IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

UE TLFT

Lab 3: Deep Learning

Gábor Bella, Yannis Haralambous (IMT Atlantique)

Contents

1 Exploration of a word embedding vector space

1.1 Visual Exploration

Load the word embedding visualisation tool from `https://www.cs.cmu.edu/~dst/WordEmbeddingDemo/`. Explore the behaviour of vector arithmetic through examples such as:

```
Spain is to Madrid as France is to... ?
France is to wine as Germany is to... ?
Man is to researcher as woman is to... ?
```

Do try as many examples as you can invent. You may, for example, try to find out whether the vector space is biased with respect to race or gender, or in any other manner. In order to do so, do not only observe the closest vector obtained, but also the top-ten results, using the graphical tool, hovering over the dot that represents the result of the vector subtraction and addition.

1.2 Programmatic Exploration

We will use `gensim`:

```
pip install gensim
```

You can load a pre-trained embedding from the web as follows:

```
import gensim.downloader
vectors = gensim.downloader.load('word2vec-google-news-300')
```

Beware, the model above has a size of 1.6GB!

You can then obtain a word vector simply by writing, e.g., `vectors["dog"]`. You can do arithmetic with these numpy vectors. Finally, for obtaining the word vectors closest to a given vector `v`, you can use the method:

```
vectors.most_similar(v)
```

1.3 Topic Classification using Word Vectors

Using `gensim` and simple vector arithmetic, implement a topic classifier on a predefined set of topics of your choice. You can use this simple method, for example, to detect when a user mentions a specific topic to your chatbot.

A topic will be defined as:

- a topic label;
- a small set of words characteristic of the topic.

For example, you could define four topics as follows:

```
topics = {
  "weather": ["weather", "sun", "rain", "wind", "cold", "warm", "hot", "temperature"], \
  "traffic": ["car", "bus", "road", "street", "busy", "intersection", "lane", "drive"], \
  "cooking": ["food", "drink", "cook", "bake", "fry", "oven", "meat", "recipe","ingredient"], \
  "travel":  ["fly", "plane", "train", "drive", "holiday", "travel", "vacation", "beach"] \
        }
```

Of course,

You can classify an input text according to these four topics by:

1. pre-computing an embedding vector for each topic;
2. computing an embedding vector for the input text;
3. choosing the topic that has the vector the most similar to the that of the input text.

To compute an embedding vector for a bag of words, you can simply sum up the individual word vectors:

$$v_{\text{hello world}} = v_{\text{hello}} + v_{\text{world}}.$$

To compare two vectors, the simplest method is to use cosine similarity, which depends only of the angle between vectors and not of their magnitude:

$$\text{sim}(v_1, v_2) = \frac{v_1 \cdot v_2}{\|v_1\|\|v_2\|}.$$

In `numpy`, you can use `numpy.dot` for the dot product and `numpy.linalg.norm` for the norm.

You can introduce a minimum similarity threshold, below which no match will be found.

2 Adding Memory to an LLM

LLMs are trained to complete the text they are given as input with a likely continuation (e.g. response to a question). They are stateless: by default, they have no mechanism to remember past inputs. This means that an LLM, by itself, cannot be used for conversations beyond simple question/answer pairs. An LLM is not a chatbot, yet.

The most common solution to this problem is to add an algorithmic wrapper to the LLM that keeps the entire past conversation in memory and passes it over and over to the LLM, In this way, the wrapper successfully simulates that the LLM remembers the conversation. This solution has an important limitation: the maximum input length (also called *context length*, in terms of the number of subword tokens)

allowed by the LLM, which is a hyperparameter fixed at training time. When the length of the conversation reaches this hard limit, there is no other solution but to reduce the size of the past conversation to make place for new inputs and outputs. Multiple strategies exist for doing so.

In the following, we are aussming that you already have a basic implementation of accessing an LLM through an API such as OpenRouter.

2.1 Adding Memory via a Simple Wrapper

Write a wrapper in Python for the LLM of your choice. The wrapper should implement the chatbot memory as described above, based on the maximum input length of the model of your choice. The algorithm should follow the steps below:

1. take the user input and store it in memory;
2. feed this input, together with all past inputs from the same user and the same session, into the LLM;
3. store the LLM's response in memory as well;
4. return the LLM's last response to the user.

Test the behaviour of your wrapper by asking questions such as 'Do you remember my name?'.

2.2 Intelligent Memory Management

Various strategies, of differing levels of intelligence, can be imagined to get around the context length limit:

Throw away older history. Following a FIFO (first-in first-out) model, upon memory being full, the wrapper always throws away the oldest message(s).

Throw away only LLM responses. You could throw away only the LLM's past responses, thus being able to keep more of the user's questions.

Throw away less relevant passages. Based on your own custom measure of relevance, which could be statistical (e.g. based on TF-IDF computed over the entire conversation history), compute a relevance score for each sentence and throw away the least relevant ones.

Summarisation. You could apply text summarisation, a classic NLP task, on long prompts or long LLM responses. While the implementation of a custom text summarisation tool is beyond the scope of this lab, you can run a local language model fine-tuned for summarisation or, even simpler, ask the same LLM, via an appropriate prompt, to summarize the conversation history. The formulation of the summarisation prompt will be crucial to keeping relevant information from the history. Note that the LLM-based method will slow down the conversation due to two calls being made instead of one, especially if summarisation needs to be applied frequently.