model_exploration

November 11, 2020

[1]:
```
import os
import yaml

import librosa.display
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
```

[2]:
```
ROOT = '/home/thomas/Dir/ccny/ccny-masters-thesis'
```

[3]:
```
os.chdir(os.path.join(ROOT, 'tensorflow'))
from dataset import GE2EDatasetLoader
from loss import *
```

1 Load

In the process of training, we output two separate model objects. The first, is the `full` version, which simply uses the `model.save()` method as described here. The second is the `embedding` model which has the cosine similarity layer at the end removed, thus its output is the *d*-vector. In the prediction step, all we care about is the eventual output of this embedding layer, which should serve as the speaker's *voiceprint* for a given utterance. When enrolling new speakers, typically we take a few utterances and *average* them to get their global voiceprint.

This is, however, not ultimately the model deployed on the edge device, as we must first compress it using TFLite (discussed below).

Here, we'll load the `embedding` model to get a sense of its performance on the task of speaker recognition and, more importantly, separation within the embedding space. We can also explore its capabilities with regards to *enrolling* new and unseen speakers.

[4]:
```
embedding_models_path = os.path.join(ROOT, 'tensorflow/frozen_models/embedding')
full_models_path = os.path.join(ROOT, 'tensorflow/frozen_models/full')
```

[5]:
```
os.listdir(embedding_models_path)
```

[5]:
```
['1605081214']
```

[6]:
```
most_recent_embedding_model = max([ directory for directory in os.
↪listdir(embedding_models_path) ])
most_recent_full_model= max([ directory for directory in os.
↪listdir(full_models_path) ])
```

[7]:
```
# print out the conf for this epoch
model_conf = os.path.join(ROOT, 'tensorflow/frozen_models/confs',␣
↪f'{most_recent_embedding_model}.yaml')
with open(model_conf, 'r') as stream:
    d = yaml.safe_load(stream)
d
```

[7]:

```
{'sr': 16000,
 'raw_data': {'path': '/media/thomas/TPD EX/thesis-data',
  'datasets': {'train': {'LibriSpeech': ['train-clean-100',
     'train-clean-360',
     'train-other-500'],
    'VoxCeleb1': ['dev']},
   'test': {'out_of_sample': {'LibriSpeech': ['test-clean', 'test-other']}}}},
 'feature_data': {'path': '/home/thomas/Dir/ccny/ccny-masters-thesis/feature-
data',
  'speakers_per_batch': 8,
  'utterances_per_speaker': 8},
 'features': {'type': 'melspectrogram',
  'window_length': 1.2,
  'overlap_percent': 0.5,
  'frame_length': 0.025,
  'hop_length': 0.01,
  'n_fft': 512,
  'n_mels': 40,
  'trim_top_db': 20},
 'train': {'epochs': 50,
  'network': {'optimizer': {'type': 'SGD', 'lr': 0.01, 'clipnorm': 3.0},
   'dropout': 0.1,
   'layers': [{'lstm': {'units': 128, 'return_sequences': True}},
    {'lstm': {'units': 128}},
    {'embedding': {'nodes': 128}},
    {'similarity_matrix': {'embedding_length': 128}}],
   'callbacks': {'lr_scheduler': {'cutoff_epoch': 25, 'decay': 'exponential'},
    'csv_logger': {'dir': 'training_logs'},
    'checkpoint': {'dir': 'model_checkpoints'}}}}}
```


[8]:
```
embedding_model = tf.keras.models.load_model(os.path.
↪join(embedding_models_path, most_recent_embedding_model))
embedding_model.summary()
```

```
WARNING:tensorflow:No training configuration found in save file, so the model
```

```
was *not* compiled. Compile it manually.

[WARNING] {tensorflow} 2020-11-11 19:23:32,613 No training configuration found
in save file, so the model was *not* compiled. Compile it manually.

Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 40, 128)           128000
_________________________________________________________________
lstm_1 (LSTM)                (None, 128)               131584
_________________________________________________________________
dense (Dense)                (None, 128)               16512
=================================================================
Total params: 276,096
Trainable params: 276,096
Non-trainable params: 0
_________________________________________________________________
```

[9]:

```
full_model = tf.keras.models.load_model(os.path.join(full_models_path,
 →most_recent_full_model), compile=False)
full_model.compile(
    loss=get_embedding_loss(N=8, M=8)
)
full_model.summary()
```

```
Model: "speaker_verification_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 40, 128)           128000
_________________________________________________________________
lstm_1 (LSTM)                (None, 128)               131584
_________________________________________________________________
dense (Dense)                (None, 128)               16512
_________________________________________________________________
speaker_similarity_matrix_la (64, 8)                   2
_________________________________________________________________
sequential (Sequential)      (64, 8)                   276098
=================================================================
Total params: 276,098
Trainable params: 276,098
Non-trainable params: 0
_________________________________________________________________
```

2 Example Predict

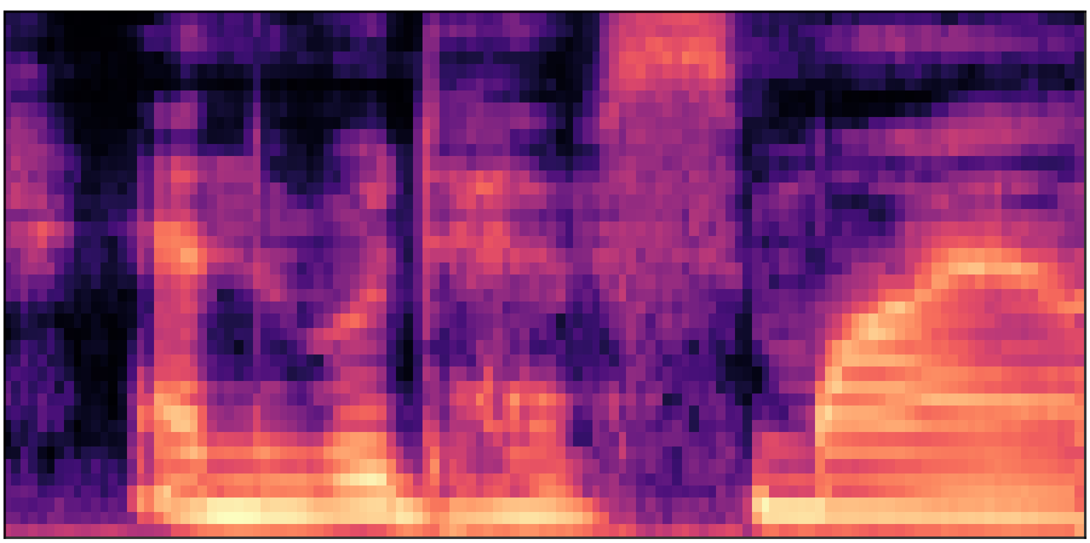
Now that we have de-serialized our model into `m`, it's only useful if we can make predictions. Here we'll demonstrate how to use the `__call__` method to generate embedding for a batch from the training dataset. Notice, the first dimension of each layer is `None`, which corresponds to our `batch_size`. This means we should be able to pass a feature of arbitrary batches, e.g. $[N, h, w]$ where N is the number of samples, h is the height of the spectrogram feature and w is the width of the spectrogram feature. This is preferable than needing to batch examples in a specific number at inference time.

[10]:
```
dataset = GE2EDatasetLoader(
    root_dir=os.path.join(ROOT, 'feature-data')
)
metadata = dataset.get_metadata()
inputs, targets = dataset.get_single_train_batch()
```

[11]:
```
inputs.shape
```

[11]:
```
TensorShape([64, 40, 121])
```

[12]:
```
# here's a sample feature: Mel-based spectrogram
plt.figure(figsize=(8,4))
_ = librosa.display.specshow(inputs[0].numpy())
```



[13]:
```
embedding_length = embedding_model.layers[-1].output.shape[1]
embedding_length
```

[13]:
```
128
```

[14]:
```
embeddings = embedding_model(inputs)
embeddings.shape # [batch_size x embedding_length]
```

[14]: TensorShape([64, 128])

[15]:
```
# how to visualize?
```

3 Speaker separation

Now that we've demonstrated how to generate predictions, we'll explore how our model performs at its primary task: separating unique speakers within the embedding space. Given that we're working in a high dimensional space, we'll need to use techniques that will allow us humans to actually *understand* that separation.

[16]:
```
train_dataset, test_dataset = dataset.get_datasets()
train_it, test_it = iter(train_dataset), iter(test_dataset)

metadata = dataset.get_metadata()
speaker_id_mapping = { v:k for k, v in metadata['speaker_id_mapping'].items() }
```

[17]:
```
def get_orig_speakers(tensor, mapping):
    return [mapping[t] for t in tensor.numpy()]
```

3.0.1 t-SNE

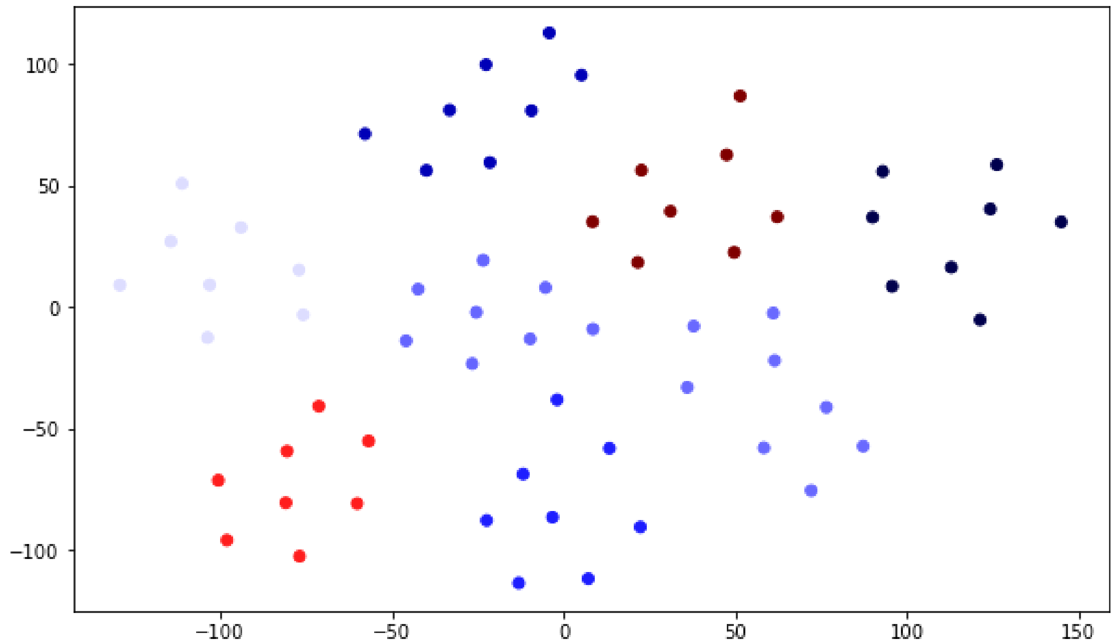
In order to *visualize* high dimensional data we need to make it intelligble for humans. Here we'll attempt to gain an understanding of the separation in our embedding space by using the t-SNE procedure. Here, points in our high dimensional space which are *similar* (i.e. their dot product is high) should be grouped together.

[18]:
```
from sklearn.manifold import TSNE
```

[19]:
```
batch_size = metadata['batch_size']
n_samples = batch_size * 1 # batch_size x n_batches

inputs = np.zeros((n_samples, embedding_length))
targets = np.zeros((n_samples,))

i = 0
while i < n_samples:
    x, y = next(train_it)
    embeddings = embedding_model(x)

    inputs[i:i+batch_size,:] = embeddings
    targets[i:i+batch_size] = y
```

```
    i += batch_size
```

[79]:
```
X_embedded = TSNE(n_components=2).fit_transform(inputs)

plt.figure(figsize=(10,6))
_ = plt.scatter(X_embedded[:, 0], X_embedded[:, 1], c=targets, cmap='seismic')
```



3.1 Cosine similarity

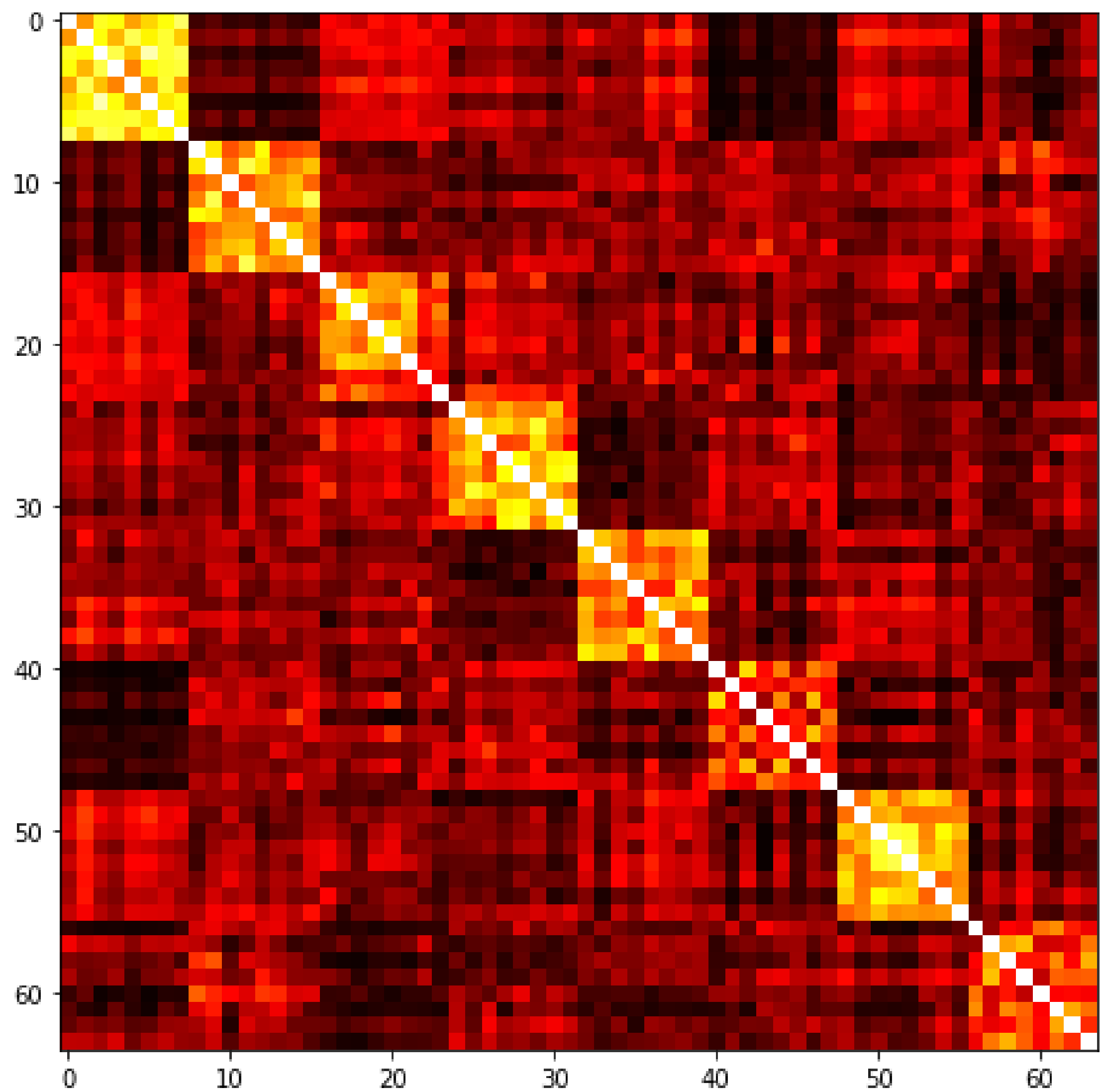
Similarly, we can visualize the similarity of embedding vectors for each utterance in a batch. This should yield a block diagonal matrix where each NxM block along the diagonal (the i^{th} speaker's embeddings) has a higher cosine similarity than the rest of the embedding vectors.

[80]:
```
from sklearn.metrics.pairwise import cosine_similarity
```

[81]:
```
cos = cosine_similarity(embeddings)
cos.shape
```

[81]: (64, 64)

[82]:
```
plt.figure(figsize=(12,8))
_ = plt.imshow(cos, cmap='hot', interpolation='nearest')
```



4 Validation

4.1 Out-of-Sample Dataset

[24]:
```
full_model.evaluate(dataset.get_test_dataset())
```

```
288/288 [==============================] - 54s 188ms/step - loss: 0.1659
```

[24]: 0.16587024927139282

4.2 Equal Error Rate (EER)

In many biometric security systems the equal error rate gives us a performance idea about the model. The equal error ratio depends on two statistics the false acceptance ratio, FAR, and the false rejection ratio, FRR.

FAR is the ratio of falsely accepted scored impostors over the total scored imposters, e.g. the number of utterances which should have been rejected as being close to a true speaker d-vector.

Conversely, the FRR are the utterances which should have been close to a true speaker d-vector, but weren't.

We find the EER when the average of these two is at its minimum for some threshold t.

[25]:
```
eer_inputs, _ = dataset.get_single_test_batch()
eer_preds = full_model(eer_inputs)
eer_pred_norm = tf.math.l2_normalize(eer_preds, axis=0)
```

[28]:
```
threshold, eer, far, frr = equal_error_ratio(eer_pred_norm, 8, 8, 0.0)
threshold, eer
```

[28]:
```
(0.26000000000000006, 0.1328125)
```

[41]:
```
# calculate averages
count, sum_thres, sum_eer = 0, 0.0, 0.0
for x, _ in dataset.get_test_dataset():
    S = full_model(eer_inputs)
    S_norm = tf.math.l2_normalize(S, axis=0)
    threshold, eer, _, _ = equal_error_ratio(S_norm, 8, 8, 0.0)
    sum_thres += threshold
    sum_eer += eer
    count += 1
```

[42]:
```
# avg threshold
sum_thres / count
```

[42]:
```
0.2600000000000002
```

[43]:
```
# avg EER
sum_eer / count
```

[43]:
```
0.1328125
```

5 Enrollment

As previously stated, the generalized approach is intended to demonstrate that our model can *learn* how to separate speakers in a high dimensional embedding space such that we can distinguish utterances spoken by different people. This is useful in a whole bunch of tasks that require biometric security, personalization, etc. Just think of how your Google Home or Amazon Alexa is able to tell *which* person in your household is talking to it. This is how.

So, we'll walk through *enrolling* a new speaker. Here that means passing some of their samples through our model and then averaging out the embedding vector to get their d-vector. This will then serve as the reference point for that person. When we get a **new** utterance, we will check it against that d-vector in order to accept/reject based on some threshold.

[44]:
```
enrollment_inputs, enrollment_targets = dataset.get_single_test_batch()
enrollment_speakers = np.unique(enrollment_targets)
```

[45]:
```
idx = 0
speaker = enrollment_speakers[idx]
speaker
```

[45]: 3558

[46]:
```
orig_id = None
for full_id, mapped_id in dataset.get_metadata()['speaker_id_mapping'].items():
    if mapped_id == speaker:
        orig_id = full_id
orig_id
```

[46]: 'LibriSpeech/test-clean/8463'

[47]:
```
# how many utterances do we have for this person?
indices = [ i for i, s in enumerate(enrollment_targets) if s == speaker]

begin, end = indices[0], indices[len(indices)//2]
len(indices), indices
```

[47]: (8, [40, 41, 42, 43, 44, 45, 46, 47])

[48]:
```
# now create an embedding vector for this person from end-begin=4 utterances␣
↪above
enrollment_embeddings = embedding_model(enrollment_inputs)

d_vector = np.mean(enrollment_embeddings[begin:end], axis=0)
d_vector.shape
```

[48]: (128,)

The metric with which we compare each enrolled d-vector and a new d-vector for an unseen utterance is the cosine similarity, which we explained above.

[49]:
```
# find cosine similarity with same speaker
cosine_similarity(d_vector.reshape(1, -1), enrollment_embeddings[end+2].numpy().
↪reshape(1, -1))
```

[49]: array([[0.74438655]], dtype=float32)

[50]:
```
# with utterance of a different speaker
cosine_similarity(d_vector.reshape(1, -1), enrollment_embeddings[0].numpy().
 ↪reshape(1, -1))
```

[50]: array([[0.14067599]], dtype=float32)