1. Autoencoders

What does a deep neural network do?

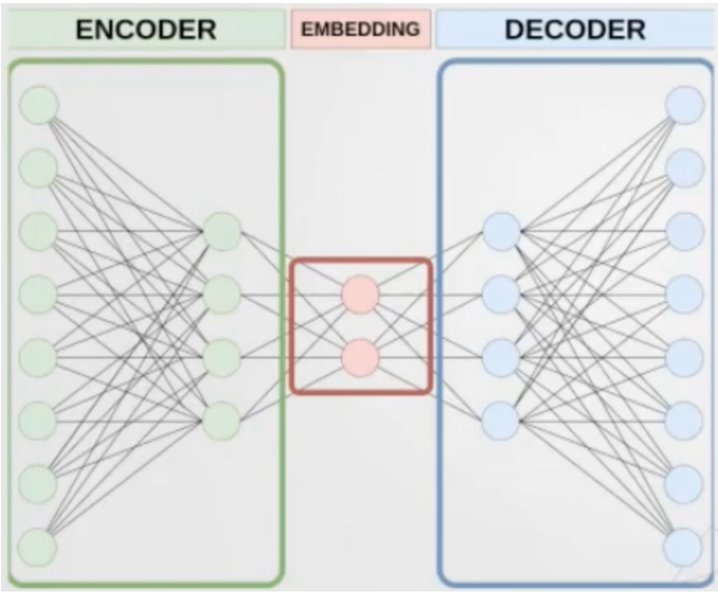
It learns important feature from the input.

Important features that allow to do specific task on the data (e.g. classification, regression, generalization.

Autoencoders are Neural networks that works in an unsupervised manner.

※ no need labeled data.

2. Autoencoder detail



⇑ maximize the similarity between input & output.

⇓ minimize the reconstruction error.

3. GAE theorey.

One convolutional Graph neural network:

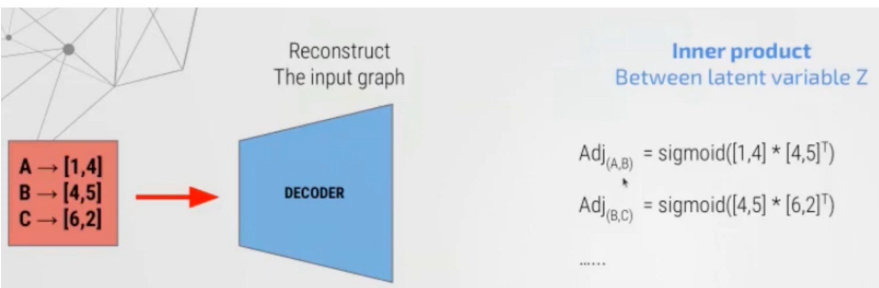
– produces a low dimentional embedding representation

$\bar{X} = GCN(A, X) = Relu(\tilde{A} X W_0)$.

with $\tilde{A} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$.

Encoder.

$Z = \bar{X}$ ⇒ Embedding Latent space.



use inner product to construct the adjacency matrix A.

Inner product between latent variable Z.

$\hat{A} = \text{logistic sigmoid}(ZZ^T)$

Jupyter Notebook:

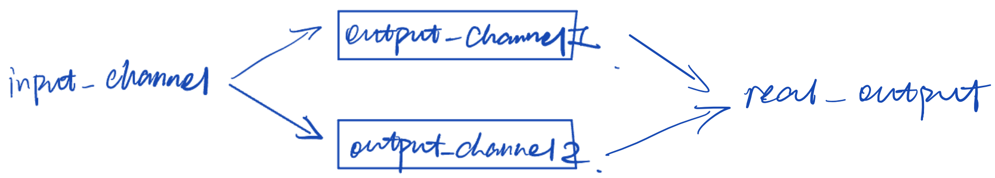
1

```
class GCNEncoder(torch.nn.Module):
  def __init__(self, in_channels, out_channels):
    super(GCNEncoder,self).__init__()
    self.conv1 = GCNConv(in_channels, 2*out_channels,cached = True)
    self.conv2 = GCNConv(2* out_channels, out_channels, cached = True)

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

⇒ initialize

→ transductive learning



2 train & test function.

```
def train():
  model.train()
  optimizer.zero_grad()
  z = model.encode(x,train_pos_edge_index)
  loss = model.recon_loss(z,train_pos_edge_index)
  loss.backward()
  optimizer.step()
  return float(loss)

def test(pos_edge_index, neg_edge_index):
  model.eval()
  with torch.no_grad():
    z = model.encode(x,train_pos_edge_index)
  return model.test(z, pos_edge_index, neg_edge_index)
```

optimizer.zero_grad() ⇒ clears old gradients from the last step o/w they will accumulate.

z = model.encode(...) ⇒ embedding the input data

loss = model.recon_loss(...) ⇒ calculate the reconstruct loss

loss.backward() ⇒ doing backpropagation to compute the gradient loss with respect to all model parameter

optimizer.step() ⇒ update the model parameter.

3. test the result.

```
for epoch in range(1, epochs+1):
  loss = train()

  auc, ap = test(data.test_pos_edge_index, data.test_neg_edge_index)
  print('Epoch: {:03d}, AUC: {:.4f}, AP: {:.4f}'.format(epoch,auc,ap))
```

4 Use tensorboard to compare the results rather than Auc & AP.

Store the results.

```
writer = SummaryWriter('runs/GAE_exprimemnt_'+'2d_100_epochs')
```

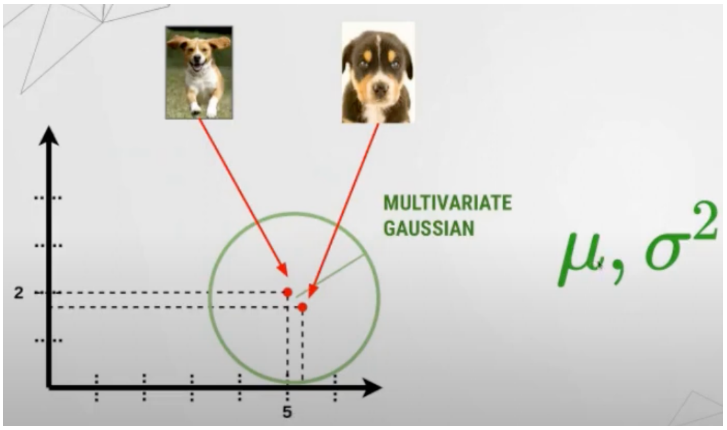
```
for epoch in range(1,epochs+1):
  loss = train()
  auc,ap = test(data.test_pos_edge_index, data.test_neg_edge_index)
  print('Epoch: {:03d}, AUC:{:.4f}, AP:{:.4f}'.format(epoch, auc, ap))

  writer.add_scalar('auc train',auc,epoch)
  writer.add_scalar('ap train',ap,epoch)
```

Then we can compare different results in Tensorboard.

different dim, different epochs.

4. variational Autoencoders



encoder ⇔ embedding ⇔ decoder

bayes.

$p(z|x)$

$q(x|z) = \frac{q(z|x)\,q(x)}{q(z)}$

Multivariate Gaussian.

☆ So we need to let $p(z|x)$ and $q(z|x)$

KL-Divergence ⇒ measures the distance between distribution.

$KL(q(z|x) \| p(z|x))$

$$\text{Loss} = -E_{p(z|x)} \log q(z|x) + KL(p(z|x) \| q(z))$$

Variational Lower Bound:

How well the network is able to reconstruct the input?

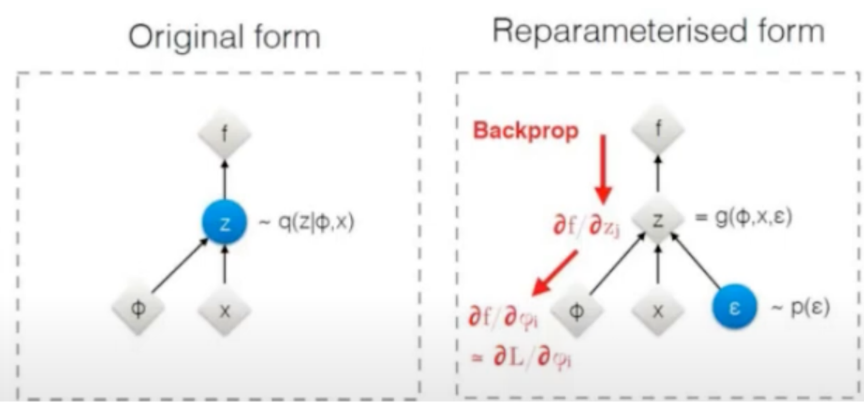
Regularizer.

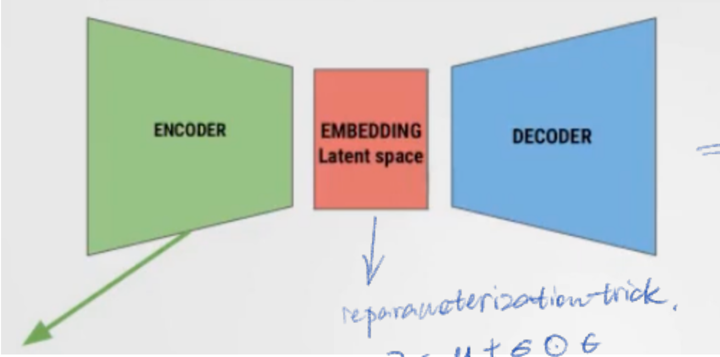
keep the distributions of $q(z|x)$ and $p(z|x)$ as much similar as possible.

(GVAE)

☆ Reparameterization trick.

$z = \mu + \sigma \odot \epsilon \qquad \epsilon \sim Norm(0,1)$





$\Rightarrow$ Still using inner product.

$\hat{A}$ = logistic sigmoid (ZZ^T).

reparameterization trick.

$z = \mu + \sigma \odot \epsilon$

$\epsilon \sim \text{Norm}(0,1)$

Two convolutional GNN.

GCN1: produces an lower dimensional embedding

$\bar{X} = GCN(A, X) = ReLU(\hat{A} X W_0)$ with $\hat{A} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$

GCN2: generates μ and $\log \sigma^2$.

$\mu = GCN_\mu(X, A) = \tilde{A} \bar{X} W_1$

$\log \sigma^2 = GCN_\sigma(X, A) = \tilde{A} \bar{X} W_1$.

Jupyter Notebook.

```
dataset = Planetoid("\..","CiteSeer",transform=T.NormalizeFeatures())
data = dataset[0]
data.train_mask = data.val_mask = data.test_mask = None
data = train_test_split_edges(data)

class VariationalGCNEncoder(torch.nn.Module):
  def __init__(self, in_channels, out_channels):
    super(VariationalGCNEncoder,self).__init__()
    self.conv1 = GCNConv(in_channels, 2*out_channels, cached = True)
    self.conv_mu = GCNConv(2*out_channels, out_channels, cached = True)
    self.conv_logstd = GCNConv(2*out_channels, out_channels, cached= True)

  def forward(self, x, edge_index):
    x = self.conv1(x,edge_index).relu()
    return self.conv_mu(x,edge_index),self.conv_logstd(x,edge_index)#returning two outputs
```

Should return μ and σ^2.

```
out_channels = 2
num_features = dataset.num_features
epochs = 300

#new line
model = VGAE(VariationalGCNEncoder(num_features,out_channels))

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
x = data.x.to(device)
train_pos_edge_index = data.train_pos_edge_index.to(device)
optimizer = torch.optim.Adam(model.parameters(),lr=0.1)
```

Apply VGAE model.

```
def train():
  model.train()
  optimizer.zero_grad()
  z = model.encode(x,train_pos_edge_index)
  loss = model.recon_loss(z,train_pos_edge_index)
  #new line
  loss = loss + (1/data.num_nodes)*model.kl_loss()
  loss.backward()
  optimizer.step()
  return float(loss)

def test(pos_edge_index, neg_edge_index):
  model.eval()
  with torch.no_grad():
    z = model.encode(x,train_pos_edge_index)
  return model.test(z, pos_edge_index, neg_edge_index)
```

Recall new loss function.

$\text{Loss} = -E_{p(z|x)} \log q(z|x) + \underbrace{KL(p(z|x) \| q(z))}_{\text{regularization part}}$.