CS224W - Colab 1

In this Colab, we will write a full pipeline for **learning node embeddings**. We will go through the following 3 steps.

To start, we will load a classic graph in network science, the [Karate Club Network](). We will explore multiple graph statistics for that graph.

We will then work together to transform the graph structure into a PyTorch tensor, so that we can perform machine learning over the graph.

Finally, we will finish the first learning algorithm on graphs: a node embedding model. For simplicity, our model here is simpler than DeepWalk / node2vec algorithms taught in the lecture. But it's still rewarding and challenging, as we will write it from scratch via PyTorch.

Now let's get started! This Colab should take roughly 1 hour to complete.

Note: Make sure to **restart and run all** before submission, so that the intermediate variables / packages will carry over to the next cell

1 Graph Basics

To start, we will load a classic graph in network science, the [Karate Club Network](). We will explore multiple graph statistics for that graph.

Setup

We will heavily use NetworkX in this Colab.

```
import networkx as nx
```

Zachary's karate club network

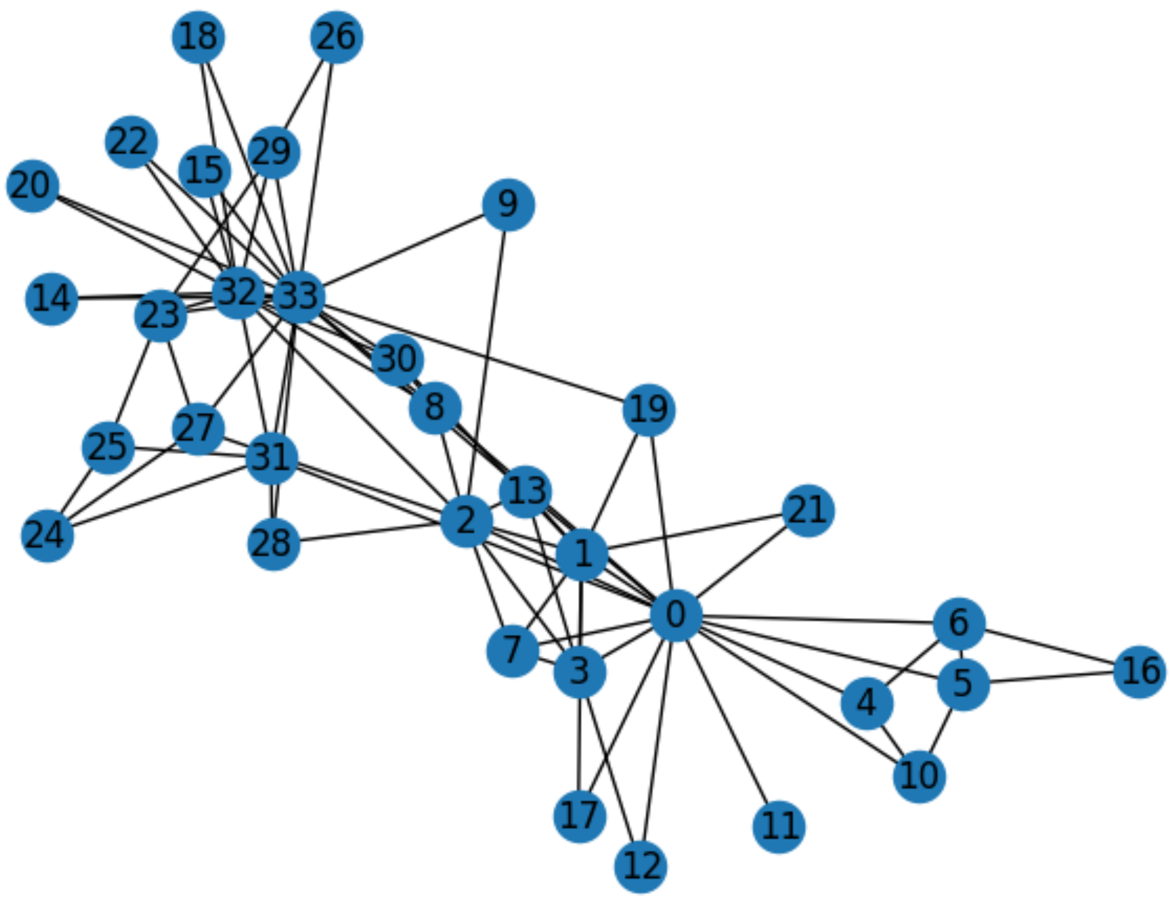
The [Karate Club Network](https://en.wikipedia.org/wiki/Zachary%27s_karate_club) is a graph which describes a social network of 34 members of a karate club and documents links between members who interacted outside the club.

```
G = nx.karate_club_graph()

# G is an undirected graph
type(G)
```

```
networkx.classes.graph.Graph
```

```
# Visualize the graph
nx.draw(G, with_labels = True)
```



Question 1: What is the average degree of the karate club

- List item
- List item

network? (5 Points)

```
def average_degree(num_edges, num_nodes):
  # TODO: Implement this function that takes number of edges
  # and number of nodes, and returns the average node degree of
  # the graph. Round the result to nearest integer (for example
  # 3.3 will be rounded to 3 and 3.7 will be rounded to 4)

  avg_degree = 0

  ############# Your code here ############
  avg_degree = (2*num_edges)/num_nodes
  avg_degree = round(avg_degree)
  #########################################

  return avg_degree

num_edges = G.number_of_edges()
num_nodes = G.number_of_nodes()
avg_degree = average_degree(num_edges, num_nodes)
print("Average degree of karate club network is {}".format(avg_degree))
```

```
Average degree of karate club network is 5
```

Question 2: What is the average clustering coefficient of the karate club network? (5 Points)

```
from networkx.algorithms.cluster import average_clustering
def average_clustering_coefficient(G):
  # TODO: Implement this function that takes a nx.Graph
  # and returns the average clustering coefficient. Round
  # the result to 2 decimal places (for example 3.333 will
  # be rounded to 3.33 and 3.7571 will be rounded to 3.76)

  avg_cluster_coef = 0

  ############# Your code here ############
  ## Note:
  ## 1: Please use the appropriate NetworkX clustering function
  avg_cluster_coef = nx.average_clustering(G)
  avg_cluster_coef = round(avg_cluster_coef,2)
  #########################################

  return avg_cluster_coef

avg_cluster_coef = average_clustering_coefficient(G)
print("Average clustering coefficient of karate club network is {}".format(avg_clu
```

```
Average clustering coefficient of karate club network is 0.57
```

Question 3: What is the PageRank value for node 0 (node with id 0) after one PageRank iteration? (5 Points)

Page Rank measures importance of nodes in a graph using the link structure of the web. A "vote" from an important page is worth more. Specifically, if a page i with importance r_i has d_i out-links, then each link gets $\frac{r_i}{d_i}$ votes. Thus, the importance of a Page j, represented as r_j is the sum of the votes on its in links.

$$r_j = \sum_{i \to j} \frac{r_i}{d_i}$$

, where d_i is the out degree of node i.

The PageRank algorithm (used by Google) outputs a probability distribution which represent the likelihood of a random surfer clicking on links will arrive at any particular page. At each time step, the random surfer has two options

- With prob. β, follow a link at random
- With prob. $1 - \beta$, jump to a random page

Thus, the importance of a particular page is calculated with the following PageRank equation:

$$r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1 - \beta) \frac{1}{N}$$

Please complete the code block by implementing the above PageRank equation for node 0.

Note - You can refer to more information from the slides here - http://snap.stanford.edu/class/cs224w-2020/slides/04-pagerank.pdf

```
def one_iter_pagerank(G, beta, r0, node_id):
  # TODO: Implement this function that takes a nx.Graph, beta, r0 and node id.
  # The return value r1 is one interation PageRank value for the input node.
  # Please round r1 to 2 decimal places.

  r1 = 0

  ############# Your code here ############
  ## Note:
  ## 1: You should not use nx.pagerank
  d_i = 0
  for neighbor in G.neighbors(node_id):
    d_i += beta * r0 / G.degree(neighbor)
  r1 += d_i + (1-beta)*(1/G.number_of_nodes())
  r1 = round(r1, 2)
  #########################################

  return r1

beta = 0.8
r0 = 1 / G.number_of_nodes()
node = 0
r1 = one_iter_pagerank(G, beta, r0, node)
print("The PageRank value for node 0 after one iteration is {}".format(r1))
```

```
The PageRank value for node 0 after one iteration is 0.13
```

Question 4: What is the (raw) closeness centrality for the karate club network node 5? (5 Points)

The equation for closeness centrality is $c(v) = \frac{1}{\sum_{u \neq v} \text{shortest path length between } u \text{ and } v}$

```
def closeness_centrality(G, node=5):
  # TODO: Implement the function that calculates closeness centrality
  # for a node in karate club network. G is the input karate club
  # network and node is the node id in the graph. Please round the
  # closeness centrality result to 2 decimal places.

  closeness = 0

  ## Note:
  ## 1: You can use networkx closeness centrality function.
  ## 2: Notice that networkx closeness centrality returns the normalized
  ## closeness directly, which is different from the raw (unnormalized)
  ## one that we learned in the lecture.
  closeness = nx.closeness_centrality(G, node)

  # unnormalized
  n = len(nx.node_connected_component(G,node))
  closeness = closeness / (n-1)
  closeness = round(closeness, 2)
  #########################################

  return closeness

node = 5
closeness = closeness_centrality(G, node=node)
print("The node 5 has closeness centrality {}".format(closeness))
```

```
The node 5 has closeness centrality 0.01
```

2 Graph to Tensor

We will then work together to transform the graph G into a PyTorch tensor, so that we can perform machine learning over the graph.

Setup

Check if PyTorch is properly installed

```
import torch
print(torch.__version__)
```

```
2.0.1+cu118
```

PyTorch tensor basics

We can generate PyTorch tensor with all zeros, ones or random values.

```
# Generate 3 x 4 tensor with all ones
ones = torch.ones(3, 4)
print(ones)

# Generate 3 x 4 tensor with all zeros
zeros = torch.zeros(3, 4)
print(zeros)

# Generate 3 x 4 tensor with random values on the interval [0, 1)
random_tensor = torch.rand(3, 4)
print(random_tensor)

# Get the shape of the tensor
print(ones.shape)
```

```
tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]])
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
tensor([[0.5628, 0.0540, 0.2396, 0.6112],
        [0.5992, 0.9141, 0.7382, 0.7669],
        [0.8531, 0.5519, 0.6613, 0.2419]])
torch.Size([3, 4])
```

PyTorch tensor contains elements for a single data type, the `dtype`.

```
# Create a 3 x 4 tensor with all 32-bit floating point zeros
zeros = torch.zeros(3, 4, dtype=torch.float32)
print(zeros.dtype)

# Change the tensor dtype to 64-bit integer
zeros = zeros.type(torch.long)
print(zeros.dtype)
```

```
torch.float32
torch.int64
```

Question 5: Get the edge list of the karate club network and transform it into `torch.LongTensor`. What is the `torch.sum` value of `pos_edge_index` tensor? (10 Points)

```
def graph_to_edge_list(G):
  # TODO: Implement the function that returns the edge list of
  # an nx.Graph. The returned edge_list should be a list of tuples
  # where each tuple is a tuple representing an edge connected
  # by two nodes.

  edge_list = []

  ############# Your code here ############
  edge_list = list(G.edges())
  #########################################

  return edge_list

def edge_list_to_tensor(edge_list):
  # TODO: Implement the function that transforms the edge_list to
  # tensor. The input edge_list is a list of tuples and the resulting
  # tensor should have the shape [2, len(edge_list)].

  edge_index = torch.tensor([])

  ############# Your code here ############
  edge_index = torch.tensor(edge_list, dtype=torch.long).t()
   #########################################

  return edge_index

pos_edge_list = graph_to_edge_list(G)
pos_edge_index = edge_list_to_tensor(pos_edge_list)
print("The pos_edge_index tensor has shape {}".format(pos_edge_index.shape))
print("The pos_edge_index tensor has sum value {}".format(torch.sum(pos_edge_index
```

```
The pos_edge_index tensor has shape torch.Size([2, 78])
The pos_edge_index tensor has sum value 2535
```

Question 6: Please implement following function that samples negative edges. Then answer which edges (edge_1 to edge_5) are the negative edges in the karate club network? (10 Points)

"Negative" edges refer to the edges/links that do not exist in the graph. The term "negative" is borrowed from "negative sampling" in link prediction. It has nothing to do with the edge weights.

For example, given an edge (src, dst), you should check that neither (src, dst) nor (dst, src) are edges in the Graph. If these hold true, then it is a negative edge.

```
import random

def sample_negative_edges(G, num_neg_samples):
  # TODO: Implement the function that returns a list of negative edges.
  # The number of sampled negative edges is num_neg_samples. You do not
  # need to consider the corner case when the number of possible negative edges
  # is less than num_neg_samples. It should be ok as long as your implementation
  # works on the karate club network. In this implementation, self loops should
  # not be considered as either a positive or negative edge. Also, notice that
  # the karate club network is an undirected graph, if (0, 1) is a positive
  # edge, do you think (1, 0) can be a negative one?

  neg_edge_list = []

  ############# Your code here ############
  nodes = list(G.nodes())
  sampled = 0
  while len(neg_edge_list) < num_neg_samples:
    src, dst = random.choice(nodes), random.choice(nodes)

    if src != dst and not G.has_edge(src,dst) and not G.has_edge(dst, src):
      neg_edge_list.append((src, dst))

  #########################################

  return neg_edge_list

# Sample 78 negative edges
neg_edge_list = sample_negative_edges(G, len(pos_edge_list))
```

```
# Transform the negative edge list to tensor
neg_edge_index = edge_list_to_tensor(neg_edge_list)
print("The neg_edge_index tensor has shape {}".format(neg_edge_index.shape))

# Which of following edges can be negative ones?
edge_1 = (7, 1)
edge_2 = (1, 33)
edge_3 = (33, 22)
edge_4 = (0, 4)
edge_5 = (4, 2)

############# Your code here ############
## Note:
## 1: For each of the 5 edges, print whether it can be negative edge
check_edges = [edge_1, edge_2, edge_3, edge_4, edge_5]
for edge in check_edges:
    if edge not in G.edges() and tuple(reversed(edge)) not in G.edges():
      print(f"{edge} can be a negative edge")
    else:
      print(f"{edge} cannot be a negative edge")

#########################################
```

```
The neg_edge_index tensor has shape torch.Size([2, 78])
(7, 1) cannot be a negative edge
(1, 33) can be a negative edge
(33, 22) cannot be a negative edge
(0, 4) cannot be a negative edge
(4, 2) can be a negative edge
```

3 Node Emebedding Learning

Finally, we will finish the first learning algorithm on graphs: a node embedding model.

Setup

```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

print(torch.__version__)
```

```
2.0.1+cu118
```

To write our own node embedding learning methods, we'll heavily use the `nn.Embedding` module in PyTorch. Let's see how to use `nn.Embedding`:

```
# Initialize an embedding layer
# Suppose we want to have embedding for 4 items (e.g., nodes)
# Each item is represented with 8 dimensional vector

emb_sample = nn.Embedding(num_embeddings=4, embedding_dim=8)
print('Sample embedding layer: {}'.format(emb_sample))
```

```
Sample embedding layer: Embedding(4, 8)
```

We can select items from the embedding matrix, by using Tensor indices

```
# Select an embedding in emb_sample
id = torch.LongTensor([1])
print(emb_sample(id))

# Select multiple embeddings
ids = torch.LongTensor([1, 3])
print(emb_sample(ids))

# Get the shape of the embedding weight matrix
shape = emb_sample.weight.data.shape
print(shape)

# Overwrite the weight to tensor with all ones
emb_sample.weight.data = torch.ones(shape)

# Let's check if the emb is indeed initilized
ids = torch.LongTensor([0, 3])
print(emb_sample(ids))
```

```
tensor([[-0.4940,  0.6790, -0.8130,  1.4449, -0.5692,  2.0466, -0.5834,  0.53
       grad_fn=<EmbeddingBackward0>)
tensor([[-0.4940,  0.6790, -0.8130,  1.4449, -0.5692,  2.0466, -0.5834,  0.53
        [-0.3065, -2.4054, -0.2976,  0.5876,  0.5912,  0.4335,  0.3929,  0.26
       grad_fn=<EmbeddingBackward0>)
torch.Size([4, 8])
tensor([[1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1.]], grad_fn=<EmbeddingBackward0>)
```

Now, it's your time to create node embedding matrix for the graph we have!

- We want to have **16 dimensional** vector for each node in the karate club network.
- We want to initalize the matrix under **uniform distribution**, in the range of $[0, 1)$. We suggest you using `torch.rand`.

```
# Please do not change / reset the random seed
torch.manual_seed(1)

def create_node_emb(num_node=34, embedding_dim=16):
  # TODO: Implement this function that will create the node embedding matrix.
  # A torch.nn.Embedding layer will be returned. You do not need to change
  # the values of num_node and embedding_dim. The weight matrix of returned
  # layer should be initialized under uniform distribution.

  emb = None

  ############# Your code here ############
  emb = torch.nn.Embedding(num_embeddings=num_node, embedding_dim=embedding_dim)
  emb.weight.data = torch.rand(emb.weight.data.shape)
  #########################################

  return emb

emb = create_node_emb()
ids = torch.LongTensor([0, 3])

# Print the embedding layer
print("Embedding: {}".format(emb))

# An example that gets the embeddings for node 0 and 3
print(emb(ids))
```

```
Embedding: Embedding(34, 16)
tensor([[0.2114, 0.7335, 0.1433, 0.9647, 0.2933, 0.7951, 0.5170, 0.2801, 0.83
         0.1185, 0.2355, 0.5599, 0.8966, 0.2858, 0.1955, 0.1808],
        [0.7486, 0.6546, 0.3843, 0.9820, 0.6012, 0.3710, 0.4929, 0.9915, 0.83
         0.4629, 0.9902, 0.7196, 0.2338, 0.0450, 0.7906, 0.9689]],
       grad_fn=<EmbeddingBackward0>)
```

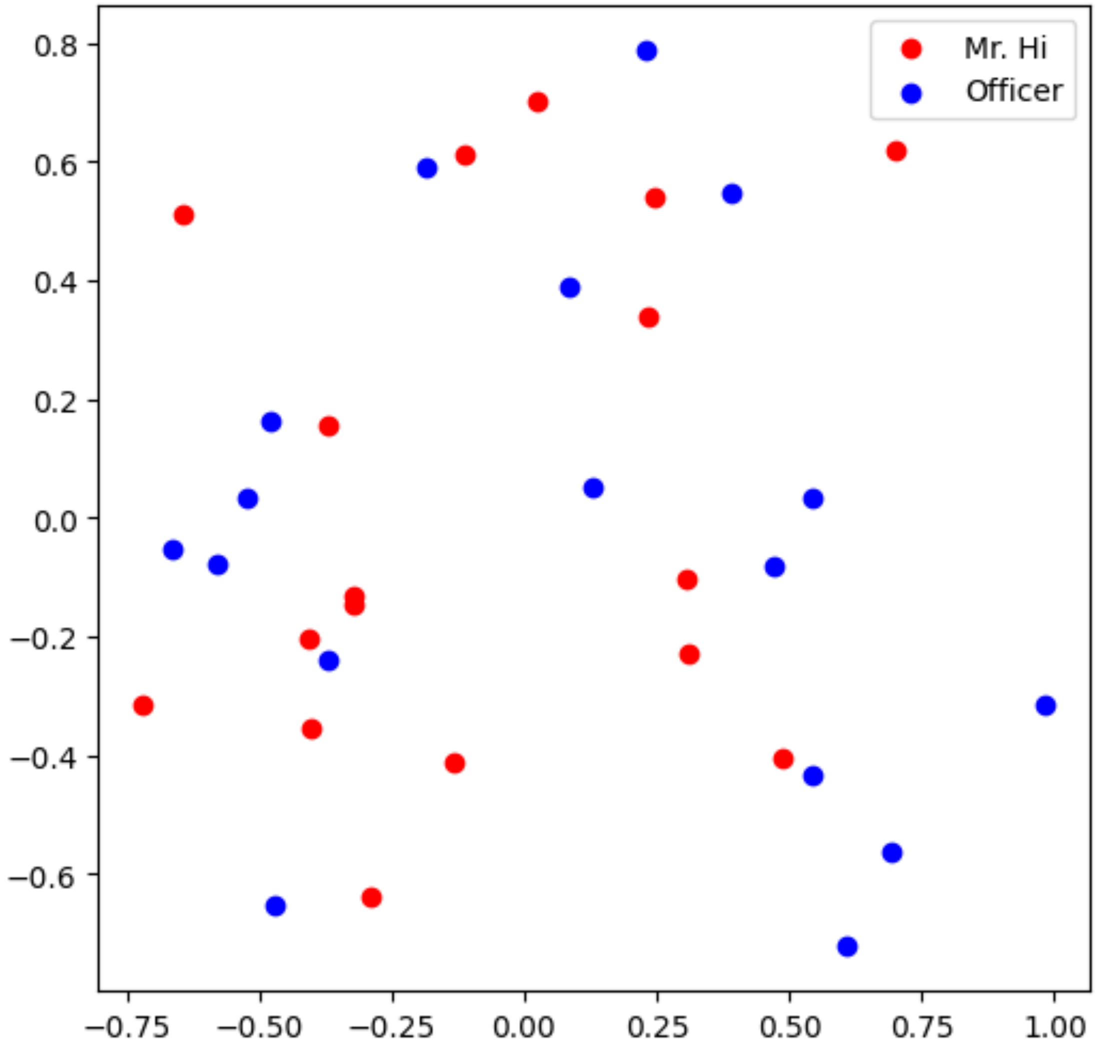
Visualize the initial node embeddings

One good way to understand an embedding matrix, is to visualize it in a 2D space. Here, we have implemented an embedding visualization function for you. We first do PCA to reduce the dimensionality of embeddings to a 2D space. Then we visualize each point, colored by the community it belongs to.

```
def visualize_emb(emb):
```

```
    X = emb.weight.data.numpy()
    pca = PCA(n_components=2)
    components = pca.fit_transform(X)
    plt.figure(figsize=(6, 6))
    club1_x = []
    club1_y = []
    club2_x = []
    club2_y = []
    for node in G.nodes(data=True):
      if node[1]['club'] == 'Mr. Hi':
        club1_x.append(components[node[0]][0])
        club1_y.append(components[node[0]][1])
      else:
        club2_x.append(components[node[0]][0])
        club2_y.append(components[node[0]][1])
    plt.scatter(club1_x, club1_y, color="red", label="Mr. Hi")
    plt.scatter(club2_x, club2_y, color="blue", label="Officer")
    plt.legend()
    plt.show()

  # Visualize the initial random embeddding
  visualize_emb(emb)
```


Mr. Hi
Officer
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

Question 7: Training the embedding! What is the best performance you can get? Please report both the best loss and accuracy on Gradescope. (20 Points)

We want to optimize our embeddings for the task of classifying edges as positive or negative. Given an edge and the embeddings for each node, the dot product of the embeddings, followed by a sigmoid, should give us the likelihood of that edge being either positive (output of sigmoid > 0.5) or negative (output of sigmoid < 0.5).

Note that we're using the functions you wrote in the previous questions, *as well as the variables initialized in previous cells*. If you're running into issues, make sure your answers to questions 1-6 are correct.

```
from torch.optim import SGD
import torch.nn as nn

def accuracy(pred, label):
  # TODO: Implement the accuracy function. This function takes the
  # pred tensor (the resulting tensor after sigmoid) and the label
  # tensor (torch.LongTensor). Predicted value greater than 0.5 will
  # be classified as label 1. Else it will be classified as label 0.
  # The returned accuracy should be rounded to 4 decimal places.
  # For example, accuracy 0.82956 will be rounded to 0.8296.

  accu = 0.0

  ############# Your code here ############
  correct_prediction = torch.sum((torch.round(pred) == label).float())
  total_predictions = label.numel()
  accu = correct_prediction / total_predictions
  accu = round(accu.item(), 4)
  #########################################

  return accu

def train(emb, loss_fn, sigmoid, train_label, train_edge):
  # TODO: Train the embedding layer here. You can also change epochs and
  # learning rate. In general, you need to implement:
  # (1) Get the embeddings of the nodes in train_edge
```

```
  # (2) Dot product the embeddings between each node pair
  # (3) Feed the dot product result into sigmoid
  # (4) Feed the sigmoid output into the loss_fn
  # (5) Print both loss and accuracy of each epoch
  # (6) Update the embeddings using the loss and optimizer
  # (as a sanity check, the loss should decrease during training)

  epochs = 500
  learning_rate = 0.1

  optimizer = SGD(emb.parameters(), lr=learning_rate, momentum=0.9)

  for i in range(epochs):

    ############# Your code here ############
    optimizer.zero_grad()

    # (1) Get Embeddings
    src_node_embeddings = emb(train_edge[0])
    dst_node_embeddings = emb(train_edge[1])

    # (2) Dot product
    dot_product = (src_node_embeddings * dst_node_embeddings).sum(dim=1)

    # (3) Apply sigmoid
    pred = sigmoid(dot_product)

    # (4) Feed to loss
    loss = loss_fn(pred, train_label)

    # Backpropagation
    loss.backward()
    optimizer.step()

    # (5) print loss and accuracy
    acc = accuracy(pred, train_label)
    print(f"Epoch {i+1}/{epochs}, Loss: {loss.item()}, Accuracy: {acc}")
    #########################################

loss_fn = nn.BCELoss()
sigmoid = nn.Sigmoid()

print(pos_edge_index.shape)

# Generate the positive and negative labels
```

```
pos_label = torch.ones(pos_edge_index.shape[1], )
neg_label = torch.zeros(neg_edge_index.shape[1], )

# Concat positive and negative labels into one tensor
train_label = torch.cat([pos_label, neg_label], dim=0)

# Concat positive and negative edges into one tensor
# Since the network is very small, we do not split the edges into val/test sets
train_edge = torch.cat([pos_edge_index, neg_edge_index], dim=1)
print(train_edge.shape)

train(emb, loss_fn, sigmoid, train_label, train_edge)
```

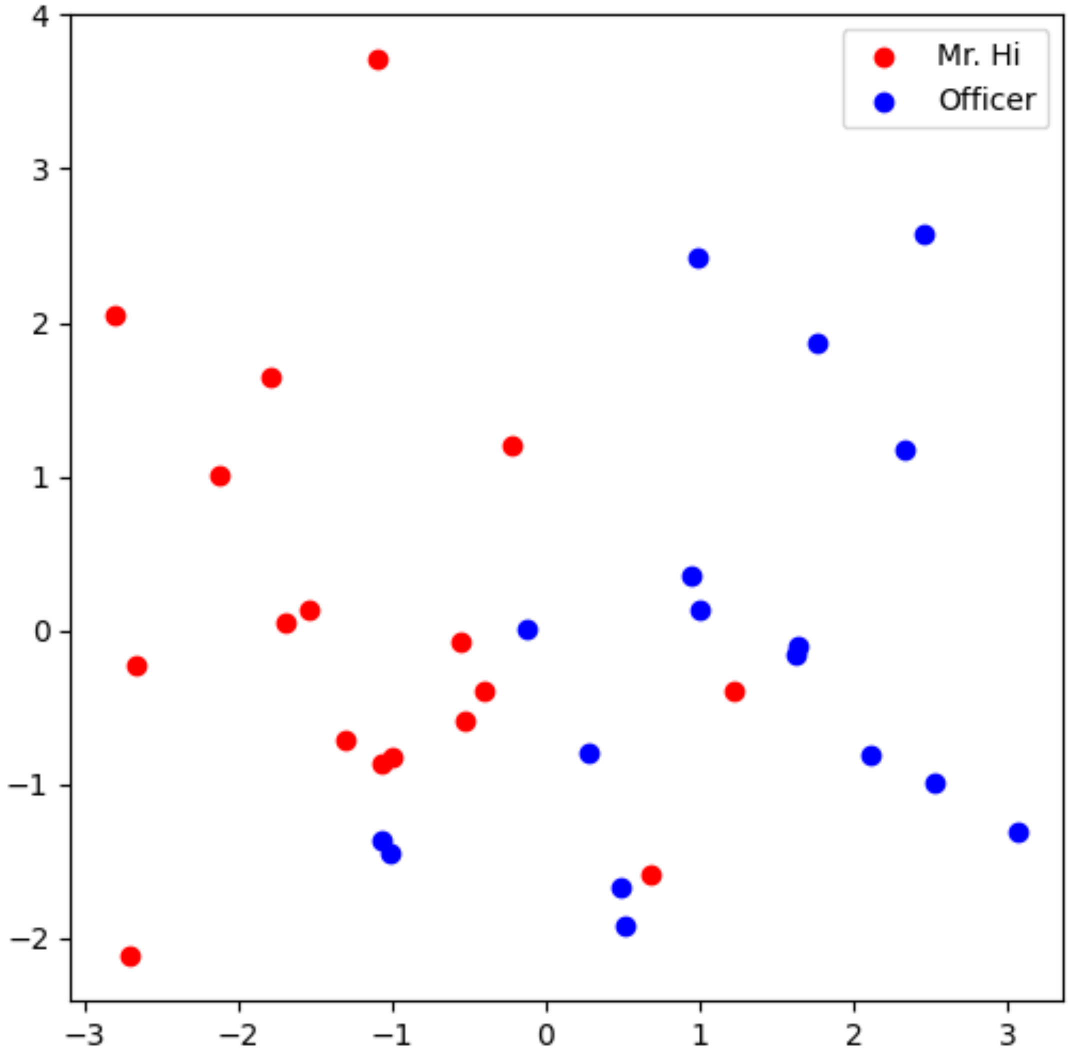
```
torch.Size([2, 78])
torch.Size([2, 156])
Epoch 1/500, Loss: 0.018004391342401505, Accuracy: 1.0
Epoch 2/500, Loss: 0.017998388037085533, Accuracy: 1.0
Epoch 3/500, Loss: 0.01798698492348194, Accuracy: 1.0
Epoch 4/500, Loss: 0.017970740795135498, Accuracy: 1.0
Epoch 5/500, Loss: 0.0179501473903656, Accuracy: 1.0
Epoch 6/500, Loss: 0.017925666645169258, Accuracy: 1.0
Epoch 7/500, Loss: 0.017897704616189003, Accuracy: 1.0
Epoch 8/500, Loss: 0.017866630107164383, Accuracy: 1.0
Epoch 9/500, Loss: 0.017832791432738304, Accuracy: 1.0
Epoch 10/500, Loss: 0.017796479165554047, Accuracy: 1.0
Epoch 11/500, Loss: 0.01775798574090004, Accuracy: 1.0
Epoch 12/500, Loss: 0.01771753840148449, Accuracy: 1.0
Epoch 13/500, Loss: 0.017675379291176796, Accuracy: 1.0
Epoch 14/500, Loss: 0.01763170026242733, Accuracy: 1.0
Epoch 15/500, Loss: 0.017586680129170418, Accuracy: 1.0
Epoch 16/500, Loss: 0.017540501430630684, Accuracy: 1.0
Epoch 17/500, Loss: 0.017493292689323425, Accuracy: 1.0
Epoch 18/500, Loss: 0.017445191740989685, Accuracy: 1.0
Epoch 19/500, Loss: 0.01739632524549961, Accuracy: 1.0
Epoch 20/500, Loss: 0.01734679378569126, Accuracy: 1.0
Epoch 21/500, Loss: 0.017296697944402695, Accuracy: 1.0
Epoch 22/500, Loss: 0.017246128991246223, Accuracy: 1.0
Epoch 23/500, Loss: 0.01719515584409237, Accuracy: 1.0
Epoch 24/500, Loss: 0.01714385487139225, Accuracy: 1.0
Epoch 25/500, Loss: 0.017092285677790642, Accuracy: 1.0
Epoch 26/500, Loss: 0.01704050973057747, Accuracy: 1.0
Epoch 27/500, Loss: 0.01698857918381691, Accuracy: 1.0
Epoch 28/500, Loss: 0.0169365257024765, Accuracy: 1.0
Epoch 29/500, Loss: 0.01688440702855587, Accuracy: 1.0
Epoch 30/500, Loss: 0.016832249239087105, Accuracy: 1.0
Epoch 31/500, Loss: 0.01678009144961834, Accuracy: 1.0
Epoch 32/500, Loss: 0.016727954149246216, Accuracy: 1.0
Epoch 33/500, Loss: 0.01667586900293827, Accuracy: 1.0
Epoch 34/500, Loss: 0.016623858362436295, Accuracy: 1.0
```

```
Epoch 35/500, Loss: 0.016571946442127228, Accuracy: 1.0
Epoch 36/500, Loss: 0.016520146280527115, Accuracy: 1.0
Epoch 37/500, Loss: 0.016468478366732597, Accuracy: 1.0
Epoch 38/500, Loss: 0.01641695387661457, Accuracy: 1.0
Epoch 39/500, Loss: 0.016365593299269676, Accuracy: 1.0
Epoch 40/500, Loss: 0.016314392909407616, Accuracy: 1.0
Epoch 41/500, Loss: 0.01626337505877018, Accuracy: 1.0
Epoch 42/500, Loss: 0.016212543472647667, Accuracy: 1.0
Epoch 43/500, Loss: 0.016161905601620674, Accuracy: 1.0
Epoch 44/500, Loss: 0.016111472621560097, Accuracy: 1.0
Epoch 45/500, Loss: 0.016061248257756233, Accuracy: 1.0
Epoch 46/500, Loss: 0.01601123809814453, Accuracy: 1.0
Epoch 47/500, Loss: 0.01596144586801529, Accuracy: 1.0
Epoch 48/500, Loss: 0.015911875292658806, Accuracy: 1.0
Epoch 49/500, Loss: 0.015862520784139633, Accuracy: 1.0
Epoch 50/500, Loss: 0.01581340655684471, Accuracy: 1.0
Epoch 51/500, Loss: 0.015764517709612846, Accuracy: 1.0
Epoch 52/500, Loss: 0.015715857967734337, Accuracy: 1.0
Epoch 53/500, Loss: 0.015667429193854332, Accuracy: 1.0
Epoch 54/500, Loss: 0.015619237907230854, Accuracy: 1.0
Epoch 55/500, Loss: 0.015571284107863903, Accuracy: 1.0
Epoch 56/500, Loss: 0.015523559413850307, Accuracy: 1.0
Epoch 57/500, Loss: 0.015476076863706112, Accuracy: 1.0
Epoch 58/500, Loss: 0.01542882714420557, Accuracy: 1.0
```

Visualize the final node embeddings

Visualize your final embedding here! You can visually compare the figure with the previous embedding figure. After training, you should oberserve that the two classes are more evidently separated. This is a great sanitity check for your implementation as well.

```
# Visualize the final learned embedding
visualize_emb(emb)
```



Submission

In order to get credit, you must go submit your answers on Gradescope.