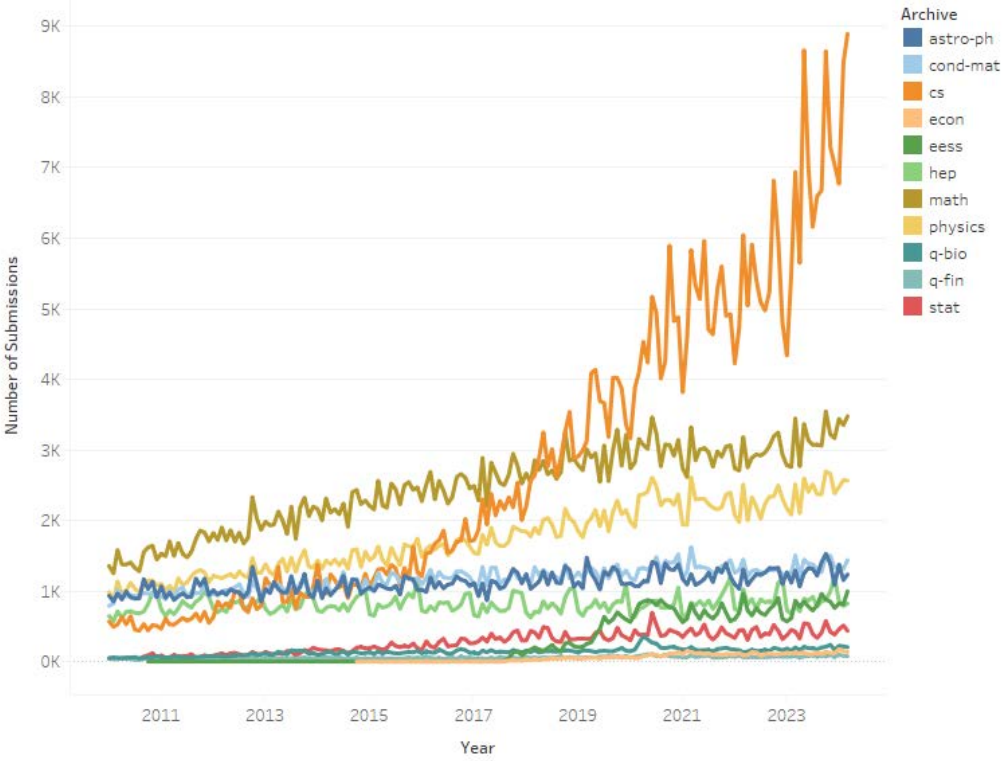
Machine Learning

Kevin Pedro (FNAL)

July 26, 2024

Intro

- Machine learning is a broad and rapidly growing topic
- No way to cover all of it in just one lecture!
- Goals for today:
    - Get a firm grounding in the basics
    - Look at some cutting-edge HEP applications
- Everything in the middle is left as an exercise for the listener
- Philosophical arguments about usage of “machine learning” vs. “artificial intelligence” are also beyond the scope
    - Practical consideration: funding agencies give you money if you say “AI”



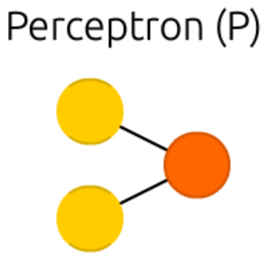
ML-related categories (AI, CV, LG) in CS archive are about ~50% of total submissions

https://info.arxiv.org/about/reports/submission_category_by_year.html

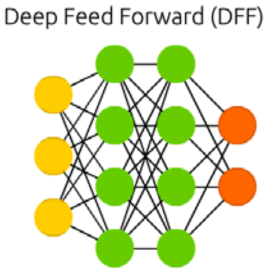
What is AI/ML?

> “AI is whatever hasn’t been done yet.”
> – Douglas Hofstadter

- ML is *function approximation*:
  - ➢ map inputs to outputs, $\vec{x} \mapsto \vec{y}$
  - o $\vec{y} = F(\vec{x})$ unknown, probably not analytic
    → try to find approximation $\vec{y} \approx F'(\vec{x}; \vec{w})$ by optimizing *weights* $\vec{w}$ (in general, any parameters)
- Deep learning:
  - o Use thousands, even millions of weights
  - o Use many *layers* with intermediate *features* derived from inputs
    - ▪ More “neurons” → more multiplications

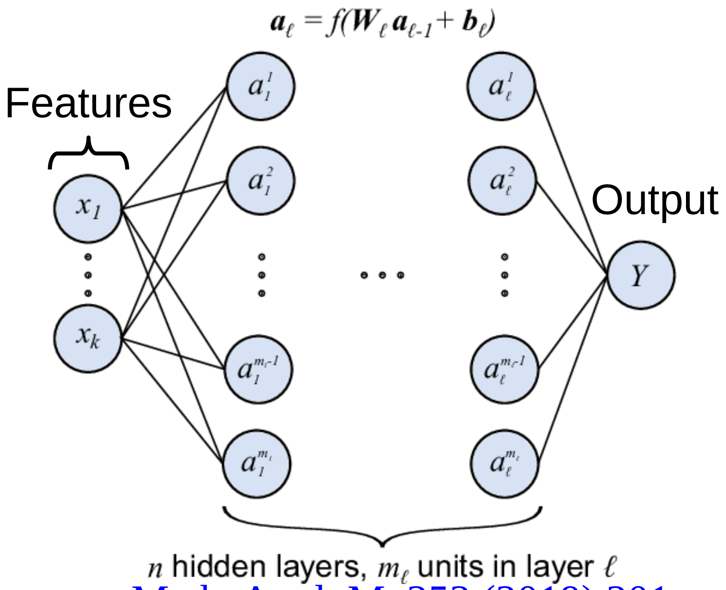


The Neural Network Zoo



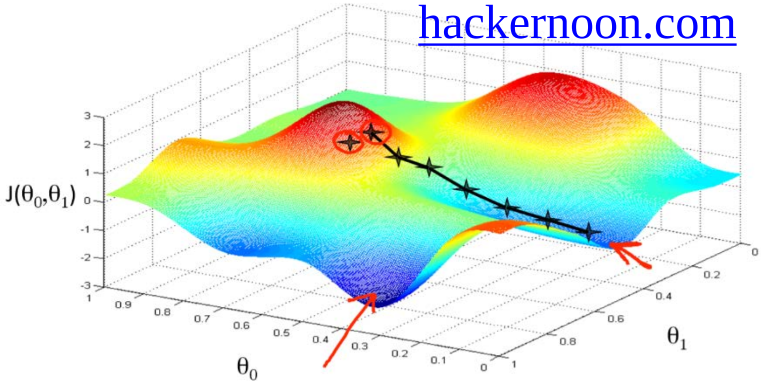
Deep Neural Networks

- Ingredients for a neural network (NN):
  - “Architecture”: implementation of mathematical operations
    - At least one layer with multiple nodes
      - Multiple layers connected to each other → *deep* (DNN)
    - For now: fully-connected network, also called multilayer perceptron (MLP) or feed-forward
  - Data: set of input features $\vec{x}$ and expected output values $\vec{y}$
  - Objective: function to compare NN output with expected output
- Training a (D)NN:
  - Modify node weights to minimize objective
- Seems simple enough…


$\boldsymbol{a}_\ell = f(\boldsymbol{W}_\ell \boldsymbol{a}_{\ell-1} + \boldsymbol{b}_\ell)$
Features
x_1
x_k
a_1^1
a_1^2
$a_1^{m_\ell-1}$
$a_1^{m_\ell}$
a_ℓ^1
a_ℓ^2
$a_\ell^{m_\ell-1}$
$a_\ell^{m_\ell}$
Output
Y
n hidden layers, m_ℓ units in layer ℓ

Comput. Meth. Appl. M. 353 (2019) 201

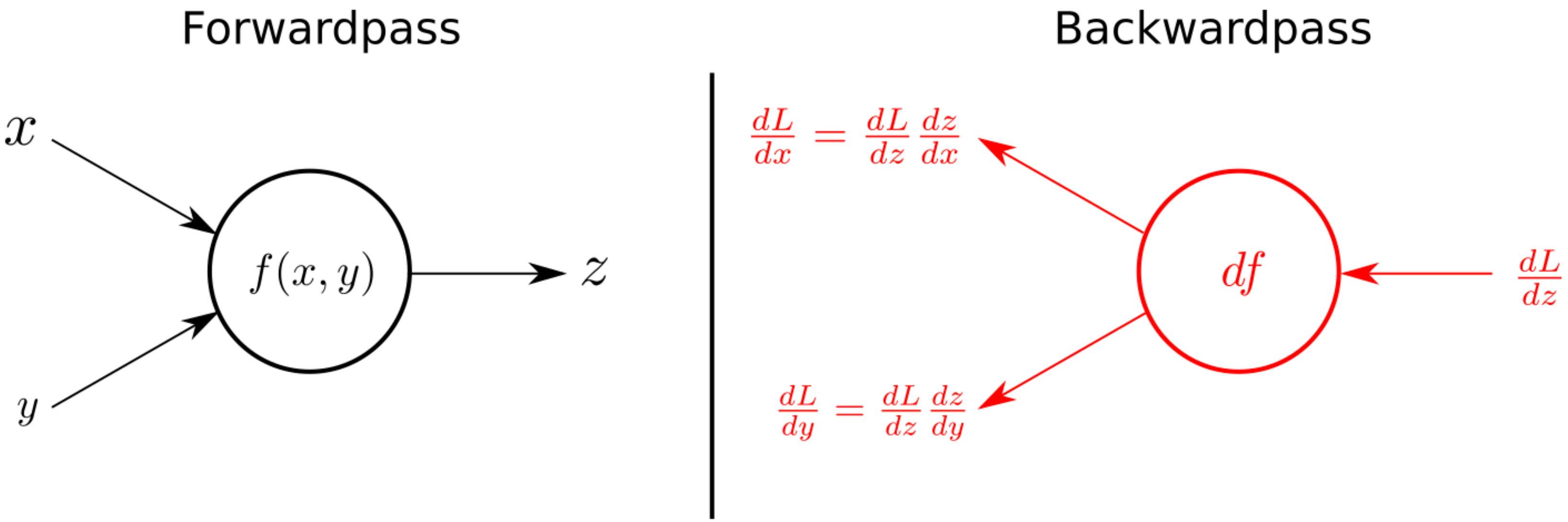
Training



hackernoon.com

- Iteratively modify weights so F′ gets “closer” to $\vec{y}$ (training data)
  - “Closer” defined by objective, also called a *loss function*
  - Use *gradient descent* to follow change in loss
- Gradient space is defined by the *combination* of NN architecture, input data, and loss function
  - Change any of these: change the gradients
  - How to make sure our NNs generalize? We’ll come back to this…
- Several algorithms to perform gradient descent:
  - Stochastic gradient descent (SGD), Adam (Adaptive Moment Estimation)
  - Different approaches to *learning rate* (controls size of update iteration → step in gradient space)
  - All rely on *backpropagation*

Backpropagation



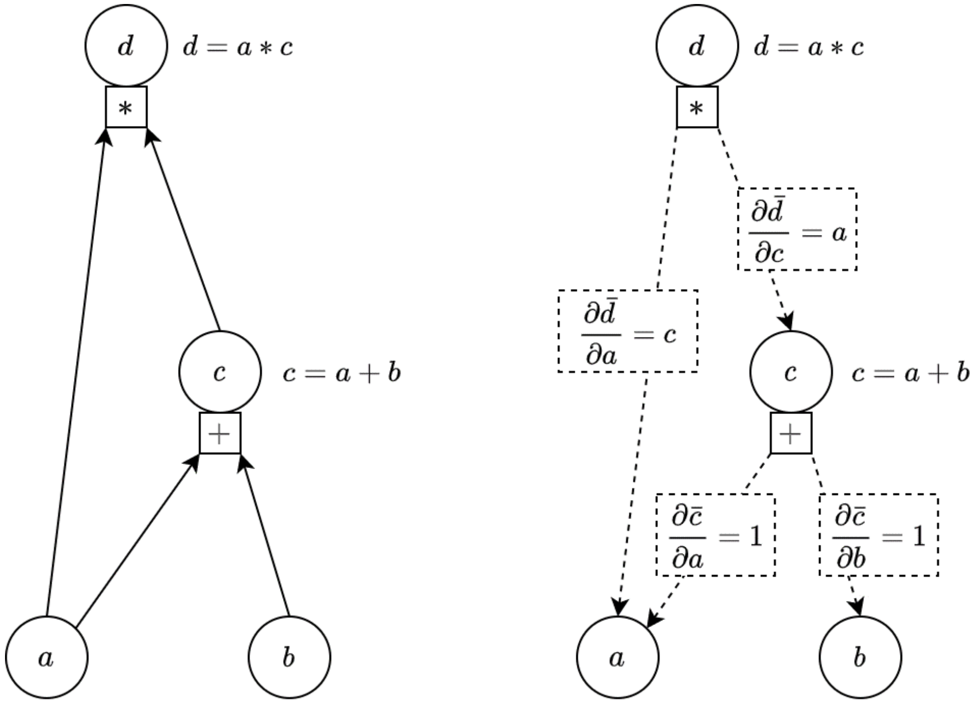
Forwardpass
x
y
$f(x, y)$
z
Backwardpass
$\frac{dL}{dx} = \frac{dL}{dz}\frac{dz}{dx}$
$\frac{dL}{dy} = \frac{dL}{dz}\frac{dz}{dy}$
df
$\frac{dL}{dz}$

- Forward pass: feed input data to current state of NN, multiply by weights, produce output
- Backward pass: compute gradient of loss function with respect to weights
  - This tells us what step to take in gradient space
    - i.e. how to modify the weights, in order to improve the loss function value
- Hidden complexity: “compute gradient”

Automatic Differentiation

- Typical approaches to differentiation:
  - o Symbolic: accurate, but expensive
  - o Numerical: fast, but limited accuracy
- Autodiff is *neither of these*!
- Computational functions largely built from elementary mathematical operations (addition, multiplication)
- Exploit the chain rule to break down complicated derivatives into simple, known operations
  - o Only need local values, not global functions
- Example: find gradient of *d* with respect to *a*

$$\frac{\partial d}{\partial a} = \frac{\partial \bar{d}}{\partial a} + \frac{\partial \bar{d}}{\partial c} * \frac{\partial \bar{c}}{\partial a}$$


d
$d = a * c$
$*$
c
$c = a + b$
$+$
a
b
d
$d = a * c$
$*$
$\frac{\partial \bar{d}}{\partial c} = a$
$\frac{\partial \bar{d}}{\partial a} = c$
c
$c = a + b$
$+$
$\frac{\partial \bar{c}}{\partial a} = 1$
$\frac{\partial \bar{c}}{\partial b} = 1$
a
b

sidsite sidsite.com

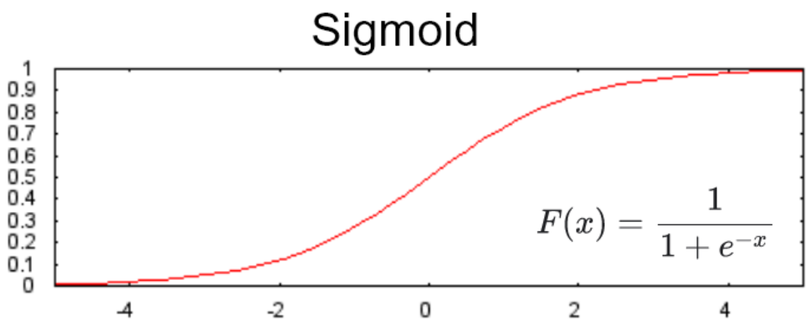
This depicts "reverse mode" autodiff, which is used for backpropagation

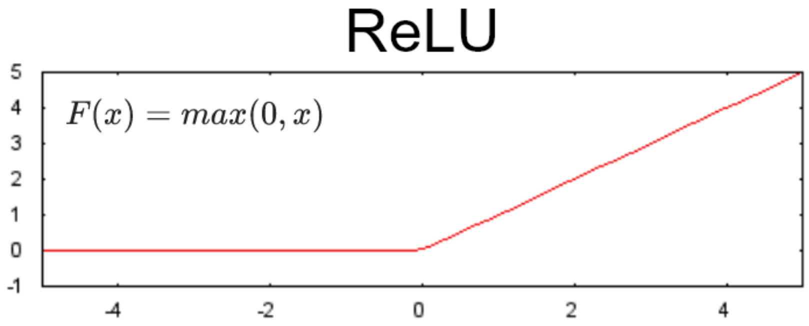
Activation Functions

- Yet another hidden detail:
  - What we’ve depicted so far is just a complicated way of writing down a linear regression: multiplying and summing inputs
- To achieve universal function approximation, need *nonlinearity*

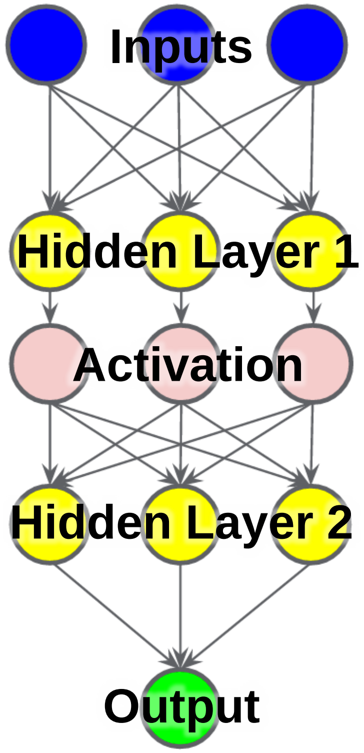
➢ Apply nonlinear functions to each layer

  - Make sure they’re differentiable!





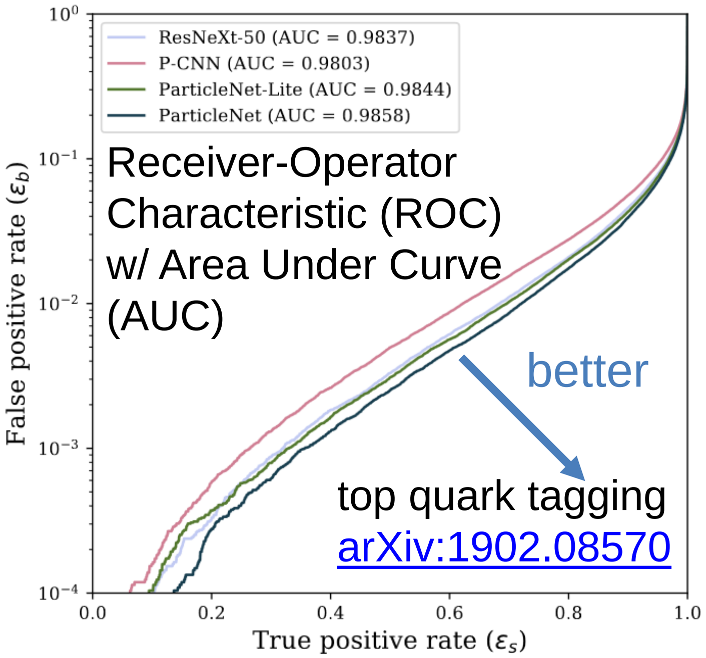
- Examples:
  - Rectified Linear Unit (ReLU) tends to be preferred
    - Fast to calculate, steeper than sigmoid
  - Options like Leaky ReLU can be employed to keep negative side
  - (more at Table of activation functions)



Google

Loss Functions

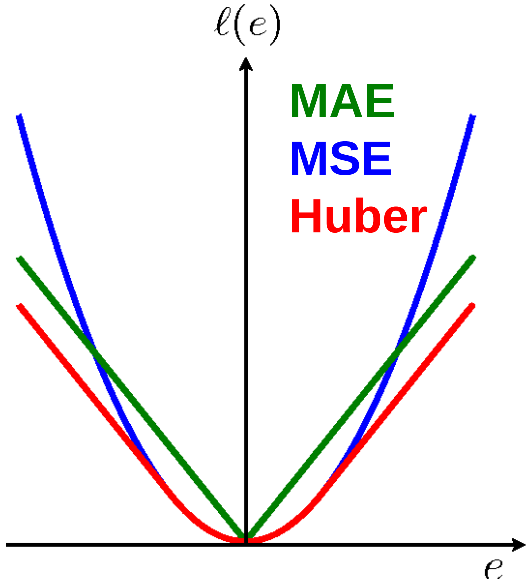
- Typical tasks and their loss functions include:
  - Classification: “is this event signal or background”?
    - Binary cross-entropy (2-class problem, labels are either 0 or 1):
      $p \in \{y, 1-y\}$, $q \in \{F', 1-F'\}$
      $L(p,q) = -\sum p \log q = -y \log F' - (1-y) \log (1-F')$
      - Minimizing BCE ↔ maximizing likelihood
    - Categorical cross-entropy (multiclass problem):
      $\sigma(\vec{z})_i = e^{z_i} / \sum e^{z_j}$ → softmax: maps to (0,1) and $\sum$outputs = 1
      $L(p,q) = -\sum p \log \sigma(q) = -\log(e^{F'_i} / \sum e^{F'_j})$
  - Regression: “what is the mass of these inputs?”
    - Mean squared error:
      $L(F', y) = \frac{1}{n} \sum (F' - y)^2$
    - Huber loss: variation that reduces outlier impact
      $$L(F',y) = \begin{cases} \frac{1}{n} \sum \frac{1}{2}(F' - y)^2, & |F' - y| \leq \delta \\ \frac{1}{n} \sum \delta[(F' - y)^2 - \frac{1}{2}\delta], & |F' - y| > \delta \end{cases}$$



Receiver-Operator Characteristic (ROC) w/ Area Under Curve (AUC)

better

top quark tagging
arXiv:1902.08570

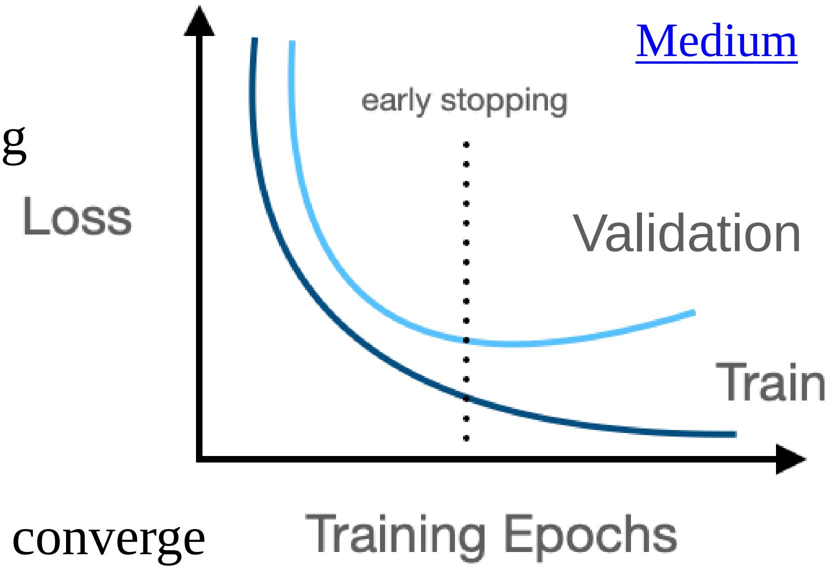


https://api.semanticscholar.org/CorpusID:17940227

Statistical Validity

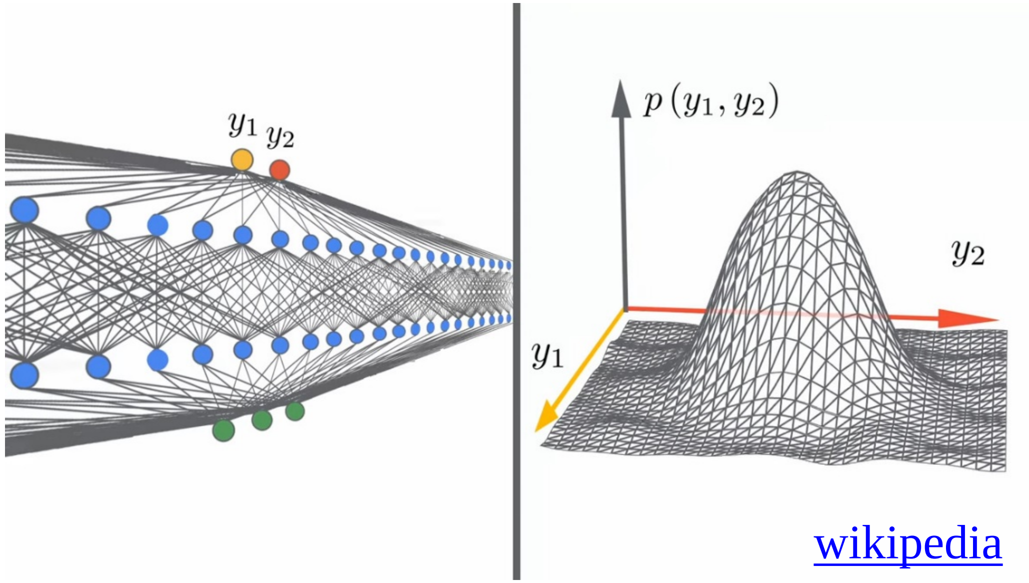
- Always be wary of *overtraining*: learning only the exact input training data rather than generalizing
- First defense:
  - Reserve some data for validation and testing
    - Validation data used to watch loss function behavior *during* training
    - Test data used to evaluate performance *after* training
    - These all must be independent to avoid bias!
  - Can increase this to k-fold cross-validation
- More defenses:
  - Early stopping: avoid over-optimizing once gradient descent starts to converge
  - Batching: shuffle training data during each training period (epoch), compute loss in each batch
    - Mini-batching: use random subset of data during each epoch
- General principle: regularization
  - Vague term, but important concept
  - Any change that encourages NN to generalize: can be in data, architecture, loss function, etc.

Medium



Universal Approximation Theorem

- Theoretically, even a single-layer NN can approximate any function
  - …if infinitely wide
- Also theoretically, gradient descent should converge to a good minimum
  - if objective is convex
- So we can be sure to get the right answer…
  - if we have an infinite network, infinite data, infinite training time, and everything is well behaved
- What should we do in the real world?
  - Real training algorithms have various parameters that have to be optimized separately: called “hyperparameters”



wikipedia

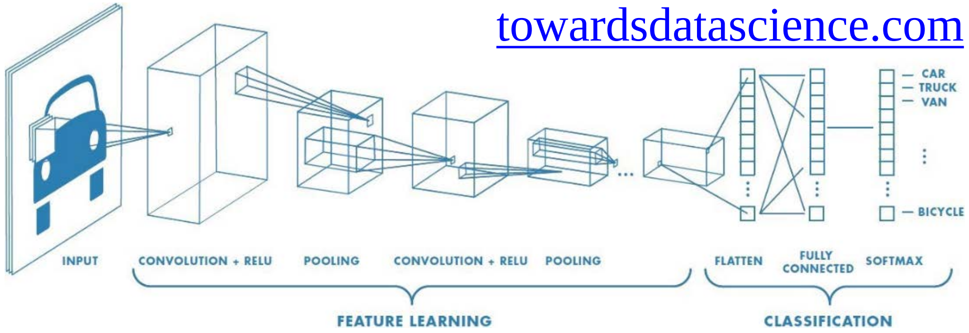
Inductive Bias

- ML researchers’ goal is to see what NN can learn: try to minimize bias
- HEP researchers’ goal is to do physics
  - It’s okay (and even advisable) to “help” the NN learn
- Like regularization, inductive biases can be added anywhere:
  - Data: feature engineering
    - Less necessary for NNs than other ML methods like BDTs
    - But can still be important to inject physics knowledge
  - Architecture:
    - Introduce assumptions about how inputs are related and what computations should be performed (going beyond MLPs)
  - Loss functions:
    - Enforcing physical constraints, preventing unwanted behavior

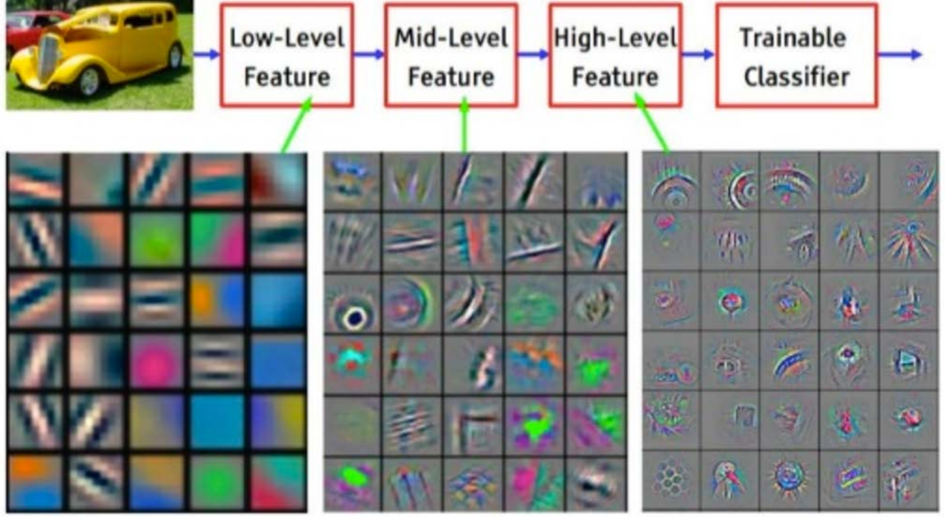
Convolutional Neural Networks

- **Convolution**: combine neighboring pixels according to matrix of weights
- *Translational invariance*: apply same operation to each subset of data
- *Locality*: assumes that pixels only relate to their neighbors
- *Feature engineering*: automatically derive features at different levels of complexity (edges, corners, etc.)

➢ Application to image recognition started modern AI revolution in 2012 (AlexNet)

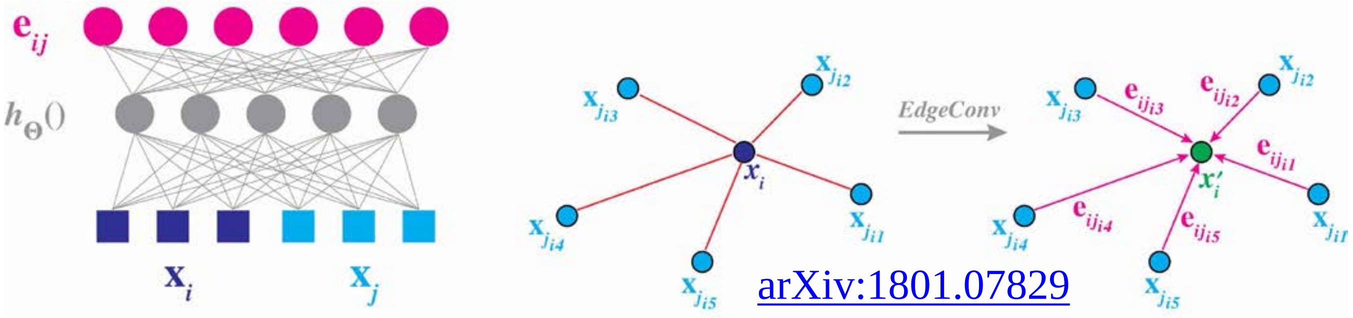

towardsdatascience.com

towardsdatascience.com



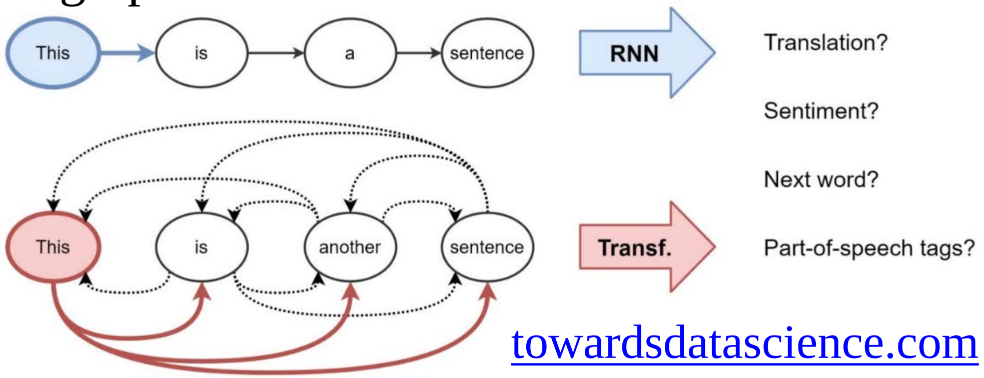
Graph Neural Networks

- Generalize convolutions → *message passing* w/ graphs (*nodes* & *edges*)
  - Derive new features for node x_i using neighbors x_j
  - Can even assign features to edges



arXiv:1801.07829

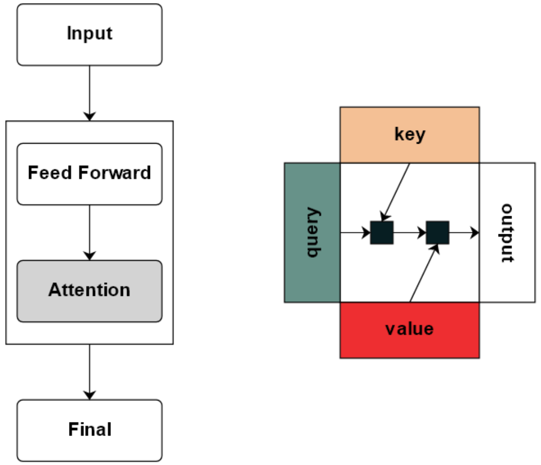
- Aside: recurrent networks (RNNs) previously used for language processing
  - Now supplanted by "Transformers" that use "attention"
  - Conceptually, these are just graphs



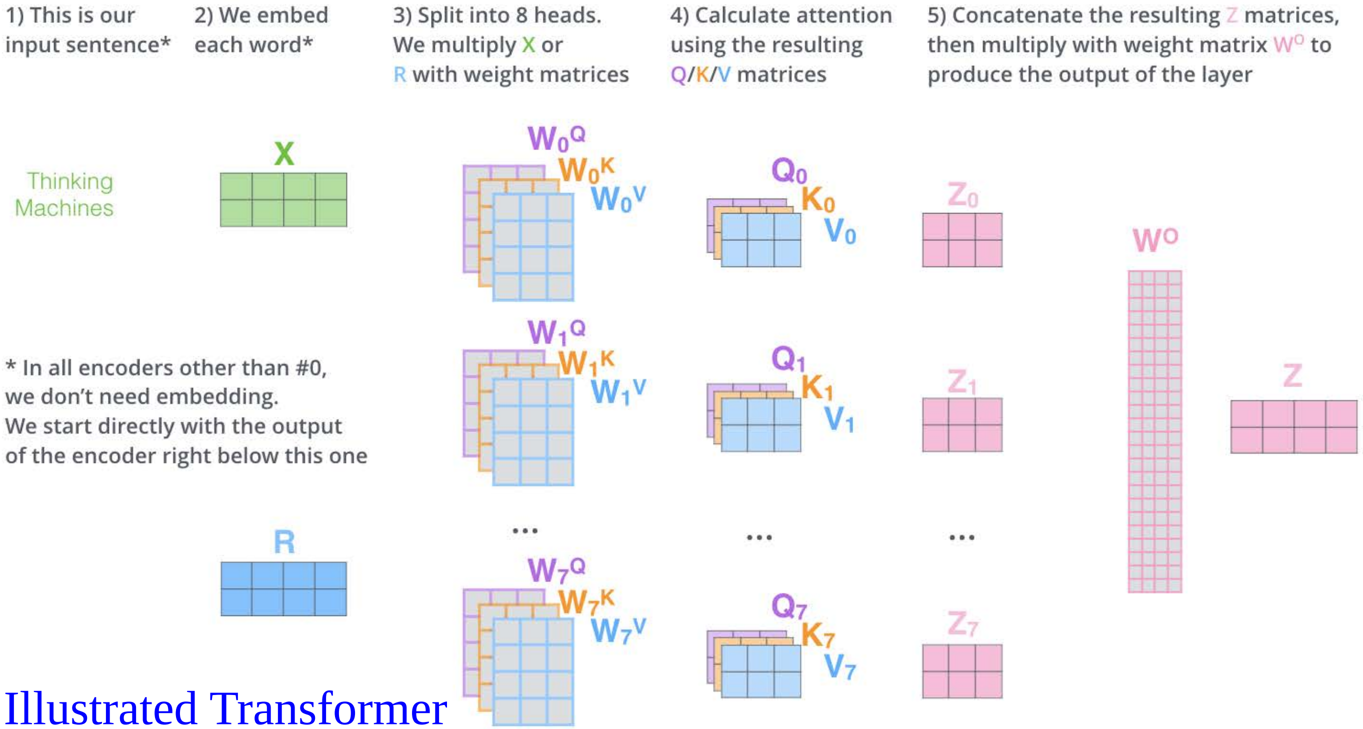
towardsdatascience.com

Transformers

- General idea: learn "importance" of each input for each other input
  - Enables long-range communication between inputs
- Specific implementation: attention mechanism with query, key, value
  - Apply query to keys, then compare to values



[Thinking like Transformer]


1) This is our input sentence*

2) We embed each word*

3) Split into 8 heads. We multiply X or R with weight matrices

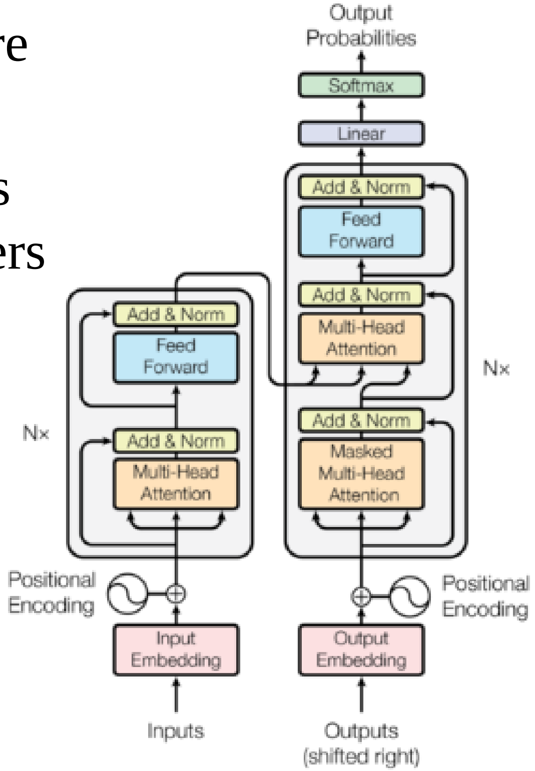
4) Calculate attention using the resulting Q/K/V matrices

5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one

[The Illustrated Transformer]

- In practice: Q, K, V are all *learnable matrices*
- Transformer combines multiple attention layers to encode inputs and then decode outputs
  - "Embedding": represent token as numerical value



[The Annotated Transformer]

Multiple Loss Terms

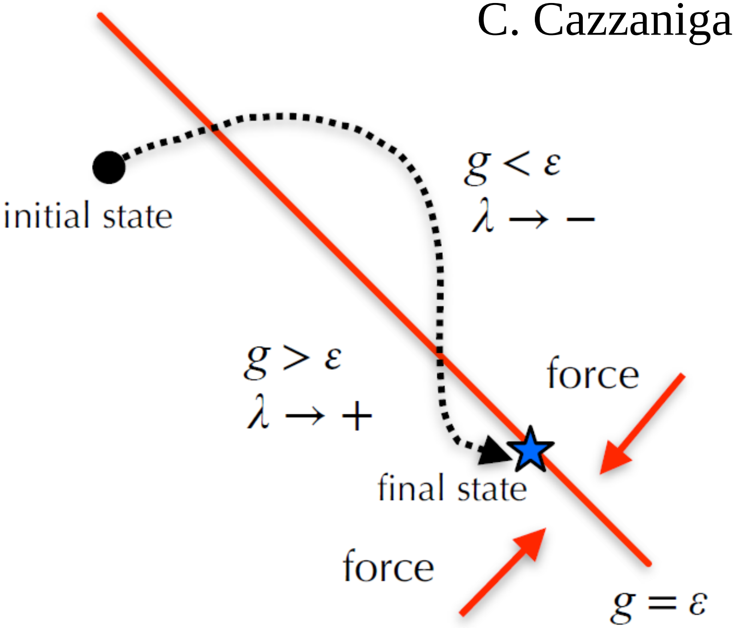
- Simplest approach: L = $f(\theta) + \lambda g(\theta)$
  - λ (relative weight) treated as a hyperparameter:
    guess its value based on magnitudes of f and g, how much you want to control an effect, etc.
  - In generalize, N–1 λ parameters for N loss terms
- Goal: find *Pareto optimal solution* such that any change to improve one criterion will degrade another
- Problems:
  - Pareto front (set of all Pareto optimal solutions) shape is *unknown* (much like gradient space)
  - Unclear relationship between λ values and loss values at Pareto front
- Underlying problem: *no mathematical guarantee* to be able to optimize for two things at once!
- Instead: optimize for one thing with *constraints* on others
  - Lagrange multiplier method, introduced in 1804

Modified Differential Method of Multipliers

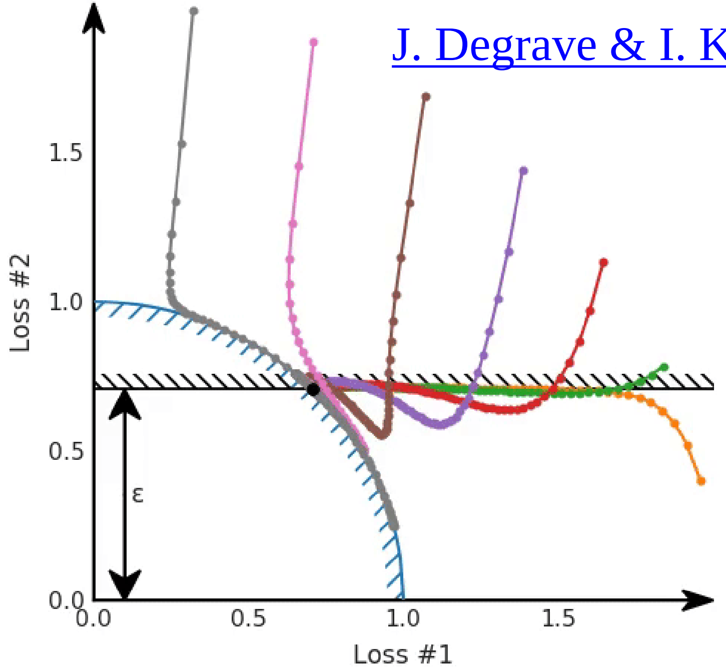
- Lagrange multiplier approach:
  combined loss is $L = f(\theta) + \lambda(\varepsilon - g(\theta)) + \delta(\varepsilon - g(\theta))^2$
  - o ε is the constraint on loss term g
  - o λ is now a *learnable* parameter
  - o δ: new hyperparameter for quadratic damping term
    → influences rate of convergence
- Need to use gradient *ascent* in λ to ensure critical points are attractors rather than saddle points

➢ Ensures convergence even for concave Pareto fronts!
  - o Constraints on loss terms are easy to interpret
  - o Mechanically sketch out Pareto front and pick preferred location → no guessing!
- PyTorch implementation at github:crowsonkb/mdmm

C. Cazzaniga



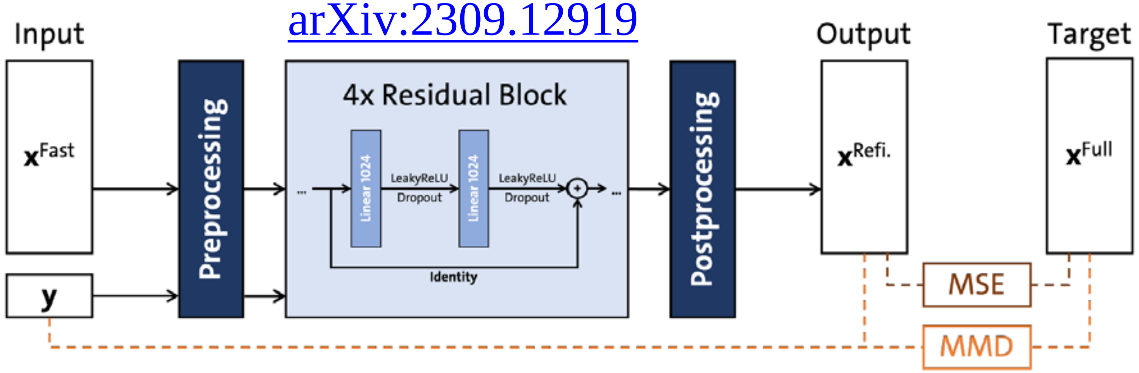
J. Degrave & I. Korshunova

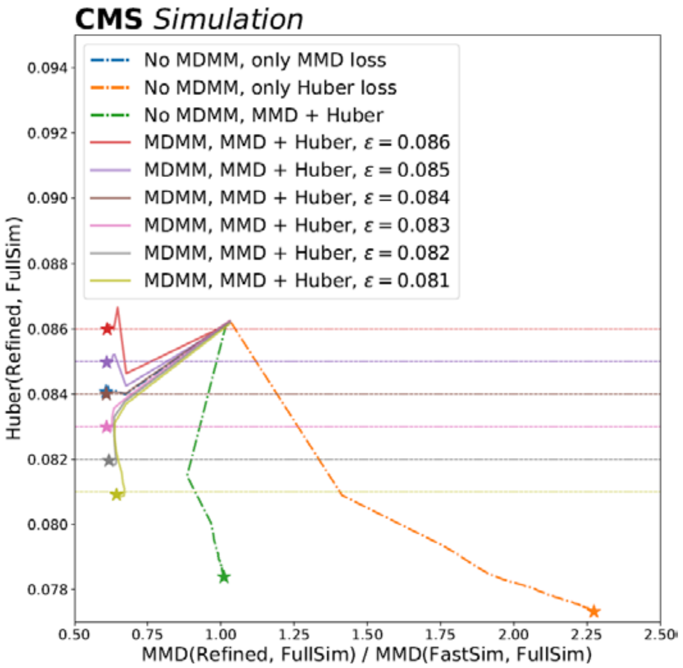


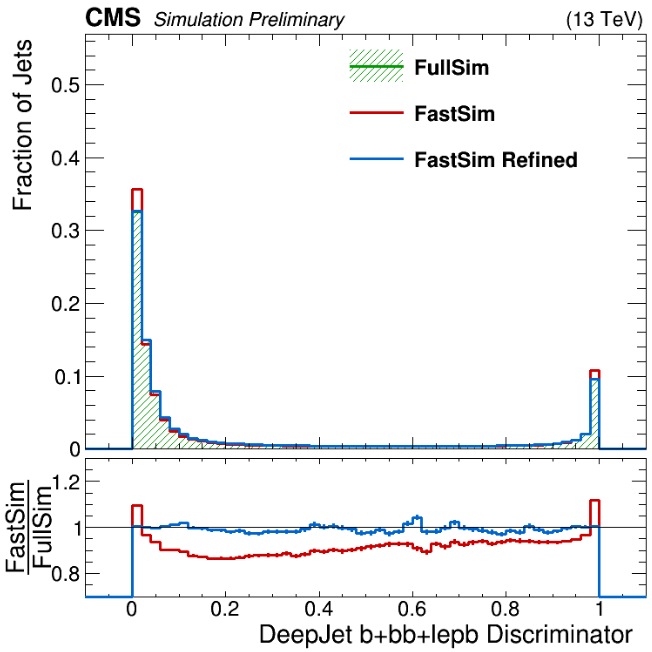
Application to Physics: Fast Simulation

- FastSim refinement: adjust high-level quantities from lower-quality fast simulation to better match high-quality (slow) full simulation
  - o Target: b-jet tagging discriminators
- Two loss terms:
  - o MSE (Huber): per-object comparison
  - o MMD: ensemble comparison
- MDMM balances optimally:
  - o Minimize MSE: bad MMD values
  - o Minimize MMD: still good MSE!
- Substantial improvement in agreement w/ FullSim
- First known usage of MDMM in HEP!

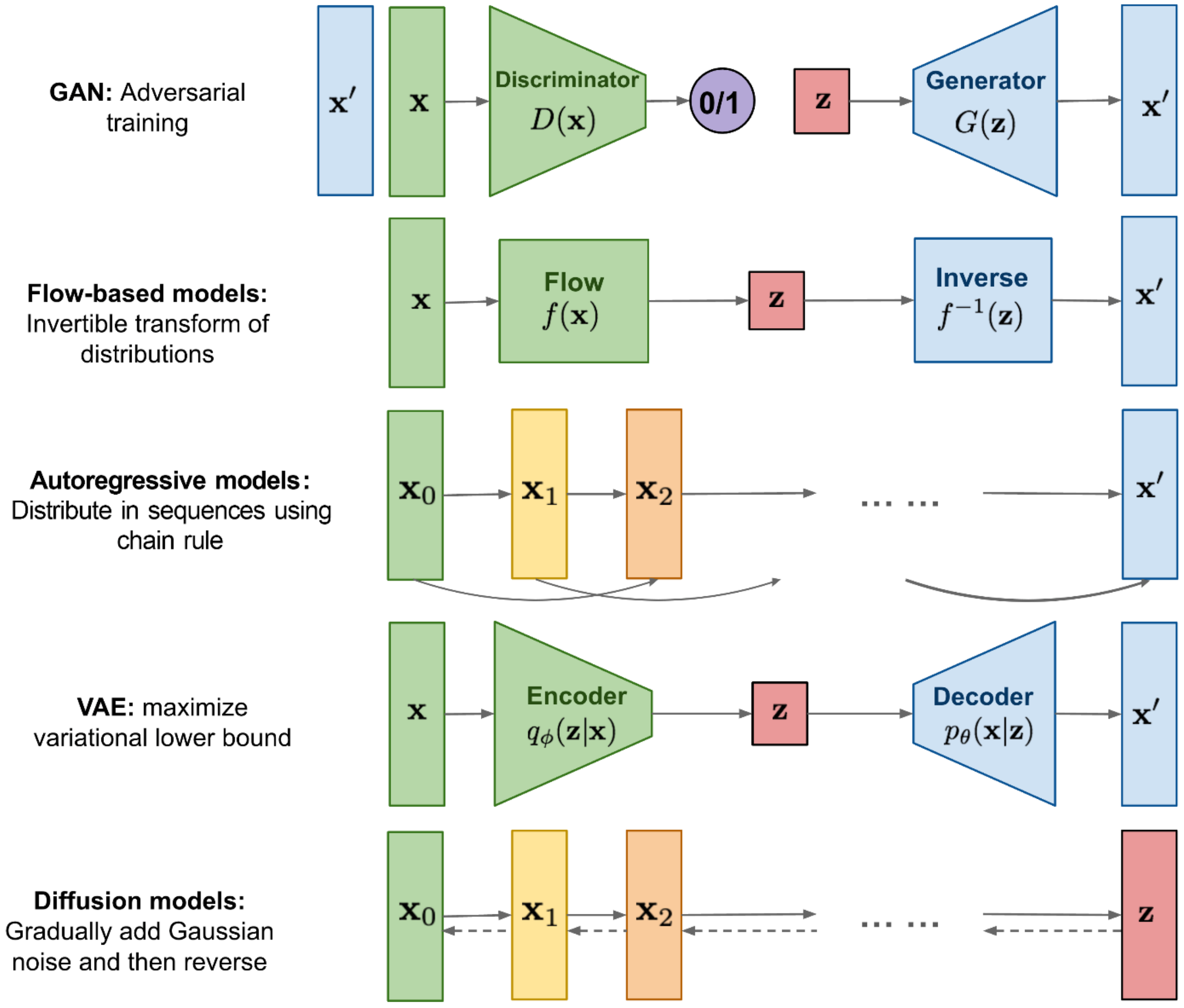
arXiv:2309.12919







Generative Models



L. Weng

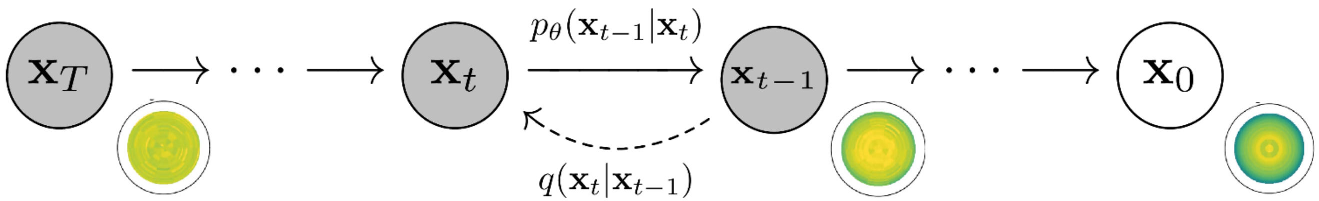
➢ Common idea: learn *probability density* of inputs

- *Implicit* density estimation: Generative Adversarial Networks (GANs)
  - Pros: fast
  - Cons: can suffer from mode collapse, lack of convergence, etc.
- *Exact* density estimation: Normalizing Flows (NFs), Autoregressive models (ARs)
  - Pros: accurate, fast in one direction
  - Cons: poor scaling, slow in other direction
- *Approximate* density estimation: Variational Autoencoders (VAEs), Diffusion Models (DMs)
  - VAEs: fast, but limited quality
  - DMs: high quality, but slow

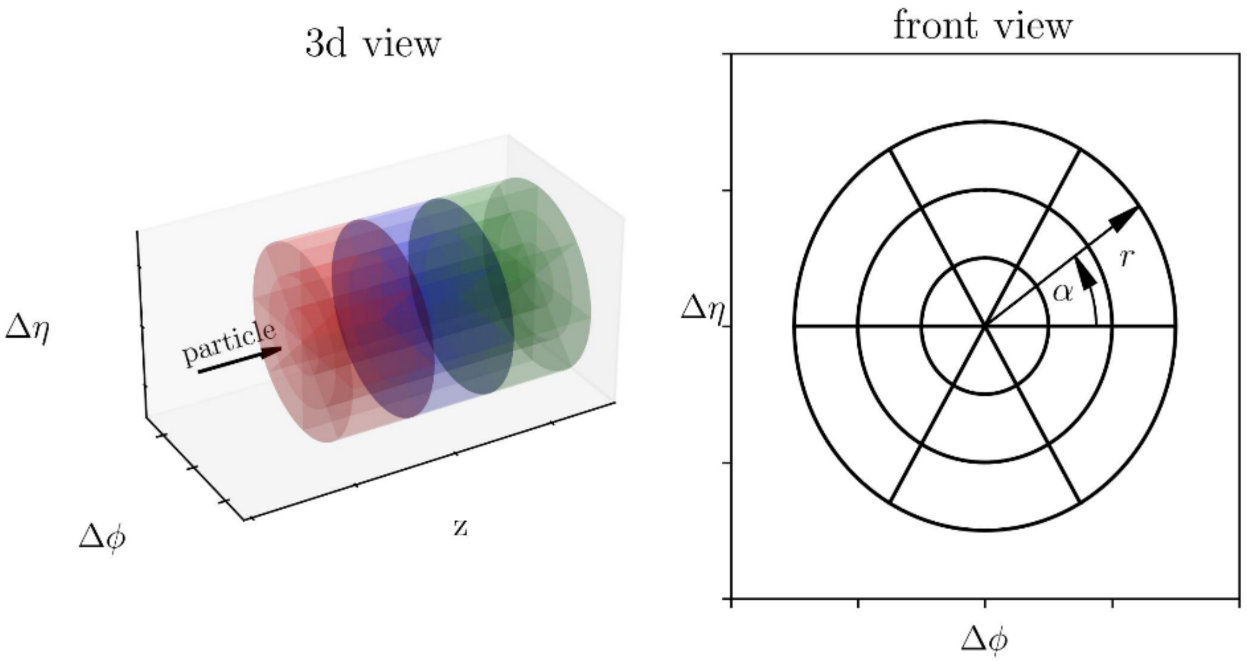
Diffusion Models

- Learn to predict result from “noising process” that iteratively adds Gaussian noise to image
  - o Learn noise prediction function directly, or learn “score function” (gradient of probability density)
    - ▪ Equivalent for variance-preserving score formulation
- Generate output from pure noise by iteratively removing noise using learned function
- Rapidly adopted for image generation in industry
- Let’s apply it to calorimeter showers!
  - o EM physics is compute-intensive
  - o Can also avoid geometry navigation in calorimeter volume

DreamStudio DALL·E 3 Midjourney


$\mathbf{x}_T$
$\mathbf{x}_t$
$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$
$\mathbf{x}_{t-1}$
$q(\mathbf{x}_t|\mathbf{x}_{t-1})$
$\mathbf{x}_0$

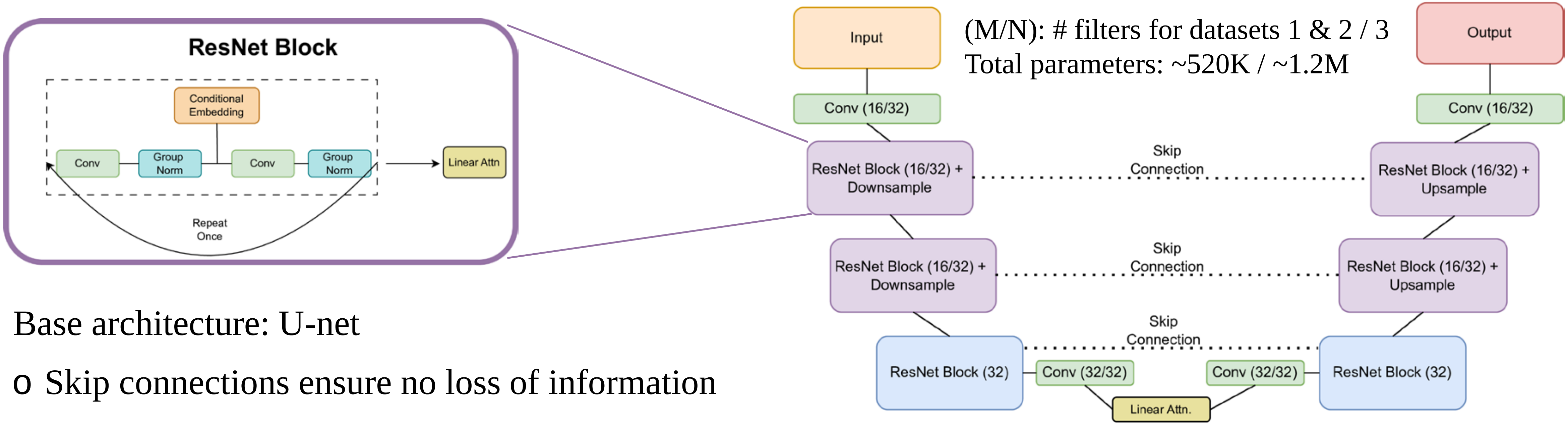
CaloChallenge



- CaloChallenge: first competition for generative ML for detector simulation
- Three public datasets provided:
  1. Low granularity, irregular geometry (based on ATLAS calorimeter), photon & pion showers
  2. Medium granularity, silicon-tungsten sampling calorimeter, electron showers
  3. High granularity, otherwise same as #2

- Common datasets are crucial to compare different generative methods
- Many new methods developed for the challenge

CaloDiffusion

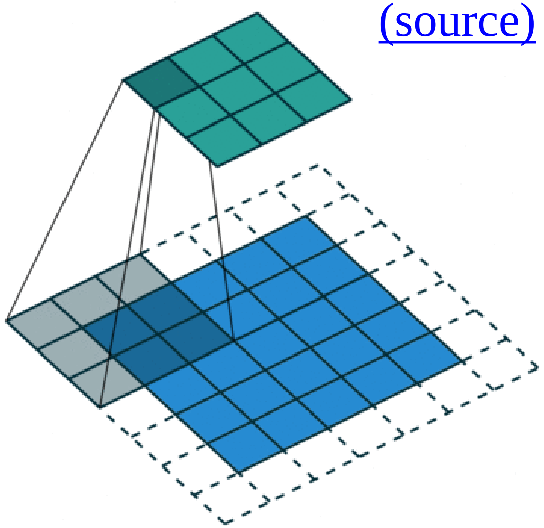


(M/N): # filters for datasets 1 & 2 / 3
Total parameters: ~520K / ~1.2M

- Base architecture: U-net
  - Skip connections ensure no loss of information
- Linear self-attention layers applied to each convolutional ResNet block
  - Allows dimensionality reduction in *z* to handle longitudinal correlations in showers
- Cosine noise schedule for training
- Stochastic sampling algorithm for generation

- Objectives: (regression)
  - Predict (normalized) noise or weighted average of noise and denoised image
- Aim for highest achievable quality first
  - Then focus on improving speed
  - Wrong answers can be obtained infinitely fast

Why Convolutions?



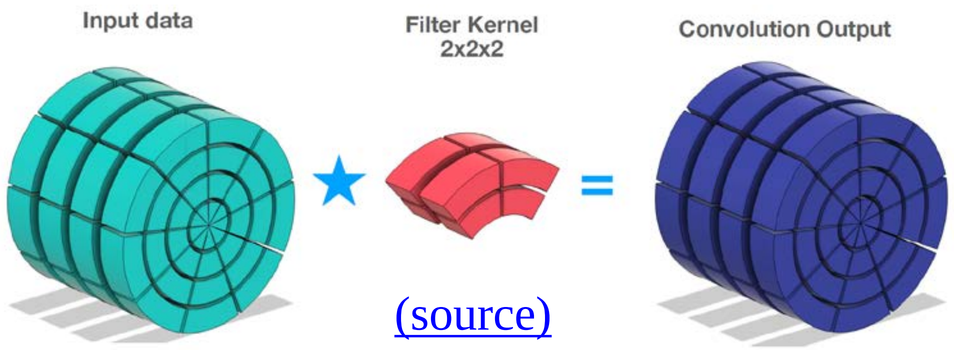
(source)

- Convolutions have many nice properties: (inductive bias)
  - *Spatial locality* and translational invariance
  - Shared weights → fewer parameters, *better scaling*
  - Highly *efficient* on GPUs: spatial locality implies memory locality
- Ideally suited for computer vision with rectangular images
  - Application to irregular geometries requires innovations
- Graph neural networks?
  - **Pro**: natural representation for irregular geometries
  - **Cons**: adjacency matrices consume substantial memory; operations less local/efficient; hard to generate arbitrary output (masking technique exists, but difficult to scale)
- Point clouds or transformers?
  - **Pro**: no adjacency matrix consuming memory
  - **Con**: discards useful geometric information, which then must be learned from (often sparse) inputs

➢ For generative applications, convolutions still have a lot to offer!
  - And they can keep up with transformers when trained properly… e.g. arXiv:2310.16764

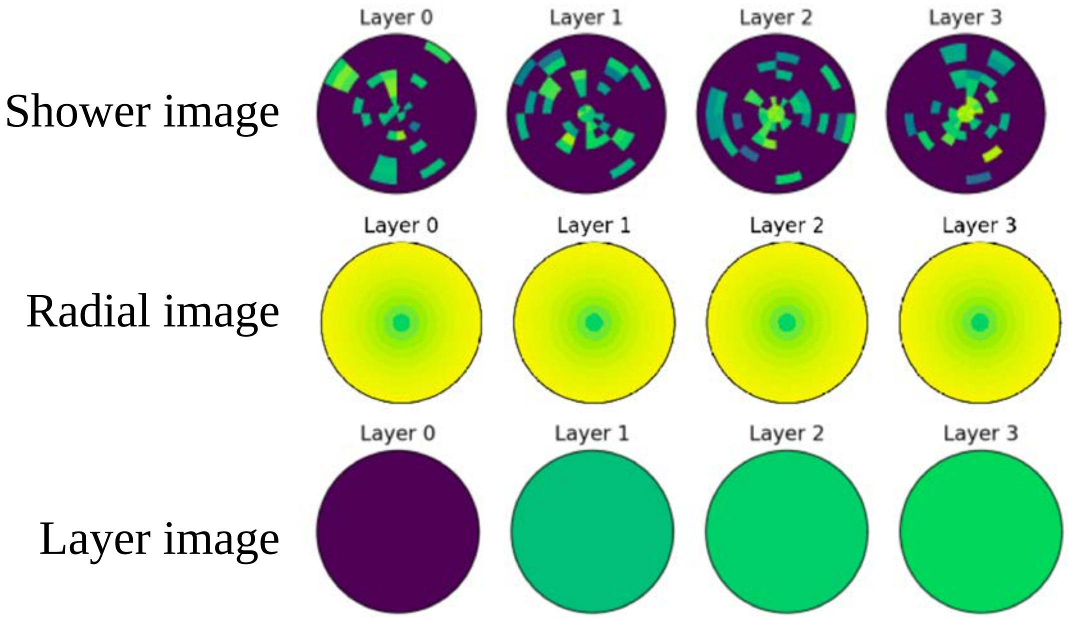
Geometric Innovations

- Particle showers are invariant & periodic in φ
  - Pad in φ so convolutions “wrap around”



(source)

- Particle showers are *not* invariant in *r* or *z*
  - Provide *r* and *z* (layer) as extra per-pixel channels (input features)
  - Convolutions become *conditional*



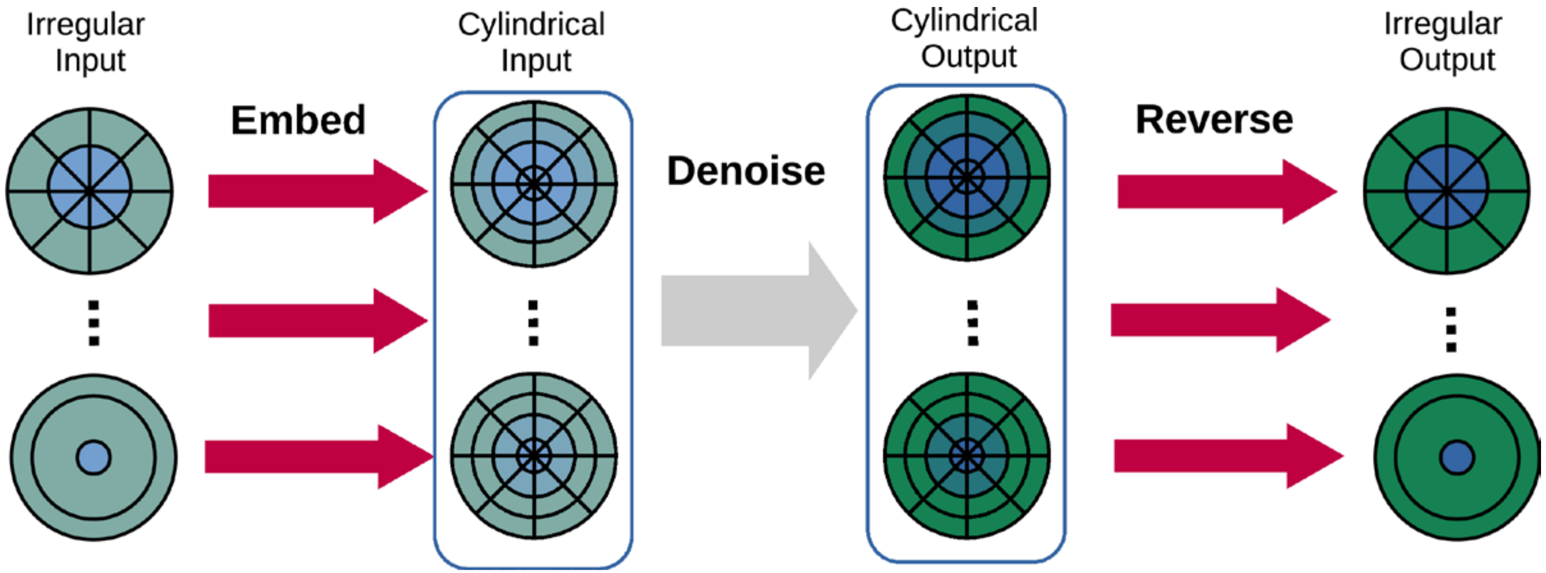
Shower image

Radial image

Layer image

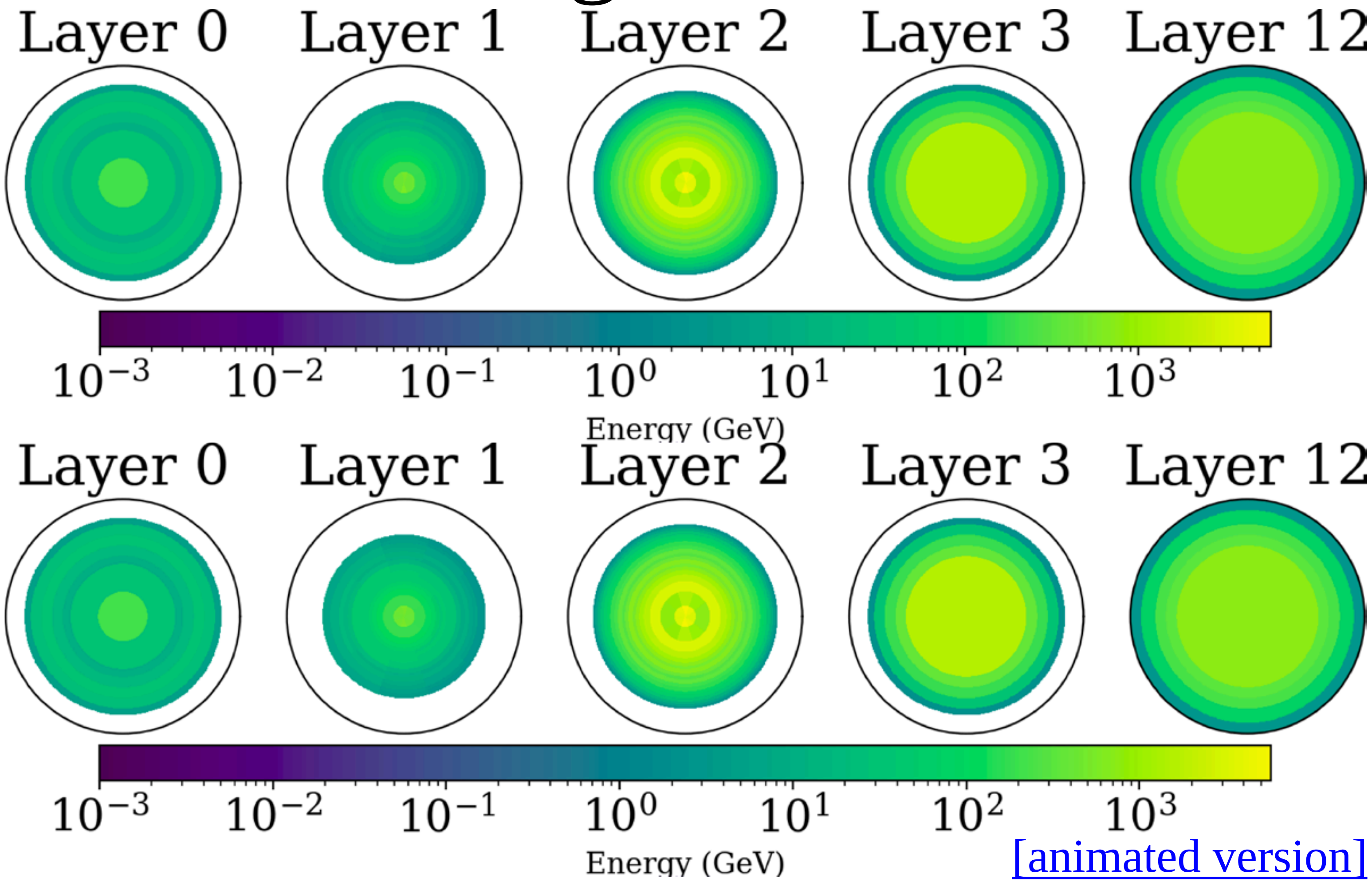
➢ *Conditional cylindrical convolutions*
  - Handle inherent features of particle detector geometry, distinct from rectangular images

Geometry Latent Mapping: GLaM



- Some calorimeters have different radial/angular bins in each layer
  - Can't directly apply convolutions, which require regular neighbor structure
- Learn forward and reverse embeddings to and from a regular geometry
  - Simple matrices C (NxM) and D (MxN)
    - C initialized to split or merge cells based on overlap between original and embedded geometries
    - D initialized as Moore-Penrose pseudoinverse of C
- Inspired by "latent diffusion" approach (apply VAE, then apply diffusion in smaller latent space)
  - But not necessarily lower-dimensional representation; actually higher-dimensional here

Average Showers



[animated version]

- Top: Geant4; bottom: CaloDiffusion (photon showers)
  - … or is it the other way around? Can you tell?

Metrics

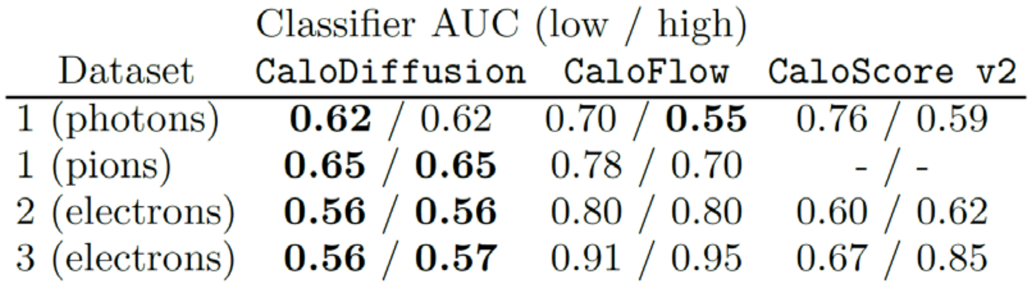
- How to compare quality of generative ML models?
- 1D histograms:
  - e.g. separation power $\langle S^2(g,h)\rangle = \frac{1}{2}\sum \frac{(g-h)^2}{(g+h)}$
  - Can miss high-dimensional correlations
- Best category: **integral probability metrics**

$$D_{\mathcal{F}}(p_{\mathrm{real}}, p_{\mathrm{gen}}) = \sup_{f\in\mathcal{F}} \left|\mathbb{E}_{\mathbf{x}\sim p_{\mathrm{real}}} f(\mathbf{x}) - \mathbb{E}_{\mathbf{y}\sim p_{\mathrm{gen}}} f(\mathbf{y})\right|$$

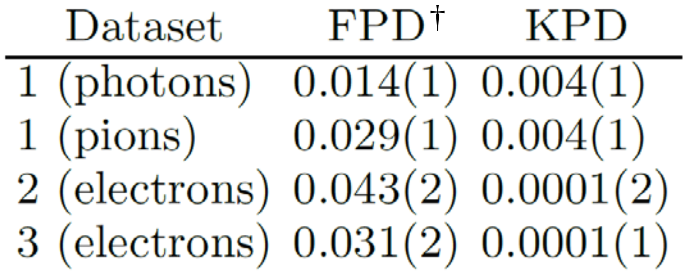
  - *Wasserstein distance* W_1: $\mathcal{F}$ is set of all K-Lipschitz functions
    - Only works well in 1D, biased in high-D
  - *Maximum mean discrepancy* (MMD): $\mathcal{F}$ is unit ball in reproducing kernel Hilbert space
    - Depends on choice of kernel
  - *Fréchet distance*: W_2 distance between Gaussian fits to (high-D) feature space
    - Features can be hand-engineered or obtained from NN activations
- Another interesting category: *classifier scores*
  - Train NN to distinguish real vs. generated
  - AUC score: range 0.5–1.0
  - Log-posterior probability in multiclass case
- *Fréchet Particle Distance* most clearly distinguishes between two similar approaches
  - see arXiv:2211.10295 for more details

Metrics for CaloDiffusion

- Classifier AUC: train a binary classifier to distinguish between Geant4 and generative model
  - 2 hidden layers, 2048 neurons each; 20% dropout after each layer
  - Two flavors w/ different inputs: (incident particle energy included in both)
    - Low-level: full showers (all voxels)
    - High-level: energy in each layer, center of energy and shower width in η and φ
  - Compared to CaloScore v2 (score-based diffusion model), (i)CaloFlow (normalizing flow)
- Integral probability metrics: Fréchet Particle Distance (FPD), Kernel Particle Distance (KPD)
  - High-level shower features used as input



	Classifier AUC (low / high)		
Dataset	`CaloDiffusion`	`CaloFlow`	`CaloScore v2`
1 (photons)	**0.62** / 0.62	0.70 / **0.55**	0.76 / 0.59
1 (pions)	**0.65** / **0.65**	0.78 / 0.70	- / -
2 (electrons)	**0.56** / **0.56**	0.80 / 0.80	0.60 / 0.62
3 (electrons)	**0.56** / **0.57**	0.91 / 0.95	0.67 / 0.85

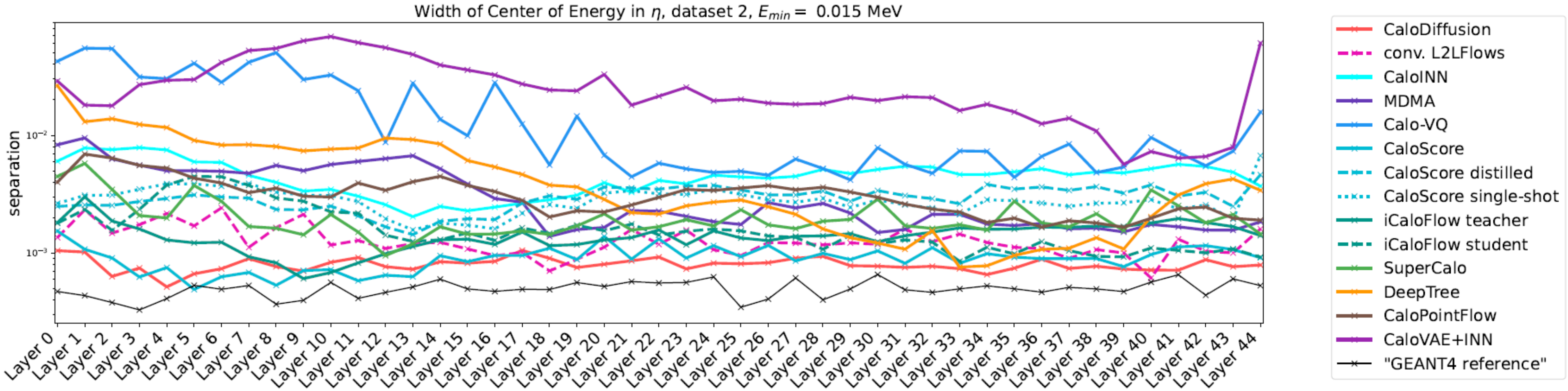


Dataset	FPD†	KPD
1 (photons)	0.014(1)	0.004(1)
1 (pions)	0.029(1)	0.004(1)
2 (electrons)	0.043(2)	0.0001(2)
3 (electrons)	0.031(2)	0.0001(1)

- CaloDiffusion wins in almost all comparisons, with very small distance values
  - Generated showers almost indistinguishable from Geant4
  - Further comparisons to come in CaloChallenge summary

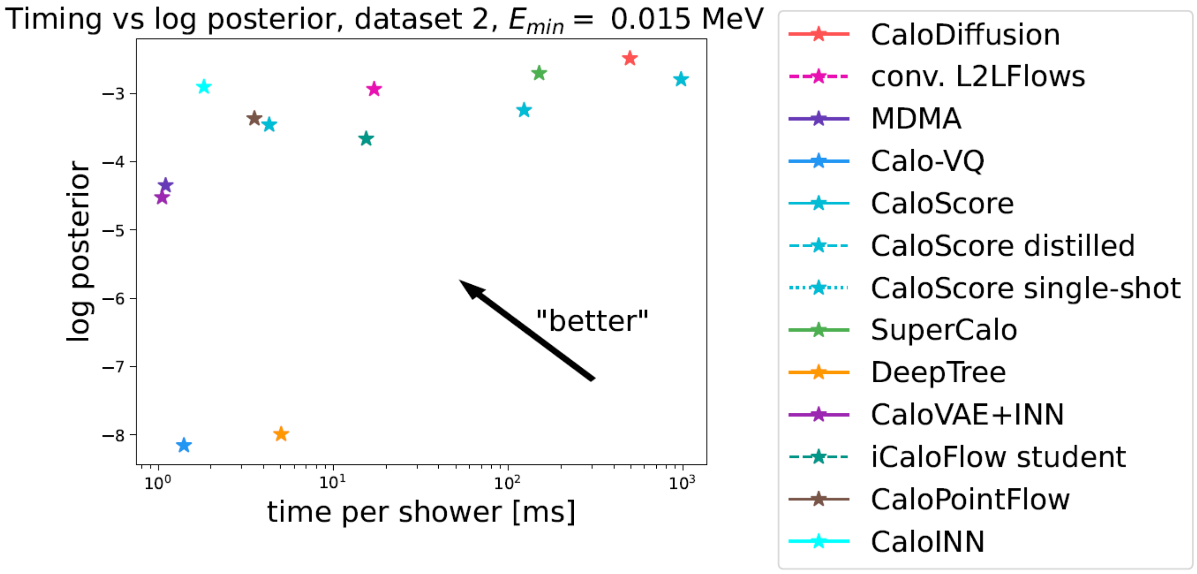
† Geant4 self-comparison values subtracted (0.008, 0.0005, 0.008, 0.011)

CaloChallenge Results

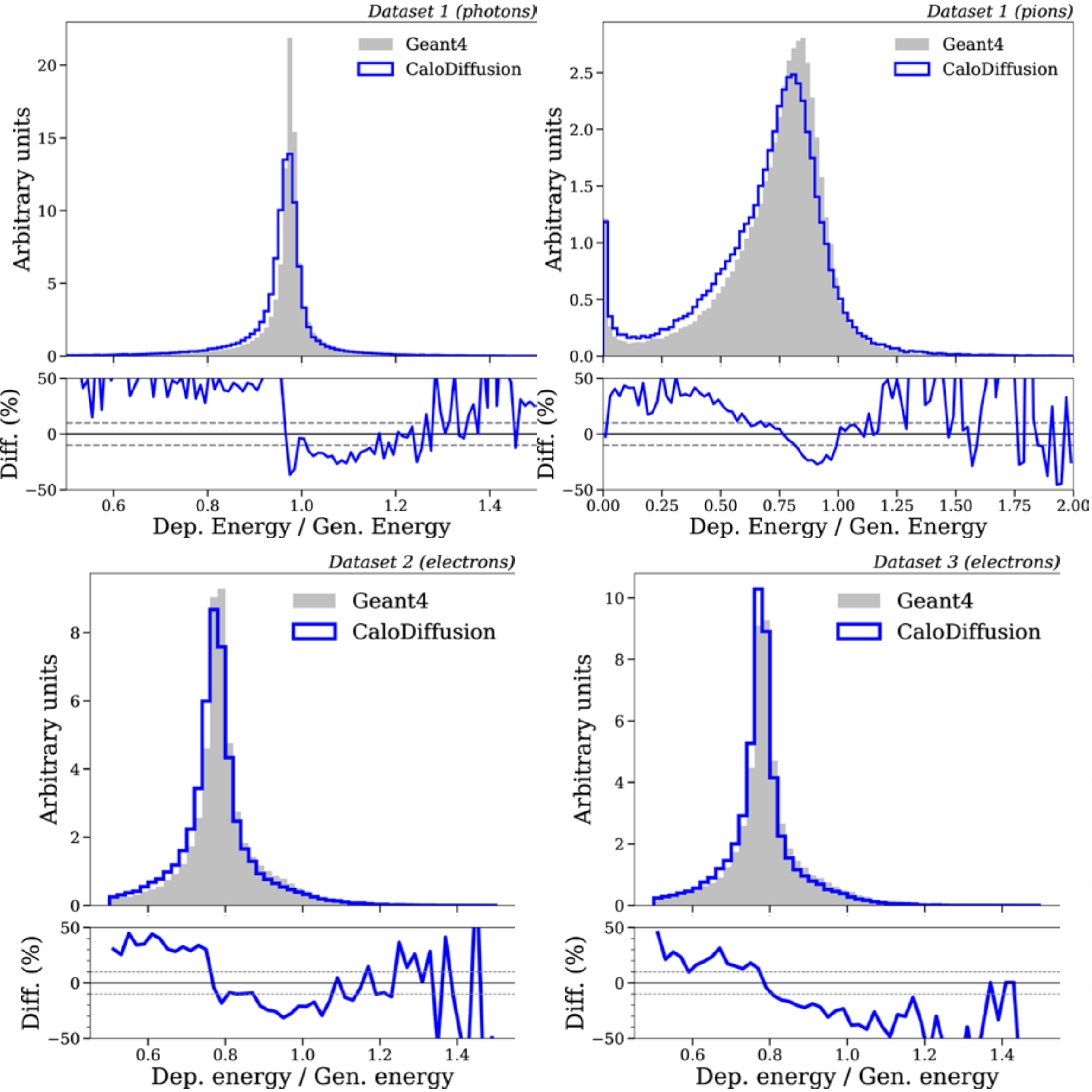


C. Krause

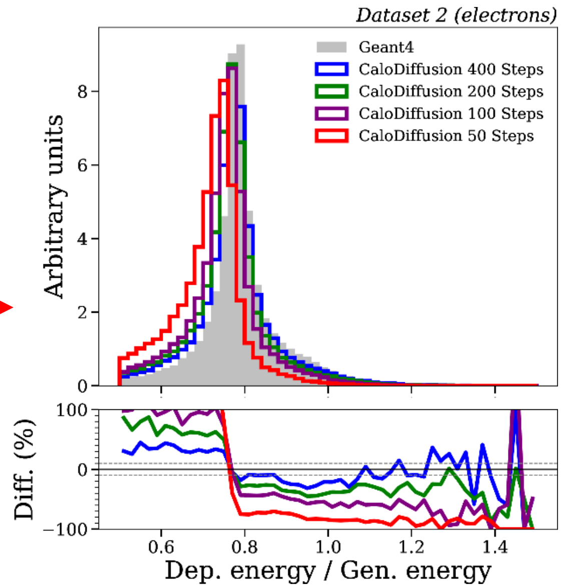
- Diffusion models and normalizing flows tend to have best performance
- However, diffusion models especially tend to be slower in inference
  - o Iterative process – multiple steps required to get highest accuracy
- Benefit of following industry trends: frequent papers with new methods to speed up diffusion models → easy to adopt in HEP



CaloDiffusion: Areas for Improvement



- Deficit in total energy modeling
- Need 400 diffusion steps to get acceptable quality
  - o Still faster than Geant4 (~100s) w/ batching on GPU
- Fewer steps:
  - o Linear speed improvement
  - o But even less accurate in this quantity →



Dataset	Batch Size	Time/Shower [s] CPU	GPU
1 (photons)	1	9.4	6.3
(368 voxels)	10	2.0	0.6
	100	1.0	0.1
1 (pions)	1	9.8	6.4
(533 voxels)	10	2.0	0.6
	100	1.0	0.1
2 (electrons)	1	14.8	6.2
(6.5K voxels)	10	4.6	0.6
	100	4.0	0.2
3 (electrons)	1	52.7	7.1
(40.5K voxels)	10	44.1	2.6
	100	-	2.0

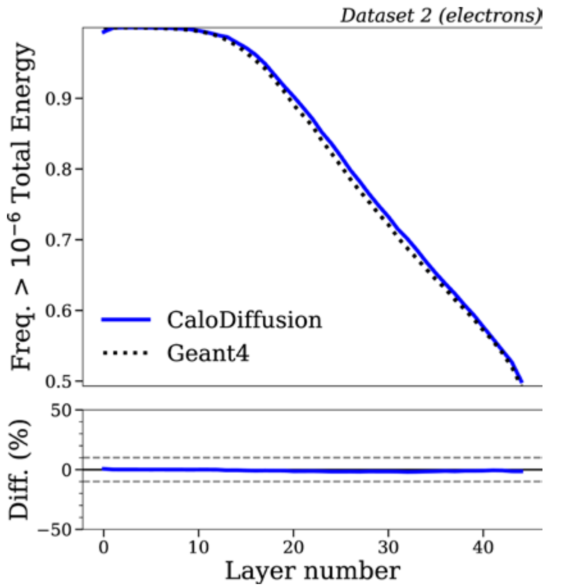
Num. Steps	Classifier AUC (low / high)	FPD	E Ratio Sep. Power
400	0.56 / 0.55	0.043(1)	0.011
200	0.61 / 0.56	0.046(1)	0.036
100	0.69 / 0.59	0.065(3)	0.079
50	0.83 / 0.67	0.110(4)	0.251

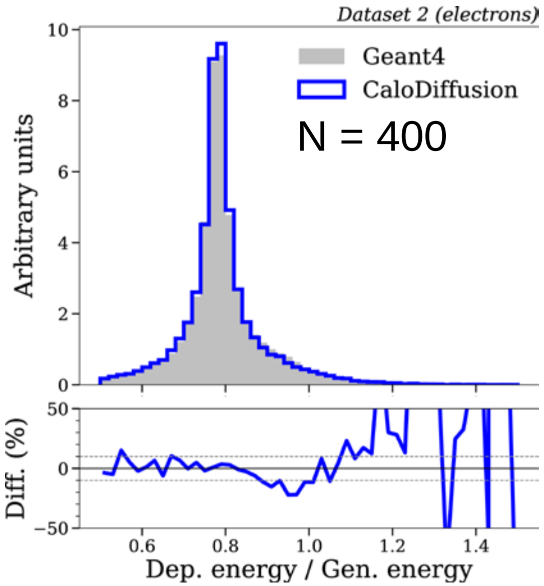
Improvement: More Diffusion!

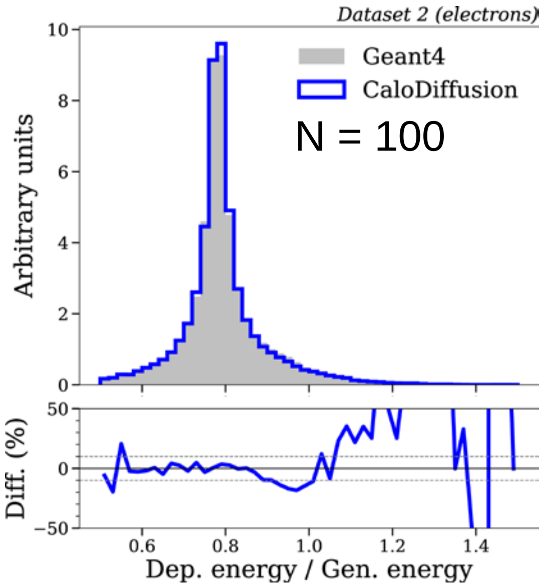
- Train LayerDiffusion to predict energy deposited per layer (1D diffusion)
  - Negligible inference time (200 steps) compared to CaloDiffusion
- Normalize CaloDiffusion output based on LayerDiffusion
  - Only if both models predict sufficiently non-zero deposited energy in a layer

➢ Substantial improvement in total energy modeling

- Number of CaloDiffusion steps can be reduced with no loss of quality
  - 4× speedup for Dataset 2! (8× for Dataset 1 & improves low-energy pions)





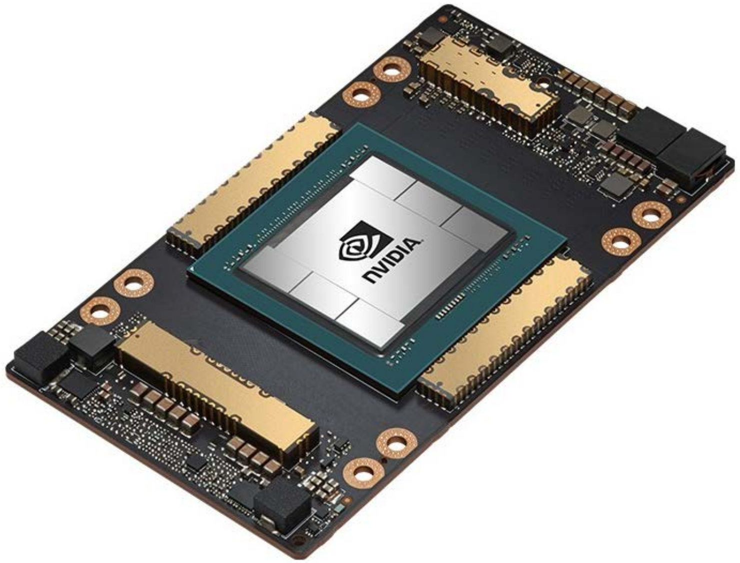


Model (2, electrons)	**AUC (low / high)**	**FPD**	**KPD**	**E Ratio Sep. Power**
Orig. (N = 400)	0.56 / 0.56	0.043	0.0001	0.011
Layer (N = 400)	0.54 / 0.58	0.045	0.00005	0.0017
Layer (N = 100)	0.54 / 0.60	0.076	0.0003	0.0017

- More speedups proposed in [arXiv:2401.13162]()

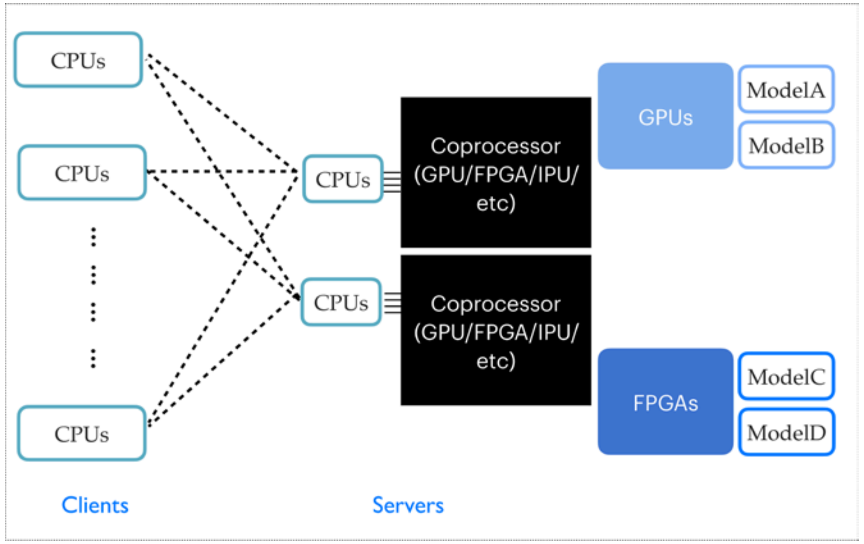
Computing for ML

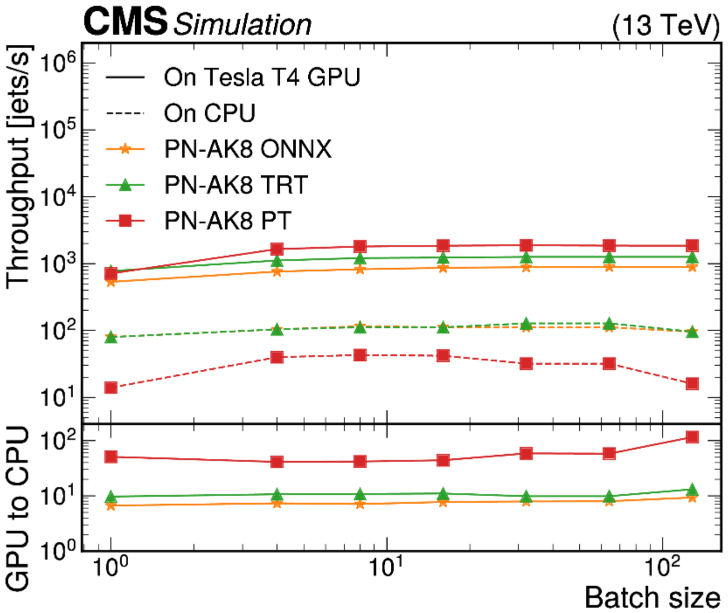
- ML algorithms use a restricted set of operations (mostly matrix multiplications)
  - Natural and easy to accelerate on “coprocessors” like GPUs (SIMD: single instruction, multiple data)
- Advent of GPU computing helped spur modern AI revolution
  - Otherwise not feasible to perform backpropagation in deep NNs
- NN training is compute-intensive
  - A100 GPUs deliver ~300 teraflops (TF32 tensor operations) with up to 80 GB of RAM
  - Often training needs multiple A100s!
- Nevertheless, inference ultimately requires more compute
  - Goal is to learn a generalized algorithm/function
  - Therefore, trained NN will be applied to much more data than was used in training
    - ➢ Billions of events, at least



Inference as a Service

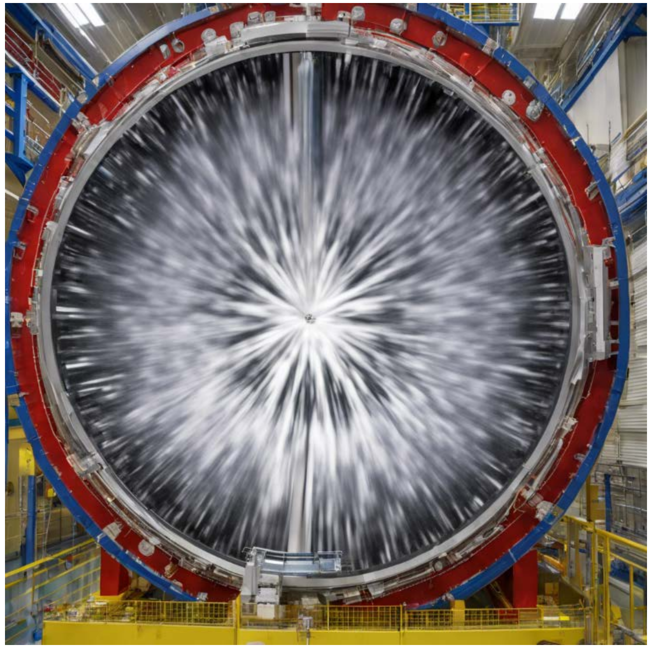
- *Most flexible* approach to use coprocessors
  - Abstract away specific computing elements: client makes request, server delivers
  - Example: ParticleNet 10–100× faster on GPU vs. CPU
    - Algorithm latency becomes essentially *invisible* with asynchronous calls in offline processing
    - Can batch *across events* for optimal GPU utilization → maximize throughput
    - Similar speedup for CaloDiffusion
- Demonstrated for CMS, protoDUNE, LIGO, analysis facilities
  - Use any kind of chip with zero code changes!
    - Including new "neuromorphic" chips: tensor processing units (TPUs), intelligence processing units (IPUs), etc.
  - Exploit GPU-based High Performance Computing (HPC) facilities





Conclusions

- AI/ML has applications throughout HEP
  - Complicated, but understandable
  - Remembering basic principles will help you debug unexpected behavior
  - A Recipe for Training Neural Networks (Karpathy) is a useful guide
- Many of these applications were not discussed at all today!
  - Clustering/tracking
  - Unsupervised learning: anomaly detection
  - Even classification given short shrift
  - Check out the HEPML LivingReview to learn more about these
- Generative ML is an especially promising application
  - Eventually produces a differentiable simulation
    → can then be part of broader optimization
- The future of AI/ML is wide open
  - All of this may be outdated in just a few years!



Generated by SDXL 1.0 w/ prompt: "A GEANT4 simulation of a pion shower with energy 100 GeV in the Compact Muon Solenoid High Granularity Calorimeter at the CERN Large Hadron Collider, a particle physics experiment"