IN5550
II: Word Embeddings and Recurrent neural networks

Language Technology Group, IFI, UiO

Deadline: 21:59, March 13 2021

1 Goals

- Learn how to play around with word embeddings, and understand the entire 'embedding space'
- Learn how to integrate pre-trained word embeddings in an arbitrary neural architecture
- Apply recurrent neural architectures (RNNs) to a classification task

2 Recommended Reading

1. **Neural network methods for natural language processing.**, Goldberg, Y., 2017[1]
2. **Speech and Language Processing.** Daniel Jurafsky and James Martin. 3rd edition, 2019. Chapter 6, 'Vector Semantics'[2]
3. `https://radimrehurek.com/gensim/models/word2vec.html`
4. **'When Are Tree Structures Necessary for Deep Learning of Representations?'** Jiwei Li et al 2015[3]
5. **Stanford Sentiment Treebank** [4]
6. `https://pytorch.org`

[1]`https://www.morganclaypool.com/doi/10.2200/S00762ED1V01Y201703HLT037`
[2]`https://web.stanford.edu/~jurafsky/slp3/6.pdf`
[3]`https://www.aclweb.org/anthology/D15-1278/`
[4]`https://nlp.stanford.edu/sentiment/`

3 Introduction

This assignment consists of two distinct 'parts'. The first of these deals with word representations, which you will play around with and write a report on, and include and evaluate in your own neural feed-forward classifier. The second is on recurrent networks, which you will learn how to implement in PyTorch and apply to downstream NLP tasks. We *highly* recommend that you use Saga for development for this assignment, as resources on your own computers are unlikely to be sufficient for training multiple models in time.

Please make sure you read through the entire assignment before you start. Solutions must be submitted via Devilry[5] by **21:59**, on the **13th of March, 2021**. Please upload a single PDF (4-8 pages) with details on your experiments and your answers to the questions in the assignment.

Your (heavily commented) code and data should be available in a separate private Git repository, on UiO's hosted GitHub. Similar to the first assignment - make your repository private but allow IN5550 staff[6] to have access to it. the PDF in Devilry should contain a link to your repository.

If you have any questions, please raise an issue in the IN5550 Git repository[7], email *in5550-help@ifi.uio.no*, or ask on the Mattermost chat. Make sure to take advantage of the group sessions on March 2 and March 9.

4 Operations with word embeddings

The goal of distributional semantic algorithms is to learn meaningful dense representations for words, such that semantically similar words have similar representations (embeddings). In this part of the assignment, you will be required to play around with pre-trained word embeddings and provide a report on what you observe. We recommend using the Gensim library[8], but this is not a hard requirement. We do provide some starter code for Gensim.

4.1 WebVectors

First, familiarise yourself with the *WebVectors* web service[9], maintained by the LTG at IFI. You do not need to download anything; this is a web service that features pre-trained models for English and Norwegian (amongst other languages), and allows you to eg. produce semantic neighbours, examine word analogies, calculate cosine similarities, plot words in 2D space, etc.

[5] `https://devilry.ifi.uio.no/`
[6] `https://github.uio.no/orgs/in5550/people`
[7] `https://github.uio.no/in5550/2021/issues`
[8] `https://github.com/RaRe-Technologies/gensim`
[9] `http://vectors.nlpl.eu/explore/embeddings/en/`

4.2 Working with pre-trained models locally

While web services are a good way to demonstrate a technology, detailed, in-depth analysis will need you to have local access to these models, to play around with. We're going to use the pre-trained English models for Wikipedia and Gigaword. Note that on Saga, these word embedding models can be accessed locally at `/cluster/shared/nlpl/data/vectors/latest`, where each model is indexed by a certain ID; in this case, **200** and **29** respectively. The zip archive (for example, `200.zip`) consists of a readme, a metadata file, and the model itself.

These models can be easily processed by any library designed to work with word embeddings. Try and play around with the provided Gensim code[10]. Note that in order to use Gensim on Saga, you need to load the `nlptools` module:

```
module load NLPL-nlptools/2021.01-gomkl-2019b-Python-3.7.4
```

You can run the script with a path to the embeddings file, as:

```
python play_with_gensim.py /cluster/shared/nlpl/data/vectors/latest/200.zip
```

This will show you some metadata and then load the chosen model (without decompressing the ZIP archive on disk) and invite you to enter a query word. If the word is present in the loaded model, its 10 nearest 'neighbours' in semantic space will be printed, along with their cosine similarities. If you enter several words separated by spaces, the model will attempt to find out which the odd one out is. Note that these models contain vectors for lemmatised words with their Universal Dependencies part-of-speech tags glued to them in the end (for example, `'table_NOUN'`).

Your task in this part of the assignment is to analyse the content words from the first sentence of the abstract to Chen and Manning's paper - *A Fast and Accurate Dependency Parser Using Neural Networks*[11]. Specifically, you are to use the English Wikipedia (ID 200) and Gigaword (ID 29) models to find the nearest semantic neighbours for these words.

1. Modify the provided script (or write your own) so it's able to take in a text file with query words (one word per line) as input; note that we are interested in content words only (you can discard functional words manually or use any off-the-shelf POS-tagger);
2. Create an input file with the (lemmatised and POS-tagged) content words from the first sentence of the abstract;
3. Modify the script so that for each input word, it outputs the 5 nearest semantic associates (with cosine similarities);
4. Save the produced associates and similarities into a file;
5. Describe the differences between the Wikipedia and Gigaword models in the produced associates' lists; what are the possible reasons for these differences?

[10] `https://github.uio.no/in5550/2021/blob/master/obligatories/2/play_with_gensim.py`

[11] `https://www.aclweb.org/anthology/D14-1082/`

5 Document classification with word embeddings

In this obligatory assignment, we will be using a part of the Stanford Sentiment Treebank (SST). You can find this dataset on our Github, at `https://github.uio.no/in5550/2021/tree/master/obligatories/2/SST`. The training portion contains approximately 7,400 English sentences. All sentences are labelled with one of two categories, positive or negative: thus, this is a binary classification task. We will evaluate your model on a held-out test set of 1,700 sentences. This test set is kept hidden until everyone submits their solution.

The dataset file consists of 3 columns. The `label` column contains the sentiment labels, while the `token` column contains the sentences themselves (already tokenised). The `lemmatized` column contains the same sentences, but automatically lemmatised and POS-tagged with the UDPipe tagger. You are free to ignore this last column, but you could also use it in some of your experiments.

In this section, you have to implement a feed-forward neural classifier that uses pretrained word embeddings of your choice (it makes sense to employ pre-trained embeddings from the NLPL Vector repository[12]). We strongly recommend to use Gensim to load external word embeddings. Otherwise, feel free to modify your code from Obligatory 1: the difference will be that now you have to represent words with their corresponding pre-trained vectors from the very beginning. Thus, the first layer of your neural architecture will be the *embedding layer* converting words to their vectors.

When you pass your sentences through this embedding layer, each sentence will yield a variable number of word vectors (depending on the number of words in the sentence). Since in this section you are using a simple feed-forward architecture, this diversity needs to be flattened into 'sentence representations' with a pre-defined shape. Experiment with different methods of composing word representations into fixed-size sentence representations: averaging them, summing them, etc. Consult the lectures and the recommended reading. Report the results on the validation set of your choice (here accuracy is a good enough measure, since the class distribution is well balanced in SST).

Experiment with using raw vs. lemmatised vs. POS-tagged sentences (**make sure your embedding model's vocabulary matches the tokens in your data**). Also note that many word embeddings models were pre-trained on on texts with functional parts of speech filtered out (since it is not even entirely clear whether they have 'meaning' on their own). However, for the particular task of sentiment analysis, some functional words (like negation particles) can actually be extremely important. Thus, you might want to check whether the vocabulary of the embedding model you are using contains words belonging to functional parts of speech. For the models from the NLPL Vector repository, you can check it by looking into the `meta.json` file which you can find inside each ZIP archive. Some examples of the ids of the NLPL models which do offer representations for functional words are 40 and 82.

Another option is to pre-train your own word embedding on a corpus of your

[12]`http://vectors.nlpl.eu/repository/`

choice: for example, you can use the full SignalMedia corpus. It is available on Saga at `/cluster/projects/nn9851k/IN5550/signal_texts_raw.txt.gz`. In this way, you can impose any pre-processing and filtering decision you want, as well as tune the embedding model hyperparameters. An example of training code using Gensim was given at one of the group sessions and can be found at the IN5550 Github repository.

Try to freeze the embedding parameters or allow them to fine-tune on in-domain sentiment data. What happens? Why?

6 Document classification with Recurrent neural networks

The goal in this section is to implement recurrent neural networks (RNNs) in PyTorch, and apply them to the task of text classification with the same SST dataset you used in the previous section. RNNs take into account the order of element in a sequence (words in a sentence), and thus are in theory more powerful than, say, simple averaging of word embeddings. This is even more pronounced in sentiment classification, where word order is of paramount importance (cf. '*it is bad, not good*' vs. '*it is good, not bad*').

As a mandatory minimum, you should experiment with PyTorch's three different kinds of recurrent network: standard RNNs, LSTMs and GRUs. The input to your RNN should be a sequence of pre-trained word embeddings. As in the previous section, you will need to compose them into a single fixed-length sentence representation. But since RNNs have some notion of sequential order, the ways to do this are different. These include, for instance:

- Using the last state of the RNN
- Using the first state of the RNN (if your network is bidirectional)
- Using both, concatenated/added/max-pooled
- Adding/max-pooling every state
- etc

Experiment with at least three of these ways, for three different kinds of recurrent architectures. How do the number of parameters (weights) in these models compare to the number of parameters in the feed-forward neural classifier? How about training time and performance? Report what you see.

NB: you will need to familiarise yourself with the PyTorch's `pack_padded_sequence` and `pad_packed_sequence` methods before you actually begin running things.

Play around with other hyperparameters. You should construct a grid, altering three of these: some examples of hyperparameters you can alter include the number of layers, the size of the representation, whether your model is bidirectional or not, the amount of dropout between layers, frozen or fine-tuned word embeddings, etc.

You are also required to experiment with different pre-trained embedding models. Compare different algorithms - *word2vec*, *fasttext*, *GloVe*. Recall that on Saga, you don't have to download the models from the NLPL Vector repository, they are available locally. Do you see any differences between them? How important is the issue of out-of-vocabulary words? Was it necessary to train your own word embeddings on an external corpus?

Motivate your choices, and describe the effect they had on your results. Report and discuss your results.

7 Conclusion

Your final submission should include:

- a PDF report discussing your results,
- your best RNN model for binary sentiment classification,
- a `predict_on_test.py` script, taking as an input the path to the test dataset and producing a version of the dataset with the existing class labels replaced with the predictions of your model:

  ```
  python3 predict_on_test.py --test <testfile>
  ```
- The resulting file must be named `predictions.tsv` and follow the same format as the test dataset (which in turn is exactly the same as the training set).

We will then evaluate your predictions against the gold labels with our own code. Please make sure you include all necessary files in your submission. If you used a word embedding model not available on Saga, put it in the `/cluster/projects/nn9851k/IN5550/YOUR_USERNAME` directory there.

If your code fails to run, you will lose points for the code section. You're supposed to provide sensible defaults, that load the appropriate model and vectoriser, and assume that they are located in the root directory. Feel free to give it a test run with a friend, before your final submission, and we will discuss this in a group session.

Good luck!