A Model Detecting Outlier by GNN

Abstract

This paper reviews three papers on outlier detection. They illustrate how graph neural networks can be used to detect outliers. In this paper, we focus on analyzing the LUNAR model and how it transforms the nodes of the graph neural network and analyzes their outliers. LUNAR is a relatively complete model that unifies all outlier detection algorithms. In this paper, we built our own LUNAR model based on the author's code. In the process of building the model, I gained a deeper understanding of deep learning and graph neural networks. Then we found through test performance that our model would perform even better.

1. Introduction

In the field of data analysis, outlier detection plays a crucial role in identifying abnormal observations that significantly deviate from the normal behavior of a data set. Traditional outlier detection methods often rely on local statistics or distance-based methods, which may not effectively capture complex patterns and relationships between data points. In the paper on the LUNAR model, the authors propose a novel approach to unify various local outlier detection methods using graph neural networks (GNN).

The key idea behind Lunar is to represent a dataset as a graph, where each data point is treated as a node and edges capture the relationships between them. By applying GNN on this graph, Lunar can effectively capture both local and global information, enabling more accurate outlier detection.

My code implementation will compare the performance of LUNAR's original model with the model I built, and I will use different data sets to compare the overall performance of my model. Finally, I will plot a learning curve to check the trend of the loss function. Official LUNAR Github Website:https://github.com/agoodge/LUNAR

In this paper, the main contributions are:

- Review the paper "LUNAR: Unifying Local Outlier Detection Methods via Graph Neural Networks"
- Review the paper "Addgraph: Anomaly detection in dynamic graph using attention-based temporal gcn"
- Review the paper "Graph neural network-based anomaly detection in multivariate time serie"
- Implement LUNAR model with my own version and interpret it with different datasets

2. Review of LUNAR

2.1. Storyline

High-level motivation/problem: When it comes to anomaly detection algorithms, the most common idea is to determine whether a sample point is an anomaly based on the local nearest neighbor distance. This local anomaly detection method, such as LOF, DBSCAN, and KNN, often performs well on feature-based unstructured data sets. However, these algorithms generally lack learnable parameters, which makes them difficult to adapt to different data sets. At the same time, because anomaly detection algorithms are often unsupervised learning, the hyperparameters of the above algorithms are difficult to tune based on performance, and these hyperparameters have a great impact. Anomaly detection methods based on neural networks have always performed unsatisfactorily on feature-based low-structured data sets due to the limited number of annotated abnormal point data and the fact that they are mainly targeted at high-dimensional structured data sets such as image data. Therefore, it is difficult for existing local anomaly detection methods to achieve the same stable performance in different data sets.

Prior work on this problem: Local outlier methods are currently used in a wide range of fields. These methods detect anomalies by measuring the distance of a point to its nearest neighbor, such as LOF and DBSCAN. In previous experiments, the performance of these local outlier methods is also more efficient than deep learning-based methods, because deep learning-based outlier detection methods are mainly designed for highly structured high-dimensional data, but in many Their performance is often poor in low-dimensional data structures. Therefore, today's local outlier method is still our default choice.

Research gap: Although today's local outlier methods are suitable for most situations, they lack learnability. They do not use information from the training set to optimize model parameters for better anomaly scores. Instead, they are based on predefined heuristics and hyperparameters.

Contributions: 1. A message passing scheme based on a graph neural network (GNN) is introduced to implement a simple and general framework that unifies local outlier methods.
2. A new method called LUNAR (Learnable Unified Neighborhood Based Anomaly Ranking) is proposed, which exploits the learnability of graph neural networks to address the shortcomings of local outlier methods and thereby handle the outlier problem more effectively.

2.2. Proposed solution

Given the problem: Suppose there are m normal training samples and n test samples, each of which may be normal or abnormal. For a test sample, the algorithm in the article should output an anomaly score. The higher the score, the higher the outlier value of the test sample.

The author first proposed a unified framework of current local anomaly detection algorithms on graph neural networks (Graph Neural Networks). Later, based on this graph neural network framework, the author proposed LUNAR (Learnable Unified Neighborhood-based Anomaly Ranking). With more trainable parameters, LUNAR is more flexible and adaptable in multiple data sets than existing local anomaly detection algorithms. In terms of performance and robustness, LUNAR also performs better than traditional anomaly detection algorithms and neural network-based anomaly detection algorithms.

To unify the framework of current local outlier methods, the local outlier methods collect information from neighboring sample points, calculate a statistic, and then use this statistic to determine whether the current sample point is an abnormal point. This process is incorporated into the message passing framework of GNNs.

The k^th layer of a GNN calculates the hidden representation of a node via the following equation:

$$\mathbf{h}_{\mathcal{N}_i}^{(k)} = \square_{j \in \mathcal{N}_i} \phi^{(k)}(\mathbf{h}_i^{(k-1)}, \mathbf{h}_j^{(k-1)}, \mathbf{e}_{j,i})$$
$$\mathbf{h}_i^k = \gamma^k(\mathbf{h}_i^{k-1}, \mathbf{h}_{\mathcal{N}_i}^k)$$

where $h_i^0 = \mathbf{x}_i$ and $\mathcal{N}_i$ is the set of adjacent ndes to i. $\mathbf{h}_{\mathcal{N}_i}^{(k)}$ is the aggregation of the messages from its neighbours.

LUNAR first builds a k-NN graph for the data set, using each sample in the data set as a node, and connecting the k nearest neighbors of each node to it. Messages are vectors of directed edges. Unlike other GNNs, the aggregation function in LUNAR is a learnable aggregation function. LUNAR can not only model graph data but also model feature-based structured data.

When LUNAR is aggregating, it does not perform unified maximum pooling on the information from k neighbors to convert it into final information. Instead, it encodes the k information into a k-dimensional vector and sends it to the neural network for processing. operation, thereby achieving the learnability of the policy during aggregation through the weight update of the neural network. The aggregation results obtained by the neural network operation are finally used for classification, with 0 representing normal samples and 1 representing abnormal samples. Since the data in the training set are considered to be normal samples, a reasonable abnormal sample generation strategy is needed to enable the model to learn the dividing line between positive and negative samples.

Negative Sampling: The first negative sampling method is to generate negative samples through a uniform distribution:

$$x^{(negative)} \sim \mathcal{U}(-\epsilon, 1+\epsilon) \in \mathbb{R}^d$$

where ϵ is a very small positive number(i.e. 0.1). Because abnormal samples may be too far away from normal samples, making it difficult for the model to learn the decision boundary, it is necessary to generate some abnormal samples that are closer to normal samples and more difficult to distinguish.

The second negative sampling method is to select a subspace among all feature dimensions of the normal sample and generate negative samples by adding Gaussian noise to the features in the subspace:

$$z \sim \mathcal{N}(0, I) \in \mathbb{R}^d$$

$$x_i^{(negative)} = x_i^{(train)} + M \circ \epsilon z$$

where ϵ is a very small positive number(i.e. 0.1) and p is set to 0.3. $M \in \mathbb{R}^d$ is a vector composed of binary random variables. Each element in M has p with probability 1 and (1-p) with probability 0. Each dimension set to 1 will be affected by noise.

2.3. Claims-Evidence

Claim 1 LUNAR gives the best performance on all datasets except SATELLITE, for which KNN is slightly better.

Evidence: *Table 3 in Section 8.4 from (Goodge et al., 2022)* shows that LUNAR gives the best performance on all datasets except **SATELLITE**, for which KNN is slightly better. All the best AUC score are highlighted in bold, and the average scores are marked following by **.

Claim 2 LUNAR gives the best performance in the vast majority of datasets and k settings. It is more robust.

Evidence: *Table 4 in Section 8.5 from (Goodge et al., 2022)* shows that the AUC score of the dataset **HRSS** decreases a lot as k increases, but LUNAR only drops in performance by 3 percentage points. LUNAR gives the best performance on most of the datasets and k settings, which means LUNAR is suitable at different k settings.

Claim 3 Mixing both negative SP type and sampling U type gives the best performance in most cases.

Evidence: *Table 5 in Section 8.5 from (Goodge et al., 2022) shows the evidence.* shows the performance of subspace perturbation(SP) and uniform(U) subsampling. In general, mixing two types will give us the best performance on most of the datasets.

2.4. Critique and Discussion

The most important thing about the LUNAR method is its learnability, so the performance of local anomaly detection is naturally high. The most innovative point is to combine the existing local anomaly detection algorithm with the graph model to do message, aggregation, and update, so as to obtain a unified framework. A major contribution of LUNAR is that it successfully unifies local anomaly detection methods including KNN, LOF, and DBSCAN. Under a unified framework, anomalies after different transformations can be classified based on their transformation consistency.

3. Review of AddGraph Anomaly Detection

3.1. Storyline

High-level motivation/problem: This paper proposes AddGraph, a general end-to-end anomaly edge detection framework, which builds an extended temporal GCN through the attention model, which can capture long-term patterns and short-term patterns in dynamic graphs. In order to solve the problem of insufficient label data, this paper uses selective negative sampling and edge loss methods to conduct semi-supervised training of AddGraph.

Prior work on this problem: There are currently some works such as CAD and Netwalk that apply graph embedding methods to dynamic graphs, but they cannot capture the long-term and short-term patterns of nodes, which is required for anomaly detection in dynamic graphs.

Research gap: Anomaly detection lies in insufficient labeled data. Even if the initial data is normal, over time, abnormal data will eventually be mixed with normal data. Labeling data is a big trouble for detection.

Contributions: 1. A semi-supervised abnormal edge detection framework AddGraph is proposed, which uses an attention-based GRU to extend the temporal GCN, which can combine the hidden state of long-term behavioral patterns with window information containing the short-term patterns of nodes.
2. Inspired by the knowledge graph embedding representation technology, a selective negative sampling strategy and marginal loss are introduced in the training of AddGraph. These strategies are used to handle situations where abnormal data labeling is insufficient.

3.2. Proposed solution

AddGraph utilizes GCN to process the previous node state of an edge in the current snapshot by considering the structural and content characteristics of the node. Then the context-sensitive attention model is used to summarize the node status information of the short window into short information. We input the output of GCN and short message into GRU to get the hidden state of the node at a new timestamp. The hidden state of the node at each timestamp is used to calculate the anomaly probability of an existing edge and a negative sampled edge, which is then fed back into the marginal loss.

Algorithm 1 AddGraph algorithm

Input: Edge stream $\{\mathcal{E}^t\}_{t=1}^T$
Parameter: $\beta, \mu, \lambda, \gamma, L, \omega, d$
Output: $\{\mathbf{H}^t\}_{t=1}^T$

Initialize $\mathbf{H}^0$
repeat
  for $t = 1$ to T **do**
    Let $\mathcal{L}^t = 0$
    $\mathbf{Current}^t = \mathrm{GCN}(\mathbf{H}^{t-1})$
    $\mathbf{Short}^t = \mathrm{CAB}(\mathbf{H}^{t-w}; ...; \mathbf{H}^{t-1})$
    $\mathbf{H}^t = \mathrm{GRU}(\mathbf{Current}^t, \mathbf{Short}^t)$
    for all $(i, j, w) \in \mathcal{E}^t$ **do**
      Sample(i', j', w) for $f(i, j, w)$
      $\mathcal{L}^t = \mathcal{L}^t + \max(0, \gamma + f(i, j, w) + f(i', j', w))$
    end for
    $\mathcal{L}^t = \mathcal{L}^t + \mathcal{L}_{reg}$
    Minimize $\mathcal{L}^t$
  end for
until Convergence
return $\{\mathbf{H}^t\}_{t=1}^T$

Figure 1. AddGraph Algorithm

Current^t: represents the current state of a node integrating the current input and long-term hidden state.
Short^t: represents window information that captures short-term patterns of nodes of interest.
GRU(Current^t*,Short*^{t}*)*: record long-term feature information while avoiding gradient disappearance and gradient explosion.

3.3. Claims-Evidence

Claim 1 AddGraph is able to catch temporal features.

Evidence: *Figure 2 in Section 4.2 from (Zheng et al., 2019)* shows AddGraph fits all the snapshots except the last one on Digg as the black line. Its AUC scores are close to 0.85 for the UCI Message dataset and 0.95 for the Digg dataset at the highest point.

Claim 2 As the number of newly generated points increases, the AUC of the algorithm will decrease because there are no previous features.

Evidence: *Figure 2 in Section 4.2 from (Zheng et al., 2019)* shows the tendency to increase at first and then decrease because AddGraph applies the long-term and short-term patterns successfully based on the GRU algorithm.

Claim 3 Different Layers will impact the AUC score.

Evidence: As shown in *Figure 3 in Section 4.2 from (Zheng et al., 2019)*, AUC increases significantly when L increases from 1 to 2 and reaches its peak when L is 3.

3.4. Critique and Discussion

AddGraph has been able to detect long-term and short-term outliers, but what still needs to be improved is that as time increases, some of its important features may be replaced. In any case, it achieves the optimal solution compared to other algorithms.

4. Review of GNN-based Anomaly Detection

4.1. Storyline

High-level motivation/problem: Existing methods do not explicitly learn the structure of existing relationships between variables, nor use them to predict the expected behavior of time series, so a Graph Deviation Network (GDN) is proposed, which aims to learn the relationship between sensors in the form of a graph. relationships between them, and then identify and explain deviations in learning patterns.

Prior work on this problem: Deep learning has made progress in anomaly detection in high-dimensional data sets, however existing methods do not explicitly learn the structure of existing relationships between variables or use them to predict the behavior of time series.

Research gap: First, different sensors have very different performance. Second, the relationships between sensors are initially unknown and need to be learned using our model.

Contributions: GDN, an attention-based graph neural network, uses it to learn the relationship dependence graph between sensors and identify and explain the deviation of these relationships.

4.2. Proposed solution

GDN methods aim to learn the relationships between sensors in the form of graphs and then identify and explain deviations from the learned patterns. GDN consists of four main parts. Sensor embedding uses embedding vectors to flexibly capture the unique characteristics of each sensor and learn a feature vector representation of each sensor. Graph structure learning learns the relationship between pairs of sensors (i.e. learns the graph structure representation of the dependencies between sensors) and encodes them as edges in the graph. Predict the future behavior of the sensor based on the attention function for adjacent sensors in the graph, that is, predict the next value of each sensor. Graph deviation score identifies and explains biases in sensor relationships learned from graphs, and localizes and explains these biases. *Figure 1 in Section 3 from (Deng & Hooi, 2021)* shows the GDN in graph.

4.3. Claims-Evidence

Claim 1 Removing the attention mechanism has the greatest impact on model performance.

Evidence: *Table 3 in Section 4.5 from (Deng & Hooi, 2021)* shows that when disabling the attention mechanism and using the same weight to aggregate all neighbors, the accuracy is dropped 28 percentage points compared with the full GDN.

Claim 2 The features learned by GDN reflect the effectiveness of local sensor behavioral similarities.

Evidence: *Figure 2 in Section 4.6 from (Deng & Hooi, 2021)* shows that there are Exhibit local clustering in projected 2D space. Also, the claim is supported by performing similar functions in the WADI water distribution network.

Claim 3 GDN's predictions of the expected behavior of each sensor allow people to understand how anomalies deviate from expectations.

Evidence: *Figure 3(Right) in Section 4.8 from (Deng & Hooi, 2021)* shows that GDN helps localize anomalies by comparing predicted and observed sensor values to understand individual anomalies.

4.4. Critique and Discussion

It is very reasonable to use directed graphs in the graph structure learning part, but the similarity calculation and top-k composition method are used. The similarity calculation is symmetrical and obviously cannot achieve the desired directed graph effect.

5. Implementations

5.1. Implementation motivation

I want to learn this new local outlier detection method via GNN because I am not familiar with GNN. Implementing GNN will help me to understand how it works. By understanding how to do outlier detection, I can apply it to many areas. What I hope to learn is that I can detect network intrusions, such as illegally stealing customer information and obtaining sensitive file information. Also, detecting anomalies can also help me learn how to prevent financial fraud, and maybe even make my own fraud detection software. Understanding outlier detection facilitates learning in many areas. In this paper, I expect to optimize LUNAR by implementing my own GNN class. To design this GNN model, train it, and evaluate it, all the work is going to give me more deep learning on not only model building but also local outlier detection working.

5.2. Implementation setup and plan

I plan to design my own GNN model, and I will reuse the author's model as a model comparison. My goal is to surpass the performance of the original LUNAR model. By trying different GNN model designs and different optimization algorithms, I hope to optimize the LUNAR model designed by the author.

Dataset The datasets that I will use are going to be the provided datasets (https://odds.cs.stonybrook.edu/) by the author. From the website that provides the datasets, we can see the size of each data set and the proportion of outliers it contains. For example, the MNIST data set has a total of 7603 points, its dimension is 100, and the proportion of outliers is 9.2%, which is 700. I will use the MNIST data set to complete the initial testing and implementation of my code. After my code is completed, the datasets I will test are in *Table 1*. Each dataset consists of a normal class (0) and an anomaly class (1). We will use the Area-Under-Curve (AUC) to measure the performance of each dataset.

Code LUNAR uses graph neural networks to construct a data set into a graph. Consider a single sample data as a point on the way, and then define the sample point as the target node and its nearest neighbor node as the source node.

Dataset	#points	#dim.	#outliers(%)
PENDIGITS	6870	16	156(2.27%)
MUSK	3062	166	97(3.2%)
MAMMOGRAPHY	11183	6	260(2.32%)
SHUTTLE	49097	9	3511(7%)
THYROID	3772	6	93(2.5%)
SATELLITE	6435	36	2036(32%)
MNIST	7603	100	700(9.2%)

Table 1. Dataset information

A target node is described by k source nodes. LUNAR uses a learnable aggregation method for k neighbor nodes. I will reuse the author's code, which constructs the dataset into a graph. I will also use some of the author's original parameters, such as learning rate and number of training epochs. I will write my own test function so that it can accept all the features I plan to implement.

The code will be based on Python and https://github.com/agoodge/LUNAR

Training My GNN module has the same hidden layer size 256 as the author's build. However, my output layer will be LogSigmoid() function and there are reasons detailed in section 5.3. For the loss function, I will reuse the mean squared error as the loss function because there is no obvious difference between using MSE and MAE. Both performances on MINIST are almost the same. The optimizer is AdamW with a learning rate of 0.001 and weight decay of 0.1 since there will be a small improvement in the performance. The total epochs are 200.

Evaluation Because MSE and MAE do not have a clear difference, the evaluation of the model will be using MSE as the loss function for the predicted y label.

Plot I plan to extend the plot feature for LUNAR. When running with the model, I want to see the loss function tendency and score tendency in each epoch, so the plot is an important feature to implement for visualization. I will go into detail in section 5.3.

Priority of implementation efforts I will build a GNN model and find the best optimizer as the highest priority for my implementation. Then, the plot feature comes to the next priority implementation. Splitting the train function and test function into two Python files and designing the other helper Python files are my lowest priority work because I need to make sure my model will work after implementing features.

5.3. Implementation details

GNN model build When replacing Tanh() by ReLU():
1. ReLU() drops a little bit for score.
2. Softmax() gives Warnig, and score is 50.00. *(Figure 2)*
3. LogSigmoid() gives a score of about 92.5. Better than Sigmoid(), Sigmoid() gives 91.5.

```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/module.py:1518: U
serWarning: Implicit dimension choice for softmax has been deprecated. Cha
nge the call to include dim=X as an argument.
  return self._call_impl(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/module.py:1518: U
serWarning: Implicit dimension choice for softmax has been deprecated. Cha
nge the call to include dim=X as an argument.
  return self._call_impl(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/module.py:1518: U
serWarning: Implicit dimension choice for softmax has been deprecated. Cha
nge the call to include dim=X as an argument.
  return self._call_impl(*args, **kwargs)
Dataset: MNIST
Samples: MIXED   Score: 50.0000
Runtime: 10.99 seconds
```

Figure 2. Softmax activate function has poor performance

When replacing Tanh() by Mish():
Using Mish() dropped a lot compared with the other two at first, but gets 92.8 after a few train.py calls.

When use Tanh():
1. Better than ReLU() with LogSigmoid(). LogSigmoid() reaches 93.2
2. Using LogSigmoid() is better than Sigmoid(). Sigmoid() reaches 90.

I also determined the best-hidden layers by adjusting the size and number of hidden layers. Below is my conclusion to this:

- Adding 5 hidden layers affects the score. The score dropped to about 90.
- Adding 4 hidden layers affects the score. The score dropped to about 80.
- If there is only one hidden layer, the score is going to be lower than 70.

Thus, three hidden layers are the best choice. Also, I have investigated the hidden layer size.

- Hidden layer size changes to 512 leading to the best score of about 91.
- Hidden layer size changes to 128 leading to the best score of about 86.

If I map three hidden layers with different sizes, the score is not increased, so keeping all the layers with the same size 256 is optimal.

Thus, my initial GNN model just changed the last activate function (i.e. Sigmoid()) to LogSigmoid().

Loss function There are two loss functions that can be used to evaluate the LUNAR model.

- **nn.L1Loss:** Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y.
- **nn.MSELoss:** Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.

I tried these two loss functions for evaluating my own LUNAR model, but the performance of the model gave me almost the same AUC score. Hence, I use L1Loss function, which is MAE loss, because a reliable system or metric should exhibit minimal sensitivity to outliers. In this context, one can infer that Mean Squared Error (MSE) might be less resilient compared to Mean Absolute Error (MAE) because squaring the errors in MSE accentuates the influence of outliers.

Optimizer There are two optimization methods that can be used to evaluate the LUNAR model. They are Adam and AdamW. The implementations of Adam and AdamW are different.

In Adam source code, weight decay is implemented as

$$grad = grad.add(parameters, alpha = weight_decay$$

but in AdamW source code, weight decay is implemented as

$$parameters.mul_(1 - lr * weight_decay)$$

During each iteration in Adam, the gradient undergoes an update using the estimated parameters from the previous iteration, with the influence of weight decay factored in. Conversely, in AdamW, the parameters are updated using the parameters from the preceding iteration, considering the impact of weight decay. The distinction is evident in the provided pseudocode in *figure 3* from the documentation, where the weight decay is highlighted for emphasis.

Plot First, we need to store the score value and loss value generated in each epoch from the training process into a list, and then return the optimal model parameters and these two list values through the $train$ function. If we add the $plot$ parameter when calling the $test$ function, then we will draw the score and loss values for each epoch through the **matplotlib** package. It should be noted that our list of loss values may be a tensor list, so we need to convert it to numpy format first. Finally, we will store this image in a

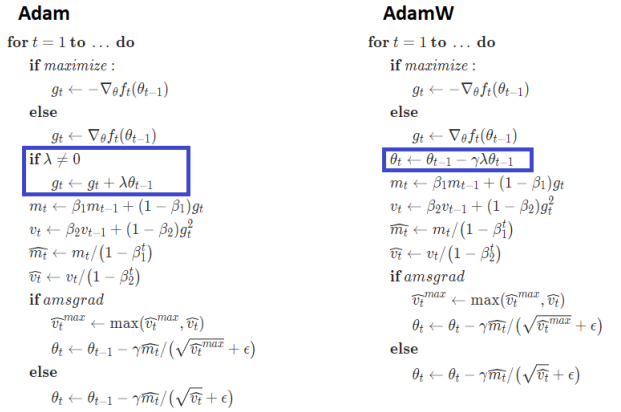

Figure 3. Algorithm of Adam and AdamW

folder called plot. For example, if our data set is MNIST, k is 100, and seed is 42, then the final file will be stored in ./plot/MNIST/training_metrics_plot_100_42.png.

By visualizing the loss and score values, we can judge whether the training results of the model are optimal.

Test Compared with the original LUNAR model testing method, I added two new parameters. The first parameter is $plot$, which is the visual training result mentioned in the previous section. The plot parameter defaults to false, which means no graph is drawn. Only when we add this parameter, the training result graph will be stored in the specified folder. The second parameter is the choosing model, which defaults to 1, which means the default is the model we reimplemented, whereas 0 is the original LUNAR model. The $models$ parameter is added because it can run the code better and compare the differences in my re-implementation.

However, I need to emphasize that the original LUNAR model does not have the $plot$ parameter, so no need to add the $plot$ parameter when running the code if need to test the original LUNAR model, otherwise an error will be reported.

I also have a new implementation that if we do not specify k, then we will use this data set to run all k, and k is 2, 50, 100, 150, 200, and 300 respectively. The $samples$ parameter is MIXED as I keep the author's conclusion.

5.4. Results and interpretation

We now conduct the experiments with real datasets to answer the following research questions?

1. Does the loss value and score value converge?
2. Is our model better than the original LUNAR model?
3. What are the final scores for each dataset we test?

For the questions 1 and 2, we use the MNIST dataset as an example. The parameters are: k=100, seed=42, and samples=MIXED.

Does the loss value and score value converge? The answer is **YES**. The loss value and score value converge. The *figure 4* shows two line graphs, each representing a different metric across training epochs for our LUNAR model.

The top graph is labeled "Score across Epochs" and shows the training score of the model. The x-axis represents the epoch number, ranging from 0 to 200. The y-axis represents the score, with values starting just above 0.735 and rising slightly before stabilizing around 0.739. The score increases rapidly in the first few epochs, then levels off, indicating the model quickly reached a stable performance and didn't significantly improve after the initial epochs.

The bottom graph is labeled "Loss across Epochs" and depicts the training loss of the model. Similar to the top graph, the x-axis represents the epoch number from 0 to 200, while the y-axis represents the loss, starting from just above 0.70 and dropping sharply to below 0.50 within the first 25 epochs. The loss decreases rapidly at the beginning and then flattens out, suggesting that the model's ability to minimize the loss plateaued early during training.

Overall, both graphs indicate that our LUNAR model learned most of what it could from the data in the early epochs, with little to no improvement in score or reduction in loss as training continued. These initial gains are quickly made, but improvements become marginal as the model converges to its optimal state.

Is our model better than the original LUNAR model? The answer is **YES**. Because I optimized the last activation function of the GNN, and I tried different loss functions and optimizers, I found the situation where the model performed best.

From the *figure 4*, there are two outputs. The first one is the original LUNAR model. Given the random seed is 42, k is 100, and samples is MIXED, the score is 88.6882. However, our model has the score 93.1735.

I have tried to run many times given the same parameters, and the scores keep changing every time, but in most cases our model performance will be higher.

What are the final scores for each dataset we test? In this part, I am going to test all the scores for each k and each dataset and compare the performance of two models, one is original model and another one is my own LUNAR model. 2, 10, 50, 100, 150, 200, 300

Table 2 shows all the performance of datasets with our LUNAR model. Compared with the performance of the

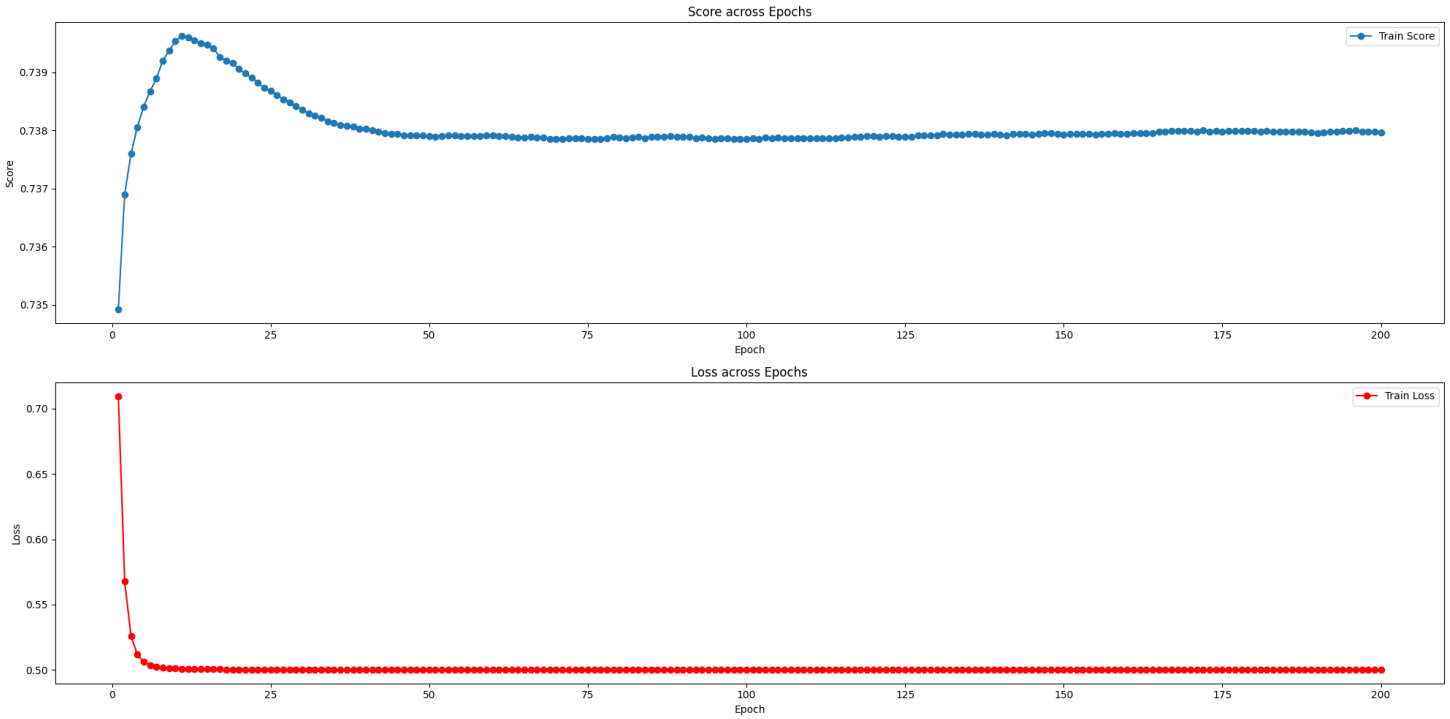

Figure 4. Score and Loss tendency across 200 epochs

Dataset/k	2	50	100	150	200	300	Average
PENDIGITS	99.9671	99.9384	99.8233	99.6918	99.5809	99.2973	99.7165
MUSK	100	100	100	100	100	99.2454	99.8742
MAMMOGRAPHY	76.1043	88.2803	87.9697	87.6117	87.4564	87.3913	85.8023
THYROID	89.4786	98.3582	98.1732	98.0229	97.9651	97.9073	96.6509
SHUTTLE	99.9818	99.8811	99.7800	99.7093	99.5964	99.4911	99.7400
SATELLITE	88.8361	87.1647	87.0037	86.7944	86.4622	85.8479	87.0182
MNIST	87.4592	92.1852	89.5218	90.9941	85.9250	90.5165	89.4336

Table 2. Final AUC scores for different values of k and the Avg. over all k.

```
/content/drive/MyDrive/ColabNotebooks/ECE570/LUNAR# python test.py --datas
et MNIST --samples MIXED --k 100 --seed 42 --train_new_model --models 0
Running trial with random seed = 42
Running trial with k = 100
GNN1
Dataset: MNIST
Samples: MIXED   Score: 88.6882
Runtime: 8.48 seconds
/content/drive/MyDrive/ColabNotebooks/ECE570/LUNAR# python test.py --datas
et MNIST --samples MIXED --k 100 --seed 42 --train_new_model --models 1
Running trial with random seed = 42
Running trial with k = 100
Our own GNN
Dataset: MNIST
Samples: MIXED   Score: 93.1735
Runtime: 8.48 seconds
```

Figure 5. The comparison between our model and original model

original LUNAR model, the common datasets for testing are **PENDIGITS,SATELLITE,SHUTTLE,THYROID**. Based on the *Table 4* from (Goodge et al., 2022), the performance are:

PENDIGITS: 99.79

SATELLITE: 86.08

SHUTTLE: 99.96

THYROID: 85.31

Now, we compare the average AUC score with *Table 2*. We have almost the same performance on the dataset **PENDIGITS** and **SHUTTLE**. For the dataset **SATELLITE**, there are some minor improvements in our model performance. For the dataset **THYROID**, the performance of our model

has been significantly improved, reaching nearly 100 points.

6. Conclusion and Discussion

In conclusion, by learning and implementing the LUNAR model code, I have a deeper understanding of GNN and outlier detection. By comparing the code at each step to find the optimal performance, I also learned more about the optimizer, loss function, and model building.

LUNAR provides a framework that unifies all outlier detection methods. Based on this framework, it can learn and adapt to different graph neural network data sets. This unique learning ability is not available in today's outlier detection methods. In my future work, I may use these anomaly detection to detect vulnerabilities or security detection. The LUNAR model has given me great help on my journey of learning artificial intelligence.

References

Deng, A. and Hooi, B. Graph neural network-based anomaly detection in multivariate time series. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pp. 4027–4035, 2021.

Goodge, A., Hooi, B., Ng, S.-K., and Ng, W. S. Lunar: Unifying local outlier detection methods via graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pp. 6737–6745, 2022.

Zheng, L., Li, Z., Li, J., Li, Z., and Gao, J. Addgraph: Anomaly detection in dynamic graph using attention-based temporal gcn. In *IJCAI*, volume 3, pp. 7, 2019.