Tratamiento Estadístico Computacional de la Información

UNIVERSIDAD
COMPLUTENSE
MADRID

POLITÉCNICA

Clasificación de Proteínas con Redes Neuronales de Grafos

Realizado por:

Javier Castellano Soria

Ignacio Fernández Sánchez-Pascuala

1 de marzo de 2024

Índice

Resumen

El trabajo se centra en el uso de redes neuronales de grafos (*Graph Neural Networks*) para la clasificación de proteínas en enzimas y no enzimas. Se introducen conceptos clave de las *GNN*, detallando su estructura paso por paso y destacando su utilidad y aplicaciones en diversas áreas.

Se comparan tres modelos implementados, indicando sus arquitecturas y técnicas utilizadas, y entrenados con el objetivo de obtener el mejor rendimiento posible en el conjunto de datos *PROTEINS*, para la clasificación de proteínas. Los resultados muestran que hay modelos que alcanzan un 72.5 % de precisión en el conjunto de validación, destacando la eficacia de las *GNN*.

1. Introducción

Las redes neuronales de grafos, también conocidas como *Graph Neural Networks* (*GNN*) son una extensión natural de las redes neuronales convencionales, adaptadas para trabajar con datos en forma de grafo. Este enfoque surge debido a la necesidad de desarrollar modelos de aprendizaje automático capaces de capturar e interpretar la estructura de grafo, ya que en muchas ocasiones estos ofrecen una representación más rica y detallada que otros modelos de datos lineales o tabulares.

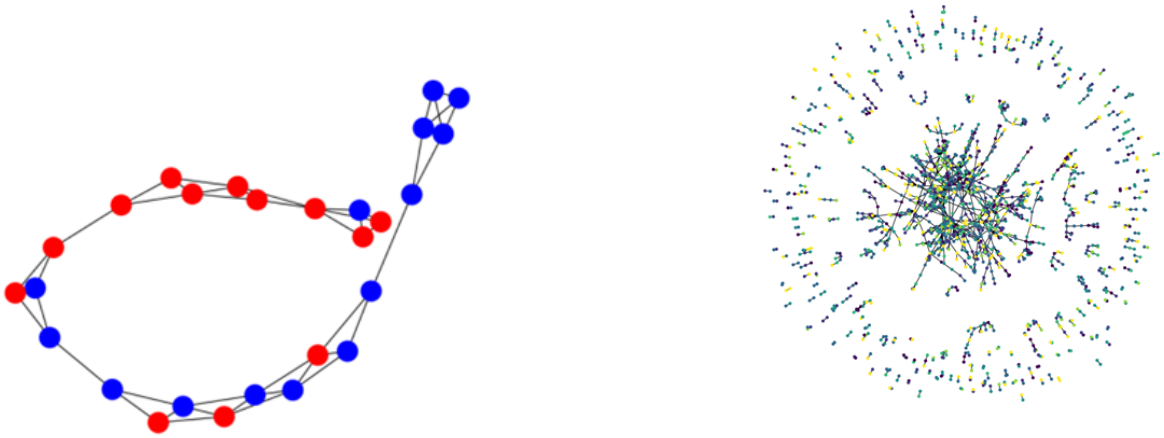
Las *GNN* surgieron en 2009 cuando investigadores italianos acuñaron el término, pero fue en 2017 cuando la variante *Graph Convolutional Network* (*GCN*) demostró su potencial. Esta innovación inspiró a Jure Leskovec y su equipo en Stanford a desarrollar *GraphSage*, aplicada exitosamente en Pinterest como *PinSage*, un sistema de recomendación líder. Actualmente, han surgido variantes como *Graph Recurrent Networks* (*GRN*) y *Graph Attention Networks* (*GAT*), que incorporan el mecanismo de atención de los modelos *Transformer*, ampliando las aplicaciones y mejorando su capacidad en contextos específicos.

1.1. Grafos

Los grafos $G = (V, E)$ son representaciones versátiles que permiten modelar y analizar relaciones entre entidades. Un grafo se compone de nodos $V = \{v_1, \ldots, v_n\}$, que representan entidades, y aristas $E \subseteq V \times V$, que denotan relaciones entre estas entidades, pudiendo ser dirigidas o no dirigidas. Dos nodos se consideran adyacentes si están conectados por una arista.

Esta estructura de datos es fundamental para describir interacciones en diversas áreas, como redes sociales, biología molecular, recomendación de productos, entre otros. Por ejemplo, en redes sociales, los nodos pueden representar usuarios, mientras que los enlaces indican amistades o interacciones. En biología molecular, los nodos podrían corresponder a átomos, y las aristas representarían enlaces químicos.

En la Figura 1a se puede observar el grafo asociado a la representación de una proteína. Por otro lado, en la Figura 1b se muestra el grafo del conjunto de datos *CORA*, donde los nodos representan documentos académicos y las aristas indican citaciones entre ellos. Cada documento está asociado con un conjunto de características, como términos clave y contenido del texto. Este conjunto de datos es comúnmente utilizado en la investigación de redes neuronales de grafos.



(a) Grafo asociado a una proteína

(b) Grafo del conjunto de datos CORA

Figura 1: Descripción general de dos grafos

1.2. Ventajas respecto a otros modelos

Un desafío fundamental en la aplicación de modelos de aprendizaje automático es la variabilidad en las dimensiones de entrada de los datos. En el contexto de los grafos, esta variabilidad se presenta debido a las diferencias en la cantidad de nodos y aristas entre distintos grafos. Las redes neuronales de grafos abordan este problema de dimensión variable de manera efectiva, ya que están diseñadas para manejar estructuras de datos flexibles y permiten adaptarse a la topología específica de cada grafo.

Otra de las ventajas de las redes neuronales de grafos es su capacidad para realizar clasificaciones tanto a nivel de grafo, de nodo y de arista:

- **Clasificación a Nivel de Grafo:** Las *GNN* pueden clasificar grafos enteros, por ejemplo, para predecir propiedades globales de una red social o determinar la función general de una proteína.
- **Clasificación a Nivel de Nodo:** Permiten asignar etiquetas o categorías a cada nodo individual, facilitando tareas como la categorización de documentos en un conjunto de datos académicos como *CORA*, donde se puede clasificar cada artículo según su tema o área de estudio.
- **Clasificación a Nivel de Arista:** Las *GNN* pueden predecir la naturaleza de las relaciones entre los nodos. Por ejemplo, en una red social, podrían detectar el tipo de relación entre

dos personas, como amistad o colaboración profesional. En biología molecular, podrían identificar el tipo de interacción química entre moléculas.

1.3. Usos en la actualidad

Dentro del ámbito de pronóstico meteorológico, uno de los modelos destacados es *GraphCast* [1]. Es un método basado en redes neuronales de grafos para la predicción del tiempo a nivel mundial hasta 10 días de antelación. Este modelo se entrena directamente con información histórica meteorológica. La capacidad de *GraphCast* para predecir cientos de variables meteorológicas en múltiples escalas y con gran velocidad lo coloca como un sistema destacado en el pronóstico del tiempo. De hecho, ha demostrado superar significativamente a los sistemas deterministas operativos más precisos en el 90 % de 1380 objetivos de verificación, incluyendo ciclones tropicales, ríos atmosféricos y temperaturas extremas.

En otras aplicaciones destacadas de las *GNN*, empresas líderes han adoptado esta tecnología. Amazon, por ejemplo, en 2017 anunció el uso de *GNN* para la detección de fraudes, y posteriormente, en 2020, implementó un servicio público de *GNN* que permite a otros utilizarlo para la detección de fraudes, sistemas de recomendación y diversas aplicaciones.

En el ámbito de la biotecnología y la farmacéutica, biopharma *GSK* mantiene un gráfico de conocimiento con casi 500,000 millones de nodos, empleado en muchos de sus modelos de lenguaje de máquina.

LinkedIn, una plataforma social profesional, utiliza *GNN* para proporcionar recomendaciones en redes sociales y comprender las relaciones entre las habilidades de las personas y sus puestos de trabajo.

2. Marco Teórico

Formalmente, las *GNN* se definen como un conjunto de modelos matemáticos que se caracterizan por procesar la información de los nodos y combinarla con la de sus adyacentes a medida que se avanzan en las capas. La forma en que se procesa la información y se combina, determinará la arquitectura del modelo.

A continuación, se detallan algunos de sus aspectos básicos, como la entrada del modelo, el procesamiento de información en una capa, diversas técnicas de *Pooling* y un ejemplo sencillo.

2.1. Entrada del modelo

La entrada del modelo será un grafo representado por un conjunto de nodos V, donde cada nodo $v \in V$ es un vector real de dimensión n. Además, se añade la matriz de adyacencia A.

La matriz de adyacencia A es una matriz cuadrada de tamaño $|V| \times |V|$, donde $|V|$ es el número de nodos en el grafo. Cada entrada a_{ij} de la matriz indica la existencia de una arista entre los nodos v_i y v_j. Si $a_{ij} = 1$, hay una arista entre v_i y v_j; de lo contrario, si $a_{ij} = 0$, no hay arista.

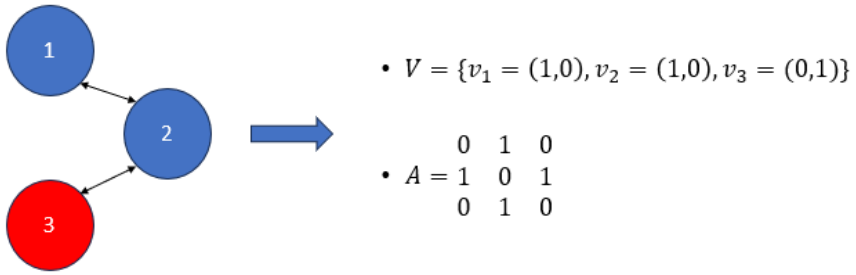


Figura 2: Codificación de un grafo

En la Figura 2 se muestra la codificación de un grafo en su conjunto de nodos V y su matriz de adyacencia A.

2.2. Capa Sencilla

Una capa sencilla tomará como entrada el grafo y devolverá otro grafo con la misma topología, excepto los vectores asociados a los nodos, que se habrán transformado. En la Figura 3 se puede ver cómo funciona el procesamiento en una capa sencilla.

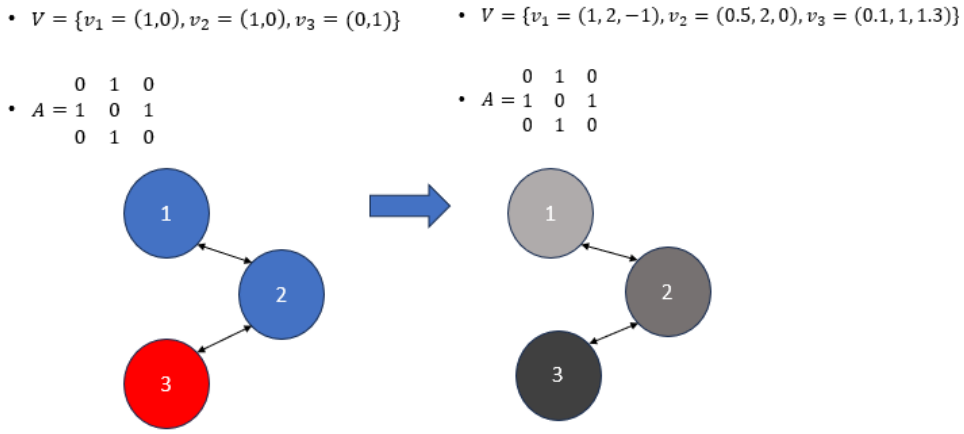


Figura 3: Procesamiento de una capa

Los valores asociados a los nodos se transforman de la siguiente manera:

1. Se aplica una red neuronal *MLP* (*Multilayer Perceptron*) de una capa, $f : \mathbb{R}^n \to \mathbb{R}^m$, a cada uno de los vectores de los nodos. Esta red neuronal compartirá parámetros para todos los nodos del grafo.

2. El valor del nodo en el nuevo grafo procesado será la suma de su nuevo vector, obtenido aplicando f, sumado con los vectores procesados por f de sus nodos adyacentes según la matriz de adyacencia A, utilizando la fórmula:

$$v_i^* = \sum_{v \in Adj(v_i) \cup \{v_i\}} f(v)$$

donde $Adj(v_i)$ representa el conjunto de nodos adyacentes al nodo v_i. Esta técnica es conocida como *message passing*.

Esta transformación mediante una red neuronal y el paso de mensajes proporciona una representación mejorada de los nodos en el grafo, capturando información local para cada nodo.

Este tipo de capa de red neuronal de grafo fue presentado por Kipf, T. N., & Welling, M (2016) en la publicación "Semi-supervised classification with graph convolutional networks" [2].

Existen variaciones en la forma de procesar la información en esta capa, considerando aspectos como los tipos de aristas, si los hubiera, en la transformación. Algunas implementaciones utilizan las *Graph Conv Layers* [3], que involucran dos redes neuronales de una capa, f y f', aplicadas a cada vector de los nodos. La fórmula de procesamiento en este caso es:

$$v_i^* = f(v_i) + \sum_{v \in \text{Adj}(v_i)} f'(v)$$

Además, se han propuesto enfoques más sofisticados, como el uso de *Graph Attention Networks (GAT)* [4]. Las *GAT* están basadas en mecanismos de atención inspirados en las estructuras de las Redes Neuronales *Transformer*. También se han explorado generalizaciones de las operaciones de agregación (suma de los vectores procesados de nodos adyacentes) [5], ampliando la capacidad de procesamiento de información en las capas.

Cabe destacar que la descripción anterior se centra en cómo se procesa la información en una capa de una *GNN*. En la práctica, estos procesamientos se pueden apilar, utilizando tantas capas como sea necesario para capturar de manera efectiva la complejidad y las relaciones presentes en los datos del grafo.

2.3. Técnicas de *Pooling*

Después de procesar los nodos con las capas anteriores, se aplican técnicas de *Pooling* en el grafo para resumir la información en sus nodos. El objetivo es transformar todos los datos a un espacio de igual dimensión, independientemente de la entrada del modelo, para poder realizar la clasificación a partir de este nuevo vector.

Formalmente, las técnicas de *Pooling* consisten en aplicar una función, $p : \mathbb{R}^{m \times |V|} \to \mathbb{R}^m$, al conjunto de todos los nodos tal que nos devuelva un vector que resuma la información del grafo.

Este vector podría ser obtenido mediante diversas estrategias, como la media de los nodos en el grafo:

$$w = \frac{1}{|V|} \sum_{v \in V} v$$

También se pueden utilizar otras operaciones como el máximo o incluso la concatenación de diferentes métricas sobre los vectores de cada nodo.

Finalmente, se aplica una capa de perceptrón multicapa (*MLP*) a este vector, donde la dimensión de salida corresponde al número de clases en el problema de clasificación. En la Figura 4 se puede ver la representación del procesamiento entero de una *GNN* sencilla, desde la entrada de los datos, hasta su clasificación.

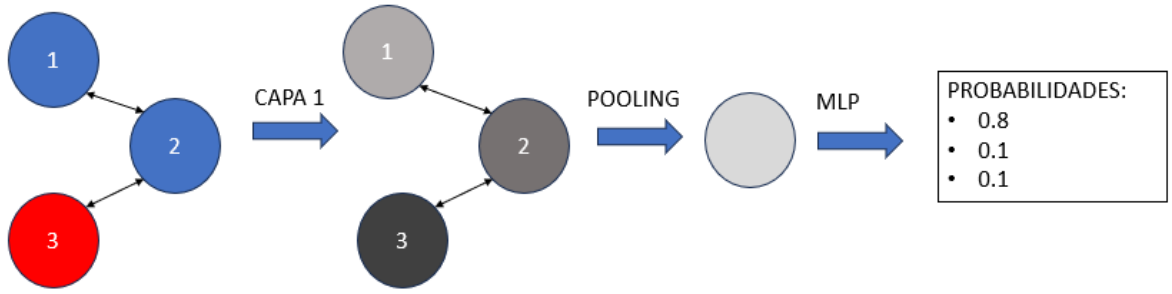


Figura 4: Ejemplo de una GNN sencilla

A veces, es necesario reducir gradualmente la cantidad de nodos en el proceso. Para ello, se recurre a métodos de *TopK pooling*, como se describe en los trabajos de Gao y Ji [6] y Zhang et al. [7]. Esta técnica asigna un puntaje aprendible a cada nodo, seleccionando únicamente los k nodos con las puntuaciones más altas. Este puntaje es el resultado del producto escalar entre cada nodo y un vector de parámetros w unitario.

Es esencial destacar que la técnica de *TopK pooling*, se combina con la capa sencilla desarrollada en la anterior sección, siguiendo un patrón de Capa sencilla → *TopK pooling* → Capa sencilla → *TopK pooling*, y así sucesivamente hasta obtener un vector que resuma toda la información del grafo. La reducción progresiva del número de nodos puede ser beneficiosa, especialmente cuando la cantidad de nodos es considerable, ya que, por ejemplo, algunas métricas como la media podría perder la varianza de la información del grafo.

3. Implementación práctica

En esta sección, se llevará a cabo la implementación práctica de la teoría presentada anteriormente. En nuestro caso, nos hemos centrado en la clasificación de proteínas, una aplicación clave en la biología molecular. El objetivo será distinguir entre proteínas enzimas y no enzimas,

lo cual es esencial para comprender las funciones y características específicas de estas biomoléculas. Para lograr esto, se detallarán diferentes aspectos de la implementación, comenzando con la introducción del conjunto de datos y los modelos utilizados.

3.1. Conjunto de datos

El conjunto de datos utilizado ha sido *PROTEINS*. Se compone de 1113 proteínas, cada una etiquetada según su clasificación como enzima o no enzima. Cada proteína es presentada como un grafo, donde los nodos corresponden a los aminoácidos y están codificados utilizando la técnica *one-hot encoding*. La conexión entre dos nodos se establece si los aminoácidos respectivos se encuentran a una distancia menor de 6 ángstroms (equivalente a 6×10^{-10} metros).

En la Figura 5 se presentan dos ejemplos gráficos que ilustran cómo se transforman las proteínas en grafos.

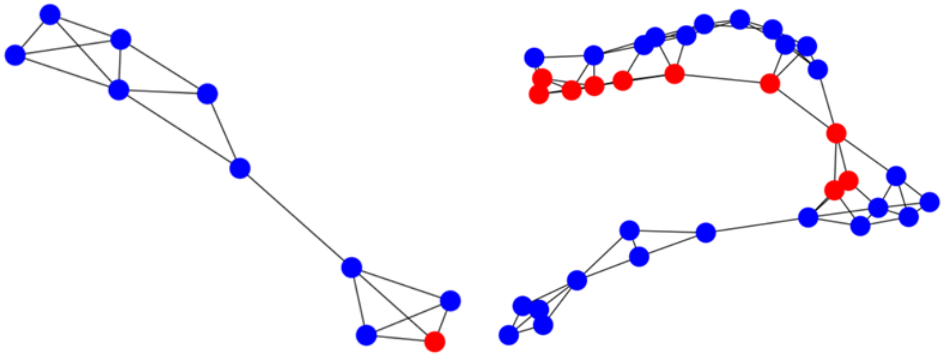


Figura 5: Dos ejemplos de proteínas

3.2. Modelos utilizados

Se presentan tres modelos diferentes que han sido evaluados para la clasificación de proteínas en enzimas y no enzimas. Cada modelo tiene sus propias características y métodos de procesamiento de grafos. Los códigos utilizados para implementar los modelos se encuentran en el Anexo A, realizados con la librería *PyTorch*.

3.2.1. Modelo 1

El Modelo 1 utiliza tres capas sencillas [2] para procesar la información del grafo. Luego, se usa la media (*Global Average Pooling*) y el máximo (*Global Max Pooling*) como técnicas de *Pooling* para obtener representaciones globales del grafo.

Posteriormente, estas medidas se concatenan y se pasan a través de tres capas densas de una Red Neuronal Perceptrón Multicapa (*MLP*) con funciones de activación *ReLu* en cada capa.

En la capa de salida se usa una función de activación *log-softmax*, que emite un valor único que representa la log-probabilidad de que sea enzima o no enzima la proteína.

3.2.2. Modelo 2

El Modelo 2 consta de tres capas de *Graph Conv Layers* [3] intercaladas con dos capas de *TopK Pooling* intermedias, almacenando la media y el máximo depués de cada capa de *Pooling*. Posteriormente, se emplea una *MLP* con tres capas densas que toma como entrada los valores de la media y el máximo almacenados. Este esquema es usado por muchos autores en la literatura para tratar este tipo de problemas.

3.2.3. Modelo 3

El Modelo 3 consta también de tres capas de *Graph Conv Layers* sin utilizar *Pooling*, seguido de una *MLP* de tres capas densas que toma como entrada un nodo global artificial. Este nodo global se inicializa como la media de las características de los nodos originales en cada grafo, y es adyacente a todos los demás. Con esto se busca que este nodo resuma toda la información del grafo con el paso de las capas.

En la Figura 6 se puede observar la representación de una proteína con la incorporación de un nodo global.

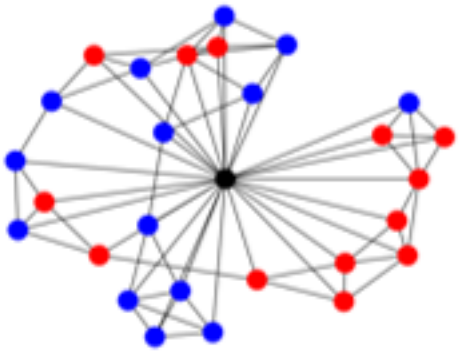


Figura 6: Ejemplo proteína con nodo global

3.3. Entrenamiento

Durante el proceso de entrenamiento, se aplicaron diversas técnicas y configuraciones para optimizar el rendimiento de los modelos.

- **Partición de Conjunto de Datos:** Se dividió el conjunto de datos en un 70 % para entrenamiento, 15 % para validación y otro 15 % para pruebas.
- **Algoritmo Aprendizaje:** Para la optimización de los modelos se empleó el algoritmo Adam. Adam es un algoritmo de optimización que adapta dinámicamente la tasa de aprendizaje de cada parámetro durante el entrenamiento. La adaptación se basa en la combinación de información de primer y segundo orden. La información de primer orden

se refiere a los gradientes de la función de pérdida, mientras que la información de segundo orden implica información sobre cómo cambian estos gradientes en cada iteración. La combinación de ambas proporciona una tasa de aprendizaje adaptativa que facilita la convergencia eficiente en problemas de optimización no convexos y complejos.

La actualización de los parámetros en el algoritmo Adam se realiza utilizando las siguientes fórmulas:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla J(\theta_{t-1})$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot \nabla J(\theta_{t-1}) \times \nabla J(\theta_{t-1})$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_t = \theta_{t-1} + \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

donde m_t y v_t son estimaciones del primer y segundo momento no centrado de los gradientes, J la función de perdida, β_1 y β_2 son hiperparámetros de suavizado, t indica el número de iteraciones, $\hat{m}_t$ y $\hat{v}_t$ son las versiones sesgadas de m_t y v_t, θ_t son los parámetros a actualizar, η es la tasa de aprendizaje, y ϵ es un término pequeño para evitar la división por cero. El símbolo "×"denota la multiplicación elemento a lemento.

- **Early Stopping:** Se implementó la técnica de *Early Stopping* con una paciencia de 30 épocas. Esta estrategia detiene el entrenamiento si el rendimiento del modelo en el conjunto de validación no mejora después de un cierto número de épocas, evitando el sobreajuste.

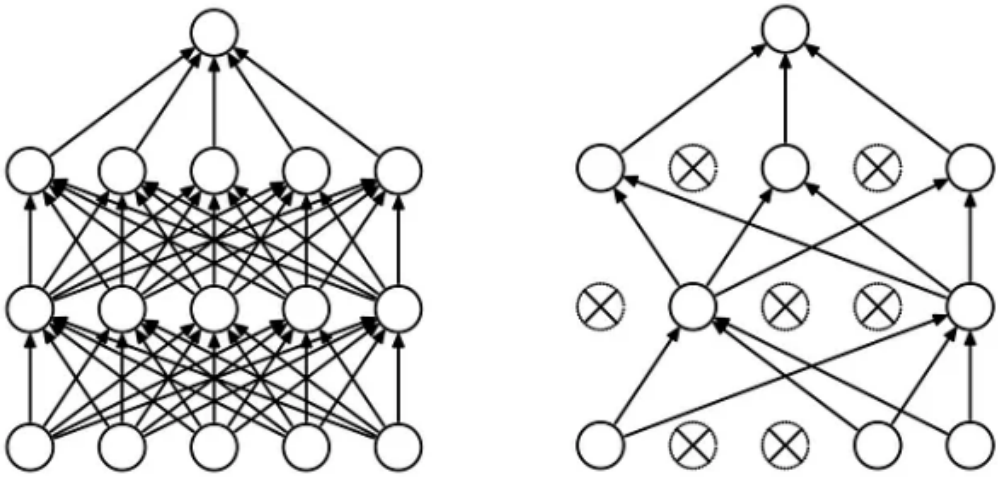
- **Función de pérdida:** Durante el entrenamiento, se utilizó la log-verosimilitud negativa como función de pérdida. Este criterio busca ajustar los parámetros del modelo de manera que la probabilidad asignada a la clase correcta para cada instancia de entrenamiento sea máxima.

  La fórmula para un conjunto de datos D con etiquetas y, y predicciones del modelo p se expresa como:

$$\mathrm{J}(\theta) = -\sum_{i=1}^{N}\sum_{j=1}^{C}[y_i = j]\, f_{ij}(\theta)$$

donde N es el número de ejemplos en el conjunto de datos, C es el número de clases, y_i es la etiqueta de la i-ésima instancia y $f_{ij}(\theta)$ es la log-probabilidad predicha por el modelo para la clase j en la i-ésima instancia, por esto no se toma logaritmo en la expresión anterior.

- **Regularización:** Se aplicó la técnica de regularización *Dropout* durante el entrenamiento de los modelos. Consiste en apagar un porcentaje determinado de neuronas, fijándolas a cero, de forma aleatoria en las capas indicadas con la finalidad de adquirir mayor capacidad de generalización en el entrenamiento. Esto obliga a la red a predecir sin la necesidad de utilizar todos los parámetros. El valor de dropout utilizado fue de 0.4 para los modelos 1 y 3, y de 0.6 para el modelo 2. En la siguiente figura se puede ver una representación gráfica de esta técnica:



(a) Red Neuronal Estándar (b) Después de aplicar Dropout

- **Configuración Adicional:** La tasa de aprendizaje inicial se estableció en 10^{-3}, y se entrenaron los modelos durante 200 épocas.

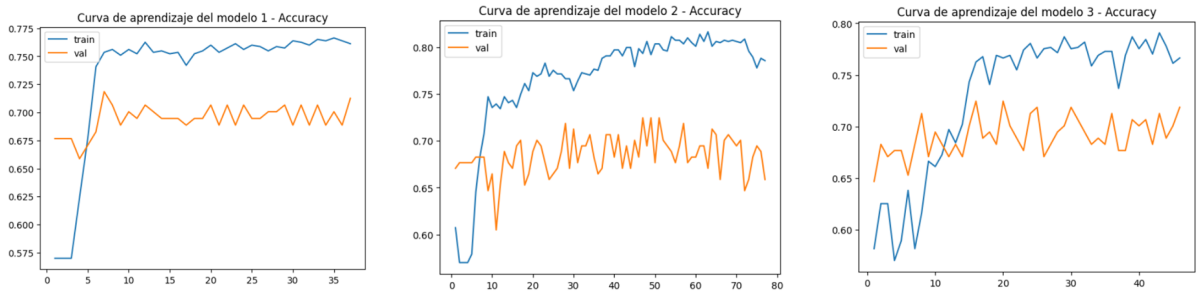
3.4. Resultados

Se ha evaluado el rendimiento de los modelos en el conjunto de validación. Los porcentajes de aciertos (*accuracy*) obtenidos fueron los siguientes:

Modelo	Accuracy en Validación (%)
1	71.9
2	72.5
3	72.5

Además, se presentan en la Figura 8 las curvas de aprendizaje de los modelos en el conjunto de entrenamiento y validación a lo largo de las épocas. Estas curvas ofrecen una representación

de cómo el rendimiento del modelo evoluciona durante el entrenamiento.



(a) Modelo 1 (b) Modelo 2 (c) Modelo 3

Figura 8: Curvas de aprendizaje

Debido al empate obtenido en validación entre los modelos 2 y 3, se ha optado por el Modelo 2 debido a que la entrada de la red neuronal preserva la estructura original de los datos, ya que no es necesario añadir un nodo global.

En cuanto a la evaluación en el conjunto de prueba, el porcentaje de aciertos obtenido fue del 75.4 %.

Además, para este modelo elegido, se ha hecho un Análisis de Componentes Principales (*PCA*), siguiendo la idea mostrada en [8] sobre el vector de la penúltima capa de nuestra *GNN*, que en este caso corresponde con la penúltima capa de la *MLP* final, para visualizar el correcto aprendizaje de nuestro modelo. El *PCA* es una técnica de reducción de dimensionalidad que permite visualizar datos complejos en un espacio bidimensional o tridimensional.

Así, se puede proyectar en dos dimensiones las representaciones de los grafos dadas por la penúltima capa de proteínas según el paso de las épocas, esto proporciona información sobre cómo evoluciona la capacidad del modelo para separar las clases. Si las clases se vuelven más distintas a medida que pasan las épocas, es un indicativo de que el modelo está aprendiendo a diferenciar de manera efectiva las características de los grafos de proteínas.

En la Figura 9 se muestran algunos fotogramas de un vídeo creado para representar estas proyecciones respecto al número de épocas. Se puede observar que a medida que aumentan las épocas, las proteínas se van separando mejor entre enzimas (rojo) y no enzimas (azul).

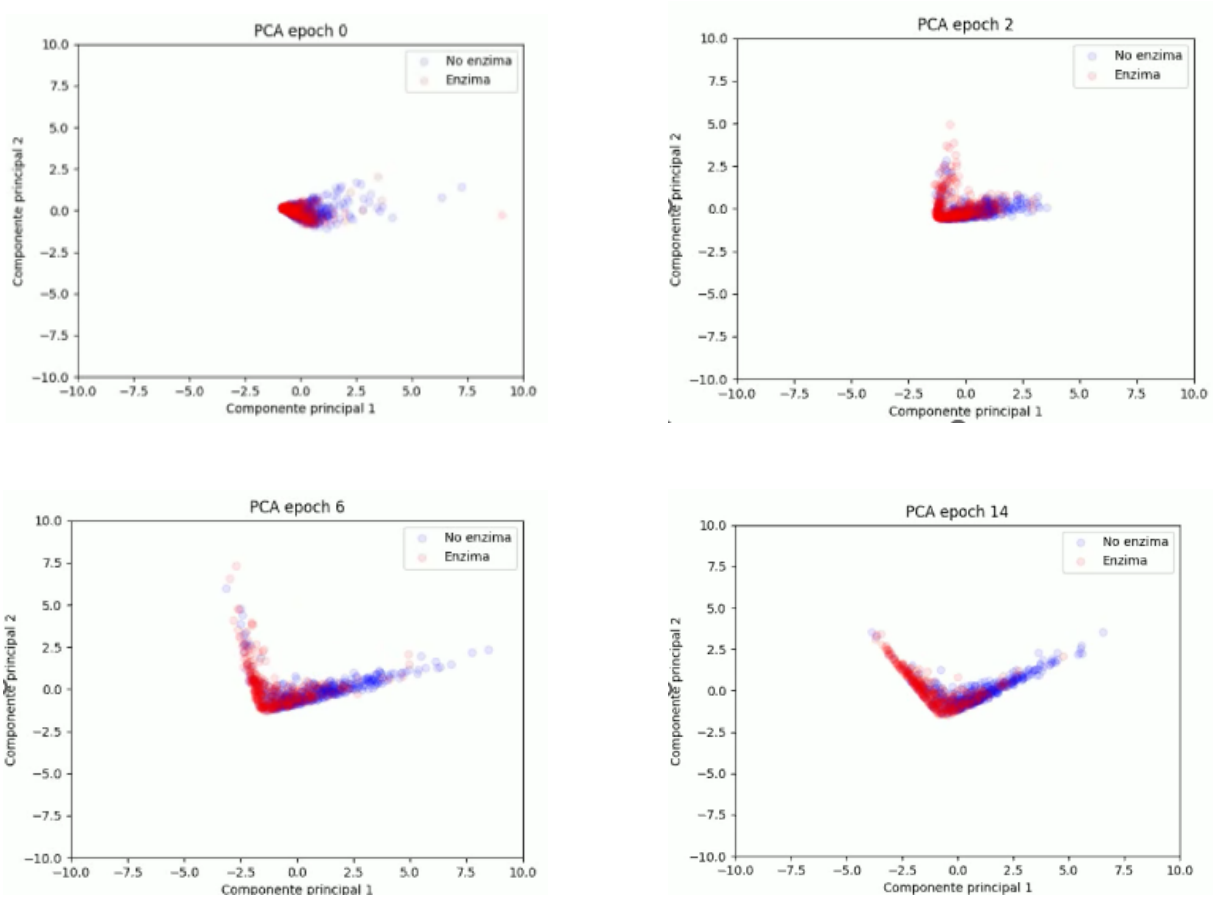


Figura 9: Proyección sobre las 2 Componentes Principales de las representaciones de las proteínas dadas por la penúltima capa.

4. Conclusión

En conclusión, en este trabajo se ha realizado una introducción a las redes neuronales de grafos (*GNN*) y una aplicación sencilla en la clasificación de proteínas. Sin embargo, esto solo es una primera vista del gran potencial de las *GNN* en el campo del aprendizaje automático. Se anima al lector a explorar técnicas más avanzadas y a investigar en arquitecturas más sofisticadas, ya que las *GNN* están en constante evolución actualmente y se posicionan como una de las herramientas más relevantes en los próximos años. En el futuro promete la posibilidad de aprovechar las capacidades de las redes neuronales de grafos para diversas aplicaciones. De hecho, ya se están empezando a aplicar en el campo del procesamiento de imágenes y del lenguaje natural, debido a la flexibilidad que ofrece la estructura de grafo.

A. Anexo: Códigos Modelos

A.1. Modelo 1

```
class Modelo1(torch.nn.Module):
    def __init__(self, hidden_channels=80):
        super(Modelo1, self).__init__()
        self.dropout_ratio = 0.4
        self.conv1 = GCNConv(3, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, hidden_channels)
        self.conv3 = GCNConv(hidden_channels, hidden_channels)
        self.lin1 = Linear(2*hidden_channels, hidden_channels)
        self.lin2 = Linear(hidden_channels, hidden_channels//2)
        self.lin3 = Linear(hidden_channels//2, 2)

    def forward(self, x, edge_index, batch):
        x = self.conv1(x, edge_index)
        x = x.relu()
        x = self.conv2(x, edge_index)
        x = x.relu()
        x = self.conv3(x, edge_index)
        x1 = gap(x, batch)
        x2 = gmp(x, batch)
        x = torch.cat([x1, x2], dim=1)

        x = F.relu(self.lin1(x))
        x = F.dropout(x, p=self.dropout_ratio, training=self.training)
        x = F.relu(self.lin2(x))
        x = F.dropout(x, p=self.dropout_ratio, training=self.training)
        x = self.lin3(x)
        x = F.log_softmax(x, dim=-1)

        return x
```

```
Modelo1(
  (conv1): GCNConv(3, 80)
  (conv2): GCNConv(80, 80)
  (conv3): GCNConv(80, 80)
  (lin1): Linear(in_features=160, out_features=80, bias=True)
  (lin2): Linear(in_features=80, out_features=40, bias=True)
```

```
  (lin3): Linear(in_features=40, out_features=2, bias=True)
)
Total de params: 29482
```

A.2. Modelo 2

```
class Modelo2(torch.nn.Module):
    def __init__(self, hidden_channels=80):
        super(Modelo2, self).__init__()
        self.dropout_ratio = 0.6
        self.conv1 = GraphConv(3, hidden_channels)
        self.conv2 = GraphConv(hidden_channels, hidden_channels)
        self.conv3 = GraphConv(hidden_channels, hidden_channels)
        self.pool1 = TopKPooling(hidden_channels, 0.5)
        self.pool2 = TopKPooling(hidden_channels, 0.5)
        self.lin1 = Linear(2*hidden_channels, hidden_channels)
        self.lin2 = Linear(hidden_channels, hidden_channels//2)
        self.lin3 = Linear(hidden_channels//2, 2)

    def forward(self, x, edge_index, batch):
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x, edge_index, _, batch, _, _ = self.pool1(x, edge_index, None, batch)
        x1 = torch.cat([gmp(x, batch), gap(x, batch)], dim=1)

        x = self.conv2(x, edge_index)
        x = F.relu(x)
        x, edge_index, _, batch, _, _ = self.pool2(x, edge_index, None, batch)
        x2 = torch.cat([gmp(x, batch), gap(x, batch)], dim=1)

        x = self.conv3(x, edge_index)
        x = F.relu(x)
        x3 = torch.cat([gmp(x, batch), gap(x, batch)], dim=1)

        x = F.relu(x1) + F.relu(x2) + F.relu(x3)

        x = F.relu(self.lin1(x))
        x = F.dropout(x, p=self.dropout_ratio, training=self.training)
```

```
        x = F.relu(self.lin2(x))
        x = F.dropout(x, p=self.dropout_ratio, training=self.training)
        x = self.lin3(x)
        x = F.log_softmax(x, dim=-1)

        return x

Modelo2(
  (conv1): GraphConv(3, 80)
  (conv2): GraphConv(80, 80)
  (conv3): GraphConv(80, 80)
  (pool1): TopKPooling(80, ratio=0.5, multiplier=1.0)
  (pool2): TopKPooling(80, ratio=0.5, multiplier=1.0)
  (lin1): Linear(in_features=160, out_features=80, bias=True)
  (lin2): Linear(in_features=80, out_features=40, bias=True)
  (lin3): Linear(in_features=40, out_features=2, bias=True)
)
Total de params: 42682
```

A.3. Modelo 3

```
class Modelo3(torch.nn.Module):
    def __init__(self, hidden_channels=80):
        super(Modelo3, self).__init__()
        self.dropout_ratio = 0.4
        self.conv1 = GraphConv(3, hidden_channels)
        self.conv2 = GraphConv(hidden_channels, hidden_channels)
        self.conv3 = GraphConv(hidden_channels, hidden_channels)
        self.lin1 = Linear(hidden_channels, hidden_channels)
        self.lin2 = Linear(hidden_channels, hidden_channels//2)
        self.lin3 = Linear(hidden_channels//2, 2)

    def forward(self, x, edge_index, batch):
        x = self.conv1(x, edge_index)
        x = x.relu()
        x = self.conv2(x, edge_index)
        x = x.relu()
        x = self.conv3(x, edge_index)
```

```
        x = torch.stack([x[batch == B][0] for B in torch.unique(batch)], dim=0)

        x = F.relu(self.lin1(x))
        x = F.dropout(x, p=self.dropout_ratio, training=self.training)
        x = F.relu(self.lin2(x))
        x = F.dropout(x, p=self.dropout_ratio, training=self.training)
        x = self.lin3(x)
        x = F.log_softmax(x, dim=-1)

        return x

Modelo3(
  (conv1): GraphConv(3, 80)
  (conv2): GraphConv(80, 80)
  (conv3): GraphConv(80, 80)
  (lin1): Linear(in_features=80, out_features=80, bias=True)
  (lin2): Linear(in_features=80, out_features=40, bias=True)
  (lin3): Linear(in_features=40, out_features=2, bias=True)
)
Total de params: 36122
```

Referencias

[1] R. Lam, A. Sanchez-Gonzalez, M. Willson, P. Wirnsberger, M. Fortunato, F. Alet, ..., and P. Battaglia. Graphcast: Learning skillful medium-range global weather forecasting. *arXiv preprint arXiv:2212.12794*, 2022.

[2] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[3] Christopher Morris, Marcel Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gopal Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 4602–4609, 2019, month=July.

[4] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[5] Gabriele Corso, Luca Cavalleri, Dominique Beaini, Pietro Liò, and Petar Veličković. Principal neighbourhood aggregation for graph nets. *Advances in Neural Information Processing Systems*, 33:13260–13271, 2020.

[6] Hongyang Gao and Shuiwang Ji. Graph u-nets. In *International Conference on Machine Learning*, pages 2083–2092. PMLR, 2019.

[7] Zonghan Zhang, Jiajun Bu, Martin Ester, Jian Zhang, Chengwei Yao, Zhi Yu, and Can Wang. Hierarchical graph pooling with structure learning. *arXiv preprint arXiv:1911.05954*, 2019.

[8] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alexander B. Wiltschko. A gentle introduction to graph neural networks. *Distill*, 2021. https://distill.pub/2021/gnn-intro.