GNNs

November 30, 2023

1 Redes neuronales gráficas

Las redes neuronales gráficas generalizan las redes neuronales recurrenctes, convolucionales o atencionales. Este tipo de redes, trabajan con datos de entrada que tienen una estructura relacional representada en una gráfica $G = (V, E)$, donde los nodos $n \in V$ están asociados a vectores x_n y las aristas $e_{i,j} \in E$ indican la existencia de una relación entre los vectores x_n.

Un ejemplo de estos son las redes convolucionales, en donde un píxel es representado por un vector y se asumen que los píxeles vecinos están relacionados, por tanto, se define una estructura de malla (grid). Por su parte, las redes recurrentes asumen una gráfica dirigida donde un vector tiene como vecino al vector del elemento que anterior a éste en la secuencia. Finalmente, las redes atencionales (usadas en los Transformers) asumen una gráfica completamente conectada.

Para ejemplificar el uso de este tipo de redes, definimos un problema sencillo: las clasificación de nubes de puntos que representan 2-variedades: la esfera y el toro. Para poder hacer la clasificación, definimos relaciones entre los puntos a partir de los vecinos más cercanos y clasificamos usando una Red neuronal gráfica o Graph Neural Network (GNN).

[1]:
```
import numpy as np
import torch
import torch.nn as nn
import torch_geometric
import torch_geometric.nn as geo_nn
import matplotlib.pyplot as plt
from tqdm import tqdm
from torch_cluster import knn_graph
from sklearn.decomposition import PCA
from sklearn.metrics import classification_report
#Utilizamos cuda
device = 'device' if torch.cuda.is_available() else 'cpu'
```

1.1 Generando figuras para clasificación

Antes de adentrarnos en los detalles de la red neuronal gráfica, definimos el problema a tratar: nuestros datos son nubes de puntos que representan a esferas y toros. Cada punto es un vector en $\mathbb{R}^3$. Definimos funciones para generar ambas variedades en base a parámetros determinados.

1.1.1 Creando esferas

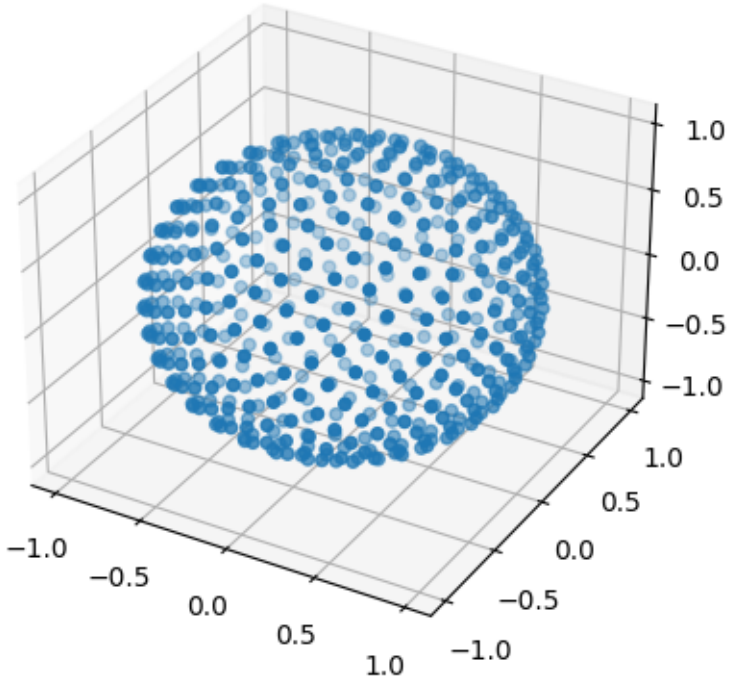
Para crear las esferas, definimos los parámetros de radio y distancia de tal forma que las esferas que generemos dependeran únicamente de estos dos parámetros. Por cada esfera tomamos un número fijo de puntos (500).

[2]:
```
def create_sphere(radius = 1,  dist = 1, n_samples = 500, noise = 0):
    """Crea una muestra de puntos en la 2-esfera"""
    n_samples = int(n_samples)
    indices = np.arange(0, n_samples, dtype=float)
    phi = np.arccos(1 - 2 * indices / n_samples)
    theta = np.pi * (1 + 5**0.5) * indices
    x, y, z = radius * np.cos(theta) * np.sin(phi), radius * np.sin(theta) * np.
 ↪sin(phi), dist*radius*np.cos(phi);
    points  = np.vstack( (x, y, z)).T + noise*np.random.normal(0,1,(500,3))

    return torch.Tensor(points)

#Visualización de esfera
Sphere = create_sphere(radius=1)
print(Sphere.shape)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(Sphere[:,0], Sphere[:,1], Sphere[:,2])
plt.show()
```

```
torch.Size([500, 3])
```



1.1.2 Creando toros

La creación de los toros es similar, pero requerimos de los parámetros R y r. De igual forma, se determina un número dado de puntos (500).

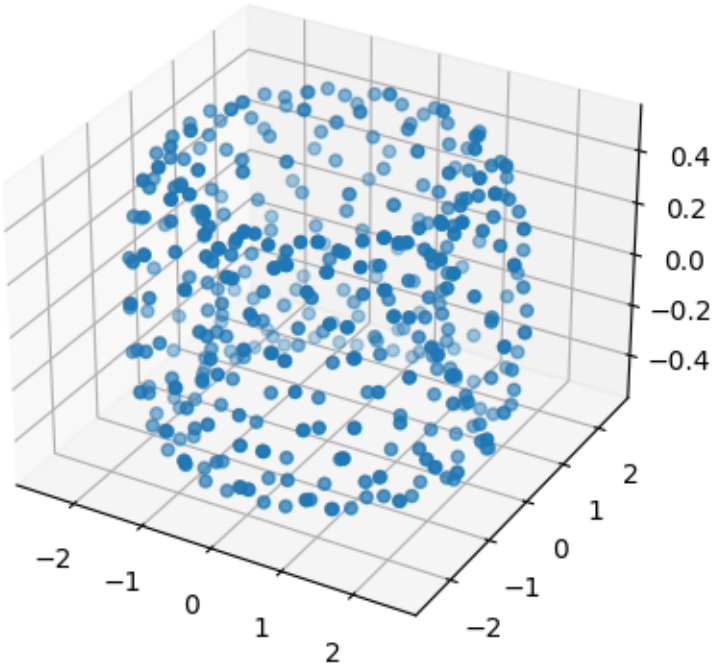
[3]:
```
def create_torus(r = 0.5, R = 2,  n_samples = 500, noise = 0):
    indices = np.arange(0, n_samples, dtype=float)
    phi = 0.1 * indices #np.arccos(1 - 2 * indices / n_samples)
    theta = np.pi * (1 + 5**0.5) * indices
    x = (R + r * np.cos(phi)) * np.cos(theta)
    y = (R + r * np.cos(phi)) * np.sin(theta)
    z = r * np.sin(phi)
    points  = np.vstack( (x, y, z)).T + noise*np.random.normal(0,1,(500,3))

    return torch.Tensor(points)

Torus = create_torus()

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(Torus[:,0], Torus[:,1], Torus[:,2])
plt.show()
```

```
print(Torus.shape)
```




```
torch.Size([500, 3])
```

1.2 De nubes de puntos a gráficas

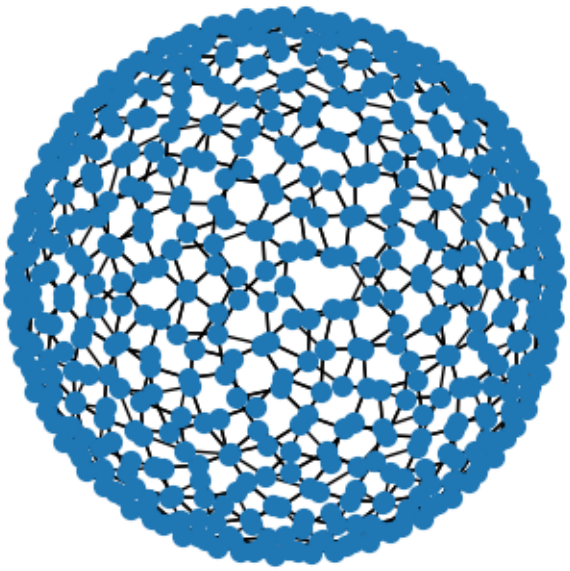
Las GNNs requieren de un conjunto de puntos $X \in \mathbb{R}^{N\times d}$ (donde N es el número de puntos y d la dimensión de cada uno de estos puntos) y de una matriz de adyacencia que determine las relaciones establecidas entre los puntos $A \in \mathbb{R}^{N\times N}$. De fal forma que una GNN será una función $f(X, A)$ que tome tanto los puntos como la información de las adyacencias entre estos puntos.

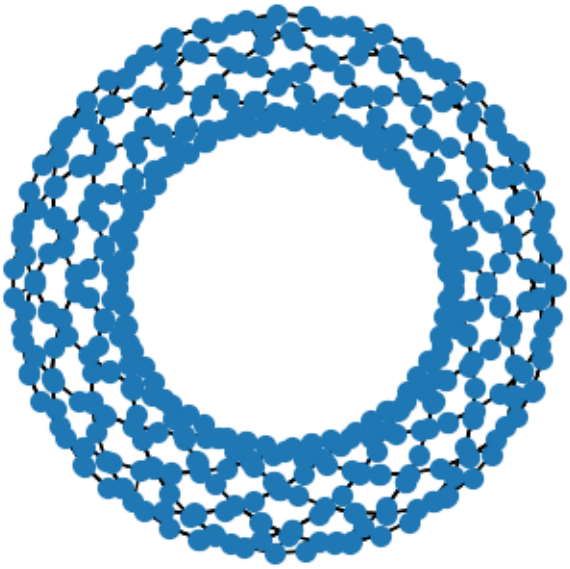
Para definir las adyacencias en las nubes de puntos que hemos definido sobre la esfera y el toro, utilizamos el método de k vecinos más cercanos. En este caso, tomamos $k = 3$. Es decir, los puntos serán adyacentes si se encuentran dentro de los k más cercanos. Utilizamos la función knn_graph que genera la matriz de adyacencia (o la representación de ésta) de estos puntos.

[4]:
```
def visualize_points(pos, edge_index=None, index=None):
    """Función para visualizar puntos con aristas relacionales"""
    fig = plt.figure(figsize=(4, 4))
    if edge_index is not None:
        for (src, dst) in edge_index.t().tolist():
            src = pos[src].tolist()
            dst = pos[dst].tolist()
```

```
            plt.plot([src[0], dst[0]], [src[1], dst[1]], linewidth=1,␣
↪color='black')
    if index is None:
        plt.scatter(pos[:, 0], pos[:, 1], s=50, zorder=1000)
    else:
        mask = torch.zeros(pos.size(0), dtype=torch.bool)
        mask[index] = True
        plt.scatter(pos[~mask, 0], pos[~mask, 1], s=50, color='lightgray',␣
↪zorder=1000)
        plt.scatter(pos[mask, 0], pos[mask, 1], s=50, zorder=1000)
    plt.axis('off')
    plt.show()
```

[5]:
```
visualize_points(Sphere, edge_index=knn_graph(Sphere, k=3))
visualize_points(Torus, edge_index=knn_graph(Torus, k=3))
```





1.2.1 Dataset de variedades (esferas y toros)

Finalmente, con esto establecido podemos generar un dataset que tenga varios ejemplos de variedades de esferas y toros con diferentes parámetros. Utilizamos estos datos para entrenar.

[6]:
```
#Dataset de entrenamiento
x = []
y = []
for i in range(1, 10):
    si = create_sphere(radius=i)
    edges_si = knn_graph(si, k=3)
    x.append((si.to(device), edges_si))
    y.append(0)
    ti = create_torus(r=i/10)
    edges_ti = knn_graph(si, k=3)
    x.append((ti, edges_ti))
    y.append(1)
```

1.3 Arquitectura de la red

Las GNNs pueden definirse como redes neuronales que tienen una capa de convolución sobre gráficas. De forma general, las capas graficas aprenden a representar los elementos de una estructura gráfica $G = (V, E)$; es decir, aprenden una representación de los vertices de entrada $u \in V$, que están asociados a un vector $x_u \in \mathbb{R}^d$. La forma general de las capas gráficas es:

$$h_u = \phi\Big(x_u, \bigoplus_{v \in \mathcal{N}_u} \psi(x_u, x_v)\Big)$$

Donde el operador $\bigoplus$ de agregación es un operador conmutativa (como la suma, max o min), la función ψ es la función de mensaje que determina cómo se transmite la información de los vecinos x_v hacia x_u. Y x_u es una función que regresa la representación a partir de x_u y $\bigoplus_{v \in \mathcal{N}_u} \psi(x_u, x_v)$.

Finalmente, es importante mencionar que la operación $\bigoplus$ se realiza sólo sobre los vecinos de u. Por tanto, este tipo de capas toman, además de los datos de entrada, la información de adyacencia a aprtir de una matriz de adyacencia A. Es decir, una GNN es una función $f(X, A)$ donde X es la matriz de puntos x_u que representan los nodos y A es la matriz de adyacencia.

Según el tipo de problemas y de datos que contemos, los elementos pueden estar determinadas de diferentes formas: por ejemplo, si tomamos el operador de agregación como la suma, $\psi(x_u, x_v) = w_{u,v} x_v$ y asumimos que los vecinos de un punto x_v esán definidos por una rejilla o grida (determinada por un kernel), entonces tenemos una capa de convolución sobre imagen.

1.3.1 Creación de capa de GNN

Dado que nuestro problema es determinar si el objeto definido por los puntos de entrada es una esfera o un toro a partir de las relaciones de vecinos más cercanos, definiremos la capa que queremos determinar de la siguiente forma:

$$h_i = \sum_{j \in \mathcal{N}_i} \phi(x_i, x_j) \cdot \psi(x_j)$$

donde tomamos a $\psi(x_j) = W x_j + b$, es decir una capa lineal. Mientras que la función ϕ la definimos de la siguiente forma:

$$\phi(x_i, x_j) = \sigma(w \cdot x_i + w' \cdot x_j + b)$$

tal que σ es la función sigmoide. Las adyacencias están determinadas por el método devecinos más cercanos. Para definir esta capa usamos pytorch geometric. Como definimos la capa dentro de una clase que hereda propiedades de la clase de capas de GNN, la forma en realizar la agregación la indicamos como:

```
super(...,self).__init__(aggr='add')
```

Asimismo, usamos el método de message para hacer la agregación a partir de las funciones determinadas. El resultado de esta capa serán nuevas representaciones de los puntos h_i que contiene la información de las relaciones en la estructura gráfica.

[7]:
```
class EdgeWeight(geo_nn.MessagePassing):
    """Capa de GNN"""
    def __init__(self, in_channels, out_channels):
        #Indicamos el operador de agregación
        super(EdgeWeight, self).__init__(aggr='add')
        #Tipos de capas
        self.psi = nn.Linear(in_channels, out_channels)
        self.dot = nn.Sequential(nn.Linear(2*in_channels, 1), nn.Sigmoid())

    def forward(self, x, edge_index):
        #Envía los vecinos con función psi
```

```
        return self.propagate(edge_index, x=x)

    def message(self, x_i, x_j):
        #Calcula la función phi
        phi = self.dot(torch.cat([x_i, x_j], dim=1))
        self.phi = phi
        self.h = phi*self.psi(x_j)

        return self.h
```

1.3.2 Construcción de la red

Una vez que hemos definido la capa de GNN, podemo definir las red neuronal completa que tome la nube de puntos, sus adyacencias y nos devuelva la clasificación del objeto de entrada. Nuestra red estará definida por las siguientes capas:

1. Capa de GNN: Que se ha definido anteriormente.
2. Max pooling: Para obtener un vector que entre a una Feedforward, primer aplicamos un pooling (similar a los utilizados en imágenes).
3. Flattening: Una vez aplicado el max pooling, aplicamos una capa Feedforward con activación tanh y finalmente la capa de salida con activación Softmax para obtener las probabilidades de las clases.

[8]:
```
class GNNModel(nn.Module):
    """Red neuronal con capa GNN"""
    def __init__(self, in_channels=3, hidden_dims=10, outputs=2):
        super(GNNModel, self).__init__()
        #Capa GNN
        self.gnn_conv = EdgeWeight(in_channels, hidden_dims)
        #Max pooling
        self.pooling = nn.MaxPool1d(3,stride=3)
        #Feedforward
        self.output = nn.Sequential(nn.Linear(1660, 2*hidden_dims), nn.Tanh(),
                                    nn.Linear(2*hidden_dims, outputs), nn.
↪Softmax(dim=0))

    def forward(self, x):
        #Paso forward
        x, edge_index = x
        h = self.gnn_conv(x, edge_index)
        self.h_pool = self.pooling(h.T).T.flatten()
        out = self.output(self.h_pool)

        return out
```

1.3.3 Entrenamiento del modelo

Finalmente, podemos entrenar el modelo. Para esto, lo definimos a partir del modelo establecido, con los hiperparámetros necesarios.

[9]:
```
#Definimos el modelo
model = GNNModel(in_channels=3, hidden_dims=10, outputs=2).to(device)
model.train()
```

[9]:
```
GNNModel(
  (gnn_conv): EdgeWeight()
  (pooling): MaxPool1d(kernel_size=3, stride=3, padding=0, dilation=1,
ceil_mode=False)
  (output): Sequential(
    (0): Linear(in_features=1660, out_features=20, bias=True)
    (1): Tanh()
    (2): Linear(in_features=20, out_features=2, bias=True)
    (3): Softmax(dim=0)
  )
)
```

Ya obtenido el modelo, podemos entrenarlo. Ya que se trata de un problema de clasificación, utilizamos como función de riesgo la entropía cruzada. Utilizamos el optimizador Adam.

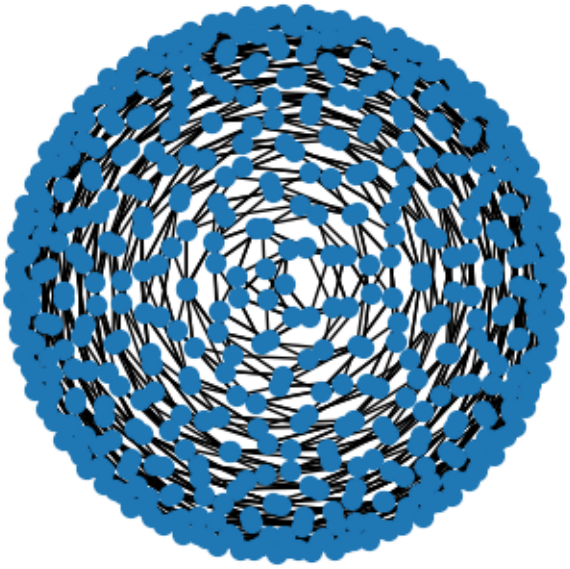
[10]:
```
#Función de riesgo
criterion = torch.nn.CrossEntropyLoss()
#Optimizados
optimizer = torch.optim.Adam(model.parameters(), betas=(0.9,0.98), lr=0.001)
#Núm. épocas
epochs = 100
for t in tqdm(range(epochs)):
    for x_i, y_i in zip(x,y):
        y_pred= model(x_i)
        optimizer.zero_grad()
        y_i = torch.tensor(y_i)
        loss = criterion(y_pred, y_i)
        loss.backward()
        optimizer.step()
```

```
100%|                    | 100/100 [00:08<00:00, 12.08it/s]
```

1.3.4 Evaluación

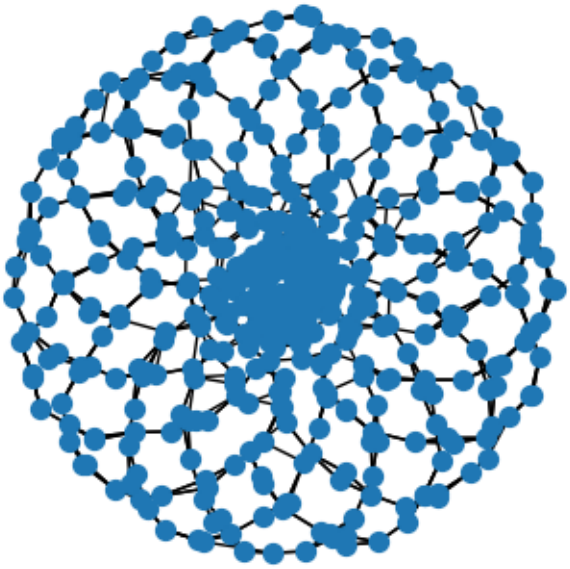
Finalmente, podemos ver cómo funciona el clasificador que hemos construido con GNNs. En primer lugar, definimos dos superficies, una esfera y un toro con parámetros que no han sido vistos en el entrenamiento y agrgegamos una traslación y ruido para determinar cómo son clasificados por le modelo:

[11]:
```
model.eval()
#Crea esfera con traslación
sphere_eval = create_sphere(radius=100, dist=10, noise=0.1) + 100
edges_sphere = knn_graph(sphere_eval, k=3)
visualize_points(sphere_eval, edges_sphere)
#Clasifica la superficie
pred = model((sphere_eval, edges_sphere)).argmax(axis=0)
print('Clase {}'.format(pred.detach()))
```



```
Clase 0
```

[12]:
```
#Crea el toro con traslación
torus_eval = create_torus(R=4, r=3.5, noise=0.1) + 100
edges_torus = knn_graph(torus_eval, k=3)
visualize_points(torus_eval, edges_torus)
#Clasifica la superficie
pred = model((torus_eval, edges_torus)).argmax(axis=0)
print('Clase {}'.format(pred.detach()))
```



Clase 1

Ahora creamos un dataset de evaluación con parámetros distintos al entrenamiento, y predecimos las clases por medio del modelo entrenado. Obtenemos el reporte de clasificación, que como se puede observar, tiene valores buenos.

[13]:
```
#Tamaño del test set
eval_size = 100
#Genera esferas y toros
x_eval = [create_sphere(radius=k[0],dist=k[1],noise=0.1) + k[2]
          for k in np.random.
 ↪randint(1,30,size=(eval_size,3))]+[create_torus(R=k[0],r=k[1],noise=0.1) +␣
 ↪k[2]
                                                                  for k in np.
 ↪random.randint(1,30,size=(eval_size,3))]
#Clases reales de los datos
y_eval = [0 for k in range(eval_size)] + [1 for k in range(eval_size)]

#Predicción hecha por el clasificador
y_pred = [model((x, knn_graph(x, k=3))).argmax().detach() for x in x_eval]
#Informe de clasificación
print(classification_report(y_eval, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99       100
           1       0.98      1.00      0.99       100
```

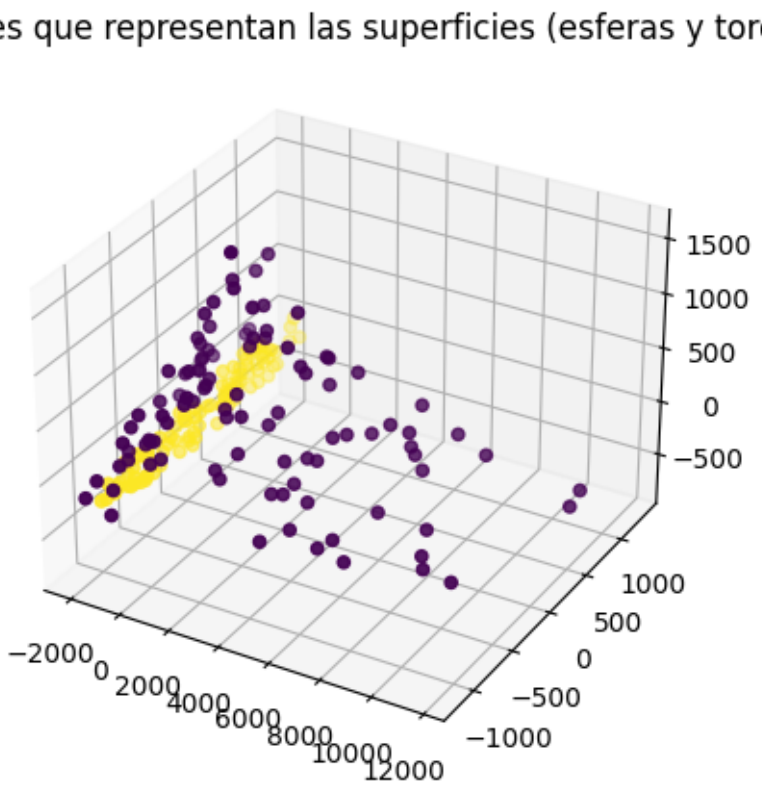
```
    accuracy                           0.99       200
   macro avg       0.99      0.99      0.99       200
weighted avg       0.99      0.99      0.99       200
```

Visualización de las representaciones de los datos El modelo que hemos definido toma un conjunto de puntos y sus adyacencias y genera un vector (después de pasar por la capa de GNN, max pooling y flattening) que representa a la superficie completa. Podemos visualizar justamente qué tipo de distribución aprende la red neuronal para poder clasificarlos. Como se puede ver, la red aprende una representación que hace los datos linealmente separables.

[14]:
```python
#Representaciones de los datos
vectors = []
for x_i in x_eval:
    model((x_i, knn_graph(x_i, k=3)))
    vectors.append(model.h_pool.detach().numpy().reshape(1660))
vectors = np.array(vectors)

#Reducción de dimensionalidad
pca = PCA(3)
X = pca.fit_transform(vectors)

#Visualización
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X[:,0], X[:,1], X[:,2], c=y_eval)
plt.title('Vectores que representan las superficies (esferas y toros)')
plt.show()
```



1.4 Ejercicio

Adaptar el ejemplo anterior pero utilizando una capa gráfica con convolución de gráfica; es decir, en donde se implemente la función de agregación:

$$x_i = \max_{j \in \mathcal{N}} \phi(x_i, x_j - x_i)$$

Donde

$$\phi(x_i, x_j - x_i) = W'' \cdot ReLU\Big(W \cdot x_i + W' \cdot (x_j - x_i) + b\Big) + b'$$

Recordar que aquí se debe señalar que se usara un método de agregación por maximización, por lo que se debe sustituir 'add' por 'max' en la clase de la capa gráfica:

```
super(EdgeWeight, self).__init__(aggr='max')
```