ALTEGRAD is all you need

Mahdi KALLEL[1] , **Yassine NAJI**[2] , **Ahmed LACHTAR**[2,3]
[1]Telecom Paris
[2]Telecom Paris
[3]Ensta Paris
{mahdi.kallel, yassine.naji,ahmed.lachtar@ip-paris.fr}@ip-paris.fr,

Abstract

In this paper we present our work for the ALTEGRAD H-index prediction competition. The paper is structured as follows: In the first section we will discuss the exploration of the dataset.

In the second and third section we will discuss of unsupervised and self supervised approaches to extract embeddings from the text and graph data respectively.

In the fourth we will dive into the prediction architectures and in the last section we will discuss our results.

1 Dataset:

1.1 Dataset Exploration :

Our dataset consists of around 230k authors, and their corresponding 1.08 M abstracts.

We noticed that around 2.500 authors didn't have any corresponding abstracts. Therefore we chose to give them the embedding of the closest neighbour which has at least one abstract.

Exploring the abstracts We wanted to first check the distribution of the number of papers per author. This can be seen in the following plot. We notice that the maximum number of papers is 10 which leads us to wonder if we only consider 10 abstracts per author at a maximum, and if the missing abstracts belong to these authors.

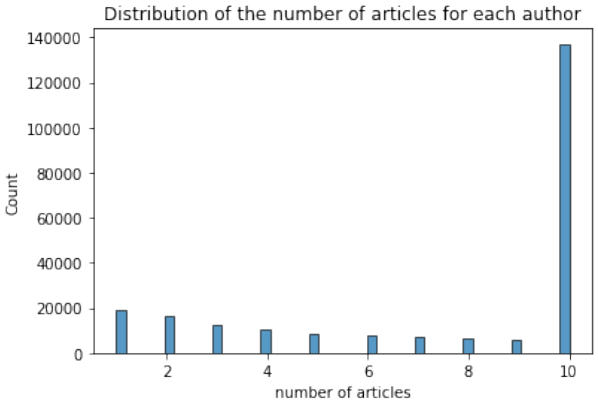

Figure 1: Distribution of the number of articles per author

We then moved to check for the number of words per abstract, and if it differs largely from paper to per. We can notice through this graph that some authors have an abstract of 7000+ words. This explains the time consumed by the text baseline to extract representations from the papers. The average however is 597 words.

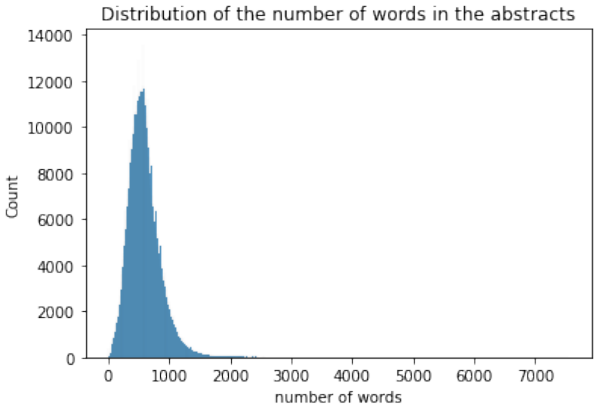

Figure 2: Distribution of the number of words per abstract

Exploring the graph properties The graph is an important part of the prediction model we want to accomplish. Therefore, it can be useful to explore its properties, mainly the ones

used in the baseline model.

We begin by exploring the average degree of nodes. This can gives an idea on how dense the graph is, given the important number of nodes.

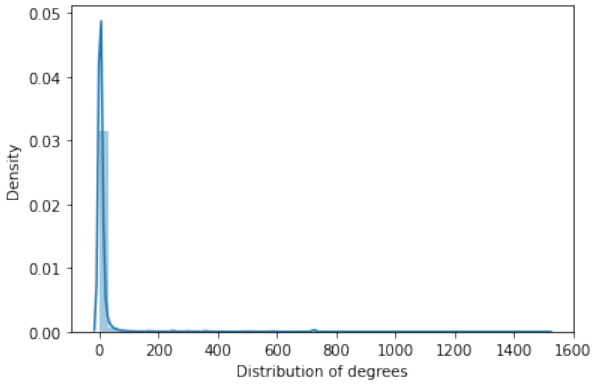

Figure 3: Distribution of the average degree

We can see that most nodes have on average a degree of 15. There are however nodes with a degree of 1508 which is interesting, suggesting that some nodes are central in this graph.

When we look to the "H-index" target we see it is heavily right skewed. This can cause issues for the predictor as the targets are dominated by small values with some few but really extreme others.

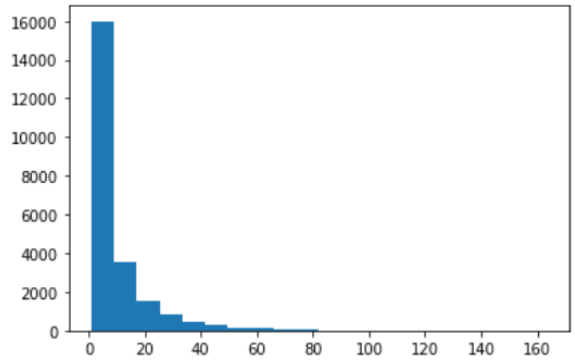

Figure 4: H index distribution

We try to correct for this skew with a boxcox transformation and we get a more smooth distribution, we use this new distribution as our target and we use the inverse transform for submission.

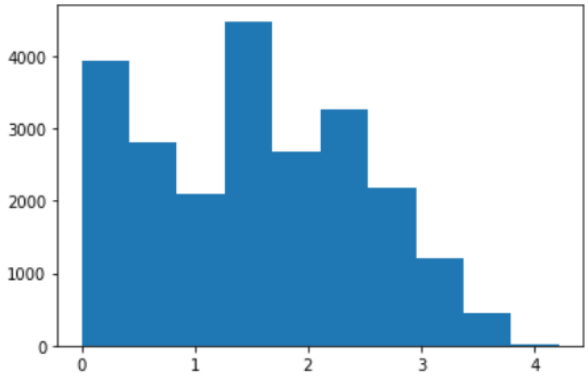

Figure 5: H index distribution after boxcox transform

1.2 Spectral clustering:

We wanted to explore the structure of the graph. Hoping to find a clustering with high variability.

For this purpose we use the spectral clustering algorithm to look for strongly connected components. Sklearn's implementation is really slow for a dataset of this size.Therefore, we implemented our own version using Pytorch and GPU.

Here we plot the modularity of the clustering as a function of the number of clusters :

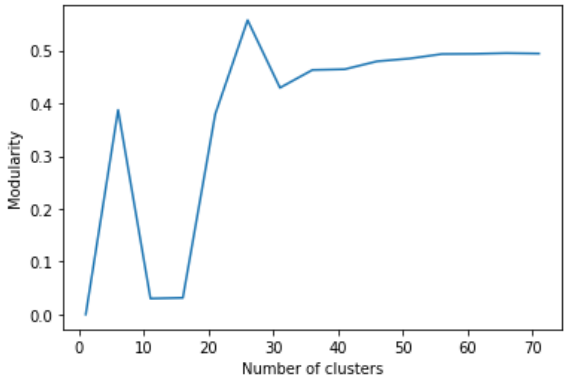

Figure 6: Modularity of the partitioning as a function of the number of clusters.

For the rest of this work, we add to each node the information about each cluster average H-index Standard deviation and the $10/percent$ percentiles. We chose to work with 70 clusters as this offers a higher standard deviation between the median targets at each cluster.

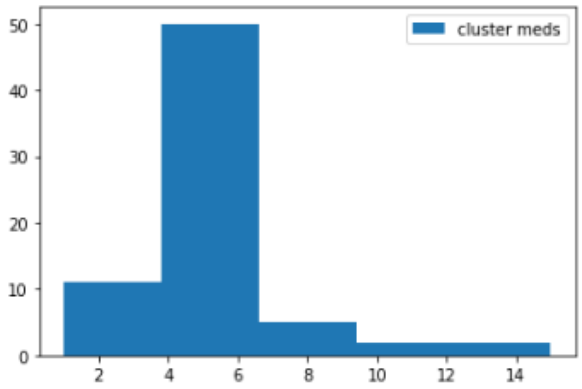

Figure 7: Histogram of clusters median target, for n=70 clusters.

1.3 Centrality metrics:

The are indictors of the most important vertices within a graph.

The problem of such metrics is that they are quite hard to compute. Like the closeness and betwenness centrality who both rely on pairwise shortest paths.

The only centrality metrics we could compute in a reasonable time where "PageRank" and "Degree centrality".

We tried computing the above metrics using Neo4J which is a database optimized for graph workloads. However we were limited by our laptops' RAM sizes.

2 Abstracts embedding

For each author, we have a list of all the papers he wrote. We also have most of the abstracts of these papers. The goal is to generate author embeddings from these abstracts and to use those to predict the H-index. So, there are 2 challenges, getting paper representations and then extracting author representations from them, because each author has in average 7.5 papers.

2.1 Key words extraction:

Compared to the baseline provided with the challenged, we believed that averaging the Doc2vec embeddings of the abstracts over authors loses some information. Plus the abstracts contained much repetitive words like "paper", "section.." which can bias the embeddings.

We therefore decided to have a fixed size summary representing each author. This summary is obtained by extracting the keywords from the concatenation of all the papers of such an author.

For the key-word extraction we used two algorithms:

1. **KeyBert:** This technique is based on BERT-embeddings and a simple cosine similarity to find the sub-phrases in a document that are the most similar to the document itself.

   First, document embeddings are extracted with BERT to get a document-level representation. Then, word embeddings are extracted for N-gram words/phrases. Finally, we use cosine similarity to find the words/phrases that are the most similar to the document. The most similar words could then be identified as the words that best describe the entire document.

Figure 8: Wordcloud of key words using KeyBERT

2. **Tf-IDF:** The TF-IDF algorithm is well known for it's effectiveness in extracting keywords from a large set of documents [**?**]. The algorithm calculates for each word the frequency in the document and the frequency in all the documents and then extracts the ratio :

$$\mathbf{tf}(t,d) = \frac{f_d(t)}{\max_{w \in d} f_d(w)}$$

$$\mathrm{idf}(t,D) = \ln\left(\frac{|D|}{|\{d \in D : t \in d\}|}\right)$$

$$\mathrm{tfidf}(t,d,D) = \mathrm{tf}(t,d) \cdot \mathrm{idf}(t,D)$$

$$f_d(t) := \text{ frequency of term t in document d}$$

$$D := \text{ corpus of documents}$$

We extract the 10 words with the biggest ratio for each paper. We now have a list of the paper ids and their respective keywords.

For each author, we extract the keywords and their $Tf - Idf$ from all his papers. We then sum the ratios for keywords that are the same and select the 10 words that have maximum ratios (Here, we used single words, a combination of 2 and 3 words). We therefore obtain the list of authors and their respective keywords.

This list is then converted to a vector using Doc2Vec from the gensim package. We tried various configurations for the vector size (64,128 and 256). The best results were obtained using the 256 size configuration

We noticed that this method doesn't yield the best results. However, execution time is much more inferior to the baseline

method executing in less than 10 minutes. This algorithm was optimised in various ways (by using better search algorithms for the search of maximums for example) to get these results.

Another approach we tried is coding the keywords into dummy variables. We have 5002 different keywords. For each author, we extract 10 keywords, code them each in a vector of 5002 and then sum these vectors. We get a vector where we have all the keywords mentioned in the author's work. These vectors will enable us to see which authors have worked on similar topics. We used these embeddings in our model and it yielded better results.

Figure 9: Wordcloud of keywords using Tf-IDF

2.2 Document embeddings:

The obtained author summaries are fed into a model to obtain an embedding. We tried two approaches using a pretrained Bert and Doc2vec.

Although Bert architecture is a lot faster (since it requires no training and is highly parralel on gpu), the resulting embeddings perform poorly when it comes to prediction power. This may be due to the fact that Bert is trained for inference on concrete phrases and not stacked words, another factor is the training vocabulary which is majoritarely non scientific.

Doc2vec seems to be the best model for our specific use case as is allows to have more personalized embeddings and does not require much ressources to train.

3 Node embedding

We tried several approaches for node embedding, some of them catches the positional information of the nodes and others encode the graph structure. We will detail in the following the algorithms that we used.

- **Deep Walk :** That was the first approach we used for node embedding since it was the less computationally expensive. We used the implementation of the following repository [1], we tried the following set of parameters : number of walks : 10, walk length : 40 , window size : 5 , embedding space : 64, the execution took around 4h. We achieved a MAE of 6 using only this embedding, we find out also that feeding the deep walk features to our final model improved slightly the MAE.
- **Line:** We manage to use more sophisticated algorithms which highly performed in mutli-classification tasks. As we can see in the following graphic, SDNE seemed to be a good candidate.

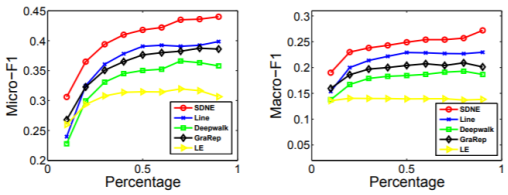

Figure 4: Micro-F1 and Macro-F1 on BLOGCATALOG. The results show that our method achieves better classification performance than that of baselines.

We manged to run it on the whole graph using the implementation of the git repertory [2] however we had memory issues while executing it. So we decided to run instead LINE algorithm which preserves the structural information of the graph by learning representations which reproduces the first and the second order proximity in the graph. More precisely, in case of the LINE with first order proximity, the quantity is maximized:

$$\underset{\mathbf{f}}{maximize} \sum_{(v_i,v_j)\in E} w_{i,j} \log P_1(v_i, v_j)$$

Where :

$$P_1(v_i, v_j) = \frac{1}{1 + e^{-f(v_i)^T f(v_j)}}$$

For the second order proximity we maximize the objective:

$$\underset{\mathbf{f},\mathbf{f'}}{maximize} \sum_{(v_i,v_j)\in E} w_{i,j} \log P_2(v_i|v_j)$$

Where :

$$P_2(v_i|v_j) = \frac{e^{f'(v_j)^T f(v_i)}}{\sum_{k=1}^{|V|} e^{f'(v_k)^T f(v_i)}}$$

In our case, we tried LINE with only the first order and with both, we found out that learning both orders gives better features and helped our final model decreasing the MAE.

- **Watch your steps:** This paper uses a straight forward approach to compute embeddings. Indeed, if we denote by $\mathbf{Y} \in \mathbb{R}^{n,d}$, the embedding that we want to learn, it can decompose it into $\mathbf{Y} = [\mathbf{L}|\mathbf{R}]$, where $\mathbf{L}, \mathbf{R} \in \mathbb{R}^{n,\frac{d}{2}}$.

This decomposition allows to compute the loss function. In fact, if we consider $g(\mathbf{Y}) = \mathbf{L}\mathbf{R}^T$, we can compute the following loss function:

$$\min_{\mathbf{Y}} \| - D \circ \log(\sigma(g(\mathbf{Y}))) - \mathbf{1}[\mathbf{A} = 0] \circ \log(1 - \sigma(g(\mathbf{Y}))\|_1$$

We used the implementation of Stellar Graphs [3] in order to comput e embeddings

$D \in \mathbb{R}^{|V| \times |V|}$ is the co-occurrence matrix from random walks, and σ is simply the sigmoid function. Training of the algorithm took form us more than 16h, however the embedding that we get where less good than Line or deep walk. In fact, when we feed it alone to our deep model we achieved an MSE of 7, which is high comparing to other embedding algorithms, so we didn't use those embeddings for our final model.

- **Struc2Vec & Graph Wave**: We found out that structural information of the graph are very important for our task, since the centrality of authors in the graph is correlated to their h-score. Thus we manage to use the well known algorithm : Struc2Vec [4], we tried the official implementation which is available in [5], with many optimization, but the execution on Google Colab exceeded 12 hours and we couldn't execute it on our machines since algorithm has a complexity of $O(n \log(n))$ so we had memory errors. We tried another algorithm which is called **Graph Wave** [6], which also learn the graph's structural information via diffusion wavelets, we used the implementation of Stellar Graphs [7]. Similarly to Struc2Vec, the algorithm was very slow even on GPU and it exceeded 12h on colab.
- **Diffusion:** In order to estimate the h-index for test nodes, we tried to use a semi supervised approach which consist of label propagation (adapted to the case of regression). Each unlabeled node receive the average of the h-scores of it's neighbors while keeping the labels of the training nodes fixed. This approach can be seen as a heat diffusion phenomenon, where the labeled nodes have a fixed temperature and edges allows heat propagation. The temperature of the nodes converges to the harmonic solution, where the temperature of each node is the average of it's neighbors. We implemented 2 variant of heat diffusion, the first one with considering all edges equivalent and the second by taking into account the text representation of the authors, for that we took the scalar product of the representations as the weight of the edge between 2 authors, the latter variant gave us slightly better results than the first one, however, in general this method didn't give good results in terms of MAE, this is probably due to the fact that it overestimates the h-index of unlabeled nodes, since a node which is connected to a labeled node with a high h-index will have a relatively high score, which is not true in general.

4 Models:

Having both the author and the graph data offers a wide choice of models for the task.

In what follows we discuss the various architectures we implemented.

4.1 Deep networks:

We implemented 3 architectures :

- **Dense network**: This model is composed of 6 fully connected layers with ReLU activation, and a dropout on the top of each layer (with a dropout rate of 0.04). The number of hidden neurons decreases from a layer to another following this equation :

$$n_{hidden}^{(l)} = \max\left(100, n_{input} - \left\lfloor \alpha l \frac{n_{input}}{n_{layers}} \right\rfloor\right)$$

We designed this equation so that we have a linear decrease of the number of hidden neurons, the α parameter controls the decrease, after tuning, we found that $\alpha = 3.93$ is the best value.

The final layers is also fully connected with 1 output.

Using this model we achieved our best score on the test set :3.55.

- **Multi-Dense Network :** In order to prevent overfitting, we designed a model which is composed of 8 sub networks, each sub network access to a part of the input. More precisely, we list the sub-networks and their corresponding features:
  - Sub-Network 1 : Deep walk embeddings
  - Sub-Network 2 : LINE embeddings
  - Sub-Network 3 : Keywords embeddings
  - Sub-Network 4 : Mean of paper representations (using Doc2Vec)
  - Sub-Network 5 : Std of paper representations
  - Sub-Network 6 : Max of paper representations
  - Sub-Network 7 : Min of paper representations
  - Sub-Network 8 : other features (average degree, core number, average node degree , number of papers, clustering number)

  Each sub-Network is composed of 2 dense layers with ReLU activations, the hidden size is half of the size of the features and the output is 8 for all sub-networks. We add finally a fully connected layer that outputs the predicted h-score.

  This network showed that it overfit less, more stable and converges more quickly, but it didn't give the best validation results (it achieved only 3.65 on the validation), on the test we achieved a score of 3.68 using it.
- **Attention Network :** In order to evaluate features relevance, we implemented also an attention network. It's composed of 2 sub-networks, each of them access to a part of the input, however unlike the Multi-head Network, each network access to some graph and text features jointly and predict directly a h-score:
  - Sub Network 1 :Deep Walk , mean paper representation and std paper representation.

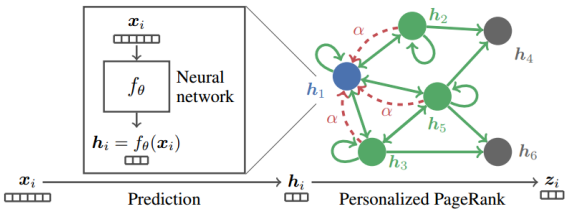

Figure 10: Illustrated APPNP, algorithm

– Sub Network 1 : LINE , max paper representation, min paper representation, graph features.

The attention mechanism access to the whole input and choose the weights for averaging the 2 networks predictions. This model performed less than the 2 other networks, we achieved only a MAE of 3.88 on the test set using it.

4.2 Graph networks:

For our experiments with GNN's we used the "StellarGraph" libray.

We tested simple Graph convolution networks and Graph Attention Networks with unsupervised pretraining using "DeepGraphInfomax", this algorithm allows learning in a self supervised manner representations that capture the global information content of the entire graph. This is done by tasking the model to distinguish pairs of corrupt and real nodes and thus local mutual information.

but the models performance was quite poor around 4.5 MAE.

We find a glimpse of explanation in the following paragraph.

4.3 APPNP, GNNs Meets Deep:

APPNP stands for "Approximate personalized page rank network propagation".

This algorithms is similar to GNN's but works by decoupling the node prediction and the message passing phase.

It's a semi-supervised learning scheme were we generate predictions for each node based on its own features and then propagate them via fully personalized PageRank scheme to generate the final predictions.

This algorithm is characterized by a value α which determines the length of the random walk from the node and thus the influence we let the neighbours have on a certain node.

We find that most of our parameter optimization trials converge to a very low value of $\alpha = 0.05$ meaning that each node value is quite independant of it's neighbours.

Which can explain why all the previous GNN schemes break for this task.

4.4 LightGBM:

LightGBM is an implementation of the Gradient Boosting Decision Tree (GBDT) algorithm ([**?**]. The GBDT has proven to reliable in classification and regression tasks. However, it still lacked scalability and efficency when the number of features is high and the data size is large. To tackle this problem, LightGBM proposes two novel techniques: Gradient-based One-Side Sampling(GOSS) and Exclusive Feature Bundling(EFB). With GOSS, a significant proportion of data instances with small gradients is excluded, and only the rest is used to estimate the information gain. With EFB, exclusive features are bundled mutually (i.e., they rarely take nonzerovalues simultaneously), to reduce the number of features. This algorithm has proven to be 20 times faster in training than conventional GBDT models.

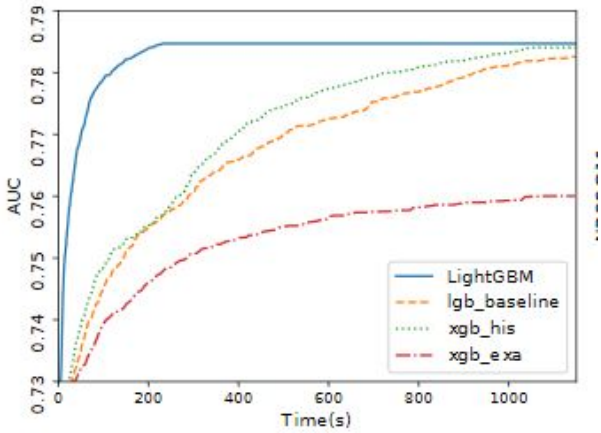

Figure 1: Time-AUC curve on Flight Delay.

Figure 11: Performance regarding time of LGBM

This figure shows the difference in performance compared to training time of LightGBM and conventional GBDT implementations on the FlightDelay Dataset.

5 Experiments Results:

For all our submissions we trained on 95% of the data with an early stopping on the remaining 5%.

Hyper parameter tuning was performed using Optuna library. Unlike grid-search this library allows for relatively faster tuning via bayesian methods. And thus it leverages the history of tested parameters to define the next set to to test. [9]

We also tried a k-fold bagging scheme where we have k-models each trained on (k-1) folds and validated on the last. Then we average the predictions of all these models. We believed that such a scheme can be better as it leverages 100% of the available data. However in practice the first simple strategy proved more efficient.

The main parameters to tune we're the learning rate, the depth of the network and the optimizer choice (LAMB[11]or NovoGrad[12]). Introducing these recent optimizers improved our submission score by 0.2.

The following table summarizes the submission results of our 4 architectures:

Model	Submission score
LGBM	3.9
Multi dense	3.65
Sequential + APPNP	3.58
Sequential	3.55

5.1 Results:

References

[1] **Deep Walk implemetation:**
https://github.com/phanein/deepwalk

[2] **SDNE , LINE implementations:**
https://github.com/shenweichen/GraphEmbedding

[3] **Watch Your Step: Learning Node Embeddings via Graph Attention** by Sami Abu-El-Haija, Bryan Perozzi, Rami Al-Rfou, Alex Alemi

[4] **Struc2vec: Learning Node Representations from Structural Identity** by Leonardo F. R. Ribeiro, Pedro H. P. Savarese, Daniel R. Figueiredo

[5] **Struct2vec implemetation:**
https://github.com/leoribeiro/struc2vec

[6] **Learning Structural Node Embeddings via Diffusion Wavelets** by Claire Donnat, Marinka Zitnik, David Hallac, Jure Leskovec

[7] **Stellar Graphs**
https://stellargraph.readthedocs.io/en/stable/

[8] **Deep Graph Infomax**
https://arxiv.org/pdf/1809.10341.pdf

[9] **Optuna libray:**
https://github.com/optuna/optuna

[10] **Key Bert Library:** https://github.com/MaartenGr/KeyBERT

[11] **Large Batch Optimization for Deep Learning: Training BERT in 76 minute**
https://arxiv.org/abs/1904.00962

[12] Stochastic Gradient Methods with Layer-wise Adaptive Moments for Training of Deep Networks
https://arxiv.org/abs/1905.11286