河南工業大學

智能信息处理

大作业

题　　目：基于 Pytorch 框架完成语音指令训练大作业

院系名称：信息学院　专业班级：电信 2101 班

学生姓名：陈闯　学　　号：211040200102

2024 年 05 月 06 日

摘要

从智能语音助手，到复杂的搜索引擎，再到用户服务聊天机器人，NLP 已稳坐创新技术的前沿位置。这使得如《谷歌新语音指令训练识别》这样的主题变得关键重要。毫无疑问，语音识别技术席卷全球科技界，深入我们的商业行业，家庭生活，甚至是我们的休闲时光。此次课题旨在，使用 Pytorch 这一导向，以探索新的语音识别训练方法。我们借助 Pytorch 这个卓越的深度学习框架，选取了 DenseNet，Vgg，ResNext，和 WidereseNet 四种网络模型，进行了深入的理论和实践调研。在对比实验中，我们得出 ResNext 网络对新谷歌新语音指令训练识别的精度比其他三个经典网络更好。通过对比、研究和理解不同网络模型对于语音指令训练识别的表现，我们进一步对每个网络的优点，挑战和适用场景有了更深入的理解。因此，本研究不仅仅是一份对比研究报告，其中包含的学术思考和实践数据均将为语音识别领域提供有益的启示。本文中所涉及的相关核心代码已上传至：https://github.com/Gan1Ser/NLP-GoogleVoicebyPytorch.git。

关键字：Pytorch 谷歌新语音指令 深度学习 DenseNet Vgg WidereseNet Vgg

目录

1 引言

在信息科技发展的今天，自然语言处理(NLP)已经成为人工智能领域的一个重要分支，它在我们的生活中越来越显著，以至于我们很难想象没有它我们的生活会怎样。从智能语音助手，到复杂的搜索引擎，再到用户服务聊天机器人，NLP 已稳坐创新技术的前沿位置。这使得如《谷歌新语音指令训练识别》这样的主题变得关键重要。毫无疑问，语音识别技术席卷全球科技界，深入我们的商业行业，家庭生活，甚至是我们的休闲时光。然而，这个科技巨擘之所以能够如此深入，是因为它拥有让复杂的事务变得简单的力量。不过，学习、发展并确保这项技术的表现最佳，依然需要我们的细心研究和检验。此次课题旨在，使用 Pytorch 这一导向，以探索新的语音识别训练方法。我们借助 Pytorch 这个卓越的深度学习框架，选取了 DenseNet，Vgg，ResNext，和 WidereseNet 四种网络模型，进行了深入的理论和实践调研。首先，DenseNet 是一个深度卷积网络，它以其独特的连接方式赢得了科研界的关注。DenseNet 的独特之处在于它在网络中引入了全连接的线性路径，以解决信息在网络中传播消失的问题。其次，经典的 VGG 模型以其深层次卷积网络和相对简洁的构造方式，在多个图像识别任务上获得了优秀的表现。接下来介绍的是 ResNext，它是 ResNet 的一个变体，ResNext 在模型架构上引入分组卷积，提高了网络的容量，优化了信息流动，减少了计算量。最后，我们引入了 WideResNet 模型。WideResNet 是通过增加网络的宽度，来提升网络性能的尝试。相比于经典的深度网络，WideResNet 提供了一种新的思考角度，即通过增加网络中每层的神经元数量，来提升模型的性能。这四个模型都在深度学习领域有着广泛的应用，而我们尝试引入他们进行语音识别的训练，这是一种勇敢和有创新的尝试。在对模型进行深入研究和实践后，我们期待找到适合于我们课题的模型，从而推动 NLP 领域，特别是语音识别的发展。

通过对比、研究和理解不同网络模型对于语音指令训练识别的表现，我们进一步对每个网络的优点，挑战和适用场景有了更深入的理解。因此，本研究不仅仅是一份对比研究报告，其中包含的学术思考和实践数据均将为语音识别领域提供有益的启示。期待通过我们的探索，将语音识别技术推向新的高度，为这个领域带来新的思考和突破。让我们一起创新、探索并实现科技对于生活的改变，让未来的语音技术更好地服务于我们的生活。

2 框架及网络介绍

PyTorch 是一个开源的深度学习平台，它为研究和开发提供了强大的适应性和灵活性。PyTorch 特别支持动态计算图，同时提供丰富的 API，可以方便用户进行快速原型设计和高效的扩展。因此，无论是从事学术研究，还是从事实用项目的开发，PyTorch 都能够提供强大的支持。此次课题基于 Pytorch 框架完成。

2.1 Pytorch 网络框架

PyTorch 是一个开放源码的深度学习框架，由 Facebook 的人工智能研究小组开发，被广大研究者和开发者应用于计算机视觉，自然语言处理等多个领域。自 2017 年推出以来，PyTorch 凭借其易用性和灵活性，迅速在深度学习领域获得了极高的人气。它具有动态计算图的特性。计算图是描述深度学习模型中运算和数据之间关系的一种方式。与常见的静态计算图框架如 TensorFlow 等不同，PyTorch 中的计算图在每次前向传播过程中都是动态生成的，这使得模型的调试更为方便，同时也为复杂的模型和动态控制流提供了可能。并且，PyTorch 提供了丰富的 API 和工具，如 Autograd 模块用于自动求解梯度，nn 模块提供大量预定义的深度学习层，以及大量预训练模型等。这些使得使用 PyTorch 进行快速原型设计和调试更为方便。当然，PyTorch 也具有强大的扩展性。用户可以通过自定义 C++或 CUDA 函数来进行低级别的扩展。同时，PyTorch 因为其 Python 背景，使得其易于与其他的 Python 库，如 NumPy、SciPy 等进行交互。其优秀的多 GPU 支持和分布式支持，利用多进程和 CUDA 流的方式实现了数据并行，使得在多个 GPU 上进行训练模型变得十分方便。同时，PyTorch 还提供了一套完整的分布式训练方式，支持包括数据并行、模型并行和分布式参数服务器等多种方式。

PyTorch 在不断的升级和完善中，提供了更为高效和精细的模型保存和加载方式，更好的支持了模型的部署。同时还提供了可视化工具，如 TensorBoard 的支持和自家的可视化库 Visdom，使得训练过程的可视化更为方便。PyTorch 以其独特的优势，在深度学习领域中脱颖而出，为深度学习的研究和应用提供了一个优秀的平台。无论是做研究，还是做实际的开发，PyTorch 都能提供强大的支持，它不仅方便了训练模型，而且也使得部署模型更加的快捷和方便。因此，选择 PyTorch 作为深度学习的工具，无疑是一个明智的决定。

2.2 CNN 卷积神经网络

卷积神经网络（Convolutional Neural Networks，简称 CNN）是一种深度学习的算法，在图像和视频处理领域有广泛应用。它的主要特征是能有效地处理具有网格结构的数据（例如，像素构成的图像）。

对于卷积的定义，其连续形式如下$(f \times g)(n) = \int_{-\infty}^{\infty} f(\tau)g(n-\tau)d$，其离散形式如下：$(f \times g)(n) = \sum_{\tau=-\infty}^{\infty} f(\tau)g(n-\tau)$。上述公式中有一个共同的特征：$n = \tau + (n-\tau)$。对于这个特征，可以令$x = \tau, y = n - \tau$，那么$x + y = n$就是一些直线如下图所示 2-1。

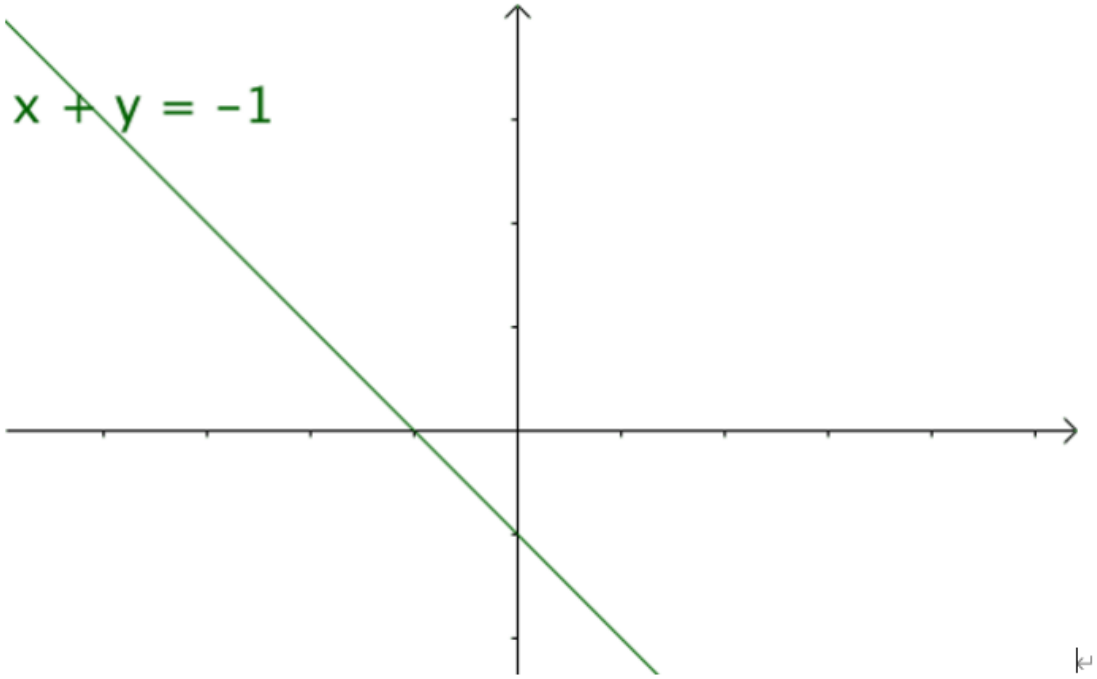


图 2-1 卷积定义

采用卷积提取图像特征，卷积核和图像进行点乘（dot product), 就代表卷积核里的权重单独对相应位置的 Pixel 进行作用。这里所说的点乘，虽说我们称为卷积，实际上是位置一一对应的点乘，不是真正意义的卷积。比如图像位置（1,1）乘以卷积核位置（1,1），仔细观察右上角你就会发现。一般而言卷积神经网络的构造如下图 2-2 所示，其中输入层接收原始图像数据。卷积层将输入数据与卷积核进行卷积操作。然后，通过应用激活函数（如 ReLU）来引入非线性。这一步使网络能够学习复杂的特征。池化层通过减小特征图的大小来减少计算复杂性。它通过选择池化窗口内的最大值或平均值来实现。这有助于提取最重要的特征。CNN 通常

由多个卷积和池化层的堆叠组成，池化运算是一种下采样技术，用于减少数据的维度和计算量，同时保留重要的特征信息。在 CNN 中，池化运算通常被应用于卷积运算之后，对卷积运算得到的特征图进行降采样，以逐渐提取更高级别的特征。深层次的特征可以表示更复杂的模式。最后，全连接层将提取的特征映射转化为网络的最终输出。这可以是一个分类标签、回归值或其他任务的结果。

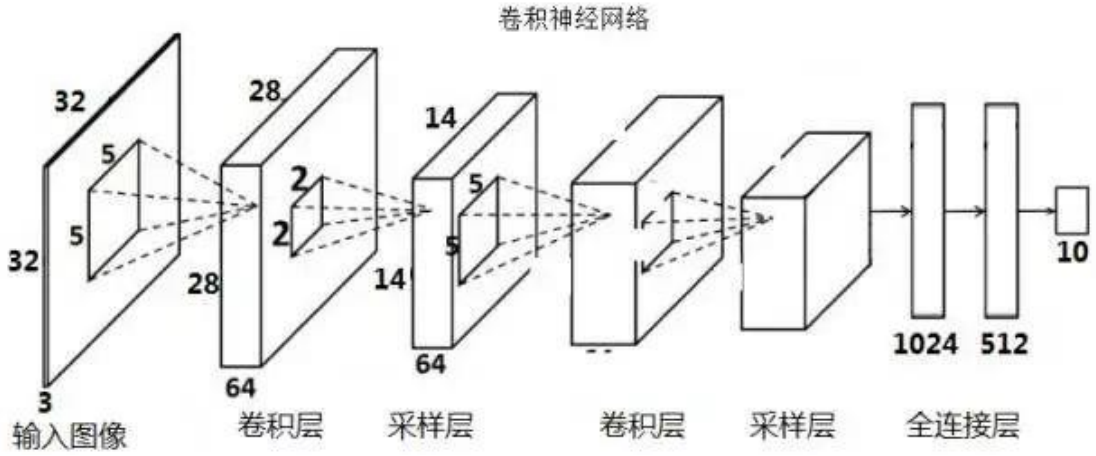


图 2-2 卷积神网络基本结构

CNN 的训练方法主要是基于反向传播算法和梯度下降算法。在训练过程中，CNN 会根据输入数据和对应的目标标签进行学习，通过不断调整权重参数来最小化损失函数。具体来说，CNN 的训练过程包括前向传播和反向传播两个步骤。在前向传播过程中，输入数据经过多个卷积层和池化层的处理后得到输出结果；在反向传播过程中，根据输出结果和目标标签的误差来计算损失函数，并使用梯度下降算法来更新权重参数。通过不断迭代训练过程，CNN 可以逐渐学习到输入数据的特征表示和分类能力。CNN 采用了权重共享的策略，同一个滤波器在整个输入图像或特征图上移动，这大大减少了模型的参数数量，提高了模型的训练效率。同时，卷积的操作方式使得 CNN 具有对平移的不变性，这对许多视觉任务是非常重要的。在过去的几年里，许多基于 CNN 的模型被提出，比如 LeNet，AlexNet，VGG，GoogLeNet， ResNet 等，这些模型在许多视觉任务中取得了领先的表现。同时 CNN 也被拓宽应用到其他领域，例如自然语言处理，时间序列分析等，显示出其强大的能力。

2.3 DenseNet 网络介绍

DenseNet，即 Densely Connected Convolutional Networks，是一种深度卷积网络架构，由 Cornell University, Tsinghua University 和 Facebook AI Research 在 2017 年提出。DenseNet 的核心思想是“密集连接”，即在网络中，每一层都与前面所有层直接相连，与后续的所有层也直接相连。这种设计促进了特征的复用，也就是一个特征在网络中可以被多次使用，提高了信息流动的效率。其连接方式如下图 2-3 所示。

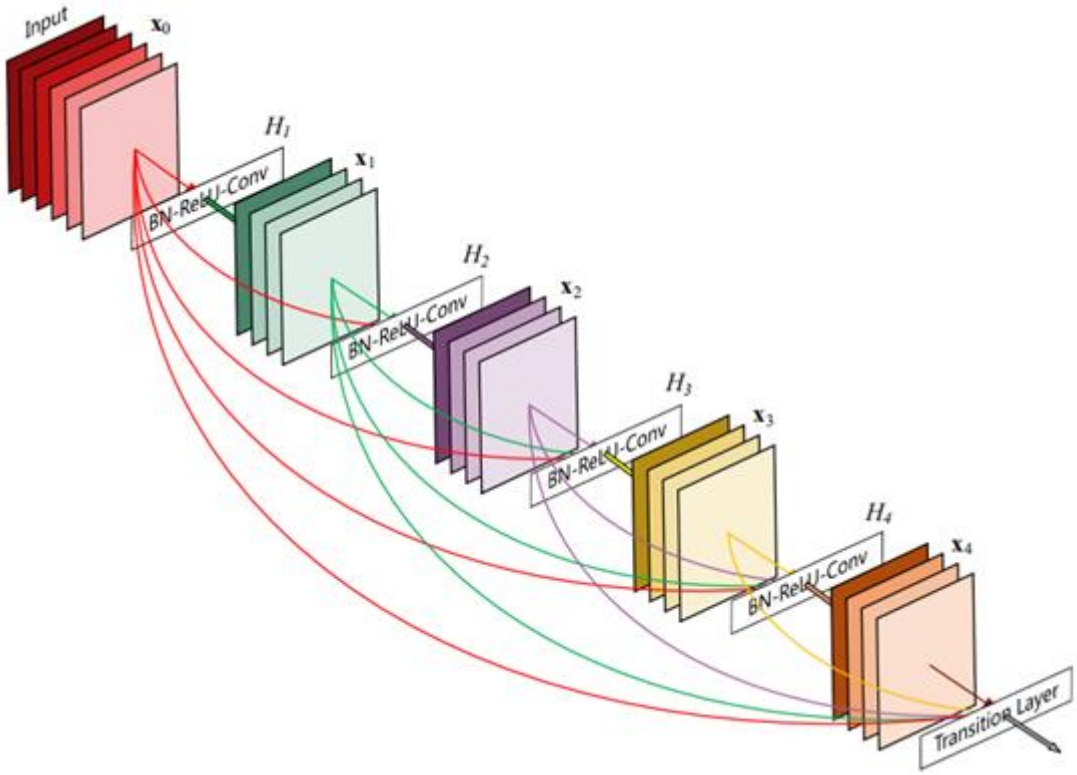


图 2-3 DenseNet 连接方式

标准的 L 层卷及网络有 L 个连接，即每一层与他的前一层和后一层连接，而 DenseNet 将前面所有层与后面层连接，故有$(1+2+\cdots+L)*L=(L+1)*L/2$个连接。DenseNet 主要有以下特点：①增强的特征传播；②特征复用；③参数效率高；④优秀的性能。当然其具有的缺点也十分明显：①计算资源需求大；②计算复杂度高；③设计挑战；④过拟合风险。

DenseNet 是一种强大的深度学习模型，但也需要根据具体的任务需求和资源条件去权衡其使用。下图 2-4 是 DenseNet 网络的结构参数，增长率 K=32，采用 pre-activation，即 BN-ReLU-Conv 的顺序。以 DenseNet-121 为例，其网络构

成如下：DenseNet-121 由 121 层权重层组成，其中 4 个 Dense block，共计 2×(6+12+24+16)＝116 层权重，加上初始输入的 1 卷积层+3 过渡层+最后输出的全连接层，共计 121 层；练时采用了 DenseNet-BC 结构，压缩因子 0.5，增长率 k = 32；初始卷积层有 2k 个通道数，经过 7×7 卷积将 224×224 的输入图片缩减至 112×112；Denseblock 块由 layer 堆叠而成，layer 的尺寸都相同：1×1+3×3 的两层 conv（每层 conv = BN+ReLU+Conv）；Denseblock 间由过渡层构成，过渡层通过 1×1 卷积层来减小通道数，并使用步幅为 2 的平均池化层减半高和宽。最后经过全局平均池化 + 全连接层的 1000 路 softmax 得到输出。

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv; 3 × 3 conv] × 24	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 48	[1 × 1 conv; 3 × 3 conv] × 64
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv; 3 × 3 conv] × 16	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 48
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

图 2-4 DenseNet-121 网络结构参数

2.4 Vgg 网络介绍

VGG 网络是由牛津大学的视觉几何组（Visual Geometry Group）所提出，在 2014 年的 ImageNet 挑战中取得优秀的成绩并搅乱了深度学习领域的格局。其主要思想是采用连续的几个 3x3 的卷积核替代较大尺寸的卷积核（比如 7x7、11x11）。例如，三个连续的 3x3 卷积层的感受野与一个 7x7 的卷积层相同。这样的设计能够减少参数，同时增强了网络的拟合能力。VGG 网络最常见的两种结构是 VGG16 和 VGG19，分别包含 16 层（13 个卷积层和 3 个全连接层）和 19 层（16 个卷积层和 3 个全连接层）。VGG16 相比 AlexNet 的一个改进是采用连续的几个 3x3 的卷积核代替 AlexNet 中的较大卷积核（11x11，7x7，5x5）。对于给定的感受野（与输出有关的输入图片的局部大小），采用堆积的小卷积核是优于采用大的卷积核，因为多层非线性层可以增加网络深度来保证学习更复杂的模式，而且代价还比较小（参数更少）其连接方式

如下图 2-5。

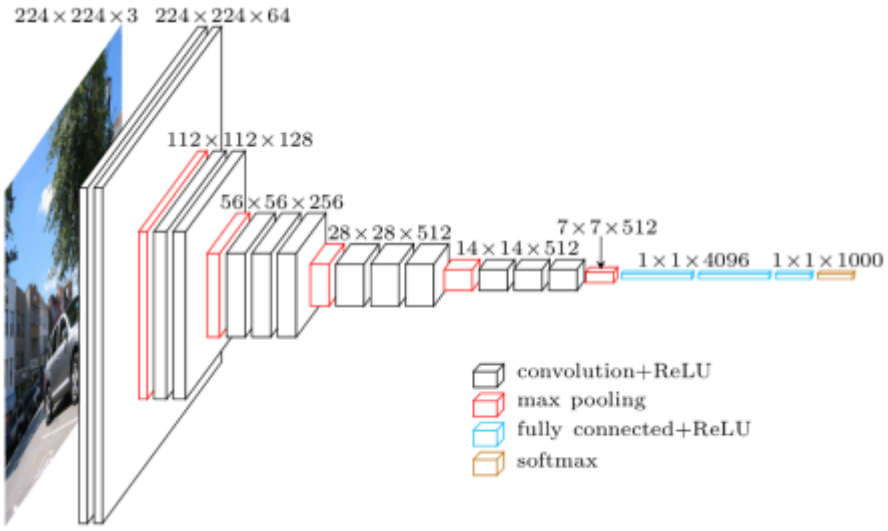


图 2-5 Vgg16 网络连接方式

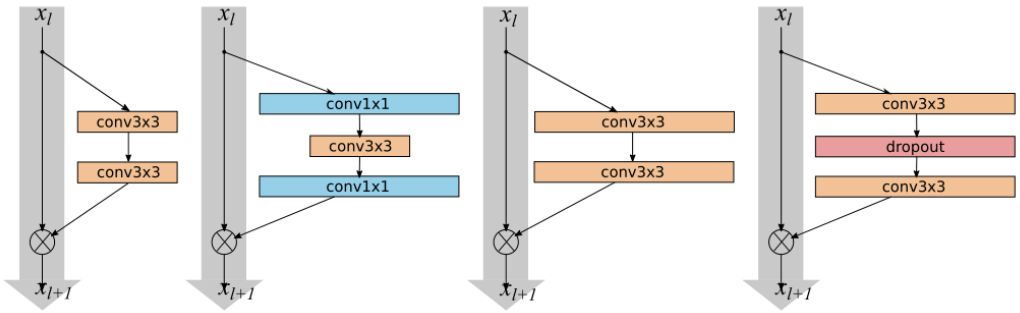
VGG 的结构与 AlexNet 类似，区别是深度更深，但形式上更加简单。VGG 由 5 层卷积层、3 层全连接层、1 层 softmax 输出层构成，层与层之间使用 maxpool（最大化池）分开，所有隐藏层的激活单元都采用 ReLU 函数。作者在原论文中，根据卷积层不同的子层数量，设计了 A、A-LRN、B、C、D、E 这 6 种网络结构配制如图 2-6 所示。

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

图 2-6 VGG 网络中提供的 6 种网络配置

2.5 WideResNet 网络介绍

Wide Residual Networks（WRN，即宽残差网络）是一种改进的深度残差网络（ResNet）。基本上，WRN 的设计思想是通过增加网络的宽度（也就是每一层的通道数），而非深度，来提高网络的性能。与原有的 ResNet 相比，WRN 具有更少的深度和更多的宽度。WRN 的结构基本上与 ResNet 类似，都包括短路连接和残差块。在 WRN 中，一般明显增加每个残差块中卷积层的过滤器数量。例如，如果我们提高了每个残差块的宽度因子 k，那么每个残差块中的过滤器数量就可以从 64，128，256 提高到 64k，128k，256k。其中宽度因子，在宽残差网络（WRN）中，宽度因子 k 是决定每个残差块中卷积层过滤器数量的关键参数。将 k 设置为更大的值，意味着每个残差块中卷积层的过滤器会更多，也就是说，网络的宽度会更大。这种改变主要带来两个好处：①减少网络深度：增加网络宽度可以减小网络深度，从而减少了模型梯度消失和梯度爆发的风险，使模型训练更稳定。②提高性能：无需增加额外的网络深度或连接性，而是通过简单地使网络变宽，WN 可以大幅度提高预测性能。但增加网络的宽度也可能增加了计算量，以及模型的复杂性和过拟合的风险。WRN 的连接方式如下图 2-7 所示。



(a) basic (b) bottleneck (c) basic-wide (d) wide-dropout

Figure 1: Various residual blocks used in the paper. Batch normalization and ReLU precede each convolution (omitted for clarity)

图 2-7 WRN 网络结构单元

在上述结构单元中，a 是最基本的 ResNet 结构，b 是用了 bottleneck（瓶颈）的 ResNet 结构，d 是在最基本的 ResNet 结构上加上 Dropout 层的 WideResNet 结构。增加 Conv 的 Output channels 数目即使用更多的 conv filters 进行计算，所谓的增宽 block。Residual block 里面使用的 conv 层次结构，设 B（M）表示剩余块结构，其中 M 是块中卷积层的核大小列表。例如，B（3,1）表示具有 3×3 和 1×1 两个卷积

层的剩余块，B(3,1,1)表示 3×3 和 1×1 和 1×1 三个卷积层组成，以此类推；作者做实验设计了几个不同的 conv 层次，以此来验证 residual block 中最佳的 conv 结构。设计的不同 conv 层次结构具体参数如下图 2-8 所示。

block type	depth	# params	time,s	CIFAR-10
$B(1,3,1)$	40	1.4M	85.8	6.06
$B(3,1)$	40	1.2M	67.5	5.78
$B(1,3)$	40	1.3M	72.2	6.42
$B(3,1,1)$	40	1.3M	82.2	5.86
$B(3,3)$	28	1.5M	67.5	5.73
$B(3,1,3)$	22	1.1M	59.9	5.78

Table 2: Test error (%, median over 5 runs) on CIFAR-10 of residual networks with $k = 2$ and different block types. Time column measures one training epoch.

图 2-8 WRN 网络不同 conv 层次结构参数

2.6 ResNext 网络介绍

ResNeXt 是一种基于 ResNet 的变种网络结构，由 Facebook 的研究团队在 2016 年提出，目标是使深度神经网络更易于设计和扩展。ResNeXt 的名字源自"ResNet"和"next"，寓意着这是对 ResNet 的下一步改进。其关键思想是引入了"群卷积"（group convolution）的概念。这是一种将输入分为多个组然后进行卷积操作的方式，最后将结果重新结合在一起。这种设计方式可以在保持网络模型复杂性不增加的同时增加模型的容量，以便模型学习到更多的模式和特征。具体来说，ResNeXt 可以看作是将 ResNet 的残差块换成了包含多个并行卷积路径的新型残差块。这些并行路径可以被看作是网络中的“子网络”或“模块”，每个路径包含了不同的卷积操作。所有路径的输出会在最后进行加权求和，形成这个新型残差块的输出。如此设计的优点是，这些路径可以并行处理，并且每个路径可以专门学习到一种特定的特征，增强了模型的表达能力。而且，相比于原始的 ResNet，ResNeXt 可以在保持参数数量不变的情况下提供更好的性能。在 ResNext 的结构设计中，假设输入有 256 个通道,接着使用两组 1x1 卷积，将 256 维输入拆分 Split 为低维数据（例如 3x3 分支通道数为 160，5x5 分支通道数为 128），转换 Transform 它们（即分别应

用 3x3 和 5x 5 卷积), 最后通过合并 merge 它们（合并通道数 288=160+128), 这样会得到很好的结果。而 ResNeXt 则追求一种架构，该架构利用 Inception 的拆分-转换-合并策略，同时牢记 VGG 重复具有相同结构的 block 块和 Resnet 跨层分支的理念。如图 2-9 为一个 ResNeXt 块结构，具有许多相同分支的模块,每个分支将输入进行通道裁剪,然后通过 3x3 卷积进行转换,最后将结果进行合并.其中, 分支的数量被称为 cardinality，在图中为 32。cardinality 被视为神经网络(ResNeXt)除了深度和宽度外的另一个维度。

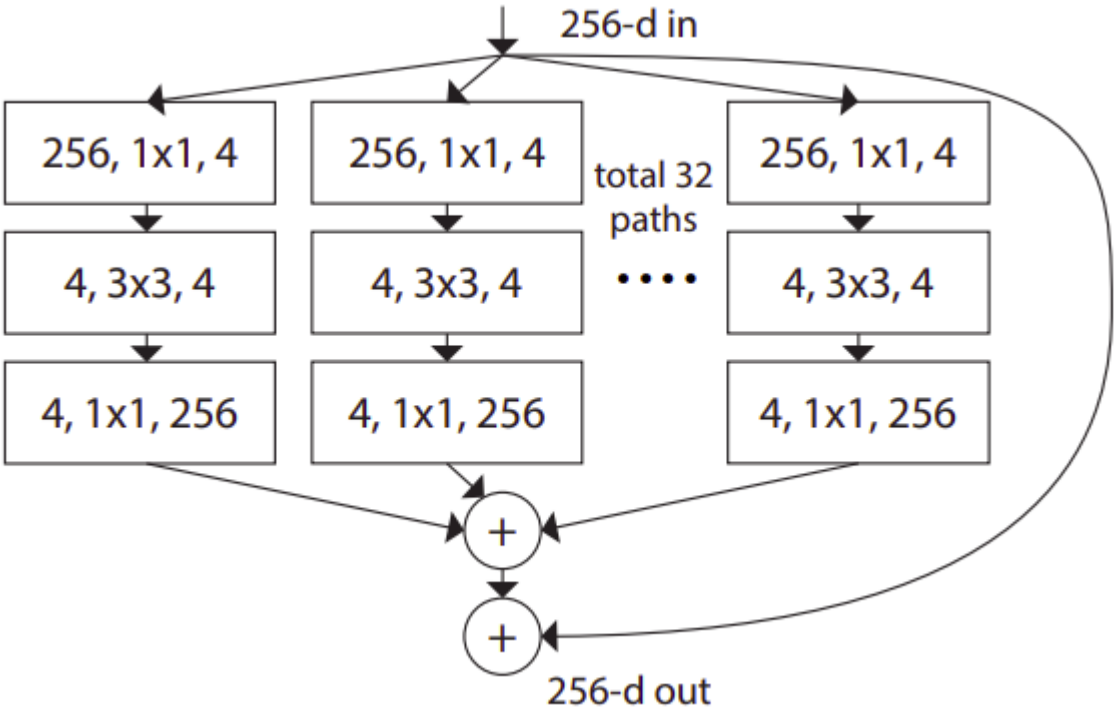


图 2-9 一个 ResNeXt 块结构

深度学习网络模型 VGG、ResNet，以及两种在 ResNet 基础上优化改进的模型 WRN 和 ResNeXt，各具特色和优势，同时也存在着一定的局限性。VGG 模型以其极其简洁的架构和优秀的特征提取能力在深度学习的早期应用中发挥了重要作用，但其在参数量和计算复杂性方面的较大需求划定了性能的上限。ResNet 通过引入残差模块和跳跃连接以增加网络深度，有效克服了深层网络训练困难的问题，提高了网络的性能。而 ResNeXt 更进一步，以组的思维方式对网络进行设计，既节省了计算资源，又凸显出其在处理复杂问题时的优越性能。WRN 则试图通过增加网络的宽度来提升性能。这些模型在深度学习领域的发展过程中，都起到了不可忽视的关键作用。

3 谷歌语音指令训练识别实现方案

训练神经网络模型一般按照以下步骤进行：①参数解析；②数据预处理和加载；③模型定义；④定义损失函数和优化器；⑤训练和验证循环；⑥学习率调整；⑦记录和保存；⑧测试预测本地语音。

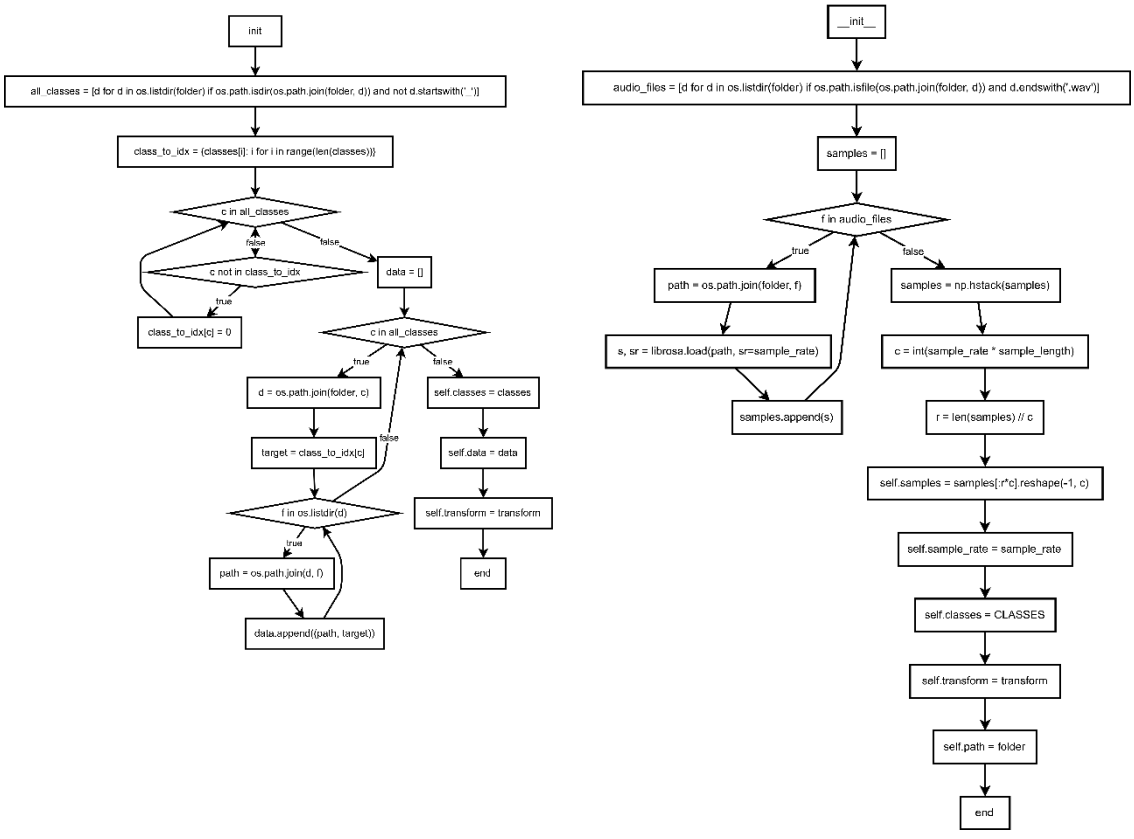
3.1 数据预处理

为保证模型具有很好的泛化能力以及鲁棒性，对新语音指令数据集进行预处理是有必要。为完成数据预处理工作，预处理文件定义了两个类，一个是SpeechCommandsDataset，另一个是 BackgroundNoiseDataset，它们都是继承自torch.utils.data 的 Dataset 类，用于数据集的分割以及加噪。SpeechCommandsDataset用于处理语音命令数据集，是将语音数据转化为可以输入模型的数据结构。在初始化时，首先定义所有的类别 all_classes，然后为每一个类别赋予一个索引值class_to_idx，并以此构造出数据集 data，每一个元素包括路径 path 和对应的目标类别 target。__len__函数返回的是数据集的长度，也就是所有元素的数量。__getitem__函数则是返回索引为 index 的数据和相对应的标签，如果定义了变换函数 transform，就会对数据和标签进行这个变换。make_weights_for_balanced_classes函数则是创建每个类的样本权重，在训练过程中将根据这个权重对不同类别的样本进行重采样，以解决类别不均衡的问题。其结构流程图如下图 3-1a 所示。

BackgroundNoiseDataset 类则是处理背景噪音数据。它将.wav 文件加载并缩放至指定的采样率，每个样本的长度为 sample_length 的声音片段。整个数据集被一维化并重塑为(-1, c)的形状。主要用于处理和加载背景噪音数据。当初始化这个类时，会首先在指定的文件夹 folder 中查找所有以".wav"为扩展名的音频文件。然后，使用 librosa.load 函数加载这些音频文件，并以设定的采样率 sample_rate 对音频进行重采样。将所有加载的音频文件合并到一个大的 NumPy 数组 samples 中。接下来，为了让每个样本都有固定的长度，会将音频的长度调整为整数倍的sample_length 秒（这是通过乘以采样率得到的样本数量 c）。对于加载的音频，如果其长度超过了 sample_length 秒，多余的部分将会被舍弃。然后，剩余的音频数据将会被重新塑造到一个形状为(-1, c)的数组中，即，每行代表一个长度为sample_length 秒的样本。在__getitem__方法中，会返回一个包含数据和标签的词

典。数据 samples 是实际的音频数据，标签 target 默认为 1，表示这是背景噪音样本。如果设置了 transform 函数，将会对返回的数据进行转换。数据增广可以扩大数据集，并构造更多样化的训练样本，例如这里通过变换振幅，变速和音调等手段，能够生成多样化的语音数据。这些在原始数据中不存在的数据为模型提供了更多样化的训练环境，能够帮助模型提高对语音数据的识别能力，从而提高模型的泛化能力故 BackgroundNoiseDataset 类是通过加载和处理背景噪音.wav 文件，使得每个音频样本对应固定时长（sample_length 秒）的音频数据，并设定了默认标签为背景噪音。这样就构造出了一个背景噪音的数据集，可以用于训练模型来识别或者抑制背景噪音。其结构流程图如下图 3-1b 所示。

以上方法实现了从原始的语音命令数据文件和背景噪声文件中预处理和提取数据集的过程，使得数据符合 PyTorch 深度学习框架的数据输入要求，让原始的音频数据可以被机器学习模型读取和使用。在这个过程中，实现了类别标签的索引化，数据的读取，转换，加载，和整合等功能，并且处理了样本不平衡问题，使得所有类别的样本可以按照均匀的比例进行训练。



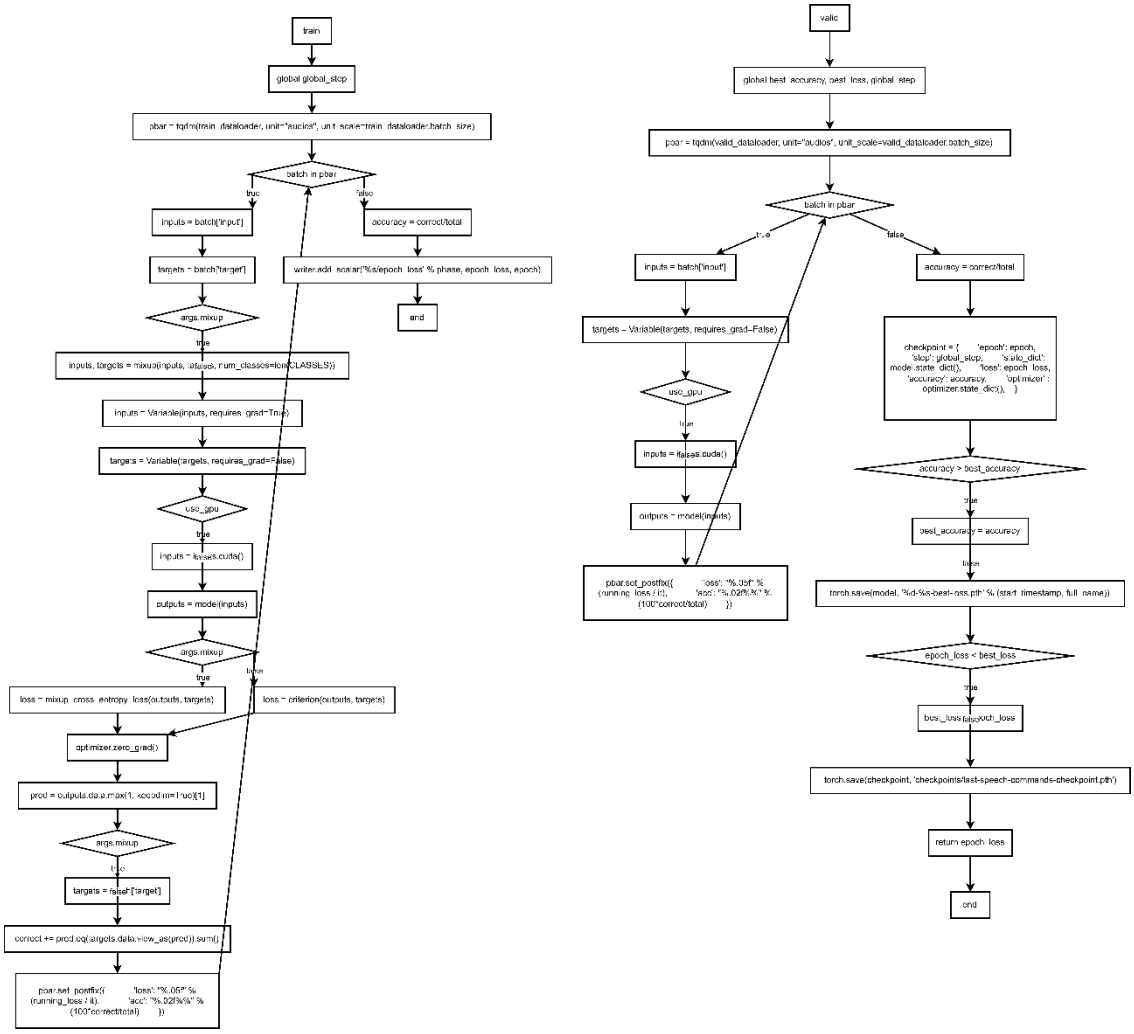
a. SpeechCommandsDataset 类流程　　b. BackgroundNoiseDataset 类流程

图 3-1 数据集预处理流程

3.2 网络训练实现

使用 PyTorch 库进行语音指令分类的深度学习模型训练的脚本。它包含了从数据加载，模型建立，到模型训练与验证的整个过程，其细节如下，参数解析：argparse.ArgumentParser 用于解析运行脚本时传递的命令行参数。例如，训练集和验证集的路径，模型类型，优化器类型，批次大小，学习率，学习率调整策略，最大训练周期等。数据加载和预处理：首先进行数据增广，包括改变音频的振幅，速度和音调，以及将音频长度固定为一秒。然后，加载了含有背景噪声的音频数据集，用于混合到语音指令样本中模拟真实环境。接下来，加载并处理训练集和验证集，先将音频转为梅尔频谱图，然后转为对应的 Tensor 数据格式。模型定义：根据参数选择相应的模型，并创建模型实例。如果计算机有支持 CUDA 的 GPU，模型将在 GPU 上进行训练。损失函数与优化器设定：定义了交叉熵损失函数 torch.nn.CrossEntropyLoss()；根据输入参数选择使用 SGD 或 Adam 优化器，并设定了优化器的学习率和权重衰减参数。加载先前训练的模型：如果提供了有效的模型检查点路径(--resume 参数)，则加载预训练模型的状态。学习率调度策略：根据参数选择使用 torch.optim.lr_scheduler.ReduceLROnPlateau（当一定周期内模型性能没有改善时，降低学习率）或 torch.optim.lr_scheduler.StepLR（每隔一定的训练周期就降低学习率）。训练以及验证的流程如下图 3-2a、图 3-2b 所示。

值得注意的是在深度学习模型训练过程中，调整学习率是常见的一种方法，有助于提高模型性能和加快训练速度。这段代码中，根据参数的选择，采用了两种常见的学习率调度策略：torch.optim.lr_scheduler.ReduceLROnPlateau：这种策略是当模型的验证误差在一定周期内没有改善时，就降低学习率。也就是说，如果模型在训练过程中，经过一定的周期(由 args.lr_scheduler_patience 决定)，模型性能没有提升，那么就会将学习率乘以一个系数(由 args.lr_scheduler_gamma 决定)来降低学习率。这种策略可以帮助模型在快速收敛到某个最小值后，通过减小学习率来更细致地在最小值附近寻找更好的解。torch.optim.lr_scheduler.StepLR：这种策略是每隔固定的训练周期就会降低学习率。具体来说，它会在每个 args.lr_scheduler_step_size 个 epoch 后，将学习率乘以 args.lr_scheduler_gamma，以此降低学习率。这种策略可以在初始阶段让模型以较大的学习率快速收敛，然后逐步减小学习率使得模型可以更平滑地逼近最优解。



a. 训练流程　　b. 验证流程

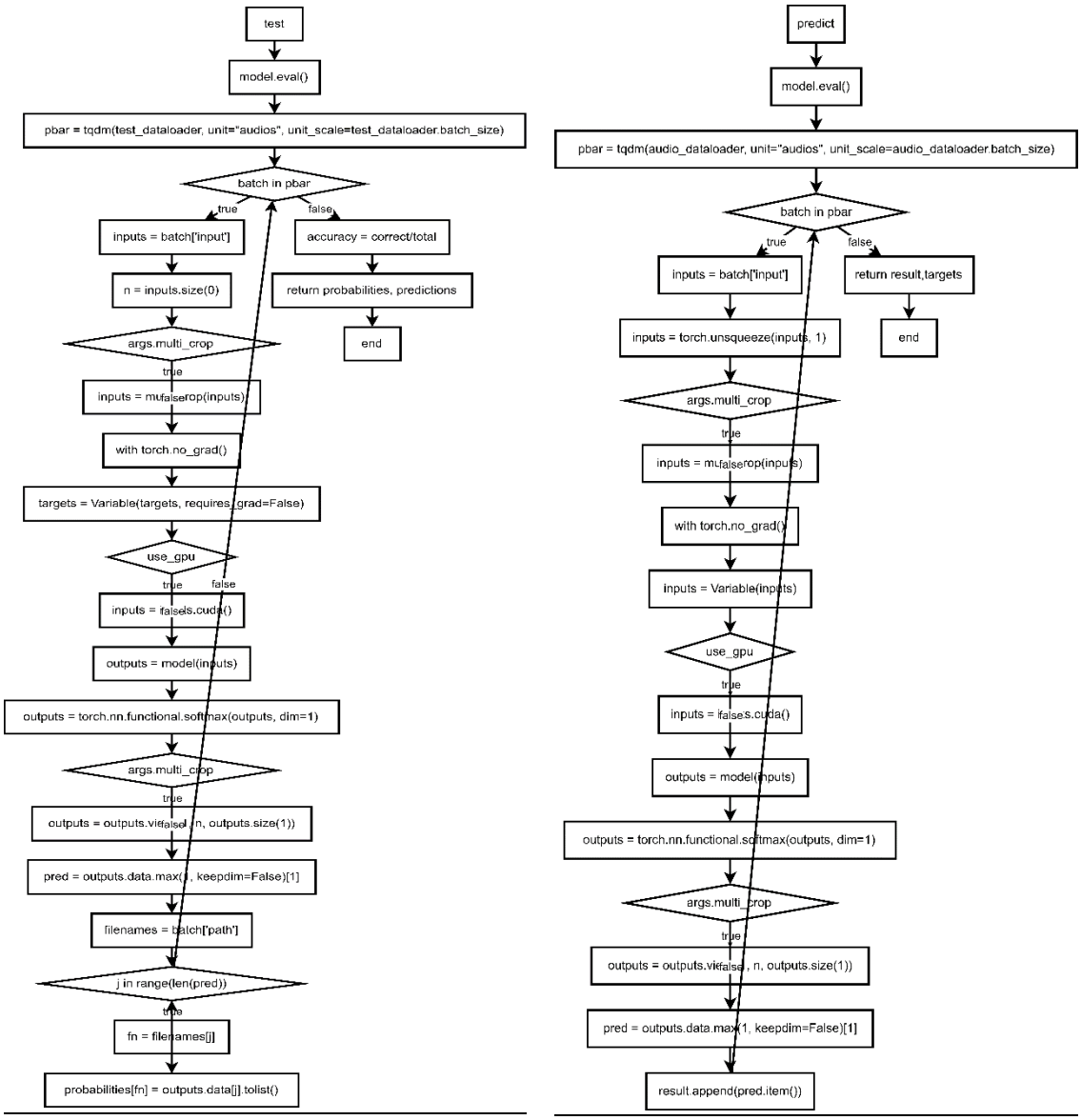
图 3-2 网络训练及验证流程

3.3 网络测试实现

这部分主要用于测试训练好的语音识别模型的性能，其详细步骤如下：同训练网络一样首先进行参数解析。对之前训练得到的模型进行加载，使用 torch.load()加载预训练的模型，并将其转换为浮点数形式。如果有可用的 CUDA 设备，模型将在 GPU 上运行并数据加载和预处理：代码首先加载音频并将其长度修正为固定长度，然后将其转换为梅尔频谱图并再将其转化为 Tensor 数据格式。然后，它创建一个数据生成器来在测试过程中批量加载数据。定义损失函数：定义了交叉熵损失函数，用于在测试阶段计算模型的损失。之后测试，在测试函数中，代码将模型切换到评估模式并初始化了一些统计变量。然后，它遍历了测试数据，对每一批次的数据进行预测，并将预测结果和真实标签进行比较，同时计算混淆矩阵和模型的准

确率。最后使用预测函数：用于对指定的音频文件进行预测，并输出预测结果，该函数在测试过程结束后被调用。

通过评估模型在测试数据集上的表现以及各类别之间混淆情况等，总结了模型的预测性能，为进一步优化模型提供了依据。并且，预测函数则提供了在实际应用中对单个音频样本进行预测的功能。这样可以完美的完成此次课题任务的预测自己录制的语音。下面图 3-3a，图 3-3b 是测试以及预测的流程图。



a. 测试流程　　b. 预测流程

图 3-3 网络测试及预测流程

4 实验结果

此次课题任务本报告使用四个经典的卷积网络对谷歌语音指令训练识别。下面是对比这四个经典网络的对比分析。

4.1 不同网络的精确度

训练测试数据都是基于对网络进行了 70 个 epoch 的 sgd 训练得到的结果。对比结果如下表 4-1 所示。

Model	Speech Commands test set accuracy	Speech Commands test set accuracy with crop	Remarks
VGG19 BN	97.337235%	97.527432%	
ResNet32	96.181419%	96.196050%	
WRN-28-10	97.937089%	97.922458%	
WRN-28-10-dropout	97.702999%	97.717630%	
WRN-52-10	98.039503%	97.980980%	
ResNext29 8x64	97.190929%	97.161668%	最好的网络
DPN92	97.190929%	97.249451%	
DenseNet-BC (L=100, k=12)	97.161668%	97.147037%	
DenseNet-BC (L=190, k=40)	97.117776%	97.147037%	

表 4-1 不同网络的对比结果

由上表我们可以分析得到如下的结果，VGG19 BN：该模型在不使用裁剪时的测试集准确率为 97.337235%，使用裁剪后确有提升，准确率上升到了 97.527432%。这表明裁剪对于这个模型的影响是正面的，有助于提高模型的表现。ResNet32：此模型未裁剪和裁剪后的测试集准确率分别为 96.181419%和 96.196050%。同样，裁剪对于这个模型有所提升，尽管幅度相当微小。WRN-28-10：在未裁剪的情况下，该模型的测试集准确率为 97.937089%，而在裁剪后，准确率降低了一点，下降到了 97.922458%。可以看出，裁剪对于该模型并未带来性能提升。WRN-28-10-dropout：这个模型的未裁剪和裁剪后的测试集准确率分别为 97.702999%和 97.717630%，裁剪对该模型的影响微妙，其实裁剪后的改善非常小。WRN-52-10：在未裁剪的情况下，该模型的测试集准确率高达 98.039503%，为表中所有模型中最高的。然而，裁剪后准确率反而降低了一点，获得了

97.980980%的准确率，说明裁剪对此模型并未提升性能。ResNext29 8x64：这款模型未裁剪时的测试集准确率为 97.190929%，经过裁剪后，准确率反而降低了，为 97.161668%。DPN92：未裁剪时的准确率为 97.190929%，裁剪后的准确率微升为 97.249451%，展示了裁剪有益于此模型的性能。DenseNet-BC (L=100, k=12)：在未裁剪时，模型的测试集准确率为 97.161668%，裁剪后略有下降，为 97.147037%。DenseNet-BC (L=190, k=40)：此模型未进行裁剪时，测试集准确率为 97.117776%。和上一个模型类似，裁剪后的准确率（97.147037%）相比未裁剪有所下降。可以看出，模型的表现并非完全和复杂性或大小成正比。多次裁剪并非对所有模型都有提升作用，具体效果需要在独立的实验中来验证。综合两种情况下的结果来看，ResNext29 8x64 在表现中相较于其他模型表现较好，被认为是"最好的网络"。

4.2 训练及验证可视化

针对 ResNext29 8x64 网络的训练过程，通过可视化可以得到该网络在训练集及测试集的表现。下图 4-1a、图 4-1b、图 4-1c 和图 4-2d 分别表示该网络在训练过程中的 train_accuracy、train_epoch_loss、train_learning_rate 和 train_loss。

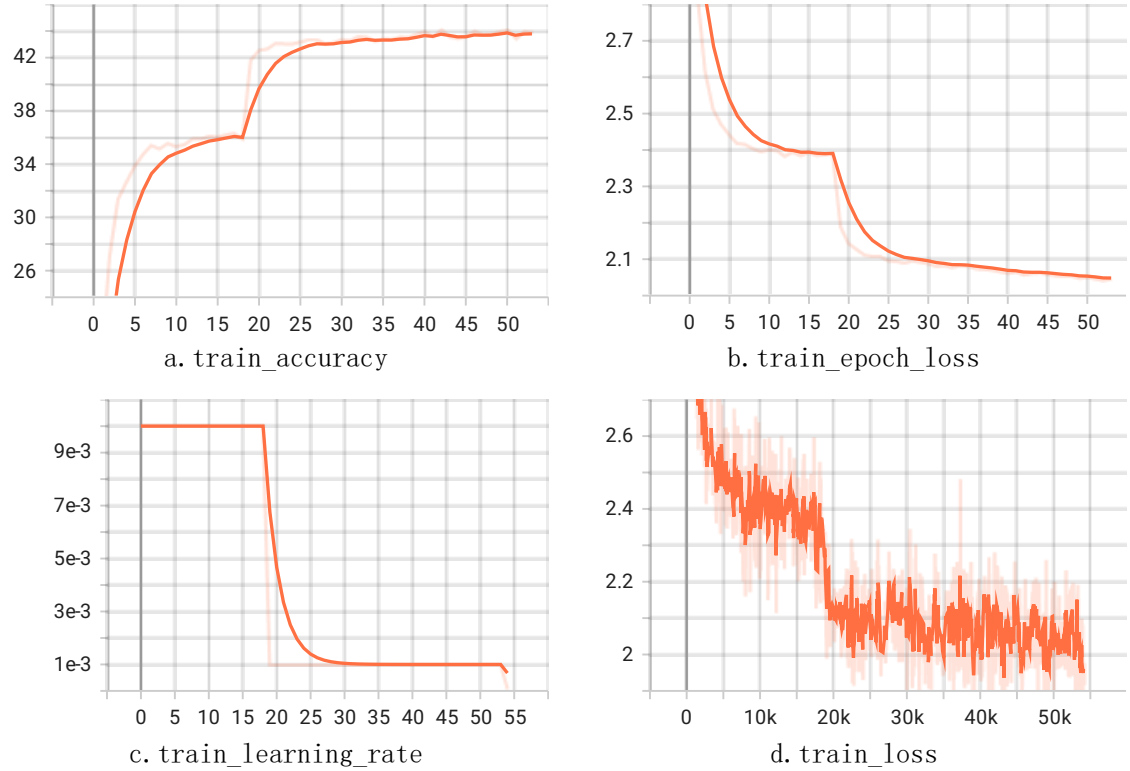
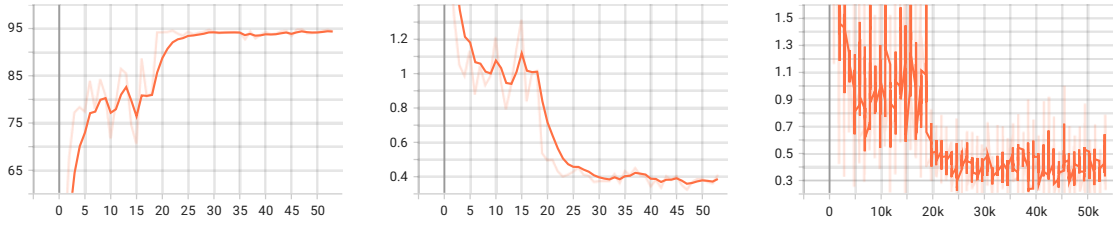


图 4-1 网络训练过程可视化

验证过程可视化结果如下图 4-2a、4-2b 和图 4-2c 所示。



a. valid_accuracy　　b. valid_epoch_loss　　c. valid_loss

图 4-2 网络验证过程可视化

4.3 模型预测

针对训练模型保存的预训练模型，对测试集进行测试，得到如下的混淆矩阵如图 4-3 所示，可以分析得到，在一定限度范围内所选择的网络可以很好的区别不同的语音指令。同时，可以得到 Accuracy 为 94.466156%。

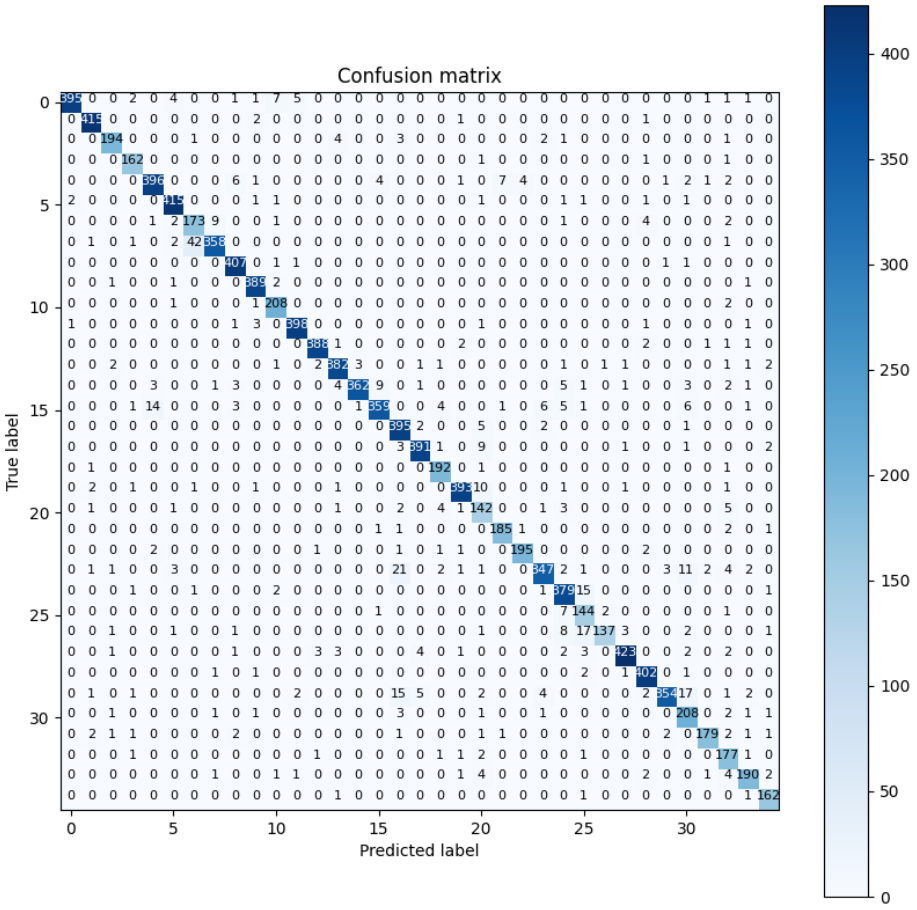


图 4-3 测试混淆矩阵

之后，通过录制 1s 的语音指令，再次对训练得到的模型进行测试，得到如下结果。其中，共有 35 个类别，蓝色框出的为预测正确的。第一行代表正确标签，第二行代表预测标签。如图 4-4 所示。

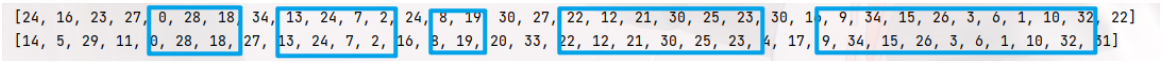


图 4-4 本人语音测试结果

可以看到，能够正确预测得到正确结果。其中，由于本人发言的原因不能很好的与数据集的相匹配，所以正确率达不到测试集测试得到的数据。

5 总结

在此次课题任务中，我们试验了使用多个经典的深度学习模型来识别谷歌语音指令。这些模型包括 VGG19 BN, ResNet32, Wide Residual Network (WRN-28-10), WRN-28-10 with dropout, WRN-52-10，ResNext29 8x64，DPN92，以及 DenseNet-BC with varying layers and growth rates (L=100, k=12), and (L=190, k=40)。其目标是为了提高认识谷歌语音指令的准确性。

这些模型分别在测试集上达到了相当高的准确性，证明了深度学习在语音识别方面的能力。我们发现，最高的测试集精度达到了 98.039503%，由模型 Wide Residual Networks (WRN-52-10) 实现。此外，与未采用 "crop" 方法的模型相比，采用 "crop" 方法对最终的测试集精度影响微乎其微。

尽管所有模型在测试集上的表现都相当出色，但我们也注意到在模型复杂性和测试精度之间并没有直观的正相关性。例如，ResNet32 的精度低于 DenseNet-BC (L=190, k=40)，尽管后者的模型构架更复杂。这表明了选取合适的深度学习模型需要根据具体的任务和数据集来确定，并非总是模型越复杂效果越好。

此外，本项目中展示了深度学习模型处理语音数据的能力。考虑到语音数据的特性，采用适当的数据增强策略，如 "crop"，可以提高模型的泛化能力，但对于精度的提高帮助有限。应注意的是，尽管我们在这个项目中获得了非常高的精度，但还有许多其他因素可以考虑以进一步提升深度学习模型的性能。比如，可以尝试应用不同的预处理方法，调整损失函数，优化学习率策略等等。未来的研究可以探索使用更复杂或者新颖的模型架构，以及定制的训练策略来进一步提升语音识别的准确性。总的来说，这个项目展示了深度学习在语音识别中的强大能力，以及深度神经网络在处理复杂问题时的潜力。

参考文献

[1] Zagoruyko S , Komodakis N .Wide Residual Networks[J]. 2016.DOI:10.5244/C.30.87.

[2] Xie S , Girshick R ,Dollár, Piotr,et al.Aggregated Residual Transformations for Deep Neural Networks[J].IEEE, 2016.DOI:10.1109/CVPR.2017.634.

[3] Simonyan K , Zisserman A .Very Deep Convolutional Networks for Large-Scale Image Recognition[J].Computer Science, 2014.DOI:10.48550/arXiv.1409.1556.

[4] Xie S , Girshick R ,Dollár, Piotr,et al.Aggregated Residual Transformations for Deep Neural Networks[J].IEEE, 2016.DOI:10.1109/CVPR.2017.634.

[5] Zhong Z , Li J , Luo Z ,et al.Spectral-Spatial Residual Network for Hyperspectral Image Classification: A 3-D Deep Learning Framework[J].IEEE Transactions on Geoscience & Remote Sensing, 2017:1-12.DOI:10.1109/TGRS.2017.2755542.

附录

```
# TODO：模型训练，对训练数据加噪，增加鲁棒性以及泛化能力
__author__ = 'chenchuang'

import argparse
import time
from tqdm import *
import torch
from torch.autograd import Variable
from torch.utils.data import DataLoader
from torch.utils.data.sampler import WeightedRandomSampler
from torchvision.transforms import *
from tensorboardX import SummaryWriter
import models
from datasets import *
from transforms import *
from mixup import *

parser = argparse.ArgumentParser(description=__doc__,
formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument("--train-dataset", type=str,
default='datasets/speech_commands/train', help='path of train dataset')
parser.add_argument("--valid-dataset", type=str,
default='datasets/speech_commands/valid', help='path of validation dataset')
parser.add_argument("--background-noise", type=str,
default='datasets/speech_commands/train/_background_noise_', help='path of
background noise')
parser.add_argument("--comment", type=str, default='', help='comment in
tensorboard title')
parser.add_argument("--batch-size", type=int, default=128, help='batch size')
parser.add_argument("--dataload-workers-nums", type=int, default=6, help='number
of workers for dataloader')
parser.add_argument("--weight-decay", type=float, default=1e-2, help='weight
decay')
parser.add_argument("--optim", choices=['sgd', 'adam'], default='sgd',
help='choices of optimization algorithms')
parser.add_argument("--learning-rate", type=float, default=1e-4, help='learning
rate for optimization')
parser.add_argument("--lr-scheduler", choices=['plateau', 'step'],
default='plateau', help='method to adjust learning rate')
parser.add_argument("--lr-scheduler-patience", type=int, default=5, help='lr
scheduler plateau: Number of epochs with no improvement after which
learning rate will be reduced')
```

```
parser.add_argument("--lr-scheduler-step-size", type=int, default=50, help='lr
scheduler step: number of epochs of learning rate decay.')
parser.add_argument("--lr-scheduler-gamma", type=float, default=0.1,
help='learning rate is multiplied by the gamma to decrease it')
parser.add_argument("--max-epochs", type=int, default=70, help='max number of
epochs')
parser.add_argument("--resume", type=str, help='checkpoint file to resume')
parser.add_argument("--model", choices=models.available_models,
default=models.available_models[0], help='model of NN')
parser.add_argument("--input", choices=['mel32'], default='mel32', help='input
of NN')
parser.add_argument('--mixup', action='store_true', help='use mixup')
args = parser.parse_args()

use_gpu = torch.cuda.is_available()
print('use_gpu', use_gpu)
if use_gpu:
    torch.backends.cudnn.benchmark = True
n_mels = 32
if args.input == 'mel40':
    n_mels = 40
# 定义一个数据预处理的变换序列 data_aug_transform
# 依次改变音频的幅度、速度和音调，然后修复音频长度，将其转换为 STFT 表示，然后在 STFT 表示上进行
拉伸和时间位移，最后修复 STFT 的维度。
data_aug_transform = Compose([ChangeAmplitude(), ChangeSpeedAndPitchAudio(),
FixAudioLength(), ToSTFT(), StretchAudioOnSTFT(), TimeshiftAudioOnSTFT(),
FixSTFTDimension()])
bg_dataset = BackgroundNoiseDataset(args.background_noise, data_aug_transform)
# 创建一个新的数据增强变换 add_bg_noise 添加背景噪声。
add_bg_noise = AddBackgroundNoiseOnSTFT(bg_dataset)
# 创建 DataLoader
train_feature_transform = Compose([ToMelSpectrogramFromSTFT(n_mels=n_mels),
DeleteSTFT(), ToTensor('mel_spectrogram', 'input')])
train_dataset = SpeechCommandsDataset(args.train_dataset,
                            Compose([LoadAudio(),
                                    data_aug_transform,
                                    add_bg_noise,
                                    train_feature_transform]))
valid_feature_transform = Compose([ToMelSpectrogram(n_mels=n_mels),
ToTensor('mel_spectrogram', 'input')])
valid_dataset = SpeechCommandsDataset(args.valid_dataset,
                              Compose([LoadAudio(),
                                      FixAudioLength(),
                                      valid_feature_transform]))
```

```
weights = train_dataset.make_weights_for_balanced_classes()
sampler = WeightedRandomSampler(weights, len(weights))
train_dataloader = DataLoader(train_dataset, batch_size=args.batch_size,
sampler=sampler,
                          pin_memory=use_gpu,
num_workers=args.dataload_workers_nums)
valid_dataloader = DataLoader(valid_dataset, batch_size=args.batch_size,
shuffle=False,
                          pin_memory=use_gpu,
num_workers=args.dataload_workers_nums)

# a name used to save checkpoints etc.
full_name = '%s_%s_%s_bs%d_lr%.1e_wd%.1e' % (args.model, args.optim,
args.lr_scheduler, args.batch_size, args.learning_rate, args.weight_decay)
if args.comment:
    full_name = '%s_%s' % (full_name, args.comment)

# 定义模型
model = models.create_model(model_name=args.model, num_classes=len(CLASSES),
in_channels=1)

if use_gpu:
    model = torch.nn.DataParallel(model).cuda()
# 交叉熵损失函数
criterion = torch.nn.CrossEntropyLoss()
# 学习率调度器用于在训练过程中动态调整学习率
if args.optim == 'sgd':
    optimizer = torch.optim.SGD(model.parameters(), lr=args.learning_rate,
momentum=0.9, weight_decay=args.weight_decay)
else:
    optimizer = torch.optim.Adam(model.parameters(), lr=args.learning_rate,
weight_decay=args.weight_decay)

start_timestamp = int(time.time()*1000)
start_epoch = 0
best_accuracy = 0
best_loss = 1e100
global_step = 0
if args.resume:
    print("resuming a checkpoint '%s'" % args.resume)
    checkpoint = torch.load(args.resume)
    model.load_state_dict(checkpoint['state_dict'])
    model.float()
    optimizer.load_state_dict(checkpoint['optimizer'])
```

```
    best_accuracy = checkpoint.get('accuracy', best_accuracy)
    best_loss = checkpoint.get('loss', best_loss)
    start_epoch = checkpoint.get('epoch', start_epoch)
    global_step = checkpoint.get('step', global_step)

    del checkpoint  # reduce memory

if args.lr_scheduler == 'plateau':
    lr_scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,
patience=args.lr_scheduler_patience, factor=args.lr_scheduler_gamma)
else:
    lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
step_size=args.lr_scheduler_step_size, gamma=args.lr_scheduler_gamma,
last_epoch=start_epoch-1)

def get_lr():
    return optimizer.param_groups[0]['lr']

writer = SummaryWriter(comment=('_speech_commands_' + full_name))
def train(epoch):
    # TODO：轮流在训练数据和验证数据上运行模型，然后根据模型的预测结果和真实标签计算损失。每次迭代后，我们都会使用优化器更新模型的参数以最小化损失。
    global global_step

    print("epoch %3d with lr=%.02e" % (epoch, get_lr()))
    phase = 'train'
    writer.add_scalar('%s/learning_rate' % phase,  get_lr(), epoch)

    model.train()  # Set model to training mode

    running_loss = 0.0
    it = 0
    correct = 0
    total = 0
pbar = tqdm(train_dataloader, unit="audios",
unit_scale=train_dataloader.batch_size)
for batch in pbar:
    inputs = batch['input']
    inputs = torch.unsqueeze(inputs, 1)
    targets = batch['target']

if args.mixup:
    inputs, targets = mixup(inputs, targets, num_classes=len(CLASSES))
```

```
    inputs = Variable(inputs, requires_grad=True)
    targets = Variable(targets, requires_grad=False)

    if use_gpu:
      inputs = inputs.cuda()
      targets = targets.cuda(non_blocking=True)

   # forward/backward
    outputs = model(inputs)
    if args.mixup:
        loss = mixup_cross_entropy_loss(outputs, targets)
    else:
       loss = criterion(outputs, targets)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
   # statistics
    it += 1
    global_step += 1
    # running_loss += loss.data[0]
    running_loss += loss.item()
    pred = outputs.data.max(1, keepdim=True)[1]
    if args.mixup:
        targets = batch['target']
        targets = Variable(targets,
requires_grad=False).cuda(non_blocking=True)
correct += pred.eq(targets.data.view_as(pred)).sum()
total += targets.size(0)
    # 所有的训练数据和验证数据都循环一遍后，我们会计算每个阶段的平均损失和精度，并将结果记录在
tensorboard 的 writer 里。
    writer.add_scalar('%s/loss' % phase, loss.item(), global_step)
    # update the progress bar
    pbar.set_postfix({
        'loss': "%.05f" % (running_loss / it),
        'acc': "%.02f%%" % (100*correct/total)
    })

    accuracy = correct/total
    epoch_loss = running_loss / it
    writer.add_scalar('%s/accuracy' % phase, 100*accuracy, epoch)
    writer.add_scalar('%s/epoch_loss' % phase, epoch_loss, epoch)
def valid(epoch):
    # TODO: 模型验证和训练一样的步骤
    global best_accuracy, best_loss, global_step
```

```
  phase = 'valid'
  model.eval()  # Set model to evaluate mode

  running_loss = 0.0
  it = 0
  correct = 0
  total = 0

  pbar = tqdm(valid_dataloader, unit="audios",
  unit_scale=valid_dataloader.batch_size)
  for batch in pbar:
      inputs = batch['input']
      inputs = torch.unsqueeze(inputs, 1)
      targets = batch['target']
      with torch.no_grad():
          inputs = Variable(inputs)
      targets = Variable(targets, requires_grad=False)

      if use_gpu:
          inputs = inputs.cuda()
          targets = targets.cuda(non_blocking=True)

      # forward
      outputs = model(inputs)
      loss = criterion(outputs, targets)
    # statistics
    it += 1
    global_step += 1
    running_loss += loss.item()
    pred = outputs.data.max(1, keepdim=True)[1]
    correct += pred.eq(targets.data.view_as(pred)).sum()
    total += targets.size(0)

    writer.add_scalar('%s/loss' % phase, loss.item(), global_step)

    # update the progress bar
    pbar.set_postfix({
        'loss': "%.05f" % (running_loss / it),
        'acc': "%.02f%%" % (100*correct/total)
    })

accuracy = correct/total
epoch_loss = running_loss / it
```

```
    writer.add_scalar('%s/accuracy' % phase, 100*accuracy, epoch)
    writer.add_scalar('%s/epoch_loss' % phase, epoch_loss, epoch)

    checkpoint = {
        'epoch': epoch,
        'step': global_step,
        'state_dict': model.state_dict(),
        'loss': epoch_loss,
        'accuracy': accuracy,
        'optimizer' : optimizer.state_dict(),
    }

    if accuracy > best_accuracy:
        best_accuracy = accuracy
        torch.save(checkpoint, 'checkpoints/best-loss-speech-commands-
checkpoint-%s.pth' % full_name)
        torch.save(model, '%d-%s-best-loss.pth' % (start_timestamp, full_name))
    if epoch_loss < best_loss:
        best_loss = epoch_loss
        torch.save(checkpoint, 'checkpoints/best-acc-speech-commands-
checkpoint-%s.pth' % full_name)
        torch.save(model, '%d-%s-best-acc.pth' % (start_timestamp, full_name))
    torch.save(checkpoint, 'checkpoints/last-speech-commands-checkpoint.pth')
    del checkpoint  # reduce memory
    return epoch_loss
print("training %s for Google speech commands..." % args.model)
since = time.time()
for epoch in range(start_epoch, args.max_epochs):
    if args.lr_scheduler == 'step':
        lr_scheduler.step()
    train(epoch)
    epoch_loss = valid(epoch)
    if args.lr_scheduler == 'plateau':
        lr_scheduler.step(metrics=epoch_loss)
    time_elapsed = time.time() - since
    time_str = 'total time elapsed: {:.0f}h {:.0f}m {:.0f}s
'.format(time_elapsed // 3600, time_elapsed % 3600 // 60, time_elapsed % 60)
    print("%s, best accuracy: %.02f%%, best loss %f" % (time_str,
100*best_accuracy, best_loss))
print("finished")
```