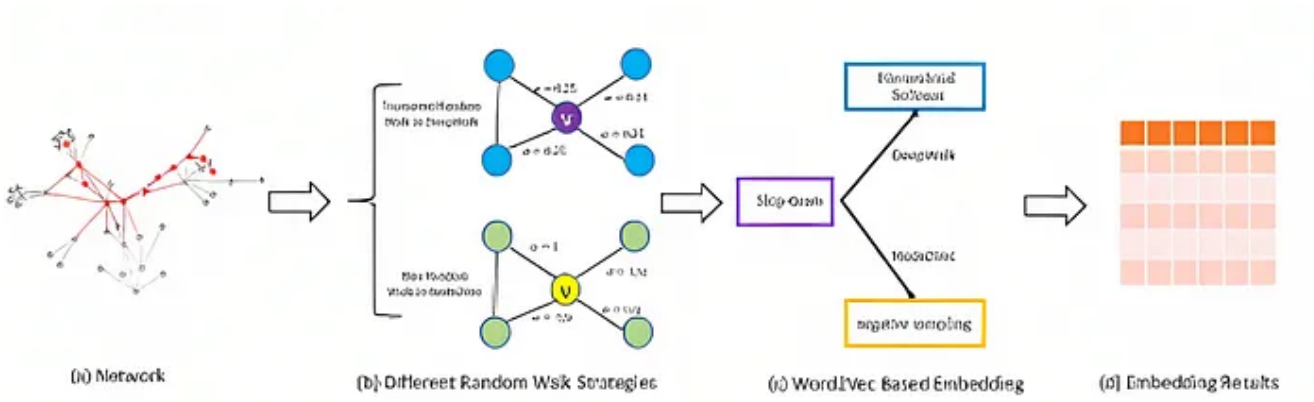
Deep Walk and Node2Vec: Graph Embeddings

Tejpal Kumawat · Following
6 min read · Mar 14

Investigating Node2Vec and DeepWalk to extract embeddings from graphs



Graph

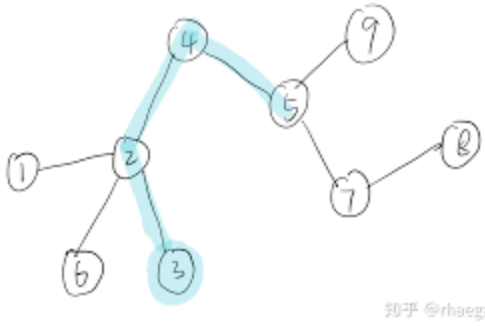
G = (V, E), where V is a collection of nodes and E is a list of edges, can be used to define a graph. A link between two nodes is called an edge. For instance, an edge connects nodes A and D. It's also crucial to remember that a graph can be either directed or undirected. For instance, the graph below is undirected because A and D are linked together, and vice versa. Another

issue is that a graph can have various node and edge qualities, although for our purposes today, neither are significant.

Why we need Node Embeddings

- In a social network, users communicate with one another, and we must forecast when two users will link. Users are represented by nodes, whereas relationships between individuals are shown by edges. We need to forecast the topics of each research article in a network of citations (link prediction task).
- Publications are represented by nodes, while citations between publications are represented by edges. (Task of Node Prediction)
- There are specific proteins that fall under the categories of enzymes and non-enzymes. The amino acids are represented by nodes, and if two nodes are less than 6 Angstroms apart, they are connected by an edge. (Task of graph categorization)

Deep Walk



DeepWalk was presented by Stony Brook University researchers in the paper "DeepWalk: Online Learning of Social Representations" (2014).

DeepWalk is a method for learning representations of nodes in a graph. The main idea behind DeepWalk is to generate random walks in the graph and use these random walks to learn representations of nodes using the Word2Vec algorithm.

In more detail, given a graph, DeepWalk first generates a set of random walks starting from each node in the graph. Each random walk is simply a sequence of nodes that starts at a particular node and moves to one of its neighbors at each step. These random walks capture the local structure of the graph and provide a way to learn representations of nodes that capture their relationships with nearby nodes.

Next, the Word2Vec algorithm is used to learn representations of nodes based on the generated random walks. Word2Vec is a popular algorithm for learning representations of words in natural language processing, but it can also be applied to learn representations of nodes in a graph. The basic idea behind Word2Vec is to learn a neural network that predicts the probability of a word given its context (i.e., the words that appear before and after it).

In the case of DeepWalk, the random walks generated from the graph serve as the “sentences” and the nodes in the graph serve as the “words”. The Word2Vec algorithm is then used to learn a neural network that predicts the probability of a node appearing in a random walk given its neighbors in the graph. The learned representations of nodes can then be used for a variety of downstream tasks, such as node classification or link prediction.

Overall, DeepWalk is a powerful and widely used method for learning representations of nodes in graphs that captures the local structure of the graph.

```
import networkx as nx
import random
import numpy as np
from typing import List
from tqdm import tqdm
from gensim.models.word2vec import Word2Vec


class DeepWalk:
    def __init__(self, window_size: int, embedding_size: int, walk_length: int,
        """
        :param window_size: window size for the Word2Vec model
        :param embedding_size: size of the final embedding
        :param walk_length: length of the walk
        :param walks_per_node: number of walks per node
        """
        self.window_size = window_size
        self.embedding_size = embedding_size
        self.walk_length = walk_length
        self.walk_per_node = walks_per_node

    def random_walk(self, g: nx.Graph, start: str, use_probabilities: bool = Fal
        """
        Generate a random walk starting on start
        :param g: Graph
        :param start: starting node for the random walk
        :param use_probabilities: if True take into account the weights assigned
        :return:
        """
        walk = [start]
        for i in range(self.walk_length):
            neighbours = g.neighbors(walk[i])
            neighs = list(neighbours)
            if use_probabilities:
                probabilities = [g.get_edge_data(walk[i], neig)["weight"] for ne
                sum_probabilities = sum(probabilities)
                probabilities = list(map(lambda t: t / sum_probabilities, probab
                p = np.random.choice(neighs, p=probabilities)
            else:
                p = random.choice(neighs)
            walk.append(p)
        return walk

    def get_walks(self, g: nx.Graph, use_probabilities: bool = False) -> List[Li
        """
        Generate all the random walks
        :param g: Graph
        :param use_probabilities:
```

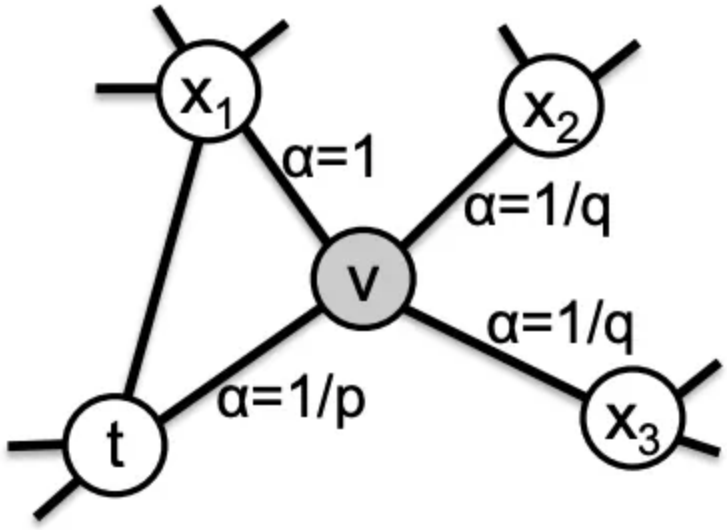
```
            :return:
            """
            random_walks = []
            for _ in range(self.walk_per_node):
                random_nodes = list(g.nodes)
                random.shuffle(random_nodes)
                for node in tqdm(random_nodes):
                    random_walks.append(self.random_walk(g=g, start=node, use_probab
            return random_walks

        def compute_embeddings(self, walks: List[List[str]]):
            """
            Compute the node embeddings for the generated walks
            :param walks: List of walks
            :return:
            """
            model = Word2Vec(sentences=walks, window=self.window_size, vector_size=s
            return model.wv
```

Node2Vec

Node2Vec was presented by Stanford University researchers in the paper: "node2vec: Scalable Feature Learning for Networks" (2016).

While using some of Deepwalk's concepts, this method takes them a step farther. To extract the random walks, it combines the techniques DFS and BFS. Two parameters, P (return parameter) and Q, govern this combination of algorithms (in-out parameter).

In essence, if P is big, the random walks will be big, so it explores, and if P is little, we stick close to home. Q exhibits a similar but opposite behaviour; if Q is little, it will engage in exploration, whereas if Q is large, it will remain local. The original paper contains further information.



Node2vec's sampling strategy accepts 4 arguments:
— **Number of walks**: Number of random walks to be generated from each node in the graph
— **Walk length**: How many nodes are in each random walk
— **P**: Return hyperparameter
— **Q**: Inout hyperparameters
and also the standard skip-gram parameters (context window size, number of iterations, etc.)

We can use PyTorch geometric to test Node2Vec. To expedite the use of GNN, this package implements a variety of graph neural network topologies and techniques. I'm going to test it out using a small section of the Pytorch geometric instruction. They make use of the Cora dataset for that. The 2708 scientific publications in the Cora dataset are divided into seven categories. There are 5429 links in the network of citations. A 0/1-valued word vector denoting the absence/presence of the matching word from the dictionary is used to describe each publication in the dataset. 1433 distinct words make up the dictionary.

```
from torch_geometric.nn import Node2Vec
import os.path as osp
import torch
from torch_geometric.datasets import Planetoid
from tqdm.notebook import tqdm

dataset = 'Cora'
path = osp.join('.', 'data', dataset)
dataset = Planetoid(path, dataset)  # dowload or load the Cora dataset
data = dataset[0]
device = 'cuda' if torch.cuda.is_available() else 'cpu'  # check if cuda is avai
model = Node2Vec(data.edge_index, embedding_dim=128, walk_length=20,
                 context_size=10, walks_per_node=10,
                 num_negative_samples=1, p=1, q=1, sparse=True).to(device)

loader = model.loader(batch_size=128, shuffle=True, num_workers=4)  # data loade
optimizer = torch.optim.SparseAdam(list(model.parameters()), lr=0.01)  # initzia


def train():
    model.train()  # put model in train model
    total_loss = 0
    for pos_rw, neg_rw in tqdm(loader):
        optimizer.zero_grad()  # set the gradients to 0
        loss = model.loss(pos_rw.to(device), neg_rw.to(device))  # compute the l
        loss.backward()
        optimizer.step()  # optimize the parameters
        total_loss += loss.item()
    return total_loss / len(loader)


for epoch in range(1, 100):
    loss = train()
    print(f'Epoch: {epoch:02d}, Loss: {loss:.4f}')

all_vectors = ""
for tensor in model(torch.arange(data.num_nodes, device=device)):
    s = "\t".join([str(value) for value in tensor.detach().cpu().numpy()])
    all_vectors += s + "\n"
# save the vectors
with open("vectors.txt", "w") as f:
    f.write(all_vectors)
# save the labels
with open("labels.txt", "w") as f:
    f.write("\n".join([str(label) for label in data.y.numpy()]))
```

Reference

- For skip-gram and word2vec refer to this video
- [1] https://en.wikipedia.org/wiki/Graph_theory
- [2] Graph Machine Learning: Take Graph Data to the Next Level by Applying Machine Learning Techniques and Algorithms by Aldo Marzullo, Claudio Stamile, and Enrico Deusebio
- [3] https://pypi.org/project/node2vec/
- [4] https://arxiv.org/pdf/1607.00653.pdf

Deepwalk | Node2vec | Graph | Graph Neural Networks | Deep Learning

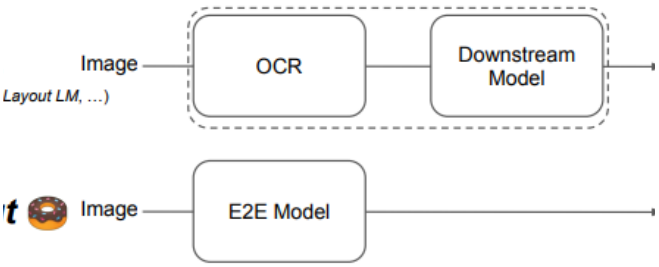
Written by Tejpal Kumawat

Following

199 Followers

Artificial Intelligence enthusiast that is constantly looking for new challenges and researching cutting-edge technology to improve the world !!!!!!!!!!!!!!!!!!

More from Tejpal Kumawat



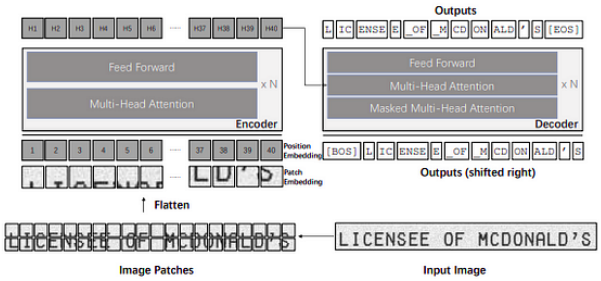
Tejpal Kumawat

Donut: OCR-Free Document Understanding with Donut

Introduction

9 min read · Mar 6

46 2



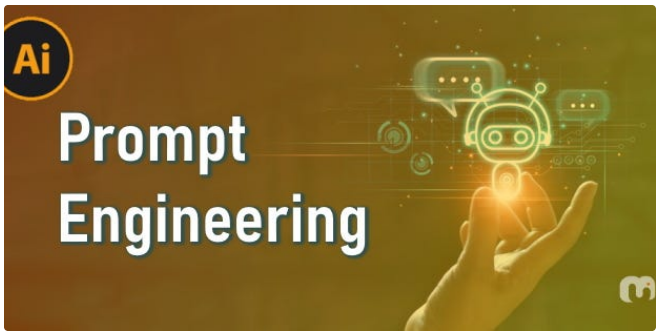
Tejpal Kumawat

TrOCR — Transformer-based Optical Recognition Model

Introduction

5 min read · Mar 5

36



Tejpal Kumawat

Basics of Prompt Engineering

What is LLMs ?

30 min read · Jun 4

12

Tejpal Kumawat

Is CNN Extincted? Transformer-Based Vison Models Explained →...

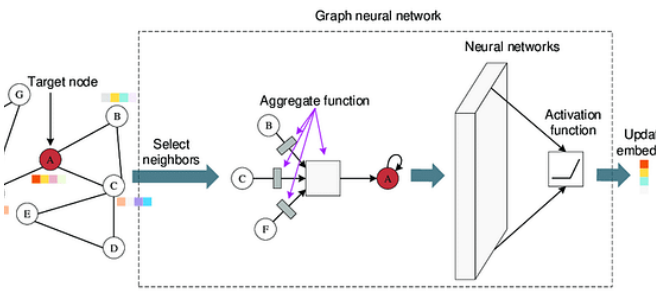
Introduction

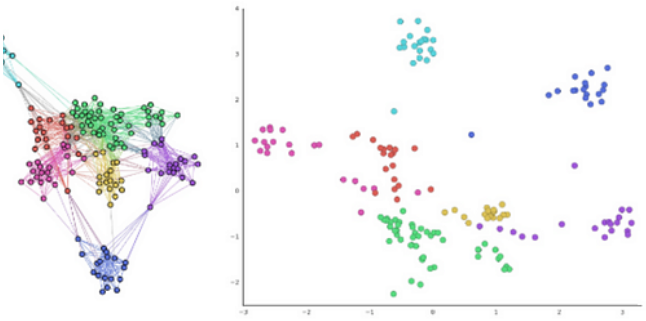
11 min read · Apr 5

36

See all from Tejpal Kumawat

Recommended from Medium





Sahil Sheikh

Exploring SageConv: A Powerful Graph Neural Network...

Graph Neural Networks (GNNs) are a class of deep learning models that are designed to...

3 min read · May 1

14

Artem Shlezinger

Graph Embedding: Nonlinear Optimization

Simplified approach for finding vertex vectors

4 min read · Jul 30

9 1

Lists

Natural Language Processing

669 stories · 283 saves

Practical Guides to Machine Learning

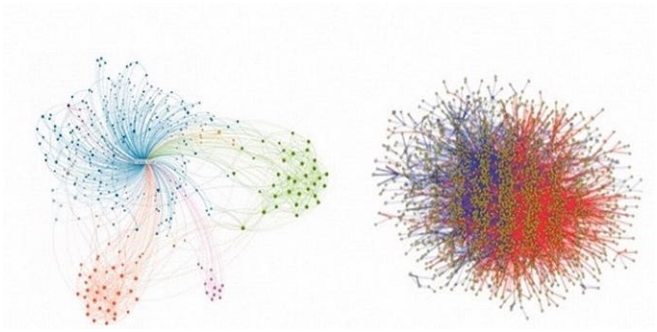
10 stories · 519 saves

New_Reading_List

174 stories · 133 saves

Staff Picks

465 stories · 317 saves

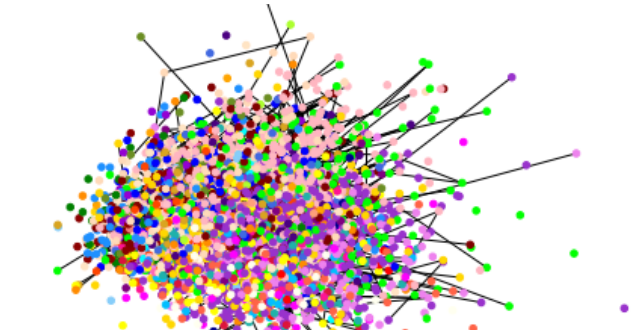


Lei Wang in graphscope

Graphs and Graph Applications

In this post, we will introduce basic concepts of graphs, and some typical applications of...

7 min read · May 22



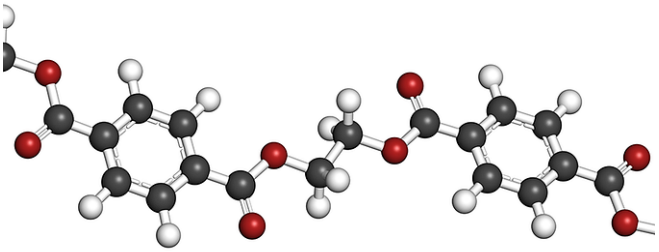
Yunqi Li

Citation Network Prediction with Graph Neural Networks

By Xixuan Julie Liu, Yihe Tang, Yunqi Li as part of Stanford CS224W course project(Winter...

14 min read · Jun 29

7



Thomas Little

Block Copolymer Phase Prediction using Graph Neural Network...

By Thomas Little and Jorge Martinez Alba as part of the Stanford CS224W course project

11 min read · May 26

0xdevshah in AI Skunks

Knowledge Graphs: A Comprehensive Analysis

Knowledge graphs are becoming increasingly important in NLP due to their ability to mode...

8 min read · Apr 9

63

See more recommendations