Embedding compression using trainable codebook
NLP 3-cfu Project Work

Samuele Bortolato
Master's Degree in Artificial Intelligence, University of Bologna
samuele.bortolato@studio.unibo.it

Abstract

Most language models use word embeddings to encode words in a dense space, storing a vector of fixed size for each word in the vocabulary. The size of these embeddings is often a problem when trying to deploy these models on a low memory device. In this work I propose to use a new way to compress the embedding matrix by using multilayer codebook with trainable indices, which can achieves significantly higher SNR with respect to the commonly used SVD while still remaining quite fast to train and retrain, achieving better results both with and without finetuning the embeddings.

1 Introduction

Transformers have achieved state of the art performance in many NLP tasks, but their adoption in less powerfull machine is heavily limited by both their computational cost and memory footprint. For this reason there have been many attempts to reduce the size of these models while still preserving their performance. There are two components that make up the total size of the model: the word embeddings and the actual model architecture. With this work I focus on the compression of the embedding matrix, which for smaller models is non negligible part of the total memory. The word embedding matrix stores a vector for each word of the dictionary, which for normal sized dictionary is a huge matrix. For example the embedding matrix of TinyBERT (Bhargava et al., 2021)(Turc et al., 2019) has 30522 vetors of 128 dimensions, for a total of 3.9M parameters.

Common techniques for compressing the embeddings are using reducing the bit resolution of the embeddings and using a low rank approximation of the matrix.

Reducing the bit resolution of the embeddinges is extremely effective, thanks to neural networks being quite robust to low resolution inputs. The problem with this technique is that retraining or finetuning a compressed model becomes quite complex, requiring in most cases the reconversion to the usual 32 or 16 bit representation, or using a pass through estimator that, needing a second copy of the weights in full resolution, uses even more memory than the original during training. Other techniques that work on the bit representation of the matrix have similar issues being extremely difficult to retrain while compressed.

The most common approach to compress the embeddings is instead the use of a low rank matrix approximation, which, being a linear operation, allows the training of the embeddings even after the compression. The most common technique to compute the factorization is to simply use the SVD approximation of the matrix, for which we have a closed form solution. Even though SVD is quite easy to compute it has the problem that by minimizing the L2 norm it's quite sensitive to the presence of 'outliers' and often misses the important information to focus on the reduction of larger errors. For this reason other works (Tukan et al., 2020) (Balazy et al., 2021) tried to decompose the the matrix using a lower p-norms or different objectives. Computing a p-norm factorization is an NP-hard problem, so often the optimization of the approximation is learned through gradient descent.

Taking inspiration from (Takikawa et al., 2022) I trained a codebook with trainable indices using a straight-through estimator. I further modified the estimator by changing changing the fake derivative from a softmax to a linear combination of the indices, which led both to a computation speedup and a convergence speedup.

To test the technique I chose to use the TinyBERT architecture , which being extremely small in proportion uses the embedding to store information more than bigger models with hundreds of millions of parameters. To test the effectiveness of the compression I first compared the MSE obtained with this technique with those obtained with

SVD. Then I tested the trainability of the system by finetuing the model on the GLUE benchmark (Wang et al., 2018). Then, since the MSE loss for the training can be changed with other losses to minimize other p-norms or other objectives, I tested the performance of the approximation without finetuing the approximation by first training the whole model on the task and later compressing the embeddings. Finally I test a possible approach to decrease the memory requirements for training the approximation.

2 Background

In Variable Bitrate Neural Fields (Takikawa et al., 2022) they present a new way to map 3D coordinates to an higher dimensional space embedding to increase the performance performance for neural fields while reducing the size of the model.

In particular they made a decoder only architecture composed of a pyramid of grids with increasing resolution dividing the space in voxels. To compute the embedding of a point they then linear interpolate the embeddings of the corners of the voxels at each level and concatenate the results before passing them through a small MLP. The fundamental point is that, instead of storing a dense grid of features that would waste a lot of memory for regions where there are not many details, they defined a compressed representation $V \in \mathbb{Z}^m$, in $[0, 2^b - 1]$. Notice that this representation can be stored using only b bits for entry. The decoder function is just an indexing operation, so the optimization problem is

$$\arg\min_{D,V,\theta} \mathbb{E}_{x,y} ||\psi_\theta(x, interp(x, D[V])) - y||$$

Solving this optimization is hard due to the non-differentiable indexing, so they propose to use a straight-through estimator (Bengio et al., 2013) by substituting the indexing function with a softmax during the backward pass. This requires to store a second matrix with a softened version of the indexing $C \in \mathbb{Z}^{m \times 2^b}$ during training.

$$\arg\min_{D,V,\theta} \mathbb{E}_{x,y} ||\psi_\theta(x, interp(x, \sigma(C)D)) - y||$$

3 System description

The framework proposed in Variable Bitrate Neural Fields can be adapted with minor changes to approximate the word embeddings. In this case we don't need to create the grids nor to compute interpolation of points since the starting space (the words of the dictionary) is already quantized. Usual sizes for the tables are 256 entries (8 bits coding) or smaller, causing a huge number of collisions. For this reason even though we don't use the grids is still better to use a multilayer indexing and passing the concatenated results to a small MLP. As highlighted in Instant Neural Graphics Primitives this is important to allow the table to learn a robust representation even though we are not explicitly handling table collisions, effectively allowing the network to distinguish words by indexing a different position for each level.

I also propose to substitute the Softmax function in the backward pass of the straight-through estimator with a one sided estimator that behaves as a linear combination of feature. In particular the derivative for the grid remains the same for the hard indexing, while the derivative for the indices are the scalar product between the derivative of the output and the embedding of the values of that index.

$$forward: \quad Out = D[V] = Hardmax(C) \times D$$

$$fake\ forward\ for\ indices: \quad Out = C \times D$$

where Hardmax is the onehot encoding of the maximum (over the embedding dimension). This can be also seen as an unnormalized cosine similarity between the output and the values of the table. The derivative tries to increase the score of the indices that are more aligned to the direction that decreases the loss of the output. By leaving the derivative of the table the same as the real derivative of the indexing we also push the values only on the desired direction without introducing additional terms to the loss that are not correlated with what we want to minimize (as happens using a Softmax straight through), that changing the loss landscape could lead to a worse minimum for our task.

$$\frac{\partial L}{\partial D} = \frac{\partial L}{\partial O} \times Hardmax(C)$$

$$\frac{\partial L}{\partial C} = \frac{\partial L}{\partial O} \times D$$

This can be easily implemented in pytorch or tensorflow with a function with custom backward pass, and since it just requires an indexing for the derivative of table and a matrix multiplication for the derivative of the indices the implementation results approximately 4-5 times faster than using a softmax straight-through.

It's worth to mention that the formulation as is doesn't prevent the scores for each index to grow extremely big or small, potentially allowing it to get stuck on a certain indices and taking a very long time to swap to another. To solve this issue I propose to use weight decay on the soft indices to keep them in a reasonable range. Even a strong weight decay works well, because during the forward pass we take the argmax of the scores, so we don't really care if an index is the maximum by a big or a small margin. On the other hand a weight decay too big could make the model change index too frequently making the training unstable. In my experiments a weight decay of 0.1 or 1 lead to the fastest convergence.

Let's now consider the memory and computational cost of this technique.

During training we have to store the softened version of the indices in order to be able to train them. This forces us to keep in memory a table of the shape $2^b \times l \times w$ where b is the number of bits used in the final indexing (usually 8 for the higher memory efficiency), l is the number of levels of the table and w is the number of words to be embedded. This table is way bigger not only of the final compressed approximation but also of the original embedding matrix. For example, in the tests later, the biggest approximation I ran used 8 bit indexing, and 32 layers, leading to a total memory consumption during training of approximately 1GB (compared to the 16MB of the original embeddings and 1.5 / 2.7 MB of the compression). The memory consumption can slightly be reduced by using 16 or 8 bit for the soft indices, but still remains a problem to be considered. The computational cost of this technique can be split in two parts: the indexing and the decoding MLP.

The indexing, requiring a matrix multiplication for the backward pass, has roughly $2^b/k$ the cost of using a low rank approximation of rank k. This cost is greatly reduced once the indices are fixed. On the other hand the decoding MLP requires additional computation which varies depending on its size. Since the MLP that we need is quite small we can leverage efficient implementations like in Real-time Neural Radiance Caching for Path Tracing fitting the whole network in the cache of a single GPU core and computing the output with a single kernel call, greatly reducing the training and inference time.

Summarizing, the computational cost and memory requirement after fixing the indices is approximately linear to both the number of levels and the number of channels, while during training the memory requirement for the indices becomes the main factor (in normal settings 1024 times bigger than fixed) so the memory required is roughly linear with the number of levels.

One possible solution to decrease the memory requirement would be to use an approximation for the soft indices while we train them. While this would reduce the memory requirement of the training it both reduces the quality of the overall approximation and slightly increases the computational cost. Using a low matrix approximation of the whole soft indices matrix correlates too much the indices of different words, effectively using only a subset of the table. Experimentally I fount it worked better to still store separate indices for each word. I stored two 16 dimensional vectors for each word for each word and compute the indices as the flattened outer product of the two. This decomposition requires 32 float for word instead of the original 256, obtaining an 8x reduction in memory requirement.

Finally, since the proposed approximator is just a model, it can be trained to minimize any objective function, not only the L2 error. It can for example be traind to minimize other p-norms, like suggested in, or maximize cosine similarity like proposed in.

4 Data

The datasets used to test the framework belong to the Glue bechmark, which while being still quite small is still challenging enough for small models like TinyBERT that I used.

5 Experimental setup and results

5.1 Softmax vs linear straight-through

Firstly I compared using a linear straight-through estimator instead of a softmax one. For this test I used the embeddings from the pretrained TinyBERT available on huggingface. The approximator that I used for this test is composed of an 8 layer table with 256 entries (8 bit indices) and 8 dimension per position, with a 2 hidden layer MLP with 64 neurons each. The optimizer used is Adam, learning rate of 0.01, batch size 2^{14}, trained for 5000 epochs, which in my tests lead to the fastest convergence. The plot of the training losses is showed in figure 1 (bilograrthmic scale).

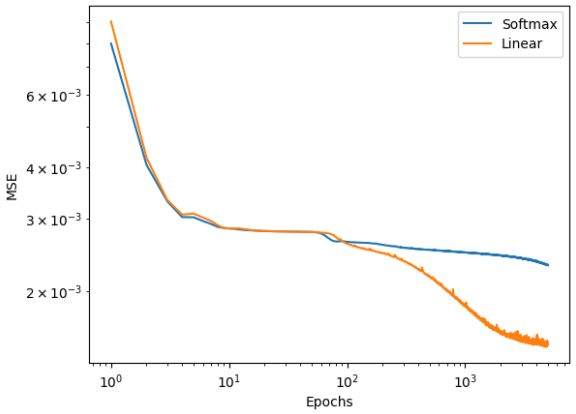


Figure 1: Softmax vs Linear straight-throug

5.2 L2 approximation error

Then I test the MSE obtainable with this approximation compared to SVD for different sizes of the models. The MSE just measures the error compared to the original embedding and is not directly indicative of the performance in real world datasets, instead it's just a measure of how expressive the model is. In the approximation I tested I changed only the number of layers of the table (different colors in the graph) and the number of output channels for each level (different points in the same colored line). I tested two variants of the decorder MLP, the first with 1 layer of 128 hidden neurons and the second with 2 hidden layers of 64 neurons (the output dimension is 128, the size of the original embedding) The layers tested and channels tested are $[2, 4, 8, 16, 32]$. The results are summarized in figure 2. The original size was approximately 16MB.

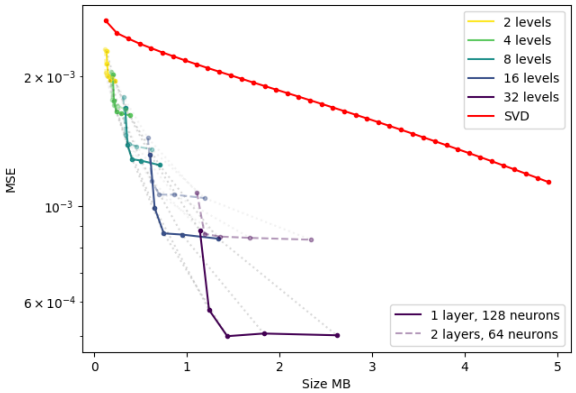


Figure 2: MSE and size for different approximations

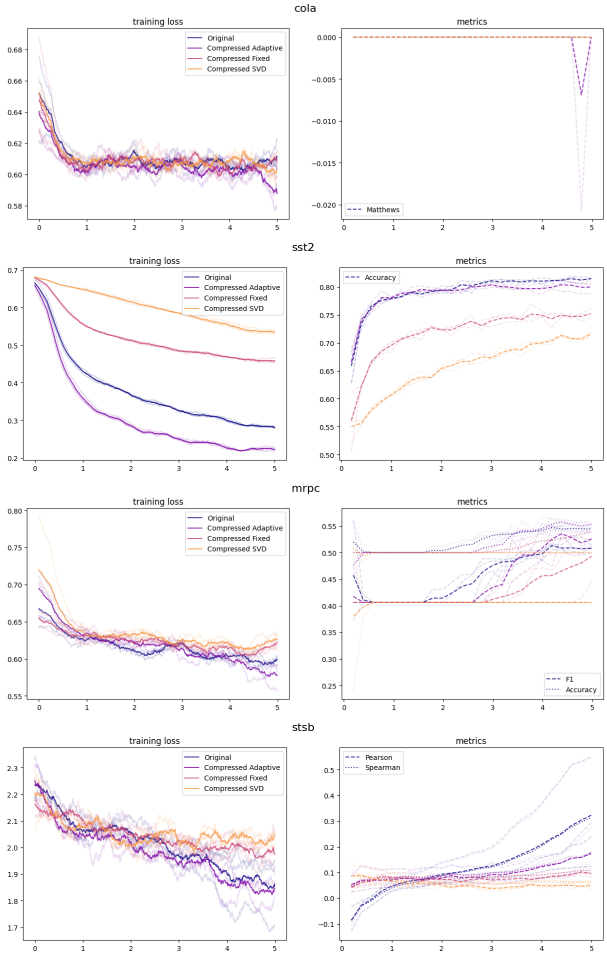
It's important to point out that the sizes plotted are after fixing the indices.

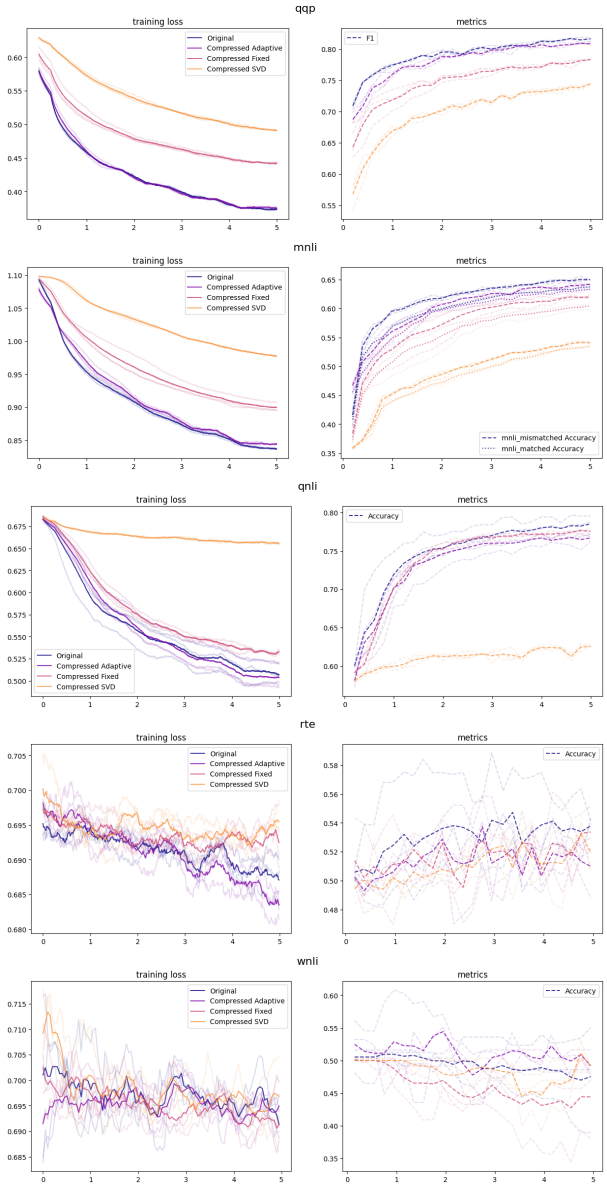
5.3 Convergence speed in retraining

Here I test the convergence speed and final performance of the model with substituted embeddings. The four model considered in this experiment have the following embeddings:

- the original embedding matrix;
- the compressed approximation mantaining the indices trainable: it's heavier than the final fixed embedding because it also has to store the soft indices matrix;
- the compressed approximation with fixed indices: way lighter than the previous but less flexible;
- SVD low rank approximation of the embeddings.

This allows us to see how well the model can perform compared to the original and low matrix approximation both in case we can train also the indices and in case training together model and approximation is too expensive so we have to fix the indices before retraining.





The model was trained over the Glue benchmark datasets for 5 epochs, adam optimizer with learning rate of 5e-5 and batch size of 32. I didn't do hyperparameter tuning and I just took some sensible parameters from the google github repository of tinybert, since the focus is not on the absolute performance of the model but instead on the relative behavior of the different approximations. For this test I took the approximated embedding of the pretrained model computed in the previous experiment, I used the one with 8 levels, and 8 channels and MLP with 1 hidden-layer and 128 neurons, and compared it with the slightly larger (with respect with the fixed indices version) rank 5 SVD approximation. I ran the test 3 times with seeds 42, 101 and 255.

5.4 Performance without fine tuning

In this test I analyzed the performance of the model with approximated embedding without finetuning the embeddings. I first trained the TinyBERT model on the Glue tasks then computed the approximations of the already trained embeddings and computed the validation metrics with the approximations. Following prior works I tested also if changing the p-norm used to train the embeddings (Tukan et al., 2020) or using an additional cosine similarity (Balazy et al., 2021) produces significant difference in performance in this setting. The results are summarized in table 3. As before, the test was run with seeds 42, 101 and 255.

5.5 Performance approximating indexes

Finally I ran again the L2 error test using the approximated indices during training to reduce the memory footprint during training, as proposed at the end of the system description. again it was ran with an MLP of 1 hidden layer of 128 neurons, testing all the possible combinations of layers and channels in [2, 4, 8, 16, 32]. Fig.4 plots the results and compares them with the complete version.

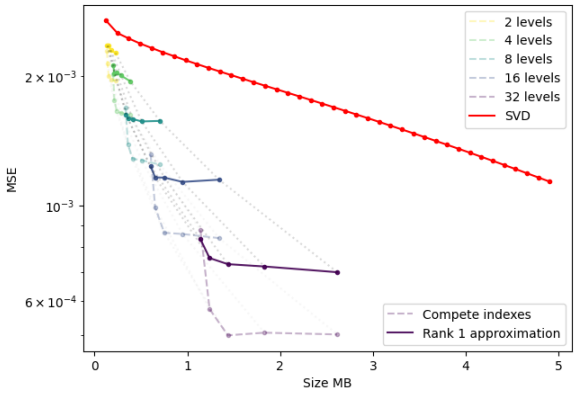


Figure 4: Approximated soft indices vs complete

6 Discussion

The comparison between the softmax and the linear straight-throught estimator show that not only the linear straight through is faster to compute, but it also converges faster to a lower minimum.

The comparison between the proposed framework and the low matrix approximation shows a superior expressiveness for each size of the model tested. The plot also shows that increasing the number of channels per level hits a point of diminishing return, especially if there are a lot of layers. A possible explaination could be that the decoding MLP

	cola	sst2	mrpc	stsb	qqp	mnli	qnli	rte	wnli
Original	0.0±0.1	81.3±0.4	50.0±1.0 / 54.0±0.5	36.8±17.8 / 36.1±17.3	81.4±0.7	65.2±0.3 / 63.9±0.4	78.1±0.8	54.6±1.6	44.1±2.4
Norm 1	0.0±0.1	70.5±2.8	41.2±0.8 / 50.3±0.4	22.0±8.8 / 21.6±9.8	51.4±2.8	34.9±2.2 / 35.0±2.2	53.0±4.2	50.2±0.3	50.1±0.1
Norm 1.25	0.0±0.1	70.7±2.5	41.2±0.8 / 50.3±0.4	19.6±8.8 / 19.6±9.7	55.3±2.3	34.3±1.0 / 34.3±1.1	53.8±5.1	51.2±1.6	48.3±2.5
Norm 1.5	0.0±0.1	70.9±4.3	40.9±0.4 / 50.1±0.2	22.5±9.9 / 20.9±9.8	45.0±3.2	36.1±2.6 / 36.0±2.4	54.2±5.6	50.5±0.2	50.0±0.1
Norm 1.75	0.0±0.1	68.9±1.0	40.9±0.4 / 50.1±0.2	20.7±8.1 / 20.4±9.5	48.3±6.3	34.7±1.0 / 34.4±0.8	55.2±5.3	51.5±1.3	49.3±0.5
Norm 2	0.0±0.1	69.3±0.3	40.9±0.4 / 50.1±0.2	21.6±8.4 / 22.0±10.2	48.3±6.1	36.7±2.8 / 36.3±2.4	57.2±4.9	49.9±0.1	49.5±1.7
Norm 1 Cosine	0.0±0.1	73.4±1.0	42.0±1.4 / 50.6±0.7	15.6±3.2 / 15.1±3.5	49.9±7.7	34.7±1.3 / 34.6±1.1	67.3±3.4	51.6±3.7	49.2±2.5
SVD	0.0±0.1	52.6±1.7	40.6±0.1 / 50.0±0.1	5.7±1.4 / 6.4±2.0	38.9±0.2	33.3±0.2 / 33.4±0.2	51.3±0.9	50.0±0.1	51.0±1.5

Figure 3: Results on Glue without finetuning the approximation

doesn't need to store many information per level, but instead relies more on the number of levels to better handle the collisions and better distinguish the words. On the other hand, while adding more channels per level increases the size of the final model, it requires a relatively small amount of additional memory during training, so if the bottleneck is the memory, training a model with less levels and more channels could be a viable option.

The glue retraining test in all the tests where the model shows to be learning (some datasets are just too complex even for the original TinyBERT model) the proposed approximation substantially outperforms the low rank approximation, decreasing the training loss faster and scoring higher results in the validation metrics, both with tranable and with fixed indices.

In the setting without retraining we can see that on average the proposed model is better than the SVD approximation, but it's not too clear the advantage of using one loss over another. As expected the results are still way worse than what is achievable finetuning the approximation.

Finally from the test of approximating the indices matrix we can see that the obtained model is still way more expressive than the low rank approximation, but at the same time we can see a clear degradation in performance when training the straight through with approximated indices, and could start to make more sense to use a lower bitwidth table instead.

7 Conclusion

In this work I tested the feasibility of using a codebook with trainable indices for compressing the embedding matrix of language models obtaining quite promising results, substantially outperforming the commonly used low rank approximation in most tests. I also proposed a new formulation of the straight through estimator that was able to improve the convergence of the indices. Finally I tested the performance of the proposed model against the common low rank approximation in a variety of settings, and tested some possible approach to reduce the big amount of memory required by the method during the training of the approximation.

8 Links to external resources

Github repository: https://github.com/samuele-bortolato/CodeBookEmbeddings

References

Klaudia Balazy, Mohammadreza Banaei, Rémi Lebret, Jacek Tabor, and Karl Aberer. 2021. Direction is what you need: Improving word embedding compression in large language models. *CoRR*, abs/2106.08181.

Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. 2013. Estimating or propagating gradients through stochastic neurons for conditional computation. *CoRR*, abs/1308.3432.

Prajjwal Bhargava, Aleksandr Drozd, and Anna Rogers. 2021. Generalization in nli: Ways (not) to go beyond simple heuristics.

Towaki Takikawa, Alex Evans, Jonathan Tremblay, Thomas Müller, Morgan McGuire, Alec Jacobson, and Sanja Fidler. 2022. Variable bitrate neural fields.

Murad Tukan, Alaa Maalouf, Matan Weksler, and Dan Feldman. 2020. Compressed deep networks: Goodbye svd, hello robust low-rank approximation. *CoRR*, abs/2009.05647.

Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Well-read students learn better: The impact of student initialization on knowledge distillation. *CoRR*, abs/1908.08962.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *CoRR*, abs/1804.07461.