NetRA Documentation

---pytorch version

Wei Cheng([weicheng@nec-labs.com](mailto:weicheng@nec-labs.com))

This is the document of NetRA for pytorch version. For technical details, please refer to the paper:

Learning Deep Network Representations with Adversarially Regularized Autoencoders.  
Wenchao Yu, Cheng Zheng, Wei Cheng, Charu Aggarwal, Dongjing Song, Bo Zong, Haifeng Chen, Wei Wang. *The Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD’18)*, 2018.

The entry of running is ./src/train.py. It conducts embedding of nodes in a static network (karate network) and visualizes the 2-dimensional embedding results of nodes. The html docs for each python file are included in ./doc\_html. To better understand the framework, the reader is supposed to be familiar with [**pytorch**](https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html) framework and [**Wasserstein GAN**](https://arxiv.org/pdf/1701.07875.pdf).

**Overview**

The problem of network embedding arises in many machine learning tasks with the assumption that there may exist a small number of variabilities in the vertex representations which can capture the “semantics” of the original network structure. Most existing network embedding models, with shallow or deep architectures, perform to learn discrete vertex representations by maintaining the locality-preserving property and/or global reconstruction capability. The resultant discrete representations have proven to be difficult for model generalization because of the sparsity of the sampled walks derived from the input networks. Ideally, we could learn the vertex representations regularized by a prior distribution to avoid this problem. However, in a common situation, it’s not the case that the prior distribution will exist in a low dimensional manifold. In this study, we handle the aforementioned challenge by proposing to learn the network embedding with adversarial regularization, NETAR for short, which doesn’t need to pre-define an explicit density distribution for the hidden representations, but still can represent distributions confined to a low dimensional manifold. In this framework, the vertex representations are learned through both locality-preserving and global reconstruction constraints, and regularized by generative adversarial training. NETAR learns smooth regularized vertex representations while still being able to capture the network structure of the underlying network structure. This is guaranteed by the joint minimization of the network embedding loss and the auto-encoder reconstruction error through generative adversarial training process. FIG. 1 illustrates the framework of NETAR. Basically, the upper layer part is the deep auto-encoder for learning the low dimensional embedding of graph information. The bottom layer is the generative adversarial part to generator negative samples for the discriminator to distinguish from positive embedding in the auto-encoder graph embedding.



FIG. 1. The model of NETRA

The workflow diagram is shown in FIG. 2.

Input a network data

Low dimensional representations of nodes as positive samples

Truncated random walks rooted on each node

LSTM Auto-encoding

Random walk generator

Final network embedding vectors

01

00

04

10

05

GAN discriminator

GAN generator

03

09

Negative fake node embedding

07

08

FIG. 2. The workflow of the proposed NETAR method for network embedding

The tree diagram of each module in FIG. 2 is shown below.

**00**

Input a network data

Input a network data

**04**

**01**

LSTM Auto-encoding

Random walk generator

**041**

**111**

With the input of graph G(V, E), the random walk generator in [10]

is utilized to obtain truncated random walks rooted on each nodes

in the graph.

By mapping the input to latent space, deep autoencoder provides efficient low dimensional representation of the input data

**001**

**001**

The network can be either direct or undirected

**112**

A walk sample randomly from the neighbors of the last visited node until the preset maximum length is reached. The random walk generation procedure preserves the localized property of graph, which provides information corpus for the following GANAE training.

**09**

**07**

GAN discriminator

GAN generator

**091**

**071**

**001**

The discriminator updates when comparing the latent space from autoencoder and fake sample from generator.

The generator generates fake embedding vectors

092

**072**

As the concurrent update of encoder and discriminator, the latent space of autoencoder provide optimal embedding for the nodes in the network.

We use LSTM in encoder and decoder networks [5] because of its power in language models.

FIG. 3. Tree diagram of each module

**System Requirements**

We need python 2.7 and torch 0.3.1.

**1). Install using the**[**Homebrew**](https://brew.sh/)**package manager:**

**Mac:**

/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

export PATH="/usr/local/bin:/usr/local/sbin:$PATH"

brew update

brew install python@2  # Python 2

sudo pip install -U virtualenv  # system-wide install

**Ubuntu:**

sudo apt update

sudo apt install python-dev python-pip

sudo pip install -U virtualenv  # system-wide install

**2). Create a new virtual environment by choosing a Python interpreter**

virtualenv --system-site-packages -p python2.7 ***./venv***

source ***./venv***/bin/activate

pip install --upgrade pip

**3). install torch version 0.3.1**

pip install torch==0.3.1

**4). install packages**

pip install networkx

pip install scipy

**Code Input/Output**

**Input:**

The input is the linked list format of graph.

For example, the ./data/ karate.adjlist is with following format:

1 2 3 4 5 6 7 8 9 11 12 13 14 18 20 22 32

2 1 3 4 8 14 18 20 22 31

3 1 2 4 8 9 10 14 28 29 33

.

.

.

format:

startNodeID endNodeID1 endNodeID2 …..

**output:**

1. One output is the embeddings of each node, each node is one line vector

The embeddings of nodes after each epoch are save in src/output/example/

Vis() function will visualize the final epoch output embeddings.

1. Another output is the visualization of node embedding, as shown in FIG. 10.

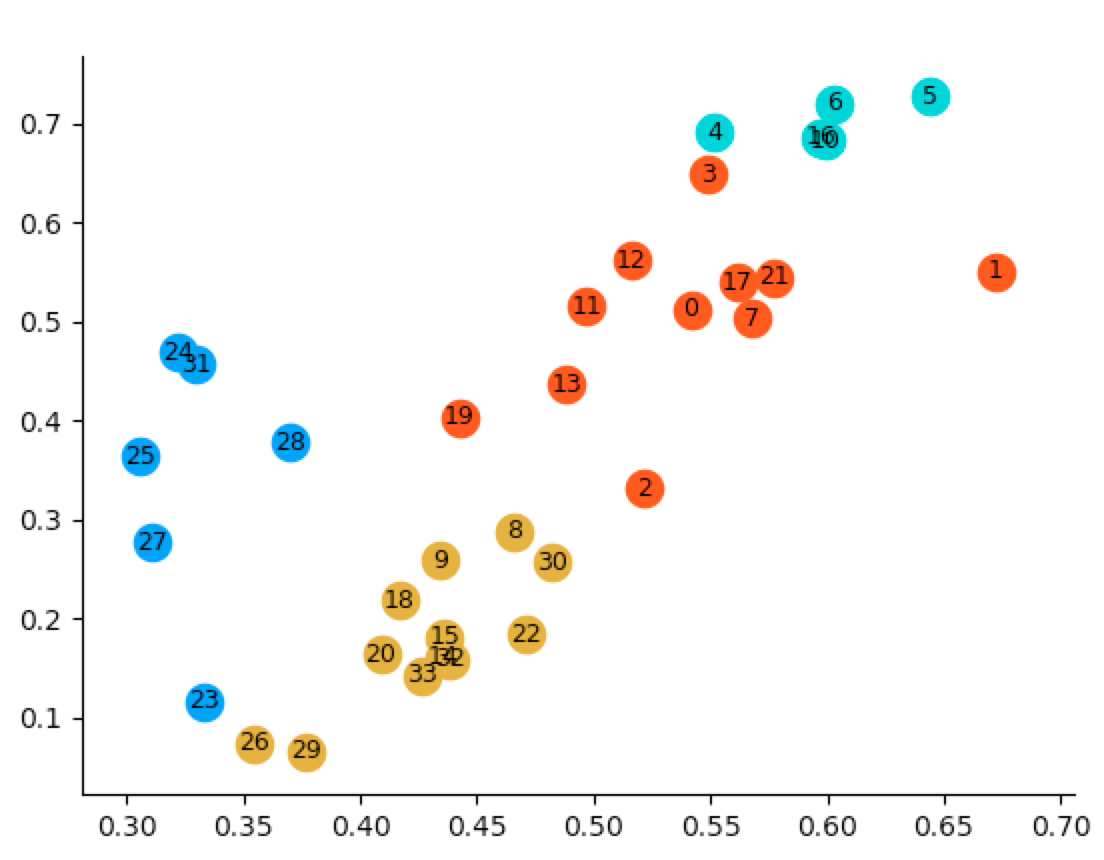


FIG. 10 2-dimensional embedding of karate network

**Code Structure**

train.py

----- the main entrance of running;

----- it defines different hyper parameters to use; these hyper parameters are with default values;

----- by **$**python train.py you can run the example with karate network;

----- train.py first generates walks for training, then define LSTM AE module and GAN modules, together with the optimization of them. Then for loop epochs for feeding batches to optimizing model parameters. The ebeddings after each epoch is stored in src/output/example/

The hyper-parameters can be pass in the command line, such as:

**$**python --nhidden 5 --epoch 100 train.py

**This will modify the model to embed nodes to 5 dimensions with epoch number 100**

"""Parameters to parse  
 Path Arguments: The input and output directory  
 Data Processing Arguments: data preprocessing for generating ``walks'' from the graph  
 Model Arguments: parameters for the model  
 Training Arguments, Evaluation Arguments, and others like   
"""  
parser = argparse.ArgumentParser(description='NetRA')  
# Path Arguments  
parser.add\_argument('--data\_path', type=str, default='../data/karate.adjlist',  
 help='location of the data corpus') # location of the graph with linked list format  
parser.add\_argument('--outf', type=str, default='example',  
 help='output directory name') # location of output embeddings in different epochs  
  
# Data Processing Arguments  
parser.add\_argument('--maxlen', type=int, default=100,  
 help='maximum sentence length') # the parameter is for random walk to generating walks,  
 # this is the upbound of the walk length, in the code we generate walks with the same length  
  
  
# Model Arguments  
################### important hyper-parameters ################################  
parser.add\_argument('--nhidden', type=int, default=2,  
 help='number of hidden units per layer') # dimension of embedding vectors, since we want to visualize to 2-dimensional  
parser.add\_argument('--emsize', type=int, default=30,  
 help='size of word embeddings') # large graph 100-300, this is the size of input after original one hot embedding's mapping  
  
  
################### typically below are set to default ones ###################  
parser.add\_argument('--nlayers', type=int, default=1,  
 help='number of layers') # number of stacked LSTM for autoencoding  
parser.add\_argument('--noise\_radius', type=float, default=0.2,  
 help='stdev of noise for autoencoder (regularizer)') # stard deviation of noise for autoencoder  
parser.add\_argument('--noise\_anneal', type=float, default=0.995,  
 help='anneal noise\_radius exponentially by this'  
 'every 100 iterations') # decay rate for exponentially decaying noise on autoencoder  
parser.add\_argument('--hidden\_init', action='store\_true',  
 help="initialize decoder hidden state with encoder's")  
parser.add\_argument('--arch\_g', type=str, default='300-300',  
 help='generator architecture (MLP)') # specify the MLP structure of generator in GAN;  
 # for example, 300-300 means two layers, each layer includes 300 nodes  
parser.add\_argument('--arch\_d', type=str, default='300-300',  
 help='critic/discriminator architecture (MLP)') # specify the MLP structure of discriminator in GAN;  
 # for example, 300-300 means two layers, each layer includes 300 nodes  
parser.add\_argument('--z\_size', type=int, default=100,  
 help='dimension of random noise z to feed into generator') # random noise to be feed into the generator  
parser.add\_argument('--temp', type=float, default=1,  
 help='softmax temperature (lower --> more discrete)') # specify the temperature of softmax, \tau  
parser.add\_argument('--enc\_grad\_norm', type=bool, default=True,  
 help='norm code gradient from critic->encoder')  
parser.add\_argument('--gan\_toenc', type=float, default=-0.01,  
 help='weight factor passing gradient from gan to encoder') # weight factor passing from gradient of GAN to encoder, thi is used by grad\_hook  
parser.add\_argument('--dropout', type=float, default=0.0,  
 help='dropout applied to layers (0 = no dropout)') # dropout to prevent overfitting, by default, there is no dropout  
  
  
# Training Arguments  
################### important hyper-parameters ################################  
parser.add\_argument('--epochs', type=int, default=50,  
 help='maximum number of epochs') # epochs for training, usually small graph 50, large graph 100  
parser.add\_argument('--walk\_length', type=int, default=20,  
 help='length of walk sampled from the graph') # the length of walk sampled rooted from each node  
parser.add\_argument('--numWalks\_per\_node', type=int, default=30,  
 help='number of walks sampled for each node') # number of walks sampled for each node  
parser.add\_argument('--batch\_size', type=int, default=64, metavar='N',  
 help='batch size') # batch size for training  
parser.add\_argument('--niters\_ae', type=int, default=1,  
 help='number of autoencoder iterations in training') # in each epoch, number of iterations for training autoencoder  
parser.add\_argument('--niters\_gan\_d', type=int, default=5,  
 help='number of discriminator iterations in training') # in each epoch, number of iterations for training discriminator  
parser.add\_argument('--niters\_gan\_g', type=int, default=1,  
 help='number of generator iterations in training') # in each epoch, number of iterations for training generator  
  
parser.add\_argument('--niters\_gan\_schedule', type=str, default='2-4-6-10-20-30-40',  
 help='epoch counts to increase number of GAN training '  
 ' iterations (increment by 1 each time)') # in different epochs, dynamically increase the GAN iterations,  
 # for example, 2-4-6 means, 2 epochs then increase one, 4 epochs then increase again  
  
################### typically below are set to default ones ###################  
parser.add\_argument('--min\_epochs', type=int, default=6,  
 help="minimum number of epochs to train for") # minimum nuber of epochs for training  
parser.add\_argument('--no\_earlystopping', action='store\_true',  
 help="won't use KenLM for early stopping") # if conduct early stopping  
parser.add\_argument('--lr\_ae', type=float, default=1,  
 help='autoencoder learning rate') # learning rate for AE, because it is using SDG, by default it is 1  
parser.add\_argument('--lr\_gan\_g', type=float, default=5e-05,  
 help='generator learning rate') # learning rate for generator, because it is using ADM, by default it is a smaller one  
parser.add\_argument('--lr\_gan\_d', type=float, default=1e-05,  
 help='critic/discriminator learning rate') # learning rate for discriminator, because it is using ADM, by default it is a smaller one  
parser.add\_argument('--beta1', type=float, default=0.9,  
 help='beta1 for adam. default=0.9') # beta for adam  
parser.add\_argument('--clip', type=float, default=1,  
 help='gradient clipping, max norm') # gradient clipping  
parser.add\_argument('--gan\_clamp', type=float, default=0.01,  
 help='WGAN clamp') # WGAN clamp  
  
# Evaluation Arguments  
parser.add\_argument('--sample', action='store\_true',  
 help='sample when decoding for generation')  
parser.add\_argument('--log\_interval', type=int, default=200,  
 help='interval to log autoencoder training results')  
  
# Other  
parser.add\_argument('--seed', type=int, default=1111,  
 help='random seed') # random seeds for parameter initialization  
parser.add\_argument('--cuda', action='store\_true',  
 help='use CUDA') # use CUDA for training

models.py

----- define each modules of the model, such as MLP generator, MLP discriminator, LSTM Autoencoder

utils.py

----- codes to load data, preparing walks from the graph, preparing batches for training,

and building dictionary map between ids and nodes

viz\_karate.py

----- visualize the embedding vectors to 2-dimensional space.

**Important Parameters**

Basically, expect from file paths for input graph, here are some important parameters for model performance.

1). parser.add\_argument('--nhidden', type=int, default=2,  
 help='number of hidden units per layer') # dimension of embedding vectors, since we want to visualize to 2-dimensional

Usually, for larger graph, this should be typically 50 to 400. Here, we use 2 dimensions because the graph is small and we want to visualize it in 2-dimensional space.

2). parser.add\_argument('--emsize', type=int, default=30,  
 help='size of word embeddings') # large graph 100-300, this is the size of input after original one hot embedding's mapping

This is the size of input after one-hot embedding, typically for large graph 100-300 will be ok.

3). parser.add\_argument('--epochs', type=int, default=50,  
 help='maximum number of epochs') # epochs for training, usually small graph 50, large graph 100

The number of epochs for optimizing. Typically, 50-100 are enough.

4). parser.add\_argument('--walk\_length', type=int, default=20,  
 help='length of walk sampled from the graph') # the length of walk sampled rooted from each node

The length of walks to train LSTM. Typically, 20 is enough.

5). parser.add\_argument('--niters\_ae', type=int, default=1,  
 help='number of autoencoder iterations in training') # in each epoch, number of iterations for training autoencoder  
parser.add\_argument('--niters\_gan\_d', type=int, default=5,  
 help='number of discriminator iterations in training') # in each epoch, number of iterations for training discriminator  
parser.add\_argument('--niters\_gan\_g', type=int, default=1,  
 help='number of generator iterations in training') # in each epoch, number of iterations for training generator  
  
parser.add\_argument('--niters\_gan\_schedule', type=str, default='2-4-6-10-20-30-40',  
 help='epoch counts to increase number of GAN training '  
 ' iterations (increment by 1 each time)') # in different epochs, dynamically increase the GAN iterations, for example, 2-4-6 means, 2 epochs then increase one, 4 epochs then increase again

For good performance from GAN, the trick here is that training discriminator more times than generator, and gradually increasing the GAN training part more iterations compared with main model part.