# GRAPHS ALL THE WAY DOWN

DESIGNING GNN ARCHITECTURES FOR PARTICLE TRACKING

# OVERVIEW

**PART I: GNNs for Particle Tracking**

1. Short review of tracking problem
2. GNN solution
3. A taste of some GNN architectures

**PART II: ML Pipeline and Architecture Design with Graphs**

1. *The GNN Sandbox*: A place for fun and games
2. *DVC + Weights & Biases*: Reproducible hyperparameter optimisation
3. *ArchiOpTrX*: A GNN-based Neural Architecture Search

# PART I: GNNS FOR PARTICLE TRACKING

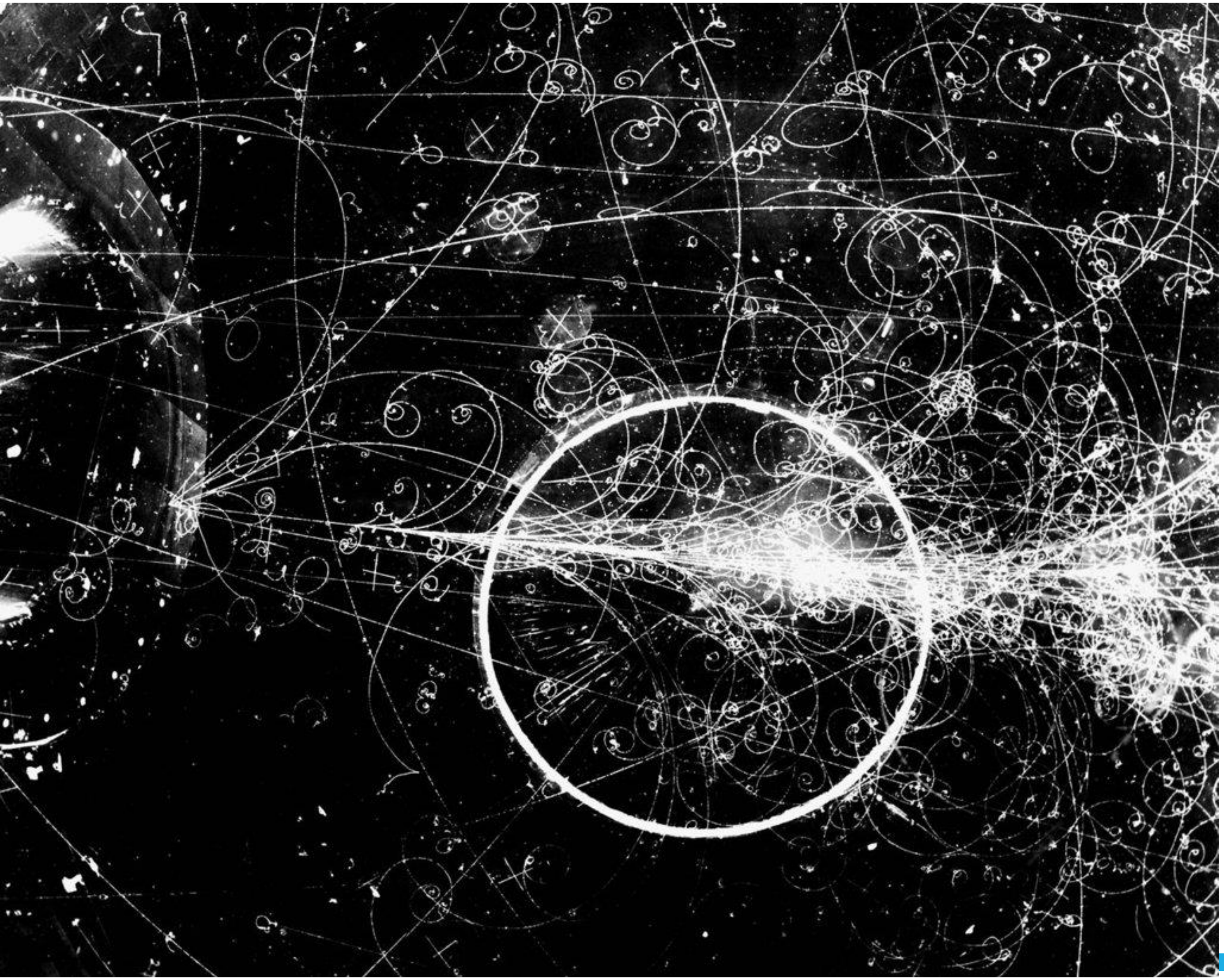
# NEW PHYSICS NEEDS COLLISIONS...

- Higgs boson (LHC),
- Quarks (SLAC, Fermilab), and
- Neutrino mass (Super-Kamiokande)

Discovered with collisions

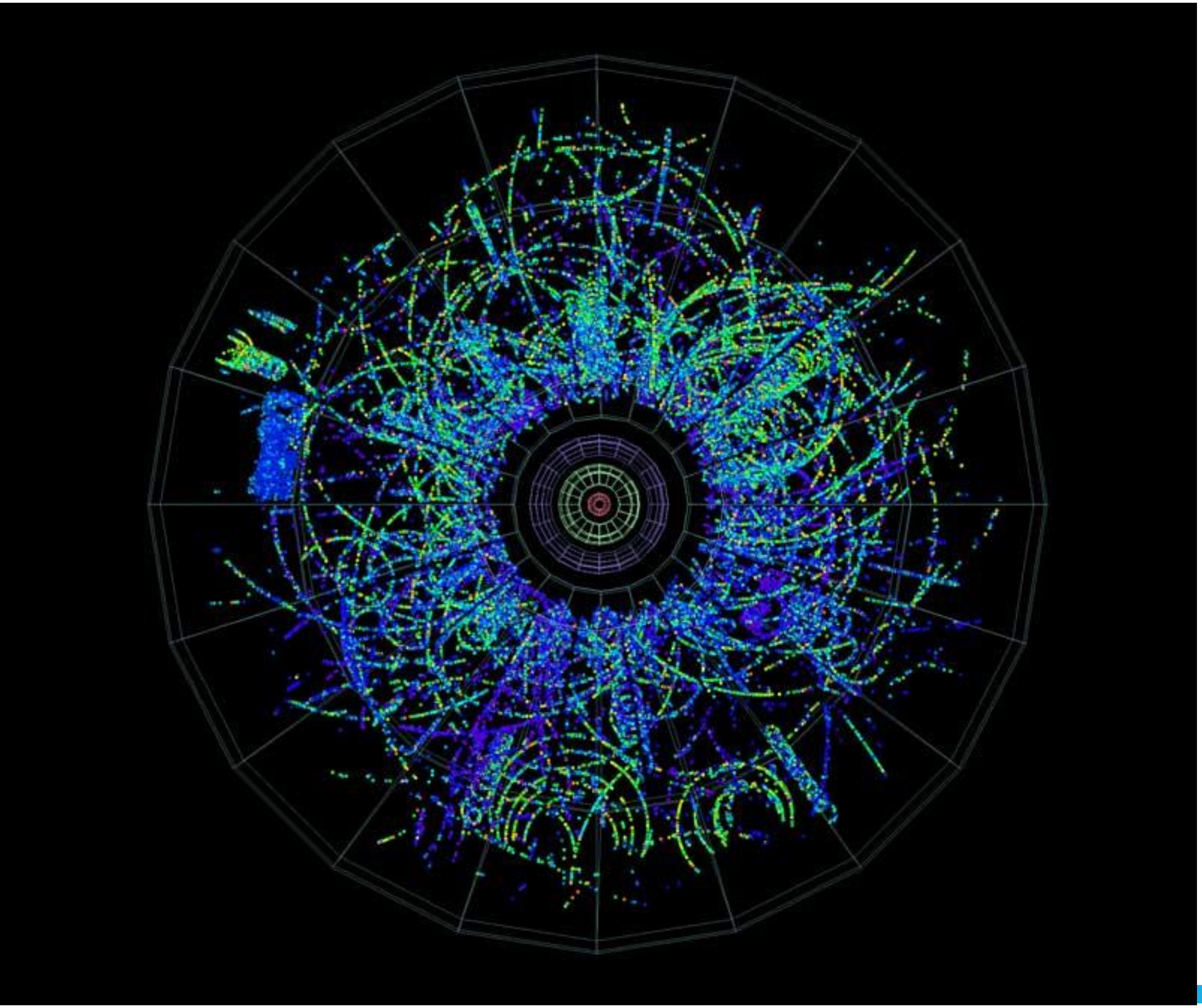
- Supersymmetry,
- Composite Higgs,
- Dark matter,
- Leptoquarks,
- W/Z prime, and
- Axions

Could be discovered with collisions



# … BUT COLLISIONS ARE MESSY

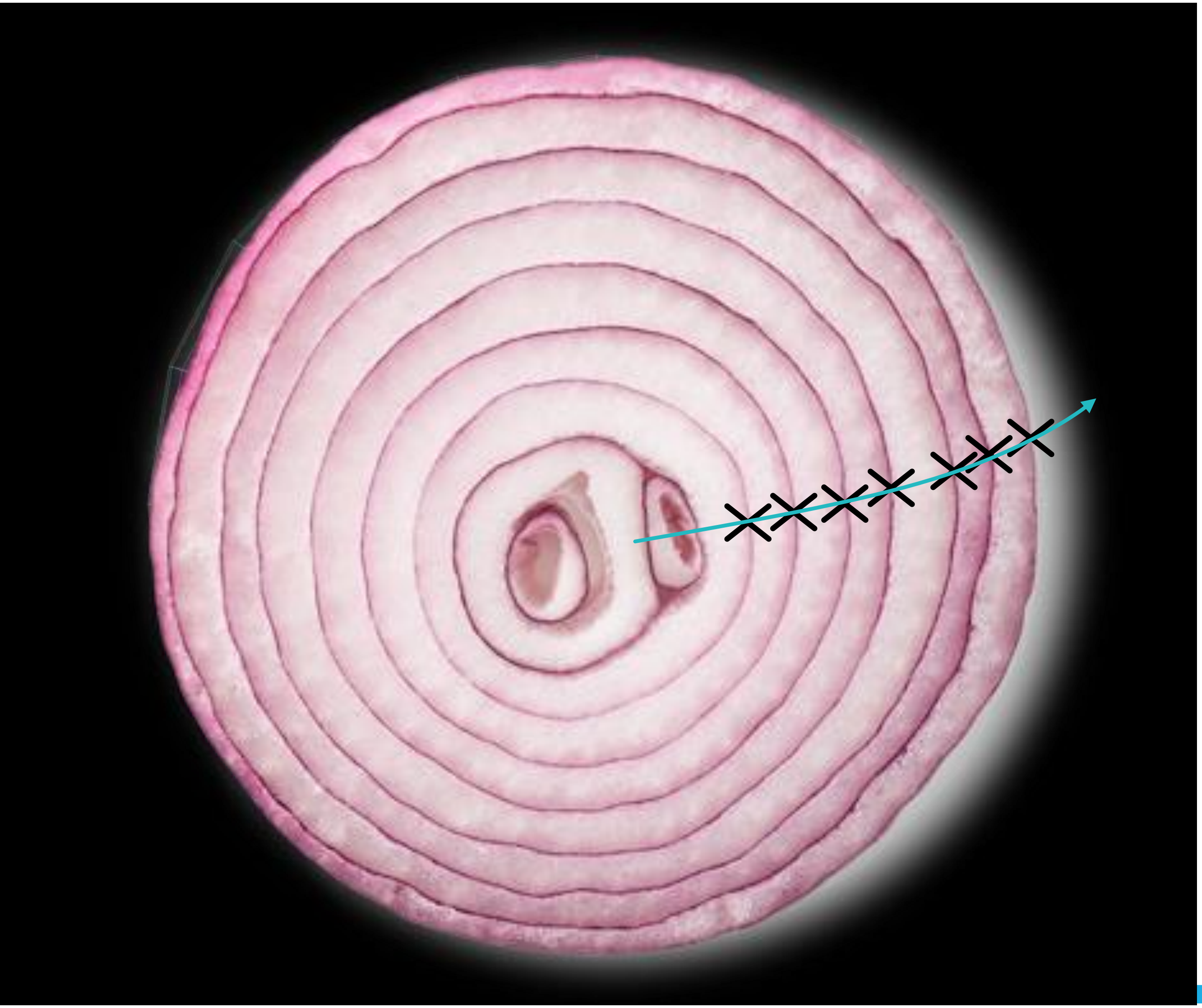
- High energy collisions bring huge numbers of particles (unfortunately)
- Want to see the particles coming out of the collisions, which we can get from the curves (“tracks”) moving through a magnetic field



# THE "TRACKING PROBLEM" OF NEW PHYSICS

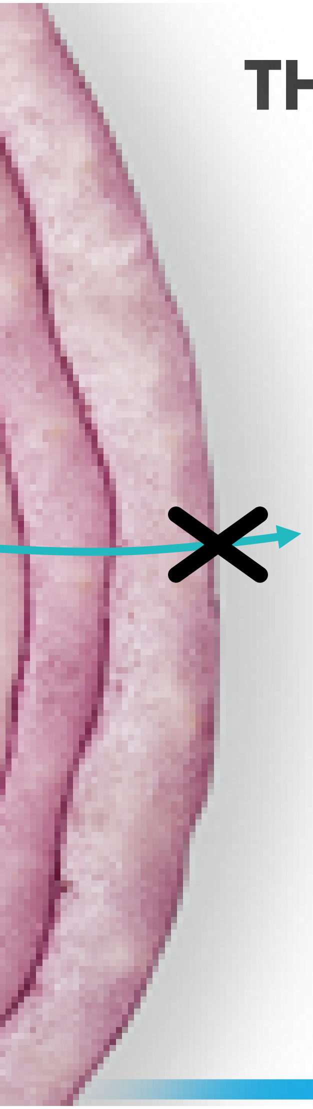
- New physics requires high energy and high precision
- This implies carefully tracking millions of particles per event through the (as-few-as-possible) layers of a detector
- Each collision comprises of dozens of events
- Each second produces tens of millions of collisions

# THE "TRACKING PROBLEM" OF NEW PHYSICS



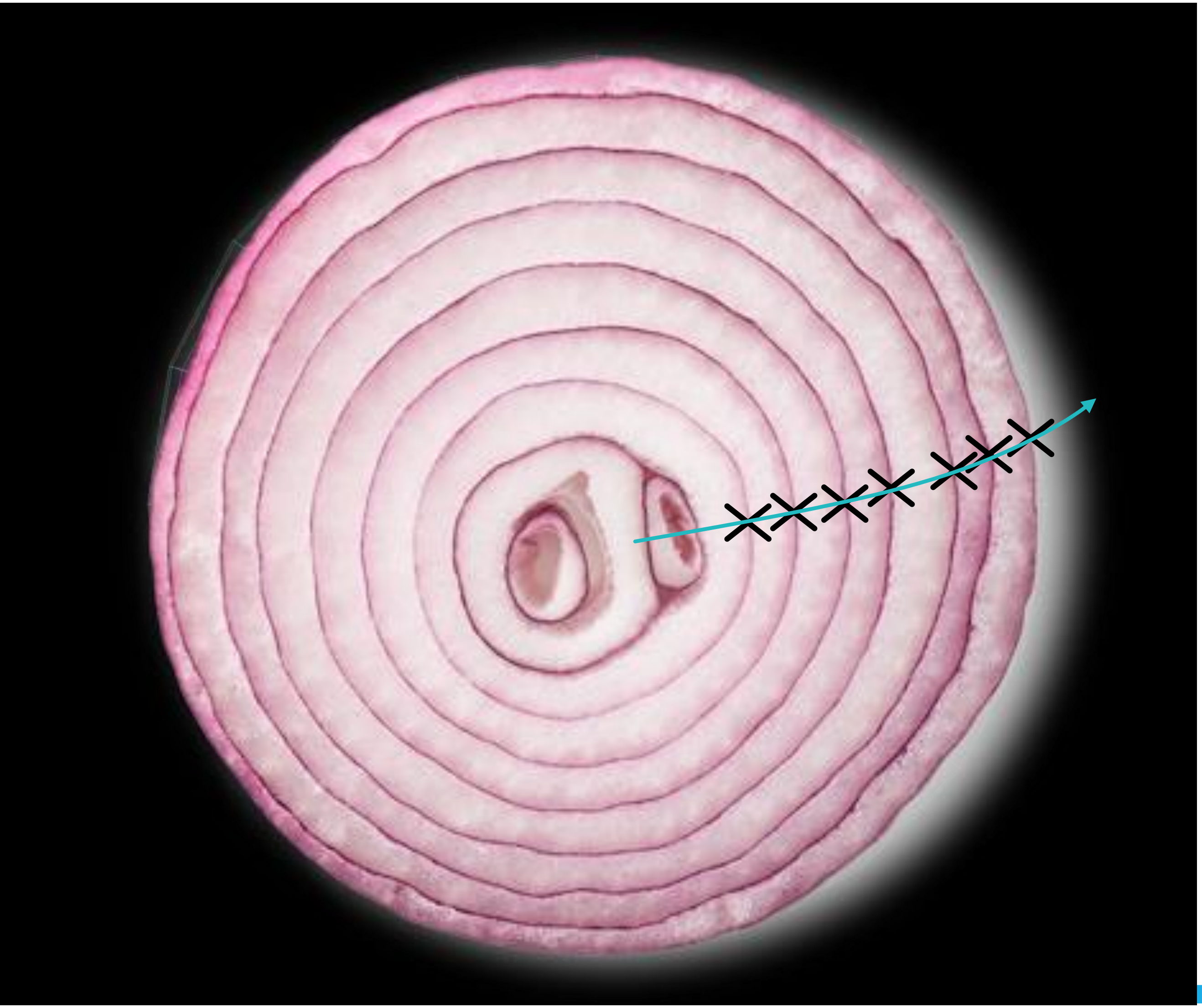
- New physics requires high energy and high precision
- This implies carefully tracking millions of particles per event **through the (as-few-as-possible) layers of a detector**
- Each collision comprises of dozens of events
- Each second produces tens of millions of collisions

# THE “TRACKING PROBLEM” OF NEW PHYSICS



A particle interacting with a layer is a “hit”

- New physics requires high energy and high precision
- This implies carefully tracking millions of particles per event **through the (as-few-as-possible) layers of a detector**
- Each collision comprises of dozens of events
- Each second produces tens of millions of collisions

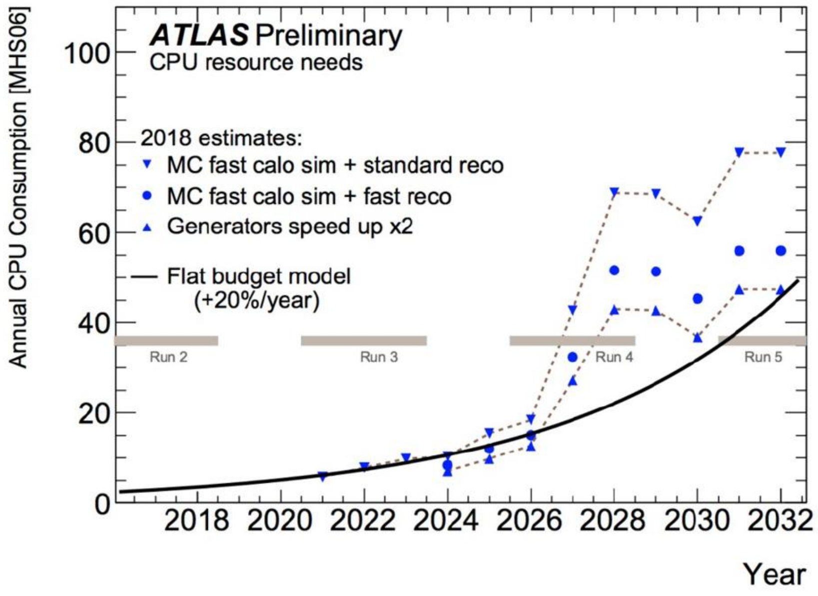


## THE “TRACKING PROBLEM” OF NEW PHYSICS

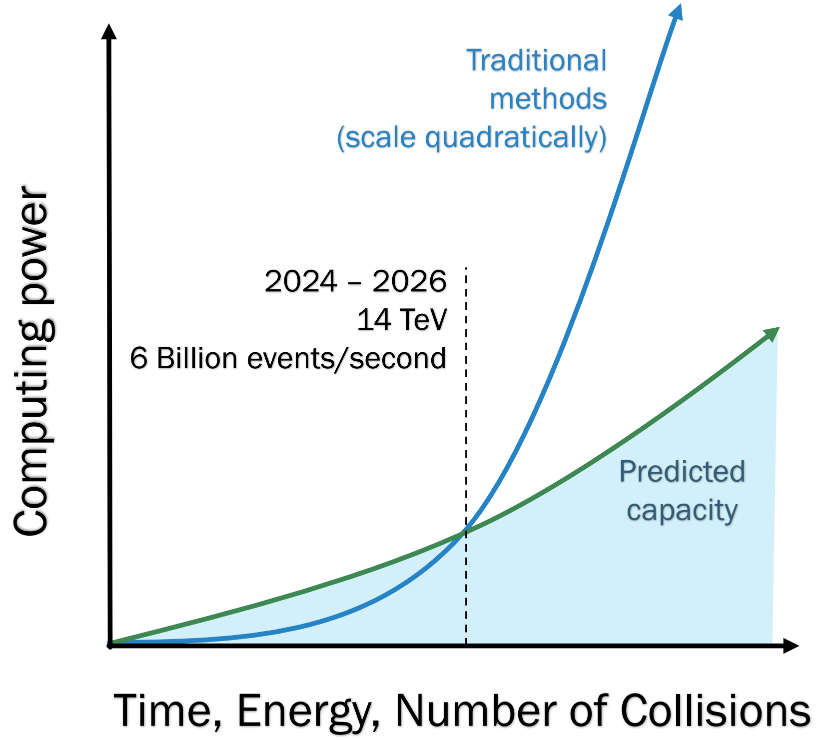
We need a fast, high-accuracy method to connect hits into tracks to determine the types and energies of particles coming out of every event

# CURRENT TECHNIQUES WILL* NOT WORK ON NEXT-GEN COLLIDERS

Standard doom-and-gloom plot




In other words…




*probably

# WE WANT TO BUILD TRACKS

1. Observe hits on layers

2. Join hits into track

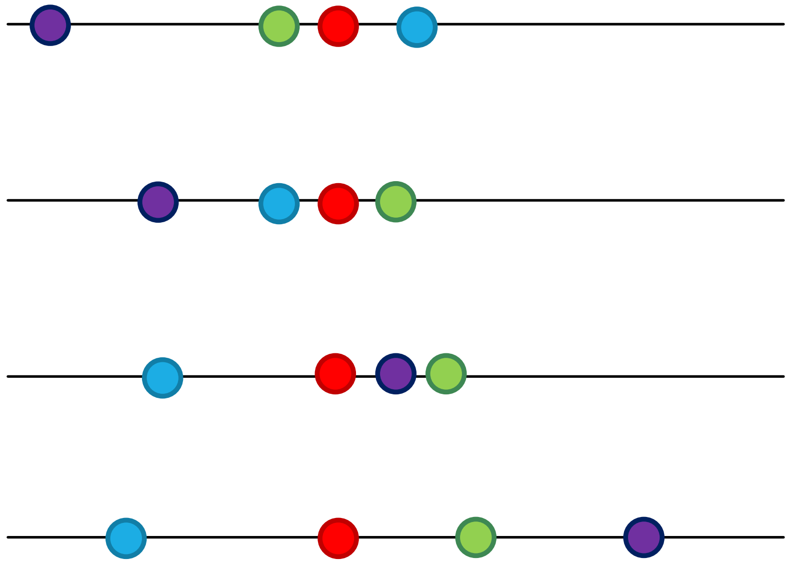
This is our focus

3. Convert track into particle information
4. (Dis)prove supersymmetry

## GRAPHS ARE A NATURAL WAY TO REPRESENT TRACKS

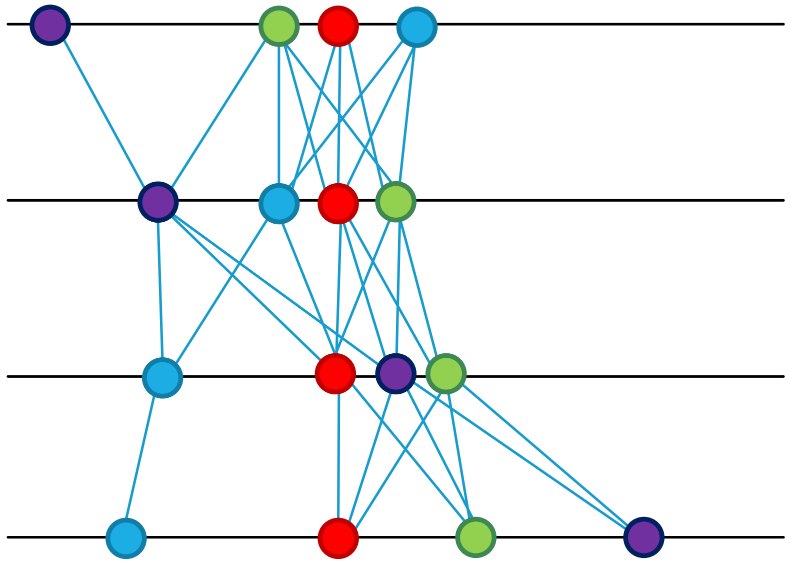
- We have a collection of hits
- Want to “Connect the Dots” (the conference name for this problem)
- A natural way is to represent the problem is as a graph

# TRACKS AS GRAPHS



- **Graph is a collection of nodes, connected by edges**
- Graph construction scales poorly, but is not the bottleneck – it is a simple combinatorial process
- Graph prediction scales well – approximately as O(e) with number of edges e

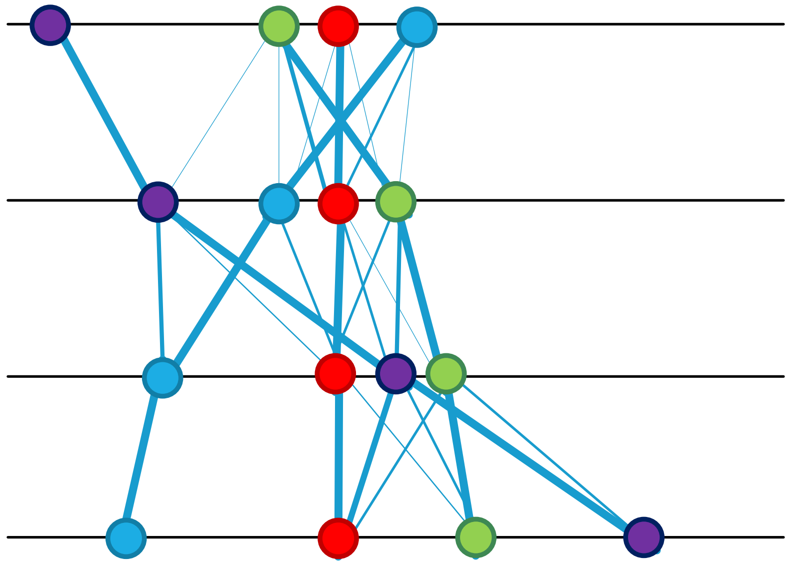
# TRACKS AS GRAPHS



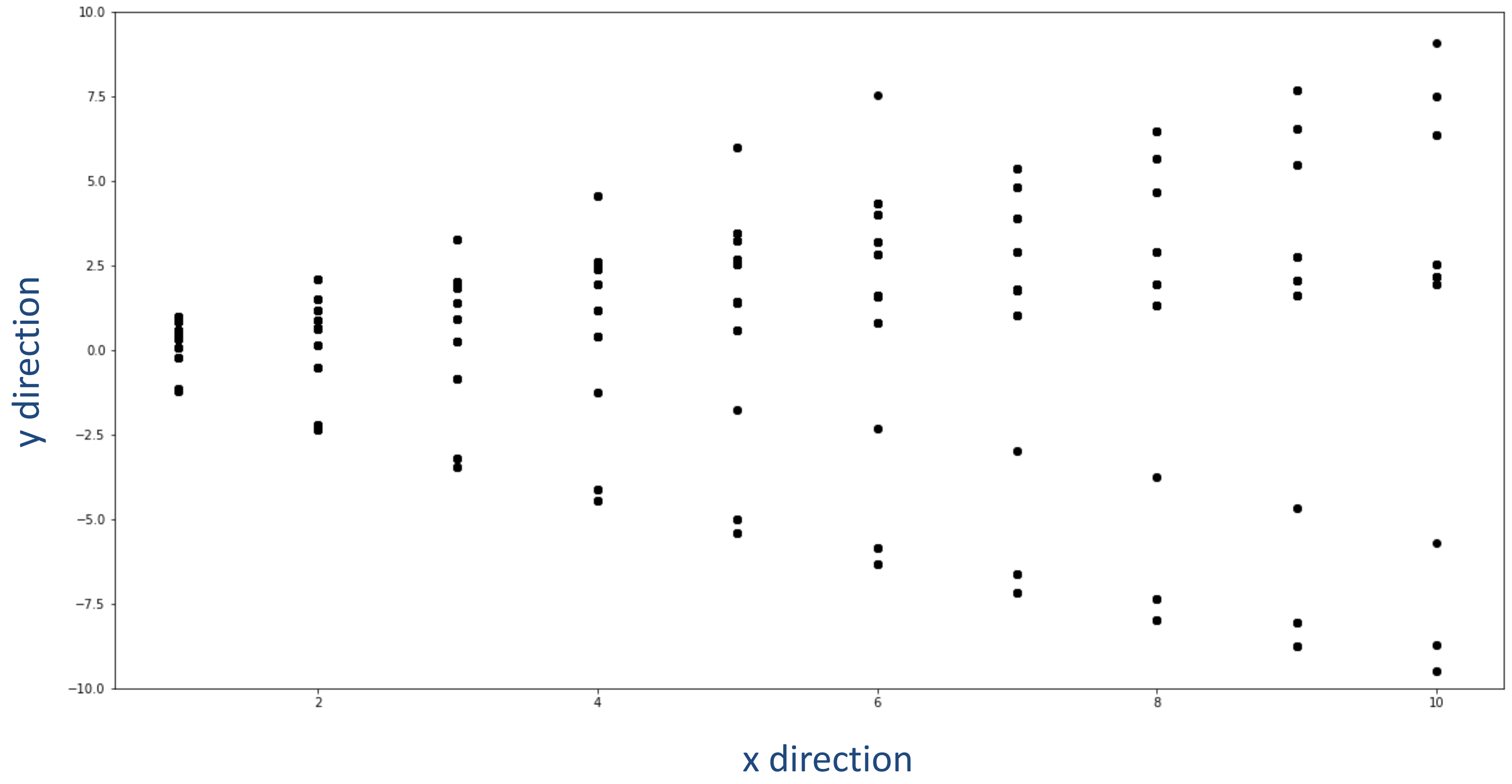
- Graph is a collection of nodes, connected by edges
- **Graph construction scales poorly, but is not the bottleneck – it is a simple combinatorial process**
- Graph prediction scales well – approximately as O(e) with number of edges e

# TRACKS AS GRAPHS

- Graph is a collection of nodes, connected by edges
- Graph construction scales poorly, but is not the bottleneck – it is a simple combinatorial process
- **Graph prediction scales well – approximately as O(e) with number of edges e**

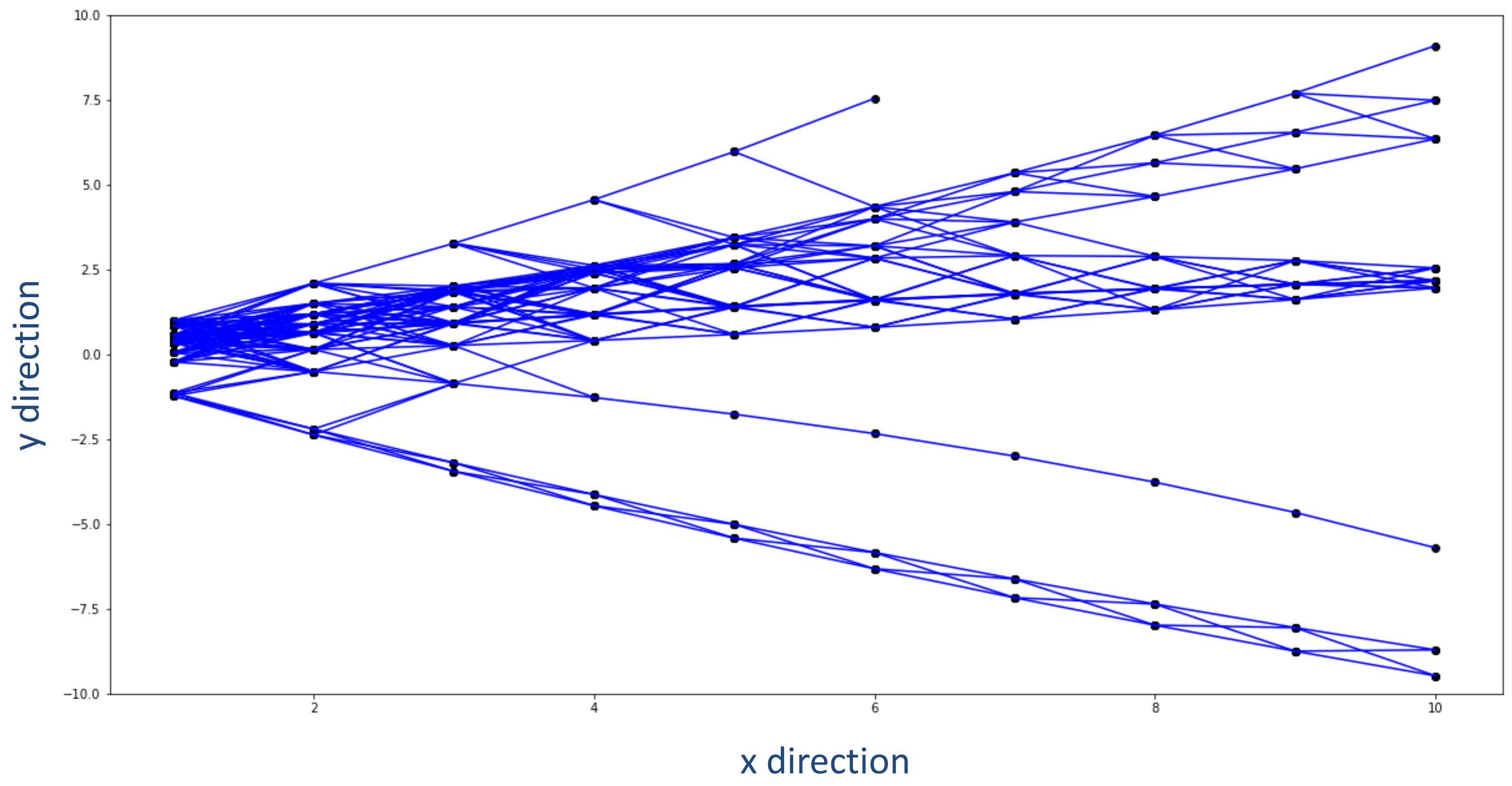


# GRAPHS ARE A NATURAL WAY TO REPRESENT TRACKS



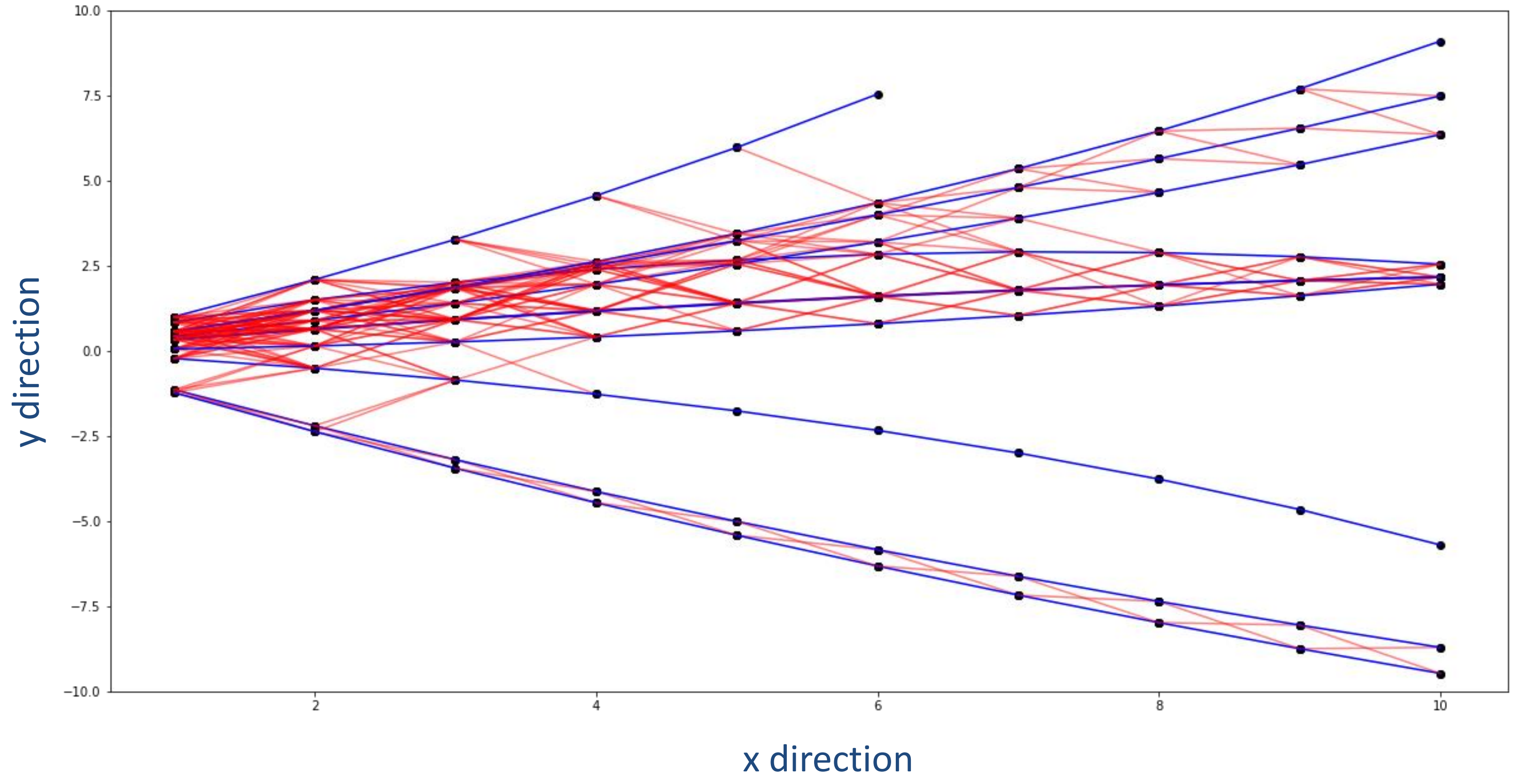

Some toy data…

# GRAPHS ARE A NATURAL WAY TO REPRESENT TRACKS



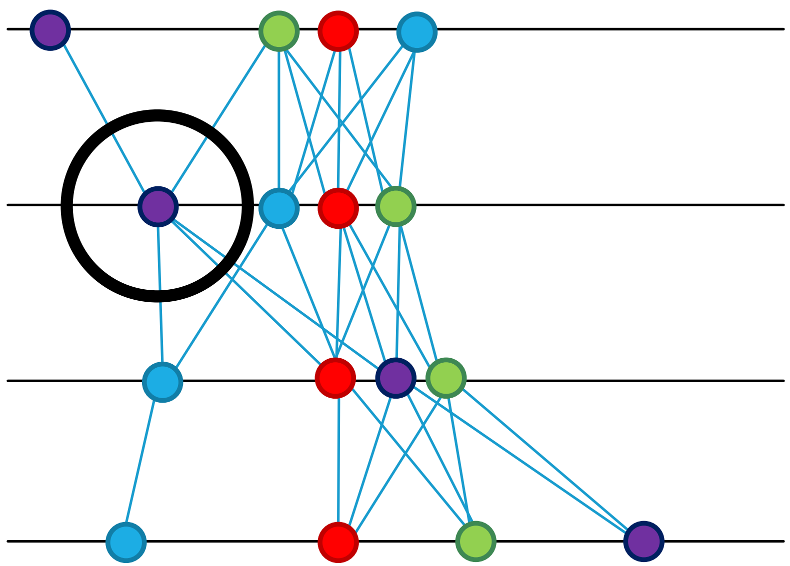

Join the hits in some clever/dumb way…

# GRAPHS ARE A NATURAL WAY TO REPRESENT TRACKS




The tracks should be in here

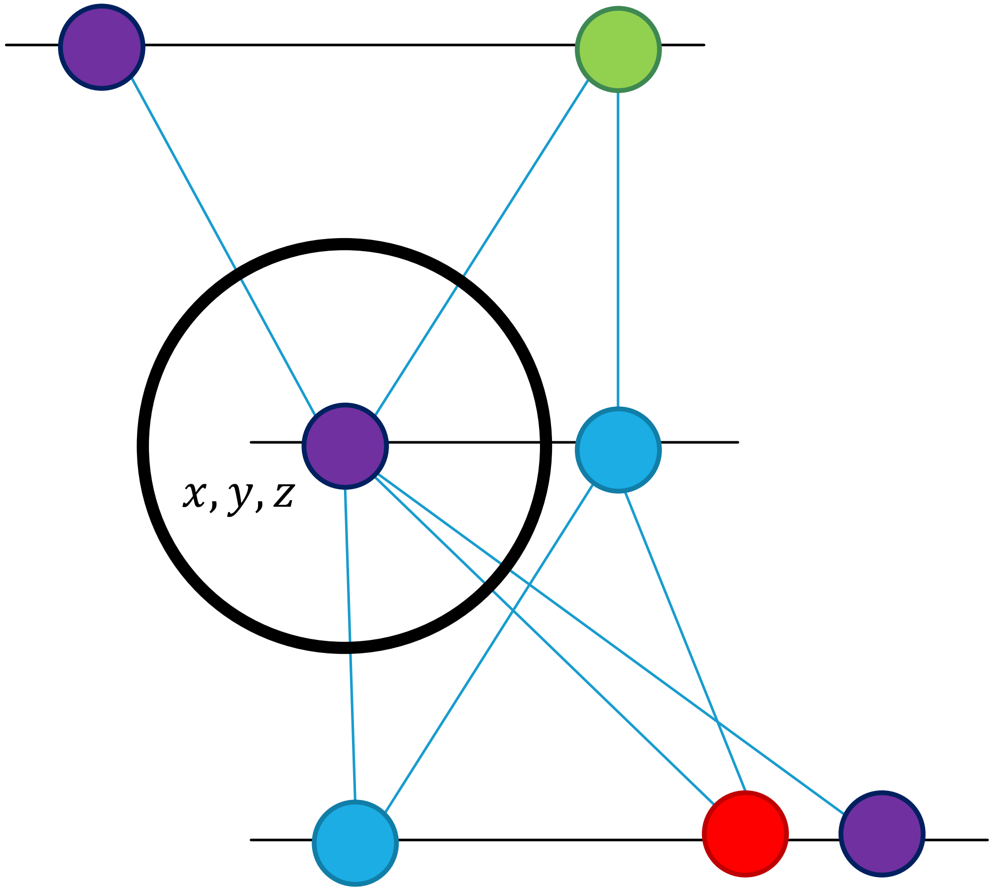
# GRAPH INPUT FEATURES



- Can attach low-level or high-level features to each node (i.e. hit)

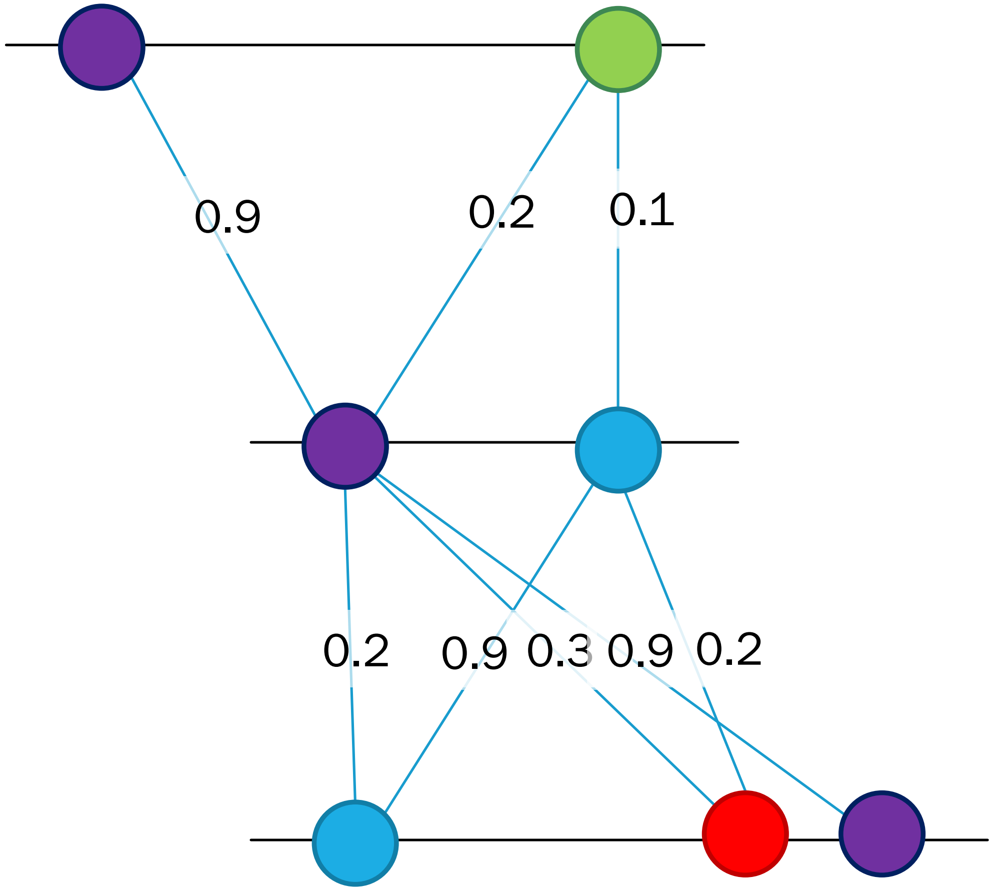
# GRAPH INPUT FEATURES

- Can attach low-level or high-level features to each node (i.e. hit)
- Low level
  - $x, y, z$
  - $r, \phi, z$
- Higher level
  - $\eta$
  - Cell/cluster information



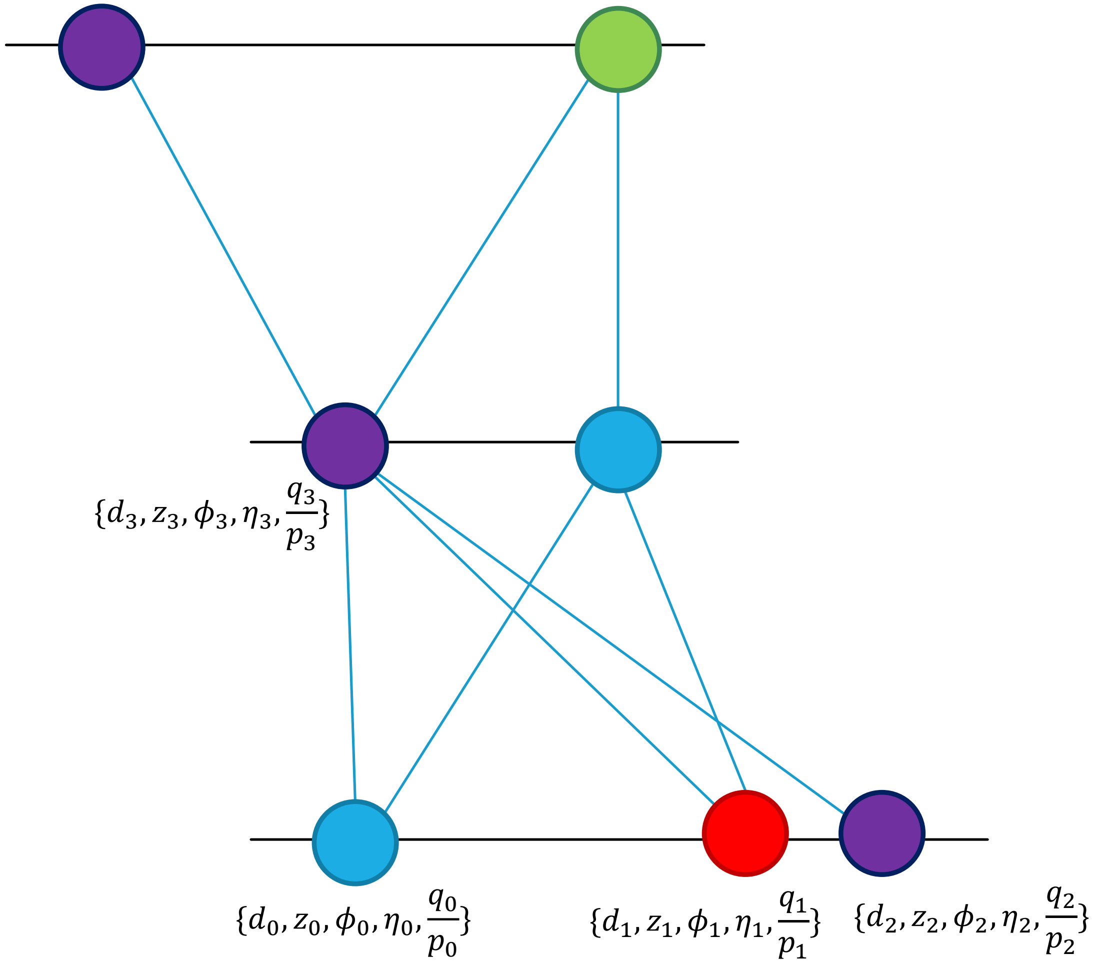
# GRAPH OUTPUT FEATURES

- Output depends on our question…
- **Output as edge feature** – score of each edge being true



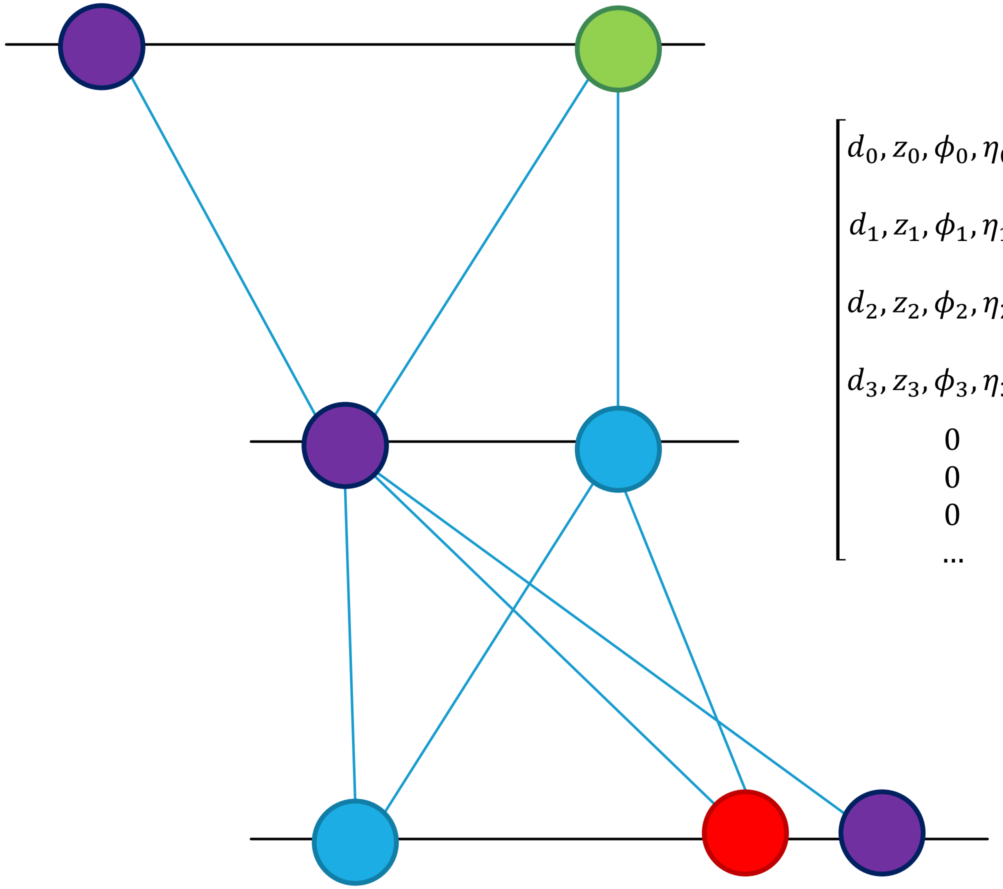
# GRAPH OUTPUT FEATURES

- Output depends on our question…
- Output as edge feature – score of each edge being true
- **Output as node feature** – track parameters of a track associated to each hit



# GRAPH OUTPUT FEATURES

- Output depends on our question…
- Output as edge features – score of each doublet being true
- Output as node features – track parameters of a track associated to each hit
- **Output as graph features** – padded prediction of n tracks and their parameters

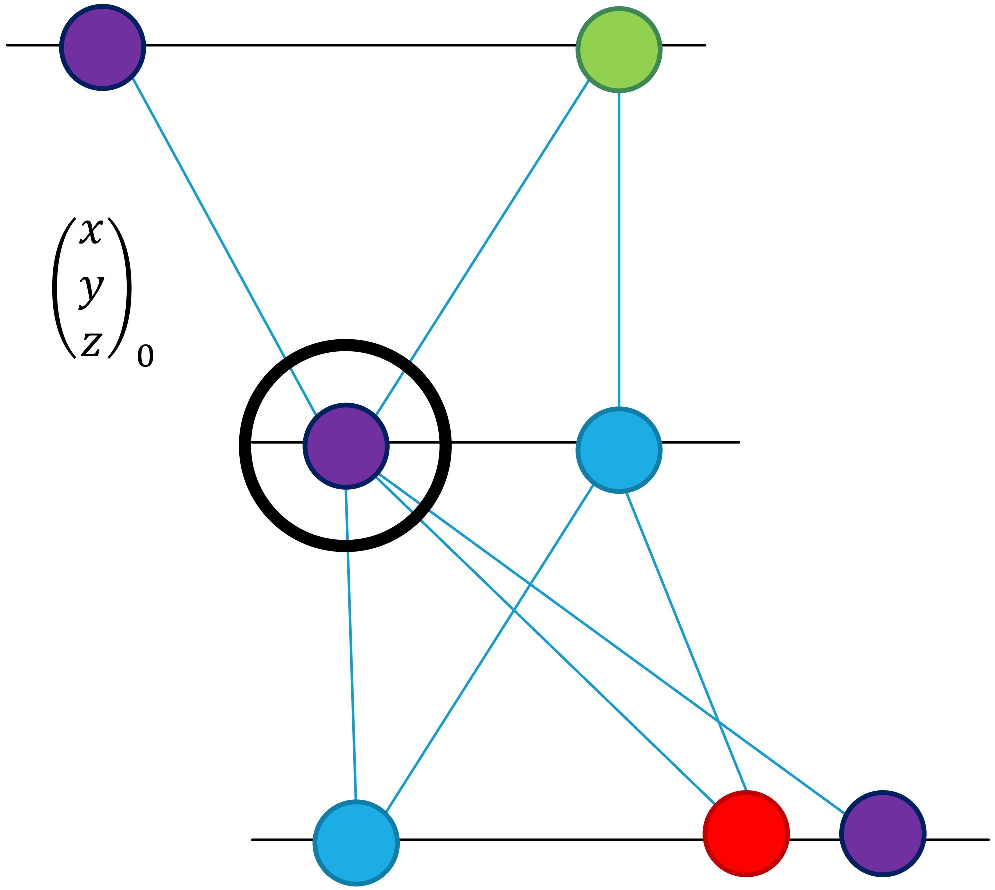


$$\begin{bmatrix} d_0, z_0, \phi_0, \eta_0, \frac{q_0}{p_0} \\ d_1, z_1, \phi_1, \eta_1, \frac{q_1}{p_1} \\ d_2, z_2, \phi_2, \eta_2, \frac{q_2}{p_2} \\ d_3, z_3, \phi_3, \eta_3, \frac{q_3}{p_3} \\ 0 \\ 0 \\ 0 \\ \dots \end{bmatrix}$$

# EDGE PREDICTION ARCHITECTURE

- **Input node features**
- Hidden node features
- Hidden edge features
- Edge score
- Attention aggregation
- New hidden node features
- New hidden edge features
- New edge score

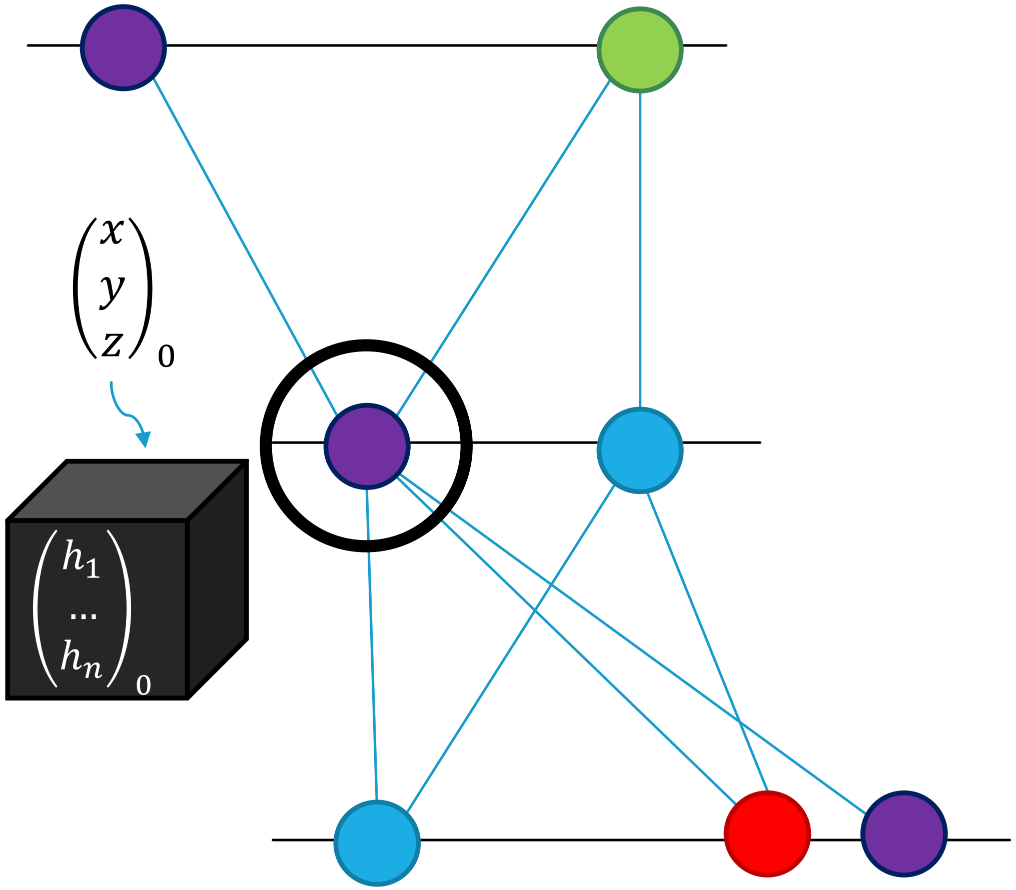
x n iterations
(hyperparameter)



# EDGE PREDICTION ARCHITECTURE

- Input node features
- **Hidden node features**
- Hidden edge features
- Edge score
- Attention aggregation
- New hidden node features
- New hidden edge features
- New edge score

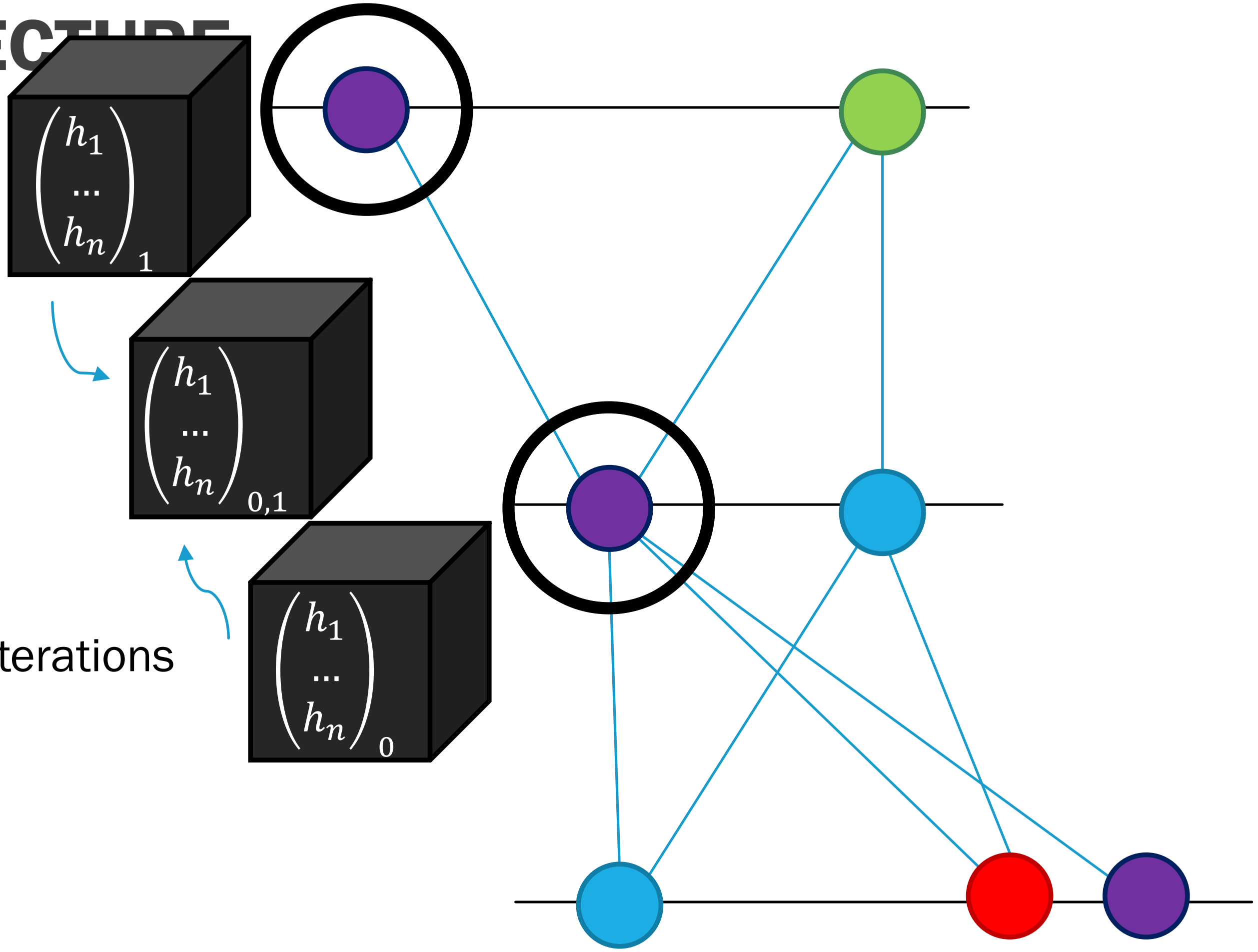
x n iterations



# EDGE PREDICTION ARCHITECTURE

- Input node features
- Hidden node features
- **Hidden edge features**
- Edge score
- Attention aggregation
- New hidden node features
- New hidden edge features
- New edge score

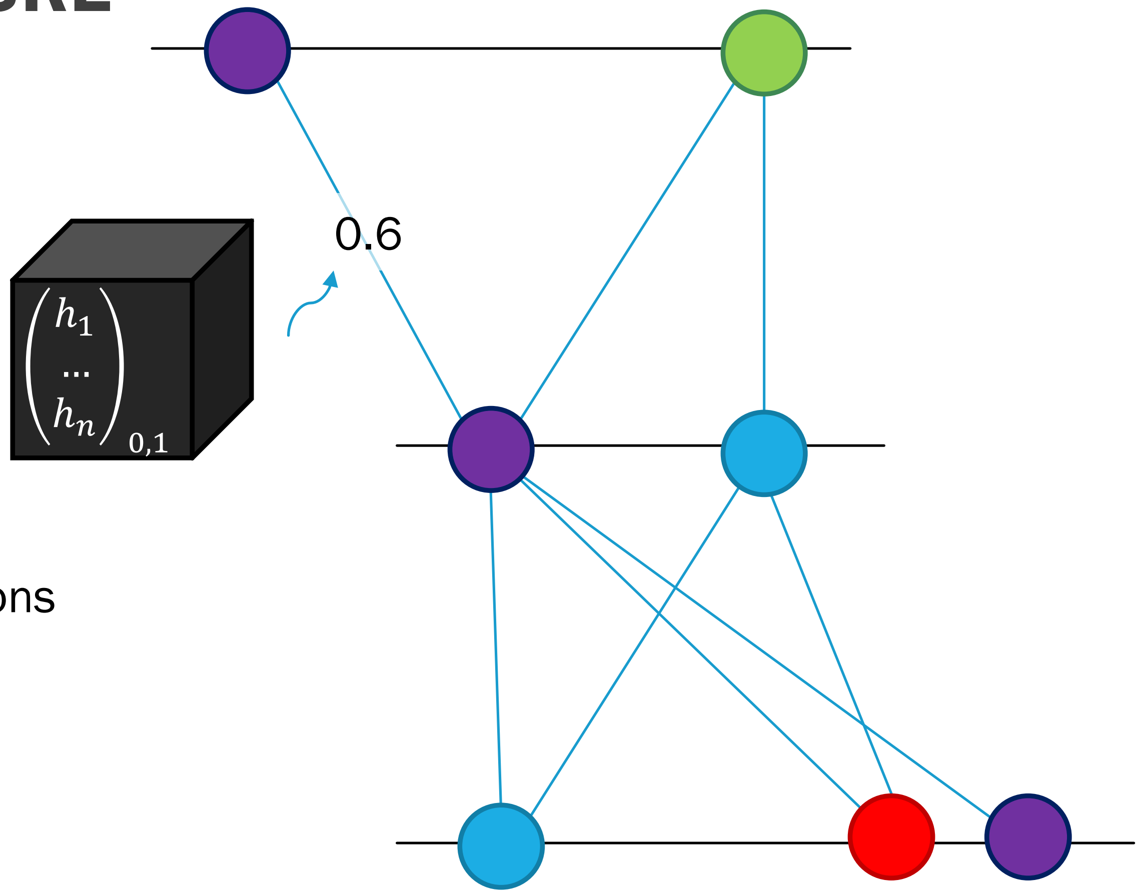
x n iterations



# EDGE PREDICTION ARCHITECTURE

- Input node features
- Hidden node features
- Hidden edge features
- **Edge score**
- Attention aggregation
- New hidden node features
- New hidden edge features
- New edge score

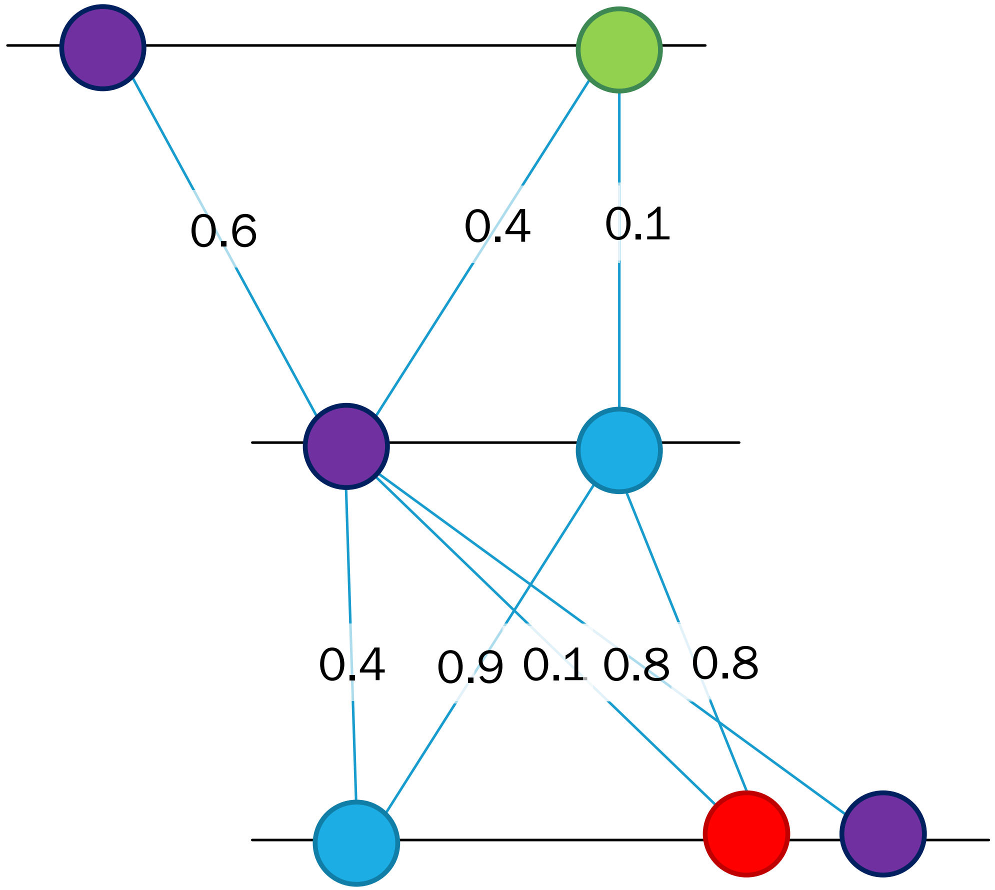
x n iterations



# EDGE PREDICTION ARCHITECTURE

- Input node features
- Hidden node features
- Hidden edge features
- **Edge score**
- Attention aggregation
- New hidden node features
- New hidden edge features
- New edge score

x n iterations



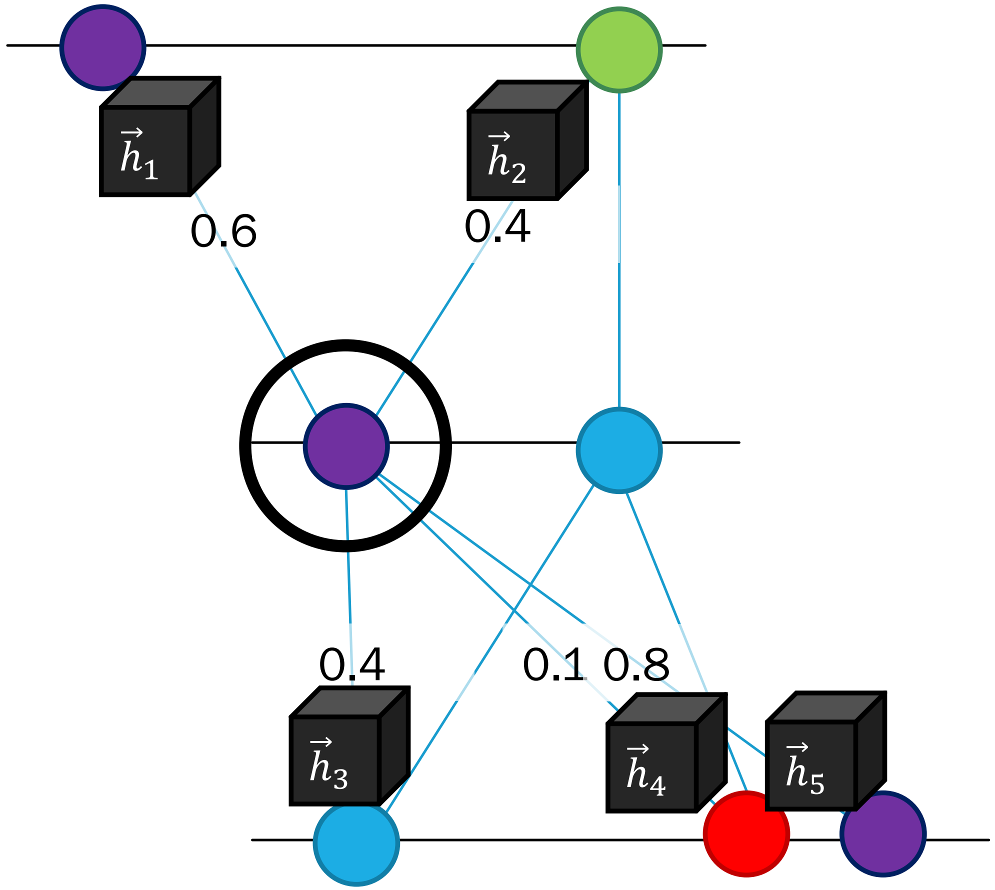

BERKELEY LAB

EXA TrkX

# EDGE PREDICTION ARCHITECTURE

- Input node features
- Hidden node features
- Hidden edge features
- Edge score
- **Attention aggregation**
- New hidden node features
- New hidden edge features
- New edge score

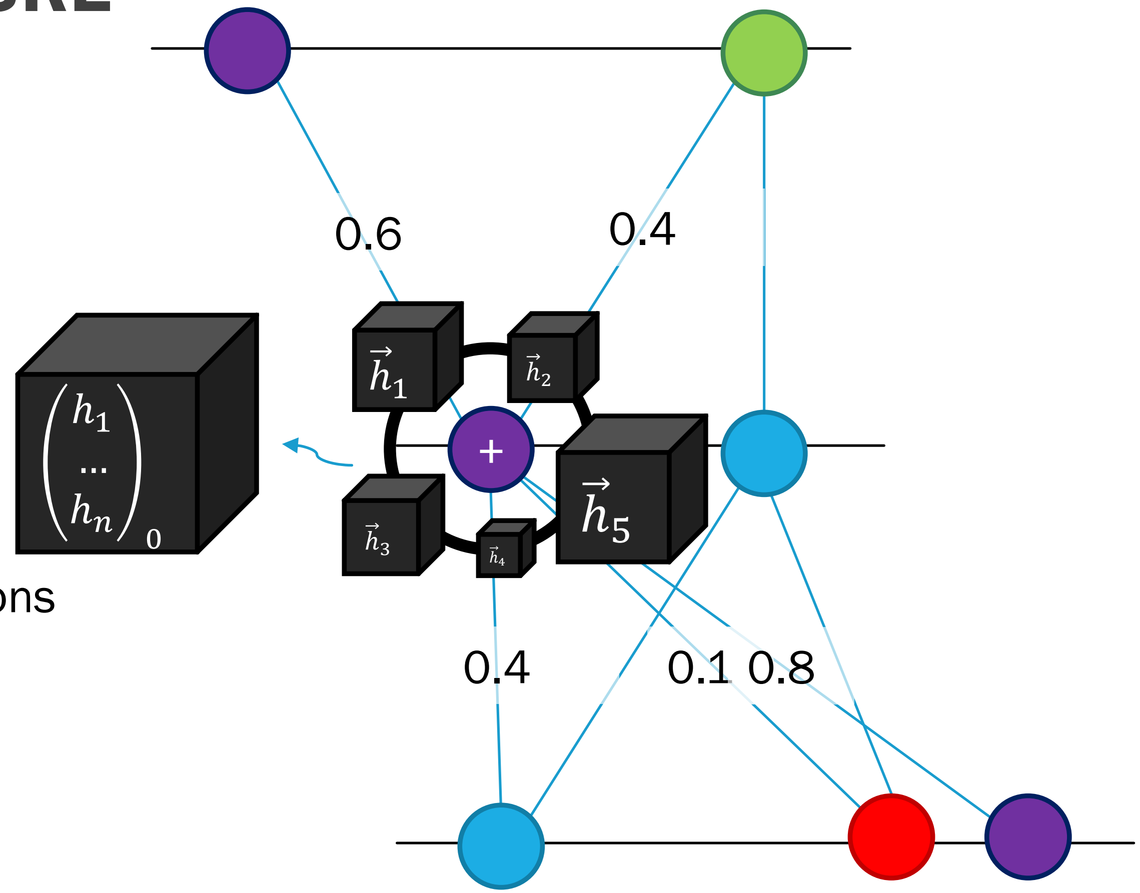
x n iterations



# EDGE PREDICTION ARCHITECTURE

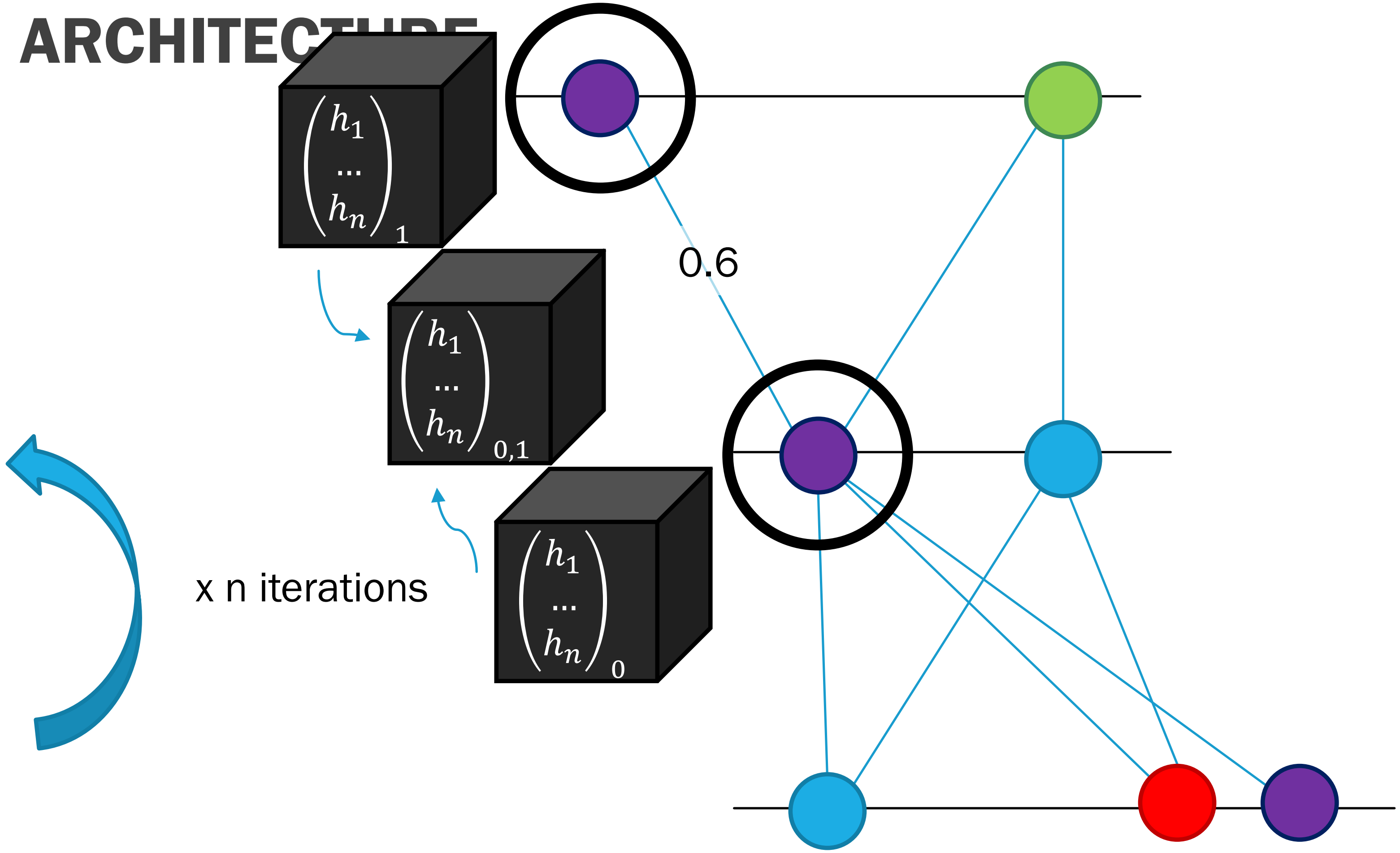
- Input node features
- Hidden node features
- Hidden edge features
- Edge score
- **Attention aggregation**
- **New hidden node features**
- New hidden edge features
- New edge score

x n iterations



# EDGE PREDICTION ARCHITECTURE

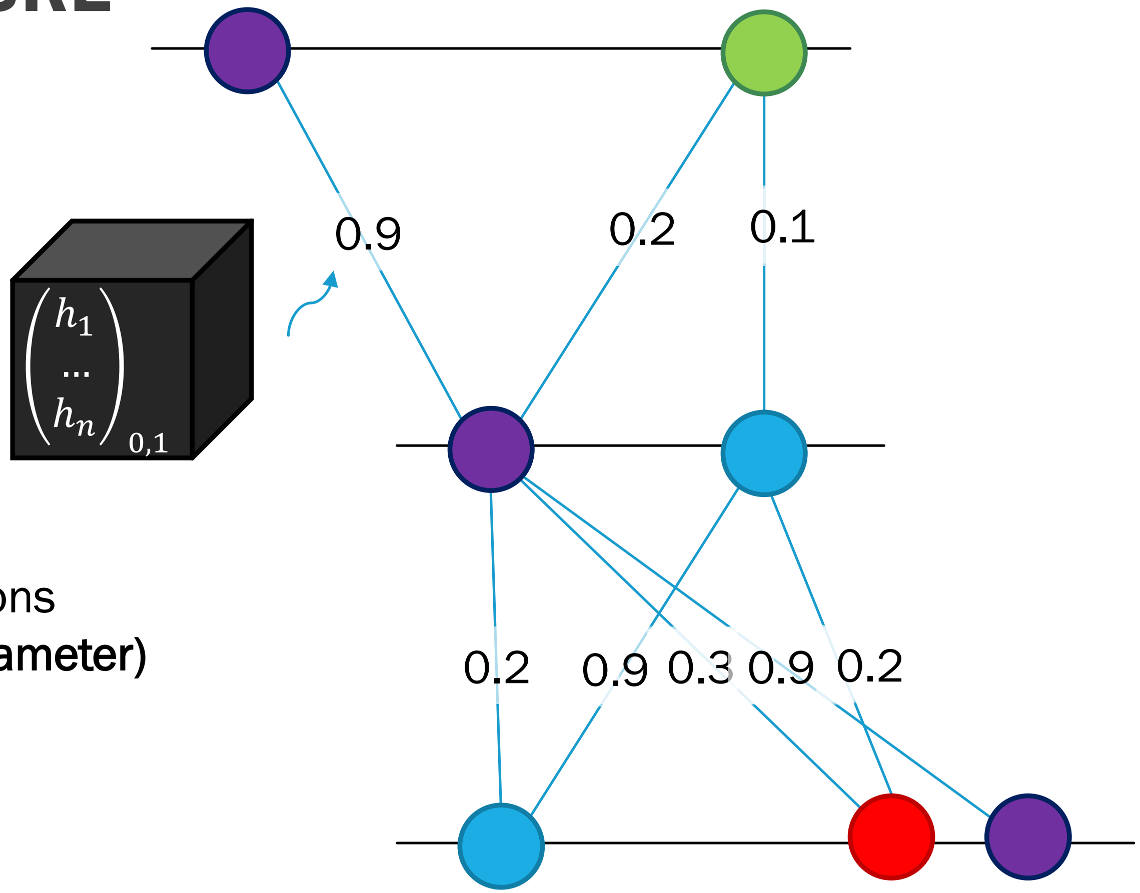
- Input node features
- Hidden node features
- Hidden edge features
- Edge score
- Attention aggregation
- New hidden node features
- **New hidden edge features**
- New edge score



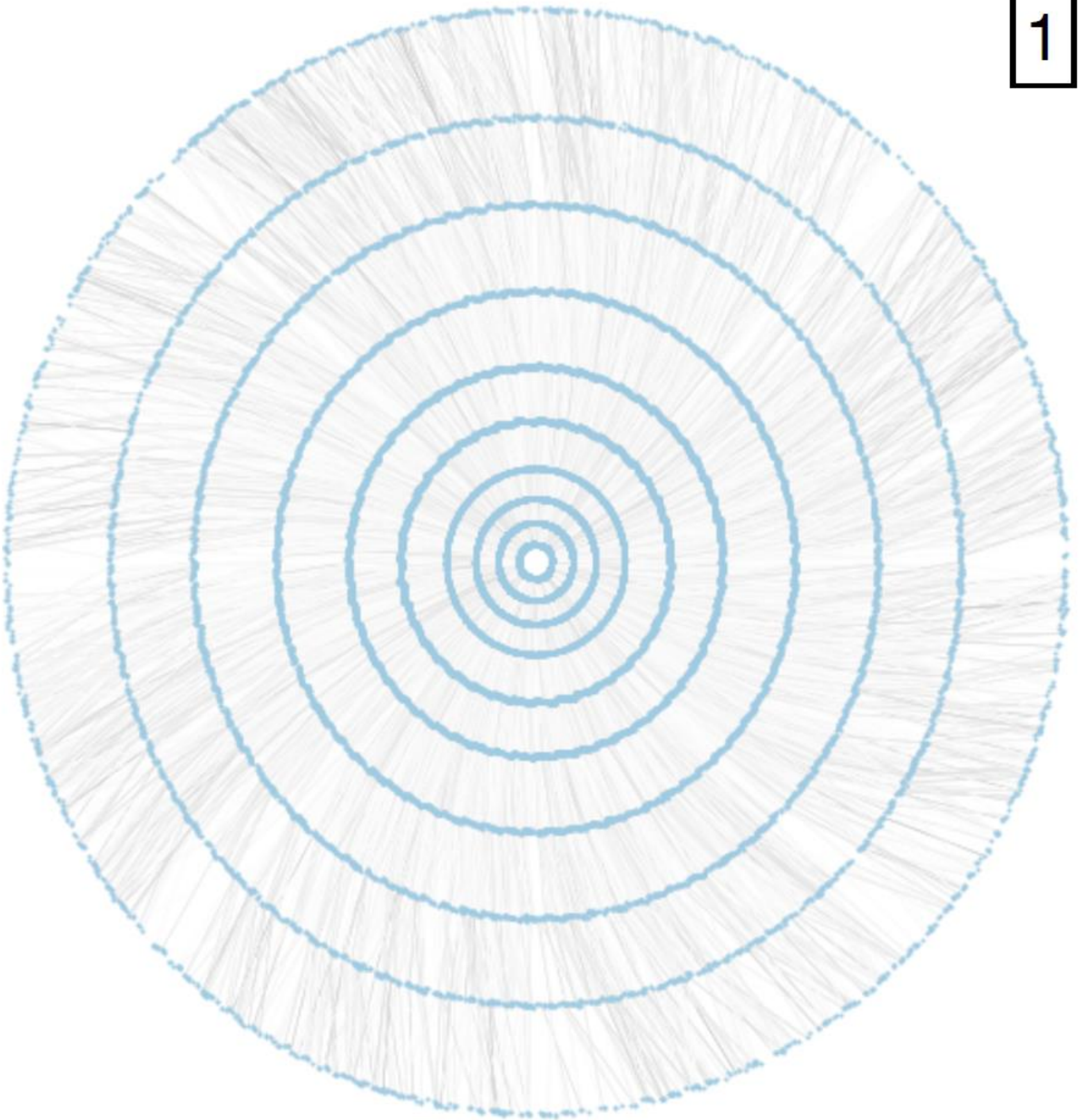
# EDGE PREDICTION ARCHITECTURE

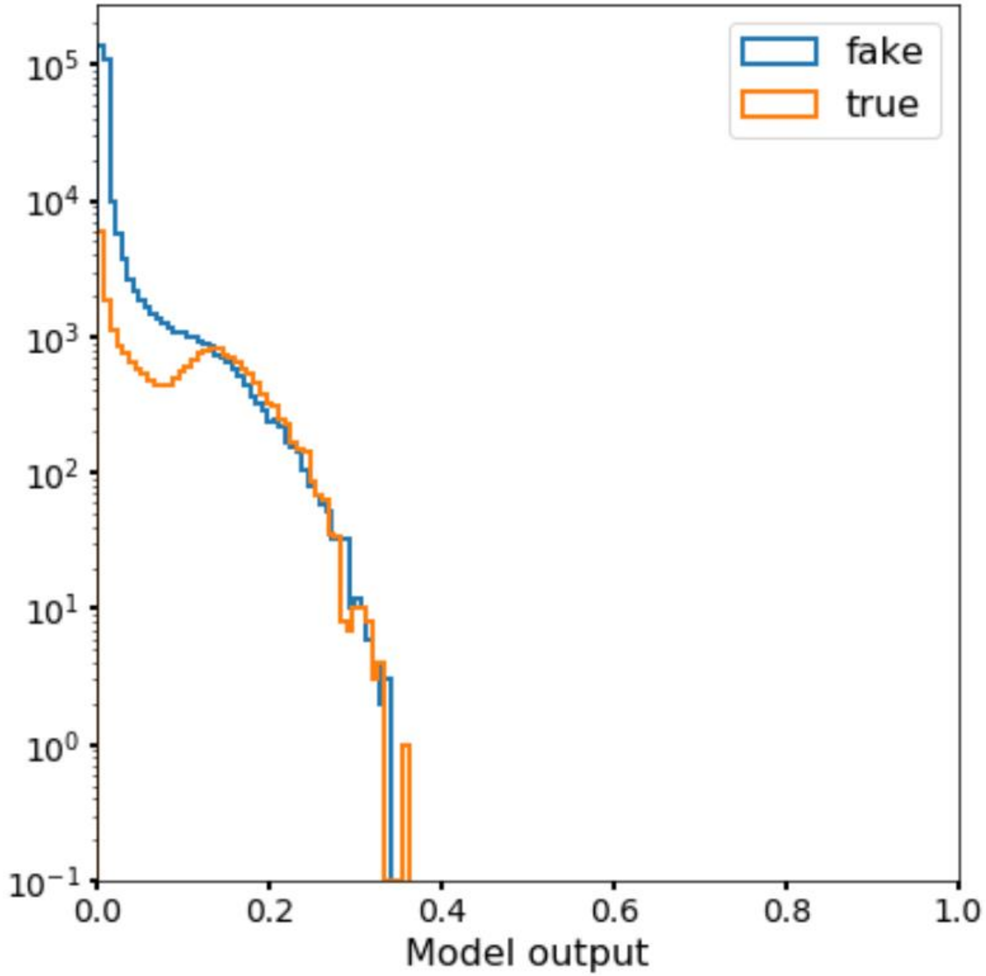
- Input node features
- Hidden node features
- Hidden edge features
- Edge score
- Attention aggregation
- New hidden node features
- New hidden edge features
- **New edge score**

x n iterations
**(hyperparameter)**



1

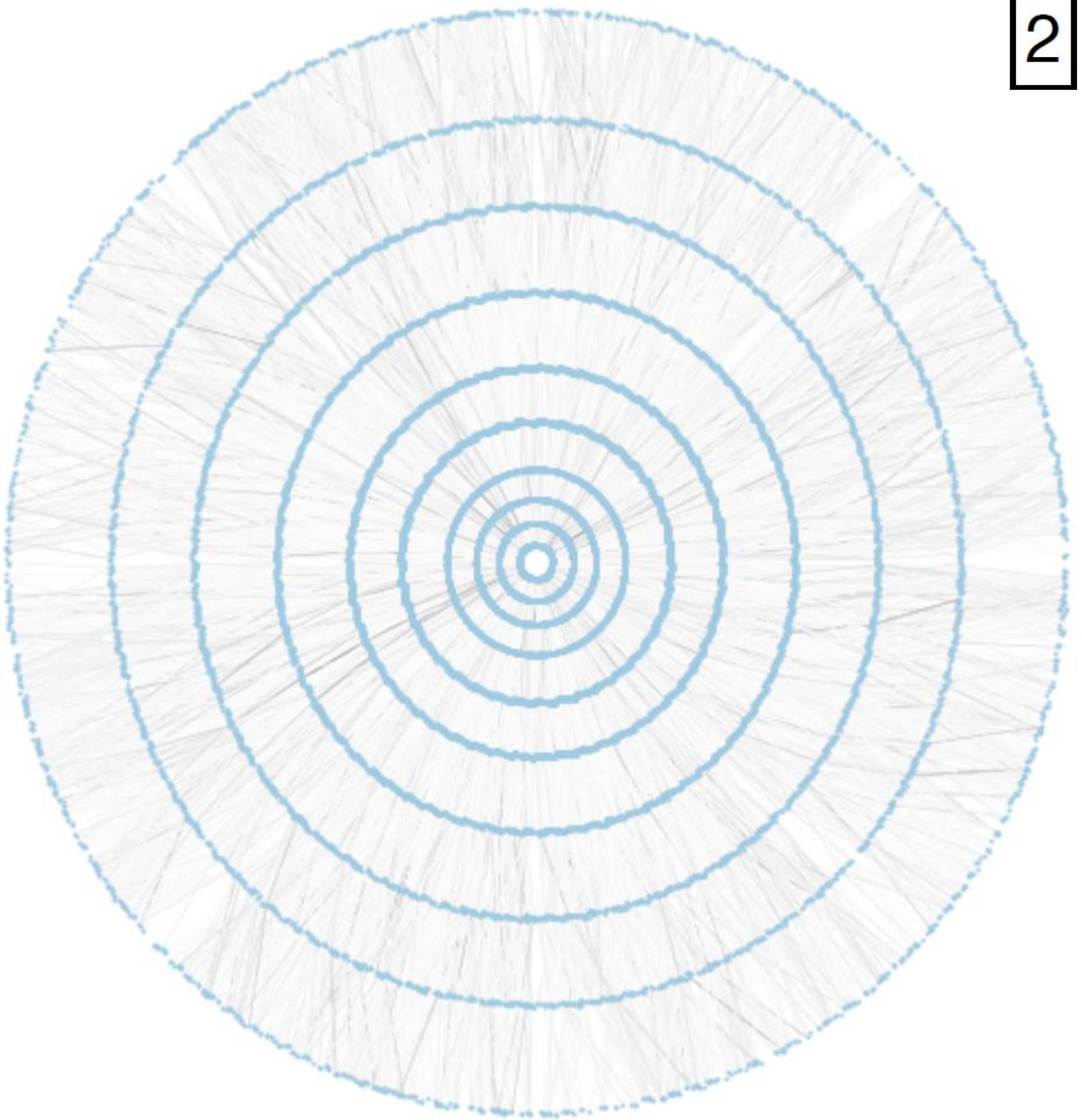


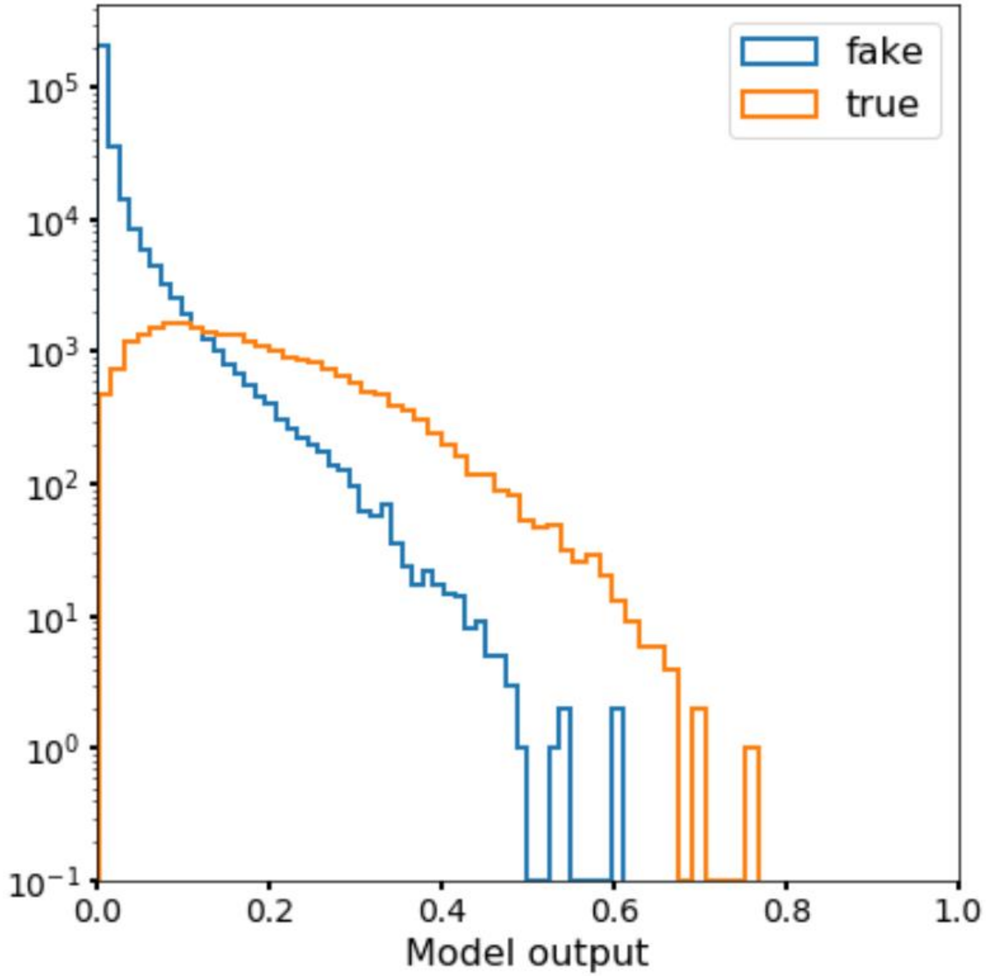



Edges with higher scores are darker than that with lower scores

Edges with scores < 0.01 are removed for visualization purpose.

2

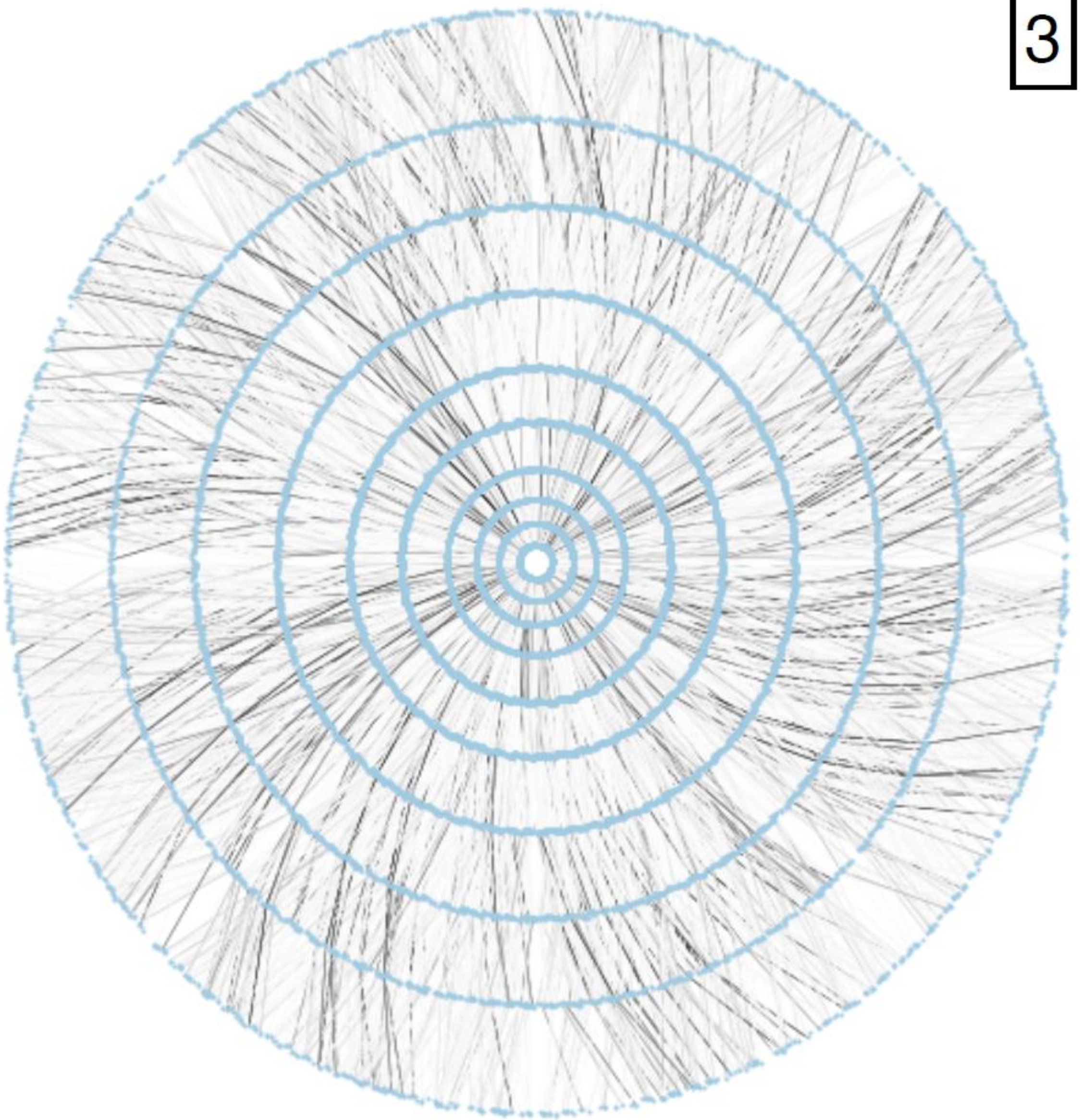


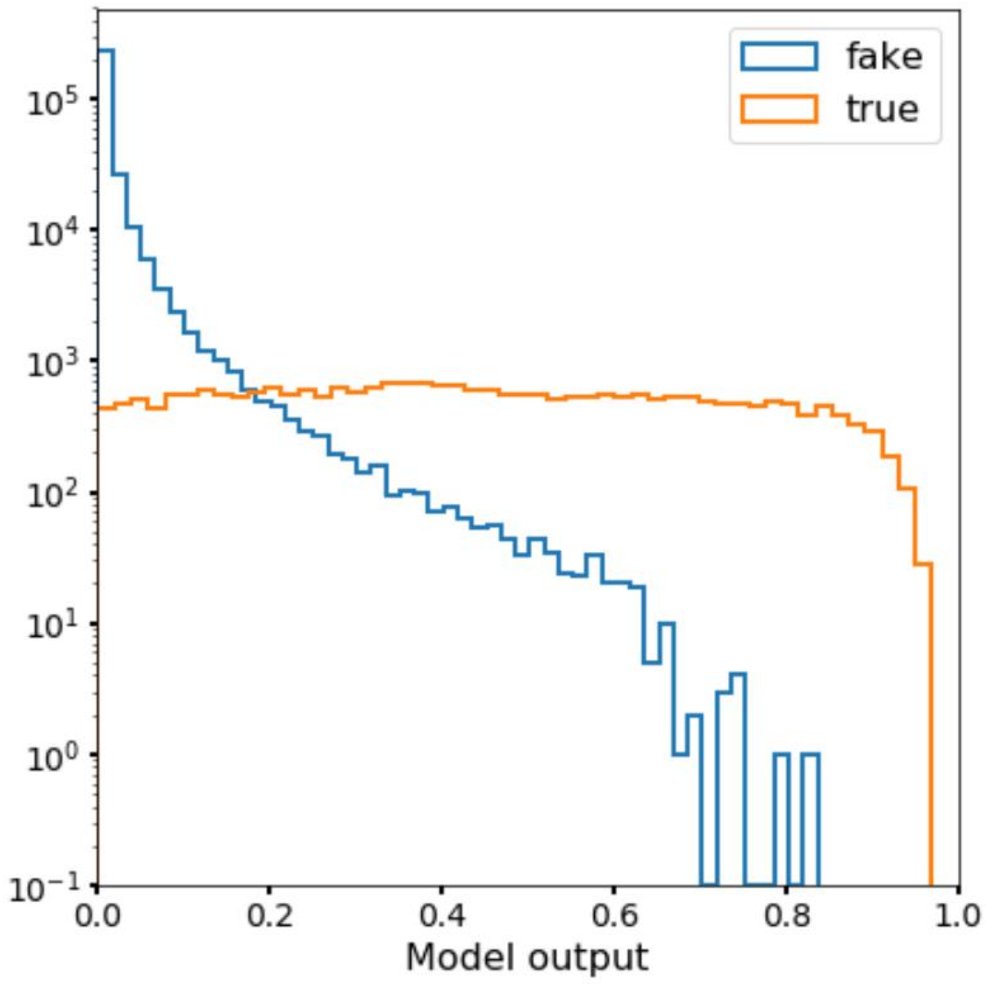



Edges with higher scores are darker than that with lower scores

Edges with scores < 0.01 are removed for visualization purpose.

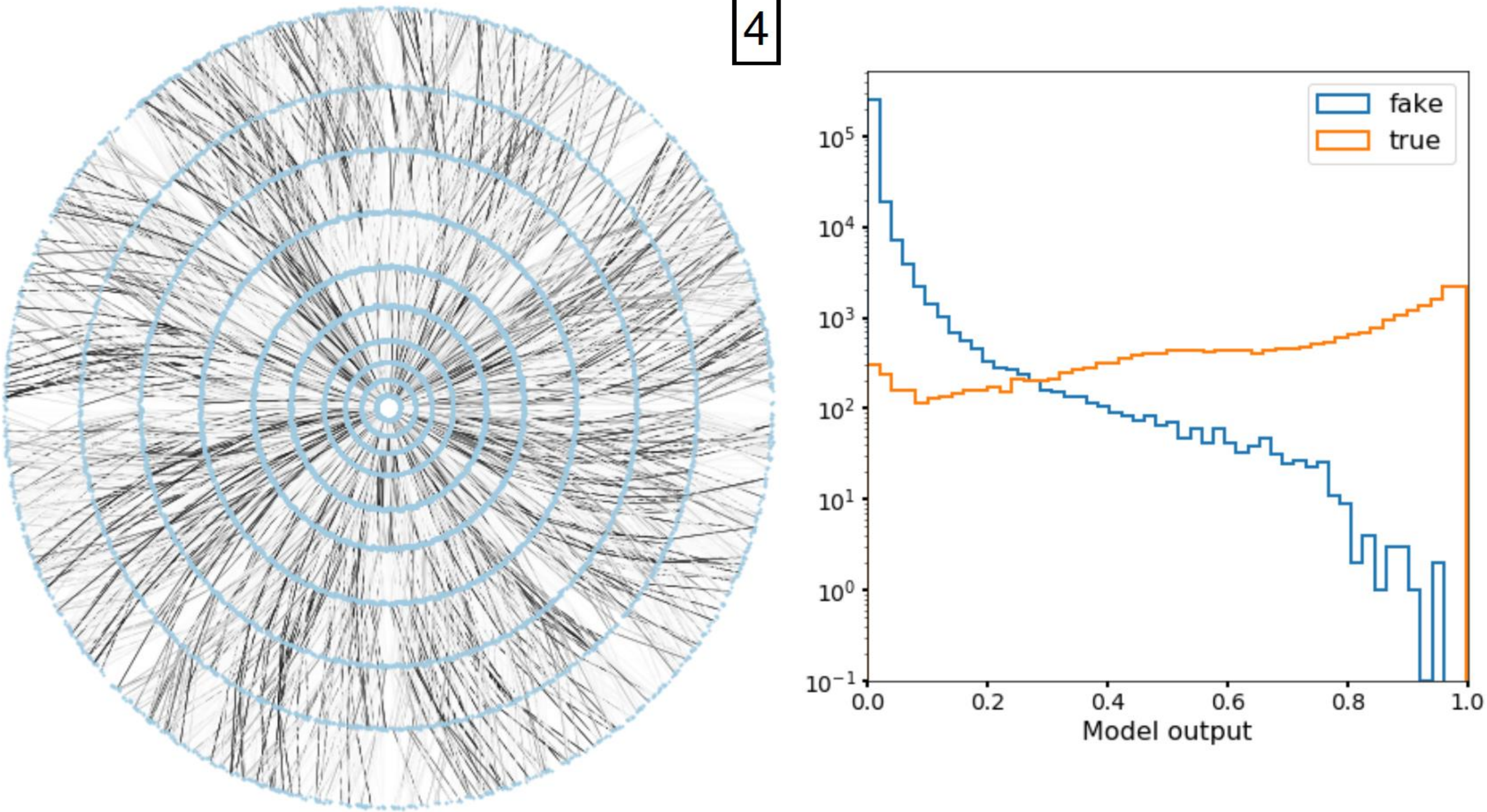
3





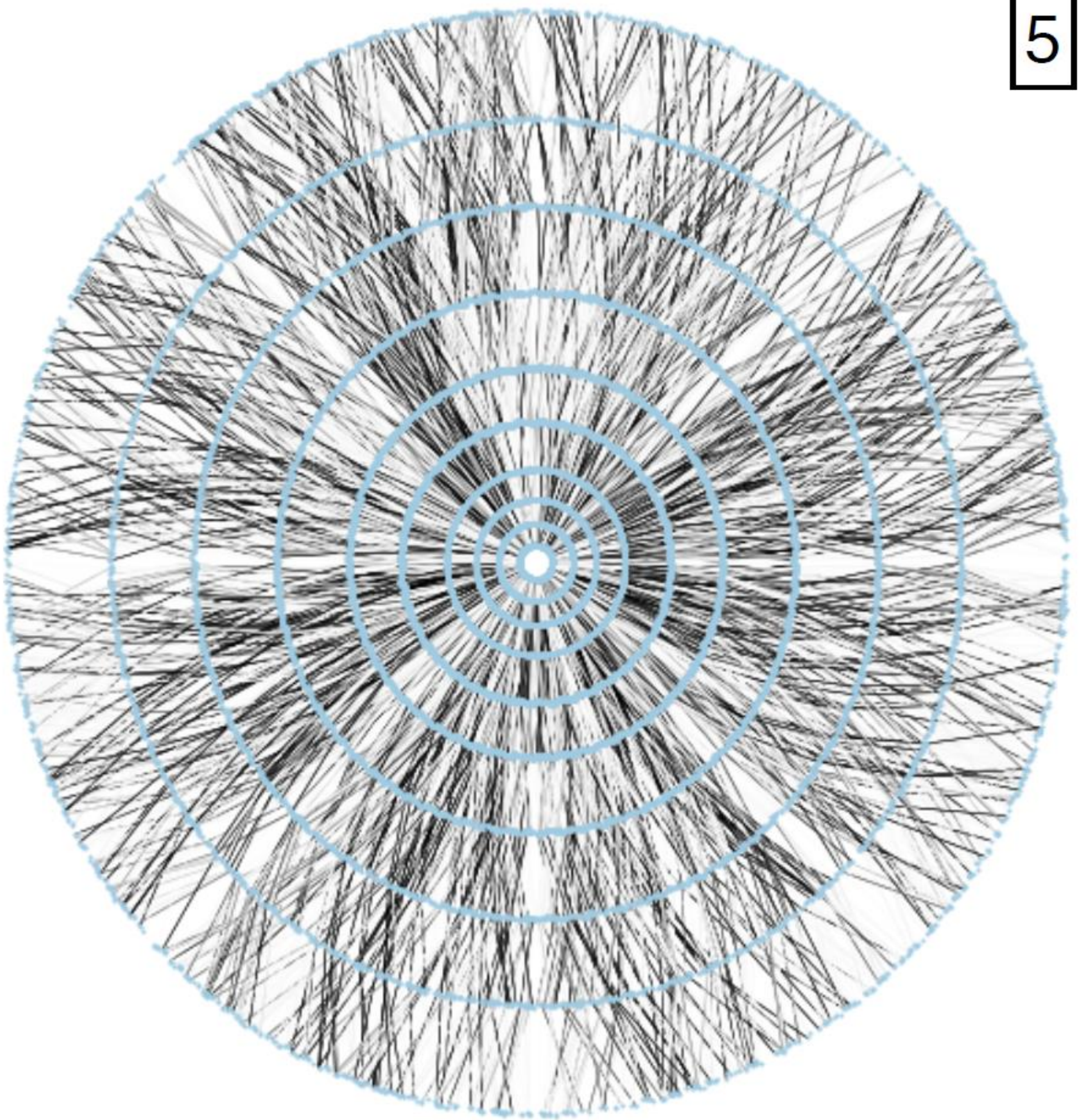

Edges with higher scores are darker than that with lower scores

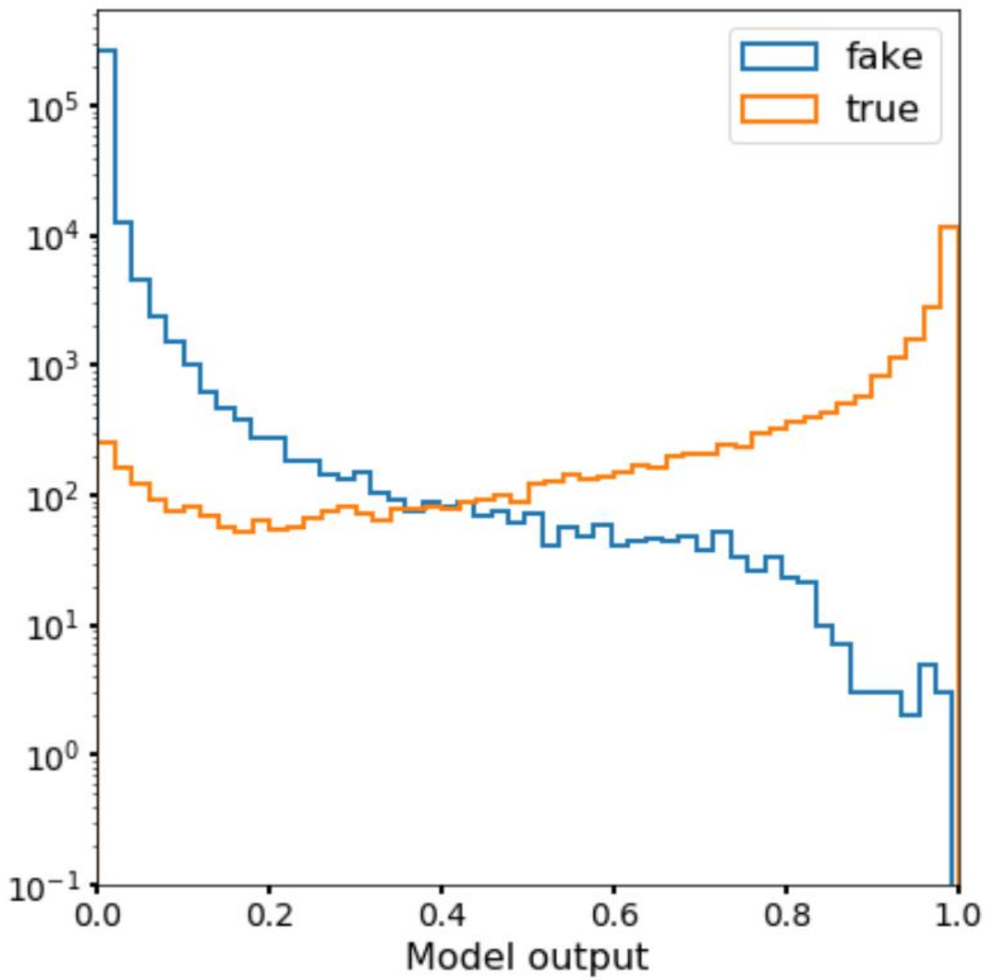
Edges with scores < 0.01 are removed for visualization purpose.




Edges with higher scores are darker than that with lower scores

Edges with scores < 0.01 are removed for visualization purpose.

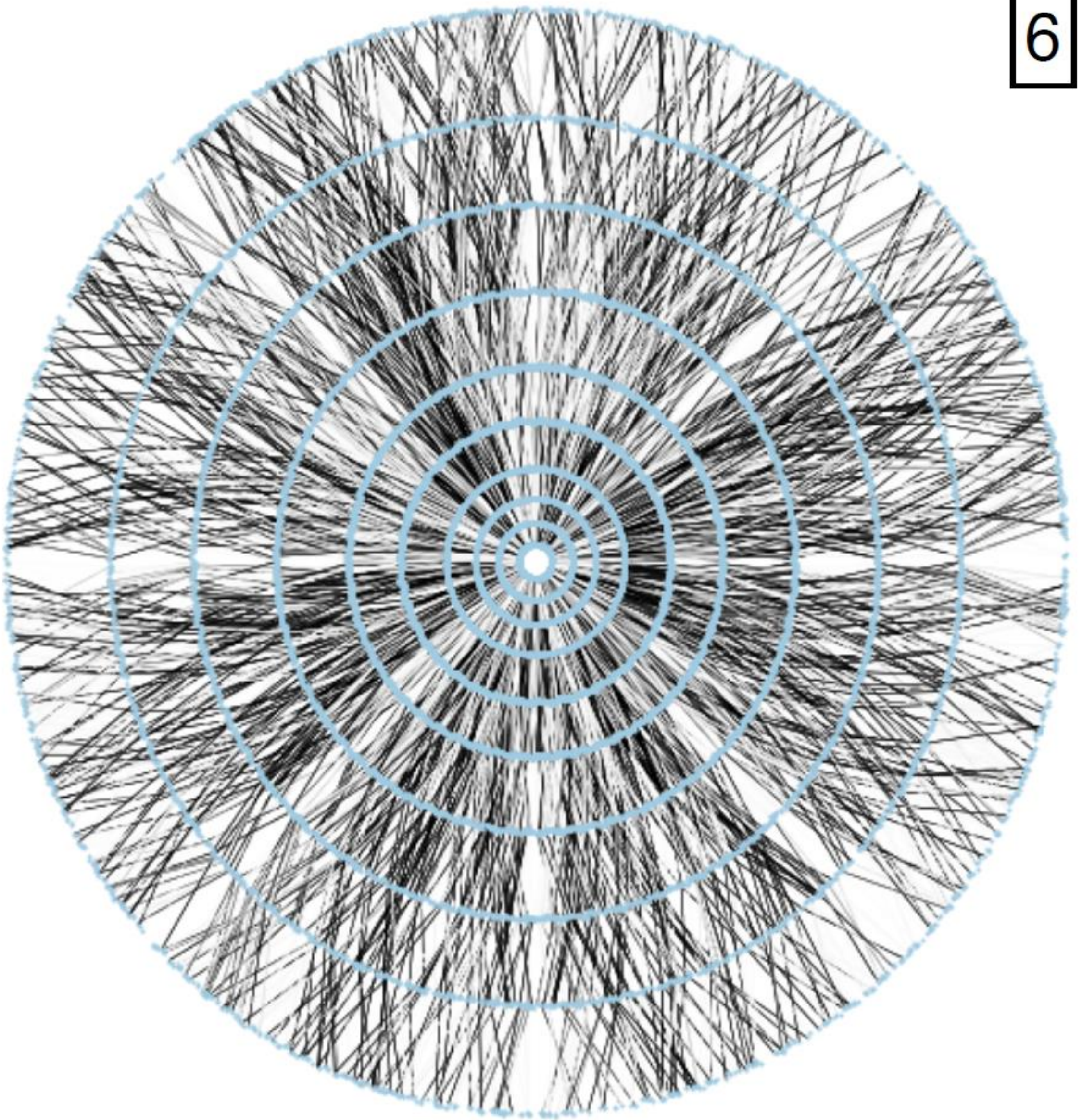


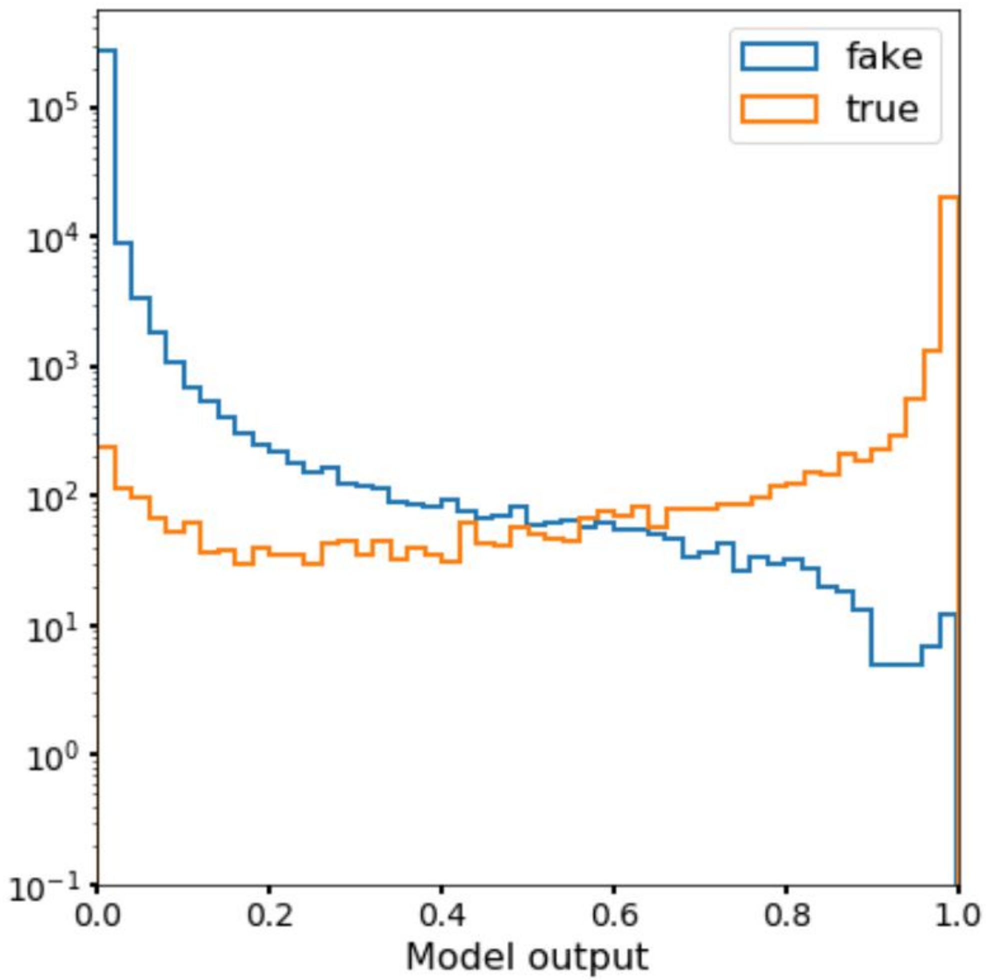



Edges with higher scores are darker than that with lower scores

Edges with scores < 0.01 are removed for visualization purpose.

6

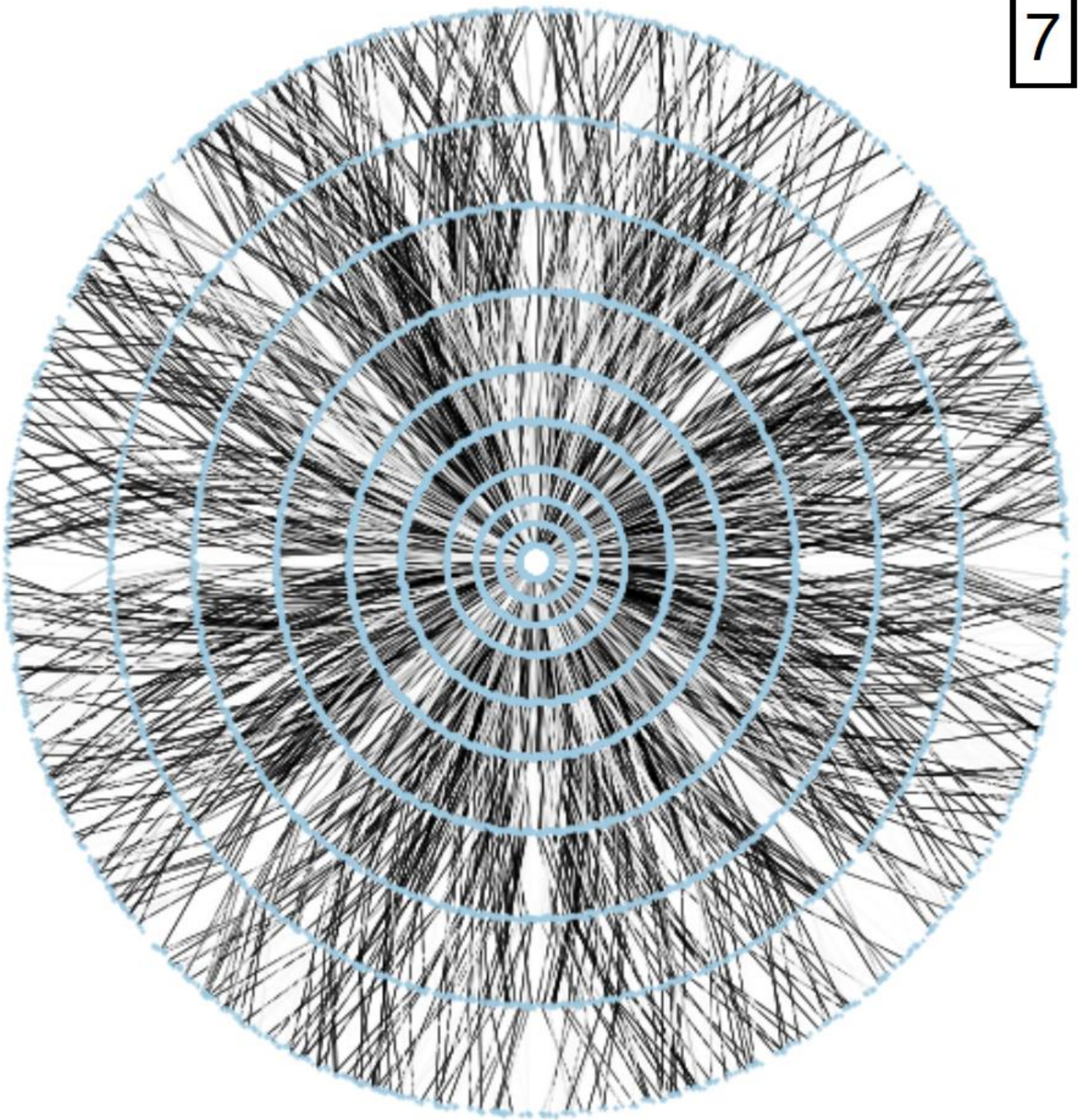


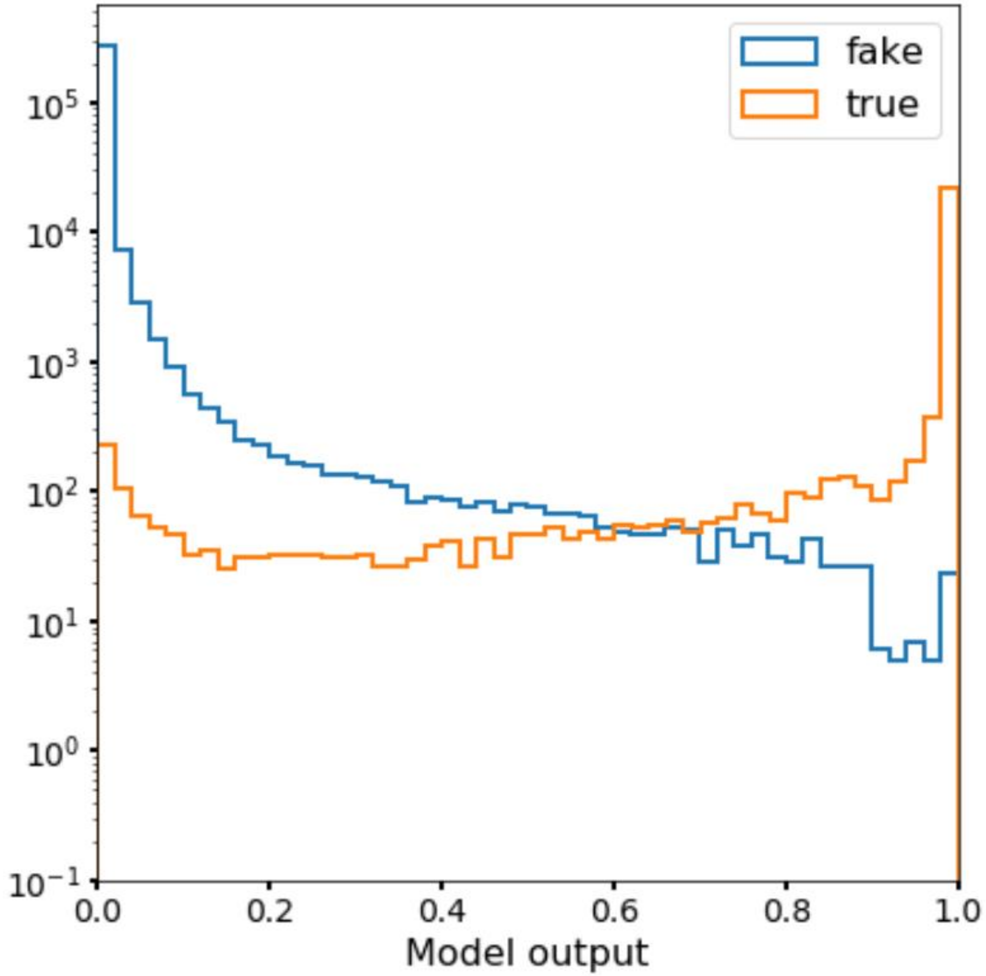



Edges with higher scores are darker than that with lower scores

Edges with scores < 0.01 are removed for visualization purpose.

7

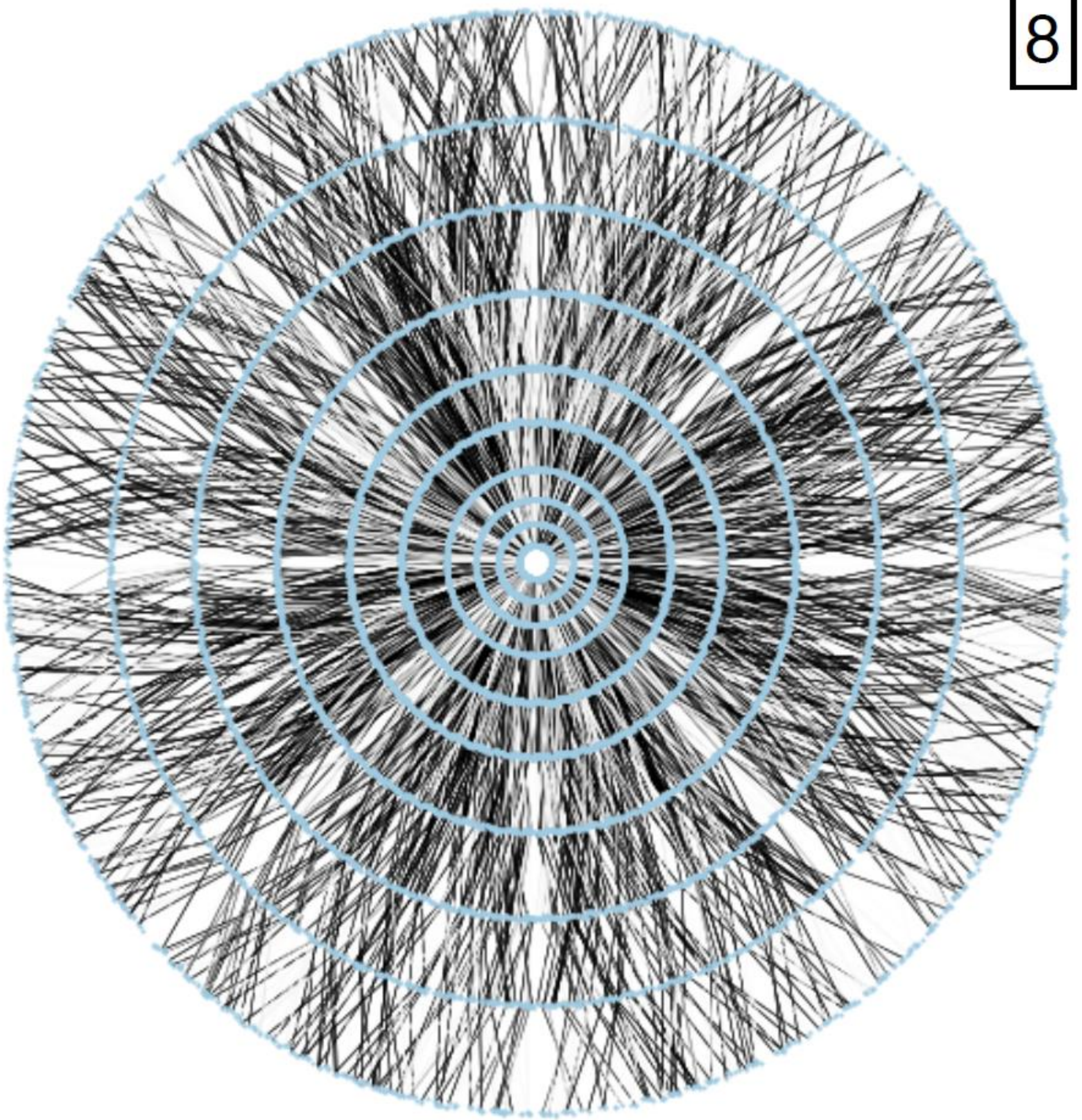


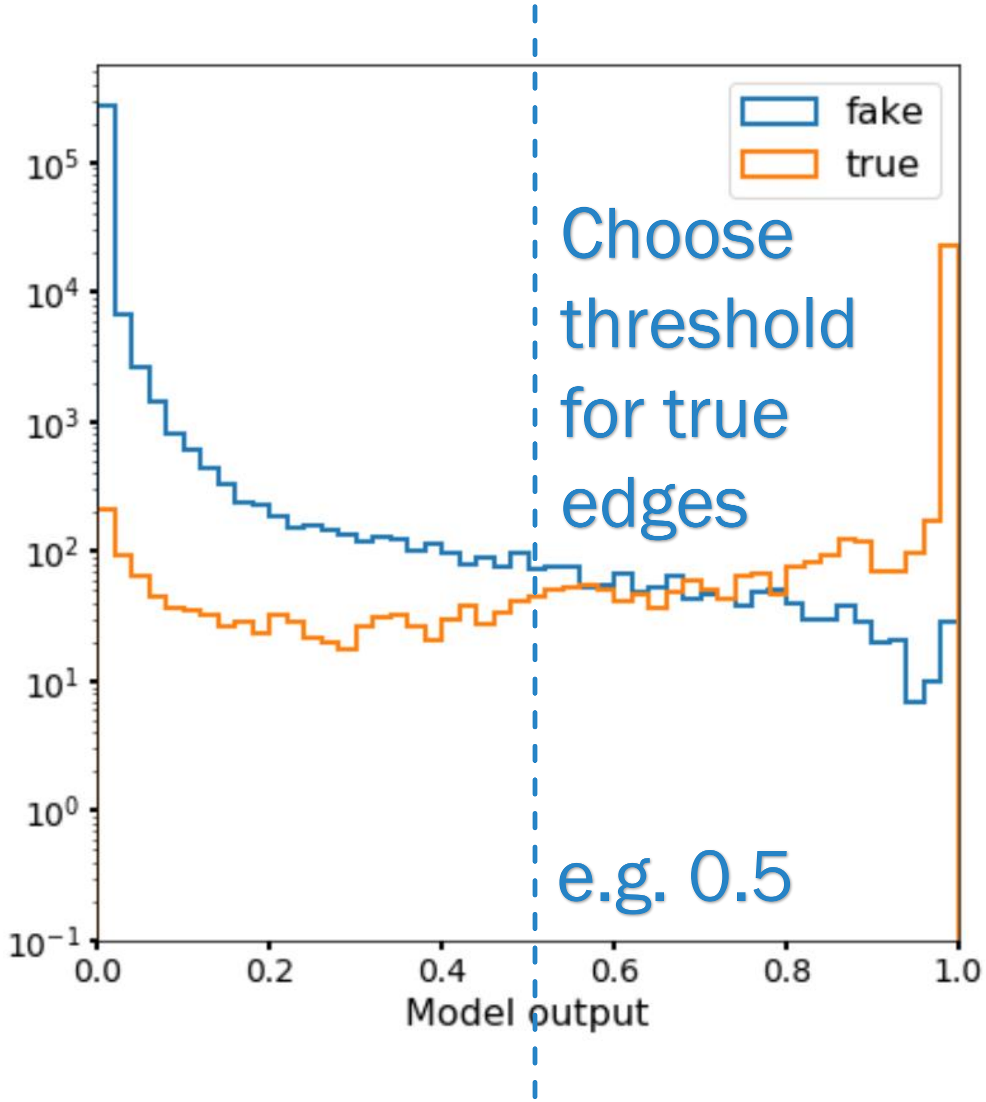



Edges with higher scores are darker than that with lower scores

Edges with scores < 0.01 are removed for visualization purpose.

8

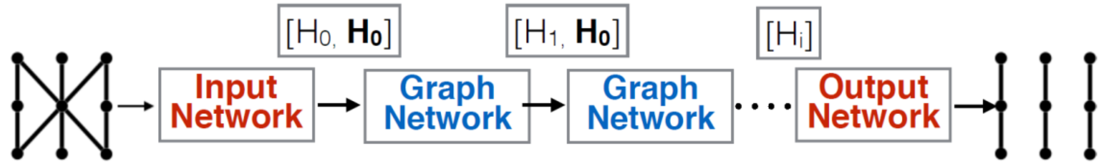




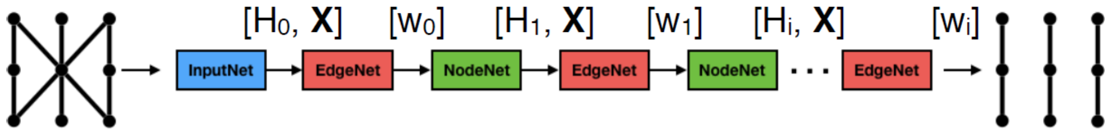

Edges with higher scores are darker than that with lower scores

Edges with scores < 0.01 are removed for visualization purpose.

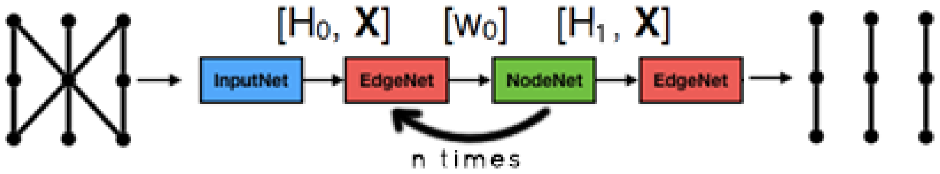
# EDGE PREDICTION ARCHITECTURE




- Message Passing

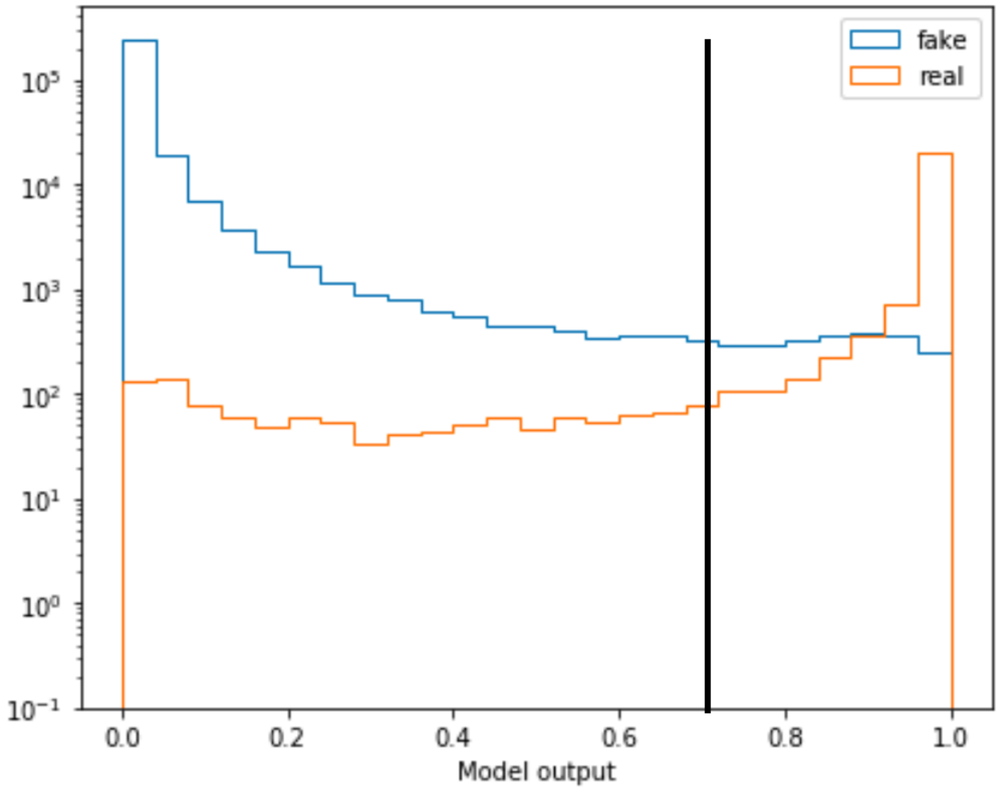



- Attention Message Passing




- Attention Message Passing with Recursion

# EDGE PREDICTION PERFORMANCE

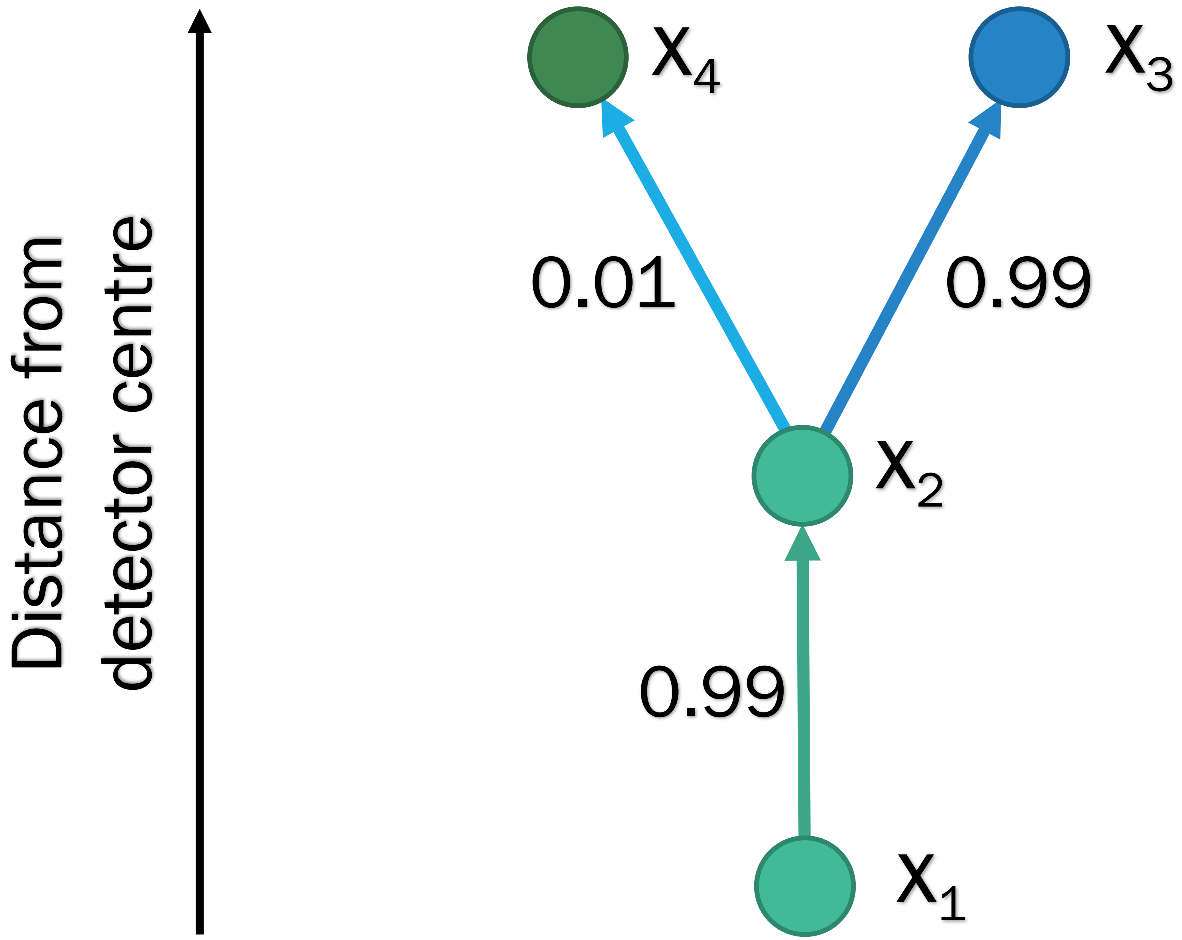



**Recursive attention GNN**

- ~ 43k parameters in Pytorch
- Trained on NVIDIA V100 GPU for ~ 60 epochs
- Binary logit loss function
- With "truth" cut-off of 0.7
  - Edge efficiency: 95.2%
  - Edge purity: 90.2%

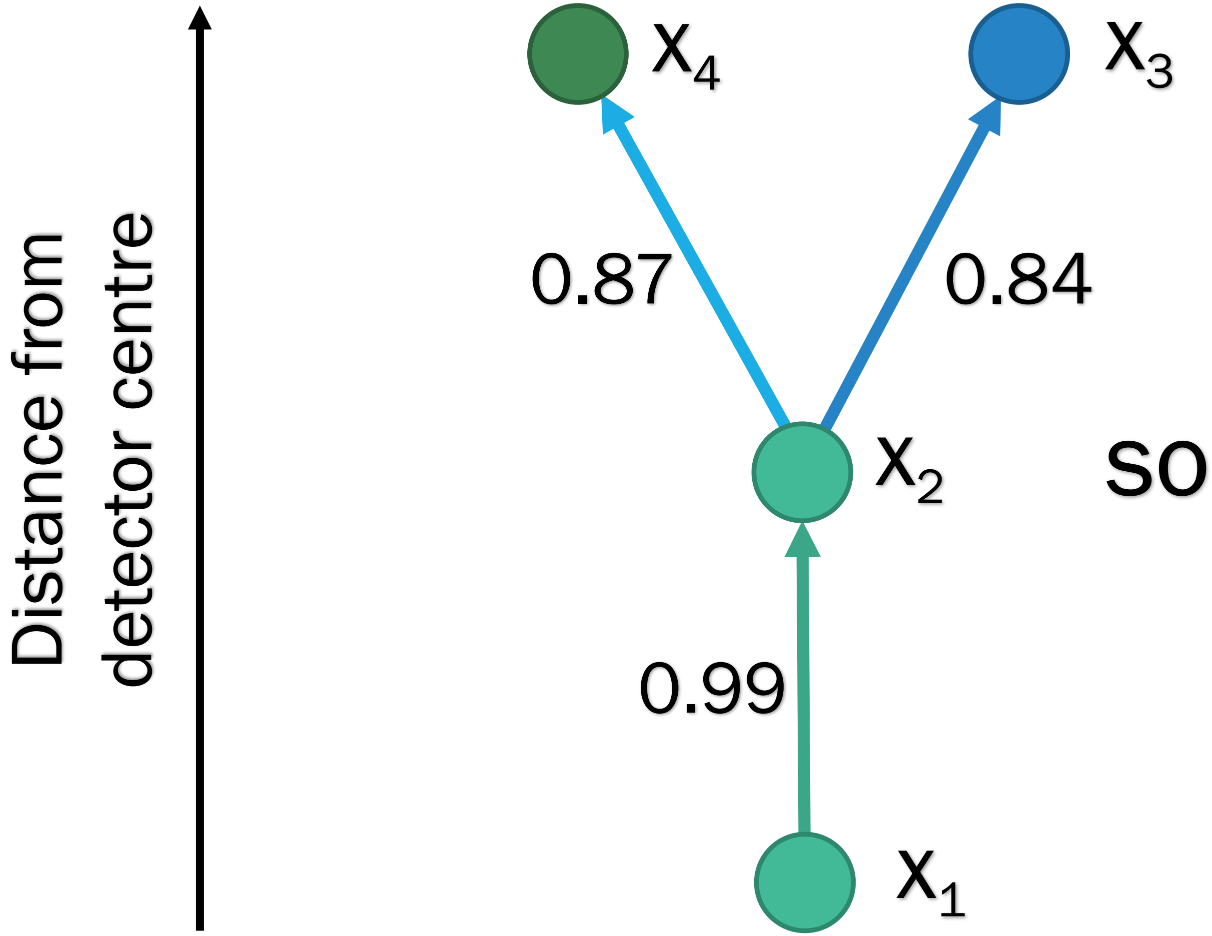
# FROM DOUBLETS TO TRIPLETS…

# WHY NOT SIMPLY JOIN TOGETHER OUR DOUBLET PREDICTIONS?



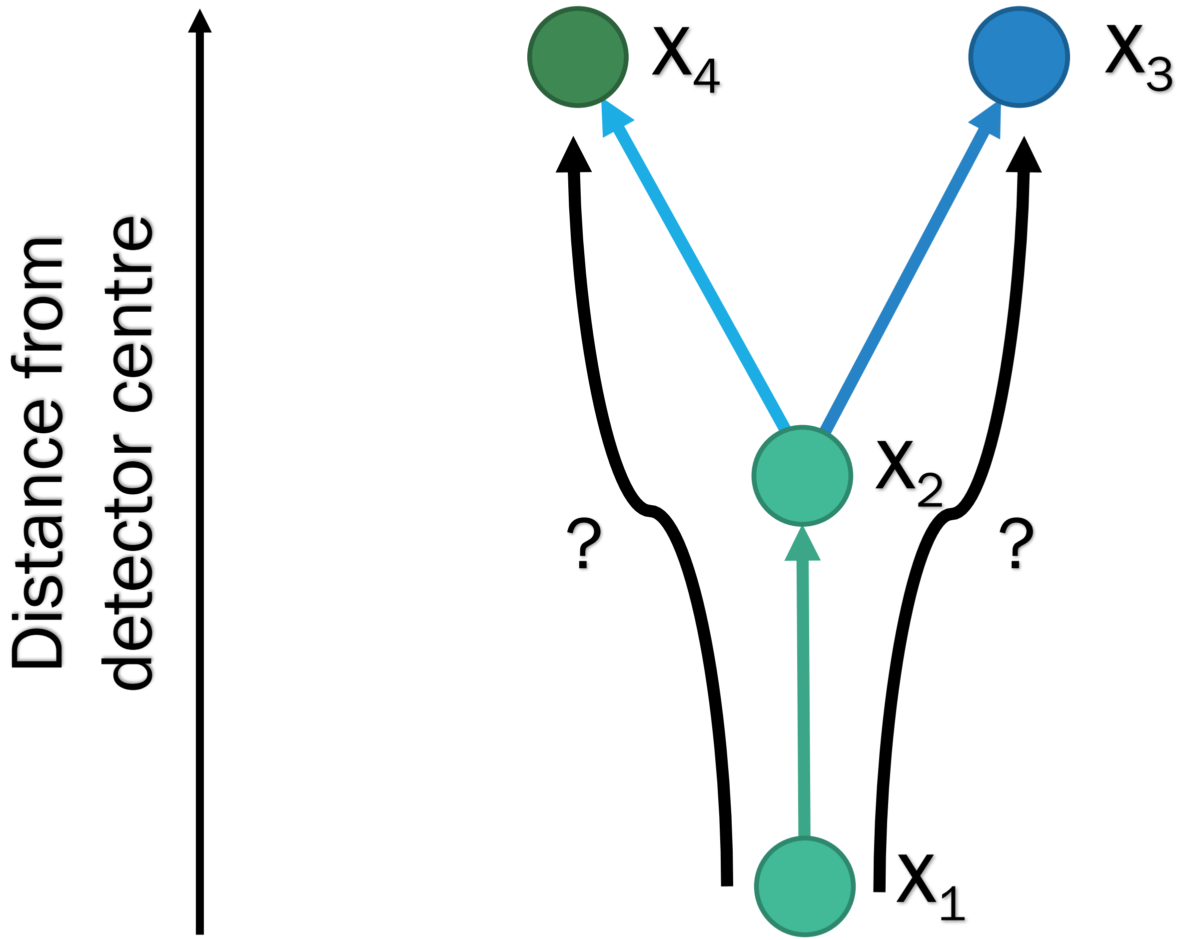

Pretty easy decision

# DOUBLET CHOICE CAN BE AMBIGUOUS



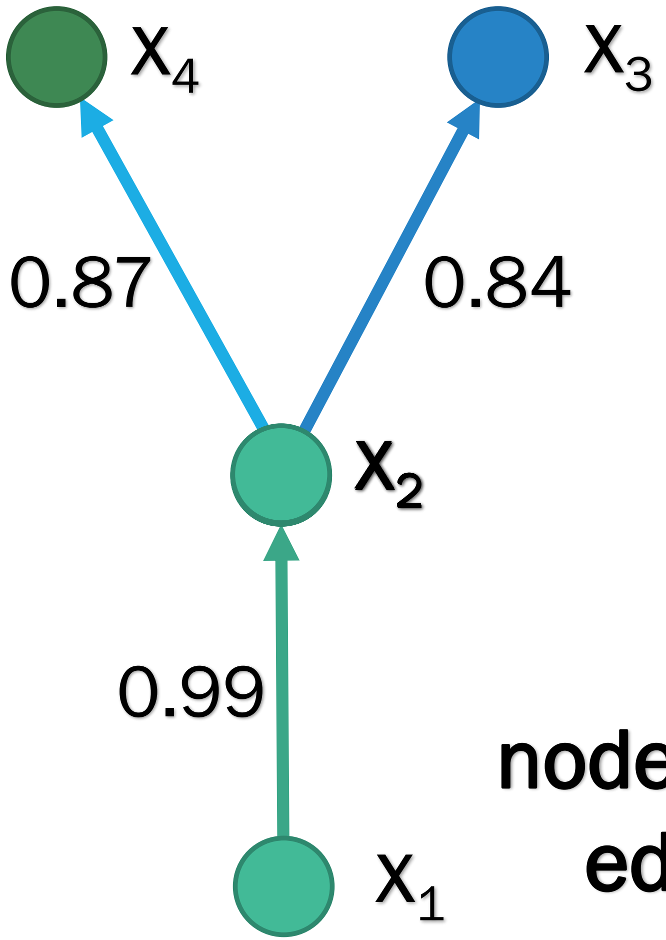

Not so easy...
so teach the network how to combine

# BUT A GNN DOESN'T KNOW ABOUT "TRIPLETS"



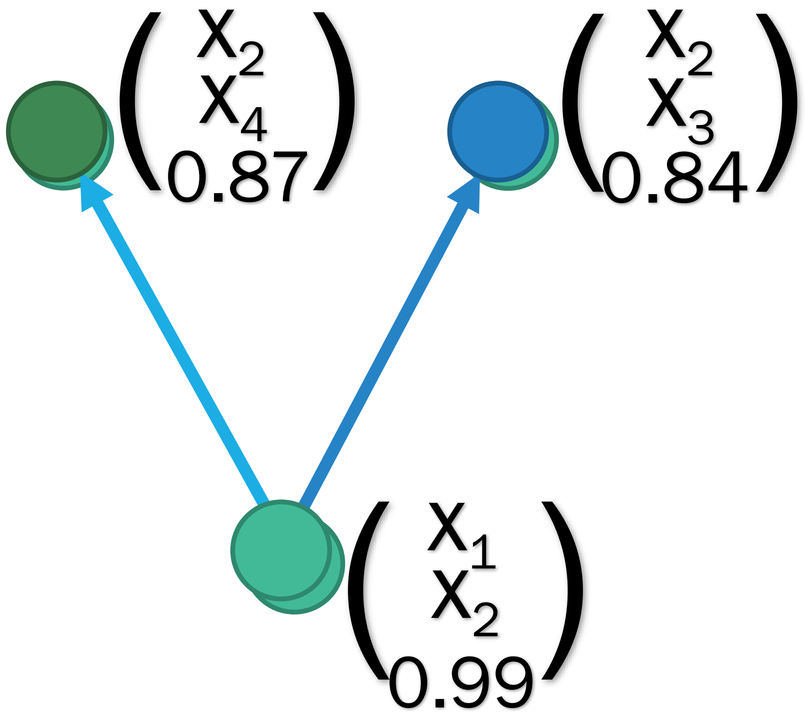

A GNN only knows about nodes and edge

# MOVING TO A "DOUBLET GRAPH" GIVES US BACK GNN POWER




Now…
**nodes** represent **doublets**,
**edges** represent **triplets**

# MOVING TO A "DOUBLET GRAPH" GIVES US BACK GNN POWER




Now…
**nodes** represent **doublets,**
**edges** represent **triplets**

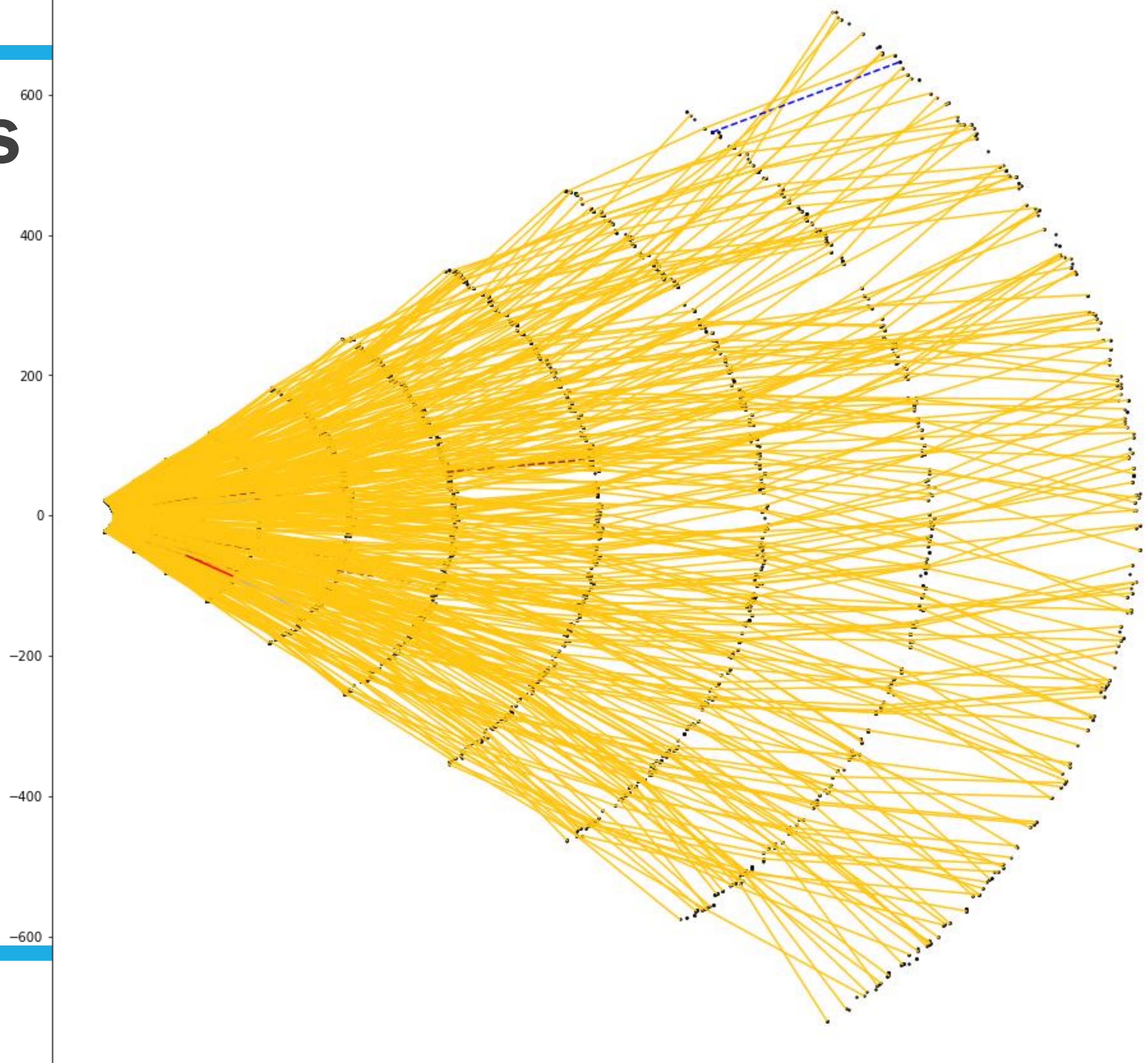
## THE TRIPLET CLASSIFIER RUNS WITH ALL THE BENEFITS OF THE DOUBLET CLASSIFIER

- Aim is to beat all traditional methods of finding true triplets
- Can then either continue to 4, 5, …-plets in order to create and end-to-end GNN track builder…
- …or hand off the triplets as seeds to the traditional techniques, knowing we can be confident in their accuracy

# TRIPLET GNN PERFORMS VERY WELL

- **Gold**: Unambiguously correct triplet or quadruplet
- **Other colours**: False positive/negative

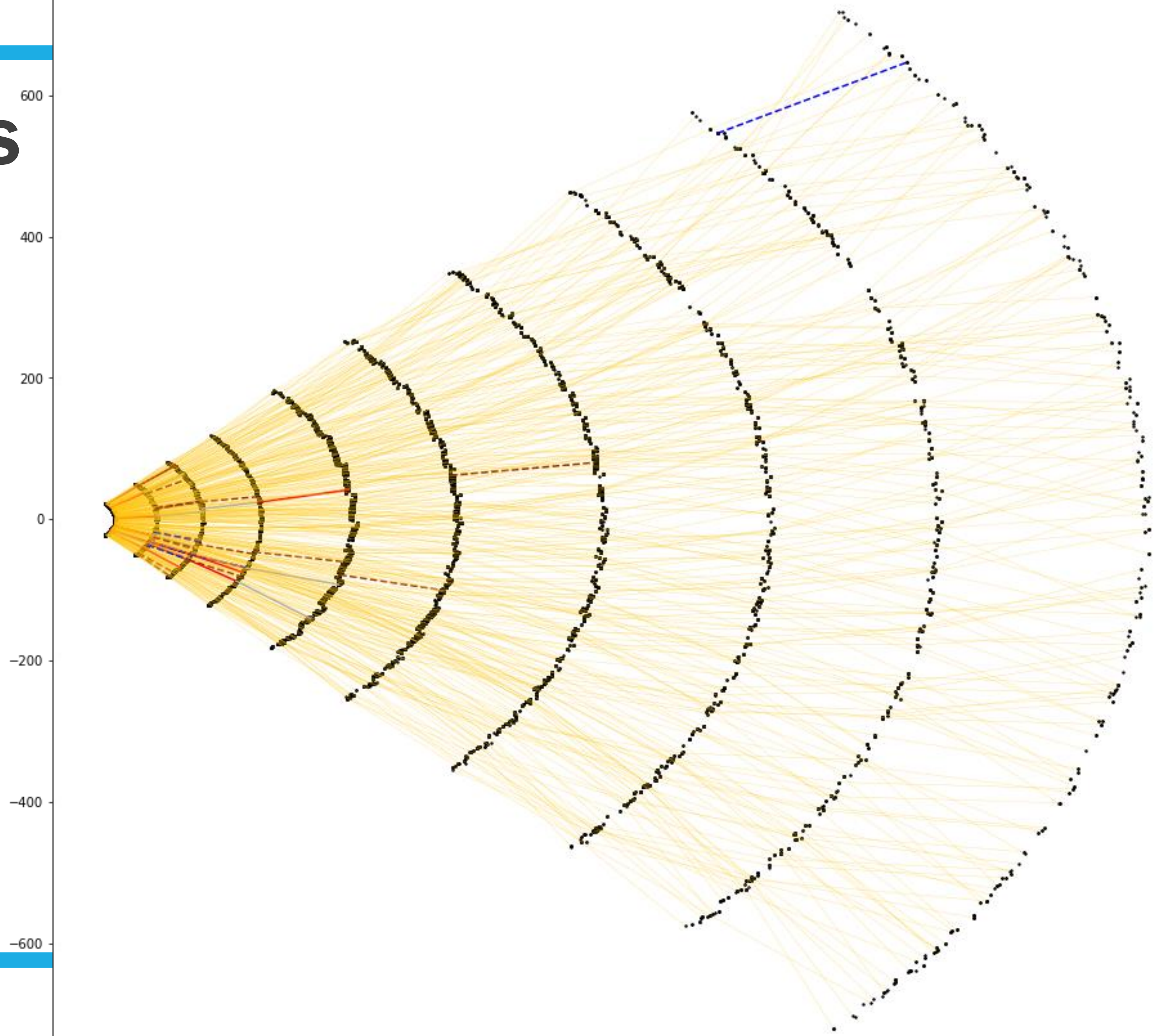
- **Key**:
- Silver: Ambiguously correct triplet or quadruplet (i.e. edge shared by correct triplet and false positive triplet)
- Bronze dashed: Correct triplet, but missed quadruplet (i.e. edge shared by correct triplet and false negative triplet)
- Red: Completely false positive triplet
- Blue dashed: Completely false negative triplet



# TRIPLET GNN PERFORMS VERY WELL

- **Gold**: Unambiguously correct triplet or quadruplet
- **Other colours**: False positive/negative

- **Key**:
- Silver: Ambiguously correct triplet or quadruplet (i.e. edge shared by correct triplet and false positive triplet)
- Bronze dashed: Correct triplet, but missed quadruplet (i.e. edge shared by correct triplet and false negative triplet)
- Red: Completely false positive triplet
- Blue dashed: Completely false negative triplet

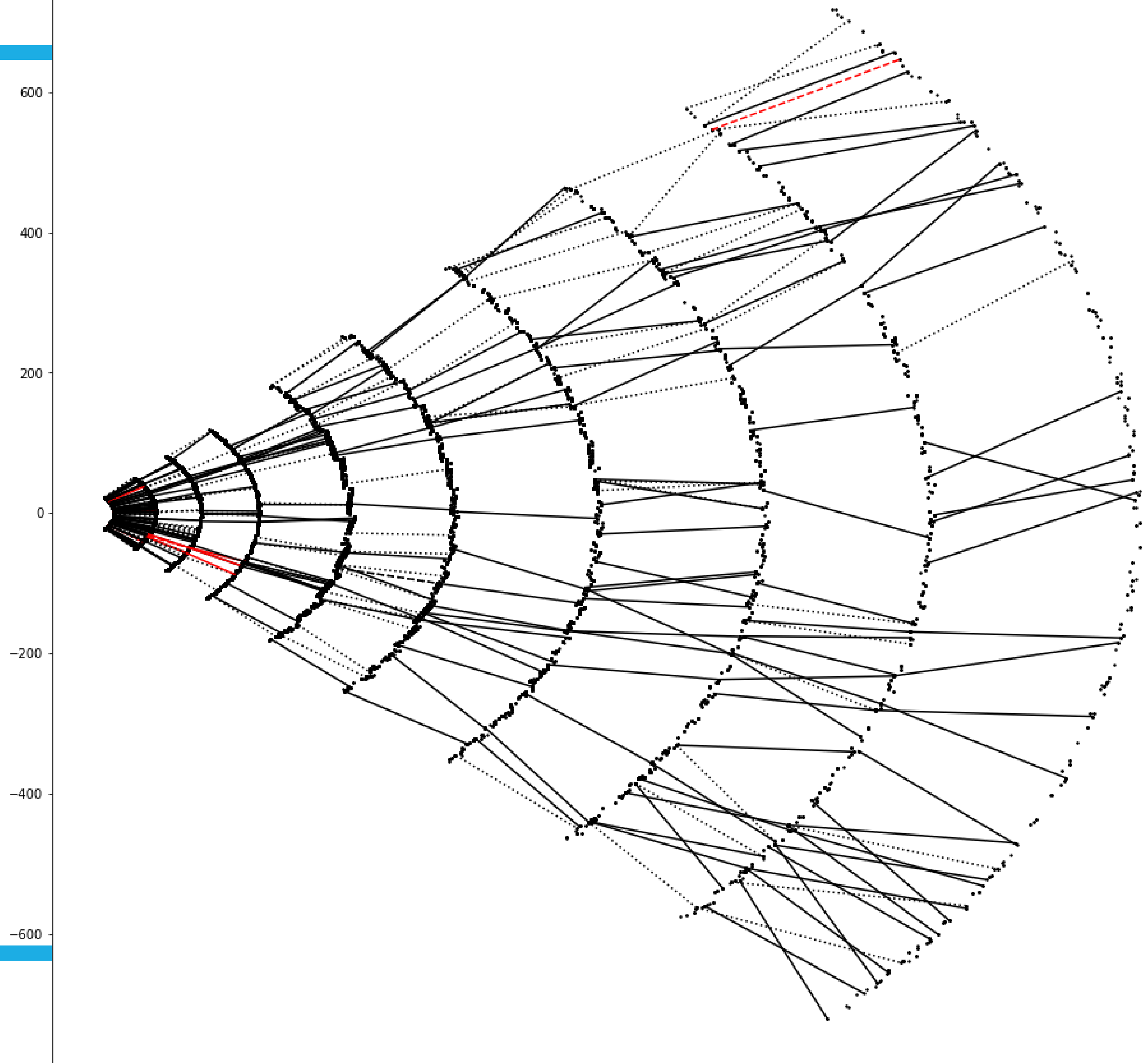


# TRIPLET GNN IMPROVES DOUBLET GNN RESULTS

- **Black:** Triplet classifier correctly labelled, doublet classifier mislabelled
- **Red:** Doublet classifier correctly labelled, triplet classifier mislabelled

In this graph, triplet classifier

- Fixes 389 edges
- Worsens 10 edges



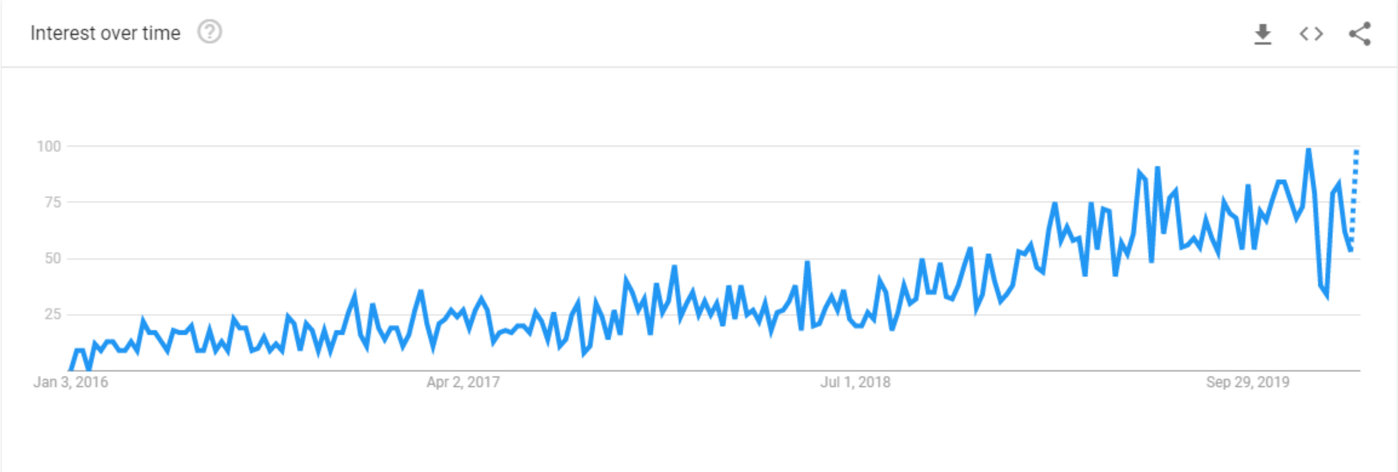
# PART II: ML PIPELINE AND ARCHITECTURE DESIGN WITH GRAPHS

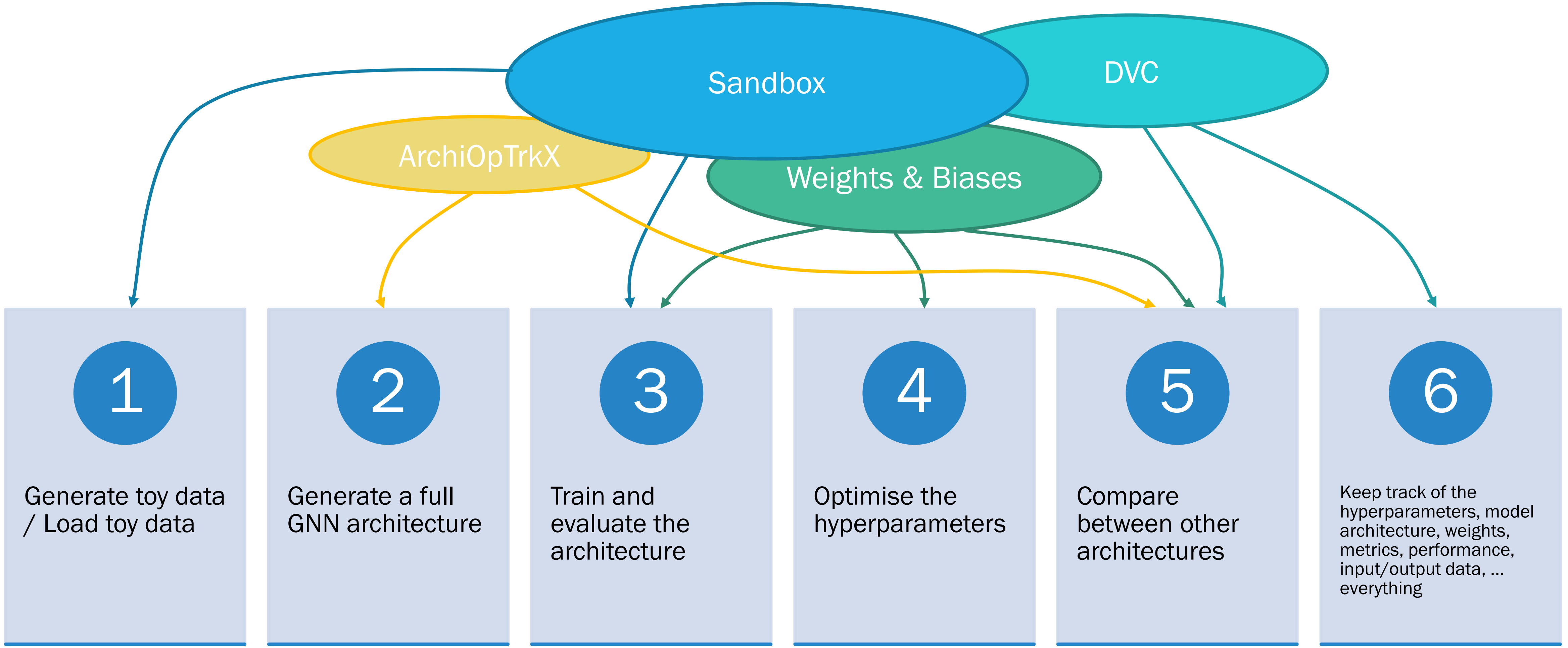
# GNN SANDBOX

- Problem statement: There are many possible input/output node/edge/graph features, there are many GNN architectures each working on different types of features, each architecture needs HPO to understand its strength/weakness
- Would like to create a GNN sandbox – a pipeline that is modular and reproducible that can both manually and automatically
  1. Generate toy data / Load toy data
  2. Generate a full GNN architecture
  3. Train and evaluate the architecture
  4. Optimise the hyperparameters
  5. Compare between other architectures
  6. Keep track of the hyperparameters, model architecture, weights, metrics, performance, input/output data, … everything, so that any of these can be tweaked or optimised further with no extra coding/work

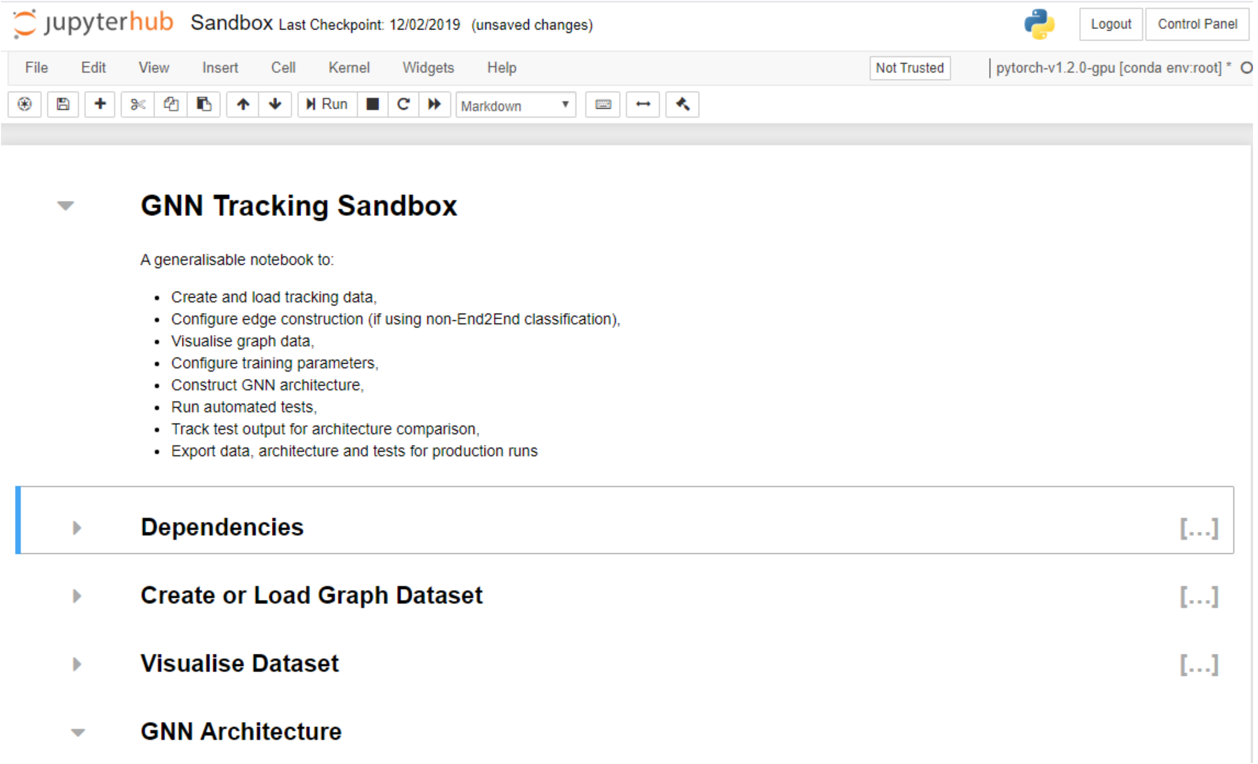
# TOO MANY GNNS TO CODE BY HAND

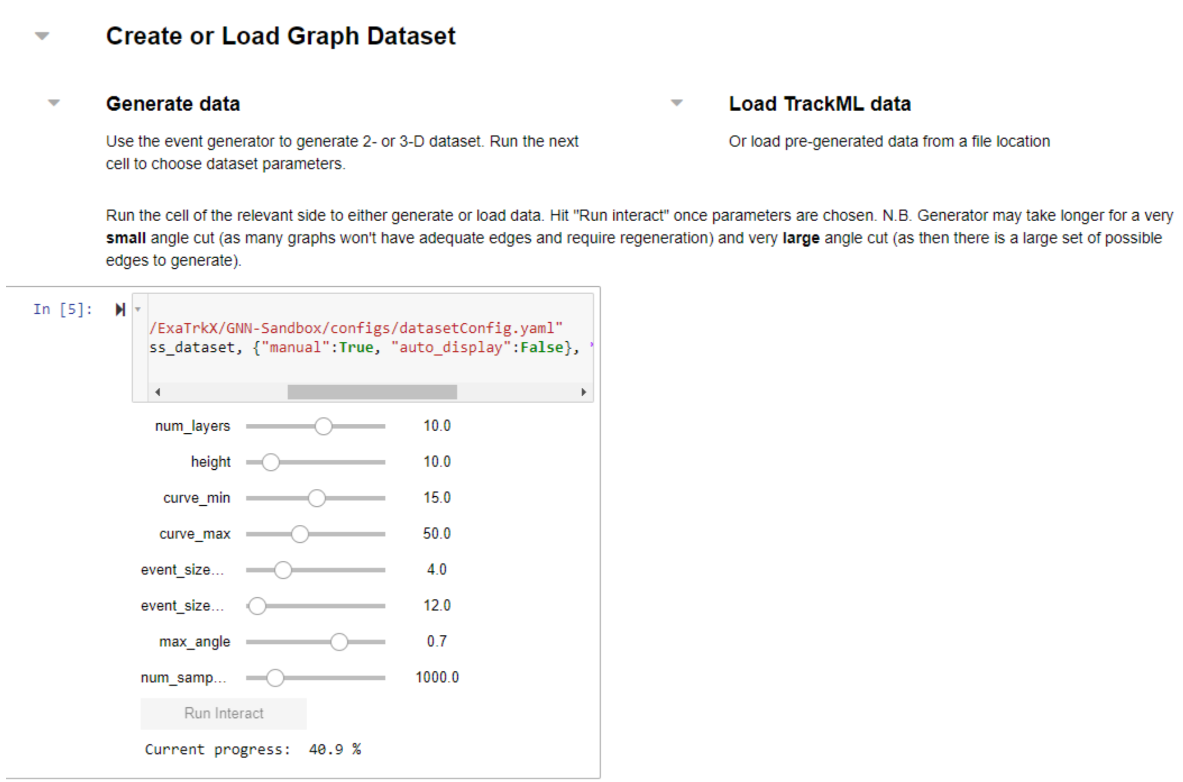
- In 2019, approx. 1 GNN paper every two days
- Presumably will increase in 2020
- Google Trend of "Graph neural network" shows year-by-year increase





Sandbox
DVC
ArchiOpTrkX
Weights & Biases
1
Generate toy data / Load toy data
2
Generate a full GNN architecture
3
Train and evaluate the architecture
4
Optimise the hyperparameters
5
Compare between other architectures
6
Keep track of the hyperparameters, model architecture, weights, metrics, performance, input/output data, … everything

# 1. GENERATE / LOAD DATA



# 1. GENERATE / LOAD DATA

## Create or Load Graph Dataset

### Generate data

Use the event generator to generate 2- or 3-D dataset. Run the next cell to choose dataset parameters.

### Load TrackML data

Or load pre-generated data from a file location

Run the cell of the relevant side to either generate or load data. Hit "Run interact" once parameters are chosen. N.B. Generator may take longer for a very **small** angle cut (as many graphs won't have adequate edges and require regeneration) and very **large** angle cut (as then there is a large set of possible edges to generate).

In [5]:
```
/ExaTrkX/GNN-Sandbox/configs/datasetConfig.yaml"
ss_dataset, {"manual":True, "auto_display":False}, '
```

| | |
|---|---|
| num_layers | 10.0 |
| height | 10.0 |
| curve_min | 15.0 |
| curve_max | 50.0 |
| event_size... | 4.0 |
| event_size... | 12.0 |
| max_angle | 0.7 |
| num_samp... | 1000.0 |

Run Interact

Current progress:  40.9 %

Run the next cell to store the generated data to a dataset list. Change the split as required.

In [6]:
```
dataset = generator.result
dataset_split = interactive(split_dataset, dataset=fixed(dataset), **sliders( "split", config_path) )
dataset_split
```

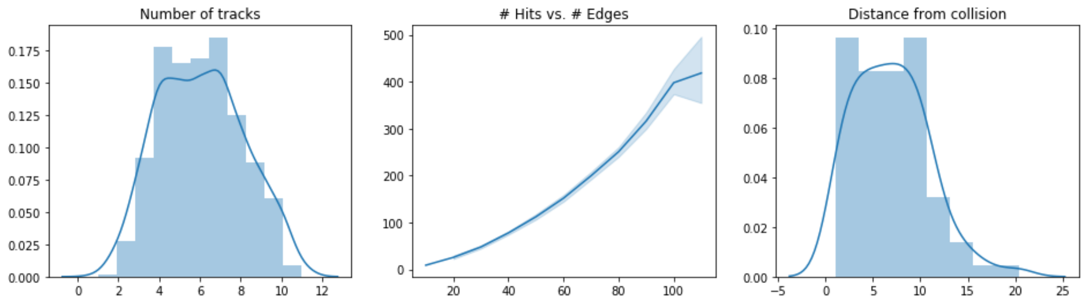
train_percent 60.0

In [7]:
```
train_dataset, test_dataset = dataset_split.result
```

## Visualise Dataset

We can visualise some basic information about the training dataset here.

In [36]:
```
visualise_training_dataset(train_dataset)
```



# 2. AN ATTEMPT TO GENERATE GNN ARCHITECTURE

### GNN Architecture

We can build a new architecture from scratch, or load a YAML architecture, which is then built into a GNN model. Use the builder to:

- Define the overall structure (number of convolutions, number of poolings, type of classification - node, edge or graph)
- Choose the convolutions and poolings from a custom set in the /architecture folder, or from the predefined methods in Pytorch Geometric
- Structure the order of these convolutions and poolings, and the number of recursions
- Choose the MLP structure for each convolution (number of channels, number of layers)
- Choose the MLP structure for input and output layers (number of layers)

The input and output MLP channels are defined automatically by the type of classification.

#### Generate Architecture

Use the dropboxes below to choose the number of {node, edge, graph} {convolutions, poolings}

#### Load Architecture

Or load a pre-written **.sketch** file

In [4]:
```
# Generate architecture↔
```

| | |
|---|---|
| multiples of... | 2 |
| multiples of... | 1 |
| multiples of... | 3 |
| multiples of... | 0 |
| multiples of... | 0 |
| multiples of... | 0 |

Then use the following dropboxes to choose the methods used for each of the above layers

In [5]:
```
[display(i) for i in GNN_layer_generator]
```

| | |
|---|---|
| methods of... | GCN |
| methods of... | GAT |
| methods of... | p1 |
| methods of... | EdgeAttention |
| methods of... | GATEdge |
| methods of... | N-GCN |

# 2. AN ATTEMPT TO GENERATE GNN ARCHITECTURE

### GNN Architecture

We can build a new architecture from scratch, or load a YAML architecture

- Define the overall structure (number of convolutions, number of pooli
- Choose the convolutions and poolings from a custom set in the /arch
- Structure the order of these convolutions and poolings, and the numb
- Choose the MLP structure for each convolution (number of channels,
- Choose the MLP structure for input and output layers (number of laye

The input and output MLP channels are defined automatically by the type

### Generate Architecture

Use the dropboxes below to choose the number of {node, edge, graph} {convolutions, poolings}

In [4]: `# Generate architecture↔`

multiples of... Number of node convolutions

multiples of... Number of node poolings

multiples of... Number of edge convolutions

multiples of... Number of edge poolings

multiples of... Number of graph convolutions

multiples of... Number of graph poolings

archConfig.yaml

```yaml
multiples:
    node:
        convolutions:
            max: 10
            min: 0
            default: 0
        poolings:
            max: 10
            min: 0
            default: 0
    edge:
        convolutions:
            max: 10
            min: 0
            default: 0
        poolings:
            convolutions:
            max: 10
            min: 0
            default: 0
    graph:
        convolutions:
            max: 10
            min: 0
            default: 0
        poolings:
            max: 10
            min: 0
            default: 0

methods:
    node:
        convolutions: ["GCN", "GAT", "N-GCN"]
        poolings: ["p1", "p2", "p3"]
    edge:
        convolutions: ["EdgeAttention", "GATEdge", "N-GCN"]
        poolings: ["EdgePool", "DiffPool, p3"]
    graph:
        convolutions: ["Sum", "Mean", "SortMean"]
        poolings: ["p1", "p2", "p3"]
```

Then use the following dropboxes to choose the methods used for each of the above layers

In [5]: `[display(i) for i in GNN_layer_generator]`

| | |
|---|---|
| Method for node convolution #1 | GCN |
| Method for node convolution #2 | GAT |
| Method for node pooling #1 | p1 |
| Method for edge convolution #1 | EdgeAttention |
| Method for edge convolution #2 | GATEdge |
| Method for edge convolution #3 | N-GCN |

These are defined in a dictionary of methods

# 2. AN ATTEMPT TO GENERATE GNN ARCHITECTURE

**Save/Load Architecture**

```
In [8]:  with open("sketches/steve_model.yaml","r") as f:
             modelConfig = yaml.safe_load(f)

Out[8]: {'pipeline': ['input',
          'shortcut',
          {'loop1': ['edgenet', 'nodenet', 'shortcut']},
          'edgenet'],
         'architecture': {'input': {'type': 'node', 'mlp': True},
          'edgenet': {'type': 'edge',
           'convolution': 'end_concatenation',
           'pooling': 'agg',
           'mlp': True},
          'nodenet': {'type': 'node',
           'convolution': {'attention': 'edgenet'},
           'mlp': True},
          'shortcut': {'type': 'node',
           'convolution': 'shortcut_concatenation',
           'mlp': False}},
         'loops': {'loop1': 4}}
```

```
In [9]:  pipeline = modelConfig['pipeline']
         architecture = modelConfig['architecture']
         loops = modelConfig['loops']
```

```
In [37]: Network(pipeline, architecture, loops)
```

The GNN is generated from the dropdown boxes, or loaded from a “sketch” file.

e.g. *steve_model.yaml*

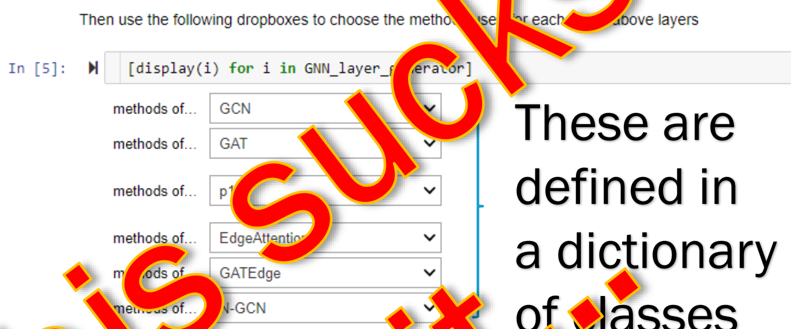
1. States a pipeline with user-defined labels
2. Associates those labels with the dictionary of function
3. Handles any exceptional behaviour (e.g. loops)

```yaml
pipeline:
    - input
    - shortcut
    - loop1:
        - edgenet
        - nodenet
        - shortcut
    - edgenet


architecture:
    input:
        type: node
        mlp: True
    edgenet:
        type: edge
        convolution: end_concatenation
        pooling: agg
        mlp: True
    nodenet:
        type: node
        convolution:
            attention: edgenet
        mlp: True
    shortcut:
        type: node
        convolution: shortcut_concatenation
        mlp: False

loops:
    loop1: 4
```

# 2. AN ATTEMPT TO GENERATE GNN ARCHITECTURE




These are defined in a dictionary of classes

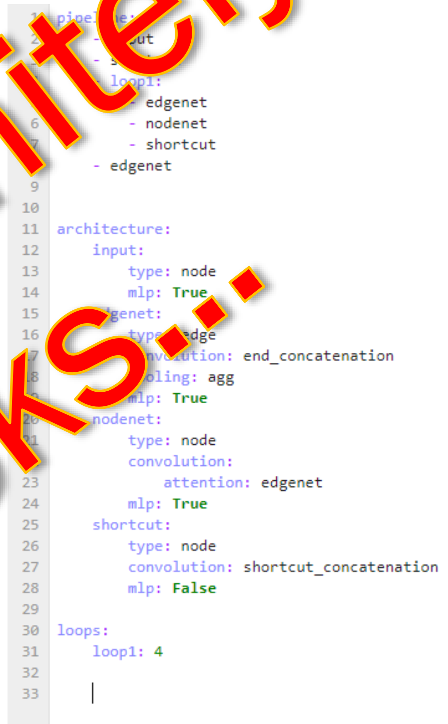
This sucks a bit...

The GNN is generated from the dropdown boxes, or loaded from a “sketch” file.

e.g. *steve_model.yaml*

1. States a pipeline with user-defined labels
2. Associates those labels with the dictionary of functions
3. Handles any exceptional behaviour (e.g. loops)

This definitely sucks...



# 3. TRAIN & EVALUATE THE MODEL

Sketch.yaml

```yaml
pipeline:
    - input
    - shortcut
    - loop1:
        - edgenet
        - nodenet
        - shortcut
    - edgenet


architecture:
    input:
        type: node
        mlp: True
    edgenet:
        type: edge
        convolution: end_concatenation
        pooling: agg
        mlp: True
    nodenet:
        type: node
        convolution:
            attention: edgenet
        mlp: True
    shortcut:
        type: node
        convolution: shortcut_concatenation
        mlp: False

loops:
    loop1: 4

```

```python
def end_concatenation(**kwargs):
    start, end = kwargs["edge_index"]
    x1, x2 = kwargs["x"][start], kwargs["x"][end]
    concat_edge = torch.cat([x1, x2], dim=1)
    return concat_edge

def shortcut(**kwargs):
    inputs = kwargs["inputs"]
    x = kwargs["x"]
    x = torch.cat([x, inputs.x], dim=-1)
    new_kwarg = {"x": x}
    return kwargs.update(new_kwarg)

```

Dictionary of methods

+

PyTorch Network

```python
class EdgeNetwork(nn.Module):
    """
    A module which computes weights for edges of the graph.
    For each edge, it selects the associated nodes' features
    and applies some fully-connected network layers with a final
    sigmoid activation.
    """
    def __init__(self, input_dim, hidden_dim=8, hidden_activation=nn.Tanh,
                 layer_norm=True):
        super(EdgeNetwork, self).__init__()
        self.network = make_mlp(input_dim*2,
                                [hidden_dim, hidden_dim, hidden_dim, 1],
                                hidden_activation=hidden_activation,
                                output_activation=None,
                                layer_norm=layer_norm)

    def forward(self, x, edge_index):
        # Select the features of the associated nodes
        start, end = edge_index
        x1, x2 = x[start], x[end]
        edge_inputs = torch.cat([x[start], x[end]], dim=1)
        return self.network(edge_inputs).squeeze(-1)
```

# 3. TRAIN & EVALUATE THE MODEL

## PyTorch Network

```python
class EdgeNetwork(nn.Module):
    """
    A module which computes weights for edges of the graph.
    For each edge, it selects the associated nodes' features
    and applies some fully-connected network layers with a final
    sigmoid activation.
    """
    def __init__(self, input_dim, hidden_dim=8, hidden_activation=nn.Tanh,
                 layer_norm=True):
        super(EdgeNetwork, self).__init__()
        self.network = make_mlp(input_dim*2,
                                [hidden_dim, hidden_dim, hidden_dim, 1],
                                hidden_activation=hidden_activation,
                                output_activation=None,
                                layer_norm=layer_norm)

    def forward(self, x, edge_index):
        # Select the features of the associated nodes
        start, end = edge_index
        x1, x2 = x[start], x[end]
        edge_inputs = torch.cat([x[start], x[end]], dim=1)
        return self.network(edge_inputs).squeeze(-1)
```

Nothing special here...

## PyTorch Training + Evaluation

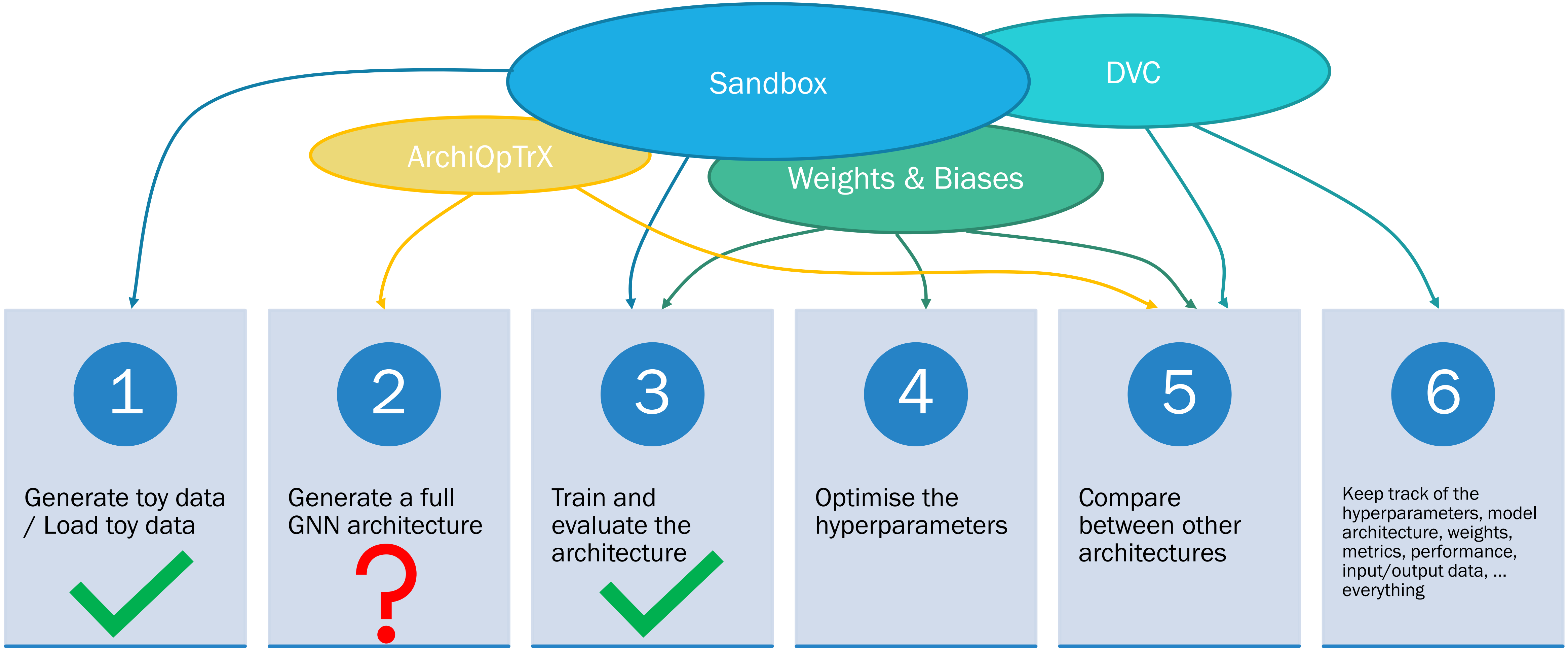
### Edge Classification Testing

In [51]:

```python
model.train()
loss_v = []
acc_v = []
ep = 0
for epoch in range(500):
    ep += 1
    correct = 0
    total = 0
    for batch in train_loader:
#         print(batch.x)
        optimizer.zero_grad()
        data = batch.to(device)
        pred = model(data)
        loss = F.binary_cross_entropy_with_logits(pred.float(), data.y.float())
        loss.backward()
        optimizer.step()
        correct += ((pred > 0.5) == (data.y > 0.5)).sum().item()
#         print(correct, pred, data.y)
        total += len(pred)
#         print(out, data.y, )
    acc = correct/total
    print("Epoch: " , ep, ", loss: ", loss.item(), ", accuracy: ", acc)
    loss_v.append(loss)
    acc_v.append(acc)
plt.plot(np.arange(len(loss_v)), loss_v)
plt.plot(np.arange(len(acc_v)), acc_v)
plt.ylim(0.1,1)
```

In [52]:

```python
model.eval()
for batch in test_loader:
    print(batch)
    data = batch.to(device)
    pred = model(data)
    correct = ((pred > 0.5) == (data.y > 0.5)).sum().item()
    acc = correct / len(pred)
    print('Accuracy: {:.4f}'.format(acc))
```


Sandbox
DVC
ArchiOpTrX
Weights & Biases
1
Generate toy data / Load toy data
2
Generate a full GNN architecture
?
3
Train and evaluate the architecture
4
Optimise the hyperparameters
5
Compare between other architectures
6
Keep track of the hyperparameters, model architecture, weights, metrics, performance, input/output data, ... everything

# 4. OPTIMISE HYPERPARAMETERS WITH WEIGHTS & BIASES (W&B)

**Edge Classification Testing**

In [51]:
```
model.train()
loss_v = []
acc_v = []
ep = 0
for epoch in range(500):
    ep += 1
    correct = 0
    total = 0
    for batch in train_loader:
#         print(batch.x)
        optimizer.zero_grad()
        data = batch.to(device)
        pred = model(data)
        loss = F.binary_cross_entropy_with_logits(pred.float(), data.y.float())
```

Preface with...

1.
```
print("Initialising W&B...")
wandb.init()
```
Initialise W&B

2.
```
model = Edge_Track_Truth_Net(**m_configs).to(device)
wandb.watch(model, log='all')
```
Watch model (weights & gradients)

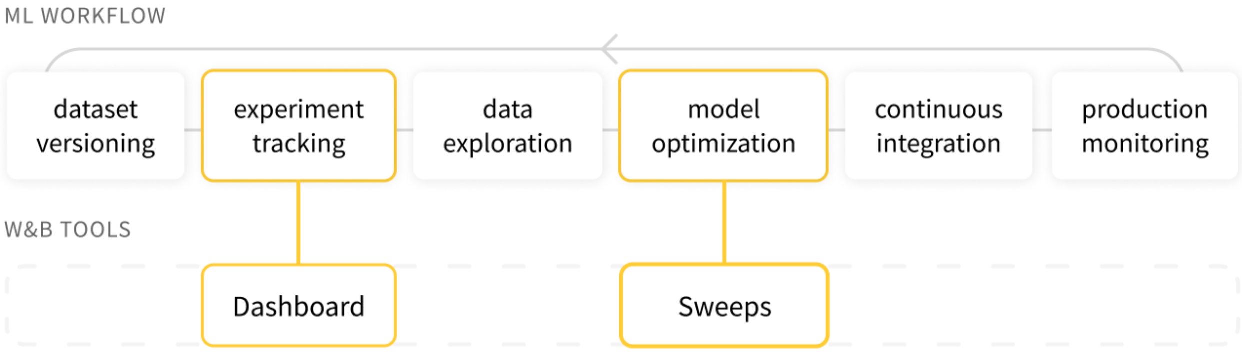
3.
```
wandb.log({"Validation Accuracy": val_acc, "Best Accuracy": best_acc, "Validation Loss": val_loss, "Learning Rate":
```
Log metrics of interest

4.
```
wandb.agent(sweep_id, function=train)
```
Define sweep agent

5.
```
sweep = wandb.controller(sweep_id)
sweep.run()
```
Run sweep

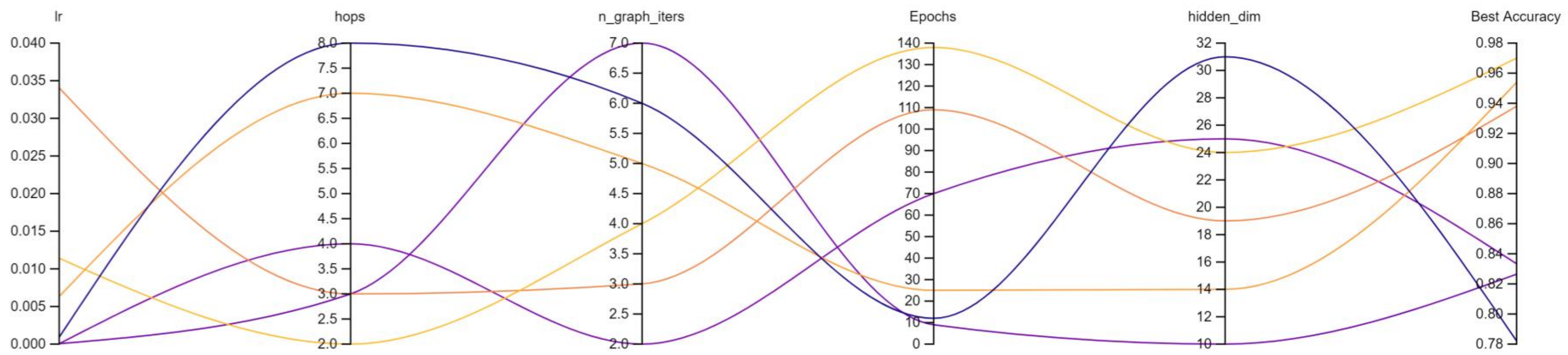
# WEIGHTS & BIASES PROPAGANDA SLIDE

- Wandb.ai
- *"Those who don't track training are doomed to repeat it."*
- Hard workers – have made 5 or 6 suggestions to them, and most were quickly implemented



# 4. OPTIMISE HYPERPARAMETERS WITH WEIGHTS & BIASES (W&B)

- Sweeps compare HPO results



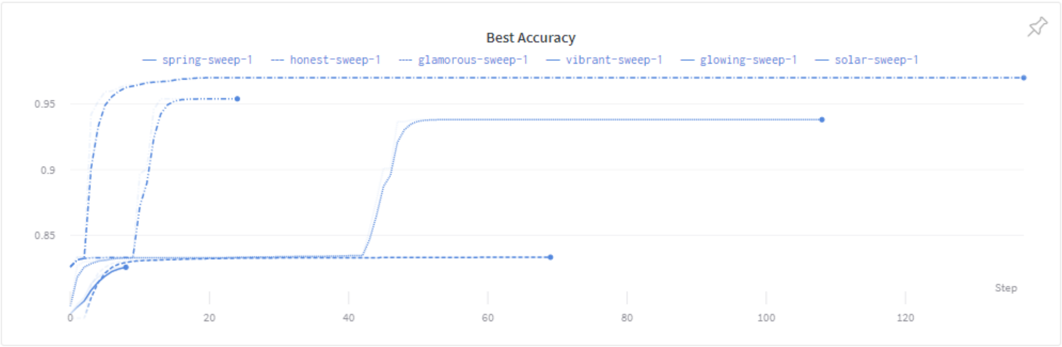

An HPO sweep for doublet classification

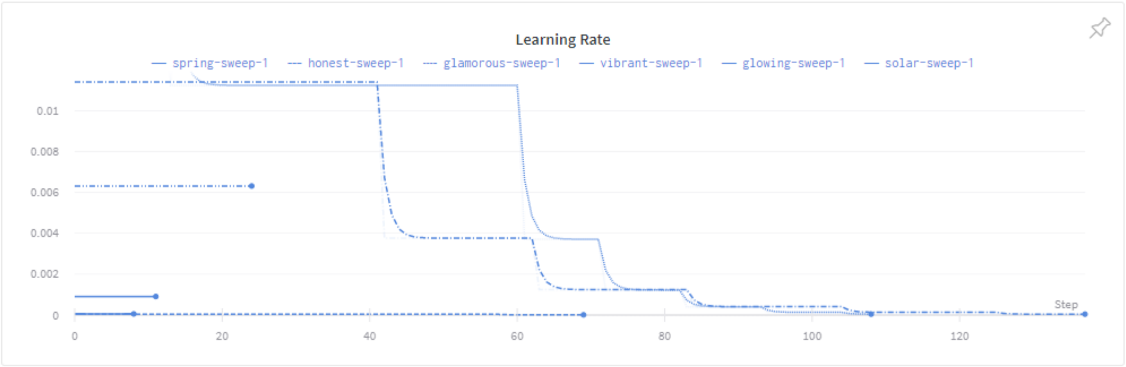
- W&B is an amazing metric/weights/gradients tracker and visualiser, and a standard HPO platform (implements Ray's *Tune* library)

# 5. COMPARE ARCHITECTURES

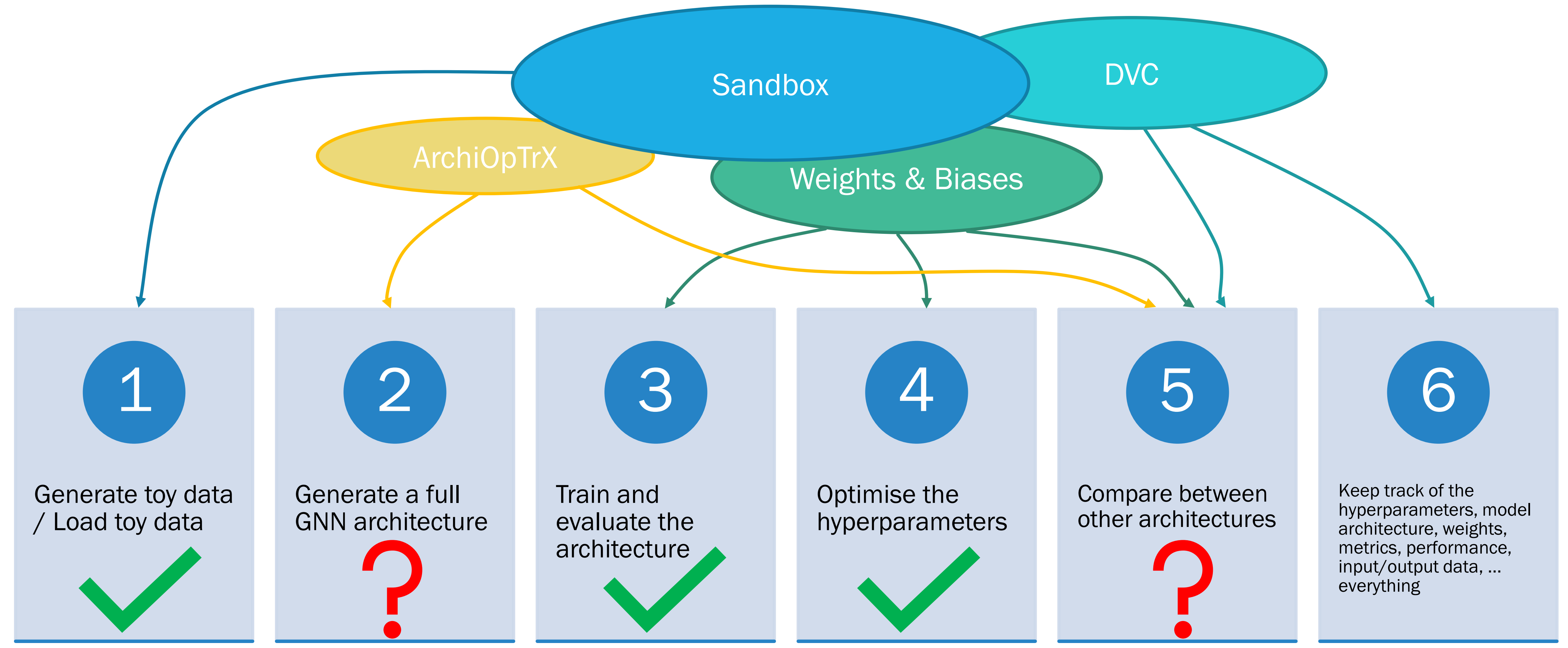
- W&B can be used to compare between sweeps

CHARTS (5)



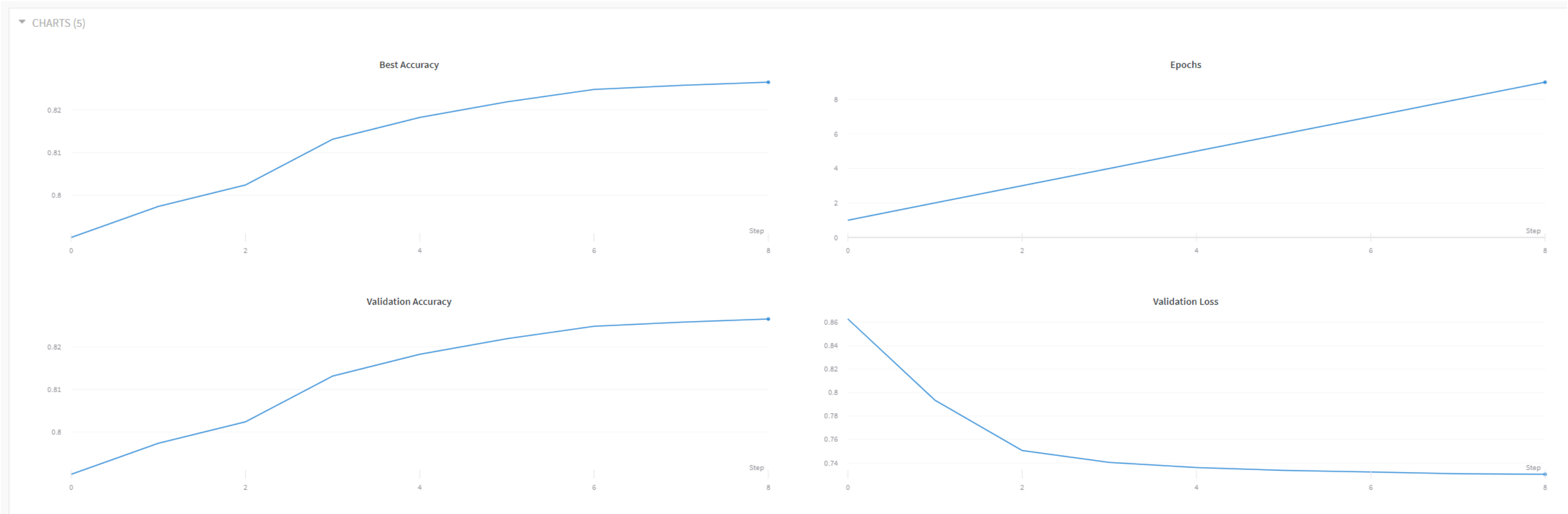




- W&B *can* do discrete Bayesian optimisation over architectures (but this doesn't really work well in practice)


Sandbox
DVC
ArchiOpTrX
Weights & Biases
1
Generate toy data / Load toy data
2
Generate a full GNN architecture
3
Train and evaluate the architecture
4
Optimise the hyperparameters
5
Compare between other architectures
6
Keep track of the hyperparameters, model architecture, weights, metrics, performance, input/output data, … everything

# 6. TRACK *EVERYTHING*

- W&B tracks *every* element of a model:
  - Metrics



# 6. TRACK *EVERYTHING*

- W&B tracks *every* element of a model:
  - Metrics
  - Gradients + weights

# 6. TRACK *EVERYTHING*

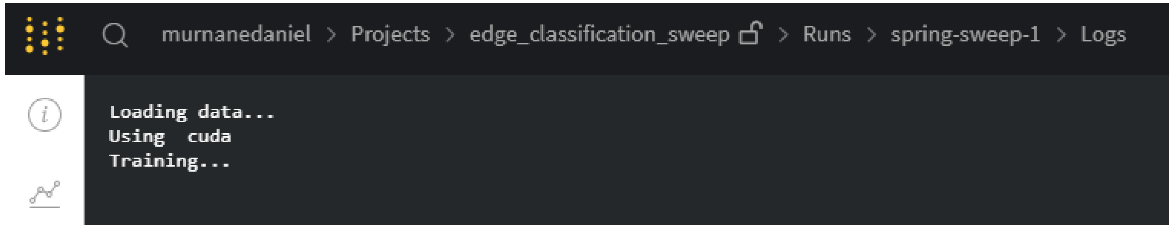
- W&B tracks *every* element of a model:
  - Metrics
  - Gradients + weights
  - Model architecture

graph_0

| | Name | Type | # Parameters | Output Shape |
|---|---|---|---|---|
| • | input_network.0 | Linear(in_features=3, out_features=10, bias=True) | 30, 10 | 12420,10 |
| • | input_network.1 | ReLU() | | 12420,10 |
| • | edge_network | EdgeNetwork( (network): Sequential( (0): Linear(in_features=26, out_features=10, bias=True) (1): ReLU() (2): Linear(in_fea… | 260, 10, 100, 10, 100, 10, 10, 1 | 42789 |
| • | node_network | NHopAttNetwork( (network): Sequential( (0): Linear(in_features=91, out_features=10, bias=True) (1): LayerNorm((10,), ep… | 910, 10, 10, 10, 100, 10, 10, 10, 100, 10, 10, 10, 100, 10, 10, 10 | 12420,10 |

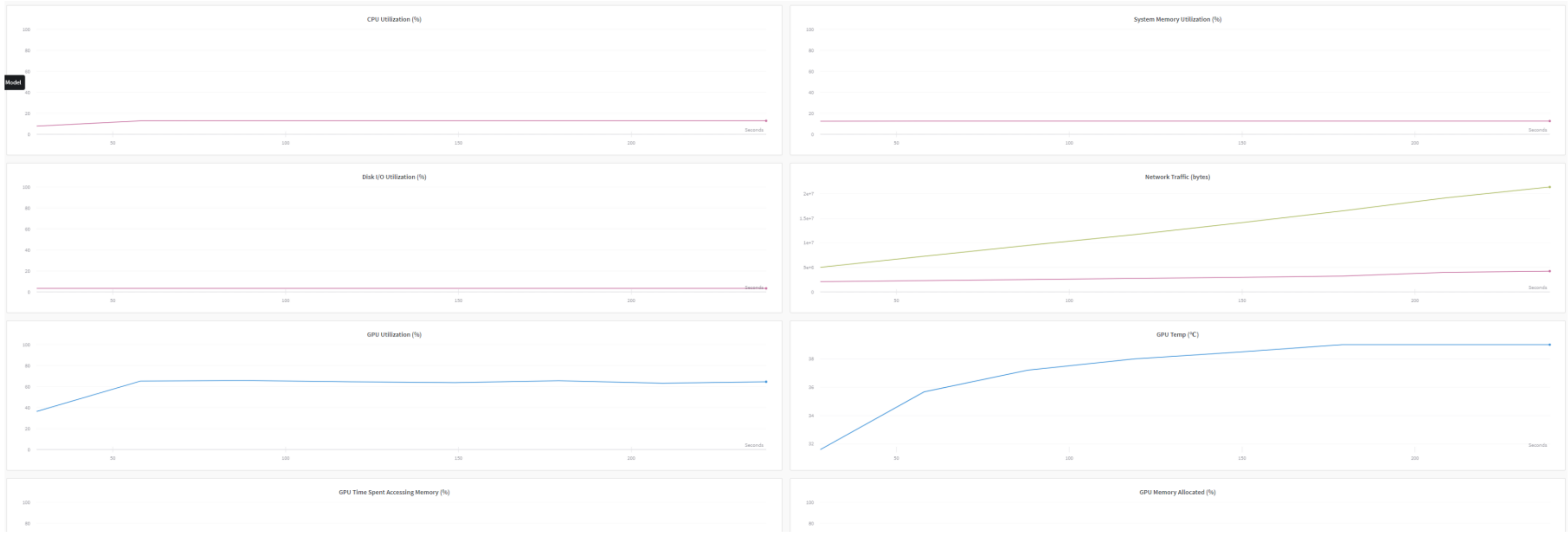
# 6. TRACK *EVERYTHING*

- W&B tracks *every* element of a model:
  - Metrics
  - Gradients + weights
  - Model architecture
  - Terminal logs



# 6. TRACK *EVERYTHING*

- W&B tracks *every* element of a model:
  - Metrics
  - Gradients + weights
  - Model architecture
  - Terminal logs
  - Performance

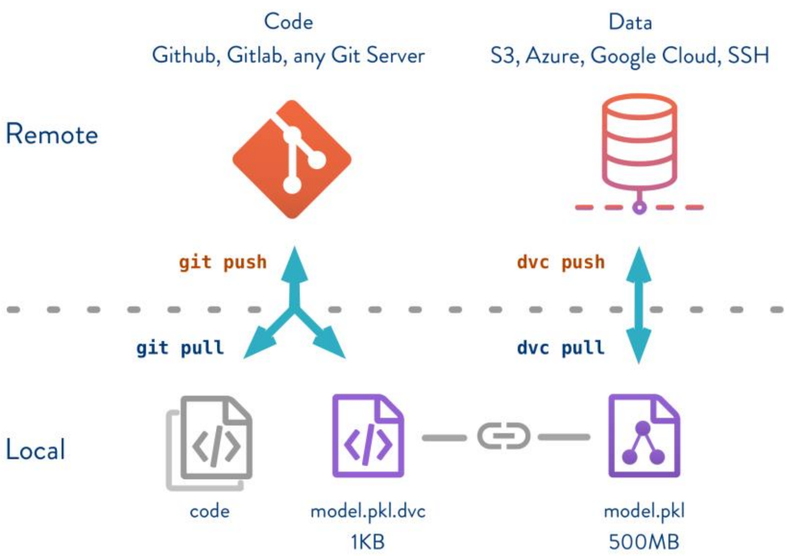


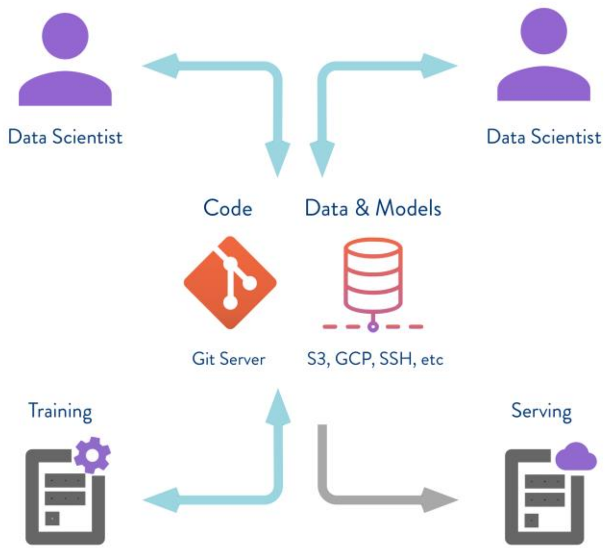
# 6. TRACK *EVERYTHING*

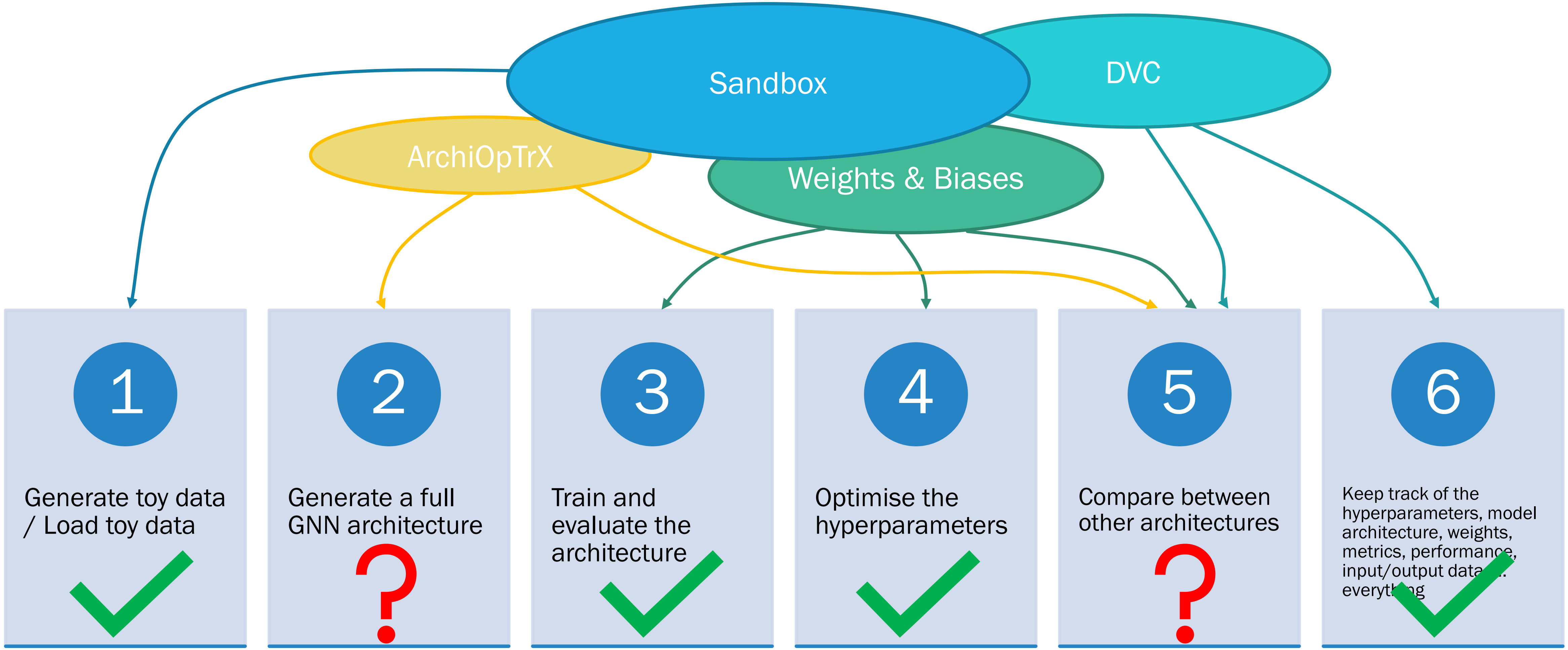
- W&B tracks *every* element of a model:
  - Metrics
  - Gradients + weights
  - Model architecture
  - Terminal logs
  - Performance
- Not reproducible, per se
- Need *Data* (and model) *Version Control* (DVC)

# DVC PROPAGANDA SLIDE

- DVC.org
- A full model & data versioning solution
- Excellent tutorials
- Also work very hard
- Pretty reliable Google Drive API



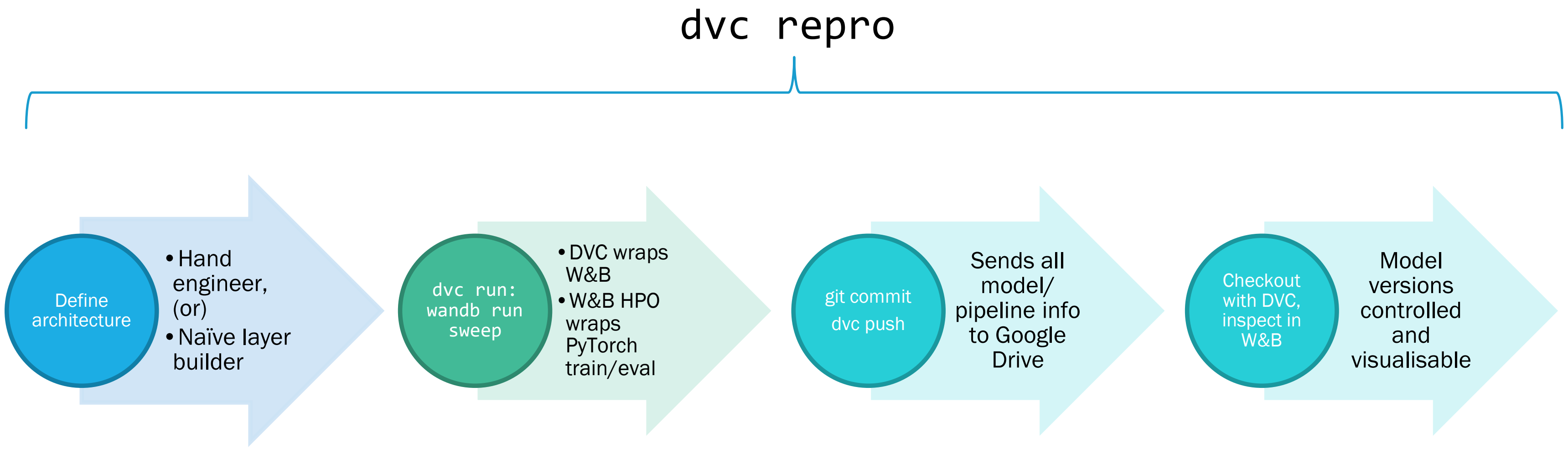





Sandbox
DVC
ArchiOpTrX
Weights & Biases
1
Generate toy data / Load toy data
2
Generate a full GNN architecture
3
Train and evaluate the architecture
4
Optimise the hyperparameters
5
Compare between other architectures
6
Keep track of the hyperparameters, model architecture, weights, metrics, performance, input/output data

# 6. TRACK *EVERYTHING*

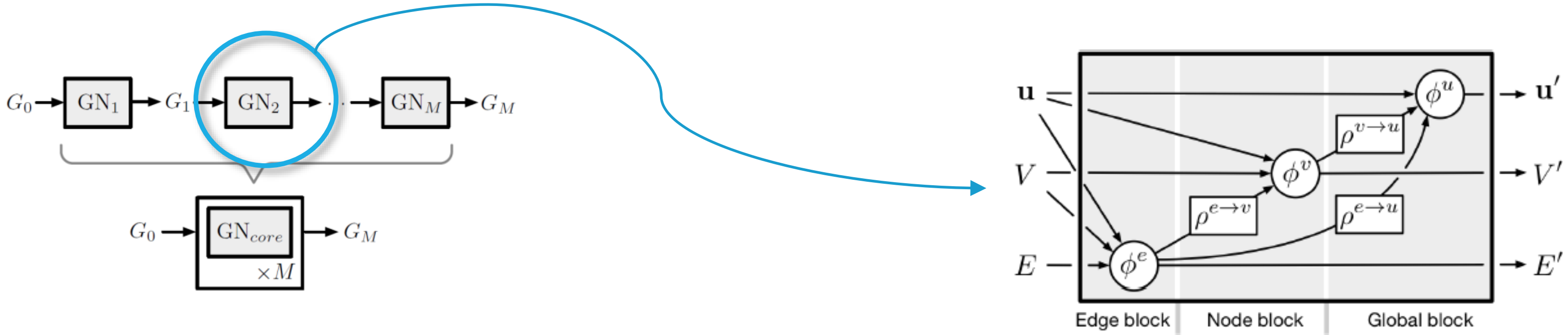
Pipeline is thus:



## 2. GENERATE GNN ARCHITECTURE

# REDUX: GRAPHS ALL THE WAY DOWN

- Linear pipeline is not a good representation for a GNN
- C.f. *Relational Inductive Biases, Deep Learning, and Graph Networks* [Battaglia et al., 2018, Google DeepMind + Brain + U of Edinburgh]
- Google paper uses "blocks", where each block contains a flow of one node, edge and graph function

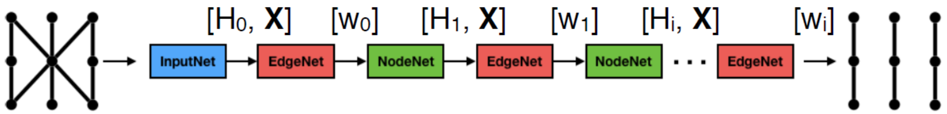


## 2. GENERATE GNN ARCHITECTURE

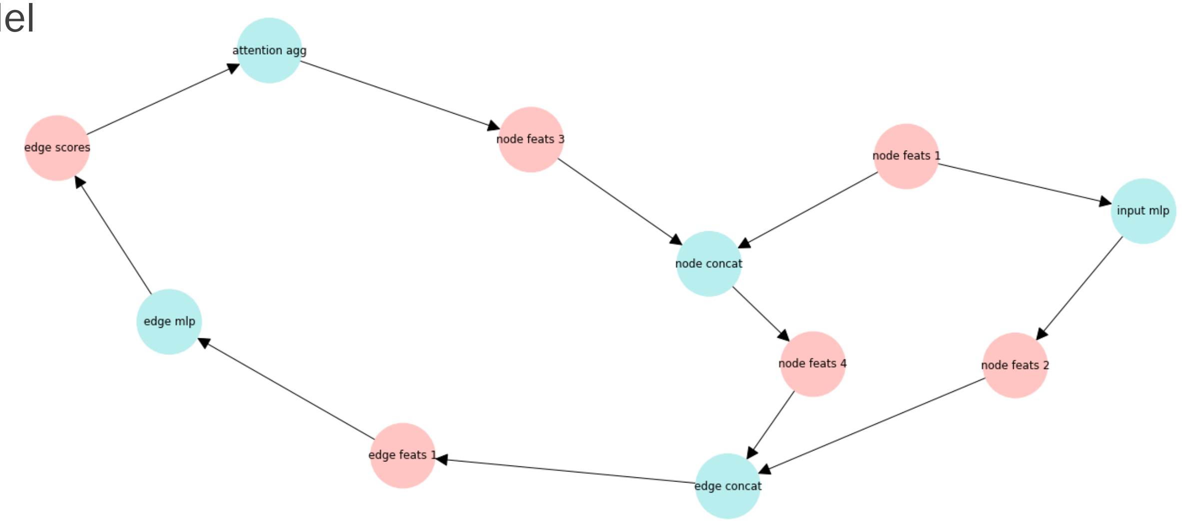
# REDUX: GRAPHS ALL THE WAY DOWN

Graph representation

- ML Network as computations graph, e.g. GNN Attention Model




Linear representation




- *Neural Architecture Search Over a Graph Search Space* [Jastrzebski et al., 2019, Google AI]
- Suggests graph representation can be optimised better than linear representation

Data node

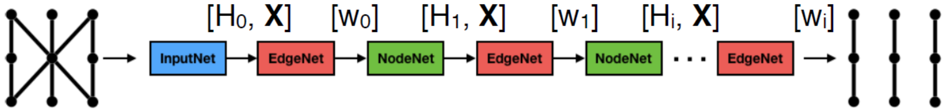
Function node

## 2. GENERATE GNN ARCHITECTURE

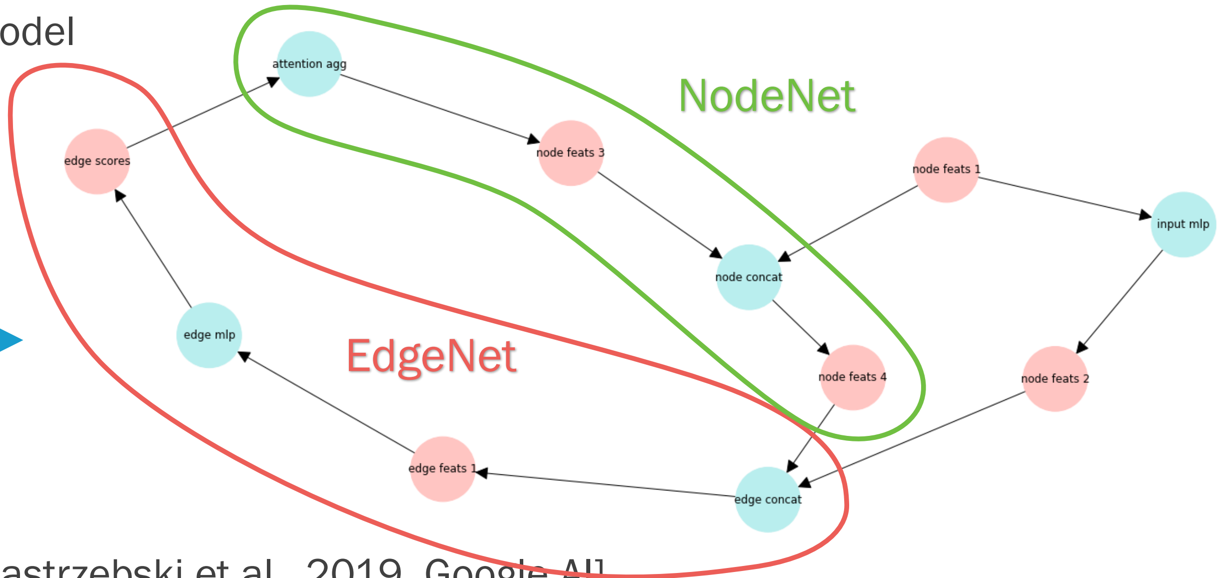
# REDUX: GRAPHS ALL THE WAY DOWN

Graph representation

- ML Network as computations graph, e.g. GNN Attention Model




Linear representation




Data node

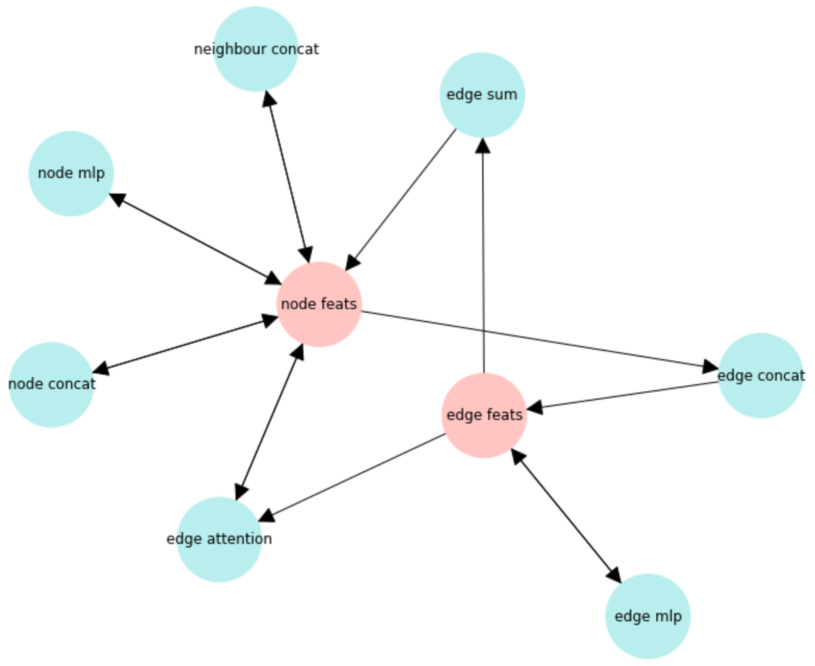
Function node

- *Neural Architecture Search Over a Graph Search Space* [Jastrzebski et al., 2019, Google AI]
- Suggests graph representation can be optimised better than linear representation

### 2. GENERATE GNN ARCHITECTURE

# REDUX: GRAPHS ALL THE WAY DOWN

- New plan: generate GNN architecture as a *computation* graph
- Obvious problem: Cannot arbitrarily connect data-function-data
- To choose the structure of the graph, walk through a *function* graph
- As we walk through *function* graph, we are constructing the *computation* graph



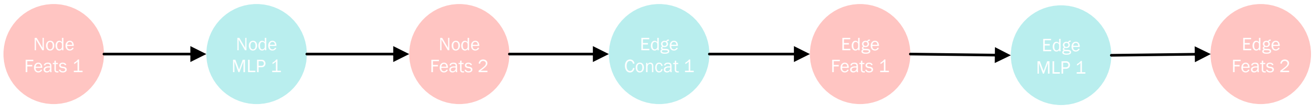

Example function graph

# ARCHITECTUREOPTIMISATION(WITH/FOR)TRACKX: ARCHIOPTRX

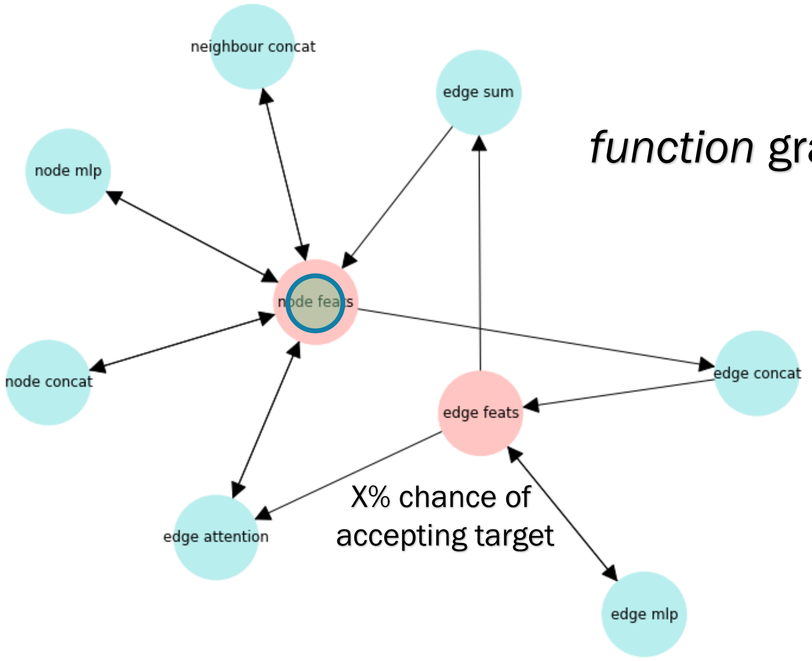
1. Define sources and targets (e.g. source: node data, target: edge data)
2. Start random walk in *function* graph at source node
3. From data nodes, randomly choose a child function
4. From function nodes, proceed to child data

# ARCHITECTUREOPTIMISATION(WITH/FOR)TRACKX: ARCHIOPTRX

1. Define sources and targets
   (e.g. source: node features, target: edge features)
2. Start random walk in *function* graph at source node
3. From data nodes, randomly choose a child function
4. From function nodes, proceed to child data



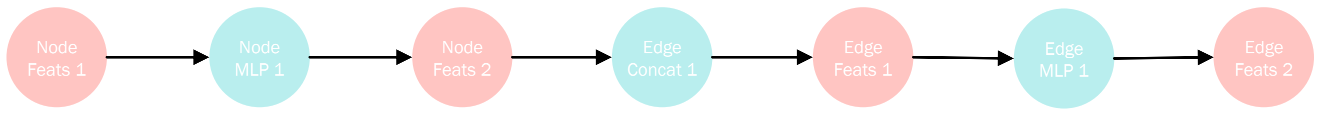

*computation* graph



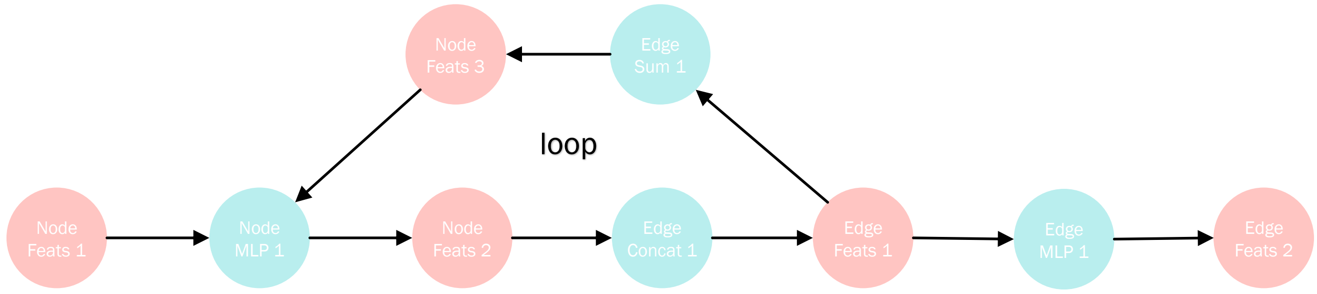

*function* graph

# ARCHITECTUREOPTIMISATION(WITH/FOR)TRACKX: ARCHIOPTRX

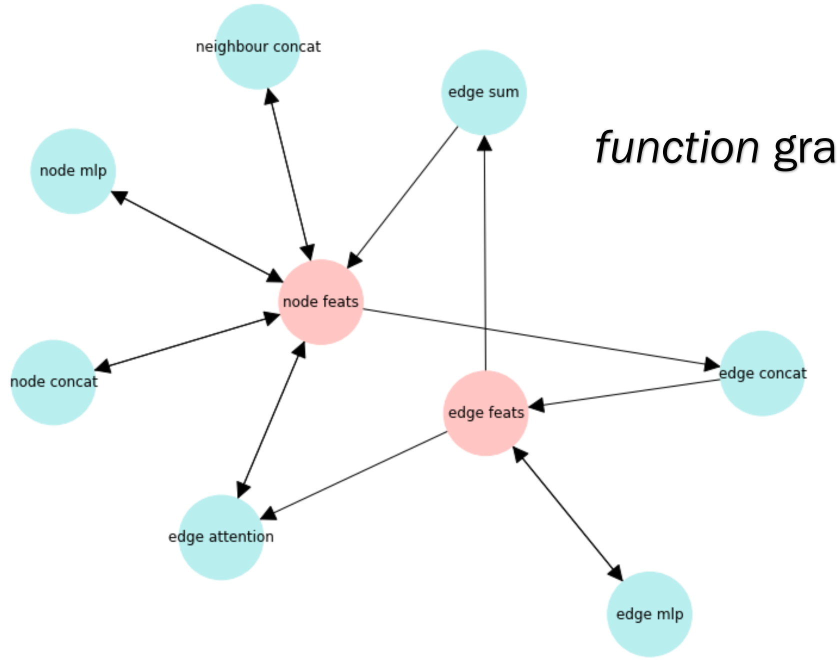
1. First pass track (could be good enough)




2. Then, can add subsequent tracks with Y% percent likelihood, randomly choosing a new source/target. E.g. recurrence:

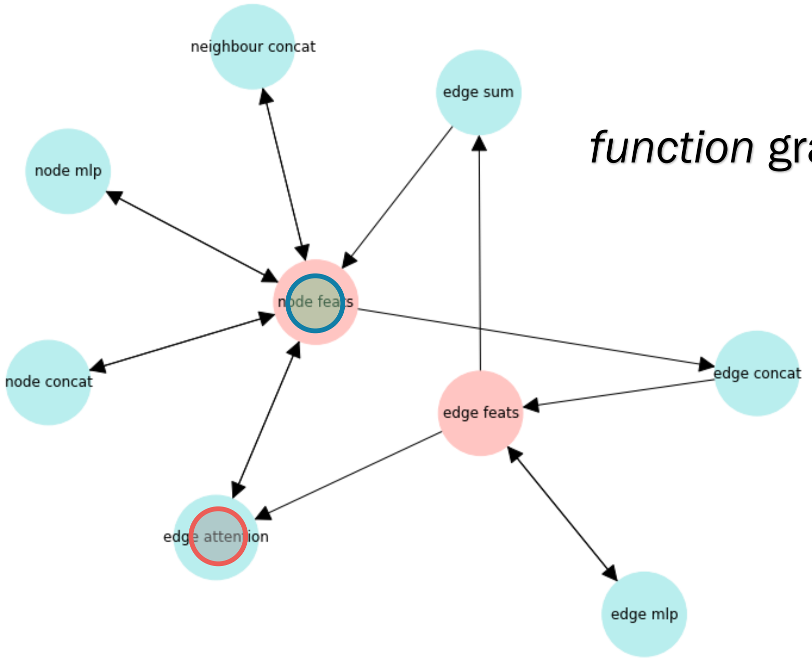



*function graph*

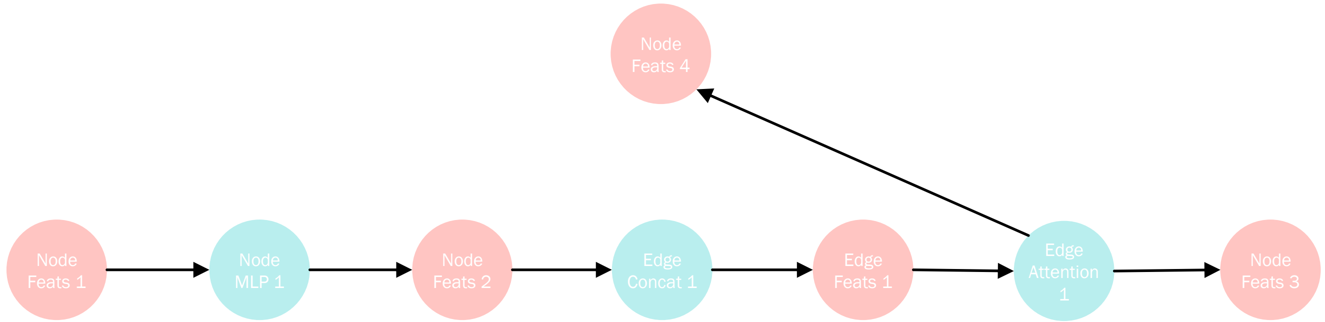


# HANDLING MULTI-INPUT FUNCTIONS

- What happens when we reach a function that has # parents > 1
- We reverse random walk – follows the exact prescription of regular walk but with source ⟷ target



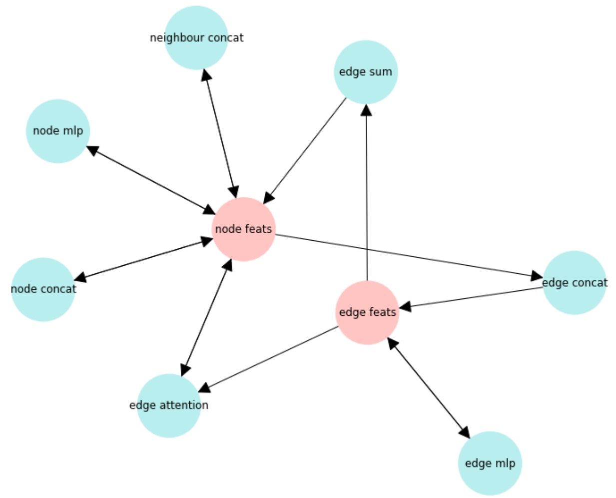

*function* graph

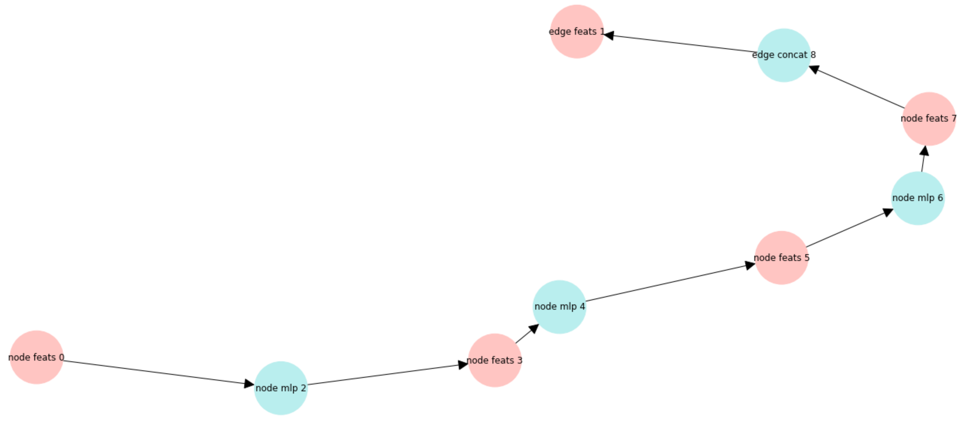



*computation* graph

# ARCHIOPTRX RANDOM GENERATION WORKS

With function graph pictured, we get with source=node, target=edge:

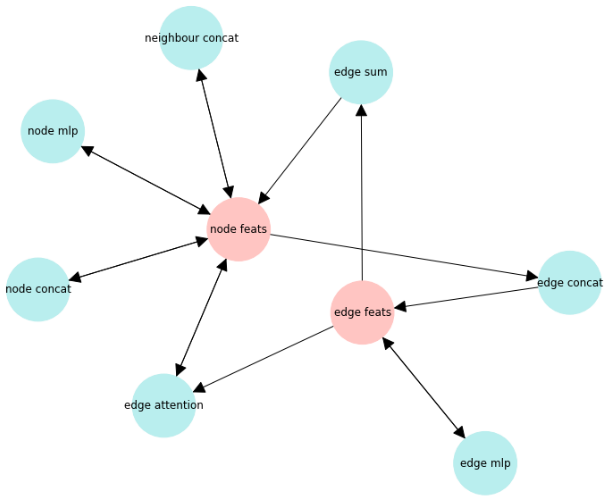


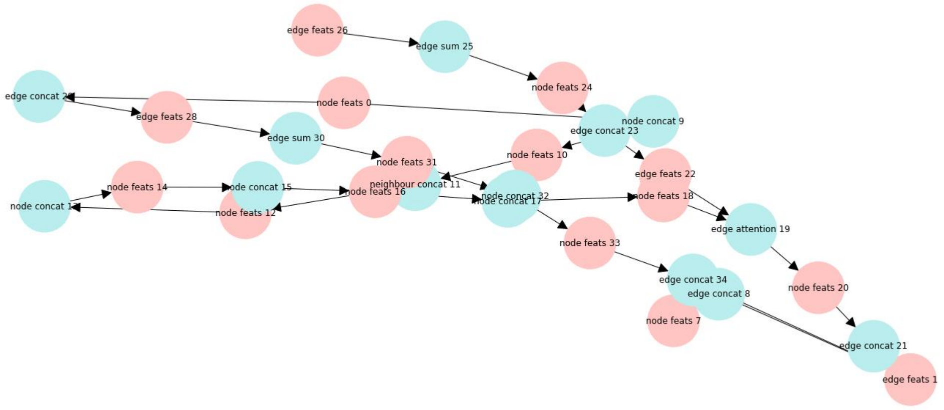




First pass track

# ARCHIOPTRX RANDOM GENERATION WORKS

With function graph pictured, we get with source=node, target=edge:




Two track iterations

# ARCHIOPTRX: PLUGGING INTO PIPELINE

`dvc repro`

- **Architecture generation** — Function graph random walk
- **Generate hyperpar-ameters** — Given the full archit-ecture
- **`dvc run: wandb run sweep`**
  - DVC wraps W&B
  - W&B HPO wraps PyTorch train/eval
- **git commit dvc push** — Sends all model/ pipeline info to Google Drive
- **Checkout with DVC, inspect in W&B** — Model versions controlled and visualisabl e

# NEXT STEP

- RNN Reinforcement Learning controller for function graph walk

# BACKUP

# ASIDE: QUICK NOTATION

- Recall ≡ Efficiency
- Precision ≡ Purity