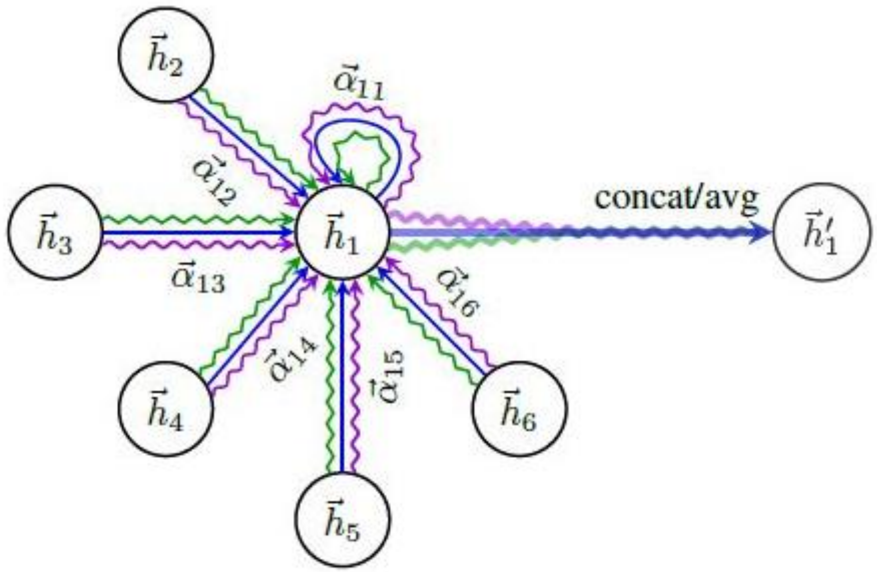
ALDE 2021 SUMMER
GNN SEMINAR
GRAPH ATTENTION NETWORK

Pusan National University, Korea
Sang Un, Park
sangwoon.alde@pusan.ac.kr

1. Graph Attention Networks(1)

- Graph에 대해 Node Classification을 수행하는 네트워크
  - 그래프 구조와 노드-레벨 Feature를 활용하면 node classification에서 좋은 성능을 보였음 (Graph Convolutional Network, GCN)
- Transformer 모델에서 사용된 Multi-head attention을 활용
  - GCN과 달리 그래프 구조에 의존하지 않고 정보를 종합
  - 주변 노드 feature 평균을 사용하는 GraphSAGE에서 조금 더 발전된 형태
- Inductive Learning이 가능한 모델
  - Supervised Learning
  - 학습된 모델을 바탕으로 새로운 그래프에 대해서 node classification을 할 수 있음

1. Graph Attention Networks(2)



- 인접 노드로부터 받는 Attention에 따라 Feature 갱신
  - $\overrightarrow{h_1}$ 은 자기 자신을 포함한 주변 노드로부터 영향을 받아 $\overrightarrow{h_1'}$ 으로 갱신됨
  - 받는 정보가 여러 줄인 것은 Multi-Head Attention임을 의미

2. Multi-Head Attention(1)

- RNN의 단점
  - Long-term dependency problem → 데이터 처리의 부정확성
  - Parallelization 불가 → 느린 계산 속도

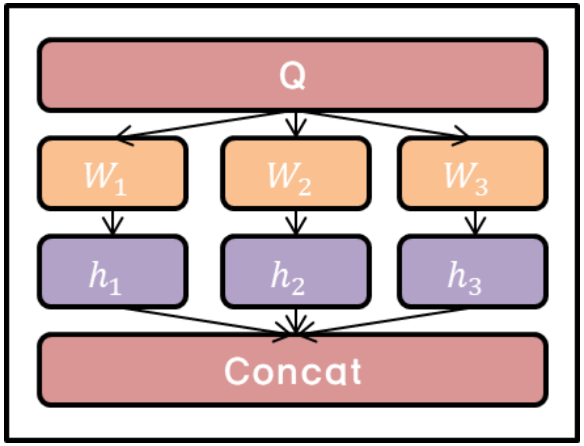
- Attention
  - Query : 값을 탐색하고자 하는 대상
  - Key : 이미 값이 정해진 대상
  - Value : 값
  - 들어온 Query에 대해서 각 Key와의 유사도를 바탕으로 Value를 계산

- Transformer model
  - Attention을 활용하여 RNN의 단점을 해결한 모델
  - sequence data에서 서로 거리가 먼 정보도 같이 활용할 수 있음
  - Masking을 활용하여 Parallelization이 가능

2. Multi-Head Attention(2)

- Attention 계산의 종류
  - Dot product attention : Luong et al.(2015)
  - Scaled dot product attention : Vaswani et al.(2017)
  - …

- Multi-Head Attention
  - 일반적인 Attention의 계산에는 Query(Q), Key(K), Value(V)가 동시에 argument로 들어감
  - Q, K, V를 서로 다른 가중치 행렬을 곱하여 병렬 연산을 수행하는 기법
  - Transformer model에서는 Scaled dot product Attention을 사용



3. Graph Attentional Layer

- GAT 전반에 걸쳐서 사용되는 Layer

- $N(i)$: 인접 노드의 집합
- σ : 활성 함수
- $W^{(l)}$: 가중치 행렬
- F : 각 노드의 Feature 수
- F\` : Length of Hidden Layer

① $e_{ij} = LeakyReLU(W\overrightarrow{h_i}, W\overrightarrow{h_j})$

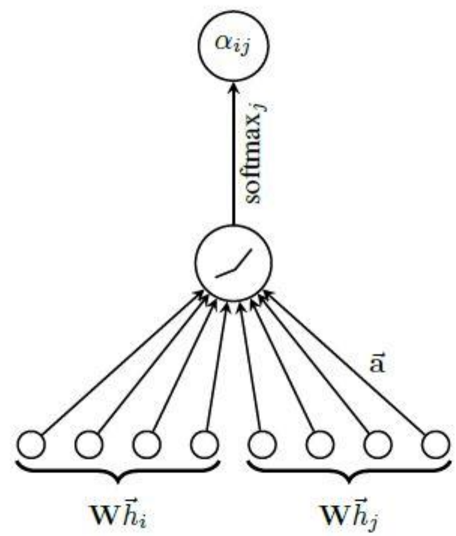
- 노드 i와 노드 j의 coefficient를 구하는 식
- output : F x F' 크기의 행렬

② $\alpha_{ij} = softmax_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in N(i)} \exp(e_{ik})}$

- 어텐션을 구하는 식
- 이웃의 전체 coefficient 합 중 e_{ij} 가 차지하는 비율
- output : F x F' 크기의 행렬

③ $\overrightarrow{h_i'} = \sigma\left(\sum_{j \in N(i)} \alpha_{ij}^{(l)} W^{(l)} \overrightarrow{h_j}\right)$

- 어텐션 α 를 바탕으로 Feature를 구함
- K개의 Attention Network의 concat의 Multi-Head Attention을 최종적으로 사용함
- output : K x F' 크기의 벡터



4. GAT vs GCN(1)

- GCN과 GraphSAGE는 인접 노드의 Feature을 그대로 사용

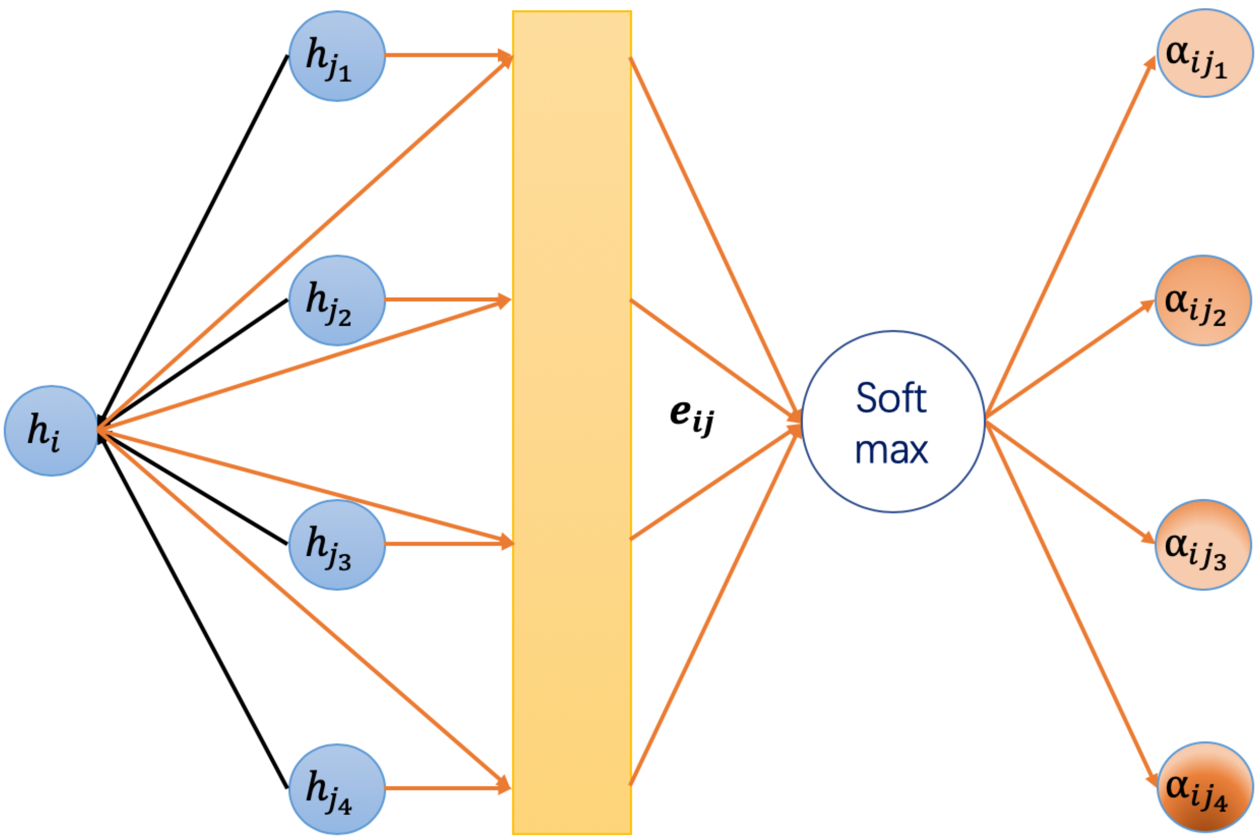
$$h_i^{(l+1)} = \sigma\left(\sum_{j \in N(i)} \frac{1}{c_{ij}} W^{(l)} h_j^{(l)}\right)$$

  - $N(i)$: 인접 노드의 집합(자기 자신이 포함될 수 있음)
  - c_{ij} : 그래프 구조에 따른 정규화 상수
  - σ : 활성 함수(ReLU)
  - $W^{(l)}$: Feature Transformation을 위한 공유 가중치 행렬

- GCN에서 $c_{ij} = \sqrt{|N(i)|}\sqrt{|N(j)|}$
- GraphSAGE에서 $c_{ij} = |N(i)|$

4. GAT vs GCN(2)

- GAT는 인접 노드의 Attention을 구하여 사용



4. GAT vs GCN(3)

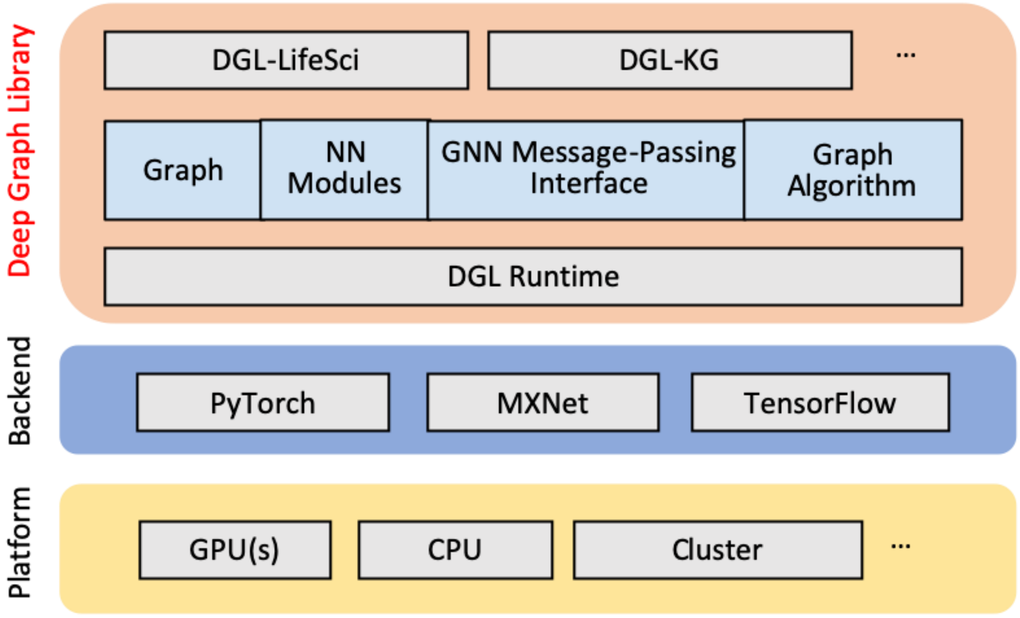
- 인접 노드가 서로 다른 중요도를 가짐
  - Model Capacity가 개선
  - 구조 해석에도 도움이 됨

- Attention Mechanism은 부분그래프로도 학습이 가능
  - 전체 그래프 구조에 대한 접근이 필요 X
  - Inductive Learning 가능

- 계산 과정에서 인접 노드를 전부 접근 가능
  - GraphSAGE는 k개의 이웃을 추출하므로 추출되는 이웃에 따라서 Aggregation 의 효율이 급변할 수 있음
  - GAT Paper에서 가장 효율적이라고 보고된 LSTM Aggregator를 GraphSAGE는 효과적으로 사용할 수 없음

5. DGL(Deep Graph Library)

- Graph Deep Learning과 관련된 여러 기능들을 모아둔 라이브러리
  - https://github.com/dmlc/dgl/
  - GCN / LGNN / GAT / Tree-LSTM등의 다양한 모델을 지원
- Pytorch / Apache MXNet에서 구동



5. DGL(Deep Graph Library)

- 실습 환경
  - Windows 10
  - Anaconda Prompt(Admin) 4.10.3
  - Python 3.9.6 (3.6 이상 요구)
  - DGL 0.7.0

- DGL 설치
  - conda install pytorch torchvision torchaudio cudatoolkit=10.2 -c pytorch
  - pip3 install torch==1.9.0+cu102 torchvision==0.10.0+cu102 torchaudio===0.9.0 -f https://download.pytorch.org/whl/torch_stable.html
  - conda install -c dglteam dgl

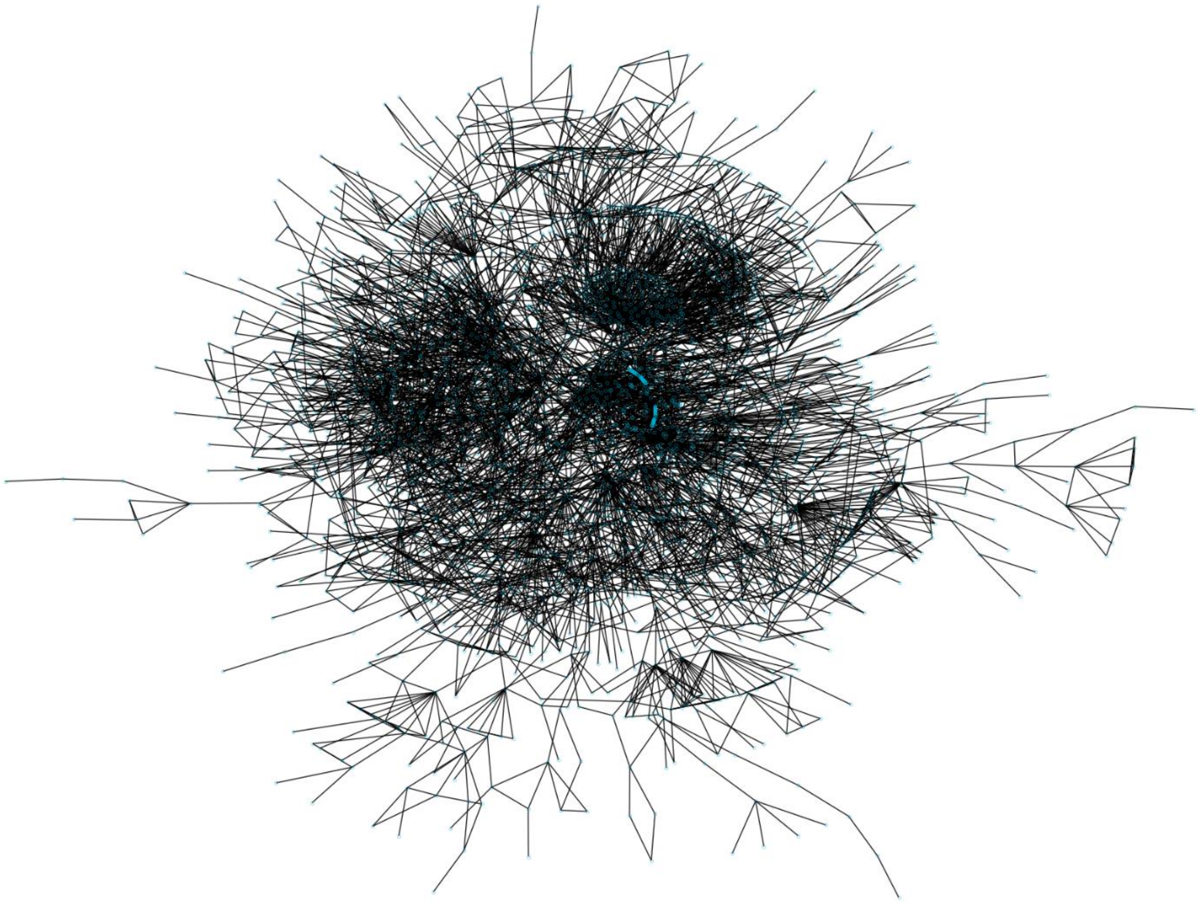
6. Citation Network Dataset

- Citation Network Dataset
  - Directed Graph 형태의 학습 데이터
  - Node : 논문(Paper)
  - Edge : A→B의 간선은 논문 A가 논문 B에서 인용되었음을 의미함
  - 해당 데이터셋을 총 7개의 Category로 분리(Node Classification)

- 데이터셋 정보
  - 2708 Nodes
  - 10556 Edges
  - # of Classes : 7
  - 140 Train, 500 Valid, 1000 Test

  - https://paperswithcode.com/dataset/citeseer

6. Citation Network Dataset



7. GATLayer

① $e_{ij} = LeakyReLU(W\vec{h_i}, W\vec{h_j})$

② $\alpha_{ij} = softmax_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in N(i)} \exp(e_{ik})}$

③ $\vec{h'_i} = \sigma\left(\sum_{j \in N(i)} \alpha_{ij}^{(l)} W^{(l)} \vec{h_j}\right)$

```
# GAT Layer 구현
class GATLayer(nn.Module):
    def __init__(self, g, in_dim, out_dim):
        super(GATLayer, self).__init__()
        self.g = g
        self.fc = nn.Linear(in_dim, out_dim, bias=False)
        # coefficient를 구하는 부분
        # equation (1)
        self.attn_fc = nn.Linear(2 * out_dim, 1, bias=False)
        self.reset_parameters()

    def reset_parameters(self):
        """Reinitialize learnable parameters."""
        gain = nn.init.calculate_gain('relu')
        nn.init.xavier_normal_(self.fc.weight, gain=gain)
        nn.init.xavier_normal_(self.attn_fc.weight, gain=gain)

    def edge_attention(self, edges):
        # edge UDF for equation (1)
        z2 = torch.cat([edges.src['z'], edges.dst['z']], dim=1)
        a = self.attn_fc(z2)
        return {'e': F.leaky_relu(a)}
```

7. GATLayer

① $e_{ij} = LeakyReLU(W\overrightarrow{h_i}, W\overrightarrow{h_j})$

② $\alpha_{ij} = softmax_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in N(i)} \exp(e_{ik})}$

③ $\overrightarrow{h_i'} = \sigma\left(\sum_{j \in N(i)} \alpha_{ij}^{(l)} W^{(l)} \overrightarrow{h_j}\right)$

```
    def message_func(self, edges):
        # message UDF for equation (2) & (3)
        return {'z': edges.src['z'], 'e': edges.data['e']}

    def reduce_func(self, nodes):
        # reduce UDF for equation (2) & (3)
        # mailbox에 저장된 coefficient를 softmax로 aggregate
        # equation (2)
        alpha = F.softmax(nodes.mailbox['e'], dim=1)
        # equation (3)
        h = torch.sum(alpha * nodes.mailbox['z'], dim=1)
        return {'h': h}

    def forward(self, h):
        # 각 노드 사이의 텐서 업데이트
        z = self.fc(h)
        self.g.ndata['z'] = z
        # equation (1)
        # coefficient 갱신
        self.g.apply_edges(self.edge_attention)
        # equation (2) & (3)
        # attention propagation
        self.g.update_all(self.message_func, self.reduce_func)
        return self.g.ndata.pop('h')
```

8. MultiHeadGATLayer

```python
# GATLayer 클래스를 바탕으로 Multi-Head Attention을 구현
class MultiHeadGATLayer(nn.Module):
    def __init__(self, g, in_dim, out_dim, num_heads, merge='cat'):
        super(MultiHeadGATLayer, self).__init__()
        self.heads = nn.ModuleList()
        for i in range(num_heads):
            self.heads.append(GATLayer(g, in_dim, out_dim))
        self.merge = merge

    def forward(self, h):
        head_outs = [attn_head(h) for attn_head in self.heads]
        if self.merge == 'cat':
            # concat on the output feature dimension (dim=1)
            return torch.cat(head_outs, dim=1)
        else:
            # merge using average
            return torch.mean(torch.stack(head_outs))
```

8. GAT

```
class GAT(nn.Module):
    def __init__(self, g, in_dim, hidden_dim, out_dim, num_heads):
        super(GAT, self).__init__()
        self.layer1 = MultiHeadGATLayer(g, in_dim, hidden_dim, num_heads)
        # Be aware that the input dimension is hidden_dim*num_heads since
        # multiple head outputs are concatenated together. Also, only
        # one attention head in the output layer.
        self.layer2 = MultiHeadGATLayer(g, hidden_dim * num_heads,
        out_dim, 1)

    def forward(self, h):
        h = self.layer1(h)
        h = F.elu(h)
        h = self.layer2(h)
        return h
```

9. Dataset Loading

```
# citeseer data를 불러오는 함수
def load_cora_data():
    data = citegrh.load_cora()
    features = torch.FloatTensor(data.features)
    labels = torch.LongTensor(data.labels)
    mask = torch.BoolTensor(data.train_mask)
    g = DGLGraph(data.graph)

    # draw citeseer graph
    '''
    nx_G = g.to_networkx().to_undirected()
    pos = nx.kamada_kawai_layout(nx_G)
    nx.draw(nx_G, pos, with_labels=False, node_size = 0.01, node_color='#00b4d9', width=0.3)
    plt.savefig("data_viz.png",dpi=300)
    '''
    return g, features, labels, mask

g, features, labels, mask = load_cora_data()

# create the model, 2 heads, each head has hidden size 8
net = GAT(g,
          in_dim=features.size()[1],
          hidden_dim=8,
          out_dim=7,
          num_heads=2)

# create optimizer
optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
```

10. Main Routine

```python
# main loop
dur = []
epoch_arr = []
loss_arr = []
for epoch in range(30):
    if epoch >= 3:
        t0 = time.time()

    logits = net(features)
    logp = F.log_softmax(logits, 1)
    loss = F.nll_loss(logp[mask], labels[mask])

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch >= 3:
        dur.append(time.time() - t0)

    pred = np.argmax(logp[mask].detach().numpy(), axis = 1)
    answ = labels[mask].numpy()
    acc = np.sum([1 if pred[i] == answ[i] else 0 for i in range(len(pred))]) / len(pred) * 100

    epoch_arr.append(epoch)
    loss_arr.append(loss.item())

    print("Epoch {:05d} | Loss {:.4f} | Accuracy {:.2f}% | Time(s) {:.4f}".format(epoch, loss.item(), acc, np.mean(dur)))
```

11. OpenMP Error

```
OMP: Error #15: Initializing libiomp5md.dll, but found libiomp5md.dll already initialized.
OMP: Hint This means that multiple copies of the OpenMP runtime have been linked into the program. That is dangerous, si
nce it can degrade performance or cause incorrect results. The best thing to do is to ensure that only a single OpenMP r
untime is linked into the process, e.g. by avoiding static linking of the OpenMP runtime in any library. As an unsafe, u
nsupported, undocumented workaround you can set the environment variable KMP_DUPLICATE_LIB_OK=TRUE to allow the program
to continue to execute, but that may cause crashes or silently produce incorrect results. For more information, please s
ee http://www.intel.com/software/products/support/.
```

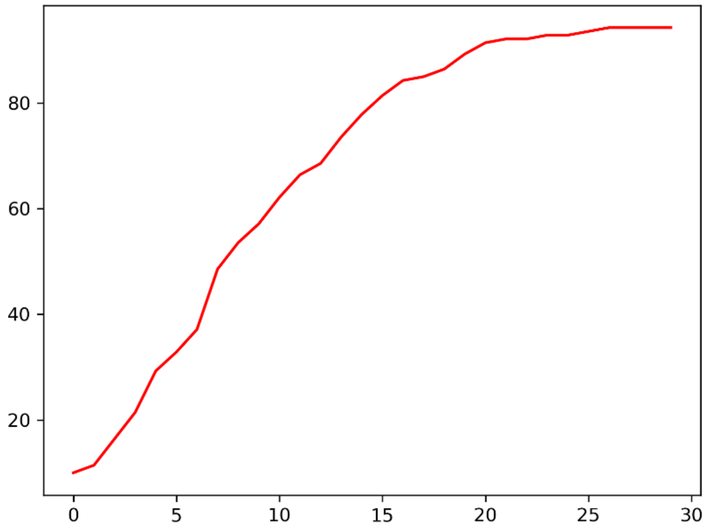
- 원인
  - 연산에 사용되는 OpenMP 라이브러리가 중복으로 호출되어 생기는 문제
- 해결법
  - OS 환경변수 추가
  - os.environ['KMP_DUPLICATE_LIB_OK']='True'

12. 실험 결과

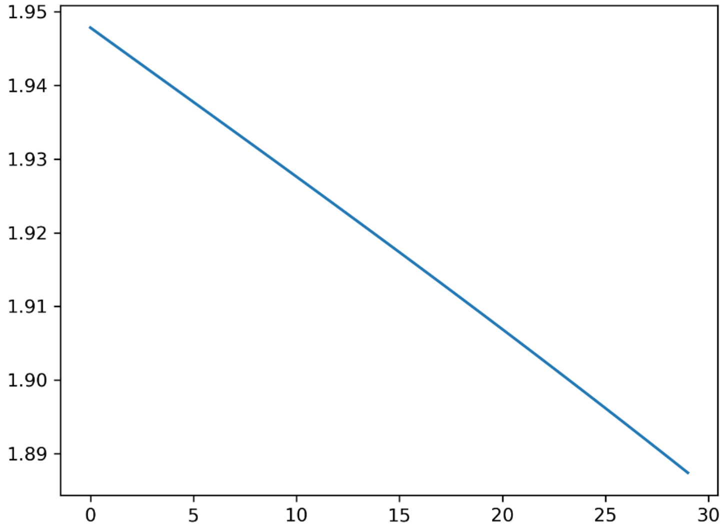
- Case 1
  - Epoch 30
  - hidden_dim = 8
  - out_dim = 7
  - num_heads = 2
- Result 1
  - Loss 1.8821
  - 92.14% Accuracy

```
Epoch 00000 | Loss 1.9452 | Accuracy 17.86% | Time(s) nan
Epoch 00001 | Loss 1.9431 | Accuracy 22.14% | Time(s) nan
Epoch 00002 | Loss 1.9410 | Accuracy 26.43% | Time(s) nan
Epoch 00003 | Loss 1.9389 | Accuracy 35.71% | Time(s) 0.1008
Epoch 00004 | Loss 1.9368 | Accuracy 40.71% | Time(s) 0.1037
Epoch 00005 | Loss 1.9347 | Accuracy 47.14% | Time(s) 0.1061
Epoch 00006 | Loss 1.9326 | Accuracy 50.00% | Time(s) 0.1032
Epoch 00007 | Loss 1.9305 | Accuracy 55.00% | Time(s) 0.1055
Epoch 00008 | Loss 1.9284 | Accuracy 60.00% | Time(s) 0.1039
Epoch 00009 | Loss 1.9263 | Accuracy 61.43% | Time(s) 0.1033
Epoch 00010 | Loss 1.9241 | Accuracy 65.71% | Time(s) 0.1025
Epoch 00011 | Loss 1.9220 | Accuracy 69.29% | Time(s) 0.1032
Epoch 00012 | Loss 1.9199 | Accuracy 73.57% | Time(s) 0.1027
Epoch 00013 | Loss 1.9177 | Accuracy 75.00% | Time(s) 0.1023
Epoch 00014 | Loss 1.9156 | Accuracy 77.86% | Time(s) 0.1021
Epoch 00015 | Loss 1.9134 | Accuracy 80.00% | Time(s) 0.1018
Epoch 00016 | Loss 1.9112 | Accuracy 80.00% | Time(s) 0.1012
Epoch 00017 | Loss 1.9090 | Accuracy 82.14% | Time(s) 0.1019
Epoch 00018 | Loss 1.9069 | Accuracy 83.57% | Time(s) 0.1019
Epoch 00019 | Loss 1.9047 | Accuracy 85.00% | Time(s) 0.1020
Epoch 00020 | Loss 1.9024 | Accuracy 85.71% | Time(s) 0.1020
Epoch 00021 | Loss 1.9002 | Accuracy 87.14% | Time(s) 0.1018
Epoch 00022 | Loss 1.8980 | Accuracy 87.86% | Time(s) 0.1021
Epoch 00023 | Loss 1.8958 | Accuracy 89.29% | Time(s) 0.1019
Epoch 00024 | Loss 1.8935 | Accuracy 90.00% | Time(s) 0.1016
Epoch 00025 | Loss 1.8913 | Accuracy 90.71% | Time(s) 0.1016
Epoch 00026 | Loss 1.8890 | Accuracy 92.14% | Time(s) 0.1014
Epoch 00027 | Loss 1.8867 | Accuracy 92.14% | Time(s) 0.1015
Epoch 00028 | Loss 1.8844 | Accuracy 92.14% | Time(s) 0.1012
Epoch 00029 | Loss 1.8821 | Accuracy 92.14% | Time(s) 0.1010
```

12. 실험 결과



epoch-accuracy graph of Case 1



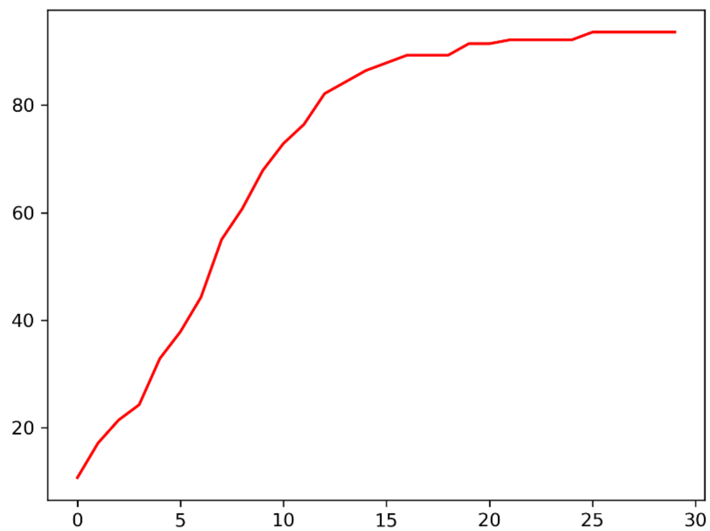
epoch-loss graph of Case 1

12. 실험 결과

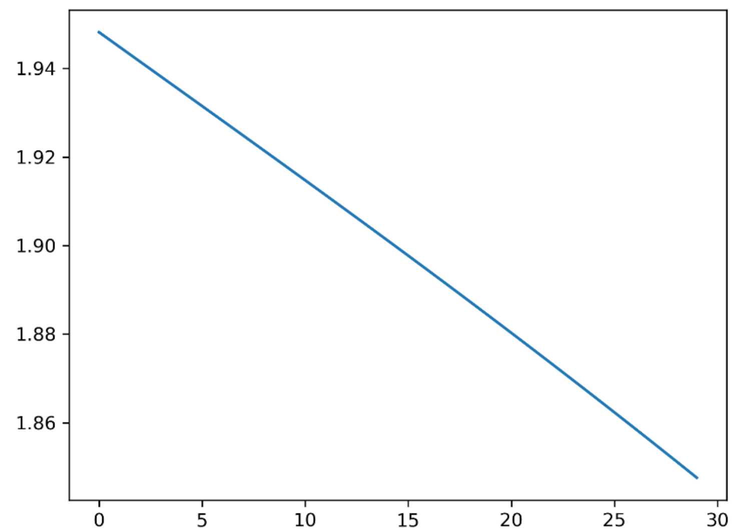
- Case 2
  - Epoch 30
  - hidden_dim = 8
  - out_dim = 7
  - num_heads = 4
- Result 2
  - Loss 1.8527
  - 96.43% Accuracy

```
Epoch 00000 | Loss 1.9470 | Accuracy 8.57% | Time(s) nan
Epoch 00001 | Loss 1.9439 | Accuracy 15.00% | Time(s) nan
Epoch 00002 | Loss 1.9408 | Accuracy 21.43% | Time(s) nan
Epoch 00003 | Loss 1.9377 | Accuracy 30.00% | Time(s) 0.1616
Epoch 00004 | Loss 1.9346 | Accuracy 40.00% | Time(s) 0.1656
Epoch 00005 | Loss 1.9315 | Accuracy 50.00% | Time(s) 0.1676
Epoch 00006 | Loss 1.9284 | Accuracy 55.00% | Time(s) 0.1688
Epoch 00007 | Loss 1.9253 | Accuracy 62.86% | Time(s) 0.1692
Epoch 00008 | Loss 1.9222 | Accuracy 70.00% | Time(s) 0.1684
Epoch 00009 | Loss 1.9190 | Accuracy 75.71% | Time(s) 0.1673
Epoch 00010 | Loss 1.9159 | Accuracy 79.29% | Time(s) 0.1674
Epoch 00011 | Loss 1.9127 | Accuracy 81.43% | Time(s) 0.1696
Epoch 00012 | Loss 1.9096 | Accuracy 82.86% | Time(s) 0.1702
Epoch 00013 | Loss 1.9064 | Accuracy 85.00% | Time(s) 0.1705
Epoch 00014 | Loss 1.9032 | Accuracy 85.00% | Time(s) 0.1696
Epoch 00015 | Loss 1.9000 | Accuracy 87.14% | Time(s) 0.1694
Epoch 00016 | Loss 1.8967 | Accuracy 87.86% | Time(s) 0.1694
Epoch 00017 | Loss 1.8935 | Accuracy 89.29% | Time(s) 0.1698
Epoch 00018 | Loss 1.8902 | Accuracy 89.29% | Time(s) 0.1692
Epoch 00019 | Loss 1.8869 | Accuracy 90.71% | Time(s) 0.1692
Epoch 00020 | Loss 1.8836 | Accuracy 91.43% | Time(s) 0.1688
Epoch 00021 | Loss 1.8803 | Accuracy 93.57% | Time(s) 0.1689
Epoch 00022 | Loss 1.8769 | Accuracy 93.57% | Time(s) 0.1686
Epoch 00023 | Loss 1.8735 | Accuracy 93.57% | Time(s) 0.1685
Epoch 00024 | Loss 1.8701 | Accuracy 93.57% | Time(s) 0.1685
Epoch 00025 | Loss 1.8667 | Accuracy 94.29% | Time(s) 0.1685
Epoch 00026 | Loss 1.8633 | Accuracy 95.00% | Time(s) 0.1686
Epoch 00027 | Loss 1.8598 | Accuracy 95.00% | Time(s) 0.1684
Epoch 00028 | Loss 1.8563 | Accuracy 95.71% | Time(s) 0.1682
Epoch 00029 | Loss 1.8527 | Accuracy 96.43% | Time(s) 0.1684
```

12. 실험 결과



epoch-accuracy graph of Case 2



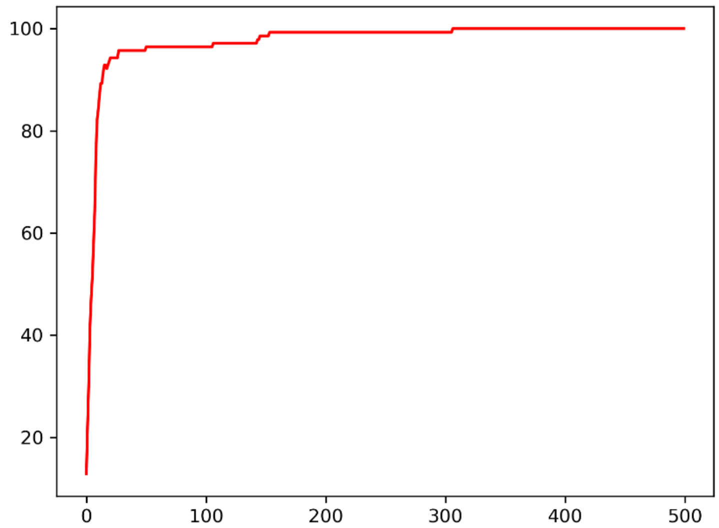
epoch-loss graph of Case 2

12. 실험 결과

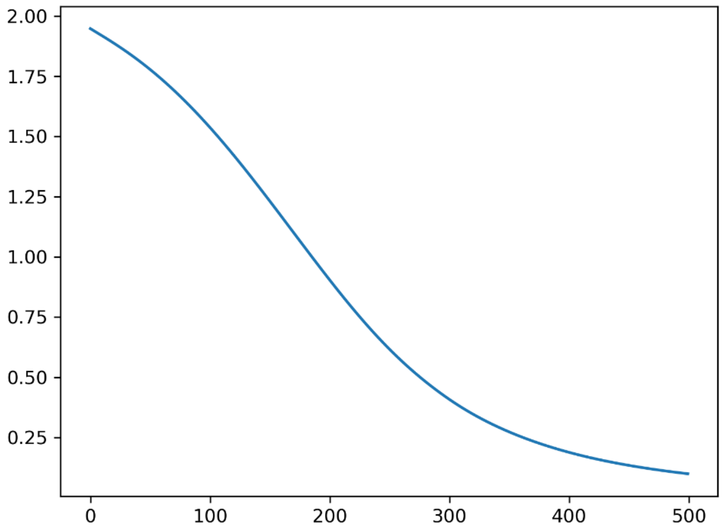
- Case 3
  - Epoch 500
  - hidden_dim = 8
  - out_dim = 7
  - num_heads = 4
- Result 3
  - Loss 0.9992
  - 100.00% Accuracy

```
Epoch 00475 | Loss 0.1145 | Accuracy 100.00% | Time(s) 0.1694
Epoch 00476 | Loss 0.1138 | Accuracy 100.00% | Time(s) 0.1694
Epoch 00477 | Loss 0.1131 | Accuracy 100.00% | Time(s) 0.1694
Epoch 00478 | Loss 0.1124 | Accuracy 100.00% | Time(s) 0.1694
Epoch 00479 | Loss 0.1117 | Accuracy 100.00% | Time(s) 0.1694
Epoch 00480 | Loss 0.1110 | Accuracy 100.00% | Time(s) 0.1694
Epoch 00481 | Loss 0.1104 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00482 | Loss 0.1097 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00483 | Loss 0.1091 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00484 | Loss 0.1084 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00485 | Loss 0.1078 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00486 | Loss 0.1071 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00487 | Loss 0.1065 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00488 | Loss 0.1059 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00489 | Loss 0.1052 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00490 | Loss 0.1046 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00491 | Loss 0.1040 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00492 | Loss 0.1034 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00493 | Loss 0.1028 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00494 | Loss 0.1022 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00495 | Loss 0.1016 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00496 | Loss 0.1010 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00497 | Loss 0.1004 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00498 | Loss 0.0998 | Accuracy 100.00% | Time(s) 0.1695
Epoch 00499 | Loss 0.0992 | Accuracy 100.00% | Time(s) 0.1695
```

12. 실험 결과



epoch-accuracy graph of Case 3



epoch-loss graph of Case 3

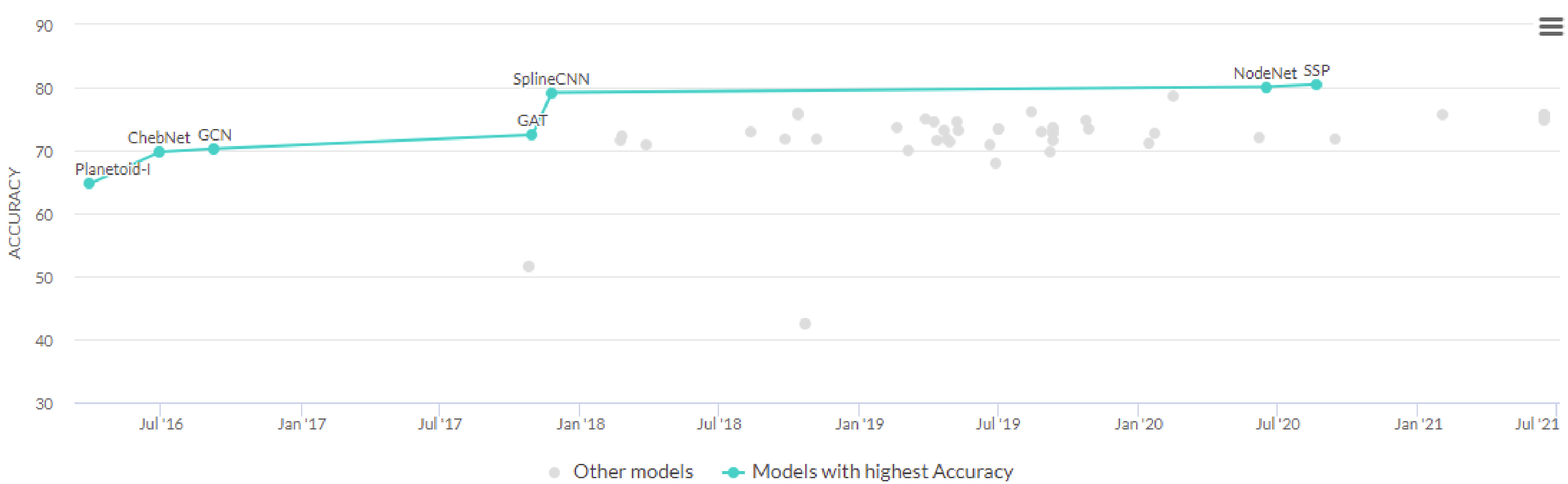
13. SOTA 현황

- Citeseer Dataset

Node Classification on Citeseer

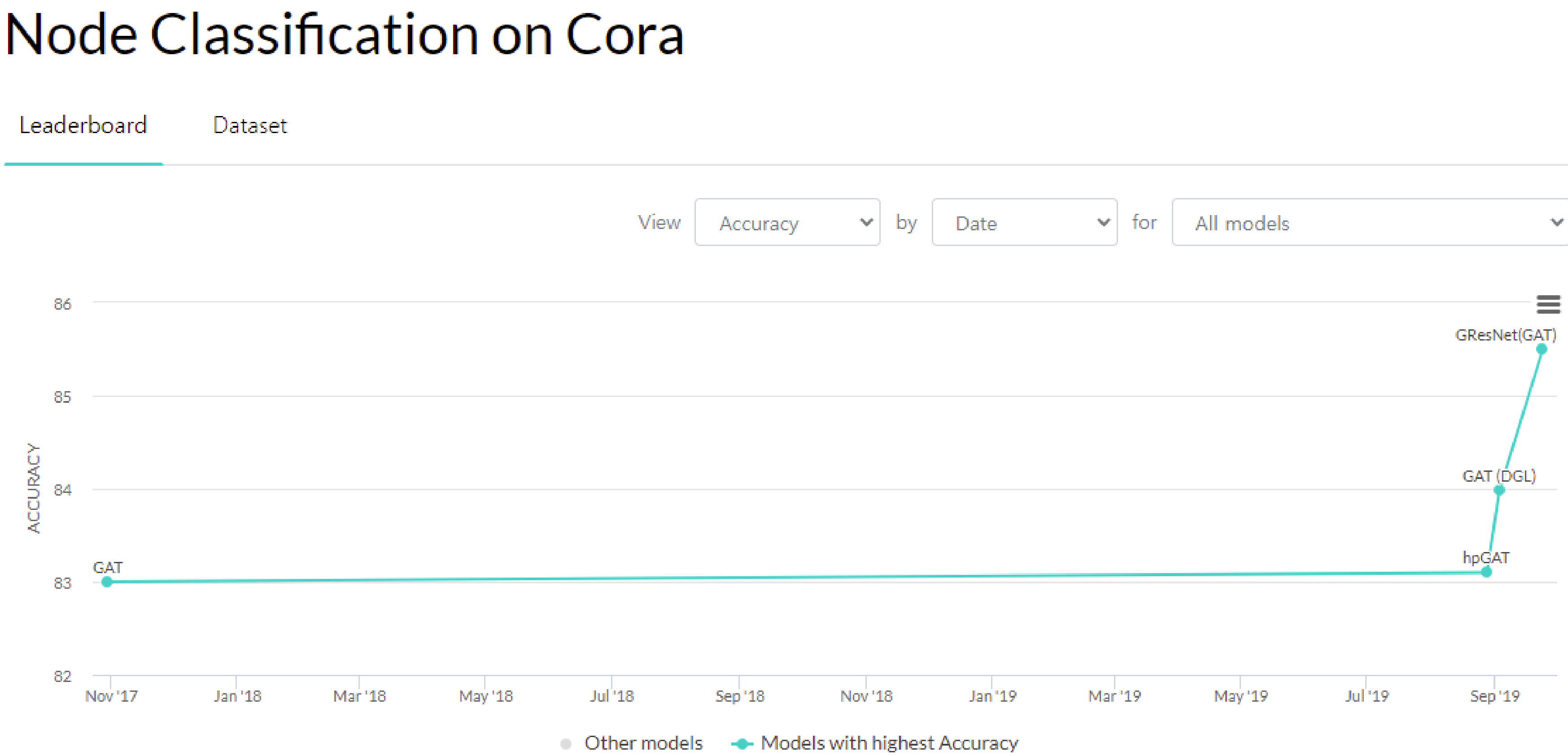
Leaderboard Dataset

View Accuracy by Date for All models



13. SOTA 현황

- GAT(Cora Dataset)



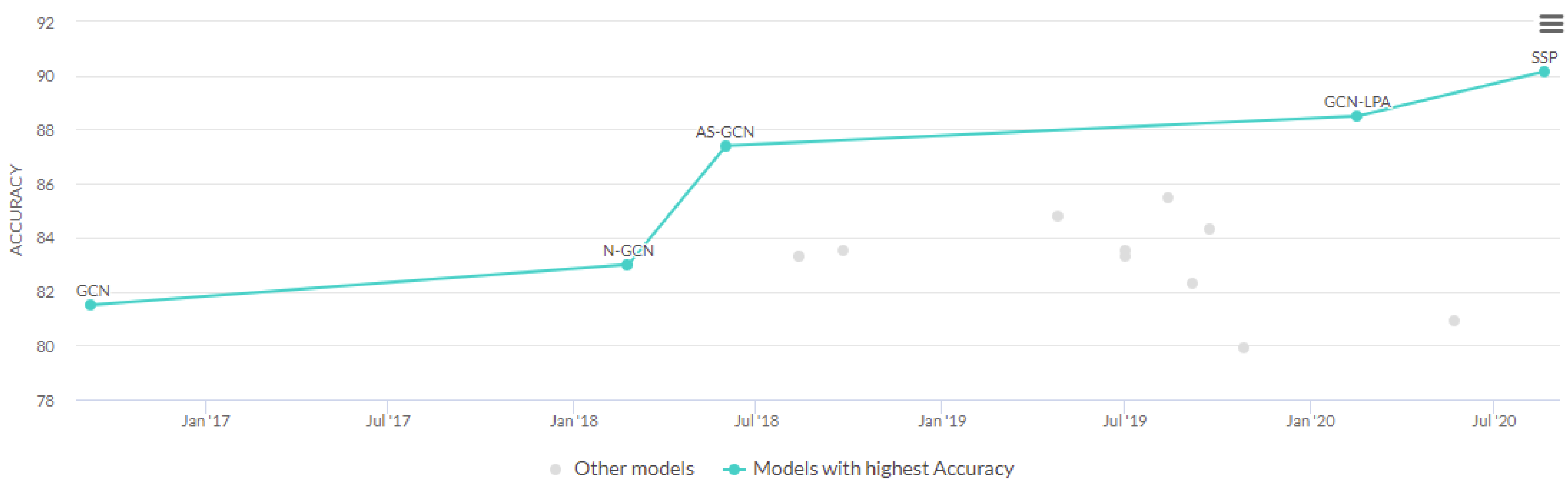
13. SOTA 현황

- GCN(Cora Dataset)

Node Classification on Cora

Leaderboard Dataset

View Accuracy by Date for All models



Filter: GCN GAT untagged

Edit Leaderboard

#. References

- Paper
  - P. Veličković et. al., "Graph Attention Networks," 2018, https://arxiv.org/abs/1710.10903
  - T. N. Kipf et al., "Semi-Supervised Classification with Graph Convolutional Networks," 2017, https://arxiv.org/abs/1609.02907
- Documents
  - https://docs.dgl.ai/en/0.6.x/tutorials/models/1_gnn/9_gat.html
  - 위 사이트에서 Pytorch와 dgl을 활용한 실습이 설명되어있음
- Blogs / Articles
  - https://pozalabs.github.io/transformer/
  - https://chioni.github.io/posts/gat/
  - https://greeksharifa.github.io/machine_learning/2021/05/29/GAT/
  - https://woosikyang.github.io/Graph-Attention-Network.html
  - https://medium.com/@eakhil711/an-introduction-to-graph-attention-networks-d41ed52e5b1e