COM S 673: Advanced Topics in Computational Models of Learning
Assignment #3 (part a)

---

1. You need to submit a report and your code to Canvas. Your hard-copy report should include (1) answers to the non-programming part, and (2) analysis and results of the programming part. Please put all your code files and report into a compressed file named "HW#_FirstName_LastName.zip"
2. Unlimited number of submissions are allowed and the latest one will be timed and graded.
3. Please read and follow submission instructions. No exception will be made to accommodate incorrectly submitted files/reports.
4. All students are required to typeset their reports using latex.
5. Only write your code between the following lines. Do not modify other parts.

   ### YOUR CODE HERE

   ### END YOUR CODE

---

1. (70 points) (Coding Task) Graph Neural Networks.

   In this assignment, you will use PyTorch to implement some graph deep learning layers including graph convolution layer, graph attention layer, graph differentiable pooling layer, and graph pooling layer. You are given a graph classification dataset. The code folder provides the starting code. You must implement the layers using the starting code. In this assignment, you may need a GPU.

   (a) (15 points) Complete the class GConv in "src/utils/layers.py", which implements the graph convolution layer proposed in [1].

   (b) (15 points) Complete the class GAttn in "src/utils/layers.py", which implements the graph attention layer proposed in [2].

   (c) (15 points) Complete the class GDiffPool in "src/utils/layers.py", which implements the graph differentiable pooling layer proposed in [3].

   (d) (15 points) Complete the class GPool in "src/utils/layers.py", which implements the graph pooling layer proposed in [4].

   (e) (10 points) Run the model by "bash run_GNN.sh" and report the mean 10-fold cross-validation performance. After running the model, the 10-fold results are automatically store in the "results/PROTEINS.txt" file. Please try different graph convolution layers (GConv or GAttn) and different graph pooling layers (GDiffPool or GPool) in "configs/PROTEINS" file and provide a short analysis of the results.

   Required Materials:

   [1] Semi-Supervised Classification with Graph Convolutional Networks (`https://arxiv.org/pdf/1609.02907.pdf`)

[2] Graph Attention Networks (https://arxiv.org/pdf/1710.10903.pdf)

[3] Hierarchical Graph Representation Learning with Differentiable Pooling (https://arxiv.org/pdf/1806.08804.pdf)

[4] Graph U-Nets (https://arxiv.org/pdf/1905.05178.pdf)

Answer. The results are provided in the file named "re" for reproduction. Different weight initialization methods were tested for the weight matrix in the Graph Convolutional Neural Network, and Xavier initialization consistently outperformed the others. Therefore, we proceeded with Xavier initialization in our experiments. See Table 1 for the primarily results of each version.

GConv	GPool	Accuracy	# Heads	Nodes Drop	Drop Out	LR	Batch
GConv	GPool	73.21%	–	0.3	0.3	0.001	64
GConv	GDiffPool	64.29%	–	0.3	0.3	0.001	64
GAttn	**GDiffPool**	**78.57%**	**3**	**0.3**	**0.3**	**0.001**	**64**
GAttn	GPool	73.21%	3	0.3	0.3	0.001	64
GAttn	GDiffPool	71.43%	3	0.2	0.2	0.001	64
GAttn	GPool	72.32%	3	0.2	0.2	0.001	64
GConv	GPool	74.11%	–	0.2	0.2	0.001	64
GConv	GDiffPool	72.32%	–	0.2	0.2	0.001	64
GAttn	GDiffPool	70.54%	9	0.2	0.2	0.001	64
GAttn	GDiffPool	68.75%	9	512	48	0.001	128
GAttn	GDiffPool	72.32%	4	512	48	0.001	128
GAttn	GDiffPool	69.64%	2	512	48	0.001	128
GAttn	GDiffPool	70.54%	3	512	48	0.001	128

Table 1: GNN configurations and corresponding accuracy

Graph Attention Networks (GAttn) generally outperform standard Graph Convolutional Networks (GConv), especially when combined with DiffPool. This highlights the strength of attention mechanisms in learning more informative node representations through the adaptive weighting of neighbors. While GPool performs better with GConv layers (e.g., 74.11% vs. 64.29%), DiffPool shows stronger synergy with GAttn, particularly when using optimized hyperparameters. Finally, moderate numbers of attention heads (e.g., 3 or 4) yield better performance compared to higher (9) or lower (2) head counts. Too many heads may dilute attention focus, while too few may limit expressivity.

2. (30 points) As introduced in class, the attention mechanism can be written into:

$$\text{Attention}(Q, K, V) = \text{softmax}(QK^T)V.$$

By adding linear transformations on Q, K, and V, it turns into:

$$\text{Attention}(QW^Q, KW^K, VW^V) = \text{softmax}(QW^Q(KW^K)^T)VW^V.$$

Here, we set $Q \in \mathbb{R}^{n\times d}$, $W^Q \in \mathbb{R}^{d\times d}$, $K \in \mathbb{R}^{n\times d}$, $W^K \in \mathbb{R}^{d\times d}$, $V \in \mathbb{R}^{n\times d}$, $W^V \in \mathbb{R}^{d\times d}$.

In practice, the multi-head attention is used, which is defined as:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(head_1, \ldots, head_h),$$

where

$$head_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V),\ i = 1, \ldots, h.$$

Here, Q, K, V are the same as defined above. We set $W_i^Q \in \mathbb{R}^{d\times \frac{d}{h}}$, $W_i^K \in \mathbb{R}^{d\times \frac{d}{h}}$, $W_i^V \in \mathbb{R}^{d\times \frac{d}{h}}$.

(a) (10 points) Compute and compare the number of parameters between the single-head and multi-head attention.

Answer. For a given $Q, K, V \in \mathbb{R}^{n \times d}$

i. For a single-head: $W^Q, W^K, W^V \in \mathbb{R}^{d \times d}$

ii. For a multi-head: we have h heads, and for each head: $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d \times \frac{d}{h}}$

Single-Head Attention Parameters

$$\text{Params} = \underbrace{d \times d}_{W^Q} + \underbrace{d \times d}_{W^K} + \underbrace{d \times d}_{W^V} = 3d^2$$

Multi-Head Attention Parameters

Each of the h heads has:

$$\text{Params per head} = d \times \frac{d}{h} + d \times \frac{d}{h} + d \times \frac{d}{h} = 3d \cdot \frac{d}{h}$$

So for h heads:

$$\text{Total params} = h \cdot 3d \cdot \frac{d}{h} = 3d^2$$

(b) (10 points) Compute and compare the amount of computation between the single-head and multi-head attention, including the softmax step. Use the big-O notation to show your results.

(Hint2: Quoted from the paper (`https://arxiv.org/pdf/1706.03762.pdf`), "Due to the reduced dimension of each head, the total computational cost is similar to that of single-head attention with full dimensionality.")

Answer. For a given $Q, K, V \in \mathbb{R}^{n \times d}$, we compare single-head and multi-head self-attention.

i. **Single-head attention:**
- Learnable projections: $W^Q, W^K, W^V \in \mathbb{R}^{d \times d}$
- Computing $Q = XW^Q$, $K = XW^K$, $V = XW^V$ costs $\mathcal{O}(nd^2)$.
- Attention scores: $QK^\top \in \mathbb{R}^{n \times n}$ costs $\mathcal{O}(n^2 d)$.
- Softmax and multiplying by V: each step $\mathcal{O}(n^2 d)$.

Total cost:

$$\mathcal{O}(nd^2 + n^2 d)$$

ii. **Multi-head attention with h heads:**
- Each head uses projections $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d \times \frac{d}{h}}$.
- Projecting X to Q_i, K_i, V_i for all heads costs $\mathcal{O}(nd^2)$ in total (same as single-head).
- Computing $Q_i K_i^\top$ and attention per head is $\mathcal{O}(n^2 \cdot \frac{d}{h})$, and there are h heads, so total attention cost: $\mathcal{O}(n^2 d)$.
- Concatenating the outputs and applying a final projection (of size $d \times d$) costs another $\mathcal{O}(nd^2)$.

Total cost:

$$\mathcal{O}(nd^2 + n^2 d)$$

Conclusion: Both single-head and multi-head attention have similar total computational complexity of $\mathcal{O}(nd^2 + n^2 d)$, because in multi-head attention, each head operates on reduced dimension $\frac{d}{h}$, effectively balancing out the cost when multiplied by h heads.

(c) (10 points) In a variant of attention layer, the softmax operator is replaced by $1/n$, which can save a lot of computational resources. Show how this change can reduce the computational cost. Please ignore the computational differences between softmax and $1/n$. You can assume $d < n$, which is common in computer vision applications.
Answer. In the simplified version, the softmax is replaced by $\frac{1}{n}$, so the attention weights matrix becomes:

$$\text{Attention}(Q, K, V) = \left(\frac{1}{n} \cdot \mathbf{1}_{n\times n}\right) V = \text{avg}(V)$$

This implies every query attends equally to every key without any no dynamic weighting, just averaging. So, the computation with be

$$\underbrace{\frac{1}{n}\sum_{i=1}^{n} V_i}_{\text{precomputed average}} \qquad \in \mathbb{R}^d$$

broadcasting this result across all n output rows results in the computational cost $\in \mathcal{O}(nd)$ rather than $\mathcal{O}(n^2 d)$ which is considered as substantial reduction in complexity.