GNN, CNN Models with Cross Validation

In [1]:
```
import os
import random
import time
import numpy as np
import pandas as pd
from sklearn.model_selection import KFold

import torch
from torch import tensor, optim, nn
import torch.nn.functional as F

from torch_geometric.data import Data
from torch_geometric.loader import DataLoader

from torch_geometric.nn import GCNConv, SAGEConv, Linear, global_mean_pool
from torch_geometric_temporal.nn.recurrent import DCRNN

from torch.utils.data import TensorDataset, random_split
from torch.utils.data import DataLoader as CNNLoader
from torch.nn.functional import normalize
```

In [2]:
```
mi_dir_old = r'C:\Users\yl646\Documents\ADHD_research\DATA\OUTPUT\step_6_test\MI_TABLE
mi_dir_adhd   = r'C:\Users\yl646\Documents\ADHD_research\DATA\OUTPUT\step_6_test_compl

mi_dir_adhd_overlap = r'C:\Users\yl646\Documents\ADHD_Research\DATA\OUTPUT\step_6_test
mi_dir_control_overlap = r'C:\Users\yl646\Documents\ADHD_Research\DATA\OUTPUT\step_6_t

mi_dir_control   = r'C:\Users\yl646\Documents\ADHD_research\DATA\OUTPUT\step_6_test_co
result_dir = r'C:\Users\yl646\Documents\ADHD_Research\DATA\OUTPUT\step_6_test_complete
model_dir = r'C:\Users\yl646\Documents\ADHD_Research\DATA\OUTPUT\step_6_test_complete\

epoch_adhd_dir = r'C:\Users\yl646\Documents\ADHD_Research\DATA\OUTPUT\step_6_test_comp
epoch_control_dir = r'C:\Users\yl646\Documents\ADHD_Research\DATA\OUTPUT\step_6_test_c
```

SAGE

In [3]:
```
class SAGE(torch.nn.Module):
    def __init__(self, hidden_channels, k_hop):
        super(SAGE, self).__init__()
        self.k_hop = k_hop
        #torch.manual_seed(12345)
        self.conv1 = SAGEConv(data.num_node_features, hidden_channels)
        self.conv2 = SAGEConv(hidden_channels, hidden_channels)
        self.lin = Linear(hidden_channels, 2)

    def forward(self, x, edge_index, batch):
        if self.k_hop==1:
            x = self.conv1(x, edge_index)
            x = x.relu()

        elif self.k_hop==2:
            x = self.conv1(x, edge_index)
```

```
            x = x.relu()
            x = self.conv2(x, edge_index)
            x = x.relu()

        elif self.k_hop==3:
            x = self.conv1(x, edge_index)
            x = x.relu()
            x = self.conv2(x, edge_index)
            x = x.relu()
            x = self.conv2(x, edge_index)
            x = x.relu()

        elif self.k_hop==4:
            x = self.conv1(x, edge_index)
            x = x.relu()
            x = self.conv2(x, edge_index)
            x = x.relu()
            x = self.conv2(x, edge_index)
            x = x.relu()
            x = self.conv2(x, edge_index)
            x = x.relu()

        # 2. Readout layer
        x = global_mean_pool(x, batch)  # [batch_size, hidden_channels]

        # 3. Apply a final classifier
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.lin(x)
        return x
```

GCN

In [4]:

```
class GCN(torch.nn.Module):
    def __init__(self, hidden_channels, k_hop):
        super(GCN, self).__init__()
        self.k_hop = k_hop
        torch.manual_seed(12345)
        self.conv1 = GCNConv(data.num_node_features, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, hidden_channels)
        self.lin = Linear(hidden_channels, 2)

    def forward(self, x, edge_index, batch):
        if self.k_hop==1:
            x = self.conv1(x, edge_index)
            x = x.relu()

        elif self.k_hop==2:
            x = self.conv1(x, edge_index)
            x = x.relu()
            x = self.conv2(x, edge_index)
            x = x.relu()

        elif self.k_hop==3:
            x = self.conv1(x, edge_index)
            x = x.relu()
            x = self.conv2(x, edge_index)
            x = x.relu()
            x = self.conv2(x, edge_index)
```

```
            x = x.relu()

        elif self.k_hop==4:
            x = self.conv1(x, edge_index)
            x = x.relu()
            x = self.conv2(x, edge_index)
            x = x.relu()
            x = self.conv2(x, edge_index)
            x = x.relu()
            x = self.conv2(x, edge_index)
            x = x.relu()

        # 2. Readout layer
        x = global_mean_pool(x, batch)  # [batch_size, hidden_channels]

        # 3. Apply a final classifier
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.lin(x)
        return x
```

DIFF

In [5]:

```
class DIFF(torch.nn.Module):
    def __init__(self, hidden_channels,K):
        super(DIFF, self).__init__()
        torch.manual_seed(12345)
        self.conv1 = DCRNN(data.num_node_features, hidden_channels, K)
        self.conv2 = DCRNN(hidden_channels, hidden_channels, K)
        self.lin = Linear(hidden_channels, 2)

    def forward(self, x, edge_index, batch):
        x = self.conv1(x, edge_index)
        x = x.relu()

        # 2. Readout layer
        x = global_mean_pool(x, batch)  # [batch_size, hidden_channels]

        # 3. Apply a final classifier

        x = F.dropout(x, p=0.5, training=self.training)
        x = self.lin(x)

        return x
```

CNN

In [6]:

```
def weight_init(m):
    if type(m) == nn.Conv2d:
        nn.init.normal_(m.weight)

class CNN(nn.Module):
    def __init__(self, n_ch=4):
        super(CNN, self).__init__()
        self.n_ch = n_ch

        self.part_one = nn.Sequential(
            nn.Conv2d(in_channels=n_ch, out_channels=16, kernel_size=(3,3), padding=1)
```

```
            nn.MaxPool2d(kernel_size=(3,3)),
            nn.Dropout(),
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=(3,3), padding=1))

        self.part_two_a = nn.Sequential(nn.Conv2d(in_channels=32, out_channels=32, ker

        self.part_three_1 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=(1,1), padding=1),
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=(3,3), padding=1)
        )

        self.maxpool = nn.MaxPool2d(kernel_size=(3,3))
        self.conv1 = nn.Conv2d(in_channels=128, out_channels=8, kernel_size=(3,3), pad
        self.conv2 = nn.Conv2d(in_channels=128, out_channels=8, kernel_size=(3,3), pad
        self.conv3 = nn.Conv2d(in_channels=128, out_channels=8, kernel_size=(3,3), pad
        self.conv4 = nn.Conv2d(in_channels=128, out_channels=8, kernel_size=(3,3), pad

        self.fc = nn.Linear(in_features=32, out_features=2)
        self.softmax = nn.Softmax(dim=1)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()
        self.bn = nn.BatchNorm2d(8)

        self.part_one.apply(weight_init)
        self.part_two_a.apply(weight_init)
        self.part_three_1.apply(weight_init)
        nn.init.normal_(self.conv1.weight)
        nn.init.normal_(self.conv2.weight)
        nn.init.normal_(self.conv3.weight)
        nn.init.normal_(self.conv4.weight)
        nn.init.normal_(self.fc.weight)

    def part_three(self, x):

        x = self.part_three_1(x)
        x1 = self.maxpool(x)
        x1 = self.conv1(x1)

        x1 = self.bn(x1)
        x1 = nn.functional.max_pool2d(input=x1, kernel_size=x1.shape[2:])
        x1 = self.relu(x1)

        x2 = self.maxpool(x)
        x2 = self.conv2(x2)
        x2 = self.bn(x2)
        x2 = nn.functional.avg_pool2d(input=x2, kernel_size=x2.shape[2:])
        x2 = self.tanh(x2)

        x3 = self.maxpool(x)
        x3 = self.conv3(x3)
        x3 = self.bn(x3)
        x3 = nn.functional.max_pool2d(input=x3, kernel_size=x3.shape[2:])
        x3 = self.relu(x3)

        x4 = self.maxpool(x)
        x4 = self.conv4(x4)
        x4 = self.bn(x4)
        x4 = nn.functional.avg_pool2d(input=x4, kernel_size=x4.shape[2:])
        x4 = self.tanh(x4)
```

```
        x = torch.cat((x1,x2,x3,x4),1)
        x = torch.squeeze(x)
        return x

    def forward(self, x):
        x = self.part_one(x)
        x = self.part_two_a(x)
        x = self.part_three(x)
        x = self.fc(x)
        x = self.softmax(x)
        return x
```

Create Dataset from MI tables

In [7]:
```
def getGraph(mi_table, y):
    """
    Input: Adjacency table with shape (epoch, channels, channels). dtype: np.array / y
    output: List that contains pyG graph data objects for each MI table.
    """
    dataset_graph=[]
    (epochs, channels, channels) = mi_table.shape # Get number of epochs and channels

    for epoch in range(epochs):
        edges_np = np.array([[0],[0]]) # Initialize edges matrix
        for row in range(channels):
            for col in range(channels):
                edge = np.array([[row],[col]]) # define fully connected edge matrix of
                edges_np = np.concatenate((edges_np,edge),axis=1)

                # our data is unweighted
                #weight = np.array([[ADHD_mi[epoch,row,col]]])
                #weights_np = np.concatenate((weights_np, weight),axis=0)

        edges_np = edges_np[:,1:]
        edges = tensor(edges_np, dtype=torch.long)

        # data types are required by the loss function
        y = torch.tensor([y], dtype=torch.int64)
        x = torch.tensor(mi_table[epoch,:,:], dtype=torch.float) # entire MI table is

        graph = Data(x=x, edge_index=edges, y=y) # Graph data stucture
        dataset_graph.append(graph)
    return dataset_graph
```

Graph - Training/Validation

This dataset will be split into training set and validation set

In [8]:
```
#  Load mutual information matrices
ADHD_mi = np.load(mi_dir_adhd)
CONTROL_mi = np.load(mi_dir_control)

#  Construct graph from ADHD, CONTROL groups
adhd_train_val_graph = getGraph(ADHD_mi, y=1)
control_train_val_graph = getGraph(CONTROL_mi, y=0)
```

```
#  Combine and shuffle
dataset_graph = adhd_train_val_graph + control_train_val_graph
random.shuffle(dataset_graph)

print("MI table shape: ",ADHD_mi.shape, "(epochs, channels, temp)")
print("MI table shape: ",CONTROL_mi.shape, "(epochs, channels, temp)")
print("# of graphs: ",len(dataset_graph))
```

```
MI table shape:  (2231, 20, 20) (epochs, channels, temp)
MI table shape:  (1757, 20, 20) (epochs, channels, temp)
# of graphs:  3988
```

Graph - Test

Test set is created by taking the average value of all mutual information tables for each patient.

In [9]:
```
#  Get number of epochs for each patient.
epo_per_adhd = np.load(epoch_adhd_dir)
epo_per_control = np.load(epoch_control_dir)

#  For each patient, find mean value of all MI tables
#  Then create new test datapoint.
adhd_test_mi = np.zeros((61,20,20))
n=0
for i, num_epo in enumerate(epo_per_adhd):
    num_epo = int(num_epo)
    adhd_test_mi[i, :, :]= np.mean(ADHD_mi[n:n+num_epo, : , : ])
    n+=num_epo

# same procedure for control group
control_test_mi = np.zeros((60,20,20))
n=0
for i, num_epo in enumerate(epo_per_control):
    num_epo = int(num_epo)
    control_test_mi[i, :, :]= np.mean(CONTROL_mi[n:n+num_epo, : , : ])
    n+=num_epo

# getGraph function to turn this into a pyG graph data format
adhd_test_graph = getGraph(ADHD_mi, y=1)
control_test_graph = getGraph(CONTROL_mi, y=0)

test_graph = adhd_test_graph + control_test_graph
# very important to shuffle. Unshuffled data does not learn.
random.shuffle(test_graph)
```

This is the summary of graph data objects.

In [19]:
```
data = dataset_graph[300]
print(data)
print(f'Number of nodes: {data.num_nodes}')
print(f'Number of edges: {data.num_edges}')
print(f'Has isolated nodes: {data.has_isolated_nodes()}')
print(f'Has self-loops: {data.has_self_loops()}')
print(f'Is undirected: {data.is_undirected()}')
print(f'Number of features: {data.num_node_features}')
```

```
Data(x=[20, 20], edge_index=[2, 400], y=[1])
Number of nodes: 20
Number of edges: 400
Has isolated nodes: False
Has self-loops: True
Is undirected: True
Number of features: 20
```

Image - Training/Validation

In [11]:

```
ADHD_mi_overlap = np.load(mi_dir_adhd_overlap)
CONTROL_mi_overlap = np.load(mi_dir_control_overlap)

n_ch = 4 # Motivated by R, G, B, Alpha channels
(ADHD_epochs, channels, channels) = ADHD_mi_overlap.shape
(CONTROL_epochs, channels, channels) = CONTROL_mi_overlap.shape

#  Number of images
num_img_ADHD = int(ADHD_epochs/n_ch)
num_img_CONTROL = int(CONTROL_epochs/n_ch)
n_img = num_img_ADHD+num_img_CONTROL

#  Target dataset dimension
img_data = np.zeros((n_img, n_ch, channels, channels))
label = np.zeros(n_img)

# select every 4 MI tables and assign it to img_data. This simply raises ADHD_mi_overl
for img in range(num_img_ADHD):
    img_data[img, :, :, :] = ADHD_mi_overlap[n_ch*img:n_ch*(img+1), :, :]
    label[img] = 1
for img in range(num_img_CONTROL):
    img_data[num_img_ADHD+img, :, :, :] = CONTROL_mi_overlap[n_ch*img : n_ch*(img+1),
    label[num_img_ADHD+img] = 0

# just like any other image dataset, all values are normalized to values between 0 and
for img in range(n_img):
    for ch in range(n_ch):
        img_data[img, ch, :, :] = (img_data[img, ch, :, :]) / (np.max(img_data[img, ch

# TensorDataset class does not have in-built shuffle function.
# list of integers upt to 995 is shuffled and used as an index to shuffle label and im
rand_idx = np.arange(996)
random.shuffle(rand_idx)
img_data = img_data[rand_idx, : , : , :]
label = label[rand_idx]

img_data = torch.Tensor(img_data)
label = torch.Tensor(label)
label = label.long() # loss function requires this
dataset_image = TensorDataset(img_data, label) #  Dataset class construction

print('Number of ADHD images',num_img_ADHD)
print('Number of CONTROL images',num_img_CONTROL)
print('Total images',n_img)
print('Train-Validation Image Dataset Shape',img_data.shape)
```

```
Number of ADHD images 557
Number of CONTROL images 439
Total images 996
Train-Validation Image Dataset Shape torch.Size([996, 4, 29, 29])
```

Image Dataset - Test

In [12]:
```
epo_per_adhd = np.load(epoch_adhd_dir)
epo_per_control = np.load(epoch_control_dir)

adhd_test_img = np.zeros((61,4,29,29))
control_test_img = np.zeros((60,4,29,29))
test_label = np.zeros(121)

# since image data is 3 dimensional, there were several ways to average image for pati
n=0
for i, n_epo_patient in enumerate(epo_per_adhd):
    n_im_patient = int(num_epo/n_ch)
    im_patient = np.zeros((n_im_patient, 4, 29, 29))
    for j in range(n_im_patient):
        im_patient[j , : , : , :] = ADHD_mi_overlap[n+j*4 : n+(j+1)*4] # I found the i
    adhd_test_img[i, :, :, :]= np.mean(im_patient, axis=0) # then averaged the images
    test_label[i]=1
    n+=num_epo

n=0
for i, n_epo_patient in enumerate(epo_per_control):
    n_im_patient = int(num_epo/n_ch)
    im_patient = np.zeros((n_im_patient, 4, 29, 29))
    for j in range(n_im_patient):
        im_patient[j , : , : , :] = CONTROL_mi_overlap[n+j*4 : n+(j+1)*4]
    control_test_img[i, :, :, :]= np.mean(im_patient, axis=0)
    n+=num_epo

test_img = np.concatenate((adhd_test_img, control_test_img), axis = 0)

# test set is shuffled in the same manner
rand_idx = np.arange(121)
random.shuffle(rand_idx)
test_img = test_img[rand_idx, : , : , :]
test_label = test_label[rand_idx]

test_img = torch.Tensor(test_img)
test_label = torch.Tensor(test_label)
test_label = test_label.long()
test_dataset_img = TensorDataset(test_img, test_label)
```

Main Model Training and Testing with K-fold

Train / Test Functions

In [13]:
```
def train(model, loader, loss_fn, optimizer):
    model.train()

    for data in loader:  # Iterate batches
        out = model(data.x, data.edge_index, data.batch)  # forward pass
```

```
        loss = loss_fn(out, data.y) # loss
        loss.backward()  # gradient
        optimizer.step()  # update weights
        optimizer.zero_grad()  # clear gradients
    return

def test(model, loader, dataset):
    correct = 0
    for data in loader:
        out = model(data.x, data.edge_index, data.batch)
        pred = out.argmax(dim=1)  # make prediction based on returned softmax values
        correct += int((pred == data.y).sum()) # count correct predictions
    return correct / len(dataset)

# since CNN model does not have data object like GNN, train and test functions were im
def train_cnn(model, loader, loss_fn, optimizer):
    correct=0
    for i, data in enumerate(loader):
        inputs, labels = data
        if inputs.shape[0] ==1:  # k-fold would randomly return a fold size that would
            return
        pred = model(inputs)
        loss = loss_fn(pred, labels)
        loss.backward()

        optimizer.step()
        optimizer.zero_grad()

        pred_argmax = pred.argmax(dim=1)
        correct += int((pred_argmax == labels).sum())

def test_cnn(model, loader, dataset):
    correct = 0
    running_loss = 0.0
    for i, data in enumerate(loader):
        inputs, labels = data
        if inputs.shape[0] ==1:
            break
        pred = model(inputs)
        pred_argmax = pred.argmax(dim=1)
        correct += int((pred_argmax == labels).sum())

    return correct / len(dataset)
```

Main Function

In [14]:

```
def main(model_name, dataset, test_dataset, model_dir, result_dir, k_hop=1, k_fold=10,
    start = time.time()
    # Define model, optimizer, and loss function
    if model_name == 'SAGE':
        model = SAGE(hidden_channels=h_ch, k_hop=k_hop)
    elif model_name == 'GCN':
        model = GCN(hidden_channels=h_ch, k_hop=k_hop)
    elif model_name == 'DIFF':
        model = DIFF(hidden_channels=h_ch, K=k_hop)
    elif model_name =='CNN':
        model = CNN()
    else:
```

```
        print('Error: model not defined')
        return

    #opt = torch.optim.Adam(model.parameters(), lr=0.01)
    opt = optim.NAdam(model.parameters(), lr=0.001, betas = (0.9,0.999), momentum_deca
    loss_fnc = torch.nn.CrossEntropyLoss()

    list_train_acc = []
    list_valid_acc = []
    list_test_acc = []

    # Implement k-fold cross validation
    kf = KFold(n_splits=k_fold, shuffle=True)

    # For each fold
    for fold, (train_index, valid_index) in enumerate(kf.split(dataset)):

        # Split train, test set and define dataloader
        train_dataset = [dataset[i] for i in train_index]
        valid_dataset = [dataset[i] for i in valid_index]
        if model_name =='CNN':
            train_loader = CNNLoader(train_dataset, batch_size=128, shuffle=False)
            valid_loader = CNNLoader(valid_dataset, batch_size=32, shuffle=False)
            test_loader = CNNLoader(test_dataset, batch_size=32, shuffle=False)

        else:
            train_loader = DataLoader(train_dataset, batch_size=128, shuffle=False)
            valid_loader = DataLoader(valid_dataset, batch_size=128, shuffle=False)
            test_loader = DataLoader(test_dataset, batch_size=121, shuffle=False)  # w

        # For each epoch
        for epoch in range(n_epo):
            if model_name == 'CNN':
                train_cnn(model, train_loader, loss_fnc, opt)
            else:
                train(model, train_loader, loss_fnc, opt)

            # Get accuracy for train and validation set
            if model_name =='CNN':
                train_acc = test_cnn(model, train_loader, train_dataset)
                valid_acc = test_cnn(model, valid_loader, valid_dataset)
                test_acc = test_cnn(model, test_loader, test_dataset)
            else:
                train_acc = test(model, train_loader, train_dataset)
                valid_acc = test(model, valid_loader, valid_dataset)
                test_acc = test(model, test_loader, test_dataset)

            list_train_acc.append(train_acc)
            list_valid_acc.append(valid_acc)
            list_test_acc.append(test_acc)
            print(f'Fold: {fold+1}, Epoch: {epoch+1:03d}, Train: {train_acc:.4f}, Vali

    ####################################
    # Save the results for visualization and analysis
    ####################################

    # Turn accuracy to numpy array
    list_train_acc = np.array(list_train_acc)
    list_valid_acc = np.array(list_valid_acc)
    list_test_acc = np.array(list_test_acc)
```

```
    # Reshape results as column vector
    list_train_acc = np.reshape(list_train_acc, (-1,1))
    list_valid_acc = np.reshape(list_valid_acc, (-1,1))
    list_test_acc = np.reshape(list_test_acc, (-1,1))
    results = np.concatenate((list_train_acc,list_valid_acc,list_test_acc), axis=1)
    results = pd.DataFrame(results, columns=['Train', 'Valid', 'Test'])

    # Save accuracy log
    filename = result_dir+r'\kfold_'
    if model_name == 'CNN':
        filename += f'{model_name}_ndam_epo_{n_epo}.csv'
    else:
        filename += f'{model_name}_k_{k_hop}_ndam_epo_{n_epo}.csv'
    results.to_csv(filename, float_format='%.3f', index=False, header=True)

    # Save model for later use
    filename_model = model_dir+r'\kfold_'
    if model_name == 'CNN':
        filename_model += f'{model_name}.pth'
    else:
        filename_model += f'{model_name}_k_{k_hop}.pth'
    torch.save(model, filename_model)

    # Retain saved model
    # This may not work for other environments due to different path names
    model1 = torch.load(filename_model)
    #test_acc = test(model1, test_loader, test_dataset)
    #print(f'Acc: {test_acc:.4f}')
    print('\nElapsed Time: ',time.time()-start)
```

CNN Training

In [19]:
```
main('CNN', dataset_image, test_dataset_img, model_dir, result_dir, n_epo=50)
```

```
Fold: 10, Epoch: 031, Train: 0.9877, Valid: 1.0000, Test: 0.9421
Fold: 10, Epoch: 032, Train: 0.9889, Valid: 0.9495, Test: 0.9339
Fold: 10, Epoch: 033, Train: 0.9922, Valid: 0.9798, Test: 0.9421
Fold: 10, Epoch: 034, Train: 0.9933, Valid: 0.9798, Test: 0.9339
Fold: 10, Epoch: 035, Train: 0.9877, Valid: 0.9697, Test: 0.9256
Fold: 10, Epoch: 036, Train: 0.9877, Valid: 1.0000, Test: 0.9339
Fold: 10, Epoch: 037, Train: 0.9911, Valid: 1.0000, Test: 0.9421
Fold: 10, Epoch: 038, Train: 0.9944, Valid: 0.9899, Test: 0.9421
Fold: 10, Epoch: 039, Train: 0.9900, Valid: 0.9899, Test: 0.9339
Fold: 10, Epoch: 040, Train: 0.9933, Valid: 0.9899, Test: 0.9587
Fold: 10, Epoch: 041, Train: 0.9933, Valid: 0.9596, Test: 0.9504
Fold: 10, Epoch: 042, Train: 0.9922, Valid: 0.9899, Test: 0.9256
Fold: 10, Epoch: 043, Train: 0.9922, Valid: 0.9899, Test: 0.9504
Fold: 10, Epoch: 044, Train: 0.9889, Valid: 1.0000, Test: 0.9504
Fold: 10, Epoch: 045, Train: 0.9900, Valid: 0.9899, Test: 0.9256
Fold: 10, Epoch: 046, Train: 0.9810, Valid: 0.9899, Test: 0.9174
Fold: 10, Epoch: 047, Train: 0.9900, Valid: 1.0000, Test: 0.9339
Fold: 10, Epoch: 048, Train: 0.9933, Valid: 0.9697, Test: 0.9256
Fold: 10, Epoch: 049, Train: 0.9922, Valid: 0.9899, Test: 0.9339
Fold: 10, Epoch: 050, Train: 0.9933, Valid: 1.0000, Test: 0.9421

Elapsed Time:  647.8403980731964
```

SAGE Training with k = 1, 2, 3, 4

In [15]:
```
main('SAGE', dataset_graph, test_graph, model_dir, result_dir, k_hop=1, k_fold=10, n_e
```

```
Fold: 10, Epoch: 031, Train: 0.8245, Valid: 0.8392, Test: 0.8272
Fold: 10, Epoch: 032, Train: 0.8184, Valid: 0.8317, Test: 0.8167
Fold: 10, Epoch: 033, Train: 0.8295, Valid: 0.8191, Test: 0.8327
Fold: 10, Epoch: 034, Train: 0.8184, Valid: 0.8317, Test: 0.8257
Fold: 10, Epoch: 035, Train: 0.8234, Valid: 0.8342, Test: 0.8175
Fold: 10, Epoch: 036, Train: 0.8212, Valid: 0.8543, Test: 0.8172
Fold: 10, Epoch: 037, Train: 0.8248, Valid: 0.8492, Test: 0.8212
Fold: 10, Epoch: 038, Train: 0.8304, Valid: 0.8191, Test: 0.8272
Fold: 10, Epoch: 039, Train: 0.8156, Valid: 0.8367, Test: 0.8235
Fold: 10, Epoch: 040, Train: 0.8262, Valid: 0.8543, Test: 0.8260
Fold: 10, Epoch: 041, Train: 0.8279, Valid: 0.8467, Test: 0.8295
Fold: 10, Epoch: 042, Train: 0.8220, Valid: 0.8392, Test: 0.8217
Fold: 10, Epoch: 043, Train: 0.8326, Valid: 0.8392, Test: 0.8235
Fold: 10, Epoch: 044, Train: 0.8240, Valid: 0.8317, Test: 0.8280
Fold: 10, Epoch: 045, Train: 0.8281, Valid: 0.8367, Test: 0.8167
Fold: 10, Epoch: 046, Train: 0.8228, Valid: 0.8317, Test: 0.8305
Fold: 10, Epoch: 047, Train: 0.8223, Valid: 0.8266, Test: 0.8222
Fold: 10, Epoch: 048, Train: 0.8312, Valid: 0.8317, Test: 0.8335
Fold: 10, Epoch: 049, Train: 0.8306, Valid: 0.8065, Test: 0.8280
Fold: 10, Epoch: 050, Train: 0.8226, Valid: 0.8291, Test: 0.8205

Elapsed Time:  537.4762423038483
```

In [17]:
```
main('SAGE', dataset_graph, test_graph, model_dir, result_dir, k_hop=2, k_fold=10, n_e
```

```
Fold: 10, Epoch: 031, Train: 0.9554, Valid: 0.9296, Test: 0.9506
Fold: 10, Epoch: 032, Train: 0.9510, Valid: 0.9347, Test: 0.9544
Fold: 10, Epoch: 033, Train: 0.9529, Valid: 0.9322, Test: 0.9466
Fold: 10, Epoch: 034, Train: 0.9577, Valid: 0.9372, Test: 0.9493
Fold: 10, Epoch: 035, Train: 0.9563, Valid: 0.9347, Test: 0.9524
Fold: 10, Epoch: 036, Train: 0.9521, Valid: 0.9322, Test: 0.9476
Fold: 10, Epoch: 037, Train: 0.9560, Valid: 0.9146, Test: 0.9536
Fold: 10, Epoch: 038, Train: 0.9454, Valid: 0.9296, Test: 0.9506
Fold: 10, Epoch: 039, Train: 0.9560, Valid: 0.9271, Test: 0.9509
Fold: 10, Epoch: 040, Train: 0.9524, Valid: 0.9271, Test: 0.9514
Fold: 10, Epoch: 041, Train: 0.9515, Valid: 0.9196, Test: 0.9471
Fold: 10, Epoch: 042, Train: 0.9526, Valid: 0.9372, Test: 0.9551
Fold: 10, Epoch: 043, Train: 0.9540, Valid: 0.9246, Test: 0.9509
Fold: 10, Epoch: 044, Train: 0.9529, Valid: 0.9196, Test: 0.9531
Fold: 10, Epoch: 045, Train: 0.9552, Valid: 0.9221, Test: 0.9524
Fold: 10, Epoch: 046, Train: 0.9510, Valid: 0.9397, Test: 0.9534
Fold: 10, Epoch: 047, Train: 0.9493, Valid: 0.9146, Test: 0.9476
Fold: 10, Epoch: 048, Train: 0.9526, Valid: 0.9146, Test: 0.9526
Fold: 10, Epoch: 049, Train: 0.9524, Valid: 0.9322, Test: 0.9488
Fold: 10, Epoch: 050, Train: 0.9552, Valid: 0.9246, Test: 0.9546

Elapsed Time:  941.2452042102814
```

In [18]:
```
main('SAGE', dataset_graph, test_graph, model_dir, result_dir, k_hop=3, k_fold=10, n_e
```

```
Fold: 10, Epoch: 031, Train: 0.9780, Valid: 0.9698, Test: 0.9772
Fold: 10, Epoch: 032, Train: 0.9799, Valid: 0.9698, Test: 0.9804
Fold: 10, Epoch: 033, Train: 0.9827, Valid: 0.9749, Test: 0.9782
Fold: 10, Epoch: 034, Train: 0.9844, Valid: 0.9774, Test: 0.9845
Fold: 10, Epoch: 035, Train: 0.9791, Valid: 0.9724, Test: 0.9794
Fold: 10, Epoch: 036, Train: 0.9850, Valid: 0.9698, Test: 0.9809
Fold: 10, Epoch: 037, Train: 0.9774, Valid: 0.9623, Test: 0.9799
Fold: 10, Epoch: 038, Train: 0.9825, Valid: 0.9648, Test: 0.9799
Fold: 10, Epoch: 039, Train: 0.9833, Valid: 0.9623, Test: 0.9824
Fold: 10, Epoch: 040, Train: 0.9794, Valid: 0.9698, Test: 0.9807
Fold: 10, Epoch: 041, Train: 0.9816, Valid: 0.9573, Test: 0.9794
Fold: 10, Epoch: 042, Train: 0.9802, Valid: 0.9573, Test: 0.9797
Fold: 10, Epoch: 043, Train: 0.9844, Valid: 0.9698, Test: 0.9832
Fold: 10, Epoch: 044, Train: 0.9758, Valid: 0.9447, Test: 0.9712
Fold: 10, Epoch: 045, Train: 0.9819, Valid: 0.9673, Test: 0.9767
Fold: 10, Epoch: 046, Train: 0.9855, Valid: 0.9598, Test: 0.9832
Fold: 10, Epoch: 047, Train: 0.9825, Valid: 0.9698, Test: 0.9779
Fold: 10, Epoch: 048, Train: 0.9833, Valid: 0.9673, Test: 0.9797
Fold: 10, Epoch: 049, Train: 0.9847, Valid: 0.9648, Test: 0.9802
Fold: 10, Epoch: 050, Train: 0.9822, Valid: 0.9648, Test: 0.9797

Elapsed Time:  1316.321543931961
```

In [19]:
```
main('SAGE', dataset_graph, test_graph, model_dir, result_dir, k_hop=4, k_fold=10, n_e
```

```
Fold: 10, Epoch: 031, Train: 0.9864, Valid: 0.9598, Test: 0.9819
Fold: 10, Epoch: 032, Train: 0.9777, Valid: 0.9548, Test: 0.9744
Fold: 10, Epoch: 033, Train: 0.9747, Valid: 0.9472, Test: 0.9724
Fold: 10, Epoch: 034, Train: 0.9903, Valid: 0.9724, Test: 0.9865
Fold: 10, Epoch: 035, Train: 0.9872, Valid: 0.9724, Test: 0.9872
Fold: 10, Epoch: 036, Train: 0.9646, Valid: 0.9397, Test: 0.9641
Fold: 10, Epoch: 037, Train: 0.9875, Valid: 0.9698, Test: 0.9870
Fold: 10, Epoch: 038, Train: 0.9702, Valid: 0.9447, Test: 0.9672
Fold: 10, Epoch: 039, Train: 0.9811, Valid: 0.9724, Test: 0.9802
Fold: 10, Epoch: 040, Train: 0.9855, Valid: 0.9698, Test: 0.9829
Fold: 10, Epoch: 041, Train: 0.9869, Valid: 0.9598, Test: 0.9845
Fold: 10, Epoch: 042, Train: 0.9889, Valid: 0.9724, Test: 0.9867
Fold: 10, Epoch: 043, Train: 0.9911, Valid: 0.9749, Test: 0.9880
Fold: 10, Epoch: 044, Train: 0.9847, Valid: 0.9698, Test: 0.9822
Fold: 10, Epoch: 045, Train: 0.9875, Valid: 0.9648, Test: 0.9840
Fold: 10, Epoch: 046, Train: 0.9880, Valid: 0.9648, Test: 0.9855
Fold: 10, Epoch: 047, Train: 0.9914, Valid: 0.9724, Test: 0.9897
Fold: 10, Epoch: 048, Train: 0.9894, Valid: 0.9698, Test: 0.9895
Fold: 10, Epoch: 049, Train: 0.9861, Valid: 0.9598, Test: 0.9829
Fold: 10, Epoch: 050, Train: 0.9886, Valid: 0.9698, Test: 0.9865

Elapsed Time:  1705.7111928462982
```

GCN Training with k = 1, 2, 3, 4

In [20]:
```
main('GCN', dataset_graph, test_graph, model_dir, result_dir, k_hop=1, k_fold=10, n_ep
```

```
Fold: 10, Epoch: 031, Train: 0.7554, Valid: 0.7789, Test: 0.7558
Fold: 10, Epoch: 032, Train: 0.7632, Valid: 0.7462, Test: 0.7696
Fold: 10, Epoch: 033, Train: 0.7630, Valid: 0.7563, Test: 0.7643
Fold: 10, Epoch: 034, Train: 0.7616, Valid: 0.7688, Test: 0.7548
Fold: 10, Epoch: 035, Train: 0.7677, Valid: 0.7688, Test: 0.7655
Fold: 10, Epoch: 036, Train: 0.7699, Valid: 0.7412, Test: 0.7696
Fold: 10, Epoch: 037, Train: 0.7680, Valid: 0.7764, Test: 0.7560
Fold: 10, Epoch: 038, Train: 0.7727, Valid: 0.7563, Test: 0.7711
Fold: 10, Epoch: 039, Train: 0.7604, Valid: 0.7663, Test: 0.7648
Fold: 10, Epoch: 040, Train: 0.7671, Valid: 0.7538, Test: 0.7630
Fold: 10, Epoch: 041, Train: 0.7624, Valid: 0.7513, Test: 0.7673
Fold: 10, Epoch: 042, Train: 0.7735, Valid: 0.7337, Test: 0.7623
Fold: 10, Epoch: 043, Train: 0.7618, Valid: 0.7362, Test: 0.7701
Fold: 10, Epoch: 044, Train: 0.7786, Valid: 0.7412, Test: 0.7633
Fold: 10, Epoch: 045, Train: 0.7632, Valid: 0.7764, Test: 0.7643
Fold: 10, Epoch: 046, Train: 0.7655, Valid: 0.7638, Test: 0.7731
Fold: 10, Epoch: 047, Train: 0.7682, Valid: 0.7286, Test: 0.7605
Fold: 10, Epoch: 048, Train: 0.7613, Valid: 0.7638, Test: 0.7635
Fold: 10, Epoch: 049, Train: 0.7593, Valid: 0.7462, Test: 0.7643
Fold: 10, Epoch: 050, Train: 0.7691, Valid: 0.7613, Test: 0.7553

Elapsed Time:  872.9826145172119
```

In [21]:
```
main('GCN', dataset_graph, test_graph, model_dir, result_dir, k_hop=2, k_fold=10, n_ep
```

```
Fold: 10, Epoch: 031, Train: 0.8259, Valid: 0.8719, Test: 0.8235
Fold: 10, Epoch: 032, Train: 0.8214, Valid: 0.8417, Test: 0.8187
Fold: 10, Epoch: 033, Train: 0.8273, Valid: 0.8543, Test: 0.8267
Fold: 10, Epoch: 034, Train: 0.8245, Valid: 0.8593, Test: 0.8302
Fold: 10, Epoch: 035, Train: 0.8237, Valid: 0.8492, Test: 0.8260
Fold: 10, Epoch: 036, Train: 0.8309, Valid: 0.8693, Test: 0.8300
Fold: 10, Epoch: 037, Train: 0.8203, Valid: 0.8618, Test: 0.8185
Fold: 10, Epoch: 038, Train: 0.8309, Valid: 0.8442, Test: 0.8265
Fold: 10, Epoch: 039, Train: 0.8226, Valid: 0.8492, Test: 0.8255
Fold: 10, Epoch: 040, Train: 0.8228, Valid: 0.8568, Test: 0.8230
Fold: 10, Epoch: 041, Train: 0.8251, Valid: 0.8568, Test: 0.8190
Fold: 10, Epoch: 042, Train: 0.8209, Valid: 0.8367, Test: 0.8277
Fold: 10, Epoch: 043, Train: 0.8201, Valid: 0.8442, Test: 0.8235
Fold: 10, Epoch: 044, Train: 0.8284, Valid: 0.8241, Test: 0.8245
Fold: 10, Epoch: 045, Train: 0.8251, Valid: 0.8518, Test: 0.8312
Fold: 10, Epoch: 046, Train: 0.8281, Valid: 0.8618, Test: 0.8270
Fold: 10, Epoch: 047, Train: 0.8267, Valid: 0.8417, Test: 0.8310
Fold: 10, Epoch: 048, Train: 0.8159, Valid: 0.8518, Test: 0.8290
Fold: 10, Epoch: 049, Train: 0.8231, Valid: 0.8266, Test: 0.8257
Fold: 10, Epoch: 050, Train: 0.8281, Valid: 0.8442, Test: 0.8192

Elapsed Time:  1438.8208475112915
```

In [22]:
```
main('GCN', dataset_graph, test_graph, model_dir, result_dir, k_hop=3, k_fold=10, n_ep
```

```
Fold: 10, Epoch: 031, Train: 0.9008, Valid: 0.8970, Test: 0.9017
Fold: 10, Epoch: 032, Train: 0.8994, Valid: 0.8844, Test: 0.8937
Fold: 10, Epoch: 033, Train: 0.8983, Valid: 0.8769, Test: 0.8927
Fold: 10, Epoch: 034, Train: 0.9008, Valid: 0.8869, Test: 0.8992
Fold: 10, Epoch: 035, Train: 0.9042, Valid: 0.8794, Test: 0.9042
Fold: 10, Epoch: 036, Train: 0.8961, Valid: 0.8869, Test: 0.8949
Fold: 10, Epoch: 037, Train: 0.8972, Valid: 0.8794, Test: 0.8939
Fold: 10, Epoch: 038, Train: 0.8928, Valid: 0.8693, Test: 0.8984
Fold: 10, Epoch: 039, Train: 0.8992, Valid: 0.8819, Test: 0.8972
Fold: 10, Epoch: 040, Train: 0.8953, Valid: 0.8794, Test: 0.8969
Fold: 10, Epoch: 041, Train: 0.8986, Valid: 0.8869, Test: 0.8982
Fold: 10, Epoch: 042, Train: 0.8992, Valid: 0.8844, Test: 0.8982
Fold: 10, Epoch: 043, Train: 0.9025, Valid: 0.8844, Test: 0.8937
Fold: 10, Epoch: 044, Train: 0.8958, Valid: 0.8744, Test: 0.8942
Fold: 10, Epoch: 045, Train: 0.9033, Valid: 0.8794, Test: 0.8952
Fold: 10, Epoch: 046, Train: 0.9008, Valid: 0.8769, Test: 0.9022
Fold: 10, Epoch: 047, Train: 0.8992, Valid: 0.8693, Test: 0.8967
Fold: 10, Epoch: 048, Train: 0.8944, Valid: 0.8744, Test: 0.8969
Fold: 10, Epoch: 049, Train: 0.8997, Valid: 0.8769, Test: 0.9002
Fold: 10, Epoch: 050, Train: 0.9028, Valid: 0.8844, Test: 0.8992

Elapsed Time:  2026.9459819793701
```

In [23]:
```
main('GCN', dataset_graph, test_graph, model_dir, result_dir, k_hop=4, k_fold=10, n_ep
```

```
Fold: 10, Epoch: 031, Train: 0.9036, Valid: 0.9121, Test: 0.9045
Fold: 10, Epoch: 032, Train: 0.9084, Valid: 0.9196, Test: 0.9100
Fold: 10, Epoch: 033, Train: 0.9131, Valid: 0.8945, Test: 0.9160
Fold: 10, Epoch: 034, Train: 0.9053, Valid: 0.9070, Test: 0.9030
Fold: 10, Epoch: 035, Train: 0.9031, Valid: 0.9095, Test: 0.9060
Fold: 10, Epoch: 036, Train: 0.9109, Valid: 0.9121, Test: 0.9122
Fold: 10, Epoch: 037, Train: 0.9067, Valid: 0.9070, Test: 0.9010
Fold: 10, Epoch: 038, Train: 0.9089, Valid: 0.9045, Test: 0.9082
Fold: 10, Epoch: 039, Train: 0.9056, Valid: 0.9095, Test: 0.9050
Fold: 10, Epoch: 040, Train: 0.9070, Valid: 0.9095, Test: 0.9040
Fold: 10, Epoch: 041, Train: 0.9097, Valid: 0.9095, Test: 0.9092
Fold: 10, Epoch: 042, Train: 0.9047, Valid: 0.9020, Test: 0.9025
Fold: 10, Epoch: 043, Train: 0.8983, Valid: 0.8995, Test: 0.9020
Fold: 10, Epoch: 044, Train: 0.9084, Valid: 0.8945, Test: 0.9060
Fold: 10, Epoch: 045, Train: 0.9036, Valid: 0.8995, Test: 0.9080
Fold: 10, Epoch: 046, Train: 0.9072, Valid: 0.9070, Test: 0.9057
Fold: 10, Epoch: 047, Train: 0.8992, Valid: 0.8920, Test: 0.8967
Fold: 10, Epoch: 048, Train: 0.9084, Valid: 0.9095, Test: 0.9060
Fold: 10, Epoch: 049, Train: 0.9050, Valid: 0.8995, Test: 0.9002
Fold: 10, Epoch: 050, Train: 0.9084, Valid: 0.8995, Test: 0.9025

Elapsed Time:  2626.9574761390686
```

DIFF Training with k = 1, 2, 3, 4

In [15]:
```
main('DIFF', dataset_graph, test_graph, model_dir, result_dir, k_hop=1, k_fold=10, n_e
```

```
Fold: 10, Epoch: 031, Train: 0.8136, Valid: 0.8241, Test: 0.8159
Fold: 10, Epoch: 032, Train: 0.8167, Valid: 0.7965, Test: 0.8124
Fold: 10, Epoch: 033, Train: 0.8189, Valid: 0.7839, Test: 0.8185
Fold: 10, Epoch: 034, Train: 0.8153, Valid: 0.7915, Test: 0.8142
Fold: 10, Epoch: 035, Train: 0.8142, Valid: 0.7990, Test: 0.8159
Fold: 10, Epoch: 036, Train: 0.8156, Valid: 0.8266, Test: 0.8107
Fold: 10, Epoch: 037, Train: 0.8170, Valid: 0.7940, Test: 0.8159
Fold: 10, Epoch: 038, Train: 0.8228, Valid: 0.7965, Test: 0.8205
Fold: 10, Epoch: 039, Train: 0.8078, Valid: 0.7990, Test: 0.8119
Fold: 10, Epoch: 040, Train: 0.8134, Valid: 0.7940, Test: 0.8152
Fold: 10, Epoch: 041, Train: 0.8072, Valid: 0.7889, Test: 0.8092
Fold: 10, Epoch: 042, Train: 0.8106, Valid: 0.8116, Test: 0.8002
Fold: 10, Epoch: 043, Train: 0.8156, Valid: 0.7990, Test: 0.8162
Fold: 10, Epoch: 044, Train: 0.8117, Valid: 0.8065, Test: 0.8119
Fold: 10, Epoch: 045, Train: 0.8095, Valid: 0.8116, Test: 0.8132
Fold: 10, Epoch: 046, Train: 0.8175, Valid: 0.7990, Test: 0.8172
Fold: 10, Epoch: 047, Train: 0.8162, Valid: 0.8216, Test: 0.8119
Fold: 10, Epoch: 048, Train: 0.8175, Valid: 0.8015, Test: 0.8190
Fold: 10, Epoch: 049, Train: 0.8301, Valid: 0.7965, Test: 0.8114
Fold: 10, Epoch: 050, Train: 0.8109, Valid: 0.8015, Test: 0.8159

Elapsed Time:  7795.776322126389
```

In [16]:
```
main('DIFF', dataset_graph, test_graph, model_dir, result_dir, k_hop=2, k_fold=10, n_e
```

```
Fold: 10, Epoch: 031, Train: 0.8251, Valid: 0.8342, Test: 0.8197
Fold: 10, Epoch: 032, Train: 0.8231, Valid: 0.8065, Test: 0.8177
Fold: 10, Epoch: 033, Train: 0.8256, Valid: 0.8543, Test: 0.8260
Fold: 10, Epoch: 034, Train: 0.8217, Valid: 0.8442, Test: 0.8245
Fold: 10, Epoch: 035, Train: 0.8148, Valid: 0.8492, Test: 0.8242
Fold: 10, Epoch: 036, Train: 0.8253, Valid: 0.7990, Test: 0.8222
Fold: 10, Epoch: 037, Train: 0.8256, Valid: 0.8266, Test: 0.8240
Fold: 10, Epoch: 038, Train: 0.8173, Valid: 0.8191, Test: 0.8257
Fold: 10, Epoch: 039, Train: 0.8198, Valid: 0.8417, Test: 0.8272
Fold: 10, Epoch: 040, Train: 0.8262, Valid: 0.8191, Test: 0.8260
Fold: 10, Epoch: 041, Train: 0.8209, Valid: 0.8417, Test: 0.8187
Fold: 10, Epoch: 042, Train: 0.8223, Valid: 0.8241, Test: 0.8222
Fold: 10, Epoch: 043, Train: 0.8181, Valid: 0.8467, Test: 0.8190
Fold: 10, Epoch: 044, Train: 0.8248, Valid: 0.8342, Test: 0.8175
Fold: 10, Epoch: 045, Train: 0.8240, Valid: 0.8116, Test: 0.8245
Fold: 10, Epoch: 046, Train: 0.8251, Valid: 0.8040, Test: 0.8295
Fold: 10, Epoch: 047, Train: 0.8231, Valid: 0.8367, Test: 0.8290
Fold: 10, Epoch: 048, Train: 0.8259, Valid: 0.8317, Test: 0.8292
Fold: 10, Epoch: 049, Train: 0.8292, Valid: 0.8065, Test: 0.8217
Fold: 10, Epoch: 050, Train: 0.8198, Valid: 0.8467, Test: 0.8285

Elapsed Time:  10949.794477462769
```

In [17]:
```
main('DIFF', dataset_graph, test_graph, model_dir, result_dir, k_hop=3, k_fold=10, n_e
```

```
Fold: 10, Epoch: 031, Train: 0.8630, Valid: 0.8543, Test: 0.8709
Fold: 10, Epoch: 032, Train: 0.8735, Valid: 0.8693, Test: 0.8731
Fold: 10, Epoch: 033, Train: 0.8724, Valid: 0.8543, Test: 0.8781
Fold: 10, Epoch: 034, Train: 0.8766, Valid: 0.8618, Test: 0.8656
Fold: 10, Epoch: 035, Train: 0.8738, Valid: 0.8794, Test: 0.8726
Fold: 10, Epoch: 036, Train: 0.8710, Valid: 0.8869, Test: 0.8721
Fold: 10, Epoch: 037, Train: 0.8855, Valid: 0.8543, Test: 0.8749
Fold: 10, Epoch: 038, Train: 0.8708, Valid: 0.8467, Test: 0.8739
Fold: 10, Epoch: 039, Train: 0.8774, Valid: 0.8769, Test: 0.8696
Fold: 10, Epoch: 040, Train: 0.8802, Valid: 0.8593, Test: 0.8791
Fold: 10, Epoch: 041, Train: 0.8710, Valid: 0.8568, Test: 0.8799
Fold: 10, Epoch: 042, Train: 0.8758, Valid: 0.8467, Test: 0.8646
Fold: 10, Epoch: 043, Train: 0.8758, Valid: 0.8668, Test: 0.8786
Fold: 10, Epoch: 044, Train: 0.8749, Valid: 0.8668, Test: 0.8796
Fold: 10, Epoch: 045, Train: 0.8783, Valid: 0.8693, Test: 0.8676
Fold: 10, Epoch: 046, Train: 0.8777, Valid: 0.8342, Test: 0.8804
Fold: 10, Epoch: 047, Train: 0.8769, Valid: 0.8668, Test: 0.8704
Fold: 10, Epoch: 048, Train: 0.8741, Valid: 0.8618, Test: 0.8736
Fold: 10, Epoch: 049, Train: 0.8766, Valid: 0.8668, Test: 0.8769
Fold: 10, Epoch: 050, Train: 0.8749, Valid: 0.8693, Test: 0.8729

Elapsed Time:  13594.784582138062
```

In [18]:
```
main('DIFF', dataset_graph, test_graph, model_dir, result_dir, k_hop=4, k_fold=10, n_e
```

```
Fold: 10, Epoch: 031, Train: 0.8772, Valid: 0.8869, Test: 0.8914
Fold: 10, Epoch: 032, Train: 0.8811, Valid: 0.8794, Test: 0.8889
Fold: 10, Epoch: 033, Train: 0.8827, Valid: 0.8995, Test: 0.8819
Fold: 10, Epoch: 034, Train: 0.8889, Valid: 0.9146, Test: 0.8854
Fold: 10, Epoch: 035, Train: 0.8900, Valid: 0.9045, Test: 0.8824
Fold: 10, Epoch: 036, Train: 0.8880, Valid: 0.8794, Test: 0.8887
Fold: 10, Epoch: 037, Train: 0.8864, Valid: 0.8970, Test: 0.8899
Fold: 10, Epoch: 038, Train: 0.8788, Valid: 0.8819, Test: 0.8899
Fold: 10, Epoch: 039, Train: 0.8833, Valid: 0.8894, Test: 0.8919
Fold: 10, Epoch: 040, Train: 0.8900, Valid: 0.8894, Test: 0.8806
Fold: 10, Epoch: 041, Train: 0.8850, Valid: 0.8844, Test: 0.8869
Fold: 10, Epoch: 042, Train: 0.8841, Valid: 0.8794, Test: 0.8892
Fold: 10, Epoch: 043, Train: 0.8844, Valid: 0.8894, Test: 0.8919
Fold: 10, Epoch: 044, Train: 0.8914, Valid: 0.8894, Test: 0.8882
Fold: 10, Epoch: 045, Train: 0.8903, Valid: 0.8794, Test: 0.8857
Fold: 10, Epoch: 046, Train: 0.8897, Valid: 0.8769, Test: 0.8824
Fold: 10, Epoch: 047, Train: 0.8791, Valid: 0.9045, Test: 0.8859
Fold: 10, Epoch: 048, Train: 0.8933, Valid: 0.8995, Test: 0.8917
Fold: 10, Epoch: 049, Train: 0.8864, Valid: 0.9020, Test: 0.8877
Fold: 10, Epoch: 050, Train: 0.8975, Valid: 0.8945, Test: 0.8884

Elapsed Time:  16847.374128580093
```