Probabilistic (Graphical) Models

and inference

Oliver Obst · Autumn 2024

WESTERN SYDNEY
UNIVERSITY

Probabilistic (Graphical) Models and Inference

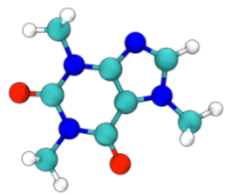
(PGM: Probabilistic Graphical Models: Principles and Techniques by Daphne Koller and Nir Friedman. MIT Press)
(PMLI: Probabilistic Machine Learning: An introduction by Kevin Murphy. MIT Press)

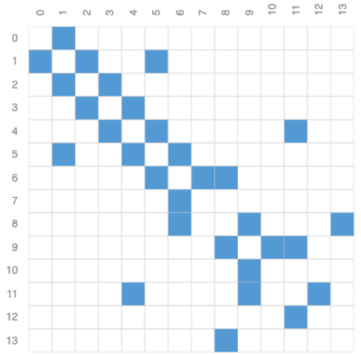
Week	Lecture	Required reading	Assessment
1 Monday, 4 March 2024	Introduction, Probability Theory	PGM Chapter 2, PMLI Chapter 6.1	
2 Monday, 11 March 2024	Directed and undirected networks introduction	PGM Chapter 3 & 4	Quiz 1
3 Monday, 18 March 2024	Variable elimination	PGM Chapter 9	
4 Monday, 25 March 2024	Belief propagation	PGM Chapter 10/11	Quiz 2
5 Monday, 1 April 2024	public holiday		5 April 2024: census date
6 Monday, 8 April 2024	Message passing / Graph neural networks	https://distill.pub/2021/gnn-intro/	
7 Monday, 15 April 2024	Sampling	PGM Chapter 12	Quiz 3
8 Monday, 22 April 2024	Mid-term break		
9 Monday, 29 April 2024	Variational inference		Intra-session exam
10 Monday, 6 May 2024	Autoregressive models		Quiz 4
11 Monday, 13 May 2024	Variational Auto-Encoders		
12 Monday, 20 May 2024	GANs		Quiz 5
13 Monday, 27 May 2024	Energy-based models		
14 Monday, 3 June 2024	Evaluating generative models		Quiz 6
Monday, 17 June 2024			Project due

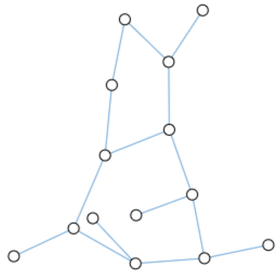
Previously in Probabilistic Graphical Models

- Belief propagation using a graph of clusters connected by separation sets
- Message passing between clusters about specific variables
- Computes probabilities (pseudo-marginals)
- Knowledge is encoded in the graph

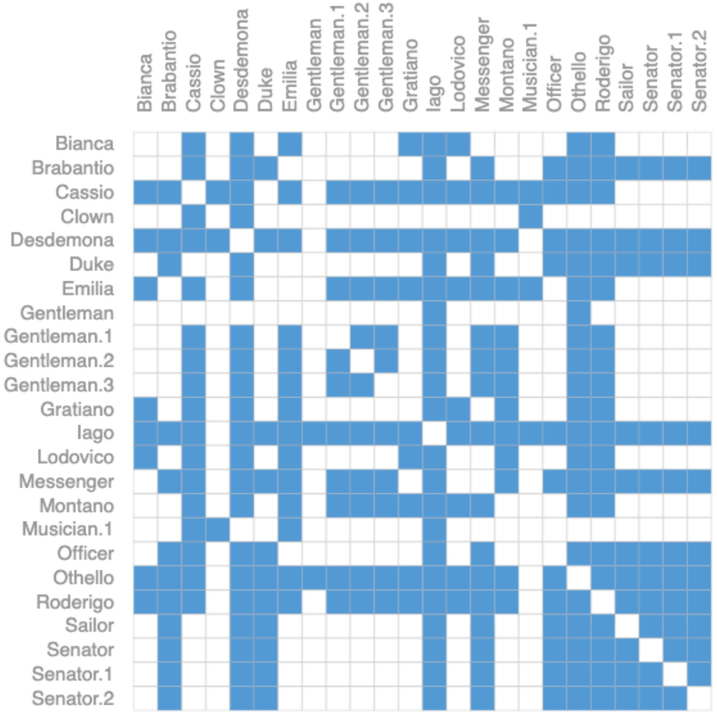
Graphs capture real-world relationships

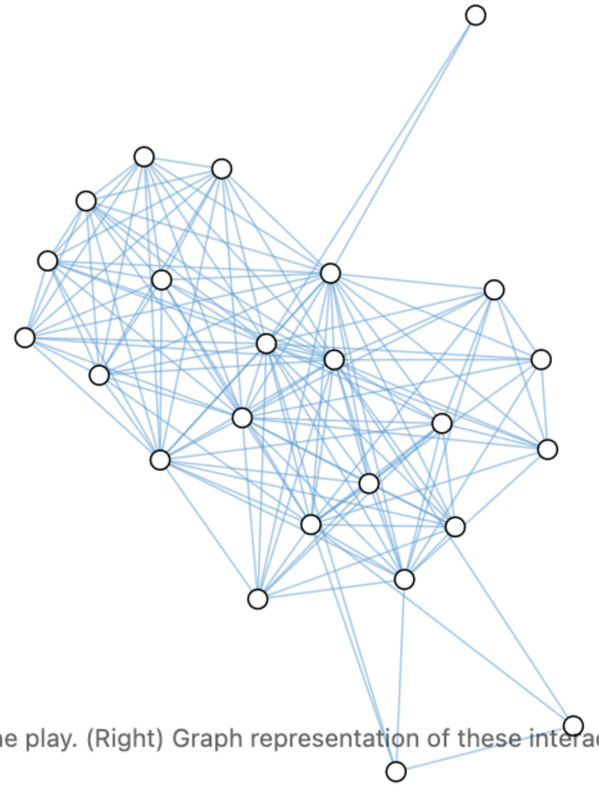






(Left) 3d representation of the Caffeine molecule (Center) Adjacency matrix of the bonds in the molecule (Right) Graph representation of the molecule.





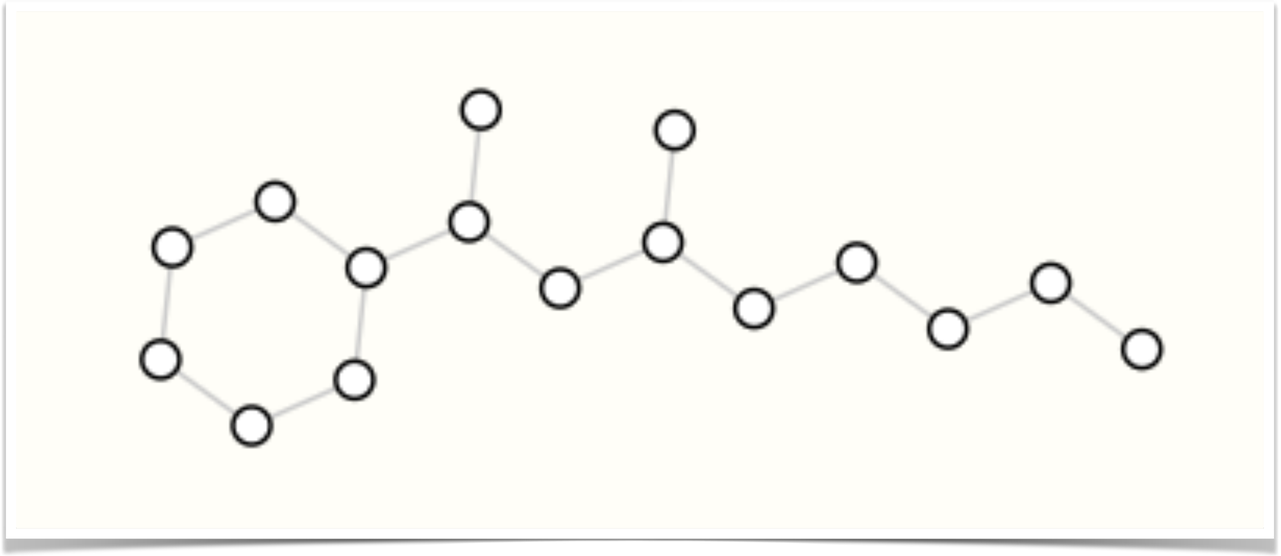
(Left) Image of a scene from the play "Othello". (Center) Adjacency matrix of the interaction between characters in the play. (Right) Graph representation of these interactions.

PGM (so far) used to organise knowledge

But what about making different predictions about graphs?

For most Machine learning models: need input data to be regularly organised

- e.g., same number of attributes per example; grid-like organisation (images)
- But what about this:



- Graph neural networks are neural networks that can work with graphs.
- They also use graphs (and message passing).

Graph problems

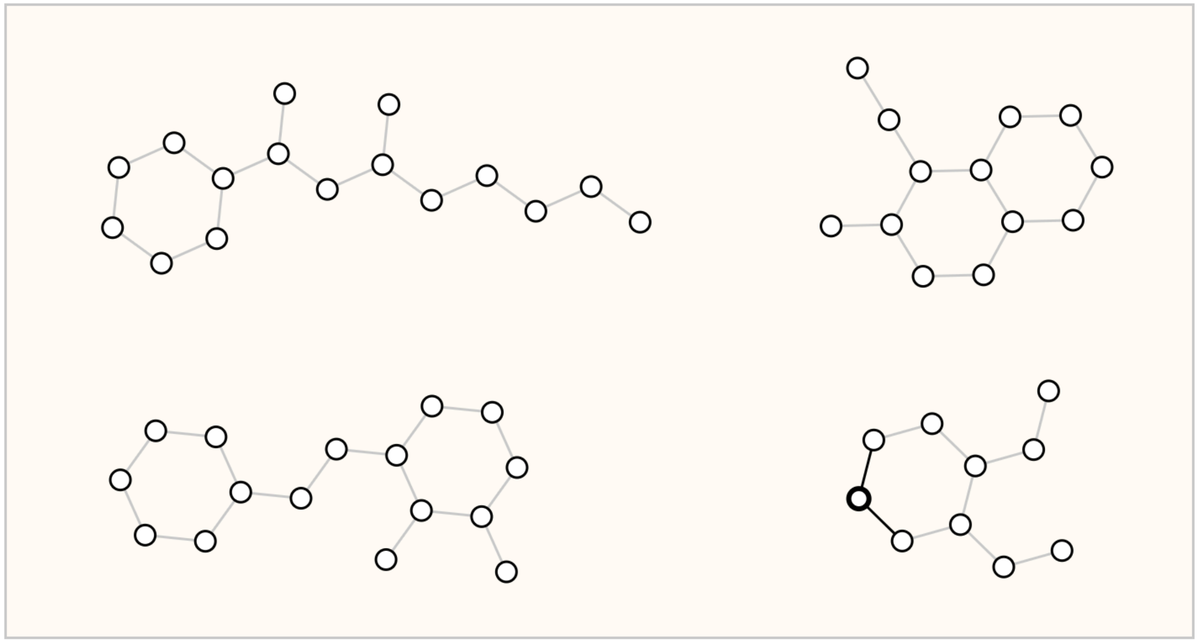
What kind of problems have graph structured data?

3 general types of prediction tasks on graphs: graph-level, node-level, and edge-level.

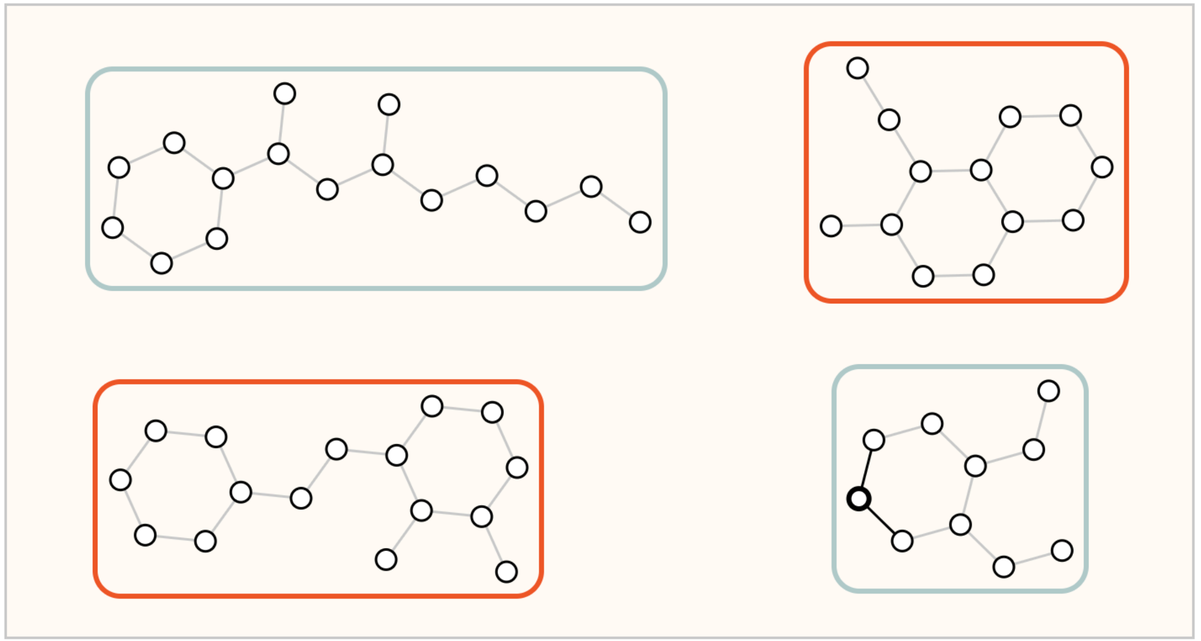
1. Graph-level task: we predict a single property for a whole graph
2. Node-level task: we predict some property for each node in a graph
3. Edge-level task: we want to predict the property or presence of edges in a graph

Graph-level tasks

Example



Input: graphs



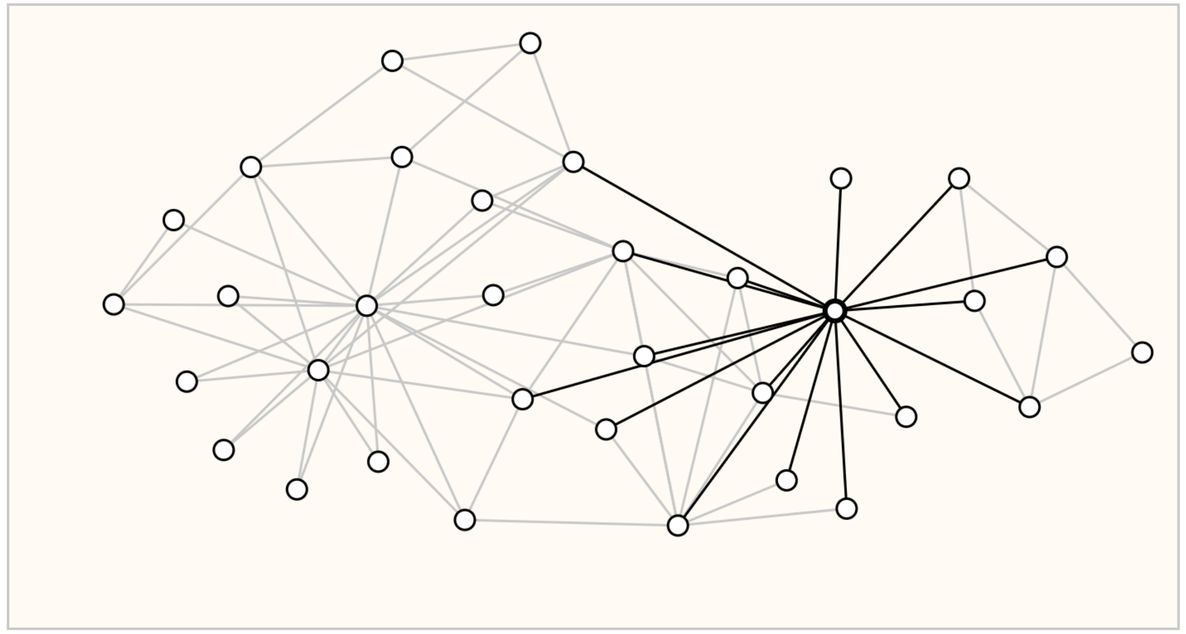
Output: labels for each graph, (e.g., "does the graph contain two rings?")

Similar to image classification tasks (Input: entire image, Output: “cat”, or “dog”, ...).
Other graph-level tasks: predict smell of molecule, ...

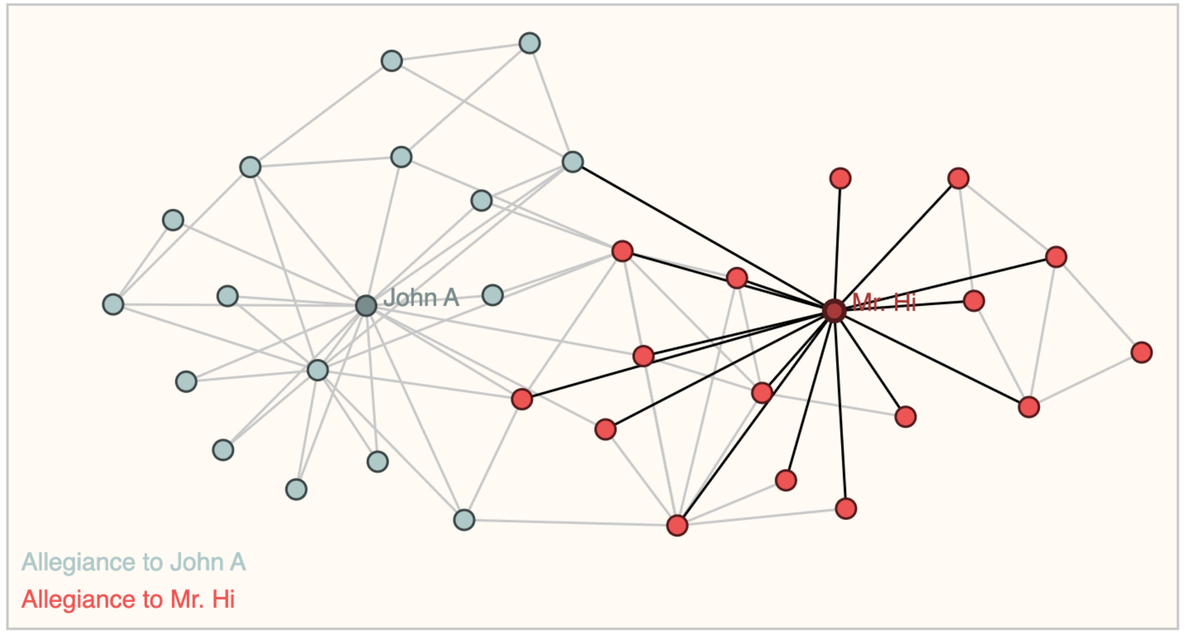
Node-level tasks

Zach's karate club example

Individual members have sworn allegiance the club president John A and a popular instructor Mr Hi. A conflict between the two led to the club dividing into two distinct groups. The problem is to identify the community structure of the karate club network based on the connections between its members.



→



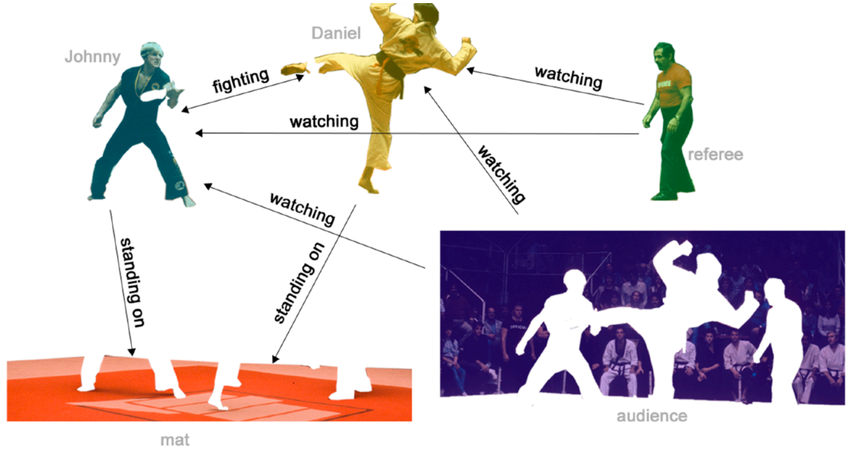
Input: graph with unlabled nodes

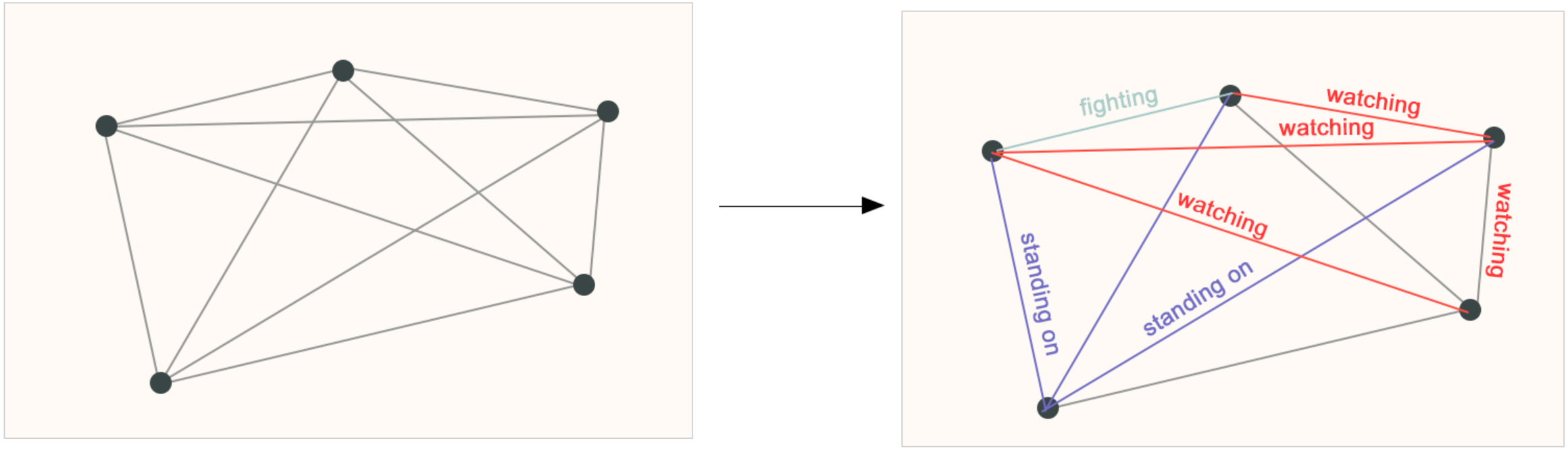
Output: graph node labels

Analogous to image segmentation, where we are trying to label the role of each pixel in an image.

Edge-level tasks

Scene understanding example





Input: fully connected graph, unlabeled edges

Output: labels for edges

We wish to predict which of these nodes share an edge or what the value of that edge is.

Challenges of using graphs

With neural networks

Representation: how to provide graph information to neural networks?
Neural networks "prefer" rectangular arrays as input.

Information from (or attached to) a graph: nodes, edges, global context, connectivity.

We can assign "features" to nodes, edges, context: e.g., N nodes, M features: $N \times M$ feature matrix.

Connectivity: graphs are matrices ($N \times N$)... but not ideal representation.

1. Often these matrices are very sparse.

   Not many connections, so a big matrix takes a lot of space for little information.

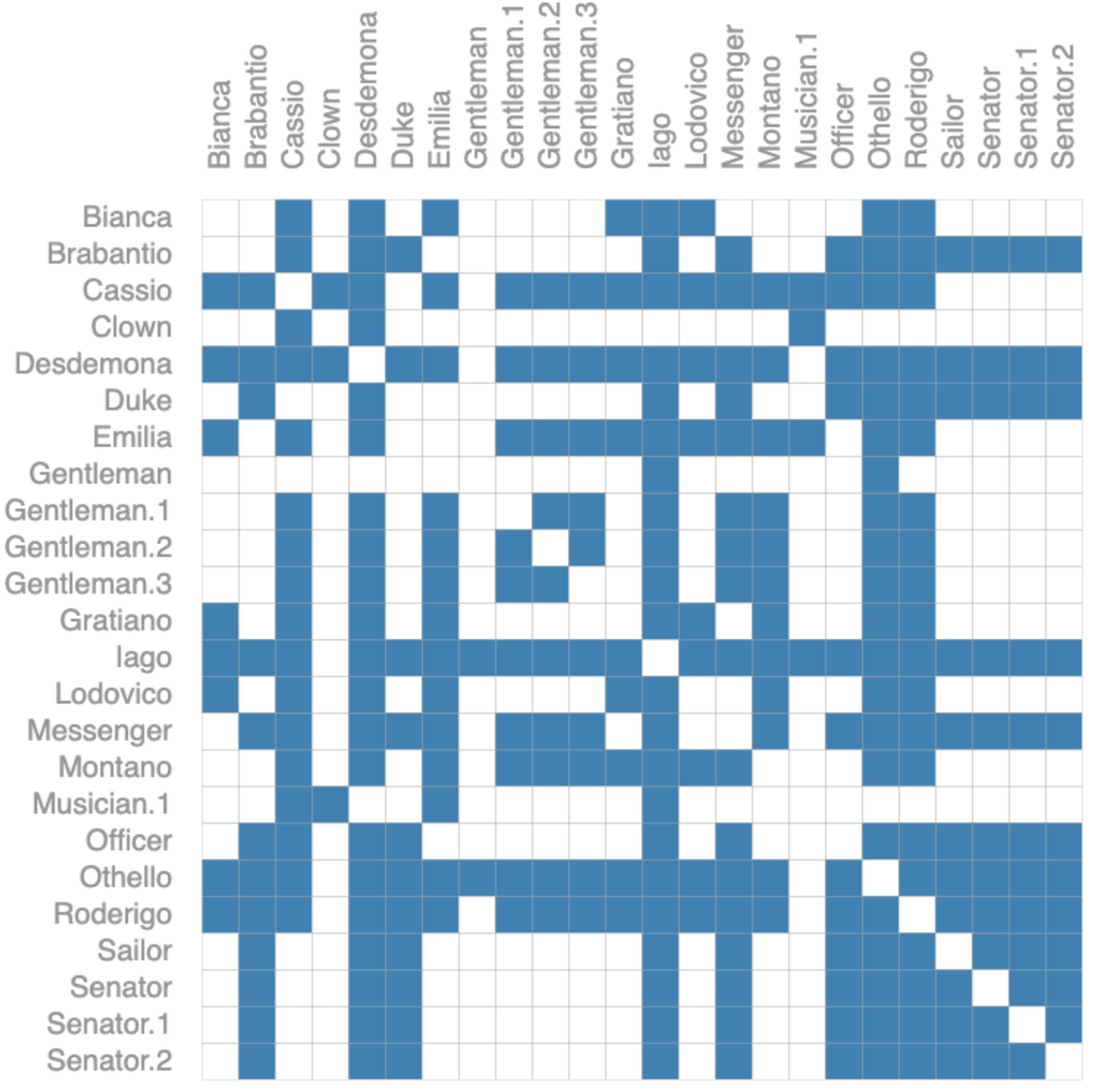
2. Same graph can be encoded in many ways.

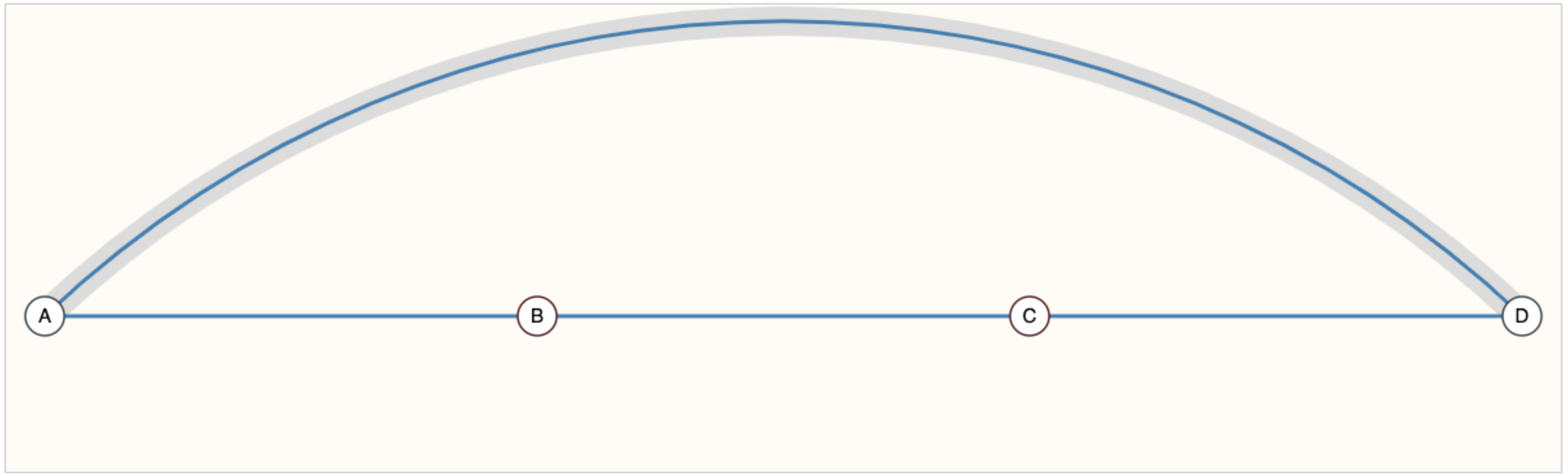
   Different inputs ⇝ different outputs (?)

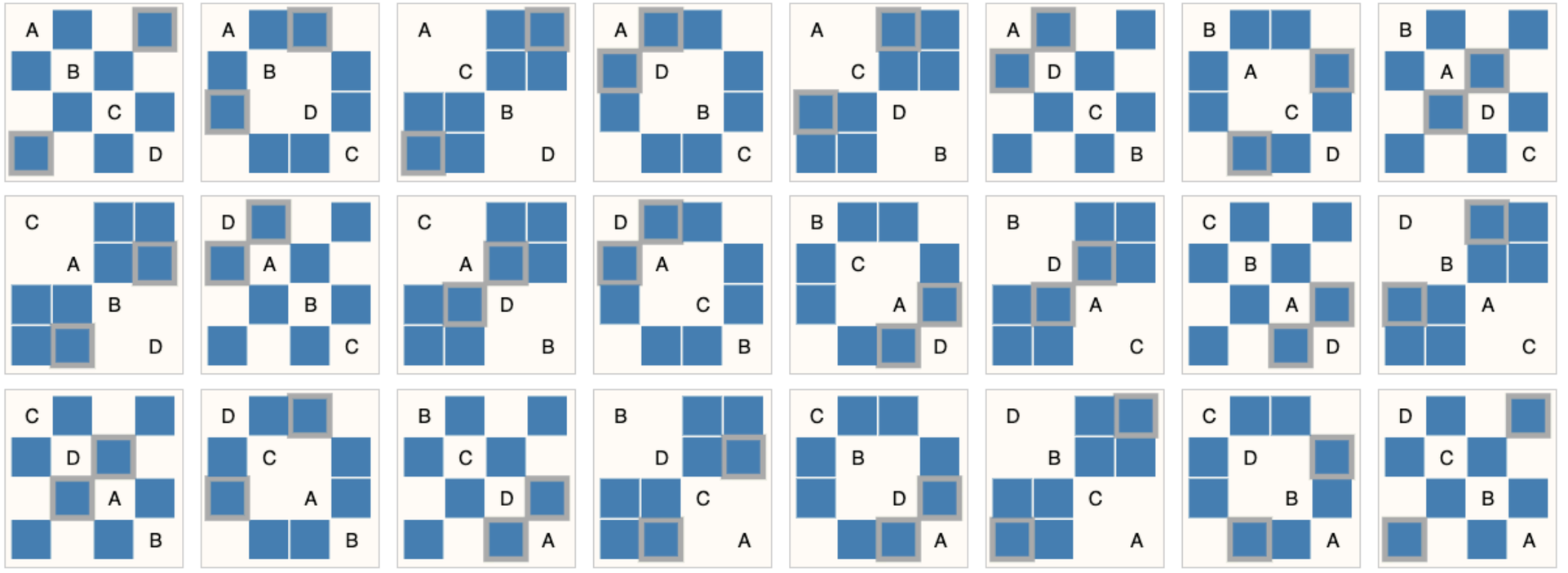
					Edges per node (degree)		
Dataset	**Domain**	**graphs**	**nodes**	**edges**	**min**	**mean**	**max**
karate club	Social network	1	34	78		4.5	17
qm9	Small molecules	134k	≤ 9	≤26	1	2	5
Cora	Citation network	1	23,166	91,500	1	7.8	379
Wikipedia links, English	Knowledge graph	1	12M	378M		62.24	1M

Summary statistics on graphs found in the real world. Numbers are dependent on featurization decisions. More useful statistics and graphs can be found in KONECT [14]


Gentleman.2
Sailor
Gratiano
Roderigo
Gentleman
Senator.2
Lodovico
Gentleman.3
Duke
Emilia
Officer
Desdemona
Brabantio
Clown
Gentleman.1
Senator
Senator.1
Bianca
Iago
Montano
Messenger
Othello
Musician.1
Cassio


Bianca
Brabantio
Cassio
Clown
Desdemona
Duke
Emilia
Gentleman
Gentleman.1
Gentleman.2
Gentleman.3
Gratiano
Iago
Lodovico
Messenger
Montano
Musician.1
Officer
Othello
Roderigo
Sailor
Senator
Senator.1
Senator.2



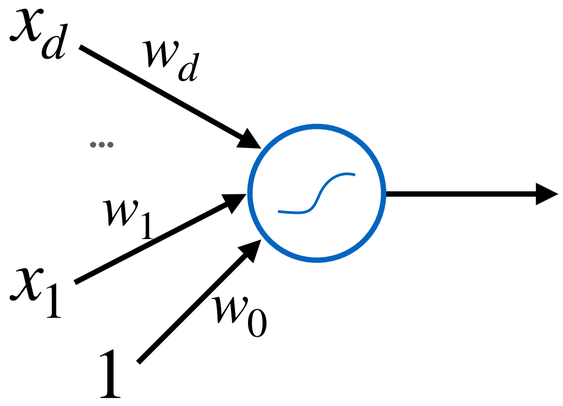


https://distill.pub/2021/gnn-intro/

Feedforward neural networks

Neural Networks

Artificial “neurons” as building blocks



An artificial neuron is inspired by neurons in a brain.
We can connect them to each other, and then they form a network.

Neuron net-input (“pre-activation”):

$$a(\mathbf{x}) = w_0 + \sum_{i=1}^{d} w_i x_i = \mathbf{w}^\top \mathbf{x}$$

$\mathbf{w}$ are the weights or parameters

w_0 is also called the bias

$\sigma(\cdot)$ is the activation function

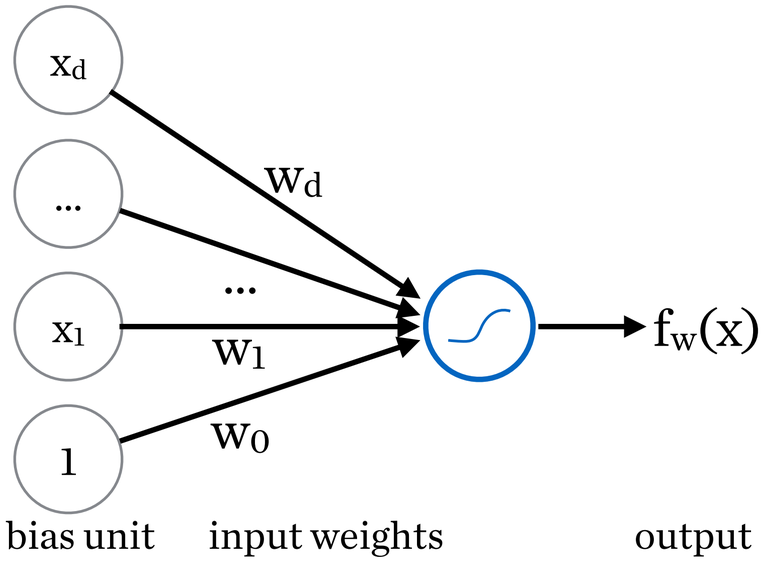
Neuron output (“post-activation”):

$$h(\mathbf{x}) = \sigma(a(\mathbf{x})) = \sigma(\mathbf{w}^\top \mathbf{x})$$

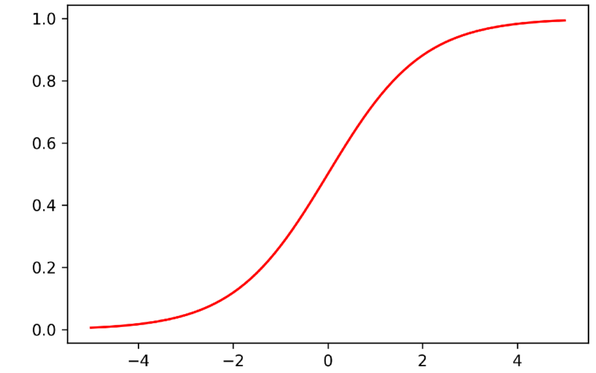
Inside a neural network

Activation functions

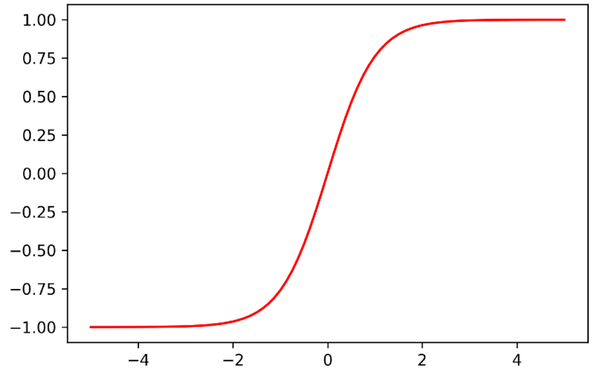
Different activation functions in modern neural networks



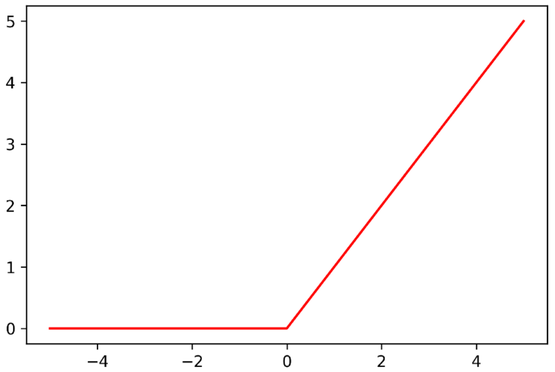
1. Logistic: $f_{\boldsymbol{w}}(\boldsymbol{x}) = \dfrac{1}{1 + e^{-\boldsymbol{w}^\top \boldsymbol{x}}}$



2. Hyperbolic tangent:
$f_{\boldsymbol{w}}(\boldsymbol{x}) = \tanh(\boldsymbol{w}^\top \boldsymbol{x})$



3. Rectified Linear Unit (ReLU):
$f_{\boldsymbol{w}}(\boldsymbol{x}) = \max(0, \boldsymbol{w}^\top \boldsymbol{x})$

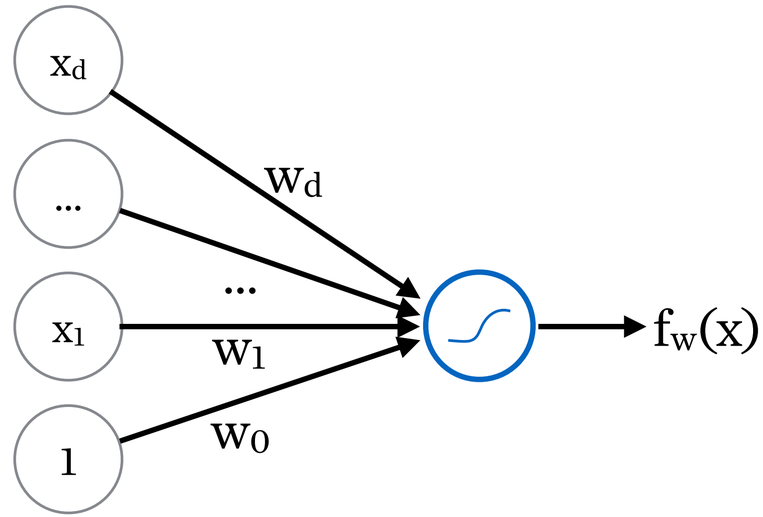


Inside a neural network

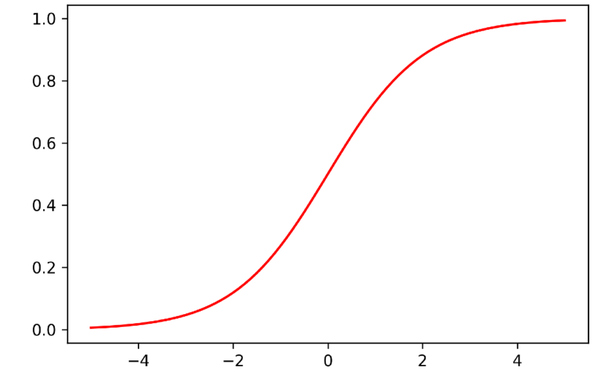
Activation functions

Different activation functions in modern neural networks

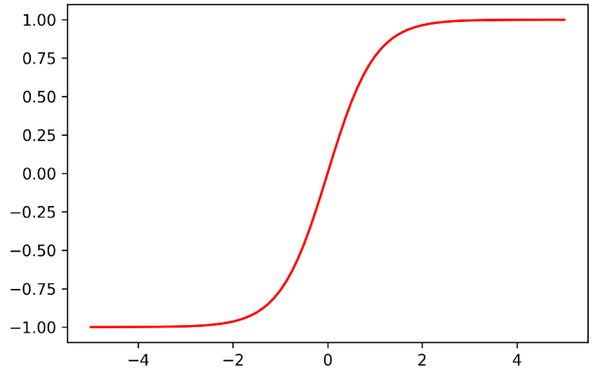
Instead of a linear function $\sigma(x) = x$



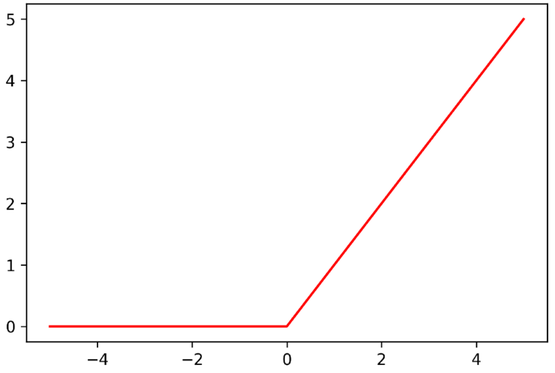
1. Logistic: $f_{\boldsymbol{w}}(\boldsymbol{x}) = \dfrac{1}{1 + e^{-\boldsymbol{w}^\top \boldsymbol{x}}}$



2. Hyperbolic tangent: $f_{\boldsymbol{w}}(\boldsymbol{x}) = \tanh(\boldsymbol{w}^\top \boldsymbol{x})$

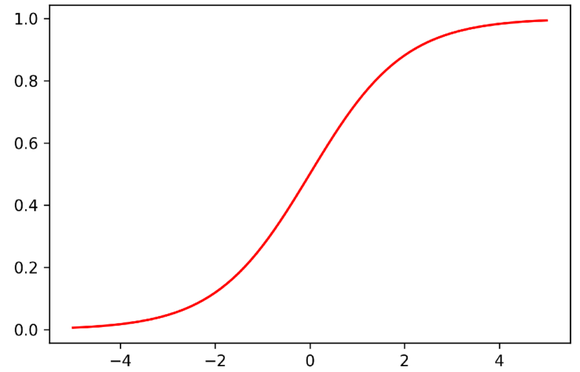


3. Rectified Linear Unit (ReLU): $f_{\boldsymbol{w}}(\boldsymbol{x}) = \max(0, \boldsymbol{w}^\top \boldsymbol{x})$



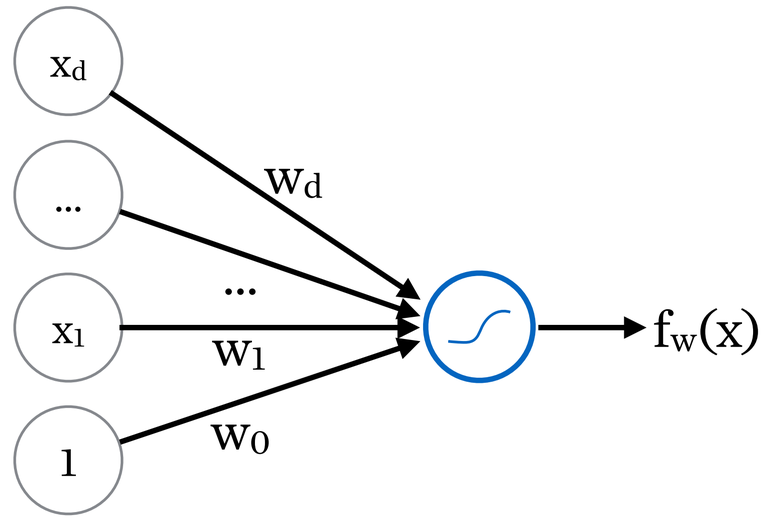
Inside a neural network

Activation functions

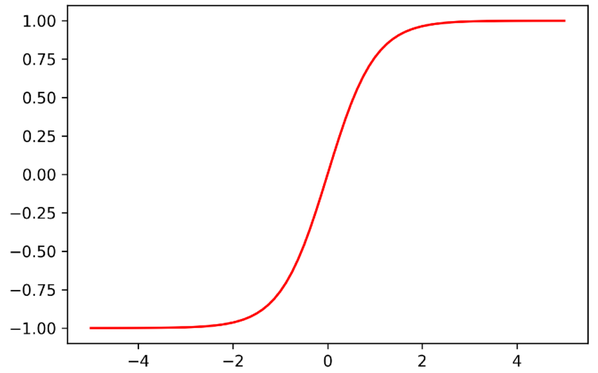


Different activation functions in modern neural networks

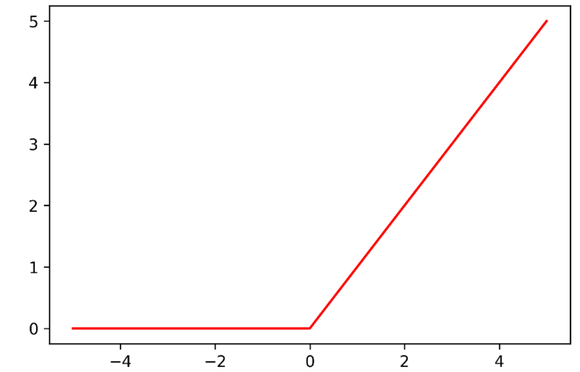
Instead of a linear function $\sigma(x) = x$



1. Logistic: $f_{\boldsymbol{w}}(\boldsymbol{x}) = \dfrac{1}{1 + e^{-\boldsymbol{w}^\top \boldsymbol{x}}}$



2. Hyperbolic tangent: $f_{\boldsymbol{w}}(\boldsymbol{x}) = \tanh(\boldsymbol{w}^\top \boldsymbol{x})$



3. Rectified Linear Unit (ReLU): $f_{\boldsymbol{w}}(\boldsymbol{x}) = \max(0, \boldsymbol{w}^\top \boldsymbol{x})$

Remember that $\boldsymbol{w}^\top \boldsymbol{x} = w_0 x_0 + \ldots + w_n x_n$: we compute a scalar value (a number), summing up the products of weights and inputs.

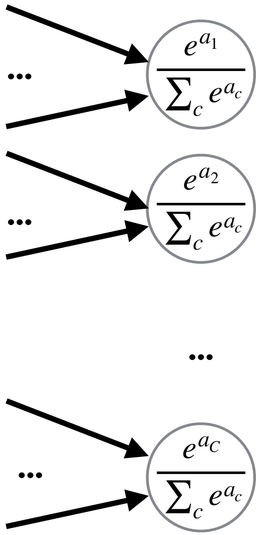
Softmax output activation

Multi-class classification

To calculate conditional probabilities at the output: $P(\text{output} = c \,|\, \mathbf{x})$

Softmax activation:

$$\text{softmax}(\mathbf{a}) = \sigma(\mathbf{a}) = \left[\frac{e^{a_1}}{\sum_c e^{a_c}}, \ldots, \frac{e^{a_C}}{\sum_c e^{a_c}} \right]^\top$$



Predicts probabilities for each class, sums to one.

Single hidden layer network

Hidden layer pre-activation:

$$\mathbf{a}(\mathbf{x}) = \mathbf{W}^{(0)}\mathbf{x}$$

$$\text{or: } \mathbf{a}(\mathbf{x})_i = \sum_j \mathbf{W}^{(0)}_{i,j}\mathbf{x}_j$$

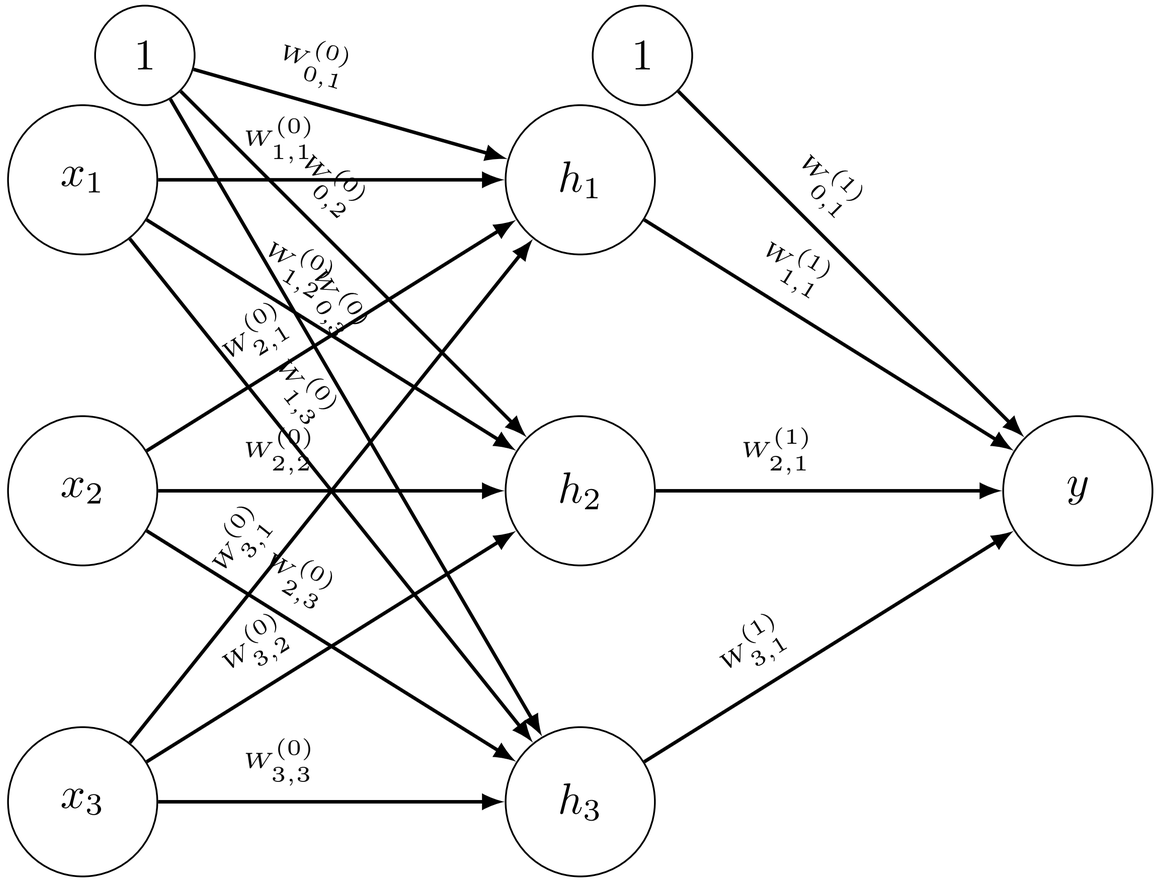
Hidden layer post-activation:

$$\mathbf{h}(\mathbf{x}) = \sigma(\mathbf{a}(\mathbf{x}))$$

Output layer activation:

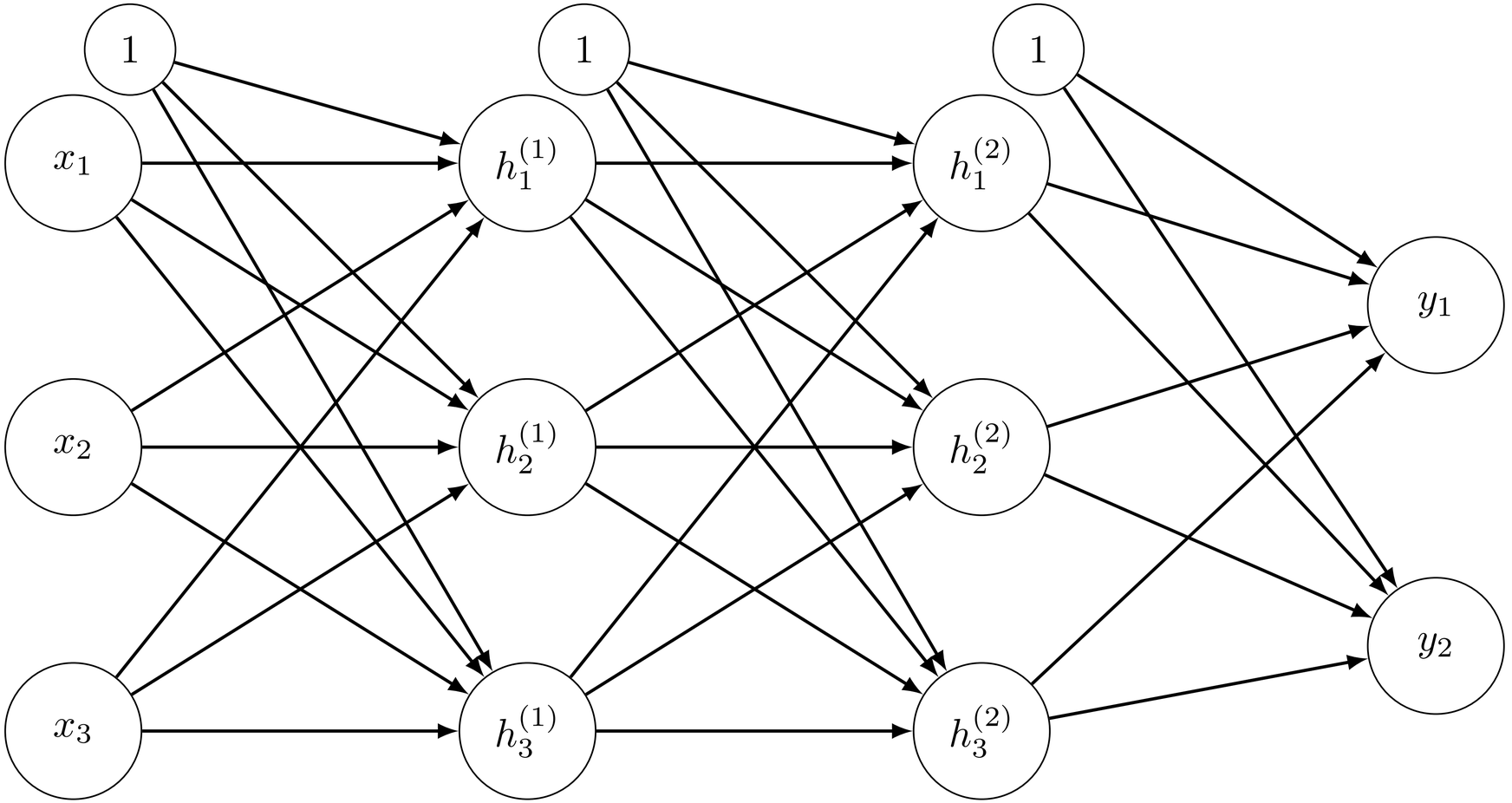
$$\mathbf{f}(\mathbf{x}) = o(\mathbf{W}^{(1)}\mathbf{h}(\mathbf{x}))$$

Output activation function



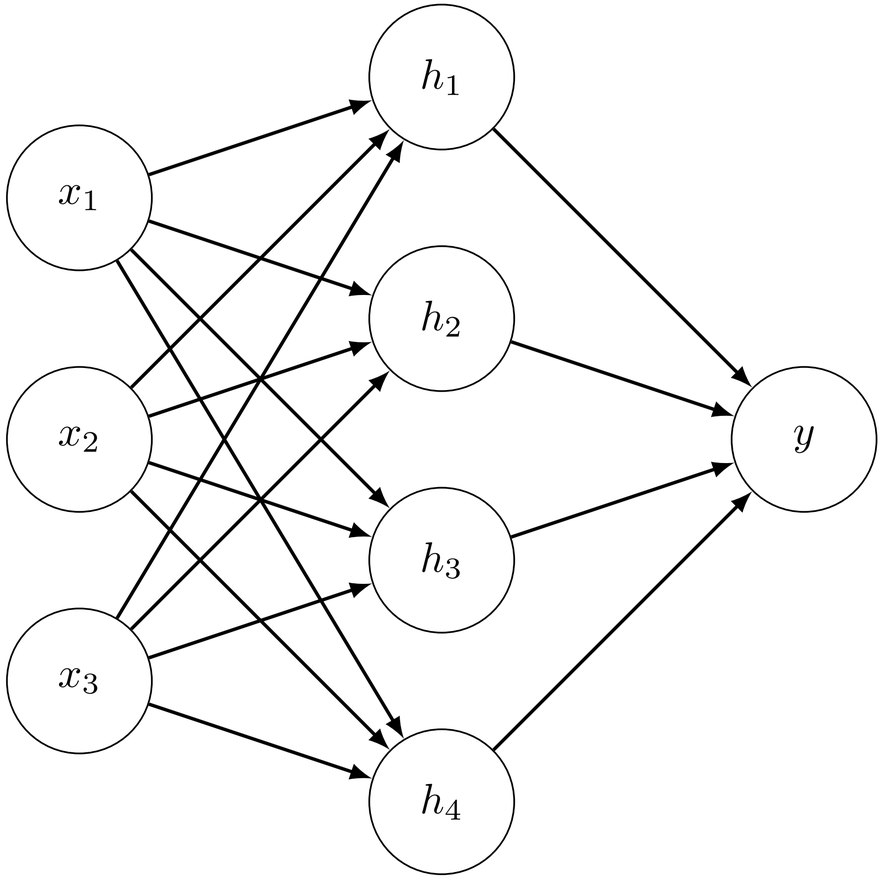
Multi-layer neural network

“Multi-layer perceptron” (MLP)



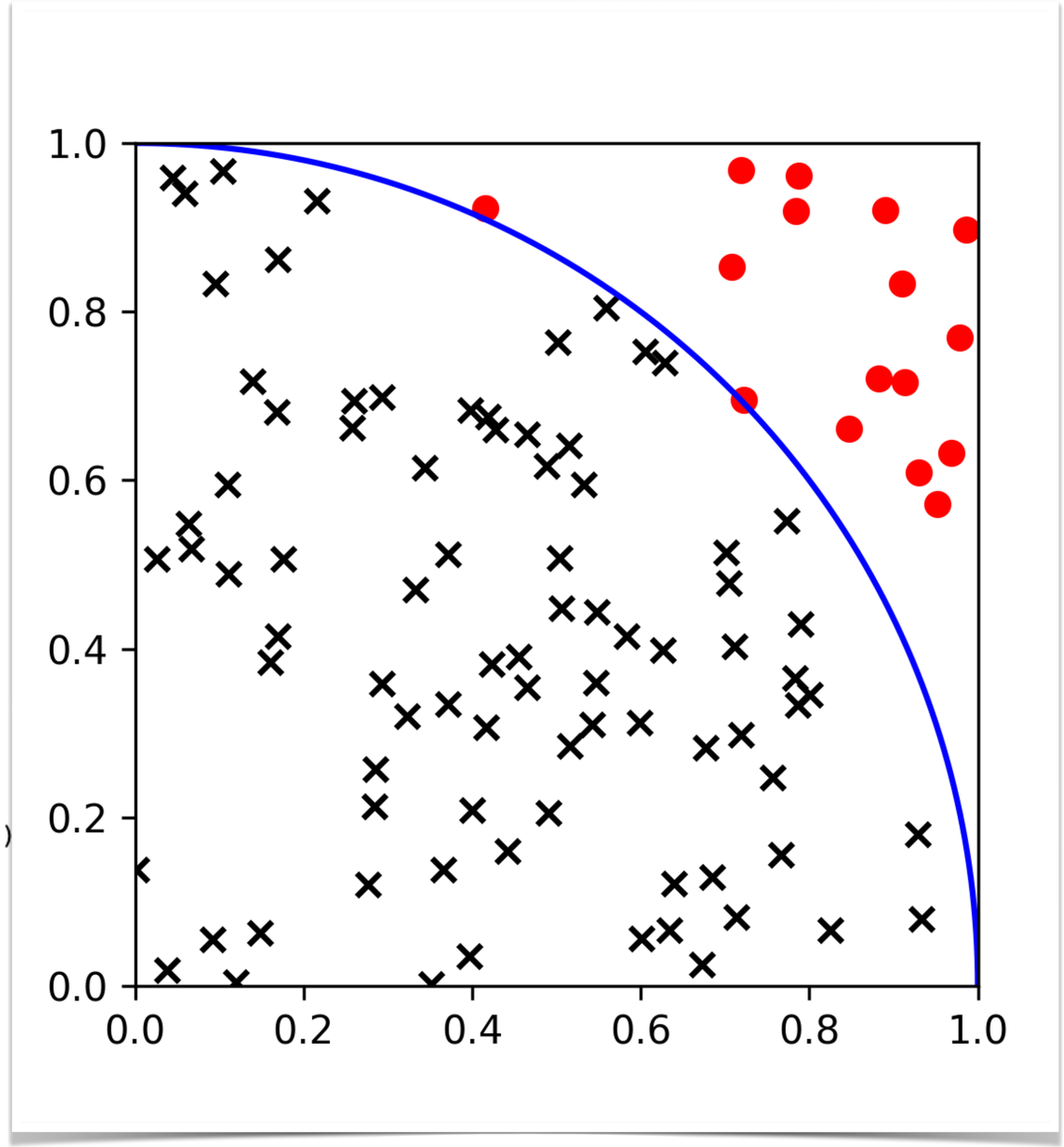
Example 1: setup

```python
import torch
import torch.nn as nn

class MyFeedForwardNetwork(nn.Module):
    def __init__(self):
        super(MyFeedForwardNetwork, self).__init__()
        self.hidden_layer = nn.Linear(3, 4)
        self.relu = nn.ReLU()
        self.output_layer = nn.Linear(4, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.hidden_layer(x)
        x = self.relu(x)
        x = self.output_layer(x)
        x = self.sigmoid(x)
        return x

# Create an instance of the network
model = MyFeedForwardNetwork()

# Example input tensor with 3 features
input_tensor = torch.tensor([[0.5, 0.3, 0.8]], dtype=torch.float32)

# Forward pass
output = model(input_tensor)

# The output is a probability value for the binary classification
print(output.item())
```



Example 1: data

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

# Number of samples
n_samples = 1000

# Generate random values for x1 and x2 in the range [0, 1)
x1 = np.random.rand(n_samples)
x2 = np.random.rand(n_samples)

# Generate random values for x3 in the range [0, 1), rounded to one decimal
x3 = np.round(np.random.rand(n_samples), 1)

# Calculate Euclidean distance from the origin (0, 0) for each (x1, x2) point
distances = np.sqrt(x1 ** 2 + x2 ** 2)

# Create labels: 1 if (x1, x2) is within a circle of radius x3, 0 otherwise
y = (distances <= x3).astype(int)

# Combine the three inputs into a single dataset
X = np.column_stack((x1, x2, x3))

# Split the data into training and testing
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

# Convert the datasets to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32).view(-1, 1)
X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
y_val_tensor = torch.tensor(y_val, dtype=torch.float32).view(-1, 1)

# Create DataLoaders for training and validation sets
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
val_dataset = TensorDataset(X_val_tensor, y_val_tensor)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
```

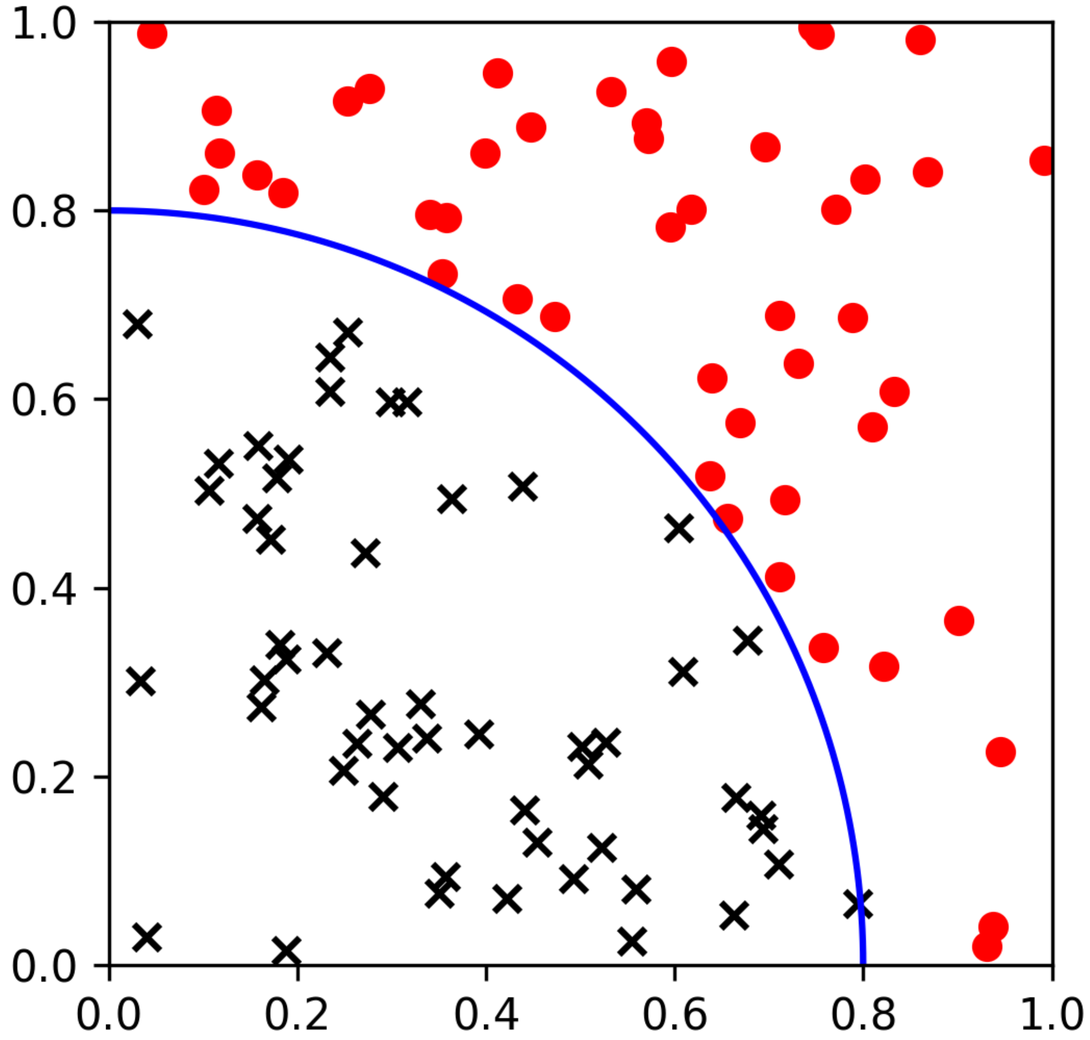


Example 1: training

```
import torch
import torch.nn as nn
import torch.optim as optim

# Initialize the network, loss function, and optimizer
model = MyFeedForwardNetwork()
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train the network
num_epochs = 200

for epoch in range(num_epochs):
    # Training loop
    model.train()
    train_loss = 0
    for batch_X, batch_y in train_loader:
        optimizer.zero_grad()
        predictions = model(batch_X)
        loss = criterion(predictions, batch_y)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    # Validation loop
    model.eval()
    val_loss = 0
    with torch.no_grad():
        for batch_X, batch_y in val_loader:
            predictions = model(batch_X)
            loss = criterion(predictions, batch_y)
            val_loss += loss.item()

    # Print the average loss for this epoch
    print(f"Epoch {epoch + 1}/{num_epochs}, Train Loss: "
          "{train_loss / len(train_loader):.4f}, Val Loss: "
          "{val_loss / len(val_loader):.4f}")
```

Example 1: evaluation

```python
import numpy as np
import torch

# Number of samples
n_samples = 100

# Generate random values for x1 and x2 in the range [0, 1)
x1 = np.random.rand(n_samples)
x2 = np.random.rand(n_samples)
# Set test radius
x3 = 0.8 * np.ones(n_samples)

# Calculate Euclidean distance from the origin (0, 0) for each (x1, x2) point
distances = np.sqrt(x1 ** 2 + x2 ** 2)

# Create labels: 1 if (x1, x2) is within a circle of radius x3, 0 otherwise
ytest = (distances <= x3).astype(int)

# Combine the three inputs into a single dataset
Xtest = np.column_stack((x1, x2, x3))

# Convert the test data to PyTorch tensors
X_test_tensor = torch.tensor(Xtest, dtype=torch.float32)
y_test_tensor = torch.tensor(ytest, dtype=torch.float32).view(-1, 1)

# Set the model to evaluation mode
model.eval()

# Run the model on the test data
with torch.no_grad():
    test_predictions = model(X_test_tensor)

# Convert the predictions to binary labels (0 or 1)
test_binary_predictions = (test_predictions > 0.5).float()

# Calculate the accuracy by comparing predicted labels with true labels
correct_predictions = (test_binary_predictions == y_test_tensor).float().sum()
test_accuracy = correct_predictions / y_test_tensor.size(0)

print(f"Test Accuracy: {test_accuracy:.4f}")
```



Graph Neural Networks

Adjacency lists

Representing sparse matrices

Describing connectivity:

- edge e_k between node n_i and n_j as a tuple (i, j) in kth entry of an adjacency list.

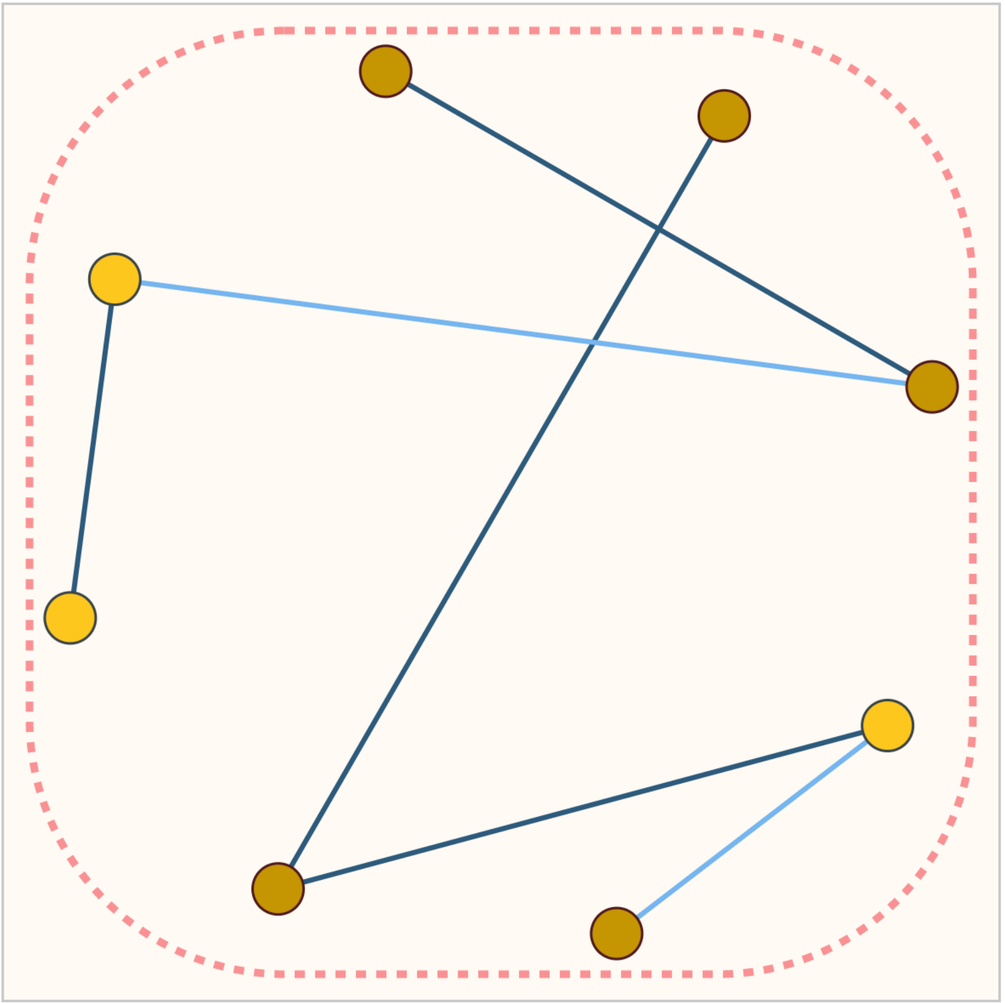
- not wasting space on disconnected nodes

Adjacency lists

Representing sparse matrices

Describing connectivity:

- edge e_k between node n_i and n_j as a tuple (i, j) in kth entry of an adjacency list.
- not wasting space on disconnected nodes



Nodes

```
[ 0 , 1 , 1 , 0 , 0 , 1 , 1 , 1 ]
```

Edges

```
[ 2 , 1 , 1 , 1 , 2 , 1 ]
```

Adjacency List

```
[ [1, 0] , [2, 0] , [4, 3] , [6, 2] ,
  [7, 4] , [7, 5] ]
```

Global

```
0
```

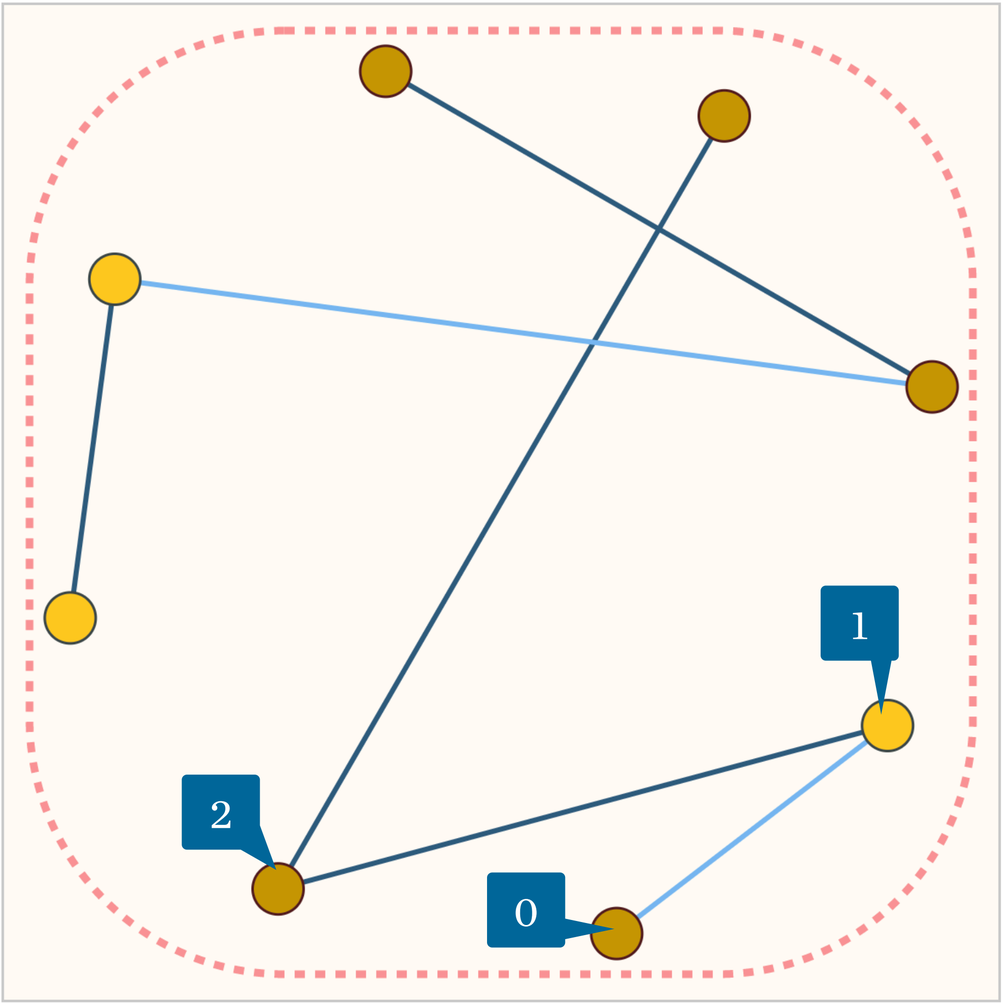
https://distill.pub/2021/gnn-intro/

Adjacency lists

Representing sparse matrices

Describing connectivity:

- edge e_k between node n_i and n_j as a tuple (i, j) in kth entry of an adjacency list.
- not wasting space on disconnected nodes



Nodes

```
[ 0 , 1 , 1 , 0 , 0 , 1 , 1 , 1 ]
```

Edges

```
[ 2 , 1 , 1 , 1 , 2 , 1 ]
```

Adjacency List

```
[ [1, 0] , [2, 0] , [4, 3] , [6, 2] ,
  [7, 4] , [7, 5] ]
```

Global

```
0
```

Here: scalar values per node/edge/global.

Most practical representations have *vectors* per graph attribute.

Instead of a node tensor of size $[n_{\text{nodes}}]$, you will then have a node tensor of size $[n_{\text{nodes}}, \text{node}_{\text{dim}}]$.

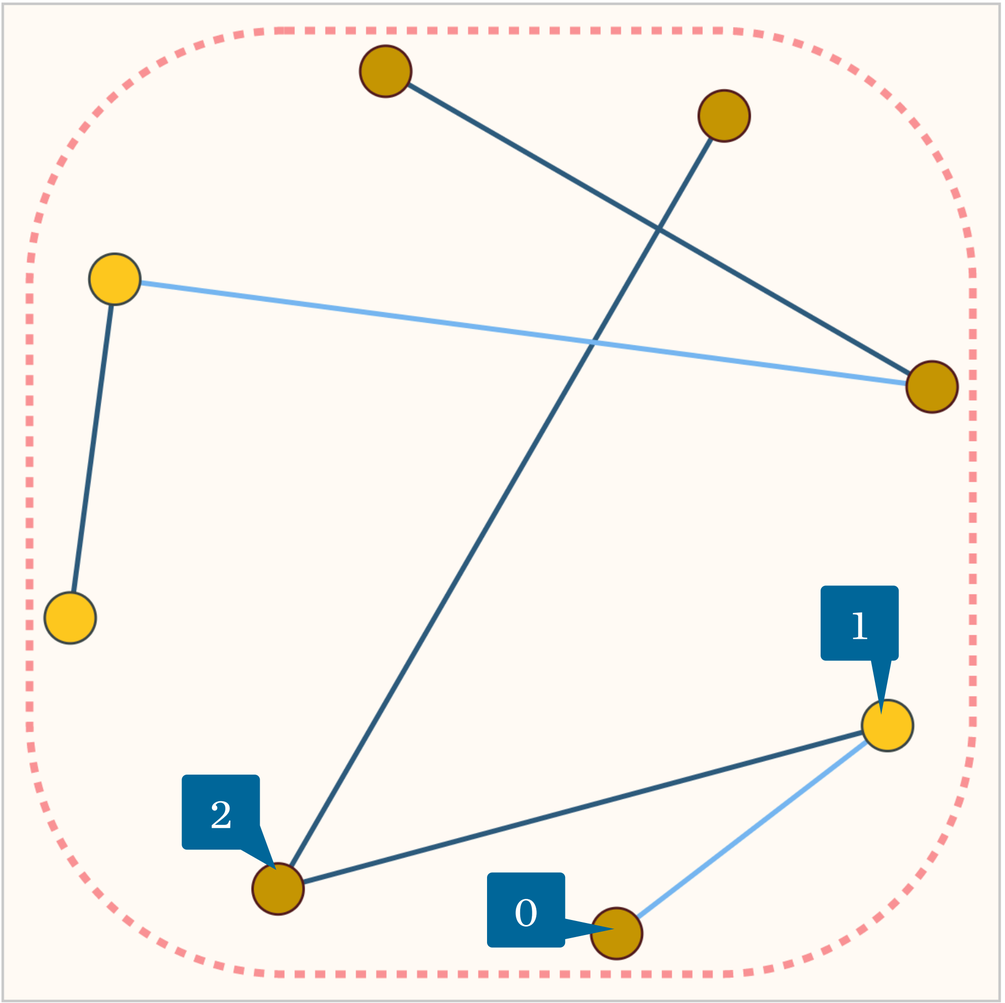
Same for other graph attributes.

Adjacency lists

Representing sparse matrices

Describing connectivity:

- edge e_k between node n_i and n_j as a tuple (i, j) in kth entry of an adjacency list.
- not wasting space on disconnected nodes



Nodes

```
[ 0 , 1 , 1 , 0 , 0 , 1 , 1 , 1 ]
```

Edges

```
[ 2 , 1 , 1 , 1 , 2 , 1 ]
```

Adjacency List

```
[ [1, 0] , [2, 0] , [4, 3] , [6, 2] ,
  [7, 4] , [7, 5] ]
```

Global

```
0
```

Here: scalar values per node/edge/global.

Most practical representations have *vectors* per graph attribute.

Instead of a node tensor of size $[n_{\text{nodes}}]$, you will then have a node tensor of size $[n_{\text{nodes}}, \text{node}_{\text{dim}}]$.

Same for other graph attributes.

nodes V : edges E : global U

Graph neural networks (GNNs)

A GNN is an optimisable transformation on all attributes of the graph (nodes, edges, global-context) that preserves graph symmetries (permutation invariances).

Our GNNs use a “graph-in, graph-out” architecture:

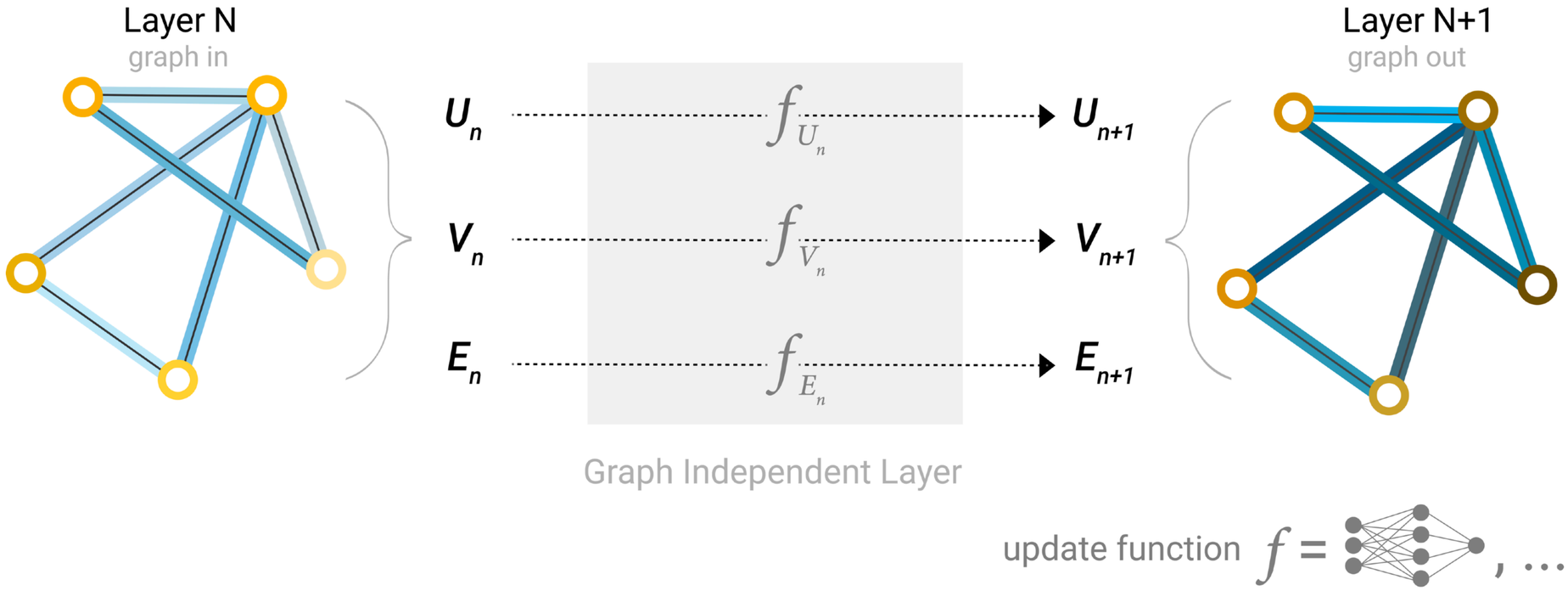
- Accept graph as an input
- Information attached to nodes, edges, global context: “embeddings”
- GNN then transform these embeddings. Connectivity remains unchanged.

GNN version 0

The simplest GNN

We learn to update embeddings for all graph attributes. We don't use connectivity yet.

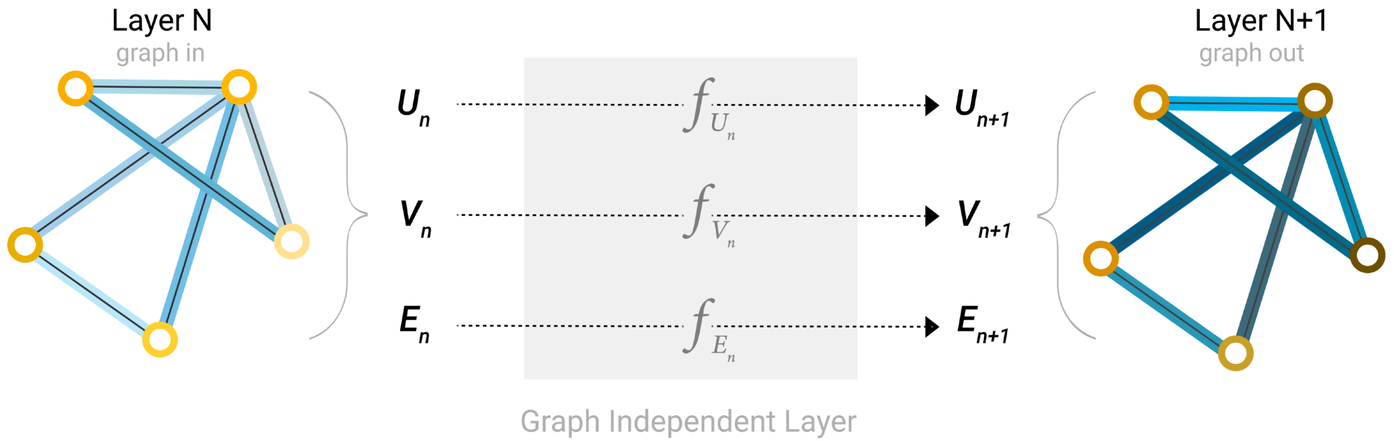
We use 3 separate multilayer perceptrons for each component: nodes, edges, global.



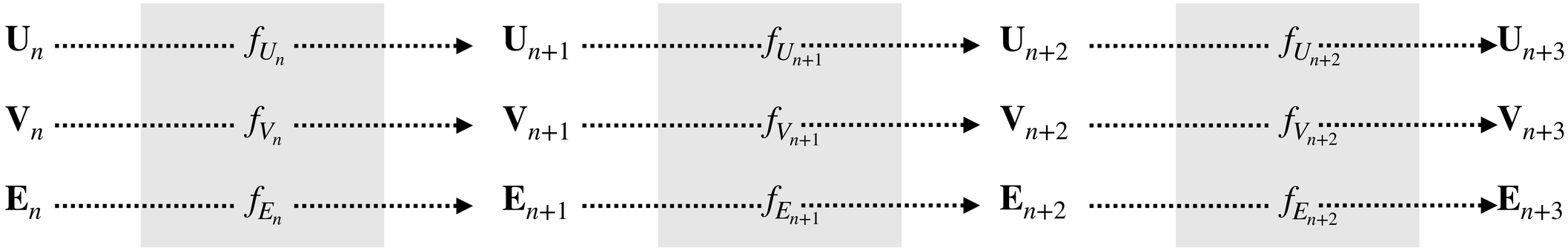
```python
import torch
import torch.nn as nn

class SimpleGNNLayer(nn.Module):
    def __init__(self, num_node_features, num_edge_features, num_global_features):
        super(SimpleGNNLayer, self).__init__()

        self.node_mlp = nn.Sequential(
            nn.Linear(num_node_features, num_node_features),
            nn.ReLU(),
            nn.Linear(num_node_features, num_node_features),
        )

        self.edge_mlp = nn.Sequential(
            nn.Linear(num_edge_features, num_edge_features),
            nn.ReLU(),
            nn.Linear(num_edge_features, num_edge_features),
        )

        self.global_mlp = nn.Sequential(
            nn.Linear(num_global_features, num_global_features),
            nn.ReLU(),
            nn.Linear(num_global_features, num_global_features),
        )

    def forward(self, node_features, edge_features, global_features):
        updated_node_features = self.node_mlp(node_features)
        updated_edge_features = self.edge_mlp(edge_features)
        updated_global_features = self.global_mlp(global_features)

        return updated_node_features, updated_edge_features, updated_global_features

# Define the layer
gnn_layer = SimpleGNNLayer(num_node_features=3, num_edge_features=2, num_global_features=4)

# Create some example input features
node_features = torch.randn(5, 3)  # 5 nodes, 3 features per node
edge_features = torch.randn(6, 2)  # 6 edges, 2 features per edge
global_features = torch.randn(1, 4)  # 1 global context, 4 features

# Run the GNN layer on the input features
updated_node_features, updated_edge_features, updated_global_features = gnn_layer(node_features, edge_features, global_features)

```

GNN version 0

Stacking layers



- our GNN does not update the connectivity of the input graph
- the output has the same number of feature vectors as the input graph
- we can stack multiple graph layers ("GNN blocks") to form more complex GNNs



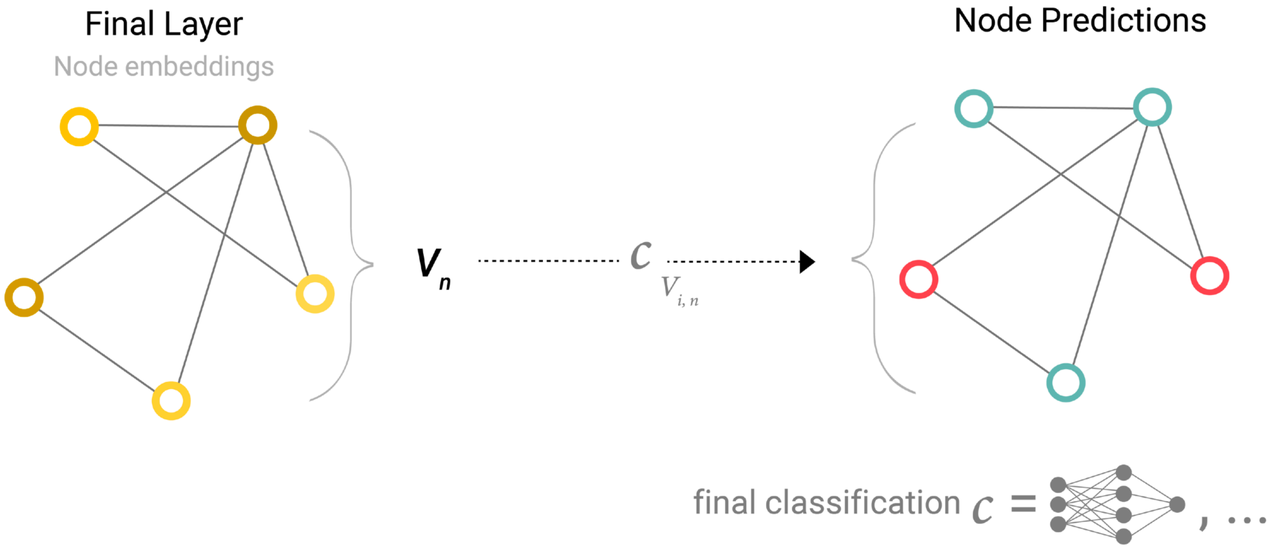
```python
import torch
import torch.nn as nn

class GNN(nn.Module):
    def __init__(self, num_node_features, num_edge_features, num_global_features):
        super(GNN, self).__init__()

        self.layer1 = SimpleGNNLayer(num_node_features, num_edge_features, num_global_features)
        self.layer2 = SimpleGNNLayer(num_node_features, num_edge_features, num_global_features)
        self.layer3 = SimpleGNNLayer(num_node_features, num_edge_features, num_global_features)

    def forward(self, node_features, edge_features, global_features):
        # Pass the input features through the first GNN layer
        node_features, edge_features, global_features = self.layer1(node_features, edge_features, global_features)

        # Pass the updated features through the second GNN layer
        node_features, edge_features, global_features = self.layer2(node_features, edge_features, global_features)

        # Pass the updated features through the third GNN layer
        node_features, edge_features, global_features = self.layer3(node_features, edge_features, global_features)

        return node_features, edge_features, global_features

# Create the GNN with 3 SimpleGNNLayer instances
gnn = GNN(num_node_features=3, num_edge_features=2, num_global_features=4)

# Create some example input features
node_features = torch.randn(5, 3)  # 5 nodes, 3 features per node
edge_features = torch.randn(6, 2)  # 6 edges, 2 features per edge
global_features = torch.randn(1, 4)  # 1 global context, 4 features

# Run the GNN on the input features
updated_node_features, updated_edge_features, updated_global_features = gnn(node_features, edge_features, global_features)
```

GNN outputs

Making predictions

For example: binary classification for each node

- Graph (embeddings) already contain information about nodes.
- We can apply a binary classifier for each node



Pooling

Node predictions, using information in edges

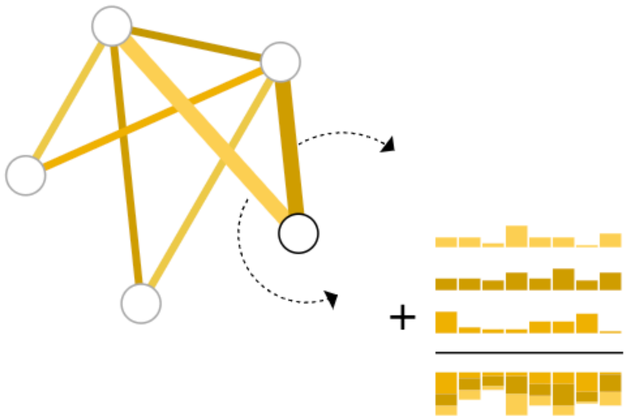
We need to collect information from edges, pass to nodes for prediction.

Pooling:

- For each item to be pooled, gather each of the embeddings, and concatenate into a matrix
- Aggregate the gathered embeddings, e.g., sum them up.

The pooling operation is represented by the letter ρ (“rho”).

To say we gather information from edges to nodes,
we write $\rho_{E_n \to V_n}$.



Aggregate information
from adjacent edges

Pooling

Node predictions, using information in edges

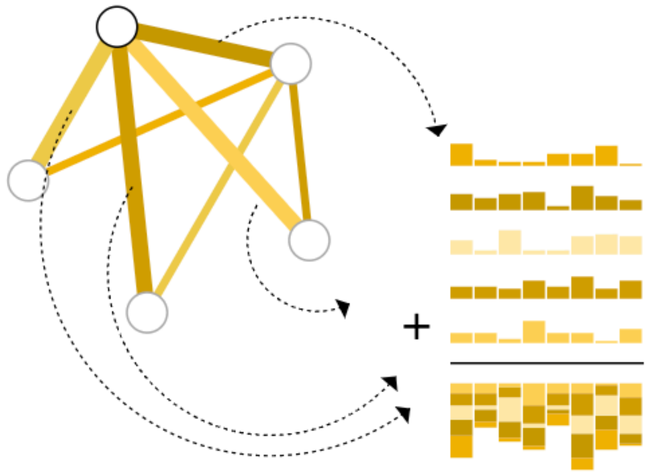
We need to collect information from edges, pass to nodes for prediction.

Pooling:

- For each item to be pooled, gather each of the embeddings, and concatenate into a matrix
- Aggregate the gathered embeddings, e.g., sum them up. We use $g(\cdot)$ for the aggregation function.

The pooling operation is represented by the letter ρ (“rho”).

To say we gather and pool information from edges to nodes, we write $\rho_{E_n \to V_n}$.

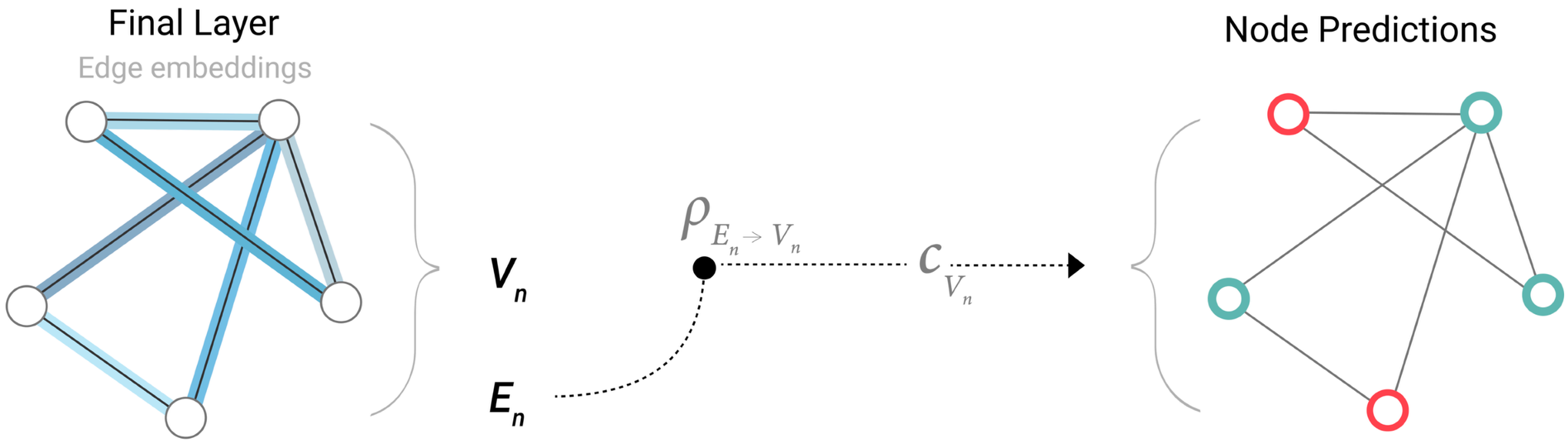


Aggregate information from adjacent edges

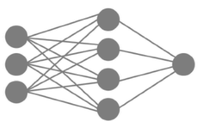
Pooling and classification

For node classification from edges

If we only have edge-level features, and are trying to predict binary node information, we can use pooling to route (or pass) information to where it needs to go.

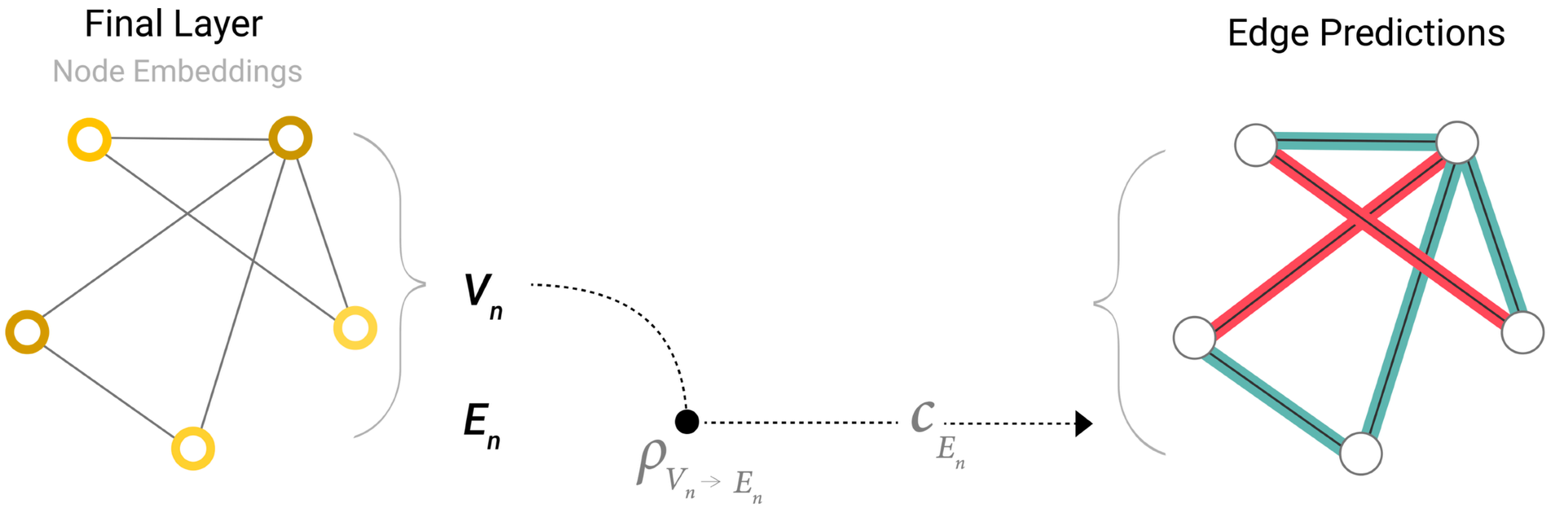


pooling function ρ

final classification $c =$  , ...

Pooling and classification

Predicting (binary classification) on edges from nodes

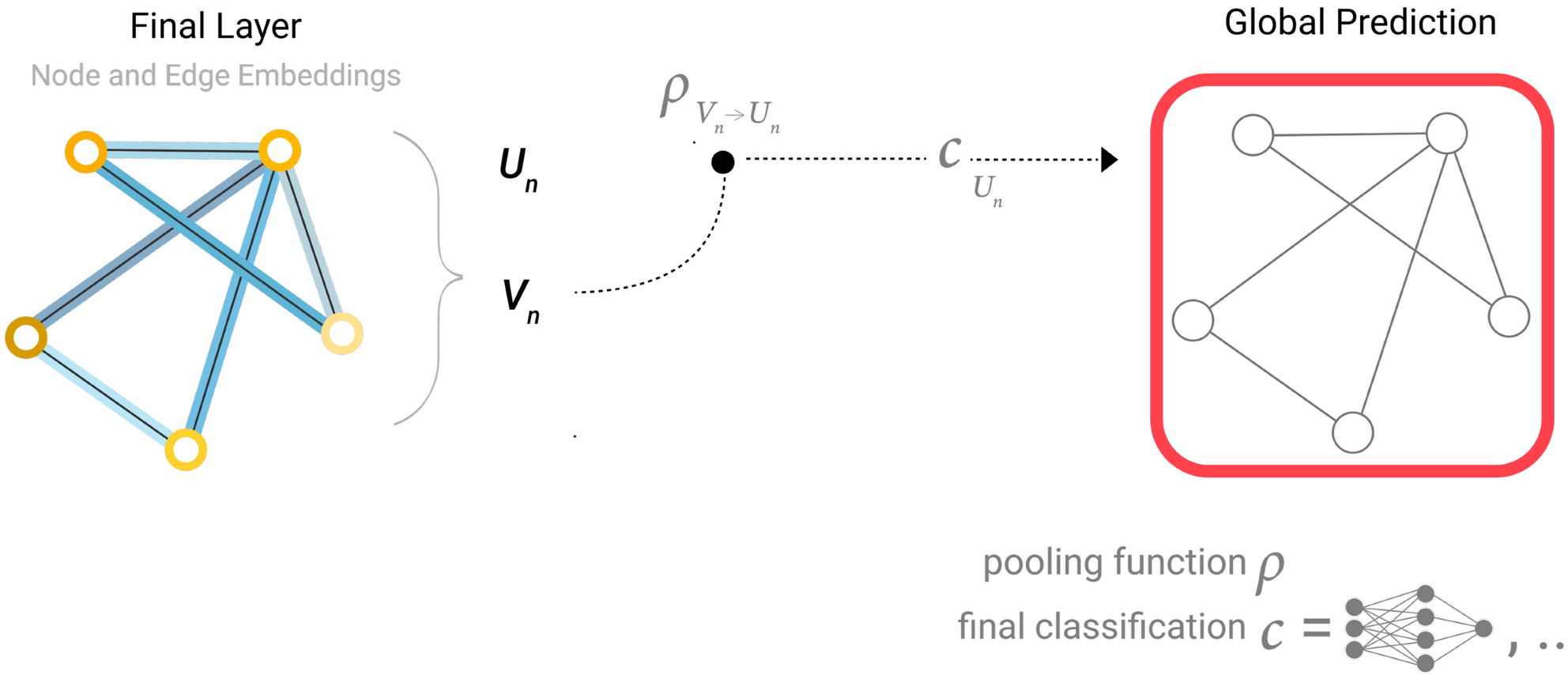


pooling function ρ

final classification c = , ...

Pooling and classification

Predict a (binary) global property

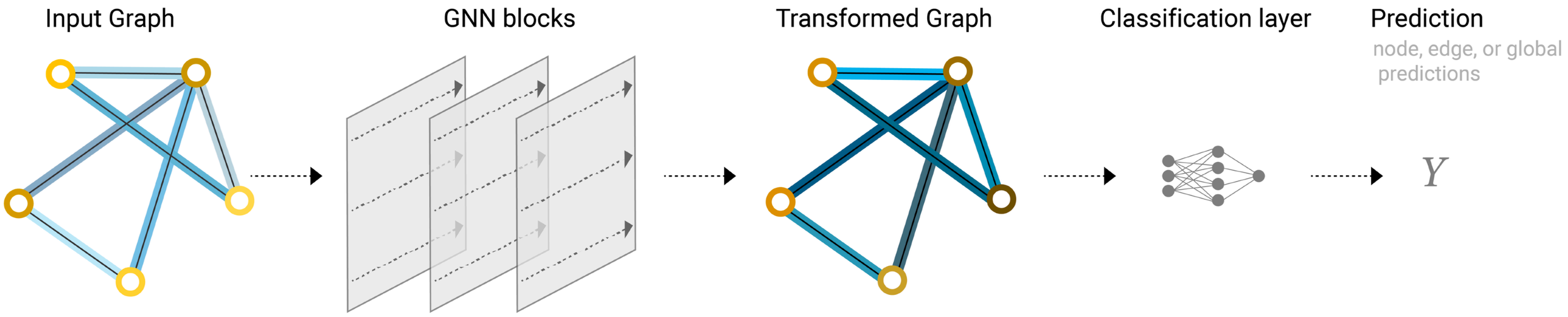


This is a common scenario for predicting molecular properties. For example, we have atomic information, connectivity and we would like to know the toxicity of a molecule (toxic/not toxic), or if it has a particular odor (rose/not rose).

GNN version 0

End-to-end prediction

The classification model can easily be replaced with any differentiable model, or adapted to multi-class classification using a generalised linear model.



```python
    def forward(self, node_features, edge_index):
        # Aggregate neighbour features
#         neighbour_features = torch.zeros_like(node_features)
        neighbour_features = node_features.clone()
        for src, dst in edge_index.t():
            neighbour_features[dst] += node_features[src]
        # Pass the node features through the node MLP
```

```
In : data.edge_index
Out:
tensor([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  1,
          1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  3,
          3,  3,  3,  3,  3,  4,  4,  4,  5,  5,  5,  5,  6,  6,  6,  6,  7,  7,
          7,  7,  8,  8,  8,  8,  8,  9,  9, 10, 10, 10, 11, 12, 12, 13, 13, 13,
         13, 13, 14, 14, 15, 15, 16, 16, 17, 17, 18, 18, 19, 19, 19, 20, 20, 21,
         21, 22, 22, 23, 23, 23, 23, 23, 24, 24, 24, 25, 25, 25, 26, 26, 27, 27,
         27, 27, 28, 28, 28, 29, 29, 29, 29, 30, 30, 30, 30, 31, 31, 31, 31, 31,
         31, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 33, 33, 33, 33, 33,
         33, 33, 33, 33, 33, 33, 33, 33, 33, 33, 33, 33],
        [ 1,  2,  3,  4,  5,  6,  7,  8, 10, 11, 12, 13, 17, 19, 21, 31,  0,  2,
          3,  7, 13, 17, 19, 21, 30,  0,  1,  3,  7,  8,  9, 13, 27, 28, 32,  0,
          1,  2,  7, 12, 13,  0,  6, 10,  0,  6, 10, 16,  0,  4,  5, 16,  0,  1,
          2,  3,  0,  2, 30, 32, 33,  2, 33,  0,  4,  5,  0,  0,  3,  0,  1,  2,
          3, 33, 32, 33, 32, 33,  5,  6,  0,  1, 32, 33,  0,  1, 33, 32, 33,  0,
          1, 32, 33, 25, 27, 29, 32, 33, 25, 27, 31, 23, 24, 31, 29, 33,  2, 23,
         24, 33,  2, 31, 33, 23, 26, 32, 33,  1,  8, 32, 33,  0, 24, 25, 28, 32,
         33,  2,  8, 14, 15, 18, 20, 22, 23, 29, 30, 31, 33,  8,  9, 13, 14, 15,
         18, 19, 20, 22, 23, 26, 27, 28, 29, 30, 31, 32]])
```

Summary so far

- We can build a simple GNN model.
- We can make binary predictions by passing information between different parts of the graph.
- Pooling will serve as a building block for constructing more sophisticated GNN models. If we have new graph attributes, we just have to define how to pass information from one attribute to another.
- Within each layer, so far, we're not using connectivity of the graph *at all*.
- Each node is processed independently, as is each edge, as well as the global context. We only use connectivity when pooling information for prediction.

Message passing in GNN

Passing messages between parts of the graph

So far, embeddings at the next layer are independent of graph connectivity.

We could use pooling within each GNN block, to make better predictions.

This can be achieved using message passing, similar to message passing in PGMs.

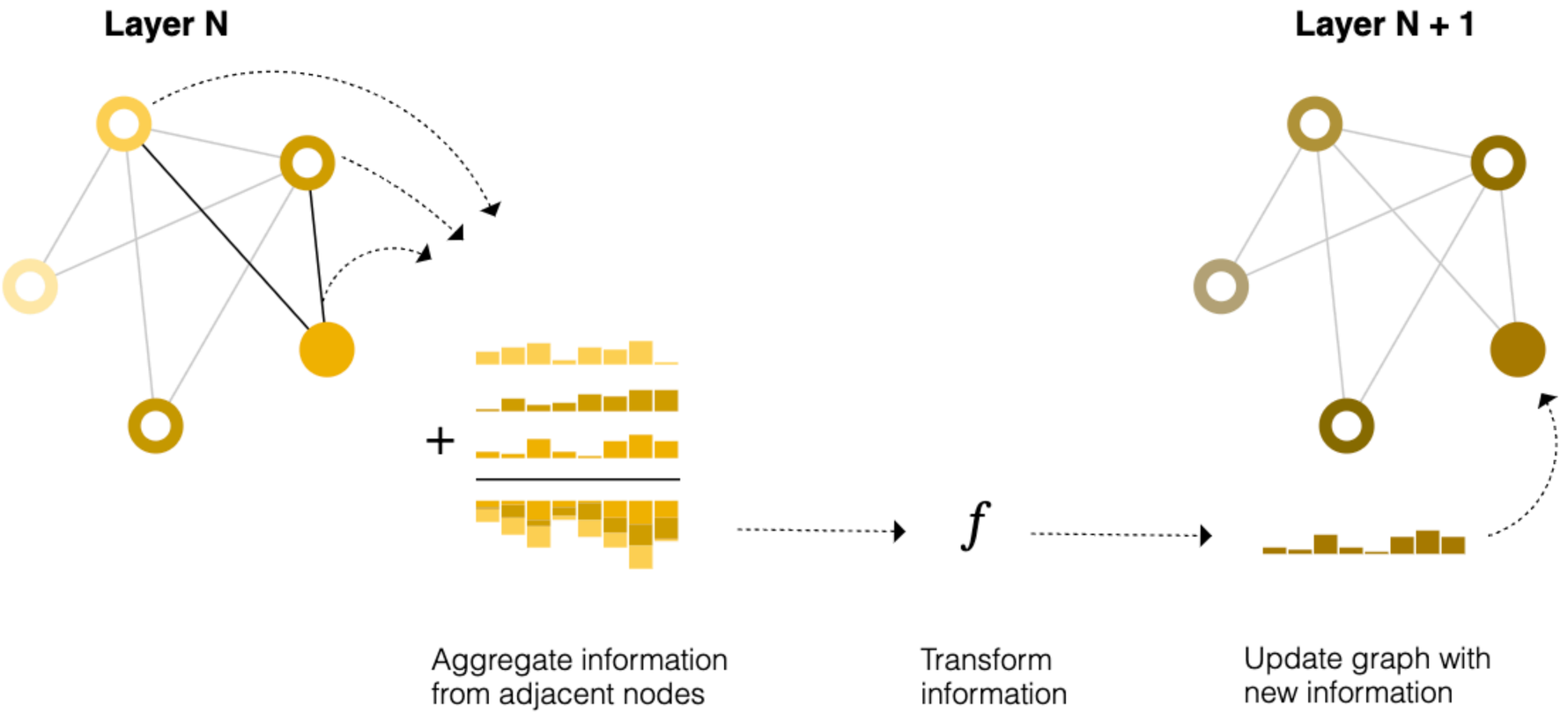
In GNNs, message passing works in 3 steps:

1. For each node in the graph, gather all the neighbouring node embeddings (or “messages”).
2. Aggregate all messages via an aggregate function (for example, sum them up).
3. All pooled messages are passed through an *update function*, usually a learned neural network.

Just as pooling can be applied to either nodes or edges, message passing can occur between either nodes or edges.

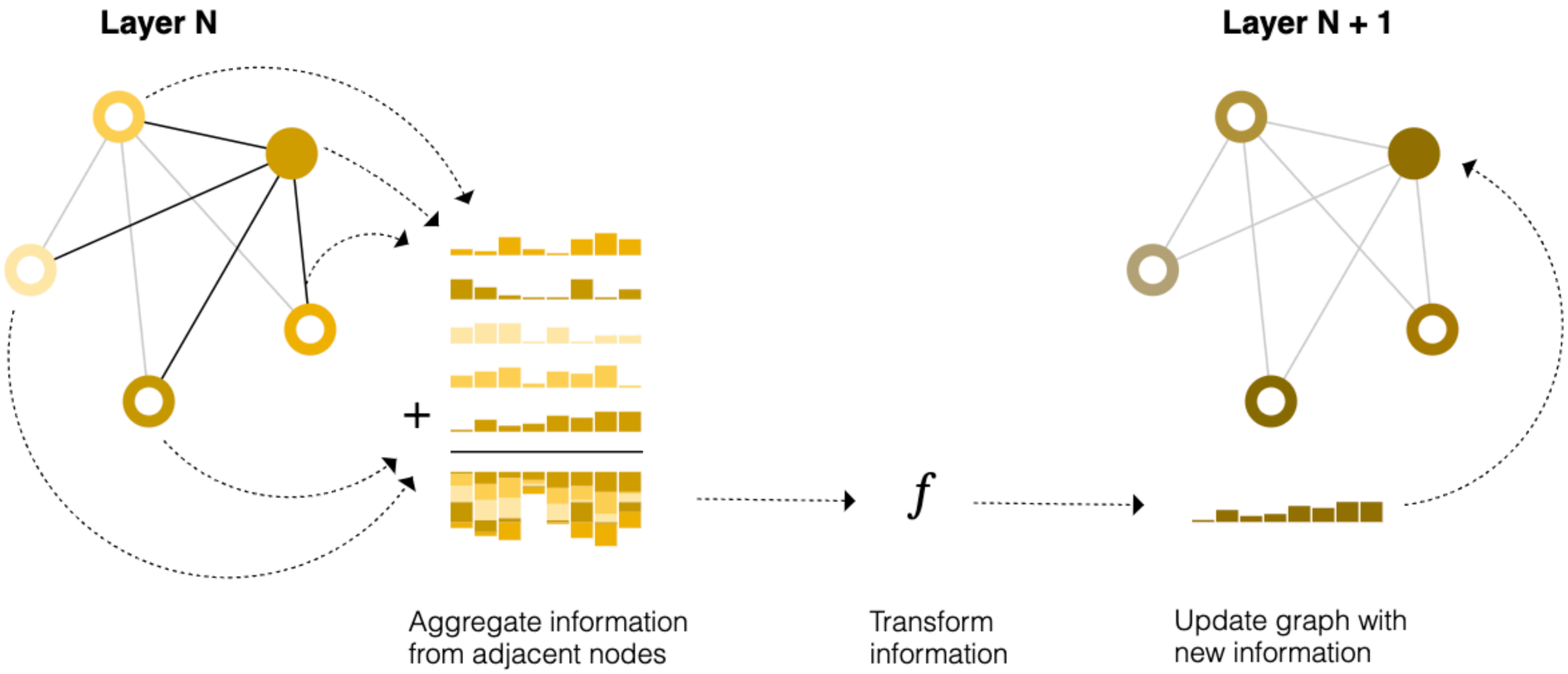
First message passing example

Making use of the graph connectivity



First message passing example

Making use of the graph connectivity



Message passing

Like convolution

Message passing is similar to convolution in CNNs:

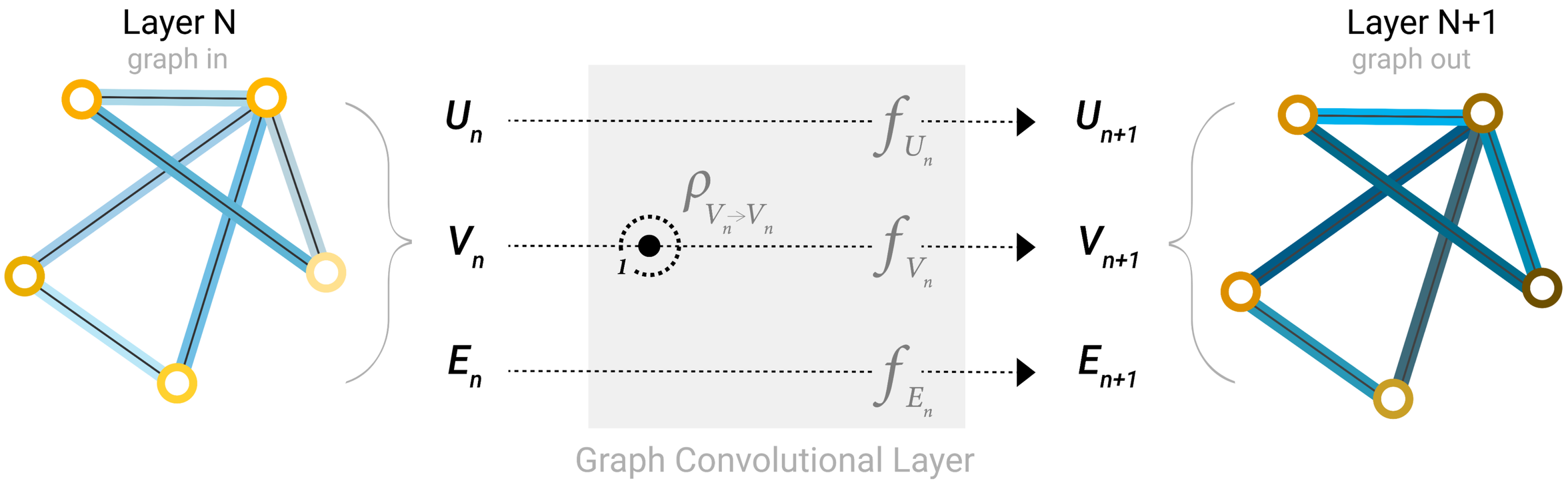
- Both operations aggregate and process information of an element's neighbours to update the element's value

- In graphs (GNNs), element is a node, and in images (CNNs), the element is a pixel

- In GNNs, number of neighbours in a graph can be variable.
  In an image, each pixel has a set number of neighbouring elements.

By stacking message passing GNN layers together, a node can eventually incorporate information from across the entire graph: after three layers, a node has information about the nodes three steps away from it.

Message passing in GNNs so far



update function $f =$, ...

pooling function ρ

Schematic for a GCN architecture, which updates node representations of a graph by pooling neighbouring nodes at a distance of one degree.

Learning edge representations

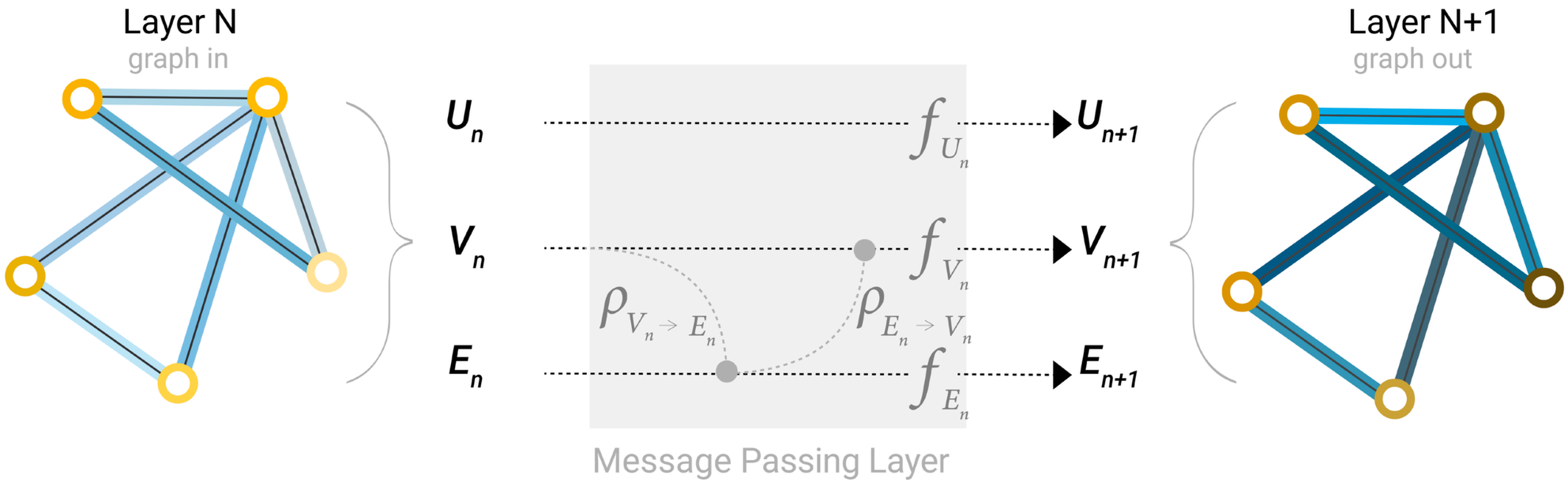
Our dataset does not always contain all types of information (node, edge, and global).

Already used pooling to route information from edges to nodes – in the final step.

We can also share information between nodes and edges within layer: message passing .

Issue: node and edge information may not be the same size (e.g., # of attributes).

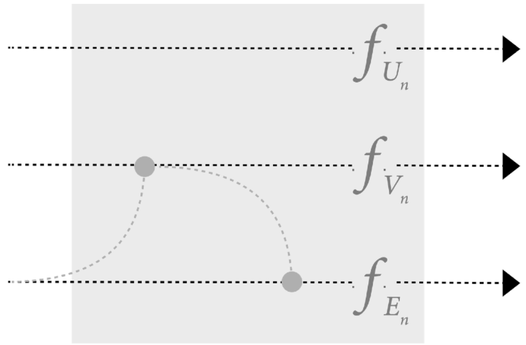
Can learn a linear mapping (e.g., from edges to nodes, vice versa), or concatenate.



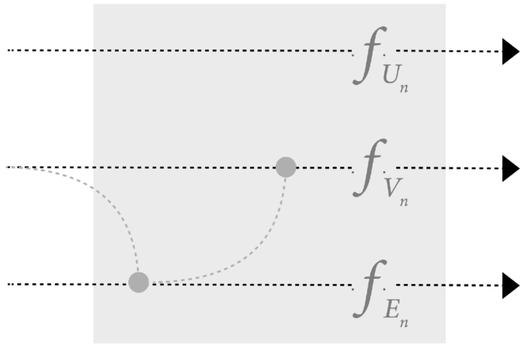
Design decisions

Which one comes first?

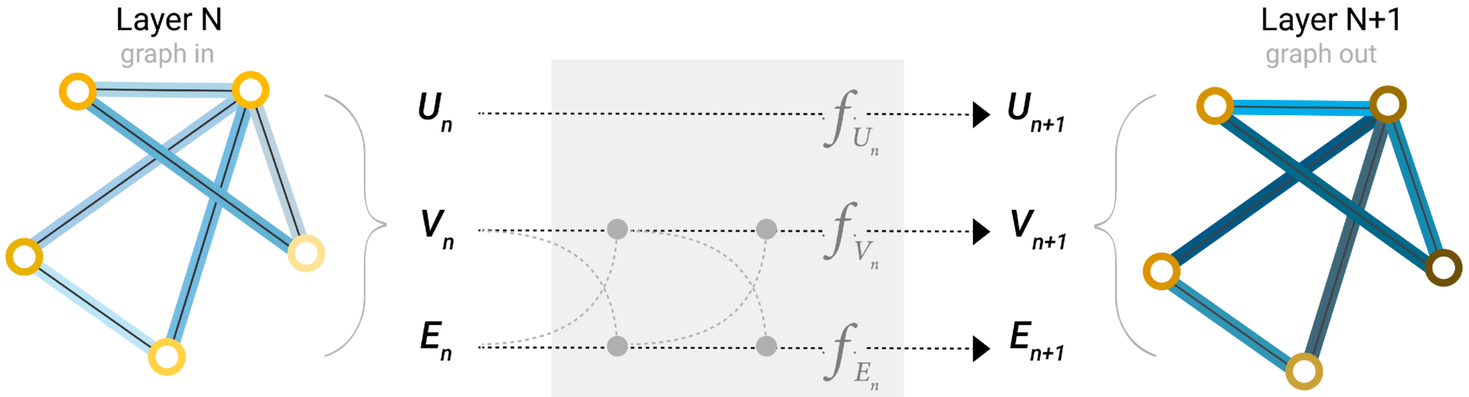
Which graph attributes to update, and in which order? It depends ... (open question)



Node Then Edge Learning



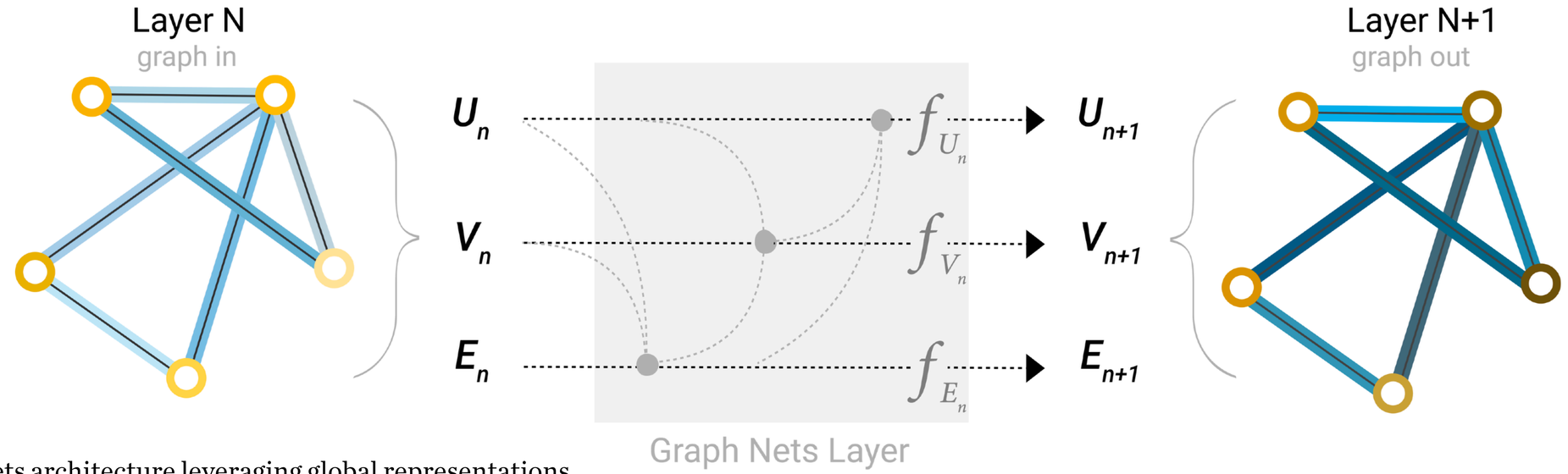
Edge Then Node Learning



Weave Layer

Adding global representations

- In large graphs, message passing would take many steps (layers of the GNN)
- k layers: information propagates k steps
- One solution: pass information to *all* nodes (not great for large graphs: expensive) possible for smaller graphs.
- Other option: pass information to global context vector U, also called “master node”.



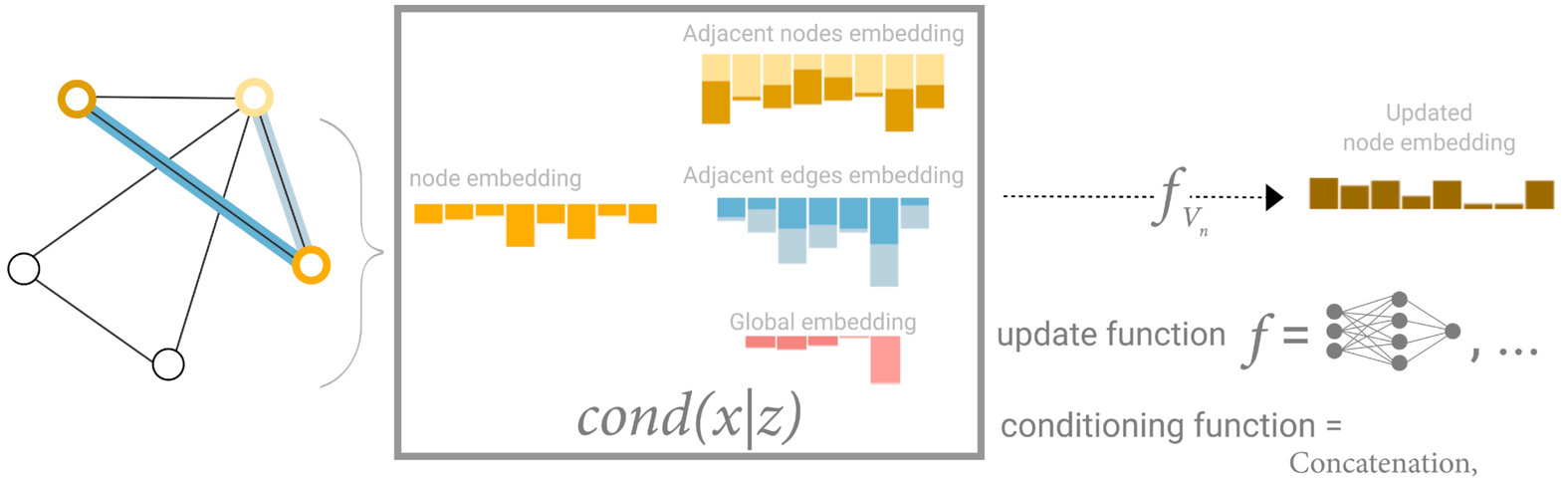
Schematic of a Graph Nets architecture leveraging global representations.

Feature-wise Linear Modulation (FiLM)

Now all graph attributes have learned representations (all connected by a neural net)

For a node, we can use information from neighbours, connected edges, and global.

- To condition new node embedding on all these, we just concatenate them.
- We can also apply a *feature-wise modulation layer* (FiLM): output $= \gamma$input $+ \beta$
- 'input' is the input feature map, γ and β are learned by another part of the network



Applications of GNN

Molecules

Leffingwell Odor dataset

- Dataset: small molecules, labels: odour information
- First task: classifying if a molecular graph smells “pungent” or not
- We represent each molecule as a graph:
- atoms are nodes containing a one-hot encoding for its atomic identity (Carbon, Nitrogen, Oxygen, Fluorine)
- bonds are edges containing a one-hot encoding its bond type (single, double, triple or aromatic).
- Our general modelling template for this problem will be built up using sequential GNN layers, followed by a linear model with a sigmoid activation for classification.

Leffingwell Odor Dataset, B. Sanchez-Lengeling, J.N. Wei, B.K. Lee, R.C. Gerkin, A. Aspuru-Guzik, A.B. Wiltschko. 2020.
Machine Learning for Scent: Learning Generalizable Perceptual Representations of Small Molecules, B. Sanchez-Lengeling et al., 2019.

Creating the GNN

Design space

Decisions:

- The number of GNN layers, also called the *depth*.
- The dimensionality of each attribute when updated. The update function is a 1-layer MLP with a `relu` activation function and a layer norm for normalisation of activations.
- The aggregation function used in pooling: max, mean or sum.
- The graph attributes that get updated, or styles of message passing: nodes, edges and global representation. We control these via boolean toggles (on or off).
- A baseline model would be a graph-independent GNN (all message-passing off) which aggregates all data at the end into a single global attribute. Toggling on all message-passing functions yields a GraphNets architecture.

Example application

- Create visualisation for the last layer (before classification)
- Embeddings are high-dimensional (difficult to visualise) — use PCA to reduce to 2 dim

Edit the molecule to see how the prediction changes, or change the model params to load a different model. Select a different molecule in the scatter plot.

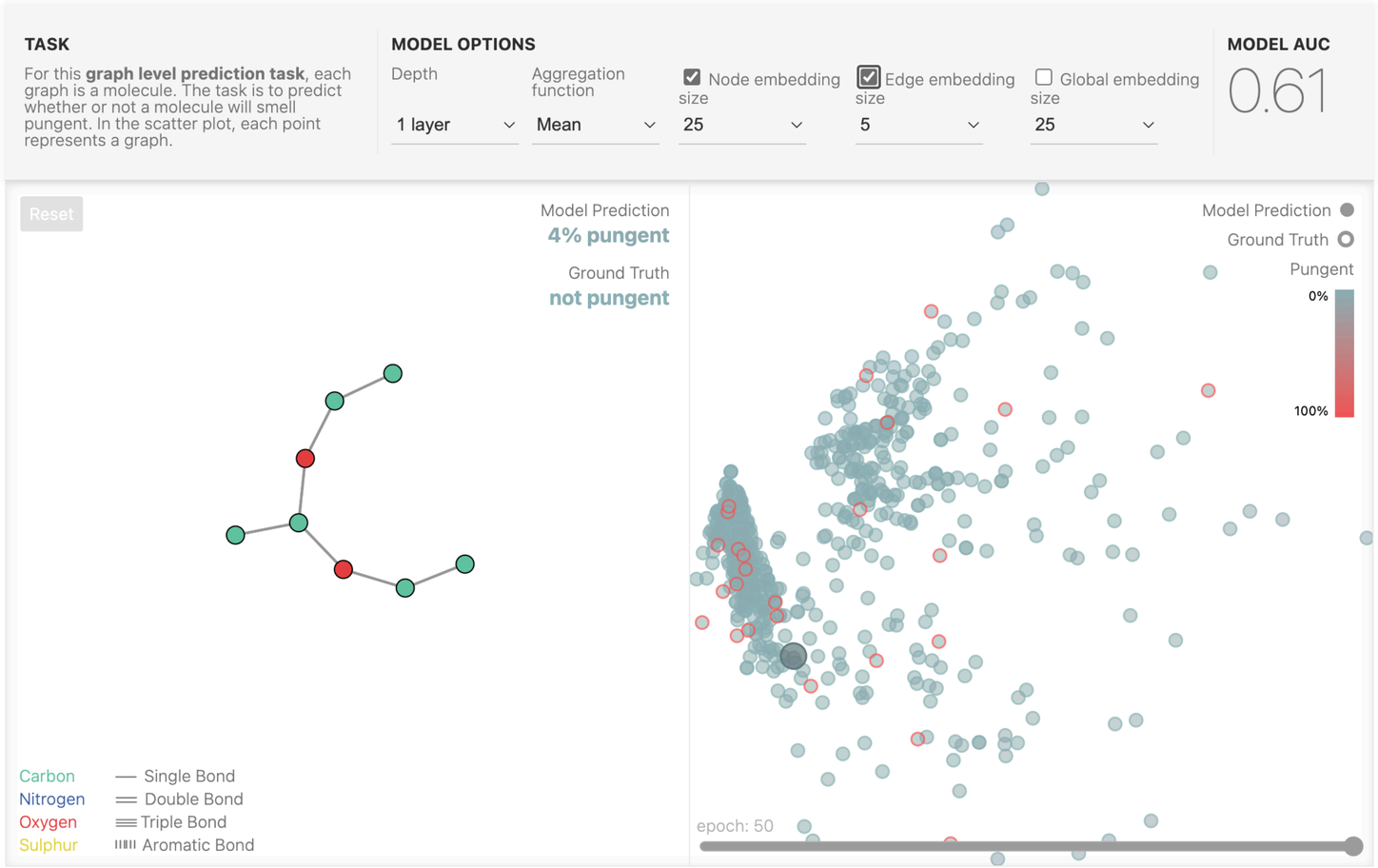
Example application

- Create visualisation for the last layer (before classification)
- Embeddings are high-dimensional (difficult to visualise) — use PCA to reduce to 2 dim

Edit the molecule to see how the prediction changes, or change the model params to load a different model. Select a different molecule in the scatter plot.

Example application

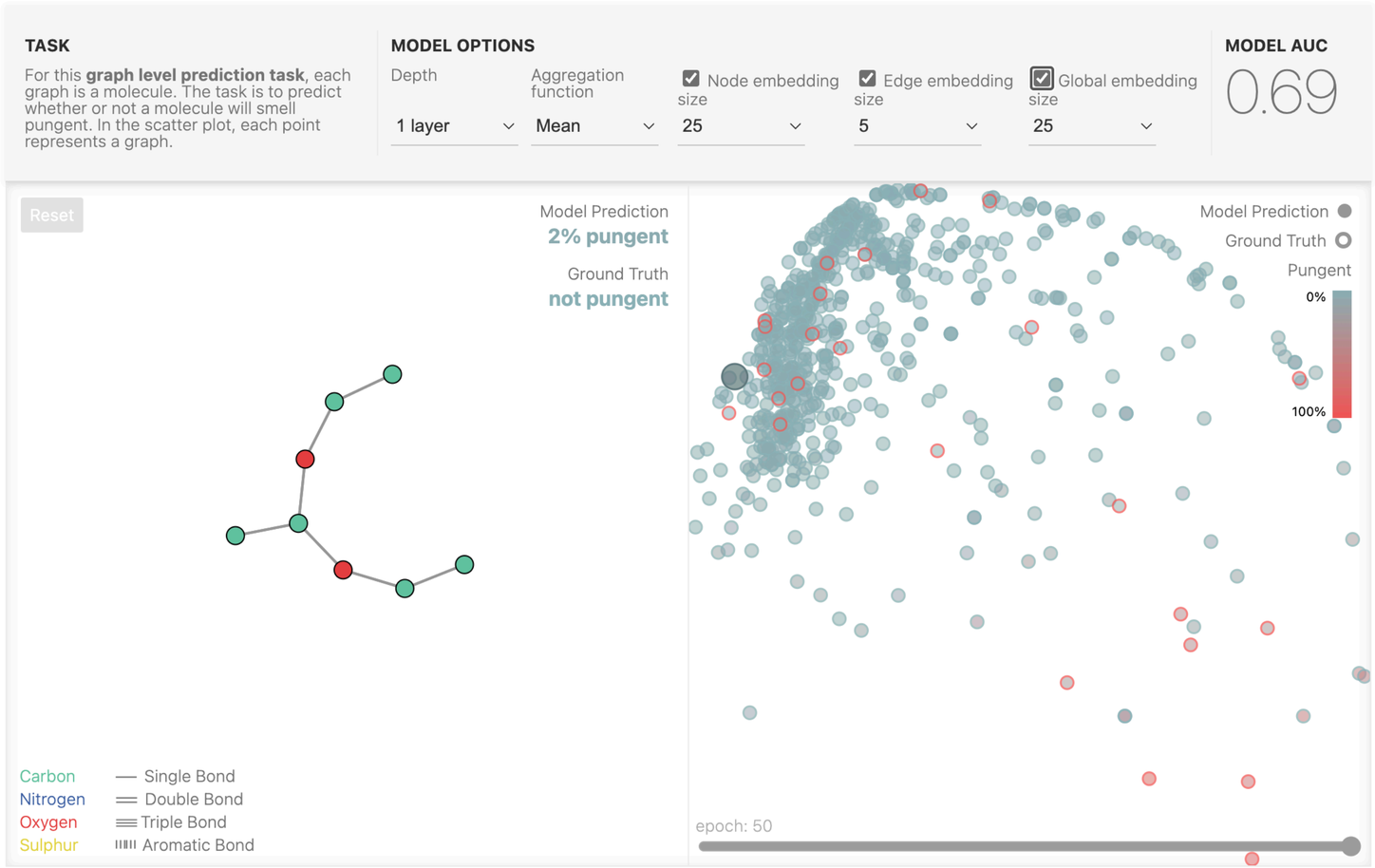
- Create visualisation for the last layer (before classification)
- Embeddings are high-dimensional (difficult to visualise) — use PCA to reduce to 2 dim



Edit the molecule to see how the prediction changes, or change the model params to load a different model. Select a different molecule in the scatter plot.

Example application

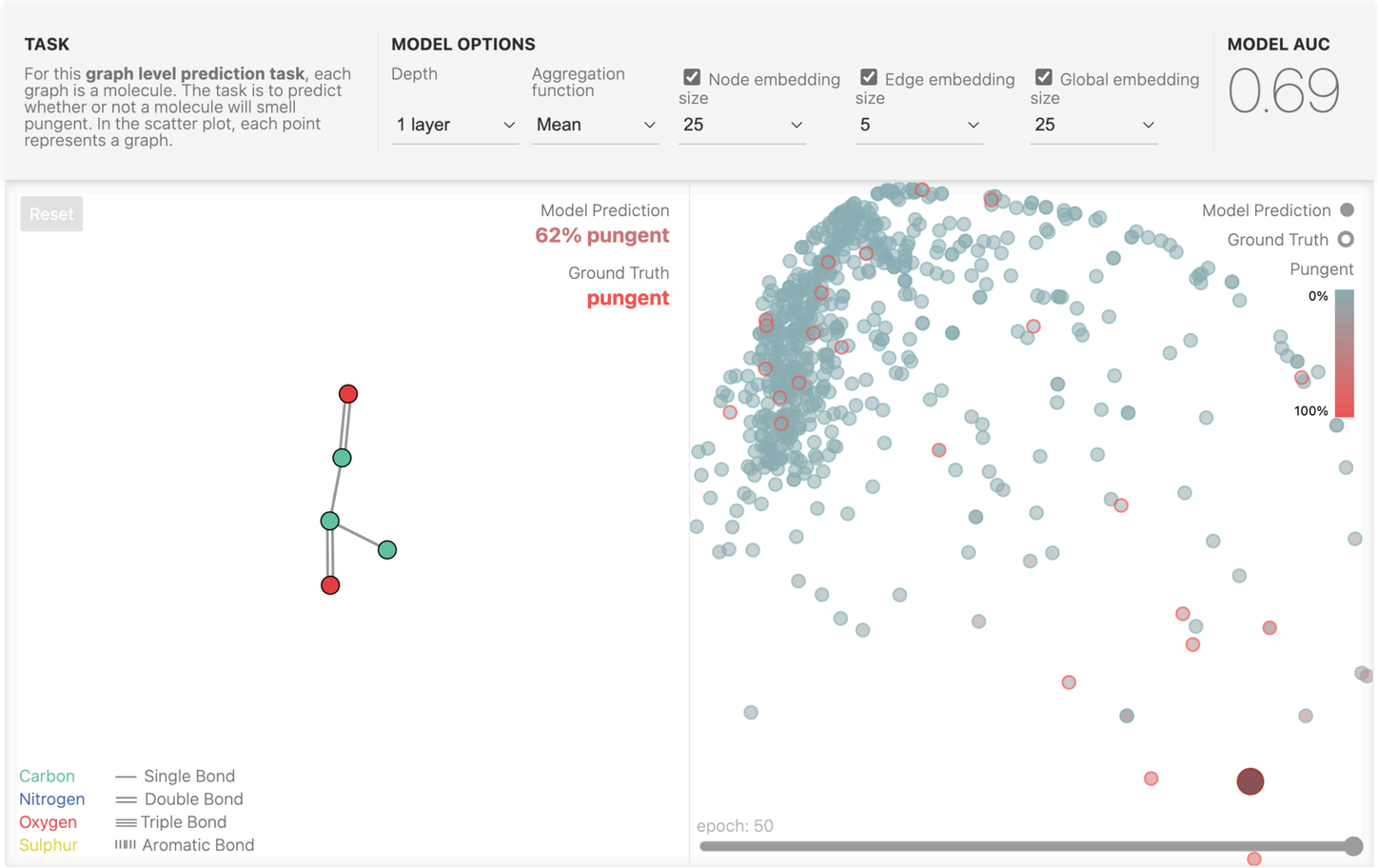
- Create visualisation for the last layer (before classification)
- Embeddings are high-dimensional (difficult to visualise) — use PCA to reduce to 2 dim



Edit the molecule to see how the prediction changes, or change the model params to load a different model. Select a different molecule in the scatter plot.

Example application

- Create visualisation for the last layer (before classification)
- Embeddings are high-dimensional (difficult to visualise) — use PCA to reduce to 2 dim

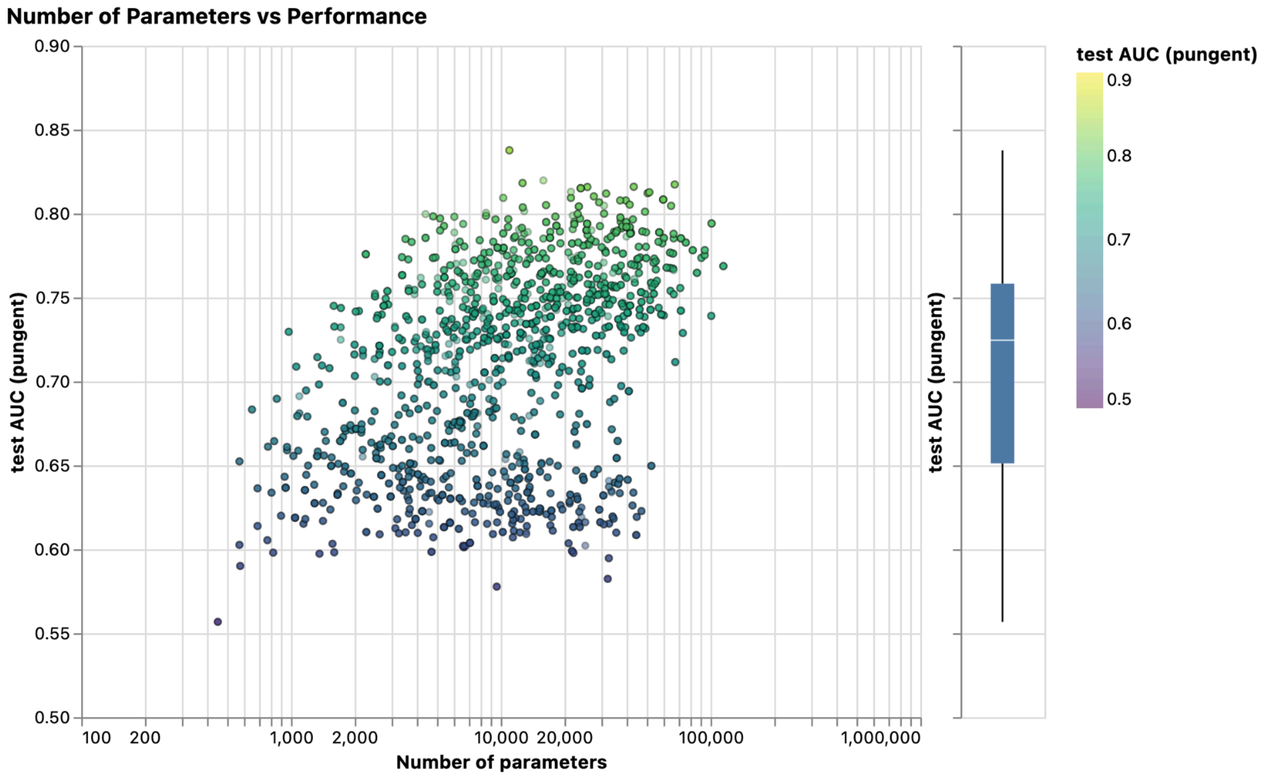


Edit the molecule to see how the prediction changes, or change the model params to load a different model. Select a different molecule in the scatter plot.

GNN design lessons

Some models perform better than others

- It depends … (on the data set)
- But some empirical results here

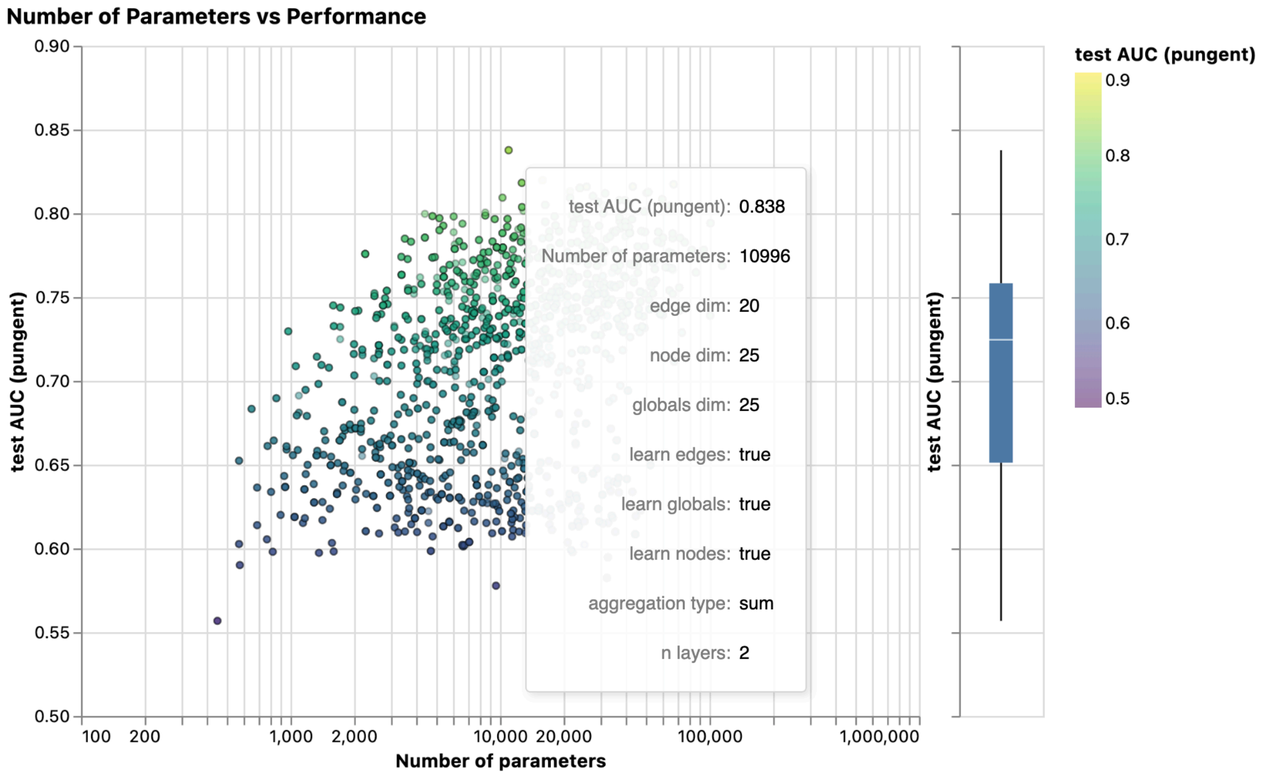


Scatterplot of each model's performance vs its number of trainable variables. Hover over a point to see the GNN architecture parameters.

GNN design lessons

Some models perform better than others

- It depends ... (on the data set)
- But some empirical results here



Scatterplot of each model's performance vs its number of trainable variables. Hover over a point to see the GNN architecture parameters.

Summary

Graph neural networks intro

With message passing

- Introduced Graph neural networks
- Starting from simple feed-forward networks
- Message passing: similar (but different) to message passing in probabilistic models
- More “free-form”: different messages are possible - since everything is learnt
- But: no direct probabilistic interpretation within the network
- We can create outputs that can be interpreted as probabilities though