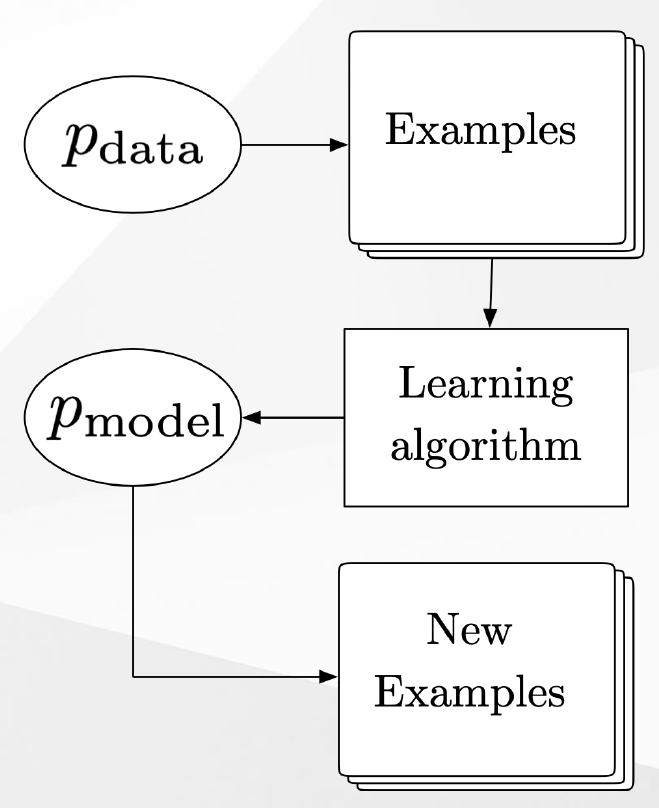
Generative Adversarial Networks

# Modello generativo

Un modello generativo è un qualsiasi sistema che prende in input un training set di esempi estratti in modo **indipendente e identicamente distribuito** da una distribuzione fissa P\_data e impara a rappresentare questa distribuzione.

In questo contesto si vuole estrarre nuovi esempi dalla distribuzione.



Una distribuzione P\_data genera un certo numero di esempi. Non è nota. Abbiamo solo l’accesso agli esempi. Si vuole costruire un modello della distribuzione dei dati e grazie a questo modello si vuole estrarre nuovi esempi da questa nuova distribuzione P\_model.

Un modello generativo, per definizione, modella l’intera distribuzione p(x) senza generare nuovi esempi.

Le GANs sono in genere usate per costruire nuovi esempi, non necessariamente devono rappresentare in modo esplicito la distribuzione P\_model ma possono essere usate per entrambi gli scopi: modellare l’intera distribuzione ed estrarre nuovi esempi.

# Modello generativo vs Modello discriminativo

**Modello generativo** → si vuole modellare una intera distribuzione p(x).

**Modello discriminativo** → si vuole modellare p(y|x).

# Perchè studiare i modelli generativi?

I modelli generativi sono molto importanti in diversi ambiti applicativi.

Nel campo della matematica e dell’ingegneria è importante trovare modelli in grado di rappresentare una **distribuzione con dimensionalità molto alta** (problema molto difficile).

Da ora in avanti si consideri un dataset di immagini. Si vuole apprendere p(x) dove x è un’immagine. Si vogliono generare nuove immagini x’.

* training set : insieme di immagini
* x : immagine
* x’ : nuova immagine

Questo tipo di **distribuzione** è **ad alta dimensionalità**: per ogni pixel dell’immagine si ha una variabile stocastica, si tratta di una distribuzione di centinaia/migliaia di dimensioni.

Nel campo del **Reinforcement Learning** (RL) i modelli generativi possono essere molto utili. L’applicazione più nota è far imparare ad un’agente (robot) a muoversi in un ambiente che non conosce. Il problema che si affronta è che in genere l’agente ha un certo numero di attuatori con cui interagisce con l’ambiente e ha un feedback dopo un certo numero di azioni che compie. E’ un problema particolarmente molto complesso in quanto è difficile capire dove è stato commesso l’errore.

Nei modelli generativi, spesso il sistema di RL deve ipotizzare dei possibili scenari per capire qual è l’**azione migliore da compiere, data la conoscenza** in quel momento. Se ho il modello generativo, conosco tutto di quel modello e posso porre qualsiasi tipo di query e non solo quelle per cui è stato addestrato in modo discriminativo.

I modelli generativi sono alla base di molti sistemi **Semi-Supervised Learning**. Il modello generativo permette di sfruttare meglio i dati non etichettati, a volte espandendo le etichette agli esempi vicini secondo il modello, altre volte perché il modello condivide una struttura nei dati e questa struttura può corrispondere a qualche struttura nelle etichette. Ciò semplifica la vita successivamente.

Altri esempi più applicativi.

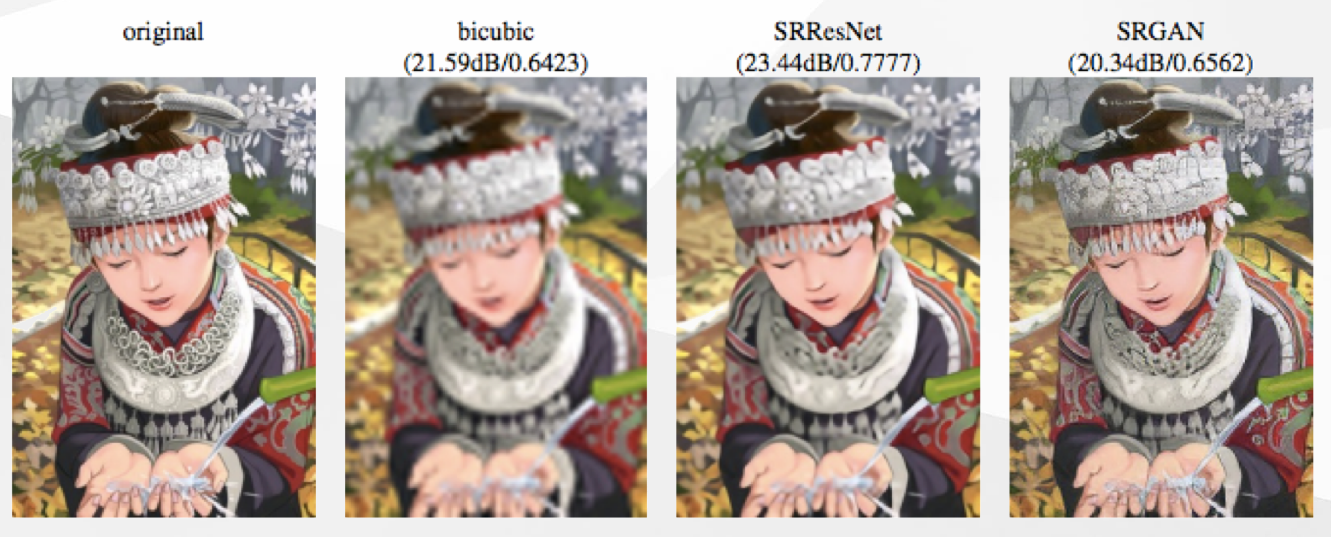
Le GANs sono molto adatte a **predire** **output multi-modali** (accennato in Representation Learning).



L’orecchio del modello MSE non viene ricostruito correttamente perché la forma dell’orecchio è una distribuzione multi-modale, esistono molte forme probabili. I modelli costruiti con MSE tendono a mediare tutte le possibili forme fino a farlo scomparire.

Le GANs (e i modelli generativi in generale) sono più adatte a modellare le varie mode dei dati e quindi riescono a selezionare da una sola moda una forma corretta.

Un altro esempio: le GANs nel task di **image super resolution**.



Si parte da un’immagine con più bassa risoluzione, si trasforma l’immagine ad una più alta risoluzione.

Le GANs costruiscono **spazi latenti** che hanno molte utilità.

<https://www.youtube.com/watch?v=9c4z6YsBGQ0>

In un modello tradizionale, la borsa viene deformata in un modo che ha poco a che fare con la realtà. Invece in un modello basato su GAN, si disegna la forma di una possibile borsa e la GAN ne immagina una possibile versione con quella versione.

Anche sulle scarpe: ciò che imparano è una rappresentazione latente che può essere manipolata: cambiando la rappresentazione latente si ottengono oggetti diversi.

Per fare produzione artistica: generazione del prato, di una montagna, cielo, neve.

<https://www.youtube.com/watch?v=FDELBFSeqQs>

Disegnando sul viso, cambiano i connotati della persona. Persone generate dalla GAN, manipolando le rappresentazioni.

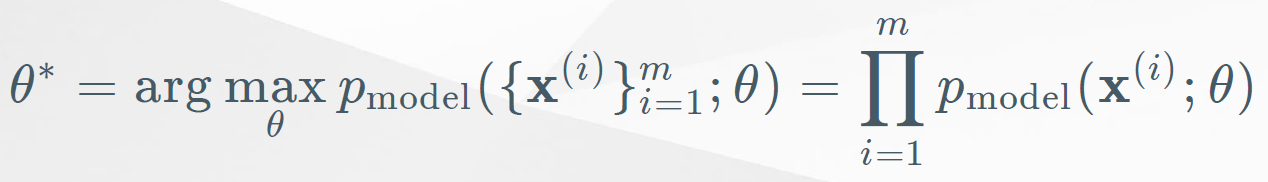
# Confronto tra GANs e altri modelli generativi

Vantaggi e svantaggi delle GANs rispetto ad altri modelli generativi.

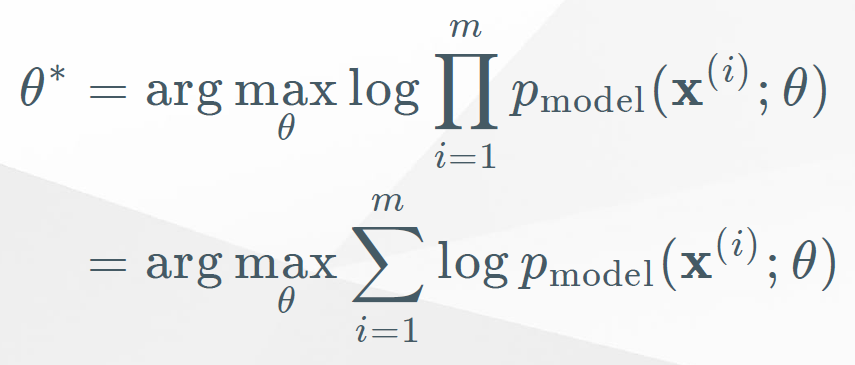
# Maximum Likelihood models

Si prendono in considerazione i **modelli basati sulla Maximum Likelihood**. E’ più facile considerare questo insieme di modelli perché **ottimizzano lo stesso tipo di loss** (con stessa idea dietro) ed è più semplice per fare confronti, altrimenti la complessità dei modelli diventa molto grande.

La **maximum likelihood** è un modo per trovare i parametri per un modello di apprendimento ed è basato sull’idea di **massimizzare la verosimiglianza** del dataset sotto la condizione che i parametri sono quelli che stiamo osservando in questo momento. Per ogni parametro scelto, si può calcolare la verosimiglianza del dataset quando la distribuzione segue questi parametri e prendere la distribuzione che massimizza questa funzione.



Essendo gli esempi indipendenti e identicamente distribuiti, la probabilità dell’intero dataset è uguale al prodotto delle probabilità dei singoli esempi. Quando si calcola la maximum likelihood, nella maggior parte delle situazioni, non si lavora con la formula appena descritta perché questo tipo di prodotto ha problemi numerici forti, ma si lavora in log space.



Si considera il logaritmo del prodotto. Siamo interessati all’insieme di parametri theta che massimizzano il log del prodotto. Il logaritmo è una funzione monotona crescente: il theta che massimizza il prodotto è lo stesso che massimizza il logaritmo del prodotto.

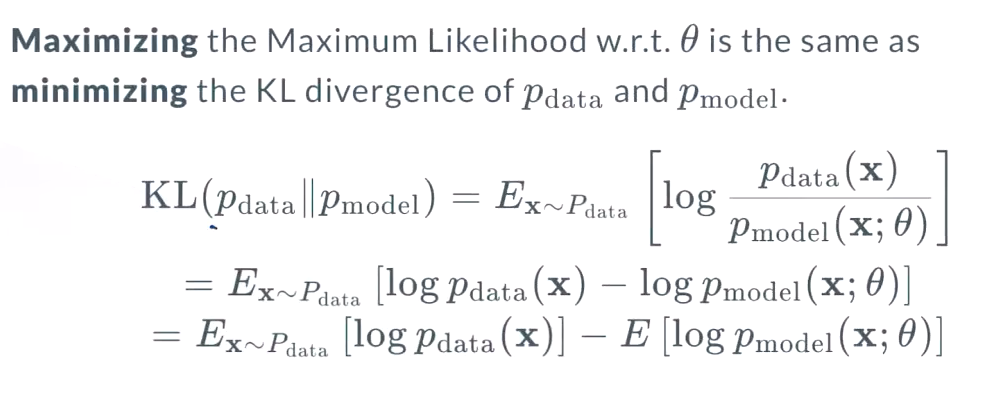
Il vantaggio dell’uso del log: il log del prodotto è la somma dei logaritmi.

Molti modelli basati sulla maximum likelihood anziché usare questa formulazione, fanno uso della **divergenza di Kullback-Leibler**.

# Divergenza di Kullback-Leibler

Si tratta di una divergenza fra due distribuzioni, ovvero una misura della “distanza” fra due distribuzioni. Non si chiama “distanza” perché non è simmetrica.

Divergenza di Kullback-Leibler tra due distribuzioni p\_data e p\_model



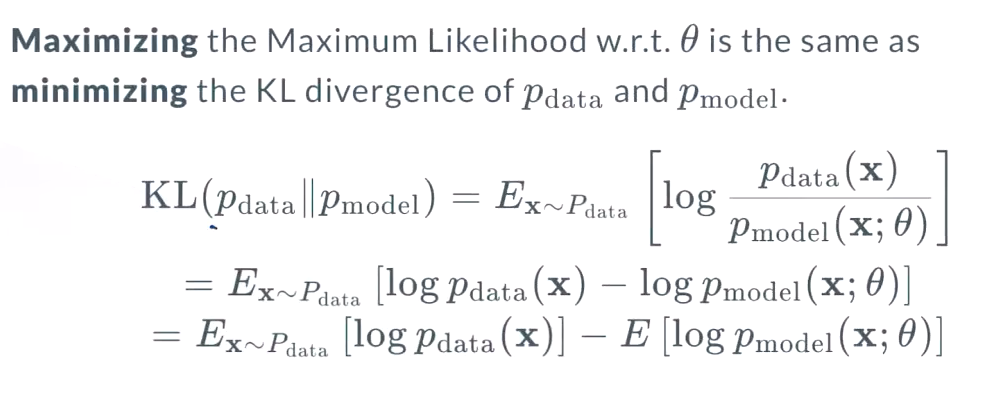
Il valore atteso preso secondo la distribuzione P\_data del logaritmo del rapporto tra le due distribuzioni P\_data e P\_model.

Il valore atteso è preso estraendo un numero di esempi dalla distribuzione P\_data. Per ognuno di questi esempi, si calcola il log del rapporto della probabilità dell’esempio secondo il primo modello P\_data(x) e la probabilità dell’esempio secondo il secondo modello P\_model(x;theta).

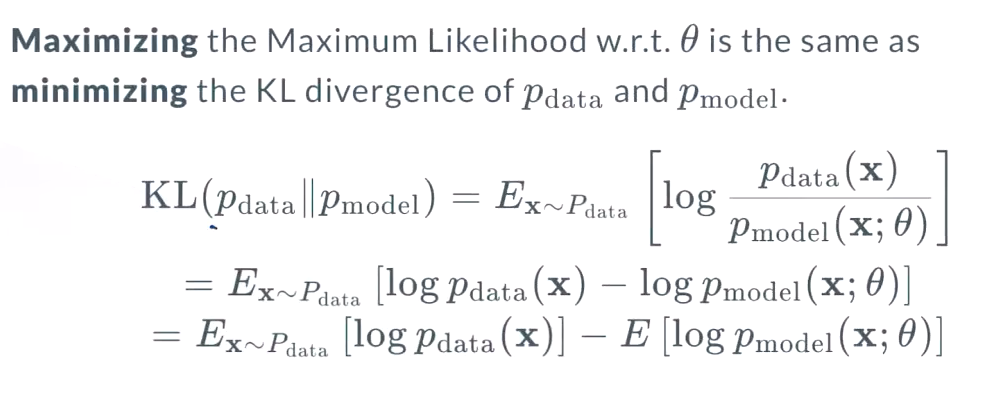
Si noti che questa formulazione può essere riscritta come differenza dei logaritmi. Il valore atteso è un operatore lineare, si può separare la somma (ultimo passo).

Nella maximum likelihood si massimizza la verosimiglianza dei dati.

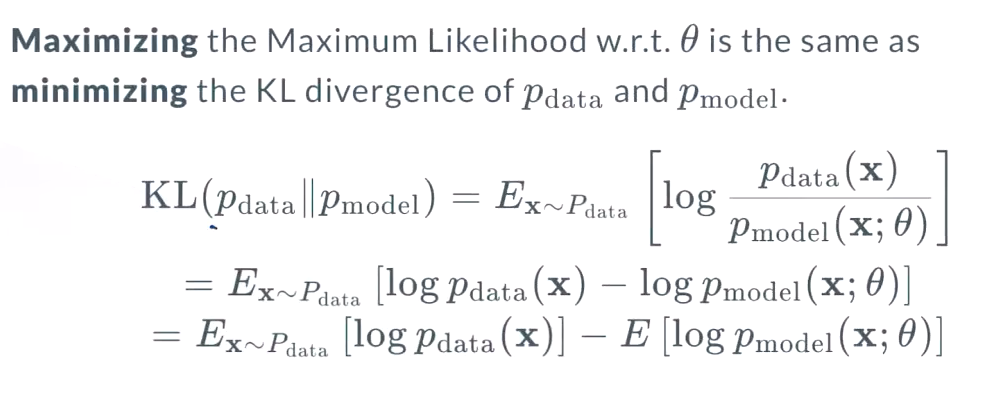
Nel momento in cui si massimizza la verosimiglianza, considerando l’ultima espressione della divergenza KL



il primo addendo E [ log P\_data(x)] non dipende dai parametri theta che stiamo osservando per fare la massimizzazione.



Ci si può concentrare sulla seconda parte della formula che corrisponde alla maximum likelihood dei dati secondo il modello.



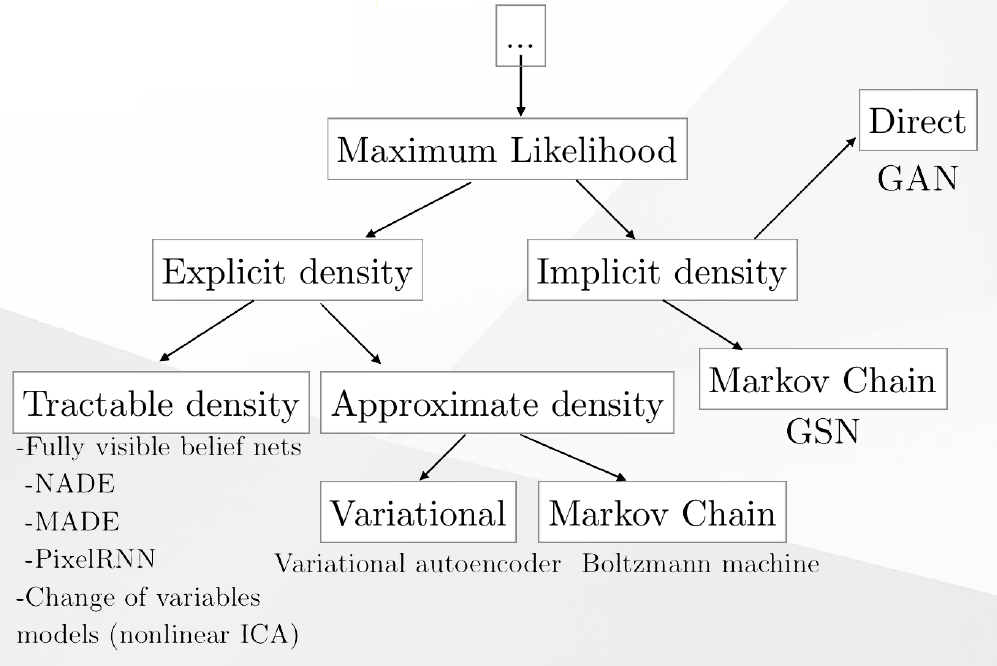
Essendoci un meno davanti, massimizzare la maximum likelihood corrisponde a minimizzare la curva di divergenza KL.

**Esiste una relazione molto forte tra Maximum Likelihood e la divergenza KL: massimizzando la maximum likelihood, si minimizza la divergenza KL**.

Una tassonomia dei modelli generativi profondi

Si vuole confrontare le GANs con gli altri modelli generativi.

Nella gerarchia ci sono … perché esistono anche altri modelli generativi che non consideriamo perché non fanno ottimizzazione della **maximum likelihood**.



Fra i modelli generativi che ottimizzano la maximum likelihood, si distinguono modelli che modellano in modo esplicito la densità (Explicit Density) e modelli che modellano in modo implicito la densità (Implicit Density).

**Explicit Density**: modelli che modellano in modo esplicito la densità:

* **Tractable density**: modelli che utilizzano la densità probabilistica e rinunciano a priori a considerare le densità che non sono trattabili
  + Fully Visible Belief Nets
  + Nonlinear Independent Component Analysis (ICA)
* **Approximate density**: modelli in cui la densità è approssimata.
  + **Variational**
    - Variational auto-encoders
  + **Markov Chain**
    - Boltzmann Machine

Dovendo modellare in modo esplicito la densità abbiamo due opzioni: se la densità è trattabile allora si può gestire oppure bisogna approssimarla in qualche modo.

Fra queste approssimazioni: compromessi quando si guarda un approccio variazionale (Variational) oppure compromessi dovuti all’utilizzo di una Markov Chain (Markov Chain).

**Implicit Density**: modelli c modellare in modo esplicito la densità. La utilizzano solo in modo implicito.

* **Markov Chain**: modelli basati su Markov Chain
  + Generative Stochastic Networks (GSN)
* **Direct**: modelli che modellano in modo implicito la densità. La generazione dei samples avviene in maniera diretta in un singolo step.
  + Generative Adversarial Networks (GANs)

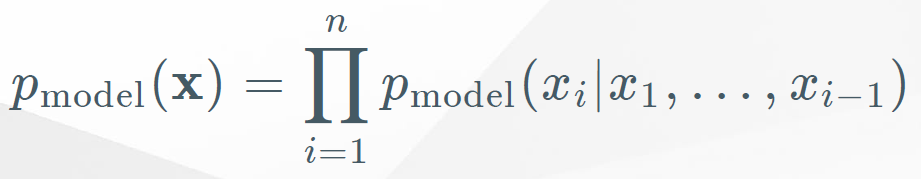
Modelli generativi di densità esplicita

# Fully Visible Belief Nets

I modelli generativi **Fully Visible Belief Nets** trattano solo distribuzioni che possono essere modellate in modo esplicito e sono trattabili.

Per modellare questi oggetti bisogna far ricorso alla **regola della catena (chain rule)** della probabilità.

Per quanto riguarda l’esempio x, la probabilità dell’esempio x può essere calcolata con la seguente formula.



Sostanzialmente si applica n volte la regola della catena (chain rule).

e così via fino ad ottenere la produttoria.

Fully Visible Belief Nets sono alla base di modelli generativi sofisticati proposti dal team di DeepMind, gruppo di ricerca che lavora per Google, che ha costruito su questo una rete famosa WaveNet, una rete per generare l’audio, in particolare per il parlato.

Il problema di questi modelli è che questa formula è **costosa da calcolare**. Quando il numero di attributi cresce nelle migliaia.

Per generare un nuovo sample bisogna impiegare **O(n)**.

**Questi modelli sono ottimi dal punto di vista della qualità del risultato, ma sono troppo lenti.** Per generare pochi secondi di audio, servono minuti di computazione. Ogni volta che si vuole generare un nuovo esempio, si dovrà eseguire nuovamente il modello. Non può essere fatto in parallelo.

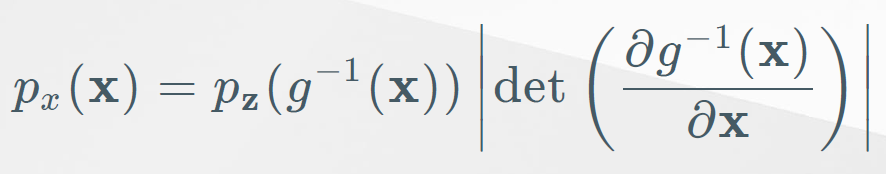
# Nonlinear Independent Component Analysis (ICA)

|  |
| --- |
| Funzione di massa per variabili casuali discrete  <http://progettomatematica.dm.unibo.it/Prob2/5funzionedimassa.html>  Funzione di densità per variabili casuali continue  <http://progettomatematica.dm.unibo.it/Prob2/6funzionedidens.html> |

L’idea per gestire la complessità è molto diversa dal Fully Visible Belief Nets. E’ simile all’idea che c’è dietro al Kernel e nella Representation Learning.

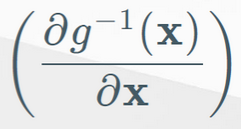
Nonlinear ICA si basano sul definire **trasformazioni non lineari continue** tra due spazi differenti.

Se esiste un vettore di variabili latenti z e una trasformazione g non lineare, differenziabile (derivabile), continua e invertibile tale che g(z) produca un campione dal modello nello spazio x, allora



Gestire nello spazio originale è molto difficile, allora si usa la funzione non lineare per mappare x in un nuovo spazio e poi si gestisce la probabilità in questo nuovo spazio.

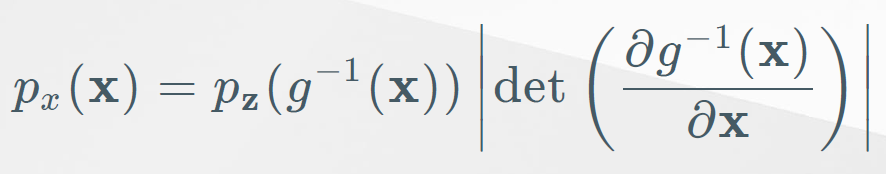
Questo termine rappresenta la matrice Jacobiana che contiene tutte le possibili derivate parziali dell’inversa della funzione g.



A cosa serve? Per capirlo bene bisogna sapere cos’è il determinante.

|  |
| --- |
| **Domanda: Qual è l’interpretazione geometrica del determinante di una matrice det(A)?**  E’ una misura di **quanto la matrice A distorce i volumi dello spazio su cui è applicata.** Di conseguenza, permette di sapere anche quando la matrice è invertibile.  Se **det(A) = 0** allora la matrice è **invertibile**.    Idea: i e j sono i versori che generano lo spazio, quindi generano il quadrato giallo con volume=1 (Area, siamo in uno spazio bidimensionale).  Nel momento in cui si applica la matrice A al volume:   * quando si applica la matrice al vettore j si ottiene il vettore Aj * quando si applica la matrice al vettore i si ottiene il vettore Ai   Si traccia nel grafico la somma dei vettori Ai e Aj e si proiettano tutti i punti del volume nell’area arancione.  Il determinante indica il rapporto tra l’area arancione e l’area gialla.    Proprietà vera per qualsiasi volume: presa una qualsiasi forma, applicando la matrice A, si ottiene un’altra forma. Il modo in cui è cambiato il volume della nuova forma rispetto alla precedente è indicato dal determinante di A.  **Esempio**    Il parallelogramma indicato in verde/rosso corrisponde al determinante di A.    Il determinante corrisponde per quanto devo moltiplicare il volume iniziale per ottenere il volume dopo l’applicazione della matrice A. |

Tornando alla domanda di prima:



A cosa serve il determinante nella formula del calcolo di probabilità di x?

E’ una conseguenza di **come cambiano le probabilità quando si applica una trasformazione** alle probabilità stesse, si sta parlando di **densità probabilistica calcolata su variabili nel continuo**.

Quando si parla di densità che ha a che fare con il continuo, la probabilità di un singolo punto è sempre infinitesimale.

Potrei avere tutta la massa di probabilità che si vuole, ma la sua probabilità sarà sempre 0.

Per fare in modo che non abbia probabilità pari a 0, **si considera un volume in cui è incluso il punto di interesse e si calcola la probabilità del volume**.

Quando devo calcolare la probabilità p(z), viene calcolato:

p(z) dz.

Quindi **in genere non si calcola la probabilità di un singolo punto, ma la probabilità di un volume che contiene quel punto**.

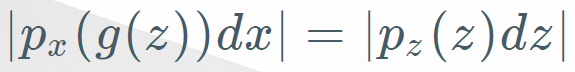
Se x = g(z), g(z) trasforma tutto il volume p(z)dz in un nuovo volume. Quindi nel calcolare p(z)dz bisogna tenere in considerazione la distorsione causata da g al volume dx, poiché altrimenti l'integrale sull'intero spazio non ottengo più 1.

Se prima avevo tutti 1 e poi l’ho trasformato, e il volume trasformato è più grande di 1 si ottiene che l’integrale è diverso da 1.

**Perché il determinante della matrice Jacobiana?**

Anziché considerare un vettore di variabili reali, è più semplice considerare il caso di una singola variabile appartenente ai numeri reali.

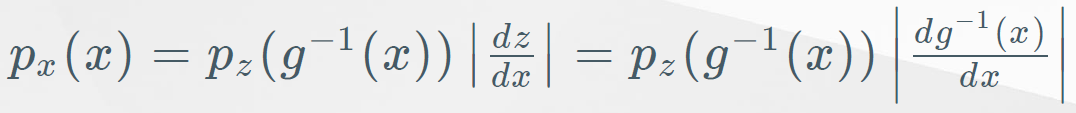
Si impone il seguente vincolo:



allora

La probabilità è sempre positiva

divido entrambe le parti per |dx| e ottengo



E’ la stessa formula di prima con la sola differenza che anziché avere il determinante, c’è il modulo.

Il modulo indica quanto si deforma una linea retta in quanto si sta considerando una singola variabile.

Grazie al fatto che abbiamo capito che il determinante è una misura di quanto la matrice deforma il volume. Quando si parla di volume, al posto del modulo si prende il determinante del Jacobiano.

Quindi il modello generativo Nonlinear Independent Component Analysis cerca di risolvere il problema del fatto che sta lavorando con una distribuzione complicata, usando una **trasformazione** che semplifica il lavoro su questa distribuzione.

**Il determinante è importante perché tiene conto della deformazione dello spazio della probabilità**, **nel momento in cui si applica la trasformazione.**

Per come sono state costruite le cose, implica anche che la **funzione g deve essere invertibile**.

L’invertibilità di g implica che se lo spazio in cui vivono x e z siano della stessa dimensione, questo limita di molto le funzioni g che possono essere applicate.

Nei Kernel questa scelta è molto importante perché uno dei grossi vantaggi è che i Kernel mappano in spazi molto più grandi rispetto allo spazio originario, quindi riescono a semplificare molto di più il problema.

In questi modelli invece si trasforma lo spazio contando sulla **non linearità della funzione g**.

I Nonlinear ICA impongono vincoli sulla scelta di g. In particolare **la restrizione di invertibilità impone che il numero di dimensione di x è uguale al numero di dimensioni di z**.

# Riassunto

Sono stati presentati due modelli generativi che modellano in modo esplicito la distribuzione. Questi modelli presentano due problemi.

I modelli Fully Visible Belief Nets riescono a **generare nuovi samples in modo molto lento** perché lavorano in modo sequenziale.

Le GANs invece riescono a generare nuovi samples in modo più veloce perché lavorano in modo parallelo.

Nei modelli Nonlinear ICA, i **vincoli posti sulla funzione g** sono abbastanza **onerosi e quindi non si ha** tutta la **flessibilità** che si ha sulle GANs che invece lavorano su uno spazio più ricco.

Sono modelli trattabili.

# Auto-encoders Variazionali

Esistono altri approcci che rinunciano alla trattabilità del modello: si considerano anche **distribuzioni** **intrattabili** in cui il calcolo di alcuni termini richiede troppo tempo e sono richiesti molti esempi.

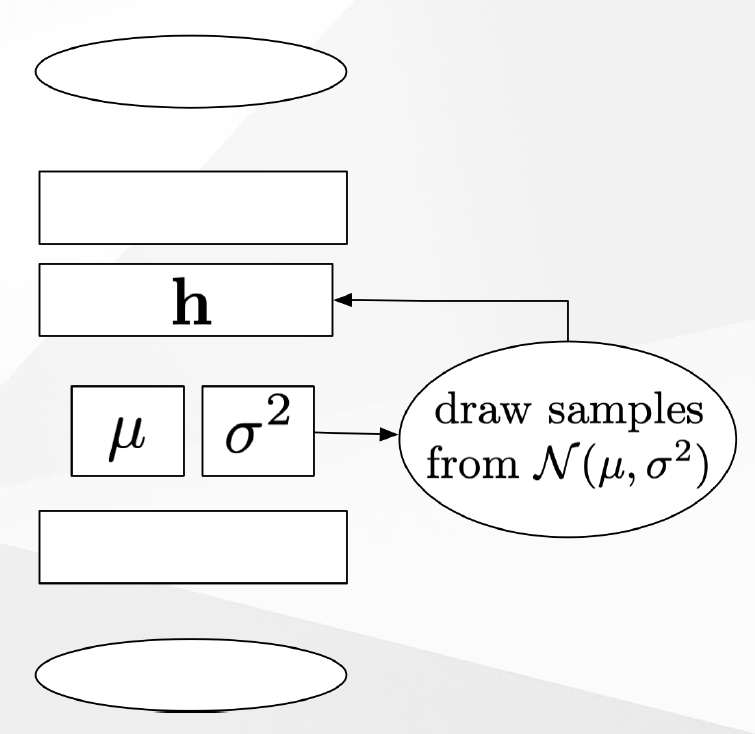
Per sopperire al fatto che sono intrattabili, si introduce qualche **approssimazione**.

I metodi di approssimazione variazionale sono una grossa classe di metodi originati nell’ambito di modelli grafici e applicati anche alle reti neurali.

Idea alla base dei **variational auto-encoders**: dato un modello non trattabile che non può essere ottimizzato, al posto di usare questo modello, si usa una versione trattabile che lo approssima. Se questa versione dà una probabilità di likelihood dell’esempio che è un lower bound della probabilità di likelihood che darebbe il modello a quello stesso esempio, allora si può ottimizzare il sistema massimizzando il lower bound e garantendo di ottenere almeno una certa prestazione rispetto al modello.



I variational auto-encoders sono tecniche più note applicate nelle reti neurali. Si tratta di un auto-encoder che ricostruisce un input.



La differenza rispetto agli auto-encoder classici è che, anziché avere una rappresentazione h costruita utilizzando la prima parte della rete e imponendo qualche vincolo, in questi modelli la rappresentazione h è un insieme di numeri estratti da una particolare distribuzione che è la distribuzione usata per approssimare la distribuzione che non è trattabile.

Di solito si usa una Gaussiana multi-variata. Quindi il livello subito dopo l’input apprende i parametri della Gaussiana (media e varianza).

Dopo aver appreso i parametri, si genera la rappresentazione h estraendo a caso dalla Gaussiana.

Dopodiché si ha il decoder che ricostruisce l’input a partire dalla rappresentazione h.

Questi modelli funzionano bene. Prima delle GAN, erano i modelli allo stato dell’arte per la generazione di samples, anche nel contesto delle immagini.

Però rispetto alle GANs hanno **performance meno convincenti**.

In particolare, quando l’approssimazione che si introduce è troppo cruda, la **distanza tra** **modello vero dei dati ed il modello approssimato** appreso fa sì che anche arrivando all’ottimo rispetto alla distribuzione L siamo comunque distanti rispetto alla vera distribuzione dei dati.

In pratica, questi auto-encoder danno buoni risultati ma la qualità degli esempi non sono quelli delle GANs.

Nell’esempio delle foto ad esempio danno risultati più sfocati rispetto alle GANs.

Rispetto alle Fully Visible Belief Networks, gli Auto-Encoders Variazionali sono più **difficili da addestrare**. Sfortunatamente anche le GANs sono difficili da addestrare.

Parte dei problemi di addestramento sono risolti dalle WGAN.

# Approssimazioni di Markov Chain

Un’altra idea per trattare distribuzioni intrattabili è usare qualche forma di **approssimazione stocastica**.

Il problema qui è trovare un modo efficiente per generare i dati da P\_model(x) quando la distribuzione non è trattabile.

I metodi Markov Chain generano dati mediante una **catena di Markov**.

Generare P(x1...xn) è troppo oneroso perché presenta troppe dimensioni, la distribuzione è troppo complicata, il fattore di normalizzazione è troppo complicato da calcolare.

Markov Chain trova un **operatore di transizione q(x’|x)** che estrae dalla distribuzione completa delle approssimazioni e, iterando questo processo, si giungerà ad una approssimazione molto buona rispetto alla distribuzione originaria.

Caso più semplice: per approssimare P(x1...xn), si mettono numeri a caso per x2...xn e si estrae solo x1. Si è in una situazione in cui x1 distribuzione che si sta cercando di modellare, però x2..xn sono presi a caso.

Il prossimo punto lo si ottiene dicendo che:

* x1 è l’ultimo appena estratto
* x2 viene estratto
* x3...xn scelti a caso

Ripetendo questa operazione, si estraggono tutti i punti uno ad uno e iterando più volte questo processo, si ottiene un vettore x1...xn che sembra sia estratto dalla distribuzione originale.

Questo tipo di approccio è molto lento. In particolare quando si usa questo approccio anche a **tempo di apprendimento** (quando si modella la distribuzione) rende le cose molto lente e impraticabili.

Se invece il modello può essere addestrato senza le Markov Chain, quindi si usa Markov Chain solo a **tempo di inferenza** funziona abbastanza bene ma rispetto alle GANs sono estremamente più lento perché le GANs generano i sample in un unico passaggio.

Modelli generativi densità implicita

# Generative Stochastic Networks

Al posto di modellare una certa distribuzione P(x1...xn) che è complicato, ci si dimentica del termine P(x1...xn) e si usano le Markov Chain: se si ottiene in maniera ottimale l’operatore di transizione, è inutile imparare P(x1...xn) perché l’operatore di transazione è tutto ciò che serve per lavorare nella distribuzione originale.

Problema: la generazione degli esempi è molto lenta.

# Generative Adversarial Networks

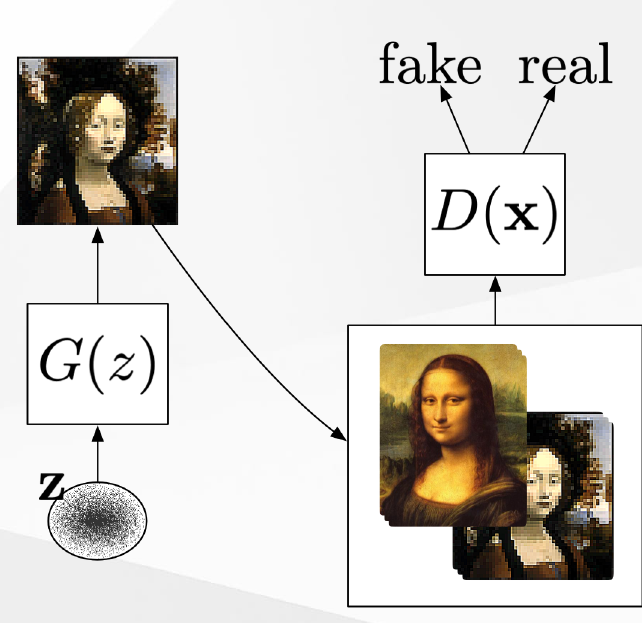
## **Vantaggi rispetto a tutti gli altri modelli generativi**

Le GANs modellano la distribuzione solo implicitamente e generano gli esempi in modo diretto, in un singolo step.

1. Si modella la distribuzione in modo implicito: non si riesce a modellare P(x1...xn) in modo esplicita, non sappiamo esattamente qual è questa distribuzione. Ma si riesce comunque a lavorarci perché si riesce ad estrarre da quella distribuzione utilizzando un metodo molto veloce che richiede un singolo passo.
2. Gli esempi possono essere generati in parallelo: le componenti del singolo esempio sono generate in parallelo.
3. La funzione che genera gli esempi ha poche restrizioni, quasi nessuna
4. Non abbiamo bisogno di usare Markov Chain per fare approssimazioni.
5. Non abbiamo bisogno di nessun limite variazionale: abbiamo più libertà rispetto ai metodi variazionali per scegliere i modelli.
6. Dal punto di vista dei risultati, test empirici soggettivi (si guarda l’immagine e si dice se l’immagine è realistica oppure no), si ha una unanimità nel considerare le GAN superiori ai modelli riguardo questo aspetto.

Come lavorano le GANs?

E’ un regime di addestramento molto diverso da ciò che esisteva prima.



Le GAN sono modellate come un gioco tra due attori:

* il **generatore** → funzione G(z)
* il **discriminatore →** funzione D(x)

Il **generatore** prende in input un vettore che solitamente è un vettore di numeri casuali (vettore generato casualmente) e usa questo vettore come spunto per generare una nuova immagine.

La nuova immagine viene data in input al **discriminatore** insieme ad altre immagini reali. Lo scopo del discriminatore è dire quale delle immagini è stata generata casualmente oppure arriva dalla realtà, è vera.

Si noti la **difficoltà** cui è posto il generatore: **il generatore deve produrre immagini senza aver mai visto un’immagine** in vita sua. Ciò che vede è “**rumore bianco**” (vettore generato casualmente), deve **capire com’è fatto il mondo reale e com’è fatta l’immagine che deve generare soltanto dal feedback restituito dal discriminatore**.

Inizialmente sarà rumore bianco e il discriminatore riconosce che non è immagine reale. Dopodiché genera qualcos’altro e il discriminatore riconosce nuovamente che è fake, e così via…

Osservando con che confidenza il discriminatore dice che l’oggetto è fake o reale, il generatore deve aggiornare il proprio modello per riuscire a costruire immagini il più realistiche possibili.

**G e D sono funzioni differenziabili** (potrebbero essere due reti neurali). Devono essere differenziabili perché si vuole addestrarle usando la **discesa del gradiente**.

* La funzione G
  + prende in input un vettore z
  + è definita in termini di parametri che rappresentano i pesi della rete neurale
  + restituisce in output un valore x’ dallo stesso spazio di x (se x è un’immagine, allora anche x’ sarà un’immagine)
* La funzione D
  + prende in input x
  + è definita in termini di parametri che rappresentano i pesi della rete neurale
  + restituisce in output un valore . Si tratta di un numero compreso tra 0 e 1 che indica la confidenza.
    - più vicino a 0 allora è fake
    - più vicino a 1 allora è real
  + è molto importante che l’output sia un valore compreso tra 0 e 1 per la continuità della loss function (quando l’output sarà l’input della loss function).

Il generatore e il discriminatore hanno funzioni di costo (loss function) definite in termini di e .

Di seguito sono definite le funzioni di costo (loss function).

Per ottimizzare queste due loss function, le due funzioni **non possono agire sui parametri dell’altra funzione**.

Nel momento in cui si addestra il generatore e si ottimizza J(G), si può intervenire solo su .

Nel momento in cui si addestra il discriminatore e si ottimizza J(D), si può intervenire solo su .

Nel **caso più semplice** si è di fronte ad un gioco fra queste due componenti a “**somma nulla**”:

oppure

In un gioco a somma nulla, ogni volta che una delle due componenti ha un vantaggio, questo si tramuta in un svantaggio di eguale entità per l’altra componente.

E’ il caso del gioco degli scacchi: le funzioni che minimizzo sono tali per cui se guadagno qualcosa nel fare una mossa, l’avversario sta perdendo di una eguale entità.

I giochi a somma nulla sono stati analizzati dal famoso matematico John Nash che ha dimostrato che tutti questi giochi hanno soluzione ottima in un punto detto **punto di equilibrio di Nash**: è un punto nello spazio tale per cui è un minimo locale secondo J(D) ed è un minimo locale anche secondo J(G).

* minimo locale di J(D) rispetto a
* minimo locale di J(G) rispetto a

Riassumendo, esistono due giocatori che ottimizzano due funzioni di loss in contrapposizione l’una con l’altra. Si ottiene una soluzione quando il primo giocatore non può fare nient’altro agendo sui propri parametri per migliorare la propria situazione e il secondo giocatore non può fare nient’altro agendo sui propri parametri per migliorare la propria situazione.

# Addestramento delle GANs

In genere per addestrare le GANs si utilizza la **discesa del gradiente stocastica SGD** simultaneamente sia su J(G) che su J(D).

* si estraggono due mini-batches (uno dai dati reali e uno dai dati generati)
* si valutano le due loss J(G) e J(D), quindi si aggiorna il vettore usando i gradienti che arrivano da J(D) e si aggiorna il vettore usando i gradienti che arrivano da J(G).

I dati generati permettono di costruire un batch di esempi che sono forniti al discriminatore. Il discriminatore restituirà delle risposte.

In base alle risposte date dal discriminatore, si calcolano le due funzioni di costo J(D) e J(G) e si possono far fluire i gradienti dal discriminatore fino al generatore.

Il generatore viene aggiornato in funzione delle risposte date dal discriminatore.

A volte si decide di **addestrare di più il discriminatore rispetto al generatore** perché se il discriminatore è molto scarso, da pochissima informazione al generatore. Nel tentativo di bilanciare i due oggetti, si tende a dare più attenzione al discriminatore per fare in modo di avere un feedback più pulito per il generatore.

In questo gioco, sia il generatore che il discriminatore stanno imparando. Quando inizia il gioco, anche il discriminatore non sa distinguere tra immagine reale e falsa, quindi farà degli errori. La funzione di loss è usata dal discriminatore per migliorare il task di distinguere tra immagini vere e false.

L’informazione poi passa alla funzione di loss del generatore per capire quanto è stato bravo a generare l’immagine per migliorare nel suo task.

Entrambe le funzioni di costo del generatore e discriminatore sono definite in funzione di entrambi i due parametri .

Nel momento in cui vengono ottimizzate le funzioni di costo, la funzione di costo del discriminatore è usata dal discriminatore usando solo sui suoi parametri theta D. Analogamente per la funzione costo del generatore.

# Funzione costo del discriminatore

La funzione costo del discriminatore è molto importante perché, fino a poco tempo fa, quasi tutte le varianti GANs usavano la stessa identica loss del discriminatore, apportando piccole modifiche solo alla loss del generatore.

Un grosso avanzamento dello sviluppo delle GANs è stato quando si è deciso di modificare la funzione loss del discriminatore.

La funzione costo del discriminatore è definita:



Nella prima componente, l’output del discriminatore D(x) è calcolato su x, quindi x è un esempio reale (non generato): il discriminatore funziona bene quando il valore di D(x) è vicino a 1 → il logaritmo è vicino a 0. Non si paga pegno per ciò che si sta predicendo.

Si è penalizzati di ½ solo quando si etichetta un esempio reale come esempio non reale.

Nella seconda componente, l’output del discriminatore D(G(x)) è calcolato su G(x), ovvero un esempio generato dal generatore. L’output desiderato per il discriminatore dovrebbe essere 0 e l’input del logaritmo è 1 - D(G(x)).

Quando il discriminatore pensa che G(x) sia un esempio reale, predice un valore vicino a 1, quindi 1 - 1 = 0 → log 0 tende a +inf.

Si paga pegno quando il discriminatore sostiene che l’esempio generato è reale → D(G(x)) = 1.

Questa funzione costo può sembrare complicata ma corrisponde alla cross-entropy applicata a questo particolare problema.

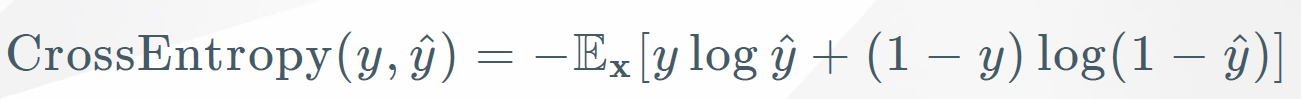
|  |
| --- |
| **CROSS ENTROPY**  La cross-entropy è una misura nata nell’ambito della teoria dell’informazione. Misura quanti bit di informazione in media dobbiamo trasmettere quando il codice che stiamo inviando è ottimizzato per una distribuzione q e invece i dati seguono una distribuzione p.    L’entropia di un canale è il numero medio di bit che bisogna trasmettere sul canale per trasportare una certa informazione.  L’informazione in genere è codificata per ottimizzare l’invio dei dati secondo la distribuzione dei dati stessi.  Esempio:  Si vogliono inviare i messaggi A,B,C,D.   * A ha una probabilità di essere inviato del 95% * B,C,D molto improbabili   allora si vuole codificare i messaggi usando pochi bit:   * A = 1 * B = 01 * C = 001 * D = 0001   Le stringhe binarie dei messaggi B,C,D sono più lunghe ma sono molto improbabili allora si trasmette in media la codifica del valore più frequente (nel seguente caso 1 bit).  Nel momento in cui si vuole inviare i messaggi ma si ottimizza il codice in funzione di un’altra distribuzione q, ci si può interrogare sul numero medio di bit necessari per trasmettere l’informazione considerando la nuova distribuzione q.  p distribuzione dei dati  q nuova distribuzione  H(p) : numero di bit da trasmettere quando il codice è costruito sulla distribuzione p e i dati sono inviati secondo la distribuzione p.  H(p,q): numero di bit da trasmettere quando i dati sono distribuiti secondo la distribuzione p ma il codice è ottimizzato secondo la distribuzione q.  La media di una funzione f(x) è calcolata nel seguente modo:  Se la distribuzione è uniforme, allora p(x)=1/N, quindi la media E corrisponde alla media aritmetica.  **Nota bene:** La cross entropy misura la lunghezza media del messaggio da trasmettere che è più lunga rispetto alla lunghezza del messaggio ottimale.  Di quanto è più lunga la lunghezza media?  dove H(p) è l’entropia secondo la distribuzione p, ovvero la lunghezza media ottimale.  Questa formula è detta **entropia relativa** (coincide con la **Divergenza di Kullback-Leibler**). |

**La cross entropy è utilizzata in apprendimento automatico come funzione di loss** perché nel caso ideale, la cross-entropy si può ridurre all’entropia. Ciò avviene quando la distribuzione p diventa identica alla distribuzione q.

Se si sta modellando la distribuzione p utilizzando come approssimazione la distribuzione q, si vuole diminuire la cross entropy fino a quando non arrivo a H(p).

La **minimizzazione della cross entropy** è un modo per ottimizzare il modello per modellare perfettamente la distribuzione p.

**Cross-entropy in machine learning nel caso della classificazione binaria (usata dalle GANs)**.



y → distribuzione p

1-y → distribuzione p

y^ → distribuzione q

Nel caso di un esempio reale:

* y=1 : esempio reale
* y^=D(x) : l’argomento della funzione D è esempio x reale
* la seconda parte della cross-entropy svanisce perché (1-y) con y=1 si annulla

Nel caso di un esempio generato (falso, non reale):

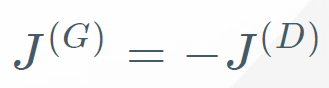
* y=0 : esempio generato
* y^=D(G(x)) : l’argomento della funzione D è l’esempio generato dal generatore G, ovvero G(z). Il generatore parte dal rumore z.
* la prima parte della cross-entropy si annulla perché y\*log... con y=0 è 0.

Dalle osservazioni fatte, si ottiene la funzione costo del discriminatore



# Funzione costo del generatore

Nella forma più semplice, la funzione costo del generatore è l’inverso della funzione costo del discriminatore.

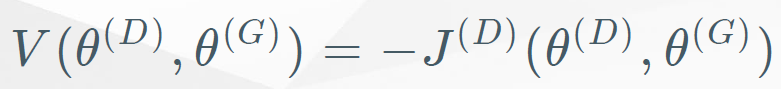


Questa è la situazione in cui si determina un gioco a somma nulla: ogni volta che si ha un guadagno per il generatore, questo corrisponde alla perdita per il discriminatore. Guadagno e perdita hanno stessa grandezza, sono di eguale entità.

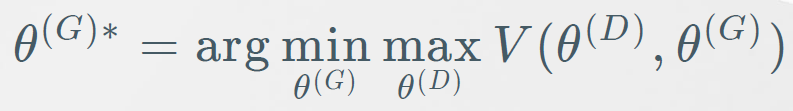
# Minimax

I giochi a somma nulla di solito vengono risolti definendo un **problema minimax**.

Si definisce una funzione V in funzione dei due parametri come la funzione costo del generatore (inversa della funzione costo del discriminatore).

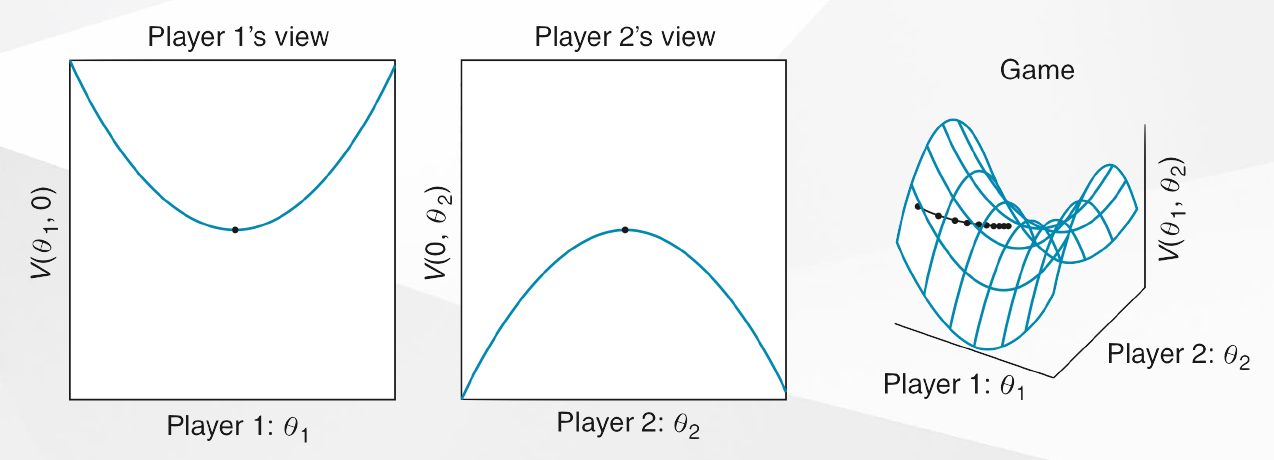


L’insieme di parametri ottimali per il generatore è l’argomento che risolve questa espressione:



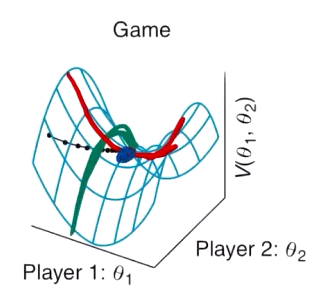
della funzione V, si trova il max secondo theta D e poi il minimo secondo theta G.

Si ottiene così il punto ottimale di detto **punto di equilibrio di Nash**.



Il primo giocatore ottimizza la funzione minimizzandola.

Il secondo giocatore ottimizza la “stessa” funzione massimizzandola (stessa funzione ma ottimizza una coordinata diversa della funzione).



In 3D, il giocatore 1 ottimizza la funzione verde. Il giocatore 2 ottimizza la funzione rossa.

Il punto minimo per entrambi è quello che determina l’equilibrio di Nash indicato con il colore blu.

# Vantaggi teorici di questa configurazione: Divergenza di Jensen-Shannon

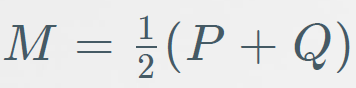
Questo tipo di configurazione ha il vantaggio che è relativamente facile da analizzare dal punto di vista teorico. Corrisponde a un gioco a somma zero e si può realizzare utilizzando le tecniche della teoria dei giochi.

Si può notare che questo gioco corrisponde a **minimizzare la divergenza di Jensen-Shannon** tra la distribuzione dei dati (P) e la distribuzione del modello (Q).

**Divergenza di Jensen-Shannon**



KL: divergenza di Kullback-Leibler



Il gioco converge ad un equilibrio che minimizza la divergenza di Jensen-Shannon se è possibile ottimizzare i due giocatori nello spazio funzionale in cui vivono. Nella pratica non può essere fatto: questo corrisponde ad utilizzare tecniche dell’analisi funzionale dove lo spazio in cui vivono gli oggetti che si analizzano non sono punti a 1-N dimensioni ma sono spazi di funzioni. Quando si risolvono i problemi reali non si riesce a fare, quindi non si ottimizzano gli oggetti nello spazio funzionale ma si ottimizzano gli oggetti nello spazio dei loro parametri (si ottimizzano i parametri e non direttamente le funzioni).

# JSD è simmetrica

KL(P||Q) è notoriamente non simmetrica, ovvero KL(P||Q) è diverso da KL(Q||P).

La divergenza JSD è definita in funzione della divergenza KL.

La divergenza Jensen-Shannon JSD(P||Q) è non simmetrica?

Risposta: No. La JSD è simmetrica.

Calcolando

ed essendo la somma simmetrica, allora

Quindi la JSD è simmetrica.

# La derivata della loss è prossima a 0

Si supponga che

e che il discriminatore sia diventato molto bravo a discriminare con alta confidenza tra immagini reali e false.

Cosa ci aspettiamo a proposito del gradiente della loss usata dal generatore? E’ un valore alto oppure un valore basso?

Risposta: La derivata della loss è un numero vicino a 0.

Nel caso in cui il discriminatore sia diventato molto bravo, tutte le volte che si genera un’immagine falsa il discriminatore riesce a capire che è falsa. Questo è un feedback negativo per il generatore, quindi ci si aspetta che la funzione di loss del generatore sia molto alto.

Purtroppo, la funzione loss è stata definita solo in termini del discriminatore. Quando il discriminatore riesce a discriminare molto bene, in verità è arrivato vicino al punto di ottimalità.

La derivata della loss è un numero vicino a 0. Il valore della loss non è necessariamente verso 0, ma è necessariamente verso un punto del suo minimo.

La loss J(D) è molto vicina al punto del suo minimo, quindi avrà una derivata molto piccola.

Avendo una derivata molto piccola, allora anche la derivata della funzione di loss del generatore sarà molto piccola in quanto: .

Quando il discriminatore riesce a discriminare molto bene, il generatore riceve dei gradienti che sono molto prossimi a 0 e non riuscirà più a migliorare.

Questo è uno dei problemi di questa versione delle GANs:

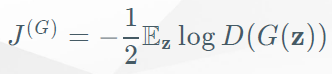
E’ semplice analizzarlo dal punto di vista teorico ma non funziona bene nella pratica, è difficile da addestrare.

# Heuristic, Non Saturating Game

Impostando il costo è buono per l’analisi teorica ma non funziona particolarmente bene nella pratica, è difficile da addestrare.

Nel gioco minimax il discriminatore minimizza la cross-entropy, ma il generatore la massimizza. Questo è uno svantaggio perché il gradiente della loss svanisce (**vanishing gradient**) quando il discriminatore rifiuta gli esempi del generatore con alta confidenza.

Per risolvere il problema del vanishing gradient, si può cambiare la formulazione del generatore per minimizzare un termine cross-entropy adatto alla sua visione del problema:



In questo nuovo gioco, **il generatore massimizza la probabilità che il discriminatore sbagli**.

E’ una loss di tipo **euristica**, non è motivata da ragioni teoriche, ma l’idea è di dare tutta l’informazione possibile al generatore perché riesca ad apprendere il più velocemente possibile.

Come si costruisce?

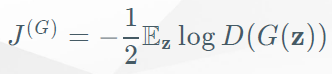
Si parte dalla funzione loss del discriminatore:



si prende la seconda componente

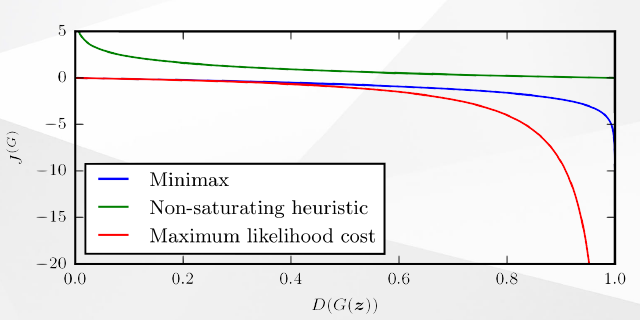


si rimuove “1-” (in quanto la formula è ottimizzata per il discriminatore: 1-D(G(z)) penalizza il discriminatore) e si ottiene la nuova funzione loss del generatore.



E’ analoga ma penalizza il generatore quando il discriminatore sbaglia a discriminare, quindi D(G(z)) = 0 per G(z) esempio generatore dal generatore.

Si consideri la minimax e la non-saturating heuristic.



Sull’asse x → D(G(z)): output del discriminatore che prende in input un’immagine generata. Più si va verso destra, più il discriminatore pensa che l’immagine sia reale.

Sull’asse y → J(G) loss del generatore.

**Minimax**:

Si consideri la funzione di minimax (colore blu) per valori di D(G(z)) compresi tra 0.8 e 1.0 → la funzione ha una curva. Ciò significa che quando il discriminatore sbaglia (predice immagine reale di un’immagine generata), il gradiente è positivo. Ciò permette al generatore di migliorare le sue performance, il gradiente fornisce molta informazione.

Quando il discriminatore si è accorto perfettamente che l’input è un’immagine generata, la curva è molto piatta (per valori iniziali di D(G(z)). Ciò fornisce pochissima informazione al generatore per migliorare le sue performance. E’ il caso in cui il generatore sta sbagliando tanto.

La loss del generatore non può funzionare molto bene: aiuta molto quando il generatore funziona già molto bene, ma non aiuta affatto quando il generatore è scarso in quanto non dà alcuna informazione per migliorare.

**Non-saturating heuristic**:

Quando il generatore genera immagini che il discriminatore non è più in grado di distinguere allora il gradiente della loss è 0, non si sa più come migliorare. Funziona già molto bene.

Quando il discriminatore discrimina invece molto bene gli esempi reali da quelli generati, la loss è alta, il gradiente è alto e il generatore riesce a migliorare più velocemente. Fornisce molta informazione.

Riassumendo, il nuovo gioco è configurato tra generatore e discriminatore che minimizzano le proprie funzioni di loss.





Non si è più nel caso del gioco a somma zero in quanto la loss del generatore non è più l’inversa della loss del discriminatore. Non è più detto che esista il punto di equilibrio di Nash.

# Perché le GANs funzionano così meglio rispetto agli altri modelli generativi Maximum Likelihood?

Per un po’ di tempo, si pensava che le buone performance delle GANs erano dovute a minimizzare la Jensen-Shannon divergenze anziché minimizzare la Kullback-Leibler divergence.

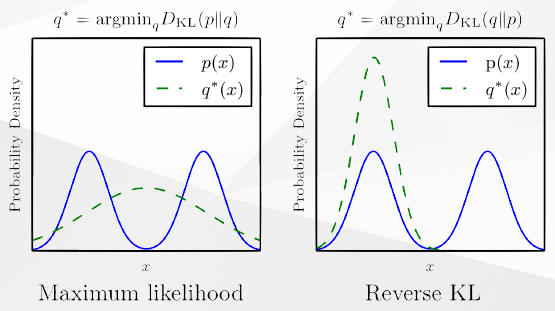
La KL divergence non è simmetrica ed è collegata con la stima maximum likelihood.

Tutti gli altri modelli generativi cercano di stimare la massima verosimiglianza minimizzando la KL(P\_data||P\_model), ovvero la KL della distribuzione dei dati rispetto alla distribuzione del modello.

Minimizzare JSD è più simile a minimizzare KL(P\_model||P\_data), la KL della distribuzione del modello rispetto alla distribuzione dei dati.

=> La KL non è una misura simmetrica.

Differenze delle KL e come interpretare i due modi di ottimizzare il modello.



p(x): distribuzione dei dati (blu) → bi-modale

q(x): distribuzione del modello (verde) → uni-modale

Si vuole rappresentare con una rappresentazione uni-modale (una sola moda), una distribuzione bi-modale (2 mode).

Nella prima figura, si vuole minimizzare la KL(p||q) di p rispetto a q.

Nella seconda figura, si vuole minimizzare la KL(q||p) di q rispetto a p.

Minimizzando KL(p||q) [fig.1], la soluzione tende a mediare tra le due mode in quanto non riesce ad approssimare tutta la distribuzione in modo ottimale.

Minimizzando invece KL(q||p) [fig.2], la soluzione si concentra su una delle due mode e ignora completamente l’altra.

Il motivo per cui si pensava che le GANs funzionassero meglio degli altri modelli è perché le GANs riuscivano a focalizzarsi di di più su una moda anziché mediare tra le varie mode.

Mediare tra le varie mode, nel caso delle immagini, voleva dire produrre immagini sfocate.

Invece concentrandosi su una sola moda, si poteva avere un’immagine più nitida, definita.

Perché è vera?

Essendo



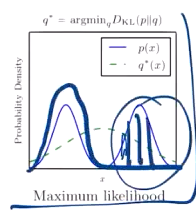
Per minimizzare KL(p||q), in entrambi i casi si vuole che il rapporto p(x)/q(x) sia il più vicino possibile a 1, quindi che q tenda a p il più possibile.

La differenza è come le due formule pesano gli errori.

**Nel caso KL(p||q), si pesano gli errori rispetto a p: se p(x) è bimodale e q(x) unimodale, è meglio distribuire q tra le due mode, altrimenti l’errore sulla seconda parte della distribuzione verrebbe pesato troppo**.



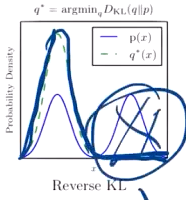
Se ci si focalizzasse sulla prima moda e sulla restante avesse valore 0, allora l’errore sulla (non si dice nulla sulla seconda moda) verrebbe pesato molto dalla loss, in quanto nella seconda moda la probabilità p(x) è alta ma si sta dicendo tutt’altro. Per non incorrere in questo tipo di errore, la soluzione ottima KL è mediare tra le due mode.



**Nel caso KL(q||p) si pesano gli errori rispetto a q. Allora è importante che dove q(x) è alto sia alto anche p(x). Concentrare tutta la massa di probabilità in uno solo dei due modi funzionerebbe bene**.



Se ci si concentra sulla prima moda (nella seconda moda la massa di probabilità è pari a 0), non importa perché l’errore è pesato secondo q. Secondo q, la seconda moda vale 0 come peso dell’errore e quindi non viene considerata. Non si paga nulla per aver modellato male la seconda moda. Conviene modellare solo la prima moda.



Questo ragionamento ha fatto pensare alle persone che le GANs funzionassero bene perché non faceva la media tra le mode.

Ultimamente si è dimostrato che questo vecchio modo di ragionare è sbagliato. Alcune nuove prove suggeriscono che l’uso della divergenza JSD non spiega il motivo per cui le GANs producano esempi più “nitidi”:

* Usando la maximum likelihood per ottimizzare, le GANs non mostrano problemi nello scegliere un numero ridotto di mode e generare immagini nitide.
* Le GANs spesso scelgono di generare esempi concentrandosi su pochissime mode (meno mode rispetto al numero di mode consentite dalla capacità del modello). KL(q||p) potrebbe selezionare tante mode quante sono le mode consentite dal modello.

Questo suggerisce che le GANs scelgono di generare un numero ridotto di mode **a causa di un difetto nella fase di training**. Non è un problema dovuto alla divergenza ma al modo in cui si sta facendo apprendimento.

La ragione per cui questo accade non è chiaro. Forse perché fa approssimazioni diverse rispetto ad altri modelli, oppure ottimizza una diversa famiglia di funzioni completamente diversa dalla famiglia di funzioni ottimizzate dagli altri modelli.

Ciò che sappiamo è che le GANs funzionano molto bene. Non abbiamo ancora capito qual è la ragione teorica dietro questi successi.

Deep Convolutional GAN Architecture (DCGAN)

Nel momento in cui sono state proposte le GAN, furono proposte diversi tipi modello convolutivo migliore da associare alle GAN. Si è affermato il modello DCGAN.

Sono caratterizzate da avere livelli batch normalization in quasi tutti i livelli sia nel generatore che nel discriminatore. Nel fare batch normalization, i batch D e G sono analizzati normalizzati separatamente.

L’ultimo livello del generatore il primo livello del discriminatore non sono batch normalizzati. Questo per permettere al discriminatore di imparare la distribuzione vera dei dati nei suoi primi livelli.

|  |
| --- |
| La **Batch Normalization** è una tecnica molto usata in quanto da ottimi risultati nell’ambito delle reti neurali. Insieme alle PAC, sono tecniche che permettono di migliorare in modo generale le performance.  **L’idea**: Spesso, si normalizza il dataset prima di darlo in input alla rete neurale: si elimina la media e si divide per la varianza. Aiuta alla rete a convergere più velocemente.  Con la batch normalization si porta questa idea al livello successivo. Non solo per i miei dati, ma batch a batch.   * batch a batch   + quando arriva un batch di dati, calcolo il fattore di normalizzazione su quel singolo batch   + i fattori di normalizzazione non vengono calcolati una volta per tutte, ma sono calcolati livello per livello.   In una rete neurale, ad esempio di questo tipo,    si ha un livello intermedio di batch normalization: per ogni attributo del livello precedente (attributi che descrive i dati), si apprendono due parametri   * varianza → valore per cui bisogna moltiplicare l’attributo * media → valore che bisogna sottrarre   per fare la normalizzazione del batch di dati.  Normalizzazione di un attributo v\_i    La rete trasforma i dati mentre si procede dal basso verso l’alto e ad ogni trasformazione, la normalizzazione fatta all’inizio viene persa.  Ad ogni passo si normalizzano in modo facile da calcolare per la rete stessa. |

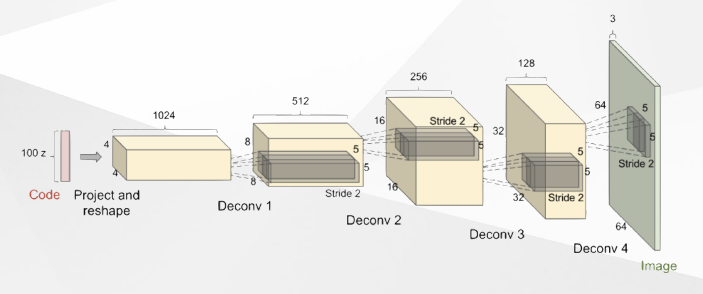
Nelle DCGANs, tutti i livelli (eccetto l’ultimo livello del generatore e il primo livello del discriminatore) hanno livelli di batch normalization.

L’architettura non contiene nè livelli di pooling nè livelli di unpooling. Quando il generatore ha bisogno di aumentare la dimensione spaziale della rappresentazione, viene usata la **convoluzione trasposta** (deconvoluzione) con uno stride più grande di 1.

L’ottimizzatore scelto è **Adam**, anziché la discesa del gradiente stocastica con momentum.

|  |
| --- |
| Un livello di pooling esegue una aggregazione delle informazioni nel volume di input, generando feature map di dimensione inferiore. Obiettivo è conferire invarianza rispetto a semplici trasformazioni dell'input mantenendo al tempo stesso le informazioni significative ai fini della discriminazione dei pattern.  Un livello di unpooling esegue una estensione delle informazioni nel volume di input, generando feature map di dimensione superiore.    La convoluzione parte da un pezzettino di immagine e un kernel di convoluzione, restituisce un valore di attivazione del kernel.  La deconvoluzione parte dal valore di attivazione del kernel e cerca di ricostruire l’immagine. |

L’architettura delle DCGANs



Si parte da z (rumore bianco), dopodiché si costruiscono una serie di livelli usando operatori di deconvoluzione per ottenere un’immagine di dimensioni appropriate per il problema che si sta affrontando.

Wasserstein GANs (WGANs)

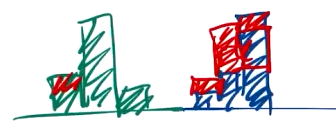
E’ una recente innovazione che ha avuto molto successo, funziona molto bene, sono più facili da addestrare rispetto alle GANs.

L’idea è quella di cambiare la funzione di loss alla base delle GANs. Nella prima parte dell’articolo viene argomentato in modo convincente che nei casi reali, per le distribuzioni reali che si vogliono modellare, è difficile far convergere il processo di apprendimento quando si usano le loss usate fino a quel momento per le GANs.

Quindi per evitare problemi di convergenza, si utilizza una nuova funzione di loss basata sulla **distanza di Wasserstein**, detta anche **distanza earth-mover**, tra due distribuzioni.

# Distanza di Wasserstein

Siano date due distribuzioni p e q

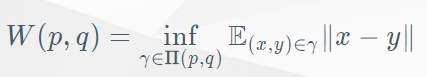
 

e si vuole calcolare la distanza tra le due distribuzioni.

Un modo per calcolare la distanza è immaginare che queste distribuzioni sono cumuli di terra da spostare da una parte all’altra. La distanza tra le due distribuzioni è data dalla minima quantità di terra che si deve spostare da una parte all’altra per trasformare una distribuzione nell’altra (renderle uguali).

La distanza tra due distribuzioni p e q è data dal numero minimo di mosse da effettuare per trasformare una distribuzione nell’altra.

In termini formali nel caso di distribuzioni continue è descritto dalla seguente formula:



Il minimo fra tutte le distribuzioni congiunte fra p e q del valore atteso.

Si prende il valore atteso E quando si estrae (x,y) seguendo della norma ||x - y|| → quanto x e y prese da lontane.

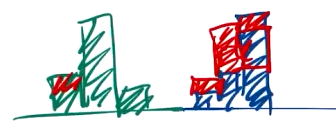
Si vuole trovare la ottimale per minimizzare questo valore atteso.

 **denota l’insieme di tutte le distribuzioni continue su di x e y le cui distribuzioni marginali siano rispettivamente p e q**. Altrimenti prenderei distribuzioni che non hanno niente a che fare con p e q.

Marginalizzando su y si vuole ottenere q e marginando su x si vuole ottenere di nuovo q.

Fra tutte le distribuzioni compatibili con questa proprietà, si sceglie che minimizza il valore atteso.

Si tratta sostanzialmente dell’applicazione nel continuo dell’idea



Si può mostrare che questa distanza abbia proprietà migliori per quanto riguarda la convergenza delle GANs.

Nel seguito, si utilizzerà f per denotare il discriminatore e g per denotare il generatore.

Sfortunatamente **l’infimum** su tutte le possibili distribuzioni compatibili con p e q **è altamente non trattabile**.

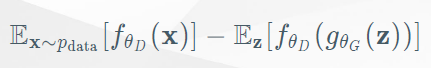
Però si può far vedere che **sotto l’assunzione che la famiglia di funzioni  considerate dalle reti siano continue rispetto a Lipschitz**.

Esiste un unico k su tutto lo spazio tale per cui questa espressione è sempre vera.

Tutte le funzioni Lipschitz continue sono uniformemente continue. Non è vero il contrario: esistono alcune funzioni uniformemente continue che non sono Lipschitz continue.

Se è vero che le reti che utilizzano questo tipo di funzioni riesco a vincolarle in modo tale da garantire questa proprietà, allora il calcolo della funzione W si semplifica.

**Si addestra il discriminatore a massimizzare la distanza W:** il discriminatore vuole rendere più evidente possibile il fatto che l’immagine generata sia diversa dall’immagine reale.



E’ molto simile all’espressione della funzione loss per il discriminatore ma è senza logaritmi.

**Il generatore invece è ottimizzato per minimizzare la distanza W**:



Con questo tipo di accorgimento, c’è evidenza teorica che le WGANs funzionano meglio. Ma prima di fare addestramento, **bisogna garantire che la famiglia di funzioni** ** siano continue rispetto a Lipschitz.**

Un trucco da utilizzare per garantire la proprietà sopra descritta è il seguente:

**Ogni volta che i parametri sono aggiornati, sono troncate nell’intervallo [-c, c] dove c è una costante definita dall’utente**.

Se c=0.1, quindi l’intervallo è [-0.1,0.1]. Se il parametro , si tronca in 0.1 per farlo cadere nell’intervallo [-0.1,0.1]. Si tratta di un undo di un passo che ha fatto l’ottimizzazione far fare la discesa del gradiente.

Si può garantire che la funzione è continua rispetto a Lipschitz, allora si può addestrare la GAN usando la distanza W.

Nei problemi legati alle WGAN, ci sono alcuni tentativi che cercano di forzare la continuità di Lipschitz senza utilizzare questo trucco però non sembra che funzionino particolarmente meglio rispetto a questa versione. Quindi questa versione viene utilizzata ancora frequentemente.

# Vantaggi WGANs

La nuova metrica di loss è **più interpretabile** e in particolare **correla** questa loss con la convergenza del generatore, quindi correla con la qualità del generatore.

A differenza delle altre loss, quando il generatore inizia a produrre immagini buone e man mano che le immagini generate migliorano, la loss scende invariabilmente.

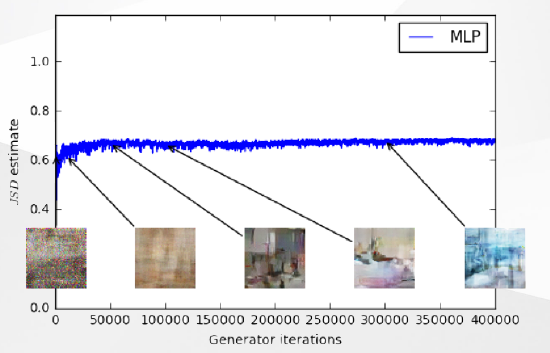
Con le altre loss questo non è garantito.

Questo tipo di approccio, spesso si cerca di addestrare il discriminatore fino a raggiungere una situazione di quasi ottimalità del discriminatore. In questa situazione non si addestra più il discriminatore, quindi si continua ad addestrare il generatore sotto una particolare loss. Questo **semplifica molto il processo di addestramento**.

Quando il discriminatore f è molto vicino all’ottimalità, la loss effettivamente è una stima abbastanza buona della distanza earth-mover e corretta fino ad un fattore determinato dalla costante c.

Evidenza empirica mostra che questa metrica **correla con la qualità degli esempi generati**.

## Altre loss spesso non correlano con la qualità degli esempi generati



**Caso 1**: **Multi-Layer Perceptron** **per generare le immagini**.

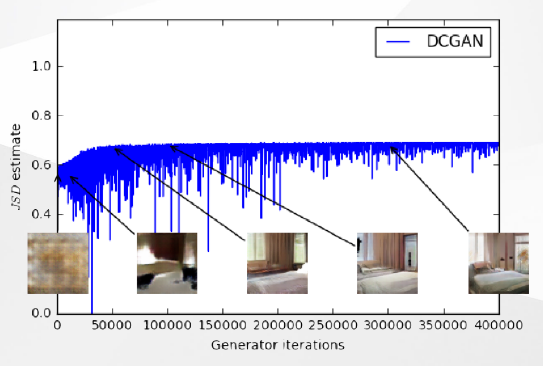
Sull’asse x, il numero di iterazioni.

Sull’asse y, stima della divergenza JSD.

Nella figura si osserva che la JSD sta crescendo leggermente, al limite è rimasta stabile. Non sta scendendo come vorremmo per avere avere immagini di qualità migliore. Eppure le immagini stanno cambiando e stanno andando verso la direzione giusta.

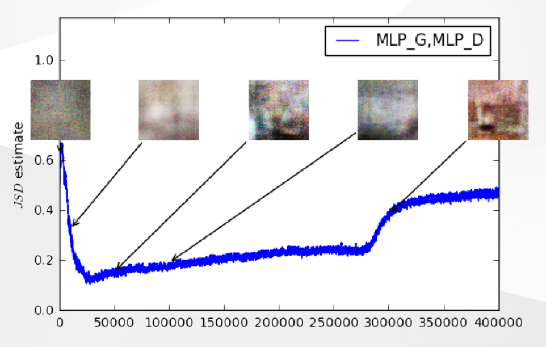
Anche se la JSD non da informazione giusta, si sta apprendendo bene.

Non si può osservare la JSD per capire quando si è giunti ad un buon punto dell’apprendimento, **non c’è correlazione tra qualità degli esempi generati e il valore della loss**.



**Caso 2: DCGAN per generare le immagini**.

La qualità delle immagini è migliorata tanto anche se osservando la stima della JSD è rimasta stabile (cresce solo all’inizio).



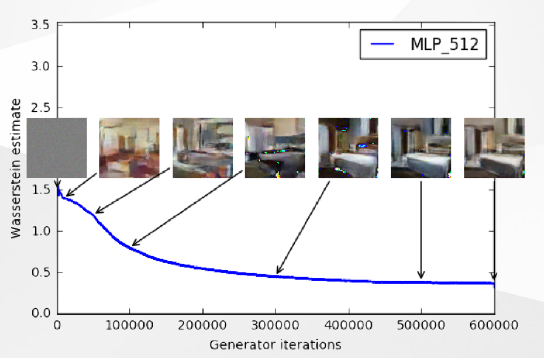
Caso 3: Multi-Layer Perceptron

E’ un caso molto importante.

Non c’è stata convergenza della rete: le immagini rimangono brutte.

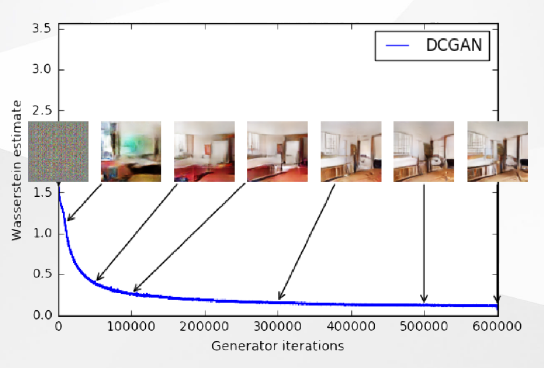
Non si riesce a leggere bene questa figura, non c’è una vera correlazione.

## La funzione di loss WGAN correla con la qualità degli esempi generati

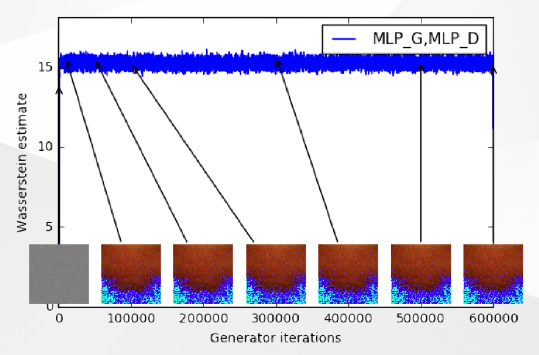


Man mano che le immagini migliorano di qualità, la loss scende costantemente.

C’è una forte correlazione tra la funzione di loss e la qualità delle immagini.



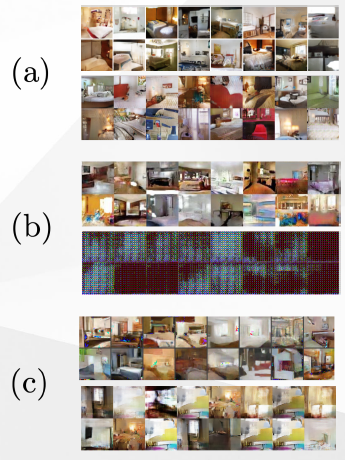
Anche nel caso delle DGAN, la funzione di loss correla molto bene con la qualità delle immagini.



Caso in cui la rete non converge: la qualità delle immagini è stabile e la loss rimane costante.

Uno dei vantaggi delle WGANs è che il discriminatore può essere addestrato fino all’ottimalità. Il fatto di continuare ad addestrare il generatore senza usare il discriminatore, rende più robusto l’addestramento del generatore. Cambiando l’architettura sottostante del generatore, i risultati non cambiano di molto.

Alcuni esempi



1. Le immagini sopra sono state addestrate con WGAN e quelle sotto con DCGAN. Stessa architettura. Entrambi sono abbastanza buone.
2. Stessa architettura del punto (a) ma sono stati rimossi i livelli di batch normalization. La WGAN è peggiorata di poco, ma la DCGAN ha perso ogni parvenza di ragionevolezza.
3. Architettura Multi-Layer Perceptron con stesso numero di pesi delle convoluzionali. Le WGAN funzionano peggio perché i livelli convoluzionali sono utili però continuano a funzionare. Per quanto riguarda invece le GAN normali, abbiamo un collasso del modello: inizia a generare sempre le stesse immagini. Problema tipico delle GAN: non riesce a generare immagini diverse.

# Alcuni trucchi

**Apprendimento supervisionato.** Incorporare le etichette nell’apprendimento migliora drammaticamente i risultati. Non è tanto importante come viene fatto ma è importante che ci sia l’informazione delle labels. Potremo anche non avere alcun meccanismo nel generatore per incorporare le classi, e addestrare semplicemente il discriminatore e il generatore a riconoscere/generare una particolare classe di immagini. Cambiare il dataset in modo tale da selezionare le classi. Questo sembra migliori la qualità delle immagini generate. Non è chiaro il motivo per cui questo funziona e non è neanche chiaro che effettivamente ci sia un miglioramento vero nel modo in cui gli oggetti stanno modellando la distribuzione sottostante. Semplicemente potrebbe essere che questa aggiunta di informazioni guidi la loss a generare immagini che soggettivamente noi esseri umani riteniamo siano più piacevoli da vedere.

Uno dei problemi che hanno le reti neurali, in particolare le GANs, è che spesso tendono a produrre output troppo confidenti. Può introdurre problemi nell’addestramento delle GANs. Si è cercato di trovare un escamotage per impedire questo comportamento. Si suggerisce di usare la tecnica **one side label smoothing**: consiste di sostuire l’etichetta 1 (positiva) con un valore leggermente più piccolo di 1

1 - alfa

con alfa numero reale piccolo scelto dall’utente, ad esempio 0.1 o 0.05

in modo tale da penalizzare il discriminatore quando restituisce valori vicino ad 1 perché è un valore troppo alto.

Questa tecnica funziona abbastanza bene per diverse ragioni:

* questa tecnica equivalesse a introdurre un fattore di regolarizzazione nella funzione appresa
* è un modo per far sì che le reti siano resistenti ad attacchi avversari

|  |
| --- |
| **Attacco avversario alla rete neurale**  Uno dei problemi legati alle reti neurali, è possibile aggiungere del rumore non visibile ad occhio nudo visibile ma calcolato in modo tale per cui si riesce a cambiare l’output della rete dalla classe A alla classe B senza cambiare in modo percettibile l’immagine.  Questo è un attacco avversario alla rete che fa la classificazione dell’immagine.  Esistono diverse tecniche per farlo: spesso si guardano i pesi della rete per capire come sfruttare pesi troppo alti o troppo bassi per fargli cambiare idea a seconda del cambiamento di pochi pixels. |

Il trucco one side label smoothing è un buon regolarizzatore per le reti neurali ed è un buon modo per resistere agli attacchi avversari.

Si noti che bisogna farla sulla classe positiva, e mai sulla classe negativa. Se si fa la stessa cosa sulla classe negativa (0+beta), il discriminatore potrebbe forzare il generatore a insistere sui propri errori.

**Virtual Batch Normalization**. Per fare la normalizzazione, la rete impara i pesi da sé. Se i batch sono troppo piccoli, i pesi hanno poco supporto statistico e possono variare molto, introducendo del rumore durante l’addestramento della rete che può essere dannoso all’addestramento stesso.

E’ abbastanza frequente questo problema nelle GANs perché si stanno usando modelli grossi e i vincoli hardware possono forzare a usare batch piccoli in modo tale che tutta la computazione stia nella memoria della GPU. Per risolvere questo problema è calcolare la normalizzazione su un batch più grande. Si tiene da parte un batch separato dagli altri e quando si fa normalizzazione, si usa sia il batch attuale che il batch tenuto da parte. L’insieme degli esempi è più grande, la fluttuazione è più piccola ed il problema si risolve.