MEMO 模型解析和成果复现报告

赖继杰 张昊昀

一、背景介绍

MEMO（Memory-efficient Expandable MOdel，内存高效的可扩展模型）是一种基于模型的类增量学习方法，该模型根据新数据对已有的网络进行扩展和剪枝操作，以使模型动态地匹配增量学习任务的需求。

选题参考：A Model or 603 Exemplars: Towards Memory-Efficient Class-Incremental Learning

结果代码已上传到 Github 上：https://github.com/Alan-season/lab_ML_MEMO

二、问题建模

类增量学习旨在不断学习包含新类的数据流，其核心问题在于对抗“灾难性遗忘”。基于模型和样例的增量学习方法通过保存已知类的代表样例来维持过往任务的知识。假设增量学习 Base-x, Inc-y，表示将 x 个类作为基类并在每个任务中学习 y 个新类的设置，如 Base-0, Inc-10 表示每个任务增量训练十个类（这种情况下将第一次任务的 10 个类作为基类）。假设I_i为 Inc-y 中的第i项任务，从I_i之前的类中选定一组代表性实例作为样本集$\mathcal{E}$，则在训练任务I_i的过程中，模型只能访问I_i和$\mathcal{E}$的数据。CIL 在每个步骤的目标不仅是从当前任务I_i中获取知识，还要保持以前任务的知识。

图像分类任务常用交叉熵损失作为损失函数。记当前任务I_i已知的所有类别数为$|\mathcal{Y}_i|$，ϕ为提取图像特征的嵌入，包含为增量任务创建的i个嵌入，每个嵌入输出特征维数为d，则最后的全连接层$W_{main} \in R^{di\times|\mathcal{Y}_i|}$，交叉熵损失：

$$Loss_{main}(x,y)=\sum_{k=1}^{|Y_i|}-I(y=k)\log S_k\left(W_{main}^{\top}\phi(x)\right) \tag{1}$$

此外，为了区分旧类和新类，引入辅助损失。记第i个增量阶段的数据集I_i包含$|Y_i|$个类，创建一个辅助全连接层$W_{aux} \in R^{d\times(|Y_i|+1)}$。辅助损失表示为：

$$Loss_{aux}(x,y)=\sum_{k=1}^{|Y_i|+1}-I(\hat{y}=k)\log S_k\left(W_{aux}^{\top}\phi_i(x)\right) \tag{2}$$

其中，ϕ_i是为I_i创建的第i个嵌入，$\hat{y}$将真实标签重新分配到$|Y_i|+1$个类中，并将$y \notin Y_i$视为

类$|Y_i|+1$。最终损失使用权重λ平衡模型在旧类和新类上的性能表现：

$$Loss = (1-\lambda)Loss_{main} + \lambda Loss_{aux} \tag{3}$$

三、项目分析

3.1 模型架构

3.1.1 MEMO：通用块-专用块架构

MEMO 的设计思路是使用动态网络，根据新数据对已有的网络进行扩展和剪枝操作，以使模型动态地匹配增量学习任务的需求，该方法是基于模型的增量学习方法的改进。对于一个 n 次的增量任务，一般的基于模型的增量学习方法为每个增量任务训练一个新模型并保存，整个过程一共需要训练 n 个模型，对应 n 个主干网络。MEMO 的作者研究了模型的主干网络，发现网络的浅层块和深层块在模型中起到的作用并不对等。

具体而言，作者对比了网络中浅层块和深层块的块方向梯度、偏移和相似度差异，发现：在增量学习过程中，模型的深层参数的梯度变化和偏移比浅层参数要剧烈，以及在最终得到的 n 个模型中，各主干的深层特征存在较大差异，而浅层特征高度相似。这说明了网络的浅层块倾向于提供通用的表示，而较深的层块提供专用的表示。

为此，MEMO 将主干网络划分为通用块（Generalized Blocks）和专用块（Specialized Blocks），所有增量任务共用一块通用块，对于不同的增量任务，训练和存储不同的专用块，这种做法相比一般的方法节省了 n-1 块通用块的存储空间，显著地减少了主干网络的占用，节约了内存空间。MEMO 将节约得到的空间用于存储更多的样例，进一步改善 MEMO 的性能。

对于 MEMO 的通用块和专用块，作者进行了进一步的讨论。实验表明，在增量学习中，冻结先前的专用块，使专用块之间互不影响，能使模型取得更好的性能。此外，通用块并不总是不需要更新，对于初始训练阶段基类数据过少的情况，通用块的可泛化性和可移植性不足以捕获特征表示，因此需要进行增量更新。

3.1.2 主干网络

主干网络（backbone network）是深度学习中用于特征提取的核心网络结构，通常用于处理输入数据（如图像）并提取其特征表示，以供后续任务（如分类、检测、分割等）使用。在图像识别领域，常用的主干网络有 AlexNet、VGG、GoogLeNet/Inception、ResNet 等。MEMO 在 DER 的基础上进行修改，沿用 DER 的主干网络，在 CIFAR100 上使用修改后的 ResNet18 作为

基准主干网络，而其他方法使用 ResNet32 作为基准主干网络。

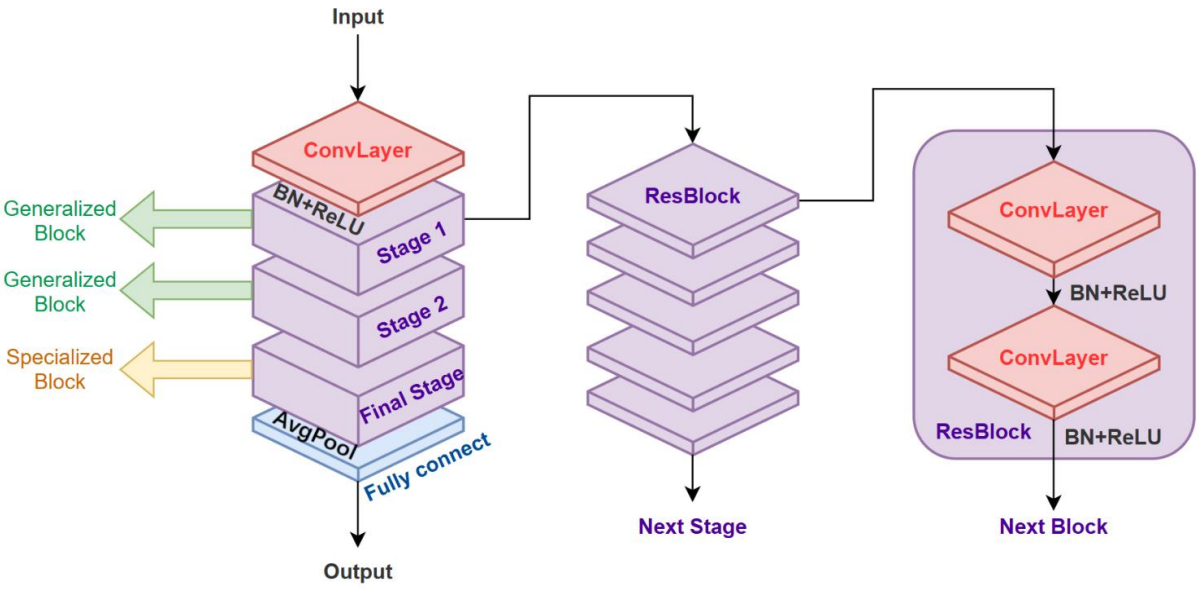


图 1 使用 ResNet32 作为主干网络的 MEMO 结构示意图

在实际应用中往往需要根据内存预算的大小选择不同规模的主干网络。

3.2 模型训练

3.2.1 数据下载、预处理和读入

接下来以数据集 CIFAR100，主干网络 ResNet32 为例说明项目的复现流程。在 Python 中使用 torchvision.datasets 可以便捷地下载 CIFAR100 数据集。CIFAR100 数据集中一共包含 5 万张训练图像和 1 万张测试图像，共 100 个类，每个图像由 RGB 三个通道，32 × 32 像素表示。

在数据集分割上，首先根据 CIL 中的常用设置，使用随机种子 1993 对训练类的类顺序进行打乱。根据 Base-x, Inc-y 的设置划分初始训练和增量训练的数据集。在训练过程中将数据集划分成分割成训练集和测试集，并在每个 epoch 中以 batch_size=128 为批次大小进行批量训练。

在读入数据之前，还需要对数据进行预处理以增强结果的鲁棒性。对于所有数据，将图像转换成 Pytorch 的张量（Torch）以方便后续训练，并且使用 CIFAR100 数据集的均值和标准差对图像进行标准化处理。对于训练集，额外进行数据增强操作：在边界填充 4 个像素宽的边缘然后随机裁剪图像至 32×32，随机水平翻转图像，随机调整图像的亮度。对于测试集，则不进行额外的处理操作。

经过以上操作后，每个 batch 读取到的数据的形状为(128, 3, 32, 32)。

3.2.2 初始训练

模型的训练过程可以分为两个阶段：初始训练阶段和增量训练阶段，在初始训练阶段主要训练通用块，在增量训练阶段则训练专用块。通用块随着模型的初始化一起初始化，而专用块和全连接层则在每个训练任务开始时初始化。专用块和全连接层的扩展是 MEMO 动态网络特性的体现。

在初始训练前，设置优化器和学习率调度器。使用带动量（0.9）的 SGD 优化器，学习率从 0.1 开始，在第 80 和 150 次训练时衰减至 0.01。初始训练设置为 200 个 epoch。

每个 batch 的输入数据形状为(128, 3, 32, 32)，则：

1. 初始卷积层和批归一化（BN）层：

输入经过一个卷积层（16 个输出通道，卷积核大小为 3x3），然后通过 BN 层和 ReLU 激活。输入形状从(128, 3, 32, 32)变为(128, 16, 32, 32)。

2. 第一阶段（Stage 1，通用块）：

包含 5 个基本残差块（“输入→卷积层→BN→ReLU→卷积层→BN→ReLU→输出”）。输入形状保持为(128, 16, 32, 32)。

3. 第二阶段（Stage 2，通用块）：

结构与第一阶段类似，但第一个基本残差块包含一次下采样（使用平均池化，将输入形状变为(128, 32, 16, 16)）。此后，输入形状保持为(128, 32, 16, 16)。

4. 最终阶段（Final Stage，专用块）：

结构与第二阶段相同，经过下采样后，输入形状变为(128, 64, 8, 8)。然后进行全局平均池化，得到形状为(128, 64)的特征值。

5. 全连接层：

特征值输入到一个（64，10）的全连接层，得到(128，10)的概率预测结果。

通过这些卷积层、批归一化层和 ResNet 基本块，输入特征图的空间维度逐渐减小，通道数逐渐增加，提取出更加抽象的高层次特征。在初始训练阶段使用(1)式作为损失函数调节网络参数。

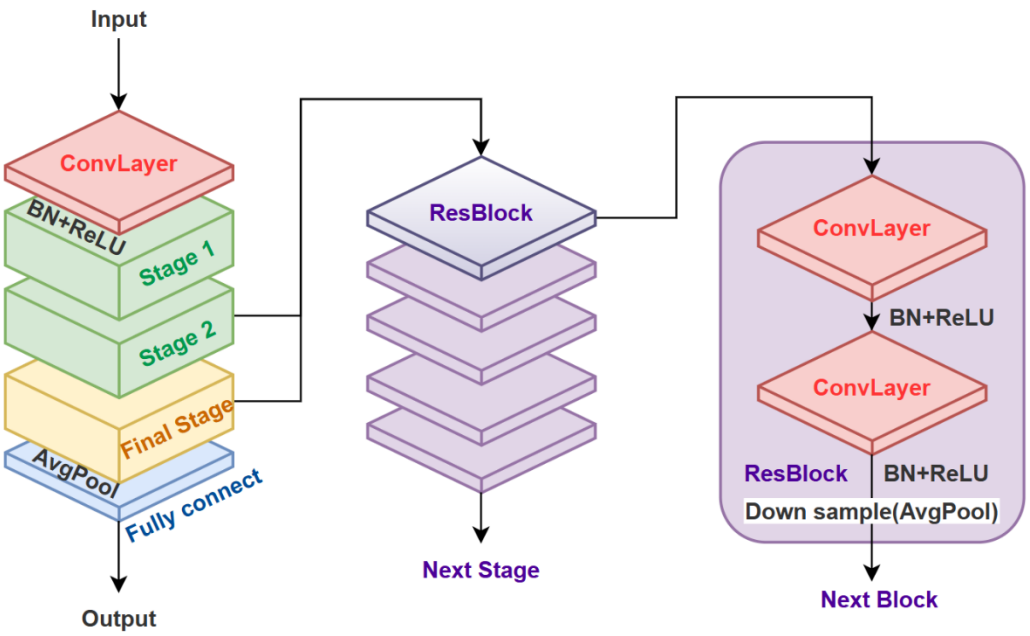


图 2 需要进行下采样的残差块位置和结构示意

3.2.3 构建范例

在每次训练任务结束后，需要对已知的类构建范例集，以保持对过往的类分类的准确性。MEMO 使用羊群算法从每个类中选择样本。羊群算法是一种用于选择代表性样本的技术，目的是在保持数据分布的同时减少数据集的规模。简单来说，在每次迭代中，羊群算法从类中选择一个样本，使样本与当前选择的样本集的质心最接近。当选取的样本数量达到预定数量或达到了某种停止标准时，算法停止。羊群算法的关键在于逐步逼近原始数据的分布，通过选择与质心最接近的样本，确保选取的样本集能够很好地代表原始数据集。

3.2.4 增量训练

增量训练阶段的优化器等设置与初始训练一致。增量训练设置为 170 个 epoch。

增量训练阶段需要不断扩展专用块和全连接层以使网络适应不断到来的新类。在阶段的一开始初始化当前任务所需的专用块，并将全连接层的输出维扩展至当前已知所有类的数量。此外，创建一个辅助全连接层，输出维数为当前任务类数加 1，用于提升模型在当前任务上的识别性能。在增量训练中专用块的数量不断增加，在增量训练阶段使用(3)式作为损失函数调节网络参数。

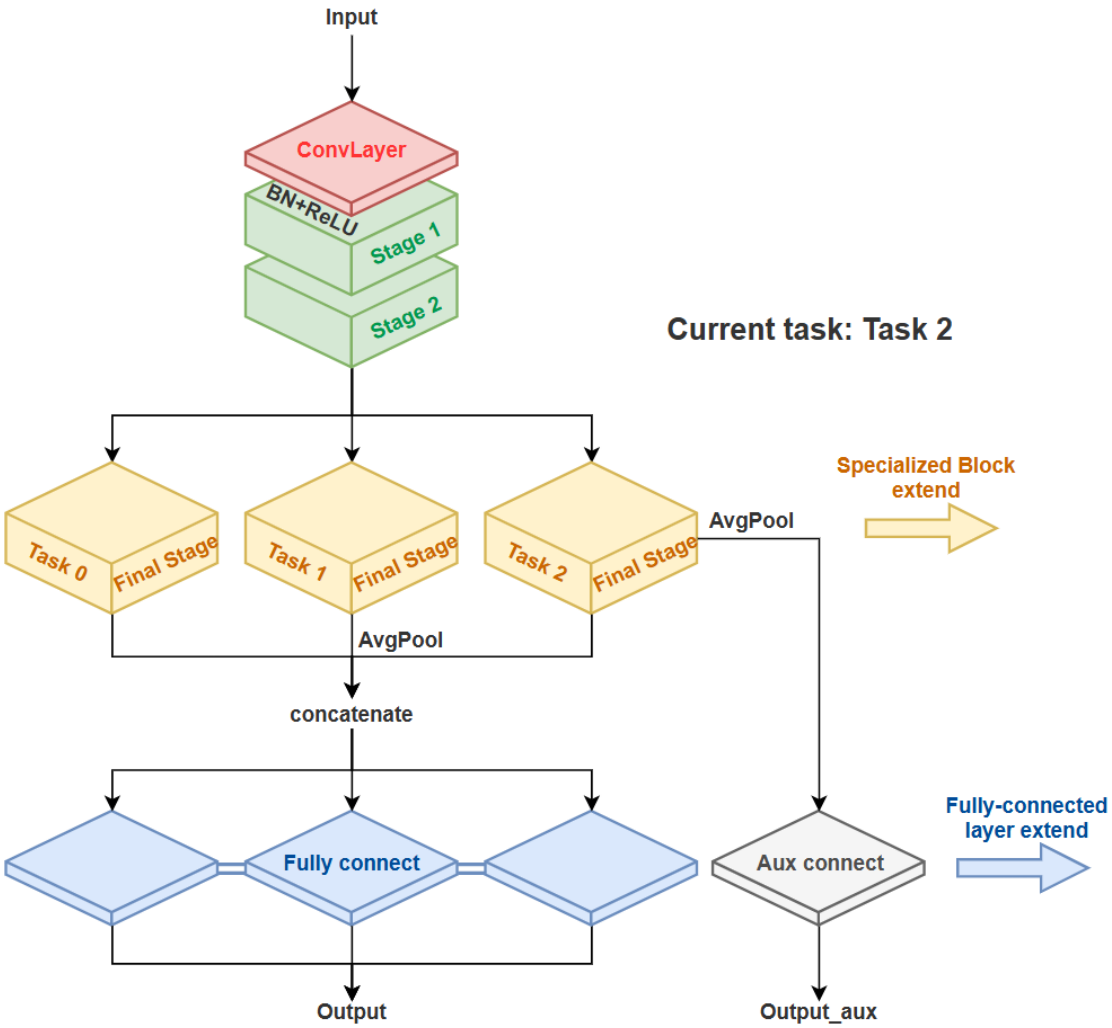


图 3 以当前任务为 Task2 为例示意专用块和全连接层的扩展以及辅助全连接层

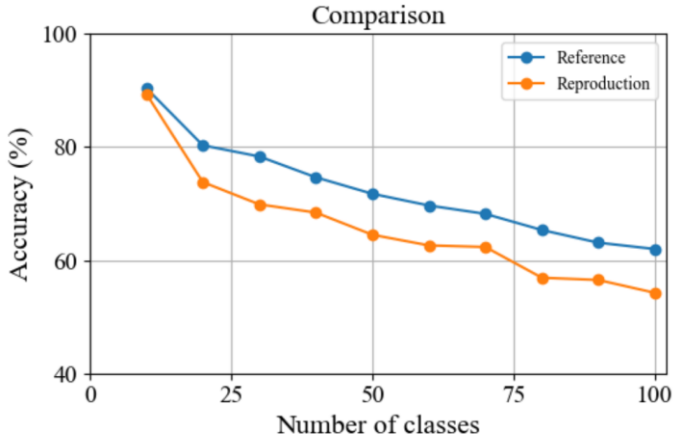
四、实验结果及分析

本次实验主要基于 CIFAR-100 数据集进行结果分析。以下为不同超参数（模型与内存大小）设置下训练 10 个 task 的实验结果与论文中结果的对比：

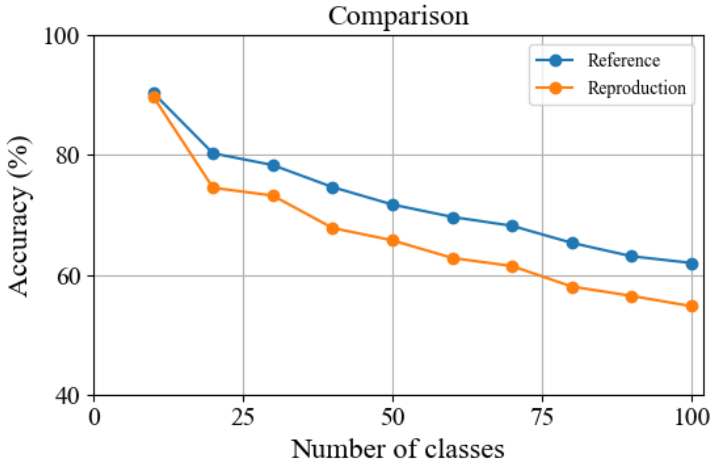
1. 我们首先探究训练次数对类增量图像分类效果的影响。使用 resnet32 模型，并设置内存大小为 3312：

--convnet_type resnet32 --memory_size 3312 --init_epoch 71 --epochs 51

cnn_top1:



nme_top1:

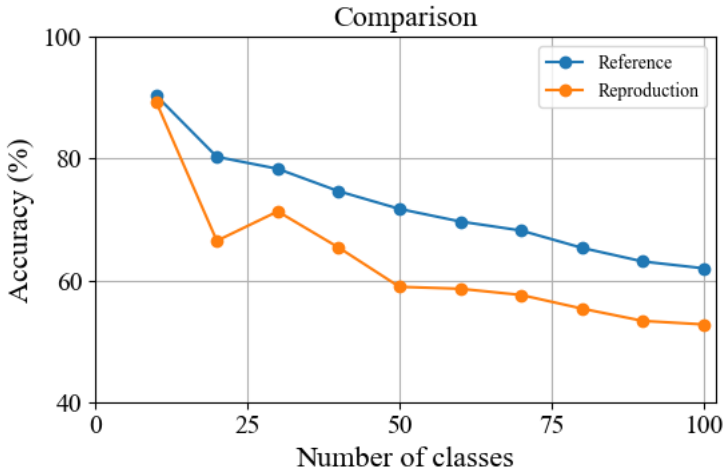


如图像所示，在减少训练次数（原论文为 200）后，在类别增量训练前期与原论文差别较小，准确率很高，达到 90%左右。随着类别的不断增加，实验结果与原论文结果差距增大，但差距仍低于 10%。由此可见，在减少训练次数后实验结果误差并无显著下降，但是节省了巨大的训练时间开销。

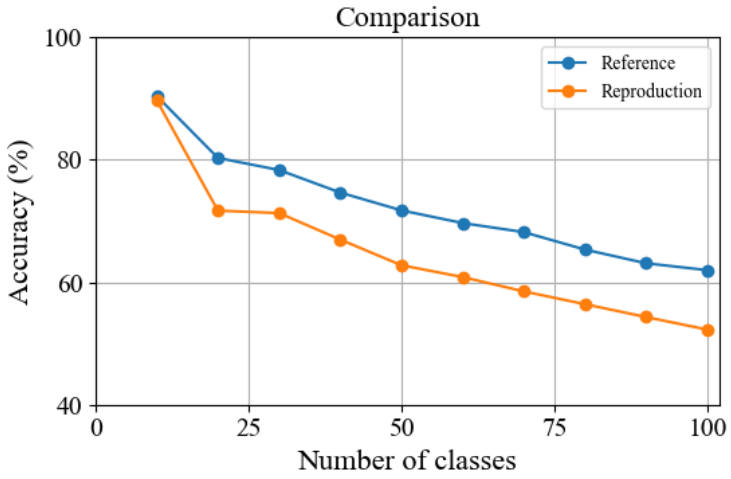
2. 为了验证内存大小对训练效果的影响，我们将内存大小限制在 2495 下进行实验：

--convnet_type resnet32 --memory_size 2495 --init_epoch 71 --epochs 51

cnn_top1:



nme_top1:



与 1.　中实验结果对比表

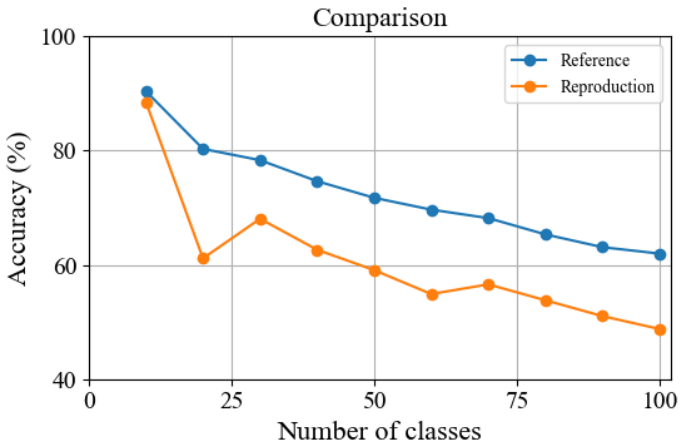
超参数设置/task 测试集准确率	0	1	2	3	4	5	6	7	8	9
--convnet_type resnet32 --memory_size 3312	89.4	73.8	69.87	68.42	64.5	62.62	62.34	56.91	56.51	54.24
--convnet_type resnet32 --memory_size 2495	89.4	66.5	71.33	65.4	58.94	58.62	57.6	55.36	53.34	52.76

由此可见，只减小内存大小对测试集准确率的影响很小。在类别增量训练最后的测试集准确率只下降了 1~2%。与原论文的实验结果准确率的差异也几乎与 1.　　中结果一致。体现出该模型在有限内存开销中的表现较好。

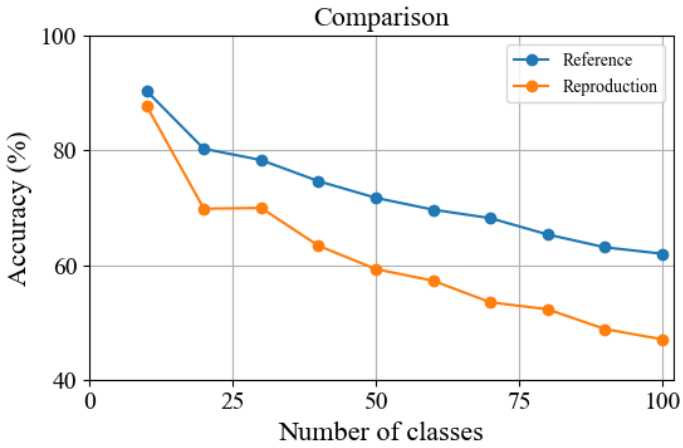
3.　我们随后尝试在进一步减小硬件开销的情况下进行实验。因此在将内存大小限制在 2495 的条件下，同时采用复杂度更低的模型 resnet14 进行实验：

--convnet_type memo_resnet14_cifar　--memory_size 2495 --init_epoch 71 --epochs 51

cnn_top1:



nme_top1:



与 1.　中实验结果对比表

超参数设置/task 测试集准确率	0	1	2	3	4	5	6	7	8	9
--convnet_type resnet32 --memory_size 3312	89.4	73.8	69.87	68.42	64.5	62.62	62.34	56.91	56.51	54.24
--convnet_type memo_resnet14_cifar --memory_size 2495	88.5	61.15	68.07	62.65	59.08	54.9	56.59	53.81	51.04	48.78

将两次实验的结果进行对比，我们发现在减小硬件开销的情况下，在进行类别增量训练前期测试集准确率差别较小。虽然最终测试集准确率小幅度下降 5%左右，但是训练时间也有所减少，并且与原论文的实验结果准确率相比仍在可接受范围内（10%左右）。

五、总结

MEMO 模型具有以下优点：

1. 通过保持模型的通用块，为不同的增量任务训练不同的专用块，从而适应不同的增量任务需求；
2. 通过维持每个类的样例应对灾难性遗忘问题；
3. 使用主交叉熵损失和辅助交叉熵损失结合的方法实现模型在旧类和新类之间的性能平衡；
4. “通用块-专用块”结构设计节约了内存中的模型占比，从而有更多的内存空间存储样例，具有较高的内存效率。

在实验报告中，我们并没有过多地直接照搬原论文作者的理解，而是根据自己的理解重新定义了类增量学习和损失函数。并且根据自己的理解制作了一些 MEMO 的结构和运作示意图。

我们对原作者的代码进行了小幅度的修改（比如 torch 版本兼容，变量类型调整等），以使得代码能够在我们的电脑环境上正常运行。受限于精力和算力限制，我们并没有进行太多的优化以及复现实验，这是一大遗憾。

六、小组分工

赖继杰：问题建模、项目分析和总结、代码编写和上传、撰写实验报告（除第四部分以外的所有部分，3364 字）、制图、制作 ppt 及演示

张昊昀：实验结果采集、撰写实验报告（第四部分，697 字）