Adversarially Regularized Graph Autoencoder

Sung-Hwan Kim

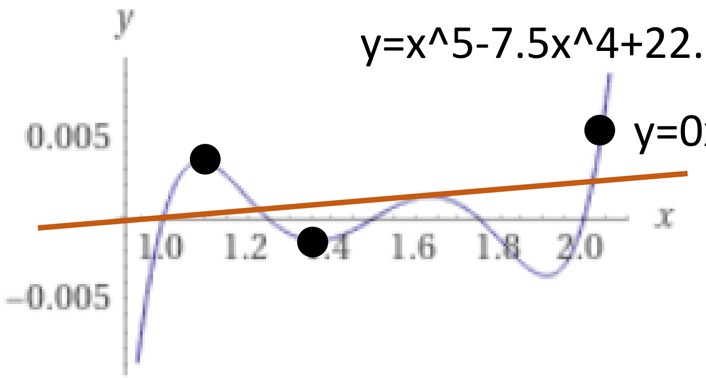
Regularization

- Methods to prevent OVERFITTING

1. Adding some constraints to objective function:
   - L1-, L2-regularization
   - Kullback-Leibler Divergence
2. Adding some (noisy) information to data/model:
   - Noise Layer
   - Dropout Layer
   - Batch Normalization
3. Early stopping
4. ...

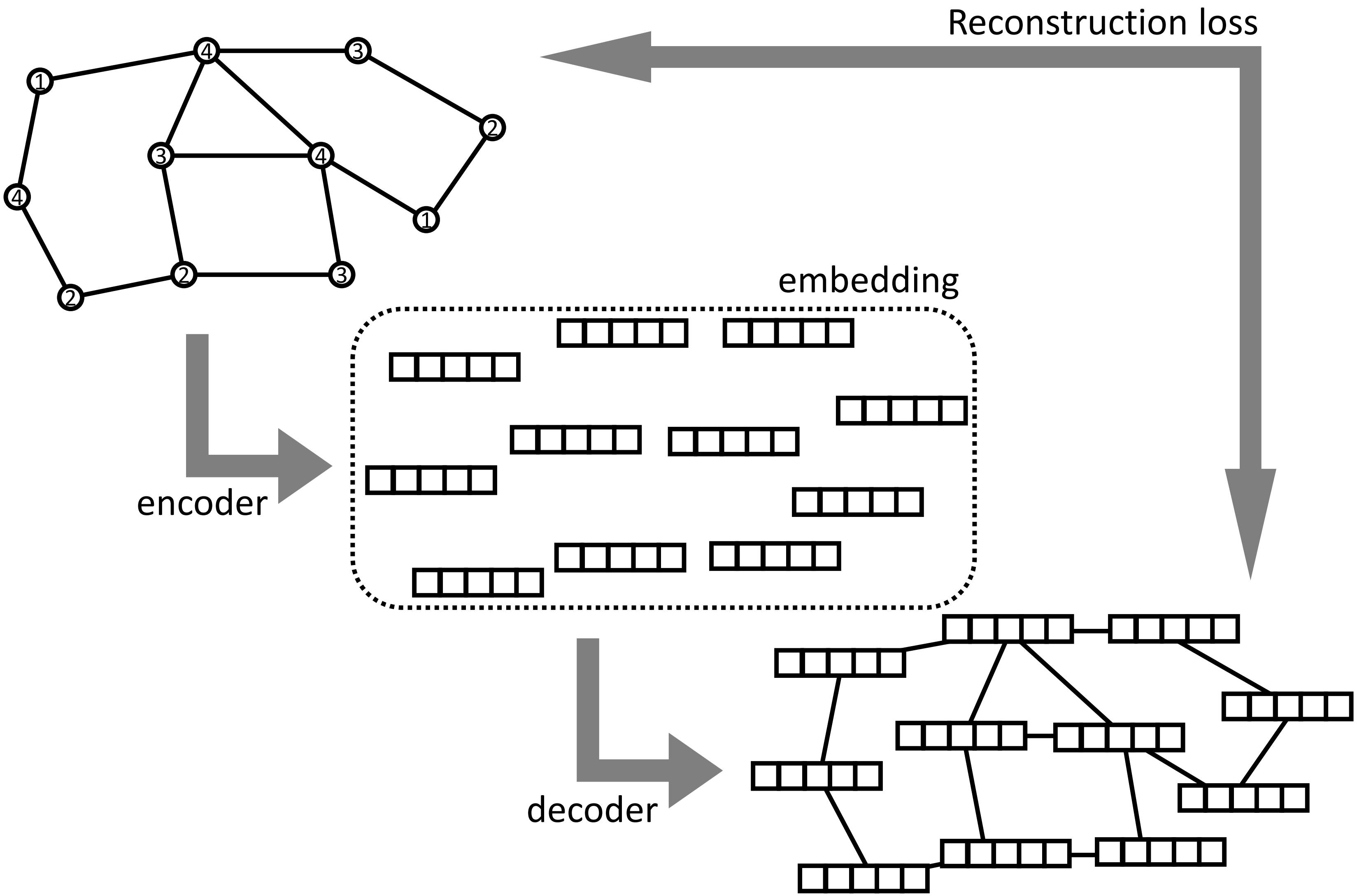
Regularization in Regression

- L1, L2 Regularization



- Basic Idea:
  - To make parameters as small as possible.

Graph Autoencoder



Regularization in Autoencoder

- We want the embeddings to follow a certain distribution (such as Gaussian).

1. KL-regularization?

2. Adversarial regularization
   - We make a discriminator to distinguish real embeddings from random embeddings (e.g. drawn from N(0,1)).
   - We train both encoder and discriminator adversarially.

Adversarially Regularized Graph Autoencoder

- Reconstruction loss:

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

  - X: input feature, A: graph structure
  - Z: embedding
  - Â: reconstructed graph structure: $\hat{\mathbf{A}} = \mathrm{sigmoid}(\mathbf{Z}\mathbf{Z}^\top)$

- Adversarial regularization

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

  - D(z): discriminator
    - Returns 1 if z is randomly generated, 0 otherwise.
  - G(X,A): encoder
    - Returns the embedding given X and A

Performance (described in the paper)

Approaches	Cora		Citeseer		PubMed	
	AUC	AP	AUC	AP	AUC	AP
SC	84.6 ± 0.01	88.5 ± 0.00	80.5 ± 0.01	85.0 ± 0.01	84.2 ± 0.02	87.8 ± 0.01
DW	83.1 ± 0.01	85.0 ± 0.00	80.5 ± 0.02	83.6 ± 0.01	84.4 ± 0.00	84.1 ± 0.00
GAE*	84.3 ± 0.02	88.1 ± 0.01	78.7 ± 0.02	84.1 ± 0.02	82.2 ± 0.01	87.4 ± 0.00
VGAE*	84.0 ± 0.02	87.7 ± 0.01	78.9 ± 0.03	84.1 ± 0.02	82.7 ± 0.01	87.5 ± 0.01
GAE	91.0 ± 0.02	92.0 ± 0.03	89.5 ± 0.04	89.9 ± 0.05	96.4 ± 0.00	96.5 ± 0.00
VGAE	91.4 ± 0.01	92.6 ± 0.01	90.8 ± 0.02	92.0 ± 0.02	94.4 ± 0.02	94.7 ± 0.02
ARGE	92.4 ± 0.003	$\mathbf{93.2 \pm 0.003}$	91.9 ± 0.003	93.0 ± 0.003	$\mathbf{96.8 \pm 0.001}$	$\mathbf{97.1 \pm 0.001}$
ARVGE	$\mathbf{92.4 \pm 0.004}$	92.6 ± 0.004	$\mathbf{92.4 \pm 0.003}$	$\mathbf{93.0 \pm 0.003}$	96.5 ± 0.001	96.8 ± 0.001

Table 2: Results for Link Prediction. GAE* and VGAE* are variants of GAE, which only explore topological structure, i.e., $\mathbf{X} = \mathbf{I}$.

Cora	Acc	NMI	F1	Precision	ARI
K-means	0.492	0.321	0.368	0.369	0.230
Spectral	0.367	0.127	0.318	0.193	0.031
GraphEncoder	0.325	0.109	0.298	0.182	0.006
DeepWalk	0.484	0.327	0.392	0.361	0.243
DNGR	0.419	0.318	0.340	0.266	0.142
RTM	0.440	0.230	0.307	0.332	0.169
RMSC	0.407	0.255	0.331	0.227	0.090
TADW	0.560	0.441	0.481	0.396	0.332
GAE	0.596	0.429	0.595	0.596	0.347
VGAE	0.609	0.436	0.609	0.609	0.346
ARGE	**0.640**	0.449	0.619	**0.646**	0.352
ARVGE	0.638	**0.450**	**0.627**	0.624	**0.374**

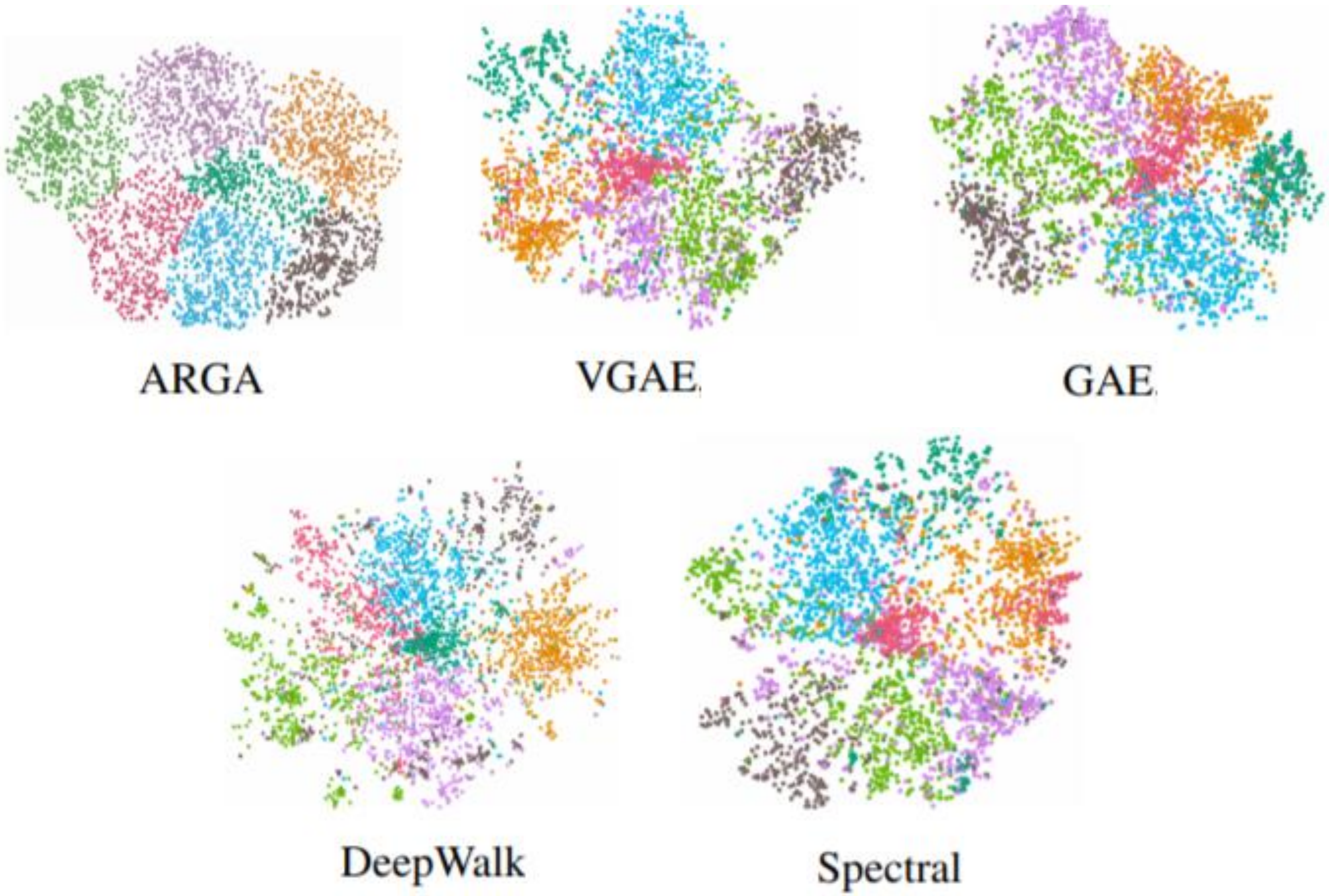
Table 3: Clustering Results on Cora

Citeseer	Acc	NMI	F1	Precision	ARI
K-means	0.540	0.305	0.409	0.405	0.279
Spectral	0.239	0.056	0.299	0.179	0.010
GraphEncoder	0.225	0.033	0.301	0.179	0.010
DeepWalk	0.337	0.088	0.270	0.248	0.092
DNGR	0.326	0.180	0.300	0.200	0.044
RTM	0.451	0.239	0.342	0.349	0.203
RMSC	0.295	0.139	0.320	0.204	0.049
TADW	0.455	0.291	0.414	0.312	0.228
GAE	0.408	0.176	0.372	0.418	0.124
VGAE	0.344	0.156	0.308	0.349	0.093
ARGE	**0.573**	**0.350**	**0.546**	**0.573**	**0.341**
ARVGE	0.544	0.261	0.529	0.549	0.245

Table 4: Clustering Results on Citeseer

Visualization (described in the paper)

- Dimension reduction with tSNE



ARGA in PyTorch Geometric

```
class Encoder(torch.nn.Module):
    def __init__(self, InputDim, HiddenDim, EmbeddingDim):
        super(Encoder, self).__init__()
        self.conv1 = GCNConv(InputDim , HiddenDim)
        self.conv2 = GCNConv(HiddenDim, EmbeddingDim)

    def forward(self, x, edge_index):
        x = F.relu( self.conv1(x, edge_index) )
        x = self.conv2(x, edge_index)
        return x
```

Encoder: 2-layer GCN

```
class Discriminator(torch.nn.Module):
    def __init__(self,EmbeddingDim,HiddenDim1,HiddenDim2):
        super(Discriminator, self).__init__()
        self.linear1 = Linear(EmbeddingDim, HiddenDim1)
        self.linear2 = Linear(HiddenDim1, HiddenDim2)
        self.linear3 = Linear(HiddenDim2, 1)

    def forward(self, x):
        x = F.relu( self.linear1(x) )
        x = F.relu( self.linear2(x) )
        x = self.linear3(x)
        x = x.squeeze(dim=1)
        return x
```

Discriminator: MLP

see 01-ARGA-link-prediction.py

```
encoder = Encoder(1433,32,32)
discriminator = Discriminator(32,64,32)

model = ARGA(encoder,discriminator).to(device)
```

Decoder: InnerProduct (default)

ARGA in PyTorch Geometric

```
model.train()
for i in range(200):
    optimizerE.zero_grad()
    optimizerD.zero_grad()

    Z = encoder( data.x, data.train_pos_edge_index )

    loss = model.discriminator_loss( Z )
    loss.backward()
    optimizerD.step()

    loss = model.recon_loss( Z, data.train_pos_edge_index )
    loss += model.reg_loss( Z )
    loss.backward()
    optimizerE.step()
```

training the discriminator

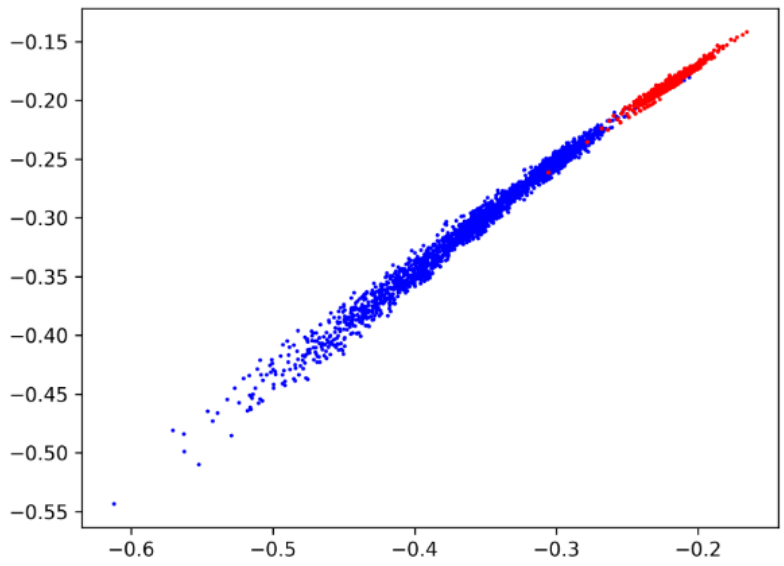
reconstruction loss

loss regarding the discriminator
(without this line, it is identical to a GAE)

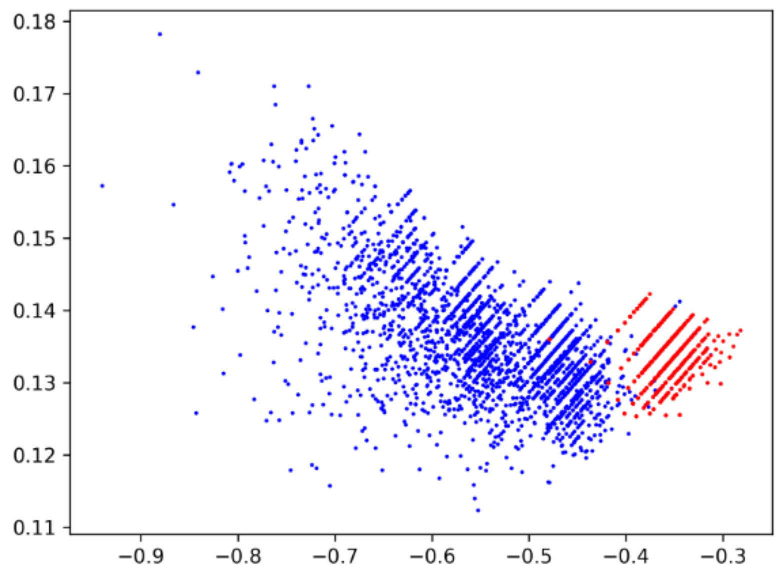
see 01-ARGA-link-prediction.py

GAE vs ARGA (자체 실험)

- Data: Synthetic random tree
- Embedding: 2D vector
- Color: Leaf/Nonleaf



GAE



ARGA

- see 02-ARGA-on-tree.py