第 1 章

语音识别之路

随着信息科技的不断发展，语音转换文字技术逐渐成熟，并被广泛应用于社会生产中各个领域。特别是在移动应用领域，越来越多的语音转换文字 App 进入市场，并且得到了广大用户的青睐。

基于多模态的语音识别，语音文本转换则是深度学习语音技术研究的重点，也是最有前途的方向之一。掌握多模态语音转换的方法可以使深度学习从业者加深对模型和项目的理解，从而超越一般技术人员，获得极大的职业竞争优势。

本书将以此为目标，手把手地教读者从零开始，学习和掌握基于深度学习语音识别的基础内容，最终完成多模态语音文本转换的实际项目。

1.1 何谓语音识别

语音识别技术是将声音转换成文字的一种技术，类似于人类的耳朵，拥有听懂他人的说话内容并将其转换成可以辨识内容的能力。

不妨设想如下场景：

当你加完班回到家中，疲惫地躺在沙发上，随口说一句“打开电视”，沙发前的电视按语音命令开启，然后一个温柔的声音问候你，“今天想看什么类型的电影？”，或者主动向你推荐目前流行的一些影片。

这个例子是音频识别能够处理的场景，虽然看似科幻，但是实际上这些场景已经不再是以往人们的设想，目前已经正在悄悄地走进你我的生活。

2018 年，谷歌在开发者大会上演示了一个预约理发店的聊天机器人，语气惟妙惟肖，表现相当令人惊艳。另外，相信很多读者都接到过人工智能的推销电话，不去仔细分辨的话，根本不知道电话那头只是一个能够做出音频处理的聊天机器人程序。

“语音转换”“人机对话”“机器人客服”是语音识别应用最广泛的三个部分，也是商业价值较高的一些方向。此外，还有“看图说话”等一些带有娱乐性质的应用。这些都是语音识别技术的常见应用。

语音识别通常称为自动语音识别（Automatic Speech Recognition，ASR），主要是将人类语音中的词汇内容转换为计算机可读的输入，一般都是可以理解的文本内容，也有可能是二进制编码或者

字符序列。语音识别是一项融合多学科知识的前沿技术，覆盖了数学与统计学、声学与语言学、计算机与人工智能等基础学科和前沿学科，是人机自然交互技术中的关键环节。但是，语音识别自诞生以来的半个多世纪，一直没有在实际应用中得到普遍认可。一方面，语音识别技术存在缺陷，其识别精度和速度都达不到实际应用的要求；另一方面，业界对语音识别的期望过高，实际上语音识别与键盘、鼠标或触摸屏等应该是融合关系，而非替代关系。

深度学习技术自 2015 年兴起之后，已经取得了长足进步。语音识别的精度和速度取决于实际应用环境，但在安静环境、标准口音、常见词汇场景下的语音识别率已经超过 95%，意味着具备与人类相仿的语言识别能力，而这也是语音识别技术当前发展比较火热的原因。

随着技术的发展，现在口音、方言、噪声等场景下的语音识别也达到了可用状态，特别是远场语音识别已经随着智能音箱的兴起，成为全球消费电子领域应用最成功的技术之一。由于语音交互提供了更自然、更便利、更高效的沟通形式，因此语音必定成为未来主要的人机互动接口之一。

当然，当前技术还存在很多不足，如对于强噪声、超远场、强干扰、多语种、大词汇等场景下的语音识别，还需要很大的提升；另外，多人语音识别和离线语音识别也是当前需要重点解决的问题。虽然语音识别还无法做到无限制领域、无限制人群地应用，但是至少从应用实践中我们看到了一些希望。当然，实际上自然语言处理并不限于前文所讲的这些，随着人们对深度学习的了解，更多应用正在不停地开发出来，相信读者会亲眼见证这一切的发生。

1.2　语音识别为什么那么难

语音识别在生活中的应用范围越来越广，可以很明显地看到或者感觉到，世界顶尖科技公司都在语音识别方面做了很多投入。目前，亚马逊 Alexa、Google 以及国内大型厂商的语音助手和设备越来越受欢迎，它们正在改变我们的购物方式、搜索方式、与设备的互动方式以及彼此之间的互动方式。

然而相较于图像识别以及自然语言处理领域，语音识别的发展并不如其他领域发展得迅捷。原因可以说是多种多样的，但是最基本的还是对于不同的语言生成者，其在产生语音的条件和形式上大有不同。甚至于最简单的对于语速的控制，每个人就有着不同的特征。

- 有声读物的推荐速度约为150~160 wpm（word per minute，每分钟的字词数）。
- 幻灯片演示建议接近100~125 wpm。
- 拍卖师的语速可以达到每秒约250 wpm。
- 小约翰·莫斯基塔(John Moschitta，Jr)曾保持吉尼斯世界纪录，作为世界上最快的讲话者，每分钟能说586个单词。他的记录在1990年被史蒂夫·伍德莫尔打破了，他每分钟讲637个单词，然后是肖恩香农，他在1995年8月30日每分钟讲655个单词。肖恩香农背诵哈姆雷特的独白“成为或不成为”（260字）在23.8秒。

听力比我们想象得更难、更复杂，而如果将这一过程传送给机器，需要经历以下过程：

（1）声波接收：声波是人类交流的基础，我们通过耳朵感知它们。对于机器而言，这些声波首先以模拟信号的形式存在，必须被转换为数字形式后，才能够进行后续的处理和分析。

（2）噪声分离：在嘈杂的环境中，如餐馆，我们的听觉系统能够出色地从各种背景噪声中筛选出话语信号。这些噪声可能来源于电话铃声、房间声学效应、他人谈话声、交通噪声等。机器也需要具备相似的噪声分离能力，以确保语音的清晰度。

（3）断点处理：人类的话语充满了变数，包括讲话速度、句子结构和单词之间的界限。有时，快速讲述或句子间缺乏停顿，可能使话语听起来像是一连串的单词，难以区分单词的起止。机器需要适应这种不规律性，并准确地处理这些断点。

（4）个性尊重：每个人的声音都是独特的，受其年龄、性别、口音、风格、个性、背景和意图等多种因素的影响。这种独特性甚至在同一个人的多次讲述中也会有所变化。因此，机器在处理语音时，必须充分考虑这些因素，以更准确地识别和理解语音。

（5）口音理解：面对不同地区的方言和口音，语音处理面临巨大的挑战。人们的语言可能因为其独特的发音和用语习惯而难以被理解。机器需要具有足够的适应性，以准确解析并理解各种口音和方言。

（6）同义翻译：语言中的同音异义词为语音识别带来了额外的难度。这些词汇虽然听起来相似或相同，但其意义可能截然不同。为了准确捕捉说话者的意图，机器需要细微地区分这些词汇的差异。

（7）助词过滤：在日常交流中，人们常常使用“嗯”“呃”等无意义的填充词。尽管这些词汇不影响人类之间的交流，但机器可能会因此受到干扰。为此，机器需要学习如何过滤这些词汇，以确保更准确地理解说话者的意图。

因此，通过这个过程的分析可以得知，即使是两个人在餐厅中进行简单的闲聊，对于语音的分析也是困难重重的，更不必说多人在闲聊时需要分辨出发声者是谁。

1.3　语音识别之路——语音识别的发展历程

现代语音识别可以追溯到 1952 年，Davis 等研制了世界上第一个能识别 10 个英文数字发音的实验系统，从此正式开启了语音识别的技术发展进程。语音识别发展到今天已经有 70 多年，它从技术方向上大体可以分为三个阶段。

如图 1.1 所示是 1993—2017 年在 Switchboard 上语音识别率的进展情况。从图 1-1 中可以看出，1993—2009 年，语音识别一直处于高斯混合-隐马尔科夫（GMM-HMM）时代，语音识别率提升缓慢，尤其是 2000—2009 年，语音识别率基本处于停滞状态；2009 年，随着深度学习技术，特别是循环神经网络（Decurrent Neural Network，DNN）的兴起，语音识别框架变为循环神经网络-隐马尔科夫（DNN-HMM），并且使得语音识别进入了神经网络深度学习时代，语音识别的精准率得到了显著提升；2015 年以后，由于“端到端”技术兴起，语音识别进入了百花齐放的时代，语音界都在训练更深、更复杂的网络，同时利用端到端技术进一步大幅提升了语音识别的性能，直到 2017 年，微软在 Switchboard 上达到词错误率 5.1%，从而让语音识别的准确性首次超越了人类，当然这是在一定条件下的实验结果，还不具有普遍代表性。

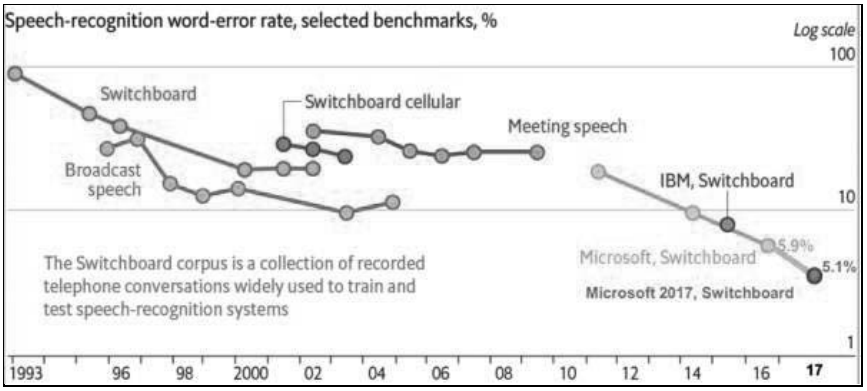

图 1-1　不同时代的语音识别

1.3.1　高斯混合-隐马尔科夫时代

70 年代，语音识别主要集中在小词汇量、孤立词识别方面，使用的方法主要是简单的模板匹配方法，即首先提取语音信号的特征构建参数模板，然后将测试语音与参考模板参数一一进行比较和匹配，取距离最近的样本所对应的词标注为该语音信号的发音。该方法对解决孤立词识别是有效的，但对于大词汇量、非特定人的连续语音识别就无能为力了。因此，进入 80 年代后，研究思路发生了重大变化，传统的基于模板匹配的技术思路开始转向基于隐马尔科夫模型（Hidden Markov Model，HMM）的技术思路。

HMM 的理论基础在 1970 年前后就已经由 Baum 等建立起来，随后由 CMU 的 Baker 和 IBM 的 Jelinek 等将其应用到语音识别中。HMM 模型假定一个音素含有 3~5 个状态，同一状态的发音相对稳定，不同状态间可以按照一定概率进行跳转，某一状态的特征分布可以用概率模型来描述，使用最广泛的模型是 GMM。因此，在 GMM-HMM 框架中，HMM 描述的是语音的短时平稳的动态性，GMM 用来描述 HMM 每一状态内部的发音特征，如图 1-2 所示。

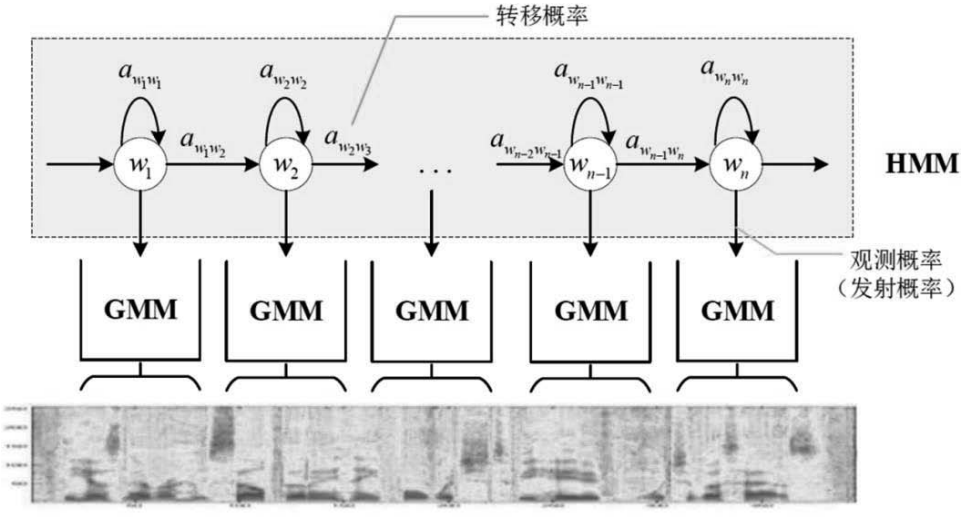

图 1-2　GMM-HMM 语音识别模型

基于 GMM-HMM 框架，研究者提出了各种改进方法，如结合上下文信息的动态贝叶斯方法、区分性训练方法、自适应训练方法、HMM/NN 混合模型方法等。这些方法都对语音识别研究产生了深远影响，并为下一代语音识别技术的产生做好了准备。自 20 世纪 90 年代语音识别声学模型的区分性训练准则和模型自适应方法被提出以后，在很长一段时间内语音识别的发展都比较缓慢，语音识别的错误率一直没有明显的下降。

1.3.2　深度神经网络-隐马尔科夫时代

2006 年，Hinton 提出了深度置信网络（Deep Belief Network，DBN），直接促进深度神经网络（Deep Neural Network，DNN）研究的复苏。2009 年，Hinton 将 DNN 应用于语音的声学建模，在 TIMIT 上获得了当时最好的结果。2011 年年底，微软研究院的俞栋、邓力又把 DNN 技术应用在大词汇量连续语音识别的任务上，大大降低了语音识别的错误率。从此，语音识别进入 DNN-HMM 时代。

DNN-HMM 主要是用 DNN 模型代替原来的 GMM 模型，对每个状态进行建模，如图 1-3 所示。DNN 带来的好处是不再需要对语音数据分布进行假设，将相邻的语音帧拼接，又包含语音的时序结构信息，使得对于状态的分类概率有了明显提升，同时 DNN 还具有强大的环境学习能力，可以提升对噪声和口音的鲁棒性。

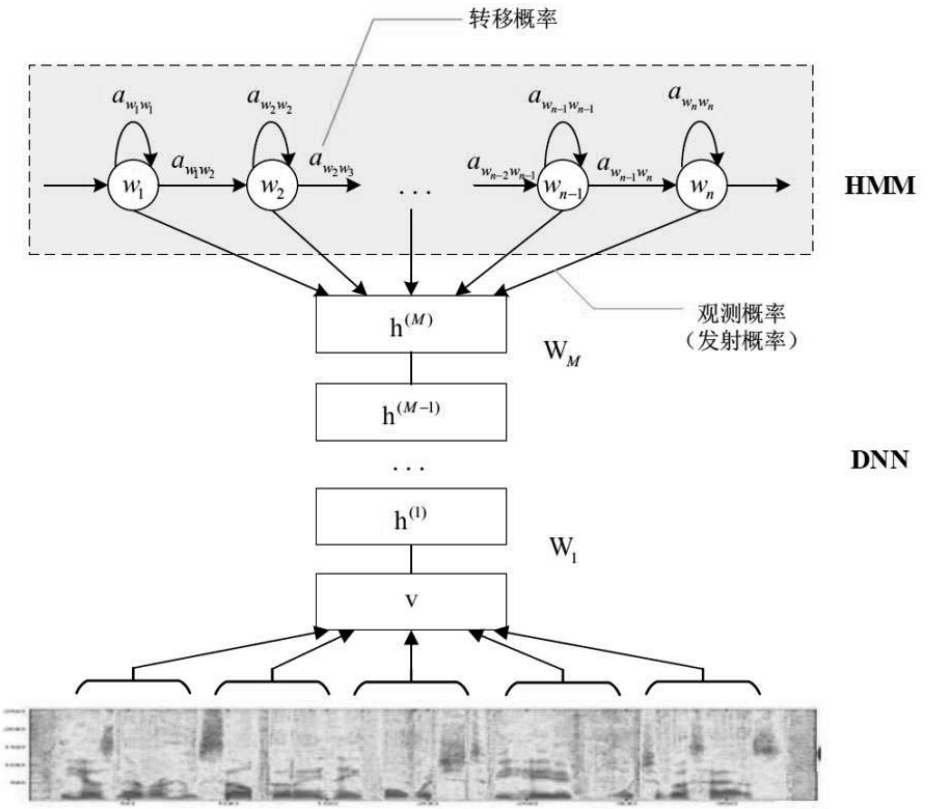

图 1-3　DNN-HMM 模型

简单来说，DNN 就是给出输入的一串特征所对应的状态概率。由于语音信号是连续的，不仅各个音素、音节以及词之间没有明显的边界，各个发音单位还会受到上下文的影响。虽然拼帧可以增加上下文信息，但对于语音来说还是不够。而递归神经网络（Recurrent Neural Network，RNN）的出现可以记住更多历史信息，更有利于对语音信号的上下文信息进行建模。

1.3.3　基于深度学习的端到端语音识别时代

随着深度学习的发展，语音识别由 DNN-HMM 时代发展到基于深度学习的“端到端”时代，这个时代的主要特征是代价函数发生了变化，但基本的模型结构并没有太大变化。总体来说，端到端技术解决了输入序列长度远大于输出序列长度的问题。

采用 CTC 作为损失函数的声学模型序列不需要预先将数据对齐，只需要一个输入序列和一个输出序列就可以进行训练。CTC 关心的是预测输出的序列是否和真实的序列相近，而不关心预测输出的序列中每个结果在时间点上是否和输入的序列正好对齐。CTC 建模单元是音素或者字，因此它引入了 Blank。对于一段语音，CTC 最后输出的是尖峰的序列，尖峰的位置对应建模单元的 Label，其他位置都是 Blank。

Sequence-to-Sequence 方法原来主要应用于机器翻译领域。2017 年，Google 将其应用于语音识别领域，取得了非常好的效果，将词错误率降低至 5.6%。如图 1-4 所示，Google 提出的新系统框架由三部分组成：Encoder 编码器组件，它和标准的声学模型相似，输入的是语音信号的时频特征；经过一系列神经网络，映射成高级特征 henc，然后传递给 Attention 组件，其使用 henc 特征学习输入 x 和预测子单元之间的对齐方式，子单元可以是一个音素或一个字；最后，Attention 模块的输出传递给 Decoder，生成一系列假设词的概率分布，类似于传统的语言模型。

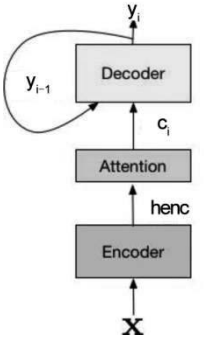

图 1-4　Sequence-to-Sequence 方法

而随着 Whisper 语音转换模型的推出开启了可以用于实际任务的端到端（Task End-to-End）的时代。Whisper 是一种自动语音识别（Automatic Speech Recognition，ASR）系统，旨在将语音转换为文本。作为一款多任务模型，它不仅可以执行多语言语音识别，还可以执行语音翻译和语言识别等任务。Whisper 采用了 Transformer 架构的编码器-解码器模型，使其在各种语音处理任务中表现出色。Whisper 模型架构如图 1-5 所示。

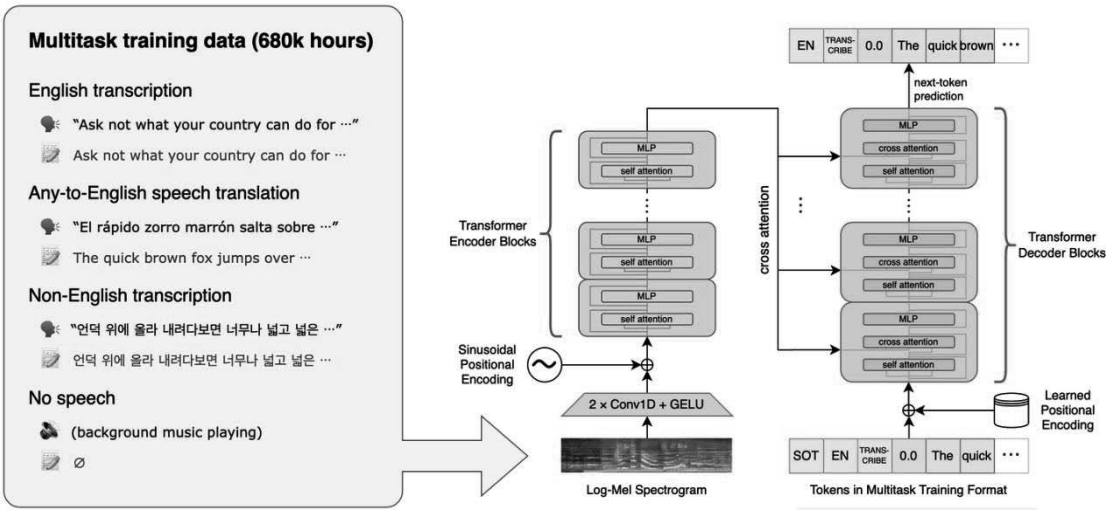

图 1-5　Whisper 模型架构

Whisper 的核心技术在于其端到端的架构。输入的语音首先被分成 30 秒的模块，然后转换为 log-Mel 频谱图，再通过编码器计算注意力，最后将数据传递给解码器。解码器被训练用来预测相应的文本，并添加特殊标记，用于执行诸如语言识别、多语言语音转录和英语语音翻译等任务。Whisper 还在 Transformer 模型中使用了多任务训练格式，利用一组特殊的令牌作为任务说明符或分类目标。Whisper 的优点在于其强大的语音识别能力，能够处理各种口音、背景噪声和技术语言。

随着端到端技术的突破，深度学习模型不再需要对音素内部状态的变化进行描述，而是将语音识别的所有模块统一成神经网络模型，使语音识别朝着更简单、更高效、更准确的方向发展。

1.3.4　多模态架构的语音识别与转换

近年来，随着人们对人工智能领域和深度学习技术的认识不断加强，多模态模型在语音和文本转换之间发挥了至关重要的作用，架起了两者之间的桥梁。这种模型的优势在于，它可以从多个模态中感知和理解事物，处理来自不同类型的数据信息，例如语音和上下文内容等。

多模态模型不仅可以应用于语音转换领域，还可以在其他领域中发挥重要作用。例如，在自然语言处理领域中，多模态模型可以结合语言、图像和语音等多种信息，实现更为准确的理解和生成任务。在语音转换领域，多模态模型可以结合多种临场情况和非目标语音特征提高转换和预测的准确性。

多模态模型的核心在于特征的融合。模型可以从每个输入数据中提取出特征向量，然后将这些特征向量融合成一个整体的特征输入。对于不同的数据类型，可以使用不同的模型进行特征提取。例如，对于图像数据，可以使用卷积神经网络提取特征；对于语音数据，可以使用注意力模型提取特征。

多模态语音转换模型是深度学习语音转换领域中的一个重要研究方向。这种模型相对于以往的人工智能系统更具创造性和协作性，拥有更为准确的语音辨识度和文本生成能力。通过结合多种模态的数据信息，多模态语音转换模型可以应付更加复杂的场景和语境，为未来的语音转换研究提供更为广阔的发展空间。

1.4　基于深度学习的语音识别的未来

基于深度学习的语音识别是当前人工智能领域的研究热点之一。随着语音技术的不断发展，语音识别技术将在未来扮演更加重要的角色。

语音识别技术的发展已经有几十年的历史，但是基于深度学习的语音识别技术在近年来才取得了突破性的进展。深度学习技术可以通过学习大量的语音数据自动提取语音特征，从而提高语音识别的准确率和鲁棒性。

基于深度学习的语音识别技术的基本原理是，通过训练大量的语音数据让深度学习模型自动提取语音特征，并利用这些特征对语音进行分类。其中，最关键的步骤是训练数据的选择和预处理、模型结构的确定以及模型的训练和优化。

基于深度学习的语音识别技术的发展历程可以分为三个阶段：第一个阶段是模型的初步探索和验证阶段；第二个阶段是模型的优化和完善阶段；第三个阶段是模型的应用和推广阶段。目前，基于深度学习的语音识别技术已经广泛应用于语音助手、智能客服、智能家居、汽车电子等领域，未来还将继续拓展应用领域。

基于深度学习的语音识别技术的优点在于，它可以自动提取语音特征，提高语音识别的准确率和鲁棒性。同时，深度学习技术还可以通过对语音数据的分析和挖掘发现更多的语音信息，为语音识别提供更多的可能性。但是，该技术也存在一些缺点，例如对语音数据的依赖性较高、模型的可解释性较差等。

随着人工智能技术的不断发展，基于深度学习的语音识别技术也将继续发展。未来，基于深度学习的语音识别技术将更加注重情感识别、语义识别等高级应用的研究。同时，随着自然语言处理技术的不断发展，基于深度学习的语音识别技术将更加注重与自然语言处理的结合，实现更加智能的语音交互。此外，基于深度学习的语音识别技术还将促进多模态信息融合技术的发展，将语音识别与其他信息来源进行结合，提高语音识别的准确率和鲁棒性。

基于深度学习的语音识别技术是当前人工智能领域的研究热点之一，其未来的发展前景广阔。同时，随着自然语言处理技术和多模态信息融合技术的发展，基于深度学习的语音识别技术还将实现更加智能的语音交互，为人们的生活和工作带来更多的便利和价值。

1.5　本章小结

本章介绍了语音识别的发展历程，以及在各个不同时期语音识别与语音转换的技术方案和解决实际问题的方法。可以看到，随着科技的进步，研究者和从业者会使用越来越先进的技术和方法来解决实践中遇到的问题和困难。

从下一章开始将引领读者完成基于深度学习框架 PyTorch 2.0 的多模态语音识别之路。在这个学习过程中，读者将了解深度学习的基本原理以及 PyTorch 2.0 框架的使用方法，使读者从零开始逐渐掌握目前前沿的多模态语音文本转换方法。

第 2 章

PyTorch 2.0 深度学习环境搭建

工欲善其事，必先利其器。第 1 章介绍了 PyTorch 与深度学习神经网络之间的关系，本章开始正式进入 PyTorch 2.0 的学习。

首先我们需要知道，无论是构建深度学习应用程序，还是应用已完成训练的项目到某个具体的项目中，都需要使用编程语言完成设计者的目的，在本书中使用 Python 作为开发的基本语言。

Python 在深度学习领域中被广泛采用，这得益于许多第三方提供的集成了大量科学计算库的 Python 安装工具，其中最常用的是 Miniconda。Python 是一种脚本语言，如果不使用 Miniconda，那么第三方库的安装可能会变得相当复杂，同时各个库之间的依赖性也很难得到妥善的处理。因此，为了简化安装过程并确保库之间的良好配合，推荐安装 Miniconda 来替代原生的 Python。

本章首先介绍 Miniconda 和 PyCharm 的安装，之后将完成一个基于特征词的语音唤醒项目，帮助读者了解完整的 PyTorch 项目实现过程。

2.1 环境搭建 1：安装 Python

2.1.1 Miniconda 的下载与安装

1. 下载和安装

（1）通过百度访问 Miniconda 官方网站，如图 2-1 所示。按页面左侧菜单提示进入下载页面。

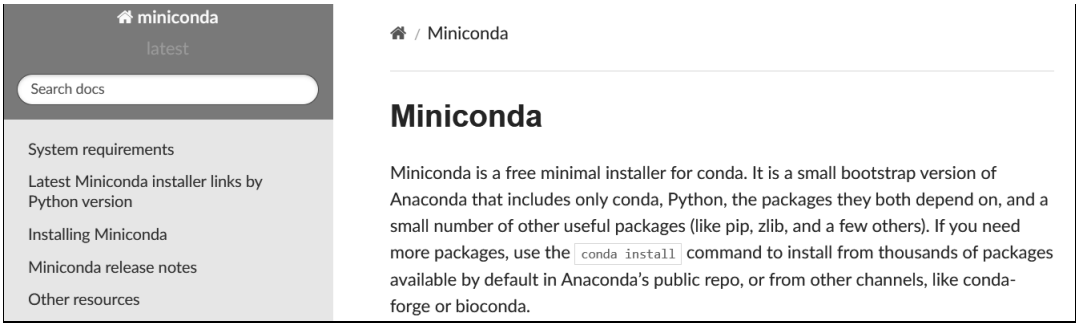

图 2-1 Miniconda 下载页面

（2）下载页面如图 2-2 所示，可以看到官方网站支持不同 Python 版本的 Miniconda。我们根据

自己的操作系统选择相应的 Miniconda 下载即可。这里推荐使用 Windows Python 3.9 版本，相对于更高版本，3.9 版本经过一段时间的使用具有一定的稳定性。当然，读者也可根据自己的喜好选择。

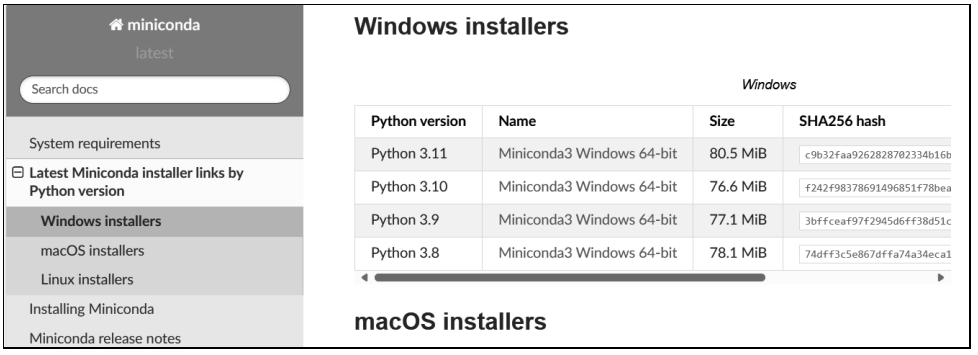

图 2-2　Miniconda 在官方网站提供的下载

注意：建议读者选择 Python 3.9 的版本，方便后面与 PyTorch 2.0.1 GPU、CUDA 11.8 版本配合起来使用。如果想使用其他更高版本的配合方式，请参考 PyTorch 官方文档确认。

（3）下载完成后，得到的文件是 EXE 版本，直接运行即可进入安装过程，安装目录选择默认的目录即可。安装完成后，出现如图 2-3 所示的目录结构，说明安装正确。

图 2-3　Miniconda 安装目录

2. 打开控制台

之后依次单击“开始”→“所有程序”→Miniconda3→Miniconda Prompt，打开 Miniconda Prompt 窗口，它与 CMD 控制台类似，输入命令就可以控制和配置 Python。在 Miniconda 中最常用的是 conda 命令，该命令可以执行一些基本操作，读者可以自行测试一下它的用法。

3. 验证Python

接下来验证一下是否安装好了 Python。在控制台中输入 python，如安装正确，会打印出版本号以及控制符号。在控制符号下输入代码：

```
print("hello Python")
```

输出结果如图 2-4 所示。

```
(base) C:\Users\xiaohua>python
Python 3.9.10 | packaged by conda-forge | (main, Feb  1 2022, 21:22:07) [MSC v.1929 64 bit
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello")
hello
>>>
```

图 2-4　验证 Miniconda Python 安装成功

4. 使用pip命令

使用 Miniconda 工具包的好处在于，它能帮助我们安装和使用大量的第三方类库并能维护相互建的依赖关系。查看已安装的第三方类库的代码如下：

```
pip list
```

注意：如果此时命令行还在>>>状态，可以输入 exit()退出。

在 Miniconda Prompt 控制台输入 pip list，结果如图 2-5 所示。

```
PS C:\Users\xiayu> pip list
Package                   Version
------------------------- ---------------
absl-py                   1.4.0
aiohttp                   3.7.4.post0
aiosignal                 1.3.1
ansicon                   1.89.0
antlr4-python3-runtime    4.9.3
anyio                     3.7.0
appdirs                   1.4.4
arrow                     1.2.3
asttokens                 2.2.1
astunparse                1.6.3
async-timeout             3.0.1
attention                 5.0.0
```

图 2-5　列出已安装的第三方类库

Miniconda 中使用 pip 进行操作的方法还有很多，其中最重要的是安装第三方类库，命令如下：

```
pip install name
```

这里的 name 是需要安装的第三方类库名，假设需要安装 NumPy 包（这个包已经安装过），那么输入的命令就是：

```
pip install numpy
```

结果如图 2-6 所示。

```
PS C:\Users\xiayu> pip install numpy
Collecting numpy
  Obtaining dependency information for numpy from https://files.pythonhoste
d.org/packages/df/18/181fb40f03090c6fbd061bb8b1f4c32453f7c602b0dc7c08b307ba
ca7cd7/numpy-1.25.2-cp39-cp39-win_amd64.whl.metadata
  Using cached numpy-1.25.2-cp39-cp39-win_amd64.whl.metadata (5.7 kB)
Using cached numpy-1.25.2-cp39-cp39-win_amd64.whl (15.6 MB)
Installing collected packages: numpy
Successfully installed numpy-1.25.2
```

图 2-6　列出已安装的第三方类库

使用 Miniconda 工具包的一个好处是默认安装了大部分学习所需的第三方类库，这样可以避免我们在安装和使用某个特定的类库时，出现依赖类库缺失的情况。

2.1.2　PyCharm 的下载与安装

和其他编程语言类似，Python 程序的编写可以使用 Windows 自带的控制台。但是这种方式对于较为复杂的程序工程来说，容易混淆相互之间的层级和交互文件，因此在编写程序工程时，建议使用专用的 Python 编译器 PyCharm。

1. PyCharm的下载和安装

（1）进入 PyCharm 官网主页，单击 Download 图标，进入下载页面，页面上可以选择下载收费的专业版或者免费的社区版。这里我们选择免费的社区版，如图 2-7 所示。

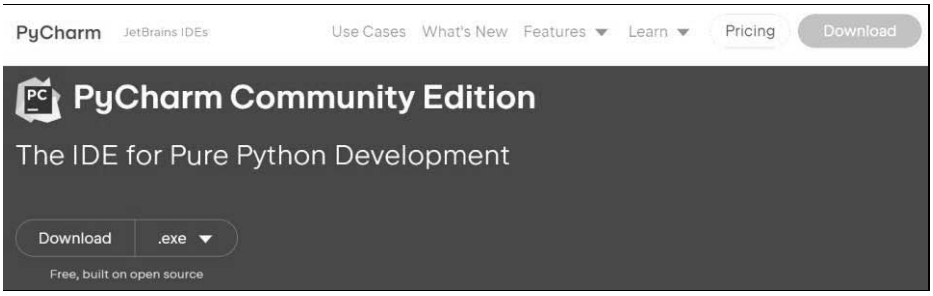

图 2-7　选择 PyCharm 的免费版

（2）下载下来的安装文件名为 pycharm-community-2023.3.exe。双击运行后进入安装界面，如图 2-8 所示。直接单击 Next 按钮，进入下一个安装界面。

（3）如图 2-9 所示，在安装 PyCharm 的过程中需要确定相关的配置选项，这里建议读者把窗口上的检查框都选中，再单击 Next 按钮，进入下一步安装。

图 2-8　安装界面

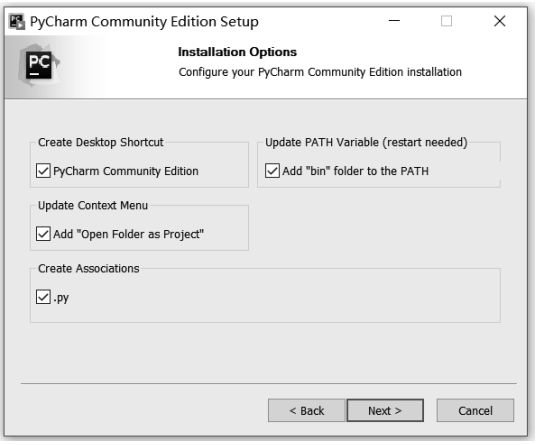

图 2-9　选中所有的检查框

（4）中间安装过程比较简单，按提示单击 Next 按钮安装即可。安装完成后，单击 Finish 按钮退出安装向导，如图 2-10 所示。

图 2-10　安装完成

2. 使用PyCharm创建程序

（1）单击桌面上新生成的图标进入 PyCharm 程序界面，首先是第一次启动的定位，如图 2-11 所示。这里是对程序存储的定位，一般建议选择第 2 个 Do not import settings。

（2）单击 OK 按钮后进入 PyCharm 配置窗口，如图 2-12 所示。

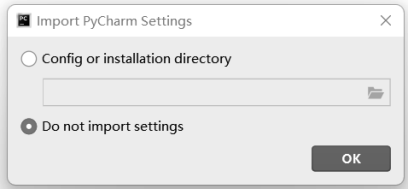

图 2-11　由 PyCharm 自动指定

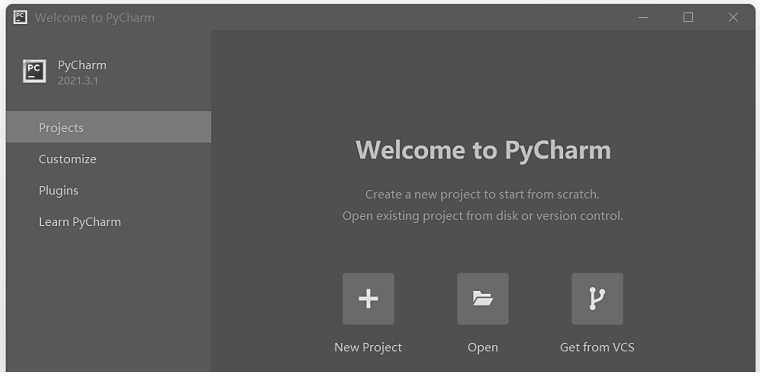

图 2-12　界面配置

（3）在配置窗口可以对 PyCharm 的界面进行配置，选择自己喜欢的使用风格。如果对其不熟悉，直接使用默认配置即可，如图 2-13 所示。

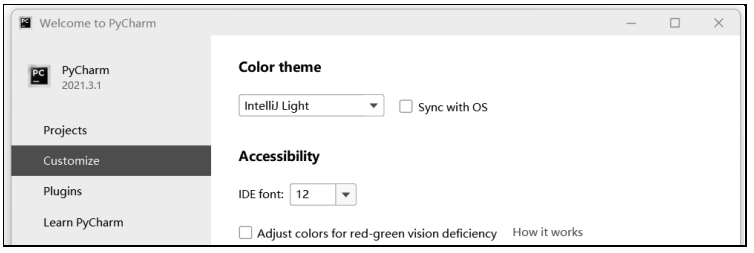

图 2-13　对 PyCharm 的界面进行配置

（4）在如图 2-12 所示的界面上，单击 New Project 创建一个新项目 jupyter_book，如图 2-14 所示。

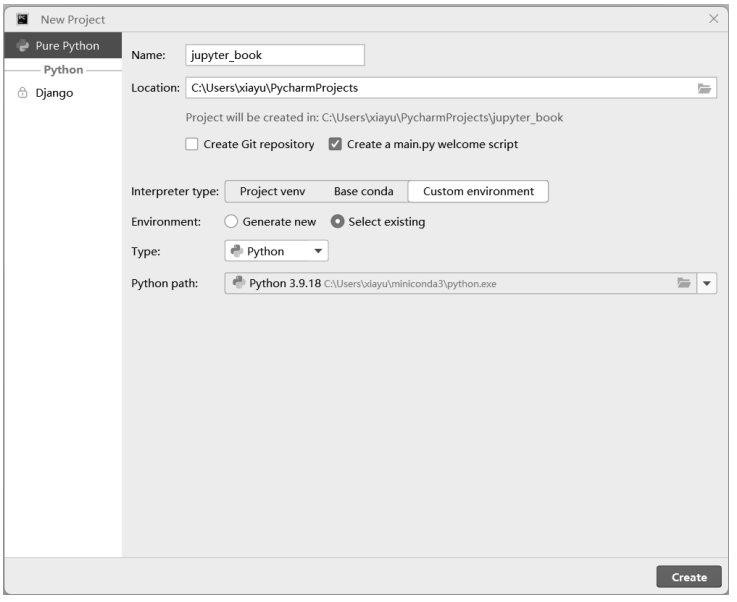

图 2-14　创建一个新的工程

单击 Create 按钮，新建一个 PyCharm 项目，如图 2-15 所示。之后右击新建的项目名 jupyter_book，选择 New 菜单项，可以在本项目下创建目录、Python 文件等。比如，选择 New→Python File 菜单，新建一个 helloworld.py 文件，并输入一条简单的打印代码，内容如图 2-16 所示。

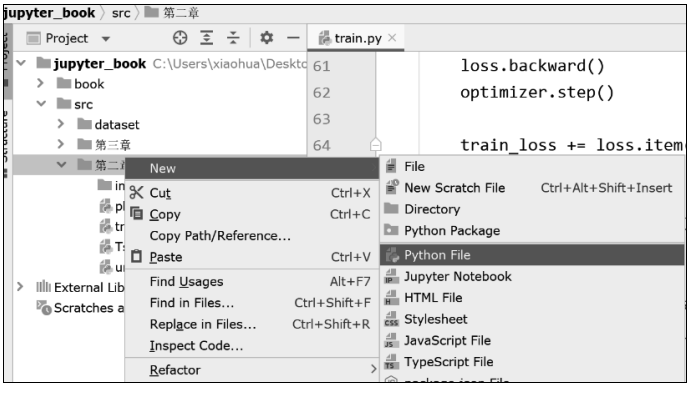

图 2-15　新建一个 PyCharm 的工程文件

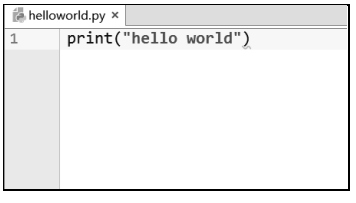

图 2-16　helloworld.py

输入代码后，单击菜单栏的 Run→run…运行代码，或者直接右击 helloworld.py 文件名，在弹出的快捷菜单中选择 run。如果成功输出 hello world，那么恭喜你，Python 与 PyCharm 的配置就顺利完成了。读者可以尝试把本书配套的示例代码作为项目加入 PyCharm 中调试和运行。

2.1.3　Python 代码小练习：计算 softmax 函数

对于 Python 科学计算来说，一个简单的想法是将数学公式直接表达成程序语言，可以说，Python 满足了这个想法。本小节将使用 Python 实现和计算一个深度学习中最为常见的函数——softmax 函数。至于这个函数的作用，现在不加以说明，只是带领读者尝试实现其程序的编写。

softmax 计算公式如下：

$$S_i = \frac{e^{V_i}}{\sum_{0}^{j} e^{V_i}}$$

其中，V_i 是长度为 j 的数列 V 中的一个数，代入 softmax 的结果其实就是先对每一个 V_i 取以 e 为底的指数计算变成非负，然后除以所有项之和进行归一化，之后每个 V_i 就可以解释成：在观察到的数据集类别中，特定的 V_i 属于某个类别的概率，或者称作似然（Likelihood）。

提示：softmax 用以解决概率计算中概率结果大而占绝对优势的问题。例如，函数计算结果中的两个值 a 和 b，且 a>b，如果简单地以值的大小为单位衡量的话，那么在后续的使用过程中，a 永远被选用，而 b 由于数值较小而不会被选择，但是有时候也需要使用数值较小的 b，softmax 就可以解决这个问题。

softmax 按照概率选择 a 和 b，由于 a 的概率值大于 b，在计算时 a 经常会被取得，而 b 由于概率较小，取得的可能性也较小，但是也有概率被取得。

softmax 公式的代码如下：

```
#演示的是 softmax 函数，目标是让读者熟悉 Python 程序设计
import numpy
def softmax(inMatrix):
m,n = numpy.shape(inMatrix)
outMatrix = numpy.mat(numpy.zeros((m,n)))
soft sum = 0
for idx in range(0,n):
    outMatrix[0,idx] = math.exp(inMatrix[0,idx])
    soft sum += outMatrix[0,idx]
for idx in range(0,n):
    outMatrix[0,idx] = outMatrix[0,idx] / soft sum
return outMatrix
```

可以看到，当传入一个数列后，分别计算每个数值对应的指数函数值，之后将其相加后计算每个数值在数值和中的概率。

```
a = numpy.array([[1,2,1,2,1,1,3]])
```

结果请读者自行打印验证。

2.2　环境搭建 2：安装 PyTorch 2.0

Python 运行环境调试完毕后，下面的重点就是安装本书的主角——PyTorch 2.0。如果没有 GPU 显卡，从 CPU 版本的 PyTorch 开始深度学习之旅是完全可以的，但却不是推荐的方式。相对于 GPU 版本的 PyTorch 来说，CPU 版本的运行速度存在着极大的劣势，很有可能会让你的深度学习止步于前。

PyTorch 2.0 CPU 版本的安装命令如下：

```
    pip install numpy --pre torch==2.0.1 torchvision torchaudio --force-reinstall
--extra-index-url https://download.pytorch.org/whl/nightly/cpu
```

2.2.1　Nvidia 10/20/30/40 系列显卡选择的 GPU 版本

由于 40 系列显卡的推出，目前市场上会有 Nvidia 10、20、30、40 系列显卡并存的情况。对于需要调用专用编译器的 PyTorch 来说，不同的显卡需要安装不同的依赖计算包，在此总结了不同显卡的 PyTorch 版本以及 CUDA 和 cuDNN 的对应关系，如表 2-1 所示。

表 2-1　Nvidia 10/20/30/40 系列显卡的版本对比

显卡型号	PyTorch GPU 版本	CUDA 版本	cuDNN 版本
10 系列及以前	PyTorch 2.0 以前的版本	11.1	7.65
20/30/40 系列	PyTorch 2.0 向下兼容	11.6+	8.1+

这里主要是显卡运算库 CUDA 与 cuDNN 版本的搭配。在 10 系列版本的显卡上，建议优先使用 2.0 版本以前的 PyTorch。在 20/30/40 系列显卡上使用 PyTorch 时，可以参考官方网站 https://developer.nvidia.com/rdp/cudnn-archive，按照本机安装的 CUDA 版本选择相应的 cuDNN 版本进行搭配。比如：

- cuDNN v8.9.6 (November 1st, 2023), for CUDA 12.x
- cuDNN v8.9.6 (November 1st, 2023), for CUDA 11.x
- cuDNN v8.9.5 (October 27th, 2023), for CUDA 12.x
- cuDNN v8.9.5 (October 27th, 2023), for CUDA 11.x

下面以 PyTorch 2.0 为例，演示完整的 CUDA 和 cuDNN 的安装步骤，不同的版本安装过程基本一致，只是需要注意一下软件版本之间的搭配问题。

2.2.2　PyTorch 2.0 GPU Nvidia 运行库的安装

本小节讲解 PyTorch 2.0 GPU 版本的前置软件的安装。对于 GPU 版本的 PyTorch 来说，由于调用了 NVIDA 显卡作为其代码运行的主要工具，因此额外需要 NVIDA 提供的运行库作为运行基础。

我们选择 PyTorch 2.0 版本进行讲解。对于 PyTorch 2.0 的安装来说，最好的方法是根据官方网站提供的安装命令进行安装，具体参考官方网站文档 https://pytorch.org/get-started/previous-versions/。从页面上可以看到，针对 Windows 版本的 PyTorch 2.0 官方网站提供了几种安装模式，分别对应

CUDA 11.7、CUDA 11.8 和 CPU only。使用 conda 安装的命令如下：

```
    # CUDA 11.7
    conda install pytorch==2.0.1 torchvision==0.15.2 torchaudio==2.0.2
pytorch-cuda=11.7 -c pytorch -c nvidia
    # CUDA 11.8
    conda install pytorch==2.0.1 torchvision==0.15.2 torchaudio==2.0.2
pytorch-cuda=11.8 -c pytorch -c nvidia
    # CPU Only
    conda install pytorch==2.0.1 torchvision==0.15.2 torchaudio==2.0.2 cpuonly -c
pytorch
```

使用 pip 直接安装的命令如下：

```
    # CUDA 11.7
    pip install torch==2.0.1 torchvision==0.15.2 torchaudio==2.0.2
    # CUDA 11.8
    pip install torch==2.0.1 torchvision==0.15.2 torchaudio==2.0.2 --index-url
https://download.pytorch.org/whl/cu118
    # CPU only
    pip install torch==2.0.1 torchvision==0.15.2 torchaudio==2.0.2 --index-url
https://download.pytorch.org/whl/cpu
```

下面我们以 CUDA 11.8+cuDNN 8.9 为例讲解安装的方法。

（1）CUDA 的安装，在百度搜索 CUDA 11.8 download，进入官方网站下载页面，选择适合的操作系统安装方式（推荐使用 exe(local)本地化安装方式），如图 2-17 所示。

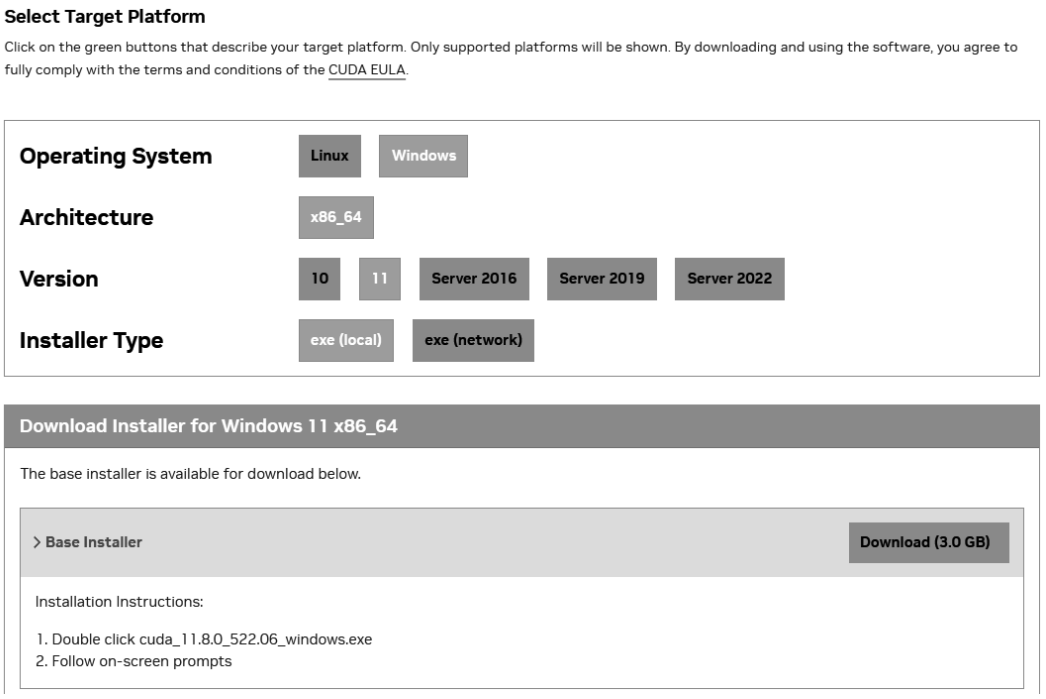

图 2-17　CUDA 11.8 下载页面

此时下载下来的是一个 EXE 文件，读者自行安装，不要修改其中的路径信息，完全使用默认路径安装即可。

（2）下载和安装对应的 cuDNN 文件。cuDNN 的下载需要先注册一个用户，相信读者可以很快完成，之后直接进入下载页面，如图 2-18 所示。注意：不要选择错误的版本，一定要找到对应 CUDA 的版本号。另外，如果使用的是 Windows 64 位的操作系统，需要下载 x86_64 版本的 cuDNN。

图 2-18　cuDNN 8.9.4 下载页面

（3）下载的 cuDNN 是一个压缩文件，将其解压并把其所有的目录复制到 CUDA 安装主目录中（直接覆盖原来的目录），CUDA 安装主目录如图 2-19 所示。

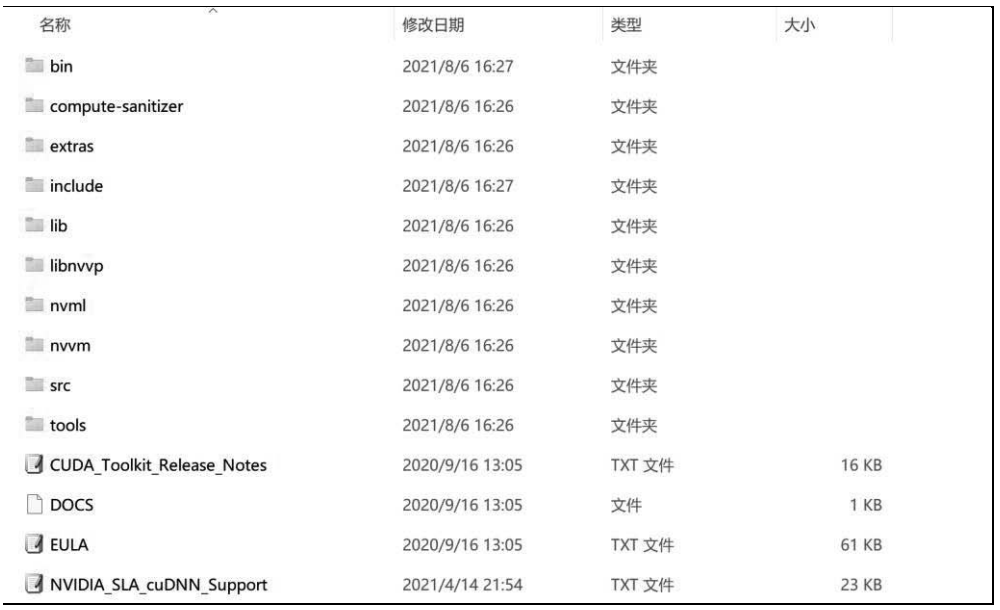

图 2-19　CUDA 安装主目录

（4）接下来确认一下 PATH 环境变量，这里需要将 CUDA 运行路径加载到环境变量的 PATH 路径中。安装 CUDA 时，安装向导能自动加入这个环境变量值，确认一下即可，如图 2-20 所示。

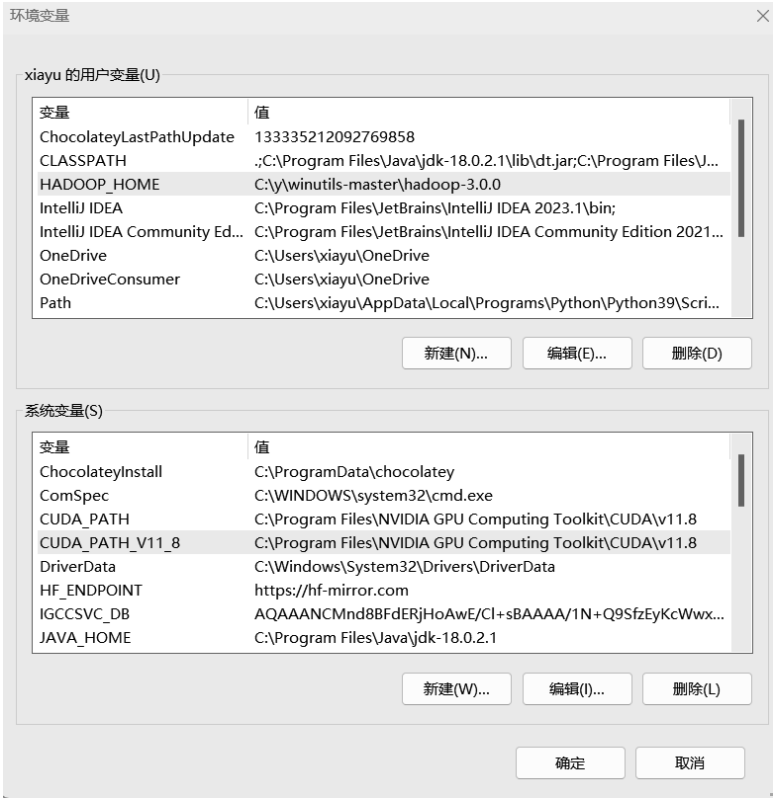

图 2-20　将 CUDA 路径加载到环境变量的 path 中

（5）最后完成 PyTorch 2.0 GPU 版本的安装，只需要在终端窗口中执行本小节开始给出的 PyTorch 安装命令即可。

2.2.3　PyTorch 2.0 小练习：Hello PyTorch

恭喜读者，至此已经完成了 PyTorch 2.0 的安装。打开 CMD 窗口，执行 python 命令进入交互模式，在窗口中输入如下代码，可以验证安装是否成功：

```
#验证安装 PyTorch
import torch
result = torch.tensor(1) + torch.tensor(2.0)
result
```

结果如图 2-21 所示。

```
PS C:\Users\xiayu> python
Python 3.9.10 (tags/v3.9.10:f2f3f53, Jan 17 2022, 15:14:21) [MSC v.1929 64 bit (AMD64)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> result = torch.tensor(1) + torch.tensor(2.0)
>>> result
tensor(3.)
>>>
```

图 2-21　验证安装是否成功

或者使用 PyCharm 打开本书示例项目，在第 2 章目录下新建一个 hello_pytorch.py 文件，输入如下代码：

```
#验证安装 PyTorch
import torch
result = torch.tensor(1) + torch.tensor(2.0)
print(result)
```

最终结果请读者自行验证。

2.3　实战：基于特征词的语音唤醒

本章前面介绍了纯理论知识，目的是阐述语音识别的方法。接着搭建了开发环境，让读者可以动手编写代码。下面以识别特定词为例，使用深度学习方法和Python语言实现一个实战项目——基于特征词的语音唤醒。

说明：本例的目的是演示一个语音识别的 Demo。如果读者已经安装开发环境，可以直接复制代码运行；如果没有，可学习完本章后再回头练习。我们会在本节中详细介绍每一步的操作过程和设计方法。

2.3.1　数据的准备

深度学习的第一步（也是重要的步骤）是数据的准备。数据的来源多种多样，既有不同类型的数据集，也有根据项目需求由项目组自行准备的数据集。由于本例的目的是识别特定词语而进行语音唤醒，因此采用一整套专门的语音命令数据集 SpeechCommands，我们可以使用 PyTorch 专门的下载代码获取完整的数据集，代码如下：

```
#直接下载 PyTorch 数据库的语音文件
from torchaudio import datasets

datasets.SPEECHCOMMANDS(
    root="./dataset",                # 保存数据的路径
    url='speech_commands_v0.02',     # 下载数据版本 URL
    folder_in_archive='SpeechCommands',
    download=True                    # 这个记得选 True
)
```

SpeechCommands 的数据量约为 2GB，等待下载完毕后，可以在下载路径中查看下载的数据集，如图 2-22 所示。

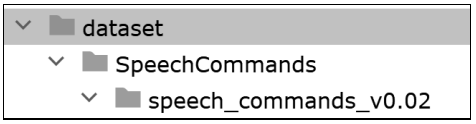

图 2-22　下载的数据集

打开数据集可以看到，根据不同的文件夹名称，其中内部被分成了 40 个类别，每个类别以名称命名，包含符合该文件名的语音发音，如图 2-23 和图 2-24 所示。

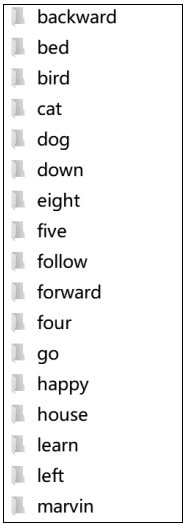

图 2-23　SpeechCommands 数据集

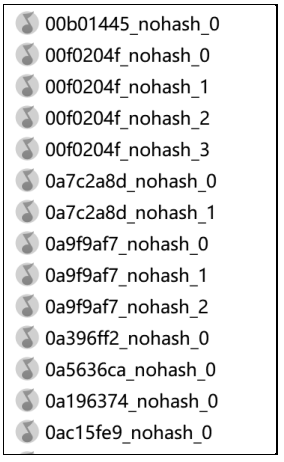

图 2-24　特定文件夹内部的内容

可以看到，根据文件名对每个发音进行归类，其中包含：

- 训练集包含51088个WAV语音文件。
- 验证集包含6798个WAV语音文件。
- 测试集包含6835个WAV语音文件。

读者可以使用计算机自带的语音播放程序试听部分语音。

2.3.2　数据的处理

下面开始进入这个语音识别 Demo 的代码实现部分。相信读者已经试听过部分语音内容，摆在读者面前的第一个难题是，如何将语音转换成计算机可以识别的信号。

梅尔频率是基于人耳听觉特性提出来的，它与 Hz 频率呈非线性对应关系。梅尔频率倒谱系数（Mel-Frequency Cepstral Coefficients，MFCC）则是利用它们之间的这种关系计算得到的 Hz 频谱特征，主要用于语音数据特征提取和降低运算维度。例如，对于一帧有 512 维（采样点）的数据，经过 MFCC 后可以提取出最重要的 40 维（一般而言），数据同时也达到了降维的目的。

这里，我们将 MFCC 理解成使用一个“数字矩阵”来替代一段语音即可。计算 MFCC 实际上是一个烦琐的任务，需要使用专门的类库来实现对语音 MFCC 的提取，代码处理如下：

【程序2-1】

```
import os
import numpy as np

#获取文件夹中所有的文件地址
def list_files(path):
```

```
        files = []
        for item in os.listdir(path):
            file = os.path.join(path, item)
            if os.path.isfile(file):
                files.append(file)
        return files

    # 这里是一个对单独序列的 cut 或者 pad 的操作
    # 这里输入的 y 是一个一维的序列，将输入的一维序列 y 拉伸或者裁剪到 length 长度
    def crop or pad(y, length, is train=True, start=None):
        if len(y) < length:     #对长度进行判断
            y = np.concatenate([y, np.zeros(length - len(y))])  #若长度过短则进行补
全操作

            n_repeats = length // len(y)
            epsilon = length % len(y)
            y = np.concatenate([y] * n_repeats + [y[:epsilon]])

        elif len(y) > length:   #对长度进行判断，若长度过长，则进行裁剪操作
            if not is_train:
                start = start or 0
            else:
                start = start or np.random.randint(len(y) - length)

            y = y[start:start + length]
        return y

    import librosa as lb
    # 计算梅尔频率图
    def compute_melspec(y, sr, n_mels, fmin, fmax):
        """
        :param y:传入的语音序列，每帧的采样
        :param sr: 采样率
        :param n_mels: 梅尔滤波器的频率倒谱系数
        :param fmin: 短时傅里叶变换(STFT)的分析范围 min
        :param fmax: 短时傅里叶变换(STFT)的分析范围 max
        :return:
        """
        # 计算 Mel 频谱图的函数
        melspec = lb.feature.melspectrogram(y=y, sr=sr, n_mels=n_mels, fmin=fmin,
fmax=fmax)  # (128, 1024) 这个是输出一个声音的频谱矩阵
        # Python 中用于将语音信号的功率值转换为分贝(dB)值的函数
        melspec = lb.power_to_db(melspec).astype(np.float32)
        return melspec

    # 对输入的频谱矩阵进行正则化处理
    def mono_to_color(X, eps=1e-6, mean=None, std=None):
        mean = mean or X.mean()
        std = std or X.std()
        X = (X - mean) / (std + eps)
```

```
        min,  max = X.min(), X.max()

        if ( max -  min) > eps:     #对越过阈值的内容进行处理
            V = np.clip(X,  min,  max)
            V = 255. * (V -  min) / ( max -  min)
            V = V.astype(np.uint8)
        else:
            V = np.zeros like(X, dtype=np.uint8)
        return V

    #创建语音特征矩阵
    def audio_to_image(audio, sr, n_mels, fmin, fmax):
        #获取梅尔频率图
        melspec = compute_melspec(audio, sr, n_mels, fmin, fmax)
        #进行正则化处理
        image = mono_to_color(melspec)
        return image
```

使用创建好的 MFCC 生产函数获取特定语音的 MFCC 矩阵也很容易，代码如下：

```
    import numpy as np
    from torchaudio import datasets
    import sound_untils
    import soundfile as sf

    print("开始数据处理")
    target_classes = ["bed","bird","cat","dog","four"]

    counter = 0
    sr = 16000
    n_mels = 128
    fmin = 0
    fmax = sr//2
    file_folder = "./dataset/SpeechCommands/speech_commands_v0.02/"
    labels = []
    sound features = []
    for classes in target classes:
        target folder = file folder + classes
        files = sound untils.list files(target folder)
        for  file in  files:
            audio, orig sr = sf.read( file, dtype="float32")  # 这里均值是 1308338，
0.8 中位数是 1730351，所以这里采用了中位数的部分
            audio = sound untils.crop or pad(audio, length=orig sr)  # 作者的想法是
把 audio 做一个整体输入，在这里所有的都做了输入
            image = sound untils.audio to image(audio, sr, n mels, fmin, fmax)
            sound features.append(image)
            label = target classes.index(classes)
            labels.append(label)
    sound features = np.array(sound features)
    print(sound_features.shape) #(11965, 128, 32)
```

```
print(len(labels))                #(11965, 128, 32)
```

最终打印结果如下：

```
(11965,128,32)
11965
```

可以看到，根据作者设定的参数，特定路径指定的语音被转换成一个固定大小的矩阵，这也是根据前面超参数的设定而计算出的一个特定矩阵。有兴趣的读者可以将其打印出来并观察其内容。

2.3.3　模型的设计

对于深度学习而言，模型的设计是非常重要的步骤，由于本节的实战案例只是用于演示，因此采用了最简单的判别模型，实现代码如下（仅供读者演示，详细的内容在后续章节中介绍）：

【程序2-2】

```
#这里使用 ResNet 作为特征提取模型，仅供读者演示，详细的内容在后续章节中介绍
import torch

class ResNet(torch.nn.Module):
    def __init__(self,inchannels = 32):
        super(ResNet, self).__init__()
        #定义初始化神经网络层
        self.cnn_1 = torch.nn.Conv1d(inchannels,inchannels*2,3,padding=1)
        self.batch_norm = torch.nn.BatchNorm1d(inchannels*2)
        self.cnn_2 = torch.nn.Conv1d(inchannels*2,inchannels,3,padding=1)
        self.logits = torch.nn.Linear(128 * 32,5)

    def forward(self,x):
        #使用初始化定义的神经网络计算层进行计算
        y = self.cnn_1((x.permute(0, 2, 1)))
        y = self.batch_norm(y)
        y = y.permute(0, 2, 1)
        y = torch.nn.ReLU()(y)
        y = self.cnn_2((y.permute(0, 2, 1))).permute(0, 2, 1)
        output = x + y
        output = torch.nn.Flatten()(output)
        logits = self.logits(output)
        return logits

if __name__ == '__main__':
    image = torch.randn(size=(3,128,32))
    ResNet()(image)
```

上面代码中的 ResNet 类继承自 torch 中的 nn.Module 类，目的是创建一个可以运行的深度学习模型，并在 forward 函数中通过神经网络进行计算，最终将计算结果作为返回值返回。

2.3.4　模型的数据输入方法

接下来设定模型的数据输入方法。深度学习模型的每一步都需要数据内容的输入，但是，一般

情况下，由于计算硬件——显存的大小有限制，因此在输入数据时需要分步骤将数据一块一块地输入训练模型中。此处数据输入的实现代码如下：

【程序2-3】

```
#创建基于 PyTorch 的数据读取格式
import torch
class SoundDataset(torch.utils.data.Dataset):
    #初始化数据读取地址
    def __init__(self, sound_features = sound_features,labels = labels):

        self.sound_features = sound_features
        self.labels = labels

    def __len__(self):
        return len(self.labels)  #获取完整的数据集长度

    def __getitem__(self, idx):
        #对数据进行读取，在模板中每次读取一个序号指向的数据内容
        image = self.sound_features[idx]
        image = torch.tensor(image).float()      #对读取的数据进行类型转换

        label = self.labels[idx]
        label = torch.tensor(label).long()       #对读取的数据进行类型转换

        return image, label
```

在上面代码中，首先根据传入的数据在初始化时生成供训练使用的训练数据和对应的标签，之后的 getitem 函数建立了一个“传送带”，目的是源源不断地将待训练数据传递给训练模型，从而完成模型的训练。

2.3.5　模型的训练

对模型进行训练时，需要定义模型的一些训练参数，如优化器、损失函数、准确率以及训练的循环次数等。模型训练的实现代码如下（这里不要求读者理解，能够运行即可）：

【程序2-4】

```
import torch
from torch.utils.data import DataLoader, Dataset
from tqdm import tqdm

device = "cuda"

from sound_model import ResNet
sound_model = ResNet().to(device)

BATCH_SIZE = 32
LEARNING_RATE = 2e-5
```

```
    import get data
    #导入数据集
    train dataset = get data.SoundDataset()
    #以 PyTorch 生成数据模板进行数据的读取和输出操作
    train loader = (DataLoader(train dataset,
batch size=BATCH SIZE,shuffle=True,num workers=0))

    #PyTorch 中的优化器
    optimizer = torch.optim.AdamW(sound model.parameters(), lr = LEARNING RATE)
    #对学习率进行修正的函数
    lr scheduler = torch.optim.lr scheduler.CosineAnnealingLR(optimizer,T max =
1600,eta_min=LEARNING_RATE/20,last_epoch=-1)
    #定义损失函数
    criterion = torch.nn.CrossEntropyLoss()

    for epoch in range(9):
        pbar = tqdm(train_loader,total=len(train_loader))
        train_loss = 0.
        for token_inp,token_tgt in pbar:
            #将数据传入硬件中
            token_inp = token_inp.to(device)
            token_tgt = token_tgt.to(device)
            #采用模型进行计算
            logits = sound_model(token_inp)
            #使用损失函数计算差值
            loss = criterion(logits, token_tgt)
            #计算梯度
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            lr_scheduler.step()  # 执行优化器
            train_accuracy = ((torch.argmax(torch.nn.Softmax(dim=-1)(logits),
dim=-1) == (token_tgt)).type(torch.float).sum().item() / len(token_tgt))

            pbar.set_description(
                f"epoch:{epoch + 1}, train_loss:{loss.item():.4f},,
train_accuracy:{train_accuracy:.2f}, lr:{lr_scheduler.get_last_lr()[0] *
1000:.5f}")
```

上面代码完成了一个可以运行并持续对结果进行输出的训练模型，首先初始化模型的实例，之后建立数据的传递通道，而优化函数和损失函数也可以通过显式定义完成。

2.3.6　模型的结果和展示

在模型的结果展示中，我们使用 epochs=9，即运行 9 轮对数据进行训练，结果如图 2-25 所示。

可以看到，经过 9 轮训练后，准确率达到了 0.72%，这个成绩目前差强人意。提高模型准确率的方法有很多，例如采用更好的优化函数、变换损失函数的计算方式、增加训练次数、修正模型的设计、采用对比学习等。对于初学者来说，目前只需要掌握基本的流程即可。

```
开始数据处理
(11965, 128, 32)
11965
数据处理完毕~
epoch:1, train_loss:13.4581,, train_accuracy:0.45, lr:0.01755: 100%|██████████| 374/374 [00:03<00:00, 117.92it/s]
epoch:2, train_loss:7.6630,, train_accuracy:0.45, lr:0.01147: 100%|██████████| 374/374 [00:01<00:00, 294.65it/s]
epoch:3, train_loss:5.5695,, train_accuracy:0.62, lr:0.00489: 100%|██████████| 374/374 [00:01<00:00, 290.15it/s]
epoch:4, train_loss:5.5164,, train_accuracy:0.59, lr:0.00120: 100%|██████████| 374/374 [00:01<00:00, 301.74it/s]
epoch:5, train_loss:5.6408,, train_accuracy:0.55, lr:0.00230: 100%|██████████| 374/374 [00:01<00:00, 306.62it/s]
epoch:6, train_loss:4.5648,, train_accuracy:0.62, lr:0.00764: 100%|██████████| 374/374 [00:01<00:00, 313.19it/s]
epoch:7, train_loss:5.8655,, train_accuracy:0.48, lr:0.01444: 100%|██████████| 374/374 [00:01<00:00, 303.27it/s]
epoch:8, train_loss:6.4659,, train_accuracy:0.48, lr:0.01922: 100%|██████████| 374/374 [00:01<00:00, 307.03it/s]
epoch:9, train_loss:3.0568,, train_accuracy:0.72, lr:0.01950: 100%|██████████| 374/374 [00:01<00:00, 303.81it/s]
```

图 2-25　结果展示

2.4　本章小结

本章是PyTorch实战程序设计的开始，重点介绍了PyTorch程序设计的环境与基本软件的安装，演示了第一个基于 PyTorch 语音唤醒应用程序的整体设计过程，并结合审计过程介绍了部分模型组件（比如数据输入、数据处理、优化函数、损失函数等）。

实际上可以看到，深度学习应用程序的设计就是由一个个模型组件组合起来完成的，本书的后续章节会针对每个组件进行深入讲解。