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

3rd place solution to Freesound Audio Tagging 2019 Challenge

My approach is outlined below.

Models

I used two types of models, both are based on convolutions. The first type uses 2d convolutions and works on top of mel-scale spectrograms, while the second uses 1d-convolutions on top of raw STFT representations with relatively small window size like 256, so it's only 5 ms per frame or so. Both types of models are relatively shallow and consist of 10-12 convolutional layers (or 5-6 resnet blocks) with a small number of filters. I use a form of deep supervision by applying global max pooling after each block (typically starting from the first or second block) and then concatenating maxpool outputs from each layer to form the final feature vector which then goes to a 2-layer fully-connected classifier. I also tried using RNNs instead of a max pooling for some models. It made results a bit worse, but RNN seemed to make different mistakes, so it turned out to be a good member of the final ensemble.

Frequency encoding

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

This is not really a classification task

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

Data augmentation

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

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

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

Training

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

Noisy data

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

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

Inference

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

Final ensemble

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

Project structure

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

Setting up the environment

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

Preparing data

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

```bash
python create_class_map.py --train_df data/train_curated.csv --o
utput_file data/classmap.json
```

This simply creates a JSON file with deterministic classname->la
bel mapping used in all future experiments.

Running a basic 2d model

```bash
python train_2d_cnn.py \
  --train_df data/train_curated.csv \
  --train_data_dir data/train_curated/ \
  --classmap data/classmap.json \
  --device=cuda \
  --optimizer=adam \
  --folds 0 1 2 3 4 \
  --n_folds=5 \
  --log_interval=10 \
  --batch_size=20 \
  --epochs=20 \
  --accumulation_steps=1 \
  --save_every=20 \
  --num_conv_blocks=5 \
  --conv_base_depth=50 \
  --growth_rate=1.5 \
  --weight_decay=0.0 \
  --start_deep_supervision_on=1 \
  --aggregation_type=max \
  --lr=0.003 \
  --scheduler=1cycle_0.0001_0.005 \
  --test_data_dir data/test \
  --sample_submission data/sample_submission.csv \
  --num_workers=6 \
  --output_dropout=0.0 \
  --p_mixup=0.0 \
  --switch_off_augmentations_on=15 \
  --features=mel_2048_1024_128 \
  --max_audio_length=15 \
  --p_aug=0.0 \
  --label=basic_2d_cnn
```

Running a 2d model with augmentations


```bash
python train_2d_cnn.py \
  --train_df data/train_curated.csv \
  --train_data_dir data/train_curated/ \
```

```
  --classmap data/classmap.json \
  --device=cuda \
  --optimizer=adam \
  --folds 0 1 2 3 4 \
  --n_folds=5 \
  --log_interval=10 \
  --batch_size=20 \
  --epochs=100 \
  --accumulation_steps=1 \
  --save_every=20 \
  --num_conv_blocks=5 \
  --conv_base_depth=100 \
  --growth_rate=1.5 \
  --weight_decay=0.0 \
  --start_deep_supervision_on=1 \
  --aggregation_type=max \
  --lr=0.003 \
  --scheduler=1cycle_0.0001_0.005 \
  --test_data_dir data/test \
  --sample_submission data/sample_submission.csv \
  --num_workers=16 \
  --output_dropout=0.5 \
  --p_mixup=0.5 \
  --switch_off_augmentations_on=90 \
  --features=mel_2048_1024_128 \
  --max_audio_length=15 \
  --p_aug=0.75 \
  --label=2d_cnn
```

Note that each such run is followed by a creation of a new experiment subdirectory in the `experiments` folder. Each experiment has the following structure:

```bash
experiments/some_experiment/
¿¿¿ checkpoints
¿¿¿ command
¿¿¿ commit_hash
¿¿¿ config.json
¿¿¿ log
¿¿¿ predictions
¿¿¿ results.json
¿¿¿ summaries
```

Using a clean model to select noisy samples

Create a new predictions directory:

```mkdir predictions/```

Then, running

```bash
python predict_2d_cnn.py \
```
```

```
  --experiment=path_to_an_experiment (see above) \
  --test_df=data/train_noisy.csv \
  --test_data_dir=data/train_noisy/ \
  --output_df=predictions/noisy_probabilities.csv \
  --classmap=data/classmap.json \
  --device=cuda
```

creates a new csv file in the predictions folder with the class probabilties for the noisy dataset.

Running

```bash
python relabel_noisy_data.py \
  --noisy_df=data/train_noisy.csv \
  --noisy_predictions_df=predictions/noisy_probabilities.csv \
  --output_df=predictions/train_noisy_relabeled_1k.csv \
  --mode=scoring_1000
```

creates a new noisy dataframe where only top 1k labels in terms of agreement between the model and the actual labels are kept.

Running a 2d model with noisy data

```bash
python train_2d_cnn.py \
  --train_df data/train_curated.csv \
  --train_data_dir data/train_curated/ \
  --noisy_train_df predictions/ train_noisy_relabeled_1k.csv \
  --noisy_train_data_dir data/train_noisy/ \
  --classmap data/classmap.json \
  --device=cuda \
  --optimizer=adam \
  --folds 0 1 2 3 4 \
  --n_folds=5 \
  --log_interval=10 \
  --batch_size=20 \
  --epochs=150 \
  --accumulation_steps=1 \
  --save_every=20 \
  --num_conv_blocks=6 \
  --conv_base_depth=100 \
  --growth_rate=1.5 \
  --weight_decay=0.0 \
  --start_deep_supervision_on=1 \
  --aggregation_type=max \
  --lr=0.003 \
  --scheduler=1cycle_0.0001_0.005 \
  --test_data_dir data/test \
  --sample_submission data/sample_submission.csv \
  --num_workers=16 \
  --output_dropout=0.7 \
```

```
  --p_mixup=0.5 \
  --switch_off_augmentations_on=140 \
  --features=mel_2048_1024_128 \
  --max_audio_length=15 \
  --p_aug=0.75 \
  --label=2d_cnn_noisy
```

Note that `relabel_noisy_data.py` script supports multiple relab
eling straregies. I mostly followed "scoring" strategy (selectin
g top-k noisy samples based on the agreement between the model a
nd the actual labels), but after 5k noisy samples I switched to
"relabelall-replacenan" strategy which is just a pseudolabeling
(usage of the old model outputs) where the samples without any p
redictions are discarded.

```
======
absl-py==0.7.1
astor==0.7.1
attrs==19.1.0
audioread==2.1.6
backcall==0.1.0
bleach==3.1.0
certifi==2019.3.9
cffi==1.12.2
chardet==3.0.4
cycler==0.10.0
decorator==4.4.0
defusedxml==0.5.0
entrypoints==0.3
gast==0.2.2
grpcio==1.19.0
h5py==2.9.0
idna==2.8
ipykernel==5.1.0
ipython==7.4.0
ipython-genutils==0.2.0
ipywidgets==7.4.2
iterative-stratification==0.1.6
jedi==0.13.3
Jinja2==2.10.1
joblib==0.13.2
jsonschema==3.0.1
jupyter==1.0.0
jupyter-client==5.2.4
jupyter-console==6.0.0
jupyter-core==4.4.0
kaggle==1.5.3
Keras-Applications==1.0.7
Keras-Preprocessing==1.0.9
kiwisolver==1.0.1
librosa==0.6.3
llvmlite==0.28.0
mag==0.1
Markdown==3.1
MarkupSafe==1.1.1
matplotlib==3.0.3
```

```
mistune==0.8.4
mock==2.0.0
munch==2.3.2
nbconvert==5.4.1
nbformat==4.4.0
notebook==5.7.8
numba==0.43.1
numpy==1.16.2
pandas==0.24.2
pandocfilters==1.4.2
parso==0.4.0
pbr==5.1.3
pexpect==4.7.0
pickleshare==0.7.5
Pillow==6.0.0
pkg-resources==0.0.0
pretrainedmodels==0.7.4
prometheus-client==0.6.0
prompt-toolkit==2.0.9
protobuf==3.7.1
ptyprocess==0.6.0
pycparser==2.19
Pygments==2.3.1
pyparsing==2.3.1
pyrsistent==0.14.11
pysndfx==0.3.6
python-dateutil==2.8.0
python-slugify==3.0.2
pytz==2018.9
pyzmq==18.0.1
qtconsole==4.4.3
requests==2.21.0
resampy==0.2.1
scikit-learn==0.20.3
scipy==1.2.1
Send2Trash==1.5.0
six==1.12.0
SoundFile==0.10.2
tensorboard==1.13.1
tensorboardX==1.6
tensorflow==1.13.1
tensorflow-estimator==1.13.0
termcolor==1.1.0
terminado==0.8.2
testpath==0.4.2
text-unidecode==1.2
torch==1.0.1.post2
torchcontrib==0.0.2
torchvision==0.2.2.post3
tornado==6.0.2
tqdm==4.31.1
traitlets==4.3.2
umap-learn==0.3.8
urllib3==1.24.1
wcwidth==0.1.7
webencodings==0.5.1
```

```
Werkzeug==0.15.2
widgetsnbextension==3.4.2

git+https://github.com/ex4sperans/mag
import os
import gc
import argparse
import json
import math
from functools import partial

import tqdm
import pandas as pd
import numpy as np
import torch
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score

from datasets.sound_dataset import SoundDataset
from networks.classifiers import HierarchicalCNNClassificationMo
del
from ops.folds import train_validation_data
from ops.transforms import (
    Compose, DropFields, LoadAudio,
    AudioFeatures, MapLabels, RenameFields,
    MixUp, SampleSegment, SampleLongAudio)
from ops.utils import load_json, get_class_names_from_classmap,
lwlrap
from ops.padding import make_collate_fn
from networks.classifiers import ResnetBlock

torch.manual_seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(42)

parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--train_df", required=True, type=str,
    help="path to train dataframe"
)
parser.add_argument(
    "--train_data_dir", required=True, type=str,
    help="path to train data"
)
parser.add_argument(
    "--test_data_dir", required=True, type=str,
    help="path to test data"
)
parser.add_argument(
    "--test_df", required=True, type=str,
    help="path to train dataframe"
)
```

```
parser.add_argument(
    "--val_size", required=True, type=float,
    help="size of the validation set"
)
parser.add_argument(
    "--device", type=str, required=True,
    help="whether to train on cuda or cpu",
    choices=("cuda", "cpu")
)
parser.add_argument(
    "--batch_size", type=int, default=64,
    help="minibatch size"
)
parser.add_argument(
    "--epochs", type=int, default=100,
    help="number of epochs"
)
parser.add_argument(
    "--lr", default=0.01, type=float,
    help="starting learning rate"
)
parser.add_argument(
    "--max_samples", type=int,
    help="maximum number of samples to use"
)
parser.add_argument(
    "--features", type=str, required=True,
    help="feature descriptor"
)
parser.add_argument(
    "--max_audio_length", type=int, default=10,
    help="max audio length in seconds. For longer clips are samp
led"
)
parser.add_argument(
    "--batches_to_save", type=int, default=3,
    help="how many batches to save"
)
parser.add_argument(
    "--classmap", required=True, type=str,
    help="path to class map json"
)

args = parser.parse_args()

train_df = pd.read_csv(args.train_df)
test_df = pd.read_csv(args.test_df)

if args.max_samples:
    train_df = train_df.sample(args.max_samples).reset_index(dro
p=True)
    test_df = test_df.sample(args.max_samples).reset_index(drop=
True)

all_train_fnames = [
    os.path.join(args.train_data_dir, fname) for fname in train_
```

```
df.fname.values]
all_test_fnames = [
    os.path.join(args.test_data_dir, fname) for fname in test_df
.fname.values]

fnames = np.concatenate([all_train_fnames, all_test_fnames])
labels = np.concatenate([np.ones(len(train_df)), np.zeros(len(te
st_df))])

train_fnames, val_fnames, train_labels, val_labels = train_test_
split(
    fnames, labels, test_size=args.val_size, shuffle=True)

audio_transform = AudioFeatures(args.features)


class Model(torch.nn.Module):

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

        self.features = torch.nn.Sequential(
            torch.nn.BatchNorm1d(audio_transform.n_features),
            torch.nn.Conv1d(audio_transform.n_features, 32, kern
el_size=1),
            ResnetBlock(32),
            torch.nn.MaxPool1d(kernel_size=2, stride=2),
            torch.nn.BatchNorm1d(32),
            torch.nn.Conv1d(32, 32, kernel_size=3),
            ResnetBlock(32),
            torch.nn.MaxPool1d(kernel_size=2, stride=2),
            torch.nn.BatchNorm1d(32),
            torch.nn.Conv1d(32, 64, kernel_size=3),
            ResnetBlock(64)
        )

        self.pool = torch.nn.AdaptiveMaxPool1d(1)

        self.classifier = torch.nn.Sequential(
            torch.nn.BatchNorm1d(64),
            torch.nn.Conv1d(64, 1, kernel_size=1)
        )

    def forward(self, x):

        x = x.permute(0, 2, 1)
        x = self.features(x)
        x = self.classifier(x)
        x = torch.sigmoid(x)
        nonpooled = x
        x = self.pool(x).squeeze(-1)

        return x.squeeze(1), nonpooled.squeeze(1)


train_loader = torch.utils.data.DataLoader(
```

```
    SoundDataset(
        audio_files=train_fnames,
        labels=train_labels,
        transform=Compose([
            LoadAudio(),
            SampleLongAudio(max_length=args.max_audio_length),
            audio_transform,
            RenameFields({"raw_labels": "labels"}),
            DropFields(("audio", "filename", "sr")),
        ]),
        clean_transform=Compose([
            LoadAudio(),
        ])
    ),
    shuffle=True,
    drop_last=True,
    batch_size=args.batch_size,
    num_workers=4,
    collate_fn=make_collate_fn({"signal": audio_transform.paddin
g_value}),
)

validation_loader = torch.utils.data.DataLoader(
    SoundDataset(
        audio_files=val_fnames,
        labels=val_labels,
        transform=Compose([
            LoadAudio(),
            SampleLongAudio(max_length=args.max_audio_length),
            audio_transform,
            RenameFields({"raw_labels": "labels"}),
            DropFields(("audio", "filename", "sr")),
        ]),
        clean_transform=Compose([
            LoadAudio(),
        ])
    ),
    shuffle=False,
    drop_last=False,
    batch_size=args.batch_size,
    num_workers=4,
    collate_fn=make_collate_fn({"signal": audio_transform.paddin
g_value}),
)

model = Model().to(args.device)
optimizer = torch.optim.Adam(model.parameters(), args.lr)


for epoch in range(args.epochs):

    print(
        "\n" + " " * 10 + "****** Epoch {epoch} ******\n"
        .format(epoch=epoch)
    )
```

```
    model.train()

    with tqdm.tqdm(total=len(train_loader), ncols=80) as pb:

        for sample in train_loader:

            signal, labels = (
                sample["signal"].to(args.device),
                sample["labels"].to(args.device).float()
            )

            probs, nonpooled = model(signal)

            optimizer.zero_grad()
            loss = torch.nn.functional.binary_cross_entropy(prob
s, labels)
            loss.backward()
            optimizer.step()

            pb.update()
            pb.set_description("Loss: {:.4f}".format(loss.item()
))

    model.eval()

    val_probs = []
    val_labels = []

    with torch.no_grad():

        for sample in validation_loader:

            signal, labels = (
                sample["signal"].to(args.device),
                sample["labels"].to(args.device).float()
            )

            probs, nonpooled = model(signal)

            val_probs.extend(probs.data.cpu().numpy())
            val_labels.extend(labels.data.cpu().numpy())

    auc = roc_auc_score(val_labels, val_probs)

    print("\nEpoch: {}, AUC: {}".format(epoch, auc))

model.eval()

# plot probabilities
loader = iter(validation_loader)
directory = "plots/"
os.makedirs(directory, exist_ok=True)

for n in range(args.batches_to_save):

    with torch.no_grad():
```

```
        sample = next(loader)
        signal, labels = (
            sample["signal"].to(args.device),
            sample["labels"].to(args.device).float()
        )

        probs, nonpooled = model(signal)

        nonpooled = nonpooled.data.cpu().numpy()
        signal = signal.data.cpu().numpy()
        labels = labels.data.cpu().numpy()

    for k in range(len(signal)):

        fig = plt.figure(figsize=(20, 7))
        fig.suptitle(str(labels[k]))
        ax = fig.add_subplot(211)
        ax.imshow(np.transpose(signal[k]))
        ax = fig.add_subplot(212)
        ax.plot(nonpooled[k])
        ax.set_ylim(0, 1)
        ax.set_xlim(0, len(nonpooled[k]) - 1)

        fig.savefig(os.path.join(directory, "plot_{}_{}.png".for
mat(n, k)))
        plt.close()

# compute average scores for classes

class_map = load_json(args.classmap)

names_with_labels = [
    fname for fname in val_fnames if fname in all_train_fnames]
labels = pd.DataFrame({
    "fname": [os.path.basename(fname) for fname in names_with_la
bels]}).merge(
        train_df, on="fname", how="left").labels.values

loader = torch.utils.data.DataLoader(
    SoundDataset(
        audio_files=names_with_labels,
        labels=[item.split(",") for item in labels],
        transform=Compose([
            LoadAudio(),
            MapLabels(class_map),
            SampleLongAudio(max_length=args.max_audio_length),
            audio_transform,
            DropFields(("audio", "filename", "sr")),
        ])
    ),
    shuffle=False,
    drop_last=False,
    batch_size=args.batch_size,
    num_workers=4,
    collate_fn=make_collate_fn({"signal": audio_transform.paddin
```

```
g_value}),
)

all_probs = []
all_labels = []

with torch.no_grad():

    for sample in loader:

        signal, labels = (
            sample["signal"].to(args.device),
            sample["labels"].to(args.device).float()
        )

        probs, nonpooled = model(signal)

        all_probs.extend(probs.data.cpu().numpy())
        all_labels.extend(labels.data.cpu().numpy())

all_probs = np.array(all_probs)
all_labels = np.array(all_labels)

scores = all_labels * np.expand_dims(all_probs, -1)
mean_scores = scores.sum(axis=0) / all_labels.sum(axis=0)

classnames = get_class_names_from_classmap(class_map)

pd.options.display.max_rows = 100

print()
print(pd.DataFrame({"classname": classnames, "scores": mean_scor
es}))

======
import json
import argparse

import pandas as pd


parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--train_df", required=True, type=str,
    help="path to train dataframe"
)
parser.add_argument(
    "--output_file", type=str, required=True,
    help="where to save classmap"
)

args = parser.parse_args()
```

```
df = pd.read_csv(args.train_df)

all_labels = set()
for item in df.labels:
    all_labels.update(item.split(","))


classmap = dict((v, k) for k, v in enumerate(sorted(all_labels))
)

with open(args.output_file, "w") as file:
    json.dump(classmap, file, indent=4, sort_keys=True)======
import os
import gc
import argparse
import json
import math
from functools import partial

import pandas as pd
import numpy as np
import torch
from mag.experiment import Experiment
from mag.utils import green, bold
import mag

from datasets.sound_dataset import SoundDataset
from networks.classifiers import TwoDimensionalCNNClassification
Model
from ops.folds import train_validation_data_stratified
from ops.transforms import (
    Compose, DropFields, LoadAudio,
    AudioFeatures, MapLabels, RenameFields,
    MixUp, SampleSegment, SampleLongAudio,
    AudioAugmentation, FlipAudio, ShuffleAudio)
from ops.utils import load_json, get_class_names_from_classmap,
lwlrap
from ops.padding import make_collate_fn

mag.use_custom_separator("-")

parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--experiment", type=str, required=True,
    help="path to an experiment"
)
parser.add_argument(
    "--train_df", required=True, type=str,
    help="path to train dataframe"
)
parser.add_argument(
    "--train_data_dir", required=True, type=str,
```

```
    help="path to train data"
)
parser.add_argument(
    "--noisy_train_df", type=str,
    help="path to noisy train dataframe (optional)"
)
parser.add_argument(
    "--noisy_train_data_dir", type=str,
    help="path to noisy train data (optional)"
)
parser.add_argument(
    "--classmap", required=True, type=str,
    help="path to class map json"
)
parser.add_argument(
    "--batch_size", type=int, default=32,
    help="batch size used for prediction"
)
parser.add_argument(
    "--max_audio_length", type=int, default=10,
    help="max audio length in seconds. For longer clips are samp
led"
)
parser.add_argument(
    "--n_tta", type=int, default=1,
    help="number of tta"
)
parser.add_argument(
    "--device", type=str, required=True,
    help="whether to train on cuda or cpu",
    choices=("cuda", "cpu")
)
parser.add_argument(
    "--num_workers", type=int, default=4,
    help="number of workers for data loader",
)

args = parser.parse_args()

class_map = load_json(args.classmap)

train_df = pd.read_csv(args.train_df)

with Experiment(resume_from=args.experiment) as experiment:

    config = experiment.config

    audio_transform = AudioFeatures(config.data.features)

    splits = list(train_validation_data_stratified(
            train_df.fname, train_df.labels, class_map,
            config.data._n_folds, config.data._kfold_seed))

    all_labels = np.zeros(
        shape=(len(train_df), len(class_map)), dtype=np.float32)
    all_predictions = np.zeros(
```

```
        shape=(len(train_df), len(class_map)), dtype=np.float32)

    for fold in range(config.data._n_folds):

        print("\n\n   -----  Fold {}\n".format(fold))

        train, valid = splits[fold]

        loader_kwargs = (
            {"num_workers": args.num_workers, "pin_memory": True
}
            if torch.cuda.is_available() else {})

        valid_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[valid]],
                labels=[item.split(",") for item in train_df.lab
els.values[valid]],
                transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                    SampleLongAudio(args.max_audio_length),
                    ShuffleAudio(chunks_range=(12, 20), p=1.0),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ]),
                clean_transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                ]),


            ),
            shuffle=False,
            batch_size=args.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        model = TwoDimensionalCNNClassificationModel(
                experiment, device=args.device)
        model.load_best_model(fold)
        model.eval()

        val_preds = model.predict(valid_loader, n_tta=args.n_tta
)
        val_labels = np.array([item["labels"] for item in valid_
loader.dataset])

        all_labels[valid] = val_labels
        all_predictions[valid] = val_preds

        metric = lwlrap(val_labels, val_preds)
```

```
        print("Fold metric:", metric)

    metric = lwlrap(all_labels, all_predictions)

    print("\nOverall metric:", green(bold(metric)))


======
import os
import gc
import argparse
import json
import math
from functools import partial

import pandas as pd
import numpy as np
import torch
from mag.experiment import Experiment
import mag
from sklearn.model_selection import train_test_split

from datasets.sound_dataset import SoundDataset
from networks.classifiers import HierarchicalCNNClassificationMo
del
from ops.folds import train_validation_data
from ops.transforms import (
    Compose, DropFields, LoadAudio,
    STFT, MapLabels, RenameFields, MixUp)
from ops.utils import load_json, get_class_names_from_classmap,
lwlrap
from ops.padding import make_collate_fn

torch.manual_seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(42)

mag.use_custom_separator("-")

parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--train_df", required=True, type=str,
    help="path to train dataframe"
)
parser.add_argument(
    "--train_data_dir", required=True, type=str,
    help="path to train data"
)
parser.add_argument(
    "--test_data_dir", required=True, type=str,
    help="path to test data"
)
parser.add_argument(
```

```
    "--sample_submission", required=True, type=str,
    help="path sample submission"
)
parser.add_argument(
    "--pretrained_model", required=True, type=str,
    help="path to old experiment"
)
parser.add_argument(
    "--pretrained_fold", required=True, type=int,
    help="pretrained fold"
)
parser.add_argument(
    "--classmap", required=True, type=str,
    help="path to class map json"
)
parser.add_argument(
    "--log_interval", default=10, type=int,
    help="how frequently to log batch metrics"
    "in terms of processed batches"
)
parser.add_argument(
    "--batch_size", type=int, default=64,
    help="minibatch size"
)
parser.add_argument(
    "--lr", default=0.01, type=float,
    help="starting learning rate"
)
parser.add_argument(
    "--max_samples", type=int,
    help="maximum number of samples to use"
)
parser.add_argument(
    "--holdout_size", type=float, default=0.0,
    help="size of holdout set"
)
parser.add_argument(
    "--epochs", default=100, type=int,
    help="number of epochs to train"
)
parser.add_argument(
    "--scheduler", type=str, default="steplr_1_0.5",
    help="scheduler type",
)
parser.add_argument(
    "--accumulation_steps", type=int, default=1,
    help="number of gradient accumulation steps",
)
parser.add_argument(
    "--save_every", type=int, default=1,
    help="how frequently to save a model",
)
parser.add_argument(
    "--device", type=str, required=True,
    help="whether to train on cuda or cpu",
    choices=("cuda", "cpu")
```

```
)
parser.add_argument(
    "--weight_decay", type=float, default=1e-5,
    help="weight decay"
)
parser.add_argument(
    "--dropout", type=float, default=0.0,
    help="internal dropout"
)
parser.add_argument(
    "--output_dropout", type=float, default=0.0,
    help="output dropout"
)
parser.add_argument(
    "--p_mixup", type=float, default=0.0,
    help="probability of the mixup augmentation"
)
parser.add_argument(
    "--switch_off_augmentations_on", type=int, default=20,
    help="on which epoch to remove augmentations"
)
parser.add_argument(
    "--optimizer", type=str, required=True,
    help="which optimizer to use",
    choices=("adam", "momentum")
)
parser.add_argument(
    "--folds", type=int, required=True, nargs="+",
    help="which folds to use"
)
parser.add_argument(
    "--n_folds", type=int, default=4,
    help="number of folds"
)
parser.add_argument(
    "--kfold_seed", type=int, default=42,
    help="kfold seed"
)
parser.add_argument(
    "--num_workers", type=int, default=4,
    help="number of workers for data loader",
)
parser.add_argument(
    "--label", type=str, default="finetuned_hierarchical_cnn_cla
ssifier",
    help="optional label",
)
args = parser.parse_args()

class_map = load_json(args.classmap)

pretrained = Experiment(resume_from=args.pretrained_model)

with Experiment({
    "network": {
        "num_conv_blocks": pretrained.config.network.num_conv_bl
```

```
ocks,
        "start_deep_supervision_on": pretrained.config.network.s
tart_deep_supervision_on,
        "conv_base_depth": pretrained.config.network.conv_base_d
epth,
        "growth_rate": pretrained.config.network.growth_rate,
        "dropout": args.dropout,
        "output_dropout": args.output_dropout,
    },
    "data": {
        "_n_folds": args.n_folds,
        "_kfold_seed": args.kfold_seed,
        "n_fft": pretrained.config.data.n_fft,
        "hop_size": pretrained.config.data.hop_size,
        "_input_dim": pretrained.config.data.n_fft // 2 + 1,
        "_n_classes": len(class_map),
        "_holdout_size": args.holdout_size,
        "p_mixup": args.p_mixup
    },
    "train": {
        "accumulation_steps": args.accumulation_steps,
        "batch_size": args.batch_size,
        "learning_rate": args.lr,
        "scheduler": args.scheduler,
        "optimizer": args.optimizer,
        "epochs": args.epochs,
        "_save_every": args.save_every,
        "weight_decay": args.weight_decay,
        "switch_off_augmentations_on": args.switch_off_augmentat
ions_on,
        "_pretrained_experiment": args.pretrained_model,
        "_pretrained_fold": args.pretrained_fold,
    },
    "label": args.label
}) as experiment:

    config = experiment.config
    print()
    print("     ////// CONFIG //////")
    print(experiment.config)

    train_df = pd.read_csv(args.train_df)
    test_df = pd.read_csv(args.sample_submission)

    if args.max_samples:
        train_df = train_df.sample(args.max_samples).reset_index
(drop=True)
        test_df = test_df.sample(
            min(args.max_samples, len(test_df))).reset_index(dro
p=True)

    if args.holdout_size:
        keep, holdout = train_test_split(
            np.arange(len(train_df)), test_size=args.holdout_siz
e,
            random_state=args.kfold_seed)
```

```
        holdout_df = train_df.iloc[holdout].reset_index(drop=Tru
e)
        train_df = train_df.iloc[keep].reset_index(drop=True)

    splits = list(train_validation_data(
        train_df.fname, train_df.labels,
        config.data._n_folds, config.data._kfold_seed))

    for fold in args.folds:

        print("\n\n   -----  Fold {}\n".format(fold))

        train, valid = splits[fold]

        loader_kwargs = (
            {"num_workers": args.num_workers, "pin_memory": True
}
            if torch.cuda.is_available() else {})

        experiment.register_directory("checkpoints")
        experiment.register_directory("predictions")

        train_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[train]],
                labels=[item.split(",") for item in train_df.lab
els.values[train]],
                transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                    MixUp(p=args.p_mixup),
                    STFT(n_fft=config.data.n_fft, hop_size=confi
g.data.hop_size),
                    DropFields(("audio", "filename", "sr")),
                    RenameFields({"stft": "signal"})
                ]),
                clean_transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                ])
            ),
            shuffle=True,
            drop_last=True,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": math.log(STFT.
eps)}),
            **loader_kwargs
        )

        valid_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[valid]],
```

```
                labels=[item.split(",") for item in train_df.lab
els.values[valid]],
                transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                    STFT(n_fft=config.data.n_fft, hop_size=confi
g.data.hop_size),
                    DropFields(("audio", "filename", "sr")),
                    RenameFields({"stft": "signal"})
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": math.log(STFT.
eps)}),
            **loader_kwargs
        )

        model = HierarchicalCNNClassificationModel(experiment, d
evice=args.device)
        # load pretrained model
        model.load_state_dict(
            torch.load(
                os.path.join(
                    pretrained.checkpoints,
                    "fold_{}".format(args.pretrained_fold),
                    "best_model.pth"
                )
            )
        )

        scores = model.fit_validate(
            train_loader, valid_loader,
            epochs=experiment.config.train.epochs, fold=fold,
            log_interval=args.log_interval
        )

        best_metric = max(scores)
        experiment.register_result("fold{}.metric".format(fold),
 best_metric)

        torch.save(
            model.state_dict(),
            os.path.join(
                experiment.checkpoints,
                "fold_{}".format(fold),
                "final_model.pth")
        )

        # predictions
        model.load_best_model(fold)

        # validation

        val_preds = model.predict(valid_loader)
        val_predictions_df = pd.DataFrame(
```

```
            val_preds, columns=get_class_names_from_classmap(cla
ss_map))
        val_predictions_df["fname"] = train_df.fname[valid].valu
es
        val_predictions_df.to_csv(
            os.path.join(
                experiment.predictions,
                "val_preds_fold_{}.csv".format(fold)
            ),
            index=False
        )
        del val_predictions_df

        # test
        test_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.test_data_dir, fname)
                    for fname in test_df.fname.values],
                transform=Compose([
                    LoadAudio(),
                    STFT(n_fft=config.data.n_fft, hop_size=confi
g.data.hop_size),
                    DropFields(("audio", "filename", "sr")),
                    RenameFields({"stft": "signal"})
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": math.log(STFT.
eps)}),
            **loader_kwargs
        )

        test_preds = model.predict(test_loader)
        test_predictions_df = pd.DataFrame(
            test_preds, columns=get_class_names_from_classmap(cl
ass_map))
        test_predictions_df["fname"] = test_df.fname
        test_predictions_df.to_csv(
            os.path.join(
                experiment.predictions,
                "test_preds_fold_{}.csv".format(fold)
            ),
            index=False
        )
        del test_predictions_df

        # holdout
        if args.holdout_size:
            holdout_loader = torch.utils.data.DataLoader(
                SoundDataset(
                    audio_files=[
                        os.path.join(args.train_data_dir, fname)
                        for fname in holdout_df.fname.values],
                    labels=[item.split(",") for item in holdout_
```

```
df.labels.values],
                    transform=Compose([
                        LoadAudio(),
                        MapLabels(class_map),
                        STFT(n_fft=config.data.n_fft, hop_size=c
onfig.data.hop_size),
                        DropFields(("audio", "filename", "sr")),
                        RenameFields({"stft": "signal"})
                    ])
                ),
                shuffle=False,
                batch_size=config.train.batch_size,
                collate_fn=make_collate_fn({"signal": math.log(S
TFT.eps)}),
                **loader_kwargs
            )

            holdout_metric = model.evaluate(holdout_loader)
            experiment.register_result(
                "fold{}.holdout_metric".format(fold), holdout_me
tric)

            print("\nHoldout metric: {:.4f}".format(holdout_metr
ic))

        if args.device == "cuda":
            torch.cuda.empty_cache()

    # global metric

    if all(
        "fold{}".format(k) in experiment.results.to_dict()
        for k in range(config.data._n_folds)):

        val_df_files = [
            os.path.join(
                experiment.predictions,
                "val_preds_fold_{}.csv".format(fold)
            )
            for fold in range(config.data._n_folds)
        ]

        val_predictions_df = pd.concat([
            pd.read_csv(file) for file in val_df_files]).reset_i
ndex(drop=True)

        labels = np.asarray([
            item["labels"] for item in SoundDataset(
                audio_files=train_df.fname.tolist(),
                labels=[item.split(",") for item in train_df.lab
els.values],
                transform=MapLabels(class_map)
            )
        ])

        val_labels_df = pd.DataFrame(
```

```
            labels, columns=get_class_names_from_classmap(class_
map))
        val_labels_df["fname"] = train_df.fname

        assert set(val_predictions_df.fname) == set(val_labels_d
f.fname)

        val_predictions_df.sort_values(by="fname", inplace=True)
        val_labels_df.sort_values(by="fname", inplace=True)

        metric = lwlrap(
            val_labels_df.drop("fname", axis=1).values,
            val_predictions_df.drop("fname", axis=1).values
        )

        experiment.register_result("metric", metric)

    # submission

    test_df_files = [
        os.path.join(
            experiment.predictions,
            "test_preds_fold_{}.csv".format(fold)
        )
        for fold in range(config.data._n_folds)
    ]

    if all(os.path.isfile for file in test_df_files):
        test_dfs = [pd.read_csv(file) for file in test_df_files]
        submission_df = pd.DataFrame({"fname": test_dfs[0].fname
.values})
        for c in get_class_names_from_classmap(class_map):
            submission_df[c] = np.mean([d[c].values for d in tes
t_dfs], axis=0)
        submission_df.to_csv(
            os.path.join(experiment.predictions, "submission.csv
"), index=False)======
```

other entities that control, are controlled by, or are und
er common
control with that entity. For the purposes of this definit
ion,
"control" means (i) the power, direct or indirect, to caus
e the
direction or management of such entity, whether by contrac
t or
otherwise, or (ii) ownership of fifty percent (50%) or mor
e of the
outstanding shares, or (iii) beneficial ownership of such
entity.

"You" (or "Your") shall mean an individual or Legal Entity
exercising permissions granted by this License.

"Source" form shall mean the preferred form for making mod
ifications,
including but not limited to software source code, documen
tation
source, and configuration files.

"Object" form shall mean any form resulting from mechanica
l
transformation or translation of a Source form, including
but
not limited to compiled object code, generated documentati
on,
and conversions to other media types.

"Work" shall mean the work of authorship, whether in Sourc
e or
Object form, made available under the License, as indicate
d by a
copyright notice that is included in or attached to the wo
rk
(an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source
or Object
form, that is based on (or derived from) the Work and for
which the
editorial revisions, annotations, elaborations, or other m
odifications
represent, as a whole, an original work of authorship. For
the purposes
of this License, Derivative Works shall not include works
that remain
separable from, or merely link (or bind by name) to the in
terfaces of,
the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, includin
g
the original version of the Work and any modifications or
additions

```
      to that Work or Derivative Works thereof, that is intentio
nally
      submitted to Licensor for inclusion in the Work by the cop
yright owner
      or by an individual or Legal Entity authorized to submit o
n behalf of
      the copyright owner. For the purposes of this definition,
"submitted"
      means any form of electronic, verbal, or written communica
tion sent
      to the Licensor or its representatives, including but not
limited to
      communication on electronic mailing lists, source code con
trol systems,
      and issue tracking systems that are managed by, or on beha
lf of, the
      Licensor for the purpose of discussing and improving the W
ork, but
      excluding communication that is conspicuously marked or ot
herwise
      designated in writing by the copyright owner as "Not a Con
tribution."

      "Contributor" shall mean Licensor and any individual or Le
gal Entity
      on behalf of whom a Contribution has been received by Lice
nsor and
      subsequently incorporated within the Work.

   2. Grant of Copyright License. Subject to the terms and condi
tions of
      this License, each Contributor hereby grants to You a perp
etual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevoc
able
      copyright license to reproduce, prepare Derivative Works o
f,
      publicly display, publicly perform, sublicense, and distri
bute the
      Work and such Derivative Works in Source or Object form.

   3. Grant of Patent License. Subject to the terms and conditio
ns of
      this License, each Contributor hereby grants to You a perp
etual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevoc
able
      (except as stated in this section) patent license to make,
 have made,
      use, offer to sell, sell, import, and otherwise transfer t
he Work,
      where such license applies only to those patent claims lic
ensable
      by such Contributor that are necessarily infringed by thei
r
      Contribution(s) alone or by combination of their Contribut
```

```
ion(s)
      with the Work to which such Contribution(s) was submitted.
 If You
      institute patent litigation against any entity (including
a
      cross-claim or counterclaim in a lawsuit) alleging that th
e Work
      or a Contribution incorporated within the Work constitutes
 direct
      or contributory patent infringement, then any patent licen
ses
      granted to You under this License for that Work shall term
inate
      as of the date such litigation is filed.

   4. Redistribution. You may reproduce and distribute copies of
 the
      Work or Derivative Works thereof in any medium, with or wi
thout
      modifications, and in Source or Object form, provided that
 You
      meet the following conditions:

      (a) You must give any other recipients of the Work or
          Derivative Works a copy of this License; and

      (b) You must cause any modified files to carry prominent n
otices
          stating that You changed the files; and

      (c) You must retain, in the Source form of any Derivative
Works
          that You distribute, all copyright, patent, trademark,
 and
          attribution notices from the Source form of the Work,
          excluding those notices that do not pertain to any par
t of
          the Derivative Works; and

      (d) If the Work includes a "NOTICE" text file as part of i
ts
          distribution, then any Derivative Works that You distr
ibute must
          include a readable copy of the attribution notices con
tained
          within such NOTICE file, excluding those notices that
do not
          pertain to any part of the Derivative Works, in at lea
st one
          of the following places: within a NOTICE text file dis
tributed
          as part of the Derivative Works; within the Source for
m or
          documentation, if provided along with the Derivative W
orks; or,
          within a display generated by the Derivative Works, if
```

```
 and
          wherever such third-party notices normally appear. The
 contents
          of the NOTICE file are for informational purposes only
 and
          do not modify the License. You may add Your own attrib
ution
          notices within Derivative Works that You distribute, a
longside
          or as an addendum to the NOTICE text from the Work, pr
ovided
          that such additional attribution notices cannot be con
strued
          as modifying the License.

      You may add Your own copyright statement to Your modificat
ions and
      may provide additional or different license terms and cond
itions
      for use, reproduction, or distribution of Your modificatio
ns, or
      for any such Derivative Works as a whole, provided Your us
e,
      reproduction, and distribution of the Work otherwise compl
ies with
      the conditions stated in this License.

   5. Submission of Contributions. Unless You explicitly state o
therwise,
      any Contribution intentionally submitted for inclusion in
the Work
      by You to the Licensor shall be under the terms and condit
ions of
      this License, without any additional terms or conditions.
      Notwithstanding the above, nothing herein shall supersede
or modify
      the terms of any separate license agreement you may have e
xecuted
      with Licensor regarding such Contributions.

   6. Trademarks. This License does not grant permission to use
the trade
      names, trademarks, service marks, or product names of the
Licensor,
      except as required for reasonable and customary use in des
cribing the
      origin of the Work and reproducing the content of the NOTI
CE file.

   7. Disclaimer of Warranty. Unless required by applicable law
or
      agreed to in writing, Licensor provides the Work (and each
      Contributor provides its Contributions) on an "AS IS" BASI
S,
      WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either expre
ss or
```

implied, including, without limitation, any warranties or conditions
of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
PARTICULAR PURPOSE. You are solely responsible for determining the
appropriateness of using or redistributing the Work and assume any
risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
whether in tort (including negligence), contract, or otherwise,
unless required by applicable law (such as deliberate and grossly
negligent acts) or agreed to in writing, shall any Contributor be
liable to You for damages, including any direct, indirect, special,
incidental, or consequential damages of any character arising as a
result of this License or out of the use or inability to use the
Work (including but not limited to damages for loss of goodwill,
work stoppage, computer failure or malfunction, or any and all
other commercial damages or losses), even if such Contributor
has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
the Work or Derivative Works thereof, You may choose to offer,
and charge a fee for, acceptance of support, warranty, indemnity,
or other liability obligations and/or rights consistent with this
License. However, in accepting such obligations, You may act only
on Your own behalf and on Your sole responsibility, not on behalf
of any other Contributor, and only if You agree to indemnify,
defend, and hold each Contributor harmless for any liability
incurred by, or claims asserted against, such Contributor by reason
of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

```
   APPENDIX: How to apply the Apache License to your work.

      To apply the Apache License to your work, attach the follo
wing
      boilerplate notice, with the fields enclosed by brackets "
[]"
      replaced with your own identifying information. (Don't inc
lude
      the brackets!)  The text should be enclosed in the appropr
iate
      comment syntax for the file format. We also recommend that
 a
      file or class name and description of purpose be included
on the
      same "printed page" as the copyright notice for easier
      identification within third-party archives.

   Copyright [yyyy] [name of copyright owner]

   Licensed under the Apache License, Version 2.0 (the "License"
);
   you may not use this file except in compliance with the Licen
se.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, so
ftware
   distributed under the License is distributed on an "AS IS" BA
SIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
or implied.
   See the License for the specific language governing permissio
ns and
   limitations under the License.
======
import argparse
import glob
from pathlib import Path

import pandas as pd
import numpy as np
import scipy.optimize
from scipy.stats import rankdata
from mag.utils import blue, green, bold

from ops.utils import lwlrap


parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--experiments", type=str, required=True, nargs="+",
```

```
    help="experiments to blend"
)
parser.add_argument(
    "--train_df", type=str, required=True,
    help="path to train df"
)
parser.add_argument(
    "--rankdata", action="store_true", default=False,
    help="whether to use ranks instead of raw scores"
)
parser.add_argument(
    "--output_df", type=str, required=True,
    help="where to save test submission"
)

args = parser.parse_args()


n = len(args.experiments)


def load_predictions(experiment):

    prediction_files = (
        "experiments" / Path(experiment) / "predictions").glob("
val_preds*")
    dfs = [pd.read_csv(f) for f in prediction_files]
    df = pd.concat(dfs).reset_index(drop=True)
    df = df.sort_values(by="fname")
    df = df[sorted(df.columns.tolist())]
    return df


def to_ranks(values):
    return np.array([rankdata(r) for r in values])


predictions = [load_predictions(exp) for exp in args.experiments
]
class_cols = predictions[0].columns.drop("fname")
prediction_values = [p[class_cols].values for p in predictions]
if args.rankdata:
    prediction_values = [to_ranks(p) for p in prediction_values]

train_df = pd.read_csv(args.train_df)


def make_actual_labels(train_df):

    classname_to_idx = dict((c, i) for i, c in enumerate(class_c
ols))
    actual_labels = np.zeros((len(train_df), len(class_cols)), d
type=np.float32)
    for k in range(train_df.labels.values.size):
        for label in str(train_df.labels.values[k]).split(","):
            actual_labels[k, classname_to_idx[label]] = 1
```

```
    return actual_labels


actual_labels = make_actual_labels(train_df)


def constraints():
    A = np.ones(n)
    yield scipy.optimize.LinearConstraint(A=A, lb=0.01, ub=0.99)
    for k in range(n):
        A = np.zeros(n)
        A[k] = 1
        yield scipy.optimize.LinearConstraint(A=A, lb=0, ub=1)


def initial():
    return np.ones(n) / n

def target(alphas, *args):
    prediction = np.sum([a * p for a, p in zip(alphas, predictio
n_values)], axis=0)
    return -lwlrap(actual_labels, prediction)


alphas = scipy.optimize.minimize(
    target,
    initial(),
    constraints=list(constraints()),
    method="COBYLA").x

print()
for experiment, alpha in zip(args.experiments, alphas):
    print("{}: {}".format(green(bold(experiment)), blue(bold(alp
ha))))

print()
print("Final lwlrap:", bold(green(-target(alphas))))


def load_test_predictions(experiment):

    prediction_files = (
        "experiments" / Path(experiment) / "predictions").glob("
test_preds*")
    dfs = [pd.read_csv(f) for f in prediction_files]
    dfs = [df.sort_values(by="fname") for df in dfs]
    return dfs


test_preds = []

for alpha, exp in zip(alphas, args.experiments):
    experiment_test_predictions = load_test_predictions(experime
nt)
    for p in experiment_test_predictions:
```

```
        if args.rankdata:
            test_preds.append(to_ranks(p[class_cols].values) * a
lpha)
        else:
            test_preds.append(p[class_cols].values * alpha)

test_preds = np.sum(test_preds, 0)

sub = pd.DataFrame(test_preds, columns=class_cols)
sub["fname"] = p.fname

sub.to_csv(args.output_df, index=False)======
import os
import gc
import argparse
import json
import math
from functools import partial

import pandas as pd
import numpy as np
import torch
from mag.experiment import Experiment
from mag.utils import green, bold
import mag

from datasets.sound_dataset import SoundDataset
from networks.classifiers import TwoDimensionalCNNClassification
Model
from ops.folds import train_validation_data_stratified
from ops.transforms import (
    Compose, DropFields, LoadAudio,
    AudioFeatures, MapLabels, RenameFields,
    MixUp, SampleSegment, SampleLongAudio,
    AudioAugmentation, FlipAudio, ShuffleAudio)
from ops.utils import load_json, get_class_names_from_classmap,
lwlrap
from ops.padding import make_collate_fn

mag.use_custom_separator("-")

parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--experiment", type=str, required=True,
    help="path to an experiment"
)
parser.add_argument(
    "--test_df", required=True, type=str,
    help="path to test dataframe"
)
parser.add_argument(
    "--output_df", required=True, type=str,
    help="where to save resulting dataframe"
```

```
)
parser.add_argument(
    "--test_data_dir", required=True, type=str,
    help="path to test data directory"
)
parser.add_argument(
    "--classmap", required=True, type=str,
    help="path to class map json"
)
parser.add_argument(
    "--batch_size", type=int, default=32,
    help="batch size used for prediction"
)
parser.add_argument(
    "--device", type=str, required=True,
    help="whether to train on cuda or cpu",
    choices=("cuda", "cpu")
)
parser.add_argument(
    "--num_workers", type=int, default=4,
    help="number of workers for data loader",
)

args = parser.parse_args()

class_map = load_json(args.classmap)

test_df = pd.read_csv(args.test_df)

with Experiment(resume_from=args.experiment) as experiment:

    config = experiment.config

    audio_transform = AudioFeatures(config.data.features)

    all_predictions = np.zeros(
        shape=(len(test_df), len(class_map)), dtype=np.float32)

    for fold in range(config.data._n_folds):

        print("\n\n   -----  Fold {}\n".format(fold))

        loader_kwargs = (
            {"num_workers": args.num_workers, "pin_memory": True
}
            if torch.cuda.is_available() else {})

        test_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.test_data_dir, fname)
                    for fname in test_df.fname.values],
                labels=None,
                transform=Compose([
                    LoadAudio(),
                    audio_transform,
```

```
                    DropFields(("audio", "filename", "sr")),
                ]),
                clean_transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                ]),
            ),
            shuffle=False,
            batch_size=args.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        model = TwoDimensionalCNNClassificationModel(
                experiment, device=args.device)
        model.load_best_model(fold)
        model.eval()

        val_preds = model.predict(test_loader)

        all_predictions += val_preds / config.data._n_folds


result = pd.DataFrame(
    all_predictions, columns=get_class_names_from_classmap(class
_map))
result["fname"] = test_df.fname

result.to_csv(args.output_df, index=False)======
import os
import gc
import argparse
import json
import math
from functools import partial

from scipy.sparse import csr_matrix
from scipy.stats import rankdata

import pandas as pd
import numpy as np

parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--noisy_df", required=True, type=str,
    help="path to noisy dataframe"
)
parser.add_argument(
    "--noisy_predictions_df", required=True, type=str,
    help="path to noisy predictions"
)
parser.add_argument(
```

```
    "--output_df", required=True, type=str,
    help="where to save relabeled dataframe"
)
parser.add_argument(
    "--mode", required=True, type=str,
    help="relabeling strategy"
)

args = parser.parse_args()

noisy_df = pd.read_csv(args.noisy_df)
noisy_predictions_df = pd.read_csv(args.noisy_predictions_df)

noisy_df.sort_values(by="fname", inplace=True)
noisy_predictions_df.sort_values(by="fname", inplace=True)

mode, *params = args.mode.split("_")

class_cols = noisy_predictions_df.columns.drop("fname").values
classname_to_idx = dict((c, i) for i, c in enumerate(class_cols)
)
idx_to_classname = dict(enumerate(class_cols))
noisy_labels = np.zeros((len(noisy_df), len(class_cols)), dtype=
np.float32)
for k in range(noisy_df.labels.values.size):
    for label in str(noisy_df.labels.values[k]).split(","):
        noisy_labels[k, classname_to_idx[label]] = 1


def binary_to_labels(binary):
    labels = []
    for row in binary:
        labels.append(",".join(idx_to_classname[k] for k in nonz
ero(row)))

    return labels


def find_threshold(probs, expected_classes_per_sample):

        thresholds = np.linspace(0, 1, 10000)
        classes_per_sample = np.zeros_like(thresholds)

        for k in range(thresholds.size):
            c = (probs > thresholds[k]).sum(-1).mean()
            classes_per_sample[k] = c

        k = np.argmin(np.abs(classes_per_sample - expected_class
es_per_sample))

        return thresholds[k]

def nonzero(x):
    return np.nonzero(x)[0]
```

```
def merge_labels(first, second):

    merged = []
    for f, s in zip(first, second):
        m = set(f.split(",")) | set(s.split(","))
        if "" in m:
            m.remove("")
        merged.append(",".join(m))

    return merged


def score_samples(y_true, y_score):
    scores = []

    y_true = csr_matrix(y_true)
    y_score = -y_score

    n_samples, n_labels = y_true.shape

    for i, (start, stop) in enumerate(zip(y_true.indptr, y_true.
indptr[1:])):
        relevant = y_true.indices[start:stop]

        if (relevant.size == 0 or relevant.size == n_labels):
            # If all labels are relevant or unrelevant, the scor
e is also
            # equal to 1. The label ranking has no meaning.
            aux = 1.
        else:
            scores_i = y_score[i]
            rank = rankdata(scores_i, 'max')[relevant]
            L = rankdata(scores_i[relevant], 'max')
            aux = (L / rank).mean()

        scores.append(aux)

    return np.array(scores)


if mode == "fullmatch":

    expected_classes_per_sample, = params
    expected_classes_per_sample = float(expected_classes_per_sam
ple)

    probs = noisy_predictions_df[class_cols].values
    threshold = find_threshold(probs, expected_classes_per_sampl
e)
    binary = probs > threshold

    match = (binary == noisy_labels).all(-1)

    relabeled = noisy_df[match]

elif mode == "relabelall":
```

```
    expected_classes_per_sample, = params
    expected_classes_per_sample = float(expected_classes_per_sam
ple)

    probs = noisy_predictions_df[class_cols].values
    threshold = find_threshold(probs, expected_classes_per_sampl
e)
    binary = probs > threshold

    new_labels = binary_to_labels(binary)

    noisy_df.labels = new_labels
    noisy_df = noisy_df[noisy_df.labels != ""]

    relabeled = noisy_df

elif mode == "relabelall-replacenan":

    expected_classes_per_sample, = params
    expected_classes_per_sample = float(expected_classes_per_sam
ple)

    probs = noisy_predictions_df[class_cols].values
    threshold = find_threshold(probs, expected_classes_per_sampl
e)
    binary = probs > threshold

    new_labels = pd.Series(binary_to_labels(binary))
    where_non_empty = (new_labels != "")
    noisy_df = noisy_df[where_non_empty]
    noisy_df.labels = new_labels[where_non_empty]

    relabeled = noisy_df

elif mode == "relabelall-merge":

    expected_classes_per_sample, = params
    expected_classes_per_sample = float(expected_classes_per_sam
ple)

    probs = noisy_predictions_df[class_cols].values
    threshold = find_threshold(probs, expected_classes_per_sampl
e)
    binary = probs > threshold

    new_labels = binary_to_labels(binary)
    noisy_df.labels = merge_labels(noisy_df.labels.values, new_l
abels)

    relabeled = noisy_df

elif mode == "scoring":

    topk, = params
    topk = int(topk)
```

```
    probs = noisy_predictions_df[class_cols].values
    scores = score_samples(noisy_labels, probs)

    selection = np.argsort(-scores)[:topk]

    relabeled = noisy_df.iloc[selection]

print("Relabeled df shape:", relabeled.shape)

relabeled.to_csv(args.output_df, index=False)
======
import os
import gc
import argparse
import json
import math
from functools import partial

import pandas as pd
import numpy as np
import torch
from mag.experiment import Experiment
import mag
from sklearn.model_selection import train_test_split

from datasets.sound_dataset import SoundDataset
from networks.classifiers import TwoDimensionalCNNClassification
Model
from ops.folds import train_validation_data, train_validation_da
ta_stratified
from ops.transforms import (
    Compose, DropFields, LoadAudio,
    AudioFeatures, MapLabels, RenameFields,
    MixUp, SampleSegment, SampleLongAudio,
    AudioAugmentation, ShuffleAudio, CutOut, Identity)
from ops.utils import load_json, get_class_names_from_classmap,
lwlrap
from ops.padding import make_collate_fn

torch.manual_seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(42)

mag.use_custom_separator("-")

parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--train_df", required=True, type=str,
    help="path to train dataframe"
)
parser.add_argument(
    "--train_data_dir", required=True, type=str,
```

```
    help="path to train data"
)
parser.add_argument(
    "--noisy_train_df", type=str,
    help="path to noisy train dataframe (optional)"
)
parser.add_argument(
    "--noisy_train_data_dir", type=str,
    help="path to noisy train data (optional)"
)
parser.add_argument(
    "--share_noisy", action="store_true", default=False,
    help="whether to share noisy files across folds"
)
parser.add_argument(
    "--resume", action="store_true", default=False,
    help="allow resuming even if experiment exists"
)
parser.add_argument(
    "--test_data_dir", required=True, type=str,
    help="path to test data"
)
parser.add_argument(
    "--sample_submission", required=True, type=str,
    help="path sample submission"
)
parser.add_argument(
    "--classmap", required=True, type=str,
    help="path to class map json"
)
parser.add_argument(
    "--log_interval", default=10, type=int,
    help="how frequently to log batch metrics"
    "in terms of processed batches"
)
parser.add_argument(
    "--batch_size", type=int, default=64,
    help="minibatch size"
)
parser.add_argument(
    "--max_audio_length", type=int, default=10,
    help="max audio length in seconds. For longer clips are samp
led"
)
parser.add_argument(
    "--lr", default=0.01, type=float,
    help="starting learning rate"
)
parser.add_argument(
    "--max_samples", type=int,
    help="maximum number of samples to use"
)
parser.add_argument(
    "--holdout_size", type=float, default=0.0,
    help="size of holdout set"
)
```

```
parser.add_argument(
    "--epochs", default=100, type=int,
    help="number of epochs to train"
)
parser.add_argument(
    "--scheduler", type=str, default="steplr_1_0.5",
    help="scheduler type",
)
parser.add_argument(
    "--accumulation_steps", type=int, default=1,
    help="number of gradient accumulation steps",
)
parser.add_argument(
    "--save_every", type=int, default=1,
    help="how frequently to save a model",
)
parser.add_argument(
    "--device", type=str, required=True,
    help="whether to train on cuda or cpu",
    choices=("cuda", "cpu")
)
parser.add_argument(
    "--aggregation_type", type=str, required=True,
    help="how to aggregate outputs",
    choices=("max", "rnn")
)
parser.add_argument(
    "--num_conv_blocks", type=int, default=5,
    help="number of conv blocks"
)
parser.add_argument(
    "--start_deep_supervision_on", type=int, default=2,
    help="from which layer to start aggregating features for cla
ssification"
)
parser.add_argument(
    "--conv_base_depth", type=int, default=64,
    help="base depth for conv layers"
)
parser.add_argument(
    "--growth_rate", type=float, default=2,
    help="how quickly to increase the number of units as a funct
ion of layer"
)
parser.add_argument(
    "--weight_decay", type=float, default=1e-5,
    help="weight decay"
)
parser.add_argument(
    "--output_dropout", type=float, default=0.0,
    help="output dropout"
)
parser.add_argument(
    "--p_mixup", type=float, default=0.0,
    help="probability of the mixup augmentation"
)
```

```
parser.add_argument(
    "--p_aug", type=float, default=0.0,
    help="probability of audio augmentation"
)
parser.add_argument(
    "--switch_off_augmentations_on", type=int, default=20,
    help="on which epoch to remove augmentations"
)
parser.add_argument(
    "--features", type=str, required=True,
    help="feature descriptor"
)
parser.add_argument(
    "--optimizer", type=str, required=True,
    help="which optimizer to use",
    choices=("adam", "momentum")
)
parser.add_argument(
    "--folds", type=int, required=True, nargs="+",
    help="which folds to use"
)
parser.add_argument(
    "--n_folds", type=int, default=4,
    help="number of folds"
)
parser.add_argument(
    "--kfold_seed", type=int, default=42,
    help="kfold seed"
)
parser.add_argument(
    "--num_workers", type=int, default=4,
    help="number of workers for data loader",
)
parser.add_argument(
    "--label", type=str, default="2d_cnn",
    help="optional label",
)
args = parser.parse_args()

class_map = load_json(args.classmap)

audio_transform = AudioFeatures(args.features)

with Experiment({
    "network": {
        "num_conv_blocks": args.num_conv_blocks,
        "start_deep_supervision_on": args.start_deep_supervision
_on,
        "conv_base_depth": args.conv_base_depth,
        "growth_rate": args.growth_rate,
        "output_dropout": args.output_dropout,
        "aggregation_type": args.aggregation_type
    },
    "data": {
        "features": args.features,
        "_n_folds": args.n_folds,
```

```
        "_kfold_seed": args.kfold_seed,
        "_input_dim": audio_transform.n_features,
        "_n_classes": len(class_map),
        "_holdout_size": args.holdout_size,
        "p_mixup": args.p_mixup,
        "p_aug": args.p_aug,
        "max_audio_length": args.max_audio_length,
        "noisy": args.noisy_train_df is not None,
        "_train_df": args.train_df,
        "_train_data_dir": args.train_data_dir,
        "_noisy_train_df": args.noisy_train_df,
        "_noisy_train_data_dir": args.noisy_train_data_dir,
        "_share_noisy": args.share_noisy
    },
    "train": {
        "accumulation_steps": args.accumulation_steps,
        "batch_size": args.batch_size,
        "learning_rate": args.lr,
        "scheduler": args.scheduler,
        "optimizer": args.optimizer,
        "epochs": args.epochs,
        "_save_every": args.save_every,
        "weight_decay": args.weight_decay,
        "switch_off_augmentations_on": args.switch_off_augmentat
ions_on
    },
    "label": args.label
}, implicit_resuming=args.resume) as experiment:

    config = experiment.config
    print()
    print("     ////// CONFIG //////")
    print(experiment.config)

    train_df = pd.read_csv(args.train_df)
    test_df = pd.read_csv(args.sample_submission)

    if args.noisy_train_df:
        noisy_train_df = pd.read_csv(args.noisy_train_df)

    if args.max_samples:
        train_df = train_df.sample(args.max_samples).reset_index
(drop=True)
        test_df = test_df.sample(
            min(args.max_samples, len(test_df))).reset_index(dro
p=True)

    if args.holdout_size:
        keep, holdout = train_test_split(
            np.arange(len(train_df)), test_size=args.holdout_siz
e,
            random_state=args.kfold_seed)
        holdout_df = train_df.iloc[holdout].reset_index(drop=Tru
e)
        train_df = train_df.iloc[keep].reset_index(drop=True)
```

```
    splits = list(train_validation_data_stratified(
        train_df.fname, train_df.labels, class_map,
        config.data._n_folds, config.data._kfold_seed))

    if args.noisy_train_df:
        noisy_splits = list(train_validation_data(
            noisy_train_df.fname, noisy_train_df.labels,
            config.data._n_folds, config.data._kfold_seed))

    for fold in args.folds:

        print("\n\n   -----  Fold {}\n".format(fold))

        train, valid = splits[fold]

        loader_kwargs = (
            {"num_workers": args.num_workers, "pin_memory": True
}
            if torch.cuda.is_available() else {})

        experiment.register_directory("checkpoints")
        experiment.register_directory("predictions")

        if args.noisy_train_df:

            noisy_train, noisy_valid = noisy_splits[fold]

            if config.data._share_noisy:
                noisy_audio_files = [
                    os.path.join(args.noisy_train_data_dir, fnam
e)
                    for fname in noisy_train_df.fname.values]
                noisy_labels = [
                    item.split(",") for item in
                    noisy_train_df.labels.values]
            else:
                noisy_audio_files = [
                    os.path.join(args.noisy_train_data_dir, fnam
e)
                    for fname in noisy_train_df.fname.values[noi
sy_valid]]
                noisy_labels = [
                    item.split(",") for item in
                    noisy_train_df.labels.values[noisy_valid]]
        else:
            noisy_audio_files = []
            noisy_labels = []

        train_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[train]] +
 noisy_audio_files,
                labels=[
                    item.split(",") for item in
```

```
                    train_df.labels.values[train]] + noisy_label
s,
                is_noisy=[0] * len(train) + [1] * len(noisy_labe
ls),
                transform=Compose([
                    LoadAudio(),
                    SampleLongAudio(max_length=args.max_audio_le
ngth),
                    MapLabels(class_map=class_map),
                    (
                        ShuffleAudio(chunk_length=0.5, p=0.5)
                        if config.network.aggregation_type != "r
nn" else Identity()
                    ),
                    MixUp(p=args.p_mixup),
                    AudioAugmentation(p=args.p_aug),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ]),
                clean_transform=Compose([
                    LoadAudio(),
                    SampleLongAudio(max_length=args.max_audio_le
ngth),
                    MapLabels(class_map=class_map),
                ])
            ),
            shuffle=True,
            drop_last=True,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        valid_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[valid]],
                labels=[item.split(",") for item in train_df.lab
els.values[valid]],
                transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        model = TwoDimensionalCNNClassificationModel(
```

```
            experiment, device=args.device)

        scores = model.fit_validate(
            train_loader, valid_loader,
            epochs=experiment.config.train.epochs, fold=fold,
            log_interval=args.log_interval
        )

        best_metric = max(scores)
        experiment.register_result("fold{}.metric".format(fold),
 best_metric)

        torch.save(
            model.state_dict(),
            os.path.join(
                experiment.checkpoints,
                "fold_{}".format(fold),
                "final_model.pth")
        )

        # predictions
        model.load_best_model(fold)

        # validation

        val_preds = model.predict(valid_loader)
        val_predictions_df = pd.DataFrame(
            val_preds, columns=get_class_names_from_classmap(cla
ss_map))
        val_predictions_df["fname"] = train_df.fname[valid].valu
es
        val_predictions_df.to_csv(
            os.path.join(
                experiment.predictions,
                "val_preds_fold_{}.csv".format(fold)
            ),
            index=False
        )
        del val_predictions_df

        # test
        test_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.test_data_dir, fname)
                    for fname in test_df.fname.values],
                transform=Compose([
                    LoadAudio(),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
```

```
            **loader_kwargs
        )

        test_preds = model.predict(test_loader)
        test_predictions_df = pd.DataFrame(
            test_preds, columns=get_class_names_from_classmap(cl
ass_map))
        test_predictions_df["fname"] = test_df.fname
        test_predictions_df.to_csv(
            os.path.join(
                experiment.predictions,
                "test_preds_fold_{}.csv".format(fold)
            ),
            index=False
        )
        del test_predictions_df

        # holdout
        if args.holdout_size:
            holdout_loader = torch.utils.data.DataLoader(
                SoundDataset(
                    audio_files=[
                        os.path.join(args.train_data_dir, fname)
                        for fname in holdout_df.fname.values],
                    labels=[item.split(",") for item in holdout_
df.labels.values],
                    transform=Compose([
                        LoadAudio(),
                        MapLabels(class_map),
                        audio_transform,
                        DropFields(("audio", "filename", "sr")),
                    ])
                ),
                shuffle=False,
                batch_size=config.train.batch_size,
                collate_fn=make_collate_fn({"signal": audio_tran
sform.padding_value}),
                **loader_kwargs
            )

            holdout_metric = model.evaluate(holdout_loader)
            experiment.register_result(
                "fold{}.holdout_metric".format(fold), holdout_me
tric)

            print("\nHoldout metric: {:.4f}".format(holdout_metr
ic))

        if args.device == "cuda":
            torch.cuda.empty_cache()

    # global metric

    if all(
        "fold{}".format(k) in experiment.results.to_dict()
        for k in range(config.data._n_folds)):
```

```
        val_df_files = [
            os.path.join(
                experiment.predictions,
                "val_preds_fold_{}.csv".format(fold)
            )
            for fold in range(config.data._n_folds)
        ]

        val_predictions_df = pd.concat([
            pd.read_csv(file) for file in val_df_files]).reset_i
ndex(drop=True)

        labels = np.asarray([
            item["labels"] for item in SoundDataset(
                audio_files=train_df.fname.tolist(),
                labels=[item.split(",") for item in train_df.lab
els.values],
                transform=MapLabels(class_map)
            )
        ])

        val_labels_df = pd.DataFrame(
            labels, columns=get_class_names_from_classmap(class_
map))
        val_labels_df["fname"] = train_df.fname

        assert set(val_predictions_df.fname) == set(val_labels_d
f.fname)

        val_predictions_df.sort_values(by="fname", inplace=True)
        val_labels_df.sort_values(by="fname", inplace=True)

        metric = lwlrap(
            val_labels_df.drop("fname", axis=1).values,
            val_predictions_df.drop("fname", axis=1).values
        )

        experiment.register_result("metric", metric)

    # submission

    test_df_files = [
        os.path.join(
            experiment.predictions,
            "test_preds_fold_{}.csv".format(fold)
        )
        for fold in range(config.data._n_folds)
    ]

    if all(os.path.isfile for file in test_df_files):
        test_dfs = [pd.read_csv(file) for file in test_df_files]
        submission_df = pd.DataFrame({"fname": test_dfs[0].fname
.values})
        for c in get_class_names_from_classmap(class_map):
            submission_df[c] = np.mean([d[c].values for d in tes
```

```
t_dfs], axis=0)
        submission_df.to_csv(
            os.path.join(experiment.predictions, "submission.csv
"), index=False)======
import os
import gc
import argparse
import json
import math
from functools import partial

import pandas as pd
import numpy as np
import torch
from mag.experiment import Experiment
import mag
from sklearn.model_selection import train_test_split

from datasets.sound_dataset import SoundDataset
from networks.classifiers import CNNBackboneClassificationModel
from ops.folds import train_validation_data, train_validation_da
ta_stratified
from ops.transforms import (
    Compose, DropFields, LoadAudio,
    AudioFeatures, MapLabels, RenameFields,
    MixUp, SampleSegment, SampleLongAudio,
    AudioAugmentation, ShuffleAudio, CutOut, Identity)
from ops.utils import load_json, get_class_names_from_classmap,
lwlrap
from ops.padding import make_collate_fn

torch.manual_seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(42)

mag.use_custom_separator("-")

parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--train_df", required=True, type=str,
    help="path to train dataframe"
)
parser.add_argument(
    "--train_data_dir", required=True, type=str,
    help="path to train data"
)
parser.add_argument(
    "--noisy_train_df", type=str,
    help="path to noisy train dataframe (optional)"
)
parser.add_argument(
    "--noisy_train_data_dir", type=str,
    help="path to noisy train data (optional)"
```

```
)
parser.add_argument(
    "--share_noisy", action="store_true", default=False,
    help="whether to share noisy files across folds"
)
parser.add_argument(
    "--resume", action="store_true", default=False,
    help="allow resuming even if experiment exists"
)
parser.add_argument(
    "--test_data_dir", required=True, type=str,
    help="path to test data"
)
parser.add_argument(
    "--sample_submission", required=True, type=str,
    help="path sample submission"
)
parser.add_argument(
    "--classmap", required=True, type=str,
    help="path to class map json"
)
parser.add_argument(
    "--log_interval", default=10, type=int,
    help="how frequently to log batch metrics"
    "in terms of processed batches"
)
parser.add_argument(
    "--batch_size", type=int, default=64,
    help="minibatch size"
)
parser.add_argument(
    "--max_audio_length", type=int, default=10,
    help="max audio length in seconds. For longer clips are samp
led"
)
parser.add_argument(
    "--lr", default=0.01, type=float,
    help="starting learning rate"
)
parser.add_argument(
    "--max_samples", type=int,
    help="maximum number of samples to use"
)
parser.add_argument(
    "--holdout_size", type=float, default=0.0,
    help="size of holdout set"
)
parser.add_argument(
    "--epochs", default=100, type=int,
    help="number of epochs to train"
)
parser.add_argument(
    "--scheduler", type=str, default="steplr_1_0.5",
    help="scheduler type",
)
parser.add_argument(
```

```
        "--accumulation_steps", type=int, default=1,
        help="number of gradient accumulation steps",
    )
    parser.add_argument(
        "--save_every", type=int, default=1,
        help="how frequently to save a model",
    )
    parser.add_argument(
        "--device", type=str, required=True,
        help="whether to train on cuda or cpu",
        choices=("cuda", "cpu")
    )
    parser.add_argument(
        "--backbone", type=str, required=True,
        help="which backbone to use",
        choices=("resnet18", "resnet34")
    )
    parser.add_argument(
        "--weight_decay", type=float, default=1e-5,
        help="weight decay"
    )
    parser.add_argument(
        "--output_dropout", type=float, default=0.0,
        help="output dropout"
    )
    parser.add_argument(
        "--p_mixup", type=float, default=0.0,
        help="probability of the mixup augmentation"
    )
    parser.add_argument(
        "--p_aug", type=float, default=0.0,
        help="probability of audio augmentation"
    )
    parser.add_argument(
        "--switch_off_augmentations_on", type=int, default=20,
        help="on which epoch to remove augmentations"
    )
    parser.add_argument(
        "--features", type=str, required=True,
        help="feature descriptor"
    )
    parser.add_argument(
        "--optimizer", type=str, required=True,
        help="which optimizer to use",
        choices=("adam", "momentum")
    )
    parser.add_argument(
        "--folds", type=int, required=True, nargs="+",
        help="which folds to use"
    )
    parser.add_argument(
        "--n_folds", type=int, default=4,
        help="number of folds"
    )
    parser.add_argument(
        "--kfold_seed", type=int, default=42,
```

```
    help="kfold seed"
)
parser.add_argument(
    "--num_workers", type=int, default=4,
    help="number of workers for data loader",
)
parser.add_argument(
    "--label", type=str, default="backbone",
    help="optional label",
)
args = parser.parse_args()

class_map = load_json(args.classmap)

audio_transform = AudioFeatures(args.features)

with Experiment({
    "network": {
        "backbone": args.backbone,
        "output_dropout": args.output_dropout,
    },
    "data": {
        "features": args.features,
        "_n_folds": args.n_folds,
        "_kfold_seed": args.kfold_seed,
        "_input_dim": audio_transform.n_features,
        "_n_classes": len(class_map),
        "_holdout_size": args.holdout_size,
        "p_mixup": args.p_mixup,
        "p_aug": args.p_aug,
        "max_audio_length": args.max_audio_length,
        "noisy": args.noisy_train_df is not None,
        "_train_df": args.train_df,
        "_train_data_dir": args.train_data_dir,
        "_noisy_train_df": args.noisy_train_df,
        "_noisy_train_data_dir": args.noisy_train_data_dir,
        "_share_noisy": args.share_noisy
    },
    "train": {
        "accumulation_steps": args.accumulation_steps,
        "batch_size": args.batch_size,
        "learning_rate": args.lr,
        "scheduler": args.scheduler,
        "optimizer": args.optimizer,
        "epochs": args.epochs,
        "_save_every": args.save_every,
        "weight_decay": args.weight_decay,
        "switch_off_augmentations_on": args.switch_off_augmentat
ions_on
    },
    "label": args.label
}, implicit_resuming=args.resume) as experiment:

    config = experiment.config
    print()
    print("     ////// CONFIG //////")
```

```
    print(experiment.config)

    train_df = pd.read_csv(args.train_df)
    test_df = pd.read_csv(args.sample_submission)

    if args.noisy_train_df:
        noisy_train_df = pd.read_csv(args.noisy_train_df)

    if args.max_samples:
        train_df = train_df.sample(args.max_samples).reset_index
(drop=True)
        test_df = test_df.sample(
            min(args.max_samples, len(test_df))).reset_index(dro
p=True)

    if args.holdout_size:
        keep, holdout = train_test_split(
            np.arange(len(train_df)), test_size=args.holdout_siz
e,
            random_state=args.kfold_seed)
        holdout_df = train_df.iloc[holdout].reset_index(drop=Tru
e)
        train_df = train_df.iloc[keep].reset_index(drop=True)

    splits = list(train_validation_data_stratified(
        train_df.fname, train_df.labels, class_map,
        config.data._n_folds, config.data._kfold_seed))

    if args.noisy_train_df:
        noisy_splits = list(train_validation_data(
            noisy_train_df.fname, noisy_train_df.labels,
            config.data._n_folds, config.data._kfold_seed))

    for fold in args.folds:

        print("\n\n   -----  Fold {}\n".format(fold))

        train, valid = splits[fold]

        loader_kwargs = (
            {"num_workers": args.num_workers, "pin_memory": True
}
            if torch.cuda.is_available() else {})

        experiment.register_directory("checkpoints")
        experiment.register_directory("predictions")

        if args.noisy_train_df:

            noisy_train, noisy_valid = noisy_splits[fold]

            if config.data._share_noisy:
                noisy_audio_files = [
                    os.path.join(args.noisy_train_data_dir, fnam
e)
                    for fname in noisy_train_df.fname.values]
```

```
                noisy_labels = [
                    item.split(",") for item in
                    noisy_train_df.labels.values]
            else:
                noisy_audio_files = [
                    os.path.join(args.noisy_train_data_dir, fnam
e)
                    for fname in noisy_train_df.fname.values[noi
sy_valid]]
                noisy_labels = [
                    item.split(",") for item in
                    noisy_train_df.labels.values[noisy_valid]]
        else:
            noisy_audio_files = []
            noisy_labels = []

        train_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[train]] +
 noisy_audio_files,
                labels=[
                    item.split(",") for item in
                    train_df.labels.values[train]] + noisy_label
s,
                is_noisy=[0] * len(train) + [1] * len(noisy_labe
ls),
                transform=Compose([
                    LoadAudio(),
                    SampleLongAudio(max_length=args.max_audio_le
ngth),
                    MapLabels(class_map=class_map),
                    ShuffleAudio(chunk_length=0.5, p=0.5),
                    MixUp(p=args.p_mixup),
                    AudioAugmentation(p=args.p_aug),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ]),
                clean_transform=Compose([
                    LoadAudio(),
                    SampleLongAudio(max_length=args.max_audio_le
ngth),
                    MapLabels(class_map=class_map),
                ])
            ),
            shuffle=True,
            drop_last=True,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        valid_loader = torch.utils.data.DataLoader(
            SoundDataset(
```

```
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[valid]],
                labels=[item.split(",") for item in train_df.lab
els.values[valid]],
                transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        model = CNNBackboneClassificationModel(experiment, devic
e=args.device)

        scores = model.fit_validate(
            train_loader, valid_loader,
            epochs=experiment.config.train.epochs, fold=fold,
            log_interval=args.log_interval
        )

        best_metric = max(scores)
        experiment.register_result("fold{}.metric".format(fold),
 best_metric)

        torch.save(
            model.state_dict(),
            os.path.join(
                experiment.checkpoints,
                "fold_{}".format(fold),
                "final_model.pth")
        )

        # predictions
        model.load_best_model(fold)

        # validation

        val_preds = model.predict(valid_loader)
        val_predictions_df = pd.DataFrame(
            val_preds, columns=get_class_names_from_classmap(cla
ss_map))
        val_predictions_df["fname"] = train_df.fname[valid].valu
es
        val_predictions_df.to_csv(
            os.path.join(
                experiment.predictions,
                "val_preds_fold_{}.csv".format(fold)
            ),
```

```
            index=False
        )
        del val_predictions_df

        # test
        test_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.test_data_dir, fname)
                    for fname in test_df.fname.values],
                transform=Compose([
                    LoadAudio(),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        test_preds = model.predict(test_loader)
        test_predictions_df = pd.DataFrame(
            test_preds, columns=get_class_names_from_classmap(cl
ass_map))
        test_predictions_df["fname"] = test_df.fname
        test_predictions_df.to_csv(
            os.path.join(
                experiment.predictions,
                "test_preds_fold_{}.csv".format(fold)
            ),
            index=False
        )
        del test_predictions_df

        # holdout
        if args.holdout_size:
            holdout_loader = torch.utils.data.DataLoader(
                SoundDataset(
                    audio_files=[
                        os.path.join(args.train_data_dir, fname)
                        for fname in holdout_df.fname.values],
                    labels=[item.split(",") for item in holdout_
df.labels.values],
                    transform=Compose([
                        LoadAudio(),
                        MapLabels(class_map),
                        audio_transform,
                        DropFields(("audio", "filename", "sr")),
                    ])
                ),
                shuffle=False,
                batch_size=config.train.batch_size,
                collate_fn=make_collate_fn({"signal": audio_tran
```

```
sform.padding_value}),
                **loader_kwargs
            )

            holdout_metric = model.evaluate(holdout_loader)
            experiment.register_result(
                "fold{}.holdout_metric".format(fold), holdout_me
tric)

            print("\nHoldout metric: {:.4f}".format(holdout_metr
ic))

        if args.device == "cuda":
            torch.cuda.empty_cache()

    # global metric

    if all(
        "fold{}".format(k) in experiment.results.to_dict()
        for k in range(config.data._n_folds)):

        val_df_files = [
            os.path.join(
                experiment.predictions,
                "val_preds_fold_{}.csv".format(fold)
            )
            for fold in range(config.data._n_folds)
        ]

        val_predictions_df = pd.concat([
            pd.read_csv(file) for file in val_df_files]).reset_i
ndex(drop=True)

        labels = np.asarray([
            item["labels"] for item in SoundDataset(
                audio_files=train_df.fname.tolist(),
                labels=[item.split(",") for item in train_df.lab
els.values],
                transform=MapLabels(class_map)
            )
        ])

        val_labels_df = pd.DataFrame(
            labels, columns=get_class_names_from_classmap(class_
map))
        val_labels_df["fname"] = train_df.fname

        assert set(val_predictions_df.fname) == set(val_labels_d
f.fname)

        val_predictions_df.sort_values(by="fname", inplace=True)
        val_labels_df.sort_values(by="fname", inplace=True)

        metric = lwlrap(
            val_labels_df.drop("fname", axis=1).values,
            val_predictions_df.drop("fname", axis=1).values
```

```
        )

        experiment.register_result("metric", metric)

    # submission

    test_df_files = [
        os.path.join(
            experiment.predictions,
            "test_preds_fold_{}.csv".format(fold)
        )
        for fold in range(config.data._n_folds)
    ]

    if all(os.path.isfile for file in test_df_files):
        test_dfs = [pd.read_csv(file) for file in test_df_files]
        submission_df = pd.DataFrame({"fname": test_dfs[0].fname
.values})
        for c in get_class_names_from_classmap(class_map):
            submission_df[c] = np.mean([d[c].values for d in tes
t_dfs], axis=0)
        submission_df.to_csv(
            os.path.join(experiment.predictions, "submission.csv
"), index=False)======
import os
import gc
import argparse
import json
import math
from functools import partial

import pandas as pd
import numpy as np
import torch
from mag.experiment import Experiment
import mag
from sklearn.model_selection import train_test_split

from ops.utils import load_json, get_class_names_from_classmap
from datasets.sound_dataset import SoundDataset
from networks.cpc import CPCModel
from ops.folds import train_validation_data
from ops.transforms import (
    Compose, DropFields, LoadAudio,
    AudioFeatures, MapLabels, RenameFields,
    MixUp, SampleSegment, SampleLongAudio,
    AudioAugmentation)
from ops.padding import make_collate_fn

torch.manual_seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(42)

mag.use_custom_separator("-")

parser = argparse.ArgumentParser(
```

```
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--train_df", required=True, type=str,
    help="path to train dataframe"
)
parser.add_argument(
    "--train_data_dir", required=True, type=str,
    help="path to train data"
)
parser.add_argument(
    "--classmap", required=True, type=str,
    help="path to class map json"
)
parser.add_argument(
    "--log_interval", default=10, type=int,
    help="how frequently to log batch metrics"
    "in terms of processed batches"
)
parser.add_argument(
    "--proj_interval", default=10, type=int,
    help="how frequently to make projection in terms of epochs"
)
parser.add_argument(
    "--batch_size", type=int, default=64,
    help="minibatch size"
)
parser.add_argument(
    "--max_audio_length", type=int, default=10,
    help="max audio length in seconds. For longer clips are samp
led"
)
parser.add_argument(
    "--lr", default=0.01, type=float,
    help="starting learning rate"
)
parser.add_argument(
    "--max_samples", type=int,
    help="maximum number of samples to use"
)
parser.add_argument(
    "--epochs", default=100, type=int,
    help="number of epochs to train"
)
parser.add_argument(
    "--scheduler", type=str, default="steplr_1_0.5",
    help="scheduler type",
)
parser.add_argument(
    "--accumulation_steps", type=int, default=1,
    help="number of gradient accumulation steps",
)
parser.add_argument(
    "--save_every", type=int, default=1,
    help="how frequently to save a model",
```

```
)
parser.add_argument(
    "--device", type=str, required=True,
    help="whether to train on cuda or cpu",
    choices=("cuda", "cpu")
)
parser.add_argument(
    "--n_encoder_layers", type=int, default=5,
    help="number of encoder layers"
)
parser.add_argument(
    "--conv_base_depth", type=int, default=64,
    help="base depth for conv layers"
)
parser.add_argument(
    "--context_size", type=int, default=64,
    help="context size for c network"
)
parser.add_argument(
    "--growth_rate", type=float, default=2,
    help="how quickly to increase the number of units as a funct
ion of layer"
)
parser.add_argument(
    "--prediction_steps", type=int, default=10,
    help="how many steps to predict in the future"
)
parser.add_argument(
    "--weight_decay", type=float, default=1e-5,
    help="weight decay"
)
parser.add_argument(
    "--p_aug", type=float, default=0.0,
    help="probability of audio augmentation"
)
parser.add_argument(
    "--switch_off_augmentations_on", type=int, default=20,
    help="on which epoch to remove augmentations"
)
parser.add_argument(
    "--features", type=str, required=True,
    help="feature descriptor"
)
parser.add_argument(
    "--optimizer", type=str, required=True,
    help="which optimizer to use",
    choices=("adam", "momentum")
)
parser.add_argument(
    "--folds", type=int, required=True, nargs="+",
    help="which folds to use"
)
parser.add_argument(
    "--n_folds", type=int, default=4,
    help="number of folds"
)
```

```
parser.add_argument(
    "--kfold_seed", type=int, default=42,
    help="kfold seed"
)
parser.add_argument(
    "--num_workers", type=int, default=4,
    help="number of workers for data loader",
)
parser.add_argument(
    "--label", type=str, default="cpc",
    help="optional label",
)
args = parser.parse_args()

class_map = load_json(args.classmap)

audio_transform = AudioFeatures(args.features)

with Experiment({
    "network": {
        "n_encoder_layers": args.n_encoder_layers,
        "conv_base_depth": args.conv_base_depth,
        "growth_rate": args.growth_rate,
        "prediction_steps": args.prediction_steps,
        "context_size": args.context_size
    },
    "data": {
        "features": args.features,
        "_n_folds": args.n_folds,
        "_kfold_seed": args.kfold_seed,
        "_input_dim": audio_transform.n_features,
        "p_aug": args.p_aug,
        "max_audio_length": args.max_audio_length
    },
    "train": {
        "_proj_interval": args.proj_interval,
        "accumulation_steps": args.accumulation_steps,
        "batch_size": args.batch_size,
        "learning_rate": args.lr,
        "scheduler": args.scheduler,
        "optimizer": args.optimizer,
        "epochs": args.epochs,
        "_save_every": args.save_every,
        "weight_decay": args.weight_decay,
        "switch_off_augmentations_on": args.switch_off_augmentat
ions_on
    },
    "label": args.label
}) as experiment:

    config = experiment.config
    print()
    print("     ////// CONFIG //////")
    print(experiment.config)

    train_df = pd.read_csv(args.train_df)
```

```
    if args.max_samples:
        train_df = train_df.sample(args.max_samples).reset_index
(drop=True)

    splits = list(train_validation_data(
        train_df.fname, train_df.labels,
        config.data._n_folds, config.data._kfold_seed))

    for fold in args.folds:

        print("\n\n   -----  Fold {}\n".format(fold))

        train, valid = splits[fold]

        loader_kwargs = (
            {"num_workers": args.num_workers, "pin_memory": True
}
            if torch.cuda.is_available() else {})

        experiment.register_directory("checkpoints")
        experiment.register_directory("predictions")

        train_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[train]],
                labels=[
                    item.split(",") for item in
                    train_df.labels.values[train]],
                transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                    SampleLongAudio(max_length=args.max_audio_le
ngth),
                    AudioAugmentation(p=args.p_aug),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ])
            ),
            shuffle=True,
            drop_last=True,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        valid_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[valid]],
                labels=[
                    item.split(",") for item in
```

```
                    train_df.labels.values[valid]],
                transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                    SampleLongAudio(max_length=args.max_audio_le
ngth),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        model = CPCModel(experiment, device=args.device)

        scores = model.fit_validate(
            train_loader, valid_loader,
            epochs=experiment.config.train.epochs, fold=fold,
            log_interval=args.log_interval
        )

        best_metric = max(scores)
        experiment.register_result("fold{}.metric".format(fold),
 best_metric)

        torch.save(
            model.state_dict(),
            os.path.join(
                experiment.checkpoints,
                "fold_{}".format(fold),
                "final_model.pth")
        )

        # predictions
        model.load_best_model(fold)

        if args.device == "cuda":
            torch.cuda.empty_cache()
======
import os
import gc
import argparse
import json
import math
from functools import partial

import pandas as pd
import numpy as np
import torch
from mag.experiment import Experiment
import mag
from sklearn.model_selection import train_test_split
```

```
from datasets.sound_dataset import SoundDataset
from networks.classifiers import HierarchicalCNNClassificationMo
del
from ops.folds import train_validation_data, train_validation_da
ta_stratified
from ops.transforms import (
    Compose, DropFields, LoadAudio,
    AudioFeatures, MapLabels, RenameFields,
    MixUp, SampleSegment, SampleLongAudio,
    AudioAugmentation, ShuffleAudio, CutOut, Identity)
from ops.utils import load_json, get_class_names_from_classmap,
lwlrap
from ops.padding import make_collate_fn

torch.manual_seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(42)

mag.use_custom_separator("-")

parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)

parser.add_argument(
    "--train_df", required=True, type=str,
    help="path to train dataframe"
)
parser.add_argument(
    "--train_data_dir", required=True, type=str,
    help="path to train data"
)
parser.add_argument(
    "--noisy_train_df", type=str,
    help="path to noisy train dataframe (optional)"
)
parser.add_argument(
    "--noisy_train_data_dir", type=str,
    help="path to noisy train data (optional)"
)
parser.add_argument(
    "--share_noisy", action="store_true", default=False,
    help="whether to share noisy files across folds"
)
parser.add_argument(
    "--resume", action="store_true", default=False,
    help="allow resuming even if experiment exists"
)
parser.add_argument(
    "--test_data_dir", required=True, type=str,
    help="path to test data"
)
parser.add_argument(
    "--sample_submission", required=True, type=str,
    help="path sample submission"
```

```
)
parser.add_argument(
    "--classmap", required=True, type=str,
    help="path to class map json"
)
parser.add_argument(
    "--log_interval", default=10, type=int,
    help="how frequently to log batch metrics"
    "in terms of processed batches"
)
parser.add_argument(
    "--batch_size", type=int, default=64,
    help="minibatch size"
)
parser.add_argument(
    "--max_audio_length", type=int, default=10,
    help="max audio length in seconds. For longer clips are samp
led"
)
parser.add_argument(
    "--lr", default=0.01, type=float,
    help="starting learning rate"
)
parser.add_argument(
    "--max_samples", type=int,
    help="maximum number of samples to use"
)
parser.add_argument(
    "--holdout_size", type=float, default=0.0,
    help="size of holdout set"
)
parser.add_argument(
    "--epochs", default=100, type=int,
    help="number of epochs to train"
)
parser.add_argument(
    "--scheduler", type=str, default="steplr_1_0.5",
    help="scheduler type",
)
parser.add_argument(
    "--accumulation_steps", type=int, default=1,
    help="number of gradient accumulation steps",
)
parser.add_argument(
    "--save_every", type=int, default=1,
    help="how frequently to save a model",
)
parser.add_argument(
    "--device", type=str, required=True,
    help="whether to train on cuda or cpu",
    choices=("cuda", "cpu")
)
parser.add_argument(
    "--aggregation_type", type=str, required=True,
    help="how to aggregate outputs",
    choices=("max", "rnn")
```

```
)
parser.add_argument(
    "--num_conv_blocks", type=int, default=5,
    help="number of conv blocks"
)
parser.add_argument(
    "--start_deep_supervision_on", type=int, default=2,
    help="from which layer to start aggregating features for cla
ssification"
)
parser.add_argument(
    "--conv_base_depth", type=int, default=64,
    help="base depth for conv layers"
)
parser.add_argument(
    "--growth_rate", type=float, default=2,
    help="how quickly to increase the number of units as a funct
ion of layer"
)
parser.add_argument(
    "--weight_decay", type=float, default=1e-5,
    help="weight decay"
)
parser.add_argument(
    "--output_dropout", type=float, default=0.0,
    help="output dropout"
)
parser.add_argument(
    "--p_mixup", type=float, default=0.0,
    help="probability of the mixup augmentation"
)
parser.add_argument(
    "--p_aug", type=float, default=0.0,
    help="probability of audio augmentation"
)
parser.add_argument(
    "--switch_off_augmentations_on", type=int, default=20,
    help="on which epoch to remove augmentations"
)
parser.add_argument(
    "--features", type=str, required=True,
    help="feature descriptor"
)
parser.add_argument(
    "--optimizer", type=str, required=True,
    help="which optimizer to use",
    choices=("adam", "momentum")
)
parser.add_argument(
    "--folds", type=int, required=True, nargs="+",
    help="which folds to use"
)
parser.add_argument(
    "--n_folds", type=int, default=4,
    help="number of folds"
)
```

```
parser.add_argument(
    "--kfold_seed", type=int, default=42,
    help="kfold seed"
)
parser.add_argument(
    "--num_workers", type=int, default=4,
    help="number of workers for data loader",
)
parser.add_argument(
    "--label", type=str, default="1d_cnn",
    help="optional label",
)
args = parser.parse_args()

class_map = load_json(args.classmap)

audio_transform = AudioFeatures(args.features)

with Experiment({
    "network": {
        "num_conv_blocks": args.num_conv_blocks,
        "start_deep_supervision_on": args.start_deep_supervision
_on,
        "conv_base_depth": args.conv_base_depth,
        "growth_rate": args.growth_rate,
        "output_dropout": args.output_dropout,
        "aggregation_type": args.aggregation_type
    },
    "data": {
        "features": args.features,
        "_n_folds": args.n_folds,
        "_kfold_seed": args.kfold_seed,
        "_input_dim": audio_transform.n_features,
        "_n_classes": len(class_map),
        "_holdout_size": args.holdout_size,
        "p_mixup": args.p_mixup,
        "p_aug": args.p_aug,
        "max_audio_length": args.max_audio_length,
        "noisy": args.noisy_train_df is not None,
        "_train_df": args.train_df,
        "_train_data_dir": args.train_data_dir,
        "_noisy_train_df": args.noisy_train_df,
        "_noisy_train_data_dir": args.noisy_train_data_dir,
        "_share_noisy": args.share_noisy
    },
    "train": {
        "accumulation_steps": args.accumulation_steps,
        "batch_size": args.batch_size,
        "learning_rate": args.lr,
        "scheduler": args.scheduler,
        "optimizer": args.optimizer,
        "epochs": args.epochs,
        "_save_every": args.save_every,
        "weight_decay": args.weight_decay,
        "switch_off_augmentations_on": args.switch_off_augmentat
ions_on
```

```
    },
    "label": args.label
}, implicit_resuming=args.resume) as experiment:

    config = experiment.config
    print()
    print("     ////// CONFIG //////")
    print(experiment.config)

    train_df = pd.read_csv(args.train_df)
    test_df = pd.read_csv(args.sample_submission)

    if args.noisy_train_df:
        noisy_train_df = pd.read_csv(args.noisy_train_df)

    if args.max_samples:
        train_df = train_df.sample(args.max_samples).reset_index
(drop=True)
        test_df = test_df.sample(
            min(args.max_samples, len(test_df))).reset_index(dro
p=True)

    if args.holdout_size:
        keep, holdout = train_test_split(
            np.arange(len(train_df)), test_size=args.holdout_siz
e,
            random_state=args.kfold_seed)
        holdout_df = train_df.iloc[holdout].reset_index(drop=Tru
e)
        train_df = train_df.iloc[keep].reset_index(drop=True)

    splits = list(train_validation_data_stratified(
        train_df.fname, train_df.labels, class_map,
        config.data._n_folds, config.data._kfold_seed))

    if args.noisy_train_df:
        noisy_splits = list(train_validation_data(
            noisy_train_df.fname, noisy_train_df.labels,
            config.data._n_folds, config.data._kfold_seed))

    for fold in args.folds:

        print("\n\n   -----  Fold {}\n".format(fold))

        train, valid = splits[fold]

        loader_kwargs = (
            {"num_workers": args.num_workers, "pin_memory": True
}
            if torch.cuda.is_available() else {})

        experiment.register_directory("checkpoints")
        experiment.register_directory("predictions")

        if args.noisy_train_df:
```

```
            noisy_train, noisy_valid = noisy_splits[fold]

            if config.data._share_noisy:
                noisy_audio_files = [
                    os.path.join(args.noisy_train_data_dir, fnam
e)
                    for fname in noisy_train_df.fname.values]
                noisy_labels = [
                    item.split(",") for item in
                    noisy_train_df.labels.values]
            else:
                noisy_audio_files = [
                    os.path.join(args.noisy_train_data_dir, fnam
e)
                    for fname in noisy_train_df.fname.values[noi
sy_valid]]
                noisy_labels = [
                    item.split(",") for item in
                    noisy_train_df.labels.values[noisy_valid]]
        else:
            noisy_audio_files = []
            noisy_labels = []

        train_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[train]] +
 noisy_audio_files,
                labels=[
                    item.split(",") for item in
                    train_df.labels.values[train]] + noisy_label
s,
                is_noisy=[0] * len(train) + [1] * len(noisy_labe
ls),
                transform=Compose([
                    LoadAudio(),
                    SampleLongAudio(max_length=args.max_audio_le
ngth),
                    MapLabels(class_map=class_map),
                    (
                        ShuffleAudio(chunk_length=0.5, p=0.5)
                        if config.network.aggregation_type != "r
nn" else Identity()
                    ),
                    MixUp(p=args.p_mixup),
                    AudioAugmentation(p=args.p_aug),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ]),
                clean_transform=Compose([
                    LoadAudio(),
                    SampleLongAudio(max_length=args.max_audio_le
ngth),
                    MapLabels(class_map=class_map),
                ])
```

```
            ),
            shuffle=True,
            drop_last=True,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        valid_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.train_data_dir, fname)
                    for fname in train_df.fname.values[valid]],
                labels=[item.split(",") for item in train_df.lab
els.values[valid]],
                transform=Compose([
                    LoadAudio(),
                    MapLabels(class_map=class_map),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        model = HierarchicalCNNClassificationModel(experiment, d
evice=args.device)

        scores = model.fit_validate(
            train_loader, valid_loader,
            epochs=experiment.config.train.epochs, fold=fold,
            log_interval=args.log_interval
        )

        best_metric = max(scores)
        experiment.register_result("fold{}.metric".format(fold),
 best_metric)

        torch.save(
            model.state_dict(),
            os.path.join(
                experiment.checkpoints,
                "fold_{}".format(fold),
                "final_model.pth")
        )

        # predictions
        model.load_best_model(fold)

        # validation
```

```
        val_preds = model.predict(valid_loader)
        val_predictions_df = pd.DataFrame(
            val_preds, columns=get_class_names_from_classmap(cla
ss_map))
        val_predictions_df["fname"] = train_df.fname[valid].valu
es
        val_predictions_df.to_csv(
            os.path.join(
                experiment.predictions,
                "val_preds_fold_{}.csv".format(fold)
            ),
            index=False
        )
        del val_predictions_df

        # test
        test_loader = torch.utils.data.DataLoader(
            SoundDataset(
                audio_files=[
                    os.path.join(args.test_data_dir, fname)
                    for fname in test_df.fname.values],
                transform=Compose([
                    LoadAudio(),
                    audio_transform,
                    DropFields(("audio", "filename", "sr")),
                ])
            ),
            shuffle=False,
            batch_size=config.train.batch_size,
            collate_fn=make_collate_fn({"signal": audio_transfor
m.padding_value}),
            **loader_kwargs
        )

        test_preds = model.predict(test_loader)
        test_predictions_df = pd.DataFrame(
            test_preds, columns=get_class_names_from_classmap(cl
ass_map))
        test_predictions_df["fname"] = test_df.fname
        test_predictions_df.to_csv(
            os.path.join(
                experiment.predictions,
                "test_preds_fold_{}.csv".format(fold)
            ),
            index=False
        )
        del test_predictions_df

        # holdout
        if args.holdout_size:
            holdout_loader = torch.utils.data.DataLoader(
                SoundDataset(
                    audio_files=[
                        os.path.join(args.train_data_dir, fname)
                        for fname in holdout_df.fname.values],
                    labels=[item.split(",") for item in holdout_
```

```
df.labels.values],
                    transform=Compose([
                        LoadAudio(),
                        MapLabels(class_map),
                        audio_transform,
                        DropFields(("audio", "filename", "sr")),
                    ])
                ),
                shuffle=False,
                batch_size=config.train.batch_size,
                collate_fn=make_collate_fn({"signal": audio_tran
sform.padding_value}),
                **loader_kwargs
            )

            holdout_metric = model.evaluate(holdout_loader)
            experiment.register_result(
                "fold{}.holdout_metric".format(fold), holdout_me
tric)

            print("\nHoldout metric: {:.4f}".format(holdout_metr
ic))

        if args.device == "cuda":
            torch.cuda.empty_cache()

    # global metric

    if all(
        "fold{}".format(k) in experiment.results.to_dict()
        for k in range(config.data._n_folds)):

        val_df_files = [
            os.path.join(
                experiment.predictions,
                "val_preds_fold_{}.csv".format(fold)
            )
            for fold in range(config.data._n_folds)
        ]

        val_predictions_df = pd.concat([
            pd.read_csv(file) for file in val_df_files]).reset_i
ndex(drop=True)

        labels = np.asarray([
            item["labels"] for item in SoundDataset(
                audio_files=train_df.fname.tolist(),
                labels=[item.split(",") for item in train_df.lab
els.values],
                transform=MapLabels(class_map)
            )
        ])

        val_labels_df = pd.DataFrame(
            labels, columns=get_class_names_from_classmap(class_
map))
```

```
        val_labels_df["fname"] = train_df.fname

        assert set(val_predictions_df.fname) == set(val_labels_d
f.fname)

        val_predictions_df.sort_values(by="fname", inplace=True)
        val_labels_df.sort_values(by="fname", inplace=True)

        metric = lwlrap(
            val_labels_df.drop("fname", axis=1).values,
            val_predictions_df.drop("fname", axis=1).values
        )

        experiment.register_result("metric", metric)

    # submission

    test_df_files = [
        os.path.join(
            experiment.predictions,
            "test_preds_fold_{}.csv".format(fold)
        )
        for fold in range(config.data._n_folds)
    ]

    if all(os.path.isfile for file in test_df_files):
        test_dfs = [pd.read_csv(file) for file in test_df_files]
        submission_df = pd.DataFrame({"fname": test_dfs[0].fname
.values})
        for c in get_class_names_from_classmap(class_map):
            submission_df[c] = np.mean([d[c].values for d in tes
t_dfs], axis=0)
        submission_df.to_csv(
            os.path.join(experiment.predictions, "submission.csv
"), index=False)======
```

```
import os
import math
import itertools
from collections import defaultdict, OrderedDict, deque

from tqdm import tqdm
import numpy as np
import torch
import torch.nn as nn
import torchvision.utils
from tensorboardX import SummaryWriter
from torch.nn.functional import binary_cross_entropy_with_lo
gits

from ops.training import OPTIMIZERS, make_scheduler, make_st
ep
from networks.losses import binary_cross_entropy, focal_loss
, lsep_loss
from ops.utils import plot_projection


class APCModel(nn.Module):

    def __init__(self, experiment, device="cuda"):
        super().__init__()

        self.device = device

        self.experiment = experiment
        self.config = experiment.config

        self.input_norm = nn.LayerNorm(
            (self.config.data._input_dim,), elementwise_affi
ne=False)

        self.rnn = nn.LSTM(
            self.config.data._input_dim, self.config.network
.rnn_size,
            num_layers=self.config.network.rnn_layers,
            batch_first=True
        )

        self.output_norm = nn.LayerNorm((self.config.network
.rnn_size,))

        self.prediction_transforms = torch.nn.ModuleList([
            torch.nn.Sequential(
                torch.nn.Linear(
                    self.config.network.rnn_size,
                    self.config.data._input_dim)
            )
            for steps in range(self.config.network.predictio
```

```
n_steps)
        ])

        self.to(self.device)

    def forward(self, signal):

        # signal = signal.permute(0, 2, 1)
        signal = self.input_norm(signal)
        # signal = signal.permute(0, 2, 1)

        output, state = self.rnn(signal)
        output = self.output_norm(output)

        losses = []
        predictions = []

        for step, affine in enumerate(self.prediction_transf
orms, start=1):

            shifted_output = output[:, :-step, :]
            shifted_signal = signal.detach()[:, step:, :]

            prediction = affine(shifted_output)
            predictions.append(prediction)

            loss = torch.abs(shifted_signal - prediction)
            loss = loss.sum(-1)
            loss = loss.mean()

            losses.append(loss)

        r = dict(
            losses=losses,
            output=output,
            predictions=predictions
        )

        return r

    def add_scalar_summaries(
        self, losses, writer, global_step):

        # scalars
        for k, loss in enumerate(losses, start=1):
            writer.add_scalar("loss_{k}".format(k=k), loss,
global_step)

    def add_image_summaries(
        self, signal, output, predictions, global_step, writ
er, to_plot=8):
```

```
        if len(signal) > to_plot:
            signal = signal[:to_plot]
            output = output[:to_plot]
            predictions = [p[:to_plot] for p in predictions]

        # signal
        image_grid = torchvision.utils.make_grid(
            signal.data.cpu().unsqueeze(1),
            normalize=True, scale_each=True
        )
        writer.add_image("signal", image_grid, global_step)
        # output
        image_grid = torchvision.utils.make_grid(
            output.data.cpu().unsqueeze(1),
            normalize=True, scale_each=True
        )
        writer.add_image("output", image_grid, global_step)

        for k, p in enumerate(predictions, start=1):
            image_grid = torchvision.utils.make_grid(
                p.data.cpu().unsqueeze(1),
                normalize=True, scale_each=True
            )
            writer.add_image(
                "prediction_{k}".format(k=k), image_grid, gl
obal_step)

    def add_projection_summary(self, image, global_step, wri
ter, name="projection"):
        writer.add_image(name, image.transpose(2, 0, 1), glo
bal_step)

    def train_epoch(self, train_loader,
                    epoch, log_interval, write_summary=True)
:

        self.train()

        print(
            "\n" + " " * 10 + "****** Epoch {epoch} ******\n
"
            .format(epoch=epoch)
        )

        history = deque(maxlen=30)

        self.optimizer.zero_grad()
        accumulated_loss = 0

        with tqdm(total=len(train_loader), ncols=80) as pb:

            for batch_idx, sample in enumerate(train_loader)
```

```
:
                self.global_step += 1

                make_step(self.scheduler, step=self.global_s
tep)

                signal, labels = (
                    sample["signal"].to(self.device),
                    sample["labels"].to(self.device)
                )

                outputs = self(signal)

                losses = outputs["losses"]

                loss = (
                    sum(losses)
                ) / self.config.train.accumulation_steps

                loss.backward()
                accumulated_loss += loss

                if batch_idx % self.config.train.accumulatio
n_steps == 0:
                    self.optimizer.step()
                    accumulated_loss = 0
                    self.optimizer.zero_grad()

                history.append(loss.item())

                pb.update()
                pb.set_description(
                    "Loss: {:.4f}".format(
                        np.mean(history)))

                if batch_idx % log_interval == 0:
                    self.add_scalar_summaries(
                        [loss.item() for loss in losses],
                        self.train_writer, self.global_step)

                if batch_idx == 0:
                    self.add_image_summaries(
                        signal,
                        outputs["output"],
                        outputs["predictions"],
                        self.global_step, self.train_writer)

    def evaluate(self, loader, verbose=False, write_summary=
False, epoch=None):

        self.eval()
```

```
        valid_losses = [0 for _ in range(self.config.network
.prediction_steps)]

        all_outputs = []
        all_labels = []

        with torch.no_grad():
            for batch_idx, sample in enumerate(loader):

                signal, labels = (
                    sample["signal"].to(self.device),
                    sample["labels"].to(self.device)
                )

                outputs = self(signal)

                losses = outputs["losses"]

                multiplier = len(signal) / len(loader.datase
t)

                for k, loss in enumerate(losses):
                    valid_losses[k] += loss.item() * multipl
ier

                all_outputs.extend(
                    outputs["output"].data.cpu().numpy())
                all_labels.extend(labels.data.cpu().numpy())

        valid_loss = sum(valid_losses)

        all_labels = np.array(all_labels)

        if write_summary:
            self.add_scalar_summaries(
                valid_losses,
                writer=self.valid_writer, global_step=self.g
lobal_step
            )
            if epoch % self.config.train._proj_interval == 0
:
                self.add_projection_summary(
                    plot_projection(
                        all_outputs, all_labels, frames_per_
example=5, newline=True),
                    writer=self.valid_writer, global_step=se
lf.global_step,
                    name="projection_output")

        if verbose:
            print("\nValidation loss: {:.4f}".format(valid_l
```

```
oss))

        return -valid_loss

    def validation(self, valid_loader, epoch):
        return self.evaluate(
            valid_loader,
            verbose=True, write_summary=True, epoch=epoch)

    def predict(self, loader):

        self.eval()

        all_class_probs = []

        with torch.no_grad():
            for sample in loader:

                signal = sample["signal"].to(self.device)

                outputs = self(signal)

                class_logits = outputs["class_logits"].squee
ze()

                class_probs = torch.sigmoid(class_logits).da
ta.cpu().numpy()
                all_class_probs.extend(class_probs)

        all_class_probs = np.asarray(all_class_probs)

        return all_class_probs

    def fit_validate(self, train_loader, valid_loader, epoch
s, fold,
                     log_interval=25):


        self.experiment.register_directory("summaries")
        self.train_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "train"
            )
        )
        self.valid_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "valid"
            )
```

```
        )

        os.makedirs(
            os.path.join(
                self.experiment.checkpoints,
                "fold_{}".format(fold)),
            exist_ok=True
        )

        self.global_step = 0
        self.make_optimizer(max_steps=len(train_loader) * ep
ochs)

        scores = []
        best_score = 0

        for epoch in range(epochs):

            make_step(self.scheduler, epoch=epoch)

            if epoch == self.config.train.switch_off_augment
ations_on:
                train_loader.dataset.transform.switch_off_au
gmentations()

            self.train_epoch(
                train_loader, epoch,
                log_interval, write_summary=True
            )
            validation_score = self.validation(valid_loader,
 epoch)
            scores.append(validation_score)

            if epoch % self.config.train._save_every == 0:
                print("\nSaving model on epoch", epoch)
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "model_on_epoch_{}.pth".format(epoch
)
                    )
                )

            if validation_score > best_score:
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "best_model.pth"
```

```
                    )
                )
                best_score = validation_score

        return scores

    def make_optimizer(self, max_steps):

        optimizer = OPTIMIZERS[self.config.train.optimizer]
        optimizer = optimizer(
            self.parameters(),
            self.config.train.learning_rate,
            weight_decay=self.config.train.weight_decay
        )
        self.optimizer = optimizer
        self.scheduler = make_scheduler(
            self.config.train.scheduler, max_steps=max_steps
)(optimizer)

    def load_best_model(self, fold):

        self.load_state_dict(
            torch.load(
                os.path.join(
                    self.experiment.checkpoints,
                    "fold_{}".format(fold),
                    "best_model.pth"
                )
            )
        )

======
import os
import math
import itertools
from collections import defaultdict, OrderedDict, deque

from tqdm import tqdm
import numpy as np
import torch
import torch.nn as nn
import torch.utils.model_zoo as model_zoo
import torchvision.utils
from tensorboardX import SummaryWriter
from pretrainedmodels.models import resnet18, resnet34


from ops.training import OPTIMIZERS, make_scheduler, make_st
ep
from networks.losses import binary_cross_entropy, focal_loss
, lsep_loss
from ops.utils import lwlrap, make_mel_filterbanks, is_mel,
```

```
is_stft, compute_torch_stft


class ConvLockedDropout(nn.Module):
    def __init__(self, dropout_rate=0.0):
        super().__init__()
        self.dropout_rate = dropout_rate

    def forward(self, x):
        if not self.training or not self.dropout_rate:
            return x

        n, s, t = x.size()

        m = torch.zeros(n, s, 1, device=x.device).bernoulli_
(1 - self.dropout_rate)
        m = m.expand_as(x)
        return m * x


class ResnetBlock(nn.Module):

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

        self.conv1 = nn.Conv1d(depth, depth, kernel_size=1)
        self.bn1 = nn.BatchNorm1d(depth)
        self.conv2 = nn.Conv1d(depth, depth, kernel_size=3,
padding=1)
        self.bn2 = nn.BatchNorm1d(depth)
        self.conv3 = nn.Conv1d(depth, depth, kernel_size=1)
        self.bn3 = nn.BatchNorm1d(depth)
        self.prelu1 = nn.PReLU(depth)
        self.prelu2 = nn.PReLU(depth)
        self.prelu3 = nn.PReLU(depth)

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.prelu1(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.prelu2(out)

        out = self.conv3(out)
        out = self.bn3(out)

        out += identity
        out = self.prelu3(out)
```

```
        return out


class ResnetBlock2d(nn.Module):

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

        self.conv1 = nn.Conv2d(depth, depth, kernel_size=1)
        self.bn1 = nn.BatchNorm2d(depth)
        self.conv2 = nn.Conv2d(depth, depth, kernel_size=3,
padding=1)
        self.bn2 = nn.BatchNorm2d(depth)
        self.conv3 = nn.Conv2d(depth, depth, kernel_size=1)
        self.bn3 = nn.BatchNorm2d(depth)
        self.prelu1 = nn.PReLU(depth)
        self.prelu2 = nn.PReLU(depth)
        self.prelu3 = nn.PReLU(depth)

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.prelu1(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.prelu2(out)

        out = self.conv3(out)
        out = self.bn3(out)

        out += identity
        out = self.prelu3(out)

        return out


class HierarchicalCNNClassificationModel(nn.Module):

    def __init__(self, experiment, device="cuda"):
        super().__init__()

        self.device = device

        self.experiment = experiment
        self.config = experiment.config

        if is_mel(self.config.data.features):
            self.filterbanks = torch.from_numpy(
```

```
                make_mel_filterbanks(self.config.data.featur
es)).to(self.device)

        self.conv_modules = torch.nn.ModuleList()
        self.rnns = torch.nn.ModuleList()

        total_depth = 0
        rnn_size = 128

        for k in range(self.config.network.num_conv_blocks):

            input_size = self.config.data._input_dim if not
k else depth
            depth = int(
                self.config.network.growth_rate ** k
                * self.config.network.conv_base_depth)

            if k >= self.config.network.start_deep_supervisi
on_on:
                if self.config.network.aggregation_type == "
max":
                    total_depth += depth
                elif self.config.network.aggregation_type ==
 "rnn":
                    total_depth += rnn_size * 2
                    self.rnns.append(
                        nn.Sequential(
                            nn.LayerNorm((depth,)),
                            nn.GRU(
                                depth, rnn_size, batch_first
=True, bidirectional=True)
                        )
                    )

            modules = [nn.BatchNorm1d(input_size)]
            modules.extend([
                nn.Conv1d(
                    input_size,
                    depth,
                    kernel_size=3,
                    padding=1
                ),
                nn.MaxPool1d(kernel_size=2, stride=2),
                nn.BatchNorm1d(depth),
                nn.PReLU(depth),
                ResnetBlock(depth)
            ])

            self.conv_modules.append(nn.Sequential(*modules)
)

        self.global_maxpool = nn.AdaptiveMaxPool1d(1)
```

```
        self.output_transform = nn.Sequential(
            nn.BatchNorm1d(total_depth),
            nn.Linear(total_depth, total_depth),
            nn.BatchNorm1d(total_depth),
            nn.PReLU(total_depth),
            nn.Dropout(p=self.config.network.output_dropout)
,
            nn.Linear(total_depth, self.config.data._n_class
es)
        )

        self.to(self.device)

    def forward(self, signal):

        if is_stft(self.config.data.features) or is_mel(self
.config.data.features):
            signal = compute_torch_stft(
                signal.squeeze(-1),
                self.config.data.features
            )

            if is_stft(self.config.data.features):
                signal = torch.log(signal + 1e-4)

        if is_mel(self.config.data.features):
            signal = nn.functional.conv1d(
                signal,
                self.filterbanks.unsqueeze(-1)
            )
            signal = torch.log(signal + 1e-4)

        features = []

        h = signal
        for k, module in enumerate(self.conv_modules):
            h = module(h)
            if k >= self.config.network.start_deep_supervisi
on_on:
                if self.config.network.aggregation_type == "
max":
                    features.append(self.global_maxpool(h).s
queeze(-1))
                elif self.config.network.aggregation_type ==
 "rnn":
                    rnn_input = h.permute(0, 2, 1)
                    outputs, state = self.rnns[
                        k - self.config.network.start_deep_s
upervision_on](rnn_input)
                    features.append(
                        state.permute(1, 0, 2).contiguous().
```

```
view(rnn_input.size(0), -1))

        features = torch.cat(features, -1)

        class_logits = self.output_transform(features)

        r = dict(
            class_logits=class_logits
        )

        return r

    def add_scalar_summaries(
        self, loss, metric, writer, global_step):

        # scalars
        writer.add_scalar("loss", loss, global_step)
        writer.add_scalar("metric", metric, global_step)

    def add_image_summaries(self, signal, global_step, write
r, to_plot=8):

        if len(signal) > to_plot:
            signal = signal[:to_plot]

        # image
        image_grid = torchvision.utils.make_grid(
            signal.data.cpu().unsqueeze(1),
            normalize=True, scale_each=True
        )
        writer.add_image("signal", image_grid, global_step)

    def train_epoch(self, train_loader,
                    epoch, log_interval, write_summary=True)
:

        self.train()

        print(
            "\n" + " " * 10 + "****** Epoch {epoch} ******\n
"
            .format(epoch=epoch)
        )

        history = deque(maxlen=30)

        self.optimizer.zero_grad()
        accumulated_loss = 0

        with tqdm(total=len(train_loader), ncols=80) as pb:

            for batch_idx, sample in enumerate(train_loader)
```

```
:
                self.global_step += 1

                make_step(self.scheduler, step=self.global_s
tep)

                signal, labels = (
                    sample["signal"].to(self.device),
                    sample["labels"].to(self.device).float()
                )

                outputs = self(signal)

                class_logits = outputs["class_logits"].squee
ze()

                loss = (
                    lsep_loss(
                        class_logits,
                        labels
                    )
                ) / self.config.train.accumulation_steps

                loss.backward()
                accumulated_loss += loss

                if batch_idx % self.config.train.accumulatio
n_steps == 0:
                    self.optimizer.step()
                    accumulated_loss = 0
                    self.optimizer.zero_grad()

                probs = torch.sigmoid(class_logits).data.cpu
().numpy()
                labels = labels.data.cpu().numpy()

                metric = lwlrap(labels, probs)
                history.append(metric)

                pb.update()
                pb.set_description(
                    "Loss: {:.4f}, Metric: {:.4f}".format(
                        loss.item(), np.mean(history)))

                if batch_idx % log_interval == 0:
                    self.add_scalar_summaries(
                        loss.item(), metric, self.train_writ
er, self.global_step)

                if batch_idx == 0:
                    self.add_image_summaries(
```

```
                        signal, self.global_step, self.train
_writer)

    def evaluate(self, loader, verbose=False, write_summary=
False, epoch=None):

        self.eval()

        valid_loss = 0

        all_class_probs = []
        all_labels = []

        with torch.no_grad():
            for batch_idx, sample in enumerate(loader):

                signal, labels = (
                    sample["signal"].to(self.device),
                    sample["labels"].to(self.device).float()
                )

                outputs = self(signal)

                class_logits = outputs["class_logits"].squee
ze()

                loss = (
                    lsep_loss(
                        class_logits,
                        labels,
                    )
                ).item()

                multiplier = len(labels) / len(loader.datase
t)

                valid_loss += loss * multiplier

                class_probs = torch.sigmoid(class_logits).da
ta.cpu().numpy()
                labels = labels.data.cpu().numpy()

                all_class_probs.extend(class_probs)
                all_labels.extend(labels)

            all_class_probs = np.asarray(all_class_probs)
            all_labels = np.asarray(all_labels)

            metric = lwlrap(all_labels, all_class_probs)

            if write_summary:
                self.add_scalar_summaries(
```

```
                    valid_loss,
                    metric,
                    writer=self.valid_writer, global_step=se
lf.global_step
                )

            if verbose:
                print("\nValidation loss: {:.4f}".format(val
id_loss))
                print("Validation metric: {:.4f}".format(met
ric))

            return metric

    def validation(self, valid_loader, epoch):
        return self.evaluate(
            valid_loader,
            verbose=True, write_summary=True, epoch=epoch)

    def predict(self, loader):

        self.eval()

        all_class_probs = []

        with torch.no_grad():
            for sample in loader:

                signal = sample["signal"].to(self.device)

                outputs = self(signal)

                class_logits = outputs["class_logits"].squee
ze()

                class_probs = torch.sigmoid(class_logits).da
ta.cpu().numpy()
                all_class_probs.extend(class_probs)

        all_class_probs = np.asarray(all_class_probs)

        return all_class_probs

    def fit_validate(self, train_loader, valid_loader, epoch
s, fold,
                     log_interval=25):


        self.experiment.register_directory("summaries")
        self.train_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
```

```
                "fold_{}".format(fold),
                "train"
            )
        )
        self.valid_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "valid"
            )
        )

        os.makedirs(
            os.path.join(
                self.experiment.checkpoints,
                "fold_{}".format(fold)),
            exist_ok=True
        )

        self.global_step = 0
        self.make_optimizer(max_steps=len(train_loader) * ep
ochs)

        scores = []
        best_score = 0

        for epoch in range(epochs):

            make_step(self.scheduler, epoch=epoch)

            if epoch == self.config.train.switch_off_augment
ations_on:
                train_loader.dataset.transform.switch_off_au
gmentations()

            self.train_epoch(
                train_loader, epoch,
                log_interval, write_summary=True
            )
            validation_score = self.validation(valid_loader,
 epoch)
            scores.append(validation_score)

            if epoch % self.config.train._save_every == 0:
                print("\nSaving model on epoch", epoch)
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "model_on_epoch_{}.pth".format(epoch
)
```

```
                    )
                )

            if validation_score > best_score:
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "best_model.pth"
                    )
                )
                best_score = validation_score

        return scores

    def make_optimizer(self, max_steps):

        optimizer = OPTIMIZERS[self.config.train.optimizer]
        optimizer = optimizer(
            self.parameters(),
            self.config.train.learning_rate,
            weight_decay=self.config.train.weight_decay
        )
        self.optimizer = optimizer
        self.scheduler = make_scheduler(
            self.config.train.scheduler, max_steps=max_steps
)(optimizer)

    def load_best_model(self, fold):

        self.load_state_dict(
            torch.load(
                os.path.join(
                    self.experiment.checkpoints,
                    "fold_{}".format(fold),
                    "best_model.pth"
                )
            )
        )


class TwoDimensionalCNNClassificationModel(nn.Module):

    def __init__(self, experiment, device="cuda"):
        super().__init__()

        self.device = device

        self.experiment = experiment
        self.config = experiment.config
```

```
        if is_mel(self.config.data.features):
            self.filterbanks = torch.from_numpy(
                make_mel_filterbanks(self.config.data.featur
es)).to(self.device)

        self.conv_modules = torch.nn.ModuleList()
        self.rnns = torch.nn.ModuleList()

        total_depth = 0

        for k in range(self.config.network.num_conv_blocks):

            input_size = 2 if not k else depth
            depth = int(
                self.config.network.growth_rate ** k
                * self.config.network.conv_base_depth)

            rnn_size = 128

            if k >= self.config.network.start_deep_supervisi
on_on:
                if self.config.network.aggregation_type == "
max":
                    total_depth += depth
                elif self.config.network.aggregation_type ==
 "rnn":
                    total_depth += rnn_size * 2
                    self.rnns.append(
                        nn.Sequential(
                            nn.LayerNorm((depth,)),
                            nn.GRU(
                                depth, rnn_size, batch_first
=True, bidirectional=True)
                        )
                    )

            modules = [nn.BatchNorm2d(input_size)]
            modules.extend([
                nn.Conv2d(
                    input_size,
                    depth,
                    kernel_size=3,
                    padding=1
                ),
                nn.MaxPool2d(kernel_size=2, stride=2),
                nn.BatchNorm2d(depth),
                nn.PReLU(depth),
                ResnetBlock2d(depth)
            ])

            self.conv_modules.append(nn.Sequential(*modules)
)
```

```
        self.global_maxpool = nn.AdaptiveMaxPool2d(1)

        self.output_transform = nn.Sequential(
            nn.BatchNorm1d(total_depth),
            nn.Linear(total_depth, total_depth),
            nn.BatchNorm1d(total_depth),
            nn.PReLU(total_depth),
            nn.Dropout(p=self.config.network.output_dropout)
,
            nn.Linear(total_depth, self.config.data._n_class
es)
        )

        self.to(self.device)

    def _add_frequency_encoding(self, x):
        n, d, h, w = x.size()

        vertical = torch.linspace(-1, 1, h, device=x.device)
.view(1, 1, -1, 1)
        vertical = vertical.repeat(n, 1, 1, w)

        x = torch.cat([x, vertical], dim=1)

        return x

    def forward(self, signal):

        if is_stft(self.config.data.features) or is_mel(self
.config.data.features):
            signal = compute_torch_stft(
                signal.squeeze(-1),
                self.config.data.features
            )

            if is_stft(self.config.data.features):
                signal = torch.log(signal + 1e-4)

        if is_mel(self.config.data.features):
            signal = nn.functional.conv1d(
                signal,
                self.filterbanks.unsqueeze(-1)
            )
            signal = torch.log(signal + 1e-4)

        signal = signal.unsqueeze(1)
        signal = self._add_frequency_encoding(signal)

        features = []

        h = signal
```

```
        for k, module in enumerate(self.conv_modules):
            h = module(h)
            if k >= self.config.network.start_deep_supervisi
on_on:
                if self.config.network.aggregation_type == "
max":
                    features.append(self.global_maxpool(h).s
queeze(-1).squeeze(-1))
                elif self.config.network.aggregation_type ==
 "rnn":
                    rnn_input = torch.mean(h, 2).permute(0,
2, 1)
                    outputs, state = self.rnns[
                        k - self.config.network.start_deep_s
upervision_on](rnn_input)
                    features.append(
                        state.permute(1, 0, 2).contiguous().
view(rnn_input.size(0), -1))

        features = torch.cat(features, -1)

        class_logits = self.output_transform(features)

        r = dict(
            class_logits=class_logits
        )

        return r

    def add_scalar_summaries(
        self, loss, metric, writer, global_step):

        # scalars
        writer.add_scalar("loss", loss, global_step)
        writer.add_scalar("metric", metric, global_step)

    def add_histogram_summaries(
        self, losses, writer, global_step):

        writer.add_histogram("losses", np.array(losses), glo
bal_step=global_step)

    def add_image_summaries(self, signal, global_step, write
r, to_plot=8):

        if len(signal) > to_plot:
            signal = signal[:to_plot]

        # image
        image_grid = torchvision.utils.make_grid(
            signal.data.cpu().unsqueeze(1),
            normalize=True, scale_each=True
```

```
        )
        writer.add_image("signal", image_grid, global_step)

    def train_epoch(self, train_loader,
                    epoch, log_interval, write_summary=True)
:

        self.train()

        print(
            "\n" + " " * 10 + "****** Epoch {epoch} ******\n
"
            .format(epoch=epoch)
        )

        training_losses = []

        history = deque(maxlen=30)

        self.optimizer.zero_grad()
        accumulated_loss = 0

        with tqdm(total=len(train_loader), ncols=80) as pb:

            for batch_idx, sample in enumerate(train_loader)
:

                self.global_step += 1

                make_step(self.scheduler, step=self.global_s
tep)

                signal, labels, is_noisy = (
                    sample["signal"].to(self.device),
                    sample["labels"].to(self.device).float()
,
                    sample["is_noisy"].to(self.device).float
()
                )

                outputs = self(signal)

                class_logits = outputs["class_logits"]

                loss = (
                    lsep_loss(
                        class_logits,
                        labels,
                        average=False
                    )
                ) / self.config.train.accumulation_steps
```

```
                training_losses.extend(loss.data.cpu().numpy
())
                loss = loss.mean()

                loss.backward()
                accumulated_loss += loss

                if batch_idx % self.config.train.accumulatio
n_steps == 0:
                    self.optimizer.step()
                    accumulated_loss = 0
                    self.optimizer.zero_grad()

                probs = torch.sigmoid(class_logits).data.cpu
().numpy()
                labels = labels.data.cpu().numpy()

                metric = lwlrap(labels, probs)
                history.append(metric)

                pb.update()
                pb.set_description(
                    "Loss: {:.4f}, Metric: {:.4f}".format(
                        loss.item(), np.mean(history)))

                if batch_idx % log_interval == 0:
                    self.add_scalar_summaries(
                        loss.item(), metric, self.train_writ
er, self.global_step)

                if batch_idx == 0:
                    self.add_image_summaries(
                        signal, self.global_step, self.train
_writer)

        self.add_histogram_summaries(
            training_losses, self.train_writer, self.global_
step)

    def evaluate(self, loader, verbose=False, write_summary=
False, epoch=None):

        self.eval()

        valid_loss = 0

        all_class_probs = []
        all_labels = []

        with torch.no_grad():
            for batch_idx, sample in enumerate(loader):
```

```
                signal, labels = (
                    sample["signal"].to(self.device),
                    sample["labels"].to(self.device).float()
                )

                outputs = self(signal)

                class_logits = outputs["class_logits"]

                loss = (
                    lsep_loss(
                        class_logits,
                        labels,
                    )
                ).item()

                multiplier = len(labels) / len(loader.datase
t)

                valid_loss += loss * multiplier

                class_probs = torch.sigmoid(class_logits).da
ta.cpu().numpy()
                labels = labels.data.cpu().numpy()

                all_class_probs.extend(class_probs)
                all_labels.extend(labels)

            all_class_probs = np.asarray(all_class_probs)
            all_labels = np.asarray(all_labels)

            metric = lwlrap(all_labels, all_class_probs)

            if write_summary:
                self.add_scalar_summaries(
                    valid_loss,
                    metric,
                    writer=self.valid_writer, global_step=se
lf.global_step
                )

            if verbose:
                print("\nValidation loss: {:.4f}".format(val
id_loss))
                print("Validation metric: {:.4f}".format(met
ric))

            return metric

    def validation(self, valid_loader, epoch):
        return self.evaluate(
            valid_loader,
```

```
            verbose=True, write_summary=True, epoch=epoch)

    def predict(self, loader, n_tta=1):

        self.eval()

        all_class_probs = []

        for k in range(n_tta):

            tta_probs = []

            with torch.no_grad():
                for sample in loader:

                    signal = sample["signal"].to(self.device
)

                    outputs = self(signal)

                    class_logits = outputs["class_logits"]

                    class_probs = torch.sigmoid(class_logits
).data.cpu().numpy()
                    tta_probs.extend(class_probs)

            tta_probs = np.array(tta_probs)
            all_class_probs.append(tta_probs)

        all_class_probs = np.mean(all_class_probs, 0)

        return all_class_probs

    def fit_validate(self, train_loader, valid_loader, epoch
s, fold,
                     log_interval=25):


        self.experiment.register_directory("summaries")
        self.train_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "train"
            )
        )
        self.valid_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "valid"
            )
```

```
        )

        os.makedirs(
            os.path.join(
                self.experiment.checkpoints,
                "fold_{}".format(fold)),
            exist_ok=True
        )

        self.global_step = 0
        self.make_optimizer(max_steps=len(train_loader) * ep
ochs)

        scores = []
        best_score = 0

        for epoch in range(epochs):

            make_step(self.scheduler, epoch=epoch)

            if epoch == self.config.train.switch_off_augment
ations_on:
                train_loader.dataset.transform.switch_off_au
gmentations()

            self.train_epoch(
                train_loader, epoch,
                log_interval, write_summary=True
            )
            validation_score = self.validation(valid_loader,
 epoch)
            scores.append(validation_score)

            if epoch % self.config.train._save_every == 0:
                print("\nSaving model on epoch", epoch)
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "model_on_epoch_{}.pth".format(epoch
)
                    )
                )

            if validation_score > best_score:
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "best_model.pth"
```

```
                    )
                )
                best_score = validation_score

        return scores

    def make_optimizer(self, max_steps):

        optimizer = OPTIMIZERS[self.config.train.optimizer]
        optimizer = optimizer(
            self.parameters(),
            self.config.train.learning_rate,
            weight_decay=self.config.train.weight_decay
        )
        self.optimizer = optimizer
        self.scheduler = make_scheduler(
            self.config.train.scheduler, max_steps=max_steps
)(optimizer)

    def load_best_model(self, fold):

        self.load_state_dict(
            torch.load(
                os.path.join(
                    self.experiment.checkpoints,
                    "fold_{}".format(fold),
                    "best_model.pth"
                )
            )
        )



class CNNBackboneClassificationModel(nn.Module):

    def __init__(self, experiment, device="cuda"):
        super().__init__()

        self.device = device

        self.experiment = experiment
        self.config = experiment.config

        if is_mel(self.config.data.features):
            self.filterbanks = torch.from_numpy(
                make_mel_filterbanks(self.config.data.featur
es)).to(self.device)

        self.input_norm = nn.BatchNorm2d(3)

        if self.config.network.backbone == "resnet18":
            self.backbone = resnet18(pretrained=None)
```

```
        elif self.config.network.backbone == "resnet34":
            self.backbone = resnet34(pretrained=None)

        self.global_maxpool = nn.AdaptiveMaxPool2d(1)

        total_depth = self.backbone.last_linear.in_features

        self.output_transform = nn.Sequential(
            nn.BatchNorm1d(total_depth),
            nn.Linear(total_depth, total_depth),
            nn.BatchNorm1d(total_depth),
            nn.PReLU(total_depth),
            nn.Dropout(p=self.config.network.output_dropout)
,
            nn.Linear(total_depth, self.config.data._n_class
es)
        )

        self.to(self.device)

    def forward(self, signal):

        if is_stft(self.config.data.features) or is_mel(self
.config.data.features):
            signal = compute_torch_stft(
                signal.squeeze(-1),
                self.config.data.features
            )

            if is_stft(self.config.data.features):
                signal = torch.log(signal + 1e-4)

        if is_mel(self.config.data.features):
            signal = nn.functional.conv1d(
                signal,
                self.filterbanks.unsqueeze(-1)
            )
            signal = torch.log(signal + 1e-4)

        signal = signal.unsqueeze(1)
        signal = signal.repeat(1, 3, 1, 1)
        signal = self.input_norm(signal)

        h = self.backbone.features(signal)

        features = self.global_maxpool(h).squeeze(-1).squeez
e(-1)

        class_logits = self.output_transform(features)

        r = dict(
            class_logits=class_logits
```

```
        )

        return r

    def add_scalar_summaries(
        self, loss, metric, writer, global_step):

        # scalars
        writer.add_scalar("loss", loss, global_step)
        writer.add_scalar("metric", metric, global_step)

    def add_histogram_summaries(
        self, losses, writer, global_step):

        writer.add_histogram("losses", np.array(losses), glo
bal_step=global_step)

    def add_image_summaries(self, signal, global_step, write
r, to_plot=8):

        if len(signal) > to_plot:
            signal = signal[:to_plot]

        # image
        image_grid = torchvision.utils.make_grid(
            signal.data.cpu().unsqueeze(1),
            normalize=True, scale_each=True
        )
        writer.add_image("signal", image_grid, global_step)

    def train_epoch(self, train_loader,
                    epoch, log_interval, write_summary=True)
:

        self.train()

        print(
            "\n" + " " * 10 + "****** Epoch {epoch} ******\n
"
            .format(epoch=epoch)
        )

        training_losses = []

        history = deque(maxlen=30)

        self.optimizer.zero_grad()
        accumulated_loss = 0

        with tqdm(total=len(train_loader), ncols=80) as pb:

            for batch_idx, sample in enumerate(train_loader)
```

```
:
                    self.global_step += 1

                    make_step(self.scheduler, step=self.global_s
tep)

                    signal, labels, is_noisy = (
                        sample["signal"].to(self.device),
                        sample["labels"].to(self.device).float()
,
                        sample["is_noisy"].to(self.device).float
()
                    )

                    outputs = self(signal)

                    class_logits = outputs["class_logits"]

                    loss = (
                        lsep_loss(
                            class_logits,
                            labels,
                            average=False
                        )
                    ) / self.config.train.accumulation_steps

                    training_losses.extend(loss.data.cpu().numpy
())
                    loss = loss.mean()

                    loss.backward()
                    accumulated_loss += loss

                    if batch_idx % self.config.train.accumulatio
n_steps == 0:
                        self.optimizer.step()
                        accumulated_loss = 0
                        self.optimizer.zero_grad()

                    probs = torch.sigmoid(class_logits).data.cpu
().numpy()
                    labels = labels.data.cpu().numpy()

                    metric = lwlrap(labels, probs)
                    history.append(metric)

                    pb.update()
                    pb.set_description(
                        "Loss: {:.4f}, Metric: {:.4f}".format(
                            loss.item(), np.mean(history)))
```

```
                if batch_idx % log_interval == 0:
                    self.add_scalar_summaries(
                        loss.item(), metric, self.train_writ
er, self.global_step)

                if batch_idx == 0:
                    self.add_image_summaries(
                        signal, self.global_step, self.train
_writer)

        self.add_histogram_summaries(
            training_losses, self.train_writer, self.global_
step)

    def evaluate(self, loader, verbose=False, write_summary=
False, epoch=None):

        self.eval()

        valid_loss = 0

        all_class_probs = []
        all_labels = []

        with torch.no_grad():
            for batch_idx, sample in enumerate(loader):

                signal, labels = (
                    sample["signal"].to(self.device),
                    sample["labels"].to(self.device).float()
                )

                outputs = self(signal)

                class_logits = outputs["class_logits"]

                loss = (
                    lsep_loss(
                        class_logits,
                        labels,
                    )
                ).item()

                multiplier = len(labels) / len(loader.datase
t)

                valid_loss += loss * multiplier

                class_probs = torch.sigmoid(class_logits).da
ta.cpu().numpy()
                labels = labels.data.cpu().numpy()
```

```
                all_class_probs.extend(class_probs)
                all_labels.extend(labels)

            all_class_probs = np.asarray(all_class_probs)
            all_labels = np.asarray(all_labels)

            metric = lwlrap(all_labels, all_class_probs)

            if write_summary:
                self.add_scalar_summaries(
                    valid_loss,
                    metric,
                    writer=self.valid_writer, global_step=se
lf.global_step
                )

            if verbose:
                print("\nValidation loss: {:.4f}".format(val
id_loss))
                print("Validation metric: {:.4f}".format(met
ric))

            return metric

    def validation(self, valid_loader, epoch):
        return self.evaluate(
            valid_loader,
            verbose=True, write_summary=True, epoch=epoch)

    def predict(self, loader, n_tta=1):

        self.eval()

        all_class_probs = []

        for k in range(n_tta):

            tta_probs = []

            with torch.no_grad():
                for sample in loader:

                    signal = sample["signal"].to(self.device
)

                    outputs = self(signal)

                    class_logits = outputs["class_logits"]

                    class_probs = torch.sigmoid(class_logits
).data.cpu().numpy()
                    tta_probs.extend(class_probs)
```

```
            tta_probs = np.array(tta_probs)
            all_class_probs.append(tta_probs)

        all_class_probs = np.mean(all_class_probs, 0)

        return all_class_probs

    def fit_validate(self, train_loader, valid_loader, epoch
s, fold,
                     log_interval=25):


        self.experiment.register_directory("summaries")
        self.train_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "train"
            )
        )
        self.valid_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "valid"
            )
        )

        os.makedirs(
            os.path.join(
                self.experiment.checkpoints,
                "fold_{}".format(fold)),
            exist_ok=True
        )

        self.global_step = 0
        self.make_optimizer(max_steps=len(train_loader) * ep
ochs)

        scores = []
        best_score = 0

        for epoch in range(epochs):

            make_step(self.scheduler, epoch=epoch)

            if epoch == self.config.train.switch_off_augment
ations_on:
                train_loader.dataset.transform.switch_off_au
gmentations()
```

```
            self.train_epoch(
                train_loader, epoch,
                log_interval, write_summary=True
            )
            validation_score = self.validation(valid_loader,
 epoch)
            scores.append(validation_score)

            if epoch % self.config.train._save_every == 0:
                print("\nSaving model on epoch", epoch)
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "model_on_epoch_{}.pth".format(epoch
)
                    )
                )

            if validation_score > best_score:
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "best_model.pth"
                    )
                )
                best_score = validation_score

        return scores

    def make_optimizer(self, max_steps):

        optimizer = OPTIMIZERS[self.config.train.optimizer]
        optimizer = optimizer(
            self.parameters(),
            self.config.train.learning_rate,
            weight_decay=self.config.train.weight_decay
        )
        self.optimizer = optimizer
        self.scheduler = make_scheduler(
            self.config.train.scheduler, max_steps=max_steps
)(optimizer)

    def load_best_model(self, fold):

        self.load_state_dict(
            torch.load(
                os.path.join(
                    self.experiment.checkpoints,
```

```
                    "fold_{}".format(fold),
                    "best_model.pth"
                )
            )
        )
======
import os
import math
import itertools
from collections import defaultdict, OrderedDict, deque

from tqdm import tqdm
import numpy as np
import torch
import torch.nn as nn
import torchvision.utils
from tensorboardX import SummaryWriter
from torch.nn.functional import binary_cross_entropy_with_lo
gits


from ops.training import OPTIMIZERS, make_scheduler, make_st
ep
from networks.losses import binary_cross_entropy, focal_loss
, lsep_loss
from ops.utils import plot_projection


class CausalConv1d(nn.Module):

    def __init__(self, in_channels, out_channels, kernel_siz
e, stride=1):
        super().__init__()
        self.kernel_size = kernel_size
        self.conv = nn.Conv1d(
            in_channels, out_channels, kernel_size,
            stride=stride, padding=kernel_size)

    def forward(self, x):
        x = self.conv(x)
        return x[:, :, :-self.kernel_size]


class CPCModel(nn.Module):

    def __init__(self, experiment, device="cuda"):
        super().__init__()

        self.device = device

        self.experiment = experiment
        self.config = experiment.config
```

```
        encoder_layers = []

        for k in range(self.config.network.n_encoder_layers)
:
            input_size = self.config.data._input_dim if not
k else depth
            depth = int(
                self.config.network.growth_rate ** k
                * self.config.network.conv_base_depth)
            modules = [nn.BatchNorm1d(input_size)] if not k
else []
            modules.extend([
                CausalConv1d(
                    input_size,
                    depth,
                    kernel_size=3,
                    stride=2
                ),
                nn.PReLU(depth)
            ])
            encoder_layers.extend(modules)

        encoder_layers.append(nn.BatchNorm1d(depth))

        self.encoder = nn.Sequential(*encoder_layers)

        self.context_network = nn.GRU(
            depth, self.config.network.context_size,
            num_layers=1,
            batch_first=True
        )

        self.coupling_transforms = torch.nn.ModuleList([
            torch.nn.Sequential(
                torch.nn.Conv1d(
                    self.config.network.context_size, depth,
 kernel_size=1)
            )
            for steps in range(self.config.network.predictio
n_steps)
        ])

        self.to(self.device)

    def forward(self, signal):

        signal = signal.permute(0, 2, 1)

        # z is (n, depth, steps)
        z = self.encoder(signal)
        # c is (n, context_size, steps)
```

```
        c, state = self.context_network(z.permute(0, 2, 1))
        c = c.permute(0, 2, 1)

        losses = []

        for step, affine in enumerate(self.coupling_transfor
ms, start=1):

            a = affine(c)

            # logits is (n, steps, steps)
            logits = torch.bmm(z.permute(0, 2, 1), a)

            labels = torch.eye(logits.size(2) - step, device
=z.device)
            labels = torch.nn.functional.pad(labels, (0, ste
p, step, 0))
            labels = labels.unsqueeze(0).expand_as(logits)

            loss = binary_cross_entropy_with_logits(logits,
labels)
            losses.append(loss)

        r = dict(
            losses=losses,
            z=z,
            c=c
        )

        return r

    def add_scalar_summaries(
        self, losses, writer, global_step):

        # scalars
        for k, loss in enumerate(losses, start=1):
            writer.add_scalar("loss_{k}".format(k=k), loss,
global_step)

    def add_image_summaries(self, signal, c, z, global_step,
 writer, to_plot=8):

        if len(c) > to_plot:
            signal = signal[:to_plot]
            c = c[:to_plot]
            z = z[:to_plot]

        # signal
        image_grid = torchvision.utils.make_grid(
            signal.data.cpu().unsqueeze(1),
            normalize=True, scale_each=True
        )
```

```
        writer.add_image("signal", image_grid, global_step)
        # z
        image_grid = torchvision.utils.make_grid(
            z.data.cpu().unsqueeze(1),
            normalize=True, scale_each=True
        )
        writer.add_image("z", image_grid, global_step)
        # c
        image_grid = torchvision.utils.make_grid(
            c.data.cpu().unsqueeze(1),
            normalize=True, scale_each=True
        )
        writer.add_image("c", image_grid, global_step)

    def add_projection_summary(self, image, global_step, wri
ter, name="projection"):
        writer.add_image(name, image.transpose(2, 0, 1), glo
bal_step)

    def train_epoch(self, train_loader,
                    epoch, log_interval, write_summary=True)
:

        self.train()

        print(
            "\n" + " " * 10 + "****** Epoch {epoch} ******\n
"
            .format(epoch=epoch)
        )

        history = deque(maxlen=30)

        self.optimizer.zero_grad()
        accumulated_loss = 0

        with tqdm(total=len(train_loader), ncols=80) as pb:

            for batch_idx, sample in enumerate(train_loader)
:

                self.global_step += 1

                make_step(self.scheduler, step=self.global_s
tep)

                signal, labels = (
                    sample["signal"].to(self.device),
                    sample["labels"].to(self.device)
                )

                outputs = self(signal)
```

```
                losses = outputs["losses"]

                loss = (
                    sum(losses)
                ) / self.config.train.accumulation_steps

                loss.backward()
                accumulated_loss += loss

                if batch_idx % self.config.train.accumulatio
n_steps == 0:
                    self.optimizer.step()
                    accumulated_loss = 0
                    self.optimizer.zero_grad()

                history.append(loss.item())

                pb.update()
                pb.set_description(
                    "Loss: {:.4f}".format(
                        np.mean(history)))

                if batch_idx % log_interval == 0:
                    self.add_scalar_summaries(
                        [loss.item() for loss in losses],
                        self.train_writer, self.global_step)

                if batch_idx == 0:
                    self.add_image_summaries(
                        signal,
                        outputs["c"].permute(0, 2, 1),
                        outputs["z"].permute(0, 2, 1),
                        self.global_step, self.train_writer)

    def evaluate(self, loader, verbose=False, write_summary=
False, epoch=None):

        self.eval()

        valid_losses = [0 for _ in range(self.config.network
.prediction_steps)]

        all_c = []
        all_z = []
        all_labels = []

        with torch.no_grad():
            for batch_idx, sample in enumerate(loader):

                signal, labels = (
                    sample["signal"].to(self.device),
```

```
                        sample["labels"].to(self.device)
                    )

                    outputs = self(signal)

                    losses = outputs["losses"]

                    multiplier = len(signal) / len(loader.datase
t)

                    for k, loss in enumerate(losses):
                        valid_losses[k] += loss.item() * multipl
ier

                    all_c.extend(
                        outputs["c"].permute(0, 2, 1).data.cpu()
.numpy())
                    all_z.extend(
                        outputs["z"].permute(0, 2, 1).data.cpu()
.numpy())
                    all_labels.extend(labels.data.cpu().numpy())

        valid_loss = sum(valid_losses)

        all_labels = np.array(all_labels)

        if write_summary:
            self.add_scalar_summaries(
                valid_losses,
                writer=self.valid_writer, global_step=self.g
lobal_step
            )
            if epoch % self.config.train._proj_interval == 0
:
                self.add_projection_summary(
                    plot_projection(
                        all_c, all_labels, frames_per_exampl
e=5, newline=True),
                    writer=self.valid_writer, global_step=se
lf.global_step,
                    name="projection_c")
                self.add_projection_summary(
                    plot_projection(all_z, all_labels, frame
s_per_example=5),
                    writer=self.valid_writer, global_step=se
lf.global_step,
                    name="projection_z")

        if verbose:
            print("\nValidation loss: {:.4f}".format(valid_l
oss))
```

```
        return -valid_loss

    def validation(self, valid_loader, epoch):
        return self.evaluate(
            valid_loader,
            verbose=True, write_summary=True, epoch=epoch)

    def predict(self, loader):

        self.eval()

        all_class_probs = []

        with torch.no_grad():
            for sample in loader:

                signal = sample["signal"].to(self.device)

                outputs = self(signal)

                class_logits = outputs["class_logits"].squee
ze()

                class_probs = torch.sigmoid(class_logits).da
ta.cpu().numpy()
                all_class_probs.extend(class_probs)

        all_class_probs = np.asarray(all_class_probs)

        return all_class_probs

    def fit_validate(self, train_loader, valid_loader, epoch
s, fold,
                     log_interval=25):


        self.experiment.register_directory("summaries")
        self.train_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "train"
            )
        )
        self.valid_writer = SummaryWriter(
            log_dir=os.path.join(
                self.experiment.summaries,
                "fold_{}".format(fold),
                "valid"
            )
        )
```

```
        os.makedirs(
            os.path.join(
                self.experiment.checkpoints,
                "fold_{}".format(fold)),
            exist_ok=True
        )

        self.global_step = 0
        self.make_optimizer(max_steps=len(train_loader) * ep
ochs)

        scores = []
        best_score = 0

        for epoch in range(epochs):

            make_step(self.scheduler, epoch=epoch)

            if epoch == self.config.train.switch_off_augment
ations_on:
                train_loader.dataset.transform.switch_off_au
gmentations()

            self.train_epoch(
                train_loader, epoch,
                log_interval, write_summary=True
            )
            validation_score = self.validation(valid_loader,
 epoch)
            scores.append(validation_score)

            if epoch % self.config.train._save_every == 0:
                print("\nSaving model on epoch", epoch)
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "model_on_epoch_{}.pth".format(epoch
)
                    )
                )

            if validation_score > best_score:
                torch.save(
                    self.state_dict(),
                    os.path.join(
                        self.experiment.checkpoints,
                        "fold_{}".format(fold),
                        "best_model.pth"
                    )
                )
```

```
                best_score = validation_score

        return scores

    def make_optimizer(self, max_steps):

        optimizer = OPTIMIZERS[self.config.train.optimizer]
        optimizer = optimizer(
            self.parameters(),
            self.config.train.learning_rate,
            weight_decay=self.config.train.weight_decay
        )
        self.optimizer = optimizer
        self.scheduler = make_scheduler(
            self.config.train.scheduler, max_steps=max_steps
)(optimizer)

    def load_best_model(self, fold):

        self.load_state_dict(
            torch.load(
                os.path.join(
                    self.experiment.checkpoints,
                    "fold_{}".format(fold),
                    "best_model.pth"
                )
            )
        )

======
import torch
import torch.nn.functional as F


def focal_loss(input, target, focus=2.0, raw=True):

    if raw:
        input = torch.sigmoid(input)

    eps = 1e-7

    prob_true = input * target + (1 - input) * (1 - target)
    prob_true = torch.clamp(prob_true, eps, 1-eps)
    modulating_factor = (1.0 - prob_true).pow(focus)

    return (-modulating_factor * prob_true.log()).mean()


def binary_cross_entropy(input, target, raw=True):
    if raw:
        input = torch.sigmoid(input)
    return torch.nn.functional.binary_cross_entropy(input, t
```

```
arget)


def lsep_loss_stable(input, target, average=True):

    n = input.size(0)

    differences = input.unsqueeze(1) - input.unsqueeze(2)
    where_lower = (target.unsqueeze(1) < target.unsqueeze(2)
).float()

    differences = differences.view(n, -1)
    where_lower = where_lower.view(n, -1)

    max_difference, index = torch.max(differences, dim=1, ke
epdim=True)
    differences = differences - max_difference
    exps = differences.exp() * where_lower

    lsep = max_difference + torch.log(torch.exp(-max_differe
nce) + exps.sum(-1))

    if average:
        return lsep.mean()
    else:
        return lsep


def lsep_loss(input, target, average=True):

    differences = input.unsqueeze(1) - input.unsqueeze(2)
    where_different = (target.unsqueeze(1) < target.unsqueez
e(2)).float()

    exps = differences.exp() * where_different
    lsep = torch.log(1 + exps.sum(2).sum(1))

    if average:
        return lsep.mean()
    else:
        return lsep======
import random

import numpy as np
import librosa
import scipy.signal

from sklearn.utils import gen_even_slices


def compute_stft(audio, window_size, hop_size, log=True, eps
=1e-4):
```

```
    f, t, s = scipy.signal.stft(
        audio, nperseg=window_size, noverlap=hop_size)

    s = np.abs(s)

    if log:
        s = np.log(s + eps)

    return s


def trim_audio(audio):
    audio, interval = librosa.effects.trim(audio, top_db=60)
    return audio


def read_audio(file):
    audio, sr = librosa.load(file, sr=None)
    return audio, sr


def mix_audio_and_labels(first_audio, second_audio, first_la
bels, second_labels):

    new_labels = np.clip(first_labels + second_labels, 0, 1)

    a = np.random.uniform(0.4, 0.6)

    shorter, longer = first_audio, second_audio

    if shorter.size == longer.size:
        return (shorter + longer) / 2, new_labels

    if first_audio.size > second_audio.size:
        shorter, longer = longer, shorter

    start = random.randint(0, longer.size - 1 - shorter.size
)
    end = start + shorter.size

    longer *= a
    longer[start:end] =+ shorter * (1 - a)

    return longer, new_labels


def shuffle_audio(audio, chunk_length=0.5, sr=None):

    n_chunks = int((audio.size / sr) / chunk_length)

    if n_chunks in (0, 1):
        return audio
```

```
    slices = list(gen_even_slices(audio.size, n_chunks))
    random.shuffle(slices)

    shuffled = np.concatenate([audio[s] for s in slices])

    return shuffled


def cutout(audio, area=0.25):

    area = int(audio.size * area)

    start = random.randrange(audio.size)
    end = start + area

    audio[start:end] = 0

    return audio
======
from sklearn.model_selection import KFold
from iterstrat.ml_stratifiers import MultilabelStratifiedKFo
ld
import numpy as np


def train_validation_data(ids, labels, n_folds, seed):

    for train, valid in KFold(
        n_folds, shuffle=True, random_state=seed).split(ids,
 labels):
        yield train, valid


def train_validation_data_stratified(
    ids, labels, classmap, n_folds, seed):

    binary_labels = np.zeros(
        (len(labels), len(classmap)), dtype=np.float32)
    for k, item in enumerate(labels.values):
        for label in item.split(","):
            binary_labels[k, classmap[label]] = 1

    for train, valid in MultilabelStratifiedKFold(
        n_folds, shuffle=True, random_state=seed).split(ids,
 binary_labels):
        yield train, valid======
import random
from copy import deepcopy

import numpy as np
from torch.utils.data.dataloader import default_collate
```

```
def make_collate_fn(padding_values):

    def _collate_fn(batch):

        for name, padding_value in padding_values.items():

            lengths = [len(sample[name]) for sample in batch
]
            max_length = max(lengths)

            for n, size in enumerate(lengths):
                p = max_length - size
                if p:
                    pad_width = [(0, p)] + [(0, 0)] * (batch
[n][name].ndim - 1)
                    if padding_value == "edge":
                        batch[n][name] = np.pad(
                            batch[n][name], pad_width,
                            mode="edge")
                    else:
                        batch[n][name] = np.pad(
                            batch[n][name], pad_width,
                            mode="constant", constant_values
=padding_value)

        return default_collate(batch)

    return _collate_fn



class BucketingSampler:

    def __init__(self, dataset, max_batch_elems, buckets):

        self.buckets = buckets
        self.dataset = dataset
        self.max_batch_elems = max_batch_elems

        self._create_batches()

    def _create_batches(self):

        self.n_bins = len(self.buckets)
        binned_sizes = np.digitize(self.dataset.lengths, sel
f.buckets)

        batches = []

        for bin_idx in range(1, self.n_bins):
```

```
            ids = np.nonzero(binned_sizes == bin_idx)[0]
            random.shuffle(ids)

            current_len = 0
            batch = []

            for id in ids:
                if current_len < self.max_batch_elems:
                    batch.append(id)
                    current_len += self.dataset.lengths[id]
                else:
                    batches.append(batch)
                    current_len = self.dataset.lengths[id]
                    batch = [id]

            if batch:
                batches.append(batch)

        random.shuffle(batches)

        self.n_batches = len(batches)
        self.batches = batches

    def __iter__(self):
        return iter(self.batches)

    def __len__(self):
        return self.n_batches======
from functools import partial

import numpy as np
import torch
from torch.optim import Optimizer
from torch.optim.lr_scheduler import StepLR, CosineAnnealing
LR, _LRScheduler


OPTIMIZERS = {
    "adam": partial(torch.optim.Adam, amsgrad=True),
    "momentum": partial(torch.optim.SGD, momentum=0.9, neste
rov=True)
}


def make_scheduler(params, max_steps):

    name, *args = params.split("_")

    if name == "steplr":

        step_size, gamma = args
        step_size = int(step_size)
```

```
        gamma = float(gamma)

        return partial(StepLR, step_size=step_size, gamma=ga
mma)

    elif name == "1cycle":

        min_lr, max_lr = args
        min_lr = float(min_lr)
        max_lr = float(max_lr)

        return partial(
            OneCycleScheduler, min_lr=min_lr, max_lr=max_lr,
 max_steps=max_steps)


def make_step(scheduler, epoch=None, step=None, val_score=No
ne):

    if isinstance(scheduler, StepLR) and epoch is not None:
        scheduler.step(epoch)

    elif isinstance(scheduler, OneCycleScheduler) and step i
s not None:
        scheduler.step()


class CyclicLR:
    """Sets the learning rate of each parameter group accord
ing to
    cyclical learning rate policy (CLR). The policy cycles t
he learning
    rate between two boundaries with a constant frequency, a
s detailed in
    the paper `Cyclical Learning Rates for Training Neural N
etworks`_.
    The distance between the two boundaries can be scaled on
 a per-iteration
    or per-cycle basis.

    Cyclical learning rate policy changes the learning rate
after every batch.
    `batch_step` should be called after a batch has been use
d for training.
    To resume training, save `last_batch_iteration` and use
it to instantiate `CycleLR`.

    This class has three built-in policies, as put forth in
the paper:
    "triangular":
        A basic triangular cycle w/ no amplitude scaling.
    "triangular2":
```

```
        A basic triangular cycle that scales initial amplitu
de by half each cycle.
    "exp_range":
        A cycle that scales initial amplitude by gamma**(cyc
le iterations) at each
        cycle iteration.

    This implementation was adapted from the github repo: `b
ckenstler/CLR`_

    Args:
        optimizer (Optimizer): Wrapped optimizer.
        base_lr (float or list): Initial learning rate which
 is the
            lower boundary in the cycle for eachparam groups
.
            Default: 0.001
        max_lr (float or list): Upper boundaries in the cycl
e for
            each parameter group. Functionally,
            it defines the cycle amplitude (max_lr - base_lr
).
            The lr at any cycle is the sum of base_lr
            and some scaling of the amplitude; therefore
            max_lr may not actually be reached depending on
            scaling function. Default: 0.006
        step_size (int): Number of training iterations per
            half cycle. Authors suggest setting step_size
            2-8 x training iterations in epoch. Default: 200
0
        mode (str): One of {triangular, triangular2, exp_ran
ge}.
            Values correspond to policies detailed above.
            If scale_fn is not None, this argument is ignore
d.
            Default: 'triangular'
        gamma (float): Constant in 'exp_range' scaling funct
ion:
            gamma**(cycle iterations)
            Default: 1.0
        scale_fn (function): Custom scaling policy defined b
y a single
            argument lambda function, where
            0 <= scale_fn(x) <= 1 for all x >= 0.
            mode paramater is ignored
            Default: None
        scale_mode (str): {'cycle', 'iterations'}.
            Defines whether scale_fn is evaluated on
            cycle number or cycle iterations (training
            iterations since start of cycle).
            Default: 'cycle'
        last_batch_iteration (int): The index of the last ba
```

```
tch. Default: -1

    Example:
        >>> optimizer = torch.optim.SGD(model.parameters(),
lr=0.1, momentum=0.9)
        >>> scheduler = torch.optim.CyclicLR(optimizer)
        >>> data_loader = torch.utils.data.DataLoader(...)
        >>> for epoch in range(10):
        >>>     for batch in data_loader:
        >>>         scheduler.batch_step()
        >>>         train_batch(...)

    .. _Cyclical Learning Rates for Training Neural Networks
: https://arxiv.org/abs/1506.01186
    .. _bckenstler/CLR: https://github.com/bckenstler/CLR
    """

    def __init__(self, optimizer, base_lr=1e-3, max_lr=6e-3,
                 step_size=2000, mode='triangular', gamma=1.
,
                 scale_fn=None, scale_mode='cycle', last_bat
ch_iteration=-1):

        if not isinstance(optimizer, Optimizer):
            raise TypeError('{} is not an Optimizer'.format(
                type(optimizer).__name__))
        self.optimizer = optimizer

        if isinstance(base_lr, list) or isinstance(base_lr,
tuple):
            if len(base_lr) != len(optimizer.param_groups):
                raise ValueError("expected {} base_lr, got {
}".format(
                    len(optimizer.param_groups), len(base_lr
)))
            self.base_lrs = list(base_lr)
        else:
            self.base_lrs = [base_lr] * len(optimizer.param_
groups)

        if isinstance(max_lr, list) or isinstance(max_lr, tu
ple):
            if len(max_lr) != len(optimizer.param_groups):
                raise ValueError("expected {} max_lr, got {}
".format(
                    len(optimizer.param_groups), len(max_lr)
))
            self.max_lrs = list(max_lr)
        else:
            self.max_lrs = [max_lr] * len(optimizer.param_gr
oups)
```

```
        self.step_size = step_size

        if mode not in ['triangular', 'triangular2', 'exp_ra
nge'] \
                and scale_fn is None:
            raise ValueError('mode is invalid and scale_fn i
s None')

        self.mode = mode
        self.gamma = gamma

        if scale_fn is None:
            if self.mode == 'triangular':
                self.scale_fn = self._triangular_scale_fn
                self.scale_mode = 'cycle'
            elif self.mode == 'triangular2':
                self.scale_fn = self._triangular2_scale_fn
                self.scale_mode = 'cycle'
            elif self.mode == 'exp_range':
                self.scale_fn = self._exp_range_scale_fn
                self.scale_mode = 'iterations'
        else:
            self.scale_fn = scale_fn
            self.scale_mode = scale_mode

        self.batch_step(last_batch_iteration + 1)
        self.last_batch_iteration = last_batch_iteration

    def batch_step(self, batch_iteration=None):
        if batch_iteration is None:
            batch_iteration = self.last_batch_iteration + 1
        self.last_batch_iteration = batch_iteration
        for param_group, lr in zip(self.optimizer.param_grou
ps, self.get_lr()):
            param_group['lr'] = lr

    def _triangular_scale_fn(self, x):
        return 1.

    def _triangular2_scale_fn(self, x):
        return 1 / (2. ** (x - 1))

    def _exp_range_scale_fn(self, x):
        return self.gamma**(x)

    def get_lr(self):
        step_size = float(self.step_size)
        cycle = np.floor(1 + self.last_batch_iteration / (2
* step_size))
        x = np.abs(self.last_batch_iteration / step_size - 2
 * cycle + 1)
```

```
        lrs = []
        param_lrs = zip(self.optimizer.param_groups, self.ba
se_lrs, self.max_lrs)
        for param_group, base_lr, max_lr in param_lrs:
            base_height = (max_lr - base_lr) * np.maximum(0,
 (1 - x))
            if self.scale_mode == 'cycle':
                lr = base_lr + base_height * self.scale_fn(c
ycle)
            else:
                lr = base_lr + base_height * self.scale_fn(s
elf.last_batch_iteration)
            lrs.append(lr)
        return lrs


def annealing_linear(start, end, r):
    return start + r * (end - start)

def annealing_cos(start, end, r):
    cos_out = np.cos(np.pi * r) + 1
    return end + (start - end) / 2 * cos_out


class OneCycleScheduler:
    def __init__(
        self, optimizer,
        min_lr, max_lr,
        max_steps, annealing=annealing_linear):

        self.optimizer = optimizer
        self.min_lr = min_lr
        self.max_lr = max_lr
        self.max_steps = max_steps
        self.annealing = annealing
        self.epoch = -1

    def step(self):
        self.epoch += 1

        mid = int(round(self.max_steps * 0.3))

        if self.epoch < mid:
            r = self.epoch / mid
            lr = self.annealing(self.min_lr, self.max_lr, r)
        else:
            r = (self.epoch - mid) / (self.max_steps - mid)
            lr = self.annealing(self.max_lr, self.min_lr / 1
e3, r)

        for param_group in self.optimizer.param_groups:
            param_group['lr'] = lr
```

```
======
import random
import math
from functools import partial
import json

import pysndfx
import librosa
import numpy as np
import torch

from ops.audio import (
    read_audio, compute_stft, trim_audio, mix_audio_and_labe
ls,
    shuffle_audio, cutout
)


SAMPLE_RATE = 44100


class Augmentation:
    """A base class for data augmentation transforms"""
    pass


class MapLabels:

    def __init__(self, class_map, drop_raw=True):

        self.class_map = class_map

    def __call__(self, dataset, **inputs):

        labels = np.zeros(len(self.class_map), dtype=np.floa
t32)
        for c in inputs["raw_labels"]:
            labels[self.class_map[c]] = 1.0

        transformed = dict(inputs)
        transformed["labels"] = labels
        transformed.pop("raw_labels")

        return transformed


class MixUp(Augmentation):

    def __init__(self, p):

        self.p = p
```

```
    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        if np.random.uniform() < self.p:
            first_audio, first_labels = inputs["audio"], inp
uts["labels"]
            random_sample = dataset.random_clean_sample()
            new_audio, new_labels = mix_audio_and_labels(
                first_audio, random_sample["audio"],
                first_labels, random_sample["labels"]
            )

            transformed["audio"] = new_audio
            transformed["labels"] = new_labels

        return transformed


class FlipAudio(Augmentation):

    def __init__(self, p):

        self.p = p

    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        if np.random.uniform() < self.p:
            transformed["audio"] = np.flipud(inputs["audio"]
)

        return transformed


class AudioAugmentation(Augmentation):

    def __init__(self, p):

        self.p = p

    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        if np.random.uniform() < self.p:
            effects_chain = (
                pysndfx.AudioEffectsChain()
                .reverb(
                    reverberance=random.randrange(50),
```

```
                    room_scale=random.randrange(50),
                    stereo_depth=random.randrange(50)
                )
                .pitch(shift=random.randrange(-300, 300))
                .overdrive(gain=random.randrange(2, 10))
                .speed(random.uniform(0.9, 1.1))
            )
            transformed["audio"] = effects_chain(inputs["aud
io"])

        return transformed


class LoadAudio:

    def __init__(self):

        pass

    def __call__(self, dataset, **inputs):

        audio, sr = read_audio(inputs["filename"])

        transformed = dict(inputs)
        transformed["audio"] = audio
        transformed["sr"] = sr

        return transformed


class STFT:

    eps = 1e-4

    def __init__(self, n_fft, hop_size):

        self.n_fft = n_fft
        self.hop_size = hop_size

    def __call__(self, dataset, **inputs):

        stft = compute_stft(
            inputs["audio"],
            window_size=self.n_fft, hop_size=self.hop_size,
            eps=self.eps)

        transformed = dict(inputs)
        transformed["stft"] = np.transpose(stft)

        return transformed
```

```
class AudioFeatures:

    eps = 1e-4

    def __init__(self, descriptor, verbose=True):

        name, *args = descriptor.split("_")

        self.feature_type = name

        if name == "stft":

            n_fft, hop_size = args
            self.n_fft = int(n_fft)
            self.hop_size = int(hop_size)

            self.n_features = self.n_fft // 2 + 1
            self.padding_value = 0.0

            if verbose:
                print(
                    "\nUsing STFT features with params:\n",
                    "n_fft: {}, hop_size: {}".format(
                        n_fft, hop_size
                    )
                )

        elif name == "mel":

            n_fft, hop_size, n_mel = args
            self.n_fft = int(n_fft)
            self.hop_size = int(hop_size)
            self.n_mel = int(n_mel)

            self.n_features = self.n_mel
            self.padding_value = 0.0

            if verbose:
                print(
                    "\nUsing mel features with params:\n",
                    "n_fft: {}, hop_size: {}, n_mel: {}".for
mat(
                        n_fft, hop_size, n_mel
                    )
                )

        elif name == "raw":

            self.n_features = 1
            self.padding_value = 0.0

            if verbose:
```

```
                print(
                    "\nUsing raw waveform features."
                )


    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        if self.feature_type == "stft":

            # stft = compute_stft(
            #     inputs["audio"],
            #     window_size=self.n_fft, hop_size=self.hop_
size,
            #     eps=self.eps, log=True
            # )

            transformed["signal"] = np.expand_dims(inputs["a
udio"], -1)

        elif self.feature_type == "mel":

            stft = compute_stft(
                inputs["audio"],
                window_size=self.n_fft, hop_size=self.hop_si
ze,
                eps=self.eps, log=False
            )

            transformed["signal"] = np.expand_dims(inputs["a
udio"], -1)

        elif self.feature_type == "raw":
            transformed["signal"] = np.expand_dims(inputs["a
udio"], -1)

        return transformed


class SampleSegment(Augmentation):

    def __init__(self, ratio=(0.3, 0.9), p=1.0):

        self.min, self.max = ratio
        self.p = p

    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        if np.random.uniform() < self.p:
```

```
            original_size = inputs["audio"].size
            target_size = int(np.random.uniform(self.min, se
lf.max) * original_size)
            start = np.random.randint(original_size - target
_size - 1)
            transformed["audio"] = inputs["audio"][start:sta
rt+target_size]

        return transformed


class ShuffleAudio(Augmentation):

    def __init__(self, chunk_length=0.5, p=0.5):

        self.chunk_length = chunk_length
        self.p = p

    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        if np.random.uniform() < self.p:
            transformed["audio"] = shuffle_audio(
                transformed["audio"], self.chunk_length, sr=
transformed["sr"])

        return transformed


class CutOut(Augmentation):

    def __init__(self, area=0.25, p=0.5):

        self.area = area
        self.p = p

    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        if np.random.uniform() < self.p:
            transformed["audio"] = cutout(
                transformed["audio"], self.area)

        return transformed


class SampleLongAudio:

    def __init__(self, max_length):
```

```
        self.max_length = max_length

    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        if (inputs["audio"].size / inputs["sr"]) > self.max_
length:

            max_length = self.max_length * inputs["sr"]

            start = np.random.randint(0, inputs["audio"].siz
e - max_length)
            transformed["audio"] = inputs["audio"][start:sta
rt+max_length]

        return transformed


class OneOf:

    def __init__(self, transforms):

        self.transforms = transforms

    def __call__(self, dataset, **inputs):

        transform = random.choice(self.transforms)
        return transform(**inputs)


class DropFields:

    def __init__(self, fields):

        self.to_drop = fields

    def __call__(self, dataset, **inputs):

        transformed = dict()

        for name, input in inputs.items():
            if not name in self.to_drop:
                transformed[name] = input

        return transformed


class RenameFields:

    def __init__(self, mapping):
```

```
        self.mapping = mapping

    def __call__(self, dataset, **inputs):

        transformed = dict(inputs)

        for old, new in self.mapping.items():
            transformed[new] = transformed.pop(old)

        return transformed


class Compose:

    def __init__(self, transforms):
        self.transforms = transforms

    def switch_off_augmentations(self):
        for t in self.transforms:
            if isinstance(t, Augmentation):
                t.p = 0.0

    def __call__(self, dataset=None, **inputs):
        for t in self.transforms:
            inputs = t(dataset=dataset, **inputs)

        return inputs


class Identity:

    def __call__(self, dataset=None, **inputs):

        return inputs======
import json

import torch
import umap
import numpy as np
from sklearn.manifold import TSNE
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import label_ranking_average_precision_
score, accuracy_score
from matplotlib import pyplot as plt
import librosa


# Calculate the overall lwlrap using sklearn.metrics functio
n.
```

```
def lwlrap(truth, scores):
  """Calculate the overall lwlrap using sklearn.metrics.lrap
."""
  # sklearn doesn't correctly apply weighting to samples wit
h no labels, so just skip them.
  sample_weight = np.sum(truth > 0, axis=1)
  nonzero_weight_sample_indices = np.flatnonzero(sample_weig
ht > 0)
  overall_lwlrap = label_ranking_average_precision_score(
      truth[nonzero_weight_sample_indices, :] > 0,
      scores[nonzero_weight_sample_indices, :],
      sample_weight=sample_weight[nonzero_weight_sample_indi
ces])
  return overall_lwlrap


def load_json(file):
    with open(file, "r") as f:
        return json.load(f)


def get_class_names_from_classmap(classmap):
    r = dict((v, k) for k, v in classmap.items())
    return [r[label] for label in sorted(classmap.values())]


def plot_projection(vectors, labels, frames_per_example=3, n
ewline=False):

    representations = []
    classes = []
    for sample, label in zip(vectors, labels):
        if sum(label) > 1:
            continue
        choices = np.random.choice(
            np.arange(len(sample)), replace=False,
            size=min(frames_per_example, len(sample)))
        representations.extend(sample[choices])
        classes.extend([label.tolist().index(1)] * len(choic
es))

    representations = np.array(representations)

    # fit a simple model to estimate the quality of the lear
ned representations
    X_train, X_valid, y_train, y_valid = train_test_split(
        representations, classes, shuffle=False, test_size=0
.2)

    scaler = StandardScaler().fit(X_train)
    X_train = scaler.transform(X_train)
    X_valid = scaler.transform(X_valid)
```

```
    model = KNeighborsClassifier(n_neighbors=5)
    model.fit(X_train, y_train)

    score = accuracy_score(y_valid, model.predict(X_valid))
    if newline:
        print()
    print("Classification accuracy: {:.4f}".format(score))

    # plot projection
    embeddings = TSNE().fit_transform(representations)

    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111)
    ax.scatter(embeddings[:, 0], embeddings[:, 1], c=classes
, s=10)

    fig.canvas.draw()

    image = np.array(fig.canvas.renderer._renderer)

    plt.close()

    return image


def make_mel_filterbanks(descriptor, sr=44100):

    name, *args = descriptor.split("_")

    n_fft, hop_size, n_mel = args
    n_fft = int(n_fft)
    hop_size = int(hop_size)
    n_mel = int(n_mel)

    filterbank = librosa.filters.mel(
        sr=sr, n_fft=n_fft, n_mels=n_mel,
        fmin=5, fmax=None
    ).astype(np.float32)

    return filterbank


def is_mel(descriptor):
    return descriptor.startswith("mel")


def is_stft(descriptor):
    return descriptor.startswith("stft")


def compute_torch_stft(audio, descriptor):
```

```
    name, *args = descriptor.split("_")

    n_fft, hop_size, *rest = args
    n_fft = int(n_fft)
    hop_size = int(hop_size)

    stft = torch.stft(
        audio,
        n_fft=n_fft,
        hop_length=hop_size,
        window=torch.hann_window(n_fft, device=audio.device)
    )

    stft = torch.sqrt((stft ** 2).sum(-1))

    return stft

======
import os
import glob
import pickle
import random
import json

import torch
from tqdm import tqdm
import torch.utils.data as data
import numpy as np
import pandas as pd


class SoundDataset(data.Dataset):

    def __init__(
        self, audio_files, labels=None,
        transform=None, is_noisy=None, clean_transform=None)
:

        self.transform = transform
        self.clean_transform = clean_transform
        self.audio_files = audio_files
        self.labels = labels
        self.is_noisy = is_noisy or np.zeros(len(self.audio_
files))

    def __getitem__(self, index):

        sample = dict(
            filename=self.audio_files[index],
            is_noisy=self.is_noisy[index]
        )
```

```
        if self.labels is not None:
            sample["raw_labels"] = self.labels[index]

        if self.transform is not None:
            sample = self.transform(dataset=self, **sample)

        return sample

    def random_clean_sample(self):

        index = random.randint(0, len(self) - 1)

        sample = dict(
            filename=self.audio_files[index],
            is_noisy=self.is_noisy[index]
        )

        if self.labels is not None:
            sample["raw_labels"] = self.labels[index]

        if self.clean_transform is not None:
            sample = self.clean_transform(dataset=self, **sa
mple)

        return sample

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