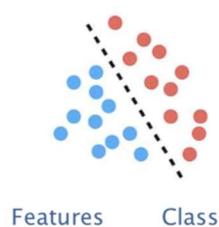


## 16 - GENERATIVE ADVERSARIAL NETWORK

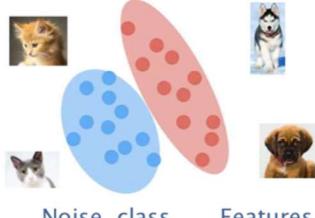
I modelli cercano di imparare a creare una generazione realistica di una determinata classe. Prendono in ingresso un vettore casuale e un'altra informazione riguardante la classe. Il rumore fa sì che la rete generi sempre un risultato diverso.

### Discriminative models



$$x \rightarrow y \\ p(y|x)$$

### Generative models



$$\xi, y \rightarrow x \\ p(x|y)$$

**Discernitori:** modello prende feature dei dati per determinare categoria/etichetta.

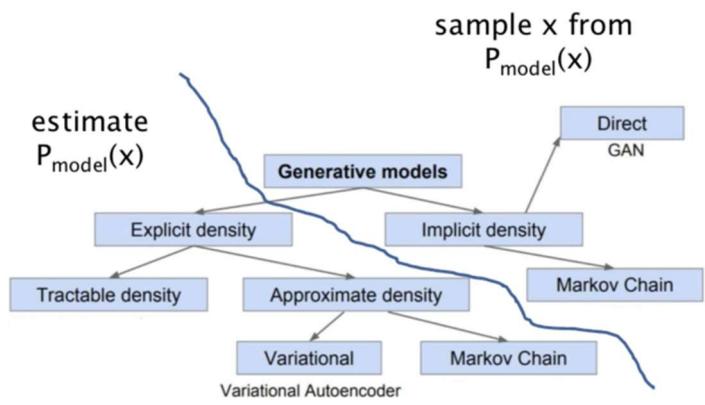
**Generativi:** cercano di imparare come fare a generare una possibile versione realistica di quell'immagine.

- Dati i dati in input, voglio generare in output
  - o Campione che io genero appartenga alla classe di gatti/cani con tutte le loro caratteristiche
  - Rumore mi serve in modo tale che la rete non generi tutte le volte lo stesso cane.

### Taxonomy of generative models

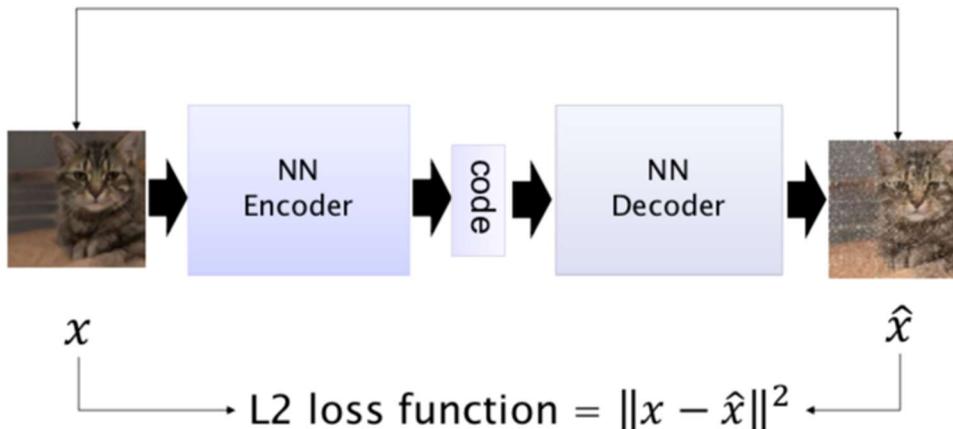
Si dividono in due grandi classi:

- **Modelli esplicativi** che cercano di modellare e definire in maniera esplicita quelle che sono le caratteristiche della distribuzione delle probabilità delle features  $x$ . (es: VAE)
- **Modelli impliciti** che creano dei modelli che sono in grado di campionare la distribuzione  $p(x)$  in modo tale da poter estrarre da questo un nuovo campione nel dominio di interesse. Però non gli interessa andare a fare una definizione esplicita del modello di distribuzione. Basta estrarre un nuovo campione che appartiene alla distribuzione di probabilità del dominio di interesse. (es: GAN e Diffusion model)
  - o GAN: ...
  - o Diffusion model: si basano su catena di markov -> serie di processi in catena che permettono di passare dal rumore al modello generato



## VARIATIONAL AUTO-ENCODERS (VAE)

Variational Autoencoders - espliciti



Funzionano con due blocchettini: encoder e decoder.

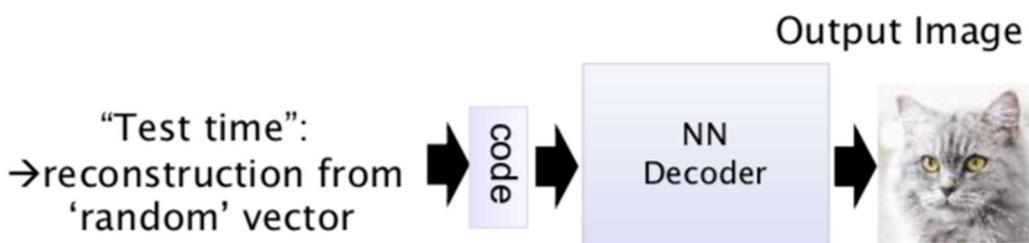
L'idea generale è quella che **l'encoder** prende in ingresso delle immagini reali e trovi un modo adeguato per andare a rappresentare queste immagini in uno spazio latente. Questo ha una dimensione ridotta rispetto a quello delle features di partenza. -> l'effetto finale è quello di ridurre la dimensione del segnale.

La fase di **decoding** fa l'opposto e cerca di ricostruire l'immagine di partenza che è stata data in pasto alla rete.

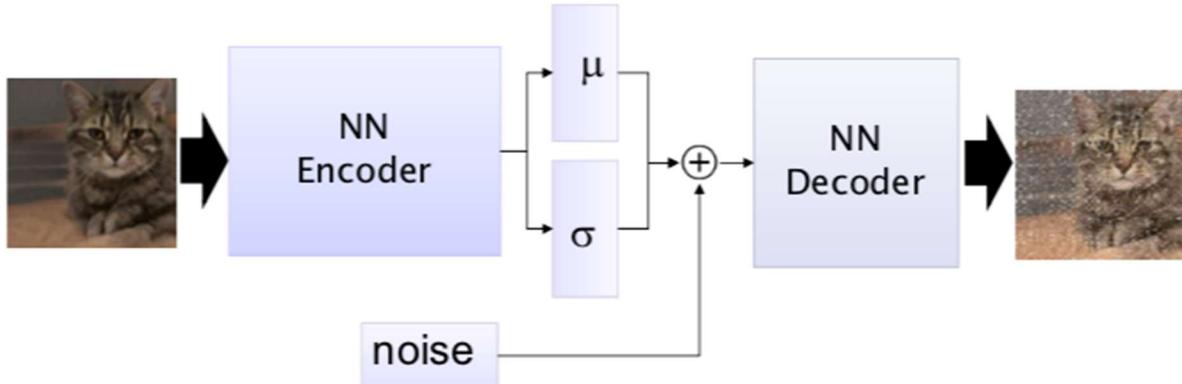
Perché tutto funzioni bisogna addestrare encoder e decoder.

La cosa più logica da utilizzare come **loss** è la differenza tra l'immagine generata e quella data in pasto alla rete.

Dopo l'addestramento per la **fase di test** posso buttare via la parte di encoder e usare solo il decoder che è il modulo che ha imparato a ricostruire le immagini dandogli in pasto vettori randomici.



Questi modelli si definiscono “**variational**” perché usano un approccio approssimato chiamato **inferenza variazionale** e che serve per andare a stimare i parametri della distribuzione che mi interessa. In questo caso è abbastanza complessa. Quello che fa la parte variazionale è andare ad aggiungere al modello **un pochino di rumore sia in training che in test**.



*Nel training la mia immagine data in pasto non sarà più un punto dopo l'encoder ma un intorno. Poi quando devo passare un campione al decoder vado a campionare in maniera casuale l'intorno corrispondente al campione d'ingresso. In questo modo il rumore fa aumentare la robustezza del sistema.*

## Latent space in generative models

Perché vogliamo ricostruire il nostro dominio di interesse partendo da uno spazio latente che ha delle dimensioni decisamente minori dello spazio di partenza?

Il motivo sta nel concetto di **manifold** che rappresenta un sottospazio ristretto dello spazio di rappresentazione.

Ese:

- se ho come spazio di rappresentazione quello tridimensionale  $R^3$ , tutti i punti che stanno in una sfera rappresentando un manifold di  $R^3$ .
- lo spazio delle facce è un manifold (16-19 parametri) dello spazio delle possibili immagini RGB



Quindi il dominio che vogliamo andare a ricostruire è un manifold dello spazio originale delle features.

Si definisce la dimensionalità intrinseca (ID) il numero minimo di variabili necessarie a rappresentare i dati.

Un manifold ha un ID inferiore dello spazio originale → posso rappresentare con meno variabili un certo dominio.

- Ese: per le facce 16/19 parametri



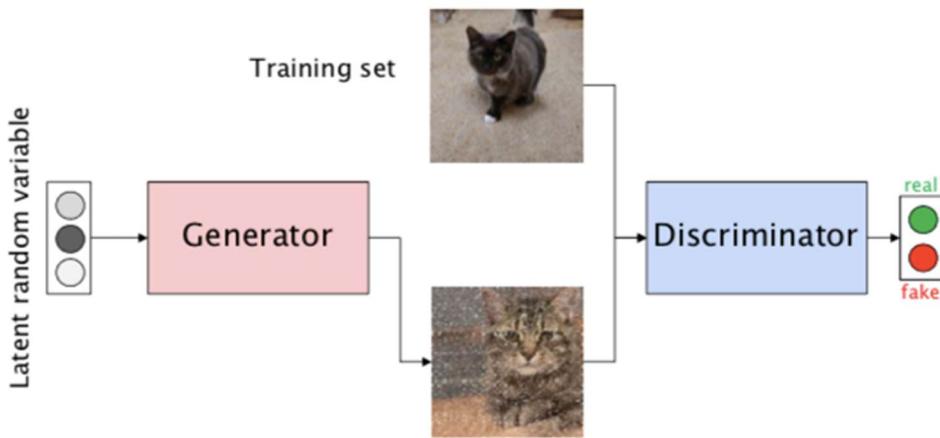
## GENERATIVE ADVERSARIAL NETWORKS (GANS)

### GAN

È un modello che funziona diversamente dal VAE.

Queste reti sono composte da due modelli:

- Un **generatore** che si preoccupa di andare a generare delle nuove immagini.
- Un **discriminatore** che non è altro che un classificatore



Il **generatore** prende un input casuale più l'indicazione della classe che deve generare; quindi, non ho uno spazio di apprendimento delle caratteristiche.

- *Output: immagini che cerca di assomigliare ad un gatto*

Il **discriminatore** riceve in input una serie di immagini reali e quelle create dal generatore e cerca di capire se le immagini sono false o vere.

*Quindi il generatore cerca di fregare il discriminatore e viceversa. Si chiamano avversari proprio per questo.*

- *In questa lotta continua tra una e l'altra, i due modelli si irribuschiscono*

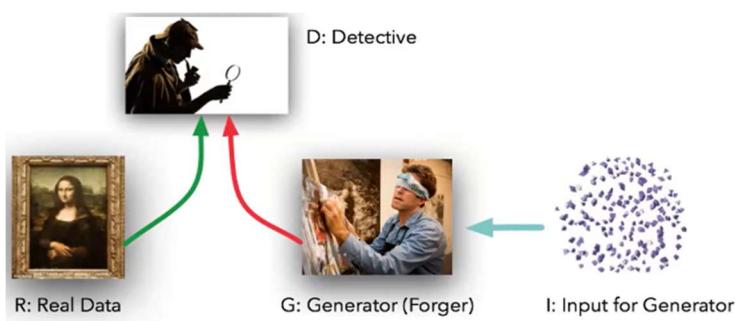
*Il motivo per cui funzionano bene e funzionano meglio di altri approcci (quelli esplicativi) è che non cercano di stimare la probabilità di distribuzione della variabile latente dato il campione  $x$  in ingresso, ma campionano la probabilità di  $x$  data una variabile presa in maniera casuale dallo spazio latente.*

Il campionamento di questa probabilità verrà definita dalla funzione  $g$  implementata dal generatore. Questa funzione la potrà imparare con la backpropagation durante la fase di addestramento.

I **potenziali problemi** di questa architettura base sono:

- Il generatore impara solo in base al training set e questo non è un problema solo per piccoli training set e piccole variazioni
- Il generatore non ha modo di capire il tipo di immagine da produrre tipo con i gatti il colore del pelo, la lunghezza della coda, etc. → immagine in uscita è casuale

### Falsario d'arte vs esperto d'arte



All'inizio il detective non si accorge  
Il detective inizia ad accorgersene  
Il falsario viene informato e prova ad attuare  
modifiche e rimanda al detective finché questo non ci  
casca -> a questo punto il falsario sa che ha aggiunto  
qualcosa che ha fatto sbagliare il detective.

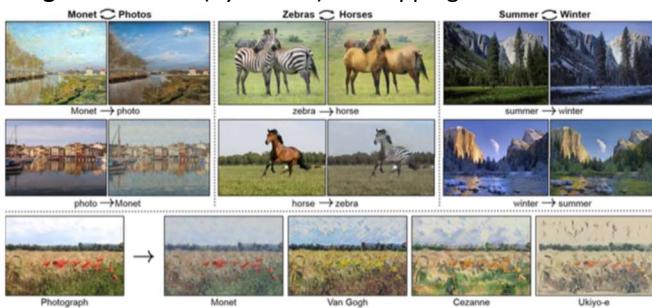
Ad una certa, esperto d'arte se ne accorge, il falsario  
cambia cosa fa etc etc. Arriveremo ad un certo punto  
dove il falsario sarà diventato bravissimo.

## GAN applications

### Synthetic image generation



### Image translation (Cycle Gan) → Mapping between different image domains



- Da video di cavallo che corre a video di sebra che corre
- Prendere una foto e trasformarla in stile dell'altro pittore

*NOTA: per imparare a fare queste traduzioni, le GAN non hanno bisogno di immagini accoppiate (non hanno bisogno di avere già un'immagine rappresentata sia in un dominio che nell'altro, non hanno bisogno di avere un cavallo e una zebra nella stessa posizione) ma hanno bisogno di una serie di immagini di un dominio e una serie di immagini in un altro dominio e riescono a fare la traduzione.*

### Sketches to images (GauGAN)

Aiutare a creare delle immagini realistiche a partire da sketch fatti dagli utenti. (integrato in Canvas)

### Talking heads from photos

Partendo dalle foto, usando movimento e nostra voce.

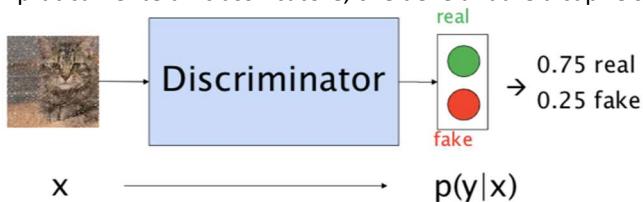
### Procedural modeling (3D-GAN)

Generare modelli 3D

## GAN ANATOMY

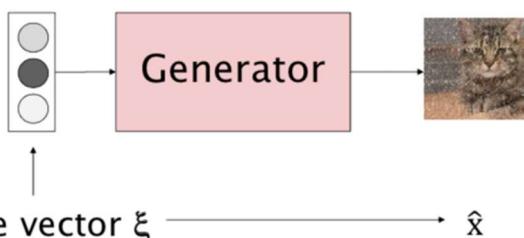
### Discriminatore

È praticamente un classificatore, che deve andare a capire se l'immagine è vera o falsa.



### Generatore

È quello su cui bisogna investire di più



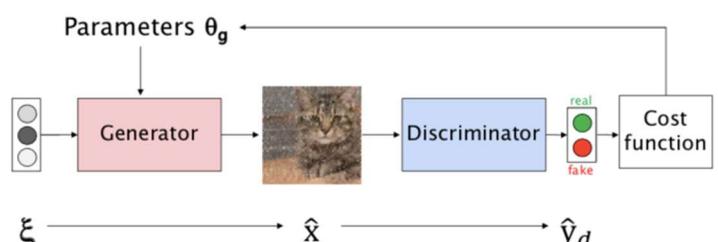
deve generare oggetti per tutte le classi su cui è stato addestrato.

- Do in pasto dei vettori di numeri completamente casuali

### Come impara il generatore

- crea in uscita partendo da vettori di numeri completamente casuali
- generatore crea campione che viene ammesso al discriminatore
- discriminatore calcola probabilità
- attraverso una funzione di costo, valuto l'immagine creata e mando al generatore che può aggiornare i suoi parametri theta.

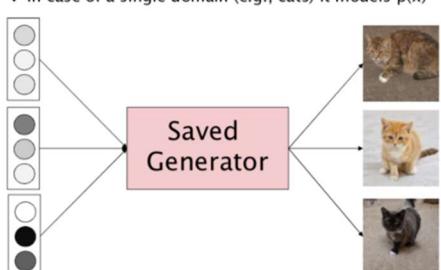
- Generator:  $\hat{y}_d$  as close to 1 (real) as possible...
- Discriminator:  $\hat{y}_d$  as close to 0 (fake) as possible



### Sampling

The generator models  $p(x|y)$

- In case of a single domain (e.g., cats) it models  $p(x)$



Non cerca di ricostruire la distribuzione, si preoccupa soltanto di aver capito come fare a creare un campione che casca all'interno della distribuzione dei dati di interesse.

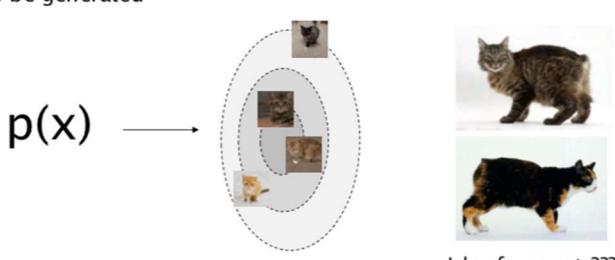
### Which distribution?

The distribution approximation depends from the characteristics of the training set

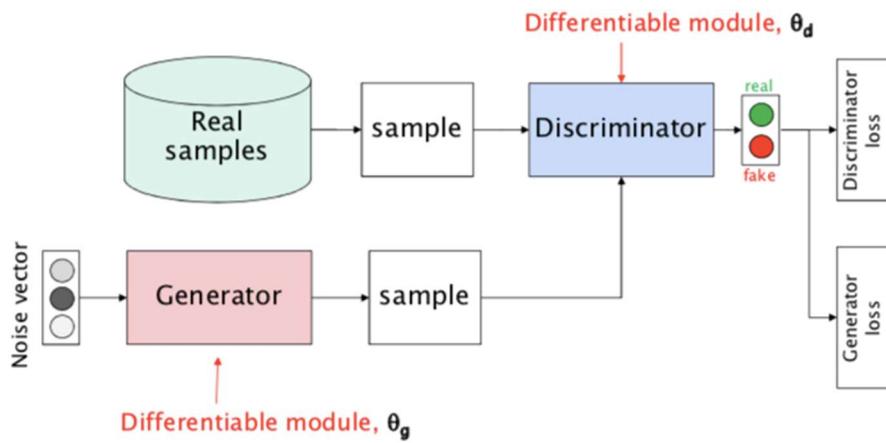
- Most "common" samples in the training sets are more likely to be generated

La distribuzione è definita all'interno dei dati del training set, dunque se il mio training set sta sotto-campionando il dominio di interesse, il mio generatore impara solo su quella sotto-partite.

- Es: se ho tanti gatti con il pelo lungo e pochi con pelo corto -> molto probabile che il generatore generi a pelo lungo.



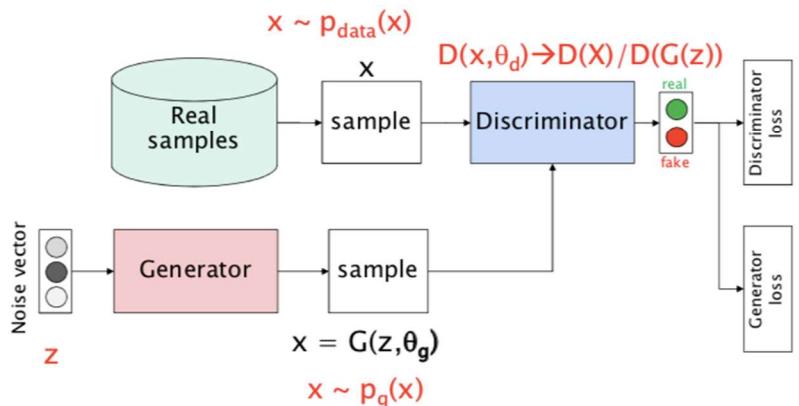
## GAN architecture



Sia il generatore che il discriminatore **sono reti neurali** e quindi sono rappresentabili con delle funzioni differenziabili e dipenderanno dai parametri  $\theta$ . L'implementazione dei due moduli dipenderà dall'architettura scelta e soprattutto dal dominio su cui sto lavorando. **X è sempre l'insieme di feature del campione.**

Definiamo con:

- $p_{\text{data}}$ : distribuzione di probabilità dei campioni reali
- $p_g$ : distribuzione di probabilità dei campioni generati.
- **X**: features del campione.
- **z** è il vettore di rumore casuale che non è altro che la rappresentazione latente dell'immagine  $x$  generata dal generatore. Questa immagine appartiene a  $p_g$  perché il generatore non vede mai i campioni reali, quindi, non può andare ad attingere dalla distribuzione  $p_{\text{data}}$ .



La **funzione del discriminatore** è indicata con  $D$  e prende in ingresso un'immagine  $x$  e i parametri del discriminatore e darà in uscita due valori diversi:

- $D(x)$  se si sta lavorando con dataset di immagini reali o
- $D(G(z))$  se si sta lavorando con l'immagine generata.

Una volta che ho l'output del discriminatore posso calcolare le due loss per poter aggiornare entrambi i moduli.

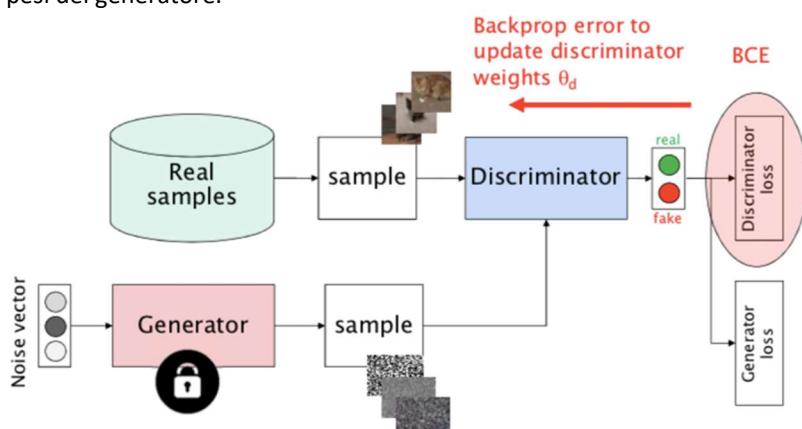
Una volta calcolate le **loss** l'**algoritmo** di addestramento delle GAN funziona in due modi separati:

- Vado ad aggiornare il discriminatore
- Vado ad aggiornare il generatore

### Training discriminator – aggiornamento discriminatore

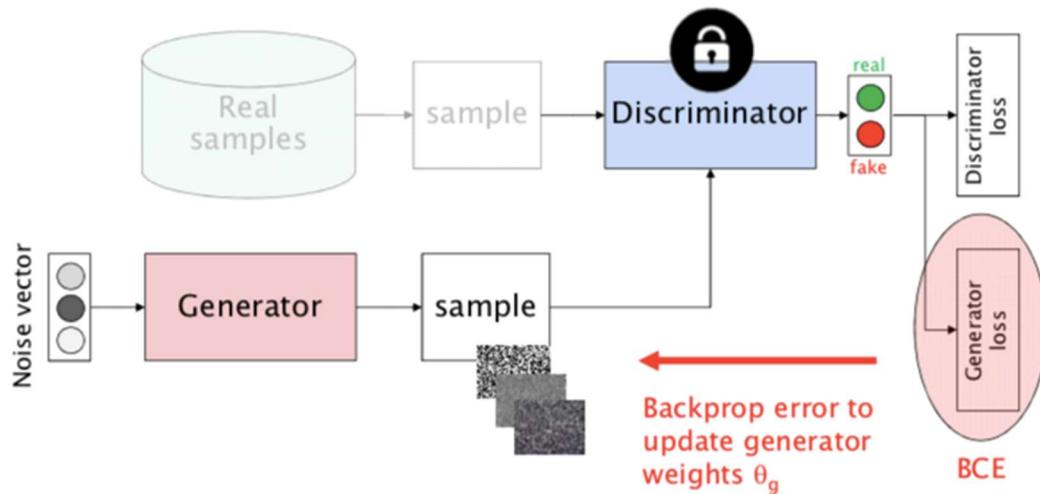
Quando vado ad addestrare il discriminatore vado ad **estrarre un mini-batch grande**  $m$  di campioni reali dal training set e **un altro mini-batch grande** sempre  $m$  di immagini che sono state generate dal generatore.

Il discriminatore va a calcolare la sua loss di tipo binary cross entropy loss e va ad aggiornare i suoi pesi mantenendo bloccati i pesi del generatore.



## Training generatore – addestramento generatore

Seleziona un mini-batch  $m$  di immagini generate e basta, le passo al discriminatore e calcolo la loss del generatore facendo poi un passo indietro per andare ad aggiornare i pesi del generatore. In questo caso mantengo bloccato il discriminatore.



## GAN's formulation

Durante la fase di training aggiorno un modello per volta congelando l'altro. Quando faccio questo processo è importante far migliorare entrambi i modelli; quindi, **fin dal training devo mantenere tra loro performance simili**. Se ad esempio avessimo un discriminatore perfetto, il generatore non avrebbe nessuna informazione per andare a migliorare e viceversa.

La soluzione di questo problema è la soluzione di quello che nella teoria dei giochi è il minimax game che è lo stesso che si usa per **far imparare un programma** a giocare a scacchi, a dama, etc.

Abbiamo una funzione obiettivo che definisce una loss per un discriminatore e una loss per un generatore che è semplicemente il negativo del costo del discriminatore. Si può pensare ad un **unico valore  $V(D,G)$**  che il discriminatore cerca di massimizzare e il generatore di minimizzare.

$$\min_G \max_D V(D, G)$$

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

i.e. real  $x$  is labelled real      i.e. fake  $x$  is labelled fake

L'equilibrio si chiama **equilibrio di Nash** ed è quel punto in cui nessuno tra i due giocatori riesce a definire delle strategie individuali che gli permettono di cambiare in meglio.

Siamo interessati a raggiungere e mantenere questo equilibrio che si verifica quando il discriminatore non è più in grado di distinguere tra un falso e un vero. Detto in altri termini si verifica quando la distribuzione di probabilità dei dati veri converge con quella dei dati falsi.

The (Nash) equilibrium of this game is achieved at

- ♦  $P_{\text{data}}(x) = P_g(x) \quad \forall x$
- ♦  $D(x) = \frac{1}{2} \quad \forall x$

Se usiamo questa loss  $V(D,G)$  dobbiamo utilizzare, per il discriminatore, non più il gradient descent ma il **gradient ascent**.

Discriminator needs to predict 1 for real image, 0 for fake ones

The loss function for the discriminator is:

$$\begin{aligned} J^D &= V(D, G) \\ &= \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_g(z)} [\log(1 - D(G(z)))] \end{aligned}$$

We want to **maximize** this loss function, so we perform **gradient ascent** as

$$\theta_d \leftarrow \theta_d + \mu \nabla V(\theta_d)$$

Generator needs to fool the discriminator (it needs that the discriminator outputs 1 for fakes)

The loss function for the generator (no real images are used) is:

$$J^G = V(D, G) = \mathbb{E}_{z \sim p_g(z)} [\log(1 - D(G(z)))]$$

We want to **minimize** this loss function, so we perform **gradient descent** as

$$\theta_g \leftarrow \theta_g - \mu \nabla V(\theta_g)$$

C'è un problema col minimax game che è quando il discriminatore diventa troppo intelligente, il gradiente del generatore tende a scomparire avendo problemi come il vanishing gradient.

Quando uso la BCE nel mio addestramento, ogni volta che D fa un errore va a scegliere quella che è la classe sbagliata e quindi il suo gradiente sarà diverso da zero.

Purtroppo, non succede lo stesso con il gradiente di G. Innanzitutto perché una parte me la sono persa e poi perché, se vado a negare il costo di D allora G sarà l'opposto.

In poche parole, se G non inganna D allora poi non ho nessun gradiente perché l'uscita del discriminatore si è saturata e quindi il gradiente del generatore tende a saturarsi.

Andiamo a calcolare il gradiente della funzione di costo

$$J^D = \mathbb{E}_{x \sim p_{data}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_g(z)} [\log(1 - D(G(z)))]$$

$$J^G = \boxed{\mathbb{E}_{z \sim p_g(z)} [\log(1 - D(G(z)))]}$$

$$\nabla_{\theta_G} V(D, G) = \nabla_{\theta_G} \mathbb{E}_{z \sim p_g(z)} [\log(1 - D(G(z)))]$$

$$\begin{aligned} \nabla_a \log(1 - \sigma(a)) &= \frac{-\nabla_a \sigma(a)}{1 - \sigma(a)} = \frac{-\sigma(a)(1 - \sigma(a))}{1 - \sigma(a)} = \\ &= -\sigma(a) = -D(G(z)) \end{aligned}$$

Quello che possiamo fare è andare ad invertire la funzione di costo del generatore quindi invece di andare a cercare di minimizzare la probabilità della risposta corretta del discriminatore, cerco di **massimizzare la probabilità della risposta sbagliata di D**.

Se vado a calcolarmi il gradiente di questa ottengo che è pari a D e quindi che non sparisce a meno che D sia assolutamente perfetto. In questo caso ho un processo di addestramento un pochino più stabile.

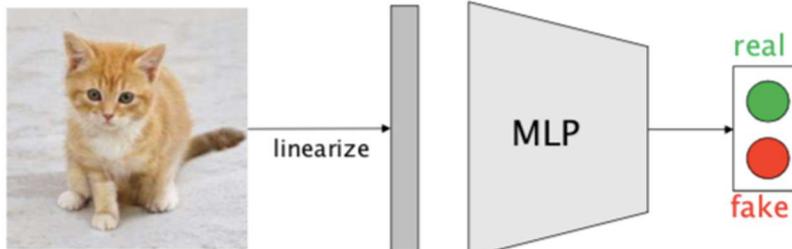
$$J^D = \mathbb{E}_{x \sim p_{data}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_g(z)} [\log(1 - D(G(z)))]$$

$$J^G = \mathbb{E}_{z \sim p_g(z)} [\log(D(G(z)))]$$

## DEEP CONVOLUTIONAL GANS (DCGANs)

### From GANs to DCGANs

L'architettura originale delle GAN utilizzava modelli fully connected che non va bene per le immagini perché c'è un'esplosione dei parametri -> non scala



3 channels, 256x256

Input vector size: 196,608

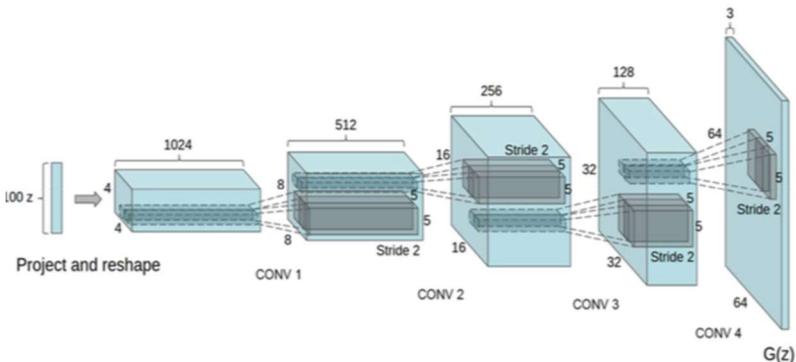
One layer with 100 output neurons:  
19,660,800 parameters....

TOO many parameters

### DCGAN architecture – buona per immagini

Per il generatore viene fatto un Upsampling convoluzionale.

#### ▪ Generator architecture



- Si basa su blocchi convoluzionali
- Da rumori casuali di ingressi a immagini RGB usando dei layer de-convoluzionali che permettono di andare ad aumentare la dimensione delle immagini o delle feature in maniera parametrica

### DCGAN first results



Figure 3: Generated bedrooms after five epochs of training. There appears to be evidence of visual under-fitting via repeated noise textures across multiple samples such as the base boards of some of the beds.

The breakthrough of the paper was putting forth the main key ideas/recommendations used in all following works

- ♦ Replace max pooling with convolutional stride
- ♦ Replace FC layers with conv layers
- ♦ Use deconvolution (transposed convolutions) in generator for upsampling
- ♦ Use batch normalization after each layer (except the generator output)
- ♦ In the generator, use ReLU in hidden layers and tanh for the output. In the discriminator, use LeakyReLU

## GANS: TRAINING CHALLENGES

L'addestramento delle GAN è tutt'altro che semplice a causa dei problemi di convergenza del modello e a quello del collasso delle mode.

### Non-convergence

Questo capita perché si ha una differenza sostanziale rispetto agli approcci standard in cui si ha un solo attore che cerca di trovare i parametri ottimali.

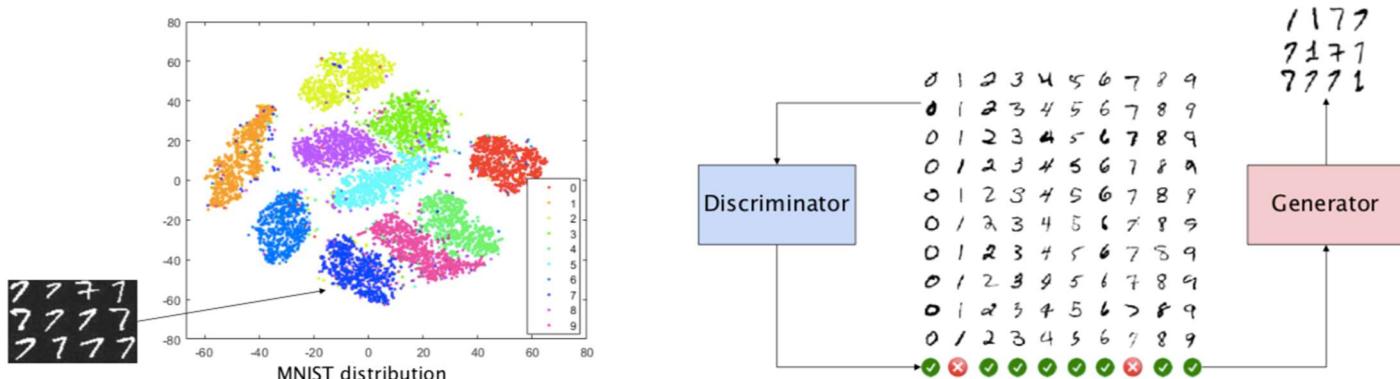
Nelle GAN ci sono i due attori che cercano di combattere e la soluzione si ha nell'equilibrio di Nash. Peccato che SGD non è progettato per trovare l'equilibrio di Nash e quindi può benissimo succedere che non si converga mai. (Ci sono delle dimostrazioni che provano che posso perlomeno arrivare ad un ottimo locale)

### Mode collapse

Perdita o collasso di una modalità della distribuzione.

In una distribuzione di dati, una moda è dove c'è una concentrazione di questi dati. Con le GAN le distribuzioni dei dati sono multimodali.

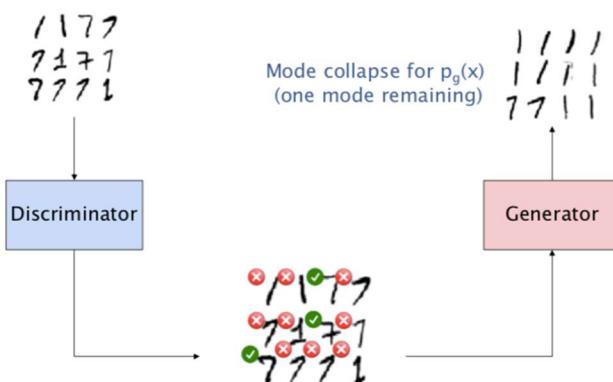
Ogni colore rappresenta una classe.



Alcune delle modalità collassano. In questo caso il discriminatore sta riuscendo a comprendere sempre tranne quando è un 1 o 7  
(ha raggiunto un minimo locale nella funzione obiettivo).

Il generatore riceve l'informazione, capisce questo e allora prova a fregare il più possibile il distributore, generando una grande quantità di immagine con 1 e 7.

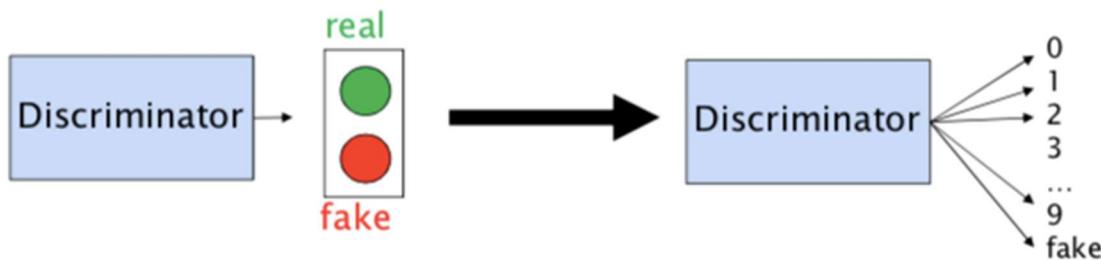
- Discriminatore impara un po'
- Generatore capisce come fregarlo e si concentra sempre di più con il numero 1
  - ...
  - ...
  - Ad una certa discriminatore capisce come lavorare sull'1
- Generatore è fregato perché non sa più generare altri numeri ma solo 1 -> COLLASSO -> posso buttare via il generatore



## Alternate GAN losses

Un modo per risolvere il mode collapse è quello di andare a cercare di **aiutare il discriminatore** modificando la loss che sto usando. Una possibilità è quella di usare un metodo puramente euristico dove non si cerca più di distinguere tra vero o falso, ma di classificare tra le varie classi.

I campioni che si ottengono sono decisamente migliori e il discriminatore è più robusto solo che non è ben chiaro perché funziona ma funziona.



Un'altra possibilità è quella di andare ad usare altre loss tipo Mean Square Error, Energy based losses che usano auto-encoder per rigenerare l'immagine partendo dall'output passando per uno spazio latente andando a fare qualche calcolo.

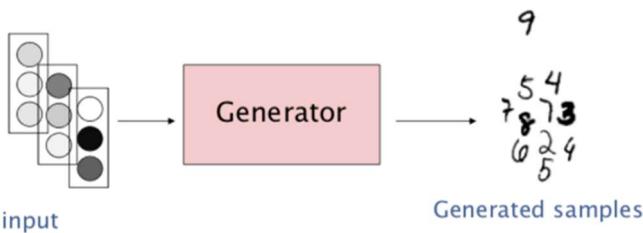
Altra opzione è usare un'altra loss -> Wasserstein loss che modifica il discriminatore in modo tale che assegna un valore reale assegnando uno score (invece che assegnare una probabilità) -> valore più alto possibile a valori che assomigliano a valori reali e basso a quello dei falsi.

## IMPROVING GANS

## *Unconditional generation*

Creo un campione che appartiene ad una determinata distribuzione (senza poter scegliere – esempio mensa)

Non è si assolutamente in grado di andare a scegliere l'output. È un vantaggio perché non ho bisogno di etichettare ogni classe.

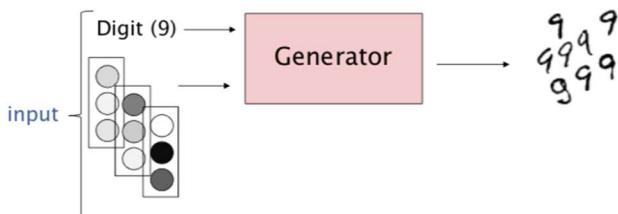


*Es: addestro il mio generatore a creare delle immagini con numeri scritti a mano, ad un certo punto lui impara a farlo, quindi ogni volta che gli do in pasto un vettore di numeri casuali, lui mi genera un numero assolutamente casuale. Non sono in grado di dire al mio generatore di generare un 9.*

*Con la generazione condizionata, si vuole poter specificare questa cosa.*

## *Conditional generation*

Si forniscono **due input al generatore**: il vettore casuale e l'etichetta della classe che si vuole generare.



Es: specificando l'etichetta, genera, a partire dai vettori casuali, il numero che richiedo. (precisazione: lo stesso vettore, cambiando l'etichetta, genera numero diverso.)

Per riuscire nell'obiettivo, il training ha bisogno di un dataset con le label. (immagine + label)

Bisogna ovviamente andare a capire come andare a modificare l'architettura per gestire quest'ulteriore informazione (label). Generator input

- ◆ Noise vector
  - ◆ Class vector (one-hot vector)



L'informazione della classe viene passata come un vettore **"one-hot"**. Quindi quello che si fa è andare a concatenare il vettore di rumore con il vettore one-hot.

*In ingresso stiamo dando la stessa identica sequenza di vettori di rumore, cambiando solamente la classe a cui deve appartenere -> in corrispondenza degli stessi identici vettori di rumori in ingresso, ottengo diversi numeri in base alla classe.*

Anche il discriminatore riceve l'informazione di classe, per poter poi determinare in uscita se l'esempio che gli è stato passato sia un esempio reale o meno di quella specifica classe.

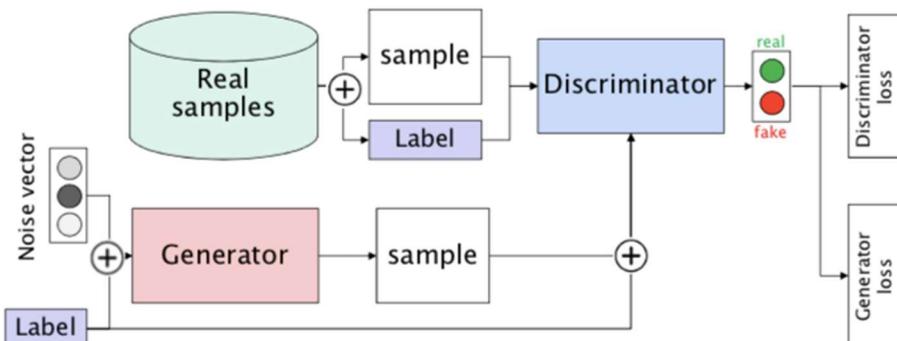


Il generatore per andare ad ingannare il discriminatore deve andare ad associare l'etichetta 9 all'immagine di un 9 che è quasi perfetto.

Il discriminatore imparerà a dire che quell'immagine è realistica solo se somiglia ad un gatto.



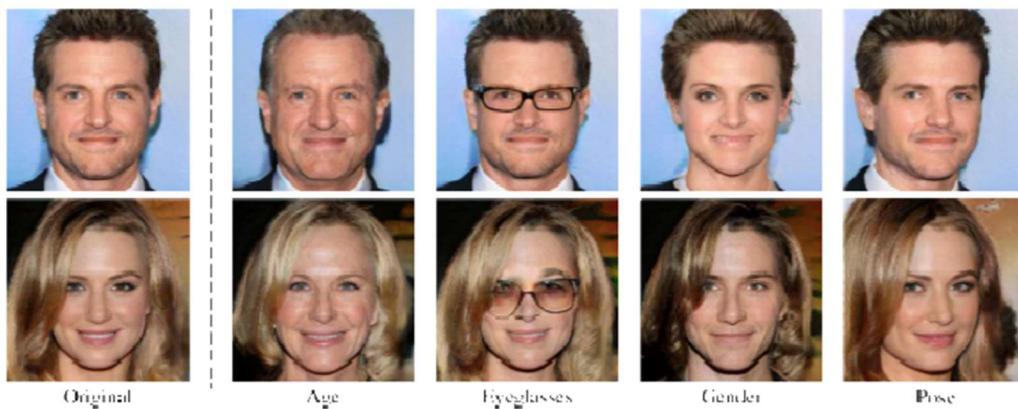
**L'architettura complessiva** è la seguente:



## Controllable generation

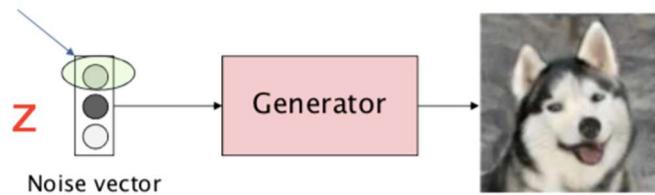
Si applica dopo che il modello è stato addestrato per controllare alcuni degli output generati. -> caratteristiche specifiche dell'immagine generata.

Si vuole ottenere l' **output con delle precise caratteristiche** (es: aggiungere barba, cambiare età, lavorare sul sesso, modificare la posa)



Si può agire sui singoli elementi del vettore di rumore in ingresso. Si può supporre che andando a modificare un singolo elemento di  $z$ , si riesca a controllare una caratteristica

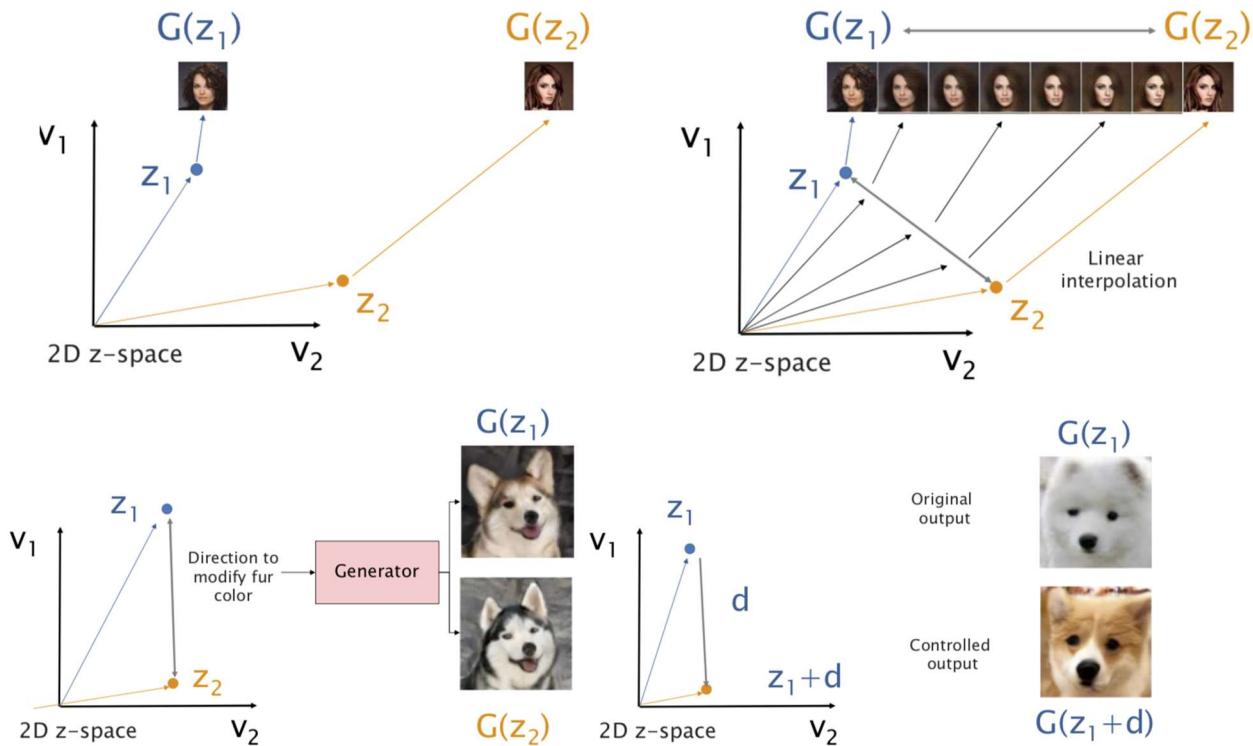
Fur color



Il segreto sta nell'interpolazione dello spazio latente che permette di **interpolare tra gli output generati**.

**Interpolating in the z-space**

$z_1$ : volto di una persona,  $z_2$ : volto di un'altra persona. Tra di loro, volti con caratteristiche miste.



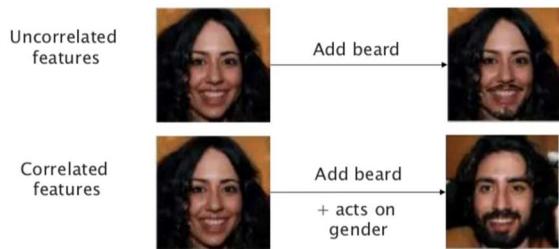
Differenza tra  $z_1-z_2$  mi indica come cambiare il colore della pelliccia (ascoli), ora so in che direzione muovermi, posso muovermi lungo la direzione che ho imparato per cambiare il colore della pelliccia ad un altro cane.

L'obiettivo generale è proprio capire la **direzione lungo cui muoversi per modificare la caratteristica** ed ottenere l'interpolazione dell'output.

Il problema non è così semplice visto che gli spazi latenti sono a più dimensioni e la direzione non è lineare.

## Output feature correlation

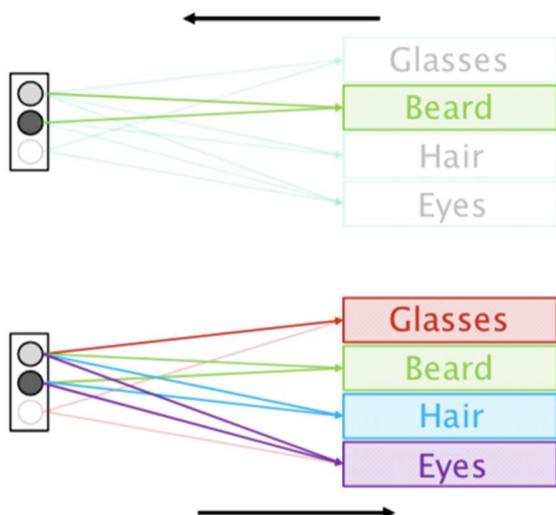
È complicato andare ad agire su una singola caratteristica senza andare ad interferire con altre. (es: la presenza di barba è correlata al genere in quanto nel dataset è molto pronabile non ci siano donne barbate -> essendo la caratteristica barba, molto legata alla caratteristica sesso all'interno del dataset di addestramento, NON riesco ad isolare i due effetti -> la TRANSformo in un essere più maschile)



Noi vorremmo riuscire ad aggiungere la barba mantenendo la donna.

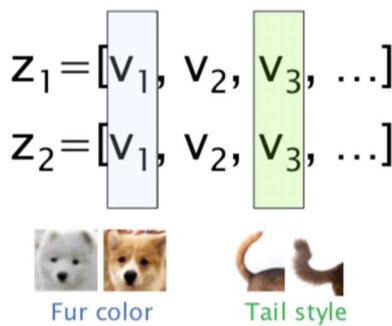
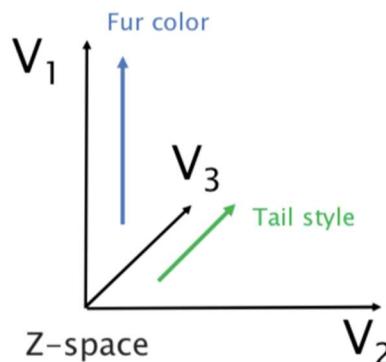
## The z-space entanglement

Il problema è la correlazione tra le variabili nello spazio latente e quelle che sono le caratteristiche specifiche di un dato output. Quindi spostandomi in una direzione potrei influire su più di una caratteristica dell'output (anche se nel mio dataset di addestramento non c'è una fortissima correlazione tra queste caratteristiche) -> correlazione si potrebbe generare durante l'addestramento nello spazio latente.



## Disentangled z-space

Spazio non troppo imbrogliato -> DISTRICATO

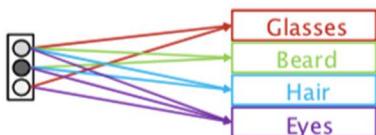


Se riuscissi ad avere uno spazio districato potrei individuare delle dimensioni ognuna associata ad una dimensione di uscita.

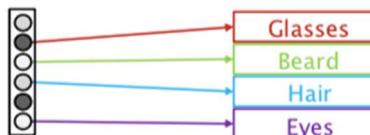
→ Riesco a distinguere lo stile dal colore.

Per ottenerlo bisogna avere lo spazio z di dimensioni maggiori del numero di dimensioni che si vogliono controllare.

Entangled Z-space



Disentangled Z-space



Però nella fase di addestramento non è assicurato che una caratteristica venga associata ad una sola variabile; quindi, bisogna utilizzare un regolarizzatore che incoraggia il disentanglement.

Posso inserire delle label extra per distinguere ma non è un granché in quanto dovrei aggiungere molte label perciò posso optare per aggiungere degli elementi alla loss, regolarizzatori che permettano di raggiungere il disentanglement.

Dunque, vado a mettere una loss di regolarizzazione per incoraggiare il modello ad associare ad ogni variabile una caratteristica diversa delle immagini -> etichettatura delle immagini.

Nota sui possibili regolarizzatori:

- **Mutual information:** si basa sull'informazione mutua -> concetto statistico che misura quanto una variabile casuale x ci dice rispetto alla variabile casuale y
  - Dipendenza statistica tra le due variabili.
  - L'idea è massimizzare la dipendenza tra i fattori latenti z1 che voglio usare per il controllo e i dati che poi vengono generati.
- **Total correlation:** minimizzare la correlazione totale tra tutte le variabili latenti in modo da riuscire ad ottenere delle variabili all'interno dello spazio latente che siano il più possibile indipendenti tra di loro.
- **Geometric regularization:** Penalizzare le discrepanze che vengono generate nell'output quando mi muovo lungo una direzione dello spazio latente.
  - bisogna trovare una funzione per rilevare la coerenza/discrepanza

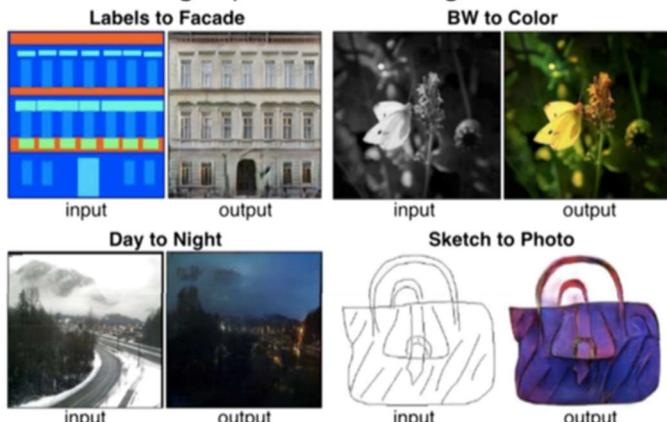
$$\mathcal{L}_{new} = \mathcal{L}_{adversarial} + \lambda_1 \cdot \mathcal{L}_{regularizer}$$

Can be any loss  
(BCE, MSE, ...)

## GAN APPLICATIONS

### Image to image translations (i2i)

- Transferring style from one image to another

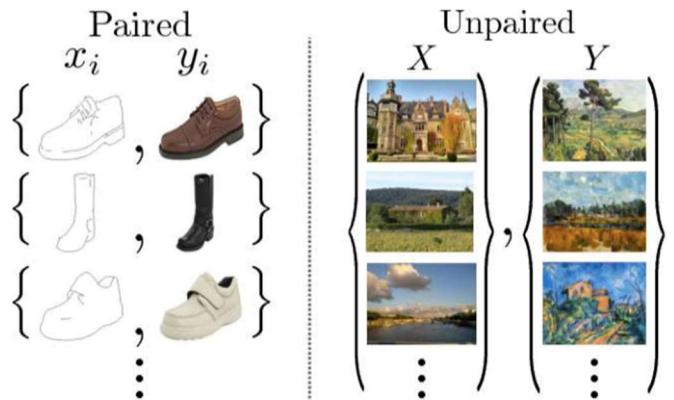


Oltre allo stile, posso modificare altro dell'immagine

- da immagini etichettate con determinate etichette a generare in output immagine realistica che ha tutti gli oggetti rappresentati dalle etichette
- da bianco e nero a colori
- da giorno a notte
- da sketch a mano a immagine realistica
- ...

Si può fare in due **modi diversi**

- Paired:** vado ad accoppiare uno specifico input ad uno specifico output dove si ha lo stesso contenuto ma lo stile diverso.  
Quindi si va a condizionare l'output in funzione di una determinata funzione di input.  
Il problema è che è complicata la creazione dei data set di dati accoppiati. (impossibile avere due immagini con cavallo e zebra nella stessa identica posizione, nello stesso identico contesto)
- Unpaired:** è un po' più complicato per la computazione perché gli elementi stilistici sono scorrellati anche tra coppie di immagini
  - Imparo gli elementi stilistici delle varie immagini e trasformo

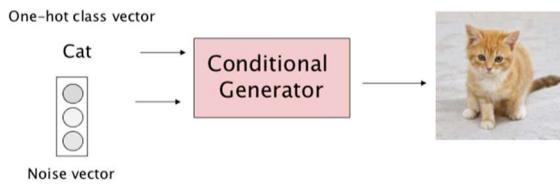
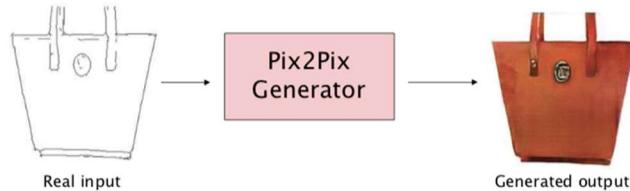


Nota: Si vuol imparare elementi stilistici dei domini e trasformare da un dominio all'altro

- Capire elementi che vanno mantenuti comuni (es: se voglio trasformare cavallo in zebra, devo mantenere lo sfondo e trasformare solamente le parti del cavallo in zebra)
  - Nel caso pair -> faccio banale confronto tra le coppie
  - Nel caso unpair -> devo impararli, in quanto sono pressoché ignoti

## Paired i2i: Pix2Pix

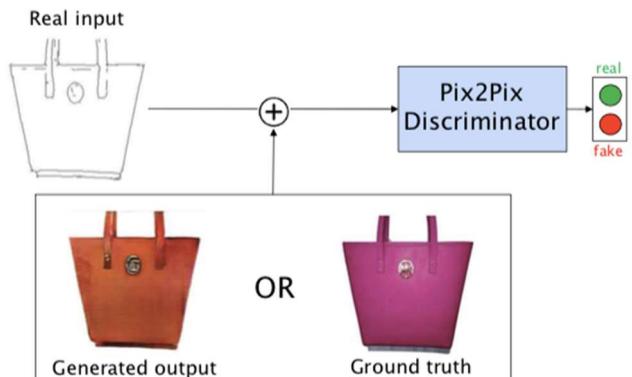
È una specie di GAN per la generazione condizionata.



Il **generatore** Pix2Pix prende un'intera immagine come input, non il vettore di rumore, e punta a produrre uno specifico output (la paired image).

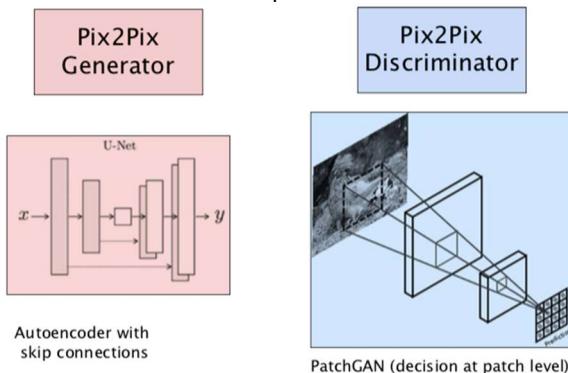
Il **discriminatore** prende come input lo stesso del generatore però concatenato con o l'immagine generata dal generatore o con il ground truth (immagine che ho accoppiato allo sketch, nel mio dataset di training).

Quindi il discriminatore deve capire se la coppia è quella del training set o quella generata. Non c'è più lo scopo della discriminazione di classe come avviene normalmente nelle GAN condizionate (non cerca di capire se è una borsa ma cerca di capire se corrisponde allo sketch che gli è stato dato in ingresso)



## Pix2Pix module architecture

I moduli sono diversi da quelli usati normalmente nelle GAN.



Il generatore è un modello U-Net ovvero un AutoEncoder con delle **skip connection**.

Il discriminatore è una ConvNet che però genera in uscita non un solo valore ma una matrice e il discriminatore prende una decisione su ogni patch della matrice generata. Questo mi permette di dare un feedback migliore al generatore in quanto è mirato nelle varie parti dell'immagine.

## Pix2Pix Loss

La loss è quella classica del modello adversarial però contiene anche un elemento in più che è un **regolarizzatore** che fornisce al generatore quelle informazioni in più che gli servono. Il termine aggiuntivo va a considerare la differenza tra il pixel del campione generato e quello del campione reale. Il parametro lambda serve ad andare ad equilibrare i contributi dei due elementi.

$$\mathcal{L}_{\text{Pix2Pix}} = \mathcal{L}_{\text{adversarial}} + \lambda \cdot \mathcal{L}_{\text{pixel}}$$

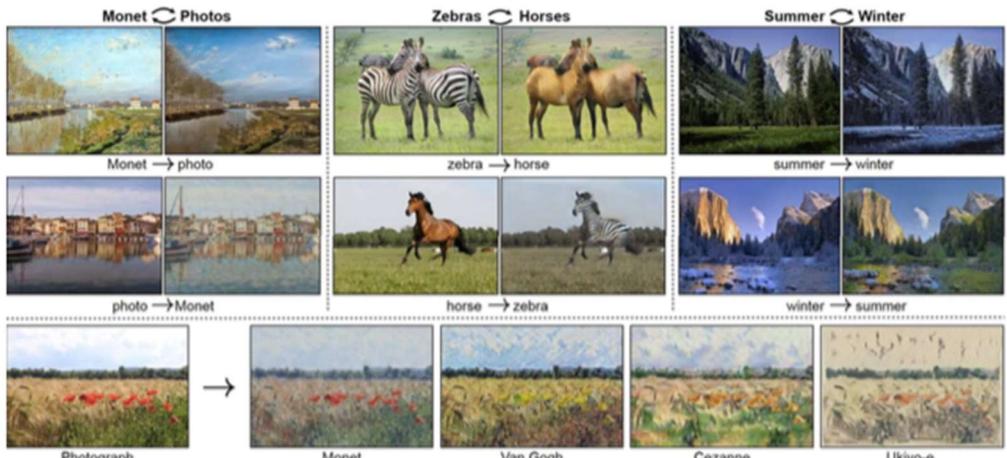
$$\mathcal{L}_{\text{pixel}} = \sum_{i=1}^n |generated - real|$$

$$\mathcal{L}_{\text{pixel}} = \sum_{i=1}^n \left| \begin{array}{c} \text{generated} \\ \text{---} \\ \text{real} \end{array} \right| - \left| \begin{array}{c} \text{generated} \\ \text{---} \\ \text{real} \end{array} \right|$$

## Unpaired i2i: CycleGAN

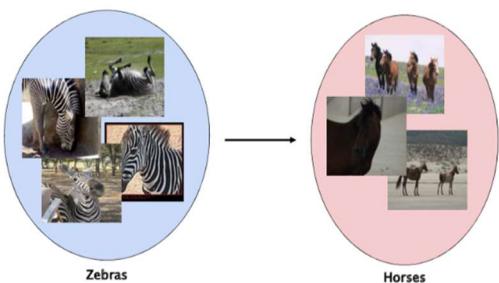
Puntano a trovare similarità e differenze tra i due domini. I modelli devono essere in grado di capire quali sono gli elementi in comune e quelli che caratterizzano le singole immagini per poter trasferire lo stile.

- Contenuto: forma generale dell'animale e sfondo immagine
- Stile: colore del manto (strisce vs colore marrone)



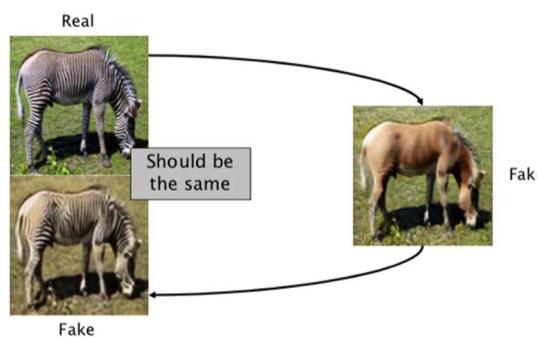
## CycleGAN

L'intuizione per far funzionare l'approccio è questa: consideriamo l'esempio di zebre e cavalli.



Raccolgo tutte le immagini dei due domini, le raccolgo e le metto nei due insiemi.

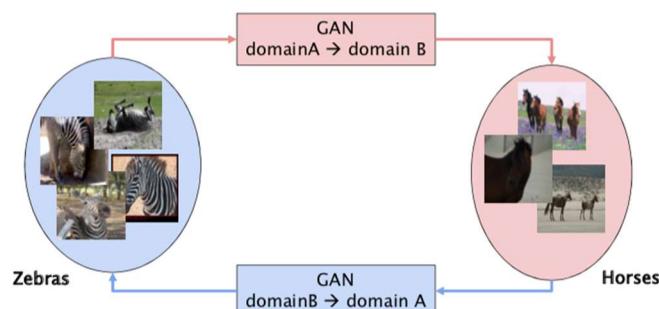
Quindi genero da un'immagine reale di una zebra una falsa di un cavallo e da quest'ultima applicando la stessa funzione di trasformazione genero una falsa di una zebra. Lo scopo è che l'ultima e la prima immagine devono essere uguali.



Questo è quello che si chiama **cycle consistency**.

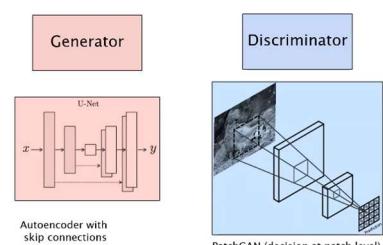
Per realizzare tutto ciò ho bisogno di due GAN per il passaggio corrispettivo da un dominio all'altro e poi applicare la consistency per preservare i contenuti.

- Cycle consistency (generators) → preserve contents
- Discriminator loss → style transfer



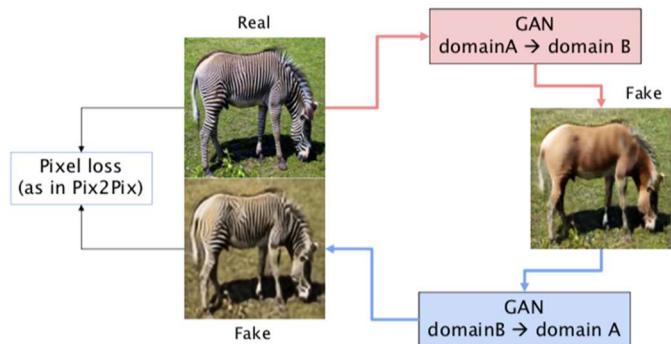
Discriminatore: assicura il realismo dell'immagine generata, quindi l'aderenza a quello stile

**Nota:** L'architettura dei due moduli è molto simile al Pix2Pix, generatore un po' più complesso.



## Loss: cycle consistency loss

Avremo un adversarial loss per ognuna delle due GAN però non mi bastano.



Ne serve una per **mantenere la condizione di consistenza** del ciclo per entrambe le direzioni quindi applico una **loss pixel** tra l'immagine di partenza e quella finale che dovrebbero essere uguali → cycle consistency loss che è condivisa da entrambi i generatori.

$$\mathcal{L}_{cycle\_consistency} = \sum_{i=1}^n \left| \text{Fake}_i - \text{Fake}_{i+1} \right| + \sum_{i=1}^n \left| \text{Fake}_{i+1} - \text{Fake}_i \right|$$

Zebra→Horse→Zebra      Horse→Zebra→Horse

Questa la sommo quindi all'adversarial loss, usando pesi opportuni.

## Adversarial loss

La formulazione originale è quella della loss L2(least square loss) perché questo tipo di loss riduce il problema del vanishing gradient e del mode collapse.

Per quanto riguarda il **discriminatore** ho il primo termine che riguarda le immagini reali (per semplicità non considero le patch dell'immagine) dove si punta ad avere l'output del generatore esattamente a 1 e nel caso di un batch vado a prendere il valore atteso su tutte le immagini reali.

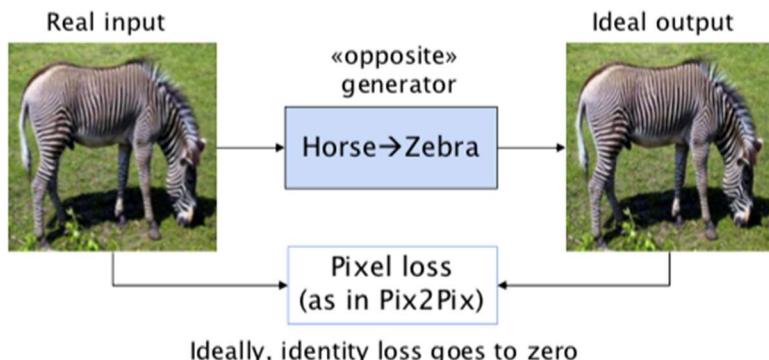
**Discriminator**  $\mathbb{E}_x[(D(x) - 1)^2] + \mathbb{E}_z[(D(G(z)) - 0)^2]$

**Generator**  $\mathbb{E}_z[(D(G(z)) - 1)^2]$

Il secondo termine riguarda le immagini prodotte dal generatore che voglio che siano tutte etichettate come fake quindi 0 e ovviamente vado a prendere il valore atteso.

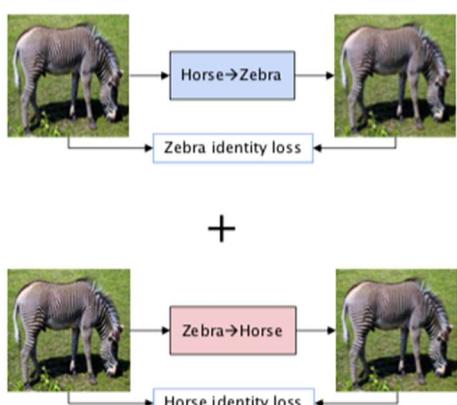
Col **generatore** faccio l'opposto, non ho dati reali e quindi il primo termine scompare e per quanto riguarda le immagini create si punta ad averle reali e quindi output 1.

## Identity loss



Si può usare una terza loss opzionale. È stata proposta per garantire il **preservarsi del colore dell'immagine**. Lavora anche lei a livello di pixel. → colori preservati il più possibile -> l'uscita del generatore deve essere il più sensata possibile.

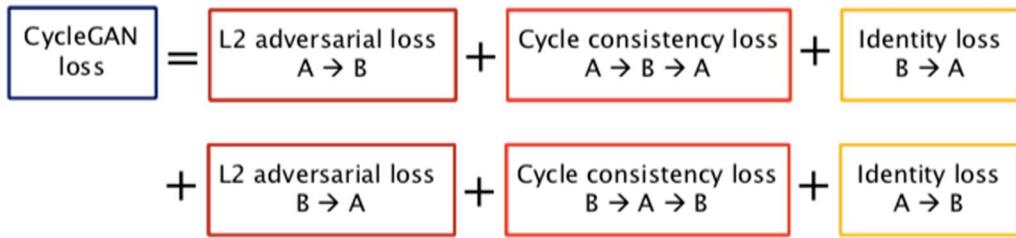
L'idea di base è che se io prendo l'immagine di un dominio e la passo per il generatore opposto, dal momento che l'input è dello stesso dominio dell'output, si dovrebbe ottenere la stessa immagine. Quindi, oltre a cercare di mantenere i contenuti si punta a mantenere il colore.



Trasferisce lo stile senza portare altre cose.

## CycleGan loss

$$\mathcal{L}_{cycleGAN} = \mathcal{L}_{adversarial} + \lambda_1 \cdot \mathcal{L}_{cycle\_consistency} + \lambda_2 \cdot \mathcal{L}_{identity}$$



Note aggiuntive sulle GAN:

Bisogna usarle in maniera opportuna:

- Ottime per trasferimento di stili (variazioni di texture/colori)
- Non vanno bene se serve anche fare delle variazioni geometriche dei contenuti all'interno dell'immagine.

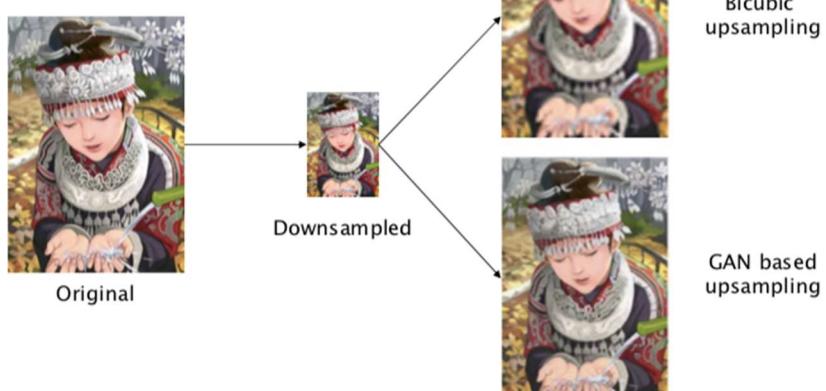


Cat to Dog translation using CycleGans (don't do it!!)

## Image super-resolution

Cerca di aumentare la risoluzione delle immagini in ingresso senza perdere qualità dei contenuti all'interno di essa.

- Increase image resolution creating «new» pixel contents



Bicubic  
upsampling

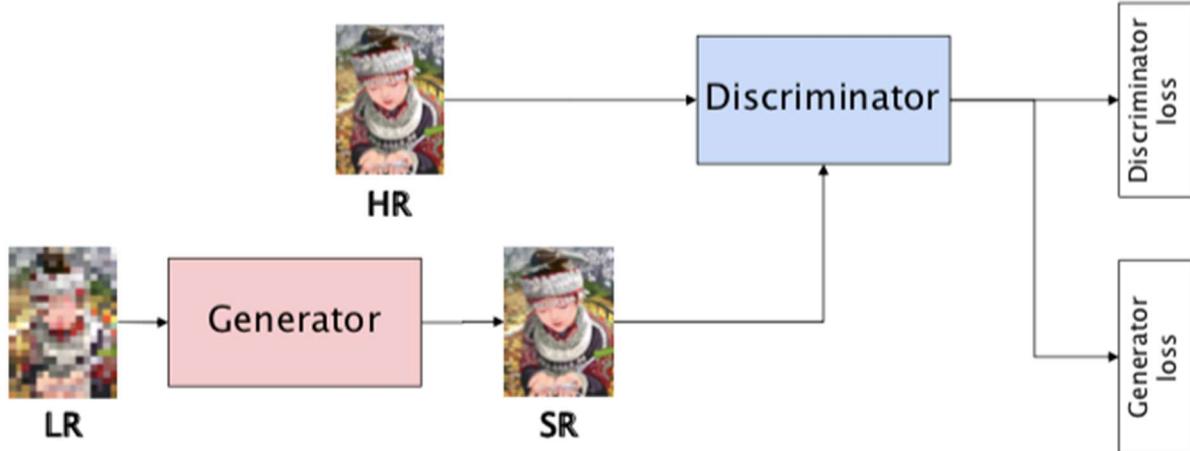
→ Creo nuovi pixel partendo da quelli di partenza, non sto migliorando la risoluzione dei dati perché non posso generare nuovi dati -> faccio interpolazioni tra ciò che ho -> sgrano informazione.

GAN based  
upsampling

→ Cerca di ingrandire l'immagine andando a migliorare la qualità, i dettagli.

## SRResNet (Super Resolution Residual Network)

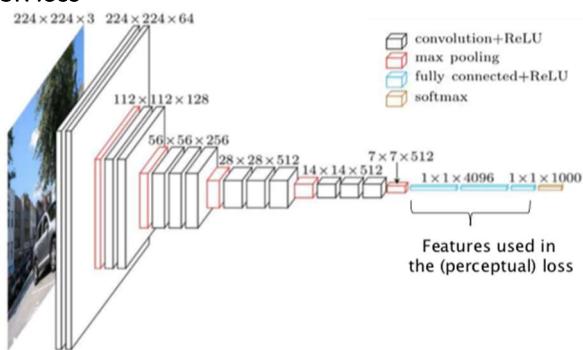
Prima soluzione presentata in letteratura per fare GAN base unsampling.



Il funzionamento è simile a quello delle classiche GAN dove si parte da una immagine in alta risoluzione HR che viene rimpicciolita perdendo qualità LR e questa viene data al generatore che ne fa un upsampling per cercare di ricreare l'immagine originale (ingrandendola) -> immagine a super-risoluzione che dovrebbe essere il più possibile uguale a quella di partenza.

- Mando immagine ad alta risoluzione di partenza e l'immagine SR al discriminatore che deve essere in grado di distinguere se è originale o SR.

SR loss



Il loss è basato sulle reti VGG (usato come feature extractor delle immagini). Le feature VGG sono abbastanza generali e si applicano facilmente ad altri applicativi.

## Descriminator loss

È la combinazione di elementi diversi

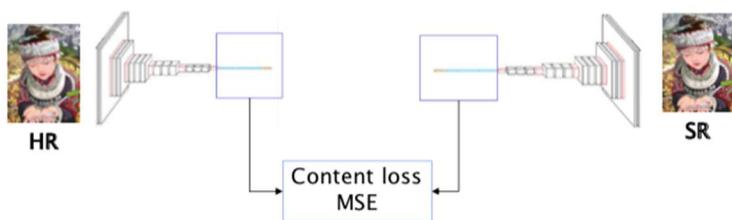
$$\mathbb{E}_{HR \sim p_{data}(HR)}[\log D(HR)] + \mathbb{E}_{LR \sim p_{\pi}(LR)}[\log(1 - D(G(LR)))]$$

## Generator loss

Abbiamo sicuramente bisogno di una loss adversarial qualsiasi a cui aggiungo una loss sulla perdita di contenuti.

- Adversarial loss + content loss
    - ◆ Adversarial loss is the standard log loss
    - ◆ Content loss aims at minimizing the perceptual loss between HR and SR → MSE between VGG-19 features (from FC layers)
    - ◆ VGG-19 features encode structural information

Questa loss va a confrontare i contenuti dell’immagine HR di partenza e di quella SR generata. Una possibile soluzione è quella di usare una *pixel loss* ma in questo caso è stata usata una **loss percettiva** e quindi quello che fanno è confrontare le features estratte da una VGG delle due immagini e confrontarle con l’MSE.



Viene fatto perché queste features sono ad alto livello e catturano le informazioni strutturali dell'immagine. In questo modo capisco se ho mantenuto la struttura corretta dell'immagine nel processo di incremento della risoluzione.

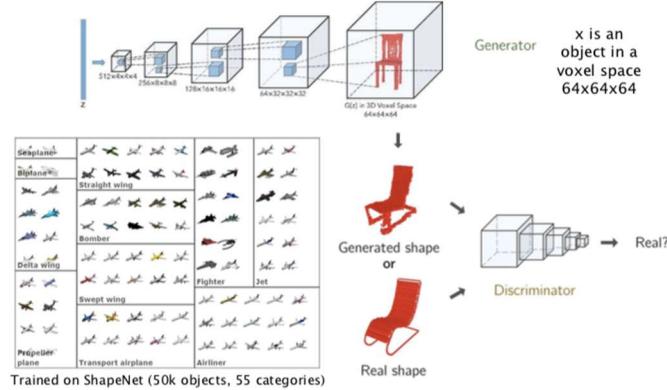
Faccio tutto ciò perché la pixel loss viene già usata dal discriminatore e quindi qui si cerca di irrobustire.

## 3D-GAN

Genera immagini 3D invece di immagini 2D



## 3D-GAN network



Generatore + discriminatore.

Quello che è diverso è il dominio su cui lavorano.

Gli oggetti sono rappresentati come oggetti a voxel (controparte 3D del pixel).

Utilizza lo stesso approccio delle GAN dove ovviamente si hanno convoluzioni 3D.

## Training tricks

C'è un forte squilibrio tra generatore e discriminatore perché generare oggetti 3D è molto più complicato (quindi discriminatore imparava molto più velocemente del generatore). Sono stati usati 3 trucchetti:

- Utilizzo learning rate diversi dove quello di D lo abbasso in modo che impari molto più lentamente
- Uso batch size molto grande
- Aggiorno discriminatore solo se l'accuracy sul batch è maggiore dell'80% -> penalizzo in quanto aggiorno solo generatore e poi di nuovo entrambi...

## From images to 3D

Si può condizionare il vettore di partenza con l'immagine scattata da cui si vuole ottenere l'oggetto 3D. Quindi si cerca di creare un mapping tra l'immagine 2D e il modello voxelizzato 3D.

Riprende l'approccio del VAE dove si mappa un'immagine in un punto in uno spazio latente.