Principled training of Generative Adversarial Networks with Wasserstein Metric and Gradient Penalty

Autor: Gabriel Alfonso Patrón Herrera

Universidad Nacional de Colombia

Facultad de Ciencias
Departamento de Estadística

Table of contents

Introduction

Generative Models

Generative models are defined as algorithms that model data distributions $p(\mathrm{x})$.

Generative Modeling Framework

1. There is a data set of observations $\mathbf{x}$.
2. The observations are assumed to have been generated by some unknown distribution p_r.
3. The generative model p_g tries to learn p_r. If successful, p_g can be used to generate observations that appear to have been drawn from p_r.
4. A proper generative model should:
    4.1 Generate examples that could plausibly have been drawn from p_r.
    4.2 Generate examples different from the original observations $\mathbf{x}$.

Generative Models Tree

Generative models are defined as algorithms that model data distributions $p(\mathbf{x})$.

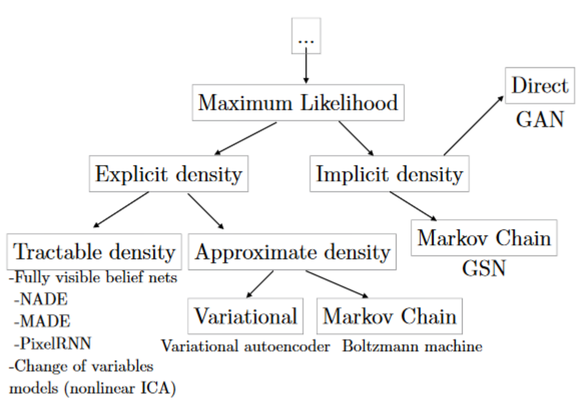


Figure 1: Caption

Generative Adversarial Networks

The Framework

Two deep neural networks compete with each other. The Generator, G tries to output fake data that seems as real as possible while the Discriminator, D, tries to correctly classify real data as real and fake data (provided by G) as fake.

The Definitions

- We have P_{data}
- Z is latent dimension
- We define $G : Z \to X$, where X is the domain of P_{data}
- We define $D : X \to [0, 1]$
- G and D compete. P_g[1] converges to P_{data} in the limit.

[1] P_g is the distribution induced by G

The Equation

$$\min_G \max_D E_{x \sim p_r(x)}[\log(D(x)] + E_{z \sim p_z(z)}[\log(1 - D(G(z)))] \qquad (1)$$

The Ideal

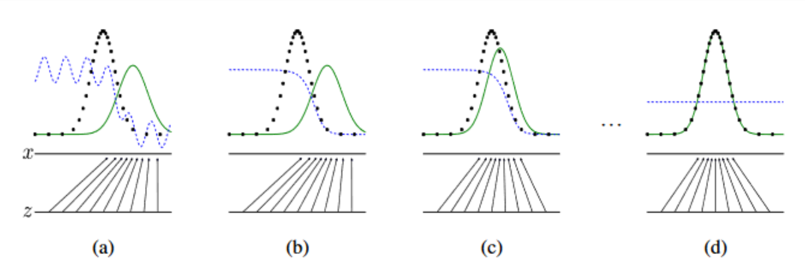


Figure 2: GAN Progression

The Architecture

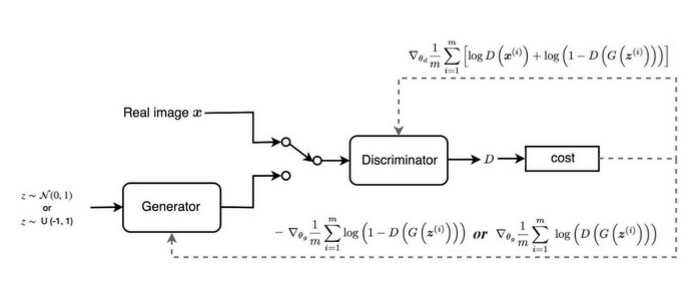


Figure 3: GAN architecture

The Algorithm

Algorithm 1: GAN Algorithm

Require: The number of steps to apply to the discriminator, k, is a hyperparameter. The number of samples in the minibatch, m.;

for *number of training iterations* **do**

  for *k steps* **do**

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

$$\nabla_{\theta_d} \frac{1}{m} \sum_{n=1}^{m} log(D(x^i) + log(1 - D(G(z^i)))$$

  end

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

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

end

The Problems

The biggest problem facing GANs is the issue of non-convergence [15]. GANs have a number of common failures and all of these are active areas of research.

Hard to Achieve Nash Equilibrium

Two agents are trained to find a Nash Equilibrium to a two-player non-cooperative game, and as such, updating via gradient descent does not guarantee convergence due to the high-dimensional, non-convex nature of the cost function induced by the parameters

Low Support Manifolds

The dimensions of real world data sets usually have their support on low dimensional manifolds. Intuitively, p_{data} and p_g have supports that rest in low dimensional manifolds and almost certainly are going to be disjoint. This phenomenon has the effect of making the distances between distributions work poorly because they have disjoint supports

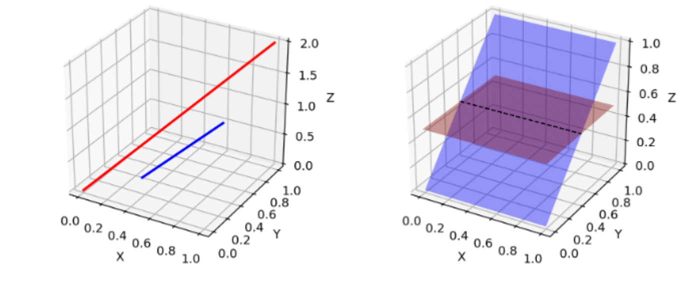


Figure 4: Low Dimensional Supports. In high dimensional spaces, the support of the real distribution P_r and the generated distribution P_g are often disjoint, as observed on the left, or negligible (intersection has measure 0), as on the right. Image source [17]

Vanishing Gradients

p_{data} and p_g have supports that rest in low dimensional manifolds and almost certainly are going to be disjoint and therefore we are always capable of finding a so-called *Perfect Discriminator*[2]. A perfect Discriminator produces vanishing gradients, as shown in

[2] A perfect discriminator is a discriminator that separates real and fake samples correctly every time.

Low Support Manifolds

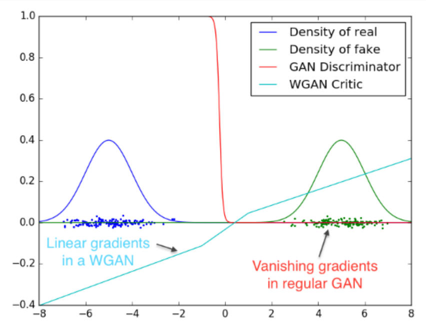


Figure 5: Various gradients that correspond to GAN and WGAN losses. It should be noted that an almost zero gradient is obtained via regular GAN methods, as illustrated by a red line. A linear (constant, non-zero, gradient) is obtained with WGAN methods, as described by the teal line. Image source: [2]

Mode Collapse

The generator's task is successfully completed whenever it can trick the discriminator into classifying fake samples as real. Such a task *does not imply generalization*. The generator can perform very well even when it is stuck in small spaces with very low variety, as seen in Figure 6.

Mode Collapse

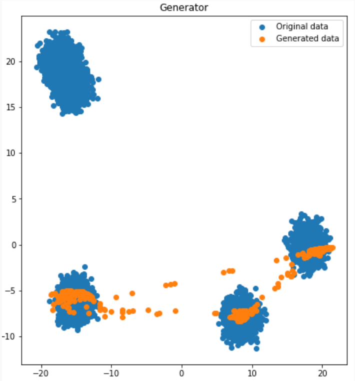


Figure 6: An instance of Mode Collapse, the Generator gets stuck in the lowest blobs of the mixture, unable to reproduce the data from the upper blobs. In this situation, it can still fool the discriminator as long as it closely reproduces the data in these three blobs.

Lack of a proper evaluation metric

Gradient descent methods depend on a series of discernible objective functions that work for a predefined task. Either a *Regression* or *Classification* model's performance can usually be tracked.

Lack of a proper evaluation metric

How to ”objectively” measure a generative model’s output? How can we compare them?

Empirical Recommendations [13]

- Use Strided Convolutions.

Empirical Recommendations [13]

- Use Strided Convolutions.
- Remove fully-connected layers.

Empirical Recommendations [13]

- Use Strided Convolutions.
- Remove fully-connected layers.
- Use batch Normalization.

Empirical Recommendations [13]

- Use Strided Convolutions.
- Remove fully-connected layers.
- Use batch Normalization.
- Use ReLu, Leaky ReLu, and Tanh.

Empirical Recommendations [13]

- Use Strided Convolutions.
- Remove fully-connected layers.
- Use batch Normalization.
- Use ReLu, Leaky ReLu, and Tanh.
- Use Adam Optimization.

Empirical Recommendations [13]

- Use Strided Convolutions.
- Remove fully-connected layers.
- Use batch Normalization.
- Use ReLu, Leaky ReLu, and Tanh.
- Use Adam Optimization.
- Train with labels.

Empirical Recommendations [13]

- Use Strided Convolutions.
- Remove fully-connected layers.
- Use batch Normalization.
- Use ReLu, Leaky ReLu, and Tanh.
- Use Adam Optimization.
- Train with labels.
- Balance G and D.

Theoretical Recommendations [1]

- Add noise to Discriminator Input.

Theoretical Recommendations [1]

- Add noise to Discriminator Input.
- Use a weaker distance metric.

What does the original GAN actually do?

The original GAN *minimizes the JS divergence* (defined below) between the distribution of real and fake examples [7]

Distance Between Distributions

There are many ways to measure them:

- Total variation distance

$$\delta(P_r, P_g) = \sup_{A \in \Sigma} |P_r(A) - P_g(A)| \tag{2}$$

- Kullback-Leibler (KL) divergence

$$KL(P_r \| P_g) = \int \log(\frac{P_r(x)}{P_g(x)}) P_r(x) d\mu(x) \tag{3}$$

  Both P_r and P_g are assumed to be absolutely continuous with respect to the same measure μ defined on X. KL divergence is asymmetric and can have infinite values.

Distance Between Distributions

There are many ways to measure them:

- Jensen-Shannon (JS)

$$JS(P_r, P_g) = KL(P_r \| P_m) + KL(P_g \| P_m) \tag{4}$$

  where P_m is the mixture $\frac{P_r + P_g}{2}$. This divergence is symmetrical.

- Eart-Mover or Wasserstein distance

$$W(P_r, P_g) = \inf_{\gamma \in \prod(P_r, P_g)} E_{(x,y) \sim \gamma}[\|x - y\|] \tag{5}$$

  Where $\prod(P_r, P_g)$ denotes the set of all joint distributions $\gamma(x, y)$ whose marginals are respectively P_r and P_g. The Wasserstein distance is in itself an optimization problem representing the minimum cost of transporting mass to convert one distribution into another.

The Impact of Choice Between Them

The most fundamental difference between them is their impact on the convergence of sequences of probability distributions. Wasserstein distance is weakest among these[3] . [2].

[3]The relative strength of an induced topology understood as how hard it is for sequences of probability distributions to converge.

Example

Let $Z \sim \mathbb{U}[0, 1]$ the uniform distribution on the unit interval. Let P_0 be the distribution of $(0, Z) \in \mathbb{R}^2$, uniform on a straight vertical line passing through the origin. Let $g_\theta(z) = (\theta, z)$ with θ a single real parameter.

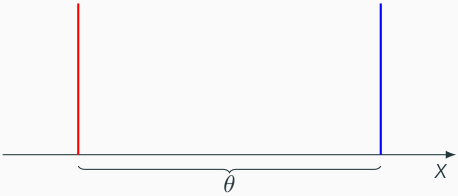


Figure 7: Parallel lines separated by a horizontal distance θ.

Example Distances

$$W(P_0, P_\theta) = |\theta|$$

$$KL(P_0, P_\theta) = \begin{cases} +\infty, & \text{if } \theta \neq 0 \\ 0, & \text{if } \theta = 0 \end{cases}$$

$$JS(P_0, P_\theta) = \begin{cases} \log(2), & \text{if } \theta \neq 0 \\ 0, & \text{if } \theta = 0 \end{cases}$$

$$\delta(P_0, P_\theta) = \begin{cases} 1, & \text{if } \theta \neq 0 \\ 0, & \text{if } \theta = 0 \end{cases}$$

Distances

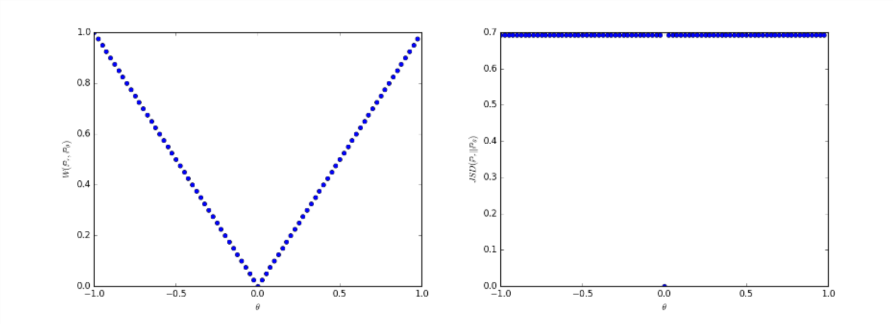


Figure 8: From left to right, distance as measured by Wasserstein and Jensen-Shannon divergence between the distributions in **Example 1**. Image source: [2]

The Real Problem

This result holds not only when distributions have disjoint supports but also, more generally, when their intersection is contained in a set of measure zero, which is almost always the case when two low dimensional manifolds intersect [1].

Why Wasserstein then?

Wasserstein distance between the distribution of real and fake examples is continuous everywhere and differentiable almost everywhere [2] and thus provides a reliable linear gradient that guarantees constant updates to the generator's parameters.

What Issues Arise?

Direct application of Wasserstein distance definition poses a highly intractable optimization problem.

Kantorovich-Rubinstein

The Kantorovich-Rubinstein duality, which tells us that:

$$W(P_r, P_g) = \sup_{||f||_L \leq 1} E_{\mathbf{x} \sim P_r}[f(\mathbf{x})] - E_{\mathbf{z} \sim P_z}[f_W(g_\theta(\mathbf{z}))] \tag{6}$$

where f belongs to the 1-Lipschitz set of functions such that $f : X \rightarrow \mathbf{R}$.

A Caveat

If f belongs to the class of K-Lipschitz functions, then the expression above corresponds to the Wasserstein distance multiplied by K [2].

Existence of f

Theorem (Existence of f)
Let P_r be any distribution. Let P_θ be the distribution of $g_\theta(Z)$ with Z being a random variable with density p and g_θ a function satisfying Assumption 1[4].

Then, there is a solution $f : X \to \mathbb{R}$ to the problem

$$\max_{||f||_L \leq 1} E_{x \sim P_r}[f(x)] - E_{x \sim P_\theta}[f(x))]$$

and we have

$$\nabla_\theta W(P_r, P_g) = -\mathbb{E}_{z \sim p(z)}[\nabla_\theta f(g_\theta(z)]$$

when both terms are well-defined.

---

[4]See Appendix

W must satisfy compactness

- f[5], parametrized by w, that solves this problem has its weights lying on a compact space.

[5] At this point, the name is changed from Discriminator to Critic

W must satisfy compactness

- f[5], parametrized by w, that solves this problem has its weights lying on a compact space.
- Compactness enforces the K-Lipschitz condition f_w for every $w \in W$ for some K that depends only on W and the Critic's capacity [2].

[5] At this point, the name is changed from Discriminator to Critic

How to enforce compactness?

Weight Clipping[6]

[6] Element-wise values to a range of values determined and therefore making all parameters be inside the $[-c, c]$ interval.

In Summary

- The change in objective function:

$$\min_{G} \max_{||f||_L \leq 1} E_{\mathbf{x} \sim P_r}[f(\mathbf{x})] - E_{\mathbf{z} \sim P_z}[f_W(G_\theta(\mathbf{z}))]$$

In Summary

- The change in objective function:

$$\min_{G} \max_{||f||_L \leq 1} E_{\mathbf{x} \sim P_r}[f(\mathbf{x})] - E_{\mathbf{z} \sim P_z}[f_W(G_\theta(\mathbf{z}))]$$

- The clipping determined by hyperparameter c.

WGAN Architecture

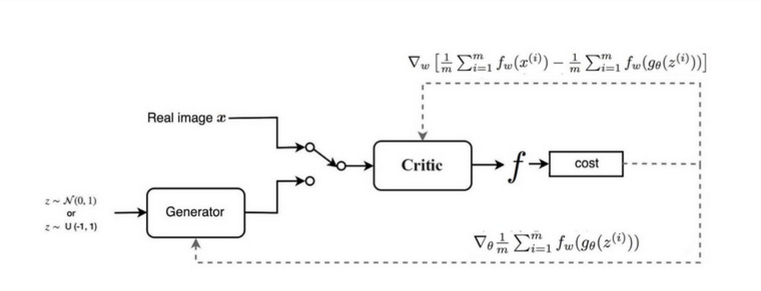


Figure 9: Wasserstein GAN architecture

Improvements

Several improvements are reported with respect to traditional GAN [2]:

- Improved stability

Improvements

Several improvements are reported with respect to traditional GAN [2]:

- Improved stability
- Less mode collapse

Improvements

Several improvements are reported with respect to traditional GAN [2]:

- Improved stability
- Less mode collapse
- Discriminator's Loss correlated with Image Quality

A Direct Quote

'Weight clipping is clearly a terrible way of enforcing compactness' [2]

What to do?

Enforce Gradient Penalty based on the following results [8]

Wasserstein GAN's gradient properties

Theorem (WGAN Gradient's Property)
Let P_r and P_g be any two distributions in X, a compact space. Then there is a 1-Lipschitz function f^* which is the optimal solution of the expression $\max_{||f||_L \leq 1} E_{y \sim P_r}[f(y)] - E_{x \sim P_\theta}[f(x)]$. Let π be the optimal coupling between P_r and P_g, the minimizer of

$$W(P_r, P_g) = \inf_{\pi \in \prod(P_r, P_g)} E_{(x,y)\sim\gamma}[||x - y||],$$

where $\prod(P_r, P_g)$ is the set of joint distributions $\pi(x, y)$. If f^* is differentiable and $\pi(x = y) = 0$, then it holds that $P_{(x,y)\sim\pi}[\nabla f^*(x_t) = \frac{y - x_t}{||y - x_t||}] = 1$ where $x_t = tx + (1 - t)y$ with $0 \leq t \leq 1$.

Corollary

Corollary
f^* has gradient norm 1 almost everywhere under P_r and P_g.

The New Crtic

The new Critic Loss the becomes:

$$L_D = E_x[D(\mathsf{x})] - E[D(G(\mathsf{z}))] + \gamma(||\nabla_{\hat{\mathsf{x}}} D(\hat{\mathsf{x}})|| - 1)^2 \tag{7}$$

where the sampling distribution $P_{\hat{\mathsf{x}}}$ samples uniformly along straight lines between pairs of points from P_r and P_g:

$$\hat{\mathsf{x}} = \epsilon \mathsf{x} + (1 - \epsilon) G(\mathsf{z}) \tag{8}$$

In Summary

- The optimal WGAN critic has unit gradient norm almost everywhere under the real and generated data distributions

In Summary

- The optimal WGAN critic has unit gradient norm almost everywhere under the real and generated data distributions
- Constrain the critic’s gradient norm.

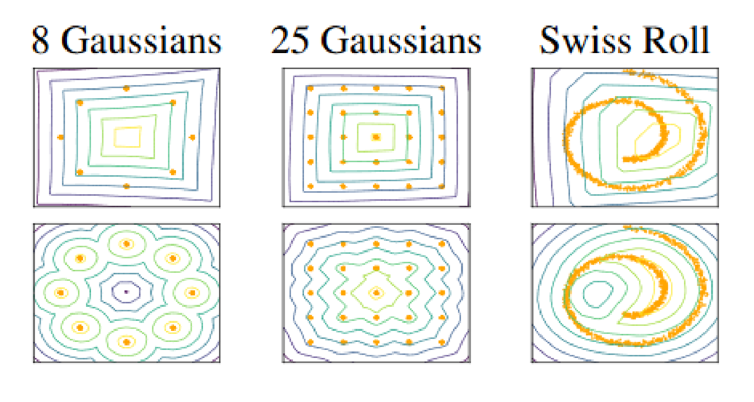


Figure 10: Top row: Wasserstein GAN's learned value surfaces trained to optimality in several toy datasets. Bottom row: Wasserstein GAN with Gradient Penalty value surfaces trained on the same toy datasets. Image source: [8]

The WGAN-GP Algorithm

Algorithm 2: WGAN-GP Algorithm

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

while *θ has not converged* **do**

  for $t = 0, \ldots, n_{critic}$ **do**

    for $i = 1, \cdot, m$ **do**

- Sample real data $\mathbf{x} \sim P_r$
- Sample latent variable $\mathbf{z} \sim p(\mathbf{z})$
- Sample a random number $\epsilon \sim U[0, 1]$
- $\tilde{\mathbf{x}} \leftarrow G_\theta(z)$
- $\hat{\mathbf{x}} \leftarrow \epsilon \mathbf{x} + (1 - \epsilon)\tilde{\mathbf{x}}$
- $L^{(i)} \leftarrow D_w(\tilde{x}) - D_w(\mathbf{x}) + \lambda(||\nabla_{\hat{\mathbf{x}}} D_w(\hat{\mathbf{x}})||_2 - 1)^2$

    end

    $w \leftarrow Adam(\nabla_w \frac{1}{m} \sum_{i=1}^{m} L^{(i)}, w, \alpha, \beta_1, \beta_2)$

  end

- Sample a batch of latent variables $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$.
- $\theta \leftarrow Adam(\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} -D_w(G_\theta(\mathbf{z})), \theta, \alpha, \beta_1, \beta_2)$

end

Evaluation Measures

Inception Score and Frechet Inception Distance both depend on InceptionV3 network. A pre-trained deep learning neural network model for image classification.

Inception Score

1. The images generated should contain clear objects, or $p(y|x)$ should be low entropy. The Inception Network should be highly confident there is a meaningful object in the image.

Inception Score

1. The images generated should contain clear objects, or $p(y|x)$ should be low entropy. The Inception Network should be highly confident there is a meaningful object in the image.
2. The generative model should output a high diversity of images from all the different classes in ImageNet, or $p(y)$ should be high entropy.

Inception Score

The generator model should reliably output a high variety of high-quality meaningful objects.

Frechet Inception Distance

The distance between the distributions (real and generated) is then calculated using the *Frechet Distance*[7]:

$$FID = \|\mu_r - \mu_g\|^2 + Tr(\Sigma_r + \Sigma_g - 2(\Sigma_r \Sigma_g)^{\frac{1}{2}}) \tag{9}$$

where $X_r \sim \mathbb{N}(\mu_r, \Sigma_r)$ and $X_g \sim \mathbb{N}(\mu_g . \Sigma_g)$ are the 2048-dimensional activations of the Inception-v3 pool3 layer [9].

[7] Also known as Wasserstein-2 distance

Experiments

Toy Data

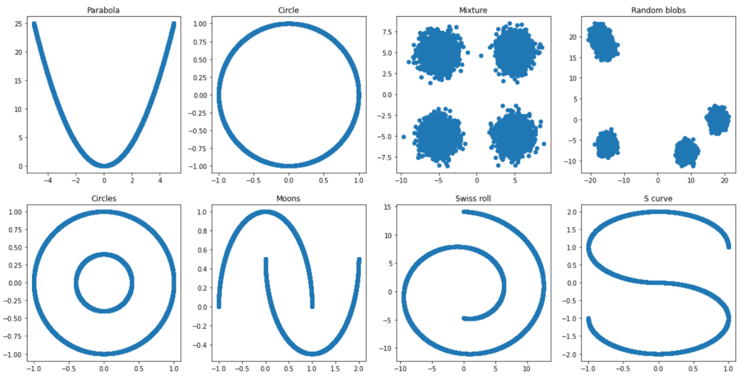


Figure 11: Toy Data used for this Section

Parabolas

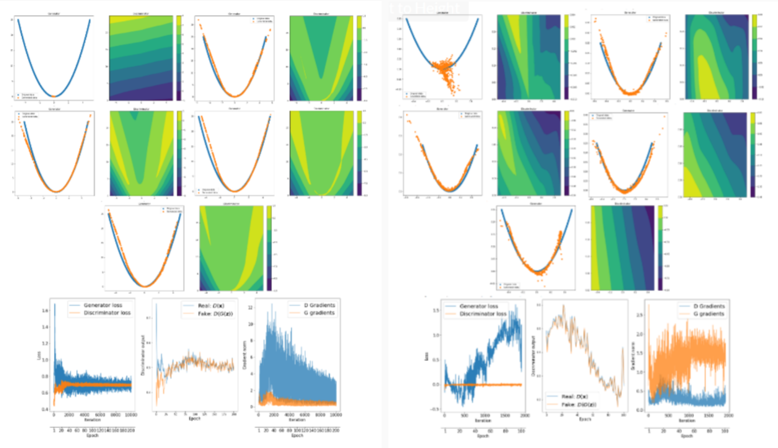


Figure 12: Parabola distribution

Circle

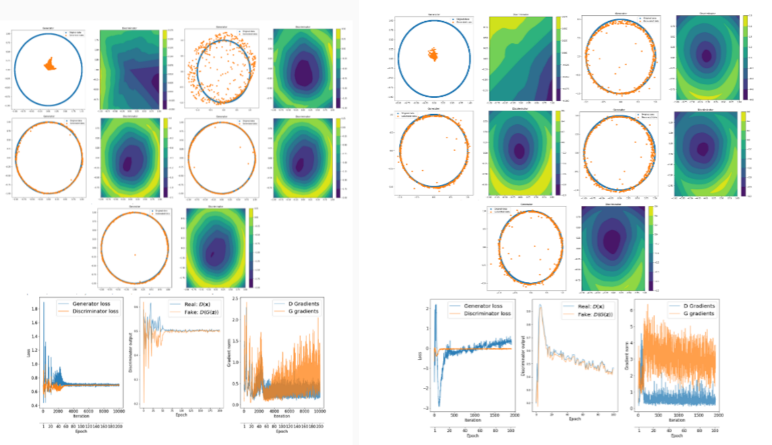


Figure 13: Circle distribution

Mixture

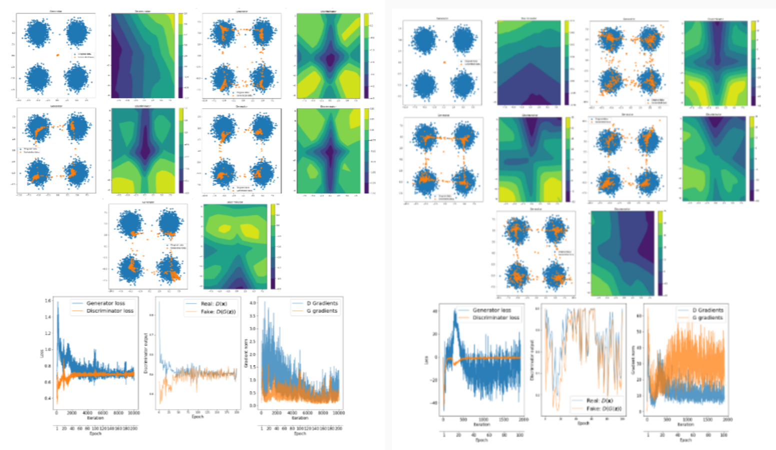


Figure 14: Mixture distribution

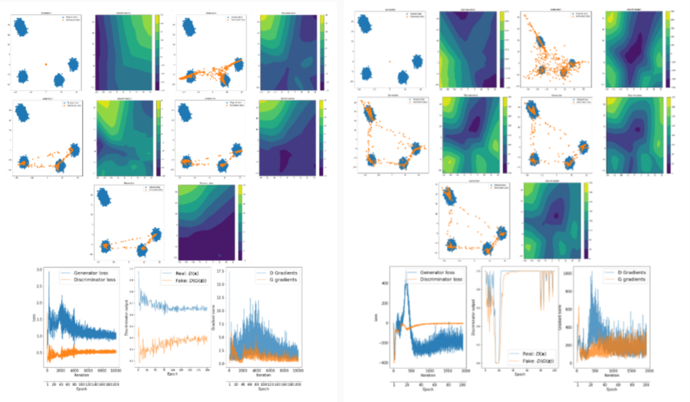


Figure 15: Random Blobs distribution

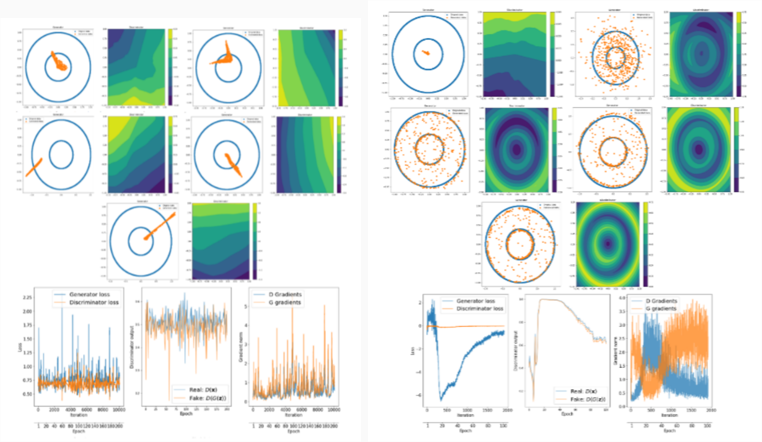


Figure 16: Circles distribution

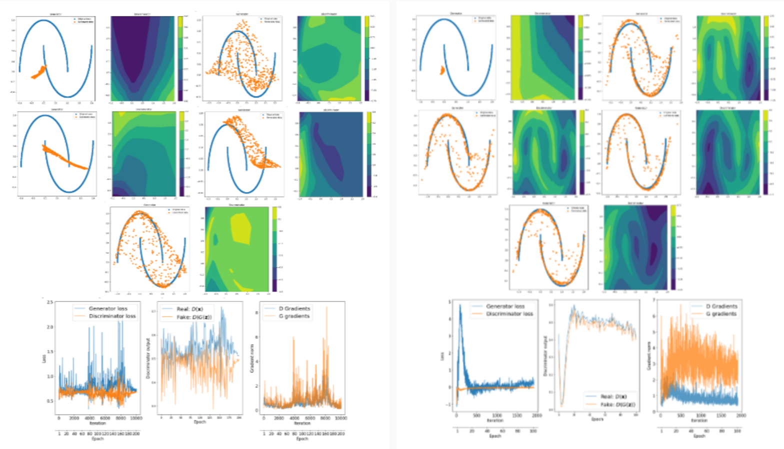


Figure 17: Moons distribution

Spiral

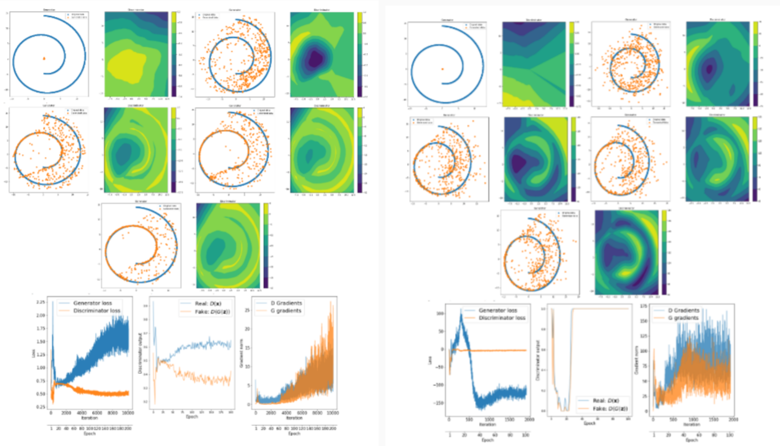


Figure 18: Spiral distribution

S Curve

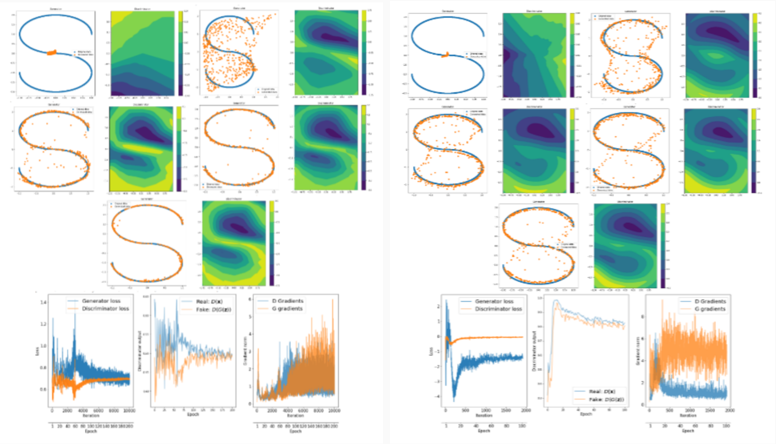


Figure 19: S curve distribution

Applications

Sample of MNIST

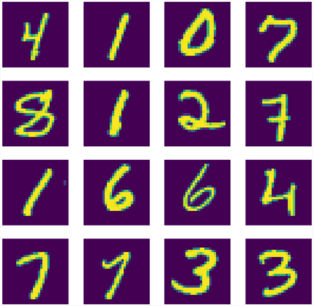


Figure 20: A sample from MNIST

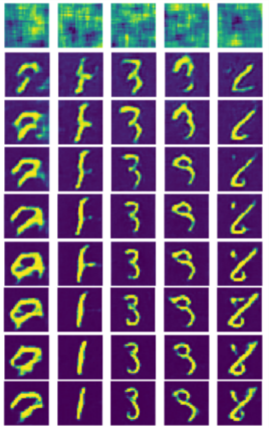


Figure 21: MNIST progression for the same z sampled evey 5 epochs. Epochs 0 to 40 shown.

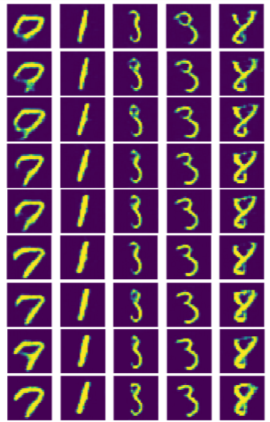


Figure 22: MNIST progression for the same z sampled evey 5 epochs. Epochs 45 to 95 shown.

MNIST

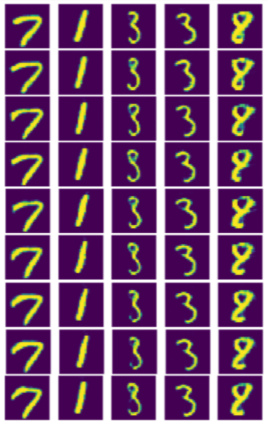


Figure 23: MNIST progression for the same z sampled evey 5 epochs. Epochs 100 to 140 shown.

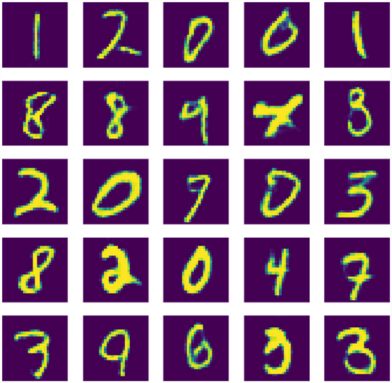


Figure 24: MNIST Sample results.

MNIST plots

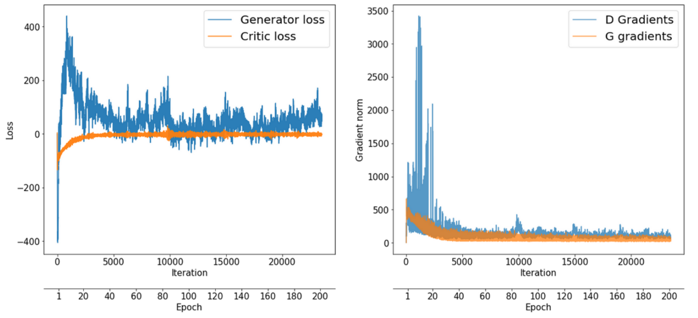


Figure 25: MNIST plots

Figure 26: A sample from CARS196 [10]

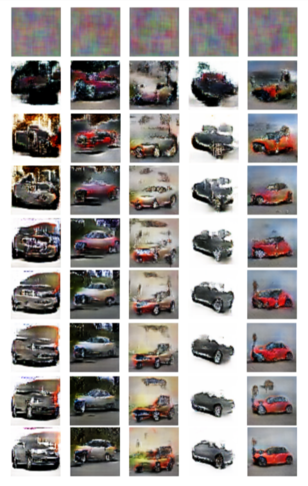


Figure 27: Samples from CARS Generator. Fixed z, sampled every 50 epochs from 0 to 400.

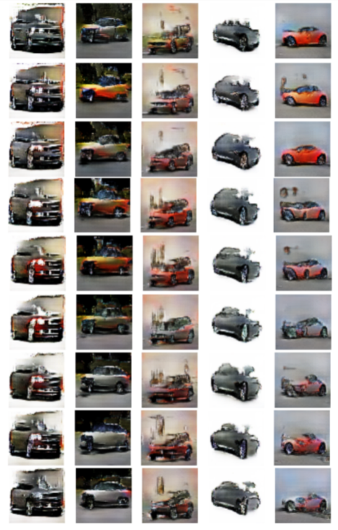


Figure 28: Samples from CARS Generator. Fixed z, sampled every 50 epochs from 450 to 900.

CARS Results

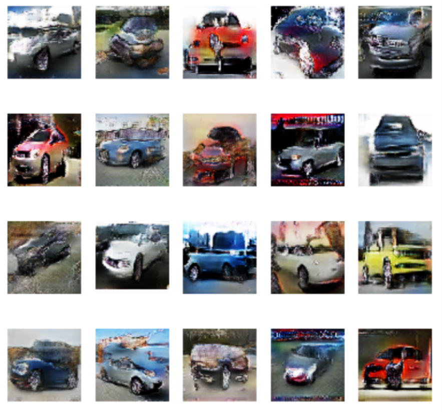


Figure 29: Samples from CARS Generator.

Additional Remarks

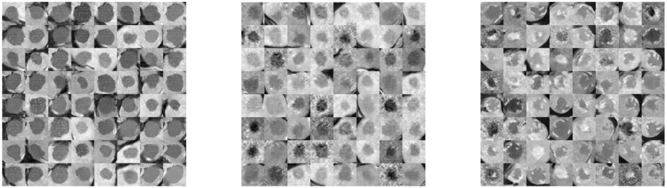
Generalization

Generalization properties of GANs are not well understood [16]. In [3], it is shown that training of GANs may not have good generalization properties, in contrast to what was suggested in the foundational paper [7], that is, GANs learn the distribution if the *deep nets* are sufficiently large. Theoretical analysis in [3] showed that the training objective can approach its optimum value even if the generated distribution suffers from *mode collapse*.

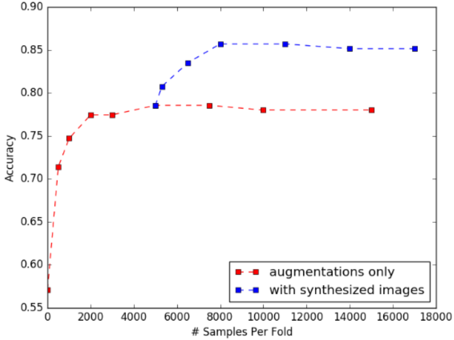
The GAN Zoo

There are many types of GANs and it is hard to classify them. They can be classified according to their objective (e.g. MuseGAN for music) and their architecture (e.g. SN-GAN [12]) or their type (e.g. Conditional GAN [11])

Medical Data Augmentation



(a) (b) (c)



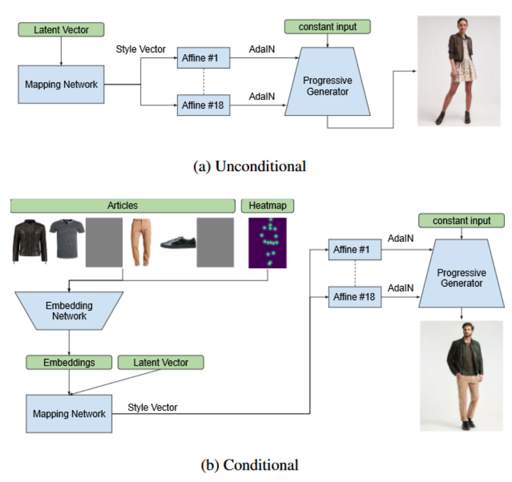


Figure 31: Generated Models. Source: Zalando Research

Text to Image Synthesis

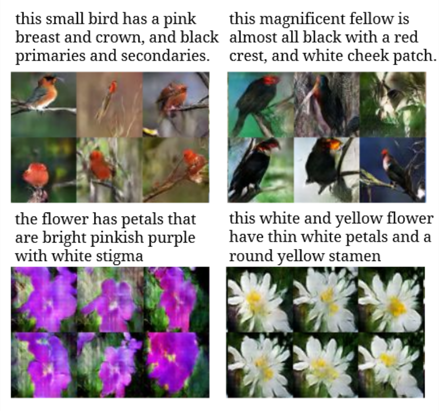


Figure 32: Text to image synthesis for flowers and birds. Source: [14]

Video Generation

Figure 33: 48 frames of generated watermelon video. Source: [5]

State of the Art

GAN research has been overwhelmingly focused on image generation, the state of the art, as of today, corresponds to Latent Optimization GAN [18], which applies the SN-GAN methodology with latent optimization. Previous SOTA was BigGAN [4], that used only SN-GAN and the biggest architecture to date.

Conclusion

Conclusion

GAN's and their variants have proven to be a Generative model with very high potential and applicability. From data augmentation to media synthesis, this generative modeling framework is growing each day, improving and breaking new barriers. There are still some difficult challenges ahead but steady progress is being made towards solving these issues; although the GAN world can look messy and aimless, several **winning** ideas are starting to emerge.

Questions?

Assumption 1

Let $g : Z \times \mathbb{R}^d \to X$ be locally Lipschitz between finite dimensional vector spaces and denote by $g_\theta(z)$ it's evaluation con coordinates $(z{,}\theta)$. We say that g satisfies *Assumption 1* For a certain probability distribution p over Z if there are local Lipschitz constants $L(\theta{,}z)$ such that:

$$\mathbb{E}_{z\sim p}[L(\theta, z)] + \infty.$$

Architectures - Experiments Discriminator

```
_________________________________________________________________
Model: "Discriminator"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 512)               1536
_________________________________________________________________
dense_5 (Dense)              (None, 512)               262656
_________________________________________________________________
dense_6 (Dense)              (None, 512)               262656
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 513
=================================================================
Total params: 527,361
Trainable params: 527,361
Non-trainable params: 0
```

Figure 34: Experiments discriminator

Architectures - Experiments Generator

```
Model: "Generator"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 512)               1536
_________________________________________________________________
dense_1 (Dense)              (None, 512)               262656
_________________________________________________________________
dense_2 (Dense)              (None, 512)               262656
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 1026
=================================================================
Total params: 527,874
Trainable params: 527,874
Non-trainable params: 0
```

Figure 35: Experiments Generator

Architectures - MNIST Critic

```
Model: "discriminator"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_7 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
zero_padding2d_3 (ZeroPaddin (None, 32, 32, 1)         0
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 16, 16, 64)        1664
_________________________________________________________________
leaky_re_lu_21 (LeakyReLU)   (None, 16, 16, 64)        0
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 8, 8, 128)         204928
_________________________________________________________________
leaky_re_lu_22 (LeakyReLU)   (None, 8, 8, 128)         0
_________________________________________________________________
dropout_9 (Dropout)          (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 4, 4, 256)         819456
_________________________________________________________________
leaky_re_lu_23 (LeakyReLU)   (None, 4, 4, 256)         0
_________________________________________________________________
dropout_10 (Dropout)         (None, 4, 4, 256)         0
_________________________________________________________________
conv2d_24 (Conv2D)           (None, 2, 2, 512)         3277312
_________________________________________________________________
leaky_re_lu_24 (LeakyReLU)   (None, 2, 2, 512)         0
_________________________________________________________________
flatten_3 (Flatten)          (None, 2048)              0
_________________________________________________________________
dropout_11 (Dropout)         (None, 2048)              0
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 2049
=================================================================
Total params: 4,305,409
Trainable params: 4,305,409
Non-trainable params: 0
_________________________________________________________________
```

Figure 36: MNIST Critic

Architectures - MNIST Generator

```
Model: "generator"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_8 (InputLayer)         [(None, 128)]             0
_________________________________________________________________
dense_7 (Dense)              (None, 4096)              524288
_________________________________________________________________
batch_normalization_12 (Batc (None, 4096)              16384
_________________________________________________________________
leaky_re_lu_25 (LeakyReLU)   (None, 4096)              0
_________________________________________________________________
reshape_3 (Reshape)          (None, 4, 4, 256)         0
_________________________________________________________________
up_sampling2d_9 (UpSampling2 (None, 8, 8, 256)         0
_________________________________________________________________
conv2d_25 (Conv2D)           (None, 8, 8, 128)         294912
_________________________________________________________________
batch_normalization_13 (Batc (None, 8, 8, 128)         512
_________________________________________________________________
leaky_re_lu_26 (LeakyReLU)   (None, 8, 8, 128)         0
_________________________________________________________________
up_sampling2d_10 (UpSampling (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_26 (Conv2D)           (None, 16, 16, 64)        73728
_________________________________________________________________
batch_normalization_14 (Batc (None, 16, 16, 64)        256
_________________________________________________________________
leaky_re_lu_27 (LeakyReLU)   (None, 16, 16, 64)        0
_________________________________________________________________
up_sampling2d_11 (UpSampling (None, 32, 32, 64)        0
_________________________________________________________________
conv2d_27 (Conv2D)           (None, 32, 32, 1)         576
_________________________________________________________________
batch_normalization_15 (Batc (None, 32, 32, 1)         4
_________________________________________________________________
activation_3 (Activation)    (None, 32, 32, 1)         0
_________________________________________________________________
cropping2d_3 (Cropping2D)    (None, 28, 28, 1)         0
=================================================================
Total params: 910,660
Trainable params: 902,082
Non-trainable params: 8,578
```

Figure 37: MNIST Generator

Architectures - CARS Critic

```
Model: "Critic"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 64, 64, 3)]       0
_________________________________________________________________
zero_padding2d (ZeroPadding2 (None, 68, 68, 3)         0
_________________________________________________________________
conv2d (Conv2D)              (None, 34, 34, 64)        4864
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 34, 34, 64)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 17, 17, 128)       204928
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 17, 17, 128)       0
_________________________________________________________________
dropout (Dropout)            (None, 17, 17, 128)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 9, 9, 256)         819456
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 9, 9, 256)         0
_________________________________________________________________
dropout_1 (Dropout)          (None, 9, 9, 256)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 5, 5, 512)         3277312
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 5, 5, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 12800)             0
_________________________________________________________________
dropout_2 (Dropout)          (None, 12800)             0
_________________________________________________________________
dense (Dense)                (None, 1)                 12801
=================================================================
Total params: 4,319,361
Trainable params: 4,319,361
Non-trainable params: 0
_________________________________________________________________
```

Figure 38: CARS critic

Architectures - CARS Genator

```
Model: "generator"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 256)]             0
_________________________________________________________________
dense_1 (Dense)              (None, 4096)              1048576
_________________________________________________________________
batch_normalization (BatchNo (None, 4096)              16384
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 4096)              0
_________________________________________________________________
reshape (Reshape)            (None, 4, 4, 256)         0
_________________________________________________________________
up_sampling2d (UpSampling2D) (None, 8, 8, 256)         0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 128)         294912
_________________________________________________________________
batch_normalization_1 (Batch (None, 8, 8, 128)         512
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 8, 8, 128)         0
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 16, 16, 64)        73728
_________________________________________________________________
batch_normalization_2 (Batch (None, 16, 16, 64)        256
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 16, 16, 64)        0
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 32, 32, 64)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 32, 32, 32)        18432
_________________________________________________________________
batch_normalization_3 (Batch (None, 32, 32, 32)        128
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 32, 32, 32)        0
_________________________________________________________________
up_sampling2d_3 (UpSampling2 (None, 64, 64, 32)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 64, 64, 3)         864
_________________________________________________________________
batch_normalization_4 (Batch (None, 64, 64, 3)         12
_________________________________________________________________
activation (Activation)      (None, 64, 64, 3)         0
=================================================================
Total params: 1,453,804
Trainable params: 1,445,158
Non-trainable params: 8,646
_________________________________________________________________
```

Figure 39: CARS Generator

References i

M. Arjovsky and L. Bottou.
Towards principled methods for training generative adversarial networks, 2017.

M. Arjovsky, S. Chintala, and L. Bottou.
Wasserstein generative adversarial networks.
In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 214–223, International Convention Centre, Sydney, Australia, 08 2017. PMLR.

S. Arora, R. Ge, Y. Liang, T. Ma, and Y. Zhang.
Generalization and equilibrium in generative adversarial nets (GANs).
In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 224–232, International Convention Centre, Sydney, Australia, 8 2017. PMLR.

A. Brock, J. Donahue, and K. Simonyan.
Large scale gan training for high fidelity natural image synthesis, 2018.

A. Clark, J. Donahue, and K. Simonyan.
Adversarial video generation on complex datasets, 2019.

M. Frid-Adar, I. Diamant, E. Klang, M. Amitai, J. Goldberger, and H. Greenspan.
Gan-based synthetic medical image augmentation for increased cnn performance in liver lesion classification.
Neurocomputing, 321:321–331, Dec 2018.

I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio.
Generative adversarial networks, 2014.

I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville.
Improved training of wasserstein gans, 2017.

M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, G. Klambauer, and S. Hochreiter.
Gans trained by a two time-scale update rule converge to a nash equilibrium.
CoRR, abs/1706.08500, 2017.

J. Krause, M. Stark, J. Deng, and L. Fei-Fei.
3d object representations for fine-grained categorization.
In *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*, Sydney, Australia, 2013.

M. Mirza and S. Osindero.
Conditional generative adversarial nets, 2014.

T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida.
Spectral normalization for generative adversarial networks, 2018.

A. Radford, L. Metz, and S. Chintala.
Unsupervised representation learning with deep convolutional generative adversarial networks, 2015.

S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee.
Generative adversarial text to image synthesis, 2016.

T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, X. Chen, and X. Chen.
Improved techniques for training gans.
In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2234–2242. Curran Associates, Inc., 2016.

H. Thanh-Tung, T. Tran, and S. Venkatesh.
Improving generalization and stability of generative adversarial networks.
In *International Conference on Learning Representations*, 2019.

L. Weng.
From gan to wgan.
lilianweng.github.io/lil-log, 2017.

Y. Wu, J. Donahue, D. Balduzzi, K. Simonyan, and T. Lillicrap.
Logan: Latent optimisation for generative adversarial networks, 2019.