Few-Shot Learning with Graph Neural Networks

Jonggwon Park

Seoul National University

Few-Shot Learning with Graph Neural Networks

- International Conference on Learning Representations (ICLR) 2018
- Authors : Victor Garcia, Joan Bruna

- In this paper, end-to-end learning methods of few-shot learning and semi-supervised learning were proposed using graph neural networks (GNN).

- It showed state of the art performance in few-shot learning.

Problem Set-up

- Input-output pairs : $(T_i, Y_i)_i \sim P$, P : partially-labeled image collections

$$T = \left\{\{(x_1, l_1), \dots (x_s, l_s)\}, \{\tilde{x}_1, \dots, \tilde{x}_r\}, \{\bar{x}_1, \dots, \bar{x}_t\}; l_i \in \{1, K\}, x_i, \tilde{x}_j, \bar{x}_j \sim \rho_l(R^N)\right\},$$
$$Y = (y_1, \dots, y_t) \in \{1, K\}^t$$

  s : the number of labeled samples
  r : the number of unlabeled samples
  t : the number of samples to classify, t = 1 only in this paper
  K : the number of classes
  $\rho_l(R^N)$: class-specific image distribution over R^N

- The standard supervised learning objective

$$\min_\theta \frac{1}{L} \sum_{i \le L} \mathcal{L}(\Phi(T_i, \theta), Y_i) + \mathcal{R}(\theta)$$

  $\Phi(T_i, \theta) = p(Y|T)$
  $\mathcal{R}$: standard regularization objective

- Few-shot learning : r = 0, t = 1, $s = qK$ (each label appears exactly q times) => q-shot, K-way learning
- Semi-supervised learning : r > 0, t = 1
- Active learning : the learner has the ability to request labels from the sub-collection $\{\tilde{x}_1, \dots, \tilde{x}_r\}$

Graph Representations

- The input T is represented with a fully-connected graph $G_T = (V, E)$ where nodes $v_a \in V$ correspond to the images in T (both labeled and unlabeled)

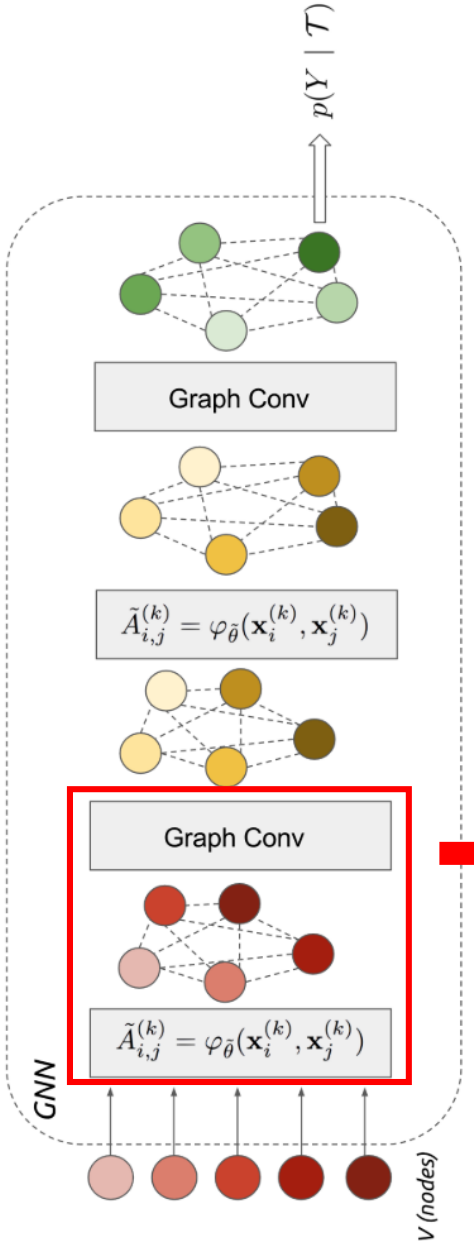
- Graph Neural Networks
  - An input signal $F \in \mathbb{R}^{V \times d}$ on the vertices of a weighted graph G
  - $\mathcal{A}$: a family of graph intrinsic linear operator
    ex) adjacency operator $A : F \rightarrow A(F)\ where\ (AF)_i = \sum_{j \sim i} w_{i,j} F_j\ , with\ i \sim j$
    iff $(i, j) \in E\ and\ w_{i,j}$ its associated weight
  - GNN layer $Gc(\cdot)$ receives as input a signal $x^k \in \mathbb{R}^{V \times d_k}$ and produce $x^{k+1} \in \mathbb{R}^{V \times d_{k+1}}$

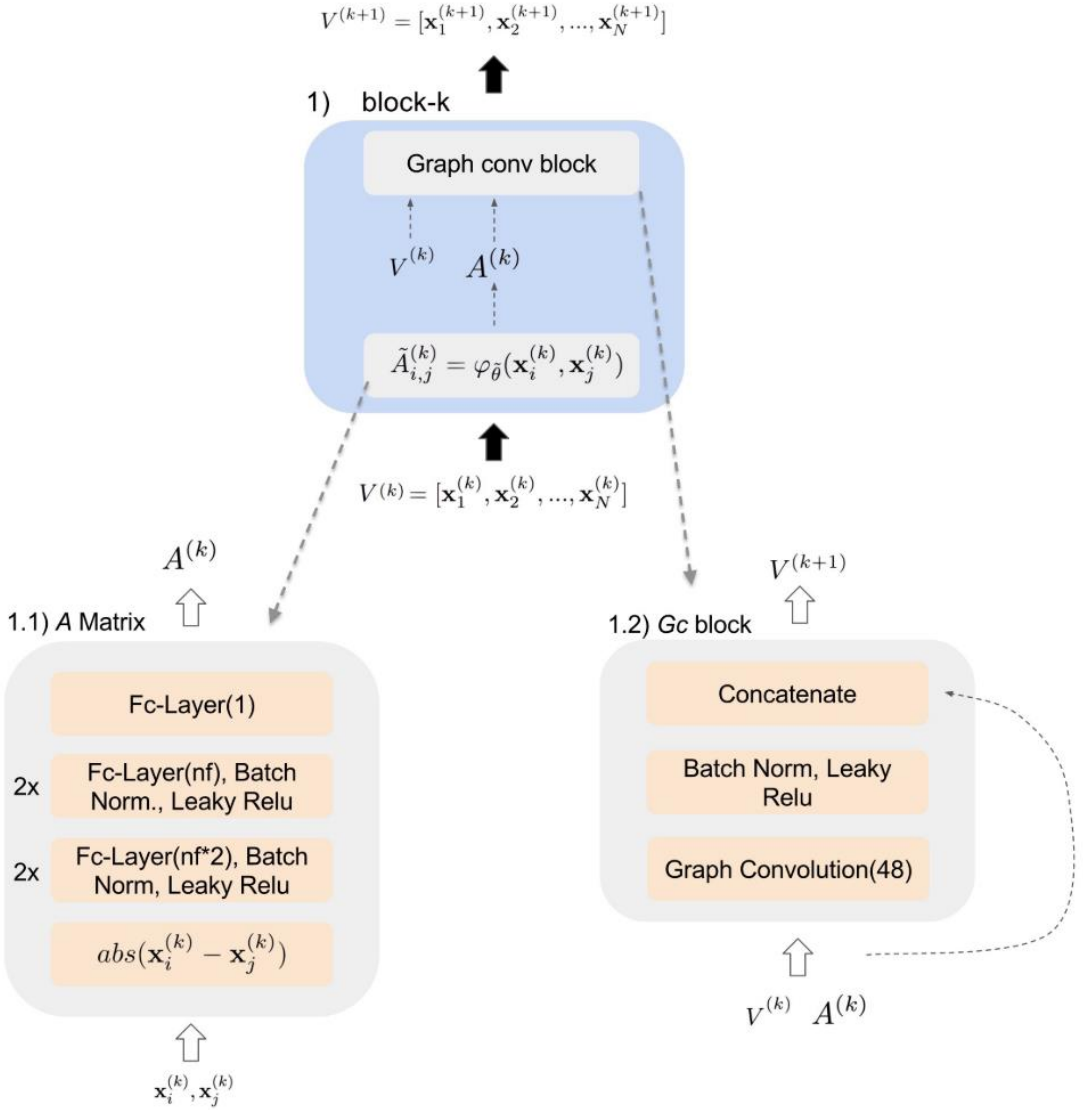
$$x^{k+1} = Gc\left(x^k\right) = \rho\left(\sum_{B \in \mathcal{A}} B x^k \theta_B^k\right) \quad \text{where } \Theta = \left\{\theta_1^k, \ldots, \theta_{|\mathcal{A}|}^k\right\}_k, \theta_B^k \in \mathbb{R}^{d_k \times d_{k+1}}$$

Graph Representations

- Edge features
  - Edge features $\tilde{A}^k$ from the current node hidden representation
$$\tilde{A}_{i,j}^k = \varphi_{\tilde{\theta}}\left(x_i^k, x_j^k\right) = MLP_{\tilde{\theta}}\left(abs\left(x_i^k - x_j^k\right)\right)$$
  - $\varphi_{\tilde{\theta}}$ is a distance metric : Symmetry $\varphi_{\tilde{\theta}}(a,b) = \varphi_{\tilde{\theta}}(\mathrm{b},\mathrm{a})$, Identity $\varphi_{\tilde{\theta}}(a,a) = 0$
  - This trainable adjacency is normalized using a softmax along each row
  - Update rules for node features : applying GNN(Gc) with the generator family $\mathcal{A} = \{\tilde{A}^k, 1\}$
- Initial node features
  - For images $x_i \in T$ with known label l_i, initial node feature $x_i^0 = \left(\phi(x_i), h(l_i)\right)$ where ϕ is a CNN and $h(l) \in \mathbb{R}_+^K$ is a one-hot encoding of the label
  - For images $\tilde{x}_j, \tilde{x}_{j'}$ with unknown label l_i, $h(l)$ is replaced with the uniform distribution over the K-simplex ($K^{-1}1_K$)

Model


$p(Y \mid \mathcal{T})$
Graph Conv
$\tilde{A}_{i,j}^{(k)} = \varphi_{\tilde{\theta}}(\mathbf{x}_i^{(k)}, \mathbf{x}_j^{(k)})$
Graph Conv
$\tilde{A}_{i,j}^{(k)} = \varphi_{\tilde{\theta}}(\mathbf{x}_i^{(k)}, \mathbf{x}_j^{(k)})$
GNN
V (nodes)

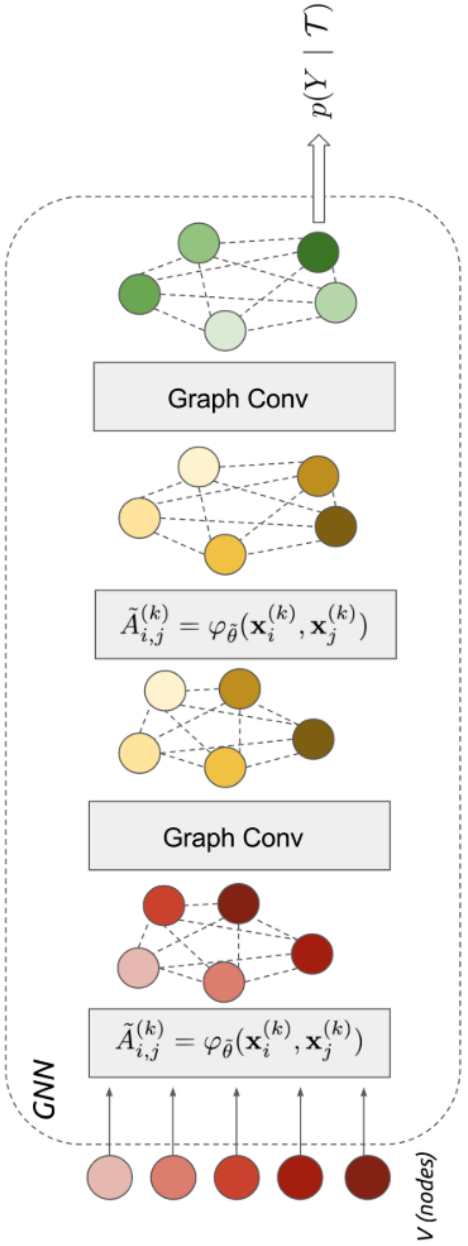

$V^{(k+1)} = [\mathbf{x}_1^{(k+1)}, \mathbf{x}_2^{(k+1)}, ..., \mathbf{x}_N^{(k+1)}]$
1) block-k
Graph conv block
$V^{(k)}$
$A^{(k)}$
$\tilde{A}_{i,j}^{(k)} = \varphi_{\tilde{\theta}}(\mathbf{x}_i^{(k)}, \mathbf{x}_j^{(k)})$
$V^{(k)} = [\mathbf{x}_1^{(k)}, \mathbf{x}_2^{(k)}, ..., \mathbf{x}_N^{(k)}]$
$A^{(k)}$
1.1) A Matrix
Fc-Layer(1)
2x
Fc-Layer(nf), Batch Norm., Leaky Relu
2x
Fc-Layer(nf*2), Batch Norm, Leaky Relu
$abs(\mathbf{x}_i^{(k)} - \mathbf{x}_j^{(k)})$
$\mathbf{x}_i^{(k)}, \mathbf{x}_j^{(k)}$
$V^{(k+1)}$
1.2) Gc block
Concatenate
Batch Norm, Leaky Relu
Graph Convolution(48)
$V^{(k)}$ $A^{(k)}$

Training

- Few-shot and semi-supervised learning
  - The model is asked only to predict the label Y corresponding to the image to classify $\bar{x} \in T$, associated with node * in the graph
  - Cross-entropy loss

$$\mathcal{L}(\Phi(T;\theta), Y) = -\sum_{k} y_k \log P(Y_* = y_k | T)$$

- Active learning
  - The model has the intrinsic ability to query for one of the labels from $\{\tilde{x}_1, \dots, \tilde{x}_r\}$
  - The querying is done after the first layer of the GNN
    $Attention = Softmax(g(x^1_{\{1,\dots,r\}}))$, $g(x^1_i) \in \mathbb{R}^1$, g is parametrized by neural network
  - at test time keep the maximum value, at train time randomly sample one value based on its multinomial probability => the label of the queried vector $h(l_{i*})$ is obtained
  - This value is summed to the current representation
    $x^1_{i*} = [Gc(x^0_{i*}), x^0_{i*}] = [Gc(x^0_{i*}), (\phi(x_{i*}), h(l_{i*}))]$
  - This attention part is trained end-to-end by backpropagating the loss



Experiments

- Mini-Imagenet dataset
  - 84 X 84 RGB images from 100 different classes with 600 samples per class
  - Split based on class : 64 classes for training, 16 for validation, 20 for testing

- The embedding architecture for initial node features
  - {3×3-conv. layer (64 filters), batch normalization, max pool(2, 2), leaky relu},
    {3×3-conv. layer (96 filters), batch normalization, max pool(2, 2), leaky relu},
    {3×3-conv. layer (128 filters), batch normalization, max pool(2, 2), leaky relu, dropout(0.5)},
    {3×3-conv. layer (256 filters), batch normalization, max pool(2, 2), leaky relu, dropout(0.5)},
    {fc-layer (128 filters), batch normalization}

Experiments

- Few-shot learning accuracy
  - TCML performed better, but has a much more complex embedding network.
  - TCML(~11M) has more parameters than 3 layer GNN(~400K)

	5-Way	
Model	**1-shot**	**5-shot**
Matching Networks Vinyals et al. (2016)	43.6%	55.3%
Prototypical Networks Snell et al. (2017)	46.61% ±0.78%	65.77% ±0.70%
Model Agnostic Meta-learner Finn et al. (2017)	48.70% ±1.84%	63.1% ±0.92%
Meta Networks Munkhdalai & Yu (2017)	49.21% ±0.96	-
Ravi & Larochelle Ravi & Larochelle (2016)	43.4% ±0.77%	60.2% ±0.71%
TCML Mishra et al. (2017)	55.71% ±0.99%	68.88% ±0.92%
Our metric learning + KNN	49.44% ±0.28%	64.02% ±0.51%
Our GNN	50.33% ±0.36%	66.41% ±0.63%

Experiments

- Semi-supervised accuracy
  - Trained only with labeled means experimenting with ignoring unlabeled images
  - In the semi supervised setting, unlabeled images were also constructed as a graph
  - using unlabeled images helps to classify test image

Model	5-Way 5-shot 20%-labeled	40%-labeled	100%-labeled
GNN - Trained only with labeled	50.33% ±0.36%	56.91% ±0.42%	66.41% ±0.63%
GNN - Semi supervised	52.45% ±0.88%	58.76% ±0.86%	66.41% ±0.63%

Experiments

- Active learning
  - The network query for the label of one sample from the unlabeled ones
  - Random baseline : the network chooses a random sample to be labeled
  - Performance improvement with AL : GNN manages to correctly choose a more informative sample than a random one

Method	5-Way 5-shot 20%-labeled
GNN - AL	55.99% ±1.35%
GNN - Random	52.56% ±1.18%

Conclusions

- This paper explored graph neural representations for few-shot, semi-supervised and active learning

- The graph formulation is helpful to unify several training setups (few-shot, active, semi-supervised) under the same framework

실험 결과

- 논문과 동일하게 Mini-Imagenet dataset으로 실험한 결과

method	5-way 1-shot	5-way 5-shot	5-way 5-shot 20% labeled
GNN	50.107 %	66.000 %	50.923 %

- semi-supervised 세팅(5-way 5-shot 20% labeled)의 성능이 5-way 1-shot 세팅에 비해 크게 개선되지 않음
  - 이는 논문에 기록된 semi-supervised 세팅의 성능(52.45%)에 크게 못 미치는 성능
- Few-shot learning의 성능은 논문에 기록된 것과 유사하였음

코드 제출

- 제출 코드 Github 주소 : https://github.com/jayg996/few-shot-gnn
- 공개된 코드를 활용하였음
- 공개 코드는 Pytorch 버전이 0.3.1에 맞게 짜여져 있어서, 1.3.0버전에 맞도록 수정함
  - 주로 Variable 사용하는 부분을 제거함
  - 수정한 부분 : https://github.com/jayg996/few-shot-gnn/commit/6afa395337949e1814cfc4b9bb895fde3927a9a3

코드 분석 : 각 파일 설명

- main.py : 실험을 실행하는 코드
- test.py : 실험에서 test iteration을 수행하는 코드
- data/mini_imagenet.py : mini_imagenet 데이터셋을 불러오는 코드
- data/generator.py : 불러온 데이터셋을 이용하기 적절하게 batch로 만들어주는 코드
- models/models.py : 사용되는 embedding network와 GNN을 구현한 코드
- models/gnn_iclr.py : GNN에서 Graph conv와 adjacency matrix를 계산하는 것을 구현한 코드

코드 분석 (main.py)

```python
def train_batch(model, data):
    [enc_nn, metric_nn, softmax_module] = model
    [batch_x, label_x, batches_xi, labels_yi, oracles_yi, hidden_labels] = data

    # Compute embedding from x and xi_s
    z = enc_nn(batch_x)[-1]
    zi_s = [enc_nn(batch_xi)[-1] for batch_xi in batches_xi]

    # Compute metric from embeddings
    out_metric, out_logits = metric_nn(inputs=[z, zi_s, labels_yi, oracles_yi, hidden_labels])
    logsoft_prob = softmax_module.forward(out_logits)

    # Loss
    label_x_numpy = label_x.cpu().data.numpy()
    formatted_label_x = np.argmax(label_x_numpy, axis=1)
    formatted_label_x = torch.LongTensor(formatted_label_x)
    if args.cuda:
        formatted_label_x = formatted_label_x.cuda()
    loss = F.nll_loss(logsoft_prob, formatted_label_x)
    loss.backward()

    return loss
```

- 하나의 mini batch data와 모델을 불러와서 학습하는 함수
- Labeled sample과 test sample을 enc_nn을 이용하여 embedding 값으로 계산
- Metric_nn (GNN)을 이용하여 test sample에 대한 logit값을 계산하여, 정답 label로 loss를 계산하여 return함

코드 분석 (main.py)

```
def train():
    train_loader = generator.Generator(args.dataset_root, args, partition='train', dataset=args.dataset)
    io.cprint('Batch size: '+str(args.batch_size))

    #Try to load models
    enc_nn = models.load_model('enc_nn', args, io)
    metric_nn = models.load_model('metric_nn', args, io)

    if enc_nn is None or metric_nn is None:
        enc_nn, metric_nn = models.create_models(args=args)
    softmax_module = models.SoftmaxModule()

    if args.cuda:
        enc_nn.cuda()
        metric_nn.cuda()

    io.cprint(str(enc_nn))
    io.cprint(str(metric_nn))

    weight_decay = 0
    if args.dataset == 'mini_imagenet':
        print('Weight decay '+str(1e-6))
        weight_decay = 1e-6
    opt_enc_nn = optim.Adam(enc_nn.parameters(), lr=args.lr, weight_decay=weight_decay)
    opt_metric_nn = optim.Adam(metric_nn.parameters(), lr=args.lr, weight_decay=weight_decay)

    enc_nn.train()
    metric_nn.train()
    counter = 0
    total_loss = 0
    val_acc, val_acc_aux = 0, 0
    test_acc = 0
    for batch_idx in range(args.iterations):

        ###################
        # Train
        ###################
        data = train_loader.get_task_batch(batch_size=args.batch_size, n_way=args.train_N_way,
                                           unlabeled_extra=args.unlabeled_extra, num_shots=args.train_N_shots,
                                           cuda=args.cuda)
        [batch_x, label_x, _, _, batches_xi, labels_yi, oracles_yi, hidden_labels] = data

        opt_enc_nn.zero_grad()
        opt_metric_nn.zero_grad()

        loss_d_metric = train_batch(model=[enc_nn, metric_nn, softmax_module],
                                    data=[batch_x, label_x, batches_xi, labels_yi, oracles_yi, hidden_labels])

        opt_enc_nn.step()
        opt_metric_nn.step()

        adjust_learning_rate(optimizers=[opt_enc_nn, opt_metric_nn], lr=args.lr, iter=batch_idx)
```

- main.py 파일이 실행될 때, train 함수가 실행됨
- Data loader를 준비하고, model을 선언
- 각 iteration에서는 batch data를 불러와서 앞 slide의 train_batch 함수를 이용하여 loss를 계산
- 계산된 loss를 이용하여 enc_nn과 metric_nn을 학습

코드 분석 (main.py)

```
        ###################
        # Display
        ###################
        counter += 1
        total_loss += loss_d_metric.item()
        if batch_idx % args.log_interval == 0:
            display_str = 'Train Iter: {}'.format(batch_idx)
            display_str += '\tLoss_d_metric: {:.6f}'.format(total_loss/counter)
            io.cprint(display_str)
            counter = 0
            total_loss = 0

        ###################
        # Test
        ###################
        if (batch_idx + 1) % args.test_interval == 0 or batch_idx == 20:
            if batch_idx == 20:
                test_samples = 100
            else:
                test_samples = 3000
            if args.dataset == 'mini_imagenet':
                val_acc_aux = test.test_one_shot(args, model=[enc_nn, metric_nn, softmax_module],
                                                 test_samples=test_samples*5, partition='val')
            test_acc_aux = test.test_one_shot(args, model=[enc_nn, metric_nn, softmax_module],
                                              test_samples=test_samples*5, partition='test')
            test.test_one_shot(args, model=[enc_nn, metric_nn, softmax_module],
                               test_samples=test_samples, partition='train')
            enc_nn.train()
            metric_nn.train()

            if val_acc_aux is not None and val_acc_aux >= val_acc:
                test_acc = test_acc_aux
                val_acc = val_acc_aux

            if args.dataset == 'mini_imagenet':
                io.cprint("Best test accuracy {:.4f} \n".format(test_acc))

        ###################
        # Save model
        ###################
        if (batch_idx + 1) % args.save_interval == 0:
            torch.save(enc_nn, 'checkpoints/%s/models/enc_nn.t7' % args.exp_name)
            torch.save(metric_nn, 'checkpoints/%s/models/metric_nn.t7' % args.exp_name)

    # Test after training
    test.test_one_shot(args, model=[enc_nn, metric_nn, softmax_module],
                       test_samples=args.test_samples)
```

- 학습 결과 loss의 변화를 기록함
- 일정 iteration 마다 validation과 test를 수행함
  - test.test_one_shot : train_batch와 유사하고, loss 계산하는 부분이 없음
- 또한, train data에서의 성능도 계산하여 기록함

코드 분석 (models/models.py)

```python
class EmbeddingImagenet(nn.Module):
    ''' In this network the input image is supposed to be 28x28 '''

    def __init__(self, args, emb_size):
        super(EmbeddingImagenet, self).__init__()
        self.emb_size = emb_size
        self.ndf = 64
        self.args = args

        # Input 84x84x3
        self.conv1 = nn.Conv2d(3, self.ndf, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(self.ndf)

        # Input 42x42x64
        self.conv2 = nn.Conv2d(self.ndf, int(self.ndf*1.5), kernel_size=3, bias=False)
        self.bn2 = nn.BatchNorm2d(int(self.ndf*1.5))

        # Input 20x20x96
        self.conv3 = nn.Conv2d(int(self.ndf*1.5), self.ndf*2, kernel_size=3, padding=1, bias=False)
        self.bn3 = nn.BatchNorm2d(self.ndf*2)
        self.drop_3 = nn.Dropout2d(0.4)

        # Input 10x10x128
        self.conv4 = nn.Conv2d(self.ndf*2, self.ndf*4, kernel_size=3, padding=1, bias=False)
        self.bn4 = nn.BatchNorm2d(self.ndf*4)
        self.drop_4 = nn.Dropout2d(0.5)

        # Input 5x5x256
        self.fc1 = nn.Linear(self.ndf*4*5*5, self.emb_size, bias=True)
        self.bn_fc = nn.BatchNorm1d(self.emb_size)

    def forward(self, input):
        e1 = F.max_pool2d(self.bn1(self.conv1(input)), 2)
        x = F.leaky_relu(e1, 0.2, inplace=True)
        e2 = F.max_pool2d(self.bn2(self.conv2(x)), 2)
        x = F.leaky_relu(e2, 0.2, inplace=True)
        e3 = F.max_pool2d(self.bn3(self.conv3(x)), 2)
        x = F.leaky_relu(e3, 0.2, inplace=True)
        x = self.drop_3(x)
        e4 = F.max_pool2d(self.bn4(self.conv4(x)), 2)
        x = F.leaky_relu(e4, 0.2, inplace=True)
        x = self.drop_4(x)
        x = x.view(-1, self.ndf*4*5*5)
        output = self.bn_fc(self.fc1(x))

        return [e1, e2, e3, e4, None, output]
```

- 각 이미지에 대한 embedding을 계산하는 CNN 모델 (코드에서는 enc_nn이라고 부름)
- 각 이미지를 emb_size의 vector 하나로 embedding함

코드 분석 (models/models.py)

```python
class MetricNN(nn.Module):
    def __init__(self, args, emb_size):
        super(MetricNN, self).__init__()

        self.metric_network = args.metric_network
        self.emb_size = emb_size
        self.args = args

        if self.metric_network == 'gnn_iclr_nl':
            assert(self.args.train_N_way == self.args.test_N_way)
            num_inputs = self.emb_size + self.args.train_N_way
            if self.args.dataset == 'mini_imagenet':
                self.gnn_obj = gnn_iclr.GNN_nl(args, num_inputs, nf=96, J=1)
            elif 'omniglot' in self.args.dataset:
                self.gnn_obj = gnn_iclr.GNN_nl_omniglot(args, num_inputs, nf=96, J=1)
        elif self.metric_network == 'gnn_iclr_active':
            assert(self.args.train_N_way == self.args.test_N_way)
            num_inputs = self.emb_size + self.args.train_N_way
            self.gnn_obj = gnn_iclr.GNN_active(args, num_inputs, 96, J=1)
        else:
            raise NotImplementedError

    def gnn_iclr_forward(self, z, zi_s, labels_yi):
        # Creating WW matrix
        zero_pad = torch.zeros(labels_yi[0].size())
        if self.args.cuda:
            zero_pad = zero_pad.cuda()

        labels_yi = [zero_pad] + labels_yi
        zi_s = [z] + zi_s

        nodes = [torch.cat([zi, label_yi], 1) for zi, label_yi in zip(zi_s, labels_yi)]
        nodes = [node.unsqueeze(1) for node in nodes]
        nodes = torch.cat(nodes, 1)

        logits = self.gnn_obj(nodes).squeeze(-1)
        outputs = F.sigmoid(logits)

        return outputs, logits
```

- Embedding으로 표현된 각 image와 label 정보를 이용하여 node로 표현
- 전체 node의 feature를 GNN 모델의 input으로 넣어주면, output으로 test sample의 class에 대해 예측한 logit 값을 계산함

코드 분석 (models/gnn_iclr.py)

```python
class GNN_nl(nn.Module):
    def __init__(self, args, input_features, nf, J):
        super(GNN_nl, self).__init__()
        self.args = args
        self.input_features = input_features
        self.nf = nf
        self.J = J

        if args.dataset == 'mini_imagenet':
            self.num_layers = 2
        else:
            self.num_layers = 2

        for i in range(self.num_layers):
            if i == 0:
                module_w = Wcompute(self.input_features, nf, operator='J2', activation='softmax', ratio=[2, 2, 1, 1])
                module_l = Gconv(self.input_features, int(nf / 2), 2)
            else:
                module_w = Wcompute(self.input_features + int(nf / 2) * i, nf, operator='J2', activation='softmax', ratio=[2, 2, 1, 1])
                module_l = Gconv(self.input_features + int(nf / 2) * i, int(nf / 2), 2)
            self.add_module('layer_w{}'.format(i), module_w)
            self.add_module('layer_l{}'.format(i), module_l)

        self.w_comp_last = Wcompute(self.input_features + int(self.nf / 2) * self.num_layers, nf, operator='J2', activation='softmax', ratio=[2, 2, 1, 1])
        self.layer_last = Gconv(self.input_features + int(self.nf / 2) * self.num_layers, args.train_N_way, 2, bn_bool=False)

    def forward(self, x):
        W_init = torch.eye(x.size(1)).unsqueeze(0).repeat(x.size(0), 1, 1).unsqueeze(3)
        if self.args.cuda:
            W_init = W_init.cuda()

        for i in range(self.num_layers):
            Wi = self._modules['layer_w{}'.format(i)](x, W_init)

            x_new = F.leaky_relu(self._modules['layer_l{}'.format(i)]([Wi, x])[1])
            x = torch.cat([x, x_new], 2)

        Wl=self.w_comp_last(x, W_init)
        out = self.layer_last([Wl, x])[1]

        return out[:, 0, :]
```

- GNN 모델을 pytorch로 구현한 부분
- 각 GNN의 layer는 node feature를 이용하여 Wi(adjacency matrix)를 계산하고, 이를 이용하여 GraphConv를 계산하는 것으로 구성

코드 분석 (models/gnn_iclr.py)

```python
class Wcompute(nn.Module):
    def __init__(self, input_features, nf, operator='J2', activation='softmax', ratio=[2,2,1,1], num_operators=1, drop=False):
        super(Wcompute, self).__init__()
        self.num_features = nf
        self.operator = operator
        self.conv2d_1 = nn.Conv2d(input_features, int(nf * ratio[0]), 1, stride=1)
        self.bn_1 = nn.BatchNorm2d(int(nf * ratio[0]))
        self.drop = drop
        if self.drop:
            self.dropout = nn.Dropout(0.3)
        self.conv2d_2 = nn.Conv2d(int(nf * ratio[0]), int(nf * ratio[1]), 1, stride=1)
        self.bn_2 = nn.BatchNorm2d(int(nf * ratio[1]))
        self.conv2d_3 = nn.Conv2d(int(nf * ratio[1]), nf*ratio[2], 1, stride=1)
        self.bn_3 = nn.BatchNorm2d(nf*ratio[2])
        self.conv2d_4 = nn.Conv2d(nf*ratio[2], nf*ratio[3], 1, stride=1)
        self.bn_4 = nn.BatchNorm2d(nf*ratio[3])
        self.conv2d_last = nn.Conv2d(nf, num_operators, 1, stride=1)
        self.activation = activation

    def forward(self, x, W_id):
        W1 = x.unsqueeze(2)
        W2 = torch.transpose(W1, 1, 2) #size: bs x N x N x num_features
        W_new = torch.abs(W1 - W2) #size: bs x N x N x num_features
        W_new = torch.transpose(W_new, 1, 3) #size: bs x num_features x N x N

        W_new = self.conv2d_1(W_new)
        W_new = self.bn_1(W_new)
        W_new = F.leaky_relu(W_new)
        if self.drop:
            W_new = self.dropout(W_new)

        W_new = self.conv2d_2(W_new)
        W_new = self.bn_2(W_new)
        W_new = F.leaky_relu(W_new)

        W_new = self.conv2d_3(W_new)
        W_new = self.bn_3(W_new)
        W_new = F.leaky_relu(W_new)

        W_new = self.conv2d_4(W_new)
        W_new = self.bn_4(W_new)
        W_new = F.leaky_relu(W_new)

        W_new = self.conv2d_last(W_new)
        W_new = torch.transpose(W_new, 1, 3) #size: bs x N x N x 1

        if self.activation == 'softmax':
            W_new = W_new - W_id.expand_as(W_new) * 1e8
            W_new = torch.transpose(W_new, 2, 3)
            # Applying Softmax
            W_new = W_new.contiguous()
            W_new_size = W_new.size()
            W_new = W_new.view(-1, W_new.size(3))
            W_new = F.softmax(W_new)
            W_new = W_new.view(W_new_size)
            # Softmax applied
            W_new = torch.transpose(W_new, 2, 3)
        return W_new
```

- Node feature를 받아서 adjacency matri를 계산하는 layer
- Edge feature 값은 두개의 node feature 값의 차이 값에 fc layer를 여러 번 곱해주어서 계산
  - 여기서는 각각 다른 edge에 같은 fc layer를 이용하기 위해 Conv2d를 이용함
- 마지막에는 softmax 함수로 한 node에서 모든 edge의 값의 합이 1이 되도록 normalization 함
- 위 과정으로 adjacency matrix를 계산

코드 분석 (models/gnn_iclr.py)

```python
def gmul(input):
    W, x = input
    # x is a tensor of size (bs, N, num_features)
    # W is a tensor of size (bs, N, N, J)
    x_size = x.size()
    W_size = W.size()
    N = W_size[-2]
    W = W.split(1, 3)
    W = torch.cat(W, 1).squeeze(3) # W is now a tensor of size (bs, J*N, N)
    output = torch.bmm(W, x) # output has size (bs, J*N, num_features)
    output = output.split(N, 1)
    output = torch.cat(output, 2) # output has size (bs, N, J*num_features)
    return output


class Gconv(nn.Module):
    def __init__(self, nf_input, nf_output, J, bn_bool=True):
        super(Gconv, self).__init__()
        self.J = J
        self.num_inputs = J*nf_input
        self.num_outputs = nf_output
        self.fc = nn.Linear(self.num_inputs, self.num_outputs)

        self.bn_bool = bn_bool
        if self.bn_bool:
            self.bn = nn.BatchNorm1d(self.num_outputs)

    def forward(self, input):
        W = input[0]
        x = gmul(input) # out has size (bs, N, num_inputs)
        #if self.J == 1:
        #    x = torch.abs(x)
        x_size = x.size()
        x = x.contiguous()
        x = x.view(-1, self.num_inputs)
        x = self.fc(x) # has size (bs*N, num_outputs)

        if self.bn_bool:
            x = self.bn(x)

        x = x.view(*x_size[:-1], self.num_outputs)
        return W, x
```

- GraphConv layer를 pytorch로 구현한 코드
- Input으로 W (adjacency matrix)와 x (node feature)를 받음
- gmul 함수에서 adjacency operation (edge value를 이용하여 연결된 node의 feature를 weighted sum) 해줌
- Graph signa을 fc layer로 다음 layer의 dimension에 맞게 바꾸어 줌
- 새롭게 계산된 x (node feature)를 원래의 shape로 변환 후 return

코드 분석

- 기본적인 Graph convolutional layer를 pytorch로 구현하는 것은 많이 어렵지 않은 것으로 보임
- 따라서 어떤 상황에서 GCN을 사용하는 것이 도움이 될지에 대한 이해가 필요하다고 생각함
- 또한, graph의 구조와 node 및 edge feature 값을 정의하는 것이 매우 중요할 것으로 보임