NetRA Documentation
---pytorch version

Wei Cheng(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** framework and **Wasserstein GAN**.

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.

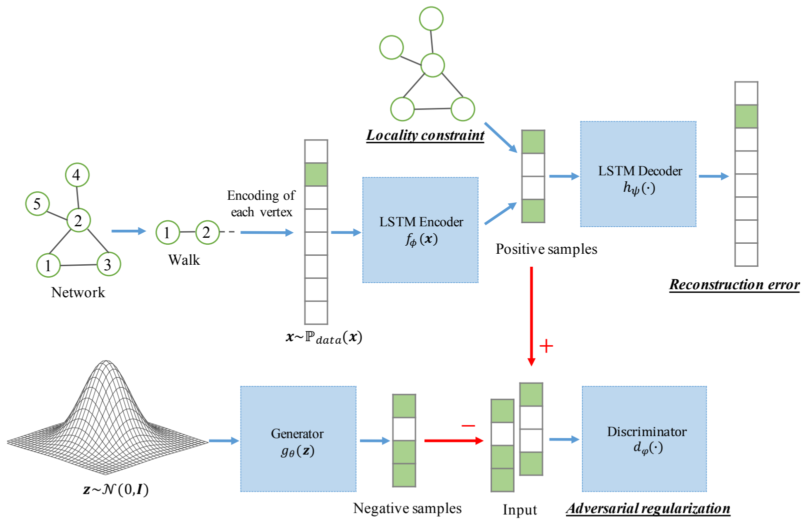

FIG. 1. The model of NETRA

The workflow diagram is shown in FIG. 2.

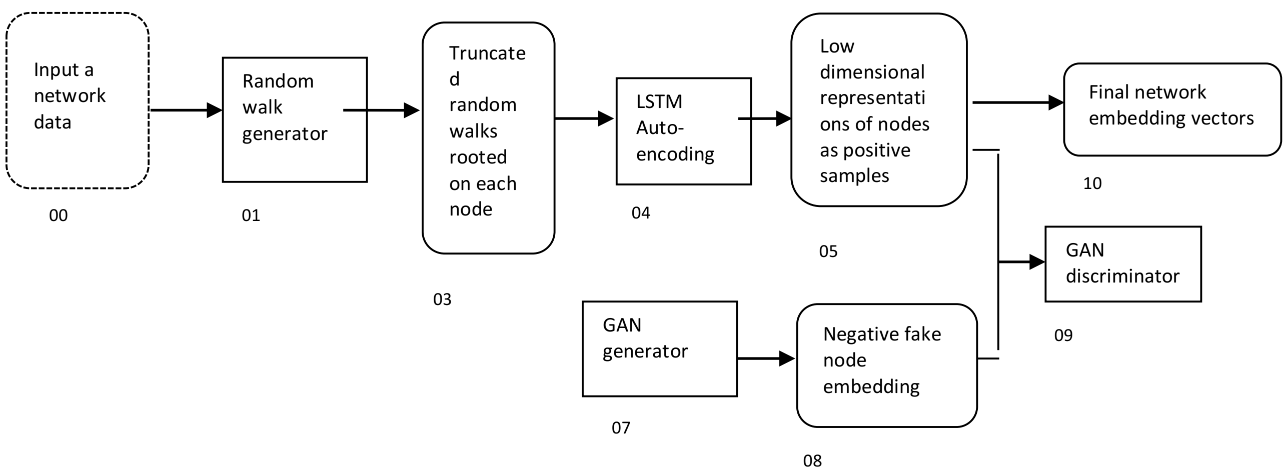

FIG. 2. The workflow of the proposed NETAR method for network embedding

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

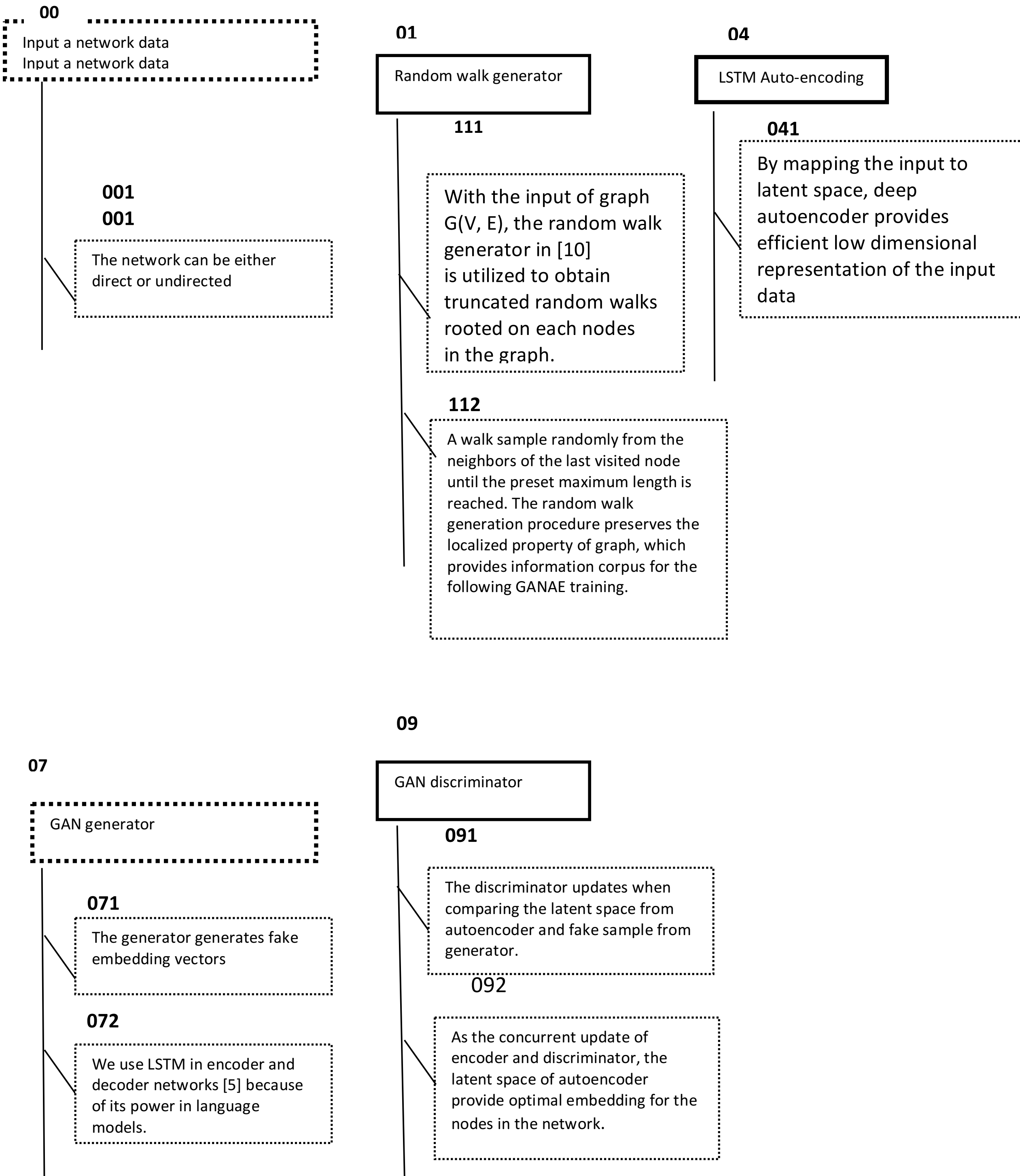

FIG. 3. Tree diagram of each module

System Requirements

We need python 2.7 and torch 0.3.1.

1). Install using the Homebrew 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.

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

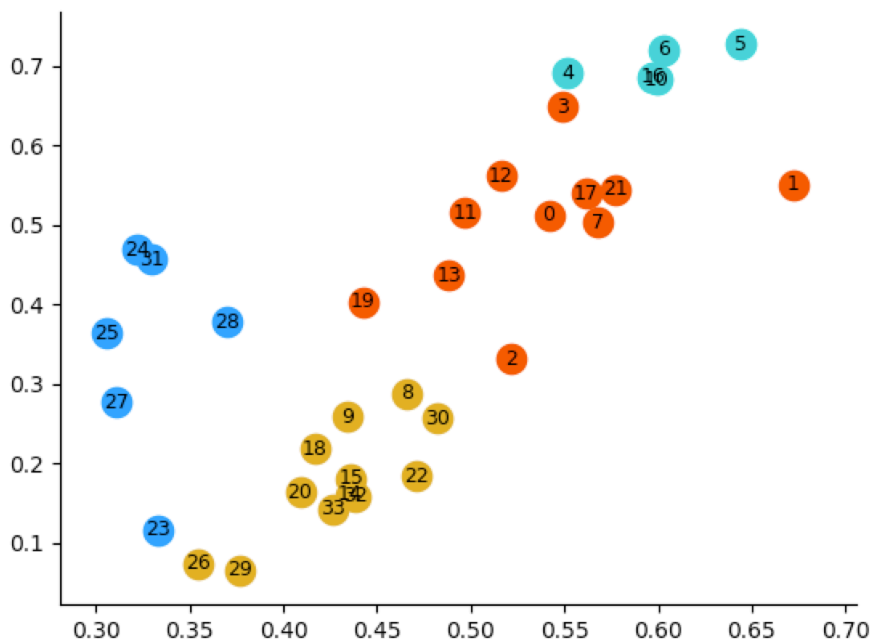

FIG. 10 2-dimensional embedding of karate network

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.