Universidad Alfonso X

TÉCNICAS DE OPTIMIZACIÓN Y CONTROL

Práctica 11

Autores:
Pelayo Huerta Mijares
Rubén Nogueras González

Noviembre 2023

1. Redes de Atención Gráfica (GAT)

1.1. Atención en Redes Neuronales

Antes de profundizar en las GAT, es importante entender el concepto de atención en las redes neuronales. La atención es un mecanismo que permite a la red dar más peso a ciertas partes de la entrada en función de su relevancia para la tarea en cuestión. Las redes de atención son útiles para procesar secuencias y seleccionar partes importantes de ellas.

1.2. Redes de Atención Gráfica (GAT)

Las GAT aplican la idea de atención a los grafos. En lugar de tratar todos los nodos por igual, las GAT asignan pesos de atención a los nodos vecinos en función de su importancia relativa para el nodo central. Cada nodo en una GAT calcula un conjunto de pesos de atención para sus nodos vecinos utilizando una función de atención. Estos pesos se utilizan para combinar las representaciones de los nodos vecinos en una representación agregada para el nodo central. La fórmula típica para calcular los pesos de atención en una GAT es mediante una operación de atención basada en redes neuronales (por ejemplo, una red neuronal feedforward). Esta operación asigna pesos a los nodos vecinos en función de sus características y las del nodo central. Los pesos de atención pueden ser normalizados utilizando funciones de activación como la función Softmax para que sumen 1, lo que garantiza que se distribuya la atención correctamente.

1.3. Capas Múltiples

Las GAT pueden tener varias capas, lo que permite realizar múltiples pasos de atención. Esto ayuda a capturar patrones de relaciones más complejos en los grafos.

1.4. Aplicaciones

Las GAT se utilizan en una variedad de aplicaciones, como la recomendación de redes sociales, el procesamiento del lenguaje natural, la bioinformática y la detección de anomalías en redes, entre otras.

2. Capa de atencion

En esta sección vamos a explicar más detalladamente la capa de atención (mencionada anteriormente) de una Red de Atención Gráfica (GAT).

La autotención permite que un nodo en un grafo considere la información de sus nodos vecinos, asignando pesos a esos vecinos en función de su importancia relativa para el nodo central. En otras palabras, algunos nodos tienen mayor importancia.

Cabe destacar que hablamos de autoatención y no de atención ya que las entradas de los nodos se comparan entre sí.

Cada nodo en un grafo, denotado como 'i', posee un vector de características 'Xi' que describe sus atributos o información específica. La capa de Red de Atención Gráfica (GAT) calcula la representación actualizada del nodo 'i' al combinar información de sus vecinos en el grafo. Esta combinación se logra mediante el cálculo de un conjunto de coeficientes de atención que ponderan las características de los nodos vecinos, y luego se multiplican por una matriz de pesos compartida 'W'.

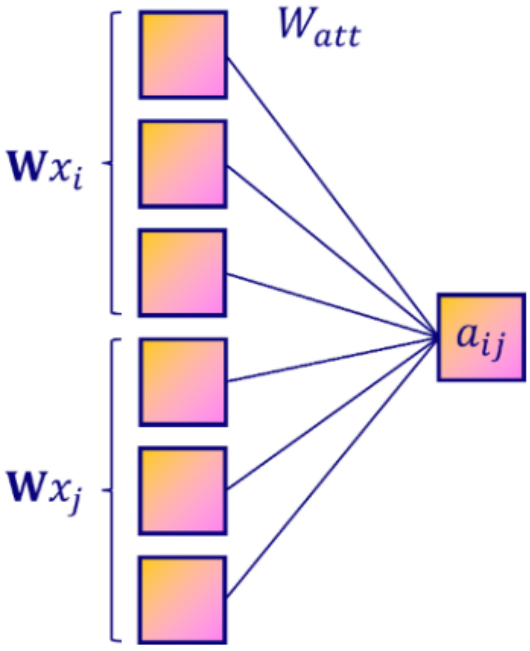
$$h_i = \alpha_{11} W x_1 + \alpha_{12} W x_2 + \alpha_{13} W x_3 + \alpha_{14} W x_4 \tag{1}$$

Y para calcular dichos coeficientes de atención, podemos obtener sus valores mediante una red neuronal. Destacamos 4 pasos:

2.1. Transformación Lineal

Concatenamos vectores de atributos de un pares de nodos, por lo que podemos aplicar una transformación lineal, con una matriz de peso correspondiente W_{att}.

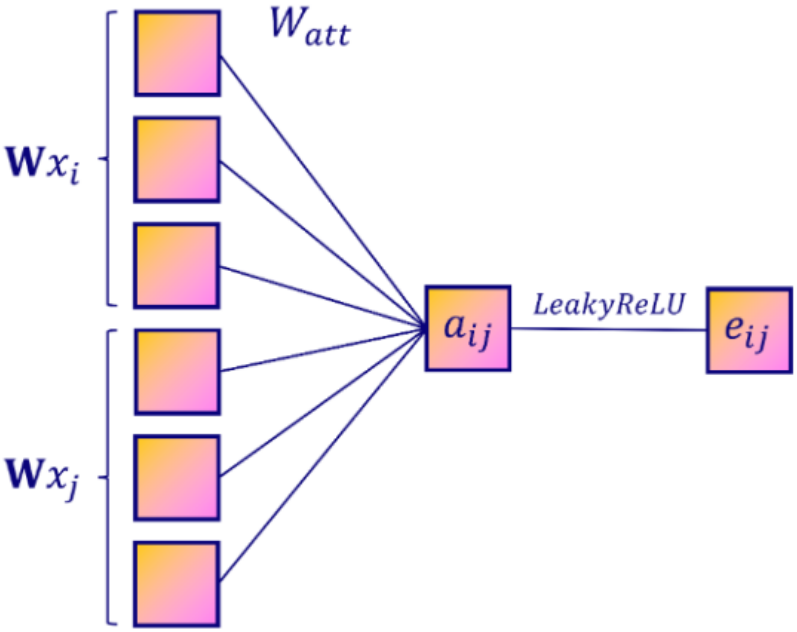
$$a_{ij} = W_{att}^t [W x_i || W x_j] \tag{2}$$



2.2. Función de activación

Integramos no linealidad, ya que estamos con una red neuronal, y esto lo hacemos mediante una función de activación (elegimos la función LeakyReLU).

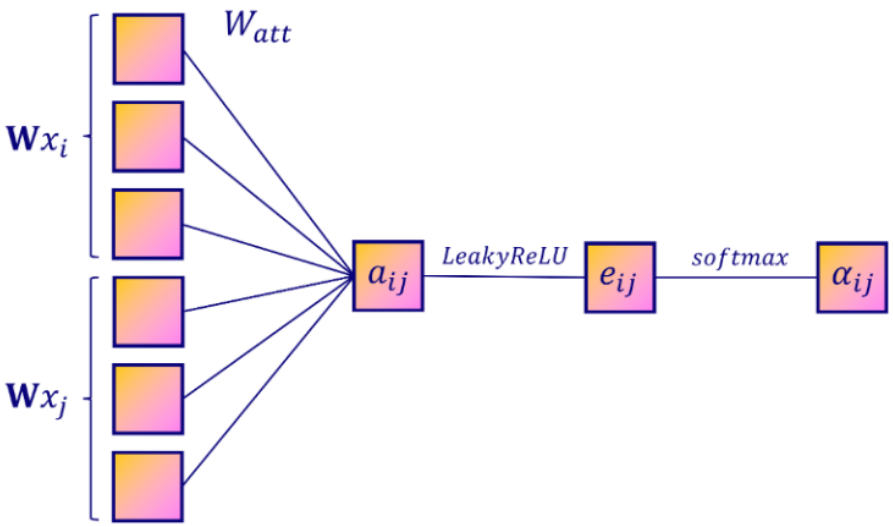
$$e_{ij} = LeakyReLU(a_{ij}) \tag{3}$$



2.3. Normalización Softmax

Hay un problema, y es que la salida de dicha red neuronal no está normalizada, ya que queremos comparar los coeficientes. Por lo tanto aplicamos la función softmax a todos los nodos vecinos:

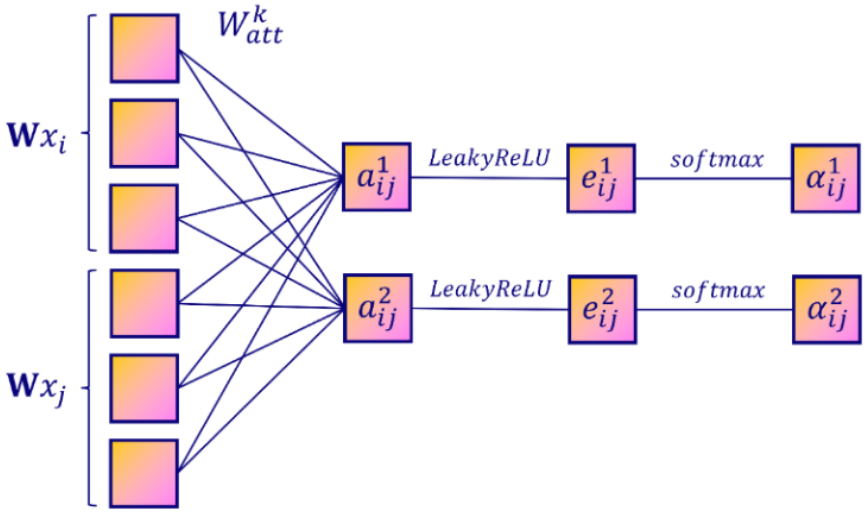
$$\alpha_{ij} = softmax_j(e_{ij}) = \frac{exp(e_{ij})}{\sum_{k \in N_i} exp(e_{ik})} \tag{4}$$



2.4. Atención de múltiples cabezas

Con softmax, ya tenemos los coeficientes de atención, el último problema qu se nos presenta es que la autoatención no es muy estable, y para solucionar esto, introducimos la atención de múltiples cabezas en la arquitectura del transformador.

Dichos transformadores, son un caso particular de GNN, de ahí su similitud, y gracias a ellos podemos incluir la atención de varias cabezas (una idea del procesamiento del lenguaje natural).



En los GAT, en el proceso de atención multicabezal se repiten los mismos 3 pasos para hacer un promedio o concatenar los resultados:

- **Promedio**:

$$h_i = \frac{1}{n}\sum_{k=1}^{n} h_i^k \tag{5}$$

- **Concatenación**:

$$h_i = ||_{k=1}^{n} h_i^k \tag{6}$$

En la implementación práctica, empleamos el método de concatenación cuando se trata de una capa oculta, mientras que utilizamos el enfoque de promedio cuando se trata de la capa final (de salida). En la mayoría de los casos, apilamos múltiples capas GAT para incluir un vecindario más extenso, lo que nos permite combinar estos dos enfoques dentro del mismo modelo GAT.

3. Implementando una Red de Atención Gráfica

En este apartado implementaremos un GAT en Python usando PyTorch Geometric usando la capa Gatv2Conv.

```
import torch.nn.functional as F
from torch.nn import Linear, Dropout
from torch_geometric.nn import GCNConv, GATv2Conv

class GCN(torch.nn.Module):
    """Graph Convolutional Network"""
    def __init__(self, dim_in, dim_h, dim_out):
      super().__init__()
      self.gcn1 = GCNConv(dim_in, dim_h)
      self.gcn2 = GCNConv(dim_h, dim_out)
      self.optimizer = torch.optim.Adam(self.parameters(),
                                        lr=0.01,
                                        weight_decay=5e-4)
    def forward(self, x, edge_index):
        h = F.dropout(x, p=0.5, training=self.training)
        h = self.gcn1(h, edge_index).relu()
        h = F.dropout(h, p=0.5, training=self.training)
        h = self.gcn2(h, edge_index)
        return h, F.log_softmax(h, dim=1)

class GAT(torch.nn.Module):
    """Graph Attention Network"""
    def __init__(self, dim_in, dim_h, dim_out, heads=8):
        super().__init__()
        self.gat1 = GATv2Conv(dim_in, dim_h, heads=heads)
```

```
        self.gat2 = GATv2Conv(dim_h*heads, dim_out, heads=1)
        self.optimizer = torch.optim.Adam(self.parameters(),
                                          lr=0.005,
                                          weight_decay=5e-4)

    def forward(self, x, edge_index):
        h = F.dropout(x, p=0.6, training=self.training)
        h = self.gat1(h, edge_index)
        h = F.elu(h)
        h = F.dropout(h, p=0.6, training=self.training)
        h = self.gat2(h, edge_index)
        return h, F.log_softmax(h, dim=1)

def accuracy(pred_y, y):
    """Calculate accuracy."""
    return ((pred_y == y).sum() / len(y)).item()

def train(model, data):
    """Train a GNN model and return the trained model."""
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = model.optimizer
    epochs = 200
    model.train()
    for epoch in range(epochs+1):
        # Training
        optimizer.zero_grad()
        _, out = model(data.x, data.edge_index)
        loss = criterion(out[data.train_mask], data.y[data.train_mask])
        acc = accuracy(out[data.train_mask].argmax(dim=1), data.y[data.
            train_mask])
        loss.backward()
        optimizer.step()
        # Validation
        val_loss = criterion(out[data.val_mask], data.y[data.val_mask])
        val_acc = accuracy(out[data.val_mask].argmax(dim=1), data.y[data.
            val_mask])
        # Print metrics every 10 epochs
        if(epoch % 10 == 0):
            print(f'Epoch {epoch:>3} | Train Loss: {loss:.3f} | Train Acc: '
                  f'{acc*100:>6.2f}% | Val Loss: {val_loss:.2f} | '
                  f'Val Acc: {val_acc*100:.2f}%')
    return model

@torch.no_grad()
def test(model, data):
    """Evaluate the model on test set and print the accuracy score."""
    model.eval()
    _, out = model(data.x, data.edge_index)
    acc = accuracy(out.argmax(dim=1)[data.test_mask], data.y[data.test_mask])
    return acc
%%time
# Create GCN model
```

```
gcn = GCN(dataset.num_features, 16, dataset.num_classes)
print(gcn)

# Train and test
train(gcn, data)
acc = test(gcn, data)
print(f'\nGCN test accuracy: {acc*100:.2f}%\n')
```

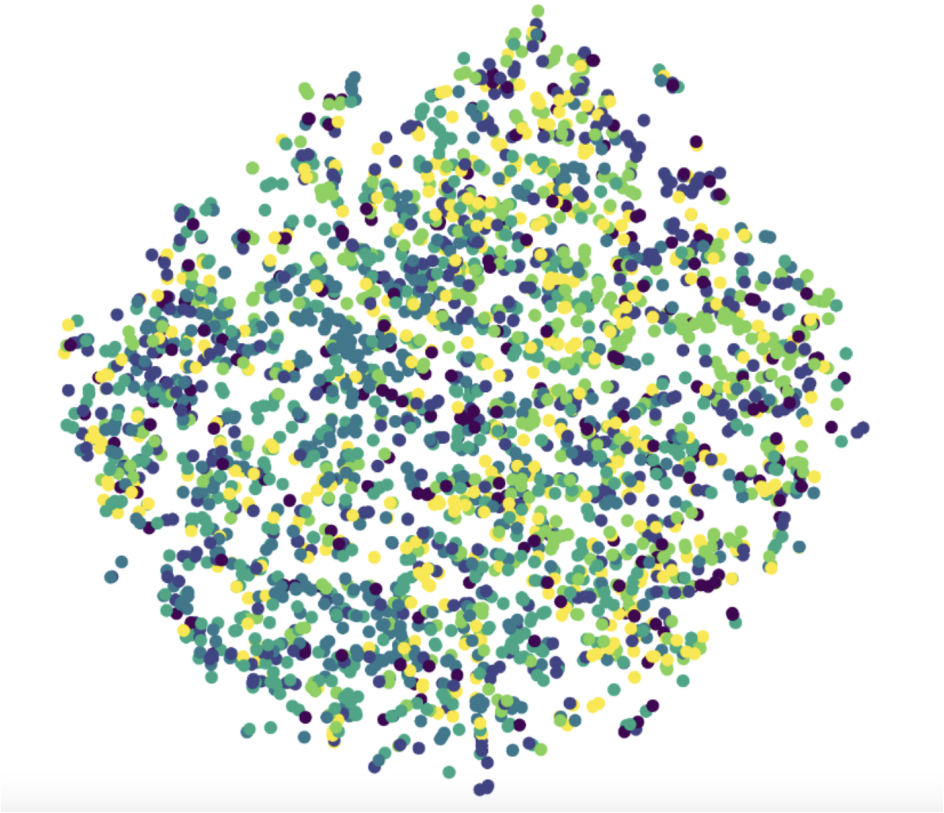
```
GCN test accuracy: 67.70%

CPU times: user 25.1 s, sys: 847 ms, total: 25.9 s
Wall time: 32.4 s
```

```
GAT test accuracy: 70.00%

CPU times: user 53.4 s, sys: 2.68 s, total: 56.1 s
Wall time: 55.9 s
```

En este ejemplo, que a pesar de que no es super riguroso, podemos ver que GAT obtiene una mayor precisión (70,0 %) que GCN (67,7 %). Sin embargo, GAT tarda 55,9 segundos en entrenarse y GCN tan solo 32,4 segundos. Esto es una compensación que puede causar problemas de escalabilidad cuando se trabaja con gráficos grandes.

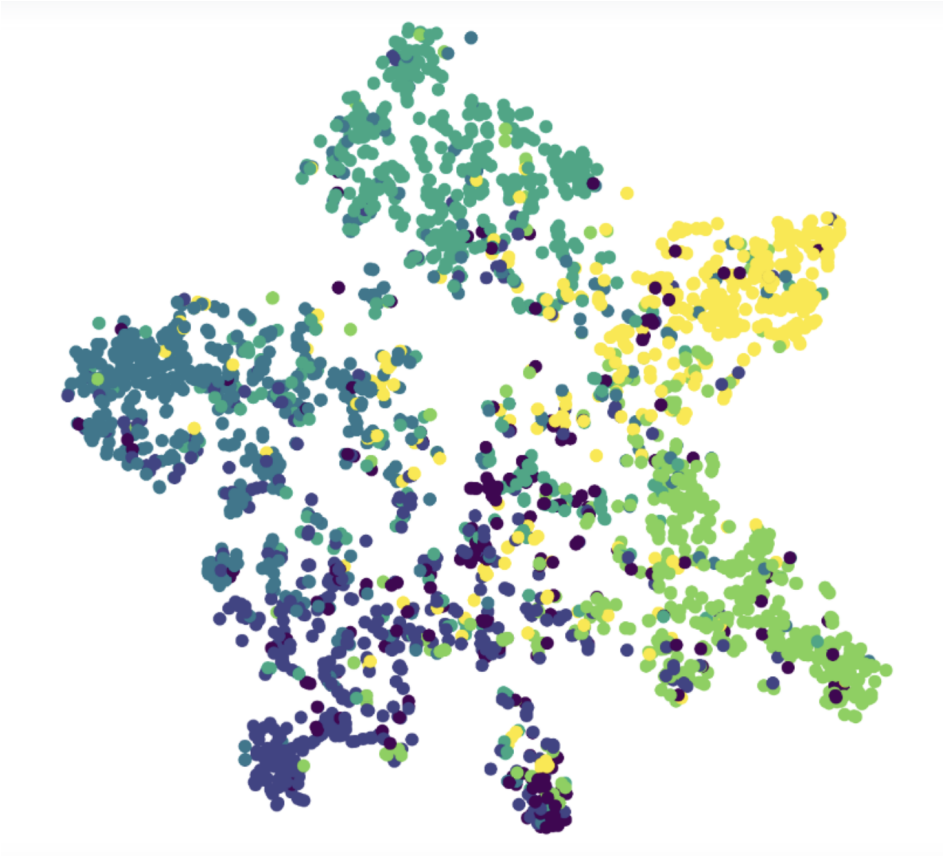
Veamos lo que realmente aprendió el GAT. Esto podemos visualizarlo con el diagrama t-SNE un método poderoso para representar datos de alta dimensión en 2D o 3D. Primero veamos como se veían las incrustaciones antes del entrenamiento:

```
# Initialize new untrained model
untrained_gat = GAT(dataset.num_features, 8, dataset.num_classes)

# Get embeddings
h, _ = untrained_gat(data.x, data.edge_index)

# Train TSNE
tsne = TSNE(n_components=2, learning_rate='auto',
         init='pca').fit_transform(h.detach())

# Plot TSNE
plt.figure(figsize=(10, 10))
plt.axis('off')
plt.scatter(tsne[:, 0], tsne[:, 1], s=50, c=data.y)
plt.show()
```



Ahora veamos como se ven las incrustaciones con el modelo entrenado:

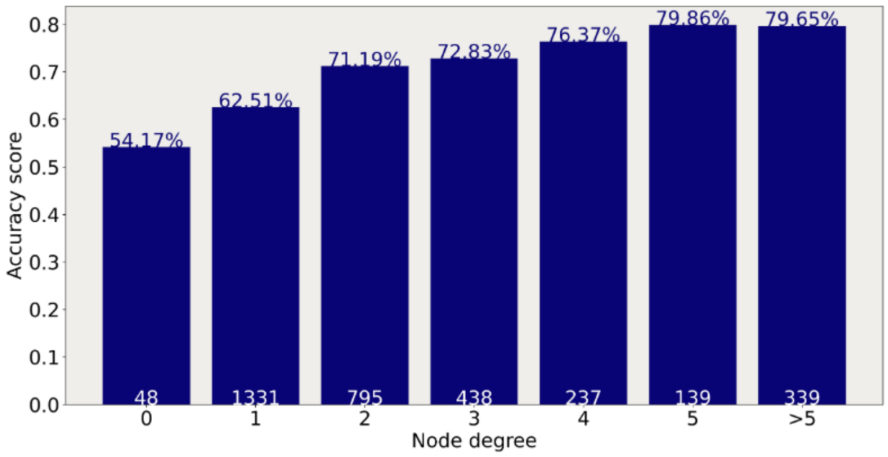
```
# Get embeddings
h, _ = gat(data.x, data.edge_index)

# Train TSNE
tsne = TSNE(n_components=2, learning_rate='auto',
         init='pca').fit_transform(h.detach())

# Plot TSNE
plt.figure(figsize=(10, 10))plt.axis('off')
plt.scatter(tsne[:, 0], tsne[:, 1], s=50, c=data.y)
plt.show()
```



Se ven algunas diferencias: en el modelo entrenado vemos como los nodos que pretenecen a las mismas clases, se agrupan. Mientras que en el gráfico del modelo no entrenado, no se ve relación de ningún tipo. En esta última imagen, podemos ver seis grupos, que corresponden a las seis clases de documentos. También hay valores atípicos, pero es normal ya que nuestra precisión no es perfecta. Verifiquemos que los nodos mal conectados tengan un impacto negativo en el rendimiento de CiteSeer, calculando la precisión del modelo para cada grado:

```
from torch_geometric.utils import degree

# Get model's classifications
_, out = gat(data.x, data.edge_index)

# Calculate the degree of each node
degrees = degree(data.edge_index[0]).numpy()

# Store accuracy scores and sample sizes
accuracies = []
sizes = []

# Accuracy for degrees between 0 and 5
for i in range(0, 6):
  mask = np.where(degrees == i)[0]
  accuracies.append(accuracy(out.argmax(dim=1)[mask], data.y[mask]))
  sizes.append(len(mask))
```

```
18
19 # Accuracy for degrees > 5
20 mask = np.where(degrees > 5)[0]
21 accuracies.append(accuracy(out.argmax(dim=1)[mask], data.y[mask]))
22 sizes.append(len(mask))
23
24 # Bar plot
25 fig, ax = plt.subplots(figsize=(18, 9))
26 ax.set_xlabel('Node degree')
27 ax.set_ylabel('Accuracy score')
28 ax.set_facecolor('#EFEEEA')
29 plt.bar(['0','1','2','3','4','5','>5'],
30         accuracies,
31         color='#0A047A')
32 for i in range(0, 7):
33     plt.text(i, accuracies[i], f'{accuracies[i]*100:.2f}%',
34              ha='center', color='#0A047A')for i in range(0, 7):
35     plt.text(i, accuracies[i]//2, sizes[i],
36              ha='center', color='white')
```



Estos resultados confirman que los nodos con pocos vecinos son más difíciles de clasificar. Cuantas más conexiones relevantes tenga más información podrá agregar.

4. Conclusión y generalidaz

Los Grafos de Autoatención (GAT) han emergido como una herramienta sumamente importante en múltiples contextos, desempeñando un papel crucial en una amplia gama de aplicaciones. Su capacidad para modelar relaciones no lineales entre elementos en conjuntos de datos commplejos, los hacen especialmente poderosos en campos como la inteligencia artificial (IA), ciencia de datos y el procesamiento del lenguaje natural.

En el ámbito de la inteligencia artificial, los GAT han revolucionado la forma en que abordamos problemas de aprendizaje automatico. al permitir la captura de relaciones entre entidades en datos estructurados y no estructurados. Su capacidad para aprender representaciones jerárquicas de datos y reconoces patrones complejos es invaluable en tareas que van desde la recomendación personalizada hasta la visión por computadora.

En el procesamiento del leguaje natural, los GAT se han convertido en un componente fundamental en el análisis y la comprensión de textos. Su capacidad para capturar la semántica y la relación entre palabras y oraciones permite un procesamiento más preciso y contextual mejorando la comprensión del lenguaje natural en sistemas de traducción automática, resumen de texto y análisis de sentimiento.

Por último, en el ámbito de las ciencias biológicas, los GAT se han aplicado con éxito en el análisis de redes moleculares y biológicas, lo que permite entender mejor las interacciones complejas entre diferentes moléculas y proteínas. Esto tiene implicaciones muy significativas en la investigación farmacéutica, el descubrimiento de medicamentos y la comprensión de enfermedade.

La versatilidad de los grafos de automatención (GAT) ha demostrado ser una herramienta de gran relevancia en diversos campos, desde la intelingecia artificial hasta la biología. Su capacidad para modelar relaciones complejas entre entidades lo posiciona como un pilar en la resolución de problemas que involucran datos estructurados y no estructurados, y su continua evolución promete avances significativos en numerosos dominios.