EEG classification of ADHD patients with Graph Neural Network

Artificial Intelligence (CSE 4633) Course Project Final Report

Yongjun Lee
Department of Mathematics and Statistics
Mississippi State University
Starkville, United States
yl646@msstate.edu

***Abstract*—Attention Deficit Hyperactivity Disorder (ADHD) is one of the most common mental disorders in the United States. In recent years, a growing number of researchers have been applying Deep Learning (DL) algorithms, specifically Convolutional Neural Networks (CNN), to classify ADHD patients from brain imaging data. While this approach yields high accuracy, it fails to provide insight into the brain connectivity of ADHD patients for clinical purposes. Previous work attempted to solve this issue by constructing a correlation network from EEG channels and training a CNN model with this data. In this research, we will construct a similar network using mutual information between each channel. However, a Graph Neural Network (GNN) model will be used instead to better accommodate the nature of constructed network data structure. We expect our model to outperform the previous CNN model while allowing us to understand the brain network connectivity of ADHD patients. Minimal preprocessing steps will be taken for network construction and training due to the nature of DL models. Further research can investigate how the model performance varies with different adjacency matrix construction or convolution methods.**

***Index Terms*—EEG, ADHD, Mutual Information, Graph Neural Network, Classification**

I. Introduction

Attention Deficit/Hyperactivity Disorder (ADHD) is one of the most prominent mental illness in the United States. An estimated 6.1 million (9.4%) of children 2–17 years of age, is said to have ever received an ADHD diagnosis [1]. Electroencephalogram (EEG) has been used for diagnosis and causality research for ADHD. EEG is a electrical brain activity recording technique that is more than a century old. Due to advancement in deep learning (DL) techniques, many models were proposed for feature extraction, and classification. Especially, convolutional neural networks (CNN) have been one of the most popular model for EEG research purposes. CNN has proven its performance in classification tasks of image-like data. However, use of CNN in EEG has its limitations. CNN does not provide useful interpretable results. Machine learning applications in EEG research is still facing a “black box” problem. Because of this reason, EEG is not considered as a single reliable source for ADHD patient diagnosis in the neuroscience community [3].

On the other hand, graph neural networks (GNN) has been gaining popularity since 2019. It is a deep learning model that can train on graph dataset and perform node classification, graph classification, and graph clustering tasks. It uses aggregation function to find node embeddings and/or graph embeddings. Function parameters are learned from feature vectors of the nodes in a graph. Instead color mapping EEG singals to image-like data to train CNN, each EEG electrode can be viewed as a node and their relationship can be embedded in a graph structure. We believe that this is a better representation of EEG signals, therefore outperforming CNN. We also expect GNN to provide insights to the connectivity of ADHD brain with the use of graph features. Also, by varying model structures, we expect to learn the intricacies of ADHD brain connectivity.

II. Related Work

A. Deep Learning in EEG Research

Roy et al. “review[ed] 154 papers that apply DL to EEG, published between January 2010 and July 2018, and spanning different application domains such as epilepsy, sleep, brain-computer interfacing, and cognitive and affective monitoring”. According to the review, DL models using raw EEG data instead of hand-crafted features has achieved promising results. However, preprocessing and artifact removal is still considered as a crucial step. The median of subjects was 13, and median total recording time in minutes was 360, which tells us our number of recordings (n=121) and recording time (260 minutes) is sufficiently large. Out of all the reviewed papers, 40.3% used convolutional neural networks which is three times more compared to the second most used model, recurrent neural network (13%) [2].

B. Limits of Current EEG research

Adamou et al. systematically reviewed 21 papers that used EEG specifically for ADHD research. They point out the shortcomings of EEG for ADHD diagnosis and causality analysis. “The describing of raw EEG data without a theory driven study and standardized protocol, is problematic. It is possible that this approach is inhibiting the development of useful information regarding this potentially valuable method to aid diagnosis” [3]. They conclude that EEG is not yet a reliable form of ADHD diagnosis because of the apparent inconsistencies. Another limitation in ADHD research is that most datasets are not publicly available making it difficult for acquiring large dataset that includes various gender and age groups. Datasets from 42% of studies in the previous review was not publicly available [2].

C. Previous Method: Mutual Information with CNN

In an attempt to solve the “black box” problem, Chen et al. proposed a new method that utilizes mutual information (MI) adjacency matrices for their image construction. They stacked MI matrices to create a three dimensional image-like data. This was used to train their CNN models. They claim that using MI and hand-crafted features allows them to have better interpretability. Their model reached 94.67% accuracy. We wanted to compare their model to our GNN models. Dataset used in this research is different from the one used in ours. For comparable results, their data preprocessing steps and CNN construction was replicated as closely as possible [4].

III. Proposed Method

This paper consists of three main parts. 1) We preprocessed our data using EPOS pipeline and obtained a node adjacency table by calculating MI for every pair of channels. 2) For graph construction, instead of using the MI values as weights, we used them as feature vectors. GNN models were created with varying k-hop values. We used Graph Convolutional Network (GCN), GraphSAGE (SAGE), and Diffusion Convolutional Recurrent Neural Network (DCRNN) [6-7]. 3) To recreate the results in the previous paper, the mutual information table was reorganized and turned into image-like 3 dimensional data in a similar fashion. Previous research proposes four CNN models. We selected their best performing model for comparison.

IV. Experiments

A. Data Collection

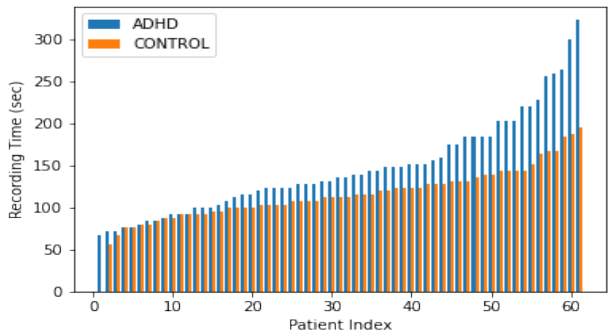
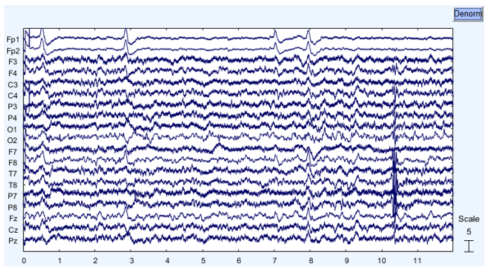
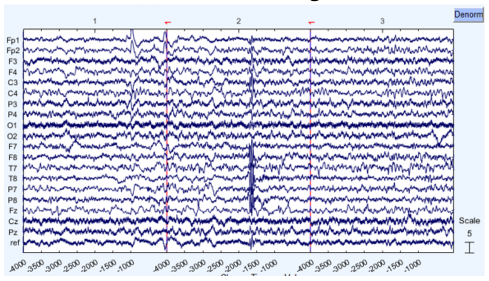


Fig. 1: Recording time comparison.

Due to the scarcity of publicly available EEG recordings of ADHD patients, we had limited options for acquiring the needed dataset. Our dataset was acquired from an open-source IEEE dataport [5]. Our data consists of EEG recordings of 61 children from the ADHDfrom ADHD group and 60 children from the control group (boys and girls,



(a) Raw EEG signal



(b) Preprocessed EEG signal

Fig. 2: Preprocessed signal has been epoched and outlier segments have been rejected.

ages 7-12). The recordings were made with a standard 10-20 EEG setup with 19 channels at 128 Hz. Children were asked to identify and count the number of characters on the computer screen. Recording was stopped once all of the given tasks were completed. The minimum recording time was 57 seconds whereas maximum recording time was 381 seconds. It can be observed that ADHD group generally took longer time to finish the tasks.

B. Preprocessing

EPOS is a "EEG (pre-)processing pipeline to achieve an automated method based on the semi-automated analysis proposed by Delorme and Makeig" [6]. EPOS was designed in an attempt to standardize EEG preprocessing procedures. This pipeline outlines 17 feature extraction steps for analysis. However, only steps one through five were applied since our model does not require extracted features. Detailed description of each steps are as follows:

- Average reference was calculated for detection of bad channels.
- Z-value statistical testing was performed to detect and reject bad channels.
- Rejected channel value was interpolated based on the neighboring channel values.
- All recordings were epoched with 4 seconds window with no overlap. This window size was determined from previous work.
- High pass filtering (1 Hz) was performed to dampen the signal for better interpretability.
- Independent Component Analysis was finally performed to detect and reject bad segments detection.

C. MI Adjacency Matrix Construction

$$\begin{aligned} I(X;Y) &= H(X) - H(X \mid Y) \\ &= H(Y) - H(Y \mid X) \\ &= H(X,Y) - H(X \mid Y) - H(Y \mid X) \\ &= \iint_{x,y} P_{XY}(x,y)\dot{l}og\left[P_X Y(x,y) - P_X(x)P_Y(y)\right] \\ Cov(X;Y) &= \iint_{x,y} xy\left[P_X Y(x,y) \dot{-} P_X(x)P_Y(y)\right] \end{aligned}$$

For each epoch, MI was calculated for every pair of channels. MI considers two signals as a probability distribution, and calculates the distance of joint probability distribution and marginal distribution with joint distribution as weights. Pearson correlation coefficient is frequently used to compare two variables. However, MI was chosen to closely replicate the environment from the proposed CNN model. There were a total of 3988 MI adjacency matrices. Explicit equations for MI in comparison to pearson correlation are shown above.

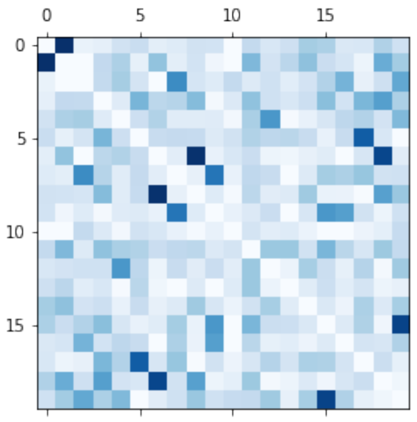


Fig. 3: Sample mutual information adjacency matrix. Graph edges are undirectional because matrix is symmetrical.

D. Graph Construction

MI adjacency matrix represents how close two node values are in probability. It makes more intuitive sense to represent these values as weights of a graph. However, the majority of the current GNN models train to learn node embeddings from feature vectors. So we considered each row in the MI table to be the initial feature vector for GNN layers with unweighted edges. Self-loops were allowed, but diagonal values in the MI matrix were set to 0.

E. Image Construction

Order of the channels in MI matrices is insignificant for graph construction. However, it affects CNN because MI matrices are considered as an image where convolution kernel is applied to nearby values. We followed the previous model's method to divide the nodes into 7 brain regions and reorder and reiterate the channels. This is to have electrodes that are physically close to each other get mapped close by in MI representation as well. Exact electrode channel sequence that we recreated is as follows: $\{3, 11, 1, 17, 2, 12, 4, 6, 14, 16, 10, 8, 19, 8, 18, 6, 4, 17, 3, 5, 7, 19, 9, 15, 7, 13, 5, 3, 11\}$. 9 channels were duplicated and 1 channel was repeated three times resulting in a MI table of size 2929. Four of the MI matrices were then stacked into a three dimensional tensor for CNN training.

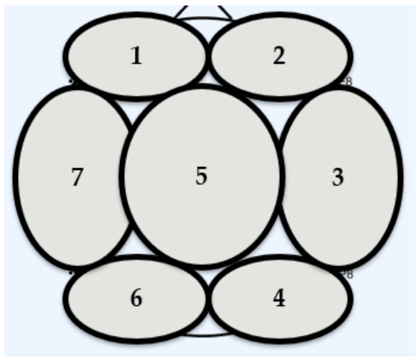


Fig. 4: Nodes were grouped by regions. Starting from 1, we have left frontal, right frontal, right temporal, parietal, left occipital, left temporal

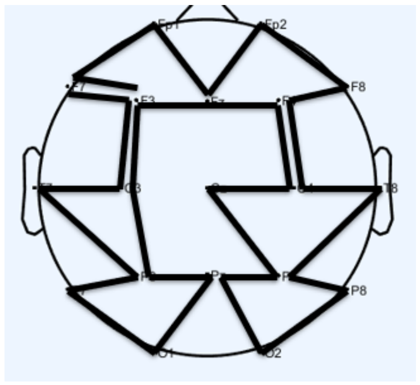


Fig. 5: Detailed node ordering is shown. It is clear that there are overlapping nodes and repeated nodes.

F. Modeling and Training

Here we describe an overview of the GNN convolution layer. K-hop neighborhood of a node refers to all the nodes that can be reached from starting node with k-many edges. Let k denote k^{th}-hop layer; let h_v^{k-1} be a node with index v;let $\mathcal{N}(v)$ denote the set of neighbors of h_v; let $h_{\mathcal{N}(v)} = \{h_u^{k-1} \mid u \in \mathcal{N}(v)\}$ be set of all $k-1$ hop neighbors of h_v. Then, when $AGGREGATE$ function is applied to a node vector, h_v^{k-1}, it combines all the feature vectors in $h_{\mathcal{N}(v)}$ by taking an average. Learnable weight matrix W^k is applied to transform the number of features and find a new embedding for a given node [7]. Each convolution layer was followed by ReLU nonlinearity, and global mean was calculated to obtain graph-level embedding. Dropout layer with probability=0.5 was applied before the linear layer for binary classification. Softmax was used before the final prediction.

GCN is one of the most popular convolution methods for GNNs. It takes the mean of all the neighboring nodes including itself [7]. On the other hand, SAGE concatenates node feature vector with the mean of neighbors before applying weight matrix [8]. We also modeled DCRNN for comparison [9]. GNN is a fast growing area of research and there are more convolution layers that can be taken into consideration. Each of the three models were trained for k-hop values ranging from one to four. Models were trained using 10-fold cross validation with 50 epochs per fold. Test dataset was constructed separately by taking the mean of all MI matrices for each patient. For the CNN model, we replicated proposed model 'a' [4]. Mean of all images for each patient was used for the test dataset for CNN as well. For all models, NAdam

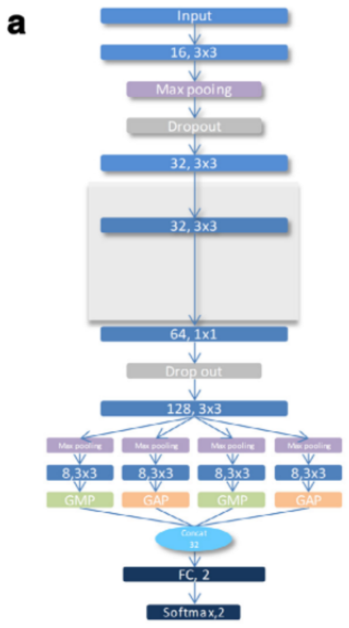


Fig. 6: Comparison CNN Model (Chen et al.)

optimizer was used with the following parameters: learning rate=0.001, betas = (0.9,0.999), momentum decay=0.004. All models were implemented using PyTorch, and PyGeometric.

V. Results and Discussions

TABLE I: Accuracy Comparison of All Models

DCRNN				
K-hop	1	2	3	4
Train	81.09	81.98	85.59	86.35
Validation	80.15	84.67	85.46	89.47
Test	81.59	82.85	85.68	87.14
GCN				
K-hop	1	2	3	4
Train	76.91	82.81	90.28	90.84
Validation	76.13	84.42	88.44	89.95
Test	75.53	81.92	89.92	90.25
SAGE				
K-hop	1	2	3	4
Train	82.26	95.52	98.22	98.86
Validation	82.91	92.46	96.48	96.98
Test	82.05	95.46	97.97	98.65

CNN	
Train	99.33
Validation	100
Test	94.21

We first noticed that CNN suffered overfitting problems more than any GNN models. CNN ob-

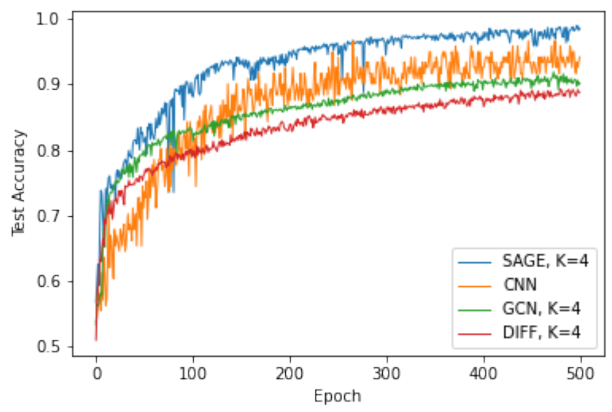


Fig. 7: Best performance models were SAGE, CNN, GCN, DCRNN in order.

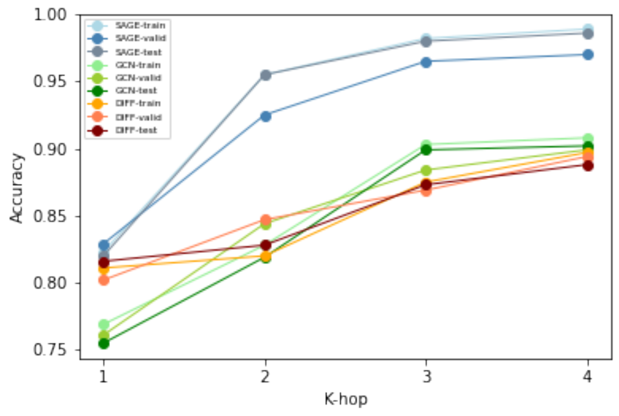


Fig. 8: For all three GNN models, accuracy increased along with k-hop values.

tained 100% accuracy on training and validation set, but test performance was 94.21% showing around 6% gap. This is likely due to the difference in number of layers used in GNN models and CNN model. Implimented CNN model has six convolution layers total whereas GNN models had four at maximum. Also, SAGE outperformed proposed CNN for k-hop values two, three, and four, showing our initial claim. For K=4, SAGE gave test accuracy of 98.65%.

GCN's accuracy was noticeably lower than SAGE. This was also an expected result since GCN "do not scale to large graphs or are designed for whole-graph classification." [8]. We believe that SAGE's structure being less prone to over-smoothing contributed to this result. Interesting observation was that accuracy of all three models seemed to be increasing as k-hop values increased. As k-hop value gets bigger, model training is influenced by nodes that are farther away. Generally, if k-hop values get too big, GNN suffers from over-smoothing problem. We suspect that the nature of the brain connectivity data of ADHD patients allows our models to perform better when information from nodes that are far is taken into consideration.

VI. Further Research

There are number of ways this work can be extended for better accuracy.

- There are number of hyperparameters that can be tuned in the model such as: optimizer, hidden channels, number of epochs.
- To account for individual patient information and prevent data leaking, we will consider group k-fold method.
- Various node-level, graph-level metrics for sensitivity analysis can be added to our feature vector. We will observe how the model adjusts when different types of data are introduced.

There is room for better interpretibility as well.

- First, it will be interesting to observe what happens to model performance as we continue to increase k-hop values.
- By using a different method to construct Adjacency matrix or a feature matrix, we will be able to infer the type of information that is most relevant to ADHD research.
- We will omit some of the channels at random or by region to observe if there is a significant change in model performance. We will be able to infer which nodes or regions contribute more for prediction.
- Other models such as Graph Isomorphism Network, or BrainGNN model can be considered as well. These models provide their own interpretable results that can be reviewed.

References

[1] Danielson, Melissa L et al. "Prevalence of Parent-Reported ADHD Diagnosis and Associated Treatment Among U.S. Children and Adolescents, 2016." Journal of clinical child and adolescent psychology : the official journal for the Society of Clinical Child and Adolescent Psychology, American Psychological Association, Division 53 vol. 47,2 (2018): 199-212. doi:10.1080/15374416.2017.1417860

[2] Y. Roy, H. Banville, I. Albuquerque, A. Gramfort, T. H. Falk, and J. Faubert, "Deep learning-based Electroencephalography Analysis: A systematic review," Journal of Neural Engineering, vol. 16, no. 5, p. 051001, 2019.

[3] M. Adamou, T. Fullen, and S. L. Jones, "EEG for diagnosis of ADULT ADHD: A systematic review with narrative analysis," Frontiers in Psychiatry, vol. 11, 2020.

[4] H. Chen, Y. Song, and X. Li, "A deep learning framework for identifying children with ADHD using an EEG-based Brain Network," Neurocomputing, vol. 356, pp. 83–96, 2019.

[5] Ali Motie Nasrabadi, Armin Allahverdy, Mehdi Samavati, Mohammad Reza Mohammadi, June 10, 2020, "EEG data for ADHD / Control children", IEEE Dataport, doi: https://dx.doi.org/10.21227/rzfh-zn36.

[6] J. Rodrigues, M. Weiß, J. Hewig, and J. J. Allen, "EPOS: EEG processing open-source scripts," Frontiers in Neuroscience, vol. 15, 2021.

[7] Kipf, Thomas N., and Max Welling. "Semi-supervised classification with graph convolutional networks." arXiv preprint arXiv:1609.02907 (2016).

[8] Hamilton, Will, Zhitao Ying, and Jure Leskovec. "Inductive representation learning on large graphs." Advances in neural information processing systems 30 (2017).

[9] Li, Yaguang, et al. "Diffusion convolutional recurrent neural network: Data-driven traffic forecasting." arXiv preprint arXiv:1707.01926 (2017).

[10] Demir, Andac, et al. "EEG-GNN: Graph Neural Networks for Classification of Electroencephalogram (EEG) Signals." 2021 43rd Annual International Conference of the IEEE Engineering in Medicine Biology Society (EMBC). IEEE, 2021.

In [1]:
```
import numpy as np
import pandas as pd
from sklearn.feature_selection import mutual_info_regression
import os
import matplotlib.pyplot as plt
from datetime import datetime
from statistics import mean, median
%matplotlib inline
```

In [2]:
```
read_dir_adhd = r'C:\Users\yl646\Documents\ADHD_Research\DATA\OUTPUT\step_6_test_compl
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_te
epoch_adhd_dir = r'C:\Users\yl646\Documents\ADHD_Research\DATA\OUTPUT\step_6_test_comp


read_dir_control = r'C:\Users\yl646\Documents\ADHD_Research\DATA\OUTPUT\step_6_test_co
mi_dir_control   = r'C:\Users\yl646\Documents\ADHD_research\DATA\OUTPUT\step_6_test_co
mi_dir_control_overlap   = r'C:\Users\yl646\Documents\ADHD_research\DATA\OUTPUT\step_6
epoch_control_dir = r'C:\Users\yl646\Documents\ADHD_Research\DATA\OUTPUT\step_6_test_c
```

Create Mutual Information Table with 20 Channels for Graph Construction

In [3]:
```
# create ADHD dataset with correct dimension
list_of_ADHD = []
total_epoch = 0
n_epoch_each_adhd = []
n_epoch_each_control = []

# Each file has column labels where first column is the time stamp.
# Data was epoched at 4 second window, so time stampt is repeated values 0~512
for i in os.listdir(read_dir_adhd):
    df = pd.read_csv(read_dir_adhd+"\\"+i)
    arr = df.to_numpy()
    num_epoch = arr.shape[0] / 512 # 512 data points in 1 epoch (4 sec x 128 hz)

    n_epoch_each_adhd.append(num_epoch)
    total_epoch += num_epoch

    list_of_epoch = []
    for i in range(int(num_epoch)):
        single_epoch = arr[ i*512 : (i+1)*512 , 1: ].transpose() # slice for each epoc
        list_of_epoch.append(single_epoch)
    list_of_ADHD.append(list_of_epoch)

all_epoch = []
for patient in list_of_ADHD:
    for epoch in patient:
        all_epoch.append(epoch)
ADHD_dataset = np.stack(all_epoch)

print('Total Epoch: ',total_epoch)
print('ADHD dataset dimension: ',ADHD_dataset.shape, '(epoch, channel, time)')
np.save(epoch_adhd_dir, n_epoch_each_adhd) # save number of epochs per patient. this f
```

```
Total Epoch:  2231.0
ADHD dataset dimension:  (2231, 20, 512) (epoch, channel, time)
```

In [5]:

```
# create CONTROL dataset with correct dimension
# Same process is repeated for control group
list_of_CONTROL = []
total_epoch = 0
n_epoch_each_control = []

for i in os.listdir(read_dir_control):
    df = pd.read_csv(read_dir_control+"\\"+i)
    arr = df.to_numpy()
    num_epoch = arr.shape[0] / 512

    n_epoch_each_control.append(num_epoch)
    total_epoch += num_epoch

    list_of_epoch = []
    total_epoch += num_epoch
    for i in range(int(num_epoch)):
        single_epoch = arr[ i*512 : (i+1)*512 , 1: ].transpose()
        list_of_epoch.append(single_epoch)
    list_of_CONTROL.append(list_of_epoch)

all_epoch = []
for patient in list_of_CONTROL:
    for epoch in patient:
        all_epoch.append(epoch)

CONTROL_dataset = np.stack(all_epoch)

print('Total Epoch: ',total_epoch)
print('CONTROL dataset dimension: ',CONTROL_dataset.shape, '(epoch, channel, time)')
np.save(epoch_control_dir, n_epoch_each_control)
```

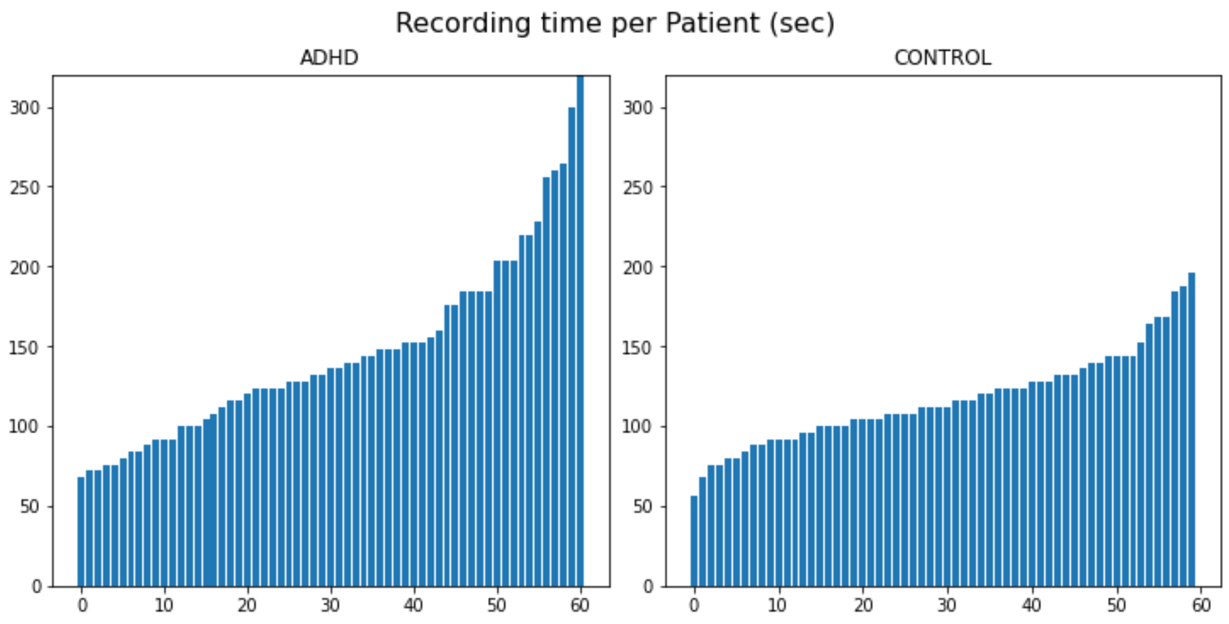
```
Total Epoch:  3514.0
CONTROL dataset dimension:  (1757, 20, 512) (epoch, channel, time)
```

In [8]:

```
n_adhd = list(range(len(n_epoch_each_adhd)))
n_control = list(range(len(n_epoch_each_control)))

t_adhd=np.multiply(n_epoch_each_adhd,4)
t_control = np.multiply(n_epoch_each_control,4)

print(f'Mean Recording Time ADHD: {mean(t_adhd):.2f}')
print(f'Mean Recording Time CONTROL: {mean(t_control):.2f}')
print(f'Median Recording Time ADHD: {median(t_adhd):.2f}')
print(f'Median Recording Time CONTROL: {median(t_control):.2f}')
print(f'Min Recording Time ADHD: {min(t_adhd):.2f}')
print(f'Min Recording Time CONTROL: {min(t_control):.2f}')
print(f'Max Recording Time ADHD: {max(t_adhd):.2f}')
print(f'Max Recording Time CONTROL: {max(t_control):.2f}')
```

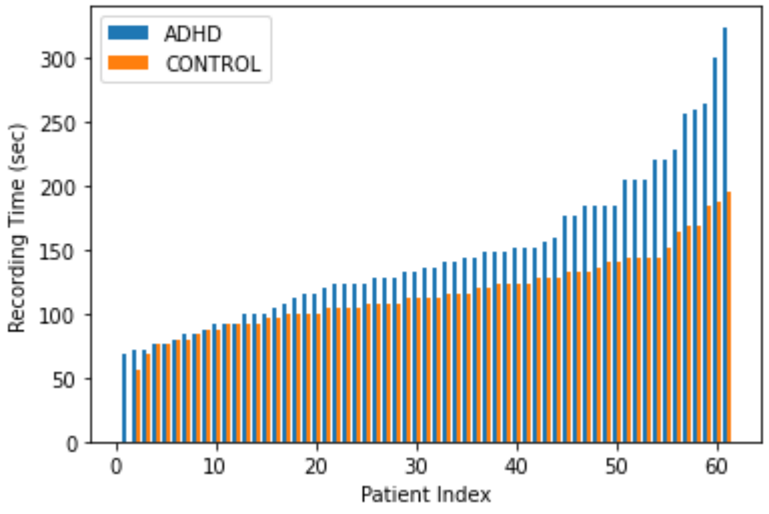
```
Mean Recording Time ADHD: 146.30
Mean Recording Time CONTROL: 117.13
Median Recording Time ADHD: 136.00
Median Recording Time CONTROL: 112.00
Min Recording Time ADHD: 68.00
Min Recording Time CONTROL: 56.00
Max Recording Time ADHD: 324.00
Max Recording Time CONTROL: 196.00
```

In [9]:
```
fig, axs = plt.subplots(1, 2, constrained_layout=True, figsize=(10,5))
fig.suptitle('Recording time per Patient (sec)', fontsize=16)

axs[0].bar(n_adhd, sorted(list(t_adhd)))
axs[0].set_title('ADHD')
axs[0].set_ylim([0,320])

axs[1].bar(n_control, sorted(list(t_control)))
axs[1].set_title('CONTROL')
axs[1].set_ylim([0,320])

fig.savefig('recording_time_per_patient.png')
```



In [7]:
```
x=list(range(1, 62))
x1=[i-0.2 for i in x]
x2=[i+0.2 for i in x]
t_control=list(t_control)
if len(t_control)!=61:
    t_control.append(0)
print(len(x1))
print(len(x2))
print(len(list(t_adhd)))
print(len(t_control))
plt.bar(x1, sorted(list(t_adhd)), label='ADHD',width=0.4)
plt.bar(x2, sorted(t_control), label='CONTROL',width=0.4)
plt.xlabel('Patient Index')
plt.ylabel('Recording Time (sec)')
plt.legend()
plt.savefig('recording_time.png')
plt.show()
```

```
61
61
61
61
```



Create MI Table for GNN Construction

In [6]:
```
# create mutual information table
(epochs, channels, frames) = ADHD_dataset.shape
mi_table = np.zeros([epochs, channels, channels])
for j in range(epochs):
    if j%10==0:
        now = datetime.now()
        current_time = now.strftime("%H:%M:%S")
        print("ADHD", j, current_time)
    example = ADHD_dataset[j,:,:]
    for k in range(channels):
        x = np.delete(example,k,axis=0)
        y = example[k,:]
        mi = mutual_info_regression(x.transpose(),y) #  This is where MI is calculatec
        mi = np.insert(mi,k,0)  # assign 0 for position (k,k) (self-loop value 0)

        mi_table[j,k,:] = mi

# mi_table dimension: (patients, epochs, channel, channel)
np.save(mi_dir_adhd, mi_table)


# create mutual information table
(epochs, channels, frames) = CONTROL_dataset.shape
mi_table = np.zeros([epochs, channels, channels])
for j in range(epochs):
    if j%10==0:
        now = datetime.now()
        current_time = now.strftime("%H:%M:%S")
        print("CONTROL", j, current_time)
    example = CONTROL_dataset[j,:,:]
    for k in range(channels):
        x = np.delete(example,k,axis=0)
        y = example[k,:]
        mi = mutual_info_regression(x.transpose(),y)
        mi = np.insert(mi,k,0)
        mi_table[j,k,:] = mi

# mi_table dimension: (patients, epochs, channel, channel)
np.save(mi_dir_control, mi_table)
```

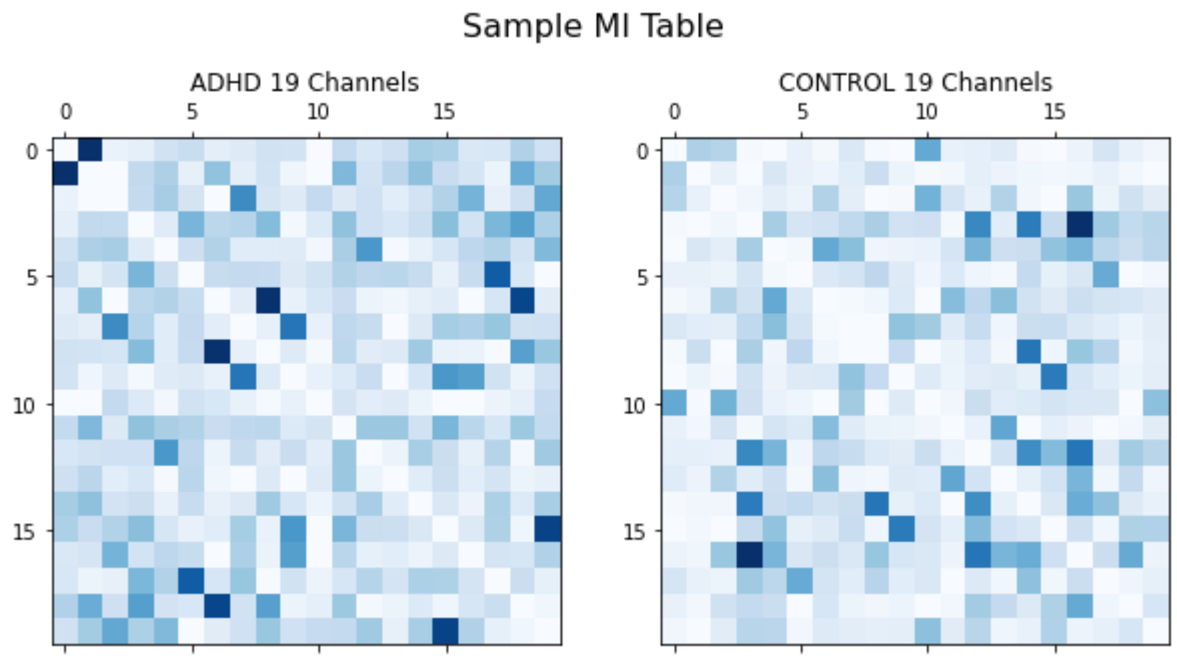
```
CONTROL 1360 19:05:45
CONTROL 1370 19:05:57
CONTROL 1380 19:06:08
CONTROL 1390 19:06:20
CONTROL 1400 19:06:32
CONTROL 1410 19:06:44
CONTROL 1420 19:06:56
CONTROL 1430 19:07:08
CONTROL 1440 19:07:20
CONTROL 1450 19:07:31
CONTROL 1460 19:07:43
CONTROL 1470 19:07:55
CONTROL 1480 19:08:06
CONTROL 1490 19:08:18
CONTROL 1500 19:08:29
CONTROL 1510 19:08:41
CONTROL 1520 19:08:53
CONTROL 1530 19:09:05
CONTROL 1540 19:09:16
CONTROL 1550 19:09:28
CONTROL 1560 19:09:40
CONTROL 1570 19:09:51
CONTROL 1580 19:10:03
CONTROL 1590 19:10:15
CONTROL 1600 19:10:27
CONTROL 1610 19:10:39
CONTROL 1620 19:10:50
CONTROL 1630 19:11:02
CONTROL 1640 19:11:14
CONTROL 1650 19:11:26
CONTROL 1660 19:11:37
CONTROL 1670 19:11:49
CONTROL 1680 19:12:00
CONTROL 1690 19:12:12
CONTROL 1700 19:12:23
CONTROL 1710 19:12:35
CONTROL 1720 19:12:46
CONTROL 1730 19:12:58
CONTROL 1740 19:13:10
CONTROL 1750 19:13:21
```

Sample MI table

In [7]:
```
a= np.load(mi_dir_adhd)
b = np.load(mi_dir_control)
print(f'ADHD MI table shape: {a.shape}')
print(f'CONTROL MI table shape: {b.shape}')
fig, axs = plt.subplots(1,2, figsize=(10,5.5))
fig.suptitle('Sample MI Table', fontsize=16)
axs[0].matshow(a[1700,:,:], cmap=plt.cm.Blues)
axs[0].set_title('ADHD 19 Channels')
axs[1].matshow(b[1700,:,:], cmap=plt.cm.Blues)
axs[1].set_title('CONTROL 19 Channels')
```

```
ADHD MI table shape: (2231, 20, 20)
CONTROL MI table shape: (1757, 20, 20)
```

Out[7]:
```
Text(0.5, 1.0, 'CONTROL 19 Channels')
```



Create Mutual Information with Overlap for CNN model

Create a overlapped numpy channel array with preprocessed signals

In [8]:
```
#  Dataset contained channel label file. I was able to manually match the numeric valu
overlap_seq = [3,11,1,17,2,12,4,6,14,16,10,8,19,8,18,6,4,17,3,5,7,19,9,15,7,13,5,3,11]

(epochs, channel, time)=ADHD_dataset.shape
overlap_channels = len(overlap_seq)

ADHD_overlap = np.zeros((epochs,overlap_channels,512))

for epoch in range(epochs):  # for each epoch, rearange channels.
    for i in range(overlap_channels):
        index = overlap_seq[i]-1
        ADHD_overlap[epoch, i, :] = ADHD_dataset[epoch,index,:]

print('ADHD overlap dimension: ', ADHD_overlap.shape)
```

```
ADHD overlap dimension:  (2231, 29, 512)
```

In [9]:
```
# Repeat for contol group
overlap_seq = [3,11,1,17,2,12,4,6,14,16,10,8,19,8,18,6,4,17,3,5,7,19,9,15,7,13,5,3,11]

(epochs, channel, time)=CONTROL_dataset.shape
overlap_channels = len(overlap_seq)

CONTROL_overlap = np.zeros((epochs,overlap_channels,512))

for epoch in range(epochs):
    for i in range(overlap_channels):
        index = overlap_seq[i]-1
        CONTROL_overlap[epoch, i, :] = CONTROL_dataset[epoch,index,:]

print('CONTROL overlap dimension: ', CONTROL_overlap.shape)
```

```
CONTROL overlap dimension:  (1757, 29, 512)
```

Create MI table with overlap. 29x29 adjacency matrix

In [10]:

```
overlap_seq = np.array([3,11,1,17,2,12,4,6,14,16,10,8,19,8,18,6,4,17,3,5,7,19,9,15,7,1

# create mutual information table
(epochs, channels, frames) = ADHD_overlap.shape
mi_table = np.zeros([epochs, channels, channels])
for j in range(epochs):
    if j%10==0:
        now = datetime.now()
        current_time = now.strftime("%H:%M:%S")
        print("ADHD overlap", j, current_time)

    example = ADHD_overlap[j,:,:]

    for k in range(channels):
        pos = np.where(np.array(overlap_seq) == overlap_seq[k])[0] # find indices for
        x = np.delete(example,k,axis=0)
        y = example[k,:]
        mi = mutual_info_regression(x.transpose(),y)  # calculate MI here
        mi = np.insert(mi,k,0) # assign 0 for position (k,k) (self-loop value 0)
        mi_table[j,k,:] = mi
        mi_table[j,k,pos] = 0 # assign 0 for other self loop indices that were found

# mi_table dimension: (patients, epochs, channel, channel)
np.save(mi_dir_adhd_overlap, mi_table)


# create mutual information table
(epochs, channels, frames) = CONTROL_overlap.shape
mi_table = np.zeros([epochs, channels, channels])
for j in range(epochs):
    if j%10==0:
        now = datetime.now()
        current_time = now.strftime("%H:%M:%S")
        print("CONTROL overlap", j, current_time)

    example = CONTROL_overlap[j,:,:]

    for k in range(channels):
        pos = np.where(np.array(overlap_seq) == overlap_seq[k])[0] # find indices for
        x = np.delete(example,k,axis=0)
        y = example[k,:]
        mi = mutual_info_regression(x.transpose(),y)
        mi = np.insert(mi,k,0) # assign 0 for position (k,k)
        mi_table[j,k,:] = mi
        mi_table[j,k,pos] = 0 # assign 0 for other self loop indices that were found

# mi_table dimension: (patients, epochs, channel, channel)
np.save(mi_dir_control_overlap, mi_table)
```

```
CONTROL overlap 1360 21:42:51
CONTROL overlap 1370 21:43:16
CONTROL overlap 1380 21:43:40
CONTROL overlap 1390 21:44:05
CONTROL overlap 1400 21:44:30
CONTROL overlap 1410 21:44:54
CONTROL overlap 1420 21:45:20
CONTROL overlap 1430 21:45:45
CONTROL overlap 1440 21:46:10
CONTROL overlap 1450 21:46:35
CONTROL overlap 1460 21:47:00
CONTROL overlap 1470 21:47:24
CONTROL overlap 1480 21:47:50
CONTROL overlap 1490 21:48:14
CONTROL overlap 1500 21:48:39
CONTROL overlap 1510 21:49:03
CONTROL overlap 1520 21:49:28
CONTROL overlap 1530 21:49:54
CONTROL overlap 1540 21:50:19
CONTROL overlap 1550 21:50:44
CONTROL overlap 1560 21:51:09
CONTROL overlap 1570 21:51:33
CONTROL overlap 1580 21:51:58
CONTROL overlap 1590 21:52:22
CONTROL overlap 1600 21:52:47
CONTROL overlap 1610 21:53:12
CONTROL overlap 1620 21:53:36
CONTROL overlap 1630 21:54:01
CONTROL overlap 1640 21:54:26
CONTROL overlap 1650 21:54:51
CONTROL overlap 1660 21:55:15
CONTROL overlap 1670 21:55:40
CONTROL overlap 1680 21:56:05
CONTROL overlap 1690 21:56:30
CONTROL overlap 1700 21:56:55
CONTROL overlap 1710 21:57:19
CONTROL overlap 1720 21:57:44
CONTROL overlap 1730 21:58:09
CONTROL overlap 1740 21:58:33
CONTROL overlap 1750 21:58:57
```

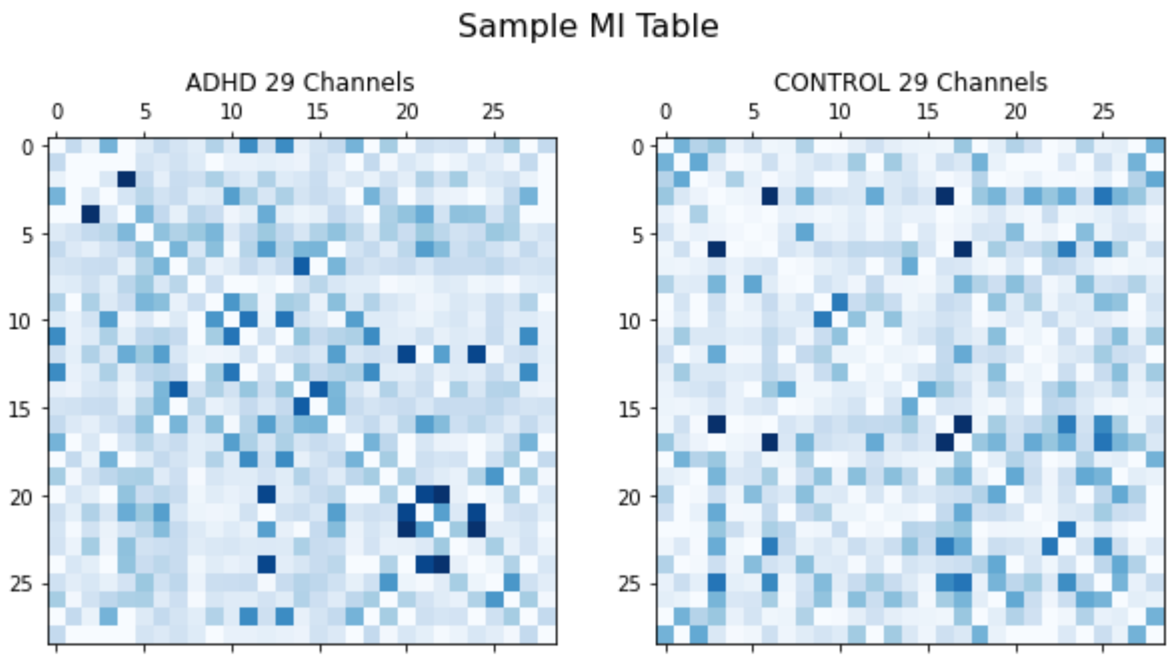
In [11]:

```
a= np.load(mi_dir_adhd_overlap)
b = np.load(mi_dir_control_overlap)
print(f'ADHD MI table shape: {a.shape}')
print(f'CONTROL MI table shape: {b.shape}')
fig2, axs2 = plt.subplots(1,2, figsize=(10,5.5))
fig2.suptitle('Sample MI Table', fontsize=16)
axs2[0].matshow(a[1700,:,:], cmap=plt.cm.Blues)
axs2[0].set_title('ADHD 29 Channels')
axs2[1].matshow(b[1700,:,:], cmap=plt.cm.Blues)
axs2[1].set_title('CONTROL 29 Channels')
```

```
ADHD MI table shape: (2231, 29, 29)
CONTROL MI table shape: (1757, 29, 29)
```

Out[11]:

```
Text(0.5, 1.0, 'CONTROL 29 Channels')
```


Sample MI Table
ADHD 29 Channels
CONTROL 29 Channels
0
5
10
15
20
25

SAGE model with k-fold 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, temp) = mi_table.shape

    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]])
                edges_np = np.concatenate((edges_np,edge),axis=1)
                #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)
        dataset_graph.append(graph)
    return dataset_graph
```

Graph - Training/Validation

This dataset will be split into training set and validation set

In [8]:
```
ADHD_mi = np.load(mi_dir_adhd)
CONTROL_mi = np.load(mi_dir_control)

adhd_train_val_graph = getGraph(ADHD_mi, y=1)
control_train_val_graph = getGraph(CONTROL_mi, y=0)

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 and 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

num_img_ADHD = int(ADHD_epochs/n_ch)
num_img_CONTROL = int(CONTROL_epochs/n_ch)
n_img = num_img_ADHD+num_img_CONTROL

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 in
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)

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 in
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
```

$\longrightarrow$ CNN

```
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
```
$\longrightarrow$ SAGE k=1

```
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
```

$\rightarrow$ SAGE k=2

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
```
$\rightarrow$ SAGE k=3

```
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→SAGE = k=4

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
```

$\rightarrow$ GCN $k=1$

```
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
```

$\rightarrow$ GCN $k=2$

```
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
```
$\rightarrow$ GCN $k=3$

```
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
```

$\rightarrow$ GCN k=4

```
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
```
$\rightarrow$ DIFF k=1

```
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
```

$\rightarrow$ DIFF k=2

```
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 → DIFF k=3

Elapsed Time:  13594.784582138062
```

In [18]:

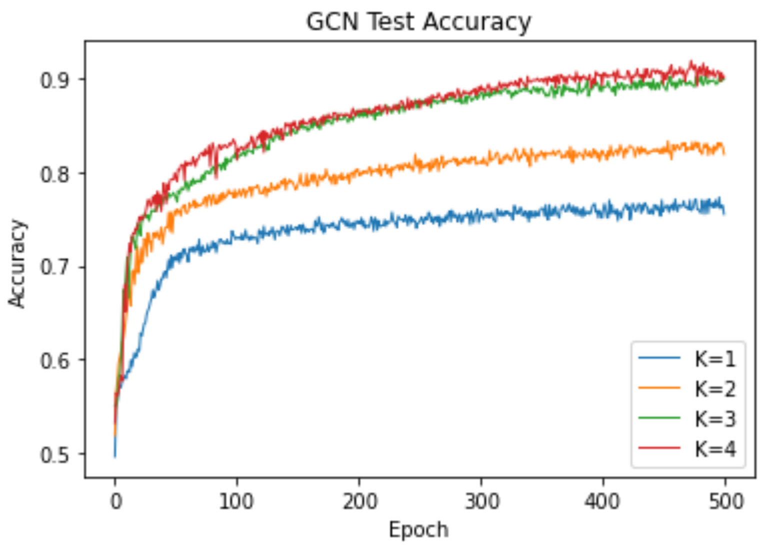
```
main('DIFF', dataset_graph, test_graph, model_dir, result_dir, k_hop=4, k_fold=10, n_e
Fold: 10, Epoch: 031, Train: 0.8772, Valid: 0.8869, Test: 0.8914
Fold: 10, Epoch: 032, Train: 0.8811, Valid: 0.8794, Test: 0.8889
Fold: 10, Epoch: 033, Train: 0.8827, Valid: 0.8995, Test: 0.8819
Fold: 10, Epoch: 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
```
→ DIFF k=4

```
Elapsed Time:  16847.374128580093
```

```
In [1]: import pandas as pd
        import matplotlib.pyplot as plt
        import numpy as np

        result_dir = r'C:\Users\yl646\Documents\ADHD_Research\DATA\OUTPUT\step_6_test_complete
        filename = result_dir+r'\kfold_'
```

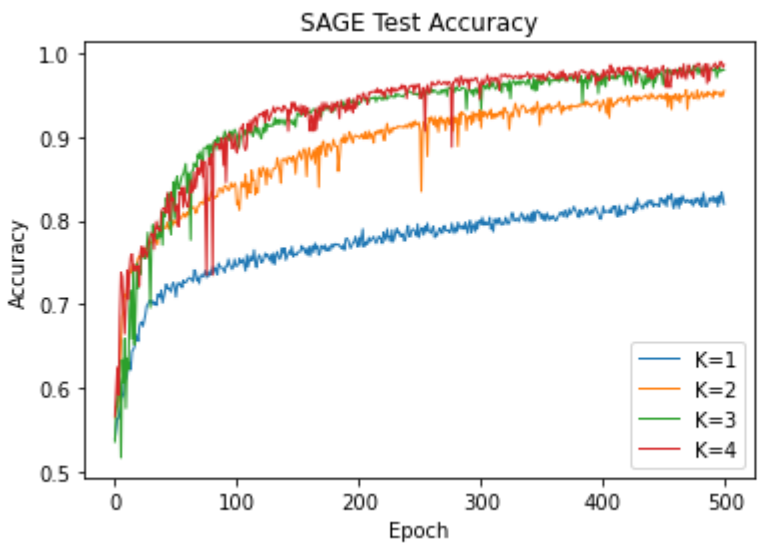
```
In [2]: def getValue(model, k_hop=1):
            if model == 'CNN':
                results = pd.read_csv(filename+f'{model}_ndam_epo_50.csv')
            else:
                results = pd.read_csv(filename+f'{model}_k_{k_hop}_ndam_epo_50.csv')
            test = results["Test"]
            train = results["Train"]
            valid = results["Valid"]
            return train, valid, test
```

```
In [3]: (train11, valid11, test11) = getValue('GCN', 1)
        (train12, valid12, test12) = getValue('GCN', 2)
        (train13, valid13, test13) = getValue('GCN', 3)
        (train14, valid14, test14) = getValue('GCN', 4)

        plt.title('GCN Test Accuracy')
        plt.plot(test11, label='K=1', linewidth =1)
        plt.plot(test12, label='K=2', linewidth =1)
        plt.plot(test13, label='K=3', linewidth =1)
        plt.plot(test14, label='K=4', linewidth =1)

        plt.xlabel('Epoch')
        plt.ylabel('Accuracy')
        plt.legend()
        plt.savefig('GCN_Test.png')
        plt.show()
```

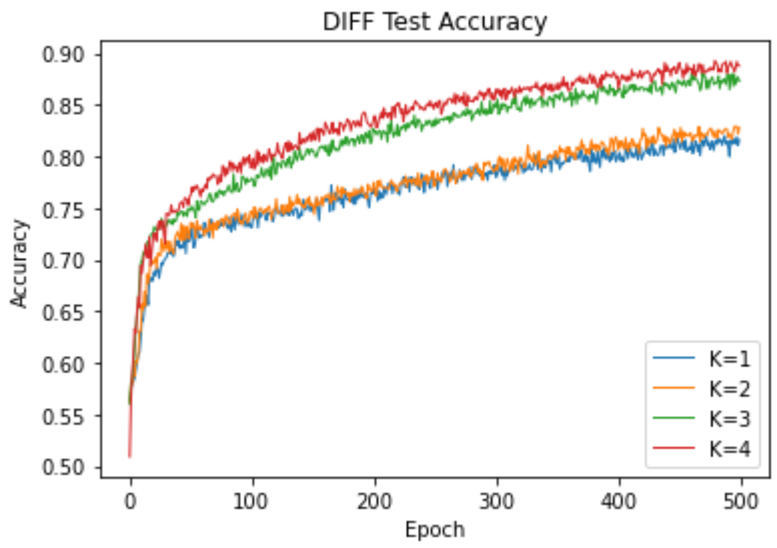


```
In [4]: (train21, valid21, test21) = getValue('SAGE', 1)
        (train22, valid22, test22) = getValue('SAGE', 2)
        (train23, valid23, test23) = getValue('SAGE', 3)
        (train24, valid24, test24) = getValue('SAGE', 4)
```

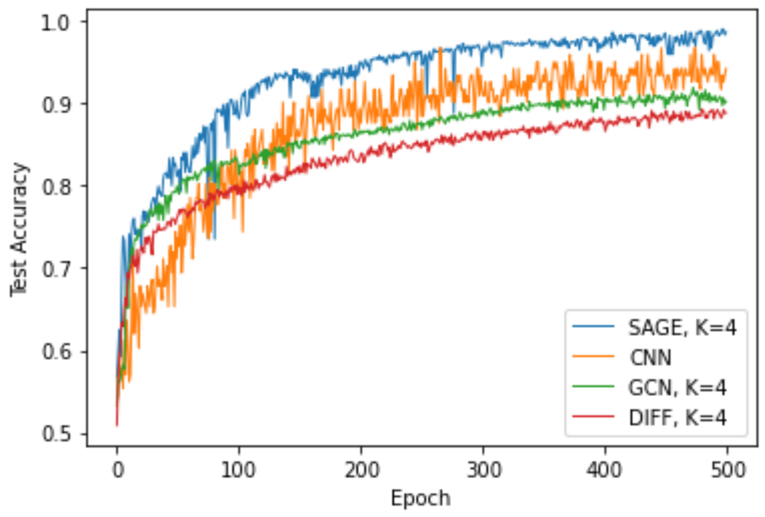
```
plt.title('SAGE Test Accuracy')
plt.plot(test21, label='K=1', linewidth =1)
plt.plot(test22, label='K=2', linewidth =1)
plt.plot(test23, label='K=3', linewidth =1)
plt.plot(test24, label='K=4', linewidth =1)

plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.savefig('SAGE_Test.png')
plt.show()
```



In [5]:
```
(train31, valid31, test31) = getValue('DIFF', 1)
(train32, valid32, test32) = getValue('DIFF', 2)
(train33, valid33, test33) = getValue('DIFF', 3)
(train34, valid34, test34) = getValue('DIFF', 4)

plt.title('DIFF Test Accuracy')
plt.plot(test31, label='K=1', linewidth =1)
plt.plot(test32, label='K=2', linewidth =1)
plt.plot(test33, label='K=3', linewidth =1)
plt.plot(test34, label='K=4', linewidth =1)

plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.savefig('DIFF_Test.png')
plt.show()
```



In [9]:
```
(train4, valid4, test4) = getValue('CNN')

#plt.title('Model Comparison')
plt.plot(test24, label='SAGE, K=4', linewidth =1)
plt.plot(test4, label='CNN', linewidth =1)
plt.plot(test14, label='GCN, K=4', linewidth =1)
plt.plot(test34, label='DIFF, K=4', linewidth =1)


plt.xlabel('Epoch')
plt.ylabel('Test Accuracy')
plt.legend()
plt.savefig('Model_Comparison.png')
plt.show()
```

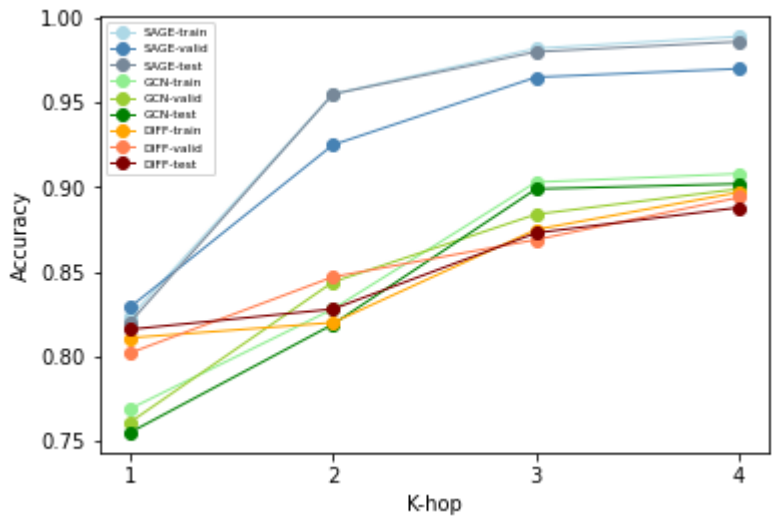


In [7]:
```
gcn_train = [train11.iloc[-1], train12.iloc[-1], train13.iloc[-1], train14.iloc[-1]]
gcn_valid = [valid11.iloc[-1], valid12.iloc[-1], valid13.iloc[-1], valid14.iloc[-1]]
gcn_test = [test11.iloc[-1], test12.iloc[-1], test13.iloc[-1], test14.iloc[-1]]

sage_train = [train21.iloc[-1], train22.iloc[-1], train23.iloc[-1], train24.iloc[-1]]
sage_valid = [valid21.iloc[-1], valid22.iloc[-1], valid23.iloc[-1], valid24.iloc[-1]]
sage_test = [test21.iloc[-1], test22.iloc[-1], test23.iloc[-1], test24.iloc[-1]]

diff_train = [train31.iloc[-1], train32.iloc[-1], train33.iloc[-1], train34.iloc[-1]]
```

```
diff_valid = [valid31.iloc[-1], valid32.iloc[-1], valid33.iloc[-1], valid34.iloc[-1]]
diff_test = [test31.iloc[-1], test32.iloc[-1], test33.iloc[-1], test34.iloc[-1]] #, te

cnn_train = [train4.iloc[-1], train4.iloc[-1], train4.iloc[-1], train4.iloc[-1]]
cnn_valid = [valid4.iloc[-1], valid4.iloc[-1], valid4.iloc[-1], valid4.iloc[-1]]
cnn_test = [test4.iloc[-1], test4.iloc[-1], test4.iloc[-1], test4.iloc[-1]]

x=[1,2,3,4]
```

In [8]:

```
#plt.title('K-hop Comparison')
plt.plot(x, sage_train, label='SAGE-train', linewidth =1, color='lightblue', marker='c
plt.plot(x, sage_valid, label='SAGE-valid', linewidth =1, color='steelblue', marker='c
plt.plot(x, sage_test, label='SAGE-test', linewidth =1, color='lightslategrey', marker
plt.plot(x, gcn_train, label='GCN-train', linewidth =1, color='lightgreen', marker='o'
plt.plot(x, gcn_valid, label='GCN-valid', linewidth =1, color='yellowgreen', marker='c
plt.plot(x, gcn_test, label='GCN-test', linewidth =1, color='g', marker='o')
plt.plot(x, diff_train, label='DIFF-train', linewidth =1, color='orange', marker='o')
plt.plot(x, diff_valid, label='DIFF-valid', linewidth =1, color='coral', marker='o')
plt.plot(x, diff_test, label='DIFF-test', linewidth =1, color='maroon', marker='o')
plt.xlabel('K-hop')
plt.xticks(x,x)
plt.ylabel('Accuracy')
plt.legend(prop={'size': 6})
plt.savefig('k_hop_Comparison.png')
plt.show()
```


SAGE-train
SAGE-valid
SAGE-test
GCN-train
GCN-valid
GCN-test
DIFF-train
DIFF-valid
DIFF-test
Accuracy
K-hop