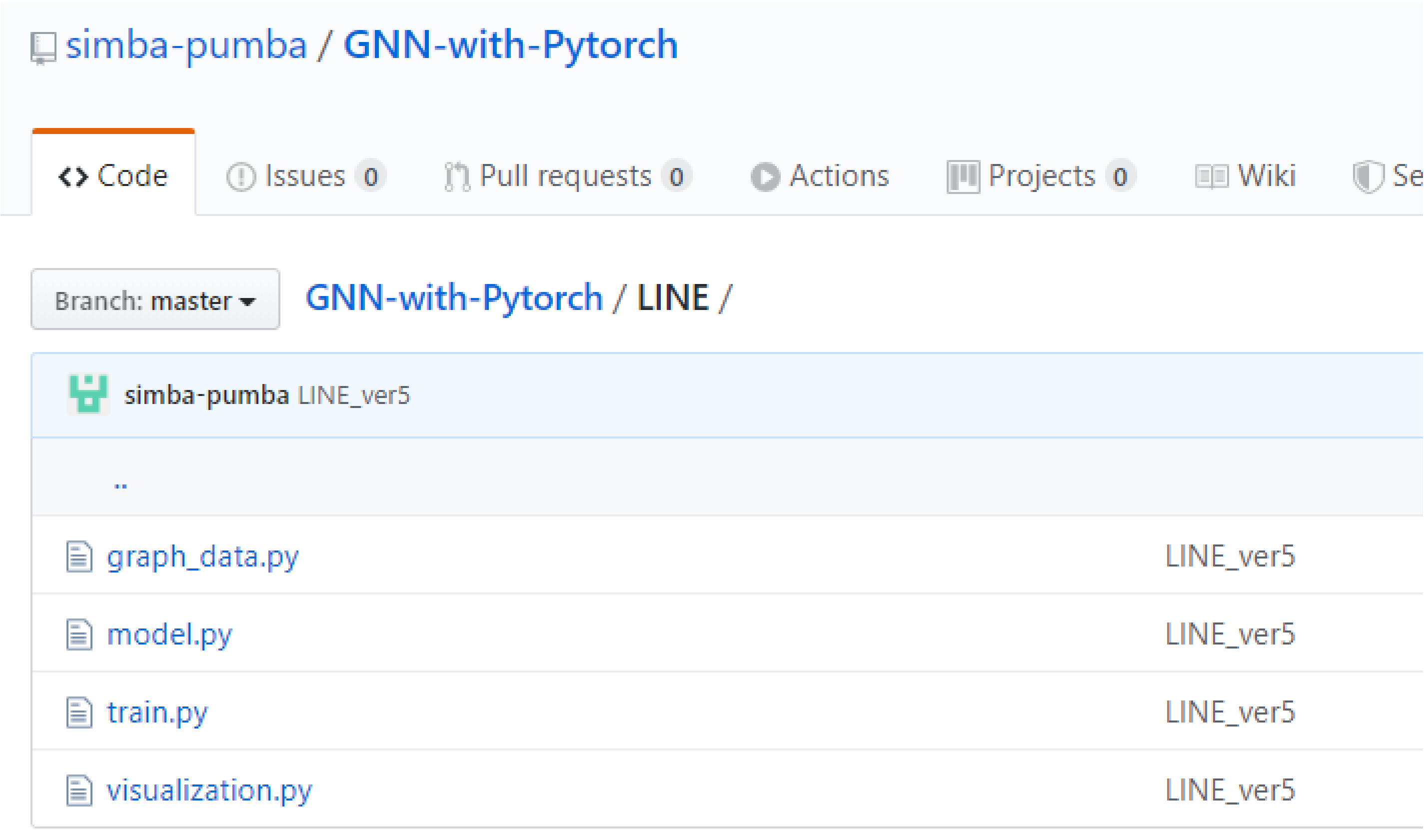
LINE: Large-scale Information Network Embedding in Pytorch

GNN Study

Presenter: Ye-ji Lee

Type	Spatial
	Transductive setting
Loss	Unsupervised learning (negative sampling)
Function	Embedding layer(?)
Input	Adjacency matrix
Task	Embedding

Hierarchical structure



Reference code: https://github.com/DMPierre/LINE

Code review

GraphData.py

```python
import networkx as nx
from torch.utils.data.sampler import WeightedRandomSample

class GraphData:
    def __init__(self, G):

        '''
        G: networkx 객체
        G.nodes, G.edges, G.neighbors(i)
        '''
        self.G=G


    def NegativeSampler(self, v, k=2 ,with_replacement=False):
        '''
        k: sample의 개수
        i: vertex
        d_v^(3/4), where d_v = the out-degree of vertex v
        '''

        # v의 negative nodes set 구하기
        neg_neighbor = set(self.G.nodes)-set(self.G.neighbors(v))-set([v])

        neg_neighbor = list(neg_neighbor)

        # out degree^(3/4) 구하기
        out_degree = list( map ( lambda a: self.G.degree(a) **(3/4 ), neg_neighbor ) )

        # out_degree 분포를 고려하여 k개의 랜덤 샘플을 뽑음
        idx = list(WeightedRandomSampler(out_degree, k, replacement=with_replacement))


        return list(map(lambda i: neg_neighbor[i], idx))
```

GraphData.py

```python
def NegativeSampler(self, v, k=2 ,with_replacement=False):
    '''
    k: sample의 개수
    i: vertex
    d_v^(3/4), where d_v = the out-degree of vertex v
    '''

    # v의 negative nodes set 구하기
    neg_neighbor = set(self.G.nodes)-set(self.G.neighbors(v))-set([v])

    neg_neighbor = list(neg_neighbor)

    # out degree^(3/4) 구하기
    out_degree = list( map ( lambda a: self.G.degree(a) **(3/4), neg_neighbor ) )

    # out_degree 분포를 고려하여 k개의 랜덤 샘플을 뽑음
    idx = list(WeightedRandomSampler(out_degree, k, replacement=with_replacement))


    return list(map(lambda i: neg_neighbor[i], idx))
```

Negative neighbor set

```
neg_neighbor = set(self.G.nodes)-set(self.G.neighbors(v))-set([v])
```

```
neg_neighbor = 전체 노드 - 나의 이웃 - 나
```

Out degree distribution

```
out_degree = list( map ( lambda a: self.G.degree(a) **(3/4), neg_neighbor ) )
```

Random sampling

```
list(WeightedRandomSampler(out_degree, k, replacement=with_replacement))
```

```python
import torch
from torch import nn, optim
import torch.nn.functional as F
from tqdm import tqdm
import networkx as nx
import numpy as np

import graph_data
```

model.py

```python
class LINE(nn.Module):
    def __init__(self, G, emd_dim, neg_k = 2, second_order = False):
        '''

        emd_dim: embedding dimension
        neg_k: # negative samples


        '''
        super(LINE, self).__init__()
        self.G = GraphData(G)
        self.V = len(G.nodes) # number of nodes
        self.neg_k = neg_k

        self.emd_dim = emd_dim

        self.emd_layer = nn.Embedding(num_embeddings = self.V,
                                      embedding_dim = emd_dim)

        self.second_order = second_order
        if self.second_order:  # second_order를 위한 context_layer 생성
            self.context_layer = nn.Embedding(num_embeddings = self.V,
                                      embedding_dim = emd_dim)


    def forward(self, i, j, device):
        v_i = self.emd_layer(i).to(device) # target node의 embeddding vector
        if self.second_order:  # first neighbor의 embeddding vector
            v_j = self.context_layer(j).to(device)
# second_order일 경우, context layer에서 가져옴.
        else: # first neighbor의 embeddding vector
            v_j = self.emd_layer(j).to(device)

        # negative sampling의 postive 파트
        pos_loss = F.logsigmoid(torch.matmul(v_i, v_j)).to(device)

        # negative sampling의 negative 파트
        neg_set = self.G.NegativeSampler(v = i.tolist() , k = self.neg_k)
# v: target node, k: # of negative samples
        neg_set = torch.tensor(neg_set).to(device)
        neg_loss = 0

        for v_n in neg_set:
            if self.second_order:
                prob = torch.matmul(v_i,self.context_layer(v_n)).to(device)
                neg_loss += F.logsigmoid(prob).to(device)

            else: # first_order
                prob = torch.matmul(v_i,self.emd_layer(v_n)).to(device)
                neg_loss += F.logsigmoid(prob).to(device)

        negative_sampling = pos_loss + neg_loss

        return negative_sampling
```

model.py

```python
class LINE(nn.Module):
    def __init__(self, G, emd_dim, neg_k = 2, second_order = False):
        '''

        emd_dim: embedding dimension
        neg_k: # negative samples


        '''
        super(LINE, self).__init__()
        self.G = GraphData(G)
        self.V = len(G.nodes) # number of nodes
        self.neg_k = neg_k

        self.emd_dim = emd_dim

        self.emd_layer = nn.Embedding(num_embeddings = self.V,
                                      embedding_dim = emd_dim)

        self.second_order = second_order
        if self.second_order:  # second_order를 위한 context_layer 생성
            self.context_layer = nn.Embedding(num_embeddings = self.V,
                                      embedding_dim = emd_dim)
```

Embedding layer

```python
self.emd_layer = nn.Embedding(num_embeddings = self.V,
                              embedding_dim = emd_dim)
```

```python
if self.second_order:  # second_order를 위한 context_layer 생성
            self.context_layer = nn.Embedding(num_embeddings = self.V,
                                      embedding_dim = emd_dim)
```

model.py

```python
def forward(self, i, j, device):
    v_i = self.emd_layer(i).to(device) # target node의 embeddding vector
    if self.second_order:  # first neighbor의 embeddding vector
        v_j = self.context_layer(j).to(device) # second_order일 경우, context layer에서 가져옴.
    else: # first neighbor의 embeddding vector
        v_j = self.emd_layer(j).to(device)

    # negative sampling의 postive 파트
    pos_loss = F.logsigmoid(torch.matmul(v_i, v_j)).to(device)

    # negative sampling의 negative 파트
    neg_set = self.G.NegativeSampler(v = i.tolist() , k = self.neg_k)
# v: target node, k: # of negative samples
    neg_set = torch.tensor(neg_set).to(device)
    neg_loss = 0

    for v_n in neg_set:
        if self.second_order:
            prob = torch.matmul(v_i,self.context_layer(v_n)).to(device)
            neg_loss += F.logsigmoid(-prob).to(device)

        else: # first_order
            prob = torch.matmul(v_i,self.emd_layer(v_n)).to(device)
            neg_loss += F.logsigmoid(-prob).to(device)

    negative_sampling = pos_loss + neg_loss

    return negative_sampling
```

Embedding vector

```python
v_i = self.emd_layer(i).to(device) # target node의 embeddding vector
if self.second_order:
    v_j = self.context_layer(j).to(device)

# second_order일 경우, context layer에서 가져옴.
else: # first neighbor의 embeddding vector
    v_j = self.emd_layer(j).to(device)
```

Positive part

$$\underline{\log(\sigma(\mathbf{u}_j'^T \mathbf{u}_i))} - \sum_{i=1}^{K} \cdot \mathbb{E}_{v_n \sim P_n(v)} \log(\sigma(-\mathbf{u}_n'^T \mathbf{u}_i))$$

```python
pos_loss = F.logsigmoid(torch.matmul(v_i, v_j)).to(device)
```

Code review

model.py

```python
def forward(self, i, j, device):
    v_i = self.emd_layer(i).to(device) # target node의 embeddding vector
    if self.second_order:  # first neighbor의 embeddding vector
        v_j = self.context_layer(j).to(device) # second_order일 경우, context layer에서 가져옴.
    else: # first neighbor의 embeddding vector
        v_j = self.emd_layer(j).to(device)

    # negative sampling의 postive 파트
    pos_loss = F.logsigmoid(torch.matmul(v_i, v_j)).to(device)

    # negative sampling의 negative 파트
    neg_set = self.G.NegativeSampler(v = i.tolist() , k = self.neg_k)
# v: target node, k: # of negative samples
    neg_set = torch.tensor(neg_set).to(device)
    neg_loss = 0

    for v_n in neg_set:
        if self.second_order:
            prob = torch.matmul(v_i,self.context_layer(v_n)).to(device)
            neg_loss += F.logsigmoid(-prob).to(device)

        else: # first_order
            prob = torch.matmul(v_i,self.emd_layer(v_n)).to(device)
            neg_loss += F.logsigmoid(-prob).to(device)

    negative_sampling = pos_loss + neg_loss

    return negative_sampling
```

Negative part

$$\log(\sigma(\mathbf{u}_j'^T \mathbf{u}_i)) - \sum_{i=1}^{K} \cdot \mathbb{E}_{v_n \sim P_n(v)} \underline{\log(\sigma(-\mathbf{u}_n'^T \mathbf{u}_i))}$$

```python
neg_set = self.G.NegativeSampler(v = i.tolist() , k = self.neg_k)
```

```python
for v_n in neg_set:
    if self.second_order:
        prob = torch.matmul(v_i,self.context_layer(v_n)).to(device)
        neg_loss += F.logsigmoid(-prob).to(device)

    else: # first_order
        prob = torch.matmul(v_i,self.emd_layer(v_n)).to(device)
        neg_loss += F.logsigmoid(-prob).to(device)
```

model.py

```python
def forward(self, i, j, device):
    v_i = self.emd_layer(i).to(device) # target node의 embeddding vector
    if self.second_order:  # first neighbor의 embeddding vector
        v_j = self.context_layer(j).to(device) # second_order일 경우, context layer에서 가져옴.
    else: # first neighbor의 embeddding vector
        v_j = self.emd_layer(j).to(device)

    # negative sampling의 postive 파트
    pos_loss = F.logsigmoid(torch.matmul(v_i, v_j)).to(device)

    # negative sampling의 negative 파트
    neg_set = self.G.NegativeSampler(v = i.tolist() , k = self.neg_k)
# v: target node, k: # of negative samples
    neg_set = torch.tensor(neg_set).to(device)
    neg_loss = 0

    for v_n in neg_set:
        if self.second_order:
            prob = torch.matmul(v_i,self.context_layer(v_n)).to(device)
            neg_loss += F.logsigmoid(-prob).to(device)

        else: # first_order
            prob = torch.matmul(v_i,self.emd_layer(v_n)).to(device)
            neg_loss += F.logsigmoid(-prob).to(device)

    negative_sampling = pos_loss + neg_loss

    return negative_sampling
```

Objective function(loss)

$$\log(\sigma(\mathbf{u}_j'^T \mathbf{u}_i)) - \sum_{i=1}^{K} \cdot \mathbb{E}_{v_n \sim P_n(v)} \log(\sigma(-\mathbf{u}_n'^T \mathbf{u}_i))$$

```
negative_sampling = pos_loss + neg_loss
```

visualization.py

```python
import matplotlib.pyplot as plt
import seaborn as sns
import networkx as nx
import numpy as np
from sklearn.manifold import TSNE


class Visualization:
    def __init__(self, labels = None):
        # karate graph label 임시로 고정
        if labels is None:
            self.labels = [0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,1,0,0,1,0,1,0,1,1,1,1,1,1,1,1,1,1,1,1]

    def TSNE(self, matrix,iter_num):
        model = TSNE(n_components=2, random_state=0)
        model.fit_transform(matrix)
        sns.scatterplot(matrix[:,0],matrix[:,1],hue = self.labels)
        plt.savefig('fig'+str(iter_num)+'.png', dpi=300)
        plt.close()
```

Code review

train.py

```python

import torch
from torch import nn, optim
from tqdm import tqdm
import torch.nn.functional as F
from torch.utils.data.sampler import WeightedRandomSampler
import networkx as nx
import numpy as np

import model
import visualization
```

```python
class TrainLINE:
    def __init__(self, G, emd_dim, epochs,learning_rate = 0.01, neg_k = 2, second_order =
 False):
        '''
        G: networkx 객체
        emd_dim: embedding dimension
        neg_k: # negative samples

LINE(G = G,emd_dim = emd_dim, neg_k = 2, second_order = second_order)
g_rate = learning_rate

list(G.edges)
len(G.nodes)
= epochs

SGD(self.line.parameters(), lr=self.learning_rate)
ch.device("cuda" if torch.cuda.is_available() else "cpu")
da()

range(self.epochs):
e.train()

로 쓰일 edges를 epoch마다 random하게 섞어줌
m.shuffle(self.batch)

poch {}".format(epoch))

tqdm(self.batch):
.line.zero_grad()
= self.line(i=torch.tensor(b[0],device=device), j=torch.tensor(b[1],
ce=device)
.backward()
step()

            # epoch마다 embedding vectors의 plot을 뽑음.
            self.line.eval()
            Visualization().TSNE(self.line.emd_layer(torch.tensor(range(self.nodes),device=
device)).cpu().detach().numpy(),epoch)


        print("\nDone\n")


    if __name__ == "__main__":
    model = TrainLINE(nx.karate_club_graph(), epochs = 100, learning_rate = 0.0001, emd_dim =
 5, neg_k=2, second_order = True)
    model.train()
```

train.py

```python
def __init__(self, G, emd_dim, epochs,learning_rate = 0.01, neg_k = 2, second_order = False):
        '''
        G: networkx 객체
        emd_dim: embedding dimension
        neg_k: # negative samples
        batch: edges
        '''
        self.line = LINE(G = G,emd_dim = emd_dim, neg_k = 2, second_order = second_order)
        self.learning_rate = learning_rate

        self.batch = list(G.edges)
        self.nodes = len(G.nodes)
        self.epochs = epochs
```

Model object

```python
self.line = LINE(G = G,emd_dim = emd_dim, neg_k = 2, second_order = second_order)
```

train.py

```python
def train(self):
    opt = optim.SGD(self.line.parameters(), lr=self.learning_rate)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    self.line.cuda()

    for epoch in range(self.epochs):
        self.line.train()

        # batch로 쓰일 edges를 epoch마다 random하게 섞어줌
        np.random.shuffle(self.batch)

        print("Epoch {}".format(epoch))

        for b in tqdm(self.batch):
            self.line.zero_grad()
            loss = self.line(i=torch.tensor(b[0],device=device), j=torch.tensor(b[1],device=
device), device=device)
            loss.backward()
            opt.step()

        # epoch마다 embedding vectors의 plot을 뽑음.
        self.line.eval()
        Visualization().TSNE(self.line.emd_layer(torch.tensor(range(self.nodes),device=
device)).cpu().detach().numpy(),epoch)


    print("\nDone\n")
```

Two parts(train, eval; test)

```
for epoch in range(self.epochs):
    self.line.train()
    for batch in tqdm(self.batch):
        파라미터 업데이트

    self.line.eval()
    plot 뽑아내기
```

Code review

train.py

```python
def train(self):
    opt = optim.SGD(self.line.parameters(), lr=self.learning_rate)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    self.line.cuda()

    for epoch in range(self.epochs):
        self.line.train()

        # batch로 쓰일 edges를 epoch마다 random하게 섞어줌
        np.random.shuffle(self.batch)

        print("Epoch {}".format(epoch))

        for b in tqdm(self.batch):
            self.line.zero_grad()
            loss = self.line(i=torch.tensor(b[0],device=device), j=torch.tensor(b[1],device=device), device=device)
            loss.backward()
            opt.step()

        # epoch마다 embedding vectors의 plot을 뽑음.
        self.line.eval()
        Visualization().TSNE(self.line.emd_layer(torch.tensor(range(self.nodes),device=device)).cpu().detach().numpy(),epoch)


    print("\nDone\n")
```

Batch(train)

```python
# batch로 쓰일 edges를 epoch마다 random하게 섞어줌
np.random.shuffle(self.batch)
```

```python
for b in tqdm(self.batch):
    self.line.zero_grad()
    loss = self.line(i=torch.tensor(b[0],device=device), j=torch.tensor(b[1],device=device), device=device)
    loss.backward()
    opt.step()
```

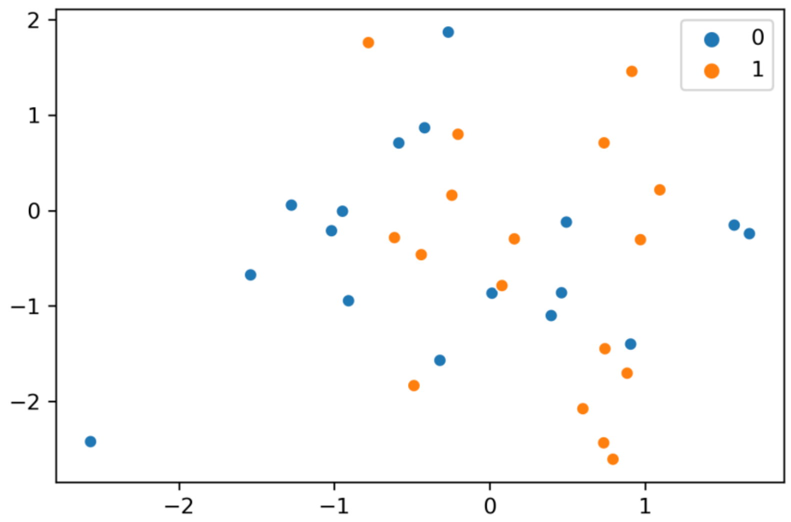
Call function;
LINE().forward()

train.py

```python
def train(self):
    opt = optim.SGD(self.line.parameters(), lr=self.learning_rate)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    self.line.cuda()

    for epoch in range(self.epochs):
        self.line.train()

        # batch로 쓰일 edges를 epoch마다 random하게 섞어줌
        np.random.shuffle(self.batch)

        print("Epoch {}".format(epoch))

        for b in tqdm(self.batch):
            self.line.zero_grad()
            loss = self.line(i=torch.tensor(b[0],device=device), j=torch.tensor(b[1],device=
device), device=device)
            loss.backward()
            opt.step()

        # epoch마다 embedding vectors의 plot을 뽑음.
        self.line.eval()
        Visualization().TSNE(self.line.emd_layer(torch.tensor(range(self.nodes),device=
device)).cpu().detach().numpy(),epoch)


    print("\nDone\n")
```

Eval(test)

```python
self.line.eval()
Visualization().TSNE(self.line.emd_layer(torch.tensor(range(self.nodes),
device=device)).cpu().detach().numpy(),epoch)
```



Code review

train.py

```python
def train(self):
    opt = optim.SGD(self.line.parameters(), lr=self.learning_rate)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    self.line.cuda()

    for epoch in range(self.epochs):
        self.line.train()

        # batch로 쓰일 edges를 epoch마다 random하게 섞어줌
        np.random.shuffle(self.batch)

        print("Epoch {}".format(epoch))

        for b in tqdm(self.batch):
            self.line.zero_grad()
            loss = self.line(i=torch.tensor(b[0],device=device), j=torch.tensor(b[1],device=device), device=device)
            loss.backward()
            opt.step()

        # epoch마다 embedding vectors의 plot을 뽑음.
        self.line.eval()
        Visualization().TSNE(self.line.emd_layer(torch.tensor(range(self.nodes),device=device)).cpu().detach().numpy(),epoch)


    print("\nDone\n")
```

Main function

```python
if __name__ == "__main__":
    model = TrainLINE(nx.karate_club_graph(), epochs = 100, learning_rate = 0.0001, emd_dim = 5, neg_k=2, second_order = True)
    model.train()
```