GANs and VAEs testing

A.M. Marshall

August 2018

1 Introduction

Here is presented a brief summary of the set up and procedure for GAN and VAE testing. All approaches here make heavy use of the Keras deep learning framework built on top of TensorFlow.

2 Pre-processing

Data is pulled from Thomas' production files

`/eos/experiment/ship/data/Mbias/background-prod-2018/`
`pythia8_Geant4_10.0_withCharmandBeauty%d000_mu.root"%file_number`

This sample is weighted with various contribution weighted up (vector meson decay, contributions from charm and beauty). Firstly note the muons I save are muons that got to the end of the hadron absorber in the `run_fixedTarget.py` production. These are muons that have hits in the `vetoPoint` tree. These muons are split into 4 categories and data is saved separately. The categories are as follows:

- Category 1: Positive muons `vetoPoint.PdgCode() == -13` that have `MCTrack.StartX() == 0` and `MCTrack.StartY() == 0`
- Category 2: Negative muons `vetoPoint.PdgCode() == 13` that have `MCTrack.StartX() == 0` and `MCTrack.StartY() == 0`
- Category 3: Positive muons `vetoPoint.PdgCode() == -13` that have `MCTrack.StartX() != 0` and `MCTrack.StartY() != 0`
- Category 4: Negative muons `vetoPoint.PdgCode() == 13` that have `MCTrack.StartX() != 0` and `MCTrack.StartY() != 0`

for Category 1 and Category 2 data is stored in the following format for each muon:

```
[[e.GetWeight()/768.75,e.GetStartZ(), e.GetPx(), e.GetPy(), e.GetPz()]]
```

where the value `768.75` is the largest weight in the sample and is used to normalize the `e.GetWeight()` value to a probability. For Category 3 and Category 4 more `StartX` and `StartY` data is also stored:

```
[[e.GetWeight()/768.75,e.GetStartX(),e.GetStartY(),e.GetStartZ(),
e.GetPx(), e.GetPy(), e.GetPz()]]
```

Category 1 and Category 2 will be the training data for generative networks with `input_dims = 4`, while Category 3 and Category 4 will be for networks with `input_dims = 6`.

Thomas' sample is split over 66 files. To increase the speed of this pre-processing step data is pulled from all of these [1]. From each file a separate array is saved containing information on the minimum and maximum values seen for each muon parameter in each file. Thomas' full sample is currently run over until ~ 1.8 million muons are obtained in each Category. The distributions for muon parameters for Category 4 can be seen in figure 1.

The data is then combined into single `.npy` files and the global minimum and maximum values are found and also stored. The global minimum and maximum values are used to normalize the distributions between values of -1 and 1 (input for GAN), the VAE script takes normalised values between 0 and 1 but this is dealt with at the start of the script.

The `StartX` and `StartY` distributions in Category 3 and Category 4 are very sharply peaked around a single value, this proves difficult to model for these generative networks. To avoid this another step of pre-processing is completed to broaden these distributions. A mean value is found and subtracted from all values in each distribution, the absolute value of every point is then square rooted and correct sign is then applied. The mean value is then added back to every value and the whole distribution is again normalized. Appropriate secondary minimum, maximum values and mean values used are saved for the post-generation conversion from output to real values. In the initialization of each GAN and VAE the first dimension of these training data arrays containing weight information is split off. This is to feed into `numpy.random.choice` function which is used for producing each mini-batch training sample from the full training data.

[1] some of the 66 files are blacklisted as they contain rouge muons originating from a bad charm production file.

3 Blue Crystal Training Set-up

GAN scripts are stored in organized folder structures in my home directory on Blue Crystal (`/mnt/storage/home/am13743/gans_low_memory/`) in the following structure, where `gan_1` is a copy of `TEMPLATE`:

```
gans_low_memory
├── TEMPLATE
│   ├── submit.sh
│   └── gan_6.py
├── gan_1
├── gan_2
├── gan_3
├── ...
├── gan_n
└── run_jobs.sh
```

Large groups of jobs testing various combinations of hyper parameters can be set up and run with shell scripts from these directories.

The training data is stored in `/mnt/storage/scratch/am13743/gan_training_data/`. The output directory is set to `(/mnt/storage/scratch/am13743/low_memory_gan_out/%d/test_output)%file_number` where `file_number` is set corresponding to the test number.

This set up is mimicked for VAE training.

4 Figure of Merit and Training Output

A function is set up in both GAN and VAE scripts to be called every `n` training steps to check progress of the network output. Histograms of real and generated values for each muon parameter are generated see figure 1, along with two dimensional histograms showing the correlations between each unique pair of muon parameters, see figure 3. Alignment of these plots show successful training of the network.

An accurate figure of merit (FOM) is needed to track progress and network accuracy. The approach taken is at every training step to train a classifier. Boosted decision tree (BDT) was chosen because it is fast to train and simple to implement. A BDT is trained on a selected training sample of 50k real and generated muons. Then feed the BDT 50k different real and generated muons and plot the output [3]. The measure of the goodness of the network then is the reduced χ^2 or the overlap between the BDT output distribution from the generated muons and that of the real muons. The BDT acts to systematically reduce the dimensionalality of the higher dimensional muon parameter space into just 1 dimension, see figure 4. Traditional goodness of fit methods such as χ^2 break down in high dimensions when data is sparse and almost all bins have low or no occupancy. These methods work well on this reduced space. The reduced χ^2 and the overlap between generated and real distributions is watched and training is stopped when improvement plateaus.

The BDT used currently is `sklearn.GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, max_depth=4)`. If the `max_depth` parameter is too low the representation of the real and generated samples on the 1D BDT output space is discontinuous. `max_depth = 4` was chosen to have as continuous as possible distributions for fastest training time.

5 GAN Structure

Generative adversarial networks (GANs) employ two neural networks, the generator and the discriminator [2]. The generator has an input of latent space and outputs muon 'images'. The discriminator learns to distinguish between generator output images (fake images) and real training images. The generator learns from tricking the discriminator. The discriminator and generator are iteratively trained as to both become more powerful together, learning form each others output. The goal of the GAN is to produce realistic muon images.

As with all GANs, generator (G) and discriminator (D) models are stacked into one `stacked_generator_discriminator` model inside which the weights of D are frozen (`D.trainable = False`).

The GAN has the G and D models both having inverted pyramid structure with the number of nodes increasing deeper into the network, the base number of nodes is the same. Every layer is a dense fully connected layer. The number of nodes is structured as follows, in order of depth in the layer: 256*n, 512*n, 1024*n, 2048*n (where n is the node factor, `-nG` or `-nD` for G and D respectively, editable with `argparse`). This inverted pyramid structure is not at all required. I will experiment with a flat structure soon.

G has a latent space input of size (n,100) where n is the number of 'images' to generate. The value of 100 is picked arbitrarily.

Two loss functions are used in the GAN, standard `binary_crossentropy` (measuring the difference between input and output labels) from `Keras` and `_loss_generator` which is user defined as:

```
K.mean(-(K.log(y_pred)), axis=-1)
```

where `y_pred` is the output of D.

D is complied with the `binary_crossentropy` loss function, D sees the input labels and output labels are tested against this. Loss function is high if the output of D does not accurately match the labels. G and the `stacked_generator_discriminator` are complied with `_loss_generator`, although note G is never directly trained. Labels throughout the GAN are such that if `stacked_generator_discriminator` is outputting a high mean label the `_loss_generator` loss function is lower. In this setup real data is labeled 1 and generated data is labeled 0.

The training order is as followed:

- Fake images are sampled from G, these are `synthetic_images`.
- D is trained to distinguish `synthetic_images` from training sample (`legit_images`).
- `stacked_generator_discriminator` is trained on normal latent noise which is miss-labeled as true images (`label = 1`). Back propagation works here to reduce the loss function `_loss_generator`. This is minimal if the mean discriminator output is closer to 1, which is equivalent to more generated images staying mislabeled as true images. D is frozen at this point so the weights of G are updated to minimize this loss function.
- Repeat, this time however the fake images from G will be slightly improved.

Two techniques are employed to prevent over fitting and mode collapse. Mode collapse is a common failure in GANs, where the generator locks onto one particularly successful output image which is able to constantly trick the discriminator, this single output acts like a local minimum. Firstly the labels on both `legit_images` and `synthetic_images` that are seen by D are blurred with Gaussian noise of $\mu = 0$ and $\sigma = 0.3$. Secondly, the true values for muon parameters in the training sample are blurred with a very small amount of Gaussian noise ($\mu = 0$ and $\sigma = 0.001$) every time a mini-batch sample is created. Testing will be undertaken to understand the effectiveness of these approaches, this will occur when a final structure for the GAN is settled on.

6 VAE Structure

VAE structure has the encoder and decoder built up of dense fully connected layers. The encoder has the goal of transferring input muon 'images' into a latent space. This latent space is of lower dimensions. Via a KL-divergence term in the loss function the encoder is forced into molding the full samples' representation in latent space into an uncorrelated multivariate Gaussian. This latent space representation is then fed to the decoder which learns to transfer it back into real space, recreating the samples originally fed to the encoder. The requirement for reconstruction of the same samples means the encoder has a complex job of creating a latent space that makes sense to the decoder.

Forcing the full sample to have an uncorrelated multivariate Gaussian representation in latent space is what makes generation of new muons possible. The KL-divergence term encourages these Gaussian distributions (one in each dimension of latent space) to have $\mu = 0$ and $\sigma = 1$ [1]. Generating data then only requires sampling simple $\mu = 0$ and $\sigma = 1$ normal distributions to create latent samples and running these through the decoder network.

The loss function for the VAE (`vae_loss`) is:

```
xent_loss = original_dim * objectives.binary_crossentropy(x, x_decoded_mean)
kl_loss = - 0.5 * K.sum(1 + z_log_var - K.square(z_mean) - K.exp(z_log_var), axis=-1)
vae_loss = xent_loss*xent_factor + kl_loss
```

`xent_loss` is the standard binary crossentropy loss, this is high if the output images from the decoder are far from the input images fed to the encoder. The `kl_loss` term is high if distributions of samples in each dimensions of latent space are far from unit Gaussians. `xent_factor` refers to a weighting. Without the `xent_factor` value, variations in the `kl_loss` term dominate the progress of the loss function and the output of the VAE is nonsense.

The training of the VAE is simple, it is fed the training sample which is labeled with itself:

```
vae.train_on_batch(legit_images, legit_images)
```

7 GAN Training

The GAN script is built with `argparse` allowing easy change of hyper parameters, controlling GAN architecture and training.

The GAN takes the following options:

- -l, learning rate, default set to 0.0002
- -o, optimizer choice, default set to `Adam` with `amsgrad=True`
- -nG, node factor, default set to 1
- -nD, node factor, default set to 1
- -layersG, number of layers in generator, default set to 3
- -layersD, number of layers in discriminator, default set to 3

Preliminary testing proved learning was most stable with default optimizer `Adam` with `amsgrad=True` instead of alternatives of `Adam` without `amsgrad=True` and `RMSprop`. So for testing since then I have focused on default optimizer. A lower learning rate increases stability of training, a lower than default value of 0.00005 was found in preliminary testing to be stable whilst not making training too slow.

The `-layersG` and `-layersD` parameters control the numbers of layers in each network. These are found to be optimal when they are both set equal to 2. Combinations of values were checked between 1 to 4 for each parameter.

The `-nG` and `-nD` hyper parameters controls the number of nodes in each layer of corresponding networks. Performance appears to be improved when `-nG`>`-nD`, optimum appears to be when `-nG = 6` and `-nD = 3` for the 6 dimensional GAN case (training on Category 3 and Category 4 from section 2).

Batch size is also important for training. This parameter controls the difference between gradient descent (GD), stochastic GD and min-batch GD. It was found in preliminary testing that a value of ~ 100 was good, 1000 being too high and 10 too low.

8 VAE Training

The VAE script is also built with `argparse`.

The VAE takes the following options:

- -lr, learning rate, default set at 0.001 (higher than GAN)
- -x, cross-entropy factor, default set at 1000
- -iE, intermediate dimensions (nodes) in every layer of encoder (E), default set to 100
- -iD, intermediate dimensions (nodes) in every layer of decoder (D), default set to 100
- -layersE, number of layers in E, default set to 3
- -layersD, number of layers in D, default set to 3

the choice of value for the learning rate is very dependent on the optimizer chosen, currently running with a high learning rate value for the `Adam` optimizer but this needs investigation.

The number of dimensions in latent space is variable, in preliminary tests it was found if this value was too small (~ 2), output was worse, probably too much information was lost and VAE found it harder to produce good output from decoder. Currently the number of dimensions in latent space is set to 4.

The cross-entropy factor `-x` as discussed in section 6 this should be high, ~ 1000.

The number of nodes in each layer of each network needs to be optimized.

The number of layers in each network needs to be optimized.

9 GAN Output

Follows is currently the best achieved output from a 6 input dimension GAN, this was trained on Category 4 (from section 2) training data. The values for relevant hyperparameters were:

```
-nG = 6
-nD = 2
-layersG = 2
-layersD = 2
```

this is the output after 2,710,000 training steps, ~ 30 hours of training on Blue Crystal.

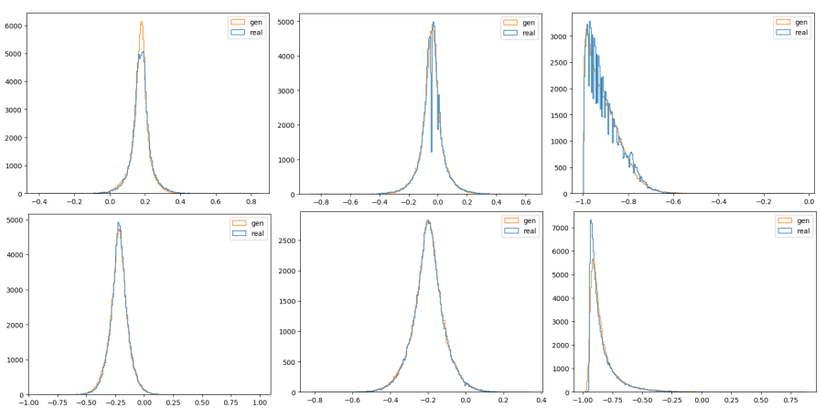

Figure 1: One dimensional histograms of output for each the 6 muon parameters overlayed against training sample. Distributions 0 and 1 here are displayed post-broadening as they are fed to the GAN.

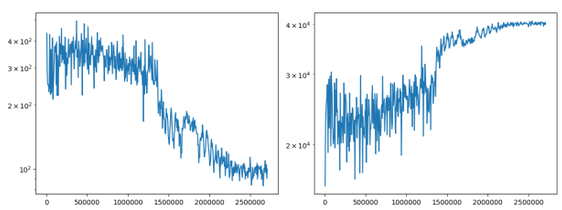

Figure 2: Left: progress of reduced χ^2 estimator for the distance between distributions in figure 4. Right: progress of overlap between distributions in figure 4.

I have begun optimization testing on 4 dimension GANs, in theory this should be an easier optimization.

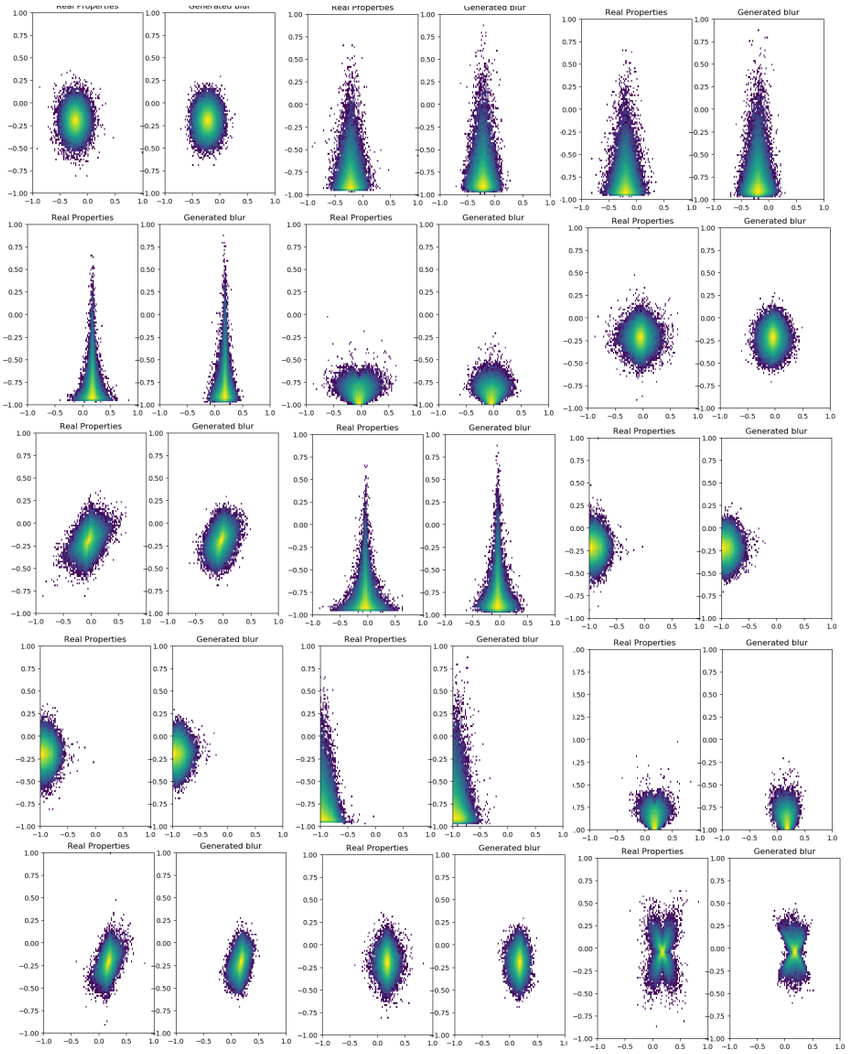

Figure 3: Two dimensional histograms displaying the correlations between each unique pair of the 6 muon parameters, generated and training samples are paired up.

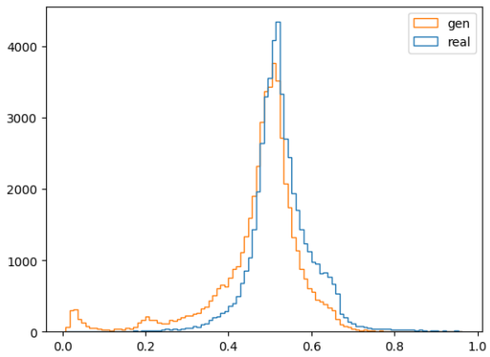

Figure 4: Output distributions of BDT for both fake and real images.

10 VAE Output

Do not yet have an optimized VAE for 4 dimensions or 6 dimensions.

11 Generation and Benchmarking

Generation will require 4 fully trained and optimized networks. When this achieved generation will be completed in accordance to the ratios of appearance of muons in the full sample using Poisson sampling.

If the a high fraction of events in a single category are weighted up this could cause a problem. You generate based on weights of full samples to avoid this?

Benchmarking must include full I/O to ROOT files. A technique must be developed to make this as fast as possible.

References

[1] Carl Doersch. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*, 2016.

[2] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[3] Constantin Weisser and Mike Williams. Machine learning and multivariate goodness of fit. *arXiv preprint arXiv:1612.07186*, 2016.