

项目 1



AI 作曲

本项目基于 TensorFlow 开发环境,使用 LSTM 模型,搜集 MIDI 文件,进行特征筛选和提取,训练生成合适的机器学习模型,实现人工智能作曲。

1.1 总体设计

本部分包括整体框架和系统流程。

1.1.1 整体框架

整体框架如图 1-1 所示。

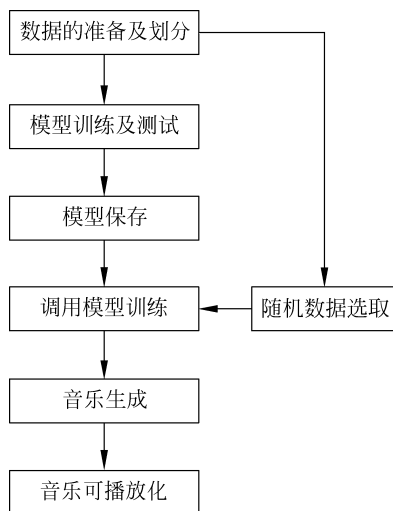


图 1-1 整体框架

1.1.2 系统流程

系统流程如图 1-2 所示。

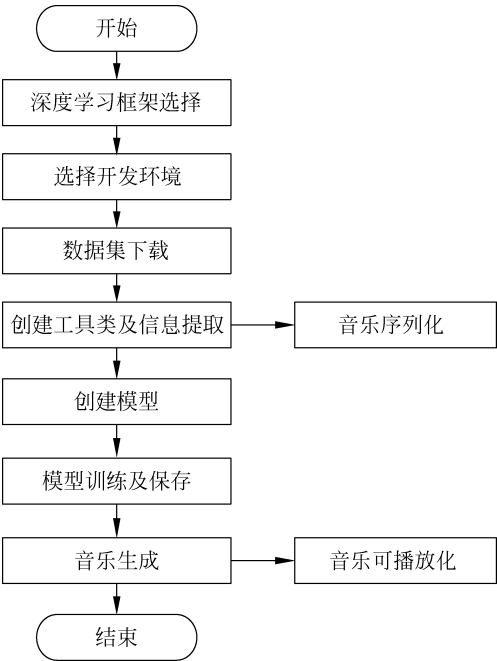


图 1-2 系统流程

1.2 运行环境

本部分包括 Python 环境、虚拟机环境、TensorFlow 环境及 Python 类库。

1.2.1 Python 环境

在 Windows 环境下下载 Anaconda,完成 Python 2.7 及以上版本的环境配置,如图 1-3 所示。

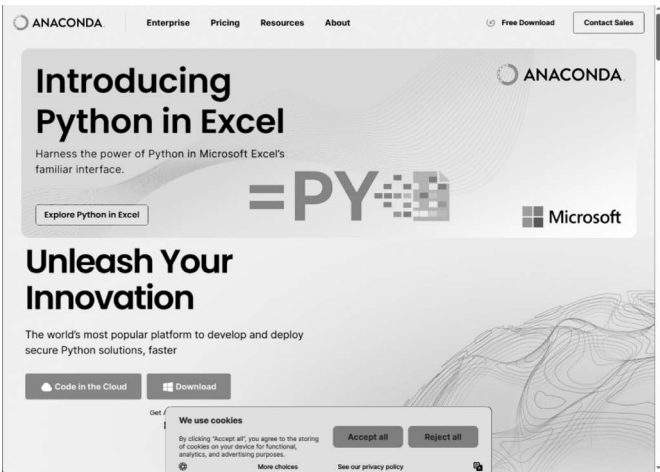


图 1-3 下载 Anaconda 界面

1.2.2 虚拟机环境

安装 VirtualBox,如图 1-4 所示。

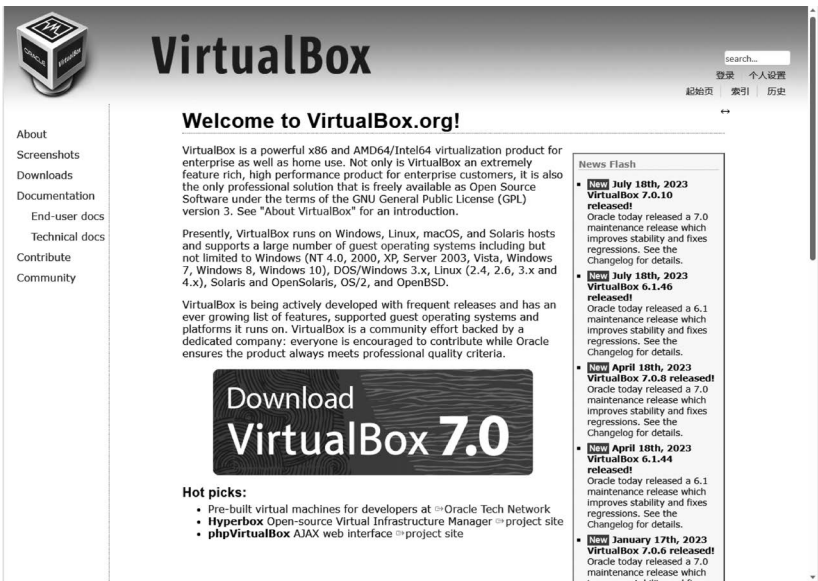


图 1-4 安装 VirtualBox 界面

如图 1-5 所示,安装 Ubuntu 系统,如果版本号是 16.04,那么它属于长期支持版。下载 Ubuntu GNOME,如图 1-6 所示。打开 VirtualBox,如图 1-7 所示。



图 1-5 安装 Ubuntu 系统

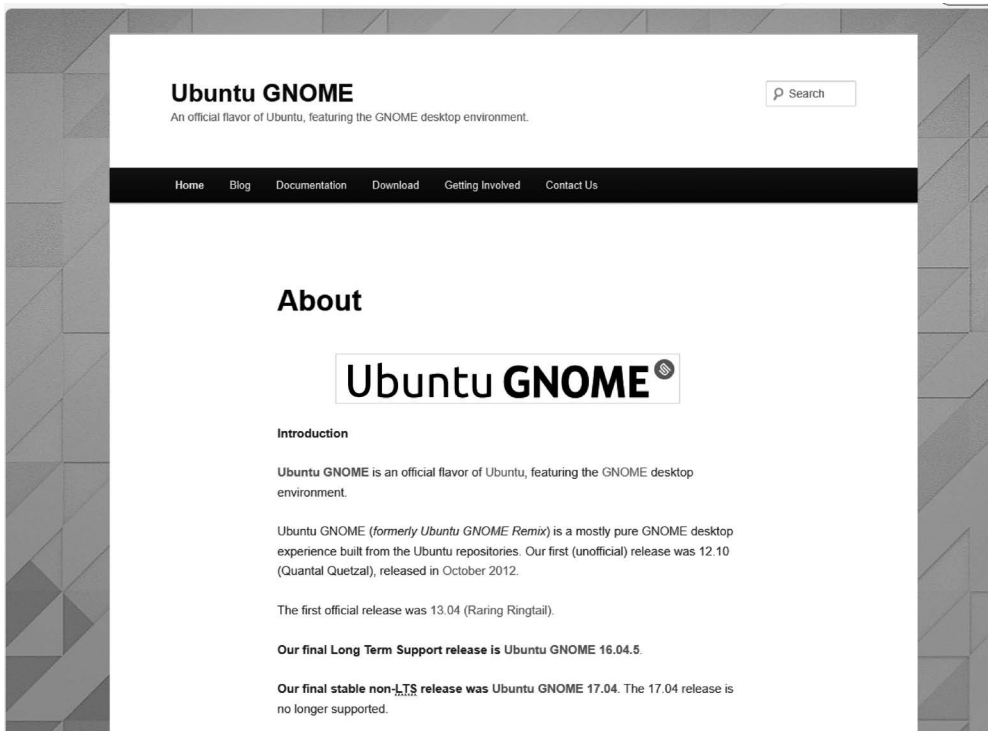


图 1-6 下载 Ubuntu GNOME 界面



图 1-7 打开 VirtualBox 界面

单击“新建”按钮,出现如图 1-8 所示的对话框。



图 1-8 虚拟机创建界面

名称可自行定义;类型选择 Linux;版本选择 Ubuntu(64-bit);内存大小可自行设置,建议设置为 2048MB 以上;虚拟硬盘选择默认选项,即现在创建虚拟硬盘,之后单击创建按钮,在文件位置和大小对话框中将虚拟硬盘更改为 20GB,虚拟机映像文件创建完成。对该映像文件右击进行设置,单击存储按钮进行保存。

选择没有盘片→分配光驱→虚拟光盘文件,添加下载好的 Ubuntu GnomeISO 镜像文件,然后单击 OK 按钮。选择 install Ubuntu GNOME→Continue→Install Now→Continue→Continue,在 Keyboard layout 对话框中选择 Chinese,单击 Continue 按钮,等待安装完成后单击 Restart Now 按钮即可。

(1) 进行 Ubuntu 的基本配置。

(2) 打开 Terminal,安装谷歌输入法,输入如下命令:

```
sudo apt install fcitx fcitx-googlepinyin im-config
```

(3) 安装 VIM,输入如下命令:

```
sudo apt install vim
```

(4) 创建与主机共享的文件夹,输入如下命令:

```
mkdir share_folder
sudo apt install virtualbox-guest-utils
```

(5) 创建主机文件夹 AIMM_Shared, 建立主机与虚拟机的共享路径, 输入如下命令:

```
sudo mount -t vboxsf AIMM_Shared home/share_folder
```

1.2.3 TensorFlow 环境

安装 TensorFlow, 如图 1-9 所示, 打开 Terminal, 输入如下命令:

```
sudo apt - get install python - pip python - dev python - virtualenv
virtualenv -- system - site - packages tensorflow
source ~/tensorflow/bin/activate
easy_install -U pip
pip install -- upgrade tensorflow
deactivate
```

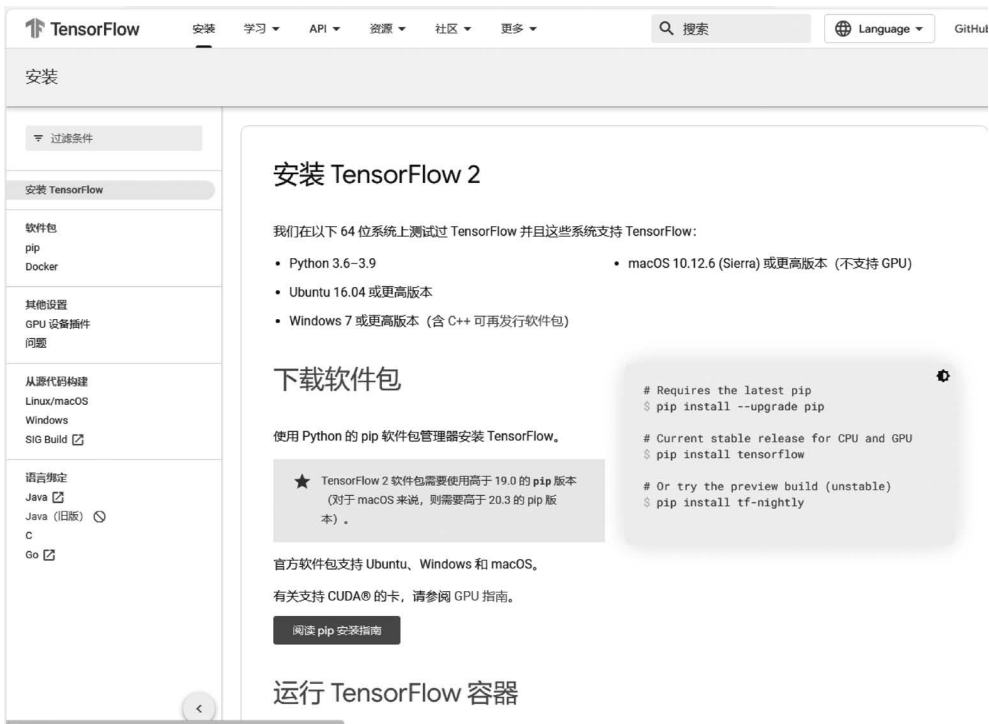


图 1-9 安装 TensorFlow 界面

1.2.4 Python 类库

安装 Python 的相关类库, 输入如下命令:

```
pip install numpy
pip install pandas
pip install matplotlib
sudo pip install keras
```

```
sudo pip install music21
sudo pip install h5py
sudo apt install ffmpeg
sudo apt install timidity
```

1.3 模块实现

本部分包括数据准备、信息提取、模型构建、模型训练及保存、音乐模块,下面分别给出各模块的功能介绍及相关代码。

1.3.1 数据准备

数据来自互联网下载的 70 首音乐文件,格式为 MIDI,相关数据见“数据集文件 1-1”,如图 1-10 所示。

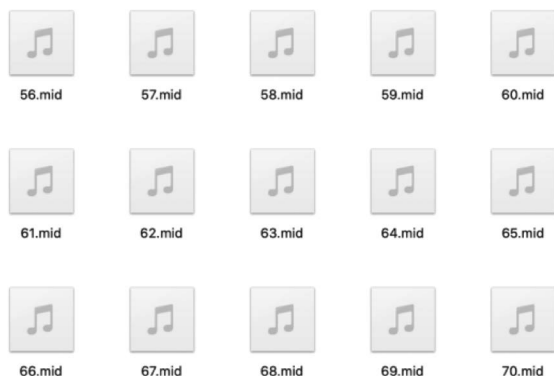


图 1-10 训练数据集

1.3.2 信息提取

数据准备完成后,需要进行文件格式转换及音乐信息提取。

1. 文件格式转换

使用 Timidity 软件,实现 MIDI 转换为 MP3 等其他流媒体格式的操作。相关代码见“代码文件 1-1”。

2. 音乐信息提取

将 MIDI 文件中的音符数据全部提取,包括 note 和 chord 的处理;note 指音符,而 chord 指和弦。使用的软件是 Music21,它可以对 MIDI 文件进行一些数据提取或者写入,相关代码如下。

```
import os
from music21 import converter, instrument
def print_notes():
    if not os.path.exists("1.mid"):
```

```

        raise Exception("MIDI 文件 1.mid 不在此目录下,请添加")
    # 读取 MIDI 文件,输出 Stream 流类型
    stream = converter.parse("1.mid")    # 解析 1.mid 的内容
    # 获得所有乐器部分
    parts = instrument.partitionByInstrument(stream)
    if parts:
        notes = parts.parts[0].recurse() # 如果有乐器部分,取第一个乐器部分,先采取一个音轨
    else:
        notes = stream.flat.notes
    # 输出每个元素
    for element in notes:
        print(str(element))
if __name__ == "__main__":
    print_notes()

```

1.3.3 模型构建

数据加载后,需要进行定义结构并优化损失函数。

1. 定义结构

图 1-11 为图形化的神经网络搭建模型,共 9 层,只用 LSTM 的 70%,舍弃 30%,这是为了防止过拟合,最后全连接层的音调数就是初始定义 num_pitch 的数目,用神经网络去预测每次生成的新音调属于所有音调中的哪一个,利用交叉熵和 Softmax(激活层)计算出概率最高那个作为输出(输出为预测音调对应的序列)。还需要在代码后面添加指定模型的损失函数并进行优化器设置。

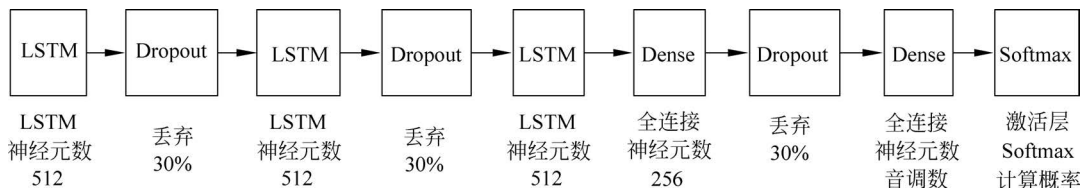


图 1-11 神经网络模型结构

```

# RNN - LSTM 循环神经网络
import tensorflow as tf
# 神经网络模型
def network_model(inputs, num_pitch, weights_file = None):
    model = tf.keras.models.Sequential()

```

构建一个神经网络模型(其中 Sequential 是序列的意思),在 TensorFlow 官网中可以看到基本用法,通过 add() 函数添加需要的层。Sequential 相当于一个汉堡模型,根据自己的需要按顺序填充不同层,相关代码如下。

```

# 模型框架,第 n 层输出会成为第 n+1 层的输入,一共 9 层
model.add(tf.keras.layers.LSTM(
    512, # LSTM 层神经元的数目是 512,也是 LSTM 层输出的维度
    input_shape = (inputs.shape[1], inputs.shape[2]),

```

```

# 输入的形状,对第一个 LSTM 层必须设置
# return_sequences: 控制返回类型
# True: 返回所有的输出序列
# False: 返回输出序列的最后一个输出
# 在堆叠 LSTM 层时必须设置,最后一层 LSTM 可以不用设置
return_sequences = True          # 返回所有的输出序列
))
# 丢弃 30 % 神经元,防止过拟合
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.LSTM(512, return_sequences = True))
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.LSTM(512))
# return_sequences 是默认的 False,只返回输出序列的最后一个
# 256 个神经元的全连接层
model.add(tf.keras.layers.Dense(256))
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.Dense(num_pitch))
# 输出的数目等于所有不重复的音调数目: num_pitch

```

2. 优化损失函数

确定神经网络模型架构之后,需要对模型进行编译,这是回归分析问题,因此先用 Softmax 计算百分比概率,再用 Cross entropy(交叉熵)计算概率和独热码之间的误差,使用 RMSProp 优化器优化模型参数,相关代码如下。

```

model.add(tf.keras.layers.Activation('softmax'))          # Softmax 激活函数计算概率
# 交叉熵计算误差,使用 RMSProp 优化器
# 计算误差
model.compile(loss = 'categorical_crossentropy', optimizer = 'rmsprop')
# 损失函数 loss,优化器 optimizer
if weights_file is not None: # 如果是生成音乐
    # 从 HDF5 文件中加载所有神经网络层的参数
    model.load_weights(weights_file)
return model

```

1.3.4 模型训练及保存

构建完整模型后,在训练模型之前需要准备输入序列。创建一个字典,用于映射音调和整数,同时还需要通过字典将整数映射成音调。除此之外,将输入序列的形状转换成神经网络模型可接收的形式,输入归一化。前面在构建神经网络模型时定义损失函数,用布尔的形式计算交叉熵,所以要将期望输出转换成 0 和 1 组成的布尔矩阵。

1. 模型训练

模型训练相关代码如下。

```

import numpy as np
import tensorflow as tf
from utils import *
from network import *

```

```
# 训练神经网络
def train():
    notes = get_notes()
    # 得到所有不重复的音调数目
    num_pitch = len(set(notes))
    network_input, network_output = prepare_sequences(notes, num_pitch)
    model = network_model(network_input, num_pitch)
    filepath = "weights-{epoch:02d}-{loss:.4f}.hdf5"
```

在训练模型之前,需要定义一个检查点,其目的是在每轮结束时保存模型参数(weights),在训练过程中不会丢失模型参数,而且在对损失满意时随时停止训练。根据官方文件提供的示例格式设置文件路径,不断更新保存模型参数 weights,格式为.hdf5。其中checkpoint 中参数设置 save_best_only=True,是指监视器 monitor=“loss”监视保存最好的损失,如果这次损失比上次损失小,则上次参数就会被覆盖,相关代码如下。

```
checkpoint = tf.keras.callbacks.ModelCheckpoint(
    filepath,                                # 保存的文件路径
    monitor='loss',                          # 监控的对象是损失(loss)
    verbose=0,
    save_best_only=True,
    mode='min'                               # 取损失最小的
)
callbacks_list = [checkpoint]
# 用 fit()函数训练模型
model.fit(network_input, network_output, epochs = 100, batch_size = 64, callbacks =
callbacks_list)
# 为神经网络准备好训练的序列
def prepare_sequences(notes, num_pitch):
    sequence_length = 100                    # 序列长度
    # 得到所有不重复音调的名字
    pitch_names = sorted(set(item for item in notes)) # sorted 用于字母排序
    # 创建一个字典,用于映射音调和整数
    pitch_to_int = dict((pitch, num) for num, pitch in enumerate(pitch_names))
    # enumerate 是枚举
    # 创建神经网络的输入序列和输出序列
    network_input = []
    network_output = []
    for i in range(0, len(notes) - sequence_length, 1):
        # 每隔一个音符就取前面的一百个音符用来训练
        sequence_in = notes[i: i + sequence_length]
        sequence_out = notes[i + sequence_length]
        network_input.append([pitch_to_int[char] for char in sequence_in])
```

Batch size 是批次(样本)数目,它是一次迭代所用的样本数目。Iteration 是迭代,每次迭代更新一次权重(网络参数),每次权重更新需要 Batch size 个数据前向运算后再进行反向运算,一个 Epoch 指所有的训练样本完成一次迭代。

2. 模型保存

训练神经网络后,将 weights 参数保存在 HDF5 文件中,相关代码如下。

```
# 将 sequence_in 中的每个字符转换成数字后存入 network_input 中
network_output.append(pitch_to_int[sequence_out])
n_patterns = len(network_input)
# 将输入的形状转换成神经网络模型可以接收的形式
network_input = np.reshape(network_input, (n_patterns, sequence_length, 1))
# 输入标准化/归一化
# 归一化可以使之后的优化器(optimizer)更快更好地找到误差最小值
network_input = network_input / float(num_pitch)
# 将期望输出转换成{0, 1}组成的布尔矩阵,配合误差算法使用
network_output = tf.keras.utils.to_categorical(network_output)
return network_input, network_output
if __name__ == '__main__':
    train()
```

1.3.5 音乐模块

本模块主要有序列准备、音符生成和音乐生成,主要作用如下:①为神经网络准备好供训练的序列;②基于序列音符,用神经网络生成新的音符;③用训练好的神经网络模型参数作曲。

在训练模型时用 fit() 函数,模型预测数据时用 predict() 函数得到最大的维度,也就是概率最高的音符。将实际预测的整数转换成音调保存,输入序列向后移动,不断生成新的音调。

1. 序列准备

序列准备相关代码如下。

```
def prepare_sequences(notes, pitch_names, num_pitch):
    # 为神经网络准备好供训练的序列
    sequence_length = 100
    # 创建一个字典,用于映射音调调整数
    pitch_to_int = dict((pitch, num) for num, pitch in enumerate(pitch_names))
    # 创建神经网络的输入序列和输出序列
    network_input = []
    network_output = []
    for i in range(0, len(notes) - sequence_length, 1):
        sequence_in = notes[i: i + sequence_length]
        sequence_out = notes[i + sequence_length]
        network_input.append([pitch_to_int[char] for char in sequence_in])
        network_output.append(pitch_to_int[sequence_out])
    n_patterns = len(network_input)
    # 将输入的形状转换成神经网络模型可以接收的形式
    normalized_input = np.reshape(network_input, (n_patterns, sequence_length, 1))
    # 输入标准化/归一化
    normalized_input = normalized_input / float(num_pitch)
    return network_input, normalized_input
```

2. 音符生成

音符生成相关代码如下。

```
def generate_notes(model, network_input, pitch_names, num_pitch):
    # 基于序列音符,用神经网络生成新的音符
    # 从输入中随机选择一个序列,作为预测/生成音乐的起始点
    start = np.random.randint(0, len(network_input) - 1)
    # 创建一个字典,用于映射整数和音调
    int_to_pitch = dict((num, pitch) for num, pitch in enumerate(pitch_names))
    pattern = network_input[start]
    # 神经网络实际生成的音调
    prediction_output = []
    # 生成 700 个音符/音调
    for note_index in range(700):
        prediction_input = np.reshape(pattern, (1, len(pattern), 1))
        # 输入归一化
        prediction_input = prediction_input / float(num_pitch)
        # 用载入了训练所得最佳参数文件的神经网络预测/生成新的音调
        prediction = model.predict(prediction_input, verbose=0)
        # argmax 取最大的维度
        index = np.argmax(prediction)
        # 将整数转成音调
        result = int_to_pitch[index]
        prediction_output.append(result)
        # 向后移动
        pattern.append(index)
        pattern = pattern[1:len(pattern)]
    return prediction_output
if __name__ == '__main__':
    generate()
```

3. 音乐生成

音乐生成相关代码如下。

```
# 使用之前训练所得的最佳参数生成音乐
def generate():
    # 加载用于训练神经网络的音乐数据
    with open('data/notes', 'rb') as filepath:
        notes = pickle.load(filepath)
    # 得到所有音调的名字
    pitch_names = sorted(set(item for item in notes))
    # 得到所有不重复的音调数目
    num_pitch = len(set(notes))
    network_input, normalized_input = prepare_sequences(notes, pitch_names, num_pitch)
    # 载入之前训练时最好的参数文件,生成神经网络模型
    model = network_model(normalized_input, num_pitch, "best-weights.hdf5")
    # 用神经网络生成音乐数据
```

```
prediction = generate_notes(model, network_input, pitch_names, num_pitch)
# 用预测的音乐数据生成 MIDI 文件,再转换成 MP3 格式
create_music(prediction)
```

1.4 系统测试

本部分包括训练过程及测试效果。

1.4.1 训练过程

运行 python train.py 并开始训练,默认训练 100 个 epoch,可使用组合键 Ctrl+C 结束训练,如图 1-12 所示。

```
Epoch 27/400
42685/42685 [=====]42685/42685 [=====] - 1858s 44ms/step - loss: 4.5118

Epoch 28/400
42685/42685 [=====]42685/42685 [=====] - 1855s 43ms/step - loss: 4.4739

Epoch 29/400
42685/42685 [=====]42685/42685 [=====] - 1853s 43ms/step - loss: 4.3547

Epoch 30/400
42685/42685 [=====]42685/42685 [=====] - 1853s 43ms/step - loss: 4.2431

Epoch 31/400
42685/42685 [=====]42685/42685 [=====] - 1850s 43ms/step - loss: 4.1182

Epoch 32/400
42685/42685 [=====]42685/42685 [=====] - 1849s 43ms/step - loss: 3.9861

Epoch 33/400
42685/42685 [=====]42685/42685 [=====] - 1847s 43ms/step - loss: 3.8438

Epoch 34/400
42685/42685 [=====]42685/42685 [=====] - 1845s 43ms/step - loss: 3.6849

Epoch 35/400
42685/42685 [=====]42685/42685 [=====] - 1842s 43ms/step - loss: 3.5315

Epoch 36/400
42685/42685 [=====]42685/42685 [=====] - 1844s 43ms/step - loss: 3.3884

Epoch 37/400
42685/42685 [=====]42685/42685 [=====] - 1845s 43ms/step - loss: 3.2341

Epoch 38/400
42685/42685 [=====]42685/42685 [=====] - 1845s 43ms/step - loss: 3.0969

Epoch 39/400
42685/42685 [=====]42685/42685 [=====] - 1849s 43ms/step - loss: 2.9628
```

图 1-12 训练过程

随着 epoch 增加,损失率越来越低,模型在训练数据、测试数据上的损失和准确率逐渐收敛,最终趋于稳定。

生成 MP3 格式的音乐时,先从 output.mid 生成 MIDI 文件,再从 output.mid 生成 output.mp3 文件——确保其位于 generate.py 同级目录下,运行 python generate.py 即可生成 MP3 格式的音乐。

1.4.2 测试效果

生成结果如图 1-13 所示,output.mid 是直接生成的 MIDI 文件,output.mp3 是转换后的 MP3 流媒体格式文件。



图 1-13 生成的 MIDI 文件和 MP3 文件

通过 Garage Band 尝试播放生成的音乐,如图 1-14 所示。



图 1-14 播放 output.mid

项目 2



语音识别

本项目基于卷积神经网络(Convolutional Neural Networks,CNN)对采样音频不同的声谱图进行识别,实现辨别数字声音。

2.1 总体设计

本部分包括整体框架和系统流程。

2.1.1 整体框架

整体框架如图 2-1 所示。

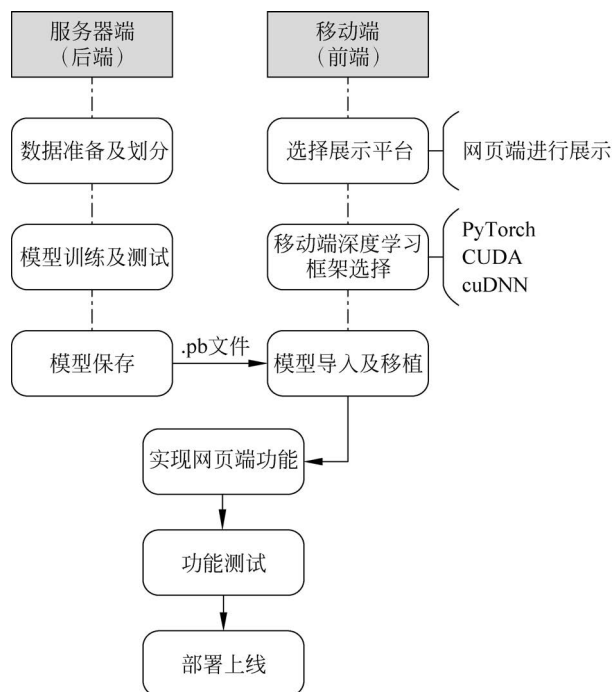


图 2-1 整体框架

2.1.2 系统流程

系统流程如图 2-2 所示。

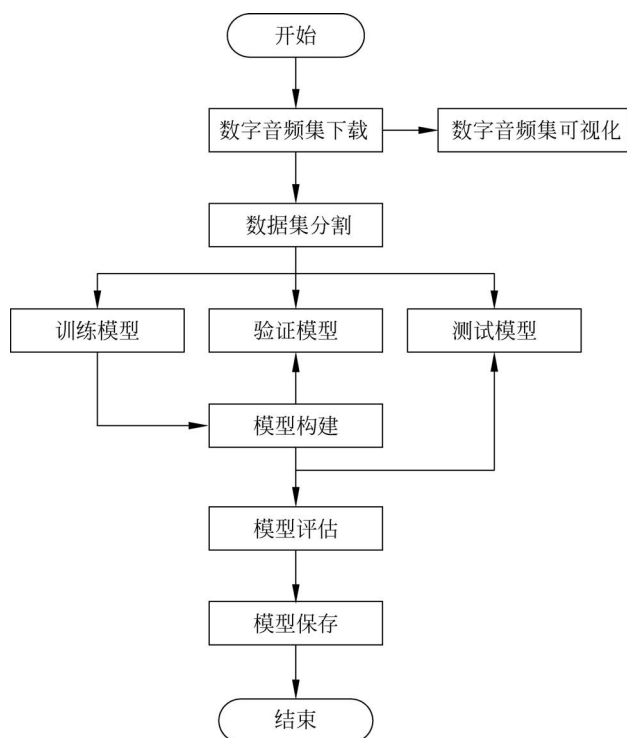


图 2-2 系统流程

2.2 运行环境

本部分包括 Python 环境、PyCharm 环境、PyTorch 环境、CUDA 和 cuDNN 环境。

2.2.1 Python 环境

在 Windows 环境下下载 Anaconda,完成 Python 的环境配置,如图 1-3 所示。

2.2.2 PyCharm 环境

PyCharm 需要在 Python 下载安装成功后再进行安装,如图 2-3 所示。

单击安装程序后选择路径配置相关环境,勾选所有选项后完成下载。选择项目所在路径→Previously configured interpreter→Create main.py→Create 进行项目搭建。右击 main.py,单击运行按钮,运行成功则代表环境配置成功。

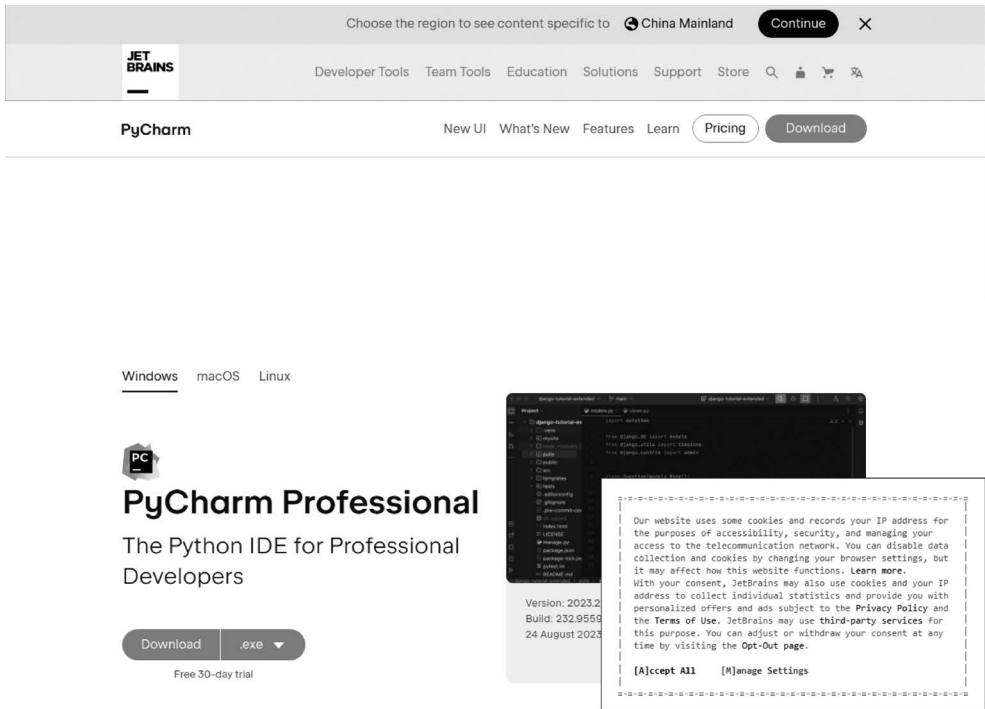


图 2-3 下载 PyCharm 界面

2.2.3 PyTorch 环境

首先,打开 Anaconda 命令提示行;其次,打开 Anaconda Prompt,前面显示(base)说明已经进入 Anaconda 的基础环境;最后,输入相关代码。

```
conda create -n pytorch1_11 python = 3.7
```

pytorch1_11 是创建环境的名称。python=3.7 是 Python 的版本。

选择与自己版本对应的命令行输入即可。Package 表示安装方式,在 Windows 下使用 Conda 即可,也可以选择 pip 虚拟指令对所有的第三方软件进行统一安装。

2.2.4 CUDA 和 cuDNN 环境

检查计算机所支持的下载版本,如图 2-4 所示,在 CUDA Toolkit 12.1.1(April 2023)中右击选择 nvidia→系统信息→组件。临时解压路径,建议默认即可,也可以自定义。安装结束后,临时解压文件夹会自动删除;选择自定义安装,安装完成后配置 CUDA 的环境变量;在命令行中,测试是否安装成功;双击.exe 文件,选择下载路径(推荐默认路径)。

cuDNN 作为 CUDA 的补丁环境,需要用到与 CUDA 相匹配的 11.3 版本,如图 2-5 所示。

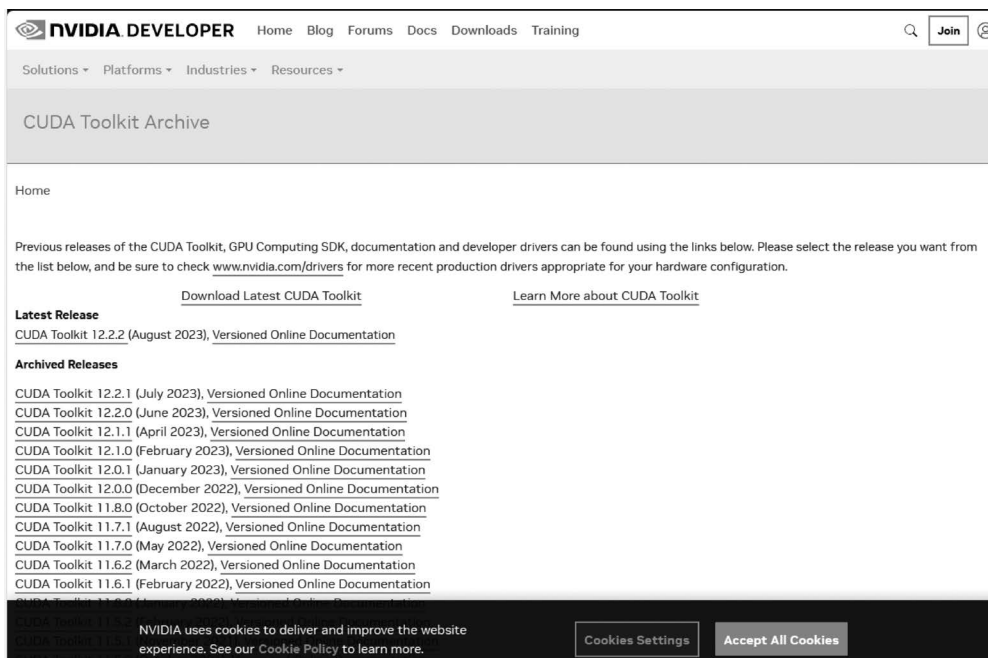


图 2-4 安装 CUDA 界面

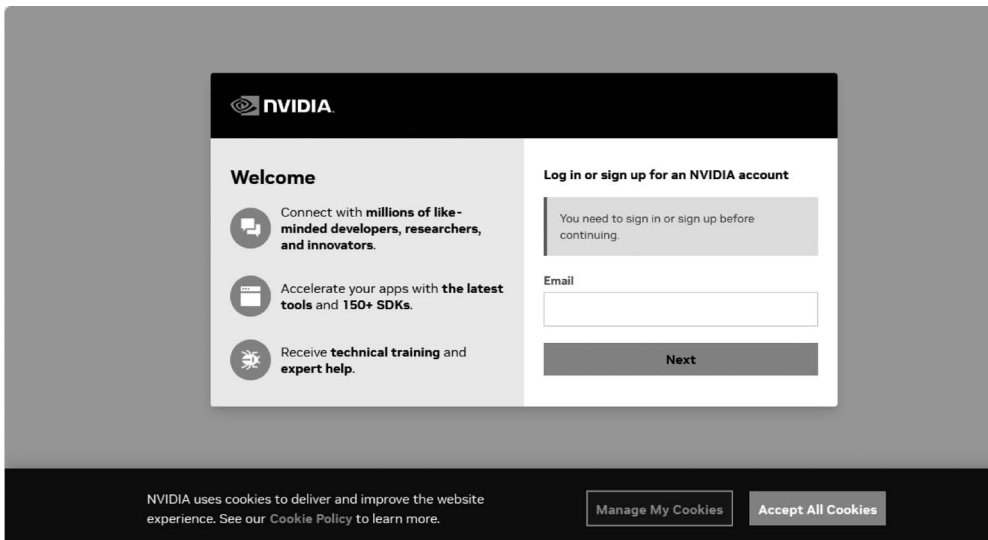


图 2-5 安装 cuDNN 界面

2.2.5 网页端配置环境

通过配置静态服务器端代码,将后端的结果展示于前端。如果请求的资源是 HTML 文件,可以认为是动态资源请求,需要将请求封装成 WSGI 协议要求的格式,并传输到 Web

框架处理。WSGI 协议要求服务器端将请求报文的各项信息组合成一个字典 environ, 同样传输到 Web 框架, 相关代码如下。

```
import socket
import threading
class MyWebServer(object):
    def __init__(self, port):
        # 初始化: 创建套接字
        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
        self.server_socket.bind(("localhost", port))
        self.server_socket.listen(128)
    def start(self):
        # 启动: 建立连接, 并开启子线程
        while True:
            new_socket, address = self.server_socket.accept()
            print("已连接", address, sep="from")
            sub_thread = threading.Thread(target = self.handle, args = (new_socket, ), daemon = True)
            sub_thread.start()
        @staticmethod
        def handle(handle_socket):
            # 利用子线程收发 HTTP 格式数据
            request_data = handle_socket.recv(4096)
            if len(request_data) == 0:
                print("浏览器已断开连接...")
                handle_socket.close()
            return
            request_content = request_data.decode("utf-8")
            print(request_content)
            # 以\r\n分割各项信息
            request_list = request_content.split("\r\n")
            # 提取请求的资源路径
            request_line = request_list[0]
            request_line_list = request_line.split(" ")
            request_method, request_path, request_version = request_line_list
            # 首页
            if request_path == "/":
                request_path = "/index.html"
            # 响应行与响应头信息置空
            response_line = response_header = ""
            # 根据请求路径准备好响应行和响应体
            try:
                with open(".", request_path, "rb") as request_file:
                    response_body = request_file.read()
            except (FileExistsError, FileNotFoundError):
                response_line += f"{request_version} 404 Not Found\r\n"
                with open("./error.html", "rb") as request_file:
                    response_body = request_file.read()
            else:
                response_line += f"{request_version} 200 OK\r\n"
            finally:
```

```
# 准备好响应头信息
response_header += "Server:MyWebServer2.0\r\n"
# 向浏览器发送响应报文
response_data = (response_line + response_header + "\r\n").encode("utf-8") + response_body
handle_socket.send(response_data)
# 断开与浏览器的连接
handle_socket.close()
def main():
    port = 8888
    my_web_server = MyWebServer(port)
    my_web_server.start()
    if __name__ == "__main__":
        main()
```

2.3 模块实现

本部分包括数据准备、模型构建、模型训练及保存、模型应用，下面分别给出各模块的功能介绍及相关代码。

2.3.1 数据准备

数据集使用 speech_commands，speech_commands 的 1.1v 数据集包含 500ms 的录音/波形，其中有一些随机的数字音频或者空白的音频。数据集界面如图 2-6 所示。每个波形



图 2-6 数据集界面