DT2119 Lab3: Phoneme Recognition with Deep Neural Networks

1 Objectives

After this exercise you should be able to:

- create phonetic annotations of speech recordings using predefined phonetic models
- use software libraries[1] to define and train Deep Neural Networks (DNNs) for phoneme recognition
- explain the difference between HMM and DNN training
- compare which speech features are more suitable for each model and explain why

You will be able to use credits for the Google Cloud Platform for GPU resources, see instructions in Canvas.

2 Task

Train and test a phone recogniser based on digit speech material from the TIDIGIT database:

- using predefined Gaussian-emission HMM phonetic models, create time aligned phonetic transcriptions of the TIDIGITS database,
- define appropriate DNN models for phoneme recognition using PyTorch,
- train and evaluate the DNN models on a frame-by-frame recognition score,
- repeat the training by varying model parameters and input features

Optional:

- perform and evaluate continuous speech recognition at the phoneme and word level using Gaussian-emission HMM models
- perform and evaluate continuous speech recognition at the phoneme and word level using DNN-HMM models

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. 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.

Most of the lab can be performed on any machine running python. The Deep Neural Network training is best performed on a GPU, for example using the Google Cloud Platform, check instructions on Canvas for the GCP.

[1] In this implementation you will use PyTorch `https://pytorch.org/` (recommended) but you are also free to use TensorFlow `https://www.tensorflow.org/`

3 Data

The speech data used in this lab is from the full TIDIGIT database (rather than a small subset as in Lab 1 and Lab 2). The database is stored on the AFS cell `kth.se` at the following path:

`/afs/kth.se/misc/csc/dept/tmh/corpora/tidigits`

If you have continuous access to AFS during the lab, for example if you use a CSC Ubuntu machine, create a symbolic link in the lab directory with the command:

`ln -s /afs/kth.se/misc/csc/dept/tmh/corpora/tidigits`

Otherwise, copy the data into a directory called `tidigits` in the lab directory, but be aware of the fact that the database is covered by copyright[2].

The data is divided into disks. The training data is under:

`tidigits/disc_4.1.1/tidigits/train/`

whereas the test data is under:

`tidigits/disc_4.2.1/tidigits/test/`

The next level of hierarchy in the directory tree determines the gender of the speakers (`man`, `woman`). The next level determines the unique two letter speaker identifier (`ae`, `aw`, ...). Finally, under the speaker specific directories you find all the wave files in NIST SPHERE file format. The file name contains information about the spoken digits. For example, the file `52o82a.wav` contains the utterance "five two oh eight two". The last character in the file name represents repetitions (`a` is the first repetition and `b` the second). Every isolated digit is repeated twice, whereas the sequences of digits are only repeated once.

To simplify parsing this information, the `path2info` function in `lab3_tools.py` is provided that accepts a path name as input and returns gender, speaker id, sequence of digits, and repetition, for example:

```
>>> path2info('tidigits/disc_4.1.1/tidigits/train/man/ae/z9z6531a.wav')
('man', 'ae', 'z9z6531', 'a')
```

In `lab3_tools.py` you also find the function `loadAudio` that takes an input path and returns speech samples and sampling rate, for example:

```
>>> loadAudio('tidigits/disc_4.1.1/tidigits/train/man/ae/z9z6531a.wav')
(array([ 10.99966431,  12.99960327,  ...,   8.99972534]), 20000)
```

The function relies on the package `soundfile` that can be installed in python from standard repositories. If you want to know the details and motivation for this function, please refer the documentation in `lab3_tools.py`.

4 Preparing the Data for DNN Training

4.1 Target Class Definition

In this exercise you will use the emitting states in the `phoneHMMs` models from Lab 2 as target classes for the deep neural networks. It is beneficial to create a list of unique states for reference, to make sure that the output of the DNNs always refer to the right HMM state. You can do this with the following commands:

[2]See `https://catalog.ldc.upenn.edu/LDC93S10` for more information.

```
>>> phoneHMMs = np.load('lab2_models_all.npz')['phoneHMMs'].item()
>>> phones = sorted(phoneHMMs.keys())
>>> nstates = {phone: phoneHMMs[phone]['means'].shape[0] for phone in phones}
>>> stateList = [ph + '_' + str(id) for ph in phones for id in range(nstates[ph])]
>>> stateList
['ah_0', 'ah_1', 'ah_2', 'ao_0', 'ao_1', 'ao_2', 'ay_0', 'ay_1', 'ay_2', ...,
 ..., 'w_0', 'w_1', 'w_2', 'z_0', 'z_1', 'z_2']
```

If you want to recover the numerical index of a particular state in the list, you can do for example:

```
>>> stateList.index('ay_2')
8
```

It might be a good idea to save this list in a file, to make sure you always use the same order for the states.

4.2 Forced Alignment

In order to train and test Deep Neural Networks, you will need time aligned transcriptions of the data. In other words, you will need to know the right target class for every time step or feature vector. The Gaussian-emission HMM models in `phoneHMMs` can be used to align the states to each utterance by means of *forced alignment*. To do this, you will build a combined HMM concatenating the models for all the phones in the utterance, and then you will run the Viterbi decoder to recover the best path through this model.

In this section we will do this for a specific file as an example. You can find the intermediate steps in the `lab3_example.npz` file. In the next section you will repeat this process for the whole database. First read the audio and compute liftered MFCC features as you did in Lab 1:

```
>>> filename = 'tidigits/disc_4.1.1/tidigits/train/man/nw/z43a.wav'
>>> samples, samplingrate = loadAudio(filename)
>>> lmfcc = mfcc(samples)
```

Now, use the file name, and possibly the `path2info` function described in Section 3, to recover the sequence of digits (word level transcription) in the file. For example:

```
>>> wordTrans = list(path2info(filename)[2])
>>> wordTrans
['z', '4', '3']
```

The file `z43a.wav` contains, as expected, the digits "zero four three". Write the `words2phones` function in `lab3_proto.py` that, given a word level transcription and the pronunciation dictionary (`prondict` from Lab 2), returns a phone level transcription, including initial and final silence and short pause models after each word. For example:

```
>>> from prondict import prondict
>>> phoneTrans = words2phones(wordTrans, prondict)
>>> phoneTrans
['sil', 'z', 'iy', 'r', 'ow', 'sp', 'f', 'ao', 'r', 'sp', 'th', 'r', 'iy', 'sp', 'sil']
```

Now, use the `concatHMMs` function you implemented in Lab 2 to create a combined model for this specific utterance:

```
>>> utteranceHMM = concatHMMs(phoneHMMs, phoneTrans)
```

Note that the *short pause* model `phoneHMMs['sp']` is different from the other HMM models. It has a single emitting state and can be skipped in case there is no silence.

We also need to be able to map the states in `utteranceHMM` into the unique state names in `stateList`, and, in turns, into the unique state indexes by `stateList.index()`. In order to do this for this particular utterance, you can run:

```
>>> stateTrans = [phone + '_' + str(stateid) for phone in phoneTrans
                  for stateid in range(nstates[phone])]
```

This array gives you, for each state in `utteranceHMM`, the corresponding unique state identifier, for example:

```
>>> stateTrans[10]
'r_1'
```

Use the `log_multivariate_normal_density_diag` and the `viterbi` function you implemented in Lab 2 to align the states in the `utteranceHMM` model to the sequence of feature vectors in `lmfcc`. Use `stateTrans` to convert the sequence of Viterbi states (corresponding to the `utteranceHMM` model) to the unique state names in `stateList`.

At this point it would be good to check your alignment. You can use an external program such as `wavesurfer`[3] to visualise the speech file and the transcription. The `frames2trans` function in `lab3_tools.py`, can be used to convert the *frame-by-frame* sequence of symbols into a transcription in standard format (start time, end time, symbol...). For example, assuming you saved the sequence of symbols you got from the Viterbi path into `viterbiStateTrans`, you can run:

```
>>> frames2trans(viterbiStateTrans, outfilename='z43a.lab')
```

which will save the transcription to the `z43a.lab` file. If you try with other files, save the transcription with the same name as the `wav` file, but with `lab` extension. Then open the `wav` file with `wavesurfer`. Unfortunately, `wavesurfer` does not not recognise the NIST file format automatically. You will get a window to choose the parameters of the file. Choose 20000 for "Sampling Rate", and 1024 for "Read Offset (bytes)". When asked to choose a configuration, choose "Transcription". Your transcription should be loaded automatically, if you saved it with the right file name. Select the speech corresponding to the phonemes that make up a digit, and listen to the sound. Is the alignment correct? What can you say observing the alignment between the sound file and the classes?

4.3 Feature Extraction

Once you are satisfied with your forced aligned transcriptions, extract features and targets for the whole database. To save memory, convert the targets to indices with `stateList.index()`. You should extract both the liftered MFCC features that are used with the Gaussian-emission HMMs and the DNNs, and the filterbank features (`mspec` in Lab 1) that are used for the DNNs. One way of traversing the files in the database is:

```
>>> import os
>>> traindata = []
>>> for root, dirs, files in os.walk('tidigits/disc_4.1.1/tidigits/train'):
>>>     for file in files:
```

[3] `https://sourceforge.net/projects/wavesurfer/`

```
>>>         if file.endswith('.wav'):
>>>             filename = os.path.join(root, file)
>>>             samples, samplingrate = loadAudio(filename)
>>>             ...your code for feature extraction and forced alignment
>>>             traindata.append({'filename': filename, 'lmfcc': lmfcc,
                                  'mspec': 'mspec', 'targets': targets})
```

Extracting features and computing forced alignment for the full training set took around 10 minutes and 270 megabytes on a computer with 8 Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz. You probably want to save the data to file to avoid computing it again. For example with:

```
>>> np.savez('traindata.npz', traindata=traindata)
```

Do the same with the test set files at `tidigits/disc_4.2.1/tidigits/test`

4.4 Training and Validation Sets

Split the training data into a training set (roughly 90%) and validation set (remaining 10%). Make sure that there is a similar distribution of men and women in both sets, and that each speaker is only included in one of the two sets. The last requirement is to ensure that we do not get artificially good results on the validation set. Explain how you selected the two data sets.

4.5 Acoustic Context (Dynamic Features)

It is often beneficial to include some indication of the time evolution of the feature vectors as input to the models. In GMM-HMMs this is usually done by computing first and second order derivatives of the features. In DNN modelling it is more common to stack several consecutive feature vectors together.

For each utterance and time step, stack 7 MFCC or filterbank features symmetrically distributed around the current time step. That is, at time n, stack the features at times $[n-3, n-2, n-1, n, n+1, n+2, n+3]$). At the beginning and end of each utterance, use mirrored feature vectors in place of the missing vectors. For example at the beginning use feature vectors with indexes $[3, 2, 1, 0, 1, 2, 3]$ for the first time step, $[2, 1, 0, 1, 2, 3, 4]$ for the second time step, and so on. The "boundary effect" is usually not very important because each utterance begins and ends with silence.

4.6 Feature Standardisation

Normalise the features over the training set so that each feature coefficient has zero mean and unit variance. This process is called "standardisation". In speech there are at least three ways of doing this:

1. normalise over the whole training set,
2. normalise over each speaker separately, or
3. normalise each utterance individually.

Think about the implications of these different strategies. In the third case, what will happen with the very short utterances in the files containing isolated digits?

You can use the `StandardScaler` from `sklearn.preprocessing` in order to achieve this. In case you normalise over the whole training set, save the normalisation coefficients and reuse them

to normalise the validation and test set. In this case, it is also easier to perform the following step *before* standardisation.

Once the features are standardised, for each of the training, validation and test sets, flatten the data structures, that is, concatenate all the feature matrices so that you obtain a single matrix per set that is $N \times D$, where D is the dimension of the features and N is the total number of frames in each of the sets. Do the same with the targets, making sure you concatenate them in the same order. To clarify, you should create the following arrays $N \times D$ (the dimensions vary slightly depending on how you split the training data into train and validation set), where in parentheses you have the dynamic version of the features:

Name	Content	set	N	D
`(d)lmfcc_train_x`	MFCC features	train	~ 1356000	13 (91)
`(d)lmfcc_val_x`	MFCC features	validation	~ 150000	13 (91)
`(d)lmfcc_test_x`	MFCC features	test	1527014	13 (91)
`(d)mspec_train_x`	Filterbank features	train	~ 1356000	40 (280)
`(d)mspec_val_x`	Filterbank features	validation	~ 150000	40 (280)
`(d)mspec_test_x`	Filterbank features	test	1527014	40 (280)
`train_y`	targets	train	~ 1356000	1
`val_y`	targets	validation	~ 150000	1
`test_y`	targets	test	1527014	1

You will also need to convert feature arrays to 32 bits floating point format because of the hardware limitation in most GPUs, for example:

```
>>> lmfcc_train_x = lmfcc_train_x.astype('float32')
```

and the target arrays into a one-hot encoding, for example:

```
>>> import torch.nn.functional as F
>>> output_dim = len(stateList)
>>> train_y = F.one_hot(torch.tensor(train_y),num_classes=output_dim)
```

5 Phoneme Recognition with Deep Neural Networks

With the help of PyTorch [4] , define a deep neural network that will classify every single feature vector into one of the states in `stateList`, defined in Section 4. Refer to the PyTorch documentation to learn the details of defining and training models and layers. In the following instructions we only give hints to the classes and methods to use for every step.

- Note that PyTorch can run both on CPUs and GPUs. Because it will be faster on a fast GPU it is advised to run large training sessions on Google Colab or on the Google Cloud Platform.
- It is strongly advised to test a simpler version of the models on your own computer first to avoid bugs in your code.
- If you get out-of-memory errors on the GPU, try *reducing the batch size*.
- If for some reason you do not manage to run on GPUs, you can still perform the lab, running simpler models on your own computer. The goal of the lab is not to achieve

[4] `https://pytorch.org/`

state-of-the-art performance, but to be able to compare different aspects of modelling, feature extraction, and optimisation.

Define the model in PyTorch by creating a subclass of `torch.nn.Module` (see the pytoch documentation for examples) to define each layer in the model.

Define the proper size for the input and output layers depending on your feature vectors and number of states. Choose the appropriate activation function for the output layer, given that you want to perform classification. Be prepared to explain why you chose the specific activation and what alternatives there are. For the intermediate layers you can choose, for example, between `relu` and `sigmoid` activation functions.

Decide the kind of loss criterion most appropriate for classification. You can also decide what optimizer to use - here you can choose for example between Stochastic Gradient Descent (`torch.optim.SGD()`) or the Adam optimiser (`torch.optim.Adam()`). Each has a set of parameters to tune. You can use the default values for this exercise, unless you have a reason to do otherwise.

Use the training loop provided in the example code `pytorch_example.py`. Try to understand and be prepared to explain what happens inside the loop. What is the purpose of the validation data? One of the important parameters is the batch size. A typical value is 256, but you can experiment with this to see if convergence becomes faster or slower. If you run out of GPU memory, try reducing the batch size.

Here are the minimum list of configurations to test, but you can test your favourite models if you manage to run the training in reasonable time. Also, depending of the speed of your hardware you can reduce the size of the layers, and skip the models with 2 and 3 hidden layers:

1. input: liftered MFCCs, one to four hidden layers of size 256, rectified linear units
2. input: filterbank features, one to four hidden layers of size 256, rectified linear units
3. same as 1. but with dynamic features as explained in Section 4.5
4. same as 2. but with dynamic features as explained in Section 4.5

Note the evolution of the loss function and the accuracy of the model for every epoch. What can you say comparing the results on the training and validation data?

There are many other parameters that you can vary, if you have time to play with the models. For example:

- different activation functions than ReLU
- different number of hidden layers
- different number of nodes per layer
- different length of context input window
- strategy to update learning rate and momentum
- initialisation with DBNs instead of random
- different normalisation of the feature vectors

If you have time, chose a parameter to test.

5.1 Detailed Evaluation

After experimenting with different models in the previous section, select one or two models to test properly. Evaluate the output of the network given the test frames in `FEATKIND_test_x`. Plot the posteriors for each class for an example utterance and compare them to the target values. What properties can you observe?

For all the test material, evaluate the classification performance from the DNN in the following ways:

1. *frame-by-frame at the state level*: count the number of frames (time steps) that were correctly classified over the total
2. *frame-by-frame at the phoneme level*: same as 1., but merge all states that correspond to the same phoneme, for example `ox_0`, `ox_1` and `ox_2` are merged to `ox`
3. *edit distance at the state level*: convert the *frame-by-frame* sequence of classifications into a transcription by merging all the consequent identical states, for example `ox_0 ox_0 ox_0 ox_1 ox_1 ox_2 ox_2 ox_2 ox_2...` becomes `ox_0 ox_1 ox_2 ...`. Then measure the Phone Error Rate (PER), that is the length normalised edit distance between the sequence of states from the DNN and the correct transcription (that has also been converted this way).
4. *edit distance at the phoneme level*: same as 3. but merging the states into phonemes as in 2.

For the first two types of evaluations, besides the global scores, compute also confusion matrices.

5.2 Possible questions

- what is the influence of feature kind and size of input context window?
- what is the purpose of normalising (standardising) the input feature vectors depending on the activation functions in the network?
- what is the influence of the number of units per layer and the number of layers?
- what is the influence of the activation function (when you try other activation functions than ReLU, you do not need to reach convergence in case you do not have enough time)
- what is the influence of the learning rate/learning rate strategy?
- how stable are the posteriograms from the network in time?
- how do the errors distribute depending on phonetic class?