DT2119 Lab4: End-to-end Speech Recognition

1 Objective

In this excercise you will build and train a complete end-to-end speech recognizer, and evaluate it's performance. You will work in python using the PyTorch and PyTorch Audio libraries that implement most of the underlying functionality required. Similar functionality is also available in TensorFlow.

The recognizer you will build is a version of the Deep Speech 2[1].

2 Task

- Prepare functions for data loading, pre-processing and data augmentation.
- Decode the network output into text using a greedy algorithm.
- Measure character- and word error rates.
- Train and evaluate the recognizer.
- Compare results without and with an N-gram langage model, and tune it's parameters.

The model you will train takes spectrogram frames as input, and produces character propabilities as output, which are then decoded into text. It consists of a single neural network that is trained using Connectionist Temporal Classification (CTC) loss. The CTC loss function is key to training the network without requiring alinged labels (as in lab 3). Instead, it only needs examples consisting of (speech, text) pairs.

In order to pass the lab, you will need to follow the steps described in this document, and present your results to a teaching assistant.

Parts of the code, including the model and the training loop, is provided, but you will need to implement certain functions yourself in order to train the model and carry out the experiments. In the file `lab4_main.py` you will find the skeleton code, and in lab4_proto.py there are placeholders for the functions you need to write.

Use Canvas to book a time slot for the presentation. Remember that the goal is not to show your code, but rather to show that you have understood all the steps.

[1] Amodei, D., Ananthanarayanan, S., Anubhai, R., Bai, J., Battenberg, E., Case, C., ... & Zhu, Z. (2016, June). Deep speech 2: End-to-end speech recognition in english and mandarin. In International conference on machine learning (pp. 173-182). PMLR.

3 Data

You will work with the LibriSpeech dataset, which is a large English dataset widely used in ASR training, mainly derived from audiobooks. The full LibriSpeech corpus contains about 1000 hours of speeh, here you will use a subset of 100 hours for training. You will use the *clean* speech set (the corpus also contains an *other* set which is noisy more challenging speech).

Conveniently, LibriSpeech is available through `torchaudio.datasets`, meaning that the following lines is all you need to get all the required data into your project:

```
import torchaudio
train_dataset = torchaudio.datasets.LIBRISPEECH('.',
                url='train-clean-100', download=True)
val_dataset = torchaudio.datasets.LIBRISPEECH('.',
                url='dev-clean', download=True)
test_dataset = torchaudio.datasets.LIBRISPEECH('.',
                url='test-clean', download=True)
```

Besides the 100 hours of training data, there is also a validation set and a test set. The validation set is used after each epoch, and to tune hyper parameters, while the test set is used for final evaluation.

If you wish, you can run the above lines already now to start downloading the data (it will take some time).

The data contains pairs of waveforms and text transcriptions. Your first task will be to write functions required to turn this data into a format the model can work with.

3.1 Representing text

In the network, we'll represent characters as integer labels. We'll need to represent the letters of the englsih alphabet (`a-z`), space (`_`), and apostrophe (`'`). We will only use lowercase letters.

Write two functions, `text_to_int(text)` and `int_to_text(labels)`, that turns strings into lists of integers and vice versa, according to the following mapping:

code	char
0	'
1	_
2	a
3	b
...	
27	z

3.2 Representation and augmentation of the speech signal

3.2.1 Audio transforms

To the model, the speech signal will be represented as a mel-spectrogram (i.e. mel filter bank outputs). You will use `torchaudio.transforms.MelSpectrogram()` to do the converesion of waveforms into mel spectrograms with $nmels = 80$. To help the model to generalize, one can use data augmentations, i.e. aritificially and modifying the inputs in some stochastic way to increase diversity. There are many possible audio augmentations that can work, such as adding

noise, shifting/stretching in time and frequency etc. Here we suggest a simple yet efficient form of augmentation, namely spectrogram masking. This means simply blanking out certain parts of the spectrogram at random, across time or frequency. You can use the transforms

```
torchaudio.transforms.FrequencyMasking()
torchaudio.transforms.TimeMasking()
```

to perform the augmentations. Suggested values for *freq_ mask_ param* and *time_ mask_ param* are 15 and 35 respectively.

The transform functions return a *transform object* which can be used subsequently in the data processing pipeline. Use `torch.nn.Sequential()` to combine several transforms (e.g. spectrogram and augmentations) into one a single transform object.

Important: the data augmentation should only be applied during training, not at test time. Therefore, your data processing function needs to apply different transforms depending on wether it is train or test.

Define two audio transforms such that

`spec = train_audio_transform(wave)`

`spec = test_audio_transform(wave)`

will convert waveforms tensors to spectrogram with and without augmentations, as described above.

3.2.2 Data processing

Pytorch provides a `DataLoader` class that handles a lot of the complexities of training. You just need to provide a data procsessing function that takes the data and performs the required transformations so it can be fed to the network.

Write a function `dataProcessing(data, audio_transform)` that takes one *batch* of data and transforms it accordingly. `data` is a list of tuples, representing one batch. Each tuple is of the form `(waveform, sample_rate, utterance, speaker_id, chapter_id, utterance_id)`. You only need to care about waveform and utterance, the rest can be safely ignored (for Librispeech, sample rate is always 16000 Hz). `waveform` will be a 1D tensor, and `utterance` will be a string.

The `audio_transform` argument is the transformation object to apply to the waveform. By giving it as a parameter, it is possible to use the same data processing function for train and test.

The spectrogram transforms will output a 3D tensor `(channel, n_mels, time)`. In our case, audio is mono (num channles=1) and we want time first `(time,n_mels)` for the padding to work (see below). The following operation can be used to rearrange the spectrogram tensor accordingly:

`spec = spec.squeeze(0).transpose(0, 1)`

The `dataProcessing()` function will loop over the tuples in the input, transform each waveform according to the `audio_transform` and turn the stings into integer label sequences according to the text mapping functions above. For efficiency reasons, PyTorch needs the input and output for each batch represented as tensors. Because the inputs and outputs will be of varying lengths, they will have to be padded to the length of the longest item, to fit in a tensor where batch is the first dimension. Use `nn.utils.rnn.pad_sequence(..., batch_first=True)` to do this. Finally, the stack of padded spectrograms needs to be rearranged from `(batch,time,mels)` to `(batch,channel,mels,time)` which is the order expected by the model.

`x.unsqueeze(1).transpose(2, 3)`

may be used for this operation (take a minute to read about the `squeeze()/unsqueeze()` operations if you are unfamilliar with them).

The `dataProcessing()` function should return a tuple of four elements: `(spectrograms, labels, input_lengths, label_lengths)` - the last two represent the lengths before padding, and are used in the loss function for masking.

- `spectrograms` is a tensor of shape `B x C x M x T` where B is batch, C is channels (=1), T is time (frames) and M is mel bands (=80).
- `labels` is a tensor of shape `B x L` where L is label length.
- `input_lengths` list of integers $L_i = T_i/2$ where T_i is the number of frames in spectrogram i (before padding)[2].
- `label_lengths` list of integers corresponding to the lengths of the label strings.

3.2.3 Example data

To help you verify that your data processing function does the right thing, a small example batch of size 5 including input and output of the function is provided. To load it:

```
import torch
example = torch.load('lab4_example.pt')
```

Here, `example` is a dictionary containing both the input `data` (list of tuples), as well as the four output fields `spectrograms`, `labels`, `input_lengths` and `label_lengths`.

4 The model

In this lab you are given the network architecture. The main model is defined in the class `SpeechRecognitionModel`. It takes spectrograms as input, and produces *character probabilities per time step* as output. The model is a variant of Deep Speech 2 network, and it uses a stack of 2D convolutional layers, very similar to many image processing networks. The convolutions operate on the MelSpectrograms just as they would on images; by extracting feature maps that for each layer becomes more specialized to the task. This network uses *ResNet* layers, which means that there are skip-connections allowing a path for the data to pass directly to the next layer. Look at `ResidualCNN` and investigate how this is implemented.

After the convolutions, follows a stack of recursive layers. These are *bidirectional GRU-layers*, (defined in the `BidirectionalGRU` class). Since they are bidirectional ,they can model time dependencies both forwards and backwards in time, which is beneficial to the task of speech recognition, but the forward time dependency limits the model to only work in non-casual scenarios when the entire input is known up front.

The output of the last recursive layer feeds into a classifier part. This is a fully connected layer with 29 output classes, followed by a `LogSoftMax` layer. The number 29 equals the size of the character set (28), plus the blank token used by the CTC algorithm (see below).

The file `lab4_main.py` contains a dict called `hparams`. Here, different hyper parameters of the model and training procedure can be set. Feel free to experiment with these as you see fit, or keep them as they are. You might need to change the batch size to accommodate your hardware (amount of GPU memory).

[2] `input_lengths` is used in the loss function, which operates on the output side of the network. The network reduces the time dimension by a factor of 2, i.e. i.e. each output frame corresponds to two spectrogram time slices, which is why the number of spectrogram frames is divided by two in the result

5 Training and testing

5.1 CTC loss

A critical part in any DNN training is the choice of loss function, i.e. how to calculate the error with respect to the labels Y, for each example X that is fed through the network. Given the nature of the task where we don't know the temporal alignment between inputs and outputs, we need a loss function that can deal with this lack of alignment. The CTC loss does just this. The output of the network will be a matrix containing the probabilities for each character at each step in time, and the CTC loss function uses dynamic programming to evaluate the probability of the output sequence Y given X. The training criterion for the network will be to minimize $-\log p(Y|X)$.

Most of the time one character is spread across many time steps, and CTC works by merging consecutive repititions of the same character into one `aaaabbbaaa -> aba`. This would however lead to any double spelling being lost `helloo -> helo`. To prevent this, CTC introduces a special character called the *blank token*, often denoted ϵ, that prevents characters on either side from being merged.

See `distill.pub/2017/ctc/` for a comprehensive overview of the CTC algorithm.

You will use the PyTorch implementation of CTC loss. Read the documentation for

`torch.nn.CTCLoss()`

and make sure you understand how it is used.

5.2 First epoch

Now you should have everything you need to start *training* the network. The provided main file `lab4_main.py` defines the network as well as the main training loop. The function `train()` will train the model for one epoch (one pass through all the training data). Start training and let it run through one epoch to verify that everything works as expected:

```
python lab4_main.py --mode train
```

If all goes well, the network should be instantiated and the training loop should begin iterating throgh the training data in batches. After each batch, it will print how much of the epoch is completed, and the current loss on the latest batch.

If you get out-of-memory errors, try reducing the batch size.

At the end of the first epoch, the main loop will call the `test()` function. This function follows the structure of `train()` but on the validation set, and without the backward pass. Additionally, it will also decode the output into a string and calculate the accuracy.

5.3 Decoding and accuracy

The output of the network is a matrix that for each time step contains the probability for each of the characters in the alphabet. In order to turn this into text, we need to *decode* the output. Write a function

```
greedyDecoder(output, blank_label=28)
```

that takes in the network output tensor `output` of shape `(batch,time,character)` and the id of the CTC *blank token*, and decodes the string in a greedy fashion, which implies that each time step gets assigned the character with the highest probability at that step. Thus, the decoder needs to:

- extract a sequence containing the *most probable* character for each time step
- merge any adjacent *identical characters*

- get rid of the *blank tokens*

The function should then return a list of decoded strings of the batch.

The `test()` function will also calculate the accuracy in terms of charachter error rate, CER, and word error rate, WER. Both of these are defined in terms of the *edit distance* between the reference transcription string and the decoded output. Specifically, we will use the *Levensthein distance*, which corresponds to the number of inertions, deletions and substitiontions required to go from one string to the other. It can be calculated using dynamic programming.

Implement the function `levenshteinDistance(sequence1,sequence2)` according to algorithm 1.

With this function in place, both the provided `wer()` and `cer()` should work. Try them on a few simple strings and verify that the results are correct before you continue.

Input : Two sequences *sequence*1 and *sequence*2
Output: Levenshtein distance between *sequence*1 and *sequence*2
Initialize a matrix *matrix* with dimensions $(\text{len}(sequence1)+1) \times (\text{len}(sequence2)+1)$;
Fill the first row of *matrix* with values ranging from 0 to len(*sequence*2);
Fill the first column of *matrix* with values ranging from 0 to len(*sequence*1);
for *i from 1 to len*(*sequence*1) **do**
  for *j from 1 to len*(*sequence*2) **do**
    if $sequence1[i-1] = sequence2[j-1]$ **then**
      $matrix[i][j] = matrix[i-1][j-1]$;
    else

$$matrix[i][j] = \min \begin{pmatrix} matrix[i-1][j]+1, \\ matrix[i][j-1]+1, \\ matrix[i-1][j-1]+1 \end{pmatrix};$$

    end
  end
end
Levenshtein distance = $matrix[\text{len}(sequence1)][\text{len}(sequence2)]$;

Algorithm 1: Levenshtein distance calculation using iterative full matrix method

5.4 Train the model and check the results

Now everything is ready to start training for real. The script is by default set up to save the network to a new file after each epoch. This can take up a lot of space. If you wish, you can change the code so it saves it to the same file, but only when it gets a better WER or CER than before.

Run the training for at least 10 epochs, but more if you can. This will take seveal hours to complete on a GPU system.

If you wish, you can add monitoring of losses, accuracy and more via `torch.utils.tensorboard`.

If you need to stop training and continue later, you can load a previously saved model (checkpoint) using the `--model` option to the main script.

The same option can also be used if you want to run a trained model on new data, by specifying `--mode recognize` and giving one or more `.wav`-files on the command line.

When you are done training, try your model on a few different speech files: download from internet, or record your own! How does it perform?

5.5 Language model

You will notice that the model yields far from perfect results. Since it works on character level, it often produces strange spellings and funny words. This is where a language model can help a lot.

The other good news is that adding a language model will not require a re-training of the model (*phew!*). Rather, the language model is integrated at the decoding stage. This is known as shallow fusion[3]. You are provided with a pre-trained N-gram language model, `wiki-interpolate.3gram.arpa`. It is trained on about 2 million words of the *wikitext-2* corpus.

You should now replace your `greedyDecoder()` function with a decoder that incorporates a language model. You will use the `pyctcdecode` package[4]. It's quite straight forward - first you instantiate the decoder:

```
decoder = build_ctcdecoder(
    labels,
    kenlm_model_path=model,
    alpha=0.5,
    beta=1.0
)
```

where `labels` is a list of the characters. Then, the following line can be used to decode the first utterance in the batch: `text = decoder.decode(output[0].cpu().detach().numpy())`[5].

Run your model with and without language model and compare the results, both subjectively, and by calculating the CER and WER over the test set. What observations can you make?

5.5.1 Tuning the language model weights

You will notice two parameters, `alpha` and `beta` above ($0 \leq \alpha, \beta \leq 1$). They control the relative influence of the language model probabilities and the word insertion penalty, respectively. For optimal results they should be tuned on the validation set.

Implement a grid search that steps through different combinations of α and β, in order to find the values that yield the lowest WER on the validation set!

5.6 Summing up

Finally, evaluate the model with the tuned parameters on the test set, and compare the results against the version witout a language model, and discuss your results!

How do you think the accuracy of the model could be improved further? Can you propose any changes to he model?

[3]It's called *shallow* fusion because it fuses the ASR and language models at the final output level, without integrating them deeply into a single model. This is in contrast to *deep* fusion methods, which attempt to integrate the language model more thoroughly into the ASR model.

[4]https://github.com/kensho-technologies/pyctcdecode

[5]the part `.cpu().detach().numpy()` turns a tensor in GPU memory into a numpy array in CPU memory