```
1 Makefile
2 Dockerfile
3 make_folds.py
4 train_folds.py
5 predict_folds.py
6 ensemble_pipeline.sh
7 kernel_template.py
8 train_stacking.py
9 stacking_val_predict.py
10 stacking_random_search.py
11 stacking_predict.py
12 stacking_kernel_template.py
13 random_search.py
14 make_fol
15 build_kernel.py
====src:
16
    101 after_train_folds.py
     27 blend_kernel_template.py
     74 blend_predict.py
     70 build_kernel.py
     74 corrections.json
      0 data
     43 Dockerfile
      1 empty.txt
     57 ensemble_pipeline.sh
     31 kernel_template.py
     20 LICENSE
     21 Makefile
     27 make_folds.py
   8745 noisy_corrections.json
    151 predict_folds.py
    126 random_search.py
      0 readme_images
    188 README.md
      0 src
     26 stacking_kernel_template.py
    124 stacking_predict.py
    126 stacking_random_search.py
     99 stacking_val_predict.py
    146 train_folds.py
    109 train_stacking.py
  10386 total
======

1 Makefile
```

```
NAME=argus-freesound

.PHONY: all build stop run

all: stop build run

build:
	docker build -t $(NAME) .

stop:
	-docker stop $(NAME)
	-docker rm $(NAME)

run:
	nvidia-docker run --rm -it \
		--net=host \
		--ipc=host \
		-v $(shell pwd):/workdir \
		--name=$(NAME) \
		$(NAME) \
		bash
======
2 Dockerfile
2 Dockerfile

FROM nvidia/cuda:10.0-cudnn7-devel-ubuntu18.04

RUN apt-get update &&\
    apt-get -y install build-essential yasm nasm cmake
unzip git wget \
    sysstat libtcmalloc-minimal4 pkgconf autoconf
libtool \
    python3 python3-pip python3-dev python3-setuptools \
    libsm6 libxext6 libxrender1 &&\
    ln -s /usr/bin/python3 /usr/bin/python &&\
    ln -s /usr/bin/pip3 /usr/bin/pip &&\
    apt-get clean &&\
    apt-get autoremove &&\
    rm -rf /var/lib/apt/lists/* &&\
    rm -rf /var/cache/apt/archives/*

RUN pip3 install --no-cache-dir numpy==1.16.2

# Install PyTorch
RUN pip3 install https://download.pytorch.org/whl/cu100/
```

```
torch-1.0.1.post2-cp36-cp36m-linux_x86_64.whl &&\
    pip3 install torchvision==0.2.2 &&\
    rm -rf ~/.cache/pip

# Install python ML packages
RUN pip3 install --no-cache-dir \
    opencv-python==3.4.2.17 \
    scipy==1.2.1 \
    matplotlib==3.0.3 \
    pandas==0.24.1 \
    jupyter==1.0.0 \
    scikit-learn==0.20.2 \
    scikit-image==0.14.2 \
    librosa==0.6.3 \
    pytorch-argus==0.0.8

RUN git clone https://github.com/NVIDIA/apex &&\
    cd apex &&\
    git checkout 855808f &&\
    pip install -v --no-cache-dir --global-option="--
cpp_ext" --global-option="--cuda_ext" . &&\
    cd .. && rm -rf apex

ENV PYTHONPATH $PYTHONPATH:/workdir
ENV TORCH_HOME=/workdir/data/.torch

WORKDIR /workdir
======
3 make_folds.py
import random
import numpy as np
import pandas as pd

from sklearn.model_selection import KFold

from src import config


if __name__ == '__main__':
    random_state = 42

    random.seed(random_state)
    np.random.seed(random_state)

    train_curated_df =
pd.read_csv(config.train_curated_csv_path)
```

```
    train_curated_df['fold'] = -1
    file_paths = train_curated_df.fname.apply(lambda x:
config.train_curated_dir / x)
    train_curated_df['file_path'] = file_paths

    kf = KFold(n_splits=config.n_folds,
random_state=random_state, shuffle=True)

    for fold, (_, val_index) in
enumerate(kf.split(train_curated_df)):
        train_curated_df.iloc[val_index, 2] = fold

    train_curated_df.to_csv(config.train_folds_path,
index=False)
    print(f"Train folds saved to
'{config.train_folds_path}'")
======
4 train_folds.py
import json
import argparse

from argus.callbacks import MonitorCheckpoint, \
    EarlyStopping, LoggingToFile, ReduceLROnPlateau

from torch.utils.data import DataLoader

from src.datasets import FreesoundDataset,
FreesoundNoisyDataset, RandomDataset
from src.datasets import get_corrected_noisy_data,
FreesoundCorrectedNoisyDataset
from src.mixers import RandomMixer, AddMixer,
SigmoidConcatMixer, UseMixerWithProb
from src.transforms import get_transforms
from src.argus_models import FreesoundModel
from src.utils import load_noisy_data, load_folds_data
from src import config


parser = argparse.ArgumentParser()
parser.add_argument('--experiment', required=True,
type=str)
args = parser.parse_args()

BATCH_SIZE = 128
CROP_SIZE = 256
DATASET_SIZE = 128 * 256
```

```
NOISY_PROB = 0.01
CORR_NOISY_PROB = 0.42
MIXER_PROB = 0.8
WRAP_PAD_PROB = 0.5
CORRECTIONS = True
if config.kernel:
    NUM_WORKERS = 2
else:
    NUM_WORKERS = 8
SAVE_DIR = config.experiments_dir / args.experiment
PARAMS = {
    'nn_module': ('AuxSkipAttention', {
        'num_classes': len(config.classes),
        'base_size': 64,
        'dropout': 0.4,
        'ratio': 16,
        'kernel_size': 7,
        'last_filters': 8,
        'last_fc': 4
    }),
    'loss': ('OnlyNoisyLSoftLoss', {
        'beta': 0.7,
        'noisy_weight': 0.5,
        'curated_weight': 0.5
    }),
    'optimizer': ('Adam', {'lr': 0.0009}),
    'device': 'cuda',
    'aux': {
        'weights': [1.0, 0.4, 0.2, 0.1]
    },
    'amp': {
        'opt_level': 'O2',
        'keep_batchnorm_fp32': True,
        'loss_scale': "dynamic"
    }
}


def train_fold(save_dir, train_folds, val_folds,
               folds_data, noisy_data,
corrected_noisy_data):
    train_transfrom = get_transforms(train=True,
                                     size=CROP_SIZE,

wrap_pad_prob=WRAP_PAD_PROB,
                                     resize_scale=(0.8,
```

```
1.0),
                                            resize_ratio=(1.7,
2.3),
                                            resize_prob=0.33,
                                            spec_num_mask=2,

spec_freq_masking=0.15,

spec_time_masking=0.20,
                                            spec_prob=0.5)

    mixer = RandomMixer([
        SigmoidConcatMixer(sigmoid_range=(3, 12)),
        AddMixer(alpha_dist='uniform')
    ], p=[0.6, 0.4])
    mixer = UseMixerWithProb(mixer, prob=MIXER_PROB)

    curated_dataset = FreesoundDataset(folds_data,
train_folds,

transform=train_transfrom,
                                              mixer=mixer)
    noisy_dataset = FreesoundNoisyDataset(noisy_data,

transform=train_transfrom,
                                                 mixer=mixer)
    corr_noisy_dataset =
FreesoundCorrectedNoisyDataset(corrected_noisy_data,

transform=train_transfrom,

mixer=mixer)
    dataset_probs = [NOISY_PROB, CORR_NOISY_PROB, 1 -
NOISY_PROB - CORR_NOISY_PROB]
    print("Dataset probs", dataset_probs)
    print("Dataset lens", len(noisy_dataset),
len(corr_noisy_dataset), len(curated_dataset))
    train_dataset = RandomDataset([noisy_dataset,
corr_noisy_dataset, curated_dataset],
                                 p=dataset_probs,
                                 size=DATASET_SIZE)

    val_dataset = FreesoundDataset(folds_data, val_folds,
                                  get_transforms(False,
CROP_SIZE))
    train_loader = DataLoader(train_dataset,
```

```
batch_size=BATCH_SIZE,
                              shuffle=True,
drop_last=True,
                              num_workers=NUM_WORKERS)
    val_loader = DataLoader(val_dataset,
batch_size=BATCH_SIZE * 2,
                            shuffle=False,
num_workers=NUM_WORKERS)

    model = FreesoundModel(PARAMS)

    callbacks = [
        MonitorCheckpoint(save_dir,
monitor='val_lwlrap', max_saves=1),
        ReduceLROnPlateau(monitor='val_lwlrap',
patience=6, factor=0.6, min_lr=1e-8),
        EarlyStopping(monitor='val_lwlrap', patience=18),
        LoggingToFile(save_dir / 'log.txt'),
    ]

    model.fit(train_loader,
              val_loader=val_loader,
              max_epochs=700,
              callbacks=callbacks,
              metrics=['multi_accuracy', 'lwlrap'])


if __name__ == "__main__":
    if not SAVE_DIR.exists():
        SAVE_DIR.mkdir(parents=True, exist_ok=True)
    else:
        print(f"Folder {SAVE_DIR} already exists.")

    with open(SAVE_DIR / 'source.py', 'w') as outfile:
        outfile.write(open(__file__).read())

    print("Model params", PARAMS)
    with open(SAVE_DIR / 'params.json', 'w') as outfile:
        json.dump(PARAMS, outfile)

    folds_data =
load_folds_data(use_corrections=CORRECTIONS)
    noisy_data = load_noisy_data()
    corrected_noisy_data = get_corrected_noisy_data()

    for fold in config.folds:
```

```
        val_folds = [fold]
        train_folds = list(set(config.folds) -
set(val_folds))
        save_fold_dir = SAVE_DIR / f'fold_{fold}'
        print(f"Val folds: {val_folds}, Train folds:
{train_folds}")
        print(f"Fold save dir {save_fold_dir}")
        train_fold(save_fold_dir, train_folds, val_folds,
                   folds_data, noisy_data,
corrected_noisy_data)
======
5 predict_folds.py
import json
import argparse
import numpy as np
import pandas as pd

from src.predictor import Predictor
from src.audio import read_as_melspectrogram
from src.transforms import get_transforms
from src.metrics import LwlrapBase
from src.utils import get_best_model_path,
gmean_preds_blend
from src.datasets import get_test_data
from src import config


parser = argparse.ArgumentParser()
parser.add_argument('--experiment', required=True,
type=str)
args = parser.parse_args()


EXPERIMENT_DIR = config.experiments_dir / args.experiment
PREDICTION_DIR = config.predictions_dir / args.experiment
DEVICE = 'cuda'
CROP_SIZE = 256
BATCH_SIZE = 16


def pred_val_fold(predictor, fold):
    fold_prediction_dir = PREDICTION_DIR /
f'fold_{fold}' / 'val'
    fold_prediction_dir.mkdir(parents=True,
exist_ok=True)
```

```
    train_folds_df = pd.read_csv(config.train_folds_path)
    train_folds_df = train_folds_df[train_folds_df.fold
== fold]

    fname_lst = []
    pred_lst = []
    for i, row in train_folds_df.iterrows():
        image = read_as_melspectrogram(row.file_path)
        pred = predictor.predict(image)

        pred_path = fold_prediction_dir /
f'{row.fname}.npy'
        np.save(pred_path, pred)

        pred = pred.mean(axis=0)
        pred_lst.append(pred)
        fname_lst.append(row.fname)

    preds = np.stack(pred_lst, axis=0)
    probs_df = pd.DataFrame(data=preds,
                            index=fname_lst,
                            columns=config.classes)
    probs_df.index.name = 'fname'
    probs_df.to_csv(fold_prediction_dir / 'probs.csv')


def pred_test_fold(predictor, fold, test_data):
    fold_prediction_dir = PREDICTION_DIR /
f'fold_{fold}' / 'test'
    fold_prediction_dir.mkdir(parents=True,
exist_ok=True)

    fname_lst, images_lst = test_data
    pred_lst = []
    for fname, image in zip(fname_lst, images_lst):
        pred = predictor.predict(image)

        pred_path = fold_prediction_dir / f'{fname}.npy'
        np.save(pred_path, pred)

        pred = pred.mean(axis=0)
        pred_lst.append(pred)

    preds = np.stack(pred_lst, axis=0)
    subm_df = pd.DataFrame(data=preds,
                           index=fname_lst,
```

```
                              columns=config.classes)
    subm_df.index.name = 'fname'
    subm_df.to_csv(fold_prediction_dir / 'probs.csv')


def blend_test_predictions():
    probs_df_lst = []
    for fold in config.folds:
        fold_probs_path = PREDICTION_DIR /
f'fold_{fold}' / 'test' / 'probs.csv'
        probs_df = pd.read_csv(fold_probs_path)
        probs_df.set_index('fname', inplace=True)
        probs_df_lst.append(probs_df)

    blend_df = gmean_preds_blend(probs_df_lst)

    if config.kernel:
        blend_df.to_csv('submission.csv')
    else:
        blend_df.to_csv(PREDICTION_DIR / 'probs.csv')


def calc_lwlrap_on_val():
    probs_df_lst = []
    for fold in config.folds:
        fold_probs_path = PREDICTION_DIR /
f'fold_{fold}' / 'val' / 'probs.csv'
        probs_df = pd.read_csv(fold_probs_path)
        probs_df.set_index('fname', inplace=True)
        probs_df_lst.append(probs_df)

    probs_df = pd.concat(probs_df_lst, axis=0)
    train_curated_df =
pd.read_csv(config.train_curated_csv_path)

    lwlrap = LwlrapBase(config.classes)
    for i, row in train_curated_df.iterrows():
        target = np.zeros(len(config.classes))
        for label in row.labels.split(','):
            target[config.class2index[label]] = 1.

        pred = probs_df.loc[row.fname].values
        lwlrap.accumulate(target[np.newaxis],
pred[np.newaxis])

    result = {
```

```
        'overall_lwlrap': lwlrap.overall_lwlrap(),
        'per_class_lwlrap': {cls: lwl for cls, lwl in
zip(config.classes,

lwlrap.per_class_lwlrap())}
    }
    print(result)
    with open(PREDICTION_DIR / 'val_lwlrap.json', 'w')
as file:
        json.dump(result, file, indent=2)


if __name__ == "__main__":
    transforms = get_transforms(False, CROP_SIZE)
    test_data = get_test_data()

    for fold in config.folds:
        print("Predict fold", fold)
        fold_dir = EXPERIMENT_DIR / f'fold_{fold}'
        model_path = get_best_model_path(fold_dir)
        print("Model path", model_path)
        predictor = Predictor(model_path, transforms,
                              BATCH_SIZE,
                              (config.audio.n_mels,
CROP_SIZE),
                              (config.audio.n_mels,
CROP_SIZE//4),
                              device=DEVICE)

        if not config.kernel:
            print("Val predict")
            pred_val_fold(predictor, fold)

        print("Test predict")
        pred_test_fold(predictor, fold, test_data)

    print("Blend folds predictions")
    blend_test_predictions()

    if not config.kernel:
        print("Calculate lwlrap metric on cv")
        calc_lwlrap_on_val()
======
6 ensemble_pipeline.py
#!/usr/bin/env bash
set -e
```

```
NAME="argus-freesound"
DOCKER_OPTIONS="--rm -it --ipc=host -v $(pwd):/workdir --
name=${NAME} ${NAME}"

git checkout master
docker build -t ${NAME} .

# Build kernel
git checkout ddbe02ae88b6bd05c1b9726d2fd30c38854be4fd
nvidia-docker run ${DOCKER_OPTIONS} python
build_kernel.py

# Make folds split
nvidia-docker run ${DOCKER_OPTIONS} python make_folds.py

# Experiment auxiliary_016
git checkout 31156c79e470ffacc494ba846aef3bd80faf0d10
nvidia-docker run ${DOCKER_OPTIONS} python
train_folds.py --experiment auxiliary_016

# Experiment auxiliary_019
git checkout 9639288b9240e7e45db497feb7593f05a4f463d1
nvidia-docker run ${DOCKER_OPTIONS} python
train_folds.py --experiment auxiliary_019

# Experiment corr_noisy_003
git checkout 1fb2eea443d99df4538420fa42daf098c94322c2
nvidia-docker run ${DOCKER_OPTIONS} python
train_folds.py --experiment corr_noisy_003

# Experiment corr_noisy_004
git checkout db945ac11df559e0e1c0a2be464faf46122f1bef
nvidia-docker run ${DOCKER_OPTIONS} python
train_folds.py --experiment corr_noisy_004

# Experiment corr_noisy_007
git checkout bdb9150146ad8d500b4e19fa6b9fe98111fb28b0
nvidia-docker run ${DOCKER_OPTIONS} python
train_folds.py --experiment corr_noisy_007

# Experiment corrections_002
git checkout 05a7aee7c50148677735531bdddf32902b468bea
nvidia-docker run ${DOCKER_OPTIONS} python
train_folds.py --experiment corrections_002

# Experiment corrections_003
```

```
git checkout 24a4f20ffc284d22b38bbabfe510ed194f62e496
nvidia-docker run ${DOCKER_OPTIONS} python
train_folds.py --experiment corrections_003


# Experiment stacking_008_fcnet_43040
git checkout 1e1c265fc6e45c103d8d741c1bdcc5959f71348d
nvidia-docker run ${DOCKER_OPTIONS} python
train_stacking.py

# Stacking train stacking_008_fcnet_45041
git checkout bc48f8a17ac4452ee3f2a3d18fd7caa31f812b27
nvidia-docker run ${DOCKER_OPTIONS} python
train_stacking.py

# Stacking train stacking_008_fcnet_50013
git checkout 493908aeaff4b0e1df8298003b10af1cf56e6b3c
nvidia-docker run ${DOCKER_OPTIONS} python
train_stacking.py

git checkout master
======
7 kernel_template.py

import gzip
import base64
import os
from pathlib import Path
from typing import Dict

EXPERIMENT_NAME = 'corr_noisy_007'
KERNEL_MODE = "predict"  # "train" or "predict"

# this is base64 encoded source code
file_data: Dict = {file_data}


for path, encoded in file_data.items():
    print(path)
    path = Path(path)
    path.parent.mkdir(parents=True, exist_ok=True)

path.write_bytes(gzip.decompress(base64.b64decode(encoded)))


def run(command):
```

```
    os.system('export PYTHONPATH=${PYTHONPATH}:/kaggle/
working && '
              f'export MODE={KERNEL_MODE} && ' + command)


run('python make_folds.py')
if KERNEL_MODE == "train":
    run(f'python train_folds.py --experiment
{EXPERIMENT_NAME}')
else:
    run(f'python predict_folds.py --experiment
{EXPERIMENT_NAME}')
run('rm -rf argus src')
======
8train_stacking.py
import json

from argus.callbacks import MonitorCheckpoint, \
    EarlyStopping, LoggingToFile, ReduceLROnPlateau

from torch.utils.data import DataLoader

from src.stacking.datasets import get_out_of_folds_data,
StackingDataset
from src.stacking.transforms import get_transforms
from src.stacking.argus_models import StackingModel
from src import config


STACKING_EXPERIMENT = "stacking_008_fcnet_50013"

EXPERIMENTS = [
    'auxiliary_016',
    'auxiliary_019',
    'corr_noisy_003',
    'corr_noisy_004',
    'corr_noisy_007',
    'corrections_002',
    'corrections_003'
]
RS_PARAMS = {"base_size": 512, "reduction_scale": 1,
"p_dropout": 0.1662788540244386, "lr":
2.5814932060476834e-05,
             "patience": 7, "factor":
0.5537460438294733, "batch_size": 128}
BATCH_SIZE = RS_PARAMS['batch_size']
```

```
DATASET_SIZE = 128 * 256
CORRECTIONS = True
if config.kernel:
    NUM_WORKERS = 2
else:
    NUM_WORKERS = 8
SAVE_DIR = config.experiments_dir / STACKING_EXPERIMENT
PARAMS = {
    'nn_module': ('FCNet', {
        'in_channels': len(config.classes) *
len(EXPERIMENTS),
        'num_classes': len(config.classes),
        'base_size': RS_PARAMS['base_size'],
        'reduction_scale': RS_PARAMS['reduction_scale'],
        'p_dropout': RS_PARAMS['p_dropout']
    }),
    'loss': 'BCEWithLogitsLoss',
    'optimizer': ('Adam', {'lr': RS_PARAMS['lr']}),
    'device': 'cuda',
}


def train_fold(save_dir, train_folds, val_folds,
folds_data):
    train_dataset = StackingDataset(folds_data,
train_folds,
                                    get_transforms(True),
                                    DATASET_SIZE)
    val_dataset = StackingDataset(folds_data, val_folds,
                                  get_transforms(False))

    train_loader = DataLoader(train_dataset,
batch_size=BATCH_SIZE,
                              shuffle=True,
drop_last=True,
                              num_workers=NUM_WORKERS)
    val_loader = DataLoader(val_dataset,
batch_size=BATCH_SIZE * 2,
                            shuffle=False,
num_workers=NUM_WORKERS)

    model = StackingModel(PARAMS)

    callbacks = [
        MonitorCheckpoint(save_dir,
monitor='val_lwlrap', max_saves=1),
```

```
        ReduceLROnPlateau(monitor='val_lwlrap',
                          patience=RS_PARAMS['patience'],
                          factor=RS_PARAMS['factor'],
                          min_lr=1e-8),
        EarlyStopping(monitor='val_lwlrap', patience=30),
        LoggingToFile(save_dir / 'log.txt'),
    ]

    model.fit(train_loader,
              val_loader=val_loader,
              max_epochs=700,
              callbacks=callbacks,
              metrics=['multi_accuracy', 'lwlrap'])


if __name__ == "__main__":
    if not SAVE_DIR.exists():
        SAVE_DIR.mkdir(parents=True, exist_ok=True)
    else:
        print(f"Folder {SAVE_DIR} already exists.")

    with open(SAVE_DIR / 'source.py', 'w') as outfile:
        outfile.write(open(__file__).read())

    print("Model params", PARAMS)
    with open(SAVE_DIR / 'params.json', 'w') as outfile:
        json.dump(PARAMS, outfile)

    if CORRECTIONS:
        with open(config.corrections_json_path) as file:
            corrections = json.load(file)
        print("Corrections:", corrections)
    else:
        corrections = None

    folds_data = get_out_of_folds_data(EXPERIMENTS,
corrections)

    for fold in config.folds:
        val_folds = [fold]
        train_folds = list(set(config.folds) -
set(val_folds))
        save_fold_dir = SAVE_DIR / f'fold_{fold}'
        print(f"Val folds: {val_folds}, Train folds:
{train_folds}")
        print(f"Fold save dir {save_fold_dir}")
```

```
        train_fold(save_fold_dir, train_folds,
val_folds, folds_data)
======
9 stacking_val_predict.py

import json
import numpy as np
import pandas as pd

from src.stacking.datasets import load_fname_probs
from src.stacking.predictor import StackPredictor
from src.metrics import LwlrapBase
from src.utils import get_best_model_path
from src import config


STACKING_EXPERIMENT = "stacking_008_fcnet_50013"

EXPERIMENTS = [
    'auxiliary_016',
    'auxiliary_019',
    'corr_noisy_003',
    'corr_noisy_004',
    'corr_noisy_007',
    'corrections_002',
    'corrections_003'
]

EXPERIMENT_DIR = config.experiments_dir /
STACKING_EXPERIMENT
PREDICTION_DIR = config.predictions_dir /
STACKING_EXPERIMENT
DEVICE = 'cuda'
BATCH_SIZE = 256


def pred_val_fold(predictor, fold):
    fold_prediction_dir = PREDICTION_DIR /
f'fold_{fold}' / 'val'
    fold_prediction_dir.mkdir(parents=True,
exist_ok=True)

    train_folds_df = pd.read_csv(config.train_folds_path)
    train_folds_df = train_folds_df[train_folds_df.fold
== fold]
```

```
    fname_lst = []
    probs_lst = []
    for i, row in train_folds_df.iterrows():
        probs = load_fname_probs(EXPERIMENTS, fold,
row.fname)

        probs_lst.append(probs.mean(axis=0))
        fname_lst.append(row.fname)

    stack_probs = np.stack(probs_lst, axis=0)
    preds = predictor.predict(stack_probs)

    probs_df = pd.DataFrame(data=list(preds),
                            index=fname_lst,
                            columns=config.classes)
    probs_df.index.name = 'fname'
    probs_df.to_csv(fold_prediction_dir / 'probs.csv')


def calc_lwlrap_on_val():
    probs_df_lst = []
    for fold in config.folds:
        fold_probs_path = PREDICTION_DIR /
f'fold_{fold}' / 'val' / 'probs.csv'
        probs_df = pd.read_csv(fold_probs_path)
        probs_df.set_index('fname', inplace=True)
        probs_df_lst.append(probs_df)

    probs_df = pd.concat(probs_df_lst, axis=0)
    train_curated_df =
pd.read_csv(config.train_curated_csv_path)

    lwlrap = LwlrapBase(config.classes)
    for i, row in train_curated_df.iterrows():
        target = np.zeros(len(config.classes))
        for label in row.labels.split(','):
            target[config.class2index[label]] = 1.

        pred = probs_df.loc[row.fname].values
        lwlrap.accumulate(target[np.newaxis],
pred[np.newaxis])

    result = {
        'overall_lwlrap': lwlrap.overall_lwlrap(),
        'per_class_lwlrap': {cls: lwl for cls, lwl in
zip(config.classes,
```

```
lwlrap.per_class_lwlrap())}
    }
    print(result)
    with open(PREDICTION_DIR / 'val_lwlrap.json', 'w')
as file:
        json.dump(result, file, indent=2)


if __name__ == "__main__":
    for fold in config.folds:
        print("Predict fold", fold)
        fold_dir = EXPERIMENT_DIR / f'fold_{fold}'
        model_path = get_best_model_path(fold_dir)
        print("Model path", model_path)
        predictor = StackPredictor(model_path,
                                   BATCH_SIZE,
                                   device=DEVICE)

        print("Val predict")
        pred_val_fold(predictor, fold)

    print("Calculate lwlrap metric on cv")
    calc_lwlrap_on_val()
======
10 stacking_random_search.py
10 stacking_random_search.py
10 stacking_random_search.py
10 stacking_random_search.py
10 stacking_random_search.py
10 stacking_random_search.py
10 stacking_random_search.py
10 stacking_random_search.py
10 stacking_random_search.py
10 stacking_random_search.py

import json
import time
import torch
import random
import numpy as np
from pprint import pprint

from argus.callbacks import MonitorCheckpoint, \
    EarlyStopping, LoggingToFile, ReduceLROnPlateau
```

```
from torch.utils.data import DataLoader

from src.stacking.datasets import get_out_of_folds_data,
StackingDataset
from src.stacking.transforms import get_transforms
from src.stacking.argus_models import StackingModel
from src import config

EXPERIMENT_NAME = 'fcnet_stacking_rs_004'
START_FROM = 0
EXPERIMENTS = [
    'auxiliary_007',
    'auxiliary_010',
    'auxiliary_012',
    'auxiliary_014'
]
DATASET_SIZE = 128 * 256
CORRECTIONS = True
if config.kernel:
    NUM_WORKERS = 2
else:
    NUM_WORKERS = 4
SAVE_DIR = config.experiments_dir / EXPERIMENT_NAME


def train_folds(save_dir, folds_data):
    random_params = {
        'base_size': int(np.random.choice([64, 128, 256,
512])),
        'reduction_scale': int(np.random.choice([2, 4,
8, 16])),
        'p_dropout': float(np.random.uniform(0.0, 0.5)),
        'lr': float(np.random.uniform(0.0001, 0.00001)),
        'patience': int(np.random.randint(3, 12)),
        'factor': float(np.random.uniform(0.5, 0.8)),
        'batch_size': int(np.random.choice([32, 64,
128])),
    }
    pprint(random_params)

    save_dir.mkdir(parents=True, exist_ok=True)
    with open(save_dir / 'random_params.json', 'w') as
outfile:
        json.dump(random_params, outfile)

    params = {
```

```
        'nn_module': ('FCNet', {
            'in_channels': len(config.classes) *
len(EXPERIMENTS),
            'num_classes': len(config.classes),
            'base_size': random_params['base_size'],
            'reduction_scale':
random_params['reduction_scale'],
            'p_dropout': random_params['p_dropout']
        }),
        'loss': 'BCEWithLogitsLoss',
        'optimizer': ('Adam', {'lr':
random_params['lr']}),
        'device': 'cuda',
    }

    for fold in config.folds:
        val_folds = [fold]
        train_folds = list(set(config.folds) -
set(val_folds))
        save_fold_dir = save_dir / f'fold_{fold}'
        print(f"Val folds: {val_folds}, Train folds:
{train_folds}")
        print(f"Fold save dir {save_fold_dir}")

        train_dataset = StackingDataset(folds_data,
train_folds,

get_transforms(True),
                                        DATASET_SIZE)
        val_dataset = StackingDataset(folds_data,
val_folds,

get_transforms(False))

        train_loader = DataLoader(train_dataset,

batch_size=random_params['batch_size'],
                                  shuffle=True,
drop_last=True,

num_workers=NUM_WORKERS)
        val_loader = DataLoader(val_dataset,

batch_size=random_params['batch_size'] * 2,
                                shuffle=False,
num_workers=NUM_WORKERS)
```

```
        model = StackingModel(params)

        callbacks = [
            MonitorCheckpoint(save_fold_dir,
monitor='val_lwlrap', max_saves=1),
            ReduceLROnPlateau(monitor='val_lwlrap',

patience=random_params['patience'],

factor=random_params['factor'],
                              min_lr=1e-8),
            EarlyStopping(monitor='val_lwlrap',
patience=20),
            LoggingToFile(save_fold_dir / 'log.txt'),
        ]

        model.fit(train_loader,
                  val_loader=val_loader,
                  max_epochs=300,
                  callbacks=callbacks,
                  metrics=['multi_accuracy', 'lwlrap'])


if __name__ == "__main__":
    SAVE_DIR.mkdir(parents=True, exist_ok=True)
    with open(SAVE_DIR / 'source.py', 'w') as outfile:
        outfile.write(open(__file__).read())

    if CORRECTIONS:
        with open(config.corrections_json_path) as file:
            corrections = json.load(file)
        print("Corrections:", corrections)
    else:
        corrections = None

    folds_data = get_out_of_folds_data(EXPERIMENTS,
corrections)

    for num in range(START_FROM, 10000):
        np.random.seed(num)
        random.seed(num)

        save_dir = SAVE_DIR / f'{num:04}'
        train_folds(save_dir, folds_data)
        time.sleep(5.0)
```

```
        torch.cuda.empty_cache()
        time.sleep(5.0)
======
11 stacking_predict.py
import numpy as np
import pandas as pd
from scipy.stats.mstats import gmean

from src.predictor import Predictor
from src.transforms import get_transforms
from src.utils import get_best_model_path
from src.datasets import get_test_data
from src import config

from src.stacking.predictor import StackPredictor


NAME = "stacking_008"

EXPERIMENTS = [
    'auxiliary_016',
    'auxiliary_019',
    'corr_noisy_003',
    'corr_noisy_004',
    'corr_noisy_007',
    'corrections_002',
    'corrections_003'
]

STACKING_EXPERIMENTS = [
    'stacking_008_fcnet_43040',
    'stacking_008_fcnet_45041',
    'stacking_008_fcnet_50013'
]

DEVICE = 'cuda'
CROP_SIZE = 256
BATCH_SIZE = 16
STACK_BATCH_SIZE = 256
TILE_STEP = 2


def pred_test(predictor, images_lst):
    pred_lst = []
    for image in images_lst:
        pred = predictor.predict(image)
```

```
        pred = pred.mean(axis=0)
        pred_lst.append(pred)

    preds = np.stack(pred_lst, axis=0)
    return preds


def experiment_pred(experiment_dir, images_lst):
    print(f"Start predict: {experiment_dir}")
    transforms = get_transforms(False, CROP_SIZE)

    pred_lst = []
    for fold in config.folds:
        print("Predict fold", fold)
        fold_dir = experiment_dir / f'fold_{fold}'
        model_path = get_best_model_path(fold_dir)
        print("Model path", model_path)
        predictor = Predictor(model_path, transforms,
                              BATCH_SIZE,
                              (config.audio.n_mels,
CROP_SIZE),
                              (config.audio.n_mels,
CROP_SIZE//TILE_STEP),
                              device=DEVICE)

        pred = pred_test(predictor, images_lst)
        pred_lst.append(pred)

    preds = gmean(pred_lst, axis=0)
    return preds


def stacking_pred(experiment_dir, stack_probs):
    print(f"Start predict: {experiment_dir}")

    pred_lst = []
    for fold in config.folds:
        print("Predict fold", fold)
        fold_dir = experiment_dir / f'fold_{fold}'
        model_path = get_best_model_path(fold_dir)
        print("Model path", model_path)
        predictor = StackPredictor(model_path,
STACK_BATCH_SIZE,
                                   device=DEVICE)
        pred = predictor.predict(stack_probs)
```

```
        pred_lst.append(pred)

    preds = gmean(pred_lst, axis=0)
    return preds


if __name__ == "__main__":
    print("Name", NAME)
    print("Experiments", EXPERIMENTS)
    print("Stacking experiments", STACKING_EXPERIMENTS)
    print("Device", DEVICE)
    print("Crop size", CROP_SIZE)
    print("Batch size", BATCH_SIZE)
    print("Stacking batch size", STACK_BATCH_SIZE)
    print("Tile step", TILE_STEP)

    fname_lst, images_lst = get_test_data()

    exp_pred_lst = []
    for experiment in EXPERIMENTS:
        experiment_dir = config.experiments_dir /
experiment
        exp_pred = experiment_pred(experiment_dir,
images_lst)
        exp_pred_lst.append(exp_pred)

    stack_probs = np.concatenate(exp_pred_lst, axis=1)

    stack_pred_lst = []
    for experiment in STACKING_EXPERIMENTS:
        experiment_dir = config.experiments_dir /
experiment
        stack_pred = stacking_pred(experiment_dir,
stack_probs)
        stack_pred_lst.append(stack_pred)

    stack_pred = gmean(exp_pred_lst + stack_pred_lst,
axis=0)

    stack_pred_df = pd.DataFrame(data=stack_pred,
                                 index=fname_lst,
                                 columns=config.classes)
    stack_pred_df.index.name = 'fname'
    stack_pred_df.to_csv('submission.csv')
======
12 stacking_kernel_template.py
```

```
import gzip
import base64
import os
from pathlib import Path
from typing import Dict

KERNEL_MODE = "predict"

# this is base64 encoded source code
file_data: Dict = {file_data}


for path, encoded in file_data.items():
    print(path)
    path = Path(path)
    path.parent.mkdir(parents=True, exist_ok=True)

path.write_bytes(gzip.decompress(base64.b64decode(encoded)))


def run(command):
    os.system('export PYTHONPATH=${PYTHONPATH}:/kaggle/
working && '
              f'export MODE={KERNEL_MODE} && ' + command)


run('python stacking_predict.py')
run('rm -rf argus src')
======
13random_search.py

import torch
import numpy as np
import random
import json
import time
from pprint import pprint

from argus.callbacks import MonitorCheckpoint, \
    EarlyStopping, LoggingToFile, ReduceLROnPlateau

from torch.utils.data import DataLoader

from src.datasets import FreesoundDataset,
CombinedDataset, FreesoundNoisyDataset
from src.transforms import get_transforms
```

```
from src.argus_models import FreesoundModel
from src.utils import load_folds_data, load_noisy_data
from src import config


EXPERIMENT_NAME = 'noisy_lsoft_rs_002'
VAL_FOLDS = [0]
TRAIN_FOLDS = [1, 2, 3, 4]
BATCH_SIZE = 128
CROP_SIZE = 128
DATASET_SIZE = 128 * 256
if config.kernel:
    NUM_WORKERS = 2
else:
    NUM_WORKERS = 8
SAVE_DIR = config.experiments_dir / EXPERIMENT_NAME
START_FROM = 0


def train_experiment(folds_data, noisy_data, num):
    experiment_dir = SAVE_DIR / f'{num:04}'
    np.random.seed(num)
    random.seed(num)

    random_params = {
        'p_dropout': float(np.random.uniform(0.1, 0.3)),
        'batch_size': int(np.random.choice([128])),
        'lr': float(np.random.choice([0.001, 0.0006,
0.0003])),
        'add_prob': float(np.random.uniform(0.0, 1.0)),
        'noisy_prob': float(np.random.uniform(0.0, 1.0)),
        'lsoft_beta': float(np.random.uniform(0.2, 0.8)),
        'noisy_weight': float(np.random.uniform(0.3,
0.7)),
        'patience': int(np.random.randint(2, 10)),
        'factor': float(np.random.uniform(0.5, 0.8))
    }
    pprint(random_params)

    params = {
        'nn_module': ('SimpleKaggle', {
            'num_classes': len(config.classes),
            'dropout': random_params['p_dropout'],
            'base_size': 64
        }),
        'loss': ('OnlyNoisyLSoftLoss', {
```

```
            'beta': random_params['lsoft_beta'],
            'noisy_weight':
random_params['noisy_weight'],
            'curated_weight': 1 -
random_params['noisy_weight']
        }),
        'optimizer': ('Adam', {'lr':
random_params['lr']}),
        'device': 'cuda',
        'amp': {
            'opt_level': 'O2',
            'keep_batchnorm_fp32': True,
            'loss_scale': "dynamic"
        }
    }
    pprint(params)
    try:
        train_transfrom = get_transforms(True, CROP_SIZE)
        curated_dataset = FreesoundDataset(folds_data,
TRAIN_FOLDS,

transform=train_transfrom,

add_prob=random_params['add_prob'])
        noisy_dataset = FreesoundNoisyDataset(noisy_data,

transform=train_transfrom)
        train_dataset = CombinedDataset(noisy_dataset,
curated_dataset,

noisy_prob=random_params['noisy_prob'],

size=DATASET_SIZE)

        val_dataset = FreesoundDataset(folds_data,
VAL_FOLDS,

get_transforms(False, CROP_SIZE))
        train_loader = DataLoader(train_dataset,
batch_size=random_params['batch_size'],
                                  shuffle=True,
drop_last=True,

num_workers=NUM_WORKERS)
        val_loader = DataLoader(val_dataset,
batch_size=random_params['batch_size'] * 2,
```

```
                                shuffle=False,
num_workers=NUM_WORKERS)

        model = FreesoundModel(params)

        callbacks = [
            MonitorCheckpoint(experiment_dir,
monitor='val_lwlrap', max_saves=1),
            ReduceLROnPlateau(monitor='val_lwlrap',

patience=random_params['patience'],

factor=random_params['factor'],
                              min_lr=1e-8),
            EarlyStopping(monitor='val_lwlrap',
patience=20),
            LoggingToFile(experiment_dir / 'log.txt'),
        ]

        with open(experiment_dir / 'random_params.json',
'w') as outfile:
            json.dump(random_params, outfile)

        model.fit(train_loader,
                  val_loader=val_loader,
                  max_epochs=100,
                  callbacks=callbacks,
                  metrics=['multi_accuracy', 'lwlrap'])
    except KeyboardInterrupt as e:
        raise e
    except BaseException as e:
        print(f"Exception '{e}' with random params
'{random_params}'")


if __name__ == "__main__":
    print("Start load train data")
    noisy_data = load_noisy_data()
    folds_data = load_folds_data()

    for i in range(START_FROM, 10000):
        train_experiment(folds_data, noisy_data, i)
        time.sleep(5.0)
        torch.cuda.empty_cache()
        time.sleep(5.0)
======
```

```
14 make_fol

import random
import numpy as np
import pandas as pd

from sklearn.model_selection import KFold

from src import config


if __name__ == '__main__':
    random_state = 42

    random.seed(random_state)
    np.random.seed(random_state)

    train_curated_df =
pd.read_csv(config.train_curated_csv_path)
    train_curated_df['fold'] = -1
    file_paths = train_curated_df.fname.apply(lambda x:
config.train_curated_dir / x)
    train_curated_df['file_path'] = file_paths

    kf = KFold(n_splits=config.n_folds,
random_state=random_state, shuffle=True)

    for fold, (_, val_index) in
enumerate(kf.split(train_curated_df)):
        train_curated_df.iloc[val_index, 2] = fold

    train_curated_df.to_csv(config.train_folds_path,
index=False)
    print(f"Train folds saved to
'{config.train_folds_path}'")
======
15 build_kernel.py

#!/usr/bin/env python3
# Kaggle script build system template: https://
github.com/lopuhin/kaggle-script-template
import os
import base64
import gzip
from pathlib import Path
```

```
IGNORE_LIST = ["data", "build"]

PACKAGES = [
    'https://github.com/lRomul/argus.git'
]


def encode_file(path: Path) -> str:
    compressed = gzip.compress(path.read_bytes(),
compresslevel=9)
    return base64.b64encode(compressed).decode('utf-8')


def check_ignore(path: Path, ignore_list):
    if not path.is_file():
        return False
    for ignore in ignore_list:
        if str(path).startswith(ignore):
            return False
    return True


def clone_package(git_url):
    name = Path(git_url).stem
    os.system('mkdir -p tmp')
    os.system(f'rm -rf tmp/{name}')
    os.system(f'cd tmp && git clone {git_url}')
    os.system(f'cp -R tmp/{name}/{name} .')
    os.system(f'rm -rf tmp/{name}')


def build_script(ignore_list, packages,
template_name='kernel_template.py'):
    to_encode = []

    for path in Path('.').glob('**/*.py'):
        if check_ignore(path, ignore_list + packages):
            to_encode.append(path)

    for package in packages:
        clone_package(package)
        package_name = Path(package).stem
        for path in Path(package_name).glob('**/*'):
            if check_ignore(path, ignore_list):
                to_encode.append(path)
```

```
    file_data = {str(path): encode_file(path) for path
in to_encode}
    print("Encoded python files:")
    for path in file_data:
        print(path)
    template = Path(template_name).read_text('utf8')
    (Path('kernel') / template_name).write_text(
        template.replace('{file_data}', str(file_data)),
        encoding='utf8')


if __name__ == '__main__':
    os.system('rm -rf kernel && mkdir kernel')
    build_script(IGNORE_LIST, PACKAGES,
                 template_name='kernel_template.py')
    build_script(IGNORE_LIST, PACKAGES,

template_name='blend_kernel_template.py')
    build_script(IGNORE_LIST, PACKAGES,

template_name='stacking_kernel_template.py')
======
```

```
1    120 argus_models.py
2     79 audio.py
3    173 config.py
4    295 datasets.py
5      2 __init__.py
6    134 losses.py
7     83 lr_scheduler.py
8    144 metrics.py
9     78 mixers.py
10     0 models
11    47 predictor.py
12   162 random_resized_crop.py
13     0 stacking
14   251 tiles.py
15   243 transforms.py
16   129 utils.py
17  1940 total

1 argus_models.py
======

import torch

from argus import Model
from argus.utils import deep_detach, deep_to

from src.models import resnet
from src.models import senet
from src.models.feature_extractor import FeatureExtractor
from src.models.simple_kaggle import SimpleKaggle
from src.models.simple_attention import SimpleAttention
from src.models.skip_attention import SkipAttention
from src.models.aux_skip_attention import
AuxSkipAttention
from src.models.rnn_aux_skip_attention import
RnnAuxSkipAttention
from src.losses import OnlyNoisyLqLoss,
OnlyNoisyLSoftLoss, BCEMaxOutlierLoss
from src import config


class FreesoundModel(Model):
    nn_module = {
        'resnet18': resnet.resnet18,
        'resnet34': resnet.resnet34,
        'FeatureExtractor': FeatureExtractor,
```

```
        'SimpleKaggle': SimpleKaggle,
        'se_resnext50_32x4d': senet.se_resnext50_32x4d,
        'SimpleAttention': SimpleAttention,
        'SkipAttention': SkipAttention,
        'AuxSkipAttention': AuxSkipAttention,
        'RnnAuxSkipAttention': RnnAuxSkipAttention
    }
    loss = {
        'OnlyNoisyLqLoss': OnlyNoisyLqLoss,
        'OnlyNoisyLSoftLoss': OnlyNoisyLSoftLoss,
        'BCEMaxOutlierLoss': BCEMaxOutlierLoss
    }
    prediction_transform = torch.nn.Sigmoid

    def __init__(self, params):
        super().__init__(params)

        if 'aux' in params:
            self.aux_weights = params['aux']['weights']
        else:
            self.aux_weights = None

        self.use_amp = not config.kernel and 'amp' in
params
        if self.use_amp:
            from apex import amp
            self.amp = amp
            self.nn_module, self.optimizer =
self.amp.initialize(
                self.nn_module, self.optimizer,
                opt_level=params['amp']['opt_level'],
                keep_batchnorm_fp32=params['amp']
['keep_batchnorm_fp32'],
                loss_scale=params['amp']['loss_scale']
            )

    def prepare_batch(self, batch, device):
        input, target, noisy = batch
        input = deep_to(input, device, non_blocking=True)
        target = deep_to(target, device,
non_blocking=True)
        noisy = deep_to(noisy, device, non_blocking=True)
        return input, target, noisy

    def train_step(self, batch)-> dict:
        if not self.nn_module.training:
```

```
            self.nn_module.train()
        self.optimizer.zero_grad()
        input, target, noisy = self.prepare_batch(batch,
self.device)
        prediction = self.nn_module(input)
        if self.aux_weights is not None:
            loss = 0
            for pred, weight in zip(prediction,
self.aux_weights):
                loss += self.loss(pred, target, noisy) *
weight
        else:
            loss = self.loss(prediction, target, noisy)
        if self.use_amp:
            with self.amp.scale_loss(loss,
self.optimizer) as scaled_loss:
                scaled_loss.backward()
        else:
            loss.backward()
        self.optimizer.step()

        prediction = deep_detach(prediction)
        target = deep_detach(target)
        return {
            'prediction':
self.prediction_transform(prediction[0]),
            'target': target,
            'loss': loss.item(),
            'noisy': noisy
        }

    def val_step(self, batch) -> dict:
        if self.nn_module.training:
            self.nn_module.eval()
        with torch.no_grad():
            input, target, noisy =
self.prepare_batch(batch, self.device)
            prediction = self.nn_module(input)
            if self.aux_weights is not None:
                loss = 0
                for pred, weight in zip(prediction,
self.aux_weights):
                    loss += self.loss(pred, target,
noisy) * weight
            else:
                loss = self.loss(prediction, target,
```

```
noisy)
            return {
                'prediction':
self.prediction_transform(prediction[0]),
                'target': target,
                'loss': loss.item(),
                'noisy': noisy
            }

    def predict(self, input):
        assert self.predict_ready()
        with torch.no_grad():
            if self.nn_module.training:
                self.nn_module.eval()
            input = deep_to(input, self.device)
            prediction = self.nn_module(input)
            if self.aux_weights is not None:
                prediction = prediction[0]
            prediction =
self.prediction_transform(prediction)
            return prediction
======
2 audio.py2
# Source: https://www.kaggle.com/daisukelab/creating-
fat2019-preprocessed-data
import numpy as np

import librosa
import librosa.display

from src.config import audio as config


def get_audio_config():
    return config.get_config_dict()


def read_audio(file_path):
    min_samples = int(config.min_seconds *
config.sampling_rate)
    try:
        y, sr = librosa.load(file_path,
sr=config.sampling_rate)
        trim_y, trim_idx = librosa.effects.trim(y)  #
trim, top_db=default(60)
```

```
        if len(trim_y) < min_samples:
            center = (trim_idx[1] - trim_idx[0]) // 2
            left_idx = max(0, center - min_samples // 2)
            right_idx = min(len(y), center +
min_samples // 2)
            trim_y = y[left_idx:right_idx]

            if len(trim_y) < min_samples:
                padding = min_samples - len(trim_y)
                offset = padding // 2
                trim_y = np.pad(trim_y, (offset, padding
- offset), 'constant')
        return trim_y
    except BaseException as e:
        print(f"Exception while reading file {e}")
        return np.zeros(min_samples, dtype=np.float32)


def audio_to_melspectrogram(audio):
    spectrogram = librosa.feature.melspectrogram(audio,

sr=config.sampling_rate,

n_mels=config.n_mels,

hop_length=config.hop_length,

n_fft=config.n_fft,

fmin=config.fmin,

fmax=config.fmax)
    spectrogram = librosa.power_to_db(spectrogram)
    spectrogram = spectrogram.astype(np.float32)
    return spectrogram


def show_melspectrogram(mels, title='Log-frequency power
spectrogram'):
    import matplotlib.pyplot as plt

    librosa.display.specshow(mels, x_axis='time',
y_axis='mel',
                             sr=config.sampling_rate,
hop_length=config.hop_length,
                             fmin=config.fmin,
```

```
fmax=config.fmax)
    plt.colorbar(format='%+2.0f dB')
    plt.title(title)
    plt.show()


def read_as_melspectrogram(file_path, time_stretch=1.0,
pitch_shift=0.0,
                           debug_display=False):
    x = read_audio(file_path)
    if time_stretch != 1.0:
        x = librosa.effects.time_stretch(x, time_stretch)

    if pitch_shift != 0.0:
        librosa.effects.pitch_shift(x,
config.sampling_rate, n_steps=pitch_shift)

    mels = audio_to_melspectrogram(x)
    if debug_display:
        import IPython
        IPython.display.display(IPython.display.Audio(x,
rate=config.sampling_rate))
        show_melspectrogram(mels)
    return mels


if __name__ == "__main__":
    x =
read_as_melspectrogram(config.train_curated_dir /
'0b9906f7.wav')
    print(x.shape)
======
3 config.py
import os
import json
from pathlib import Path
from hashlib import sha1


kernel = False
kernel_mode = ""
if 'MODE' in os.environ:
    kernel = True
    kernel_mode = os.environ['MODE']
    assert kernel_mode in ["train", "predict"]
```

```
if kernel:
    if kernel_mode == "train":
        input_data_dir = Path('/kaggle/input/')
    else:
        input_data_dir = Path('/kaggle/input/freesound-
audio-tagging-2019/')
    save_data_dir = Path('/kaggle/working/')
else:
    input_data_dir = Path('/workdir/data/')
    save_data_dir = Path('/workdir/data/')

train_curated_dir = input_data_dir / 'train_curated'
train_noisy_dir = input_data_dir / 'train_noisy'
train_curated_csv_path = input_data_dir /
'train_curated.csv'
train_noisy_csv_path = input_data_dir / 'train_noisy.csv'
test_dir = input_data_dir / 'test'
sample_submission = input_data_dir /
'sample_submission.csv'

train_folds_path = save_data_dir / 'train_folds.csv'
predictions_dir = save_data_dir / 'predictions'
if kernel and kernel_mode == "predict":
    def find_kernel_data_dir():
        kaggle_input = Path('/kaggle/input/')
        train_kernel_name = 'freesound-train'
        default = kaggle_input / train_kernel_name
        if default.exists():
            return default
        else:
            for path in kaggle_input.glob('*'):
                if path.is_dir():
                    if
path.name.startswith(train_kernel_name):
                        return path
        return default
    experiments_dir = find_kernel_data_dir() /
'experiments'
else:
    experiments_dir = save_data_dir / 'experiments'

folds_data_pkl_dir = save_data_dir / 'folds_data'
augment_folds_data_pkl_dir = save_data_dir /
'augment_folds_data'
noisy_data_pkl_dir = save_data_dir / 'noisy_data'
corrections_json_path = Path('/workdir/corrections.json')
```

```
noisy_corrections_json_path = Path('/workdir/
noisy_corrections.json')

n_folds = 5
folds = list(range(n_folds))


class audio:
    sampling_rate = 44100
    hop_length = 345 * 2
    fmin = 20
    fmax = sampling_rate // 2
    n_mels = 128
    n_fft = n_mels * 20
    min_seconds = 0.5

    @classmethod
    def get_config_dict(cls):
        config_dict = dict()
        for key, value in cls.__dict__.items():
            if key[:1] != '_' and \
                    key not in ['get_config_dict',
'get_hash']:
                config_dict[key] = value
        return config_dict

    @classmethod
    def get_hash(cls, **kwargs):
        config_dict = cls.get_config_dict()
        config_dict = {**config_dict, **kwargs}
        hash_str = json.dumps(config_dict,
                              sort_keys=True,
                              ensure_ascii=False,
                              separators=None)
        hash_str = hash_str.encode('utf-8')
        return sha1(hash_str).hexdigest()[:7]


classes = [
    'Accelerating_and_revving_and_vroom',
    'Accordion',
    'Acoustic_guitar',
    'Applause',
    'Bark',
    'Bass_drum',
    'Bass_guitar',
```

```
    'Bathtub_(filling_or_washing)',
    'Bicycle_bell',
    'Burping_and_eructation',
    'Bus',
    'Buzz',
    'Car_passing_by',
    'Cheering',
    'Chewing_and_mastication',
    'Child_speech_and_kid_speaking',
    'Chink_and_clink',
    'Chirp_and_tweet',
    'Church_bell',
    'Clapping',
    'Computer_keyboard',
    'Crackle',
    'Cricket',
    'Crowd',
    'Cupboard_open_or_close',
    'Cutlery_and_silverware',
    'Dishes_and_pots_and_pans',
    'Drawer_open_or_close',
    'Drip',
    'Electric_guitar',
    'Fart',
    'Female_singing',
    'Female_speech_and_woman_speaking',
    'Fill_(with_liquid)',
    'Finger_snapping',
    'Frying_(food)',
    'Gasp',
    'Glockenspiel',
    'Gong',
    'Gurgling',
    'Harmonica',
    'Hi-hat',
    'Hiss',
    'Keys_jangling',
    'Knock',
    'Male_singing',
    'Male_speech_and_man_speaking',
    'Marimba_and_xylophone',
    'Mechanical_fan',
    'Meow',
    'Microwave_oven',
    'Motorcycle',
    'Printer',
```

```
    'Purr',
    'Race_car_and_auto_racing',
    'Raindrop',
    'Run',
    'Scissors',
    'Screaming',
    'Shatter',
    'Sigh',
    'Sink_(filling_or_washing)',
    'Skateboard',
    'Slam',
    'Sneeze',
    'Squeak',
    'Stream',
    'Strum',
    'Tap',
    'Tick-tock',
    'Toilet_flush',
    'Traffic_noise_and_roadway_noise',
    'Trickle_and_dribble',
    'Walk_and_footsteps',
    'Water_tap_and_faucet',
    'Waves_and_surf',
    'Whispering',
    'Writing',
    'Yell',
    'Zipper_(clothing)'
]

class2index = {cls: idx for idx, cls in
enumerate(classes)}
======
4 datase.py
import json
import time
import torch
import random
import numpy as np
import pandas as pd
from functools import partial
import multiprocessing as mp
from torch.utils.data import Dataset

from src.audio import read_as_melspectrogram,
get_audio_config
from src import config
```

```
N_WORKERS = mp.cpu_count()


def get_test_data():
    print("Start load test data")
    fname_lst = []
    wav_path_lst = []
    for wav_path in
sorted(config.test_dir.glob('*.wav')):
        wav_path_lst.append(wav_path)
        fname_lst.append(wav_path.name)

    with mp.Pool(N_WORKERS) as pool:
        images_lst = pool.map(read_as_melspectrogram,
wav_path_lst)

    return fname_lst, images_lst


def get_folds_data(corrections=None):
    print("Start generate folds data")
    print("Audio config", get_audio_config())
    train_folds_df = pd.read_csv(config.train_folds_path)

    audio_paths_lst = []
    targets_lst = []
    folds_lst = []
    for i, row in train_folds_df.iterrows():
        labels = row.labels

        if corrections is not None:
            if row.fname in corrections:
                action = corrections[row.fname]
                if action == 'remove':
                    print(f"Skip {row.fname}")
                    continue
                else:
                    print(f"Replace labels {row.fname}
from {labels} to {action}")
                    labels = action

        folds_lst.append(row.fold)
        audio_paths_lst.append(row.file_path)
        target = torch.zeros(len(config.classes))
```

```
        for label in labels.split(','):
            target[config.class2index[label]] = 1.
        targets_lst.append(target)

    with mp.Pool(N_WORKERS) as pool:
        images_lst = pool.map(read_as_melspectrogram,
audio_paths_lst)

    return images_lst, targets_lst, folds_lst


def get_augment_folds_data_generator(time_stretch_lst,
pitch_shift_lst):
    print("Start generate augment folds data")
    print("Audio config", get_audio_config())
    print("time_stretch_lst:", time_stretch_lst)
    print("pitch_shift_lst:", pitch_shift_lst)
    train_folds_df = pd.read_csv(config.train_folds_path)

    audio_paths_lst = []
    targets_lst = []
    folds_lst = []
    for i, row in train_folds_df.iterrows():
        folds_lst.append(row.fold)
        audio_paths_lst.append(row.file_path)
        target = torch.zeros(len(config.classes))
        for label in row.labels.split(','):
            target[config.class2index[label]] = 1.
        targets_lst.append(target)

    with mp.Pool(N_WORKERS) as pool:
        images_lst = pool.map(read_as_melspectrogram,
audio_paths_lst)

    yield images_lst, targets_lst, folds_lst
    images_lst = []

    for pitch_shift in pitch_shift_lst:
        pitch_shift_read =
partial(read_as_melspectrogram, pitch_shift=pitch_shift)
        with mp.Pool(N_WORKERS) as pool:
            images_lst = pool.map(pitch_shift_read,
audio_paths_lst)

        yield images_lst, targets_lst, folds_lst
        images_lst = []
```

```
    for time_stretch in time_stretch_lst:
        time_stretch_read =
partial(read_as_melspectrogram,
time_stretch=time_stretch)
        with mp.Pool(N_WORKERS) as pool:
            images_lst = pool.map(time_stretch_read,
audio_paths_lst)

        yield images_lst, targets_lst, folds_lst
        images_lst = []


class FreesoundDataset(Dataset):
    def __init__(self, folds_data, folds,
                 transform=None,
                 mixer=None):
        super().__init__()
        self.folds = folds
        self.transform = transform
        self.mixer = mixer

        self.images_lst = []
        self.targets_lst = []
        for img, trg, fold in zip(*folds_data):
            if fold in folds:
                self.images_lst.append(img)
                self.targets_lst.append(trg)

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

    def __getitem__(self, idx):
        image = self.images_lst[idx].copy()
        target = self.targets_lst[idx].clone()

        if self.transform is not None:
            image = self.transform(image)

        if self.mixer is not None:
            image, target = self.mixer(self, image,
target)

        noisy = torch.tensor(0, dtype=torch.uint8)
        return image, target, noisy
```

```
def get_noisy_data_generator():
    print("Start generate noisy data")
    print("Audio config", get_audio_config())
    train_noisy_df =
pd.read_csv(config.train_noisy_csv_path)

    with open(config.noisy_corrections_json_path) as
file:
        corrections = json.load(file)

    audio_paths_lst = []
    targets_lst = []
    for i, row in train_noisy_df.iterrows():
        labels = row.labels

        if row.fname in corrections:
            action = corrections[row.fname]
            if action == 'remove':
                continue
            else:
                labels = action

        audio_paths_lst.append(config.train_noisy_dir /
row.fname)
        target = torch.zeros(len(config.classes))
        for label in labels.split(','):
            target[config.class2index[label]] = 1.
        targets_lst.append(target)

        if len(audio_paths_lst) >= 5000:
            with mp.Pool(N_WORKERS) as pool:
                images_lst =
pool.map(read_as_melspectrogram, audio_paths_lst)

            yield images_lst, targets_lst

            audio_paths_lst = []
            images_lst = []
            targets_lst = []

    with mp.Pool(N_WORKERS) as pool:
        images_lst = pool.map(read_as_melspectrogram,
audio_paths_lst)

    yield images_lst, targets_lst
```

```
class FreesoundNoisyDataset(Dataset):
    def __init__(self, noisy_data, transform=None,
                 mixer=None):
        super().__init__()
        self.transform = transform
        self.mixer = mixer

        self.images_lst = []
        self.targets_lst = []
        for img, trg in zip(*noisy_data):
            self.images_lst.append(img)
            self.targets_lst.append(trg)

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

    def __getitem__(self, idx):
        image = self.images_lst[idx].copy()
        target = self.targets_lst[idx].clone()

        if self.transform is not None:
            image = self.transform(image)

        if self.mixer is not None:
            image, target = self.mixer(self, image,
target)

        noisy = torch.tensor(1, dtype=torch.uint8)
        return image, target, noisy


class RandomDataset(Dataset):
    def __init__(self, datasets, p=None, size=4096):
        self.datasets = datasets
        self.p = p
        self.size = size

    def __len__(self):
        return self.size

    def __getitem__(self, idx):
        seed = int(time.time() * 1000.0) + idx
        random.seed(seed)
        np.random.seed(seed % (2**31))
```

```
        dataset_idx = np.random.choice(
            range(len(self.datasets)), p=self.p)
        dataset = self.datasets[dataset_idx]
        idx = random.randint(0, len(dataset) - 1)
        return dataset[idx]


def get_corrected_noisy_data():
    print("Start generate corrected noisy data")
    print("Audio config", get_audio_config())
    train_noisy_df =
pd.read_csv(config.train_noisy_csv_path)

    with open(config.noisy_corrections_json_path) as
file:
        corrections = json.load(file)

    audio_paths_lst = []
    targets_lst = []
    for i, row in train_noisy_df.iterrows():
        labels = row.labels

        if row.fname in corrections:
            action = corrections[row.fname]
            if action == 'remove':
                continue
            else:
                labels = action
        else:
            continue

        audio_paths_lst.append(config.train_noisy_dir /
row.fname)
        target = torch.zeros(len(config.classes))

        for label in labels.split(','):
            target[config.class2index[label]] = 1.
        targets_lst.append(target)

    with mp.Pool(N_WORKERS) as pool:
        images_lst = pool.map(read_as_melspectrogram,
audio_paths_lst)

    return images_lst, targets_lst
```

```
class FreesoundCorrectedNoisyDataset(Dataset):
    def __init__(self, noisy_data, transform=None,
                 mixer=None):
        super().__init__()
        self.transform = transform
        self.mixer = mixer

        self.images_lst = []
        self.targets_lst = []
        for img, trg in zip(*noisy_data):
            self.images_lst.append(img)
            self.targets_lst.append(trg)

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

    def __getitem__(self, idx):
        image = self.images_lst[idx].copy()
        target = self.targets_lst[idx].clone()

        if self.transform is not None:
            image = self.transform(image)

        if self.mixer is not None:
            image, target = self.mixer(self, image,
target)

        noisy = torch.tensor(0, dtype=torch.uint8)
        return image, target, noisy
======
5 init

import src.argus_models
import src.metrics
======
6 losses.py

import torch
from torch import nn
import torch.nn.functional as F


def lq_loss(y_pred, y_true, q):
    eps = 1e-7
    loss = y_pred * y_true
```

```
    # loss, _ = torch.max(loss, dim=1)
    loss = (1 - (loss + eps) ** q) / q
    return loss.mean()


class LqLoss(nn.Module):
    def __init__(self, q=0.5):
        super().__init__()
        self.q = q

    def forward(self, output, target):
        output = torch.sigmoid(output)
        return lq_loss(output, target, self.q)


def l_soft(y_pred, y_true, beta):
    eps = 1e-7

    y_pred = torch.clamp(y_pred, eps, 1.0)

    # (1) dynamically update the targets based on the
current state of the model:
    # bootstrapped target tensor
    # use predicted class proba directly to generate
regression targets
    with torch.no_grad():
        y_true_update = beta * y_true + (1 - beta) *
y_pred

    # (2) compute loss as always
    loss = F.binary_cross_entropy(y_pred, y_true_update)
    return loss


class LSoftLoss(nn.Module):
    def __init__(self, beta=0.5):
        super().__init__()
        self.beta = beta

    def forward(self, output, target):
        output = torch.sigmoid(output)
        return l_soft(output, target, self.beta)


class NoisyCuratedLoss(nn.Module):
    def __init__(self, noisy_loss, curated_loss,
```

```
                 noisy_weight=0.5, curated_weight=0.5):
        super().__init__()
        self.noisy_loss = noisy_loss
        self.curated_loss = curated_loss
        self.noisy_weight = noisy_weight
        self.curated_weight = curated_weight

    def forward(self, output, target, noisy):
        batch_size = target.shape[0]

        noisy_indexes = noisy.nonzero().squeeze(1)
        curated_indexes = (noisy ==
0).nonzero().squeeze(1)

        noisy_len = noisy_indexes.shape[0]
        if noisy_len > 0:
            noisy_target = target[noisy_indexes]
            noisy_output = output[noisy_indexes]
            noisy_loss = self.noisy_loss(noisy_output,
noisy_target)
            noisy_loss = noisy_loss * (noisy_len /
batch_size)
        else:
            noisy_loss = 0

        curated_len = curated_indexes.shape[0]
        if curated_len > 0:
            curated_target = target[curated_indexes]
            curated_output = output[curated_indexes]
            curated_loss =
self.curated_loss(curated_output, curated_target)
            curated_loss = curated_loss * (curated_len /
batch_size)
        else:
            curated_loss = 0

        loss = noisy_loss * self.noisy_weight
        loss += curated_loss * self.curated_weight
        return loss


class OnlyNoisyLqLoss(nn.Module):
    def __init__(self, q=0.5,
                 noisy_weight=0.5,
                 curated_weight=0.5):
        super().__init__()
```

```
        lq = LqLoss(q=q)
        bce = nn.BCEWithLogitsLoss()
        self.loss = NoisyCuratedLoss(lq, bce,
                                     noisy_weight,
                                     curated_weight)

    def forward(self, output, target, noisy):
        return self.loss(output, target, noisy)


class OnlyNoisyLSoftLoss(nn.Module):
    def __init__(self, beta,
                 noisy_weight=0.5,
                 curated_weight=0.5):
        super().__init__()
        soft = LSoftLoss(beta)
        bce = nn.BCEWithLogitsLoss()
        self.loss = NoisyCuratedLoss(soft, bce,
                                     noisy_weight,
                                     curated_weight)

    def forward(self, output, target, noisy):
        return self.loss(output, target, noisy)


class BCEMaxOutlierLoss(nn.Module):
    def __init__(self, alpha=0.8):
        super().__init__()
        self.alpha = alpha

    def forward(self, output, target, noisy):
        loss =
F.binary_cross_entropy_with_logits(output, target,

reduction='none')
        loss = loss.mean(dim=1)

        with torch.no_grad():
            outlier_mask = loss > self.alpha * loss.max()
            outlier_mask = outlier_mask * noisy
            outlier_idx = (outlier_mask ==
0).nonzero().squeeze(1)

        loss = loss[outlier_idx].mean()
        return loss
======
```

```
7 lr_scheduler.py
import math
from torch.optim.lr_scheduler import _LRScheduler

from argus.callbacks.lr_schedulers import LRScheduler


class CosineAnnealingWarmRestarts(_LRScheduler):
    r"""Set the learning rate of each parameter group
using a cosine annealing
    schedule, where :math:`\eta_{max}` is set to the
initial lr, :math:`T_{cur}`
    is the number of epochs since the last restart
and :math:`T_{i}` is the number
    of epochs between two warm restarts in SGDR:
    .. math::
        \eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} -
\eta_{min})(1 +
        \cos(\frac{T_{cur}}{T_{i}}\pi))
    When :math:`T_{cur}=T_{i}`, set :math:`\eta_t =
\eta_{min}`.
    When :math:`T_{cur}=0`(after restart),
set :math:`\eta_t=\eta_{max}`.
    It has been proposed in
    `SGDR: Stochastic Gradient Descent with Warm
Restarts`_.
    Args:
        optimizer (Optimizer): Wrapped optimizer.
        T_0 (int): Number of iterations for the first
restart.
        T_mult (int, optional): A factor
increases :math:`T_{i}` after a restart. Default: 1.
        eta_min (float, optional): Minimum learning
rate. Default: 0.
        last_epoch (int, optional): The index of last
epoch. Default: -1.
    .. _SGDR\: Stochastic Gradient Descent with Warm
Restarts:
        https://arxiv.org/abs/1608.03983
    """

    def __init__(self, optimizer, T_0, T_mult=1,
eta_min=0, last_epoch=-1):
        if T_0 <= 0 or not isinstance(T_0, int):
            raise ValueError("Expected positive integer
T_0, but got {}".format(T_0))
```

```
        if T_mult < 1 or not isinstance(T_mult, int):
            raise ValueError("Expected integer T_mult >=
1, but got {}".format(T_mult))
        self.T_0 = T_0
        self.T_i = T_0
        self.T_mult = T_mult
        self.eta_min = eta_min
        super(CosineAnnealingWarmRestarts,
self).__init__(optimizer, last_epoch)
        self.T_cur = last_epoch

    def get_lr(self):
        return [self.eta_min + (base_lr - self.eta_min)
* (1 + math.cos(math.pi * self.T_cur / self.T_i)) / 2
                for base_lr in self.base_lrs]

    def step(self, epoch=None):
        """Step could be called after every update, i.e.
if one epoch has 10 iterations
        (number_of_train_examples / batch_size), we
should call SGDR.step(0.1), SGDR.step(0.2), etc.
        This function can be called in an interleaved
way.
        Example:
            >>> scheduler = SGDR(optimizer, T_0, T_mult)
            >>> for epoch in range(20):
            >>>     scheduler.step()
            >>> scheduler.step(26)
            >>> scheduler.step() # scheduler.step(27),
instead of scheduler(20)
        """
        if epoch is None:
            epoch = self.last_epoch + 1
            self.T_cur = self.T_cur + 1
            if self.T_cur >= self.T_i:
                self.T_cur = self.T_cur - self.T_i
                self.T_i = self.T_i * self.T_mult
        else:
            if epoch >= self.T_0:
                if self.T_mult == 1:
                    self.T_cur = epoch % self.T_0
                else:
                    n = int(math.log((epoch / self.T_0 *
(self.T_mult - 1) + 1), self.T_mult))
                    self.T_cur = epoch - self.T_0 *
(self.T_mult ** n - 1) / (self.T_mult - 1)
```

```
                    self.T_i = self.T_0 * self.T_mult **
(n)
            else:
                self.T_i = self.T_0
                self.T_cur = epoch
        self.last_epoch = math.floor(epoch)
        for param_group, lr in
zip(self.optimizer.param_groups, self.get_lr()):
            param_group['lr'] = lr


class CosineAnnealing(LRScheduler):
    def __init__(self, T_0, T_mult=1, eta_min=0):
        super().__init__(lambda opt:
CosineAnnealingWarmRestarts(opt,

T_0,

T_mult=T_mult,

eta_min=eta_min))
======
8 metrics.py
import torch
import numpy as np

from argus.metrics.metric import Metric

from src import config


class MultiCategoricalAccuracy(Metric):
    name = 'multi_accuracy'
    better = 'max'

    def __init__(self, threshold=0.5):
        self.threshold = threshold

    def reset(self):
        self.correct = 0
        self.count = 0

    def update(self, step_output: dict):
        pred = step_output['prediction']
        trg = step_output['target']
        pred = (pred > self.threshold).to(torch.float32)
```

```
        correct = torch.eq(pred, trg).all(dim=1).view(-1)
        self.correct += torch.sum(correct).item()
        self.count += correct.shape[0]

    def compute(self):
        if self.count == 0:
            raise Exception('Must be at least one
example for computation')
        return self.correct / self.count


# Source: https://github.com/DCASE-REPO/
dcase2019_task2_baseline/blob/master/evaluation.py
class LwlrapBase:
    """Computes label-weighted label-ranked average
precision (lwlrap)."""

    def __init__(self, class_map):
        self.num_classes = 0
        self.total_num_samples = 0
        self._class_map = class_map

    def accumulate(self, batch_truth, batch_scores):
        """Accumulate a new batch of samples into the
metric.
        Args:
          truth: np.array of (num_samples, num_classes)
giving boolean
            ground-truth of presence of that class in
that sample for this batch.
          scores: np.array of (num_samples, num_classes)
giving the
            classifier-under-test's real-valued score
for each class for each
            sample.
        """
        assert batch_scores.shape == batch_truth.shape
        num_samples, num_classes = batch_truth.shape
        if not self.num_classes:
            self.num_classes = num_classes
            self._per_class_cumulative_precision =
np.zeros(self.num_classes)
            self._per_class_cumulative_count =
np.zeros(self.num_classes,

dtype=np.int)
```

```
        assert num_classes == self.num_classes
        for truth, scores in zip(batch_truth,
batch_scores):
            pos_class_indices, precision_at_hits = (

self._one_sample_positive_class_precisions(scores,
truth))

self._per_class_cumulative_precision[pos_class_indices]
+= (
                precision_at_hits)

self._per_class_cumulative_count[pos_class_indices] += 1
        self.total_num_samples += num_samples

    def _one_sample_positive_class_precisions(self,
scores, truth):
        """Calculate precisions for each true class for
a single sample.
        Args:
          scores: np.array of (num_classes,) giving the
individual classifier scores.
          truth: np.array of (num_classes,) bools
indicating which classes 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]]
= True
        # Num hits for every truncated retrieval list.
        retrieved_cumulative_hits =
np.cumsum(retrieved_class_true)
        # Precision of retrieval list truncated at each
hit, in order of pos_labels.
        precision_at_hits = (

retrieved_cumulative_hits[class_rankings[pos_class_indices]] /
                (1 +
class_rankings[pos_class_indices].astype(np.float)))
        return pos_class_indices, precision_at_hits

    def per_class_lwlrap(self):
        """Return a vector of the per-class lwlraps for
the accumulated samples."""
        return (self._per_class_cumulative_precision /
                np.maximum(1,
self._per_class_cumulative_count))

    def per_class_weight(self):
        """Return a normalized weight vector for the
contributions of each class."""
        return (self._per_class_cumulative_count /

float(np.sum(self._per_class_cumulative_count)))

    def overall_lwlrap(self):
        """Return the scalar overall lwlrap for
cumulated samples."""
        return np.sum(self.per_class_lwlrap() *
self.per_class_weight())

    def __str__(self):
        per_class_lwlrap = self.per_class_lwlrap()
        # List classes in descending order of lwlrap.
        s = (['Lwlrap(%s) = %.6f' % (name, lwlrap) for
(lwlrap, name) in
              sorted([(per_class_lwlrap[i],
self._class_map[i]) for i in range(self.num_classes)],
                     reverse=True)])
        s.append('Overall lwlrap = %.6f' %
```

```
(self.overall_lwlrap()))
        return '\n'.join(s)


class Lwlrap(Metric):
    name = 'lwlrap'
    better = 'max'

    def __init__(self, classes=None):
        self.classes = classes
        if self.classes is None:
            self.classes = config.classes

        self.lwlrap = LwlrapBase(self.classes)

    def reset(self):
        self.lwlrap.num_classes = 0
        self.lwlrap.total_num_samples = 0

    def update(self, step_output: dict):
        pred = step_output['prediction'].cpu().numpy()
        trg = step_output['target'].cpu().numpy()
        self.lwlrap.accumulate(trg, pred)

    def compute(self):
        return self.lwlrap.overall_lwlrap()
======
9 miserx.py
import torch
import random
import numpy as np


def get_random_sample(dataset):
    rnd_idx = random.randint(0, len(dataset) - 1)
    rnd_image = dataset.images_lst[rnd_idx].copy()
    rnd_target = dataset.targets_lst[rnd_idx].clone()
    rnd_image = dataset.transform(rnd_image)
    return rnd_image, rnd_target


class AddMixer:
    def __init__(self, alpha_dist='uniform'):
        assert alpha_dist in ['uniform', 'beta']
        self.alpha_dist = alpha_dist
```

```
    def sample_alpha(self):
        if self.alpha_dist == 'uniform':
            return random.uniform(0, 0.5)
        elif self.alpha_dist == 'beta':
            return np.random.beta(0.4, 0.4)

    def __call__(self, dataset, image, target):
        rnd_image, rnd_target =
get_random_sample(dataset)

        alpha = self.sample_alpha()
        image = (1 - alpha) * image + alpha * rnd_image
        target = (1 - alpha) * target + alpha *
rnd_target
        return image, target


class SigmoidConcatMixer:
    def __init__(self, sigmoid_range=(3, 12)):
        self.sigmoid_range = sigmoid_range

    def sample_mask(self, size):
        x_radius = random.randint(*self.sigmoid_range)

        step = (x_radius * 2) / size[1]
        x = np.arange(-x_radius, x_radius, step=step)
        y = torch.sigmoid(torch.from_numpy(x)).numpy()
        mix_mask = np.tile(y, (size[0], 1))
        return
torch.from_numpy(mix_mask.astype(np.float32))

    def __call__(self, dataset, image, target):
        rnd_image, rnd_target =
get_random_sample(dataset)

        mix_mask = self.sample_mask(image.shape[-2:])
        rnd_mix_mask = 1 - mix_mask

        image = mix_mask * image + rnd_mix_mask *
rnd_image
        target = target + rnd_target
        target = np.clip(target, 0.0, 1.0)
        return image, target


class RandomMixer:
```

```
    def __init__(self, mixers, p=None):
        self.mixers = mixers
        self.p = p

    def __call__(self, dataset, image, target):
        mixer = np.random.choice(self.mixers, p=self.p)
        image, target = mixer(dataset, image, target)
        return image, target


class UseMixerWithProb:
    def __init__(self, mixer, prob=.5):
        self.mixer = mixer
        self.prob = prob

    def __call__(self, dataset, image, target):
        if random.random() < self.prob:
            return self.mixer(dataset, image, target)
        return image, target
======
11 predictors.py
import torch
from torch.utils.data import DataLoader

from argus import load_model

from src.tiles import ImageSlicer


@torch.no_grad()
def tile_prediction(model, image, transforms,
                    tile_size, tile_step, batch_size):
    tiler = ImageSlicer(image.shape,
                        tile_size=tile_size,
                        tile_step=tile_step)

    tiles = tiler.split(image, value=float(image.min()))
    tiles = [transforms(tile) for tile in tiles]

    loader = DataLoader(tiles, batch_size=batch_size)

    preds_lst = []

    for tiles_batch in loader:
        pred_batch = model.predict(tiles_batch)
        preds_lst.append(pred_batch)
```

```
    pred = torch.cat(preds_lst, dim=0)

    return pred.cpu().numpy()


class Predictor:
    def __init__(self, model_path, transforms,
                 batch_size, tile_size, tile_step,
                 device='cuda'):
        self.model = load_model(model_path,
device=device)
        self.transforms = transforms
        self.tile_size = tile_size
        self.tile_step = tile_step
        self.batch_size = batch_size

    def predict(self, image):
        pred = tile_prediction(self.model, image,
self.transforms,
                               self.tile_size,
                               self.tile_step,
                               self.batch_size)
        return pred
======
12 random_resized_crop.py

import math
import random
import numpy as np
from PIL import Image


def resize(img, size, interpolation=Image.BILINEAR):
    r"""Resize the input PIL Image to the given size.
    Args:
        img (PIL Image): Image to be resized.
        size (sequence or int): Desired output size. If
size is a sequence like
            (h, w), the output size will be matched to
this. If size is an int,
            the smaller edge of the image will be
matched to this number maintaing
            the aspect ratio. i.e, if height > width,
then image will be rescaled to
            :math:`\left(\text{size} \times
```

```
\frac{\text{height}}{\text{width}}, \text{size}\right)`
        interpolation (int, optional): Desired
interpolation. Default is
            ``PIL.Image.BILINEAR``
    Returns:
        PIL Image: Resized image.
    """
    if isinstance(size, int):
        w, h = img.size
        if (w <= h and w == size) or (h <= w and h ==
size):
            return img
        if w < h:
            ow = size
            oh = int(size * h / w)
            return img.resize((ow, oh), interpolation)
        else:
            oh = size
            ow = int(size * w / h)
            return img.resize((ow, oh), interpolation)
    else:
        return img.resize(size[::-1], interpolation)


def crop(img, i, j, h, w):
    """Crop the given PIL Image.
    Args:
        img (PIL Image): Image to be cropped.
        i (int): i in (i,j) i.e coordinates of the upper
left corner.
        j (int): j in (i,j) i.e coordinates of the upper
left corner.
        h (int): Height of the cropped image.
        w (int): Width of the cropped image.
    Returns:
        PIL Image: Cropped image.
    """
    return img.crop((j, i, j + w, i + h))


def resized_crop(img, i, j, h, w, size,
interpolation=Image.BILINEAR):
    """Crop the given PIL Image and resize it to desired
size.
    Notably used
in :class:`~torchvision.transforms.RandomResizedCrop`.
```

```
    Args:
        img (PIL Image): Image to be cropped.
        i (int): i in (i,j) i.e coordinates of the upper
left corner
        j (int): j in (i,j) i.e coordinates of the upper
left corner
        h (int): Height of the cropped image.
        w (int): Width of the cropped image.
        size (sequence or int): Desired output size.
Same semantics as ``resize``.
        interpolation (int, optional): Desired
interpolation. Default is
            ``PIL.Image.BILINEAR``.
    Returns:
        PIL Image: Cropped image.
    """
    img = crop(img, i, j, h, w)
    img = resize(img, size, interpolation)
    return img


class RandomResizedCrop(object):
    """Crop the given PIL Image to random size and
aspect ratio.
    A crop of random size (default: of 0.08 to 1.0) of
the original size and a random
    aspect ratio (default: of 3/4 to 4/3) of the
original aspect ratio is made. This crop
    is finally resized to given size.
    This is popularly used to train the Inception
networks.
    Args:
        size: expected output size of each edge
        scale: range of size of the origin size cropped
        ratio: range of aspect ratio of the origin
aspect ratio cropped
        interpolation: Default: PIL.Image.BILINEAR
    """

    def __init__(self, size=None, scale=(0.08, 1.0),
ratio=(3. / 4., 4. / 3.), interpolation=Image.BILINEAR):
        if isinstance(size, tuple) or size is None:
            self.size = size
        else:
            self.size = (size, size)
        if (scale[0] > scale[1]) or (ratio[0] >
```

```
ratio[1]):
            warnings.warn("range should be of kind (min,
max)")

        self.interpolation = interpolation
        self.scale = scale
        self.ratio = ratio

    @staticmethod
    def get_params(img, scale, ratio):
        """Get parameters for ``crop`` for a random
sized crop.
        Args:
            img (PIL Image): Image to be cropped.
            scale (tuple): range of size of the origin
size cropped
            ratio (tuple): range of aspect ratio of the
origin aspect ratio cropped
        Returns:
            tuple: params (i, j, h, w) to be passed to
``crop`` for a random
                sized crop.
        """
        area = img.size[0] * img.size[1]

        for attempt in range(10):
            target_area = random.uniform(*scale) * area
            log_ratio = (math.log(ratio[0]),
math.log(ratio[1]))
            aspect_ratio =
math.exp(random.uniform(*log_ratio))

            w = int(round(math.sqrt(target_area *
aspect_ratio)))
            h = int(round(math.sqrt(target_area /
aspect_ratio)))

            if w <= img.size[0] and h <= img.size[1]:
                i = random.randint(0, img.size[1] - h)
                j = random.randint(0, img.size[0] - w)
                return i, j, h, w

        # Fallback to central crop
        in_ratio = img.size[0] / img.size[1]
        if (in_ratio < min(ratio)):
            w = img.size[0]
```

```
            h = w / min(ratio)
        elif (in_ratio > max(ratio)):
            h = img.size[1]
            w = h * max(ratio)
        else:  # whole image
            w = img.size[0]
            h = img.size[1]
        i = (img.size[1] - h) // 2
        j = (img.size[0] - w) // 2
        return i, j, h, w

    def __call__(self, np_image):
        """
        Args:
            img (PIL Image): Image to be cropped and
resized.
        Returns:
            PIL Image: Randomly cropped and resized
image.
        """

        if self.size is None:
            size = np_image.shape
        else:
            size = self.size

        image = Image.fromarray(np_image)
        i, j, h, w = self.get_params(image, self.scale,
self.ratio)
        image = resized_crop(image, i, j, h, w, size,
self.interpolation)
        np_image = np.array(image)
        return np_image

    def __repr__(self):
        interpolate_str =
_pil_interpolation_to_str[self.interpolation]
        format_string = self.__class__.__name__ +
'(size={0}'.format(self.size)
        format_string += ',
scale={0}'.format(tuple(round(s, 4) for s in self.scale))
        format_string += ',
ratio={0}'.format(tuple(round(r, 4) for r in self.ratio))
        format_string += ',
interpolation={0})'.format(interpolate_str)
        return format_string
```

```
======
14 tiles.py
"""Implementation of tile-based inference allowing to
predict huge images that does not fit into GPU memory
entirely
in a sliding-window fashion and merging prediction mask
back to full-resolution.
Source: https://github.com/BloodAxe/pytorch-toolbelt/
blob/develop/pytorch_toolbelt/inference/tiles.py
"""
from typing import List

import numpy as np
import cv2
import math
import torch


def compute_pyramid_patch_weight_loss(width, height) ->
np.ndarray:
    """Compute a weight matrix that assigns bigger
weight on pixels in center and
    less weight to pixels on image boundary.
    This weight matrix then used for merging individual
tile predictions and helps dealing
    with prediction artifacts on tile boundaries.

    :param width: Tile width
    :param height: Tile height
    :return: Since-channel image [Width x Height]
    """
    xc = width * 0.5
    yc = height * 0.5
    xl = 0
    xr = width
    yb = 0
    yt = height
    Dc = np.zeros((width, height))
    De = np.zeros((width, height))

    for i in range(width):
        for j in range(height):
            Dc[i, j] = np.sqrt(np.square(i - xc + 0.5) +
np.square(j - yc + 0.5))
            De_l = np.sqrt(np.square(i - xl + 0.5) +
np.square(j - j + 0.5))
```

```
            De_r = np.sqrt(np.square(i - xr + 0.5) +
np.square(j - j + 0.5))
            De_b = np.sqrt(np.square(i - i + 0.5) +
np.square(j - yb + 0.5))
            De_t = np.sqrt(np.square(i - i + 0.5) +
np.square(j - yt + 0.5))
            De[i, j] = np.min([De_l, De_r, De_b, De_t])

    alpha = (width * height) / np.sum(np.divide(De,
np.add(Dc, De)))
    W = alpha * np.divide(De, np.add(Dc, De))
    return W, Dc, De


class ImageSlicer:
    """
    Helper class to slice image into tiles and merge
them back
    """

    def __init__(self, image_shape, tile_size,
tile_step=0, image_margin=0, weight='mean'):
        """

        :param image_shape: Shape of the source image
(H, W)
        :param tile_size: Tile size (Scalar or tuple (H,
W)
        :param tile_step: Step in pixels between tiles
(Scalar or tuple (H, W))
        :param image_margin:
        :param weight: Fusion algorithm. 'mean' -
avergaing
        """
        self.image_height = image_shape[0]
        self.image_width = image_shape[1]

        if isinstance(tile_size, (tuple, list)):
            assert len(tile_size) == 2
            self.tile_size = int(tile_size[0]),
int(tile_size[1])
        else:
            self.tile_size = int(tile_size),
int(tile_size)

        if isinstance(tile_step, (tuple, list)):
```

```
            assert len(tile_step) == 2
            self.tile_step = int(tile_step[0]),
int(tile_step[1])
        else:
            self.tile_step = int(tile_step),
int(tile_step)

        weights = {
            'mean': self._mean,
            'pyramid': self._pyramid
        }

        self.weight = weight if isinstance(weight,
np.ndarray) else weights[weight](self.tile_size)

        if self.tile_step[0] < 1 or self.tile_step[0] >
self.tile_size[0]:
            raise ValueError()
        if self.tile_step[1] < 1 or self.tile_step[1] >
self.tile_size[1]:
            raise ValueError()

        overlap = [
            self.tile_size[0] - self.tile_step[0],
            self.tile_size[1] - self.tile_step[1],
        ]

        self.margin_left = 0
        self.margin_right = 0
        self.margin_top = 0
        self.margin_bottom = 0

        if image_margin == 0:
            # In case margin is not set, we compute it
manually

            nw = max(1, math.ceil((self.image_width -
overlap[1]) / self.tile_step[1]))
            nh = max(1, math.ceil((self.image_height -
overlap[0]) / self.tile_step[0]))

            extra_w = self.tile_step[1] * nw -
(self.image_width - overlap[1])
            extra_h = self.tile_step[0] * nh -
(self.image_height - overlap[0])
```

```
            self.margin_left = extra_w // 2
            self.margin_right = extra_w -
self.margin_left
            self.margin_top = extra_h // 2
            self.margin_bottom = extra_h -
self.margin_top

        else:
            if (self.image_width - overlap[1] + 2 *
image_margin) % self.tile_step[1] != 0:
                raise ValueError()

            if (self.image_height - overlap[0] + 2 *
image_margin) % self.tile_step[0] != 0:
                raise ValueError()

            self.margin_left = image_margin
            self.margin_right = image_margin
            self.margin_top = image_margin
            self.margin_bottom = image_margin

        crops = []
        bbox_crops = []

        for y in range(0, self.image_height +
self.margin_top + self.margin_bottom - self.tile_size[0]
+ 1, self.tile_step[0]):
            for x in range(0, self.image_width +
self.margin_left + self.margin_right - self.tile_size[1]
+ 1, self.tile_step[1]):
                crops.append((x, y, self.tile_size[1],
self.tile_size[0]))
                bbox_crops.append((x - self.margin_left,
y - self.margin_top, self.tile_size[1],
self.tile_size[0]))

        self.crops = np.array(crops)
        self.bbox_crops = np.array(bbox_crops)

    def split(self, image,
border_type=cv2.BORDER_CONSTANT, value=0):
        assert image.shape[0] == self.image_height
        assert image.shape[1] == self.image_width

        orig_shape_len = len(image.shape)
        image = cv2.copyMakeBorder(image,
```

```
self.margin_top, self.margin_bottom, self.margin_left,
self.margin_right, borderType=border_type, value=value)

        # This check recovers possible lack of last
dummy dimension for single-channel images
        if len(image.shape) != orig_shape_len:
            image = np.expand_dims(image, axis=-1)

        tiles = []
        for x, y, tile_width, tile_height in self.crops:
            tile = image[y:y + tile_height, x:x +
tile_width].copy()
            assert tile.shape[0] == self.tile_size[0]
            assert tile.shape[1] == self.tile_size[1]

            tiles.append(tile)

        return tiles

    def cut_patch(self, image: np.ndarray, slice_index,
border_type=cv2.BORDER_CONSTANT, value=0):
        assert image.shape[0] == self.image_height
        assert image.shape[1] == self.image_width

        orig_shape_len = len(image.shape)
        image = cv2.copyMakeBorder(image,
self.margin_top, self.margin_bottom, self.margin_left,
self.margin_right, borderType=border_type, value=value)

        # This check recovers possible lack of last
dummy dimension for single-channel images
        if len(image.shape) != orig_shape_len:
            image = np.expand_dims(image, axis=-1)

        x, y, tile_width, tile_height =
self.crops[slice_index]

        tile = image[y:y + tile_height, x:x +
tile_width].copy()
        assert tile.shape[0] == self.tile_size[0]
        assert tile.shape[1] == self.tile_size[1]
        return tile

    @property
    def target_shape(self):
        target_shape = self.image_height +
```

```
self.margin_bottom + self.margin_top, self.image_width +
self.margin_right + self.margin_left
        return target_shape

    def merge(self, tiles: List[np.ndarray],
dtype=np.float32):
        if len(tiles) != len(self.crops):
            raise ValueError

        channels = 1 if len(tiles[0].shape) == 2 else
tiles[0].shape[2]
        target_shape = self.image_height +
self.margin_bottom + self.margin_top, self.image_width +
self.margin_right + self.margin_left, channels

        image = np.zeros(target_shape, dtype=np.float64)
        norm_mask = np.zeros(target_shape,
dtype=np.float64)

        w = np.dstack([self.weight] * channels)

        for tile, (x, y, tile_width, tile_height) in
zip(tiles, self.crops):
            # print(x, y, tile_width, tile_height,
image.shape)
            image[y:y + tile_height, x:x + tile_width]
+= tile * w
            norm_mask[y:y + tile_height, x:x +
tile_width] += w

        # print(norm_mask.min(), norm_mask.max())
        norm_mask = np.clip(norm_mask,
a_min=np.finfo(norm_mask.dtype).eps, a_max=None)
        normalized = np.divide(image,
norm_mask).astype(dtype)
        crop = self.crop_to_orignal_size(normalized)
        return crop

    def crop_to_orignal_size(self, image):
        assert image.shape[0] == self.target_shape[0]
        assert image.shape[1] == self.target_shape[1]
        crop = image[self.margin_top:self.image_height +
self.margin_top, self.margin_left:self.image_width +
self.margin_left]
        assert crop.shape[0] == self.image_height
        assert crop.shape[1] == self.image_width
```

```
        return crop

    def _mean(self, tile_size):
        return np.ones((tile_size[0], tile_size[1]),
dtype=np.float32)

    def _pyramid(self, tile_size):
        w, _, _ =
compute_pyramid_patch_weight_loss(tile_size[0],
tile_size[1])
        return w


class CudaTileMerger:
    """
    Helper class to merge final image on GPU. This
generally faster than moving individual tiles to CPU.
    """

    def __init__(self, image_shape, channels, weight):
        """

        :param image_shape: Shape of the source image
        :param image_margin:
        :param weight: Weighting matrix
        """
        self.image_height = image_shape[0]
        self.image_width = image_shape[1]

        self.weight =
torch.from_numpy(np.expand_dims(weight,
axis=0)).float().cuda()
        self.channels = channels
        self.image = torch.zeros((channels,
self.image_height, self.image_width)).cuda()
        self.norm_mask = torch.zeros((1,
self.image_height, self.image_width)).cuda()

    def integrate_batch(self, batch: torch.Tensor,
crop_coords):
        """
        Accumulates batch of tile predictions
        :param batch: Predicted tiles
        :param crop_coords: Corresponding tile crops
w.r.t to original image
        """
```

```
        if len(batch) != len(crop_coords):
            raise ValueError("Number of images in batch 
does not correspond to number of coordinates")

        for tile, (x, y, tile_width, tile_height) in 
zip(batch, crop_coords):
            self.image[:, y:y + tile_height, x:x + 
tile_width] += tile * self.weight
            self.norm_mask[:, y:y + tile_height, x:x + 
tile_width] += self.weight

    def merge(self) -> torch.Tensor:
        return self.image / self.norm_mask
======
15 transforms.py
import cv2
import torch
import random
import librosa
import numpy as np

from src.random_resized_crop import RandomResizedCrop

cv2.setNumThreads(0)


def image_crop(image, bbox):
    return image[bbox[1]:bbox[3], bbox[0]:bbox[2]]


def gauss_noise(image, sigma_sq):
    h, w = image.shape
    gauss = np.random.normal(0, sigma_sq, (h, w))
    gauss = gauss.reshape(h, w)
    image = image + gauss
    return image


# Source: https://www.kaggle.com/davids1992/specaugment-
quick-implementation
def spec_augment(spec: np.ndarray,
                 num_mask=2,
                 freq_masking=0.15,
                 time_masking=0.20,
                 value=0):
    spec = spec.copy()
```

```
    num_mask = random.randint(1, num_mask)
    for i in range(num_mask):
        all_freqs_num, all_frames_num  = spec.shape
        freq_percentage = random.uniform(0.0,
freq_masking)

        num_freqs_to_mask = int(freq_percentage *
all_freqs_num)
        f0 = np.random.uniform(low=0.0,
high=all_freqs_num - num_freqs_to_mask)
        f0 = int(f0)
        spec[f0:f0 + num_freqs_to_mask, :] = value

        time_percentage = random.uniform(0.0,
time_masking)

        num_frames_to_mask = int(time_percentage *
all_frames_num)
        t0 = np.random.uniform(low=0.0,
high=all_frames_num - num_frames_to_mask)
        t0 = int(t0)
        spec[:, t0:t0 + num_frames_to_mask] = value
    return spec


class SpecAugment:
    def __init__(self,
                 num_mask=2,
                 freq_masking=0.15,
                 time_masking=0.20):
        self.num_mask = num_mask
        self.freq_masking = freq_masking
        self.time_masking = time_masking

    def __call__(self, image):
        return spec_augment(image,
                            self.num_mask,
                            self.freq_masking,
                            self.time_masking,
                            image.min())


class Compose:
    def __init__(self, transforms):
        self.transforms = transforms
```

```
    def __call__(self, image, trg=None):
        if trg is None:
            for t in self.transforms:
                image = t(image)
            return image
        else:
            for t in self.transforms:
                image, trg = t(image, trg)
            return image, trg


class UseWithProb:
    def __init__(self, transform, prob=.5):
        self.transform = transform
        self.prob = prob

    def __call__(self, image, trg=None):
        if trg is None:
            if random.random() < self.prob:
                image = self.transform(image)
            return image
        else:
            if random.random() < self.prob:
                image, trg = self.transform(image, trg)
            return image, trg


class OneOf:
    def __init__(self, transforms, p=None):
        self.transforms = transforms
        self.p = p

    def __call__(self, image, trg=None):
        transform = np.random.choice(self.transforms,
p=self.p)
        if trg is None:
            image = transform(image)
            return image
        else:
            image, trg = transform(image, trg)
            return image, trg


class Flip:
    def __init__(self, flip_code):
        assert flip_code == 0 or flip_code == 1
```

```
        self.flip_code = flip_code

    def __call__(self, image):
        image = cv2.flip(image, self.flip_code)
        return image


class HorizontalFlip(Flip):
    def __init__(self):
        super().__init__(1)


class VerticalFlip(Flip):
    def __init__(self):
        super().__init__(0)


class GaussNoise:
    def __init__(self, sigma_sq):
        self.sigma_sq = sigma_sq

    def __call__(self, image):
        if self.sigma_sq > 0.0:
            image = gauss_noise(image,
                                np.random.uniform(0,
self.sigma_sq))
        return image


class RandomGaussianBlur:
    '''Apply Gaussian blur with random kernel size
    Args:
        max_ksize (int): maximal size of a kernel to
apply, should be odd
        sigma_x (int): Standard deviation
    '''
    def __init__(self, max_ksize=5, sigma_x=20):
        assert max_ksize % 2 == 1, "max_ksize should be
odd"
        self.max_ksize = max_ksize // 2 + 1
        self.sigma_x = sigma_x

    def __call__(self, image):
        kernel_size = tuple(2 * np.random.randint(0,
self.max_ksize, 2) + 1)
        blured_image = cv2.GaussianBlur(image,
```

```
kernel_size, self.sigma_x)
        return blured_image


class ImageToTensor:
    def __call__(self, image):
        delta = librosa.feature.delta(image)
        accelerate = librosa.feature.delta(image,
order=2)
        image = np.stack([image, delta, accelerate],
axis=0)
        image = image.astype(np.float32) / 100
        image = torch.from_numpy(image)
        return image


class RandomCrop:
    def __init__(self, size):
        self.size = size

    def __call__(self, signal):
        start = random.randint(0, signal.shape[1] -
self.size)
        return signal[:, start: start + self.size]


class CenterCrop:
    def __init__(self, size):
        self.size = size

    def __call__(self, signal):

        if signal.shape[1] > self.size:
            start = (signal.shape[1] - self.size) // 2
            return signal[:, start: start + self.size]
        else:
            return signal


class PadToSize:
    def __init__(self, size, mode='constant'):
        assert mode in ['constant', 'wrap']
        self.size = size
        self.mode = mode

    def __call__(self, signal):
```

```
        if signal.shape[1] < self.size:
            padding = self.size - signal.shape[1]
            offset = padding // 2
            pad_width = ((0, 0), (offset, padding -
offset))
            if self.mode == 'constant':
                signal = np.pad(signal, pad_width,
                                'constant',
constant_values=signal.min())
            else:
                signal = np.pad(signal, pad_width,
'wrap')
        return signal


def get_transforms(train, size,
                   wrap_pad_prob=0.5,
                   resize_scale=(0.8, 1.0),
                   resize_ratio=(1.7, 2.3),
                   resize_prob=0.33,
                   spec_num_mask=2,
                   spec_freq_masking=0.15,
                   spec_time_masking=0.20,
                   spec_prob=0.5):
    if train:
        transforms = Compose([
            OneOf([
                PadToSize(size, mode='wrap'),
                PadToSize(size, mode='constant'),
            ], p=[wrap_pad_prob, 1 - wrap_pad_prob]),
            RandomCrop(size),
            UseWithProb(
                RandomResizedCrop(scale=resize_scale,
ratio=resize_ratio),
                prob=resize_prob
            ),

UseWithProb(SpecAugment(num_mask=spec_num_mask,

freq_masking=spec_freq_masking,

time_masking=spec_time_masking), spec_prob),
            ImageToTensor()
        ])
    else:
        transforms = Compose([
```

```
            PadToSize(size),
            CenterCrop(size),
            ImageToTensor()
        ])
    return transforms
======
16 utils.py
import re
import json
import pickle
import numpy as np
from pathlib import Path
from scipy.stats.mstats import gmean

from src.datasets import get_noisy_data_generator,
get_folds_data, get_augment_folds_data_generator
from src import config


def gmean_preds_blend(probs_df_lst):
    blend_df = probs_df_lst[0]
    blend_values =
np.stack([df.loc[blend_df.index.values].values
                             for df in probs_df_lst],
axis=0)
    blend_values = gmean(blend_values, axis=0)

    blend_df.values[:] = blend_values
    return blend_df


def get_best_model_path(dir_path: Path,
return_score=False):
    model_scores = []
    for model_path in dir_path.glob('*.pth'):
        score = re.search(r'-(\d+(?:\.\d+)?).pth',
str(model_path))
        if score is not None:
            score = float(score.group(0)[1:-4])
            model_scores.append((model_path, score))
    model_score = sorted(model_scores, key=lambda x:
x[1])
    best_model_path = model_score[-1][0]
    if return_score:
        best_score = model_score[-1][1]
        return best_model_path, best_score
```

```
    else:
        return best_model_path


def pickle_save(obj, filename):
    print(f"Pickle save to: {filename}")
    with open(filename, 'wb') as f:
        pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)


def pickle_load(filename):
    print(f"Pickle load from: {filename}")
    with open(filename, 'rb') as f:
        return pickle.load(f)


def load_folds_data(use_corrections=True):
    if use_corrections:
        with open(config.corrections_json_path) as file:
            corrections = json.load(file)
        print("Corrections:", corrections)
        pkl_name =
f'{config.audio.get_hash(corrections=corrections)}.pkl'
    else:
        corrections = None
        pkl_name = f'{config.audio.get_hash()}.pkl'

    folds_data_pkl_path = config.folds_data_pkl_dir /
pkl_name

    if folds_data_pkl_path.exists():
        folds_data = pickle_load(folds_data_pkl_path)
    else:
        folds_data = get_folds_data(corrections)
        if not config.folds_data_pkl_dir.exists():

config.folds_data_pkl_dir.mkdir(parents=True,
exist_ok=True)
        pickle_save(folds_data, folds_data_pkl_path)
    return folds_data


def load_noisy_data():
    with open(config.noisy_corrections_json_path) as
file:
        corrections = json.load(file)
```

```
    pkl_name_glob =
f'{config.audio.get_hash(corrections=corrections)}_*.pkl'
    pkl_paths =
sorted(config.noisy_data_pkl_dir.glob(pkl_name_glob))

    images_lst, targets_lst = [], []

    if pkl_paths:
        for pkl_path in pkl_paths:
            data_batch = pickle_load(pkl_path)
            images_lst += data_batch[0]
            targets_lst += data_batch[1]
    else:
        if not config.noisy_data_pkl_dir.exists():

config.noisy_data_pkl_dir.mkdir(parents=True,
exist_ok=True)

        for i, data_batch in
enumerate(get_noisy_data_generator()):
            pkl_name =
f'{config.audio.get_hash(corrections=corrections)}_{i:
02}.pkl'
            noisy_data_pkl_path =
config.noisy_data_pkl_dir / pkl_name
            pickle_save(data_batch, noisy_data_pkl_path)

            images_lst += data_batch[0]
            targets_lst += data_batch[1]

    return images_lst, targets_lst


def load_augment_folds_data(time_stretch_lst,
pitch_shift_lst):
    config_hash =
config.audio.get_hash(time_stretch_lst=time_stretch_lst,

pitch_shift_lst=pitch_shift_lst)
    pkl_name_glob = f'{config_hash}_*.pkl'
    pkl_paths =
sorted(config.augment_folds_data_pkl_dir.glob(pkl_name_glob))

    images_lst, targets_lst, folds_lst = [], [], []
```

```
    if pkl_paths:
        for pkl_path in pkl_paths:
            data_batch = pickle_load(pkl_path)
            images_lst += data_batch[0]
            targets_lst += data_batch[1]
            folds_lst += data_batch[2]
    else:
        if not
config.augment_folds_data_pkl_dir.exists():

config.augment_folds_data_pkl_dir.mkdir(parents=True,
exist_ok=True)

        generator =
get_augment_folds_data_generator(time_stretch_lst,
pitch_shift_lst)
        for i, data_batch in enumerate(generator):
            pkl_name = f'{config_hash}_{i:02}.pkl'
            augment_data_pkl_path =
config.augment_folds_data_pkl_dir / pkl_name
            pickle_save(data_batch,
augment_data_pkl_path)

            images_lst += data_batch[0]
            targets_lst += data_batch[1]
            folds_lst += data_batch[2]

    return images_lst, targets_lst, folds_lst
======
```

Kaggle Freesound Audio Tagging 2019 2nd place code

Usage

- Download the datasets and place them in the input folder.
- Unzip the train_curated.zip and train_noisy.zip, then put all the audio clips into audio_train.
- sh run.sh

requirements

tensorflow_gpu==1.11.0 numpy==1.14.2 tqdm==4.22.0 librosa==0.6.3 scipy==1.0.0 iterative_stratification==0.1.6 Keras==2.1.5 pandas==0.24.2 scikit_learn==0.21.2

Hardware

- 64GB of RAM
- 1 tesla P100

Solution

single model CV: 0.89763

ensemble CV: 0.9108

feature engineering

- log mel (441,64) (time,mels)
- global feature (128,12) (Split the clip evenly, and create 12 features for each frame. local cv +0.005)
- length

```
def get_global_feat(x,num_steps):
    stride = len(x)/num_steps
    ts = []
    for s in range(num_steps):
        i = s * stride
        wl = max(0,int(i - stride/2))
        wr = int(i + 1.5*stride)
        local_x = x[wl:wr]
        percent_feat = np.percentile(local_x, [0, 1, 25, 30, 50, 60, 75,
        range_feat = local_x.max()-local_x.min()
        ts.append([np.mean(local_x),np.std(local_x),range_feat]+percent_f
    ts = np.array(ts)
    assert ts.shape == (128,12),(len(x),ts.shape)
    return ts
```

prepocess

- audio clips are first trimmed of leading and trailing silence
- random select a 5s clip from audio clip

model

For details, please refer to code/models.py *Melspectrogram Layer(code from kapre,We use it to search the hyperparameter of log mel end2end)* Our main model is a 9-layer CNN. In this competition, we consider that the two axes of the log mel feature have different physical meanings, so the max pooling and average pooling in the model are replaced by one axis using max pooling and the other axis using average pooling. (Our local cv gain a lot from it, but the exact number is forgotten). *global pooling: pixelshuffle + max pooling in time axes + ave pooling in mel axes.* se block (several of our models use se block) *highway + 1*1 conv (several of our models use se block) * label smoothing

```
# log mel layer
x_mel = Melspectrogram(n_dft=1024, n_hop=cfg.stride, input_shape=(1, K.in
                           # n_hop -> stride   n_dft kernel_size
                           padding='same', sr=44100, n_mels=64,
                           power_melgram=2, return_decibel_melgram=True,
                           trainable_fb=False, trainable_kernel=False,
                           image_data_format='channels_last', trainable=F


# pooling mode
x = AveragePooling2D(pool_size=(pool_size1,1), padding='same', strides=(s
x = MaxPool2D(pool_size=(1,pool_size2), padding='same', strides=(1,stride


# model head
def pixelShuffle(x):
    _,h,w,c = K.int_shape(x)
    bs = K.shape(x)[0]
    assert w%2==0
    x = K.reshape(x,(bs,h,w//2,c*2))

    # assert h % 2 == 0
    # x = K.permute_dimensions(x,(0,2,1,3))
    # x = K.reshape(x,(bs,w//2,h//2,c*4))
    # x = K.permute_dimensions(x,(0,2,1,3))
    return x
x = Lambda(pixelShuffle)(x)
x = Lambda(lambda x: K.max(x, axis=1))(x)
x = Lambda(lambda x: K.mean(x, axis=1))(x)
```

data augmentation

- mixup (local cv +0.002, lb +0.008)
- random select 5s clip + random padding
- 3TTA

pretrain

- train a model only on train_noisy as pretrained model

ensemble

For details, please refer to code/ensemble.py * We use nn for stacking, which uses localconnect1D to learn the ensemble weights of each class, then use fully connect to learn about label correlation, using some initialization and weight constraint tricks.

```
def stacker(cfg,n):
    def kinit(shape, name=None):
        value = np.zeros(shape)
        value[:, -1] = 1
        return K.variable(value, name=name)


    x_in = Input((80,n))
    x = x_in
    # x = Lambda(lambda x: 1.5*x)(x)
    x = LocallyConnected1D(1,1,kernel_initializer=kinit,kernel_constraint
    x = Flatten()(x)
    x = Dense(80, use_bias=False, kernel_initializer=Identity(1))(x)
    x = Lambda(lambda x: (x - 1.6))(x)
    x = Activation('tanh')(x)
    x = Lambda(lambda x:(x+1)*0.5)(x)

    model = Model(inputs=x_in, outputs=x)
    model.compile(
        loss='binary_crossentropy',
        optimizer=Nadam(lr=cfg.lr),
    )
    return model
```

1 run.sh
2 utils.py
3 pretrain.py
4 train.py
5 predict.py
6 ensemble.py
----
1 run.sh

```
#!/usr/bin/env bash


python utils.py
python pretrain.py
python train.py
python predict.py
python ensemble.py
```

----
2 utils.py

```
import numpy as np
from tqdm import tqdm
import pandas as pd
from keras.utils.data_utils import Sequence
import librosa
from keras.preprocessing.sequence import pad_sequences
from config import *
import multiprocessing as mp
import pickle
from models import cnn_model
from sklearn.preprocessing import StandardScaler
from collections import defaultdict, Counter
import scipy

class FreeSound(Sequence):
    def __init__(self,X,Gfeat,Y,cfg,mode,epoch):

        self.X, self.Gfeat, self.Y, self.cfg =
X,Gfeat,Y,cfg
        self.bs = cfg.bs
        self.mode = mode
        self.ids = list(range(len(self.X)))
        self.epoch = epoch

        self.aug = None
```

```
        if mode == 'train':
            self.get_offset = np.random.randint
            np.random.shuffle(self.ids)

        elif mode == 'pred1':
            self.get_offset = lambda x: 0
        elif mode == 'pred2':
            self.get_offset = lambda x: int(x/2)
        elif mode == 'pred3':
            self.get_offset = lambda x: x
        else:
            raise RuntimeError("error")


    def __len__(self):
        return (len(self.X)+self.bs-1) // self.bs


    def __getitem__(self,idx):

        batch_idx = self.ids[idx*self.bs:(idx+1)*self.bs]
        batch_x = {
            'audio':[],
            'other':[],
            'global_feat':self.Gfeat[batch_idx],
        }
        for i in batch_idx:
            audio_sample = self.X[i]

            feature = [audio_sample.shape[0] / 441000]
            batch_x['other'].append(feature)

            max_offset = audio_sample.shape[0] -
self.cfg.maxlen
            data = self.get_sample(audio_sample,
max_offset)

            batch_x['audio'].append(data)

        batch_y = np.array(self.Y[batch_idx])
        batch_x = {k: np.array(v) for k, v in
batch_x.items()}

        if self.mode == 'train':
            batch_y = self.cfg.lm * (1-batch_y) + (1 -
self.cfg.lm) * batch_y
```

```
        if self.mode == 'train' and np.random.rand() <
self.cfg.mixup_prob and self.epoch <
self.cfg.milestones[0]:
            batch_idx =
np.random.permutation(list(range(len(batch_idx))))
            rate = self.cfg.x1_rate

            batch_x['audio'] = rate * batch_x['audio'] +
(1-rate) * batch_x['audio'][batch_idx]
            batch_y = rate * batch_y + (1-rate) *
batch_y[batch_idx]


        batch_x['y'] = batch_y
        return batch_x, None

    def augment(self,data):
        # if self.mode == 'train' and self.epoch <
self.cfg.milestones[0] and np.random.rand() < 0.5:
        #     mask_len = int(data.shape[0] * 0.02)
        #     s = np.random.randint(0,data.shape[0]-
mask_len)
        #     data[s:s+mask_len] = 0
        return data

    def get_sample(self,data,max_offset):
        if max_offset > 0:
            offset = self.get_offset(max_offset)
            data = data[offset:(self.cfg.maxlen +
offset)]
            if self.mode == 'train':
                data = self.augment(data)

        elif max_offset < 0:
            max_offset = -max_offset
            offset = self.get_offset(max_offset)
            if self.mode == 'train':
                data = self.augment(data)
            if len(data.shape) == 1:
                data = np.pad(data, ((offset, max_offset
- offset)), "constant")
            else:
                data = np.pad(data, ((offset, max_offset
- offset),(0,0),(0,0)), "constant")
        return data
```

```
    def on_epoch_end(self):
        if self.mode == 'train':
            np.random.shuffle(self.ids)


def get_global_feat(x,num_steps):
    stride = len(x)/num_steps
    ts = []
    for s in range(num_steps):
        i = s * stride
        wl = max(0,int(i - stride/2))
        wr = int(i + 1.5*stride)
        local_x = x[wl:wr]
        percent_feat = np.percentile(local_x, [0, 1, 25, 
30, 50, 60, 75, 99, 100]).tolist()
        range_feat = local_x.max()-local_x.min()

ts.append([np.mean(local_x),np.std(local_x),range_feat]
+percent_feat)
    ts = np.array(ts)
    assert ts.shape == (128,12),(len(x),ts.shape)
    return ts


def worker_cgf(file_path):
    result = []
    for path in tqdm(file_path):
        data, _ = librosa.load(path, 44100)
        result.append(get_global_feat(data, 
num_steps=128))
    return result


def create_global_feat():

    df = pd.concat([pd.read_csv(f'../input/
train_curated.csv'),pd.read_csv('../input/
train_noisy.csv',usecols=['fname','labels'])])
    df = df.reset_index(drop=True)
    file_path = train_dir + df['fname']

    workers = mp.cpu_count() // 2
    pool = mp.Pool(workers)
    results = []
    ave_task = (len(file_path) + workers - 1) // workers
```

```
    for i in range(workers):
        res = pool.apply_async(worker_cgf,
                               args=(file_path[i *
ave_task:(i + 1) * ave_task],))
        results.append(res)
    pool.close()
    pool.join()

    results = np.concatenate([res.get() for res in
results],axis=0)
    print(results.shape)
    np.save('../input/gfeat', np.array(results))

    df = pd.read_csv(f'../input/sample_pred.csv')

    file_path = train_dir + df['fname']

    workers = mp.cpu_count() // 2
    pool = mp.Pool(workers)
    results = []
    ave_task = (len(file_path) + workers - 1) // workers
    for i in range(workers):
        res = pool.apply_async(worker_cgf,
                               args=(file_path[i *
ave_task:(i + 1) * ave_task],))
        results.append(res)
    pool.close()
    pool.join()

    results = np.concatenate([res.get() for res in
results], axis=0)
    print(results.shape)
    np.save('../input/te_gfeat', np.array(results))

def split_and_label(rows_labels):
    row_labels_list = []
    for row in rows_labels:
        row_labels = row.split(',')
        labels_array = np.zeros((n_classes))
        for label in row_labels:
            index = label2i[label]
            labels_array[index] = 1
        row_labels_list.append(labels_array)
    return np.array(row_labels_list)
```

```
if __name__ == '__main__':

    create_global_feat()
----------

3 pretrain.py


from tqdm import tqdm
from sklearn.metrics import
label_ranking_average_precision_score
from utils import *
from config import *


def main(cfg,get_model):

    if True: # load data
        df = pd.read_csv(f'../input/train_noisy.csv')
        y = split_and_label(df['labels'].values)
        x = train_dir + df['fname'].values
        x = [librosa.load(path, 44100)[0] for path in
tqdm(x)]
        x = [librosa.effects.trim(data)[0] for data in
tqdm(x)]

        gfeat = np.load('../input/gfeat.npy')[-len(x):]


        df = pd.read_csv(f'../input/train_curated.csv')
        val_y = split_and_label(df['labels'].values)
        val_x = train_dir + df['fname'].values
        val_x = [librosa.load(path, 44100)[0] for path
in tqdm(val_x)]
        val_x = [librosa.effects.trim(data)[0] for data
in tqdm(val_x)]
        val_gfeat = np.load('../input/gfeat.npy')
[:len(val_x)]

    print(cfg)

    if True: # init
        K.clear_session()
        model = get_model(cfg)
        best_score = -np.inf
```

```
    for epoch in range(35):

        if epoch in cfg.milestones:
            K.set_value(model.optimizer.lr,
K.get_value(model.optimizer.lr) * cfg.gamma)

        tr_loader = FreeSound(x, gfeat, y, cfg, 'train',
epoch)
        val_loaders = [FreeSound(val_x, val_gfeat,
val_y, cfg, f'pred{i+1}', epoch) for i in range(3)]

        model.fit_generator(
            tr_loader,
            steps_per_epoch=len(tr_loader),
            verbose=0,
            workers=6
        )
        val_pred = [model.predict_generator(vl,
workers=4) for vl in val_loaders]
        ave_val_pred = np.average(val_pred, axis=0)
        score =
label_ranking_average_precision_score(val_y,
ave_val_pred)

        if epoch >= 28 and score > best_score:
            best_score = score
            model.save_weights(f"../model/{cfg.name}
pretrainedbest.h5")

        if epoch >= 28:
            model.save_weights(f"../model/{cfg.name}
pretrained{epoch}.h5")
            print(f'{epoch} score {score},  best
{best_score}...')





if __name__ == '__main__':
    from models import *

    cfg = Config(
        duration=5,
        name='v1mix',
```

```
        lr=0.0005,
        batch_size=32,
        rnn_unit=128,
        momentum=0.85,
        mixup_prob=0.7,
        lm=0.01,
        pool_mode=('max', 'avemax1'),
        x1_rate=0.7,
        milestones=(8,12,16),
        get_backbone=get_conv_backbone
    )
    main(cfg, cnn_model)

    cfg = Config(
        duration=5,
        name='model_MSC_se_r4_1.0_10fold',
        lr=0.0005,
        batch_size=32,
        rnn_unit=128,
        momentum=0.85,
        mixup_prob=0.7,
        lm=0.01,
        pool_mode=('max', 'avemax1'),
        x1_rate=0.7,
        milestones=(8, 12, 16),
        get_backbone=model_se_MSC,
        w_ratio=1,
    )
    main(cfg, cnn_model)

    cfg = Config(
        duration=5,
        name='model_MSC_se_r4_2.0_10fold',
        lr=0.0005,
        batch_size=32,
        rnn_unit=128,
        momentum=0.85,
        mixup_prob=0.7,
        lm=0.01,
        pool_mode=('max', 'avemax1'),
        x1_rate=0.7,
        milestones=(8, 12, 16),
        get_backbone=model_se_MSC,
        w_ratio=2.0,
    )
    main(cfg, cnn_model)
```

```
    cfg = Config(
        duration=5,
        name='model_se_r4_1.5_10fold',
        lr=0.0005,
        batch_size=32,
        rnn_unit=128,
        momentum=0.85,
        mixup_prob=0.7,
        lm=0.01,
        pool_mode=('max', 'avemax1'),
        x1_rate=0.7,
        milestones=(8, 12, 16),
        get_backbone=model_se_MSC,
        w_ratio=1.5,
    )
    main(cfg, cnn_model)

    cfg = Config(
        duration=5,
        name='se',
        lr=0.0005,
        batch_size=32,
        rnn_unit=128,
        momentum=0.85,
        mixup_prob=0.7,
        lm=0.01,
        pool_mode=('max', 'avemax1'),
        x1_rate=0.7,
        milestones=(8, 12, 16),
        get_backbone=get_se_backbone
    )
    main(cfg, cnn_model)
```

```
----------
4. train.py
import tensorflow as tf
import keras.backend.tensorflow_backend as KTF
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
sess = tf.Session(config=config)
```

```
KTF.set_session(sess)
from sklearn.metrics import
label_ranking_average_precision_score
from sklearn.model_selection import StratifiedKFold
from utils import *
from config import *
from iterstrat.ml_stratifiers import
MultilabelStratifiedKFold
from models import *
import pickle
import multiprocessing as mlp
# seed = 3921
# random.seed(seed)
# os.environ['PYTHONHASHSEED'] = f'{seed}'
# np.random.seed(seed)

def worker_prepocess(file_path):

    result = []
    for path in tqdm(file_path):
        data = librosa.load(path, 44100)[0]
        data = librosa.effects.trim(data)[0]

        result.append(data)
    return result

def prepocess_para(file_path):

    workers = mp.cpu_count() // 2
    pool = mp.Pool(workers)
    results = []
    ave_task = (len(file_path) + workers - 1) // workers
    for i in range(workers):
        res = pool.apply_async(worker_prepocess,
                               args=(file_path[i *
ave_task:(i + 1) * ave_task],))
        results.append(res)
    pool.close()
    pool.join()

    dataset = []
    for res in results:
        dataset += res.get()
    return dataset
```

```
def main(cfg,get_model):

    if True: # load data
        df = pd.read_csv(f'../input/train_curated.csv')
        y = split_and_label(df['labels'].values)
        x = train_dir + df['fname'].values
        # # x = prepocess_para(x)

        x = [librosa.load(path, 44100)[0] for path in
tqdm(x)]
        x = [librosa.effects.trim(data)[0] for data in
tqdm(x)]
        # with open('../input/tr_logmel.pkl', 'rb') as f:
        #     x = pickle.load(f)
        gfeat = np.load('../input/gfeat.npy')[:len(y)]



    print(cfg)
    mskfold = MultilabelStratifiedKFold(cfg.n_folds,
shuffle=False, random_state=66666)
    folds = list(mskfold.split(x,y))[::-1]
    # te_folds = list(mskfold.split(te_x,
(te_y>0.5).astype(int)))

    oofp = np.zeros_like(y)
    for fold, (tr_idx, val_idx) in enumerate(folds):
        if fold not in cfg.folds:
            continue
        print("Beginning fold {}".format(fold + 1))

        if True: # init
            K.clear_session()
            model = get_model(cfg)
            best_epoch = 0
            best_score = -1

        for epoch in range(40):
            if epoch >=35 and epoch - best_epoch > 10:
                break

            if epoch in cfg.milestones:

K.set_value(model.optimizer.lr,K.get_value(model.optimizer.lr)
* cfg.gamma)
```

```
            tr_x, tr_y, tr_gfeat = [x[i] for i in
tr_idx], y[tr_idx], gfeat[tr_idx]
            val_x, val_y, val_gfeat = [x[i] for i in
val_idx], y[val_idx], gfeat[val_idx]

            tr_loader = FreeSound(tr_x, tr_gfeat, tr_y,
cfg, 'train',epoch)
            val_loaders = [FreeSound(val_x, val_gfeat,
val_y, cfg, f'pred{i+1}',epoch) for i in range(3)]

            model.fit_generator(
                tr_loader,
                steps_per_epoch=len(tr_loader),
                verbose=0,
                workers=6
            )
            val_pred =
[model.predict_generator(vl,workers=4) for vl in
val_loaders]
            ave_val_pred = np.average(val_pred,axis=0)
            score =
label_ranking_average_precision_score(val_y,ave_val_pred)

            if score > best_score:
                best_score = score
                best_epoch = epoch
                oofp[val_idx] = ave_val_pred
                model.save_weights(f"../model/{cfg.name}
{fold}.h5")
            print(f'{epoch} score {score} ,  best
{best_score}...')

    print('lrap:
',label_ranking_average_precision_score(y,oofp))
        # best_threshold, best_score, raw_score =
threshold_search(Y, oofp)
        # print(f'th {best_threshold}, val raw_score
{raw_score}, val best score:{best_score}')

if __name__ == '__main__':
    from models import *

    cfg = Config(
        duration=5,
        name='v1mix',
```

```
        lr=0.0005,
        batch_size=32,
        rnn_unit=128,
        momentum=0.85,
        mixup_prob=0.6,
        lm=0.01,
        pool_mode=('max', 'avemax1'),
        x1_rate=0.7,
        n_folds=10,
        get_backbone=get_conv_backbone,
        pretrained='../model/v1mixpretrainedbest.h5',
    )
    main(cfg, cnn_model)

    cfg = Config(
        duration=5,
        name='max3exam',
        lr=0.0005,
        batch_size=32,
        rnn_unit=128,
        momentum=0.85,
        mixup_prob=0.6,
        lm=0.01,
        pool_mode=('max', 'avemax3'),
        x1_rate=0.7,
        n_folds=10,
        get_backbone=get_conv_backbone,
        pretrained='../model/v1mixpretrainedbest.h5',
    )
    main(cfg, cnn_model)

    cfg = Config(
        duration=5,
        name='model_MSC_se_r4_1.0_10fold',
        lr=0.0005,
        batch_size=32,
        rnn_unit=128,
        momentum=0.85,
        mixup_prob=0.6,
        lm=0.01,
        pool_mode=('max', 'avemax1'),
        x1_rate=0.7,
        n_folds=10,
        get_backbone=model_se_MSC,
        w_ratio=1,
        pretrained='../model/
```

```
model_MSC_se_r4_1.0_10foldpretrainedbest.h5',
    )
    main(cfg, cnn_model)

    cfg = Config(
        duration=5,
        name='model_MSC_se_r4_2.0_10fold',
        lr=0.0005,
        batch_size=32,
        rnn_unit=128,
        momentum=0.85,
        mixup_prob=0.6,
        lm=0.01,
        pool_mode=('max', 'avemax1'),
        x1_rate=0.7,
        n_folds=10,
        get_backbone=model_se_MSC,
        w_ratio=2.0,
        pretrained='../model/
model_MSC_se_r4_2.0_10foldpretrainedbest.h5',
    )
    main(cfg, cnn_model)

    cfg = Config(
        duration=5,
        name='model_se_r4_1.5_10fold',
        lr=0.0005,
        batch_size=32,
        rnn_unit=128,
        momentum=0.85,
        mixup_prob=0.6,
        lm=0.01,
        pool_mode=('max', 'avemax1'),
        x1_rate=0.7,
        n_folds=10,
        get_backbone=model_se_MSC,
        w_ratio=1.5,
        pretrained='../model/
model_se_r4_1.5_10foldpretrainedbest.h5',
    )
    main(cfg, cnn_model)

    cfg = Config(
        duration=5,
        name='se',
        lr=0.0005,
```

```
        batch_size=32,
        rnn_unit=128,
        momentum=0.85,
        mixup_prob=0.6,
        lm=0.01,
        pool_mode=('max', 'avemax1'),
        x1_rate=0.7,
        n_folds=10,
        get_backbone=get_se_backbone,
        pretrained='../model/sepretrainedbest.h5',
    )
    main(cfg, cnn_model)
```

----
5 predict.py

```
import pandas as pd
from utils import *
from iterstrat.ml_stratifiers import
MultilabelStratifiedKFold
import keras.backend as K
from sklearn.metrics import
label_ranking_average_precision_score
from tqdm import tqdm
from models import *

def get_oofp(cfg, get_model):
    if True: # load data
        df = pd.read_csv(f'../input/train_curated.csv')
        y = split_and_label(df['labels'].values)
        x = train_dir + df['fname'].values
        # # x = prepocess_para(x)

        x = [librosa.load(path, 44100)[0] for path in
tqdm(x)]
        x = [librosa.effects.trim(data)[0] for data in
tqdm(x)]
        # with open('../input/tr_logmel.pkl', 'rb') as f:
        #     x = pickle.load(f)
        gfeat = np.load('../input/gfeat.npy')[:len(y)]

    mskfold = MultilabelStratifiedKFold(cfg.n_folds,
```

```
shuffle=False, random_state=66666)
    folds = list(mskfold.split(x, y))
    # te_folds = list(mskfold.split(te_x,
(te_y>0.5).astype(int)))

    oofp = np.zeros_like(y)
    model = get_model(cfg)
    for fold, (tr_idx, val_idx) in
tqdm(enumerate(folds)):

        if True: # init
            model.load_weights(f"../model/{cfg.name}
{fold}.h5")

        val_x, val_y, val_gfeat = [x[i] for i in
val_idx], y[val_idx], gfeat[val_idx]
        val_loaders = [FreeSound(val_x, val_gfeat,
val_y, cfg, f'pred{i + 1}', 40) for i in range(3)]

        val_pred = [model.predict_generator(vl,
workers=4) for vl in val_loaders]
        ave_val_pred = np.average(val_pred, axis=0)
        oofp[val_idx] = ave_val_pred

    print(label_ranking_average_precision_score(y,oofp))

    np.save(f'../output/{cfg.name}oof',oofp)

def predict_test(cfg,get_model):
    test = pd.read_csv('../input/sample_submission.csv')
    x = [librosa.load(path, 44100)[0] for path in
tqdm('../input/audio_test/' + test['fname'].values)]
    Gfeat = np.array([get_global_feat(data, 128) for
data in tqdm(x)])
    x = [librosa.effects.trim(data)[0] for data in
tqdm(x)]

    y =
test[test.columns[1:].tolist()].values.astype(float)
    model = get_model(cfg)
    for fold in range(cfg.n_folds):
        val_loaders = [FreeSound(x, Gfeat, y, cfg,
f'pred{i + 1}',40) for i in range(3)]
        model.load_weights(f"../model/{cfg.name}
{fold}.h5")
        y += np.average([model.predict_generator(vl,
```

```
workers=4, verbose=1) for vl in val_loaders], axis=0)
    y /= cfg.n_folds

    np.save(f'../output/{cfg.name}pred',y)



if __name__ == '__main__':

    cfg = Config(
        duration=5,
        name='v1mix',
        lr=0.0005,
        batch_size=32,
        rnn_unit=128,
        momentum=0.85,
        mixup_prob=0.6,
        lm=0.01,
        pool_mode=('max', 'avemax1'),
        n_folds=10,
        get_backbone=get_conv_backbone,
    )
    get_oofp(cfg, cnn_model)
    predict_test(cfg, cnn_model)

    cfg = Config(
        duration=5,
        name='max3exam',
        lr=0.0005,
        batch_size=32,
        rnn_unit=128,
        momentum=0.85,
        mixup_prob=0.6,
        lm=0.01,
        pool_mode=('max', 'avemax3'),
        x1_rate=0.7,
        n_folds=10,
        get_backbone=get_conv_backbone,
    )
    get_oofp(cfg, cnn_model)
    predict_test(cfg, cnn_model)

    cfg = Config(
        duration=5,
        name='model_MSC_se_r4_1.0_10fold',
        lr=0.0005,
```

```
        batch_size=32,
        rnn_unit=128,
        momentum=0.85,
        mixup_prob=0.6,
        lm=0.01,
        pool_mode=('max', 'avemax1'),
        x1_rate=0.7,
        n_folds=10,
        get_backbone=model_se_MSC,
        w_ratio=1,
    )
    get_oofp(cfg, cnn_model)
    predict_test(cfg, cnn_model)

    cfg = Config(
        duration=5,
        name='model_MSC_se_r4_2.0_10fold',
        lr=0.0005,
        batch_size=32,
        rnn_unit=128,
        momentum=0.85,
        mixup_prob=0.6,
        lm=0.01,
        pool_mode=('max', 'avemax1'),
        x1_rate=0.7,
        n_folds=10,
        get_backbone=model_se_MSC,
        w_ratio=2.0,
    )
    get_oofp(cfg, cnn_model)
    predict_test(cfg, cnn_model)

    cfg = Config(
        duration=5,
        name='model_se_r4_1.5_10fold',
        lr=0.0005,
        batch_size=32,
        rnn_unit=128,
        momentum=0.85,
        mixup_prob=0.6,
        lm=0.01,
        pool_mode=('max', 'avemax1'),
        x1_rate=0.7,
        n_folds=10,
        get_backbone=model_se_MSC,
        w_ratio=1.5,
```

```
    )
    get_oofp(cfg, cnn_model)
    predict_test(cfg, cnn_model)

    cfg = Config(
        duration=5,
        name='se',
        lr=0.0005,
        batch_size=32,
        rnn_unit=128,
        momentum=0.85,
        mixup_prob=0.6,
        lm=0.01,
        pool_mode=('max', 'avemax3'),
        x1_rate=0.7,
        n_folds=10,
        get_backbone=get_se_backbone,
    )
    get_oofp(cfg, cnn_model)
    predict_test(cfg, cnn_model)


----------
6 ensemble.py

from utils import *
from sklearn.metrics import
label_ranking_average_precision_score
from iterstrat.ml_stratifiers import
MultilabelStratifiedKFold
from models import stacker
from keras import backend as K





def stacking(cfg,files):

    print(list(files.keys()))
    ave_oof, ave_pred = average(cfg,files,True)
    tr_oof_files = [np.load(f'../output/{name}oof.npy')
[:,:,np.newaxis] for name in files.keys()] +
[ave_oof[:,:,np.newaxis]]
    tr_oof = np.concatenate(tr_oof_files,axis=-1)
    test_files = [np.load(f'../output/{name}pred.npy')
[:,:,np.newaxis] for name in files.keys()] +
```

```
[ave_pred[:,:,np.newaxis]]
    test_pred = np.concatenate(test_files,axis=-1)
    df = pd.read_csv(f'../input/train_curated.csv')
    y = split_and_label(df['labels'].values)


    mskfold = MultilabelStratifiedKFold(cfg.n_folds,
shuffle=False, random_state=66666)
    folds = list(mskfold.split(y, y))

    predictions = np.zeros_like(test_pred)[:,:,0]
    oof = np.zeros_like((y))
    for fold, (tr_idx, val_idx) in enumerate(folds):
        print('fold ',fold)
        if True:  # init
            K.clear_session()
            model = stacker(cfg,tr_oof.shape[2])
            best_epoch = 0
            best_score = -1

        for epoch in range(1000):
            if epoch - best_epoch > 15:
                break


            tr_x, tr_y = tr_oof[tr_idx], y[tr_idx]
            val_x, val_y = tr_oof[val_idx], y[val_idx]

            val_pred = model.predict(val_x)

            score =
label_ranking_average_precision_score(val_y, val_pred)

            if score > best_score:
                best_score = score
                best_epoch = epoch
                oof[val_idx] = val_pred
                model.save_weights(f"../model/
stacker{cfg.name}{fold}.h5")

            model.fit(x=tr_x, y=tr_y, batch_size=cfg.bs,
verbose=0)
            print(f'{epoch} score {score} ,  best
{best_score}...')

        model.load_weights(f"../model/stacker{cfg.name}
```

```
{fold}.h5")
        predictions += model.predict(test_pred)

    print('lrap: ',
label_ranking_average_precision_score(y, oof))
    predictions /= cfg.n_folds
    print(label_ranking_average_precision_score(y,oof))
    test = pd.read_csv('../input/sample_submission.csv')
    test.loc[:, test.columns[1:].tolist()] = predictions
    test.to_csv('submission.csv', index=False)

def average(cfg,files,return_pred = False):
    df = pd.read_csv(f'../input/train_curated.csv')
    y = split_and_label(df['labels'].values)

    result = 0
    oof = 0
    all_w = 0
    for name,w in files.items():
        oof += w * np.load(f'../output/{name}oof.npy')
        print(name,'lrap
',label_ranking_average_precision_score(y,np.load(f'../
output/{name}oof.npy')))
        result += w * np.load(f'../output/{name}
pred.npy')
        all_w += w

    oof /= all_w
    result /= all_w
    print(label_ranking_average_precision_score(y,oof))
    if return_pred:
        return oof,result
    test = pd.read_csv('../input/sample_submission.csv')
    test.loc[:, test.columns[1:].tolist()] = result
    test.to_csv('../submissions/submission.csv',
index=False)
    # print(test)



if __name__ == '__main__':

    cfg = Config(n_folds=10,lr = 0.0001, batch_size=40)
    # stacking(cfg,{
    #     'model_MSC_se_r4_1.0_10fold_withpretrain_e28_':
1.0,
```

```
    #     'max3exam':2.1,
    #     'v1mix':2.4,
    #     'model_MSC_se_r4_2.0_10fold_withpretrain_e28_':
1.0,
    #     # 'model_se_r4_1.5_10fold_withpretrain_e28_':
1.0,
    #     'se_':1,
    #     # 'concat_v1':0,
    #     'se_concat':1,
    #
    # })

    # stacking(cfg, {
    #
'model_MSC_se_r4_1.0_10fold_withpretrain_e28_': 1.0,
    #     'max3exam': 1.9,
    #     'v1mix': 2.1,
    #
'model_MSC_se_r4_2.0_10fold_withpretrain_e28_': 1.0,
    #     'model_se_r4_1.5_10fold_withpretrain_e28_':1.0,
    #     'se_': 0,
    # })

    stacking(cfg, {
        'model_MSC_se_r4_1.0_10fold': 1.0,
        'max3exam': 1.9,
        'v1mix': 2.1,
        'model_MSC_se_r4_2.0_10fold': 1.0,
        'model_se_r4_1.5_10fold': 1.0,
        'se_': 0,
    })

----------
```

----

1 config.py

```
import pandas as pd

train_dir = '../input/audio_train/'

submit = pd.read_csv('../input/sample_submission.csv')
i2label = label_columns = submit.columns[1:].tolist()
label2i = {label:i for i,label in enumerate(i2label)}

n_classes = 80

assert len(label2i) == n_classes




class Config(object):
    def __init__(self,
        batch_size=32,
        n_folds=5,
        lr=0.0005,
        duration = 5,
        name = 'v1',
        milestones = (14,21,28),
        rnn_unit = 128,
        lm = 0.0,
        momentum = 0.85,
        mixup_prob = -1,
        folds=None,
        pool_mode = ('max','avemax1'),
        pretrained = None,
        gamma = 0.5,
        x1_rate = 0.7,
        w_ratio = 1,
        get_backbone = None
    ):
```

```
        self.maxlen = int((duration*44100))
        self.bs = batch_size
        self.n_folds = n_folds
        self.name = name
        self.lr = lr
        self.milestones = milestones
        self.rnn_unit = rnn_unit
        self.lm = lm
        self.momentum = momentum
        self.mixup_prob = mixup_prob
        self.folds = list(range(n_folds)) if folds is
None else folds
        self.pool_mode = pool_mode
        self.pretrained = pretrained
        self.gamma = gamma
        self.x1_rate = x1_rate
        self.w_ratio = w_ratio
        self.get_backbone = get_backbone

    def __str__(self):
        return ',\t'.join(['%s:%s' % item for item in
self.__dict__.items()])
```

```
----------
2 models.py

from keras.layers import *
from time_frequency import Melspectrogram, AdditiveNoise
from keras.optimizers import Nadam,SGD
from keras.constraints import *
from keras.initializers import *
from keras.models import Model
from config import *

EPS = 1e-8

def squeeze_excitation_layer(x, out_dim, ratio = 4):
    '''
    SE module performs inter-channel weighting.
    '''
    squeeze = GlobalAveragePooling2D()(x)
    excitation = Dense(units=out_dim // ratio)(squeeze)
```

```
    excitation = Activation('relu')(excitation)
    excitation = Dense(units=out_dim)(excitation)
    excitation = Activation('sigmoid')(excitation)
    excitation = Reshape((1, 1, out_dim))(excitation)

    scale = multiply([x, excitation])
    return scale

def
conv_se_block(x,filters,pool_stride,pool_size,pool_mode,cfg,
ratio = 4):

    x = Conv2D(filters=filters, kernel_size=3,
strides=1, padding='same')(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = squeeze_excitation_layer(x,
out_dim=filters,ratio=ratio)
    x = pooling_block(x, pool_size[0], pool_stride[0],
pool_mode[0], cfg)

    x = Conv2D(filters=filters, kernel_size=3,
strides=1, padding='same')(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = squeeze_excitation_layer(x,
out_dim=filters,ratio=ratio)
    x = pooling_block(x, pool_size[1], pool_stride[1],
pool_mode[1], cfg)

    return x

def AveMaxPool(x, pool_size,stride, ave_axis):
    if isinstance(pool_size,int):
        pool_size1,pool_size2 = pool_size, pool_size
    else:
        pool_size1,pool_size2 = pool_size
    if ave_axis == 2:
        x = AveragePooling2D(pool_size=(1,pool_size1),
padding='same', strides=(1,stride))(x)
        x = MaxPool2D(pool_size=(pool_size2,1),
padding='same', strides=(stride,1))(x)
    elif ave_axis == 1:
        x = AveragePooling2D(pool_size=(pool_size1,1),
padding='same', strides=(stride,1))(x)
        x = MaxPool2D(pool_size=(1,pool_size2),
```

```
padding='same', strides=(1,stride))(x)
    elif ave_axis == 3:
        x = MaxPool2D(pool_size=(1,pool_size1),
padding='same', strides=(1,stride))(x)
        x = AveragePooling2D(pool_size=(pool_size2, 1),
padding='same', strides=(stride, 1))(x)
    elif ave_axis == 4:
        x = MaxPool2D(pool_size=(pool_size1, 1),
padding='same', strides=(stride, 1))(x)
        x = AveragePooling2D(pool_size=(1, pool_size2),
padding='same', strides=(1, stride))(x)
    else:
        raise RuntimeError("axis error")
    return x

def pooling_block(x,pool_size,stride,pool_mode, cfg):
    if pool_mode == 'max':
        x = MaxPool2D(pool_size=pool_size,
padding='same', strides=stride)(x)
    elif pool_mode == 'ave':
        x = AveragePooling2D(pool_size=pool_size,
padding='same', strides=stride)(x)
    elif pool_mode == 'avemax1':
        x = AveMaxPool(x, pool_size=pool_size,
stride=stride, ave_axis=1)
    elif pool_mode == 'avemax2':
        x = AveMaxPool(x, pool_size=pool_size,
stride=stride, ave_axis=2)
    elif pool_mode == 'avemax3':
        x = AveMaxPool(x, pool_size=pool_size,
stride=stride, ave_axis=3)
    elif pool_mode == 'avemax4':
        x = AveMaxPool(x, pool_size=pool_size,
stride=stride, ave_axis=4)
    elif pool_mode == 'conv':
        x = Lambda(lambda
x:K.expand_dims(K.permute_dimensions(x,
(0,3,1,2)),axis=-1))(x)
        x = TimeDistributed(Conv2D(filters=1,
kernel_size=pool_size, strides=stride, padding='same',
use_bias=False))(x)
        x = Lambda(lambda
x:K.permute_dimensions(K.squeeze(x,axis=-1),(0,2,3,1)))
(x)
    elif pool_mode is None:
        x = x
```

```
    else:
        raise RuntimeError('pool mode error')
    return x

def
conv_block(x,filters,pool_stride,pool_size,pool_mode,cfg):


    x = Conv2D(filters=filters, kernel_size=3,
strides=1, padding='same')(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = pooling_block(x, pool_size[0], pool_stride[0],
pool_mode[0], cfg)

    x = Conv2D(filters=filters, kernel_size=3,
strides=1, padding='same')(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = pooling_block(x, pool_size[1], pool_stride[1],
pool_mode[1], cfg)
    return x


def conv_cat_block(x, filters, pool_stride, pool_size,
pool_mode, cfg):
    x = Conv2D(filters=filters, kernel_size=3,
strides=1, padding='same')(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = pooling_block(x, pool_size[0], pool_stride[0],
pool_mode[0], cfg)

    x1 = x
    x = Conv2D(filters=filters, kernel_size=3,
strides=1, padding='same')(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)

    ## concat
    x = concatenate([x1, x])
    x = Conv2D(filters=filters, kernel_size=1,
strides=1, padding='same')(x)
    x = pooling_block(x, pool_size[1], pool_stride[1],
pool_mode[1], cfg)
```

```
    return x


def conv_se_cat_block(x, filters, pool_stride,
pool_size, pool_mode, cfg):
    x = Conv2D(filters=filters, kernel_size=3,
strides=1, padding='same')(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = squeeze_excitation_layer(x, out_dim=filters,
ratio=4)
    x = pooling_block(x, pool_size[0], pool_stride[0],
pool_mode[0], cfg)
    x1 = x
    x = Conv2D(filters=filters, kernel_size=3,
strides=1, padding='same')(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = squeeze_excitation_layer(x, out_dim=filters,
ratio=4)
    ## concat
    x = concatenate([x1, x])
    x = Conv2D(filters=filters, kernel_size=1,
strides=1, padding='same')(x)
    x = pooling_block(x, pool_size[1], pool_stride[1],
pool_mode[1], cfg)

    return x



def pixelShuffle(x):
    _,h,w,c = K.int_shape(x)
    bs = K.shape(x)[0]
    assert w%2==0
    x = K.reshape(x,(bs,h,w//2,c*2))

    # assert h % 2 == 0
    # x = K.permute_dimensions(x,(0,2,1,3))
    # x = K.reshape(x,(bs,w//2,h//2,c*4))
    # x = K.permute_dimensions(x,(0,2,1,3))
    return x

def get_se_backbone(x, cfg):

    x = Conv2D(64, kernel_size=3, padding='same',
```

```
use_bias=False)(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = squeeze_excitation_layer(x, out_dim=64, ratio=4)
    # backbone
    x = conv_se_block(x, 96, (1, 2), (3, 2),
cfg.pool_mode, cfg)
    x = conv_se_block(x, 128, (1, 2), (3, 2),
cfg.pool_mode, cfg)
    x = conv_se_block(x, 256, (1, 2), (3, 3),
cfg.pool_mode, cfg)
    x = conv_se_block(x, 512, (1, 2), (3, 2), (None,
None), cfg)  ## [bs,  54, 8, 512]

    # global pooling
    x = Lambda(pixelShuffle)(x)  ## [bs,  54, 4, 1024]
    x = Lambda(lambda x: K.max(x, axis=1))(x)
    x = Lambda(lambda x: K.mean(x, axis=1))(x)

    return x

def get_conv_backbone(x, cfg):

    # input stem
    x = Conv2D(64, kernel_size=3, padding='same',
use_bias=False)(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)

    # backbone
    x = conv_block(x, 96, (1, 2), (3, 2), cfg.pool_mode,
cfg)
    x = conv_block(x, 128, (1, 2), (3, 2),
cfg.pool_mode, cfg)
    x = conv_block(x, 256, (1, 2), (3, 3),
cfg.pool_mode, cfg)
    x = conv_block(x, 512, (1, 2), (3, 2), (None, None),
cfg)  ## [bs,  54, 8, 512]

    # global pooling
    x = Lambda(pixelShuffle)(x)  ## [bs,  54, 4, 1024]
    x = Lambda(lambda x: K.max(x, axis=1))(x)
    x = Lambda(lambda x: K.mean(x, axis=1))(x)

    return x
```

```
def get_se_cat_backbone(x,cfg):


    x = Conv2D(64, kernel_size=3,
padding='same',use_bias=False)(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = squeeze_excitation_layer(x, out_dim=64,ratio=4)
    # backbone
    x = conv_se_cat_block(x, 96, (1,2), (3,2),
cfg.pool_mode, cfg)
    x = conv_se_cat_block(x, 128, (1,2), (3,2),
cfg.pool_mode, cfg)
    x = conv_se_cat_block(x, 256, (1,2), (3,3),
cfg.pool_mode, cfg)
    x = conv_se_cat_block(x, 512, (1,2), (3,2),
(None,None), cfg) ## [bs,  54, 8, 512]

    # global pooling
    x = Lambda(pixelShuffle)(x)  ## [bs,  54, 4, 1024]
    x = Lambda(lambda x: K.max(x, axis=1))(x)
    x = Lambda(lambda x: K.mean(x, axis=1))(x)
    return x

def get_concat_backbone(x, cfg):

    # input stem
    x = Conv2D(64, kernel_size=3, padding='same',
use_bias=False)(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)

    # backbone
    x = conv_cat_block(x, 96, (1, 2), (3, 2),
cfg.pool_mode, cfg)
    x = conv_cat_block(x, 128, (1, 2), (3, 2),
cfg.pool_mode, cfg)
    x = conv_cat_block(x, 256, (1, 2), (3, 3),
cfg.pool_mode, cfg)
    x = conv_cat_block(x, 512, (1, 2), (3, 2), (None,
None), cfg)  ## [bs,  54, 8, 512]

    # global pooling
    x = Lambda(pixelShuffle)(x)  ## [bs,  54, 4, 1024]
    x = Lambda(lambda x: K.max(x, axis=1))(x)
    x = Lambda(lambda x: K.mean(x, axis=1))(x)
```

```
    return x

def model_se_MSC(x, cfg):
    ratio = 4
    # input stem
    x_3 = Conv2D(32, kernel_size=3, padding='same', 
use_bias=False)(x)
    x_5 = Conv2D(32, kernel_size=5, padding='same', 
use_bias=False)(x)
    x_7 = Conv2D(32, kernel_size=7, padding='same', 
use_bias=False)(x)

    x = concatenate([x_3, x_5, x_7])
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = squeeze_excitation_layer(x, out_dim=96, 
ratio=ratio)

    w_ratio = cfg.w_ratio
    # backbone
    x = conv_se_block(x, int(96 * w_ratio), (1, 2), (3, 
2), cfg.pool_mode, cfg, ratio=ratio)
    x = conv_se_block(x, int(128 * w_ratio), (1, 2), (3, 
2), cfg.pool_mode, cfg, ratio=ratio)
    x = conv_se_block(x, int(256 * w_ratio), (1, 2), (3, 
3), cfg.pool_mode, cfg, ratio=ratio)
    x = conv_se_block(x, int(512 * w_ratio), (1, 2), (3, 
2), (None, None), cfg, ratio=ratio)

    # global pooling
    x = Lambda(pixelShuffle)(x)
    x = Lambda(lambda x: K.max(x, axis=1))(x)
    x = Lambda(lambda x: K.mean(x, axis=1))(x)
    return x

def cnn_model(cfg):


    x_in = Input((cfg.maxlen,), name='audio')
    feat_in = Input((1,), name='other')
    feat = feat_in

    gfeat_in = Input((128, 12), name='global_feat')
    gfeat = BatchNormalization()(gfeat_in)
    gfeat = Bidirectional(CuDNNGRU(cfg.rnn_unit, 
```

```
return_sequences=True), merge_mode='sum')(gfeat)
    gfeat = Bidirectional(CuDNNGRU(cfg.rnn_unit,
return_sequences=True), merge_mode='sum')(gfeat)
    gfeat = GlobalMaxPooling1D()(gfeat)

    x = Lambda(lambda t: K.expand_dims(t, axis=1))(x_in)
    x_mel = Melspectrogram(n_dft=1024, n_hop=512,
input_shape=(1, K.int_shape(x_in)[1]),
                           # n_hop -> stride   n_dft
kernel_size
                           padding='same', sr=44100,
n_mels=64,
                           power_melgram=2,
return_decibel_melgram=True,
                           trainable_fb=False,
trainable_kernel=False,

image_data_format='channels_last', trainable=False)(x)

    x_mel = Lambda(lambda x: K.permute_dimensions(x,
pattern=(0, 2, 1, 3)))(x_mel)
    x = cfg.get_backbone(x_mel, cfg)
    x = concatenate([x, gfeat, feat])
    output = Dense(units=n_classes, activation='sigmoid')
(x)

    y_in = Input((n_classes,), name='y')
    y = y_in

    def get_loss(x):
        y_true, y_pred = x
        loss1 = K.mean(K.binary_crossentropy(y_true,
y_pred))
        return loss1

    loss = Lambda(get_loss)([y, output])
    model = Model(inputs=[x_in, feat_in, gfeat_in,
y_in], outputs=[output])

    if cfg.pretrained is not None:
        model.load_weights("../model/
{}.h5".format(cfg.pretrained))
        print('load_pretrained_success...')

    model.add_loss(loss)
    model.compile(
```

```
        # loss=get_loss,
        optimizer=Nadam(lr=cfg.lr),
    )
    return model


class normNorm(Constraint):
    def __init__(self, axis=0):
        self.axis = axis

    def __call__(self, w):
        # w = K.relu(w)
        # w = K.clip(w,-0.5,1)
        w /= (K.sum(w**2, axis=self.axis,
keepdims=True)**0.5)
        return w

    def get_config(self):
        return {'axis': self.axis}

def stacker(cfg,n):
    def kinit(shape, name=None):
        value = np.zeros(shape)
        value[:, -1] = 1
        return K.variable(value, name=name)


    x_in = Input((80,n))
    x = x_in
    # x = Lambda(lambda x: 1.5*x)(x)
    x =
LocallyConnected1D(1,1,kernel_initializer=kinit,kernel_constraint=n
(x)
    x = Flatten()(x)
    x = Dense(80, use_bias=False,
kernel_initializer=Identity(1))(x)
    x = Lambda(lambda x: (x - 1.6))(x)
    x = Activation('tanh')(x)
    x = Lambda(lambda x:(x+1)*0.5)(x)

    model = Model(inputs=x_in, outputs=x)
    model.compile(
        loss='binary_crossentropy',
        optimizer=Nadam(lr=cfg.lr),
    )
    return model
```

```
if __name__ == '__main__':
    cfg = Config()
    model = cnn_model(cfg)
    print(model.summary())
```

```
----------
3 time_frequency.py

# -*- coding: utf-8 -*-
from __future__ import absolute_import
import numpy as np
import keras
from keras import backend as K
from keras.engine import Layer
from keras.utils.conv_utils import conv_output_length
import librosa


def mel(sr, n_dft, n_mels=128, fmin=0.0, fmax=None,
htk=False, norm=1):
    """[np] create a filterbank matrix to combine stft
bins into mel-frequency bins
    use Slaney (said Librosa)

    n_mels: numbre of mel bands
    fmin : lowest frequency [Hz]
    fmax : highest frequency [Hz]
        If `None`, use `sr / 2.0`
    """
    return librosa.filters.mel(sr=sr, n_fft=n_dft,
n_mels=n_mels,
                               fmin=fmin, fmax=fmax,
                               htk=htk,
norm=norm).astype(K.floatx())

def amplitude_to_decibel(x, amin=1e-10,
dynamic_range=80.0):
    """[K] Convert (linear) amplitude to decibel
(log10(x)).
```

```
    x: Keras *batch* tensor or variable. It has to be
batch because of sample-wise `K.max()`.
    amin: minimum amplitude. amplitude smaller than
`amin` is set to this.
    dynamic_range: dynamic_range in decibel
    """
    log_spec = 10 * K.log(K.maximum(x, amin)) /
np.log(10).astype(K.floatx())
    if K.ndim(x) > 1:
        axis = tuple(range(K.ndim(x))[1:])
    else:
        axis = None

    log_spec = log_spec - K.max(log_spec, axis=axis,
keepdims=True)  # [-?, 0]
    log_spec = K.maximum(log_spec, -1 * dynamic_range)
# [-80, 0]
    return log_spec

def get_stft_kernels(n_dft):
    """[np] Return dft kernels for real/imagnary parts
assuming
        the input . is real.
    An asymmetric hann window is used
(scipy.signal.hann).

    Parameters
    ----------
    n_dft : int > 0 and power of 2 [scalar]
        Number of dft components.

    Returns
    -------
        |  dft_real_kernels : np.ndarray
[shape=(nb_filter, 1, 1, n_win)]
        |  dft_imag_kernels : np.ndarray
[shape=(nb_filter, 1, 1, n_win)]

    * nb_filter = n_dft/2 + 1
    * n_win = n_dft

    """
    assert n_dft > 1 and ((n_dft & (n_dft - 1)) == 0), \
        ('n_dft should be > 1 and power of 2, but n_dft
== %d' % n_dft)
```

```
    nb_filter = int(n_dft // 2 + 1)

    # prepare DFT filters
    timesteps = np.array(range(n_dft))
    w_ks = np.arange(nb_filter) * 2 * np.pi /
float(n_dft)
    dft_real_kernels = np.cos(w_ks.reshape(-1, 1) *
timesteps.reshape(1, -1))
    dft_imag_kernels = -np.sin(w_ks.reshape(-1, 1) *
timesteps.reshape(1, -1))

    # windowing DFT filters
    dft_window = librosa.filters.get_window('hann',
n_dft, fftbins=True)  # _hann(n_dft, sym=False)
    dft_window = dft_window.astype(K.floatx())
    dft_window = dft_window.reshape((1, -1))
    dft_real_kernels = np.multiply(dft_real_kernels,
dft_window)
    dft_imag_kernels = np.multiply(dft_imag_kernels,
dft_window)

    dft_real_kernels = dft_real_kernels.transpose()
    dft_imag_kernels = dft_imag_kernels.transpose()
    dft_real_kernels = dft_real_kernels[:, np.newaxis,
np.newaxis, :]
    dft_imag_kernels = dft_imag_kernels[:, np.newaxis,
np.newaxis, :]

    return dft_real_kernels.astype(K.floatx()),
dft_imag_kernels.astype(K.floatx())

class Spectrogram(Layer):
    """
    ### `Spectrogram`

    ```python
    kapre.time_frequency.Spectrogram(n_dft=512,
n_hop=None, padding='same',

power_spectrogram=2.0, return_decibel_spectrogram=False,

trainable_kernel=False, image_data_format='default',
                                     **kwargs)
    ```

    Spectrogram layer that outputs spectrogram(s) in 2D
image format.
```

Parameters
* n_dft: int > 0 [scalar]
  - The number of DFT points, presumably power of 2.
  - Default: ``512``

* n_hop: int > 0 [scalar]
  - Hop length between frames in sample, probably <= ``n_dft``.
  - Default: ``None`` (``n_dft / 2`` is used)

* padding: str, ``'same'`` or ``'valid'``.
  - Padding strategies at the ends of signal.
  - Default: ``'same'``

* power_spectrogram: float [scalar],
  - ``2.0`` to get power-spectrogram, ``1.0`` to get amplitude-spectrogram.
  - Usually ``1.0`` or ``2.0``.
  - Default: ``2.0``

* return_decibel_spectrogram: bool,
  - Whether to return in decibel or not, i.e. returns log10(amplitude spectrogram) if ``True``.
  - Recommended to use ``True``, although it's not by default.
  - Default: ``False``

* trainable_kernel: bool
  - Whether the kernels are trainable or not.
  - If ``True``, Kernels are initialised with DFT kernels and then trained.
  - Default: ``False``

* image_data_format: string, ``'channels_first'`` or ``'channels_last'``.
  - The returned spectrogram follows this image_data_format strategy.
  - If ``'default'``, it follows the current Keras session's setting.
  - Setting is in ``./keras/keras.json``.
  - Default: ``'default'``

Notes
* The input should be a 2D array, ``(audio_channel, audio_length)``.

```
     * E.g., ``(1, 44100)`` for mono signal, ``(2,
44100)`` for stereo signal.
     * It supports multichannel signal input, so
``audio_channel`` can be any positive integer.
     * The input shape is not related to keras
`image_data_format()` config.

    #### Returns

    A Keras layer

     * abs(Spectrogram) in a shape of 2D data, i.e.,
     * `(None, n_channel, n_freq, n_time)` if
``'channels_first'``,
     * `(None, n_freq, n_time, n_channel)` if
``'channels_last'``,


    """

    def __init__(self, n_dft=512, n_hop=None,
padding='same',
                 power_spectrogram=2.0,
return_decibel_spectrogram=False,
                 trainable_kernel=False,
image_data_format='default', **kwargs):
        assert n_dft > 1 and ((n_dft & (n_dft - 1)) ==
0), \
            ('n_dft should be > 1 and power of 2, but
n_dft == %d' % n_dft)
        assert isinstance(trainable_kernel, bool)
        assert isinstance(return_decibel_spectrogram,
bool)
        # assert padding in ('same', 'valid')
        if n_hop is None:
            n_hop = n_dft // 2

        assert image_data_format in ('default',
'channels_first', 'channels_last')
        if image_data_format == 'default':
            self.image_data_format =
K.image_data_format()
        else:
            self.image_data_format = image_data_format

        self.n_dft = n_dft
```

```
        assert n_dft % 2 == 0
        self.n_filter = n_dft // 2 + 1
        self.trainable_kernel = trainable_kernel
        self.n_hop = n_hop
        self.padding = padding
        self.power_spectrogram = float(power_spectrogram)
        self.return_decibel_spectrogram =
return_decibel_spectrogram
        super(Spectrogram, self).__init__(**kwargs)

    def build(self, input_shape):
        self.n_ch = input_shape[1]
        self.len_src = input_shape[2]
        self.is_mono = (self.n_ch == 1)
        if self.image_data_format == 'channels_first':
            self.ch_axis_idx = 1
        else:
            self.ch_axis_idx = 3
        if self.len_src is not None:
            assert self.len_src >= self.n_dft, 'Hey! The
input is too short!'

        self.n_frame = conv_output_length(self.len_src,
                                          self.n_dft,
                                          self.padding,
                                          self.n_hop)

        dft_real_kernels, dft_imag_kernels =
get_stft_kernels(self.n_dft)
        self.dft_real_kernels =
K.variable(dft_real_kernels, dtype=K.floatx(),
name="real_kernels")
        self.dft_imag_kernels =
K.variable(dft_imag_kernels, dtype=K.floatx(),
name="imag_kernels")
        # kernels shapes: (filter_length, 1, input_dim,
nb_filter)?
        if self.trainable_kernel:

self.trainable_weights.append(self.dft_real_kernels)

self.trainable_weights.append(self.dft_imag_kernels)
        else:

self.non_trainable_weights.append(self.dft_real_kernels)
```

```
self.non_trainable_weights.append(self.dft_imag_kernels)

        super(Spectrogram, self).build(input_shape)
        # self.built = True

    def compute_output_shape(self, input_shape):
        if self.image_data_format == 'channels_first':
            return input_shape[0], self.n_ch,
self.n_filter, self.n_frame
        else:
            return input_shape[0], self.n_filter,
self.n_frame, self.n_ch

    def call(self, x):
        output = self._spectrogram_mono(x[:, 0:1, :])
        if self.is_mono is False:
            for ch_idx in range(1, self.n_ch):
                output = K.concatenate((output,

self._spectrogram_mono(x[:, ch_idx:ch_idx + 1, :])),

axis=self.ch_axis_idx)
        if self.power_spectrogram != 2.0:
            output = K.pow(K.sqrt(output),
self.power_spectrogram)
        if self.return_decibel_spectrogram:
            output = amplitude_to_decibel(output)
        return output

    def get_config(self):
        config = {'n_dft': self.n_dft,
                  'n_hop': self.n_hop,
                  'padding': self.padding,
                  'power_spectrogram':
self.power_spectrogram,
                  'return_decibel_spectrogram':
self.return_decibel_spectrogram,
                  'trainable_kernel':
self.trainable_kernel,
                  'image_data_format':
self.image_data_format}
        base_config = super(Spectrogram,
self).get_config()
        return dict(list(base_config.items()) +
list(config.items()))
```

```
    def _spectrogram_mono(self, x):
        '''x.shape : (None, 1, len_src),
        returns 2D batch of a mono power-spectrogram'''
        x = K.permute_dimensions(x, [0, 2, 1])
        x = K.expand_dims(x, 3)  # add a dummy dimension
(channel axis)
        subsample = (self.n_hop, 1)
        output_real = K.conv2d(x, self.dft_real_kernels,
                               strides=subsample,
                               padding=self.padding,

data_format='channels_last')
        output_imag = K.conv2d(x, self.dft_imag_kernels,
                               strides=subsample,
                               padding=self.padding,

data_format='channels_last')
        output = output_real ** 2 + output_imag ** 2
        # now shape is (batch_sample, n_frame, 1, freq)
        if self.image_data_format == 'channels_last':
            output = K.permute_dimensions(output, [0, 3,
1, 2])
        else:
            output = K.permute_dimensions(output, [0, 2,
3, 1])
        return output


class Melspectrogram(Spectrogram):
    '''
    ### `Melspectrogram`
    ```python
    kapre.time_frequency.Melspectrogram(sr=22050,
n_mels=128, fmin=0.0, fmax=None,

power_melgram=1.0, return_decibel_melgram=False,

trainable_fb=False, **kwargs)
    ```
d
    Mel-spectrogram layer that outputs mel-
spectrogram(s) in 2D image format.

    Its base class is ``Spectrogram``.

    Mel-spectrogram is an efficient representation using
```

```
the property of human
    auditory system -- by compressing frequency axis
into mel-scale axis.

    #### Parameters
     * sr: integer > 0 [scalar]
       - sampling rate of the input audio signal.
       - Default: ``22050``

     * n_mels: int > 0 [scalar]
       - The number of mel bands.
       - Default: ``128``

     * fmin: float > 0 [scalar]
       - Minimum frequency to include in Mel-spectrogram.
       - Default: ``0.0``

     * fmax: float > ``fmin`` [scalar]
       - Maximum frequency to include in Mel-spectrogram.
       - If `None`, it is inferred as ``sr / 2``.
       - Default: `None`

     * power_melgram: float [scalar]
       - Power of ``2.0`` if power-spectrogram,
       - ``1.0`` if amplitude spectrogram.
       - Default: ``1.0``

     * return_decibel_melgram: bool
       - Whether to return in decibel or not, i.e.
returns log10(amplitude spectrogram) if ``True``.
       - Recommended to use ``True``, although it's not
by default.
       - Default: ``False``

     * trainable_fb: bool
       - Whether the spectrogram -> mel-spectrogram
filterbanks are trainable.
       - If ``True``, the frequency-to-mel matrix is
initialised with mel frequencies but trainable.
       - If ``False``, it is initialised and then frozen.
       - Default: `False`

     * htk: bool
       - Check out Librosa's `mel-spectrogram` or `mel`
option.
```

```
     * norm: float [scalar]
       - Check out Librosa's `mel-spectrogram` or `mel`
option.

     * **kwargs:
       - The keyword arguments of ``Spectrogram`` such
as ``n_dft``, ``n_hop``,
       - ``padding``, ``trainable_kernel``,
``image_data_format``.

    #### Notes
     * The input should be a 2D array, ``(audio_channel,
audio_length)``.
    E.g., ``(1, 44100)`` for mono signal, ``(2, 44100)``
for stereo signal.
     * It supports multichannel signal input, so
``audio_channel`` can be any positive integer.
     * The input shape is not related to keras
`image_data_format()` config.

    #### Returns

    A Keras layer
     * abs(mel-spectrogram) in a shape of 2D data, i.e.,
     * `(None, n_channel, n_mels, n_time)` if
``'channels_first'``,
     * `(None, n_mels, n_time, n_channel)` if
``'channels_last'``,

    '''

    def __init__(self,
                 sr=22050, n_mels=128, fmin=0.0,
fmax=None,
                 power_melgram=1.0,
return_decibel_melgram=False,
                 trainable_fb=False, htk=False, norm=1,
**kwargs):

        super(Melspectrogram, self).__init__(**kwargs)
        assert sr > 0
        assert fmin >= 0.0
        if fmax is None:
            fmax = float(sr) / 2
        assert fmax > fmin
        assert isinstance(return_decibel_melgram, bool)
```

```
        if 'power_spectrogram' in kwargs:
            assert kwargs['power_spectrogram'] == 2.0, \
                'In Melspectrogram, power_spectrogram
should be set as 2.0.'

        self.sr = int(sr)
        self.n_mels = n_mels
        self.fmin = fmin
        self.fmax = fmax
        self.return_decibel_melgram =
return_decibel_melgram
        self.trainable_fb = trainable_fb
        self.power_melgram = power_melgram
        self.htk = htk
        self.norm = norm

    def build(self, input_shape):
        super(Melspectrogram, self).build(input_shape)
        self.built = False
        # compute freq2mel matrix -->
        mel_basis = mel(self.sr, self.n_dft,
self.n_mels, self.fmin, self.fmax,
                                self.htk, self.norm)  #
(128, 1025) (mel_bin, n_freq)
        mel_basis = np.transpose(mel_basis)

        self.freq2mel = K.variable(mel_basis,
dtype=K.floatx())
        if self.trainable_fb:
            self.trainable_weights.append(self.freq2mel)
        else:

self.non_trainable_weights.append(self.freq2mel)
        self.built = True

    def compute_output_shape(self, input_shape):
        if self.image_data_format == 'channels_first':
            return input_shape[0], self.n_ch,
self.n_mels, self.n_frame
        else:
            return input_shape[0], self.n_mels,
self.n_frame, self.n_ch

    def call(self, x):
        power_spectrogram = super(Melspectrogram,
self).call(x)
```

```
        # now,  channels_first: (batch_sample, n_ch,
n_freq, n_time)
        #       channels_last: (batch_sample, n_freq,
n_time, n_ch)
        if self.image_data_format == 'channels_first':
            power_spectrogram =
K.permute_dimensions(power_spectrogram, [0, 1, 3, 2])
        else:
            power_spectrogram =
K.permute_dimensions(power_spectrogram, [0, 3, 2, 1])
        # now, whatever image_data_format,
(batch_sample, n_ch, n_time, n_freq)
        output = K.dot(power_spectrogram, self.freq2mel)
        if self.image_data_format == 'channels_first':
            output = K.permute_dimensions(output, [0, 1,
3, 2])
        else:
            output = K.permute_dimensions(output, [0, 3,
2, 1])
        if self.power_melgram != 2.0:
            output = K.pow(K.sqrt(output),
self.power_melgram)
        if self.return_decibel_melgram:
            output = amplitude_to_decibel(output)
        return output

    def get_config(self):
        config = {'sr': self.sr,
                  'n_mels': self.n_mels,
                  'fmin': self.fmin,
                  'fmax': self.fmax,
                  'trainable_fb': self.trainable_fb,
                  'power_melgram': self.power_melgram,
                  'return_decibel_melgram':
self.return_decibel_melgram,
                  'htk': self.htk,
                  'norm': self.norm}
        base_config = super(Melspectrogram,
self).get_config()
        return dict(list(base_config.items()) +
list(config.items()))



class AdditiveNoise(Layer):
```

```
    def __init__(self, power=0.1, random_gain=False,
noise_type='white', **kwargs):
        assert noise_type in ['white']
        self.supports_masking = True
        self.power = power
        self.random_gain = random_gain
        self.noise_type = noise_type
        self.uses_learning_phase = True
        super(AdditiveNoise, self).__init__(**kwargs)

    def call(self, x):
        if self.random_gain:
            noise_x = x +
K.random_normal(shape=K.shape(x),
                                                mean=0.,

stddev=np.random.uniform(0.0, self.power))
        else:
            noise_x = x +
K.random_normal(shape=K.shape(x),
                                                mean=0.,

stddev=self.power)

        return K.in_train_phase(noise_x, x)

    def get_config(self):
        config = {'power': self.power,
                  'random_gain': self.random_gain,
                  'noise_type': self.noise_type}
        base_config = super(AdditiveNoise,
self).get_config()
        return dict(list(base_config.items()) +
list(config.items()))

----------
```

```
   79 audio.py
   23 folds.py
    0 __init__.py
   80 padding.py
  235 training.py
  377 transforms.py
  128 utils.py
  922 total
  359 apc.py
 1249 classifiers.py
  395 cpc.py
    0 __init__.py
   57 losses.py
 2060 total
   0 __init__.py
  59 sound_dataset.py
   2 wcl_datasets.txt
  61 total
1     348 adversarial_test.py
2      32 create_class_map.py
3       1 dash.txt
4       0 datasets
5     154 evaluate_2d_cnn.py
6     439 finetune_hierarchical_cnn.py
7     201 LICENSE
8     133 linear_blend.py
9       0 networks
10      0 ops
11    124 predict_2d_cnn.py
12     220 README.md
13     190 relabel_noisy_data.py
14     100 requirements.txt
15     510 train_2d_cnn.py
16     278 train_apc.py
17     486 train_backbone_cnn.py
18    288 train_cpc.py
19     509 train_hierarchical_cnn.py
   4013 total
# 3rd place solution to Freesound Audio Tagging 2019 Challenge

My approach is outlined below.

**Models**

I used two types of models, both are based on convolutions. The
first type uses 2d convolutions and works on top of mel-scale sp
ectrograms, while the second uses 1d-convolutions on top of raw
STFT representations with relatively small window size like 256,
 so it's only 5 ms per frame or so. Both types of models are rel
atively shallow and consist of 10-12 convolutional layers (or 5-
6 resnet blocks) with a small number of filters. I use a form of
 deep supervision by applying global max pooling after each bloc
k (typically starting from the first or second block) and then c
oncatenating maxpool outputs from each layer to form the final f
eature vector which then goes to a 2-layer fully-connected class
ifier. I also tried using RNNs instead of a max pooling for some
```

models. It made results a bit worse, but RNN seemed to make different mistakes, so it turned out to be a good member of the final ensemble.

Frequency encoding

2d convolutions are position-invariant, so the output of a convolution would be the same regardless of where the feature is located. Spectrograms are not images, Y-axis corresponds to signal frequency, so it would be nice to assist a model by providing this sort of information. For this purpose, I used a linear frequency map going from -1 to 1 and concatenated it to input spectrogram as a second channel. It's hard to estimate now without retraining all the models how much gain I got from this little modification, but I can say It was no less than 0.005 in terms of local CV score.

This is not really a classification task

Most teams treated the problem as a multilabel classification and used a form of a binary loss such as binary cross entropy or focal loss. This approach is definitely valid, but in my experiments, it appeared to be a little suboptimal. The reason is the metric (lwlrap) is not a pure classification metric. Contrary to accuracy or f-score, it is based on *ranks*. So it wasn't really a surprise for me when I used a loss function based on ranks rather than on binary outputs, I got a huge improvement. Namely, I used something called LSEP (https://arxiv.org/abs/1704.03135) which is just a soft version of pairwise rank loss. It makes your model to score positive classes higher than negative ones, while a binary loss increases positive scores and decreases negative scores independently. When I switched to LSEP from BCE, I immediately got approximately 0.015 of improvement, and, as a nice bonus, my models started to converge much faster.

Data augmentation

I used two augmentation strategies. The first one is a modified MixUp. In contrast to the original approach, I used OR rule for mixing labels. I did so because a mix of two sounds still allows you to hear both. I tried the original approach with weighted targets on some point and my results got worse.

The second strategy is augmentations based on audio effects such as reverb, pitch, tempo and overdrive. I chose the parameters of these augmentations by carefully listening to augmented samples.

I have found augmentations to be very important for getting good results. I guess the total improvement I got from these two strategies is about 0.05 or so. I also tried several other approaches such as splitting the audio into several chunks and then shuffling them, replacing some parts of the original signals with silence and some other, but they didn't make my models better.

Training

I used quite large audio segments for training. For most of my models, I used segments from 8 to 12 seconds. I didn't use TTA for inference and used full-length audio instead.

Noisy data

I tried several unsupervised approaches such as [Contrastive Predicting Coding](https://arxiv.org/abs/1807.03748), but never managed to get good results from it.

I ended up applying a form of iterative pseudolabeling. I predicted new labels for the noisy subset using a model trained on curated data only, chose best 1k in terms of the agreement between the predicted labels and actual labels and added these samples to the curated subset with the original labels. I repeated the procedure using top 2k labels this time. I applied this approach several times until I reached 5k best noisy samples. At that point, predictions generated by a model started to diverge significantly from the actual noisy labels. I decided to discard the labels of the remaining noisy samples and simply used model prediction as actual labels. In total, I trained approximately 20 models using different subsets of the noisy train set with different pseudolabeling strategies.

Inference

I got a great speed-up by computing both STFT spectrograms and mel spectrograms on a GPU. I also grouped samples with similar lengths together to avoid excessive padding. These two methods combined with relatively small models allowed me to predict the first stage test set in only 1 minute by any of my models (5 folds).

Final ensemble

For the final solution, I used a simple average of 11 models trained with slightly different architectures (1d/2d cnn, rnn/no-rnn), slightly different subsets of the noisy set (see "noisy data" section) and slightly different hyperparameters.

Project structure

Main training scripts are `train_2d_cnn.py` and `train_hierarcical_cnn.py`. All classification models are defined in `networks/classifiers`. All data augmentations are defined in `ops/transforms`.

Setting up the environment

I recommend using some environment manager such as conda or virtualenv in order to avoid potential conflicts between different versions of packages. To install all required packages, simply run `pip install -r requirements.txt`. This might take up to 15 minutes depending on your internet connection speed.

Preparing data

I place all the data into `data/` directory, please adjust the f
ollowing code to match yours data location. Run

```bash
python create_class_map.py --train_df data/train_curated.csv --o
utput_file data/classmap.json
```

This simply creates a JSON file with deterministic classname->la
bel mapping used in all future experiments.

Running a basic 2d model

```bash
python train_2d_cnn.py \
  --train_df data/train_curated.csv \
  --train_data_dir data/train_curated/ \
  --classmap data/classmap.json \
  --device=cuda \
  --optimizer=adam \
  --folds 0 1 2 3 4 \
  --n_folds=5 \
  --log_interval=10 \
  --batch_size=20 \
  --epochs=20 \
  --accumulation_steps=1 \
  --save_every=20 \
  --num_conv_blocks=5 \
  --conv_base_depth=50 \
  --growth_rate=1.5 \
  --weight_decay=0.0 \
  --start_deep_supervision_on=1 \
  --aggregation_type=max \
  --lr=0.003 \
  --scheduler=1cycle_0.0001_0.005 \
  --test_data_dir data/test \
  --sample_submission data/sample_submission.csv \
  --num_workers=6 \
  --output_dropout=0.0 \
  --p_mixup=0.0 \
  --switch_off_augmentations_on=15 \
  --features=mel_2048_1024_128 \
  --max_audio_length=15 \
  --p_aug=0.0 \
  --label=basic_2d_cnn
```

Running a 2d model with augmentations


```bash
python train_2d_cnn.py \
  --train_df data/train_curated.csv \
  --train_data_dir data/train_curated/ \
```

```
  --classmap data/classmap.json \
  --device=cuda \
  --optimizer=adam \
  --folds 0 1 2 3 4 \
  --n_folds=5 \
  --log_interval=10 \
  --batch_size=20 \
  --epochs=100 \
  --accumulation_steps=1 \
  --save_every=20 \
  --num_conv_blocks=5 \
  --conv_base_depth=100 \
  --growth_rate=1.5 \
  --weight_decay=0.0 \
  --start_deep_supervision_on=1 \
  --aggregation_type=max \
  --lr=0.003 \
  --scheduler=1cycle_0.0001_0.005 \
  --test_data_dir data/test \
  --sample_submission data/sample_submission.csv \
  --num_workers=16 \
  --output_dropout=0.5 \
  --p_mixup=0.5 \
  --switch_off_augmentations_on=90 \
  --features=mel_2048_1024_128 \
  --max_audio_length=15 \
  --p_aug=0.75 \
  --label=2d_cnn
```

Note that each such run is followed by a creation of a new experiment subdirectory in the `experiments` folder. Each experiment has the following structure:

```bash
experiments/some_experiment/
¿¿¿ checkpoints
¿¿¿ command
¿¿¿ commit_hash
¿¿¿ config.json
¿¿¿ log
¿¿¿ predictions
¿¿¿ results.json
¿¿¿ summaries
```

Using a clean model to select noisy samples

Create a new predictions directory:

```mkdir predictions/```

Then, running

```bash
python predict_2d_cnn.py \

```
  --experiment=path_to_an_experiment (see above) \
  --test_df=data/train_noisy.csv \
  --test_data_dir=data/train_noisy/ \
  --output_df=predictions/noisy_probabilities.csv \
  --classmap=data/classmap.json \
  --device=cuda
```

creates a new csv file in the predictions folder with the class probabilties for the noisy dataset.

Running

```bash
python relabel_noisy_data.py \
  --noisy_df=data/train_noisy.csv \
  --noisy_predictions_df=predictions/noisy_probabilities.csv \
  --output_df=predictions/train_noisy_relabeled_1k.csv \
  --mode=scoring_1000
```

creates a new noisy dataframe where only top 1k labels in terms of agreement between the model and the actual labels are kept.

Running a 2d model with noisy data

```bash
python train_2d_cnn.py \
  --train_df data/train_curated.csv \
  --train_data_dir data/train_curated/ \
  --noisy_train_df predictions/ train_noisy_relabeled_1k.csv \
  --noisy_train_data_dir data/train_noisy/ \
  --classmap data/classmap.json \
  --device=cuda \
  --optimizer=adam \
  --folds 0 1 2 3 4 \
  --n_folds=5 \
  --log_interval=10 \
  --batch_size=20 \
  --epochs=150 \
  --accumulation_steps=1 \
  --save_every=20 \
  --num_conv_blocks=6 \
  --conv_base_depth=100 \
  --growth_rate=1.5 \
  --weight_decay=0.0 \
  --start_deep_supervision_on=1 \
  --aggregation_type=max \
  --lr=0.003 \
  --scheduler=1cycle_0.0001_0.005 \
  --test_data_dir data/test \
  --sample_submission data/sample_submission.csv \
  --num_workers=16 \
  --output_dropout=0.7 \
```

```
  --p_mixup=0.5 \
  --switch_off_augmentations_on=140 \
  --features=mel_2048_1024_128 \
  --max_audio_length=15 \
  --p_aug=0.75 \
  --label=2d_cnn_noisy
```

Note that `relabel_noisy_data.py` script supports multiple relabeling straregies. I mostly followed "scoring" strategy (selecting top-k noisy samples based on the agreement between the model and the actual labels), but after 5k noisy samples I switched to "relabelall-replacenan" strategy which is just a pseudolabeling (usage of the old model outputs) where the samples without any predictions are discarded.

======
absl-py==0.7.1
astor==0.7.1
attrs==19.1.0
audioread==2.1.6
backcall==0.1.0
bleach==3.1.0
certifi==2019.3.9
cffi==1.12.2
chardet==3.0.4
cycler==0.10.0
decorator==4.4.0
defusedxml==0.5.0
entrypoints==0.3
gast==0.2.2
grpcio==1.19.0
h5py==2.9.0
idna==2.8
ipykernel==5.1.0
ipython==7.4.0
ipython-genutils==0.2.0
ipywidgets==7.4.2
iterative-stratification==0.1.6
jedi==0.13.3
Jinja2==2.10.1
joblib==0.13.2
jsonschema==3.0.1
jupyter==1.0.0
jupyter-client==5.2.4
jupyter-console==6.0.0
jupyter-core==4.4.0
kaggle==1.5.3
Keras-Applications==1.0.7
Keras-Preprocessing==1.0.9
kiwisolver==1.0.1
librosa==0.6.3
llvmlite==0.28.0
mag==0.1
Markdown==3.1
MarkupSafe==1.1.1
matplotlib==3.0.3

```
mistune==0.8.4
mock==2.0.0
munch==2.3.2
nbconvert==5.4.1
nbformat==4.4.0
notebook==5.7.8
numba==0.43.1
numpy==1.16.2
pandas==0.24.2
pandocfilters==1.4.2
parso==0.4.0
pbr==5.1.3
pexpect==4.7.0
pickleshare==0.7.5
Pillow==6.0.0
pkg-resources==0.0.0
pretrainedmodels==0.7.4
prometheus-client==0.6.0
prompt-toolkit==2.0.9
protobuf==3.7.1
ptyprocess==0.6.0
pycparser==2.19
Pygments==2.3.1
pyparsing==2.3.1
pyrsistent==0.14.11
pysndfx==0.3.6
python-dateutil==2.8.0
python-slugify==3.0.2
pytz==2018.9
pyzmq==18.0.1
qtconsole==4.4.3
requests==2.21.0
resampy==0.2.1
scikit-learn==0.20.3
scipy==1.2.1
Send2Trash==1.5.0
six==1.12.0
SoundFile==0.10.2
tensorboard==1.13.1
tensorboardX==1.6
tensorflow==1.13.1
tensorflow-estimator==1.13.0
termcolor==1.1.0
terminado==0.8.2
testpath==0.4.2
text-unidecode==1.2
torch==1.0.1.post2
torchcontrib==0.0.2
torchvision==0.2.2.post3
tornado==6.0.2
tqdm==4.31.1
traitlets==4.3.2
umap-learn==0.3.8
urllib3==1.24.1
wcwidth==0.1.7
webencodings==0.5.1
```

```
Werkzeug==0.15.2
widgetsnbextension==3.4.2

git+https://github.com/ex4sperans/mag
import os
import gc
import argparse
import json
import math
from functools import partial

import tqdm
import pandas as pd
import numpy as np
import torch
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score

from datasets.sound_dataset import SoundDataset
from networks.classifiers import HierarchicalCNNClassificationMo
del
from ops.folds import train_validation_data
from ops.transforms import (
    Compose, DropFields, LoadAudio,
    AudioFeatures, MapLabels, RenameFields,
    MixUp, SampleSegment, SampleLongAudio)
from ops.utils import load_json, get_class_names_from_classmap,
lwlrap
from ops.padding import make_collate_fn
from networks.classifiers import ResnetBlock

torch.manual_seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(42)

parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--train_df", required=True, type=str,
    help="path to train dataframe"
)
parser.add_argument(
    "--train_data_dir", required=True, type=str,
    help="path to train data"
)
parser.add_argument(
    "--test_data_dir", required=True, type=str,
    help="path to test data"
)
parser.add_argument(
    "--test_df", required=True, type=str,
    help="path to train dataframe"
)
```

```
parser.add_argument(
    "--val_size", required=True, type=float,
    help="size of the validation set"
)
parser.add_argument(
    "--device", type=str, required=True,
    help="whether to train on cuda or cpu",
    choices=("cuda", "cpu")
)
parser.add_argument(
    "--batch_size", type=int, default=64,
    help="minibatch size"
)
parser.add_argument(
    "--epochs", type=int, default=100,
    help="number of epochs"
)
parser.add_argument(
    "--lr", default=0.01, type=float,
    help="starting learning rate"
)
parser.add_argument(
    "--max_samples", type=int,
    help="maximum number of samples to use"
)
parser.add_argument(
    "--features", type=str, required=True,
    help="feature descriptor"
)
parser.add_argument(
    "--max_audio_length", type=int, default=10,
    help="max audio length in seconds. For longer clips are samp
led"
)
parser.add_argument(
    "--batches_to_save", type=int, default=3,
    help="how many batches to save"
)
parser.add_argument(
    "--classmap", required=True, type=str,
    help="path to class map json"
)

args = parser.parse_args()

train_df = pd.read_csv(args.train_df)
test_df = pd.read_csv(args.test_df)

if args.max_samples:
    train_df = train_df.sample(args.max_samples).reset_index(dro
p=True)
    test_df = test_df.sample(args.max_samples).reset_index(drop=
True)

all_train_fnames = [
    os.path.join(args.train_data_dir, fname) for fname in train_
```

```
df.fname.values]
all_test_fnames = [
    os.path.join(args.test_data_dir, fname) for fname in test_df
.fname.values]

fnames = np.concatenate([all_train_fnames, all_test_fnames])
labels = np.concatenate([np.ones(len(train_df)), np.zeros(len(te
st_df))])

train_fnames, val_fnames, train_labels, val_labels = train_test_
split(
    fnames, labels, test_size=args.val_size, shuffle=True)

audio_transform = AudioFeatures(args.features)


class Model(torch.nn.Module):

    def __init__(self):
        super().__init__()

        self.features = torch.nn.Sequential(
            torch.nn.BatchNorm1d(audio_transform.n_features),
            torch.nn.Conv1d(audio_transform.n_features, 32, kern
el_size=1),
            ResnetBlock(32),
            torch.nn.MaxPool1d(kernel_size=2, stride=2),
            torch.nn.BatchNorm1d(32),
            torch.nn.Conv1d(32, 32, kernel_size=3),
            ResnetBlock(32),
            torch.nn.MaxPool1d(kernel_size=2, stride=2),
            torch.nn.BatchNorm1d(32),
            torch.nn.Conv1d(32, 64, kernel_size=3),
            ResnetBlock(64)
        )

        self.pool = torch.nn.AdaptiveMaxPool1d(1)

        self.classifier = torch.nn.Sequential(
            torch.nn.BatchNorm1d(64),
            torch.nn.Conv1d(64, 1, kernel_size=1)
        )

    def forward(self, x):

        x = x.permute(0, 2, 1)
        x = self.features(x)
        x = self.classifier(x)
        x = torch.sigmoid(x)
        nonpooled = x
        x = self.pool(x).squeeze(-1)

        return x.squeeze(1), nonpooled.squeeze(1)


train_loader = torch.utils.data.DataLoader(
```

```
    SoundDataset(
        audio_files=train_fnames,
        labels=train_labels,
        transform=Compose([
            LoadAudio(),
            SampleLongAudio(max_length=args.max_audio_length),
            audio_transform,
            RenameFields({"raw_labels": "labels"}),
            DropFields(("audio", "filename", "sr")),
        ]),
        clean_transform=Compose([
            LoadAudio(),
        ])
    ),
    shuffle=True,
    drop_last=True,
    batch_size=args.batch_size,
    num_workers=4,
    collate_fn=make_collate_fn({"signal": audio_transform.paddin
g_value}),
)

validation_loader = torch.utils.data.DataLoader(
    SoundDataset(
        audio_files=val_fnames,
        labels=val_labels,
        transform=Compose([
            LoadAudio(),
            SampleLongAudio(max_length=args.max_audio_length),
            audio_transform,
            RenameFields({"raw_labels": "labels"}),
            DropFields(("audio", "filename", "sr")),
        ]),
        clean_transform=Compose([
            LoadAudio(),
        ])
    ),
    shuffle=False,
    drop_last=False,
    batch_size=args.batch_size,
    num_workers=4,
    collate_fn=make_collate_fn({"signal": audio_transform.paddin
g_value}),
)

model = Model().to(args.device)
optimizer = torch.optim.Adam(model.parameters(), args.lr)


for epoch in range(args.epochs):

    print(
        "\n" + " " * 10 + "****** Epoch {epoch} ******\n"
        .format(epoch=epoch)
    )
```

```
    model.train()

    with tqdm.tqdm(total=len(train_loader), ncols=80) as pb:

        for sample in train_loader:

            signal, labels = (
                sample["signal"].to(args.device),
                sample["labels"].to(args.device).float()
            )

            probs, nonpooled = model(signal)

            optimizer.zero_grad()
            loss = torch.nn.functional.binary_cross_entropy(prob
s, labels)
            loss.backward()
            optimizer.step()

            pb.update()
            pb.set_description("Loss: {:.4f}".format(loss.item()
))

    model.eval()

    val_probs = []
    val_labels = []

    with torch.no_grad():

        for sample in validation_loader:

            signal, labels = (
                sample["signal"].to(args.device),
                sample["labels"].to(args.device).float()
            )

            probs, nonpooled = model(signal)

            val_probs.extend(probs.data.cpu().numpy())
            val_labels.extend(labels.data.cpu().numpy())

    auc = roc_auc_score(val_labels, val_probs)

    print("\nEpoch: {}, AUC: {}".format(epoch, auc))

model.eval()

# plot probabilities
loader = iter(validation_loader)
directory = "plots/"
os.makedirs(directory, exist_ok=True)

for n in range(args.batches_to_save):

    with torch.no_grad():
```

```
        sample = next(loader)
        signal, labels = (
            sample["signal"].to(args.device),
            sample["labels"].to(args.device).float()
        )

        probs, nonpooled = model(signal)

        nonpooled = nonpooled.data.cpu().numpy()
        signal = signal.data.cpu().numpy()
        labels = labels.data.cpu().numpy()

    for k in range(len(signal)):

        fig = plt.figure(figsize=(20, 7))
        fig.suptitle(str(labels[k]))
        ax = fig.add_subplot(211)
        ax.imshow(np.transpose(signal[k]))
        ax = fig.add_subplot(212)
        ax.plot(nonpooled[k])
        ax.set_ylim(0, 1)
        ax.set_xlim(0, len(nonpooled[k]) - 1)

        fig.savefig(os.path.join(directory, "plot_{}_{}.png".for
mat(n, k)))
        plt.close()

# compute average scores for classes

class_map = load_json(args.classmap)

names_with_labels = [
    fname for fname in val_fnames if fname in all_train_fnames]
labels = pd.DataFrame({
    "fname": [os.path.basename(fname) for fname in names_with_la
bels]}).merge(
        train_df, on="fname", how="left").labels.values

loader = torch.utils.data.DataLoader(
    SoundDataset(
        audio_files=names_with_labels,
        labels=[item.split(",") for item in labels],
        transform=Compose([
            LoadAudio(),
            MapLabels(class_map),
            SampleLongAudio(max_length=args.max_audio_length),
            audio_transform,
            DropFields(("audio", "filename", "sr")),
        ])
    ),
    shuffle=False,
    drop_last=False,
    batch_size=args.batch_size,
    num_workers=4,
    collate_fn=make_collate_fn({"signal": audio_transform.paddin
```

```
g_value}),
)

all_probs = []
all_labels = []

with torch.no_grad():

    for sample in loader:

        signal, labels = (
            sample["signal"].to(args.device),
            sample["labels"].to(args.device).float()
        )

        probs, nonpooled = model(signal)

        all_probs.extend(probs.data.cpu().numpy())
        all_labels.extend(labels.data.cpu().numpy())

all_probs = np.array(all_probs)
all_labels = np.array(all_labels)

scores = all_labels * np.expand_dims(all_probs, -1)
mean_scores = scores.sum(axis=0) / all_labels.sum(axis=0)

classnames = get_class_names_from_classmap(class_map)

pd.options.display.max_rows = 100

print()
print(pd.DataFrame({"classname": classnames, "scores": mean_scor
es}))

======
import json
import argparse

import pandas as pd


parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--train_df", required=True, type=str,
    help="path to train dataframe"
)
parser.add_argument(
    "--output_file", type=str, required=True,
    help="where to save classmap"
)

args = parser.parse_args()
```

```
df = pd.read_csv(args.train_df)

all_labels = set()
for item in df.labels:
    all_labels.update(item.split(","))


classmap = dict((v, k) for k, v in enumerate(sorted(all_labels))
)

with open(args.output_file, "w") as file:
    json.dump(classmap, file, indent=4, sort_keys=True)======
import os
import gc
import argparse
import json
import math
from functools import partial

import pandas as pd
import numpy as np
import torch
from mag.experiment import Experiment
from mag.utils import green, bold
import mag

from datasets.sound_dataset import SoundDataset
from networks.classifiers import TwoDimensionalCNNClassification
Model
from ops.folds import train_validation_data_stratified
from ops.transforms import (
    Compose, DropFields, LoadAudio,
    AudioFeatures, MapLabels, RenameFields,
    MixUp, SampleSegment, SampleLongAudio,
    AudioAugmentation, FlipAudio, ShuffleAudio)
from ops.utils import load_json, get_class_names_from_classmap,
lwlrap
from ops.padding import make_collate_fn

mag.use_custom_separator("-")

parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--experiment", type=str, required=True,
    help="path to an experiment"
)
parser.add_argument(
    "--train_df", required=True, type=str,
    help="path to train dataframe"
)
parser.add_argument(
    "--train_data_dir", required=True, type=str,
```

```
    help="path to train data"
)
parser.add_argument(
    "--noisy_train_df", type=str,
    help="path to noisy train dataframe (optional)"
)
parser.add_argument(
    "--noisy_train_data_dir", type=str,
    help="path to noisy train data (optional)"
)
parser.add_argument(
    "--classmap", required=True, type=str,
    help="path to class map json"
)
parser.add_argument(
    "--batch_size", type=int, default=32,
    help="batch size used for prediction"
)
parser.add_argument(
    "--max_audio_length", type=int, default=10,
    help="max audio length in seconds. For longer clips are samp
led"
)
parser.add_argument(
    "--n_tta", type=int, default=1,
    help="number of tta"
)
parser.add_argument(
    "--device", type=str, required=True,
    help="whether to train on cuda or cpu",
    choices=("cuda", "cpu")
)
parser.add_argument(
    "--num_workers", type=int, default=4,
    help="number of workers for data loader",
)

args = parser.parse_args()

class_map = load_json(args.classmap)

train_df = pd.read_csv(args.train_df)

with Experiment(resume_from=args.experiment) as experiment:

    config = experiment.config

    audio_transform = AudioFeatures(config.data.features)

    splits = list(train_validation_data_stratified(
            train_df.fname, train_df.labels, class_map,
            config.data._n_folds, config.data._kfold_seed))

    all_labels = np.zeros(
        shape=(len(train_df), len(class_map)), dtype=np.float32)
    all_predictions = np.zeros(
```

```
        shape=(len(train_df), len(class_map)), dtype=np.float32)

    for fold in range(config.data._n_folds):

        print("\n\n   -----  Fold {}\n".format(fold))

        train, valid = splits[fold]

        loader_kwargs = (
            {"num_workers": args.num_workers, "pin_memory": True
}
            if torch.cuda.is_available() else {})

        valid_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[valid]],
                labels=[item.split(",") for item in train_df.lab
els.values[valid]],
                transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                    SampleLongAudio(args.max_audio_length),
                    ShuffleAudio(chunks_range=(12, 20), p=1.0),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ]),
                clean_transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                ]),


            ),
            shuffle=False,
            batch_size=args.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        model = TwoDimensionalCNNClassificationModel(
                experiment, device=args.device)
        model.load_best_model(fold)
        model.eval()

        val_preds = model.predict(valid_loader, n_tta=args.n_tta
)
        val_labels = np.array([item["labels"] for item in valid_
loader.dataset])

        all_labels[valid] = val_labels
        all_predictions[valid] = val_preds

        metric = lwlrap(val_labels, val_preds)
```

```
        print("Fold metric:", metric)

    metric = lwlrap(all_labels, all_predictions)

    print("\nOverall metric:", green(bold(metric)))


======
import os
import gc
import argparse
import json
import math
from functools import partial

import pandas as pd
import numpy as np
import torch
from mag.experiment import Experiment
import mag
from sklearn.model_selection import train_test_split

from datasets.sound_dataset import SoundDataset
from networks.classifiers import HierarchicalCNNClassificationMo
del
from ops.folds import train_validation_data
from ops.transforms import (
    Compose, DropFields, LoadAudio,
    STFT, MapLabels, RenameFields, MixUp)
from ops.utils import load_json, get_class_names_from_classmap,
lwlrap
from ops.padding import make_collate_fn

torch.manual_seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(42)

mag.use_custom_separator("-")

parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--train_df", required=True, type=str,
    help="path to train dataframe"
)
parser.add_argument(
    "--train_data_dir", required=True, type=str,
    help="path to train data"
)
parser.add_argument(
    "--test_data_dir", required=True, type=str,
    help="path to test data"
)
parser.add_argument(
```

```
    "--sample_submission", required=True, type=str,
    help="path sample submission"
)
parser.add_argument(
    "--pretrained_model", required=True, type=str,
    help="path to old experiment"
)
parser.add_argument(
    "--pretrained_fold", required=True, type=int,
    help="pretrained fold"
)
parser.add_argument(
    "--classmap", required=True, type=str,
    help="path to class map json"
)
parser.add_argument(
    "--log_interval", default=10, type=int,
    help="how frequently to log batch metrics"
    "in terms of processed batches"
)
parser.add_argument(
    "--batch_size", type=int, default=64,
    help="minibatch size"
)
parser.add_argument(
    "--lr", default=0.01, type=float,
    help="starting learning rate"
)
parser.add_argument(
    "--max_samples", type=int,
    help="maximum number of samples to use"
)
parser.add_argument(
    "--holdout_size", type=float, default=0.0,
    help="size of holdout set"
)
parser.add_argument(
    "--epochs", default=100, type=int,
    help="number of epochs to train"
)
parser.add_argument(
    "--scheduler", type=str, default="steplr_1_0.5",
    help="scheduler type",
)
parser.add_argument(
    "--accumulation_steps", type=int, default=1,
    help="number of gradient accumulation steps",
)
parser.add_argument(
    "--save_every", type=int, default=1,
    help="how frequently to save a model",
)
parser.add_argument(
    "--device", type=str, required=True,
    help="whether to train on cuda or cpu",
    choices=("cuda", "cpu")
```

```
)
parser.add_argument(
    "--weight_decay", type=float, default=1e-5,
    help="weight decay"
)
parser.add_argument(
    "--dropout", type=float, default=0.0,
    help="internal dropout"
)
parser.add_argument(
    "--output_dropout", type=float, default=0.0,
    help="output dropout"
)
parser.add_argument(
    "--p_mixup", type=float, default=0.0,
    help="probability of the mixup augmentation"
)
parser.add_argument(
    "--switch_off_augmentations_on", type=int, default=20,
    help="on which epoch to remove augmentations"
)
parser.add_argument(
    "--optimizer", type=str, required=True,
    help="which optimizer to use",
    choices=("adam", "momentum")
)
parser.add_argument(
    "--folds", type=int, required=True, nargs="+",
    help="which folds to use"
)
parser.add_argument(
    "--n_folds", type=int, default=4,
    help="number of folds"
)
parser.add_argument(
    "--kfold_seed", type=int, default=42,
    help="kfold seed"
)
parser.add_argument(
    "--num_workers", type=int, default=4,
    help="number of workers for data loader",
)
parser.add_argument(
    "--label", type=str, default="finetuned_hierarchical_cnn_cla
ssifier",
    help="optional label",
)
args = parser.parse_args()

class_map = load_json(args.classmap)

pretrained = Experiment(resume_from=args.pretrained_model)

with Experiment({
    "network": {
        "num_conv_blocks": pretrained.config.network.num_conv_bl
```

```
ocks,
        "start_deep_supervision_on": pretrained.config.network.s
tart_deep_supervision_on,
        "conv_base_depth": pretrained.config.network.conv_base_d
epth,
        "growth_rate": pretrained.config.network.growth_rate,
        "dropout": args.dropout,
        "output_dropout": args.output_dropout,
    },
    "data": {
        "_n_folds": args.n_folds,
        "_kfold_seed": args.kfold_seed,
        "n_fft": pretrained.config.data.n_fft,
        "hop_size": pretrained.config.data.hop_size,
        "_input_dim": pretrained.config.data.n_fft // 2 + 1,
        "_n_classes": len(class_map),
        "_holdout_size": args.holdout_size,
        "p_mixup": args.p_mixup
    },
    "train": {
        "accumulation_steps": args.accumulation_steps,
        "batch_size": args.batch_size,
        "learning_rate": args.lr,
        "scheduler": args.scheduler,
        "optimizer": args.optimizer,
        "epochs": args.epochs,
        "_save_every": args.save_every,
        "weight_decay": args.weight_decay,
        "switch_off_augmentations_on": args.switch_off_augmentat
ions_on,
        "_pretrained_experiment": args.pretrained_model,
        "_pretrained_fold": args.pretrained_fold,
    },
    "label": args.label
}) as experiment:

    config = experiment.config
    print()
    print("     ////// CONFIG //////")
    print(experiment.config)

    train_df = pd.read_csv(args.train_df)
    test_df = pd.read_csv(args.sample_submission)

    if args.max_samples:
        train_df = train_df.sample(args.max_samples).reset_index
(drop=True)
        test_df = test_df.sample(
            min(args.max_samples, len(test_df))).reset_index(dro
p=True)

    if args.holdout_size:
        keep, holdout = train_test_split(
            np.arange(len(train_df)), test_size=args.holdout_siz
e,
            random_state=args.kfold_seed)
```

```
        holdout_df = train_df.iloc[holdout].reset_index(drop=Tru
e)
        train_df = train_df.iloc[keep].reset_index(drop=True)

    splits = list(train_validation_data(
        train_df.fname, train_df.labels,
        config.data._n_folds, config.data._kfold_seed))

    for fold in args.folds:

        print("\n\n   -----  Fold {}\n".format(fold))

        train, valid = splits[fold]

        loader_kwargs = (
            {"num_workers": args.num_workers, "pin_memory": True
}
            if torch.cuda.is_available() else {})

        experiment.register_directory("checkpoints")
        experiment.register_directory("predictions")

        train_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[train]],
                labels=[item.split(",") for item in train_df.lab
els.values[train]],
                transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                    MixUp(p=args.p_mixup),
                    STFT(n_fft=config.data.n_fft, hop_size=confi
g.data.hop_size),
                    DropFields(("audio", "filename", "sr")),
                    RenameFields({"stft": "signal"})
                ]),
                clean_transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                ])
            ),
            shuffle=True,
            drop_last=True,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": math.log(STFT.
eps)}),
            **loader_kwargs
        )

        valid_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[valid]],
```

```
                labels=[item.split(",") for item in train_df.lab
els.values[valid]],
                transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                    STFT(n_fft=config.data.n_fft, hop_size=confi
g.data.hop_size),
                    DropFields(("audio", "filename", "sr")),
                    RenameFields({"stft": "signal"})
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": math.log(STFT.
eps)}),
            **loader_kwargs
        )

        model = HierarchicalCNNClassificationModel(experiment, d
evice=args.device)
        # load pretrained model
        model.load_state_dict(
            torch.load(
                os.path.join(
                    pretrained.checkpoints,
                    "fold_{}".format(args.pretrained_fold),
                    "best_model.pth"
                )
            )
        )

        scores = model.fit_validate(
            train_loader, valid_loader,
            epochs=experiment.config.train.epochs, fold=fold,
            log_interval=args.log_interval
        )

        best_metric = max(scores)
        experiment.register_result("fold{}.metric".format(fold),
 best_metric)

        torch.save(
            model.state_dict(),
            os.path.join(
                experiment.checkpoints,
                "fold_{}".format(fold),
                "final_model.pth")
        )

        # predictions
        model.load_best_model(fold)

        # validation

        val_preds = model.predict(valid_loader)
        val_predictions_df = pd.DataFrame(
```

```
            val_preds, columns=get_class_names_from_classmap(cla
ss_map))
        val_predictions_df["fname"] = train_df.fname[valid].valu
es
        val_predictions_df.to_csv(
            os.path.join(
                experiment.predictions,
                "val_preds_fold_{}.csv".format(fold)
            ),
            index=False
        )
        del val_predictions_df

        # test
        test_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.test_data_dir, fname)
                    for fname in test_df.fname.values],
                transform=Compose([
                    LoadAudio(),
                    STFT(n_fft=config.data.n_fft, hop_size=confi
g.data.hop_size),
                    DropFields(("audio", "filename", "sr")),
                    RenameFields({"stft": "signal"})
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": math.log(STFT.
eps)}),
            **loader_kwargs
        )

        test_preds = model.predict(test_loader)
        test_predictions_df = pd.DataFrame(
            test_preds, columns=get_class_names_from_classmap(cl
ass_map))
        test_predictions_df["fname"] = test_df.fname
        test_predictions_df.to_csv(
            os.path.join(
                experiment.predictions,
                "test_preds_fold_{}.csv".format(fold)
            ),
            index=False
        )
        del test_predictions_df

        # holdout
        if args.holdout_size:
            holdout_loader = torch.utils.data.DataLoader(
                SoundDataset(
                    audio_files=[
                        os.path.join(args.train_data_dir, fname)
                        for fname in holdout_df.fname.values],
                    labels=[item.split(",") for item in holdout_
```

```
df.labels.values],
                    transform=Compose([
                        LoadAudio(),
                        MapLabels(class_map),
                        STFT(n_fft=config.data.n_fft, hop_size=c
onfig.data.hop_size),
                        DropFields(("audio", "filename", "sr")),
                        RenameFields({"stft": "signal"})
                    ])
                ),
                shuffle=False,
                batch_size=config.train.batch_size,
                collate_fn=make_collate_fn({"signal": math.log(S
TFT.eps)}),
                **loader_kwargs
            )

            holdout_metric = model.evaluate(holdout_loader)
            experiment.register_result(
                "fold{}.holdout_metric".format(fold), holdout_me
tric)

            print("\nHoldout metric: {:.4f}".format(holdout_metr
ic))

        if args.device == "cuda":
            torch.cuda.empty_cache()

    # global metric

    if all(
        "fold{}".format(k) in experiment.results.to_dict()
        for k in range(config.data._n_folds)):

        val_df_files = [
            os.path.join(
                experiment.predictions,
                "val_preds_fold_{}.csv".format(fold)
            )
            for fold in range(config.data._n_folds)
        ]

        val_predictions_df = pd.concat([
            pd.read_csv(file) for file in val_df_files]).reset_i
ndex(drop=True)

        labels = np.asarray([
            item["labels"] for item in SoundDataset(
                audio_files=train_df.fname.tolist(),
                labels=[item.split(",") for item in train_df.lab
els.values],
                transform=MapLabels(class_map)
            )
        ])

        val_labels_df = pd.DataFrame(
```

```
            labels, columns=get_class_names_from_classmap(class_
map))
        val_labels_df["fname"] = train_df.fname

        assert set(val_predictions_df.fname) == set(val_labels_d
f.fname)

        val_predictions_df.sort_values(by="fname", inplace=True)
        val_labels_df.sort_values(by="fname", inplace=True)

        metric = lwlrap(
            val_labels_df.drop("fname", axis=1).values,
            val_predictions_df.drop("fname", axis=1).values
        )

        experiment.register_result("metric", metric)

    # submission

    test_df_files = [
        os.path.join(
            experiment.predictions,
            "test_preds_fold_{}.csv".format(fold)
        )
        for fold in range(config.data._n_folds)
    ]

    if all(os.path.isfile for file in test_df_files):
        test_dfs = [pd.read_csv(file) for file in test_df_files]
        submission_df = pd.DataFrame({"fname": test_dfs[0].fname
.values})
        for c in get_class_names_from_classmap(class_map):
            submission_df[c] = np.mean([d[c].values for d in tes
t_dfs], axis=0)
        submission_df.to_csv(
            os.path.join(experiment.predictions, "submission.csv
"), index=False)======
```

```
      other entities that control, are controlled by, or are und
er common
      control with that entity. For the purposes of this definit
ion,
      "control" means (i) the power, direct or indirect, to caus
e the
      direction or management of such entity, whether by contrac
t or
      otherwise, or (ii) ownership of fifty percent (50%) or mor
e of the
      outstanding shares, or (iii) beneficial ownership of such
entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making mod
ifications,
      including but not limited to software source code, documen
tation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanica
l
      transformation or translation of a Source form, including
but
      not limited to compiled object code, generated documentati
on,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Sourc
e or
      Object form, made available under the License, as indicate
d by a
      copyright notice that is included in or attached to the wo
rk
      (an example is provided in the Appendix below).

      "Derivative Works" shall mean any work, whether in Source
or Object
      form, that is based on (or derived from) the Work and for
which the
      editorial revisions, annotations, elaborations, or other m
odifications
      represent, as a whole, an original work of authorship. For
 the purposes
      of this License, Derivative Works shall not include works
that remain
      separable from, or merely link (or bind by name) to the in
terfaces of,
      the Work and Derivative Works thereof.

      "Contribution" shall mean any work of authorship, includin
g
      the original version of the Work and any modifications or
additions
```

to that Work or Derivative Works thereof, that is intentio
nally
submitted to Licensor for inclusion in the Work by the cop
yright owner
or by an individual or Legal Entity authorized to submit o
n behalf of
the copyright owner. For the purposes of this definition,
"submitted"
means any form of electronic, verbal, or written communica
tion sent
to the Licensor or its representatives, including but not
limited to
communication on electronic mailing lists, source code con
trol systems,
and issue tracking systems that are managed by, or on beha
lf of, the
Licensor for the purpose of discussing and improving the W
ork, but
excluding communication that is conspicuously marked or ot
herwise
designated in writing by the copyright owner as "Not a Con
tribution."

"Contributor" shall mean Licensor and any individual or Le
gal Entity
on behalf of whom a Contribution has been received by Lice
nsor and
subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and condi
tions of
this License, each Contributor hereby grants to You a perp
etual,
worldwide, non-exclusive, no-charge, royalty-free, irrevoc
able
copyright license to reproduce, prepare Derivative Works o
f,
publicly display, publicly perform, sublicense, and distri
bute the
Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditio
ns of
this License, each Contributor hereby grants to You a perp
etual,
worldwide, non-exclusive, no-charge, royalty-free, irrevoc
able
(except as stated in this section) patent license to make,
have made,
use, offer to sell, sell, import, and otherwise transfer t
he Work,
where such license applies only to those patent claims lic
ensable
by such Contributor that are necessarily infringed by thei
r
Contribution(s) alone or by combination of their Contribut

```
ion(s)
      with the Work to which such Contribution(s) was submitted.
 If You
      institute patent litigation against any entity (including
a
      cross-claim or counterclaim in a lawsuit) alleging that th
e Work
      or a Contribution incorporated within the Work constitutes
 direct
      or contributory patent infringement, then any patent licen
ses
      granted to You under this License for that Work shall term
inate
      as of the date such litigation is filed.

   4. Redistribution. You may reproduce and distribute copies of
 the
      Work or Derivative Works thereof in any medium, with or wi
thout
      modifications, and in Source or Object form, provided that
 You
      meet the following conditions:

      (a) You must give any other recipients of the Work or
          Derivative Works a copy of this License; and

      (b) You must cause any modified files to carry prominent n
otices
          stating that You changed the files; and

      (c) You must retain, in the Source form of any Derivative
Works
          that You distribute, all copyright, patent, trademark,
 and
          attribution notices from the Source form of the Work,
          excluding those notices that do not pertain to any par
t of
          the Derivative Works; and

      (d) If the Work includes a "NOTICE" text file as part of i
ts
          distribution, then any Derivative Works that You distr
ibute must
          include a readable copy of the attribution notices con
tained
          within such NOTICE file, excluding those notices that
do not
          pertain to any part of the Derivative Works, in at lea
st one
          of the following places: within a NOTICE text file dis
tributed
          as part of the Derivative Works; within the Source for
m or
          documentation, if provided along with the Derivative W
orks; or,
          within a display generated by the Derivative Works, if
```

```
 and
          wherever such third-party notices normally appear. The
 contents
          of the NOTICE file are for informational purposes only
 and
          do not modify the License. You may add Your own attrib
ution
          notices within Derivative Works that You distribute, a
longside
          or as an addendum to the NOTICE text from the Work, pr
ovided
          that such additional attribution notices cannot be con
strued
          as modifying the License.

      You may add Your own copyright statement to Your modificat
ions and
      may provide additional or different license terms and cond
itions
      for use, reproduction, or distribution of Your modificatio
ns, or
      for any such Derivative Works as a whole, provided Your us
e,
      reproduction, and distribution of the Work otherwise compl
ies with
      the conditions stated in this License.

   5. Submission of Contributions. Unless You explicitly state o
therwise,
      any Contribution intentionally submitted for inclusion in
the Work
      by You to the Licensor shall be under the terms and condit
ions of
      this License, without any additional terms or conditions.
      Notwithstanding the above, nothing herein shall supersede
or modify
      the terms of any separate license agreement you may have e
xecuted
      with Licensor regarding such Contributions.

   6. Trademarks. This License does not grant permission to use
the trade
      names, trademarks, service marks, or product names of the
Licensor,
      except as required for reasonable and customary use in des
cribing the
      origin of the Work and reproducing the content of the NOTI
CE file.

   7. Disclaimer of Warranty. Unless required by applicable law
or
      agreed to in writing, Licensor provides the Work (and each
      Contributor provides its Contributions) on an "AS IS" BASI
S,
      WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either expre
ss or
```

implied, including, without limitation, any warranties or conditions
of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
PARTICULAR PURPOSE. You are solely responsible for determining the
appropriateness of using or redistributing the Work and assume any
risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
whether in tort (including negligence), contract, or otherwise,
unless required by applicable law (such as deliberate and grossly
negligent acts) or agreed to in writing, shall any Contributor be
liable to You for damages, including any direct, indirect, special,
incidental, or consequential damages of any character arising as a
result of this License or out of the use or inability to use the
Work (including but not limited to damages for loss of goodwill,
work stoppage, computer failure or malfunction, or any and all
other commercial damages or losses), even if such Contributor
has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
the Work or Derivative Works thereof, You may choose to offer,
and charge a fee for, acceptance of support, warranty, indemnity,
or other liability obligations and/or rights consistent with this
License. However, in accepting such obligations, You may act only
on Your own behalf and on Your sole responsibility, not on behalf
of any other Contributor, and only if You agree to indemnify,
defend, and hold each Contributor harmless for any liability
incurred by, or claims asserted against, such Contributor by reason
of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

```
======
import argparse
import glob
from pathlib import Path

import pandas as pd
import numpy as np
import scipy.optimize
from scipy.stats import rankdata
from mag.utils import blue, green, bold

from ops.utils import lwlrap


parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--experiments", type=str, required=True, nargs="+",
```

```
    help="experiments to blend"
)
parser.add_argument(
    "--train_df", type=str, required=True,
    help="path to train df"
)
parser.add_argument(
    "--rankdata", action="store_true", default=False,
    help="whether to use ranks instead of raw scores"
)
parser.add_argument(
    "--output_df", type=str, required=True,
    help="where to save test submission"
)

args = parser.parse_args()


n = len(args.experiments)


def load_predictions(experiment):

    prediction_files = (
        "experiments" / Path(experiment) / "predictions").glob("
val_preds*")
    dfs = [pd.read_csv(f) for f in prediction_files]
    df = pd.concat(dfs).reset_index(drop=True)
    df = df.sort_values(by="fname")
    df = df[sorted(df.columns.tolist())]
    return df


def to_ranks(values):
    return np.array([rankdata(r) for r in values])


predictions = [load_predictions(exp) for exp in args.experiments
]
class_cols = predictions[0].columns.drop("fname")
prediction_values = [p[class_cols].values for p in predictions]
if args.rankdata:
    prediction_values = [to_ranks(p) for p in prediction_values]

train_df = pd.read_csv(args.train_df)


def make_actual_labels(train_df):

    classname_to_idx = dict((c, i) for i, c in enumerate(class_c
ols))
    actual_labels = np.zeros((len(train_df), len(class_cols)), d
type=np.float32)
    for k in range(train_df.labels.values.size):
        for label in str(train_df.labels.values[k]).split(","):
            actual_labels[k, classname_to_idx[label]] = 1
```

```
    return actual_labels


actual_labels = make_actual_labels(train_df)


def constraints():
    A = np.ones(n)
    yield scipy.optimize.LinearConstraint(A=A, lb=0.01, ub=0.99)
    for k in range(n):
        A = np.zeros(n)
        A[k] = 1
        yield scipy.optimize.LinearConstraint(A=A, lb=0, ub=1)


def initial():
    return np.ones(n) / n

def target(alphas, *args):
    prediction = np.sum([a * p for a, p in zip(alphas, predictio
n_values)], axis=0)
    return -lwlrap(actual_labels, prediction)


alphas = scipy.optimize.minimize(
    target,
    initial(),
    constraints=list(constraints()),
    method="COBYLA").x

print()
for experiment, alpha in zip(args.experiments, alphas):
    print("{}: {}".format(green(bold(experiment)), blue(bold(alp
ha))))

print()
print("Final lwlrap:", bold(green(-target(alphas))))


def load_test_predictions(experiment):

    prediction_files = (
        "experiments" / Path(experiment) / "predictions").glob("
test_preds*")
    dfs = [pd.read_csv(f) for f in prediction_files]
    dfs = [df.sort_values(by="fname") for df in dfs]
    return dfs


test_preds = []

for alpha, exp in zip(alphas, args.experiments):
    experiment_test_predictions = load_test_predictions(experime
nt)
    for p in experiment_test_predictions:
```

```
        if args.rankdata:
            test_preds.append(to_ranks(p[class_cols].values) * a
lpha)
        else:
            test_preds.append(p[class_cols].values * alpha)

test_preds = np.sum(test_preds, 0)

sub = pd.DataFrame(test_preds, columns=class_cols)
sub["fname"] = p.fname

sub.to_csv(args.output_df, index=False)======
import os
import gc
import argparse
import json
import math
from functools import partial

import pandas as pd
import numpy as np
import torch
from mag.experiment import Experiment
from mag.utils import green, bold
import mag

from datasets.sound_dataset import SoundDataset
from networks.classifiers import TwoDimensionalCNNClassification
Model
from ops.folds import train_validation_data_stratified
from ops.transforms import (
    Compose, DropFields, LoadAudio,
    AudioFeatures, MapLabels, RenameFields,
    MixUp, SampleSegment, SampleLongAudio,
    AudioAugmentation, FlipAudio, ShuffleAudio)
from ops.utils import load_json, get_class_names_from_classmap,
lwlrap
from ops.padding import make_collate_fn

mag.use_custom_separator("-")

parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--experiment", type=str, required=True,
    help="path to an experiment"
)
parser.add_argument(
    "--test_df", required=True, type=str,
    help="path to test dataframe"
)
parser.add_argument(
    "--output_df", required=True, type=str,
    help="where to save resulting dataframe"
```

```
)
parser.add_argument(
    "--test_data_dir", required=True, type=str,
    help="path to test data directory"
)
parser.add_argument(
    "--classmap", required=True, type=str,
    help="path to class map json"
)
parser.add_argument(
    "--batch_size", type=int, default=32,
    help="batch size used for prediction"
)
parser.add_argument(
    "--device", type=str, required=True,
    help="whether to train on cuda or cpu",
    choices=("cuda", "cpu")
)
parser.add_argument(
    "--num_workers", type=int, default=4,
    help="number of workers for data loader",
)

args = parser.parse_args()

class_map = load_json(args.classmap)

test_df = pd.read_csv(args.test_df)

with Experiment(resume_from=args.experiment) as experiment:

    config = experiment.config

    audio_transform = AudioFeatures(config.data.features)

    all_predictions = np.zeros(
        shape=(len(test_df), len(class_map)), dtype=np.float32)

    for fold in range(config.data._n_folds):

        print("\n\n   -----  Fold {}\n".format(fold))

        loader_kwargs = (
            {"num_workers": args.num_workers, "pin_memory": True
}
            if torch.cuda.is_available() else {})

        test_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.test_data_dir, fname)
                    for fname in test_df.fname.values],
                labels=None,
                transform=Compose([
                    LoadAudio(),
                    audio_transform,
```

```
                    DropFields(("audio", "filename", "sr")),
                ]),
                clean_transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                ]),
            ),
            shuffle=False,
            batch_size=args.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        model = TwoDimensionalCNNClassificationModel(
                experiment, device=args.device)
        model.load_best_model(fold)
        model.eval()

        val_preds = model.predict(test_loader)

        all_predictions += val_preds / config.data._n_folds


result = pd.DataFrame(
    all_predictions, columns=get_class_names_from_classmap(class
_map))
result["fname"] = test_df.fname

result.to_csv(args.output_df, index=False)======
import os
import gc
import argparse
import json
import math
from functools import partial

from scipy.sparse import csr_matrix
from scipy.stats import rankdata

import pandas as pd
import numpy as np

parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--noisy_df", required=True, type=str,
    help="path to noisy dataframe"
)
parser.add_argument(
    "--noisy_predictions_df", required=True, type=str,
    help="path to noisy predictions"
)
parser.add_argument(
```

```
    "--output_df", required=True, type=str,
    help="where to save relabeled dataframe"
)
parser.add_argument(
    "--mode", required=True, type=str,
    help="relabeling strategy"
)

args = parser.parse_args()

noisy_df = pd.read_csv(args.noisy_df)
noisy_predictions_df = pd.read_csv(args.noisy_predictions_df)

noisy_df.sort_values(by="fname", inplace=True)
noisy_predictions_df.sort_values(by="fname", inplace=True)

mode, *params = args.mode.split("_")

class_cols = noisy_predictions_df.columns.drop("fname").values
classname_to_idx = dict((c, i) for i, c in enumerate(class_cols)
)
idx_to_classname = dict(enumerate(class_cols))
noisy_labels = np.zeros((len(noisy_df), len(class_cols)), dtype=
np.float32)
for k in range(noisy_df.labels.values.size):
    for label in str(noisy_df.labels.values[k]).split(","):
        noisy_labels[k, classname_to_idx[label]] = 1


def binary_to_labels(binary):
    labels = []
    for row in binary:
        labels.append(",".join(idx_to_classname[k] for k in nonz
ero(row)))

    return labels


def find_threshold(probs, expected_classes_per_sample):

        thresholds = np.linspace(0, 1, 10000)
        classes_per_sample = np.zeros_like(thresholds)

        for k in range(thresholds.size):
            c = (probs > thresholds[k]).sum(-1).mean()
            classes_per_sample[k] = c

        k = np.argmin(np.abs(classes_per_sample - expected_class
es_per_sample))

        return thresholds[k]

def nonzero(x):
    return np.nonzero(x)[0]
```

```
def merge_labels(first, second):

    merged = []
    for f, s in zip(first, second):
        m = set(f.split(",")) | set(s.split(","))
        if "" in m:
            m.remove("")
        merged.append(",".join(m))

    return merged


def score_samples(y_true, y_score):
    scores = []

    y_true = csr_matrix(y_true)
    y_score = -y_score

    n_samples, n_labels = y_true.shape

    for i, (start, stop) in enumerate(zip(y_true.indptr, y_true.
indptr[1:])):
        relevant = y_true.indices[start:stop]

        if (relevant.size == 0 or relevant.size == n_labels):
            # If all labels are relevant or unrelevant, the scor
e is also
            # equal to 1. The label ranking has no meaning.
            aux = 1.
        else:
            scores_i = y_score[i]
            rank = rankdata(scores_i, 'max')[relevant]
            L = rankdata(scores_i[relevant], 'max')
            aux = (L / rank).mean()

        scores.append(aux)

    return np.array(scores)


if mode == "fullmatch":

    expected_classes_per_sample, = params
    expected_classes_per_sample = float(expected_classes_per_sam
ple)

    probs = noisy_predictions_df[class_cols].values
    threshold = find_threshold(probs, expected_classes_per_sampl
e)
    binary = probs > threshold

    match = (binary == noisy_labels).all(-1)

    relabeled = noisy_df[match]

elif mode == "relabelall":
```

```
    expected_classes_per_sample, = params
    expected_classes_per_sample = float(expected_classes_per_sam
ple)

    probs = noisy_predictions_df[class_cols].values
    threshold = find_threshold(probs, expected_classes_per_sampl
e)
    binary = probs > threshold

    new_labels = binary_to_labels(binary)

    noisy_df.labels = new_labels
    noisy_df = noisy_df[noisy_df.labels != ""]

    relabeled = noisy_df

elif mode == "relabelall-replacenan":

    expected_classes_per_sample, = params
    expected_classes_per_sample = float(expected_classes_per_sam
ple)

    probs = noisy_predictions_df[class_cols].values
    threshold = find_threshold(probs, expected_classes_per_sampl
e)
    binary = probs > threshold

    new_labels = pd.Series(binary_to_labels(binary))
    where_non_empty = (new_labels != "")
    noisy_df = noisy_df[where_non_empty]
    noisy_df.labels = new_labels[where_non_empty]

    relabeled = noisy_df

elif mode == "relabelall-merge":

    expected_classes_per_sample, = params
    expected_classes_per_sample = float(expected_classes_per_sam
ple)

    probs = noisy_predictions_df[class_cols].values
    threshold = find_threshold(probs, expected_classes_per_sampl
e)
    binary = probs > threshold

    new_labels = binary_to_labels(binary)
    noisy_df.labels = merge_labels(noisy_df.labels.values, new_l
abels)

    relabeled = noisy_df

elif mode == "scoring":

    topk, = params
    topk = int(topk)
```

```
    probs = noisy_predictions_df[class_cols].values
    scores = score_samples(noisy_labels, probs)

    selection = np.argsort(-scores)[:topk]

    relabeled = noisy_df.iloc[selection]

print("Relabeled df shape:", relabeled.shape)

relabeled.to_csv(args.output_df, index=False)
======
import os
import gc
import argparse
import json
import math
from functools import partial

import pandas as pd
import numpy as np
import torch
from mag.experiment import Experiment
import mag
from sklearn.model_selection import train_test_split

from datasets.sound_dataset import SoundDataset
from networks.classifiers import TwoDimensionalCNNClassification
Model
from ops.folds import train_validation_data, train_validation_da
ta_stratified
from ops.transforms import (
    Compose, DropFields, LoadAudio,
    AudioFeatures, MapLabels, RenameFields,
    MixUp, SampleSegment, SampleLongAudio,
    AudioAugmentation, ShuffleAudio, CutOut, Identity)
from ops.utils import load_json, get_class_names_from_classmap,
lwlrap
from ops.padding import make_collate_fn

torch.manual_seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(42)

mag.use_custom_separator("-")

parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--train_df", required=True, type=str,
    help="path to train dataframe"
)
parser.add_argument(
    "--train_data_dir", required=True, type=str,
```

```
    help="path to train data"
)
parser.add_argument(
    "--noisy_train_df", type=str,
    help="path to noisy train dataframe (optional)"
)
parser.add_argument(
    "--noisy_train_data_dir", type=str,
    help="path to noisy train data (optional)"
)
parser.add_argument(
    "--share_noisy", action="store_true", default=False,
    help="whether to share noisy files across folds"
)
parser.add_argument(
    "--resume", action="store_true", default=False,
    help="allow resuming even if experiment exists"
)
parser.add_argument(
    "--test_data_dir", required=True, type=str,
    help="path to test data"
)
parser.add_argument(
    "--sample_submission", required=True, type=str,
    help="path sample submission"
)
parser.add_argument(
    "--classmap", required=True, type=str,
    help="path to class map json"
)
parser.add_argument(
    "--log_interval", default=10, type=int,
    help="how frequently to log batch metrics"
    "in terms of processed batches"
)
parser.add_argument(
    "--batch_size", type=int, default=64,
    help="minibatch size"
)
parser.add_argument(
    "--max_audio_length", type=int, default=10,
    help="max audio length in seconds. For longer clips are samp
led"
)
parser.add_argument(
    "--lr", default=0.01, type=float,
    help="starting learning rate"
)
parser.add_argument(
    "--max_samples", type=int,
    help="maximum number of samples to use"
)
parser.add_argument(
    "--holdout_size", type=float, default=0.0,
    help="size of holdout set"
)
```

```
parser.add_argument(
    "--epochs", default=100, type=int,
    help="number of epochs to train"
)
parser.add_argument(
    "--scheduler", type=str, default="steplr_1_0.5",
    help="scheduler type",
)
parser.add_argument(
    "--accumulation_steps", type=int, default=1,
    help="number of gradient accumulation steps",
)
parser.add_argument(
    "--save_every", type=int, default=1,
    help="how frequently to save a model",
)
parser.add_argument(
    "--device", type=str, required=True,
    help="whether to train on cuda or cpu",
    choices=("cuda", "cpu")
)
parser.add_argument(
    "--aggregation_type", type=str, required=True,
    help="how to aggregate outputs",
    choices=("max", "rnn")
)
parser.add_argument(
    "--num_conv_blocks", type=int, default=5,
    help="number of conv blocks"
)
parser.add_argument(
    "--start_deep_supervision_on", type=int, default=2,
    help="from which layer to start aggregating features for cla
ssification"
)
parser.add_argument(
    "--conv_base_depth", type=int, default=64,
    help="base depth for conv layers"
)
parser.add_argument(
    "--growth_rate", type=float, default=2,
    help="how quickly to increase the number of units as a funct
ion of layer"
)
parser.add_argument(
    "--weight_decay", type=float, default=1e-5,
    help="weight decay"
)
parser.add_argument(
    "--output_dropout", type=float, default=0.0,
    help="output dropout"
)
parser.add_argument(
    "--p_mixup", type=float, default=0.0,
    help="probability of the mixup augmentation"
)
```

```
parser.add_argument(
    "--p_aug", type=float, default=0.0,
    help="probability of audio augmentation"
)
parser.add_argument(
    "--switch_off_augmentations_on", type=int, default=20,
    help="on which epoch to remove augmentations"
)
parser.add_argument(
    "--features", type=str, required=True,
    help="feature descriptor"
)
parser.add_argument(
    "--optimizer", type=str, required=True,
    help="which optimizer to use",
    choices=("adam", "momentum")
)
parser.add_argument(
    "--folds", type=int, required=True, nargs="+",
    help="which folds to use"
)
parser.add_argument(
    "--n_folds", type=int, default=4,
    help="number of folds"
)
parser.add_argument(
    "--kfold_seed", type=int, default=42,
    help="kfold seed"
)
parser.add_argument(
    "--num_workers", type=int, default=4,
    help="number of workers for data loader",
)
parser.add_argument(
    "--label", type=str, default="2d_cnn",
    help="optional label",
)
args = parser.parse_args()

class_map = load_json(args.classmap)

audio_transform = AudioFeatures(args.features)

with Experiment({
    "network": {
        "num_conv_blocks": args.num_conv_blocks,
        "start_deep_supervision_on": args.start_deep_supervision
_on,
        "conv_base_depth": args.conv_base_depth,
        "growth_rate": args.growth_rate,
        "output_dropout": args.output_dropout,
        "aggregation_type": args.aggregation_type
    },
    "data": {
        "features": args.features,
        "_n_folds": args.n_folds,
```

```
        "_kfold_seed": args.kfold_seed,
        "_input_dim": audio_transform.n_features,
        "_n_classes": len(class_map),
        "_holdout_size": args.holdout_size,
        "p_mixup": args.p_mixup,
        "p_aug": args.p_aug,
        "max_audio_length": args.max_audio_length,
        "noisy": args.noisy_train_df is not None,
        "_train_df": args.train_df,
        "_train_data_dir": args.train_data_dir,
        "_noisy_train_df": args.noisy_train_df,
        "_noisy_train_data_dir": args.noisy_train_data_dir,
        "_share_noisy": args.share_noisy
    },
    "train": {
        "accumulation_steps": args.accumulation_steps,
        "batch_size": args.batch_size,
        "learning_rate": args.lr,
        "scheduler": args.scheduler,
        "optimizer": args.optimizer,
        "epochs": args.epochs,
        "_save_every": args.save_every,
        "weight_decay": args.weight_decay,
        "switch_off_augmentations_on": args.switch_off_augmentat
ions_on
    },
    "label": args.label
}, implicit_resuming=args.resume) as experiment:

    config = experiment.config
    print()
    print("     ////// CONFIG //////")
    print(experiment.config)

    train_df = pd.read_csv(args.train_df)
    test_df = pd.read_csv(args.sample_submission)

    if args.noisy_train_df:
        noisy_train_df = pd.read_csv(args.noisy_train_df)

    if args.max_samples:
        train_df = train_df.sample(args.max_samples).reset_index
(drop=True)
        test_df = test_df.sample(
            min(args.max_samples, len(test_df))).reset_index(dro
p=True)

    if args.holdout_size:
        keep, holdout = train_test_split(
            np.arange(len(train_df)), test_size=args.holdout_siz
e,
            random_state=args.kfold_seed)
        holdout_df = train_df.iloc[holdout].reset_index(drop=Tru
e)
        train_df = train_df.iloc[keep].reset_index(drop=True)
```

```
    splits = list(train_validation_data_stratified(
        train_df.fname, train_df.labels, class_map,
        config.data._n_folds, config.data._kfold_seed))

    if args.noisy_train_df:
        noisy_splits = list(train_validation_data(
            noisy_train_df.fname, noisy_train_df.labels,
            config.data._n_folds, config.data._kfold_seed))

    for fold in args.folds:

        print("\n\n   -----  Fold {}\n".format(fold))

        train, valid = splits[fold]

        loader_kwargs = (
            {"num_workers": args.num_workers, "pin_memory": True
}
            if torch.cuda.is_available() else {})

        experiment.register_directory("checkpoints")
        experiment.register_directory("predictions")

        if args.noisy_train_df:

            noisy_train, noisy_valid = noisy_splits[fold]

            if config.data._share_noisy:
                noisy_audio_files = [
                    os.path.join(args.noisy_train_data_dir, fnam
e)
                    for fname in noisy_train_df.fname.values]
                noisy_labels = [
                    item.split(",") for item in
                    noisy_train_df.labels.values]
            else:
                noisy_audio_files = [
                    os.path.join(args.noisy_train_data_dir, fnam
e)
                    for fname in noisy_train_df.fname.values[noi
sy_valid]]
                noisy_labels = [
                    item.split(",") for item in
                    noisy_train_df.labels.values[noisy_valid]]
        else:
            noisy_audio_files = []
            noisy_labels = []

        train_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[train]] +
 noisy_audio_files,
                labels=[
                    item.split(",") for item in
```

```
                    train_df.labels.values[train]] + noisy_label
s,
                is_noisy=[0] * len(train) + [1] * len(noisy_labe
ls),
                transform=Compose([
                    LoadAudio(),
                    SampleLongAudio(max_length=args.max_audio_le
ngth),
                    MapLabels(class_map=class_map),
                    (
                        ShuffleAudio(chunk_length=0.5, p=0.5)
                        if config.network.aggregation_type != "r
nn" else Identity()
                    ),
                    MixUp(p=args.p_mixup),
                    AudioAugmentation(p=args.p_aug),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ]),
                clean_transform=Compose([
                    LoadAudio(),
                    SampleLongAudio(max_length=args.max_audio_le
ngth),
                    MapLabels(class_map=class_map),
                ])
            ),
            shuffle=True,
            drop_last=True,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        valid_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[valid]],
                labels=[item.split(",") for item in train_df.lab
els.values[valid]],
                transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        model = TwoDimensionalCNNClassificationModel(
```

```
            experiment, device=args.device)

        scores = model.fit_validate(
            train_loader, valid_loader,
            epochs=experiment.config.train.epochs, fold=fold,
            log_interval=args.log_interval
        )

        best_metric = max(scores)
        experiment.register_result("fold{}.metric".format(fold),
 best_metric)

        torch.save(
            model.state_dict(),
            os.path.join(
                experiment.checkpoints,
                "fold_{}".format(fold),
                "final_model.pth")
        )

        # predictions
        model.load_best_model(fold)

        # validation

        val_preds = model.predict(valid_loader)
        val_predictions_df = pd.DataFrame(
            val_preds, columns=get_class_names_from_classmap(cla
ss_map))
        val_predictions_df["fname"] = train_df.fname[valid].valu
es
        val_predictions_df.to_csv(
            os.path.join(
                experiment.predictions,
                "val_preds_fold_{}.csv".format(fold)
            ),
            index=False
        )
        del val_predictions_df

        # test
        test_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.test_data_dir, fname)
                    for fname in test_df.fname.values],
                transform=Compose([
                    LoadAudio(),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
```

```
            **loader_kwargs
        )

        test_preds = model.predict(test_loader)
        test_predictions_df = pd.DataFrame(
            test_preds, columns=get_class_names_from_classmap(cl
ass_map))
        test_predictions_df["fname"] = test_df.fname
        test_predictions_df.to_csv(
            os.path.join(
                experiment.predictions,
                "test_preds_fold_{}.csv".format(fold)
            ),
            index=False
        )
        del test_predictions_df

        # holdout
        if args.holdout_size:
            holdout_loader = torch.utils.data.DataLoader(
                SoundDataset(
                    audio_files=[
                        os.path.join(args.train_data_dir, fname)
                        for fname in holdout_df.fname.values],
                    labels=[item.split(",") for item in holdout_
df.labels.values],
                    transform=Compose([
                        LoadAudio(),
                        MapLabels(class_map),
                        audio_transform,
                        DropFields(("audio", "filename", "sr")),
                    ])
                ),
                shuffle=False,
                batch_size=config.train.batch_size,
                collate_fn=make_collate_fn({"signal": audio_tran
sform.padding_value}),
                **loader_kwargs
            )

            holdout_metric = model.evaluate(holdout_loader)
            experiment.register_result(
                "fold{}.holdout_metric".format(fold), holdout_me
tric)

            print("\nHoldout metric: {:.4f}".format(holdout_metr
ic))

        if args.device == "cuda":
            torch.cuda.empty_cache()

    # global metric

    if all(
        "fold{}".format(k) in experiment.results.to_dict()
        for k in range(config.data._n_folds)):
```

```
        val_df_files = [
            os.path.join(
                experiment.predictions,
                "val_preds_fold_{}.csv".format(fold)
            )
            for fold in range(config.data._n_folds)
        ]

        val_predictions_df = pd.concat([
            pd.read_csv(file) for file in val_df_files]).reset_i
ndex(drop=True)

        labels = np.asarray([
            item["labels"] for item in SoundDataset(
                audio_files=train_df.fname.tolist(),
                labels=[item.split(",") for item in train_df.lab
els.values],
                transform=MapLabels(class_map)
            )
        ])

        val_labels_df = pd.DataFrame(
            labels, columns=get_class_names_from_classmap(class_
map))
        val_labels_df["fname"] = train_df.fname

        assert set(val_predictions_df.fname) == set(val_labels_d
f.fname)

        val_predictions_df.sort_values(by="fname", inplace=True)
        val_labels_df.sort_values(by="fname", inplace=True)

        metric = lwlrap(
            val_labels_df.drop("fname", axis=1).values,
            val_predictions_df.drop("fname", axis=1).values
        )

        experiment.register_result("metric", metric)

    # submission

    test_df_files = [
        os.path.join(
            experiment.predictions,
            "test_preds_fold_{}.csv".format(fold)
        )
        for fold in range(config.data._n_folds)
    ]

    if all(os.path.isfile for file in test_df_files):
        test_dfs = [pd.read_csv(file) for file in test_df_files]
        submission_df = pd.DataFrame({"fname": test_dfs[0].fname
.values})
        for c in get_class_names_from_classmap(class_map):
            submission_df[c] = np.mean([d[c].values for d in tes
```

```
t_dfs], axis=0)
        submission_df.to_csv(
            os.path.join(experiment.predictions, "submission.csv
"), index=False)======
import os
import gc
import argparse
import json
import math
from functools import partial

import pandas as pd
import numpy as np
import torch
from mag.experiment import Experiment
import mag
from sklearn.model_selection import train_test_split

from datasets.sound_dataset import SoundDataset
from networks.classifiers import CNNBackboneClassificationModel
from ops.folds import train_validation_data, train_validation_da
ta_stratified
from ops.transforms import (
    Compose, DropFields, LoadAudio,
    AudioFeatures, MapLabels, RenameFields,
    MixUp, SampleSegment, SampleLongAudio,
    AudioAugmentation, ShuffleAudio, CutOut, Identity)
from ops.utils import load_json, get_class_names_from_classmap,
lwlrap
from ops.padding import make_collate_fn

torch.manual_seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(42)

mag.use_custom_separator("-")

parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--train_df", required=True, type=str,
    help="path to train dataframe"
)
parser.add_argument(
    "--train_data_dir", required=True, type=str,
    help="path to train data"
)
parser.add_argument(
    "--noisy_train_df", type=str,
    help="path to noisy train dataframe (optional)"
)
parser.add_argument(
    "--noisy_train_data_dir", type=str,
    help="path to noisy train data (optional)"
```

```
)
parser.add_argument(
    "--share_noisy", action="store_true", default=False,
    help="whether to share noisy files across folds"
)
parser.add_argument(
    "--resume", action="store_true", default=False,
    help="allow resuming even if experiment exists"
)
parser.add_argument(
    "--test_data_dir", required=True, type=str,
    help="path to test data"
)
parser.add_argument(
    "--sample_submission", required=True, type=str,
    help="path sample submission"
)
parser.add_argument(
    "--classmap", required=True, type=str,
    help="path to class map json"
)
parser.add_argument(
    "--log_interval", default=10, type=int,
    help="how frequently to log batch metrics"
    "in terms of processed batches"
)
parser.add_argument(
    "--batch_size", type=int, default=64,
    help="minibatch size"
)
parser.add_argument(
    "--max_audio_length", type=int, default=10,
    help="max audio length in seconds. For longer clips are samp
led"
)
parser.add_argument(
    "--lr", default=0.01, type=float,
    help="starting learning rate"
)
parser.add_argument(
    "--max_samples", type=int,
    help="maximum number of samples to use"
)
parser.add_argument(
    "--holdout_size", type=float, default=0.0,
    help="size of holdout set"
)
parser.add_argument(
    "--epochs", default=100, type=int,
    help="number of epochs to train"
)
parser.add_argument(
    "--scheduler", type=str, default="steplr_1_0.5",
    help="scheduler type",
)
parser.add_argument(
```

```
        "--accumulation_steps", type=int, default=1,
        help="number of gradient accumulation steps",
    )
    parser.add_argument(
        "--save_every", type=int, default=1,
        help="how frequently to save a model",
    )
    parser.add_argument(
        "--device", type=str, required=True,
        help="whether to train on cuda or cpu",
        choices=("cuda", "cpu")
    )
    parser.add_argument(
        "--backbone", type=str, required=True,
        help="which backbone to use",
        choices=("resnet18", "resnet34")
    )
    parser.add_argument(
        "--weight_decay", type=float, default=1e-5,
        help="weight decay"
    )
    parser.add_argument(
        "--output_dropout", type=float, default=0.0,
        help="output dropout"
    )
    parser.add_argument(
        "--p_mixup", type=float, default=0.0,
        help="probability of the mixup augmentation"
    )
    parser.add_argument(
        "--p_aug", type=float, default=0.0,
        help="probability of audio augmentation"
    )
    parser.add_argument(
        "--switch_off_augmentations_on", type=int, default=20,
        help="on which epoch to remove augmentations"
    )
    parser.add_argument(
        "--features", type=str, required=True,
        help="feature descriptor"
    )
    parser.add_argument(
        "--optimizer", type=str, required=True,
        help="which optimizer to use",
        choices=("adam", "momentum")
    )
    parser.add_argument(
        "--folds", type=int, required=True, nargs="+",
        help="which folds to use"
    )
    parser.add_argument(
        "--n_folds", type=int, default=4,
        help="number of folds"
    )
    parser.add_argument(
        "--kfold_seed", type=int, default=42,
```

```
    help="kfold seed"
)
parser.add_argument(
    "--num_workers", type=int, default=4,
    help="number of workers for data loader",
)
parser.add_argument(
    "--label", type=str, default="backbone",
    help="optional label",
)
args = parser.parse_args()

class_map = load_json(args.classmap)

audio_transform = AudioFeatures(args.features)

with Experiment({
    "network": {
        "backbone": args.backbone,
        "output_dropout": args.output_dropout,
    },
    "data": {
        "features": args.features,
        "_n_folds": args.n_folds,
        "_kfold_seed": args.kfold_seed,
        "_input_dim": audio_transform.n_features,
        "_n_classes": len(class_map),
        "_holdout_size": args.holdout_size,
        "p_mixup": args.p_mixup,
        "p_aug": args.p_aug,
        "max_audio_length": args.max_audio_length,
        "noisy": args.noisy_train_df is not None,
        "_train_df": args.train_df,
        "_train_data_dir": args.train_data_dir,
        "_noisy_train_df": args.noisy_train_df,
        "_noisy_train_data_dir": args.noisy_train_data_dir,
        "_share_noisy": args.share_noisy
    },
    "train": {
        "accumulation_steps": args.accumulation_steps,
        "batch_size": args.batch_size,
        "learning_rate": args.lr,
        "scheduler": args.scheduler,
        "optimizer": args.optimizer,
        "epochs": args.epochs,
        "_save_every": args.save_every,
        "weight_decay": args.weight_decay,
        "switch_off_augmentations_on": args.switch_off_augmentat
ions_on
    },
    "label": args.label
}, implicit_resuming=args.resume) as experiment:

    config = experiment.config
    print()
    print("     ////// CONFIG //////")
```

```
    print(experiment.config)

    train_df = pd.read_csv(args.train_df)
    test_df = pd.read_csv(args.sample_submission)

    if args.noisy_train_df:
        noisy_train_df = pd.read_csv(args.noisy_train_df)

    if args.max_samples:
        train_df = train_df.sample(args.max_samples).reset_index
(drop=True)
        test_df = test_df.sample(
            min(args.max_samples, len(test_df))).reset_index(dro
p=True)

    if args.holdout_size:
        keep, holdout = train_test_split(
            np.arange(len(train_df)), test_size=args.holdout_siz
e,
            random_state=args.kfold_seed)
        holdout_df = train_df.iloc[holdout].reset_index(drop=Tru
e)
        train_df = train_df.iloc[keep].reset_index(drop=True)

    splits = list(train_validation_data_stratified(
        train_df.fname, train_df.labels, class_map,
        config.data._n_folds, config.data._kfold_seed))

    if args.noisy_train_df:
        noisy_splits = list(train_validation_data(
            noisy_train_df.fname, noisy_train_df.labels,
            config.data._n_folds, config.data._kfold_seed))

    for fold in args.folds:

        print("\n\n   -----  Fold {}\n".format(fold))

        train, valid = splits[fold]

        loader_kwargs = (
            {"num_workers": args.num_workers, "pin_memory": True
}
            if torch.cuda.is_available() else {})

        experiment.register_directory("checkpoints")
        experiment.register_directory("predictions")

        if args.noisy_train_df:

            noisy_train, noisy_valid = noisy_splits[fold]

            if config.data._share_noisy:
                noisy_audio_files = [
                    os.path.join(args.noisy_train_data_dir, fnam
e)
                    for fname in noisy_train_df.fname.values]
```

```
                noisy_labels = [
                    item.split(",") for item in
                    noisy_train_df.labels.values]
            else:
                noisy_audio_files = [
                    os.path.join(args.noisy_train_data_dir, fnam
e)
                    for fname in noisy_train_df.fname.values[noi
sy_valid]]
                noisy_labels = [
                    item.split(",") for item in
                    noisy_train_df.labels.values[noisy_valid]]
        else:
            noisy_audio_files = []
            noisy_labels = []

        train_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[train]] +
 noisy_audio_files,
                labels=[
                    item.split(",") for item in
                    train_df.labels.values[train]] + noisy_label
s,
                is_noisy=[0] * len(train) + [1] * len(noisy_labe
ls),
                transform=Compose([
                    LoadAudio(),
                    SampleLongAudio(max_length=args.max_audio_le
ngth),
                    MapLabels(class_map=class_map),
                    ShuffleAudio(chunk_length=0.5, p=0.5),
                    MixUp(p=args.p_mixup),
                    AudioAugmentation(p=args.p_aug),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ]),
                clean_transform=Compose([
                    LoadAudio(),
                    SampleLongAudio(max_length=args.max_audio_le
ngth),
                    MapLabels(class_map=class_map),
                ])
            ),
            shuffle=True,
            drop_last=True,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        valid_loader = torch.utils.data.DataLoader(
            SoundDataset(
```

```
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[valid]],
                labels=[item.split(",") for item in train_df.lab
els.values[valid]],
                transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        model = CNNBackboneClassificationModel(experiment, devic
e=args.device)

        scores = model.fit_validate(
            train_loader, valid_loader,
            epochs=experiment.config.train.epochs, fold=fold,
            log_interval=args.log_interval
        )

        best_metric = max(scores)
        experiment.register_result("fold{}.metric".format(fold),
 best_metric)

        torch.save(
            model.state_dict(),
            os.path.join(
                experiment.checkpoints,
                "fold_{}".format(fold),
                "final_model.pth")
        )

        # predictions
        model.load_best_model(fold)

        # validation

        val_preds = model.predict(valid_loader)
        val_predictions_df = pd.DataFrame(
            val_preds, columns=get_class_names_from_classmap(cla
ss_map))
        val_predictions_df["fname"] = train_df.fname[valid].valu
es
        val_predictions_df.to_csv(
            os.path.join(
                experiment.predictions,
                "val_preds_fold_{}.csv".format(fold)
            ),
```

```
            index=False
        )
        del val_predictions_df

        # test
        test_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.test_data_dir, fname)
                    for fname in test_df.fname.values],
                transform=Compose([
                    LoadAudio(),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        test_preds = model.predict(test_loader)
        test_predictions_df = pd.DataFrame(
            test_preds, columns=get_class_names_from_classmap(cl
ass_map))
        test_predictions_df["fname"] = test_df.fname
        test_predictions_df.to_csv(
            os.path.join(
                experiment.predictions,
                "test_preds_fold_{}.csv".format(fold)
            ),
            index=False
        )
        del test_predictions_df

        # holdout
        if args.holdout_size:
            holdout_loader = torch.utils.data.DataLoader(
                SoundDataset(
                    audio_files=[
                        os.path.join(args.train_data_dir, fname)
                        for fname in holdout_df.fname.values],
                    labels=[item.split(",") for item in holdout_
df.labels.values],
                    transform=Compose([
                        LoadAudio(),
                        MapLabels(class_map),
                        audio_transform,
                        DropFields(("audio", "filename", "sr")),
                    ])
                ),
                shuffle=False,
                batch_size=config.train.batch_size,
                collate_fn=make_collate_fn({"signal": audio_tran
```

```
sform.padding_value}),
                **loader_kwargs
            )

            holdout_metric = model.evaluate(holdout_loader)
            experiment.register_result(
                "fold{}.holdout_metric".format(fold), holdout_me
tric)

            print("\nHoldout metric: {:.4f}".format(holdout_metr
ic))

        if args.device == "cuda":
            torch.cuda.empty_cache()

    # global metric

    if all(
        "fold{}".format(k) in experiment.results.to_dict()
        for k in range(config.data._n_folds)):

        val_df_files = [
            os.path.join(
                experiment.predictions,
                "val_preds_fold_{}.csv".format(fold)
            )
            for fold in range(config.data._n_folds)
        ]

        val_predictions_df = pd.concat([
            pd.read_csv(file) for file in val_df_files]).reset_i
ndex(drop=True)

        labels = np.asarray([
            item["labels"] for item in SoundDataset(
                audio_files=train_df.fname.tolist(),
                labels=[item.split(",") for item in train_df.lab
els.values],
                transform=MapLabels(class_map)
            )
        ])

        val_labels_df = pd.DataFrame(
            labels, columns=get_class_names_from_classmap(class_
map))
        val_labels_df["fname"] = train_df.fname

        assert set(val_predictions_df.fname) == set(val_labels_d
f.fname)

        val_predictions_df.sort_values(by="fname", inplace=True)
        val_labels_df.sort_values(by="fname", inplace=True)

        metric = lwlrap(
            val_labels_df.drop("fname", axis=1).values,
            val_predictions_df.drop("fname", axis=1).values
```

```
        )

        experiment.register_result("metric", metric)

    # submission

    test_df_files = [
        os.path.join(
            experiment.predictions,
            "test_preds_fold_{}.csv".format(fold)
        )
        for fold in range(config.data._n_folds)
    ]

    if all(os.path.isfile for file in test_df_files):
        test_dfs = [pd.read_csv(file) for file in test_df_files]
        submission_df = pd.DataFrame({"fname": test_dfs[0].fname
.values})
        for c in get_class_names_from_classmap(class_map):
            submission_df[c] = np.mean([d[c].values for d in tes
t_dfs], axis=0)
        submission_df.to_csv(
            os.path.join(experiment.predictions, "submission.csv
"), index=False)======
import os
import gc
import argparse
import json
import math
from functools import partial

import pandas as pd
import numpy as np
import torch
from mag.experiment import Experiment
import mag
from sklearn.model_selection import train_test_split

from ops.utils import load_json, get_class_names_from_classmap
from datasets.sound_dataset import SoundDataset
from networks.cpc import CPCModel
from ops.folds import train_validation_data
from ops.transforms import (
    Compose, DropFields, LoadAudio,
    AudioFeatures, MapLabels, RenameFields,
    MixUp, SampleSegment, SampleLongAudio,
    AudioAugmentation)
from ops.padding import make_collate_fn

torch.manual_seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(42)

mag.use_custom_separator("-")

parser = argparse.ArgumentParser(
```

```
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--train_df", required=True, type=str,
    help="path to train dataframe"
)
parser.add_argument(
    "--train_data_dir", required=True, type=str,
    help="path to train data"
)
parser.add_argument(
    "--classmap", required=True, type=str,
    help="path to class map json"
)
parser.add_argument(
    "--log_interval", default=10, type=int,
    help="how frequently to log batch metrics"
    "in terms of processed batches"
)
parser.add_argument(
    "--proj_interval", default=10, type=int,
    help="how frequently to make projection in terms of epochs"
)
parser.add_argument(
    "--batch_size", type=int, default=64,
    help="minibatch size"
)
parser.add_argument(
    "--max_audio_length", type=int, default=10,
    help="max audio length in seconds. For longer clips are samp
led"
)
parser.add_argument(
    "--lr", default=0.01, type=float,
    help="starting learning rate"
)
parser.add_argument(
    "--max_samples", type=int,
    help="maximum number of samples to use"
)
parser.add_argument(
    "--epochs", default=100, type=int,
    help="number of epochs to train"
)
parser.add_argument(
    "--scheduler", type=str, default="steplr_1_0.5",
    help="scheduler type",
)
parser.add_argument(
    "--accumulation_steps", type=int, default=1,
    help="number of gradient accumulation steps",
)
parser.add_argument(
    "--save_every", type=int, default=1,
    help="how frequently to save a model",
```

```
)
parser.add_argument(
    "--device", type=str, required=True,
    help="whether to train on cuda or cpu",
    choices=("cuda", "cpu")
)
parser.add_argument(
    "--n_encoder_layers", type=int, default=5,
    help="number of encoder layers"
)
parser.add_argument(
    "--conv_base_depth", type=int, default=64,
    help="base depth for conv layers"
)
parser.add_argument(
    "--context_size", type=int, default=64,
    help="context size for c network"
)
parser.add_argument(
    "--growth_rate", type=float, default=2,
    help="how quickly to increase the number of units as a funct
ion of layer"
)
parser.add_argument(
    "--prediction_steps", type=int, default=10,
    help="how many steps to predict in the future"
)
parser.add_argument(
    "--weight_decay", type=float, default=1e-5,
    help="weight decay"
)
parser.add_argument(
    "--p_aug", type=float, default=0.0,
    help="probability of audio augmentation"
)
parser.add_argument(
    "--switch_off_augmentations_on", type=int, default=20,
    help="on which epoch to remove augmentations"
)
parser.add_argument(
    "--features", type=str, required=True,
    help="feature descriptor"
)
parser.add_argument(
    "--optimizer", type=str, required=True,
    help="which optimizer to use",
    choices=("adam", "momentum")
)
parser.add_argument(
    "--folds", type=int, required=True, nargs="+",
    help="which folds to use"
)
parser.add_argument(
    "--n_folds", type=int, default=4,
    help="number of folds"
)
```

```
parser.add_argument(
    "--kfold_seed", type=int, default=42,
    help="kfold seed"
)
parser.add_argument(
    "--num_workers", type=int, default=4,
    help="number of workers for data loader",
)
parser.add_argument(
    "--label", type=str, default="cpc",
    help="optional label",
)
args = parser.parse_args()

class_map = load_json(args.classmap)

audio_transform = AudioFeatures(args.features)

with Experiment({
    "network": {
        "n_encoder_layers": args.n_encoder_layers,
        "conv_base_depth": args.conv_base_depth,
        "growth_rate": args.growth_rate,
        "prediction_steps": args.prediction_steps,
        "context_size": args.context_size
    },
    "data": {
        "features": args.features,
        "_n_folds": args.n_folds,
        "_kfold_seed": args.kfold_seed,
        "_input_dim": audio_transform.n_features,
        "p_aug": args.p_aug,
        "max_audio_length": args.max_audio_length
    },
    "train": {
        "_proj_interval": args.proj_interval,
        "accumulation_steps": args.accumulation_steps,
        "batch_size": args.batch_size,
        "learning_rate": args.lr,
        "scheduler": args.scheduler,
        "optimizer": args.optimizer,
        "epochs": args.epochs,
        "_save_every": args.save_every,
        "weight_decay": args.weight_decay,
        "switch_off_augmentations_on": args.switch_off_augmentat
ions_on
    },
    "label": args.label
}) as experiment:

    config = experiment.config
    print()
    print("     ////// CONFIG //////")
    print(experiment.config)

    train_df = pd.read_csv(args.train_df)
```

```
    if args.max_samples:
        train_df = train_df.sample(args.max_samples).reset_index
(drop=True)

    splits = list(train_validation_data(
        train_df.fname, train_df.labels,
        config.data._n_folds, config.data._kfold_seed))

    for fold in args.folds:

        print("\n\n   -----  Fold {}\n".format(fold))

        train, valid = splits[fold]

        loader_kwargs = (
            {"num_workers": args.num_workers, "pin_memory": True
}
            if torch.cuda.is_available() else {})

        experiment.register_directory("checkpoints")
        experiment.register_directory("predictions")

        train_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[train]],
                labels=[
                    item.split(",") for item in
                    train_df.labels.values[train]],
                transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                    SampleLongAudio(max_length=args.max_audio_le
ngth),
                    AudioAugmentation(p=args.p_aug),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ])
            ),
            shuffle=True,
            drop_last=True,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        valid_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[valid]],
                labels=[
                    item.split(",") for item in
```

```
                    train_df.labels.values[valid]],
                transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                    SampleLongAudio(max_length=args.max_audio_le
ngth),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        model = CPCModel(experiment, device=args.device)

        scores = model.fit_validate(
            train_loader, valid_loader,
            epochs=experiment.config.train.epochs, fold=fold,
            log_interval=args.log_interval
        )

        best_metric = max(scores)
        experiment.register_result("fold{}.metric".format(fold),
 best_metric)

        torch.save(
            model.state_dict(),
            os.path.join(
                experiment.checkpoints,
                "fold_{}".format(fold),
                "final_model.pth")
        )

        # predictions
        model.load_best_model(fold)

        if args.device == "cuda":
            torch.cuda.empty_cache()
======
import os
import gc
import argparse
import json
import math
from functools import partial

import pandas as pd
import numpy as np
import torch
from mag.experiment import Experiment
import mag
from sklearn.model_selection import train_test_split
```

```
from datasets.sound_dataset import SoundDataset
from networks.classifiers import HierarchicalCNNClassificationMo
del
from ops.folds import train_validation_data, train_validation_da
ta_stratified
from ops.transforms import (
    Compose, DropFields, LoadAudio,
    AudioFeatures, MapLabels, RenameFields,
    MixUp, SampleSegment, SampleLongAudio,
    AudioAugmentation, ShuffleAudio, CutOut, Identity)
from ops.utils import load_json, get_class_names_from_classmap,
lwlrap
from ops.padding import make_collate_fn

torch.manual_seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(42)

mag.use_custom_separator("-")

parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--train_df", required=True, type=str,
    help="path to train dataframe"
)
parser.add_argument(
    "--train_data_dir", required=True, type=str,
    help="path to train data"
)
parser.add_argument(
    "--noisy_train_df", type=str,
    help="path to noisy train dataframe (optional)"
)
parser.add_argument(
    "--noisy_train_data_dir", type=str,
    help="path to noisy train data (optional)"
)
parser.add_argument(
    "--share_noisy", action="store_true", default=False,
    help="whether to share noisy files across folds"
)
parser.add_argument(
    "--resume", action="store_true", default=False,
    help="allow resuming even if experiment exists"
)
parser.add_argument(
    "--test_data_dir", required=True, type=str,
    help="path to test data"
)
parser.add_argument(
    "--sample_submission", required=True, type=str,
    help="path sample submission"
```

```
)
parser.add_argument(
    "--classmap", required=True, type=str,
    help="path to class map json"
)
parser.add_argument(
    "--log_interval", default=10, type=int,
    help="how frequently to log batch metrics"
    "in terms of processed batches"
)
parser.add_argument(
    "--batch_size", type=int, default=64,
    help="minibatch size"
)
parser.add_argument(
    "--max_audio_length", type=int, default=10,
    help="max audio length in seconds. For longer clips are samp
led"
)
parser.add_argument(
    "--lr", default=0.01, type=float,
    help="starting learning rate"
)
parser.add_argument(
    "--max_samples", type=int,
    help="maximum number of samples to use"
)
parser.add_argument(
    "--holdout_size", type=float, default=0.0,
    help="size of holdout set"
)
parser.add_argument(
    "--epochs", default=100, type=int,
    help="number of epochs to train"
)
parser.add_argument(
    "--scheduler", type=str, default="steplr_1_0.5",
    help="scheduler type",
)
parser.add_argument(
    "--accumulation_steps", type=int, default=1,
    help="number of gradient accumulation steps",
)
parser.add_argument(
    "--save_every", type=int, default=1,
    help="how frequently to save a model",
)
parser.add_argument(
    "--device", type=str, required=True,
    help="whether to train on cuda or cpu",
    choices=("cuda", "cpu")
)
parser.add_argument(
    "--aggregation_type", type=str, required=True,
    help="how to aggregate outputs",
    choices=("max", "rnn")
```

```
)
parser.add_argument(
    "--num_conv_blocks", type=int, default=5,
    help="number of conv blocks"
)
parser.add_argument(
    "--start_deep_supervision_on", type=int, default=2,
    help="from which layer to start aggregating features for cla
ssification"
)
parser.add_argument(
    "--conv_base_depth", type=int, default=64,
    help="base depth for conv layers"
)
parser.add_argument(
    "--growth_rate", type=float, default=2,
    help="how quickly to increase the number of units as a funct
ion of layer"
)
parser.add_argument(
    "--weight_decay", type=float, default=1e-5,
    help="weight decay"
)
parser.add_argument(
    "--output_dropout", type=float, default=0.0,
    help="output dropout"
)
parser.add_argument(
    "--p_mixup", type=float, default=0.0,
    help="probability of the mixup augmentation"
)
parser.add_argument(
    "--p_aug", type=float, default=0.0,
    help="probability of audio augmentation"
)
parser.add_argument(
    "--switch_off_augmentations_on", type=int, default=20,
    help="on which epoch to remove augmentations"
)
parser.add_argument(
    "--features", type=str, required=True,
    help="feature descriptor"
)
parser.add_argument(
    "--optimizer", type=str, required=True,
    help="which optimizer to use",
    choices=("adam", "momentum")
)
parser.add_argument(
    "--folds", type=int, required=True, nargs="+",
    help="which folds to use"
)
parser.add_argument(
    "--n_folds", type=int, default=4,
    help="number of folds"
)
```

```
parser.add_argument(
    "--kfold_seed", type=int, default=42,
    help="kfold seed"
)
parser.add_argument(
    "--num_workers", type=int, default=4,
    help="number of workers for data loader",
)
parser.add_argument(
    "--label", type=str, default="1d_cnn",
    help="optional label",
)
args = parser.parse_args()

class_map = load_json(args.classmap)

audio_transform = AudioFeatures(args.features)

with Experiment({
    "network": {
        "num_conv_blocks": args.num_conv_blocks,
        "start_deep_supervision_on": args.start_deep_supervision
_on,
        "conv_base_depth": args.conv_base_depth,
        "growth_rate": args.growth_rate,
        "output_dropout": args.output_dropout,
        "aggregation_type": args.aggregation_type
    },
    "data": {
        "features": args.features,
        "_n_folds": args.n_folds,
        "_kfold_seed": args.kfold_seed,
        "_input_dim": audio_transform.n_features,
        "_n_classes": len(class_map),
        "_holdout_size": args.holdout_size,
        "p_mixup": args.p_mixup,
        "p_aug": args.p_aug,
        "max_audio_length": args.max_audio_length,
        "noisy": args.noisy_train_df is not None,
        "_train_df": args.train_df,
        "_train_data_dir": args.train_data_dir,
        "_noisy_train_df": args.noisy_train_df,
        "_noisy_train_data_dir": args.noisy_train_data_dir,
        "_share_noisy": args.share_noisy
    },
    "train": {
        "accumulation_steps": args.accumulation_steps,
        "batch_size": args.batch_size,
        "learning_rate": args.lr,
        "scheduler": args.scheduler,
        "optimizer": args.optimizer,
        "epochs": args.epochs,
        "_save_every": args.save_every,
        "weight_decay": args.weight_decay,
        "switch_off_augmentations_on": args.switch_off_augmentat
ions_on
```

```
    },
    "label": args.label
}, implicit_resuming=args.resume) as experiment:

    config = experiment.config
    print()
    print("     ////// CONFIG //////")
    print(experiment.config)

    train_df = pd.read_csv(args.train_df)
    test_df = pd.read_csv(args.sample_submission)

    if args.noisy_train_df:
        noisy_train_df = pd.read_csv(args.noisy_train_df)

    if args.max_samples:
        train_df = train_df.sample(args.max_samples).reset_index
(drop=True)
        test_df = test_df.sample(
            min(args.max_samples, len(test_df))).reset_index(dro
p=True)

    if args.holdout_size:
        keep, holdout = train_test_split(
            np.arange(len(train_df)), test_size=args.holdout_siz
e,
            random_state=args.kfold_seed)
        holdout_df = train_df.iloc[holdout].reset_index(drop=Tru
e)
        train_df = train_df.iloc[keep].reset_index(drop=True)

    splits = list(train_validation_data_stratified(
        train_df.fname, train_df.labels, class_map,
        config.data._n_folds, config.data._kfold_seed))

    if args.noisy_train_df:
        noisy_splits = list(train_validation_data(
            noisy_train_df.fname, noisy_train_df.labels,
            config.data._n_folds, config.data._kfold_seed))

    for fold in args.folds:

        print("\n\n   -----  Fold {}\n".format(fold))

        train, valid = splits[fold]

        loader_kwargs = (
            {"num_workers": args.num_workers, "pin_memory": True
}
            if torch.cuda.is_available() else {})

        experiment.register_directory("checkpoints")
        experiment.register_directory("predictions")

        if args.noisy_train_df:
```

```
            noisy_train, noisy_valid = noisy_splits[fold]

            if config.data._share_noisy:
                noisy_audio_files = [
                    os.path.join(args.noisy_train_data_dir, fnam
e)
                    for fname in noisy_train_df.fname.values]
                noisy_labels = [
                    item.split(",") for item in
                    noisy_train_df.labels.values]
            else:
                noisy_audio_files = [
                    os.path.join(args.noisy_train_data_dir, fnam
e)
                    for fname in noisy_train_df.fname.values[noi
sy_valid]]
                noisy_labels = [
                    item.split(",") for item in
                    noisy_train_df.labels.values[noisy_valid]]
        else:
            noisy_audio_files = []
            noisy_labels = []

        train_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[train]] +
 noisy_audio_files,
                labels=[
                    item.split(",") for item in
                    train_df.labels.values[train]] + noisy_label
s,
                is_noisy=[0] * len(train) + [1] * len(noisy_labe
ls),
                transform=Compose([
                    LoadAudio(),
                    SampleLongAudio(max_length=args.max_audio_le
ngth),
                    MapLabels(class_map=class_map),
                    (
                        ShuffleAudio(chunk_length=0.5, p=0.5)
                        if config.network.aggregation_type != "r
nn" else Identity()
                    ),
                    MixUp(p=args.p_mixup),
                    AudioAugmentation(p=args.p_aug),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ]),
                clean_transform=Compose([
                    LoadAudio(),
                    SampleLongAudio(max_length=args.max_audio_le
ngth),
                    MapLabels(class_map=class_map),
                ])
```

```
            ),
            shuffle=True,
            drop_last=True,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        valid_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[valid]],
                labels=[item.split(",") for item in train_df.lab
els.values[valid]],
                transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        model = HierarchicalCNNClassificationModel(experiment, d
evice=args.device)

        scores = model.fit_validate(
            train_loader, valid_loader,
            epochs=experiment.config.train.epochs, fold=fold,
            log_interval=args.log_interval
        )

        best_metric = max(scores)
        experiment.register_result("fold{}.metric".format(fold),
 best_metric)

        torch.save(
            model.state_dict(),
            os.path.join(
                experiment.checkpoints,
                "fold_{}".format(fold),
                "final_model.pth")
        )

        # predictions
        model.load_best_model(fold)

        # validation
```

```
        val_preds = model.predict(valid_loader)
        val_predictions_df = pd.DataFrame(
            val_preds, columns=get_class_names_from_classmap(cla
ss_map))
        val_predictions_df["fname"] = train_df.fname[valid].valu
es
        val_predictions_df.to_csv(
            os.path.join(
                experiment.predictions,
                "val_preds_fold_{}.csv".format(fold)
            ),
            index=False
        )
        del val_predictions_df

        # test
        test_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.test_data_dir, fname)
                    for fname in test_df.fname.values],
                transform=Compose([
                    LoadAudio(),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        test_preds = model.predict(test_loader)
        test_predictions_df = pd.DataFrame(
            test_preds, columns=get_class_names_from_classmap(cl
ass_map))
        test_predictions_df["fname"] = test_df.fname
        test_predictions_df.to_csv(
            os.path.join(
                experiment.predictions,
                "test_preds_fold_{}.csv".format(fold)
            ),
            index=False
        )
        del test_predictions_df

        # holdout
        if args.holdout_size:
            holdout_loader = torch.utils.data.DataLoader(
                SoundDataset(
                    audio_files=[
                        os.path.join(args.train_data_dir, fname)
                        for fname in holdout_df.fname.values],
                    labels=[item.split(",") for item in holdout_
```

```
df.labels.values],
                    transform=Compose([
                        LoadAudio(),
                        MapLabels(class_map),
                        audio_transform,
                        DropFields(("audio", "filename", "sr")),
                    ])
                ),
                shuffle=False,
                batch_size=config.train.batch_size,
                collate_fn=make_collate_fn({"signal": audio_tran
sform.padding_value}),
                **loader_kwargs
            )

            holdout_metric = model.evaluate(holdout_loader)
            experiment.register_result(
                "fold{}.holdout_metric".format(fold), holdout_me
tric)

            print("\nHoldout metric: {:.4f}".format(holdout_metr
ic))

        if args.device == "cuda":
            torch.cuda.empty_cache()

    # global metric

    if all(
        "fold{}".format(k) in experiment.results.to_dict()
        for k in range(config.data._n_folds)):

        val_df_files = [
            os.path.join(
                experiment.predictions,
                "val_preds_fold_{}.csv".format(fold)
            )
            for fold in range(config.data._n_folds)
        ]

        val_predictions_df = pd.concat([
            pd.read_csv(file) for file in val_df_files]).reset_i
ndex(drop=True)

        labels = np.asarray([
            item["labels"] for item in SoundDataset(
                audio_files=train_df.fname.tolist(),
                labels=[item.split(",") for item in train_df.lab
els.values],
                transform=MapLabels(class_map)
            )
        ])

        val_labels_df = pd.DataFrame(
            labels, columns=get_class_names_from_classmap(class_
map))
```

```
        val_labels_df["fname"] = train_df.fname

        assert set(val_predictions_df.fname) == set(val_labels_d
f.fname)

        val_predictions_df.sort_values(by="fname", inplace=True)
        val_labels_df.sort_values(by="fname", inplace=True)

        metric = lwlrap(
            val_labels_df.drop("fname", axis=1).values,
            val_predictions_df.drop("fname", axis=1).values
        )

        experiment.register_result("metric", metric)

    # submission

    test_df_files = [
        os.path.join(
            experiment.predictions,
            "test_preds_fold_{}.csv".format(fold)
        )
        for fold in range(config.data._n_folds)
    ]

    if all(os.path.isfile for file in test_df_files):
        test_dfs = [pd.read_csv(file) for file in test_df_files]
        submission_df = pd.DataFrame({"fname": test_dfs[0].fname
.values})
        for c in get_class_names_from_classmap(class_map):
            submission_df[c] = np.mean([d[c].values for d in tes
t_dfs], axis=0)
        submission_df.to_csv(
            os.path.join(experiment.predictions, "submission.csv
"), index=False)======
```

```
import os
import math
import itertools
from collections import defaultdict, OrderedDict, deque

from tqdm import tqdm
import numpy as np
import torch
import torch.nn as nn
import torchvision.utils
from tensorboardX import SummaryWriter
from torch.nn.functional import binary_cross_entropy_with_lo
gits

from ops.training import OPTIMIZERS, make_scheduler, make_st
ep
from networks.losses import binary_cross_entropy, focal_loss
, lsep_loss
from ops.utils import plot_projection


class APCModel(nn.Module):

    def __init__(self, experiment, device="cuda"):
        super().__init__()

        self.device = device

        self.experiment = experiment
        self.config = experiment.config

        self.input_norm = nn.LayerNorm(
            (self.config.data._input_dim,), elementwise_affi
ne=False)

        self.rnn = nn.LSTM(
            self.config.data._input_dim, self.config.network
.rnn_size,
            num_layers=self.config.network.rnn_layers,
            batch_first=True
        )

        self.output_norm = nn.LayerNorm((self.config.network
.rnn_size,))

        self.prediction_transforms = torch.nn.ModuleList([
            torch.nn.Sequential(
                torch.nn.Linear(
                    self.config.network.rnn_size,
                    self.config.data._input_dim)
            )
            for steps in range(self.config.network.predictio
```

```
n_steps)
        ])

        self.to(self.device)

    def forward(self, signal):

        # signal = signal.permute(0, 2, 1)
        signal = self.input_norm(signal)
        # signal = signal.permute(0, 2, 1)

        output, state = self.rnn(signal)
        output = self.output_norm(output)

        losses = []
        predictions = []

        for step, affine in enumerate(self.prediction_transf
orms, start=1):

            shifted_output = output[:, :-step, :]
            shifted_signal = signal.detach()[:, step:, :]

            prediction = affine(shifted_output)
            predictions.append(prediction)

            loss = torch.abs(shifted_signal - prediction)
            loss = loss.sum(-1)
            loss = loss.mean()

            losses.append(loss)

        r = dict(
            losses=losses,
            output=output,
            predictions=predictions
        )

        return r

    def add_scalar_summaries(
        self, losses, writer, global_step):

        # scalars
        for k, loss in enumerate(losses, start=1):
            writer.add_scalar("loss_{k}".format(k=k), loss,
global_step)

    def add_image_summaries(
        self, signal, output, predictions, global_step, writ
er, to_plot=8):
```

```
        if len(signal) > to_plot:
            signal = signal[:to_plot]
            output = output[:to_plot]
            predictions = [p[:to_plot] for p in predictions]

        # signal
        image_grid = torchvision.utils.make_grid(
            signal.data.cpu().unsqueeze(1),
            normalize=True, scale_each=True
        )
        writer.add_image("signal", image_grid, global_step)
        # output
        image_grid = torchvision.utils.make_grid(
            output.data.cpu().unsqueeze(1),
            normalize=True, scale_each=True
        )
        writer.add_image("output", image_grid, global_step)

        for k, p in enumerate(predictions, start=1):
            image_grid = torchvision.utils.make_grid(
                p.data.cpu().unsqueeze(1),
                normalize=True, scale_each=True
            )
            writer.add_image(
                "prediction_{k}".format(k=k), image_grid, gl
obal_step)

    def add_projection_summary(self, image, global_step, wri
ter, name="projection"):
        writer.add_image(name, image.transpose(2, 0, 1), glo
bal_step)

    def train_epoch(self, train_loader,
                    epoch, log_interval, write_summary=True)
:

        self.train()

        print(
            "\n" + " " * 10 + "****** Epoch {epoch} ******\n
"
            .format(epoch=epoch)
        )

        history = deque(maxlen=30)

        self.optimizer.zero_grad()
        accumulated_loss = 0

        with tqdm(total=len(train_loader), ncols=80) as pb:

            for batch_idx, sample in enumerate(train_loader)
```

```
:
                self.global_step += 1

                make_step(self.scheduler, step=self.global_s
tep)

                signal, labels = (
                    sample["signal"].to(self.device),
                    sample["labels"].to(self.device)
                )

                outputs = self(signal)

                losses = outputs["losses"]

                loss = (
                    sum(losses)
                ) / self.config.train.accumulation_steps

                loss.backward()
                accumulated_loss += loss

                if batch_idx % self.config.train.accumulatio
n_steps == 0:
                    self.optimizer.step()
                    accumulated_loss = 0
                    self.optimizer.zero_grad()

                history.append(loss.item())

                pb.update()
                pb.set_description(
                    "Loss: {:.4f}".format(
                        np.mean(history)))

                if batch_idx % log_interval == 0:
                    self.add_scalar_summaries(
                        [loss.item() for loss in losses],
                        self.train_writer, self.global_step)

                if batch_idx == 0:
                    self.add_image_summaries(
                        signal,
                        outputs["output"],
                        outputs["predictions"],
                        self.global_step, self.train_writer)

    def evaluate(self, loader, verbose=False, write_summary=
False, epoch=None):

        self.eval()
```

```
        valid_losses = [0 for _ in range(self.config.network
.prediction_steps)]

        all_outputs = []
        all_labels = []

        with torch.no_grad():
            for batch_idx, sample in enumerate(loader):

                signal, labels = (
                    sample["signal"].to(self.device),
                    sample["labels"].to(self.device)
                )

                outputs = self(signal)

                losses = outputs["losses"]

                multiplier = len(signal) / len(loader.datase
t)

                for k, loss in enumerate(losses):
                    valid_losses[k] += loss.item() * multipl
ier

                all_outputs.extend(
                    outputs["output"].data.cpu().numpy())
                all_labels.extend(labels.data.cpu().numpy())

        valid_loss = sum(valid_losses)

        all_labels = np.array(all_labels)

        if write_summary:
            self.add_scalar_summaries(
                valid_losses,
                writer=self.valid_writer, global_step=self.g
lobal_step
            )
            if epoch % self.config.train._proj_interval == 0
:
                self.add_projection_summary(
                    plot_projection(
                        all_outputs, all_labels, frames_per_
example=5, newline=True),
                    writer=self.valid_writer, global_step=se
lf.global_step,
                    name="projection_output")

        if verbose:
            print("\nValidation loss: {:.4f}".format(valid_l
```

```
oss))

        return -valid_loss

    def validation(self, valid_loader, epoch):
        return self.evaluate(
            valid_loader,
            verbose=True, write_summary=True, epoch=epoch)

    def predict(self, loader):

        self.eval()

        all_class_probs = []

        with torch.no_grad():
            for sample in loader:

                signal = sample["signal"].to(self.device)

                outputs = self(signal)

                class_logits = outputs["class_logits"].squee
ze()

                class_probs = torch.sigmoid(class_logits).da
ta.cpu().numpy()
                all_class_probs.extend(class_probs)

        all_class_probs = np.asarray(all_class_probs)

        return all_class_probs

    def fit_validate(self, train_loader, valid_loader, epoch
s, fold,
                     log_interval=25):


        self.experiment.register_directory("summaries")
        self.train_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "train"
            )
        )
        self.valid_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "valid"
            )
```

```
        )

        os.makedirs(
            os.path.join(
                self.experiment.checkpoints,
                "fold_{}".format(fold)),
            exist_ok=True
        )

        self.global_step = 0
        self.make_optimizer(max_steps=len(train_loader) * ep
ochs)

        scores = []
        best_score = 0

        for epoch in range(epochs):

            make_step(self.scheduler, epoch=epoch)

            if epoch == self.config.train.switch_off_augment
ations_on:
                train_loader.dataset.transform.switch_off_au
gmentations()

            self.train_epoch(
                train_loader, epoch,
                log_interval, write_summary=True
            )
            validation_score = self.validation(valid_loader,
 epoch)
            scores.append(validation_score)

            if epoch % self.config.train._save_every == 0:
                print("\nSaving model on epoch", epoch)
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "model_on_epoch_{}.pth".format(epoch
)
                    )
                )

            if validation_score > best_score:
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "best_model.pth"
```

```
                    )
                )
                best_score = validation_score

        return scores

    def make_optimizer(self, max_steps):

        optimizer = OPTIMIZERS[self.config.train.optimizer]
        optimizer = optimizer(
            self.parameters(),
            self.config.train.learning_rate,
            weight_decay=self.config.train.weight_decay
        )
        self.optimizer = optimizer
        self.scheduler = make_scheduler(
            self.config.train.scheduler, max_steps=max_steps
)(optimizer)

    def load_best_model(self, fold):

        self.load_state_dict(
            torch.load(
                os.path.join(
                    self.experiment.checkpoints,
                    "fold_{}".format(fold),
                    "best_model.pth"
                )
            )
        )

======
import os
import math
import itertools
from collections import defaultdict, OrderedDict, deque

from tqdm import tqdm
import numpy as np
import torch
import torch.nn as nn
import torch.utils.model_zoo as model_zoo
import torchvision.utils
from tensorboardX import SummaryWriter
from pretrainedmodels.models import resnet18, resnet34


from ops.training import OPTIMIZERS, make_scheduler, make_st
ep
from networks.losses import binary_cross_entropy, focal_loss
, lsep_loss
from ops.utils import lwlrap, make_mel_filterbanks, is_mel,
```

```
is_stft, compute_torch_stft


class ConvLockedDropout(nn.Module):
    def __init__(self, dropout_rate=0.0):
        super().__init__()
        self.dropout_rate = dropout_rate

    def forward(self, x):
        if not self.training or not self.dropout_rate:
            return x

        n, s, t = x.size()

        m = torch.zeros(n, s, 1, device=x.device).bernoulli_
(1 - self.dropout_rate)
        m = m.expand_as(x)
        return m * x


class ResnetBlock(nn.Module):

    def __init__(self, depth):
        super().__init__()

        self.conv1 = nn.Conv1d(depth, depth, kernel_size=1)
        self.bn1 = nn.BatchNorm1d(depth)
        self.conv2 = nn.Conv1d(depth, depth, kernel_size=3,
padding=1)
        self.bn2 = nn.BatchNorm1d(depth)
        self.conv3 = nn.Conv1d(depth, depth, kernel_size=1)
        self.bn3 = nn.BatchNorm1d(depth)
        self.prelu1 = nn.PReLU(depth)
        self.prelu2 = nn.PReLU(depth)
        self.prelu3 = nn.PReLU(depth)

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.prelu1(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.prelu2(out)

        out = self.conv3(out)
        out = self.bn3(out)

        out += identity
        out = self.prelu3(out)
```

```
        return out


class ResnetBlock2d(nn.Module):

    def __init__(self, depth):
        super().__init__()

        self.conv1 = nn.Conv2d(depth, depth, kernel_size=1)
        self.bn1 = nn.BatchNorm2d(depth)
        self.conv2 = nn.Conv2d(depth, depth, kernel_size=3,
padding=1)
        self.bn2 = nn.BatchNorm2d(depth)
        self.conv3 = nn.Conv2d(depth, depth, kernel_size=1)
        self.bn3 = nn.BatchNorm2d(depth)
        self.prelu1 = nn.PReLU(depth)
        self.prelu2 = nn.PReLU(depth)
        self.prelu3 = nn.PReLU(depth)

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.prelu1(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.prelu2(out)

        out = self.conv3(out)
        out = self.bn3(out)

        out += identity
        out = self.prelu3(out)

        return out


class HierarchicalCNNClassificationModel(nn.Module):

    def __init__(self, experiment, device="cuda"):
        super().__init__()

        self.device = device

        self.experiment = experiment
        self.config = experiment.config

        if is_mel(self.config.data.features):
            self.filterbanks = torch.from_numpy(
```

```
                make_mel_filterbanks(self.config.data.featur
es)).to(self.device)

        self.conv_modules = torch.nn.ModuleList()
        self.rnns = torch.nn.ModuleList()

        total_depth = 0
        rnn_size = 128

        for k in range(self.config.network.num_conv_blocks):

            input_size = self.config.data._input_dim if not
k else depth
            depth = int(
                self.config.network.growth_rate ** k
                * self.config.network.conv_base_depth)

            if k >= self.config.network.start_deep_supervisi
on_on:
                if self.config.network.aggregation_type == "
max":
                    total_depth += depth
                elif self.config.network.aggregation_type ==
 "rnn":
                    total_depth += rnn_size * 2
                    self.rnns.append(
                        nn.Sequential(
                            nn.LayerNorm((depth,)),
                            nn.GRU(
                                depth, rnn_size, batch_first
=True, bidirectional=True)
                        )
                    )

            modules = [nn.BatchNorm1d(input_size)]
            modules.extend([
                nn.Conv1d(
                    input_size,
                    depth,
                    kernel_size=3,
                    padding=1
                ),
                nn.MaxPool1d(kernel_size=2, stride=2),
                nn.BatchNorm1d(depth),
                nn.PReLU(depth),
                ResnetBlock(depth)
            ])

            self.conv_modules.append(nn.Sequential(*modules)
)

        self.global_maxpool = nn.AdaptiveMaxPool1d(1)
```

```
        self.output_transform = nn.Sequential(
            nn.BatchNorm1d(total_depth),
            nn.Linear(total_depth, total_depth),
            nn.BatchNorm1d(total_depth),
            nn.PReLU(total_depth),
            nn.Dropout(p=self.config.network.output_dropout)
,
            nn.Linear(total_depth, self.config.data._n_class
es)
        )

        self.to(self.device)

    def forward(self, signal):

        if is_stft(self.config.data.features) or is_mel(self
.config.data.features):
            signal = compute_torch_stft(
                signal.squeeze(-1),
                self.config.data.features
            )

            if is_stft(self.config.data.features):
                signal = torch.log(signal + 1e-4)

        if is_mel(self.config.data.features):
            signal = nn.functional.conv1d(
                signal,
                self.filterbanks.unsqueeze(-1)
            )
            signal = torch.log(signal + 1e-4)

        features = []

        h = signal
        for k, module in enumerate(self.conv_modules):
            h = module(h)
            if k >= self.config.network.start_deep_supervisi
on_on:
                if self.config.network.aggregation_type == "
max":
                    features.append(self.global_maxpool(h).s
queeze(-1))
                elif self.config.network.aggregation_type ==
 "rnn":
                    rnn_input = h.permute(0, 2, 1)
                    outputs, state = self.rnns[
                        k - self.config.network.start_deep_s
upervision_on](rnn_input)
                    features.append(
                        state.permute(1, 0, 2).contiguous().
```

```
view(rnn_input.size(0), -1))

        features = torch.cat(features, -1)

        class_logits = self.output_transform(features)

        r = dict(
            class_logits=class_logits
        )

        return r

    def add_scalar_summaries(
        self, loss, metric, writer, global_step):

        # scalars
        writer.add_scalar("loss", loss, global_step)
        writer.add_scalar("metric", metric, global_step)

    def add_image_summaries(self, signal, global_step, write
r, to_plot=8):

        if len(signal) > to_plot:
            signal = signal[:to_plot]

        # image
        image_grid = torchvision.utils.make_grid(
            signal.data.cpu().unsqueeze(1),
            normalize=True, scale_each=True
        )
        writer.add_image("signal", image_grid, global_step)

    def train_epoch(self, train_loader,
                    epoch, log_interval, write_summary=True)
:

        self.train()

        print(
            "\n" + " " * 10 + "****** Epoch {epoch} ******\n
"
            .format(epoch=epoch)
        )

        history = deque(maxlen=30)

        self.optimizer.zero_grad()
        accumulated_loss = 0

        with tqdm(total=len(train_loader), ncols=80) as pb:

            for batch_idx, sample in enumerate(train_loader)
```

```
:
                self.global_step += 1

                make_step(self.scheduler, step=self.global_s
tep)

                signal, labels = (
                    sample["signal"].to(self.device),
                    sample["labels"].to(self.device).float()
                )

                outputs = self(signal)

                class_logits = outputs["class_logits"].squee
ze()

                loss = (
                    lsep_loss(
                        class_logits,
                        labels
                    )
                ) / self.config.train.accumulation_steps

                loss.backward()
                accumulated_loss += loss

                if batch_idx % self.config.train.accumulatio
n_steps == 0:
                    self.optimizer.step()
                    accumulated_loss = 0
                    self.optimizer.zero_grad()

                probs = torch.sigmoid(class_logits).data.cpu
().numpy()
                labels = labels.data.cpu().numpy()

                metric = lwlrap(labels, probs)
                history.append(metric)

                pb.update()
                pb.set_description(
                    "Loss: {:.4f}, Metric: {:.4f}".format(
                        loss.item(), np.mean(history)))

                if batch_idx % log_interval == 0:
                    self.add_scalar_summaries(
                        loss.item(), metric, self.train_writ
er, self.global_step)

                if batch_idx == 0:
                    self.add_image_summaries(
```

```
                        signal, self.global_step, self.train
_writer)

    def evaluate(self, loader, verbose=False, write_summary=
False, epoch=None):

        self.eval()

        valid_loss = 0

        all_class_probs = []
        all_labels = []

        with torch.no_grad():
            for batch_idx, sample in enumerate(loader):

                signal, labels = (
                    sample["signal"].to(self.device),
                    sample["labels"].to(self.device).float()
                )

                outputs = self(signal)

                class_logits = outputs["class_logits"].squee
ze()

                loss = (
                    lsep_loss(
                        class_logits,
                        labels,
                    )
                ).item()

                multiplier = len(labels) / len(loader.datase
t)

                valid_loss += loss * multiplier

                class_probs = torch.sigmoid(class_logits).da
ta.cpu().numpy()
                labels = labels.data.cpu().numpy()

                all_class_probs.extend(class_probs)
                all_labels.extend(labels)

            all_class_probs = np.asarray(all_class_probs)
            all_labels = np.asarray(all_labels)

            metric = lwlrap(all_labels, all_class_probs)

            if write_summary:
                self.add_scalar_summaries(
```

```
                    valid_loss,
                    metric,
                    writer=self.valid_writer, global_step=se
lf.global_step
                )

            if verbose:
                print("\nValidation loss: {:.4f}".format(val
id_loss))
                print("Validation metric: {:.4f}".format(met
ric))

            return metric

    def validation(self, valid_loader, epoch):
        return self.evaluate(
            valid_loader,
            verbose=True, write_summary=True, epoch=epoch)

    def predict(self, loader):

        self.eval()

        all_class_probs = []

        with torch.no_grad():
            for sample in loader:

                signal = sample["signal"].to(self.device)

                outputs = self(signal)

                class_logits = outputs["class_logits"].squee
ze()

                class_probs = torch.sigmoid(class_logits).da
ta.cpu().numpy()
                all_class_probs.extend(class_probs)

        all_class_probs = np.asarray(all_class_probs)

        return all_class_probs

    def fit_validate(self, train_loader, valid_loader, epoch
s, fold,
                     log_interval=25):


        self.experiment.register_directory("summaries")
        self.train_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
```

```
                "fold_{}".format(fold),
                "train"
            )
        )
        self.valid_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "valid"
            )
        )

        os.makedirs(
            os.path.join(
                self.experiment.checkpoints,
                "fold_{}".format(fold)),
            exist_ok=True
        )

        self.global_step = 0
        self.make_optimizer(max_steps=len(train_loader) * ep
ochs)

        scores = []
        best_score = 0

        for epoch in range(epochs):

            make_step(self.scheduler, epoch=epoch)

            if epoch == self.config.train.switch_off_augment
ations_on:
                train_loader.dataset.transform.switch_off_au
gmentations()

            self.train_epoch(
                train_loader, epoch,
                log_interval, write_summary=True
            )
            validation_score = self.validation(valid_loader,
 epoch)
            scores.append(validation_score)

            if epoch % self.config.train._save_every == 0:
                print("\nSaving model on epoch", epoch)
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "model_on_epoch_{}.pth".format(epoch
)
```

```
                )
            )

            if validation_score > best_score:
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "best_model.pth"
                    )
                )
                best_score = validation_score

        return scores

    def make_optimizer(self, max_steps):

        optimizer = OPTIMIZERS[self.config.train.optimizer]
        optimizer = optimizer(
            self.parameters(),
            self.config.train.learning_rate,
            weight_decay=self.config.train.weight_decay
        )
        self.optimizer = optimizer
        self.scheduler = make_scheduler(
            self.config.train.scheduler, max_steps=max_steps
)(optimizer)

    def load_best_model(self, fold):

        self.load_state_dict(
            torch.load(
                os.path.join(
                    self.experiment.checkpoints,
                    "fold_{}".format(fold),
                    "best_model.pth"
                )
            )
        )


class TwoDimensionalCNNClassificationModel(nn.Module):

    def __init__(self, experiment, device="cuda"):
        super().__init__()

        self.device = device

        self.experiment = experiment
        self.config = experiment.config
```

```
        if is_mel(self.config.data.features):
            self.filterbanks = torch.from_numpy(
                make_mel_filterbanks(self.config.data.featur
es)).to(self.device)

        self.conv_modules = torch.nn.ModuleList()
        self.rnns = torch.nn.ModuleList()

        total_depth = 0

        for k in range(self.config.network.num_conv_blocks):

            input_size = 2 if not k else depth
            depth = int(
                self.config.network.growth_rate ** k
                * self.config.network.conv_base_depth)

            rnn_size = 128

            if k >= self.config.network.start_deep_supervisi
on_on:
                if self.config.network.aggregation_type == "
max":
                    total_depth += depth
                elif self.config.network.aggregation_type ==
 "rnn":
                    total_depth += rnn_size * 2
                    self.rnns.append(
                        nn.Sequential(
                            nn.LayerNorm((depth,)),
                            nn.GRU(
                                depth, rnn_size, batch_first
=True, bidirectional=True)
                        )
                    )

            modules = [nn.BatchNorm2d(input_size)]
            modules.extend([
                nn.Conv2d(
                    input_size,
                    depth,
                    kernel_size=3,
                    padding=1
                ),
                nn.MaxPool2d(kernel_size=2, stride=2),
                nn.BatchNorm2d(depth),
                nn.PReLU(depth),
                ResnetBlock2d(depth)
            ])

            self.conv_modules.append(nn.Sequential(*modules)
)
```

```
        self.global_maxpool = nn.AdaptiveMaxPool2d(1)

        self.output_transform = nn.Sequential(
            nn.BatchNorm1d(total_depth),
            nn.Linear(total_depth, total_depth),
            nn.BatchNorm1d(total_depth),
            nn.PReLU(total_depth),
            nn.Dropout(p=self.config.network.output_dropout)
,
            nn.Linear(total_depth, self.config.data._n_class
es)
        )

        self.to(self.device)

    def _add_frequency_encoding(self, x):
        n, d, h, w = x.size()

        vertical = torch.linspace(-1, 1, h, device=x.device)
.view(1, 1, -1, 1)
        vertical = vertical.repeat(n, 1, 1, w)

        x = torch.cat([x, vertical], dim=1)

        return x

    def forward(self, signal):

        if is_stft(self.config.data.features) or is_mel(self
.config.data.features):
            signal = compute_torch_stft(
                signal.squeeze(-1),
                self.config.data.features
            )

            if is_stft(self.config.data.features):
                signal = torch.log(signal + 1e-4)

        if is_mel(self.config.data.features):
            signal = nn.functional.conv1d(
                signal,
                self.filterbanks.unsqueeze(-1)
            )
            signal = torch.log(signal + 1e-4)

        signal = signal.unsqueeze(1)
        signal = self._add_frequency_encoding(signal)

        features = []

        h = signal
```

```
        for k, module in enumerate(self.conv_modules):
            h = module(h)
            if k >= self.config.network.start_deep_supervisi
on_on:
                if self.config.network.aggregation_type == "
max":
                    features.append(self.global_maxpool(h).s
queeze(-1).squeeze(-1))
                elif self.config.network.aggregation_type ==
 "rnn":
                    rnn_input = torch.mean(h, 2).permute(0,
2, 1)
                    outputs, state = self.rnns[
                        k - self.config.network.start_deep_s
upervision_on](rnn_input)
                    features.append(
                        state.permute(1, 0, 2).contiguous().
view(rnn_input.size(0), -1))

        features = torch.cat(features, -1)

        class_logits = self.output_transform(features)

        r = dict(
            class_logits=class_logits
        )

        return r

    def add_scalar_summaries(
        self, loss, metric, writer, global_step):

        # scalars
        writer.add_scalar("loss", loss, global_step)
        writer.add_scalar("metric", metric, global_step)

    def add_histogram_summaries(
        self, losses, writer, global_step):

        writer.add_histogram("losses", np.array(losses), glo
bal_step=global_step)

    def add_image_summaries(self, signal, global_step, write
r, to_plot=8):

        if len(signal) > to_plot:
            signal = signal[:to_plot]

        # image
        image_grid = torchvision.utils.make_grid(
            signal.data.cpu().unsqueeze(1),
            normalize=True, scale_each=True
```

```
        )
        writer.add_image("signal", image_grid, global_step)

    def train_epoch(self, train_loader,
                    epoch, log_interval, write_summary=True)
:

        self.train()

        print(
            "\n" + " " * 10 + "****** Epoch {epoch} ******\n
"
            .format(epoch=epoch)
        )

        training_losses = []

        history = deque(maxlen=30)

        self.optimizer.zero_grad()
        accumulated_loss = 0

        with tqdm(total=len(train_loader), ncols=80) as pb:

            for batch_idx, sample in enumerate(train_loader)
:

                self.global_step += 1

                make_step(self.scheduler, step=self.global_s
tep)

                signal, labels, is_noisy = (
                    sample["signal"].to(self.device),
                    sample["labels"].to(self.device).float()
,
                    sample["is_noisy"].to(self.device).float
()
                )

                outputs = self(signal)

                class_logits = outputs["class_logits"]

                loss = (
                    lsep_loss(
                        class_logits,
                        labels,
                        average=False
                    )
                ) / self.config.train.accumulation_steps
```

```
                training_losses.extend(loss.data.cpu().numpy
())
                loss = loss.mean()

                loss.backward()
                accumulated_loss += loss

                if batch_idx % self.config.train.accumulatio
n_steps == 0:
                    self.optimizer.step()
                    accumulated_loss = 0
                    self.optimizer.zero_grad()

                probs = torch.sigmoid(class_logits).data.cpu
().numpy()
                labels = labels.data.cpu().numpy()

                metric = lwlrap(labels, probs)
                history.append(metric)

                pb.update()
                pb.set_description(
                    "Loss: {:.4f}, Metric: {:.4f}".format(
                        loss.item(), np.mean(history)))

                if batch_idx % log_interval == 0:
                    self.add_scalar_summaries(
                        loss.item(), metric, self.train_writ
er, self.global_step)

                if batch_idx == 0:
                    self.add_image_summaries(
                        signal, self.global_step, self.train
_writer)

        self.add_histogram_summaries(
            training_losses, self.train_writer, self.global_
step)

    def evaluate(self, loader, verbose=False, write_summary=
False, epoch=None):

        self.eval()

        valid_loss = 0

        all_class_probs = []
        all_labels = []

        with torch.no_grad():
            for batch_idx, sample in enumerate(loader):
```

```
                signal, labels = (
                    sample["signal"].to(self.device),
                    sample["labels"].to(self.device).float()
                )

                outputs = self(signal)

                class_logits = outputs["class_logits"]

                loss = (
                    lsep_loss(
                        class_logits,
                        labels,
                    )
                ).item()

                multiplier = len(labels) / len(loader.datase
t)

                valid_loss += loss * multiplier

                class_probs = torch.sigmoid(class_logits).da
ta.cpu().numpy()
                labels = labels.data.cpu().numpy()

                all_class_probs.extend(class_probs)
                all_labels.extend(labels)

            all_class_probs = np.asarray(all_class_probs)
            all_labels = np.asarray(all_labels)

            metric = lwlrap(all_labels, all_class_probs)

            if write_summary:
                self.add_scalar_summaries(
                    valid_loss,
                    metric,
                    writer=self.valid_writer, global_step=se
lf.global_step
                )

            if verbose:
                print("\nValidation loss: {:.4f}".format(val
id_loss))
                print("Validation metric: {:.4f}".format(met
ric))

            return metric

    def validation(self, valid_loader, epoch):
        return self.evaluate(
            valid_loader,
```

```
            verbose=True, write_summary=True, epoch=epoch)

    def predict(self, loader, n_tta=1):

        self.eval()

        all_class_probs = []

        for k in range(n_tta):

            tta_probs = []

            with torch.no_grad():
                for sample in loader:

                    signal = sample["signal"].to(self.device
)

                    outputs = self(signal)

                    class_logits = outputs["class_logits"]

                    class_probs = torch.sigmoid(class_logits
).data.cpu().numpy()
                    tta_probs.extend(class_probs)

            tta_probs = np.array(tta_probs)
            all_class_probs.append(tta_probs)

        all_class_probs = np.mean(all_class_probs, 0)

        return all_class_probs

    def fit_validate(self, train_loader, valid_loader, epoch
s, fold,
                     log_interval=25):


        self.experiment.register_directory("summaries")
        self.train_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "train"
            )
        )
        self.valid_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "valid"
            )
```

```
        )

        os.makedirs(
            os.path.join(
                self.experiment.checkpoints,
                "fold_{}".format(fold)),
            exist_ok=True
        )

        self.global_step = 0
        self.make_optimizer(max_steps=len(train_loader) * ep
ochs)

        scores = []
        best_score = 0

        for epoch in range(epochs):

            make_step(self.scheduler, epoch=epoch)

            if epoch == self.config.train.switch_off_augment
ations_on:
                train_loader.dataset.transform.switch_off_au
gmentations()

            self.train_epoch(
                train_loader, epoch,
                log_interval, write_summary=True
            )
            validation_score = self.validation(valid_loader,
 epoch)
            scores.append(validation_score)

            if epoch % self.config.train._save_every == 0:
                print("\nSaving model on epoch", epoch)
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "model_on_epoch_{}.pth".format(epoch
)
                    )
                )

            if validation_score > best_score:
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "best_model.pth"
```

```
                    )
                )
                best_score = validation_score

        return scores

    def make_optimizer(self, max_steps):

        optimizer = OPTIMIZERS[self.config.train.optimizer]
        optimizer = optimizer(
            self.parameters(),
            self.config.train.learning_rate,
            weight_decay=self.config.train.weight_decay
        )
        self.optimizer = optimizer
        self.scheduler = make_scheduler(
            self.config.train.scheduler, max_steps=max_steps
)(optimizer)

    def load_best_model(self, fold):

        self.load_state_dict(
            torch.load(
                os.path.join(
                    self.experiment.checkpoints,
                    "fold_{}".format(fold),
                    "best_model.pth"
                )
            )
        )



class CNNBackboneClassificationModel(nn.Module):

    def __init__(self, experiment, device="cuda"):
        super().__init__()

        self.device = device

        self.experiment = experiment
        self.config = experiment.config

        if is_mel(self.config.data.features):
            self.filterbanks = torch.from_numpy(
                make_mel_filterbanks(self.config.data.featur
es)).to(self.device)

        self.input_norm = nn.BatchNorm2d(3)

        if self.config.network.backbone == "resnet18":
            self.backbone = resnet18(pretrained=None)
```

```
        elif self.config.network.backbone == "resnet34":
            self.backbone = resnet34(pretrained=None)

        self.global_maxpool = nn.AdaptiveMaxPool2d(1)

        total_depth = self.backbone.last_linear.in_features

        self.output_transform = nn.Sequential(
            nn.BatchNorm1d(total_depth),
            nn.Linear(total_depth, total_depth),
            nn.BatchNorm1d(total_depth),
            nn.PReLU(total_depth),
            nn.Dropout(p=self.config.network.output_dropout)
,
            nn.Linear(total_depth, self.config.data._n_class
es)
        )

        self.to(self.device)

    def forward(self, signal):

        if is_stft(self.config.data.features) or is_mel(self
.config.data.features):
            signal = compute_torch_stft(
                signal.squeeze(-1),
                self.config.data.features
            )

            if is_stft(self.config.data.features):
                signal = torch.log(signal + 1e-4)

        if is_mel(self.config.data.features):
            signal = nn.functional.conv1d(
                signal,
                self.filterbanks.unsqueeze(-1)
            )
            signal = torch.log(signal + 1e-4)

        signal = signal.unsqueeze(1)
        signal = signal.repeat(1, 3, 1, 1)
        signal = self.input_norm(signal)

        h = self.backbone.features(signal)

        features = self.global_maxpool(h).squeeze(-1).squeez
e(-1)

        class_logits = self.output_transform(features)

        r = dict(
            class_logits=class_logits
```

```
        )

        return r

    def add_scalar_summaries(
        self, loss, metric, writer, global_step):

        # scalars
        writer.add_scalar("loss", loss, global_step)
        writer.add_scalar("metric", metric, global_step)

    def add_histogram_summaries(
        self, losses, writer, global_step):

        writer.add_histogram("losses", np.array(losses), glo
bal_step=global_step)

    def add_image_summaries(self, signal, global_step, write
r, to_plot=8):

        if len(signal) > to_plot:
            signal = signal[:to_plot]

        # image
        image_grid = torchvision.utils.make_grid(
            signal.data.cpu().unsqueeze(1),
            normalize=True, scale_each=True
        )
        writer.add_image("signal", image_grid, global_step)

    def train_epoch(self, train_loader,
                    epoch, log_interval, write_summary=True)
:

        self.train()

        print(
            "\n" + " " * 10 + "****** Epoch {epoch} ******\n
"
            .format(epoch=epoch)
        )

        training_losses = []

        history = deque(maxlen=30)

        self.optimizer.zero_grad()
        accumulated_loss = 0

        with tqdm(total=len(train_loader), ncols=80) as pb:

            for batch_idx, sample in enumerate(train_loader)
```

```
:
                    self.global_step += 1

                    make_step(self.scheduler, step=self.global_s
tep)

                    signal, labels, is_noisy = (
                        sample["signal"].to(self.device),
                        sample["labels"].to(self.device).float()
,
                        sample["is_noisy"].to(self.device).float
()
                    )

                    outputs = self(signal)

                    class_logits = outputs["class_logits"]

                    loss = (
                        lsep_loss(
                            class_logits,
                            labels,
                            average=False
                        )
                    ) / self.config.train.accumulation_steps

                    training_losses.extend(loss.data.cpu().numpy
())
                    loss = loss.mean()

                    loss.backward()
                    accumulated_loss += loss

                    if batch_idx % self.config.train.accumulatio
n_steps == 0:
                        self.optimizer.step()
                        accumulated_loss = 0
                        self.optimizer.zero_grad()

                    probs = torch.sigmoid(class_logits).data.cpu
().numpy()
                    labels = labels.data.cpu().numpy()

                    metric = lwlrap(labels, probs)
                    history.append(metric)

                    pb.update()
                    pb.set_description(
                        "Loss: {:.4f}, Metric: {:.4f}".format(
                            loss.item(), np.mean(history)))
```

```
                if batch_idx % log_interval == 0:
                    self.add_scalar_summaries(
                        loss.item(), metric, self.train_writ
er, self.global_step)

                if batch_idx == 0:
                    self.add_image_summaries(
                        signal, self.global_step, self.train
_writer)

        self.add_histogram_summaries(
            training_losses, self.train_writer, self.global_
step)

    def evaluate(self, loader, verbose=False, write_summary=
False, epoch=None):

        self.eval()

        valid_loss = 0

        all_class_probs = []
        all_labels = []

        with torch.no_grad():
            for batch_idx, sample in enumerate(loader):

                signal, labels = (
                    sample["signal"].to(self.device),
                    sample["labels"].to(self.device).float()
                )

                outputs = self(signal)

                class_logits = outputs["class_logits"]

                loss = (
                    lsep_loss(
                        class_logits,
                        labels,
                    )
                ).item()

                multiplier = len(labels) / len(loader.datase
t)

                valid_loss += loss * multiplier

                class_probs = torch.sigmoid(class_logits).da
ta.cpu().numpy()
                labels = labels.data.cpu().numpy()
```

```
                all_class_probs.extend(class_probs)
                all_labels.extend(labels)

            all_class_probs = np.asarray(all_class_probs)
            all_labels = np.asarray(all_labels)

            metric = lwlrap(all_labels, all_class_probs)

            if write_summary:
                self.add_scalar_summaries(
                    valid_loss,
                    metric,
                    writer=self.valid_writer, global_step=se
lf.global_step
                )

            if verbose:
                print("\nValidation loss: {:.4f}".format(val
id_loss))
                print("Validation metric: {:.4f}".format(met
ric))

            return metric

    def validation(self, valid_loader, epoch):
        return self.evaluate(
            valid_loader,
            verbose=True, write_summary=True, epoch=epoch)

    def predict(self, loader, n_tta=1):

        self.eval()

        all_class_probs = []

        for k in range(n_tta):

            tta_probs = []

            with torch.no_grad():
                for sample in loader:

                    signal = sample["signal"].to(self.device
)

                    outputs = self(signal)

                    class_logits = outputs["class_logits"]

                    class_probs = torch.sigmoid(class_logits
).data.cpu().numpy()
                    tta_probs.extend(class_probs)
```

```
            tta_probs = np.array(tta_probs)
            all_class_probs.append(tta_probs)

        all_class_probs = np.mean(all_class_probs, 0)

        return all_class_probs

    def fit_validate(self, train_loader, valid_loader, epoch
s, fold,
                     log_interval=25):


        self.experiment.register_directory("summaries")
        self.train_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "train"
            )
        )
        self.valid_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "valid"
            )
        )

        os.makedirs(
            os.path.join(
                self.experiment.checkpoints,
                "fold_{}".format(fold)),
            exist_ok=True
        )

        self.global_step = 0
        self.make_optimizer(max_steps=len(train_loader) * ep
ochs)

        scores = []
        best_score = 0

        for epoch in range(epochs):

            make_step(self.scheduler, epoch=epoch)

            if epoch == self.config.train.switch_off_augment
ations_on:
                train_loader.dataset.transform.switch_off_au
gmentations()
```

```
            self.train_epoch(
                train_loader, epoch,
                log_interval, write_summary=True
            )
            validation_score = self.validation(valid_loader,
 epoch)
            scores.append(validation_score)

            if epoch % self.config.train._save_every == 0:
                print("\nSaving model on epoch", epoch)
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "model_on_epoch_{}.pth".format(epoch
)
                    )
                )

            if validation_score > best_score:
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "best_model.pth"
                    )
                )
                best_score = validation_score

        return scores

    def make_optimizer(self, max_steps):

        optimizer = OPTIMIZERS[self.config.train.optimizer]
        optimizer = optimizer(
            self.parameters(),
            self.config.train.learning_rate,
            weight_decay=self.config.train.weight_decay
        )
        self.optimizer = optimizer
        self.scheduler = make_scheduler(
            self.config.train.scheduler, max_steps=max_steps
)(optimizer)

    def load_best_model(self, fold):

        self.load_state_dict(
            torch.load(
                os.path.join(
                    self.experiment.checkpoints,
```

```
                    "fold_{}".format(fold),
                    "best_model.pth"
                )
            )
        )
======
import os
import math
import itertools
from collections import defaultdict, OrderedDict, deque

from tqdm import tqdm
import numpy as np
import torch
import torch.nn as nn
import torchvision.utils
from tensorboardX import SummaryWriter
from torch.nn.functional import binary_cross_entropy_with_lo
gits


from ops.training import OPTIMIZERS, make_scheduler, make_st
ep
from networks.losses import binary_cross_entropy, focal_loss
, lsep_loss
from ops.utils import plot_projection


class CausalConv1d(nn.Module):

    def __init__(self, in_channels, out_channels, kernel_siz
e, stride=1):
        super().__init__()
        self.kernel_size = kernel_size
        self.conv = nn.Conv1d(
            in_channels, out_channels, kernel_size,
            stride=stride, padding=kernel_size)

    def forward(self, x):
        x = self.conv(x)
        return x[:, :, :-self.kernel_size]


class CPCModel(nn.Module):

    def __init__(self, experiment, device="cuda"):
        super().__init__()

        self.device = device

        self.experiment = experiment
        self.config = experiment.config
```

```
        encoder_layers = []

        for k in range(self.config.network.n_encoder_layers)
:
            input_size = self.config.data._input_dim if not
k else depth
            depth = int(
                self.config.network.growth_rate ** k
                * self.config.network.conv_base_depth)
            modules = [nn.BatchNorm1d(input_size)] if not k
else []
            modules.extend([
                CausalConv1d(
                    input_size,
                    depth,
                    kernel_size=3,
                    stride=2
                ),
                nn.PReLU(depth)
            ])
            encoder_layers.extend(modules)

        encoder_layers.append(nn.BatchNorm1d(depth))

        self.encoder = nn.Sequential(*encoder_layers)

        self.context_network = nn.GRU(
            depth, self.config.network.context_size,
            num_layers=1,
            batch_first=True
        )

        self.coupling_transforms = torch.nn.ModuleList([
            torch.nn.Sequential(
                torch.nn.Conv1d(
                    self.config.network.context_size, depth,
 kernel_size=1)
            )
            for steps in range(self.config.network.predictio
n_steps)
        ])

        self.to(self.device)

    def forward(self, signal):

        signal = signal.permute(0, 2, 1)

        # z is (n, depth, steps)
        z = self.encoder(signal)
        # c is (n, context_size, steps)
```

```
        c, state = self.context_network(z.permute(0, 2, 1))
        c = c.permute(0, 2, 1)

        losses = []

        for step, affine in enumerate(self.coupling_transfor
ms, start=1):

            a = affine(c)

            # logits is (n, steps, steps)
            logits = torch.bmm(z.permute(0, 2, 1), a)

            labels = torch.eye(logits.size(2) - step, device
=z.device)
            labels = torch.nn.functional.pad(labels, (0, ste
p, step, 0))
            labels = labels.unsqueeze(0).expand_as(logits)

            loss = binary_cross_entropy_with_logits(logits,
labels)
            losses.append(loss)

        r = dict(
            losses=losses,
            z=z,
            c=c
        )

        return r

    def add_scalar_summaries(
        self, losses, writer, global_step):

        # scalars
        for k, loss in enumerate(losses, start=1):
            writer.add_scalar("loss_{k}".format(k=k), loss,
global_step)

    def add_image_summaries(self, signal, c, z, global_step,
 writer, to_plot=8):

        if len(c) > to_plot:
            signal = signal[:to_plot]
            c = c[:to_plot]
            z = z[:to_plot]

        # signal
        image_grid = torchvision.utils.make_grid(
            signal.data.cpu().unsqueeze(1),
            normalize=True, scale_each=True
        )
```

```
        writer.add_image("signal", image_grid, global_step)
        # z
        image_grid = torchvision.utils.make_grid(
            z.data.cpu().unsqueeze(1),
            normalize=True, scale_each=True
        )
        writer.add_image("z", image_grid, global_step)
        # c
        image_grid = torchvision.utils.make_grid(
            c.data.cpu().unsqueeze(1),
            normalize=True, scale_each=True
        )
        writer.add_image("c", image_grid, global_step)

    def add_projection_summary(self, image, global_step, wri
ter, name="projection"):
        writer.add_image(name, image.transpose(2, 0, 1), glo
bal_step)

    def train_epoch(self, train_loader,
                    epoch, log_interval, write_summary=True)
:

        self.train()

        print(
            "\n" + " " * 10 + "****** Epoch {epoch} ******\n
"
            .format(epoch=epoch)
        )

        history = deque(maxlen=30)

        self.optimizer.zero_grad()
        accumulated_loss = 0

        with tqdm(total=len(train_loader), ncols=80) as pb:

            for batch_idx, sample in enumerate(train_loader)
:

                self.global_step += 1

                make_step(self.scheduler, step=self.global_s
tep)

                signal, labels = (
                    sample["signal"].to(self.device),
                    sample["labels"].to(self.device)
                )

                outputs = self(signal)
```

```
                losses = outputs["losses"]

                loss = (
                    sum(losses)
                ) / self.config.train.accumulation_steps

                loss.backward()
                accumulated_loss += loss

                if batch_idx % self.config.train.accumulatio
n_steps == 0:
                    self.optimizer.step()
                    accumulated_loss = 0
                    self.optimizer.zero_grad()

                history.append(loss.item())

                pb.update()
                pb.set_description(
                    "Loss: {:.4f}".format(
                        np.mean(history)))

                if batch_idx % log_interval == 0:
                    self.add_scalar_summaries(
                        [loss.item() for loss in losses],
                        self.train_writer, self.global_step)

                if batch_idx == 0:
                    self.add_image_summaries(
                        signal,
                        outputs["c"].permute(0, 2, 1),
                        outputs["z"].permute(0, 2, 1),
                        self.global_step, self.train_writer)

    def evaluate(self, loader, verbose=False, write_summary=
False, epoch=None):

        self.eval()

        valid_losses = [0 for _ in range(self.config.network
.prediction_steps)]

        all_c = []
        all_z = []
        all_labels = []

        with torch.no_grad():
            for batch_idx, sample in enumerate(loader):

                signal, labels = (
                    sample["signal"].to(self.device),
```

```
                    sample["labels"].to(self.device)
                )

                outputs = self(signal)

                losses = outputs["losses"]

                multiplier = len(signal) / len(loader.datase
t)

                for k, loss in enumerate(losses):
                    valid_losses[k] += loss.item() * multipl
ier

                all_c.extend(
                    outputs["c"].permute(0, 2, 1).data.cpu()
.numpy())
                all_z.extend(
                    outputs["z"].permute(0, 2, 1).data.cpu()
.numpy())
                all_labels.extend(labels.data.cpu().numpy())

        valid_loss = sum(valid_losses)

        all_labels = np.array(all_labels)

        if write_summary:
            self.add_scalar_summaries(
                valid_losses,
                writer=self.valid_writer, global_step=self.g
lobal_step
            )
            if epoch % self.config.train._proj_interval == 0
:
                self.add_projection_summary(
                    plot_projection(
                        all_c, all_labels, frames_per_exampl
e=5, newline=True),
                    writer=self.valid_writer, global_step=se
lf.global_step,
                    name="projection_c")
                self.add_projection_summary(
                    plot_projection(all_z, all_labels, frame
s_per_example=5),
                    writer=self.valid_writer, global_step=se
lf.global_step,
                    name="projection_z")

        if verbose:
            print("\nValidation loss: {:.4f}".format(valid_l
oss))
```

```
        return -valid_loss

    def validation(self, valid_loader, epoch):
        return self.evaluate(
            valid_loader,
            verbose=True, write_summary=True, epoch=epoch)

    def predict(self, loader):

        self.eval()

        all_class_probs = []

        with torch.no_grad():
            for sample in loader:

                signal = sample["signal"].to(self.device)

                outputs = self(signal)

                class_logits = outputs["class_logits"].squee
ze()

                class_probs = torch.sigmoid(class_logits).da
ta.cpu().numpy()
                all_class_probs.extend(class_probs)

        all_class_probs = np.asarray(all_class_probs)

        return all_class_probs

    def fit_validate(self, train_loader, valid_loader, epoch
s, fold,
                     log_interval=25):


        self.experiment.register_directory("summaries")
        self.train_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "train"
            )
        )
        self.valid_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "valid"
            )
        )
```

```
        os.makedirs(
            os.path.join(
                self.experiment.checkpoints,
                "fold_{}".format(fold)),
            exist_ok=True
        )

        self.global_step = 0
        self.make_optimizer(max_steps=len(train_loader) * ep
ochs)

        scores = []
        best_score = 0

        for epoch in range(epochs):

            make_step(self.scheduler, epoch=epoch)

            if epoch == self.config.train.switch_off_augment
ations_on:
                train_loader.dataset.transform.switch_off_au
gmentations()

            self.train_epoch(
                train_loader, epoch,
                log_interval, write_summary=True
            )
            validation_score = self.validation(valid_loader,
 epoch)
            scores.append(validation_score)

            if epoch % self.config.train._save_every == 0:
                print("\nSaving model on epoch", epoch)
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "model_on_epoch_{}.pth".format(epoch
)
                    )
                )

            if validation_score > best_score:
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "best_model.pth"
                    )
                )
```

```
                best_score = validation_score

        return scores

    def make_optimizer(self, max_steps):

        optimizer = OPTIMIZERS[self.config.train.optimizer]
        optimizer = optimizer(
            self.parameters(),
            self.config.train.learning_rate,
            weight_decay=self.config.train.weight_decay
        )
        self.optimizer = optimizer
        self.scheduler = make_scheduler(
            self.config.train.scheduler, max_steps=max_steps
)(optimizer)

    def load_best_model(self, fold):

        self.load_state_dict(
            torch.load(
                os.path.join(
                    self.experiment.checkpoints,
                    "fold_{}".format(fold),
                    "best_model.pth"
                )
            )
        )

======
import torch
import torch.nn.functional as F


def focal_loss(input, target, focus=2.0, raw=True):

    if raw:
        input = torch.sigmoid(input)

    eps = 1e-7

    prob_true = input * target + (1 - input) * (1 - target)
    prob_true = torch.clamp(prob_true, eps, 1-eps)
    modulating_factor = (1.0 - prob_true).pow(focus)

    return (-modulating_factor * prob_true.log()).mean()


def binary_cross_entropy(input, target, raw=True):
    if raw:
        input = torch.sigmoid(input)
    return torch.nn.functional.binary_cross_entropy(input, t
```

```
arget)


def lsep_loss_stable(input, target, average=True):

    n = input.size(0)

    differences = input.unsqueeze(1) - input.unsqueeze(2)
    where_lower = (target.unsqueeze(1) < target.unsqueeze(2)
).float()

    differences = differences.view(n, -1)
    where_lower = where_lower.view(n, -1)

    max_difference, index = torch.max(differences, dim=1, ke
epdim=True)
    differences = differences - max_difference
    exps = differences.exp() * where_lower

    lsep = max_difference + torch.log(torch.exp(-max_differe
nce) + exps.sum(-1))

    if average:
        return lsep.mean()
    else:
        return lsep


def lsep_loss(input, target, average=True):

    differences = input.unsqueeze(1) - input.unsqueeze(2)
    where_different = (target.unsqueeze(1) < target.unsqueez
e(2)).float()

    exps = differences.exp() * where_different
    lsep = torch.log(1 + exps.sum(2).sum(1))

    if average:
        return lsep.mean()
    else:
        return lsep======
import random

import numpy as np
import librosa
import scipy.signal

from sklearn.utils import gen_even_slices


def compute_stft(audio, window_size, hop_size, log=True, eps
=1e-4):
```

```
    f, t, s = scipy.signal.stft(
        audio, nperseg=window_size, noverlap=hop_size)

    s = np.abs(s)

    if log:
        s = np.log(s + eps)

    return s


def trim_audio(audio):
    audio, interval = librosa.effects.trim(audio, top_db=60)
    return audio


def read_audio(file):
    audio, sr = librosa.load(file, sr=None)
    return audio, sr


def mix_audio_and_labels(first_audio, second_audio, first_la
bels, second_labels):

    new_labels = np.clip(first_labels + second_labels, 0, 1)

    a = np.random.uniform(0.4, 0.6)

    shorter, longer = first_audio, second_audio

    if shorter.size == longer.size:
        return (shorter + longer) / 2, new_labels

    if first_audio.size > second_audio.size:
        shorter, longer = longer, shorter

    start = random.randint(0, longer.size - 1 - shorter.size
)
    end = start + shorter.size

    longer *= a
    longer[start:end] =+ shorter * (1 - a)

    return longer, new_labels


def shuffle_audio(audio, chunk_length=0.5, sr=None):

    n_chunks = int((audio.size / sr) / chunk_length)

    if n_chunks in (0, 1):
        return audio
```

```
    slices = list(gen_even_slices(audio.size, n_chunks))
    random.shuffle(slices)

    shuffled = np.concatenate([audio[s] for s in slices])

    return shuffled


def cutout(audio, area=0.25):

    area = int(audio.size * area)

    start = random.randrange(audio.size)
    end = start + area

    audio[start:end] = 0

    return audio
======
from sklearn.model_selection import KFold
from iterstrat.ml_stratifiers import MultilabelStratifiedKFo
ld
import numpy as np


def train_validation_data(ids, labels, n_folds, seed):

    for train, valid in KFold(
        n_folds, shuffle=True, random_state=seed).split(ids,
 labels):
        yield train, valid


def train_validation_data_stratified(
    ids, labels, classmap, n_folds, seed):

    binary_labels = np.zeros(
        (len(labels), len(classmap)), dtype=np.float32)
    for k, item in enumerate(labels.values):
        for label in item.split(","):
            binary_labels[k, classmap[label]] = 1

    for train, valid in MultilabelStratifiedKFold(
        n_folds, shuffle=True, random_state=seed).split(ids,
 binary_labels):
        yield train, valid======
import random
from copy import deepcopy

import numpy as np
from torch.utils.data.dataloader import default_collate
```

```
def make_collate_fn(padding_values):

    def _collate_fn(batch):

        for name, padding_value in padding_values.items():

            lengths = [len(sample[name]) for sample in batch
]
            max_length = max(lengths)

            for n, size in enumerate(lengths):
                p = max_length - size
                if p:
                    pad_width = [(0, p)] + [(0, 0)] * (batch
[n][name].ndim - 1)
                    if padding_value == "edge":
                        batch[n][name] = np.pad(
                            batch[n][name], pad_width,
                            mode="edge")
                    else:
                        batch[n][name] = np.pad(
                            batch[n][name], pad_width,
                            mode="constant", constant_values
=padding_value)

        return default_collate(batch)

    return _collate_fn



class BucketingSampler:

    def __init__(self, dataset, max_batch_elems, buckets):

        self.buckets = buckets
        self.dataset = dataset
        self.max_batch_elems = max_batch_elems

        self._create_batches()

    def _create_batches(self):

        self.n_bins = len(self.buckets)
        binned_sizes = np.digitize(self.dataset.lengths, sel
f.buckets)

        batches = []

        for bin_idx in range(1, self.n_bins):
```

```
            ids = np.nonzero(binned_sizes == bin_idx)[0]
            random.shuffle(ids)

            current_len = 0
            batch = []

            for id in ids:
                if current_len < self.max_batch_elems:
                    batch.append(id)
                    current_len += self.dataset.lengths[id]
                else:
                    batches.append(batch)
                    current_len = self.dataset.lengths[id]
                    batch = [id]

            if batch:
                batches.append(batch)

        random.shuffle(batches)

        self.n_batches = len(batches)
        self.batches = batches

    def __iter__(self):
        return iter(self.batches)

    def __len__(self):
        return self.n_batches======
from functools import partial

import numpy as np
import torch
from torch.optim import Optimizer
from torch.optim.lr_scheduler import StepLR, CosineAnnealing
LR, _LRScheduler


OPTIMIZERS = {
    "adam": partial(torch.optim.Adam, amsgrad=True),
    "momentum": partial(torch.optim.SGD, momentum=0.9, neste
rov=True)
}


def make_scheduler(params, max_steps):

    name, *args = params.split("_")

    if name == "steplr":

        step_size, gamma = args
        step_size = int(step_size)
```

```
        gamma = float(gamma)

        return partial(StepLR, step_size=step_size, gamma=ga
mma)

    elif name == "1cycle":

        min_lr, max_lr = args
        min_lr = float(min_lr)
        max_lr = float(max_lr)

        return partial(
            OneCycleScheduler, min_lr=min_lr, max_lr=max_lr,
 max_steps=max_steps)


def make_step(scheduler, epoch=None, step=None, val_score=No
ne):

    if isinstance(scheduler, StepLR) and epoch is not None:
        scheduler.step(epoch)

    elif isinstance(scheduler, OneCycleScheduler) and step i
s not None:
        scheduler.step()


class CyclicLR:
    """Sets the learning rate of each parameter group accord
ing to
    cyclical learning rate policy (CLR). The policy cycles t
he learning
    rate between two boundaries with a constant frequency, a
s detailed in
    the paper `Cyclical Learning Rates for Training Neural N
etworks`_.
    The distance between the two boundaries can be scaled on
 a per-iteration
    or per-cycle basis.

    Cyclical learning rate policy changes the learning rate
after every batch.
    `batch_step` should be called after a batch has been use
d for training.
    To resume training, save `last_batch_iteration` and use
it to instantiate `CycleLR`.

    This class has three built-in policies, as put forth in
the paper:
    "triangular":
        A basic triangular cycle w/ no amplitude scaling.
    "triangular2":
```

```
        A basic triangular cycle that scales initial amplitu
de by half each cycle.
    "exp_range":
        A cycle that scales initial amplitude by gamma**(cyc
le iterations) at each
        cycle iteration.

    This implementation was adapted from the github repo: `b
ckenstler/CLR`_

    Args:
        optimizer (Optimizer): Wrapped optimizer.
        base_lr (float or list): Initial learning rate which
 is the
            lower boundary in the cycle for eachparam groups
.
            Default: 0.001
        max_lr (float or list): Upper boundaries in the cycl
e for
            each parameter group. Functionally,
            it defines the cycle amplitude (max_lr - base_lr
).
            The lr at any cycle is the sum of base_lr
            and some scaling of the amplitude; therefore
            max_lr may not actually be reached depending on
            scaling function. Default: 0.006
        step_size (int): Number of training iterations per
            half cycle. Authors suggest setting step_size
            2-8 x training iterations in epoch. Default: 200
0
        mode (str): One of {triangular, triangular2, exp_ran
ge}.
            Values correspond to policies detailed above.
            If scale_fn is not None, this argument is ignore
d.
            Default: 'triangular'
        gamma (float): Constant in 'exp_range' scaling funct
ion:
            gamma**(cycle iterations)
            Default: 1.0
        scale_fn (function): Custom scaling policy defined b
y a single
            argument lambda function, where
            0 <= scale_fn(x) <= 1 for all x >= 0.
            mode paramater is ignored
            Default: None
        scale_mode (str): {'cycle', 'iterations'}.
            Defines whether scale_fn is evaluated on
            cycle number or cycle iterations (training
            iterations since start of cycle).
            Default: 'cycle'
        last_batch_iteration (int): The index of the last ba
```

```
tch. Default: -1

    Example:
        >>> optimizer = torch.optim.SGD(model.parameters(),
lr=0.1, momentum=0.9)
        >>> scheduler = torch.optim.CyclicLR(optimizer)
        >>> data_loader = torch.utils.data.DataLoader(...)
        >>> for epoch in range(10):
        >>>     for batch in data_loader:
        >>>         scheduler.batch_step()
        >>>         train_batch(...)

    .. _Cyclical Learning Rates for Training Neural Networks
: https://arxiv.org/abs/1506.01186
    .. _bckenstler/CLR: https://github.com/bckenstler/CLR
    """

    def __init__(self, optimizer, base_lr=1e-3, max_lr=6e-3,
                 step_size=2000, mode='triangular', gamma=1.
,
                 scale_fn=None, scale_mode='cycle', last_bat
ch_iteration=-1):

        if not isinstance(optimizer, Optimizer):
            raise TypeError('{} is not an Optimizer'.format(
                type(optimizer).__name__))
        self.optimizer = optimizer

        if isinstance(base_lr, list) or isinstance(base_lr,
tuple):
            if len(base_lr) != len(optimizer.param_groups):
                raise ValueError("expected {} base_lr, got {
}".format(
                    len(optimizer.param_groups), len(base_lr
)))
            self.base_lrs = list(base_lr)
        else:
            self.base_lrs = [base_lr] * len(optimizer.param_
groups)

        if isinstance(max_lr, list) or isinstance(max_lr, tu
ple):
            if len(max_lr) != len(optimizer.param_groups):
                raise ValueError("expected {} max_lr, got {}
".format(
                    len(optimizer.param_groups), len(max_lr)
))
            self.max_lrs = list(max_lr)
        else:
            self.max_lrs = [max_lr] * len(optimizer.param_gr
oups)
```

```
        self.step_size = step_size

        if mode not in ['triangular', 'triangular2', 'exp_ra
nge'] \
                and scale_fn is None:
            raise ValueError('mode is invalid and scale_fn i
s None')

        self.mode = mode
        self.gamma = gamma

        if scale_fn is None:
            if self.mode == 'triangular':
                self.scale_fn = self._triangular_scale_fn
                self.scale_mode = 'cycle'
            elif self.mode == 'triangular2':
                self.scale_fn = self._triangular2_scale_fn
                self.scale_mode = 'cycle'
            elif self.mode == 'exp_range':
                self.scale_fn = self._exp_range_scale_fn
                self.scale_mode = 'iterations'
        else:
            self.scale_fn = scale_fn
            self.scale_mode = scale_mode

        self.batch_step(last_batch_iteration + 1)
        self.last_batch_iteration = last_batch_iteration

    def batch_step(self, batch_iteration=None):
        if batch_iteration is None:
            batch_iteration = self.last_batch_iteration + 1
        self.last_batch_iteration = batch_iteration
        for param_group, lr in zip(self.optimizer.param_grou
ps, self.get_lr()):
            param_group['lr'] = lr

    def _triangular_scale_fn(self, x):
        return 1.

    def _triangular2_scale_fn(self, x):
        return 1 / (2. ** (x - 1))

    def _exp_range_scale_fn(self, x):
        return self.gamma**(x)

    def get_lr(self):
        step_size = float(self.step_size)
        cycle = np.floor(1 + self.last_batch_iteration / (2
* step_size))
        x = np.abs(self.last_batch_iteration / step_size - 2
 * cycle + 1)
```

```
        lrs = []
        param_lrs = zip(self.optimizer.param_groups, self.ba
se_lrs, self.max_lrs)
        for param_group, base_lr, max_lr in param_lrs:
            base_height = (max_lr - base_lr) * np.maximum(0,
 (1 - x))
            if self.scale_mode == 'cycle':
                lr = base_lr + base_height * self.scale_fn(c
ycle)
            else:
                lr = base_lr + base_height * self.scale_fn(s
elf.last_batch_iteration)
            lrs.append(lr)
        return lrs


def annealing_linear(start, end, r):
    return start + r * (end - start)

def annealing_cos(start, end, r):
    cos_out = np.cos(np.pi * r) + 1
    return end + (start - end) / 2 * cos_out


class OneCycleScheduler:
    def __init__(
        self, optimizer,
        min_lr, max_lr,
        max_steps, annealing=annealing_linear):

        self.optimizer = optimizer
        self.min_lr = min_lr
        self.max_lr = max_lr
        self.max_steps = max_steps
        self.annealing = annealing
        self.epoch = -1

    def step(self):
        self.epoch += 1

        mid = int(round(self.max_steps * 0.3))

        if self.epoch < mid:
            r = self.epoch / mid
            lr = self.annealing(self.min_lr, self.max_lr, r)
        else:
            r = (self.epoch - mid) / (self.max_steps - mid)
            lr = self.annealing(self.max_lr, self.min_lr / 1
e3, r)

        for param_group in self.optimizer.param_groups:
            param_group['lr'] = lr
```

```
======
import random
import math
from functools import partial
import json

import pysndfx
import librosa
import numpy as np
import torch

from ops.audio import (
    read_audio, compute_stft, trim_audio, mix_audio_and_labe
ls,
    shuffle_audio, cutout
)


SAMPLE_RATE = 44100


class Augmentation:
    """A base class for data augmentation transforms"""
    pass


class MapLabels:

    def __init__(self, class_map, drop_raw=True):

        self.class_map = class_map

    def __call__(self, dataset, **inputs):

        labels = np.zeros(len(self.class_map), dtype=np.floa
t32)
        for c in inputs["raw_labels"]:
            labels[self.class_map[c]] = 1.0

        transformed = dict(inputs)
        transformed["labels"] = labels
        transformed.pop("raw_labels")

        return transformed


class MixUp(Augmentation):

    def __init__(self, p):

        self.p = p
```

```
    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        if np.random.uniform() < self.p:
            first_audio, first_labels = inputs["audio"], inp
uts["labels"]
            random_sample = dataset.random_clean_sample()
            new_audio, new_labels = mix_audio_and_labels(
                first_audio, random_sample["audio"],
                first_labels, random_sample["labels"]
            )

            transformed["audio"] = new_audio
            transformed["labels"] = new_labels

        return transformed


class FlipAudio(Augmentation):

    def __init__(self, p):

        self.p = p

    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        if np.random.uniform() < self.p:
            transformed["audio"] = np.flipud(inputs["audio"]
)

        return transformed


class AudioAugmentation(Augmentation):

    def __init__(self, p):

        self.p = p

    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        if np.random.uniform() < self.p:
            effects_chain = (
                pysndfx.AudioEffectsChain()
                .reverb(
                    reverberance=random.randrange(50),
```

```
                    room_scale=random.randrange(50),
                    stereo_depth=random.randrange(50)
                )
                .pitch(shift=random.randrange(-300, 300))
                .overdrive(gain=random.randrange(2, 10))
                .speed(random.uniform(0.9, 1.1))
            )
            transformed["audio"] = effects_chain(inputs["aud
io"])

        return transformed


class LoadAudio:

    def __init__(self):

        pass

    def __call__(self, dataset, **inputs):

        audio, sr = read_audio(inputs["filename"])

        transformed = dict(inputs)
        transformed["audio"] = audio
        transformed["sr"] = sr

        return transformed


class STFT:

    eps = 1e-4

    def __init__(self, n_fft, hop_size):

        self.n_fft = n_fft
        self.hop_size = hop_size

    def __call__(self, dataset, **inputs):

        stft = compute_stft(
            inputs["audio"],
            window_size=self.n_fft, hop_size=self.hop_size,
            eps=self.eps)

        transformed = dict(inputs)
        transformed["stft"] = np.transpose(stft)

        return transformed
```

```
class AudioFeatures:

    eps = 1e-4

    def __init__(self, descriptor, verbose=True):

        name, *args = descriptor.split("_")

        self.feature_type = name

        if name == "stft":

            n_fft, hop_size = args
            self.n_fft = int(n_fft)
            self.hop_size = int(hop_size)

            self.n_features = self.n_fft // 2 + 1
            self.padding_value = 0.0

            if verbose:
                print(
                    "\nUsing STFT features with params:\n",
                    "n_fft: {}, hop_size: {}".format(
                        n_fft, hop_size
                    )
                )

        elif name == "mel":

            n_fft, hop_size, n_mel = args
            self.n_fft = int(n_fft)
            self.hop_size = int(hop_size)
            self.n_mel = int(n_mel)

            self.n_features = self.n_mel
            self.padding_value = 0.0

            if verbose:
                print(
                    "\nUsing mel features with params:\n",
                    "n_fft: {}, hop_size: {}, n_mel: {}".for
mat(
                        n_fft, hop_size, n_mel
                    )
                )

        elif name == "raw":

            self.n_features = 1
            self.padding_value = 0.0

            if verbose:
```

```
                print(
                    "\nUsing raw waveform features."
                )


    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        if self.feature_type == "stft":

            # stft = compute_stft(
            #     inputs["audio"],
            #     window_size=self.n_fft, hop_size=self.hop_
size,
            #     eps=self.eps, log=True
            # )

            transformed["signal"] = np.expand_dims(inputs["a
udio"], -1)

        elif self.feature_type == "mel":

            stft = compute_stft(
                inputs["audio"],
                window_size=self.n_fft, hop_size=self.hop_si
ze,
                eps=self.eps, log=False
            )

            transformed["signal"] = np.expand_dims(inputs["a
udio"], -1)

        elif self.feature_type == "raw":
            transformed["signal"] = np.expand_dims(inputs["a
udio"], -1)

        return transformed


class SampleSegment(Augmentation):

    def __init__(self, ratio=(0.3, 0.9), p=1.0):

        self.min, self.max = ratio
        self.p = p

    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        if np.random.uniform() < self.p:
```

```
            original_size = inputs["audio"].size
            target_size = int(np.random.uniform(self.min, se
lf.max) * original_size)
            start = np.random.randint(original_size - target
_size - 1)
            transformed["audio"] = inputs["audio"][start:sta
rt+target_size]

        return transformed


class ShuffleAudio(Augmentation):

    def __init__(self, chunk_length=0.5, p=0.5):

        self.chunk_length = chunk_length
        self.p = p

    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        if np.random.uniform() < self.p:
            transformed["audio"] = shuffle_audio(
                transformed["audio"], self.chunk_length, sr=
transformed["sr"])

        return transformed


class CutOut(Augmentation):

    def __init__(self, area=0.25, p=0.5):

        self.area = area
        self.p = p

    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        if np.random.uniform() < self.p:
            transformed["audio"] = cutout(
                transformed["audio"], self.area)

        return transformed


class SampleLongAudio:

    def __init__(self, max_length):
```

```
        self.max_length = max_length

    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        if (inputs["audio"].size / inputs["sr"]) > self.max_
length:

            max_length = self.max_length * inputs["sr"]

            start = np.random.randint(0, inputs["audio"].siz
e - max_length)
            transformed["audio"] = inputs["audio"][start:sta
rt+max_length]

        return transformed


class OneOf:

    def __init__(self, transforms):

        self.transforms = transforms

    def __call__(self, dataset, **inputs):

        transform = random.choice(self.transforms)
        return transform(**inputs)


class DropFields:

    def __init__(self, fields):

        self.to_drop = fields

    def __call__(self, dataset, **inputs):

        transformed = dict()

        for name, input in inputs.items():
            if not name in self.to_drop:
                transformed[name] = input

        return transformed


class RenameFields:

    def __init__(self, mapping):
```

```
        self.mapping = mapping

    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        for old, new in self.mapping.items():
            transformed[new] = transformed.pop(old)

        return transformed


class Compose:

    def __init__(self, transforms):
        self.transforms = transforms

    def switch_off_augmentations(self):
        for t in self.transforms:
            if isinstance(t, Augmentation):
                t.p = 0.0

    def __call__(self, dataset=None, **inputs):
        for t in self.transforms:
            inputs = t(dataset=dataset, **inputs)

        return inputs


class Identity:

    def __call__(self, dataset=None, **inputs):

        return inputs======
import json

import torch
import umap
import numpy as np
from sklearn.manifold import TSNE
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import label_ranking_average_precision_
score, accuracy_score
from matplotlib import pyplot as plt
import librosa


# Calculate the overall lwlrap using sklearn.metrics functio
n.
```

```
def lwlrap(truth, scores):
  """Calculate the overall lwlrap using sklearn.metrics.lrap
."""
  # sklearn doesn't correctly apply weighting to samples wit
h no labels, so just skip them.
  sample_weight = np.sum(truth > 0, axis=1)
  nonzero_weight_sample_indices = np.flatnonzero(sample_weig
ht > 0)
  overall_lwlrap = label_ranking_average_precision_score(
      truth[nonzero_weight_sample_indices, :] > 0,
      scores[nonzero_weight_sample_indices, :],
      sample_weight=sample_weight[nonzero_weight_sample_indi
ces])
  return overall_lwlrap


def load_json(file):
    with open(file, "r") as f:
        return json.load(f)


def get_class_names_from_classmap(classmap):
    r = dict((v, k) for k, v in classmap.items())
    return [r[label] for label in sorted(classmap.values())]


def plot_projection(vectors, labels, frames_per_example=3, n
ewline=False):

    representations = []
    classes = []
    for sample, label in zip(vectors, labels):
        if sum(label) > 1:
            continue
        choices = np.random.choice(
            np.arange(len(sample)), replace=False,
            size=min(frames_per_example, len(sample)))
        representations.extend(sample[choices])
        classes.extend([label.tolist().index(1)] * len(choic
es))

    representations = np.array(representations)

    # fit a simple model to estimate the quality of the lear
ned representations
    X_train, X_valid, y_train, y_valid = train_test_split(
        representations, classes, shuffle=False, test_size=0
.2)

    scaler = StandardScaler().fit(X_train)
    X_train = scaler.transform(X_train)
    X_valid = scaler.transform(X_valid)
```

```
    model = KNeighborsClassifier(n_neighbors=5)
    model.fit(X_train, y_train)

    score = accuracy_score(y_valid, model.predict(X_valid))
    if newline:
        print()
    print("Classification accuracy: {:.4f}".format(score))

    # plot projection
    embeddings = TSNE().fit_transform(representations)

    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111)
    ax.scatter(embeddings[:, 0], embeddings[:, 1], c=classes
, s=10)

    fig.canvas.draw()

    image = np.array(fig.canvas.renderer._renderer)

    plt.close()

    return image


def make_mel_filterbanks(descriptor, sr=44100):

    name, *args = descriptor.split("_")

    n_fft, hop_size, n_mel = args
    n_fft = int(n_fft)
    hop_size = int(hop_size)
    n_mel = int(n_mel)

    filterbank = librosa.filters.mel(
        sr=sr, n_fft=n_fft, n_mels=n_mel,
        fmin=5, fmax=None
    ).astype(np.float32)

    return filterbank


def is_mel(descriptor):
    return descriptor.startswith("mel")


def is_stft(descriptor):
    return descriptor.startswith("stft")


def compute_torch_stft(audio, descriptor):
```

```
    name, *args = descriptor.split("_")

    n_fft, hop_size, *rest = args
    n_fft = int(n_fft)
    hop_size = int(hop_size)

    stft = torch.stft(
        audio,
        n_fft=n_fft,
        hop_length=hop_size,
        window=torch.hann_window(n_fft, device=audio.device)
    )

    stft = torch.sqrt((stft ** 2).sum(-1))

    return stft

======
import os
import glob
import pickle
import random
import json

import torch
from tqdm import tqdm
import torch.utils.data as data
import numpy as np
import pandas as pd


class SoundDataset(data.Dataset):

    def __init__(
        self, audio_files, labels=None,
        transform=None, is_noisy=None, clean_transform=None)
:

        self.transform = transform
        self.clean_transform = clean_transform
        self.audio_files = audio_files
        self.labels = labels
        self.is_noisy = is_noisy or np.zeros(len(self.audio_
files))

    def __getitem__(self, index):

        sample = dict(
            filename=self.audio_files[index],
            is_noisy=self.is_noisy[index]
        )
```

```
        if self.labels is not None:
            sample["raw_labels"] = self.labels[index]

        if self.transform is not None:
            sample = self.transform(dataset=self, **sample)

        return sample

    def random_clean_sample(self):

        index = random.randint(0, len(self) - 1)

        sample = dict(
            filename=self.audio_files[index],
            is_noisy=self.is_noisy[index]
        )

        if self.labels is not None:
            sample["raw_labels"] = self.labels[index]

        if self.clean_transform is not None:
            sample = self.clean_transform(dataset=self, **sa
mple)

        return sample

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