Appendix

1 Model Architecture

1.1 ResNet

We utilize the standard ResNet18 and ResNet101 architectures introduced in [2] with a few modifications. The first pooling layers are removed, and the kernel size of the first convolution layers is changed to (3, 3). Additionally, the linear layers are set to have an output dimension of 512.

The following PyTorch-style code demonstrates how we transform the original ResNet into the version used in our study:

```python
def resnetToCifar(resnet):
    resnet.conv1 = nn.Conv2d(3, 64, (3, 3), padding='same')
    resnet.maxpool = nn.Identity()
    resnet.fc = nn.Linear(resnet.fc.in_features, 512)
    return resnet
```

1.2 Vision Transformer

The basic architecture of the Vision Transformer (ViT) we employ is consistent with the initial version presented in [1]. However, we adapt its scale to match that of ResNet101 and set the output dimension to 512. The following code defines the modified ViT:

```python
def ViTcifar():
    model = VisionTransformer(image_size=32, patch_size=4,
                              num_layers=16, num_heads=8, hidden_dim=512,
                              mlp_dim=1536, dropout=0.2, attention_dropout=0.2)
    model.heads = nn.Linear(512, 512)
    return model
```

1.3 Graph Convolution Network

The Graph Convolutional Network (GCN) we utilize consists of three layers, including one embedding layer and two graph convolution layers proposed in [3]. The input dimension is set to 11, considering that CIFAR-10 [4] has 10 classes, and we add an additional "masked" class during training. The hidden dimension is 16, and the output dimension is 10. We also introduce a learnable scale parameter $\alpha$ in each convolution layer, transforming the adjacency matrix as $\hat{A} = \alpha A + I_N$.

```python
class GCNhead(nn.Module):
    def __init__(self):
        self.embedding = nn.Embedding(11, 16)
        self.GCN = GCNSequential(
            GCNWithLoop(16, 16, activation=nn.SELU()),
            GCNWithLoop(16, 10))
```

```
def forward(self, A, node_feat):
    return self.GCN(A, self.embedding(node_feat))

The parameter counts for each model are listed in Table 1.

## 2 Training Recipe

We present some key hyperparameters in Table 2. It is important to note that these parameters are tuned to ensure stable training processes and maximize GPU usage, rather than achieving state-of-the-art performance.

### Table 2: Hyperparameters in the Training Recipe.

<table>
<thead>
<tr>
<th></th>
<th>Contrastive</th>
<th>Supervised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Epochs</td>
<td>1200</td>
<td>120</td>
</tr>
<tr>
<td>Batch size</td>
<td>1200</td>
<td>600</td>
</tr>
<tr>
<td>Learning rate</td>
<td>1.E-05</td>
<td>1.E-05</td>
</tr>
<tr>
<td>Optimizer</td>
<td>AdamW</td>
<td>Cosine Annealing</td>
</tr>
<tr>
<td>Learning rate scheduler</td>
<td>Linear</td>
<td></td>
</tr>
<tr>
<td>Warm-up learning rate</td>
<td>1.E-03</td>
<td>1.E-03</td>
</tr>
<tr>
<td>Warm-up epochs</td>
<td>150</td>
<td>100</td>
</tr>
<tr>
<td>Minimum learning rate</td>
<td>1.E-08</td>
<td>1.E-08</td>
</tr>
<tr>
<td>Negative sample for contrast</td>
<td>128,000</td>
<td>N/A</td>
</tr>
</tbody>
</table>

## References


