Generating LCD Calorimeter Showers with Generative Adversarial Networks (GANs)

Dawit Belayneh, University of Chicago
Supervisor: Mia Liu, Fermilab

Introduction

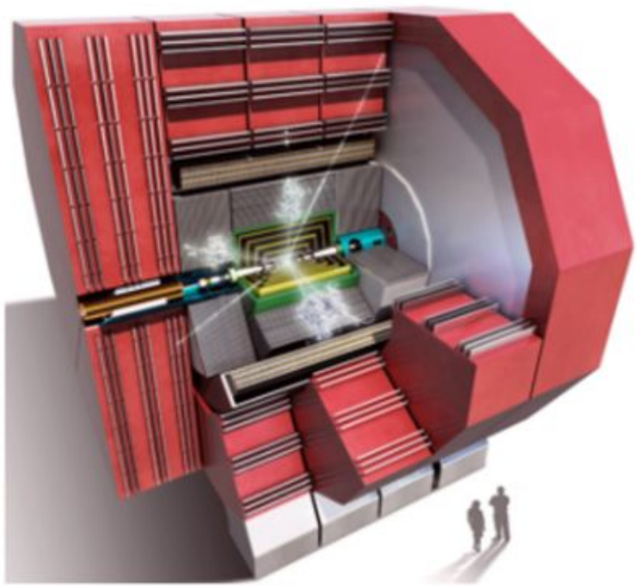
- Worked on a Machine Learning framework (Triforce) during my internship at Fermilab.
- Triforce is a machine learning framework based on Pytorch:
    - Tries to synchronize efforts on Classification/Regression/GAN for reproducible results.
    - Input dataset: Linear collider detector (LCD) calorimeter dataset.
    - Implements various Networks for ML Tasks: Particle ID, Energy regression, and Calorimeter shower simulation.

Introduction

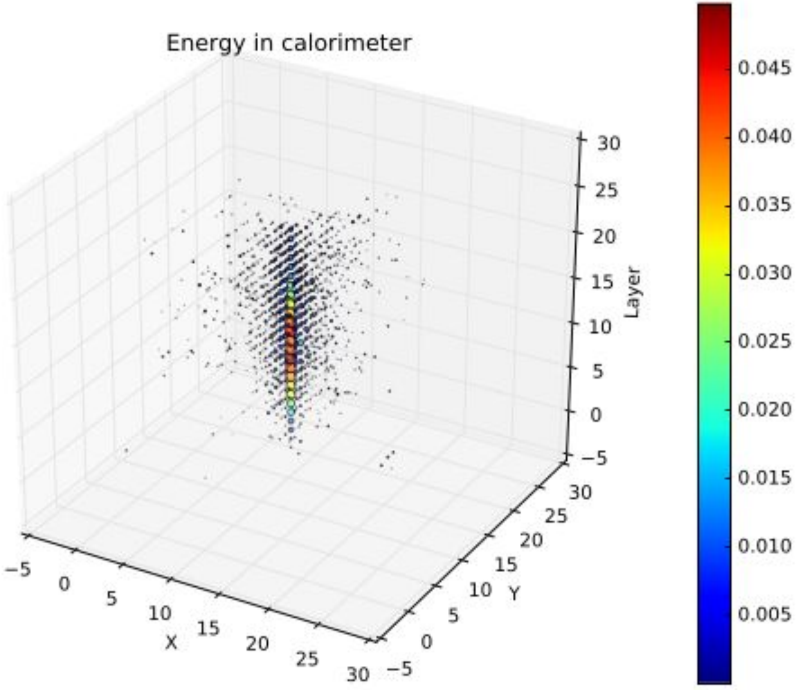
- My main focus:
  - Improving training speed (with focus on I/O speed) in Triforce.
  - Implement Generative adversarial networks (GANs) to simulate Linear Collider Detector (LCD) calorimeter showers

LCD Dataset

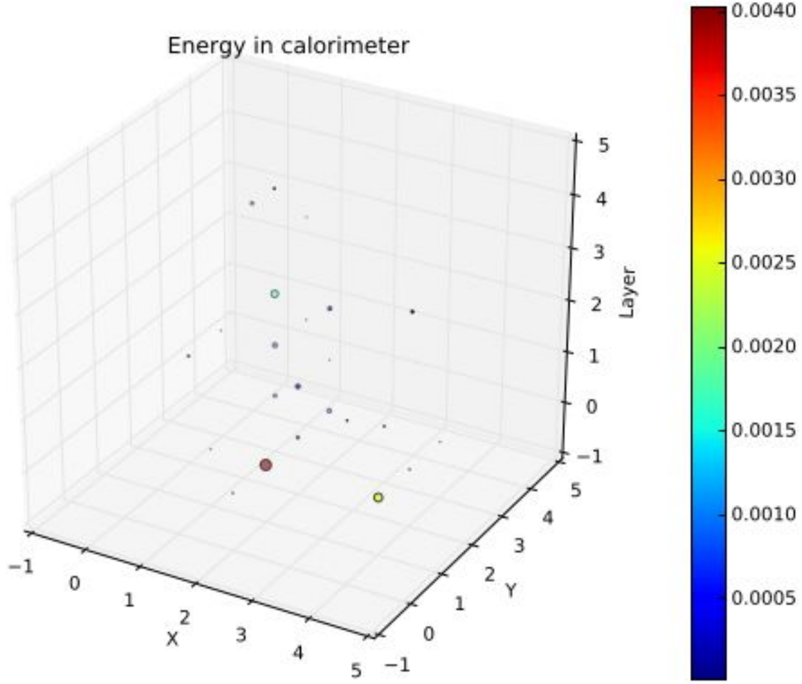
- Data generated using geometry for the LCD detector of the Compact Linear Collider (CLIC) linear electron - positron collider.
- A 51x51x25 ECAL slice and a 11x11x60 HCAL slice is cut around each object interacting with the calorimeters. Energy deposit from each cell is saved in a 3D array.
- Particle types: neutral pion, charged pion, electron, and photon
- Energy range: 10 - 500 Gev.
    - Classification: 60 Gev
    - Regression: 10 - 500 Gev
    - GAN: 100 - 500 Gev



60 GeV Photon Sample

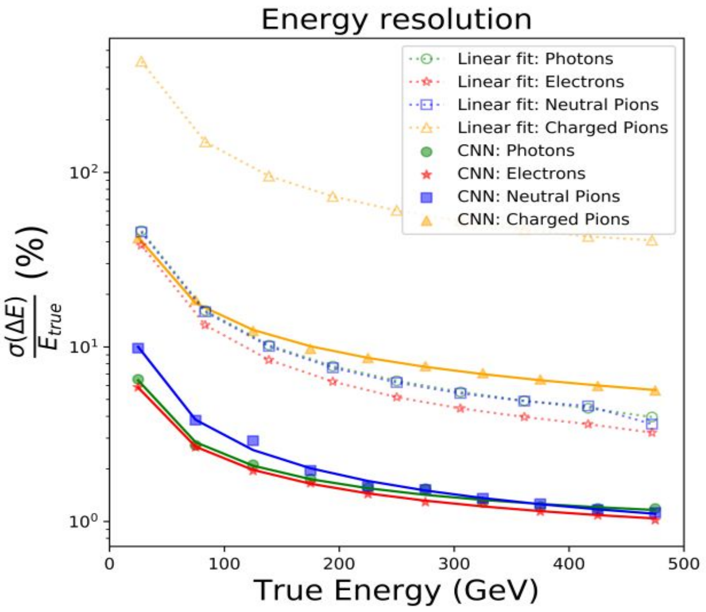


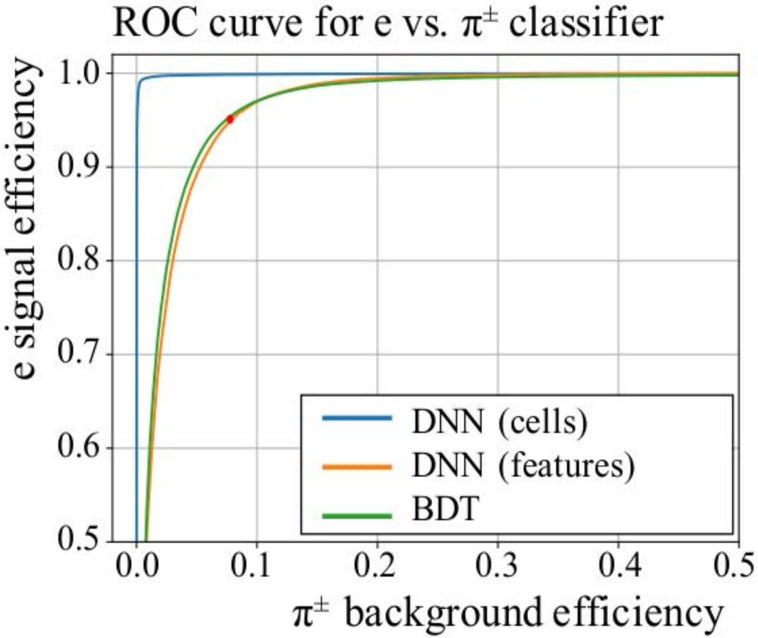
ECAL



HCAL

Classification + Regression in Triforce





- Simple 4 layer, 256 neuron DNN classifier
- Separate regression DNN composed of ECAL and HCAL CNNs
- Both architectures implemented for the next iteration of the results

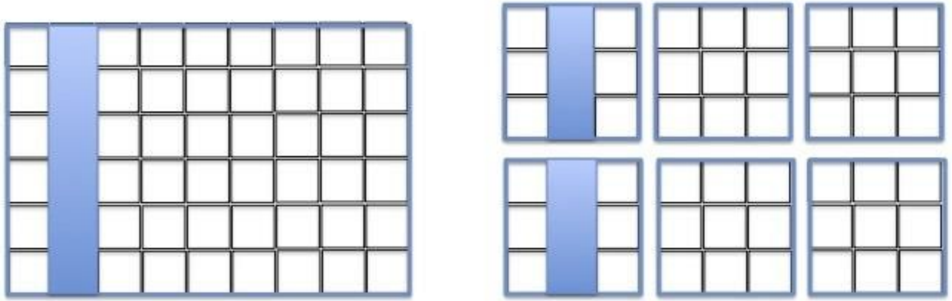
I/O in Triforce: Problem

- I/O is bottleneck for training.
  - Single file load speed: ~250s.
  - Typical training session spends **~ 7 hrs on puerly I/O tasks**.
- As a result, training time ~ days (a week for GANs) !!
- Reasons for I/O speed:
  - Size (10,000x51x51x25) and high sparsity (more than 90 % of all values are zero) of our ‘ECAL’ array.
  - Decompression takes time, data compression settings were not optimized.
  - Files were loaded sequentially.

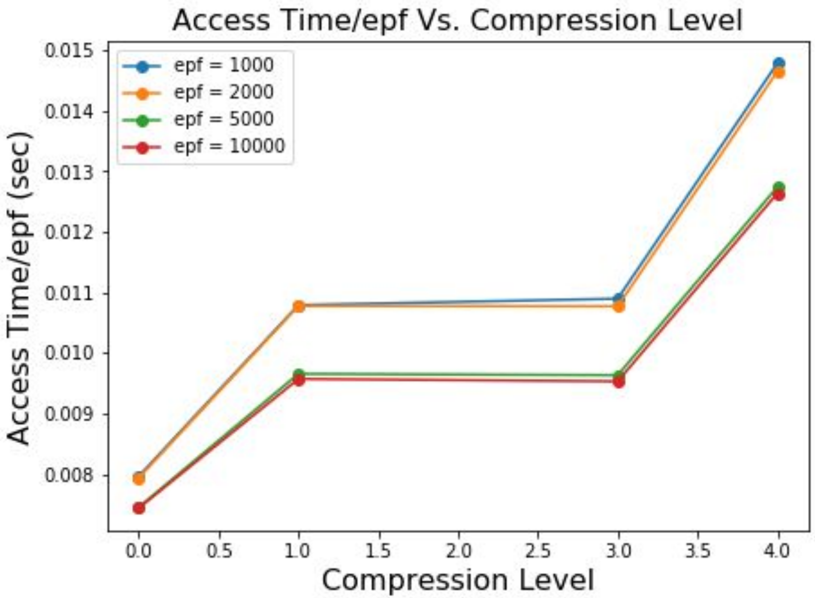
I/O in Triforce: Attempts

- Improve I/O by optimizing compression settings
    - Some I/O speed up
- Load a single file using multiple processes.
    - Required re-building most of our software to enable single file parallel reading
    - **Promising option for other projects**
- Sparse array representations.
    - Storing indices of non-zero values
    - Minimal support in Pytorch
- Parallel I/O with smaller datasets ⟸ **OUR FIX.**

Compression Optimization



Chunking in HDF5

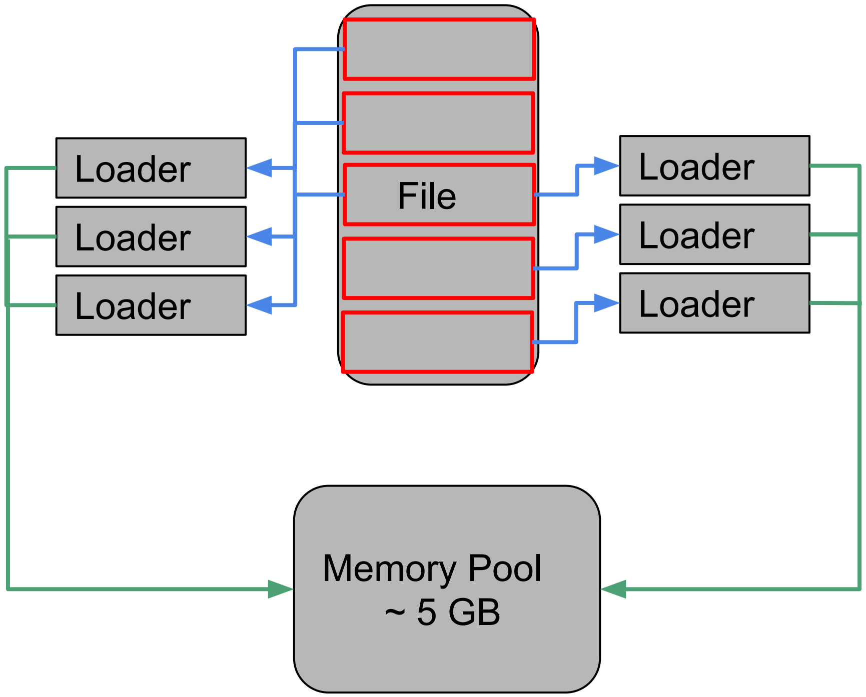


- Experienced speed up (~ 20 seconds/file): ~40% by optimizing compression level.
- 2x I/O speed up by switching to smaller chunk sizes (10x51x51x25)

Compression Level	0	1	2	3	4 (default)
Size (MB)	6000	214	212	209	183
Access Time Increase (%)	0	14	14	13	52

Attempt: Single File Parallel I/O

- Implemented single file parallel I/O with python concurrency methods
    - ~10 times speedup
- Unfortunately Incompatible with Triforce environment on caltech cluster and Bluewaters at UIUC
    - **Requires rebuilding HDF5, h5py, MPI** with parallel config
- Can be an option for other projects.

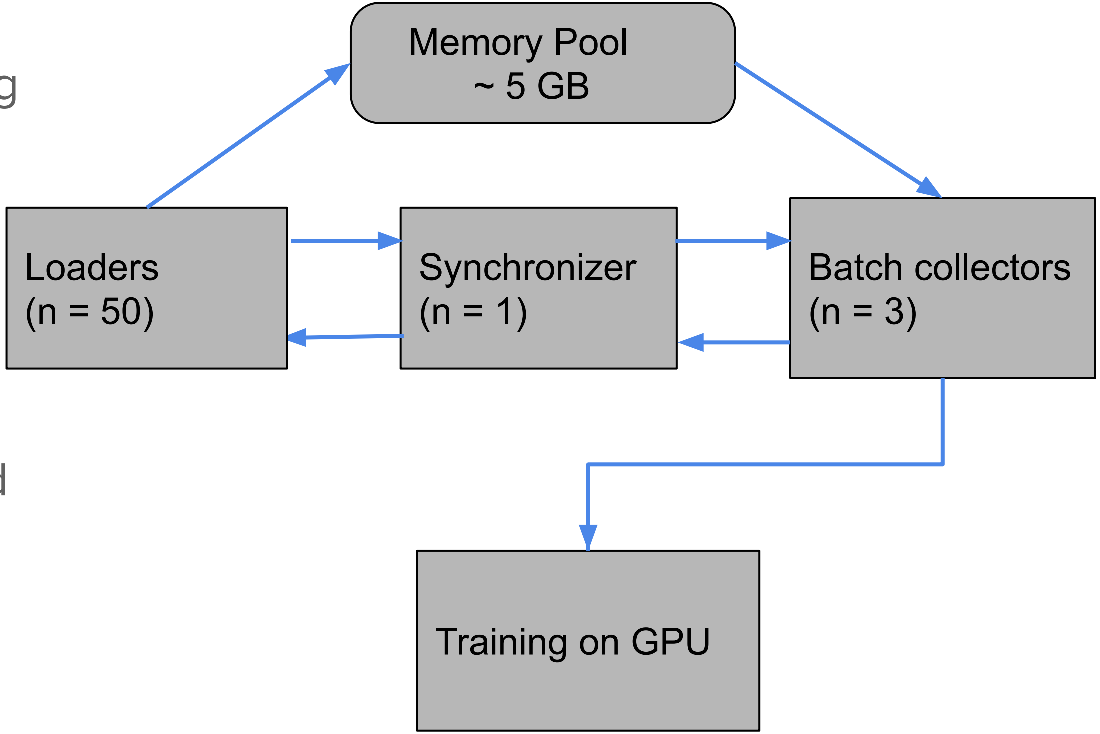


I/O in Triforce: Our Fix

- Split files into small files containing no more than 200 events per file
- Instead of loading a single file using multiple processes, load smaller files at the same time using multiple processes (Parallel I/O).
- Optimize compression settings (amplified by multiprocessing)
- With these changes we managed to **speed up I/O 40 times**!

Parallel I/O: Code Implementation

- Over allocate memory during initialization
- Spawn multiple processes (workers)
- Single worker synchronization
- Differentiate b/n loaders and batch collectors



Parallel I/O workflow in Triforce

I/O in Triforce: Lessons Learnt

- Multiprocessing can offer great speed up when loading big data.
- Make sure default compression settings are optimal for task
- Avoid using Python slice syntax on big arrays (*big_array[:]*) and *append* calls
- Over allocate memory during init for data to be read into
- Smaller files (less events per file) are easier for use by parallel I/O
- Avoid passing large arrays between multiple processes in Python.

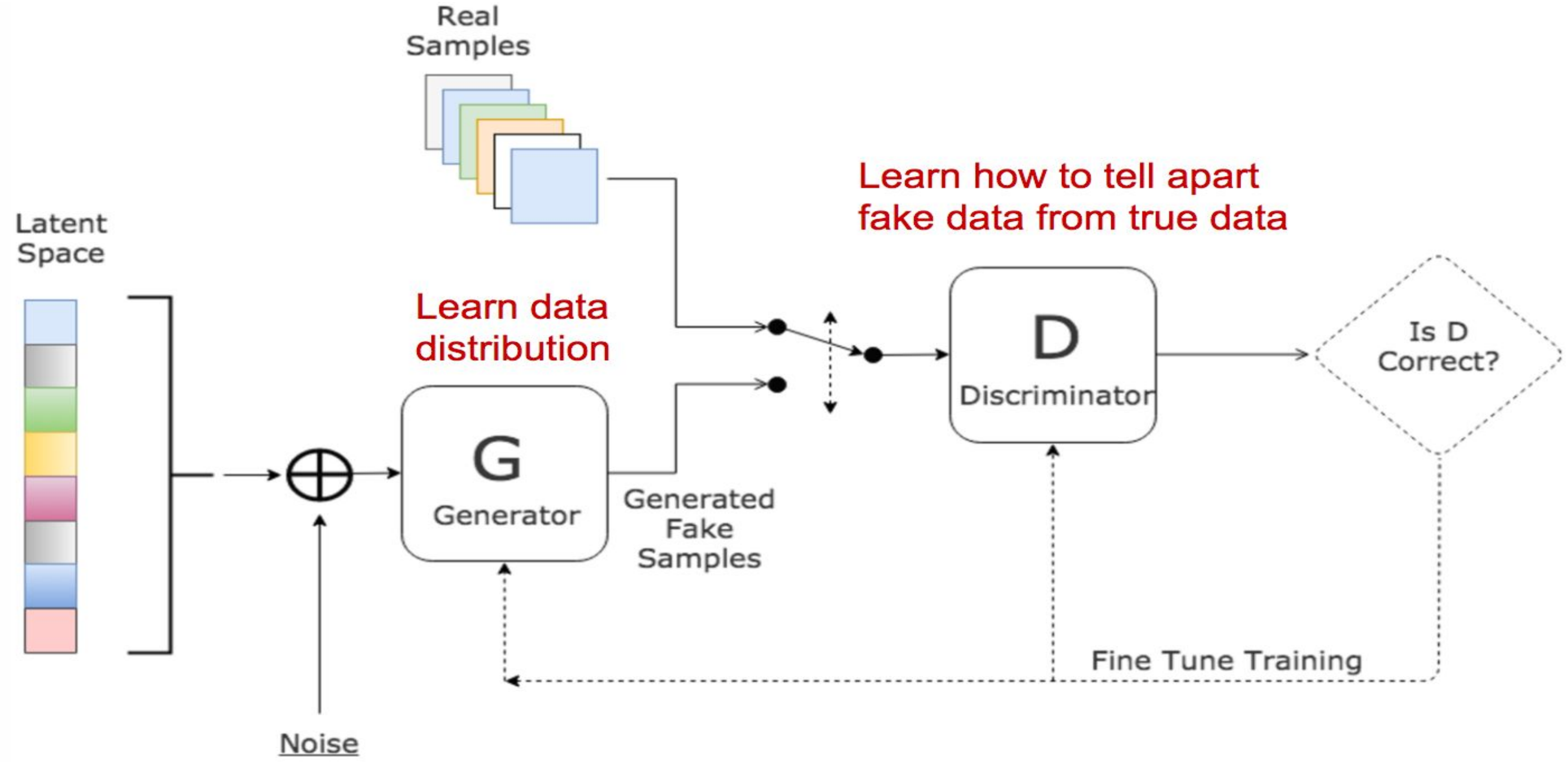
GANs

- A generator and discriminator network are in competition.
- Generator tries to mimic a given dataset (e.g: faces of celebrities)
- Discriminator differentiates between real and generated images and feedback is sent to generator.
- As a result, generator learns complex underlying behaviour of a dataset.

High-Res fake celebrity images generated using GANs.

Source: arXiv:1710.10196 [cs.NE]

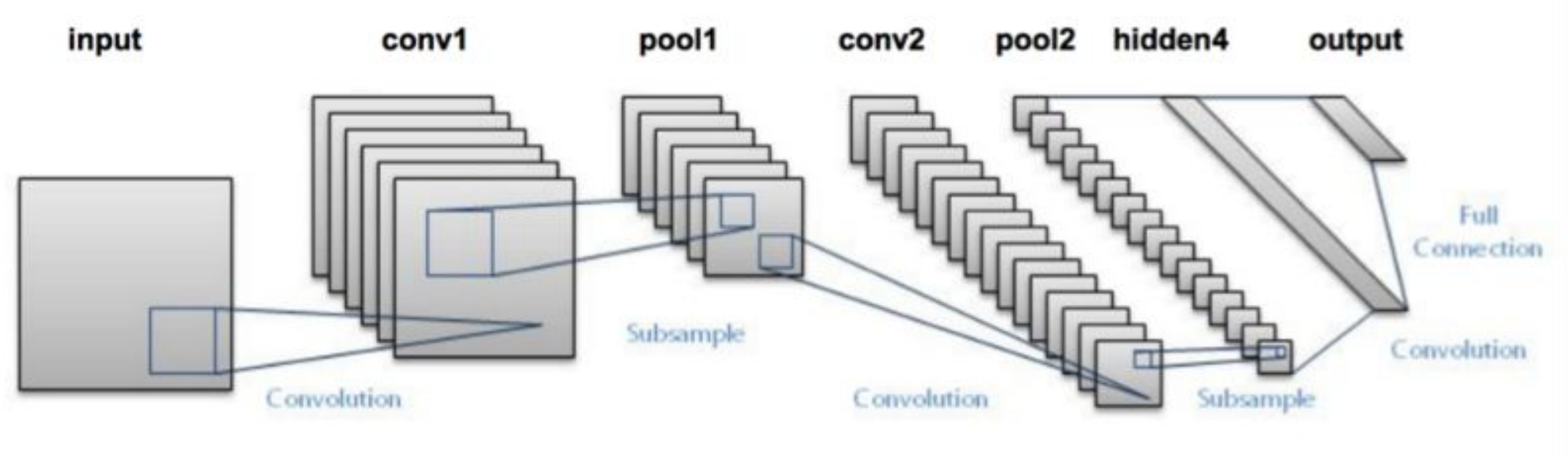
GAN training: general routine for training GANs



Source: http://www.kdnuggets.com/2017/01/generative-adversarial-networks-hot-topic-machine-learning.html

GAN Implementation

- Discriminator network is identical to Binary Classifiers. (In this case classifying between real and fake images)



- Generator network is similar to the above architecture but in reverse.

Status of implementing GAN in Triforce

- Implemented Generative Adversarial Networks for calorimeter simulation
  - Debugging, training, and testing in progress
- For past GAN results (based on keras, older version of the dataset) please see the group's paper: https://dl4physicalsciences.github.io/files/nips_dlps_2017_15.pdf

Future Work

- Debug and improve GAN networks to better simulate LCD calorimeter data.
- Implement Auto-Encoders in Triforce for calorimeter simulation.
- Minimize the memory footprint of Triforce.
- Extend Triforce to work on other calorimeter datasets with boosted signatures.

Thank You!

Discriminator Architecture

```
----------------------------------------------------------------------------
      Layer (type)              Output Shape         Param #
================================================================
          Conv3d-1       [-1, 32, 25, 25, 25]          4,032
         Dropout-2       [-1, 32, 25, 25, 25]              0
          Conv3d-4        [-1, 8, 25, 25, 25]         32,008
     BatchNorm3d-5        [-1, 8, 25, 25, 25]             16
         Dropout-6        [-1, 8, 25, 25, 25]              0
         Conv3d-12        [-1, 8, 23, 23, 23]          8,008
          (......)
    BatchNorm3d-13        [-1, 8, 23, 23, 23]             16
        Dropout-14        [-1, 8, 23, 23, 23]              0
      AvgPool3d-15        [-1, 8, 11, 11, 11]              0
         Linear-16                       [-1]         10,649
================================================================
Total params: 73,402
Trainable params: 73,402
Non-trainable params: 0
----------------------------------------------------------------------------
```

- Takes in an Input Image of shape (1, 25, 25, 25) either from dataset or generated data.
- Outputs a single number signifying the probability that a given image is fake or real.

Generator Architecture

Layer (type)	Output Shape	Param #
Linear-1	[-1, 3136]	3,214,400
Conv3d-2	[-1, 64, 6, 6, 7]	147,520
BatchNorm3d-3	[-1, 64, 6, 6, 7]	128
Upsample-4	[-1, 64, 12, 12, 14]	0
Conv3d-5	[-1, 6, 13, 14, 9]	92,166
Upsample-6	[-1, 6, 26, 28, 27]	0
Conv3d-7	[-1, 6, 26, 26, 26]	2,598
Conv3d-8	[-1, 1, 25, 25, 25]	49

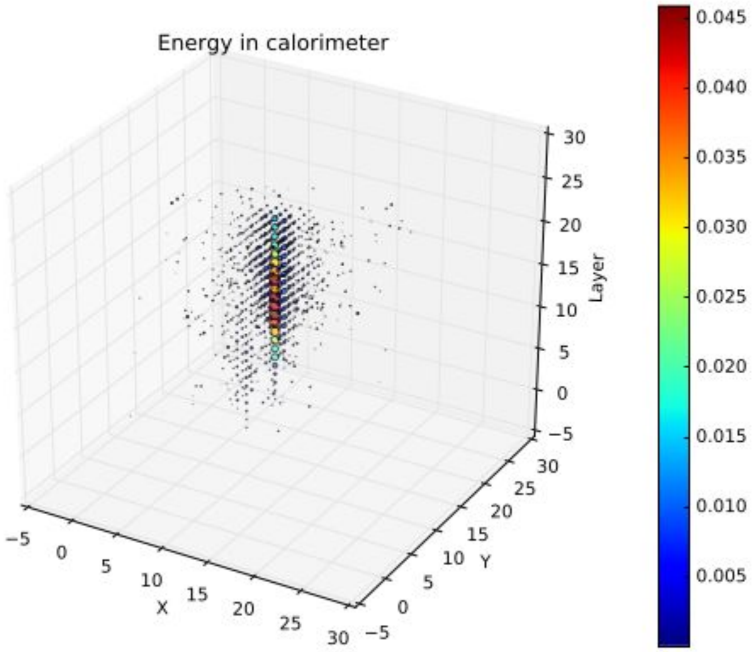
Total params: 3,456,861

Trainable params: 3,456,861

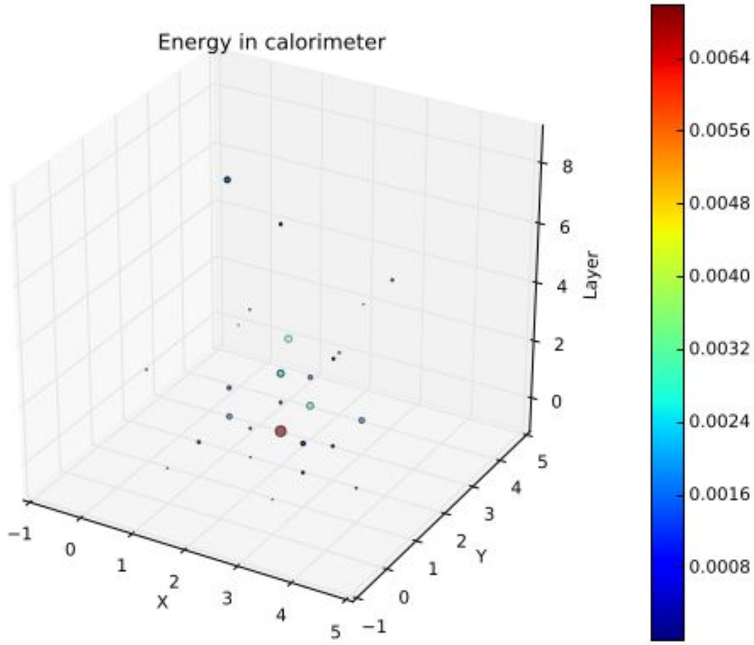
Non-trainable params: 0

- Takes in a tensor of shape (1, 1024) sampled from a random distribution
- Outputs an image of shape (1, 25, 25, 25) mimicking the features of the dataset we trained it on.
- By the end of training our generator has hopefully learnt to mimic the physics inside our calorimeter.

60 GeV $\pi_0 \rightarrow \gamma\gamma$ Sample w/ Opening Angle < 0.01 rad



ECAL



HCAL