



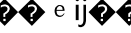


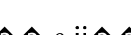


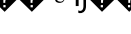



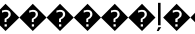



Task description

- Classify the speakers of given features.
- Main goal: Learn how to use transformer.
- Baselines:
 - Easy: Run sample code and know how to use transformer.
 - Medium: Know how to adjust parameters of transformer.
 - Hard: Construct [conformer](https://arxiv.org/abs/2005.08100) (<https://arxiv.org/abs/2005.08100>), which is a variety of transformer.

Download dataset

'wget'  X  e ij 
 'wget'  X  e ij 
 'wget'  X  e ij 
 'wget'  X  e ij 
 'wget'  X  e ij 
 'wget'  X  e ij 
 'cat'  X  e ij 

Dataset

- Original dataset is [Voxceleb1](https://www.robots.ox.ac.uk/~vgg/data/voxceleb/) (<https://www.robots.ox.ac.uk/~vgg/data/voxceleb/>).
- The [license](https://creativecommons.org/licenses/by/4.0/) (<https://creativecommons.org/licenses/by/4.0/>) and [complete version](https://www.robots.ox.ac.uk/~vgg/data/voxceleb/files/license.txt) (<https://www.robots.ox.ac.uk/~vgg/data/voxceleb/files/license.txt>) of Voxceleb1.
- We randomly select 600 speakers from Voxceleb1.
- Then preprocess the raw waveforms into mel-spectrograms.
- Args:
 - data_dir: The path to the data directory.
 - metadata_path: The path to the metadata.
 - segment_len: The length of audio segment for training.
- The architecture of data directory \
 - data directory \ |---- metadata.json \ |---- testdata.json \ |---- mapping.json \ |---- uttr-{random string}.pt \
- The information in metadata
 - "n_mels": The dimension of mel-spectrogram.
 - "speakers": A dictionary.
 - Key: speaker ids.
 - value: "feature_path" and "mel_len"

For efficiency, we segment the mel-spectrograms into segments in the training step.

```
In [ ]: import os
import json
import torch
import random
from pathlib import Path
from torch.utils.data import Dataset
from torch.nn.utils.rnn import pad_sequence

class myDataset(Dataset):
    def __init__(self, data_dir, segment_len=128):
        self.data_dir = data_dir
        self.segment_len = segment_len

        # Load the mapping from speaker name to their corresponding id.
        mapping_path = Path(data_dir) / "mapping.json"
        mapping = json.load(mapping_path.open())
        self.speaker2id = mapping["speaker2id"]

        # Load metadata of training data.
        metadata_path = Path(data_dir) / "metadata.json"
        metadata = json.load(open(metadata_path))["speakers"]

        # Get the total number of speaker.
        self.speaker_num = len(metadata.keys())
        self.data = []
        for speaker in metadata.keys():
            for utterances in metadata[speaker]:
                self.data.append([utterances["feature_path"], self.speaker2id[speaker]])

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

    def __getitem__(self, index):
        feat_path, speaker = self.data[index]
        # Load preprocessed mel-spectrogram.
        mel = torch.load(os.path.join(self.data_dir, feat_path))

        # Segment mel-spectrogram into "segment_len" frames.
        if len(mel) > self.segment_len:
            # Randomly get the starting point of the segment.
```

```
        start = random.randint(0, len(mel) - self.segment_len)
        # Get a segment with "segment_len" frames.
        mel = torch.FloatTensor(mel[start:start+self.segment_len])
    else:
        mel = torch.FloatTensor(mel)
    # Turn the speaker id into long for computing loss later.
    speaker = torch.FloatTensor([speaker]).long()
    return mel, speaker

def get_speaker_number(self):
    return self.speaker_num
```

Dataloader

- Split dataset into training dataset(90%) and validation dataset(10%).
- Create dataloader to iterate the data.

```
In [ ]: import torch
from torch.utils.data import DataLoader, random_split
from torch.nn.utils.rnn import pad_sequence

def collate_batch(batch):
    # Process features within a batch.
    """Collate a batch of data."""
    mel, speaker = zip(*batch)
    # Because we train the model batch by batch, we need to pad the features in the same batch to make their lengths the same.
    mel = pad_sequence(mel, batch_first=True, padding_value=-20) # pad log 10^(-20) which is very small value.
    # mel: (batch size, length, 40)
    return mel, torch.FloatTensor(speaker).long()

def get_dataloader(data_dir, batch_size, n_workers):
    """Generate dataloader"""
    dataset = myDataset(data_dir)
    speaker_num = dataset.get_speaker_number()
    # Split dataset into training dataset and validation dataset
    trainlen = int(0.9 * len(dataset))
    lengths = [trainlen, len(dataset) - trainlen]
    trainset, validset = random_split(dataset, lengths)

    train_loader = DataLoader(
        trainset,
        batch_size=batch_size,
        shuffle=True,
        drop_last=True,
        num_workers=n_workers,
        pin_memory=True,
        collate_fn=collate_batch,
    )
    valid_loader = DataLoader(
        validset,
        batch_size=batch_size,
        num_workers=n_workers,
        drop_last=True,
        pin_memory=True,
        collate_fn=collate_batch,
    )
```

```
return train_loader, valid_loader, speaker_num
```

Model

- TransformerEncoderLayer:
 - Base transformer encoder layer in [Attention Is All You Need \(https://arxiv.org/abs/1706.03762\)](https://arxiv.org/abs/1706.03762)
 - Parameters:
 - d_model: the number of expected features of the input (required).
 - nhead: the number of heads of the multiheadattention models (required).
 - dim_feedforward: the dimension of the feedforward network model (default=2048).
 - dropout: the dropout value (default=0.1).
 - activation: the activation function of intermediate layer, relu or gelu (default=relu).
- TransformerEncoder:
 - TransformerEncoder is a stack of N transformer encoder layers
 - Parameters:
 - encoder_layer: an instance of the TransformerEncoderLayer() class (required).
 - num_layers: the number of sub-encoder-layers in the encoder (required).
 - norm: the layer normalization component (optional).

```

In [ ]: import torch
import torch.nn as nn
import torch.nn.functional as F

class Classifier(nn.Module):
    def __init__(self, d_model=80, n_spks=600, dropout=0.1):
        super().__init__()
        # Project the dimension of features from that of input into d_model.
        self.prenet = nn.Linear(40, d_model)
        # TODO:
        #   Change Transformer to Conformer.
        #   https://arxiv.org/abs/2005.08100
        self.encoder_layer = nn.TransformerEncoderLayer(
            d_model=d_model, dim_feedforward=256, nhead=2
        )
        # self.encoder = nn.TransformerEncoder(self.encoder_layer, num_layers=2)

        # Project the the dimension of features from d_model into speaker nums.
        self.pred_layer = nn.Sequential(
            nn.Linear(d_model, d_model),
            nn.ReLU(),
            nn.Linear(d_model, n_spks),
        )

    def forward(self, mels):
        """
        args:
            mels: (batch size, length, 40)
        return:
            out: (batch size, n_spks)
        """
        # out: (batch size, length, d_model)
        out = self.prenet(mels)
        # out: (length, batch size, d_model)
        out = out.permute(1, 0, 2)
        # The encoder layer expect features in the shape of (length, batch size, d_model).
        out = self.encoder_layer(out)
        # out: (batch size, length, d_model)
        out = out.transpose(0, 1)
        # mean pooling

```



```
stats = out.mean(dim=1)

# out: (batch, n_spks)
out = self.pred_layer(stats)
return out
```

Learning rate schedule

- For transformer architecture, the design of learning rate schedule is different from that of CNN.
- Previous works show that the warmup of learning rate is useful for training models with transformer architectures.
- The warmup schedule
 - Set learning rate to 0 in the beginning.
 - The learning rate increases linearly from 0 to initial learning rate during warmup period.

```

In [ ]: import math

import torch
from torch.optim import Optimizer
from torch.optim.lr_scheduler import LambdaLR

def get_cosine_schedule_with_warmup(
    optimizer: Optimizer,
    num_warmup_steps: int,
    num_training_steps: int,
    num_cycles: float = 0.5,
    last_epoch: int = -1,
):
    """
    Create a schedule with a learning rate that decreases following the values of the cosine function between the
    initial lr set in the optimizer to 0, after a warmup period during which it increases linearly between 0 and the
    initial lr set in the optimizer.

    Args:
        optimizer (:class:`~torch.optim.Optimizer`):
            The optimizer for which to schedule the learning rate.
        num_warmup_steps (:obj:`int`):
            The number of steps for the warmup phase.
        num_training_steps (:obj:`int`):
            The total number of training steps.
        num_cycles (:obj:`float`, `optional`, defaults to 0.5):
            The number of waves in the cosine schedule (the defaults is to just decrease from the max value to 0
            following a half-cosine).
        last_epoch (:obj:`int`, `optional`, defaults to -1):
            The index of the last epoch when resuming training.

    Return:
        :obj:`torch.optim.lr_scheduler.LambdaLR` with the appropriate schedule.
    """

    def lr_lambda(current_step):
        # Warmup
        if current_step < num_warmup_steps:
            return float(current_step) / float(max(1, num_warmup_steps))
        # decadence

```

```
progress = float(current_step - num_warmup_steps) / float(
    max(1, num_training_steps - num_warmup_steps)
)
return max(
    0.0, 0.5 * (1.0 + math.cos(math.pi * float(num_cycles) * 2.0 * progress))
)

return LambdaLR(optimizer, lr_lambda, last_epoch)
```

Model Function

- Model forward function.

```
In [ ]: import torch

def model_fn(batch, model, criterion, device):
    """Forward a batch through the model."""

    mels, labels = batch
    mels = mels.to(device)
    labels = labels.to(device)

    outs = model(mels)

    loss = criterion(outs, labels)

    # Get the speaker id with highest probability.
    preds = outs.argmax(1)
    # Compute accuracy.
    accuracy = torch.mean((preds == labels).float())

    return loss, accuracy
```

Validate

- Calculate accuracy of the validation set.

```
In [ ]: from tqdm import tqdm
import torch

def valid(dataloader, model, criterion, device):
    """Validate on validation set."""

    model.eval()
    running_loss = 0.0
    running_accuracy = 0.0
    pbar = tqdm(total=len(dataloader.dataset), ncols=0, desc="Valid", unit=" uttr")

    for i, batch in enumerate(dataloader):
        with torch.no_grad():
            loss, accuracy = model_fn(batch, model, criterion, device)
            running_loss += loss.item()
            running_accuracy += accuracy.item()

        pbar.update(dataloader.batch_size)
        pbar.set_postfix(
            loss=f"{running_loss / (i+1):.2f}",
            accuracy=f"{running_accuracy / (i+1):.2f}",
        )

    pbar.close()
    model.train()

    return running_accuracy / len(dataloader)
```

Main function

```
In [ ]: from tqdm import tqdm

import torch
import torch.nn as nn
from torch.optim import AdamW
from torch.utils.data import DataLoader, random_split

def parse_args():
    """arguments"""
    config = {
        "data_dir": "./Dataset",
        "save_path": "model.ckpt",
        "batch_size": 32,
        "n_workers": 8,
        "valid_steps": 2000,
        "warmup_steps": 1000,
        "save_steps": 10000,
        "total_steps": 70000,
    }

    return config

def main(
    data_dir,
    save_path,
    batch_size,
    n_workers,
    valid_steps,
    warmup_steps,
    total_steps,
    save_steps,
):
    """Main function."""
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"[Info]: Use {device} now!")

    train_loader, valid_loader, speaker_num = get_dataloader(data_dir, batch_size, n_workers)
    train_iterator = iter(train_loader)
    print(f"[Info]: Finish loading data!", flush = True)
```

```
model = Classifier(n_spks=speaker_num).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = AdamW(model.parameters(), lr=1e-3)
scheduler = get_cosine_schedule_with_warmup(optimizer, warmup_steps, total_steps)
print(f"[Info]: Finish creating model!", flush = True)

best_accuracy = -1.0
best_state_dict = None

pbar = tqdm(total=valid_steps, ncols=0, desc="Train", unit=" step")

for step in range(total_steps):
    # Get data
    try:
        batch = next(train_iterator)
    except StopIteration:
        train_iterator = iter(train_loader)
        batch = next(train_iterator)

    loss, accuracy = model_fn(batch, model, criterion, device)
    batch_loss = loss.item()
    batch_accuracy = accuracy.item()

    # Updata model
    loss.backward()
    optimizer.step()
    scheduler.step()
    optimizer.zero_grad()

    # Log
    pbar.update()
    pbar.set_postfix(
        loss=f"{batch_loss:.2f} ",
        accuracy=f"{batch_accuracy:.2f} ",
        step=step + 1,
    )

    # Do validation
    if (step + 1) % valid_steps == 0:
        pbar.close()
```

```

valid_accuracy = valid(valid_loader, model, criterion, device)

# keep the best model
if valid_accuracy > best_accuracy:
    best_accuracy = valid_accuracy
    best_state_dict = model.state_dict()

pbar = tqdm(total=valid_steps, ncols=0, desc="Train", unit=" step")

# Save the best model so far.
if (step + 1) % save_steps == 0 and best_state_dict is not None:
    torch.save(best_state_dict, save_path)
    pbar.write(f"Step {step + 1}, best model saved. (accuracy={best_accuracy:.4f})")

pbar.close()

if __name__ == "__main__":
    main(**parse_args())

```

[Info]: Use cuda now!

[Info]: Finish loading data!

[Info]: Finish creating model!

```

Train: 100% 2000/2000 [03:34<00:00, 9.32 step/s, accuracy=0.31, loss=3.89, step=2000]
Valid: 100% 6944/6944 [00:23<00:00, 300.22 uttr/s, accuracy=0.18, loss=3.98]
Train: 100% 2000/2000 [03:04<00:00, 10.81 step/s, accuracy=0.19, loss=3.79, step=4000]
Valid: 100% 6944/6944 [00:24<00:00, 285.84 uttr/s, accuracy=0.29, loss=3.27]
Train: 100% 2000/2000 [02:50<00:00, 11.74 step/s, accuracy=0.41, loss=2.76, step=6000]
Valid: 100% 6944/6944 [00:25<00:00, 273.22 uttr/s, accuracy=0.35, loss=2.98]
Train: 100% 2000/2000 [02:47<00:00, 11.93 step/s, accuracy=0.53, loss=2.37, step=8000]
Valid: 100% 6944/6944 [00:25<00:00, 269.05 uttr/s, accuracy=0.41, loss=2.68]
Train: 71% 1416/2000 [02:01<01:06, 8.79 step/s, accuracy=0.50, loss=2.06, step=9416]

```

Inference

Dataset of inference

```
In [ ]: import os
import json
import torch
from pathlib import Path
from torch.utils.data import Dataset

class InferenceDataset(Dataset):
    def __init__(self, data_dir):
        testdata_path = Path(data_dir) / "testdata.json"
        metadata = json.load(testdata_path.open())
        self.data_dir = data_dir
        self.data = metadata["utterances"]

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

    def __getitem__(self, index):
        utterance = self.data[index]
        feat_path = utterance["feature_path"]
        mel = torch.load(os.path.join(self.data_dir, feat_path))

        return feat_path, mel

def inference_collate_batch(batch):
    """Collate a batch of data."""
    feat_paths, mels = zip(*batch)

    return feat_paths, torch.stack(mels)
```

Main funcrion of Inference


```
In [ ]: import json
import csv
from pathlib import Path
from tqdm.notebook import tqdm

import torch
from torch.utils.data import DataLoader

def parse_args():
    """arguments"""
    config = {
        "data_dir": "./Dataset",
        "model_path": "./model.ckpt",
        "output_path": "./output.csv",
    }

    return config

def main(
    data_dir,
    model_path,
    output_path,
):
    """Main function."""
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"[Info]: Use {device} now!")

    mapping_path = Path(data_dir) / "mapping.json"
    mapping = json.load(mapping_path.open())

    dataset = InferenceDataset(data_dir)
    dataloader = DataLoader(
        dataset,
        batch_size=1,
        shuffle=False,
        drop_last=False,
        num_workers=8,
        collate_fn=inference_collate_batch,
    )
    print(f"[Info]: Finish loading data!", flush = True)
```

```
speaker_num = len(mapping["id2speaker"])
model = Classifier(n_spks=speaker_num).to(device)
model.load_state_dict(torch.load(model_path))
model.eval()
print(f"[Info]: Finish creating model!", flush = True)

results = [["Id", "Category"]]
for feat_paths, mels in tqdm(dataloader):
    with torch.no_grad():
        mels = mels.to(device)
        outs = model(mels)
        preds = outs.argmax(1).cpu().numpy()
        for feat_path, pred in zip(feat_paths, preds):
            results.append([feat_path, mapping["id2speaker"][str(pred)]])

with open(output_path, 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerows(results)

if __name__ == "__main__":
    main(**parse_args())
```