Universidad de la República

Facultad de Ingeniería

Aprendizaje Automático para Datos en Grafos

Laboratorio 4 | Introducción a Graph Neural Networks

Autores:
Federico Bello
Gonzalo Chiarlone

28 de septiembre de 2025

UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

1.

Dado un grafo con n nodos se define la convolución a nivel de grafo (*graph convolution*) como:

$$\mathbf{Y} = \sum_{k=0}^{K} \mathbf{S}^k \mathbf{X} \mathbf{H}_k,$$

donde $\mathbf{S}$ es un *graph shift operator* (GSO), $\mathbf{X} \in \mathbb{R}^{n \times F_{in}}$ es la matriz con la señal de entrada y $\mathbf{H}_k \in \mathbb{R}^{F_{in} \times F_{out}}$ la matriz con los coeficientes del filtro. Por lo tanto, la señal de salida $\mathbf{Y}$ tiene tamaño $n \times F_{out}$. Recordando que el GSO $\mathbf{S}^k_{i,j}$ tiene la propiedad de ser distinto a cero sii existe un camino de largo k entre los nodos i, j se tiene que el orden del filtro K indica a que tantos nodos de distancia se toma en consideración la información. Por otro lado, los coeficientes $\mathbf{H}_k$ determinan cuanto peso se le da a los nodos a distancia k.

A continuación se ve la implementación de la convolución, donde el método *propagate* implementa el GSO, es decir, para cada nodo calcula la suma de las señales de sus vecinos.

```
def forward(self, x, edge_index):
        out = self.filters[0](x)
        for filter in self.filters[1:]:
            x = self.propagate(edge_index, x=x, edge_weight=None, size=None)
            out += filter.forward(x)
        return out
```

2.

Para verificar que la parte anterior es correcta, se aplica una convolución usando el modelo de la parte anterior y utilizando solamente Pytorch. Como se puede verificar en el *notebook*, los resultados son los mismos en ambos casos.

3.

Ahora se volverá a verificar el resultado utilizando la clase *TAGConv* de *PyTorch Geometric*. Para esto, es importante igualar los pesos de ambos filtros.

4.

En el *notebook* se encuentra el código para construir el *Graph Perceptron* pedido.

5.

Volviendo a la definición de la convolución se tiene que los únicos parámetros a aprender son los de las matrices de filtros $\mathbf{H}$, donde hay $K + 1$ matrices de filtros las cuales son de dimensión $F_{in} \times F_{out}$. Por lo tanto, hay un total de $(K+1) \times F_{in} \times F_{out}$ parámetros a aprender (mas los términos de bias y la capa lineal), es decir, por cada *feature* que se desee agregar a la señal de entrada la cantidad de parámetros a aprender aumenta en $(K + 1) \times F_{out}$

6.

Por el mismo razonamiento de la sección anterior, se tiene que la cantidad de parámetros no varia al variar la cantidad de nodos del grafo. Esta es una gran ventaja de de las GNNs, ya que implica que las mismas tienen la capacidad de generalizar para distintos tamaños de grafos. Es decir, se pueden adaptar a tamaños de grafos nunca antes vistos, pudiendo entrenar en grafos mas pequeños con el objetivo de luego generalizar para grafos grandes. De esta forma, se disminuye considerablemente los recursos necesarios para el entrenamiento.

7.

El objetivo de estas siguientes secciones es utilizar el *dataset ogbn-arxiv* de *Open Graph Benchmark* (OGB). El mismo consiste de 169343 *papers* representados con 128 *features* cada uno. El objetivo es predecir la categoría de los *papers* dentro de un conjunto de 40 categorías posibles.

8.

En la figura 1 se ve un modelo de *graph convolution network* (GCN) el cual consiste de una determinada cantidad de capas compuestas cada una por una GCN, una normalización, una función de activación (en este caso una ReLu) y un *Dropout* el cual tiene el objetivo de regularizar. Por lo tanto, para implementar este modelo alcanza con apilar las capas con este formato y luego agregar un *LogSoftmax* antes de la salida. El único detalle a tener en cuenta es que la primera y la ultima capa podrían tener dimensiones de entradas distinto al resto, ya que la primera tiene como entrada un vector de tamaño F_{in} y salida F_{hidden} y la ultima F_{hidden} y salida F_{out}. Todas las capas ocultas tienen tanto la entrada como la salida de tamaño F_{hidden}.

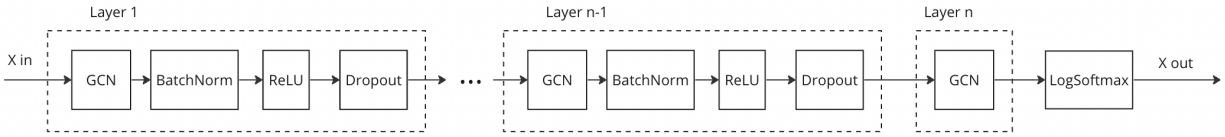

Figura 1: Diagrama de bloques del modelo GCN

9.

El bloque de código para entrenar el modelo se encuentra en el *notebook*, donde se llega a una *accuracy* de 72 % en conjunto de entrenamiento y 70 % en conjunto de validación. Por otra parte, la *accuracy* en *test* es del 69 % Los resultados fueron obtenidos entrenando durante 100 epocas un modelo con un filtro de orden 1, dimensiones ocultas de tamaño 128, 2 capas y un valor de *dropout* de 0.5.

10.

Con el objetivo de mejorar el resultado se experimento con distintos valores del orden del filtro y la cantidad total de capas. En las tablas 1 y 2 se ve la máxima *accuracy* obtenida luego de entrenar el modelo durante 200 épocas (desempeño en entrenamiento y validación respectivamente). Se ve como el mejor resultado (en validación) se obtiene para un orden de filtro de 5 y solamente dos capas.

num_layers \ K	1	5	10	20
2	74,36 %	**74.35 %**	73,22 %	70,65 %
5	73,70 %	68,32 %	50,62 %	34,94 %
10	62,92 %	48,83 %	9,75 %	17,07 %

Cuadro 1: *Accuracy* en conjunto de entrenamiento para distintos valores de los parámetros

num_layers \ K	1	5	10	20
2	71,11 %	**73.17 %**	71,38 %	70,40 %
5	72,01 %	65,26 %	48,84 %	20,10 %
10	60,76 %	46,83 %	11,81 %	15,37 %

Cuadro 2: *Accuracy* en conjunto de validación para distintos valores de los parámetros

Se ve como al aumentar el orden del filtro o la cantidad de capas el resultado comienza a empeorar. Sin embargo, es posible que esto se deba a la baja cantidad de épocas. Observar como ninguno de los modelos se encuentra sobreajustado al conjunto de entrenamiento. Esto puede indicar que el modelo no esta logrando ser suficientemente expresivo o que la cantidad de épocas no están siendo suficientes. Debido que la cantidad de parámetros aumenta junto con el orden del filtro y la cantidad de capas es probable que el problema este en el segundo punto, dado que debería estar aumentando su expresividad y no disminuyendo. A efectos de verificar esta hipótesis se entreno el modelo con $K = 10$ y $num\_layers = 5$ durante 2000 épocas. De esta forma el resultado paso de ser el de la tabla a un 74 % en entrenamiento y un 71 % en validación. Esto indica que con paciencia una mayor cantidad de parámetros se podría mejorar aun mas el resultado.

Finalmente, para los valores de $K = 5$ y $num\_layers = 2$ se obtuvo una *accuracy* en el conjunto de *test* de 72 %, mejorando así en un 3 % el resultado inicial.

11.

A continuación se implementara una versión de *graph attention*. El objetivo de estos modelos es, a diferencia de las GNNs, brindarle mas peso a los vecinos que tengan información mas útil. Estos pesos se calculan mediante una función de *scoring*, la cual se calcula como: $e(\mathbf{x}_i, \mathbf{x}_j) = \text{LeakyReLU}(a^T[\mathbf{H}\mathbf{x}_i || \mathbf{H}\mathbf{x}_j])$ donde $\mathbf{x}_i$, $\mathbf{x}_j$ son las señales en el nodo i y j respectivamente y $\mathbf{a}$ es el nuevo parámetro a aprender. Con esta función de *scoring*, los parámetros del *attention* (cuanto peso se le da a cada nodo) se calcula como: $\alpha_{ij} = \text{softmax}(e(\mathbf{x}_i, \mathbf{x}_j))$.

El bloque de código que implementa el *forward* de este modelo se encuentra en el *notebook*.

12.

A su vez, la prueba de dicho *forward* también se encuentra en el *notebook*.

13.

Para esta sección se comparo el resultado de la implementación anterior con la implementación de *GATConv* de *PyTorch*. Al igual que en la sección 3, es importante fijar todos los parametros del modelo con iguales valores. De esta forma, se obtienen resultados muy similares entre el *forward* implementado en el *notebook* y el implementado por *PyTorch*.

14.

Al implementar el modelo de la figura 1 pero con *Graph Attention* en lugar de *Graph Convolution* y entrenar con los mismos parámetros que en la sección 9, se obtiene una *accuracy* en validación cercana a 70 %, menor a la obtenida previamente con los mismos parámetros. Sin embargo, el modelo comienza a sobreajustarse levemente, indicando que la expresividad del modelo va en aumento y que con las debidas técnicas de regularización el resultado debería mejorar.

15.

Al utilizar la clase *GATv2Conv* en lugar de *GATConv* el resultado mejora. Esta implementación de *Graph Attention* se basa en [1], donde afirman que la implementación anterior de tiene una limitante importante: solamente implementa atención *estatica*. Esto quiere decir que el orden de los valores de atención es igual en todos los todos los nodos, no esta condicionado a cual es el nodo que se este consultando. La *GATv2Conv* rompe con esta limitante el agregar una no linealidad (*LeakyReLu)* antes de multiplicar por el vector $\mathbf{a}$. De esta forma, la función de *scoring* pasa a ser: $e(\mathbf{x}_i, \mathbf{x}_j) = a^T \text{LeakyReLU}([\mathbf{H}\mathbf{x}_i || \mathbf{H}\mathbf{x}_j])$. De esta forma, el resultado mejora cerca de un 2 % en el conjunto de *validación* con los parámetros de la sección 9. Mas allá del resultado en validación, el modelo incrementa considerablemente el desempeño en entrenamiento, viendo claramente como la no linealidad incrementa la expresividad del modelo.

Referencias

[1] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks?, 2021.

16. Apendice

Link al *notebook*: `https://drive.google.com/file/d/1lLFBg3glS3VsFRkkl7TbXlodnp5ynqdB/view?usp=drive_link`