Challenging the "One Single Vector per Token" Assumption

Anonymous ACL submission

Abstract

In this paper we question the almost universal assumption that in neural networks each token should be represented by a single vector. In fact, it is so natural to use one vector per word that most people do not even consider it an assumption of their various models. Via a series of experiments on dependency parsing, in which we let each token in a sentence be represented by a sequence of vectors, we show that the "one single vector per token" assumption might be too strong for recurrent neural networks. Indeed, biaffine parsers seem to work better when their encoder accesses its input's tokens' representations in several time steps rather than all at once. This seems to indicate that having only one occasion to look at a token through its vector is too strong a constraint for recurrent neural networks and calls for further studies on the way dense representations such as words embeddings are fed to our models.

1 Introduction

Embeddings have become ubiquitous in natural language processing. However, the overwhelming majority of works that use them, use a single vector to represent each token (word or character) in a sequence. While this is a very strong assumption, it is hardly ever presented as such, namely, that is *just* an assumption and that their could be other possibilities to represent input tokens. This is an especially strong assumption when working with recurrent neural networks (RNN) since by the time they have reached a token, it is directly time to more to the next, and thus an RNN encoder only has one chance to extract all the necessary information from the representation of each token.

We make the hypothesis that giving encoders more time (in term of computation steps) to extract the relevant information from token representations is beneficial.

Indeed, while words can easily linger in someone's mind for several minutes and often much longer after having been read or heard, the most frequent flavors of recurrent neural networks only have very limited storage capacity. A Long-Short Term Memory unit (LSTM (Hochreiter and Schmidhuber, 1997)) has two internal vectors that store information, while a Gated Recurrent Unit (GRU (Cho et al., 2014)) only has one such vector. Moreover, their internal machinery is too simplistic to allow actual prefect recording of independent words and thus they have too make the best of the information available in both the input representation and their current hidden states right away.

Furthermore, having a single vector per word[1] prevents their representations from having a temporal structure[2] which could in principle be beneficial to the extraction of information from said word representations by recurrent neural networks.

In this paper, we use dependency parsing as a benchmark to test our hypothesis. We conduct two sets of experiments where we train syntactic parsers whose input words representations are either split in one, two, four or eight vectors. In the first set of experiments, word representations are learned from scratch with the parsing loss, while in the second, word representations are taken from a modern transformer based pre-trained language model.

An increase in parsing scores as the number of vectors used per word increases seems to support our hypothesis.

[1] In this paper we use the terms "word" and "token" quite liberally. Since we test our hypothesis on dependency parsing, a "word" should be understood as an actual word or a punctuation symbol (what is usually called a "token"). When necessary we use the term "word" to make it clear that we are speaking of "parsing tokens" and not of "(sub-word) tokens" of modern transformer based language models. This means that in a different context, a "word" could actually be a character or anything object we want to pass a representation of to an encoder.

[2] By temporal structure we mainly refer to the iterative nature of the computation carried out by recurrent neural networks.

The remaining of this paper is organized as follows. In section 2, we present some works that have tried learning several vectors per words. In section 3, we introduce the idea of stratified vectors and their implications for parsing methods. In section 4, we describe our experimental setting and present the results. In section 5, we discuss some limitations of the present study. In section 6, we draw the main directions for future research on stratified vectors and state a number of questions opened by the results presented in section 4. Eventually, section 7 closes the present work.

2 State of Affairs

To the best of our knowledge, this paper is the first to question the otherwise universal assumption that each token in a sequence should be represented by a single vector. This being said, other researchers have looked at related yet orthogonal problems about word representations.

For example, Huang et al. (2012) proposed a method for learning multiple vectors per word form as a mean to deal with polysemy and homonymy and thus allow words with the same form but different senses not to interfere with each other's representation. Yet, at encoding time, only one embedding from the set of available prototypes is used and thus an encoder still sees a word only once through its chosen vector.

More recently, with the emergence of transformer based language models (Devlin et al., 2019) that use sub-word tokenizer, some words are indeed represented by multiple vectors. However this is not due to an attempt at giving an internal structure to word representations, but rather this is an artifact appearing from the way they handle rare and out-of-vocabulary words. Thus not all words end up being represented by the same number of vectors and one needs to find proper ways to deal with them when applying those language models to tasks such as part-of-speech tagging or dependency parsing where one needs to predict an output for each of the original words (and punctuation symbols) rather than for the tokenized sub-words for which contextualized representations are computed.

In fact, what may actually be the closest to our proposal, not in design, but rather in potential effect is multi-head attention (Vaswani et al., 2017). Indeed, multi-head attention is a way to extract different aspects/views from a single vector. While multi-head attention does not give a temporal structure to word representations (and in fact transformers and attention heads are quite agnostic to time which need to be artificially reintroduce with position embeddings), it can disentangle various relevant aspects of a word, all stored in a single vector, according to a given context.

3 Stratified Vectors

RNN encoders have the opportunity to make the best of the vector they are shown only once. If they do not extract the necessary information the first (and only) time they meet a token, they never have a second chance. This may be especially detrimental for word with a high perplexity given the current encoder's hidden state, since it likely to be harder for the encoder to extract relevant information from a vector that is unexpected from the context.

We thus propose to add an extra dimension to word representations. Instead of learning a single vector per word, we propose to learn a sequence of vectors for each word that will always come together. We hypothesize that it will be useful for three main reasons: (i) it allows different aspects of a word to be disentangled in the representation which can be useful for task where words have different roles such as in dependency parsing where a word can be both a dependent and a governor, (ii) since the vectors always come together, if a word is unexpected in a context, while the first vector will have a high perplexity, the following one should have a much smaller one, and thus first vector could act as a warning mechanism to prepare the encoder to make the best out of the incoming vectors, and (iii) having more computation step to extract useful information should be beneficial.

There are two questions readily appearing when we decide to abandon the "one vector per word" assumption, namely: (i) Should every token have the same number of vectors? (ii) Should these vectors be the same or different?

For this work, we decided to keep the same number of vectors per tokens. Indeed, allowing the number of vectors to vary, even on a per word-class basis, would greatly increase the complexity of the learning process. So we give a positive answer to the first question as a simplifying starting point.

Regarding the second question, from the idea of giving more time to spend on each token to the encoder alone, it could seem natural to simply repeat the same vector several time. However, the

encoder is left in a different state after having seen the first vector of a given token than the state is was in just a computation step before, and so it might in fact be more interesting to have a different second vector in order to reflect this fact. Furthermore, if we want to be able to disentangle different aspects of a word, it might be necessary to have different vectors. But then, we should realize that if instead of a single vector of d dimensions, we allow two or more vectors of d dimensions per token, then the number of parameters of our model also increases, and thus its information storing capacity increases too, not only its computation time. In that case, any increase in accuracy could just as well be due to the increase in storing capacity as in the increase in computation time.

Thus, in order to keep a comparable number of parameters per token, we decided to use k vectors of $\lfloor \frac{d}{k} \rfloor$ dimensions per token. We call these k vectors the strata of a word's representation. In practice, we can use a transposed convolution tensor to turn a vector (a $1 \times d$ matrix) into a $k \times \lfloor \frac{d}{k} \rfloor$ matrix. We thus call the stored vector a stratified vector. Using this new representation, a sentence of t tokens will be represented as sequence of td vectors of $\lfloor \frac{d}{k} \rfloor$ dimensions rather than the usual t vectors of d dimensions. This is depicted in figure 1.

We should note that, since the input vectors are of length $\lfloor \frac{d}{k} \rfloor$ instead of d, assuming the encoder has the same output/hidden dimension in both cases, let us call h the number of hidden dimensions, then the matrix used to feed the input vectors to the encoder is of size $h\lfloor \frac{d}{k} \rfloor < hd$. This means, that every other things being equal, the model based on stratified vectors is slightly smaller than the original one, even though marginally so, since in practice most of the memory will be taken by the representations themselves and in the case of a biaffine parse (Dozat et al., 2017) by the relation label decoder.

Another effect non negligible effect of using stratified vectors is the linear increase in computational time, since it takes k times longer to process a k time longer input. We also expect training to be slightly more difficult since the loss gradient will need to be back-propagated through k times more recurrent cells.

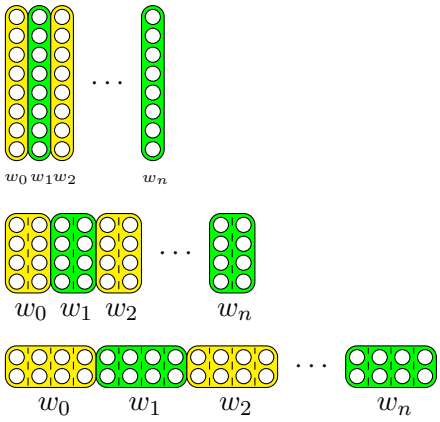

Figure 1: A representation of stratified vectors used to represent a sentence of length n. The top row depicts the traditional way of using word embeddings with a single vector of d dimensions per word. The middle row represents a situation where each word is represented by two vectors of $\lfloor \frac{d}{2} \rfloor$ dimensions. In the bottom row, each word is now represented by four vectors of $\lfloor \frac{d}{4} \rfloor$ dimensions. The dashed lines highlight the fact that even though the different strata of a word are trained together and form a single coherent entity, they are read one by one by the RNN.

3.1 Dependency Parsing

Our task of choice for testing our hypothesis is dependency parsing. Since we use a graph-based parser similar to the biaffine parser of Dozat et al. (2017), each pair of tokens needs to be scored before we can apply a maximum spanning tree algorithm to recover the actual best parse tree. However, since each token in a sentence is now represented by k vectors in the encoded sequence, the typical scoring mechanism of using a biaffine function applied to each pair of encoded vectors would now give k^2 scores per pair of tokens. While many strategies could be used in order to use these k^2 scores, we decided to use a simple max-pooling strategy to only retain a single score per pair of tokens. We do the same for the dependency relation labels.

4 Experiments

We conducted two sets of experiments in order to test our hypothesis. In both cases, we train biaffine style dependency parsers (Dozat et al., 2017). The main difference is the source of the word representations fed to the encoders. In the first case, word embeddings are trained from scratch with the parsing loss, while in the second case, we use a frozen pre-trained transformer model as feature extractor.

4.1 Parsing Architecture

Beside the major difference regarding the source of word representations, both architectures are very similar and revolve around a bidirectional recurrent neural network encoder made of gated recurrent units (GRU (Cho et al., 2014)). The outputs of the encoder, of which there are k for each input token, are then passed through a biaffine layer in order to produce scores for potential dependencies and for relation labels. A final max-pooling layer only keeps the best score from the k^2 computed ones for each pair of word.

During development, we use the argmax function that is very fast to compute to estimate attachment scores and perform model selection. Only at test time, do we use the Chu-Liu-Edmonds spanning tree algorithm (Chu and Liu, 1965; Edmonds, 1967) in order to build actual trees.

Note that since word representation now have a temporal structure, it is not the same to read them from left to right and from right to left, and we could in principle choose the right-to-left RNN to read the sentence in the reverse direction but the word in their original direction. In this work we decided to stick to the traditional way of using a bi-directional RNN, therefore each encoder reads the words in an opposite direction.

4.2 Experimental Setting

The encoder is a bi-directional GRU with a hidden state of 200 dimensions in each direction. Models are trained on the train set of each corpus and after each training epoch the unlabeled attachment score (UAS) and labeled attachment score (LAS) are computed on the development set. We save model states when the UAS or LAS or both increase with respect to the previous maximum scores reached. We stop training when there has not been any improvement for 50 epochs. Models are optimized with the ADAM optimizer (Kingma and Ba, 2014). The code will be released upon publication of this paper.

4.2.1 Embedding Trained with the Model

We perform this set of experiments on French using the GSD corpus and on English using the EWT corpus from the Universal Dependency project (Zeman et al., 2022). For a given language, word forms appearing only once in the training set and forms that appear only in the development set or the test set are replaced by a special <*ink*> token.

In this first set of experiments, words are simply represented using embeddings directly trained alongside the model with the parsing loss. Embeddings of length 120 are either distributed in a single vector of 120 dimensions, two vectors of 60 dimensions, four vectors of 30 dimensions or eight vectors of 15 dimensions each using a transposed convolution layer.

This model has about 10.4 millions parameters when $k = 1$ and the count slightly decreases as k increases. On top of the core parameters, there are 1.9 millions parameters (16096×120) for the French embeddings and 1.2 millions parameters (9665×120) for the English ones. It took 2 days to run the whole set of experiments on a server equiped with a GeForce RTX 3090 graphics card.

Table 1 gives the results for the first set of experiments where word embeddings are trained from scratch with the parsing loss. From French results, it seems clear that distributing a word's vectors over multiple encoding step is beneficial. On average, parsers whose encoder have seen input words' representation in k steps rather than one have higher unlabeled attachment scores for $k \in \{2, 4, 8\}$ and better labeled attachment scores for $k \in \{2, 4\}$. For English, the effect seems less pronounced. However, English parsers still have better attachment scores (unlabeled and labeled) on average when $k \in \{4, 8\}$ than $k = 1$. We also see that when a model does not perform as well when $k > 1$ as when $k = 1$, the scores of the model with $k > 1$ are never far behind from the ones of the model with $k = 1$.

As we noted above, since the k vectors of a word are of length $\lfloor \frac{d}{k} \rfloor$ instead of d, the GRU cell has $h(d - \lfloor \frac{d}{k} \rfloor)$ less parameters, where h is the dimension of the hidden state. Furthermore, having k vectors per word instead of one, means that the input sequence to be encoded is of length kn for an input sentence of length n. Beside an actual increase in computation time, this has two main effects. First, at encoding time, the last and first vectors of two words separated by l words in an input sentence are now kl vectors apart in the new representation and therefore kl computation steps apart, which gives more time for information erasure and thus could make it harder to detect long distance relations. Second, at update time, this means that while the parsing loss is essentially the same as in the "one vector per word" case, its gradient has to be back-propagated through the encoder

Language	Selection Metric	Metric	Average Score / Standard Deviation							
			$k=1$		$k=2$		$k=4$		$k=8$	
French GSD	UAS	UAS	86,24	0,30	**86,58**	0,32	**86,36**	0,34	**86,27**	0,32
	LAS	LAS	80,95	0,47	**81,15**	0,28	**81,02**	0,49	80,54	0,22
	Both	UAS	86,13	0,24	**86,54**	0,29	**86,48**	0,43	**86,24**	0,28
		LAS	80,77	0,41	**81,07**	0,25	**80,99**	0,43	80,56	0,25
English EWT	UAS	UAS	79,12	0,33	79,08	0,25	**79,52**	0,44	**79,20**	0,25
	LAS	LAS	73,69	0,44	73,51	0,54	**74,08**	0,45	**73,84**	0,23
	Both	UAS	79,05	0,45	79,01	0,29	**79,36**	0,22	**79,25**	0,32
		LAS	73,63	0,56	73,54	0,46	73,53	0,35	**73,81**	0,22

Table 1: Results for the parsing experiments on French and English when tokens embeddings are learnt directly from scratch with the parsing loss. Since there are two main metrics used to test parsers : unlabeled attachment score (UAS) and labeled attachment score (LAS), we applied two different epoch selection strategies. We either pick the best model with regard to the desired target metric (UAS for UAS and LAS for LAS) or picked the last model that improved both metrics at once. These different model selections are marked with horizontal lines, thus UAS and LAS scores reported in the "Both" rows are computed from the very same models. In bold are the averages that are higher than the corresponding average when $k = 1$. Each scores is averaged over five different runs with random seeds set from [0, 1, 2, 3, 4].

RNN for k times more computation steps. Yet, we still see an increase in performance in spite of these two potential problems. This indeed seems to support that having multiple occasion to encode a word into the hidden state of an RNN is beneficial.

4.2.2 Pre-trained Representations

In the previous experiments, we trained the word representations from scratch. However, most current works make use of contextualized representations from language models pre-trained on large amounts of data. Thus, in order to see if the above analysis carries on to more recent pre-trained representations, in the second set of experiments, we used the French transformer based language model Flaubert (Le et al., 2020) as a feature extractor and used the output of its final layer as input to our model. When a word is split in several tokens by Flaubert's tokenizer, we average their vectors in order to get a single vector for the original word.

Since, Flaubert is not trained with our stratified vector representation in mind, we learn an extra transposed convolution tensor of size $1 \times 768 \times k \times \lfloor \frac{768}{k} \rfloor$ in order to distribute the original Flaubert's 768 dimensions representation into k vectors of $\lfloor \frac{768}{k} \rfloor$ dimensions per word which will then be fed to the actual parsing model.

This model has between 10.4 and 10.8 millions parameters depending on k and not counting Flaubert's own parameters since we can run it only once and store its outputs. This set of experiments took 12 hours to run on a laptop equiped with a Quadro RTX 4000 graphics card. Extracting Flaubert's representations took roughly 20 minutes and we then serialized them so as to avoid running Flaubert for each new run.

Table 2 presents the results for the second set of experiments where word embeddings are taken from a frozen Flaubert model. In this second set of experiments, we only trained models for $k \in \{1, 2, 4\}$ because the bigger models take more time to train. In this table, it appears even clearer that having more vectors per word is beneficial. The average parsing scores (UAS and LAS) for models with $k = 1$ and $k = 4$ (resp. $k = 1$ and $k = 2$) are now several standard deviations apart, making the case even stronger in favor of using multiple embedding per words.

We remark that the results of parsers using Flaubert's contextual representations are on average lower than those of parsers that learn their own word representations. While it can be surprising, it could easily be explained by a number of factors. The main factor being that the development set and the test set of the French GSD corpus seem to come from different distributions. While the scores computed on the development and test set of the English EWT corpus in the previous experiments are very similar: the average absolute score difference between the development and test is of 0.28 points for the UAS and 0.44 for the LAS with the test scores being both sometimes above and sometimes below the corresponding development scores, the scores computed on the French test set are con-

Language	Selection Metric	Metric	Average Score / Standard Deviation					
			$k=1$		$k=2$		$k=4$	
French GSD	UAS	UAS	83,04	0,49	**84,54**	0,35	**85,14**	0,38
	LAS	LAS	75,33	0,61	**77,45**	0,38	**78,36**	0,36
	Both	UAS	83,05	0,44	**84,54**	0,35	**85,14**	0,38
		LAS	75,50	0,35	**77,32**	0,33	**78,23**	0,34

Table 2: Results for the parsing experiments on French when tokens embeddings are taken from a frozen Flaubert model. Like in the previous experiments, we either pick the best model with regard to the desired target metric (UAS for UAS and LAS for LAS) or picked the last model that improved both metrics at once. These different model selections are marked with horizontal lines, thus UAS and LAS scores reported in the "Both" rows are computed from the very same models. In bold are the averages that are higher than the corresponding average when $k = 1$. Each scores is averaged over five different runs with random seeds set from [0, 1, 2, 3, 4].

sistently below the scores for the development set (2.50 points below for the UAS and 3.10 for the LAS) which seems to indicate a difference in data distribution. The score difference is exacerbated by Flaubert's representations with an average of 4.07 UAS points and 5.32 LAS points difference between the development set and the test set when $k = 4$. Still, for $k = 4$ the average development UAS is 88.76 ± 0.22 when embeddings are trained from scratch and 89.31 ± 0.09 with Flaubert's representations. So it seems that our model can indeed benefit from Flaubert's contextual embeddings and it argues in the direction of a distribution difference.

Thus, both experiments' results support the idea that using stratified vectors is beneficial for RNN as least in the case of dependency parsing.

4.3 Probing the Embedding Spaces

Once we have seen that giving more opportunities to an RNN encoder to extract relevant information from a word is beneficial, we start to wonder what information is encoded in these different vectors and how the different vectors of a word relate to each other.

In this study, we tested whether we could find some general relations between the structure of the different induced embedding spaces. Since the k vectors of a word are always used in the same order in every sentences the word occurs, we wondered if these vectors would store information about their respective position with respect to each other. Our first probe is thus trained to predict the position $i \in [1..k]$ of a vector within its word's representation.

Our first probe is a simple linear layer trained for one epoch on $8000k$ vectors (we include the k vectors of a word at the same time) and tested on $2000k$ vectors. We ran in on French and English representations learnt during the first set of experiments. Since the classification accuracy sticks to the random baseline, it seems that the embeddings do not bare any readily accessible information about their respective position, at least not in a linear way. In fact this is not so surprising since if the vectors were linearly separable a great portion of the different embedding spaces would be rendered unusable which would easily be sub-optimal.

The second test we conducted on the learnt embeddings was to see if there was some high level structural similarities between the different embedding spaces. Given a pair of jointly trained embedding spaces, we sample 500 pairs of words at random and compute the cosine similarity between their respective vectors in the first space and in the second space respectively. If both spaces had similar structure up to rotation and/or reflection, we would expect to see a correlation between the different cosine similarities of pairs of words, however here again, we fail to see any correlation with Pearson correlation coefficients being close to 0.

These two simple probing experiments show that the structure of the stratified embedding spaces is not straightforward. Indeed, it is neither a simple division of the embedding space nor is it a simple geometrical transformation of one stratum into another. But there are still many possibilities to probe a deeper structure and since stratified vectors seems to help, we would expect there to be a structure of some kind.

5 Limitations

This work is limited in two main regards. First, we only tested our hypothesis on dependency parsing. At this point, it is not clear how this result should apply to other linguistic tasks if at all. Since

in dependency parsing a word plays several roles (governor and dependent), it could be that having multiple output vectors helps more here than for other tasks. However, early experiments seems to indicate that only having several output vectors per word is not enough to see similar parsing gains.

The second limitation is clearly the language selection. We only experimented on French and English, which are fairly similar Indo-European languages. While there is nothing inherent about French or English that should make them more likely to disagree with the "one vector per word" assumption, it is still possible that if stratified vectors are particularly useful for dependency parsing and not for other tasks, since French and English have a fairly similar syntax the results presented here would not directly apply to other languages and/or tasks.

However at this point, there is no strong evidence pointing in that direction and we simply need more work to see how these results do or do not generalize.

6 Future Work

These first results open many new avenues for future research and begs for a better understanding of what is actually captured by neural networks and by word embeddings. Here we only present a few of the many questions that will need to be answered.

First and foremost, we need to understand the information structure of stratified vectors. As discussed in section 4.3, the structure of the stratified vector space is not a simple division or transformation of said space. However, there are still plenty of possible structures that would encode information in a non straightforward way. We thus need to answer the following questions: Do the different vectors of a word encode different pieces of information, or is it the repetition that allows the RNN to extract it better? If it indeed is the former, how is that reflected into the different sub-spaces internal structures?

Since the k vectors of a word always come together, we guess that it reduces the overall perplexity of the underlying language model, as the first vector of a word prepares the model for its successors. We hypothesize that the first vector of a word brings the RNN to a state where it is better able to make the best of the subsequent vectors of that very word. So we need to investigate this hypothesis: Is it just the expected reduction in perplexity that makes the model more powerful or is there something else entirely?

Then, as mentioned in section 5, we have only experimented on dependency parsing, and thus we need to know if and how it would transfer to other tasks? Do stratified vectors work only for task where there is a strong role difference between tokens as in dependency parsing (governor vs. dependent)? Related to that question, is the fact that in RNN, more inputs implies more outputs and therefore more encoding space, so we also need to investigate the impact of these added degrees of freedom on the end results.

From a technical standpoint, it is clear that the increase in computation time discussed in section 3 is a major limitation of our proposal. However, this need not be a fatality. If instead of having several vectors for words in isolation, we used compositionally crafted n-gram representations, we could still have information about a given word passed to the encoder for several computation steps while only incurring a additive linear overhead rather than a multiplicative one. For example, instead of representing a sequence abc as $[\mathbf{a}_1, \mathbf{a}_2, \mathbf{b}_1, \mathbf{b}_2, \mathbf{c}_1, \mathbf{c}_2]$ (with $\mathbf{a}_1$ being the first vector for a and so on) which is twice as long as the original sentence, we could represent it as $[\mathbf{f}(\#ab), \mathbf{f}(abc), \mathbf{f}(bc\#)]$ (where $\mathbf{f}$ is some compositional embedding function and $\#$ representing sentence boundaries) which still has every word appearing at least twice and yet has the same length as the original sentence. This needs to be investigated further.

We mentioned in section 4.1 that since stratified vectors have a temporal structure, it is not the same to read them in one direction or the other. This becomes a new parameter for RNN that needs to be understood. Moreover, we introduce stratified vectors in the context of recurrent neural networks, but if it is the multiple outputs that make them powerful then they could also be applied to transformer type architectures, which as we said earlier are time agnostic. This would beg even further research on the information structure of the embedding spaces and their relation to each other.

Eventually, regarding dependency parsing more specifically, there are many possibilities for extracting trees from multiple scores beyond max-pooling. We could always use a single fixed cell and thus let the remaining vectors encode any useful informa-

tion. We could have different biaffine matrices for different cells. We could use the different cells to reconstruct several trees and effectively train several parsers at the same time and then have them vote for example.

As we see, the results presented in this paper open a lot of new questions that will need to be answered if we want to make the best of embedding spaces.

7 Conclusion

In this paper, we have introduced stratified vectors as a way to challenge the ubiquitous "one vector per word" assumption. Via a series of experiments on dependency parsing of French and English, using either representations learnt from scratch or extracted from pre-trained language models, we showed that stratified vectors indeed seem useful, at least in the context of graph based parsing with RNN encoders.

We then discussed the current limited scope of our results and the necessary questions that need to be answered in order to better challenge the "one vector per word" assumption and the many directions for future research granted by these questions.

8 Ethical Considerations

As far as we can tell, this work should not raise any ethical concerns.

The only potential impact, yet very theoretical at this point, is due to the increase in computation time brought by the increased sequences length. But as we mentioned in section 6, it should be possible to reach similar results with a better n-gram based encoding, which would therefore bring our proposal back in line with other RNN based methods in term of computation time.

References

Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, Doha, Qatar. Association for Computational Linguistics.

Y. J. Chu and T. H. Liu. 1965. On the shortest arborescence of a directed graph. *Science Sinica*, 14.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Timothy Dozat, Peng Qi, and Christopher D. Manning. 2017. Stanford's graph-based neural dependency parser at the conll 2017 shared task. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 20–30, Vancouver, Canada. Association for Computational Linguistics.

Jack Edmonds. 1967. Optimum branchings. *Journal Research of the National Bureau of Standards*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9:1735–80.

Eric Huang, Richard Socher, Christopher Manning, and Andrew Ng. 2012. Improving word representations via global context and multiple word prototypes. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 873–882, Jeju Island, Korea. Association for Computational Linguistics.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization.

Hang Le, Loïc Vial, Jibril Frej, Vincent Segonne, Maximin Coavoux, Benjamin Lecouteux, Alexandre Allauzen, Benoît Crabbé, Laurent Besacier, and Didier Schwab. 2020. Flaubert: Unsupervised language model pre-training for french. In *Proceedings of The 12th Language Resources and Evaluation Conference*, pages 2479–2490, Marseille, France. European Language Resources Association.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Daniel Zeman, Joakim Nivre, and al. 2022. Universal dependencies 2.10. LINDAT/CLARIAH-CZ digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.