IN4080 Autumn 2020, Assignment 3

Pierre Lison, plison@nr.no

October 12, 2020

Practical details

Your answer should be delivered in `devilry` no later than Friday, November 6 at 23:59.

Mandatory assignment 3 consists of three parts. In Part 1, you will develop a chatbot model based on a Dual Encoder architecture (40 points). In Part 2, you will implement a simple silence detector using speech processing (35 points). Finally, Part 3 will help you understand some core concepts in dialogue management through the construction of a very simple, simulated talking elevator (25 points).

You should answer all three parts. You are required to get at least 60 points to pass. The most important is that you try to answer each question (possibly with some mistakes), to help you gain a better and more concrete understanding of the topics covered during the lectures.

1. We assume that you have read and are familiar with IFI's requirements and guidelines for mandatory assignments, see here and here.
2. This is an individual assignment. You should not deliver joint submissions.
3. You may redeliver in Devilry before the deadline, but include all files in the last delivery. Only the last delivery will be read! If you deliver more than one file, put them into a zip-archive.
4. The preferred format for the assignment is a **Jupyter notebook** containing both your code and explanations about the steps you followed. *The explanations are at least as important as the code itself!*
5. Name your submission `your_username_in4080_mandatory_3`

Important note

You can work on this assignment either on the IFI machines or on your own computer. But note that this assignment requires packages that are not part of the version of Anaconda installed by default on the IFI machines, so you should add the following to your environment variable if you work on IFI machines:

```
export PATH=/hom/plison/anaconda3/bin/:$PATH
```

If you are using the VDI solution put in place by the department, note that the directory will not be visible from there – but you can copy the Anaconda directory to your home directory using `ssh` and then run Python from that copy.
If you wish to work on your own private computer, you can install the necessary packages through `pip install tensorflow==2.3.0 tensorflor-text==2.3.0 tensorflow-hub glog` and `conda install tk`. Send us an email if you run into problems. Note that `tensorflow-text` (required to run the utterance encoder of Part 1) is currently not supported on Windows.

Technical tip: Some of the tasks in this assignment will require you to extend methods in classes that are already partly implement. In order to implement those methods directly within a Jupyter notebook, you can use the function `setattr` to attach a method to a given class:

```
class A:
    pass
a = A()

def foo(self):
    print('hello world!')

setattr(A, 'foo', foo)
a.foo() # hello world!
```

Using this simple trick, you can extend classes defined in external files directly from Jupyter, without having to edit the file. This way, you can deliver the full assignment in the form of a single Jupyter notebook.

Part 1: A data-driven chatbot

We will build a retrieval-based chatbot based on a dialogue corpus of movie and TV subtitles. More specifically, we will rely on a so-called *dual encoder* model. In this model, both the user utterance (called the "context") and the chatbot response are converted into embeddings, and the dot product of these two vectors can be seen as a numerical score that expresses how good the response is for the provided context. In other words, given a user utterance, we first compute its embedding, and then compute its dot product against all possible responses (represented by their respective embeddings). We can then simply take the response with the highest score. See Fig. 1 for an illustration.

Have a look at the code in the file `chatbot.py`, where three methods should be implemented. We'll work with a dataset of movie and TV subtitles for English from the

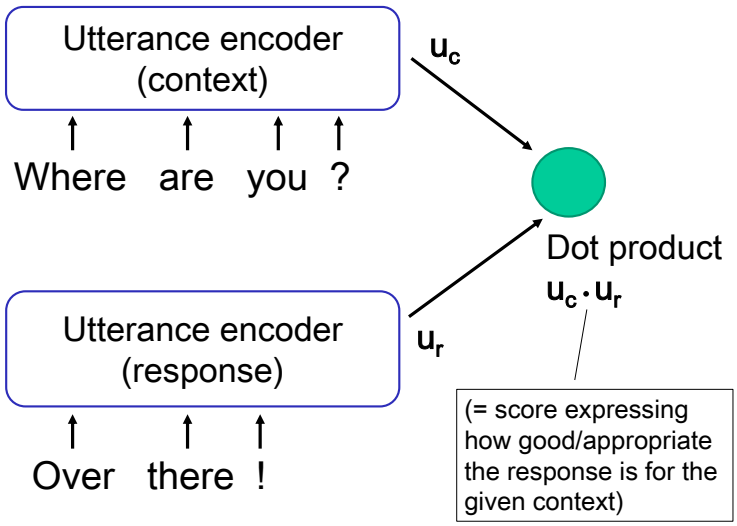

Figure 1: Dual encoder architecture.

OpenSubtitles dataset[1], and restricted for this assignment to comedies.

The dataset is in the (compressed) file `data/en-comedy.txt.gz`. The data contains one line per utterance. Dialogues from different movies or TV series are separated by lines starting with ###.

Basic system

The first task will be to read through the dataset in order to extract pairs of (context, response) utterances. But we want to limit our extract to "high-quality" pairs, and discard pairs that contain text that is not part of a dialogue, such as subtitles indicating "(music in background)". We want to enforce the following criteria:

- The two utterances should be consecutive, and part of the same movie/TV series.
- ### delimiters between movies or TV series should be discarded.
- Pairs in which one string contains parentheses, brackets, colons, semi-colons or double quotes should be discarded.
- Pairs in which one string is entirely in uppercase should be discarded.
- Pairs in which one string contains more than 10 words should be discarded.
- Pairs in which one string contains a first name should be discarded (see the JSON file `FIRST_NAMES` to detect those), as those are typically names of characters occurring in the movie or TV series.

If you wish, you can add your own criteria to further enhance the quality of the (context, response) pairs.

[1]See `http://opus.nlpl.eu/OpenSubtitles-v2018.php` and Lison and Tiedemann (2016); Lison et al. (2018) for details on this corpus.

Task 1

Implement the method `_extract_pairs` that extracts a list of (context, response) pairs, where the context and response are strings, ensuring that the aforementioned criteria are satisfied.

Once Task 1 is complete, you can load your chatbot with `chatbot.Chatbot()`. The initialisation will automatically compute embeddings for all the responses in the extracted pairs. These embeddings are computed using `ConveRT`, which is a deep, pre-trained utterance encoder described in Henderson et al. (2019). I recommend you to have a look at the paper if you wish to know more about this model.

Note that the computation of those embeddings will take a few minutes (perhaps up to half an hour depending on your machine). While working on this assignment, it may be useful to store the resulting matrix in a file to avoid having to recompute it every single time you load the chatbot.

The next step is then to implement the method that, given a new user utterance, will select the most appropriate response.

Task 2

Implement the method `get_response` that takes a new user input and returns the string with the best response according to the model. The best response should be selected from the set of responses extracted in the (context, response) pairs from Task 1. To score the response, simply compute the dot product between the (context) embedding for the user input and the possible responses. The best response is then the one yielding the largest value for this dot product.

You can then test how your chatbot works on some utterances such as "Who are you?", "How old are you?" "Where are you?", "Are you stupid?", "Did you kill him?".

Technical tip: When working with `numpy` arrays/matrices, you should always try to avoid going through explicit loops, as looping over values of a `numpy` array is highly inefficient. For instance, instead of computing the dot product of each response one after the other (within a loop), you can compute the dot product of all responses in one single dot product operation – which is way more efficient! See `https://realpython.com/numpy-array-programming/` for more details on the best way to work with `numpy` arrays.

Fine-tuning

If you have correctly implemented Task 1 and 2, your chatbot should now be able to come with reasonable/funny answers to user utterances. But the current chatbot has one limitation, namely that the utterance encoder from conveRT is not adapted to the

kind of dialogue found in subtitles. So it would be useful to fine-tune the model on domain-specific data, namely the (context, response) pairs we have extracted.

For this fine-tuning, we will simply add a single feed-forward ("dense") layer consisting of a linear transformation of the embeddings followed by a non-linear activation function such as a reLU (rectified linear unit). The parameters (=weights) of this layer will be learned based on the (context, response) pairs. In other words, the context and response utterances will first be converted into embeddings using `ConveRT`, but the response vectors will be transformed through a learned layer before being applied to compute the dot product. See Fig. 2 for an illustration.

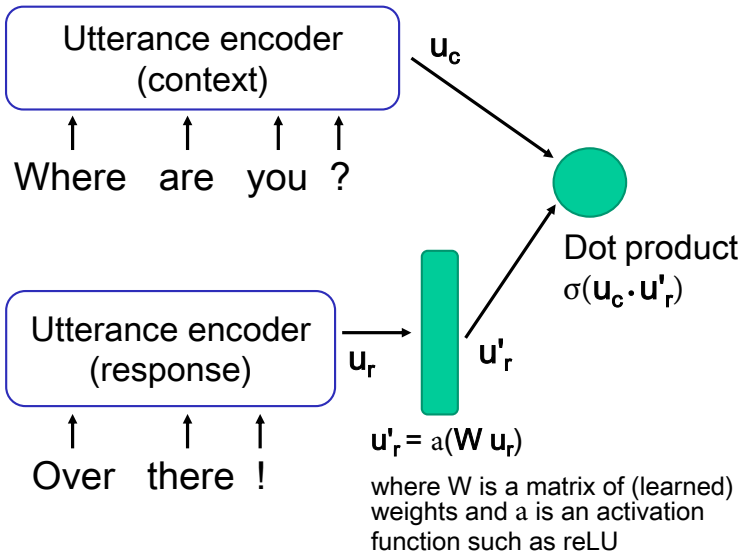

Figure 2: Dual encoder architecture with an additional feed-forward layer to fine-tune the model.

Much of the code for this fine tuning is already implemented in the method `fine_tune`, but the missing part is the code for preparing the training data.

Task 3

Implement the method `_get_training_data` that create a training set for the fine-tuning. This training set will be based on the extracted (context, response) pairs. The training set will need to include both positive examples (context and response embeddings corresponding to actual observed responses, and that should therefore have a high score) and negative examples (context and response embeddings that are not related and should therefore have a low score). The output for the positive examples should be 1 and the output for the negative examples should be 0. For the negative examples, the easiest is to take the embedding of another random response from the extracted pairs.

Once you are done, run the `fine_tune` method to fit the feed-forward layer to your examples. The method will loop 10 times through your training set. Once training is complete, the matrix containing the response embeddings will be transformed by applying the learned layer to the `ConveRT` embeddings.

You can test your chatbot again with the same examples as before. If the training set you have constructed is correct, your chatbot should now provide different responses (hopefully even better than the baseline system).

Part 2: Speech processing

In this section, we will learn how we can perform some basic speech processing using operations on `numpy` arrays! We'll work with `wav` files. A `wav` file is technically quite simple and encodes a sequence of integers, where each integer express the magnitude of the signal at a given time slice. The number of time slices per second is defined in the frame rate, which is often 8kHz, 16kHz or 44.1kHz. So a file with 4 seconds of audio using a frame rate of 16 kHz will have a sequence of 64 000 integers. The number of bits used to encode these integers may be 8-bit, 16-bit or 32-bit (more bits means that the quantization of the signal at each time slice will be more precise, as there will be more levels). Finally, a `wav` file may be either mono (one single audio channel) or stereo (two distinct audio channels, each with its sequence of integers).

To read the audio data from a `wav` file, you can use `scipy`:

```
import scipy.io.wavfile
fs, data = scipy.io.wavfile.read("data/excerpt.wav")
data = data.astype(np.int32)
```

where `data` is the numpy array containing the actual audio data, and `fs` is the frame rate. Note that I am converting the integers into 32-bit to avoid running into overflow problems later on (i.e. when squaring the values).

Task 4

- Plot (using `matplotlib.pyplot`) the waveform for the full audio clip.
- For speech analysis, looking directly at the signal magnitude is typically not very useful, as we can't typically distinguish speech sounds based on the waveform. A representation of the signal that is much more amenable to speech analysis is the *spectrogram*, which represents the spectrum of *frequencies* of a signal as it varies with time.

  Plot the spectrogram of the first second of the audio clip, using the function `matplotlib.pyplot.specgram`.

Now, let's say we wish to remove periods of silence from the audio clip. There are many methods to detect silence (including deep neural networks), but we will rely here on a simple calculation based on the energy of the signal, which is the sum of the square of the magnitudes for each value within a frame:

$$E = \sum_{i=1}^{N} X[t]^2 \tag{1}$$

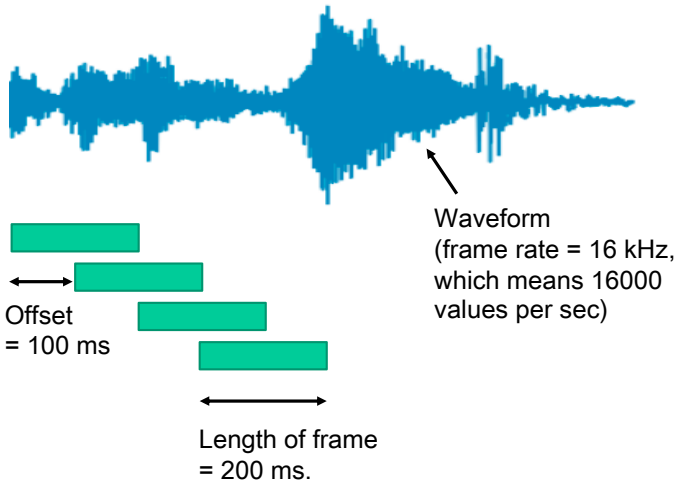

Figure 3: Moving window on audio data.

Task 5

- Loop on your audio data by moving a frame of 200 ms. with a step of 100 ms., as shown in Fig. 3.
- For each frame, compute the energy as in Eq. (1) and store the result.
- Now that we have the energy for each frame of the audio clip, we can calculate the mean energy per frame, and define silent frames as frames that have less than $1/10$ of this mean energy. Determine which frame is defined as silent according to this definition.
- Show on the plot of the waveform which segment is determined as silent, using the method `matplotlib.pyplot.hlines`.
- If your calculations are correct, you will notice that the method has marked many short, isolated segments (e.g. in the middle of a word) as silent. We want to avoid this of course, so we will rely on a stricter requirement to define whether a frame should be considered silent or not: we only mark a frame as silent if the frame *and all of its neighbouring frames* have less than $1/10$ of the mean energy. We can defines the neighbourhood of a frame as the five frames before and the five frames after.
- Show again on the plot of the waveform which segment is determined as silent using the stricter definition above.
- (Optional) Modify the audio data by cutting out frames marked as silent, and using the method `scipy.io.wavfile.write` to write back the data into a `wav` file. Don't forget to convert back the data array to 16-bit using `astype(np.int16)` before saving the data to the `wav` file.

For the bravest amongst you: Until now, we have only looked at the detection of silent frames. But what we are truly interested in is to distinguish between speech sound and the rest, which may be either silence or non-speech sounds (noise). For this, we need to analyse the audio in the *frequency domain* and look at which range of frequencies is most active within a given frame.

How do we achieve this? The key is to apply a *Fast Fourier Transform* (FFT) to go from an audio signal expressed in the time domain to its corresponding representation in the frequency domain. We can then look at frequencies in the speech range (typically between 300 Hz and 3000 Hz) and compute the total energy associated to that range. You can look here to know more about how to use `numpy` and `scipy` to perform such operations. This question is not part of the obligatory assignment, but feel free to experiment with it if you are interested in audio processing. Students that manage to come with a solution for detecting silence and non-speech frames using FFT will get a bonus of +15 points.

Part 3: Talking Elevator

Let's assume you wish to integrate a (spoken) conversational interface to one of the elevators in the IFI building. The elevator should include the following functions:

- It should greet the user and ask them where they wish to go.
- If the user express a wish to go to floor X, the elevator should go to that floor. The interface should allow for several ways to express a given intent, such as "Please go to the Xth floor" or "Floor X, please".
- The user requests can also be relative, for instance "Go one floor up". The interface should also be able to answer questions such as "Which floor are we in?".
- The elevator should provide *grounding* feedback to the user. For instance, it should respond "Ok, going to the Xth floor" after a user request to move to X.
- The elevator should handle misunderstandings and uncertainties, e.g. by asking the user to repeat, or asking the user to confirm if the intent is uncertain (say, when its confidence score is lower than 0.7).
- The elevator should also allow the user to ask where the office of a given employee is located. For instance, the user could ask "where is Erik Velldal's office?", and the elevator would provide a response such as "The office of Erik Velldal is on the 4th floor. Do you wish to go there?".
- If the user asks the elevator to stop (or if the user says "no" after a grounding feedback "Ok, going to floor X"), the elevator should stop, and ask for clarification regarding the actual user intent.

You will find in `elevator.py` an implementation of a simulated elevator in which you can test the behaviour of your conversational interface in an interactive manner. Fig. 4 shows the graphical user interface developed for this simulated elevator.

Figure 4: Simulated talking elevator.

To also simulate the occurrence of speech recognition errors (even though the actual interface is text based), the implementation introduces some artificial noise into the user utterances (e.g. swapping some random letters from time to time). Each user utterance comes associated with a confidence score (which would come from a speech recogniser in a real system, but is here also artificially generated).

Task 6

Implement the method `process(user_message, confidence_score)` in order to respond to the user (both verbally and non-verbally, through physical movements) such that the requirements above are satisfied.
Regarding the questions about the office location of particular employees, you can limit yourself to the offices of 5-6 employees at IFI.
To interpret the user request, the easiest is to rely on handcrafted regular expressions to detect specific patterns (such as "go to floor X"). But if you prefer data-driven approaches, you may alternatively rely on a classification model learned from a list of labelled examples. You are free to decide on the best way to implement your system, as long as you motivate your design choices.
Your implementation will likely require the introduction of new instance variables to keep track of some aspects of the dialogue state (such as the last user request or the last system question). Your implementation should take into account the provided confidence score (if the score is too low, the system should seek to clarify the user's request). It is up to you to decide what it the best way to handle such uncertainties.
For the actual response, the system can call the methods `go_to` (to go to a particular floor) and `respond` (to give a verbal answer to the user). Some user requests might trigger both verbal and non-verbal responses. The elevator can be stopped during a movement by setting the variable `urgent_stop` to `True`.

There is no obvious right/wrong answer for this exercise – the main goal is to reflect on how to design conversational interfaces in a sensible manner – in particular when it comes to handling uncertainties and potential misunderstandings.

References

Henderson, M., Casanueva, I., Mrkšić, N., Su, P.-H., Vulić, I., et al. (2019). ConveRT: Efficient and accurate conversational representations from transformers. *arXiv preprint arXiv:1911.03688*.

Lison, P. and Tiedemann, J. (2016). Opensubtitles2016: Extracting large parallel corpora from movie and tv subtitles. In *Proceedings of the 10th International Conference on Language Resources and Evaluation (LREC 2016)*.

Lison, P., Tiedemann, J., and Kouylekov, M. (2018). OpenSubtitles 2018: Statistical rescoring of sentence alignments in large, noisy parallel corpora. In *Proceedings of the 11th International Conference on Language Resources and Evaluation (LREC-2018)*, Miyazaki, Japan.