Assignment 2

Group: Anette Fredriksen (anettfre), Anthi Papadopoulou (anthip),
Nick Walker (nichow)

March 2021

Code: https://github.uio.no/anthip/IN9550-IN5550/tree/master/obligatory2
Note: Each sub-folder at GitHub corresponds to the sections here.

1 Operations with word embeddings

We took the first sentence of the article proposed which was
"Almost all current dependency parsers classify based on millions of sparse indicator features."[1]
From this sentence we kept only content words, discarding functional words which do not have any meaning on their own. We then lemmatised and POS-tagged the words resulting in the following output:

dependency_NOUN
parser_NOUN
classify_VERB
sparse_ADJ
indicator_NOUN
feature_NOUN

The *play_with_gensim.py* script was modified to accept a file as an input, and for each word in the file to output the 5 nearest semantic associated along with their cosine similarity score, using the English Wikipedia and the Gigaword models to do so. Table 1 shows these associates and their scores for each model.

Word	**Wikipedia**	**Cosine**	**Gigaword**	**Cosine**
dependency	dependence_NOUN	0.651	dependence_NOUN	0.788
	Dependencies_NOUN	0.518	reliance_NOUN	0.591
	Dependency_NOUN	0.500	addiction_NOUN	0.578
	dependent_ADJ	0.477	dependance_NOUN	0.575
	interdependency_NOUN	0.463	overdependence_NOUN	0.514
parser	parsing_NOUN	0.756		
	parse_VERB	0.714		
	parsing_VERB	0.677	*not present in the model*	
	parse_NOUN	0.673		
	JavaScript_PROPN	0.663		
classify	classified_VERB	0.764	classified_ADJ	0.653
	classify_ADJ	0.748	unclassified_ADJ	0.577
	categorize_VERB	0.720	class_VERB	0.527
	categorise_VERB	0.688	top-secret_ADJ	0.500
	classified_ADJ	0.676	confidential_ADJ	0.493
sparse	sparse_NOUN	0.823	patchy_ADJ	0.531
	sparse_VERB	0.789	spotty_ADJ	0.512
	dense_ADJ	0.655	sparser_ADJ	0.510
	patchy_ADJ	0.616	muted_ADJ	0.494
	sparser_NOUN	0.597	sparsely_ADV	0.481
indicator	indicators_NOUN	0.664	gauge_NOUN	0.684
	predictor_NOUN	0.614	barometer_NOUN	0.668
	correlation_NOUN	0.569	predictor_NOUN	0.541
	determinant_NOUN	0.550	index_NOUN	0.536
	quantify_VERB	0.542	Westpac-Melbourne::Institute_PROPN	0.518
feature	feature_ADJ	0.704	313.222.2260_NUM	0.515
	featur_NOUN	0.603	313.222.2480_NUM	0.506
	feature_VERB	0.582	444-8107_NOUN	0.501
	featuring_NOUN	0.517	cir-digital-diary_NOUN	0.499
	feature_PROPN	0.507	newsfeature-budget-lat-wp_NOUN	0.499

Table 1: Associates and cosine similarity per model

There is a very clear difference between the associates returned by models. The Wikipedia model was trained on 3.5 billion tokens and knows 249.212 different English words in a lemmatized form, while the Gigaword model was trained on 4.8 billion tokens and it knows 297.790 different English words in the same form. Regarding window size, the Wikipedia model has a window size of 3 words to the left and 3 words to the right, while the Gigaword model has 2 words to the left and 2 words to the right. It was also shown that the Wikipedia model performed better than the Gigaword model on the Google Analogy test set, but worse on the Simlex999 resource.[1]

As one can see the Wikipedia associates of all words are derivatives of the input word. This model is derived from an English Wikipedia dump, and in Wikipedia a lot of repetitions of the same word are used in different forms due to the format of a Wikipedia page. On the other hand the associates of the Gigaword model are more varied, since the model is derived from newswire texts. In news articles it is not so common to reuse the same words. On the contrary, writers usually try to enrich the speech with wording that is more

[1]http://vectors.nlpl.eu/explore/embeddings/en/models/

verbose, so it makes sense that the associates show a wider variety. We should also note that Gigaword, despite its larger coverage in words, did not contain the word parser.

2 Document Classification with word embeddings

For this part we used the solution provided for Obligatory 1. We modified the TSV dataset, the model, and the training script for the model so that they can work with embeddings. For the model itself we added a first embedding layer followed by a liner layer of size 128. The dropout rate is 0.2, we used cross entropy for our loss function and Adam optimizer with a learning rate of 0.001. We also implemented early stopping based on validation loss with a patience of 5 epochs. First we tried three variations of composing word representations into fixed-size sentence representations. Mainly we tried:

→ sum
→ mean
→ max

For that we used Pytorch's EmbeddingBag[2] mode options. We used raw data with the model number 40, since it was also trained on raw data. Table 2 shows the training and validation accuracy for each of these modes.

Mode	Training Accuracy	Validation accuracy
mean	0.779	0.757
max	0.702	0.688
sum	0.794	0.758

Table 2: Training and validation accuracy per mode

Seeing how summing yielded the best accuracy scores, we continue the experimentation with this as the default.

Next, we experimented with raw VS lemmatized VS POS tagged words. For the last two, we grabbed the final row of the data and modified it to either keep just the lemma, or both the lemma and the tag. We chose appropriate models, meaning models that were trained with data that had the same format as the one we chose each time, by looking at the meta.json file of each model. To be able to get a fair comparison between the different methods of formatting the sentences, we choose models that used the same corpus and algorithm. Table 3 shows the results of these experiments. To get our test and validation set on a lemmatized format we used nltk WordNetLemmatizer.

[2]https://pytorch.org/docs/stable/generated/torch.nn.EmbeddingBag.html

Type	Model	Training Accuracy	Validation accuracy
Raw	40	0.778	0.780
Lemmas	5	0.611	0.594
Lemmas&POS	29	0.717	0.709

Table 3: Training and validation accuracy per mode

We should mention here that we tried 2 appropriate models for each case. The results shown here are the best performing ones. The raw type is the best preforming in our case, so further experimentation will be done with this as a default.

Finally, we set *freeze* = *False* in our embedding layer to check how this affected the performance of the model, since this way the tensors would get updated in the learning process. This parameter led to a significantly higher training time, but also better performance in less epochs. The results are shown in Table 4.

Freeze	Training Accuracy	Validation accuracy
True	0.778	0.780
False	0.925	0.835

Table 4: Training and validation accuracy with and without freezing

3 Document Classification with Recurrent neural networks

3.1 Different RNN architectures

We then moved on to the final part of the assignment. We experimented with a simple bidirectional stacked RNN, a bidirectional stacked LSTM, and a bidirectional stacked GRU architecture. For all three, the number of layers was 2, with a dropout of 0.2 for the stacked layers. We experimented with two embedding models, but the results here correspond to the 40.zip model. Table 5 shows the result of experimenting with the final state of the model, Table 6 the addition of every state, and Table 7 global max-pooling. We report the training time as well as training and validation accuracy. As before cross entropy and Adam optimizer with a learning rate of 0.001 were utilized, as well as a dropout layer of 0.1 to prevent fluctuations higher p values caused to the training. Note that we used a frozen embedding layer here to have a manageable training time. The influence of a fine tuned embedding layer will be investigated later on.

Architecture	Training time	Training Accuracy	Validation accuracy
Simple RNN	3'	0.765	0.729
LSTM	6'	0.882	0.778
GRUS	4'	0.824	0.800

Table 5: Last state

Architecture	Training time	Training Accuracy	Validation accuracy
Simple RNN	3'	0.882	0.749
LSTM	6'	0.882	0.806
GRUS	5'	0.824	0.808

Table 6: Adding every state

Architecture	Training time	Training Accuracy	Validation accuracy
Simple RNN	2'	0.824	0.785
LSTM	**6'**	**0.824**	**0.811**
GRUS	5'	0.882	0.806

Table 7: Global max-pooling

As we can see training time was not drastically affected by the different experiments. LSTM with global max-pooling had the best validation accuracy, so it was the one chosen to be the default for experimentation on-wards.

A simple RNN performs multiplication of the input and the previous output, which is then passed through a tanh activation function. Moving from that to a GRU, an update gate is introduced whose task is to decide whether the previous state will be passed on to the next cell or not. The mathematical operations here are performed on the same inputs with a new set of weights. Finally, an LSTM has, in addition to the update gate mentioned before, two more gates, a forget and an output gate. As above the operations are performed on the same input, with two new weights introduced. So a GRU is very similar to an LSTM, but it includes less parameters since it does not have an output gate.

Seeing how, as we move from the simplest architecture to the more refined ones, we get more control of inputs per trained weights, it is reasonable that the LSTM performs better here since it provides with the most control.

Comparing FFNNs and RNNs, and to find the number of weights and parameters, we calculated them by summing the parameters after creating the model, using the same number of hyper-parameters.

Number of parameters:

- **FFNN:** 402.737.702
- **RNN:** 403.079.502
- **LSTM:** 404.164.302
- **GRU:** 403.802.702

We found that LSTM had the highest number of parameters and FFNN had the lowest. This difference was expected and also fitting to what we mentioned in previous paragraphs.

3.2 Hyperparameter tuning

We chose to experiment with the number of layers, the learning rate, and the bidirectionality of the model. Namely, we tested 1, 2, and 3 layers, a learning rate of 0.001, 0.01 and 0.1 and whether the model was bidirectional or not. The grid search was done using nested for-loops. Tables 8 and 9 below show the performance of these combinations.

Bidirectional	**Layers**	**Learning rate**	**Accuracy**
True	1	0.0001	0.791
True	2	0.0001	0.806
True	3	0.0001	0.752
True	1	0.001	0.795
True	**2**	**0.001**	**0.811**
True	3	0.001	0.782
True	1	0.01	0.798
True	2	0.01	0.785
True	3	0.01	0.536

Table 8: Bidirectional Grid search combinations and performance

Bidirectional	Layers	Learning rate	Accuracy
False	1	0.0001	0.766
False	2	0.0001	0.757
False	3	0.0001	0.715
False	**1**	**0.001**	**0.808**
False	2	0.001	0.793
False	3	0.001	0.787
False	**1**	**0.01**	**0.808**
False	2	0.01	0.795
False	3	0.01	0.783

Table 9: Non - bidirectional Grid search combinations and performance

Then getting the best model from the tables above, we tested the effect of unfreezing the embedding layer. The difference in performance can be seen in Table 10 below.

Freeze	Training Accuracy	Validation accuracy
True	0.824	0.811
False	0.882	0.825

Table 10: Frozen and unfrozen word embeddings

As expected, allowing the embeddings to be fine-tuned to the task led to better performance, but at the cost of training time increasing drammatically.

3.3 Experimentation with different embedding models

Finally, we experimented with different pre-trained word embedding models, namely 40.zip (word2vec), 78.zip (GloVe) and 10.zip (fasttext). We used this as a guide to choose from. Table 11 is a comparison between them and Table 12 shows the difference in the performance when using each of them. The embedding layer was not allowed to fine-tune when testing these.

Algorithm	Model	Vocab size	Window	Lemmatized
Word2Vec	40.zip	4.027.169	10	False
Fasttext	10.zip	302.815	5	False
GloVe	20.zip	291.392	5	False

Table 11: Comparison of embedding models

Algorithm	**Training Accuracy**	**Validation accuracy**
Word2Vec	0.824	0.811
Fasttext	0.882	0.789
GloVe	0.824	0.797

Table 12: Performance with different word embedding models

We chose models with *"lemmatized": "False"* for consistency. As we can see 40.zip with the largest vocabulary and window size yields the best performance out of the three as expected.

We did not train our own word embeddings. After that we trained our best model on the whole dataset and placed it at */cluster/projects/nn9851k/anthi* as *model_state.pth* to be used with predict_on_test.py.

References

[1] Danqi Chen and Christopher Manning. "A Fast and Accurate Dependency Parser using Neural Networks". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 740–750. DOI: 10.3115/v1/D14-1082. URL: https://www.aclweb.org/anthology/D14-1082.