DeepWalk and node2vec: Implementation details

Gabriele Santin[1]

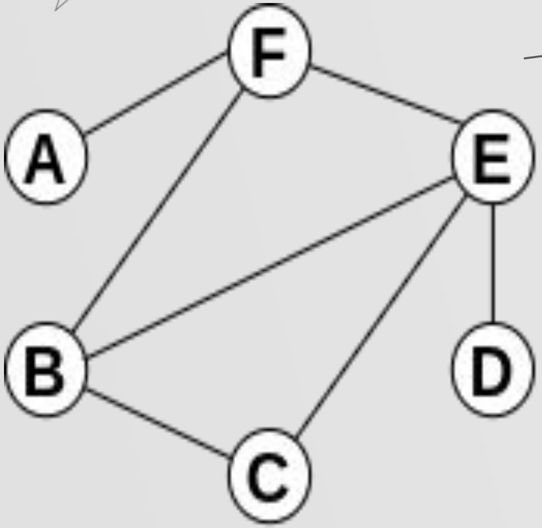
MobS[1] Lab, Fondazione Bruno Kessler,Trento, Italy

TABLE OF CONTENTS

01 Learning graph representations

Goal:

- Find a good representation of a graph G = (V, E)



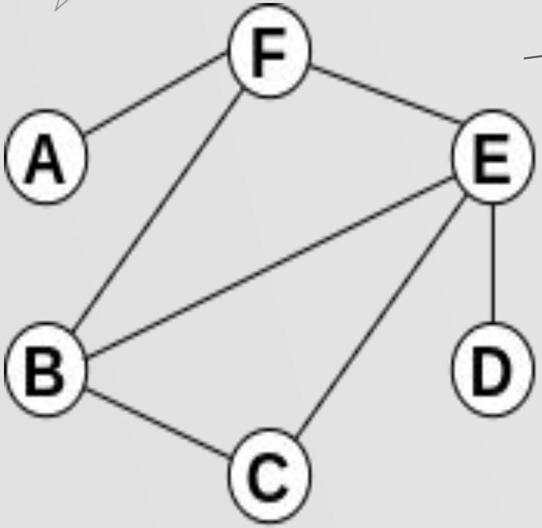
Node embedding:

$$v \mapsto [f_1(v), \ldots, f_d(v)]$$

01 Learning graph representations

Goal:

- Find a good representation of a graph G = (V, E)



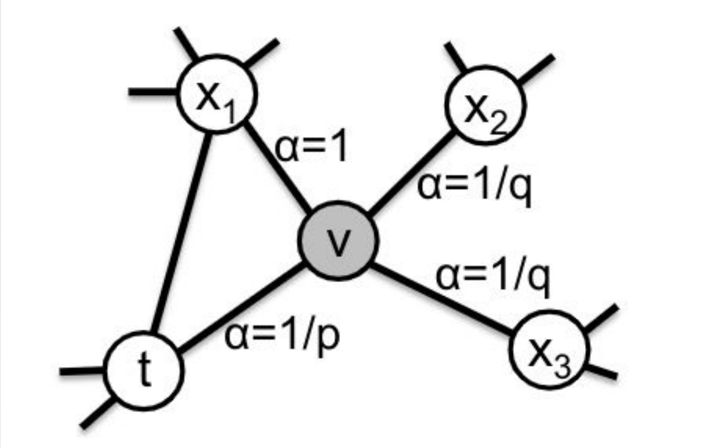
Node embedding:

$$v \mapsto [f_1(v), \ldots, f_d(v)]$$

Strategy:

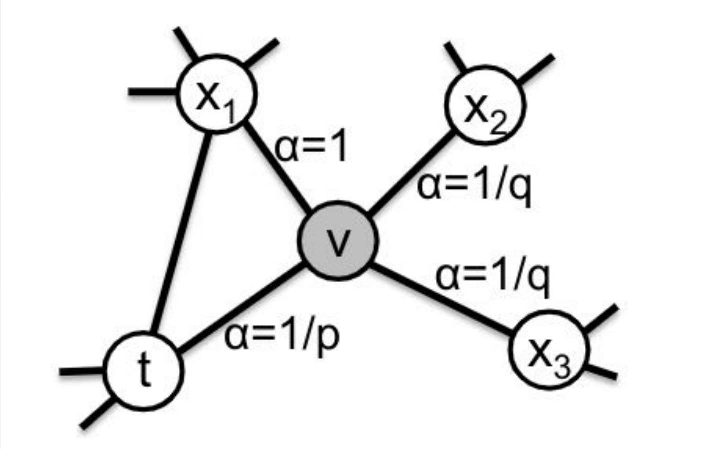
- Optimize embedding to preserve similarities
- Similarities defined as a "neighborhood" notion
- Use (biased) random walks to define neighborhood

02 Biased random walks



$$P(N_{i+1} = v | N_i = u) = \begin{cases} \frac{\pi_{vu}}{Z} \text{ if } (v, u) \in E \\ 0 \text{ otherwise} \end{cases}$$

02 Biased random walks



Unbiased random walk

$$P(N_{i+1} = v | N_i = u) = \begin{cases} \frac{\pi_{vu}}{Z} \text{ if } (v,u) \in E \\ 0 \text{ otherwise} \end{cases}$$

$$\pi_{uv}$$

Transition probability, e.g.

$$\pi_{uv} = w_{vu}$$

02 Biased random walks

Unbiased random walk

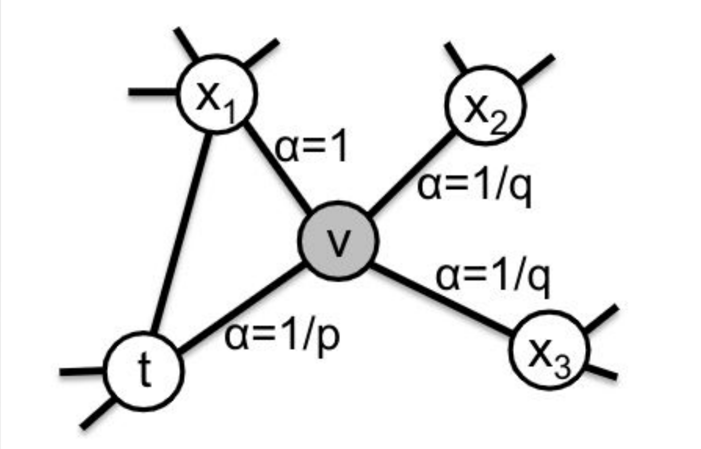
Z Normalization factor

$$P(N_{i+1} = v | N_i = u) = \begin{cases} \frac{\pi_{vu}}{Z} \text{ if } (v,u) \in E \\ 0 \text{ otherwise} \end{cases}$$

$$\pi_{uv}$$

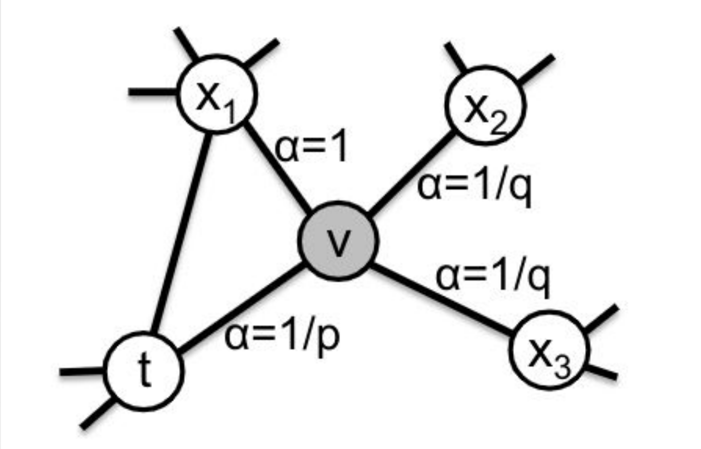
Transition probability, e.g.

$$\pi_{uv} = w_{vu}$$



02 Biased random walks

Z Normalization factor



Unbiased random walk

$$P(N_{i+1} = v | N_i = u) = \begin{cases} \frac{\pi_{vu}}{Z} \text{ if } (v,u) \in E \\ 0 \text{ otherwise} \end{cases}$$

$$\pi_{uv}$$

Transition probability, e.g.

$$\pi_{uv} = w_{vu}$$

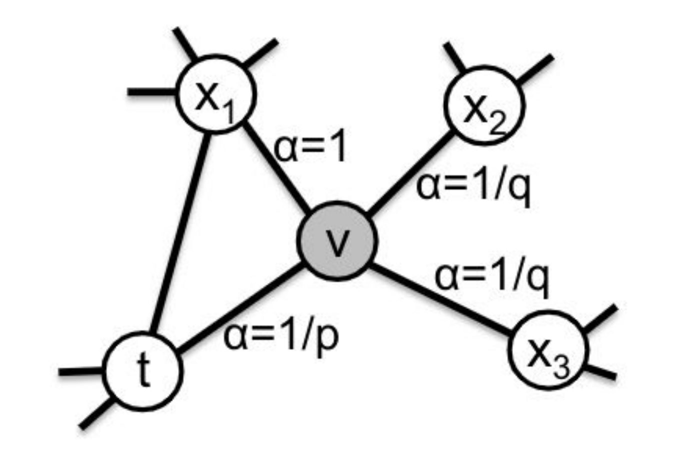
Biased random walk

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} \text{ if } d_{tx} = 0 \\ 1 \text{ if } d_{tx} = 1 \\ \frac{1}{q} \text{ if } d_{tx} = 2 \end{cases}$$

Biased transition probability

$$\pi_{vx} = \alpha_{pq}(t,x) \cdot w_{vx}$$

02 Biased random walks



Unbiased random walk

$$P(N_{i+1} = v | N_i = u) = \begin{cases} \frac{\pi_{vu}}{Z} \text{ if } (v,u) \in E \\ 0 \text{ otherwise} \end{cases}$$

Z Normalization factor

$$\pi_{uv}$$

Transition probability, e.g.

$$\pi_{uv} = w_{vu}$$

Biased random walk

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} \text{ if } d_{tx} = 0 \\ 1 \text{ if } d_{tx} = 1 \\ \frac{1}{q} \text{ if } d_{tx} = 2 \end{cases}$$

Search bias

Biased transition probability

$$\pi_{vx} = \alpha_{pq}(t,x) \cdot w_{vx}$$

02 Biased random walks

Unbiased random walk

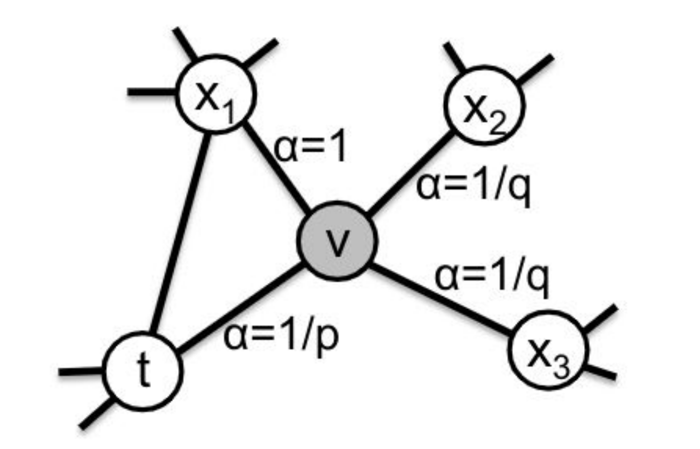
$$P(N_{i+1} = v | N_i = u) = \begin{cases} \frac{\pi_{vu}}{Z} \text{ if } (v,u) \in E \\ 0 \text{ otherwise} \end{cases}$$

Z Normalization factor

π_{uv}

Transition probability, e.g.

$$\pi_{uv} = w_{vu}$$



Biased random walk

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} \text{ if } d_{tx} = 0 \\ 1 \text{ if } d_{tx} = 1 \\ \frac{1}{q} \text{ if } d_{tx} = 2 \end{cases}$$

Search bias

Biased transition probability

$$\pi_{vx} = \alpha_{pq}(t,x) \cdot w_{vx}$$

return parameter p:

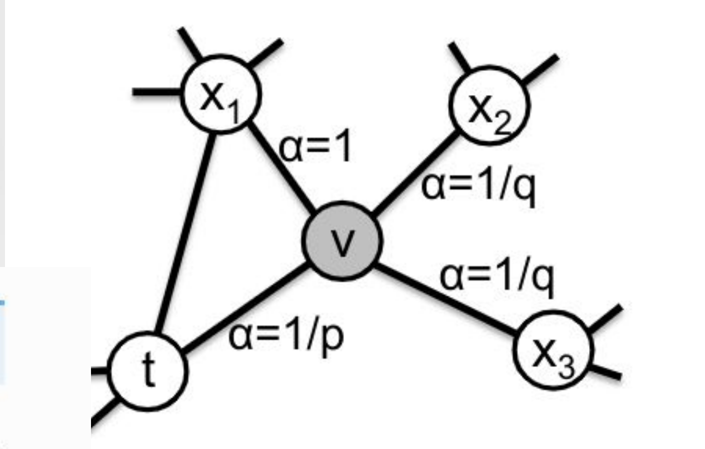
- large -> exploration
- small -> backtrack, local

in-out parameter q:

- large -> stay close to t
- small -> exploration

DeepWalk: q=p=1

02 Biased random walks



CLASS **Node2Vec** (edge_index, embedding_dim, walk_length, context_size, walks_per_node=1, p=1, q=1, num_negative_samples=1, num_nodes=None, sparse=False) [source]

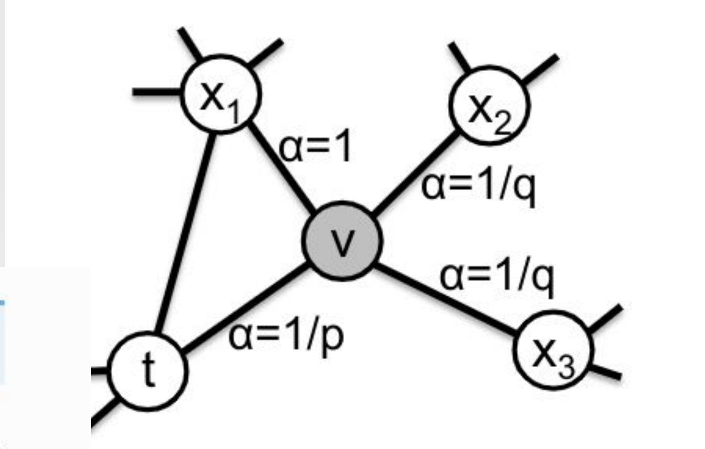
The Node2Vec model from the "node2vec: Scalable Feature Learning for Networks" paper where random walks of length `walk_length` are sampled in a given graph, and node embeddings are learned via negative sampling optimization.

02 Biased random walks

Length

CLASS `Node2Vec` (edge_index, embedding_dim, walk_length, context_size, walks_per_node=1, p=1, q=1, num_negative_samples=1, num_nodes=None, sparse=False) [source]

The Node2Vec model from the "node2vec: Scalable Feature Learning for Networks" paper where random walks of length `walk_length` are sampled in a given graph, and node embeddings are learned via negative sampling optimization.

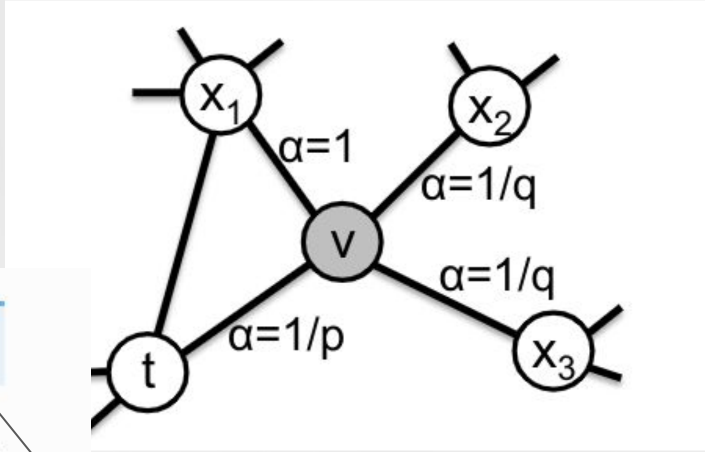


02 Biased random walks

Length

CLASS Node2Vec (edge_index, embedding_dim, walk_length, context_size, walks_per_node=1, p=1, q=1, num_negative_samples=1, num_nodes=None, sparse=False) [source]

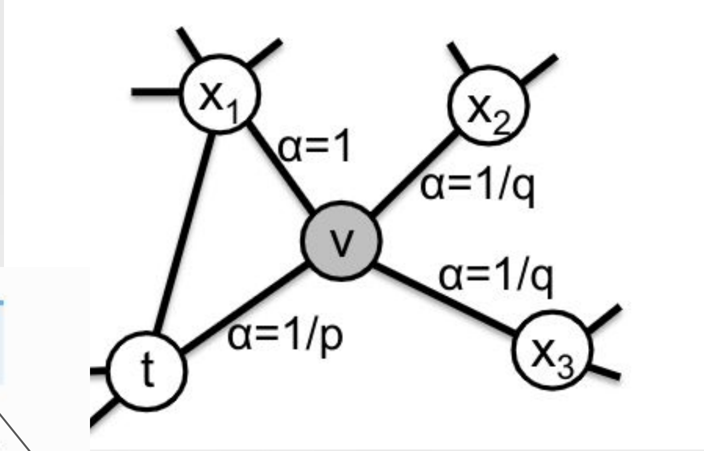
The Node2Vec model from the "node2vec: Scalable Feature Learning for Networks" paper where random walks of length `walk_length` are sampled in a given graph, and node embeddings are learned via negative sampling optimization.



Return parameter

In-out parameter

02 Biased random walks



Length

CLASS **Node2Vec** (edge_index, embedding_dim, walk_length, context_size, walks_per_node=1, p=1, q=1, num_negative_samples=1, num_nodes=None, sparse=False) [source]

The Node2Vec model from the "node2vec: Scalable Feature Learning for Networks" paper where random walks of length `walk_length` are sampled in a given graph, and node embeddings are learned via negative sampling optimization.

Return parameter

In-out parameter

Length of the RW to extract from a long sample

Sample **l=6**, **k=3**: {u, s4, s5, s6, s8, s9}

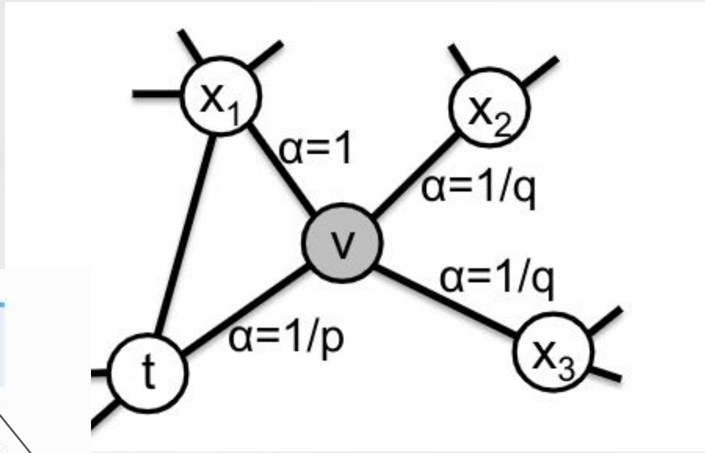
1. **u:** s4,s5,s6
2. **s4:** s5,s6,s8
3. **s5:** s6,s8,s9

02 Biased random walks

Length

CLASS **Node2Vec** (edge_index, embedding_dim, walk_length, context_size, walks_per_node=1, p=1, q=1, num_negative_samples=1, num_nodes=None, sparse=False) [source]

The Node2Vec model from the "node2vec: Scalable Feature Learning for Networks" paper where random walks of length `walk_length` are sampled in a given graph, and node embeddings are learned via negative sampling optimization.



`loader ( **kwargs )`

Return parameter

In-out parameter

Length of the RW to extract from a long sample

Sample **l=6**, **k=3**: {u, s4, s5, s6, s8, s9}

1. **u:** s4,s5,s6
2. **s4:** s5,s6,s8
3. **s5:** s6,s8,s9

02 Biased random walks

```
def loader(self, **kwargs):
    return DataLoader(range(self.adj.sparse_size(0)),
                      collate_fn=self.sample, **kwargs)
```

02 Biased random walks

```
def loader(self, **kwargs):
    return DataLoader(range(self.adj.sparse_size(0)),
                      collate_fn=self.sample, **kwargs)
```

Loader over list of nodes

02 Biased random walks

Loader over list of nodes

```python
def loader(self, **kwargs):
    return DataLoader(range(self.adj.sparse_size(0)),
                      collate_fn=self.sample, **kwargs)
```

```python
def sample(self, batch):
    if not isinstance(batch, torch.Tensor):
        batch = torch.tensor(batch)
    return self.pos_sample(batch), self.neg_sample(batch)
```

02 Biased random walks

```
def loader(self, **kwargs):
    return DataLoader(range(self.adj.sparse_size(0)),
                      collate_fn=self.sample, **kwargs)
```

Loader over list of nodes

```
def sample(self, batch):
    if not isinstance(batch, torch.Tensor):
        batch = torch.tensor(batch)
    return self.pos_sample(batch), self.neg_sample(batch)
```

A batch of indices

02 Biased random walks

```
def loader(self, **kwargs):
    return DataLoader(range(self.adj.sparse_size(0)),
                      collate_fn=self.sample, **kwargs)
```

Loader over list of nodes

```
def sample(self, batch):
    if not isinstance(batch, torch.Tensor):
        batch = torch.tensor(batch)
    return self.pos_sample(batch), self.neg_sample(batch)
```

A batch of indices

```
def pos_sample(self, batch):
    batch = batch.repeat(self.walks_per_node)

    rowptr, col, _ = self.adj.csr()
    rw = random_walk(rowptr, col, batch, self.walk_length, self.p, self.q)
    if not isinstance(rw, torch.Tensor):
        rw = rw[0]

    walks = []
    num_walks_per_rw = 1 + self.walk_length + 1 - self.context_size
    for j in range(num_walks_per_rw):
        walks.append(rw[:, j:j + self.context_size])
    return torch.cat(walks, dim=0)
```

```
def neg_sample(self, batch):
    batch = batch.repeat(self.walks_per_node * self.num_negative_samples)

    rw = torch.randint(self.adj.sparse_size(0),
                       (batch.size(0), self.walk_length))
    rw = torch.cat([batch.view(-1, 1), rw], dim=-1)


    walks = []
    num_walks_per_rw = 1 + self.walk_length + 1 - self.context_size
    for j in range(num_walks_per_rw):
        walks.append(rw[:, j:j + self.context_size])
    return torch.cat(walks, dim=0)
```

02 Biased random walks

```python
def loader(self, **kwargs):
    return DataLoader(range(self.adj.sparse_size(0)),
                      collate_fn=self.sample, **kwargs)
```

Loader over list of nodes

```python
def sample(self, batch):
    if not isinstance(batch, torch.Tensor):
        batch = torch.tensor(batch)
    return self.pos_sample(batch), self.neg_sample(batch)
```

A batch of indices

As before

```python
def pos_sample(self, batch):
    batch = batch.repeat(self.walks_per_node)

    rowptr, col, _ = self.adj.csr()
    rw = random_walk(rowptr, col, batch, self.walk_length, self.p, self.q)
    if not isinstance(rw, torch.Tensor):
        rw = rw[0]

    walks = []
    num_walks_per_rw = 1 + self.walk_length + 1 - self.context_size
    for j in range(num_walks_per_rw):
        walks.append(rw[:, j:j + self.context_size])
    return torch.cat(walks, dim=0)
```

```python
    def neg_sample(self, batch):
        batch = batch.repeat(self.walks_per_node * self.num_negative_samples)

        rw = torch.randint(self.adj.sparse_size(0),
                           (batch.size(0), self.walk_length))
        rw = torch.cat([batch.view(-1, 1), rw], dim=-1)


        walks = []
        num_walks_per_rw = 1 + self.walk_length + 1 - self.context_size
        for j in range(num_walks_per_rw):
            walks.append(rw[:, j:j + self.context_size])
        return torch.cat(walks, dim=0)
```

Initial nodes

torch.ops.torch_cluster.random_walk

02 Biased random walks

```python
def loader(self, **kwargs):
    return DataLoader(range(self.adj.sparse_size(0)),
                      collate_fn=self.sample, **kwargs)
```

Loader over list of nodes

```python
def sample(self, batch):
    if not isinstance(batch, torch.Tensor):
        batch = torch.tensor(batch)
    return self.pos_sample(batch), self.neg_sample(batch)
```

As before

A batch of indices

```python
def pos_sample(self, batch):
    batch = batch.repeat(self.walks_per_node)

    rowptr, col, _ = self.adj.csr()
    rw = random_walk(rowptr, col, batch, self.walk_length, self.p, self.q)
    if not isinstance(rw, torch.Tensor):
        rw = rw[0]

    walks = []
    num_walks_per_rw = 1 + self.walk_length + 1 - self.context_size
    for j in range(num_walks_per_rw):
        walks.append(rw[:, j:j + self.context_size])
    return torch.cat(walks, dim=0)
```

```python
def neg_sample(self, batch):
    batch = batch.repeat(self.walks_per_node * self.num_negative_samples)

    rw = torch.randint(self.adj.sparse_size(0),
                       (batch.size(0), self.walk_length))
    rw = torch.cat([batch.view(-1, 1), rw], dim=-1)


    walks = []
    num_walks_per_rw = 1 + self.walk_length + 1 - self.context_size
    for j in range(num_walks_per_rw):
        walks.append(rw[:, j:j + self.context_size])
    return torch.cat(walks, dim=0)
```

Initial nodes

A fake RW

torch.ops.torch_cluster.random_walk

02 Biased random walks

… notebook …

03 Simplifying the loss

Definition of the embedding f(v)

```
self.embedding = Embedding(N, embedding_dim, sparse=sparse)
```

torch.nn.Embedding

03 Simplifying the loss

Definition of the embedding f(v)

```
self.embedding = Embedding(N, embedding_dim, sparse=sparse)
```

torch.nn.Embedding

CLASS `torch.nn.Embedding(`*num_embeddings, embedding_dim, padding_idx=None, max_norm=None, norm_type=2.0, scale_grad_by_freq=False, sparse=False, _weight=None*`)` [SOURCE]

A simple lookup table that stores embeddings of a fixed dictionary and size.

This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

03 Simplifying the loss

Definition of the embedding f(v)

```
self.embedding = Embedding(N, embedding_dim, sparse=sparse)
```

torch.nn.Embedding

CLASS `torch.nn.Embedding(`*`num_embeddings, embedding_dim, padding_idx=None, max_norm=None, norm_type=2.0, scale_grad_by_freq=False, sparse=False, _weight=None`*`)` [SOURCE]

A simple lookup table that stores embeddings of a fixed dictionary and size.

This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

- **sparse** (*bool, optional*) – If `True`, gradient w.r.t. `weight` matrix will be a sparse tensor. See Notes for more details regarding sparse gradients.

03 Simplifying the loss

The loss maximizes the
probability of a neighborhood given u

$$P_f(v|u) := \frac{\exp(f(v)^T f(u))}{\sum_{w \in V} \exp(f(w)^T f(u))}$$

03 Simplifying the loss

The loss maximizes the
probability of a neighborhood given u

$$P_f(v|u) := \frac{\exp(f(v)^T f(u))}{\sum_{w \in V} \exp(f(w)^T f(u))}$$

$$P_f(N_s(u)|u) := \prod_{v \in N_s(u)} P_f(v|u)$$

Random walk

03 Simplifying the loss

The loss maximizes the
probability of a neighborhood given u

$$P_f(v|u) := \frac{\exp(f(v)^T f(u))}{\sum_{w\in V}\exp(f(w)^T f(u))}$$

Expensive

$$P_f(N_s(u)|u) := \prod_{v\in N_s(u)} P_f(v|u)$$

Random walk

03 Simplifying the loss

The loss maximizes the
probability of a neighborhood given u

$$P_f(v|u) := \frac{\exp(f(v)^T f(u))}{\sum_{w \in V} \exp(f(w)^T f(u))}$$

Expensive

$$P_f(N_s(u)|u) := \prod_{v \in N_s(u)} P_f(v|u)$$

Random walk

Negative sampling
(deepwalk uses hierarchical softmax)

- Manipulate the loss to

$$f(v)^T f(u) + \mathbb{E}_{w \sim p}(f(w)^T f(u))$$

03 Simplifying the loss

The loss maximizes the
probability of a neighborhood given u

$$P_f(v|u) := \frac{\exp(f(v)^T f(u))}{\sum_{w \in V} \exp(f(w)^T f(u))}$$

Expensive

$$P_f(N_s(u)|u) := \prod_{v \in N_s(u)} P_f(v|u)$$

Random walk

Negative sampling
(deepwalk uses hierarchical softmax)

- Manipulate the loss to
$$f(v)^T f(u) + \mathbb{E}_{w \sim p}(f(w)^T f(u))$$
- Approximate **p** by defining a **positive/negative class**

03 Simplifying the loss

The loss maximizes the
probability of a neighborhood given u

$$P_f(v|u) := \frac{\exp(f(v)^T f(u))}{\sum_{w \in V} \exp(f(w)^T f(u))}$$

Expensive

$$P_f(N_s(u)|u) := \prod_{v \in N_s(u)} P_f(v|u)$$

Random walk

Negative sampling
(deepwalk uses hierarchical softmax)

- Manipulate the loss to
$$f(v)^T f(u) + \mathbb{E}_{w \sim p}(f(w)^T f(u))$$
- Approximate **p** by defining a **positive/negative class**
- Sample +/- by sampling **true/fake RW**

03 Simplifying the loss

The loss maximizes the
probability of a neighborhood given u

$$P_f(v|u) := \frac{\exp(f(v)^T f(u))}{\sum_{w \in V} \exp(f(w)^T f(u))}$$

Expensive

$$P_f(N_s(u)|u) := \prod_{v \in N_s(u)} P_f(v|u)$$

Random walk

Negative sampling
(deepwalk uses hierarchical softmax)

- Manipulate the loss to
$$f(v)^T f(u) + \mathbb{E}_{w \sim p}(f(w)^T f(u))$$
- Approximate **p** by defining a **positive/negative class**
- Sample +/- by sampling **true/fake RW**

Details of math: *On word embeddings - Part 2: Approximating the Softmax*
https://ruder.io/word-embeddings-softmax/

03 Simplifying the loss

CLASS **Node2Vec** (edge_index, embedding_dim, walk_length, context_size, walks_per_node=1, p=1, q=1, num_negative_samples=1, num_nodes=None, sparse=False) [source]

The Node2Vec model from the "node2vec: Scalable Feature Learning for Networks" paper where random walks of length `walk_length` are sampled in a given graph, and node embeddings are learned via negative sampling optimization.

- **num_negative_samples** (*int*, *optional*) – The number of negative samples to use for each positive sample. (default: `1`)

03 Simplifying the loss

```python
def loss(self, pos_rw, neg_rw):
    r"""Computes the loss given positive and negative random walks."""

    # Positive loss.
    start, rest = pos_rw[:, 0], pos_rw[:, 1:].contiguous()

    h_start = self.embedding(start).view(pos_rw.size(0), 1,
                                         self.embedding_dim)
    h_rest = self.embedding(rest.view(-1)).view(pos_rw.size(0), -1,
                                                self.embedding_dim)

    out = (h_start * h_rest).sum(dim=-1).view(-1)
    pos_loss = -torch.log(torch.sigmoid(out) + EPS).mean()

    # Negative loss.
    start, rest = neg_rw[:, 0], neg_rw[:, 1:].contiguous()

    h_start = self.embedding(start).view(neg_rw.size(0), 1,
                                         self.embedding_dim)
    h_rest = self.embedding(rest.view(-1)).view(neg_rw.size(0), -1,
                                                self.embedding_dim)

    out = (h_start * h_rest).sum(dim=-1).view(-1)
    neg_loss = -torch.log(1 - torch.sigmoid(out) + EPS).mean()

    return pos_loss + neg_loss
```

03 Simplifying the loss

Divide first node from the rest of the RW

```python
def loss(self, pos_rw, neg_rw):
    r"""Computes the loss given positive and negative random walks."""

    # Positive loss.
    start, rest = pos_rw[:, 0], pos_rw[:, 1:].contiguous()

    h_start = self.embedding(start).view(pos_rw.size(0), 1,
                                         self.embedding_dim)
    h_rest = self.embedding(rest.view(-1)).view(pos_rw.size(0), -1,
                                                self.embedding_dim)

    out = (h_start * h_rest).sum(dim=-1).view(-1)
    pos_loss = -torch.log(torch.sigmoid(out) + EPS).mean()

    # Negative loss.
    start, rest = neg_rw[:, 0], neg_rw[:, 1:].contiguous()

    h_start = self.embedding(start).view(neg_rw.size(0), 1,
                                         self.embedding_dim)
    h_rest = self.embedding(rest.view(-1)).view(neg_rw.size(0), -1,
                                                self.embedding_dim)

    out = (h_start * h_rest).sum(dim=-1).view(-1)
    neg_loss = -torch.log(1 - torch.sigmoid(out) + EPS).mean()

    return pos_loss + neg_loss
```

03 Simplifying the loss

Divide first node from the rest of the RW

Compute the embeddings

```
def loss(self, pos_rw, neg_rw):
    r"""Computes the loss given positive and negative random walks."""

    # Positive loss.
    start, rest = pos_rw[:, 0], pos_rw[:, 1:].contiguous()

    h_start = self.embedding(start).view(pos_rw.size(0), 1,
                                         self.embedding_dim)
    h_rest = self.embedding(rest.view(-1)).view(pos_rw.size(0), -1,
                                                self.embedding_dim)

    out = (h_start * h_rest).sum(dim=-1).view(-1)
    pos_loss = -torch.log(torch.sigmoid(out) + EPS).mean()

    # Negative loss.
    start, rest = neg_rw[:, 0], neg_rw[:, 1:].contiguous()

    h_start = self.embedding(start).view(neg_rw.size(0), 1,
                                         self.embedding_dim)
    h_rest = self.embedding(rest.view(-1)).view(neg_rw.size(0), -1,
                                                self.embedding_dim)

    out = (h_start * h_rest).sum(dim=-1).view(-1)
    neg_loss = -torch.log(1 - torch.sigmoid(out) + EPS).mean()

    return pos_loss + neg_loss
```

03 Simplifying the loss

Divide first node from the rest of the RW

Compute the embeddings

Loss for the **positive class**: true RW

```
def loss(self, pos_rw, neg_rw):
    r"""Computes the loss given positive and negative random walks."""

    # Positive loss.
    start, rest = pos_rw[:, 0], pos_rw[:, 1:].contiguous()

    h_start = self.embedding(start).view(pos_rw.size(0), 1,
                                          self.embedding_dim)
    h_rest = self.embedding(rest.view(-1)).view(pos_rw.size(0), -1,
                                                  self.embedding_dim)

    out = (h_start * h_rest).sum(dim=-1).view(-1)
    pos_loss = -torch.log(torch.sigmoid(out) + EPS).mean()

    # Negative loss.
    start, rest = neg_rw[:, 0], neg_rw[:, 1:].contiguous()

    h_start = self.embedding(start).view(neg_rw.size(0), 1,
                                          self.embedding_dim)
    h_rest = self.embedding(rest.view(-1)).view(neg_rw.size(0), -1,
                                                  self.embedding_dim)

    out = (h_start * h_rest).sum(dim=-1).view(-1)
    neg_loss = -torch.log(1 - torch.sigmoid(out) + EPS).mean()

    return pos_loss + neg_loss
```

03 Simplifying the loss

Divide first node from the rest of the RW

Compute the embeddings

Loss for the **positive class**: true RW

The same, but for the
negative class (fake RW**)**

```
def loss(self, pos_rw, neg_rw):
    r"""Computes the loss given positive and negative random walks."""

    # Positive loss.
    start, rest = pos_rw[:, 0], pos_rw[:, 1:].contiguous()

    h_start = self.embedding(start).view(pos_rw.size(0), 1,
                                         self.embedding_dim)
    h_rest = self.embedding(rest.view(-1)).view(pos_rw.size(0), -1,
                                                self.embedding_dim)

    out = (h_start * h_rest).sum(dim=-1).view(-1)
    pos_loss = -torch.log(torch.sigmoid(out) + EPS).mean()

    # Negative loss.
    start, rest = neg_rw[:, 0], neg_rw[:, 1:].contiguous()

    h_start = self.embedding(start).view(neg_rw.size(0), 1,
                                         self.embedding_dim)
    h_rest = self.embedding(rest.view(-1)).view(neg_rw.size(0), -1,
                                                self.embedding_dim)

    out = (h_start * h_rest).sum(dim=-1).view(-1)
    neg_loss = -torch.log(1 - torch.sigmoid(out) + EPS).mean()

    return pos_loss + neg_loss
```

03 Simplifying the loss

Divide first node from the rest of the RW

Compute the embeddings

Loss for the **positive class**: true RW

The same, but for the
negative class (fake RW**)**

Total loss

```python
def loss(self, pos_rw, neg_rw):
    r"""Computes the loss given positive and negative random walks."""

    # Positive loss.
    start, rest = pos_rw[:, 0], pos_rw[:, 1:].contiguous()

    h_start = self.embedding(start).view(pos_rw.size(0), 1,
                                         self.embedding_dim)
    h_rest = self.embedding(rest.view(-1)).view(pos_rw.size(0), -1,
                                                self.embedding_dim)

    out = (h_start * h_rest).sum(dim=-1).view(-1)
    pos_loss = -torch.log(torch.sigmoid(out) + EPS).mean()

    # Negative loss.
    start, rest = neg_rw[:, 0], neg_rw[:, 1:].contiguous()

    h_start = self.embedding(start).view(neg_rw.size(0), 1,
                                         self.embedding_dim)
    h_rest = self.embedding(rest.view(-1)).view(neg_rw.size(0), -1,
                                                self.embedding_dim)

    out = (h_start * h_rest).sum(dim=-1).view(-1)
    neg_loss = -torch.log(1 - torch.sigmoid(out) + EPS).mean()

    return pos_loss + neg_loss
```

03 Simplifying the loss

Mean over the batch of RWs

Divide first node from the rest of the RW

Compute the embeddings

Loss for the **positive class**: true RW

The same, but for the **negative class** (fake RW)

Total loss

```
def loss(self, pos_rw, neg_rw):
    r"""Computes the loss given positive and negative random walks."""

    # Positive loss.
    start, rest = pos_rw[:, 0], pos_rw[:, 1:].contiguous()

    h_start = self.embedding(start).view(pos_rw.size(0), 1,
                                         self.embedding_dim)
    h_rest = self.embedding(rest.view(-1)).view(pos_rw.size(0), -1,
                                                self.embedding_dim)

    out = (h_start * h_rest).sum(dim=-1).view(-1)
    pos_loss = -torch.log(torch.sigmoid(out) + EPS).mean()

    # Negative loss.
    start, rest = neg_rw[:, 0], neg_rw[:, 1:].contiguous()

    h_start = self.embedding(start).view(neg_rw.size(0), 1,
                                         self.embedding_dim)
    h_rest = self.embedding(rest.view(-1)).view(neg_rw.size(0), -1,
                                                self.embedding_dim)

    out = (h_start * h_rest).sum(dim=-1).view(-1)
    neg_loss = -torch.log(1 - torch.sigmoid(out) + EPS).mean()

    return pos_loss + neg_loss
```

04 **node2vec:** Full implementation

```python
def forward(self, batch=None):
    """Returns the embeddings for the nodes in :obj:`batch`."""
    emb = self.embedding.weight
    return emb if batch is None else emb[batch]
```

04 **node2vec:** Full implementation

```
def forward(self, batch=None):
    """Returns the embeddings for the nodes in :obj:`batch`."""
    emb = self.embedding.weight
    return emb if batch is None else emb[batch]
```

```
def test(self, train_z, train_y, test_z, test_y, solver='lbfgs',
         multi_class='auto', *args, **kwargs):
    r"""Evaluates latent space quality via a logistic regression downstream
    task."""
    clf = LogisticRegression(solver=solver, multi_class=multi_class, *args,
                             **kwargs).fit(train_z.detach().cpu().numpy(),
                                           train_y.detach().cpu().numpy())
    return clf.score(test_z.detach().cpu().numpy(),
                     test_y.detach().cpu().numpy())
```

04 node2vec: Full implementation

… notebook …