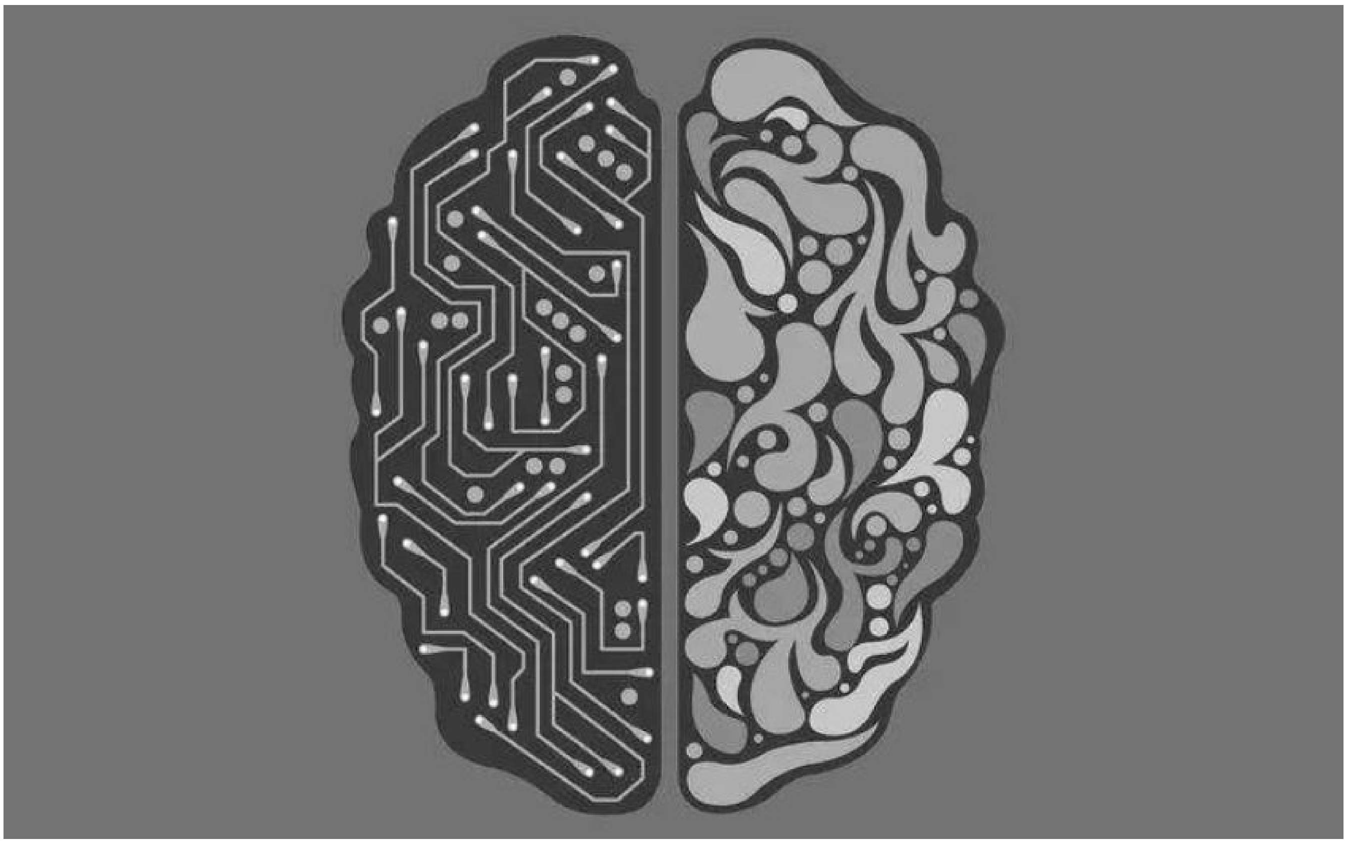
10万奖金语音识别赛进行中！自动化所博士生分享端到端Baseline

大数据文摘



大数据文摘出品

随着互联网、智能硬件的普及，智能音箱和语音助手已经深入人们的日常生活，家居场景下的语音识别技术已成为企业和研究机构竞相追逐的关键技术。

2019 年 12 月，北京智源人工智能研究院联合爱数智慧和数据评测平台 biendata，共同发布了“智源—MagicSpeechNet 家庭场景语音数据集挑战赛”（2019 年 12 月 — 2020年 3 月），总奖金为 10 万元。参赛者需要使用比赛提供的真实家庭环境中的双人对话音频数据，训练并优化语音识别（ASR）模型。比赛和数据复制下方链接查看，或点击“阅读原文”。

目前，赛事已接近半程，为便于选手熟悉和上手赛题，biendata 邀请长期处于排行榜前列的田正坤选手（中科院自动化所在读博士生，主要研究方向是语音识别，声纹识别和迁移学习）从赛题解析、数据处理、模型选择、提升方向等方面进行深入分析，希望可以抛砖引玉，为陷入瓶颈的选手提供灵感和思路，共同探索 ASR 实际应用场景中可行的解决方案。

比赛地址：
https://www.biendata.com/competition/magicdata/
Baseline 地址：
https://biendata.com/models/category/4162/L_notebook/

Baseline 详情

1. 赛题简介与分析

1.1 赛题引入

本次比赛的任务为日常家庭环境中的对话语音识别。所使用数据集为智源 MagicSpeechNet 家庭场景中文语音数据集，其中的语言材料来自数十段真实环境中的双人对话。每段对话基于多种平台进行录制，并已完全转录和标注。参赛者需要使用比赛提供的数据训练并优化模型，提升模型在家庭环境的对话语音识别效果。

1.2 数据集介绍

训练集数据，训练集数据包括音频（.wav）和对应的标注文件（.json），如音频“MDT_F2F_001.wav”对应“MDT_F2F_001.json”。 在 json 标注文件中，“start_time”表示该音频片段的开始时间，“end_time”表示音频片段的终止时间，“words”表示转录的文本,“speaker”表示音频的讲话人，“location”表示音频录制的地点，“session_id”表示音频片段所在的整段音频 ID。

开发集数据，开发集数据更为详细，每段对话有4个通道的同步录音，包括3个远讲通道和1个近讲通道。远讲通道包括由安卓平台、iOS 平台，录音笔录制的文件，如：

- MDT_Conversation_001_Android.wav
- MDT_Conversation_001_IOS.wav
- MDT_Conversation_001_Recorder.wav

近讲数据使用高保真麦克风录制，根据不同讲话人区分，如：

- MDT_Conversation_001_SPK001.wav
- MDT_Conversation_001_SPK002.wav

测试集数据，测试集数据为需要识别的音频文件，每段音频分为安卓平台、iOS 平台，录音笔录制的三个文件。为便于选手分割每段音频，比赛提供了标明起始和结束时间点信息的 json 文件，选手需使用模型识别音频中的对话，并根据 json 中对应的 uttid 提交相应的文本。

1.3 赛题分析

本次比赛的任务是基于给定数据训练一个语音识别系统，任务具有下面几个特点。

- **家庭场景日常对话**，家庭环境中的语音通常具有一定程度的噪声，在构建模型的过程中可以考虑对数据进行降噪处理，或者通过数据扩增的方法来提高模型的鲁棒性。
- **多设备录音**，开发集和测试集中均包含三种录音设备的录音，但是训练集并不是同一个录音都包含三种设备。因此最简单的方式就是不考虑设备，直接使用训练集进行模型训练（可以将开发集加入训练集进行训练）。在测试的过程中，从三种设备的录音中挑选一种设备的录音进行识别。

2. 数据分析与处理

2.1 数据分析

音频数据特点

本次比赛的场景是家庭对话场景，伴随着一定的噪声，因此在进行模型训练的过程中，应该考虑进行数据扩增，常见的方法包括速度扰动，音量扰动，谱增广（SpecAugment），可以采用多种扰动的叠加，能够有效的提高模型的鲁棒性。

模型开发和测试集包括多种设备的同步录音，可以将开发集中多通道数据直接并入训练集一起进行训练。在测试的时候，选择一种设备的录音进行解码即可（建议选择IOS，因为IOS的录音音质稍好一些）。

文本数据特点

在文本中包含一些特殊标记，其中：

- [*] 代表听不懂的说话内容(听不清句子/方言/严重口音/英文)
- [LAUGH] 代表笑声
- [SONANT] 代表由说话人发出的干扰浊音(咳嗽声/打喷嚏/清嗓子)
- [ENs] 代表各种独立噪音
- [MUSIC] 代表音乐声(唱歌声/哼唱/口哨声)

在训练集和开发集中，上述噪音在 json 标注文件中均已相应特殊符号呈现。在测试集中，如音频中出现噪音，在提供的对应 json 文件中，已经对噪音进行标注，选手无需再标注。由于在测试过程中没有仅有这些特殊符号构成的语音，因此，仅仅从比赛的角度上来讲可以将其直接过滤掉，这些无意义的符号也可以直接去掉，否则使得测试集表现下降。

2.2 数据处理

2.2.1 音频切分

由于给定音频文件太长，因此需要首先将音频文件切分出来，transcripts里面的json文件中已经包含了每句话起始和终止的时间，根据这个时间就可以对音频进行切分处理，这里可是使用sox工具。

```
# 将音频切分保存到wav文件夹中，并生成wav.scp和transcripts文件

# wav.scp文件格式:

# id1 path1

# id2 path2

# transcripts文件格式：

# id1 你好

# id2 今天天气怎么样

import os
import json
data_rootdir = './Magicdata'    # 指定解压后数据的根目录
audiodir = os.path.join(data_rootdir, 'audio')
trans_dir = os.path.join(data_rootdir, 'transcription')

\# 音频切分

def segment_wav(src_wav, tgt_wav, start_time, end_time):
        span = end_time - start_time
        cmds = 'sox %s %s trim %f %f'  % (src_wav, tgt_wav, start_time, span)
        os.system(cmds)

\# 将时间格式转化为秒为单位

def time2sec(t):
        h,m,s = t.strip().split(":")
        return  float(h) * 3600  + float(m) * 60  + float(s)

\# 读取json文件内容
```

```
def load_json(json_file):
        with open(json_file, 'r') as  f:
                lines = f.readlines()
                json_str = ''.join(lines).replace('\n', '').replace(' ', '').replace(',}',
                return  json.loads(json_str)

\# 训练集和开发集数据处理

for  name  in  ['train', 'dev']:
        save_dir = os.path.join('./data', name, 'wav')
        if  not os.path.exists(save_dir):
                os.makedirs(save_dir)
        seg_wav_list = []
        sub_audio_dir = os.path.join(audiodir, name)
        for  wav  in  os.listdir(sub_audio_dir):
                if  wav[0] == '.':
                        continue  # 跳过隐藏文件
                if  name == 'dev':
                        parts = wav.split('_')
                        jf = '_'.join(parts[:-1])+'.json'
                        suffix = parts[-1]
                else:
                        jf = wav[:-4]+'.json'
                utt_list = load_json(os.path.join(trans_dir, name, jf))
                for  i  in  range(len(utt_list)):
                        utt_info = utt_list[i]
                        session_id = utt_info['session_id']
                        if  name == 'dev':
                                tgt_id = session_id + '_'  + str(i) + '_'  + suffix
                        else:
                                tgt_id = session_id + '_'  + str(i) + '.wav'
                        # 句子切分
                        start_time = time2sec(utt_info['start_time']['original'])
                        end_time = time2sec(utt_info['end_time']['original'])
                        src_wav = os.path.join(sub_audio_dir, wav)
                        tgt_wav = os.path.join(save_dir, tgt_id)
                        segment_wav(src_wav, tgt_wav, start_time, end_time)
                        seg_wav_list.append((tgt_id, tgt_wav, utt_info['words']))
        with open(os.path.join('./data', name, 'wav.scp'), 'w') as  ww:
                with open(os.path.join('./data', name, 'transcrpts.txt'), 'w', encoding='u
                        for  uttid, wavdir, text in  seg_wav_list:
                                ww.write(uttid+' '+wavdir+'\n')
                                tw.write(uttid+' '+text+'\n')
        print('prepare %s dataset done!'  % name)
```

```
\# 测试集数据处理

save_dir = os.path.join('./data', 'test', 'wav')
if  not os.path.exists(save_dir):
        os.makedirs(save_dir)
seg_wav_list = []
sub_audio_dir = os.path.join(audiodir, 'test')
for  wav  in  os.listdir(sub_audio_dir):
        if  wav[0] == '.'  or 'IOS'  not in  wav:
                continue  # 跳过隐藏文件和非IOS的音频文件
        jf = '_'.join(wav.split('_')[:-1])+'.json'
        utt_list = load_json(os.path.join(trans_dir, 'test_no_ref_noise', jf))
        for  i in  range(len(utt_list)):
                utt_info = utt_list[i]
                session_id = utt_info['session_id']
                uttid = utt_info['uttid']
                if  'words'  in  utt_info: continue  # 如果句子已经标注，则跳过

                # 句子切分

start_time = time2sec(utt_info['start_time'])
end_time = time2sec(utt_info['end_time'])
tgt_id = uttid + '.wav'
src_wav = os.path.join(sub_audio_dir, wav)
tgt_wav = os.path.join(save_dir, tgt_id)
segment_wav(src_wav, tgt_wav, start_time, end_time)
seg_wav_list.append((uttid, tgt_wav))
with open(os.path.join('./data', 'test', 'wav.scp'), 'w') as  ww:
for  uttid, wavdir in  seg_wav_list:
ww.write(uttid+' '+wavdir+'\n')
print('prepare test dataset done!')
```

2.2.2 文本归一化处理

这里要对文本数据进行归一化处理，其中包括大写字母都转化为小写字母，过滤掉标点符号和无意义的句子。

```
# 过滤掉各种标点符号
# 过滤掉包含[*]、[LAUGH]、[SONANT]、[ENs]、[MUSIC]的句子
# 大写字母转小写

import os
import string
import zhon.hanzi

def text_normlization(seq):
        new_seq = []
        for  c  in seq:
                if  c  == '+':
                        new_seq.append(c) # 文档中有加号，所以单独处理，避免删除
                elif c  in string.punctuation or  c  in zhon.hanzi.punctuation:
                        continue  # 删除全部的半角标点和全角标点
                else:
                        if  c.encode('UTF-8').isalpha(): c  = c.lower() # 大写字母转小写
                        new_seq.append(c)
        return  ' '.join(new_seq)


for  name in ['train', 'dev']:
        with open(os.path.join('./data', name, 'transcrpts.txt'), 'r') as  tr:
                with open(os.path.join('./data', name, 'text'), 'w') as  tw:
                        for  line  in tr:
                                parts = line.split()
                                uttid = parts[0]
                                seqs = ''.join(parts[1:])
                                if  '['  in seqs:  continue  # 直接跳过包含特殊标记的句子
                                seqs = text_normlization(seqs)
                                tw.write(uttid+' '+seqs+'\n')

        print('Normlize %s TEXT!'  % name)
```

3. 思路分析

3.1 常见的语音识别方法

目前的语音识别方法可以大致分为混合模型和端到端模型结构。混合模型以Chain模型为代表（基于Kaldi工具包构建），是目前工业上主流的模型结构，其训练过程步骤相较繁琐。端到端模型是最近几年比较火的模型结构，以CTC模型、基于注意力机制的序列到序列模型，以及Transducer模型为代表，其中基于注意力机制的序列到序列模型效果最好。中文的端到端模型一般使用汉字作为建模单元，不需要发音词典等先验信息，直接将音频文本对输入模型就能直接进行训练，省去了繁琐的准备过程。并且在很多数据集例如

AISHELL上面，基于注意力机制的端到端模型的效果已经远超Chain模型，详情见ESPnet。

3.2 基线系统方法

本文中决定使用目前最好的端到端模型（Speech-Transformer）构建基线系统。

基于注意力机制的序列到序列模型结构包含编码器和解码器两部分：

- **编码器**，编码器用来将输入特征编码为高层的特征表示
- **解码器**，解码器从起始标记“”开始，利用注意力机制不断从编码特征中抽取有用信息并解码出一个汉字，不断重复这一过程，直到解码出结束标记“”，由此完成整个解码过程。

Speech-Transformer丢弃了传统序列到序列模型的循环神经网络结构，使用自注意力机制进行替代，不仅仅提高了模型的训练速度，也使得解码精度大大提高。详情可以参考论文Speech-Transformer:A No-Recurrence Sequence-to-Sequence Model for Speech Recognition和Attention is all you need

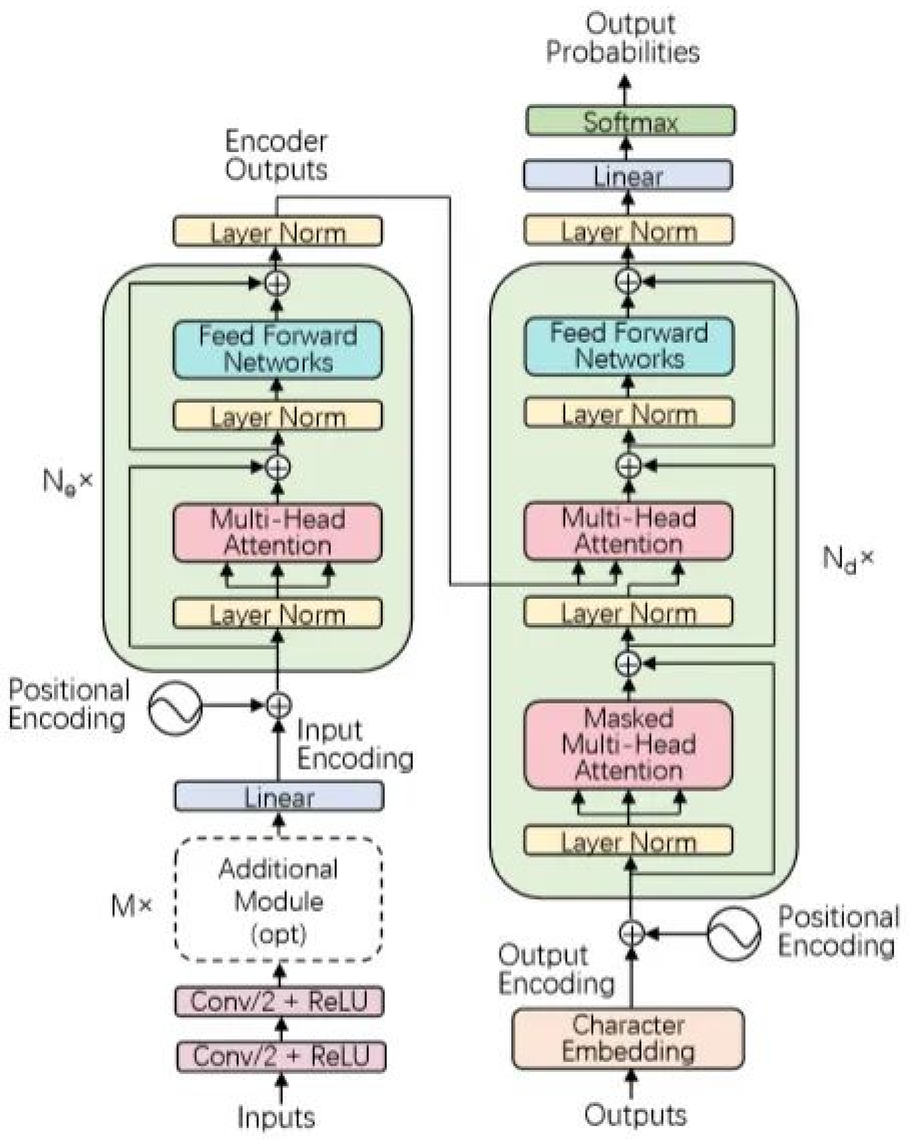


Fig. 2. Model architecture of the Speech-Transformer.

4. 基线系统的构建

4.1 实验环境

实验在Linux系统上进行，要求具备以下软件和硬件环境。

- 至少具备一个GPU
- python >= 3.6
- pytorch >= 1.2.0
- torchaudio >= 0.3.0

```
import torch
import torch.nn as nn
import torch.nn.functional as  F
```

4.2 数据处理与加载

4.2.1 词表生成

根据训练集文本生成词表，并加入起始标记<BOS>,结束标记<EOS>,填充标记<PAD>,以及未识别词标记<UNK>

4.2.2 构建特征提取与加载模块

```
# 词表生成
import os

vocab_dict = {}

for  name in ['train', 'dev']:
        with open(os.path.join('./data', name, 'text'), 'r', encoding='utf-8') as  fr:
                for  line  in fr:
                        chars = line.strip().split()[1:]
                        for  c  in chars:
                                if  c  in vocab_dict:
                                        vocab_dict[c] += 1
                                else:
                                        vocab_dict[c] = 1

vocab_list = sorted(vocab_dict.items(), key=lambda x: x[1], reverse=True)
vocab = {'<PAD>': 0, '<BOS>': 1, '<EOS>': 2, '<UNK>': 3}
for  i in range(len(vocab_list)):
        c  = vocab_list[i][0]
        vocab[c] = i + 4

print('There are %d units in Vocabulary!'  % len(vocab))
with open(os.path.join('./data', 'vocab'), 'w', encoding='utf-8') as  fw:
        for  c, id in vocab.items():
                fw.write(c+' '+ str(id) +'\n')

vocab_size = len(vocab) # 设置模型词表大小
```

4.2.2 构建特征提取与加载模块

```python
import os
import torch
import numpy as np
import torchaudio as ta
from torch.utils.data import Dataset, DataLoader

PAD = 0
BOS = 1
EOS = 2
UNK = 3

class  AudioDataset(Dataset):
    def  __init__(self, wav_list, text_list=None, unit2idx=None):

        self.unit2idx = unit2idx

        self.file_list = []
        for  wavscpfile in  wav_list:
            with open(wavscpfile, 'r', encoding='utf-8') as wr:
                for  line in  wr:
                    uttid, path = line.strip().split()
                    self.file_list.append([uttid, path])

        if  text_list is not  None:
            self.targets_dict = {}
            for  textfile in  text_list:
                with open(textfile, 'r', encoding='utf-8') as tr:
                    for  line in  tr:
                        parts = line.strip().split()
                        uttid = parts[0]
                        label = []
                        for  c in  parts[1:]:
                            label.append(self.unit2idx[c] if  c
                        self.targets_dict[uttid] = label
            self.file_list = self.filter(self.file_list) # 过滤掉没有标注的句子
            assert len(self.file_list) == len(self.targets_dict)
        else:
            self.targets_dict = None

        self.lengths = len(self.file_list)

    def  __getitem__(self, index):
        uttid, path = self.file_list[index]
        wavform, _  = ta.load_wav(path) # 加载wav文件
        feature = ta.compliance.kaldi.fbank(wavform, num_mel_bins=40) # 计算fbank
        # 特征归一化
        mean = torch.mean(feature)
        std = torch.std(feature)
        feature = (feature - mean) / std

        if  self.targets_dict is not  None:
                targets = self.targets_dict[uttid]
                return  uttid, feature, targets
        else:
                return  uttid, feature

    def  filter(self, feat_list):
        new_list = []
        for  (uttid, path) in  feat_list:
                if  uttid not  in  self.targets_dict:  continue
                new_list.append([uttid, path])
        return  new_list

    def  __len__(self):
        return  self.lengths

    @property
    def  idx2char(self):
        return  {i:  c for  (c, i) in  self.unit2idx.items()}


\# 收集函数，将同一个批内的特征填充到同样的长度，并在文本中加上起始标记和结束标记
def  collate_fn(batch):

    uttids = [data[0] for  data in  batch]
    features_length = [data[1].shape[0] for  data in  batch]
    max_feat_length = max(features_length)
    padded_features = []

    if  len(batch[0]) == 3:
        targets_length = [len(data[2]) for  data in  batch]
        max_text_length = max(targets_length)
        padded_targets = []

    for  parts in  batch:
        feat = parts[1]
        feat_len = feat.shape[0]
        padded_features.append(np.pad(feat, ((
                0, max_feat_length-feat_len), (0, 0)), mode='constant', constant_va
        if  len(batch[0]) == 3:
            target = parts[2]
            text_len = len(target)
            padded_targets.append(
                [BOS] + target + [EOS] + [PAD] * (max_text_length - text_le

    if  len(batch[0]) == 3:
        return  uttids, torch.FloatTensor(padded_features), torch.LongTensor(padde
    else:
        return  uttids, torch.FloatTensor(padded_features)
```

4.3 模型构建

模型的主体结构包含编码器和解码器。

4.3.1 基本模块构建

主要是模型的一些子模块，包括多头注意力机制，前馈模块以及位置编码模块。

（1）多头注意力机制模块

```
# 多头注意力机制

class  MultiHeadedAttention(nn.Module):

    """Multi-Head Attention layer

    :param int n_head: the number of head s
    :param int n_feat: the number of features
    :param float dropout_rate: dropout rate
    """

    def  __init__(self, n_head, n_feat, dropout_rate=0.0):
        super(MultiHeadedAttention, self).__init__()
        assert  n_feat % n_head == 0
        # We assume d_v always equals d_k
        self.d_k = n_feat // n_head
        self.h = n_head
        self.linear_q = nn.Linear(n_feat, n_feat)
        self.linear_k = nn.Linear(n_feat, n_feat)
        self.linear_v = nn.Linear(n_feat, n_feat)
        self.linear_out = nn.Linear(n_feat, n_feat)
        self.attn = None
        self.dropout = nn.Dropout(p=dropout_rate)

    def forward(self, query, key, value, mask):
        """Compute 'Scaled Dot Product Attention'

        :param torch.Tensor query: (batch, time1, size)
        :param torch.Tensor key: (batch, time2, size)
        :param torch.Tensor value: (batch, time2, size)
        :param torch.Tensor mask: (batch, time1, time2)
        :param torch.nn.Dropout dropout:
        :return torch.Tensor: attentined and transformed `value` (batch, time1, d_
                 weighted by the query dot key attention (batch, head, time1, tim
        """
        n_batch = query.size(0)
        q = self.linear_q(query).view(n_batch, -1, self.h, self.d_k)
        k = self.linear_k(key).view(n_batch, -1, self.h, self.d_k)
        v = self.linear_v(value).view(n_batch, -1, self.h, self.d_k)
        q = q.transpose(1, 2) # (batch, head, time1, d_k)
        k = k.transpose(1, 2) # (batch, head, time2, d_k)
        v = v.transpose(1, 2) # (batch, head, time2, d_k)

        scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(self.d_k) # (bat
        if mask is not None:
            mask = mask.unsqueeze(1).eq(0) # (batch, 1, time1, time2)
            min_value = float(np.finfo(torch.tensor(0, dtype=scores.dtype).num
            scores = scores.masked_fill(mask, min_value)
            self.attn = torch.softmax(scores, dim=-1).masked_fill(mask, 0.0) #
        else:
            self.attn = torch.softmax(scores, dim=-1) # (batch, head, time1, t

        p_attn = self.dropout(self.attn)
        x = torch.matmul(p_attn, v) # (batch, head, time1, d_k)
        x = x.transpose(1, 2).contiguous().view(n_batch, -1, self.h * self.d_k) #
        return self.linear_out(x) # (batch, time1, d_model)
```

（2）前馈模块

```
# 前馈模块
class PositionwiseFeedForward(nn.Module):
    """Positionwise feed forward

    :param int idim: input dimenstion
    :param int hidden_units: number of hidden units
```

```
        :param float dropout_rate: dropout rate
        """

        def  __init__(self, idim, hidden_units, dropout_rate=0.0):
                super(PositionwiseFeedForward, self).__init__()
                self.w_1 = nn.Linear(idim, hidden_units * 2)
                self.w_2 = nn.Linear(hidden_units, idim)
                self.dropout = nn.Dropout(dropout_rate)

        def  forward(self, x):
                x = self.w_1(x)
                x = F.glu(x)
                return  self.w_2(self.dropout(x))
```

（3） 位置编码模块

```
# 位置编码
class  PositionalEncoding(nn.Module):
        """Positional encoding."""

        def  __init__(self, d_model, dropout_rate=0.0, max_len=5000):
                """Initialize class.

                :param int d_model: embedding dim
                :param float dropout_rate: dropout rate
                :param int max_len: maximum input length

                """
                super(PositionalEncoding, self).__init__()
                self.d_model = d_model
                self.xscale = math.sqrt(self.d_model)
                self.dropout = nn.Dropout(p=dropout_rate)
                self.pe = None
                self.extend_pe(torch.tensor(0.0).expand(1, max_len))

        def  extend_pe(self, x):
                """Reset the positional encodings."""
                if  self.pe is  not  None:
                        if  self.pe.size(1) >= x.size(1):
                                if  self.pe.dtype != x.dtype or  self.pe.device != x.device
                                        self.pe = self.pe.to(dtype=x.dtype, device=x.device
                                return
                pe = torch.zeros(x.size(1), self.d_model)
                position = torch.arange(0, x.size(1), dtype=torch.float32).unsqueeze(1)
                div_term = torch.exp(torch.arange(0, self.d_model, 2, dtype=torch.float32)
                                                -(math.log(10000.0) / self.d_model
                pe[:, 0::2] = torch.sin(position * div_term)
                pe[:, 1::2] = torch.cos(position * div_term)
                pe = pe.unsqueeze(0)
                self.pe = pe.to(device=x.device, dtype=x.dtype)

        def  forward(self, x: torch.Tensor):
                """Add positional encoding.

                Args:
                        x (torch.Tensor): Input. Its shape is (batch, time, ...)

                Returns:
                        torch.Tensor: Encoded tensor. Its shape is (batch, time, ...)

                """
                self.extend_pe(x)
                x = x * self.xscale + self.pe[:, :x.size(1)]
                return  self.dropout(x)
```

4.3.2 编码器

```
# 编码器
# 对输入数据进行维度变换，时间维度降采样，以及进行位置编码，模块输出长度是输入长度的四分之一

class  Conv2dSubsampling(nn.Module):
    """Convolutional 2D subsampling (to 1/4 length)

    :param int idim: input dim
    :param int odim: output dim
    :param flaot dropout_rate: dropout rate
    """

    def  __init__(self, idim, odim, dropout_rate=0.0):
        super(Conv2dSubsampling, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(1, odim, 3, 2),
            nn.ReLU(),
            nn.Conv2d(odim, odim, 3, 2),
            nn.ReLU()
        )
        self.out = nn.Sequential(
            nn.Linear(odim * (((idim - 1) // 2  - 1) // 2), odim),
            PositionalEncoding(odim, dropout_rate)
        )

    def  forward(self, x, x_mask):
        """Subsample x

        :param torch.Tensor x: input tensor
        :param torch.Tensor x_mask: input mask
        :return: subsampled x and mask
        :rtype Tuple[torch.Tensor, torch.Tensor]
        """
        x = x.unsqueeze(1) # (b, c, t, f)
        x = self.conv(x)
        b, c, t, f = x.size()
        x = self.out(x.transpose(1, 2).contiguous().view(b, t, c * f))
        if  x_mask is  None:
            return  x, None
        return  x, x_mask[:, :, :-2:2][:, :, :-2:2]

\# 定义Transformer的每层的网络结构
class  TransformerEncoderLayer(nn.Module):
    def  __init__(self, attention_heads, d_model, linear_units, residual_dropout_rate)
        super(TransformerEncoderLayer, self).__init__()

        self.self_attn = MultiHeadedAttention(attention_heads, d_model)
        self.feed_forward = PositionwiseFeedForward(d_model, linear_units)

        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

        self.dropout1 = nn.Dropout(residual_dropout_rate)
        self.dropout2 = nn.Dropout(residual_dropout_rate)

    def  forward(self, x, mask):
        """Compute encoded features

        :param torch.Tensor x: encoded source features (batch, max_time_in, size)
        :param torch.Tensor mask: mask for x (batch, max_time_in)
        :rtype: Tuple[torch.Tensor, torch.Tensor]
        """
        residual = x
        x = residual + self.dropout1(self.self_attn(x, x, x, mask))
        x = self.norm1(x)

        residual = x
```

```
        x = residual + self.dropout2(self.feed_forward(x))
        x = self.norm2(x)

        return  x, mask

class  TransformerEncoder(nn.Module):

    def  __init__(self, input_size, d_model=256, attention_heads=4, linear_units=2048,
                 repeat_times=1, pos_dropout_rate=0.0, slf_attn_dropout_ra
                 residual_dropout_rate=0.1):
        super(TransformerEncoder, self).__init__()

        self.embed = Conv2dSubsampling(input_size, d_model)

        self.blocks = nn.ModuleList([
            TransformerEncoderLayer(attention_heads, d_model, linear_units, re
        ])

    def  forward(self, inputs):

        enc_mask = torch.sum(inputs, dim=-1).ne(0).unsqueeze(-2)
        enc_output, enc_mask = self.embed(inputs, enc_mask)

        enc_output.masked_fill_(~enc_mask.transpose(1, 2), 0.0)

        for  _, block in  enumerate(self.blocks):
            enc_output, enc_mask = block(enc_output, enc_mask)

        return  enc_output, enc_mask
```

4.3.3 解码器

```
# 定义解码层
class  TransformerDecoderLayer(nn.Module):

    def  __init__(self, attention_heads, d_model, linear_units, residual_dropout_rate)
        super(TransformerDecoderLayer, self).__init__()

        self.self_attn = MultiHeadedAttention(attention_heads, d_model)
        self.src_attn = MultiHeadedAttention(attention_heads, d_model)
        self.feed_forward = PositionwiseFeedForward(d_model, linear_units)

        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)

        self.dropout1 = nn.Dropout(residual_dropout_rate)
        self.dropout2 = nn.Dropout(residual_dropout_rate)
        self.dropout3 = nn.Dropout(residual_dropout_rate)

    def  forward(self, tgt, tgt_mask, memory, memory_mask):
        """Compute decoded features

        :param torch.Tensor tgt: decoded previous target features (batch, max_time
        :param torch.Tensor tgt_mask: mask for x (batch, max_time_out)
        :param torch.Tensor memory: encoded source features (batch, max_time_in, s
        :param torch.Tensor memory_mask: mask for memory (batch, max_time_in)
        """
        residual = tgt
        x = residual + self.dropout1(self.self_attn(tgt, tgt, tgt, tgt_mask))
        x = self.norm1(x)

        residual = x
        x = residual + self.dropout2(self.src_attn(x, memory, memory, memory_mask)
        x = self.norm2(x)

        residual = x
        x = residual + self.dropout3(self.feed_forward(x))
        x = self.norm3(x)

        return  x, tgt_mask

\# 遮掉未来的文本信息
def  get_seq_mask(targets):
    batch_size, steps = targets.size()
    seq_mask = torch.ones([batch_size, steps, steps], device=targets.device)
    seq_mask = torch.tril(seq_mask).bool()
    return  seq_mask

\# 定义解码器
class  TransformerDecoder(nn.Module):
    def  __init__(self, output_size, d_model=256, attention_heads=4, linear_units=2048
                            residual_dropout_rate=0.1, share_embedding=False):
        super(TransformerDecoder, self).__init__()

        self.embedding = torch.nn.Embedding(output_size, d_model)
        self.pos_encoding = PositionalEncoding(d_model)
        self.blocks = nn.ModuleList([
                TransformerDecoderLayer(attention_heads, d_model, linear_units,
                                                residual_dropout_rat
        ])

        self.output_layer = nn.Linear(d_model, output_size)

        if share_embedding:
                assert self.embedding.weight.size() == self.output_layer.weight.si
                self.output_layer.weight = self.embedding.weight

    def forward(self, targets, memory, memory_mask):

        dec_output = self.embedding(targets)
        dec_output = self.pos_encoding(dec_output)

        dec_mask = get_seq_mask(targets)

        for _, block in enumerate(self.blocks):
                dec_output, dec_mask = block(dec_output, dec_mask, memory, memory_r

        logits = self.output_layer(dec_output)

        return logits

    def recognize(self, preds, memory, memory_mask, last=True):

        dec_output = self.embedding(preds)
        dec_mask = get_seq_mask(preds)

        for _, block in enumerate(self.blocks):
                dec_output, dec_mask = block(dec_output, dec_mask, memory, memory_r

        logits = self.output_layer(dec_output)

        log_probs = F.log_softmax(logits[:, -1] if last else logits, dim=-1)

        return log_probs
```

4.3.4 整体结构

```
# 定义整体模型结构
class Transformer(nn.Module):
    def __init__(self, input_size, vocab_size, d_model=320, n_heads=4, d_ff=1280, num
        super(Transformer, self).__init__()

        self.vocab_size = vocab_size
        self.encoder = TransformerEncoder(input_size=input_size, d_model=d_model,
                                          attentior
                                          linear_ur
                                          num_block
                                          residual_

        self.decoder = TransformerDecoder(output_size=vocab_size,
                                          d_model=c
                                          attentior
                                          linear_ur
                                          num_block
                                          residual_
                                          share_emb

        self.crit = nn.CrossEntropyLoss()

    def forward(self, inputs, targets):

        # 1. forward encoder
        enc_states, enc_mask = self.encoder(inputs)

        # 2. forward decoder
        target_in = targets[:, :-1].clone()
        logits = self.decoder(target_in, enc_states, enc_mask)

        # 3. compute attention loss
        target_out = targets[:, 1:].clone()
        loss = self.crit(logits.reshape(-1, self.vocab_size), target_out.view(-1))

        return loss
```

4.4 训练过程与模型保存

```
import math

total_epochs = 60  # 模型迭代次数
model_size = 320  # 模型维度
n_heads = 4  # 注意力机制头数
num_enc_blocks = 6  # 编码器层数
```

```
num_dec_blocks = 6  # 解码器层数
residual_dropout_rate = 0.1  # 残差连接丢弃率
share_embedding = True  # 是否共享编码器词嵌入的权重

batch_size = 16  # 指定批大小
warmup_steps = 12000  # 热身步数
lr_factor = 1.0  # 学习率因子
accu_grads_steps = 8  # 梯度累计步数

input_size = 40  # 输入特征维度

unit2idx = {}
with open('./data/vocab', 'r', encoding='utf-8') as fr:
        for line in fr:
                unit, idx = line.strip().split()
                unit2idx[unit] = int(idx)
# 根据生成词表指定大小
vocab_size = len(unit2idx)
print('Set the size of vocab: %d' % vocab_size)


\# 模型定义
model = Transformer(input_size=input_size,
                                       vocab_size=vocab_size,
                                       d_model=model_size,
                                       n_heads=n_heads,
                                       d_ff=model_size * 4,
                                       num_enc_blocks=num_enc_blocks,
                                       num_dec_blocks=num_dec_blocks,
                                       residual_dropout_rate=residual_dropout_rate,
                                       share_embedding=share_embedding)

if torch.cuda.is_available():
        model.cuda() # 将模型加载到GPU中

\# 将模型加载
unit2idx = {}

train_wav_list = ['./data/train/wav.scp', './data/dev/wav.scp']
train_text_list = ['./data/train/text', './data/dev/text']
dataset = AudioDataset(train_wav_list, train_text_list, unit2idx=unit2idx)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                                                               shuffle=Tru
                                                                               collate_fn=
\# 定义优化器以及学习率更新函数
def  get_learning_rate(step):
        return  lr_factor * model_size ** (-0.5) * min(step ** (-0.5), step * warmup_steps

lr = get_learning_rate(step=1)
optimizer = torch.optim.Adam(model.parameters(), lr=lr, betas=(0.9, 0.98), eps=1e-9)

if  not  os.path.exists('./model'): os.makedirs('./model')

global_step = 1
step_loss = 0
print('Begin to Train...')
for  epoch in  range(total_epochs):
        print('***** epoch: %d *****'% epoch)
        for  step, (_, inputs, targets) in  enumerate(dataloader):
                # 将输入加载到GPU中
                if  torch.cuda.is_available():
                        inputs = inputs.cuda()
                        targets = targets.cuda()

                loss = model(inputs, targets)
                loss.backward()
                step_loss += loss.item()

                if  (step+1) % accu_grads_steps == 0:
                        # 梯度裁剪
                        grad_norm = torch.nn.utils.clip_grad_norm_(model.parameters(), 5.0)
                        optimizer.step()
                        optimizer.zero_grad()
                        if  global_step % 10  == 0:
                                print('-Training-Epoch-%d, Global Step:%d, lr:%.8f, Loss:%.
                                          (epoch, global_step, lr, step_loss / accu_grads_s
                        global_step += 1
                        step_loss = 0

                        # 学习率更新
                        lr = get_learning_rate(global_step)
                        for  param_group in  optimizer.param_groups:
                                param_group['lr'] = lr

        # 模型保存
        checkpoint = model.state_dict()
        torch.save(checkpoint, os.path.join('./model', 'model.epoch.%d.pt'  % epoch))
print('Done!')
```

4.5 解码

解码阶段使用beam search来进行解码操作。

```
\# 定义解码识别模块
class  Recognizer():
      def  __init__(self, model, unit2char=None, beam_width=5, max_len=100):

            self.model = model
            self.model.eval()
            self.unit2char = unit2char
            self.beam_width = beam_width
            self.max_len = max_len

      def  recognize(self, inputs):

            enc_states, enc_masks = self.model.encoder(inputs)

            # 将编码状态重复beam_width次
            beam_enc_states = enc_states.repeat([self.beam_width, 1, 1])
            beam_enc_mask = enc_masks.repeat([self.beam_width, 1, 1])

            # 设置初始预测标记 <BOS>, 维度为[beam_width, 1]
            preds = torch.ones([self.beam_width, 1], dtype=torch.long, device=enc_stat

            # 定义每个分支的分数, 维度为[beam_width, 1]
            global_scores = torch.FloatTensor([0.0] + [-float('inf')] * (self.beam_wid
            global_scores = global_scores.to(enc_states.device).unsqueeze(1)

            # 定义结束标记，任意分支出现停止标记1则解码结束， 维度为 [beam_width, 1]
            stop_or_not = torch.zeros_like(global_scores, dtype=torch.bool)

            def  decode_step(pred_hist, scores, flag):
                  """ decode an utterance in a stepwise way"""
                  batch_log_probs = self.model.decoder.recognize(pred_hist, beam_enc_
                  last_k_scores, last_k_preds = batch_log_probs.topk(self.beam_width)
                  # 分数更新
                  scores = scores + last_k_scores
                  scores = scores.view(self.beam_width * self.beam_width)
                  # 保存所有路径中的前k个路径
                  scores, best_k_indices = torch.topk(scores, k=self.beam_width)
                  scores = scores.view(-1, 1)
                  # 更新预测
                  pred = pred_hist.repeat([self.beam_width, 1])
                  pred = torch.cat((pred, last_k_preds.view(-1, 1)), dim=1)
                  best_k_preds = torch.index_select(pred, dim=0, index=best_k_indice
                  # 判断最后一个是不是结束标记
                  flag = torch.eq(best_k_preds[:, -1], EOS).view(-1, 1)
                  return  best_k_preds, scores, flag

            with torch.no_grad():

                  for  _  in  range(1, self.max_len+1):
                      preds, global_scores, stop_or_not = decode_step(preds, glob
                      # 判断是否停止，任意分支解码到结束标记或者达到最大解码步数则
                      if  stop_or_not.sum() > 0: break

                  max_indices = torch.argmax(global_scores, dim=-1).long()
                  preds = preds.view(self.beam_width, -1)
                  best_preds = torch.index_select(preds, dim=0, index=max_indices)

                  # 删除起始标记 BOS
                  best_preds = best_preds[0, 1:]
                  results = []
                  for  i in  best_preds:
                      if  int(i) == EOS:
                          break
                      results.append(self.unit2char[int(i)])
            return  ''.join(results)


\# 定义评估模型
eval_model = Transformer(input_size=input_size,
                                                vocab_size=vocab_size,
                                                d_model=model_size,
                                                n_heads=n_heads,
                                                d_ff=model_size * 4,
                                                num_enc_blocks=num_enc_blocks,
                                                num_dec_blocks=num_dec_blocks,
                                                residual_dropout_rate=0.0,
                                                share_embedding=share_embedding)


if  torch.cuda.is_available():
      eval_model.cuda() # 将模型加载到GPU中

\# 将模型加载
idx2unit = {}
```

```
with open('./data/vocab', 'r', encoding='utf-8') as fr:
        for  line in  fr:
                unit, idx = line.strip().split()
                idx2unit[int(idx)] = unit

wav_list = ['./data/test/wav.scp']
test_dataset = AudioDataset(wav_list)
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=1,
                                                                                                   s
                                                                                                   c

\# checkpoints = torch.load('./model/model.pt', map_location=lambda storage, loc: storage,
checkpoints = torch.load('./model/model.epoch.59.pt')
model.load_state_dict(checkpoints)

recognizer = Recognizer(model, unit2char=idx2unit)

csv_writer = open('result.csv', 'w', encoding='utf-8')
csv_writer.write('id,words\n')
print('Begin to decode test set!')
total_num = len(test_dataloader)
for  step, (uttid, inputs) in  enumerate(test_dataloader):
        # 将输入加载到GPU中
        if  torch.cuda.is_available():
                inputs = inputs.cuda()
        preds = recognizer.recognize(inputs)
        print('[%5d/%d] %s %s'  % (step, total_num, uttid[0], preds)) # 打印输出结果
        csv_writer.write(','.join([uttid[0], preds])+'\n')
csv_writer.close()
print('Done!')
```

5. 提升方向

- 代码参考 OpenTransformer 和 ESPnet
- 多种数据扩增（速度扰动，谱增广），以提升模型的鲁棒性，建议参考 SpecAugment
- 标签平滑（Lable Smoothing）
- 混合语言模型解码（shallow fusion, deep fusion）
- 迁移学习

比赛时间

竞赛分为初赛与复赛两阶段，初赛已于2019年12月23日开启，biendata 平台同步发布训练集、开发集、测试集，并开放初赛提交。2020年3月20日，初赛报名和组队时间截止。每日提交存在次数限制，请感兴趣的选手尽量选择提前参赛，以获得更多验证提交次数和优化模型的机会。

比赛数据

“智源 MagicSpeechNet 家庭场景中文语音数据集”是当前业界稀缺的优质家居环境语音数据，其中包含数百小时的真实家庭环境中的双人对话，每段对话基于多种平台进行录制，并已完全转录和标注。比赛数据分为训练集、开发集和测试集三部分，测试集数据为需要识别的音频文件，每段音频分为安卓平台、iOS 平台，录音笔录制的三个文件。为便于选手分割每段音频，比赛提供了标明起始和结束时间点信息的 json 文件，选手需使用模型识别音频中的对话，并根据 json 中对应的 uttid 提交相应的文本。 相较于国内外同类多通道语音识别比赛，本比赛数据在数量、场景、声音特性等方面具有以下优势。

（1）大量的对话数据国内的语音识别比赛基本使用朗读类型的语音数据，而本比赛使用的数据为真实的对话数据。数据为完全真实场景的对话，说话人以放松和无脚本的方式，围绕所选主题自由对话。相比基于对话数据的国际同类比赛，在数据量方面仍旧具有极大的优势。同时，合理的说话人语音交叠更真实地体现日常家庭场景下的语音识别难度。
（2）场景真实多样本数据集采集于3个真实的家庭场景，说话人以放松和无脚本的方式，围绕所选主题自由对话。不同的采集环境丰富了数据的多样性，同时增强了比赛的难度。
（3）近讲与多平台远讲数据结合每段对话有 5 个通道的同步录音，包括 3 个远讲通道和2 个近讲通道。远讲通道分别由多个型号的安卓手机，苹果手机和录音笔录制，充分体现多平台录音数据的特性；近讲数据使用高保真麦克风录制，与说话人的嘴保持10 cm 的距离。
（4）丰富均衡的声音特性本数据集拥有丰富均衡的声音特性。录制本数据集的说话人来自中国大陆不同地域，存在一定的普通话口音。同时，说话人选自不同年龄段，性别均衡。

参赛方式

点击阅读原文链接或扫描下图中的二维码直达赛事页面，注册网站-下载数据，即可参赛。

*友情提示，因涉及到数据下载，强烈建议大家登录 PC 页面报名参加。 *

智源算法大赛

2019 年 9 月，智源人工智能算法大赛正式启动。本次比赛由北京智源人工智能研究院主办，清华大学、北京大学、中科院计算所、旷视、知乎、博世、爱数智慧、国家天文台、晶泰等协办，总奖金超过 100 万元，旨在以全球领先的科研数据集与算法竞赛为平台，选拔培育人工智能创新人才。

研究院副院长刘江也表示："我们希望不拘一格来支持人工智能真正的标志性突破，即使是本科生，如果真的是好苗子，我们也一定支持。"而人工智能大赛就是发现有潜力的年轻学者的重要途径。

本次智源人工智能算法大赛有两个重要的目的，一是通过发布数据集和数据竞赛的方式，推动基础研究的进展。特别是可以让计算机领域的学者参与到其它学科的基础科学研究中。二是可以通过比赛筛选、锻炼相关领域的人才。 智源算法大赛已发布全部的10个数据集，目前仍有5个比赛（奖金50万）尚未结束。

正在角逐的比赛

智源小分子化合物性质预测挑战赛
https://www.biendata.com/competition/molecule/
智源杯天文数据算法挑战赛
https://www.biendata.com/competition/astrodata2019/
智源-INSPEC 工业大数据质量预测赛
https://www.biendata.com/competition/bosch/
智源-MagicSpeechNet 家庭场景中文语音数据集挑战赛

https://www.biendata.com/competition/magicdata/
智源-高能对撞粒子分类挑战赛
https://www.biendata.com/competition/jet/

阅读原文