Graph Neural Networks

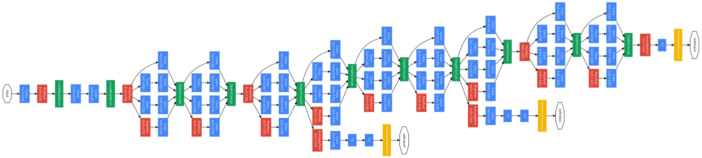
Jan Tönshoff, Timo Gervens, Prof. Dr. Martin Grohe

RWTH Aachen

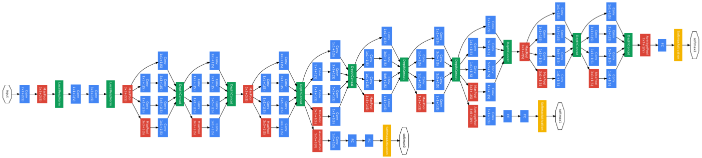
toenshoff@informatik.rwth-aachen.de
gervens@informatik.rwth-aachen.de

November 16, 2022

Neural Networks



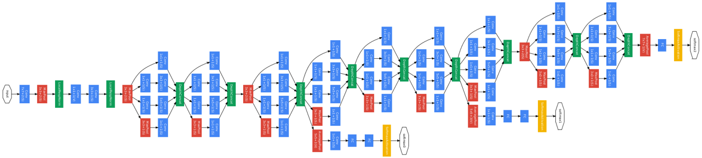
Neural Networks



Basic Idea:

- Build (large) computational graphs with differentiable tensor operations.

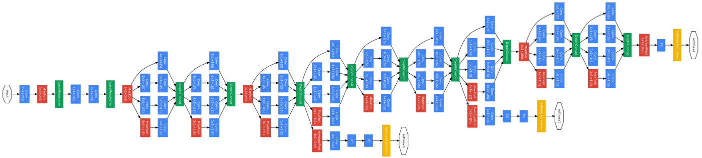
Neural Networks



Basic Idea:

- Build (large) computational graphs with differentiable tensor operations.
- Train with Stochastic Gradient Descent on large amounts of data.

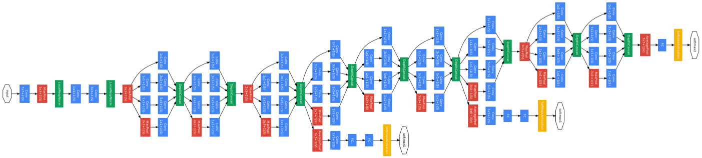
Neural Networks



Basic Idea:

- Build (large) computational graphs with differentiable tensor operations.
- Train with Stochastic Gradient Descent on large amounts of data.
- Usually used for end-2-end learning with little feature engineering.

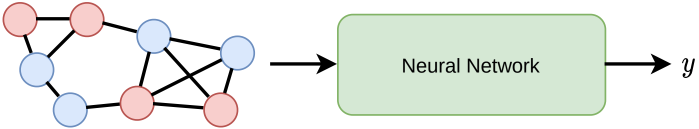
Neural Networks



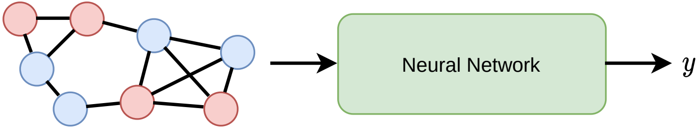
Basic Idea:

- Build (large) computational graphs with differentiable tensor operations.
- Train with Stochastic Gradient Descent on large amounts of data.
- Usually used for end-2-end learning with little feature engineering.
- State-of-the-art for almost all learning domains, including graphs.

Graph Neural Networks



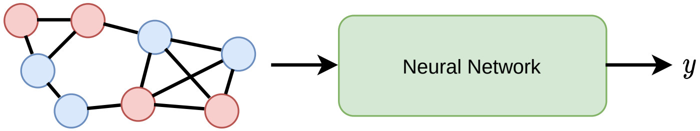
Graph Neural Networks



Fundamental Challenges of Graph Learning:

- Process graphs of any size and structure
- Permutation Invariance

Graph Neural Networks



Fundamental Challenges of Graph Learning:

- Process graphs of any size and structure
- Permutation Invariance

Solution: Message Passing

Message Passing

Input:

- Graph $G = (V, E)$
- Node Features $X_V : V \to \mathbb{R}^d$
- Edge Features $X_E : E \to \mathbb{R}^{d'}$. (optional)

Message Passing

Input:

- Graph $G = (V, E)$
- Node Features $X_V : V \to \mathbb{R}^d$
- Edge Features $X_E : E \to \mathbb{R}^{d'}$. (optional)

Idea of Message Passing:

- Initial embedding $h^0(v) = X_V(v)$.
- Update latent vertex embedding $h^\ell(v) \in \mathbb{R}^{d^\ell}$ in each layer ℓ.

Message Passing

Input:

- Graph $G = (V, E)$
- Node Features $X_V : V \to \mathbb{R}^d$
- Edge Features $X_E : E \to \mathbb{R}^{d'}$. (optional)

Idea of Message Passing:

- Initial embedding $h^0(v) = X_V(v)$.
- Update latent vertex embedding $h^\ell(v) \in \mathbb{R}^{d^\ell}$ in each layer ℓ.
- The update depends on the states of the neighbors.
- Use shared local neural network for the update.

Message Passing

Elements of a GNN Layer [Wu et al., 2020]:

- Trainable functions $\mathbf{M}^{\ell}$, $\mathbf{U}^{\ell}$
- Aggregation function $\bigoplus$ (Sum, Mean, Max,...)

Message Passing

Elements of a GNN Layer [Wu et al., 2020]:

- Trainable functions $\mathbf{M}^\ell$, $\mathbf{U}^\ell$
- Aggregation function $\bigoplus$ (Sum, Mean, Max,...)

Function computed by the layer:

$$h^\ell(v) = \mathbf{U}^\ell\left(h^{\ell-1}(v), \bigoplus_{u \in \mathcal{N}(v)} \mathbf{M}^\ell\left(h^{\ell-1}(u), X_E(vu)\right)\right)$$

Downstream Tasks

Node-Level Tasks

Apply predictive MLP to final vertex embedding:

$$y(v) = \mathrm{MLP}(h^L(v))$$

Downstream Tasks

Node-Level Tasks

Apply predictive MLP to final vertex embedding:

$$y(v) = \mathrm{MLP}(h^L(v))$$

Graph-Level Tasks

Pool vertex embedding into a graph embedding with *readout* function (Sum, Mean, ...):

$$h(G) = \sum_{v \in V} h^L(v)$$

Downstream Tasks

Node-Level Tasks

Apply predictive MLP to final vertex embedding:

$$y(v) = \mathrm{MLP}(h^L(v))$$

Graph-Level Tasks

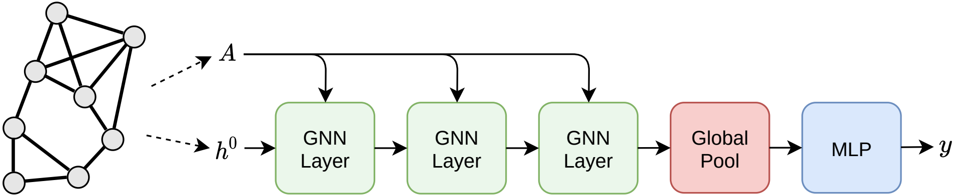
Pool vertex embedding into a graph embedding with *readout* function (Sum, Mean, ...):

$$h(G) = \sum_{v \in V} h^L(v)$$

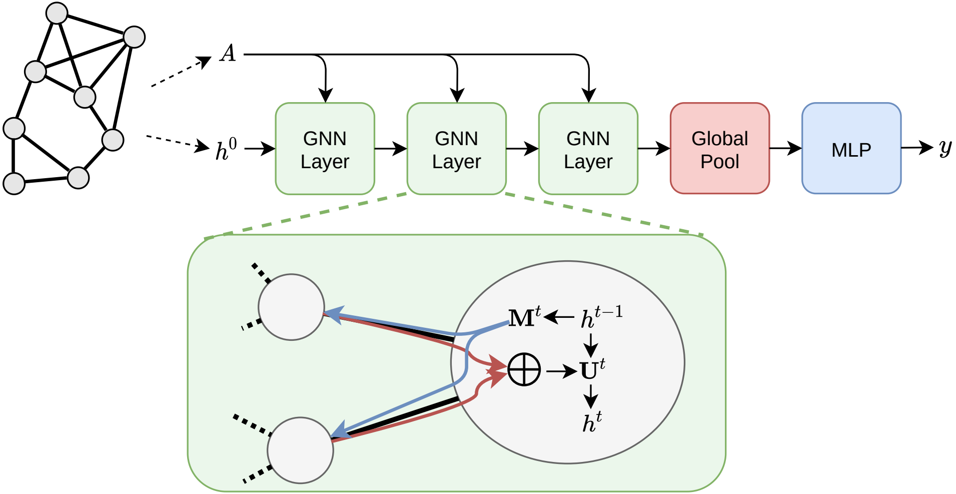
Apply predictive MLP to pooled vector:

$$y(G) = \mathrm{MLP}(h(G))$$

Graph Neural Networks



Graph Neural Networks



Expressiveness of GNNs

Do GNNs seem familiar?

Expressiveness of GNNs

Do GNNs seem familiar?

We can view latent states as vertex colors.
$\Rightarrow$ $\mathbf{M}$, $\mathbf{U}$ and $\bigoplus$ define a HASH function for the colors in the neighborhood of a vertex.

Expressiveness of GNNs

Do GNNs seem familiar?

We can view latent states as vertex colors.
$\Rightarrow$ $\mathbf{M}$, $\mathbf{U}$ and $\bigoplus$ define a HASH function for the colors in the neighborhood of a vertex.

GNNs are a differentiable version of Color Refinement!

Expressiveness of GNNs

Do GNNs seem familiar?

We can view latent states as vertex colors.
$\Rightarrow$ **M**, **U** and $\bigoplus$ define a HASH function for the colors in the neighborhood of a vertex.

GNNs are a differentiable version of Color Refinement!

GNNs are equivalent to the WL-Test. They can not detect substructures that the WL-Kernel could not already detect!

Expressiveness of GNNs

GNNs are **at most** as expressive as the WL-Kernel.

Expressiveness of GNNs

GNNs are **at most** as expressive as the WL-Kernel.

Why can we still expect them to (usually) work better?

Expressiveness of GNNs

GNNs are **at most** as expressive as the WL-Kernel.

Why can we still expect them to (usually) work better?

- WL views all subtrees as strictly different.

Expressiveness of GNNs

GNNs are **at most** as expressive as the WL-Kernel.

Why can we still expect them to (usually) work better?

- WL views all subtrees as strictly different.
- Slightly different subtrees may have identical effect on the label.

Expressiveness of GNNs

GNNs are **at most** as expressive as the WL-Kernel.

Why can we still expect them to (usually) work better?

- WL views all subtrees as strictly different.
- Slightly different subtrees may have identical effect on the label.
- A non-injective HASH function may therefore yield a more robust classifier.

Expressiveness of GNNs

GNNs are **at most** as expressive as the WL-Kernel.

Why can we still expect them to (usually) work better?

- WL views all subtrees as strictly different.
- Slightly different subtrees may have identical effect on the label.
- A non-injective HASH function may therefore yield a more robust classifier.
- GNNs effectively learn a fitting HASH function from data.

Expressiveness of GNNs

GNNs are **at most** as expressive as the WL-Kernel.

Why can we still expect them to (usually) work better?

- WL views all subtrees as strictly different.
- Slightly different subtrees may have identical effect on the label.
- A non-injective HASH function may therefore yield a more robust classifier.
- GNNs effectively learn a fitting HASH function from data.
- GNNs also seamlessly handle real-valued node and edge features.

Exercise 2 and 3

In the next two sheets you will implement GNNs:

- Sheet 2: Implement the GCN using simple dense data structures.
- Sheet 3: Implement a general GNN Layer with efficient sparse data structures.

Graph Convolutional Neural Network

The Graph Convolutional Neural Network (GCN) [Kipf and Welling, 2016]:

- One of the first GNNs that actually worked.
- Simple and effective.
- Still often used as the default GNN for many tasks.
- Limitations: Does not incorporate edge features.

The GCN Layer

Components:

- Trainable weight matrix $W^{(\ell)} \in \mathbb{R}^{d^{(\ell-1)} \times d^{(\ell)}}$
- Activation Function σ (usually ReLU)

The GCN Layer

Components:

- Trainable weight matrix $W^{(\ell)} \in \mathbb{R}^{d^{(\ell-1)} \times d^{(\ell)}}$
- Activation Function σ (usually ReLU)

Update rule:

$$h^{\ell}(v) = \sigma\Big(\sum_{u \in N(v) \cup \{v\}} \frac{1}{\sqrt{d_v d_u}} \cdot h^{\ell-1}(u) \cdot W^{(\ell)} \Big)$$

with $d_i = \deg(v_i) + 1$.

The GCN Layer

Components:

- Trainable weight matrix $W^{(\ell)} \in \mathbb{R}^{d^{(\ell-1)} \times d^{(\ell)}}$
- Activation Function σ (usually ReLU)

Update rule:

$$h^{\ell}(v) = \sigma\Big(\sum_{u \in N(v) \cup \{v\}} \frac{1}{\sqrt{d_v d_u}} \cdot h^{\ell-1}(u) \cdot W^{(\ell)} \Big)$$

with $d_i = \deg(v_i) + 1$.

In the general GNN framework: **M** is the identity function, **U** is a one layer perceptron without bias and $\bigoplus$ is a weighted sum.

Implementing the GCN Layer

Stack vertex embeddings in matrix: $H^{(\ell)} \in \mathbb{R}^{|V| \times d^{(\ell)}}$

Implementing the GCN Layer

Stack vertex embeddings in matrix: $H^{(\ell)} \in \mathbb{R}^{|V| \times d^{(\ell)}}$

Pre-Compute normalized adjacency matrix $\tilde{A} \in \mathbb{R}^{|V| \times |V|}$:

$$\tilde{A}_{i,j} = \begin{cases} \frac{1}{\sqrt{d_i d_j}} & \text{if } v_i v_j \in E \text{ or } i = j \\ 0 & \text{otherwise} \end{cases}$$

Implementing the GCN Layer

Stack vertex embeddings in matrix: $H^{(\ell)} \in \mathbb{R}^{|V| \times d^{(\ell)}}$

Pre-Compute normalized adjacency matrix $\tilde{A} \in \mathbb{R}^{|V| \times |V|}$:

$$\tilde{A}_{i,j} = \begin{cases} \frac{1}{\sqrt{d_i d_j}} & \text{if } v_i v_j \in E \text{ or } i = j \\ 0 & \text{otherwise} \end{cases}$$

The layer can now be expressed with efficient matrix operations:

$$H^{(\ell)} = \sigma(\tilde{A} \cdot H^{(\ell-1)} \cdot W^{(\ell)})$$

Considerations

Implementing GNNs with dense tensors:

- Dimension 0 is reserved as the *batch* dimension.
- For a batch of size b we have $A \in \mathbb{R}^{b \times |V| \times |V|}$ and $H^0 \in \mathbb{R}^{b \times |V| \times d}$
- Graphs have different sizes: Pad A and X_V with zeros to uniform sizes over the whole dataset.

Exercise 2

Implement the GCN in PyTorch:

- Pre-Process graphs to compute padded, normalized adjacency matrices.
- Implement the GCN Layer as PyTorch Module.
- Construct Node- and Graph-Level GNNs with your GCN Layer.

Exercise 2

Implement the GCN in PyTorch:

- Pre-Process graphs to compute padded, normalized adjacency matrices.
- Implement the GCN Layer as PyTorch Module.
- Construct Node- and Graph-Level GNNs with your GCN Layer.

Perform 10-fold Cross-Validation on 4 datasets:

- Cora
- Citeseer
- NCI1
- ENZYMES

PyTorch

Numpy-like math framework with GPU support and backpropagation.

Defacto Standard for Graph Neural Networks.

Many useful open source extensions (`torch-scatter`, `torch-vision`, ...)

PyTorch

Numpy-like math framework with GPU support and backpropagation.

Defacto Standard for Graph Neural Networks.

Many useful open source extensions (`torch-scatter`, `torch-vision`, . . .)

We will build a Multi-Layer Perceptron as an example.

A Basic PyTorch Module

```
import torch

class Layer(torch.nn.Module):

    def __init__(self, dim_in, dim_out, is_linear=False):
        super(Layer, self).__init__()
        self.is_linear = is_linear

        # use Kaiming Init when using ReLU
        self.W = torch.nn.Parameter(torch.zeros(dim_in, dim_out))
        torch.nn.init.kaiming_normal_(self.W)

        self.b = torch.nn.Parameter(torch.normal(0.0, 1.0, size=(dim_out,)))

    def forward(self, x):
        # linear transformation on input
        y = torch.matmul(x, self.W) + self.b

        # apply activation
        if not self.is_linear:
            y = torch.relu(y)
        return y

```

Combining Modules

```python
class MLP(torch.nn.Module):

    def __init__(self, input_dim, output_dim, hidden_dim, num_layers):
        super(MLP, self).__init__()
        self.num_layers = num_layers

        # add sub-modules as attribute
        self.input_layer = Layer(input_dim, hidden_dim)
        self.output_layer = Layer(hidden_dim, output_dim, is_linear=True)

        # Store multiple submodules in "ModuleList"
        self.hidden_layers = torch.nn.ModuleList(
            [Layer(hidden_dim, hidden_dim) for _ in range(num_layers-2)]
        )

    def forward(self, x):
        # apply layers
        y = self.input_layer(x)
        for i in range(self.num_layers-2):
            y = self.hidden_layers[i](y)
        y = self.output_layer(y)
        return y
```

Before Training

```python
import numpy as np
from torch.utils.data import TensorDataset, DataLoader

# load numpy data and cast to torch tensors
X, Y = np.load('training_data.npy')
X = torch.tensor(X, dtype=torch.float32)
Y = torch.tensor(Y)

# create dataset and loader for mini batches
train_dataset = TensorDataset(X, Y)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)

# set to 'cuda' if gpu is available
device = 'cpu'

# construct neural network and move it to device
model = MLP(input_dim=2, output_dim=10, hidden_dim=100, num_layers=4)
model.train()
model.to(device)

# construct optimizer
opt = torch.optim.Adam(model.parameters(), lr=0.001)

```

Training Loop

```
from torch.nn.functional import cross_entropy

for epoch in range(100):
    for x, y_true in train_loader:
        # set gradients to zero
        opt.zero_grad()

        # move data to device
        x = x.to(device)
        y_true = y_true.to(device)

        # forward pass and loss
        y_pred = model(x)
        loss = torch.nn.functional.cross_entropy(y_pred, y_true)

        # backward pass and sgd step
        loss.backward()
        opt.step()

    # Insert a validation loop and some logging here...

```

References I

Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 2020.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.