

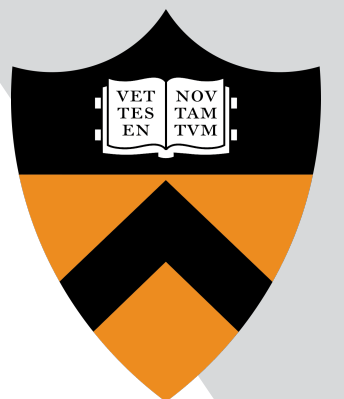
Go small then go home

Hyperparameter transfer for HEP

FastML 2025

Liv Våge, Gage de Zoort

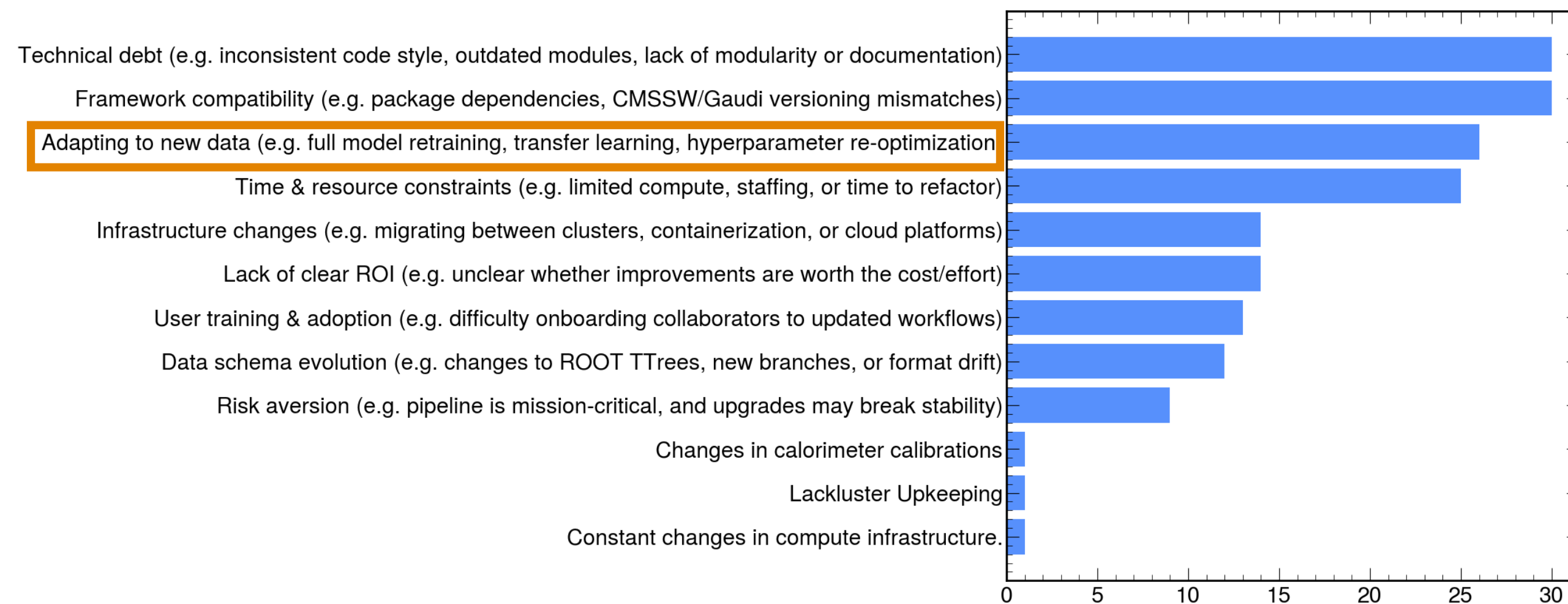
04.09.2025



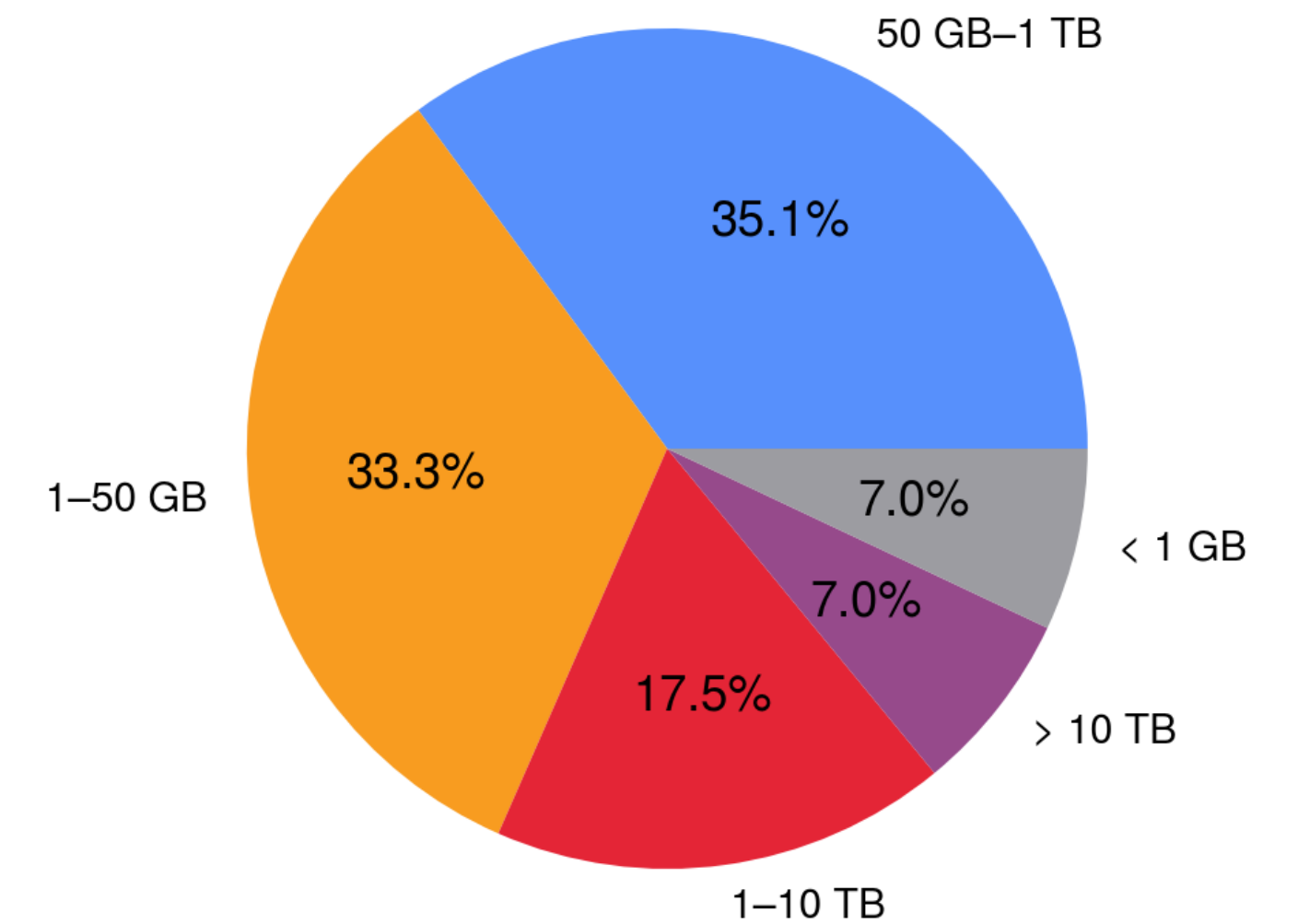
Motivation

Recent survey on ML in HEP*

The largest bottleneck when upgrading an ML pipeline



The usual data size used in ML in HEP



Hyperparameter tuning is slow, especially with large data
It can be a bottleneck for updating a model or introducing new data

* Full results will be presented in IML meeting and a paper soon

Contents

Under certain conditions, the best learning rate for a small neural network is also the best for a *deeper* and *wider* neural network

You can tune hyperparameters on a smaller model

- Theory
- Experiments on 3 real HEP architectures; a DNN, CNN and Transformer
- What to expect and how to implement

The literature

< 2022

Meta learning

Similar networks have similar best hyperparameters

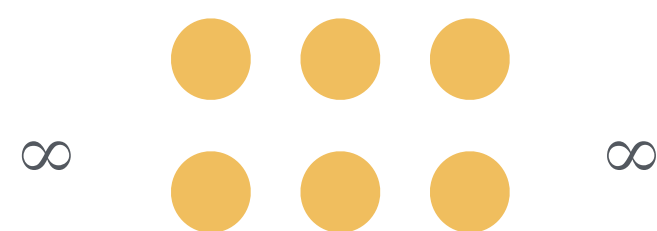


2022
Yang et al

Feature learning in infinite width NN

Usually infinite width networks can't learn features, but with a small change you can

$$W_{ij} \sim \mathcal{N}\left(0, \frac{\sigma^2}{\sqrt{\text{width}}}\right)$$

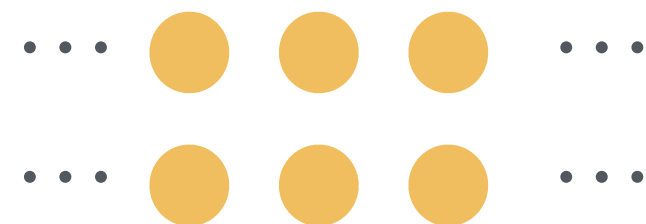


2022
Yang et al

μ Transfer

Scaling by width keeps gradient signals **invariant to width**

$$W_{ij} \sim \mathcal{N}\left(0, \frac{\sigma^2}{d_{\text{in}}}\right)$$

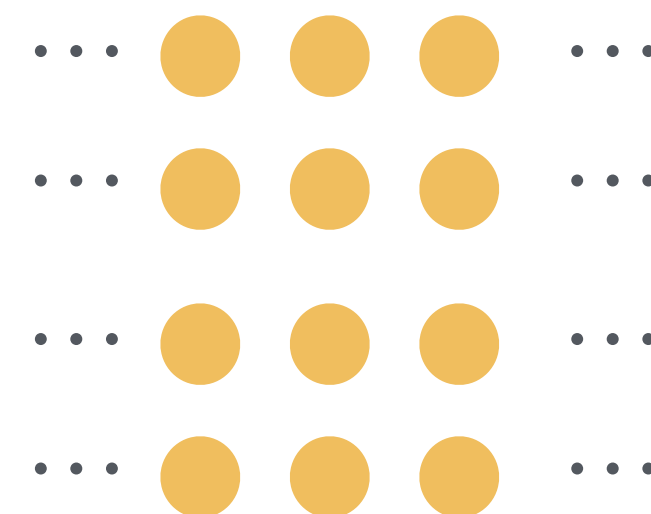


2023
Bordelon et al

Depthwise transfer

Residual layers with a depth scaling by depth lets gradients **scale across depth**

$$x_{l+1} = x_l + \frac{1}{\sqrt{\text{depth}}} f(x_l; W_l)$$



2024
Lingle

Transformers

Demonstrating μ Transfer for transformers

2024
Noci et al

Super consistency

Theoretical developments on loss landscapes

Scaling*

Architecture	Weight Initialization	Depth Scaling	Other
MLP (μP)	In $W^{(1)} \sim \mathcal{N}(0, \frac{\sigma^2}{\text{width}})$ Hidden $W^{(\ell)} \sim \mathcal{N}(0, \sigma^2)$ Out $W^{(L)} \sim \mathcal{N}(0, \sigma^2 \cdot \text{width})$	Insert residual skip scaling $x_{\ell+1} = x_{\ell} + \frac{1}{\sqrt{\text{depth}}} f(x_{\ell}; W_{\ell})$	Works best with tanh activation
CNN	Same as MLP but with conv kernels: scale by fan-in (#input channels \times kernel size)	Same as MLP	Batch normalisation helps keep gradients stable
Transformer	Attention Q, K, V, O matrices: fan-in scaling MLP sublayers: μ P rules	Each residual branch (attn + MLP) scaled by $1/\sqrt{\text{depth}}$	

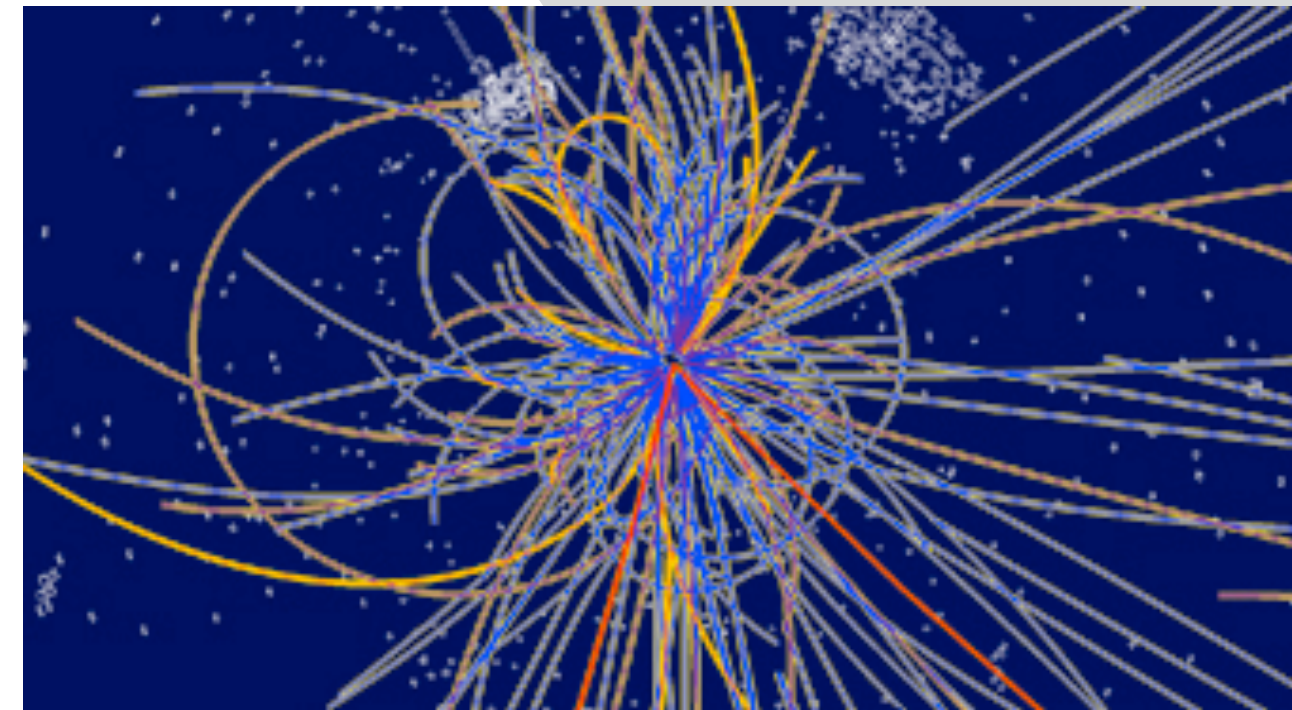
* These vary between implementations

Scaling

	Architecture	Weight Initialization	Depth Scaling	Other
MLP (μ P)	<p>TLDR theory: Keep gradients invariant</p> <p>TLDR implementation: Follow the <u>mup package</u></p>			
CNN				
Transformer		matrices: fan-in scaling MLP sublayers: μ P rules	Each residual branch (attn + MLP) scaled by $1/\sqrt{\text{depth}}$	ps

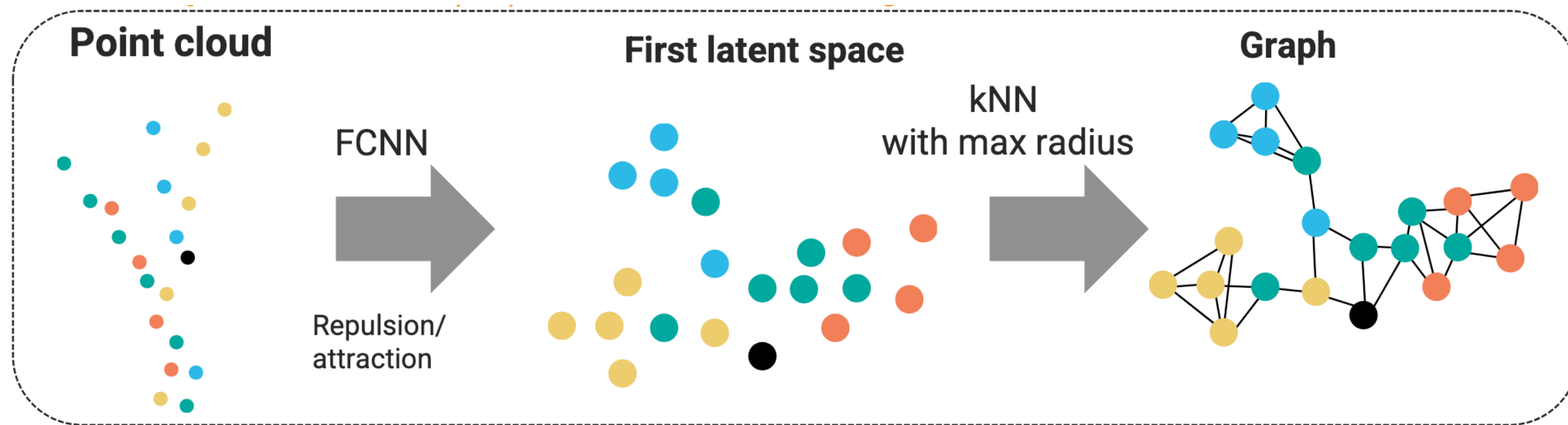
1. MLP

Metric learning for particle tracking



Metric learning for tracking

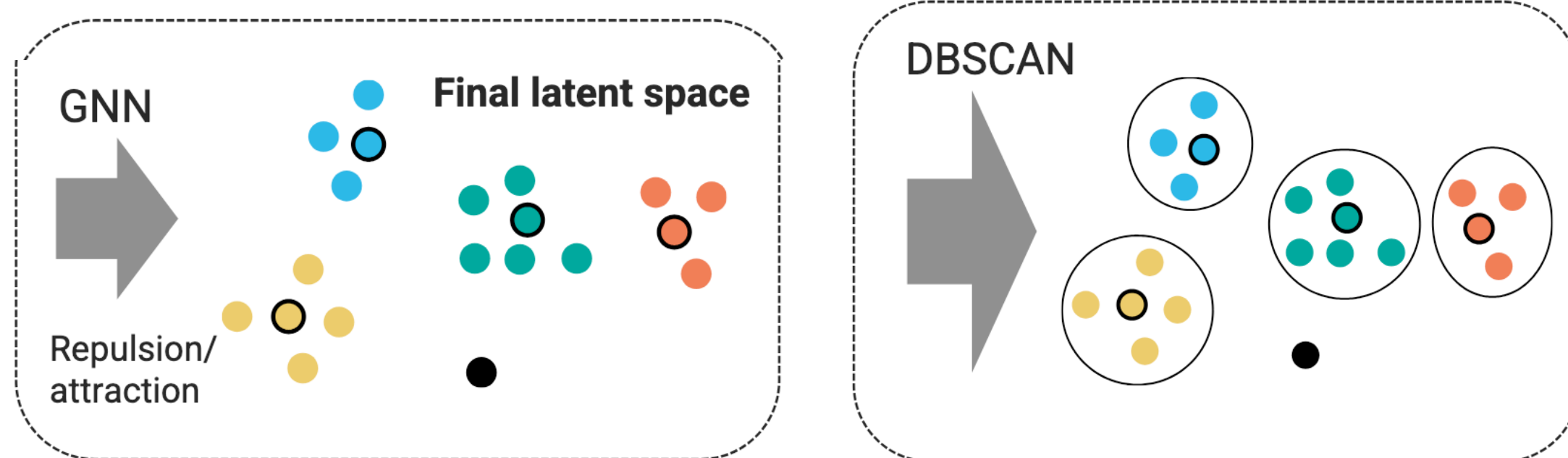
Particle tracking [1]



Hits in tracker as point cloud

Learn a representation where same track hits are close

Build clusters with kNN



GNN learns new latent space

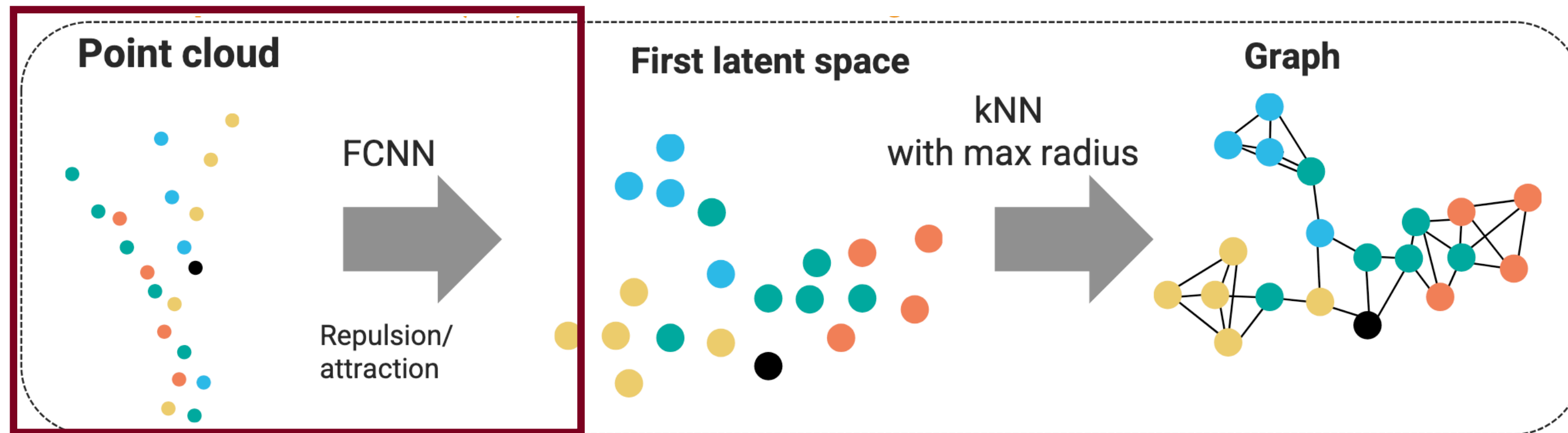
DBSCAN creates final tracks

Figures from [1]

[1] Developed by Kilian Lieret and Gage de Zoort, details [here](#) and code [here](#)

Metric learning for tracking

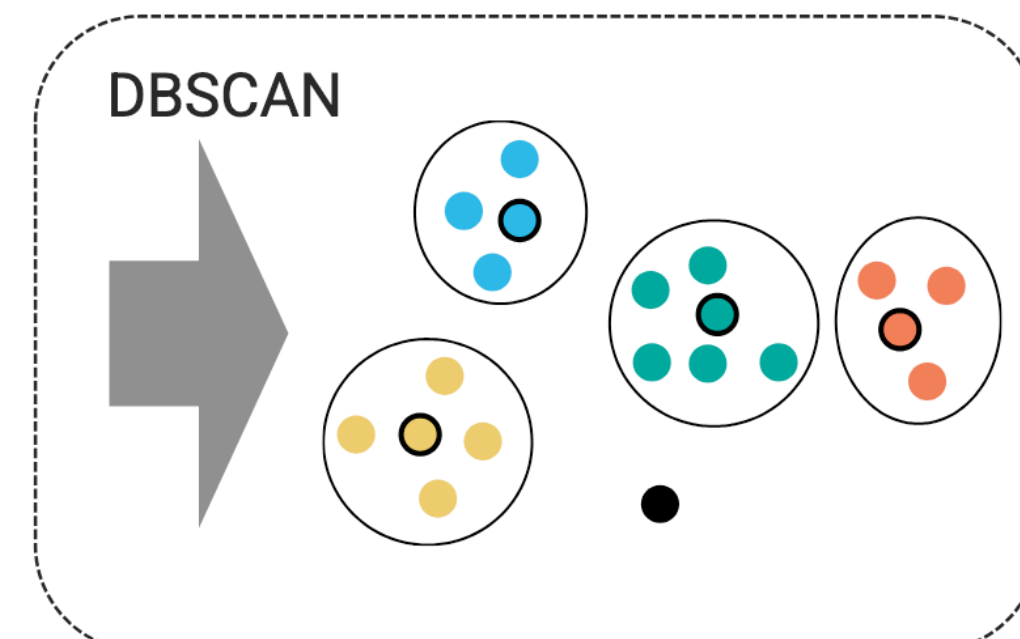
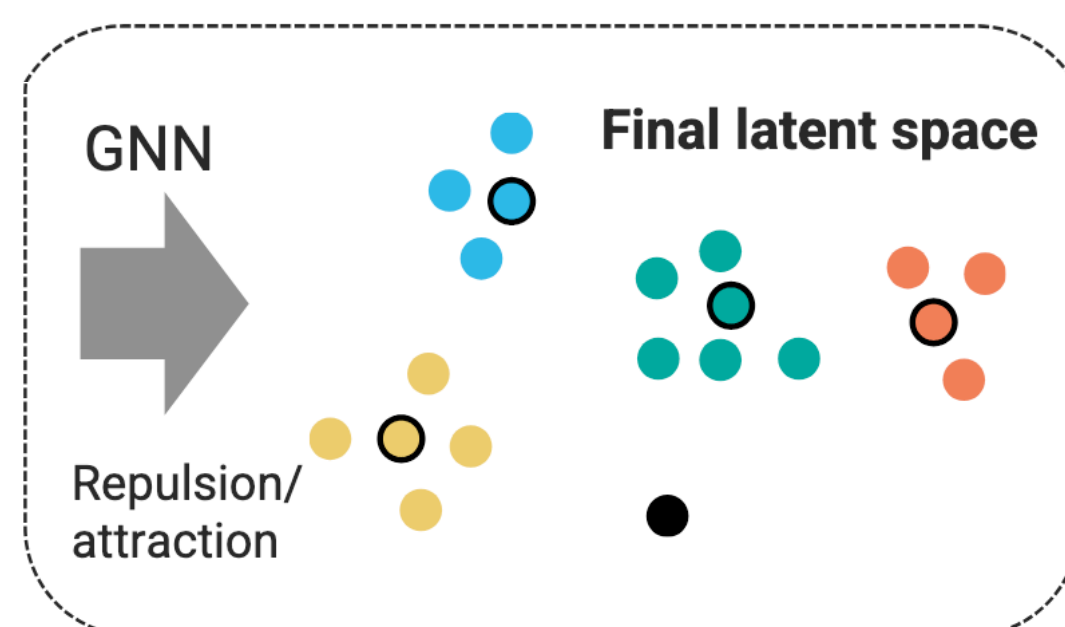
Particle tracking



We will focus on the first step

Uses TrackML data

We will only test on pixel data for speed



Uses object condensation loss

Changing the MLP

```
def reset_parameters(self):
    for _, weights in enumerate(self.layers):
        for p in weights.weight:
            nn.init.normal_(p.data, mean=0, std=1)

def get_lr(self, optimizer):
    if "sgd" in optimizer.lower():
        return self.eta_0 * self.gamma_0**2 * self.width
    elif "adam" in optimizer.lower():
        return self.eta_0 * self.gamma_0 * math.sqrt(self.width)
    else:
        raise Exception(f"Cannot locate parametrization for optimizer {optimizer}")

def forward(self, x: T):
    for layer, weights in enumerate(self.layers):
        if layer == 0:
            x = weights(x) / math.sqrt(self.in_dim)
        elif (layer > 0) and (layer < self.depth):
            x = x + (self.beta / math.sqrt(self.depth * self.width)) * weights(
                self.act(x)
            )
        else:
            x = weights(self.act(x)) / (self.width * self.gamma)
    return x
```

Already had residual connections

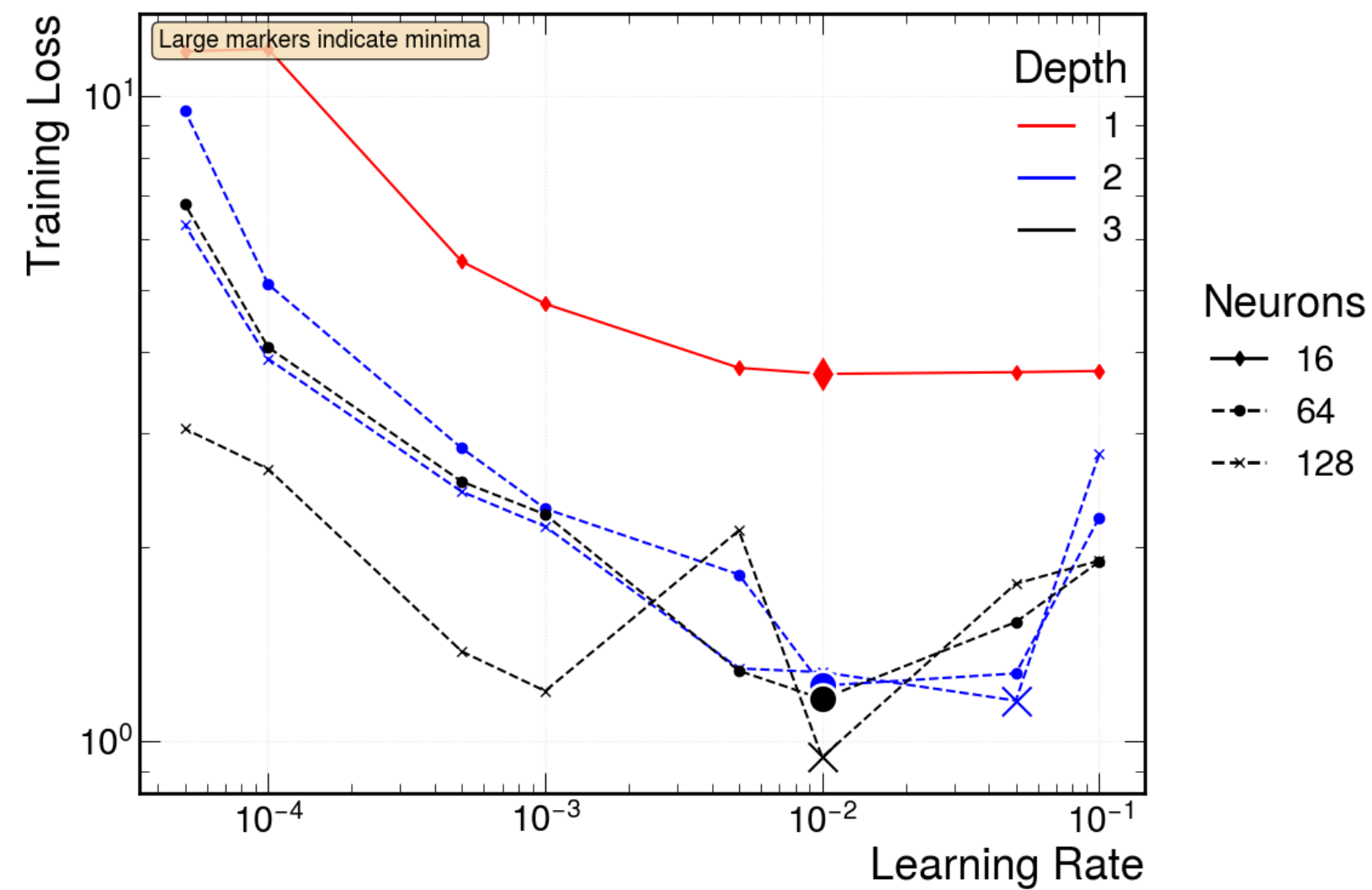
Keep ADAM as optimiser

- ✓ Scale residual by depth and width $\frac{1}{\sqrt{\text{width} * \text{depth}}}$
- ✓ Change weight initialisation

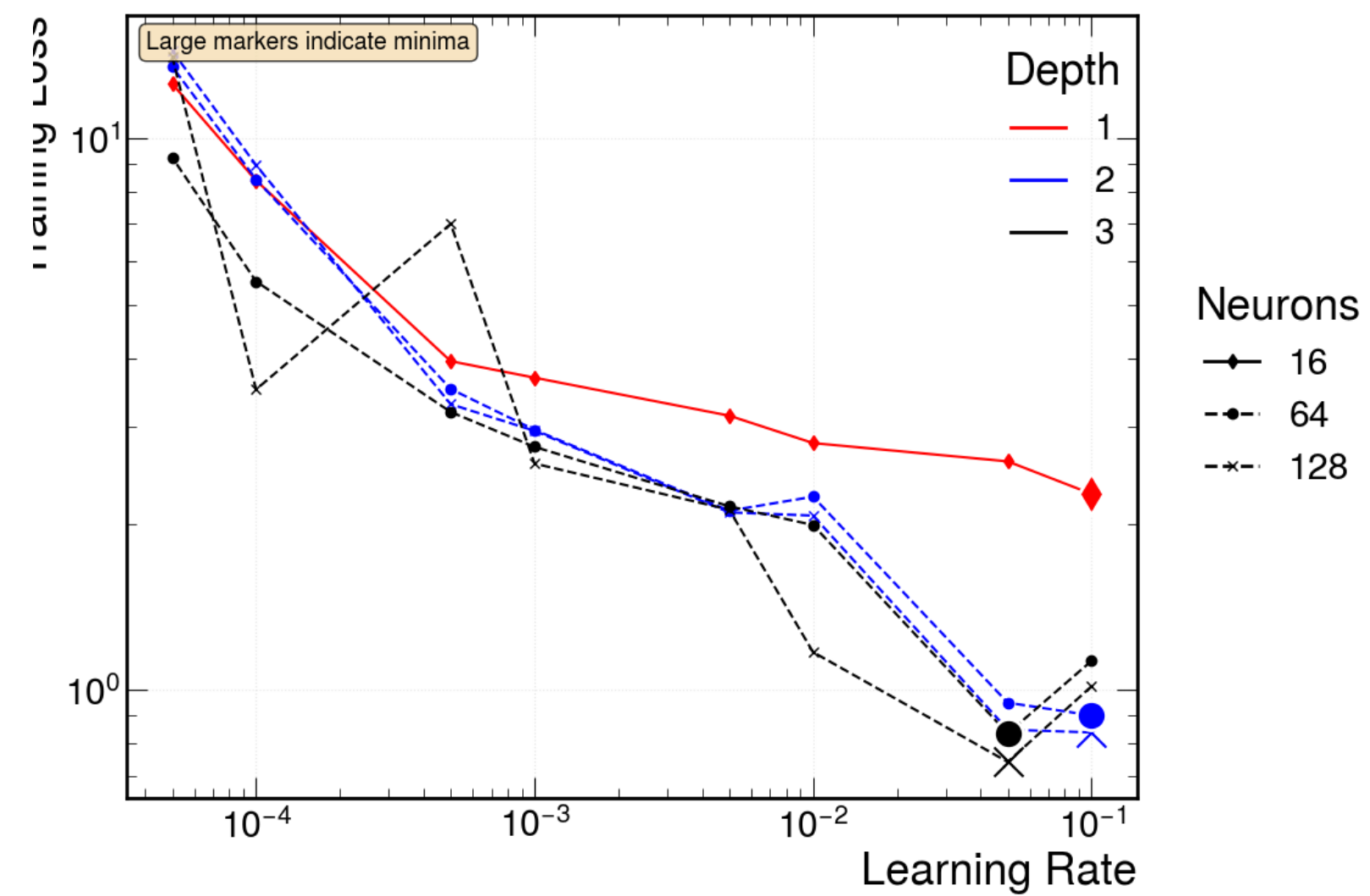
MLP learning rate transfer

Tanh is bounded, so can be more stable
ReLU often performs a bit better - e.g. more expressivity

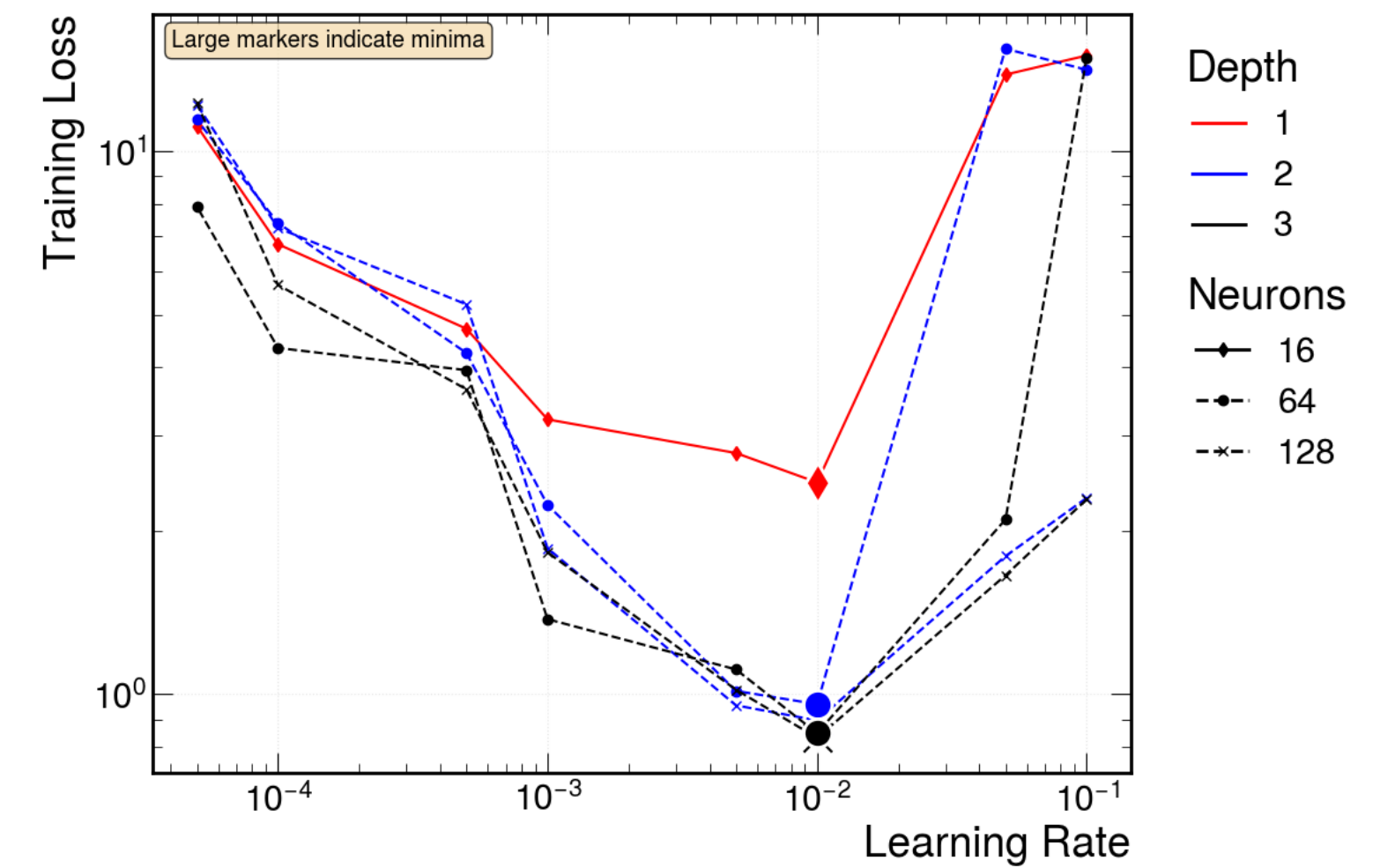
Baseline



μ P with ReLU



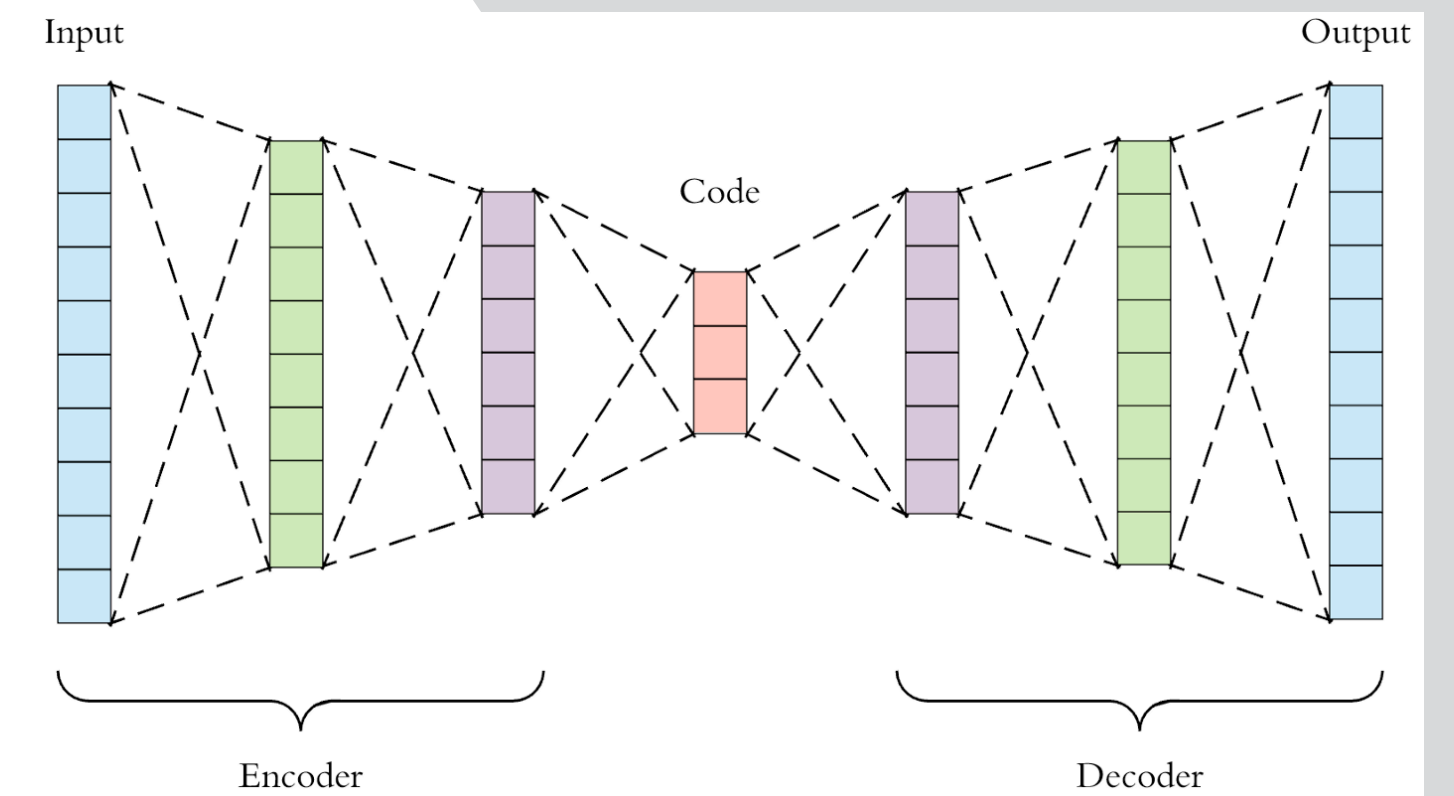
μ P with Tanh



Transfers well

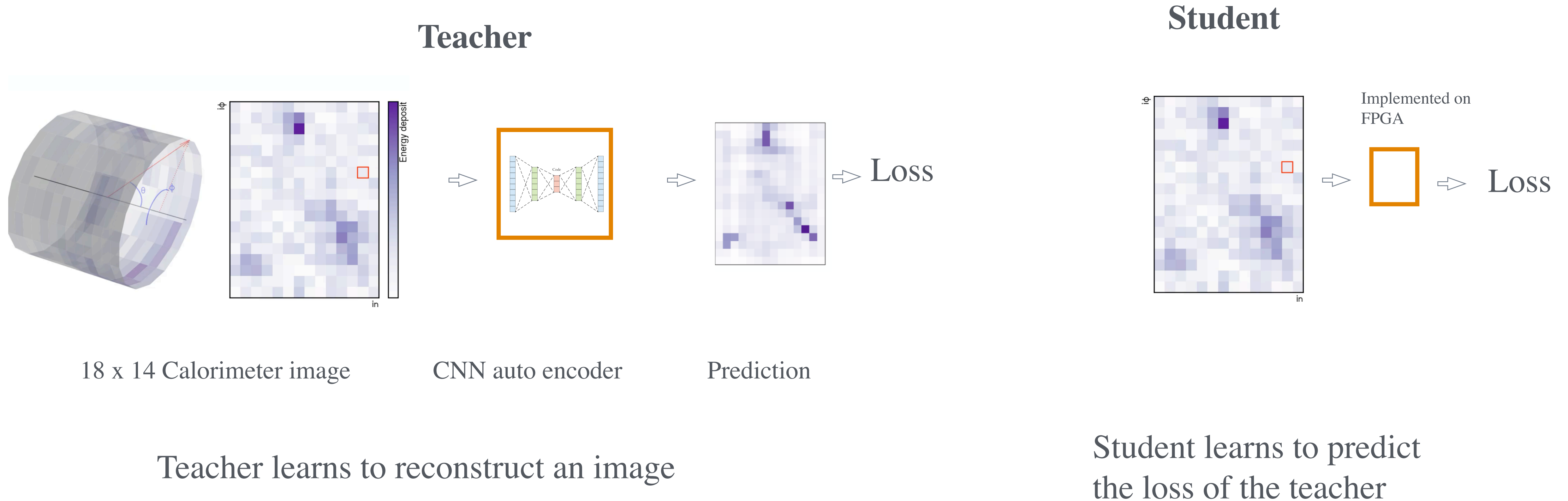
2. Autoencoder with convolutional layers

Anomaly detection for Level-1 Triggering at CMS



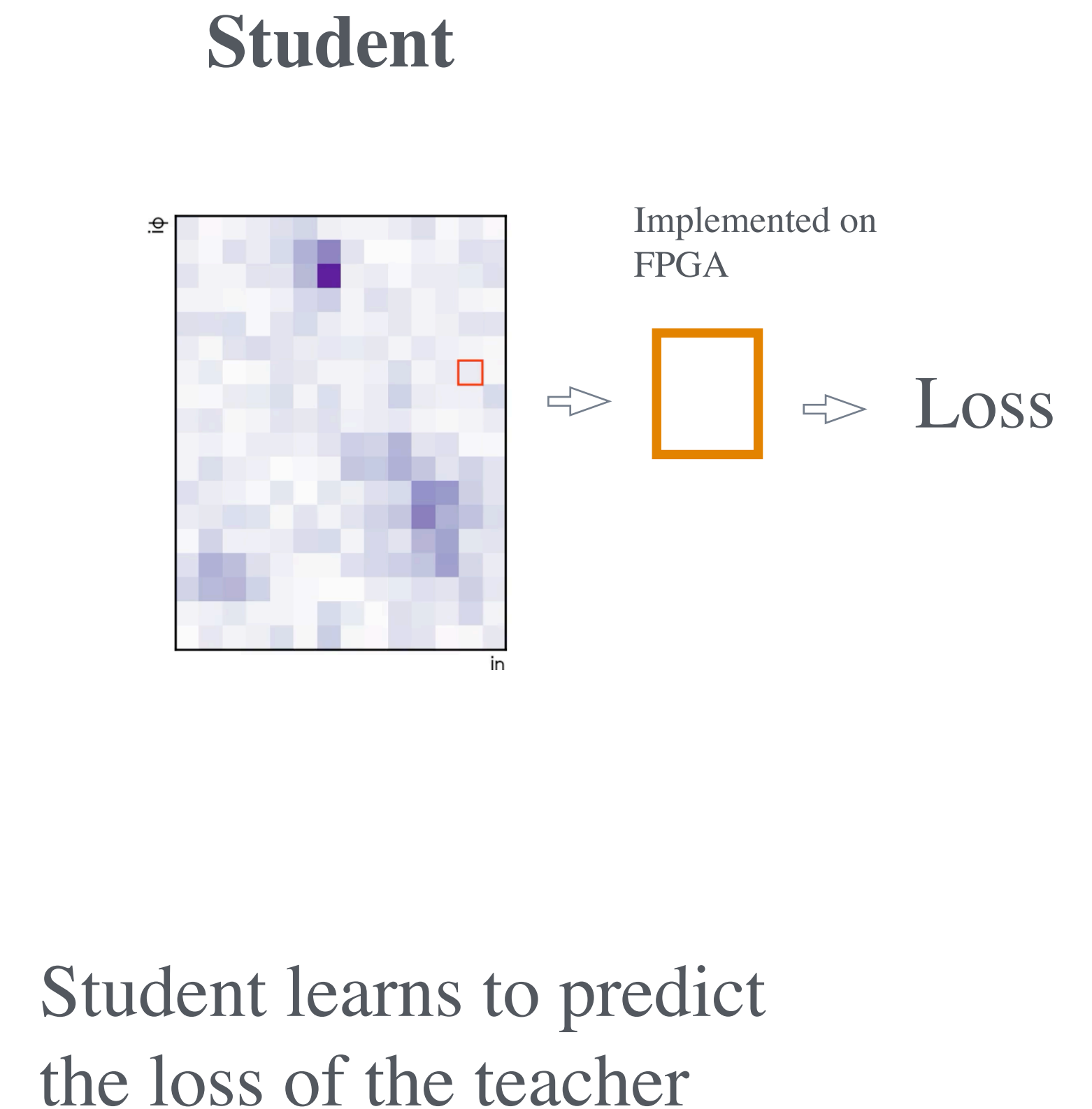
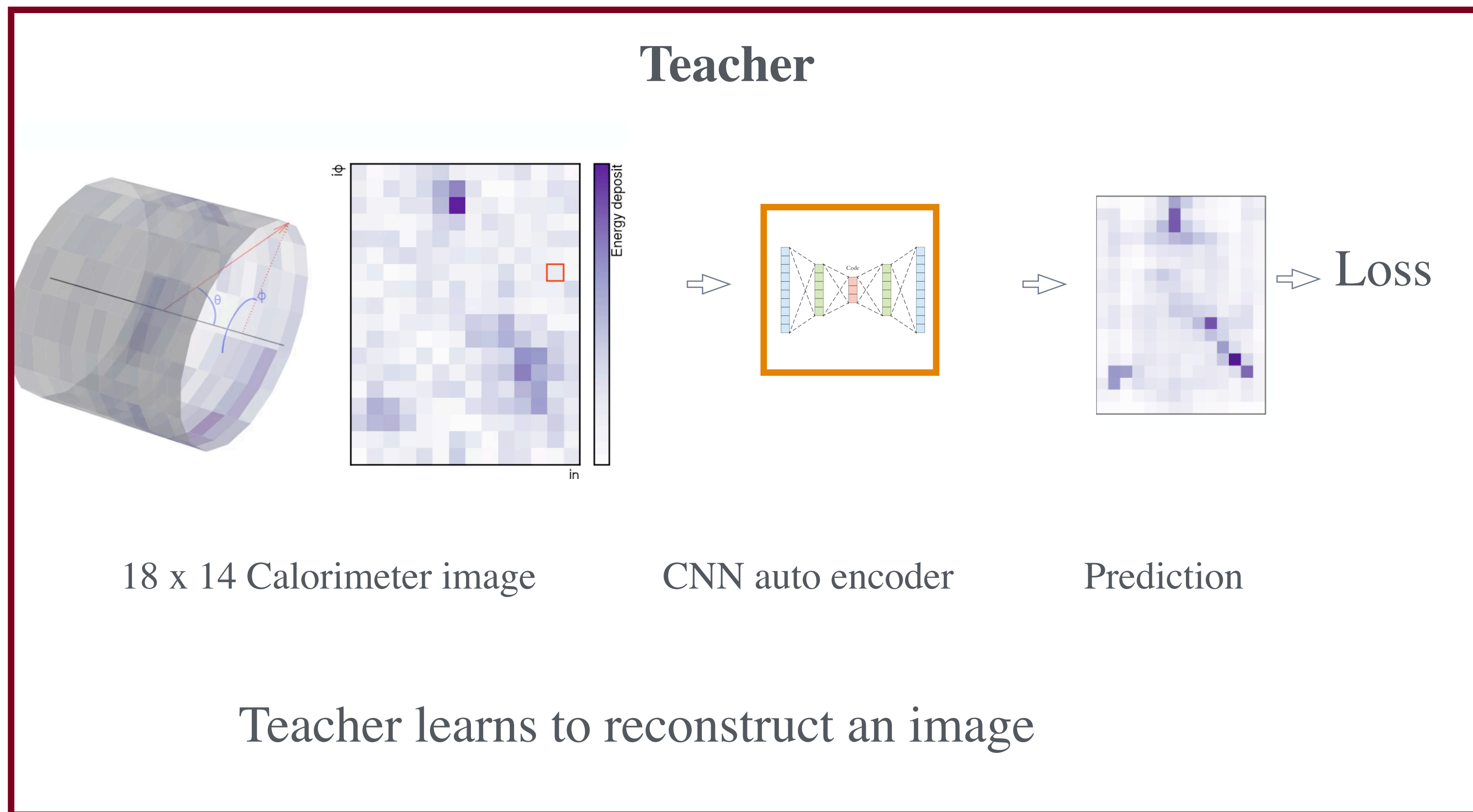
CICADA

Anomaly detection at CMS Level-1 Trigger [2]

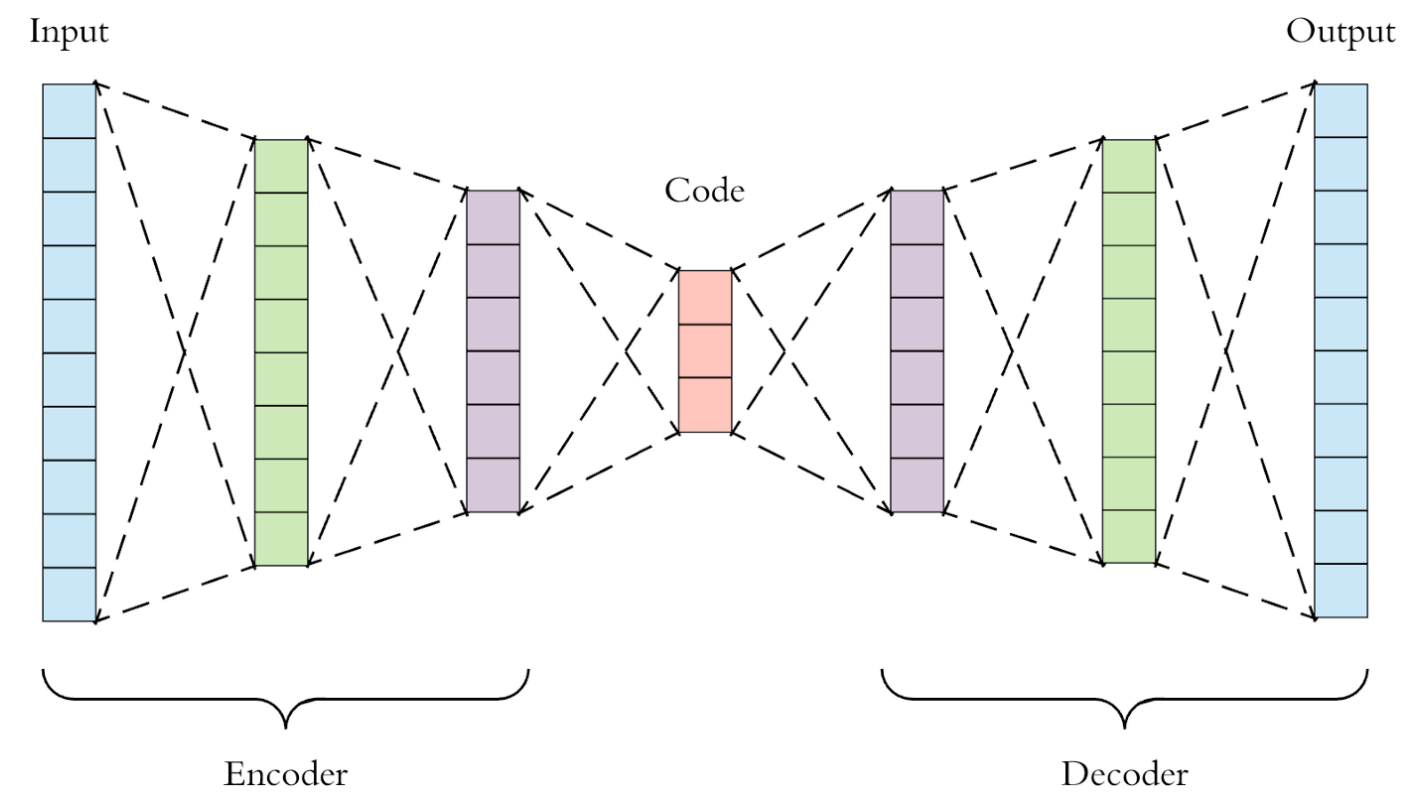


CNN Cicada

Anomaly detection at CMS Level 1 Trigger [2]

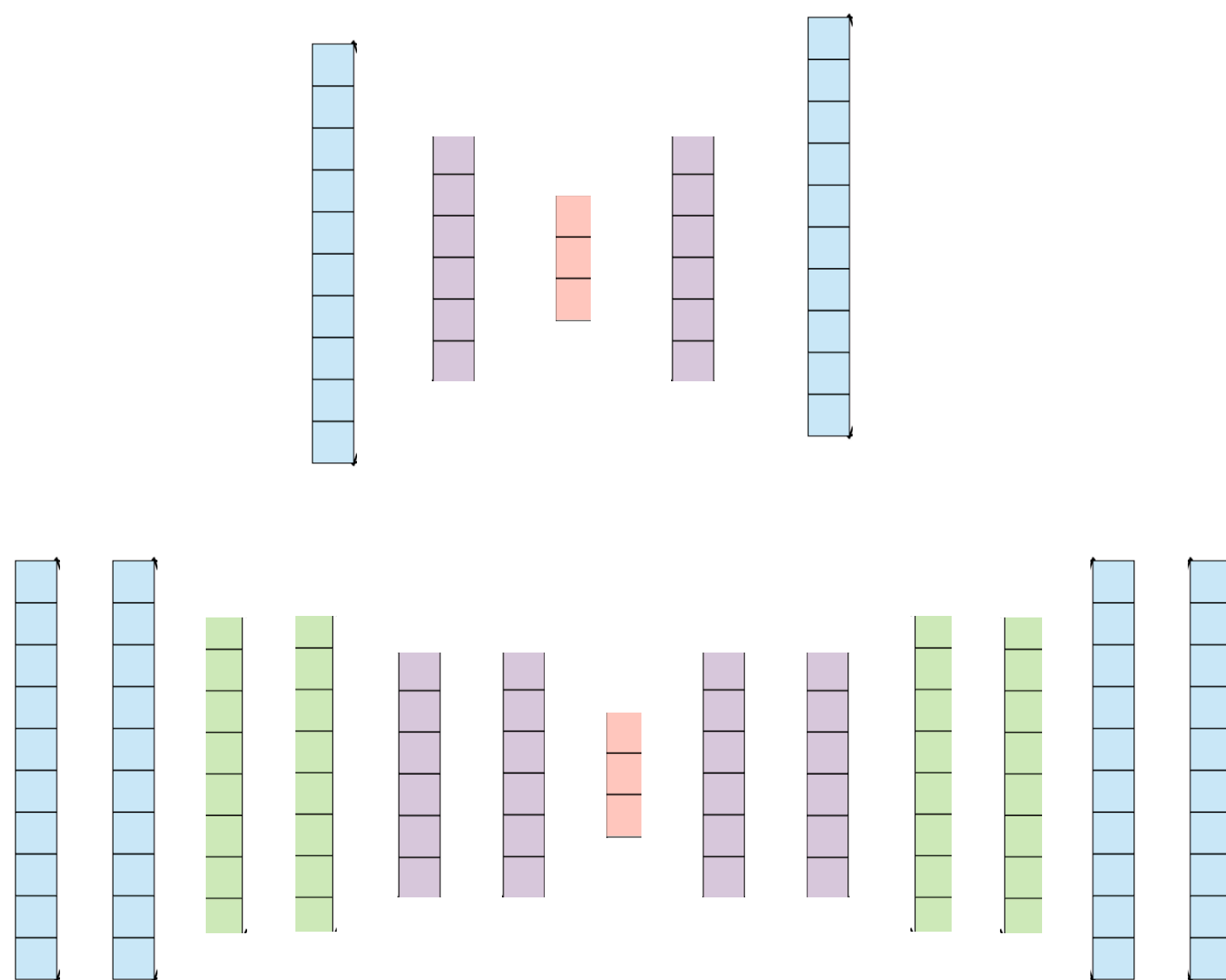


Changing the autoencoder



Create scalable blocks

- ✓ Blocks of convolutional layers
- ✓ Change weight initialisations
- ✓ Add batch normalisation
- ✓ Change readout layer to muP

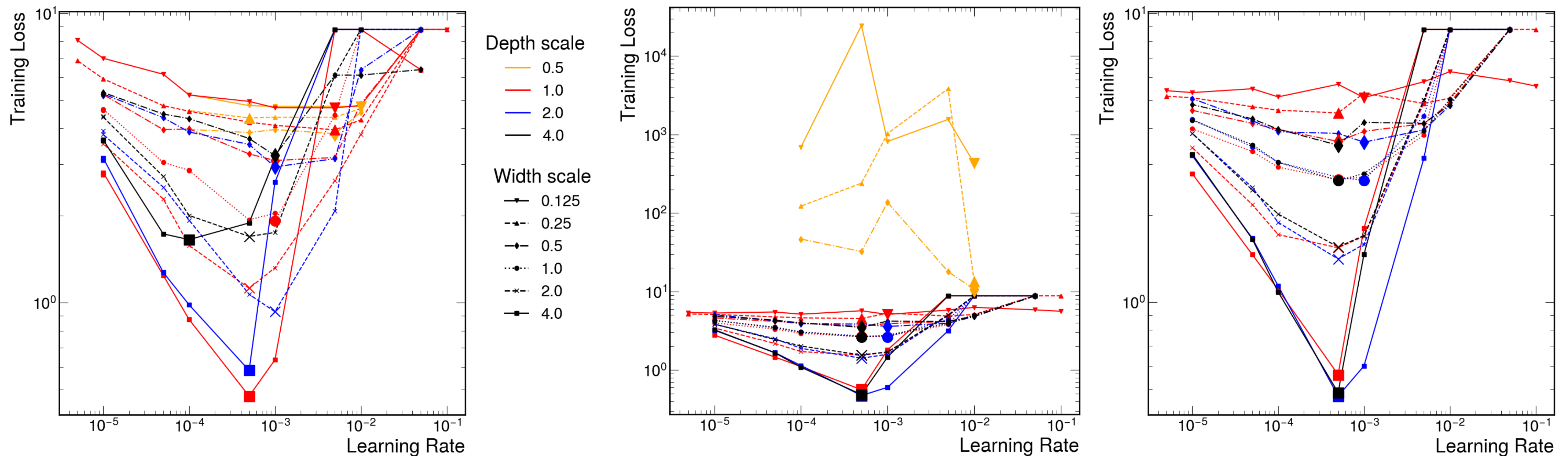


Autoencoder learning rate transfer

Baseline

μP

μP without 0.5 depth scale



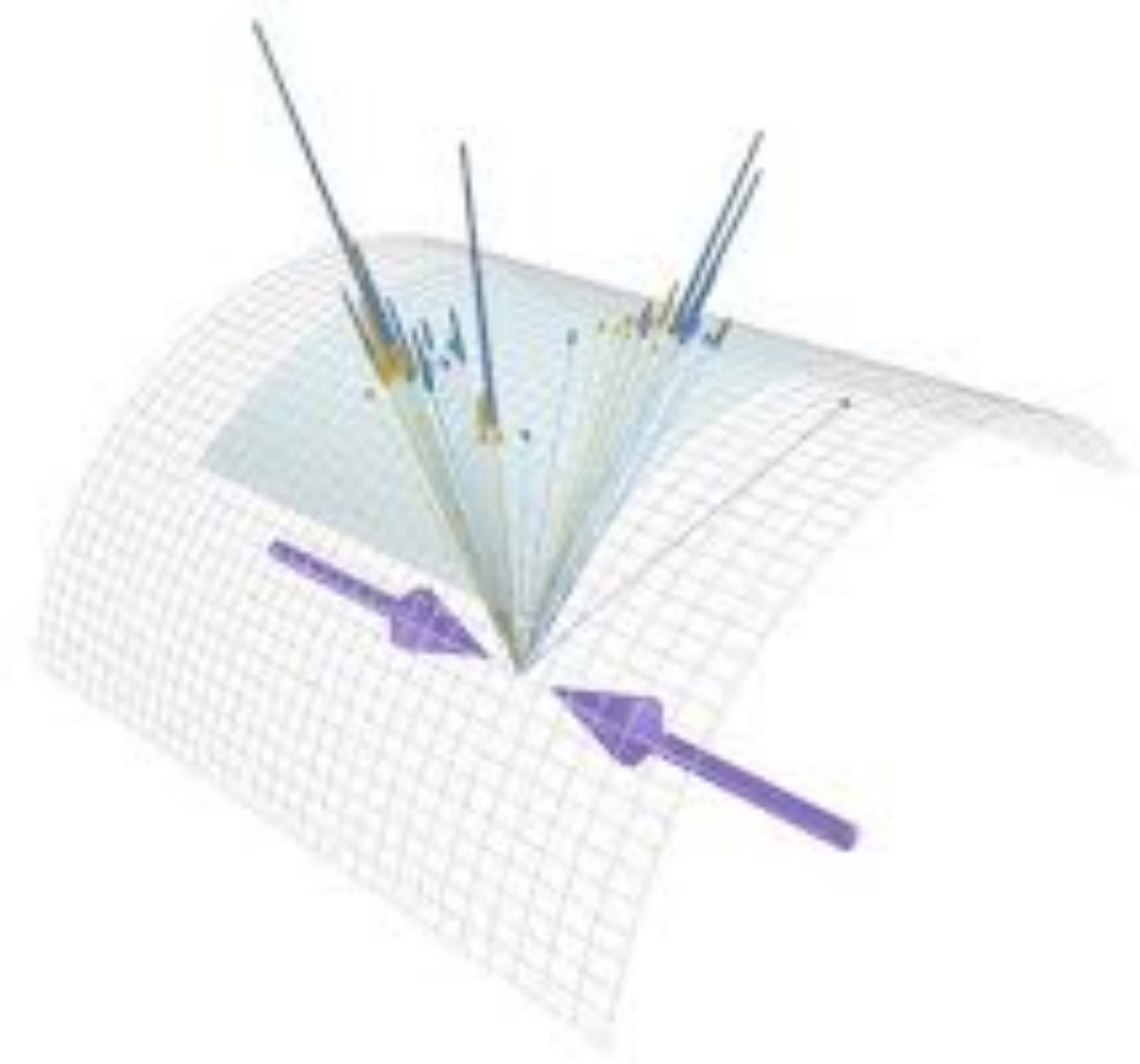
Transfers well - up to a point

If we go to 1/4 of the parameters, it breaks down

Small discrepancies - likely within uncertainties

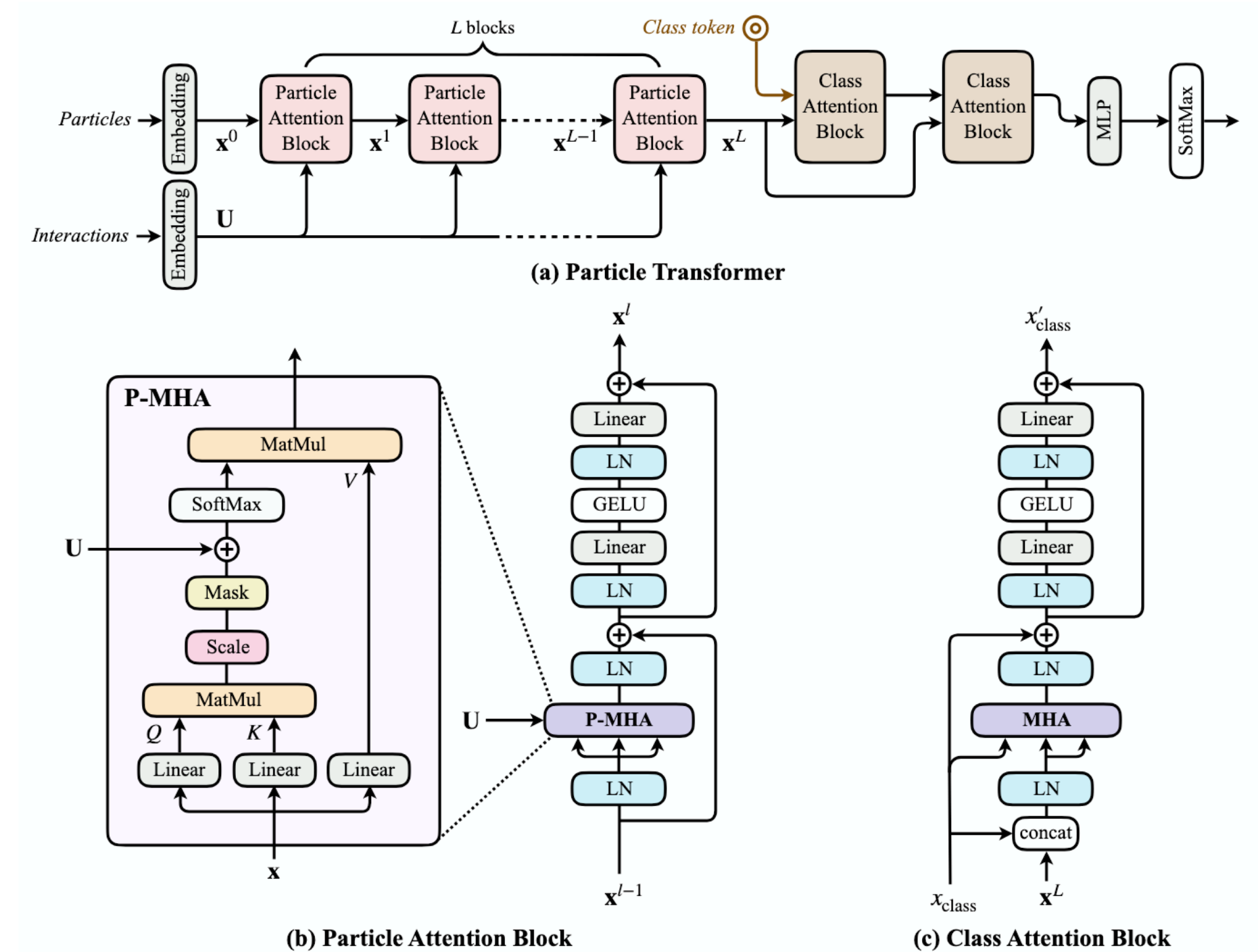
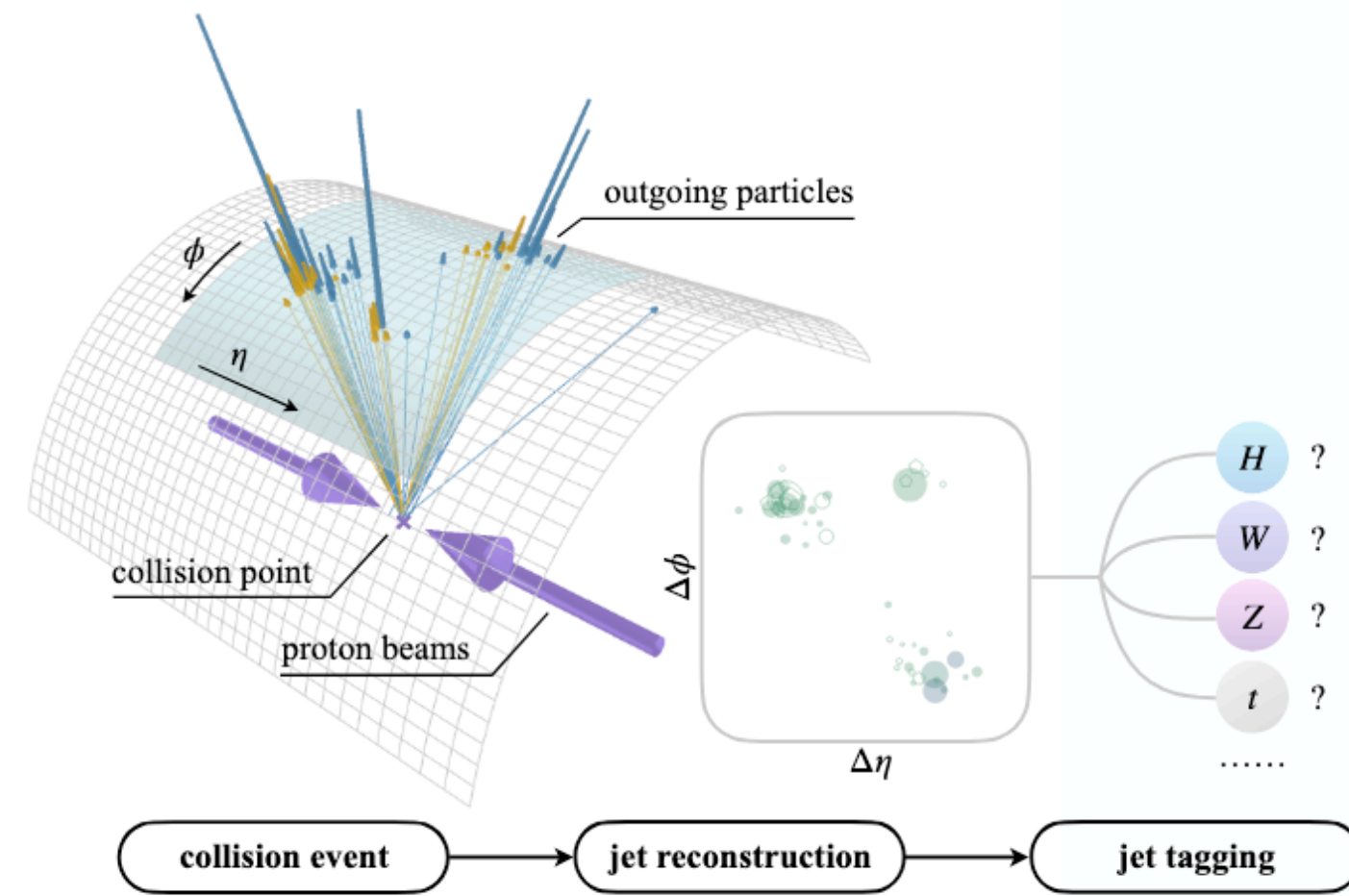
3. Transformer

Jet tagging



Particle transformer

Jet tagging [3]



The SOTA architecture for jet tagging

Has not been fully hyperparameter tuned since it is very big

Has a quite standard transformer and a more bespoke transformer

[3] [Huilin Qu et al, paper here](#) and code [here](#)

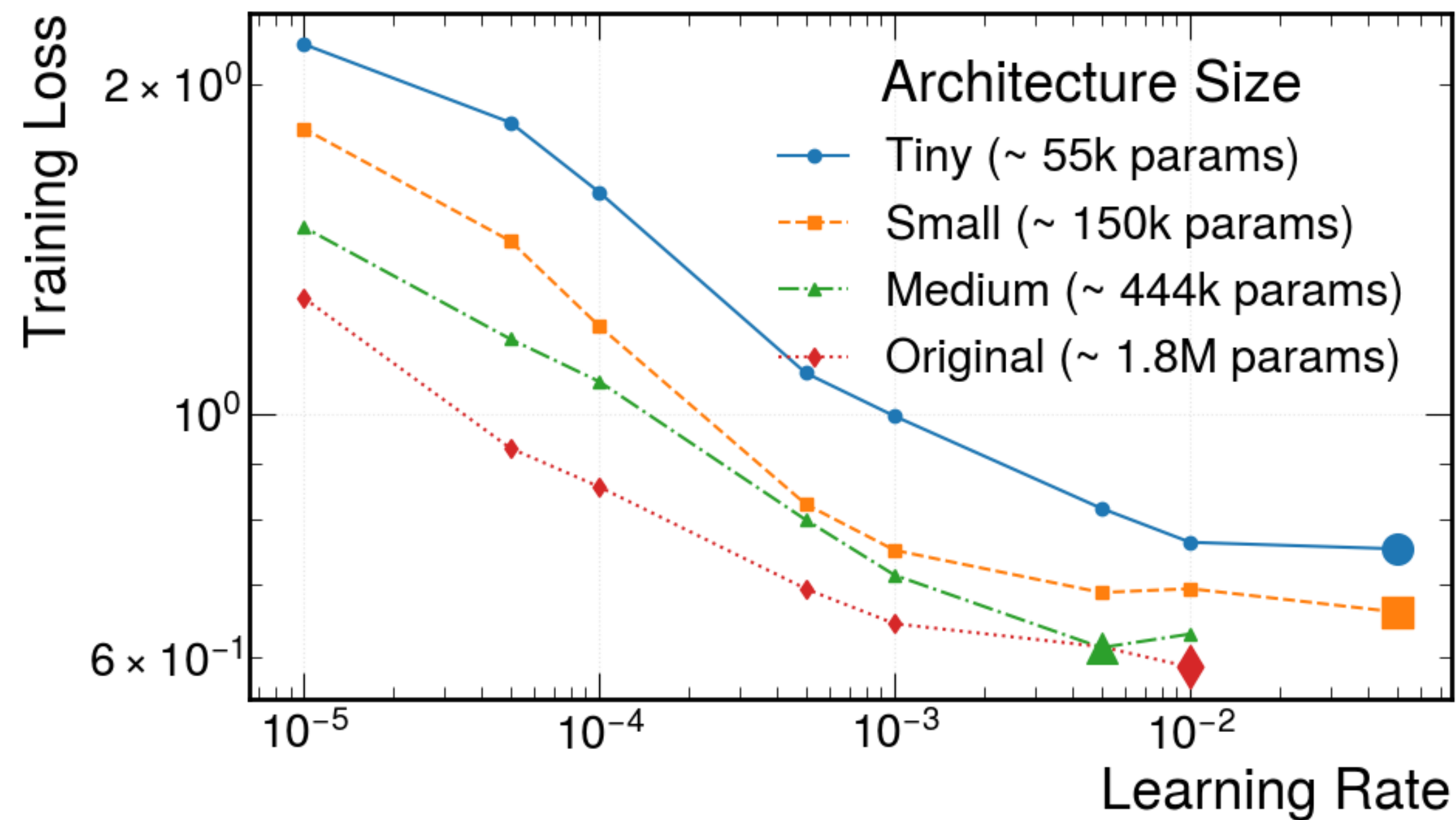
Changing the transformer architecture

Name	Embed dim	Pair embed dim	Heads	Layers	Parameters
Tiny	[16,64,16]	[8,8,8]	2	2	~50k
Small	[32,128,32]	[16,16,16]	2	4	~150k
Medium	[64,256,64]	[32,32,32]	4	6	~444k
Original	[128,512,128]	[64,64,64]	8	8	~1.8M

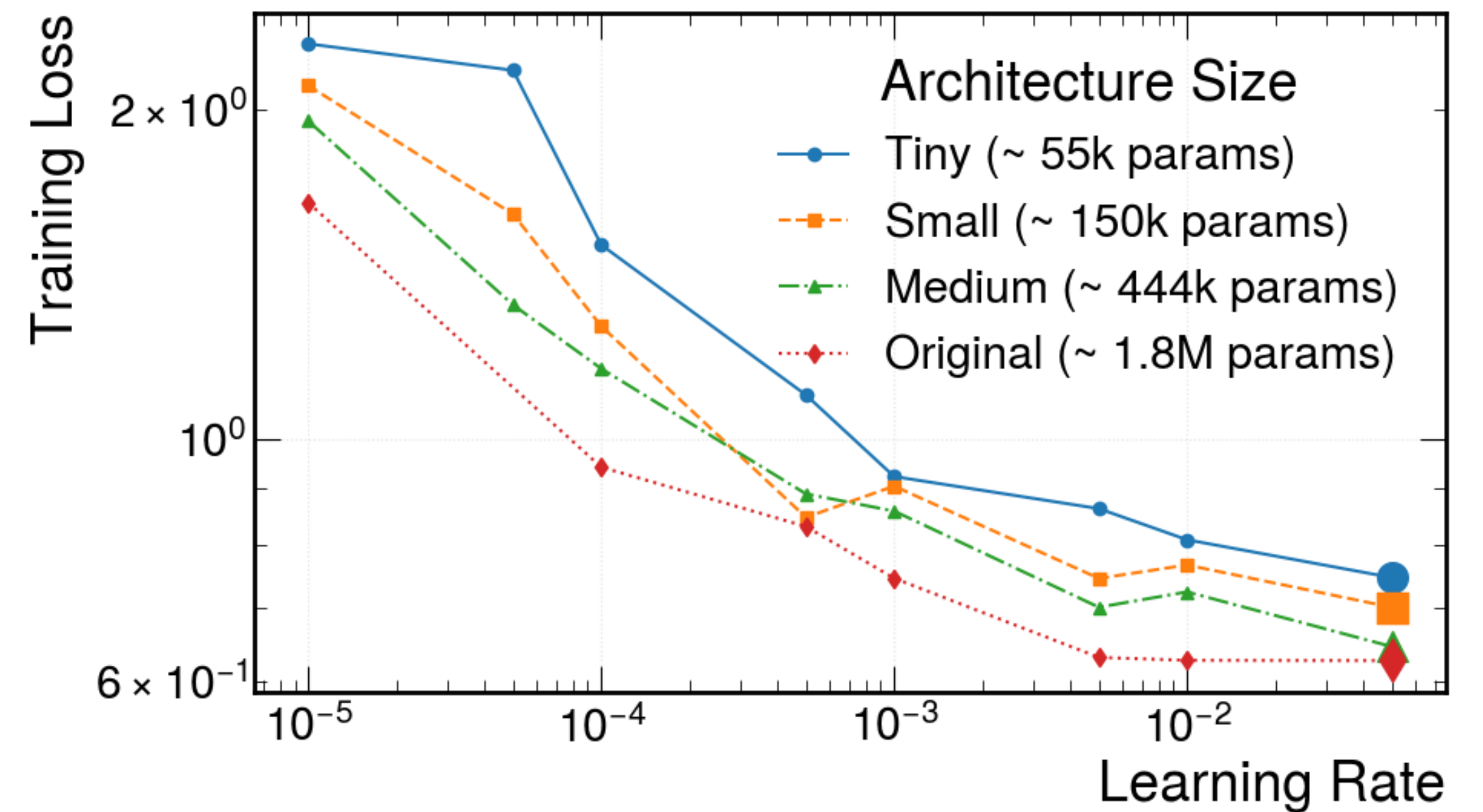
- ✓ All linear layers -> muPLinear
(scales by $\frac{1}{\sqrt{in_{dim}}}$ and special weight initialisation)
- ✓ multihead attention -> muP
multihead attention
attention logits = $(q @ k.T) / d_k$
- ✓ Readout layer -> muReadout
(initialises weights to zero)

Particle transformer transfer

Original



μP



Successfully transferred from a complex transformer architecture
Reducing parameters from $O(10^6)$ to $O(10^4)$

Caveats: most of the literature doesn't go this small, things often break down around 100k parameters
Converges on a relatively high learning rate

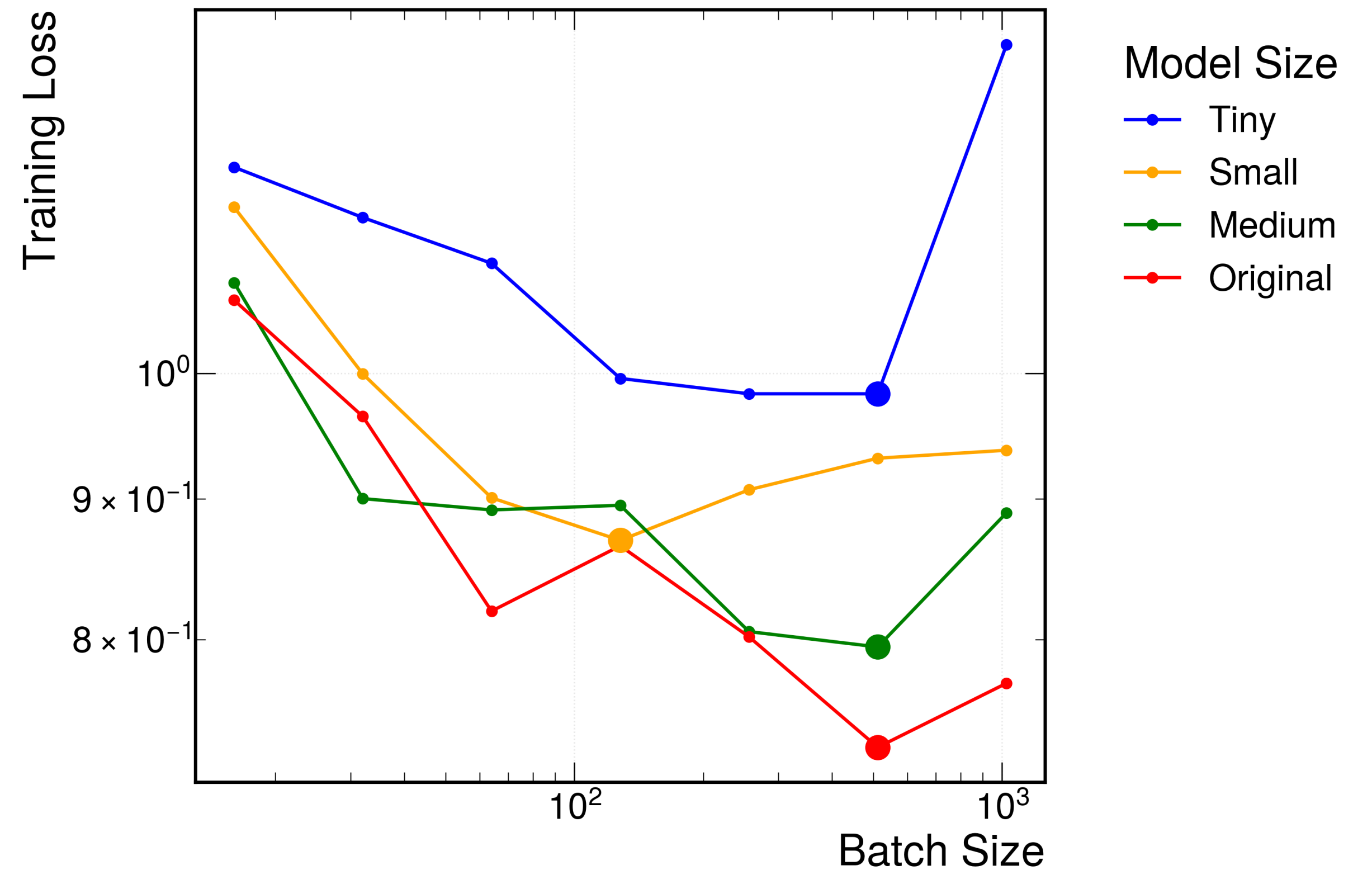
Batch size transfer

Batch size is one of the few other hyperparameters that should transfer

Testing this for the particle transformer

Transfers fairly well, but not perfectly

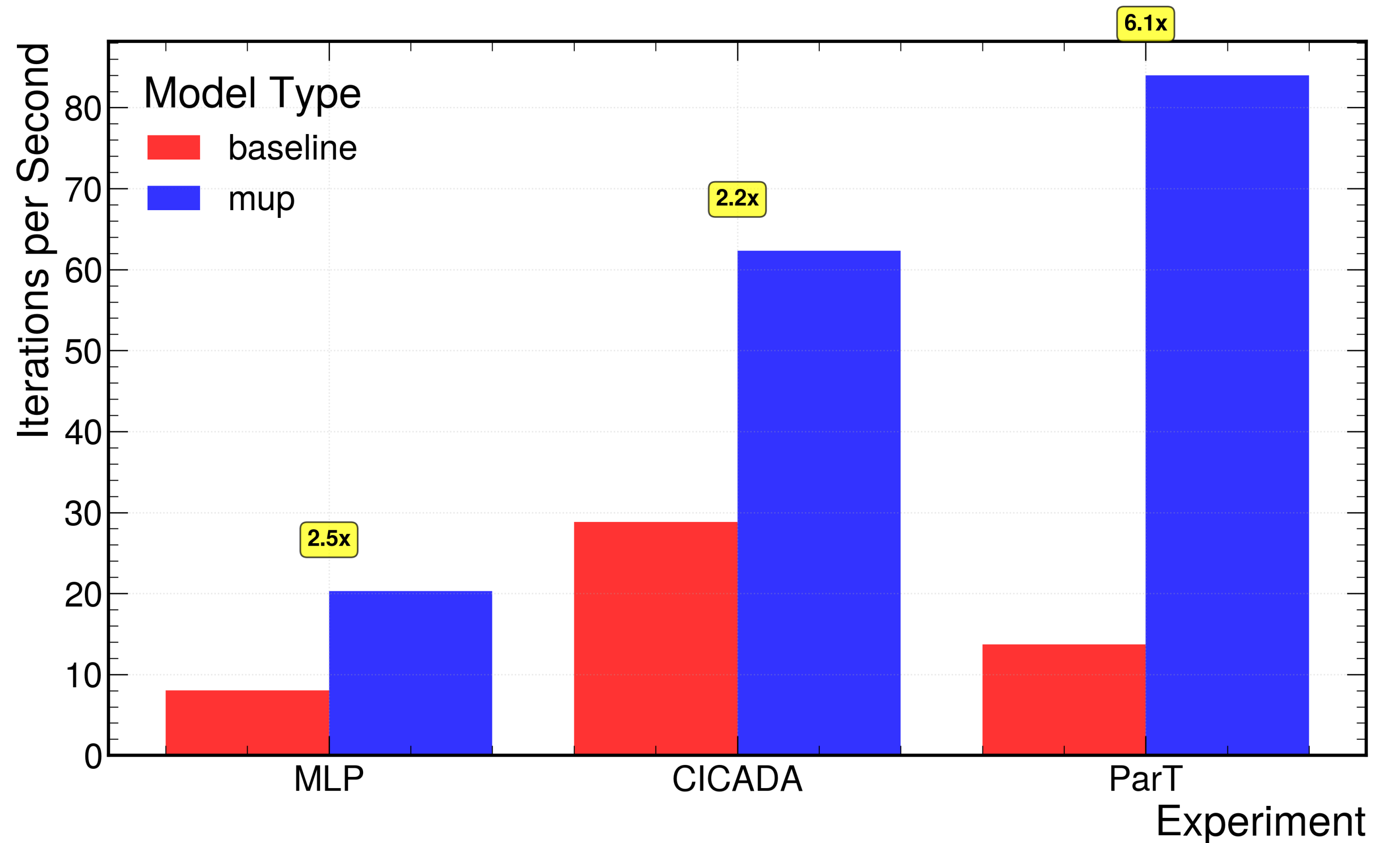
Further work to be done



Time saved

Comparing the baseline to the smallest experiment that transferred

In reality will often have even larger speedups since smaller networks often converge quicker



What we have shown

Hyperparameter transfer is possible - even for non-standard ML workflows

The training loss is at least as good as the baseline models

This enables good speedups



General advice

Mainly learning rate and batch size that are guaranteed to transfer

LLMs are not very good at implementing the muP changes unless you prompt very well; I recommend following the [mup package](#)

Standard architectures should be fine - more complex could need a little testing

You don't need to make plots like the ones I've shown - just make a small network and assume it transfers (if it's fairly standard)

Outlook

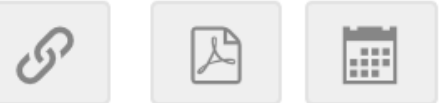
Gage will give a more detailed talk at ACAT on Tuesday

Take our ML in HEP survey!



 ML survey

Hyperparameter Transfer for Graph Transformers



 9 Sept 2025, 17:20

 20m

 ESA B

Oral

Track 2: Data Analys...

Track 2: Data Analysis -...

Speaker

 Gage DeZoort (Princeton University (US))

Description

Modern machine learning (ML) algorithms are sensitive to the specification of non-trainable parameters called hyperparameters (e.g., learning rate or weight decay). Without guiding principles, hyperparameter optimization is the computationally expensive process of sweeping over various model sizes and, at each, re-training the model over a grid of hyperparameter settings. However, recent progress from the ML theory community has given a prescription for scaling hyperparameters with respect to model size such that (1) the optimal hyperparameters identified for small models of a fixed architecture are the same for their larger counterparts (hyperparameter transfer) and (2) larger models perform better than their smaller counterparts (limiting behavior). When satisfied, these desiderata yield large computational savings and stable performance useful for computing, for example, neural scaling laws. In this talk, we will present a recipe for achieving hyperparameter transfer and limiting behavior in graph transformers, transformer variants combining simple message passing with sparse attention computed over the edges of each input graph. Though relatively new, graph transformers have been shown to outperform simple GNNs and transformers on a variety of benchmark tasks, and have particular relevance to scientific datasets where edges may encode known physical interactions and measurements. We will demonstrate the promise of these principled graph transformers on benchmark datasets and encourage discussion about how these results may be extended to tackle more challenging scenarios in particle physics.

Further reading: Feature learning in infinite width

A standing mystery in ML for a while was why overparameterized neural nets worked well

Work on Neural Tangent Kernel (NTK) showed that in the infinite limit, gradients become very small and weight initialisation becomes Gaussian by the central limit theorem

The influence of any individual neuron becomes negligible, and gradient updates become infinitesimally small. The output of the activation functions therefore become close to constant

This means we are then just changing the weights to scale a linear combination of our features

This is equivalent to a kernel regression with a fixed Gaussian kernel, which is a well understood, convex problem