DLSpeech - Ex5 046747

January 2025

Guidelines

1. Submit your files using the Moodle system.
2. You are allowed to submit in **pairs**. If you choose to do so, **both students must submit**.
3. In order to submit your solution please submit the following files:
   - `ex_5_part1.py` - Python 3.9+ code file with your implementation for part 1.
   - `ex_5_part2.py` - Python 3.9+ code file with your implementation for part 2.
   - `output.txt` - a text file with your predictions to part 2.
   - `ex_5.pdf` - a report pdf file, described in section 2.4.
   - `kenlm.arpa` - the generated language model. Described in section 2.2.
4. For any questions, open a thread on the Q&A forum on the course's Moodle site.

1 Connectionist Temporal Classification

In this exercise you will implement the CTC loss in Python. CTC calculates the probability of a specific labeling given the model's output distribution over phonemes.

Formally, CTC calculates $P(\mathbf{p}|\mathbf{x})$ where $\mathbf{x} = [x_1, x_2, \ldots, x_T]$ is an input sequence of acoustic features, $\mathbf{p} = [p_1, p_2, \ldots, p_p]$ is a sequence of transcription phonemes, and $\mathbf{y}$ is a sequence of network outputs, that is, y_k^t can be interpreted as the probability of observing label k at time t.

Defining $\mathbf{z} = [\epsilon, p_1, \epsilon, p_2, \epsilon, \ldots, p_{|\mathbf{p}|}, \epsilon]$ as the padded sequence with ϵ as separator, and define $\alpha_{s,t}$ to be the probability of the subsequence $\mathbf{z}_{1:s}$ after t time steps.

We can calculate α using the following initialization:

$$\alpha_{1,1} = y_\epsilon^1$$
$$\alpha_{2,1} = y_{z_1}^1$$
$$\alpha_{s,1} = 0, \forall s > 2$$

and the following dynamic programming:

$$\alpha_{s,t} = \begin{cases} (\alpha_{s-1,t-1} + \alpha_{s,t-1}) \cdot y_{z_s}^t, & z_s = \epsilon \text{ or } z_s = z_{s-2} \\ (\alpha_{s-2,t-1} + \alpha_{s-1,t-1} + \alpha_{s,t-1}) \cdot y_{z_s}^t, & \text{else} \end{cases}$$

1.1 Instructions

In this exercise, assume you are given a sequence of phonemes $\mathbf{p}$ and the network's output $\mathbf{y}$. In words, $\mathbf{y}$ is a matrix with the shape of $T \times K$ where T is the number of time steps, and K is the amount of phonemes (including ϵ). Each column i of $\mathbf{y}$ is a distribution over K phonemes at time i.

Your goal is to implement the CTC function to calculate $P(\mathbf{p}|\mathbf{x})$ using the above equations.

Your code should get 3 arguments:

1. A path to a 2D numpy matrix of network outputs (y). Load the matrix with `numpy.load()`.
2. The labeling you wish to calculate the probability for (e.g., "aaabb" means we want the probability of "aaabb").
3. A string specifying the possible output tokens (e.g., for an alphabet of $[a, b, c]$ the string should be "abc").

Overall, your code should run with the following command:

```
$ python ex_5_part1.py /some/path/to/mat.npy aaabb abc
```

Your code should print the calculated probability to an output file called `out.txt`. Please round your output probability using `round(x,2)` function.

For demonstration, on Fig. [1], an example is attached: calculate the probability of string 'a' from the given matrix, for an alphabet 'ab'. In that case, the possible paths are: (i) $a\epsilon$, (ii) ϵa (iii) aa. So the probability will be:

$$0.4 \cdot 0.6 + 0.6 \cdot 0.4 + 0.4 \cdot 0.4 = 0.64$$

You can try other strings, calculate the probability by hand and see that it matches your score. Just be careful to take not too long sequences. The submit system will also check against the same input/output.

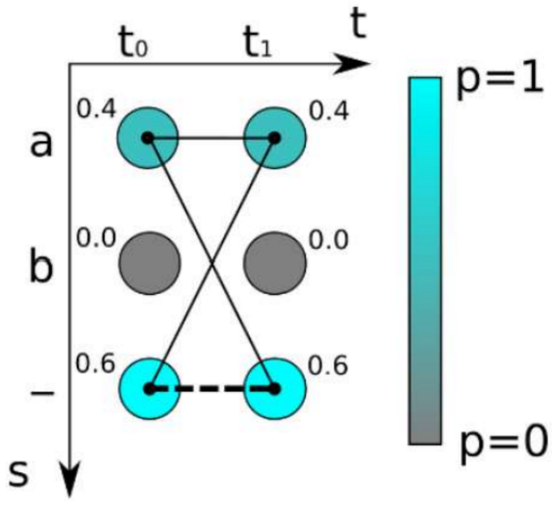

Figure 1: 'a' CTC example

2 Automatic Speech Recognition

In this exercise, you will implement your first ASR system!
The dataset you are given is called `TIDIGITS` where a sequence of **up to 7** digits is pronounced in each recording. For simplicity, this corpus's vocabulary is only composed of digits(0-9).

In class, you have learned about the key components of an ASR system - An acoustic model and a Language model.

2.1 Acoustic Model

For training an acoustic model without an aligned transcription, you should use the `CTC` loss over the **letters** composing the pronunciation of each digit.
In other words :

The digit codes in a filename indicate the digit sequence spoken and can be decoded as follows:

```
z —> zero          3 —> three          7 —> seven
o —> oh            4 —> four           8 —> eight
1 —> one           5 —> five           9 —> nine
2 —> two           6 —> six
```

e.g. a file named '7o19a.wav' is transcribed into: SEVEN OH ONE NINE.

'249z9a.wav' is transcribed into: TWO FOUR NINE ZERO NINE.

'4b.wav' is transcribed into: FOUR.

Note: the last letter in each filename is a—b and can be ignored. The training set contains 12549 recordings produced by men, women, boys, and girls.
Implementation details are up to you - implement as you wish! You are more than welcomed to fine-tune an existing pre-trained models.

2.1.1 Fine-tuning a Pre-trained Acoustic Model

In order to get better results, it is recommended to use a pre-trained acoustic model (e.g. wav2vec2.0).

In this section, you will be guided on how to fine tune a pre trained model on the TDIGITS dataset.

When we want to fine tune a pre trained model, we usually train only a subset of parameters from the original model. Alternatively, we can also add few layers on top of the pre-trained net, in order to make it more modular. This subsection will guide you using the second approach, which is also simple.

Since in the dataset we use, we transcribe only words from a closed set (the digits set), we don't need some characters (like 'a', 'b', 'c'). We can use this property in order to add a simple linear layer that will adapt from the acoustic model output, to our desired output.

The next python pseudo-code is the structure you need in order to train a pre-trained model:

```python
import torch
import torchaudio

device = "cuda" is torch.cuda.is_available() else "cpu"

# wrapper class to our model
class MyAcousticModel(torch.nn.Module):
    def __init__(self):
        bundle = torchaudio.pipelines.WAV2VEC2_ASR_BASE_960H
        self.base_acoustic_model = bundle.get_model().to(device)

        #  freeze base model parameters
        for p in self.base_acoustic_model.parameters():
            p.requires_grad = False

        # You can replace this adapter layer in any net you'd like to.
        # 29 is the output dimension of the specific wav2vec2.0 model.
        self.adapter_layer = torch.nn.Linear(29, num_of_relevant_tokens)

    def forward(self, waveform: torch.Tensor):
        with torch.no_grad():
            emission, _ = self.base_acoustic_model(waveform)

        out = self.adapter_layer(emission)


# training loop
my_model = MyAcousticModel().to(device)

# Make sure that when you run inference on test set, you use model.eval()
# instead, to run inference properly.
my_model.train() # important when training
```

```python
learning_rate = # Your choice
loss = torch.nn.CTCLoss() # The loss you need to use
opt = torch.optim.Adam(my_model.parameters(), lr=learning_rate)

for file in dataset:
    # 1. make sure to load the waveform to device!
    # 2. make sure you are loading waveform using the correct sampling rate,
    # (wav2vec2.0 assumes 16KHz).
    waveform = # load audio, you can use torchaudio.load()

    # from filename you can build a transcript
    transcript = filename_to_transcript(file)

    # As we know, each char is represented as a token - an integer
    # This function tansfers the transcript to a sequence of tokens
    tokens, tokens_lengths = transcript_to_indices(transcript)

    opt.zero_grad()

    # run forward pass, and calc log probabilities.
    out = my_model(waveform).log_softmax(dim=-1)
    out_legnths = torch.tensor(out.shape[1]).to(device)

    # calc loss
    loss = # Use out, tokens, out_lengths and tokens_lengths to calc loss

    # calc gradients
    loss.backward()

    # run backprop using an optimizer
    opt.step()

# loop ended, now let's save our model!
torch.save(my_model, 'my_model.pt')
```

The above code presents the main guidelines you need to fine tune a pre trained model. Note that some functions and variables here are missing, and you need to implement those too.

After training is done, and the model is saved, you can move on to the next section.

2.2 Language Model

The decoding process combines the acoustic model probabilities with a language model to get the most probable transcription.

As seen in class, there are many types of decoders. In this assignment, we will focus on

`Greedy_decoder` and a `CTC_decoder`. Both decoders are presented in this tutorial(click). Note - some features require updating packages. In order to evaluate the decoding components, you will have to decode **three** times for the following configurations.

- Greedy_decoder without a language model.
- CTC_decoder without a language model.
- CTC_decoder with a language model.

Note: You can use the decoders in the tutorial mentioned above.
We recommend building an n-gram LM with KenLM. Attach the generated LM file with the `.arpa` extension to your submission. (optional) You are provided with `lexicon.txt` and `train_transcription.txt` for your convenient.

For each configuration, report the Word Error Rate(WER) and Character Error Rate (CER) for different beam sizes[1, 50,500] (**beam size is only relevant for the ctc_decoder**).
Overall Given your trained acoustic model, you should include the **14** reported values in your report - in a 2-column table (cols are WER and CER). `n_rows` is **7 -** 2 ctc_decoding conf X 3 beam_size options. + 1 greedy decoder.
PyTorch metrics implementations(optional): WER and CER.

2.3 Testing

Once your system is trained, pick the best configuration for you from section 2.2 and generate predictions for the given test set. You should write them to a file named `output.txt` (should also be submitted) in the following format:

```
test_0.wav - 526883z
test_1.wav - 1
test_2.wav - 4o629
test_3.wav - 31
..
test_12546.wav - 28

format : <test_wav_name> - <prediction>
Your output.txt content can be in any order.
```

2.4 Report

You should submit a file called `ex_5.pdf` which includes the following:

- WER and CER for the configurations described in 2.2.
- A description of your acoustic model.
- instructions for running your code.

Good Luck!