Introduction to Generative Adversarial Networks (GANs).

April 29th 2019, Deep Learning and GPU Computing Seminar, DESY

Torben Ferber (torben.ferber@desy.de)

HELMHOLTZ RESEARCH FOR GRAND CHALLENGES

CLUSTER OF EXCELLENCE QUANTUM UNIVERSE

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

DESY.

Overview

- Introduction to GANs
- GAN hell: Vanishing gradients and mode collapse
- WGAN-GP in PyTorch
- GAN metrics: How to measure performance
- GAN hacks
- Real examples

Introduction to GANs

- **Goal: “synthesise artificial samples, such as images, that are indistinguishable from authentic samples”**

- Generative modelling:
  - observe samples to generate more samples
  - infer density function that describes data
- Several different generative models, GANs are one of them
- GANs introduced to ML community in 2014 by Goodfellow et al. [1], first ideas are much older

[1] GANs (https://arxiv.org/abs/1406.2661)

Edmond de Belamy ($432,5000)

https://thispersondoesnotexist.com/

StyleGAN (https://arxiv.org/abs/1812.04948)

Generative models: Why?

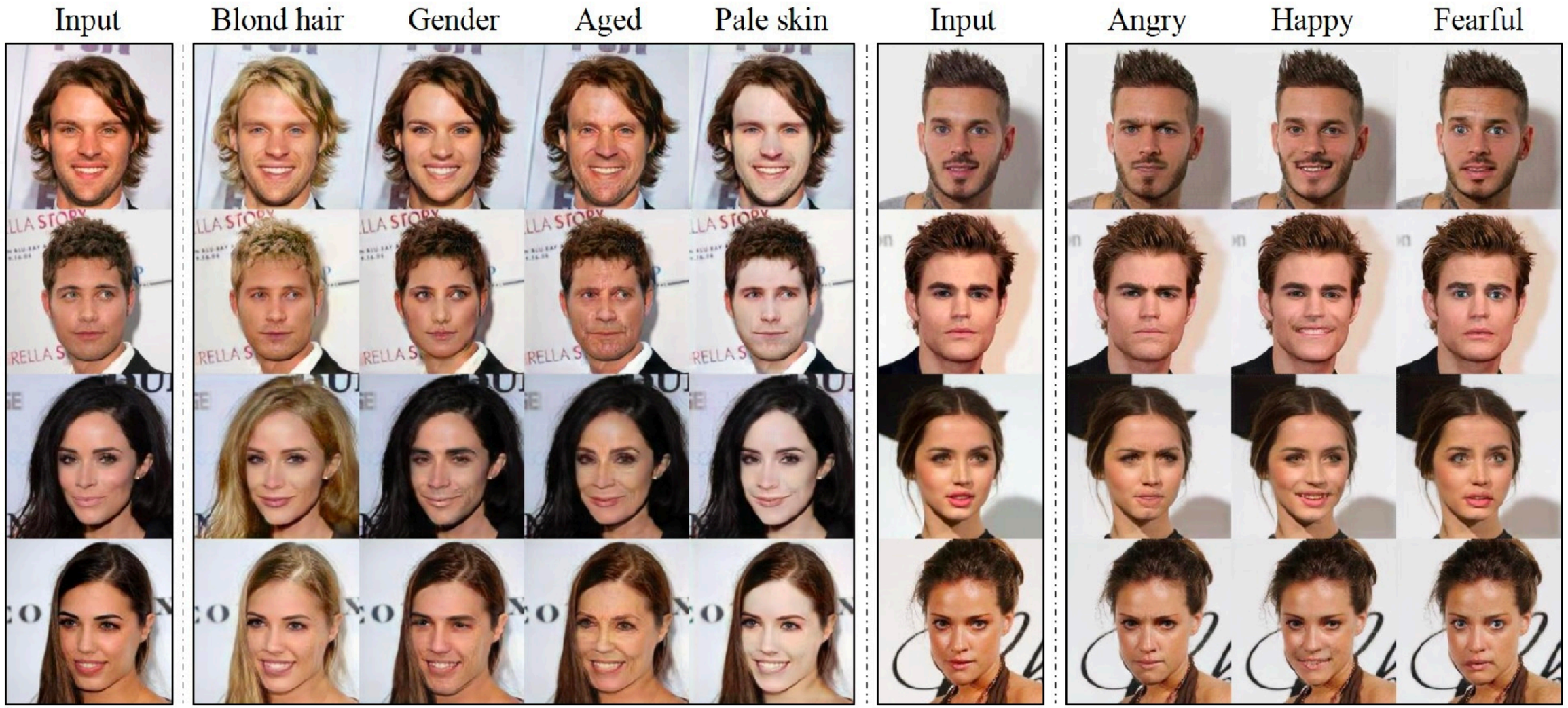
- Generate simulated environments (e.g. for RL)
- Infer possible future states
- Recover missing information
- Multi-modal outputs (= many possible outputs)
- Image-to-image translation
- Cross-domain: Text-to-image
- ...

Generative face completion

(a) Original face (b) Masked input (c) Our result

[1] http://openaccess.thecvf.com/content_cvpr_2017/papers/Li_Generative_Face_Completion_CVPR_2017_paper.pdf

Multi-domain image-to-image translation



[1] StarGAN (https://arxiv.org/abs/1711.09020)

GANs in words

- Two different models that are adversarial of each other (game theory):

  Each model tries to get the highest possible payoff

  - Player 1: Generator (G) that tries to capture a training data distribution
  - Player 2: Discriminator (D) that tries to identify if data came from the training data or from the generator

- Ideal minimax two player game if G recovers training data and D=0.5 or in other words: If the two players reach their Nash equilibrium

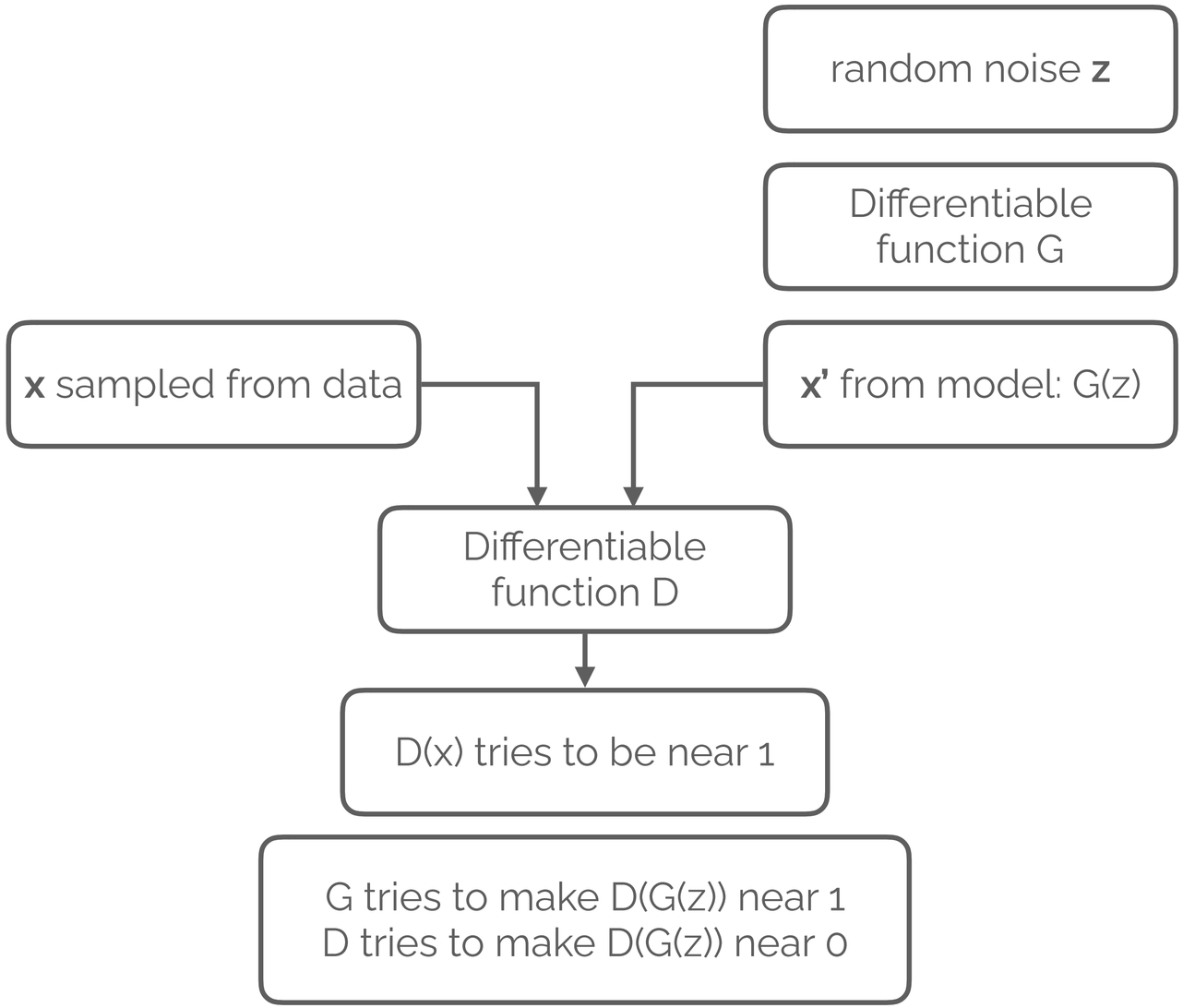
Introduction to GANs: Nash equilibrium

- Nash equilibrium first formulated by John Nash in the 1950s in the context of game theory:
  - Both players have chosen a strategy
  - No player can benefit by changing the strategy if the other player sticks to their current strategy

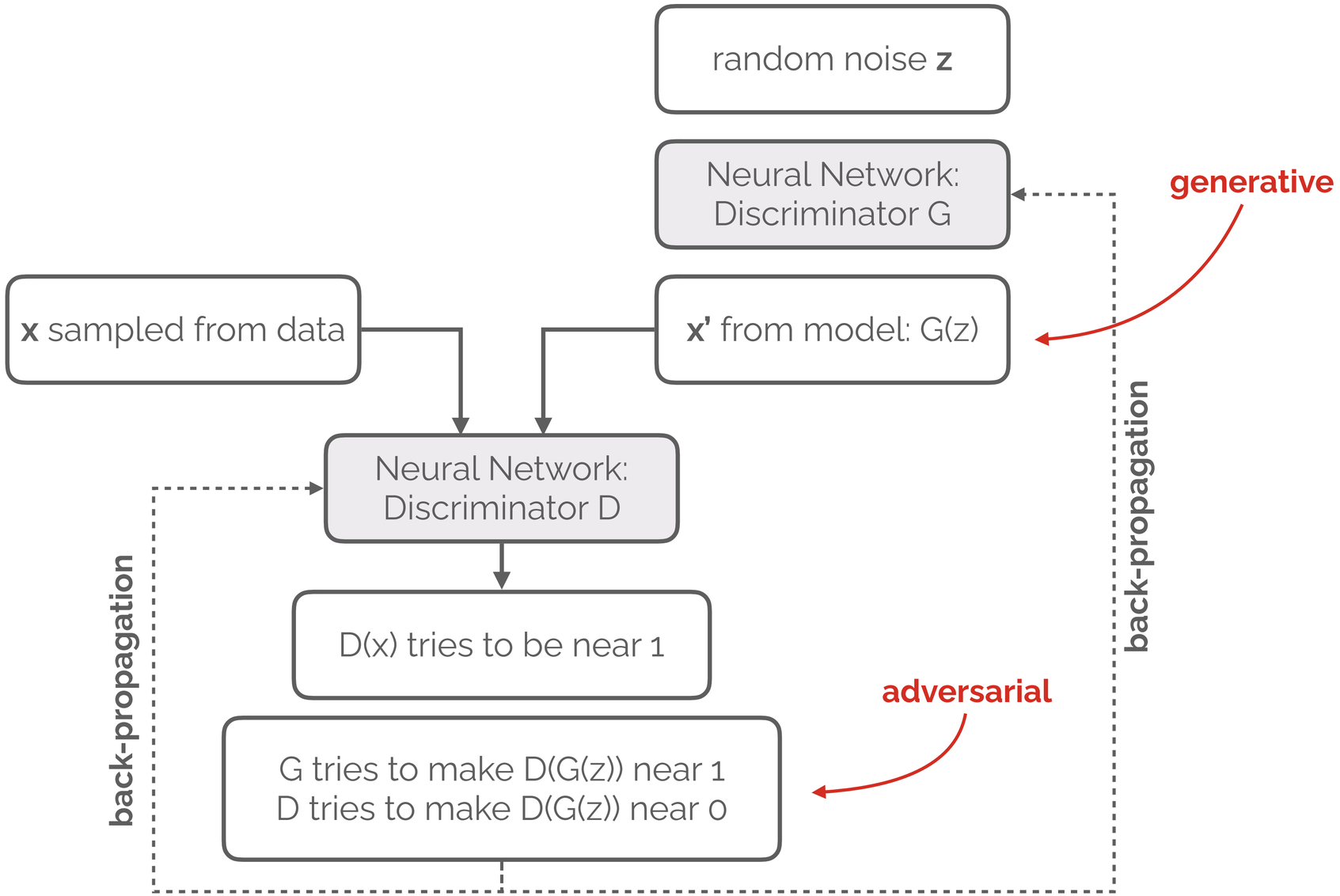
    → Both strategies and payoffs constitute a Nash equilibrium
- In other words: A change in strategy will lead to a worse result for this player

THE example: “Prisoners dilemma” (https://en.wikipedia.org/wiki/Prisoner%27s_dilemma)

Introduction to GANs: Schematic



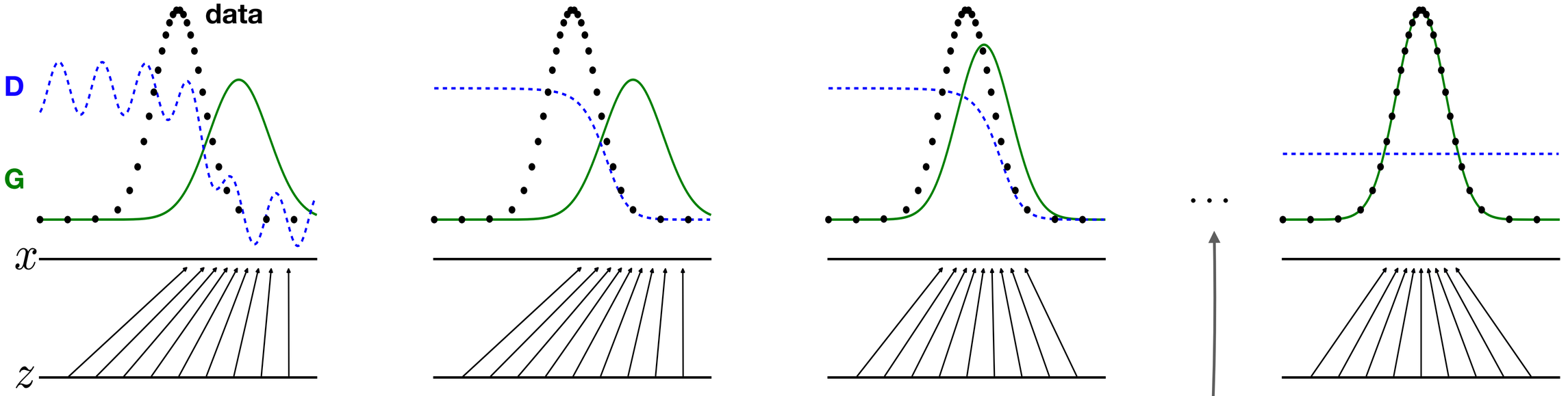
Introduction to GANs: Schematic



Introduction to GANs: Schematics

http://billhooverart.com

Introduction to GANs: Training process



$p_g \sim p_{data}$,
D is partially accurate

Update D:
D is trained until
$D^* = p_{data} / (p_{data} + p_g)$

Update G:
Gradient of D has guided G to move towards data-like regions

Nash equilibrium!
D(x) = 0.5

Repeat many times

[1] GANs (https://arxiv.org/abs/1406.2661)

GANs: Loss function

- Objective function for the discriminator D is given by the cross entropy of a binary classifier (D(x)=1 for data and =0 for fake):

$$\max_D V(D) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

- Objective function for the generator G:

$$\min_G V(G) = \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

- Minimax game value function:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

end for

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

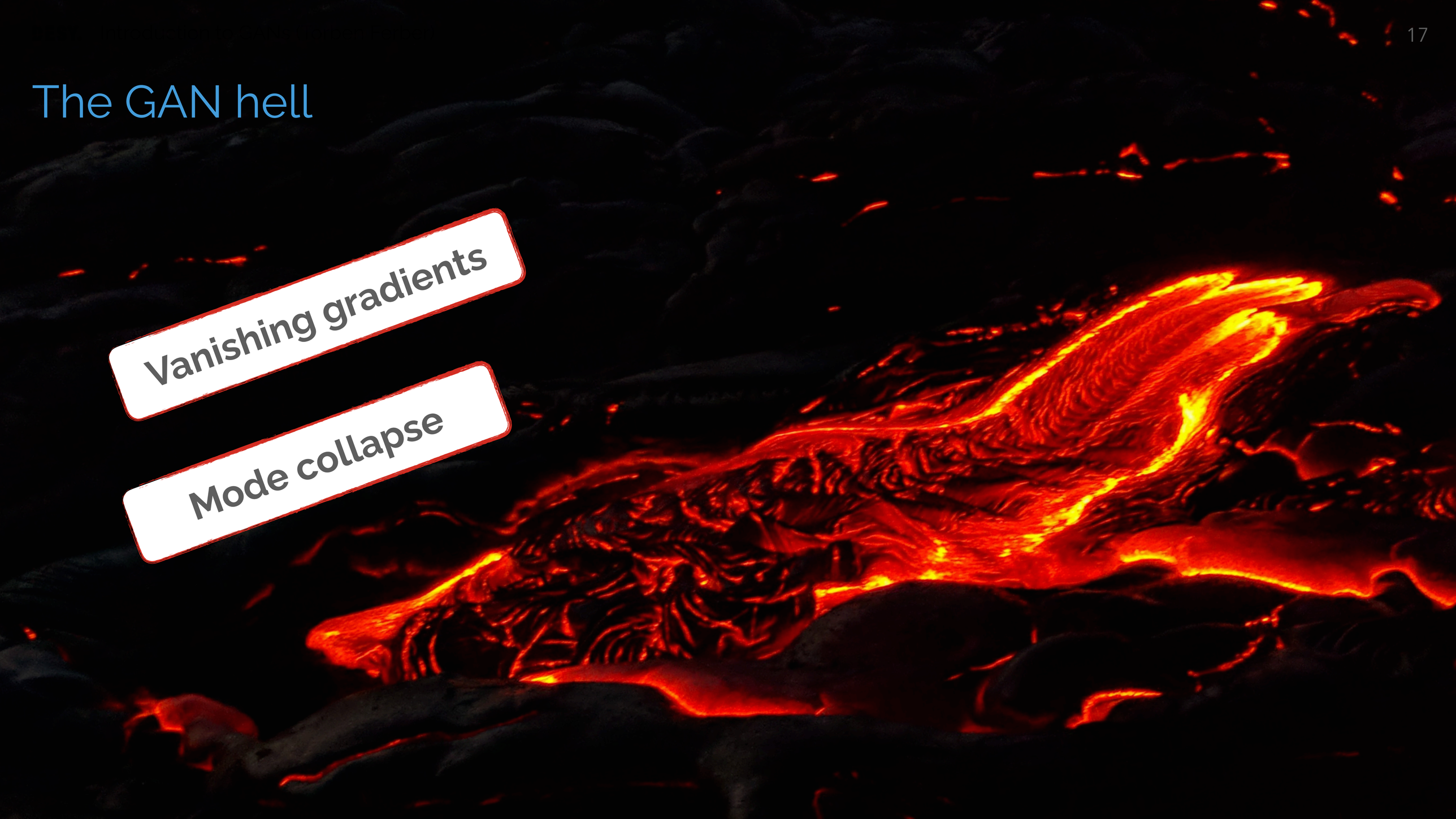
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

[1] GANs (https://arxiv.org/abs/1406.2661)

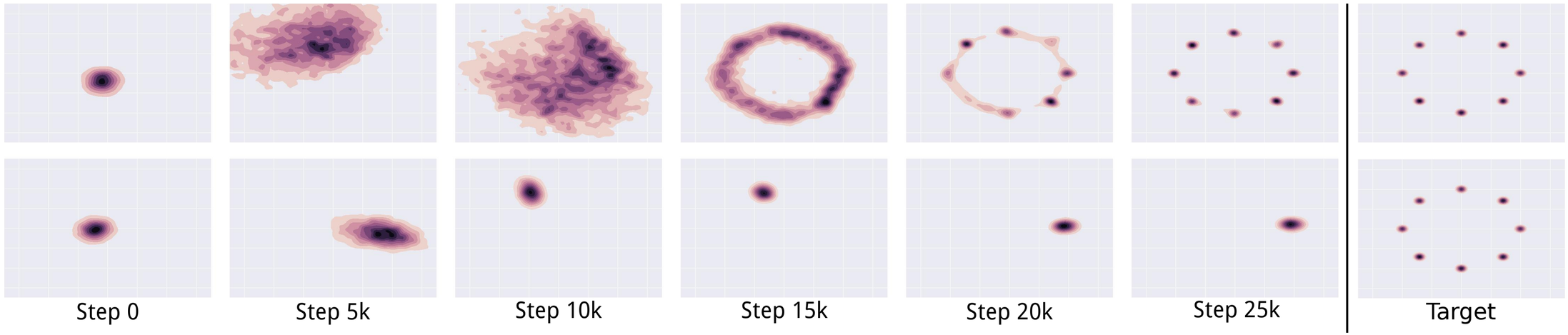
The GAN hell



MinMax or MaxMin: Mode collapse

- $\min_G \max_D V(G, D)$:
  - Good! Optimally the network will converge to $p_{gen} = p_{data}$
- $\max_D \min_G V(G, D)$:
  - Bad! Generator will converge to a point that is currently considered real rather than fake
  - If D catches that, G often just moves to the next node instead of the full distribution
- Network design does not really specify minmax over maxmin: Mode collapse

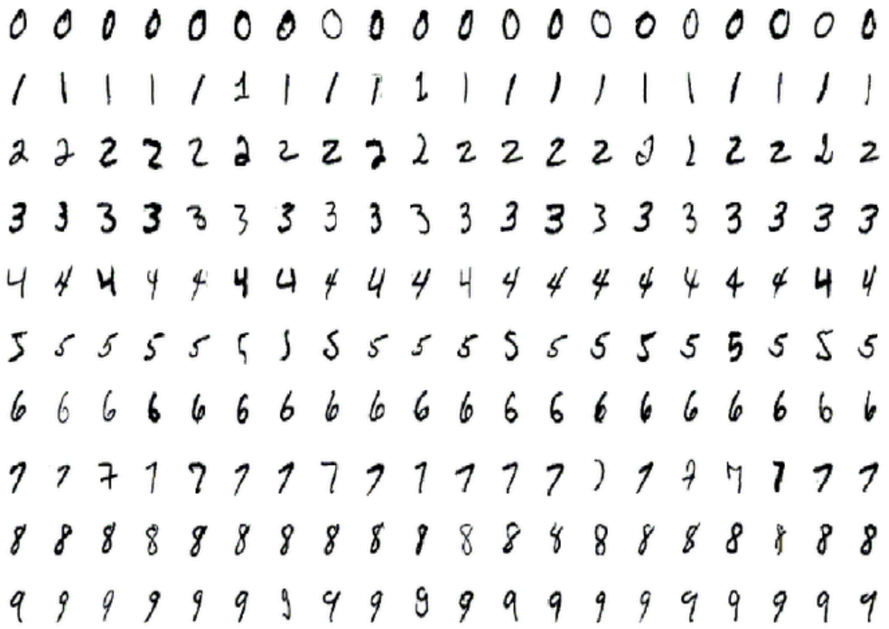
Mode collapse



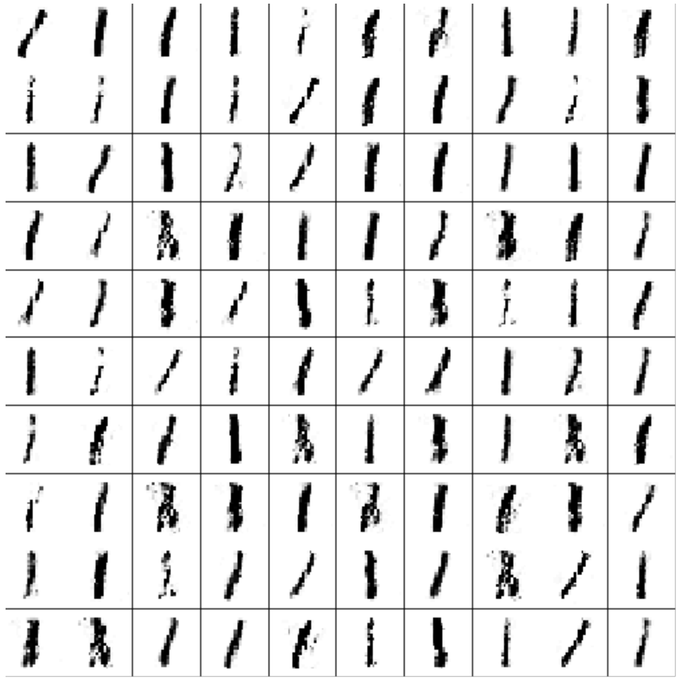
[1] arxiv:1611.02163

Mode collapse

data



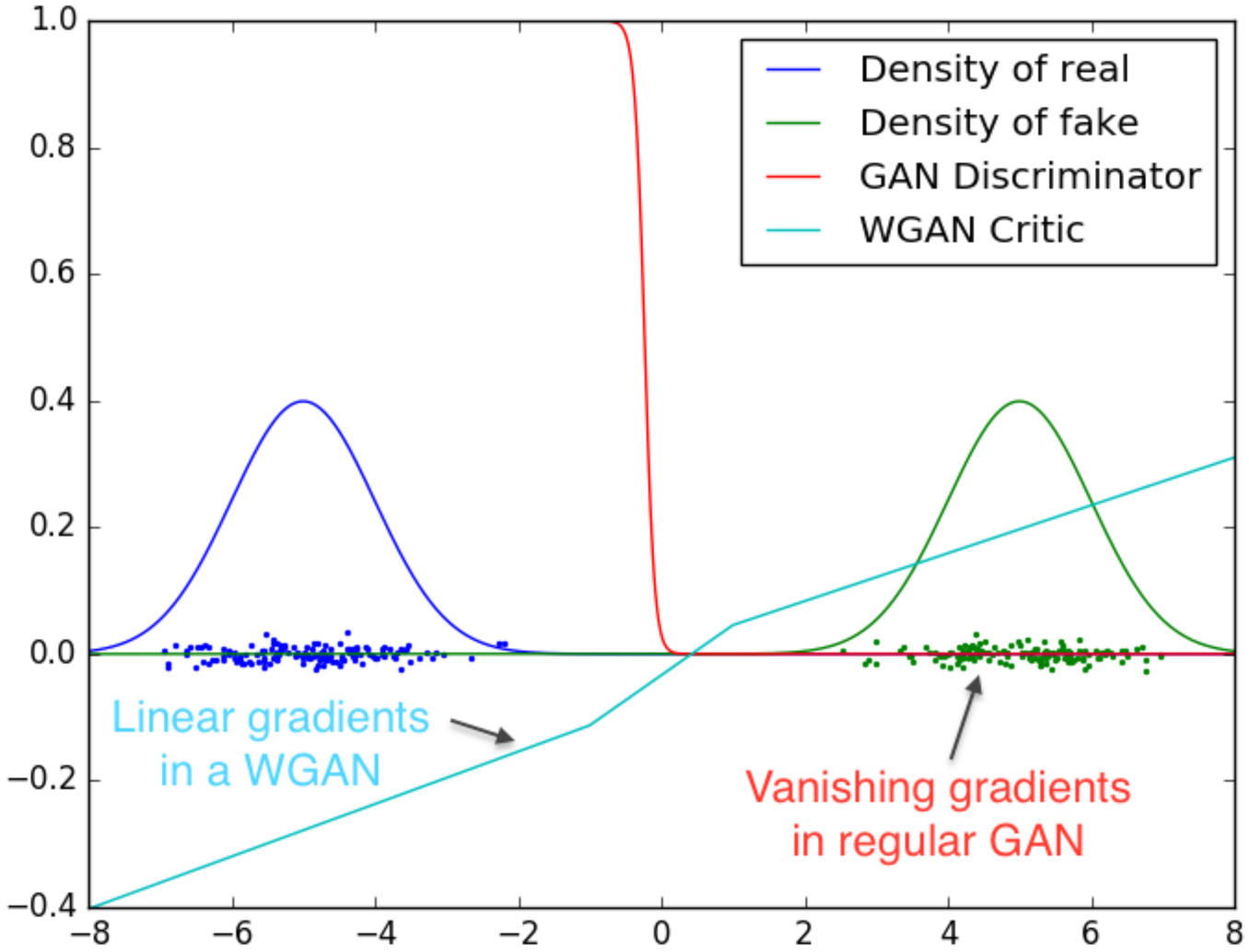
generated



Vanishing gradients: Modified loss function

- Usually D has a much easier start than G, and D can reject all generated samples as fake:
  - $\log(1-D(G(z)) \rightarrow 0$
- Slightly modified Minimax game:
  - Instead of training G to minimising $\log(1-D(G(z))$,
    train G to maximise $\log(D(G(z))$
- Math behind this: GANs are minimizing the Jenson-Shannon (JS) divergence, which saturates if data distribution **p** and fake distribution **q** do not overlap

Vanishing gradients: Wasserstein distance



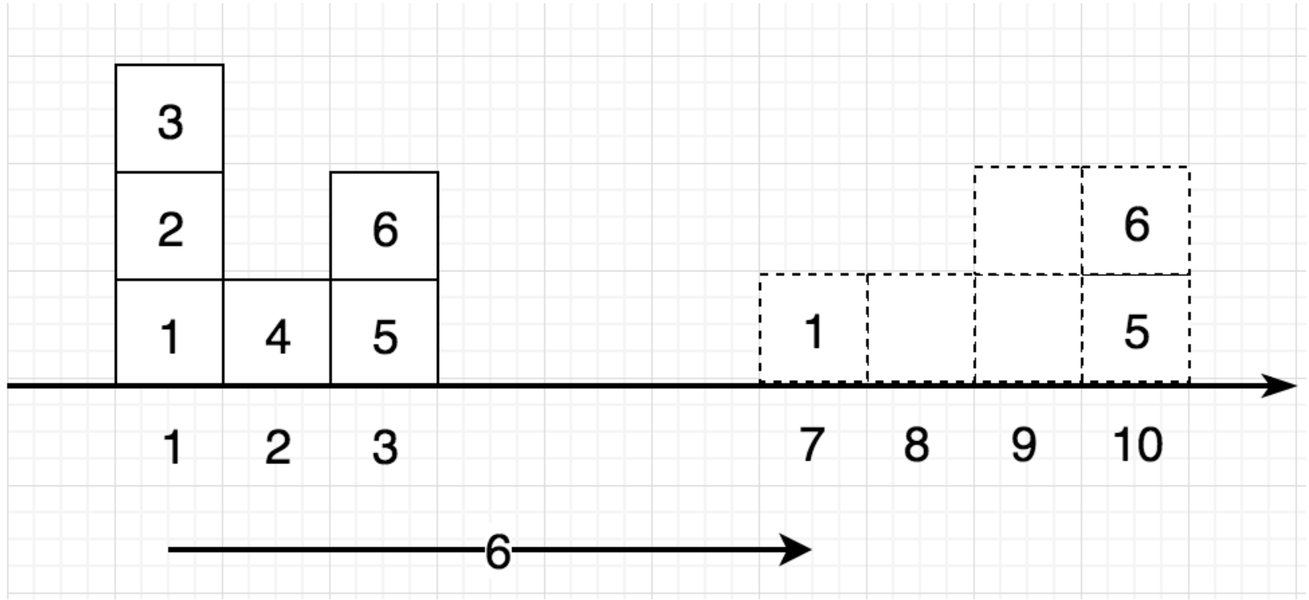
Cross-Entropy minus self entropy
→ Jenson-Shannon (JS) divergence

Maximum likelihood
→ Kullbach-Leibler (KS) divergence

[1] WGANs 1701.07875

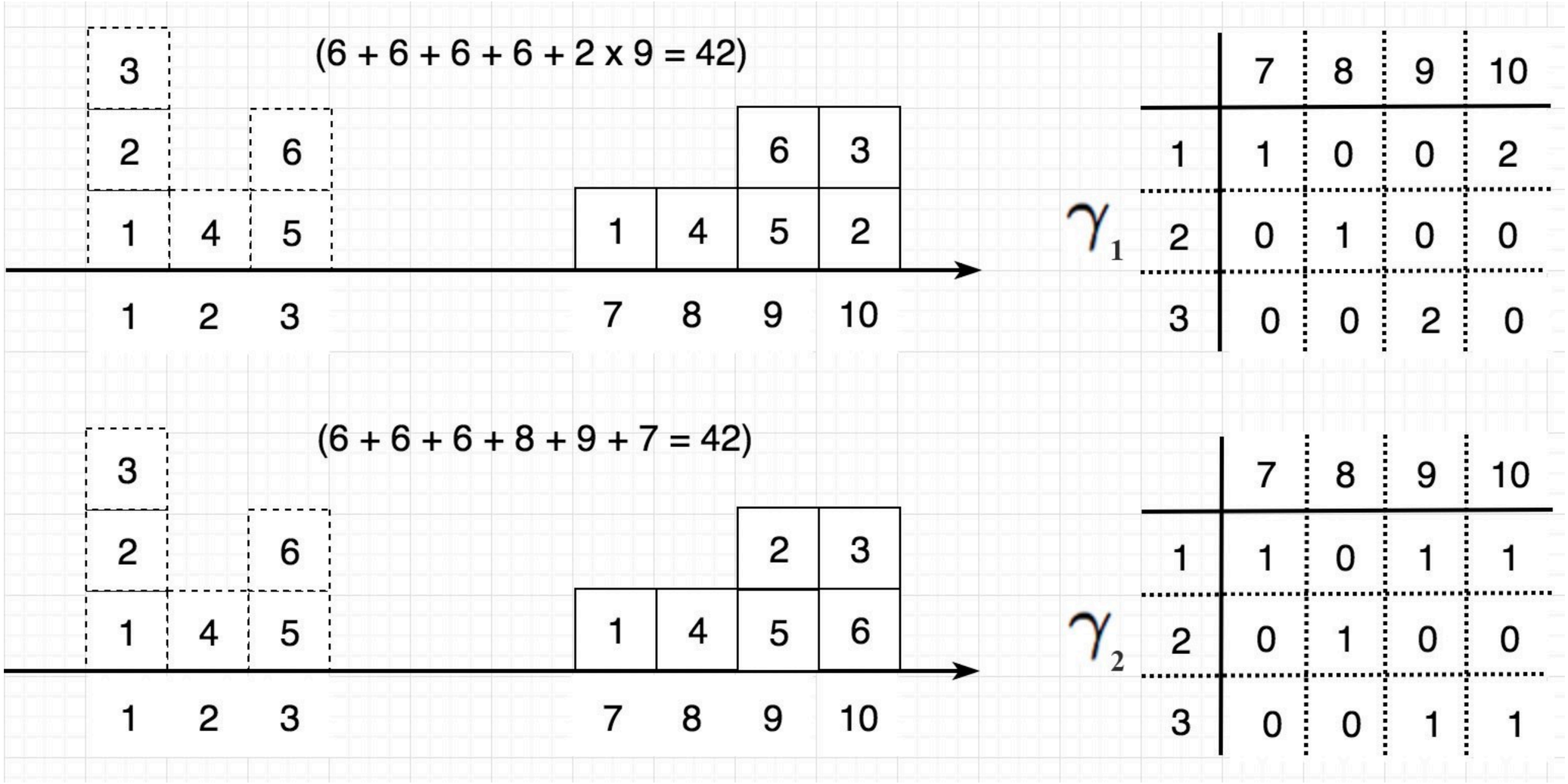
Vanishing gradients: Wasserstein distance

- Given the problems (vanishing gradients, mode collapse) of Vanilla GANs, is there a better distance to measure the difference between real and fake?



[1] https://medium.com/@jonathan_hui/gan-wasserstein-gan-wgan-gp-6a1a2aa1b490

Vanishing gradients: Wasserstein distance



+ many more possibilities

Vanishing gradients: Wasserstein distance

- EMD is the infimum (greatest lower bound) of all transport planes:

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma} \left[\|x - y\| \right]$$

- Using the Kantorovich-Rubinstein duality →

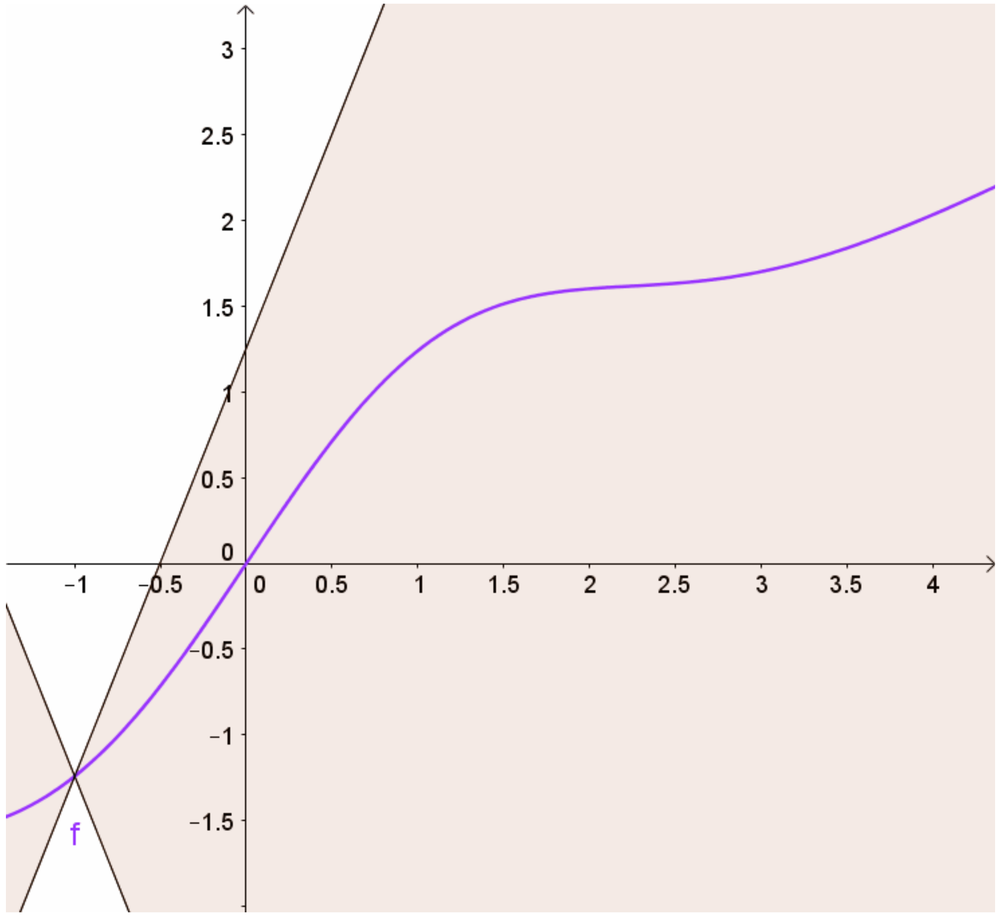
$$W(\mathbb{P}_r, \mathbb{P}_\theta) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x\sim\mathbb{P}_r}[f(x)] - \mathbb{E}_{x\sim\mathbb{P}_\theta}[f(x)]$$

  where f(x) is a 1-Lifschitz function with |f(x1)-f(x2)| ≤ |x1 - x2|.

- A neural net can learn the function f(x) with two restrictions:
  - We have to make sure that f(x) is (almost) 1-Lifschitz.
  - We have to train the critic until (almost) convergence.

[1] WGANs (arXiv:1701.07875v3)

Vanishing gradients: Wasserstein distance



[1] https://de.wikipedia.org/wiki/Lipschitz-Stetigkeit

Vanishing gradients: Wasserstein distance

- Weight clipping (original WGAN):
  - Limit weights to **-c < w < c**.

Clipping value c is hyperparameter:
Very hard to optimize

- Gradient penalty (WGAN-GP, Improved WGAN):
  "A differentiable function f is 1-Lipschitz if and only if it has gradients with norm at most 1 everywhere." → penalty term in loss function if norm > 1

Computationally expensive
Do not use batch norm in critic

$$L = \underbrace{\mathop{\mathbb{E}}_{\tilde{\boldsymbol{x}}\sim\mathbb{P}_g}\left[D(\tilde{\boldsymbol{x}})\right] - \mathop{\mathbb{E}}_{\boldsymbol{x}\sim\mathbb{P}_r}\left[D(\boldsymbol{x})\right]}_{\text{Original critic loss}} + \underbrace{\lambda \mathop{\mathbb{E}}_{\hat{\boldsymbol{x}}\sim\mathbb{P}_{\hat{\boldsymbol{x}}}}\left[\left(\|\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})\|_2 - 1\right)^2\right]}_{\text{Our gradient penalty}}.$$

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.

Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.

while θ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

[1] WGANs (arXiv:1701.07875v3)

Algorithm 1 WGAN with gradient penalty. We use default values of $\lambda = 10$, $n_{\text{critic}} = 5$, $\alpha = 0.0001$, $\beta_1 = 0$, $\beta_2 = 0.9$.

Require: The gradient penalty coefficient λ, the number of critic iterations per generator iteration n_{critic}, the batch size m, Adam hyperparameters α, β_1, β_2.
Require: initial critic parameters w_0, initial generator parameters θ_0.

while θ has not converged **do**
  for $t = 1, ..., n_{\text{critic}}$ **do**
    for $i = 1, ..., m$ **do**
      Sample real data $\boldsymbol{x} \sim \mathbb{P}_r$, latent variable $\boldsymbol{z} \sim p(\boldsymbol{z})$, a random number $\epsilon \sim U[0, 1]$.
      $\tilde{\boldsymbol{x}} \leftarrow G_\theta(\boldsymbol{z})$
      $\hat{\boldsymbol{x}} \leftarrow \epsilon \boldsymbol{x} + (1 - \epsilon)\tilde{\boldsymbol{x}}$
      $L^{(i)} \leftarrow D_w(\tilde{\boldsymbol{x}}) - D_w(\boldsymbol{x}) + \lambda(\|\nabla_{\hat{\boldsymbol{x}}} D_w(\hat{\boldsymbol{x}})\|_2 - 1)^2$
    end for
    $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^{m} L^{(i)}, w, \alpha, \beta_1, \beta_2)$
  end for
  Sample a batch of latent variables $\{\boldsymbol{z}^{(i)}\}_{i=1}^{m} \sim p(\boldsymbol{z})$.
  $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} -D_w(G_\theta(\boldsymbol{z})), \theta, \alpha, \beta_1, \beta_2)$
end while

[1] WGAN-GPs (arXiv:1704.00028v3)

WGANs: Schematics

http://billhooverart.com

Other losses?

GAN Type	Key Take-Away
GAN	The original (JSD divergence)
WGAN	EM distance objective
Improved WGAN	No weight clipping on WGAN
LSGAN	L2 loss objective
RWGAN	Relaxed WGAN framework
McGAN	Mean/covariance minimization objective
GMMN	Maximum mean discrepancy objective
MMD GAN	Adversarial kernel to GMMN
Cramer GAN	Cramer distance
Fisher GAN	Chi-square objective
EBGAN	Autoencoder instead of discriminator
BEGAN	WGAN and EBGAN merged objectives
MAGAN	Dynamic margin on hinge loss from EBGAN

← 2014 Hype

← 2017 Hype

← 2018 Hype

[1] https://towardsdatascience.com/gan-objective-functions-gans-and-their-variations-ad77340bce3c

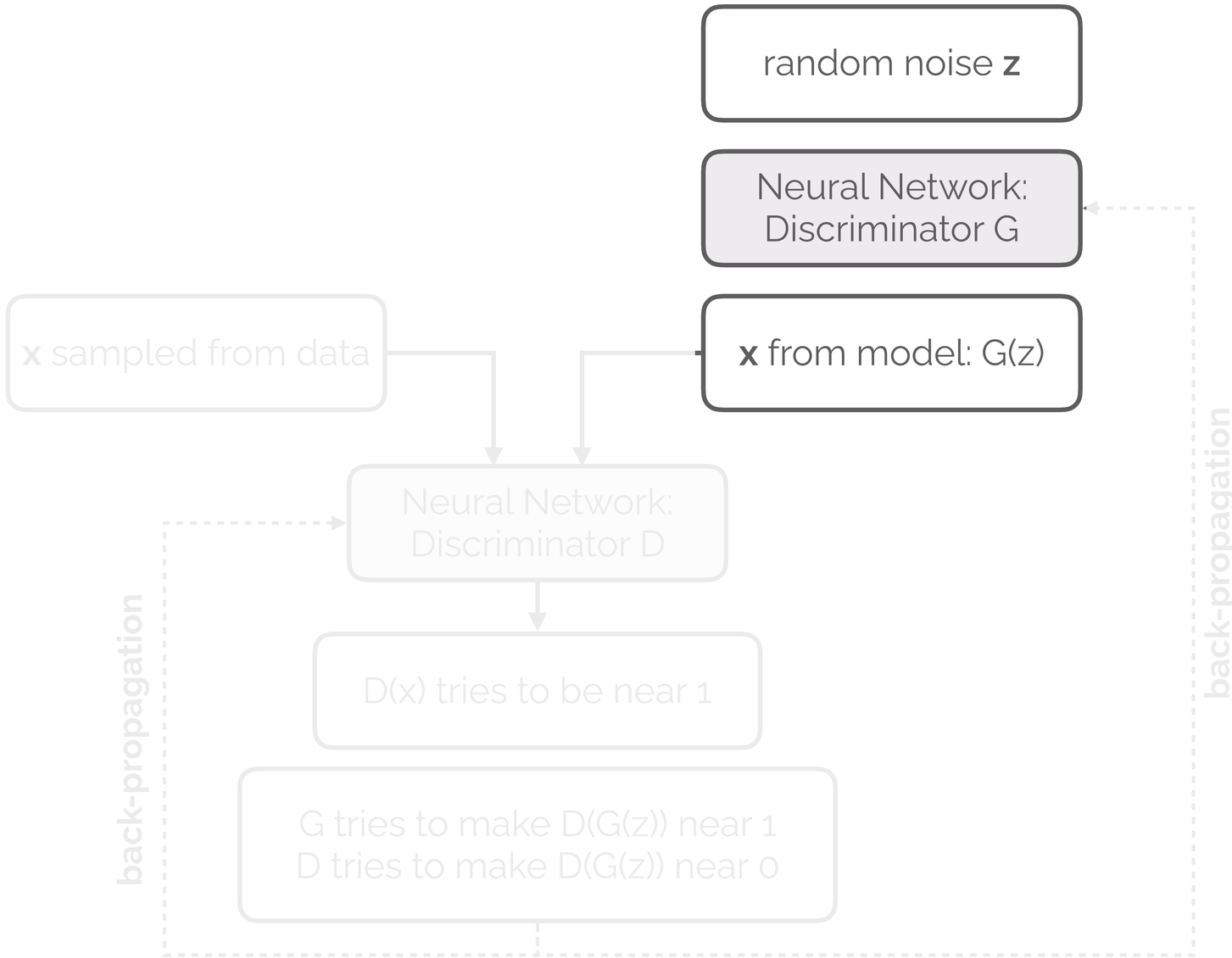
"We find that most models can reach similar scores with enough hyperparameter optimization and random restarts.

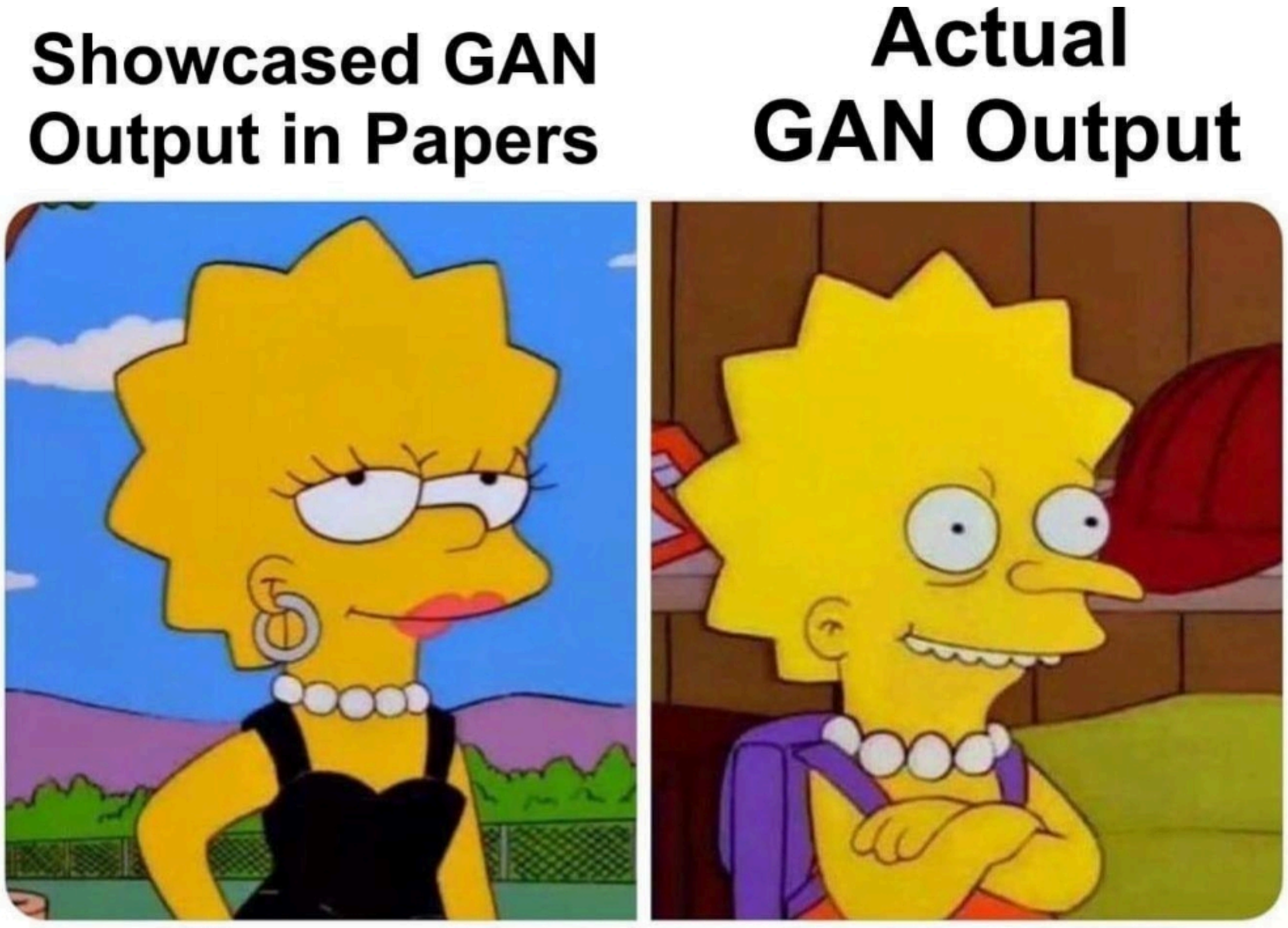
(…) improvements can arise from a higher computational budget and tuning more than fundamental algorithmic changes."

tl; dr: Hyperparameters are more important than loss functions.
(This is heavily debated and probably problem-dependent.)

[1] Google Brain, arXiv:1711.10337v4

GAN Inference: Only needs the generator




Showcased GAN Output in Papers
Actual GAN Output

ML on HTCondor

1) submit.htc

```
Universe = vanilla
Executable = submit.sh
Log    = job.log.$(Cluster)-$(Process)
Error  = job.err.$(Cluster)-$(Process)
Output = job.out.$(Cluster)-$(Process)
RequestMemory = 4096
Request_GPU = 1
Requirements = (OpSysAndVer == "CentOS7")
+RequestRuntime = 18000
notification = Error
Queue 1
```

2) submit.sh

```
#!/bin/bash
source /etc/profile.d/modules.sh
module load anaconda3

WORK_DIR=`mktemp -d`
scp <path_to_my_data_on_pnfs> $WORK_DIR

python <path_to_my_script>/myscript.py -i traindir $WORK_DIR
```

3) myscript.py

```
import matplotlib.pyplot as plt
plt.switch_backend('agg')

import torch
...
```

[1] https://pytorch.org/

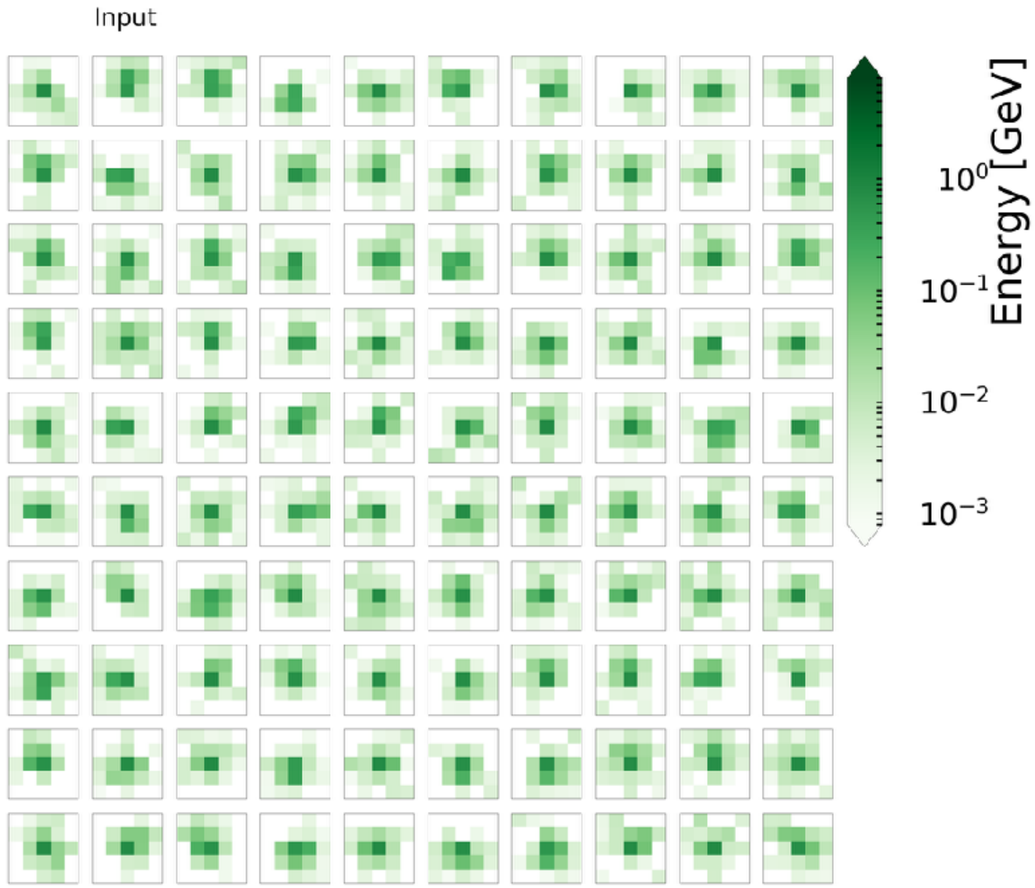
WGAN-GP in PyTorch

Algorithm 1 WGAN with gradient penalty. We use default values of $\lambda = 10$, $n_{\text{critic}} = 5$, $\alpha = 0.0001$, $\beta_1 = 0$, $\beta_2 = 0.9$.

Require: The gradient penalty coefficient λ, the number of critic iterations per generator iteration n_{critic}, the batch size m, Adam hyperparameters α, β_1, β_2.
Require: initial critic parameters w_0, initial generator parameters θ_0.

while θ has not converged **do**
  for $t = 1, ..., n_{\text{critic}}$ **do**
    for $i = 1, ..., m$ **do**
      Sample real data $\boldsymbol{x} \sim \mathbb{P}_r$, latent variable $\boldsymbol{z} \sim p(\boldsymbol{z})$, a random number $\epsilon \sim U[0, 1]$.
      $\tilde{\boldsymbol{x}} \leftarrow G_\theta(\boldsymbol{z})$
      $\hat{\boldsymbol{x}} \leftarrow \epsilon \boldsymbol{x} + (1 - \epsilon)\tilde{\boldsymbol{x}}$
      $L^{(i)} \leftarrow D_w(\tilde{\boldsymbol{x}}) - D_w(\boldsymbol{x}) + \lambda(\|\nabla_{\hat{\boldsymbol{x}}} D_w(\hat{\boldsymbol{x}})\|_2 - 1)^2$
    end for
    $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^{m} L^{(i)}, w, \alpha, \beta_1, \beta_2)$
  end for
  Sample a batch of latent variables $\{\boldsymbol{z}^{(i)}\}_{i=1}^{m} \sim p(\boldsymbol{z})$.
  $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} -D_w(G_\theta(\boldsymbol{z})), \theta, \alpha, \beta_1, \beta_2)$
end while

WGAN-GP in PyTorch: Input → Output



1 GeV photons in the Belle II barrel calorimeter (GEANT4)
Each cell is a 5×5cm CsI(Tl) crystal

WGAN-GP in PyTorch: Sample real data

```
from torch.utils.data import Dataset

class ClusterImage(Dataset):
    """
        Dataset of cluster images.
        """
    def __init__(self,
                 df_x,
                 df_monitor,
                 ):

        self.images = df_x.values.astype(dtype=np.float32)
        self.monitor = df_monitor.values.astype(dtype=np.float32)
        self.y = np.ones(len(self.images))

        # todo: clip to remove outliers
        # todo: scale to [-1, 1] using sklearn MinMaxScaler

    def __len__(self):
        return self.y.shape[0]

    def __getitem__(self, idx):
        return (self.images[idx], self.monitor[idx], self.y[idx])
```

WGAN-GP in PyTorch: Sample real data

```
import torch, uproot, pandas

# read ROOT input file(s)
tree = uproot.open('/pnfs/desy.de/../myfile.root')['mytree']
df = tree.pandas.df(['var1','var2', 'pixel_*'])

# create dataset
dataset = ClusterImage(df_x = df.filter(like='pixel_').fillna(value=0.0),
                       df_m = df[['var1', 'var2']])

# create dataloader
data_loader = torch.utils.data.DataLoader(dataset=dataset,
                                          batch_size=512,
                                          shuffle=True,
                                          num_workers=12) # Number of CPUs
```

Use ROOT without installing #$@&%*!

pip install uproot --user

WGAN-GP in PyTorch: Training

```
import time

N_CRITIC_STEPS = 5

for epoch in range(N_EPOCHS):

    # start timer
    t0 = time.time()

    #loop through batches (no need for class labels right now)
    for batch_number, (images, monitor, y) in enumerate(data_loader):

         if critic_step < N_CRITIC_STEPS:
              # train critic
         else:
              # train generator

    # end timer
    t_epoch = time.time() - t0
    print('time per epoch: %2.2f seconds' % (t_epoch))
```

hyperparameter
Important: Train WGANs until (almost) convergence

WGAN-GP in PyTorch: Networks and optimizers

```
critic_optimizer = torch.optim.RMSprop(critic.parameters(), lr=0.00005)
generator_optimizer = torch.optim.RMSprop(generator.parameters(), lr=0.00005)

if torch.cuda.is_available():
    critic.cuda()
    generator.cuda()
```

GPU

```
critic_fc(
    (layer_module): ModuleList(
         (0): Linear(in_features=25, out_features=50)
         (1): LeakyReLU(negative_slope=0.1)
         (2): Dropout(p=0.05)
         (3): Linear(in_features=50, out_features=50)
         (4): LeakyReLU(negative_slope=0.1)
         (5): Dropout(p=0.05)
         (6): Linear(in_features=50, out_features=25)
         (7): LeakyReLU(negative_slope=0.1)
         (8): Dropout(p=0.05)
         (9): Linear(in_features=25, out_features=1)
         )
    )
```

No sigmoid!

N=10

```
generator_fc(
     (layer_module): ModuleList(
        (0): Linear(in_features=10, out_features=50)
        (1): LeakyReLU(negative_slope=0.1)
        (2): Linear(in_features=50, out_features=25)
        (3): LeakyReLU(negative_slope=0.1)
        (4): Linear(in_features=25, out_features=25)
        (5): Tanh()
        )
     )
```

[-1, 1]

WGAN-GP in PyTorch: Critic and generator training

```
critic.zero_grad()

# clamp weights
# for p in critic.parameters():
#     p.data.clamp_(-C, C)

# prediction on data
pred_real = torch.mean(critic(images))

# get latent noise vector
z = to_var(torch.randn(batch_size, N_Z))

# generate images
fake_images = generator(z)

# prediction on fake data
pred_fake = torch.mean(critic(fake_images))

# gradient penalty
gradient_penalty = calc_gradient_penalty(critic,
                                         images.data,
                                         fake_images.data,
                                         batch_size,
                                         LAMBDA)

# loss and back propagation
critic_loss = pred_fake - pred_real + gradient_penalty
critic_loss.backward()
critic_optimizer.step()

critic_step += 1
```

C = 0.01

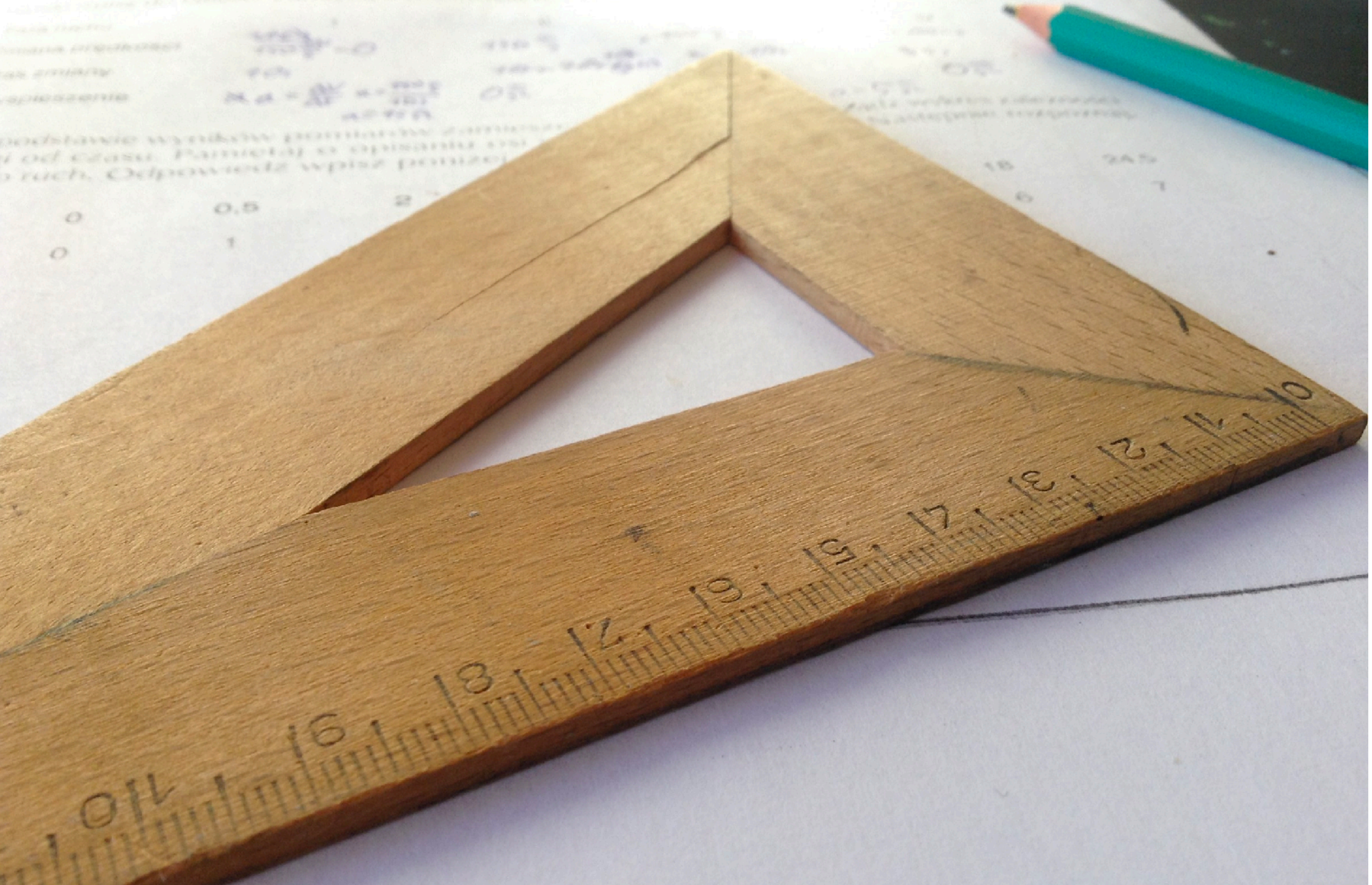
N_Z = 10

LAMBDA = 10.0

```
generator.zero_grad()

# get latent noise vector
z = to_var(torch.randn(batch_size*2, N_Z))

# generate images
fake_images = generator(z)

# prediction on fake data
pred_fake = torch.mean(critic(fake_images))

# loss and back propagation
generator_loss = -pred_fake
generator_loss.backward()
generator_optimizer.step()
```

N_Z = 10

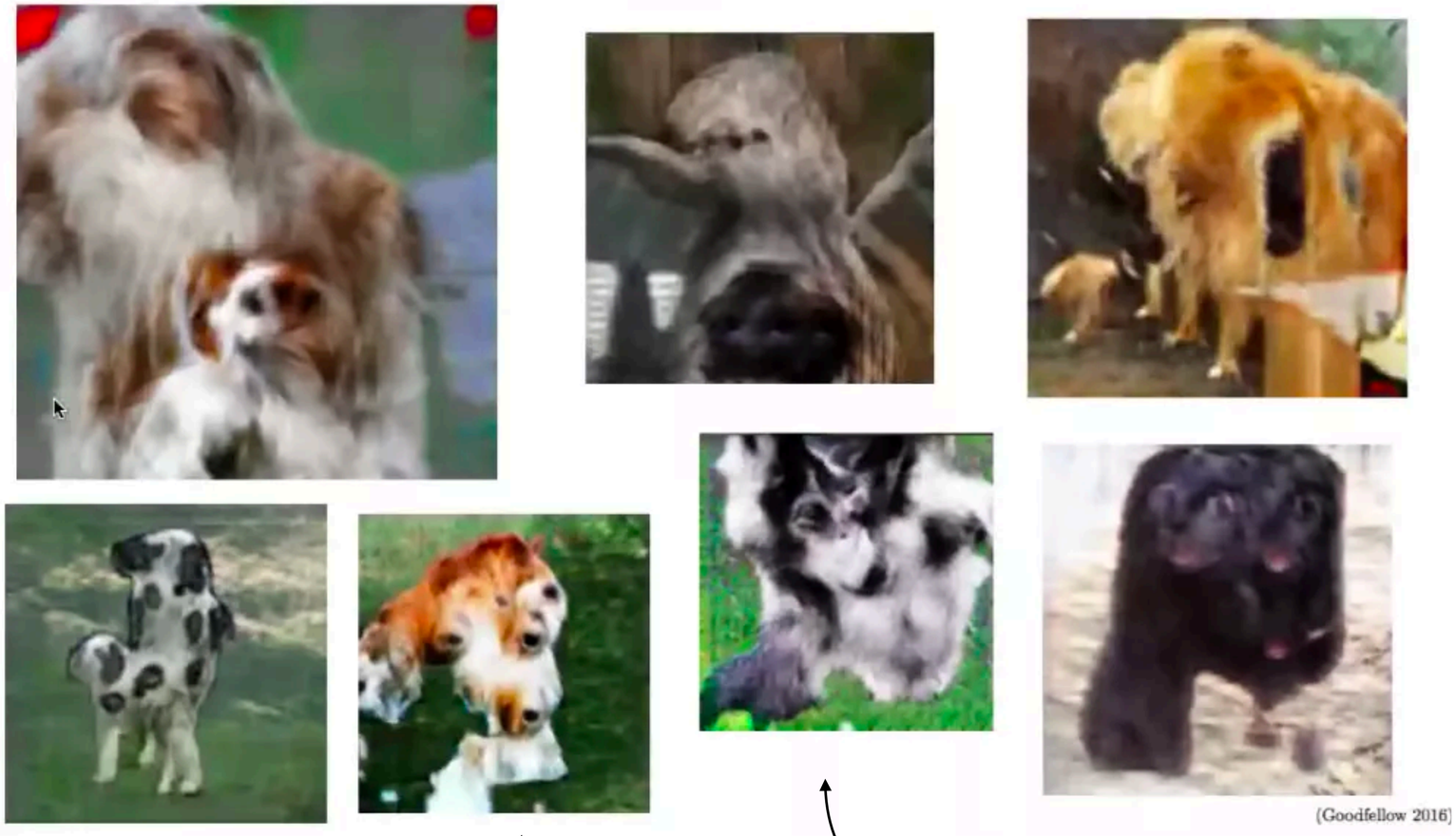
```
def to_var(x):
    if torch.cuda.is_available():
        x = x.cuda()
    return Variable(x)
```



Metrics

Metrics: Visual

- Works well for real images using real humans to judge image quality

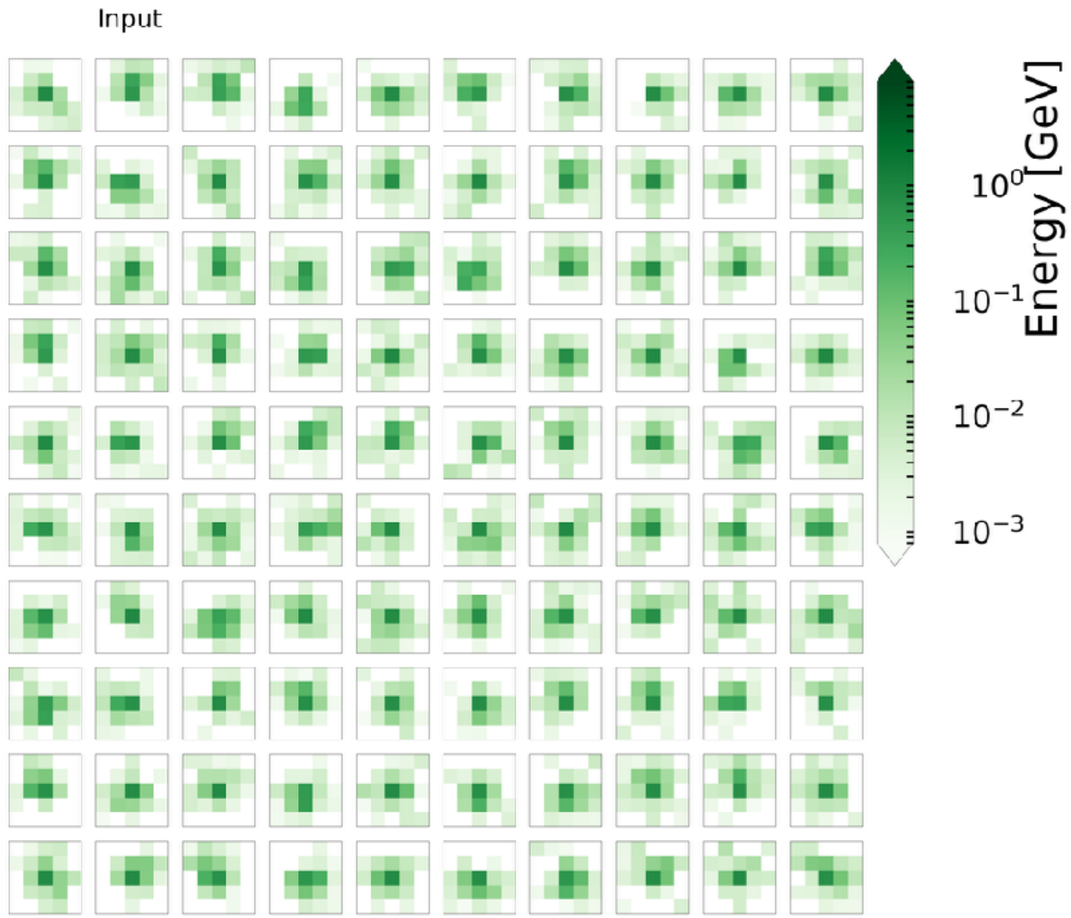


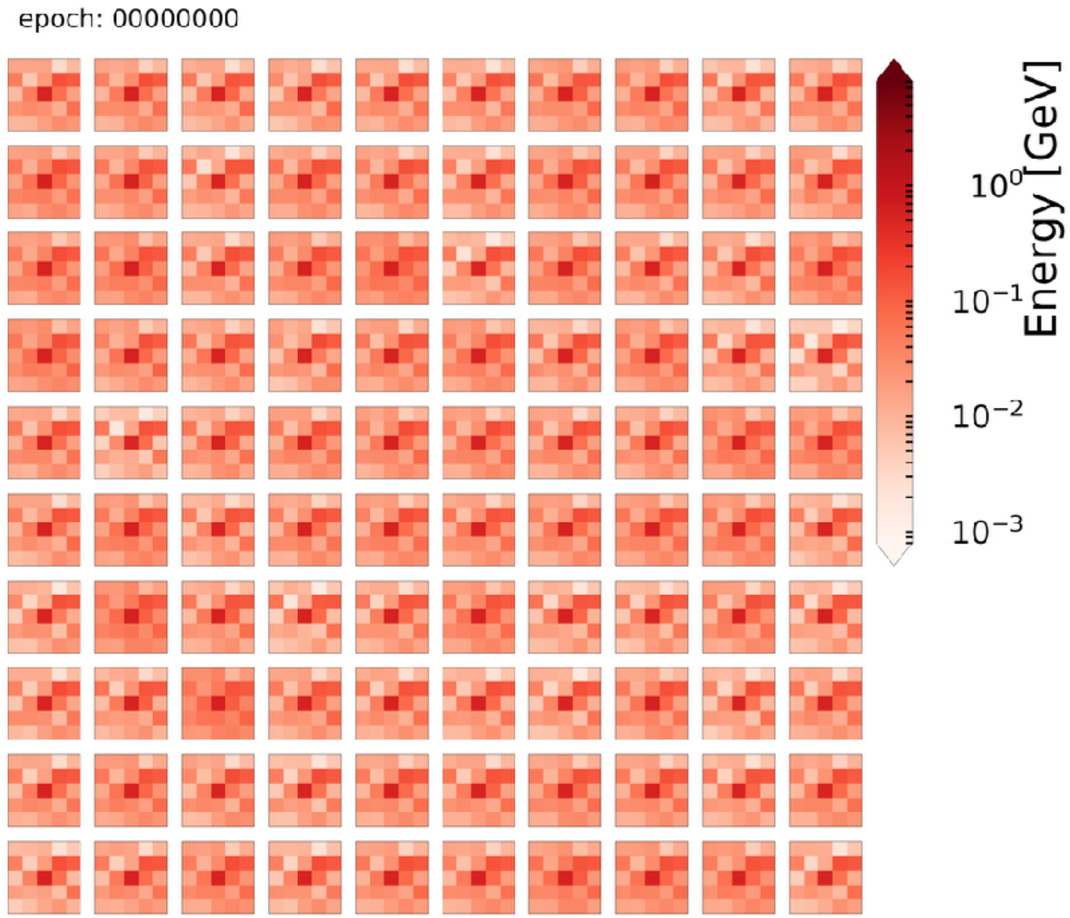
[1] Ian Goodfellow 2016

How good is the GAN?

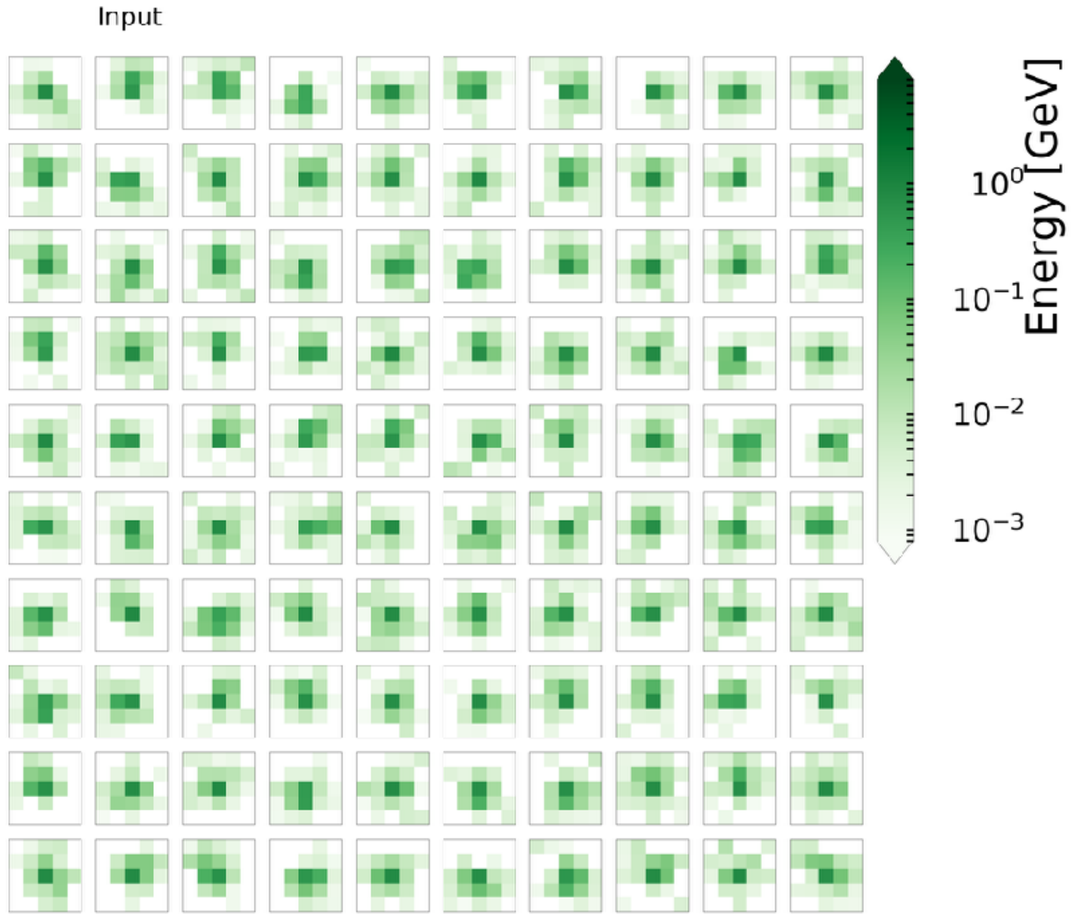
- Losses are difficult:
  - Vanilla GANs play a minimax game where losses are traded
  - WGAN loss is (somewhat) proportional to image quality
- Better:
  - Feature differences like 1D Wasserstein distances (from scipy)
  - Quality measures from image processing:
    - Structural similarity (SSIM)
    - Kernel principal component analysis (kPCA)
- Classification based metrics (Inception score, …)

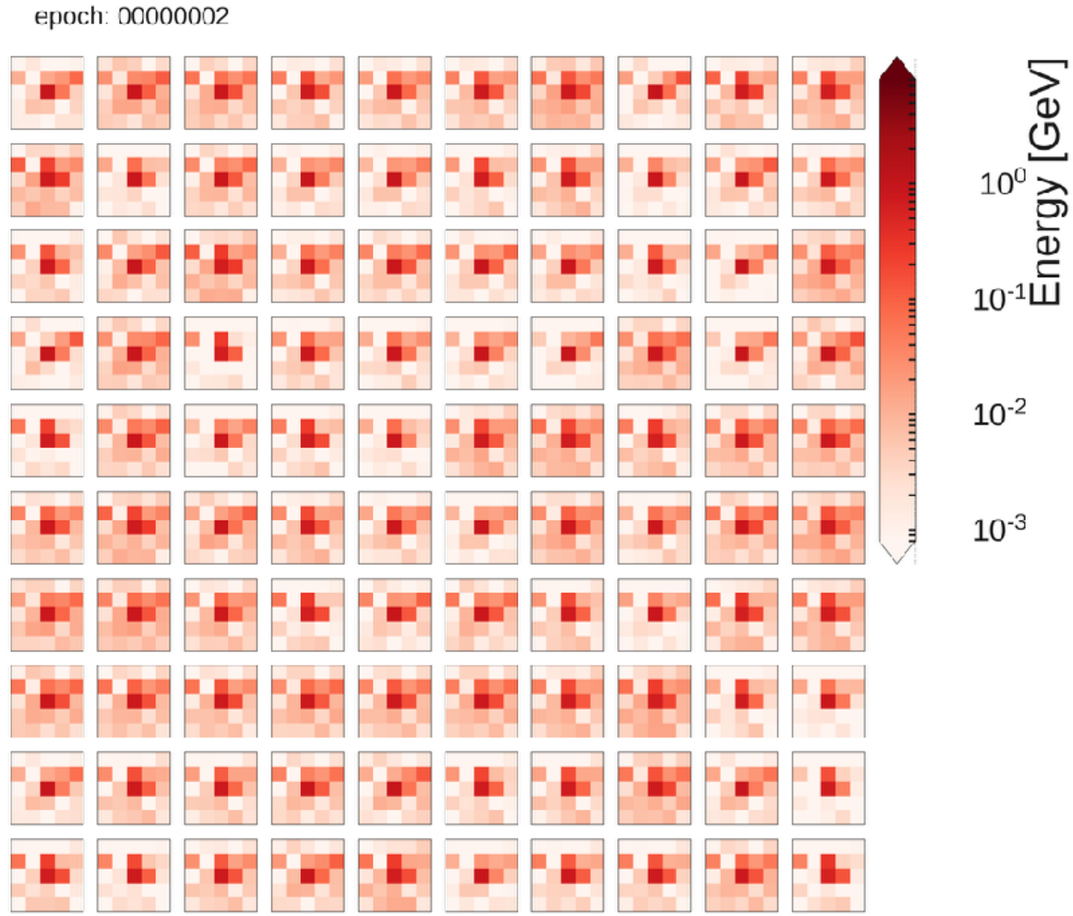
WGAN-GP in PyTorch: Input → Output



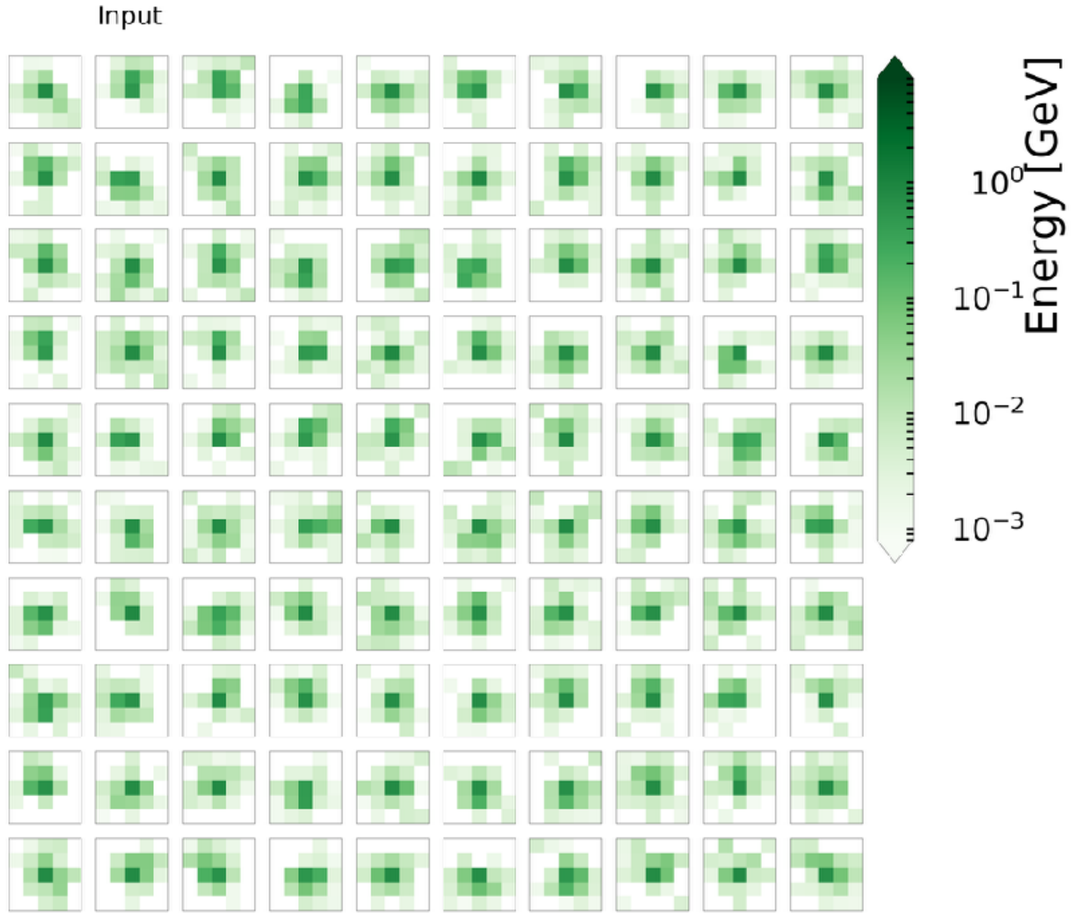


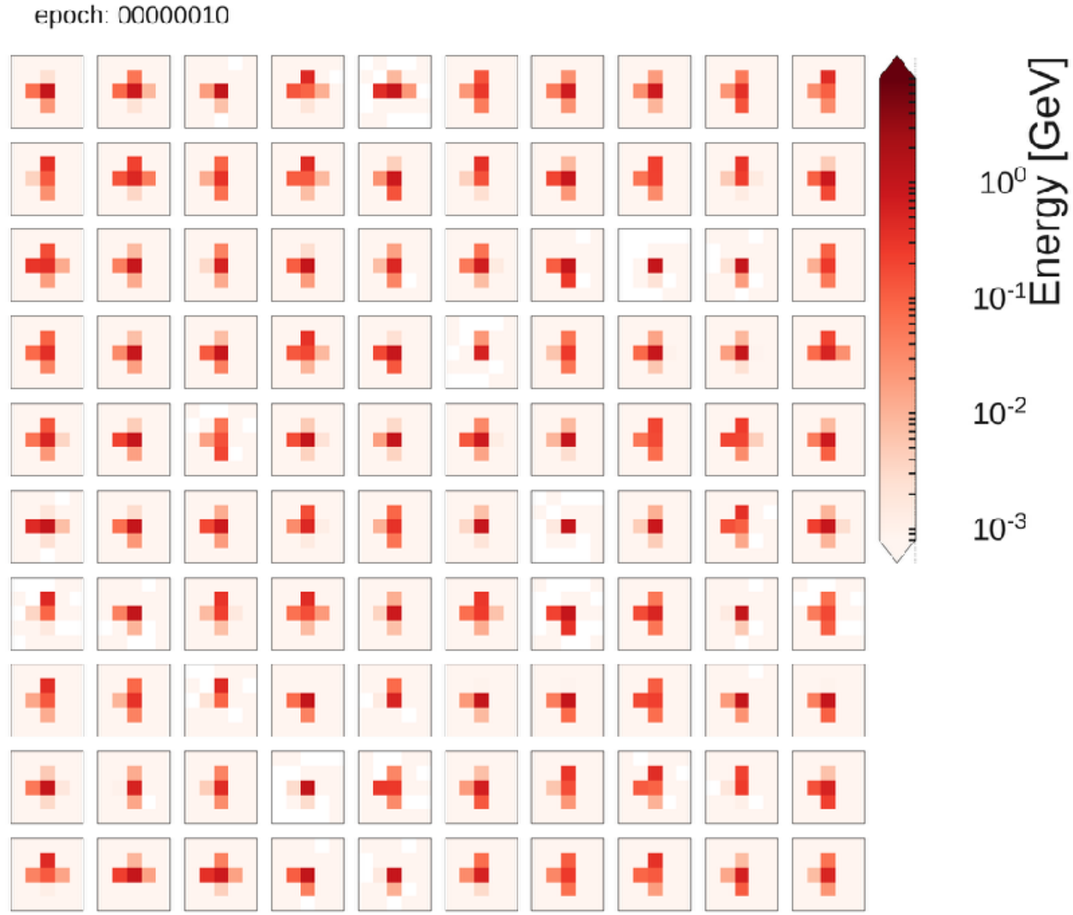
WGAN-GP in PyTorch: Input → Output



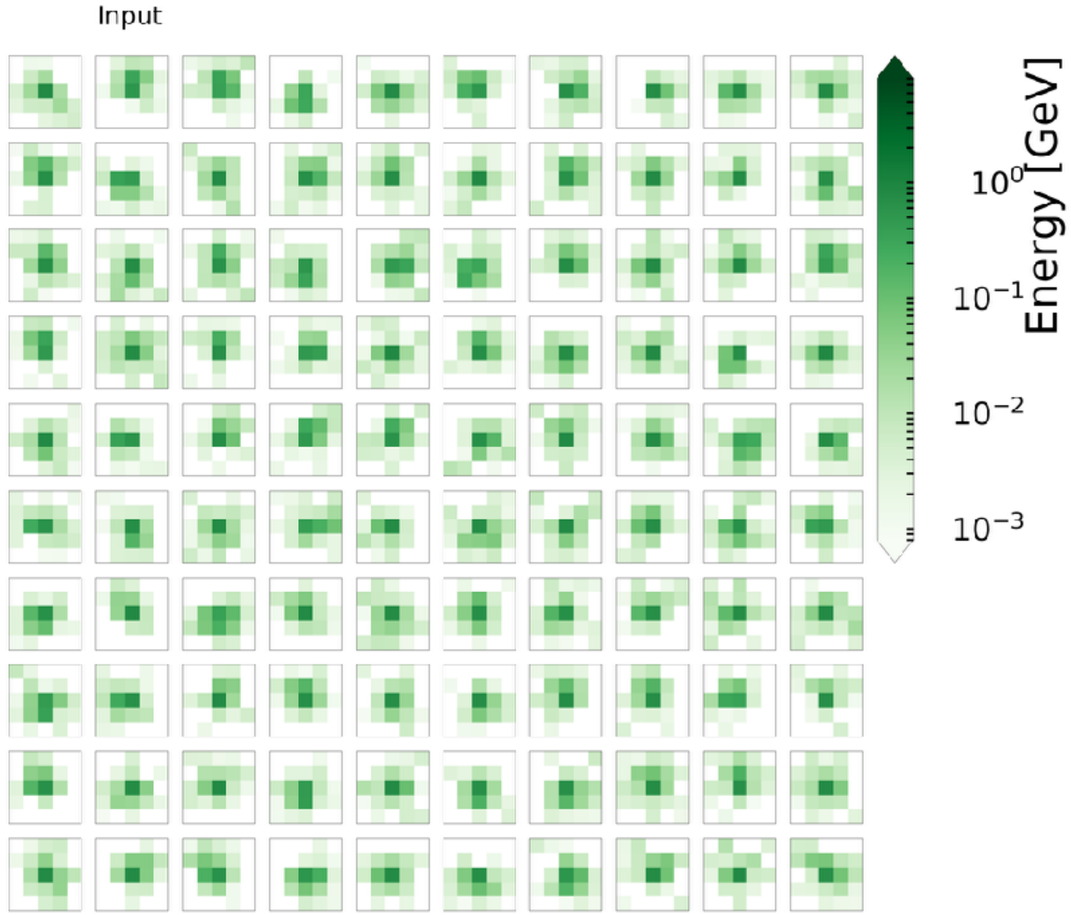


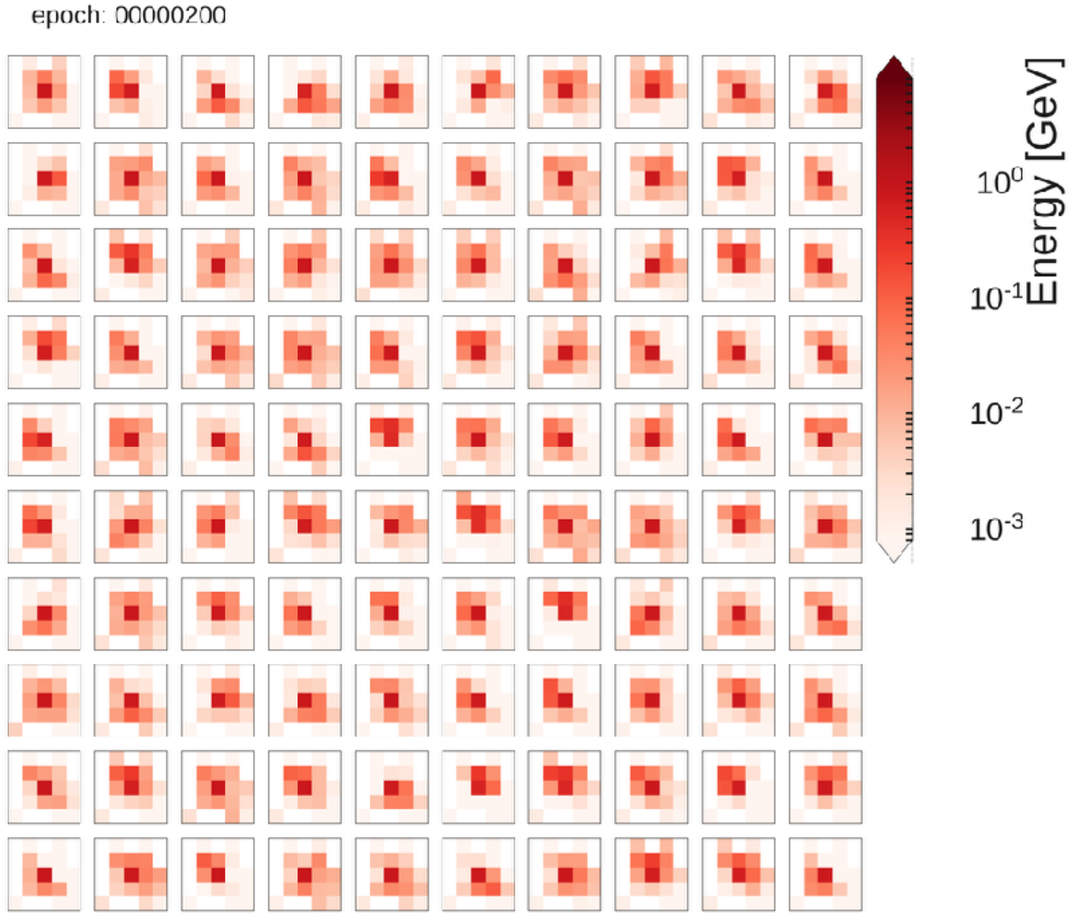
WGAN-GP in PyTorch: Input → Output



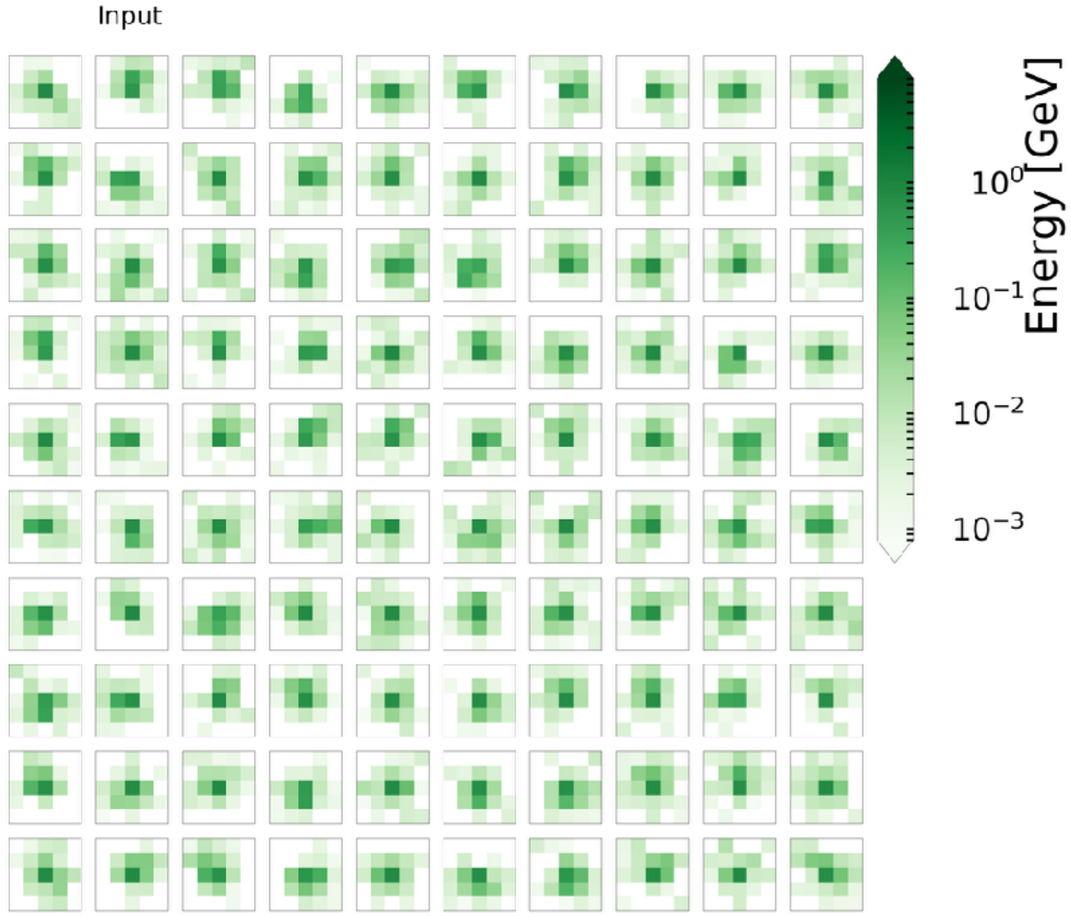


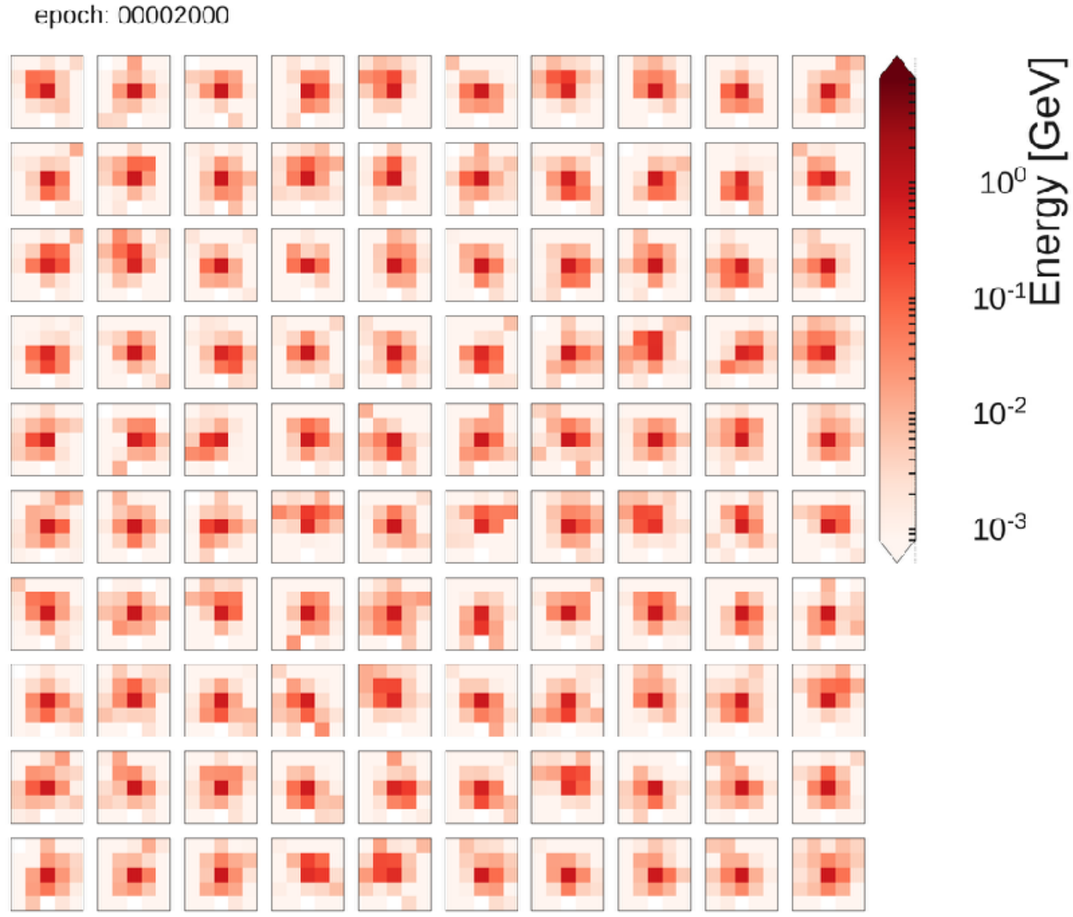
WGAN-GP in PyTorch: Input → Output



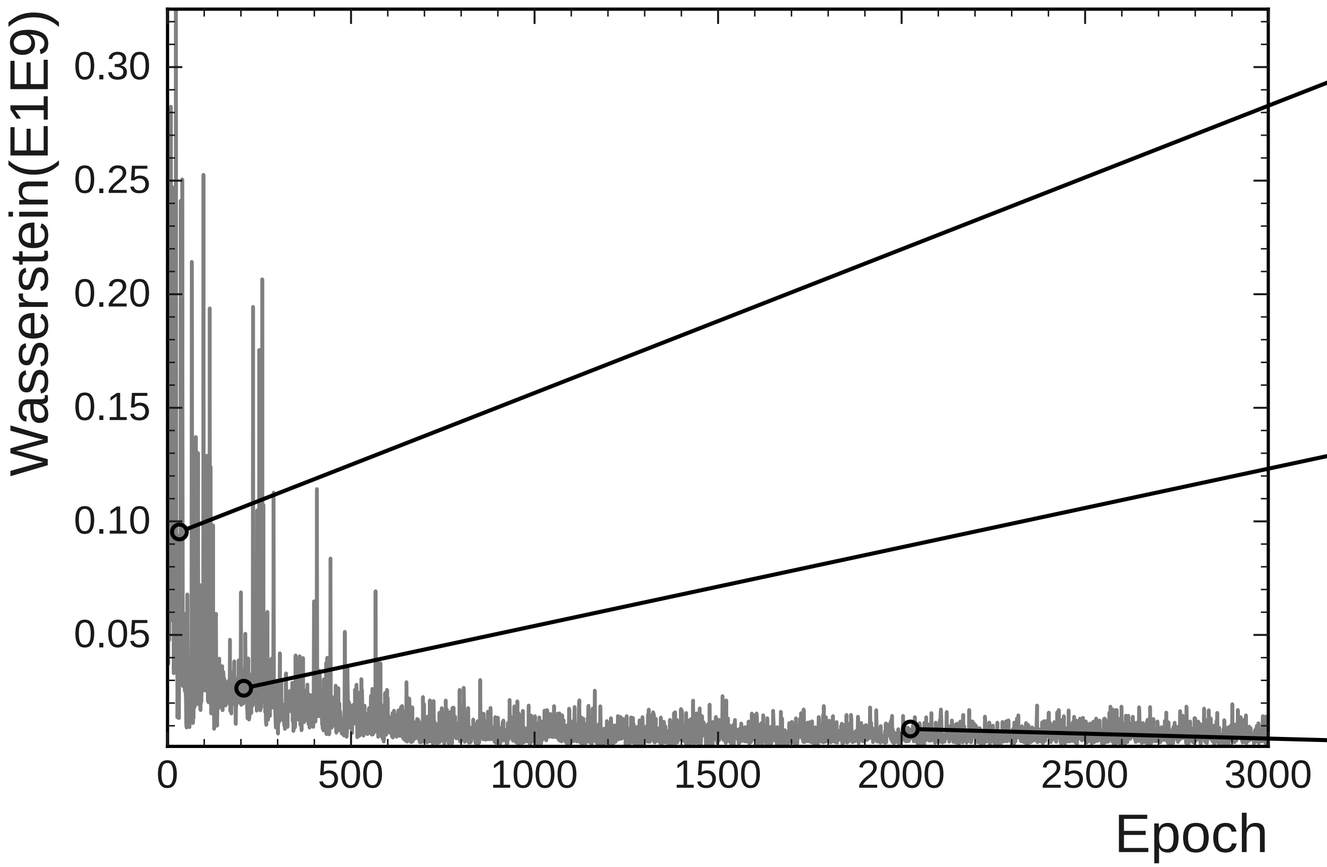


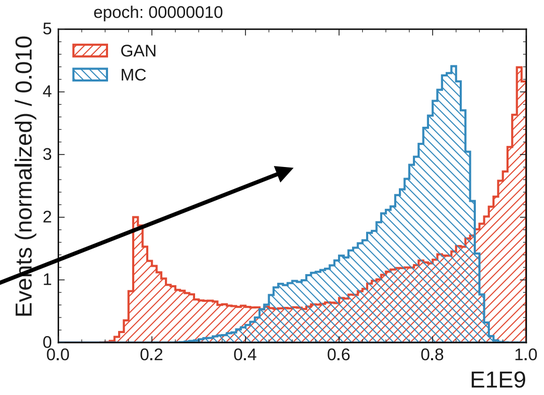
WGAN-GP in PyTorch: Input → Output

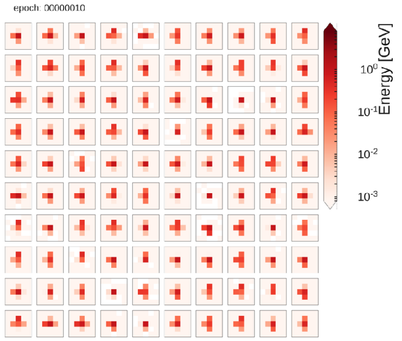


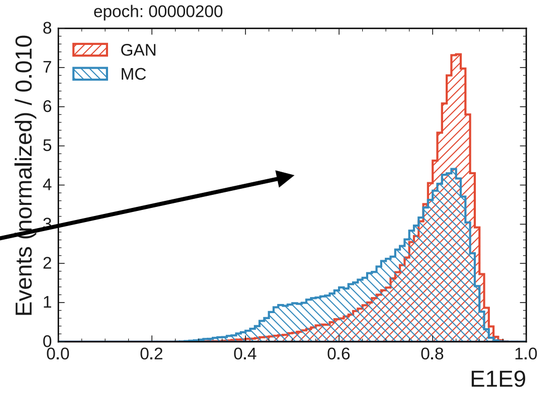


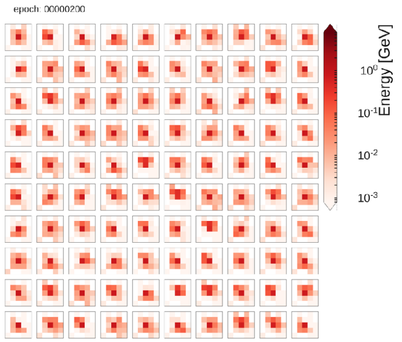
How good is the GAN?

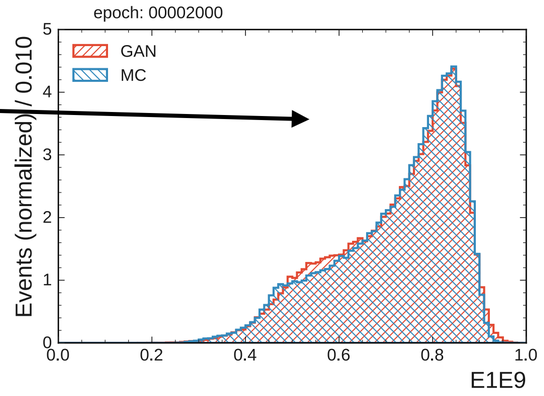


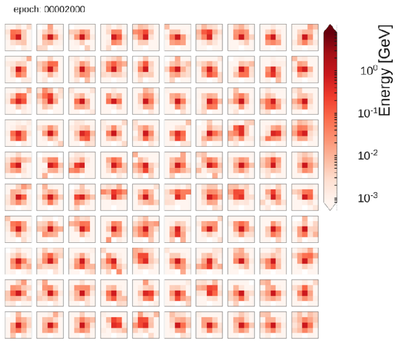






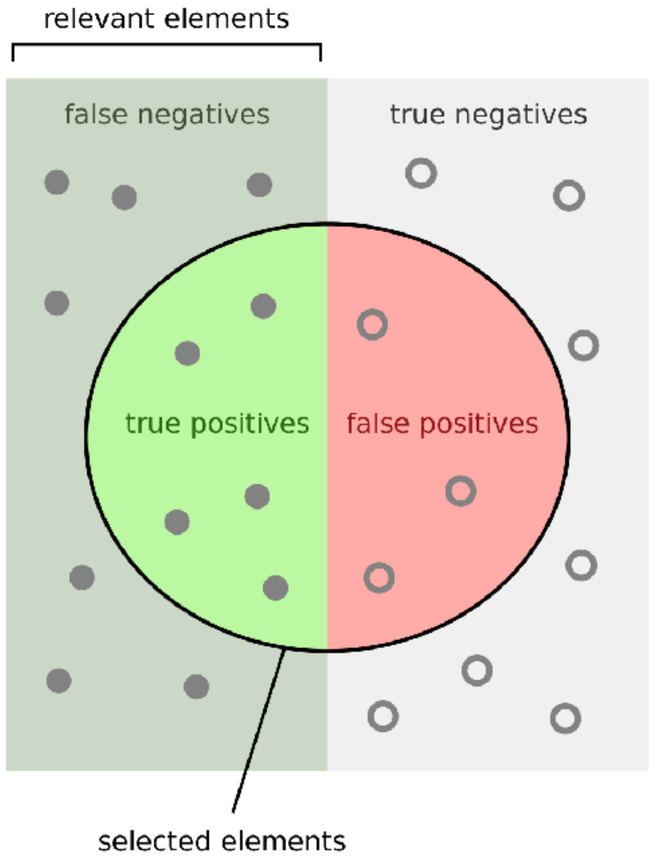


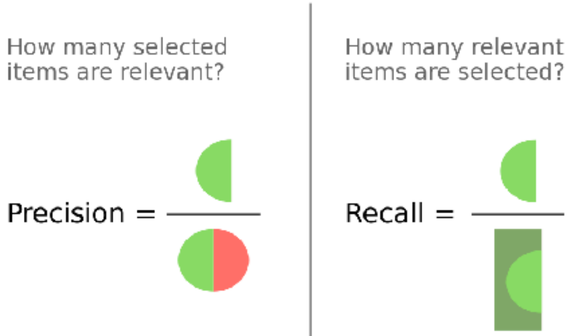




Precision, recall, and F1

- Quality indicator for (multi) mode discriminative models.
  - **Precision**: $P = T_p / (T_p + F_p)$
  - **Recall**: $R = T_p / (T_p + F_n)$
  - **F1**: $F1 = 2 \times (P \times R)/(P+R)$, Bad is 0, Good is 1
- Be aware that relative importance of precision and recall is very problem dependent:
  - Assume you want to detect nuclear missile attacks and have one false negative…





[1] https://en.wikipedia.org)

PRD Curves (Precision and recall for distributions)

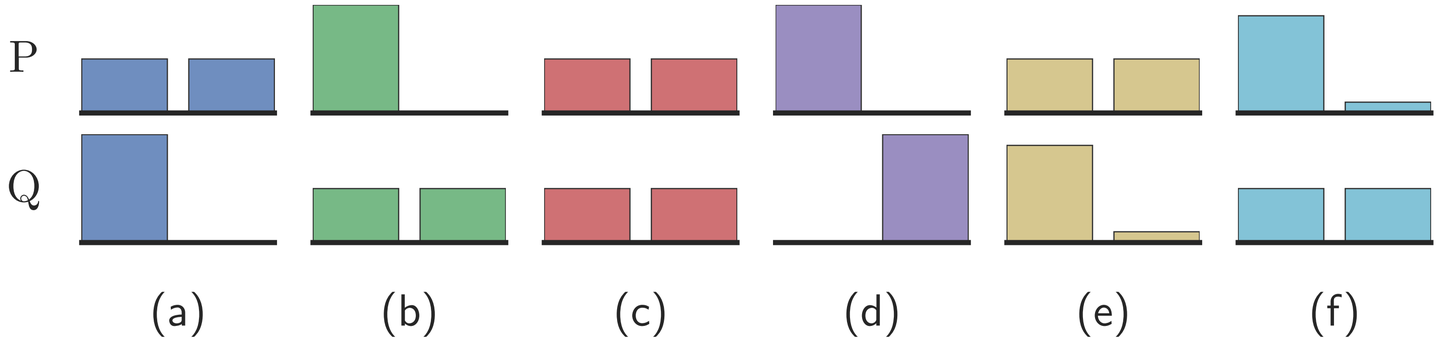


Figure 2: Intuitive examples of P and Q.

Precision α
Recall β

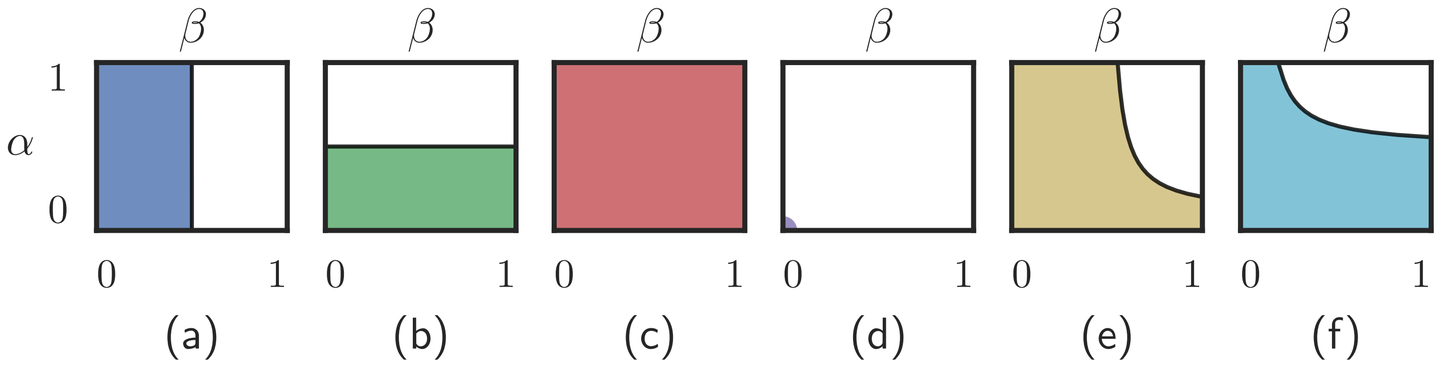


Figure 3: $\mathrm{PRD}(Q, P)$ for the examples above.

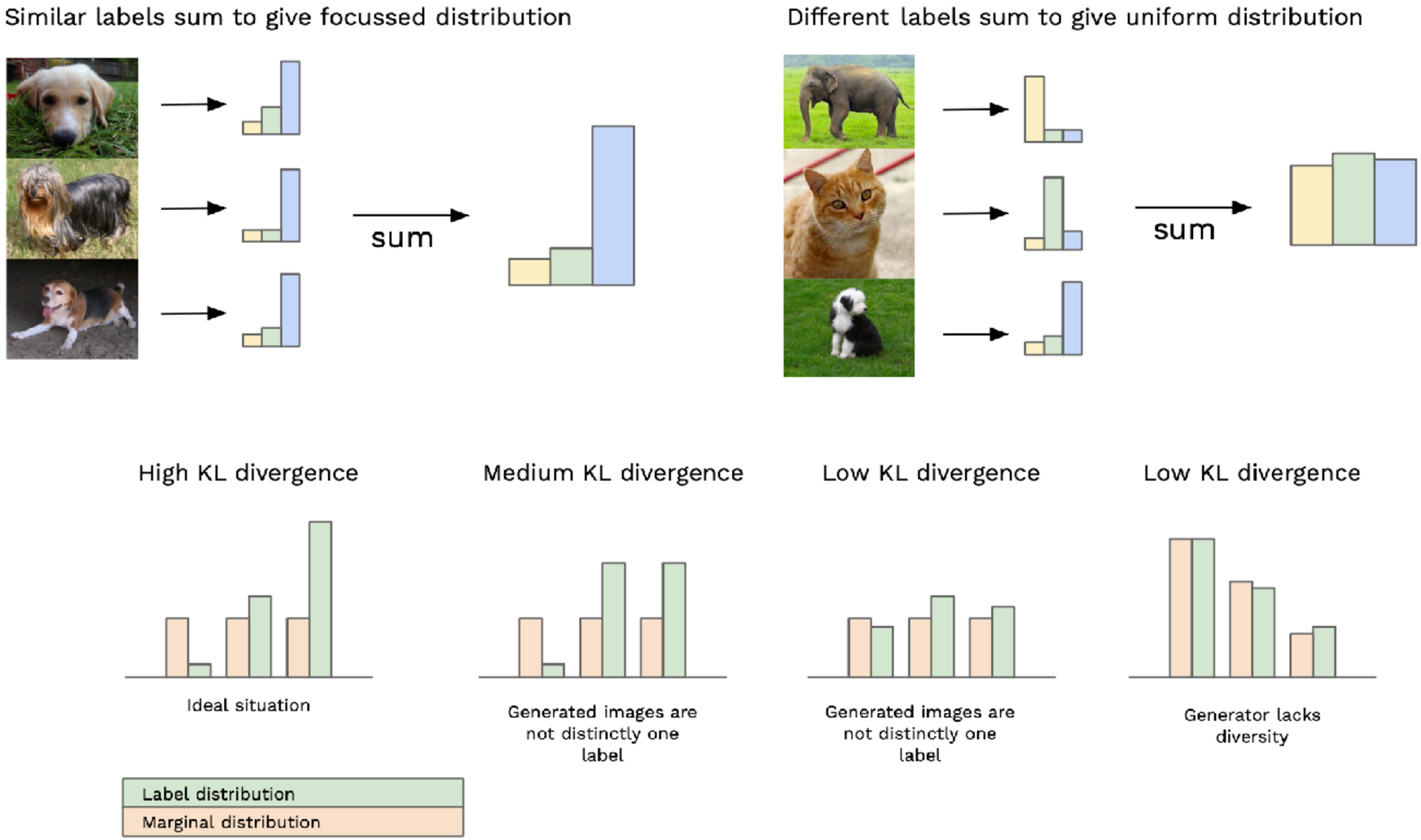
[1] "Assessing Generative Models via Precision and Recall" https://arxiv.org/abs/1806.00035
[2] https://github.com/msmsajjadi/precision-recall-distributions

Inception score (IS)

- Use a powerful and well-trained discriminator (originally Inception-v3 on ImageNet)
- A well performing GAN generator will produce:
  - Clear images for every mode (low entropy for conditional label distribution)
  - All modes, correct fraction of modes (high entropy for marginal distribution)
- IS = Exponential of KL distance between class distribution and marginal distribution
  - High score is better (unbound)

[1] arxiv:1606.03498

Inception score (IS)



[1] https://cdn-images-1.medium.com/max/2400/1*23gj_d3dxfm5FoKae_pc5Q.png

Inception score (IS): Potential problems

- IS depends on (powerful) classifier and their training data
- Classifiers often rely on CNNs which generally rely on local textures. This is not necessarily what one desires.
- Generator overtraining will not be penalized
- Low variance of images per mode will not be penalized
- → Fréchet Inception Distance (FID) is generally superior (but more complicated): Small FID is better

There is not one single metric to compare GANs.

Improvements

Ibaraki-Bosozoku

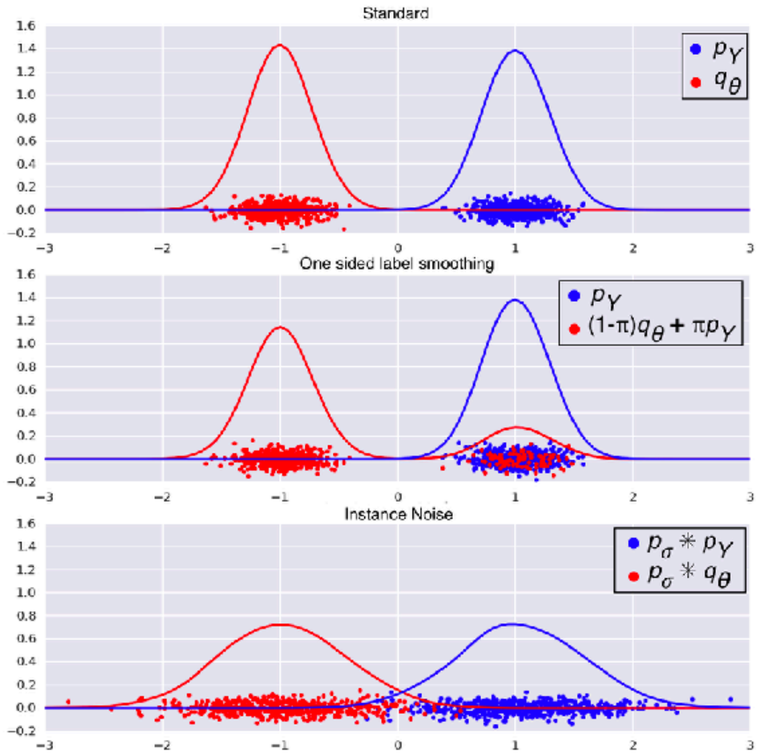
One sided-label smoothing (for Vanilla GANs)

- Goal: Avoid overconfidence of Discriminator
- Almost trivial idea, but very powerful.
- Smooth labels for real images (1→0.9): Reduce confidence for real images to reduce steep gradients.
- Never (!) smooth labels for fake images (0→0.1): This would encourage the generator to continue producing wrong images.
- A variant is to smooth with a random value instead of a fixed 0.1.

[1] https://arxiv.org/abs/1701.00160

Instance and label noise

- Goal: Create overlapping distributions
- Add noise to generated and real images before passing them to generator (instance noise)
- and/or flip labels of real images (label noise)
- Can stabilise early training but will reduce image quality



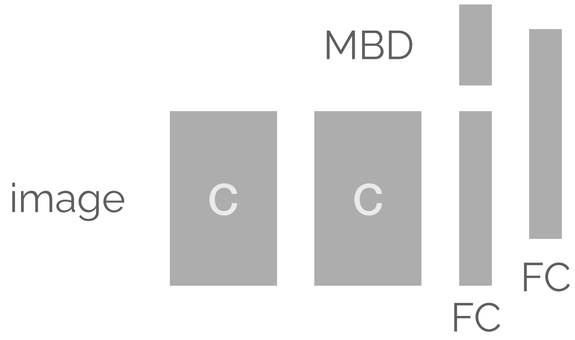

```
def add_instance_noise(images, std=0.1):

   noise = Variable(std * torch.randn(images.shape))
   return images + noise
```

[1] https://www.inference.vc/

Minibatch discrimination

- Goal: Avoid mode collapse
- Measure similarity of all images in a batch. If similarity increases: Mode collapse
- Add similarity measure(s) as input to (first) fully connected layer
- Can be computationally intensive if batch size is large
- Poor man's version: Just detect if images are the same by checking one statistics, e.g. the mean

```
class critic(nn.Module):

# ...

    def main(self, x):
        out = torch.cat((x,
                         self.mbd(x).cuda()), dim=1)

        for layer in self.layer_module:
            out = layer(out)

        return out.view(out.size(0), -1)
```



[1] arxiv:1606.03498

Batch normalization

- Goal: Stable training (used in DCGAN and generally in every CNN)

- Problem: Introduces intra-batch correlations
  Solution: Virtual Batch Norm using a reference batch chosen before training

[1] GANs (https://arxiv.org/abs/1406.2661)

Experience replay

- Keep a history of generated images
- Feed a (small) fraction of old images to discriminator to avoid overtraining of the discriminator

[1] https://xkcd.com/1757/

Historical averaging

- Keep history of model parameters **θ** for the last **t** models (or keep the running average)

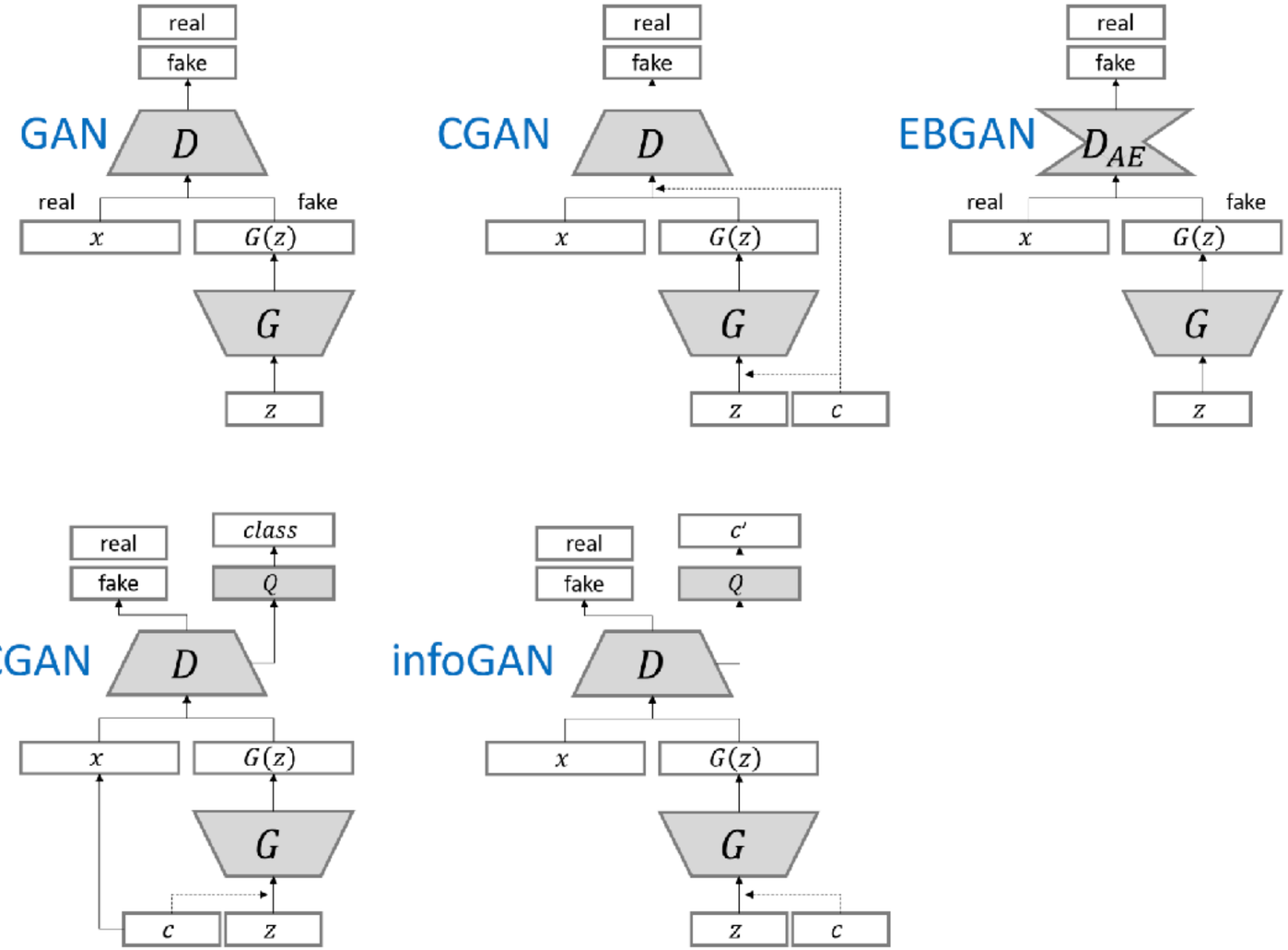
- Add L2 penalty to loss function if current model parameters deviate too much from past average:

$$||\theta - \frac{1}{t}\sum_{i=1}^{t}\theta[i]||_2$$

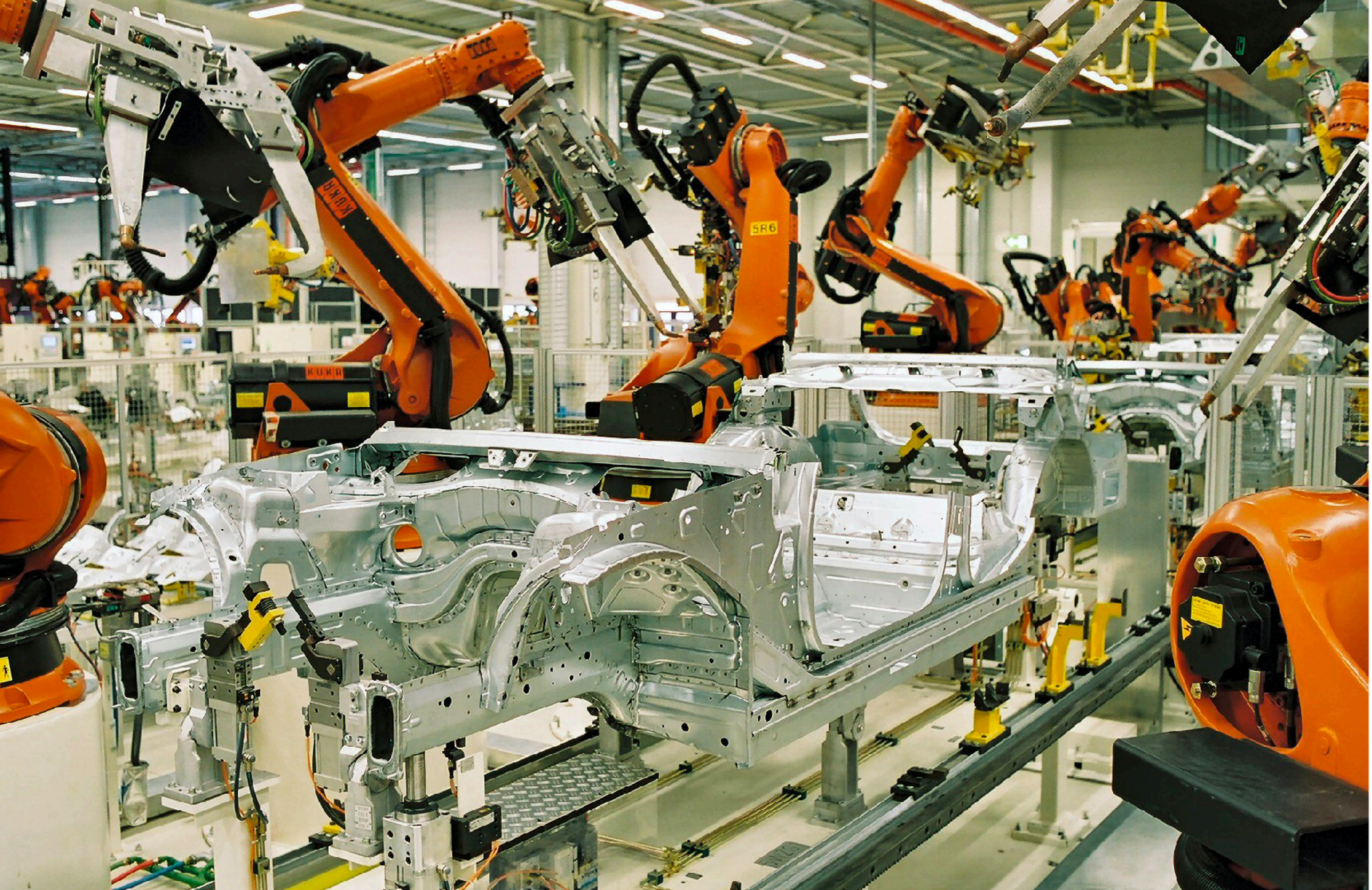
Class conditioning

- Standard (W)GANs have no additional information about the modes in data
- Several problems in GANs are related to that: Slow training, mode collapse, bad perceived image quality, …
- Possible solution is called class conditioning:
  - Class-conditional (add class to latent space of generator)
  - Class discrimination (add class discrimination to discriminator)
  - Combinations of those

Class conditioning

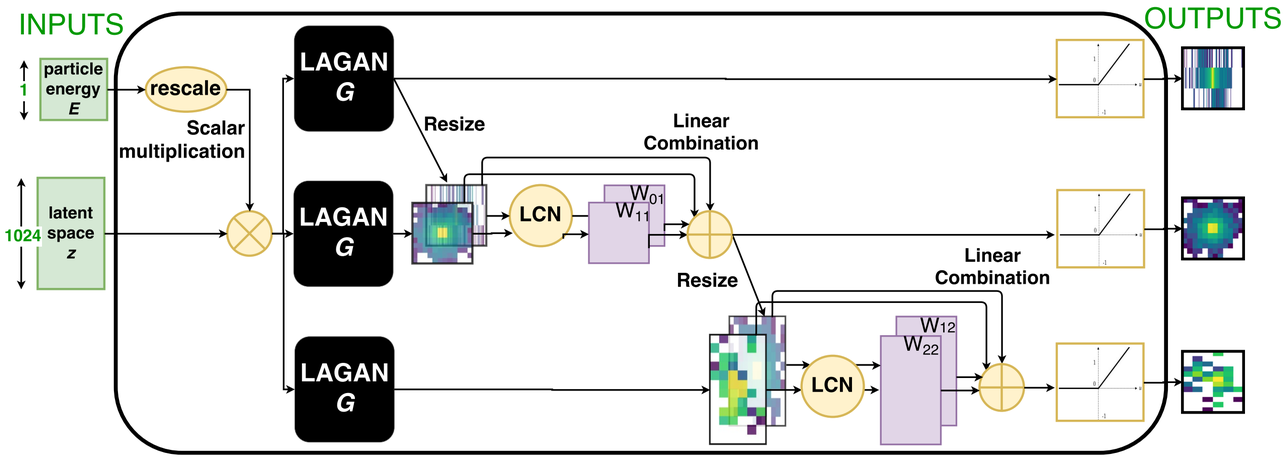


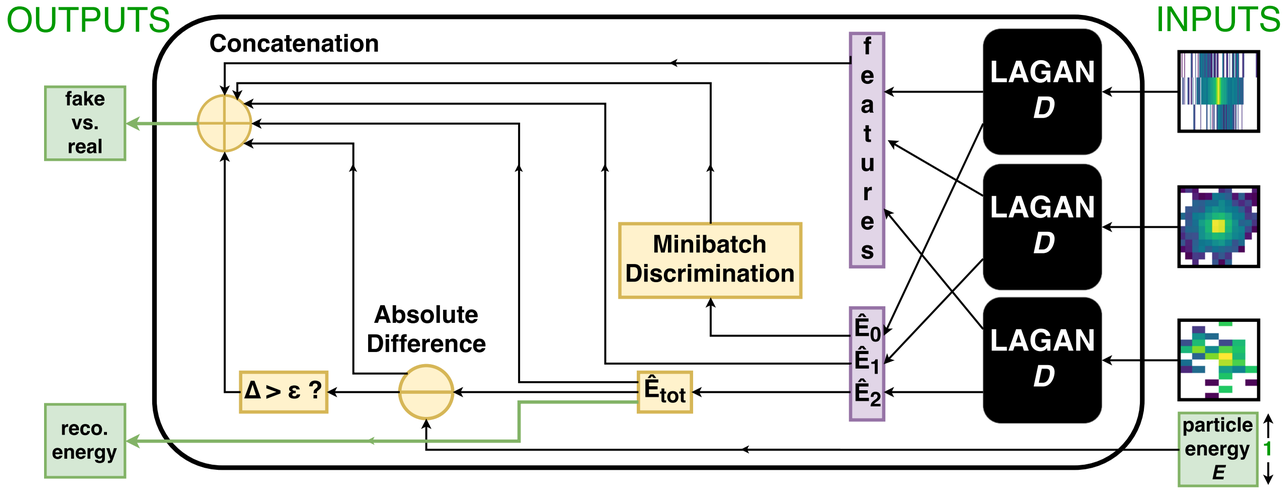
[1] https://github.com/znxlwm/pytorch-generative-model-collections



Research

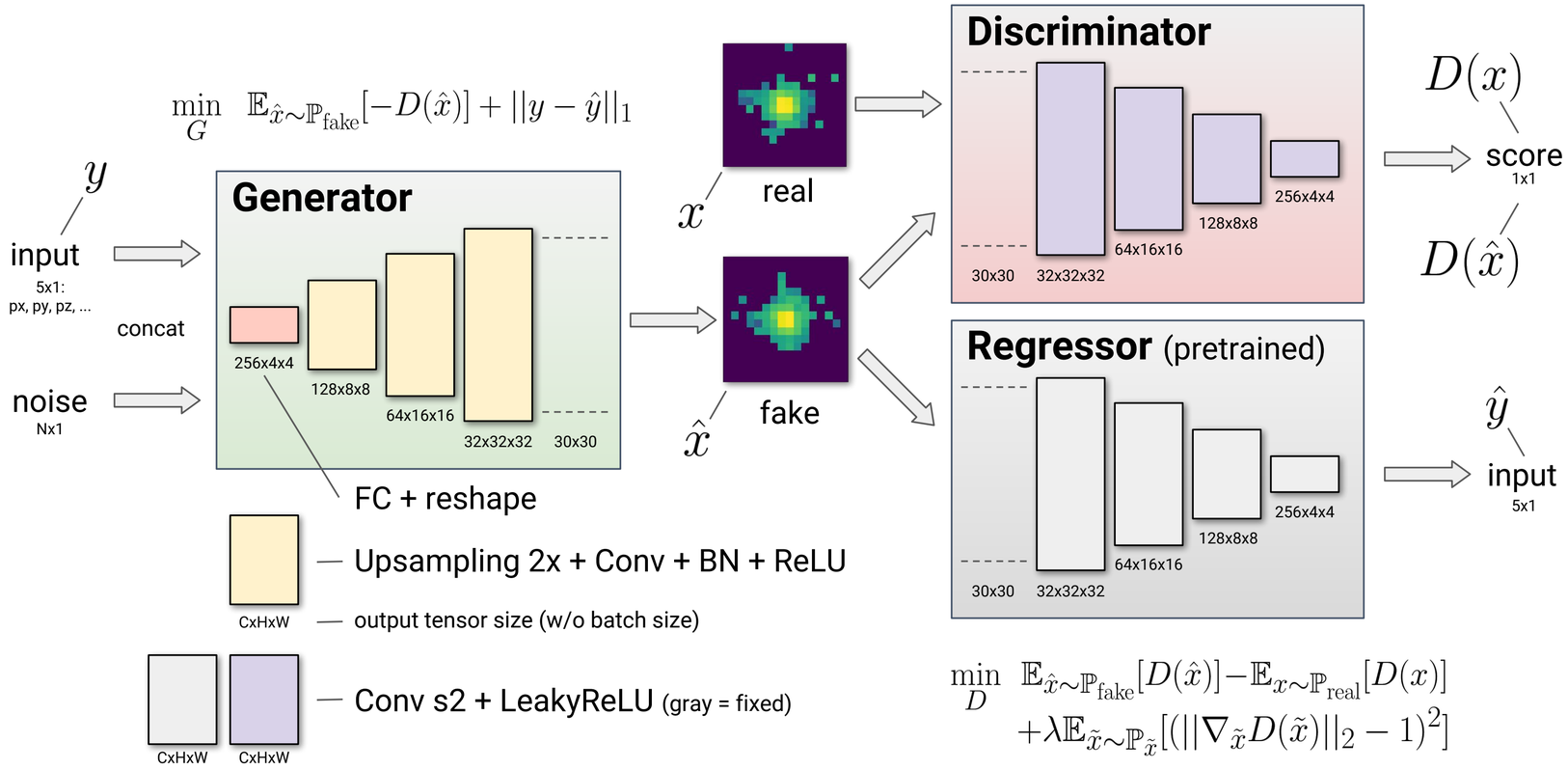
CaloGAN(s)





[1] "CaloGAN: Simulating 3D High Energy Particle Showers in Multi-Layer Electromagnetic Calorimeters with Generative Adversarial Networks" arXiv:1712.10321 (LAGAN: arXiv:1701.05927)

LHCbGAN



[2] "Generative Models for Fast Calorimeter Simulation: the LHCb case" arXiv:1812.01319v2

Summary

- GANs are cool, fun, and sometimes almost magical
- GANs are difficult
- GANs are a young field in ML with ongoing basic research
- Numerous applications in physics (simulation, calibration, understanding, …)
  - Many things that are irrelevant when creating human faces are very relevant for physics
  - Get in touch to learn from each other

Summary

- GANs are cool, fun, and sometimes almost magical
- GANs are difficult
- GANs are a young field in ML with ongoing basic research
- Numerous applications in physics (simulation, calibration, understanding, …)
  - Many things that are irrelevant when creating human faces are very relevant for physics
  - Get in touch to learn from each other

Contact

DESY.

Deutsches Elektronen Synchrotron
www.desy.de

Torben Ferber
torben.ferber@desy.de
ORCID: 0000-0002-6849-0427