

Building Neural Networks with **PyTorch**

And Graph Neural Networks
with **PyTorch Geometric**



Samuel Homberg



What is PyTorch?

- Free and open-source machine learning library
- Well documented Python interface (and C++ interface)
- Built-in GPU acceleration
- Originally developed by Meta (Facebook)
- Now governed by the Linux Foundation




```
import torch
```

- Alternatives?

PyTorch vs Tensorflow



- Free and open-source
- First developed by Meta, now by the Linux foundation
- Used mainly in research/academia but also by Tesla, ...
- Strong community movement
- Mobile support is only in beta
- PyTorch code is very “pythonic”
- PyTorch Geometric 



- Free and open-source
- Developed by Google
- Used in Googles own apps (Maps, ...) and Twitter, ...
- Production-ready deployment options
- Advanced mobile support
- Better built-in visualizations
- With Keras: Code looks like PyTorch



➔ **By now, both libraries are quite similar and can solve the same problems**

Tensors: Multidimensional Arrays

- Not to be confused with more complicated tensors from physics / differential geometry
- Similar to numpy arrays
- Can be loaded onto the GPU

```
torch.tensor(3)  
> tensor(3)
```

```
torch.tensor([[1, 1], [-1, 0]])  
> tensor([[ 1, 1],  
          [-1, 0]])
```

```
torch.tensor([[1, 1], [-1, 0])).shape  
> torch.Size([2, 2])
```

```
torch.tensor(np.array([[1],[7],[5]]))  
> tensor([[1],  
          [7],  
          [5]])
```

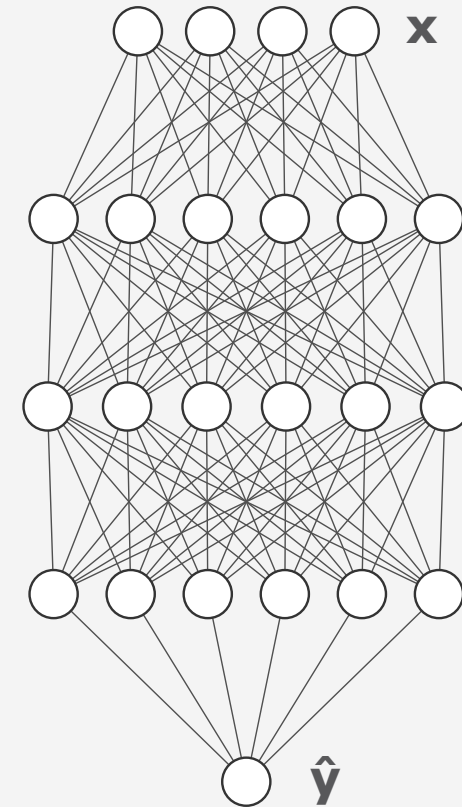
```
torch.zeros((2, 3))  
> tensor([[0., 0., 0.],  
          [0., 0., 0.]])
```

```
torch.randn(3,2)  
> tensor([[ 1.0128, -0.3360],  
          [-0.3667, 1.4099],  
          [-2.5155, 0.3394]])
```

```
torch.randn(3,2).to('cpu').device # or: 'gpu'  
> device(type='cpu')
```

Recap: How do neural networks work?

- What is a NN? → A function that predicts an output $\hat{\mathbf{y}}$ for input \mathbf{x} .
- Has different *layers* which perform mathematical operations on \mathbf{x} .
- For a linear layer: The parameters of \mathbf{W} (weights) multiplied with \mathbf{x} and other parameters \mathbf{b} (biases) are added. → $\mathbf{xW}^T + \mathbf{b}$
- Nonlinear activation functions (ReLU, sigmoid, ...) transform the input further and allow to predict nonlinear correlations.
- Final operation: sigmoid function (to predict a single value → regression task) or softmax function (to predict the probability to belong to a class → classification)
- Training:
 - The error (loss) between the output $\hat{\mathbf{y}}$ and the true label \mathbf{y} has to be minimized:
 - Derivative of NN with regards to the parameters in \mathbf{W} and \mathbf{b} → gradient
 - \mathbf{W} and \mathbf{b} are changed a little bit depending on the gradient. → backpropagation
 - $\hat{\mathbf{y}}$ is calculated again.



Mathematical Operations with Tensors and Keeping Track

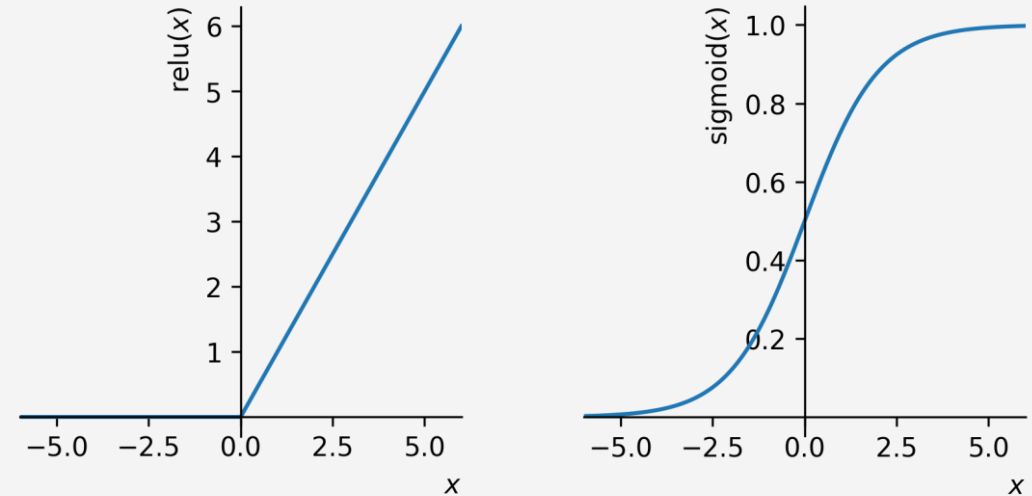
```
t1 = torch.tensor([[ 1.,  2., -1.],
                   [-2., -3.,  0.]])
```

```
t2 = torch.tensor([[ 2.,  3.],
                   [-2., -2.],
                   [ 0., -1.]])
```

```
torch.matmul(t1, t2)
> tensor([[ -2.,  0.],
          [ 2.,  0.]])
```

$$\begin{aligned} &1 \cdot 2 \\ &+ 2 \cdot (-2) \\ &+ (-1) \cdot 0 \\ &= -2 \end{aligned}$$

```
import torch.nn.functional as F
F.relu(t1)
> tensor([[1., 2., 0.],
          [0., 0., 0.]])
```



```
t1 = torch.tensor([[ 1.,  2., -1.],
                   [-2., -3.,  0.]],
                   requires_grad=True)
torch.matmul(t1, t2)
> tensor([[ -2.,  0.],
          [ 2.,  0.]])
grad_fn=<MmBackward0>
```

Backpropagation with autograd

Autograd is a “**reverse automatic differentiation system**”

Backpropagation with autograd

- A neural net is just a big function

$$f(g(h(\dots(x)))) = \hat{y}$$

- Where the nested functions are the different layers and nonlinear activation functions
- Optimizing this function means finding derivative of the error between label y and prediction \hat{y}
- The derivative of a nested function can be found with the chain rule

$$(f(g(x)))' = f'(g(x)) g'(x)$$

- It's a bit more complicated because we don't want to differentiate for x but for our weights **W** and biases **b**
- Calculate the gradient instead (partial differentiation for all parameters)

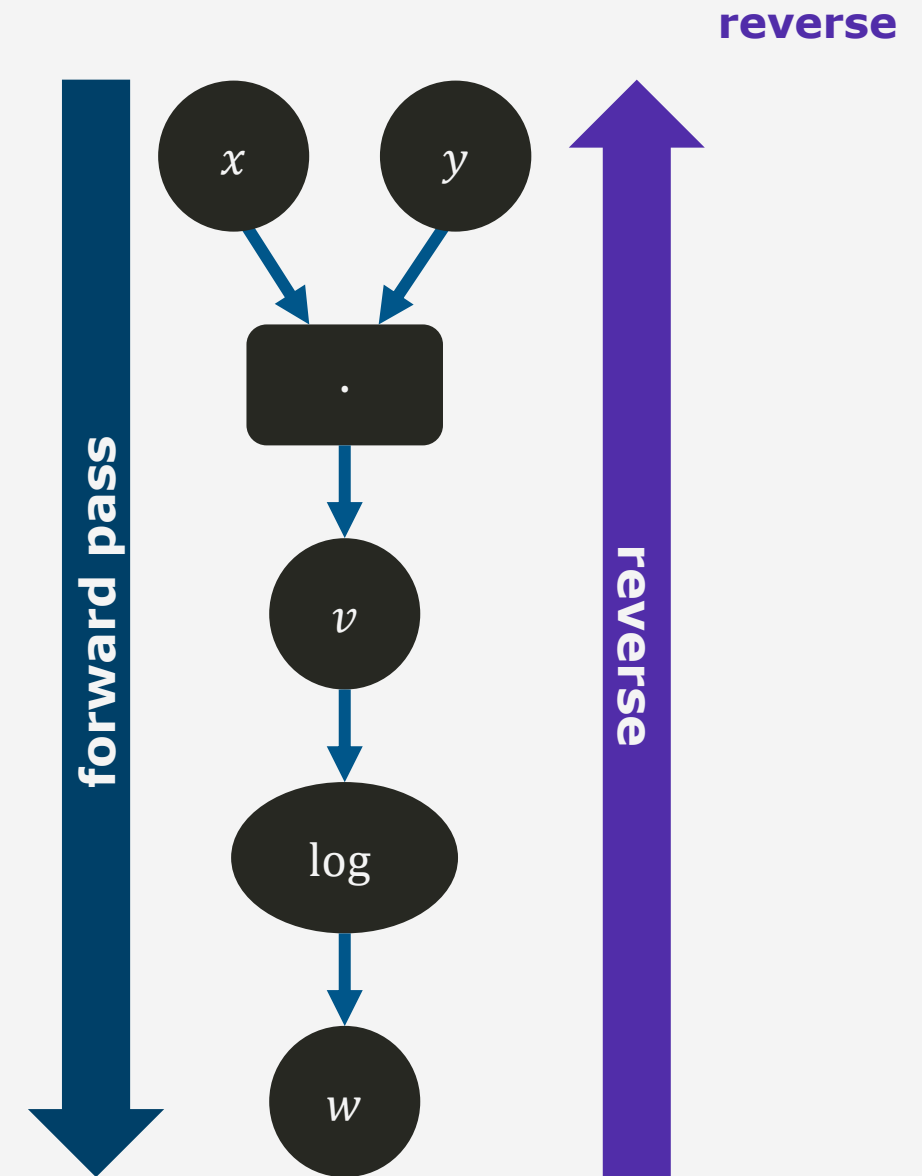
$$\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \dots, \frac{\partial f}{\partial w_n}$$

Backpropagation with autograd

- Possible by hand, but a lot of work.
 - What if we want to exchange a function in the middle of the net?
- We need automatic differentiation.
- Because of the chain rule: relatively easy

Backpropagation with autograd

- During the forward pass all operations are recorded
- Operations are stored in an acyclic *graph*
- Leaves are the input tensors, the root is the last operation of the neural network
- Example: $f(x, y) = \log(xy)$
- For backpropagation: The graph is traversed in the opposite direction, from root to leaves to calculate the gradient
- Computational graph is created every iteration



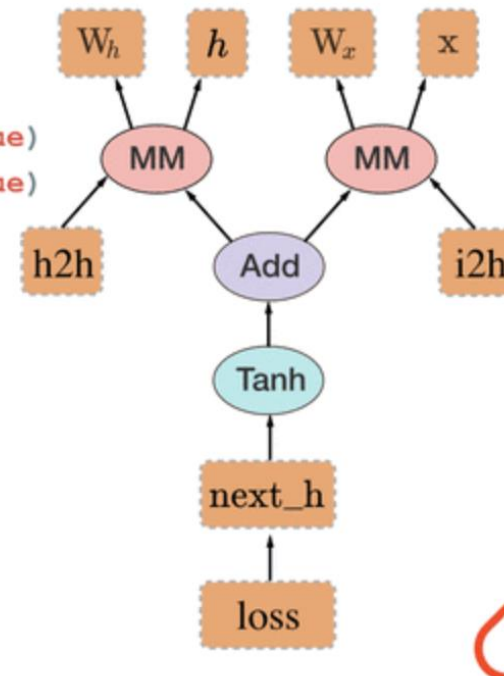
Backpropagation with autograd

Back-propagation
uses the dynamically created graph

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```

```
h2h = torch.mm(W_h, prev_h.t())
i2h = torch.mm(W_x, x.t())
next_h = h2h + i2h
next_h = next_h.tanh()
```

```
loss = next_h.sum()
loss.backward() # compute gradients!
```



Source

Machine Learning Optimization Framework

(Not just for neural networks)

- **Choose a model** describing the relationship between input and output variables
- **Define a loss** (error) function quantifying the fit to the data
 - (optional) chose a regularizer saying how much we prefer different candidates (integration of knowledge)
- **Fit the model** using an **optimization algorithm**

Choose a Model

- Combine basic mathematical operations
- Combine basic layers
 - Using `nn.Sequential`
 - Inheriting from `nn.Module`

```
torch.matmul(x, W.transpose()) + b)
```

```
F.linear(x, W, b)
```

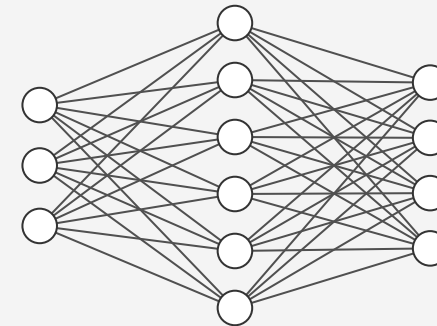
```
layer1 = nn.Linear(3, 2)
```

```
x = torch.tensor([[ 1.,  2., -1.],
                  [-2., -3.,  0.]])
```

```
layer1(x)
```

```
> tensor([[ -1.0423, -1.4743],
          [  1.3903, -0.0618]],
         grad_fn=<AddmmBackward0>)
```

```
model = nn.Sequential(
    nn.Linear(3,6),
    nn.ReLU(),
    nn.Linear(6,4),
    nn.ReLU()
)
model(x)
> tensor([[0.0692, 0.4618, 0.4903, 0.0000]
          [1.5100, 0.0000, 0.0000, 0.0000]],
         grad_fn=<ReluBackward0>)
```



Choose a Model

```
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.lin1 = nn.Linear(3, 6)
        self.lin2 = nn.Linear(6, 4)

    def forward(self, x):
        x = F.relu(self.lin1(x))
        return F.relu(self.lin2(x))

model = Model()
model(x)
> tensor([[0.4112, 0.0000, 0.0000, 0.0000],
          [0.9921, 0.6823, 0.0000, 0.0000]],
        grad_fn=<ReluBackward0>)
```

- Send model to device

```
model.to('cuda')
```

- Can be nested

- Change training mode:

```
model.train()
model.eval()
```

- Important for `nn.Dropout` and
`nn.BatchNorm`

- Many more specialized layers (convolutional, recurrent, ...)

Define a Loss

- Function to calculate error / distance between the prediction $\hat{\mathbf{y}}$ and the label \mathbf{y}

- For regression tasks (predicting a single numerical value)

- Mean squared error (MSE)

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

```
loss = nn.MSELoss()
```

- For classification tasks

- Crosseentropy loss

```
loss = nn.CrossEntropyLoss()
```

- For binary classification tasks

- Binary crosseentropy loss

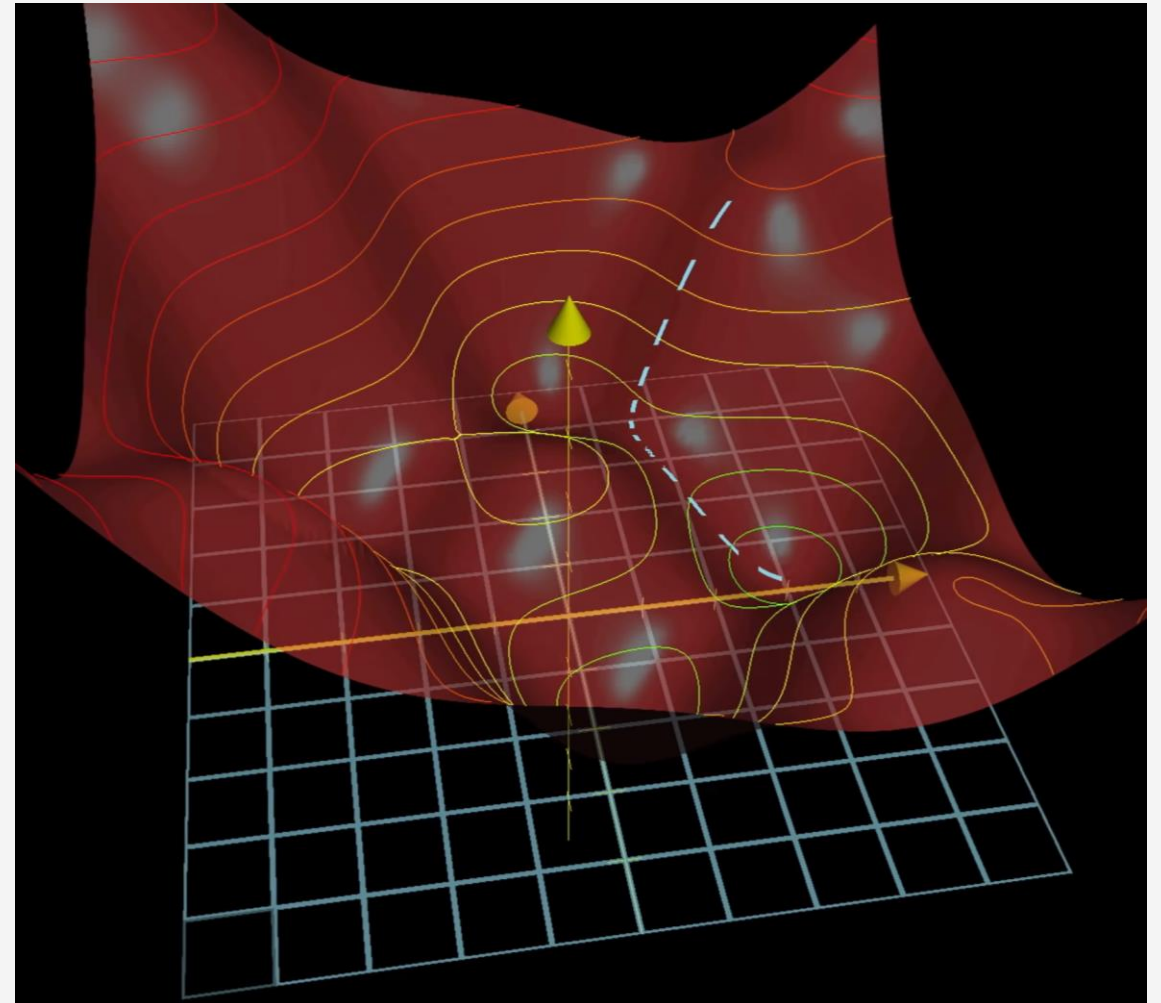
```
loss = nn.BCEWithLogitsLoss()
```

- More functions available

Choose an optimizer

- Algorithm to find the minimum of the loss function
 - Gradient points in the direction of the greatest increase
 - Step into the opposite direction
 - “Walking from a mountain into a valley”
- Different algorithms implemented
 - Stochastic gradient descent
 - Adam
 - ...

```
from torch import optim
optimizer = optim.Adam(model.parameters(),
                        lr=0.001)
```



Train the Model

```
for epoch in range(5):  
    model.train()  
    for (X, y) in dataloader:  
  
        # Compute prediction error  
        pred = model(X)  
        loss = loss(pred, y)  
  
        # Backpropagation  
        optimizer.zero_grad()  
        loss.backward()  
        optimizer.step()
```

- dataloader → see Jupyter Notebook
- After one epoch: the network has seen all training data once
- Between epochs the gradients have to be set back to zero
- Other things to add to the trainings-loop / trainings-function:
 - Sending data to the device the model is on
 - Updating the engineer on the training-process
 - **Evaluate the model**

Evaluate the Model

```
for epoch in range(5):  
    model.eval()  
    test_loss = 0  
    with torch.no_grad():  
        for X, y in testdata_loader:  
            pred = model(X)  
            test_loss += loss(pred, y).item()  
    print(test_loss)
```

- Evaluate on data different from the training set
- Use different metrics for evaluation

Other things to know

- Datasets: torchvision, torchaudio
 - Not relevant for chemistry

- Saving and loading models

```
# saving
torch.save(model.state_dict(), "model.pt")

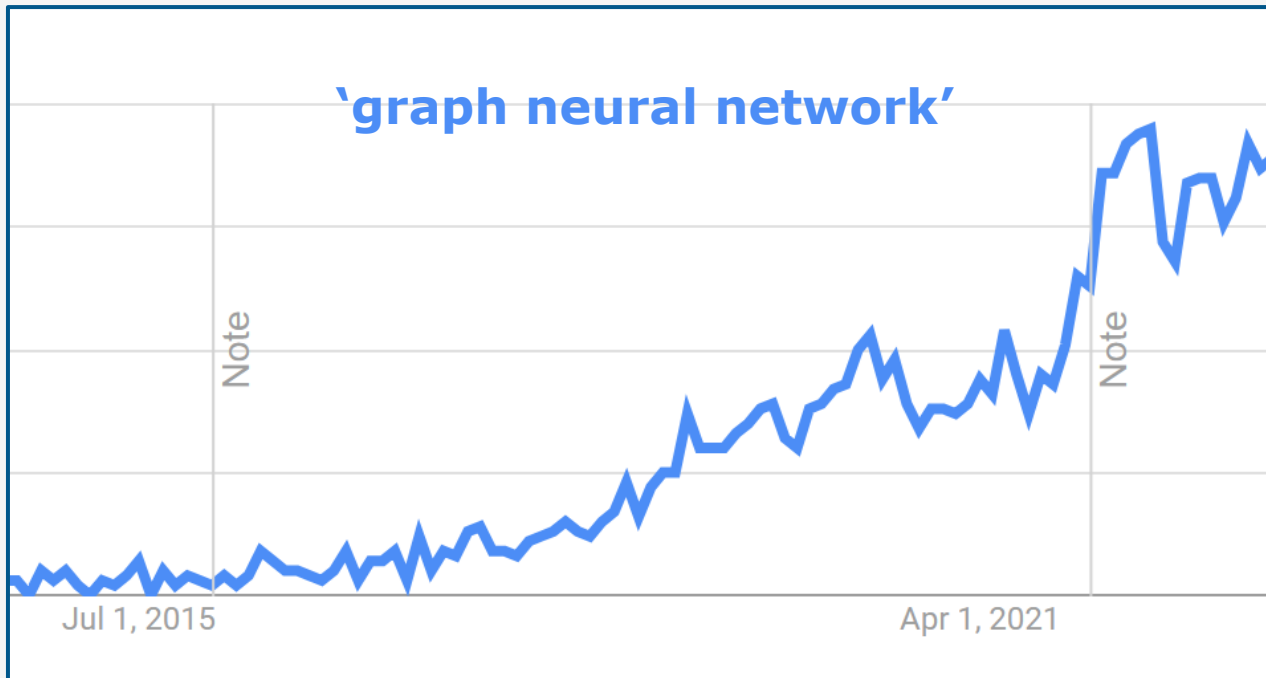
# loading
model = Model()
model.load_state_dict(torch.load("model.pt"))
```

- TorchScript: Build models with Python and run without python dependency
- PyTorch Mobile
- Captum
- Profiler

Questions?

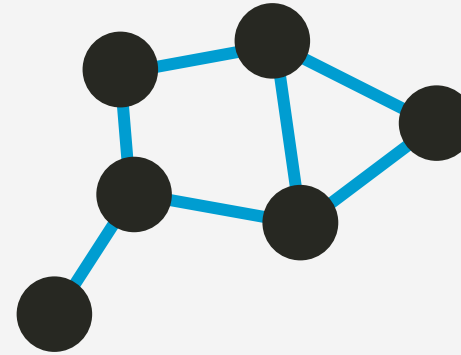
Building Graph Neural Networks with **PyTorch Geometric**

- What is PyTorch Geometric?
 - Library built upon PyTorch to easily write and train GNNs

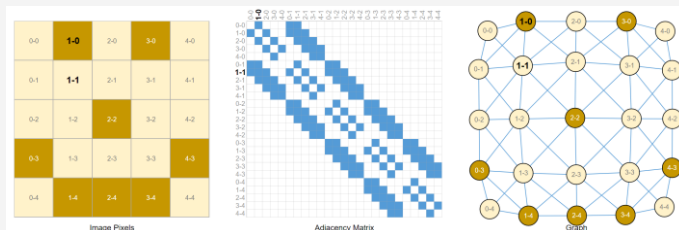


What is a Graph?

- Mathematical structure used to model pairwise relations between objects (Wikipedia)
- Models for networks in the real world
 - Social networks (in real life and online)
 - Street networks
 - **Molecules**
 - ...

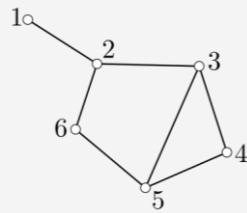


- Made up of nodes (vertices) connected by edges (links)
- Anything can be modeled as a graph



Graphs in PyTorch Geometric

- Adjacency matrix
- Edge index
- Node feature matrix
- (Edge feature matrix)



| | 1 | 2 | 3 | 4 | 5 | 6 | |
|---|---|---|---|---|---|---|-------|
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | {1,2} |
| 2 | 1 | 0 | 1 | 0 | 0 | 1 | {2,3} |
| 3 | 0 | 1 | 0 | 1 | 1 | 0 | {3,4} |
| 4 | 0 | 0 | 1 | 0 | 1 | 0 | {3,5} |
| 5 | 0 | 0 | 1 | 1 | 0 | 1 | {4,5} |
| 6 | 0 | 1 | 0 | 0 | 1 | 0 | {5,6} |

```
edge_index = torch.tensor(
    [[0, 1, 1, 2, 2, 3, 2, 4, 3, 4, 4, 5, 5, 1],
     [1, 0, 2, 1, 3, 2, 4, 2, 4, 3, 5, 4, 1, 5]],
    dtype=torch.long)
torch_geometric.utils.to_dense_adj(edge_index)
> tensor([[[0., 1., 0., 0., 0., 0.],
           [1., 0., 1., 0., 0., 1.],
           [0., 1., 0., 1., 1., 0.],
           [0., 0., 1., 0., 1., 0.],
           [0., 0., 1., 1., 0., 1.],
           [0., 1., 0., 0., 1., 0.]])])
```

```
x = torch.tensor([[1], [2],
                  [3], [4],
                  [5], [6]],
                  dtype=torch.float)
data = torch_geometric.data.Data(x=x,
                                  edge_index=edge_index)
print(data)
> Data(x=[6, 1], edge_index=[2, 14])

# dictionary structure
print(data.x)
> tensor([[1.], [2.], [3.], [4.],
          [5.], [6.]])

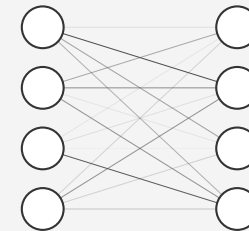
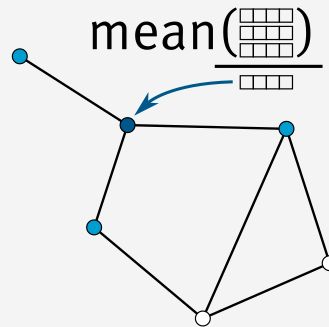
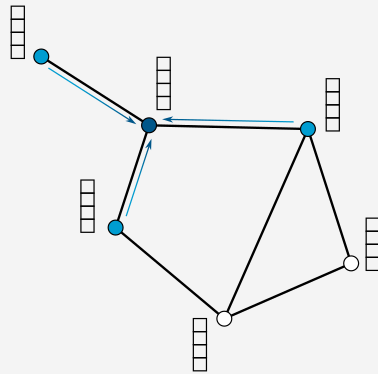
print(data.keys)
> ['x', 'edge_index']
```

What do we learn?

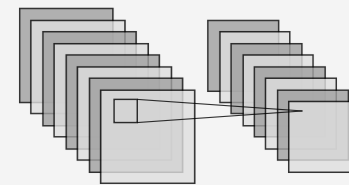
- Node prediction → Does this node belong to class xy ?
 - Example citation graph: Is this a neuroscience paper?
- Edge prediction
 - Edge classification
 - Predicting the existence of an edge between nodes i and j
 - Facebook/Twitter: You might know / like ...
- Graph prediction
 - Molecular properties / activities ...

How do we learn?

- Message passing:
 - For each node, get all the neighboring nodes/node embeddings
 - Aggregate all messages (sum / mean / max; permutation invariant)
 - Pooled messages are updated by a neural network function (linear layer)

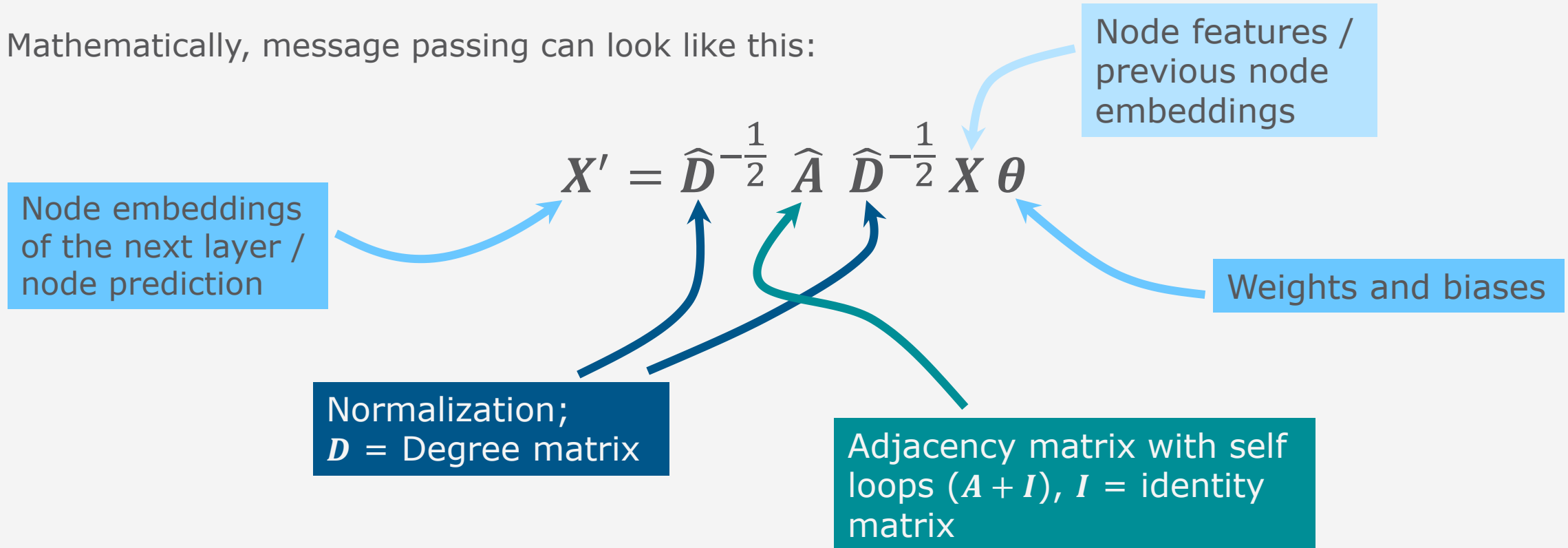


- Similar to pixel-convolutions from image recognition
- In practice: Now many implementations



GCNConv – A Simple Graph Convolution

- Mathematically, message passing can look like this:



- Note: For graph-level tasks we need to pool the node embeddings (and add a fully connected layer)

Building a GNN

```
from torch_geometric.nn import GCNConv
from torch_geometric.nn import \
    global_add_pool

class GCN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = GCNConv(1, 3)
        self.conv2 = GCNConv(3, 1)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = self.conv2(x, edge_index)
        # x = global_add_pool(x, batch)
        return F.log_softmax(x, dim=1)
```

```
model = GCN()
model(data.x, data.edge_index)
> tensor([[0.],
          [0.],
          [0.],
          [0.],
          [0.],
          [0.]],
        grad_fn=<LogSoftmaxBackward0>)
```

More Noteworthy Stuff

- PyTorch Geometric
 - Datasets (KarateClub, **TUDataset**, ..., **MoleculeNet**, ..., **ZINC**, ...)
 - Explain → XAI Algorithms
 - Transforms (ToUndirected, KNNGraph, ...)
 - Utils (add_self_loops, to_dense_adj, **from_smiles**, ...)
 - GraphGym: Design/evaluate pipeline
 - Profiler
- Other packages:
 - NetworkX → More graph analysis/creation/... tools
- (Graph) Neural Networks are not necessarily the best model for your application! (Also try RF, SVM, ...)

Questions?

Exercises / Jupyter Notebooks

https://github.com/SamuelHomberg/SPP2363_Tutorial_PyTorch

- PyTorch:
 - https://github.com/kochgroup/intro_pharma_ai
→ Notebook 8 (GER/EN; Google Colab; with exercises)
 - Other notebooks (Cheminformatics, Datascience, NNs from scratch, GNNs without PyTorch Geometric, ...)
- PyTorch Geometric:
 - Jupyter notebook with chapters on
 - Installation of PyG
 - Loading of existing datasets
 - Creating your own dataset
 - Training a simple GNN
 - No exercises; no Google Colab