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

In the process you should also familiarise with the facilities at the Parallel Data Centre (PDC)[2] at KTH. Alternatively, you will be able to use credits for the Google Cloud Platform. Instructions will be given in Canvas.

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

- perform and evaluate continuous speech recognition at the phoneme and word level using DNN-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.

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` or using the Google Cloud Platform. See instructions in Appendix B, on how to use the PDC resources, or check instructions on Canvas for the GCP.

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 `phoneHMMs['sp']` that has a single emitting state and can be skipped in case there is no silence. However, in order to use this model, you would need to modify the `concatHMMs` function you implemented in Lab 2. In Appendix A you will find instructions on how to do this, if you want to obtain more accurate transcriptions. If you follow those instructions, the `words2phones` function, will have to insert `sp` after the pronunciation of each word. You can use the `addShortPause` argument provided in the prototype function to switch this behaviour on and off.

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`[4] 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]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 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 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), 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 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[5], 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 Keras 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.

---

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

Note that Keras 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 `tegner.pdc.kth.se` ad PDC or using the Google Cloud Platform. 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 GPUs, 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. 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.

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 you can choose 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 and validation data with the respective targets. What is the purpose of the validation data? 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 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. Use the method `predict()` from the class `Sequential` to 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?

A Generalisation of `concatHMMs`: `concatAnyHMM`

The instructions in Lab 2 on how to implement the `concatHMMs` function were correct under two assumptions:

1. the a priori probability of the states π_i is non-zero only for the first state: $\pi = [1, 0, 0, \ldots]$

2. there is only one transition into the last non-emitting state, and it comes from the second last state: $a_{iN-1} = 0 \quad \forall i \in [0, N-3]$.

This situation is illustrated by the following figure:
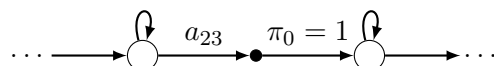
where we have only displayed the last two states of the previous model (one emitting and one non-emitting) and the first state of the next model.

This allowed us to easily skip the non-emitting state by connecting the last emitting state of the previous model to the first emitting state of the next model like this:
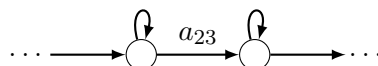
The above assumptions are verified by all the left-to-right models you have considered in Lab 2. However, in the general case, those assumptions are not fulfilled. In particular, the short pause model in `phoneHMMs['sp']` violates both these assumptions. It is defined to include a single emitting state (in case of very short pauses) and to be skipped completely, in cases there is no pause between words. The transition model looks like this:
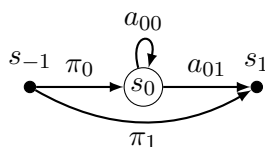
Here, we have added an extra non-emitting state s_{-1} in order to illustrate the effect of the prior probability of the states π. Adding this extra non-emitting state can be done for any model that we have seen so far. For example, the standard three state left-to-right model can be depicted like this:
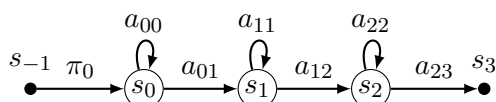
If we release the two assumptions above, once we remove the intermediate non-emitting state between two consecutive models, we will be able to go from any state s_i of the first model to any state s_j of the second. The corresponding probability of the transition is the product of probability a_{iN-1} of going from s_i to the last non-emitting state of the previous model by the prior probability π_j of starting in the s_j state in the subsequent model. Let's say we want to concatenate the following two generic models (both with three emitting states).

π_0	π_1	π_2	π_3	ρ_0	ρ_1	ρ_2	ρ_3
a_{00}	a_{01}	a_{02}	a_{03}	b_{00}	b_{01}	b_{02}	b_{03}
a_{10}	a_{11}	a_{12}	a_{13}	b_{10}	b_{11}	b_{12}	b_{13}
a_{20}	a_{21}	a_{22}	a_{23}	b_{20}	b_{21}	b_{22}	b_{23}
0	0	0	1	0	0	0	1

Here we have called π_i and a_{ij} the prior and transition probability in the first model, and ρ_i and b_{ij} the prior and transition probability in the second model to be able to distinguish them more easily. The prior vector and the transition matrix of the concatenation of the two models is:

π_0	π_1	π_2	$\pi_3\rho_0$	$\pi_3\rho_1$	$\pi_3\rho_2$	$\pi_3\rho_3$
a_{00}	a_{01}	a_{02}	$a_{03}\rho_0$	$a_{03}\rho_1$	$a_{03}\rho_2$	$a_{03}\rho_3$
a_{10}	a_{11}	a_{12}	$a_{13}\rho_0$	$a_{13}\rho_1$	$a_{13}\rho_2$	$a_{13}\rho_3$
a_{20}	a_{21}	a_{22}	$a_{23}\rho_0$	$a_{23}\rho_1$	$a_{23}\rho_2$	$a_{23}\rho_3$
0	0	0	b_{00}	b_{01}	b_{02}	b_{03}
0	0	0	b_{10}	b_{11}	b_{12}	b_{13}
0	0	0	b_{20}	b_{21}	b_{22}	b_{23}
0	0	0	0	0	0	1

You can verify that making the two assumptions at the beginning of this section, we fall back to the same solution as in Lab 2, where only the term $a_{23}\rho_0 = a_{23}$ survives.

If we iterate this process, if the model concatenated so far has M emitting states, we will need to multiply:

- the prior at the non-emitting state M by the priors of the next model,

- the transition probabilities in column M up to row $M-1$ to the prior of the next model.

This is similar to what we did with $\pi_3, a_{03}, a_{13}, a_{23}$ in the previous example.

Here is a simplified example where we concatenate a strict left-to-right model to the `sp` model, and then to a strict left-to-right model again (which is the usual case in practice):

1	0	0	0		ρ_0	ρ_1		1	0	0	0
a_{00}	a_{01}	0	0		b_{00}	b_{01}		c_{00}	c_{01}	0	0
0	a_{11}	a_{12}	0		0	1		0	c_{11}	c_{12}	0
0	0	a_{22}	a_{23}					0	0	c_{22}	c_{23}
0	0	0	1					0	0	0	1

The resulting model is:

$$
\begin{array}{cccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
a_{00} & a_{01} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & a_{11} & a_{12} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & a_{22} & a_{23}\rho_0 & a_{23}\rho_1 & 0 & 0 & 0 \\
0 & 0 & 0 & b_{00} & b_{01} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & c_{00} & c_{01} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & c_{11} & c_{12} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & c_{22} & c_{23} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}
$$

Write the function `concatAnyHMM` in `lab3_proto.py` that implements this general concatenation.

B PDC Specific Instructions

In order to run Keras and Tensorflow on GPUs, you may use nodes on `tengren.pdc.kth.se`. You can refer to the presentation from PDC you can find in the course web page for detailed information, and to the `https://www.pdc.kth.se/` website for detailed instruction. Here we give an example usage that should work for carrying out the relevant steps in this lab.

1. First you need to authenticate with the help of kerberos on your local machine. From a machine where kerberos is installed and configured run:

   `kinit -f -l 7d <username>@NADA.KTH.SE`

   to get a 7 days forwardable ticket on your local machine. If you are using a CSC Ubuntu machine, run instead

   `pdc-kinit -f -l 7d <username>@NADA.KTH.SE`

   this will keep also the ticket `<username>@KTH.SE` allowing you to see the files in your home directory on AFS.

2. then you login with `ssh`, (or `pdc-ssh` on CSC Ubuntu)[6]:

   `[pdc-]ssh -Y <username>@tegner.pdc.kth.se`

3. the lab requires several hundreds of MB of space. If you do not have enough space in your home directory, put the lab files under

   `/cfs/klemming/nobackup/<first_letter_in_username>/<username>/`

   remember that the data stored there is not backed up. If you need to copy the files back and forward with your local machine, check the `rsync` command,

4. In order to queue your job, you will use the command `sbatch`. Create a `sbatch` script called, for example, `submitjob.sh` with the following content, assuming that the script you want to run is called `lab3_dnn.py`. Note that `sbatch` uses the information in commented lines starting with `#SBATCH`. If you want to comment those out, put an extra `#` in front of the line.

   ```
   #!/bin/bash

   # Arbitrary name of the the job you want to submit
   #SBATCH -J myjob

   # This allocates a maximum of 20 minutes wall-clock time
   # to this job. You can change this according to your needs,
   # but be aware that shorter time allocations are prioritised
   #SBATCH -t 0:20:00

   # set the project to be charged for this job
   ```

---

[6]see `https://www.pdc.kth.se/resources/software/login-1/.ssh-conf` to configure ssh correctly for PDC

DT2119 Speech and Speaker Recognition • Vårterminen 2018
Giampiero Salvi

```
# The format should be edu<year>.DT2119
#SBATCH -A edu18.DT2119

# Use K80 GPUs (if not set, you might get nodes without a CUDA GPU)
# If you have troubles getting time on those nodes, try with the
# less powerful Quadro K420 GPUs with --gres=gpu:K420:1
#SBATCH --gres=gpu:K80:2

# Standard error and standard output to files
#SBATCH -e error_file.txt
#SBATCH -o output_file.txt

# Run the executable (add possible additional dependencies here)
module add cuda
module add anaconda/py35/4.2.0
source activate tensorflow
python3 lab3_dnn.py
```

5. submit your job with

   ```
   sbatch submitjob.sh
   ```

6. check the status with

   ```
   squeue -u <username>
   ```

   The column marked with `ST` displays the status. `PD` means pending, `R` means running, and so on. Check the `squeue` manual pages for more information.

   You can check the standard output and standard error messages of your job in `output_file.txt` and `error_file.txt`. If you wish to kill your job before its normal termination, use `scancel <jobid>`.

B.1   Using `salloc` instead of `sbatch`

In some cases `sbatch` might not be the best choice. This is the case, for example, when you want to debug your code on the computational node, or if `sbatch` does not work well with your code. In this case, follow the above instructions up to point number 4 and then:

5. on `tegner` you get time allocation running[7]:

   ```
   salloc -t <hours>:<minutes>:<seconds> -A edu18.DT2119 --gres=gpu:K80:2
   ```

6. you will get a message like the following:

   ```
   salloc: Granted job allocation 41999
   salloc: Waiting for resource configuration
   salloc: Nodes t02n29 are ready for job
   ```

---

[7]The last optional argument ensures you get a node with NVidia Tesla K80 GPU , otherwise you will get a node with NVidia Quadro K420. Refer to the page `https://www.pdc.kth.se/resources/computers/tegner/hardware` for more information about the hardware on tegner.

where the job number (`41999`) and the associated node (`t02n29`) will vary.

7. From another teminal window **on your local machine**, login on that specific node:

   ```
   [pdc-]ssh -Y t02n29.pdc.kth.se
   ```

   running `ssh` from `tegner.pdc.kth.se` to the node **will not work**

8. run the `screen` command. This will start a screen terminal, that will allow you to *detach* the terminal and logout without stopping the process you want to run[8]

9. in order to get the required software, from the lab main directory run

   ```
   module add cuda
   module add anaconda/py35/4.2.0
   source activate tensorflow
   ```

10. if everything went well, now you can run your script with, for example

    ```
    python3 lab3_dnn.py |& tee -a logfile
    ```

    where the `tee` command will display the standard output and standard error of the training command in the terminal as well as appending it to the `logfile`

11. if you want to logout while the program is running, hit `ctrl+a` and then `d` to detach the screen and logout. When you login again into that node, you can run `screen -r` to reattach the terminal.

12. while you are logged in on the specific node, you can check CPU usage with the command `top` and GPU usage with the command `nvidia-smi`.

   NOTE: if you use this method, the time allocation system will continue charging you time, even if the process has terminated, until you logout from the node.

   Use `squeue [-u <username>]` to see your time allocated jobs, and `scancel jobid` to remove a submitted job.

---

[8]Refer to the screen manual for further info `http://linux.die.net/man/1/screen`

C 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 the commands shown in the previous section.

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.

C.1 Keras

If you use the Anaconda[9] 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.

C.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/`.

---

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