Graph-Embedding Based Clustering for Log Anomaly Detection

Anonymous Submission

Abstract

Console logs rarely help operators detect problems in large-scale data-center services, for they often consist of the voluminous intermixing of messages from many software components written by independent developers. However, log messages are the most straightforward indicator of anomaly events. Thus, automatic anomaly detection on logs with computers is very important for automatic operation and maintenance systems. In this proposal, we leverage both the semantic knowledge and relationship among individual log messages to create meaningful and robust embedding for each event. The embedding is then aggregated to form sequence feature vectors for log anomaly detection through clustering and distance weighing. With the employment of efficient clustering assignment model, we reduce the offline computation time and complexity while preserving the distinct identification of anomalies. Our best model achieves 0.6 f1-score for the anomaly class and 0.97 weighted average f1-score for both classes, which surpasses previous state-of-art.

1. Problem Statement

Anomaly detection is a task of mining unexpected events or observations in datasets, where the anomaly deviates from the typical pattern and conveys important signals for fraud, defect, or errors. In most of the anomaly detection scenarios, the abnormal data occurs rarely comparing with normal data. Such an imbalance of data samples raises difficulties for choosing appropriate strategies for pattern mining, since typical machine learning models rely on the assumption that positive and negative samples are evenly distributed. Therefore, tackling a long-tailed distribution in datasets and designing applicable algorithms for anomaly detection is worth examining.

[*]Equal contribution . **AUTHORERR: Missing \icmlcorrespondingauthor.**

Proceedings of the 37th International Conference on Machine Learning, Online, PMLR 119, 2020. Copyright 2020 by the author(s).

In this paper, we focus on a subtopic of anomaly detection called log anomaly detection, in which the log messages are retrieved from large server clusters or cloud computing environments. We aim at detecting and diagnosing problems for such a large system architecture at a relatively low cost.

Designing scalable mining algorithm for log anomaly detection, however, is challenging due to the following reasons. First, similar to most of the anomaly detection scenarios, there are only a few of the console logs that contain error or abnormal operations. Second, the anomalies in log messages are hidden in a sequence of transactions, since most of the anomalies are only indicated by incomplete message sequences. That is to say, we should group and examine a series of log messages that associated with the same block_id (i.e. operation ID for identifying individual processes). Third, there are rich but complex semantic correlations between log messages, such as messages handling the same transaction or reading the same file. Disentangling the associations in text, therefore, helps models to learn and to generalize the patterns for normal text data.

Hence, we believe that designing robust models for log anomaly detection is valuable given its wide real life implications and existing difficulties in information mining. In the following sections, we will first introduce previous works related to this field and then propose our own methods. We design two strategies, graph-based and transformer-based, for log event embedding extraction. We leverage both the correlation and semantic information of log series in our feature embedding process. Then, we construct clustering model by aggregating the information from embeddings and assign clusters labels according to the proportion of anomalies. In testing, we calculate each test embedding's distance to clusters and fit it to either a normal cluster or abnormal cluster as the classification result.

2. Literature Review

2.1. Related Work

Throughout the past two decades, many different approaches have been investigated and implemented for the general topic of anomaly detection, largely splitting between two camps of supervised and unsupervised models. Many previous models were built using statistical methodologies (estimating distributions and finding the likelihood of the observed data) like Gaussian Mixture Model and Variational Bayesian Gaussian Mixture Model, and some borrowing tools from clustering methods like DBSCAN and agglomerative hierarchical clustering (AHC), before progressing into distance-based and density-based methods that we see in the early 2000s to 2010. For example, Local Outlier Factor (LOF) utilized density-based calculations gathered from distance information (reachability distance) similar to KNN and information on relative access (local reachability density) to data points like DBSCAN to classify and rank the "outlier-ness" of data points with an observable, intuitive quantity named local outlier factor.

Another popular method, One-Class SVM (OCSVM), operates on data points from a singular class by utilizing a margin learned from training (furthest from origin) to identify and flag data points not classified under the original class. Other linear methods like applying PCA through inverse mapping and comparisons of the error to identify outliers were also used. Examples of nonlinear methods include replicator network (example of one-class neural networks) and isolation forest (iForest) which utilizes a forest of binary trees that gives fast classifications and rankings of outliers based on geometric and probabilistic heuristics of outliers in feature spaces.

There were also methods inspired and specialized on anomaly detection tasks in large databases like invariant mining, log clustering, that rose into wide recognition, before deep-learning-based methods come into the scene such as DeepLog(Du et al., 2017) that utilized stacked LSTMs, AutoEncoder-based methods like Deep Autoencoding Gaussian Mixture Model (DAGMM)(Zong et al., 2018) and GAN-based methods like AnoGAN(Schlegl et al., 2019) (demonstrated capability of GAN for usage in anomaly detection), EGBAD (Efficient GAN-Based Anomaly Detection that implied the anomaly computation), and GANomaly(Akcay et al., 2018) (includes autoencoder, Generator-Discriminator network).

Currently, the SOTA (state-of-the-art) methods employed in database centered anomaly detection are Invariant Minings(Lou et al., 2010), DeepLog(Du et al., 2017), LogCluster(Lin et al., 2016), LogAnomaly(Meng et al., 2019), an autoencoder based method(Borghesi et al., 2019) proposed by Borghesi et al and a recent proposal of LogBERT[6], which have all been tested on the HDFS (Hadoop distributed file system) dataset for system log anomaly detections.

Invariant mining, by its nature of exploiting linear invariances, is the fastest in terms of training time, yet it suffers from being unable to perform log-based detection and only session-based detections as an inherently offline process. LogCluster(Lin et al., 2016) has the advantage of using clustering-based methods which require fewer transformations and data processing than other ensuing methods. DeepLog(Du et al., 2017) has the benefit of incorporating user feedback to improve its training results and adapt to new execution patterns. However, the above three methods all take log template indexes as input, instead of deeply processing the semantics in the logs, which can often induce false alarms (i.e. low precision scores), which is verified in LogAnomaly(Meng et al., 2019)'s experimentation. The heavy demands in training times and resources for DeepLog(Du et al., 2017), LogAnomaly(Meng et al., 2019), Autoencoder(Borghesi et al., 2019), and LogBert(Guo et al., 2021) also present practical challenges, despite achieving high F1 scores in most tasks.

Our approach is based on LogCluster(Lin et al., 2016). In this model, it consists of two phases, the construction phase and production phase, where they follow similar procedure. They first turn the log sequence results from a log parser model into vector with different weights according to two weighting methods. They then calculate the similarity value between two log sequences and apply a clustering algorithm to group the similar log sequences into clusters. Next, they extract the representative log sequences for each cluster and add the representative log sequences into knowledge base. The construction phase is used to construct the knowledge base while the production phase is used to update the knowledge base when new clusters occur.

2.2. Baseline

We choose to utilize OCSVM, iForest, PCA, LogCluster and DeepLog as our base models from previous related works. In addition, we will create a simple N-gram model to utilize as another base model. The N-gram model will be trained on events as individual grams(tokens), while the prediction of the N-gram model would come from comparing the probabilities of seeing a chain of ngrams in a given sequence under the positive (anomaly) class vs that under the negative (normal) class.

2.3. Dataset

Our experiment employs the HDFS log dataset. HDFS is the Hadoop Distributed File System designed to run on commodity hardware, in which The logs are sliced into traces according to block ids, and each trace associated with a specific block id is assigned a ground truth label.

Lineid is an indicator for each line of log message in the dataset. Date and Time are the timestamps when the log message is created. Pid is the page id. Content is the acutal log message content. EventTemplate is the content after eliminating the parameter values which is manually assigned for different types of log messages. Eventid is matched with each unique eventTemplate. More details about the generation and usage are discussed in section 3.1.

3. Method

Inspired by the method proposed by (Lin et al., 2016), we construct a similar model procedure that feeds embeddings of sequences of logs (denoted as sequences of events where events are recoded templates) into a clustering algorithm of choice. We extend their results by applying graph embedding using the Line algorithm (Tang et al., 2015), which are fed into a Gaussian Mixture Model for clustering, and then utilize different approaches for test input processing and prediction based on the clustering results by evaluating and approximating the likelihood a testing instance is abnormal.

3.1. Data Preprocessing

Following the convention of utilizing the log Parser procedure as DeepLog(Du et al., 2017) and Logcluster(Lin et al., 2016), We parse the system logs into rows of log messages, then replace all parameters within that log message with ¡*¿ to extract relevant keys, while preserving a **template** for each type of log message we encounter in training. More specifically, for an example of system log: "Receiving block blk-1608999687919862906 src: /10.250.19.102:54106 dest: /10.250.19.102:50010", we will obtain "Receiving block $\langle * \rangle$ src: /$\langle * \rangle$ dest: /$\langle * \rangle$)" as our template after parsing the relevant parameters from the system log.
Afterwards, we encode each unique template with an unique event ID, e.g. "Receiving block $\langle * \rangle$ src: / $\langle * \rangle$ dest: / $\langle * \rangle$" as "E5" as the processed input for our model. For all event ID that corresponds to the same Date + Time stamp (e.g., "Date = 81109 & Time = 203518"), we construct a container to hold all such IDs and name the container as one complete **sequence**. Hence, from 104,816 logs, we obtain a total of 7940 sequences with varying lengths.

3.2. Graph Embedding Module

In order to extract meaning from the sequences of events, we propose to utilize graph embeddings to encode neighboring information for the sequential data to obtain both a global meaning and local representation based on the co-occurrences of events and the frequencies of event appearances.
To achieve this, we choose to create event level embeddings first, viewing each individual event as a single node in our graph, and defining its neighbors as the event that precedes it and the event that follows it. We then draw edges between the selected event and its neighbors to create a single line of graph per sequence of events.
After the construction of the graph, we pass our edges and nodes to the LINE algorithm (Tang et al., 2015) to obtain a final encoding of our events, treating events not co-occurring together as negative samples.
A resultant TSNE compression in 2d is presented below:

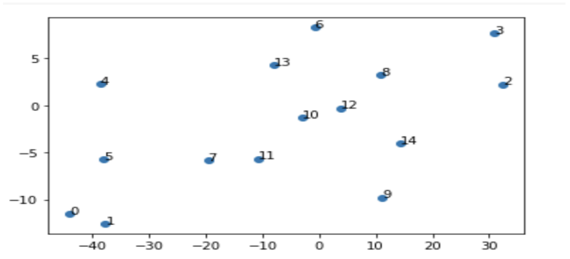


Figure 1. Embedding of coded events visualized with TSNE

We then utilize the individual event embeddings to create final embeddings for each sequences using simple summation of event embeddings based on the frequency of events appearing in that sequence. That is, viewing the sequence as a document, we sum the word (event) embeddings to obtain a final embedding per sequence. The motivation for not normalizing and using a frequency-based embedding per sequence is that we observed that most sequences are of length 12-16 (in terms of events), while only few instances are observed to have either short length (≤ 5 events) or long length (> 100 events).

We obtain another TSNE visualization on the training embeddings for individual sequences:

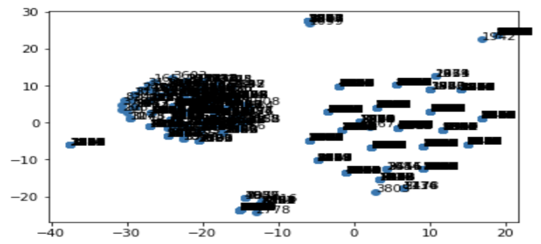


Figure 2. Embedding of coded events visualized with TSNE

Motivated by the inherent clusters that form under the training embeddings, we then fit a Gaussian Mixture Model (GMM) with n clusters on our training embeddings. We choose to include and create clusters based on the anomaly data as well, in contrast to the practice of utilizing only normal data under LogCluster(Lin et al., 2016).
For n=4, we obtain a clustering result of having 99 anomalies out of 3910 instances in cluster 1, 1 anomaly out of 2 in cluster 2, 27 anomalies out of 27 in cluster 3, and 29 anomalies out of 30 instances in cluster 4. We then assign the cluster label based on majority vote of each cluster and break ties using a weighted vote based on the closeness of each sequence's embedding to the centroid of that cluster. We utilize the GMM's predicted probability of each data point (embedding) belonging to a cluster as a measure of closeness to the centroid. Through above, we de-

termined the labels for all n = 4 clusters of our GMM model.

3.3. Test Prediction

For testing, we simply read in the parsed sequences of log events, then create the same sequence-based embedding as in the training phase by summing the embeddings for all appeared events in each sequence.
Then, based on the cluster assignment of each testing data (embedding), we determine the final prediction label of that data, so that if it were to be placed within a cluster that has a label of 1 (anomaly) based on majority vote, we flag that testing sequence as an anomaly.

For our 50%-50% train-test split with n=4 clusters created for GMM, we obtained the following result:

Test sequence embedding

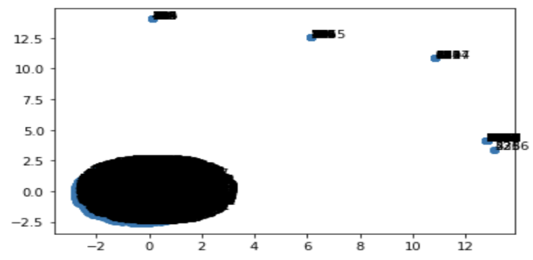
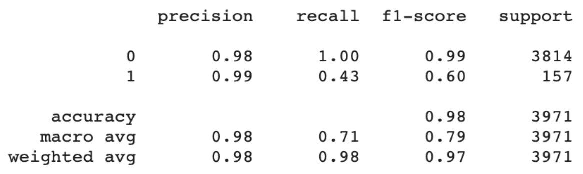


Figure 3. Embedding of coded events visualized with TSNE

Prediction evaluation



```
              precision    recall  f1-score   support

           0       0.98      1.00      0.99      3814
           1       0.99      0.43      0.60       157

    accuracy                           0.98      3971
   macro avg       0.98      0.71      0.79      3971
weighted avg       0.98      0.98      0.97      3971
```

Figure 4. Embedding of coded events visualized with TSNE

3.4. Embedding Clustering and Classification

In this subsection, we discuss the different approaches and experimentation we conducted in the process of embedding clustering and classification.

1) First, we compared the result of having equal length sequences with our previous illustrated procedure. To achieve this, we devised two different plans. The first involves padding each sequences to a a suitable size (16 = maximum length of a normal sequence, after disregarding the extreme case of having 200+ events), by appending $\langle PAD \rangle$ tokens at end of each sequence. The second involves slicing the training data into sequences of a fixed length, where sliced sequences coming from the same original sequence constitutes a session. For example, if we would like to slice our data into a fixed length of 3, and our first sequence is [”E3”, ”E1”, ”E2”, ”E8”, ”E15”], then we would obtain [[”E3”, ”E1”, ”E2”], [”E2”, ”E8”, ”E6”], [”E2”, ”E8”, ”E15”]] as our Session 1. However, we found that padding does not affect our performance much while the slicing approach decreases separability of our training and testing corpus embeddings, resulting in even poorer performances.

2) We have also attempted to change the prediction method utilized for our model. The first design involves saving not the centroid of each cluster as the embedding for that cluster, but rather utilizing the most ”eccentric” embedding from a given sequence (under the sliced dataset) as a comparison benchmark for determining the affiliation of an incoming data.

The second way involves calculating the L2-distance of a fed in testing embedding, such that if it were to be sufficiently distant from its nearest cluster with a non-anomaly cluster label, we would classify it as an anomaly.

The third way involves changing the data fed into the GMM in that we only train the GMM using normal data, and then assign the label of a given data based not on which cluster the data is assigned to but how unpredictable it is to assign a data to any cluster (i.e., if a testing embedding has close-to-uniform probability for being assigned to any cluster), we will classify that testing data point as an anomaly.

These results prove to be less promising than we imagined and so we turn to a different idea for improvement: semantic embeddings.

3.5. Transformer: Semantic Embedding

As our previous model uses only the event ID (i.e. E1) to associate log messages and find their embeddings, we believe the context of the event (i.e. ”Receiving block $\langle * \rangle$ src: / $\langle * \rangle$ dest: / $\langle * \rangle$”) is also meaningful for extracting discriminating features from messages. In order to leverage the semantic information of each log event, inspired by(Devlin et al., 2018; Nedelkoski et al., 2020), we apply the transformer with its Masked Language Modeling pretraining for embedding encoding.

For each template, we first tokenize it into words and append $\langle cls \rangle$ token to the front. We use $\langle cls \rangle$ to summarize the entire context of the message and to transfer to downstream tasks. Similar to the MLM task, we mask each position in the template at a time, and the collection of masked tokens form an array X.

We pass X to the transformer encoder, which linear projects X into key K, query Q, and value V. The multi-head attention block computes the attention score among tokens, and it thus encodes the long-range semantic relationship in a template. The attention score is calculated as:

$$Attn = softmax(\frac{Q \times K^T}{\sqrt{n}})V,$$

where n serves as the normalizing factor and is the dimension of Q and K.

We adopt the similar architecture of transformer from (Vaswani et al., 2017; Nedelkoski et al., 2020). After the last feed forward layer, we take the encoded feature vector for $\langle cls \rangle$ token to predict the masked word. We first projects it to the dimension of total number of tokens in dataset with a linear layer, then we apply a softmax to output the probability distribution for the masked word. Since the $\langle cls \rangle$ token is used to predict every masked word at different locations, it is forced to learn and summarize robust global semantic meaning of the entire template. We then extract the $\langle cls \rangle$ token as the semantic embedding of event and uses previously mentioned sequence clustering method for downstream anomaly detection task.

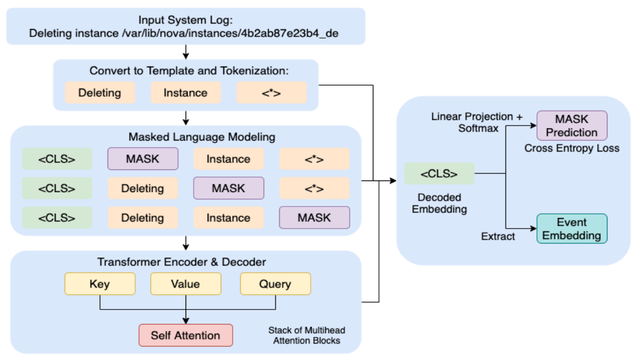


Figure 5. The Architecture of Semantic Embedding Module for Context Encoding

4. Result Evaluation

4.1. Comparison between baseline models

Models	Metrics			
	Accuracy	Precision	Recall	F-score
One Class SVM	0.97	0.03	1.00	0.05
Isolation Forest	0.96	0.54	0.69	0.60
PCA	0.96	0.06	1.00	0.11
LogCluster	0.98	0.99	0.37	0.54
DeepLog	0.96	0.58	0.12	0.20
N-gram	0.96	0	0	0
Embedding + GMM	0.98	0.99	0.43	0.60

Table 1. Performance of models on HDFS_100K

From table 1, we can see that all models have very high accuracy (above 95%). However, it is not because all models have good performance on the model, but that the dataset is very imbalanced (anomaly usually happens on a small set of blocks) and a model that always output false will also have good accuracy. Due to this internal challenges, no baseline models can reach both high precision and high recall, which means that they either tend to classify normal samples as anomaly or abnormal data as normal one. For example, one-class SVM and PCA has high recall, meaning that they can correctly classify all positive labels, but they falsely classify a lot of negative data into positive labels.

Also, it's interesting to see that Log Cluster has similar performance as our methods, which is probably because our method has very similar philosophy as the log cluster. Both models try to vectorize a given sequence and do clustering, but the way of embedding is very different. Traditional log clustering methods embed the sequence based on **TF-IDF**, an encoding based on word frequency. In contrast, our embedding utilize relative position of events, trying to maximize the probability of occurrence of consecutive pairs of events. The result proves that our model is slightly better than traditional methods.

4.2. Pros and Cons

Firstly, our model has significant advantage on memory usage. In contrast to the isolation tree, which requires the model to store the entire tree. Our model only requires the storage of embedding of each event and the cluster center, which is very memory efficient.

In addition to memory efficiency. Our model has very high accuracy, meaning that if we classify a data sample to be true, then it is highly possible that there are some anomalies within the system. In other words, we can detect some of the anomalies without falsely classifies some normal blocks into anomaly.

However, our model are not perfect either. Our model perform relatively poor in detecting anomaly samples, which is embodied in relatively low recall. In addition, our model may output very different embedding for events due to its internal randomness. Our model uses negative sampling, which is to sample multiple not encountered events along with the encountered pairs to update our model. Since the sampled data is different each time, our model shows very different embedding for events in different run. The unstableness of our models may bring undesirable uncertainty.

4.3. Further works

Firstly, we may try various ways of embedding for the events. Our embedding that exploit the order of events is only one of the options. Also, we may try to use different embedding for the sequence. For now, we sum up the events to be the embedding of the sequence. However, other methods may also perform well, such as a weighted summation of events. We may introduce another sets of learnable parameters to be the weight. we will try to conform the formats of semantic embedding and graph embedding and choose a better clustering algorithm to further improve the accuracy and training scale. In addition, to ensure better input for our GMM model, we can utilize variational autoencoders to output normally distributed data that feeds into the GMM model, which should provide better clustering results by fitting the gaussian distribution our GMM model assumes.

Besides, we only leverage part of the data. For example, given the log message "BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.203.80:50010 is added to blk_7888946331804732825 size 67108864", what we extract is only the block id (7888946331804732825) and the template of events (BLOCK* NameSystem.addStoredBlock: blockMap updated: $< * >$ is added to blk_$< * >$ size $< * >$, which is encoded as "E5" in our model). However, we can also extract the size information (67108864 in this case) and make it as an input to the model. Some other improvements include adding variational autoencoders to ensure that the output from embedding is normally distributed, so that our GMM clustering models may perform better.

5. Conclusion

log messages perform a significant role in the maintenance of large-scale computer systems, it's a challenging mission to detect anomaly in huge log files. In this paper, we propose the Graph-Embedding Based Clustering for Log Anomaly Detection, a novel embedding-based model to solve log anomaly detection problem. The model incorporates graph embedding and semantic embedding to address the problem of long training time and lack of semantic detection. After embedding, we perform the embedding clustering and classification. Of the three main methods we experiment on, we choose to classify testing embedding with GMM model. However, the other two methods, taking equal length sequences and calculating the L2-distance, are not discarded. The clustering strategy is subject to change after we integrate semantic embedding. Through our experiments and comparisons with other models, we show that our approach is effective on the HDFS Dataset.

6. Division of Work

- Baseline implementation: Yuetian Sun, Mingyang Zhang
- Graph embedding: Yuetian Sun, Rui Deng
- Semantic embedding: Mingyang Zhang, Rui Deng
- Clustering and classification methods: Yuetian Sun, Mingyang Zhang
- Experiment: Yuetian Sun, Ziyan Wang
- Report: all team members
- Presentation: all team members

References

Akcay, S., Atapour-Abarghouei, A., and Breckon, T. P. Ganomaly: Semi-supervised anomaly detection via adversarial training. In *Asian conference on computer vision*, pp. 622–637. Springer, 2018.

Borghesi, A., Bartolini, A., Lombardi, M., Milano, M., and Benini, L. Anomaly detection using autoencoders in high performance computing systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 9428–9433, 2019.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Du, M., Li, F., Zheng, G., and Srikumar, V. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1285–1298, 2017.

Guo, H., Yuan, S., and Wu, X. Logbert: Log anomaly detection via bert. *arXiv preprint arXiv:2103.04475*, 2021.

Lin, Q., Zhang, H., Lou, J.-G., Zhang, Y., and Chen, X. Log clustering based problem identification for online service systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 102–111. IEEE, 2016.

Lou, J.-G., Fu, Q., Yang, S., Xu, Y., and Li, J. Mining invariants from console logs for system problem detection. In *USENIX Annual Technical Conference*, pp. 1–14, 2010.

Meng, W., Liu, Y., Zhu, Y., Zhang, S., Pei, D., Liu, Y., Chen, Y., Zhang, R., Tao, S., Sun, P., et al. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In *IJCAI*, volume 7, pp. 4739–4745, 2019.

Nedelkoski, S., Bogatinovski, J., Acker, A., Cardoso, J., and Kao, O. Self-supervised log parsing. *arXiv preprint arXiv:2003.07905*, 2020.

Schlegl, T., Seeböck, P., Waldstein, S. M., Langs, G., and Schmidt-Erfurth, U. f-anogan: Fast unsupervised anomaly detection with generative adversarial networks. *Medical image analysis*, 54:30–44, 2019.

Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., and Mei, Q. Line: Large-scale information network embedding. *arXiv preprint arXiv:1503.03578*, 2015.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

Zong, B., Song, Q., Min, M. R., Cheng, W., Lumezanu, C., Cho, D., and Chen, H. Deep autoencoding gaussian mixture model for unsupervised anomaly detection. In *International Conference on Learning Representations*, 2018.