CS4420 Programming Project 1 (Points: 10, Due: Sep 18)

Weiran Wang
University of Iowa
weiran-wang@uiowa.edu

Abstract

In this project, you will implement the input generation component of an automatic speech recognition (ASR) system. This project can be done with less than 100 lines of python code.

1 Preliminaries

You shall be familiar with python syntax, and object-oriented programming in general. You shall learn to install packages in python, e.g., with `pip`. The packages you need to install for this project are

- `numpy`: package for manipulating matrices and tensors, a must-have for machine learning.
- `kaldiio`: package for loading precomputed audio features, see [1] for more background information.

2 Input files

I have shared with you `ark` files, as well as `json` files corresponding to the training, development, and test splits of the ASR dataset. Let us go into more details regarding the content of these files.

- Each `json` stores a list of dictionaries, and each dictionary stores the data information of a single speech utterance. Each utterance dictionary contains one key, which is the utterance id, e.g., "103-1240-0000"; the corresponding value is another dictionary containing audio features stored under the `feat` key, and the ground truth transcript of that utterance stored under the `text` key. A snippet of the `json` file is shown in Figure 1. You can load the `json` file in python by "`import json`", and then "`json.load(filename)`".
- You may have notice that the `feat` field has a string value of the form "`train_data/feats.1.ark:14`". Here "`train_data/feats.1.ark`" is the path to an `ark` file under the training data folder, whereas "`14`" is an offset of the utterance's data in that file. After installing the `kaldiio` package, you can use the "`load_mat()`" function to extract the acoustic features into a `numpy` matrix (2D array).

  The acoustic feature sequence is stored in a $T \times d$ matrix, where T is the number of *frames*, and d is the feature dimension. In other words, each row of the matrix corresponds to the d-dimensional feature vector of a frame. Roughly speaking, the features are computed from raw audio as follows. For an audio recording, we collect samples from time windows of 25ms duration (or equivalently 400 samples for a sampling rate of 16KHz), with 10ms shift between two consecutive windows. For each window, we compute the discrete Fourier transform of the samples, and take the log of magnitude of the results to get the frequency representation of that window. Due to the shift of 10ms, a 3 second long audio boils down to 300 windows, and therefore 300 frames of features. In this project, the acoustic features has $d = 83$.

```
{
    "utts": {
        "103-1240-0000": {
            "feat": "train_data/feats.1.ark:14",
            "text": "CHAPTER ONE MISSUS RACHEL LYNDE IS SURPRISED MISSUS RACHEL LYNDE LIVED JUST WHER
E THE AVONLEA MAIN ROAD DIPPED DOWN INTO A LITTLE HOLLOW FRINGED WITH ALDERS AND LADIES EARDROPS AND
TRAVERSED BY A BROOK"
        },
        "103-1240-0001": {
            "feat": "train_data/feats.1.ark:117494",
            "text": "THAT HAD ITS SOURCE AWAY BACK IN THE WOODS OF THE OLD CUTHBERT PLACE IT WAS REPU
TED TO BE AN INTRICATE HEADLONG BROOK IN ITS EARLIER COURSE THROUGH THOSE WOODS WITH DARK SECRETS OF
POOL AND CASCADE BUT BY THE TIME IT REACHED LYNDE'S HOLLOW IT WAS A QUIET WELL CONDUCTED LITTLE STREA
M"
        },
        "103-1240-0002": {
            "feat": "train_data/feats.1.ark:250412",
            "text": "FOR NOT EVEN A BROOK COULD RUN PAST MISSUS RACHEL LYNDE'S DOOR WITHOUT DUE REGAR
D FOR DECENCY AND DECORUM IT PROBABLY WAS CONSCIOUS THAT MISSUS RACHEL WAS SITTING AT HER WINDOW KEEP
ING A SHARP EYE ON EVERYTHING THAT PASSED FROM BROOKS AND CHILDREN UP"
        },
        "103-1240-0003": {
            "feat": "train_data/feats.1.ark:366730",
            "text": "AND THAT IF SHE NOTICED ANYTHING ODD OR OUT OF PLACE SHE WOULD NEVER REST UNTIL
SHE HAD FERRETED OUT THE WHYS AND WHEREFORES THEREOF THERE ARE PLENTY OF PEOPLE IN AVONLEA AND OUT OF
 IT WHO CAN ATTEND CLOSELY TO THEIR NEIGHBOR'S BUSINESS BY DINT OF NEGLECTING THEIR OWN"
        },
        "103-1240-0004": {
            "feat": "train_data/feats.1.ark:489356",
            "text": "BUT MISSUS RACHEL LYNDE WAS ONE OF THOSE CAPABLE CREATURES WHO CAN MANAGE THEIR
OWN CONCERNS AND THOSE OF OTHER FOLKS INTO THE BARGAIN SHE WAS A NOTABLE HOUSEWIFE HER WORK WAS ALWAY
S DONE AND WELL DONE SHE RAN THE SEWING CIRCLE"
        },
```


Figure 1: The beginning part of `train_data.json`.

char	id
A	1
B	2
⋮	⋮
Z	26
’ (apostrophe)	27
␣ (space)	28

Table 1: Character-to-id mapping of the tokenizer.

3 Task I: Tokenizer

The transcript of each utterance is a string. We need to *tokenize* the string into a sequence of discrete *tokens* (integers, or ids) for the machine learning component of ASR.

In this project, we use a simple character-based tokenizer. That is, we split the text string into a list of characters and map each character into a integer. The character-to-id mapping is illustrated in Table 1. Note we have reserved the id 0 which will be used in the future as a special token.

You will implement a new class `CharacterTokenizer` whose constructor stores the character-to-id mapping and its inverse mapping, which maps ids to characters. Additionally, implement two member functions: `StringToIds` takes as input a string and returns a list of ids using the character-to-id mapping, and `IdsToString` takes as input a list of ids and return a string using the id-to-character mapping. The two functions are inverse to each other. In Figure 2 we provide the testing code for this class.

4 Task II: splice and subsample

As mentioned above, the original acoustic features is computed every 10ms (each frame has 10ms of new content). Now we would like to implement two functionalities:

```
>>> tokenizer = input_generator.CharacterTokenizer()
>>> res = tokenizer.StringToIds("HELLO WORLD")
>>> print(res)
[8, 5, 12, 12, 15, 28, 23, 15, 18, 12, 4]
>>>
>>> res = tokenizer.IdsToString(res)
>>> print(res)
HELLO WORLD
```

Figure 2: Testing code for the tokenizer.

- **Splicing**: we would like to add more context to each frame, by concatenating each frame with C frames to the left in time (i.e., previous C frames), and C frames to the right in time (i.e., future C frames). As an example, let the acoustic feature sequence be $[x_0, x_1, \ldots, x_T]$ where $x_i \in R^d$. Then after splicing with $C = 1$, the first output frame is $[x_0; x_0; x_1] \in R^{3d}$ (at the boundary when there are no more left or right frame, we just use the boundary frame itself as context), the second output frame is $[x_0; x_1; x_2] \in R^{3d}$, and so on. The last output frame is $[x_{T-1}; x_T; x_T]$. The total number of frames remains the same. Hint: You could implement this functionality with `numpy.concatenate()`.

- **Subsampling**: after splicing, each new frame has more context and higher feature dimensionality, now we can apply the subsampling trick to make the sequence shorter (and make ASR system more efficient): with a subsampling rate of r, we take one frame for every r frames from the spliced feature sequence. And because each frame has already incorporated context, we do not lose much information with subsampling. Hint: You could implement this functionality using `numpy` array's indexing method `X[::r, :]`.

Write a `splice_and_subsample()` function that takes as inputs the original array, the context length C, and the subsampling rate r. If the input feature matrix has shape $T \times d$, the output of this function has shape $T' \times (2C + 1)d$ where $T' = (T + r - 1)//r$.

5 Task III: Input generator

You will implement a class `InputGenerator` for generating minibatches of data, for training and evaluating the ASR system. Here a minibatch contains the audio features as well as tokenized transcripts (also known as label sequences) for a set of `batch_size` utterances.

Central to the input generator is the notion of *epoch*, which means one pass over the dataset. For training, we typically use `batch_size` > 1 to compute higher quality training signal with more data (e.g., gradient of the loss function). As an example, let the number of training utterances be $N = 198$, and let `batch_size`=4. Then we will produce 50 minibatches in one epoch over the training set. We need to pad two utterances in the last minibatch (can be randomly sampled from the dataset) to ensure the fixed batch size. For evaluation, let us use `batch_size`=1 for simplicity.

Your implementation of `InputGenerator` should provide the following functionalities.

- The constructor takes as input the json file, the `batch_size` specified by user, binary indicator `shuffle` to indicate whether to randomly permute the order of utterances in each epoch (which is a good idea during training), and the context length C and subsampling rate r for feature processing.

- `InputGenerator` implements a `next()` function which returns a list of `batch_size` triplets (3-tuples); each triplet consists of an utterance id, the spliced and subsampled audio features, and the label sequence. You would call `kaldiio.load_mat()` to extract the original acoustic features, followed by `splice_and_subsample` (using C and r), and use a `CharacterTokenizer` to tokenize the transcripts in this function.

- The class should maintain the epoch number, and the number of steps (calls to `next()`) within the epoch. Each call to `next()` advances the step number by 1, and after finishing one epoch, the epoch number increase by 1 while the step within epoch is reset to 0. Also maintain the total number of steps across all epochs.

- If `shuffle=True`, you create at the beginning of each epoch a random permutation of $[0, \ldots, N-1]$ where N is the total number of utterances, and sequentially take chunks of `batch_size` elements from that permutation, as indices to the original list of utterances to create each minibatch. For example, assume that we have $N = 8$ utterances, `batch_size=2`, and the permutation is $[3, 5, 7, 1, 0, 4, 6, 2]$ for some epoch. Then for that epoch, the first call to `next()` returns utterances 3 and 5, the next call returns utterances 7 and 1, and so on.

6 Submission

You should implement the tasks in a file `input_generator.py` and submit that file. Test your implementation by running `input_generator_test.py` and passing all assertions without modifying the file.

References

[1] https://github.com/nttcslab-sp/kaldiio.