# GRAPH
# NEURAL NETWORK

Elif Karataş
2016510040
15.06.2021

# Table Of Contents

## Explanation about the Topic and Definition of the Problem

Graph is the name given to a set of objects to which pairs of objects are related. Objects correspond to mathematical abstractions called vertices (also called nodes or points), and each of the corresponding knot pairs is called an edge. Typically graphs are shown diagrammatically as a set of points or circles for nodes, joined by lines or curves for their edges.

There are many different types of graph. The properties of each of these different graph types differ from each other.

For example, whether a graph is direct or undirect, the number of edges and vertex, the number of degrees, etc. Many variables such as separate graphs from each other.

In the study in this project, it will be determined which graph type the randomly created graphs with help of library belong to.

## What are the Real Life Problems the Method Solves

Today, graphs are used to solve many problems.

For example, graph types determine which internet network model will be designed. These are called topologies. They are defined in 9 different subtypes and each actually represents a graph structure. To give another example, urban transportation networks are also examples of graph type. Each of the bus stops, routes and transfer points is an element of the graph. In health and science, it is used in molecular sequences. Which atoms will correlate with which atoms and the structures formed by molecules can be example of graph types.

## Description of the Dataset

The name of the data set is MiniGCDataset (Mini Graph Classification Dataset).

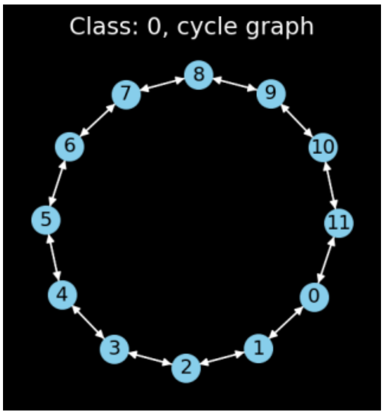
In the work to be done, it is the main purpose to determine the graph types.

Therefore, the dataset I will use contains different types of graphs. This data set, which has 8 different graph types, is obtained from a library. In this library, 8 different types of graphs are generating with randomly edges and nodes.

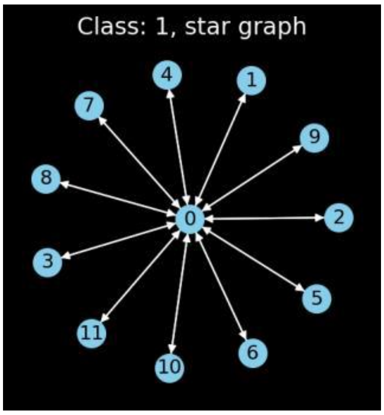
**Some Information About This Dataset**

The dataset contains 8 different types of graphs:

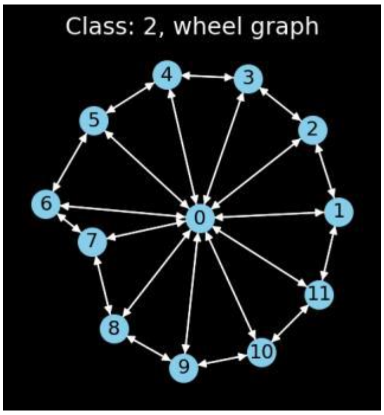
class 0 : cycle graph



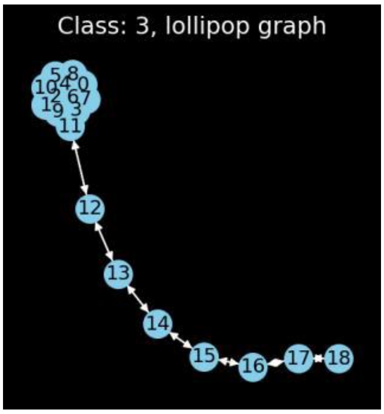

class 1 : star graph



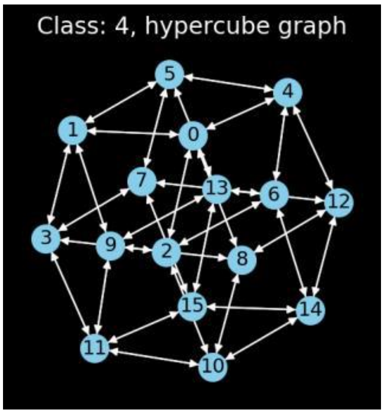

class 2 : wheel graph



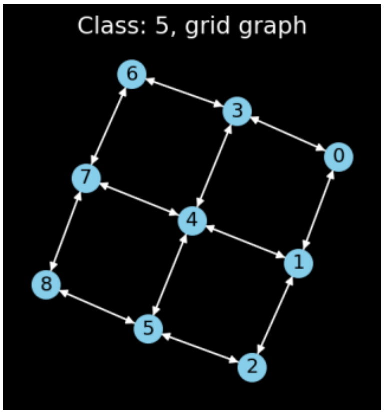

class 3 : lollipop graph



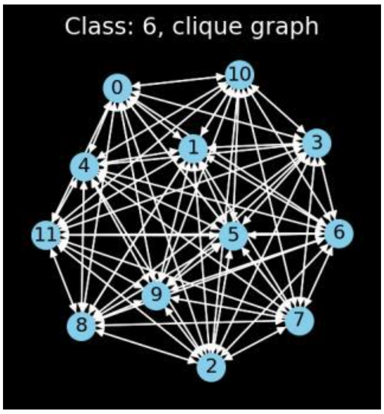

class 4 : hypercube graph



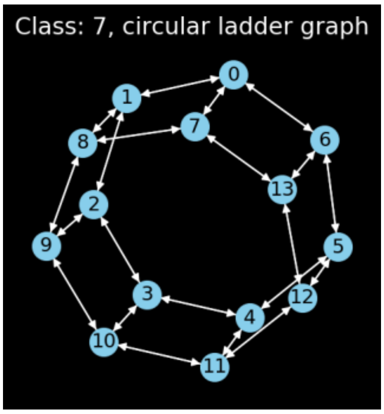

class 5 : grid graph




class 6 : clique graph




class 7 : circular ladder graph




Parameter for MiniGCDataset(num_graphs,min_num_v,max_num_v, seed=0):

- num_graphs: int (Number of graphs in this dataset)

- min_num_v: int (Minimum number of nodes for graphs)
- max_num_v: int (Maximum number of nodes for graphs)
- seed : int, default is 0
- Random seed for data generation

Attributes for MiniGCDataset:

- num_graphs : int (Number of graphs)
- min_num_v : int (The minimum number of nodes)
- max_num_v : int (The maximum number of nodes)
- num_classes : int (The number of classes) [1]

In order to use the MiniGCDataset, we first need to import the DGLGraph from DGL (Deep Graph Library) library.

It is base graph class. It helps to create graphs.

Node and edge features are stored as a dictionary from the feature name to the feature data (in tensor).

DGL graph accepts graph data of multiple formats:

- NetworkX graph,
- scipy matrix,
- DGLGraph.

If the input graph data is DGLGraph, the constructed DGLGraph only contains its graph index [2].

**Graph Normalization**

Before generating graphs, some normalization processes were defined. These are ensuring that number of nodes and number of edges take values in the range 0-1. These operations are done with the help of the PyTorch library [3].

```
In [45]: # normalization
def collate(samples):
    graphs, labels = map(list, zip(*samples)) # samples is a list of pairs (graph, label).
    labels = torch.tensor(labels)

    tab_sizes_n = [ graphs[i].number_of_nodes() for i in range(len(graphs))] # graph sizes
    print(tab_sizes_n)
    tab_snorm_n = [ torch.FloatTensor(size,1).fill_(1./float(size)) for size in tab_sizes_n ]
    print(tab_snorm_n)
    snorm_n = torch.cat(tab_snorm_n).sqrt() # graph size normalization
    print(snorm_n)
```

```
[14]
[tensor([[0.0714],
        [0.0714],
        [0.0714],
        [0.0714],
        [0.0714],
        [0.0714],
        [0.0714],
        [0.0714],
        [0.0714],
        [0.0714],
        [0.0714],
        [0.0714],
        [0.0714],
        [0.0714]])]
tensor([[0.2673],
        [0.2673],
        [0.2673],
        [0.2673],
```

As seen above, the number of nodes in the graph is 14. 1/14 value is calculated for each node. Then the normalization process is completed by squaring this calculated value.

```
tab_sizes_e = [ graphs[i].number_of_edges() for i in range(len(graphs))] # nb of edges
print (tab_sizes_e)
tab_snorm_e = [ torch.FloatTensor(size,1).fill_(1./float(size)) for size in tab_sizes_e ]
print (tab_snorm_e)
snorm_e = torch.cat(tab_snorm_e).sqrt() # graph size normalization
print (snorm_e)

batched_graph = dgl.batch(graphs) # batch graphs
return batched_graph, labels, snorm_n, snorm_e
```

```
[56]
[tensor([[0.0179],
        [0.0179],
        [0.0179],
        [0.0179],
        [0.0179],
```

```
tensor([[0.1336],
        [0.1336],
        [0.1336],
        [0.1336],
        [0.1336],
        [0.1336],
```

Same operations applied for number of edges.

After that, created artifical data feature (in degree) for each node with help of DGLGraph and PyTorch libraries [4][5][6][7].

```
# create artifical data feature (= in degree) for each node
def create_artificial_features(dataset):
    for (graph,_) in dataset:
        graph.ndata['feat'] = graph.in_degrees().view(-1, 1).float()
        graph.edata['feat'] = torch.ones(graph.number_of_edges(),1)
    return dataset
```

After these phases, graphs will be generated.

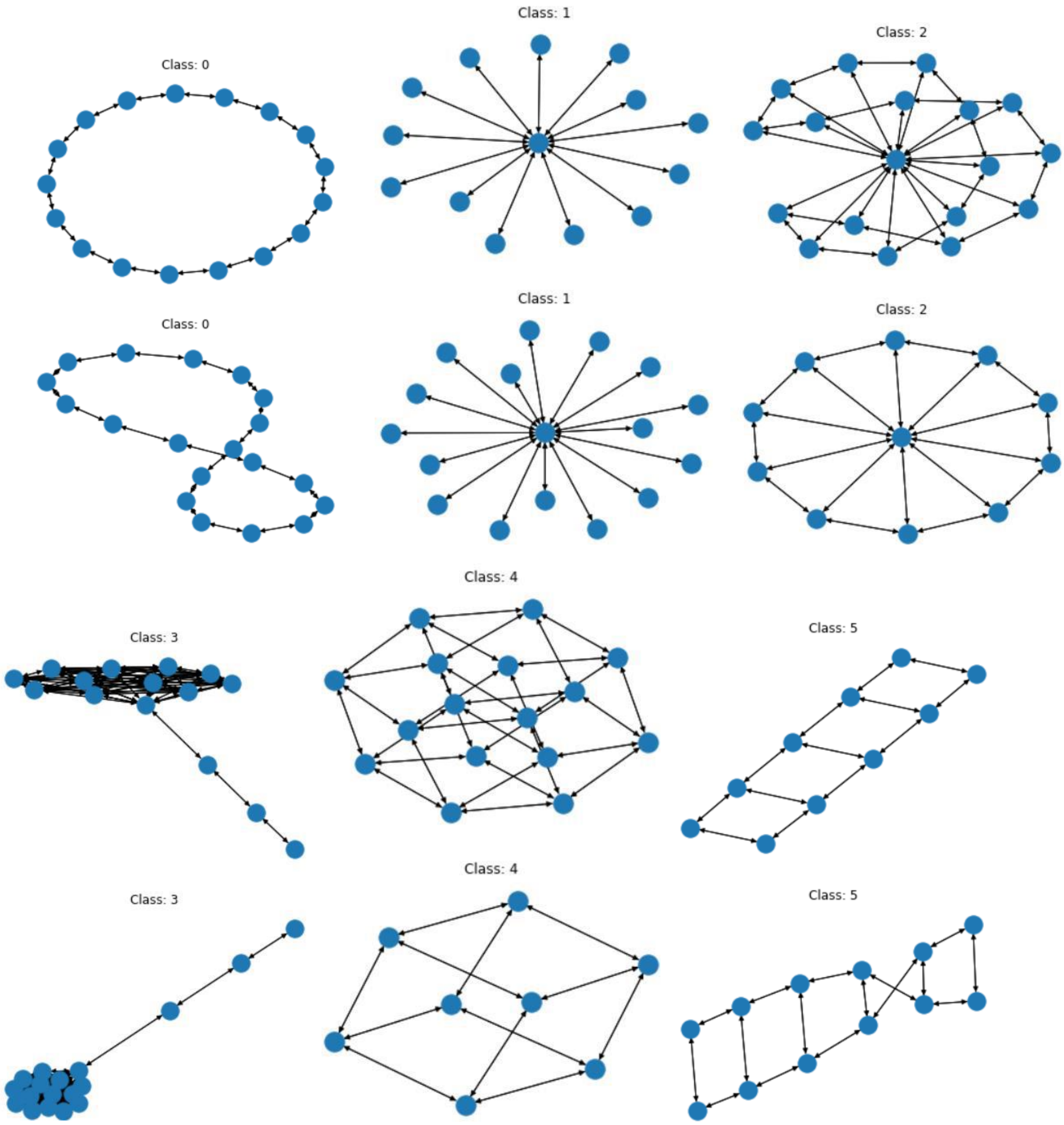
```
# generate artifical graph dataset with DGL
trainset = MiniGCDataset(50, 10, 20)
trainset = create_artificial_features(trainset)

for i in range(0,50):
    print(trainset[i])
```

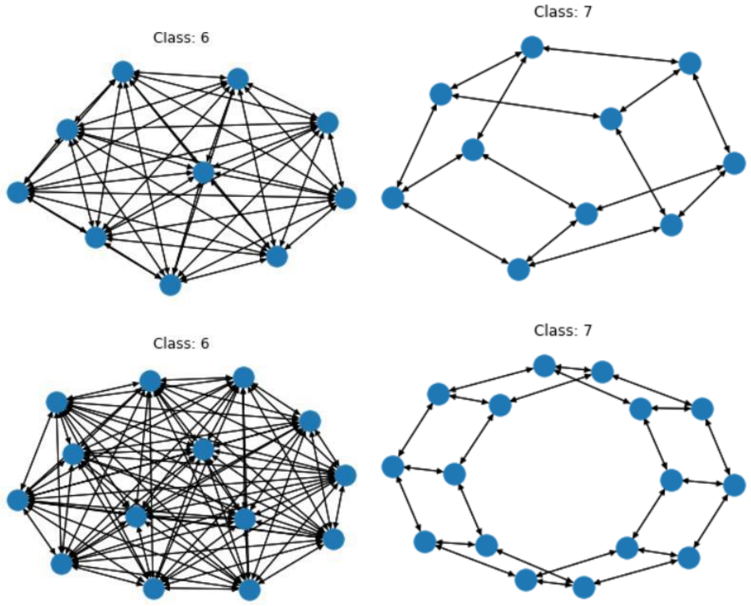
```
      ndata_schemes={'feat': Scheme(shape=(1,), dtype=torch.float32)}
      edata_schemes={'feat': Scheme(shape=(1,), dtype=torch.float32)}), tensor(6, dtype=torch.int32))
(Graph(num_nodes=16, num_edges=256,
      ndata_schemes={'feat': Scheme(shape=(1,), dtype=torch.float32)}
      edata_schemes={'feat': Scheme(shape=(1,), dtype=torch.float32)}), tensor(6, dtype=torch.int32))
(Graph(num_nodes=18, num_edges=324,
      ndata_schemes={'feat': Scheme(shape=(1,), dtype=torch.float32)}
      edata_schemes={'feat': Scheme(shape=(1,), dtype=torch.float32)}), tensor(6, dtype=torch.int32))
(Graph(num_nodes=14, num_edges=56,
      ndata_schemes={'feat': Scheme(shape=(1,), dtype=torch.float32)}
      edata_schemes={'feat': Scheme(shape=(1,), dtype=torch.float32)}), tensor(7, dtype=torch.int32))
(Graph(num_nodes=10, num_edges=40,
      ndata_schemes={'feat': Scheme(shape=(1,), dtype=torch.float32)}
      edata_schemes={'feat': Scheme(shape=(1,), dtype=torch.float32)}), tensor(7, dtype=torch.int32))
(Graph(num_nodes=14, num_edges=56,
      ndata_schemes={'feat': Scheme(shape=(1,), dtype=torch.float32)}
      edata_schemes={'feat': Scheme(shape=(1,), dtype=torch.float32)}), tensor(7, dtype=torch.int32))
(Graph(num_nodes=18, num_edges=72,
      ndata_schemes={'feat': Scheme(shape=(1,), dtype=torch.float32)}
```

After graphs are created, graphs created by graph visualization can be displayed with help of Networkx and Matplotlib libraries.

```
visualset = MiniGCDataset(50, 10, 20)
# visualise the 8 classes of graphs
for c in range(50):
    graph, label = visualset[c]
    fig, ax = plt.subplots()
    nx.draw(graph.to_networkx(), ax=ax)
    ax.set_title('Class: {:d}'.format(label))
    plt.show()
```

Some of the 50 graphs that is created:

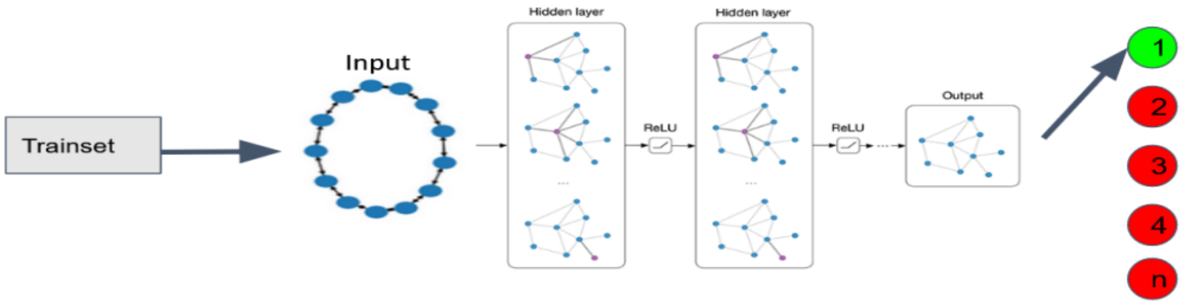





**Algorithm of the Project**

Mini graph classification dataset class is compatible with pytorch's Dataset class. That's why I couldn't use TensorFlow and scikit-learn libraries for classification algorithms. In the project, only PyTorch was used as the classification algorithm. Therefore, the project includes only the neural network model.

To put it simply, it can seperate the layers in our structure into 3 main layers. Input layer, hidden layers and output layer. A graph will be given to the input layers and the type of that graph will be displayed from the output layers. Since we have 8 different graph types, our output layer dimension will be 8.




In general, the structure of the project is as in the figure above.

**Implementing Model**

Firstly, created MLP_layer class for classification.

```
class MLP_layer(nn.Module):
    def __init__(self, input_dim, output_dim, L=2): # L = nb of hidden layers
        super(MLP_layer, self).__init__()
        list_FC_layers = [ nn.Linear( input_dim, input_dim, bias=True ) for l in range(L) ]
        list_FC_layers.append(nn.Linear( input_dim, output_dim , bias=True ))
        self.FC_layers = nn.ModuleList(list_FC_layers)
        self.L = L

    def forward(self, x):
        y = x
        for l in range(self.L):
            y = self.FC_layers[l](y)
            y = torch.relu(y)
        y = self.FC_layers[self.L](y)
        return y
```

It returns y value for classification.

After that, created GatedGCN_layer class for transfer functions.

It returns h and e values that is residual connection for transferring between neurons.

Neural Network model is created with the following neighborhood transfer functions:

$$h_i^{\ell+1} = h_i^{\ell} + \mathrm{ReLU}\left(A^{\ell} h_i^{\ell} + \sum_{j \sim i} \eta(e_{ij}^{\ell}) \odot B^{\ell} h_j^{\ell}\right)$$

$$\eta(e_{ij}^{\ell}) = \frac{\sigma(e_{ij}^{\ell})}{\sum_{j' \sim i} \sigma(e_{ij'}^{\ell}) + \varepsilon}$$

$$e_{ij}^{\ell+1} = e_{ij}^{\ell} + \mathrm{ReLU}\left(C^{\ell} e_{ij}^{\ell} + D^{\ell} h_i^{\ell+1} + E^{\ell} h_j^{\ell+1}\right)$$

Where l denotes the layer level, and ReLU is the rectified linear unit.

In GatedGCN_layer class, there is 3 main functions that provides transfer operation. These are message_func, reduce_func and forward.

In DGL, message functions are referred to as Edge User Defined Functions.

Edge UDFs take a single argument edge. It has three members src, dst, and data for accessing source node properties, target node properties, and edge properties.

Reduce functions are Node UDFs. Node UDFs have a single argument node with two member data and a mailbox. data contains node properties and mailbox contains all incoming message properties stacked along the second dimension (hence the dim=1 argument).

message_func and reduce_func send messages from all edges and update all nodes.

Finally, inputs are connected to graph convolutional neural network model and model is connected to MLP classifier in GatedGCN_Net class.

In this class, defined loss, accuracy and update functions for using in epochs.

Created neural network model has 1 input dimension, 100 hidden dimensions and 8 output dimensions. Number of hidden layers is 2.

```
GatedGCN_Net(
  (embedding_h): Linear(in_features=1, out_features=100, bias=True)
  (embedding_e): Linear(in_features=1, out_features=100, bias=True)
  (GatedGCN_layers): ModuleList(
    (0): GatedGCN_layer(
      (A): Linear(in_features=100, out_features=100, bias=True)
      (B): Linear(in_features=100, out_features=100, bias=True)
      (C): Linear(in_features=100, out_features=100, bias=True)
      (D): Linear(in_features=100, out_features=100, bias=True)
      (E): Linear(in_features=100, out_features=100, bias=True)
      (bn_node_h): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (bn_node_e): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): GatedGCN_layer(
      (A): Linear(in_features=100, out_features=100, bias=True)
      (B): Linear(in_features=100, out_features=100, bias=True)
      (C): Linear(in_features=100, out_features=100, bias=True)
      (D): Linear(in_features=100, out_features=100, bias=True)
      (E): Linear(in_features=100, out_features=100, bias=True)
      (bn_node_h): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (bn_node_e): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (MLP_layer): MLP_layer(
    (FC_layers): ModuleList(
      (0): Linear(in_features=100, out_features=100, bias=True)
      (1): Linear(in_features=100, out_features=100, bias=True)
      (2): Linear(in_features=100, out_features=8, bias=True)
    )
  )
)
```

After the creating model, test forward pass operation is defined with get first graph batch and

DataLoader.

Test backward pass operation is defined with 0.5 learning rate.

The learning rate controls how quickly the model is adapted to the problem. Smaller learning rates require more training epochs given the smaller changes made to the weights each update, whereas larger learning rates result in rapid changes and require fewer training epochs.

A learning rate that is too large can cause the model to converge too quickly to a suboptimal solution, whereas a learning rate that is too small can cause the process to get stuck [8].

There is a checking some sizes.

```
Graph(num_nodes=117, num_edges=513,
      ndata_schemes={'feat': Scheme(shape=(1,), dtype=torch.float32)}
      edata_schemes={'feat': Scheme(shape=(1,), dtype=torch.float32)})
tensor([5, 3, 4, 7, 5, 2, 7, 4, 7, 0])
torch.Size([117, 1])
torch.Size([513, 1])
batch_x torch.Size([117, 1])
batch_e torch.Size([513, 1])
batch_snorm_n torch.Size([117, 1])
batch_snorm_e torch.Size([513, 1])
torch.Size([10, 8])
```

After that, created train one epoch operation that is returning loss and accuracy values and created evaluation operation that is returning test loss and test  accuracy values.

Finally, neural network model is trained with train and test loader.

**Tests**

Our data set with 1000 graphs containing minimum 10 and maximum 20 nodes and training has 40 epoch steps.

**K-fold=10 Cross Validation Model Evaluation**

**Batch size = 200, hidden dimension = 100**

It takes 3 minutes and 45 seconds.

It has 0.9833 train accuracy, 0.97 test accuracy and 0.3299 train loss, 0.3368 test loss.

```
Epoch 23, time 5.5827, train_loss: 0.7109, test_loss: 0.6826
 train_acc: 0.8500, test_acc: 0.9100
Epoch 24, time 5.6518, train_loss: 0.7007, test_loss: 0.6426
 train_acc: 0.8789, test_acc: 0.9400
Epoch 25, time 5.5794, train_loss: 0.6323, test_loss: 0.6392
 train_acc: 0.8467, test_acc: 0.9400
Epoch 26, time 5.5303, train_loss: 0.6127, test_loss: 0.5861
 train_acc: 0.8622, test_acc: 0.9200
Epoch 27, time 5.5416, train_loss: 0.5848, test_loss: 0.5622
 train_acc: 0.8689, test_acc: 0.9700
Epoch 28, time 5.6428, train_loss: 0.5583, test_loss: 0.5385
 train_acc: 0.8878, test_acc: 0.9500
Epoch 29, time 5.6188, train_loss: 0.5446, test_loss: 0.5061
 train_acc: 0.8878, test_acc: 0.9700
Epoch 30, time 5.6170, train_loss: 0.5366, test_loss: 0.4939
 train_acc: 0.8933, test_acc: 0.9700
Epoch 31, time 5.6358, train_loss: 0.4899, test_loss: 0.5085
 train_acc: 0.8900, test_acc: 0.9400
Epoch 32, time 5.6501, train_loss: 0.4615, test_loss: 0.4373
 train_acc: 0.9100, test_acc: 0.9900
Epoch 33, time 5.6926, train_loss: 0.4554, test_loss: 0.4092
 train_acc: 0.9267, test_acc: 0.9800
Epoch 34, time 5.7207, train_loss: 0.4214, test_loss: 0.4007
 train_acc: 0.9311, test_acc: 0.9600
Epoch 35, time 5.5397, train_loss: 0.4013, test_loss: 0.3840
 train_acc: 0.9389, test_acc: 1.0000
Epoch 36, time 5.6255, train_loss: 0.3774, test_loss: 0.3569
 train_acc: 0.9556, test_acc: 0.9900
Epoch 37, time 5.5599, train_loss: 0.3909, test_loss: 0.3476
 train_acc: 0.9622, test_acc: 1.0000
Epoch 38, time 5.5542, train_loss: 0.3394, test_loss: 0.3376
 train_acc: 0.9833, test_acc: 1.0000
Epoch 39, time 5.6761, train_loss: 0.3299, test_loss: 0.3368
 train_acc: 0.9833, test_acc: 0.9700
```

**Batch size = 100, hidden dimension = 100**

It takes 3 minutes and 31 seconds.

It has 0.9967 train accuracy, 1.0 test accuracy and 0.1123 train loss, 0.1169 test loss.

```
Epoch 21, time 5.1968, train_loss: 0.4711, test_loss: 0.4424
 train_acc: 0.9311, test_acc: 0.9600
Epoch 22, time 5.2323, train_loss: 0.4288, test_loss: 0.4023
 train_acc: 0.9600, test_acc: 1.0000
Epoch 23, time 5.1914, train_loss: 0.3911, test_loss: 0.3785
 train_acc: 0.9700, test_acc: 0.9600
Epoch 24, time 5.3670, train_loss: 0.3740, test_loss: 0.3404
 train_acc: 0.9667, test_acc: 1.0000
Epoch 25, time 5.2899, train_loss: 0.3568, test_loss: 0.3205
 train_acc: 0.9700, test_acc: 1.0000
Epoch 26, time 5.2931, train_loss: 0.3075, test_loss: 0.2866
 train_acc: 0.9878, test_acc: 1.0000
Epoch 27, time 5.3559, train_loss: 0.2915, test_loss: 0.2626
 train_acc: 0.9811, test_acc: 1.0000
Epoch 28, time 5.4552, train_loss: 0.2623, test_loss: 0.2356
 train_acc: 0.9911, test_acc: 1.0000
Epoch 29, time 5.4220, train_loss: 0.2382, test_loss: 0.2308
 train_acc: 0.9889, test_acc: 1.0000
Epoch 30, time 5.3670, train_loss: 0.2307, test_loss: 0.1981
 train_acc: 0.9900, test_acc: 1.0000
Epoch 31, time 5.2625, train_loss: 0.2106, test_loss: 0.1975
 train_acc: 0.9933, test_acc: 1.0000
Epoch 32, time 5.2777, train_loss: 0.1812, test_loss: 0.1823
 train_acc: 1.0000, test_acc: 1.0000
Epoch 33, time 5.2934, train_loss: 0.1959, test_loss: 0.1798
 train_acc: 0.9700, test_acc: 1.0000
Epoch 34, time 5.3281, train_loss: 0.1765, test_loss: 0.1567
 train_acc: 0.9933, test_acc: 1.0000
Epoch 35, time 5.3249, train_loss: 0.1516, test_loss: 0.1433
 train_acc: 1.0000, test_acc: 1.0000
Epoch 36, time 5.3000, train_loss: 0.1450, test_loss: 0.1797
 train_acc: 0.9944, test_acc: 0.9600
Epoch 37, time 5.2053, train_loss: 0.1303, test_loss: 0.2014
 train_acc: 0.9967, test_acc: 1.0000
Epoch 38, time 5.2210, train_loss: 0.1471, test_loss: 0.1334
 train_acc: 0.9756, test_acc: 1.0000
Epoch 39, time 5.2392, train_loss: 0.1123, test_loss: 0.1169
 train_acc: 0.9967, test_acc: 1.0000
```

**Batch size = 200, hidden dimension = 200**

It takes 7 minutes and 54 seconds.

It has 1.0 train accuracy, 1.0 test accuracy and 0.0502 train loss, 0.0511 test loss.

```
Epoch 19, time 11.9055, train_loss: 0.3512, test_loss: 0.3569
 train_acc: 0.9422, test_acc: 0.9300
Epoch 20, time 11.6874, train_loss: 0.2998, test_loss: 0.3433
 train_acc: 0.9867, test_acc: 0.9100
Epoch 21, time 11.6950, train_loss: 0.2748, test_loss: 0.3466
 train_acc: 0.9822, test_acc: 0.9300
Epoch 22, time 11.8956, train_loss: 0.2764, test_loss: 0.3191
 train_acc: 0.9567, test_acc: 0.9800
Epoch 23, time 11.9035, train_loss: 0.2468, test_loss: 0.2587
 train_acc: 0.9689, test_acc: 0.9600
Epoch 24, time 12.0463, train_loss: 0.2243, test_loss: 0.2421
 train_acc: 0.9944, test_acc: 0.9300
Epoch 25, time 12.0922, train_loss: 0.2064, test_loss: 0.2245
 train_acc: 0.9800, test_acc: 0.9800
Epoch 26, time 11.8209, train_loss: 0.1818, test_loss: 0.1824
 train_acc: 0.9978, test_acc: 1.0000
Epoch 27, time 11.8926, train_loss: 0.1812, test_loss: 0.1871
 train_acc: 0.9922, test_acc: 1.0000
Epoch 28, time 11.7772, train_loss: 0.1597, test_loss: 0.1659
 train_acc: 0.9922, test_acc: 1.0000
Epoch 29, time 11.6940, train_loss: 0.1408, test_loss: 0.1495
 train_acc: 0.9967, test_acc: 1.0000
Epoch 30, time 11.9045, train_loss: 0.1456, test_loss: 0.1353
 train_acc: 0.9922, test_acc: 1.0000
Epoch 31, time 11.8629, train_loss: 0.1134, test_loss: 0.1241
 train_acc: 1.0000, test_acc: 1.0000
Epoch 32, time 11.8975, train_loss: 0.1176, test_loss: 0.1701
 train_acc: 0.9911, test_acc: 0.9800
Epoch 33, time 11.7606, train_loss: 0.1036, test_loss: 0.1155
 train_acc: 1.0000, test_acc: 1.0000
Epoch 34, time 11.7592, train_loss: 0.1046, test_loss: 0.1309
 train_acc: 0.9989, test_acc: 1.0000
Epoch 35, time 11.8602, train_loss: 0.0958, test_loss: 0.1022
 train_acc: 1.0000, test_acc: 1.0000
Epoch 36, time 11.8667, train_loss: 0.0722, test_loss: 0.1408
 train_acc: 1.0000, test_acc: 0.9600
Epoch 37, time 11.8477, train_loss: 0.0846, test_loss: 0.0823
 train_acc: 0.9978, test_acc: 1.0000
Epoch 38, time 11.8595, train_loss: 0.0772, test_loss: 0.0594
 train_acc: 1.0000, test_acc: 1.0000
Epoch 39, time 11.9995, train_loss: 0.0502, test_loss: 0.0511
 train_acc: 1.0000, test_acc: 1.0000
```

**Batch size = 100, hidden dimension = 200**

It takes 3 minutes and 35 seconds.

It has 1.0 train accuracy, 1.0 test accuracy and 0.0952 train loss, 0.094 test loss.

```
Epoch 19, time 5.3675, train_loss: 0.5232, test_loss: 0.5133
 train_acc: 0.9611, test_acc: 1.0000
Epoch 20, time 5.3976, train_loss: 0.5056, test_loss: 0.5139
 train_acc: 0.9311, test_acc: 0.9400
Epoch 21, time 5.3400, train_loss: 0.4557, test_loss: 0.4304
 train_acc: 0.9744, test_acc: 1.0000
Epoch 22, time 5.3731, train_loss: 0.4259, test_loss: 0.4454
 train_acc: 0.9689, test_acc: 1.0000
Epoch 23, time 5.2951, train_loss: 0.3923, test_loss: 0.3705
 train_acc: 0.9689, test_acc: 1.0000
Epoch 24, time 5.4261, train_loss: 0.3937, test_loss: 0.3817
 train_acc: 0.9622, test_acc: 0.9900
Epoch 25, time 5.4707, train_loss: 0.3585, test_loss: 0.3611
 train_acc: 0.9767, test_acc: 0.9900
Epoch 26, time 5.3355, train_loss: 0.3161, test_loss: 0.3065
 train_acc: 0.9811, test_acc: 1.0000
Epoch 27, time 5.3689, train_loss: 0.2975, test_loss: 0.2828
 train_acc: 0.9856, test_acc: 0.9600
Epoch 28, time 5.4126, train_loss: 0.2440, test_loss: 0.2508
 train_acc: 0.9989, test_acc: 1.0000
Epoch 29, time 5.2774, train_loss: 0.2327, test_loss: 0.2182
 train_acc: 0.9989, test_acc: 1.0000
Epoch 30, time 5.2592, train_loss: 0.2121, test_loss: 0.2070
 train_acc: 0.9933, test_acc: 0.9900
Epoch 31, time 5.3289, train_loss: 0.1891, test_loss: 0.1850
 train_acc: 0.9956, test_acc: 0.9900
Epoch 32, time 5.3407, train_loss: 0.1731, test_loss: 0.1655
 train_acc: 1.0000, test_acc: 1.0000
Epoch 33, time 5.3312, train_loss: 0.1610, test_loss: 0.1369
 train_acc: 0.9956, test_acc: 1.0000
Epoch 34, time 5.2954, train_loss: 0.1885, test_loss: 0.1489
 train_acc: 0.9867, test_acc: 1.0000
Epoch 35, time 5.3414, train_loss: 0.1315, test_loss: 0.1224
 train_acc: 1.0000, test_acc: 1.0000
Epoch 36, time 5.4250, train_loss: 0.1223, test_loss: 0.1292
 train_acc: 0.9989, test_acc: 1.0000
Epoch 37, time 5.8327, train_loss: 0.1161, test_loss: 0.1748
 train_acc: 1.0000, test_acc: 0.9200
Epoch 38, time 5.6261, train_loss: 0.1325, test_loss: 0.1342
 train_acc: 0.9900, test_acc: 0.9700
Epoch 39, time 5.4874, train_loss: 0.0952, test_loss: 0.0940
 train_acc: 1.0000, test_acc: 1.0000
```

**10% Test and 90% Train Split Model Evaluation**

**Batch size = 200, hidden dimension = 100**

It takes 3 minutes and 53 seconds.

It has 0.9678 train accuracy, 1.0 test accuracy and 0.4394 train loss, 0.4893 test loss.

```
Epoch 19, time 5.7990, train_loss: 1.1181, test_loss: 1.1583, val_loss: 1.1583
 train_acc: 0.8044, test_acc: 0.7800, val_acc: 0.7800
Epoch 20, time 5.8785, train_loss: 1.0948, test_loss: 1.1115, val_loss: 1.1115
 train_acc: 0.7867, test_acc: 0.7800, val_acc: 0.7800
Epoch 21, time 5.8592, train_loss: 1.0392, test_loss: 1.0687, val_loss: 1.0687
 train_acc: 0.8311, test_acc: 0.8700, val_acc: 0.8700
Epoch 22, time 5.8566, train_loss: 0.9994, test_loss: 1.0239, val_loss: 1.0239
 train_acc: 0.8300, test_acc: 0.8800, val_acc: 0.8800
Epoch 23, time 5.7432, train_loss: 0.9464, test_loss: 0.9840, val_loss: 0.9840
 train_acc: 0.8344, test_acc: 0.8200, val_acc: 0.8200
Epoch 24, time 5.8324, train_loss: 0.9016, test_loss: 0.9394, val_loss: 0.9394
 train_acc: 0.8422, test_acc: 0.8200, val_acc: 0.8200
Epoch 25, time 5.8432, train_loss: 0.8577, test_loss: 0.9134, val_loss: 0.9134
 train_acc: 0.8522, test_acc: 0.8200, val_acc: 0.8200
Epoch 26, time 5.9124, train_loss: 0.8340, test_loss: 0.8802, val_loss: 0.8802
 train_acc: 0.8556, test_acc: 0.8200, val_acc: 0.8200
Epoch 27, time 5.8352, train_loss: 0.7874, test_loss: 0.8440, val_loss: 0.8440
 train_acc: 0.8578, test_acc: 0.8200, val_acc: 0.8200
Epoch 28, time 5.8166, train_loss: 0.7849, test_loss: 0.8172, val_loss: 0.8172
 train_acc: 0.8456, test_acc: 0.8200, val_acc: 0.8200
Epoch 29, time 5.6768, train_loss: 0.7400, test_loss: 0.7809, val_loss: 0.7809
 train_acc: 0.8644, test_acc: 0.7900, val_acc: 0.7900
Epoch 30, time 5.7670, train_loss: 0.6915, test_loss: 0.7285, val_loss: 0.7285
 train_acc: 0.8789, test_acc: 0.8800, val_acc: 0.8800
Epoch 31, time 5.7619, train_loss: 0.6527, test_loss: 0.7001, val_loss: 0.7001
 train_acc: 0.9011, test_acc: 0.9300, val_acc: 0.9300
Epoch 32, time 5.8061, train_loss: 0.6111, test_loss: 0.6589, val_loss: 0.6589
 train_acc: 0.9067, test_acc: 0.9600, val_acc: 0.9600
Epoch 33, time 5.7393, train_loss: 0.6309, test_loss: 0.6367, val_loss: 0.6367
 train_acc: 0.8944, test_acc: 0.9600, val_acc: 0.9600
Epoch 34, time 5.8276, train_loss: 0.5863, test_loss: 0.6054, val_loss: 0.6054
 train_acc: 0.9178, test_acc: 0.9500, val_acc: 0.9500
Epoch 35, time 5.7801, train_loss: 0.5445, test_loss: 0.5803, val_loss: 0.5803
 train_acc: 0.9589, test_acc: 0.9900, val_acc: 0.9900
Epoch 36, time 5.8041, train_loss: 0.5262, test_loss: 0.5447, val_loss: 0.5447
 train_acc: 0.9644, test_acc: 0.9800, val_acc: 0.9800
Epoch 37, time 5.8686, train_loss: 0.5168, test_loss: 0.5141, val_loss: 0.5141
 train_acc: 0.9356, test_acc: 0.9900, val_acc: 0.9900
Epoch 38, time 5.9353, train_loss: 0.4914, test_loss: 0.4870, val_loss: 0.4870
 train_acc: 0.9233, test_acc: 0.9900, val_acc: 0.9900
Epoch 39, time 5.8923, train_loss: 0.4394, test_loss: 0.4893, val_loss: 0.4893
 train_acc: 0.9678, test_acc: 1.0000, val_acc: 1.0000
```

**Batch size = 100, hidden dimension = 100**

It takes 3 minutes and 40 seconds.

It has 0.9933 train accuracy, 1.0 test accuracy and 0.1181 train loss, 0.1112 test loss.

```
Epoch 19, time 5.5059, train_loss: 0.4605, test_loss: 0.4495, val_loss: 0.4495
 train_acc: 0.9044, test_acc: 0.9700, val_acc: 0.9700
Epoch 20, time 5.5132, train_loss: 0.4043, test_loss: 0.4431, val_loss: 0.4431
 train_acc: 0.9511, test_acc: 0.9300, val_acc: 0.9300
Epoch 21, time 5.4619, train_loss: 0.4139, test_loss: 0.4192, val_loss: 0.4192
 train_acc: 0.9344, test_acc: 0.9700, val_acc: 0.9700
Epoch 22, time 5.5519, train_loss: 0.3486, test_loss: 0.3628, val_loss: 0.3628
 train_acc: 0.9589, test_acc: 0.9700, val_acc: 0.9700
Epoch 23, time 5.4964, train_loss: 0.3252, test_loss: 0.3354, val_loss: 0.3354
 train_acc: 0.9700, test_acc: 1.0000, val_acc: 1.0000
Epoch 24, time 5.5494, train_loss: 0.2898, test_loss: 0.3191, val_loss: 0.3191
 train_acc: 0.9900, test_acc: 1.0000, val_acc: 1.0000
Epoch 25, time 5.5479, train_loss: 0.2977, test_loss: 0.3112, val_loss: 0.3112
 train_acc: 0.9644, test_acc: 0.9600, val_acc: 0.9600
Epoch 26, time 5.5194, train_loss: 0.2683, test_loss: 0.2693, val_loss: 0.2693
 train_acc: 0.9633, test_acc: 1.0000, val_acc: 1.0000
Epoch 27, time 5.5303, train_loss: 0.3048, test_loss: 0.2605, val_loss: 0.2605
 train_acc: 0.9356, test_acc: 0.9900, val_acc: 0.9900
Epoch 28, time 5.5893, train_loss: 0.2680, test_loss: 0.2404, val_loss: 0.2404
 train_acc: 0.9622, test_acc: 1.0000, val_acc: 1.0000
Epoch 29, time 5.5103, train_loss: 0.2275, test_loss: 0.2226, val_loss: 0.2226
 train_acc: 0.9889, test_acc: 1.0000, val_acc: 1.0000
Epoch 30, time 5.4478, train_loss: 0.2352, test_loss: 0.2492, val_loss: 0.2492
 train_acc: 0.9633, test_acc: 0.9700, val_acc: 0.9700
Epoch 31, time 5.5294, train_loss: 0.2041, test_loss: 0.2116, val_loss: 0.2116
 train_acc: 0.9933, test_acc: 1.0000, val_acc: 1.0000
Epoch 32, time 5.5673, train_loss: 0.1873, test_loss: 0.1864, val_loss: 0.1864
 train_acc: 0.9900, test_acc: 1.0000, val_acc: 1.0000
Epoch 33, time 5.6104, train_loss: 0.1643, test_loss: 0.1826, val_loss: 0.1826
 train_acc: 0.9933, test_acc: 1.0000, val_acc: 1.0000
Epoch 34, time 5.5455, train_loss: 0.1607, test_loss: 0.1624, val_loss: 0.1624
 train_acc: 0.9878, test_acc: 1.0000, val_acc: 1.0000
Epoch 35, time 5.6537, train_loss: 0.1526, test_loss: 0.1599, val_loss: 0.1599
 train_acc: 0.9933, test_acc: 1.0000, val_acc: 1.0000
Epoch 36, time 5.6051, train_loss: 0.1563, test_loss: 0.1949, val_loss: 0.1949
 train_acc: 0.9944, test_acc: 0.9700, val_acc: 0.9700
Epoch 37, time 5.5813, train_loss: 0.1370, test_loss: 0.1396, val_loss: 0.1396
 train_acc: 1.0000, test_acc: 1.0000, val_acc: 1.0000
Epoch 38, time 5.4631, train_loss: 0.1239, test_loss: 0.1541, val_loss: 0.1541
 train_acc: 0.9967, test_acc: 1.0000, val_acc: 1.0000
Epoch 39, time 5.5487, train_loss: 0.1181, test_loss: 0.1112, val_loss: 0.1112
 train_acc: 0.9933, test_acc: 1.0000, val_acc: 1.0000
```

**Batch size = 200, hidden dimension = 200**

It takes 8 minutes and 11 seconds.

It has 1.0 train accuracy, 1.0 test accuracy and 0.0667 train loss, 0.0831 test loss.

```
Epoch 19, time 12.1184, train_loss: 0.4236, test_loss: 0.4830, val_loss: 0.4830
 train_acc: 0.9478, test_acc: 0.8500, val_acc: 0.8500
Epoch 20, time 12.2387, train_loss: 0.3887, test_loss: 0.4590, val_loss: 0.4590
 train_acc: 0.9822, test_acc: 0.8600, val_acc: 0.8600
Epoch 21, time 12.2194, train_loss: 0.3676, test_loss: 0.4623, val_loss: 0.4623
 train_acc: 0.9656, test_acc: 0.8400, val_acc: 0.8400
Epoch 22, time 12.2651, train_loss: 0.3343, test_loss: 0.3950, val_loss: 0.3950
 train_acc: 0.9411, test_acc: 0.9500, val_acc: 0.9500
Epoch 23, time 12.2301, train_loss: 0.3009, test_loss: 0.3471, val_loss: 0.3471
 train_acc: 0.9633, test_acc: 0.9700, val_acc: 0.9700
Epoch 24, time 12.3535, train_loss: 0.2848, test_loss: 0.3872, val_loss: 0.3872
 train_acc: 0.9767, test_acc: 0.8600, val_acc: 0.8600
Epoch 25, time 12.3268, train_loss: 0.2598, test_loss: 0.3170, val_loss: 0.3170
 train_acc: 0.9733, test_acc: 0.9300, val_acc: 0.9300
Epoch 26, time 12.3470, train_loss: 0.2900, test_loss: 0.3396, val_loss: 0.3396
 train_acc: 0.9589, test_acc: 0.9400, val_acc: 0.9400
Epoch 27, time 12.2756, train_loss: 0.2622, test_loss: 0.3180, val_loss: 0.3180
 train_acc: 0.9556, test_acc: 0.9700, val_acc: 0.9700
Epoch 28, time 12.2807, train_loss: 0.2139, test_loss: 0.2567, val_loss: 0.2567
 train_acc: 0.9978, test_acc: 1.0000, val_acc: 1.0000
Epoch 29, time 12.1640, train_loss: 0.2096, test_loss: 0.2609, val_loss: 0.2609
 train_acc: 0.9900, test_acc: 0.9700, val_acc: 0.9700
Epoch 30, time 12.2868, train_loss: 0.1847, test_loss: 0.2308, val_loss: 0.2308
 train_acc: 0.9756, test_acc: 0.9800, val_acc: 0.9800
Epoch 31, time 12.2490, train_loss: 0.1606, test_loss: 0.1833, val_loss: 0.1833
 train_acc: 0.9844, test_acc: 1.0000, val_acc: 1.0000
Epoch 32, time 12.3648, train_loss: 0.1490, test_loss: 0.1707, val_loss: 0.1707
 train_acc: 0.9911, test_acc: 1.0000, val_acc: 1.0000
Epoch 33, time 12.1059, train_loss: 0.1334, test_loss: 0.1608, val_loss: 0.1608
 train_acc: 0.9922, test_acc: 1.0000, val_acc: 1.0000
Epoch 34, time 12.2151, train_loss: 0.1074, test_loss: 0.1471, val_loss: 0.1471
 train_acc: 1.0000, test_acc: 1.0000, val_acc: 1.0000
Epoch 35, time 12.1998, train_loss: 0.0982, test_loss: 0.1375, val_loss: 0.1375
 train_acc: 1.0000, test_acc: 1.0000, val_acc: 1.0000
Epoch 36, time 12.1501, train_loss: 0.0898, test_loss: 0.1330, val_loss: 0.1330
 train_acc: 1.0000, test_acc: 1.0000, val_acc: 1.0000
Epoch 37, time 12.2534, train_loss: 0.0793, test_loss: 0.1023, val_loss: 0.1023
 train_acc: 1.0000, test_acc: 1.0000, val_acc: 1.0000
Epoch 38, time 12.3543, train_loss: 0.0686, test_loss: 0.0807, val_loss: 0.0807
 train_acc: 1.0000, test_acc: 1.0000, val_acc: 1.0000
Epoch 39, time 12.4217, train_loss: 0.0667, test_loss: 0.0831, val_loss: 0.0831
 train_acc: 1.0000, test_acc: 1.0000, val_acc: 1.0000
```

**Batch size = 100, hidden dimension = 200**

It takes 7 minutes and 56 seconds.

It has 0.9989 train accuracy, 0.99 test accuracy and 0.0165 train loss, 0.0381 test loss.

```
Epoch 19, time 11.7751, train_loss: 0.2004, test_loss: 0.1644, val_loss: 0.1644
 train_acc: 0.9833, test_acc: 1.0000, val_acc: 1.0000
Epoch 20, time 11.8098, train_loss: 0.1421, test_loss: 0.1807, val_loss: 0.1807
 train_acc: 0.9944, test_acc: 0.9900, val_acc: 0.9900
Epoch 21, time 11.8530, train_loss: 0.1236, test_loss: 0.1831, val_loss: 0.1831
 train_acc: 0.9967, test_acc: 1.0000, val_acc: 1.0000
Epoch 22, time 11.6647, train_loss: 0.1098, test_loss: 0.1913, val_loss: 0.1913
 train_acc: 0.9956, test_acc: 0.9300, val_acc: 0.9300
Epoch 23, time 11.7868, train_loss: 0.0849, test_loss: 0.0920, val_loss: 0.0920
 train_acc: 0.9989, test_acc: 1.0000, val_acc: 1.0000
Epoch 24, time 11.8422, train_loss: 0.0840, test_loss: 0.0897, val_loss: 0.0897
 train_acc: 0.9944, test_acc: 1.0000, val_acc: 1.0000
Epoch 25, time 11.8395, train_loss: 0.0781, test_loss: 0.1003, val_loss: 0.1003
 train_acc: 0.9967, test_acc: 1.0000, val_acc: 1.0000
Epoch 26, time 12.0509, train_loss: 0.0718, test_loss: 0.0981, val_loss: 0.0981
 train_acc: 0.9933, test_acc: 0.9800, val_acc: 0.9800
Epoch 27, time 11.7404, train_loss: 0.0664, test_loss: 0.1207, val_loss: 0.1207
 train_acc: 0.9989, test_acc: 1.0000, val_acc: 1.0000
Epoch 28, time 11.8822, train_loss: 0.0546, test_loss: 0.0636, val_loss: 0.0636
 train_acc: 0.9989, test_acc: 1.0000, val_acc: 1.0000
Epoch 29, time 11.7624, train_loss: 0.0607, test_loss: 0.1651, val_loss: 0.1651
 train_acc: 0.9956, test_acc: 0.9200, val_acc: 0.9200
Epoch 30, time 11.9360, train_loss: 0.0515, test_loss: 0.0715, val_loss: 0.0715
 train_acc: 1.0000, test_acc: 1.0000, val_acc: 1.0000
Epoch 31, time 11.9147, train_loss: 0.0404, test_loss: 0.0530, val_loss: 0.0530
 train_acc: 1.0000, test_acc: 1.0000, val_acc: 1.0000
Epoch 32, time 12.0755, train_loss: 0.0365, test_loss: 0.0405, val_loss: 0.0405
 train_acc: 1.0000, test_acc: 1.0000, val_acc: 1.0000
Epoch 33, time 11.9859, train_loss: 0.0304, test_loss: 0.0435, val_loss: 0.0435
 train_acc: 1.0000, test_acc: 1.0000, val_acc: 1.0000
Epoch 34, time 12.0540, train_loss: 0.0368, test_loss: 0.2634, val_loss: 0.2634
 train_acc: 0.9989, test_acc: 0.8400, val_acc: 0.8400
Epoch 35, time 12.1454, train_loss: 0.0289, test_loss: 0.0471, val_loss: 0.0471
 train_acc: 1.0000, test_acc: 0.9900, val_acc: 0.9900
Epoch 36, time 12.1043, train_loss: 0.0211, test_loss: 0.0421, val_loss: 0.0421
 train_acc: 1.0000, test_acc: 1.0000, val_acc: 1.0000
Epoch 37, time 11.8390, train_loss: 0.0180, test_loss: 0.0217, val_loss: 0.0217
 train_acc: 1.0000, test_acc: 1.0000, val_acc: 1.0000
Epoch 38, time 12.1111, train_loss: 0.0184, test_loss: 0.0985, val_loss: 0.0985
 train_acc: 1.0000, test_acc: 0.9500, val_acc: 0.9500
Epoch 39, time 11.9861, train_loss: 0.0165, test_loss: 0.0381, val_loss: 0.0381
 train_acc: 0.9989, test_acc: 0.9900, val_acc: 0.9900
```

**Results**

If we want to compare k-fold cross validation with 90% train and 10% test split method, we should compare for equal number of batch size and hidden dimension.

For example, let's do it for our first test case, batch size=200 and hidden dimension=100.

**In k-fold cross validation method:**

It takes 3 minutes and 45 seconds.

It has 0.9833 train accuracy, 0.97 test accuracy and 0.3299 train loss, 0.3368 test loss.

**In with 90% train and 10% test split method:**

It takes 3 minutes and 53 seconds.

It has 0.9678 train accuracy, 1.0 test accuracy and 0.4394 train loss, 0.4893 test loss.

**K-fold cross validation method is more successful with time and accuracy.**

To compare the batch size and hidden dimension hyperparameters, let's examine the results in the k-fold cross validation method.

First, let's examine the batch size parameter. For this, let's choose 2 test cases with equal hidden dimensions.

**batch size=200 and hidden dimension=100:**

It takes 3 minutes and 45 seconds.

It has 0.9833 train accuracy, 0.97 test accuracy and 0.3299 train loss, 0.3368 test loss.

**batch size=100 and hidden dimension=100:**

It takes 3 minutes and 31 seconds.

It has 0.9967 train accuracy, 1.0 test accuracy and 0.1123 train loss, 0.1169 test loss.

Decreasing the batch size value decreased the time value, increased the accuracy value, and decreased the loss value.

**For this model, it may be better to keep the batch size low.**

After, let's examine the hidden dimension parameter. For this, let's choose 2 test cases with equal batch sizes.

**batch size=100 and hidden dimension=100:**

It takes 3 minutes and 31 seconds.

It has 0.9967 train accuracy, 1.0 test accuracy and 0.1123 train loss, 0.1169 test loss.

**batch size=100 and hidden dimension=200:**

It takes 3 minutes and 35 seconds.

It has 1.0 train accuracy, 1.0 test accuracy and 0.0952 train loss, 0.094 test loss.

Decreasing the hidden dimension value decreased the time value (very less), decreased the accuracy value, and increased the loss value.

**For this model, it will be better to keep the hidden dimension high.**

**As a result of all test case evaluations, keeping the batch size low and the hidden dimension high in the k-fold cross validation method is the best option for this model.**

**(k-fold cross validation, batch size=100, hidden dimension=200)**

## References

[1] dgl.data.minigc — DGL 0.6.1 documentation. (2021). Retrieved 12 May 2021, from https://docs.dgl.ai/en/0.6.x/_modules/dgl/data/minigc.html

[2] dgl.DGLGraph — DGL 0.4.3post2 documentation. (2021). Retrieved 12 May 2021, from https://docs.dgl.ai/en/0.4.x/api/python/graph.html

[3] torch.cat — PyTorch 1.8.1 documentation. (2021). Retrieved 12 May 2021, from https://pytorch.org/docs/stable/generated/torch.cat.html

[4] torch.ones — PyTorch 1.8.1 documentation. (2021). Retrieved 12 May 2021, from https://pytorch.org/docs/stable/generated/torch.ones.html

[5] dgl.DGLGraph.ndata — DGL 0.6.1 documentation. (2021). Retrieved 12 May 2021, from https://docs.dgl.ai/en/0.6.x/generated/dgl.DGLGraph.ndata.html

[6] PyTorch?, H., Ahmad, W., & Armas, J. (2021). How does the "view" method work in PyTorch?. Retrieved 12 May 2021, from https://stackoverflow.com/questions/42479902/how-does-the-view-method-work-in-pytorch

[7] dgl.DGLGraph.in_degrees — DGL 0.6.1 documentation. (2021). Retrieved 12 May 2021, from https://docs.dgl.ai/en/0.6.x/generated/dgl.DGLGraph.in_degrees.html

[8] Brownlee, J. (2021). Understand the Impact of Learning Rate on Neural Network Performance. Retrieved 15 June 2021, from https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/