Matemática Aplicada a Tecnologías de la Información
Curso 2023/24

Práctica 8: Graph Filters and Neural Networks (Parte 1)

Esta práctica es una primera aproximación al aprendizaje con filtros (GF) y *Graph Neural Networks* (GNN). Nuestro objetivo en esta saga de prácticas es evidenciar:

- GFs producen mejores resultados que cualquier parametrización lineal y las GNNs producen mejores resultados que cualquier perceptrón multicapa.
- Las GNNs son mejores que los GFs.
- Una GNN entrenada en un graco con una cantidad determinada de nodos puede ser ejecutada en un grafo mayor y seguir dando buenos resultados.

Ejercicio 1 (Generación de un grafo) Vamos a generar un grafo basado en un modelo estocástico en bloque (SBM). Los grafos SBM son grafos sin pesos y no dirigidos compuesto por C comunidades tales que los nodos en una comunidad se conectan con una probabilidad determinada (probabilidad intra comunidad, p_{c_i,c_i}) y los nodos en comunidades diferentes lo hacen con otra probabilidad (probabilidad entre comunidades, p_{c_i,c_j}). Define una función que genere este tipo de grafos completando la siguiente función:

```
import numpy as np


def sbm(n, c, p_intra, p_inter):
"""
n: numero de nodos
c: numero de comunidades
p_intra: probabilidad intra comunidad
p_inter: probabilidad entre comunidades
"""

    # asignar una comunidad a cada nodo.
    # comunidad tiene que ser una lista con la etiqueta de
    #cada comunidad para cada nodo.Las comunidades deben
    # estar balanceadas.
    comunidad =

    # Asegurate que el vector tenga longitud n.
    comunidad = comunidad[0:n]
```

```
    # Lo hacemos un vector columna.
    comunidad = np.expand_dims(comunidad, 1)


    # Generar una matriz de booleanos que indican si dos nodos
    # estan en la misma comunidad.
    intra = comunidad == comunidad.T

    # Generar una matriz de booleanos indicando si dos nodos estan
    # en comunidades distintas.
    inter = np.logical_not(intra)

    # Generar una matriz con entradas aleatorias entre 0 y 1.
    random =


    # Inicializa la matriz de adyacencia
    # del grafo con una matriz de ceros.
    grafo =

    # Asigna una arista una arista cuando se cumpla
    # la condicion de probabilidad usando la matriz
    # de valores aleatorios random para los intra-comunidad.
    # Al ser simetrico, actualiza solo la parte triangular superior.
    grafo[_____] = 1

    # Asigna una arista una arista cuando se cumpla
    # la condicion de probabilidad usando la matriz
    # de valores aleatorios random para los entre-comunidad.
    # Al ser simetrico, actualiza solo la parte triangular superior.
    grafo[_____] = 1

    # Haz la matriz simetrica
    grafo += grafo.T

    return grafo
```

Ejemplo:

```
S = sbm(n=50, c=5, p_intra=0.6, p_inter=0.2)
Out:
array([[0., 1., 0., ..., 1., 0., 0.],
       [1., 0., 1., ..., 0., 0., 0.],
       [0., 1., 0., ..., 0., 0., 0.],
       ...,
       [1., 0., 0., ..., 0., 1., 1.],
       [0., 0., 0., ..., 1., 0., 1.],
```

```
        [0., 0., 0., ..., 1., 1., 0.]])
```

Ejercicio 2 (Normalizar por el autovalor de mayor valor) Es interesante normalizar el espacio propio del grafo. Para ello, define una función que, dada una matriz de adyacencia de un grafo, calcule los autovalores de la matriz y luego divida por el máximo de los autovalores (en mayor absoluto).

```
def normalizar(grafo):

    # Calcula los autovalores.
    autovalores, _ =

    # Normaliza dividiendo por el autovalor con mayor valor absoluto.
    return
```

Ejemplo:

```
S = normaliza(S)
Out:
array([[0., 0.0750937, 0.0750937, ..., 0.0750937, 0.,0.],
       [0.0750937, 0., 0., ..., 0., 0.,0.],
       [0.0750937, 0., 0., ..., 0.0750937, 0.,0.],
       ...,
       [0.0750937, 0., 0.0750937, ..., 0., 0.,0.],
       [0., 0., 0., ..., 0., 0.,0.0750937],
       [0., 0., 0., ..., 0., 0.0750937,0.]])
```

Ejercicio 3 (Generación de datos) En esta práctica vamos a trabajar con un problema de localización de fuentes. Este problema consiste en identificar las fuentes en un proceso de difusión en un grafo a partir de la observación del proceso en un determinado instante $t = T$.

Considera un grafo $G = (V, A, S)$ con conjunto de vértices V con $|V| = N$, conjunto de aristas A y operador de cambio S. Sea $\mathcal{S} = \{s_1, \ldots, s_M\}$ un conjunto de M fuentes $s_i \in V$. En el momento $t = 0$, la señal del grafo $z_0 \in \mathbb{R}^N$ viene dado por

$$[z_0]_i = z \sim \mathcal{U}(a, b) \text{ si } i \in \mathcal{S} \text{ y } 0 \text{ en cualquier otro caso.}$$

Donde $\mathcal{U}$(a,b) es una distribución uniforme en el intervalo $[a, b]$. Por otro lado, para $t > 0$, z_t es el resultado de la difusión para z_{t-1} del grafo. Es decir,

$$z_t = S \cdot z_{t-1} + w_t$$

donde $w_t \in \mathbb{R}^N$ es un ruido Gausiano. Define una función que genere el proceso de difusión descrito con $a = 0$, $b = 10$ y $T = 4$, y el ruido gausiano tiene media fija 0 y varianza 10^{-3}. Puedes completar los huecos de la siguiente función:

```python
def genera_difusion(Grafo, n_muestras, n_fuentes):

    # Calcula el numero de nodos del grafo
    n =

    # Inicializa un tensor de ceros para guardar las muestras
    # de tamano n_muestras x n x T+1 x 1.
    z =

    for i in range(n_muestras):

        # Toma n_fuentes de manera aleatoria de los n nodos
        fuentes =

        # define z_0 para cada muestra
        z[i, fuentes, 0, 0] = np.random.uniform(_,_,_)

    # media y varianza del ruido
    mu = np.zeros(n)
    sigma = np.eye(n) * 1e-3

    for t in range(4):

        # Genera el ruido
        ruido = np.random.multivariate_normal(_,_,_)

        # Genera z_t
        z[:, :, t + 1] = grafo @ z[:, :, t] + np.expand_dims(ruido, -1)

    # Transpon la dimension de manera que sea
    # n_muestras x tiempo x n x 1
    z = z.transpose((0, 2, 1, 3))

    # "squeeze" la dimension al tener solo dimension 1.
    return z.squeeze()
```

Ejemplo:

```python
z = genera_difusion(S, 2100, 10)
np.shape(z)
(2100, 11, 50)
```

Ejercicio 4 (Obtención de los datos) Dada una observación z_T en un instante $t = T$, nuestro objetivo es identificar las fuentes $s_i \in \mathcal{S}$. Por lo tanto, tenemos que separar los datos en `input` y `output`. Completa la siguiente función:

```
def data_from_diffusion(z):

    # Permuta las muestras de z
    z =

    # define el tensor del output
    y = np.expand_dims(z[:, 0, :], 1)

    # inicializa el tensor del input como una matriz
    # de ceros de la misma dimension que y.
    x =

    # actualiza el tensor input como x = z_4
    for i, muestra in enumerate(z):
        x[i] = muestra[4]

    # squeeze la dimension del tiempo.
    return x.squeeze(), y.squeeze()
```

Luego, separa en conjunto entrenamiento y test, y genera las variables `xTrain`, `yTrain`, `xTest`, `yTest`. Finalmente, lo convertimos en tensores de la libreria `PyTorch`.

```
import torch
xTrain = torch.tensor(xTrain)
yTrain = torch.tensor(yTrain)
xTest = torch.tensor((xTest)
yTest = torch.tensor(yTest)
```

Ejercicio 5 (GFs) Una parametrización útil para procesar señales en grafos son los filtros convolucionales. Para definir esta operación, se introduce un filtro de orden K junto con los coeficientes h_k que agrupamos en el vector $h = [h_0, \ldots, h_{K-1}]$. En este caso, el filtro convolucional viene dado por:

$$\Phi(x; h, S) = H(S)x = \sum_{k=0}^{K-1} S^k x h_k$$

donde la salida es también una señal de un grafo y los parámetros entrenables son los coeficientes h_k. Para definir un GF en `Pytorch`, necesitamos dos componentes: una función que implementa el operador de cambio y la suma, y un módulo `torch.nn.module` que define el GF como una arquitectura que puede entrenarse. Completa el siguiente código:

```
def FuncionFiltro(h, S, x):

    K = h.shape[0] # orden del filtro
    B = x.shape[0] # tamano del batch
    N = x.shape[1] # numero de nodos

    x = x.reshape([B, 1, N])
    S = S.reshape([1, N, N])
    z = x

    for k in range(1, K):

        # paso de difusion, S^k*x
        x = torch.matmul(_, _)
        xS = x.reshape([B, 1, N])

        # Concatena S^k*x en el tensor z
        z = torch.cat((_, _), dim=1)

    # multiplica z y h
    y = torch.matmul(z.permute(0, 2, 1).reshape([B, N, K]), h)

    return y
```

Con la función anterior, podemos implementar el módulo.

```
import torch.nn as nn
import math

class GF(nn.Module):
    def __init__(self, grafo, k):

        # Inicializar
        super().__init__()

        # Guardar los hiperparametros
        self.grafo = torch.tensor(grafo)
        self.n = grafo.shape[0]
        self.k = k

        # Define e inicializa los pesos
        self.weight = nn.Parameter(torch.randn(self.k))
        self.reset_parameters()

    def reset_parameters(self):
        stdv = 1. / math.sqrt(self.k)
        self.weight.data.uniform_(-stdv, stdv)
```

```python
    def forward(self, x):
        return FilterFunction(self.weight, self.grafo, x)
```

Ejemplo:

```python
gF = GF(S, 8)
```

Solo quedaría entrenar el GF con los datos generados en ejercicios anteriores.