imports

In [1]:
```
import gc
import os
import pickle
import random
import time
from collections import Counter, defaultdict
from functools import partial
from pathlib import Path
from psutil import cpu_count

import librosa
import numpy as np
import pandas as pd
from PIL import Image
from sklearn.model_selection import train_test_split
#from skmultilearn.model_selection import iterative_train_test_split

import torch
import torch.nn as nn
import torch.nn.functional as F
from fastprogress import master_bar, progress_bar
from torch.optim import Adam
from torch.optim.lr_scheduler import CosineAnnealingLR
from torch.utils.data import Dataset, DataLoader
from torchvision.transforms import transforms
```

In [2]:
```
torch.cuda.is_available()
```

Out[2]:
```
True
```

utils

In [3]:
```
def seed_everything(seed):
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.deterministic = True

SEED = 2019
seed_everything(SEED)
```

In [4]:
```
N_JOBS = cpu_count()
os.environ['MKL_NUM_THREADS'] = str(N_JOBS)
os.environ['OMP_NUM_THREADS'] = str(N_JOBS)
DataLoader = partial(DataLoader, num_workers=N_JOBS)
```

In [5]:
```
# from official code https://colab.research.google.com/drive/1AgPdhSp7ttY18O3fEoHOQKlt_3HJDLi8#scrollTo=cRCaCIb9oguU
def _one_sample_positive_class_precisions(scores, truth):
    """Calculate precisions for each true class for a single sample.

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

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


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

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

    Returns:
      per_class_lwlrap: np.array of (num_classes,) giving the lwlrap for each
        class.
      weight_per_class: np.array of (num_classes,) giving the prior of each
        class within the truth labels.  Then the overall unbalanced lwlrap is
        simply np.sum(per_class_lwlrap * weight_per_class)
    """
    assert truth.shape == scores.shape
    num_samples, num_classes = scores.shape
    # Space to store a distinct precision value for each class on each sample.
    # Only the classes that are true for each sample will be filled in.
    precisions_for_samples_by_classes = np.zeros((num_samples, num_classes))
    for sample_num in range(num_samples):
```

dataset

In [6]:
```
dataset_dir = Path('../input/freesound-audio-tagging-2019')
preprocessed_dir = Path('../input/fat2019_prep_mels1')
```

In [7]:
```
csvs = {
    'train_curated': dataset_dir / 'train_curated.csv',
    #'train_noisy': dataset_dir / 'train_noisy.csv',
    'train_noisy': preprocessed_dir / 'trn_noisy_best50s.csv',
    'sample_submission': dataset_dir / 'sample_submission.csv',
}

dataset = {
    'train_curated': dataset_dir / 'train_curated',
    'train_noisy': dataset_dir / 'train_noisy',
    'test': dataset_dir / 'test',
}

mels = {
    'train_curated': preprocessed_dir / 'mels_train_curated.pkl',
    'train_noisy': preprocessed_dir / 'mels_trn_noisy_best50s.pkl',
    'test': preprocessed_dir / 'mels_test.pkl',  # NOTE: this data doesn't w
ork at 2nd stage
}
```

In [8]:
```
train_curated = pd.read_csv(csvs['train_curated'])
train_noisy = pd.read_csv(csvs['train_noisy'])
train_df = pd.concat([train_curated, train_noisy], sort=True, ignore_index=T
rue)
train_df.head()
```

Out[8]:

	fname	labels	singled
0	0006ae4e.wav	Bark	NaN
1	0019ef41.wav	Raindrop	NaN
2	001ec0ad.wav	Finger_snapping	NaN
3	0026c7cb.wav	Run	NaN
4	0026f116.wav	Finger_snapping	NaN

In [9]:
```
test_df = pd.read_csv(csvs['sample_submission'])
test_df.head()
```

Out[9]:

	fname	Accelerating_and_revving_and_vroom	Accordion	Acoustic_guitar	Applause	Bark	Bass
0	000ccb97.wav	0	0	0	0	0	
1	0012633b.wav	0	0	0	0	0	
2	001ed5f1.wav	0	0	0	0	0	
3	00294be0.wav	0	0	0	0	0	
4	003fde7a.wav	0	0	0	0	0	

In [10]:
```
labels = test_df.columns[1:].tolist()
labels
```

Out[10]:
```
['Accelerating_and_revving_and_vroom',
 'Accordion',
 'Acoustic_guitar',
 'Applause',
 'Bark',
 'Bass_drum',
 'Bass_guitar',
 'Bathtub_(filling_or_washing)',
 'Bicycle_bell',
 'Burping_and_eructation',
 'Bus',
 'Buzz',
 'Car_passing_by',
 'Cheering',
 'Chewing_and_mastication',
 'Child_speech_and_kid_speaking',
 'Chink_and_clink',
 'Chirp_and_tweet',
 'Church_bell',
 'Clapping',
 'Computer_keyboard',
 'Crackle',
 'Cricket',
 'Crowd',
 'Cupboard_open_or_close',
 'Cutlery_and_silverware',
 'Dishes_and_pots_and_pans',
 'Drawer_open_or_close',
 'Drip',
 'Electric_guitar',
 'Fart',
 'Female_singing',
 'Female_speech_and_woman_speaking',
 'Fill_(with_liquid)',
 'Finger_snapping',
 'Frying_(food)',
 'Gasp',
 'Glockenspiel',
 'Gong',
 'Gurgling',
 'Harmonica',
 'Hi-hat',
 'Hiss',
 'Keys_jangling',
 'Knock',
 'Male_singing',
 'Male_speech_and_man_speaking',
 'Marimba_and_xylophone',
 'Mechanical_fan',
 'Meow',
 'Microwave_oven',
 'Motorcycle',
 'Printer',
 'Purr',
 'Race_car_and_auto_racing',
 'Raindrop',
 'Run',
 'Scissors',
 'Screaming',
 'Shatter',
 'Sigh',
 'Sink_(filling_or_washing)',
 'Skateboard',
 'Slam',
 'Sneeze',
 'Squeak',
 'Stream',
 'Strum',
```

In [11]:
```
num_classes = len(labels)
num_classes
```

Out[11]: 80

In [12]:
```
y_train = np.zeros((len(train_df), num_classes)).astype(int)
for i, row in enumerate(train_df['labels'].str.split(',')):
    for label in row:
        idx = labels.index(label)
        y_train[i, idx] = 1

y_train.shape
```

Out[12]: (8970, 80)

In [13]:
```
with open(mels['train_curated'], 'rb') as curated, open(mels['train_noisy'],
'rb') as noisy:
    x_train = pickle.load(curated)
    x_train.extend(pickle.load(noisy))

with open(mels['test'], 'rb') as test:
    x_test = pickle.load(test)

len(x_train), len(x_test)
```

Out[13]: (8970, 1120)

In [14]:
```
class FATTrainDataset(Dataset):
    def __init__(self, mels, labels, transforms):
        super().__init__()
        self.mels = mels
        self.labels = labels
        self.transforms = transforms

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

    def __getitem__(self, idx):
        # crop 1sec
        image = Image.fromarray(self.mels[idx], mode='RGB')
        time_dim, base_dim = image.size
        crop = random.randint(0, time_dim - base_dim)
        image = image.crop([crop, 0, crop + base_dim, base_dim])
        image = self.transforms(image).div_(255)

        label = self.labels[idx]
        label = torch.from_numpy(label).float()

        return image, label
```

In [15]:
```
class FATTestDataset(Dataset):
    def __init__(self, fnames, mels, transforms, tta=5):
        super().__init__()
        self.fnames = fnames
        self.mels = mels
        self.transforms = transforms
        self.tta = tta

    def __len__(self):
        return len(self.fnames) * self.tta

    def __getitem__(self, idx):
        new_idx = idx % len(self.fnames)

        image = Image.fromarray(self.mels[new_idx], mode='RGB')
        time_dim, base_dim = image.size
        crop = random.randint(0, time_dim - base_dim)
        image = image.crop([crop, 0, crop + base_dim, base_dim])
        image = self.transforms(image).div_(255)

        fname = self.fnames[new_idx]

        return image, fname
```

In [16]:
```
transforms_dict = {
    'train': transforms.Compose([
        transforms.RandomHorizontalFlip(0.5),
        transforms.ToTensor(),
    ]),
    'test': transforms.Compose([
        transforms.RandomHorizontalFlip(0.5),
        transforms.ToTensor(),
    ]),
}
```

model

In [17]:
```
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()

        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
        )

        self._init_weights()

    def _init_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight)
                if m.bias is not None:
                    nn.init.zeros_(m.bias)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.zeros_(m.bias)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = F.avg_pool2d(x, 2)
        return x
```

In [18]:
```
class Classifier(nn.Module):
    def __init__(self, num_classes):
        super().__init__()

        self.conv = nn.Sequential(
            ConvBlock(in_channels=3, out_channels=64),
            ConvBlock(in_channels=64, out_channels=128),
            ConvBlock(in_channels=128, out_channels=256),
            ConvBlock(in_channels=256, out_channels=512),
        )

        self.fc = nn.Sequential(
            nn.Dropout(0.2),
            nn.Linear(512, 128),
            nn.PReLU(),
            nn.BatchNorm1d(128),
            nn.Dropout(0.1),
            nn.Linear(128, num_classes),
        )

    def forward(self, x):
        x = self.conv(x)
        x = torch.mean(x, dim=3)
        x, _ = torch.max(x, dim=2)
        x = self.fc(x)
        return x
```

In [19]:
```
Classifier(num_classes=num_classes)
```

Out[19]:
```
Classifier(
  (conv): Sequential(
    (0): ConvBlock(
      (conv1): Sequential(
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
        (2): ReLU()
      )
      (conv2): Sequential(
        (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
        (2): ReLU()
      )
    )
    (1): ConvBlock(
      (conv1): Sequential(
        (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
        (2): ReLU()
      )
      (conv2): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
        (2): ReLU()
      )
    )
    (2): ConvBlock(
      (conv1): Sequential(
        (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
        (2): ReLU()
      )
      (conv2): Sequential(
        (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
        (2): ReLU()
      )
    )
    (3): ConvBlock(
      (conv1): Sequential(
        (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
        (2): ReLU()
      )
      (conv2): Sequential(
        (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
        (2): ReLU()
      )
    )
  )
  (fc): Sequential(
    (0): Dropout(p=0.2)
```

train

In [20]:
```
def train_model(x_train, y_train, train_transforms):
    num_epochs = 80
    batch_size = 64
    test_batch_size = 256
    lr = 3e-3
    eta_min = 1e-5
    t_max = 10

    num_classes = y_train.shape[1]

    x_trn, x_val, y_trn, y_val = train_test_split(x_train, y_train, test_siz
e=0.2, random_state=SEED)

    train_dataset = FATTrainDataset(x_trn, y_trn, train_transforms)
    valid_dataset = FATTrainDataset(x_val, y_val, train_transforms)

    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=
True)
    valid_loader = DataLoader(valid_dataset, batch_size=test_batch_size, shu
ffle=False)

    model = Classifier(num_classes=num_classes).cuda()
    criterion = nn.BCEWithLogitsLoss().cuda()
    optimizer = Adam(params=model.parameters(), lr=lr, amsgrad=False)
    scheduler = CosineAnnealingLR(optimizer, T_max=t_max, eta_min=eta_min)

    best_epoch = -1
    best_lwlrap = 0.
    mb = master_bar(range(num_epochs))

    for epoch in mb:
        start_time = time.time()
        model.train()
        avg_loss = 0.

        for x_batch, y_batch in progress_bar(train_loader, parent=mb):
            preds = model(x_batch.cuda())
            loss = criterion(preds, y_batch.cuda())

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            avg_loss += loss.item() / len(train_loader)

        model.eval()
        valid_preds = np.zeros((len(x_val), num_classes))
        avg_val_loss = 0.

        for i, (x_batch, y_batch) in enumerate(valid_loader):
            preds = model(x_batch.cuda()).detach()
            loss = criterion(preds, y_batch.cuda())

            preds = torch.sigmoid(preds)
            valid_preds[i * test_batch_size: (i+1) * test_batch_size] = pred
s.cpu().numpy()

            avg_val_loss += loss.item() / len(valid_loader)

        score, weight = calculate_per_class_lwlrap(y_val, valid_preds)
        lwlrap = (score * weight).sum()

        scheduler.step()

        if (epoch + 1) % 5 == 0:
            elapsed = time.time() - start_time
            mb.write(f'Epoch {epoch+1} - avg_train_loss: {avg_loss:.4f}  avg
 val_loss: {avg_val_loss:.4f}  val_lwlrap: {lwlrap:.6f}  time: {elapsed:.0f}
```

In [21]:
```
result = train_model(x_train, y_train, transforms_dict['train'])
```

```
31.25% [25/80 12:25<27:20]
Epoch 5 - avg_train_loss: 0.0623 avg_val_loss: 0.0768 val_lwlrap: 0.081251 time: 30s

Epoch 10 - avg_train_loss: 0.0561 avg_val_loss: 0.0842 val_lwlrap: 0.145102 time: 30s

Epoch 15 - avg_train_loss: 0.0554 avg_val_loss: 0.2979 val_lwlrap: 0.099892 time: 30s

Epoch 20 - avg_train_loss: 0.0549 avg_val_loss: 0.1380 val_lwlrap: 0.070345 time: 30s

Epoch 25 - avg_train_loss: 0.0501 avg_val_loss: 1.0152 val_lwlrap: 0.084374 time: 30s

43.36% [49/113 00:11<00:15]
```

In [22]:
```
result
```

Out[22]:
```
{'best_epoch': 72, 'best_lwlrap': 0.6313720751689541}
```

predict

In [23]:
```
def predict_model(test_fnames, x_test, test_transforms, num_classes, *, tta=
5):
    batch_size = 256

    test_dataset = FATTestDataset(test_fnames, x_test, test_transforms, tta=
tta)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=Fa
lse)

    model = Classifier(num_classes=num_classes)
    model.load_state_dict(torch.load('weight_best.pt'))
    model.cuda()
    model.eval()

    all_outputs, all_fnames = [], []

    pb = progress_bar(test_loader)
    for images, fnames in pb:
        preds = torch.sigmoid(model(images.cuda()).detach())
        all_outputs.append(preds.cpu().numpy())
        all_fnames.extend(fnames)

    test_preds = pd.DataFrame(data=np.concatenate(all_outputs),
                              index=all_fnames,
                              columns=map(str, range(num_classes)))
    test_preds = test_preds.groupby(level=0).mean()

    return test_preds
```

In [24]:
```
test_preds = predict_model(test_df['fname'], x_test, transforms_dict['test
'], num_classes, tta=20)
```

```
100.00% [88/88 00:27<00:00]
```

In [25]:
```
test_df[labels] = test_preds.values
test_df.to_csv('submission.csv', index=False)
test_df.head()
```

Out[25]:

	fname	Accelerating_and_revving_and_vroom	Accordion	Acoustic_guitar	Applause	Ba
0	000ccb97.wav	0.000001	2.923243e-09	7.161557e-08	1.016948e-06	0.0000
1	0012633b.wav	0.052353	1.996648e-04	2.096502e-03	2.918867e-03	0.0008
2	001ed5f1.wav	0.000172	1.197935e-04	6.463896e-06	1.389198e-03	0.0011
3	00294be0.wav	0.000002	1.181921e-11	4.253022e-08	2.461792e-08	0.0000
4	003fde7a.wav	0.000060	9.370017e-05	2.245676e-05	4.934249e-05	0.0000