KTH Computer Science
and Communication

DT2119 Lab3: Phoneme Recognition with Deep Neural Networks

This version of the lab is not complete and will be updated in the coming days. In the meantime, you can safely start and carry out up to and including Section 4. You can even start with Section 5, keeping in mind that I might add some more explanations and tasks there.

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

In the process you should also familiarise with the facilities at the Parallel Data Centre (PDC)[2] at KTH.

2 Task

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

- using predefined Gaussian HMM phonetic models, create time aligned phonetic transcriptions of the TIDIGITS database,
- define appropriate DNN models for phoneme recognition using Keras,
- 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 HMM models

[1]In this implementation you will use TensorFlow `https://www.tensorflow.org/` and Keras `https://keras.io/`

[2]`https://www.pdc.kth.se/`, this is also intended as an introduction to the work in the final project.

- 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 by queuing your jobs onto `tegner.pdc.kth.se`. See instructions in Appendix **??**, on how to setup the system.

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[3].

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 `pysndfile` 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`.

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

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:

```
>>> phoneHMMs = np.load('lab2_models.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 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. For example:

```
>>> from prondict import prondict
>>> phoneTrans = words2phones(wordTrans, prondict)
>>> phoneTrans
['sil', 'z', 'iy', 'r', 'ow', 'f', 'ao', 'r', 'th', 'r', 'iy', '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, for simplicity, we are not allowing any silence between words. This is usually done with the help of the *short pause* model sp. However, in order to use this model, you would need to modify the concatHMMs function you implemented in Lab 2[4].

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[5] 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]The reason is that, in Lab 2, we have made two assumptions to simplify the concatenation of models: there is only one transition into the final non-emitting state of each model and the prior probability for the states π_i for every model is 1 for the first state and 0 for the others states. In the short pause model this is not true because we could skip the model entirely.

[5]https://sourceforge.net/projects/wavesurfer/

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 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:
>>>         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 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 isolated digits files?

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):

Name	Content	set	N	D
`lmfcc_train_x`	MFCC features	train	$\sim$ 1356000	13
`lmfcc_val_x`	MFCC features	validation	$\sim$ 150000	13
`lmfcc_test_x`	MFCC features	test	1527014	13
`mspec_train_x`	Filterbank features	train	$\sim$ 1356000	40
`mspec_val_x`	Filterbank features	validation	$\sim$ 150000	40
`mspec_test_x`	Filterbank features	test	1527014	40
`train_y`	targets	train	$\sim$ 1356000	1
`val_y`	targets	validation	$\sim$ 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 the Keras categorical format, for example:

```
>>> from keras.utils import np_utils
>>> output_dim = len(stateList)
>>> train_y = np_utils.to_categorical(train_y, output_dim)
```

5 Phoneme Recognition with Deep Neural Networks

With the help of Keras[6], define a deep neural network that will classify every single feature vector into one of the states in `stateList`, defined in Section 4.

Note that Keras can run both on CPUs and GPUs. Because it will be faster on a fast GPU it is advised to run the trainings on `tegner.pdc.kth.se` ad PDC. However, it is strongly advised to test a simpler version of the models on your own computer first to avoid bugs in your code. Also, if for some reason you do not manage to run on `tegner`, you can still perform the lab, running simpler models on your own computer. The goal of the lab is not to achieve state-of-the-art performance, but to be able to compare different aspects of modelling, feature extraction, and optimisation.

Use the `Sequential` class from `keras.models` to define the model and the `Dense` and `Activation` classes from `keras.layers.core` 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. For the intermediate layers you can choose, for example, between `relu` and `sigmoid` activation functions.

With the method `compile()` from the `Sequential` class, decide the kind of `loss` function, and `metrics` most appropriate for classification. The method also lets you choose an `optimizer`. Here

[6] `https://keras.io/`

you can chose for example between Stochastic Gradient Descent (`sgd`) or the Adam optimiser (`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.

For each model, use the `fit()` method in the `Sequential` class to perform the training. You should specify both the training data and targets and the validation data. The last can be used to check if the model is over training. Here, 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.

Here are the suggested configurations to test, but you can test your favourite models if you manage to run the training in reasonable time.

- input: Liftered MFCCs, one to four hidden layers of size 256, rectified linear units
- input: filterbank features, one to four hidden layers of size 256, rectified linear units

For each utterance and time step, stack 11 filterbank features (that is, at time n, stack the features at times $[n-5, n-4, n-3, n-2, n-1, n, n+1, n+2, n+3, n+4, n+5]$). 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 $[5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5]$ for the first time step, $[4, 3, 2, 1, 0, 1, 2, 3, 4, 5, 6]$ for the second time step, and so on. Define and train a network that uses these augmented features as input.

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 (but be aware that convergence is much slower)
- 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. Use the method `predict()` from the class `Sequential` to evaluate the output of the network given the test frames in `FEATKIND_test_x`. 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` 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 edit distance to the correct transcription.

4. *edit distance at the phoneme level*: same as 3. but merging the states into phonemes as in 2.

For each of the above provide both global scores and confusion matrices.

6 Optional: Continuous Speech Recognition

This part of the lab has been excluded to keep the exercise within a reasonable time limit. It is also not fully tested. It has been included here in case you want to try it out. If you do try it, it would be great to receive feedback about it.

The simplest way to perform continuous speech recognition is to combine the HMM models in a loop and to search for the best path through the model at recognition time. For this purpose, implement the `hmmLoop` function from `lab3_proto.py` that takes as arguments a set of HMM models and a list of model names. The function will return a combined model including all the models in the list. For example:

```
>>> phoneLoop = hmmLoop(phoneHMMs, ['ao', 'ih', 's'])
```

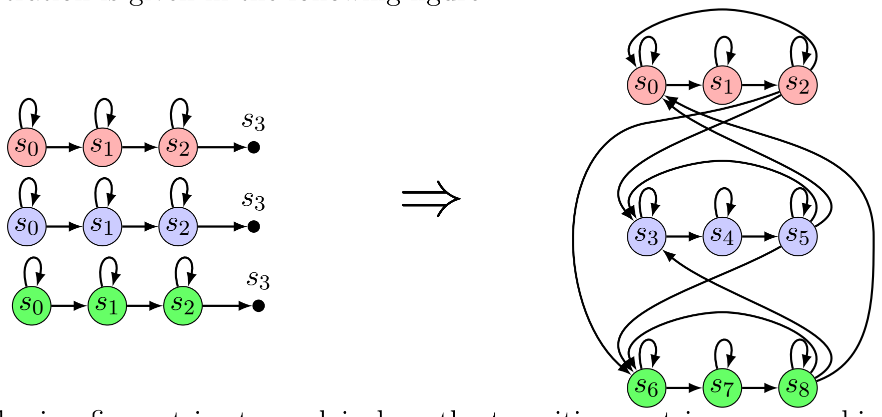
will create a loop between the three phonemes mentioned in the list. As for the `concatHMMs` function in Lab 2, we make the following assumptions for each of the input models:

1. the a priori probability of starting in the first state is one ($\pi_0 = 1$) and the probability of starting in other states is, therefore, zero,
2. the only allowed transition into the last non emitting state is from the last emitting state, that is the last raw in the transition matrix is all zeros besides the last term that is one.

Given these assumptions, you can combine the models by connecting every last emitting state in each model to every beginning emitting state in each model. We assume also that we distribute the probability mass evenly between these transitions.

Remember that each model in `phoneHMMs` has an extra state to allow the definition of the probability to leave the last emitting state. You should also keep track of the mapping between the states in the resulting model and the unique should also return a mapping between the states in the resulting model and the model names and state ids in the original models.

An illustration is given in the following figure:



The following figure tries to explain how the transition matrices are combined to form the resulting transition matrix.

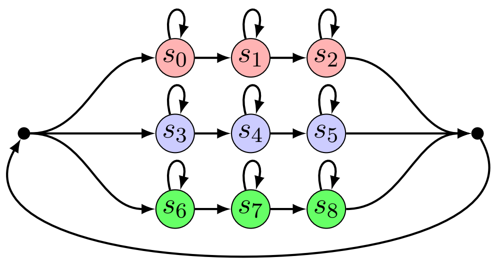
$$
\begin{array}{ccccccccc}
a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} & a_{06} & a_{07} & a_{08} \\
a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\
\enclose{circle}{a_{20}} & a_{21} & a_{22} & \enclose{circle}{a_{23}} & a_{24} & a_{25} & \enclose{circle}{a_{26}} & a_{27} & a_{28} \\
a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\
a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\
\enclose{circle}{a_{50}} & a_{51} & a_{52} & \enclose{circle}{a_{53}} & a_{54} & a_{55} & \enclose{circle}{a_{56}} & a_{57} & a_{58} \\
a_{60} & a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \\
a_{70} & a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\
\enclose{circle}{a_{80}} & a_{81} & a_{82} & \enclose{circle}{a_{83}} & a_{84} & a_{85} & \enclose{circle}{a_{86}} & a_{87} & a_{88}
\end{array}
$$

Note that the extra (non emitting) state in each model has been omitted in the final model. Note also that the transition probabilities in the circles have been added after combining the original transition probabilities. These are computed by distributing evenly the probabilities of leaving each model. For example, a_{20}, a_{23} and a_{26} are computed by dividing the probability a_{23} in the red model by 3. Similarly, a_{80}, a_{83} and a_{86} are computed by dividing the probability a_{23} in the green model by 3. Finally, the a priori probabilities of the states (not shown in the illustration), should also allow starting from any initial state of the models in the list. In this example: $\pi_0 = \pi_3 = \pi_6 = \frac{1}{3}$ and zero otherwise.

The function should allow models of different lengths. For example, you should be able to run it with your word level models that have different number of states:

```
>>> for word in prondict.keys():
>>>     wordHMMs[word] = concatHMMs(phoneHMMs, prondict[word])
>>> wordLoop = hmmLoop(wordHMMs)
```

Finally, note that the loop of models described above can be greatly simplified by adding non-emitting states in the model, as illustrated in the following figure:



However, this requires modifying the viterbi function to handle non-emitting states that do not consume any time step when traversed. In this exercise we prefer to keep the evaluation functions as simple as possible.

Run phoneme recognition using the `phoneLoop` obtained by combining all the phoneme models. Compare your results with the forced aligned transcriptions you obtained in Section 4.2.

Run continuous word recognition using the `wordLoop` obtained from all the word models. Compare your results with the true sequence of words.

Compute the accuracy in both cases using the edit distance between the recognised and reference sequence.

6.1 Hybrid HMM-DNN System

Repeat the tests in the previous section (phone and word recognition) where, instead of using the `log_multivariate_normal_density_diag` function and the Gaussian components from the HMM model, you use the posteriors estimated by the DNN models. Use the targets in the training data to estimate the a priori probability of each class and convert posteriors into likelihoods. Remember to map between the outputs of the network and the states in the HMM model (`phoneLoop` or `wordLoop`) by using the unique identifiers in `stateList`.

A Required Software on Own Computer

If you perform the lab in one of the CSC Ubuntu computers, or on `tengren.pdc.kth.se`, all the required software is already installed and can be made available by running:

```
source tools/modules
```

or

```
source tools/modules_tegner
```

from the lab package main directory.

If you wish to perform the lab on your own computer, you will need to install the required software by hand. Please refer to the documentation websites for more detailed information, here we just give quick instructions that might not be optimal.

A.1 Keras

If you use the Anaconda[7] Python distribution, you should be able to run

```
conda install keras
```

or

```
conda install keras-gpu
```

if you have a GPU that supports CUDA. With other versions of python there are similar `pip` commands.

A.2 Wavesurfer

This can be useful to visualise the results (label files) together with the wave files. The version of Wavesurfer that is part of the apt repositories unfortunately on tcl-tk 8.5, which also needs to be installed:

```
sudo apt install tk8.5 libsnack-alsa wavesurfer
```

If you don't want to istall tcl-tk 8.5, you can dowload the latest version of Wavesurfer from `https://sourceforge.net/projects/wavesurfer/`.

[7]`https://www.anaconda.com/`