PyCIL

一、整体架构

- **入口**：/main.py 负责配置加载和任务调度
- **训练调度器**：/trainer.py 管理整个训练流程
- **模型工厂**：/utils/factory.py 根据配置动态创建模型
- **方法实现**：/models/ 方法实现
- **基类**：/models/base.py 提供回放内存管理等基础功能
- **数据管理**：/utils/data_manager.py 管理数据集

二、训练入口

2.1 程序入口

当运行 `python main.py --config ./exps/icarl.json` 时，系统首先加载配置文件：

```
{
    "prefix": "reproduce",
    "dataset": "cifar100",
    "memory_size": 2000,
    "memory_per_class": 20,
    "fixed_memory": false,
    "shuffle": true,
    "init_cls": 10,
    "increment": 10,
    "model_name": "icarl",
    "convnet_type": "resnet18",
    "device": ["0"],
    "seed": [1993]
}
```

该配置指定了：

字段	含义
`dataset`	数据集名称
`memory_size`	样本记忆库总容量
`memory_per_class`	每类保留多少样本
`fixed_memory`	动态类别样本数量
`shuffle`	类别顺序随机化
`init_cls`	初始任务类别数

`increment`	增量类别数
`model_name`	增量学习算法
`convnet_type`	骨干网络
`device`	指定 GPU
`seed`	类别打乱种子

2.2 训练trainer.py

在 trainer.py 的 `_train` 函数中，系统按任务顺序进行训练：

```
for task in range(data_manager.nb_tasks):
    logging.info("All params: {}".format(count_parameters(model._network)))
    model.incremental_train(data_manager)  # 增量训练当前任务
    cnn_accy, nme_accy = model.eval_task()    # 评估当前任务
    model.after_task()                        # 任务后处理（保存旧模型、构建示例集）
```

对于每个任务，系统执行三个核心步骤：训练、评估、后处理。

三、iCaRL 训练流程

3.1 任务初始化

在 /models/icarl.py 的 `incremental_train` 方法中，首先更新任务计数并扩展网络：

```
def incremental_train(self, data_manager):
    # 1. 更新任务计数
    self._cur_task += 1

    # 2. 计算新的总类别数
    self._total_classes = self._known_classes + data_manager.get_task_size(self._cu

    # 3. 扩展网络的全连接层
    self._network.update_fc(self._total_classes, self._network)
```

这一步确保网络能够处理新增加的类别。网络扩展策略只增加了输出层的神经元数量，保持了特征提取器不变，这对于后续的知识蒸馏至关重要。

3.2 训练数据构建

系统构建包含新类样本和回放样本的混合训练集：

```
# 4. 构建训练集（新类样本 + 回放样本）
train_dataset = data_manager.get_dataset(
    np.arange(self._known_classes, self._total_classes),  # 新类索引范围
    source="train",
    mode="train",
    appendent=self._get_memory(),  # 附加回放记忆
)

# 5. 构建测试集（所有已学类别）
test_dataset = data_manager.get_dataset(
    np.arange(0, self._total_classes),  # 所有已学类别
    source="test",
    mode="test"
)
```

训练集的构建策略体现了iCaRL的核心思想：将新类数据与旧类回放样本混合训练，既学习新知识又不遗忘旧知识。测试集则包含所有已学类别，用于全面评估模型性能。

3.3 训练策略

根据任务阶段不同，系统采用两种训练策略：

任务0（初始训练）：

```
if self._cur_task == 0:
    optimizer = optim.SGD(
        self._network.parameters(),
        momentum=0.9,
        lr=init_lr,           # 初始学习率 0.1
        weight_decay=init_weight_decay,
    )
    scheduler = optim.lr_scheduler.MultiStepLR(
        optimizer=optimizer,
        milestones=init_milestones,  # [60, 120, 170]
        gamma=init_lr_decay         # 0.1
    )
    self._init_train(train_loader, test_loader, optimizer, scheduler)
```

后续任务（增量训练）：

```
else:
    optimizer = optim.SGD(
        self._network.parameters(),
        lr=lrate,              # 学习率 0.1
        momentum=0.9,
        weight_decay=weight_decay,
    )
    self._update_representation(train_loader, test_loader, optimizer, scheduler)
```

初始任务采用较高的初始学习率和权重衰减，配合多步学习率衰减策略快速收敛。后续任务则使用较低的学习率进行微调，避免对已学知识的过度破坏。

3.4 知识蒸馏损失

在 /models/icarl.py 的 `_update_representation` 方法中开始增量训练，iCaRL使用知识蒸馏损失来保持旧知识：

```
  def _update_representation(self, train_loader, test_loader, optimizer, scheduler):
      for _, epoch in enumerate(prog_bar):
          self._network.train()
          for i, (_, inputs, targets) in enumerate(train_loader):
              inputs, targets = inputs.to(self._device), targets.to(self._device)
              logits = self._network(inputs)["logits"]

              # 分类损失
              loss_clf = F.cross_entropy(logits, targets)

              # 知识蒸馏损失
              loss_kd = _KD_loss(
                  logits[:, : self._known_classes],  # 旧类别的logits
                  self._old_network(inputs)["logits"],  # 旧模型的预测
                  T=2,  # 蒸馏温度
              )

              # 总损失 = 分类损失 + 蒸馏损失
              loss = loss_clf + loss_kd
```

知识蒸馏损失函数定义如下：

```
def _KD_loss(pred, soft, T):
    pred = torch.log_softmax(pred / T, dim=1)
    soft = torch.softmax(soft / T, dim=1)
    return -1 * torch.mul(soft, pred).sum() / pred.shape[0]
```

蒸馏温度T设置为2，使得旧模型的软标签包含更丰富的类别间相似性信息。通过联合优化分类损失和蒸馏损失，模型能够在学习新类别的同时保持对旧类别的判别能力。

四、回放记忆管理

4.1 任务后处理

每个任务完成后，系统调用 /models/icarl.py `after_task` 方法：

```
def after_task(self):
    self._old_network = self._network.copy().freeze()  # 保存旧模型
    self._known_classes = self._total_classes
    logging.info("Exemplar size: {}".format(self.exemplar_size))
```

保存旧模型用于下一任务的蒸馏训练，同时更新已学类别数量。旧模型被冻结后不再更新。

4.2 样本选择与存储

在 /models/base.py 的 `_construct_exemplar` 方法中采用 **Herding策略**选择最具代表性的样本存储到回放记忆中：

```
def _construct_exemplar(self, data_manager, m):
    for class_idx in range(self._known_classes, self._total_classes):
        data, targets, idx_dataset = data_manager.get_dataset(...)
        idx_loader = DataLoader(idx_dataset, ...)
        vectors, _ = self._extract_vectors(idx_loader)
        vectors = (vectors.T / (np.linalg.norm(vectors.T, axis=0) + EPSILON)).T
        class_mean = np.mean(vectors, axis=0)

        # Herding选择策略
        selected_exemplars = []
        exemplar_vectors = []
        for k in range(1, m + 1):
            S = np.sum(exemplar_vectors, axis=0)
            mu_p = (vectors + S) / k
            i = np.argmin(np.sqrt(np.sum((class_mean - mu_p) ** 2, axis=1)))
            selected_exemplars.append(np.array(data[i]))
            exemplar_vectors.append(np.array(vectors[i]))
            vectors = np.delete(vectors, i, axis=0)
```

Herding策略的核心思想是迭代选择距离类别中心最近的样本，使得最终选择的样本集合的均值最接近类别真实均值。这种策略能够在有限的存储空间内最大化类别表示的完整性。

4.3 类别均值计算

为支持NME（Nearest Mean Exemplar）分类器，系统计算并存储各类别的均值向量：

类别均值经过L2归一化处理，确保后续距离计算的稳定性。这些均值向量在评估阶段用于NME分类器的预测。

五、评估流程

5.1 CNN评估

系统使用当前网络在测试集上进行评估：

```
def _eval_cnn(self, loader):
    self._network.eval()
    y_pred, y_true = [], []
    for _, (_, inputs, targets) in enumerate(loader):
        inputs = inputs.to(self._device)
        with torch.no_grad():
            outputs = self._network(inputs)["logits"]
```

5.2 NME评估

系统还支持基于类别均值的NME分类器评估：

```python
def _eval_nme(self, loader, class_means):
    vectors, y_true = self._extract_vectors(loader)
    vectors = (vectors.T / (np.linalg.norm(vectors.T, axis=0) + EPSILON)).T

    # 计算样本与各类别均值的距离
```

NME分类器通过比较样本特征向量与各类别均值之间的距离进行预测，提供了一种与网络预测互补的评估视角。

5.4 全部任务结束后统计准确率矩阵和Forgetting

六、完整流程时序图

```
main.py ──┬──> 加载配置 (icarl.json)
          │
          └──> train(args)
                 │
                 ├──> 初始化 DataManager
                 │     ├──> _setup_data() ──> 划分训练集/测试集
                 │     └──> _increments = [10, 10, ...]
                 │
                 ├──> factory.get_model("icarl", args)
                 │     └──> return iCaRL(args)
                 │
                 └──> for task in range(nb_tasks):
                        │
                        ├──> model.incremental_train(data_manager)
                        │     │
                        │     ├──> _cur_task += 1
                        │     ├──> _total_classes = _known_classes + task_size
                        │     ├──> _network.update_fc()  # 扩展输出层
                        │     ├──> get_dataset(new_classes, appendent=memory)  # 构建
                        │     ├──> _train()  # 训练网络
                        │     │     ├──> 任务0: _init_train()
                        │     │     └──> 任务>0: _update_representation()
                        │     │           ├──> loss_clf: 交叉熵损失
                        │     │           └──> loss_kd: 知识蒸馏损失
                        │     └──> build_rehearsal_memory()  # 构建示例集
                        │           ├──> _reduce_exemplar()  # 压缩旧示例
                        │           └──> _construct_exemplar()  # Herding选择新示例
                        │
                        ├──> eval_task()
                        │     ├──> _eval_cnn()  # CNN分类器评估
                        │     └──> _eval_nme()  # NME分类器评估
                        │
                        └──> after_task()
                              ├──> _old_network = _network.copy().freeze()
                              └──> _known_classes = _total_classes
```

七、关键要点

八、问题

九、改为自己的数据集和网络

9.1 改为自己的数据集

9.1.1 其他数据集

在 `utils/*` 中新建一个文件对数据集进行预处理，返回测试集和训练集：

```
def process_data(data_dir):
...
    return train_data, train_labels, test_data, test_labels
```

然后在 `utils/data.py` 进行数据导入：

```
class iMyDATA(iData):
    use_path = False
    train_trsf = []
    test_trsf = []
    common_trsf = []
    class_order = np.arange(53).tolist()
    def __init__(self, data_dir):
        super().__init__()
        self.data_dir = data_dir

    def download_data(self):
        self.train_data, self.train_targets, self.test_data, self.test_targets = pr
            data_dir=self.data_dir
        )
```

训练时数据在 `utils/data_manager.py` 的 `DummyDataset` 进行动态加载。

9.2 改为自己的网络

1. 在 `convs/` 下新建一个网络，例 `SimpleCNN.py` ：

```
import torch
from torch import nn

class SimpleCNN(nn.Module):
    def __init__(self, input_channels=1, num_classes=10):
        super(SimpleCNN, self).__init__()

        self.conv1 = nn.Sequential(
            nn.Conv1d(input_channels, 32, kernel_size=3, padding=1),
            nn.BatchNorm1d(32),
```

注意:

- 设置网络的**输出维度** `self.out_dim = 128`
- 无需设置 **fc**

2. 在 `utils/inc_net.py` 中注册网络: `if name == 'SimpleCNN': return SimpleCNN()`

9.3 改为自己的增量学习方法

1. 在 `models/` 下新建一个方法
2. 在 `utils/factory.py` 中注册方法:

`if name == "icarl": from models.icarl import iCaRL return iCaRL(args)`