 4: 数据集可视化

2 随机森林方法

2.1 随机森林算法介绍

要讨论随机森林算法，我们必须先回到决策树算法上来：

2.1.1 决策树算法

决策树算法通过将数据分割成更小的子集，形成树形结构来进行决策。每个内部节点代表一个特征（或属性），每个分支代表特征的一个取值，而每个叶子节点则代表最终的决策结果。

选择特征：使用算法（如信息增益、基尼指数或均方误差）来选择分裂特征。

分裂节点：根据选择的特征将数据集分成多个子集。

递归处理：对每个子集重复上述过程，直到满足停止条件（如达到最大深度、节点样本量不足等）。

2.1.2 随机森林算法

随机森林首先需要随机进行有放回抽样（同一样本可能不止被抽取一次）（Bootstrapping）。在每个节点分裂时，随机选择特定数量的特征，而不是使用全部特征。这有助于减少模型之间的相关性，提高模型的泛化能力。对于分类任务，随机森林通过所有树的投票结果来决定最终的分类结果。

2.2 初步训练

我先随便玩了七八把，有一半赢了一半死了。拿现在这个数据集去训练：绘制混淆矩阵：

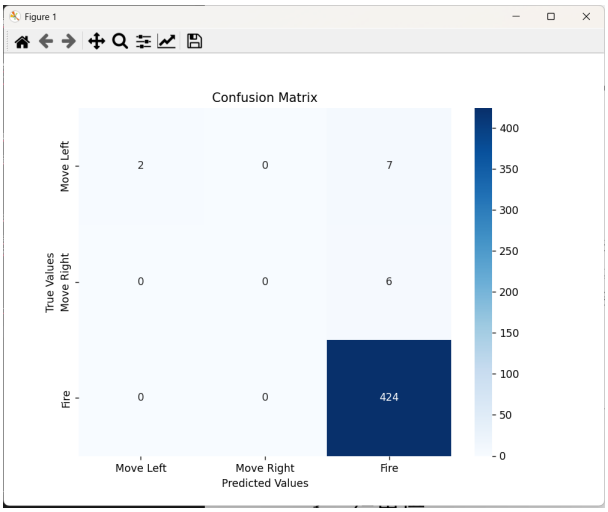

图 5: 混淆矩阵

```
数据集信息:
总样本数: 2195
特征维度: 4032
类别分布: (array([1, 2, 3]), array([  43,   45, 2107], dtype=int64))

训练集大小: 1756
测试集大小: 439

开始训练随机森林分类器...

模型性能:
准确率: 97.04%

分类报告:
              precision    recall  f1-score   support

           1       1.00      0.22      0.36         9
           2       0.00      0.00      0.00         6
           3       0.97      1.00      0.98       424

    accuracy                           0.97       439
   macro avg       0.66      0.41      0.45       439
weighted avg       0.96      0.97      0.96       439
```

图 6: 模型基础信息

可以发现策略基本就是火力全开，几乎百分百选择射击。这是由于第一关，活力全开凭运气是可以过的，在我自己玩的过程中，几乎只有一两次子弹朝我打来，如此小的数量不会被学习到有情可原。

我们现在把关卡切换到4，这一关明显感觉敌人会有意识的在你头上发射，因此需要你更积极的去运动，看看通过更积极的运动策略，能否被模型学习到：

学习效果更差了，在更积极运动的策略下，仍然还是只会火力全开，分析这种情况，可能是因为样本数量过少，欠拟合了，也有可能是因为数据特征选择的问题。我

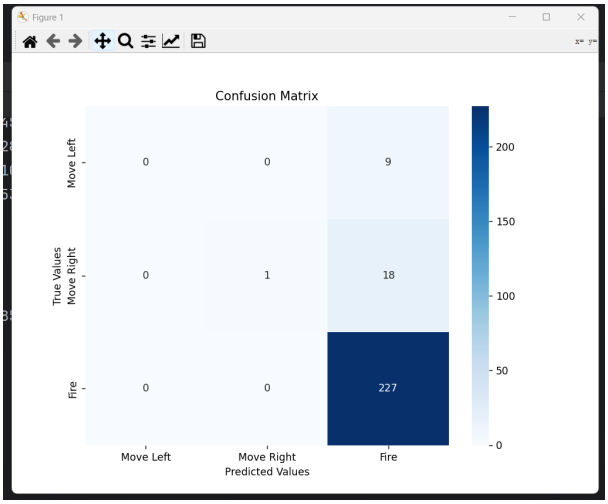

图 7: 混淆矩阵

```
数据集信息:
总样本数: 1274
特征维度: 4032
类别分布: (array([1, 2, 3]), array([  32,   85, 1157], dtype=int64))

训练集大小: 1019
测试集大小: 255

开始训练随机森林分类器...

模型性能:
准确率: 89.41%

分类报告:
              precision    recall  f1-score   support

           1       0.00      0.00      0.00         9
           2       1.00      0.05      0.10        19
           3       0.89      1.00      0.94       227

    accuracy                           0.89       255
   macro avg       0.63      0.35      0.35       255
weighted avg       0.87      0.89      0.85       255
```

图 8: 模型基础信息

们先来尝试改进这个算法：

2.3 改善思路及代码

2.3.1 前期数据处理

首先我们需要对数据进行标准化处理：标准化可以确保不同特征的尺度一致，避免某些特征在模型训练中因数值范围较大而主导其他特征的影响。尽管对于随机森林而言，特征尺度的影响相对较小，但进行标准化仍然是一个良好的改进策略。

然后我们要处理最关键的类别不平衡了。SOMTE算法专为处理这一问题而生：SMOTE 的基本思路是基于现有的少数类样本生成新的样本。具体步骤如下：从少数类中随机选择一个样本。用距离度量（通常是欧几里得距离）找到该样本的 k 个最近邻居。这些邻居同样属于少数类。对于每个邻居，SMOTE 使用以下公式生成新样本：

$$\text{new\_sample} = \text{sample} + \lambda \times (\text{neighbor} - \text{sample})$$

其中，λ 是一个在 $[0, 1]$ 区间内的随机数。这个公式可以理解为在样本与邻居之间进行线性插值。

根据需要，重复以上步骤，直至达到所需的少数类样本数量。

```python
# 特征缩放
scaler = StandardScaler()
X_scaled = scaler.fit_transform(df[feature_names])

# 处理类别不平衡
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X_scaled, df['action'])
```

图 9: 代码实现

2.3.2 超参数优化

进行模型的超参数调优，通过网格搜索来找到随机森林分类器的最佳参数组合。

```python
# 定义参数网格进行调优
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 20],
    'max_features': ['log2', 'sqrt'],
    'min_samples_split': [2, 5, 10]
}

# 使用网格搜索和交叉验证
rf = RandomForestClassifier(random_state=42)
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid,
                           cv=5, n_jobs=-1, verbose=2)
grid_search.fit(X_train, y_train)
```

图 10: 代码实现

n_estimators: 随机森林中树的数量。可以尝试 100、200 和 300 棵树。

max_depth: 控制树的深度，可以设置为 None（表示节点分裂直到叶子节点），或者限制深度为 10 或 20。

max_features: 在每次分裂时考虑的特征数量。可以选择的值包括 'log2'（对数基数），或者 'sqrt'（特征数量的平方根）。

min_samples_split: 一个节点需要的最小样本数，才能进行分裂。可以设定为 2、5 或 10。

2.4 改进后训练

进行训练：

可以看到训练准确度明显回升，三种操作的f1数据指标良好。

```
最佳参数: {'max_depth': None, 'max_features': 'log2', 'min_samples_split': 10, 'n_estimators': 100}

交叉验证平均准确率: 97.08%

测试集准确率: 96.83%

分类报告:
              precision    recall  f1-score   support

           1       1.00      0.97      0.98       215
           2       1.00      0.94      0.97       230
           3       0.92      1.00      0.96       250

    accuracy                           0.97       695
   macro avg       0.97      0.97      0.97       695
weighted avg       0.97      0.97      0.97       695
```

图 11: 训练结果

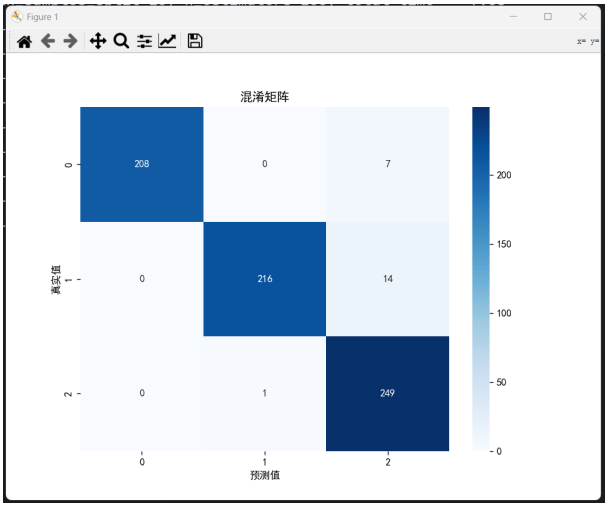

图 12: 混淆矩阵

可以看到，SMOTE操作在其中起到了非常关键的作用，能够有效扩大左移右移样本的数量，进行有效的训练。

进行实战测试，发现了严重问题，现在的决策几乎以2为主，可是2为向右动，没有办法杀怪且会给自己卡死。

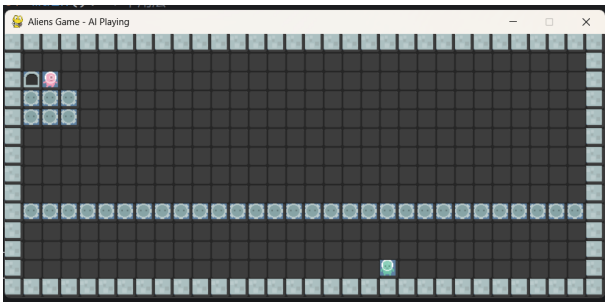

图 13: 积极运动

2.5 问题在哪里？

我比较怀疑的一点是在SOMTE方法生成新样本时，这些样本的处理有问题：由于左移和右移我都是在快要落到我这一个才进行的，但是SOMTE生成的新样本很有可能

产生偏移，比如在λ为0.5，平均参考两个样本的意见，可是这个时候新样本位置的头顶很有可能没有子弹，而此类样本的极速扩大导致了训练数据不可避免的走向失真。

我们删除SOMTE处理再次进行训练：

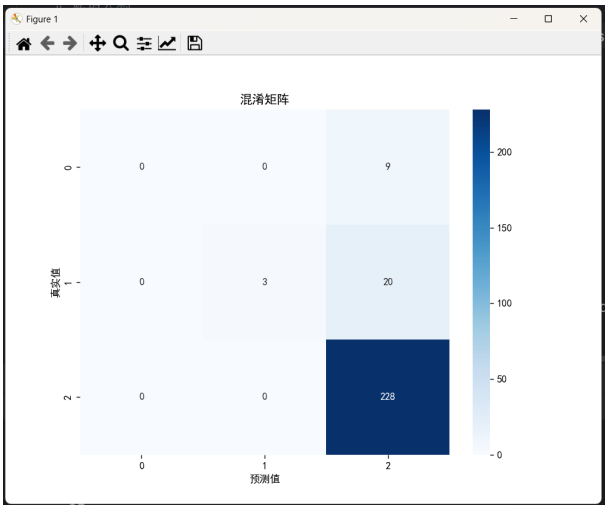

图 14: 训练结果

（到这里我才意识到大量样本的重要性写了自动调试脚本）

我们来评测一下结果：这次我们测试20组，看看结果如何：

```
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 28
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 12
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 20
Info: {}, Score: 31
Info: {}, Score: 31
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 10
Info: {}, Score: 31
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 30
Info: {}, Score: 31
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 24
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 28
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 10
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 14
Info: {}, Score: 31
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 18
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 28
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 12
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 26
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 8
Info: {}, Score: 31
Info: {}, Score: 31
```

图 15: 训练效果

可以看到，效果相比一直往右不打子弹有了很大提升，有35%的几率存活到最后，并且每次都击杀数平均在20以上，可以说不错了。

3 KNN算法

3.1 KNN算法介绍

KNN算法的核心思想是：给定一个待分类的样本，通过计算该样本与训练集中所有样本的距离，找出距离最近的K个邻居，然后根据这K个邻居的类别进行投票，确定待分类样本的类别。

主要步骤：

选择参数K：选择邻居的数量K。计算距离：对待分类样本与训练集中的每个样本计算距离。

选择邻居：找出距离最近的K个邻居。

投票：根据这K个邻居的类别进行投票。

与其他模型不同，KNN不需要建立显式的训练模型。

不足之处包括：

对于大型数据集，距离计算和存储的成本较高。对局部噪声和离群点比较敏感，可能影响分类结果。

特征的尺度对KNN的性能有重大影响，因此在使用KNN之前，通常需要对数据进行标准化（如Z-score标准化）或归一化（将特征值缩放到0到1之间）。

3.2 初步训练

吸取经验教训，我们先不使用SOMTE办法训练，同时扩大超参数网格，尽力找到最好的参数。

```
X_train, X_test, y_train, y_test = train_test_split(*arrays: X_scaled, y, test_size=0.2, random_state=42)
# 定义参数网格进行调优
param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11, 15, 21, 31, 51],  # 扩大邻居数范围
    'weights': ['uniform', 'distance'],
    'metric': [
        'euclidean',  # 欧氏距离
        'manhattan',  # 曼哈顿距离
        'chebyshev',  # 切比雪夫距离
        'cosine',  # 余弦距离
        'correlation'  # 相关系数距离
    ],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],  # 不同的近邻搜索算法
    'p': [1, 2, 3]  # Minkowski距离参数，1为曼哈顿，2为欧氏距离
}

# 使用网格搜索和交叉验证
knn = KNeighborsClassifier()
# 调整网格搜索参数
grid_search = GridSearchCV(
    estimator=knn,
    param_grid=param_grid,
    cv=5,  # 5折交叉验证
    n_jobs=-1,  # 并行计算
    verbose=2,  # 详细输出
    scoring='accuracy'  # 评分标准
)
grid_search.fit(X_train, y_train)
```

图 16: 训练代码

训练完成输出混淆矩阵：

这个结果毫不意外，这并不是KNN算法导致的问题，而是本身样本数据不平衡导致的。我们现在需要重启SOMTE，看看KNN对SOMTE的适配度如何：

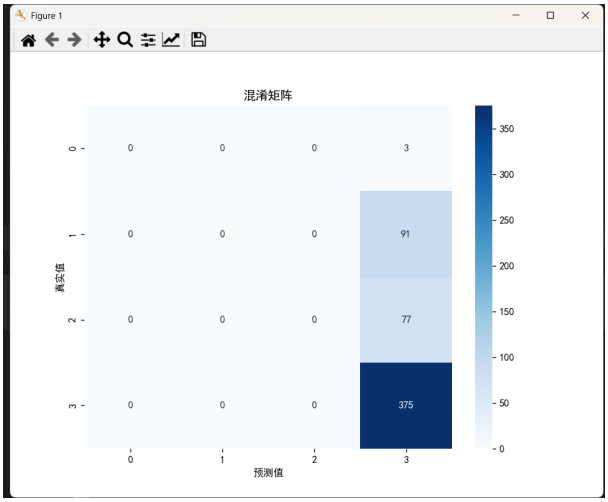

图 17: 混淆矩阵

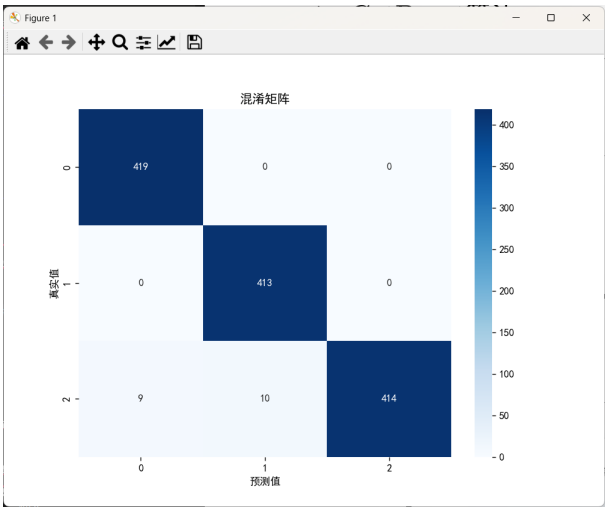

图 18: SOMTE混淆矩阵

进行实战测试，此时变为了左右随机抽搐，但仍然没有射击，可以确定SMOTE完全不实用于此任务，需要其他方法来平衡数据分类。

4 改变数据集

我们用第三关作为测试样例：这一关没有任何掩体，迫使我们积极进行运动，实现手动的数据平衡。

4.1 再次训练

可以看到，算法最终的准确率在70左右，是不太令人满意的。推测其原因，可能是“邻居”这一概念上我们最终调参得出了曼哈顿距离，然而在实际游戏环境中下落的子弹即使只差一个格子，那么也会直接影响本轮是选择射击还是移动。然而这一概念在KNN中被抹平了，变为了与怪物位置同样概念的东西。

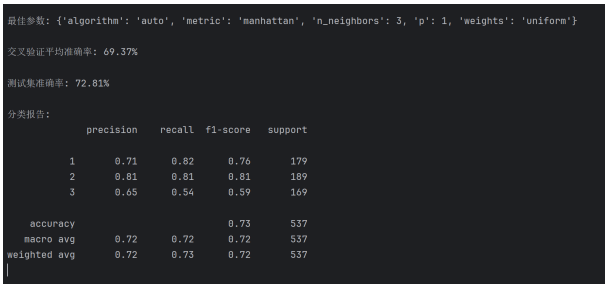

图 19: KNN算法

5 CatBoost算法

再了解CatBoost算法之前，我们应该先了解它的基础：梯度提升算法。

5.1 梯度提升算法

梯度提升（Gradient Boosting）是一种强大的机器学习集成算法，通过构建多个弱学习器（通常是决策树）来创建一个强大的预测模型。它的核心思想是逐步减小残差，不断改进模型的预测能力。

梯度提升算法的迭代学习过程包括以下步骤：

1. 初始化模型（通常为常数预测）
2. 计算当前模型的残差（预测值与真实值的差）
3. 构建新的弱学习器来拟合残差
4. 更新模型，降低整体误差
5. 重复步骤2-4，直到达到预设条件

5.1.1 数学形式化

对于第t个模型，我们定义：

- $F_{t-1}(x)$：前一轮的累积模型
- $h(x)$：新的弱学习器
- γ：学习率（步长）

新模型表示为：

$$F_t(x) = F_{t-1}(x) + \gamma \cdot h(x) \tag{1}$$

5.1.2 残差计算

使用负梯度作为残差的近似：

$$r_i = -\left[\frac{\partial L(y, F(x))}{\partial F(x)}\right] \tag{2}$$

传统决策树存在特征选择偏差，CatBoost提出对称树算法，减少过拟合风险，提高模型泛化能力。

梯度提升过程：随机梯度排序，动态调整样本权重，减少模型对训练数据的敏感性，提高模型的稳定性，自动学习率调整，根据模型训练进程动态调整学习率，避免手动调参的复杂性。

5.2 初步训练

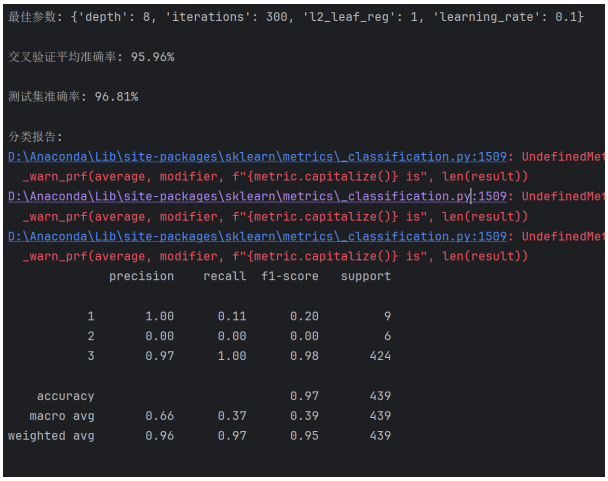

图 20: 混淆矩阵

可以看到效果仍然不理想，准备换用高级学习方法。

6 MLP网络

MLP神经网络属于前馈神经网络（Feedforward Neural Network）的一种。在网络训练过程中，需要通过反向传播算法计算梯度，将误差从输出层反向传播回输入层，用于更新网络参数。（引自https://blog.csdn.net/liaomin416100569/article/details/130572559）

输入层：接收模型的输入数据，节点数与特征数一致。

隐藏层：一个或多个层，从前层接收输入，通过神经元处理消息并传递给后续层。每个神经元通过激活函数引入非线性。

输出层：生成最终输出，节点数对应于预测的类别.

6.1 配置CUDA环境

由于这样的学习需要消耗大量资源,等待时间会很长。

但是你怎么知道我有 **NVIDIA GoForce RTX 3090 Laptop GPU?**

安装CUDA以及cuDNN，并创建虚拟conda环境与其关联，我们就获得了GPU加速计算的能力：

```
# 检查是否有可用的 GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"使用设备: {device}")
model.to(device)  # 将模型移动到 GPU
```

图 21: CUDA

6.2 MLP代码实现

我们首先需要建立模型：

```
# 定义神经网络模型
class ImprovedNN(nn.Module): 2 用法
    def __init__(self, input_size, num_classes):
        super(ImprovedNN, self).__init__()
        self.fc1 = nn.Linear(input_size, out_features: 256)
        self.bn1 = nn.BatchNorm1d(256)  # 添加 Batch Normalization
        self.fc2 = nn.Linear( in_features: 256, out_features: 128)
        self.bn2 = nn.BatchNorm1d(128)  # 添加 Batch Normalization
        self.fc3 = nn.Linear( in_features: 128, num_classes)

    def forward(self, x):
        x = torch.relu(self.bn1(self.fc1(x)))
        x = torch.relu(self.bn2(self.fc2(x)))
        x = self.fc3(x)  # 输出层
        return x
```

图 22: 网络实现

在这个网络中，其结构主要包含三层全连接层，并在前两层之后加上了批量归一化和 ReLU 激活函数。

接下来转换为torch类型，准备训练：

```
# 转换为 PyTorch 张量
X_tensor = torch.FloatTensor(X)
y_tensor = torch.LongTensor(y)

# 创建数据集和数据加载器
dataset = TensorDataset( *tensors: X_tensor, y_tensor)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)

input_size = X.shape[1]  # 输入特征数量

model = ImprovedNN(input_size, num_classes)
```

图 23: 参数传入

采用交叉熵损失作为损失函数和Adam优化器进行训练：

```python
# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()  # 适用于多分类
optimizer = optim.Adam(model.parameters(), lr=0.001)

epochs = 100

print("开始训练模型...")
for epoch in range(epochs):
    for batch_X, batch_y in dataloader:
        batch_X, batch_y = batch_X.to(device), batch_y.to(device)  # 将数据移动到 GPU

        optimizer.zero_grad()  # 清零梯度
        outputs = model(batch_X)  # 前向传播
        loss = criterion(outputs, batch_y)  # 计算损失
        loss.backward()  # 反向传播
        optimizer.step()  # 更新参数

    print(f'Epoch [{epoch + 1}/{epochs}], Loss: {loss.item():.4f}')

env = AliensEnvPygame(level=3, render=False)
```

图 24: 训练过程

我们做一点小小的改善，我们通过每轮增加20次训练步数的方法，看看到底epochs在多少合适。

```
当前训练轮数为：20
混淆矩阵:
[[196   1 125]
 [  0 172  96]
 [ 10   5 880]]
准确率: 84.04%
模型已保存到 logs/game_r
开始训练模型...
当前训练轮数为：40
混淆矩阵:
[[319   0   3]
 [  0 170  98]
 [130   1 764]]
准确率: 84.38%
模型已保存到 logs/game_r
开始训练模型...
当前训练轮数为：60
混淆矩阵:
[[191   0 131]
 [  1 266   1]
 [  2  97 796]]
准确率: 84.38%
模型已保存到 logs/game_r
开始训练模型...
当前训练轮数为：80
混淆矩阵:
[[190   1 131]
 [  0 171  97]
 [  1   3 891]]
```

图 25: 训练结果

可以发现最终准确率在84.58%处收敛。这个结果我们可以测试一下跑分：

可以看到这个网络的存活率已经达到了55%，相较于传统学习方法又有了大的提升。

```
当前训练轮数为：240
混淆矩阵：
[[321   0   1]
 [  0 265   3]
 [130  95 670]]
准确率：84.58%
模型已保存到 logs/game_
开始训练模型...
当前训练轮数为：260
混淆矩阵：
[[245   0  77]
 [  0 202  66]
 [ 54  32 809]]
准确率：84.58%
模型已保存到 logs/game_
开始训练模型...
当前训练轮数为：280
混淆矩阵：
[[191   0 131]
 [  0 170  98]
 [  0   0 895]]
准确率：84.58%
```

图 26: 训练结果

```
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 13
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 7
Info: {}, Score: 28
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 25
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 5
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 11
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 7
Info: {}, Score: 28
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 3
Info: {}, Score: 28
Info: {}, Score: 28
Info: {}, Score: 28
Info: {}, Score: 28
Info: {}, Score: 28
Info: {}, Score: 28
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 13
Info: {}, Score: 28
Info: {'message': 'Avatar destroyed. You lose.'}, Score: 23
Info: {}, Score: 28
Info: {}, Score: 28
```

图 27: 跑分结果

7 整理代码仓库

现在主目录下的代码太多了，需要按类别进行整理：
其中涉及到一个文件相对目录和绝对目录的问题，使用已经写在了md文件里。

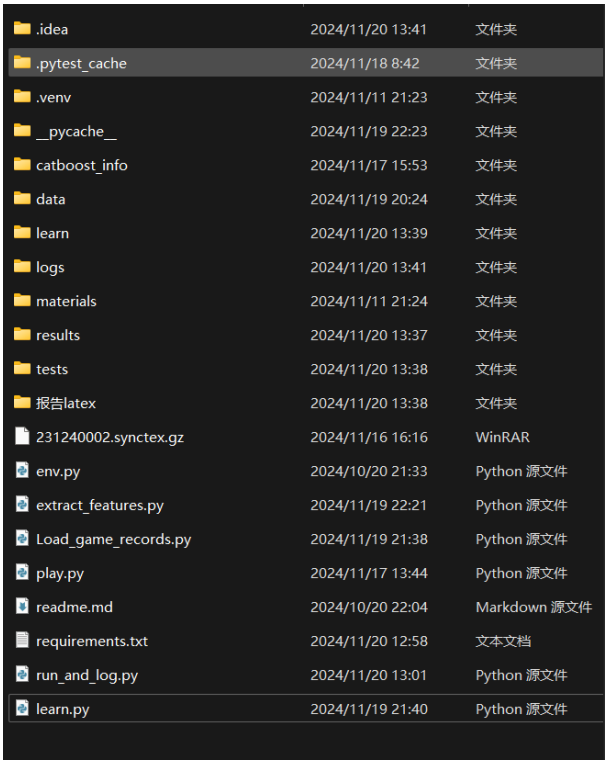

图 28: 整理文件

8 修改特征提取方式

8.1 观察记录文件

我们首先观察data[0]的形状：

图 29: data[0]的形状

这是一个元组的形式，其中第一个元素是描述当前状态的列表，第二个元素是进行的操作。

列表中的每一个元素代表一行的情况，而第三层则是每个格子中有什么（玩家，怪物，炸弹……）。

9 特征是什么

我们需要思考一个问题，在这个任务中，中间的格子究竟有无存在的必要？我们是否可以进行一定程度的特征压缩？在我看来：

1. 由于5个关卡内怪物都只是在上方活动，因此中间的格子对于怪物特征的提取并无必要。

2. 由于玩家只在最下方活动，因此对于玩家也并无意义。

3. 由于子弹打下来只要这一列有掩体，那么就会被挡住，至于在哪里挡住的无所谓。

4. 由于子弹从发射到下落的时间是固定的，那么只要发射的格子确定了，落在底下的格子也就确定了。

5. 而对于四周的墙，我不确定会不会有问题，但是理论上来讲删掉四周的墙，或者不进行特征提取，最多会导致撞墙，而玩家的行为中如果没有撞墙，自然也不会习得。

同时，高维数据可能会导致模型性能下降，因为在高维空间中样本会变得非常稀疏，难以找到有效的模式。并且可能导致过拟合。

那么现在我们可以利用这个思路来降维了：

可以看到我们成功把之前14*32*9 = 4032的维度降到了 32 * 5 = 160，相比之前的特征维度大大降低。

以上是我自己的理解，我又查阅了几篇文章

（https://cloud.tencent.com/developer/article/1882617）

（https://zhuanlan.zhihu.com/p/34450286）

（https://blog.csdn.net/nihaomabmt/article/details/102931744）

大致有这样一个观点：数据本身决定了问题的可解性和最优上限，即：对于给定的数据，是否存在某种理想模型可以达到更高准确率。

如果说我本人在同一或者相近情况下作出了不同的决策，比如三次同一分别做出了左右移动和射击各一次，那随机森林或者KNN算法就根本无法学习出我的操作，因为数据本身就没有可解性。所有这一次我重新玩游戏生成样本的时候，必须严格按照

```python
def extract_features_plus(observation):
    grid = observation

    object_mapping = {
        'wall': 1,
        'avatar': 2,
        'alien': 3,
        'bomb': 4,
        'base': 0
    }
    features = [[0, 0, 0, 0, 0] for _ in range(32)]
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            print(grid[i][j])
            for obj in grid[i][j]:
                index = object_mapping.get(obj, -1)
                if index >= 0:
                    features[j][index] = 1


    print(features)
```

图 30: 降低特征维度

```
[[0, 1, 0, 0, 0], [1, 1, 0, 0, 0], [1, 1, 0, 0, 0], [1, 1, 0, 0, 0], [0, 1, 0, 0, 0], [0, 1, 0, 0, 0],
 [0, 1, 0, 0, 0], [0, 1, 0, 0, 0], [0, 1, 0, 0, 0], [0, 1, 0, 0, 0], [0, 1, 0, 0, 0], [0, 1, 0, 0, 0],
 [0, 1, 0, 0, 0], [0, 1, 0, 0, 0], [0, 1, 0, 0, 0], [0, 1, 0, 0, 0], [0, 1, 0, 0, 0], [0, 1, 0, 0, 0],
 [0, 1, 0, 0, 0], [0, 1, 0, 0, 0], [0, 1, 0, 0, 0], [0, 1, 0, 0, 0], [0, 1, 0, 0, 0], [0, 1, 0, 0, 0],
 [0, 1, 0, 0, 0], [0, 1, 0, 0, 0], [0, 1, 0, 0, 0], [0, 1, 0, 0, 0], [0, 1, 0, 0, 0], [0, 1, 0, 0, 0],
 [0, 1, 1, 0, 0], [0, 1, 0, 0, 0]]
```

图 31: 降维之后的结果

我新特征提取的方式来玩。由于新特征提取方式不包含掩体，那么这次我将无视掩体射击，并只根据子弹的横向位置来进行躲避。

使用新样本进行下面的测试

9.1 使用随机森林方法测试

看一看之前的混淆矩阵：

```
混淆矩阵：
[[294   0  28]
 [  0 181  87]
 [110  16 769]]
准确率： 83.77%
```

图 32: 之前的结果

可以看到，在第一轮训练时，样本的不平衡掩盖了其对特征要求高的模型特征。因为我们的随机森林事实上是一些获取部分特征的决策树们构成，那么这个作为决策的特征必须更加突出。因此随机森林算法是我们由于检测特征提取是否合适的很好的算法。

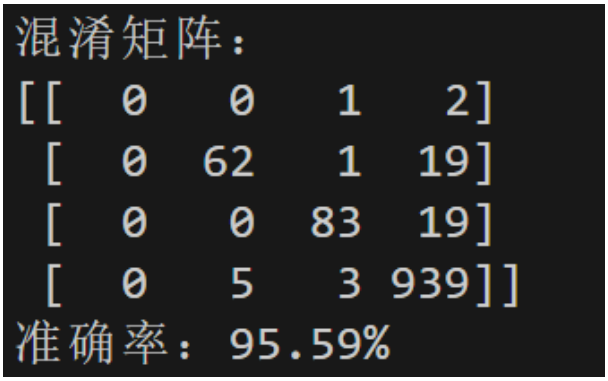

图 33: 更改特征提取之后的结果

令人震惊的提升！推测是由于更改后的特征方式更符合我的特点：只看怪物那一排，掩体可有可无。同时可以适当间隔射击，不必卡死最小射击间隔。

这里混淆矩阵从3*3变为了4*4，分类类别多了一个，推测是由于不卡死设计间隔之后，0（不运动）也进入了样本仓库。

继续进行测试：

10 使用KNN方法测试

改进前的KNN：

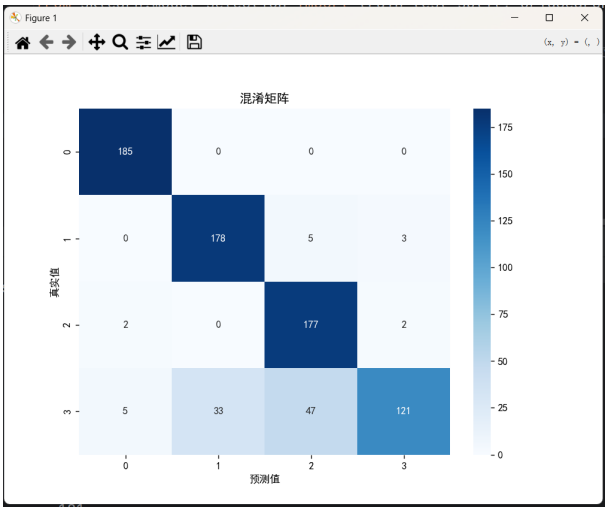

图 34: 更改特征提取之前的结果

改进体征提取后的KNN： 可以看到，改进后的KNN正确率也提高到了90%以上，提高了4个百分点。

```
最佳参数: {'metric': 'euclidean', 'n_neighbors': 5, 'p': 2, 'weights': 'uniform'}

测试集准确率: 87.20%

分类报告:
              precision    recall  f1-score   support

           0       0.96      1.00      0.98       185
           1       0.84      0.96      0.90       186
           2       0.77      0.98      0.86       181
           3       0.96      0.59      0.73       206

    accuracy                           0.87       758
   macro avg       0.89      0.88      0.87       758
weighted avg       0.89      0.87      0.86       758
```

图 35: 分类报告

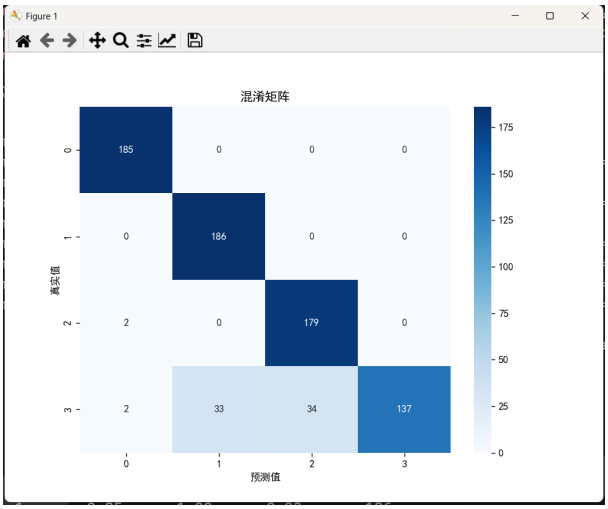

图 36: 更改特征提取之后的结果

```
最佳参数: {'metric': 'manhattan', 'n_neighbors': 3, 'p': 1, 'weights': 'distance'}

测试集准确率: 90.63%

分类报告:
              precision    recall  f1-score   support

           0       0.98      1.00      0.99       185
           1       0.85      1.00      0.92       186
           2       0.84      0.99      0.91       181
           3       1.00      0.67      0.80       206

    accuracy                           0.91       758
   macro avg       0.92      0.91      0.90       758
weighted avg       0.92      0.91      0.90       758
```

图 37: 分类报告

11 LSTM方法训练（两种特征提取方法）

11.1 RNN方法介绍

递归神经网络（Recurrent Neural Network, RNN）是一种特别适用于处理序列数据的神经网络架构，广泛应用于自然语言处理、语音识别、时间序列预测等领域。相较于传统的前馈神经网络，RNN能够通过循环连接来保留先前输入的信息，从而在处理输入序列时具有”记忆”能力。

利用RNN网络，我们把样本数量缩减至一次游戏，就能够让网络拥有记忆前序状态的能力。

11.2 LSTM方法介绍

长短期记忆（Long Short-Term Memory, LSTM）是一种特殊类型的递归神经网络（RNN），旨在解决标准RNN在处理长期依赖问题时所面临的挑战。

与传统的前馈神经网络和标准RNN不同，LSTM引入了三个重要的“门”机制：输入门、遗忘门和输出门。这些门的设计使LSTM能够动态地调整信息的流动，决定哪些信息需要保留、更新或丢弃，从而实现对长期依赖信息的有效记忆。

11.3 代码介绍

```python
class ImprovedLSTM(nn.Module):
    def __init__(self, input_size, num_classes, hidden_size=128, num_layers=2, dropout=0.5):
        super(ImprovedLSTM, self).__init__()
        # 添加批归一化层
        self.batch_norm = nn.BatchNorm1d(input_size)
        # 双向LSTM
        self.lstm = nn.LSTM(
            input_size,
            hidden_size,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout,
            bidirectional=True
        )
        # 注意力机制
        self.attention = nn.MultiheadAttention(
            embed_dim=hidden_size * 2,
            num_heads=8,
            dropout=dropout
        )
        # 多层全连接网络
        self.fc_layers = nn.Sequential(
            nn.Linear(hidden_size * 2, hidden_size),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_size, hidden_size // 2),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_size // 2, num_classes)
        )
        # 残差连接
        self.residual = nn.Linear(input_size, hidden_size * 2)
        # 层归一化
        self.layer_norm = nn.LayerNorm(hidden_size * 2)
```

图 38: 模型代码

我们定义了一个批归一化层，删去了主函数中的归一化过程。使用双向LSTM机制，增强对于前后状态的理解能力，同时引入了多头注意力机制(nn.MultiheadAttention)允许模型在处理序列时关注输入序列的不同部分，增强模型对重要信息的捕捉能力。

最后定义一个多层全连接网络，使用ReLU激活和DropOut正则化。

这是前向传播函数：输入处理: 输入 x 的形状为 (batch_size, seq_len, features)，首先通过 reshape 将其调整为 (batch_size * seq_len, features)，然后应用批归一化。将处理后的输入传入LSTM层，得到输出 lstm_out。使用多头注意力机制对LSTM输出进行处理, 将原始输入 x 通过线性层进行变换，得到 residual，并与注意力输出相加形成 combined对 combined 进行层归一化处理，以提高模型的稳定性。取最后一个时间步的输出 out，并通过多层全连接网络进行分类，最终返回模型的输出。

11.4 训练结果

极好的训练结果！:

92.74%的正确率在所有训练方法中遥遥领先。

```python
def forward(self, x):
    # 批归一化
    batch_size, seq_len, features = x.size()
    x = x.reshape(-1, features)
    x = self.batch_norm(x)
    x = x.reshape(batch_size, seq_len, features)

    # LSTM处理
    lstm_out, _ = self.lstm(x)

    # 注意力机制
    attention_out, _ = self.attention(
        lstm_out.permute(1, 0, 2),
        lstm_out.permute(1, 0, 2),
        lstm_out.permute(1, 0, 2)
    )
    attention_out = attention_out.permute(1, 0, 2)

    # 残差连接
    residual = self.residual(x)
    combined = attention_out + residual

    # Layer Normalization
    normalized = self.layer_norm(combined)

    # 取最后一个时间步
    out = normalized[:, -1, :]

    # 通过多层全连接网络
    out = self.fc_layers(out)

    return out
```

图 39: 模型代码

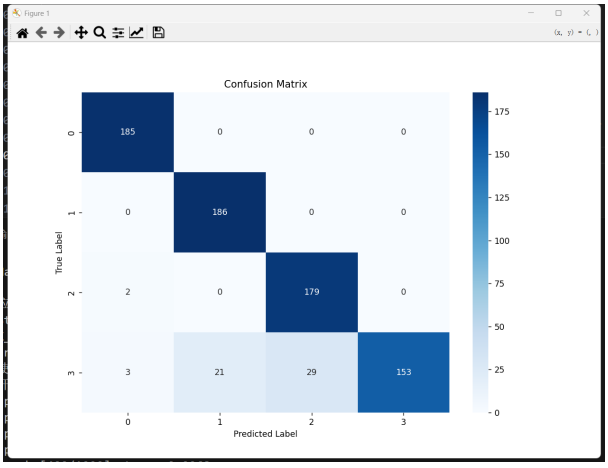

图 40: 混淆矩阵

```
准确率: 92.74%

分类报告:
              precision    recall  f1-score   support

           0       0.97      1.00      0.99       185
           1       0.90      1.00      0.95       186
           2       0.86      0.99      0.92       181
           3       1.00      0.74      0.85       206

    accuracy                           0.93       758
   macro avg       0.93      0.93      0.93       758
weighted avg       0.94      0.93      0.92       758
```

图 41: 分类报告

11.5 实战测试

图 42: 行动决策（节选）

由于可视化界面一直是卡死的，只能这么展示其的判断能力。

通过run and log跑20组试验一下：

存活率达到100%，每次击杀数都非常高，可以说任务已经圆满完成了。

具体使用方法写在md文件里面了，在pycharm里面点击即可运行感受训练和测试过程。

12 结束语

中间特征提取卡了好几天，因为数据确实是决定了模型的上限，所有必须要思考一个合理的方式，最终确定了这种。

配CUDA环境也是卡了一天，果然配环境是最让人丧失学习欲望的事情。

参考文献

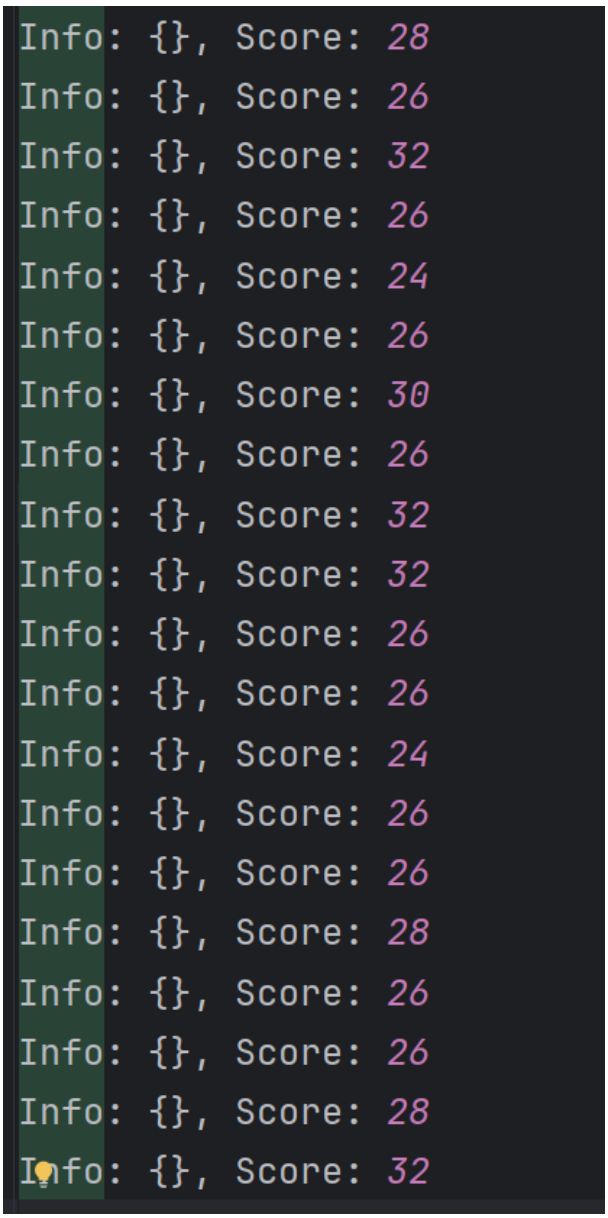
Info: {}, Score: 28
Info: {}, Score: 26
Info: {}, Score: 32
Info: {}, Score: 26
Info: {}, Score: 24
Info: {}, Score: 26
Info: {}, Score: 30
Info: {}, Score: 26
Info: {}, Score: 32
Info: {}, Score: 32
Info: {}, Score: 26
Info: {}, Score: 26
Info: {}, Score: 24
Info: {}, Score: 26
Info: {}, Score: 26
Info: {}, Score: 28
Info: {}, Score: 26
Info: {}, Score: 26
Info: {}, Score: 28
Info: {}, Score: 32

图 43: 20组结果