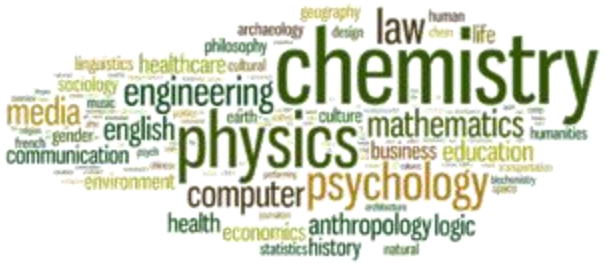
Mining & Learning on Graphs

Node Classification

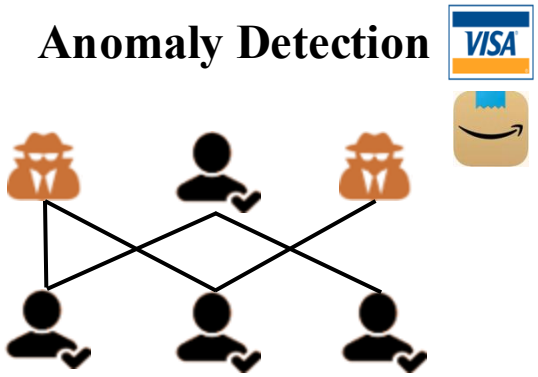
Yu Wang, Ph.D.
Assistant Professor
Computer and Information Science
University of Oregon
CS 410/510 - Fall 2024

Node Classification - Motivation

Paper Management



Anomaly Detection



Node Classification

- Problem:
  - Given a network where some nodes have labels, how do we assign labels to other nodes in the network?

Labeled node v_i with label y_i

Unlabeled node

Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{X}) = (\mathbf{A}, \mathbf{X})$

Adjacency Matrix $\mathbf{A}$

Node Feature Matrix $\mathbf{X}$

Labeled Data $\mathcal{D}_L = (\mathcal{V}_L, \mathcal{Y}_L)$

Unlabeled Data $\mathcal{D}_U = (\mathcal{V}_U)$

Node Classification – How would traditional classification work?

Data

Model

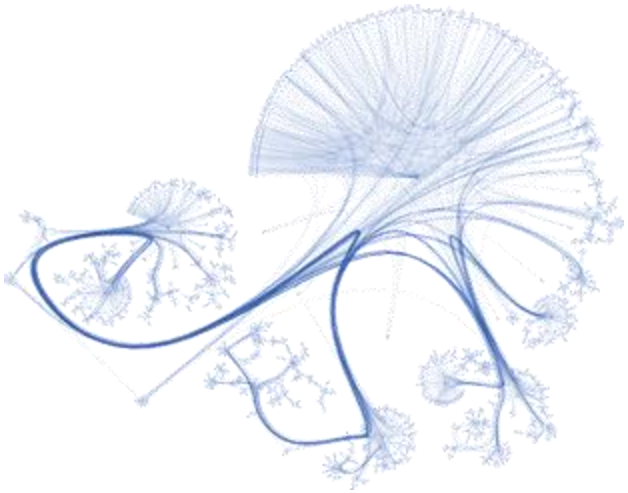
Loss

Optimization

Node Classification – How would traditional classification work?

Data	Model	Loss	Optimization

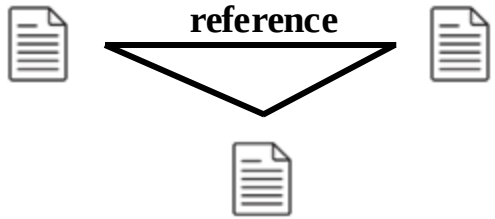
Cora - Paper Citation Networks



1 INTRODUCTION

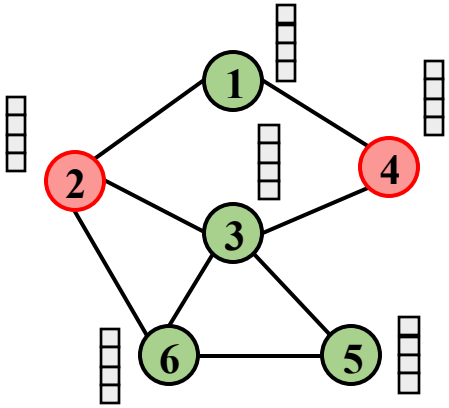
We consider the problem of classifying nodes (such as documents) in a graph (such as a citation network), where labels are only available for a small subset of nodes. This problem can be framed as graph-based semi-supervised learning, where label information is smoothed over the graph via some form of explicit graph-based regularization (Zhu et al., 2003; Zhou et al., 2004; Belkin et al., 2006; Weston et al., 2012), e.g. by using a graph Laplacian regularization term in the loss function:

$$\mathcal{L} = \mathcal{L}_0 + \lambda \mathcal{L}_{\text{reg}}, \quad \text{with} \quad \mathcal{L}_{\text{reg}} = \sum_{i,j} A_{ij} \| f(X_i) - f(X_j) \|^2 = f(X)^\top \Delta f(X). \tag{1}$$



Node Classification – How would traditional classification work?

Data | **Model** | **Loss** | **Optimization**




```
from torch_geometric.datasets import Planetoid
```

Dataset

```
dataset = Planetoid(root='/tmp/Cora', name='Cora')
```

```
dataset[0]
```

```
Data(x=[2708, 1433], edge_index=[2, 10556], y=[2708], train_mask=[2708], val_mask=[2708], test_mask=[2708])
```

Node Classification – How would traditional classification work?

Data | **Model** | **Loss** | **Optimization**

Dataset

```
dataset = Planetoid(root='/tmp/Cora', name='Cora')
```

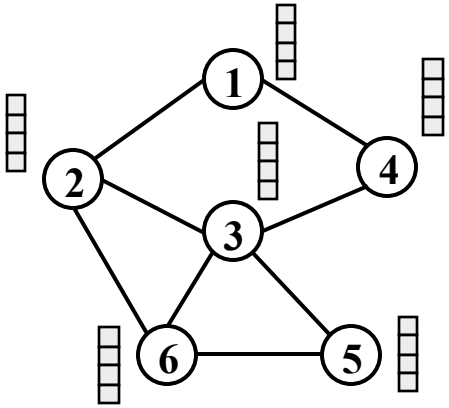
```
dataset[0]
```

```
Data(x=[2708, 1433], edge_index=[2, 10556], y=[2708], train_mask=[2708], val_mask=[2708], test_mask=[2708])
```

Node Feature

```
dataset[0].x
```

```
tensor([[0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.]])
```




```
data.raw_text[0]
```

```
' Stochastic pro-positionalization of non-determinate background knowledge. : It is a well-known fact that propositional learning algorithms require "good" features to perform well in pra
ctice. So a major step in data engineering for inductive learning is the construction of good features by domain experts. These features often represent properties of structured objects,
where a property typically is the occurrence of a certain substructure having certain properties. To partly automate the process of "feature engineering", we devised an algorithm that sea
rches for features which are defined by such substructures. The algorithm stochastically conducts a top-down search for first-order clauses, where each clause represents a binary feature.
It differs from existing algorithms in that its search is not class-blind, and that it is capable of considering clauses ("context") of almost arbitrary length (size). Preliminary experim
ents are favorable, and support the view that this approach is promising.'
```

Node Classification – How would traditional classification work?

Data | **Model** | **Loss** | **Optimization**

Dataset

```
dataset = Planetoid(root='/tmp/Cora', name='Cora')
```

```
dataset[0]
```

```
Data(x=[2708, 1433], edge_index=[2, 10556], y=[2708], train_mask=[2708], val_mask=[2708], test_mask=[2708])
```

Node label

```
dataset[0].y
```

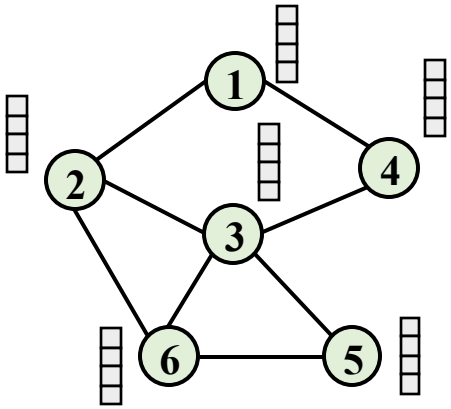
```
tensor([3, 4, 4,  ..., 3, 3, 3])
```

```
dataset[0].y.unique()
```

```
tensor([0, 1, 2, 3, 4, 5, 6])
```

```
set(data.category_names)
```

```
{'Case_Based',
 'Genetic_Algorithms',
 'Neural_Networks',
 'Probabilistic_Methods',
 'Reinforcement_Learning',
 'Rule_Learning',
 'Theory'}
```



Node Classification – How would traditional classification work?

Data | **Model** | **Loss** | **Optimization**

Dataset

```
dataset = Planetoid(root='/tmp/Cora', name='Cora')
```

```
dataset[0]
```

```
Data(x=[2708, 1433], edge_index=[2, 10556], y=[2708], train_mask=[2708], val_mask=[2708], test_mask=[2708])
```

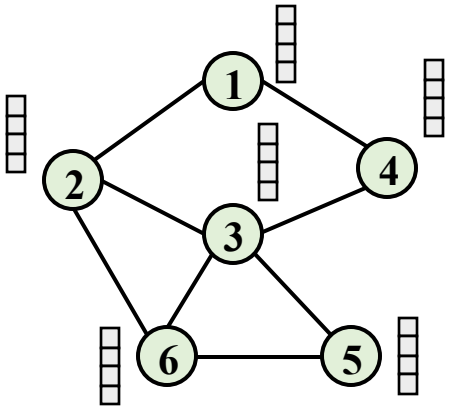
Edge Index

```
dataset[0].edge_index
```

```
tensor([[ 633, 1862, 2582,  ...,  598, 1473, 2706],
        [   0,    0,    0,  ..., 2707, 2707, 2707]])
```

```
print(is_undirected(dataset[0].edge_index))
```

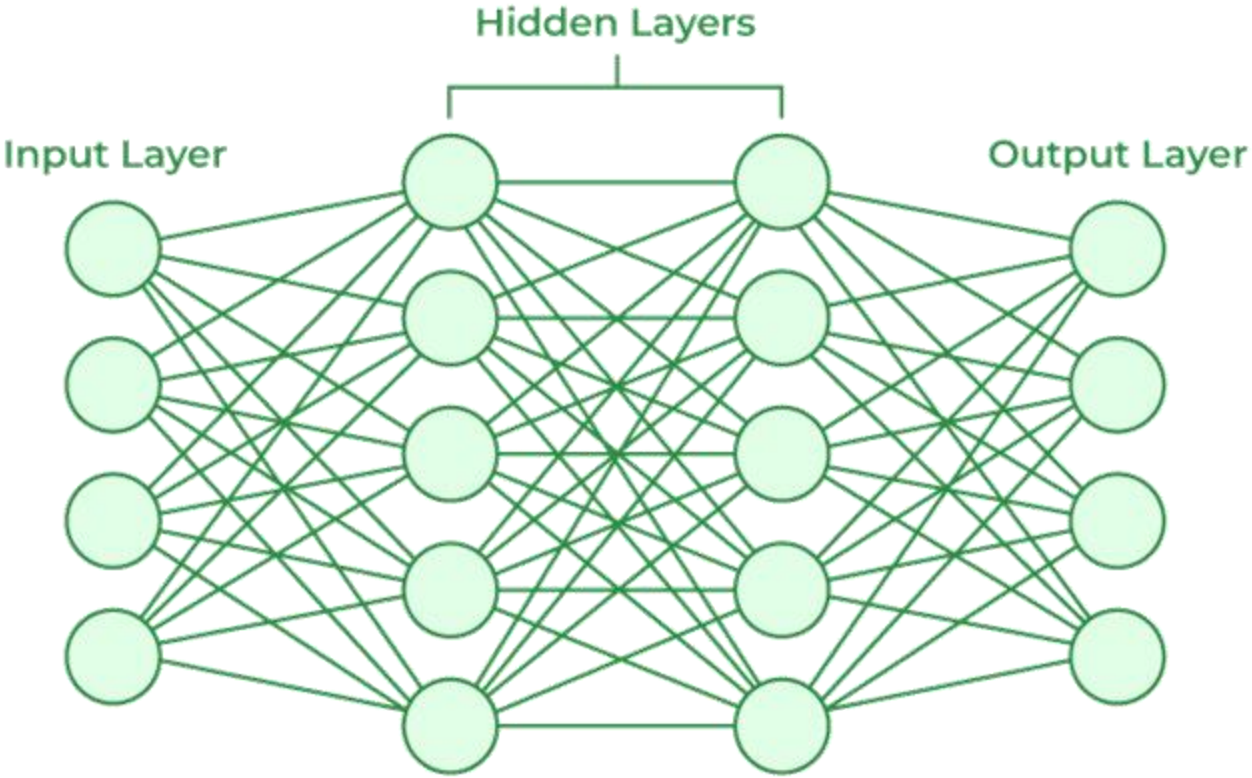
```
True
```



Node Classification – How would traditional classification work?

Data	**Model**	**Loss**	**Optimization**

Paper



Computer System

Linear Regression

- Data - $\{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$
- Regression – Find f that minimizes our uncertainty about y given x

$$y = f(x) + n$$

- Minimizing Mean Squared Error = Minimizing Negative Log-Likelihood

$$\underset{f}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} \left(y^{(i)} - f(x^{(i)})\right)^2$$

Linear Regression

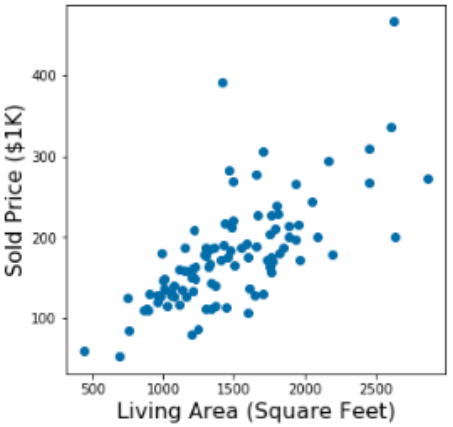
$$\underset{w}{\operatorname{argmin}} \frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)} - w^T\hat{x}^{(i)}\right)^2 = \underset{w}{\operatorname{argmin}} \frac{1}{N}\|y - Xw\|^2$$

Loss/Cost Function

Where

- $y = \left[y^{(1)}, \ldots, y^{(N)}\right]^T \in \mathbb{R}^{N\times 1}$ and
- $X = \left[\hat{x}^{(1)}, \ldots, \hat{x}^{(N)}\right]^T \in \mathbb{R}^{N\times(d+1)}$ (here $d = 1$)
- $w = [w_0, w_1, \ldots, w_d]^{\mathrm{T}} \in \mathbb{R}^{d+1}$

$$\underset{f}{\operatorname{argmin}} \frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)} - f\left(x^{(i)}\right)\right)^2$$



Nonlinear Regression

- So far, we have been using a linear function for regression:

$$f(x) = w^T x + w_0 = \sum_{i=0}^{d} w_i x_i \quad \text{(Assuming } x_0 = 1\text{)}$$

- Lets generalize this model:

$$f(x) = \sum_{i=0}^{M} w_i \phi_i(x) = w^T \phi(x)$$

where ϕ_i are fixed "basis" functions.

- For linear regression $M = d, \phi_i(x) = x_i$.

Nonlinear Regression

$$f(x) = \sum_{i=0}^{M} w_i \phi_i(x) = w^T \phi(x)$$

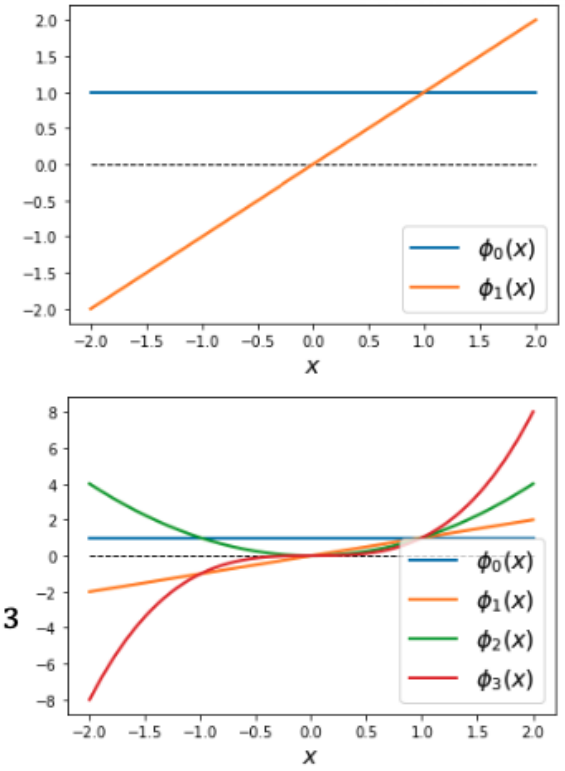
E.g., Polynomial Regression:

- 1D Polynomial Regression, $\phi(x) = [1, x, x^2, x^3]$:

$$\underset{w}{\operatorname{argmin}} \frac{1}{N} \sum_{n=1}^{N} \left(w^T \phi\left(x^{(i)}\right) - y^{(i)}\right)^2$$

To avoid confusion, note that: $\phi\left(x^{(i)}\right) = \left[1, x^{(i)}, (x^{(i)})^2, (x^{(i)})^3\right]$

$$\mathrm{f}\left(x^{(i)}\right) = w_0 + w_1 x^{(i)} + w_2 \left(x^{(i)}\right)^2 + w_3 \left(x^{(i)}\right)^3$$



Nonlinear Regression

Loss:

$$\underset{w}{\operatorname{argmin}} \frac{1}{N} \sum_{n=1}^{N} \left(w^T \phi\left(x^{(i)}\right) - y^{(i)} \right)^2 = \underset{w}{\operatorname{argmin}} \|\Phi w - y\|^2$$

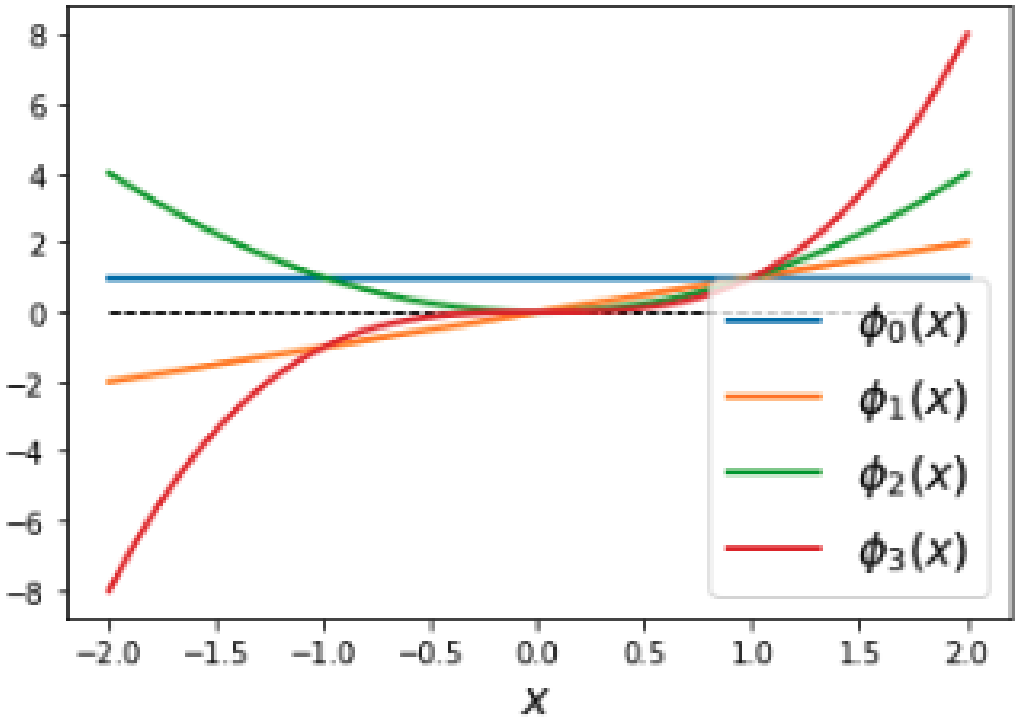
Where $\Phi = \left[\phi\left(x^{(1)}\right), \ldots, \phi\left(x^{(N)}\right)\right]^T \in \mathbb{R}^{N \times M}$ and $w \in \mathbb{R}^M$.

Optimization:

1. Closed form solution: $w^* = (\Phi^T \Phi)^{-1} \Phi^T y$
2. Gradient descent: $w^{(t)} = w^{(t-1)} - \epsilon \nabla_w \, Loss(w^{(t-1)})$

What is the problem of Nonlinear Regression?

Node Classification – How would traditional classification work?



Lower order polynomial cannot approximate higher order polynomial

The basis function is all fixed!

Can we learn the basis function?

Node Classification – How would traditional classification work?

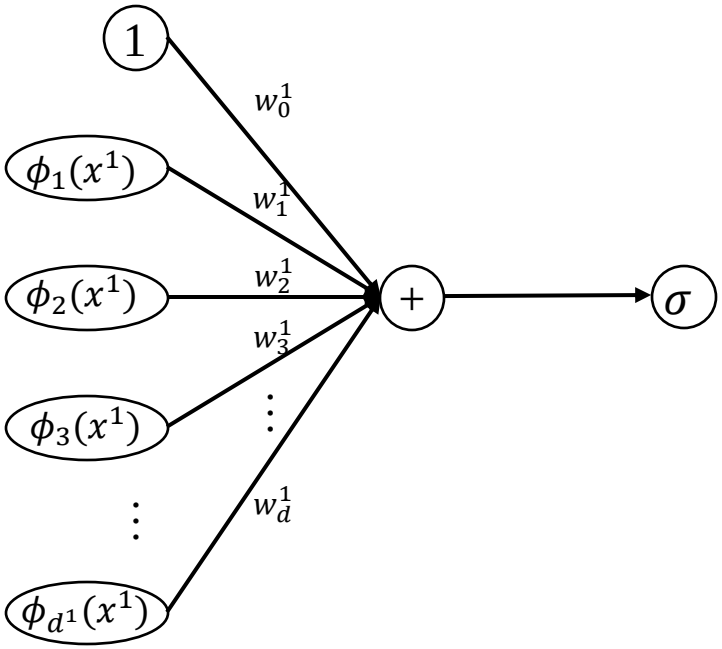
- Lets first look at what the learning problem might look like:

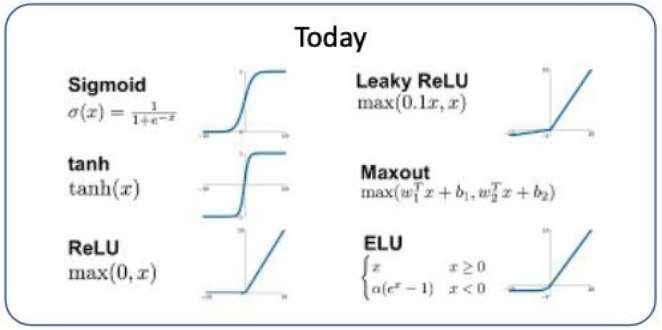
$$\underset{w}{\operatorname{argmin}} \sum_i \left(\left(\sum_j w_j \phi_j\left(x^{(i)}\right) \right) - y^{(i)} \right)^2$$

Neural Networks do this for us!

What things are learned here?
What things are fixed here?

1-layer Multi-layer Perceptron





$$\sum_{j=0}^{d^1} w_j^1 \phi_j(x^1)$$

Node Classification – How would traditional classification work?

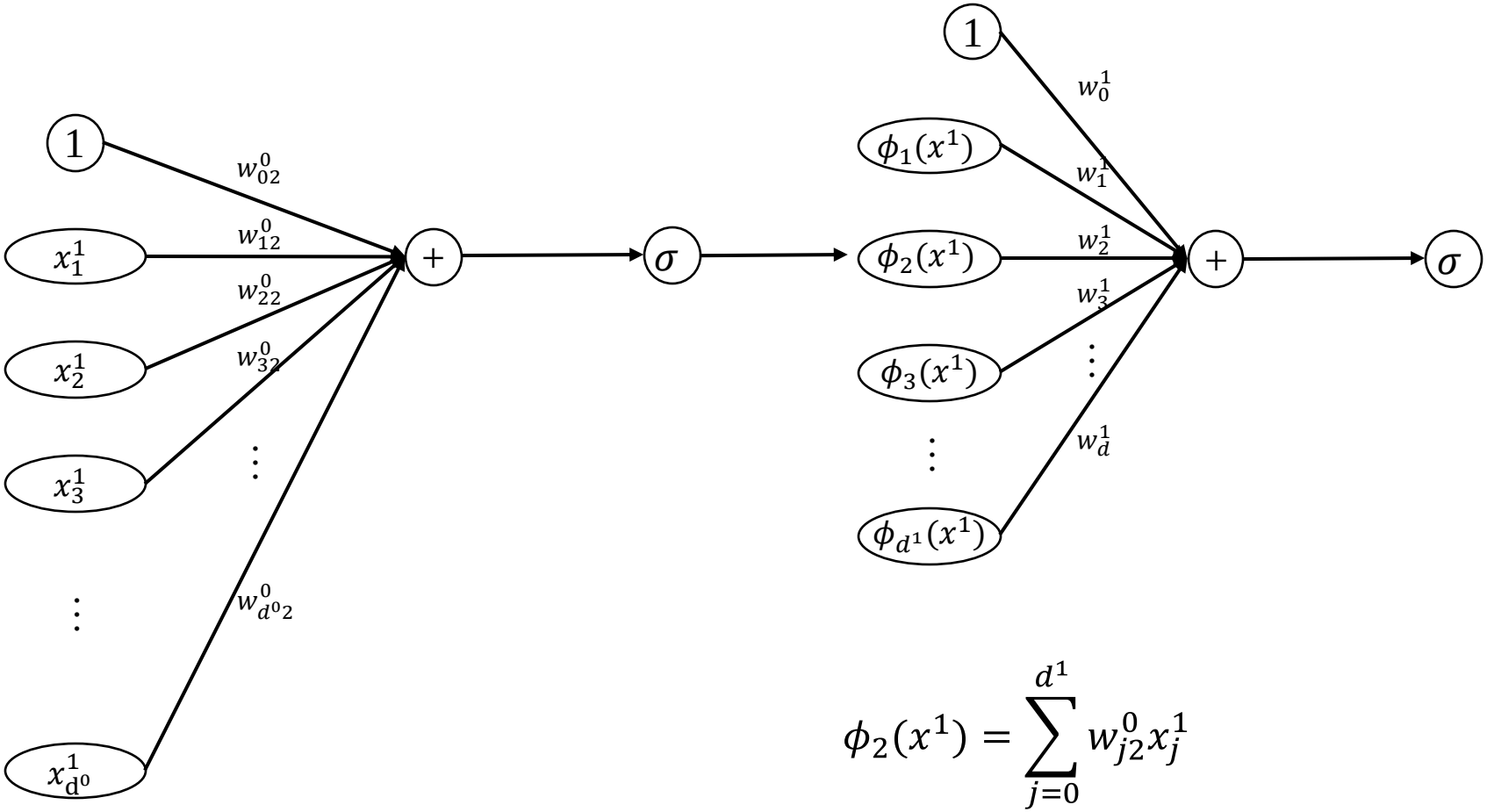
- Lets first look at what the learning problem might look like:

$$\underset{w,\{\phi_j\}_{j=1}^{M}}{\operatorname{argmin}} \sum_i \left(\left(\sum_j w_j \phi_j\left(x^{(i)}\right) \right) - y^{(i)} \right)^2$$

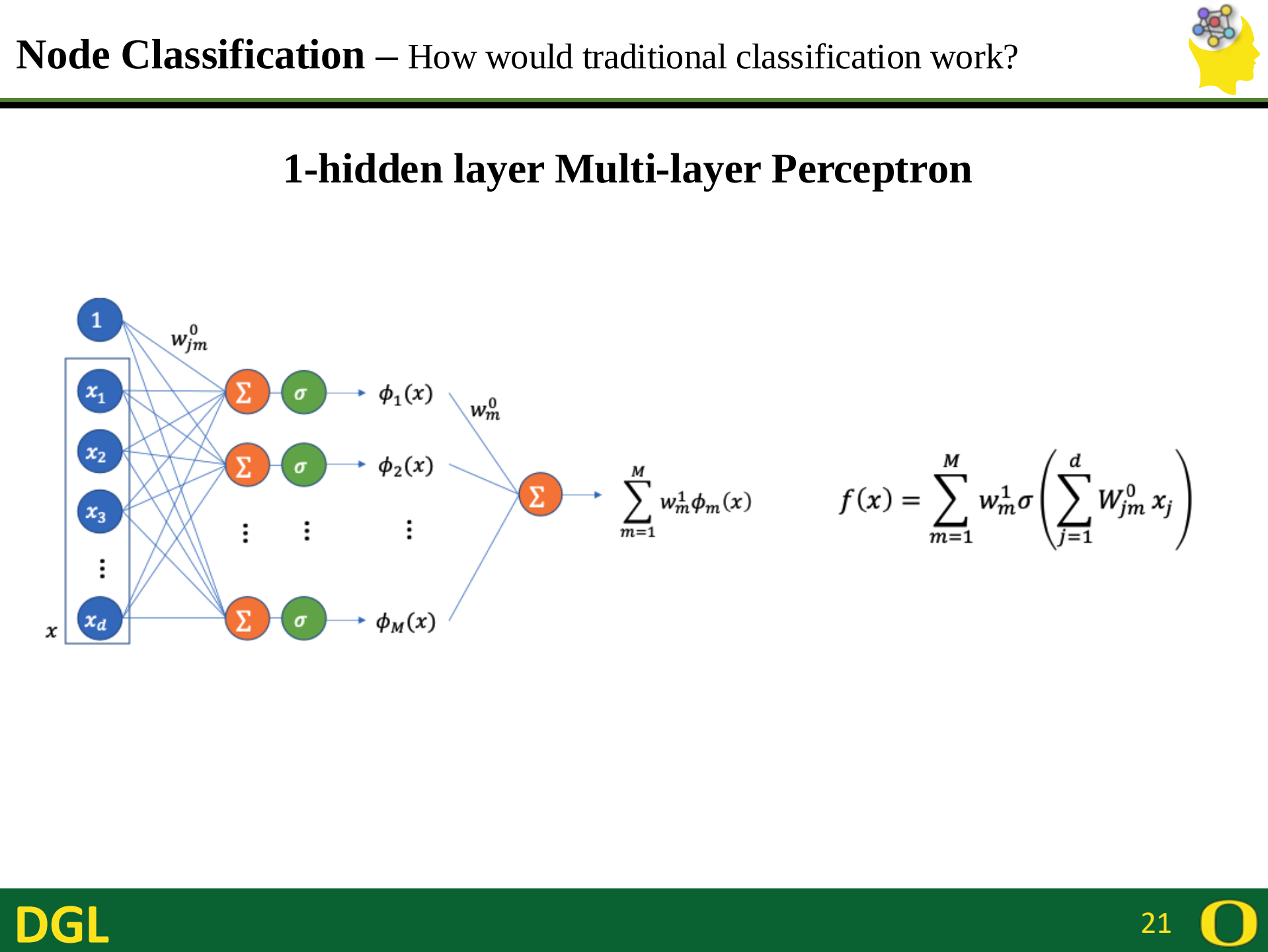
Neural Networks do this for us!

What things are learned here?
What things are fixed here?

1-hidden layer Multi-layer Perceptron



$$\phi_2(x^1) = \sum_{j=0}^{d^1} w_{j2}^0 x_j^1$$



1-hidden layer Multi-layer Perceptron

$$f(x) = \sum_{m=1}^{M} w_m^1 \sigma\left(\sum_{j=1}^{d} W_{jm}^0 x_j\right)$$

1-hidden layer Multi-layer Perceptron

```
class OneHiddenLayerMLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(OneHiddenLayerMLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)

        return x
```

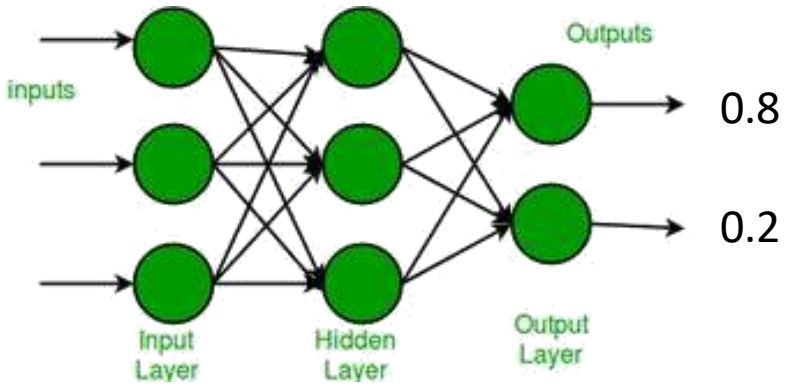
What is the input size? What is the hidden size? What is the output size?

Node Classification – How would traditional classification work?

Data | **Model** | **Loss** | **Optimization**

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-4)
```

```
loss = F.nll_loss(out[data.train_mask], data.y[data.train_mask])
```



0.8

0.2

$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$

Optimize the probably of the class corresponding to ground-truth!

Node Classification – How would traditional classification work?

Data | **Model** | **Loss** | **Optimization**

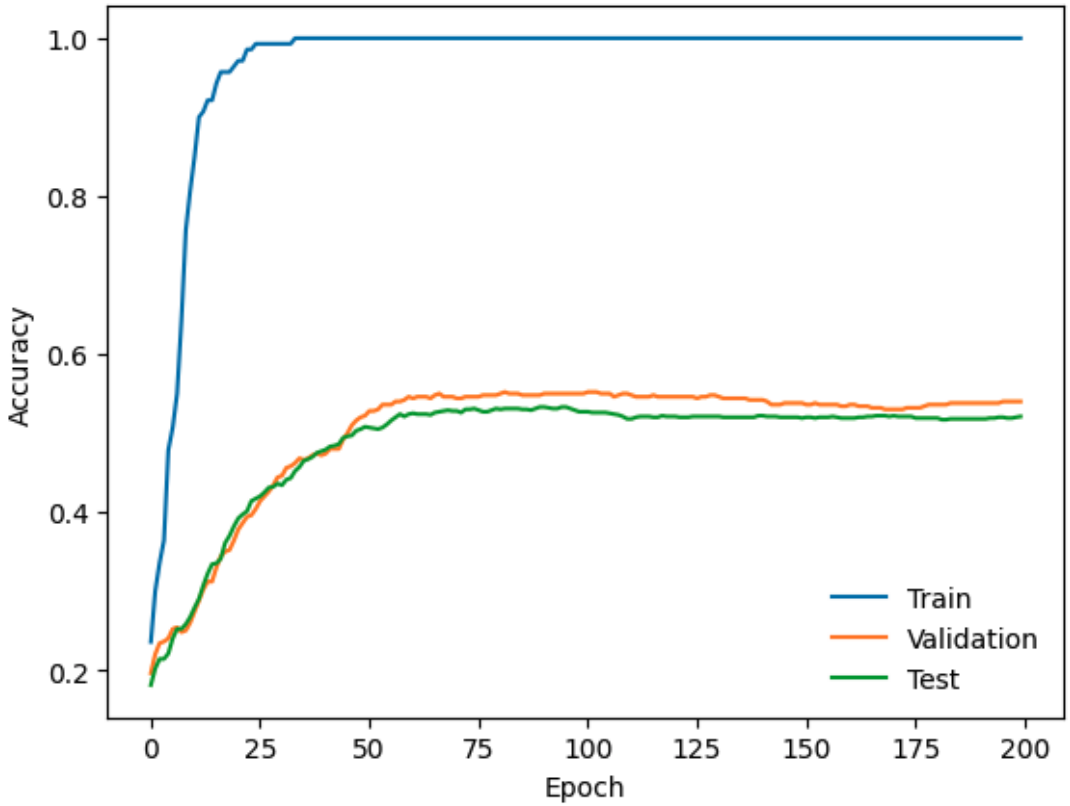
```python
class OneHiddenLayerMLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(OneHiddenLayerMLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)

        return F.log_softmax(x, dim=1)

# Example usage:
input_size = dataset[0].x.shape[1]
hidden_size = 64
output_size = torch.unique(dataset[0].y).shape[0]

MLP = OneHiddenLayerMLP(input_size, hidden_size, output_size)
optimizer = torch.optim.Adam(MLP.parameters(), lr=0.001, weight_decay=5e-4)
print(MLP)
```

```
OneHiddenLayerMLP(
  (fc1): Linear(in_features=1433, out_features=64, bias=True)
  (relu): ReLU()
  (fc2): Linear(in_features=64, out_features=7, bias=True)
)
```

```python
loss_train = []
train_accs, val_accs, test_accs = [], [], []
best_val_acc = 0

for epoch in range(200):
    model.train()
    optimizer.zero_grad()
    out = MLP(data.x)
    loss = F.nll_loss(out[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()
    loss_train.append(loss.item())

    with torch.no_grad():
        model.eval()
        pred = MLP(data.x).argmax(dim=1)
        train_acc, val_acc, test_acc = cal_acc(pred, data.y, data.train_mask), \
        cal_acc(pred, data.y, data.val_mask), cal_acc(pred, data.y, data.test_mask)

        if val_acc > best_val_acc:
            best_val_acc = val_acc
            final_test_acc = test_acc

    train_accs.append(train_acc)
    val_accs.append(val_acc)
    test_accs.append(test_acc)
```

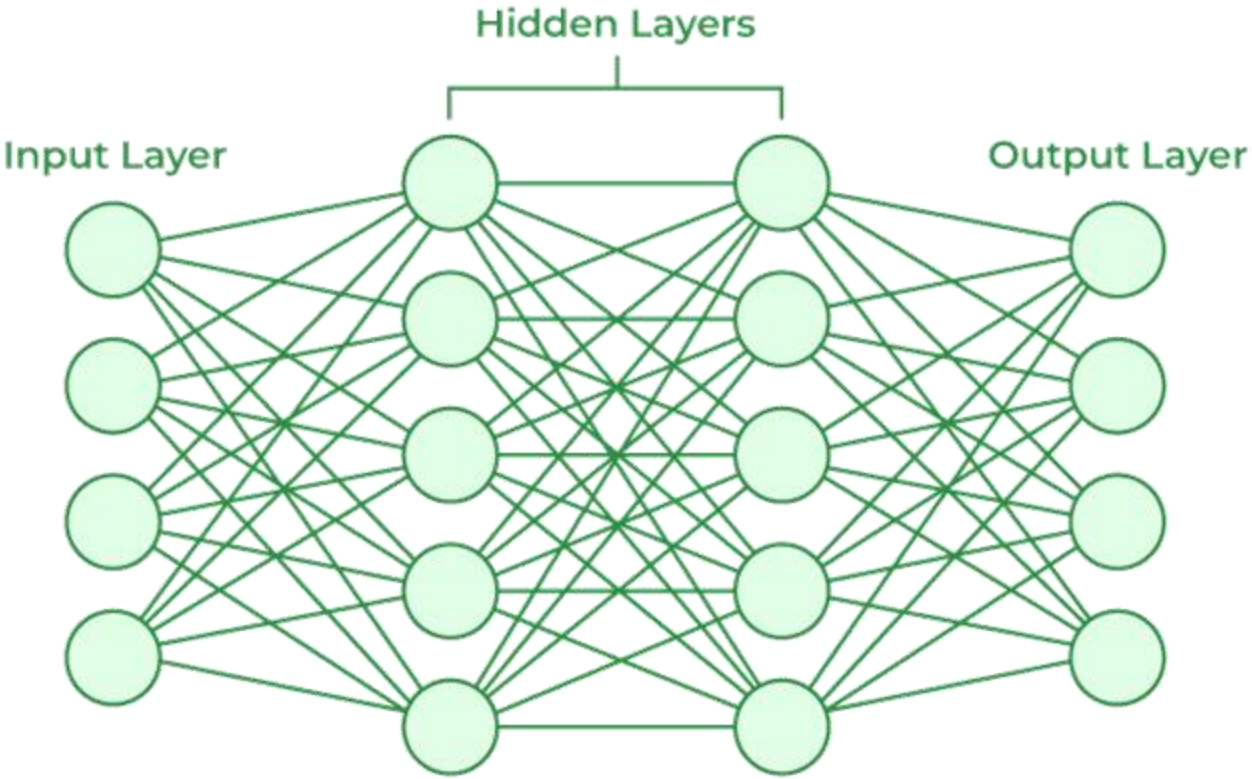
Node Classification – How would traditional classification work?

Data | **Model** | **Loss** | **Optimization**



Final Test Performance:
0.531

Node Classification



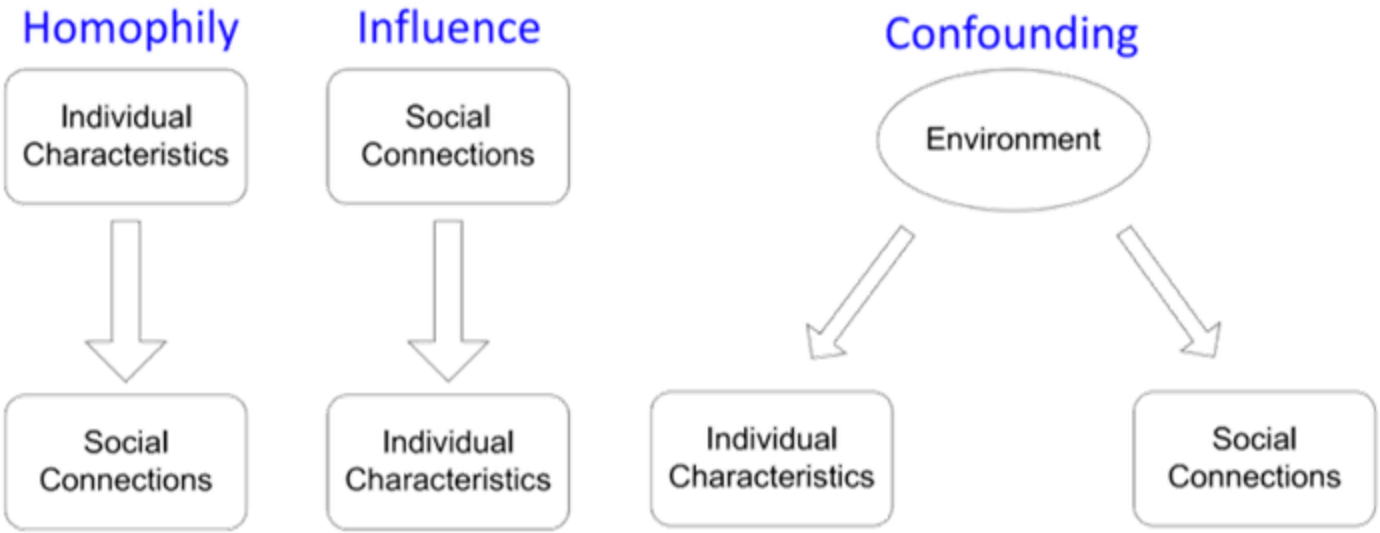
Paper

Computer System

We do not use any network information!

Correlation in networks

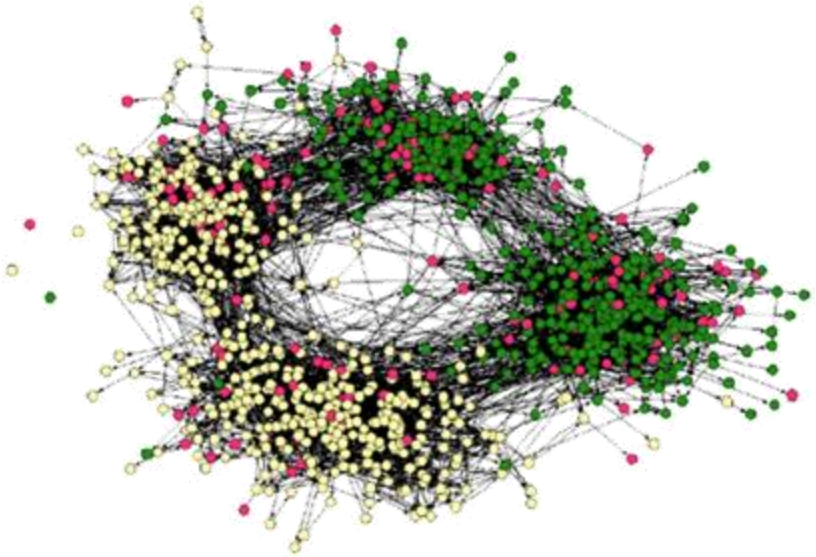
- User behaviors are correlated in social networks

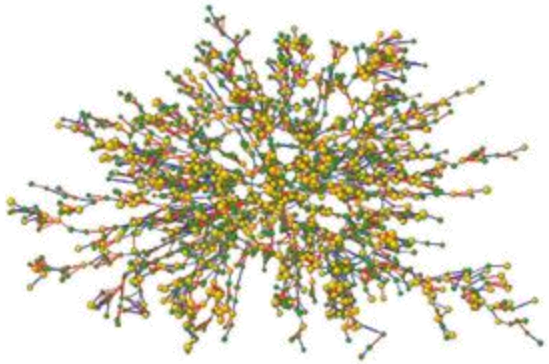


Node Classification - Motivation

Example

- Easley and Kleinberg, 2010.
- Friendships among people in a real social network where node color represents the race of the people/nodes.
- Homophily…





- Christakis and Fowler, 2007.
- Friendships among people in a real social network where node color represents the level of obesity of the people/nodes.
- Influence… and homophily?

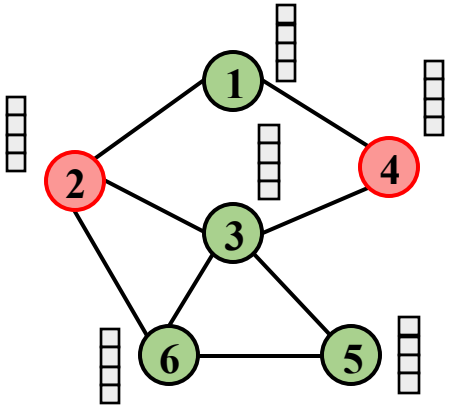
Node Classification – How would traditional classification work?

Data | **Model** | **Loss** | **Optimization**

```
from torch_geometric.datasets import Planetoid
```

Dataset

```
dataset = Planetoid(root='/tmp/Cora', name='Cora')
```



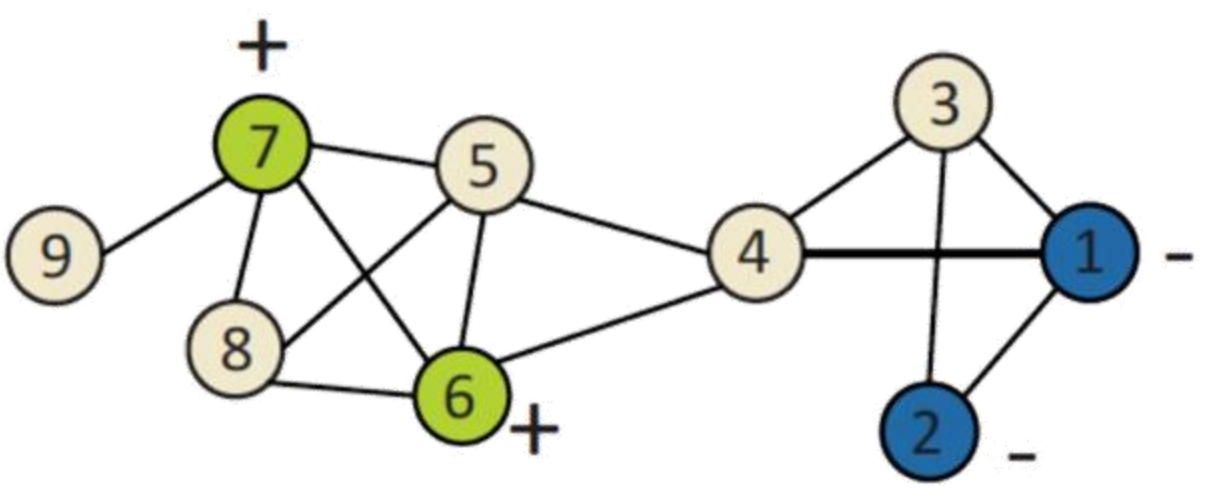

```
dataset[0]
```

```
Data(x=[2708, 1433], edge_index=[2, 10556], y=[2708], train_mask=[2708], val_mask=[2708], test_mask=[2708])
```

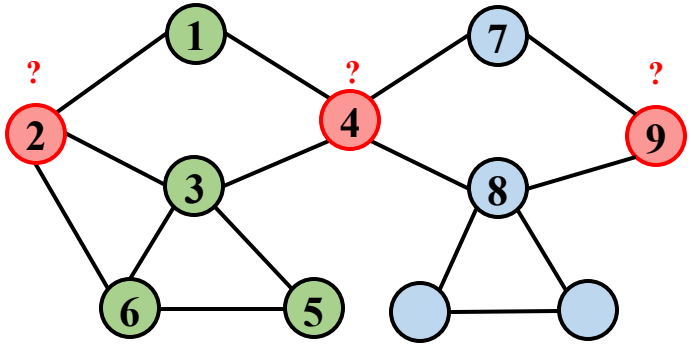
Label Propagation

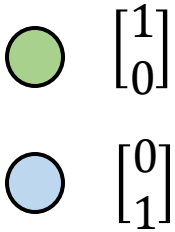
Leveraging this correlation

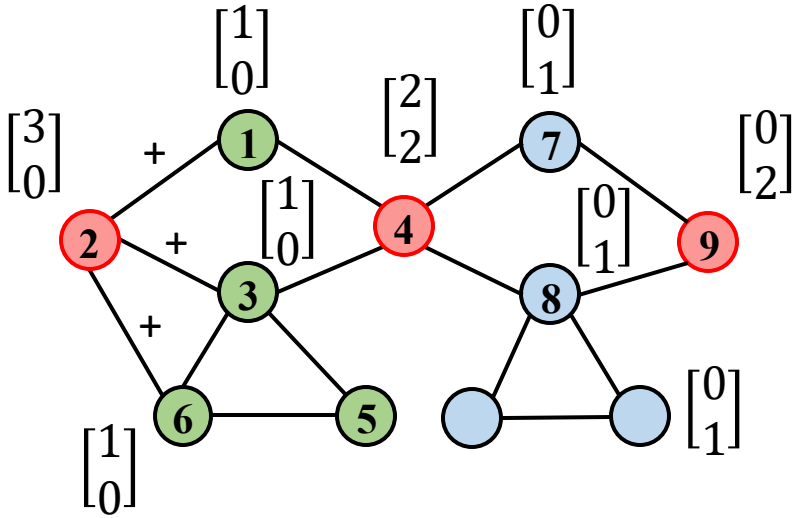
- How to leverage this correlation in networks to predict node classes?

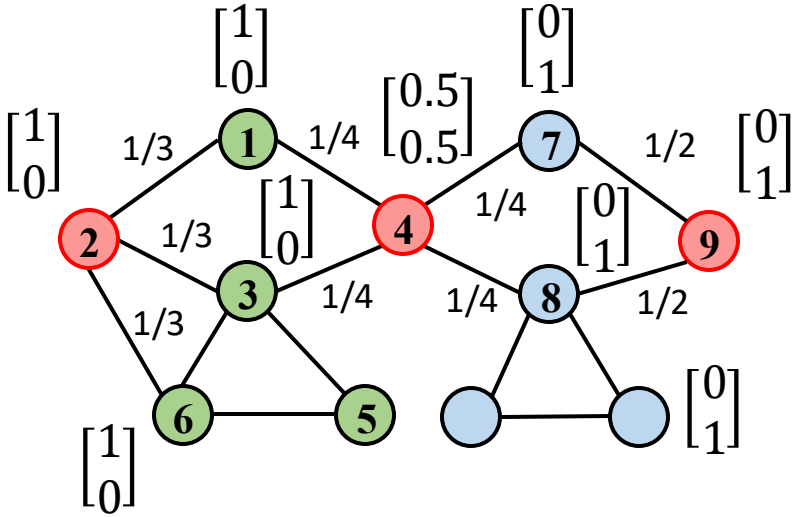


Node Classification – Label Propagation









Node Classification – Label Propagation

```
from torch_geometric.utils import add_remaining_self_loops, degree
from torch_scatter import scatter

def propagate(x, edge_index, edge_weight=None):
    """ feature propagation procedure: sparsematrix
    """
    edge_index, _ = add_remaining_self_loops(edge_index, num_nodes=x.size(0))


    # calculate the degree normalize term
    row, col = edge_index
    deg = degree(col, x.size(0), dtype=x.dtype)
    deg_inv_sqrt = deg.pow(-0.5)
    # for the first order appro of laplacian matrix in GCN, we use deg_inv_sqrt[row]*deg_inv_sqrt[col]
    if(edge_weight == None):
        edge_weight = deg_inv_sqrt[col] * deg_inv_sqrt[col]


    # normalize the features on the starting point of the edge
    out = edge_weight.view(-1, 1) * x[row]


    return scatter(out, edge_index[-1], dim=0, dim_size=x.size(0), reduce='add')
```

```
one_hot_label = F.one_hot(data.y, num_classes=torch.unique(data.y).shape[0])
one_hot_label[data.val_mask] = 0
one_hot_label[data.test_mask] = 0

pred = propagate(one_hot_label, data.edge_index)
pred = pred.argmax(dim=1)
train_acc, val_acc, test_acc = cal_acc(pred, data.y, data.train_mask), cal_acc(pred, data.y, data.val_mask), cal_acc(pred, data.y, data.test_mask)
```

```
print(train_acc, val_acc, test_acc)
```

```
0.9 0.674 0.681
```

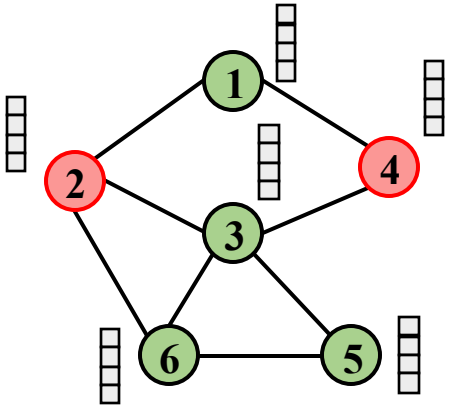
Node Classification

Data | **Model** | **Loss** | **Optimization**

```
from torch_geometric.datasets import Planetoid
```

Dataset

```
dataset = Planetoid(root='/tmp/Cora', name='Cora')
```




```
dataset[0]
```

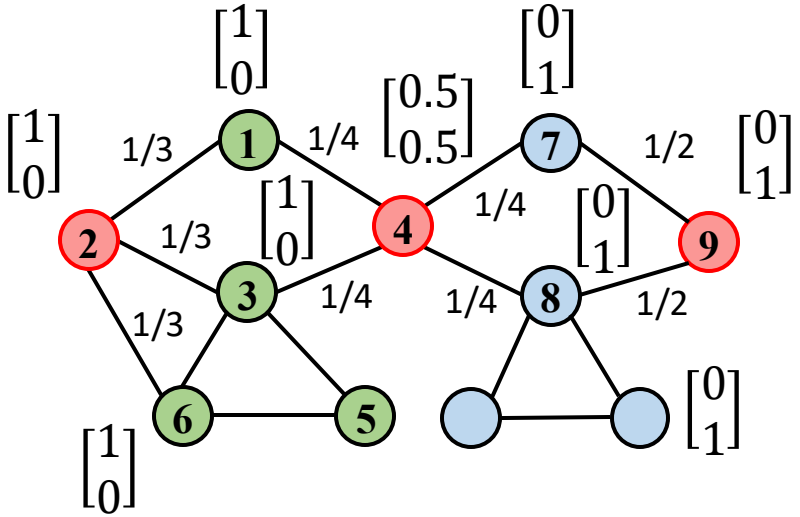
```
Data(x=[2708, 1433], edge_index=[2, 10556], y=[2708], train_mask=[2708], val_mask=[2708], test_mask=[2708])
```

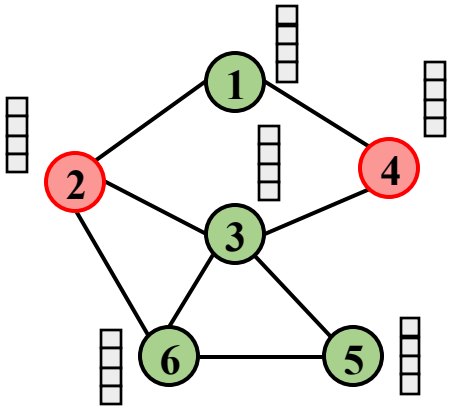
Feature + **Label** = **MLP**

Graph + **Label** = **Label Propagation**

Feature + **Label** + **Graph** = **GNN**

Node Classification – GNN



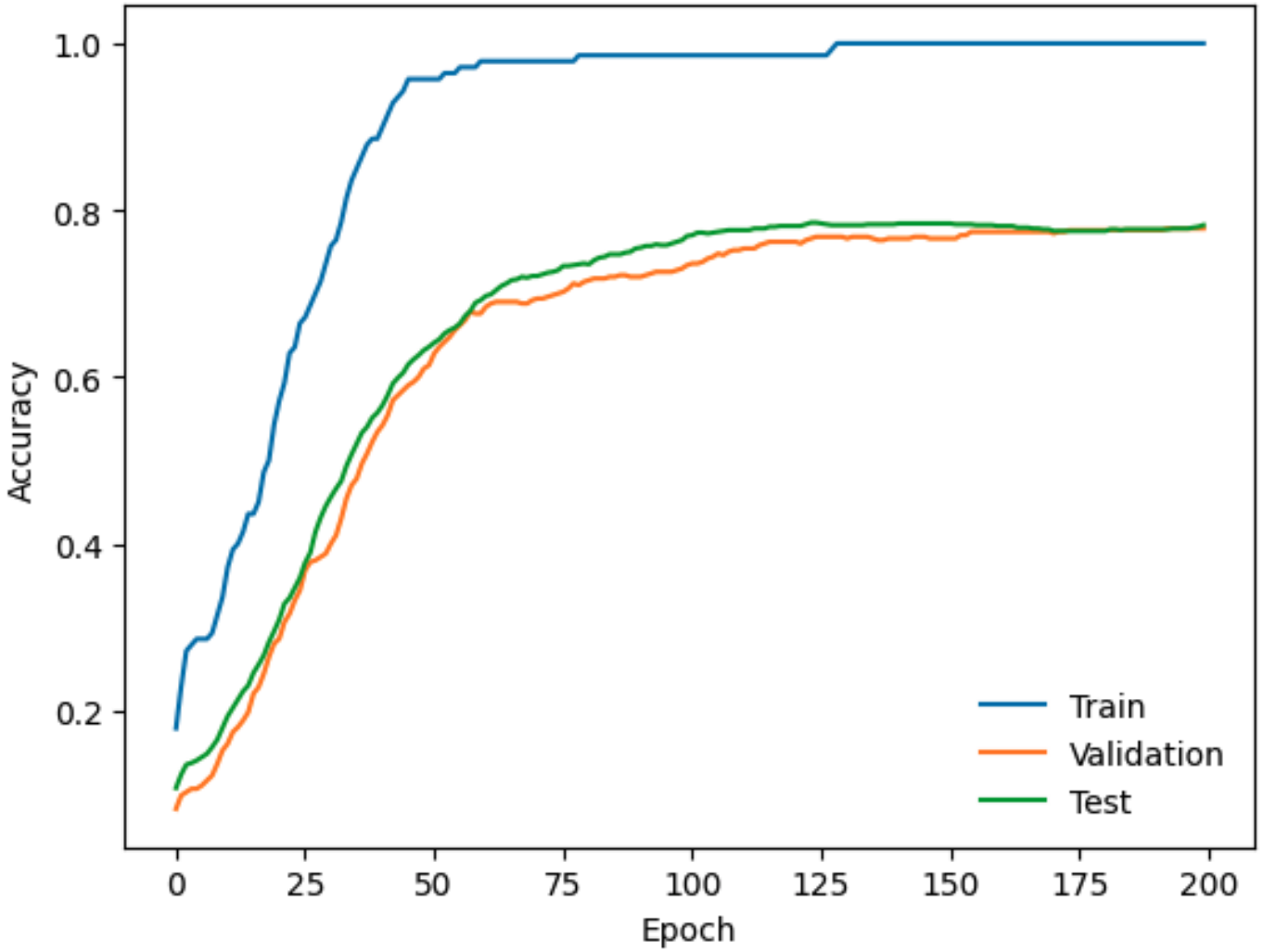


Instead of propagating Label, but we propagate feature!

Node Classification – GNN

```python
class GCN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(GCN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x, edge_index, edge_weight = None):
        x = self.fc1(x)
        x = propagate(x, edge_index)
        x = self.relu(x)
        # x = F.dropout(x, p=0.5, training=self.training)
        x = self.fc2(x)
        x = propagate(x, edge_index)

        return F.log_softmax(x, dim=1)

# Example usage:
input_size = dataset[0].x.shape[1]
hidden_size = 16
output_size = torch.unique(dataset[0].y).shape[0]

GNN = GCN(input_size, hidden_size, output_size)
print(model)
```

Node Classification – GNN



Any Question?