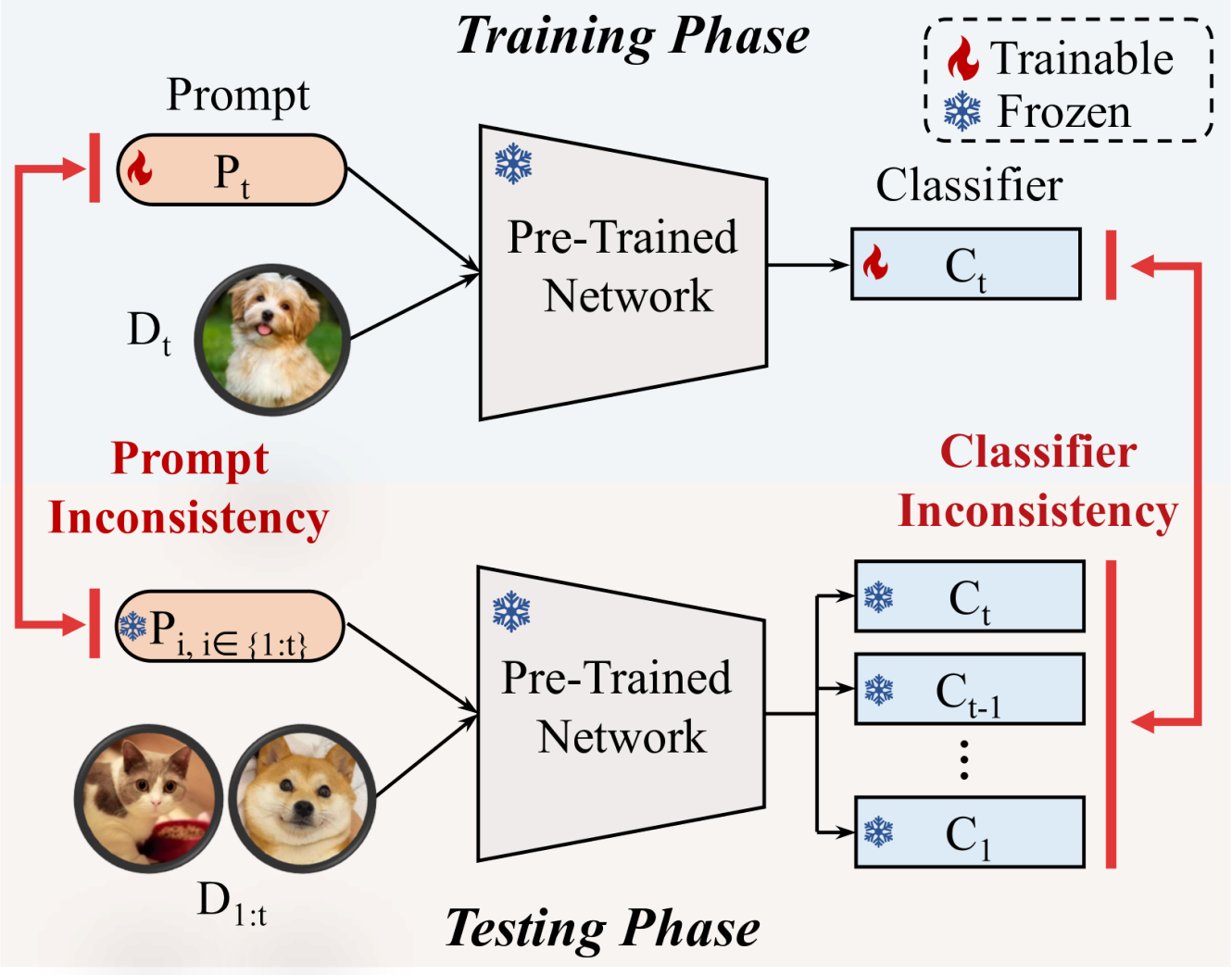
机器学习导论课程大作业实验报告

231880101 孙俊晖

一、复现的算法

我复现的论文为 Consistent Prompting for Rehearsal-Free Continual Learning (CVPR2024)，这篇论文针对现有基于提示（Prompt）的无重演的持续学习方法中存在的不一致问题，提出了一种 CPrompt 算法来解决该问题。该论文首先指出目前基于提示的无重演的持续学习方法存在两种不一致问题，分别是分类器不一致和提示不一致。分类器不一致指的是在训练过程中仅优化当前任务的分类器，但是在测试时需使用所有历史任务的分类器投票预测而不仅仅是当前任务训练得到的分类器，这种模型未在训练阶段对齐所有分类器的行为，会导致测试时预测偏差；提示不一致指的是训练和测试之间的提示不匹配，训练时固定使用与当前任务关联的正确提示，但是在测试时需通过查询机制动态选择提示，这可能会因提示选择错误而使用不匹配的提示。

下图展示了这两种不一致性问题：



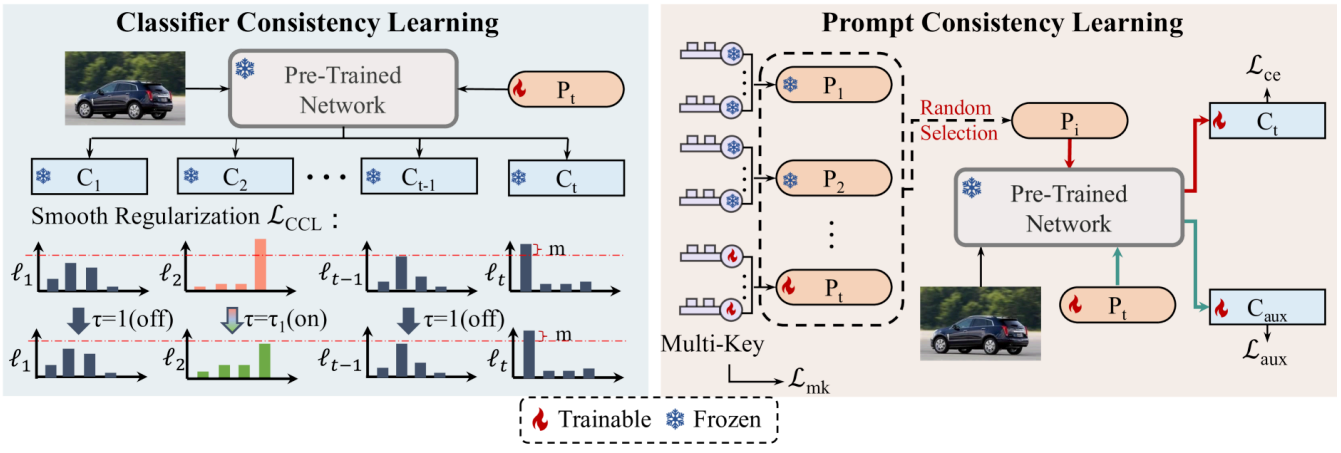
这两种不一致性均会导致模型性能下降，而这篇论文提出的 CPrompt 算法包含两个核心模块，分别解决上述的两种不一致性。

第一个核心模块是 Classifier Consistency Learning，即分类器一致性学习，简称 CCL，它的目标是使所有历史分类器在训练时对齐测试行为。设预训练网络为 f_θ，训练任务 t 的嵌入维度为 D_t，任务 t 的提示为 P_t，首先提取关于特定类别的图像特征 $h = f_\theta(x, P_t)[0]$，其中 $x \in D_t$ 是当前训练任务 t 的输入图像。然后，提取的特征 h 被输入到所有任务分类器 $\{C_1, C_2, \dots C_t\}$ 中，并通过它们获得相应的 $logit$ 值：$l_i = C_i(h), i \in \{1, 2, \dots t\}$。在CCL 中，引入了自适应平滑正则化来计算损失。对于第 i 个任务($i \in \{1, 2, \dots, t-1\}$)，基于 $logit$ 值 l_i 计算自适应熵：$\mathcal{L}_e = - < \sigma(l_i/\tau), log(\sigma(l_i)) >$，其中 σ 是 $softmax$ 函数，$<,>$是内积算子，τ 为温度。τ 用于调整分类器输出的平滑度以控制模型在持续学习过程中的一致性，$\tau = \begin{cases} \tau_1, & max(l_i) + m \geq max(l_t), \\ 1, & otherwise. \end{cases}$。最终，计算分类器一致性学习的损失：$\mathcal{L}_{CCL} = \frac{\alpha}{t-1} \sum_{i=1}^{t-1} \mathcal{L}_e(i)$，其中 α 表示正则化系数。

另一个核心模块则是 Prompt Consistency Learning，即提示一致性学习，简称 PCL，旨在建立更稳健的提示-分类器关系，尽可能确保即使使用错误的提示也能得到正确输出。在当前任务 t 的训练过程中，从当前提示池中随机选择一个特定于任务的提示 P_i，i 满足 $[1, t]$ 上的均匀分布：$i \sim Uni(1, t)$。除了 P_t 之外的所有提示都被冻结以保留编码的知识并防止灾难性遗忘。然后，可以通过 Ct 获得当前任务分类器的输出：$l_t = C_t(f_\theta(x, P_i)[0])$。同时计算损失：$\mathcal{L}_{ce} = CrossEntropy(l_t, y)$，其中 y 是输入图像 x 的真实类别标签。但是，当前任务特定的提示 P_t 仅有 $\frac{1}{t}$ 的概率被选中，这导致对 P_t 的训练不足。因此，在PCL 中采用一个辅助分类器 C_{aux} 来协助训练 P_t：$\mathcal{L}_{aux} = CrossEntry(C_{aux}(h), y)$，其中 $h = f_\theta(x, P_t)[0]$ 就是在 CCL中提取的关于特定类别的图像特征。除此之外，PCL 还引入了多键机制，用于更精确的提示选择，从而提升持续学习性能。从同一类别的预训练网络中提取的查询特征往往相似，但在同一任务中可能表现出跨类别的多样性。因此，像之前基于提示的方法那样，仅用一个键来表示每个任务是不够的。所提出的多键机制在提示中使用多个键来映射每个任务，具体来说，任务中的每个类别被分配一个唯一的键，键的数量与类别的数量相同。然后使用余弦相似度来测量查询与其对应键之间的差异：$d_{i,j} = cos(q, k_{i,j})$，其中 $q = f_\theta(x)[0]$ 为从预训练网络中提取的查询特征，$k_{i,j}$ 表示任务 i 中的 j 类的键。在选择提示时，选择相似度最大的键：$i = \underset{i \in \{1:t\} j \in \{1:|Y_i|\}}{\arg\max} d_{i,j}$，其中 $|Y_i|$ 是任务 i 中的类别数量。在训练过程中，使用 $softmax$ 交叉熵来最大化查询与其对应键之间的相似度，同时最小化与其他键之间的相似度：$\mathcal{L}_{mk} = -log(\frac{e^{d_{t,y(x)}}}{\sum_{i \in \{1:t\}, j \in \{1:|Y_i|\}} e^{d_{i,j(x)}}})$，其中 y 代表当前任务 t 中输入图像 x 的类别标签。最后，计算提示一致性学习的损失：$\mathcal{L}_{PCL} = \mathcal{L}_{ce} + \mathcal{L}_{aux} + \mathcal{L}_{mk}$。

CPrompt算法中，CCL 确保所有分类器在训练时对齐测试行为，PCL 增强模型对提示错误的容错能力。CPrompt 算法的总损失为 $\mathcal{L} = \mathcal{L}_{CCL} + \mathcal{L}_{PCL}$，同时也是 CPrompt 算法的整体学习目标。

下图展示了 CPrompt算法的两个核心模块，即 CCL 和 PCL：



二、复现过程

1.实现 CPrompt.py

(1) CPrompt_Net类的实现

我首先先实现了 CPrompt 算法的核心网络结构 CPrompt_Net，该网络结构负责处理增量学习任务中的特征提取、分类以及任务提示生成，用于辅助 CPrompt 类的实现。

我首先实现了构造函数，用于初始化网络结构。首先，我初始化了设备为GPU，选择 cifar100 作为数据集，并初始化了初始类别数、增量类别数和特征维度。CPrompt_Net 以 backbone 作为骨干网络，用于提取图像特征，并通过动态更新分类器和任务提示层来适应新的任务。分类器使用 ModuleList 存储每个任务的线性分类器权重，支持增量更新以处理新类别。任务提示层通过 ParameterList 存储提示嵌入，用于在特征空间中引入任务相关信息，从而增强模型对新任务的适应性。最后，我使用 nn.Parameter 初始化任务键，用于存储每个类别的特征键，为后面实现多键机制做准备。

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from copy import deepcopy
from core.model.finetune import Finetune

class CPrompt_Net(nn.Module):
    def __init__(self, backbone, kwargs):
        super().__init__()
        self.kwargs = kwargs
        self.device = kwargs['device'][0] if isinstance(kwargs['device'], list) else kwargs['device']
        self.dataset_name = kwargs.get("dataset", "cifar100")
        self.init_cls_num = kwargs.get("init_cls_num", 10)
        self.inc_cls_num = kwargs.get("inc_cls_num", 10)
        self.feat_dim = kwargs.get("feat_dim", 768)
        # 初始化图像编码器
        self.image_encoder = backbone
        # 初始化分类器列表
        self.clas_w = nn.ModuleList()
        self.clas_w.append(nn.Linear(self.feat_dim, self.init_cls_num))
        # 初始化提示层
        self.ts_prompts_1 = nn.ParameterList([
            nn.Parameter(torch.randn(1, self.feat_dim))
        ])
        self.ts_prompts_2 = nn.ParameterList([
            nn.Parameter(torch.randn(1, self.feat_dim))
        ])
        # 初始化键
        self.keys = nn.Parameter(torch.randn(self.init_cls_num, self.feat_dim))
```

接着，我实现了 update_fc 函数，负责动态更新分类器、任务提示层和任务键，以适应增量学习中的新任务。当模型接收到新的任务时，函数会根据当前任务的类别数和总类别数更新分类器。如果分类器已经存在，它会将旧分类器的权重和偏置复制到新分类器中，以保留旧任务的知识。此外，函数会为当前任务生成新的任务提示层ts_prompts_1和ts_prompts_2，并随机初始化提示嵌入。最后，函数会更新任务键 keys，将新任务的键与旧任务的键拼接起来，用于支持任务间的特征对齐和分类一致性。

```
def update_fc(self, total_classes, cur_task_nbclasses):
        # 更新分类器
        if len(self.clas_w) == 0:
            self.clas_w.append(nn.Linear(self.feat_dim, total_classes))
        else:
            old_fc = self.clas_w[-1]
            new_fc = nn.Linear(self.feat_dim, total_classes)
            if total_classes > cur_task_nbclasses:
                new_fc.weight.data[:total_classes - cur_task_nbclasses] =
old_fc.weight.data
                new_fc.bias.data[:total_classes - cur_task_nbclasses] =
old_fc.bias.data
            self.clas_w.append(new_fc)
        # 更新提示层
        self.ts_prompts_1.append(nn.Parameter(torch.randn(1, self.feat_dim)))
        self.ts_prompts_2.append(nn.Parameter(torch.randn(1, self.feat_dim)))
        # 更新键
        new_keys = nn.Parameter(torch.randn(cur_task_nbclasses,
self.feat_dim).to(self.device))
        if len(self.keys) == 0:
            self.keys = new_keys
        else:
            self.keys = nn.Parameter(torch.cat([self.keys, new_keys], dim=0))
```

aux_forward 函数用于 PCL 中辅助分类器的前向传播，负责提取图像特征并生成分类结果。它首先通过骨干网络 image_encoder 提取输入图像的特征，并确保特征是二维的（形状为 [batch_size, feat_dim]）。如果特征是高维的，函数会对空间维度进行平均池化以降维。然后，函数使用最新的分类器 clas_w[-1] 对提取的特征进行分类，返回分类结果 logits 和特征 features。

```
def aux_forward(self, x):
        features = self.image_encoder(x)
        # 确保特征是二维的
        if features.dim() > 2:
            features = features.mean(dim=[2, 3])
        logits = self.clas_w[-1](features)
        return logits, features
```

forward 函数实现了主要前向传播逻辑，用于处理输入图像并生成分类结果。它首先通过骨干网络image_encoder 提取图像特征，并确保特征是二维的。然后，函数应用任务提示 gen_p，将提示嵌入 P1和 P2添加到特征中，以增强模型对当前任务的适应性。最后，函数使用最新的分类器 clas_w[-1] 对增强后的特征进行分类，返回分类结果 logits。该函数支持训练和推理两种模式，是模型处理输入数据的核心。

```python
 def forward(self, x, gen_p, train=False):
        features = self.image_encoder(x)
        # 确保特征是二维的
        if features.dim() > 2:
            features = features.mean(dim=[2, 3])
        # 应用提示
        P1, P2 = gen_p
        features = features + P1.squeeze(1) + P2.squeeze(1)
        logits = self.clas_w[-1](features)
        return logits
```

最后，我实现了 fix_branch_layer函数，用于冻结当前任务的参数，以防止它们在后续任务中被更新。具体来说，函数会将最新分类器 clas_w[-1] 的所有参数设置为不可训练状态（requires_grad = False）。同时，它会冻结当前任务的提示参数 ts_prompts_1[-1] 和 ts_prompts_2[-1]，确保提示嵌入在后续任务中保持不变。通过冻结这些参数，模型能够专注于学习新任务的知识，同时保留对旧任务的记忆，从而减少遗忘现象。

```python
def fix_branch_layer(self):
        # 冻结当前任务的参数
        for param in self.clas_w[-1].parameters():
            param.requires_grad = False
        # 冻结当前任务的提示参数
        self.ts_prompts_1[-1].requires_grad = False
        self.ts_prompts_2[-1].requires_grad = False
```

(2) CPrompt类的实现

CPrompt类的父类为Finetune，与CPrompt_Net类的实现一样，我首先了实现了构造函数。它首先从传入的配置参数中提取任务相关信息，包括初始类别数 init_cls_num、增量类别数 inc_cls_num、特征维度 feat_dim、设备 device 和数据集名称 dataset_name 等。随后，构造函数初始化当前任务索引 cur_task、已知类别数 known_classes 和总类别数 total_classes，用于跟踪任务的进展和类别范围。此外，它根据配置参数创建了一个基于 backbone 的骨干网络，并通过 CPrompt_Net类初始化了分类器、任务提示层和任务键，用于支持增量学习中的动态更新和任务间特征对齐。最后，构造函数将网络迁移到指定设备，并创建存储评估指标的列表。

```python
class CPrompt(Finetune):
    def __init__(self, backbone, feat_dim, num_class, **kwargs):
        super().__init__(backbone, feat_dim, num_class, **kwargs)
        self.kwargs = kwargs
        self.device = kwargs['device'][0] if isinstance(kwargs['device'], list) else
kwargs['device']
        # 任务状态管理
        self.cur_task = 0
        self.known_classes = 0
        self.total_classes = 0
        # 数据集与超参数配置
        self.dataset_name = kwargs.get("dataset", "cifar100")
        self.init_cls_num = kwargs.get("init_cls_num", 10)
        self.inc_cls_num = kwargs.get("inc_cls_num", 10)
        self.margin = kwargs.get("margin", 0.05)
        self.tau = kwargs.get("tau", 1.02)
        self.alpha = kwargs.get("alpha", 1.0)
```

```
        # 初始化网络
        self.network = CPrompt_Net(self.backbone, kwargs)
        self.network.to(self.device)
        # 记录评估指标
        self.acc = []
        self.faa_accuracy_table = []
```

然后，我实现了 before_task 函数。函数 before_task在每个任务开始前执行，用于初始化当前任务的状态和网络结构。它首先更新当前任务的索引 cur_task 和总类别数 total_classes，并根据任务索引确定当前任务的类别范围。如果是第一个任务，则初始化类别数为 init_cls_num；否则，每次任务增加 inc_cls_num 个类别。随后，函数调用 CPrompt_Net 的 update_fc 方法，动态更新分类器、任务提示层和任务键，以适应新的任务类别。最后，将网络迁移到指定设备，确保模型能够正常运行。

```
def before_task(self, task_idx, buffer, train_loader, test_loaders):
        self.cur_task = task_idx
        # 计算当前任务的类别范围
        if task_idx == 0:
            self.total_classes = self.init_cls_num
        else:
            self.total_classes += self.inc_cls_num
        # 更新网络分类器和提示
        cur_task_nbclasses = self.inc_cls_num if task_idx > 0 else self.init_cls_num
        self.network.update_fc(self.total_classes, cur_task_nbclasses)
        self.network.to(self.device)
```

接着，我实现了 observe函数。observe 函数是训练过程中每次迭代的核心逻辑，用于处理输入数据并计算损失。它首先将输入图像和标签迁移到指定设备，并根据当前任务的已知类别数调整标签范围。函数计算辅助分类器的交叉熵损失 loss_aux，用于当前任务的分类。随后，它计算分类器一致性损失 CCL，通过比较当前任务和历史任务的分类结果，确保模型在旧类别上的预测保持一致。接着，函数计算多键机制损失 MK，通过余弦相似度匹配当前任务的特征与任务键，增强模型对任务的区分能力。最后，函数生成提示层并计算提示分类损失 PCL，通过动态选择历史提示增强模型对当前任务的适应性。最后，函数返回预测结果、准确率和总损失。

```
def observe(self, data):
        x, y = data['image'], data['label']
        x = x.to(self.device)
        y = y.to(self.device)
        # 计算新类别标签
        new_targets = y - self.known_classes
        # 辅助分类器损失
        logits, features = self.network.aux_forward(x)
        loss_aux = F.cross_entropy(logits, new_targets)
        loss = loss_aux
        # 分类器一致性损失（CCL）
        if self.cur_task > 0:
            for k in range(self.cur_task):
                old_logit = self.network.clas_w[k](features)
                cur_logit = self.network.clas_w[self.cur_task](features)
                # 获取当前任务和历史任务的类别数
                old_classes = self.init_cls_num if k == 0 else self.init_cls_num + k *
self.inc_cls_num
                cur_classes = self.init_cls_num + self.cur_task * self.inc_cls_num
```

```
                # 只在旧类别范围内计算一致性损失
                old_logit_subset = old_logit
                cur_logit_subset = cur_logit[:, :old_classes]
                # 判断是否需要应用温度系数
                bool_mask = (torch.max(cur_logit_subset, dim=1)[0] >
                        torch.max(old_logit_subset, dim=1)[0] + self.margin)
                t = torch.ones_like(bool_mask, dtype=torch.float32).to(self.device)
                t[~bool_mask] = self.tau
                t = t.unsqueeze(1).repeat(1, old_classes)
                # 平滑正则化
                ground = F.softmax(old_logit / t, dim=1).detach()
                loss_ccl = -torch.sum(ground * torch.log(F.softmax(old_logit, dim=1)),
dim=1).mean()
                loss += self.alpha * loss_ccl / self.cur_task
        # 多键机制损失（MK）
        with torch.no_grad():
            features = self.network.image_encoder(x)
            if features.dim() == 3:
                x_querry = features[:, 0, :]  # [batch_size, feature_dim]
            else:
                x_querry = features
        K = self.network.keys
        # 截取当前任务的键
        s = self.cur_task * self.inc_cls_num
        f = (self.cur_task + 1) * self.inc_cls_num
        if self.cur_task == 0:
            K = K[s:f]
        else:
            K = torch.cat([K[:s].detach(), K[s:f]], dim=0)
        # 余弦相似度计算
        n_K = F.normalize(K, dim=1)
        q = F.normalize(x_querry, dim=1)
        mk = torch.einsum('bd,kd->bk', q, n_K)
        loss_mk = F.cross_entropy(mk, y)
        loss += loss_mk
        # 生成提示损失（PCL）
        gen_p = []
        # 随机选择历史提示
        m = torch.randint(0, self.cur_task + 1, (x.size(0), 1))
        # 生成第一层提示
        P1 = torch.cat([self.network.ts_prompts_1[j].unsqueeze(0) for j in m], dim=0)
        gen_p.append(P1)
        # 生成第二层提示
        P2 = torch.cat([self.network.ts_prompts_2[j].unsqueeze(0) for j in m], dim=0)
        gen_p.append(P2)
        # 生成提示的分类损失
        out_gen = self.network(x, gen_p, train=True)
        loss_ce = F.cross_entropy(out_gen, new_targets)
        loss += loss_ce
        # 计算准确率
        _, preds = torch.max(logits, dim=1)
        acc = torch.sum(preds.eq(new_targets)).item() / x.size(0)

```

```
        return preds, acc, loss
```

inference 函数用于推理阶段的处理，负责生成预测结果和计算准确率。它首先将输入图像和标签迁移到指定设备，并通过骨干网络提取特征。随后，函数利用多键机制选择提示层，通过余弦相似度匹配当前任务的特征与任务键，动态生成提示嵌入。生成的提示嵌入用于增强模型对当前任务的适应性。最后，函数通过网络的前向传播生成分类结果，并计算预测准确率。最后，函数返回预测结果和准确率。

```
def inference(self, data):
        x, y = data['image'], data['label']
        x = x.to(self.device)
        y = y.to(self.device)
        # 生成提示
        gen_p = []
        with torch.no_grad():
            features = self.network.image_encoder(x)
            if features.dim() == 3:
                x_querry = features[:, 0, :]
            else:
                x_querry = features
        # 多键机制选择提示
        K = self.network.keys
        f = (self.cur_task + 1) * self.inc_cls_num
        K = K[:f]
        n_K = F.normalize(K, dim=1)
        q = F.normalize(x_querry, dim=1)
        mk = torch.einsum('bd,kd->bk', q, n_K)
        m = torch.max(mk, dim=1, keepdim=True)[1] // self.inc_cls_num
        # 选择对应的提示
        P1 = torch.cat([self.network.ts_prompts_1[j].detach().unsqueeze(0) for j in m],
dim=0)
        gen_p.append(P1)
        P2 = torch.cat([self.network.ts_prompts_2[j].detach().unsqueeze(0) for j in m],
dim=0)
        gen_p.append(P2)
        # 推理预测
        with torch.no_grad():
            out_logits = self.network(x, gen_p, train=False)

        preds = torch.argmax(out_logits, dim=1)
        acc = torch.sum(preds.eq(y)).item() / x.size(0)

        return preds, acc
```

然后，我实现了 after_task函数。after_task 函数在每个任务结束后执行，用于更新任务状态和冻结当前任务的参数。它首先更新已知类别数 known_classes，确保模型能够正确处理后续任务的类别范围。随后，函数调用 CPrompt_Net的 fix_branch_layer方法，冻结当前任务的分类器和提示层参数，防止它们在后续任务中被更新，以减少遗忘现象，尽可能保留对旧任务的记忆。

```
def after_task(self, task_idx, buffer, train_loader, test_loaders):
        self.known_classes = self.total_classes
        self.network.fix_branch_layer()
```

最后，get_parameters函数获取当前任务的可训练参数并返回。

```
def get_parameters(self, config):
        return filter(lambda p: p.requires_grad, self.network.parameters())
```

2.编写配置文件 CPrompt.yaml

因为我复现的这篇论文已经开源，在编写配置文件时，我尽量保证配置文件中出现的参数与论文的源代码一致。配置文件首先定义了任务的基本信息，包括初始类别数 init_cls_num、每个任务增加的类别数 inc_cls_num、总类别数 total_cls_num 和任务数量 task_num，以支持多任务增量学习的设置。接着，文件详细配置了训练过程的参数，例如训练和测试的批量大小 train_batch_size和 test_batch_size、训练轮数 epoch、验证频率 val_per_epoch 等，同时设置了随机种子 seed和确定性选项 deterministic，以确保实验的可重复性。在数据预处理部分，文件定义了训练和测试数据的转换流程，包括随机裁剪、水平翻转、归一化等操作，以适配图像输入的特征。优化器部分使用 SGD，并设置了学习率、动量和权重衰减等超参数；学习率调度器采用余弦退火策略 CosineAnnealingLR，与训练轮数一致。骨干网络部分选择了预训练的 Vision Transformer vit_base_patch16_224，并设置了特征维度 feat_dim 和类别数 num_classes；分类器部分则定义了 CPrompt 的相关参数，包括类别数、特征维度、损失权重 alpha、温度 tau 和边界 margin等，用于控制分类器一致性损失 CCL 的行为。

```
image_size: 224
# 任务配置
init_cls_num: 10
inc_cls_num: 10
total_cls_num: 100
task_num: 10
# 训练配置
epoch: 170
val_per_epoch: 10
train_batch_size: 16
test_batch_size: 16

seed: 1993
deterministic: True
# 数据转换
train_trfms:
  - RandomResizedCrop:
      size: 224
      scale: [0.05, 1.0]
      ratio: [0.75, 1.33333333]
  - RandomHorizontalFlip: {}
  - ToTensor: {}
  - Normalize:
      mean: [0.5071, 0.4866, 0.4409]
      std: [0.2009, 0.1984, 0.2023]

test_trfms:
  - Resize:
      size: 224
      interpolation: BICUBIC
  - ToTensor: {}
  - Normalize:
      mean: [0.5071, 0.4866, 0.4409]
```

```
      std: [0.2009, 0.1984, 0.2023]
# 优化器配置
optimizer:
  name: SGD
  kwargs:
    lr: 0.01
    momentum: 0.9
    weight_decay: 5e-4
# 学习率调度器
lr_scheduler:
  name: CosineAnnealingLR
  kwargs:
    T_max: 170  # 与epoch数一致
# 骨干网络配置
backbone:
  name: vit_pt_imnet
  kwargs:
    num_classes: 100
    pretrained: true
    model_name: vit_base_patch16_224
# 分类器配置
classifier:
  name: CPrompt
  kwargs:
    num_class: 100
    feat_dim: 768  # ViT-B/16的特征维度
    init_cls_num: 10  # 初始类别数
    inc_cls_num: 10   # 每任务增加的类别数
    margin: 0.1       # CCL损失边界
    tau: 1.1          # CCL温度参数
    alpha: 1.0        # CCL损失权重
```

三、复现结果

在`core/model/backbone/__init__.py`中添加实现的方法 CPrompt 和需要使用到的backbone VisionTransformer。

```
from .vit_inflora import VisionTransformer
from core.model.CPrompt import CPrompt
```

然后，就可以运行实现的 CPrompt 算法。

```
python run_trainer.py --config CPrompt.yaml
```

由于之前复现代码时用时过长，我的训练时间不足，最后只训练了3个task，这是我在训练了3个任务后的结果：

```
===============Train on train set===============
Epoch [160/170] Learning Rate [8.513450158049108e-05] | Loss: 5.1985  Average Acc: 89.96
===============Train on train set===============
Epoch [161/170] Learning Rate [6.899626323298713e-05] | Loss: 5.1913  Average Acc: 90.14
===============Train on train set===============
Epoch [162/170] Learning Rate [5.454195814427021e-05] | Loss: 5.1967  Average Acc: 90.14
===============Train on train set===============
Epoch [163/170] Learning Rate [4.177652244628627e-05] | Loss: 5.2199  Average Acc: 89.74
===============Train on train set===============
Epoch [164/170] Learning Rate [3.070431552363195e-05] | Loss: 5.2030  Average Acc: 89.76
===============Train on train set===============
Epoch [165/170] Learning Rate [2.132911852482766e-05] | Loss: 5.1847  Average Acc: 90.42
===============Train on train set===============
Epoch [166/170] Learning Rate [1.3654133071059893e-05]  | Loss: 5.1812  Average Acc: 90.12
===============Train on train set===============
Epoch [167/170] Learning Rate [7.681980162830282e-06] | Loss: 5.1348  Average Acc: 91.25
===============Train on train set===============
Epoch [168/170] Learning Rate [3.4146992848854697e-06]  | Loss: 5.1936  Average Acc: 90.34
===============Train on train set===============
Epoch [169/170] Learning Rate [8.537477097364521e-07] | Loss: 5.1926  Average Acc: 90.52
===============Validation on test set===============
```

这是原论文在10任务设置下在数据集 Split CIFAR-100 上的持续学习结果：

Method 方法	Last-acc ↑ 最后准确率 ↑	Avg-acc ↑ 平均准确率 ↑	FF ↓
UB	91.99	-	-
L2P	86.38±0.31	91.45±0.19	5.88±0.76
DualPrompt 双提示	86.61±0.22	90.82±1.47	5.86±0.62
ESN	86.42±0.80	91.65±0.67	6.08±0.48
CODAprompt	85.73±0.14	91.03±0.57	7.13±0.44
Ours 我们	**87.82±0.21**	**92.53±0.23**	**5.06±0.50**

训练了3个任务后的结果与原论文训练10个task后的结果相近，但是随着已学习任务数量的增加，准确率不可避免的会显著减小，因此，我没能达到复现的精度要求，但我认为我复现的方向没走错，因为我训练时的准确率明显上升了，已经从一开始的12.22%上升到了90%。

四、复现过程中遇到的困难

1.框架适配问题

LibContinual框架的设计与论文源代码的结构不同，需要将 CPrompt算法拆分成符合框架要求的模块，并实现before_task、observe、inference等接口。我一开始真的感到无从下手，甚至觉得这个框架非但没有方便我复现论文中的算法，甚至还有点徒增累赘，而且这个框架没有什么测试接口，跑代码经常已经跑了很长一段时间，结果突然出一个报错，非常搞人心态。

2.框架提供的backbone得根据需要进行修改

论文中源代码的提示嵌入是通过在 ViT 的输入序列中插入可学习的token实现的。但 LibContinual 中默认的 ViT 骨干网络不支持动态插入token，因此需要修改 ViT 的前向传播函数，增加了实现难度。我当时运行代码时一直有报错，一直以为是其他地方的问题，后来才发现是 CPrompt_Net 类的前向传播函数的问题。

3.算力不足

按照论文源代码的训练规模，我需要训练10个task，每个task中有170个epoch，我电脑的显卡是4060，我用我自己的电脑去训练，每个epoch得训练4-5分钟，一个task得训练11个小时，10个task得训练110个小时，大概是5天。就算去网上租云服务器，4090的显卡也最短只能缩短一半的时间，跑一次代码的时间过长，导致试错的时间成本极高，我没有足够的时间去微调参数。

五、对此次大作业的一些思考

此次大作业让我对持续学习有了更深刻的认识。通过复现CPrompt算法，我不仅掌握了基于提示的无重演持续学习的核心问题———分类器与提示不一致，更深入理解了如何通过CCL和PCL双模块设计实现模型对齐与鲁棒性提升。在实现过程中，LibContinual框架的适配问题一度让我困惑，尤其是需要将论文方法拆解为符合框架接口的模块，这一过程促使我提升了代码抽象与模块化设计能力。此外，修改ViT骨干网络以支持动态提示嵌入的经历，让我对Transformer模型的可扩展性有了更直观的认识。

算力限制是本次实践的最大挑战。由于硬件资源有限，而我的训练时间过长，可能训练了很长时间结果却不尽如人意，这让我意识到算法效率的重要性——未来可尝试优化提示选择机制或引入知识蒸馏以降低计算开销。同时，实验中发现的CCL温度系数动态调整策略对模型一致性的显著影响，也启发我进一步探索自适应正则化技术在其他持续学习场景中的应用。

总体而言，这次大作业不仅巩固了我对持续学习理论的理解，更锻炼了将前沿算法落地实施的能力。它让我明白，复现论文不仅是代码的翻译，更是对算法细节、工程实现与资源约束的综合考量。