UiO - V21: IN5550

Fábio Rodrigues Pereira - fabior@uio.no
Per Morten Halvorsen - pmhalvor@uio.no
Eivind Grønlie Guren - eivindgg@ifi.uio.no
UiO GitHub repository: https://github.uio.no/fabior/IN5550/tree/master/Oblig3

April 12th, 2021

Contents

1 Introduction

The purpose of this project was to build a Named-Entity Recognition network using a pretrained, transformer-based language model for Norwegian, namely NorBERT.

1.1 Source code

All code made for this project is written in Python v.3.7, with the network architecture built from PyTorch v1.6.0. All necessary code can be found in the GitHub repository at https://github.uio.no/fabior/IN5550/tree/master/Oblig3.
The repository contains:

- `eval_on_test.py` - Python file provided the instructors to be used to compare prediction results against gold standards
- `outputs/` - Directory containing the collection of tuning results
- `packages/` - Directory containing the collection of Python scripts developed for the project
  - `model.py` - The different model architectural setups tested on this task (3 using `BertForTokenClassification` and 3 using `BertModel`)
  - `preprocess.py` - Python script containing the `OurCONLLUDataset` class which loads and serves the `conllu`-files

  - `studies.py` - Python script containing the some custom study classes for easy hyperparameter comparing (with result-storing)

- `report/` - Directory containing the written report in latex and pdf.
- `test/` - Directory containing the test files for checking each of the architectures work

2 Discussion

2.1 NER with NorBERT

Name Entity Recognition (NER) is the task of identifying and categorizing proper names in text, using BIO tags. These tags label the beginning of each entity `B`, along with the other tokens inside this same entity `I`. The rest of the tokens, serving other grammatical purposes than referring to entities are given the `O` tag. Additionally, each `B` and `I` tag contain the type of entity as a label:

- Person (`PER`),
- Organisation (`ORG`),
- Location (`LOC`),
- Product (`PROD`),
- Event (`EVT`),
- Miscellaneous (`MISC`),
- Derived(`DRV`),
- Geo-political entity (`GPE`).

Furthermore, all `GPE` entities are additionally sub-categorized as being either `ORG` or `LOC`, with the two annotation levels separated by an underscore. For more info on these labels, check out the ltgoslo/norne GitHub repo. The `conll`-files in this repository were reformatted (as a `conllu`-file) and used as the data for this project.
The overall goal of this project was to extract these tags and labels for Norwegian input data, saving the output as a CONLLU-file. While there exists a few different architectural approaches for solving this task, this project used state-of-the-art contextual embeddings to internally represent each sequence. **NorBERT** is the transformer-based language model that was used to generate these contextual embeddings. This Norwegian flavor of BERT contains 6 layers of (Self-)Attention heads as an encoder, plus another 6 (Self-)Attention heads as a decoder. Each layer consists of 3 linear transformations, often called queries, keys, and values, depicted below as $Q, K, \& \ V$ respectively:

$$\text{Output} = softmax(Q(\text{Input}) \times K(\text{Input})) \times V(\text{Input})$$

As the equation above (and Figure 1) show, inputs are linearly transformed into the queries and keys before being matrix multiplied, then fed through a softmax activation, then matrix multiplied again with another linear transformation of the inputs, otherwise called the values. The final output (for a single sequence) is the embeddings for each token in the sequence of size:

$$[\text{sequence length}, \text{embedding dimensions}]$$

The difference between these representations and static embeddings is that each contextual embedding now contains not only information for that singular token, but also a blend of the other embeddings in the sequence. NER tasks should benefit greatly from these contextual representations, since context often helps clarify an entity's correct interpretation.

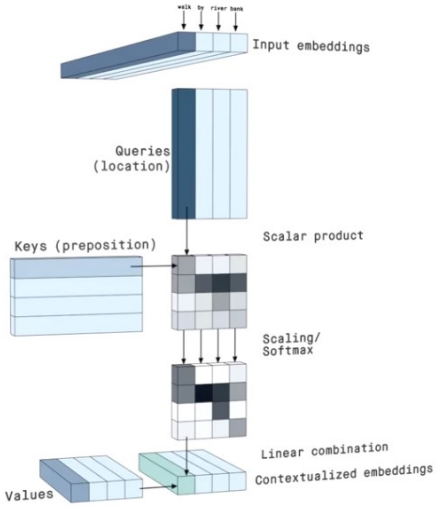

Figure 1: Single layer of BERT (Peltarion on YouTube)

2.1.1 Evaluation versus Loss

In NER, evaluating predictions at *entity*-level rather than token level makes most sense, since a single entity can contain multiple tokens. Evaluating at token level would assume that multi-token entities are more important than single entity tokens, which is not desired. Instead, the provided evaluation script enforces entity-level evaluation by only counting **exact matches** as correct.

However, when training the model, it is still desired to apply loss at *token*-level. This is to make sure the model learns how to capture entire entities, and nothing more.

For most of the architecture optimization, we made the mistake of only looking at the token level F1-score, and completely disregarded the expected evaluation method. This lead to our models giving very high class-wise F1-scores for our optimal models, but producing inferior results on the expected evaluation method. We didn't catch this mistake until very late in development, when there wasn't enough time to refactor the architecture.

2.2 Preprocessing

As mentioned above, the data used in this project was read from a `conllu`-file. Loading in and parsing this file was done with help from the `conllu` library. This tool created a `TokenList` object for each of the sentences in the file. A dictionary with the data for each of the tokens in each sentence could be obtained when iterating through the list. The BIO tags needed for this task were found at `list_item['misc']['name']`.

Desired data For each sentence in the file, i.) the `TokenList` object, ii.) the sentence as a string, and iii.) a list of the labels for each token of the string needed to be accessibly stored. This was achieved using the `load_raw_data()` method found in `preprocess.py`, which created a pandas dataframe object with columns TokenList, sentence, labels. This allowed for easy train-test-splitting with help from `sklearn.model_selection`.

Filtering entity labels The datafile contained over 18,000 examples of sentences with BIO, whereof only half had examples of entities (i.e. `B` or `I` tags). A filter method was created to eliminate these unnecessary rows of data from our dataframes, which helped solve memory errors in model development.

Further, after analyzing the training data, we observed that a good portion of the correctly predicted labels were less relevant to the task at hand, as they provided little to none information about the actual entities we were interested in. To combat this, we filtered the model to only use 'difficult' sentences containing at least five entities. This approach reduced the sample space making training and evaluation significantly faster, but it might hurt the models performance by utilizing less training data. Some trials revealed that the performance had not reduced much, from weighted F1-score 95% to 93%. With the significant rise in speed we decided the trade-off was worth it and we kept with this shrunk sample space.

Dataset `OurCONLLUDataset`, found in `packages/preprocess.py`, converted the dataframes into lists for each of the 3 desired columns. Each item was stored at the same index, making retrieval easy though index-slicing.

Tokenizer A tokenizer loaded inside of the `OurCONLLUDataset` objects converted each sentence string to a list of token ids on retrieval. This was a `BertTokenizer` object from the `transformers` library (loaded from the path to the pretrained NorBERT model). BERT tokenization splits larger or unknown words into smaller subwords, meaning that certain words were expanded into multiple token-ids. This, in turn changed the length of the list now representing tokens in the sentence, and further affected the way the model calculated loss, since now a single BIO tag would refer to multiple subwords.

Masking The first attempt to fix this was to create a label expansion method to map each label to all of its corresponding subword ids. This solved the problem of differing shapes, but still negatively affected the way loss was calculated. (The problem with loss here is similar to the evaluation problem mentioned earlier, where multiple tokens referred to a single entity; except now words that expand to many subwords would make an entity have many more `B` or `I` tokens that it should). To fix this, an attention mask needed to be built for each element of the sequence being retrieved. This attention mask would tell the BERT model which embeddings to pay attention to (i.e. the first subword) and which to ignore (the rest of the subwords) when looking at these types of tokens.

Retrieval Indexing a `OurCONLLUDataset` object would return

(i) The list of token ids for that sentence.

(ii) The list of labels for each of the token ids (expanded to match size of ids).

(iii) The attention mask for the sentence.

2.3 Sequence tagging with NorBERT

The `transformers` library was again used to load the NorBERT model, the transformer-based language model used to generate the context embeddings. This API provided 2 (relevant) ways of loading in this pretrained model:

1. `BertForTokenClassification`: where the model with a final linear output layer is loaded (output size configurable)
2. `BertModel`: where only the model is loaded (output size fixed at 768)

Benchmark The `BertForTokenClassification` was a natural starting point, since the only thing that needed to be done was state how many outputs were needed (i.e. number of labels). The models incorperating this method are the `Transformer` models found in `model.py`.
Our benchmark refers to the out-of-the-box `BertForTokenClassification` model built for this assignment. Even though this is the most straight forward of the networks implemented here, this setup still had a few configurable attributes. For example, freezing the right parameters from

being updated during training could lower compute time and protect against potential unlearning or overfitting. Also, the optimizer and loss functions used to train the model could be chosen as desired. Some of these options were explored in `studies.py` and will be discussed briefly below.

Freezing The point here is to find those right parameters that could be frozen in order to save time and memory without sacrificing performance. Every parameter that requires a gradient will be updated in every optimization step. The fewer computations a model needs for each epoch, the better.
The assignment text mentioned *chain-thaw* freezing, which refers to an initial freezing of all the parameters, then a sequential, gradual unfreezing of each, measuring the performance for each step. Another freezing technique mentioned was *discriminative*, which refers to selecting only specific parameters to freeze.
A freezer study was made (found at `studies.py`) to test a variation (combination?) of these methods. However, optimal performance was achieved with all parameters unfrozen. Since time/resources were not a being taken into consideration for the final evaluation, this fulled thawed setup was considered best.

Transfer learning → RNN This approach aims to use the knowledge acquired from the optimal BertForTokenClassification and transfer it to a new GRU layer. We stack the 12 hidden_state outputs from the former model and plug it in the initial hidden_state in the latter model. Additionally, the output embeds and linear layer from the former model are utilized in the later model. It is relevant to mention that the former model has been optimized with sentences containing at least five entities (2.2). Now, we optimize this current approach with unseen sentences containing 2 to 4 entities, bringing more information to the training steps. Our class method TransformerRNN in `pacakges/model.py` does this work accordingly, and the results are as follows:

Batch #	Train Loss	Test Loss	Weighted F1
1	2.7323827743530273	2.7281882762908936	0.8923716053100798
2	2.6962664127349854	2.6997523307800293	0.9019921281015857
3	2.6254045963287354	2.661438226699829	0.9046540024860147
4	2.614271640777588	2.619636058807373	0.908342173440435
5	2.4960508346557617	2.5614938735961914	0.9119911905909777

Table 1: Some scores for TransformerRNN model in the 1st epoch iteration.

The results in table 1 confirm that the model starts its predictions already with a high weighted f1-score, indicating that the knowledge has been transferred. Besides, the model is able to learn and converges with weighted f1-score around 91%.

Non-initialized RNN The `BertModel` class was used when training on non-initialized final output layers, loading NorBERT directly form pretrained each time. This was to see if output layer's setup would strongly affect the way the model was tuned. The results achieved were quite similar to those from above.
Setups where the optimizer updated the gradients of parameters from both the RNN and BertModel were tested against those where the optimizer only updated the RNN's parameters. The results from this study is shown in heat ...
It was also possible to fine tune the architecture of the RNN, including hidden layers, hidden size, learning rate, momentum. Studies exploring these can be found in lines 260-717 of `studies.py`.

Non-initialized MLP This same intuition was applied to the MLP set up as well. Again, both fine tuned BertModels + MLP was checked against MLP alone, along with some parameters of MLP networks tested in previous experiments: activation function (both hidden and output), number of hidden layers, units, learning rate and optimizer. The code for these studies can be found on lines 719-1207 in `studies.py`, again.

3 Conclusion

Each of the 4 architectures developed were optimized, then fine tuned, and eventually compared. This was achieved by individually training, then saving the models in the shared folder `/cluster/projects/nn9851k/IN5550/[our user names]`.
For the final evaluation, the `predict_on_test.py` was called using a synthesized version of the original data, with 2000 examples, and each of the saved models. This file took in arguments for the test conllu-file path, and saved the predicted outputs from our model. It was at this step we realized our models had been optimized based on the wrong metrics. While validation scores throughout development showed promising improvements after tuning, F1-scores on this final evaluation never reached over 60%.
At the top of `predict_on_test`, the final selection of models can be found, where our top choice is Transformer because got the highest overal score from eval_on_test.py, as follows:

Architectures	Weighted F1-score	Overal score from eval_on_test
Transformer	0.93	0.5780
TransformerRNN	0.91	0.5686
BERT_RNN	0.94	0.07
BERT_MLP	0.90	0.03

Table 2: Scores for each of our optimal architectures.

3.1 Future optimization

As mentioned earlier, the fine-tuning experiments in our assignment was conducted under the assumption that the model would be evaluated at the token level. Thus, our model performs significantly better with this evaluation method. Given this unfortunate mistake there is of course a lot of optimization yet to be done on our model. However, with the remaining time of the assignment from this realization there was not enough time to re-train all the models using this architecture. Future optimization for this model would include

- More in depth fine-tuning of model architectures
- Exploring different loss functions that better handle entity-level evaluation
- Re-training all the models with these changes

4 Bibliography

[1] T. Hastie, R. Tibshirani, J. Friedman. *The Elements of Statistical Learning data mining, inference, and prediction*; 2nd edition; Springer; New York, USA. 2017.

[2] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*; O'Reilly Media; Sebastopol, CA. 2017.

[3] Murhaf Fares, Andrei Kutuzov, Stephan Oepen, Erik Velldal. *NLPL word embedding repository*; Language Technology Group, University of Oslo; Oslo, Norway 2017

[4] Devlin, Jacob and Chang, Ming-Wei and Lee, Kenton and Toutanova, Kristina ***BERT**: Pre-training of Deep Bidirectional Transformers for Language*; Association for Computational Linguistics; Minneapolis, Minnesota, USA. 2019.

[5] Kutuzov, A. et. al. ***NorBERT**: Bidirectional Encoder Representations from Transformers for Norwegian*; Language Technology Group; University of Oslo, Norway. 2019.