

```
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
```



```

1.0),
2.3),
                                resize_ratio=(1.7,
                                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_lwlrp', max_saves=1),
        ReduceLROnPlateau(monitor='val_lwlrp',
patience=6, factor=0.6, min_lr=1e-8),
        EarlyStopping(monitor='val_lwlrp', patience=18),
        LoggingToFile(save_dir / 'log.txt'),
    ]

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

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 LwlrBase
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_lwlrwrap_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)

    lwlrwrap = LwlrwrapBase(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
        lwlrwrap.accumulate(target[np.newaxis],
pred[np.newaxis])

    result = {

```

```

        'overall_lwlrwrap': lwlrwrap.overall_lwlrwrap(),
        'per_class_lwlrwrap': {cls: lw for cls, lw in
zip(config.classes,
lwlrwrap.per_class_lwlrwrap())}
    }
    print(result)
    with open(PREDICTION_DIR / 'val_lwlrwrap.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 lwlrwrap metric on cv")
        calc_lwlrwrap_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 ddb02ae88b6bd05c1b9726d2fd30c38854be4fd
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 05a7aee7c50148677735531bddd32902b468bea
nvidia-docker run ${DOCKER_OPTIONS} python
train_folds.py --experiment corrections_002
```

```
# Experiment corrections_003
```

```
git checkout 24a4f20fffc284d22b38bbabfe510ed194f62e496
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, ReduceLR0nPlateau

```

```

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']

```



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

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

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 LwlrBase
```

```
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_lwlrwrap_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)

    lwlrwrap = LwlrwrapBase(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
        lwlrwrap.accumulate(target[np.newaxis],
pred[np.newaxis])

    result = {
        'overall_lwlrwrap': lwlrwrap.overall_lwlrwrap(),
        'per_class_lwlrwrap': {cls: lwlr for cls, lwlr in
zip(config.classes,

```

```
lwlap.per_class_lwlap()))}
    }
    print(result)
    with open(PREDICTION_DIR / 'val_lwlap.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 lwlap metric on cv")
        calc_lwlap_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, ReduceLR0nPlateau
```

```
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_lwlrp', max_saves=1),
            ReduceLR0nPlateau(monitor='val_lwlrp',

patience=random_params['patience'],

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

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

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, ReduceLR0nPlateau

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_lwlrp', max_saves=1),
        ReduceLR0nPlateau(monitor='val_lwlrp',
patience=random_params['patience'],
factor=random_params['factor'],
                                min_lr=1e-8),
        EarlyStopping(monitor='val_lwlrp',
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', 'lwlrp'])
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  $\eta_{\max}$  is set to the
    initial lr,  $T_{\text{cur}}$ 
    is the number of epochs since the last restart
    and  $T_i$  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_{\text{cur}}}{T_i}\pi))
    When  $T_{\text{cur}}=T_i$ , set  $\eta_t =
    \eta_{\min}$ .
    When  $T_{\text{cur}}=0$  (after restart),
    set  $\eta_t=\eta_{\max}$ .
    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  $T_i$  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 LwlrBase:
    """Computes label-weighted label-ranked average
precision (lwlrp)."""

    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_lwlap(self):
        """Return a vector of the per-class lwlaps 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_lwlap(self):
        """Return the scalar overall lwlap for
cumulated samples."""
        return np.sum(self.per_class_lwlap() *
self.per_class_weight())

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

```

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

```
class Lwlap(Metric):
    name = 'lwlap'
    better = 'max'

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

        self.lwlap = LwlapBase(self.classes)

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

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

    def compute(self):
        return self.lwlap.overall_lwlap()
```

```
=====
```

```
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)

```

```
\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' -
averaging
    """

```

```

        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)
        blurred_image = cv2.GaussianBlur(image,
```

```
kernel_size, self.sigma_x)
    return blurred_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
=====
```