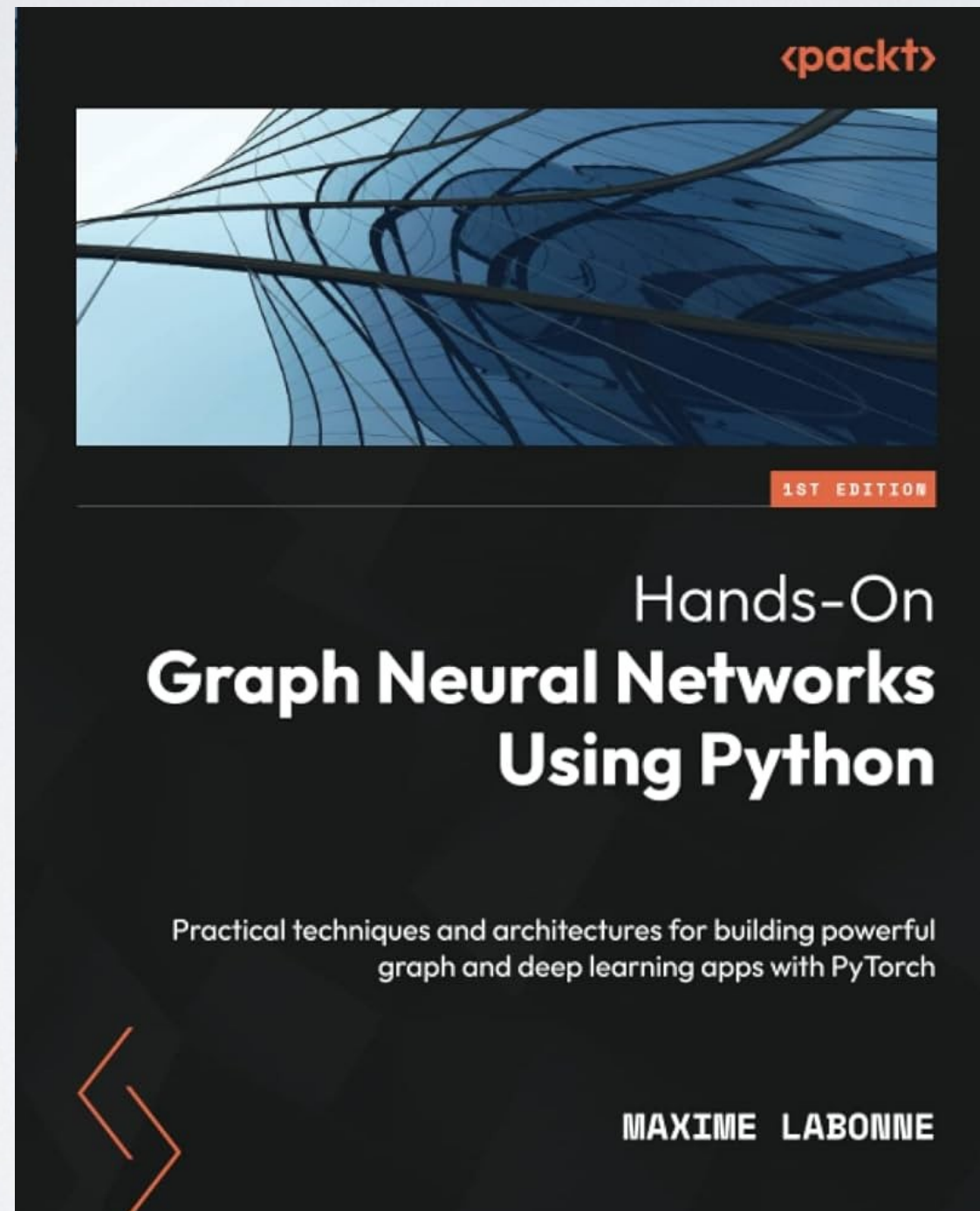


GNN WITH PYTORCH

REFERENCE



PYTORCH GEOMETRIC

- Pytorch Geometric (PyG) is an official library built on top of PyTorch to deal with graphs neural networks (and other types of structured data)
- <https://pytorch-geometric.readthedocs.io/>



PYTORCH GEOMETRIC

Implements most
Well-known GNN
layers

Graph Neural Network Operators

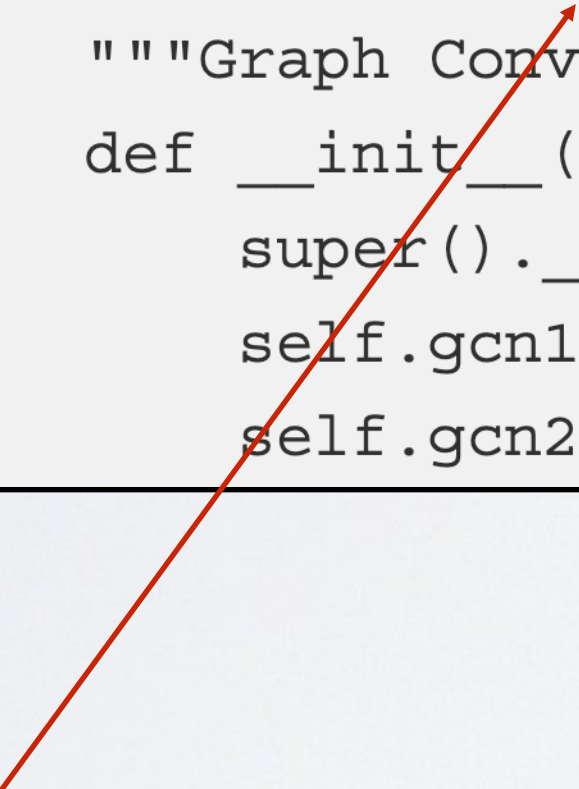
Name	SparseTensor	edge_weight	edge_attr	bipartite	static	lazy
SimpleConv	✓	✓		✓	✓	
GCNConv (Paper)	✓	✓			✓	✓
ChebConv (Paper)		✓			✓	✓
SAGEConv (Paper)	✓			✓	✓	✓
CuGraphSAGEConv (Paper)					✓	
GraphConv (Paper)	✓	✓		✓	✓	✓
GatedGraphConv (Paper)	✓	✓			✓	
ResGatedGraphConv (Paper)	✓		✓	✓	✓	✓
GATConv (Paper)	✓		✓	✓		✓
CuGraphGATConv (Paper)					✓	
FusedGATConv (Paper)					✓	
GATv2Conv (Paper)	✓		✓	✓		✓
TransformerConv (Paper)	✓		✓	✓		✓
AGNNConv (Paper)	✓				✓	
TAGConv (Paper)	✓	✓			✓	✓
GINConv (Paper)	✓			✓	✓	
GINEConv (Paper)	✓		✓	✓	✓	

PYTORCH GEOMETRIC

```
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv
```

PYTORCH GEOMETRIC

```
class GCN(torch.nn.Module):  
    """Graph Convolutional Network"""  
    def __init__(self, dim_in, dim_h, dim_out):  
        super().__init__()  
        self.gcn1 = GCNConv(dim_in, dim_h)  
        self.gcn2 = GCNConv(dim_h, dim_out)
```



Creating a torch module

PYTORCH GEOMETRIC

```
class GCN(torch.nn.Module):  
    """Graph Convolutional Network"""  
    def __init__(self, dim_in, dim_h, dim_out):  
        super().__init__()  
        self.gcn1 = GCNConv(dim_in, dim_h)  
        self.gcn2 = GCNConv(dim_h, dim_out)
```

```
def forward(self, x, edge_index):  
    h = self.gcn1(x, edge_index)  
    h = torch.relu(h)  
    h = self.gcn2(h, edge_index)  
    return F.log_softmax(h, dim=1)
```

Adjacency matrix (A)
As an edge list (2 x N matrix)

PYTORCH GEOMETRIC

```
class GCN(torch.nn.Module):  
    """Graph Convolutional Network"""  
    def __init__(self, dim_in, dim_h, dim_out):  
        super().__init__()  
        self.gcn1 = GCNConv(dim_in, dim_h)  
        self.gcn2 = GCNConv(dim_h, dim_out)
```

```
def forward(self, x, edge_index):  
    h = self.gcn1(x, edge_index)  
    h = torch.relu(h)  
    h = self.gcn2(h, edge_index)  
    return F.log_softmax(h, dim=1)
```

```
def fit(self, data, epochs):  
    criterion = torch.nn.CrossEntropyLoss()  
    optimizer = torch.optim.Adam(self.parameters(),  
                                  lr=0.01,  
                                  weight_decay=5e-4)  
    self.train()  
    for epoch in range(epochs+1):  
        optimizer.zero_grad()  
        out = self(data.x, data.edge_index)  
        loss = criterion(out[data.train_mask],  
                          data.y[data.train_mask])  
        acc = accuracy(out[data.train_mask],  
                        data.y[data.train_mask])  
        loss.backward()  
        optimizer.step()
```

Module in training
mode
Reset gradient at each
step
call forward

Mask to use only some
nodes for training

`train_mask = [True, True, True, False, False]`

PYTORCH GEOMETRIC

Test function

No gradient computation in this
function

```
@torch.no_grad()  
def test(self, data):  
    self.eval()  
    out = self(data.x, data.edge_index)  
    acc = accuracy(out.argmax(dim=1)[data.test_mask],  
data.y[data.test_mask])  
    return acc
```

Module in evaluation
mode

PYTORCH GEOMETRIC

```
class GCN(torch.nn.Module):  
    """Graph Convolutional Network"""  
    def __init__(self, dim_in, dim_h, dim_out):  
        super().__init__()  
        self.gcn1 = GCNConv(dim_in, dim_h)  
        self.gcn2 = GCNConv(dim_h, dim_out)
```

```
def fit(self, data, epochs):  
    criterion = torch.nn.CrossEntropyLoss()  
    optimizer = torch.optim.Adam(self.parameters(),  
                                  lr=0.01,  
                                  weight_decay=5e-4)  
  
    self.train()  
    for epoch in range(epochs+1):  
        optimizer.zero_grad()  
        out = self(data.x, data.edge_index)  
        loss = criterion(out[data.train_mask],  
                          data.y[data.train_mask])  
        acc = accuracy(out[data.train_mask].  
                        argmax(dim=1), data.y[data.train_mask])  
        loss.backward()  
        optimizer.step()
```

```
gcn = GCN(dataset.num_features, 16, dataset.num_classes)  
print(gcn)  
gcn.fit(data, epochs=100)
```

A MORE ADVANCED EXAMPLE

EXAMPLE 2

```
class GCN(torch.nn.Module):  
    def __init__(self, dim_in, dim_h, dim_out):  
        super().__init__()  
        self.gcn1 = GCNConv(dim_in, dim_h*4)  
        self.gcn2 = GCNConv(dim_h*4, dim_h*2)  
        self.gcn3 = GCNConv(dim_h*2, dim_h)  
        self.linear = torch.nn.Linear(dim_h, dim_out)
```

EXAMPLE 2

```
class GCN(torch.nn.Module):  
    def __init__(self, dim_in, dim_h, dim_out):  
        super().__init__()  
        self.gcn1 = GCNConv(dim_in, dim_h*4)  
        self.gcn2 = GCNConv(dim_h*4, dim_h*2)  
        self.gcn3 = GCNConv(dim_h*2, dim_h)  
        self.linear = torch.nn.Linear(dim_h, dim_out)
```

```
def forward(self, x, edge_index):  
    h = self.gcn1(x, edge_index)  
    h = torch.relu(h)  
    h = F.dropout(h, p=0.5, training=self.training)  
    h = self.gcn2(h, edge_index)  
    h = torch.relu(h)  
    h = F.dropout(h, p=0.5, training=self.training)  
    h = self.gcn3(h, edge_index)  
    h = torch.relu(h)  
    h = self.linear(h)  
    return h
```

EXAMPLE 2

```
h = F.dropout(h, p=0.5, training=self.training)
```

- Dropout: Implicit Regularization technique
 - Lasso: explicit regularization
 - Dropout: implicit
 - We do not control how to penalize weights
- Principe: put a fraction of input matrix elements to 0.
 - Hide some information about some nodes
- It makes overfitting harder
 - The NN cannot rely too much on one particular information piece
- `training=self.training`
 - We want to do it only during training, not testing!

VGAE

VGAE

```
from torch_geometric.nn import GCNConv, VGAE
```

VGAE reconstructs the graph using the dot product of embeddings.

Any encoder can be used to obtain the embeddings.
Remember: end-to-end, all weights trained simultaneously!

```
model = VGAE(Encoder(dataset.num_features, 16)).  
to(device)  
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

VGAE

```
class Encoder(torch.nn.Module):
    def __init__(self, dim_in, dim_out):
        super().__init__()
        self.conv1 = GCNConv(dim_in, 2 * dim_out)
        self.conv_mu = GCNConv(2 * dim_out, dim_out)
        self.conv_logstd = GCNConv(2 * dim_out, dim_out)

    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)
```

Our encoder computes and returns 2 objects: centroids and variance

VGAE

```
def train():  
    model.train()  
    optimizer.zero_grad()  
    z = model.encode(train_data.x, train_data.edge_index)  
    loss = model.recon_loss(z, train_data.pos_edge_label_  
index) + (1 / train_data.num_nodes) * model.kl_loss()  
    loss.backward()  
    optimizer.step()  
    return float(loss)
```

VGAE traditionally uses a combination of 2 losses:

- recon_loss: binary cross entropy

- KL divergence between parameters and priors

=>priors are that centers are at 0 and variance=1, tend to force to concentrate in the center

GRAPH ATTENTION NETWORK (GAT)

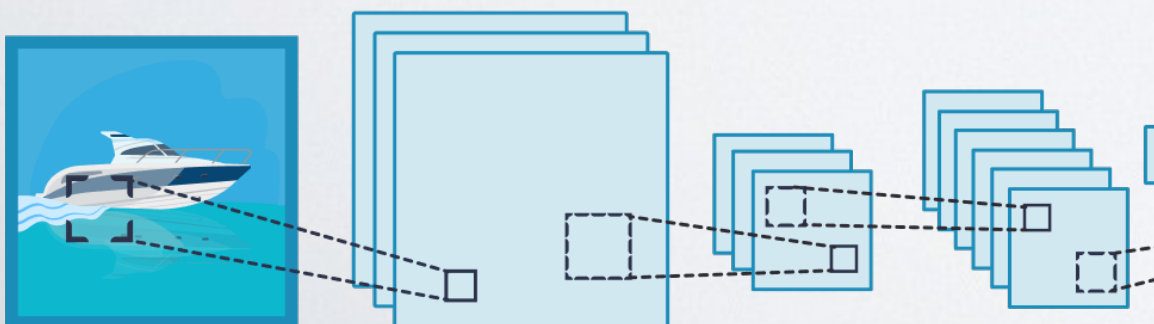
GAT

```
from torch_geometric.nn import GATv2Conv
```

```
class GAT(torch.nn.Module):  
    def __init__(self, dim_in, dim_h, dim_out, heads=8):  
        super().__init__()  
        self.gat1 = GATv2Conv(dim_in, dim_h, heads=heads)  
        self.gat2 = GATv2Conv(dim_h*heads, dim_out,  
heads=1)
```

Remember, heads can be understood as channels in convolutional layers

Each head provides a different “personalized embedding”

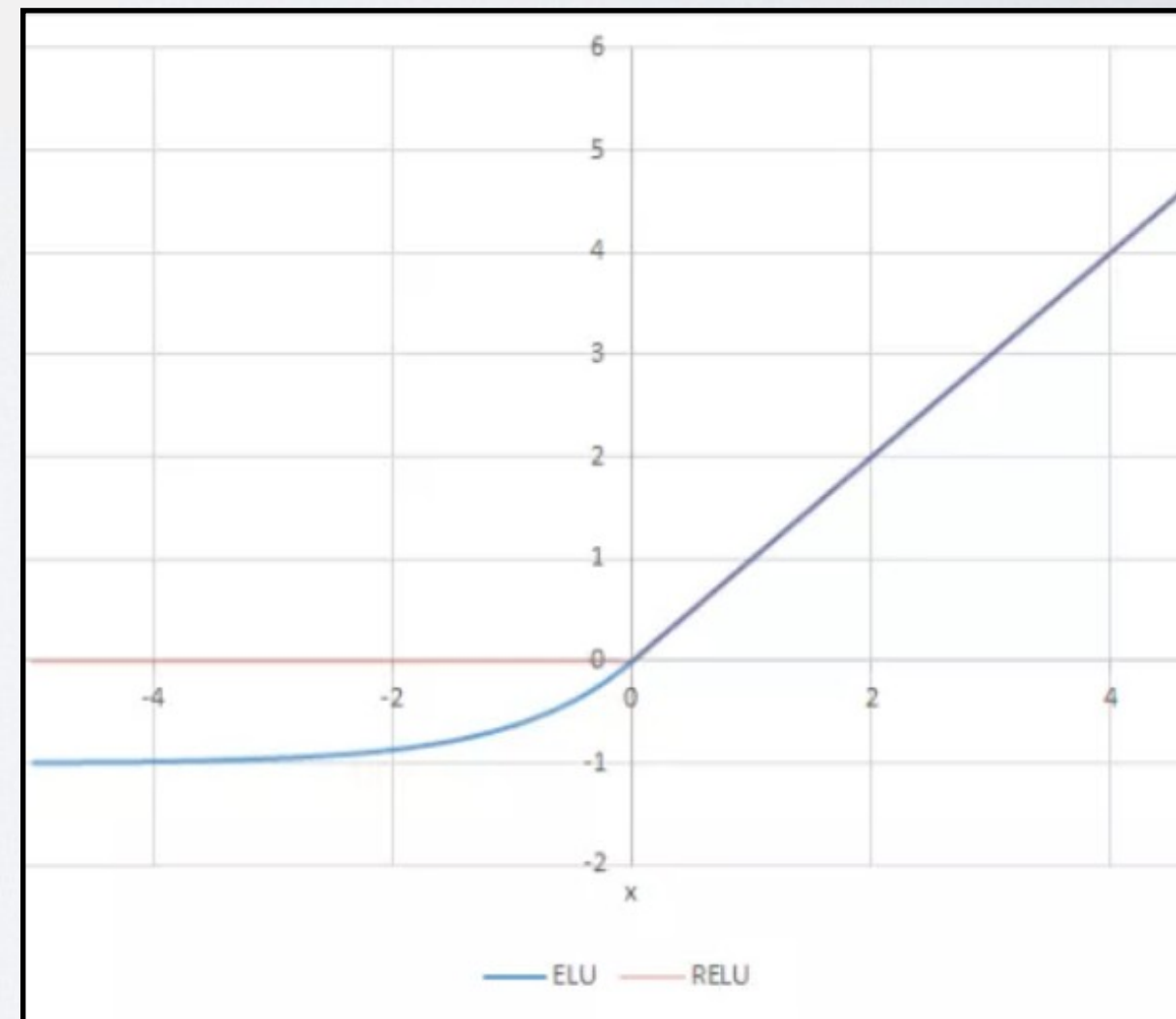


GAT

```
class GAT(torch.nn.Module):  
    def __init__(self, dim_in, dim_h, dim_out, heads=8):  
        super().__init__()  
        self.gat1 = GATv2Conv(dim_in, dim_h, heads=heads)  
        self.gat2 = GATv2Conv(dim_h*heads, dim_out,  
heads=1)
```

```
def forward(self, x, edge_index):  
    h = F.dropout(x, p=0.6, training=self.training)  
    h = self.gat1(h, edge_index)  
    h = F.elu(h)  
    h = F.dropout(h, p=0.6, training=self.training)  
    h = self.gat2(h, edge_index)  
    return F.log_softmax(h, dim=1)
```

Follows original
article



YOUR TURN!