
# GENERATIVE ADVERSARIAL NETWORKS

María Ana Ortiz
Cristian M. Amaya
Angela Castillo
María Camila Escobar

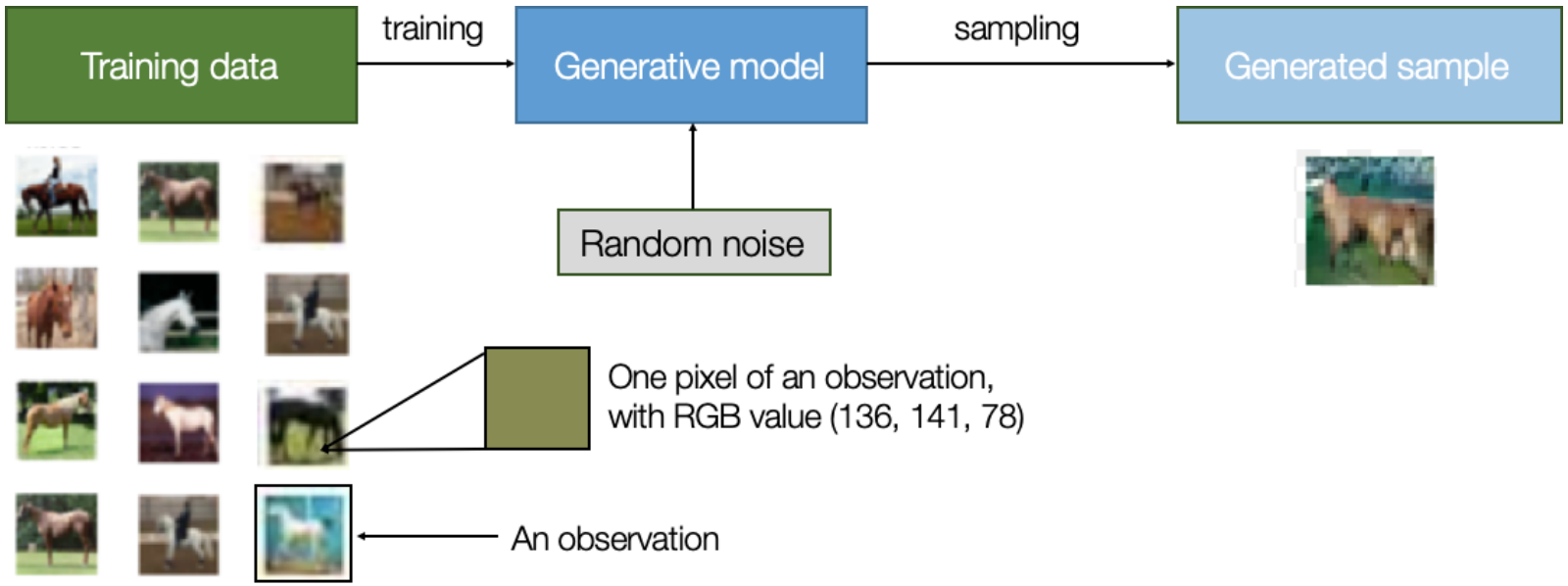
# OUTLINE

- Introduction: Generative Models
- Generative Adversarial Networks
- History
- Recent Work
- Articles
- Tutorial
- Homework

# GENERATIVE MODELS

Model that describes how a dataset or domain is generated, in terms of a probabilistic model.

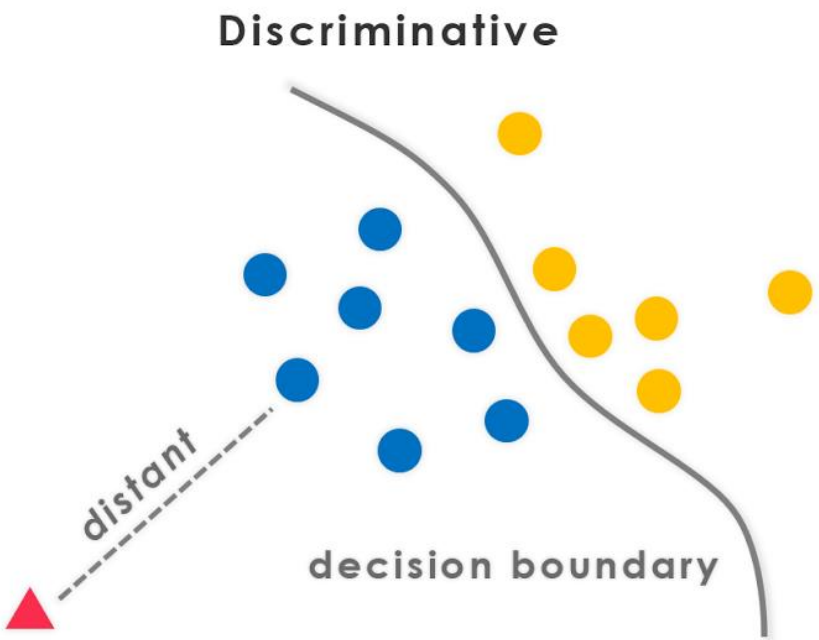
By sampling from this type of models, we can generate new observations.

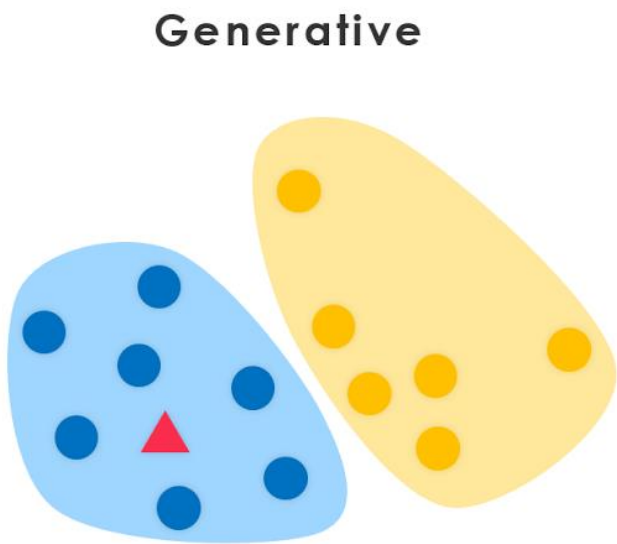


# GENERATIVE VS DISCRIMINATIVE

- **Discriminative model:** estimates p(y | x) - the probability of a label ***y*** to be given to observation ***x***

- **Generative model:** estimates p(x) - the probability of observing observation **x**. If the data is labeled, we could also estimate the joint distribution p(x,y).



# MAXIMUM LIKELIHOOD METHOD

The main idea is to define a model that provides an estimate of a probability distribution.

Likelihood:

Number of samples in the dataset

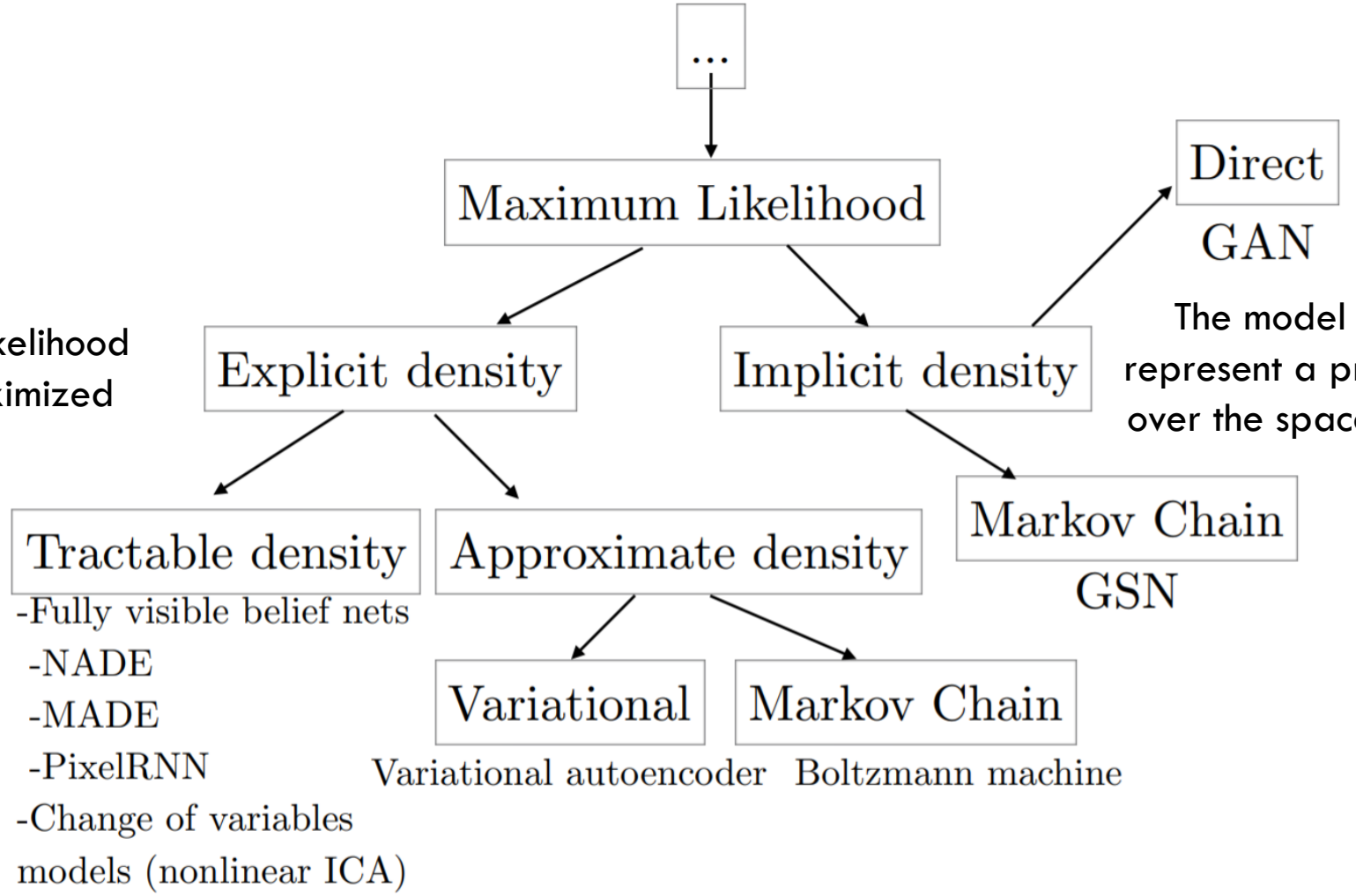
Parametrization parameters

$$\prod_{i=1}^{m} p_{\text{model}}\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right)$$

Training examples

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} D_{\text{KL}}\left(p_{\text{data}}(\boldsymbol{x}) \| p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta})\right)$$

# TAXONOMY OF GENERATIVE MODELS

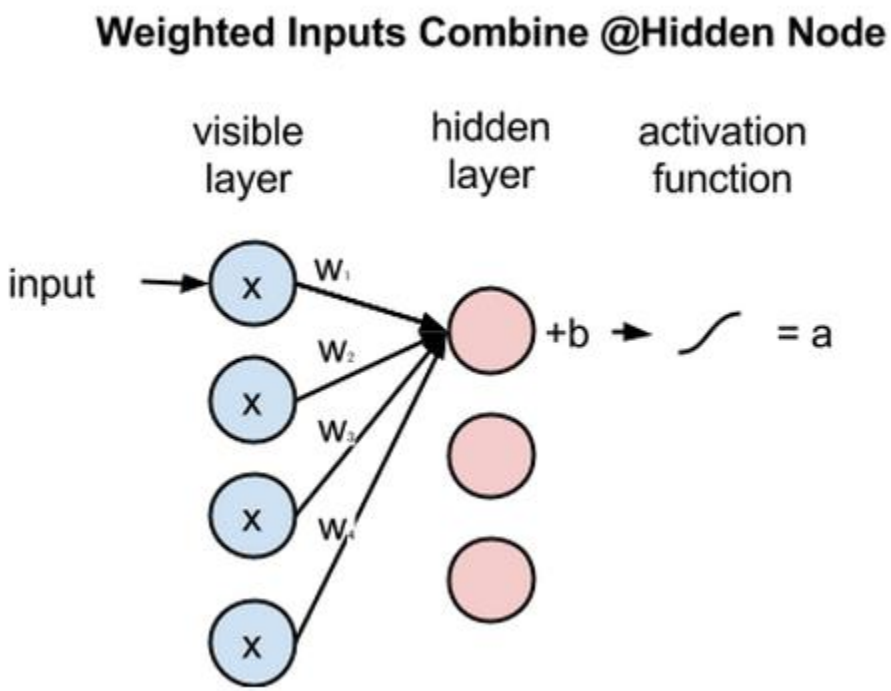


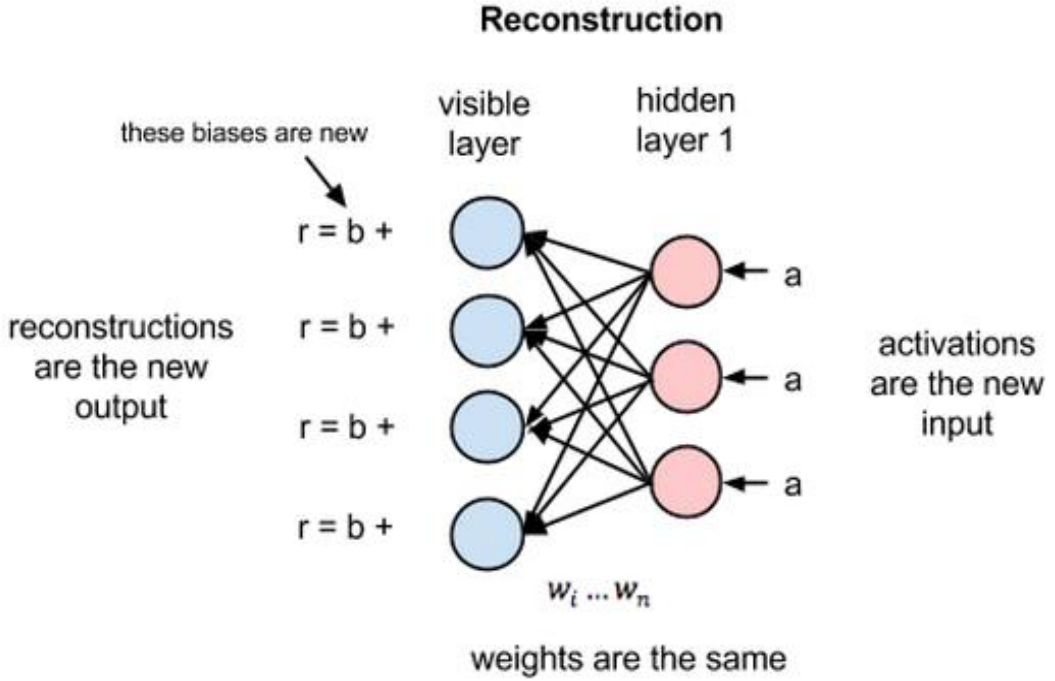

Have an explicit likelihood which can be maximized

The model does not explicitly represent a probability distribution over the space where the data lies

# RESTRICTED BOLTZMAN MACHINE

Energy based model. Learning which groups of pixels tend to co-occur for a given set of images.




$$E(v,h) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_i \sum_j v_i w_{i,j} h_j \qquad P(v,h) = \frac{1}{Z} e^{-E(v,h)}$$

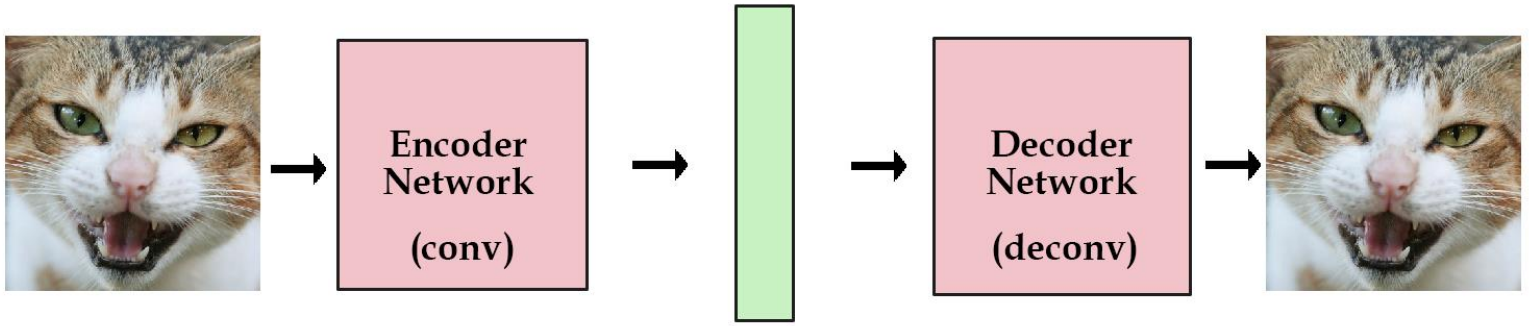
# FULLY VISIBLE BELIEF NETWORKS

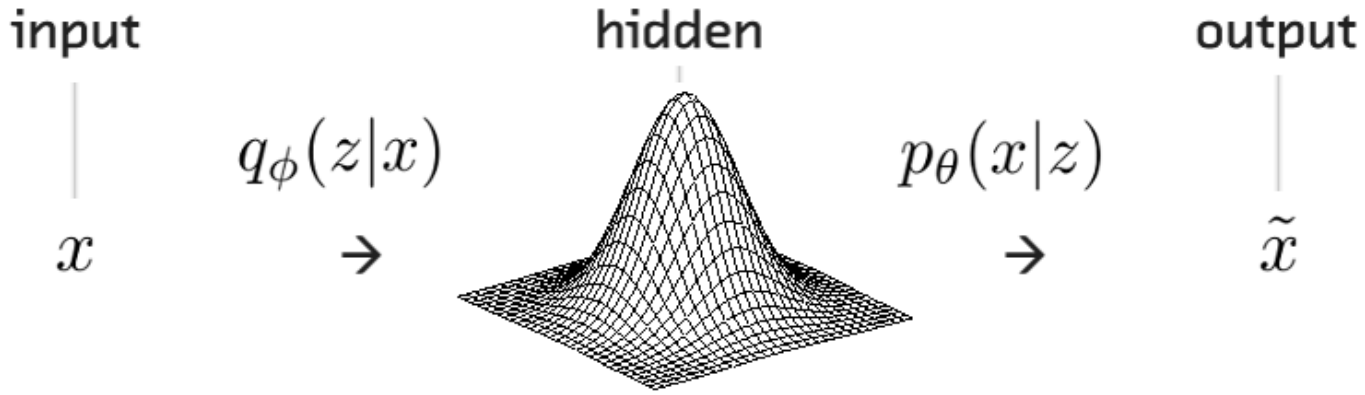
Use the chain rule to breakdown the probability distribution and turn it into a product of one-dimensional distributions.

$$p_{\text{model}}(\boldsymbol{x}) = \prod_{i=1}^{n} p_{\text{model}}\left(x_i \mid x_1, \ldots, x_{i-1}\right)$$

Because this generating scheme allows the generation of new samples one entry at a time the computational cost of this method is high. Steps cannot be parallelized.

# VARIATIONAL AUTOENCODERS



# VARIATIONAL AUTOENCODERS

Lower bound $\mathcal{L}(\boldsymbol{x};\boldsymbol{\theta}) \leq \log p_{\text{model}}(\boldsymbol{x};\boldsymbol{\theta})$

When either a too weak approximate posterior distribution or a too weak prior distribution is used, the gap between $\mathcal{L}$ and the true likelihood can lead to $p_{model}$ learning different from the real $p_{data}$.

# REGARDING GANS...

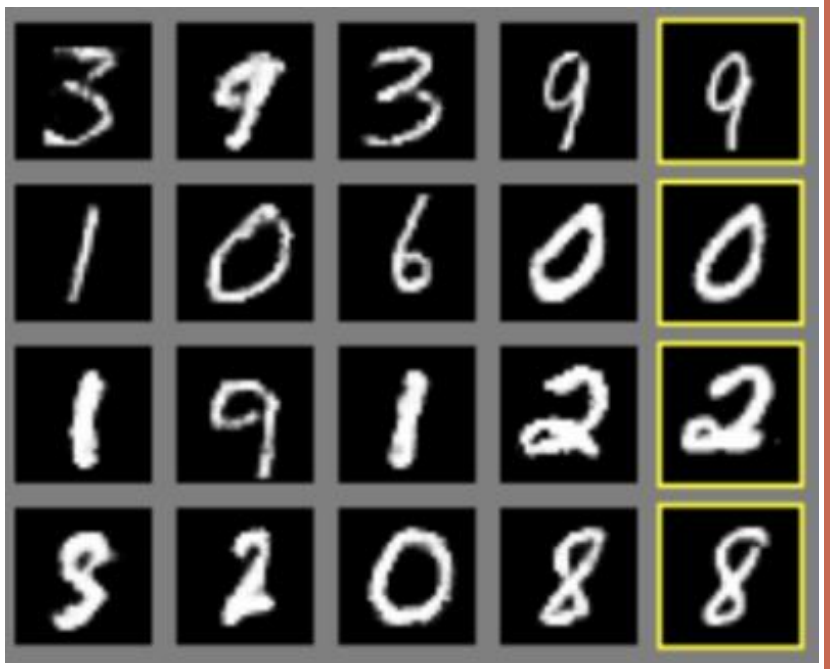
- They can generate samples in parallel, instead of using runtime proportional to the dimensionality of x.
- The design of the generator function has very few restrictions.
- No Markov chains are needed.
- No variational bound is needed, and specific model families usable within the GAN framework are already known to be universal approximators, so GANs are already known to be asymptotically consistent.
- GANs are subjectively regarded as producing better samples than other methods.

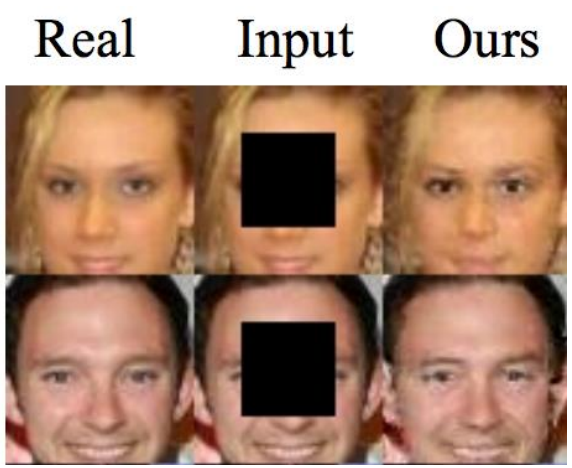
# REGARDING GANS…

At the same time, GANs have taken on a new disadvantage: training them requires finding the Nash equilibrium of a game, which is a more difficult problem than optimizing an objective function.

# EXPECTED APPLICATIONS OF GENERATIVE MODELS

- Data augmentation/ completion
  - Compensate data imbalance in classification
  - Semi-supervised learning
- Translation
  - image2image, text2image, label2image
  - Style Transfer
- Super Resolution
- **Part of AI:** Generation of never before seen objects or ideas (Music, Art, Text)



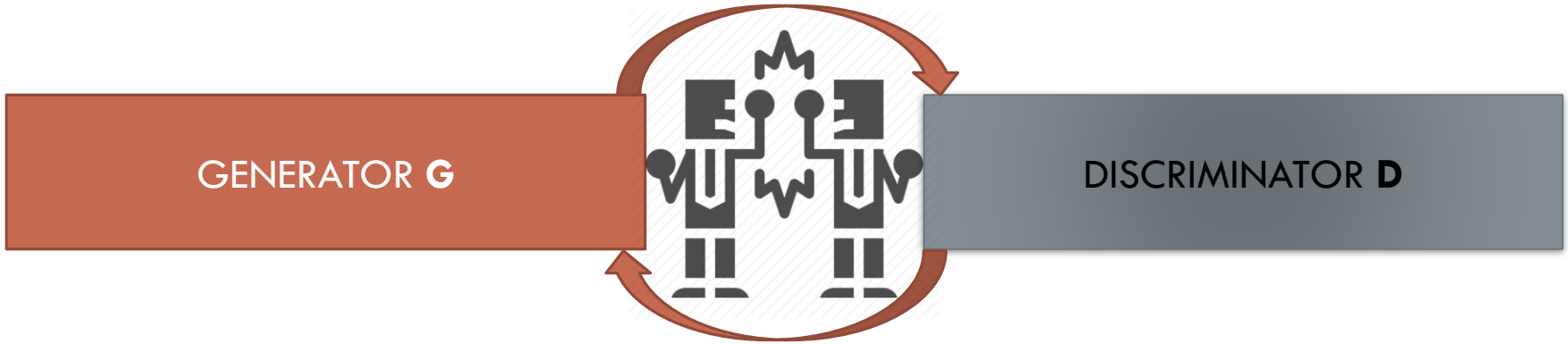


# GENERATIVE ADVERSARIAL NETS – 2014

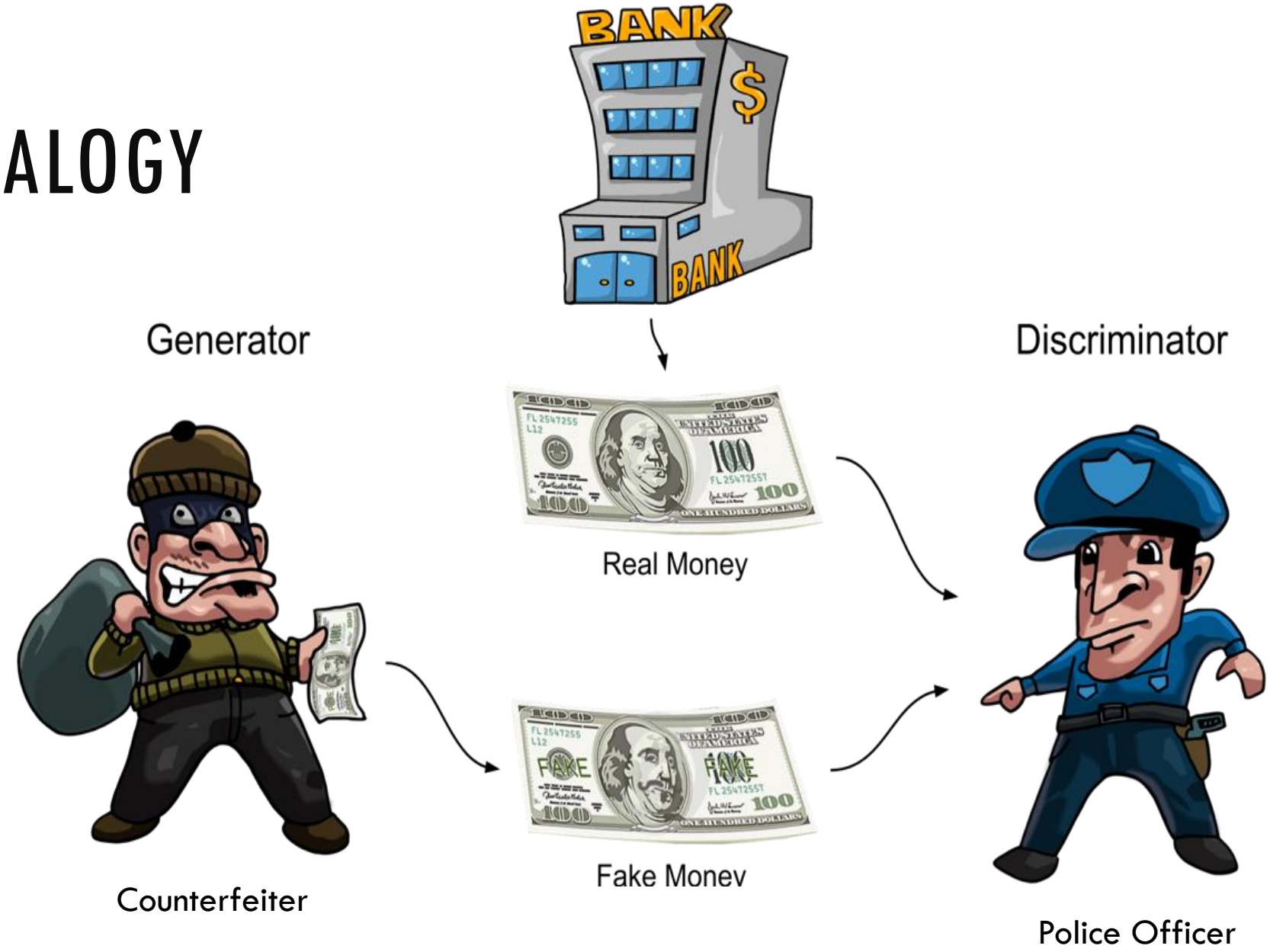
## IAN "GANFATHER" GOODFELLOW

Estimation of **generative** models through an adversarial process:

- Generative model ***G***: its goal is to capture the data distribution, generate samples and maximize the probability of D making a mistake.
- Discriminative model ***D***: its goal is to accurately estimate the probability of a sample coming from the training set and not G.

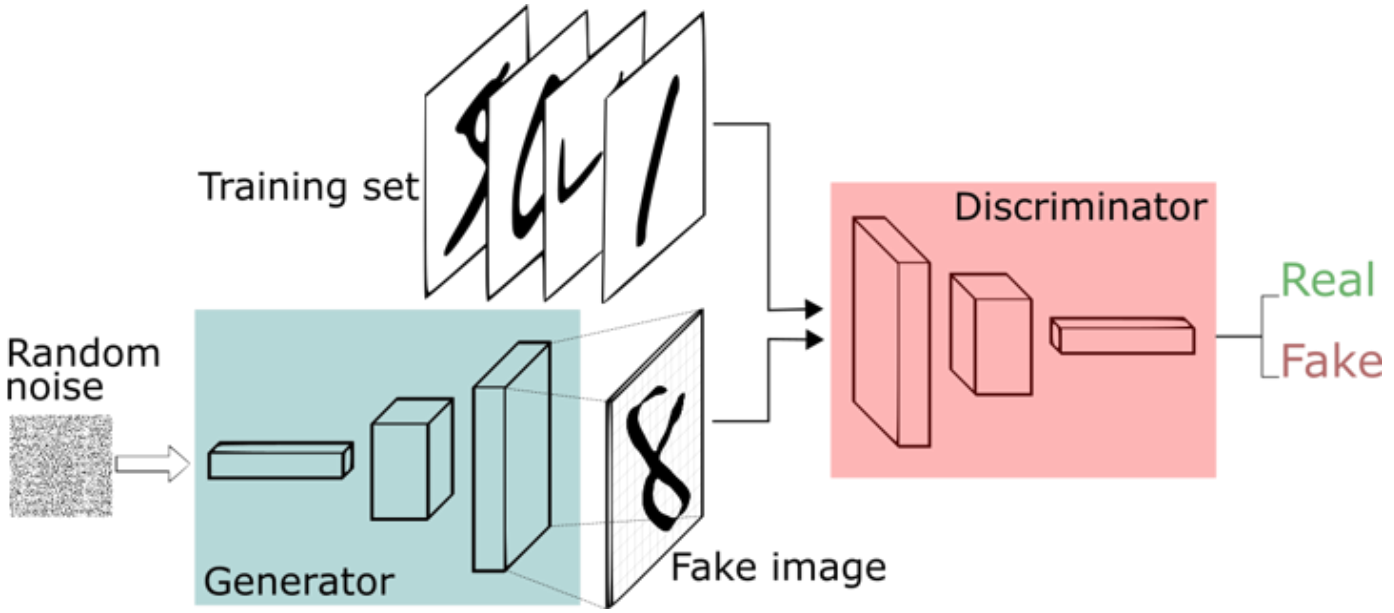


# ANALOGY


BANK
BANK
Generator
Discriminator
Real Money
Fake Money
Counterfeiter
Police Officer

# ADVERSARIAL NETWORK

- Both Generator and Discriminator are multilayered perceptron
- Generator **G** takes a random noise input $z \sim p(z)$ and generates a fake sample $G(z)$
- Discriminator **D** tries to distinguish between the fake samples $G(z)$ from real samples of the training set.



# VALUE FUNCTION

A Two-player minimax game in which:

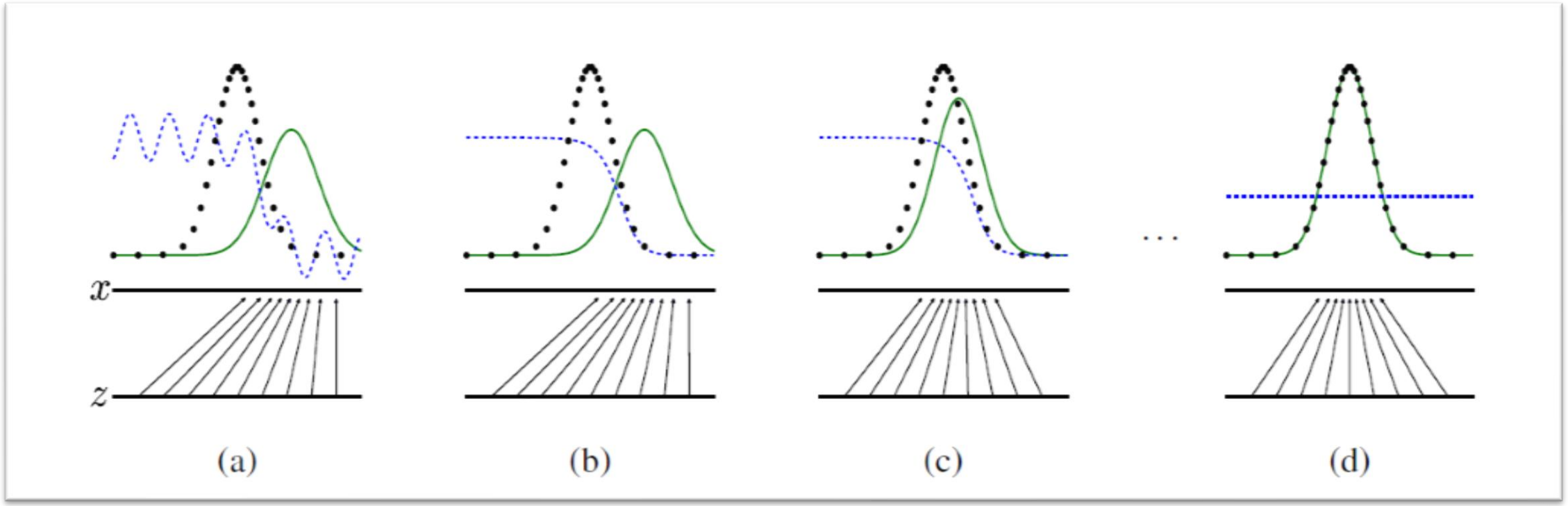
- We train **D** to maximize the possibility of correctible identified that a sample came from the dataset $(D(x))$ and not **G**.
- We train **G** to minimize the possibility of its samples being detected, in mathematical terms, we want to minimize $1 - D(G(z))$

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

- We can also train **G** to maximize $D\big(G(z)\big)$, this provides with stronger gradients early in training and keeps the same dinamic.

# IDEAL ADVERSARIAL NETWORK BEHAVIOR

$$P_g \approx P_{data} \qquad D\big(G(z)\big) = \frac{1}{2}$$




**Data generating distribution (black, dotted line)** **Discriminative distribution (blue, dashed line)** **Generative distribution (green, solid line)**

# FINAL NOTES ON GAN

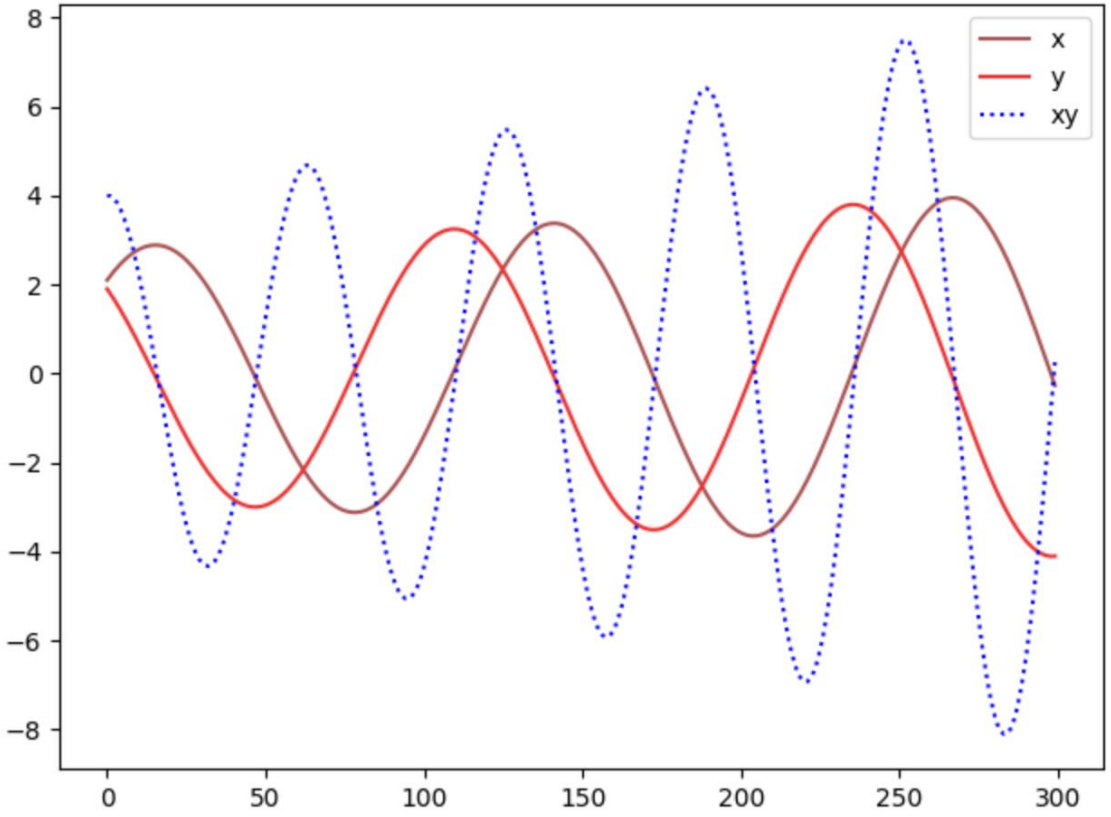
- The training scheme must be modified, **one shouldn't train D and G at the same pace.** There must be synchronization between D's and G's training.

- **Not a clear notion of how to evaluate generative models,** they use Gaussian Parzen window on samples and evaluated log-likelihood, but this is not a reliable metric.

- Future work: Conditional generative model, semi supervised learning, improvements in the training stability of GANs

# HINDSIGHT ON GANS

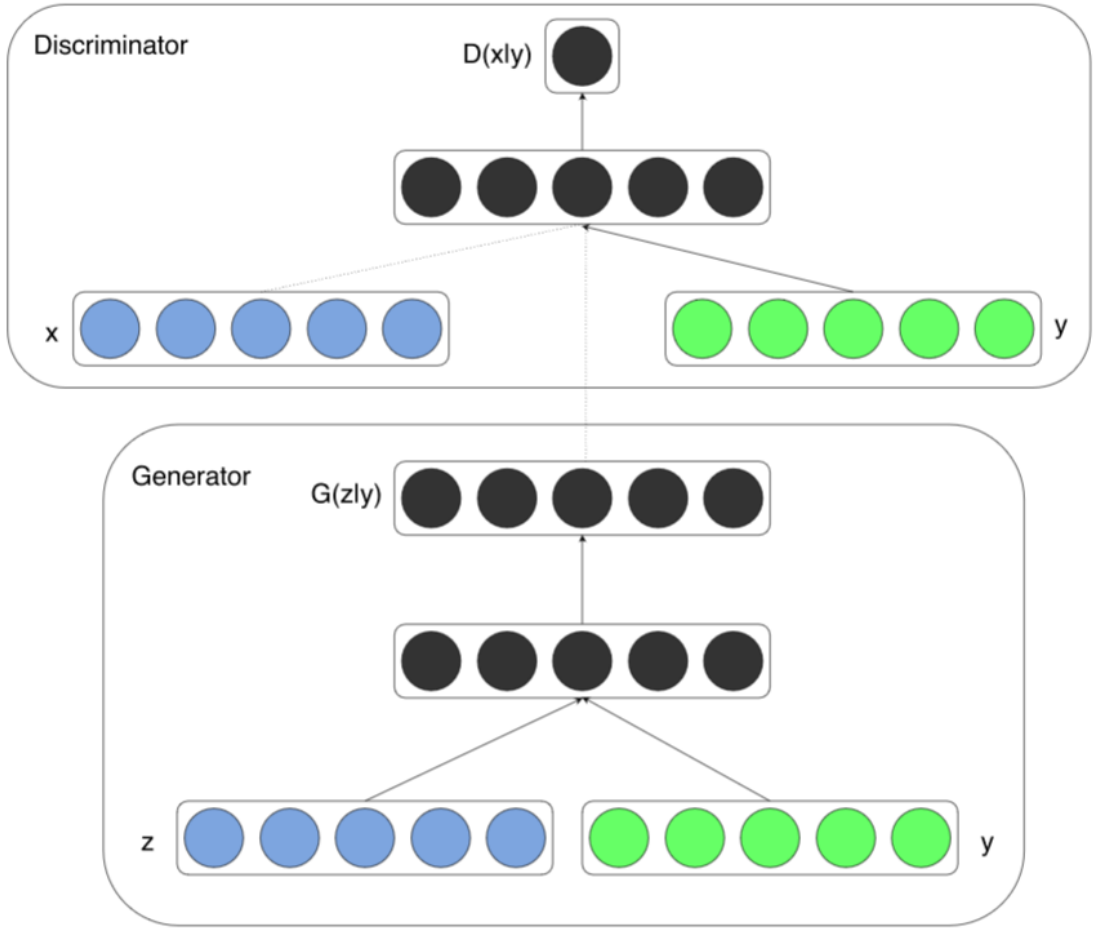
- **Training GANs can be considered as a non-convergence problem**
  - Minimax non-cooperative game
- Known Problems
  - Mode Collapse: Generator limits its output
  - Vanishing gradients: Discriminator cannot give relevant info to the Generator
- Future work has focused on finding new loss functions
  - Some based on mathematics, some on intuition and visual results.

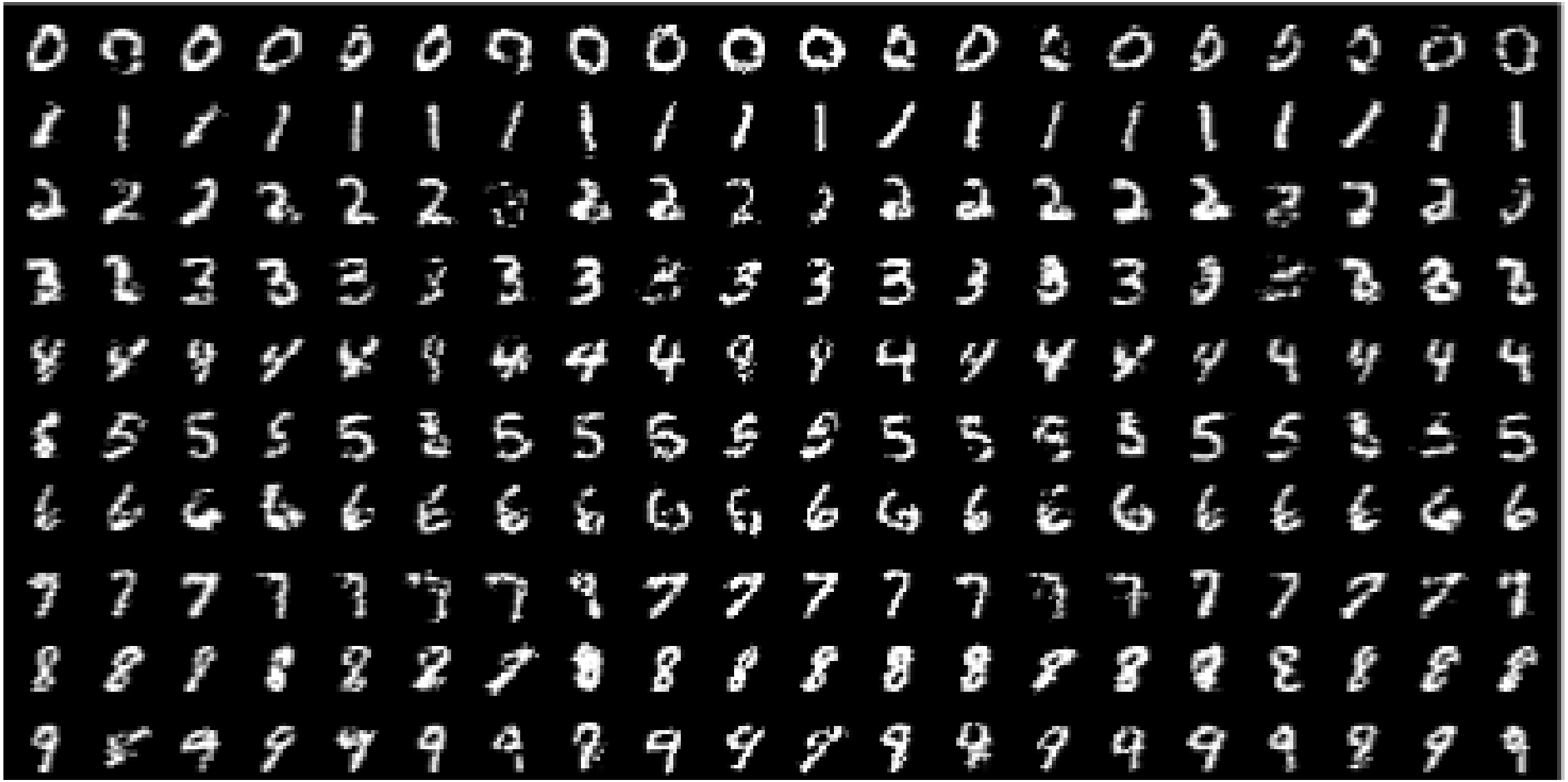


# HISTORY

# CONDITIONAL GAN (CGAN) – 2014

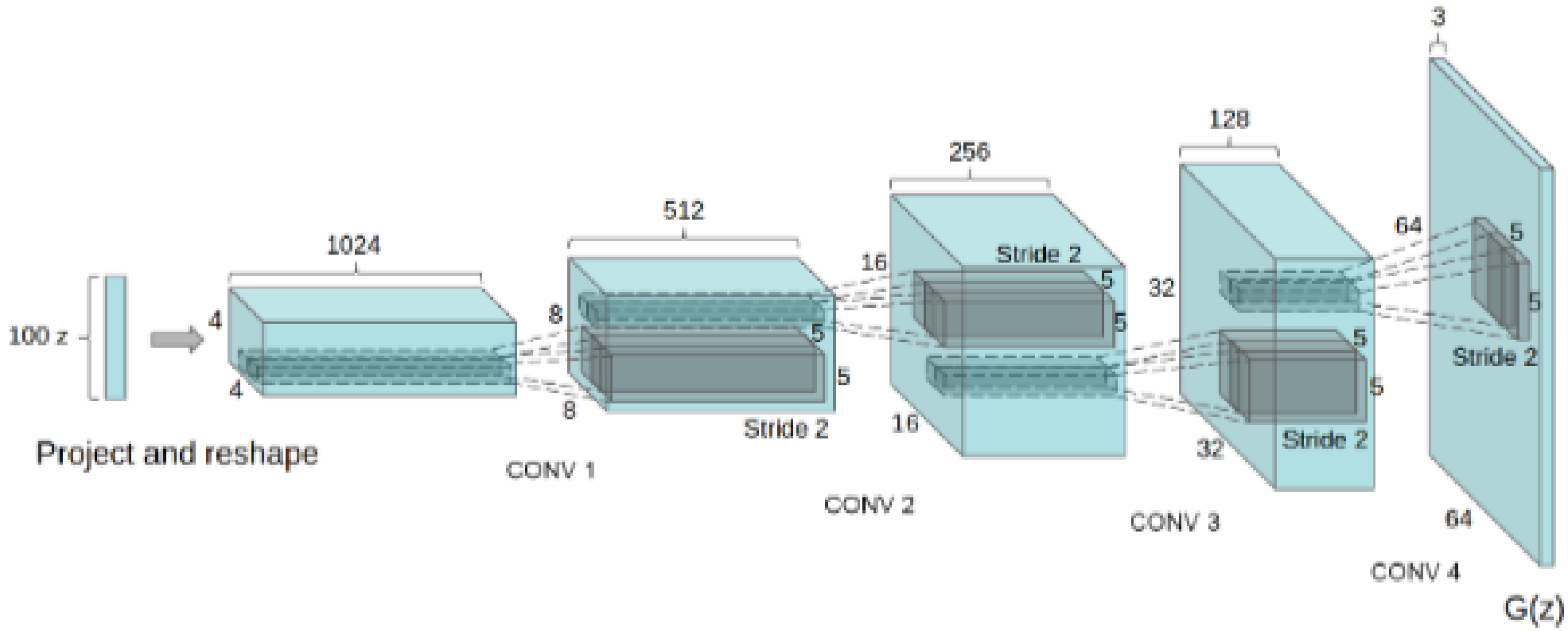
- Introduction of a conditional version of GANs. We can better control the output of **G**
- Both the generator **G** and discriminator **D** are "feed" with a label (or additional modal data) **y** in addition to their corresponding input.

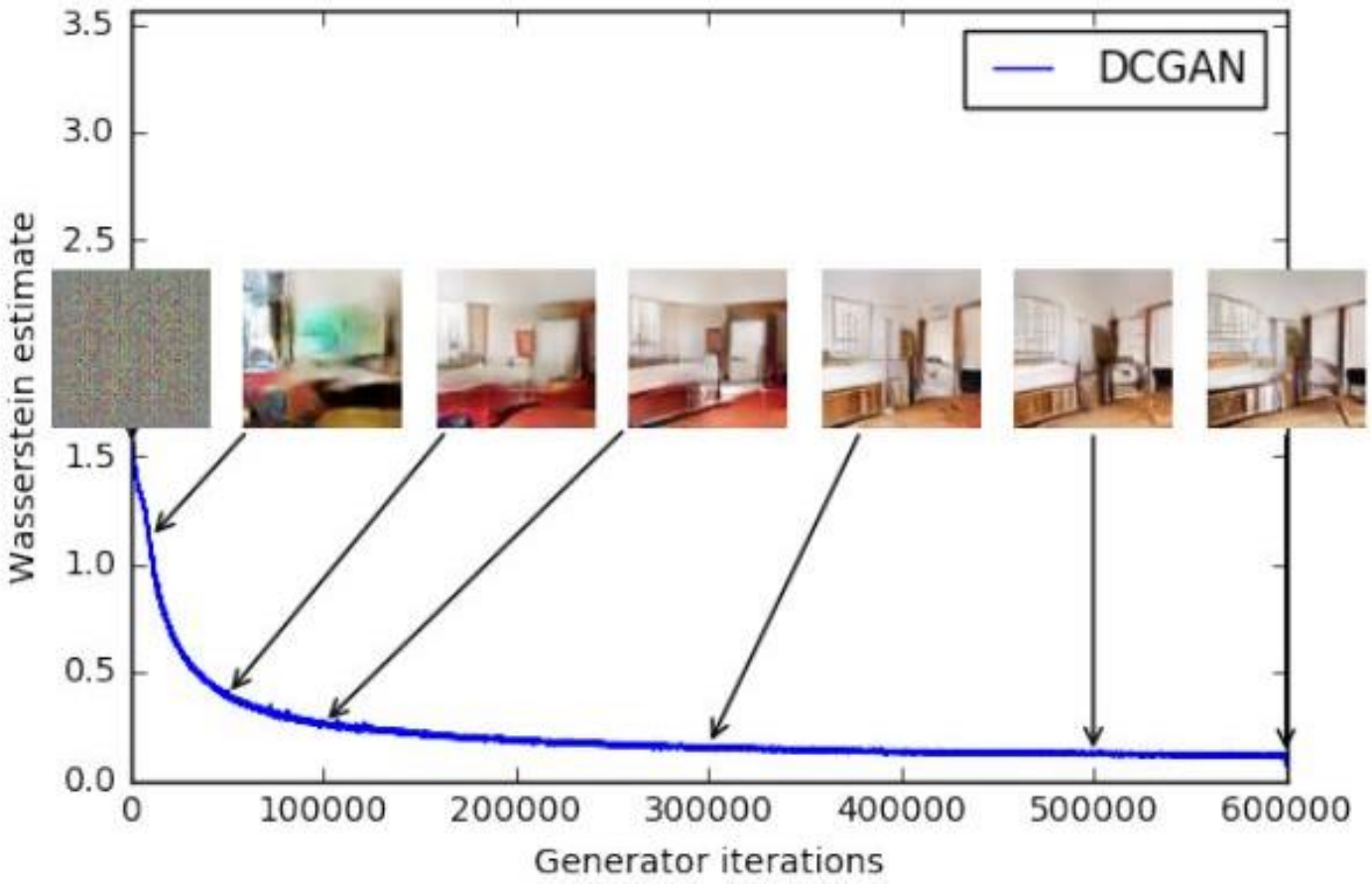




# DEEP CONVOLUTION GAN (DCGAN) – 2015

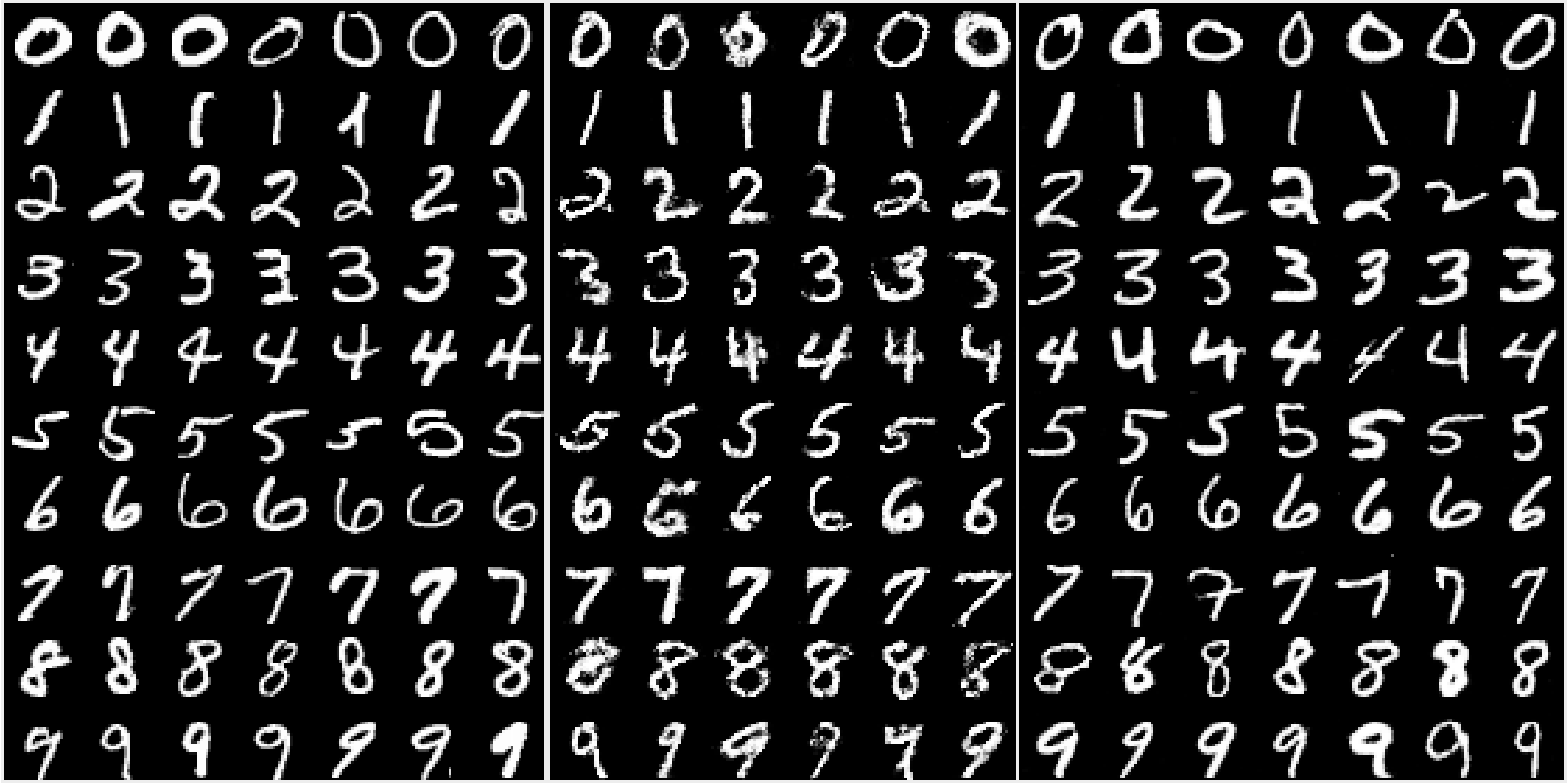
- Introduction of a class of CNNs for GANs with a set of architectural constrains for successful training.
- Most GANs nowadays are loosely based on the DCGAN architecture






# SET OF ARCHITECTURAL CONSTRAINS

- Replace pooling layers for strided convolution layers.
- Batch normalization
- Remove fully connected hidden layers
- ReLU activation in each layer of **G**, except Output (Tanh)
- LeakyReLU activation in each layer of **D**

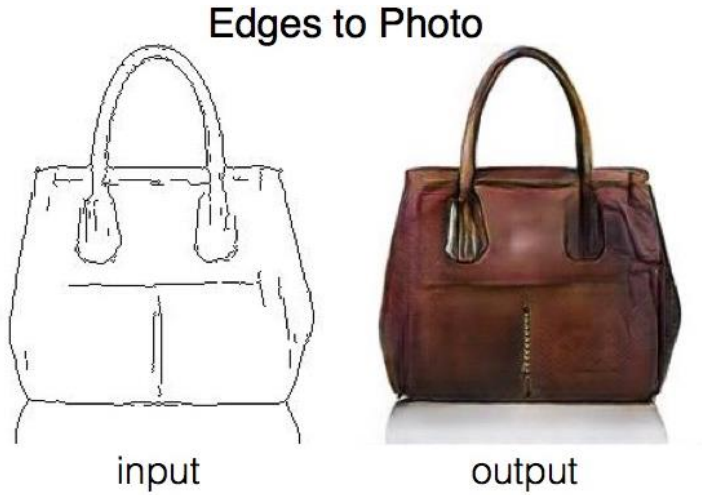

Groundtruth MNIST
GAN
DCGAN (ours)

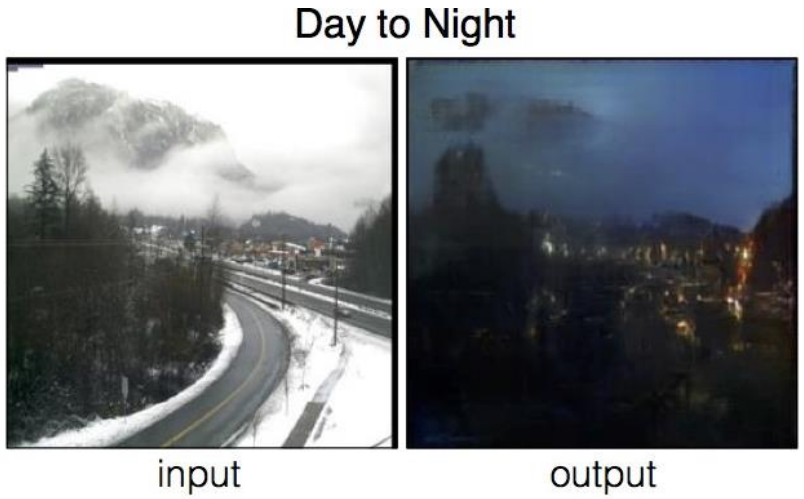
# PIX2PIX – 2017

- Image-to-Image Translation **framework** with cGANs
- GANs Goal: Learn mapping from **z** to **y** ($\boldsymbol{G}: \boldsymbol{z} \rightarrow \boldsymbol{y}$ )
- Goal: Learn mapping from image **x** and random noise **z** to output image **y** $\boldsymbol{G}: \{\boldsymbol{x}, \boldsymbol{z}\} \rightarrow \boldsymbol{y}$
- Additional L1-Loss:

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}[\|y - G(x,z)\|_1].$$

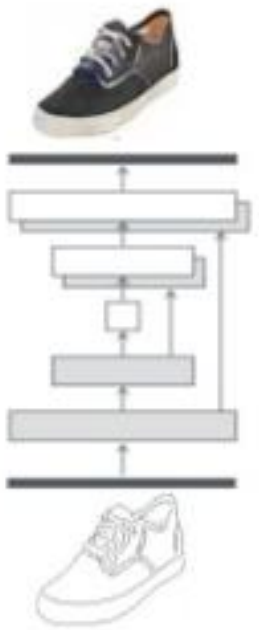
$$G^* = \arg\min_G \max_D \mathcal{L}_{cGAN}(G,D) + \lambda \mathcal{L}_{L1}(G).$$



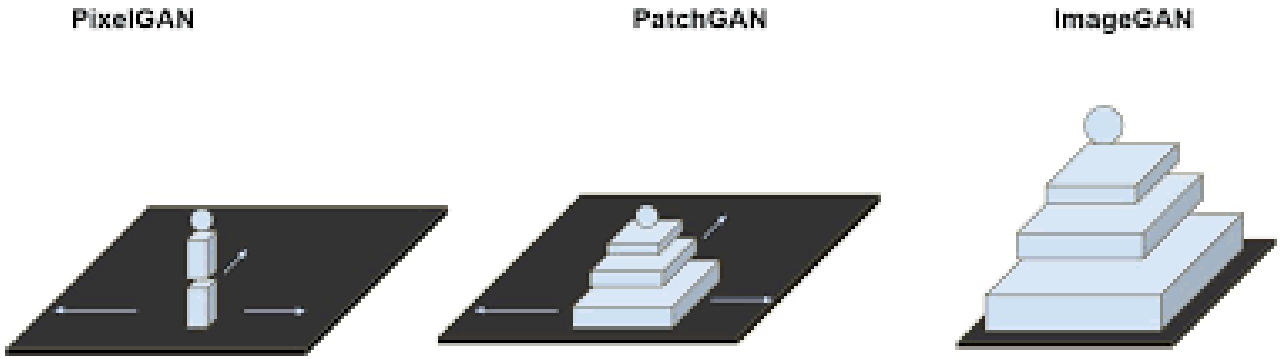
# PIX2PIX – 2017

Generator: Residual U-Net



Design to maintain underlying structure between input and output

Discriminator: Convolutional PatchGAN



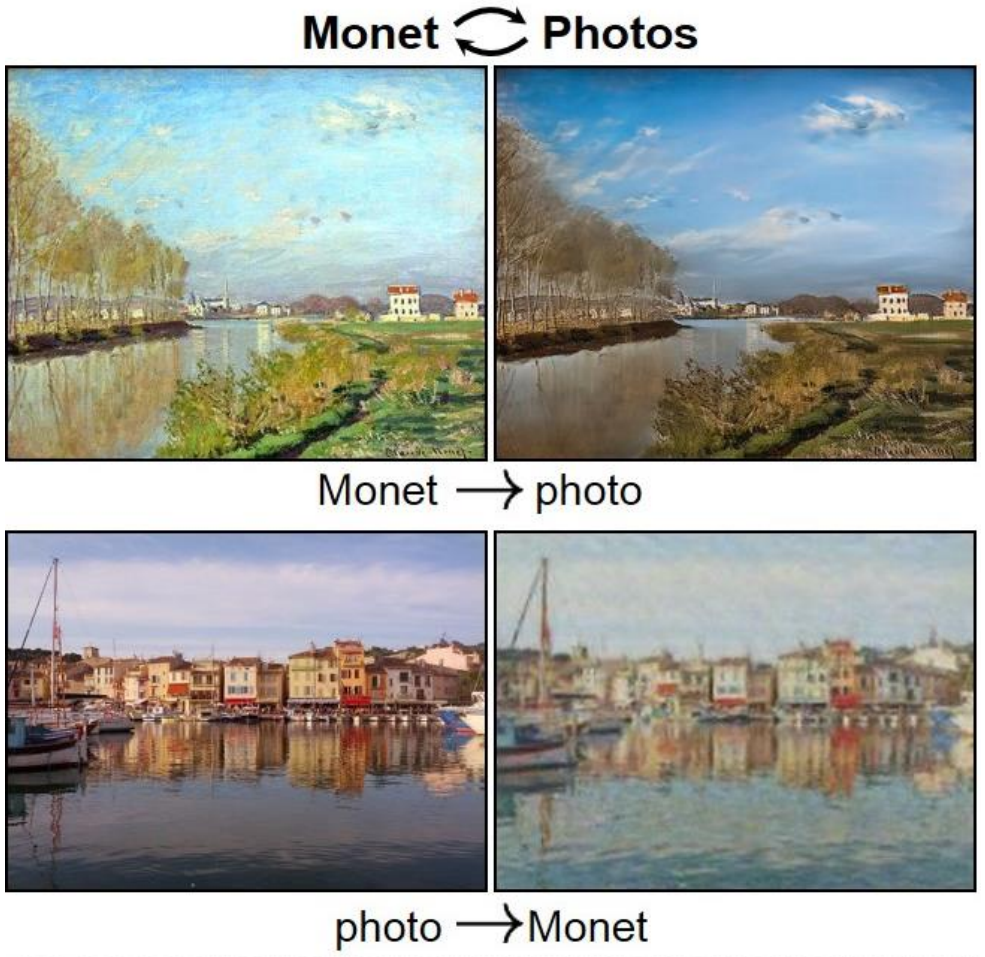

Penalizes structure at the level of image patches. Considered a Morkov Discriminator


Input
Output
Input
Output
Input
Output

# CYCLEGAN – 2017

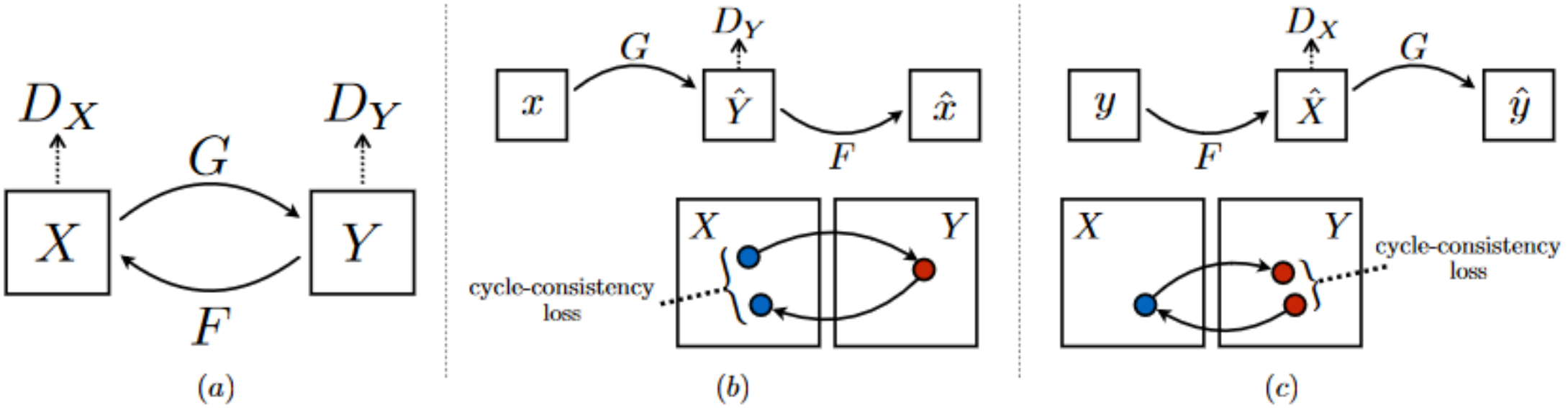
- Unpaired Image-to-Image Translation using Cycle-Consistent GANs
- Problem with Pix2Pix: Need of paired examples in each domain X and Y.
- Goal: Learn mapping $\boldsymbol{G}: \boldsymbol{X} \rightarrow \boldsymbol{Y}$, but also $\mathbf{F}: \boldsymbol{Y} \rightarrow \boldsymbol{X}$. Such that $F\big(G(x)\big) \approx X$ and vice versa.
- Approach: Add Cycle Consistency and Adversarial Loss for each mapping

$$\begin{aligned}\mathcal{L}(G, F, D_X, D_Y) =& \mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) \\ &+ \mathcal{L}_{\text{GAN}}(F, D_X, Y, X) \\ &+ \lambda \mathcal{L}_{\text{cyc}}(G, F),\end{aligned}$$



# CYCLEGAN – 2017

**Cycle Consistency:** Given two domains X and Y, and the corresponding mappings G and F, we have forward loss: $F\big(G(x)\big) \approx x$ and backwards loss $G\big(F(y)\big) \approx y$




$$\mathcal{L}_{\text{cyc}}(G, F) = \mathbb{E}_{x \sim p_{\text{data}}(x)}[\|F(G(x)) - x\|_1] + \mathbb{E}_{y \sim p_{\text{data}}(y)}[\|G(F(y)) - y\|_1].$$


Input
Monet
Van Gogh
Cezanne
Ukiyo-e

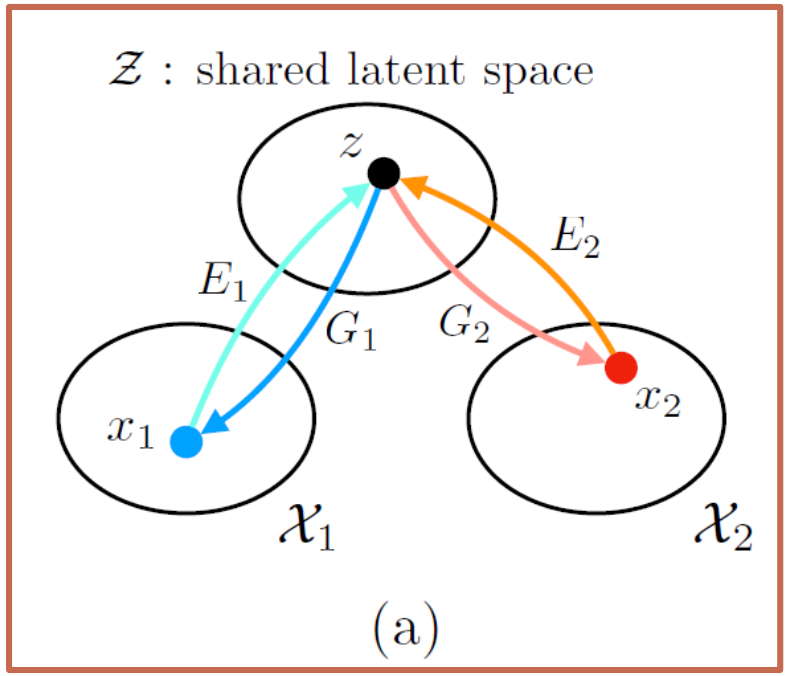
# UNIT – 2017

- Many image-to-image translation problems are mapping an image to a corresponding image
- Learn a joint distribution  of images in different domains
- Unsupervised setting → the two sets consist of images from **two marginal** distributions in two **different domains**, so the **task** is to **infer the joint distribution** using these images.
- **Infinite** set of possible joint distributions, so no assumptions can be made

# UNIT – ASSUMPTIONS

- **Shared latent space** where two images from two different sets can be mapped to same **latent representation**
- For any given pair images $(x_1, x_2)$ there exists a shared latent code $z$ in a shared-latent space $\mathcal{Z}$
- Both images can be recovered from $z$

# UNIT – ASSUMPTIONS




- Consider the following functions:

$$\boldsymbol{E_1^*, E_2^*, G_1^*, G_2^*}$$

Such that given $(x_1, x_2)$ we have:

$$\boldsymbol{z = E_1^*(x_1) = E_2^*(x_2)}$$

And reversely we have:

$$\boldsymbol{x_1 = G_1^*(z), x_2 = G_2^*(z)}$$

So for this model we have that:

$$\boldsymbol{x_2 = F_{1\to2}^*(x_1) = G_2^*(E_1^*(x_1))}$$
$$\boldsymbol{x_1 = F_{2\to1}^*(x_2) = G_1^*(E_2^*(x_2)}$$

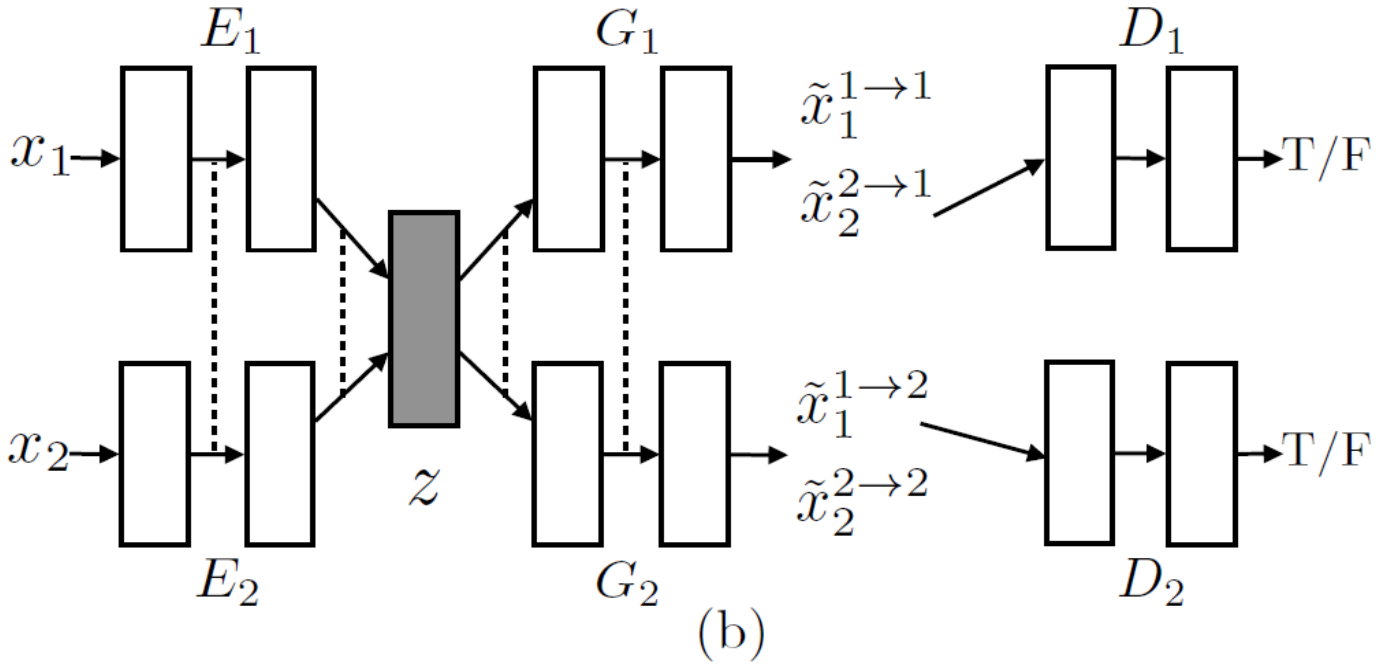
- For $F_{1\to2}^*(x_1)$ and $F_{2\to1}^*(x_2)$ it should exist a cycle-consistency constraint:

$$\boldsymbol{x_1 = F_{2\to1}^*\big(F_{1\to2}^*(x_1)\big), x_2 = F_{1\to2}^*\big(F_{2\to1}^*(x_2)\big)}$$

# UNIT – FRAMEWORK

- 6 subnetworks and two images from different domains
- **VAE** (Variational Autoencoders) are encoder-generator pair $\{E_1, G_1\}$
- We assume the components in the latent space $\mathcal{Z}$ are conditionally independent and Gaussian with unit variance.

- Encoder outputs mean vector $E_{\mu,1}(x_1)$ and distribution is given by $q_1(z_1|x_1) \equiv \mathcal{N}\left(z_1|E_{\mu,1}(x_1), I\right)$ where $I$ is and identity matrix
- Reconstructed image $\tilde{x}_1^{1\to1} = G_1\left(z_1 \sim q_1(z_1|x_1)\right)$
- The same behaviour occurs to $x_2$
- Weight sharing on last two layers



# UNIT – TRAINING

Objective Function

$$\min_{E_1,E_2,G_1,G_2} \max_{D_1,D_2} \mathcal{L}_{\text{VAE}_1}(E_1,G_1) + \mathcal{L}_{\text{GAN}_1}(E_1,G_1,D_1) + \mathcal{L}_{\text{CC}_1}(E_1,G_1,E_2,G_2)$$
$$\mathcal{L}_{\text{VAE}_2}(E_2,G_2) + \mathcal{L}_{\text{GAN}_2}(E_2,G_2,D_2) + \mathcal{L}_{\text{CC}_2}(E_2,G_2,E_1,G_1).$$

VAE Objective Function

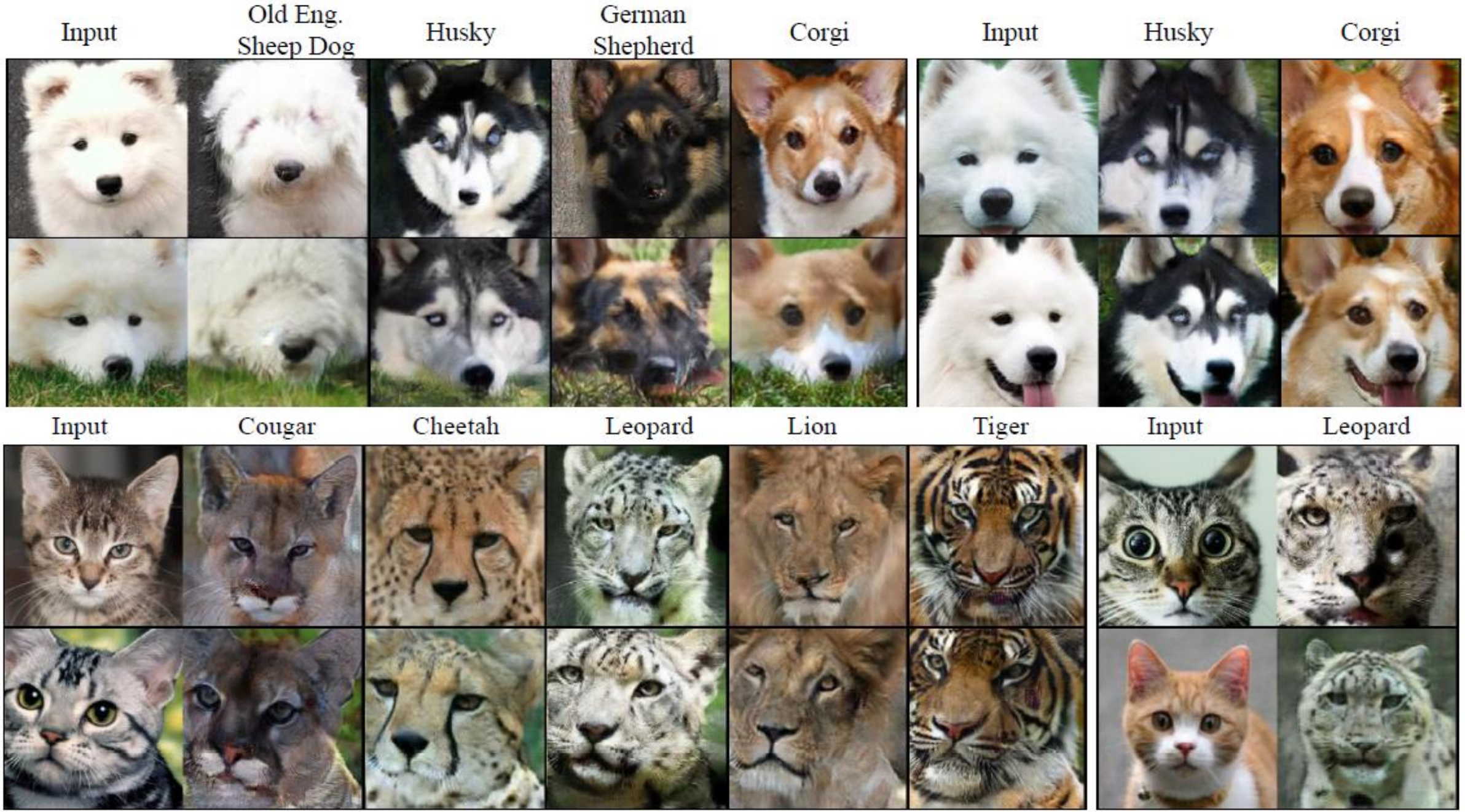
$$\mathcal{L}_{\text{VAE}_1}(E_1,G_1) = \lambda_1 \text{KL}(q_1(z_1|x_1)||p_\eta(z)) - \lambda_2 \mathbb{E}_{z_1\sim q_1(z_1|x_1)}[\log p_{G_1}(x_1|z_1)]$$
$$\mathcal{L}_{\text{VAE}_2}(E_2,G_2) = \lambda_1 \text{KL}(q_2(z_2|x_2)||p_\eta(z)) - \lambda_2 \mathbb{E}_{z_2\sim q_2(z_2|x_2)}[\log p_{G_2}(x_2|z_2)].$$

GAN Objective Function

$$\mathcal{L}_{\text{GAN}_1}(E_1,G_1,D_1) = \lambda_0 \mathbb{E}_{x_1\sim P_{\mathcal{X}_1}}[\log D_1(x_1)] + \lambda_0 \mathbb{E}_{z_2\sim q_2(z_2|x_2)}[\log(1 - D_1(G_1(z_2)))]$$
$$\mathcal{L}_{\text{GAN}_2}(E_2,G_2,D_2) = \lambda_0 \mathbb{E}_{x_2\sim P_{\mathcal{X}_2}}[\log D_2(x_2)] + \lambda_0 \mathbb{E}_{z_1\sim q_1(z_1|x_1)}[\log(1 - D_2(G_2(z_1)))].$$

Cycle Consistency Objective Function

$$\mathcal{L}_{\text{CC}_1}(E_1,G_1,E_2,G_2) = \lambda_3 \text{KL}(q_1(z_1|x_1)||p_\eta(z)) + \lambda_3 \text{KL}(q_2(z_2|x_1^{1\to 2}))||p_\eta(z)) -$$
$$\lambda_4 \mathbb{E}_{z_2\sim q_2(z_2|x_1^{1\to 2})}[\log p_{G_1}(x_1|z_2)]$$
$$\mathcal{L}_{\text{CC}_2}(E_2,G_2,E_1,G_1) = \lambda_3 \text{KL}(q_2(z_2|x_2)||p_\eta(z)) + \lambda_3 \text{KL}(q_1(z_1|x_2^{2\to 1}))||p_\eta(z)) -$$
$$\lambda_4 \mathbb{E}_{z_1\sim q_1(z_1|x_2^{2\to 1})}[\log p_{G_2}(x_2|z_1)].$$


Input
Old Eng. Sheep Dog
Husky
German Shepherd
Corgi
Input
Husky
Corgi
Input
Cougar
Cheetah
Leopard
Lion
Tiger
Input
Leopard

# MUNIT – 2018 – PREVIOUS MODEL PROBLEMS

- In many scenarios cross domain mapping of interest is multidomain.
- Current models assume deterministic or unimodal mapping
- Fail to capture full distribution of the posible outputs, even if model is stochastic by injecting noise networks learns to ignore it.

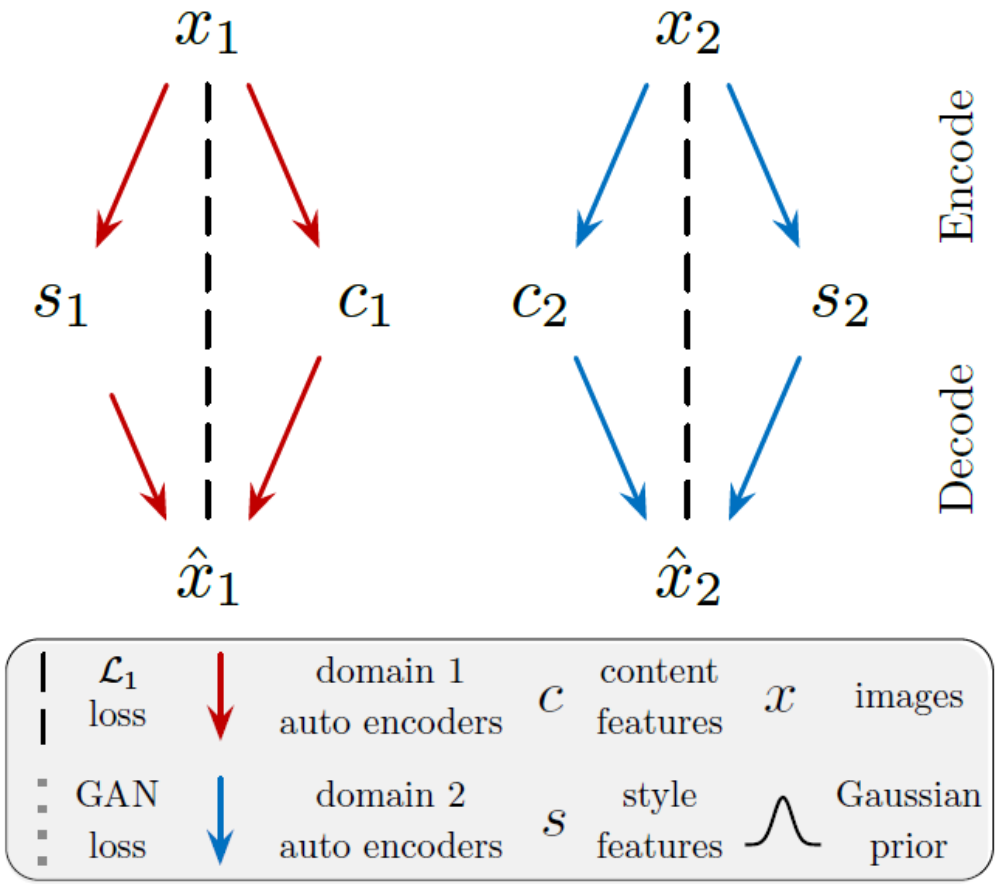
# MUNIT – ASSUMPTIONS

- Let $x_1 \in \mathcal{X}_1$ and $x_2 \in \mathcal{X}_2$, samples from two marginal distributions $p(x_1)$ and $p(x_2)$, with not access to joint distribution.
- Partially shared latent space assumption:
  - → where each image $x_i \in \mathcal{X}_i$ is generated from a content latent code $c \in C$ that is shared by both domains.
  - → And style latent code $s_i \in \mathcal{S}$ that is specific to each domain.

$$x_1 = G_1^*(c, s_1), x_2 = G_2^*(c, s_2)$$

- Learn underlying generator and encoder functions with NN.

# MUNIT – 2018

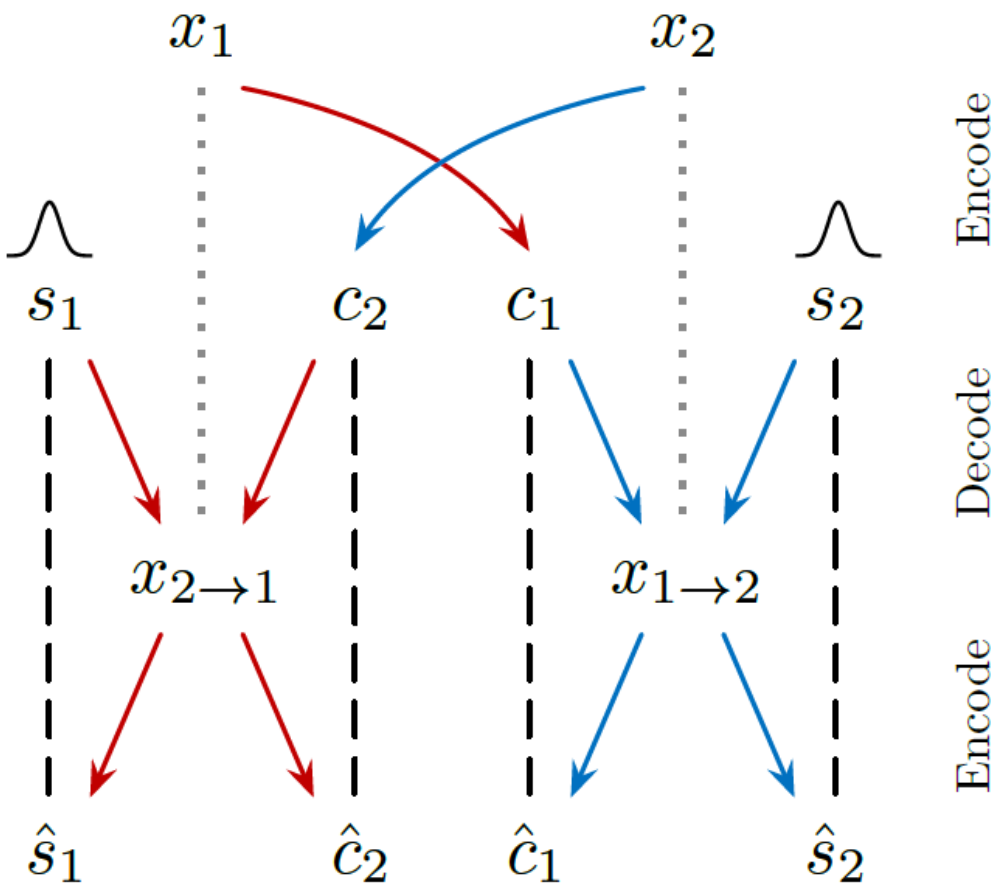



Latent code of each autoencoder is factorize in $c_i$ and style code $s_i$ where:

$$(c_i, s_i) = \left( E_i^c(x_i), E_i^s(x_i) \right) = E_i(x_i)$$

Image to image translation is performed by swapping encoder-decoder pair.

# MUNIT – 2018




To translate image $x_1 \in \mathcal{X}_1$ to $\mathcal{X}_2$, first content

$$c_1 = E_1^c(x_1)$$

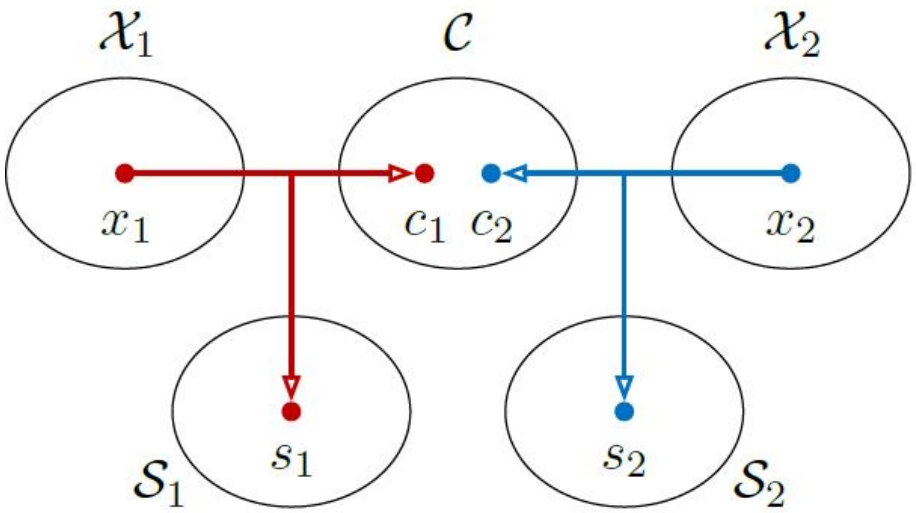
and randomly draw style latent code $s_2$ from

$$q(s_2) \sim \mathcal{N}(0, I).$$

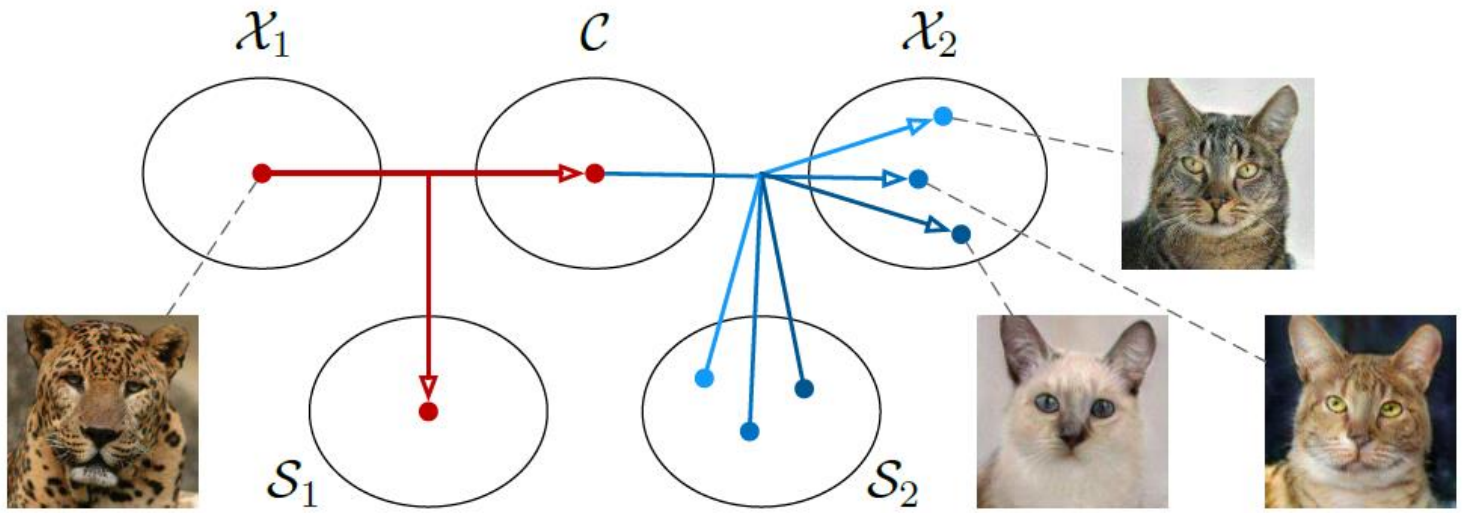
The final output is image is produced by $G_2$

Loss functions are use to encourage reconstruction (---)

# MUNIT – 2018




(a) Auto-encoding

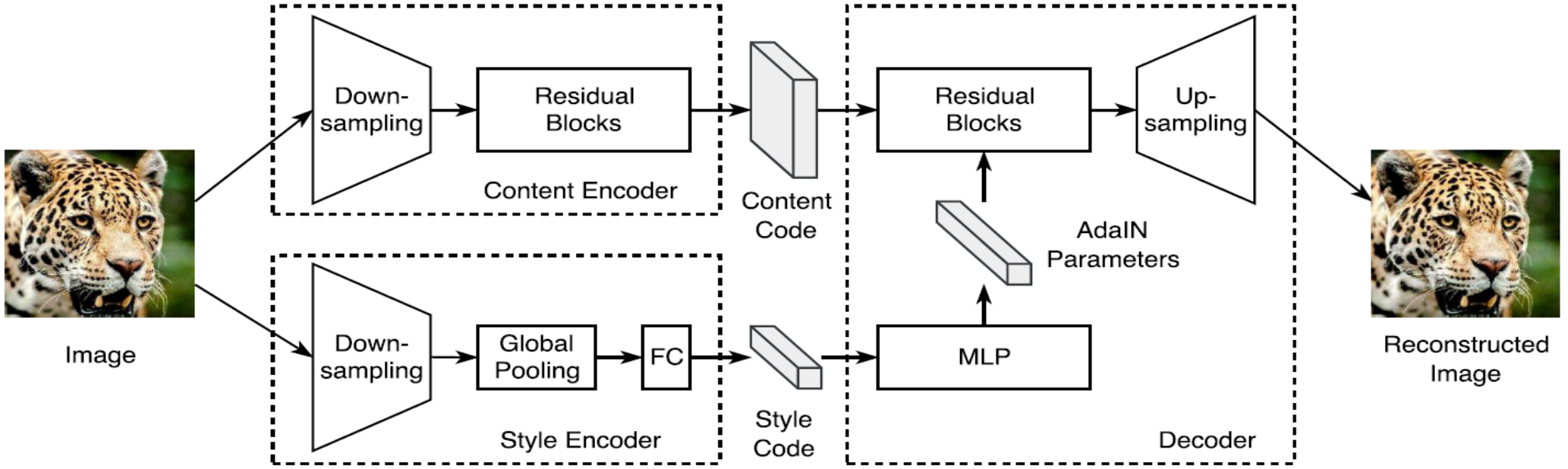



(b) Translation

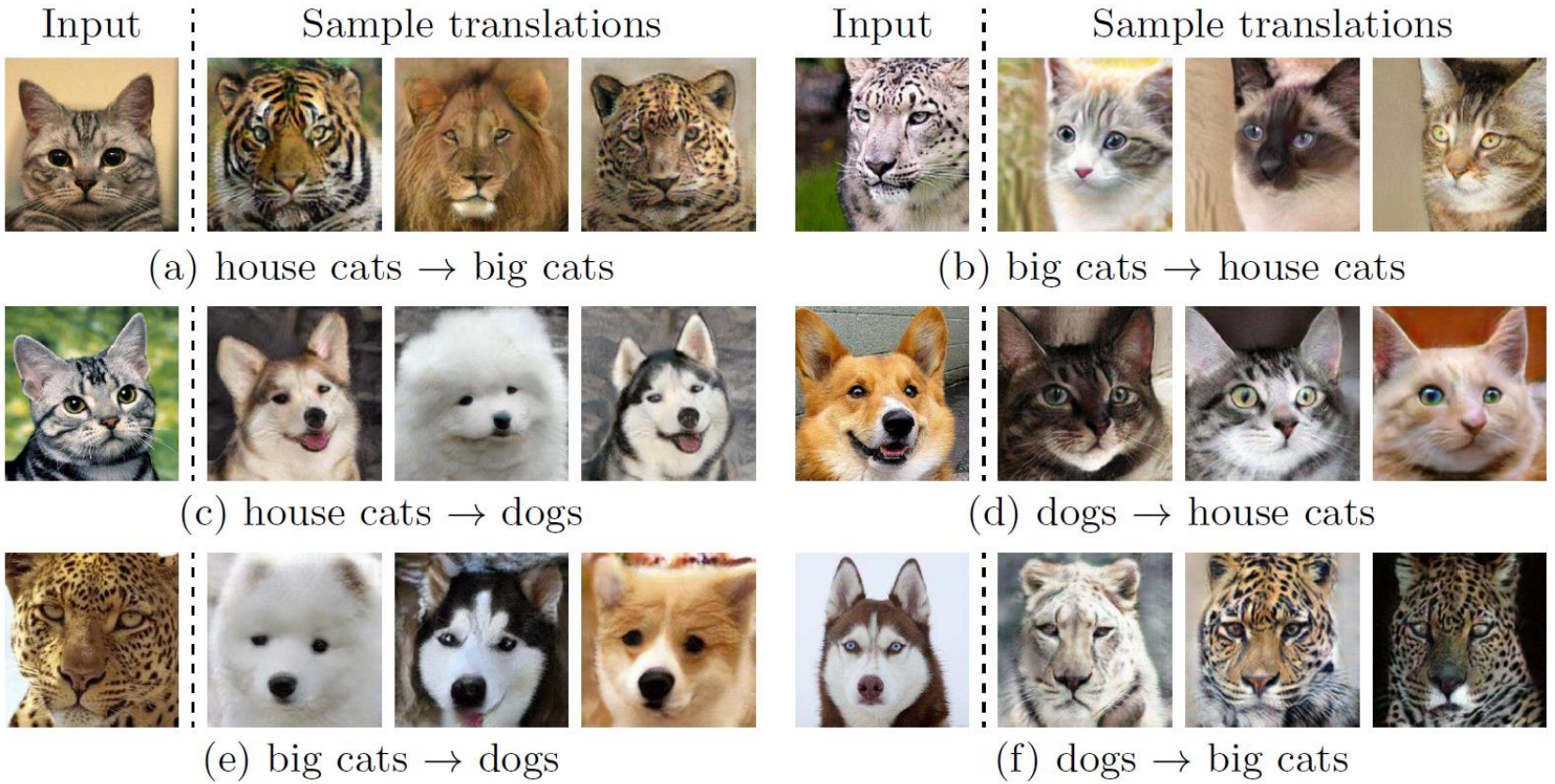
# MULTIMODAL UNSUPERVISED IMAGE TO IMAGE TRANSLATION

$$\mathrm{AdaIN}(z, \gamma, \beta) = \gamma \left( \frac{z - \mu(z)}{\sigma(z)} \right) + \beta$$

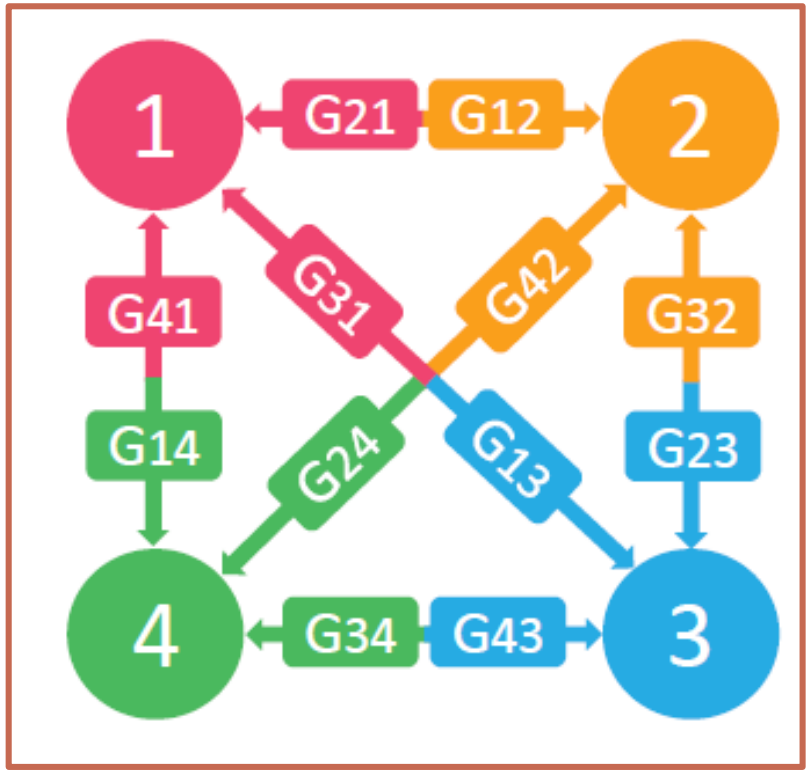
AdaIN (Adaptive Instance Normalization) parameters are generated by a multilayer perceptron (MLP)→ Linear Block.



# MUNIT – 2018

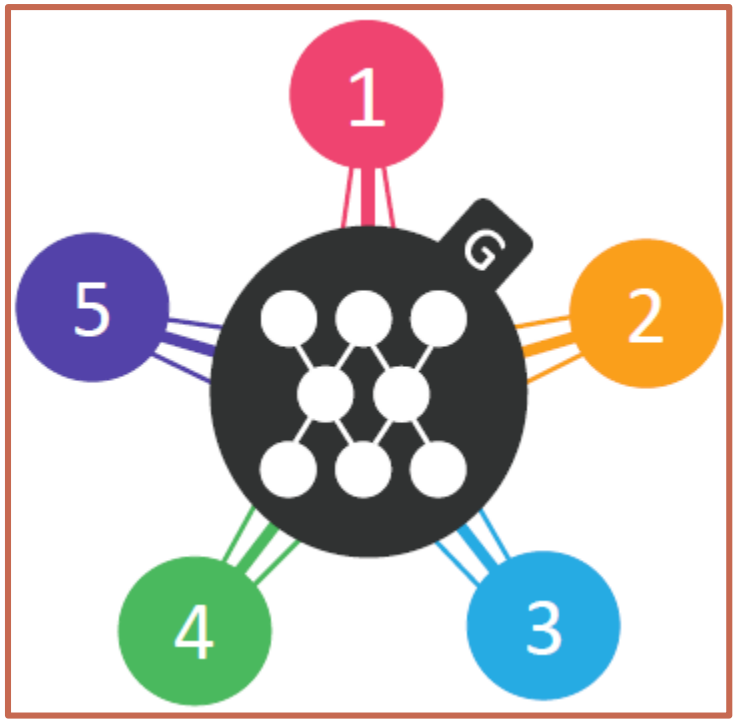

Input
Sample translations
Input
Sample translations
(a) house cats → big cats
(b) big cats → house cats
(c) house cats → dogs
(d) dogs → house cats
(e) big cats → dogs
(f) dogs → big cats

# STARGAN – 2018:




- Previous models required that among $k$ domains, $\boldsymbol{k \cdot (k-1)}$ generators had to be trained.
- There are global features such as face shapes that can be learned from images of all domains.
- Each generator does not fully utilize its training data.

# STAR GENERATIVE ADVERSARIAL NETWORKS




- Takes data from **all domains** and learns mapping from all domains using only **a single generator**
- Generator takes as input **image and domain** information (represented as a **Label**)
- AN SCALABLE GAN FRAMEWORK

# STARGAN – TRAINING INSIGHTS

- G is trained to translate an input image $x$ into an output image $y$ conditioned on the target domain $c$,$G(x, c) \rightarrow y$
- $c$ target domain label → is generated randomly so $G$
- Auxiliary classifier → allows single discriminator to control multiple domains.
- Discriminator produces probability distributions over both sources and domain labels $D: x \rightarrow \{D_{src}(x), D_{cls}(x)\}$

# STARGAN – TRAINING INSIGHTS

**Adversarial Loss:**

$$\mathcal{L}_{adv} = \mathbb{E}[\log D_{src}(x)] + \mathbb{E}_{x,c}\left[\log\left(1 - D_{src}\left(G(x,c)\right)\right)\right]$$

**Domain Classification Loss:**

$$\mathcal{L}_{cls}^{r} = \mathbb{E}_{x,c'}[-\log D_{cls}\,(c'|x)]\;(real)$$

$$\mathcal{L}_{cls}^{f} = \mathbb{E}_{x,c}[-\log D_{cls}\,(c|G(x,c))]\;(fake)$$

$D_{src(x)}$: probability distribution over sources given by $x$.
$G(x,c)$: image generated by $G$
$x$: input image
$c$: target domain label
$D_{cls}(c'|x)$: probability distribution over sources domain labels.
$D_{cls}(c|G(x,c))$: probability distribution over the image generated and the target domain labels
$\lambda_{cls}$ & $\lambda\_rec$: hyperparameters that control de importance.

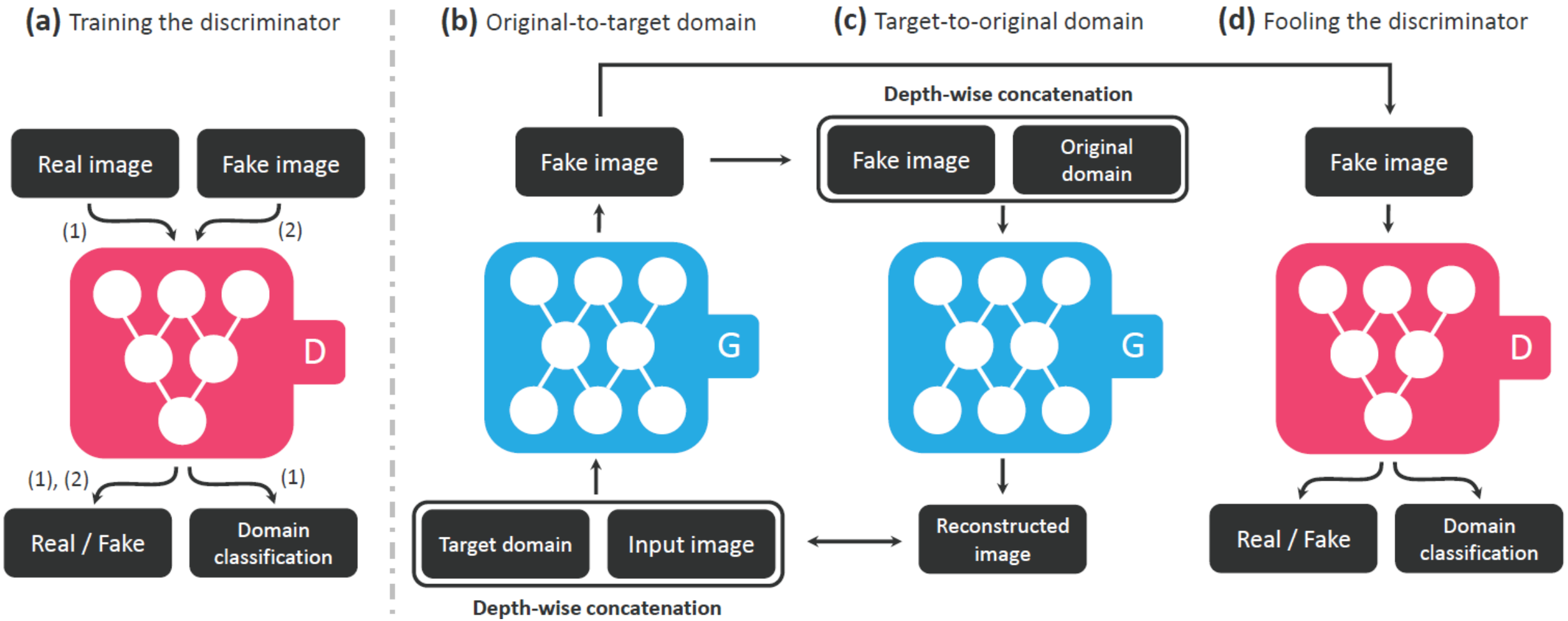
**Reconstruction Loss:**

$$\mathcal{L}_{rec} = \mathbb{E}_{x,c,c'}\left[\left|\left|x - G(G(x,c),c')\right|\right|_1\right]$$

## Objective function to minimize:

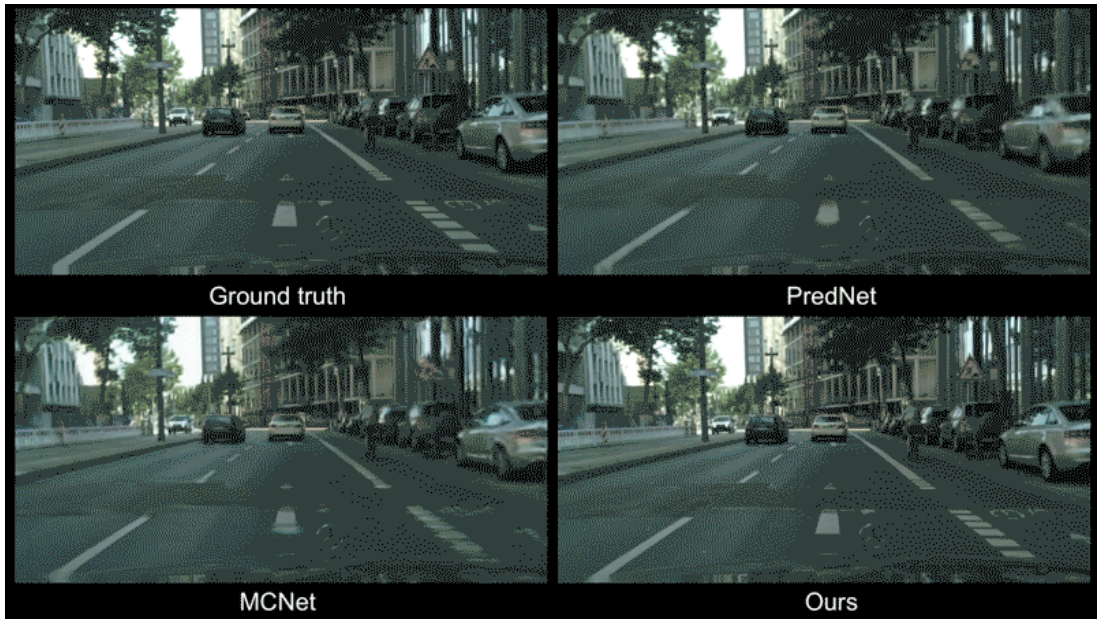
$$\mathcal{L}_D = -\mathcal{L}_{adv} + \lambda_{cls}\,\mathcal{L}_{cls}^{r}\,, \qquad \mathcal{L}_G = \mathcal{L}_{adv} + \lambda_{cls}\mathcal{L}_{cls}^{f} + \lambda_{rec}\mathcal{L}_{rec}$$

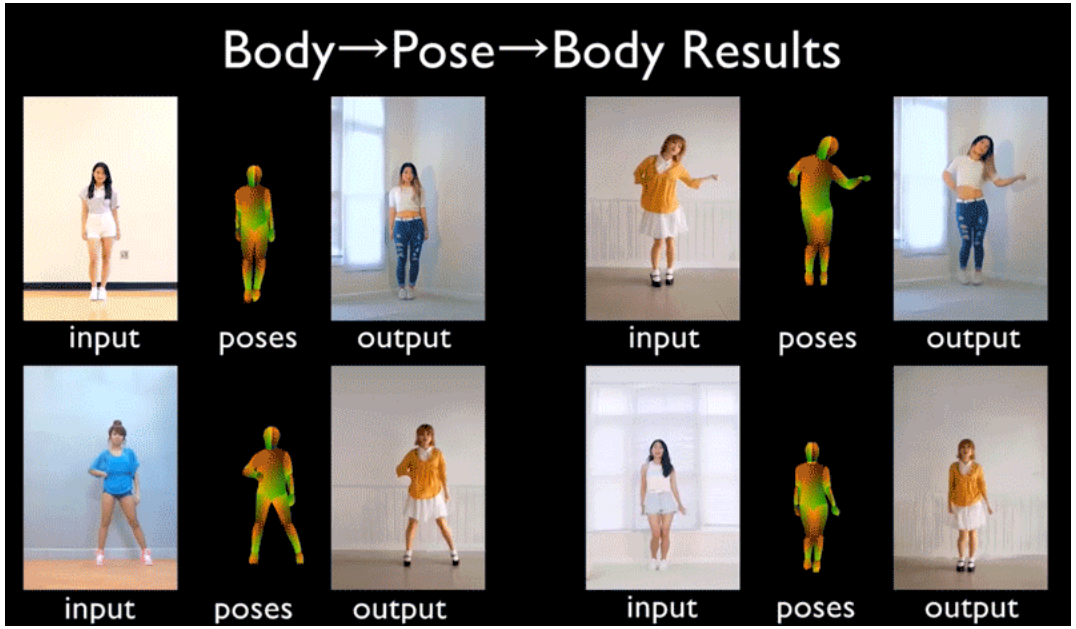
# STARGAN – TRAINING INSIGHTS



# RECENT WORK & OTHER APPLICATIONS

# VIDEO-TO-VIDEO SYNTHESIS

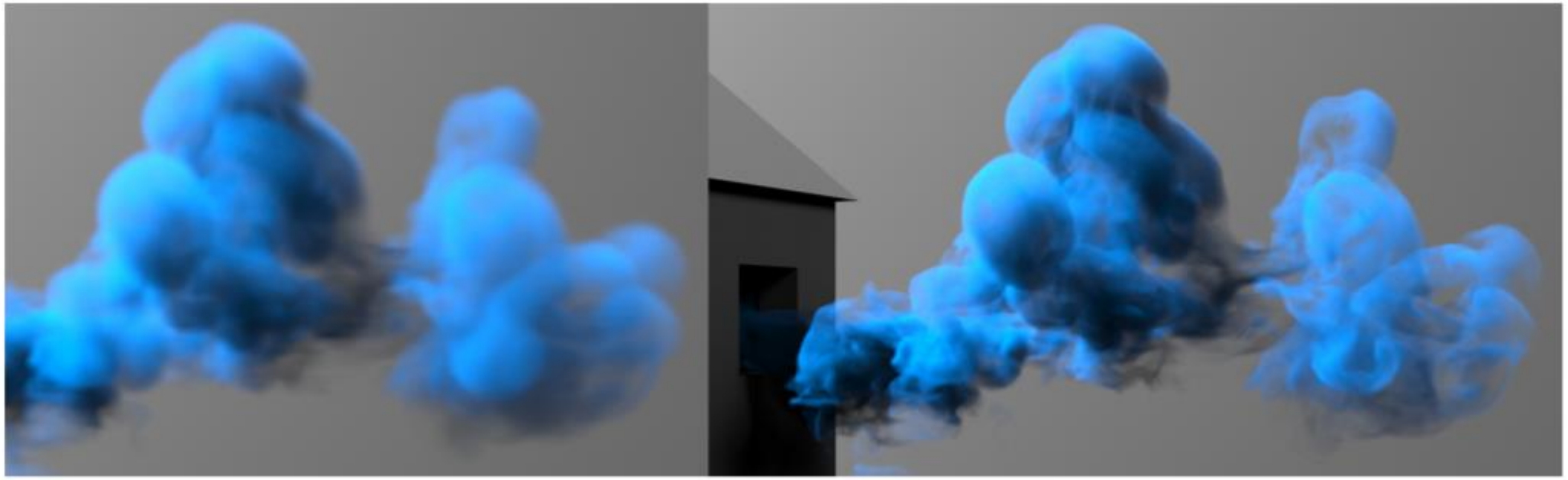



https://www.youtube.com/watch?v=ayPqjPekn7g

# TEMPORAL COHERENT & SUPER-RESOLUTION

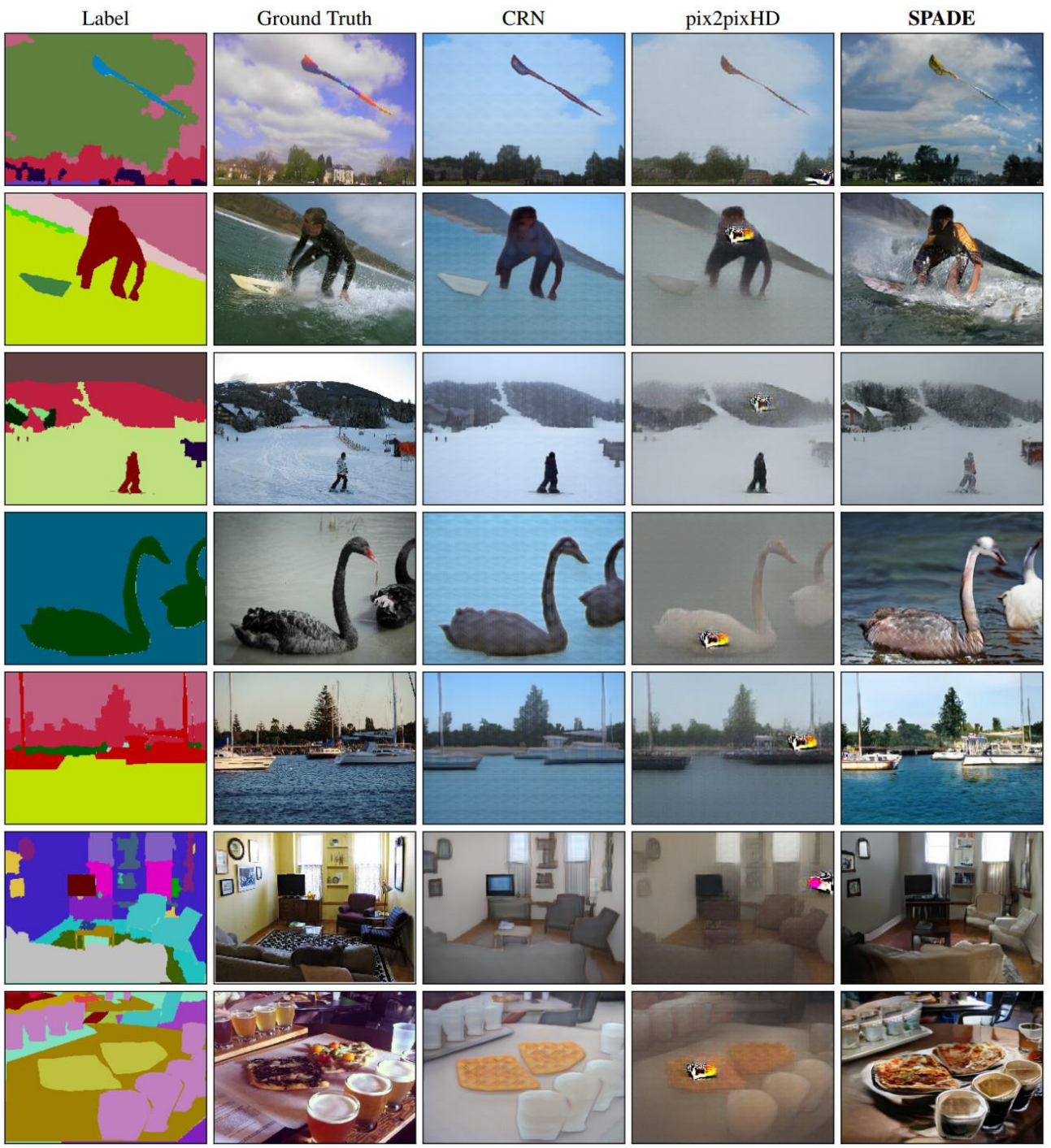
tempoGAN: A temporally Coherent, Volumetric GAN for Super-resolution fluid flow



https://www.youtube.com/watch?v=i6JwXYypZ3Y

# SEMANTIC IMAGE SYNTHESIS WITH SPATIALLY-ADAPTIVE NORMALIZATION

https://www.youtube.com/watch?v=MXWm6w4E5q0

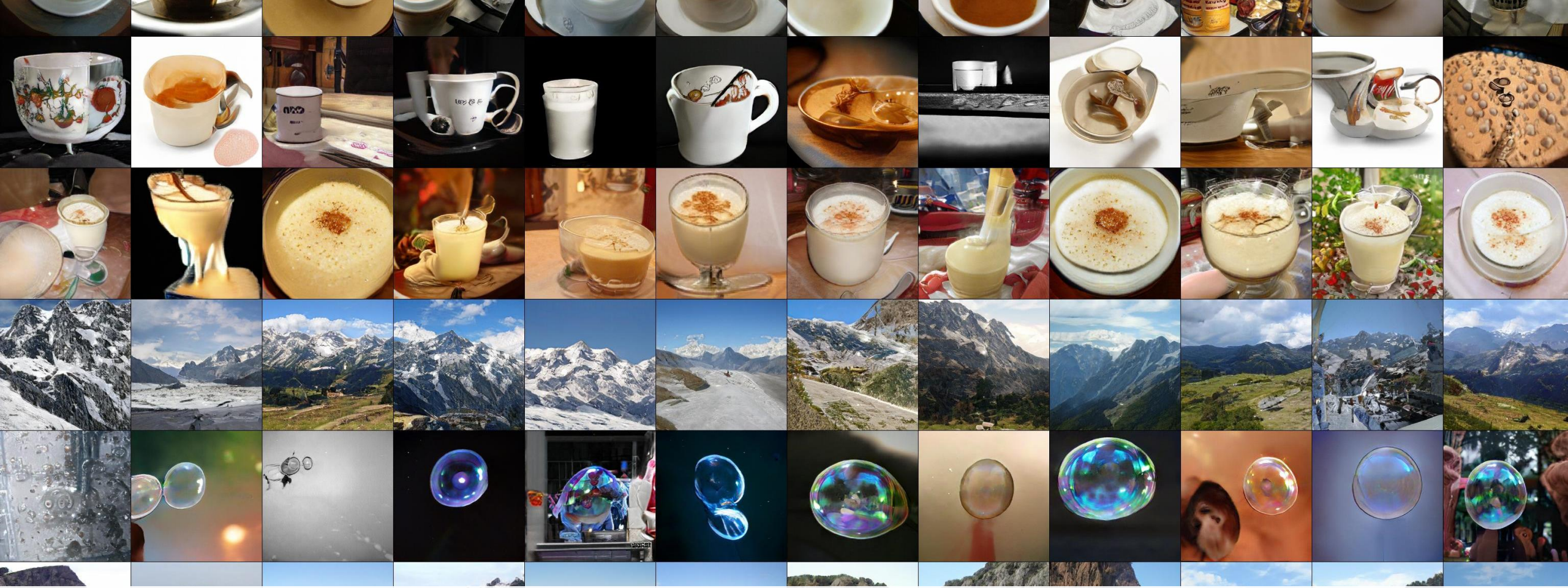


# FUN APLICATIONS

THISPERSONDOESNOTEXIST.COM & THISCATDOESNOTEXIST.COM

# INTERMISSION

# ARTICLES



# BIGGAN

Generative adversarial network

# AUTHORS

**Andrew Brock**
Heriot-Watt University

**Jeff Donahue** **Karen Simonyan**
DeepMind

# LARGE SCALE GAN TRAINING FOR HIGH FIDELITY NATURAL IMAGE SYNTHESIS

- This works focuses on closing the gap in fidelity and variety between real word images from ImageNet dataset.
- It demonstrates that GANs benefits from scaling, and training models with 2 to 4 times as many parameters and 8 times the batch size
- Two architectural changes
- Modified regularization scheme
- Model amenable to Truncation Trick (trade off between sample variety and fidelity)
- Instabilities due to Large Scale GANs

# BASELINE

SA-GAN architecture

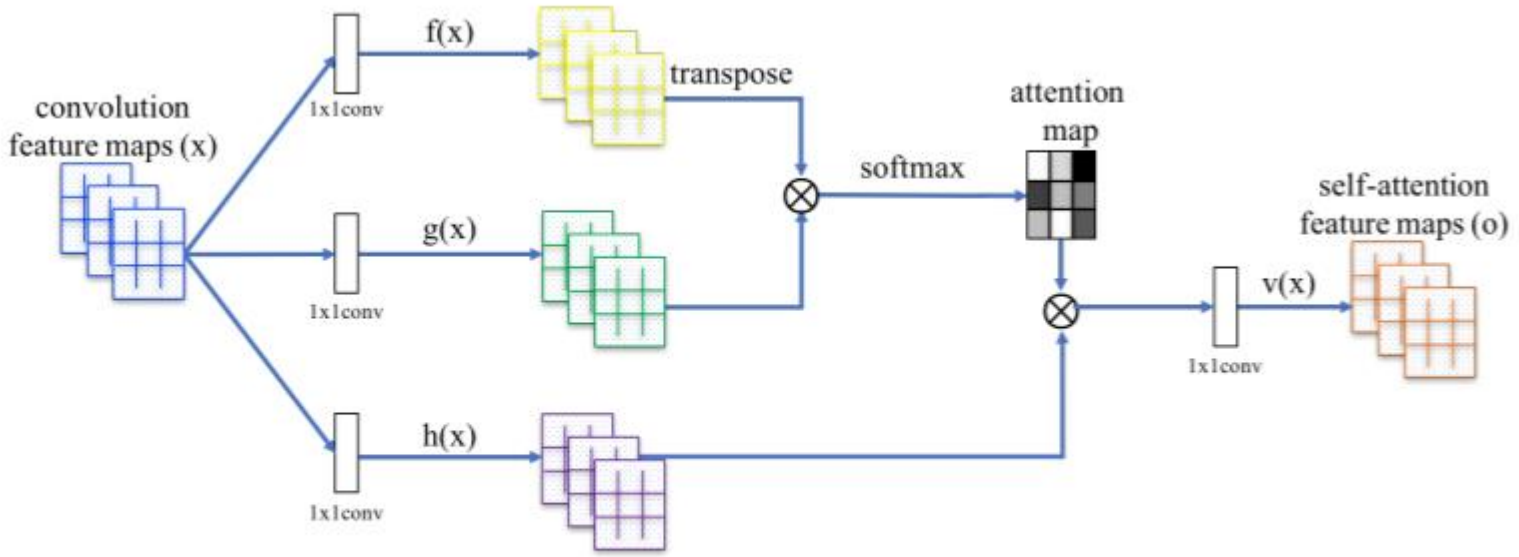



*Figure 2.* The proposed self-attention module for the SAGAN. The ⊗ denotes matrix multiplication. The softmax operation is performed on each row.

SA-GAN uses spectral normalization and imbalanced learning rate for generator and discriminator update

# BASELINE – SCALING UP

Use hinge loss as GAN objective function (SA-GAN paper):

"the proposed attention module has been applied to both the generator and the discriminator, which are trained in an alternating fashion by minimizing the hinge version of the adversarial loss"

$$
\begin{aligned}
L_D = & -\mathbb{E}_{(x,y)\sim p_{data}}[\min(0, -1 + D(x,y))] \\
& -\mathbb{E}_{z\sim p_z, y\sim p_{data}}[\min(0, -1 - D(G(z),y))], \\
L_G = & -\mathbb{E}_{z\sim p_z, y\sim p_{data}} D(G(z),y),
\end{aligned}
$$

Class-Conditional Batch Normalization to **G** and projection for **D**

**Halve learning rates and D takes two steps per G steps**

Orthogonal Initialization

Uses TPU not GPU

# SCALING UP GANS

| Batch | Ch. | Param (M) | Shared | Skip-$z$ | Ortho. | Itr $\times 10^3$ | FID | IS |
|---|---|---|---|---|---|---|---|---|
| 256 | 64 | 81.5 | SA-GAN Baseline | | | 1000 | 18.65 | 52.52 |
| 512 | 64 | 81.5 | ✗ | ✗ | ✗ | 1000 | 15.30 | 58.77($\pm$1.18) |
| 1024 | 64 | 81.5 | ✗ | ✗ | ✗ | 1000 | 14.88 | 63.03($\pm$1.42) |
| 2048 | 64 | 81.5 | ✗ | ✗ | ✗ | 732 | 12.39 | 76.85($\pm$3.83) |
| 2048 | 96 | 173.5 | ✗ | ✗ | ✗ | 295($\pm$18) | 9.54($\pm$0.62) | 92.98($\pm$4.27) |
| 2048 | 96 | 160.6 | ✓ | ✗ | ✗ | 185($\pm$11) | 9.18($\pm$0.13) | 94.94($\pm$1.32) |
| 2048 | 96 | 158.3 | ✓ | ✓ | ✗ | 152($\pm$7) | 8.73($\pm$0.45) | 98.76($\pm$2.84) |
| 2048 | 96 | 158.3 | ✓ | ✓ | ✓ | 165($\pm$13) | 8.51($\pm$0.32) | 99.31($\pm$2.10) |
| 2048 | 64 | 71.3 | ✓ | ✓ | ✓ | 371($\pm$7) | 10.48($\pm$0.10) | 86.90($\pm$0.61) |

By increasing batch size, increasing benefits, e.g. increasing batch number by 8 increases state of the art IS by 46% → increasing batch size allows covering more modes, providing better gradients for both networks

Side effect:

→ better performance on fewer iterations but becomes highly unstable and a training collapse occurs.

Conditioning **G** might improve stability, it is insufficient to ensure stability

# SCALING UP GANS

| Batch | Ch. | Param (M) | Shared | Skip-$z$ | Ortho. | Itr $\times 10^3$ | FID | IS |
|---|---|---|---|---|---|---|---|---|
| 256 | 64 | 81.5 | SA-GAN Baseline | | | 1000 | 18.65 | 52.52 |
| 512 | 64 | 81.5 | ✗ | ✗ | ✗ | 1000 | 15.30 | 58.77(±1.18) |
| 1024 | 64 | 81.5 | ✗ | ✗ | ✗ | 1000 | 14.88 | 63.03(±1.42) |
| 2048 | 64 | 81.5 | ✗ | ✗ | ✗ | 732 | 12.39 | 76.85(±3.83) |
| 2048 | 96 | 173.5 | ✗ | ✗ | ✗ | 295(±18) | 9.54(±0.62) | 92.98(±4.27) |
| 2048 | 96 | 160.6 | ✓ | ✗ | ✗ | 185(±11) | 9.18(±0.13) | 94.94(±1.32) |
| 2048 | 96 | 158.3 | ✓ | ✓ | ✗ | 152(±7) | 8.73(±0.45) | 98.76(±2.84) |
| 2048 | 96 | 158.3 | ✓ | ✓ | ✓ | 165(±13) | 8.51(±0.32) | 99.31(±2.10) |
| 2048 | 64 | 71.3 | ✓ | ✓ | ✓ | 371(±7) | 10.48(±0.10) | 86.90(±0.61) |

Increase width (number of channels) in each layer by 50%: leads to improvement which may be due to the model capacity to model the complexity of the dataset.

# SCALING UP GANS

| Batch | Ch. | Param (M) | Shared | Skip-$z$ | Ortho. | Itr $\times 10^3$ | FID | IS |
|---|---|---|---|---|---|---|---|---|
| 256 | 64 | 81.5 | SA-GAN Baseline | | | 1000 | 18.65 | 52.52 |
| 512 | 64 | 81.5 | ✗ | ✗ | ✗ | 1000 | 15.30 | 58.77($\pm$1.18) |
| 1024 | 64 | 81.5 | ✗ | ✗ | ✗ | 1000 | 14.88 | 63.03($\pm$1.42) |
| 2048 | 64 | 81.5 | ✗ | ✗ | ✗ | 732 | 12.39 | 76.85($\pm$3.83) |
| 2048 | 96 | 173.5 | ✗ | ✗ | ✗ | 295($\pm$18) | 9.54($\pm$0.62) | 92.98($\pm$4.27) |
| 2048 | 96 | 160.6 | ✓ | ✗ | ✗ | 185($\pm$11) | 9.18($\pm$0.13) | 94.94($\pm$1.32) |
| 2048 | 96 | 158.3 | ✓ | ✓ | ✗ | 152($\pm$7) | 8.73($\pm$0.45) | 98.76($\pm$2.84) |
| 2048 | 96 | 158.3 | ✓ | ✓ | ✓ | 165($\pm$13) | 8.51($\pm$0.32) | 99.31($\pm$2.10) |
| 2048 | 64 | 71.3 | ✓ | ✓ | ✓ | 371($\pm$7) | 10.48($\pm$0.10) | 86.90($\pm$0.61) |

Shared Embedding instead, which is linearly projected to each layer's gains and bias. Reduces computation and memory cost, and improves training speed

# SCALING UP GANS

Shared Embedding

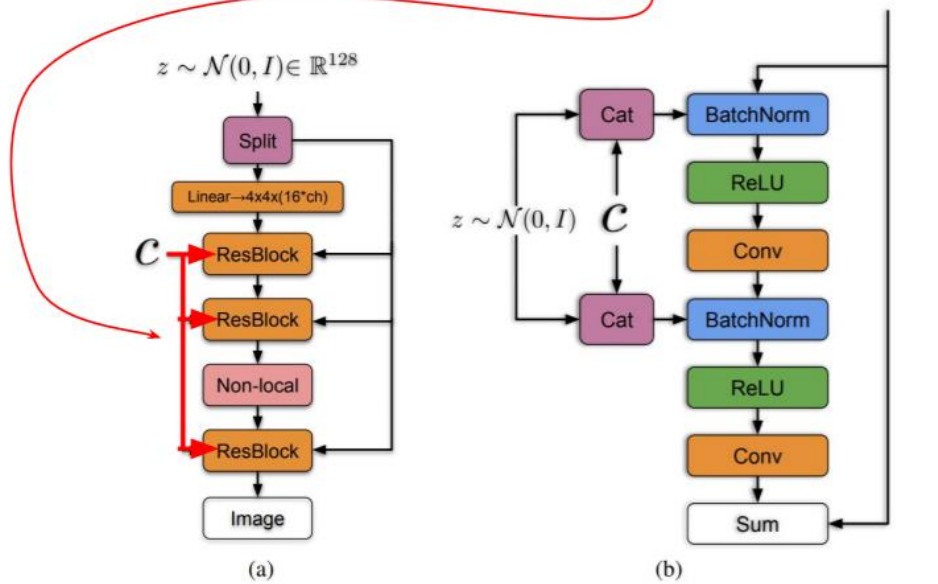
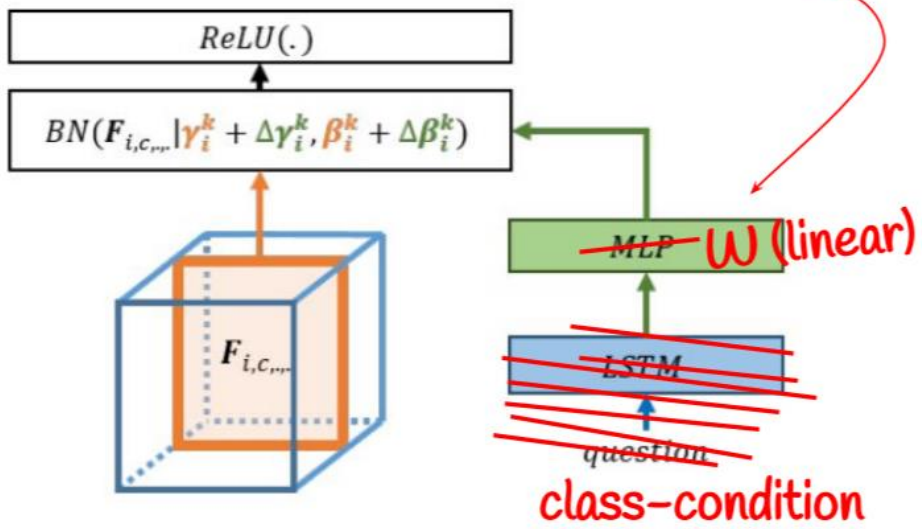



Figure 15: (a) A typical architectural layout for **G**; details are in the following tables. (b) A Residual Block in **G**. $c$ is concatenated with a chunk of $z$ and projected to the BatchNorm gains and biases.

Linearly Projected



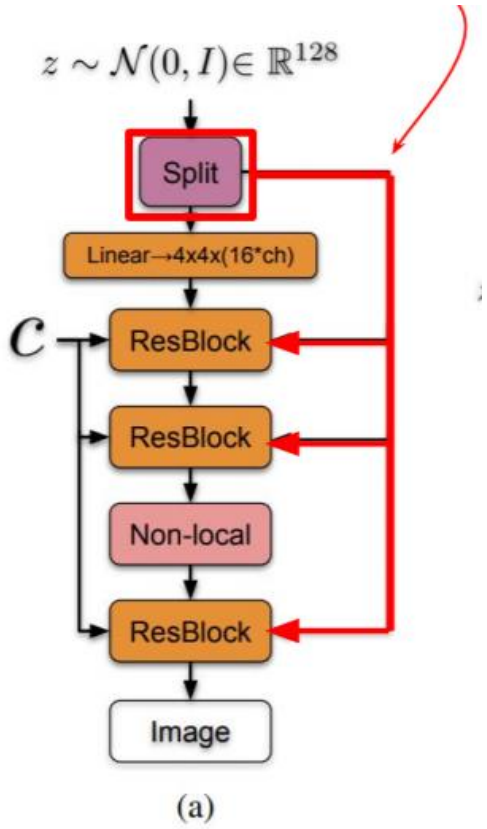
# SCALING UP GANS

| Batch | Ch. | Param (M) | Shared | Skip-$z$ | Ortho. | Itr $\times 10^3$ | FID | IS |
|---|---|---|---|---|---|---|---|---|
| 256 | 64 | 81.5 | SA-GAN Baseline | | | 1000 | 18.65 | 52.52 |
| 512 | 64 | 81.5 | ✗ | ✗ | ✗ | 1000 | 15.30 | 58.77($\pm$1.18) |
| 1024 | 64 | 81.5 | ✗ | ✗ | ✗ | 1000 | 14.88 | 63.03($\pm$1.42) |
| 2048 | 64 | 81.5 | ✗ | ✗ | ✗ | 732 | 12.39 | 76.85($\pm$3.83) |
| 2048 | 96 | 173.5 | ✗ | ✗ | ✗ | 295($\pm$18) | 9.54($\pm$0.62) | 92.98($\pm$4.27) |
| 2048 | 96 | 160.6 | ✓ | ✗ | ✗ | 185($\pm$11) | 9.18($\pm$0.13) | 94.94($\pm$1.32) |
| 2048 | 96 | 158.3 | ✓ | ✓ | ✗ | 152($\pm$7) | 8.73($\pm$0.45) | 98.76($\pm$2.84) |
| 2048 | 96 | 158.3 | ✓ | ✓ | ✓ | 165($\pm$13) | 8.51($\pm$0.32) | 99.31($\pm$2.10) |
| 2048 | 64 | 71.3 | ✓ | ✓ | ✓ | 371($\pm$7) | 10.48($\pm$0.10) | 86.90($\pm$0.61) |

Add direct skip connections Skip connections (skip $z$) from the noise vector $z$ to multiple layers of G rather than just the initial layer.

This design is to allow G to use the latent space to directly influence features at different resolutions and levels of hierarchy.

# SKIP-Z


$z \sim \mathcal{N}(0, I) \in \mathbb{R}^{128}$
Split
Linear→4x4x(16*ch)
$c$
ResBlock
ResBlock
Non-local
ResBlock
Image
(a)

# SCALING UP GANS

| Batch | Ch. | Param (M) | Shared | Skip-$z$ | Ortho. | Itr $\times 10^3$ | FID | IS |
|---|---|---|---|---|---|---|---|---|
| 256 | 64 | 81.5 | SA-GAN Baseline | | | 1000 | 18.65 | 52.52 |
| 512 | 64 | 81.5 | ✗ | ✗ | ✗ | 1000 | 15.30 | 58.77($\pm$1.18) |
| 1024 | 64 | 81.5 | ✗ | ✗ | ✗ | 1000 | 14.88 | 63.03($\pm$1.42) |
| 2048 | 64 | 81.5 | ✗ | ✗ | ✗ | 732 | 12.39 | 76.85($\pm$3.83) |
| 2048 | 96 | 173.5 | ✗ | ✗ | ✗ | 295($\pm$18) | 9.54($\pm$0.62) | 92.98($\pm$4.27) |
| 2048 | 96 | 160.6 | ✓ | ✗ | ✗ | 185($\pm$11) | 9.18($\pm$0.13) | 94.94($\pm$1.32) |
| 2048 | 96 | 158.3 | ✓ | ✓ | ✗ | 152($\pm$7) | 8.73($\pm$0.45) | 98.76($\pm$2.84) |
| 2048 | 96 | 158.3 | ✓ | ✓ | ✓ | 165($\pm$13) | 8.51($\pm$0.32) | 99.31($\pm$2.10) |
| 2048 | 64 | 71.3 | ✓ | ✓ | ✓ | 371($\pm$7) | 10.48($\pm$0.10) | 86.90($\pm$0.61) |

Truncation Trick: truncating a z vector by resampling the values with magnitude above a chosen threshold

**Orthogonal normalization:** $R_\beta(W) = \beta \|W^\top W \odot (1 - I)\|_{\mathrm{F}}^2,$

# TRUNCATION TRICK

Using a different latent distribution for sampling than was used in training.

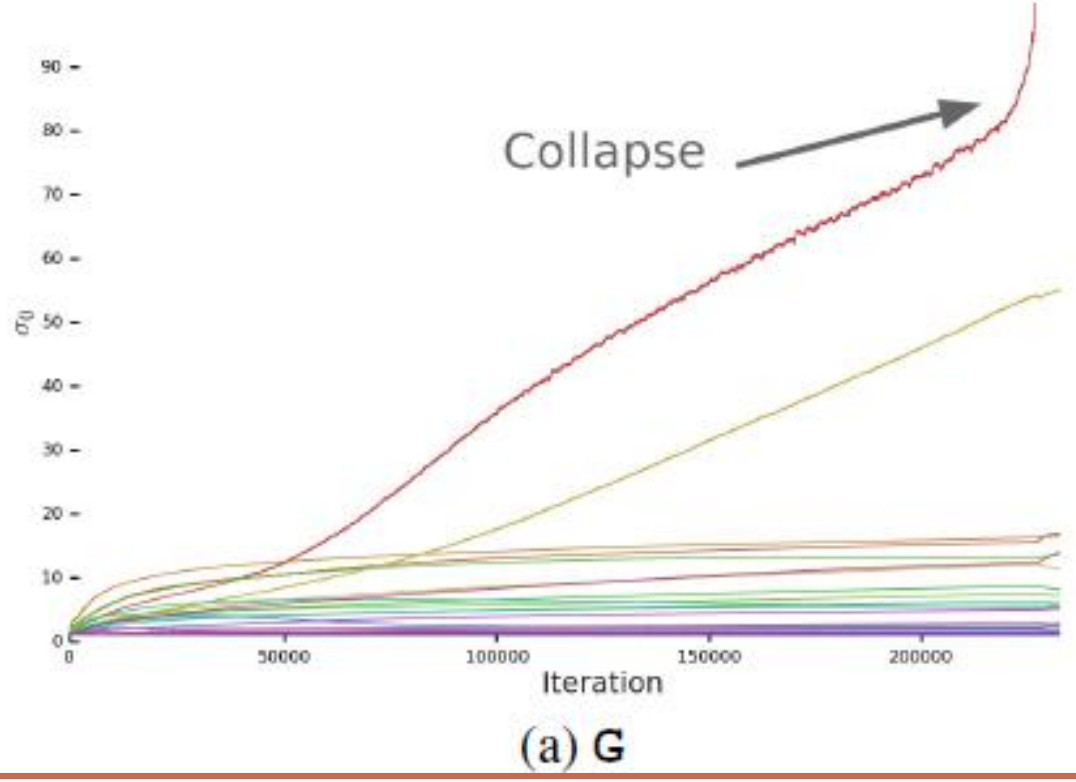
Taking a model trained in $z \sim \mathcal{N}(0, I)$ and sampling $z$ from a truncated normal has a evidence of boosting performance.

**Truncation Trick :** Resampling the values with magnitude above a threshold lead to improvement in individual sample quality at the cost of reduction in overall sample variety

This allows fine-grained, helps to the selection of the tradeoff between sample quality and variety for a given G.

To enforce amenability to truncation by conditioning **G** to be smooth, so that the full space of z will map to good output samples.

# INESTABILITY – GENERATOR



(a) G

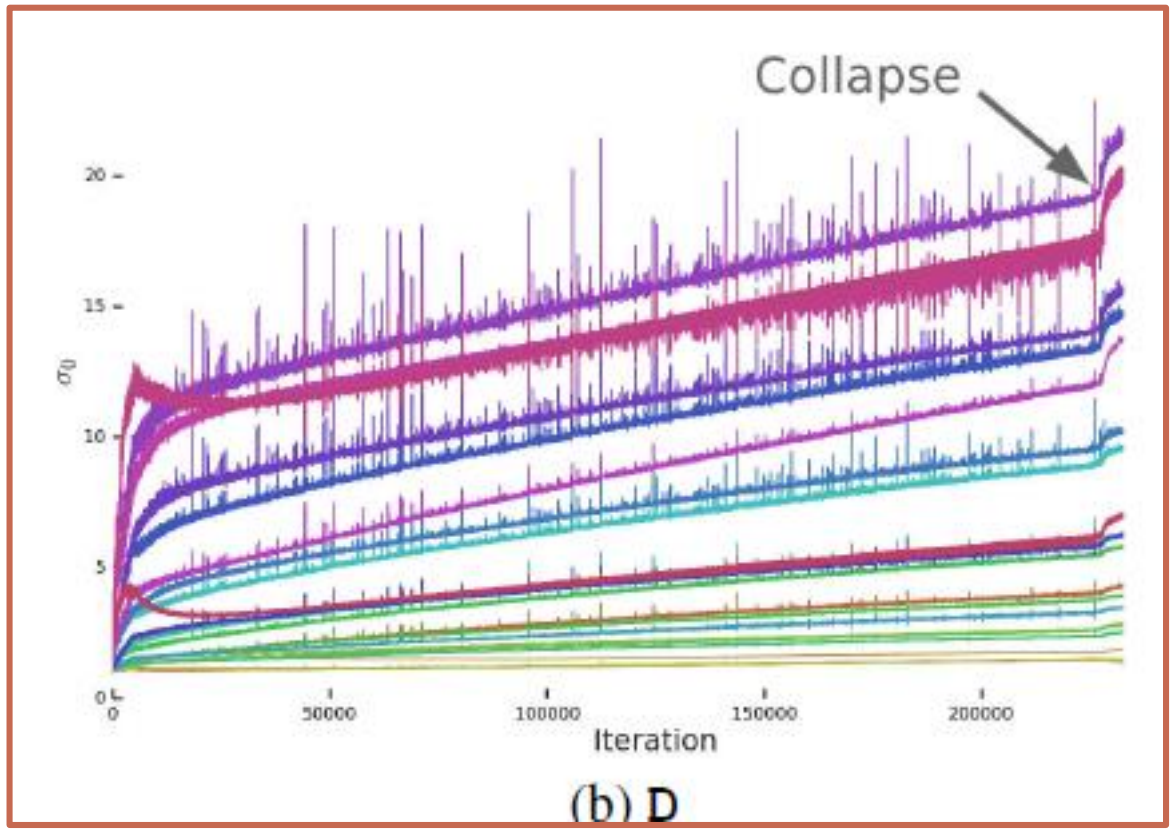
"We found the top three singular values $\sigma_0, \sigma_1, \sigma_2$ of each weight matrix to be the most informative."

**G layers** have well behave, but some (G first layer) gain instability during training and collapse.

**Conditioning G to counterattack spectral explosion,** by directing regularizing $\sigma_0$ values, but still no combination prevents training collapse.

**"while conditioning G might improve stability, it is insufficient to ensure stability."**

# INESTABILITY – DISCRIMINATOR



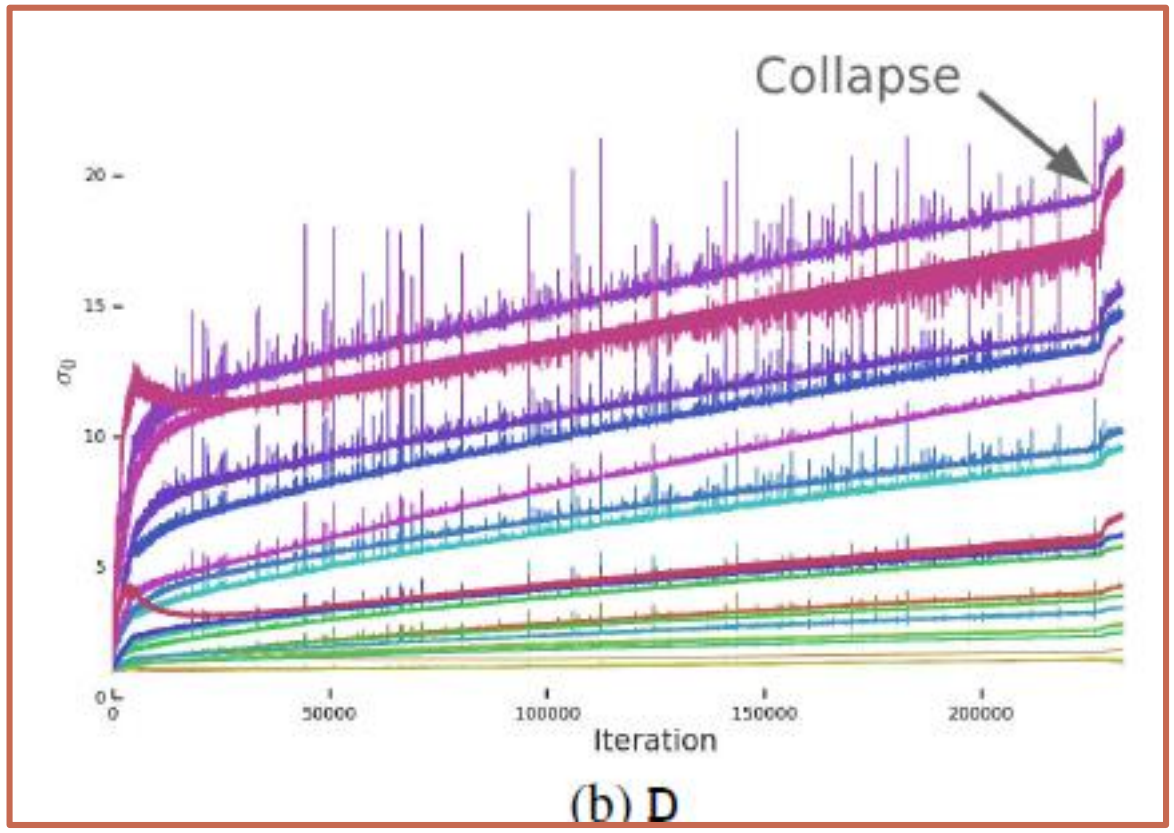
(b) D

Spectra is noisy, well behaved and singular values grow through training and only jumps at collapse, instead of exploading

Spikes in D suggest that it periodically receives large gradients, this can be an effect of the optimization through an adversarial process, where G periodically perturbs D.

To solve it gradient penalties, by using $R_1$ zero centered performance degrades, so by repeating this with different strengths of Orthogonal Regularization, L2, and dropout training stability **can be achieved** but with substantial cost of performance

# INESTABILITY – DISCRIMINATOR



(b) D

Losses approaches to zero during training, but on collapse it undergoes an upward jump

One possible explanation for this behavior is that D is overfitting to the training set, memorizing training examples rather than learning some meaningful boundary between real and generated images.

Proven with performance over validation set of ImageNet, this behaviour is attributed to D's role which job is not to generalize but to differentiate training from G generated.

# IMAGENET EVALUATION

| Model | Res. | FID/IS | (min FID) / IS | FID / (valid IS) | FID / (max IS) |
|---|---|---|---|---|---|
| SN-GAN | 128 | 27.62/36.80 | N/A | N/A | N/A |
| SA-GAN | 128 | 18.65/52.52 | N/A | N/A | N/A |
| BigGAN | 128 | $8.7 \pm .6/98.8 \pm 3$ | $7.7 \pm .2/126.5 \pm 0$ | $9.6 \pm .4/166.3 \pm 1$ | $25 \pm 2/206 \pm 2$ |
| BigGAN | 256 | $8.7 \pm .1/142.3 \pm 2$ | $7.7 \pm .1/178.0 \pm 5$ | $9.3 \pm .3/233.1 \pm 1$ | $25 \pm 5/291 \pm 4$ |
| BigGAN | 512 | 8.1/144.2 | 7.6/170.3 | 11.8/241.4 | 27.0/275 |
| BigGAN-deep | 128 | $5.7 \pm .3/124.5 \pm 2$ | $6.3 \pm .3/148.1 \pm 4$ | $7.4 \pm .6/166.5 \pm 1$ | $25 \pm 2/253 \pm 11$ |
| BigGAN-deep | 256 | $6.9 \pm .2/171.4 \pm 2$ | $7.0 \pm .1/202.6 \pm 2$ | $8.1 \pm .1/232.5 \pm 2$ | $27 \pm 8/317 \pm 6$ |
| BigGAN-deep | 512 | 7.5/152.8 | 7.7/181.4 | 11.5/241.5 | 39.7/298 |

(a) 128×128 (b) 256×256 (c) 512×512 (d)

## Failures:

Local Artifacts → texture blobs instead of objects

Collapse Mode

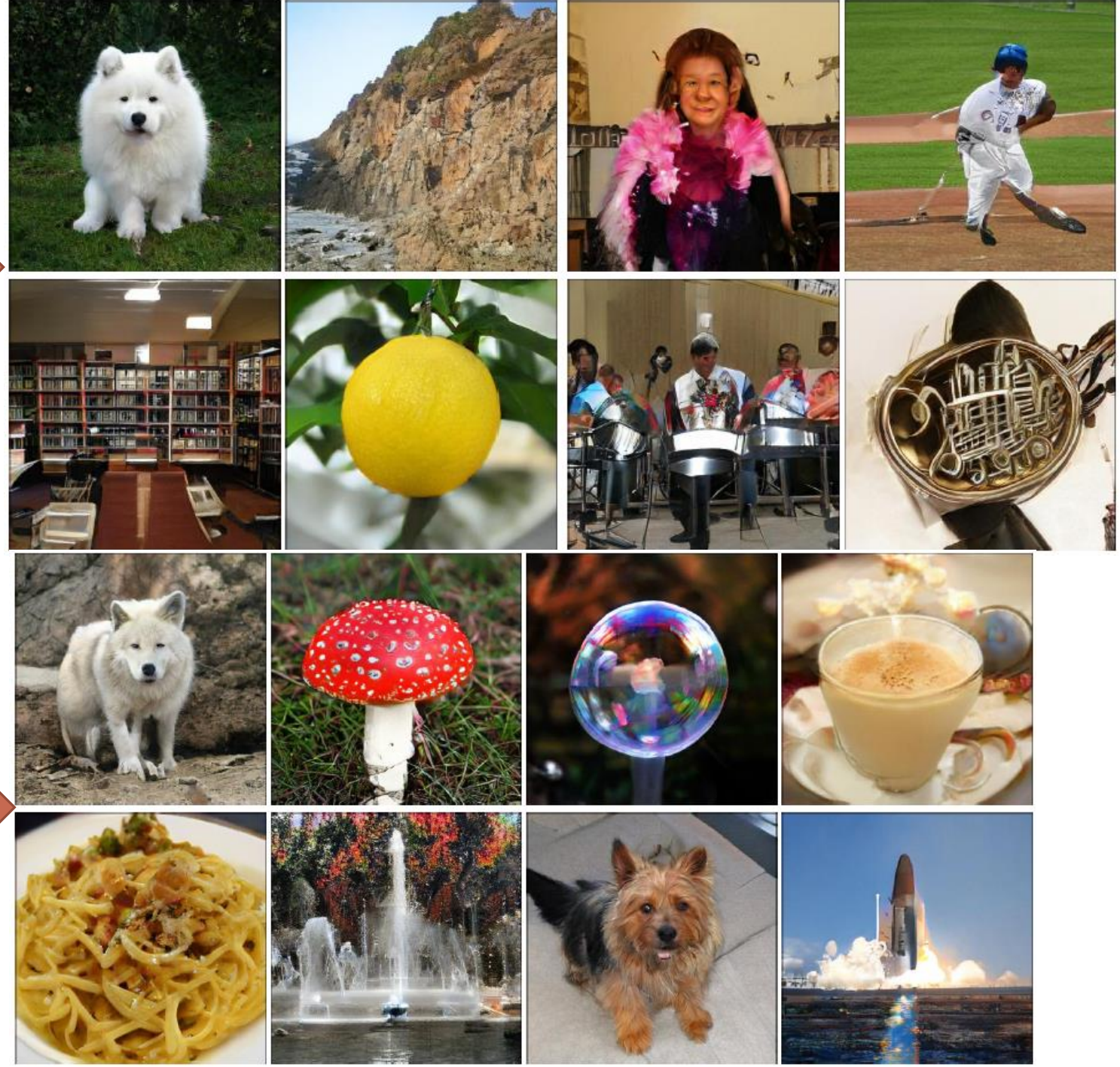
Class Leakage (d)

Some classes are easier for the model to generate

EASY CLASES VS DIFFICULT CLASSES

# IMAGENET EVALUATION

512X512 SAMPLES



# JFT-300M EVALUATION

| Ch. | Param (M) | Shared | Skip-$z$ | Ortho. | FID | IS | (min FID) / IS | FID / (max IS) |
|---|---|---|---|---|---|---|---|---|
| 64 | 317.1 | ✗ | ✗ | ✗ | 48.38 | 23.27 | 48.6/23.1 | 49.1/23.9 |
| 64 | 99.4 | ✓ | ✓ | ✓ | 23.48 | 24.78 | 22.4/21.0 | 60.9/35.8 |
| 96 | 207.9 | ✓ | ✓ | ✓ | 18.84 | 27.86 | 17.1/23.3 | 51.6/38.1 |
| 128 | 355.7 | ✓ | ✓ | ✓ | 13.75 | 30.61 | 13.0/28.0 | 46.2/47.8 |

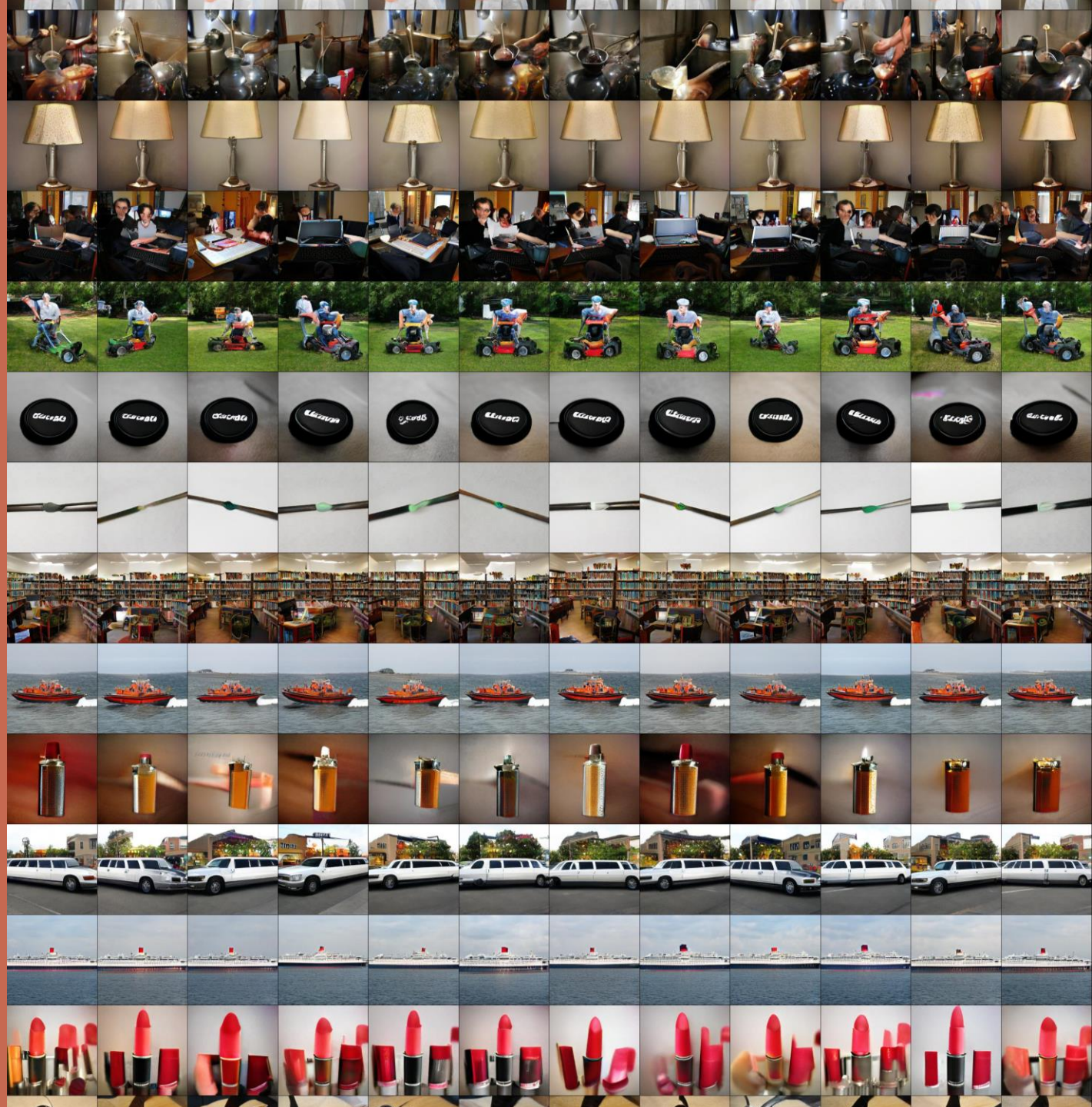
## JFT-300M:

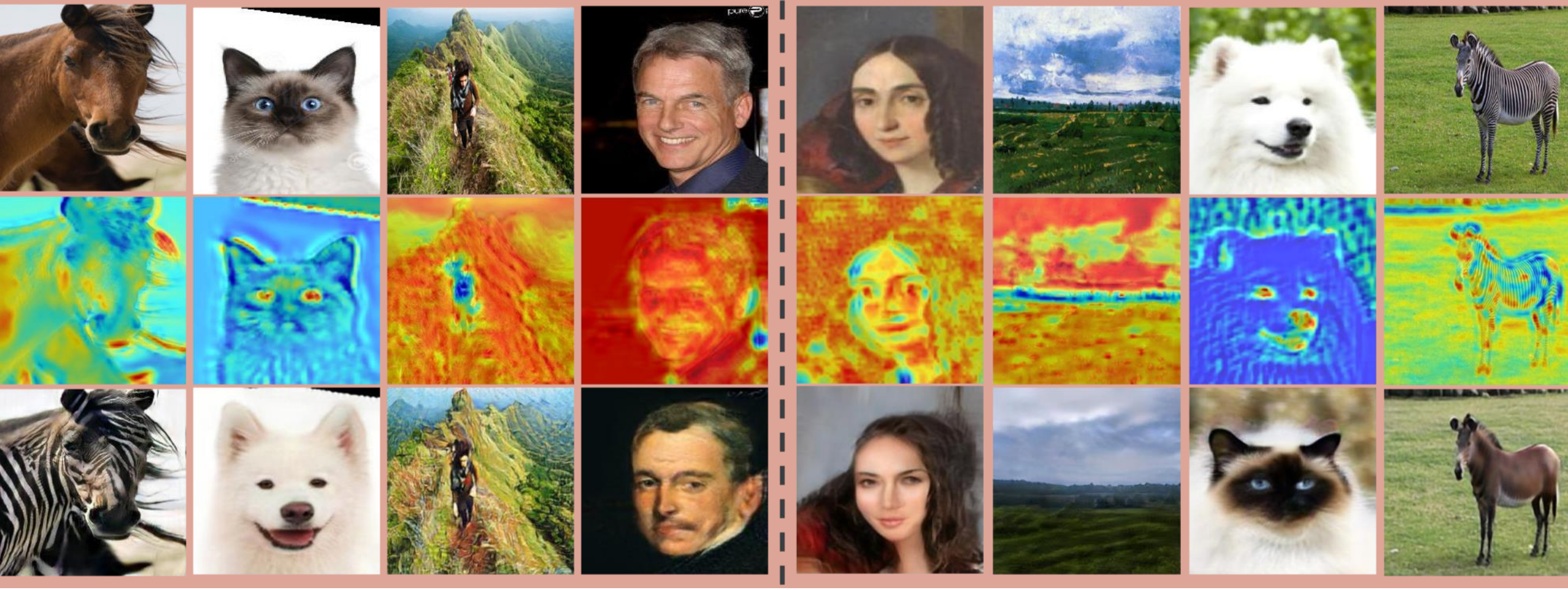
Subsampled with 8.5K most common labels, 292M images

# IMPORTANCE – RELEVANCE – CONCLUSIONS

Model natural images of multiple categories highly benefit from scaling up.

This works sets a baseline for large scale GANs for image synthesis, by not only highlighting behaviors that may affect the training but how large scale benefits the problem of modeling natural images by also contrasting its method with larger datasets that ImageNet.





# U-GAT-IT

Unsupervised Image-to-Image Translation

# AUTHORS

**Junho Kim**

**Minjae Kim**

**Hyeonwoo Kang**

**Kwang Hee Lee**
Boeing Korea Engineering and Technology Center (BKETC)

**NCSOFT AI Center Vision AI Lab**

NCSOFT

Open Sources

# INTRODUCTION

Novel method for unsupervised image-to-image translation, incorporating a new attention module and a learnable normalization function AdaLIN.

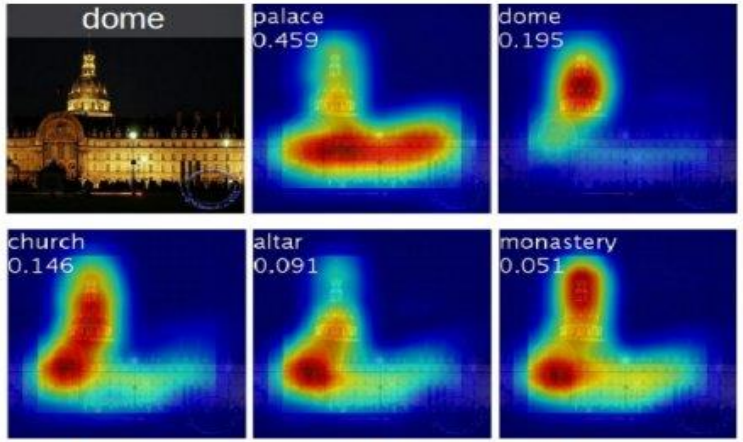
Previous methods (such as CycleGAN and pix2pix) have difficulty performing the translation between domains with significantly different shapes and textures.


Input Image
CycleGAN
DRIT
UNIT
MUNIT

# RELATED WORK: CLASS ACTIVATION MAP

Generation of class activation maps using the global average pooling (GAP) in CNNs. This technique allows to expose the implicit attention a CNN has on an image.

Specifically, it allows us to determine the most relevant regions a CNN “looks” during discriminative task.



Class activation maps of top 5 predictions

Class activation maps for one object class

# RELATED WORK: NORMALIZATION

CNN feature statistics can be used as style descriptors, an example of this is Instance Normalization (IN) that has surpassed Batch Normalization (BN) or Layer Normalization (LN) in neural style tasks.

$$\mathbf{BN}(x) = \gamma \left( \frac{x - \mu(x)}{\sigma(x)} \right) + \beta$$

$$\mathbf{IN}(x) = \gamma \left( \frac{x - \mu(x)}{\sigma(x)} \right) + \beta$$

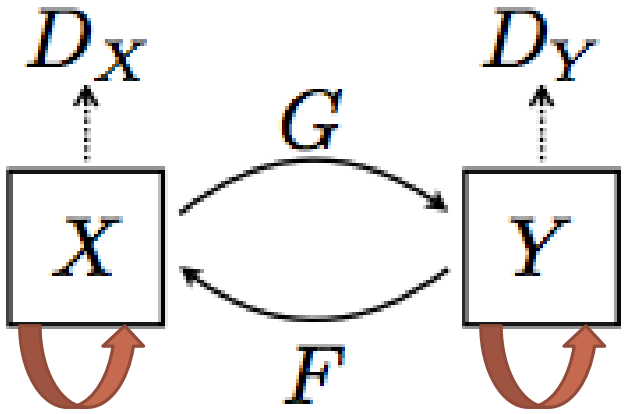
$$\mathbf{AdaIN}(x, y) = \sigma(y) \left( \frac{x - \mu(x)}{\sigma(x)} \right) + \mu(y)$$

In recent years, new Instance-based normalization methods have come out such as AdaIN, CIN or BIN.

# U-GAT-IT

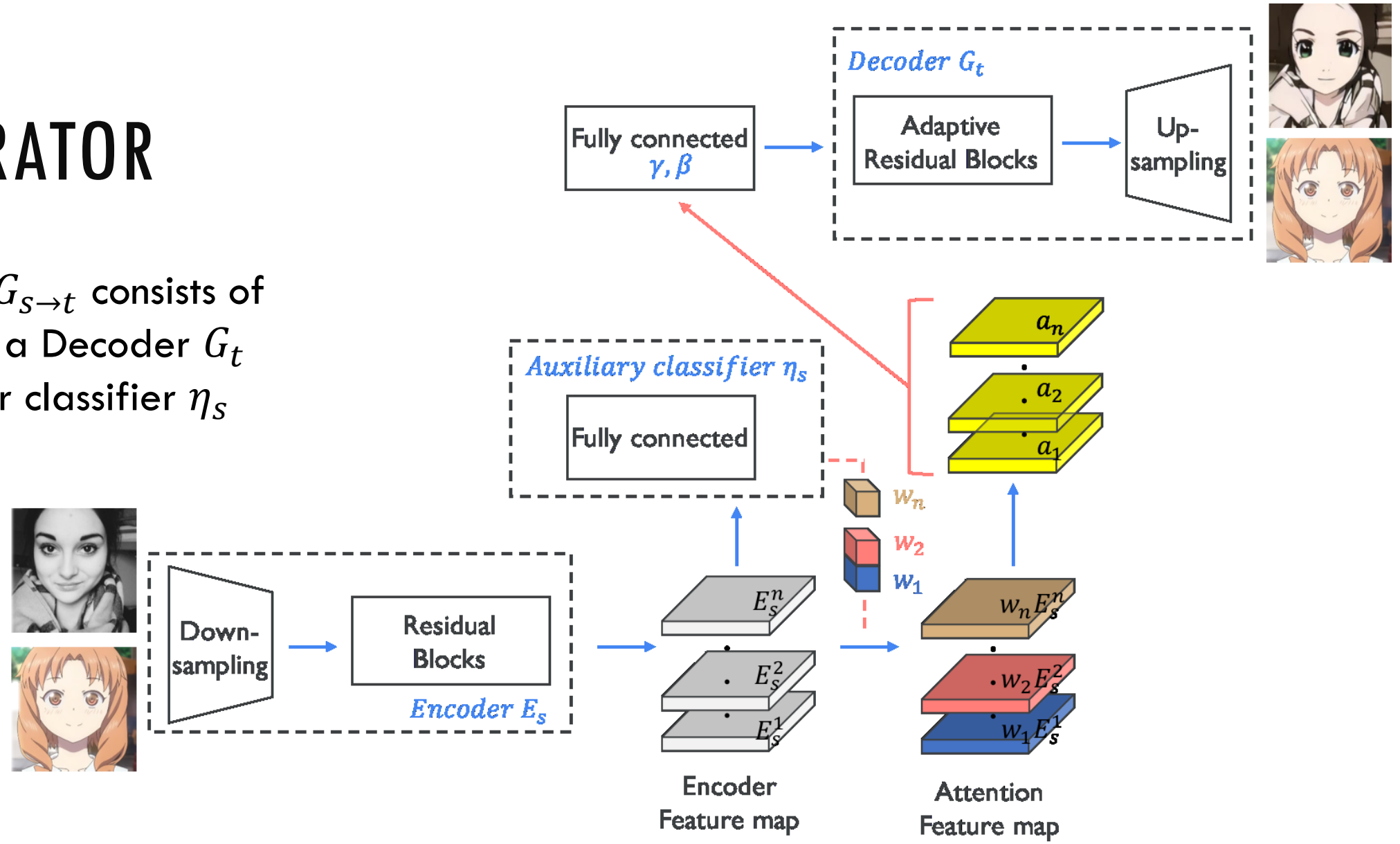
Goal: Train a function $G_{s \to t}$ that maps between domain $X_s$ to domain $X_t$

To accomplish this goal, the model consist of generators $G_{s \to t}$ and $G_{t \to s}$ and two discriminators $D_s$ and $D_t$ , each with attention modules .




# GENERATOR

The generator $G_{s\to t}$ consists of an Encoder $E_s$, a Decoder $G_t$ and the auxiliar classifier $\eta_s$



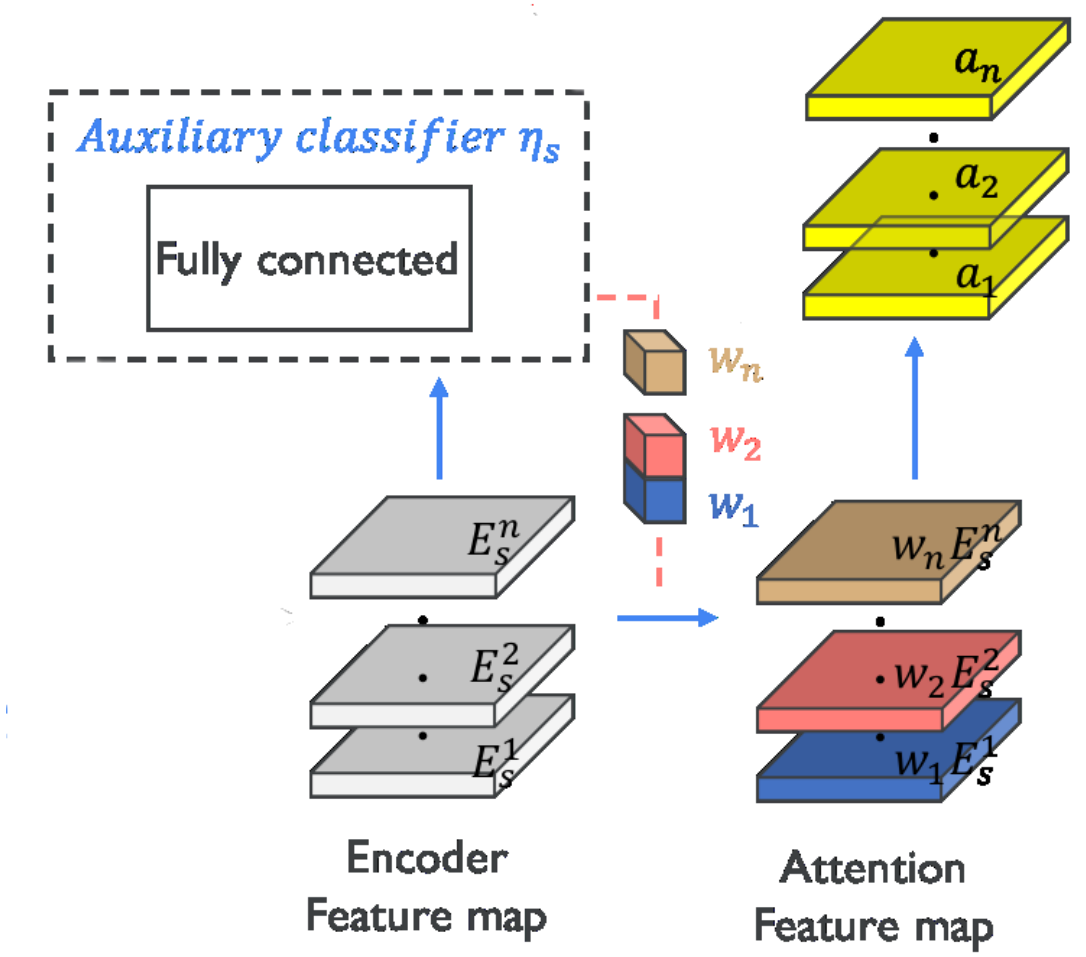

Additionally, the Residual Blocks are equipped with AdaLIN

# AUXILIER CLASIFFIER

Inspired by Class Activation Maps

Learns the importance weights $W_s^k$ by using Global Average Pooling and Global Max Pooling

With the importance weights, we can calculate the attention feature map $a_s$ by multiplying the weights with the initial feature map.




$$\eta_s(x) = \sigma\left(\sum_K W_s^k \sum_{ij} E_s^{k_{ij}}(x)\right)$$

# ADALIN: ADAPTATIVE LAYER-INSTANCE NORMALIZATION

We have:

- $\mu_I \; and \; \mu_L$ are the channel-wise and layer-wise means.
- $\sigma_I \; and \; \sigma_L$ are the channel-wise and layer-wise standard deviation.
- $\gamma \; and \; \beta$ are the parameters learn in the fully connected layer
- $\tau$: Learning rate
- $\Delta\rho$: Gradient calculated by optimizer
- $\rho$: gate parameter, helps to determine which method (IN or LN) is the most relevant during a task

$$\hat{a_I} = \frac{a - \mu_I}{\sqrt{\sigma_I^2 + \epsilon}}, \hat{a_L} = \frac{a - \mu_L}{\sqrt{\sigma_L^2 + \epsilon}},$$

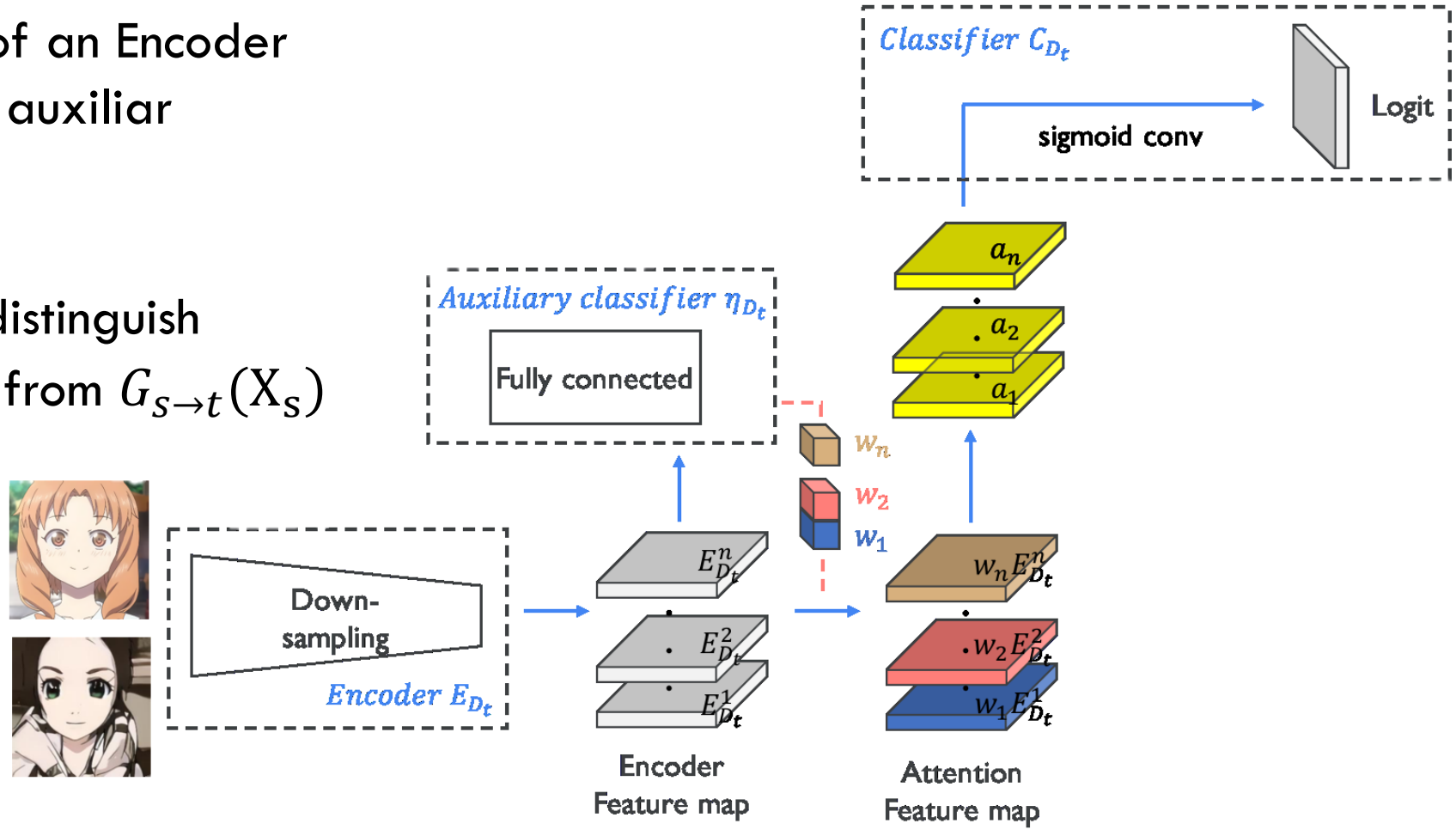
$$AdaLIN(a, \gamma, \beta) = \gamma \cdot (\rho \cdot \hat{a_I} + (1 - \rho) \cdot \hat{a_L}) + \beta$$

$$\rho \leftarrow clip_{[0,1]}(\rho - \tau \Delta\rho),$$

# DISCRIMINATOR

The discriminator $D_t$ consist of an Encoder $E_{\mathrm{D_t}}$, a classifier $C_{D_t}$ and the auxiliar classifier $\eta_{D_t}$

$C_{D_t}$ and $\eta_{D_t}$ are trained to distinguish simples from $X_t$ and simples from $G_{s\to t}(\mathrm{X_s})$



# LOSS FUNCTION

## Adversarial Loss

$$L_{gan}^{s\to t} = (\mathbb{E}_{x\sim X_t}[(D_t(x))^2] + E_{x\sim X_s}[(1 - D_t(G_{s\to t}(x)))^2])$$

## Identity Loss

$$L_{identity}^{s\to t} = \mathbb{E}_{x\sim X_t}[|x - G_{s\to t}(x)|_1]$$

## Cycle Loss

$$L_{cycle}^{s\to t} = \mathbb{E}_{x\sim X_s}[|x - G_{t\to s}(G_{s\to t}(x))|_1]$$

## CAM Loss

$$L_{cam}^{s\to t} = -(\mathbb{E}_{x\sim X_s}[log(\eta_s(x))] + \mathbb{E}_{x\sim X_t}[log(1 - \eta_s(x))],$$

$$L_{cam}^{D_t} = \mathbb{E}_{x\sim X_t}[(\eta_{D_t}(x))^2] + \mathbb{E}_{x\sim X_s}[log(1 - \eta_{D_t}(G_{s\to t}(x)))^2]$$

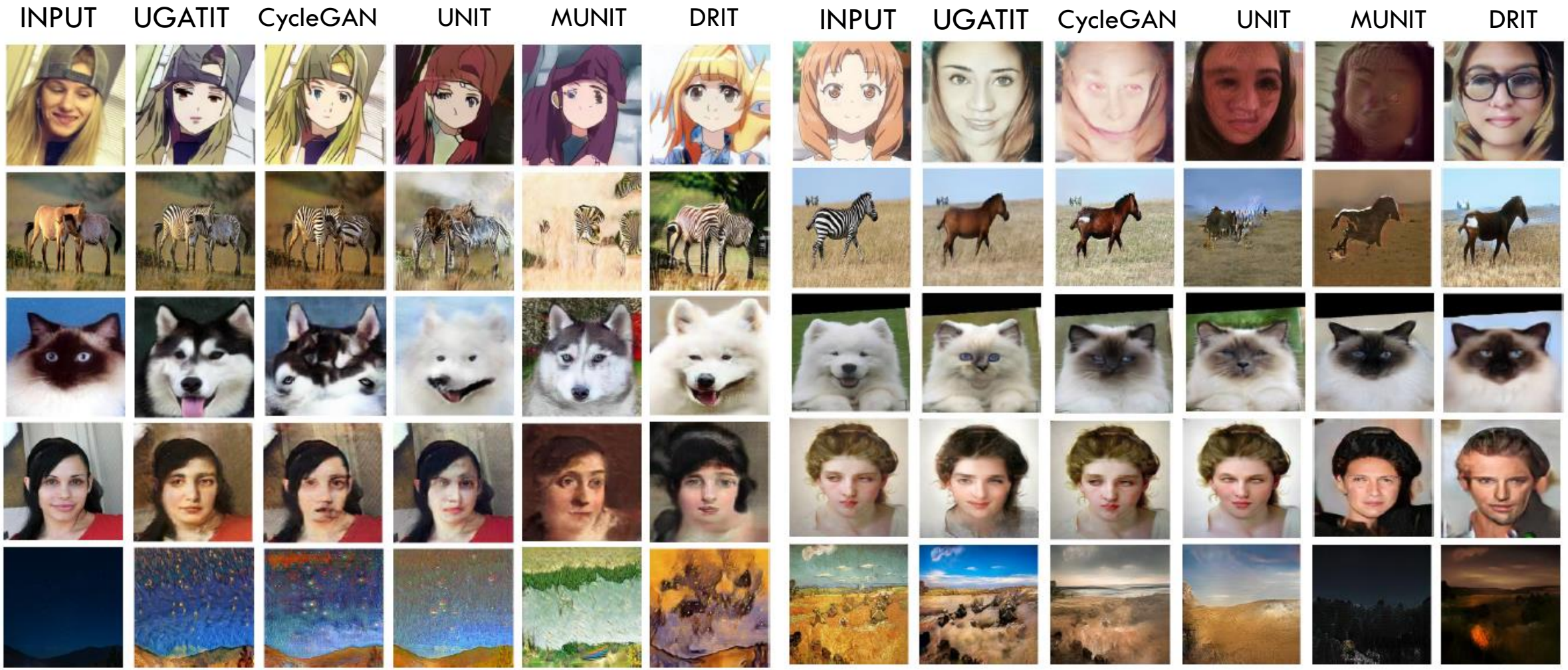
# LOSS FUNCTION

Weighted Min-Max game of multiple players:

- Both Generators $G_{s\to t}$ and $G_{t\to s}$ : $L_{GAN}$, $L_{cycle}$ and $L_{identity}$
- Both Discriminators $D_t$ and $D_s$: $L_{GAN}$
- Four Auxiliar classifiers : $L_{CAM}$
  - Both Generator classifiers: $\eta_s$ and $\eta_t$
  - Both Discriminator classifiers: $\eta_{D_s}$ and $\eta_{D_t}$

$$\min_{G_{s\to t},G_{t\to s},\eta_s,\eta_t} \max_{D_s,D_t,\eta_{D_s},\eta_{D_t}} \lambda_1 L_{gan} + \lambda_2 L_{cycle} + \lambda_3 L_{identity} + \lambda_4 L_{cam},$$

# RESULTS


INPUT
UGATIT
CycleGAN
UNIT
MUNIT
DRIT
INPUT
UGATIT
CycleGAN
UNIT
MUNIT
DRIT

# RESULTS

Metrics:

1. KID (Kernel Inception Distance): Maximum Mean Distance between the representations of the real images and the generated ones.
2. User perceptual study

Table 3. Kernel Inception Distance×100±std.×100 for difference image translation mode. Lower is better.

| Model \ Dataset | selfie2anime | horse2zebra | cat2dog | photo2portrait | photo2vangogh |
|---|---|---|---|---|---|
| U-GAT-IT | **11.61 ± 0.57** | **7.06 ± 0.8** | **7.07 ± 0.65** | 1.79 ± 0.34 | 4.28 ± 0.33 |
| CycleGAN | 13.08 ± 0.49 | 8.05 ± 0.72 | 8.92 ± 0.69 | 1.84 ± 0.34 | 5.46 ± 0.33 |
| UNIT | 14.71 ± 0.59 | 10.44 ± 0.67 | 8.15 ± 0.48 | **1.2 ± 0.31** | **4.26 ± 0.29** |
| MUNIT | 13.85 ± 0.41 | 11.41 ± 0.83 | 10.13 ± 0.27 | 4.75 ± 0.52 | 13.08 ± 0.34 |
| DRIT | 15.08 ± 0.62 | 9.79 ± 0.62 | 10.92 ± 0.33 | 5.85 ± 0.54 | 12.65 ± 0.35 |
| Model \ Dataset | anime2selfie | zebra2horse | dog2cat | portrait2photo | vangogh2photo |
| U-GAT-IT | **11.52 ± 0.57** | **7.47 ± 0.71** | **8.15 ± 0.66** | 1.69 ± 0.53 | 5.61 ± 0.32 |
| CycleGAN | 11.84 ± 0.74 | 8.0 ± 0.66 | 9.94 ± 0.36 | 1.82 ± 0.36 | **4.68 ± 0.36** |
| UNIT | 26.32 ± 0.92 | 14.93 ± 0.75 | 9.81 ± 0.34 | **1.42 ± 0.24** | 9.72 ± 0.33 |
| MUNIT | 13.94 ± 0.72 | 16.47 ± 1.04 | 10.39 ± 0.25 | 3.3 ± 0.47 | 9.53 ± 0.35 |
| DRIT | 14.85 ± 0.6 | 10.98 ± 0.55 | 10.86 ± 0.24 | 4.76 ± 0.72 | 7.72 ± 0.34 |

Table 2. Preference score on translated images by user study.

| Model \ Dataset | selfie2anime | horse2zebra | cat2dog | photo2portrait | photo2vangogh |
|---|---|---|---|---|---|
| U-GAT-IT | **73.15** | **73.56** | **58.22** | 30.59 | **48.96** |
| CycleGAN | 20.07 | 23.07 | 6.19 | 26.59 | 27.33 |
| UNIT | 1.48 | 0.85 | 18.63 | **32.11** | 11.93 |
| MUNIT | 3.41 | 1.04 | 14.48 | 8.22 | 2.07 |
| DRIT | 1.89 | 1.48 | 2.48 | 2.48 | 9.70 |

# DISCUSSION

Both the Attention module and AdaLIN help improve the performance of U-GAT-IT in comparison to state-of-the-art methods, in terms of KID.

To demonstrate it, the experiment with different normalization methods and without the attention module.

| Model \ Dataset | selfie2anime | anime2selfie |
|---|---|---|
| U-GAT-IT | **11.61 ± 0.57** | **11.52 ± 0.57** |
| U-GAT-IT w/ IN | 13.64 ± 0.76 | 13.58 ± 0.8 |
| U-GAT-IT w/ LN | 12.39 ± 0.61 | 13.17 ± 0.8 |
| U-GAT-IT w/ AdaIN | 12.29 ± 0.78 | 11.81 ± 0.77 |
| U-GAT-IT w/ GN | 12.76 ± 0.64 | 12.30 ± 0.77 |
| U-GAT-IT w/o CAM | 12.85 ± 0.82 | 14.06 ± 0.75 |
| U-GAT-IT w/o G_CAM | 12.33 ± 0.68 | 13.86 ± 0.75 |
| U-GAT-IT w/o D_CAM | 12.49 ± 0.74 | 13.33 ± 0.89 |

## CAM

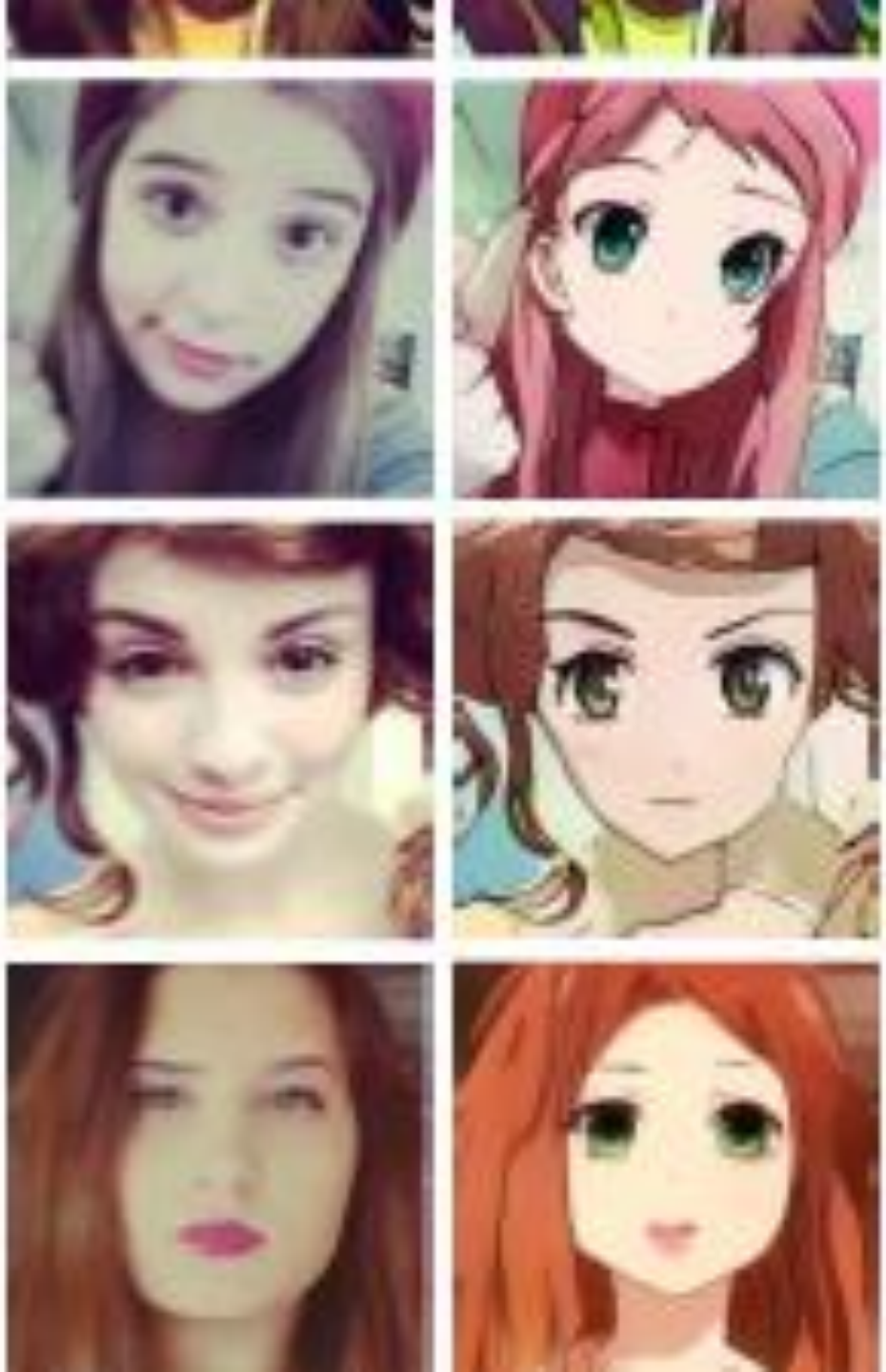
## AdaLIN

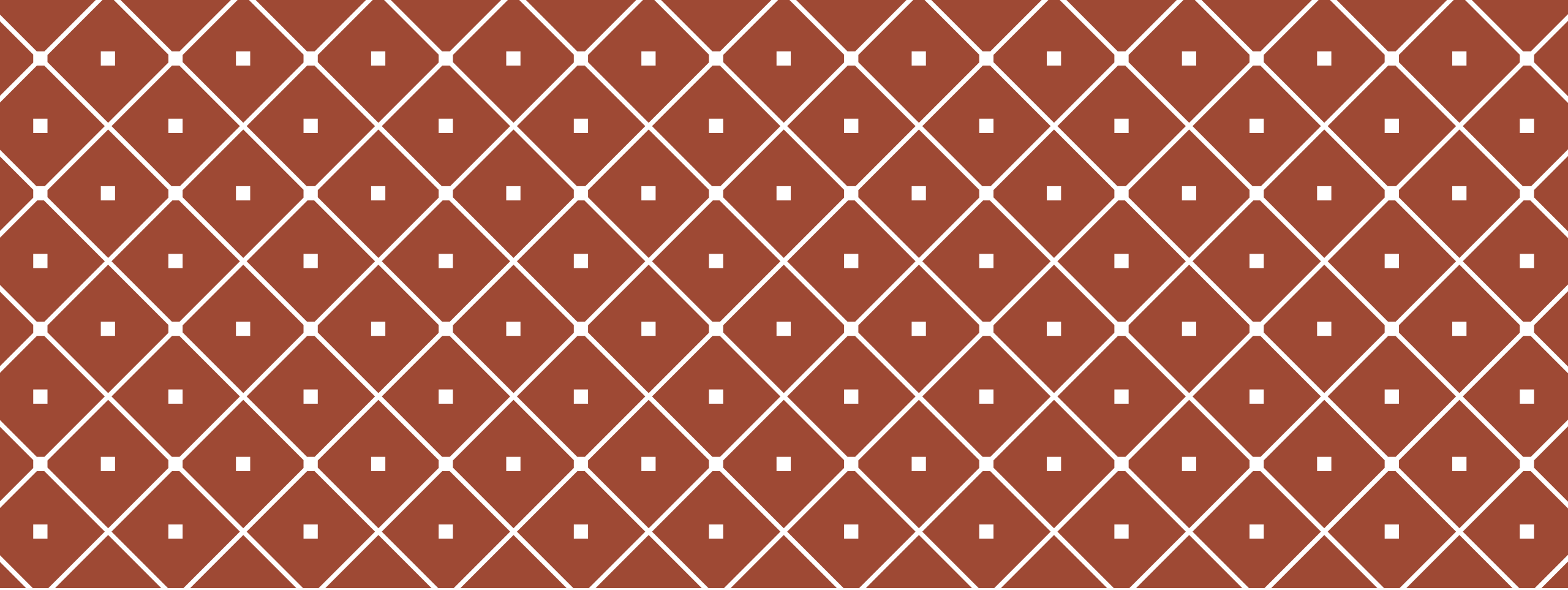
Input AdaLIN IN LN AdaLN GN

# CONCLUSIONS

The addition of the attention module and AdaLIN normalization has provided this image-to-image translation model with an advantage in comparison to other state-of-the-art methods.

This method improves the translation between image domains with a significant disparity between their shapes , this results are supported by multiple analysis on different datasets and by visual perception analysis.

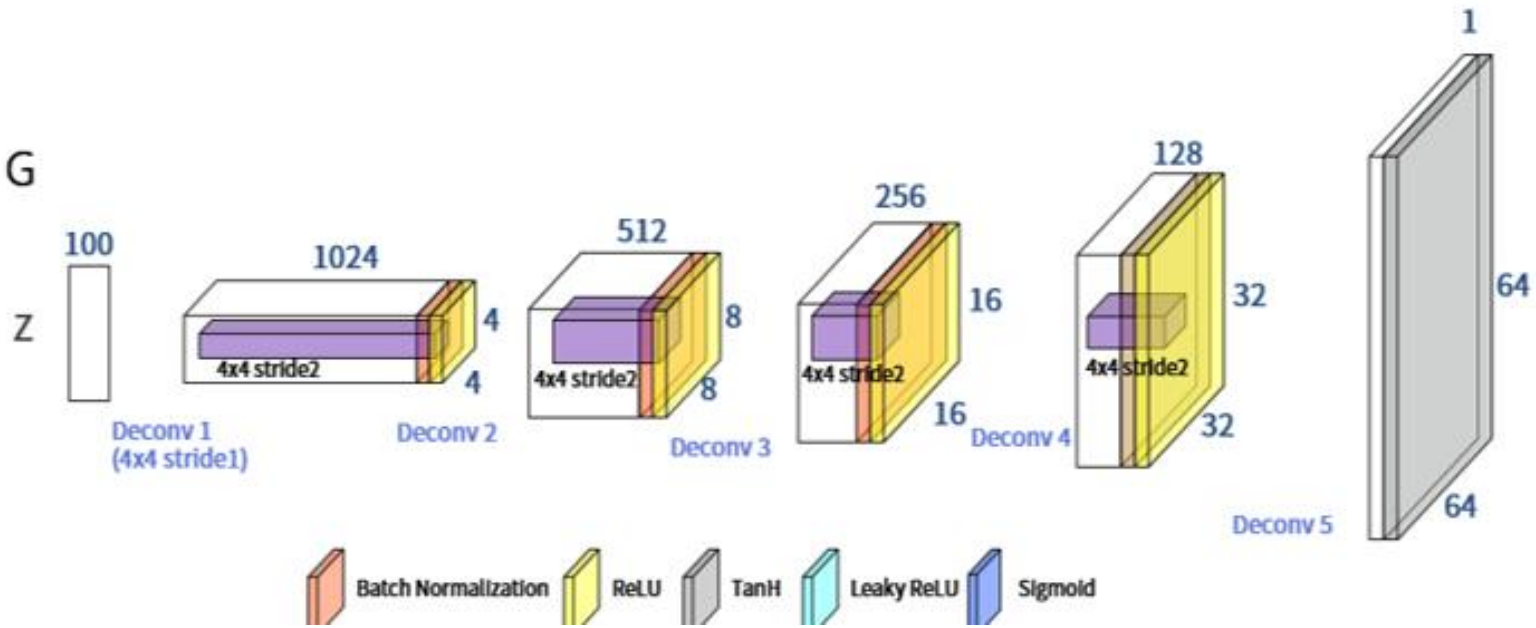




# TUTORIAL

# DCGAN: HYPERPARAMETERS

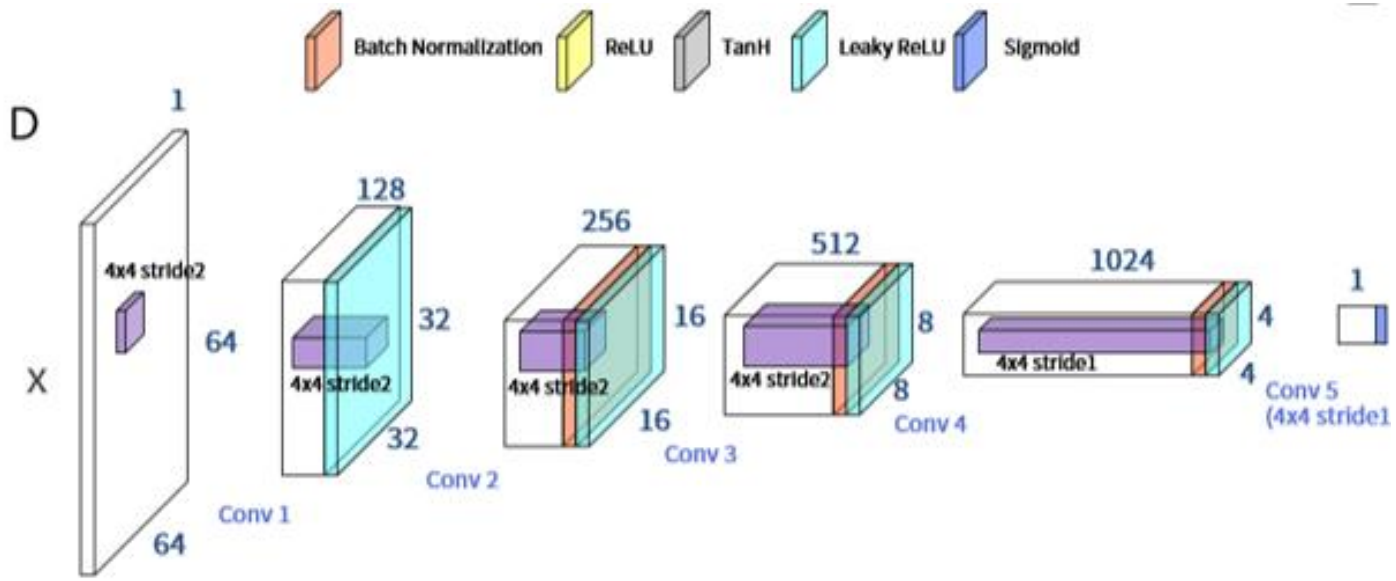
```
# Parameters
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--input", required=True, help="Path to input folder")
ap.add_argument("-r", "--results", default='results', help="Path to results folder")
ap.add_argument("-e", "--epochs", type=int, default = 20, help="Training epochs")
ap.add_argument("-lr", "--learning_rate", type=float, default=0.0002, help="Learning rate")
ap.add_argument("-is", "--img_size", type=int, default = 64,help="Image size")
ap.add_argument("-bs", "--batch_size", type=int, default = 128, help="Batch size")
ap.add_argument("-nz", "--size_vector_z", type=int, default = 100, help="Size of z latent vector")
ap.add_argument("-nc", "--num_channels", type=int, default = 3, help="Number of channels")
ap.add_argument("-ngf", "--size_feature_g", type=int, default = 64, help="Size of feature maps in generator")
ap.add_argument("-ndf", "--size_feature_d", type=int, default = 64, help="Size of feature maps in discriminator")
ap.add_argument("-b", "--beta", type=float, default = 0.5, help="Beta 1: hyperparam for Adam optimizers")
ap.add_argument("-s", "--show", type=bool, default = False, help="Show progress for each epoch")
ap.add_argument("-ng", "--ngpu", type=int, default = 1, help="Available GPU(s)")
args = vars(ap.parse_args())
```




```python
class Generator(nn.Module):
    # initializers
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
                # input is Z, going into a convolution
                nn.ConvTranspose2d(nz, ngf*8, 4, 1, 0,bias=False),
                nn.BatchNorm2d(ngf*8),
                nn.ReLU(True),
                # state size. (ngf*8) x 4 x 4
                nn.ConvTranspose2d(ngf*8, ngf*4, 4, 2, 1,bias=False),
                nn.BatchNorm2d(ngf*4),
                nn.ReLU(True),
                # state size. (ngf*4) x 8 x 8
                nn.ConvTranspose2d(ngf*4, ngf*2, 4, 2, 1,bias=False),
                nn.BatchNorm2d(ngf*2),
                nn.ReLU(True),
                # state size. (ngf*2) x 16 x 16
                nn.ConvTranspose2d(ngf*2, ngf, 4, 2, 1,bias=False),
                nn.BatchNorm2d(ngf),
                 nn.ReLU(True),
                  # state size. (ngf) x 32 x 32
                nn.ConvTranspose2d(ngf, nc, 4, 2, 1),
                nn.Tanh()
            # state size. (nc) x 64 x 64
                )
    # forward method
    def forward(self, input):
        return self.main(input)
```

# DCGAN: GENERATOR

Takes a noise vector z, of dimensions nz, and outputs an image 64x64x3.




```
class Discriminator(nn.Module):
    # initializers
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc, ndf, 4, 2, 1,bias=False),
            nn.LeakyReLU(0.2, inplace=True),
             # state size. (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf*2, 4, 2, 1,bias=False),
            nn.BatchNorm2d(ndf*2),
            nn.LeakyReLU(0.2, inplace=True),
             # state size. (ndf*2) x 16 x 16
            nn.Conv2d(ndf*2, ndf*4, 4, 2, 1,bias=False),
            nn.BatchNorm2d(ndf*4),
            nn.LeakyReLU(0.2, inplace=True),
             # state size. (ndf*4) x 8 x 8
            nn.Conv2d(ndf*4, ndf*8, 4, 2, 1,bias=False),
            nn.BatchNorm2d(ndf*8),
             nn.LeakyReLU(0.2, inplace=True),
             # state size. (ndf*8) x 4 x 4
             nn.Conv2d(ndf*8, 1, 4, 1, 0,bias=False),
                nn.Sigmoid()
                )
    # forward method
    def forward(self, input):
        return self.main(input)
```

# DCGAN: DISCRIMINATOR

Takes an image 64*64*3 and return a single value.

# DCGAN: D TRAINING

1. Pass real examples **x** through the discriminator
2. Calculate loss over D(x)
3. Generate fake samples G(z)
4. Pass fake samples G(z) through the discriminator
5. Calculate loss over D(G(z))
6. Update D

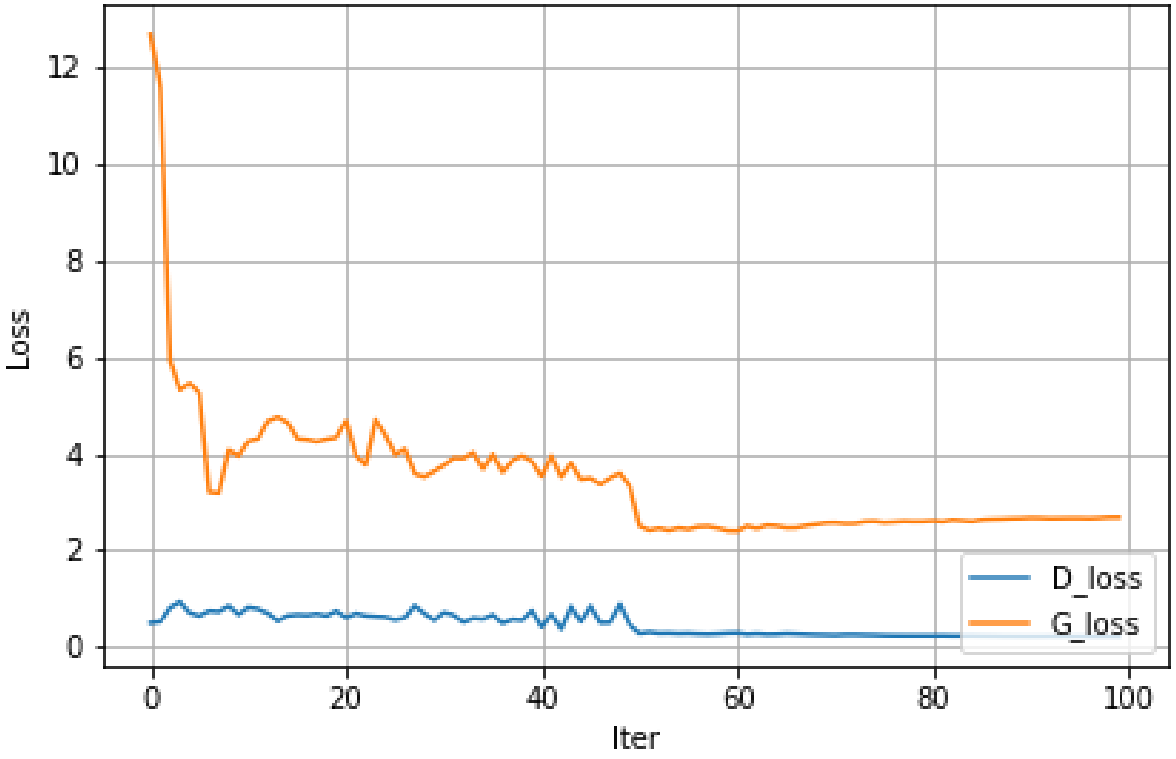
```
for i, data in enumerate(dataloader, 0):

    ############################
    # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
    ###########################
    ## Train with all-real batch
    D.zero_grad()
     # Format batch
    real_cpu = data[0].to(device)
    b_size = real_cpu.size(0)
    label = torch.full((b_size,), real_label, device=device)
     # Forward pass real batch through D
    output = D(real_cpu).view(-1)
    # Calculate loss on all-real batch
    errD_real = BCE_loss(output, label)
    # Calculate gradients for D in backward pass
    errD_real.backward()
    D_x = output.mean().item()

     ## Train with all-fake batch
    # Generate batch of latent vectors
    noise = torch.randn(b_size, nz, 1, 1, device=device)
    fake = G(noise)
    label.fill_(fake_label)
     # Classify all fake batch with D
    output=D(fake.detach()).view(-1)
     # Calculate D's loss on the all-fake batch
    errD_fake = BCE_loss(output, label)
    # Calculate the gradients for this batch
    errD_fake.backward()
    D_G_z1 = output.mean().item()
    # Add the gradients from the all-real and all-fake batches
    errD = errD_real + errD_fake
    # Update D
    D_optimizer.step()
    D_losses.append(errD.item())
```

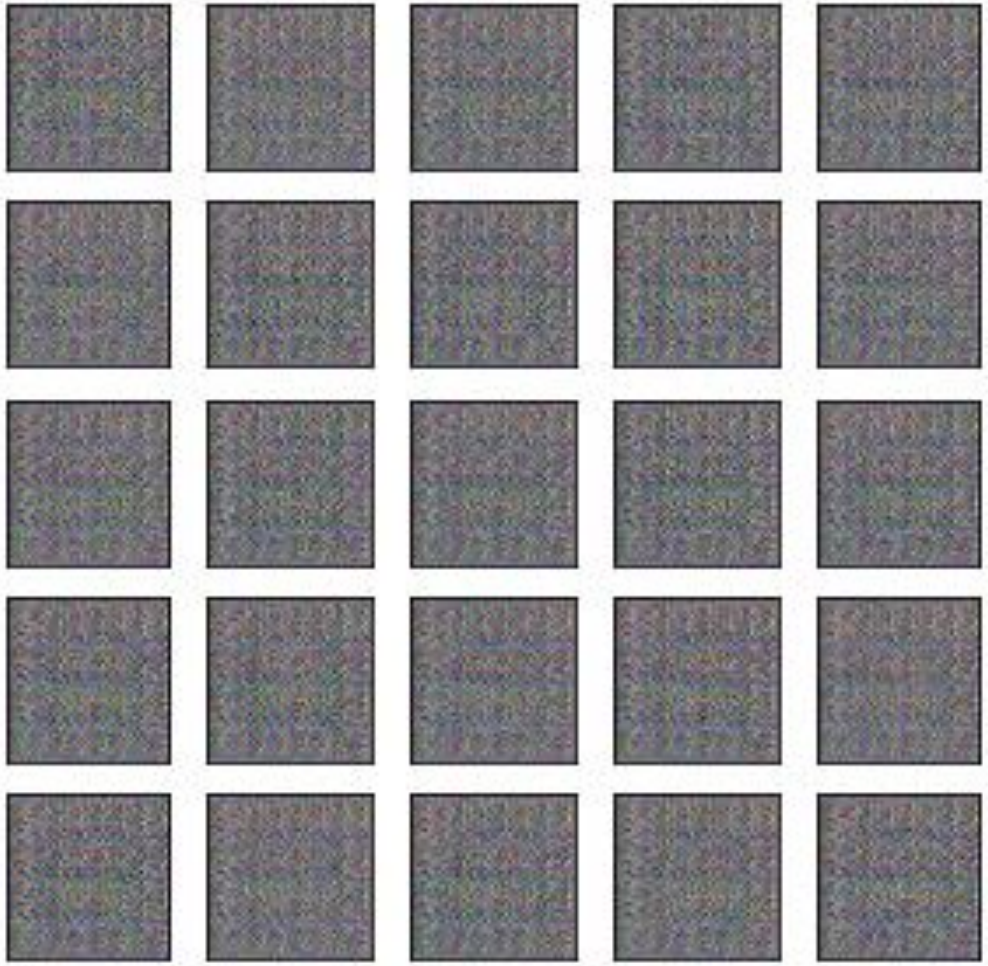
# DCGAN: G TRAINING

1. With updated D, pass fake samples G(z) through the discriminator again
2. Calculate the error D(G(z))
3. Update G

```
 ############################
# (2) Update G network: maximize log(D(G(z)))
###########################
G.zero_grad()
# fake labels are real for generator cost
label.fill_(real_label)
# Since we just updated D, perform another forward pass of all-fake batch through D
output = D(fake).view(-1)
 # Calculate G's loss based on this output
errG = BCE_loss(output, label)
# Calculate gradients for G
errG.backward()
D_G_z2 = output.mean().item()
# Update G
G_optimizer.step()

G_losses.append(errG.item())
```

# DCGAN: OUTPUT



Epoch 1

# U-GAT-IT: PARAMETERS

```
"""parsing and configuration"""

def parse_args():
    desc = "Pytorch implementation of U-GAT-IT"
    parser = argparse.ArgumentParser(description=desc)
    parser.add_argument('--phase', type=str, default='train', help='[train / test]')
    parser.add_argument('--light', type=str2bool, default=False, help='[U-GAT-IT full version / U-GAT-IT light version]'
    parser.add_argument('--dataset', type=str, default='YOUR_DATASET_NAME', help='dataset_name')

    parser.add_argument('--iteration', type=int, default=1000000, help='The number of training iterations')
    parser.add_argument('--batch_size', type=int, default=1, help='The size of batch size')
    parser.add_argument('--print_freq', type=int, default=1000, help='The number of image print freq')
    parser.add_argument('--save_freq', type=int, default=100000, help='The number of model save freq')
    parser.add_argument('--decay_flag', type=str2bool, default=True, help='The decay_flag')

    parser.add_argument('--lr', type=float, default=0.0001, help='The learning rate')
    parser.add_argument('--weight_decay', type=float, default=0.0001, help='The weight decay')
    parser.add_argument('--adv_weight', type=int, default=1, help='Weight for GAN')
    parser.add_argument('--cycle_weight', type=int, default=10, help='Weight for Cycle')
    parser.add_argument('--identity_weight', type=int, default=10, help='Weight for Identity')
    parser.add_argument('--cam_weight', type=int, default=1000, help='Weight for CAM')

    parser.add_argument('--ch', type=int, default=64, help='base channel number per layer')
    parser.add_argument('--n_res', type=int, default=4, help='The number of resblock')
    parser.add_argument('--n_dis', type=int, default=6, help='The number of discriminator layer')

    parser.add_argument('--img_size', type=int, default=256, help='The size of image')
    parser.add_argument('--img_ch', type=int, default=3, help='The size of image channel')
```

Training settings

Weights for each Loss

# U-GAT-IT: DATA LOADER

```
""" DataLoader """
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.Resize((self.img_size + 30, self.img_size+30)),
    transforms.RandomCrop(self.img_size),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
])
test_transform = transforms.Compose([
    transforms.Resize((self.img_size, self.img_size)),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
])

self.trainA = ImageFolder(os.path.join('dataset', self.dataset, 'trainA'), train_transform)
self.trainB = ImageFolder(os.path.join('dataset', self.dataset, 'trainB'), train_transform)
self.testA = ImageFolder(os.path.join('dataset', self.dataset, 'testA'), test_transform)
self.testB = ImageFolder(os.path.join('dataset', self.dataset, 'testB'), test_transform)
self.trainA_loader = DataLoader(self.trainA, batch_size=self.batch_size, shuffle=True)
self.trainB_loader = DataLoader(self.trainB, batch_size=self.batch_size, shuffle=True)
self.testA_loader = DataLoader(self.testA, batch_size=1, shuffle=False)
self.testB_loader = DataLoader(self.testB, batch_size=1, shuffle=False)
```

```
├── dataset
│   └── YOUR_DATASET_NAME
│       ├── trainA
│       │   ├── xxx.jpg (name, format doesn't matter)
│       │   ├── yyy.png
│       │   └── ...
│       ├── trainB
│       │   ├── zzz.jpg
│       │   ├── www.png
│       │   └── ...
│       ├── testA
│       │   ├── aaa.jpg
│       │   ├── bbb.png
│       │   └── ...
│       └── testB
│           ├── ccc.jpg
│           ├── ddd.png
│           └── ...
```

# U-GAT-IT: MODEL

Real network implementation of U-GAT-IT, the whole algorithm is composed of two Resnet Generators and four PatchGAN Discriminators ( 2 at different scales of Domain A, 2 for Domain B)

```
""" Define Generator, Discriminator """
self.genA2B = ResnetGenerator(input_nc=3, output_nc=3, ngf=self.ch, n_blocks=self.n_res, img_size=self.img_size, light=self.light).to(self.device)
self.genB2A = ResnetGenerator(input_nc=3, output_nc=3, ngf=self.ch, n_blocks=self.n_res, img_size=self.img_size, light=self.light).to(self.device)
self.disGA = Discriminator(input_nc=3, ndf=self.ch, n_layers=7).to(self.device)
self.disGB = Discriminator(input_nc=3, ndf=self.ch, n_layers=7).to(self.device)
self.disLA = Discriminator(input_nc=3, ndf=self.ch, n_layers=5).to(self.device)
self.disLB = Discriminator(input_nc=3, ndf=self.ch, n_layers=5).to(self.device)

""" Define Loss """
self.L1_loss = nn.L1Loss().to(self.device)
self.MSE_loss = nn.MSELoss().to(self.device)
self.BCE_loss = nn.BCEWithLogitsLoss().to(self.device)

""" Trainer """
self.G_optim = torch.optim.Adam(itertools.chain(self.genA2B.parameters(), self.genB2A.parameters()), lr=self.lr, betas=(0.5, 0.999), weight_decay=self.weight_decay)
self.D_optim = torch.optim.Adam(itertools.chain(self.disGA.parameters(), self.disGB.parameters(), self.disLA.parameters(), self.disLB.parameters()), lr=self.lr, betas=(0.5, 0.999), weight_decay=self.weight_decay)
```

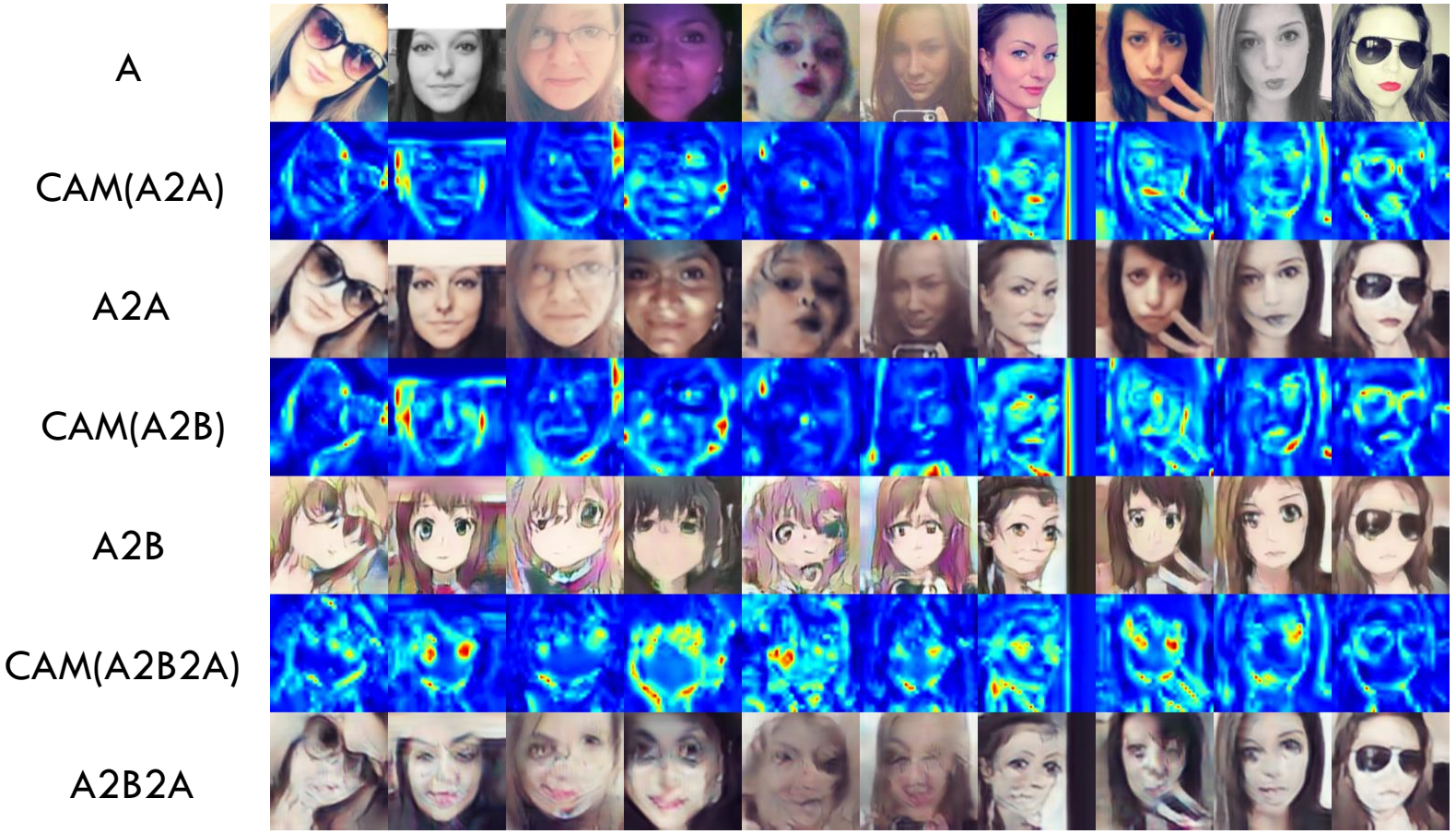
# U-GAT-IT: DISCRIMINATORS LOSSES

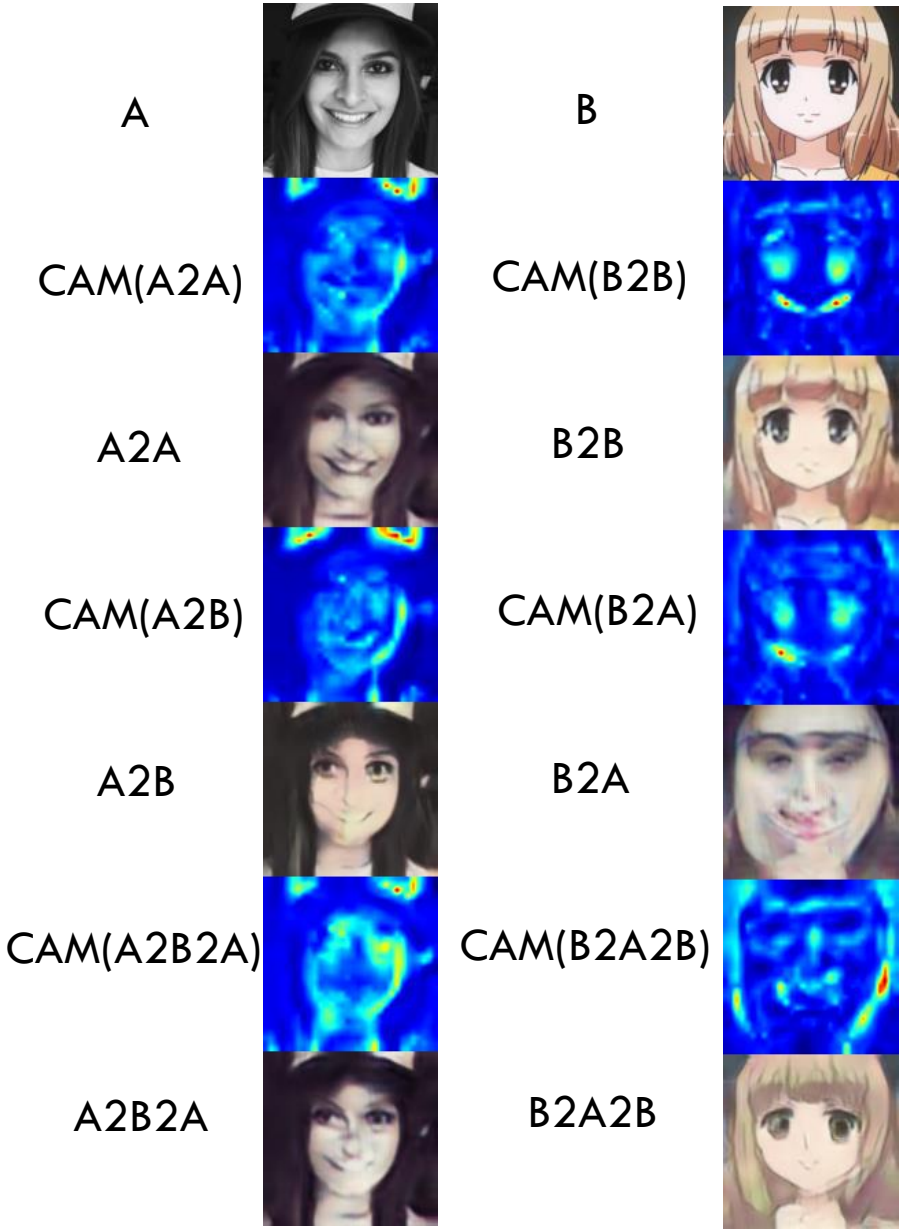
```
# Update D
self.D_optim.zero_grad()

fake_A2B, _, _ = self.genA2B(real_A)
fake_B2A, _, _ = self.genB2A(real_B)

real_GA_logit, real_GA_cam_logit, _ = self.disGA(real_A)
real_LA_logit, real_LA_cam_logit, _ = self.disLA(real_A)
real_GB_logit, real_GB_cam_logit, _ = self.disGB(real_B)
real_LB_logit, real_LB_cam_logit, _ = self.disLB(real_B)

fake_GA_logit, fake_GA_cam_logit, _ = self.disGA(fake_B2A)
fake_LA_logit, fake_LA_cam_logit, _ = self.disLA(fake_B2A)
fake_GB_logit, fake_GB_cam_logit, _ = self.disGB(fake_A2B)
fake_LB_logit, fake_LB_cam_logit, _ = self.disLB(fake_A2B)

D_ad_loss_GA = self.MSE_loss(real_GA_logit, torch.ones_like(real_GA_logit).to(self.device)) + self.MSE_loss(fake_GA_logit, torch.zeros_like(fake_GA_logit).to(self.device))
D_ad_cam_loss_GA = self.MSE_loss(real_GA_cam_logit, torch.ones_like(real_GA_cam_logit).to(self.device)) + self.MSE_loss(fake_GA_cam_logit, torch.zeros_like(fake_GA_cam_logit).to(self.device))
D_ad_loss_LA = self.MSE_loss(real_LA_logit, torch.ones_like(real_LA_logit).to(self.device)) + self.MSE_loss(fake_LA_logit, torch.zeros_like(fake_LA_logit).to(self.device))
D_ad_cam_loss_LA = self.MSE_loss(real_LA_cam_logit, torch.ones_like(real_LA_cam_logit).to(self.device)) + self.MSE_loss(fake_LA_cam_logit, torch.zeros_like(fake_LA_cam_logit).to(self.device))
D_ad_loss_GB = self.MSE_loss(real_GB_logit, torch.ones_like(real_GB_logit).to(self.device)) + self.MSE_loss(fake_GB_logit, torch.zeros_like(fake_GB_logit).to(self.device))
D_ad_cam_loss_GB = self.MSE_loss(real_GB_cam_logit, torch.ones_like(real_GB_cam_logit).to(self.device)) + self.MSE_loss(fake_GB_cam_logit, torch.zeros_like(fake_GB_cam_logit).to(self.device))
D_ad_loss_LB = self.MSE_loss(real_LB_logit, torch.ones_like(real_LB_logit).to(self.device)) + self.MSE_loss(fake_LB_logit, torch.zeros_like(fake_LB_logit).to(self.device))
D_ad_cam_loss_LB = self.MSE_loss(real_LB_cam_logit, torch.ones_like(real_LB_cam_logit).to(self.device)) + self.MSE_loss(fake_LB_cam_logit, torch.zeros_like(fake_LB_cam_logit).to(self.device))

D_loss_A = self.adv_weight * (D_ad_loss_GA + D_ad_cam_loss_GA + D_ad_loss_LA + D_ad_cam_loss_LA)
D_loss_B = self.adv_weight * (D_ad_loss_GB + D_ad_cam_loss_GB + D_ad_loss_LB + D_ad_cam_loss_LB)

Discriminator_loss = D_loss_A + D_loss_B
Discriminator_loss.backward()
self.D_optim.step()
```

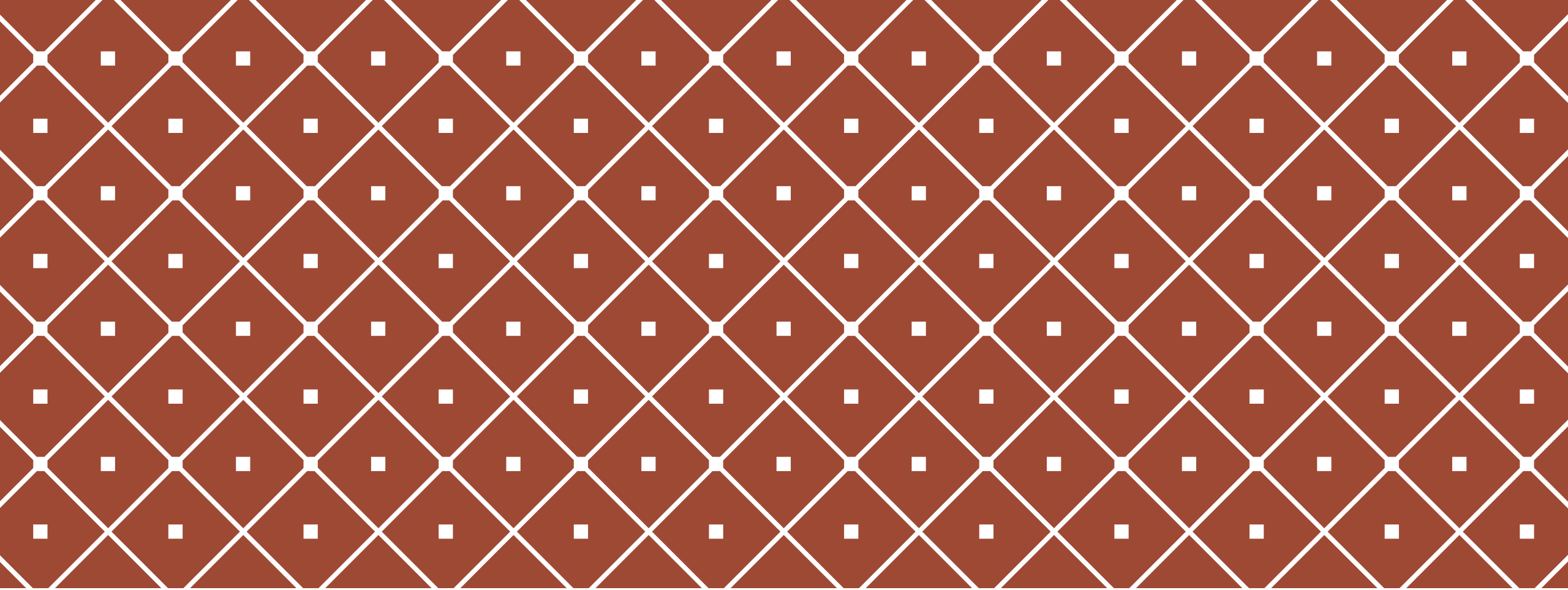
# U-GAT-IT: GENERATORS LOSSES

```
self.G_optim.zero_grad()

fake_A2B, fake_A2B_cam_logit, _ = self.genA2B(real_A)
fake_B2A, fake_B2A_cam_logit, _ = self.genB2A(real_B)

fake_A2B2A, _, _ = self.genB2A(fake_A2B)
fake_B2A2B, _, _ = self.genA2B(fake_B2A)

fake_A2A, fake_A2A_cam_logit, _ = self.genB2A(real_A)
fake_B2B, fake_B2B_cam_logit, _ = self.genA2B(real_B)

fake_GA_logit, fake_GA_cam_logit, _ = self.disGA(fake_B2A)
fake_LA_logit, fake_LA_cam_logit, _ = self.disLA(fake_B2A)
fake_GB_logit, fake_GB_cam_logit, _ = self.disGB(fake_A2B)
fake_LB_logit, fake_LB_cam_logit, _ = self.disLB(fake_A2B)

G_ad_loss_GA = self.MSE_loss(fake_GA_logit, torch.ones_like(fake_GA_logit).to(self.device))
G_ad_cam_loss_GA = self.MSE_loss(fake_GA_cam_logit, torch.ones_like(fake_GA_cam_logit).to(self.device))
G_ad_loss_LA = self.MSE_loss(fake_LA_logit, torch.ones_like(fake_LA_logit).to(self.device))
G_ad_cam_loss_LA = self.MSE_loss(fake_LA_cam_logit, torch.ones_like(fake_LA_cam_logit).to(self.device))
G_ad_loss_GB = self.MSE_loss(fake_GB_logit, torch.ones_like(fake_GB_logit).to(self.device))
G_ad_cam_loss_GB = self.MSE_loss(fake_GB_cam_logit, torch.ones_like(fake_GB_cam_logit).to(self.device))
G_ad_loss_LB = self.MSE_loss(fake_LB_logit, torch.ones_like(fake_LB_logit).to(self.device))
G_ad_cam_loss_LB = self.MSE_loss(fake_LB_cam_logit, torch.ones_like(fake_LB_cam_logit).to(self.device))

G_recon_loss_A = self.L1_loss(fake_A2B2A, real_A)
G_recon_loss_B = self.L1_loss(fake_B2A2B, real_B)

G_identity_loss_A = self.L1_loss(fake_A2A, real_A)
G_identity_loss_B = self.L1_loss(fake_B2B, real_B)

G_cam_loss_A = self.BCE_loss(fake_B2A_cam_logit, torch.ones_like(fake_B2A_cam_logit).to(self.device)) + self.BCE_loss(fake_A2A_cam_logit, torch.zeros_like(fake_A2A_cam_logit).to(self.device))
G_cam_loss_B = self.BCE_loss(fake_A2B_cam_logit, torch.ones_like(fake_A2B_cam_logit).to(self.device)) + self.BCE_loss(fake_B2B_cam_logit, torch.zeros_like(fake_B2B_cam_logit).to(self.device))

G_loss_A =  self.adv_weight * (G_ad_loss_GA + G_ad_cam_loss_GA + G_ad_loss_LA + G_ad_cam_loss_LA) + self.cycle_weight * G_recon_loss_A + self.identity_weight * G_identity_loss_A + self.cam_weight * G_cam_loss_A
G_loss_B = self.adv_weight * (G_ad_loss_GB + G_ad_cam_loss_GB + G_ad_loss_LB + G_ad_cam_loss_LB) + self.cycle_weight * G_recon_loss_B + self.identity_weight * G_identity_loss_B + self.cam_weight * G_cam_loss_B

Generator_loss = G_loss_A + G_loss_B
Generator_loss.backward()
self.G_optim.step()
```

# U-GAT-IT: OUTPUT



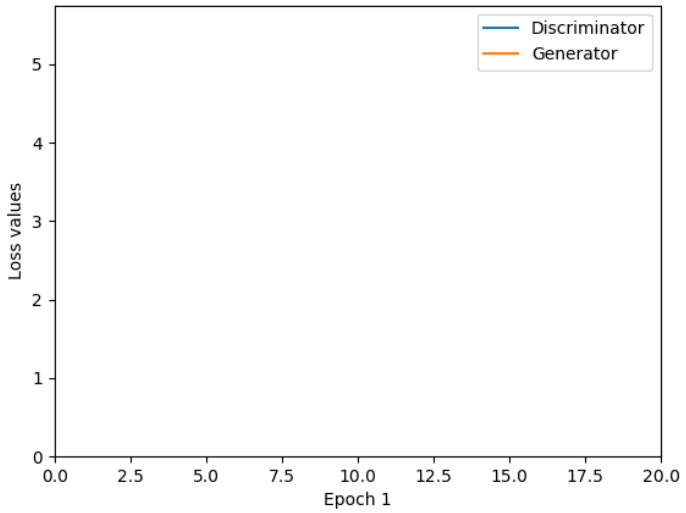


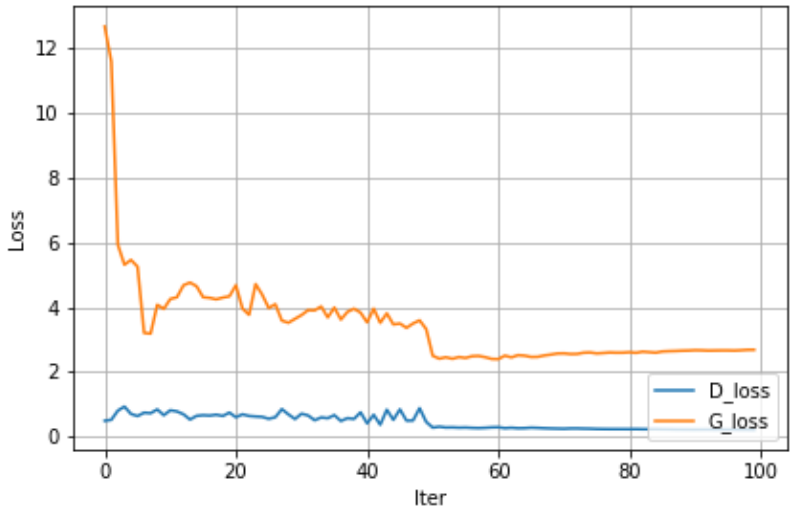
# HOMEWORK

# DCGAN: TRAINING

Successfully train a generative model on a dataset of your choosing.

- Describe how the initial training went with the default parameters and a subset of the dataset, compare it to your most successful training and its corresponding modifications and justifications.
- (Bonus) Implement a suggested technique to improve DCGANs Training (e.g. Add noise to discriminator, label swapping, label smoothing, etc.)



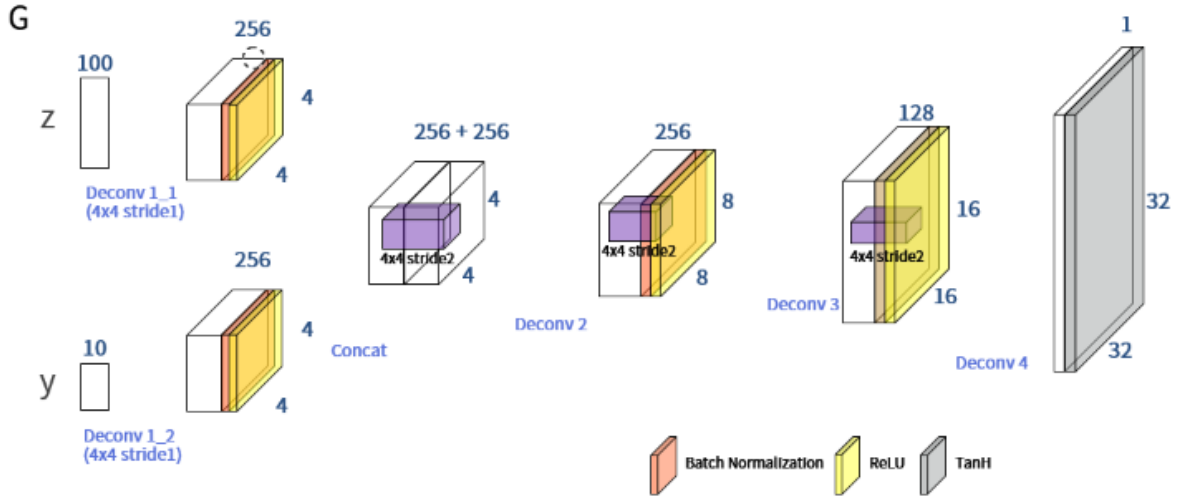
# DCGAN : TRAINING SCHEME

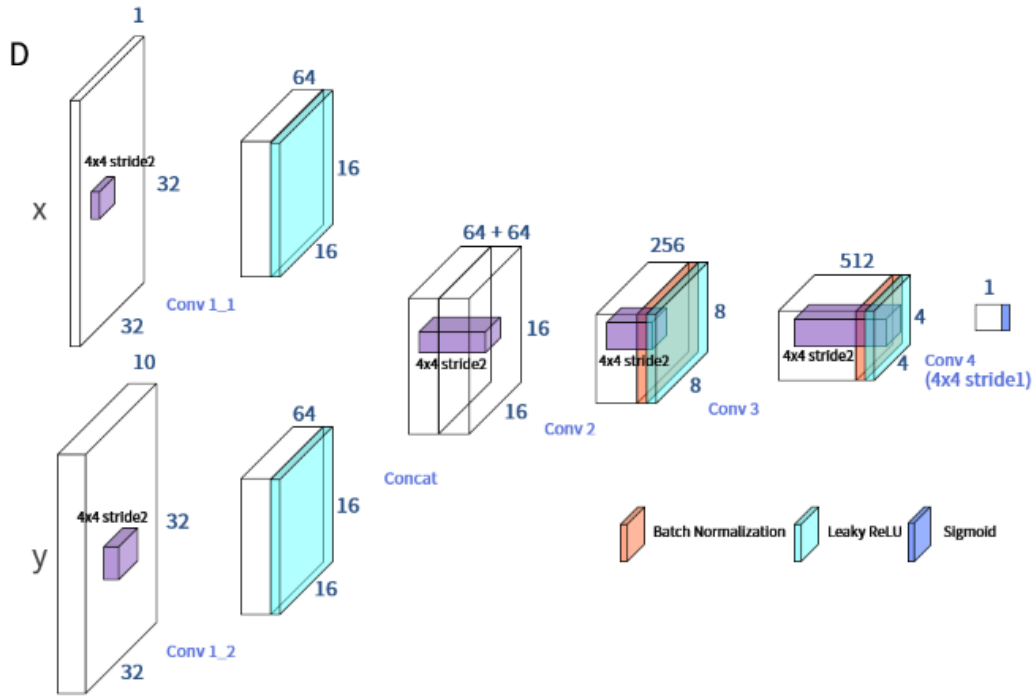
- Implement a training scheme in which the Generator and Discriminator “take turns” to update, report on how this new scheme affected the training process.
  - One of the first attempts to improve training stability
  - Examples: Update the discriminator every 2 epochs, update the generator or discriminator after a given loss value.

# C-DCGAN

Implement a condition version of the DCGAN, using the demo code as a basis, training to generate numbers with MNIST. Provide an explanation of the necessary additions, the modified code and some examples.

Recommendation: Use One Hot encoding  for the labels during preprocess, the label y must be provided to the Generator and Discriminator.



# U-GAT-IT

Training U-GAT-IT with the selfie2anime dataset, determine how important each of the loss functions are to the overall training, by changing the loss weights $\lambda$, and what perceptual effects do their modification have on the generated samples.

$$\min_{G_{s\to t}, G_{t\to s}, \eta_s, \eta_t} \max_{D_s, D_t, \eta_{D_s}, \eta_{D_t}} \lambda_1 L_{gan} + \lambda_2 L_{cycle} + \lambda_3 L_{identity} + \lambda_4 L_{cam},$$

```
parser.add_argument('--adv_weight', type=int, default=1, help='Weight for GAN')
parser.add_argument('--cycle_weight', type=int, default=10, help='Weight for Cycle')
parser.add_argument('--identity_weight', type=int, default=10, help='Weight for Identity')
parser.add_argument('--cam_weight', type=int, default=1000, help='Weight for CAM')
```

# BIBLIOGRAPHY


Goodfellow, I. (2016). *Tutorial: Generative Adversarial Networks. NIPS 2016* Retrieved from http://www.iangoodfellow.com/slides/2016-12-04-NIPS.pdf

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems* (pp. 2672-2680).

Mirza, M., & Osindero, S. (2014). Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*.

Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

Isola, P., Zhu, J. Y., Zhou, T., & Efros, A. A. (2017). Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1125-1134).

Zhu, J. Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision* (pp. 2223-2232).

# BIBLIOGRAPHY


Choi, Y., Choi, M., Kim, M., Ha, J. W., Kim, S., & Choo, J. (2018). Stargan: Unified generative adversarial networks for multi-domain image-to-image translation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 8789-8797).

Liu, M. Y., Breuel, T., & Kautz, J. (2017). Unsupervised image-to-image translation networks. In *Advances in neural information processing systems* (pp. 700-708).

Zhu, J. Y., Zhang, R., Pathak, D., Darrell, T., Efros, A. A., Wang, O., & Shechtman, E. (2017). Toward multimodal image-to-image translation. In *Advances in Neural Information Processing Systems* (pp. 465-476).

Brock, A., Donahue, J., & Simonyan, K. (2018). Large scale gan training for high fidelity natural image synthesis. *arXiv preprint arXiv:1809.11096*.

Kim, J., Kim, M., Kang, H., & Lee, K. (2019). U-GAT-IT: Unsupervised Generative Attentional Networks with Adaptive Layer-Instance Normalization for Image-to-Image Translation. *arXiv preprint arXiv:1907.10830*.

Wang, T. C., Liu, M. Y., Zhu, J. Y., Liu, G., Tao, A., Kautz, J., & Catanzaro, B. (2018). Video-to-video synthesis. *arXiv preprint arXiv:1808.06601*.