Techniques et applications du traitement automatique de la langue (TALN/NLP)
IFT-7022

TP3 - Project: SGNN-Transformer Sentence Model trained by the paragraph-skip-gram-like SimilarityBXENT

Coursework handed-in to:
Luc Lamontagne

By:
Guillaume Chevalier
guillaume.chevalier.2@ulaval.ca
111 131 607
GUCHE29

Département d'informatique et de génie logiciel
Université Laval
14 Janvier 2018

TP3 - Project: SGNN-Transformer Sentence Model trained by the paragraph-skip-gram-like SimilarityBXENT

Table of Contents

Introduction

The work hereby has been realized in solo, alone. It is mainly an experimentation.

First, a paper is reproduced (Self-Governing Neural Networks for On-Device Short Text Classification), but with a few twists so as to create something new. The purpose of this Self-Governing Neural Networks (SGNN) is to replace the embedding layer completely by a vector projection that derives from a featurization of the words and which is refined through fully-connected layers.

Second, an advanced experimentation is made in an attempt to train a sentence model in a completely novel way by reusing the code from the first experimentation and by comparing sentences to nearby sentences and to negative samples far from the sentence like how word2vec is trained.

Although the novel approach to training such a neural network (and without using embeddings) is very interesting, the model wasn't trained enough (only 2 days on a 1080 Ti card, 32 CPU cores and 32 GB of RAM). In other words : it gave approximately 70% accuracy on the sentiment classification task of the TP2 (more on TP2 later) when the sentence model was taken as a feature extractor for a logistic classifier (those accuracy-evaluation files aren't committed). Moreover, 2 days means probably only 1 day in GPU-time, since the data loading is a blocking operation as the data wasn't prefetched in parallel while training a batch. The first part of the project on implementing SGNNs is however at least considered successful.

The code for the first part is available at the following link, all of the code can be seen from reading the README Markdown document which is itself an exported Notebook :
https://github.com/guillaume-chevalier/SGNN-Self-Governing-Neural-Networks-Projection-Layer

Moreover, some online discussion is made about the first part on reproducing the paper:

1. https://stackoverflow.com/questions/53876974/how-is-hashing-implemented-in-sgnn-self-governing-neural-networks
2. https://github.com/guillaume-chevalier/SGNN-Self-Governing-Neural-Networks-Projection-Layer/issues/1

The content and discussions directly at the links above are to be considered a part of the report hereby and subject to evaluation. Especially, the discussion on the GitHub Issue #1 of the SGNN repo is quite interesting and even generates ideas.

For the second part of the project, the code is available at the following link:
https://github.com/guillaume-chevalier/SGNN-Transformer-Sentence-Model-SimilarityBXENT

It is recommended to read notebooks before reading the code located in **"src/"**. The notebooks have been exported to the README, so reading the README is the same as reading the notebooks and the correspondingly named ".py" files.

Also, the code used for converting wikipedia's wikicode/markupcode to raw text (.txt) is written from scratch (for the lack of non-copyleft python implementations after research), and is available here:
https://github.com/guillaume-chevalier/Wikipedia-XML-Markup-Code-to-Plain-Text-Parser-of-Hell

Part 1 : Self-Governing Neural Networks (SGNNs)

So as to summarize, the modified SGNN pipeline coded hereby works as such:

1. Sentence Segmentation and Word Segmentation is performed.
2. Word featurization is performed using a sklearn TF (CountVectorizer) with a char-level analyzer of n-grams (char n-gram word features). Word beginning and word end markers are added as a special character for the char n-gram word features to be more pertinent.
3. A projection of those features are made to a 1120-dimensional space (vector) using Random Projection, a Locally-Sensitive Hashing algorithm that somehow preserves the cosine similarity of the projections. The sklearn class for this is SparseRandomProjection. There are 80 instances of SparseRandomProjection in parallel with a feature union, each generating 14 features, yielding a 80*14=1120 feature per word (1120-dimensional word vectors).
4. Those word projections can now replace embeddings in sentence models, which is performed later in the part 2 of the project : coding a sentence model.

The main differences with the real SGNN are as follow :

- Here, char n-grams are used for word features, whereas apparently “skip-grams” are used to featurize words in the original paper. This remains a mystery, the StackOverflow question has 3 stars and 0 answers, and the discussion in the Issue of the SGNN code implementation (handed-in hereby) yields no clear answers for what regards word featurization.
- The SparseRandomProjection function used here doesn’t yield binary word features. Here, instead, it yield quantized integers remapped by a ~=1 scalar constant that makes them float features. Overall, it’s quite similar, there is probably and in practice just a bit more than twice the usual quantity of information in the 1120-dimensional projections used hereby.
- Also, in the SGNN paper (and/or its precursor by the same author: the “ProjectionNet”), a few trainable fully-connected layers are added at the end of the SGNN to refine the word projections to learn to remap specific words to a higher level of abstraction (not just word char n-gram hashed/subsampled features). The training procedure here is : none. The training of those layers will however and at least be implemented in the part 2 of the present project, trained jointly (end to end) with the sentence model. Therefore, the training part is ignored here and will be discussed in part 2 below.

Also, between part 1 and part 2, an important change was made. Instead of using a FeatureUnion over T=80 random hashers of d=14 dimensions (80*14=1120 word features), only one random hasher (1*1120) was finally used, which resulted in a dramatic speedup with the scikit-learn implementation. Thanks to Sava Kalbachou for the hint that 1 random hasher could be used as well, which was later discussed in the Issue #1 of the SGNN reproduction code on GitHub.

Word-Level Feature Extraction with character n-grams

The n-gram range is from 1 to 3 on characters. For lack of an optimized implementation that would use tuples of (feature_idx, feature_value) for the generated n-grams sent to the LSH, the following function is used for sending to the pipeline containing the CountVectorizer :

```
def generate_a_few_char_n_grams():
    all_possible_0_to_3_grams = [" "]
    less_printable = list(
        set(string.printable) - set(string.digits) - {" "}
    )
    for a in less_printable:
        all_possible_0_to_3_grams.append(a)
        for b in less_printable:
            all_possible_0_to_3_grams.append(a + b)
            for c in string.ascii_lowercase:
                all_possible_0_to_3_grams.append(a + c + b)
    for a in string.digits:
        all_possible_0_to_3_grams.append(a)
        for b in string.digits:
            all_possible_0_to_3_grams.append(a + b)
    return " ".join(all_possible_0_to_3_grams)
```

This yields approximately 20'000 to 30'000 features per word. With all characters, this would be in the order of millions (M). It is to be clearly noted that more features shouldn't make the program slower if the data structure for the features were a (feature_idx, feature_value) tuple instead of a matrix as implemented in scikit-learn. Because of the implementation of scikit-learn, the restriction to 30k features here was good for speed, more than that would be too much currently.

Locality-Sensitive Hashing (LSH): Random Hashing

Put as simply as possible: normally, Random Hashing consists of drawing "d" hyperplanes in the feature space to encode features with a 0 or a 1 whether they sit above or below the plane, such as to have a d-dimensional binary feature vector after hashing. It is technically a hashing function that maximizes collisions. It therefore preserves cosine similarity, but with losses.

The SparseRandomProjection implementation from scikit-learn here is a bit different. First, it yields quantized floats (a.k.a. integers multiplied by a constant) instead of booleans. Also, the current implementation requires pre-building the sparse feature matrix (CSR-like), which could be completely avoided if using a tuple of (feature_idx, feature_value) instead of a matrix as implemented in scikit-learn. In other words, the random hashing could be computed all on-the-fly simply from a seed, given the tuples of (feature_idx, feature_value), without having to compute a matrix (CSR-like), as stated in the original SGNN paper. The scikit-learn was simple to use so no extra effort were made

here for optimizing the thing. Done inside the GPU, the word featurization + lsh could be performed extremely faster.

Overall, using the proper data structure would have caused the feature extraction (char n-gram) step and the LSH step not to require being fitted. They could be completely dynamic without requiring a fitting step (only a transform step).

Part 2 : SGNN-Transformer Sentence Model trained by Skip-Gram SimilarityBXENT

The SGNN projection layer coded in part 1 replaces the embedding layer in the encode (no decoder) of the Attention Is All You Need's Transformer Network. At the encoding output, the transformer network has a max pooling operation which reduces information across the word dimension (a.k.a. the time-step-like dimension if it was an RNN instead of a transformer for example).

At the end of the network, sentence representations are obtained. One flaw here is that they are a bottleneck because they are of the same dimensionality of the word representations themselves: no dimension-augmenting linear was taken before the max pooling. Overall, the model trained has 512 dimensions per word, and once reduced, represent each sentences with 512 dimensions. It is estimated that 2000 instead of 512 would perform better at the sentence level. This wasn't coded for the lack of time this being a small holidays project / course final project.

The data

The whole is trained on Wikipedia EN. For the lack of finding a simple to use or a non-GPL python implementation of wikicode (wikipedia's markdown-like code) converter to raw text (.txt), new code is written with dirty and imperfect regexes. The code is available here: https://github.com/guillaume-chevalier/Wikipedia-XML-Markup-Code-to-Plain-Text-Parser-of-Hell

The code parser/converter is almost perfect, but has a few flaws that are detailed more in the GitHub issues, mostly, some xml-like "<tags>" aren't removed.

The post-SGNN projection layer

Sentences are converted to a list of word projections with the SGNN pipeline of the part 1.

A fully-connected layer converts the 1120 dimensions of the SGNN to d=512 dimensions for the following AIAYN encoder model and replaces word embeddings. The word representations now comes from the projections + fully-connected layer.

The Transformer Network: Attention Is All You Need (AIAYN)

The code for this part is extracted directly from the Annotated Transformer :

- Article : http://nlp.seas.harvard.edu/2018/04/03/attention.html
- Repo : https://github.com/harvardnlp/annotated-transformer

Compared to the original code in the repository linked just-above, the present code sees its dropout completely removed to make training faster (in terms of epochs spent before hoping to see convergence).

SimilarityBXENT: a similarity matrix loss trained à la skip-gram

Simply put, the **SimilarityBXENT** is simply a paragraph-level skip-gram with grouped negative sampling, where sentences are normalized, and all compared to each other with cosine similarity, which is optimized to match the **block diagonal matrix** where each block is the sentences of a paragraphs. The sentences of a block are trained to be similar to each other, and dissimilar to other paragraphs's sentences, thus imitating the negative sampling of a word2vec training, without embeddings nor softmax but rather at sentence level with a similarity matrix and binary cross-entropy.

The transformer outputs a tensor of shape **(batch_size, 512)**, where 512 is the featurized sentence.

This tensor is sent to what is hereby coined a SimilarityBXENT (or in the present code, a "Trainer model"). First, the **sentence features are normalized (not batch normalization nor layer normalization) each to a norm (radius) of 1.0**. Then, they are each compared to each other with cosine similarity, which yields a similarity matrix of shape **(batch_size, batch_size)**, which is symmetric and which's diagonal contains ones (1).

The loss function and the training method is inspired from word2vec's skip-gram training method with negative sampling, and from John Rupert Firth's famous quote:

> *You shall know a word by the company it keeps (Firth, J. R. 1957:11)*

However, here, we are at sentence-level. Which means we train the network to say that sentences in the same paragraphs are similar, and that sentences not in the same paragraph are different from the

ones of another paragraph. Here the split is really performed on Wikipedia paragraphs (with a max sentence count) rather than a predefined window of let's say K nearby sentences. Also, instead of sampling paragraphs with negative 1-v.s.-all, here there is a N-v.s.-N training setup. In practice, this makes that the expected predictions be a diagonal block matrix. For example, if we feed two paragraphs to the network and that the first contains 2 sentences and the second 1 sentence, the expected **block diagonal matrix** will be like this:

1	1	0
1	1	0
0	0	1

And the predicted matrix, of shape **(batch_size, batch_size)**, could happen to be like this for example, and if we scale it from "-1 to 1" to "0 to 1" (add 1, divide by 2):

1.0	0.6	0.2
0.6	1.0	0.3
0.2	0.3	1.0

Once this matrix is obtained, it is **compared** to the expected block diagonal matrix and optimized with a binary cross-entropy loss (xent).

As discussed in the present SGNN reproduction repo's issue 1 on GitHub, the current bxent loss function may be better than the original cosine proximity, but a better loss function might exist, such as the Von Misses-Fisher Loss: https://arxiv.org/abs/1812.04616

More info on cosine proximity can also be found here:
https://isaacchanghau.github.io/post/loss_functions/

Results and Discussion

Unfortunately, the whole thing was trained for only 2 days by a lack of time. The best accuracy on the data+task of sentiment classification (part 1) of the TP2 was of approximately 70%. Here is the code to the TP2 :
https://github.com/guillaume-chevalier/Sentiment-Classification-and-Language-Detection

To obtain 70%, the sentence representations hereby obtained were averaged throughout paragraphs of book reviews so as to have still only 1 representation of the same size per review. Then, the whole thing went through a LogisticClassifier with near-default hyperparameters (C=1e3). The same train-test split than during the TP2 was used.

For the very least, the SGNN projection layer implementation with a twist is considered a success, although it could be greatly improved with more word features and an optimized custom implementation.

Conclusion (and Ideas)

To conclude, the implementation of the SGNN projection layer (part 1) could be more resource-efficient which would allow capturing more characters and wider n-gram ranges. At least, it works and is implemented as a scikit-learn pipeline. The SGNN-Transformer Sentence Model with SimilarityBXENT (part 2) somehow works, achieving approximately 70% accuracy on the TP2's sentiment classification (part 1) task while being trained only for 2 days at 50% GPU utilization (half the time is loading the data and preprocessing with the SGNN projection layer, which is not prefetched+computed in parallel but which could). No hyperparameter tuning has been performed yet.

As a suggestion, it may accelerate learning to have multiple loss heads. For example, the post-SGNN fully-connected layer's output could be optimized with another SimilarityBXENT head to each other words. Thus, the model could have both a word-level SimilarityBXENT head and a sentence-level SimilarityBXENT head, which would be trivial to implement, and which would only add 1 hyperparameter : a ratio of importance of one loss compared to the other in the total loss to adjust the gradient strengths of one loss compared to the other. There could also be a third head mixed with those and that would also add to the total loss, which would be the loss used for training recent models such as BERT : trying to predict unknown words before taking the max pool. The ratio of importance between each loss could be faded from one to another like a variant of curriculum learning where low-level neural layers' losses are more important to train at the beginning, and then high-level layers' losses gets more important over time, and lower level layer losses fade out.

Another improvements would be to have a way to have bigger sentence representations than word representations when the max pooling is performed. This could be done by splitting the residual layers of the AIAYN encoder in two (2x3 multi-head self-attention encoder layers instead of 1x6), and concatenating the 2 resulting "checkpoints" à la "DenseNet" (Densely Connected Convolutional Networks, CVPR 2017), which would yield 1024 (2x512) features to pool instead of only 512.

The Von Misses-Fisher Loss would also be interesting to explore more and to compare it to the newly introduced SimilarityBXENT.

With the suggested changes and with tuning the hyperparameters after running at least multiple runs, it is estimated that letting the whole thing train for a full month and with more GPUs may yield near-SOTA or SOTA results.

Acknowledgements

Thanks to Luc Lamontagne (http://www2.ift.ulaval.ca/~lamontagne/lamontagne-en.html) for being so permissive with the hand-in date of this coursework project and for letting the freedom to students of choosing the subject to do things on their own that they like. Every professor (or lecturer, etc.) should let students explore subjects on their own if they want as such. This freedom of choice contributes so much more to society by letting students be creative and to actually do special projects rather than enslaving them into repeating the same work everyone already did previously.

Thanks to Francis Couture (https://github.com/brucelightyear) who supported me in doing this project and who provided me access to GPUs for free. Thank you Francis.

Also thanks to Sava Kalbachou (https://github.com/thinline72) for the informative discussion on the projection layer of the SGNN and for pointing out to pertinent resources.

License

References

Travis E, Oliphant. A guide to NumPy, USA: Trelgol Publishing, (2006).
https://www.scipy.org/citing.html

Hunter, J. D., Matplotlib: A 2D graphics environment, (2007).
https://matplotlib.org/1.2.1/index.html

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, Édouard Duchesnay. Scikit-learn: Machine Learning in Python, Journal of Machine Learning Research, 12, 2825-2830 (2011).
http://jmlr.org/papers/v12/pedregosa11a.html

Luc Lamontagne et al., IFT-7022 Techniques et applications du traitement automatique de la langue (TALN): cours, énoncé de travail pratique 2, et données publiques (2018 ou avant).
http://www2.ift.ulaval.ca/~lamontagne/

Sujith Ravi and Zornitsa Kozareva, Self-Governing Neural Networks for On-Device Short Text Classification, (2018).
https://aclweb.org/anthology/D18-1105

Sujith Ravi, ProjectionNet: Learning Efficient On-Device Deep Networks Using Neural Projections, (2017).
https://arxiv.org/abs/1708.00630

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, Attention Is All You Need, (2017).
https://arxiv.org/abs/1706.03762

Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean, Efficient Estimation of Word Representations in Vector Space (word2vec), (2013)
https://arxiv.org/abs/1301.3781

Wikipedia contributors, Wikipedia:Database download, Wikipedia, The Free Encyclopedia., (2019).
https://en.wikipedia.org/w/index.php?title=Wikipedia:Database_download#Where_do_I_get_it?&oldid=877639285

Alexander Rush, Harvard NLP, The Annotated Transformer, (2018).
http://nlp.seas.harvard.edu/2018/04/03/attention.html

Sachin Kumar, Yulia Tsvetkov, Von Mises-Fisher Loss for Training Sequence to Sequence Models with Continuous Outputs, (2018).
https://arxiv.org/pdf/1812.04616.pdf

Gao Huang, Zhuang Liu, Laurens van der Maaten, Kilian Q. Weinberger, CVPR 2017, Densely Connected Convolutional Networks, (2017).
https://arxiv.org/abs/1608.06993