Algorithm 1: GNN maps state $s$ into scalar $V(s)$

Input: State $s$, set of atoms true in $s$, set of objects

Output: $V(s)$

1. $f_0(o) \sim \mathcal{N}(0,1)^{k/2}$ for each object $o \in s$

2. for $i \in \{0, \ldots, L-1\}$ do

3. for each atom $q := p(o_1, \ldots, o_m)$ true in $s$ do

4. $m_{q,o} := [\text{MLP}_p(f_i(o_1), \ldots, f_i(o_m))]_j$:

5. for each $o \in s$ do

6. $f_{i+1}(o) := \text{MLP}_U(f_i(o), \text{agg}(\{m_{q,o} | o \in q\}))$:

7. $V := \text{MLP}_2(\sum_{o \in s} \text{MLP}_1(f_i(o)))$
GNNs are often realized as follows (Morris et al., 2019). In each layer, \( t > 0 \), we compute vertex features

\[
h_v^{(t)} := \sigma \left( h_v^{(t-1)} W_0^{(t)} + \sum_{w \in N(v)} h_{w}^{(t-1)} W_1^{(t)} \right) \in \mathbb{R}^c, \tag{1}
\]

for \( v \) in \( V(G) \), where \( W_0^{(t)} \) and \( W_1^{(t)} \) are parameter matrices from \( \mathbb{R}^{d \times c} \) and \( \sigma \) denotes an entry-wise non-linear function, e.g., a sigmoid or a ReLU function. Following Gilmer et al. (2017) and Scarselli et al. (2009), in each layer, \( t > 0 \), we can generalize the above by computing a vertex feature

\[
h_v^{(t)} := \text{UPD}^{(t)} \left( h_v^{(t-1)}, \text{AGG}^{(t)} \left( \{ h_w^{(t-1) \mid w \in N(v)} \} \right) \right),
\]

where \( \text{UPD}^{(t)} \) and \( \text{AGG}^{(t)} \) may be differentiable parameterized functions, e.g., neural networks. In the case of graph-level tasks, e.g., graph classification, one uses

\[
h_G := \text{READOUT} \left( \{ h_v^{(T) \mid v \in V(G)} \} \right),
\]

to compute a single vectorial representation based on learned vertex features after iteration \( T \). Again, \( \text{READOUT} \) may be a differentiable parameterized function. To adapt the parameters of the above three functions, they are optimized end-to-end, usually through a variant of stochastic gradient descent, e.g., (Kingma and Ba, 2015), together with the parameters of a neural network used for classification or regression.

**Graph Neural Networks for Multi-relational Graphs.** In the following, we describe GNN layers for multi-relational graphs, namely R-GCN (Schlichtkrull et al., 2018) and CompGCN (Vashishth et al., 2020). Initial features are computed in the same way as in the previous subsection.

**R-GCN.** Let \( G \) be a labeled multi-relational graph. In essence, R-GCN generalizes Equation 1 by using an additional sum iterating over the different relations. That is, we compute a vertex feature

\[
h_v^{(t)} := \sigma \left( h_v^{(t-1)} W_0^{(t)} + \sum_{i \in 1, \ldots, m} \sum_{j \in \mathcal{R}_i \cap N(v)} h_{w_i}^{(t-1)} W_j^{(t)} \right) \in \mathbb{R}^c, \tag{2}
\]
\[ Q = HW_Q, \quad K = HW_K, \quad V = HW_V, \]
\[ A = \frac{QK^T}{\sqrt{d_v}}, \quad \text{Attn}(H) = \text{softmax}(A)V. \]

**Algorithm 1: GNN maps state \( s \) into scalar \( V(s) \)**

**Input:** State \( s \): set of atoms true in \( s \), set of objects

**Output:** \( V(s) \)

1. \( f_0(o) \sim 0^{k/2} \mathcal{N}(0, 1)^{k/2} \) for each object \( o \in s \);
2. for \( i \in \{0, \ldots, L - 1\} \) do
3.     for each atom \( q := p(o_1, \ldots, o_m) \) true in \( s \) do
4.         // Msgs \( q \rightarrow o \) for each \( o = o_j \) in \( q \)
5.         \( m_{q,o} := [\text{MLP}_p(f_i(o_1), \ldots, f_i(o_m))]_j; \)
6.     for each \( o \) in \( s \) do
7.         // Aggregate, update embeddings
8.         \( f_{i+1}(o) := \text{MLP}_U(f_i(o) \circ \text{agg}(\{m_{q,o} \mid o \in q\})) \);
9.     // Final Readout
10.    \( V := \text{MLP}_2(\sum_{o \in s} \text{MLP}_1(f_L(o))) \)

**R-GCN.** Let \( G \) be a labeled multi-relational graph. In essence, R-GCN generalizes Equation 1 using an additional sum iterating over the different relations. That is, we compute a vertex feature

\[ h_{\text{R-GCN}}^{(t)} := \sigma \left(h_{\text{R-GCN}}^{(t-1)}W_0^{(t)} + \sum_{u \in N(v)} h_{u, \text{R-GCN}}^{(t-1)}W_1^{(t)} \right) \in \mathbb{R}^c, \]

for \( v \in V(G) \), where \( W_0^{(t)} \) and \( W_1^{(t)} \) are parameter matrices from \( \mathbb{R}^{d_{x \times c}} \) and \( \sigma \) denotes a non-linear function, e.g., a sigmoid or a ReLU function. Following Gilmer et al. (2017), in each layer, \( t > 0 \), we can generalize the above by computing a vertex feature

\[ h_v^{(t)} := \text{UPD}^{(t)} \left(h_v^{(t-1)} \circ \text{AGG}^{(t)}(\{h_w^{(t-1)} \mid w \in N(v)\}) \right), \]

**CompGCN.** Let \( G \) be a labeled multi-relational graph. A CompGCN layer generalizes Equation 1 by encoding relational information as edge features. That is, we compute a vertex feature

\[ h_v^{(t)} := \sigma \left(h_v^{(t-1)}W_0^{(t)} + \sum_{u(r) \in \text{rel}(v)} h_{u(r), \text{CompGCN}}^{(t-1)}W_{u(r)}^{(t)} \right) \in \mathbb{R}^c. \]

for each \( v \) in \( V(G) \), where \( W_0^{(t)} \) and \( W_{u(r)}^{(t)} \) are parameter matrices from \( \mathbb{R}^{d_{x \times c}} \) and \( \sigma \) denote entry-wise non-linear function. We note here that the original R-GCN layer defined in Schlichtkrull et al. (2018) uses a mean operation instead of a sum in the most inner sum of Equation 2. We investigate the empirical advantages of these two variations in Section 5.
Algorithm 1: GNN maps state $s$ into scalar $V(s)$

**Input:** State $s$: set of atoms true in $s$, set of objects $o$

**Output:** $V(s)$

1. $f_0(o) \sim \mathcal{N}(0, 1)^{k/2}$ for each object $o \in s$
2. for $i \in \{0, \ldots, L - 1\}$ do
3.   for each atom $q := p(o_1, \ldots, o_m)$ true in $s$ do
4.     // Msgs $q \rightarrow o$ for each $o = o_j$ in $q$
5.     $m_{q,o} := [\text{MLP}_p(f_i(o_1), \ldots, f_i(o_m))]_j$
6.     for each $o$ in $s$ do
7.       // Aggregate, update embeddings
8.       $f_{i+1}(o) := \text{MLP}_U(f_i(o), \text{agg}([m_{q,o} | o \in q]))$
9.     // Final Readout
10. $V := \text{MLP}_2(\sum_{o \in s} \text{MLP}_1(f_L(o)))$

---

**Diagram:**

The diagram illustrates the architecture of the GNN model. It includes feedforward neural networks, self-attention layers, and multiplication operations. The text mentions the use of linear maps in the diagram, indicating that the GNN model incorporates a linear transformation component, possibly for mapping or combining the output of different layers.
That's pretty much all there is to multi-headed self-attention. It's quite a handful of matrices, I realize. Let me try to put them all in one visual so we can look at them in one place.

1) This is our input sentence*  
2) We embed each word*  
3) Split into 8 heads. We multiply X or R with weight matrices  
4) Calculate attention using the resulting Q/K/V matrices  
5) Concatenate the resulting Z matrices, then multiply with weight matrix Wo to produce the output of the layer

* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one.

\[
\begin{align*}
\hat{h}_{i+1}^\ell &= O_h^\ell \left( \sum_{k=1}^H w_{ij}^{k,\ell} V_{ij}^{k,\ell} h_i^{\ell} \right), \\
\text{where, } w_{ij}^{k,\ell} &= \text{softmax}_j \left( \frac{Q_{k,ij}^{k,\ell} h_i^{\ell} V_{ij}^{k,\ell}}{\sqrt{d_k}} \right)
\end{align*}
\]