

Redes adversarias generativas

Índice

1. Motivación	2
2. La red GAN básica	3
2.1. Entrenamiento	3
2.2. Implementación	5
3. Redes GAN mejoradas	6
3.1. Generador atrofiado – PacGAN	6
3.2. Distribuciones disjuntas – WGAN	7
4. La red GAN condicional	9
5. La red GAN cíclica	10
6. Ejercicios prácticos	12

Referencias

- ☒ I. Goodfellow et al., *Generative Adversarial Nets*, NeurIPS, 2014.
- ☒ T. Salimans et al., *Improved Techniques for Training GANs*, NeurIPS, 2016.
- ☒ Z. Lin et al., *PacGAN: The power of two samples in GANs*, NeurIPS, 2018.
- ☒ M. Arjovsky, et al., *Wasserstein generative adversarial networks*, ICML, 2017.
- ☒ P. Isola, et al., *Image-To-Image Translation With Conditional Adversarial Networks*, CVPR, 2017.
- ☒ J-Y. Zhu, et al., *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, ICCV, 2017.

(Todas las imágenes son originales o ©, salvo que se indique lo contrario.)

Tiempo estimado = 1 sesión de 2 horas

1. Motivación

Cuando estudiamos los autoencoders variacionales aprendimos que la red *Encoder* producía la media y covarianza de una distribución MVN para modelar, de manera probabilística, la representación latente de la entrada. Después la red *Decoder*, dado que se trata a fin de cuentas de una transformación determinista, convierte la MVN en otra distribución de probabilidad conjunta que modela la salida. Pero como la salida de un autoencoder es la entrada, lo que en definitiva estamos obteniendo es la distribución de probabilidad generativa de los ejemplos dados.

Esta no es la única manera de obtener un modelo generativo de los ejemplos. Otra técnica muy popular y que, en la actualidad, es un tema de investigación muy productivo, es las redes generativas adversarias (*Generative Adversarial Networks*, GAN). En GitHub se puede encontrar una lista de “todas” las GAN presentadas hasta aproximadamente hace un año, recopiladas en el GAN Zoo [🔗](#). A modo de ejemplo, la Figura 1 muestra 6 caras de personas que no existen, de calidad fotorealista (M. Marchesi *et al.*, “*Megapixel Size Image Creation using Generative Adversarial Networks*”, DeepAI, 2017) [🔗](#).



Figura 1: Caras fotorealistas generadas mediante GAN (M. Marchesi *et al.*, 2017) [🔗](#).

Las GAN se parecen a los VAE en que una red neuronal, denominada *Generador* transforma muestras aleatorias de una distribución, por ejemplo la uniforme, en muestras aleatorias de otra distribución, tal y como hace el *Decoder*. Todo lo demás es diferente:

- La arquitectura de la red es más bien al contrario, primero se generan muestras aleatorias y después se utiliza un encoder para aprender a diferenciarlas de las verdaderas
- El aprendizaje no se realiza en una única pasada de *back-propagation* sobre la red.
- La función de pérdida no es ni el MSE entre las imágenes reales y las imágenes obtenidas a partir de transformar las muestras aleatorias.

El mecanismo de las GAN consiste en dos redes con intereses contrapuestos, por eso se denominan *adversarias*. La red *Generadora* se entrena para producir datos (típicamente imágenes) falsos pero con tal grado de verosimilitud que sean capaces de confundir a una segunda red *Discriminadora*. Esta segunda red, a su vez, se entrena para distinguir los datos verdaderos de los falsos, generados la otra red.

Esta idea inicial fue presentada en por primera vez en la conferencia NeurIPS de 2014 por Goodfellow *et al.*. Desde entonces, esta técnica ha generado nuevos problemas y soluciones. Entre los más interesantes se encuentran los siguientes:

- Se observa que la GAN termina por generar datos sólo de unos pocos tipos, no de todos los que, en teoría, podría generar.
→ *Mode collapse*
- ¿Se le puede pedir a la GAN que genere datos de un tipo concreto?
→ *Conditional GAN* (CGAN)
- ¿Podemos entrenar una GAN para hacer el camino inverso, es decir obtener la representación latente a partir de la imagen?
→ *Cycle GAN*

2. La red GAN básica

Una red GAN se compone de dos redes neuronales con propósitos contrarios: la red generativa \mathcal{G} y la red discriminativa \mathcal{D} , en la arquitectura que se muestra en la Figura 2.

La red **generativa** \mathcal{G} recibe un vector d -dimensional z muestreado de una distribución de probabilidad $\mathbf{P}(z)$, que habitualmente es la Uniforme($[0, 1]^d$). Este vector z hace las veces de una representación en el espacio latente de los datos reales, x . La red generativa transforma z en $\tilde{x} = \mathcal{G}(z)$, que habitualmente se conoce como datos “falsos” (*fake*). Este proceso está representado en el recuadro 1 de la Figura 2.

El objetivo es lograr que la red generativa produzca datos falsos que puedan “hacerse pasar por verdaderos”. En una primera aproximación podríamos pensar en minimizar la diferencia entre \tilde{x} y x . Pero de este modo lo que lograríamos es aprender a reproducir los datos verdaderos, no a producir nuevas muestras de los datos.

La solución que proporcionan las GAN consiste en crear un nuevo conjunto de datos juntando muestras verdaderas, x , con muestras falsas, \tilde{x} . De este modo logramos recopilar un conjunto etiquetado de datos de un mismo tipo ya que sabemos cuales son verdaderos ($y = 1$) y cuales falsos ($y = 0$). Este proceso es el recuadro 2 de la Figura 2.

La red **discriminativa** \mathcal{D} recibe este segundo conjunto de datos y se entrena para producir la probabilidad de que el ejemplo introducido sea verdadero. Para ello la capa de salida de \mathcal{D} consta de una única neurona con activación sigmoide. En otras palabras, si $x^{(i)}$ representa el i -ésimo ejemplo de la tabla del recuadro 2, entonces la salida de la red discriminativa es

$$\hat{y}^{(i)} = \mathcal{D}(x^{(i)}) = p(y = 1|x^{(i)}) = 1 - p(y = 0|x^{(i)}).$$

Este paso es el recuadro 3 de la Figura 2.

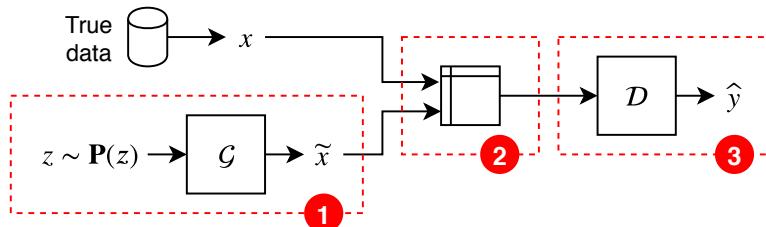


Figura 2: Arquitectura de la red GAN básica (*Vanilla GAN*)

2.1. Entrenamiento

El proceso de entrenamiento tiene una primera etapa hacia delante que sigue los pasos detallados en la Figura 2. Una vez se produce la salida \hat{y} , comienza la segunda etapa: el ajuste de los pesos de las redes \mathcal{D} y \mathcal{G} . Para ello necesitamos una función de pérdida que será el motor del algoritmo de retro-propagación (*back-propagation*). Pero primero hay que formalizar el objetivo de cada una de las redes.

Por un lado, el objetivo de la red \mathcal{D} es maximizar el valor de su salida \hat{y} cuando la entrada es un dato real x , y al mismo tiempo minimizar el valor de \hat{y} cuando la entrada es un dato falso; pero esto es equivalente a maximizar el valor de $1 - \hat{y}$. Por el otro lado, el objetivo de la red \mathcal{G} es lograr que la \mathcal{D} produzca salidas lo mayores posibles para los ejemplos falsos generados por la red \mathcal{G} . En definitiva las redes \mathcal{G} y \mathcal{D} se deben entrenar con objetivos opuestos.

En la Figura 3(a) se muestra cuál es el efecto que queremos lograr con cada red sobre la salida $\hat{y} = \mathcal{D}(\cdot)$, cuando el ejemplo que evalúa la red \mathcal{D} es real o falso, x ó $\tilde{x} = \mathcal{G}(z)$ respectivamente. Como se puede ver, cada red intenta hacer lo contrario de la otra.

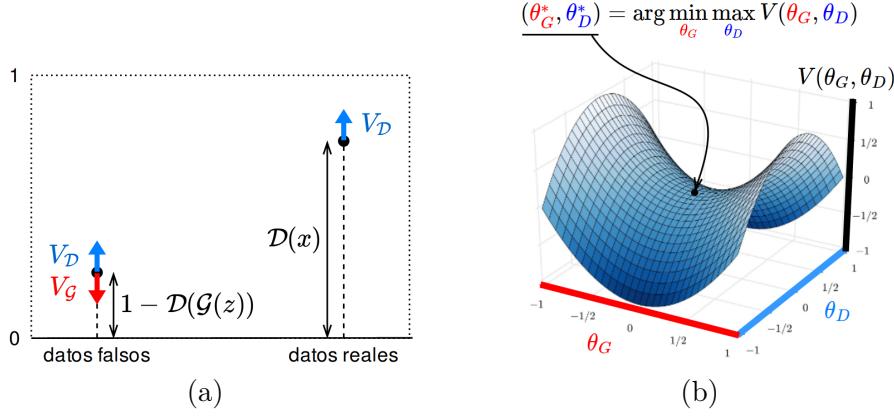


Figura 3: Juego Min-Max que se plantea para el entrenamiento de las GAN

En otras palabras, este tipo de redes plantean un juego Min-Max, en el que buscamos el conjunto óptimo de parámetros que satisfacen una minimización y una maximización al mismo tiempo de una función objetivo V donde las variables que podemos modificar son los pesos de la red \mathcal{G} y \mathcal{D} , θ_G y θ_D respectivamente, representado a la Figura 3(b).

Formulación del juego Min-Max Desde su presentación en 2014, el problema de optimización que plantean las GAN se formula del siguiente modo:

$$V(\theta_D, \theta_G) = \min_{\theta_G} \max_{\theta_D} \mathbb{E}_{x \sim \mathbf{P}(x)} [\log \mathcal{D}(x)] + \mathbb{E}_{z \sim \mathbf{P}(z)} [1 - \log \mathcal{D}(\mathcal{G}(z))] \quad (1)$$

Para comprender como se llega a esta expresión, suponemos que la tabla resultante en el paso 2 contiene m datos, de tal manera que hay m valores reales y m valores falsos.

Ya hemos visto que la red \mathcal{D} quiere maximizar \hat{y} cuando recibe un ejemplo real x y también quiere maximizar $1 - \hat{y}$ cuando recibe un ejemplo falso $\tilde{x} = \mathcal{G}(z)$. Pero como recibe m ejemplos iid de cada tipo a lo largo del proceso de entrenamiento, y como \hat{y} es una probabilidad, entonces el objetivo maximizar la probabilidad de los datos, es decir,

$$\begin{aligned} & \max [p(x^{(1)}) \cdot p(x^{(2)}) \cdots p(x^{(m)}) \cdot (1 - p(\tilde{x}^{(1)})) \cdot (1 - p(\tilde{x}^{(2)})) \cdots (1 - p(\tilde{x}^{(m)}))] \\ &= \max [\mathcal{D}(x^{(1)}) \cdot \mathcal{D}(x^{(2)}) \cdots \mathcal{D}(x^{(m)}) \cdot (1 - \mathcal{D}(\tilde{x}^{(1)})) \cdot (1 - \mathcal{D}(\tilde{x}^{(2)})) \cdots (1 - \mathcal{D}(\tilde{x}^{(m)}))] \\ &= \max \left[\prod_{i=1}^m \mathcal{D}(x^{(i)}) (1 - \mathcal{D}(\tilde{x}^{(i)})) \right] = \max \left[\sum_{i=1}^m \log \mathcal{D}(x^{(i)}) + \sum_{i=1}^m \log (1 - \mathcal{D}(\tilde{x}^{(i)})) \right]. \end{aligned}$$

Si además dividimos por el número de ejemplos de cada tipo, entonces llegamos a que el objetivo de \mathcal{D} es maximizar el promedio de la log-verosimilitud de las predicciones:

$$\begin{aligned} & \max_{\theta_D} \left[\frac{1}{m} \sum_{i=1}^m \log \mathcal{D}(\theta_D; x^{(i)}) + \frac{1}{m} \sum_{i=1}^m \log (1 - \mathcal{D}(\theta_D; \mathcal{G}(z^{(i)}))) \right] = \\ & \max_{\theta_D} [\mathbb{E}_{x \sim \mathbf{P}_x} \log \mathcal{D}(\theta_D; x) + \mathbb{E}_{z \sim \mathbf{P}_z} \log (1 - \mathcal{D}(\theta_D; \mathcal{G}(z)))] . \end{aligned} \quad (2)$$

En (2) se muestra de manera explícita θ_D , el conjunto de pesos que podemos ajustar, cuando entrenemos la red \mathcal{D} , y también se ha sustituido $\tilde{x} = \mathcal{G}(z)$.

Por otro lado, cuando entrenamos la red \mathcal{G} sólo introducimos datos falsos, y queremos ajustar sus pesos de la red θ_G para lograr que la red \mathcal{D} se confunda y los etiquete como verdaderos. En otras palabras queremos minimizar la probabilidad de que los etiquete como falsos (que es lo que realmente son), formalmente:

$$\min \left[(1 - p(\tilde{x}^{(1)})) \cdot (1 - p(\tilde{x}^{(2)})) \cdots (1 - p(\tilde{x}^{(m)})) \right] = \min \left[\sum_{i=1}^m \log (1 - \mathcal{D}(\tilde{x}^{(i)})) \right].$$

Igual que antes, si dividimos por el número de ejemplos llegamos a que el objetivo de \mathcal{D} se puede expresar en términos del promedio de un conjunto de experimentos:

$$\min_{\theta_G} \left[\mathbb{E}_{z \sim \mathbf{P}_z} \log (1 - \mathcal{D}(\theta_D; \mathcal{G}(z))) \right]. \quad (3)$$

Y combinando (2) y (3) llegamos a (1).

2.2. Implementación

En el artículo de Goodfellow et al. se proponía el algoritmo de aprendizaje de la Figura 4.

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

```

for number of training iterations do
  for  $k$  steps do
    • Sample minibatch of  $m$  noise samples  $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$  from noise prior  $p_g(\mathbf{z})$ .
    • Sample minibatch of  $m$  examples  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  from data generating distribution  $p_{\text{data}}(\mathbf{x})$ .
    • Update the discriminator by ascending its stochastic gradient:
      
$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[ \log D(\mathbf{x}^{(i)}) + \log (1 - D(G(\mathbf{z}^{(i)}))) \right].$$

  end for
  • Sample minibatch of  $m$  noise samples  $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$  from noise prior  $p_g(\mathbf{z})$ .
  • Update the generator by descending its stochastic gradient:
    
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(\mathbf{z}^{(i)}))).$$

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

```

Figura 4: Algoritmo de entrenamiento propuesto en el artículo de I. Goodfellow et al., *Generative Adversarial Nets*, NeurIPS, 2014.

Como se puede apreciar, las expresiones del valor esperado se aproximan por el promedio, tal y como habíamos obtenido en la sección anterior. También se aprecia que, en el entrenamiento de la red \mathcal{D} se requiere un ascenso de gradiente mientras que en el de la red \mathcal{G} es un descenso. Este algoritmo es genérico, independiente del lenguaje de programación, pero ¿cómo llevarlo a cabo en un lenguaje modularizado como Python/Tensorflow/Keras?.

Una solución posible consiste en pensar que todos los parámetros de la red, tanto θ_G como θ_D , se deben actualizar a partir de una perdida que depende únicamente de la salida de la red \mathcal{D} . Por otro lado dicha salida se interpreta como la probabilidad de un suceso binario, dato real vs. dato falso. De esta manera:

1. Entrenamos la red \mathcal{D} con la tabla recopilada con ejemplos de ambos tipos utilizando la entropía cruzada binaria como función de pérdida. Es decir

$$\text{Loss}_D = \text{binary_cross_entropy}(y, \hat{y}; x, \tilde{x}).$$

2. Entrenamos la red completa, es decir la red \mathcal{G} seguida de la red \mathcal{D} utilizando sólo los ejemplos falsos pero etiquetados como verdaderos y de nuevo se utilizaría la entropía cruzada binaria,

$$\text{Loss}_G = \text{binary_cross_entropy}(1, \hat{y}; \tilde{x}).$$

En este paso es esencial indicar que NO se deben actualizar los pesos de la red \mathcal{D} . Ésta simplemente actúa como un bloque derivable, que transmite el gradiente hacia \mathcal{G} .

3. Redes GAN mejoradas

3.1. Generador atrofiado – PacGAN

Cuando entrenamos la red \mathcal{G} los parámetros θ_D están fijos. En el caso límite, un sobre-entrenamiento podría dar lugar a una red \mathcal{G} que generara datos óptimos para que \mathcal{D} no los detecte como falsos. Si la distribución de probabilidad que estamos intentando aproximar con \mathcal{G} es multimodal (que es lo más probable), entonces lo más seguro es que \mathcal{G} se especialice en unos pocos modos, los que proporcionan esos datos óptimos con los que siempre engaña a \mathcal{D} . En YouTube se puede ver como una GAN entrenada con MNIST acaba generando únicamente un ejemplos de “1”. \square .

A este problema se le denomina *mode collapse*, y se han propuesto diversas soluciones. Entre estas, Lin et al. presentaron en el NeurIPS'18 una solución sencilla de implementar pero con sólidos argumentos teóricos que la respaldan denominada **PacGAN**. En su artículo *PacGAN: The power of two samples in generative adversarial networks* \square proponen una definición matemática para medir cuán severo es el *mode collapse* dadas dos distribuciones \mathbf{P} y \mathbf{Q} . De esta manera se pueden comparar dos generadores según su robustez a este problema.

La idea principal de su arquitectura consiste en NO restringir la entrada de la red \mathcal{D} a una única imagen. Esto no quiere decir que no se la pueda alimentar con un mini-lote como habitualmente se hace en el entrenamiento, sino que \mathcal{D} tiene las neuronas de entrada justas para admitir una sola imagen. Por ejemplo si ésta tiene resolución 100×100 entonces \mathcal{D} tendrá 10K neuronas de entrada; pero ¿por qué? ¿Y si utilizamos 20K y admitimos dos o más?. Esta solución se muestra esquemáticamente en la Figura 5.

A parte de los resultados teóricos, los experimentos muestran una generación eficiente de todos los modos en problemas benchmark como el que se ve en la Figura 6.

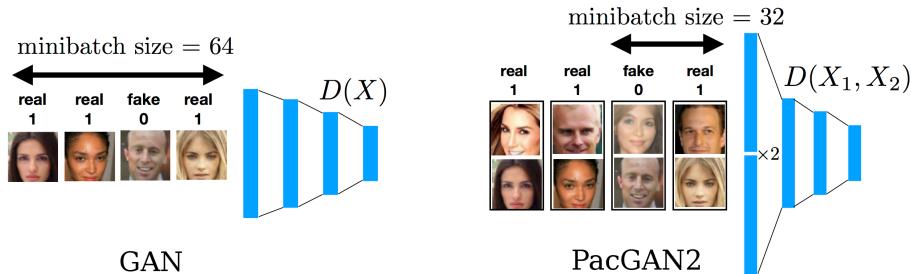


Figura 5: Comparación entre la arquitectura de la red $netD$ en una GAN básica (izquierda) y la de una PacGAN2 (derecha) (Imagen tomada de Z. Lin et al., *PacGAN: The power of two samples in GANs*, NeurIPS, 2018. \square)

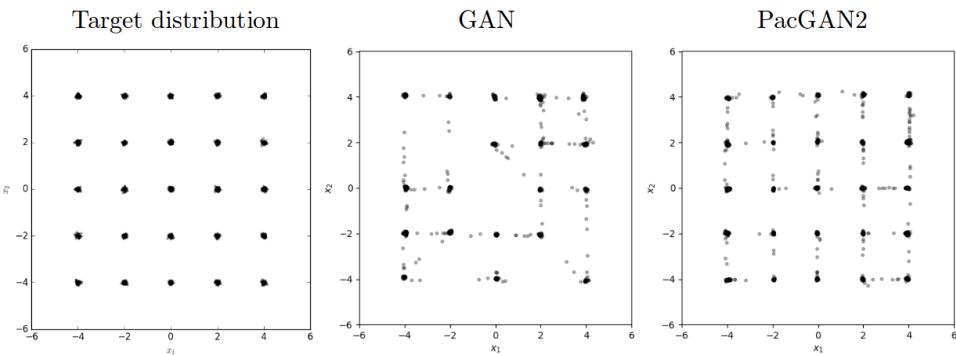


Figura 6: (Izq.) Benchmark empleado para comprobar el problema de *mode collapse*, (Cnt.) Distribución generada por una GAN básica, (Der.) Distribución generada con la solución PacGAN. El 2 hace referencia al número de muestras utilizadas al mismo tiempo. (Imagen tomada de Z. Lin et al., *PacGAN: The power of two samples in GANs*, NeurIPS, 2018. \square)

3.2. Distribuciones disjuntas – WGAN

Si denominamos \mathbf{P}_x y $\mathbf{P}_{\tilde{x}}$ a la distribución de los datos reales y los falsos respectivamente, la red \mathcal{G} será mejor cuanto más parecidas sean la distribución de los datos falsos que ésta genera a la distribución de los datos reales. Es decir, cuanto más se aproxime $\mathbf{P}_{\tilde{x}} = \mathbf{P}_{\mathcal{G}(x)}$ a \mathbf{P}_x .

Al implementar la GAN básica se ha utilizado la entropía cruzada. Pero ésta es sólo una de tantas métricas que se pueden utilizar para comparar distribuciones de probabilidad. Otras muy populares son la divergencia KL, la divergencia JS. Sin embargo todas ellas sufren de un efecto muy adverso cuando los soportes de ambas distribuciones son conjuntos disjuntos. En concreto, si dos distribuciones tienen soportes disjuntos, entonces la divergencia KL resulta ∞ , mientras que la divergencia JS tiene un salto no diferenciable.

Recordar que el soporte de una distribución es el intervalo en el que la distribución toma valores no nulos. Por ejemplo, en la Figura 7 se ven muestras de dos distribuciones cuyos soportes (los intervalos resaltados en los planos XY , YZ y XZ) son disjuntos.

Cuando los datos son imágenes, las muestras de \mathbf{P}_x y de $\mathbf{P}_{\tilde{x}}$ viven en un espacio de muy alta dimensionalidad. Una pequeña imagen en escala de grises (con intensidad escalada al intervalo unidad), de resolución 100×100 , sería una punto del espacio $[0, 1]^{10000}$. Sin embargo, como sabemos, no todos los píxeles aportan la misma información. De hecho seguramente la dimensión de las características relevantes es mucho menor. En otras palabras, las características importantes ocupan subespacios de mucha menor dimensionalidad. Como el espacio de los datos es tan grande, resulta muy fácil que las distribuciones de las características importantes y la distribución de datos falsos tenga soportes disjuntos, especialmente cuando la red \mathcal{G} está empezando a entrenarse. Podemos visualizar esto si imaginamos el soporte de \mathbf{P}_x y el de $\mathbf{P}_{\tilde{x}}$ como sendos palos de un cierto tamaño, ambos situados en un volumen como una habitación. La probabilidad de que ambos palos se toquen es muy pequeña.

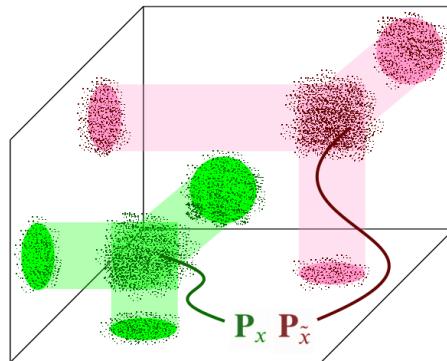


Figura 7: Dos PDF con soportes disjuntos

Una alternativa para resolver este problema es la distancia Wasserstein. Si vemos cada distribución \mathbf{P}_x y $\mathbf{P}_{\tilde{x}}$ como dos montones de arena, la distancia Wasserstein es el coste que supone transformar un montón en el otro. Este coste se define como la cantidad de masa movida multiplicada por la distancia media de lo que debemos transportar de un montón a otro. Evidentemente, podemos imaginar muchas formas de convertir un montón en otro; incurriendo en un coste con cada una de ellas. Como es lógico, de todos ellos, nos interesa el menor.

Pero esta definición es intratable porque requiere probar todas las posibilidades para asegurar que hemos encontrado la de menor coste. En el ICML de 2017 Arjovsky et al. presentaron una solución basada en una formulación dual, fundamentada en una carga teórica-matemática un tanto compleja pero que se implementa con pequeños cambios respecto de la formulación básica.

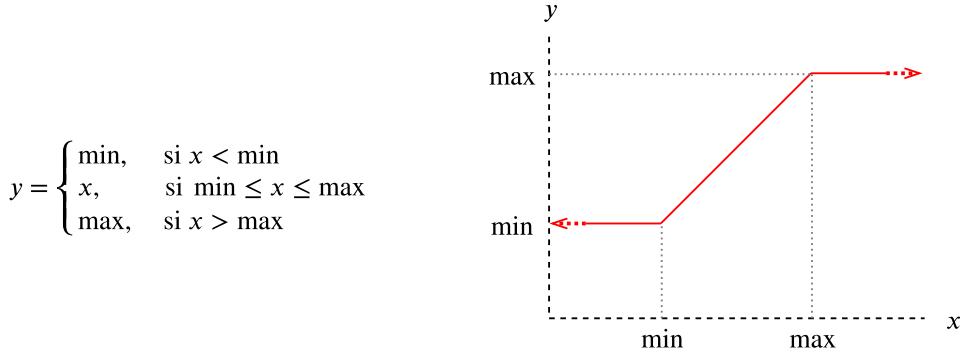


Figura 8: La operación de limitación (*clip* en Tensorflow o *clamp* en PyTorch)

En primer lugar cambia el nombre de la red \mathcal{D} , que pasa a ser denominada “Crítico”, \mathcal{C} ; en el sentido de un crítico de arte o de cine que evalúa lo bien que dibuja un artista o actúa el protagonista. En este caso \mathcal{C} evalúa el grado de realismo (o el grado de falsedad) del dato que recibe. En concreto, cuanto menor sea la pérdida de \mathcal{C} cuando evalúa imágenes falsas, mayor será la calidad que podemos esperar de éstas. Puesto que ya no interpretamos la salida como una probabilidad, no tiene sentido utilizar la activación sigmoide. Es preferible usar una activación lineal que nos permita cualquier rango de valores.

En segundo lugar, se utiliza una pérdida que aproxima razonablemente la distancia Wasserstein para entrenar tanto \mathcal{G} como \mathcal{C} . Dicha pérdida se define

$$W(\mathbf{P}_x, \mathbf{P}_{\tilde{x}}) = \max_{w \in W} \mathbb{E}_{x \sim \mathbf{P}_x} [f_w(x)] - \mathbb{E}_{\tilde{x} \sim \mathbf{P}_{\tilde{x}}} [f_w(g_\theta(z))], \quad (4)$$

donde f_w es la función que se aprende en la red \mathcal{C} y g_θ es la de \mathcal{G} . En ambas el subíndice representa los pesos. Esta pérdida se utiliza para ambas redes, que también se entranan de manera consecutiva.

- Para \mathcal{C} , la pérdida es el promedio de las valoraciones que hace el crítico de datos reales menos el promedio de las valoraciones de datos falsos. De esta manera, una mayor puntuación de \mathcal{C} para los datos reales conlleva una mayor pérdida para \mathcal{C} . Esto le empuja a reducir la puntuación para datos reales. Por ejemplo, una puntuación media de 20 para imágenes reales y 50 para imágenes falsas da como resultado una pérdida de -30 ; una puntuación media de 10 para imágenes reales y 50 para imágenes falsas da como resultado una pérdida de -40 , que es mejor, y así sucesivamente.
- Para \mathcal{G} , la pérdida sólo tiene en cuenta los datos falsos, y por tanto desaparece el término de la izquierda. Así, a mayor puntuación de \mathcal{C} menor pérdida para \mathcal{G} , lo cual mueve a \mathcal{C} a aumentar la puntuación de los datos falsos. Por ejemplo, una puntuación media de 10 se convierte en -10 , una de 50 se convierte en -50 , que es menor, y así sucesivamente.

Dicho promedio se calcula sobre el mini-lote que se recibe en cada iteración del entrenamiento. En definitiva, la pérdida para datos reales debe ser un número pequeño, y para datos falsos uno grande. Además se deben tener en cuenta las siguientes recomendaciones.

- Limitar los pesos de la red \mathcal{C} a un pequeño intervalo, por ejemplo $[-0.01, 0.01]$. Esto se puede lograr en Keras/TF con la función `clip`, que se muestra en la Figura 8.
- Actualizar los pesos de \mathcal{C} más frecuentemente que los pesos de \mathcal{G} ; por ejemplo 1 actualización de \mathcal{G} por cada 5 de \mathcal{C} .
- Utilizar el RMSprop sin *momentum* y con un ratio de aprendizaje (*learning rate*) pequeño, por ejemplo en el orden de magnitud de 10^{-5} .

El algoritmo de entrenamiento propuesto originalmente se muestra en la Figura 9.

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

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

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

```

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

Figura 9: Algoritmo WGAN propuesto en el artículo de

Para su implementación en Keras conviene pensar que los datos reales obtienen una puntuación negativa (pequeña) y los falsos una positiva (grande). De este modo, podemos obtener la pérdida de los datos falsos multiplicando por -1 el promedio de las valoraciones de estos datos falsos; y del mismo modo multiplicando por $+1$ las valoraciones de los reales. Esto se logra asignando la etiqueta -1 a los datos falsos y $+1$ a los reales y definiendo la pérdida Wasserstein como

```

from keras import backend
def wasserstein_loss(y_true, y_pred):
    return backend.mean(y_true * y_pred)
```

4. La red GAN condicional

Las redes GAN vistas hasta este momento generan datos falsos pero no de una categoría concreta a demanda. Por ejemplo si entrenamos una GAN con el MNIST, nos podría interesar pedirle al sistema que genere nuestro código postal , pero no hay ningún *mecanismo* para pedirle una cifra concreta.

Las GAN condicionales (CGAN) son una solución a este problema que sólo requiere de pequeños cambios en la arquitectura, resaltados en las líneas de colores de la Figura 10.

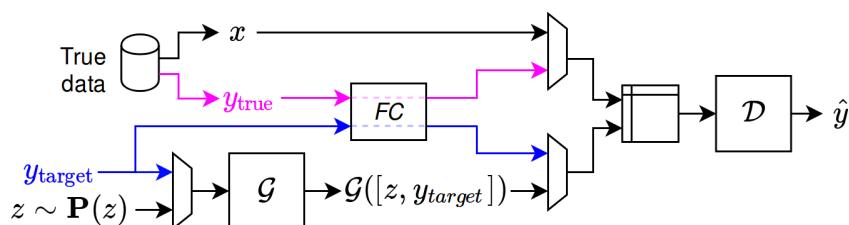


Figura 10: Arquitectura de la red (*Conditional GAN*, CGAN). Los trapecios indican la concatenación de los tensores de entrada.

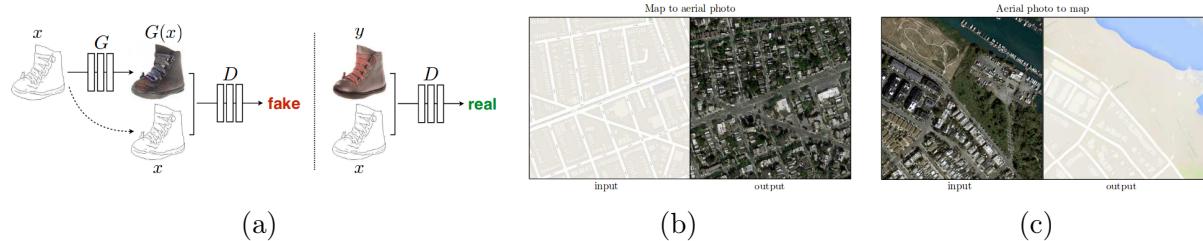


Figura 11: Traducción (a) de contorno a imagen (b) de mapa a imagen (c) de imagen a mapa. Imágenes tomadas de P. Isola, et al., *Image-To-Image Translation With Conditional Adversarial Networks*, CVPR, 2017 ↗

Tanto la red \mathcal{G} como la red \mathcal{D} (o \mathcal{C} si se trata de la versión W-GAN) reciben como información adicional la categoría objetivo (y_{target}), es decir la categoría de la cual queremos generar un dato, para condicionar el resultado. Al añadir y_{target} al vector aleatorio z , el primer efecto inmediato es que la entrada de la red \mathcal{G} ya NO es completamente aleatoria. Por otro lado, como la red \mathcal{D} ya no sólo recibe una tabla compuesta por datos reales y falsos. Ahora, a dicha tabla, se le añade información sobre la categoría del dato real que estamos recibiendo y sobre la categoría del dato falso. En la Figura 10 se utiliza una red densamente conexa que transforma la categoría en un tensor de tamaño similar a los datos reales/falsos para poderla concatenarlos. Esto puede ser útil en el caso de imágenes aunque podría eliminarse o no necesariamente producir tensores del mismo tamaño en el caso de datos tabulados. De este modo la red \mathcal{D} no sólo evalúa la similitud entre datos reales y falsos; también la correspondencia entre la imagen falsa y la categoría solicitada.

En definitiva, hemos logrado condicionar el resultado del generador con el valor de y_{target} . Pero esto es extensible a otras características (si se trata de datos tabulados) o a partes, elementos o características visuales si se trata de imágenes; por ejemplo si queremos generar caras podríamos especificar si son de hombre o mujer, el color de los ojos, el del pelo, etc. Basta con sustituir y_{target} por x_{target} , un vector (puede ser de dimensión 1) de características que queremos que condicionen el resultado.

Una de las aplicaciones más interesantes de las CGAN es la *traducción* entre imágenes. De igual modo que con dos idiomas podemos crear pares de palabras (una por cada idioma) que se corresponden, podemos pensar en una aplicación que empareje imágenes. Así, en la Figura 11 se muestran 3 ejemplos de traducción de una imagen de bordes a una imagen real o viceversa.

5. La red GAN cíclica

En la sección anterior hemos visto la *traducción* de imágenes mediante la utilización de CGAN y pares de imágenes para entrenamiento. ¿Sería posible hacer lo mismo teniendo únicamente dos conjuntos de imágenes, cada uno de un dominio distinto, pero NO emparejadas?. En este caso, lo que lograríamos es traducir el *estilo*. Es decir, ya que una imagen α_i de un dominio A puede no tener su correspondiente β_i en el dominio B , lo único que podemos lograr es que el aspecto de α_i sea similar al aspecto general del dominio b , o sea que se parezca un poco a todas las β_i . Por este motivo ha recibido el nombre de “transferencia de estilo” (*Style transfer*) En la Figura 12 se representan ambos problemas; a la izquierda el de traducción de imágenes y a la derecha el de transferencia de estilo; y en la Figura 13.

En el ICCV de 2017 Zhu, et al. presentaron las GAN cíclicas (*Cycle GAN*) para llevar a cabo la transferencia de estilo.

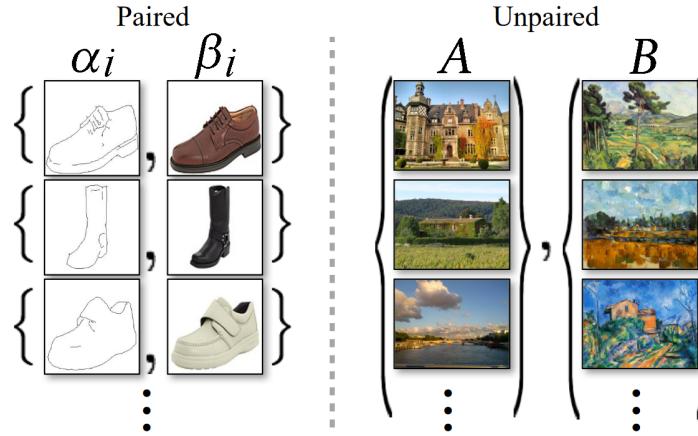


Figura 12: Dos conjuntos de imágenes emparejadas (Izq.) y desemparejadas (Der.) Imagen tomada de J-Y. Zhu, et al., *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, ICCV, 2017. \square

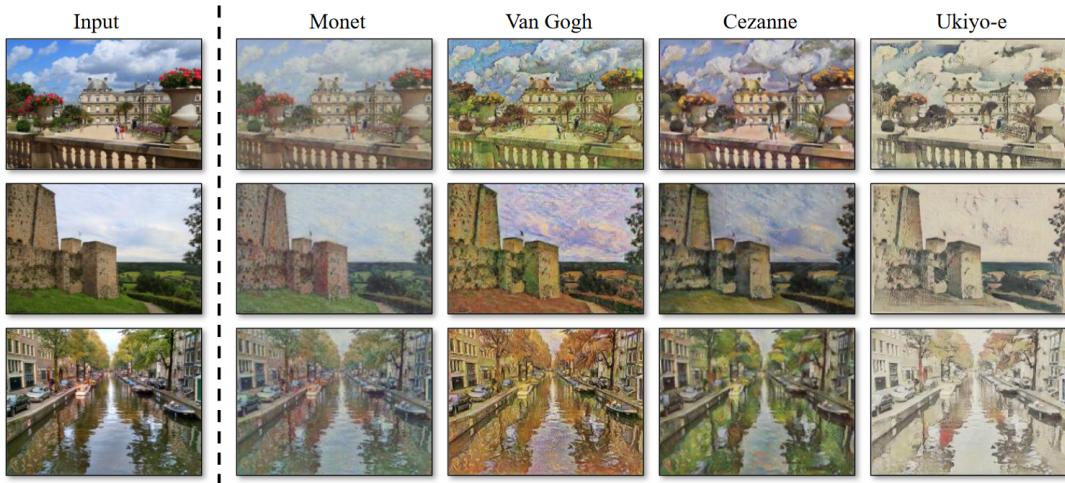


Figura 13: Resultados de una GAN cíclica donde un dominio son fotografías y el otro son cuadros impresionistas. Imagen tomada de J-Y. Zhu, et al., *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, ICCV, 2017. \square

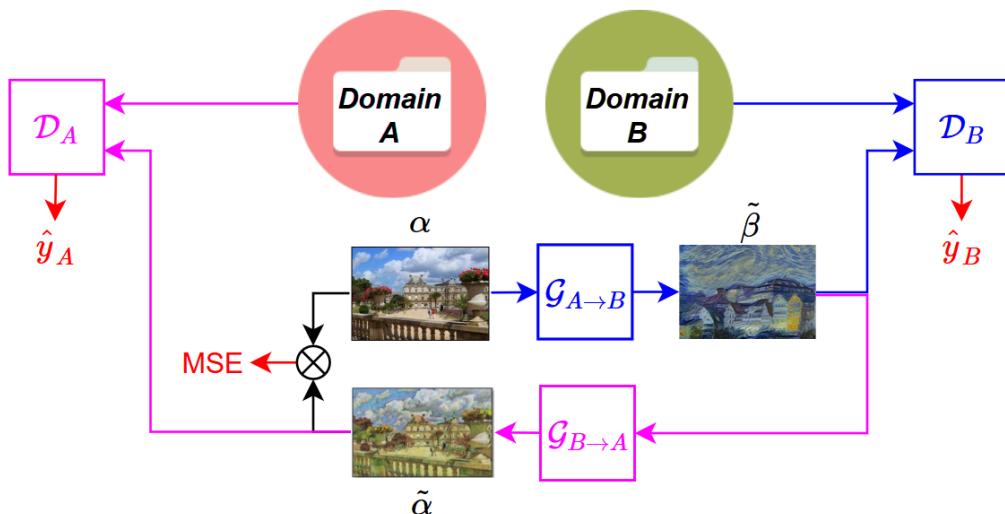


Figura 14: Arquitectura de la GAN cíclica

Una GAN cíclica consiste realmente en dos CGAN cuyo mecanismo se muestra en la Figura 14.

En la primera el generador $\mathcal{G}_{A \rightarrow B}$ produce una imagen falsa $\tilde{\beta}$ del dominio B condicionada por una imagen α del dominio A . Después su discriminador \mathcal{D}_B valora si el aspecto de $\tilde{\beta}_i$ es similar a las imágenes almacenadas en B , pero NO que su contenido sea similar a la imagen original α_i . Esto se puede apreciar en la Figura 14 en el hecho de que $\tilde{\beta}$ tiene un aspecto similar a los cuadros de Van Gogh (Dominio B) pero no un aspecto similar al de la imagen α .

Para lograr este objetivo, una segunda CGAN produce $\tilde{\alpha}$, una imagen falsa del dominio A utilizando el generador $\mathcal{G}_{B \rightarrow A}$ condicionado por la imagen $\tilde{\beta}$. Después $\tilde{\alpha}$ es evaluada por el discriminador \mathcal{D}_A para asegurar su pertenencia al dominio A y también es comparada con la imagen real α para asegurar que su aspecto es similar al original.

6. Ejercicios prácticos

- Trata de construir una red CGAN y aplícala a traducción de imágenes en alguna de las siguientes aplicaciones:
 - **Segmentación.** Necesitarás un conjunto de imágenes y sus correspondientes segmentadas para aprender a traducir de imagen a imagen segmentada.
 - **Conversión de tipografía.** Puedes hacer una aplicación que aprenda a convertir las letras de una tipografía en otra, sin necesidad de pasar por un OCR. Necesitas los caracteres emparejados de cada tipografía.
 - **Generación de comics.** Si dispones de bocetos y de resultados finales de por ejemplo personajes de cómic o de videojuegos, puedes intentar aprender a transformar automáticamente un boceto en un personaje terminado.
 - ... y tantas otras que se te ocurran.
- Trata de construir una red GAN cíclica y aplícala para la conversión imágenes en alguna de las siguientes aplicaciones:
 - **Verano → Invierno.** Necesitas un conjunto de fotos de paisajes en verano y otro en invierno.
 - **Juventud → Vejez.** Necesitas un conjunto de fotos de gente joven y otro de gente mayor.
 - **Generación fotorealista de escenarios/personas.** Si dispones de modelos de alambre, por ejemplo en OpenGL e imágenes reales del mismo tema, puedes generar una aplicación que transforme el modelo de alambre en una imagen real.
 - ... y tantas otras que se te ocurran.