基于CNN的语音活动检测（VAD）算法报告

算法设计

模型结构

我们的CNN模型包括以下层次：

1. **卷积层（Conv2D）**：负责提取局部特征。
2. **批归一化层（Batch Normalization）**：用于加速训练并提高模型的稳定性。
3. **Leaky ReLU激活函数**：引入非线性。
4. **全连接层（Fully Connected Layer）**：用于最终的分类。

模型的结构图如下：

```
rust
复制代码
Input -> Conv2D -> BatchNorm -> Leaky ReLU -> Fully Connected Layer -> Output
```

数据处理

在训练模型之前，需要对音频数据进行预处理，包括读取音频文件、降采样、生成标签和数据增强等。

模型训练

模型训练包括以下步骤：

1. 加载数据集。
2. 划分训练集和验证集。
3. 定义损失函数和优化器。
4. 训练模型并在每个epoch结束时进行验证。

推理

推理过程包括：

1. 加载训练好的模型。
2. 对测试音频数据进行处理并进行预测。
3. 计算语音段并将结果保存到文件中。

Python实现

模型定义

```
python复制代码import torch
import torch.nn as nn

class CNN(nn.Module):
    def __init__(self, input_channel=1, n_channel=2, kernel_size=2, stride=2, dilation=1, padding="valid") -> None:
        super(CNN, self).__init__()
        self.fc_size = 120 * 2
```

```
        model = nn.Sequential(
            nn.Conv2d(in_channels=input_channel, out_channels=n_channel,
kernel_size=(1, kernel_size),
                      stride=stride, dilation=dilation, padding=padding,
bias=False),
            nn.BatchNorm2d(num_features=n_channel),
            nn.LeakyReLU(inplace=True),
        )
        self.model = model
        self.output = nn.Linear(in_features=self.fc_size, out_features=2)

    def forward(self, x):
        x = self.model(x)
        x = x.view(x.size(0), -1)
        output = self.output(x)
        return output
```

数据处理与训练

```
python复制代码import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split
from pathlib import Path
import numpy as np
from util import read_wav, read_txt, sample_rate_to_8K
from model import CNN
import random
from sklearn.metrics import confusion_matrix, precision_score, recall_score

class VADDataset(Dataset):
    def __init__(self, data_dir, label_dir, frame_len, sample_rate,
augment=False):
        self.data_dir = Path(data_dir)
        self.label_dir = Path(label_dir)
        self.frame_len = frame_len
        self.sample_rate = sample_rate
        self.data_files = sorted(self.data_dir.glob("*.wav"))
        self.label_files = sorted(self.label_dir.glob("*.txt"))
        self.augment = augment
        self.data, self.labels = self.process_data()

    def process_data(self):
        data = []
        labels = []
        for data_file, label_file in zip(self.data_files, self.label_files):
            signal, signal_len, sample_rate = read_wav(str(data_file))
            signal, signal_len = sample_rate_to_8K(signal, sample_rate)
            label_data = read_txt(label_file)
            for start, end in label_data:
                for i in range(start, end, int(FS * FRAME_STEP)):
                    if i + self.frame_len > signal_len:
                        break
                    frame_data = signal[i:i + self.frame_len]
                    if self.augment:
```

```
                        frame_data = self.add_noise(frame_data)
                    label = 1 if (i >= start and i <= end) else 0
                    data.append(frame_data)
                    labels.append(label)
            non_voice_intervals = self.get_non_voice_intervals(signal_len,
label_data)
            for start, end in non_voice_intervals:
                for i in range(start, end, int(FS * FRAME_STEP)):
                    if i + self.frame_len > signal_len:
                        break
                    frame_data = signal[i:i + self.frame_len]
                    if self.augment:
                        frame_data = self.add_noise(frame_data)
                    data.append(frame_data)
                    labels.append(0)
        return np.array(data), np.array(labels)

    def get_non_voice_intervals(self, signal_len, label_data):
        non_voice_intervals = []
        previous_end = 0
        for start, end in label_data:
            if start > previous_end:
                non_voice_intervals.append((previous_end, start))
            previous_end = end
        if (previous_end < signal_len):
            non_voice_intervals.append((previous_end, signal_len))
        return non_voice_intervals

    def add_noise(self, data):
        noise = np.random.randn(len(data)) * 0.005
        augmented_data = data + noise
        return augmented_data

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        sample =
torch.from_numpy(self.data[idx]).float().unsqueeze(0).unsqueeze(0)
        label = torch.tensor(self.labels[idx]).long()
        return sample, label

def train_vad(model, train_loader, val_loader, criterion, optimizer,
num_epochs=20):
    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        for inputs, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
```

```
        print(f'Epoch {epoch+1}/{num_epochs}, Loss:
{running_loss/len(train_loader)}')

        model.eval()
        val_loss = 0.0
        all_labels = []
        all_preds = []
        with torch.no_grad():
            for inputs, labels in val_loader:
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                all_labels.extend(labels.cpu().numpy())
                all_preds.extend(predicted.cpu().numpy())

        print(f'Validation Loss: {val_loss/len(val_loader)}')
        print(f'Accuracy: {100 * (np.array(all_labels) ==
np.array(all_preds)).sum() / len(all_labels)}%')
        print(f'Precision: {precision_score(all_labels, all_preds,
average="binary", zero_division=0)}')
        print(f'Recall: {recall_score(all_labels, all_preds, average="binary",
zero_division=0)}')
        print('Confusion Matrix:')
        print(confusion_matrix(all_labels, all_preds, labels=[0, 1]))

if __name__ == "__main__":
    FS = 8000
    FRAME_T = 0.03
    FRAME_STEP = 0.015
    frame_len = int(FRAME_T * FS)

    data_dir = "./data"
    label_dir = "./label"
    augment = True
    dataset = VADDataset(data_dir, label_dir, frame_len, FS, augment=augment)

    train_size = int(0.8 * len(dataset))
    val_size = len(dataset) - train_size
    train_dataset, val_dataset = random_split(dataset, [train_size, val_size])
    train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)

    model = CNN()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    train_vad(model, train_loader, val_loader, criterion, optimizer,
num_epochs=20)

    torch.save(model.state_dict(), "./model_trained.pth")
```

推理

```
python复制代码import matplotlib.pyplot as plt
import numpy as np
from pathlib import Path
from util import *
from model import CNN  # 确保导入模型类
import torch
from torch.utils.data import Dataset, DataLoader

class VADDataset(Dataset):
    def __init__(self, data_files, frame_len, sample_rate):
        self.data_files = data_files
        self.frame_len = frame_len
        self.sample_rate = sample_rate
        self.data, self.file_indices = self.process_data()

    def process_data(self):
        data = []
        file_indices = []
        for file_idx, data_file in enumerate(self.data_files):
            signal, signal_len, sample_rate = read_wav(str(data_file))
            signal, signal_len = sample_rate_to_8K(signal, sample_rate)

            for i in range(0, signal_len, int(FRAME_STEP * FS)):
                if i + self.frame_len > signal_len:
                    break
                frame_data = signal[i:i + self.frame_len]
                data.append(frame_data)
                file_indices.append((file_idx, i))

        return np.array(data), file_indices

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        sample =
torch.from_numpy(self.data[idx]).float().unsqueeze(0).unsqueeze(0)
        file_idx, frame_idx = self.file_indices[idx]
        return sample, file_idx, frame_idx

def cal_voice_segment(pred_class, pred_idx_in_data, raw_data_len):
    if len(pred_class) != len(pred_idx_in_data):
        raise Exception("pred_class length must be pred_idx_in_data length!")

    all_voice_segment = np.array([])
    single_voice_segment = []
    diff_value = np.diff(pred_class)

    for i in range(len(diff_value)):
        if diff_value[i] == 1:
            single_voice_segment.append(pred_idx_in_data[i+1])
        if diff_value[i] == -1:
            if len(single_voice_segment) == 0:
```

```
                single_voice_segment.append(0)
            single_voice_segment.append(pred_idx_in_data[i+1])
        if len(single_voice_segment) == 2:
            if len(all_voice_segment) == 0:
                all_voice_segment = np.array(single_voice_segment).reshape(1, -1)
            else:
                all_voice_segment = np.concatenate((all_voice_segment,
np.array(single_voice_segment).reshape(1, -1)), axis=0)
            single_voice_segment = []

    if len(single_voice_segment) == 1:
        single_voice_segment.append(raw_data_len - 1)
        all_voice_segment = np.concatenate((all_voice_segment,
np.array(single_voice_segment).reshape(1, -1)), axis=0)

    if all_voice_segment.size == 0 and np.all(pred_class == 1):
        all_voice_segment = np.array([[0, raw_data_len - 1]])

    return all_voice_segment

def vad_inference(data_dir: str, model_path: str, result_dir: str):
    data_files = sorted(Path(data_dir).glob("*.wav"))
    dataset = VADDataset(data_files, frame_len, FS)
    dataloader = DataLoader(dataset, batch_size=32, shuffle=False)

    model = CNN()
    model.load_state_dict(torch.load(model_path))
    model.eval()

    results = [[] for _ in range(len(data_files))]
    with torch.no_grad():
        for inputs, file_indices, frame_indices in dataloader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)

            for file_idx, frame_idx, pred in zip(file_indices, frame_indices,
predicted):
                results[file_idx].append((frame_idx.item(), pred.item()))

    for file_idx, file_dir in enumerate(data_files):
        signal, signal_len, sample_rate = read_wav(str(file_dir))
        signal, signal_len = sample_rate_to_8K(signal, sample_rate)
        file_results = results[file_idx]
        frame_indices, preds = zip(*file_results)
        voice_segment = cal_voice_segment(np.array(preds),
np.array(frame_indices), signal_len)

        plt.figure(1, figsize=(15, 7))
        plt.clf()
        draw_time_domain_image(signal, nframes=signal_len, framerate=sample_rate,
line_style="b-")
        draw_result(signal, voice_segment)
        plt.grid()
        plt.show()

        result_dir_path = Path(result_dir)
```

```
        result_dir_path.mkdir(parents=True, exist_ok=True)
        result_file = result_dir_path / f"{file_dir.stem}_result.txt"
        np.savetxt(result_file, voice_segment, fmt="%d", delimiter=",")
```

模型参数提取

为了在C语言中实现模型，需要从Python中提取模型参数。

模型参数提取代码

```
python复制代码import torch

# 加载模型
model_path = "./model/model_microphone.pth"
model = CNN()
model.load_state_dict(torch.load(model_path))
model.eval()

# 提取卷积层参数
conv1_weight = model.model[0].weight.data.numpy().flatten()
conv1_weight_c = ", ".join(map(str, conv1_weight))

# 提取BN层参数
bn1_weight = model.model[1].weight.data.numpy().flatten()
bn1_bias = model.model[1].bias.data.numpy().flatten()
bn1_running_mean = model.model[1].running_mean.data.numpy().flatten()
bn1_running_var = model.model[1].running_var.data.numpy().flatten()

bn1_weight_c = ", ".join(map(str, bn1_weight))
bn1_bias_c = ", ".join(map(str, bn1_bias))
bn1_running_mean_c = ", ".join(map(str, bn1_running_mean))
bn1_running_var_c = ", ".join(map(str, bn1_running_var))

# 提取全连接层参数
fc_weight = model.output.weight.data.numpy().flatten()
fc_bias = model.output.bias.data.numpy().flatten()

fc_weight_c = ", ".join(map(str, fc_weight))
fc_bias_c = ", ".join(map(str, fc_bias))

# 生成C语言头文件内容
c_header = f"""
#ifndef __MODEL_PARAMETERS_H__
#define __MODEL_PARAMETERS_H__

double model_0_weight[] = {{{conv1_weight_c}}};
double model_1_weight[] = {{{bn1_weight_c}}};
double model_1_bias[] = {{{bn1_bias_c}}};
double model_1_running_mean[] = {{{bn1_running_mean_c}}};
double model_1_running_var[] = {{{bn1_running_var_c}}};
double output_weight[] = {{{fc_weight_c}}};
double output_bias[] = {{{fc_bias_c}}};

#endif
"""
```

```python
# 将内容写入C语言头文件
with open("model_parameters.h", "w") as f:
    f.write(c_header)
```

结果

```python
import numpy as np
import librosa

SAMPLE_RATE = 8000
WAV = 'data/zzh_10.wav'
LABEL_INPUT = 'label/zzh_10.txt'
PREDICT_INPUT = 'result/zzh_10_result.txt'

def evaluate(data_length, label_input, predict_input):

    voice_length = 0
    predict_voice_length = 0

    label = np.full(data_length, 0)
    label_data = np.loadtxt(label_input,delimiter=',')
    for i in range(len(label_data)):
        a = int(label_data[i][0])
        b = int(label_data[i][1])
        label[a:b+1] = 1
        voice_length += (b - a)

    predict = np.full(data_length, 0)
    predict_data = np.loadtxt(predict_input,delimiter=',')
    for i in range(len(predict_data)):
        a = int(predict_data[i][0])
        b = int(predict_data[i][1])
        predict[a:b+1] = 1
        predict_voice_length += (b - a)

    false_detection = 0
    miss_detection = 0
    acc = 0
    tp = 0
    for i in range(data_length):
        if label[i] == 0 and predict[i] == 1:
            false_detection += 1
        if label[i] == 1 and predict[i] == 0:
            miss_detection += 1
        if label[i] == predict[i]:
            acc += 1
        if label[i] == 1 and predict[i] == 1:
            tp += 1

    accuracy = acc/data_length
    recall = tp/voice_length
    precision = tp/predict_voice_length
    f1_score = (2*precision*recall)/(precision+recall)
```

```
    return f1_score,accuracy,recall,precision

if __name__ == '__main__':

    wav_input,sample_rate = librosa.load(WAV,sr=SAMPLE_RATE)
    data_length = len(wav_input)
    label_input = LABEL_INPUT
    predict_input = PREDICT_INPUT

    f1_score,accuracy,recall,precision = evaluate(data_length, label_input,
predict_input)
    print('\n')
    print('f1_score: ',f1_score)
    print('accuracy: ',accuracy)
    print('recall: ',recall)
    print('precision: ',precision)
    print('\n')
```

```
f1_score:  0.8214898678893456
accuracy:  0.7863997574268342
recall:  0.6979719110095058
precision:  0.998125
```

C语言实现

卷积、BN和激活函数的实现

`conv.h`

```
c复制代码#ifndef __CONV_H__
#define __CONV_H__

#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include "algo_error_code.h"

/**
 * Input and output data of the convolutional layer
 */
typedef struct _Conv2dData {
    uint16_t row;
    uint16_t col;
    uint16_t channel;
    double *data;
} Conv2dData;

/**
 * convolutional layer weights
 */
```

```c
typedef struct _Conv2dFilter {
    uint16_t row;
    uint16_t col;
    uint16_t channel;
    uint16_t filter_num;
    double *data;
} Conv2dFilter;

/**
 * Batch Normalization
 */
typedef struct _BatchNorm2d {
    uint16_t size;
    double *mean;
    double *var;
    double *gamma;
    double *beta;
} BatchNorm2d;

/**
 * configuration of convolutional layers, including convolution weights and BN
 */
typedef struct _Conv2dConfig {
    uint16_t stride;
    uint16_t pad;
    Conv2dFilter *filter;
    BatchNorm2d *bn;
} Conv2dConfig;

/**
 * configuration of the linear layer, including weights and biases
 */
typedef struct _LinearConfig {
    uint16_t inp_size;
    uint16_t fea_size;
    double *weight;
    double *bias;
} LinearParam;

/**
 * @brief conv2d with BN layer without bias
 *
 * @param[in] input_feat: input feature map
 * @param[in] param: configuration of the convolutional layer
 * @param[out] output_feat: output feature map
 * @return error code
 */
int conv2d_bn_no_bias(Conv2dData *input_feat, Conv2dConfig *param, Conv2dData
*output_feat);

/**
 * @brief leak_relu activation function
 *
 * @param[in] neg_slope: controls the angle of the negative slope
 * @param[in] inp: input data
 * @param[in] inp_size: input data size
```

```
 * @param[out] out: output data
 * @return error code
 */
int leaky_relu(double neg_slope, double *inp, uint16_t inp_size, double *out);

/**
 * @brief linear layer
 *
 * @param[in] inp: input data
 * @param[in] linear_config: configuration of the linear layer
 * @param[out] out: output data
 * @return error code
 */
int linear_layer(double *inp, LinearParam *linear_config, double *out);

/**
 * @brief calculate the length of different dimensions of the convolutional layer
output feature map
 *
 * @param[in] raw_len: the length of input feature map
 * @param[in] pad_len: the length of padding
 * @param[in] filter_len: kernel size
 * @param[in] stride: the stride length of conv
 * @return the length of output feature map
 */
uint16_t cal_conv_out_len(uint16_t raw_len, uint16_t pad_len, uint16_t
filter_len, uint16_t stride);

#endif
```

`conv.c`

```
c复制代码#include "conv.h"

#define BN_EPS (1e-5)

static void padding_value(const Conv2dData *raw_data, uint16_t pad_len, double
pad_value, double *paded_data)
{
    uint16_t row = raw_data->row, col = raw_data->col, chan = raw_data->channel;
    uint16_t paded_data_size = 0;
    uint16_t i = 0, j = 0, k = 0;
    uint16_t pad_idx = 0;

    paded_data_size = (row + 2 * pad_len) * (col + 2 * pad_len) * chan;

    for (i = 0; i < paded_data_size; i++) {
        paded_data[i] = pad_value;
    }

    for (i = 0; i < raw_data->channel; i++) {
        for (j = 0; j < raw_data->row; j++) {
            for (k = 0; k < raw_data->col; k++) {
                pad_idx = k + pad_len + (j + pad_len) * (col + 2 * pad_len) + i *
(row + 2 * pad_len) * (col + 2 * pad_len);
```

```
                paded_data[pad_idx] = raw_data->data[k + j * col + i * row *
col];
            }
        }
    }
}

uint16_t cal_conv_out_len(uint16_t raw_len, uint16_t pad_len, uint16_t
filter_len, uint16_t stride)
{
    return (raw_len + 2 * pad_len - filter_len) / stride + 1;
}

int conv2d_bn_no_bias(Conv2dData *input_feat, Conv2dConfig *param, Conv2dData
*output_feat)
{
    uint16_t i = 0, j = 0, k = 0, ii = 0, jj = 0, kk = 0, group_cnt = 0;
    uint16_t out_row = 0, out_col = 0, out_chan = 0;
    uint16_t paded_row = 0, paded_col = 0, paded_feat_size = 0;
    uint16_t row_start = 0, col_start = 0, filter_idx = 0, feat_idx = 0,
output_feat_idx = 0;
    BatchNorm2d *bn      = NULL;
    Conv2dFilter *filter = NULL;

    double tmp         = 0.0;
    double *paded_feat = NULL;

    if (!input_feat || !input_feat->data || !param || !param->bn || !param->bn-
>mean ||
        !param->bn->var || !param->bn->gamma || !param->bn->beta || !param-
>filter ||
        !param->filter->data || !output_feat || !output_feat->data) {
        return ALGO_POINTER_NULL;
    }

    bn     = param->bn;
    filter = param->filter;

    if (param->stride < 1 || input_feat->channel != filter->channel ||
        filter->filter_num != bn->size || filter->row > 2 * param->pad +
input_feat->row ||
        filter->col > 2 * param->pad + input_feat->col) {
        return ALGO_DATA_EXCEPTION;
    }

    if (input_feat->row == 1) {
        out_row = 1;
    } else {
        out_row = cal_conv_out_len(input_feat->row, param->pad, filter->row,
param->stride);
    }

    out_col  = cal_conv_out_len(input_feat->col, param->pad, filter->col, param-
>stride);
    out_chan = filter->filter_num;
```

```
    // padding 0
    paded_row  = input_feat->row + 2 * param->pad;
    paded_col  = input_feat->col + 2 * param->pad;
    paded_feat = input_feat->data;
    if (param->pad != 0) {
        paded_feat_size = paded_row * paded_col * input_feat->channel;
        paded_feat      = (double *)malloc(sizeof(double) * paded_feat_size);
        if (!paded_feat) {
            return ALGO_MALLOC_FAIL;
        }
        memset((void *)paded_feat, 0, sizeof(double) * paded_feat_size);
        padding_value(input_feat, param->pad, 0.0, paded_feat);
    }

    // conv calculate
    for (i = 0; i < out_chan; i++) {
        for (j = 0; j < out_row; j++) {
            for (k = 0; k < out_col; k++) {
                tmp       = 0.0;
                row_start = j * param->stride;
                col_start = k * param->stride;
                for (ii = 0; ii < filter->channel; ii++) {
                    for (jj = 0; jj < filter->row; jj++) {
                        for (kk = 0; kk < filter->col; kk++) {
                            filter_idx = kk + jj * filter->col + ii * filter->row
* filter->col +
                                         i * filter->row * filter->col * filter-
>channel;
                            feat_idx = col_start + kk + (row_start + jj) *
paded_col +
                                       ii * paded_row * paded_col;
                            tmp += filter->data[filter_idx] *
paded_feat[feat_idx];
                        }
                    }
                }

                tmp = bn->gamma[i] * (tmp - bn->mean[i]) / sqrt(bn->var[i] +
BN_EPS) + bn->beta[i];

                output_feat_idx                    = k + j * out_col + i *
out_row * out_col;
                output_feat->data[output_feat_idx] = tmp;
            }
        }
    }

    output_feat->row     = out_row;
    output_feat->col     = out_col;
    output_feat->channel = out_chan;

    if (param->pad != 0) {
        free(paded_feat);
    }

    return ALGO_NORMAL;
```

```
}

int leaky_relu(double neg_slope, double *inp, uint16_t inp_size, double *out)
{
    uint16_t i = 0;

    if (!inp || !out) {
        return ALGO_POINTER_NULL;
    }

    for (i = 0; i < inp_size; i++) {
        out[i] = inp[i];

        if (inp[i] < 0) {
            out[i] = neg_slope * inp[i];
        }
    }

    return ALGO_NORMAL;
}

int linear_layer(double *inp, LinearParam *linear_config, double *out)
{
    uint16_t i, j;

    if (!inp || !linear_config || !linear_config->weight || !linear_config->bias
|| !out) {
        return ALGO_POINTER_NULL;
    }

    for (i = 0; i < linear_config->fea_size; i++) {
        out[i] = linear_config->bias[i];
        for (j = i * linear_config->inp_size; j < (i + 1) * linear_config-
>inp_size; j++) {
            out[i] += inp[j - i * linear_config->inp_size] * linear_config-
>weight[j];
        }
    }

    return ALGO_NORMAL;
}
```

VAD预测函数实现

`vad.h`

```
c复制代码#ifndef __VAD_H__
#define __VAD_H__

#include <stdint.h>
#include <stdbool.h>
#include <stdlib.h>

#include "conv.h"
#include "algo_error_code.h"
```

```
/**
 * @brief voice detection function
 *
 * @param[in] inp_data: raw audio data
 * @param[out] is_voice: the result of voice detection
 * @return error code
 */
int vad(Conv2dData *inp_data, bool *is_voice);

#endif
```

`vad.c`

```
c复制代码#include "vad.h"
#include "model_parameters.h"

int vad(Conv2dData *inp_data, bool *is_voice)
{
    int ret               = ALGO_NORMAL;
    uint16_t conv_out_len = 0;
    double linear_out[2]  = {0};

    Conv2dFilter filter = {
        .channel = 1, .col = 2, .row = 1, .filter_num = 2, .data =
model_0_weight};
    BatchNorm2d bn            = {.beta  = model_1_bias,
                                 .gamma = model_1_weight,
                                 .mean  = model_1_running_mean,
                                 .var   = model_1_running_var,
                                 .size  = 2};
    Conv2dConfig conv_config  = {.pad = 0, .stride = 2, .bn = &bn, .filter =
&filter};
    LinearParam linear_config = {
        .inp_size = 240, .fea_size = 2, .weight = output_weight, .bias =
output_bias};

    Conv2dData conv_out;

    *is_voice = false;

    memset(&conv_out, 0, sizeof(Conv2dData));
    conv_out_len  = cal_conv_out_len(inp_data->col, 0, 2, 2);
    conv_out.data = (double *)malloc(sizeof(double) * conv_out_len * 2);
    if (!conv_out.data) {
        return ALGO_MALLOC_FAIL;
    }

    ret = conv2d_bn_no_bias(inp_data, &conv_config, &conv_out);
    if (ret != ALGO_NORMAL) {
        goto func_exit;
    }

    ret = leaky_relu(0.01, conv_out.data, conv_out.channel * conv_out.col *
conv_out.row, conv_out.data);
```

```
    if (ret != ALGO_NORMAL) {
        goto func_exit;
    }

    ret = linear_layer(conv_out.data, &linear_config, linear_out);
    if (ret != ALGO_NORMAL) {
        goto func_exit;
    }

    if (linear_out[1] > linear_out[0]) {
        *is_voice = true;
    }

func_exit:
    if (conv_out.data) {
        free(conv_out.data);
    }

    return ret;
}
```

主函数实现

`main.c`

```
c复制代码#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

#include "vad.h"
#include "algo_error_code.h"

#define RAW_FS     (8000)
#define OBJ_FS     (8000)
#define FRAME_STEP (120) // 0.015 * 8000
#define FRAME_LEN  (240) // 0.03 * 8000

uint64_t get_rows(char *file_dir)
{
    char line[1024];
    uint64_t i = 0;
    FILE *stream = fopen(file_dir, "r");
    if (stream) {
        while (fgets(line, 1024, stream)) {
            i++;
        }
        fclose(stream);
    }
    return i;
}

void get_data(char *file_dir, double *data_buf)
{
    char line[1024];
    uint64_t i = 0;
```

```c
    FILE *stream = fopen(file_dir, "r");
    if (stream) {
        while (fgets(line, 1024, stream)) {
            data_buf[i] = strtod(line, NULL);
            i++;
        }
        fclose(stream);
    }
}

void downsample(double *raw_data, uint64_t raw_size, uint16_t raw_fs, uint16_t
obj_fs, double *out, uint64_t *out_size)
{
    uint16_t interval = raw_fs / obj_fs;
    uint64_t i        = 0;
    *out_size = 0;
    for (i = 0; i < raw_size; i += interval) {
        out[(*out_size)++] = raw_data[i];
    }
}

/**
 * voice_segment: 2n: start index, 2n+1:end index
 */
void cal_voice_segment(int8_t *pred_class, const uint64_t *pred_idx_in_data,
uint64_t pred_class_size, uint64_t raw_data_size, uint64_t *voice_segment,
uint64_t *voice_segment_size)
{
    uint64_t i = 0, voice_segment_cnt = 0;
    int8_t diff_vaule = 0;
    bool is_start     = true;
    *voice_segment_size = 0;
    for (i = 1; i < pred_class_size; i++) {
        diff_vaule = pred_class[i] - pred_class[i - 1];
        if (diff_vaule == 1) {
            voice_segment[voice_segment_cnt++] = pred_idx_in_data[i];
            is_start                           = false;
        }
        if (diff_vaule == -1) {
            if (is_start) {
                voice_segment[voice_segment_cnt++] = 0;
            }
            voice_segment[voice_segment_cnt++] = pred_idx_in_data[i];
            is_start                           = true;
        }
    }
    if (!is_start) {
        voice_segment[voice_segment_cnt++] = raw_data_size - 1;
    }
    *voice_segment_size = voice_segment_cnt;
}

int main()
{
    char file_dir[] = "./data.txt";
    FILE *file      = fopen("./pred.txt", "w");
```

```
    int ret            = ALGO_NORMAL;
    uint64_t data_size = 0, down_size = 0, pred_cnt = 0, i = 0, voice_seg_size =
0;
    double *total_data       = NULL;
    bool vad_out             = false;
    int8_t *total_pred       = NULL;
    uint64_t *total_pred_idx = NULL, *all_voice_segment = NULL;
    Conv2dData vad_inp = {.channel = 1, .row = 1, .col = FRAME_LEN, .data =
NULL};
    data_size = get_rows(file_dir);
    printf("data_size = %llu\n", data_size);
    total_data = (double *)malloc(sizeof(double) * data_size);
    if (!total_data) {
        printf("malloc fail\n");
        return 0;
    }
    // get data and downsample
    get_data(file_dir, total_data);
    downsample(total_data, data_size, RAW_FS, OBJ_FS, total_data, &down_size);
    printf("down_size = %llu\n", down_size);
    total_pred = (int8_t *)malloc(sizeof(int8_t) * ((down_size - FRAME_LEN) /
FRAME_STEP + 1));
    if (!total_pred) {
        printf("malloc fail\n");
        goto exit;
    }
    total_pred_idx =
        (uint64_t *)malloc(sizeof(uint64_t) * ((down_size - FRAME_LEN) /
FRAME_STEP + 1));
    if (!total_pred_idx) {
        printf("malloc fail\n");
        goto exit;
    }
    all_voice_segment =
        (uint64_t *)malloc(sizeof(uint64_t) * ((down_size - FRAME_LEN) /
FRAME_STEP + 1));
    if (!all_voice_segment) {
        printf("malloc fail\n");
        goto exit;
    }
    // streaming audio data, frame by frame
    for (i = 0; i < down_size; i += FRAME_STEP) {
        if (i + FRAME_LEN - 1 > down_size) {
            break;
        }
        vad_inp.data = total_data + i;
        ret = vad(&vad_inp, &vad_out);
        if (ret != ALGO_NORMAL) {
            printf("ret = %d\n", ret);
            goto exit;
        }
        total_pred[pred_cnt]       = (int8_t)vad_out;
        total_pred_idx[pred_cnt++] = i;
    }
    // calaulate voice segments
```

```
    cal_voice_segment(total_pred, total_pred_idx, pred_cnt, down_size,
all_voice_segment, &voice_seg_size);
    // save the results to a file
    if (file) {
        for (i = 0; i < voice_seg_size; i += 2) {
            printf("%llu, %llu\n", all_voice_segment[i],all_voice_segment[i+1]);
            fprintf(file, "%llu, %llu\n", all_voice_segment[i],
all_voice_segment[i + 1]);
        }
    }
    fclose(file);

exit:
    free(total_data);
    free(total_pred);
    free(total_pred_idx);
    free(all_voice_segment);

    return 0;
}
```

输出结果

```
● book@100ask:~/CNN模型_C/CNN模型_C$ ./vad
data_size = 502614
down_size = 502614
1320, 1560
17640, 18840
18960, 22680
40800, 43080
43320, 46080
67560, 69960
70080, 73080
94560, 95880
96240, 96600
97080, 99960
115800, 115920
117000, 122160
122400, 122640
140640, 141720
141960, 142920
143160, 145920
163560, 164760
165960, 167400
167760, 168960
185880, 187080
187320, 188400
188760, 191640
208920, 210360
210720, 211680
211920, 213480
213720, 214560
233520, 234600
234840, 238920
256800, 258000
258120, 262440
280920, 282360
282480, 286800
304080, 305400
305760, 309600
326640, 327960
328080, 328920
329040, 331800
349920, 350880
351240, 354960
368520, 368640
371040, 376080
380040, 380160
392040, 396960
411000, 411120
411600, 412800
413040, 415440
415560, 416760
432720, 437760
452400, 453600
453840, 454680
455160, 457680
462960, 463080
463320, 463440
```