Homework 2 Part 2

Speaker Verification via Convolutional Neural Networks

11-785: Introduction to Deep Learning (Fall 2018)

OUT: September 25th, 2018

DUE: **October 15th, 2018, 11:59 PM**

1 Introduction

Determining whether two speech segments were uttered by the same speaker has several important applications. This problem is known as **speaker verification**.

In this assignment, you will use convolutional neural networks (CNNs) to design a system for speaker verification. End-to-end, your system will be given two utterances as input and will output a score that quantifies the similarity it detects among the *speakers* of these utterances. Ultimately, this allows us to decide whether the utterances belong to the same speaker, in fact, or to different speakers.

To this end, you will train on a dataset with a few thousand hours of speaker-labeled speech (i.e., a set of utterances, each labeled by an ID that uniquely identifies its speaker). In doing so, you will gain experience with the concept of *embeddings* (in this case, embeddings for speaker information), optionally several relevant loss functions, and, of course, convolutional layers as effective shift-invariant feature extractors. Additionally, you will begin to develop skills necessary for processing and training with *big data*, which is often the scale at which deep neural networks demonstrate excellent performance in practice.

2 Speaker Verification

An input to your system is a *trial*, that is, a pair of utterances that may or may not belong to the same speaker. Given a trial, your goal is to output a numeric score that quantifies how similar the speakers of the two utterances appear to be. On some scale, a higher score will indicate higher confidence that the two utterances belong to one and the same speaker in fact.

Below, we provide a brief outline of *how* you might use CNNs to compute such similarity scores effectively.

2.1 Speaker Embeddings

Intuitively, speech utterances contain a lot of acoustic features that vary across different speakers. Your main task is to train a CNN to extract and represent such important features from an utterance. These extracted features will be represented in a *fixed-length* vector of features, known as a **speaker embedding**. Given two speaker embeddings, you will use an appropriate metric (e.g., *cosine similarity* or (negative) *L2 distance* between the computed embeddings) to produce your similarity scores. We recommend you use (or at least begin by using) cosine similarity.

Accordingly, once you have trained your CNN, your end-to-end speaker verification system will use your CNN as follows. Given two utterances, you will pass each utterance through your CNN to generate two speaker embeddings, among which you will compute your similarity score. This score will be the output of your overall system. The next natural question is: how should you train your CNN to produce high-quality speaker embeddings?

2.2 N-way Classification

This is the simplest approach and the one we will (mostly) focus on in this assignment. We recommend you implement *and* tune this before considering more advanced alternatives.

Let our speaker-labeled dataset contain M utterances spoken by N unique speakers (here, $M > N$). Your goal is to use this data to train your CNN so that it produces "good" speaker embeddings. One reasonable way to do so is to optimize these embeddings for predicting the identities of the training speakers from their speech. The resulting embeddings will clearly encode a lot of speaker-varying features, just as desired. This suggests an N-way classification task, where you classify every utterance among N classes (i.e., one for every unique speaker in the training set).

More concretely, your network will consist of several (convolutional) layers for feature extraction. The input will be (possibly a part of) an utterance. The output of the *last* such feature extraction layer is your speaker embedding. You will pass this speaker embedding through a linear layer with dimensions `embedding_dim`×`num_speakers`, followed by softmax, to classify utterances among the N (i.e., `num_speakers`) speakers.

You can then use cross-entropy loss to optimize your network to predict the correct speaker for every training utterance. After the network is trained, you will remove the linear/classification layer. This leaves you with a CNN that computes speaker embeddings given arbitrary speech utterances.

2.3 Alternative Approaches

The main shortcoming of the N-way classification approach is that it does not *directly* optimize the speaker embeddings for the similarity metric used at test time (e.g., cosine similarity). More interesting approaches involve more specialized loss functions. Ordered roughly (and subjectively) by the complexity of implementation, center loss, contrastive loss, triplet loss, and angular loss are all appropriate functions to consider.

2.4 System Evaluation

This subsection briefly describes how the "quality" of your similarity scores will be evaluated. Given similarity scores for many trials, some *threshold* score is needed to actually accept or reject pairs as *same-speaker* pairs (i.e., when the similarity score is above the threshold) or *different-speaker* pairs (i.e., when the score is below the threshold), respectively. For any given threshold, some percentage of the different-speaker pairs will be accepted (known as the *false acceptance* rate) and some percentage of the same-speaker pairs will be rejected (known as the *false rejection* rate).

A natural choice of threshold is one which equates these two proportions. Indeed, your similarity scores will be evaluated using the Equal Error Rate (EER) metric, which is the percentage of pairs at which the false acceptance and false rejection rates are equated. Naturally, you want the EER of your system to be as low as possible.

3 Dataset

In this assignment, you will work with data that is fairly large in size.[1] As you probably know, large training data is often the scale at which deep neural networks demonstrate excellent performance in practice. Nonetheless, it is important to note that the assignment is specifically designed so it is entirely possible to do well while directly handling the full dataset *only* as a final stage. To this end, we have split the training data into several *chunks*, each with the utterances of a unique subset of the training speakers. Accordingly,

[1]Our raw data is around 150 GB in size, but the features you will train and evaluate with are much smaller for several reasons, including the use of 16-bit features and the application of some limited Voice Activity Detection beforehand.

you will develop and debug on a small chunk (less than 2 GB), prototype and tune on a larger subset of chunks (10 GB after preprocessing), and train your final model on the full set (40 GB after preprocessing).

3.1 S3 Instructions

Our training, validation, and test data are hosted on AWS S3. To download, you will need to have the `awscli`[2] command line utility installed. The installation can be as easy as a `pip install awscli` (possibly with `sudo`).

After installing, you will need to run `aws configure`. This command requires an *AWS Access Key ID* and an *AWS Secret Access Key*. You can obtain these under *Secrity Credentials*[3] (specifically, in the box for Access keys). Please note you might need to create a new access key if you don't have one already.

The `aws configure` command will also prompt you for a *Default Region Name*. Please insert `us-east-1`. You can leave the *Default Output Format* option empty.

3.2 File Structure

For convenient development and prototyping, the data is split into three compressed files: `hw2p2_A.tar.gz` (765 MB compressed), `hw2p2_B.tar.gz` (5.8 GB compressed), and `hw2p2_C.tar.gz` (18 GB compressed).

- `hw2p2_A.tar.gz`: This contains a single, small training chunk, namely the file `1.npz` (< 2 GB). You should use `1.npz` for local development and debugging.
  - `aws s3 cp s3://11785fall2018/hw2p2_A.tar.gz .`
- `hw2p2_B.tar.gz`: This contains two larger training chunks, namely `2.npz` (< 4 GB) and `3.npz` (< 8 GB), as well as the validation set `dev.npz`. Once you have verified the correctness of your code, we recommend that you test your model prototypes by training on the three chunks (i.e., `1-3.npz`) while monitoring your EERs using the provided validation set.
  - `aws s3 cp s3://11785fall2018/hw2p2_B.tar.gz .`
- `hw2p2_C.tar.gz`: This contains three larger training chunks, namely `4.npz` (< 8 GB), `5.npz` (< 16 GB), and `6.npz` (< 16 GB) as well as the test set `test.npz`. To save time and money/credits, we recommend that you only train your model on the full data (i.e., `1-6.npz`) after you have both verified the correctness of your code *and* tuned/tested a satisfactory prototype on a smaller subset.
  - `aws s3 cp s3://11785fall2018/hw2p2_C.tar.gz .`

3.3 Training Set

The training data is processed and structured into two arrays, `features` and `speakers`, where `features[i]` contains the i^{th} utterance, spoken by the speaker with ID at `speakers[i]`. An utterance is a (variable-length) array of frames. Each frame has 64 log mel filters, encoded as 16-bit floating-point numbers. There are 100 such frames per second of speech.

The provided starter code will help you load any desired number of training data chunks (e.g., `1.npz`, `2.npz`, and `3.npz` for prototyping). Further, note that segments of silence do exist in the audio and a significant fraction of them were *not* removed in the features we provide you. With the aid of some starter code, you will preprocess the utterances to filter out silent frames. We discuss the starter code in detail in Section 4.

[2]`https://aws.amazon.com/cli/`
[3]`https://console.aws.amazon.com/iam/home#/security_credential`

3.4 Validation and Test Sets

This section applies to both the validation and test setups. In each, you will be provided a list of *enrollment* utterances, a list of *test* utterances, and a list of *trials*. For our purposes, there is no particular difference between enrollment and test utterances; all are just speech segments. A trial is a pair of (`enrollment utterance index`, `test utterance index`). For every trial, you will output a single numerical score that measures their speaker(s) similarity.

For the validation set, the trials provided are labeled. For every trial, the label is `True` *iff* its two utterances belong to the same speaker indeed. Starter code will help you load the validation and test sets and compute your EERs on the validation set.

4 Getting Started

We provide starter code in two files: `preprocess.py` and `utils.py`. Below, we describe how you should (*i*) preprocess the provided features, (*ii*) load subsets of the training data while prototyping and experimenting, (*iii*) train with N-way classification on the variable-length utterances, and (*iv*) evaluate your EERs on the validation set.

4.1 Preprocessing

- `preprocess.py <path> <chunk>`: This is a small script that, by default, applies some basic Voice Activity Detection (VAD) and normalization to the specified `<chunk>` (among training `1-6`, validation `dev`, and test `test` features).

Voice Activity Detection (VAD): As mentioned in Section 3, the utterances contain silence segments. These are clearly not useful for training (or during testing), and hence should be filtered out in a preprocessing stage. This stage, in which we only retain frames that are detected to contain speech, is referred to as Voice Activity Detection (VAD). One simple way to apply VAD is to remove a frame if the maximum filter intensity is below some sensible threshold. In `preprocess.py`, you can find some default code and parameters. Feel free to keep it as-is, to modify the threshold, or to modify the VAD approach itself (e.g., computing the mean instead of maximum across filters). The same applies to the default normalization present in the starter code.

4.2 Loading Training Data

- `from utils import train_load`: The function `train_load(prefix, parts)` allows your to specify which (preprocessed) subsets of the training data you want to use, and loads them from disk. In particular, it stacks the utterances from the specified subsets and conveniently re-numbers the speaker IDs into a dense integer domain (i.e., $\{0,\ 1,\ 2,\ ...,\ N-1\}$ for N speakers).

For early/local development and debugging, it suffices to use the small chunk `1.npz`. For prototyping and tuning, we recommend that you use the chunks `1-3.npz`. We recommend that you only train your model on the full data (i.e., `1-6.npz`) after you have both verified the correctness of your code *and* tested a satisfactory prototype on a smaller subset.

4.3 Training with N-way Classification

At this stage, you have loaded the training data into memory in a simple (and, by now, familiar) format. You have variable-length utterances in array `features` and their corresponding speaker IDs in array `speakers`. In your training loop, you will sample a batch of utterances, run each utterance through your network (i.e., generating an embedding for each, followed by a linear layer to classify among the N classes, followed by lastly optimizing with cross-entropy loss on top of softmax).

As usual, the variation in length across the training samples presents some practical challenges during training. Luckily, we are only interested in the speaker information encoded in an utterance (i.e., and not the contents of the utterance itself). As a result, it is reasonable to decide upon some fixed number of frames (e.g., 5000 frames, which correspond to 50 seconds of speech) *a priori*, then, during training (and testing), randomly trim all utterances in any given batch to this number of frames. If an utterance happens to be too short, you can wrap around back from the beginning.

4.4 Validation with EER

- `from utils import dev_load, test_load`: These functions load the validation and test data.
- `from utils import EER`: The function `EER(labels, scores)` is useful during validation. It takes a list of (gold standard) True/False labels and the estimated similarity scores by your system. It returns the EER and the threshold at which EER occurs.

To track your progress, after an epoch of training, you can compute a similarity score for every trial in the validation set then use the `EER` function provided.

More specifically, recall from Section 3.4 that the validation data consists of features (in two arrays, `enrollment` and `test`) and a list of (labeled) trials. While you can use your CNN to generate embeddings twice per trial, significant pre-computation is possible. In particular, it will be substantially more efficient to simply embed all the utterances in the `enrollment` and `test` arrays *once* at the beginning of a validation phase. Then, when processing each trial, you can simply look up the pre-computed embeddings in order to compute the appropriate similarities (e.g., using the cosine similarity metric).

5 Conclusion

That's all. As always, feel free to ask on Piazza if you have any questions.

Good luck and enjoy the challenge!