# *SNLP assignment 5: word class prediction with neural networks*

*Deadline: Jul 11, 2018 @ 10:00 CEST*

In this set of exercises, we are going to use neural networks for predicting whether a given word in German is a noun or a verb. As formulated here, this is a toy example. However, understanding this simple example well will allow you to apply the methods to many other NLP tasks.

Why am I doing this?

- Experiment with a range of artificial neural network models
- Compare linear and non-linear solutions
- Work with sequences, and the ways to turn them into features for ANNs

## *Data*

The data for this exercise is from from the TüBA-D/Z treebank. Data is presented in two compressed tab-separated files, `train.txt.gz` and `test.txt.gz`. The first field in the data files is the word class (*noun* or *verb*) and the second field is the word itself. A short segment from the data file is presented in Figure 1. You should use `train.txt.gz` for training and development, and `test.txt.gz` only for testing (for comparing results from different models).

```
noun aalstrich
noun abänderung
verb abarbeiten
noun abarbeitung
verb abdrehen
verb abfallen
```

Figure 1: A short excerpt from the data to be used in this exercise. All words are converted to lowercase to make the classifiers' task more difficult.

The exercises below should be implemented as a single Python script, `word-class.py`. Recommended variable names for some of the data structures are printed on the left margin, and referred to with this name in instructions that follow. For exercises where you are asked to print out some information, you should just print them to the console (e.g., use the `print()` function). You do not need to write down the answers to the questions on the right margin, but trying to find the answers to these questions will help understanding the methods better. For these exercises, you are recommended to use use the Keras library.[1] Please contact the instructors if you are keen on using another library.

You do not need to re-implement the standard library functions (from `Keras`, `sklearn`, `numpy`, ...) for any of the tasks in this assignment.

[1] Keras web site, `https://keras.io/`, contains quick tutorials that you may find useful to go through before working on this set of exercises. Please also note that installing Keras and required back end libraries may take some effort in some computing environments. You are recommended to start early.

## *Exercises*

**Exercise 1.** *Data preprocessing, encoding*

Read the training and test files, into two lists (each): a list containing the labels (`noun` or `verb`) and a list corresponding words.

`train_y` Convert the training label sequence to sequences of zeros and ones (arbitrarily mapping one of the labels to `0`, the other to `1`). Use the variable name `train_y` for the resulting list.

`train_x` Map each Unicode character in the training words to a unique integer and each word to a list of integers that correspond to sequences of characters they are composed of. Reserve an integer value for 'unknown' characters. Pad the numeric sequences corresponding to words, such that they are the same length as the longest word, and prepended by `0`s when necessary. Use the variable name `train_x` for the resulting list.

TIP: you can use `keras.preprocessing.sequence.pad_sequences`. The resulting array should have as many rows as the training instances, and the number of columns should be the length of the longest sequence.

`test_y` Convert the test label sequence to an array of 0 and 1. Following the same convention as the training set.

`test_x` Convert your test words similar to the `train_x`. Note that you should use the same mapping as in the training words, using the 'unknown' for the characters that are not observed in the training set.

`test_onehot` `train_onehot` Create versions of training and test sequences, where each character is encoded using one-hot encoding. TIP: resulting one-hot arrays should be two dimensional, with same number of rows and the number of columns equal to the size length as the longest word times the number of unique characters (including unknown).

Some tips on Exercise 2 (and later exercises):

- The type and number of units at the final layer, as well as the error function is determined by the fact that we are doing binary classification. Hence, you should use a single 'sigmoid' unit at the final layer.
- `fit()` function of Keras models have a `validation_split` parameter which may be handy for automatically splitting part of the data as development/validation set.
- Although there are no set rules, `adam` is a good default as `optimizer`.
- Again, the choice of batch size depends on the data/model. It should mainly affect the training/test time, but may also affect the accuracy of your model. You can choose a `batch_size` of 32.

Although not required for this exercise, you are recommended to experiment with other options: e.g., different activation functions, number of units, or layers.

**Exercise 2.** *Simple feed-forward network (MLP)*

Train (and tune) a multi-layer perceptron (MLP) model using `train_onehot` as input and `train_y` as output. Use a single hidden (dense) layer of size 64 with `relu` activation function, and use drop-out after the hidden layer. Tune your model to pick the best drop-out rate (between 0.1 and 0.9), and the best epoch value (up to 30).[2]

With the model you tuned, print out macro-averaged precision, recall and F-score, and accuracy on the test set.[3]

[2] You are recommended to tune the model hyperparameters automatically, but manually trying a few options is also fine.

[3] Is accuracy a good metric here? Why (not)? Is your model doing well, or doing anything useful at all? Would you get better or worse performance if you used the numeric sequences (instead of one-hot)?

**Exercise 3.** *Effect of padding direction*

You were instructed to pad zeros to the beginning of your input sequences in Exercise 1.

Repeat Exercise 2 with input padded with 0s at the end of sequences instead of at the beginning.[4]

[4] Do you get better or worse result? Why (or why not)? Would you get better results if data was not 'case normalized'?

**Exercise 4.** *Back to logistic regression*

Create, train (and tune) a model with a single 'sigmoid' unit on (again) one-hot version of the training set (`train_onehot`), tuning the (L2) regularization strength.

Print out macro-averaged precision, recall and F-score, and accuracy on the test set.[5]

[5] Can this model have a chance to learn anything useful? Can you think of another input representation for this model to work (better)?

**Exercise 5.** *Recurrent networks*

Create, train (and tune) a model on training data (`train_x`, `train_y`) with an `Embedding` layer followed by a `GRU` layer, and finally a 'sigmoid unit' for binary classification. Use drop-out after both `Embedding` and `GRU` layers. You can experiment with any of the hyperparameters you like.[6]

Print out macro-averaged precision, recall and F-score, and accuracy on the test set (`test_x`, `test_y`) for the model you choose.[7]

[6] How many hyperparameters do we have?

[7] Would padding direction have an effect similar to Exercise 3? Could we use one-hot representations here? What are the benefits (or drawbacks) of using embeddings instead?