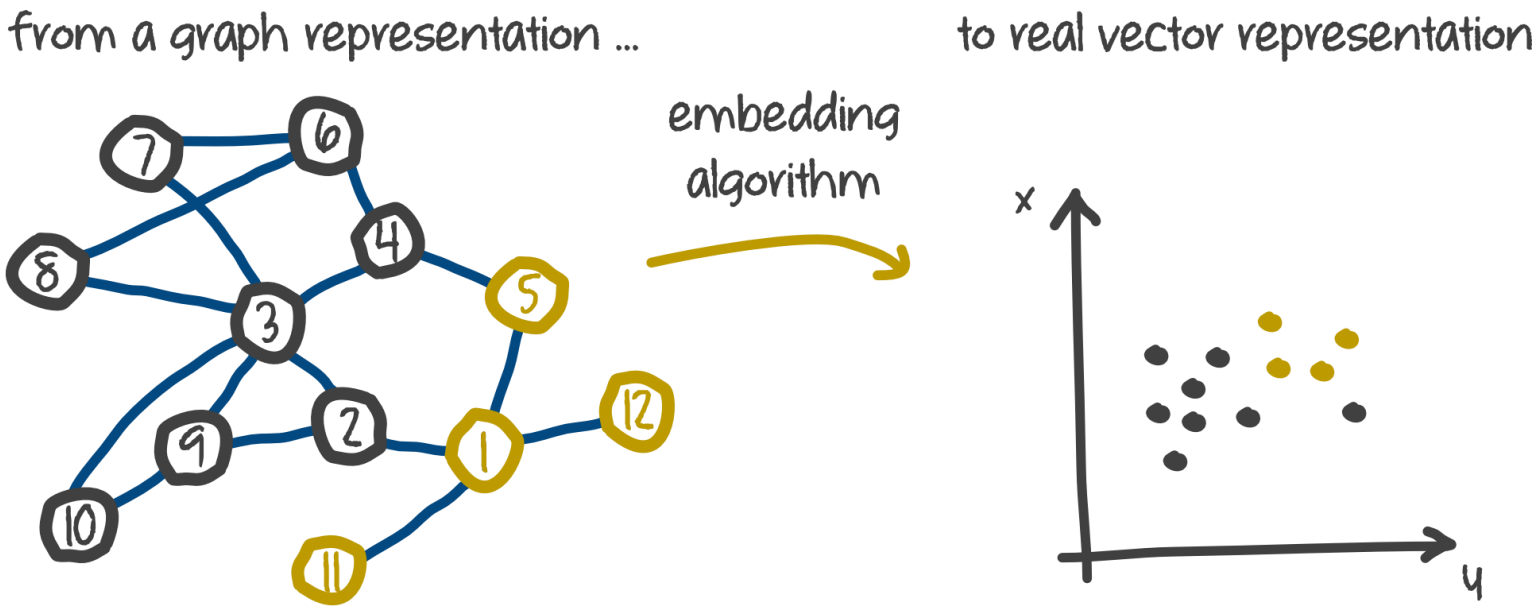
Adversarially Regularized Graph Autoencoder for Graph Embedding

Hyundo Lee

Seoul National University

Graph Embedding Method

- Graph Embedding
  - Converts graph data into a low dimensional feature space,
  - Preserving topological structure, vertex content
  - For classification, clustering, link prediction, …



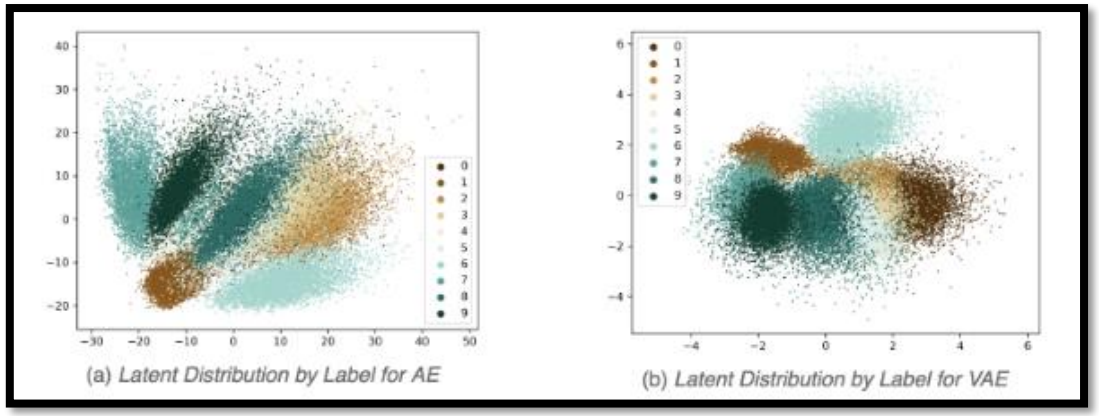
Graph Embedding Method

- Previous methods
  - Probabilistic models
  - Matrix factorization-based algorithms
  - Deep learning-based algorithms

- Typically unregularized approaches
  - Often learn a degenerate identity mapping
- Ignores the data distribution of the latent codes
- Poor representation in real-world data

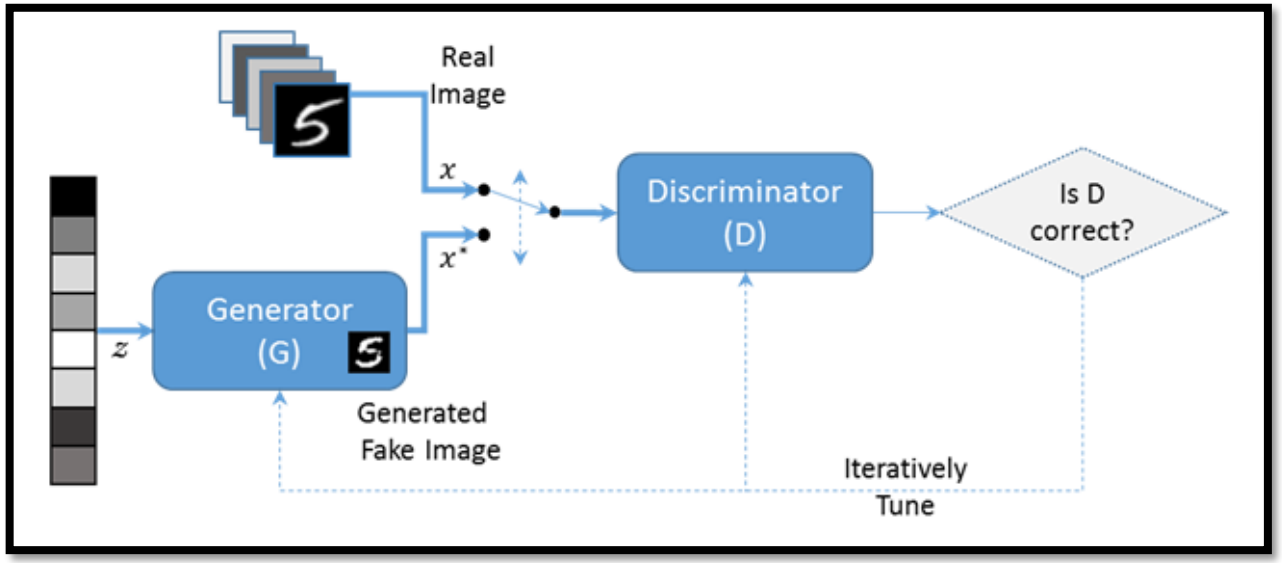
Graph Embedding Method with Prior Data Distribution

- Solutions
  - Regularization to the latent codes to follow prior data distribution
  - Generative adversarial based frameworks for robust latent representation.



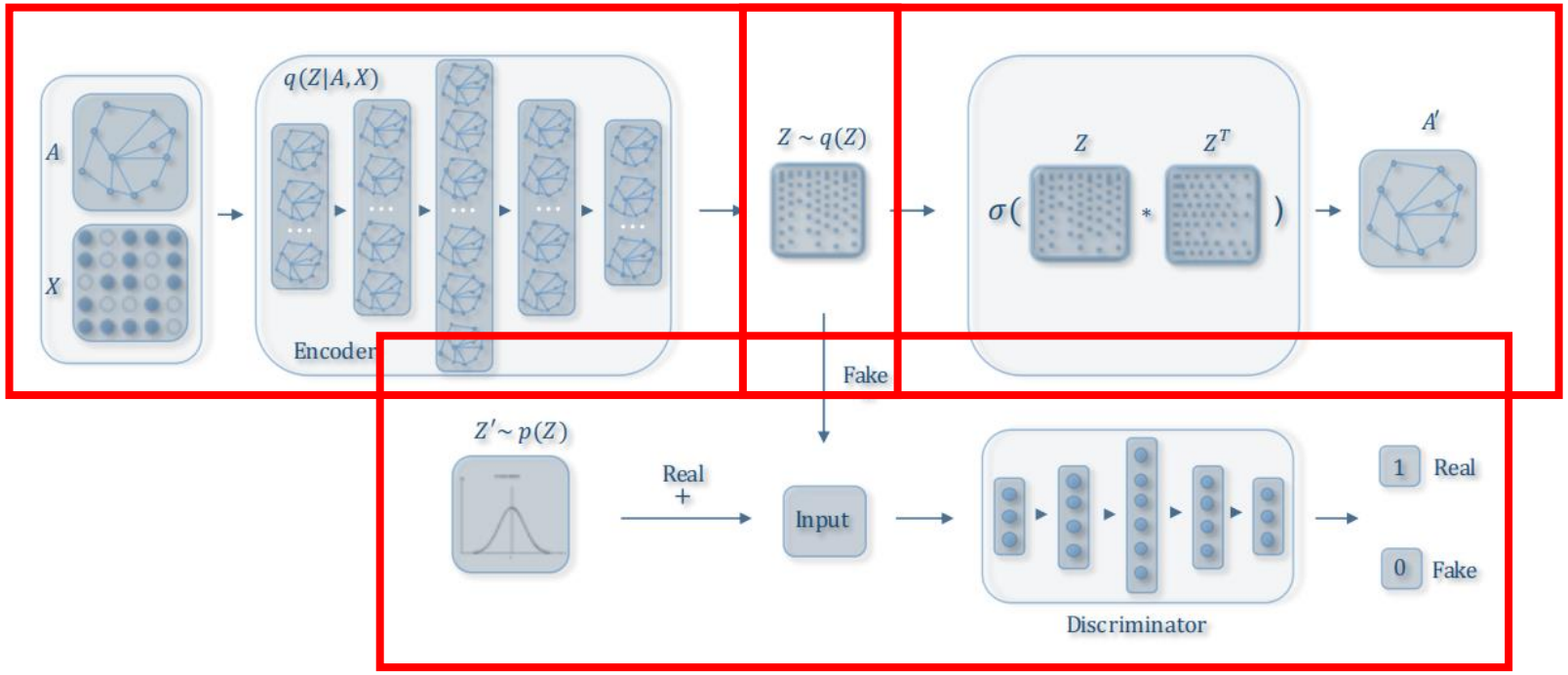
- ARGA: Adversarial Training scheme
- ARVGA: + using Variational Graph Encoder
  - Minimizing reconstruction errors + latent codes to a prior distribution

Brief Generative Adversarial Networks



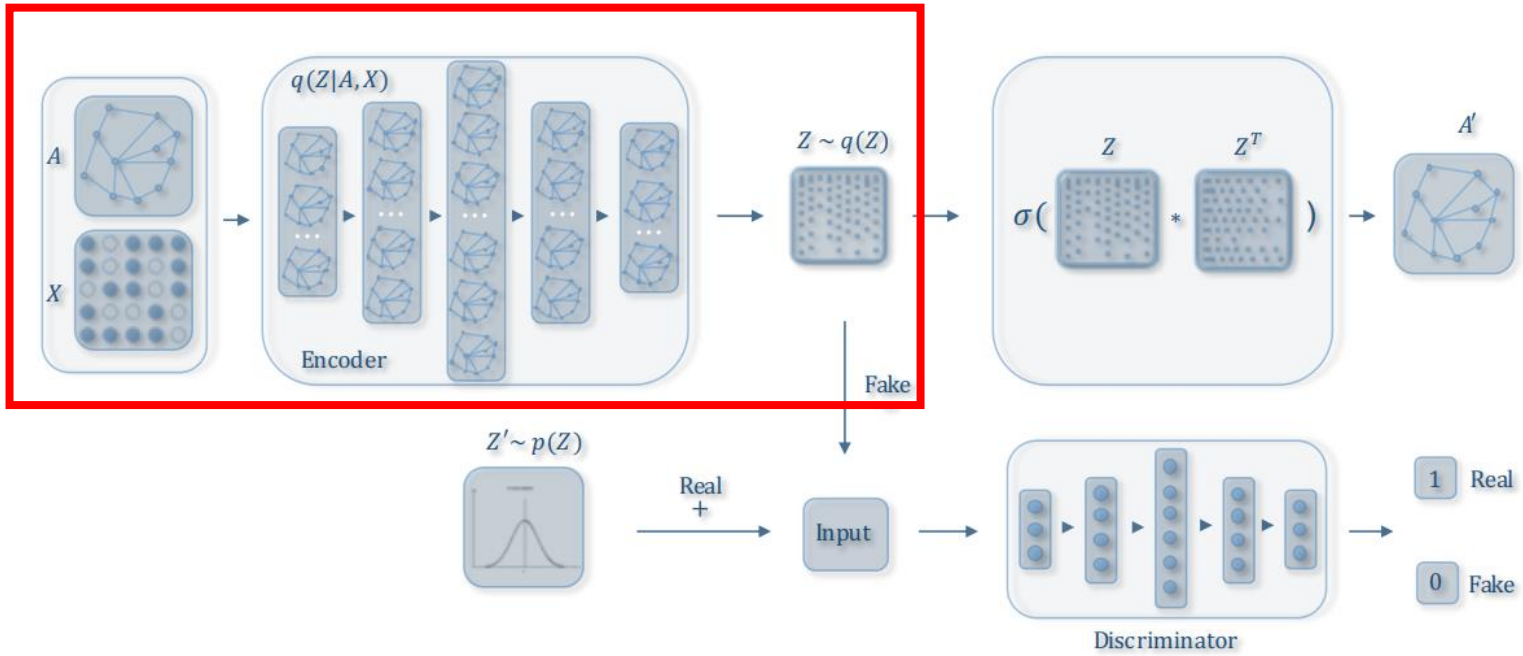
- Generator
  - Generates fake samples
  - Tries to fool the discriminator
- Discriminator
  - Tries to discriminate real/fake samples

Overall Framework



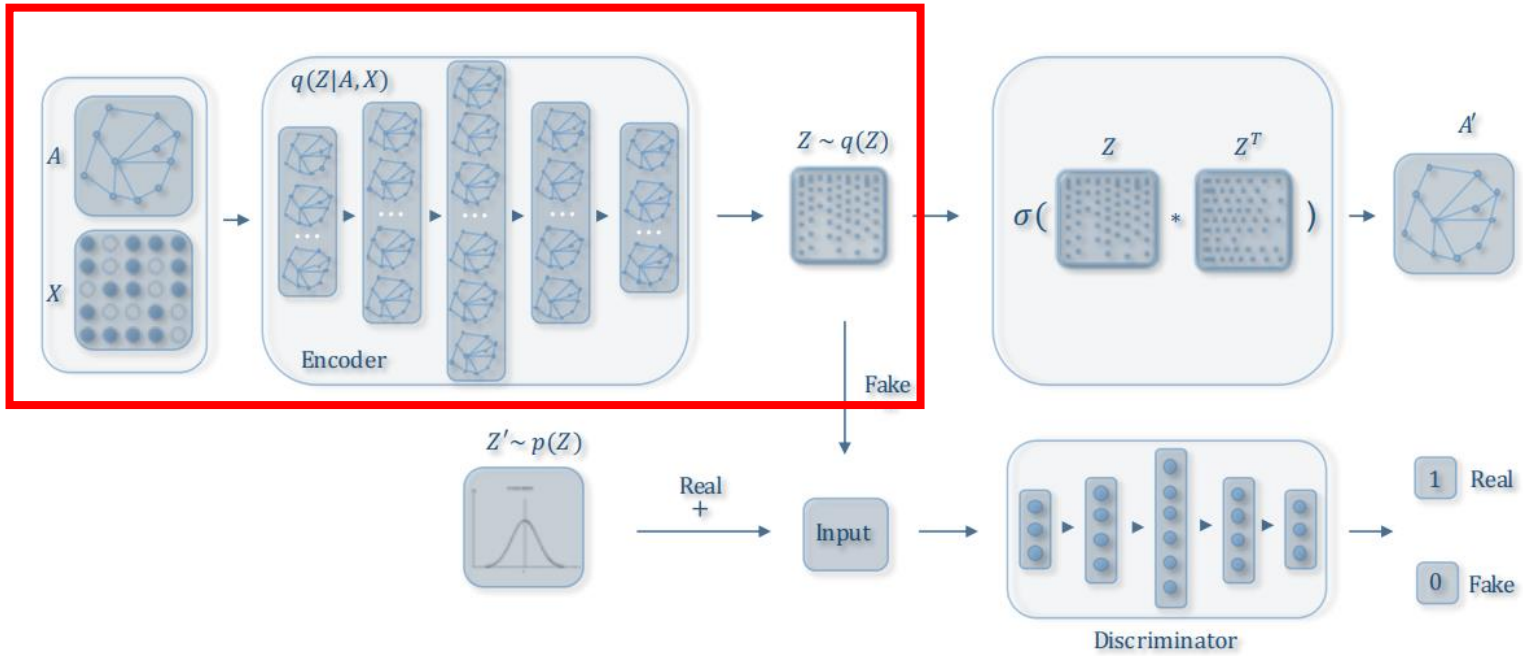
- Encoder q: $A \times X \rightarrow Z$ (or $\mu \times \sigma$)
- Decoder p: $Z \rightarrow \hat{A}$
- Discriminator D: $Z \rightarrow (0,1)$

Framework – Encoder



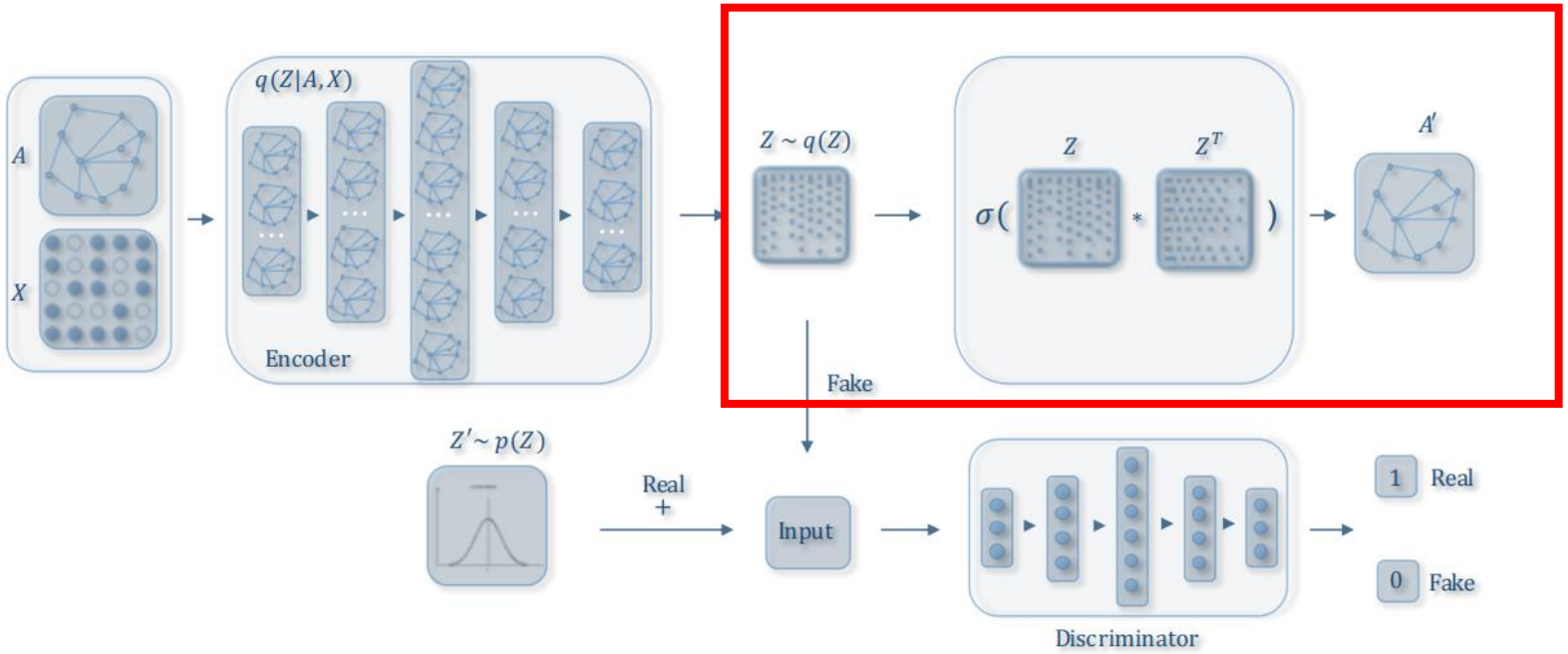
- Encoder: GCN
  - $\mathbf{Z}^{(l+1)} = f\left(\mathbf{Z}^{(l)}, \mathbf{A}\middle|\mathbf{W}^{(l)}\right) = \phi\left(\widetilde{\mathbf{D}}^{-1/2}\widetilde{\mathbf{A}}\widetilde{\mathbf{D}}^{-1/2}\mathbf{Z}^{(l)}\mathbf{W}^{(l)}\right)$
  - $\mathbf{Z}^{(0)} = \mathbf{X} \in \mathbb{R}^{n \times m}$
- 2-layered architecture
  - $\mathbf{Z}^{(1)} = f_{\text{Relu}}\left(\mathbf{X}, \mathbf{A}\middle|\mathbf{W}^{(0)}\right); \mathbf{Z}^{(2)} = f_{\text{linear}}\left(\mathbf{Z}^{(1)}, \mathbf{A}\middle|\mathbf{W}^{(1)}\right)$

Framework – Encoder



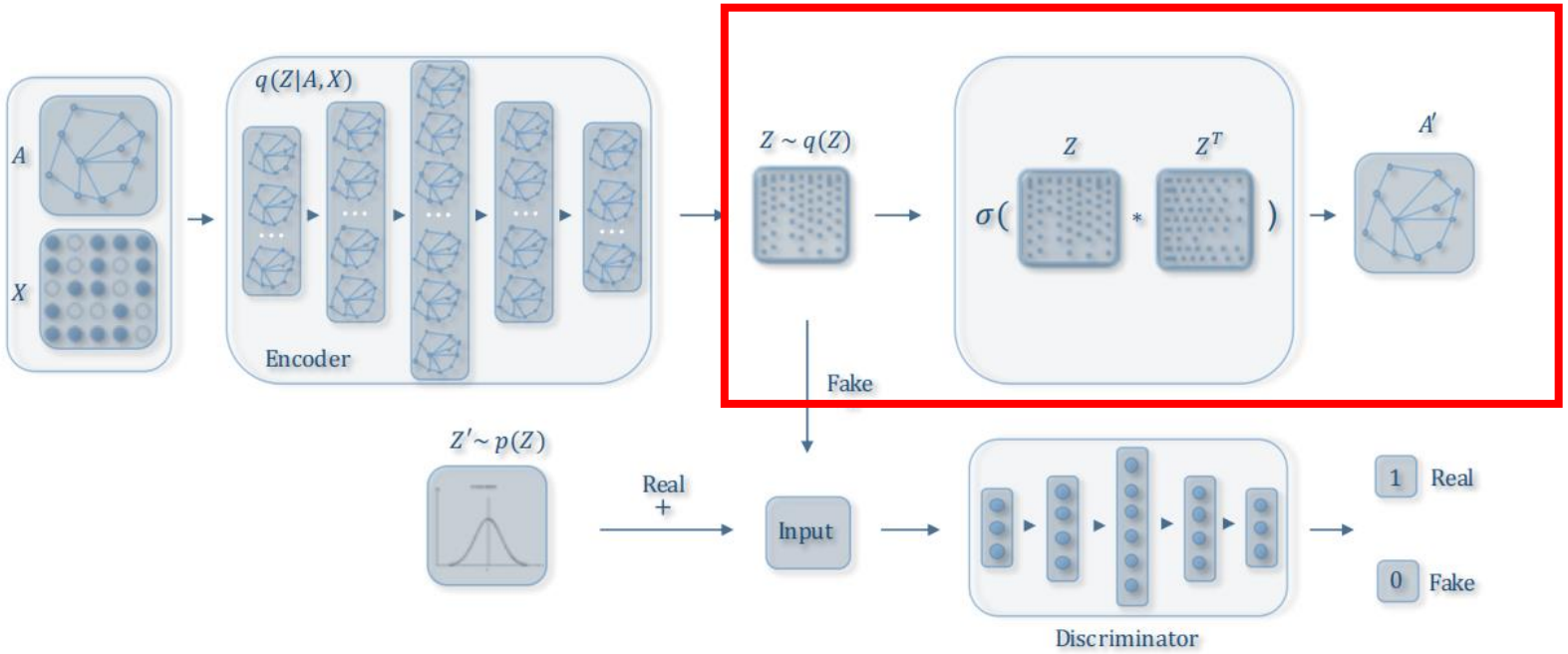
- Variational Graph Encoder: $Z^{(2)} = \mu, \log \sigma$
  - $\boldsymbol{\mu} = Z^{(2)} = f_{\text{linear}}\left(\mathbf{Z}^{(1)}, \mathbf{A} \middle| \mathbf{W}^{(1)}\right), \log \boldsymbol{\sigma} = f_{\text{linear}}\left(\mathbf{Z}^{(1)}, \mathbf{A} \middle| \mathbf{W}'^{(1)}\right)$
  - $q(\mathbf{Z}|\mathbf{X}, \mathbf{A}) = \prod_{i=1}^{n} q(\mathbf{z_i}|\mathbf{X}, \mathbf{A});\ q(\mathbf{z_i}|\mathbf{X}, \mathbf{A}) = N\left(\mathbf{z_i} \middle| \boldsymbol{\mu_i}, diag\left(\boldsymbol{\sigma}^{\mathbf{2}}\right)\right)$

Framework – Decoder



- Decoder
  - Can reconstruct $\mathbf{A}, \mathbf{X}$, or both.
  - Tries to reconstruct the topological structure from latent Z
    - Train a link prediction layer based on the graph embedding
  - $p\left(\widehat{\mathbf{A}}_{ij} = 1 \middle| \mathbf{z}_i, \mathbf{z}_j\right) = \text{sigmoid}(\mathbf{z}_i^T, \mathbf{z}_j)$, $\hat{A} = \text{sigmoid}\left(\mathbf{Z}\mathbf{Z}^{\mathrm{T}}\right)$

Framework – Decoder



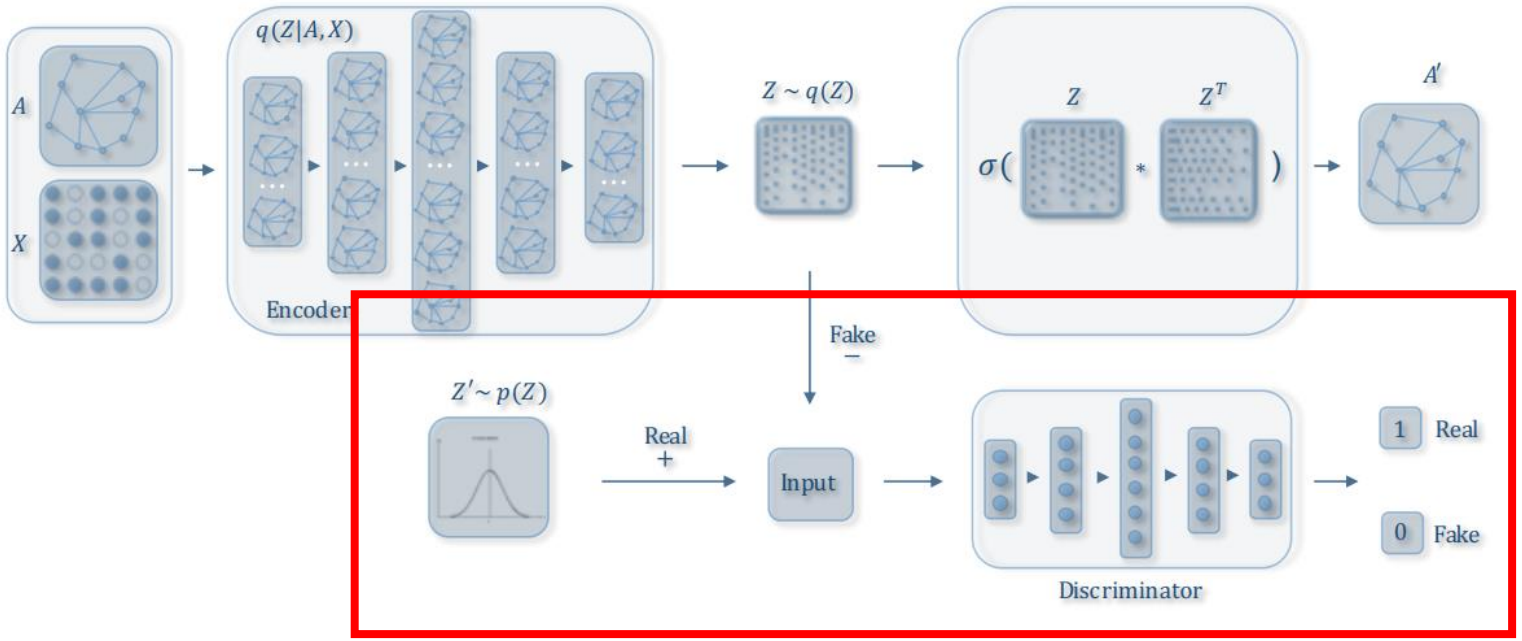
- Graph Autoencoder Model Optimization
  - For graph encoder,

$$\mathcal{L}_0 = \mathbb{E}_{q(\mathbf{Z}|(\mathbf{X},\mathbf{A}))}[\log p(\hat{\mathbf{A}}|\mathbf{Z})]$$

  - For variational graph encoder,

$$\mathcal{L}_1 = \mathbb{E}_{q(\mathbf{Z}|(\mathbf{X},\mathbf{A}))}[\log p(\hat{\mathbf{A}}|\mathbf{Z})] - \mathbf{KL}[q(\mathbf{Z}|\mathbf{X},\mathbf{A}) \parallel p(\mathbf{Z})]$$

Framework – Adversarial Model



- Discriminator: tries to discriminate
  - $Z' \sim p(Z) = N(0, I)$
  - $Z \sim G(X, A)$
- Encoder: tries to fool the discriminator
  - $G(X, A) \rightarrow N(0, I)$

$$\min_{\mathcal{G}} \max_{\mathcal{D}} \mathbb{E}_{\mathbf{z} \sim p_z}[\log \mathcal{D}(\mathbf{Z})] + \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})}[\log(1 - \mathcal{D}(\mathcal{G}(\mathbf{X}, \mathbf{A})))]$$

Overall Algorithm, Experiments

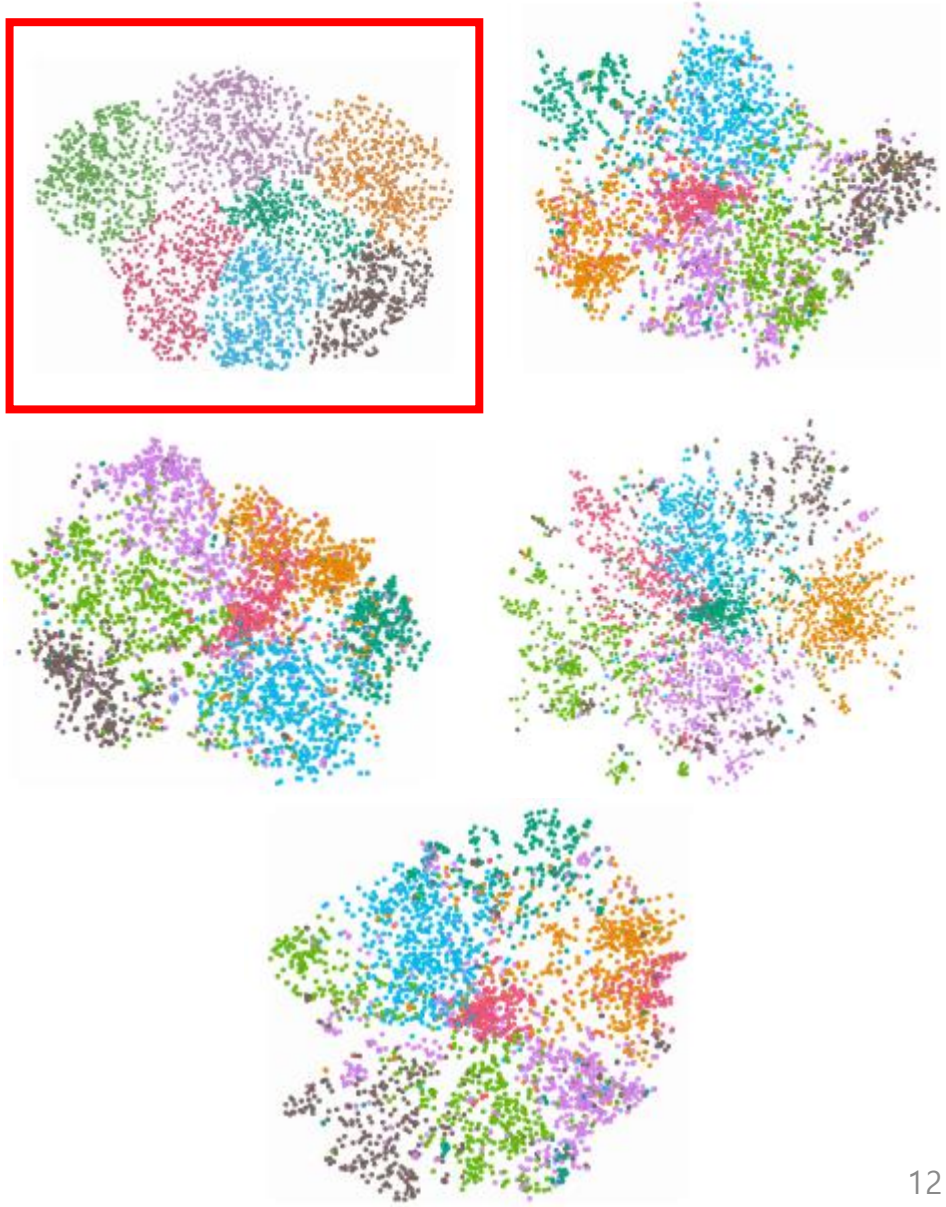
Algorithm 1 Adversarially Regularized Graph Embedding

Require:
$\mathbf{G} = \{\mathbf{V}, \mathbf{E}, \mathbf{X}\}$: a Graph with links and features;
T: the number of iterations;
K: the number of steps for iterating discriminator;
d: the dimension of the latent variable
Ensure: $\mathbf{Z} \in \mathbb{R}^{n \times d}$

1: **for** iterator = 1,2,3, $\cdots\cdots$, T **do**
2: Generate latent variables matrix $\mathbf{Z}$ through Eq.(4);
3: **for** k = 1,2, $\cdots\cdots$, K **do**
4: Sample m entities $\{\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}\}$ from latent matrix $\mathbf{Z}$
5: Sample m entities $\{\mathbf{a}^{(1)}, \ldots, \mathbf{a}^{(m)}\}$ from the prior distribution p_z
6: Update the discriminator with its stochastic gradient:

$$\nabla \frac{1}{m} \sum_{i=1}^{m} [\log \mathcal{D}(\mathbf{a}^i) + \log (1 - \mathcal{D}(\mathbf{z}^{(i)}))]$$

end for
7: Update the graph autoencoder with its stochastic gradient by Eq. (10) for ARGA or Eq. (11) for ARVGA;
end for
8: **return** $\mathbf{Z} \in \mathbb{R}^{n \times d}$



Pros & Cons

- Pros
  - Simple, intuitive approach for robust latent representation
  - Used generative model-based approach

- Cons
  - No theoretical evidence for using 2 approaches
  - Abuse of adversarial model – sufficient with using VAE

Report

코드 링크

- https://github.com/illhyhl1111/ARGA
  - (forked from https://github.com/Ruiqi-Hu/ARGA)

- 대부분 저자가 공개한 오픈소스 코드를 활용하였으며, run.py 및 visualization을 위한 코드를 추가하였고 그 외 몇몇 부분을 소폭 수정하였다.

코드 구조

- 코드의 전체 구성은 아래와 같다.

코드 구조

- 코드는 2가지 task – **clustering**과 **link prediction** – 중 하나의 task에 대하여 학습할 수 있으며, 그 외 **architecture**와 **dataset**을 선택하여 **run.py**의 argument로 넣어주어 실행한다.

```
parser = argparse.ArgumentParser(formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--dset', type=str, choices=['cora', 'citeseer', 'pubmed'], default='cora')
parser.add_argument('--arch', type=str, choices=['arga', 'arvga'], default='arga')
parser.add_argument('--task', type=str, choices=['clustering', 'link_prediction'], default='link_prediction')
parser.add_argument('--run_all', action='store_true', default=False)
args = parser.parse_args()
```

- settings.py로부터 flag를 통해 hyperparameter를 설정한 뒤, task의 종류에 따라 **Clustering_Runner** 혹은 **Link_pred_Runner** 인스턴스를 생성한 뒤 **erun()** 함수를 호출하여 학습을 수행한다.

```
else:
    setting_dict = settings.get_settings(args.dset, args.arch, args.task)

    if args.task == 'clustering':
        runner = Clustering_Runner(setting_dict)
    else:
        runner = Link_pred_Runner(setting_dict)

    runner.erun()
```

- 이후 **Link_pred_Runner**를 기준으로 코드를 설명한다. (**Clustering_Runner**도 비슷한 구조)

코드 구조

- **erun()** 함수는 전체 training 과정을 담당하는 함수이며, **format_data()** 함수를 통해 데이터를 읽어서 저장하며, **get_model()** 함수를 통해 ARGA(ARVGA) 모델을 생성하고, **get_optimizer()** 함수를 통해 optimizer를 생성한다.

```
def erun(self):
    model_str = self.model
    # formatted data
    feas = format_data(self.data_name)

    # Define placeholders
    placeholders = get_placeholder(feas['adj'])

    # construct model
    d_real, discriminator, ae_model = get_model(model_str, placeholders, feas['num_features'], feas['num_nodes'], feas['features_nonzero'])

    # Optimizer
    opt = get_optimizer(model_str, ae_model, discriminator, placeholders, feas['pos_weight'], feas['norm'], d_real, feas['num_nodes'])

    # Initialize session
    sess = tf.Session()
    sess.run(tf.global_variables_initializer())

    val_roc_score = []
```

- 이후 **update()** 함수를 통해 전체 dataset에 대하여 학습을 진행하고, 매 epoch마다 validation score와 10 epoch마다 test score를 출력한다.

```
# Train model
for epoch in range(self.iteration):

    emb, avg_cost = update(ae_model, opt, sess, feas['adj_norm'], feas['adj_label'], feas['features'], placeholders, feas['adj'])

    lm_train = linkpred_metrics(feas['val_edges'], feas['val_edges_false'])
    roc_curr, ap_curr, _ = lm_train.get_roc_score(emb, feas)
    val_roc_score.append(roc_curr)

    print("Epoch:", '%04d' % (epoch + 1), "train_loss=", "{:.5f}".format(avg_cost), "val_roc=", "{:.5f}".format(val_roc_score[-1]), "val_ap=", "{:.5

    if (epoch+1) % 10 == 0:
        lm_test = linkpred_metrics(feas['test_edges'], feas['test_edges_false'])
        roc_score, ap_score, _ = lm_test.get_roc_score(emb, feas)
        print('Test ROC score: ' + str(roc_score))
        print('Test AP score: ' + str(ap_score))
```

코드 구조

- **get_model()**, **get_optimizer()**, **update()** 함수는 모두 **constructer.py**에 정의되어 있으며, **get_model()**함수는 argument로 지정한 모델 구조에 따라 단순히 모델을 초기화하여 반환한다.

```python
def get_model(model_str, placeholders, num_features, num_nodes, features_nonzero):
    discriminator = Discriminator()
    d_real = discriminator.construct(placeholders['real_distribution'])
    model = None
    if model_str == 'arga':
        model = ARGA(placeholders, num_features, features_nonzero)

    elif model_str == 'arvga':
        model = ARVGA(placeholders, num_features, num_nodes, features_nonzero)

    return d_real, discriminator, model
```

- ARGA class는 인코더의 구조를 오른쪽과 같이 2개의 graph convolution layer를 가지도록 정의한다. 또한 decoder의 구조를 입력의 Inner product로 정의한다.

```python
class GraphConvolution(Layer):
    """Basic graph convolution layer for undirected graph without edge labels."""
    def __init__(self, input_dim, output_dim, adj, dropout=0., act=tf.nn.relu, **kwargs):
        super(GraphConvolution, self).__init__(**kwargs)
        with tf.variable_scope(self.name + '_vars'):
            self.vars['weights'] = weight_variable_glorot(input_dim, output_dim, name="weights")
        self.dropout = dropout
        self.adj = adj
        self.act = act

    def _call(self, inputs):
        x = inputs
        x = tf.nn.dropout(x, 1-self.dropout)
        x = tf.matmul(x, self.vars['weights'])
        x = tf.sparse_tensor_dense_matmul(self.adj, x)
        outputs = self.act(x)
        return outputs
```

```python
class InnerProductDecoder(Layer):
    """Decoder model layer for link prediction."""
    def __init__(self, input_dim, dropout=0., act=tf.nn.sigmoid, **kwargs):
        super(InnerProductDecoder, self).__init__(**kwargs)
        self.dropout = dropout
        self.act = act

    def _call(self, inputs):
        inputs = tf.nn.dropout(inputs, 1-self.dropout)
        x = tf.transpose(inputs)
        x = tf.matmul(inputs, x)
        x = tf.reshape(x, [-1])
        outputs = self.act(x)
        return outputs
```

```python
with tf.variable_scope('Encoder', reuse=None):
    self.hidden1 = GraphConvolutionSparse(input_dim=self.input_dim,
                                          output_dim=FLAGS.hidden1,
                                          adj=self.adj,
                                          features_nonzero=self.features_nonzero,
                                          act=tf.nn.relu,
                                          dropout=self.dropout,
                                          logging=self.logging,
                                          name='e_dense_1')(self.inputs)

    self.noise = gaussian_noise_layer(self.hidden1, 0.1)
    self.embeddings = GraphConvolution(input_dim=FLAGS.hidden1,
                                       output_dim=FLAGS.hidden2,
                                       adj=self.adj,
                                       act=lambda x: x,
                                       dropout=self.dropout,
                                       logging=self.logging,
                                       name='e_dense_2')(self.noise)


    self.z_mean = self.embeddings
    self.reconstructions = InnerProductDecoder(input_dim=FLAGS.hidden2,
                                       act=lambda x: x,
                                       logging=self.logging)(self.embeddings)
```

코드 구조

- Discriminator도 마찬가지로 2개의 fully connected layer를 가지도록 정의한다.

```
tf.set_random_seed(1)
dc_den1 = tf.nn.relu(dense(inputs, FLAGS.hidden2, FLAGS.hidden3, name='dc_den1'))
dc_den2 = tf.nn.relu(dense(dc_den1, FLAGS.hidden3, FLAGS.hidden1, name='dc_den2'))
output = dense(dc_den2, FLAGS.hidden1, 1, name='dc_output')
return output
```

- **get_optimizer()**에서는 모델 구조에 맞는 optimizer class를 선언하며, 초기화시 discriminator loss를 real과 fake distribution의 cross entropy로, generator(encoder) loss를 fake distribution의 cross entropy와 link prediction의 오차로 정의한다. 또한 각각에 맞는 optimizer를 정의한다.

```
class OptimizerAE(object):
    def __init__(self, preds, labels, pos_weight, norm, d_real, d_fake):
        preds_sub = preds
        labels_sub = labels

        self.real = d_real

        # Discriminator Loss
        self.dc_loss_real = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(labels=tf.ones_like(self.real), logits=self.real,name='dclreal'))

        self.dc_loss_fake = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(labels=tf.zeros_like(d_fake), logits=d_fake,name='dcfake'))
        self.dc_loss = self.dc_loss_fake + self.dc_loss_real

        # Generator loss
        generator_loss = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(labels=tf.ones_like(d_fake), logits=d_fake, name='gl'))
```

```
self.cost = norm * tf.reduce_mean(tf.nn.weighted_cross_entropy_with_logits(logits=preds_sub, targets=labels_sub, pos_weight=pos_weight))
self.generator_loss = generator_loss + self.cost
self.optimizer = tf.train.AdamOptimizer(learning_rate=FLAGS.learning_rate)  # Adam Optimizer

all_variables = tf.trainable_variables()
dc_var = [var for var in all_variables if 'dc_' in var.name]
en_var = [var for var in all_variables if 'e_' in var.name]


with tf.variable_scope(tf.get_variable_scope()):
    self.discriminator_optimizer = tf.train.AdamOptimizer(learning_rate=FLAGS.discriminator_learning_rate,
                                                          beta1=0.9, name='adam1').minimize(self.dc_loss, var_list=dc_var) #minimize(dc_loss r

    self.generator_optimizer = tf.train.AdamOptimizer(learning_rate=FLAGS.discriminator_learning_rate,
                                                      beta1=0.9, name='adam2').minimize(self.generator_loss, var_list=en_var)


self.opt_op = self.optimizer.minimize(self.cost)
self.grads_vars = self.optimizer.compute_gradients(self.cost)
```

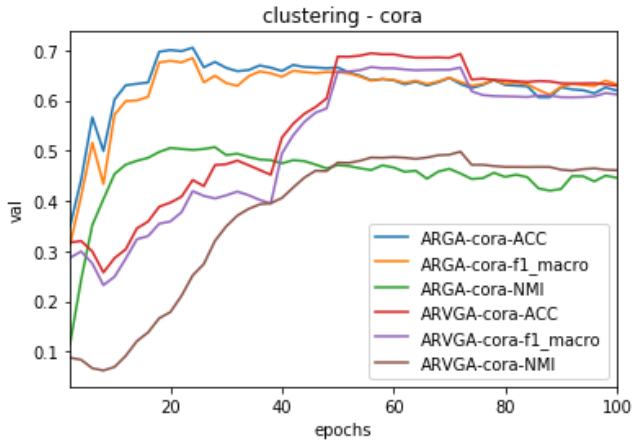
코드 구조

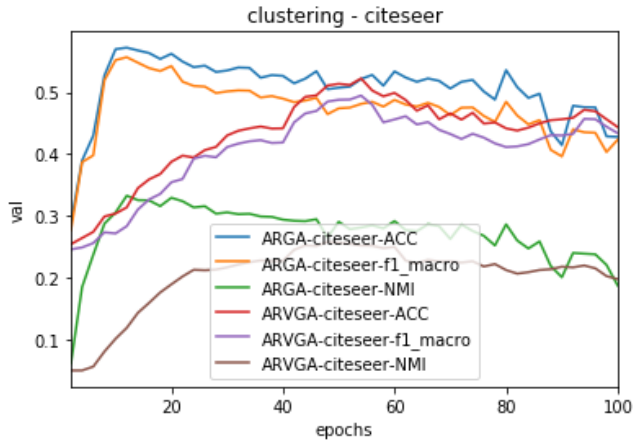
- **update()** 함수에서는 우선 인코딩을 통해 임베딩된 결과를 **emb**에 저장하고, 샘플링된 real sample을 **z_real_dist**에 저장한다. 그 이후 link prediction error를 통해 encoder를 5번 학습시키고, discriminator loss와 generator loss를 통해 discriminator와 encoder를 각각 1번씩 학습시킨다.

```python
def update(model, opt, sess, adj_norm, adj_label, features, placeholders, adj):
    # Construct feed dictionary
    feed_dict = construct_feed_dict(adj_norm, adj_label, features, placeholders)
    feed_dict.update({placeholders['dropout']: FLAGS.dropout})

    feed_dict.update({placeholders['dropout']: 0})
    emb = sess.run(model.z_mean, feed_dict=feed_dict)

    z_real_dist = np.random.randn(adj.shape[0], FLAGS.hidden2)
    feed_dict.update({placeholders['real_distribution']: z_real_dist})

    for j in range(5):
        _, reconstruct_loss = sess.run([opt.opt_op, opt.cost], feed_dict=feed_dict)
    d_loss, _ = sess.run([opt.dc_loss, opt.discriminator_optimizer], feed_dict=feed_dict)
    g_loss, _ = sess.run([opt.generator_loss, opt.generator_optimizer], feed_dict=feed_dict)

    avg_cost = reconstruct_loss

    return emb, avg_cost
```

실험 결과

- 2개의 dataset(**cora**, **citeseer**)에 대하여 실험하였고, 각각 2개의 task(**clustering**, **link_prediction**)과 2개의 architecture(**ARGA**, **ARVGA**)에 대하여 실험하였다.



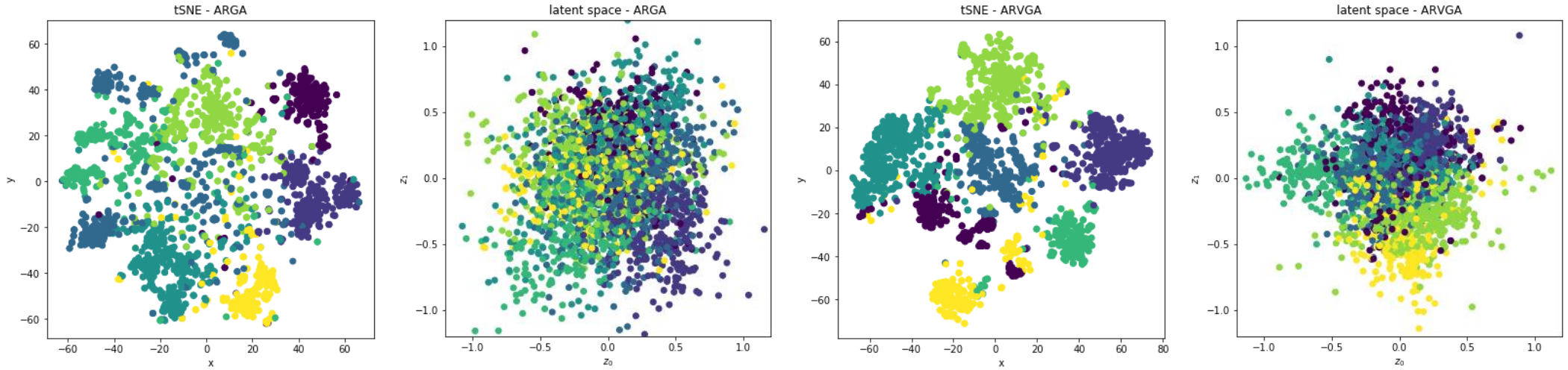


- Clustering task에 대한 실험 결과는 위와 같으며, 각각 3개의 metric(**ACC**: clustering label accuracy | **f1_macro**: precision과 recall에 대한 F1 score의 macro average | **NMI**: normalized mutual info score)으로 평가하였다.

- 실험 결과 학습 초반의 성능은 ARGA가 더 앞서지만, cora dataset의 경우 학습이 수렴하는 시점의 성능이 약간 더 앞선다. Citeseer dataset에서는 ARGA의 성능이 소폭 앞선다.

실험 결과

- 또한 cora 데이터셋에서 clustering task의 embedding 결과와 label prediction 결과를 저장하는 코드를 추가하여 latent space의 distribution을 살펴보았다.

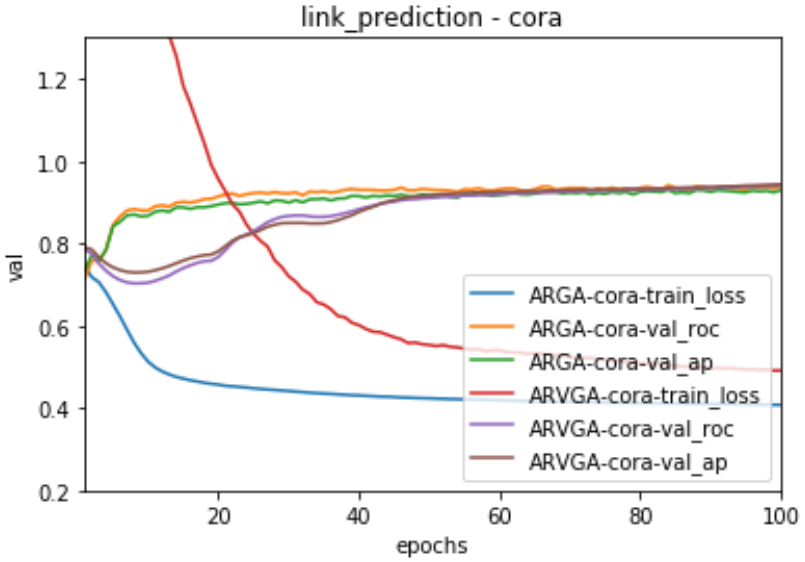


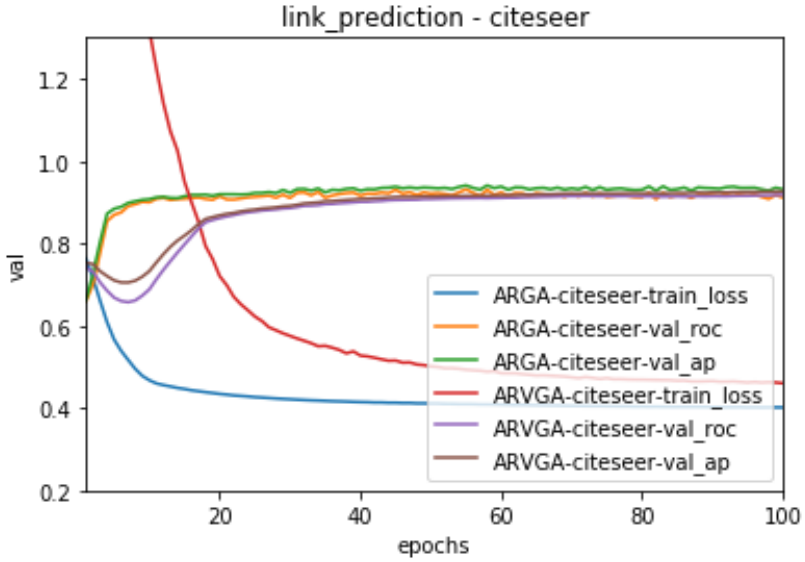
- 1, 3번째 그림은 전체 latent space에서의 embedding 결과를 나타내며, 2, 4번째 그림은 latent space의 1, 2번째 차원에 대하여 visualize한 결과이다. 또한 1, 2번째와 3, 4번째 그림은 각각 ARGA, ARVGA architecture로 실험한 결과이다.

- 실험 결과 두 architecture 모두 latent space의 각 차원이 unit gaussian distribution을 따르도록 분포가 학습되었으며, ARVGA의 embedding이 cluster의 분리가 더 잘 되었음을 확인할 수 있다.

실험 결과

- Link prediction에 대한 실험 결과는 아래와 같으며, 각각 3개의 metric(**train_loss | val_roc**: ROC curve의 AUC 넓이 | **val_ap**: average precision score)으로 평가하였다.
- 실험 결과 초반 학습은 ARGA가 더 안정적으로 학습하나, ARGA와 ARVGA 모두 같은 비슷한 성능으로 수렴하는 것을 확인할 수 있다.





실행 방법

- 코드 링크에 기재된 repository를 clone한 뒤 아래를 실행하여 dependencies를 설치한다.
  - **pip install -r requirements.txt**
- 그 이후 **ARGA/arga** 디렉토리로 들어가서 아래 명령어로 dataset, architecture, task를 지정하여 실행한다.
  - **python run.py [–-dset {cora, citeseer, pubmed}] [--arch {arga, arvga}] [--task {clustering, link_prediction}]**
- 아래 명령어 실행 시 모든 dset, arch, task에 대하여 학습이 진행된다.
  - **python run.py --run_all**

- 실험 결과는 **results/***, Visualization 코드는 **visualization.ipynb** 코드에서 확인할 수 있다.