شبکه‌های گرافی توجه: گزارش فاز پیاده‌سازی

محمد ابراهیمی، پارسا عباسی

دانشگاه علم و صنعت ایران، تهران

m_ebrahimi74@comp.iust.ac.ir, parsa_abbasi@comp.iust.ac.ir

۱) لایهٔ GAT

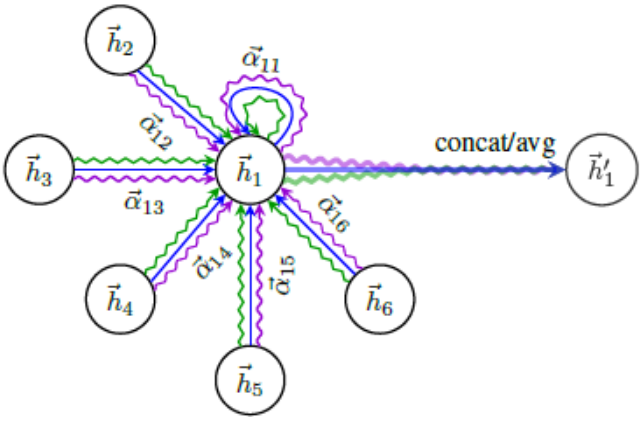
```
def __init__(self, num_in_features, num_out_features, num_of_heads, concat=True, activation=nn.ELU(),
             dropout_prob=0.6, add_skip_connection=True, bias=True, log_attention_weights=False):
```

> The input to our layer is a set of node features, $\mathbf{h} = \{\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_N\}, \vec{h}_i \in \mathbb{R}^F$, where N is the number of nodes, and F is the number of features in each node. The layer produces a new set of node features (of potentially different cardinality F'), $\mathbf{h}' = \{\vec{h}'_1, \vec{h}'_2, \ldots, \vec{h}'_N\}, \vec{h}'_i \in \mathbb{R}^{F'}$, as its output.

ورودی‌های این لایه به شرح زیر هستند:

نام متغیر	مشخصات
num_in_features	تعداد ویژگی‌های ورودی لایه
num_out_features	تعداد ویژگی‌های خروجی لایه
num_of_heads	تعداد هد (۱)
concat	مشخص کنندهٔ اینکه خروجی هر هد باید با یکدیگر الحاق شود (True) یا از آن‌ها میانگین گرفته شود (False) (۲)
activation	تابع فعالسازی (پیش فرض برابر Exponential ReLU یا به اختصار elu)
dropout_prob	احتمال حذف تصادفی
add_skip_connection	استفاده از تکنیک skip connection (۳)
bias	اضافه کردن بایاس
log_attention_weights	تعیین کنندهٔ لاگ گرفتن از وزن‌های توجه

(۱) برای پایدار کردن[1] فرایند یادگیری ممکن است از چندین هد استفاده کنیم. به عنوان مثال در شکل ۱-۱ سه هد مختلف مورد استفاده قرار گرفته که با سه رنگ بنفش، آبی و سبز نشان داده شده‌اند.



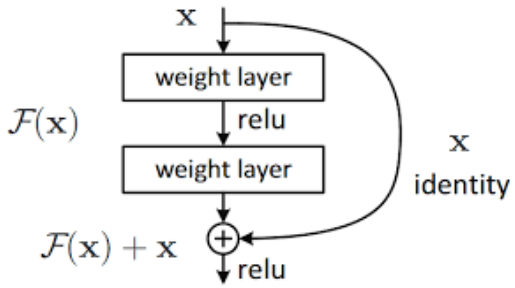
شکل ۱-۱: نحوه عملکرد هدهای مختلف در لایهٔ GAT

[1] Stabilize

(۲) متغیر مربوطه مشخص‌کنندهٔ این است که خروجی هدها را باید با یکدیگر الحاق کرد (لایه‌های میانی) یا از آن‌ها میانگین (لایهٔ خروجی) گرفت.

$$\vec{h}'_i = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}_i}\alpha_{ij}^{k}\mathbf{W}^{k}\vec{h}_j\right) \qquad \vec{h}'_i = \Big\Vert_{k=1}^{K}\sigma\left(\sum_{j\in\mathcal{N}_i}\alpha_{ij}^{k}\mathbf{W}^{k}\vec{h}_j\right)$$

(۳) استفاده از تکنیک رایج skip connection که مطابق با شکل ۲-۱ به منظور انتقال ورودی به کار گرفته می‌شود.



شکل ۲-۱: عملکرد تکنیک skip connection

۱-۱) تبدیل خطی و رگولاریزیشن[۲]

> In order to obtain sufficient expressive power to transform the input features into higher-level features, at least one learnable linear transformation is required. To that end, as an initial step, a shared linear transformation, parametrized by a *weight matrix*, $\mathbf{W} \in \mathbb{R}^{F' \times F}$, is applied to every node.

مطابق با آنچه که در مقاله بدان اشاره شده است، به منظور به دست آوردن ویژگی‌های سطح بالاتری از روی ویژگی‌های ورودی، بر روی آن‌ها حداقل یک تبدیل خطی اعمال خواهیم کرد. ماتریس خطی (معادل با W در مقاله)، وزن‌های توجه (معادل با a در مقاله) و بایاس (در مقاله بدان اشاره‌ای نشده اما در کد اصلی [1] آن پیاده‌سازی شده است)، وزن‌های قابل آموزش این لایه هستند. از آنجا که چندین هد می‌تواند مورد استفاده قرار گیرد بجای اینکه برای هرکدام ماتریس وزن جداگانه تعریف کنیم، می‌توان به شکل زیر همه را در یک لایهٔ مشترک جای داد.

```
self.linear_proj = nn.Linear(num_in_features, num_of_heads * num_out_features, bias=False)
```

✸ همانطور که در روابط زیر مشاهده می‌شود به تعداد هدها ماتریس وزن مستقل اعمال می‌شود، در نتیجه به تعدادی معادل با `num_of_heads` ماتریس وزن خواهیم داشت (که در پیاده‌سازی همگی با یکدیگر یک ماتریس واحد را تشکیل می‌دهند).

$$\vec{h}'_i = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}_i}\alpha_{ij}^{k}\mathbf{W}^{k}\vec{h}_j\right) \qquad \vec{h}'_i = \Big\Vert_{k=1}^{K}\sigma\left(\sum_{j\in\mathcal{N}_i}\alpha_{ij}^{k}\mathbf{W}^{k}\vec{h}_j\right)$$

در هنگام آموزش رو به جلو، ابتدا بر روی بردارهای ویژگی رئوس یک Dropout صورت می‌گیرد و سپس تبدیل خطی که در بالا تعریف شد بر روی آن‌ها اعمال شده و نهایتاً یکبار دیگر از تکنیک Dropout استفاده می‌شود. استفاده از Dropout قبل و بعد از تبدیل خطی مطابق با مقاله و پیاده‌سازی اصلی آن صورت گرفته و بیش‌برازش زودهنگام مدل‌های گرافی به عنوان یکی از دلایل آن ذکر شده است. کد پیاده‌سازی این مراحل در ادامه آورده شده است:

[2] Regularization

```python
def forward(self, data):
    #
    # Step 1: Linear Projection + regularization
    #

    in_nodes_features, edge_index = data  # unpack data
    num_of_nodes = in_nodes_features.shape[self.nodes_dim]
    assert edge_index.shape[0] == 2, f'Expected edge index with shape=(2,E) got {edge_index.shape}'

    # shape = (N, FIN) where N - number of nodes in the graph, FIN - number of input features per node
    # We apply the dropout to all of the input node features (as mentioned in the paper)
    # Note: for Cora features are already super sparse so it's questionable how much this actually helps
    in_nodes_features = self.dropout(in_nodes_features)

    # shape = (N, FIN) * (FIN, NH*FOUT) -> (N, NH, FOUT) where NH - number of heads, FOUT - num of output features
    # We project the input node features into NH independent output features (one for each attention head)
    nodes_features_proj = self.linear_proj(in_nodes_features).view(-1, self.num_of_heads, self.num_out_features)

    nodes_features_proj = self.dropout(nodes_features_proj)  # in the official GAT imp they did dropout here as well

    #
```

همانطور که مشاهده می‌شود بردارهای ویژگی گراف ورودی با ابعاد *(اندازه ویژگی ورودی، تعداد راس‌ها)* پس از تبدیل خطی دارای ابعادی معادل *(اندازه ویژگی خروجی، تعداد هدها، تعداد راس‌ها)* خواهد شد.

۲-۱) محاسبه‌ی ضرایب توجه

در این قسمت به محاسبهٔ ضرایب توجه نرمال نشده یعنی e_{ij} در مقاله می‌پردازیم.

We then perform *self-attention* on the nodes—a shared attentional mechanism $a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \rightarrow \mathbb{R}$ computes *attention coefficients*

$$e_{ij} = a(\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j) \quad (1)$$

در کد پیاده‌سازی شده به جای اینکه بردار a را در الحاق دو بردار Wh_i و Wh_j ضرب داخلی کنیم، a را به دو نیمه‌ی چپ و راست تقسیم کرده و نیمه‌ی چپ را در Wh_i و نیمه‌ی راست را در Wh_j ضرب داخلی می‌کنیم و سپس نتایج حاصل شده را با هم جمع خواهیم کرد. مشخص است که نتیجه نهایی هر دو رویکرد مشابه با یکدیگر خواهد بود.

لازم به ذکر است که ابعاد هر نیمه از بردار a با *(اندازه ویژگی خروجی، تعداد هدها، ۱)* برابر است و پس از ضرب داخلی با حاصل تبدیل خطی با ابعاد *(اندازه ویژگی خروجی، تعداد هدها، تعداد راس‌ها)*، e_{ij} نهایی دارای ابعاد *(تعداد هدها، تعداد راس‌ها)* خواهد شد.

```python
# After we concatenate target node (node i) and source node (node j) we apply the "additive" scoring function
# which gives us un-normalized score "e". Here we split the "a" vector - but the semantics remain the same.
# Basically instead of doing [x, y] (concatenation, x/y are node feature vectors) and dot product with "a"
# we instead do a dot product between x and "a_left" and y and "a_right" and we sum them up
self.scoring_fn_target = nn.Parameter(torch.Tensor(1, num_of_heads, num_out_features))
self.scoring_fn_source = nn.Parameter(torch.Tensor(1, num_of_heads, num_out_features))
```

```python
#
# Step 2: Edge attention calculation
#

# Apply the scoring function (* represents element-wise (a.k.a. Hadamard) product)
# shape = (N, NH, FOUT) * (1, NH, FOUT) -> (N, NH, 1) -> (N, NH) because sum squeezes the last dimension
# Optimization note: torch.sum() is as performant as .sum() in my experiments
scores_source = (nodes_features_proj * self.scoring_fn_source).sum(dim=-1)
scores_target = (nodes_features_proj * self.scoring_fn_target).sum(dim=-1)
```

با این حال، باید توجه داشت که در محاسبهٔ دو ماتریس پیشین تمام راس‌ها حضور دارند. حال آنکه به ازای هر راس (i)، نیاز است e_{ij} را صرفاً براساس راس‌های همسایه (j) محاسبه کنیم.

```
# We simply copy (lift) the scores for source/target nodes based on the edge index. Instead of preparing all
# the possible combinations of scores we just prepare those that will actually be used and those are defined
# by the edge index.
# scores shape = (E, NH), nodes_features_proj_lifted shape = (E, NH, FOUT), E - number of edges in the graph
scores_source_lifted, scores_target_lifted, nodes_features_proj_lifted = self.lift(scores_source, scores_target, nodes_features_proj, edge_index)
```

از همین روی، همانطور که در کد بالا مشاهده می‌شود تابعی به نام ***lift*** مورد استفاده قرار گرفته که به منظور محاسبه‌ی ضرب‌های داخلی $(a_{left}.Wh_i)$ و $(a_{right}.Wh_j)$ با توجه به یالهای بین راس‌های i و j پیاده‌سازی شده است.

```
def lift(self, scores_source, scores_target, nodes_features_matrix_proj, edge_index):
    """
    Lifts i.e. duplicates certain vectors depending on the edge index.
    One of the tensor dims goes from N -> E (that's where the "lift" comes from).

    """
    src_nodes_index = edge_index[self.src_nodes_dim]
    trg_nodes_index = edge_index[self.trg_nodes_dim]

    # Using index_select is faster than "normal" indexing (scores_source[src_nodes_index]) in PyTorch!
    scores_source = scores_source.index_select(self.nodes_dim, src_nodes_index)
    scores_target = scores_target.index_select(self.nodes_dim, trg_nodes_index)
    nodes_features_matrix_proj_lifted = nodes_features_matrix_proj.index_select(self.nodes_dim, src_nodes_index)

    return scores_source, scores_target, nodes_features_matrix_proj_lifted
```

مقدار e_{ij} که تاکنون نسبت به محاسبهٔ آن پرداختیم مربوط به قسمت رنگ شده در فرمول زیر است:

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_j]\right)\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_k]\right)\right)} \qquad (3)$$

حال جهت رسیدن به ضرایب توجه نرمال شده، ابتدا حاصل جمع دو $(a_{left}.Wh_i)$ و $(a_{right}.Wh_j)$ را از یک تابع غیرخطی‌ساز LeakyReLU عبور می‌دهیم.

In our experiments, the attention mechanism a is a single-layer feedforward neural network, parametrized by a weight vector $\vec{\mathbf{a}} \in \mathbb{R}^{2F'}$, and applying the LeakyReLU nonlinearity (with negative input slope $\alpha = 0.2$).

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_j]\right)\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_k]\right)\right)} \qquad (3)$$

```
self.leakyReLU = nn.LeakyReLU(0.2)  # using 0.2 as in the paper, no need to expose every setting
```

```
scores_per_edge = self.leakyReLU(scores_source_lifted + scores_target_lifted)
```

در نهایت با اعمال تابع softmax روی هر گره با توجه به همسایگانش (که خود گره را نیز شامل می‌شود) ضرایب توجه نرمال شده را به دست می‌آوریم:

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\vec{a}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_j]\right)\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\text{LeakyReLU}\left(\vec{a}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_k]\right)\right)} \qquad (3)$$

```
# shape = (E, NH, 1)
attentions_per_edge = self.neighborhood_aware_softmax(scores_per_edge, edge_index[self.trg_nodes_dim], num_of_nodes)
# Add stochasticity to neighborhood aggregation
attentions_per_edge = self.dropout(attentions_per_edge)
```

پس از به دست آوردن softmax روی خروجی آن که در گام بعدی استفاده می‌شود dropout اعمال شده است. تابعی که به منظور اعمال softmax با توجه به همسایگان یک راس و به دست آوردن ضرائب توجه نرمال طراحی شده به شرح زیر است.

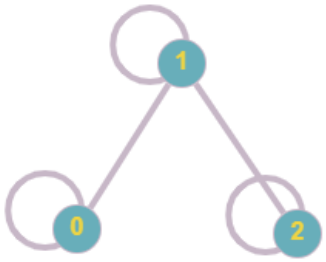
```
def neighborhood_aware_softmax(self, scores_per_edge, trg_index, num_of_nodes):
    """
    As the fn name suggest it does softmax over the neighborhoods. Example: say we have 5 nodes in a graph.
    Two of them 1, 2 are connected to node 3. If we want to calculate the representation for node 3 we should take
    into account feature vectors of 1, 2 and 3 itself. Since we have scores for edges 1-3, 2-3 and 3-3
    in scores_per_edge variable, this function will calculate attention scores like this: 1-3/(1-3+2-3+3-3)
    (where 1-3 is overloaded notation it represents the edge 1-3 and its (exp) score) and similarly for 2-3 and 3-3
     i.e. for this neighborhood we don't care about other edge scores that include nodes 4 and 5.

    Note:
    Subtracting the max value from logits doesn't change the end result but it improves the numerical stability
    and it's a fairly common "trick" used in pretty much every deep learning framework.
    Check out this link for more details:

    https://stats.stackexchange.com/questions/338285/how-does-the-subtraction-of-the-logit-maximum-improve-learning

    """
    # Calculate the numerator. Make logits <= 0 so that e^logit <= 1 (this will improve the numerical stability)
    scores_per_edge = scores_per_edge - scores_per_edge.max()
    exp_scores_per_edge = scores_per_edge.exp()  # softmax

    # Calculate the denominator. shape = (E, NH)
    neigborhood_aware_denominator = self.sum_edge_scores_neighborhood_aware(exp_scores_per_edge, trg_index, num_of_nodes)

    # 1e-16 is theoretically not needed but is only there for numerical stability (avoid div by 0) - due to the
    # possibility of the computer rounding a very small number all the way to 0.
    attentions_per_edge = exp_scores_per_edge / (neigborhood_aware_denominator + 1e-16)

    # shape = (E, NH) -> (E, NH, 1) so that we can do element-wise multiplication with projected node features
    return attentions_per_edge.unsqueeze(-1)
```

باید توجه داشت که در مخرج کسر، تنها آن راس‌هایی در نظر گرفته خواهند شد که مستقیماً (با یک یال) به راس موردنظر متصل هستند. به منظور محاسبهٔ مخرج کسر برای یک راس مشخص تابع زیر مورد استفاده قرار می‌گیرد. ابتدا به دلیل اینکه چندین هد مورد استفاده قرار گرفته لازم است ابعاد بردار شاخص trg_index را از *(تعداد یال‌ها)* به *(تعداد هدها، تعداد یال‌ها)* تبدیل کنیم. سپس به ازای هر راس، قصد داریم به کمک تابع scatter_add تورچ، مجموع امتیاز نمایی راس‌هایی که بدان اشاره می‌کنند (همسایه‌ها) را محاسبه کنیم بنابراین ماتریس نهایی دارای ابعادی معادل با *(تعداد هدها، تعداد راس‌ها)* خواهد بود. با این حال از آنجا که در حال محاسبهٔ softmax بر اساس یال‌ها بودیم، یکبار دیگر این ماتریس را بسط داده و در هرجا که راسی به راس موردنظر ما اشاره می‌کند، حاصل جمع بدست آمده برای راس موردنظر را کپی خواهیم کرد.

```
def sum_edge_scores_neighborhood_aware(self, exp_scores_per_edge, trg_index, num_of_nodes):
    # The shape must be the same as in exp_scores_per_edge (required by scatter_add_) i.e. from E -> (E, NH)
    trg_index_broadcasted = self.explicit_broadcast(trg_index, exp_scores_per_edge)

    # shape = (N, NH), where N is the number of nodes and NH the number of attention heads
    size = list(exp_scores_per_edge.shape)  # convert to list otherwise assignment is not possible
    size[self.nodes_dim] = num_of_nodes
    neighborhood_sums = torch.zeros(size, dtype=exp_scores_per_edge.dtype, device=exp_scores_per_edge.device)

    # position i will contain a sum of exp scores of all the nodes that point to the node i (as dictated by the
    # target index)
    neighborhood_sums.scatter_add_(self.nodes_dim, trg_index_broadcasted, exp_scores_per_edge)

    # Expand again so that we can use it as a softmax denominator. e.g. node i's sum will be copied to
    # all the locations where the source nodes pointed to i (as dictated by the target index)
    # shape = (N, NH) -> (E, NH)
    return neighborhood_sums.index_select(self.nodes_dim, trg_index)
```

```
def explicit_broadcast(self, this, other):
    # Append singleton dimensions until this.dim() == other.dim()
    for _ in range(this.dim(), other.dim()):
        this = this.unsqueeze(-1)

    # Explicitly expand so that shapes are the same
    return this.expand_as(other)
```

❖ به منظور درک بهتر عملکرد این تابع به شرح یک مثال از آن می‌پردازیم:



فرض کنیم سه راس و دو هد داریم و یال‌های بین رئوس به شرح زیر است:

```
edge_src tensor([0, 1, 1, 2, 2], dtype=torch.int32)
edge_trg tensor([0, 1, 0, 1, 2], dtype=torch.int32)
```

بنابراین ماتریس امتیازهای نمایی طبق یال‌ها دارای ابعاد *(تعداد هدها، تعداد یال‌ها)* یعنی *(۲، ۵)* و به عنوان مثال به شرح زیر خواهد بود:

```
exp_scores_per_edge:
tensor([[9.8566e-01, 9.7492e-01],
        [6.2299e-04, 4.5232e-01],
        [9.3425e-01, 4.3607e-01],
        [3.4102e-01, 2.7981e-01],
        [6.9603e-01, 6.3759e-02]])
```

در گام اول ابعاد بردار شاخص هدف را بسط داده و آن را به ماتریسی با ابعاد *(تعداد هدها، تعداد یال‌ها)* یعنی *(۲، ۵)* تبدیل می‌کنیم.

```
trg_index_broadcasted:
tensor([[0, 0],
        [1, 1],
        [0, 0],
        [1, 1],
        [2, 2]])
```

از آنجا که قصد داریم به ازای هر راس، جمع امتیاز رئوس همسایهٔ آن را محاسبه کنیم، ماتریس حاصل دارای ابعاد *(تعداد هدها، تعداد راس‌ها)* یعنی *(۲، ۳)* خواهد بود. بنابراین یک ماتریس اولیه با مقدار صفر و این ابعاد ایجاد می‌کنیم.

```
neighborhood_sums:
tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
```

اکنون می‌توان با توجه به ماتریس trg_index_broadcasted، نسبت به جمع امتیازهای همسایه‌های هر راس اقدام کرد. به عنوان مثال طبق این ماتریس یال‌های اول و سوم به راس شماره صفر متصل هستند، بنابراین جمع مقادیر امتیاز آن‌ها که برابر 1.9199 برای هد اول و 1.4110 برای هد دوم است را در ردیف مربوط به راس شماره صفر (ردیف نخست) در ماتریس neighborhood_sums ذخیره خواهیم کرد. پس از تکمیل این ماتریس به طریق مشابه، خواهیم داشت:

```
neighborhood_sums:
tensor([[1.9199, 1.4110],
        [0.3416, 0.7321],
        [0.6960, 0.0638]])
```

اکنون از آنجا که قصد داریم ماتریس نهایی دارای ابعاد *(تعداد هدها، تعداد یال‌ها)* یعنی *(۲، ۵)* باشد، طبق trg_index_broadcasted هرجا که به راسی اشاره شده، مجموع امتیاز آن را طبق نتایج ذخیره شده در neighborhood_sums خواهیم نوشت. خروجی نهایی برابر است با:

```
tensor([[1.9199, 1.4110],
        [0.3416, 0.7321],
        [1.9199, 1.4110],
        [0.3416, 0.7321],
        [0.6960, 0.0638]])
```

۳-۱) تجمیع همسایه‌ها

بعد از به دست آوردن ضرایب توجه نرمال شده، جهت محاسبه‌ی بردار ویژگی جدید هر گره یا همان بردار ویژگی سطح بالاتر، این ضرایب را در بردار ویژگی تبدیل‌یافته ضرب می‌کنیم. خروجی نهایی این مرحله دارای ابعاد *(اندازه ویژگی خروجی، تعداد هدها، تعداد یال‌ها)* خواهد بود.

$$\vec{h}'_i = \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}\mathbf{W}\vec{h}_j\right) \quad \vec{h}'_i = \Big\Vert_{k=1}^{K} \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}^{k}\mathbf{W}^{k}\vec{h}_j\right) \quad \vec{h}'_i = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}_i} \alpha_{ij}^{k}\mathbf{W}^{k}\vec{h}_j\right)$$

```
#
# Step 3: Neighborhood aggregation
#

# Element-wise (aka Hadamard) product. Operator * does the same thing as torch.mul
# shape = (E, NH, FOUT) * (E, NH, 1) -> (E, NH, FOUT), 1 gets broadcast into FOUT
nodes_features_proj_lifted_weighted = nodes_features_proj_lifted * attentions_per_edge
```

در آخرین قسمت از این مرحله برای به دست آوردن بردار ویژگی جدید هر گره، بردار ویژگی تبدیل یافته‌ی وزن‌دار همسایگان آن را با هم جمع می‌کنیم که توسط تابع aggregate_neighbors انجام شده است.

$$\vec{h}'_i = \Big\Vert_{k=1}^{K} \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}^{k}\mathbf{W}^{k}\vec{h}_j\right) \quad \vec{h}'_i = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}_i} \alpha_{ij}^{k}\mathbf{W}^{k}\vec{h}_j\right) \quad \vec{h}'_i = \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}\mathbf{W}\vec{h}_j\right)$$

```
# This part sums up weighted and projected neighborhood feature vectors for every target node
# shape = (N, NH, FOUT)
out_nodes_features = self.aggregate_neighbors(nodes_features_proj_lifted_weighted, edge_index, in_nodes_features, num_of_nodes)
```

پیاده‌سازی تابع aggregate_neighbors به صورت زیر بوده و عملکردی مشابه با تابع sum_edge_scores_neighborhood_aware دارد که پیش از این معرفی شد. خروجی نهایی این تابع دارای ابعادی برابر با *(اندازه ویژگی خروجی، تعداد هدها، تعداد راس‌ها)* خواهد بود.

```
def aggregate_neighbors(self, nodes_features_proj_lifted_weighted, edge_index, in_nodes_features, num_of_nodes):
    size = list(nodes_features_proj_lifted_weighted.shape)  # convert to list otherwise assignment is not possible
    size[self.nodes_dim] = num_of_nodes  # shape = (N, NH, FOUT)
    out_nodes_features = torch.zeros(size, dtype=in_nodes_features.dtype, device=in_nodes_features.device)

    # shape = (E) -> (E, NH, FOUT)
    trg_index_broadcasted = self.explicit_broadcast(edge_index[self.trg_nodes_dim], nodes_features_proj_lifted_weighted)
    # aggregation step - we accumulate projected, weighted node features for all the attention heads
    # shape = (E, NH, FOUT) -> (N, NH, FOUT)
    out_nodes_features.scatter_add_(self.nodes_dim, trg_index_broadcasted, nodes_features_proj_lifted_weighted)

    return out_nodes_features
```

۴-۱) بایاس، الحاق و skip_connection

```
#
# Step 4: Residual/skip connections, concat and bias
#

out_nodes_features = self.skip_concat_bias(attentions_per_edge, in_nodes_features, out_nodes_features)
return (out_nodes_features, edge_index)
```

در این مرحله با توجه به متغیرهای تنظیم شده درباره سه مورد زیر تصمیم گرفته و در صورت نیاز آن‌ها را اعمال می‌کنیم:

- آیا از تکینک skip connection استفاده کنیم؟
- خروجی هدها را با هم الحاق کنیم یا از آن‌ها میانگین بگیریم؟
- از bias در لایهٔ GAT استفاده کنیم یا خیر؟

پس از طی این مراحل، خروجی نهایی لایهٔ GAT تولید خواهد شد.

$$\vec{h}_i' = \Big\Vert_{k=1}^{K} \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k \vec{h}_j\right) \quad \vec{h}_i' = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k \vec{h}_j\right)$$

استفاده یا عدم استفاده از log_attention_weights مربوط به مصورسازی است و ضرایب توجه را جهت استفاده بصری از آن‌ها ذخیره می‌کند.

استفاده یا عدم استفاده از skip connection، بایاس و همچنین استفاده از الحاق برای خروجی هدها یا میانگین‌گیری از آنها همگی در ابتدا توضیح داده شده‌اند و فقط پیاده‌سازی آن‌ها در تابع زیر انجام شده است. در صورتیکه از تکنیک skip connection استفاده شود ویژگی‌های ورودی را با نتایج بدست آمده توسط لایه جمع خواهیم کرد، که البته ممکن است برای این کار نیاز به تغییر ابعاد ورودی داشته باشیم.

```python
def skip_concat_bias(self, attention_coefficients, in_nodes_features, out_nodes_features):
    if self.log_attention_weights:  # potentially log for later visualization in playground.py
        self.attention_weights = attention_coefficients

    if self.add_skip_connection:  # add skip or residual connection
        if out_nodes_features.shape[-1] == in_nodes_features.shape[-1]:  # if FIN == FOUT
            # unsqueeze does this: (N, FIN) -> (N, 1, FIN), out features are (N, NH, FOUT) so 1 gets broadcast to NH
            # thus we're basically copying input vectors NH times and adding to processed vectors
            out_nodes_features += in_nodes_features.unsqueeze(1)
        else:
            # FIN != FOUT so we need to project input feature vectors into dimension that can be added to output
            # feature vectors. skip_proj adds lots of additional capacity which may cause overfitting.
            out_nodes_features += self.skip_proj(in_nodes_features).view(-1, self.num_of_heads, self.num_out_features)

    if self.concat:
        # shape = (N, NH, FOUT) -> (N, NH*FOUT)
        out_nodes_features = out_nodes_features.view(-1, self.num_of_heads * self.num_out_features)
    else:
        # shape = (N, NH, FOUT) -> (N, FOUT)
        out_nodes_features = out_nodes_features.mean(dim=self.head_dim)

    if self.bias is not None:
        out_nodes_features += self.bias

    return out_nodes_features if self.activation is None else self.activation(out_nodes_features)
```

۲) مدل GAT

```python
class GAT(torch.nn.Module):
```

برای پیاده‌سازی مدل با استفاده از چندین لایهٔ GAT، یک کلاس با همین نام تعریف شده که از ویژگی‌های یک مدل تورچ بهره می‌برد. در تعریف این مدل علاوه بر پارامترهای لایه‌های GAT پارامتری به نام `num_of_layers` برای مشخص کردن تعداد لایه‌های GAT تعریف شده است. تعداد لایه‌های GAT به مشخصات گراف بستگی دارد و با افزایش تعداد لایه‌ها، هر راس می‌تواند تحت تاثیر راس‌های دورتری در گراف نیز قرار بگیرد.

```python
def __init__(self, num_of_layers, num_heads_per_layer, num_features_per_layer, add_skip_connection=True, bias=True,
             dropout=0.6, log_attention_weights=False):
    super().__init__()
    assert num_of_layers == len(num_heads_per_layer) == len(num_features_per_layer) - 1, f'Enter valid arch params.'

    num_heads_per_layer = [1] + num_heads_per_layer  # trick - so that I can nicely create GAT layers below

    gat_layers = []  # collect GAT layers
    for i in range(num_of_layers):
        layer = GATLayer(
            num_in_features=num_features_per_layer[i] * num_heads_per_layer[i],  # consequence of concatenation
            num_out_features=num_features_per_layer[i+1],
            num_of_heads=num_heads_per_layer[i+1],
            concat=True if i < num_of_layers - 1 else False,  # last GAT layer does mean avg, the others do concat
            activation=nn.ELU() if i < num_of_layers - 1 else None,  # last layer just outputs raw scores
            dropout_prob=dropout,
            add_skip_connection=add_skip_connection,
            bias=bias,
            log_attention_weights=log_attention_weights
        )
        gat_layers.append(layer)

    self.gat_net = nn.Sequential(
        *gat_layers,
    )
```

تعداد ویژگی‌های ورودی هر لایه برابر حاصل‌ضرب تعداد خروجی لایه قبل و تعداد هدهای آن لایه است. در تمام لایه‌ها به جز لایه‌ی آخر خروجی هدها با هم concat می‌شوند و در لایه‌ی آخر میانگین آن‌ها حساب می‌شود. برای تمام لایه‌ها به جز لایه‌ی آخر از تابع فعالسازی ELU استفاده شده است و در لایه‌ی آخر هیچ نوع تابع فعالسازی را اعمال نکردیم تا امتیاز نهایی تولید شود. در نهایت مدلی Sequential با کنار هم قرار گرفتن لایه‌ها ساخته می‌شود.

```
# data is just a (in_nodes_features, edge_index) tuple, I had to do it like this because of the nn.Sequential:
# https://discuss.pytorch.org/t/forward-takes-2-positional-arguments-but-3-were-given-for-nn-sqeuential-with-linear-layers/65698
def forward(self, data):
    return self.gat_net(data)
```

به وسیله‌ی تابع فوق مدل ساخته شده روی داده‌ی ورودی (یعنی گراف) اعمال می‌شود و خروجی لایه آخر به حل مسئله (مثلا طبقه‌بندی) کمک می‌کند.

```
def get_training_args():
    parser = argparse.ArgumentParser()

    # Training related
    parser.add_argument("--num_of_epochs", type=int, help="number of training epochs", default=10000)
    parser.add_argument("--patience_period", type=int, help="number of epochs with no improvement on val before terminating", default=1000)
    parser.add_argument("--lr", type=float, help="model learning rate", default=5e-3)
    parser.add_argument("--weight_decay", type=float, help="L2 regularization on model weights", default=5e-4)
    parser.add_argument("--should_test", type=bool, help='should test the model on the test dataset?', default=True)

    # Dataset related
    parser.add_argument("--dataset_name", choices=[el.name for el in DatasetType], help='dataset to use for training', default=DatasetType.CORA.name)
    parser.add_argument("--should_visualize", type=bool, help='should visualize the dataset?', default=False)

    # Logging/debugging/checkpoint related (helps a lot with experimentation)
    parser.add_argument("--enable_tensorboard", type=bool, help="enable tensorboard logging", default=False)
    parser.add_argument("--console_log_freq", type=int, help="log to output console (epoch) freq (None for no logging)", default=100)
    parser.add_argument("--checkpoint_freq", type=int, help="checkpoint model saving (epoch) freq (None for no logging)", default=1000)
    args = parser.parse_args("")
```

```
# Model architecture related - this is the architecture as defined in the official paper (for Cora classification)
gat_config = {
    "num_of_layers": 2,  # GNNs, contrary to CNNs, are often shallow (it ultimately depends on the graph properties)
    "num_heads_per_layer": [8, 1],
    "num_features_per_layer": [CORA_NUM_INPUT_FEATURES, 8, CORA_NUM_CLASSES],
    "add_skip_connection": False,  # hurts perf on Cora
    "bias": True,  # result is not so sensitive to bias
    "dropout": 0.6,  # result is sensitive to dropout
}

# Wrapping training configuration into a dictionary
training_config = dict()
for arg in vars(args):
    training_config[arg] = getattr(args, arg)

# Add additional config information
training_config.update(gat_config)

return training_config
```

در تابع `get_training_args` پارامترهای (هایپرپارامترهای) آموزش مثل تعداد دورها[3]، نرخ آموزشی[4]، زمان صبر[5]، رگولاریزیشن-L2[6] و پارامترهای مدل مقداردهی شده‌اند. این پارامترها به عنوان کانفیگ اولیه به تابع `train_gat` داده شدند.

[3] Epochs
[4] Learning rate
[5] Patience period
[6] L2-regularization

در تابع `train_gat` علاوه بر کنترل دسترسی به CPU یا GPU و بارگذاری دیتا، پارامترها و لایه GAT بر روی آن‌ها، در تنظیمات Transductive از تابع ضرر Crossentropy و در تنظیمات Inductive از تابع ضرر BCEWithLogitsLoss استفاده شده است. در ضمن در هر دو تنظیمات بهینه‌ساز Adam برای آموزش انتخاب شده‌است. همچنین مطابق با آنچه در مقاله گزارش داده شده برای تنظیمات Transductive از معیار Accuracy و تنظیمات Inductive از معیار F1-score استفاده خواهد شد.

```python
def train_gat(config):
    global BEST_VAL_ACC, BEST_VAL_LOSS

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  # checking whether you have a GPU, I hope so!

    # Step 1: load the graph data
    node_features, node_labels, edge_index, train_indices, val_indices, test_indices = load_graph_data(config, device)

    # Step 2: prepare the model
    gat = GAT(
        num_of_layers=config['num_of_layers'],
        num_heads_per_layer=config['num_heads_per_layer'],
        num_features_per_layer=config['num_features_per_layer'],
        add_skip_connection=config['add_skip_connection'],
        bias=config['bias'],
        dropout=config['dropout'],
        log_attention_weights=False  # no need to store attentions, used only in playground.py while visualizing
    ).to(device)

    # Step 3: Prepare other training related utilities (loss & optimizer and decorator function)
    loss_fn = nn.CrossEntropyLoss(reduction='mean')
    optimizer = Adam(gat.parameters(), lr=config['lr'], weight_decay=config['weight_decay'])
```

```python
def train_gat_ppi(config):
    """
    Very similar to Cora's training script. The main differences are:
    1. Using dataloaders since we're dealing with an inductive setting - multiple graphs per batch
    2. Doing multi-class classification (BCEWithLogitsLoss) and reporting micro-F1 instead of accuracy
    3. Model architecture and hyperparams are a bit different (as reported in the GAT paper)

    """
    global BEST_VAL_MICRO_F1, BEST_VAL_LOSS

    # Checking whether you have a strong GPU. Since PPI training requires almost 8 GBs of VRAM
    # I've added the option to force the use of CPU even though you have a GPU on your system (but it's too weak).
    device = torch.device("cuda" if torch.cuda.is_available() and not config['force_cpu'] else "cpu")

    # Step 1: prepare the data loaders
    data_loader_train, data_loader_val, data_loader_test = load_graph_data(config, device)

    # Step 2: prepare the model
    gat = GAT(
        num_of_layers=config['num_of_layers'],
        num_heads_per_layer=config['num_heads_per_layer'],
        num_features_per_layer=config['num_features_per_layer'],
        add_skip_connection=config['add_skip_connection'],
        bias=config['bias'],
        dropout=config['dropout'],
        log_attention_weights=False  # no need to store attentions, used only in playground.py for visualizations
    ).to(device)

    # Step 3: Prepare other training related utilities (loss & optimizer and decorator function)
    loss_fn = nn.BCEWithLogitsLoss(reduction='mean')
    optimizer = Adam(gat.parameters(), lr=config['lr'], weight_decay=config['weight_decay'])
```

در تابع `train_gat` و با استفاده از تابع `get_main_loop` مدل آموزش می‌بیند و عملکرد آن روی داده‌های آموزش، اعتبارسنجی و آزمون محاسبه می‌شود. به ازای هر دور، یکبار تابع `main_loop` برای آموزش مدل، و یکبار هم در فاز اعتبارسنجی فراخوانی خواهد شد. وظیفهٔ این تابع، آموزش یا اعتبارسنجی مدل GAT و محاسبهٔ هزینه براساس امتیازات خروجی مدل است.

```
# THIS IS THE CORE OF THE TRAINING (we'll define it in a minute)
# The decorator function makes things cleaner since there is a lot of redundancy between the train and val loops
main_loop = get_main_loop(
    config,
    gat,
    loss_fn,
    optimizer,
    node_features,
    node_labels,
    edge_index,
    train_indices,
    val_indices,
    test_indices,
    config['patience_period'],
    time.time())

BEST_VAL_ACC, BEST_VAL_LOSS, PATIENCE_CNT = [0, 0, 0]  # reset vars used for early stopping
```

```
# Step 4: Start the training procedure
for epoch in range(config['num_of_epochs']):
    # Training loop
    main_loop(phase=LoopPhase.TRAIN, epoch=epoch)

    # Validation loop
    with torch.no_grad():
        try:
            main_loop(phase=LoopPhase.VAL, epoch=epoch)
        except Exception as e:  # "patience has run out" exception :O
            print(str(e))
            break  # break out from the training loop

# Step 5: Potentially test your model
# Don't overfit to the test dataset - only when you've fine-tuned your model on the validation dataset should you
# report your final loss and accuracy on the test dataset. Friends don't let friends overfit to the test data. <3
if config['should_test']:
    test_acc = main_loop(phase=LoopPhase.TEST)
    config['test_acc'] = test_acc
    print(f'Test accuracy = {test_acc}')
else:
    config['test_acc'] = -1

# Save the latest GAT in the binaries directory
torch.save(get_training_state(config, gat), os.path.join(BINARIES_PATH, get_available_binary_name()))
```

```
def main_loop(phase, epoch=0):
    global BEST_VAL_ACC, BEST_VAL_LOSS, PATIENCE_CNT, writer

    # Certain modules behave differently depending on whether we're training the model or not.
    # e.g. nn.Dropout - we only want to drop model weights during the training.
    if phase == LoopPhase.TRAIN:
        gat.train()
    else:
        gat.eval()

    node_indices = get_node_indices(phase)
    gt_node_labels = get_node_labels(phase)  # gt stands for ground truth

    # Do a forwards pass and extract only the relevant node scores (train/val or test ones)
    # Note: [0] just extracts the node_features part of the data (index 1 contains the edge_index)
    # shape = (N, C) where N is the number of nodes in the split (train/val/test) and C is the number of classes
    nodes_unnormalized_scores = gat(graph_data)[0].index_select(node_dim, node_indices)

    # Example: let's take an output for a single node on Cora - it's a vector of size 7 and it contains unnormalized
    # scores like: V = [-1.393,  3.0765, -2.4445,  9.6219,  2.1658, -5.5243, -4.6247]
    # What PyTorch's cross entropy loss does is for every such vector it first applies a softmax, and so we'll
    # have the V transformed into: [1.6421e-05, 1.4338e-03, 5.7378e-06, 0.99797, 5.7673e-04, 2.6376e-07, 6.4848e-07]
    # secondly, whatever the correct class is (say it's 3), it will then take the element at position 3,
    # 0.99797 in this case, and the loss will be -log(0.99797). It does this for every node and applies a mean.
    # You can see that as the probability of the correct class for most nodes approaches 1 we get to 0 loss! <3
    loss = cross_entropy_loss(nodes_unnormalized_scores, gt_node_labels)
```

همانطور که گفته شد خروجی نهایی مدل GAT به شکل امتیازهای نرمال‌نشده است. این امتیازها به تابع ضرر داده شده و هزینهٔ نهایی محاسبه خواهد شد. همچنین در تنظیمات Transductive از بین امتیازهایی که برای هر راس بدست آمده، شاخص مربوط به امتیاز بیشینه برابر کلاس پیش‌بینی شده برای آن راس خواهد بود. در تنظیمات Inductive نیز از آنجا که از تابع هزینه Sigmoid استفاده شده است، اگر امتیاز خروجی بیشتر از ۱ باشد یعنی که Sigmoid ارزش بیشتر از ۰.۵ را داشته و کلاس مربوطه، کلاس ۱ و در غیر اینصورت کلاس ۰ خواهد بود.

```
if phase == LoopPhase.TRAIN:
    optimizer.zero_grad()  # clean the trainable weights gradients in the computational graph (.grad fields)
    loss.backward()  # compute the gradients for every trainable weight in the computational graph
    optimizer.step()  # apply the gradients to weights

# Finds the index of maximum (unnormalized) score for every node and that's the class prediction for that node.
# Compare those to true (ground truth) labels and find the fraction of correct predictions -> accuracy metric.
class_predictions = torch.argmax(nodes_unnormalized_scores, dim=-1)
accuracy = torch.sum(torch.eq(class_predictions, gt_node_labels).long()).item() / len(gt_node_labels)


#
# Logging
#

if phase == LoopPhase.TRAIN:
    # Log metrics
    if config['enable_tensorboard']:
        writer.add_scalar('training_loss', loss.item(), epoch)
        writer.add_scalar('training_acc', accuracy, epoch)

    # Save model checkpoint
    if config['checkpoint_freq'] is not None and (epoch + 1) % config['checkpoint_freq'] == 0:
        ckpt_model_name = f"gat_ckpt_epoch_{epoch + 1}.pth"
        config['test_acc'] = -1
        torch.save(get_training_state(config, gat), os.path.join(CHECKPOINTS_PATH, ckpt_model_name))
```

```
elif phase == LoopPhase.VAL:
    # Log metrics
    if config['enable_tensorboard']:
        writer.add_scalar('val_loss', loss.item(), epoch)
        writer.add_scalar('val_acc', accuracy, epoch)

    # Log to console
    if config['console_log_freq'] is not None and epoch % config['console_log_freq'] == 0:
        print(f'GAT training: time elapsed= {(time.time() - time_start):.2f} [s] | epoch={epoch + 1} | val acc={accuracy}')
    # The "patience" logic - should we break out from the training loop? If either validation acc keeps going up
    # or the val loss keeps going down we won't stop
    if accuracy > BEST_VAL_ACC or loss.item() < BEST_VAL_LOSS:
        BEST_VAL_ACC = max(accuracy, BEST_VAL_ACC)  # keep track of the best validation accuracy so far
        BEST_VAL_LOSS = min(loss.item(), BEST_VAL_LOSS)
        PATIENCE_CNT = 0  # reset the counter every time we encounter new best accuracy
    else:
        PATIENCE_CNT += 1  # otherwise keep counting

    if PATIENCE_CNT >= patience_period:
        raise Exception('Stopping the training, the universe has no more patience for this training.')

else:
    return accuracy  # in the case of test phase we just report back the test accuracy
```

۳) مجموعه‌های داده

در مقالهٔ اصلی GAT [2] چهار مجموعه داده مورد آزمایش قرار گرفته که دو مورد از آن به عنوان نمونه‌هایی از رویکرد یادگیری Transductive و Inductive جهت بررسی صحت عملکرد مدل پیاده‌سازی شده مطابق با نتایج مقاله مورد ارزیابی قرار گرفته است. مشخصات این دو مجموعه داده به شرح زیر است:

۳-۱) Cora

مجموعه داده Cora شامل مقالات مربوط به یادگیری ماشین است. این مقالات به هفت کلاس طبقه بندی می‌شوند. در این مجموعه داده، گره‌ها به اسناد یا مقالات و یال‌ها به استناد[۷] (بدون جهت) مربوط هستند. ویژگی‌های گره مطابق با اجزای نمایش BOW[8] یک مقاله هستند. پس از تبدیل کلمات به ریشه‌ی آن‌ها[۹] و حذف کلمات توقف با یک مجموعه‌ی واژگان شامل ۱۴۳۳ کلمه منحصر به فرد مواجه هستیم. مقالات به گونه‌ای انتخاب شده‌اند که در پیکره نهایی هر مقاله حداقل یک مقاله دیگر به آن استناد کند یا از آن استناد شود. در کل پیکره ۲۷۰۸ مقاله وجود دارد.

هر گره دارای یک برچسب کلاس است. ما اجازه می‌دهیم فقط ۲۰ گره در هر کلاس برای آموزش استفاده شود - با این وجود ، طبق به تنظیمات Transductive، الگوریتم آموزش به بردار ویژگی همه گره‌ها دسترسی دارد.

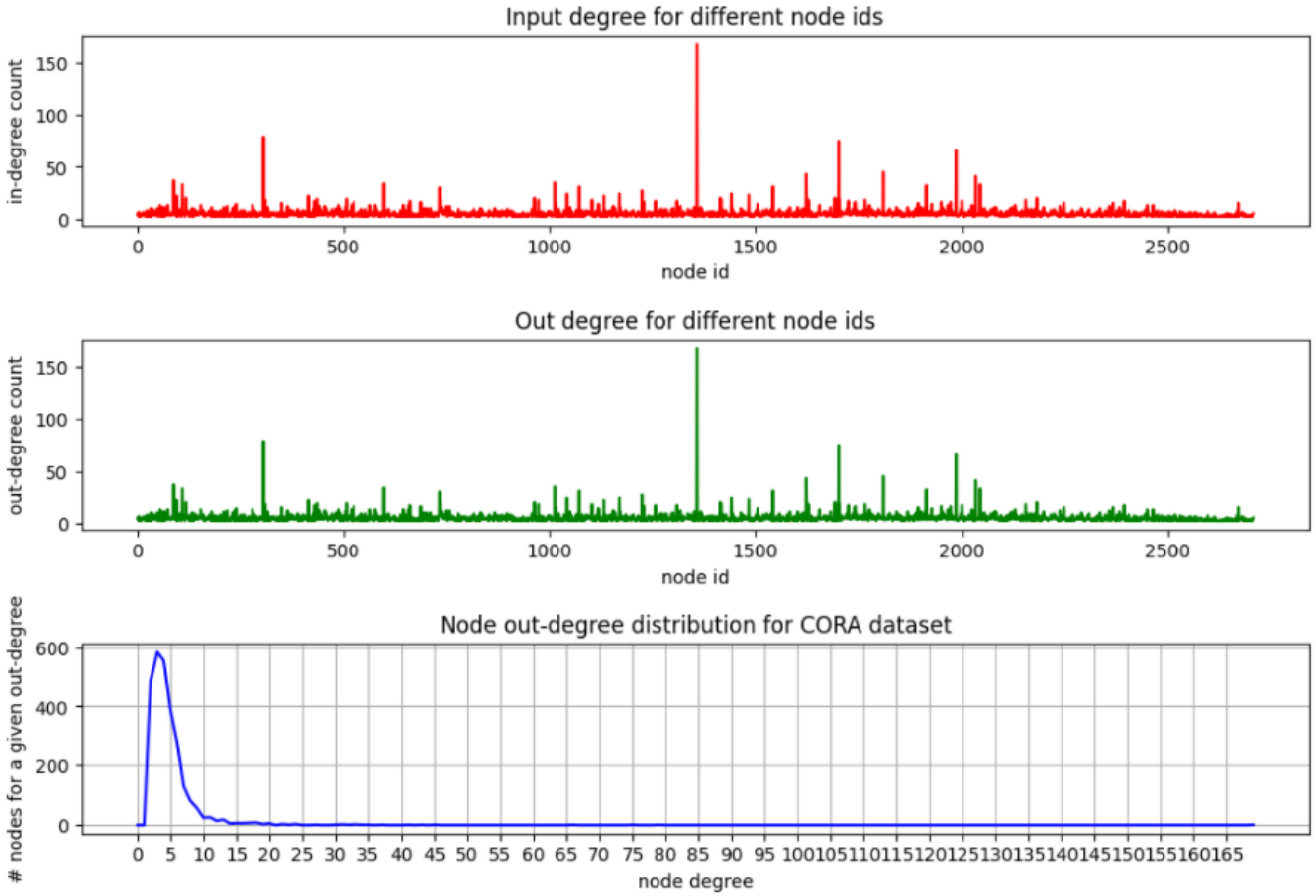
آمار کلی این مجموعه داده مطابق با جدول زیر است:

تعداد راس‌ها	تعداد یال‌ها	اندازهٔ ویژگی هر راس	نمونه‌های آموزشی	نمونه‌های اعتبارسنجی	نمونه‌های آزمون
۲۷۰۸	۵۴۲۹	۱۴۳۳	۱۴۰	۵۰۰	۱۰۰۰

نمودارهای میزان درجه ورودی هر راس، درجه خروجی هر راس و توزیع درجات خروجی در شکل ۱-۱-۳ نمایش داده شده است. همانطور که مشاهده می‌شود، دو نمودار اول مشابه با همدیگر بوده و این به دلیل بدون جهت بودن گراف است (با اینکه به طور طبیعی باید به عنوان یک

[7] Citation
[8] Bag-of-words
[9] Stemming

گراف جهت‌دار مدل شود). بعضی از گره‌ها دارای تعداد زیادی یال هستند (قله در وسط)، اما بیشتر گره ها دارای یال‌های بسیار کمتری هستند. در نمودار سوم نیز به خوبی قابل مشاهده است که رئوسی با درجات بسیار بالا، نادر هستند و پراکندگی در تعداد یال‌های کمتر، بیشتر است.



شکل ۱-۱-۳: نمودار جزئیات مجموعه داده Cora

۳-۲) Protein-Protein Interaction (PPI)

در مسئله‌ای که برای این مجموعه داده در نظر گرفته می‌شود بجای اینکه یادگیری به منظور کسب دانشی درباره ساختار جامعه انجام گیرد بر نقش راس‌ها در آن تاکید دارد، و هدف اصلی دستیابی به عمومیت[۱۰] بر روی گراف‌ها است. در اینجا قصد بر این است تا نقش پروتئین‌ها در گراف‌های مختلف تعاملات پروتئینی-پروتئینی (PPI)، که هر گراف نمایانگر یک بافت متفاوت انسانی است، رده‌بندی شود [3].

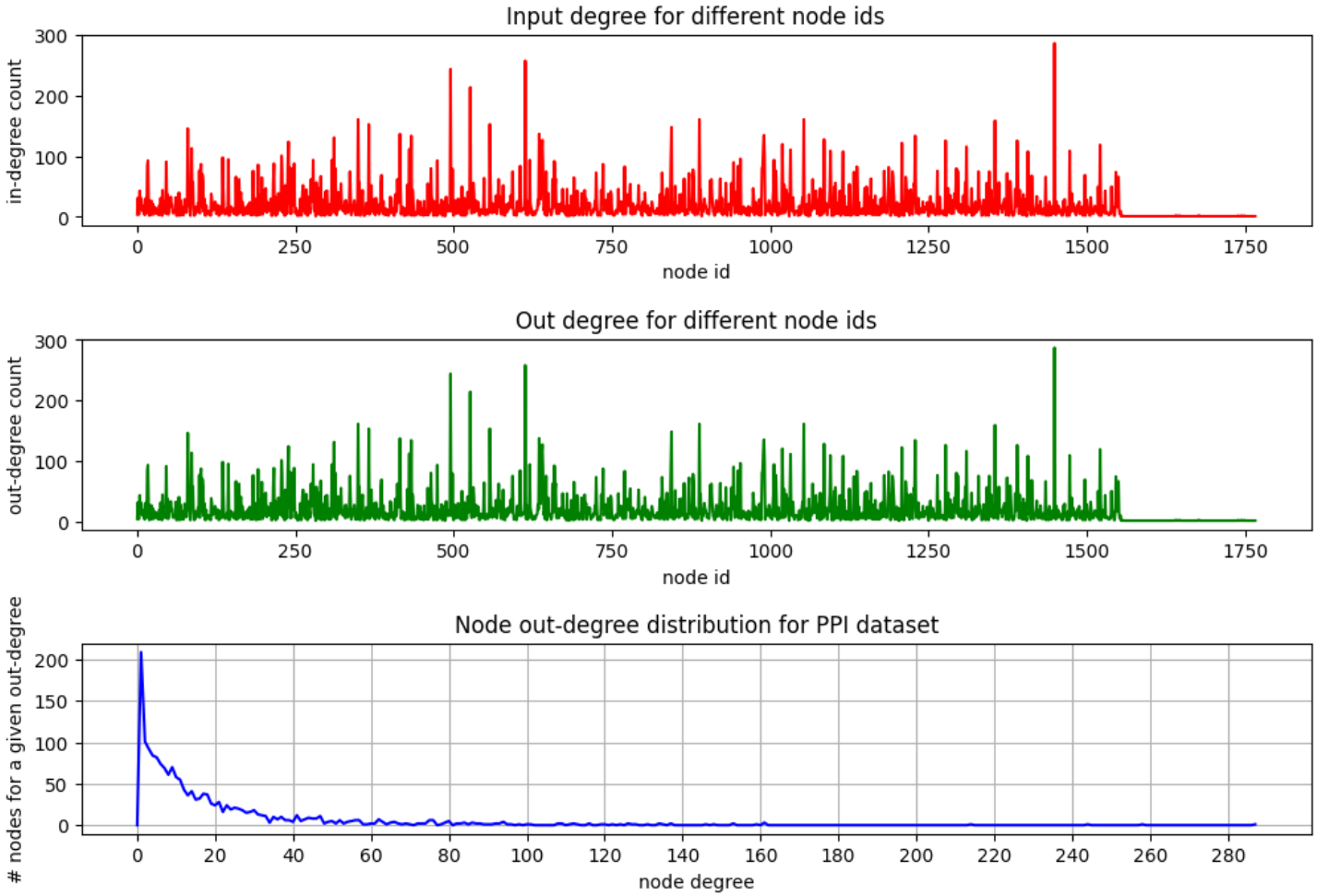
مجموعه ژن‌های موضعی[۱۱]، ژن‌های موتیف[۱۲] و امضاهای ایمونولوژیک[۱۳] به عنوان ویژگی و مجموعه آنتولوژی ژن‌ها[۱۴] به عنوان برچسب (مجموعاً ۱۲۱تا) در نظر گرفته شده است، که همگی از دیتابیس Molecular Signatures Database [4] استخراج شده‌اند.

[10] Generalization
[11] Positional gene sets
[12] Motif gene sets
[13] Immunological signatures
[14] Gene ontology sets

در مجموع ۲۴ گراف در این مجموعه داده موجود است که ۲۰ گراف آن به منظور آموزش مدل استفاده شده و میانگین امتیازات F1 نیز بر روی ۲ گراف آزمون گزارش شده است (۲ گراف دیگر نیز به منظور اعتبارسنجی مورداستفاده قرار گرفته است). میانگین درجه هر راس برابر با ۲۸.۸ است و به هر راس می‌توان از میان ۱۲۱ کلاس ممکن، چندین کلاس نیز نسبت داد (رده‌بندی چند-برچسبی[۱۵]).

تعداد راس‌ها	تعداد یال‌ها	اندازهٔ ویژگی هر راس	نمونه‌های آموزشی	نمونه‌های اعتبارسنجی	نمونه‌های آزمون
۵۶۹۴۴	۱۶۴۴۲۰۸	۵۰	۴۴۹۰۶	۶۵۱۴	۵۵۲۴

لازم به ذکر است که به دلیل حجم بالای داده در این مجموعه داده، گراف‌ها به شکل دسته‌های دوتایی به مدل داده شده و آموزش پیش خواهد رفت. در نمودارهای زیر، میزان درجات ورودی، درجات خروجی و توزیع آن‌ها برای یکی از گراف‌ها نمایش داده شده است. همانطور که از نمودار اول و دوم شکل ۱-۲-۳ مشاهده می‌شود بسیاری از راس‌ها در مقابله با مجموعه داده Cora دارای تعداد یال بیشتری هستند. همچنین طبق نمودار سوم می‌توان گفت که اکثر راس‌ها دارای ۱ تا ۲۰ یال هستند.



شکل ۱-۲-۳: نمودار جزئیات مجموعه داده PPI

[15] Multi-label classification

علاوه بر این، مدل پیاده‌سازی شده بر روی دو مجموعه داده جدید نیز مورد آزمایش قرار گرفته است که جزئیات هرکدام از آن‌ها به شرح زیر است:

Reddit (۳-۳

شبکه اجتماعی Reddit یکی از بزرگترین انجمن‌های اینترنتی است که کاربران به انتشار محتوا و نظرات خود پیرامون بسیاری از موضوعات در آن می‌پردازند. در مسئله‌ای که برای این شبکه مطرح می‌شود سعی بر این است تا پیش‌پینی کنیم که هر پست Reddit به کدام جامعه (موضوع) تعلق دارد.

در سال ۲۰۱۴ یک مجموعه داده گرافی از پست‌های این شبکه ساخته شد. برچسب‌های هر راس در این گراف، جامعه یا subreddit ای است که پست بدان تعلق دارد. گراف ساخته شده به شکل پست-به-پست بوده و دو پست در صورتی با یک یال به یکدیگر متصل هستند که کاربر مشابهی بر روی هر دوی آن‌ها دیدگاه خود را ثبت کرده باشد [5].

در مجموع این مجموعه داده شامل ۲۳۲۹۶۵ پست با میانگین درجه ۴۹۲ است که در مقایسه با سایر مجموعه داده‌های گرافی میزان درجه قابل توجهی به شمار می‌آید. داده‌های مربوط به ۲۰ روز نخست به عنوان مجموعهٔ آموزشی و سایر روزها به عنوان آزمایش (۳۰٪ آن به عنوان اعتبارسنجی) در نظر گرفته شده است.

بردار ویژگی هر راس مطابق با بردارهای کلمات ۳۰۰ بعدی GloVe CommonCrawl ساخته شده است. به ازای هر پست، ویژگی‌های زیر با یکدیگر الحاق شده‌اند و بردار نهایی را تشکیل داده‌اند:

- میانگین تعبیهٔ عنوان پست
- میانگین تعبیهٔ تمام دیدگاه‌های پست
- امتیاز پست
- تعداد دیدگاه‌های پست

آمار کلی این مجموعه داده مطابق با جدول زیر است:

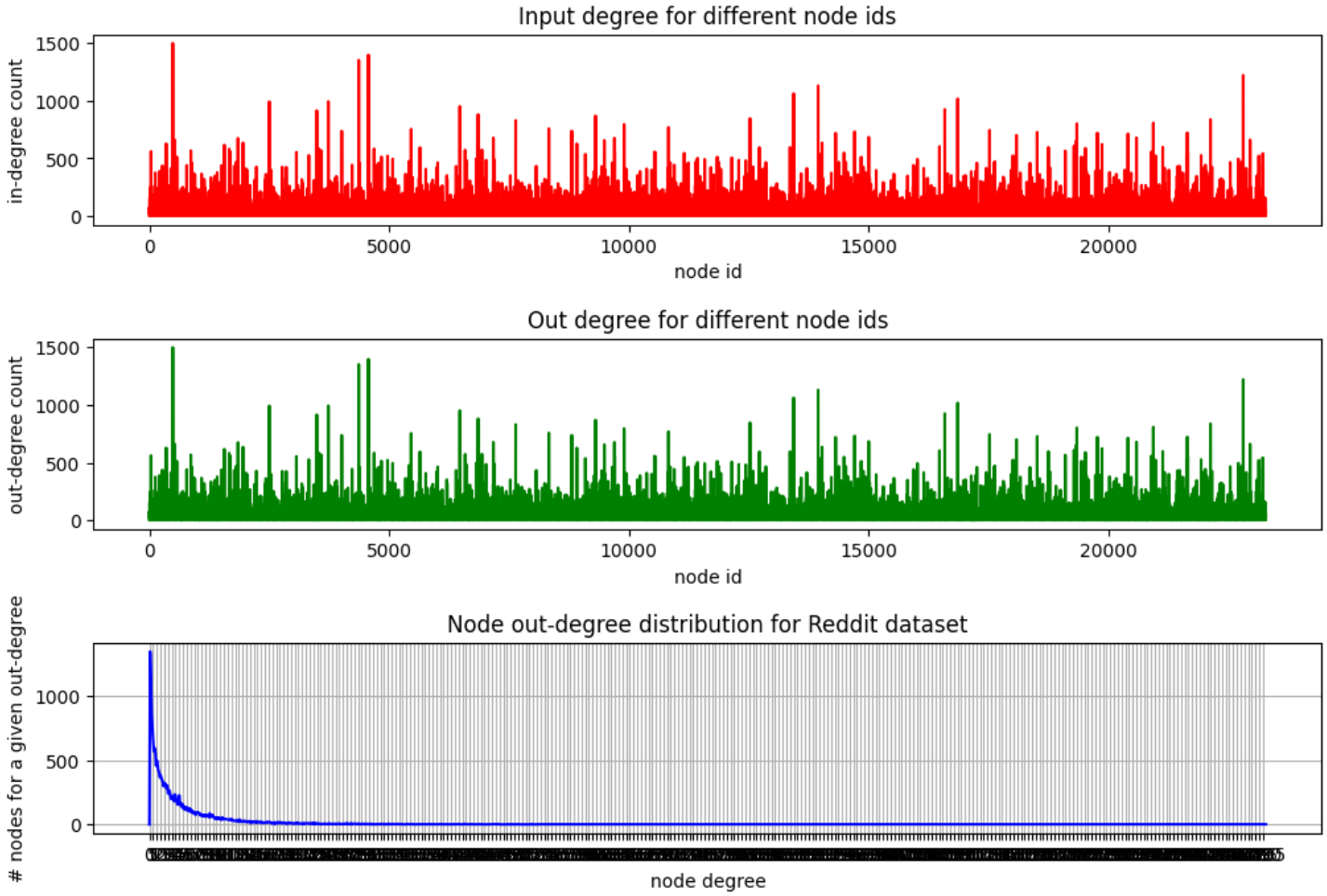
تعداد راس‌ها	تعداد یال‌ها	اندازهٔ ویژگی هر راس	نمونه‌های آموزشی	نمونه‌های اعتبارسنجی	نمونه‌های آزمون
۲۳۲۹۶۵	۱۱۴۶۱۵۸۹۲	۶۰۲	۱۵۳۴۳۱	۲۳۸۳۱	۵۵۷۰۳

با این حال، به دلیل اندازهٔ بسیار بزرگ مجموعهٔ داده و عدم دسترسی به سخت‌افزار کافی برای آموزش مدل بر روی این حجم از داده، در آزمایش‌های صورت گرفته تنها به ده درصد آن اکتفا شده و آمار آن به شرح موجود در جدول زیر است:

تعداد راس‌ها	تعداد یال‌ها	اندازهٔ ویژگی هر راس	نمونه‌های آموزشی	نمونه‌های اعتبارسنجی	نمونه‌های آزمون
۲۳۲۹۶	۱۰۵۲۵۶۰	۶۰۲	۱۵۳۴۳	۲۳۸۳	۵۵۷۰

باید توجه داشت که در برخی مقالات پیشین که بر روی این مجموعه داده نیز آزمایش خود را انجام داده‌اند از رویکرد یادگیری Inductive استفاده شده است. این بدین معنی است که در طول فرآیند آموزش مدل، رئوس مربوط به نمونه‌های آزمایشی به کلی نادیده گرفته می‌شوند. با این حال از آنجا که در این آزمایش مجموعه داده کوچک شده، آموزش مدل به شکل Transductive و مشابه با آنچه که بر روی مجموعه Cora اتفاق افتاد، انجام خواهد شد.

نمودارهای میزان درجه ورودی هر راس، درجه خروجی هر راس و توزیع درجات خروجی در شکل ۱-۳-۳ نمایش داده شده است. همانطور که مشاهده می‌شود، دو نمودار اول مشابه با همدیگر بوده و این به دلیل بدون جهت بدون گراف است. در نمودار سوم نیز به خوبی قابل مشاهده است که رئوسی با درجات بسیار بالا، نادر هستند و پراکندگی در تعداد یال‌های کمتر، بیشتر است.



شکل ۱-۳-۳: نمودار جزئیات مجموعه داده Reddit

۳-۴) Wiki_CS

مجموعه داده Wiki-CS که در پژوهش [6] معرفی شده است، یک مجموعه داده مبتنی بر ویکی‌پدیا برای آزمودن[16] شبکه های عصبی گرافی است. این مجموعه داده از دسته‌بندی‌های ویکی‌پدیا، به طور خاص ۱۰ کلاس مربوط به شاخه های علوم کامپیوتر[17] ، با اتصال بسیار بالا ساخته شده است. ویژگی‌های گره از متون مقاله‌های مربوطه گرفته شده است. این ویژگی‌ها به عنوان میانگین تعبیه کلمه از پیش آموزش دیده GloVe محاسبه شدند. در نتیجه ویژگی‌های گره ۳۰۰ بعدی حاصل می‌شود که می‌تواند برای آموزش مدل‌های بزرگ روی GPU یک مزیت باشد. مقالاتی که می‌توانستند به چند کلاس تعلق داشته باشند از این مجموعه داده حذف شده‌اند.

گره‌ها را در هر کلاس به دو مجموعه تقسیم می‌شوند، ۵۰٪ برای مجموعه آزمون و ۵۰٪ به طور بالقوه قابل مشاهده[18]. از مجموعه قابل مشاهده ، ۲۰ تقسیم مختلف برای آموزش ، اعتبارسنجی و توقف زودرس[19] ایجاد شده‌است: در هر تقسیم ۵٪ از گره‌های هر کلاس برای آموزش ،

[16] Benchmarking
[17] Computer Science
[18] Visible
[19] Early Stopping

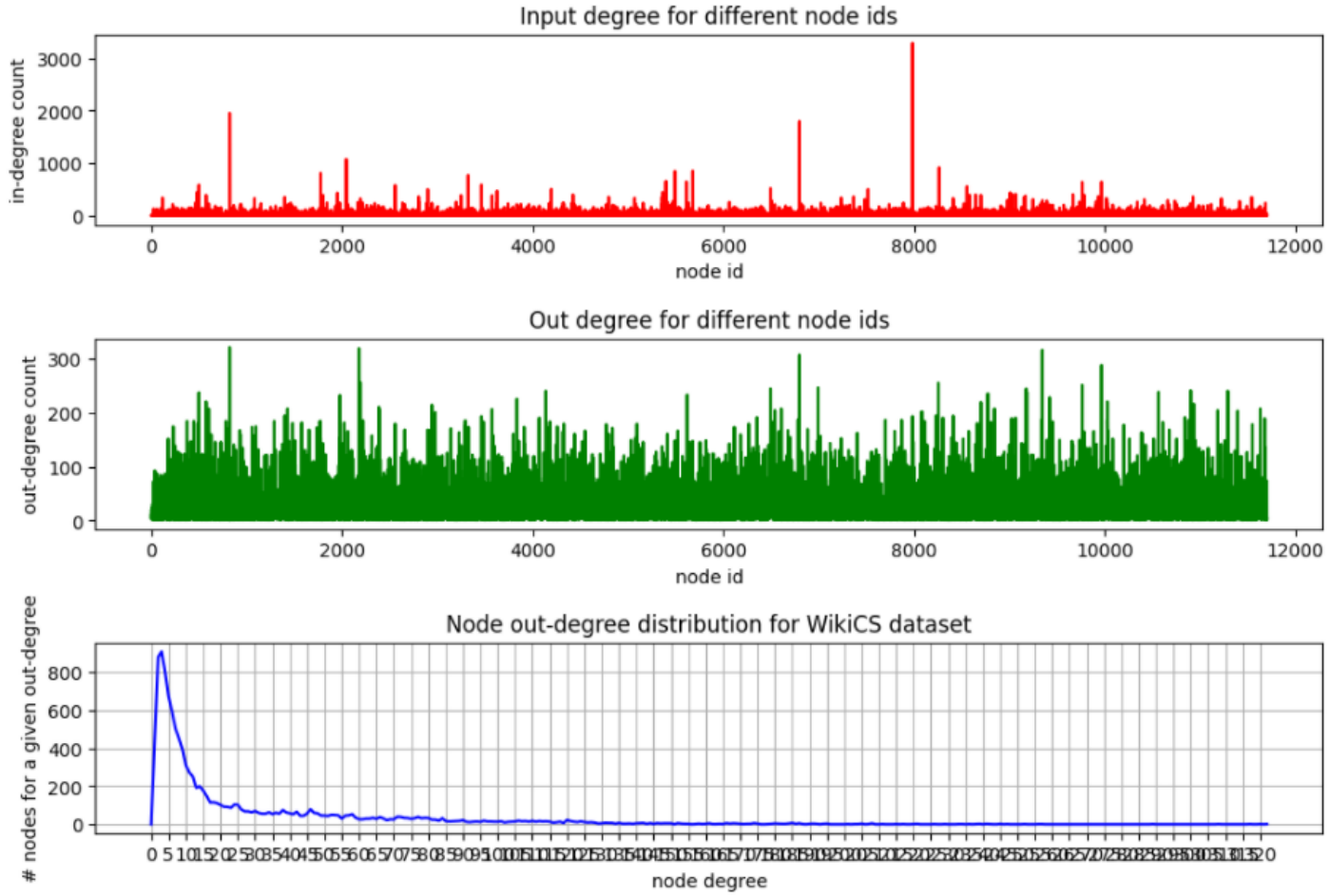
٪۲۲.۵ برای ارزیابی معیار توقف زودرس مورد استفاده قرار گرفت و ٪۲۲.۵ به عنوان مجموعه اعتبارسنجی برای تنظیم هایپر پارامتر استفاده شده‌است.

ما در آزمایشات خود فقط از یکی از ۲۰ تقسیم مختلف برای برای آموزش و اعتبارسنجی استفاده کرده و از داده‌های مربوط به توقف زودرس استفاده نمی‌کنیم.

آمار کلی این مجموعه داده مطابق با جدول زیر است:

تعداد راس‌ها	تعداد یال‌ها	اندازهٔ ویژگی هر راس	نمونه‌های آموزشی	نمونه‌های اعتبارسنجی	نمونه‌های آزمون
۱۱۷۰۱	۲۱۶۱۲۳	۳۰۰	۵۸۰	۱۷۶۹	۵۸۴۷

نمودارهای میزان درجه ورودی هر راس، درجه خروجی هر راس و توزیع درجات خروجی در شکل ۱-۴-۳ نمایش داده شده است. همانطور که مشاهده می‌شود، نمودار اول و دوم متفاوت هستند و این به دلیل جهت‌دار بودن گراف مربوطه است. در نمودار سوم نیز به خوبی قابل مشاهده است که رئوسی با درجات بسیار بالا، نادر هستند و پراکندگی در تعداد یال‌های کمتر، بیشتر است.



شکل ۱-۴-۳: نمودار جزئیات مجموعه داده Wiki_CS

۴) آزمایش‌ها

هایپرپارامترهایی که برای آموزش مدل بر روی دو مجموعه داده اصلی Cora و PPI تنظیم شده دقیقاً مطابق با مقاله بوده و برای دو مجموعه داده دیگر نیز تقریباً مشابه با آن‌ها، اما با اندکی تغییر جهت عملکرد بهتر، در نظر گرفته شده‌اند. پارامترهای اصلی تنظیم شده برای هر چهار مجموعه داده در جدول ۱-۴ خلاصه شده است.

لازم به ذکر است بردار اندازهٔ ویژگی هر لایه به شکل [تعداد کلاس‌ها، ...، تعداد ویژگی‌های ورودی] مقداردهی می‌شود. به عنوان مثال وقتی این بردار به شکل [602, 8, 41] تنظیم شده است بدین معنی است که لایهٔ نخست، بردارهای ویژگی ورودی با اندازه ۶۰۲ را گرفته و بردارهای ویژگی جدیدی با ابعاد ۸ تولید می‌کند. سپس لایهٔ دوم نیز بردارهای ابعاد ۸ را گرفته و برداری با ابعاد ۴۱ که برابر تعداد کلاس‌های مسئله هست را تولید خواهد کرد.

	Cora	Wiki_CS	Reddit	PPI
تعداد لایه‌ها	2	2	2	3
تعداد هد به ازای هر لایه	[8, 1]	[8, 1]	[8, 1]	[4, 4, 6]
اندازه ویژگی هر لایه	[1433, 8, 7]	[300, 8, 10]	[602, 8, 41]	[50, 64, 64, 121]
Skip connection	False	False	False	True
بایاس	True	True	True	True
نرخ حذف تصادفی	0.6	0.3	0.3	0.0
بیشینه تعداد دورها	10000	10000	10000	200
زمان انتظار	1000	1000	1000	100
نرخ یادگیری	5e-3	5e-3	5e-3	5e-3
کاهش وزن	5e-4	5e-4	5e-4	0

جدول ۱-۴: هایپرپارامترهای تنظیم شده براساس هر مجموعه داده

۵) نتایج

پس از آموزش مدل با توجه به هایپرپارامترهای معرفی شده در بخش پیشین، نتایج حاصل با رویکرد Transductive در جدول ۱-۵ و با رویکرد Inductive در جدول ۲-۵ آورده شده است. با مقایسهٔ نتایج بدست آمده بر روی دو مجموعه داده Cora و PPI با نتایج مقالهٔ اصلی در جدول ۳-۵ و ۴-۵ می‌توان به صحت عملکرد مدل پیاده‌سازی شده پی برد. علاوه بر این، در مقایسه عملکرد مدل آموزش داده شده بر روی مجموعه داده Wiki_CS با نتایجی که مقالهٔ اصلی آن منتشر کرده (جدول ۵-۵)، بهبود حدود ۰.۶ درصدی دقت آزمون نیز مشاهده می‌شود.

مجموعه داده	آخرین دور	زمان (ثانیه)	دقت اعتبارسنجی	دقت آزمون
Cora	1101	127.08	81.60	82.20
Wiki_CS	1501	24.30	79.31	78.24
Reddit	5701	392.49	92.15	90.95

جدول ۱-۵: نتایج آموزش مدل GAT با رویکرد Transductive

آخرین دور	زمان (ثانیه)	F1 آموزش	F1 اعتبارسنجی	F1 آزمون
191	392.58	98.85	96.62	98.02

جدول ۲-۵: نتایج آموزش مدل GAT با رویکرد Inductive

Transductive

Method	Cora	Citeseer	Pubmed
MLP	55.1%	46.5%	71.4%
ManiReg (Belkin et al., 2006)	59.5%	60.1%	70.7%
SemiEmb (Weston et al., 2012)	59.0%	59.6%	71.7%
LP (Zhu et al., 2003)	68.0%	45.3%	63.0%
DeepWalk (Perozzi et al., 2014)	67.2%	43.2%	65.3%
ICA (Lu & Getoor, 2003)	75.1%	69.1%	73.9%
Planetoid (Yang et al., 2016)	75.7%	64.7%	77.2%
Chebyshev (Defferrard et al., 2016)	81.2%	69.8%	74.4%
GCN (Kipf & Welling, 2017)	81.5%	70.3%	**79.0%**
MoNet (Monti et al., 2016)	$81.7 \pm 0.5\%$	—	$78.8 \pm 0.3\%$
GCN-64*	$81.4 \pm 0.5\%$	$70.9 \pm 0.5\%$	$\mathbf{79.0} \pm 0.3\%$
GAT (ours)	$\mathbf{83.0} \pm 0.7\%$	$\mathbf{72.5} \pm 0.7\%$	$\mathbf{79.0} \pm 0.3\%$

جدول ۳-۵: نتایج آموزش مدل GAT با رویکرد Transductive برگرفته از مقالهٔ اصلی

Inductive

Method	PPI
Random	0.396
MLP	0.422
GraphSAGE-GCN (Hamilton et al., 2017)	0.500
GraphSAGE-mean (Hamilton et al., 2017)	0.598
GraphSAGE-LSTM (Hamilton et al., 2017)	0.612
GraphSAGE-pool (Hamilton et al., 2017)	0.600
GraphSAGE*	0.768
Const-GAT (ours)	0.934 ± 0.006
GAT (ours)	$\mathbf{0.973} \pm 0.002$

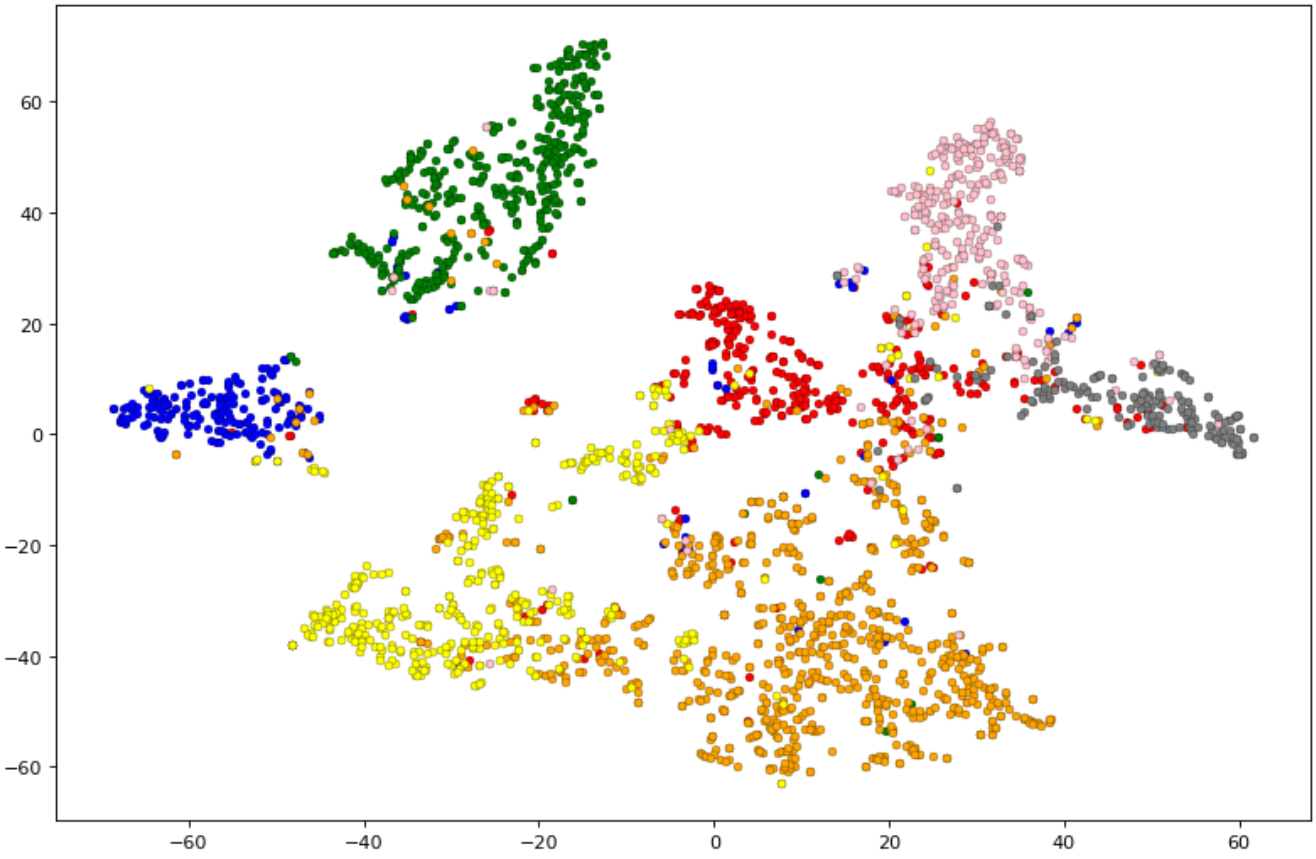
جدول ۴-۵: نتایج آموزش مدل GAT با رویکرد Inductive برگرفته از مقالهٔ اصلی

	ACCURACY
SVM	72.63%
MLP	$73.17 \pm 0.19\%$
GCN	$77.19 \pm 0.12\%$
GAT	$77.65 \pm 0.11\%$
APPNP	$79.84 \pm 0.10\%$

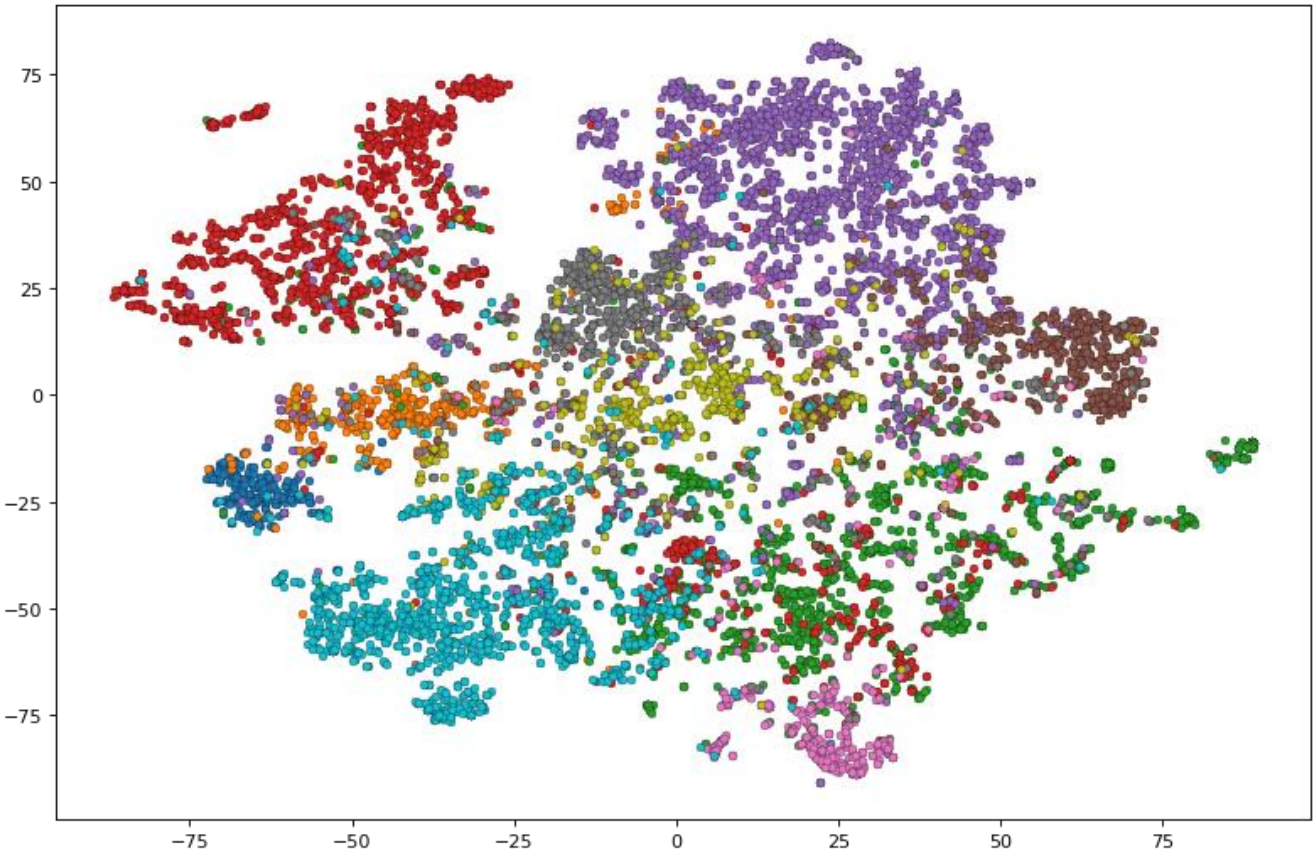
جدول ۵-۵: نتایج بدست آمده بر روی مجموعه داده Wiki_CS برگرفته از مقالهٔ اصلی

۶) نتایج

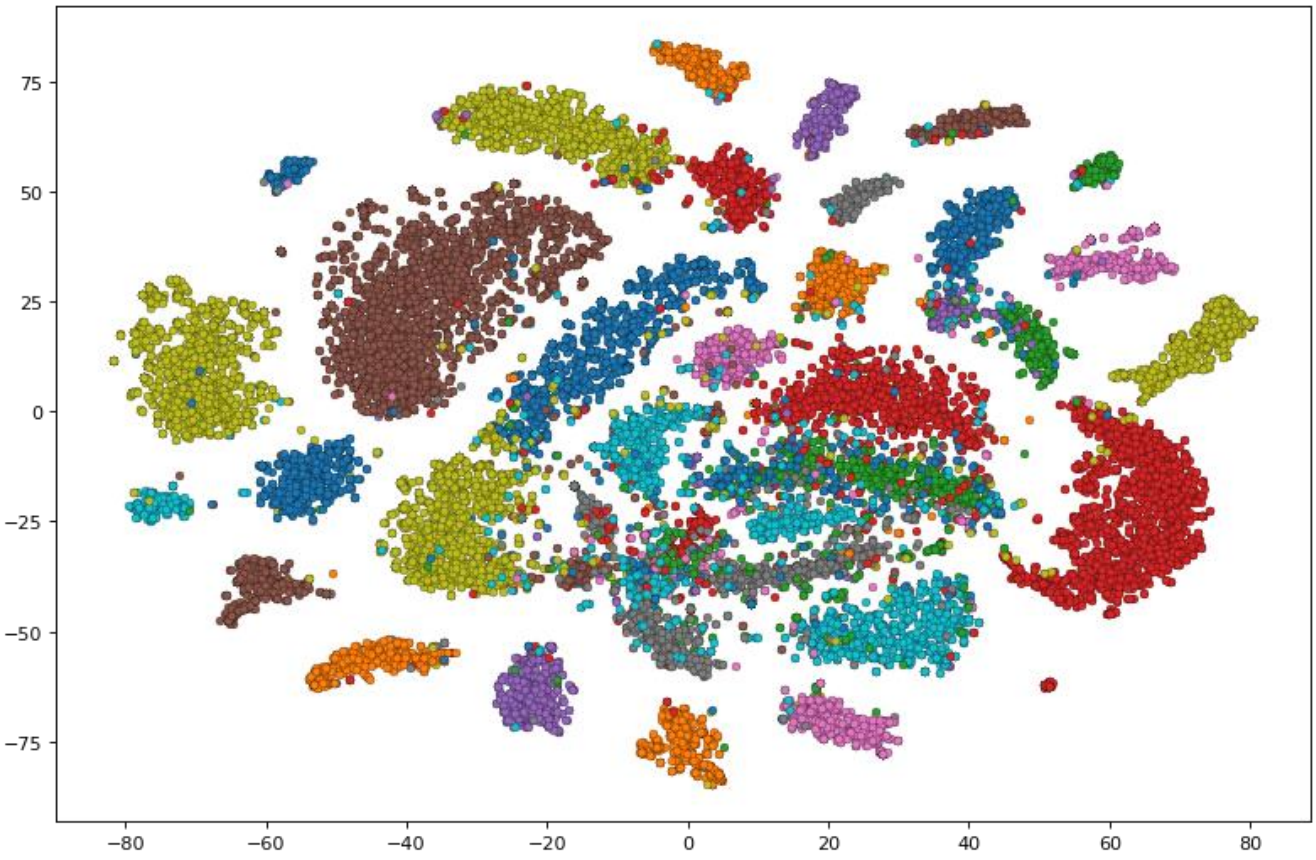
یکی از ویژگی‌های مهم مکانیزم‌های توجه فراهم‌سازی روش‌های مختلف تفسیر مدل است. به کمک روش t-SNE می‌توان بردارهای تعبیه تولید شده برای هر راس را در فضای دو بعدی رسم کرده و به بررسی عملکرد مدل در بازنمایی نزدیک‌تر راس‌های هم‌کلاس بپردازیم. شکل دو بعدی حاصل برای مجموعه داده‌هایی که با رویکرد Transductive آموزش دیده‌اند در شکل ۱-۶، ۲-۶ و ۳-۶ آورده شده است. لازم به ذکر است که رنگ هر راس نمایانگر کلاسی است که بدان تعلق دارد.



شکل ۱-۶: نمودار بازنمایی راس‌ها در مجموعه داده Cora به کمک الگوریتم t-SNE



شکل ۲-۶: نمودار بازنمایی راس‌ها در مجموعه داده Wiki_CS به کمک الگوریتم t-SNE



شکل ۳-۶: نمودار بازنمایی راس‌ها در مجموعه داده Reddit به کمک الگوریتم t-SNE

❖ پیاده‌سازی مورداستفاده برگرفته از کدهای GAT - Graph Attention Network (PyTorch) در گیت‌هاب می‌باشد.

مراجع

[1] P. Veličković, "GAT," [Online]. Available: https://github.com/PetarV-/GAT.

[2] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò and Y. Bengio, "Graph Attention Networks," in *ICLR*, 2018.

[3] M. Zitnik and L. Jure, "Predicting multicellular function through multi-layer tissue networks," in *Proceedings of the 25th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, 2017.

[4] A. T. Subramanian, V. K. Mootha, S. Mukherjee, B. L. Ebert, M. A. Gillette, A. Paulovich, S. L. Pomeroy, T. R. Golub, E. S. Lander and J. P. Mesirov, "Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles," in *Proceedings of the National Academy of Sciences*, 2005.

[5] W. L. Hamilton, R. Ying and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017.

[6] P Mernyei, C Cangea, "Wiki-CS: A Wikipedia-Based Benchmark for Graph Neural Networks," *arXiv preprint arXiv:2007.02901,* 2020.