University of Massachusetts Amherst

Machine learning: simulation and data

Rafael Coelho Lopes de Sa

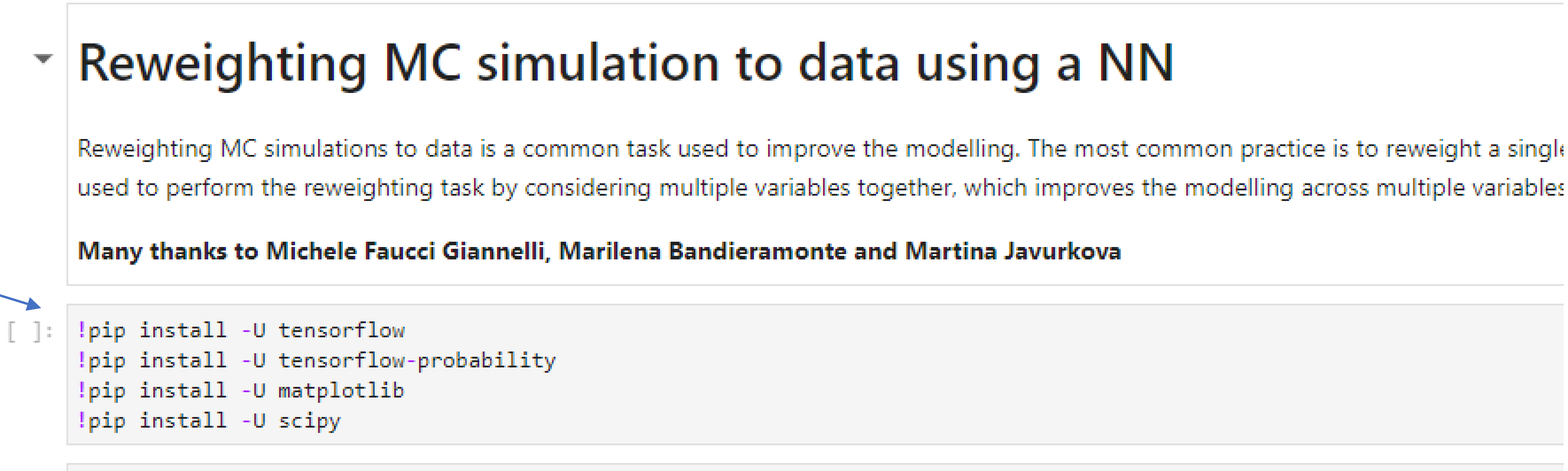
HSF India Training event - Bhubaneswar

December 21st, 2023

Before we start

- Go to my binder: https://binderhub.ssl-hep.org/v2/gh/rafaellopesdesa/hsfindia-generative/HEADgpu_false

- Go to the reweighting exercise and run the pip command.
- Let it run while I talk about what we will do…



Reweighting MC simulation to data using a NN

Reweighting MC simulations to data is a common task used to improve the modelling. The most common practice is to reweight a singl
used to perform the reweighting task by considering multiple variables together, which improves the modelling across multiple variable

Many thanks to Michele Faucci Giannelli, Marilena Bandieramonte and Martina Javurkova

```
[ ]: !pip install -U tensorflow
     !pip install -U tensorflow-probability
     !pip install -U matplotlib
     !pip install -U scipy
```

Run this cell and let's talk about physics while it installs

Analysis and simulation

The goal of LHC analysis is to compare the data to a probability model for different hypothesis, usually Standard Model (SM) vs New Physics (NP)

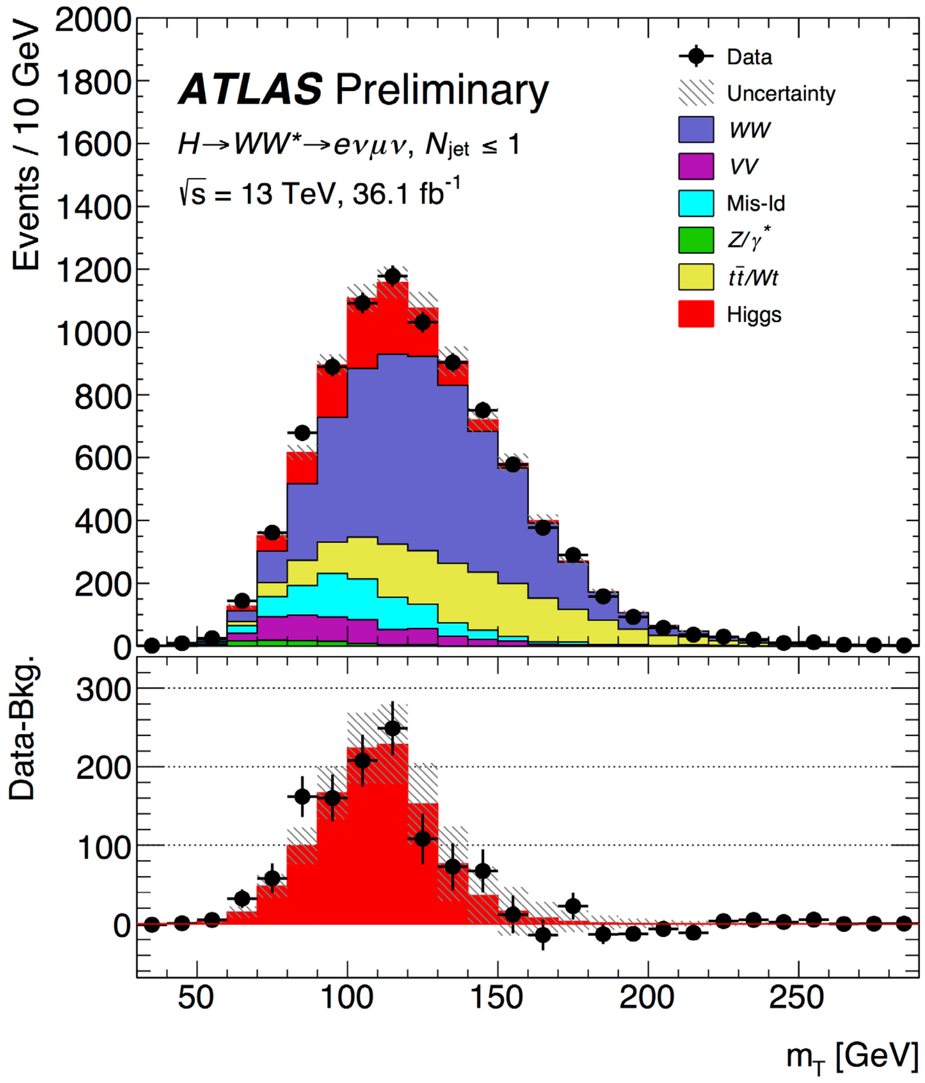
$$p\left(x^{\mathrm{data}}\middle|H_1\right) \text{ and } p\left(x^{\mathrm{data}}\middle|H_0\right)$$

Building the $p(x|H)$ models is **really complicated**.

In most cases $p(x|H)$ are approximated by histograms where the number of events (I will call it ν) in each bin comes from simulation

→ Counting MC (simulation) events.

How do we count MC events?



In practice, MC events have weights

$$p(x|H_0) = \frac{\nu_{\mathrm{I}}^{\mathrm{bkg}}}{\nu^{\mathrm{bkg}}} = \frac{\sum_{\mathrm{bkg}}^{\mathrm{bin}\ I} w_i}{\sum_{\mathrm{bkg}} w_i}$$

$$p(x|H_1) = \frac{\nu_{\mathrm{I}}^{\mathrm{bkg}} + \nu_{\mathrm{I}}^{\mathrm{sig}}}{\nu^{\mathrm{bkg}} + \nu^{\mathrm{sig}}} = \frac{\sum_{\mathrm{bkg}}^{\mathrm{bin}\ I} w_i + \sum_{\mathrm{sig}}^{\mathrm{bin}\ I} w_i}{\sum_{\mathrm{bkg}} w_i + \sum_{\mathrm{sig}} w_i}$$

But what is w_i again? In simulation, the probability of a given event is given by the differential cross section

$$\frac{d\sigma}{dz} \simeq w_i$$

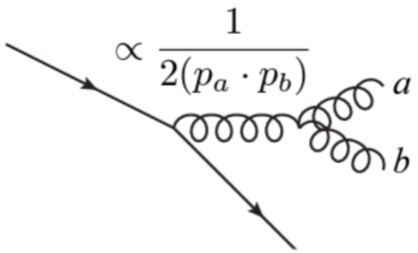
Sometimes, we can “unweight” events, but not always…

Negative weights

Cross sections are all positive. So why do we have negative weights?

Perturbation theory, **parton shower**, and **interference**

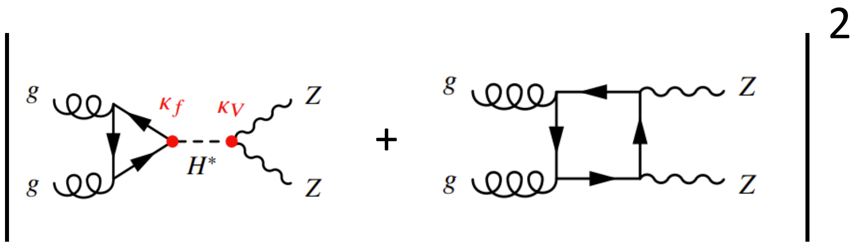
Higher order terms have divergencies



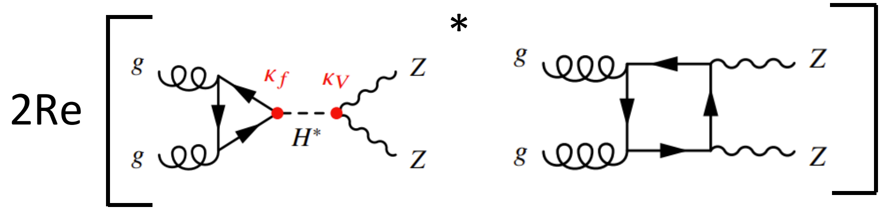
A common way to remove these divergences is to introduce negative weights.

In addition, the same kind of emission can be introduced by parton showers. Negative weights are a common way to remove double counting.

Processes with interference can be funny…



This is positive, but the interference term can be positive or negative



So, what’s the idea?

Sample a phase space point just constrained by conservation of energy-momentum

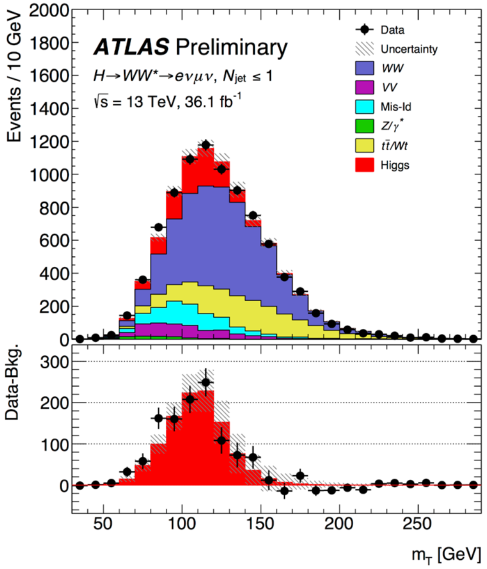
z_i

Generator

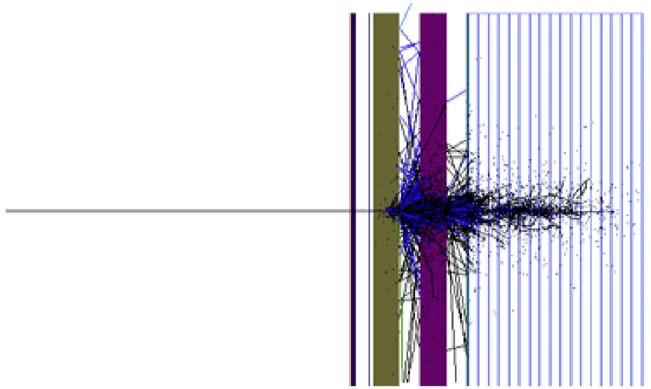
$z_i, \mathrm{w_i}$

Simulator

$x_i, \mathrm{w_i}$



Simulation



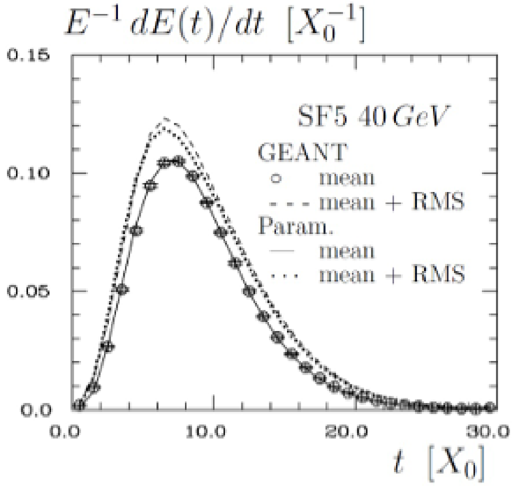
Full simulation

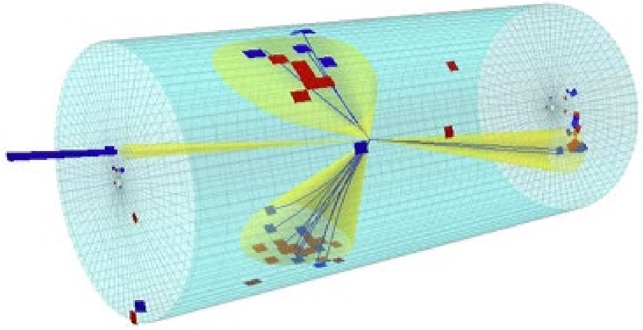
- Common software framework (usually Geant4, but others exist)
- Experiments provide additional code (digitization, reconstruction, …)
- Explicit modeling of detector geometry, materials, interactions w/ particles

  Add thousands of additional variables. Sequential sampling $z_i \sim p(z_i|z_{j<i})$ and $x \sim p(x|z_i)$

Fast simulation

- Usually experiment-specific framework
- Explicit modeling of detector geometry
- Add approximations: analytical shower shapes, truth-associated track reconstruction, …





Parametrized simulation

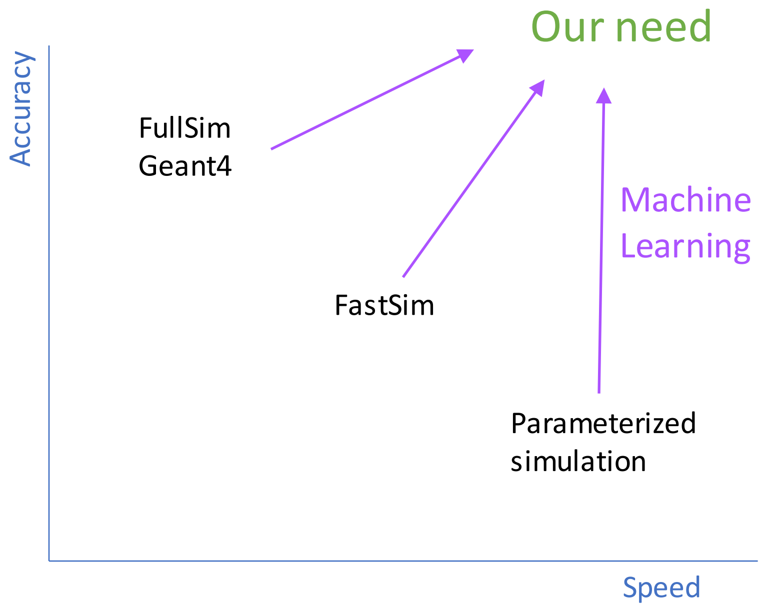
- Does not describe the detector
- Replaces entire chain (“end-to-end”)
- Can be done with analytical function or machine learning methods

$$x \sim p(x|z)$$

Simulation landscape

How can we use ML with simulation?

- Augment the full simulation
    - Improve the MC weights with data
    - Calibrate your simulation
- Replace (part of) full simulation
- Create an “end-to-end” parametrized simulation



Goals:

1. Increase speed while preserving accuracy
2. Preserve speed while increasing accuracy

Augment full simulation

- Usually *deterministic.*

Classification based

Uses a classification loss, like the one you tried in Gordon's lectures

$$L = -\frac{1}{N}\sum_i w_i[y_i \log s_i + (1-y_i)\log(1-s_i)]$$

The minimum of this loss function is achieved at:

$$s(x) = \frac{p_1(x)v_1}{p_0(x)v_0 + p_1(x)v_1}$$

If $v_0 = v_1$ (balanced) $\frac{p_1(x)}{p_0(x)} = \frac{s}{1-s}$

Regression based

Uses a regression loss, for instance MSE (there are others):

$$L = -\frac{1}{N}\sum_i w_i(y_i - s_i)^2$$

Creates a calibration function $s_i(x_i)$

But it only calibrates the average (conditional on x_i), not full distributions.

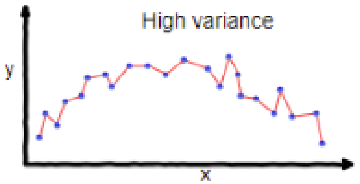
Classification-based reweighting

- Relies on finding a control region (CR).
- Reweight between CR and SR need to be validated carefully.

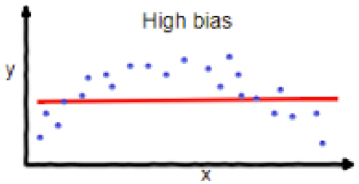
- Try the activity in binder!
  - https://binderhub.ssl-hep.org/v2/gh/rafaellopesdesa/hsfindia-generative/HEADgpu_false

$$SR_{\mathrm{data}} = \frac{CR_{\mathrm{data}}}{CR_{\mathrm{sim}}} \times SR_{\mathrm{sim}}$$

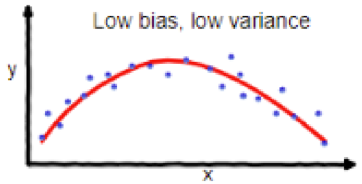
CR_{data}	SR_{data}
CR_{sim}	SR_{sim}



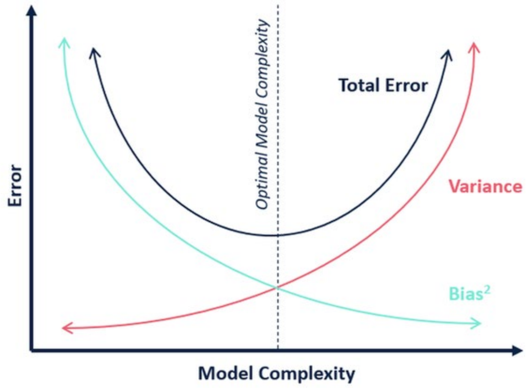
overfitting

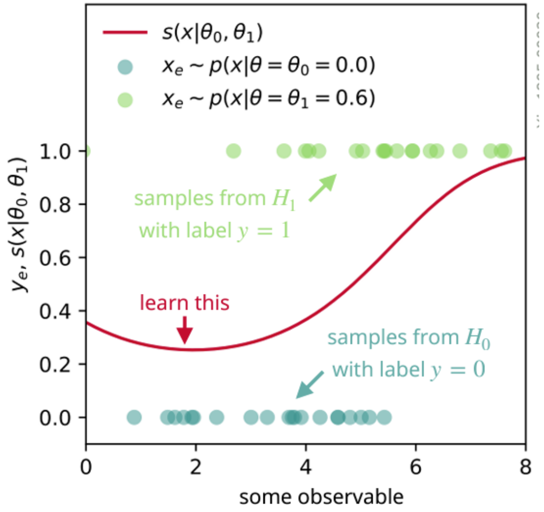


underfitting



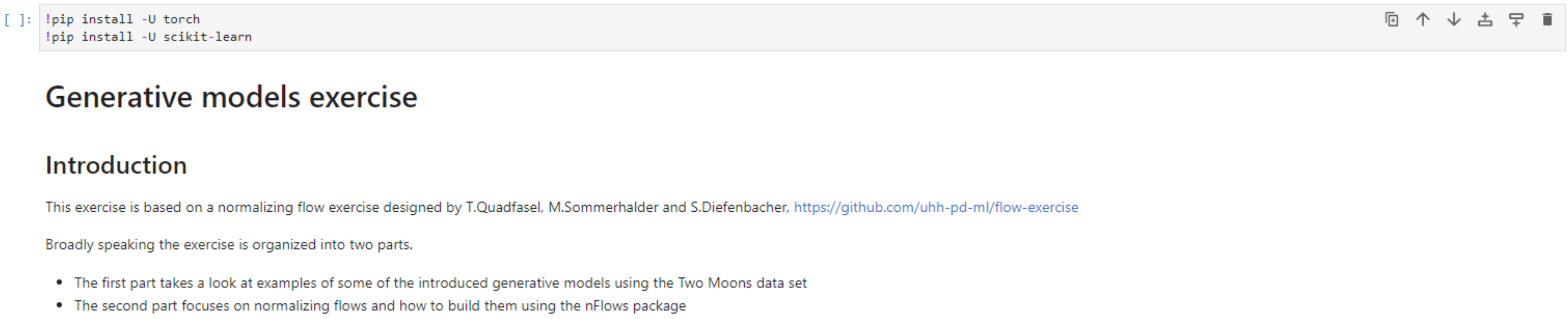
Good balance





Intermezzo

- Now let’s do the second part…
- Go to the generative directory in binder and run the pip cell (it assumes you are using the same session as the reweight one)



```
[ ]: !pip install -U torch
     !pip install -U scikit-learn
```

Generative models exercise

Introduction

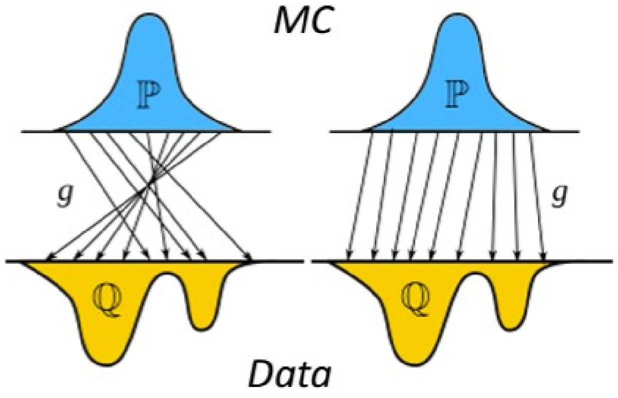
This exercise is based on a normalizing flow exercise designed by T.Quadfasel, M.Sommerhalder and S.Diefenbacher, https://github.com/uhh-pd-ml/flow-exercise

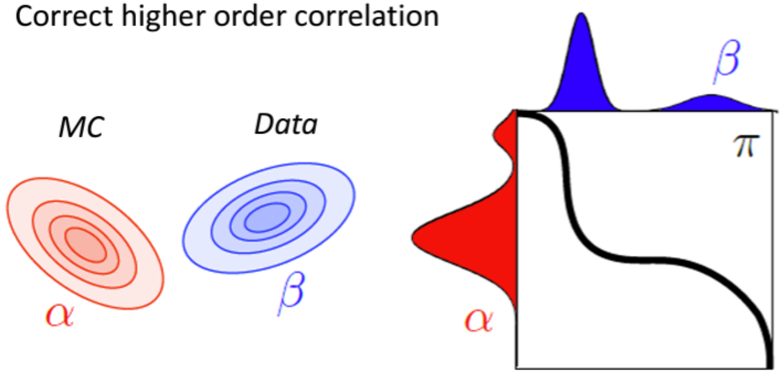
Broadly speaking the exercise is organized into two parts.

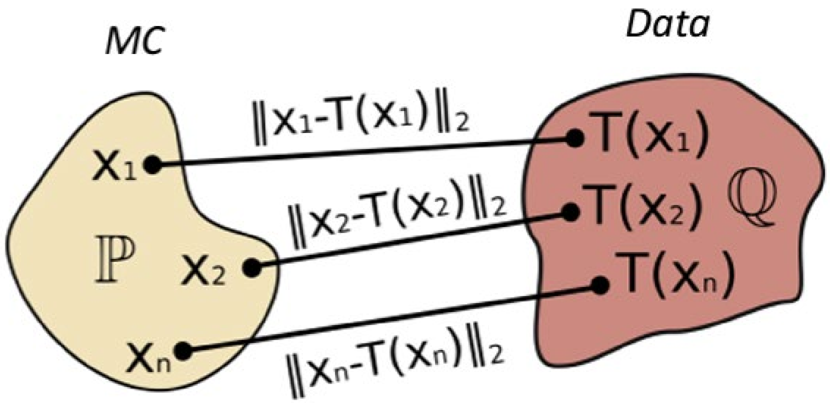
- The first part takes a look at examples of some of the introduced generative models using the Two Moons data set
- The second part focuses on normalizing flows and how to build them using the nFlows package

Optimal transport

- Moving points instead of reweighting histograms
- “Optimal” : Transport minimize some cost (L^2)
    - Order preserving transformation between P and Q
- Easily scalable to higher dimensions
- Correct higher order correlation







1D optimal transport

$p = 2$, i.e. $c(x, y) = |x - y|^2$

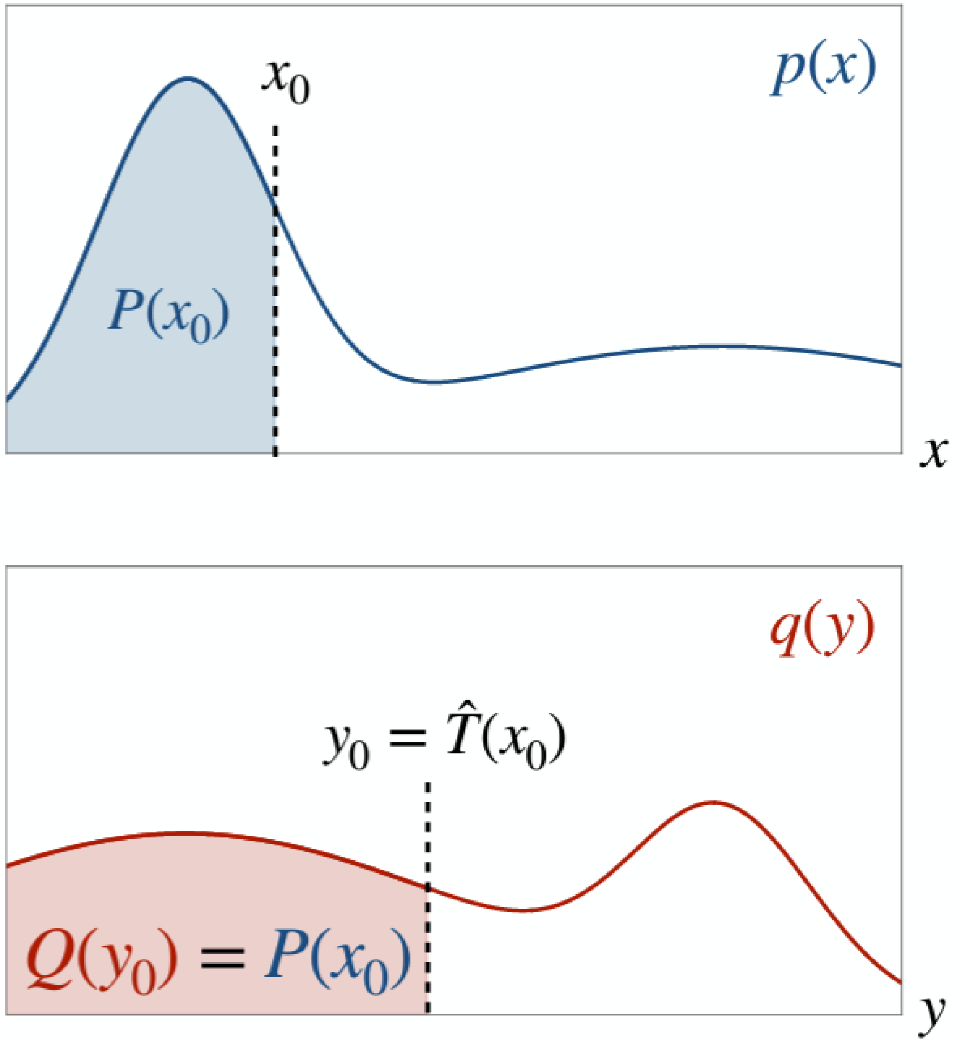
For 1-dimensional distributions:

The optimal transport solution performs quantile-matching *(works for all convex cost functions!)*

$$\hat{T}(x) = Q^{-1}(P(x))$$

Cumulative distributions of $p(x)$, $q(y)$:

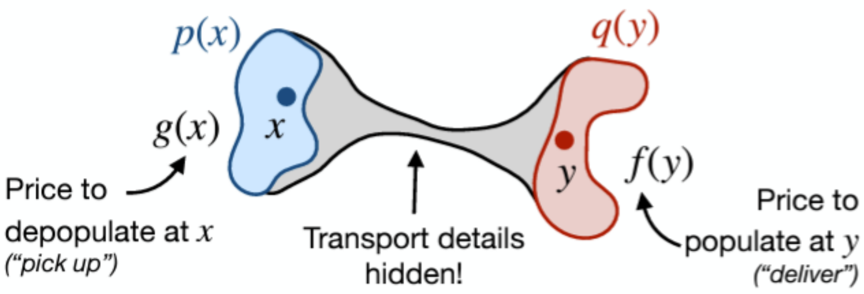
Generically: $F(x) = \int_0^x dx' f(x')$



ML optimal transport

Idea: Why not move the simulation instead of reweighting it?

- Optimal Transport
- Continuous calibration without histograms
- Easily scales to higher dimensions and cheap
- **Integral of sample unchanged**



$$\hat{T} = \arg\min_{T} \int dx\; p(x)\; c(x, T(x))$$

$$\pi(x,y) = p(x)\,\delta[y - T(x)] \qquad q(y) = p(x)\left(\frac{dT}{dx}\right)^{-1}$$

$$\hat{\pi} = \arg\min_{\pi} \int dx\,dy\; \pi(x,y)\; c(x,y)$$

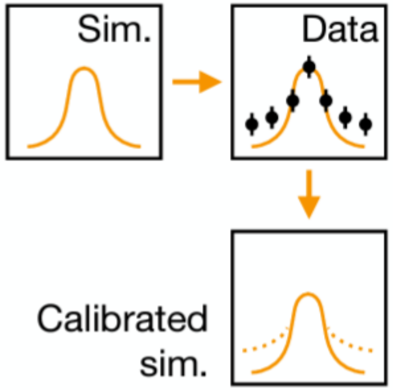
$$\int dy\; \pi(x,y) = p(x) \qquad \int dx\; \pi(x,y) = q(y)$$

$$\hat{f},\hat{g} = \arg\max_{f,g} \int dy\, q(y) f(y) + \int dx\, p(x) g(x)$$

$$g(x) + f(y) \le c(x,y)$$

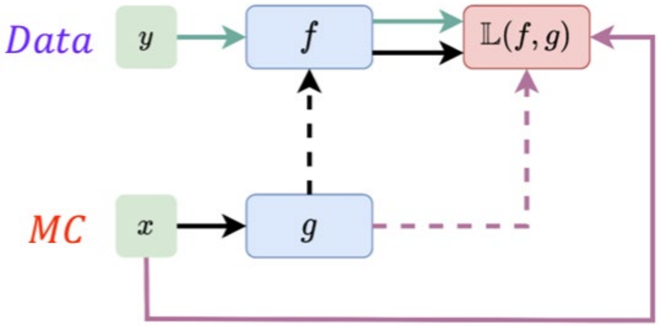
Optimal transport

- Very recently, a solution on how to train multi-dimensional OT with ML has been found.
- Brand new area of ML that is just now finding applications

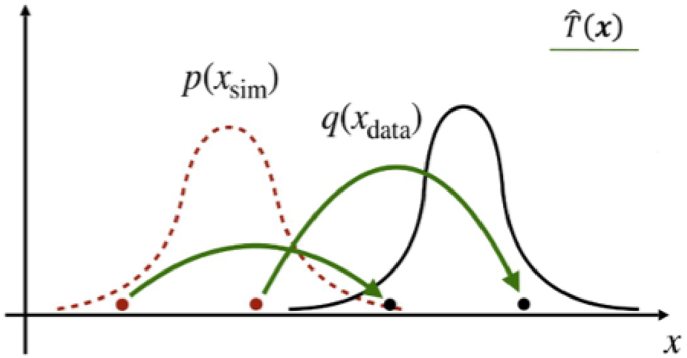


Calibration of simulation
(e.g. Monte Carlo prediction against data side bands)

[2107.08648]

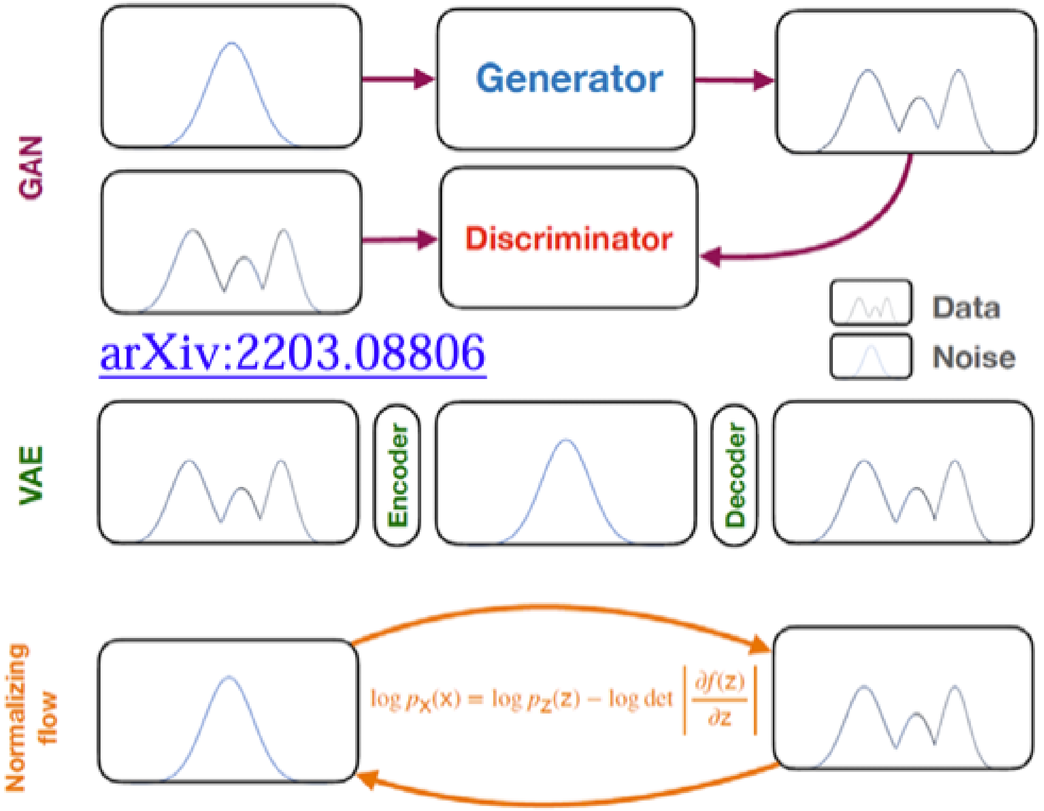


$\mathbb{L}(\phi,\psi) = \min_f \max_g \sum f_\phi(y;\theta) + x \cdot \nabla_x g_\psi(x;\theta') - f_\phi(\nabla_x g_\psi(x;\theta'),\theta')$ with the $\hat{T} = \nabla_x g(x;\theta')$



Generative methods

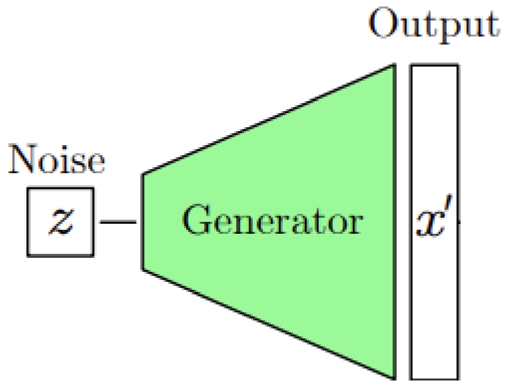
- Generative models ("replace"):
  - Usually *stochastic*
  - Generative Adversarial Networks (GANs)
  - Variational Autoencoders(VAEs)
  - Normalizing Flows (NFs)



arXiv:2203.08806

Generative Adversarial Network

- Generator Network $G(z) = x$
- Maps noise z to x

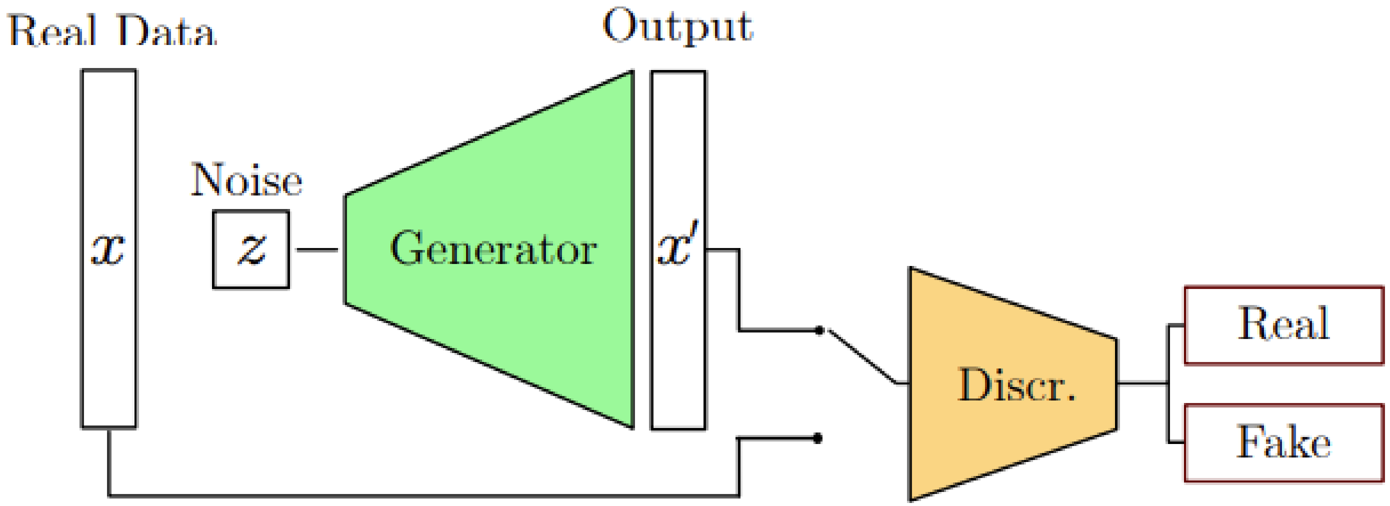


Generative Adversarial Network

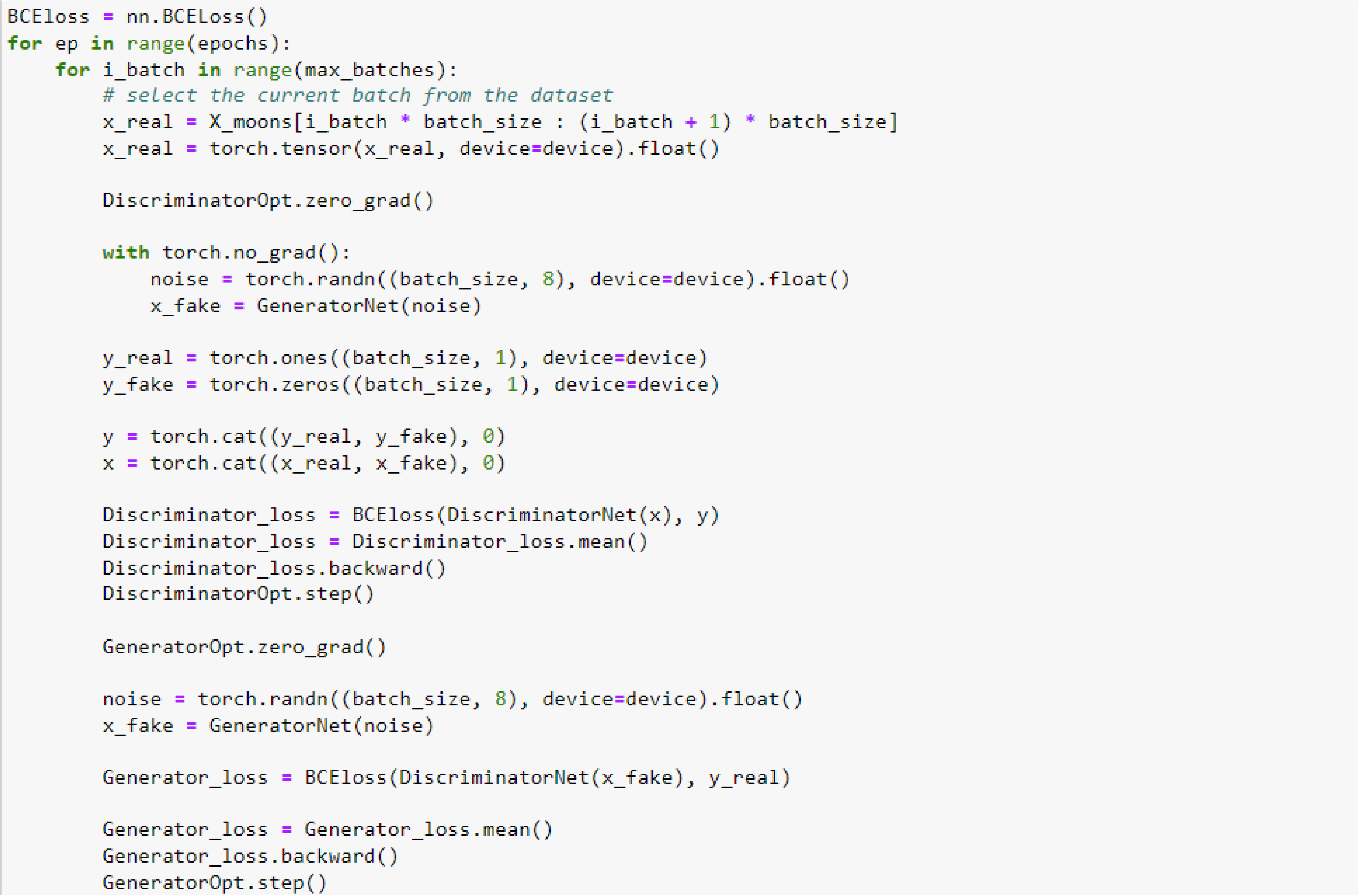
- Generator Network $G(z) = x$
- Maps noise z to x

- Discriminator $D(G(z))$ and $D(x)$
- Learns difference between real and fake

- $D(G(z))$ is differentiable function measuring performance
- Use $D(G(z))$ as loss to update G



Generative Adversarial Network



```
BCEloss = nn.BCELoss()
for ep in range(epochs):
    for i_batch in range(max_batches):
        # select the current batch from the dataset
        x_real = X_moons[i_batch * batch_size : (i_batch + 1) * batch_size]
        x_real = torch.tensor(x_real, device=device).float()

        DiscriminatorOpt.zero_grad()

        with torch.no_grad():
            noise = torch.randn((batch_size, 8), device=device).float()
            x_fake = GeneratorNet(noise)

        y_real = torch.ones((batch_size, 1), device=device)
        y_fake = torch.zeros((batch_size, 1), device=device)

        y = torch.cat((y_real, y_fake), 0)
        x = torch.cat((x_real, x_fake), 0)

        Discriminator_loss = BCEloss(DiscriminatorNet(x), y)
        Discriminator_loss = Discriminator_loss.mean()
        Discriminator_loss.backward()
        DiscriminatorOpt.step()

        GeneratorOpt.zero_grad()

        noise = torch.randn((batch_size, 8), device=device).float()
        x_fake = GeneratorNet(noise)

        Generator_loss = BCEloss(DiscriminatorNet(x_fake), y_real)

        Generator_loss = Generator_loss.mean()
        Generator_loss.backward()
        GeneratorOpt.step()
```

Generative Adversarial Network

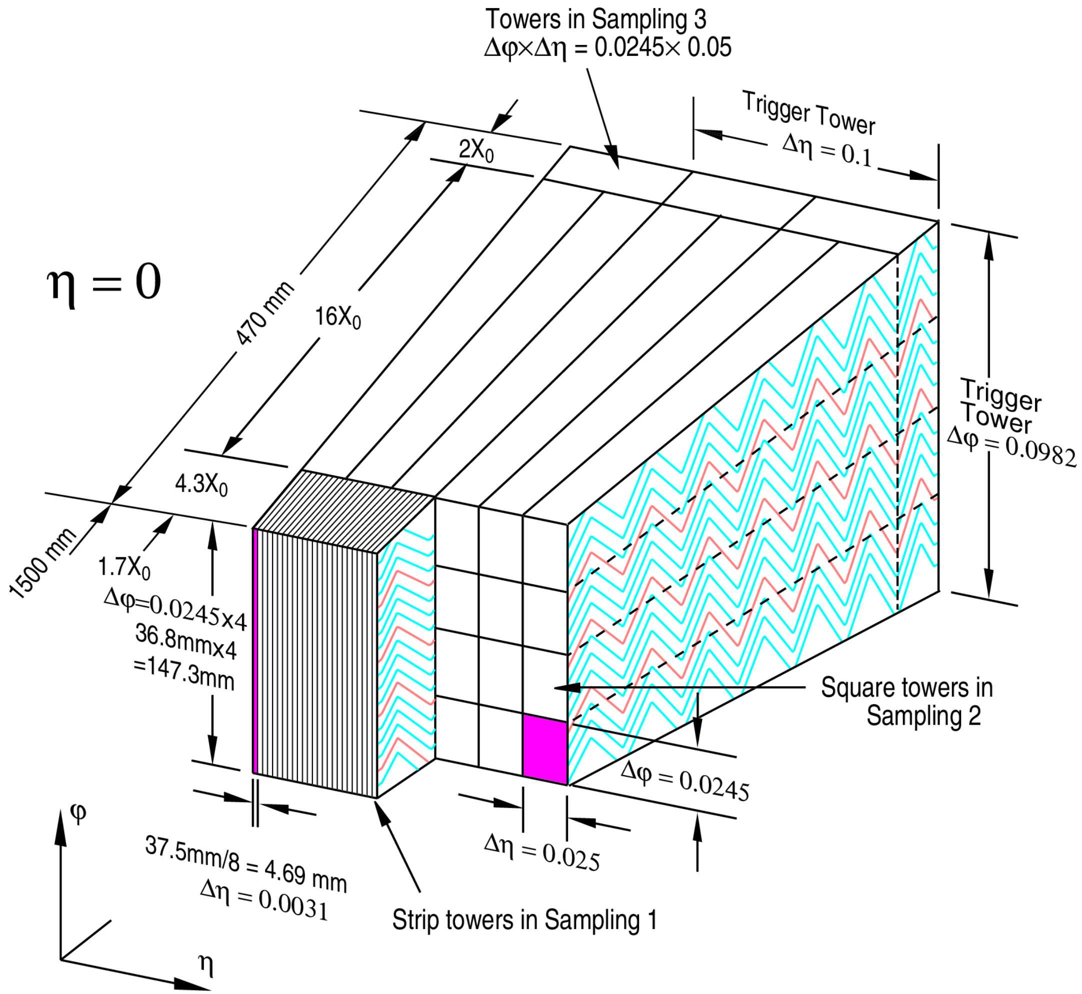
Upsides

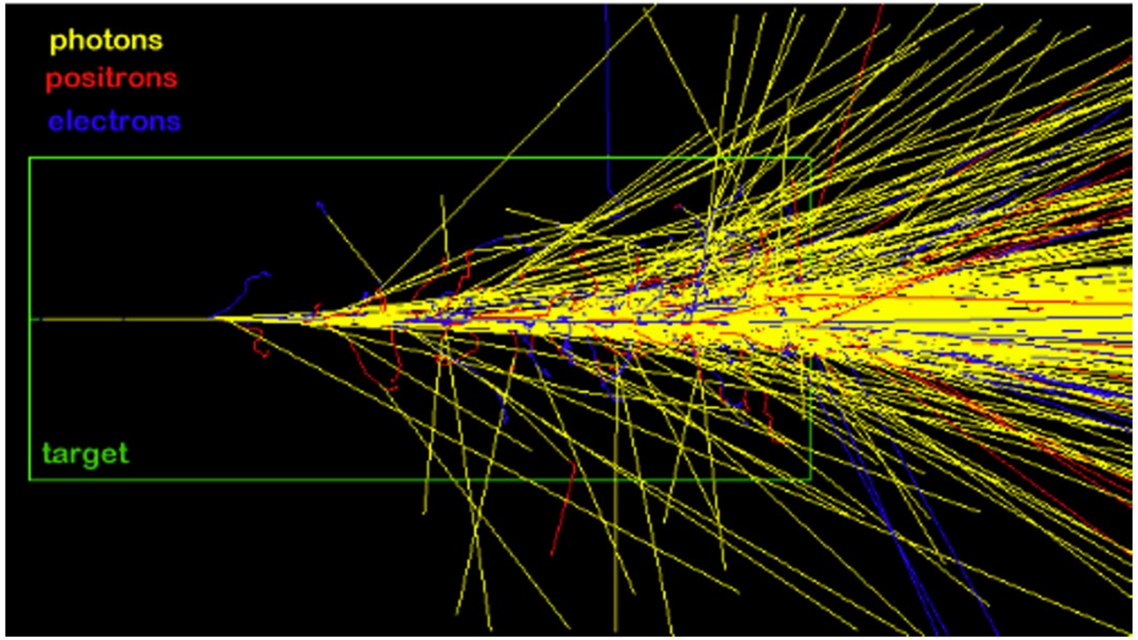
- Intuitive approach
- Easy to introduce additional constraints
- Well explored with several improvements (WGANs, normalizations)

Downsides

- Difficult to train
- Gen. and disc. needs to be balanced
- Can fail to converge
- Prone to mode collapse

Simulation of showers in ATLAS calorimeter



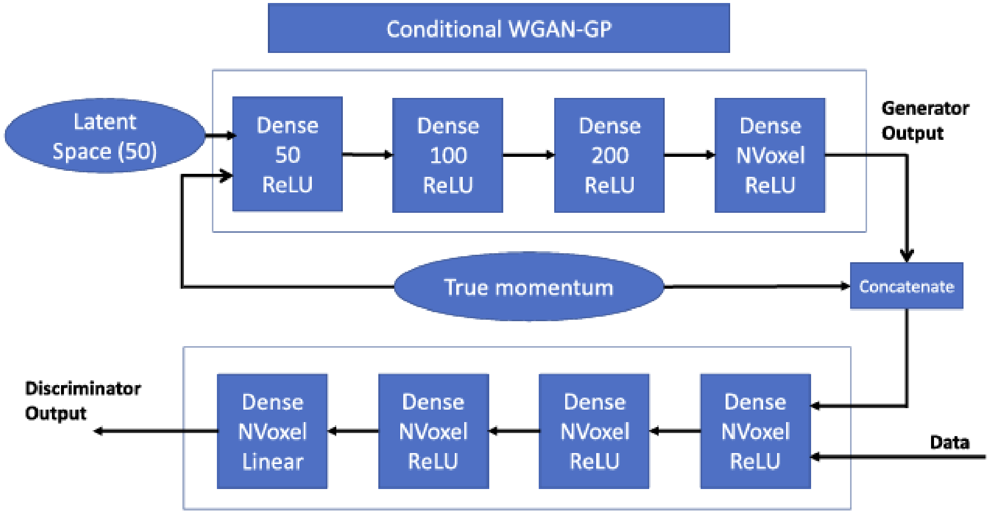


Simulation of showers in ATLAS calorimeter

FastCaloGAN V2

Different GAN for different type of particles and for different eta slices.

Prediction of deposit of energy in “voxels” which allow HITS reconstruction.



Conditional WGAN-GP

Latent Space (50)

Dense 50 ReLU

Dense 100 ReLU

Dense 200 ReLU

Dense NVoxel ReLU

Generator Output

True momentum

Concatenate

Discriminator Output

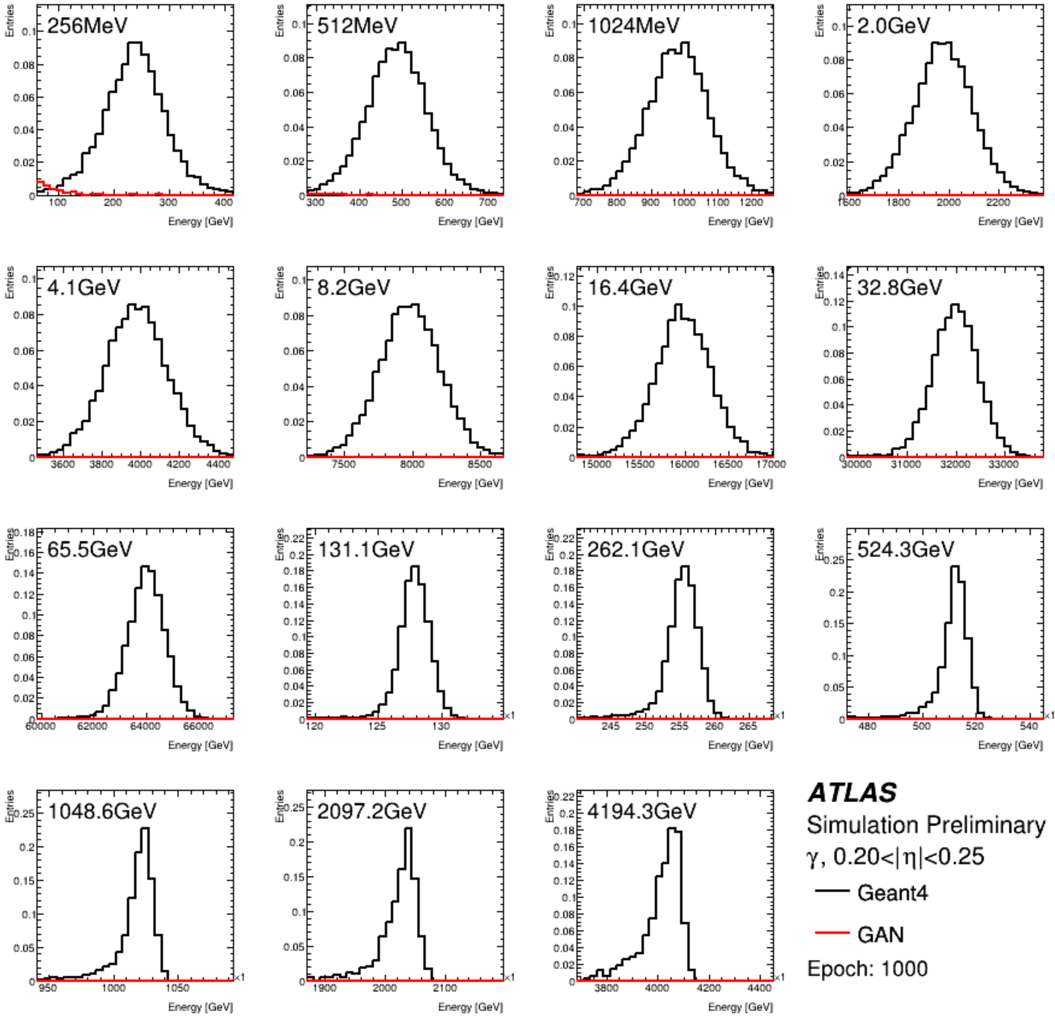
Dense NVoxel Linear

Dense NVoxel ReLU

Dense NVoxel ReLU

Dense NVoxel ReLU

Data

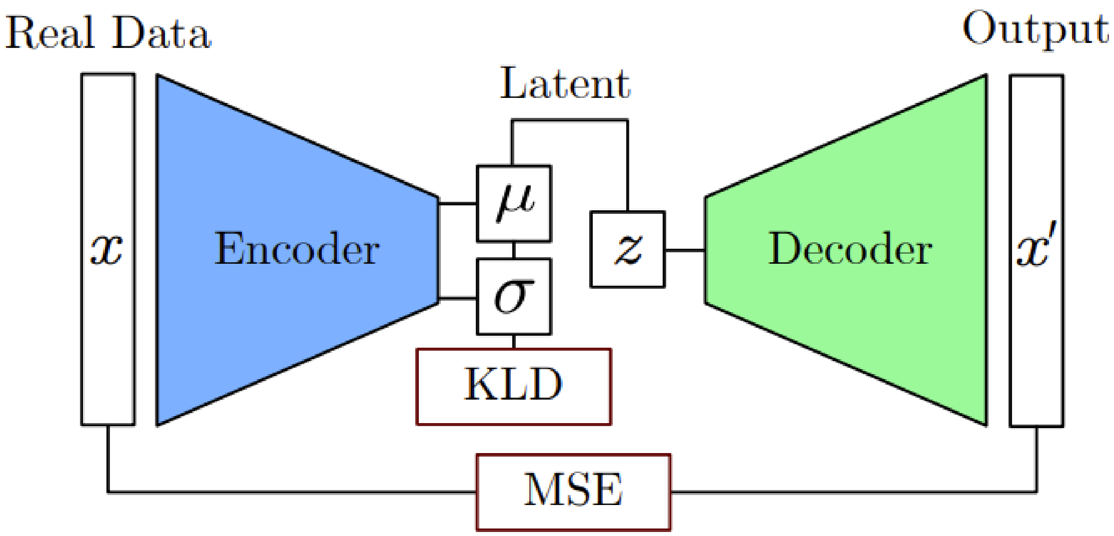


How do we use this in a fast MC?

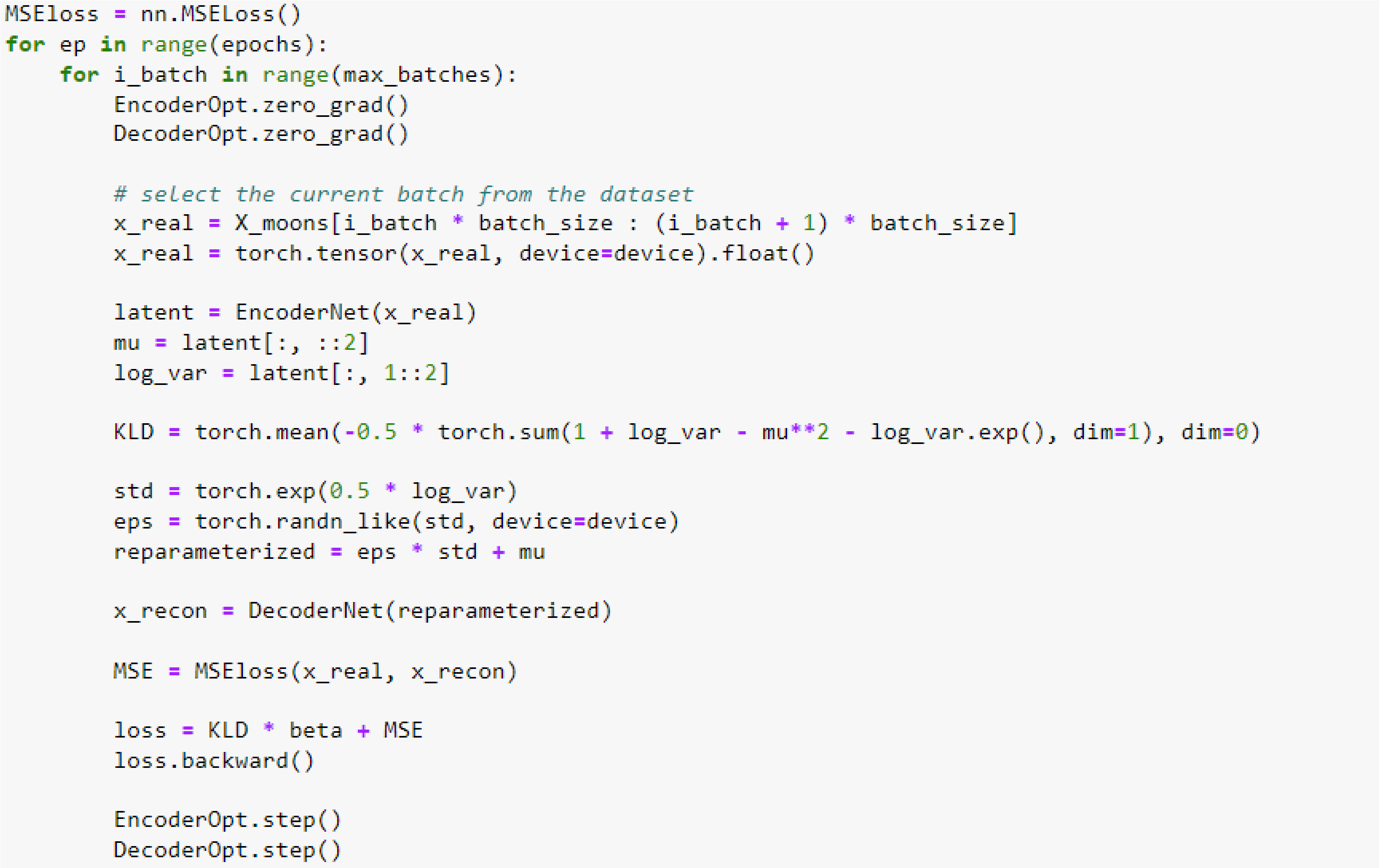
	Inner Detector	Calorimeters	Muon Spectrometer
Electrons Photons	Geant4	FastCaloSimv2	
Hadrons		Geant4 pions: $E_{kin} < 200$ MeV Other hadrons: $E_{kin} < 400$ MeV FastCalo Sim V2 $E_{kin} < (8-16)$ GeV FastCalo GAN $(8-16)$ GeV $< E_{kin}$ $< (256-512)$ GeV FastCalo Sim V2 $E_{kin} > (256-512)$ GeV	Muon Punchthrough +Geant4
Muons		Geant4	Geant4

Variational AutoEncoders (VAE)

- Encoding function $E(x) = z$ map high dimensional data X to low dimensional latent space Z
  - Decoding function $D(z) = x$ map latent space Z back to data X
  - Compare Input and Output with mean squared error
- Sample for Z and pass it to $D(Z)$ → Generate new samples
- Latent space: Series of Gaussians, regularised match $N(\mu = 0, \sigma = 1)$
  - Using Gaussians lets us use Kullback–Leibler divergence
  - $\sum_{i=1}^{n} \sigma_i^2 + \mu_i^2 - \log(\sigma_i) - 1$



Training Variation AutoEncoders



```
MSEloss = nn.MSELoss()
for ep in range(epochs):
    for i_batch in range(max_batches):
        EncoderOpt.zero_grad()
        DecoderOpt.zero_grad()

        # select the current batch from the dataset
        x_real = X_moons[i_batch * batch_size : (i_batch + 1) * batch_size]
        x_real = torch.tensor(x_real, device=device).float()

        latent = EncoderNet(x_real)
        mu = latent[:, ::2]
        log_var = latent[:, 1::2]

        KLD = torch.mean(-0.5 * torch.sum(1 + log_var - mu**2 - log_var.exp(), dim=1), dim=0)

        std = torch.exp(0.5 * log_var)
        eps = torch.randn_like(std, device=device)
        reparameterized = eps * std + mu

        x_recon = DecoderNet(reparameterized)

        MSE = MSEloss(x_real, x_recon)

        loss = KLD * beta + MSE
        loss.backward()

        EncoderOpt.step()
        DecoderOpt.step()
```

Variational AutoEncoders

Upsides

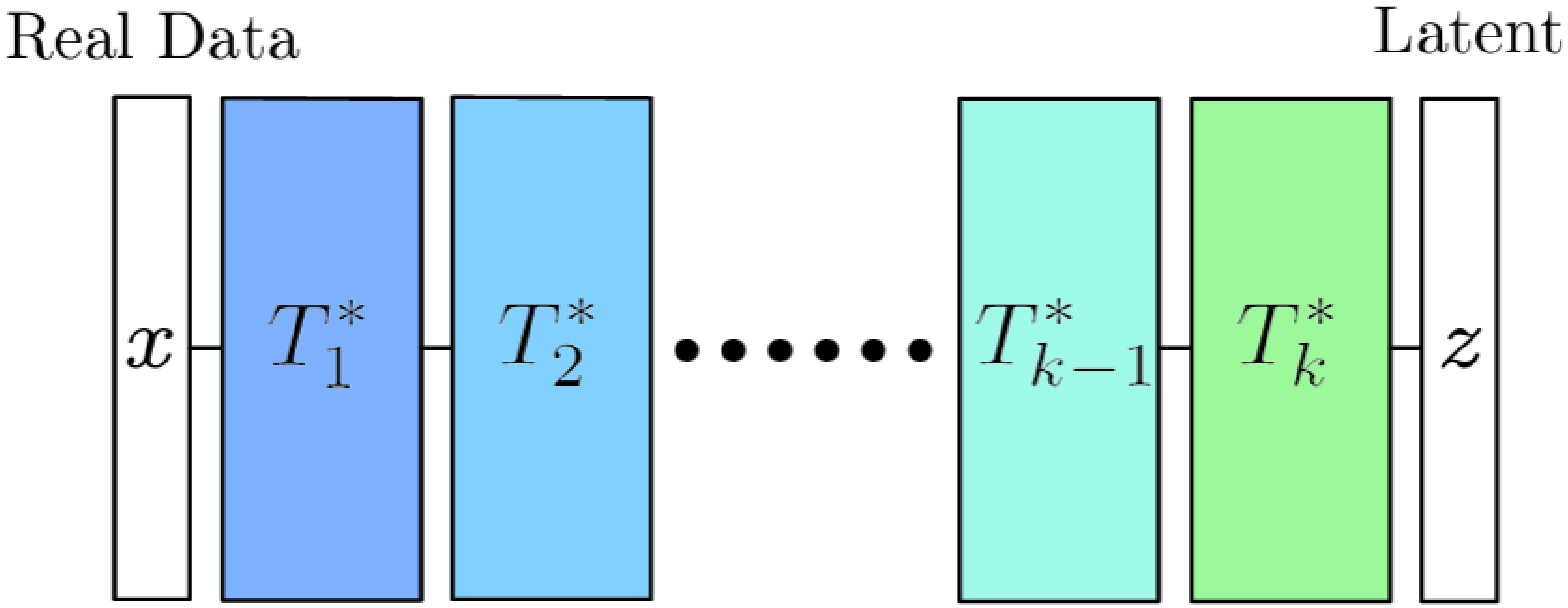
- Directly evaluates log likelihood
- Stable in training

Downsides

- MSE loss insufficient for certain data sets
- Needs to balance KLD and MSE loss terms

Normalizing flows

- Variational AutoEncoder: map data to normal distribution and back using two networks
- Can we do this with a single network instead?



Normalizing flows

- Train invertible model T^{*-1} to map data to Normal distribution
- Well understood loss function:

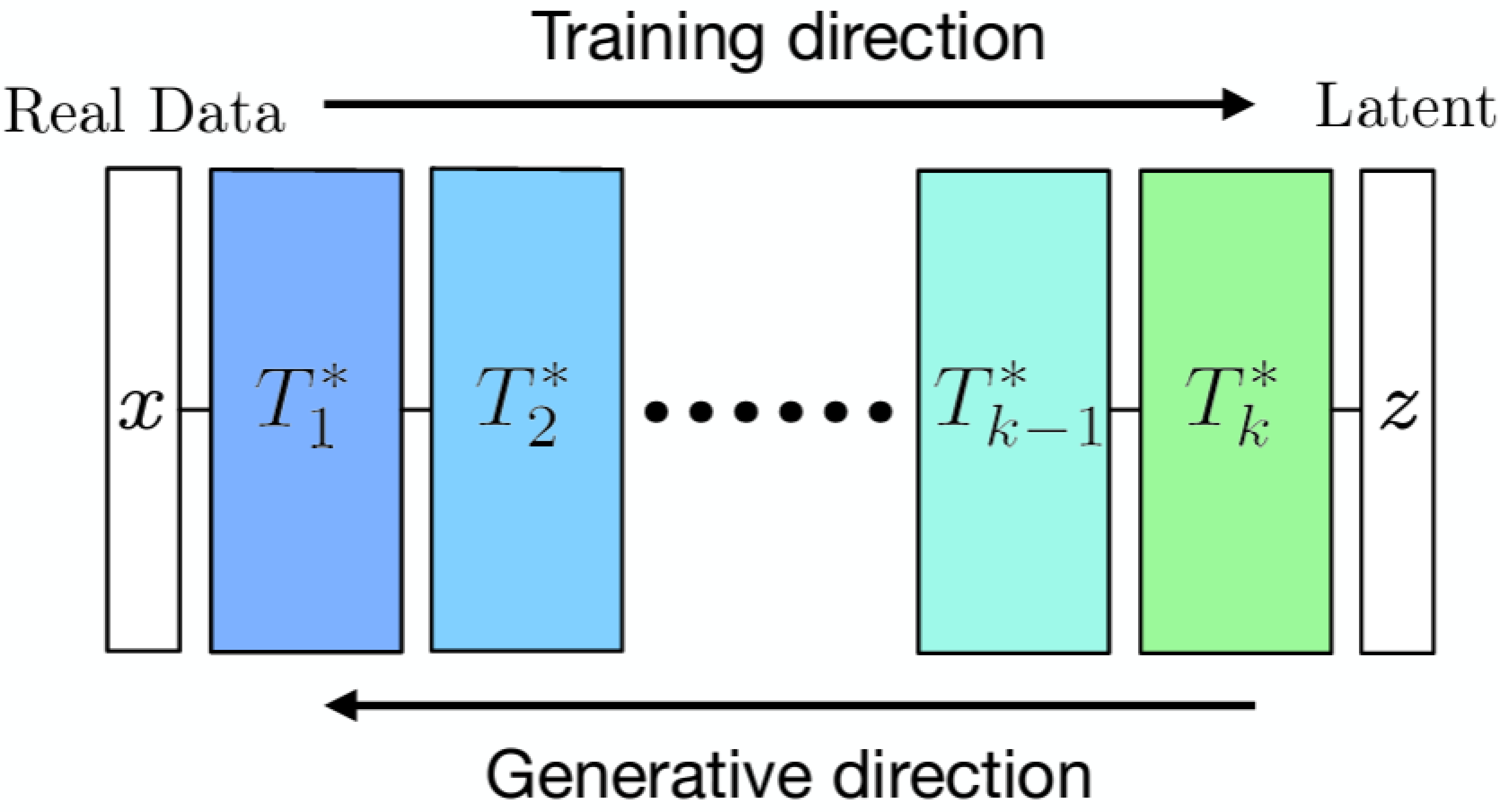
$$\mathcal{L}_{\text{Flow}} = \frac{1}{N}\sum_{n=1}^{N} \boxed{\log(p_z(T^{*-1}(\mathbf{x_n}, \theta)))} + \boxed{\log(|\det J_{T^{-1}}(\mathbf{x_n}, \theta)|)}$$

Latent Normal distribution

How well does the transformed sample match the latent distribution

Jacobean of transformation

Normalizing flows



Normalizing Flows

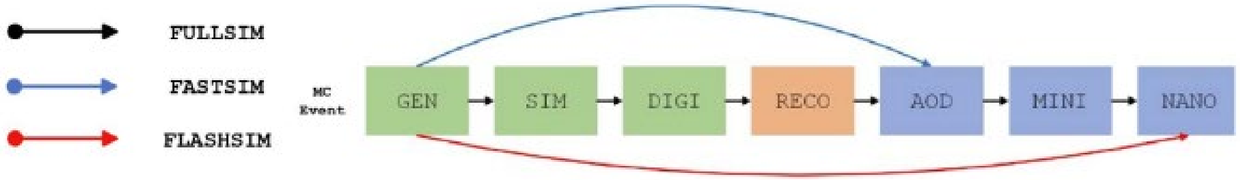
Upsides

- Directly evaluates log likelihood
- Stable in training
- High generative quality
- Easy to train and use

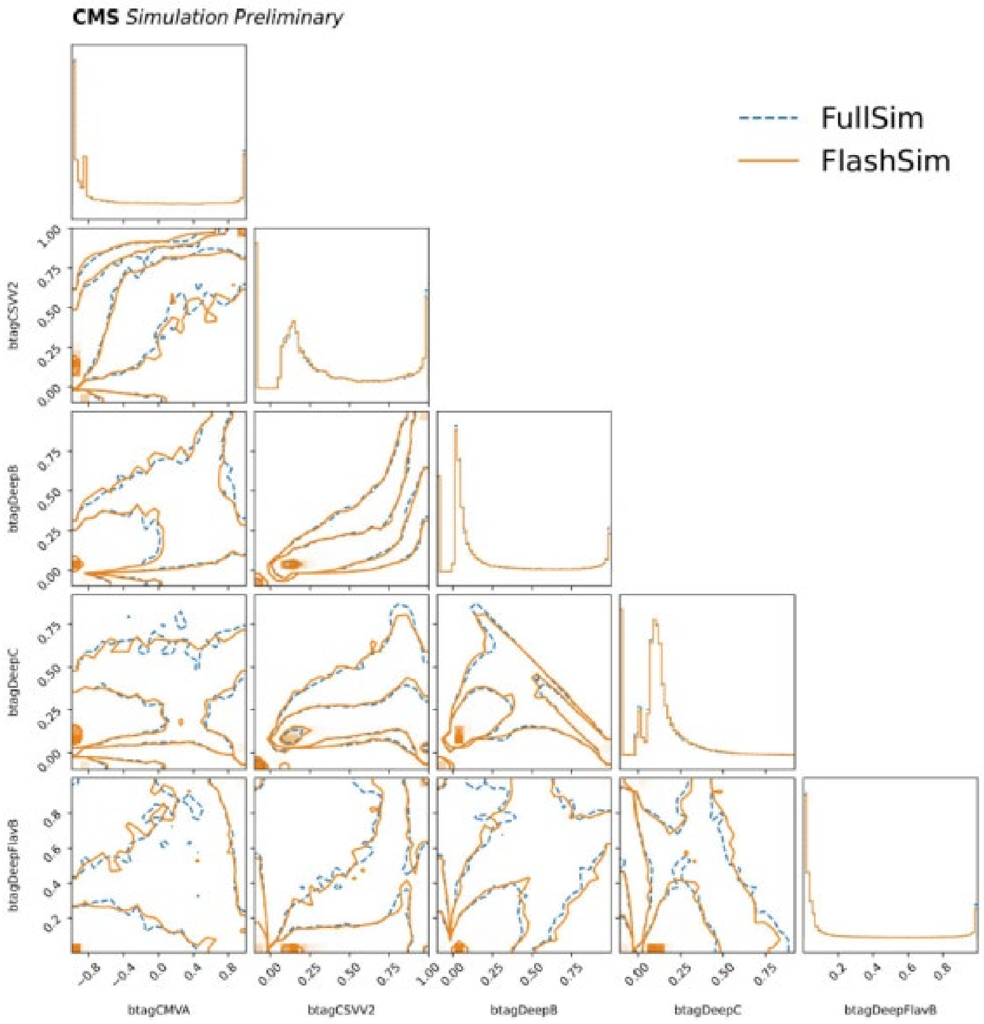
Downsides

- Fixed dimensionality through entire flow
- Slow generation times for large models/data

How do we use this?



- Normalizing flow to predict high-level analysis quantities from generator-level information
- Reproduces correlations even in ML b-tagging algorithm scores
- Very promising solution for end-stage analyses
  - Effectively infinite MC → minimize statistical fluctuations



Simulation-based inference

- Remember that in my first slide I said that the purpose of an analysis was to calculate

$$p\left(x^{\text{data}}\middle|H_1\right) \text{ and } p\left(x^{\text{data}}\middle|H_0\right)$$

- The methods presented here allows us to approximate these probabilities densities with much more precision than simple histograms