```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time, random, sys
import sklearn.metrics
from sklearn.model_selection import KFold

import torch
import torch.nn as nn
import torch.backends.cudnn as cudnn
import torch.optim as optim
from torch.utils.data import DataLoader

sys.path.append('.')
from utils import AverageMeter, cycle, CosineLR, MelDataset, cal
culate_per_class_lwlrap
from models import ResNet

# set parameters
NUM_FOLD = 5
NUM_CLASS = 80
SEED = 42
NUM_EPOCH = 64*8
NUM_CYCLE = 64
BATCH_SIZE = 64
LR = [1e-3, 1e-6]
FOLD_LIST = [1, 2, 3, 4, 5]
SLICE_LENGTH = 512
LOAD_DIR = "../models/resnet_model1"
OUTPUT_DIR = "../input/pseudo_label"

cudnn.benchmark = True
starttime = time.time()


def main():
    # load table data
    df_train = pd.read_csv("../input/train_curated.csv")
    df_noisy = pd.read_csv("../input/train_noisy.csv")
    df_test = pd.read_csv("../input/sample_submission.csv")
    labels = df_test.columns[1:].tolist()
    for label in labels:
        df_train[label] = df_train['labels'].apply(lambda x: lab
el in x)
        df_noisy[label] = df_noisy['labels'].apply(lambda x: lab
el in x)

    df_train['path'] = "../input/mel128/train/" + df_train['fnam
e']
    df_test['path'] = "../input/mel128/test/" + df_train['fname'
]
    df_noisy['path'] = "../input/mel128/noisy/" + df_noisy['fnam
e']

    # fold splitting
    folds = list(KFold(n_splits=NUM_FOLD, shuffle=True, random_s
```

```
tate=SEED).split(np.arange(len(df_train))))

    # build model
    model = ResNet(NUM_CLASS).cuda()

    # set generator
    dataset_noisy = MelDataset(df_noisy['path'], df_noisy[labels
].values)
    noisy_loader = DataLoader(dataset_noisy, batch_size=1,
                              shuffle=False, num_workers=1, pin_
memory=True,
                              )

    # predict
    preds_noisy = np.zeros([NUM_FOLD, NUM_EPOCH//NUM_CYCLE, len(
df_noisy), NUM_CLASS], np.float32)
    for fold, (ids_train_split, ids_valid_split) in enumerate(fo
lds):
        for cycle in range(NUM_EPOCH//NUM_CYCLE):
            print("fold: {} cycle: {}, sec: {:.1f}".format(fold+
1, cycle+1, time.time()-starttime))
            model.load_state_dict(torch.load("{}/weight_fold_{}_
epoch_{}.pth".format(
                LOAD_DIR, fold+1, NUM_CYCLE*(cycle+1))))
            preds_noisy[fold, cycle] = predict(noisy_loader, mod
el)

        np.save("{}/preds_noisy.npy".format(OUTPUT_DIR), preds_n
oisy)


def predict(test_loader, model):
    sigmoid = nn.Sigmoid().cuda()

    # switch to eval mode
    model.eval()

    preds = np.zeros([0, NUM_CLASS], np.float32)
    for i, (input, target) in enumerate(test_loader):
        input = torch.autograd.Variable(input.cuda())

        # compute output
        with torch.no_grad():
            output = model(input)
            pred = sigmoid(output)
            pred = pred.data.cpu().numpy()

        # measure accuracy and record loss
        preds = np.concatenate([preds, pred])
    return preds


if __name__ == '__main__':
    main()import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
import time, random, sys, os
import sklearn.metrics
from sklearn.model_selection import KFold
from multiprocessing import Pool
import gc
import shutil
from scipy.io import wavfile
import librosa
import concurrent.futures

import torch
import torch.nn as nn
import torch.backends.cudnn as cudnn
import torch.optim as optim
from torch.utils.data.dataset import Dataset
from torch.utils.data import DataLoader

sys.path.append('.')
from utils import AverageMeter, cycle, CosineLR, MelDataset, cal
culate_per_class_lwlrap
from models import ResNet, EnvNetv2


# set parameters
NUM_FOLD = 5
NUM_CLASS = 80
SEED = 42
BATCH_DIR = "batch"
SIZE_LIMIT = 20000000
WIDTH_LIMIT = 80000
MAX_LEN = 1400000
MAX_PAD = 32000
MAX_BATCHSIZE = 512
LEN_DF_WAV_LIMIT = 500000000
LEN_DF_MEL_LIMIT = 2000000000
NUMBATCH_PER_NUMDATA = 1 / 55
MAX_PATIENCE = 0.2
wav_dir = "../input/test/"

RES_LIST = [
    {'dir': '../models/resnet_model1',
     'epoch': [2 * 64, 4 * 64, 7 * 64, 8 * 64],
     'pad': [8, 64],
     },
    {'dir': '../models/resnet_model2',
     'epoch': [1 * 64, 2 * 64, 6 * 64, 7 * 64],
     'pad': [8, 64],
     },
    {'dir': '../models/resnet_model3',
     'epoch': [2 * 64, 4 * 64, 6 * 64],
     'pad': [8, 64],
     },
]
ENV_LIST = [
    {'dir': '../models/envnet_model4',
     'epoch': [2 * 80, 3 * 80],
```

```
     'pad': [0, 32000],
     'acitivation': 'sigmoid',
     },
    {'dir': '../models/envnet_model5',
     'epoch': [3 * 80, 5 * 80],
     'pad': [8000, 32000],
     'acitivation': 'softmax',
     },
    {'dir': '../models/envnet_model6',
     'epoch': [2 * 80, 4 * 80],
     'pad': [8000, 32000],
     'acitivation': 'softmax',
     },
]
LEN_RES_EPOCH = 0
for i in range(len(RES_LIST)):
    LEN_RES_EPOCH = max(LEN_RES_EPOCH, len(RES_LIST[i]['epoch'])
)
LEN_RES_PAD = 0
for i in range(len(RES_LIST)):
    LEN_RES_PAD = max(LEN_RES_PAD, len(RES_LIST[i]['pad']))
LEN_ENV_EPOCH = 0
for i in range(len(ENV_LIST)):
    LEN_ENV_EPOCH = max(LEN_ENV_EPOCH, len(ENV_LIST[i]['epoch'])
)
LEN_ENV_PAD = 0
for i in range(len(ENV_LIST)):
    LEN_ENV_PAD = max(LEN_ENV_EPOCH, len(ENV_LIST[i]['pad']))
starttime0 = time.time()

# cudnn speed up
cudnn.benchmark = True


def main():
    ### fix seed
    torch.manual_seed(SEED)
    random.seed(SEED)
    np.random.seed(SEED)
    torch.manual_seed(SEED)
    torch.cuda.manual_seed(SEED)

    # table data load
    starttime = time.time()
    df_test = pd.read_csv("../input/sample_submission.csv")
    labels = df_test.columns[1:].tolist()
    df_test['path'] = "{}/".format(wav_dir) + df_test['fname']
    print("table data loading done. {:.1f}/{:.1f}".format(time.t
ime() - starttime, time.time() - starttime0))

    # get data length
    starttime = time.time()
    p = Pool(2)
    len_list = p.map(get_len, df_test['path'].values)
    df_test['length'] = len_list
    print("getting data length done. {:.1f}/{:.1f}".format(time.
```

```
time() - starttime, time.time() - starttime0))

    # data sort
    starttime = time.time()
    df_test_sort = df_test.copy()
    df_test_sort['index'] = np.arange(len(df_test_sort))
    df_test_sort = df_test_sort.sort_values(['length', 'index'])
.reset_index(drop=True)
    print("data sort done. {:.1f}/{:.1f}".format(time.time() - s
tarttime, time.time() - starttime0))

    # batch splitting
    starttime = time.time()
    NUM_BATCH_LIMIT = 60 + int(len(df_test_sort) * NUMBATCH_PER_
NUMDATA)
    print("num batch limit: {}".format(NUM_BATCH_LIMIT))
    patience_rate = 0
    patience_rate_tmp = 0
    num_batch, count = get_num_batch(df_test_sort, patience_rate
)
    print("patience_rate_tmp: {:.2f}, patience_rate_tmp: {:.2f},
 num_batch: {:3d}".format(
        patience_rate, patience_rate_tmp, num_batch))
    while num_batch > NUM_BATCH_LIMIT and patience_rate_tmp < MA
X_PATIENCE:
        patience_rate_tmp += 0.01
        num_batch_tmp, count_tmp = get_num_batch(df_test_sort, p
atience_rate_tmp)
        if num_batch_tmp < num_batch:
            num_batch = num_batch_tmp
            count = count_tmp
            patience_rate = patience_rate_tmp
        print("patience_rate_tmp: {:.2f}, patience_rate_tmp: {:.
2f}, num_batch_tmp: {:3d}".format(
            patience_rate, patience_rate_tmp, num_batch_tmp))
    num_batch, count = get_num_batch(df_test_sort, patience_rate
)
    print("num batch: {}, rate of padding patience: {:.2f}".form
at(num_batch, patience_rate))
    print("batch splitting done. {:.1f}/{:.1f}".format(time.time
() - starttime, time.time() - starttime0))

    # store batch id
    starttime = time.time()
    batch_list = []
    for i in range(num_batch):
        batch_list += [i] * count[i][1]
    df_test_sort['batch'] = batch_list
    print(df_test_sort[['path', 'length', 'batch']].head())
    print("save batch id done. {:.1f}/{:.1f}".format(time.time()
 - starttime, time.time() - starttime0))

    # split dataframe if too big
    starttime = time.time()
    df_mel_split = get_df_split(df_test_sort, LEN_DF_MEL_LIMIT)
    print("df_mel_split")
```

```
    for i in range(len(df_mel_split)):
        print("{}: num data: {}, total length: {}".format(i + 1,
 len(df_mel_split[i]), df_mel_split[i]['length'].sum()))
    print("dataframe splitting done. {:.1f}/{:.1f}".format(time.
time() - starttime, time.time() - starttime0))

    # ### EnvNet part
    # build model
    model = EnvNetv2(NUM_CLASS).cuda()
    model.eval()

    # split df for EnvNet
    df_wav_split = get_df_split(df_test_sort, LEN_DF_WAV_LIMIT)
    print("df_wav_split")
    for i in range(len(df_wav_split)):
        print("{}: num data: {}, total length: {}".format(i + 1,
 len(df_wav_split[i]), df_wav_split[i]['length'].sum()))

    print("predict wav...")

    # parallel threading
    executor = concurrent.futures.ThreadPoolExecutor(max_workers
=2)
    threadA = executor.submit(get_mel_batch, df_mel_split[0])
    threadB = executor.submit(predict_wav_split, model, df_wav_s
plit[0], ENV_LIST)
    preds_wav_split = []
    preds_wav_split.append(threadB.result())
    executor.shutdown()
    print("parallel threading done.", time.time() - starttime, t
ime.time() - starttime0)

    # do remain EnvNet prediction
    if len(df_wav_split) > 1:
        for split in range(1, len(df_wav_split)):
            preds_wav_split.append(predict_wav_split(model, df_w
av_split[split], ENV_LIST))
            print("envnet prediction split {}/{}, done. {:.1f}/{
:.1f}".format(
                split + 1, len(df_wav_split), time.time() - star
ttime, time.time() - starttime0))
    preds_test_wav = np.concatenate(preds_wav_split, axis=4)
    print("all envnet predict done.", time.time() - starttime, t
ime.time() - starttime0)

    # build model
    starttime = time.time()
    model = ResNet(NUM_CLASS).cuda()
    model.eval()
    print("building ResNet model done. {:.1f}/{:.1f}".format(tim
e.time() - starttime, time.time() - starttime0))

    # predict split #1
    preds_test_mel = []
    preds_test_mel.append(predict_mel_split(model, df_mel_split[
0], RES_LIST))
```

```
    shutil.rmtree(BATCH_DIR)
    print("mel prediction of split {} done. {:.1f}/{:.1f}".forma
t(1, time.time() - starttime, time.time() - starttime0))

    # process remain split
    if len(df_mel_split) > 1:
        for split in range(1, len(df_mel_split)):
            # mel preprocessing
            starttime = time.time()
            df_test_sort_tmp = df_mel_split[split]
            get_mel_batch(df_test_sort_tmp)
            print("mel preprocessing of split {} done. {:.1f}/{:
.1f}".format(
                split + 1, time.time() - starttime, time.time()
- starttime0))
            preds_test_mel.append(predict_mel_split(model, df_te
st_sort_tmp, RES_LIST))
            shutil.rmtree(BATCH_DIR)
            print("mel prediction of split {} done. {:.1f}/{:.1f
}".format(
                split + 1, time.time() - starttime, time.time()
- starttime0))

    print("all prediction done. {:.1f}/{:.1f}".format(time.time(
) - starttime, time.time() - starttime0))

    # concat
    starttime = time.time()
    preds_test_mel = np.concatenate(preds_test_mel, axis=4)
    print("preds_test_mel.shape", preds_test_mel.shape)
    print("concat done.", time.time() - starttime, time.time() -
 starttime0)

    # make submission
    preds_test_avr = (
            + preds_test_mel[:, 0, :len(RES_LIST[0]['epoch'])].m
ean(axis=(0, 1, 2)) * 4 / 13
            + preds_test_mel[:, 1, :len(RES_LIST[1]['epoch'])].m
ean(axis=(0, 1, 2)) * 3 / 13
            + preds_test_mel[:, 2, :len(RES_LIST[2]['epoch'])].m
ean(axis=(0, 1, 2)) * 3 / 13
            + preds_test_wav[:, 0, :len(ENV_LIST[0]['epoch'])].m
ean(axis=(0, 1, 2)) * 1 / 13
            + preds_test_wav[:, 1, :len(ENV_LIST[1]['epoch'])].m
ean(axis=(0, 1, 2)) * 1 / 13
            + preds_test_wav[:, 2, :len(ENV_LIST[2]['epoch'])].m
ean(axis=(0, 1, 2)) * 1 / 13)
    print(preds_test_mel.shape, preds_test_wav.shape)
    print(preds_test_avr.shape)
    df_test_sort = df_test_sort.sort_values(['length', 'index'])
.reset_index(drop=True)
    df_test_sort[labels] = preds_test_avr
    df_test_sort = df_test_sort.sort_values('index').reset_index
(drop=True)
    df_test_sort[['fname'] + labels].to_csv("../output/submissio
n1.csv", index=None)
```

```
    print("save submission done. {:.1f}/{:.1f}".format(time.time
() - starttime, time.time() - starttime0))


def check_size_limit(width, num_batch):
    if (width + MAX_PAD * 2) * num_batch > SIZE_LIMIT:
        return True
    else:
        return False


def get_num_batch(df_test_sort, patience_rate):
    i = 0
    len_now = df_test_sort['length'][i]
    patience = int(len_now * patience_rate)
    count = []
    while (i < len(df_test_sort)):
        len_now = df_test_sort['length'][i]
        patience = int(len_now * patience_rate)
        if len(count) == 0 or count[-1][0] + patience < len_now
or count[-1][1] >= MAX_BATCHSIZE:
            count.append([len_now, 1])
        elif check_size_limit(len_now, count[-1][1] + 1):
            count.append([len_now, 1])
        else:
            count[-1][1] += 1
        i += 1
    return len(count), count


def predict_mel_split(model, df_split, RES_LIST):
    starttime = time.time()
    batch_idx = [df_split['batch'].min(), df_split['batch'].max(
) + 1]
    preds_test_mel_tmp = np.zeros([
        NUM_FOLD,
        len(RES_LIST),
        len(RES_LIST[0]['epoch']),
        len(RES_LIST[0]['pad']),
        len(df_split), NUM_CLASS], np.float32)

    dataset_valid = BatchDataset(df_split, 0)
    valid_loader = DataLoader(dataset_valid,
                              batch_size=1,
                              shuffle=False,
                              num_workers=1,
                              pin_memory=True,
                              collate_fn=my_collate
                              )
    for i in range(len(RES_LIST)):
        model_dir = RES_LIST[i]['dir']
        epoch_list = RES_LIST[i]['epoch']
        pad_list = RES_LIST[i]['pad']
        for fold in range(NUM_FOLD):
            for k, epoch in enumerate(epoch_list):
                model.load_state_dict(
```

```
                    torch.load("{}/weight_fold_{}_epoch_{}.pth".
format(model_dir, fold + 1, epoch)))
                for j, pad in enumerate(pad_list):
                    print("fold: {}, dir: {}, epoch: {}, pad: {}
, sec: {:.1f}".format(
                        fold + 1, model_dir, epoch, pad, time.ti
me() - starttime))
                    dataset_valid.pad = pad
                    preds_test_mel_tmp[fold, i, k, j] = predict_
resnet(model, valid_loader)
    return preds_test_mel_tmp


def get_mel_batch(df_split):
    df_split['path'] = "{}/".format(wav_dir) + df_split['fname']
    print(df_split[['path', 'batch']])
    os.makedirs(BATCH_DIR, exist_ok=True)
    p = Pool(2)  # ¿¿¿¿¿¿¿=2
    batch_idx = [df_split['batch'].min(), df_split['batch'].max(
) + 1]
    for i in range(batch_idx[0], batch_idx[1]):
        df_tmp = df_split[df_split['batch'] == i].reset_index(dr
op=True)
        args = []
        slice = int(np.ceil((df_tmp['length'].values[-1] + 1) /
347))
        slice = np.min([slice, WIDTH_LIMIT])
        for j in range(len(df_tmp)):
            args.append([df_tmp['path'][j], slice])
        batch = p.map(preprocess_mel, args)
        batch = np.array(batch)
        np.save("{}/{}.npy".format(BATCH_DIR, i), batch)


def predict_wav_split(model, df, ENV_LIST):
    starttime = time.time()
    batch_idx = [df['batch'].min(), df['batch'].max() + 1]
    p = Pool(2)  # ¿¿¿¿¿¿¿=2
    batch_list = []
    for i in range(batch_idx[0], batch_idx[1]):
        df_tmp = df[df['batch'] == i].reset_index(drop=True)
        args = []
        slice = df_tmp['length'].values[-1]
        for j in range(len(df_tmp)):
            args.append([df_tmp['path'][j], slice])
        batch = p.map(preprocess_wav, args)
        batch = np.array(batch)
        batch_list.append(batch)
    print("batch making done, sec: {:.1f}".format(time.time() -
starttime))

    # envnet predict
    starttime = time.time()
    print("predict valid...")
    preds_test_wav = np.zeros([
        NUM_FOLD,
```

```
        len(ENV_LIST),
        len(ENV_LIST[0]['epoch']),
        len(ENV_LIST[0]['pad']),
        len(df), NUM_CLASS], np.float32)

    dataset_valid = BatchWavDataset(batch_list, 0)
    valid_loader = DataLoader(dataset_valid,
                              batch_size=1,
                              shuffle=False,
                              num_workers=1,
                              pin_memory=True,
                              collate_fn=my_collate
                              )
    for i in range(len(ENV_LIST)):
        model_dir = ENV_LIST[i]['dir']
        epoch_list = ENV_LIST[i]['epoch']
        pad_list = ENV_LIST[i]['pad']
        activation = ENV_LIST[i]['acitivation']

        for fold in range(NUM_FOLD):
            for k, epoch in enumerate(epoch_list):
                model.load_state_dict(
                    torch.load("{}/weight_fold_{}_epoch_{}.pth".
format(model_dir, fold + 1, epoch),
                               map_location='cuda:0'))
                for j, pad in enumerate(pad_list):
                    print("fold: {}, dir: {}, epoch: {}, pad: {}
, sec: {:.1f}".format(
                        fold + 1, model_dir, epoch, pad, time.ti
me() - starttime))
                    dataset_valid.pad = pad
                    preds_test_wav[fold, i, k, j] = predict_envn
et(model, valid_loader, activation)
    return preds_test_wav


def get_df_split(df, size_limit):
    num_batch = df['batch'].max() + 1
    sum_len = df['length'].sum()
    df_split = []
    begin = 0
    sum_tmp = 0
    print("base df shape", df.shape)
    for i in range(num_batch):
        sum_tmp += df['length'][df['batch'] == i].sum()
        if sum_tmp > size_limit:
            df_split.append(
                df[(begin <= df['batch']) & (df['batch'] < i)].r
eset_index(drop=True))
            sum_tmp = df['length'][df['batch'] == i].sum()
            begin = i
    df_split.append(df[(begin <= df['batch'])].reset_index(drop=
True))
    return df_split
```

```
def my_collate(batch):
    return torch.Tensor(batch[0])


def get_len(path):
    _, data = wavfile.read(path)
    len_data = len(data)
    if len(data) > MAX_LEN:
        len_data = MAX_LEN
        print("File length {} is too long! This file is sliced t
o {}.".format(len(data), MAX_LEN))

    return len_data


def get_wav(path):
    _, snd = wavfile.read(path)
    return snd


def get_mel(wave):
    wave = wave.astype(np.float32) / 32768.0
    data = librosa.feature.melspectrogram(
        wave,
        sr=44100,
        n_mels=128,
        hop_length=347 * 1,
        n_fft=128 * 20,
        fmin=20,
        fmax=44100 // 2,
    ).astype(np.float32)
    return data


def preprocess_mel(args):
    path, slice = args
    wav = get_wav(path)
    mel = get_mel(wav)
    mel_new = np.zeros([mel.shape[0], slice], np.float32)
    if mel.shape[1] > slice:
        print("wav length: {}, mel length: {}".format(wav.shape[
0], mel.shape[1]))
        print("Mel file is sliced")
        mel_new[:] = mel[:, :slice]
    else:
        mel_new[:, :mel.shape[1]] = mel
    mel_new = librosa.power_to_db(mel_new)
    mel_new = mel_new.reshape([1, mel_new.shape[0], mel_new.shap
e[1]])
    return mel_new


def preprocess_wav(args):
    path, slice = args
    wav = get_wav(path)
    pad = (slice - len(wav)) // 2
```

```
    wav_new = np.zeros([1, 1, slice], np.int16)

    if wav.shape[0] > slice:
        print("wav length: {}".format(wav.shape[0]))
        print("Wav file is sliced")
        wav_new[0, 0, :] = wav[:slice]
    else:
        wav_new[0, 0, pad:pad + len(wav)] = wav
    return wav_new


class BatchDataset(Dataset):
    def __init__(self, df, pad=0):
        self.len_batch = df['batch'].max() - df['batch'].min() +
 1
        self.X = np.arange(self.len_batch) + df['batch'].min()
        self.pad = pad

    def __getitem__(self, index):
        batch_base = np.load("{}/{}.npy".format(BATCH_DIR, self.
X[index]))
        batch_pad = np.zeros(batch_base.shape[:-1] + (batch_base
.shape[-1] + self.pad * 2,), np.float32)
        batch_max = batch_base.max(axis=(1, 2, 3)) - 80
        batch_max = np.maximum(batch_max, -100)
        batch_pad[:] = batch_max[:, np.newaxis, np.newaxis, np.n
ewaxis, ]
        if self.pad != 0:
            batch_pad[:, :, :, self.pad:-self.pad] = batch_base
        else:
            batch_pad[:] = batch_base
        batch_pad = (batch_pad - batch_pad.mean(axis=(1, 2, 3))[
:, np.newaxis, np.newaxis, np.newaxis, ]) / (
                batch_pad.std(axis=(1, 2, 3))[:, np.newaxis, np.
newaxis, np.newaxis, ] + 1e-7)
        return batch_pad

    def __len__(self):
        return self.len_batch


class BatchWavDataset(Dataset):
    def __init__(self, batch_list, pad=0):
        self.X = batch_list
        self.pad = pad

    def __getitem__(self, index):
        batch_base = self.X[index]
        if batch_base.shape[-1] + self.pad * 2 < 20580:
            pad = int(np.ceil((20580 - batch_base.shape[-1]) / 2
))
        else:
            pad = self.pad
        if pad != 0:
            batch = np.zeros([batch_base.shape[0], 1, 1, batch_b
ase.shape[-1] + pad * 2], np.float32)
```

```
            batch[:, :, :, pad:-pad] = batch_base.astype(np.floa
t32) / 32768.0
        else:
            batch = batch_base.astype(np.float32) / 32768.0
        return batch

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


def predict_resnet(model, dataloader):
    sigmoid = torch.nn.Sigmoid().cuda()
    preds = []
    model.eval()
    for i, input in enumerate(dataloader):
        input = input.cuda(async=True)
        with torch.no_grad():
            pred = sigmoid(model(input)).data.cpu().numpy()
        preds.append(pred)
    preds = np.concatenate(preds)
    return preds


def predict_envnet(model, dataloader, activation='sigmoid'):
    sigmoid = torch.nn.Sigmoid().cuda()
    softmax = torch.nn.Softmax(dim=1).cuda()
    if activation == 'sigmoid':
        f_act = sigmoid
    elif activation == 'softmax':
        f_act = softmax
    preds = []
    model.eval()
    for i, input in enumerate(dataloader):
        input = input.cuda(async=True)
        with torch.no_grad():
            pred = f_act(model(input)).data.cpu().numpy()
        preds.append(pred)
    preds = np.concatenate(preds)
    return preds


if __name__ == '__main__':
    main()import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time, random, sys, os
import sklearn.metrics
from sklearn.model_selection import KFold
from multiprocessing import Pool
import gc
import shutil
from scipy.io import wavfile
import librosa
import concurrent.futures

import torch
```

```
import torch.nn as nn
import torch.backends.cudnn as cudnn
import torch.optim as optim
from torch.utils.data.dataset import Dataset
from torch.utils.data import DataLoader

sys.path.append('.')
from utils import AverageMeter, cycle, CosineLR, MelDataset, cal
culate_per_class_lwlrap
from models import ResNet, EnvNetv2


# set parameters
NUM_FOLD = 5
NUM_CLASS = 80
SEED = 42
BATCH_DIR = "batch"
SIZE_LIMIT = 20000000
WIDTH_LIMIT = 80000
MAX_LEN = 1400000
MAX_PAD = 32000
MAX_BATCHSIZE = 512
LEN_DF_WAV_LIMIT = 500000000
LEN_DF_MEL_LIMIT = 2000000000
NUMBATCH_PER_NUMDATA = 1 / 30
MAX_PATIENCE = 0.2
wav_dir = "../input/test/"

RES_LIST = [
    {'dir': '../models/resnet_model1',
     'epoch': [2*64,4*64,7*64,8*64],
     'pad': [8],
     },
    {'dir': '../models/resnet_model2',
     'epoch': [1*64,2*64,4*64,6*64,7*64],
     'pad': [8],
     },
    {'dir': '../models/resnet_model3',
     'epoch': [2*64,4*64,6*64],
     'pad': [8],
     },
]
ENV_LIST = [
    {'dir': '../models/envnet_model4',
     'epoch': [2*80,3*80],
     'pad': [8000],
     'acitivation': 'sigmoid',
     },
    {'dir': '../models/envnet_model5',
     'epoch': [3 * 80, 5 * 80],
     'pad': [8000],
     'acitivation': 'softmax',
     },
    {'dir': '../models/envnet_model6',
     'epoch': [1*80,2*80,4*80],
     'pad': [8000],
```

```
        'acitivation': 'softmax',
        },
]
LEN_RES_EPOCH = 0
for i in range(len(RES_LIST)):
    LEN_RES_EPOCH = max(LEN_RES_EPOCH, len(RES_LIST[i]['epoch'])
)
LEN_RES_PAD = 0
for i in range(len(RES_LIST)):
    LEN_RES_PAD = max(LEN_RES_PAD, len(RES_LIST[i]['pad']))
LEN_ENV_EPOCH = 0
for i in range(len(ENV_LIST)):
    LEN_ENV_EPOCH = max(LEN_ENV_EPOCH, len(ENV_LIST[i]['epoch'])
)
LEN_ENV_PAD = 0
for i in range(len(ENV_LIST)):
    LEN_ENV_PAD = max(LEN_ENV_EPOCH, len(ENV_LIST[i]['pad']))
starttime0 = time.time()

# cudnn speed up
cudnn.benchmark = True


def main():
    ### fix seed
    torch.manual_seed(SEED)
    random.seed(SEED)
    np.random.seed(SEED)
    torch.manual_seed(SEED)
    torch.cuda.manual_seed(SEED)

    # table data load
    starttime = time.time()
    df_test = pd.read_csv("../input/sample_submission.csv")
    labels = df_test.columns[1:].tolist()
    df_test['path'] = "{}/".format(wav_dir) + df_test['fname']
    print("table data loading done. {:.1f}/{:.1f}".format(time.t
ime() - starttime, time.time() - starttime0))

    # get data length
    starttime = time.time()
    p = Pool(2)
    len_list = p.map(get_len, df_test['path'].values)
    df_test['length'] = len_list
    print("getting data length done. {:.1f}/{:.1f}".format(time.
time() - starttime, time.time() - starttime0))

    # data sort
    starttime = time.time()
    df_test_sort = df_test.copy()
    df_test_sort['index'] = np.arange(len(df_test_sort))
    df_test_sort = df_test_sort.sort_values(['length', 'index'])
.reset_index(drop=True)
    print("data sort done. {:.1f}/{:.1f}".format(time.time() - s
tarttime, time.time() - starttime0))
```

```
    # batch splitting
    starttime = time.time()
    NUM_BATCH_LIMIT = 50 + int(len(df_test_sort)*NUMBATCH_PER_NU
MDATA)
    print("num batch limit: {}".format(NUM_BATCH_LIMIT))
    patience_rate = 0
    patience_rate_tmp = 0
    num_batch, count = get_num_batch(df_test_sort, patience_rate
)
    print("patience_rate_tmp: {:.2f}, patience_rate_tmp: {:.2f},
 num_batch: {:3d}".format(
        patience_rate, patience_rate_tmp, num_batch))
    while num_batch > NUM_BATCH_LIMIT and patience_rate_tmp < MA
X_PATIENCE:
        patience_rate_tmp += 0.01
        num_batch_tmp, count_tmp = get_num_batch(df_test_sort, p
atience_rate_tmp)
        if num_batch_tmp < num_batch:
            num_batch = num_batch_tmp
            count = count_tmp
            patience_rate = patience_rate_tmp
        print("patience_rate_tmp: {:.2f}, patience_rate_tmp: {:.
2f}, num_batch_tmp: {:3d}".format(
            patience_rate, patience_rate_tmp, num_batch_tmp))
    num_batch, count = get_num_batch(df_test_sort, patience_rate
)
    print("num batch: {}, rate of padding patience: {:.2f}".form
at(num_batch, patience_rate))
    print("batch splitting done. {:.1f}/{:.1f}".format(time.time
() - starttime, time.time() - starttime0))

    # store batch id
    starttime = time.time()
    batch_list = []
    for i in range(num_batch):
        batch_list += [i] * count[i][1]
    df_test_sort['batch'] = batch_list
    print(df_test_sort[['path', 'length', 'batch']].head())
    print("save batch id done. {:.1f}/{:.1f}".format(time.time()
 - starttime, time.time() - starttime0))

    # split dataframe if too big
    starttime = time.time()
    df_mel_split = get_df_split(df_test_sort, LEN_DF_MEL_LIMIT)
    print("df_mel_split")
    for i in range(len(df_mel_split)):
        print("{}: num data: {}, total length: {}".format(i + 1,
 len(df_mel_split[i]), df_mel_split[i]['length'].sum()))
    print("dataframe splitting done. {:.1f}/{:.1f}".format(time.
time() - starttime, time.time() - starttime0))

    # ### EnvNet part
    # build model
    model = EnvNetv2(NUM_CLASS).cuda()
    model.eval()
```

```
    # split df for EnvNet
    df_wav_split = get_df_split(df_test_sort, LEN_DF_WAV_LIMIT)
    print("df_wav_split")
    for i in range(len(df_wav_split)):
        print("{}: num data: {}, total length: {}".format(i + 1,
 len(df_wav_split[i]), df_wav_split[i]['length'].sum()))

    print("predict wav...")

    # parallel threading
    executor = concurrent.futures.ThreadPoolExecutor(max_workers
=2)
    threadA = executor.submit(get_mel_batch, df_mel_split[0])
    threadB = executor.submit(predict_wav_split, model, df_wav_s
plit[0], ENV_LIST)
    preds_wav_split = []
    preds_wav_split.append(threadB.result())
    executor.shutdown()
    print("parallel threading done.", time.time() - starttime, t
ime.time() - starttime0)

    # do remain EnvNet prediction
    if len(df_wav_split) > 1:
        for split in range(1, len(df_wav_split)):
            preds_wav_split.append(predict_wav_split(model, df_w
av_split[split], ENV_LIST))
            print("envnet prediction split {}/{}, done. {:.1f}/{
:.1f}".format(
                split + 1, len(df_wav_split), time.time() - star
ttime, time.time() - starttime0))
    preds_test_wav = np.concatenate(preds_wav_split, axis=4)
    print("all envnet predict done.", time.time() - starttime, t
ime.time() - starttime0)

    # build model
    starttime = time.time()
    model = ResNet(NUM_CLASS).cuda()
    model.eval()
    print("building ResNet model done. {:.1f}/{:.1f}".format(tim
e.time() - starttime, time.time() - starttime0))

    # predict split #1
    preds_test_mel = []
    preds_test_mel.append(predict_mel_split(model, df_mel_split[
0], RES_LIST))
    shutil.rmtree(BATCH_DIR)
    print("mel prediction of split {} done. {:.1f}/{:.1f}".forma
t(1, time.time() - starttime, time.time() - starttime0))

    # process remain split
    if len(df_mel_split) > 1:
        for split in range(1, len(df_mel_split)):
            # mel preprocessing
            starttime = time.time()
            df_test_sort_tmp = df_mel_split[split]
            get_mel_batch(df_test_sort_tmp)
```

```
            print("mel preprocessing of split {} done. {:.1f}/{:
.1f}".format(
                split + 1, time.time() - starttime, time.time()
- starttime0))
            preds_test_mel.append(predict_mel_split(model, df_te
st_sort_tmp, RES_LIST))
            shutil.rmtree(BATCH_DIR)
            print("mel prediction of split {} done. {:.1f}/{:.1f
}".format(
                split + 1, time.time() - starttime, time.time()
- starttime0))

    print("all prediction done. {:.1f}/{:.1f}".format(time.time(
) - starttime, time.time() - starttime0))

    # concat
    starttime = time.time()
    preds_test_mel = np.concatenate(preds_test_mel, axis=4)
    print("preds_test_mel.shape", preds_test_mel.shape)
    print("concat done.", time.time() - starttime, time.time() -
 starttime0)

    # make submission
    preds_test_avr = (
            + preds_test_mel[:, 0, :len(RES_LIST[0]['epoch'])].m
ean(axis=(0, 1, 2)) * 4 / 13
            + preds_test_mel[:, 1, :len(RES_LIST[1]['epoch'])].m
ean(axis=(0, 1, 2)) * 3 / 13
            + preds_test_mel[:, 2, :len(RES_LIST[2]['epoch'])].m
ean(axis=(0, 1, 2)) * 3 / 13
            + preds_test_wav[:, 0, :len(ENV_LIST[0]['epoch'])].m
ean(axis=(0, 1, 2)) * 1 / 13
            + preds_test_wav[:, 1, :len(ENV_LIST[1]['epoch'])].m
ean(axis=(0, 1, 2)) * 1 / 13
            + preds_test_wav[:, 2, :len(ENV_LIST[2]['epoch'])].m
ean(axis=(0, 1, 2)) * 1 / 13)
    print(preds_test_mel.shape, preds_test_wav.shape)
    print(preds_test_avr.shape)
    df_test_sort = df_test_sort.sort_values(['length', 'index'])
.reset_index(drop=True)
    df_test_sort[labels] = preds_test_avr
    df_test_sort = df_test_sort.sort_values('index').reset_index
(drop=True)
    df_test_sort[['fname'] + labels].to_csv("../output/submissio
n1.csv", index=None)
    print("save submission done. {:.1f}/{:.1f}".format(time.time
() - starttime, time.time() - starttime0))


def check_size_limit(width, num_batch):
    if (width + MAX_PAD * 2) * num_batch > SIZE_LIMIT:
        return True
    else:
        return False
```

```
def get_num_batch(df_test_sort, patience_rate):
    i = 0
    len_now = df_test_sort['length'][i]
    patience = int(len_now * patience_rate)
    count = []
    while (i < len(df_test_sort)):
        len_now = df_test_sort['length'][i]
        patience = int(len_now * patience_rate)
        if len(count) == 0 or count[-1][0] + patience < len_now
or count[-1][1] >= MAX_BATCHSIZE:
            count.append([len_now, 1])
        elif check_size_limit(len_now, count[-1][1] + 1):
            count.append([len_now, 1])
        else:
            count[-1][1] += 1
        i += 1
    return len(count), count


def predict_mel_split(model, df_split, RES_LIST):
    starttime = time.time()
    batch_idx = [df_split['batch'].min(), df_split['batch'].max(
) + 1]
    preds_test_mel_tmp = np.zeros([
        NUM_FOLD,
        len(RES_LIST),
        len(RES_LIST[0]['epoch']),
        len(RES_LIST[0]['pad']),
        len(df_split), NUM_CLASS], np.float32)

    dataset_valid = BatchDataset(df_split, 0)
    valid_loader = DataLoader(dataset_valid,
                              batch_size=1,
                              shuffle=False,
                              num_workers=1,
                              pin_memory=True,
                              collate_fn=my_collate
                              )
    for i in range(len(RES_LIST)):
        model_dir = RES_LIST[i]['dir']
        epoch_list = RES_LIST[i]['epoch']
        pad_list = RES_LIST[i]['pad']
        for fold in range(NUM_FOLD):
            for k, epoch in enumerate(epoch_list):
                model.load_state_dict(
                    torch.load("{}/weight_fold_{}_epoch_{}.pth".
format(model_dir, fold + 1, epoch)))
                for j, pad in enumerate(pad_list):
                    print("fold: {}, dir: {}, epoch: {}, pad: {}
, sec: {:.1f}".format(
                        fold + 1, model_dir, epoch, pad, time.ti
me() - starttime))
                    dataset_valid.pad = pad
                    preds_test_mel_tmp[fold, i, k, j] = predict_
resnet(model, valid_loader)
    return preds_test_mel_tmp
```

```
def get_mel_batch(df_split):
    print(1)
    df_split['path'] = "{}/".format(wav_dir) + df_split['fname']
    print(df_split[['path', 'batch']])
    os.makedirs(BATCH_DIR, exist_ok=True)
    print(2)
    p = Pool(2)  # ¿¿¿¿¿¿¿=2
    batch_idx = [df_split['batch'].min(), df_split['batch'].max(
) + 1]
    for i in range(batch_idx[0], batch_idx[1]):
        df_tmp = df_split[df_split['batch'] == i].reset_index(dr
op=True)
        args = []
        slice = int(np.ceil((df_tmp['length'].values[-1] + 1) /
347))
        slice = np.min([slice, WIDTH_LIMIT])
        for j in range(len(df_tmp)):
            args.append([df_tmp['path'][j], slice])
        batch = p.map(preprocess_mel, args)
        batch = np.array(batch)
        np.save("{}/{}.npy".format(BATCH_DIR, i), batch)


def predict_wav_split(model, df, ENV_LIST):
    starttime = time.time()
    batch_idx = [df['batch'].min(), df['batch'].max() + 1]
    p = Pool(2)  # ¿¿¿¿¿¿¿=2
    batch_list = []
    for i in range(batch_idx[0], batch_idx[1]):
        df_tmp = df[df['batch'] == i].reset_index(drop=True)
        args = []
        slice = df_tmp['length'].values[-1]
        for j in range(len(df_tmp)):
            args.append([df_tmp['path'][j], slice])
        batch = p.map(preprocess_wav, args)
        batch = np.array(batch)
        batch_list.append(batch)
    print("batch making done, sec: {:.1f}".format(time.time() -
starttime))

    # envnet predict
    starttime = time.time()
    print("predict valid...")
    preds_test_wav = np.zeros([
        NUM_FOLD,
        len(ENV_LIST),
        len(ENV_LIST[0]['epoch']),
        len(ENV_LIST[0]['pad']),
        len(df), NUM_CLASS], np.float32)

    dataset_valid = BatchWavDataset(batch_list, 0)
    valid_loader = DataLoader(dataset_valid,
                              batch_size=1,
                              shuffle=False,
```

```
                              num_workers=1,
                              pin_memory=True,
                              collate_fn=my_collate
                              )
    for i in range(len(ENV_LIST)):
        model_dir = ENV_LIST[i]['dir']
        epoch_list = ENV_LIST[i]['epoch']
        pad_list = ENV_LIST[i]['pad']
        activation = ENV_LIST[i]['acitivation']

        for fold in range(NUM_FOLD):
            for k, epoch in enumerate(epoch_list):
                model.load_state_dict(
                    torch.load("{}/weight_fold_{}_epoch_{}.pth".
format(model_dir, fold + 1, epoch),
                               map_location='cuda:0'))
                for j, pad in enumerate(pad_list):
                    print("fold: {}, dir: {}, epoch: {}, pad: {}
, sec: {:.1f}".format(
                        fold + 1, model_dir, epoch, pad, time.ti
me() - starttime))
                    dataset_valid.pad = pad
                    preds_test_wav[fold, i, k, j] = predict_envn
et(model, valid_loader, activation)
    return preds_test_wav


def get_df_split(df, size_limit):
    num_batch = df['batch'].max() + 1
    sum_len = df['length'].sum()
    df_split = []
    begin = 0
    sum_tmp = 0
    print("base df shape", df.shape)
    for i in range(num_batch):
        sum_tmp += df['length'][df['batch'] == i].sum()
        if sum_tmp > size_limit:
            df_split.append(
                df[(begin <= df['batch']) & (df['batch'] < i)].r
eset_index(drop=True))
            sum_tmp = df['length'][df['batch'] == i].sum()
            begin = i
    df_split.append(df[(begin <= df['batch'])].reset_index(drop=
True))
    return df_split


def my_collate(batch):
    return torch.Tensor(batch[0])


def get_len(path):
    _, data = wavfile.read(path)
    len_data = len(data)
    if len(data) > MAX_LEN:
        len_data = MAX_LEN
```

```
        print("File length {} is too long! This file is sliced t
o {}.".format(len(data), MAX_LEN))

    return len_data


def get_wav(path):
    _, snd = wavfile.read(path)
    return snd


def get_mel(wave):
    wave = wave.astype(np.float32) / 32768.0
    data = librosa.feature.melspectrogram(
        wave,
        sr=44100,
        n_mels=128,
        hop_length=347 * 1,
        n_fft=128 * 20,
        fmin=20,
        fmax=44100 // 2,
    ).astype(np.float32)
    return data


def preprocess_mel(args):
    path, slice = args
    wav = get_wav(path)
    mel = get_mel(wav)
    mel_new = np.zeros([mel.shape[0], slice], np.float32)
    if mel.shape[1] > slice:
        print("wav length: {}, mel length: {}".format(wav.shape[
0], mel.shape[1]))
        print("Mel file is sliced")
        mel_new[:] = mel[:, :slice]
    else:
        mel_new[:, :mel.shape[1]] = mel
    mel_new = librosa.power_to_db(mel_new)
    mel_new = mel_new.reshape([1, mel_new.shape[0], mel_new.shap
e[1]])
    return mel_new


def preprocess_wav(args):
    path, slice = args
    wav = get_wav(path)
    pad = (slice - len(wav)) // 2
    wav_new = np.zeros([1, 1, slice], np.int16)

    if wav.shape[0] > slice:
        print("wav length: {}".format(wav.shape[0]))
        print("Wav file is sliced")
        wav_new[0, 0, :] = wav[:slice]
    else:
        wav_new[0, 0, pad:pad + len(wav)] = wav
    return wav_new
```

```python
class BatchDataset(Dataset):
    def __init__(self, df, pad=0):
        self.len_batch = df['batch'].max() - df['batch'].min() + 1
        self.X = np.arange(self.len_batch) + df['batch'].min()
        self.pad = pad

    def __getitem__(self, index):
        batch_base = np.load("{}/{}.npy".format(BATCH_DIR, self.X[index]))
        batch_pad = np.zeros(batch_base.shape[:-1] + (batch_base.shape[-1] + self.pad * 2,), np.float32)
        batch_max = batch_base.max(axis=(1, 2, 3)) - 80
        batch_max = np.maximum(batch_max, -100)
        batch_pad[:] = batch_max[:, np.newaxis, np.newaxis, np.newaxis, ]
        if self.pad != 0:
            batch_pad[:, :, :, self.pad:-self.pad] = batch_base
        else:
            batch_pad[:] = batch_base
        batch_pad = (batch_pad - batch_pad.mean(axis=(1, 2, 3))[:, np.newaxis, np.newaxis, np.newaxis, ]) / (
                    batch_pad.std(axis=(1, 2, 3))[:, np.newaxis, np.newaxis, np.newaxis, ] + 1e-7)
        return batch_pad

    def __len__(self):
        return self.len_batch


class BatchWavDataset(Dataset):
    def __init__(self, batch_list, pad=0):
        self.X = batch_list
        self.pad = pad

    def __getitem__(self, index):
        batch_base = self.X[index]
        if batch_base.shape[-1] + self.pad * 2 < 20580:
            pad = int(np.ceil((20580 - batch_base.shape[-1]) / 2))
        else:
            pad = self.pad
        if pad != 0:
            batch = np.zeros([batch_base.shape[0], 1, 1, batch_base.shape[-1] + pad * 2], np.float32)
            batch[:, :, :, pad:-pad] = batch_base.astype(np.float32) / 32768.0
        else:
            batch = batch_base.astype(np.float32) / 32768.0
        return batch

    def __len__(self):
        return len(self.X)
```

```
def predict_resnet(model, dataloader):
    sigmoid = torch.nn.Sigmoid().cuda()
    preds = []
    model.eval()
    for i, input in enumerate(dataloader):
        input = input.cuda(async=True)
        with torch.no_grad():
            pred = sigmoid(model(input)).data.cpu().numpy()
        preds.append(pred)
    preds = np.concatenate(preds)
    return preds


def predict_envnet(model, dataloader, activation='sigmoid'):
    sigmoid = torch.nn.Sigmoid().cuda()
    softmax = torch.nn.Softmax(dim=1).cuda()
    if activation == 'sigmoid':
        f_act = sigmoid
    elif activation == 'softmax':
        f_act = softmax
    preds = []
    model.eval()
    for i, input in enumerate(dataloader):
        input = input.cuda(async=True)
        with torch.no_grad():
            pred = f_act(model(input)).data.cpu().numpy()
        preds.append(pred)
    preds = np.concatenate(preds)
    return preds


if __name__ == '__main__':
    main()import torch
import torch.nn as nn
from torch.utils.data.dataset import Dataset
from torch.utils.data import DataLoader
import torch.optim as optim
import torch.nn.functional as F

import pretrainedmodels


class ResNet(nn.Module):
    def __init__(self, num_classes=2):
        super(ResNet, self).__init__()

        self.num_classes = num_classes
        self.mode = 'train'

        self.base_model = pretrainedmodels.__dict__['resnet34'](
num_classes=num_classes, pretrained=None)

        self.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2, p
adding=3,
                               bias=False)
```

```
        self.bn1 = self.base_model.bn1
        self.relu = self.base_model.relu
        self.maxpool = self.base_model.maxpool
        self.layer1 = self.base_model.layer1
        self.layer2 = self.base_model.layer2
        self.layer3 = self.base_model.layer3
        self.layer4 = self.base_model.layer4
        self.gmp = nn.AdaptiveMaxPool2d((1, 1))
        self.last_linear = nn.Linear(self.base_model.layer4[1].c
onv1.in_channels, num_classes)
        self.last_linear = nn.Sequential(
            nn.Linear(self.base_model.layer4[1].conv1.in_channel
s, 1024),
            nn.ReLU(),
            nn.Dropout(p=0.2),
            nn.Linear(1024, 1024),
            nn.ReLU(),
            nn.Dropout(p=0.1),
            nn.Linear(1024, num_classes),
        )
        self.last_linear2 = nn.Sequential(
            nn.Linear(self.base_model.layer4[1].conv1.in_channel
s, 1024),
            nn.ReLU(),
            nn.Dropout(p=0.2),
            nn.Linear(1024, 1024),
            nn.ReLU(),
            nn.Dropout(p=0.1),
            nn.Linear(1024, num_classes),
        )

    def forward(self, input):
        bs, ch, h, w = input.size()
        x0 = self.conv1(input)
        x0 = self.bn1(x0)
        x0 = self.relu(x0)
        x1 = self.maxpool(x0)
        x1 = self.layer1(x1)
        x2 = self.layer2(x1)
        x3 = self.layer3(x2)
        x4 = self.layer4(x3)
        x = self.gmp(x4).view(bs, -1)
        x = self.last_linear(x)

        return x

    def noisy(self, input):
        bs, ch, h, w = input.size()
        x0 = self.conv1(input)
        x0 = self.bn1(x0)
        x0 = self.relu(x0)
        x1 = self.maxpool(x0)
        x1 = self.layer1(x1)
        x2 = self.layer2(x1)
        x3 = self.layer3(x2)
        x4 = self.layer4(x3)
```

```
        x = self.gmp(x4).view(bs, -1)
        x = self.last_linear2(x)

        return x



class ConvBnRelu(nn.Module):
    def __init__(self, in_channel, out_channel, kernel_size, str
ide=1, padding=0, dilation=1,
                 groups=1):
        super(ConvBnRelu, self).__init__()
        self.conv_bn_relu = nn.Sequential(
            nn.Conv2d(in_channel, out_channel, kernel_size, stri
de, padding, dilation, groups,
                      False),
            nn.BatchNorm2d(out_channel),
            nn.ReLU(True))

    def forward(self, x):
        return self.conv_bn_relu(x)


class Flatten(nn.Module):
    def forward(self, x):
        return x.view(x.size()[0], -1)


class EnvNetv2(nn.Module):
    def __init__(self, num_classes=1):
        super(EnvNetv2, self).__init__()
        self.conv1 = ConvBnRelu(1, 32, (1, 64), stride=(1, 2))
        self.conv2 = ConvBnRelu(32, 64, (1, 16), stride=(1, 2))
        self.conv3 = ConvBnRelu(1, 32, (8, 8))
        self.conv4 = ConvBnRelu(32, 32, (8, 8))
        self.conv5 = ConvBnRelu(32, 64, (1, 4))
        self.conv6 = ConvBnRelu(64, 64, (1, 4))
        self.conv7 = ConvBnRelu(64, 128, (1, 2))
        self.conv8 = ConvBnRelu(128, 128, (1, 2))
        self.conv9 = ConvBnRelu(128, 256, (1, 2))
        self.conv10 = ConvBnRelu(256, 256, (1, 2))
        self.maxpool1 = nn.MaxPool2d((1, 64), stride=(1, 64))
        self.maxpool2 = nn.MaxPool2d((5, 3), stride=(5, 3))
        self.maxpool3 = nn.MaxPool2d((1, 2), stride=(1, 2))
        self.gmp = nn.AdaptiveMaxPool2d((10, 1))
        self.flatten = Flatten()
        self.last_linear1 = nn.Sequential(
            nn.Linear(256 * 10, 1024),
            nn.ReLU(),
            nn.Dropout(p=0.2),
            nn.Linear(1024, 1024),
            nn.ReLU(),
            nn.Dropout(p=0.1),
            nn.Linear(1024, num_classes),
        )
        self.last_linear2 = nn.Sequential(
```

```
            nn.Linear(256 * 10, 1024),
            nn.ReLU(),
            nn.Dropout(p=0.2),
            nn.Linear(1024, 1024),
            nn.ReLU(),
            nn.Dropout(p=0.1),
            nn.Linear(1024, num_classes),
        )

    def forward(self, input):
        h = self.conv1(input)
        h = self.conv2(h)
        h = self.maxpool1(h)
        h = h.transpose(1, 2)
        h = self.conv3(h)
        h = self.conv4(h)
        h = self.maxpool2(h)
        h = self.conv5(h)
        h = self.conv6(h)
        h = self.maxpool3(h)
        h = self.conv7(h)
        h = self.conv8(h)
        h = self.maxpool3(h)
        h = self.conv9(h)
        h = self.conv10(h)
        h = self.gmp(h)
        h = self.flatten(h)
        h = self.last_linear1(h)
        return h

    def noisy(self, input):
        h = self.conv1(input)
        h = self.conv2(h)
        h = self.maxpool1(h)
        h = h.transpose(1, 2)
        h = self.conv3(h)
        h = self.conv4(h)
        h = self.maxpool2(h)
        h = self.conv5(h)
        h = self.conv6(h)
        h = self.maxpool3(h)
        h = self.conv7(h)
        h = self.conv8(h)
        h = self.maxpool3(h)
        h = self.conv9(h)
        h = self.conv10(h)
        h = self.gmp(h)
        h = self.flatten(h)
        h = self.last_linear2(h)
        return himport numpy as np
import pandas as pd
import time
import librosa

# parameters
SAMPLE_RATE = 44100
```

```
N_MELS = 128
HOP_LENGTH = 347
N_FFT = 128*20
FMIN = 20
FMAX = SAMPLE_RATE//2

starttime = time.time()


def convert(df, input_dir, output_dir):
    for i in range(len(df)):
        if (i+1)%100==0: print("{}/{}, sec: {:.1f}".format(i+1,
len(df), time.time()-starttime))
        file_path = "{}/{}".format(input_dir, df['fname'][i])
        data, _ = librosa.core.load(file_path, sr=SAMPLE_RATE, r
es_type="kaiser_fast")
        data = librosa.feature.melspectrogram(
            data,
            sr=SAMPLE_RATE,
            n_mels=N_MELS,
            hop_length=HOP_LENGTH, # 1sec -> 128
            n_fft=N_FFT,
            fmin=FMIN,
            fmax=FMAX,
        ).astype(np.float32)
        np.save("{}/{}.npy".format(output_dir, df['fname'][i][:-
4]), data)


def main():
    # load table data
    df_train = pd.read_csv("../input/train_curated.csv")
    df_noisy = pd.read_csv("../input/train_noisy.csv")
    df_test = pd.read_csv("../input/sample_submission.csv")

    # convert to logmel
    print("converting train data...")
    convert(df_train, "../input/train_curated/", "../input/mel12
8/train")
    print("converting noisy data...")
    convert(df_noisy, "../input/train_noisy/", "../input/mel128/
noisy")
    print("converting test data...")
    convert(df_test, "../input/test/", "../input/mel128/test")


if __name__ == '__main__':
    main()import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time, random, sys
import sklearn.metrics
from sklearn.model_selection import KFold

import torch
import torch.nn as nn
```

```
import torch.backends.cudnn as cudnn
import torch.optim as optim
from torch.utils.data import DataLoader

sys.path.append('.')
from utils import AverageMeter, cycle, CosineLR, MelDataset, cal
culate_per_class_lwlrap
from models import ResNet

# set parameters
NUM_FOLD = 5
NUM_CLASS = 80
SEED = 42
NUM_EPOCH = 64*8
NUM_CYCLE = 64
BATCH_SIZE = 64
LR = [1e-3, 1e-6]
FOLD_LIST = [1, 2, 3, 4, 5]
CROP_LENGTH = 512
OUTPUT_DIR = "../models/resnet_model1"

cudnn.benchmark = True
starttime = time.time()


def main():
    # load table data
    df_train = pd.read_csv("../input/train_curated.csv")
    df_noisy = pd.read_csv("../input/train_noisy.csv")
    df_test = pd.read_csv("../input/sample_submission.csv")
    labels = df_test.columns[1:].tolist()
    for label in labels:
        df_train[label] = df_train['labels'].apply(lambda x: lab
el in x)
        df_noisy[label] = df_noisy['labels'].apply(lambda x: lab
el in x)

    df_train['path'] = "../input/mel128/train/" + df_train['fnam
e']
    df_test['path'] = "../input/mel128/test/" + df_train['fname'
]
    df_noisy['path'] = "../input/mel128/noisy/" + df_noisy['fnam
e']

    # fold splitting
    folds = list(KFold(n_splits=NUM_FOLD, shuffle=True, random_s
tate=SEED).split(np.arange(len(df_train))))

    # Training
    log_columns = ['epoch', 'bce', 'lwlrap', 'bce_noisy', 'lwlra
p_noisy', 'val_bce', 'val_lwlrap', 'time']
    for fold, (ids_train_split, ids_valid_split) in enumerate(fo
lds):
        if fold+1 not in FOLD_LIST: continue
        print("fold: {}".format(fold + 1))
        train_log = pd.DataFrame(columns=log_columns)
```

```
        # build model
        model = ResNet(NUM_CLASS).cuda()

        # prepare data loaders
        df_train_fold = df_train.iloc[ids_train_split].reset_ind
ex(drop=True)
        dataset_train = MelDataset(df_train_fold['path'], df_tra
in_fold[labels].values,
                                    crop=CROP_LENGTH, crop_mode=
'random',
                                    mixup=True, freqmask=True, g
ain=True,
                                    )
        train_loader = DataLoader(dataset_train, batch_size=BATC
H_SIZE,
                                  shuffle=True, num_workers=1, p
in_memory=True,
                                  )

        df_valid = df_train.iloc[ids_valid_split].reset_index(dr
op=True)
        dataset_valid = MelDataset(df_valid['path'], df_valid[la
bels].values,)
        valid_loader = DataLoader(dataset_valid, batch_size=1,
                                  shuffle=False, num_workers=1,
pin_memory=True,
                                  )

        dataset_noisy = MelDataset(df_noisy['path'], df_noisy[la
bels].values,
                                    crop=CROP_LENGTH, crop_mode=
'random',
                                    mixup=True, freqmask=True, g
ain=True,
                                   )
        noisy_loader = DataLoader(dataset_noisy, batch_size=BATC
H_SIZE,
                                  shuffle=True, num_workers=1, p
in_memory=True,
                                  )
        noisy_itr = cycle(noisy_loader)

        # set optimizer and loss
        optimizer = optim.Adam(filter(lambda p: p.requires_grad,
 model.parameters()), lr=LR[0])
        scheduler = CosineLR(optimizer, step_size_min=LR[1], t0=
len(train_loader) * NUM_CYCLE, tmult=1)

        # training
        for epoch in range(NUM_EPOCH):
            # train for one epoch
            bce, lwlrap, bce_noisy, lwlrap_noisy = train((train_
loader, noisy_itr), model, optimizer, scheduler, epoch)

            # evaluate on validation set
```

```
            val_bce, val_lwlrap = validate(valid_loader, model)

            # print log
            endtime = time.time() - starttime
            print("Epoch: {}/{} ".format(epoch + 1, NUM_EPOCH)
                  + "CE: {:.4f} ".format(bce)
                  + "LwLRAP: {:.4f} ".format(lwlrap)
                  + "Noisy CE: {:.4f} ".format(bce_noisy)
                  + "Noisy LWLRAP: {:.4f} ".format(lwlrap_noisy)
                  + "Valid CE: {:.4f} ".format(val_bce)
                  + "Valid LWLRAP: {:.4f} ".format(val_lwlrap)
                  + "sec: {:.1f}".format(endtime)
                  )

            # save log and weights
            train_log_epoch = pd.DataFrame(
                [[epoch+1, bce, lwlrap, bce_noisy, lwlrap_noisy,
 val_bce, val_lwlrap, endtime]],
                columns=log_columns)
            train_log = pd.concat([train_log, train_log_epoch])
            train_log.to_csv("{}/train_log_fold{}.csv".format(OU
TPUT_DIR, fold+1), index=False)
            if (epoch+1)%NUM_CYCLE==0:
                torch.save(model.state_dict(), "{}/weight_fold_{
}_epoch_{}.pth".format(OUTPUT_DIR, fold+1, epoch+1))


def train(train_loaders, model, optimizer, scheduler, epoch):
    train_loader, noisy_itr = train_loaders
    bce_avr = AverageMeter()
    bce_noisy_avr = AverageMeter()
    criterion_bce = nn.BCEWithLogitsLoss().cuda()
    sigmoid = nn.Sigmoid().cuda()

    # switch to train mode
    model.train()

    # training
    preds = np.zeros([0, NUM_CLASS], np.float32)
    y_true = np.zeros([0, NUM_CLASS], np.float32)
    preds_noisy = np.zeros([0, NUM_CLASS], np.float32)
    y_true_noisy = np.zeros([0, NUM_CLASS], np.float32)
    for i, (input, target) in enumerate(train_loader):
        # get batches
        input = torch.autograd.Variable(input.cuda())
        target = torch.autograd.Variable(target.cuda())

        input_noisy, target_noisy = next(noisy_itr)
        input_noisy = torch.autograd.Variable(input_noisy.cuda()
)
        target_noisy = torch.autograd.Variable(target_noisy.cuda
())

        # compute output
        output = model(input)
        bce = criterion_bce(output, target)
```

```
        output_noisy = model.noisy(input_noisy)
        bce_noisy = criterion_bce(sigmoid(output_noisy), target_
noisy)
        loss = bce + bce_noisy
        pred = sigmoid(output)
        pred = pred.data.cpu().numpy()
        pred_noisy = sigmoid(output_noisy)
        pred_noisy = pred_noisy.data.cpu().numpy()

        # backprop
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step()

        # record log
        bce_avr.update(bce.data, input.size(0))
        bce_noisy_avr.update(bce_noisy.data, input.size(0))
        preds = np.concatenate([preds, pred])
        y_true = np.concatenate([y_true, target.data.cpu().numpy
()])
        preds_noisy = np.concatenate([preds_noisy, pred_noisy])
        y_true_noisy = np.concatenate([y_true_noisy, target_nois
y.data.cpu().numpy()])

    # calc metric
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true, preds)
    lwlrap = np.sum(per_class_lwlrap * weight_per_class)
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true_noisy, preds_noisy)
    lwlrap_noisy = np.sum(per_class_lwlrap * weight_per_class)

    return bce_avr.avg.item(), lwlrap, bce_noisy_avr.avg.item(),
 lwlrap_noisy


def validate(val_loader, model):
    bce_avr = AverageMeter()
    sigmoid = torch.nn.Sigmoid().cuda()
    criterion_bce = nn.BCEWithLogitsLoss().cuda()

    # switch to eval mode
    model.eval()

    # validate
    preds = np.zeros([0, NUM_CLASS], np.float32)
    y_true = np.zeros([0, NUM_CLASS], np.float32)
    for i, (input, target) in enumerate(val_loader):
        # get batches
        input = torch.autograd.Variable(input.cuda())
        target = torch.autograd.Variable(target.cuda())

        # compute output
        with torch.no_grad():
            output = model(input)
```

```
            bce = criterion_bce(output, target)
            pred = sigmoid(output)
            pred = pred.data.cpu().numpy()

        # record log
        bce_avr.update(bce.data, input.size(0))
        preds = np.concatenate([preds, pred])
        y_true = np.concatenate([y_true, target.data.cpu().numpy
()])

    # calc metric
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true, preds)
    lwlrap = np.sum(per_class_lwlrap * weight_per_class)

    return bce_avr.avg.item(), lwlrap


if __name__ == '__main__':
    main()import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time, random, sys
import sklearn.metrics
from sklearn.model_selection import KFold

import torch
import torch.nn as nn
import torch.backends.cudnn as cudnn
import torch.optim as optim
from torch.utils.data import DataLoader

sys.path.append('.')
from utils import AverageMeter, cycle, CosineLR, MelDataset, cal
culate_per_class_lwlrap
from models import ResNet

# set parameters
NUM_FOLD = 5
NUM_CLASS = 80
SEED = 42
NUM_EPOCH = 64*7
NUM_CYCLE = 64
BATCH_SIZE = 64
LR = [1e-3, 1e-6]
FOLD_LIST = [1, 2, 3, 4, 5]
CROP_LENGTH = 512
C_SEMI = 20
TEMPERATURE = 2
CROP_RATE = 0.25
LOAD_DIR = "../models/resnet_model1"
OUTPUT_DIR = "../models/resnet_model2"

cudnn.benchmark = True
starttime = time.time()
```

```
def main():
    # load table data
    df_train = pd.read_csv("../input/train_curated.csv")
    df_noisy = pd.read_csv("../input/train_noisy.csv")
    df_test = pd.read_csv("../input/sample_submission.csv")
    labels = df_test.columns[1:].tolist()
    for label in labels:
        df_train[label] = df_train['labels'].apply(lambda x: lab
el in x)
        df_noisy[label] = df_noisy['labels'].apply(lambda x: lab
el in x)

    df_train['path'] = "../input/mel128/train/" + df_train['fnam
e']
    df_test['path'] = "../input/mel128/test/" + df_train['fname'
]
    df_noisy['path'] = "../input/mel128/noisy/" + df_noisy['fnam
e']

    # calc sampling weight
    df_train['weight'] = 1
    df_noisy['weight'] = len(df_train) / len(df_noisy)

    # generate pseudo label with sharpening
    tmp = np.load("../input/pseudo_label/preds_noisy.npy").mean(
axis=(0,1))
    tmp = tmp ** TEMPERATURE
    tmp = tmp / tmp.sum(axis=1)[:, np.newaxis]
    df_noisy_pseudo = df_noisy.copy()
    df_noisy_pseudo[labels] = tmp

    # fold splitting
    folds = list(KFold(n_splits=NUM_FOLD, shuffle=True, random_s
tate=SEED).split(np.arange(len(df_train))))
    folds_noisy = list(KFold(n_splits=NUM_FOLD, shuffle=True, ra
ndom_state=SEED).split(np.arange(len(df_noisy))))

    # Training
    log_columns = ['epoch', 'bce', 'lwlrap', 'bce_noisy', 'lwlra
p_noisy', 'semi_mse', 'val_bce', 'val_lwlrap', 'time']
    for fold, (ids_train_split, ids_valid_split) in enumerate(fo
lds):
        if fold+1 not in FOLD_LIST: continue
        print("fold: {}".format(fold + 1))
        train_log = pd.DataFrame(columns=log_columns)

        # build model
        model = ResNet(NUM_CLASS).cuda()
        model.load_state_dict(torch.load("{}/weight_fold_{}_epoc
h_512.pth".format(LOAD_DIR, fold+1)))

        # prepare data loaders
        df_train_fold = df_train.iloc[ids_train_split].reset_ind
ex(drop=True)
        dataset_train = MelDataset(df_train_fold['path'], df_tra
```

```
in_fold[labels].values,
                                    crop=CROP_LENGTH, crop_mode=
'additional', crop_rate=CROP_RATE,
                                    mixup=True, freqmask=True, g
ain=True,
                                    )
        train_loader = DataLoader(dataset_train, batch_size=BATC
H_SIZE,
                                  shuffle=True, num_workers=1, p
in_memory=True,
                                  )

        df_valid = df_train.iloc[ids_valid_split].reset_index(dr
op=True)
        dataset_valid = MelDataset(df_valid['path'], df_valid[la
bels].values,)
        valid_loader = DataLoader(dataset_valid, batch_size=1,
                                  shuffle=False, num_workers=1,
pin_memory=True,
                                  )

        dataset_noisy = MelDataset(df_noisy['path'], df_noisy[la
bels].values,
                                    crop=CROP_LENGTH, crop_mode=
'additional', crop_rate=CROP_RATE,
                                    mixup=True, freqmask=True, g
ain=True,
                                   )
        noisy_loader = DataLoader(dataset_noisy, batch_size=BATC
H_SIZE,
                                  shuffle=True, num_workers=1, p
in_memory=True,
                                  )
        noisy_itr = cycle(noisy_loader)

        df_semi = pd.concat([df_train.iloc[ids_train_split], df_
noisy_pseudo.iloc[folds_noisy[fold][0]]]).reset_index(drop=True)
        semi_sampler = torch.utils.data.sampler.WeightedRandomSa
mpler(df_semi['weight'].values, len(df_semi))
        dataset_semi = MelDataset(df_semi['path'], df_semi[label
s].values,
                                   crop=CROP_LENGTH, crop_mode='
additional', crop_rate=CROP_RATE,
                                   mixup=True, freqmask=True, ga
in=True,
                                   )
        semi_loader = DataLoader(dataset_semi,
                                 batch_size=BATCH_SIZE,
                                 shuffle=False, num_workers=1, p
in_memory=True,
                                 sampler=semi_sampler,
                                 )
        semi_itr = cycle(semi_loader)

        # set optimizer and loss
        optimizer = optim.Adam(filter(lambda p: p.requires_grad,
```

```
 model.parameters()), lr=LR[0])
        scheduler = CosineLR(optimizer, step_size_min=LR[1], t0=
len(train_loader) * NUM_CYCLE, tmult=1)

        # training
        for epoch in range(NUM_EPOCH):
            # train for one epoch
            bce, lwlrap, bce_noisy, lwlrap_noisy, mse_semi = tra
in((train_loader, noisy_itr, semi_itr), model, optimizer, schedu
ler, epoch)

            # evaluate on validation set
            val_bce, val_lwlrap = validate(valid_loader, model)

            # print log
            endtime = time.time() - starttime
            print("Epoch: {}/{} ".format(epoch + 1, NUM_EPOCH)
                  + "CE: {:.4f} ".format(bce)
                  + "LwLRAP: {:.4f} ".format(lwlrap)
                  + "Noisy CE: {:.4f} ".format(bce_noisy)
                  + "Noisy LWLRAP: {:.4f} ".format(lwlrap_noisy)
                  + "Semi MSE: {:.4f} ".format(mse_semi)
                  + "Valid CE: {:.4f} ".format(val_bce)
                  + "Valid LWLRAP: {:.4f} ".format(val_lwlrap)
                  + "sec: {:.1f}".format(endtime)
                  )

            # save log and weights
            train_log_epoch = pd.DataFrame(
                [[epoch+1, bce, lwlrap, bce_noisy, lwlrap_noisy,
 mse_semi, val_bce, val_lwlrap, endtime]],
                columns=log_columns)
            train_log = pd.concat([train_log, train_log_epoch])
            train_log.to_csv("{}/train_log_fold{}.csv".format(OU
TPUT_DIR, fold+1), index=False)
            if (epoch+1)%NUM_CYCLE==0:
                torch.save(model.state_dict(), "{}/weight_fold_{
}_epoch_{}.pth".format(OUTPUT_DIR, fold+1, epoch+1))


def train(train_loaders, model, optimizer, scheduler, epoch):
    train_loader, noisy_itr, semi_itr = train_loaders
    bce_avr = AverageMeter()
    bce_noisy_avr = AverageMeter()
    mse_semi_avr = AverageMeter()
    criterion_bce = nn.BCEWithLogitsLoss().cuda()
    criterion_mse = nn.MSELoss().cuda()
    sigmoid = nn.Sigmoid().cuda()

    # switch to train mode
    model.train()

    # training
    preds = np.zeros([0, NUM_CLASS], np.float32)
    y_true = np.zeros([0, NUM_CLASS], np.float32)
    preds_noisy = np.zeros([0, NUM_CLASS], np.float32)
```

```
    y_true_noisy = np.zeros([0, NUM_CLASS], np.float32)
    for i, (input, target) in enumerate(train_loader):
        # get batches
        input = torch.autograd.Variable(input.cuda())
        target = torch.autograd.Variable(target.cuda())

        input_noisy, target_noisy = next(noisy_itr)
        input_noisy = torch.autograd.Variable(input_noisy.cuda()
)
        target_noisy = torch.autograd.Variable(target_noisy.cuda
())

        input_semi, target_semi = next(semi_itr)
        input_semi = torch.autograd.Variable(input_semi.cuda())
        target_semi = torch.autograd.Variable(target_semi.cuda()
)

        # compute output
        output = model(input)
        bce = criterion_bce(output, target)
        output_noisy = model.noisy(input_noisy)
        bce_noisy = criterion_bce(sigmoid(output_noisy), target_
noisy)
        output_semi = model(input_semi)
        mse_semi = criterion_mse(sigmoid(output_semi), target_se
mi)

        loss = bce + bce_noisy + C_SEMI * mse_semi
        pred = sigmoid(output)
        pred = pred.data.cpu().numpy()
        pred_noisy = sigmoid(output_noisy)
        pred_noisy = pred_noisy.data.cpu().numpy()

        # backprop
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step()

        # record log
        bce_avr.update(bce.data, input.size(0))
        bce_noisy_avr.update(bce_noisy.data, input.size(0))
        mse_semi_avr.update(mse_semi.data, input.size(0))
        preds = np.concatenate([preds, pred])
        y_true = np.concatenate([y_true, target.data.cpu().numpy
()])
        preds_noisy = np.concatenate([preds_noisy, pred_noisy])
        y_true_noisy = np.concatenate([y_true_noisy, target_nois
y.data.cpu().numpy()])

    # calc metric
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true, preds)
    lwlrap = np.sum(per_class_lwlrap * weight_per_class)
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true_noisy, preds_noisy)
```

```
    lwlrap_noisy = np.sum(per_class_lwlrap * weight_per_class)

    return bce_avr.avg.item(), lwlrap, bce_noisy_avr.avg.item(),
 lwlrap_noisy, mse_semi_avr.avg.item()


def validate(val_loader, model):
    bce_avr = AverageMeter()
    sigmoid = torch.nn.Sigmoid().cuda()
    criterion_bce = nn.BCEWithLogitsLoss().cuda()

    # switch to eval mode
    model.eval()

    # validate
    preds = np.zeros([0, NUM_CLASS], np.float32)
    y_true = np.zeros([0, NUM_CLASS], np.float32)
    for i, (input, target) in enumerate(val_loader):
        # get batches
        input = torch.autograd.Variable(input.cuda())
        target = torch.autograd.Variable(target.cuda())

        # compute output
        with torch.no_grad():
            output = model(input)
            bce = criterion_bce(output, target)
            pred = sigmoid(output)
            pred = pred.data.cpu().numpy()

        # record log
        bce_avr.update(bce.data, input.size(0))
        preds = np.concatenate([preds, pred])
        y_true = np.concatenate([y_true, target.data.cpu().numpy
()])

    # calc metric
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true, preds)
    lwlrap = np.sum(per_class_lwlrap * weight_per_class)

    return bce_avr.avg.item(), lwlrap


if __name__ == '__main__':
    main()import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time, random, sys
import sklearn.metrics
from sklearn.model_selection import KFold

import torch
import torch.nn as nn
import torch.backends.cudnn as cudnn
import torch.optim as optim
from torch.utils.data import DataLoader
```

```
sys.path.append('.')
from utils import AverageMeter, cycle, CosineLR, MelDataset, cal
culate_per_class_lwlrap
from models import ResNet

# set parameters
NUM_FOLD = 5
NUM_CLASS = 80
SEED = 42
NUM_EPOCH = 64*8
NUM_CYCLE = 64
BATCH_SIZE = 64
LR = [1e-3, 1e-6]
FOLD_LIST = [1, 2, 3, 4, 5]
CROP_LENGTH = 1024
OUTPUT_DIR = "../models/resnet_model3"

cudnn.benchmark = True
starttime = time.time()


def main():
    # load table data
    df_train = pd.read_csv("../input/train_curated.csv")
    df_noisy = pd.read_csv("../input/train_noisy.csv")
    df_test = pd.read_csv("../input/sample_submission.csv")
    labels = df_test.columns[1:].tolist()
    for label in labels:
        df_train[label] = df_train['labels'].apply(lambda x: lab
el in x)
        df_noisy[label] = df_noisy['labels'].apply(lambda x: lab
el in x)

    df_train['path'] = "../input/mel128/train/" + df_train['fnam
e']
    df_test['path'] = "../input/mel128/test/" + df_train['fname'
]
    df_noisy['path'] = "../input/mel128/noisy/" + df_noisy['fnam
e']

    # fold splitting
    folds = list(KFold(n_splits=NUM_FOLD, shuffle=True, random_s
tate=SEED).split(np.arange(len(df_train))))

    # Training
    log_columns = ['epoch', 'bce', 'lwlrap', 'bce_noisy', 'lwlra
p_noisy', 'val_bce', 'val_lwlrap', 'time']
    for fold, (ids_train_split, ids_valid_split) in enumerate(fo
lds):
        if fold+1 not in FOLD_LIST: continue
        print("fold: {}".format(fold + 1))
        train_log = pd.DataFrame(columns=log_columns)

        # build model
        model = ResNet(NUM_CLASS).cuda()
```

```
        # prepare data loaders
        df_train_fold = df_train.iloc[ids_train_split].reset_ind
ex(drop=True)
        dataset_train = MelDataset(df_train_fold['path'], df_tra
in_fold[labels].values,
                                    crop=CROP_LENGTH, crop_mode=
'random',
                                    mixup=True, freqmask=True, g
ain=True,
                                    )
        train_loader = DataLoader(dataset_train, batch_size=BATC
H_SIZE,
                                  shuffle=True, num_workers=1, p
in_memory=True,
                                  )

        df_valid = df_train.iloc[ids_valid_split].reset_index(dr
op=True)
        dataset_valid = MelDataset(df_valid['path'], df_valid[la
bels].values,)
        valid_loader = DataLoader(dataset_valid, batch_size=1,
                                  shuffle=False, num_workers=1,
pin_memory=True,
                                  )

        dataset_noisy = MelDataset(df_noisy['path'], df_noisy[la
bels].values,
                                    crop=CROP_LENGTH, crop_mode=
'random',
                                    mixup=True, freqmask=True, g
ain=True,
                                   )
        noisy_loader = DataLoader(dataset_noisy, batch_size=BATC
H_SIZE,
                                  shuffle=True, num_workers=1, p
in_memory=True,
                                  )
        noisy_itr = cycle(noisy_loader)

        # set optimizer and loss
        optimizer = optim.Adam(filter(lambda p: p.requires_grad,
 model.parameters()), lr=LR[0])
        scheduler = CosineLR(optimizer, step_size_min=LR[1], t0=
len(train_loader) * NUM_CYCLE, tmult=1)

        # training
        for epoch in range(NUM_EPOCH):
            # train for one epoch
            bce, lwlrap, bce_noisy, lwlrap_noisy = train((train_
loader, noisy_itr), model, optimizer, scheduler, epoch)

            # evaluate on validation set
            val_bce, val_lwlrap = validate(valid_loader, model)

            # print log
```

```
            endtime = time.time() - starttime
            print("Epoch: {}/{} ".format(epoch + 1, NUM_EPOCH)
                  + "CE: {:.4f} ".format(bce)
                  + "LwLRAP: {:.4f} ".format(lwlrap)
                  + "Noisy CE: {:.4f} ".format(bce_noisy)
                  + "Noisy LWLRAP: {:.4f} ".format(lwlrap_noisy)
                  + "Valid CE: {:.4f} ".format(val_bce)
                  + "Valid LWLRAP: {:.4f} ".format(val_lwlrap)
                  + "sec: {:.1f}".format(endtime)
                  )

            # save log and weights
            train_log_epoch = pd.DataFrame(
                [[epoch+1, bce, lwlrap, bce_noisy, lwlrap_noisy,
 val_bce, val_lwlrap, endtime]],
                columns=log_columns)
            train_log = pd.concat([train_log, train_log_epoch])
            train_log.to_csv("{}/train_log_fold{}.csv".format(OU
TPUT_DIR, fold+1), index=False)
            if (epoch+1)%NUM_CYCLE==0:
                torch.save(model.state_dict(), "{}/weight_fold_{
}_epoch_{}.pth".format(OUTPUT_DIR, fold+1, epoch+1))


def train(train_loaders, model, optimizer, scheduler, epoch):
    train_loader, noisy_itr = train_loaders
    bce_avr = AverageMeter()
    bce_noisy_avr = AverageMeter()
    criterion_bce = nn.BCEWithLogitsLoss().cuda()
    sigmoid = nn.Sigmoid().cuda()

    # switch to train mode
    model.train()

    # training
    preds = np.zeros([0, NUM_CLASS], np.float32)
    y_true = np.zeros([0, NUM_CLASS], np.float32)
    preds_noisy = np.zeros([0, NUM_CLASS], np.float32)
    y_true_noisy = np.zeros([0, NUM_CLASS], np.float32)
    for i, (input, target) in enumerate(train_loader):
        # get batches
        input = torch.autograd.Variable(input.cuda())
        target = torch.autograd.Variable(target.cuda())

        input_noisy, target_noisy = next(noisy_itr)
        input_noisy = torch.autograd.Variable(input_noisy.cuda()
)
        target_noisy = torch.autograd.Variable(target_noisy.cuda
())

        # compute output
        output = model(input)
        bce = criterion_bce(output, target)
        output_noisy = model.noisy(input_noisy)
        bce_noisy = criterion_bce(sigmoid(output_noisy), target_
noisy)
```

```
        loss = bce + bce_noisy
        pred = sigmoid(output)
        pred = pred.data.cpu().numpy()
        pred_noisy = sigmoid(output_noisy)
        pred_noisy = pred_noisy.data.cpu().numpy()

        # backprop
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step()

        # record log
        bce_avr.update(bce.data, input.size(0))
        bce_noisy_avr.update(bce_noisy.data, input.size(0))
        preds = np.concatenate([preds, pred])
        y_true = np.concatenate([y_true, target.data.cpu().numpy
()])
        preds_noisy = np.concatenate([preds_noisy, pred_noisy])
        y_true_noisy = np.concatenate([y_true_noisy, target_nois
y.data.cpu().numpy()])

    # calc metric
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true, preds)
    lwlrap = np.sum(per_class_lwlrap * weight_per_class)
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true_noisy, preds_noisy)
    lwlrap_noisy = np.sum(per_class_lwlrap * weight_per_class)

    return bce_avr.avg.item(), lwlrap, bce_noisy_avr.avg.item(),
 lwlrap_noisy


def validate(val_loader, model):
    bce_avr = AverageMeter()
    sigmoid = torch.nn.Sigmoid().cuda()
    criterion_bce = nn.BCEWithLogitsLoss().cuda()

    # switch to eval mode
    model.eval()

    # validate
    preds = np.zeros([0, NUM_CLASS], np.float32)
    y_true = np.zeros([0, NUM_CLASS], np.float32)
    for i, (input, target) in enumerate(val_loader):
        # get batches
        input = torch.autograd.Variable(input.cuda())
        target = torch.autograd.Variable(target.cuda())

        # compute output
        with torch.no_grad():
            output = model(input)
            bce = criterion_bce(output, target)
            pred = sigmoid(output)
            pred = pred.data.cpu().numpy()
```

```
        # record log
        bce_avr.update(bce.data, input.size(0))
        preds = np.concatenate([preds, pred])
        y_true = np.concatenate([y_true, target.data.cpu().numpy
()])

    # calc metric
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true, preds)
    lwlrap = np.sum(per_class_lwlrap * weight_per_class)

    return bce_avr.avg.item(), lwlrap


if __name__ == '__main__':
    main()import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time, random, sys
import sklearn.metrics
from sklearn.model_selection import KFold

import torch
import torch.nn as nn
import torch.backends.cudnn as cudnn
import torch.optim as optim
from torch.utils.data import DataLoader

sys.path.append('.')
from utils import AverageMeter, cycle, CosineLR, WaveDataset, ca
lculate_per_class_lwlrap
from models import EnvNetv2

# set parameters
NUM_FOLD = 5
NUM_CLASS = 80
SEED = 42
NUM_EPOCH =400*1
NUM_CYCLE = 400
BATCH_SIZE = 16
LR = [1e-1, 1e-6]
FOLD_LIST = [1, 2, 3, 4, 5]
FOLD_LIST = [1, 2,]
CROP_LENGTH = 133300
OUTPUT_DIR = "../models/envnet_model4_0"

cudnn.benchmark = True
starttime = time.time()


def main():
    # load table data
    df_train = pd.read_csv("../input/train_curated.csv")
    df_noisy = pd.read_csv("../input/train_noisy.csv")
    df_test = pd.read_csv("../input/sample_submission.csv")
```

```
    labels = df_test.columns[1:].tolist()
    for label in labels:
        df_train[label] = df_train['labels'].apply(lambda x: lab
el in x)
        df_noisy[label] = df_noisy['labels'].apply(lambda x: lab
el in x)

    df_train['path'] = "../input/train_curated/" + df_train['fna
me']
    df_test['path'] = "../input/test/" + df_train['fname']
    df_noisy['path'] = "../input/train_noisy/" + df_noisy['fname
']

    # fold splitting
    folds = list(KFold(n_splits=NUM_FOLD, shuffle=True, random_s
tate=SEED).split(np.arange(len(df_train))))

    # Training
    log_columns = ['epoch', 'kl', 'lwlrap', 'kl_noisy', 'lwlrap_
noisy', 'val_kl', 'val_lwlrap', 'time']
    for fold, (ids_train_split, ids_valid_split) in enumerate(fo
lds):
        if fold+1 not in FOLD_LIST: continue
        print("fold: {}".format(fold + 1))
        train_log = pd.DataFrame(columns=log_columns)

        # build model
        model = EnvNetv2(NUM_CLASS).cuda()

        # prepare data loaders
        df_train_fold = df_train.iloc[ids_train_split].reset_ind
ex(drop=True)
        dataset_train = WaveDataset(df_train_fold['path'], df_tr
ain_fold[labels].values,
                                    crop=CROP_LENGTH, crop_mode=
'random', padding=CROP_LENGTH//2,
                                    mixup=True, scaling=1.25, ga
in=6,
                                    )
        train_loader = DataLoader(dataset_train, batch_size=BATC
H_SIZE,
                                  shuffle=True, num_workers=1, p
in_memory=True,
                                  )

        df_valid = df_train.iloc[ids_valid_split].reset_index(dr
op=True)
        dataset_valid = WaveDataset(df_valid['path'], df_valid[l
abels].values, padding=CROP_LENGTH//2)
        valid_loader = DataLoader(dataset_valid, batch_size=1,
                                  shuffle=False, num_workers=1,
pin_memory=True,
                                  )

        dataset_noisy = WaveDataset(df_noisy['path'], df_noisy[l
abels].values,
```

```
                                        crop=CROP_LENGTH, crop_mode=
'random', padding=CROP_LENGTH//2,
                                        mixup=True, scaling=1.25, ga
in=6,
                                       )
        noisy_loader = DataLoader(dataset_noisy, batch_size=BATC
H_SIZE,
                                  shuffle=True, num_workers=1, p
in_memory=True,
                                  )
        noisy_itr = cycle(noisy_loader)

        # set optimizer and loss
        optimizer = optim.SGD(filter(lambda p: p.requires_grad,
model.parameters()), lr=LR[0],momentum = 0.9,nesterov = True)
        scheduler = CosineLR(optimizer, step_size_min=LR[1], t0=
len(train_loader) * NUM_CYCLE, tmult=1)

        # training
        for epoch in range(NUM_EPOCH):
            # train for one epoch
            kl, lwlrap, kl_noisy, lwlrap_noisy = train((train_lo
ader, noisy_itr), model, optimizer, scheduler, epoch)

            # evaluate on validation set
            val_kl, val_lwlrap = validate(valid_loader, model)

            # print log
            endtime = time.time() - starttime
            print("Epoch: {}/{} ".format(epoch + 1, NUM_EPOCH)
                  + "KL: {:.4f} ".format(kl)
                  + "LwLRAP: {:.4f} ".format(lwlrap)
                  + "Noisy KL: {:.4f} ".format(kl_noisy)
                  + "Noisy LWLRAP: {:.4f} ".format(lwlrap_noisy)
                  + "Valid KL: {:.4f} ".format(val_kl)
                  + "Valid LWLRAP: {:.4f} ".format(val_lwlrap)
                  + "sec: {:.1f}".format(endtime)
                  )

            # save log and weights
            train_log_epoch = pd.DataFrame(
                [[epoch+1, kl, lwlrap, kl_noisy, lwlrap_noisy, v
al_kl, val_lwlrap, endtime]],
                columns=log_columns)
            train_log = pd.concat([train_log, train_log_epoch])
            train_log.to_csv("{}/train_log_fold{}.csv".format(OU
TPUT_DIR, fold+1), index=False)
            if (epoch+1)%NUM_CYCLE==0:
                torch.save(model.state_dict(), "{}/weight_fold_{
}_epoch_{}.pth".format(OUTPUT_DIR, fold+1, epoch+1))


def train(train_loaders, model, optimizer, scheduler, epoch):
    train_loader, noisy_itr = train_loaders
    kl_avr = AverageMeter()
    kl_noisy_avr = AverageMeter()
```

```
    lsigmoid = nn.LogSigmoid().cuda()
    lsoftmax = nn.LogSoftmax(dim=1).cuda()
    softmax = nn.Softmax(dim=1).cuda()
    criterion_kl = nn.KLDivLoss().cuda()

    # switch to train mode
    model.train()

    # training
    preds = np.zeros([0, NUM_CLASS], np.float32)
    y_true = np.zeros([0, NUM_CLASS], np.float32)
    preds_noisy = np.zeros([0, NUM_CLASS], np.float32)
    y_true_noisy = np.zeros([0, NUM_CLASS], np.float32)
    for i, (input, target) in enumerate(train_loader):
        # get batches
        input = torch.autograd.Variable(input.cuda())
        target = torch.autograd.Variable(target.cuda())

        input_noisy, target_noisy = next(noisy_itr)
        input_noisy = torch.autograd.Variable(input_noisy.cuda()
)
        target_noisy = torch.autograd.Variable(target_noisy.cuda
())

        # compute output
        output = model(input)
        kl = criterion_kl(lsoftmax(output), target)
        output_noisy = model.noisy(input_noisy)
        kl_noisy = criterion_kl(lsoftmax(output_noisy), target_n
oisy)
        loss = kl + kl_noisy
        pred = softmax(output)
        pred = pred.data.cpu().numpy()
        pred_noisy = softmax(output_noisy)
        pred_noisy = pred_noisy.data.cpu().numpy()

        # backprop
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step()

        # record log
        kl_avr.update(kl.data, input.size(0))
        kl_noisy_avr.update(kl_noisy.data, input.size(0))
        preds = np.concatenate([preds, pred])
        y_true = np.concatenate([y_true, target.data.cpu().numpy
()])
        preds_noisy = np.concatenate([preds_noisy, pred_noisy])
        y_true_noisy = np.concatenate([y_true_noisy, target_nois
y.data.cpu().numpy()])

    # calc metric
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true, preds)
    lwlrap = np.sum(per_class_lwlrap * weight_per_class)
```

```
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true_noisy, preds_noisy)
    lwlrap_noisy = np.sum(per_class_lwlrap * weight_per_class)

    return kl_avr.avg.item(), lwlrap, kl_noisy_avr.avg.item(), l
wlrap_noisy


def validate(val_loader, model):
    kl_avr = AverageMeter()
    lsoftmax = nn.LogSoftmax(dim=1).cuda()
    softmax = torch.nn.Softmax(dim=1).cuda()
    criterion_kl = nn.KLDivLoss().cuda()

    # switch to eval mode
    model.eval()

    # validate
    preds = np.zeros([0, NUM_CLASS], np.float32)
    y_true = np.zeros([0, NUM_CLASS], np.float32)
    for i, (input, target) in enumerate(val_loader):
        # get batches
        input = torch.autograd.Variable(input.cuda())
        target = torch.autograd.Variable(target.cuda())

        # compute output
        with torch.no_grad():
            output = model(input)
            kl = criterion_kl(lsoftmax(output), target)
            pred = softmax(output)
            pred = pred.data.cpu().numpy()

        # record log
        kl_avr.update(kl.data, input.size(0))
        preds = np.concatenate([preds, pred])
        y_true = np.concatenate([y_true, target.data.cpu().numpy
()])

    # calc metric
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true, preds)
    lwlrap = np.sum(per_class_lwlrap * weight_per_class)

    return kl_avr.avg.item(), lwlrap


if __name__ == '__main__':
    main()import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time, random, sys
import sklearn.metrics
from sklearn.model_selection import KFold

import torch
import torch.nn as nn
```

```
import torch.backends.cudnn as cudnn
import torch.optim as optim
from torch.utils.data import DataLoader

sys.path.append('.')
from utils import AverageMeter, cycle, CosineLR, WaveDataset, ca
lculate_per_class_lwlrap
from models import EnvNetv2

# set parameters
NUM_FOLD = 5
NUM_CLASS = 80
SEED = 42
NUM_EPOCH =4 #80*3
NUM_CYCLE = 2 #80
BATCH_SIZE = 64
LR = [1e-1, 1e-6]
FOLD_LIST = [1, 2, 3, 4, 5]
FOLD_LIST = [1, 2,]
CROP_LENGTH = 133300
LOAD_DIR = "../models/envnet_model4_0"
OUTPUT_DIR = "../models/envnet_model4"

cudnn.benchmark = True
starttime = time.time()


def main():
    # load table data
    df_train = pd.read_csv("../input/train_curated.csv")
    df_noisy = pd.read_csv("../input/train_noisy.csv")
    df_test = pd.read_csv("../input/sample_submission.csv")
    labels = df_test.columns[1:].tolist()
    for label in labels:
        df_train[label] = df_train['labels'].apply(lambda x: lab
el in x)
        df_noisy[label] = df_noisy['labels'].apply(lambda x: lab
el in x)

    df_train['path'] = "../input/train_curated/" + df_train['fna
me']
    df_test['path'] = "../input/test/" + df_train['fname']
    df_noisy['path'] = "../input/train_noisy/" + df_noisy['fname
']

    # fold splitting
    folds = list(KFold(n_splits=NUM_FOLD, shuffle=True, random_s
tate=SEED).split(np.arange(len(df_train))))

    # Training
    log_columns = ['epoch', 'bce', 'lwlrap', 'bce_noisy', 'lwlra
p_noisy', 'val_bce', 'val_lwlrap', 'time']
    for fold, (ids_train_split, ids_valid_split) in enumerate(fo
lds):
        if fold+1 not in FOLD_LIST: continue
        print("fold: {}".format(fold + 1))
```

```
        train_log = pd.DataFrame(columns=log_columns)

        # build model
        model = EnvNetv2(NUM_CLASS).cuda()
        # model.load_state_dict(torch.load("{}/weight_fold_{}_ep
och_400.pth".format(LOAD_DIR, fold+1)))

        # prepare data loaders
        df_train_fold = df_train.iloc[ids_train_split].reset_ind
ex(drop=True)
        dataset_train = WaveDataset(df_train_fold['path'], df_tr
ain_fold[labels].values,
                                    crop=CROP_LENGTH, crop_mode=
'random', padding=CROP_LENGTH//2,
                                    mixup=True, scaling=1.25, ga
in=6,
                                    )
        train_loader = DataLoader(dataset_train, batch_size=BATC
H_SIZE,
                                  shuffle=True, num_workers=1, p
in_memory=True,
                                  )

        df_valid = df_train.iloc[ids_valid_split].reset_index(dr
op=True)
        dataset_valid = WaveDataset(df_valid['path'], df_valid[l
abels].values, padding=CROP_LENGTH//2)
        valid_loader = DataLoader(dataset_valid, batch_size=1,
                                  shuffle=False, num_workers=1,
pin_memory=True,
                                  )

        dataset_noisy = WaveDataset(df_noisy['path'], df_noisy[l
abels].values,
                                    crop=CROP_LENGTH, crop_mode=
'random', padding=CROP_LENGTH//2,
                                    mixup=True, scaling=1.25, ga
in=6,
                                   )
        noisy_loader = DataLoader(dataset_noisy, batch_size=BATC
H_SIZE,
                                  shuffle=True, num_workers=1, p
in_memory=True,
                                  )
        noisy_itr = cycle(noisy_loader)

        # set optimizer and loss
        optimizer = optim.SGD(filter(lambda p: p.requires_grad,
model.parameters()), lr=LR[0],momentum = 0.9,nesterov = True)
        scheduler = CosineLR(optimizer, step_size_min=LR[1], t0=
len(train_loader) * NUM_CYCLE, tmult=1)

        # training
        for epoch in range(NUM_EPOCH):
            # train for one epoch
            bce, lwlrap, bce_noisy, lwlrap_noisy = train((train_
```

```
loader, noisy_itr), model, optimizer, scheduler, epoch)

            # evaluate on validation set
            val_bce, val_lwlrap = validate(valid_loader, model)

            # print log
            endtime = time.time() - starttime
            print("Epoch: {}/{} ".format(epoch + 1, NUM_EPOCH)
                  + "CE: {:.4f} ".format(bce)
                  + "LwLRAP: {:.4f} ".format(lwlrap)
                  + "Noisy CE: {:.4f} ".format(bce_noisy)
                  + "Noisy LWLRAP: {:.4f} ".format(lwlrap_noisy)
                  + "Valid CE: {:.4f} ".format(val_bce)
                  + "Valid LWLRAP: {:.4f} ".format(val_lwlrap)
                  + "sec: {:.1f}".format(endtime)
                  )

            # save log and weights
            train_log_epoch = pd.DataFrame(
                [[epoch+1, bce, lwlrap, bce_noisy, lwlrap_noisy,
 val_bce, val_lwlrap, endtime]],
                columns=log_columns)
            train_log = pd.concat([train_log, train_log_epoch])
            train_log.to_csv("{}/train_log_fold{}.csv".format(OU
TPUT_DIR, fold+1), index=False)
            if (epoch+1)%NUM_CYCLE==0:
                torch.save(model.state_dict(), "{}/weight_fold_{
}_epoch_{}.pth".format(OUTPUT_DIR, fold+1, epoch+1))


def train(train_loaders, model, optimizer, scheduler, epoch):
    train_loader, noisy_itr = train_loaders
    bce_avr = AverageMeter()
    bce_noisy_avr = AverageMeter()
    criterion_bce = nn.BCEWithLogitsLoss().cuda()
    sigmoid = nn.Sigmoid().cuda()

    # switch to train mode
    model.train()

    # training
    preds = np.zeros([0, NUM_CLASS], np.float32)
    y_true = np.zeros([0, NUM_CLASS], np.float32)
    preds_noisy = np.zeros([0, NUM_CLASS], np.float32)
    y_true_noisy = np.zeros([0, NUM_CLASS], np.float32)
    for i, (input, target) in enumerate(train_loader):
        # get batches
        input = torch.autograd.Variable(input.cuda())
        target = torch.autograd.Variable(target.cuda())

        input_noisy, target_noisy = next(noisy_itr)
        input_noisy = torch.autograd.Variable(input_noisy.cuda()
)
        target_noisy = torch.autograd.Variable(target_noisy.cuda
())
```

```
        # compute output
        output = model(input)
        bce = criterion_bce(sigmoid(output), target)
        output_noisy = model.noisy(input_noisy)
        bce_noisy = criterion_bce(sigmoid(output_noisy), target_
noisy)
        loss = bce + bce_noisy
        pred = sigmoid(output)
        pred = pred.data.cpu().numpy()
        pred_noisy = sigmoid(output_noisy)
        pred_noisy = pred_noisy.data.cpu().numpy()

        # backprop
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step()

        # record log
        bce_avr.update(bce.data, input.size(0))
        bce_noisy_avr.update(bce_noisy.data, input.size(0))
        preds = np.concatenate([preds, pred])
        y_true = np.concatenate([y_true, target.data.cpu().numpy
()])
        preds_noisy = np.concatenate([preds_noisy, pred_noisy])
        y_true_noisy = np.concatenate([y_true_noisy, target_nois
y.data.cpu().numpy()])

    # calc metric
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true, preds)
    lwlrap = np.sum(per_class_lwlrap * weight_per_class)
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true_noisy, preds_noisy)
    lwlrap_noisy = np.sum(per_class_lwlrap * weight_per_class)

    return bce_avr.avg.item(), lwlrap, bce_noisy_avr.avg.item(),
 lwlrap_noisy


def validate(val_loader, model):
    bce_avr = AverageMeter()
    sigmoid = nn.Sigmoid().cuda()
    criterion_bce = nn.BCEWithLogitsLoss().cuda()

    # switch to eval mode
    model.eval()

    # validate
    preds = np.zeros([0, NUM_CLASS], np.float32)
    y_true = np.zeros([0, NUM_CLASS], np.float32)
    for i, (input, target) in enumerate(val_loader):
        # get batches
        input = torch.autograd.Variable(input.cuda())
        target = torch.autograd.Variable(target.cuda())
```

```
        # compute output
        with torch.no_grad():
            output = model(input)
            bce = criterion_bce(sigmoid(output), target)
            pred = sigmoid(output)
            pred = pred.data.cpu().numpy()

        # record log
        bce_avr.update(bce.data, input.size(0))
        preds = np.concatenate([preds, pred])
        y_true = np.concatenate([y_true, target.data.cpu().numpy
()])

    # calc metric
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true, preds)
    lwlrap = np.sum(per_class_lwlrap * weight_per_class)

    return bce_avr.avg.item(), lwlrap


if __name__ == '__main__':
    main()import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time, random, sys
import sklearn.metrics
from sklearn.model_selection import KFold

import torch
import torch.nn as nn
import torch.backends.cudnn as cudnn
import torch.optim as optim
from torch.utils.data import DataLoader

sys.path.append('.')
from utils import AverageMeter, cycle, CosineLR, WaveDataset, ca
lculate_per_class_lwlrap
from models import EnvNetv2

# set parameters
NUM_FOLD = 5
NUM_CLASS = 80
SEED = 42
NUM_EPOCH =80*5
NUM_CYCLE = 80
BATCH_SIZE = 16
LR = [1e-1, 1e-6]
FOLD_LIST = [1, 2, 3, 4, 5]
CROP_LENGTH = 133300
LOAD_DIR = "../models/envnet_model4_0"
OUTPUT_DIR = "../models/envnet_model5"

cudnn.benchmark = True
starttime = time.time()
```

```
def main():
    # load table data
    df_train = pd.read_csv("../input/train_curated.csv")
    df_noisy = pd.read_csv("../input/train_noisy.csv")
    df_test = pd.read_csv("../input/sample_submission.csv")
    labels = df_test.columns[1:].tolist()
    for label in labels:
        df_train[label] = df_train['labels'].apply(lambda x: lab
el in x)
        df_noisy[label] = df_noisy['labels'].apply(lambda x: lab
el in x)

    df_train['path'] = "../input/train_curated/" + df_train['fna
me']
    df_test['path'] = "../input/test/" + df_train['fname']
    df_noisy['path'] = "../input/train_noisy/" + df_noisy['fname
']

    # fold splitting
    folds = list(KFold(n_splits=NUM_FOLD, shuffle=True, random_s
tate=SEED).split(np.arange(len(df_train))))

    # Training
    log_columns = ['epoch', 'kl', 'lwlrap', 'kl_noisy', 'lwlrap_
noisy', 'val_kl', 'val_lwlrap', 'time']
    for fold, (ids_train_split, ids_valid_split) in enumerate(fo
lds):
        if fold+1 not in FOLD_LIST: continue
        print("fold: {}".format(fold + 1))
        train_log = pd.DataFrame(columns=log_columns)

        # build model
        model = EnvNetv2(NUM_CLASS).cuda()
        model.load_state_dict(torch.load("{}/weight_fold_{}_epoc
h_400.pth".format(LOAD_DIR, fold+1)))

        # prepare data loaders
        df_train_fold = df_train.iloc[ids_train_split].reset_ind
ex(drop=True)
        dataset_train = WaveDataset(df_train_fold['path'], df_tr
ain_fold[labels].values,
                                    crop=CROP_LENGTH, crop_mode=
'random', padding=CROP_LENGTH//2,
                                    mixup=True, scaling=1.25, ga
in=6,
                                    )
        train_loader = DataLoader(dataset_train, batch_size=BATC
H_SIZE,
                                  shuffle=True, num_workers=1, p
in_memory=True,
                                  )

        df_valid = df_train.iloc[ids_valid_split].reset_index(dr
op=True)
        dataset_valid = WaveDataset(df_valid['path'], df_valid[l
```

```
abels].values, padding=CROP_LENGTH//2)
        valid_loader = DataLoader(dataset_valid, batch_size=1,
                                  shuffle=False, num_workers=1,
pin_memory=True,
                                  )

        dataset_noisy = WaveDataset(df_noisy['path'], df_noisy[l
abels].values,
                                    crop=CROP_LENGTH, crop_mode=
'random', padding=CROP_LENGTH//2,
                                    mixup=True, scaling=1.25, ga
in=6,
                                   )
        noisy_loader = DataLoader(dataset_noisy, batch_size=BATC
H_SIZE,
                                  shuffle=True, num_workers=1, p
in_memory=True,
                                  )
        noisy_itr = cycle(noisy_loader)

        # set optimizer and loss
        optimizer = optim.SGD(filter(lambda p: p.requires_grad,
model.parameters()), lr=LR[0],momentum = 0.9, nesterov = True)
        scheduler = CosineLR(optimizer, step_size_min=LR[1], t0=
len(train_loader) * NUM_CYCLE, tmult=1)

        # training
        for epoch in range(NUM_EPOCH):
            # train for one epoch
            kl, lwlrap, kl_noisy, lwlrap_noisy = train((train_lo
ader, noisy_itr), model, optimizer, scheduler, epoch)

            # evaluate on validation set
            val_kl, val_lwlrap = validate(valid_loader, model)

            # print log
            endtime = time.time() - starttime
            print("Epoch: {}/{} ".format(epoch + 1, NUM_EPOCH)
                  + "KL: {:.4f} ".format(kl)
                  + "LwLRAP: {:.4f} ".format(lwlrap)
                  + "Noisy KL: {:.4f} ".format(kl_noisy)
                  + "Noisy LWLRAP: {:.4f} ".format(lwlrap_noisy)
                  + "Valid KL: {:.4f} ".format(val_kl)
                  + "Valid LWLRAP: {:.4f} ".format(val_lwlrap)
                  + "sec: {:.1f}".format(endtime)
                  )

            # save log and weights
            train_log_epoch = pd.DataFrame(
                [[epoch+1, kl, lwlrap, kl_noisy, lwlrap_noisy, v
al_kl, val_lwlrap, endtime]],
                columns=log_columns)
            train_log = pd.concat([train_log, train_log_epoch])
            train_log.to_csv("{}/train_log_fold{}.csv".format(OU
TPUT_DIR, fold+1), index=False)
            if (epoch+1)%NUM_CYCLE==0:
```

```
                torch.save(model.state_dict(), "{}/weight_fold_{
}_epoch_{}.pth".format(OUTPUT_DIR, fold+1, epoch+1))


def train(train_loaders, model, optimizer, scheduler, epoch):
    train_loader, noisy_itr = train_loaders
    kl_avr = AverageMeter()
    kl_noisy_avr = AverageMeter()
    lsigmoid = nn.LogSigmoid().cuda()
    lsoftmax = nn.LogSoftmax(dim=1).cuda()
    softmax = nn.Softmax(dim=1).cuda()
    criterion_kl = nn.KLDivLoss().cuda()

    # switch to train mode
    model.train()

    # training
    preds = np.zeros([0, NUM_CLASS], np.float32)
    y_true = np.zeros([0, NUM_CLASS], np.float32)
    preds_noisy = np.zeros([0, NUM_CLASS], np.float32)
    y_true_noisy = np.zeros([0, NUM_CLASS], np.float32)
    for i, (input, target) in enumerate(train_loader):
        # get batches
        input = torch.autograd.Variable(input.cuda())
        target = torch.autograd.Variable(target.cuda())

        input_noisy, target_noisy = next(noisy_itr)
        input_noisy = torch.autograd.Variable(input_noisy.cuda()
)
        target_noisy = torch.autograd.Variable(target_noisy.cuda
())

        # compute output
        output = model(input)
        kl = criterion_kl(lsoftmax(output), target)
        output_noisy = model.noisy(input_noisy)
        kl_noisy = criterion_kl(lsoftmax(output_noisy), target_n
oisy)
        loss = kl + kl_noisy
        pred = softmax(output)
        pred = pred.data.cpu().numpy()
        pred_noisy = softmax(output_noisy)
        pred_noisy = pred_noisy.data.cpu().numpy()

        # backprop
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step()

        # record log
        kl_avr.update(kl.data, input.size(0))
        kl_noisy_avr.update(kl_noisy.data, input.size(0))
        preds = np.concatenate([preds, pred])
        y_true = np.concatenate([y_true, target.data.cpu().numpy
()])
```

```
        preds_noisy = np.concatenate([preds_noisy, pred_noisy])
        y_true_noisy = np.concatenate([y_true_noisy, target_nois
y.data.cpu().numpy()])

    # calc metric
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true, preds)
    lwlrap = np.sum(per_class_lwlrap * weight_per_class)
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true_noisy, preds_noisy)
    lwlrap_noisy = np.sum(per_class_lwlrap * weight_per_class)

    return kl_avr.avg.item(), lwlrap, kl_noisy_avr.avg.item(), l
wlrap_noisy


def validate(val_loader, model):
    kl_avr = AverageMeter()
    lsoftmax = nn.LogSoftmax(dim=1).cuda()
    softmax = torch.nn.Softmax(dim=1).cuda()
    criterion_kl = nn.KLDivLoss().cuda()

    # switch to eval mode
    model.eval()

    # validate
    preds = np.zeros([0, NUM_CLASS], np.float32)
    y_true = np.zeros([0, NUM_CLASS], np.float32)
    for i, (input, target) in enumerate(val_loader):
        # get batches
        input = torch.autograd.Variable(input.cuda())
        target = torch.autograd.Variable(target.cuda())

        # compute output
        with torch.no_grad():
            output = model(input)
            kl = criterion_kl(lsoftmax(output), target)
            pred = softmax(output)
            pred = pred.data.cpu().numpy()

        # record log
        kl_avr.update(kl.data, input.size(0))
        preds = np.concatenate([preds, pred])
        y_true = np.concatenate([y_true, target.data.cpu().numpy
()])

    # calc metric
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true, preds)
    lwlrap = np.sum(per_class_lwlrap * weight_per_class)

    return kl_avr.avg.item(), lwlrap


if __name__ == '__main__':
    main()import numpy as np
```

```
import pandas as pd
import matplotlib.pyplot as plt
import time, random, sys
import sklearn.metrics
from sklearn.model_selection import KFold

import torch
import torch.nn as nn
import torch.backends.cudnn as cudnn
import torch.optim as optim
from torch.utils.data import DataLoader

sys.path.append('.')
from utils import AverageMeter, cycle, CosineLR, WaveDataset, ca
lculate_per_class_lwlrap
from models import EnvNetv2

# set parameters
NUM_FOLD = 5
NUM_CLASS = 80
SEED = 42
NUM_EPOCH =80*5
NUM_CYCLE = 80
BATCH_SIZE = 16
LR = [1e-1, 1e-6]
FOLD_LIST = [1, 2, 3, 4, 5]
CROP_LENGTH = 200000
OUTPUT_DIR = "../models/envnet_model6_0"

cudnn.benchmark = True
starttime = time.time()


def main():
    # load table data
    df_train = pd.read_csv("../input/train_curated.csv")
    df_noisy = pd.read_csv("../input/train_noisy.csv")
    df_test = pd.read_csv("../input/sample_submission.csv")
    labels = df_test.columns[1:].tolist()
    for label in labels:
        df_train[label] = df_train['labels'].apply(lambda x: lab
el in x)
        df_noisy[label] = df_noisy['labels'].apply(lambda x: lab
el in x)

    df_train['path'] = "../input/train_curated/" + df_train['fna
me']
    df_test['path'] = "../input/test/" + df_train['fname']
    df_noisy['path'] = "../input/train_noisy/" + df_noisy['fname
']

    # fold splitting
    folds = list(KFold(n_splits=NUM_FOLD, shuffle=True, random_s
tate=SEED).split(np.arange(len(df_train))))

    # Training
```

```
    log_columns = ['epoch', 'kl', 'lwlrap', 'kl_noisy', 'lwlrap_
noisy', 'val_kl', 'val_lwlrap', 'time']
    for fold, (ids_train_split, ids_valid_split) in enumerate(fo
lds):
        if fold+1 not in FOLD_LIST: continue
        print("fold: {}".format(fold + 1))
        train_log = pd.DataFrame(columns=log_columns)

        # build model
        model = EnvNetv2(NUM_CLASS).cuda()

        # prepare data loaders
        df_train_fold = df_train.iloc[ids_train_split].reset_ind
ex(drop=True)
        dataset_train = WaveDataset(df_train_fold['path'], df_tr
ain_fold[labels].values,
                                    crop=CROP_LENGTH, crop_mode=
'random', padding=CROP_LENGTH//2,
                                    mixup=True, scaling=1.25, ga
in=6,
                                    )
        train_loader = DataLoader(dataset_train, batch_size=BATC
H_SIZE,
                                  shuffle=True, num_workers=1, p
in_memory=True,
                                  )

        df_valid = df_train.iloc[ids_valid_split].reset_index(dr
op=True)
        dataset_valid = WaveDataset(df_valid['path'], df_valid[l
abels].values, padding=CROP_LENGTH//2)
        valid_loader = DataLoader(dataset_valid, batch_size=1,
                                  shuffle=False, num_workers=1,
pin_memory=True,
                                  )

        dataset_noisy = WaveDataset(df_noisy['path'], df_noisy[l
abels].values,
                                    crop=CROP_LENGTH, crop_mode=
'random', padding=CROP_LENGTH//2,
                                    mixup=True, scaling=1.25, ga
in=6,
                                   )
        noisy_loader = DataLoader(dataset_noisy, batch_size=BATC
H_SIZE,
                                  shuffle=True, num_workers=1, p
in_memory=True,
                                  )
        noisy_itr = cycle(noisy_loader)

        # set optimizer and loss
        optimizer = optim.SGD(filter(lambda p: p.requires_grad,
model.parameters()), lr=LR[0],momentum = 0.9, nesterov = True)
        scheduler = CosineLR(optimizer, step_size_min=LR[1], t0=
len(train_loader) * NUM_CYCLE, tmult=1)
```

```
        # training
        for epoch in range(NUM_EPOCH):
            # train for one epoch
            kl, lwlrap, kl_noisy, lwlrap_noisy = train((train_lo
ader, noisy_itr), model, optimizer, scheduler, epoch)

            # evaluate on validation set
            val_kl, val_lwlrap = validate(valid_loader, model)

            # print log
            endtime = time.time() - starttime
            print("Epoch: {}/{} ".format(epoch + 1, NUM_EPOCH)
                  + "KL: {:.4f} ".format(kl)
                  + "LwLRAP: {:.4f} ".format(lwlrap)
                  + "Noisy KL: {:.4f} ".format(kl_noisy)
                  + "Noisy LWLRAP: {:.4f} ".format(lwlrap_noisy)
                  + "Valid KL: {:.4f} ".format(val_kl)
                  + "Valid LWLRAP: {:.4f} ".format(val_lwlrap)
                  + "sec: {:.1f}".format(endtime)
                  )

            # save log and weights
            train_log_epoch = pd.DataFrame(
                [[epoch+1, kl, lwlrap, kl_noisy, lwlrap_noisy, v
al_kl, val_lwlrap, endtime]],
                columns=log_columns)
            train_log = pd.concat([train_log, train_log_epoch])
            train_log.to_csv("{}/train_log_fold{}.csv".format(OU
TPUT_DIR, fold+1), index=False)
            if (epoch+1)%NUM_CYCLE==0:
                torch.save(model.state_dict(), "{}/weight_fold_{
}_epoch_{}.pth".format(OUTPUT_DIR, fold+1, epoch+1))


def train(train_loaders, model, optimizer, scheduler, epoch):
    train_loader, noisy_itr = train_loaders
    kl_avr = AverageMeter()
    kl_noisy_avr = AverageMeter()
    lsigmoid = nn.LogSigmoid().cuda()
    lsoftmax = nn.LogSoftmax(dim=1).cuda()
    softmax = nn.Softmax(dim=1).cuda()
    criterion_kl = nn.KLDivLoss().cuda()

    # switch to train mode
    model.train()

    # training
    preds = np.zeros([0, NUM_CLASS], np.float32)
    y_true = np.zeros([0, NUM_CLASS], np.float32)
    preds_noisy = np.zeros([0, NUM_CLASS], np.float32)
    y_true_noisy = np.zeros([0, NUM_CLASS], np.float32)
    for i, (input, target) in enumerate(train_loader):
        # get batches
        input = torch.autograd.Variable(input.cuda())
        target = torch.autograd.Variable(target.cuda())
```

```
        input_noisy, target_noisy = next(noisy_itr)
        input_noisy = torch.autograd.Variable(input_noisy.cuda()
)
        target_noisy = torch.autograd.Variable(target_noisy.cuda
())

        # compute output
        output = model(input)
        kl = criterion_kl(lsoftmax(output), target)
        output_noisy = model.noisy(input_noisy)
        kl_noisy = criterion_kl(lsoftmax(output_noisy), target_n
oisy)
        loss = kl + kl_noisy
        pred = softmax(output)
        pred = pred.data.cpu().numpy()
        pred_noisy = softmax(output_noisy)
        pred_noisy = pred_noisy.data.cpu().numpy()

        # backprop
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step()

        # record log
        kl_avr.update(kl.data, input.size(0))
        kl_noisy_avr.update(kl_noisy.data, input.size(0))
        preds = np.concatenate([preds, pred])
        y_true = np.concatenate([y_true, target.data.cpu().numpy
()])
        preds_noisy = np.concatenate([preds_noisy, pred_noisy])
        y_true_noisy = np.concatenate([y_true_noisy, target_nois
y.data.cpu().numpy()])

    # calc metric
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true, preds)
    lwlrap = np.sum(per_class_lwlrap * weight_per_class)
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true_noisy, preds_noisy)
    lwlrap_noisy = np.sum(per_class_lwlrap * weight_per_class)

    return kl_avr.avg.item(), lwlrap, kl_noisy_avr.avg.item(), l
wlrap_noisy


def validate(val_loader, model):
    kl_avr = AverageMeter()
    lsoftmax = nn.LogSoftmax(dim=1).cuda()
    softmax = torch.nn.Softmax(dim=1).cuda()
    criterion_kl = nn.KLDivLoss().cuda()

    # switch to eval mode
    model.eval()

    # validate
```

```
    preds = np.zeros([0, NUM_CLASS], np.float32)
    y_true = np.zeros([0, NUM_CLASS], np.float32)
    for i, (input, target) in enumerate(val_loader):
        # get batches
        input = torch.autograd.Variable(input.cuda())
        target = torch.autograd.Variable(target.cuda())

        # compute output
        with torch.no_grad():
            output = model(input)
            kl = criterion_kl(lsoftmax(output), target)
            pred = softmax(output)
            pred = pred.data.cpu().numpy()

        # record log
        kl_avr.update(kl.data, input.size(0))
        preds = np.concatenate([preds, pred])
        y_true = np.concatenate([y_true, target.data.cpu().numpy
()])

    # calc metric
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true, preds)
    lwlrap = np.sum(per_class_lwlrap * weight_per_class)

    return kl_avr.avg.item(), lwlrap


if __name__ == '__main__':
    main()import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time, random, sys
import sklearn.metrics
from sklearn.model_selection import KFold

import torch
import torch.nn as nn
import torch.backends.cudnn as cudnn
import torch.optim as optim
from torch.utils.data import DataLoader

sys.path.append('.')
from utils import AverageMeter, cycle, CosineLR, WaveDataset, ca
lculate_per_class_lwlrap
from models import EnvNetv2

# set parameters
NUM_FOLD = 5
NUM_CLASS = 80
SEED = 42
NUM_EPOCH =80*5
NUM_CYCLE = 80
BATCH_SIZE = 16
LR = [1e-1, 1e-6]
FOLD_LIST = [1, 2, 3, 4, 5]
```

```
CROP_LENGTH = 200000
LOAD_DIR = "../models/envnet_model6_0"
OUTPUT_DIR = "../models/envnet_model6"

cudnn.benchmark = True
starttime = time.time()


def main():
    # load table data
    df_train = pd.read_csv("../input/train_curated.csv")
    df_noisy = pd.read_csv("../input/train_noisy.csv")
    df_test = pd.read_csv("../input/sample_submission.csv")
    labels = df_test.columns[1:].tolist()
    for label in labels:
        df_train[label] = df_train['labels'].apply(lambda x: lab
el in x)
        df_noisy[label] = df_noisy['labels'].apply(lambda x: lab
el in x)

    df_train['path'] = "../input/train_curated/" + df_train['fna
me']
    df_test['path'] = "../input/test/" + df_train['fname']
    df_noisy['path'] = "../input/train_noisy/" + df_noisy['fname
']

    # fold splitting
    folds = list(KFold(n_splits=NUM_FOLD, shuffle=True, random_s
tate=SEED).split(np.arange(len(df_train))))

    # Training
    log_columns = ['epoch', 'kl', 'lwlrap', 'kl_noisy', 'lwlrap_
noisy', 'val_kl', 'val_lwlrap', 'time']
    for fold, (ids_train_split, ids_valid_split) in enumerate(fo
lds):
        if fold+1 not in FOLD_LIST: continue
        print("fold: {}".format(fold + 1))
        train_log = pd.DataFrame(columns=log_columns)

        # build model
        model = EnvNetv2(NUM_CLASS).cuda()
        model.load_state_dict(torch.load("{}/weight_fold_{}_epoc
h_400.pth".format(LOAD_DIR, fold+1)))

        # prepare data loaders
        df_train_fold = df_train.iloc[ids_train_split].reset_ind
ex(drop=True)
        dataset_train = WaveDataset(df_train_fold['path'], df_tr
ain_fold[labels].values,
                                    crop=CROP_LENGTH, crop_mode=
'random', padding=CROP_LENGTH//2,
                                    mixup=True, scaling=1.25, ga
in=6,
                                    )
        train_loader = DataLoader(dataset_train, batch_size=BATC
H_SIZE,
```

```
                                        shuffle=True, num_workers=1, p
in_memory=True,
                                        )

        df_valid = df_train.iloc[ids_valid_split].reset_index(dr
op=True)
        dataset_valid = WaveDataset(df_valid['path'], df_valid[l
abels].values, padding=CROP_LENGTH//2)
        valid_loader = DataLoader(dataset_valid, batch_size=1,
                                        shuffle=False, num_workers=1,
pin_memory=True,
                                        )

        dataset_noisy = WaveDataset(df_noisy['path'], df_noisy[l
abels].values,
                                          crop=CROP_LENGTH, crop_mode=
'random', padding=CROP_LENGTH//2,
                                          mixup=True, scaling=1.25, ga
in=6,
                                         )
        noisy_loader = DataLoader(dataset_noisy, batch_size=BATC
H_SIZE,
                                        shuffle=True, num_workers=1, p
in_memory=True,
                                        )
        noisy_itr = cycle(noisy_loader)

        # set optimizer and loss
        optimizer = optim.SGD(filter(lambda p: p.requires_grad,
model.parameters()), lr=LR[0],momentum = 0.9, nesterov = True)
        scheduler = CosineLR(optimizer, step_size_min=LR[1], t0=
len(train_loader) * NUM_CYCLE, tmult=1)

        # training
        for epoch in range(NUM_EPOCH):
            # train for one epoch
            kl, lwlrap, kl_noisy, lwlrap_noisy = train((train_lo
ader, noisy_itr), model, optimizer, scheduler, epoch)

            # evaluate on validation set
            val_kl, val_lwlrap = validate(valid_loader, model)

            # print log
            endtime = time.time() - starttime
            print("Epoch: {}/{} ".format(epoch + 1, NUM_EPOCH)
                  + "KL: {:.4f} ".format(kl)
                  + "LwLRAP: {:.4f} ".format(lwlrap)
                  + "Noisy KL: {:.4f} ".format(kl_noisy)
                  + "Noisy LWLRAP: {:.4f} ".format(lwlrap_noisy)
                  + "Valid KL: {:.4f} ".format(val_kl)
                  + "Valid LWLRAP: {:.4f} ".format(val_lwlrap)
                  + "sec: {:.1f}".format(endtime)
                  )

            # save log and weights
            train_log_epoch = pd.DataFrame(
```

```
                [[epoch+1, kl, lwlrap, kl_noisy, lwlrap_noisy, v
al_kl, val_lwlrap, endtime]],
                columns=log_columns)
            train_log = pd.concat([train_log, train_log_epoch])
            train_log.to_csv("{}/train_log_fold{}.csv".format(OU
TPUT_DIR, fold+1), index=False)
            if (epoch+1)%NUM_CYCLE==0:
                torch.save(model.state_dict(), "{}/weight_fold_{
}_epoch_{}.pth".format(OUTPUT_DIR, fold+1, epoch+1))


def train(train_loaders, model, optimizer, scheduler, epoch):
    train_loader, noisy_itr = train_loaders
    kl_avr = AverageMeter()
    kl_noisy_avr = AverageMeter()
    lsigmoid = nn.LogSigmoid().cuda()
    lsoftmax = nn.LogSoftmax(dim=1).cuda()
    softmax = nn.Softmax(dim=1).cuda()
    criterion_kl = nn.KLDivLoss().cuda()

    # switch to train mode
    model.train()

    # training
    preds = np.zeros([0, NUM_CLASS], np.float32)
    y_true = np.zeros([0, NUM_CLASS], np.float32)
    preds_noisy = np.zeros([0, NUM_CLASS], np.float32)
    y_true_noisy = np.zeros([0, NUM_CLASS], np.float32)
    for i, (input, target) in enumerate(train_loader):
        # get batches
        input = torch.autograd.Variable(input.cuda())
        target = torch.autograd.Variable(target.cuda())

        input_noisy, target_noisy = next(noisy_itr)
        input_noisy = torch.autograd.Variable(input_noisy.cuda()
)
        target_noisy = torch.autograd.Variable(target_noisy.cuda
())

        # compute output
        output = model(input)
        kl = criterion_kl(lsoftmax(output), target)
        output_noisy = model.noisy(input_noisy)
        kl_noisy = criterion_kl(lsoftmax(output_noisy), target_n
oisy)
        loss = kl + kl_noisy
        pred = softmax(output)
        pred = pred.data.cpu().numpy()
        pred_noisy = softmax(output_noisy)
        pred_noisy = pred_noisy.data.cpu().numpy()

        # backprop
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step()
```

```
        # record log
        kl_avr.update(kl.data, input.size(0))
        kl_noisy_avr.update(kl_noisy.data, input.size(0))
        preds = np.concatenate([preds, pred])
        y_true = np.concatenate([y_true, target.data.cpu().numpy
()])
        preds_noisy = np.concatenate([preds_noisy, pred_noisy])
        y_true_noisy = np.concatenate([y_true_noisy, target_nois
y.data.cpu().numpy()])

    # calc metric
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true, preds)
    lwlrap = np.sum(per_class_lwlrap * weight_per_class)
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true_noisy, preds_noisy)
    lwlrap_noisy = np.sum(per_class_lwlrap * weight_per_class)

    return kl_avr.avg.item(), lwlrap, kl_noisy_avr.avg.item(), l
wlrap_noisy


def validate(val_loader, model):
    kl_avr = AverageMeter()
    lsoftmax = nn.LogSoftmax(dim=1).cuda()
    softmax = torch.nn.Softmax(dim=1).cuda()
    criterion_kl = nn.KLDivLoss().cuda()

    # switch to eval mode
    model.eval()

    # validate
    preds = np.zeros([0, NUM_CLASS], np.float32)
    y_true = np.zeros([0, NUM_CLASS], np.float32)
    for i, (input, target) in enumerate(val_loader):
        # get batches
        input = torch.autograd.Variable(input.cuda())
        target = torch.autograd.Variable(target.cuda())

        # compute output
        with torch.no_grad():
            output = model(input)
            kl = criterion_kl(lsoftmax(output), target)
            pred = softmax(output)
            pred = pred.data.cpu().numpy()

        # record log
        kl_avr.update(kl.data, input.size(0))
        preds = np.concatenate([preds, pred])
        y_true = np.concatenate([y_true, target.data.cpu().numpy
()])

    # calc metric
    per_class_lwlrap, weight_per_class = calculate_per_class_lwl
rap(y_true, preds)
```

```
    lwlrap = np.sum(per_class_lwlrap * weight_per_class)

    return kl_avr.avg.item(), lwlrap


if __name__ == '__main__':
    main()import numpy as np
from torch.optim.lr_scheduler import _LRScheduler
from torch.utils.data.dataset import Dataset
from math import cos, pi
import librosa
from scipy.io import wavfile
import random

class AverageMeter(object):
    """Computes and stores the average and current value"""

    def __init__(self):
        self.reset()

    def reset(self):
        self.val = 0
        self.avg = 0
        self.sum = 0
        self.count = 0

    def update(self, val, n=1):
        self.val = val
        self.sum += val * n
        self.count += n
        self.avg = self.sum / self.count


def cycle(iterable):
    """
    convert dataloader to iterator
    :param iterable:
    :return:
    """
    while True:
        for x in iterable:
            yield x


class CosineLR(_LRScheduler):
    """cosine annealing.
    """
    def __init__(self, optimizer, step_size_min=1e-5, t0=100, tm
ult=2, curr_epoch=-1, last_epoch=-1):
        self.step_size_min = step_size_min
        self.t0 = t0
        self.tmult = tmult
        self.epochs_since_restart = curr_epoch
        super(CosineLR, self).__init__(optimizer, last_epoch)

    def get_lr(self):
```

```
        self.epochs_since_restart += 1

        if self.epochs_since_restart > self.t0:
            self.t0 *= self.tmult
            self.epochs_since_restart = 0

        lrs = [self.step_size_min + (
                0.5 * (base_lr - self.step_size_min) * (1 + cos(
self.epochs_since_restart * pi / self.t0)))
               for base_lr in self.base_lrs]

        return lrs


class MelDataset(Dataset):
    def __init__(self, X, y, crop=-1,
                 mixup=False, freqmask=False, gain=False,
                 crop_mode='original',crop_rate=0.25
                 ):
        self.X= X
        self.y= y
        self.crop = crop
        self.mixup = mixup
        self.freqmask = freqmask
        self.gain = gain
        self.crop_mode = crop_mode
        self.crop_rate = crop_rate

    def do_additional_crop(self, img):
        len_img = img.shape[1]
        img_new = np.zeros([img.shape[0], self.crop], np.float32
)
        rate = np.random.random() * (1 - self.crop_rate) + self.
crop_rate
        if np.random.random() < 0.5: rate = 1

        if img.shape[1] <= self.crop:
            len_crop = int(img.shape[1] * rate)
            if img.shape[1] - len_crop == 0:
                shift_crop = 0
            else:
                shift_crop = np.random.randint(0, img.shape[1] -
 len_crop)
            img = img[:, shift_crop:shift_crop + len_crop]
            if self.crop - len_crop == 0:
                shift = 0
            else:
                shift = np.random.randint(0, self.crop - len_cro
p)
            img_new[:, shift:shift + len_crop] = img
        else:
            shift = np.random.randint(0, img.shape[1] - self.cro
p)
            img_new = img[:, shift:shift + self.crop]
            len_crop = int(self.crop * rate)
            if self.crop - len_crop == 0:
```

```
                shift_crop = 0
            else:
                shift_crop = np.random.randint(0, self.crop - le
n_crop)
            img_new[:shift_crop] = 0
            img_new[shift_crop + len_crop:] = 0
        return img_new

    def do_random_crop(self, img):
        img_new = np.zeros([img.shape[0], self.crop], np.float32
)
        if img.shape[1] < self.crop:
            shift = np.random.randint(0, self.crop - img.shape[1
])
            img_new[:, shift:shift + img.shape[1]] = img
        elif img.shape[1] == self.crop:
            img_new = img
        else:
            shift = np.random.randint(0, img.shape[1] - self.cro
p)
            img_new = img[:, shift:shift + self.crop]
        return img_new

    def do_crop(self, img):
        if self.crop_mode == 'random':
            return self.do_random_crop(img)
        elif self.crop_mode == 'additional':
            return self.do_additional_crop(img)
        elif self.crop_mode == 'original':
            return img

    def do_mixup(self, img, label, alpha=1.):
        idx = np.random.randint(0, len(self.X))
        img2 = np.load("{}.npy".format(self.X[idx][:-4]))
        img2 = self.do_crop(img2)

        label2 = self.y[idx].astype(np.float32)

        rate = np.random.beta(alpha, alpha)
        img = img * rate + img2 * (1 - rate)
        label = label * rate + label2 * (1 - rate)
        return img, label


    def do_freqmask(self, img, max=32):
        coord = np.random.randint(0, img.shape[0])
        width = np.random.randint(8, max)
        cut = np.array([coord - width, coord + width])
        cut = np.clip(cut, 0, img.shape[0])
        img[cut[0]:cut[1]] = 0
        return img

    def do_gain(self, img, max=0.1):
        rate = 1 - max + np.random.random() * max * 2
        return img * rate
```

```
    def __getitem__(self, index):
        img = np.load("{}.npy".format(self.X[index][:-4]))
        img = self.do_crop(img)
        label = self.y[index].astype(np.float32)

        if self.mixup and np.random.random() < 0.5:
            img, label = self.do_mixup(img, label)
        if self.gain and np.random.random() < 0.5:
            img = self.do_gain(img)
        if self.freqmask and np.random.random() < 0.5:
            img = self.do_freqmask(img)

        img = librosa.power_to_db(img)
        img = (img - img.mean()) / (img.std() + 1e-7)
        img = img.reshape([1, img.shape[0], img.shape[1]])

        return img, label

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


def compute_gain(sound, fs, min_db=-80.0, mode='RMSE'):
    if fs == 16000:
        n_fft = 2048
    elif fs == 44100:
        n_fft = 4096
    else:
        raise Exception('Invalid fs {}'.format(fs))
    stride = n_fft // 2

    gain = []
    for i in range(0, len(sound) - n_fft + 1, stride):
        if mode == 'RMSE':
            g = np.mean(sound[i: i + n_fft] ** 2)
        elif mode == 'A_weighting':
            spec = np.fft.rfft(np.hanning(n_fft + 1)[:-1] * soun
d[i: i + n_fft])
            power_spec = np.abs(spec) ** 2
            a_weighted_spec = power_spec * np.power(10, a_weight
(fs, n_fft) / 10)
            g = np.sum(a_weighted_spec)
        else:
            raise Exception('Invalid mode {}'.format(mode))
        gain.append(g)

    gain = np.array(gain)
    gain = np.maximum(gain, np.power(10, min_db / 10))
    gain_db = 10 * np.log10(gain)

    return gain_db


def mix(sound1, sound2, r, fs):
    gain1 = np.max(compute_gain(sound1, fs))  # Decibel
    gain2 = np.max(compute_gain(sound2, fs))
```

```
    t = 1.0 / (1 + np.power(10, (gain1 - gain2) / 20.) * (1 - r)
 / r)
    sound = ((sound1 * t + sound2 * (1 - t)) / np.sqrt(t ** 2 +
(1 - t) ** 2))
    sound = sound.astype(np.float32)

    return sound


class WaveDataset(Dataset):
    def __init__(self, X, y,
                 crop=-1, crop_mode='original', padding=0,
                 mixup=False, scaling=-1, gain=-1,
                 fs=44100,
                 ):
        self.X = X
        self.y = y
        self.crop = crop
        self.crop_mode = crop_mode
        self.padding = padding
        self.mixup = mixup
        self.scaling = scaling
        self.gain = gain
        self.fs = fs

    def preprocess(self, sound):
        for f in self.preprocess_funcs:
            sound = f(sound)

        return sound

    def do_padding(self, snd):
        snd_new = np.pad(snd, self.padding, 'constant')
        return snd_new

    def do_crop(self, snd):
        if self.crop_mode=='random':
            shift = np.random.randint(0, snd.shape[0] - self.cro
p)
            snd_new = snd[shift:shift + self.crop]
        else:
            snd_new = snd
        return snd_new

    def do_gain(self, snd):
        snd_new = snd * np.power(10, random.uniform(-self.gain,
self.gain) / 20.0)
        return snd_new

    def do_scaling(self, snd, interpolate='Nearest'):
        scale = np.power(self.scaling, random.uniform(-1, 1))
        output_size = int(len(snd) * scale)
        ref = np.arange(output_size) / scale
        if interpolate == 'Linear':
            ref1 = ref.astype(np.int32)
            ref2 = np.minimum(ref1+1, len(snd)-1)
```

```
            r = ref - ref1
            snd_new = snd[ref1] * (1-r) + snd[ref2] * r
        elif interpolate == 'Nearest':
            snd_new = snd[ref.astype(np.int32)]
        else:
            raise Exception('Invalid interpolation mode {}'.form
at(interpolate))

        return snd_new

    def do_mixup(self, snd, label, alpha=1):
        idx2 = np.random.randint(0, len(self.X))
        _, snd2 = wavfile.read("{}".format(self.X[idx2]))
        label2 = self.y[idx2].astype(np.float32)
        if self.scaling!=-1:
            snd2 = self.do_scaling(snd2)
        snd2 = self.do_padding(snd2)
        snd2 = self.do_crop(snd2)

        rate = np.random.beta(alpha, alpha)
        snd_new = mix(snd, snd, rate, self.fs)
        label_new = label * rate + label2 * (1 - rate)
        return snd_new, label_new

    def __getitem__(self, index):
        _, snd = wavfile.read("{}".format(self.X[index]))
        label = self.y[index].astype(np.float32)
        if self.scaling!=-1:
            snd = self.do_scaling(snd)
        snd = self.do_padding(snd)
        snd = self.do_crop(snd)
        if self.mixup:
            snd, label = self.do_mixup(snd, label)
        if self.gain!=-1:
            snd = self.do_gain(snd)
        snd = snd.reshape([1, 1, -1]).astype(np.float32) / 32768
.0
        return snd, label

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


def _one_sample_positive_class_precisions(scores, truth):
    """Calculate precisions for each true class for a single sam
ple.

    Args:
      scores: np.array of (num_classes,) giving the individual c
lassifier scores.
      truth: np.array of (num_classes,) bools indicating which c
lasses are true.

    Returns:
      pos_class_indices: np.array of indices of the true classes
 for this sample.
```

```
      pos_class_precisions: np.array of precisions corresponding
 to each of those
        classes.
    """
    num_classes = scores.shape[0]
    pos_class_indices = np.flatnonzero(truth > 0)
    # Only calculate precisions if there are some true classes.
    if not len(pos_class_indices):
        return pos_class_indices, np.zeros(0)
    # Retrieval list of classes for this sample.
    retrieved_classes = np.argsort(scores)[::-1]
    # class_rankings[top_scoring_class_index] == 0 etc.
    class_rankings = np.zeros(num_classes, dtype=np.int)
    class_rankings[retrieved_classes] = range(num_classes)
    # Which of these is a true label?
    retrieved_class_true = np.zeros(num_classes, dtype=np.bool)
    retrieved_class_true[class_rankings[pos_class_indices]] = Tr
ue
    # Num hits for every truncated retrieval list.
    retrieved_cumulative_hits = np.cumsum(retrieved_class_true)
    # Precision of retrieval list truncated at each hit, in orde
r of pos_labels.
    precision_at_hits = (
            retrieved_cumulative_hits[class_rankings[pos_class_i
ndices]] /
            (1 + class_rankings[pos_class_indices].astype(np.flo
at)))
    return pos_class_indices, precision_at_hits


# All-in-one calculation of per-class lwlrap.

def calculate_per_class_lwlrap(truth, scores):
    """Calculate label-weighted label-ranking average precision.

    Arguments:
      truth: np.array of (num_samples, num_classes) giving boole
an ground-truth
        of presence of that class in that sample.
      scores: np.array of (num_samples, num_classes) giving the
classifier-under-
        test's real-valued score for each class for each sample.

    Returns:
      per_class_lwlrap: np.array of (num_classes,) giving the lw
lrap for each
        class.
      weight_per_class: np.array of (num_classes,) giving the pr
ior of each
        class within the truth labels.  Then the overall unbalan
ced lwlrap is
        simply np.sum(per_class_lwlrap * weight_per_class)
    """
    assert truth.shape == scores.shape
    num_samples, num_classes = scores.shape
    # Space to store a distinct precision value for each class o
```

```
n each sample.
    # Only the classes that are true for each sample will be fil
led in.
    precisions_for_samples_by_classes = np.zeros((num_samples, n
um_classes))
    for sample_num in range(num_samples):
        pos_class_indices, precision_at_hits = (
            _one_sample_positive_class_precisions(scores[sample_
num, :],
                                                   truth[sample_n
um, :]))
        precisions_for_samples_by_classes[sample_num, pos_class_
indices] = (
            precision_at_hits)
    labels_per_class = np.sum(truth > 0, axis=0)
    weight_per_class = labels_per_class / float(np.sum(labels_pe
r_class))
    # Form average of each column, i.e. all the precisions assig
ned to labels in
    # a particular class.
    per_class_lwlrap = (np.sum(precisions_for_samples_by_classes
, axis=0) /
                        np.maximum(1, labels_per_class))
    # overall_lwlrap = simple average of all the actual per-clas
s, per-sample precisions
    #                = np.sum(precisions_for_samples_by_classes)
 / np.sum(precisions_for_samples_by_classes > 0)
    #           also = weighted mean of per-class lwlraps, weigh
ted by class label prior across samples
    #                = np.sum(per_class_lwlrap * weight_per_clas
s)
    return per_class_lwlrap, weight_per_class
```