
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

simba-pumba / GNN-with-Pytorch

<> Code ⓘ Issues 0 🔑 Pull requests 0 ▶ Actions 📁 Projects 0 📖 Wiki 🛡️ Security

Branch: master ▾ GNN-with-Pytorch / LINE /

simba-pumba LINE_ver5

..

graph_data.py	LINE_ver5
model.py	LINE_ver5
train.py	LINE_ver5
visualization.py	LINE_ver5

GraphData.py

```
1 import networkx as nx
2 from torch.utils.data.sampler import WeightedRandomSample
3
4 class GraphData:
5     def __init__(self, G):
6
7         '''
8         G: networkx 객체
9         G.nodes, G.edges, G.neighbors(i)
10        '''
11        self.G=G
12
13
14    def NegativeSampler(self, v, k=2 ,with_replacement=False):
15        '''
16        k: sample의 개수
17        i: vertex
18        d_v^(3/4), where d_v = the out-degree of vertex v
19        '''
20
21        # v의 negative nodes set 구하기
22        neg_neighbor = set(self.G.nodes)-set(self.G.neighbors(v))-set([v])
23
24        neg_neighbor = list(neg_neighbor)
25
26        # out degree^(3/4) 구하기
27        out_degree = list( map ( lambda a: self.G.degree(a) **(3/4
28                                ), neg_neighbor ) )
29
30        # out_degree 분포를 고려하여 k개의 랜덤 샘플을 뽑음
31        idx = list(WeightedRandomSampler(out_degree, k, replacement=
32                                with_replacement))
33
34        return list(map(lambda i: neg_neighbor[i], idx))
```

GraphData.py



```
1 def NegativeSampler(self, v, k=2 ,with_replacement=False):
2     '''
3     k: sample의 개수
4     i: vertex
5     d_v^(3/4), where d_v = the out-degree of vertex v
6     '''
7
8     # v의 negative nodes set 구하기
9     neg_neighbor = set(self.G.nodes)-set(self.G.neighbors(v))-set([v])
10
11     neg_neighbor = list(neg_neighbor)
12
13     # out degree^(3/4) 구하기
14     out_degree = list( map ( lambda a: self.G.degree(a) **(3/4), neg_neighbor ) )
15
16     # out_degree 분포를 고려하여 k개의 랜덤 샘플을 뽑음
17     idx = list(WeightedRandomSampler(out_degree, k, replacement=with_replacement))
18
19
20     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))
```

model.py



```

1 import torch
2 from torch import nn, optim
3 import torch.nn.functional as F
4 from tqdm import tqdm
5 import networkx as nx
6 import numpy as np
7
8 import graph_data

```



```

1 class LINE(nn.Module):
2     def __init__(self, G, emd_dim, neg_k = 2, second_order = False):
3         ...
4
5         emd_dim: embedding dimension
6         neg_k: # negative samples
7
8         ...
9
10        super(LINE, self).__init__()
11        self.G = GraphData(G)
12        self.V = len(G.nodes) # number of nodes
13        self.neg_k = neg_k
14
15        self.emd_dim = emd_dim
16
17        self.emd_layer = nn.Embedding(num_embeddings = self.V,
18                                     embedding_dim = emd_dim)
19
20        self.second_order = second_order
21        if self.second_order: # second_order를 위한 context_layer 생성
22            self.context_layer = nn.Embedding(num_embeddings = self.V,
23                                             embedding_dim = emd_dim)
24
25
26        def forward(self, i, j, device):
27            v_i = self.emd_layer(i).to(device) # target node의 embedding vector
28            if self.second_order: # first neighbor의 embedding vector
29                v_j = self.context_layer(j).to(device)
30            # second_order일 경우, context_layer에서 가져옴.
31            else: # first neighbor의 embedding vector
32                v_j = self.emd_layer(j).to(device)
33
34            # negative sampling의 positive loss
35            pos_loss = F.logsigmoid(torch.matmul(v_i, v_j)).to(device)
36
37            # negative sampling의 negative loss
38            neg_set = self.G.NegativeSampler(v = i.tolist(), k = self.neg_k)
39            # v: target node, k: # of negative samples
40            neg_set = torch.tensor(neg_set).to(device)
41            neg_loss = 0
42
43            for v_n in neg_set:
44                if self.second_order:
45                    prob = torch.matmul(v_i, self.context_layer(v_n)).to(device)
46                    neg_loss += F.logsigmoid(prob).to(device)
47
48                else: # first_order
49                    prob = torch.matmul(v_i, self.emd_layer(v_n)).to(device)
50                    neg_loss += F.logsigmoid(prob).to(device)
51
52            negative_sampling = pos_loss + neg_loss
53
54            return negative_sampling

```

model.py

[illegible]

Embedding layer

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

[illegible]

model.py

```

1  def forward(self, i, j, device):
2      v_i = self.emd_layer(i).to(device) # target node의 embeddding vector
3      if self.second_order: # first neighbor의 embeddding vector
4          v_j = self.context_layer(j).to(device) # second_order일 경우, context layer에서 가져옴.
5      else: # first neighbor의 embeddding vector
6          v_j = self.emd_layer(j).to(device)
7
8      # negative sampling의 positive 파트
9      pos_loss = F.logsigmoid(torch.matmul(v_i, v_j)).to(device)
10
11     # negative sampling의 negative 파트
12     neg_set = self.G.NegativeSampler(v = i.tolist(), k = self.neg_k)
13     # v: target node, k: # of negative samples
14     neg_set = torch.tensor(neg_set).to(device)
15     neg_loss = 0
16
17     for v_n in neg_set:
18         if self.second_order:
19             prob = torch.matmul(v_i, self.context_layer(v_n)).to(device)
20             neg_loss += F.logsigmoid(-prob).to(device)
21
22         else: # first_order
23             prob = torch.matmul(v_i, self.emd_layer(v_n)).to(device)
24             neg_loss += F.logsigmoid(-prob).to(device)
25
26     negative_sampling = pos_loss + neg_loss
27     return negative_sampling

```

Embedding vector

```

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

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

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


model.py

```

1 def forward(self, i, j, device):
2     v_i = self.emd_layer(i).to(device) # target node의 embeddding vector
3     if self.second_order: # first neighbor의 embeddding vector
4         v_j = self.context_layer(j).to(device) # second_order일 경우, context layer에서 가져옴.
5     else: # first neighbor의 embeddding vector
6         v_j = self.emd_layer(j).to(device)
7
8     # negative sampling의 positive 파트
9     pos_loss = F.logsigmoid(torch.matmul(v_i, v_j)).to(device)
10
11    # negative sampling의 negative 파트
12    neg_set = self.G.NegativeSampler(v = i.tolist() , k = self.neg_k)
13    # v: target node, k: # of negative samples
14    neg_set = torch.tensor(neg_set).to(device)
15    neg_loss = 0
16
17    for v_n in neg_set:
18        if self.second_order:
19            prob = torch.matmul(v_i, self.context_layer(v_n)).to(device)
20            neg_loss += F.logsigmoid(-prob).to(device)
21
22        else: # first_order
23            prob = torch.matmul(v_i, self.emd_layer(v_n)).to(device)
24            neg_loss += F.logsigmoid(-prob).to(device)
25
26    negative_sampling = pos_loss + neg_loss
27    return negative_sampling

```

Negative part

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

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

```

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

```

1  def forward(self, i, j, device):
2      v_i = self.emd_layer(i).to(device) # target node의 embeddding vector
3      if self.second_order: # first neighbor의 embeddding vector
4          v_j = self.context_layer(j).to(device) # second_order일 경우, context layer에서 가져옴.
5      else: # first neighbor의 embeddding vector
6          v_j = self.emd_layer(j).to(device)
7
8      # negative sampling의 positive 파트
9      pos_loss = F.logsigmoid(torch.matmul(v_i, v_j)).to(device)
10
11     # negative sampling의 negative 파트
12     neg_set = self.G.NegativeSampler(v = i.tolist(), k = self.neg_k)
13     # v: target node, k: # of negative samples
14     neg_set = torch.tensor(neg_set).to(device)
15     neg_loss = 0
16
17     for v_n in neg_set:
18         if self.second_order:
19             prob = torch.matmul(v_i, self.context_layer(v_n)).to(device)
20             neg_loss += F.logsigmoid(-prob).to(device)
21
22         else: # first_order
23             prob = torch.matmul(v_i, self.emd_layer(v_n)).to(device)
24             neg_loss += F.logsigmoid(-prob).to(device)
25
26     negative_sampling = pos_loss + neg_loss
27
28     return negative_sampling

```

Objective function(loss)

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

```
negative_sampling = pos_loss + neg_loss
```

visualization.py

```
1 import matplotlib.pyplot as plt
2 import seaborn as sns
3 import networkx as nx
4 import numpy as np
5 from sklearn.manifold import TSNE
6
7
8 class Visualization:
9     def __init__(self, labels = None):
10         # karate graph Label 임시로 고정
11         if labels is None:
12             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]
13
14     def TSNE(self, matrix, iter_num):
15         model = TSNE(n_components=2, random_state=0)
16         model.fit_transform(matrix)
17         sns.scatterplot(matrix[:,0],matrix[:,1],hue = self.labels)
18         plt.savefig('fig'+str(iter_num)+'.png', dpi=300)
19         plt.close()
```

train.py

```

1
2 import torch
3 from torch import nn, optim
4 from tqdm import tqdm
5 import torch.nn.functional as F
6 from torch.utils.data.sampler import WeightedRandomSampler
7 import networkx as nx
8 import numpy as np
9
10 import model
11 import visualization

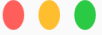
```

```

1 class TrainLINE:
2     def __init__(self, G, emd_dim, epochs, learning_rate = 0.01, neg_k = 2, second_order =
3         False):
4         """
5         G: networkx 객체
6         emd_dim: embedding dimension
7         neg_k: # negative samples
8         """
9
10        LINE(G = G, emd_dim = emd_dim, neg_k = 2, second_order = second_order)
11        g_rate = learning_rate
12
13        list(G.edges)
14        len(G.nodes)
15        = epochs
16
17        SGD(self.line.parameters(), lr=self.learning_rate)
18        ch.device("cuda" if torch.cuda.is_available() else "cpu")
19        da()
20
21        range(self.epochs):
22            e.train()
23
24            큰 쓰임 edges를 epoch마다 random하게 섞어줌
25            m.shuffle(self.batch)
26
27            epoch {}".format(epoch))
28
29            tqdm(self.batch):
30                .line.zero_grad()
31                = self.line(i=torch.tensor(b[0], device=device), j=torch.tensor(b[1],
32                    ce=device)
33                .backward()
34                step()
35
36            # epoch마다 embedding vectors의 plot을 뽑음.
37            self.line.eval()
38            Visualization().TSNE(self.line.emd_layer(torch.tensor(range(self.nodes), device=
39                device)).cpu().detach().numpy(), epoch)
40
41            print("\nDone\n")
42
43            if __name__ == "__main__":
44                model = TrainLINE(nx.karate_club_graph(), epochs = 100, learning_rate = 0.0001, emd_dim =
45                    5, neg_k=2, second_order = True)
46                model.train()

```

train.py



```
1 def __init__(self, G, emd_dim, epochs, learning_rate = 0.01, neg_k = 2, second_order = False):
2     '''
3     G: networkx 객체
4     emd_dim: embedding dimension
5     neg_k: # negative samples
6     batch: edges
7     '''
8     self.line = LINE(G = G, emd_dim = emd_dim, neg_k = 2, second_order = second_order)
9     self.learning_rate = learning_rate
10
11     self.batch = list(G.edges)
12     self.nodes = len(G.nodes)
13     self.epochs = epochs
```

Model object

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

train.py

```
1 def train(self):
2     opt = optim.SGD(self.line.parameters(), lr=self.learning_rate)
3     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
4     self.line.cuda()
5
6     for epoch in range(self.epochs):
7         self.line.train()
8
9         # batch로 쓰일 edges를 epoch마다 random하게 섞어줌
10        np.random.shuffle(self.batch)
11
12        print("Epoch {}".format(epoch))
13
14        for b in tqdm(self.batch):
15            self.line.zero_grad()
16            loss = self.line(i=torch.tensor(b[0],device=device), j=torch.tensor(b[1],device=
17            device), device=device)
18            loss.backward()
19            opt.step()
20
21            # epoch마다 embedding vectors의 plot을 뽑음.
22            self.line.eval()
23            Visualization().TSNE(self.line.emd_layer(torch.tensor(range(self.nodes),device=
24            device)).cpu().detach().numpy(),epoch)
25
26        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
```

train.py

```

1  def train(self):
2      opt = optim.SGD(self.line.parameters(), lr=self.learning_rate)
3      device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
4      self.line.cuda()
5
6      for epoch in range(self.epochs):
7          self.line.train()
8
9          # batch로 쓰일 edges를 epoch마다 random하게 섞어줌
10         np.random.shuffle(self.batch)
11
12         print("Epoch {}".format(epoch))
13
14         for b in tqdm(self.batch):
15             self.line.zero_grad()
16             loss = self.line(i=torch.tensor(b[0],device=device), j=torch.tensor(b[1],device=
device), device=device)
17             loss.backward()
18             opt.step()
19
20         # epoch마다 embedding vectors의 plot을 뽑음.
21         self.line.eval()
22         Visualization().TSNE(self.line.emd_layer(torch.tensor(range(self.nodes),device=
device)).cpu().detach().numpy(),epoch)
23
24
25     print("\nDone\n")

```

Batch(train)

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

```

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

```

1  def train(self):
2      opt = optim.SGD(self.line.parameters(), lr=self.learning_rate)
3      device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
4      self.line.cuda()
5
6      for epoch in range(self.epochs):
7          self.line.train()
8
9          # batch로 쓰일 edges를 epoch마다 random하게 섞어줌
10         np.random.shuffle(self.batch)
11
12         print("Epoch {}".format(epoch))
13
14         for b in tqdm(self.batch):
15             self.line.zero_grad()
16             loss = self.line(i=torch.tensor(b[0],device=device), j=torch.tensor(b[1],device=
device), device=device)
17             loss.backward()
18             opt.step()
19
20             # epoch마다 embedding vectors의 plot을 뽑음.
21             self.line.eval()
22             Visualization().TSNE(self.line.emd_layer(torch.tensor(range(self.nodes),device=
device)).cpu().detach().numpy(),epoch)
23
24         print("\nDone\n")

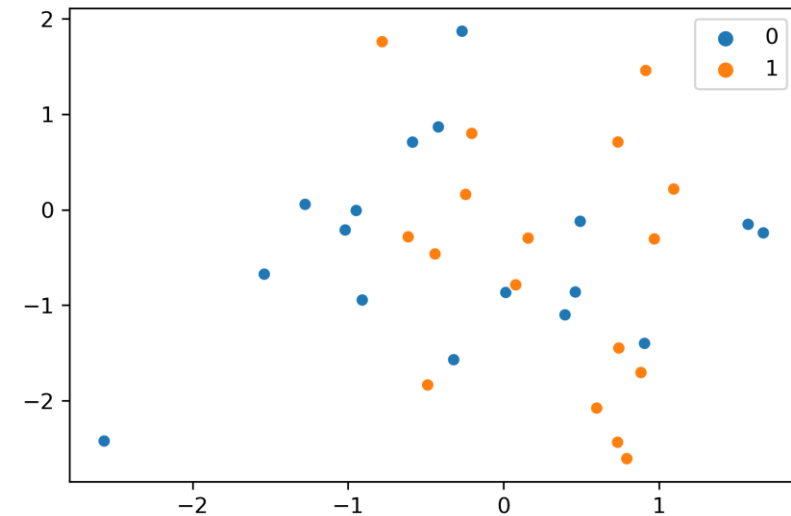
```

Eval(test)

```

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

```



train.py

```
1 def train(self):
2     opt = optim.SGD(self.line.parameters(), lr=self.learning_rate)
3     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
4     self.line.cuda()
5
6     for epoch in range(self.epochs):
7         self.line.train()
8
9         # batch로 쓰일 edges를 epoch마다 random하게 섞어줌
10        np.random.shuffle(self.batch)
11
12        print("Epoch {}".format(epoch))
13
14        for b in tqdm(self.batch):
15            self.line.zero_grad()
16            loss = self.line(i=torch.tensor(b[0],device=device), j=torch.tensor(b[1],device=
17            device), device=device)
18            loss.backward()
19            opt.step()
20
21            # epoch마다 embedding vectors의 plot을 뽑음.
22            self.line.eval()
23            Visualization().TSNE(self.line.emd_layer(torch.tensor(range(self.nodes),device=
24            device)).cpu().detach().numpy(),epoch)
25
26        print("\nDone\n")
```

Main function

```
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()
```