Graph Neural Networks with PyG on Node Classification, Link Prediction, and Anomaly Detection

Pytorch Geometric Implementations on major graph problems

Tomonori Masui · Follow
Published in Towards Data Science · 10 min read · Oct 6, 2022

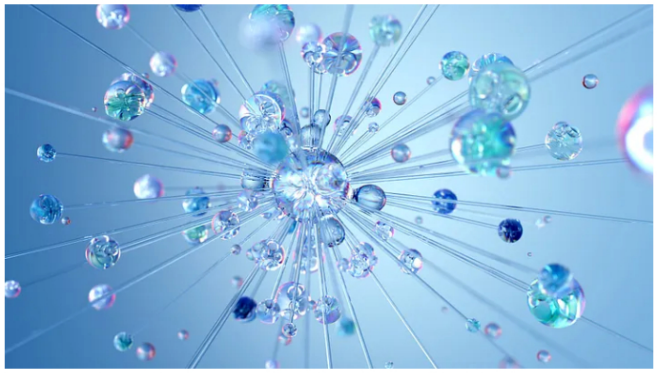


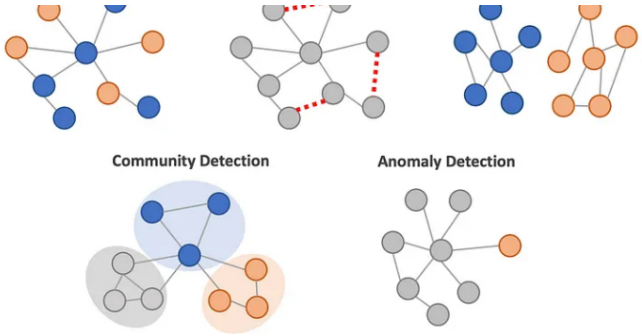
Photo by DeepMind on Unsplash

Graph Neural Networks is a machine learning algorithm designed for graph-structured data such as social graphs, networks in cybersecurity, or molecular representations. It has evolved rapidly over the last few years and is used in many different applications. In this blog post, we will review its code implementations on major graph problems along with all the basics of GNN including its applications and algorithm details.

Applications of Graph Neural Networks

GNN can be used to solve a variety of graph-related machine learning problems:

- **Node Classification**
  Predicting the classes or labels of nodes. For example, detecting fraudulent entities in the network in cybersecurity can be a node classification problem.
- **Link Prediction**
  Predicting if there are potential linkages (edges) between nodes. For example, a social networking service suggests possible friend connections based on network data.
- **Graph Classification**
  Classifying a graph itself into different categories. An example is determining if a chemical compound is toxic or non-toxic by looking at its graph structure.
- **Community Detection**
  Partitioning nodes into clusters. An example is finding different communities in a social graph.
- **Anomaly Detection**
  Finding outlier nodes in a graph in an unsupervised manner. This approach can be used if you don't have labels on your target.



GNN Applications (Image by author)

In this blog post, we will review code implementations on node classification, link prediction, and anomaly detection.

Graph Convolution — Intuition

Graph Neural Networks evolved rapidly over the last few years and many variants of it have been invented (you can see this survey for more details). In those GNN variations, Graph Convolutional Networks might be the most popular and basic algorithm. In this section, we will review the high-level introduction to its algorithm.

Graph Convolution is an effective way to extract/summarize node information based on a graph structure. It is a variant of the convolution operation from Convolutional Neural Networks which is typically used for image problems.

In images, pixels are ordered structurally in a grid, and the filters (weight matrices) in the convolution operation slide over the image with a pre-determined stride size. The neighbors of a pixel are determined by the filter size (in the below image, the filter size is 3 x 3 and the eight gray pixels within the blue filter are the neighbors), and weighted pixel values within the filter are aggregated into a single value. The output from this convolution operation has a smaller size than the input image but has a higher level view of the input which is going to be useful for making predictions on image problems such as image classifications.

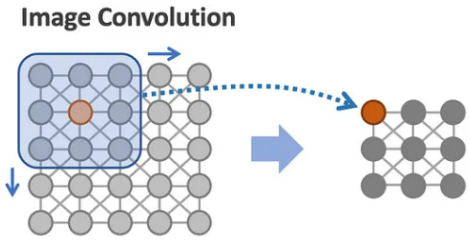


Image by author

In graphs, nodes are ordered in a nonstructural manner and the neighborhood size differs across nodes. Graph convolution takes the average of the node features of a given node (the red node in the below image) and its neighbors (the gray nodes within the blue circle) to calculate updated node representation values of the node. Through this convolution operation, the node representation captures localized graph information.

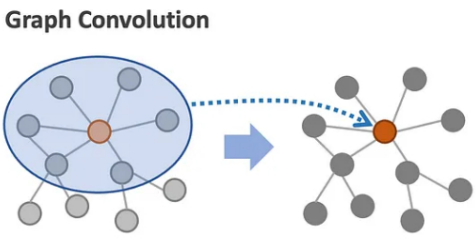


Image by author

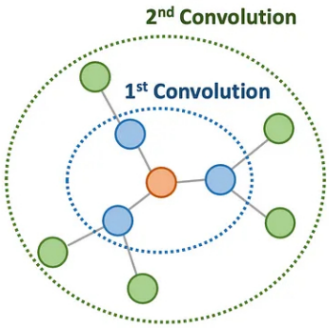
The image below shows more details on the graph convolution operation. Node features from neighborhood nodes (blues) along with those of the target node (red) are averaged. And then it is multiplied with a weight vector (W) and its output updates the node features of the target node (the updated node values are also called node embeddings).

Graph Convolution Operation (Image by author)

For those who are interested, the node features are normalized using the inverse of the degree matrix and then aggregated in the original paper instead of simple averaging (equation (8) in the paper).

Top highlight

One thing to note in this convolution operation is that the number of graph convolutions determines how many steps away node features are aggregated into each node. In the image below, the first convolution aggregates the blue nodes' features into the orange node and the second convolution can incorporate the green nodes' features into the orange node.



The number of convolutions determines how far node features you aggregate (Image by author)

Cora — Graph Benchmark Dataset

In the next few sections, we will review GCN code implementations. Before we dive into them, let us get familiar with the dataset we are going to use. The Cora dataset is a paper citation network data that consists of 2,708 scientific publications. Each node in the graph represents each publication and a pair of nodes is connected with an edge if one paper cites the other.

Through this article, we are using PyG (Pytorch Geometric) to implement GCN which is one of the popular GNN libraries. The Cora dataset can also be loaded using PyG module:

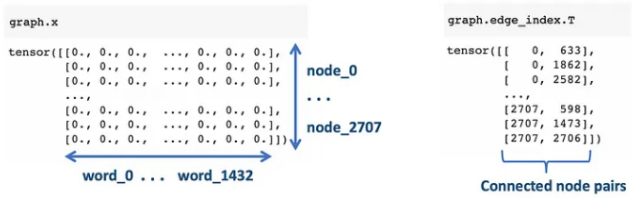
```
from torch_geometric.datasets import Planetoid

dataset = Planetoid(root='/tmp/Cora', name='Cora')
graph = dataset[0]
```

load_cora.py hosted with ♥ by GitHub view raw

The Cora dataset sourced from Pytorch Geometric is originally from the "Automating the Construction of Internet Portals with Machine Learning" paper.

The node features and the edge information look like below. The node features are 1433 word vectors indicating the absence (0) or the presence (1) of the words in each publication. The edges are represented in adjacency

lists.



Node Features and Edge Lists (Image by author)

Each node has one of seven classes which is going to be our model target/label.

```
"""
Class Definition
0: Theory
1: Reinforcement_Learning
2: Genetic_Algorithms
3: Neural_Networks
4: Probabilistic_Methods
5: Case_Based
6: Rule_Learning
"""

print("Class Distribution:")
sorted(Counter(graph.y.tolist()).items())
```

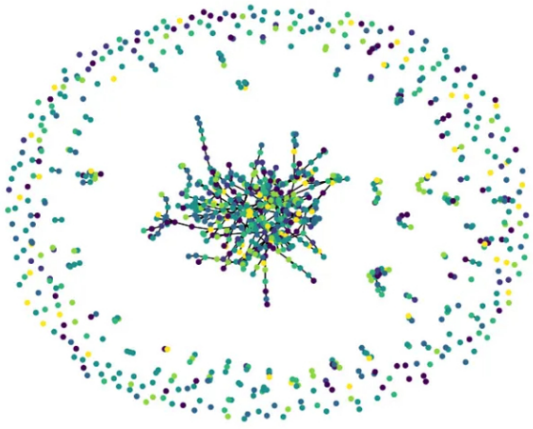
```
Class Distribution:
[(0, 351), (1, 217), (2, 418), (3, 818), (4, 426), (5, 298), (6, 180)]
```

Class Distribution (Image by author)

The graph data can be visualized using NetworkX library. The node colors represent the node classes.

```
import random
from torch_geometric.utils import to_networkx
import networkx as nx

def convert_to_networkx(graph, n_sample=None):

    g = to_networkx(graph, node_attrs=["x"])
    y = graph.y.numpy()

    if n_sample is not None:
        sampled_nodes = random.sample(g.nodes, n_sample)
        g = g.subgraph(sampled_nodes)
        y = y[sampled_nodes]

    return g, y


def plot_graph(g, y):

    plt.figure(figsize=(9, 7))
    nx.draw_spring(g, node_size=30, arrows=False, node_color=y)
    plt.show()


g, y = convert_to_networkx(graph, n_sample=1000)
plot_graph(g, y)
```

visualize_graph.py hosted with ♥ by GitHub view raw



Visualization of Cora Dataset (Image by author)

Node Classification

For the node classification problem, we are splitting the nodes into train, valid, and test using the `RandomNodeSplit` module from PyG (we are replacing the original split masks in the data as it has a too small train set).

```
import torch_geometric.transforms as T

split = T.RandomNodeSplit(num_val=0.1, num_test=0.2)
graph = split(graph)
```

split_nodes.py hosted with ❤ by GitHub — view raw

Please note the data splits are written into `mask` attributes in the graph object (see the image below) instead of splitting the graph itself. Those masks are used for training loss calculation and model evaluation, whereas graph convolutions use entire graph data.

```
graph
```

```
Data(x=[2708, 1433], edge_index=[2, 10556], y=[2708], train_mask=[2708], val_mask=[2708], test_mask=[2708])
```

Graph object has mask attributes (Image by author)

Baseline Model (MLP) on Node Classification

Before we build GCN, we are training MLP (multi-layer perceptron, i.e. feed-forward neural nets) only using node features to set a baseline performance. The model ignores the node connections (or the graph structure) and tries to classify the node labels only using the word vectors. The model class looks like below. It has two hidden layers (`Linear`) with ReLU activations followed by an output layer.

```
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
        nn.Linear(dataset.num_node_features, 64),
        nn.ReLU(),
        nn.Linear(64, 32),
        nn.ReLU(),
        nn.Linear(32, dataset.num_classes)
        )

    def forward(self, data):
        x = data.x  # only using node features (x)
        output = self.layers(x)
        return output
```

mlp_on_graph.py hosted with ❤ by GitHub — view raw

We are defining training and evaluation functions with a normal Pytorch train/eval setup.

```
def train_node_classifier(model, graph, optimizer, criterion, n_epochs=200):

    for epoch in range(1, n_epochs + 1):
        model.train()
        optimizer.zero_grad()
        out = model(graph)
        loss = criterion(out[graph.train_mask], graph.y[graph.train_mask])
        loss.backward()
        optimizer.step()

        pred = out.argmax(dim=1)
        acc = eval_node_classifier(model, graph, graph.val_mask)

        if epoch % 10 == 0:
            print(f'Epoch: {epoch:03d}, Train Loss: {loss:.3f}, Val Acc: {acc:.3f}')

    return model


def eval_node_classifier(model, graph, mask):

    model.eval()
    pred = model(graph).argmax(dim=1)
    correct = (pred[mask] == graph.y[mask]).sum()
    acc = int(correct) / int(mask.sum())
```

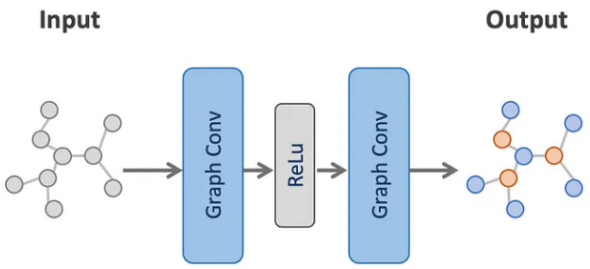
```
    return acc


mlp = MLP().to(device)
optimizer_mlp = torch.optim.Adam(mlp.parameters(), lr=0.01, weight_decay=5e-4)
criterion = nn.CrossEntropyLoss()
mlp = train_node_classifier(mlp, graph, optimizer_mlp, criterion, n_epochs=150)

test_acc = eval_node_classifier(mlp, graph, graph.test_mask)
print(f'Test Acc: {test_acc:.3f}')
```

train_node_classifier.py hosted with ❤ by GitHub view raw

The test accuracy from this model is **73.2%**.

GCN on Node Classification

Next, we are training GCN and comparing its performance to MLP. We are using a very simple model having two graph convolution layers and ReLU activation between them. This setup is the same as the original GCN paper (equation 9).



GCN Node Classification Model Architecture (Image by author)

```
from torch_geometric.nn import GCNConv
import torch.nn.functional as F

class GCN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = GCNConv(dataset.num_node_features, 16)
        self.conv2 = GCNConv(16, dataset.num_classes)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index

        x = self.conv1(x, edge_index)
        x = F.relu(x)
        output = self.conv2(x, edge_index)

        return output


gcn = GCN().to(device)
optimizer_gcn = torch.optim.Adam(gcn.parameters(), lr=0.01, weight_decay=5e-4)
criterion = nn.CrossEntropyLoss()
gcn = train_node_classifier(gcn, graph, optimizer_gcn, criterion)

test_acc = eval_node_classifier(gcn, graph, graph.test_mask)
print(f'Test Acc: {test_acc:.3f}')
```

gcn_node_classifier.py hosted with ❤ by GitHub view raw

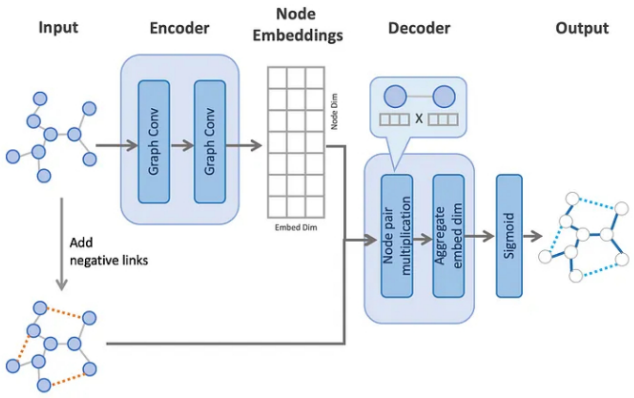
The test-set accuracy from this model is **88.0%**. We achieved around 15% accuracy improvement from MLP.

Link Prediction

Link prediction is trickier than node classification as we need some tweaks to make predictions on edges using node embeddings. The prediction steps are described below:

1. An encoder creates node embeddings by processing the graph with two convolution layers.
2. We randomly add negative links to the original graph. This makes the model task binary classifications with the positive links from the original edges and the negative links from the added edges.
3. A decoder makes link predictions (i.e. binary classifications) on all the

edges including the negative links using node embeddings. It calculates a dot product of the node embeddings from pair of nodes on each edge. Then, it aggregates the values across the embedding dimension and creates a single value on every edge that represents the probability of edge existence.



Link Prediction Model Architecture (Image by author)

This model structure is from the original link prediction implementation in Variational Graph Auto-Encoders. The code looks like something below. This is adapted from the code example in PyG repo which is based on the Graph Auto-Encoders implementation.

```python
from sklearn.metrics import roc_auc_score
from torch_geometric.utils import negative_sampling


class Net(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels):
        super().__init__()
        self.conv1 = GCNConv(in_channels, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, out_channels)

    def encode(self, x, edge_index):
        x = self.conv1(x, edge_index).relu()
        return self.conv2(x, edge_index)

    def decode(self, z, edge_label_index):
        return (z[edge_label_index[0]] * z[edge_label_index[1]]).sum(
            dim=-1
        )  # product of a pair of nodes on each edge

    def decode_all(self, z):
        prob_adj = z @ z.t()
        return (prob_adj > 0).nonzero(as_tuple=False).t()


def train_link_predictor(
    model, train_data, val_data, optimizer, criterion, n_epochs=100
):

    for epoch in range(1, n_epochs + 1):

        model.train()
        optimizer.zero_grad()
        z = model.encode(train_data.x, train_data.edge_index)

        # sampling training negatives for every training epoch
        neg_edge_index = negative_sampling(
            edge_index=train_data.edge_index, num_nodes=train_data.num_nodes,
            num_neg_samples=train_data.edge_label_index.size(1), method='sparse')

        edge_label_index = torch.cat(
            [train_data.edge_label_index, neg_edge_index],
            dim=-1,
        )
        edge_label = torch.cat([
            train_data.edge_label,
            train_data.edge_label.new_zeros(neg_edge_index.size(1))
        ], dim=0)

        out = model.decode(z, edge_label_index).view(-1)
        loss = criterion(out, edge_label)
        loss.backward()
        optimizer.step()

        val_auc = eval_link_predictor(model, val_data)

        if epoch % 10 == 0:
```

```
            print(f"Epoch: {epoch:03d}, Train Loss: {loss:.3f}, Val AUC: {val_auc:.3f}")

    return model


@torch.no_grad()
def eval_link_predictor(model, data):

    model.eval()
    z = model.encode(data.x, data.edge_index)
    out = model.decode(z, data.edge_label_index).view(-1).sigmoid()

    return roc_auc_score(data.edge_label.cpu().numpy(), out.cpu().numpy())
```

link_predictor.py hosted with ❤ by GitHub view raw

For this link prediction task, we want to randomly split links/edges into train, valid, and test data. We can use the `RandomLinkSplit` module from PyG to do that.

```
import torch_geometric.transforms as T

split = T.RandomLinkSplit(
    num_val=0.05,
    num_test=0.1,
    is_undirected=True,
    add_negative_train_samples=False,
    neg_sampling_ratio=1.0,
)
train_data, val_data, test_data = split(graph)
```

split_links.py hosted with ❤ by GitHub view raw

The output data looks like something below.

```
print('train_data:', train_data)
print('val_data:', val_data)
print('test_data:', test_data)
```

```
train_data: Data(x=[2708, 1433], edge_index=[2, 8976], y=[2708], edge_label=[4488], edge_label_index=[2, 4488])
val_data: Data(x=[2708, 1433], edge_index=[2, 8976], y=[2708], edge_label=[526], edge_label_index=[2, 526])
test_data: Data(x=[2708, 1433], edge_index=[2, 9502], y=[2708], edge_label=[1054], edge_label_index=[2, 1054])
```

RandomLinkSplit Output (image by author)

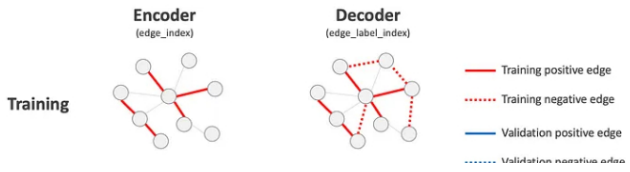
There are several things to note about this output data.

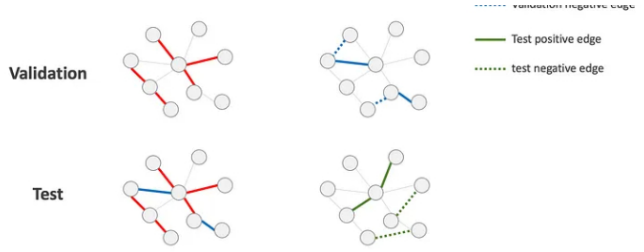
First, the split is performed on `edge_index` such that the training and the validation splits do not include the edges from the validation and the test split (i.e. only have the edges from the training split), and the test split does not include the edges from the test split. This is because `edge_index` (and `x`) is used for the encoder to create node embeddings, and this setup ensures that there are no target leaks on the node embeddings when it makes predictions on the validation/test data.

Second, two new attributes (`edge_label` and `edge_label_index`) are added to each split data. They are the edge labels and the edge indices corresponding to each split. `edge_label_index` will be used for the decoder to make predictions and `edge_label` will be used for model evaluation.

Third, negative links are added to both `val_data` and `test_data` with the same number as the positive links (`neg_sampling_ratio=1.0`). They are added to `edge_label` and `edge_label_index` attributes, but not added to `edge_index` as we do not want to use the negative links on the encoder (or node embedding creation). And also, we are not adding negative links to the training set here (by setting `add_negative_train_samples=False`) as we will add them during the training loop in `train_link_predictor` above. This randomization during training is supposed to make the model more robust.

The image below summarizes how this edge split is performed for the encoder and the decoder (the colored edges are used in each stage).





Summary of Edge Split (image by author)

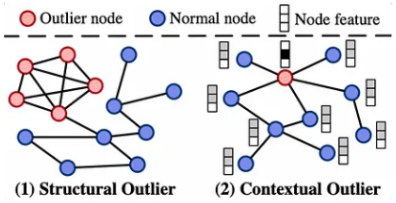
We can now train and evaluate the model with the following code.

The test AUC of this model is 92.5%.

Anomaly Detection

We are again using Cora dataset for the anomaly detection task, but it is slightly different from the previous one: the one with outliers being synthetically injected. The dataset has two different types of outliers (the outlier definition is from this paper):

- **Structural Outlier**
  Densely connected nodes in contrast to sparsely connected regular nodes
- **Contextual Outlier**
  Nodes whose attributes are significantly different from their neighboring nodes



Node Outlier Types (Source: https://arxiv.org/pdf/2206.10071.pdf)

For this anomaly detection task, we are using PyGOD library which is a graph outlier detection library built on top of PyG. We can load the outlier-injected Cora dataset via the PyGOD module.

```
0: inlier
1: contextual outlier only
2: structural outlier only
3: both contextual outlier and structural outlier
"""

Counter(graph.y.tolist())
```

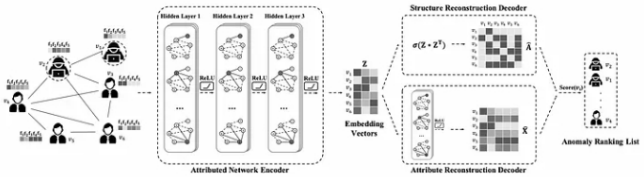
outlier_distribution.py hosted with ♥ by GitHub view raw

Output: `Counter({0: 2570, 1: 68, 2: 68, 3: 2})`

If you are interested in how these outliers are injected, you can look at the PyGOD documentation on the outlier generator modules which explains the operation details. Please note that the labels `y` will only be used for model evaluation and not be used for training labels as we are training an unsupervised model.

To detect those outliers, we are training `DOMINANT` (Deep Anomaly Detection on Attributed Networks) model from this paper. It is an auto-encoder network with graph convolution layers and its reconstruction errors are going to be the node anomaly scores. The model follows the steps below to make predictions.

1. ***The attributed network encoder*** processes the input graph with three graph convolution layers that create node embeddings.
2. ***The structure reconstruction decoder*** reconstructs the original graph edges (i.e. adjacency matrix) using the learned node embeddings. It calculates a dot product of node embeddings from every possible pair of nodes that creates a probability score on each node pair indicating edge existence.
3. ***The attribute reconstruction decoder*** reconstructs the original node attributes using the obtained node embeddings. It has a graph convolution layer to predict the attribute values.
4. In the last step, the reconstruction errors from the above two decoders are combined by weighted average on every node and the combined errors are going to be the final errors/losses. These final errors are also the abnormality scores of the nodes.



DOMINANT Model Architecture (Source: Deep Anomaly Detection on Attributed Networks)

The `DOMINANT` model can be easily implemented with PyGOD as you can see below.

```
8
9       outlier_scores = model.decision_function(graph)
10      auc = roc_auc_score(graph.y.numpy(), outlier_scores)
11      ap = average_precision_score(graph.y.numpy(), outlier_scores)
12      print(f'AUC Score: {auc:.3f}')
13      print(f'AP Score: {ap:.3f}')
14
15
16  graph.y = graph.y.bool()
17  model = DOMINANT()
18  model = train_anomaly_detector(model, graph)
19  eval_anomaly_detector(model, graph)
```

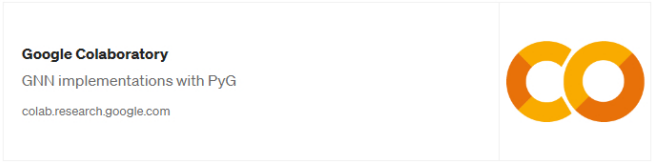
graph_anomaly_detector.py hosted with ❤ by GitHub view raw

The AUC from this model is 84.1%, whereas its average precision is 20.8%. This difference is most likely due to the target imbalance. As this is an unsupervised model, we might not be able to expect a very accurate model, but you can see in the original paper that it still outperforms any other popular anomaly detection algorithms.

. . .

That is it for this article!

If you are interested, the whole code is available in the Google Colab and the GitHub repo below.



References

- Benjamin Sanchez-Lengeling et al., A Gentle Introduction to Graph Neural Networks
- Ameya Daigavane et al., Understanding Convolutions on Graphs
- Francesco Casalegno, Graph Convolutional Networks — Deep Learning on Graphs
- Thomas N. Kipf, Max Welling, Semi-Supervised Classification with Graph Convolutional Networks (2017), ICLR 2017
- Thomas N. Kipf, Max Welling, Variational Graph Auto-Encoders (2016)
- Kaize Ding et al., Deep Anomaly Detection on Attributed Networks (2019)
- Kay Liu et al., Benchmarking Node Outlier Detection on Graphs (2022)

Graph Neural Networks | Machine Learning | Data Science | Graph Convolution Network | Deep Dives

273 3

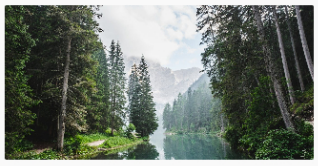
Written by Tomonori Masui

Follow

363 Followers · Writer for Towards Data Science

Data Scientist | Machine Learning https://www.linkedin.com/in/tomonori-masui/

More from Tomonori Masui and Towards Data Science



Tomonori Masui in Towards Data Science

All You Need to Know about Gradient Boosting Algorithm –...

Algorithm explained with an example, math, and code

10 min read · Jan 20, 2022

703 7

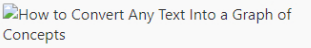
Marco Peixeiro in Towards Data Science

TimeGPT: The First Foundation Model for Time Series Forecasting

Explore the first generative pre-trained forecasting model and apply it in a project...

12 min read · Oct 24

2.2K 21



Rahul Nayak in Towards Data Science

How to Convert Any Text Into a Graph of Concepts

A method to convert any text corpus into a Knowledge Graph using Mistral 7B.

12 min read · Nov 10

1.4K 22

Tomonori Masui in Towards Data Science

Entity Resolution: Identifying Real-World Entities in Noisy Data

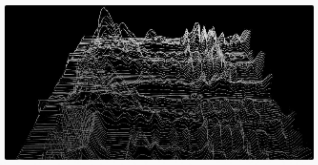
Fundamental theories and Python implementations

19 min read · Sep 21

326 1

See all from Tomonori Masui | See all from Towards Data Science

Recommended from Medium



Cristian Urbinati in Data Reply IT | DataTech

Spatio-Temporal Forecasting using Temporal Graph Neural Networks

Application of Spatio-Temporal GNN for traffic forecasting on a custom dataset...

8 min read · Jun 5

79

Rubens Zimbres in Google Developer Experts

Graph Neural Networks: the message passing algorithm

Graph Neural Networks are a type of neural networks used in data presented as graphs....

6 min read · Jun 5

37

Lists

Predictive Modeling w/ Python
20 stories · 604 saves

Practical Guides to Machine Learning
10 stories · 683 saves

Natural Language Processing
863 stories · 403 saves

New_Reading_List
174 stories · 196 saves

Brackly Murunga

Predicting a customer's next purchase using graph neural...

Part I

5 min read · Jun 4

2

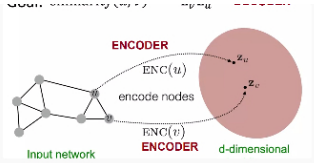
Tracyrenee in MLearning.ai

Use NetworkX to make predictions on nodes in Python

I had been working on the first Kaggle's first Covid Volunteer community competition, an...

9 min read · Jun 26

70



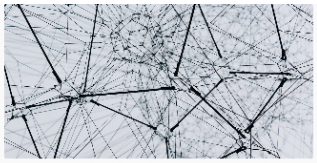
Faxi Yuan, PhD

Graph Neural Networks (GNNs): Comparison between CNNs and...

Based on my previous story on message passing framework for node classifications, I...

· 7 min read · May 29

8 1



Guangyuan(Frank) Li

Network Analysis in Python

A comprehensive introduction and hands-on practice of graph theory and network analysis

7 min read · Jun 20

18

See more recommendations