Progetto di
Machine Learning e
Sistemi Intelligenti per Internet

Deep Learning e Immagini: Impiego di Generative Adversarial Networks come modelli generativi per creare nuovi contenuti e artefatti in base all'analisi di un dataset in input.

Bianca Camilla, Valeriani Dalia
a.a. 2018/2019

Indice

Abstract

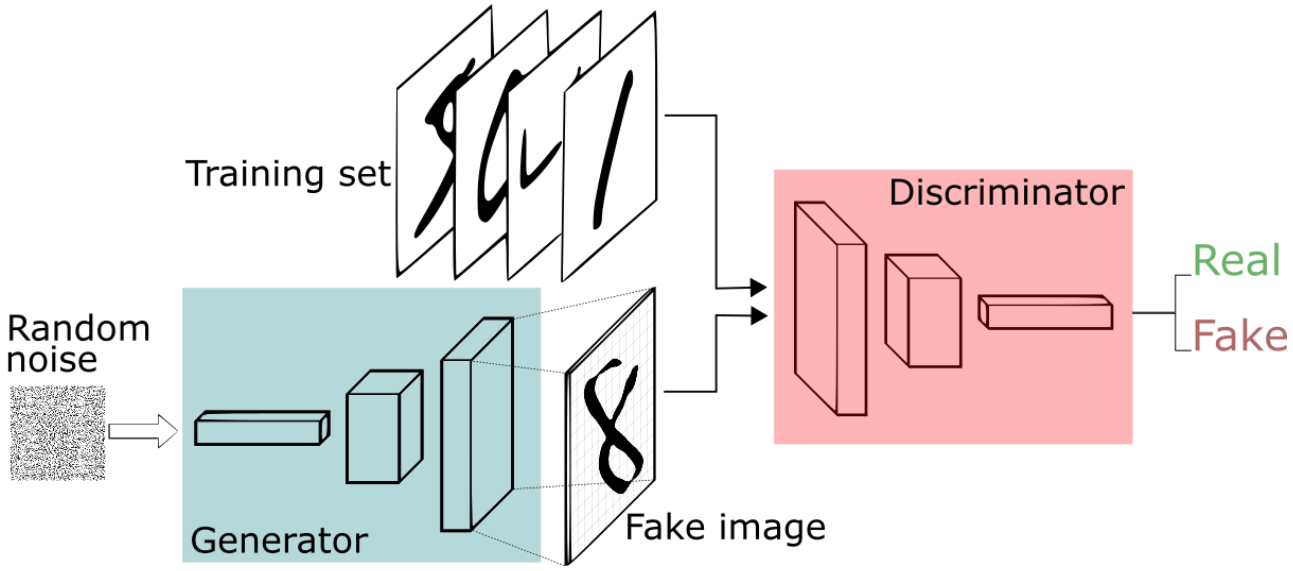
L'obiettivo del progetto consiste nell'analisi e nello sviluppo dei cambiamenti dei parametri e degli iper-parametri relativi alle reti neurali che realizzano una Generative Adversarial Network. In particolare, l'attenzione è rivolta all'evoluzione della GAN partendo da un'implementazione semplice, fino ad arrivare ad una struttura più complessa. Infatti, inizialmente è stata analizzata una rete base di tipo Vanilla, per poi far riferimento all'utilizzo di operatori convolutivi tipici di una DCGAN, richiamando la conoscenza di argomenti studiati a lezione. Inoltre, come applicazione aggiuntiva, sono stati utilizzati diversi dataset per sperimentare risultati differenti.

Il codice di riferimento, opportunamente commentato, è presente nel repository di GitHub https://github.com/daliavaleriani/gan.

GAN

"The coolest idea in deep learning in the last 20 years."—Yann LeCun sulle GAN.

Le *Generative Adversarial Networks* (reti avversarie generative) sono una combinazione di vari modelli di Deep Learning che cercano di competere l'uno contro l'altro. L'architettura della GAN presa in oggetto è la seguente:



La GAN è costituita da due reti neurali che sono in competizione tra di loro, che prendono il nome di:

- Generatore: il suo compito è generare nuovi dati simili a quelli del dataset di riferimento, partendo da rumore casuale;
- Discriminatore: il suo obiettivo è riconoscere se un dato di input è "reale" (appartenente al set di dati originale) o se è "falso" (generato dal Generatore).

Competendo l'uno contro l'altro, questi due modelli forniranno i migliori dati possibili, nuovi e mai visti.

Vanilla GAN

La Vanilla GAN è definita la "GAN originale", infatti rappresenta una delle prime versioni funzionanti mai realizzate. Essa è il tipo più semplice di GAN, in cui entrambe le reti che la compongono sono semplicemente percettroni multistrato. Proprio per la sua semplicità è stata scelta come primo oggetto di studio, allo scopo di comprendere approfonditamente le sue funzionalità ed il suo comportamento.

La GAN in oggetto presenta 3 hidden layers sia per il Generatore che per il Discriminatore, ciascuno dei quali è seguito da una funzione di attivazione non lineare Leaky-ReLU e da un livello Dropout, utilizzato per evitare l'overfitting.

Una funzione Sigmoid viene applicata all'output con valore reale del Discriminatore per ottenere un valore nell'intervallo aperto (0, 1). Nel generatore, invece, lo strato di output avrà una funzione di attivazione TanH, che mappa i valori risultanti nell'intervallo (-1, 1), lo stesso intervallo in cui le immagini MNIST pre-elaborate sono limitate.

La LeakyReLU utilizzata è una variante della ReLU, una funzione di attivazione molto usata; ha una piccola pendenza per i valori negativi, invece che 0. Questa pendenza è chiamata "negative slope". La Leaky ReLU è utile per risolvere il "dying ReLU problem".

Viene utilizzata la Cross Entropy binaria (BCE), che è un caso particolare della Cross Entropy in cui è possibile classificare solamente due classi.

Per entrambe le reti sono stati svolti dei test utilizzando come funzione di ottimizzazione prima Adam e poi SGD per studiare l'impatto che tali funzioni hanno sull'addestramento della rete.

Analisi dei parametri

Come spiegato nel codice, un’epoca corrisponde a 600 batches.

Inoltre, si considera la denominazione in figura:

z → Noise vector G(z) → Generator's output → x_{fake}

x → training sample → x_{real}

D (x) → Discriminator's output for x_{real} → P(y | x_{real}) → {0,1}

D (G(z)) → Discriminator's output for x_{fake} → P(y | x_{fake}) → {0,1}

at Discriminator D	at Generator G
D (x) → should be maximized D (G(z)) → should be minimized	D (G(z)) → should be maximized

Adam

L'algoritmo di ottimizzazione Adam è un'estensione della discesa del gradiente stocastico (SGD). Può essere usato al posto del classico SGD per aggiornare i pesi di rete iterativamente in base ai dati del training. La discesa del gradiente stocastico, infatti, mantiene un singolo learning rate (chiamato *alfa*) per tutti gli aggiornamenti di peso; inoltre la sua velocità di apprendimento non cambia durante l'addestramento. In Adam, invece, viene mantenuto un learning rate (LR) per ogni peso della rete e adattato separatamente all'apprendimento.

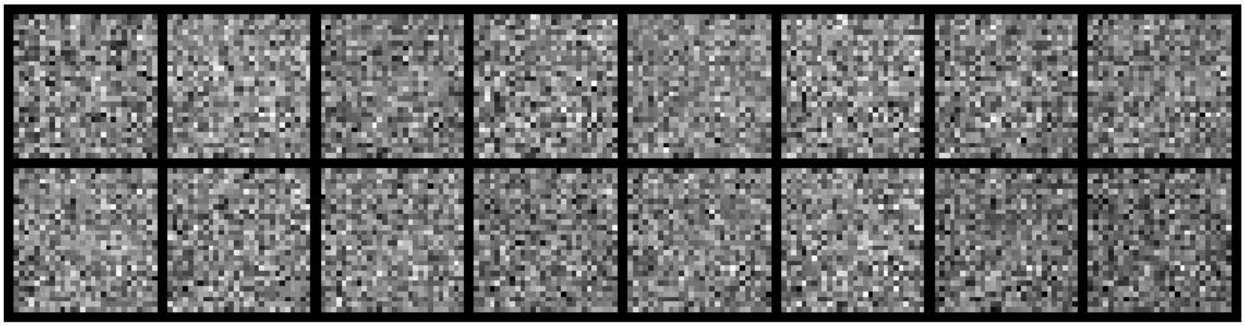
Nelle prove svolte si è analizzato il comportamento della GAN al variare del learning rate e del numero di epoche. In tutti i test è stato utilizzato un *batch_size* pari a 100, ovvero 600 batch per ogni epoca, come riportato nel codice.

50 epoche

Si analizza il comportamento della rete fissato il numero di epoche a 50 e al variare del learning rate:

- `LR=0.0002`

  Inizialmente, le immagini generate sono puro rumore (in seguito, si ometterà la figura relativa alla prima epoca di addestramento):



```
Epoch: [0/50], Batch Num: [0/600]
Discriminator Loss: 1.4111, Generator Loss: 0.6874
D(x): 0.4907, D(G(z)): 0.5028
```

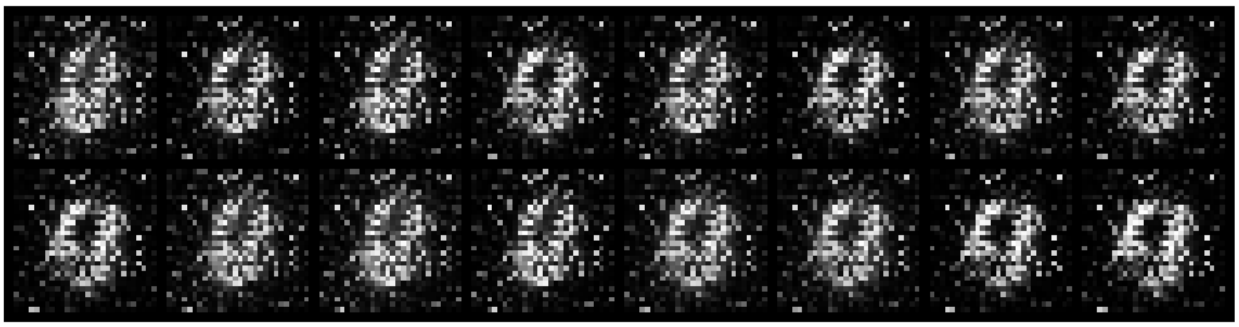
Si evidenzia l'errore del Discriminatore, che inizialmente non è in grado di discriminare, in quanto non riesce a classificare correttamente le immagini in "reali" o "false".
Dopodiché si evince un leggero miglioramento:

```
Epoch: [25/50], Batch Num: [400/600]
Discriminator Loss: 0.5927, Generator Loss: 1.5718
D(x): 0.8941, D(G(z)): 0.3095
```

Il training continua fino ad ottenere immagini "false" un po' scadenti, probabilmente a causa del basso numero di epoche:

```
Epoch: [49/50], Batch Num: [400/600]
Discriminator Loss: 1.0543, Generator Loss: 1.1091
D(x): 0.6122, D(G(z)): 0.3570
```

- LR=0.00002



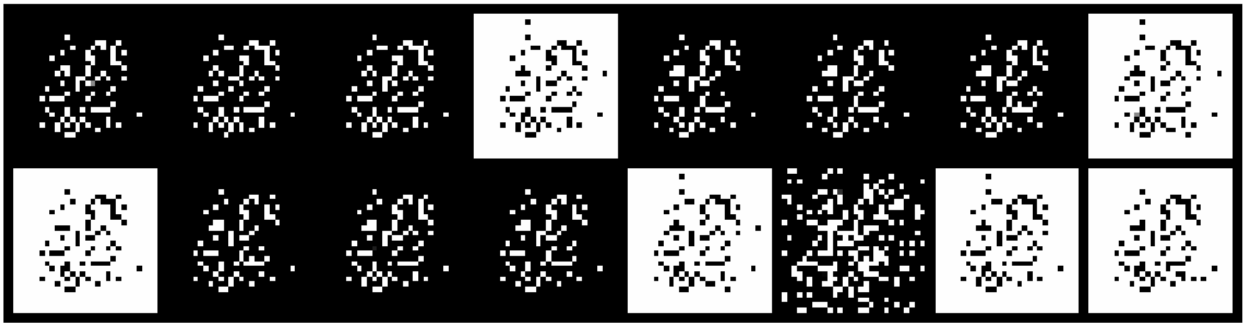
```
Epoch: [25/50], Batch Num: [400/600]
Discriminator Loss: 0.0013, Generator Loss: 11.5701
D(x): 0.9999, D(G(z)): 0.0012
```

Con un LR inferiore, si nota che l'errore del Generatore è abbastanza alto; questo indica che non è in grado di ingannare il Discriminatore.

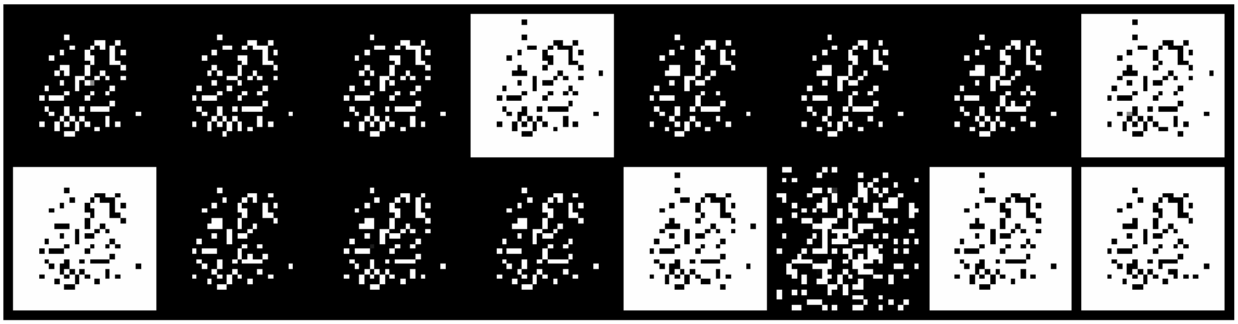
```
Epoch: [49/50], Batch Num: [400/600]
Discriminator Loss: 0.1235, Generator Loss: 5.0165
D(x): 0.9821, D(G(z)): 0.0605
```

Per un numero limitato di epoche, D(x) tende quasi a 1 e D(G(z)) tende a 0, ma in realtà il Generatore non riesce a creare riproduzioni con varietà numerica. Infatti, la loss del Generatore è leggermente alta. Questo errore prende il nome di *mode collapse*, approfondito in una sezione successiva.

- LR=0.002



```
Epoch: [25/50], Batch Num: [400/600]
Discriminator Loss: 27.6310, Generator Loss: 27.6310
D(x): 0.0000, D(G(z)): 0.0000
```



```
Epoch: [49/50], Batch Num: [400/600]
Discriminator Loss: 27.6310, Generator Loss: 27.6310
D(x): 0.0000, D(G(z)): 0.0000
```

Come si può notare dalle figure precedenti, sia a metà del training (25ª epoca) che alla fine (50ª), gli indicatori rimangono costanti; ciò è sintomo del fatto che la rete ha cessato di apprendere.

150 epoche

- `LR=0.0002`

  Con un numero maggiore di epoche ed un learning rate adeguato, il processo di apprendimento subisce notevoli miglioramenti, come si può evincere dal risultato visivo finale e dai dati sottostanti, in figura:

```
Epoch: [149/150], Batch Num: [400/600]
Discriminator Loss: 1.2764, Generator Loss: 0.9295
D(x): 0.5684, D(G(z)): 0.4429
```

Sono stati effettuati molteplici test con diversi valori di LR, ma non sono stati riscontrati comportamenti degni di nota e, perciò, non è stato considerato utile riportarli.

SGD

Nelle prove svolte si è analizzato il comportamento della GAN al variare del learning rate e del numero di epoche. Per utilizzare l'algoritmo di Stochastic Gradient Descent, è necessario sostituire la funzione di ottimizzazione con il seguente codice nel notebook di Colaboratory originario:

```
# Optimizers

d_optimezer = optim.SGD(discriminator.parameters(), lr = 0.0002, momentum=0.9)
g_optimizer = optim.SGD(generator.parameters(), lr = 0.0002, momentum=0.9)
```

50 epoche

- `LR=0.01`

  Il learning rate proposto è troppo alto; a fine addestramento è presente un enorme errore del generatore:

```
Epoch: [49/50], Batch Num: [400/600]
Discriminator Loss: 0.0000, Generator Loss: 52.4627
D(x): 1.0000, D(G(z)): 0.0000
```

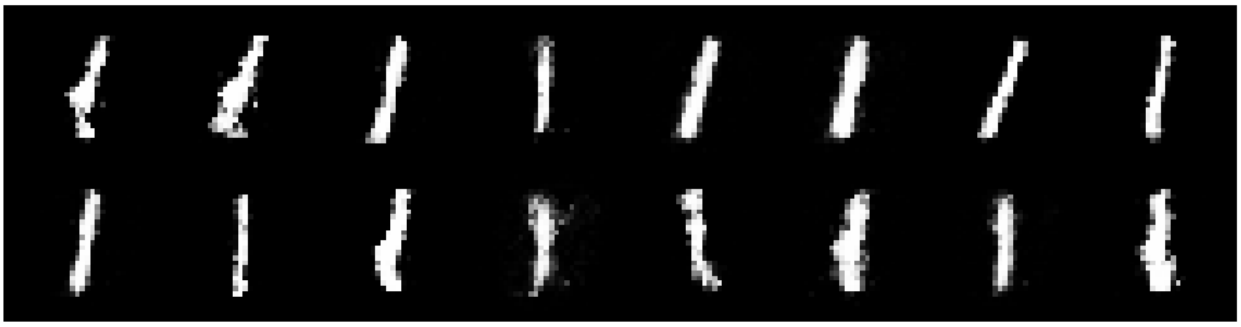
  Difatti, quando la velocità di apprendimento è troppo grande, la discesa del gradiente può inavvertitamente aumentare, piuttosto che diminuire, l'errore dell'addestramento.

- LR=0.001

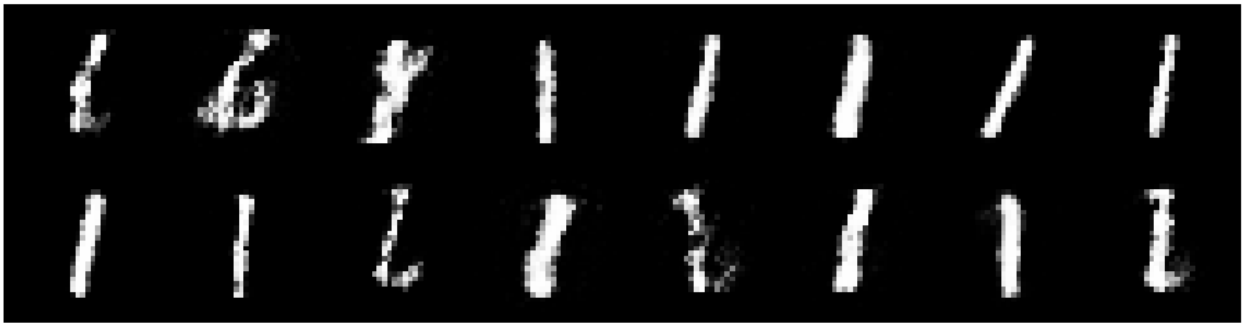
  L'addestramento della rete sembra andare a buon fine dai dati riportati, ma dall'output si evince che il Generatore non riesce a rappresentare una varietà di cifre sufficiente (problema del *mode collapse*):

```
Epoch: [49/50], Batch Num: [400/600]
Discriminator Loss: 0.3466, Generator Loss: 2.8302
D(x): 0.9263, D(G(z)): 0.1179
```

- LR=1.0e^{-3}



```
Epoch: [25/50], Batch Num: [400/600]
Discriminator Loss: 0.2485, Generator Loss: 2.7219
D(x): 0.9449, D(G(z)): 0.1256
```



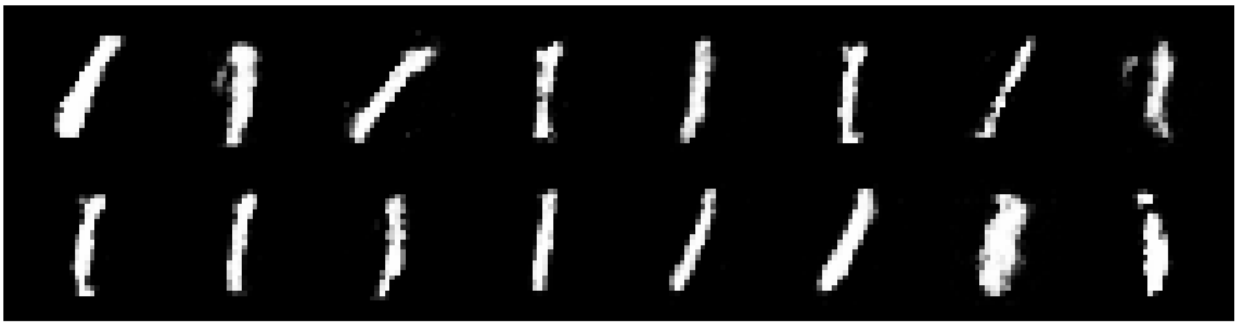
```
Epoch: [49/50], Batch Num: [400/600]
Discriminator Loss: 0.2889, Generator Loss: 2.6891
D(x): 0.9296, D(G(z)): 0.0917
```

Il risultato sembra simile al test con `LR=0.001`, ma leggermente migliore, come riportato nelle figure precedenti.

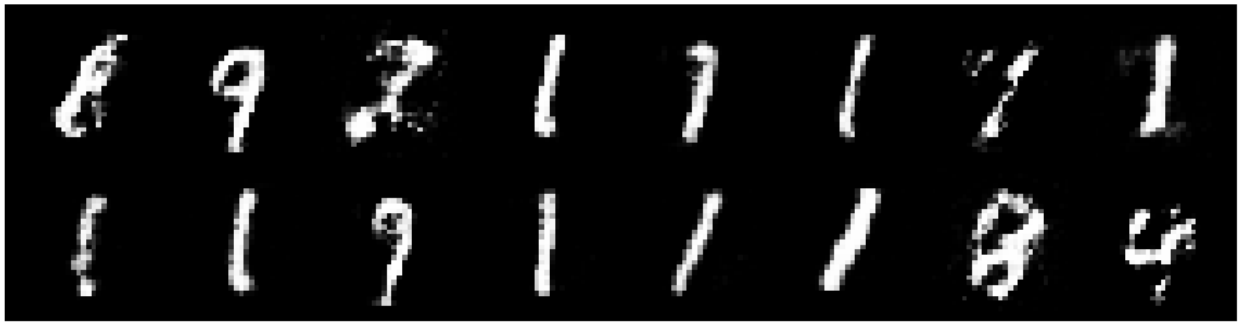
150 epoche

Nei test seguenti si è deciso di fissare il LR a e^{-3}, in quanto dimostratosi il miglior valore tra le prove svolte precedentemente con l'utilizzo della funzione di attivazione SGD.

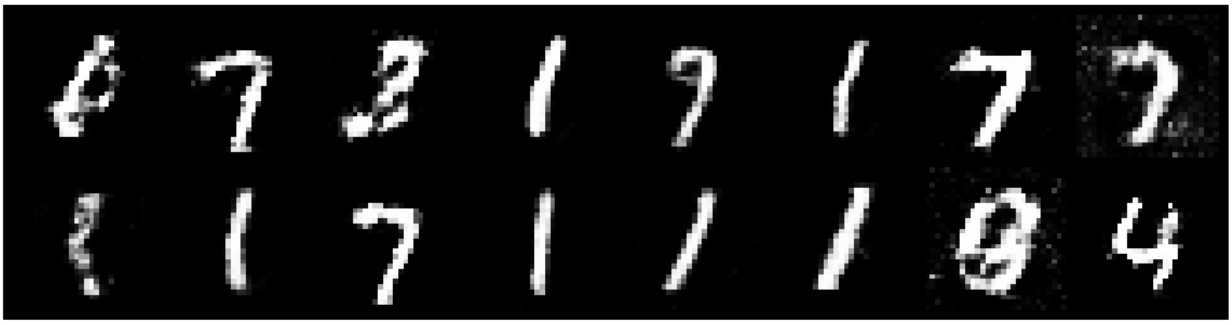
- 600 batch (`batch_size=100`)



```
Epoch: [49/150], Batch Num: [400/600]
Discriminator Loss: 0.4894, Generator Loss: 2.6947
D(x): 0.8508, D(G(z)): 0.0859
```



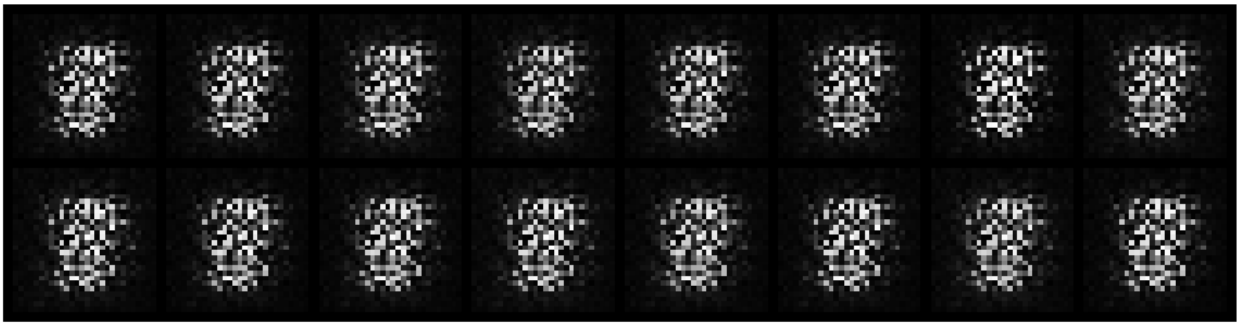
```
Epoch: [100/150], Batch Num: [400/600]
Discriminator Loss: 0.4859, Generator Loss: 2.1587
D(x): 0.8391, D(G(z)): 0.1335
```



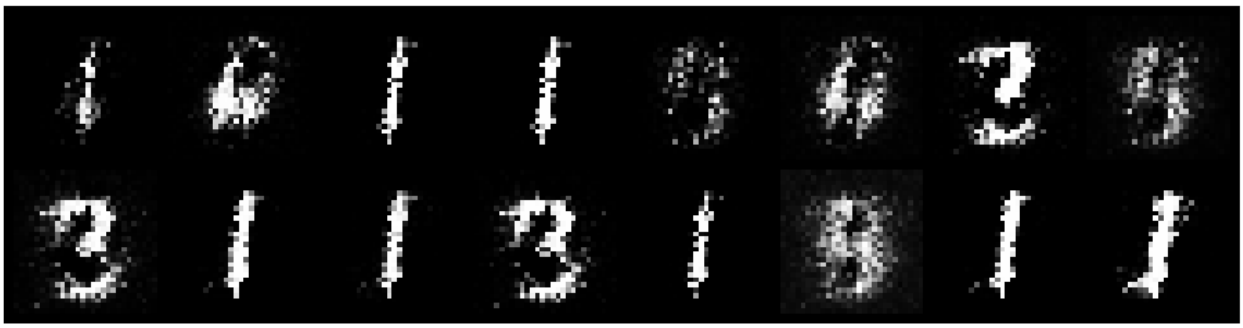
```
Epoch: [149/150], Batch Num: [400/600]
Discriminator Loss: 0.6356, Generator Loss: 1.8557
D(x): 0.8170, D(G(z)): 0.2117
```

Si può notare che, aumentando il numero di epoche, lo SGD funziona meglio: si evidenziano, infatti, miglioramenti prestazionali nell'addestramento della rete.

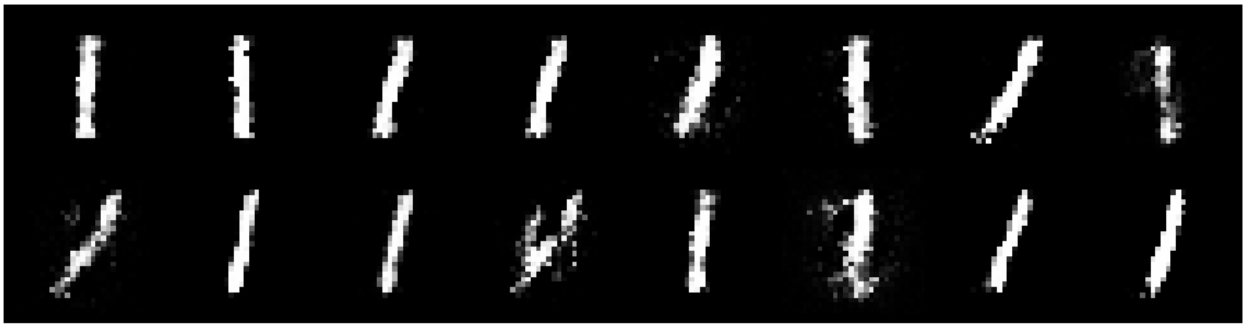
- 60 batch (`batch_size=1000`)



```
Epoch: [50/150], Batch Num: [0/60]
Discriminator Loss: 0.1483, Generator Loss: 6.2771
D(x): 0.9551, D(G(z)): 0.0610
```



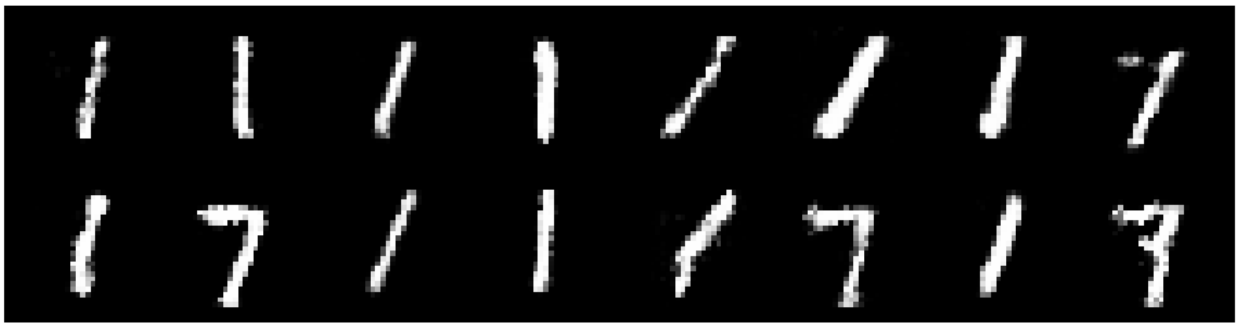
```
Epoch: [100/150], Batch Num: [0/60]
Discriminator Loss: 0.0972, Generator Loss: 5.4985
D(x): 0.9723, D(G(z)): 0.0261
```



```
Epoch: [149/150], Batch Num: [0/60]
Discriminator Loss: 0.1801, Generator Loss: 3.9579
D(x): 0.9465, D(G(z)): 0.0406
```

Nonostante l'elevato numero di epoche (rispetto agli altri test effettuati), il numero limitato di batch preclude un buon risultato, come si evince dalle figure riportate; in quest'ultime si nota un miglioramento delle cifre generate intorno alla 100ª epoca, fino a terminare con il problema del *mode collapse*.

- 1000 batch (`batch_size=60`)



```
Epoch: [50/150], Batch Num: [800/1000]
Discriminator Loss: 0.4199, Generator Loss: 2.3347
D(x): 0.8565, D(G(z)): 0.1037
```

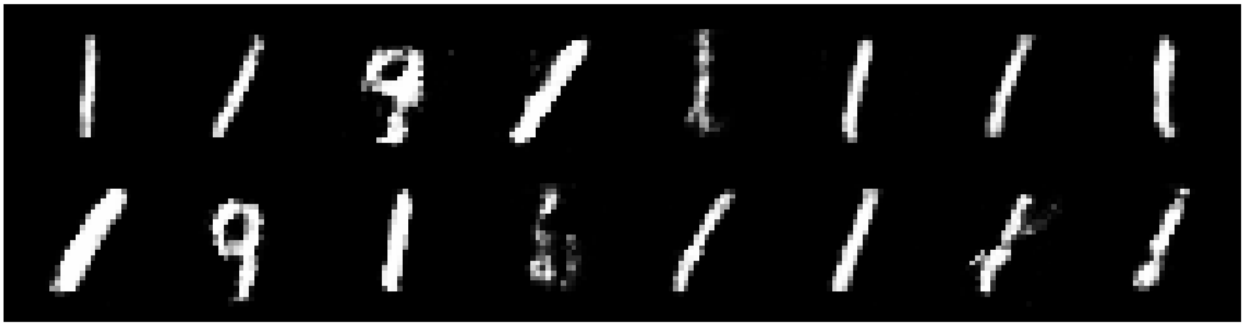
```
Epoch: [100/150], Batch Num: [800/1000]
Discriminator Loss: 0.6692, Generator Loss: 1.7542
D(x): 0.7973, D(G(z)): 0.1964
```

```
Epoch: [149/150], Batch Num: [800/1000]
Discriminator Loss: 0.5911, Generator Loss: 1.5873
D(x): 0.7964, D(G(z)): 0.2106
```

Si può notare che, aumentando il numero di batch (e, quindi, diminuendo il `batch_size`), i risultati effettuati evidenziano un discreto risultato.

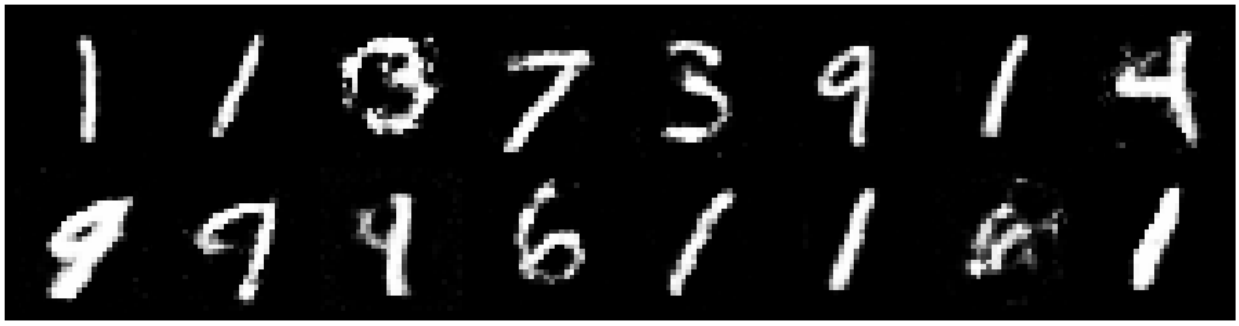
300 epoche

- `LR=1.0e-3`



```
Epoch: [100/300], Batch Num: [400/600]
Discriminator Loss: 0.3570, Generator Loss: 1.9669
D(x): 0.9258, D(G(z)): 0.1510
```

```
Epoch: [200/300], Batch Num: [400/600]
Discriminator Loss: 0.6667, Generator Loss: 1.9151
D(x): 0.7907, D(G(z)): 0.2058
```



```
Epoch: [299/300], Batch Num: [400/600]
Discriminator Loss: 0.7844, Generator Loss: 1.6517
D(x): 0.7307, D(G(z)): 0.2523
```

Con l’aumento delle epoche, il processo di addestramento presenta un’ottimizzazione dei valori delle funzioni di output; inoltre, le funzioni di loss tendono a convergere, dimostrando che il generatore riesce a generare immagini “false” abbastanza verosimili.

Osservazioni Vanilla GAN

Dai test effettuati, si evince che l’errore del Discriminatore è molto alto all’inizio, in quanto non sa come classificare correttamente le immagini come reali o false. Quando il Discriminatore diventa migliore, il suo errore diminuisce e conseguentemente l’errore del Generatore aumenta, dimostrando che il Discriminatore “vince” sul Generatore e può classificare correttamente i campioni falsi.

Durante l’allenamento, l’errore del Generatore si abbassa, il che implica che le immagini che genera sono migliori. Di conseguenza, l’errore del Discriminatore aumenta perché le immagini false stanno diventando sempre più realistiche.

Nel tempo le loss tenderanno a convergere ad un valore intorno all’1, sintomo del corretto funzionamento della GAN.

Il learning rate è un iperparametro molto delicato da impostare nelle reti neurali. La scelta del suo valore può essere abbastanza critica, poiché se è troppo piccolo la riduzione dell'errore sarà molto lenta, mentre se è troppo grande possono verificarsi oscillazioni divergenti.

In particolare, nella GAN proposta, si capisce che è difficile trovare un LR ottimale. Pertanto, non bisognerebbe utilizzare né un learning rate troppo alto, né troppo basso. Ciononostante, si cerca di configurare il modello in modo tale che, in media, un insieme di pesi "abbastanza buono" si trovi ad approssimare il problema di mappatura rappresentato dal set di dati di addestramento.

Dai risultati ottenuti limitatamente alle diverse prove effettuate in questo studio, è stato possibile concludere che, utilizzando la funzione di ottimizzazione Adam sia per il Generatore che per il Discriminatore, il LR migliore è circa 0.0002. Al contrario, impiegando la funzione di ottimizzazione SGD per entrambi i moduli, il LR che mostra risultati abbastanza buoni è e^{-3}, congiuntamente all'utilizzo di un maggior numero di epoche.

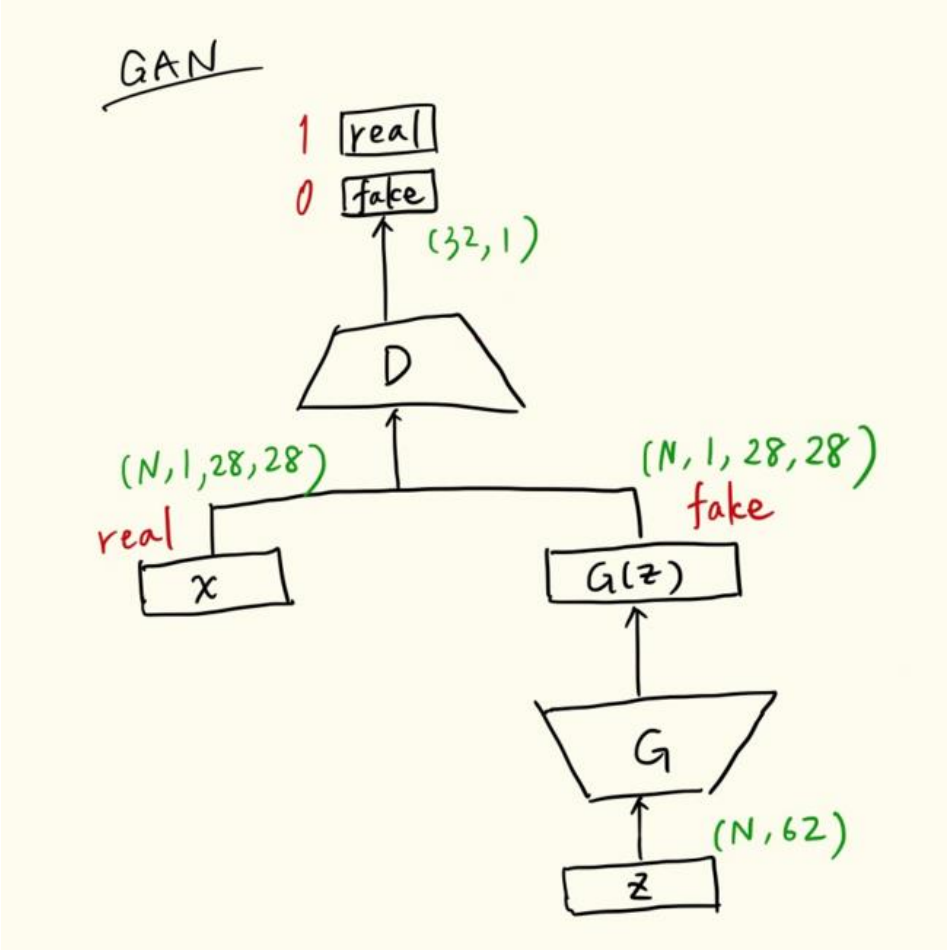
DCGAN

Una GAN di tipo Deep Convolutional è in grado di creare immagini originali fotorealistiche utilizzando una combinazione intelligente di due Deep Neural Network che competono tra loro. Sono state uno dei primi grandi miglioramenti all'interno delle architetture di tipo GAN.

Per costruire una DCGAN, si creano due reti neurali di tipo *deep*; dopodiché, si fanno combattere una contro l'altra, senza sosta, cercando di "farsi fuori" l'un l'altra. Nel processo, entrambe diventano più forti e vertono a migliorarsi.

Questo tipo di GAN utilizza specificatamente degli strati convolutivi per il Discriminatore e de-convolutivi per il Generatore, come si vedrà in seguito.

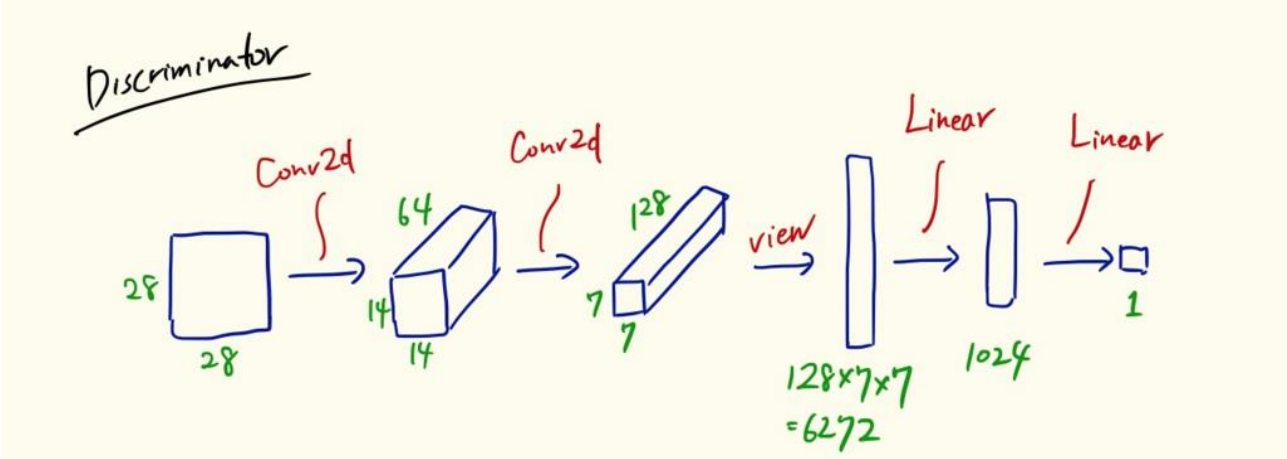
L'architettura DCGAN presa in esame è mostrata in figura seguente:



- La *N* indica la dimensione dei mini-batch;

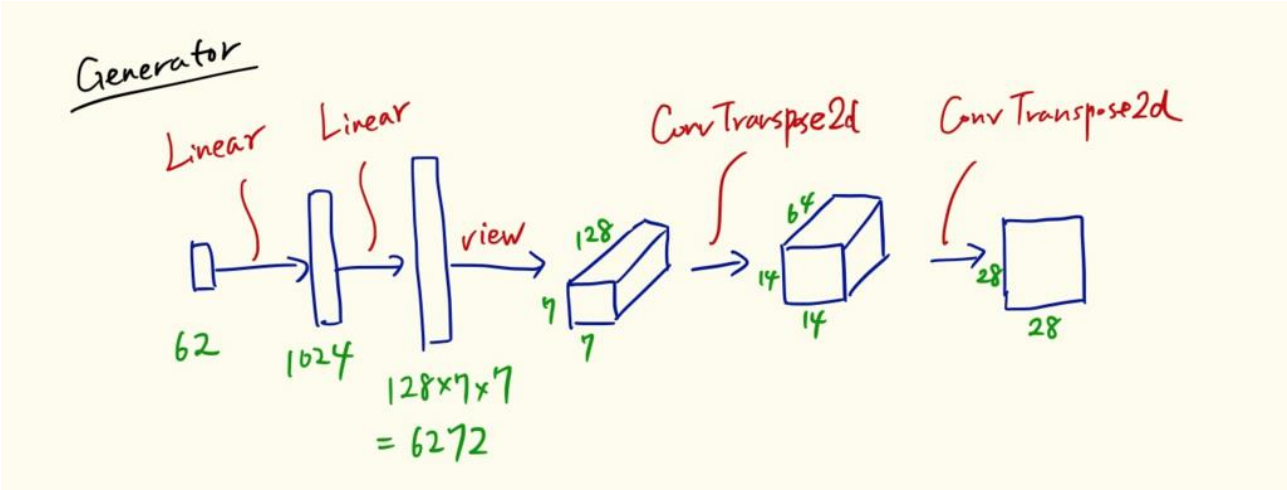
- Il Generatore genera l'immagine neurale “falsa” *G(z)* dal vettore *z*, composto da numeri casuali;
- *x* è un’immagine reale del dataset MNIST;
- Il Discriminatore è una rete neurale che identifica se l'immagine di input (che può essere l’immagine reale *x* o l’immagine falsa *G(z)*) è reale (1) o falsa (0).

L’architettura del Discriminatore usata per il dataset MNIST è riportata in figura seguente:



- Il Discriminatore è una rete neurale convoluzionale ordinaria;
- La dimensione dell'immagine viene dimezzata senza utilizzare il *MaxPooling*;
- La dimensione dell'immagine è dimezzata poiché lo *stride* è uguale a 2;

La figura seguente mostra l’architettura del Generatore usata per il dataset MNIST:



- Il Generatore prende l'input *z*, un vettore casuale a 62 dimensioni che viene utilizzato come "seed" per generare un'immagine. Infatti, uno dei punti di forza delle GAN è la generazione di un'immagine a partire da numeri casuali;
- Innanzitutto, si inserisce uno strato lineare e si espande fino a 6272 dimensioni. Dopodiché si converte con `view()` in un tensore di dimensioni 7x7x128; in realtà, nella rete è in 4D, ma le dimensioni del batch sono omesse nella figura precedente;
- La funzione `ConvTranspose2D` estende le dimensioni dell'immagine a 28 pixel per 28 pixel di MNIST e riduce il numero di canali. L'output sarà un'immagine MNIST di 1 canale 28x28 pixel.

Linee guida

- Si rimpiazzano tutti i *max pooling* con stride convolutivi;
- Si utilizza la convoluzione trasposta per il sovracampionamento delle immagini;
- Si adotta la *Batch Normalization* per tutti i livelli;
- Si impiega la funzione di attivazione ReLU per tutti gli strati del generatore, eccetto l'output, che sfrutta la tangente iperbolica;
- Si utilizza la funzione di attivazione LeakyReLU per tutti gli strati del Discriminatore.

In entrambi i moduli si utilizza la *Batch Normalization*, che aumenta la stabilità della rete neurale e permette ad ogni layer di addestrarsi in modo leggermente indipendente rispetto agli altri. Questa tecnica di normalizzazione consente di aumentare il learning rate mantenendo comunque efficiente l'addestramento. Inoltre, riduce l'overfitting perché ha un leggero effetto di regolarizzazione. Tale applicazione è simile al Dropout.

Analisi di parametri

Man mano che l'apprendimento progredisce, il Discriminatore diventa sempre più forte e il Generatore perde la competizione.

Con l'avanzare dell'apprendimento, si scopre che il Generatore è in grado di generare immagini "false" sempre più sofisticate, per cui il Discriminatore non può più trascurarle.

Adam

Rispetto alla GAN di tipo Vanilla, qui si utilizzano i parametri *beta*, che sono coefficienti utilizzati per calcolare le medie correnti del gradiente ed il suo quadrato.

25 epoche

- `batch_size=128`

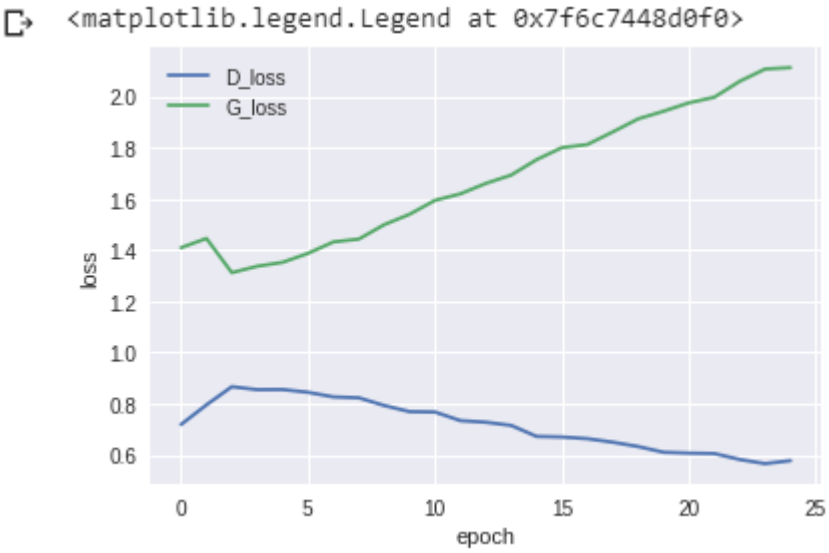
  Eseguendo l'addestramento delle GAN con gli iperparametri riportati nel codice, la rete mostra un output finale soddisfacente, nonostante il limitato numero di epoche.

Epoca 1

Epoca 10

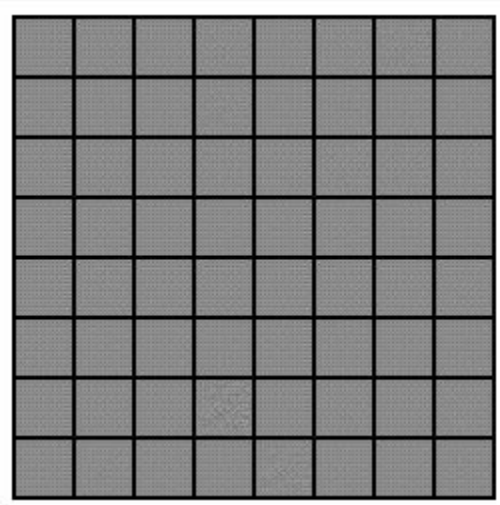
Epoca 25

Inoltre, si registrano delle loss contenute, come mostrato nella seguente figura:



- Sperimentazione: *D utilizza ReLU*

  Si utilizza la funzione ReLU per il Discriminatore, nonostante le linee guida suggeriscano di usare la Leaky ReLU per ottenere un'architettura DCGAN stabile. Infatti, i risultati ottenuti si dimostrano scadenti:



Epoca 25

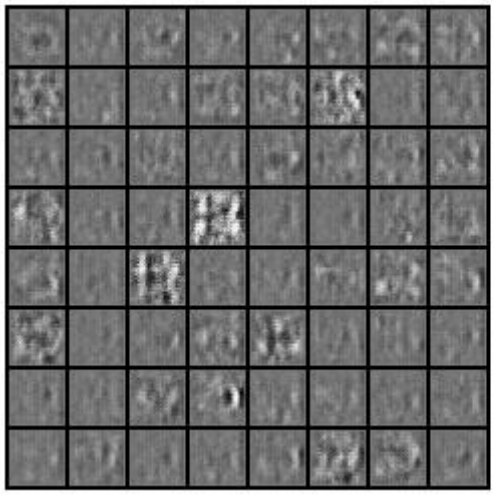
- `LR=0.002`

  A scopo sperimentale, si è incrementato notevolmente il valore del learning rate; ciò mostra risultati visivamente buoni, al contrario dell'output della GAN precedentemente analizzata (Vanilla):

Epoca 25

- `batch_size=256`

  Aumentando la dimensione del batch, si ottiene un numero di batch inferiore. Infatti, la rete si addestra in tempi più brevi ed i risultati sono leggermente peggiori:



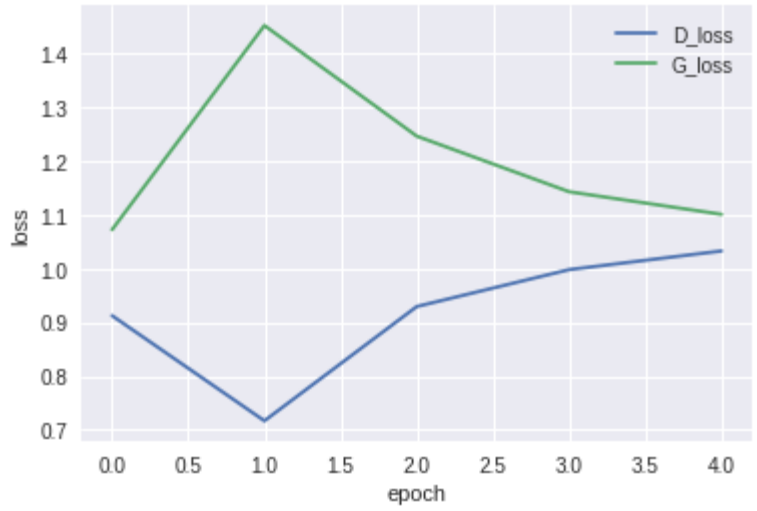
Epoca 1

Epoca 10

Epoca 25

Il grafico delle loss, alla prima epoca, mostra un evidente picco massimo nel Generatore ed un picco minimo nel Discriminatore, per poi terminare ad una convergenza intorno ad un valore che è circa 1:

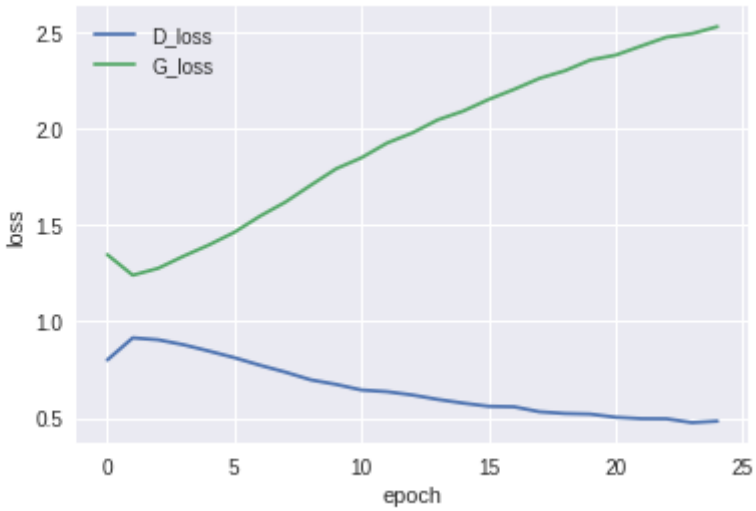
```
<matplotlib.legend.Legend at 0x7fa7d0bac780>
```



- `batch_size=64`

  Dal grafico delle loss ottenuto si nota che, dopo un primo assestamento delle loss intorno alla prima epoca per entrambi i moduli, il Discriminatore prende il sopravvento, mostrandosi nettamente più forte del Generatore. Infatti, intorno alla 25ª epoca la funzione di loss del Discriminatore si aggira intorno al valore 0.5:

```
<matplotlib.legend.Legend at 0x7f450599e470>
```

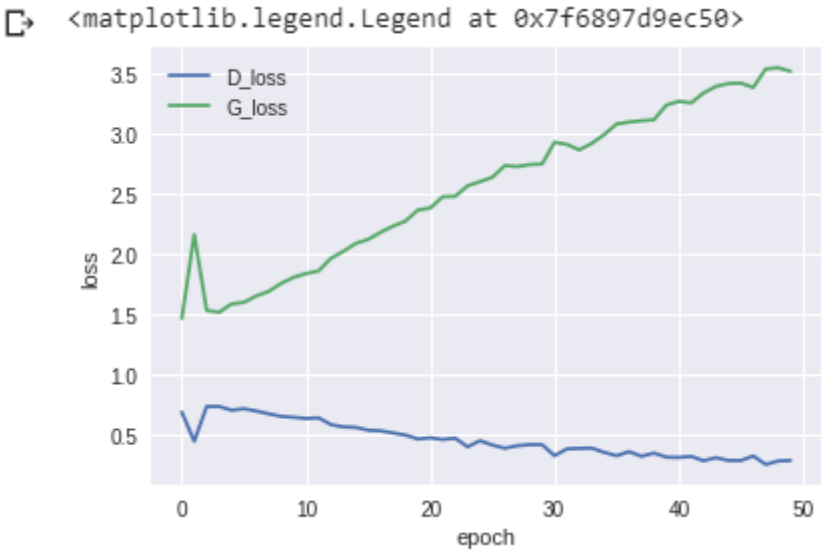


I risultati di output delle immagini del Generatore sono i seguenti:

Epoca 25

50 epoche

Con `batch_size=128`, le loss dei modelli presentano discrete oscillazioni, confermando però una funzione di loss del Discriminatore comunque “vincente”:



Le immagini falsificate dal Generatore sono sempre più fedeli alle immagini reali, nonostante siano comunque immagini nuove:

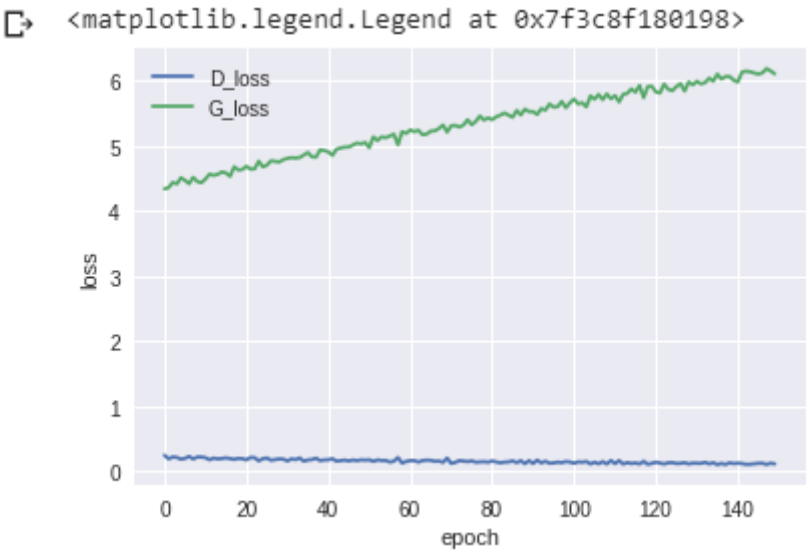
Epoca 50

150 epoche

Incrementando il numero di epoche (ma mantenendo un `batch_size` pari a 128), l'addestramento della rete produce i risultati in figura seguente:

Epoca 150

Il grafico mostra un errore molto basso e costante della loss del Discriminatore, mentre il valore della funzione di loss del Generatore subisce piccole oscillazioni ed un incremento:

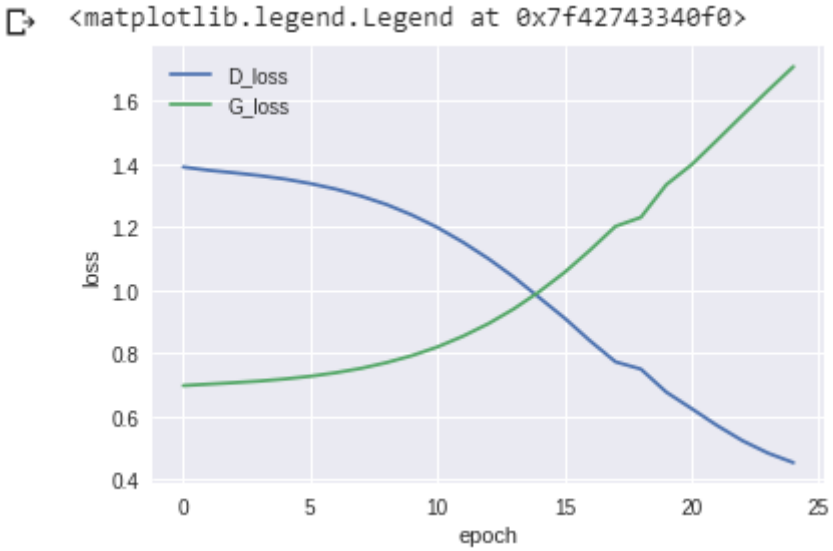


SGD

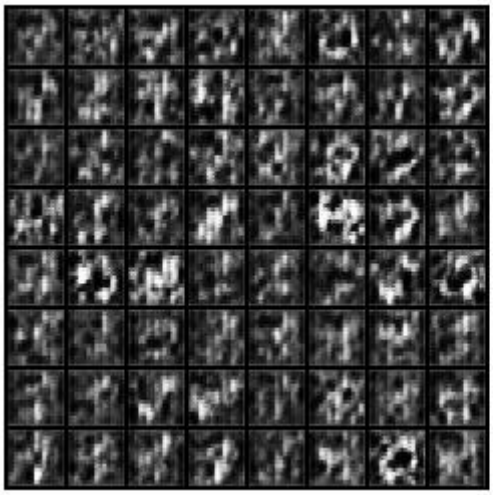
25 epoche

- `batch_size=128`

  Con la funzione di ottimizzazione SGD, i risultati dell'addestramento sono meno performanti:



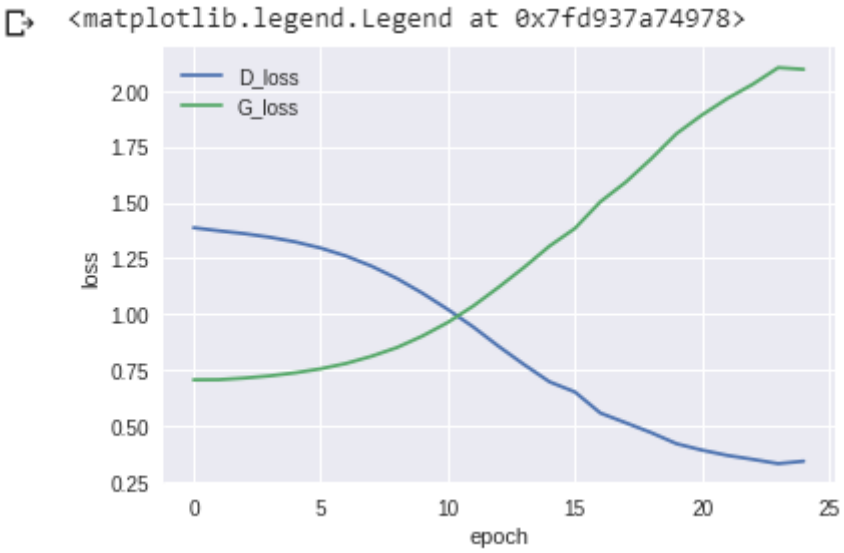
  La funzione di loss del Generatore subisce un notevole aumento, che si riflette sull'output finale:



Epoca 25

- Sperimentazione: *D utilizza ReLU*

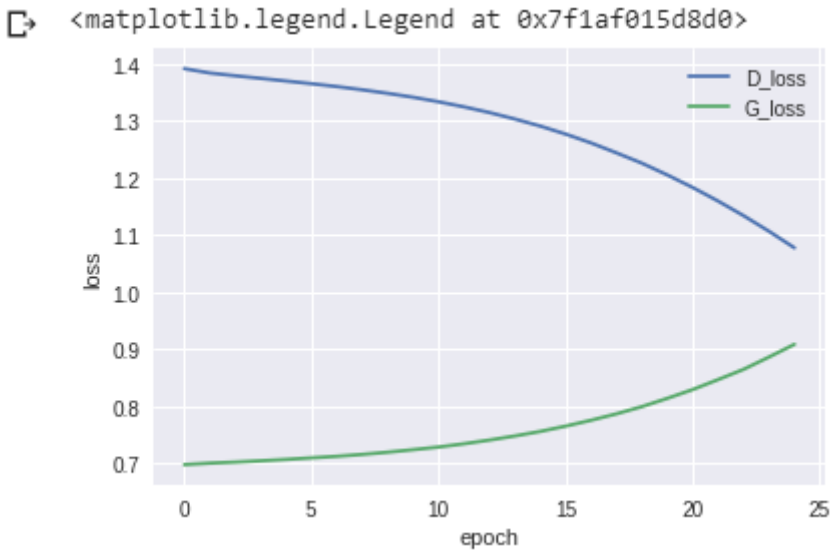
  L'utilizzo di sole ReLU nel Discriminatore aggrava l'addestramento della rete, riscontrando effetti maggiori sulla loss del Generatore:



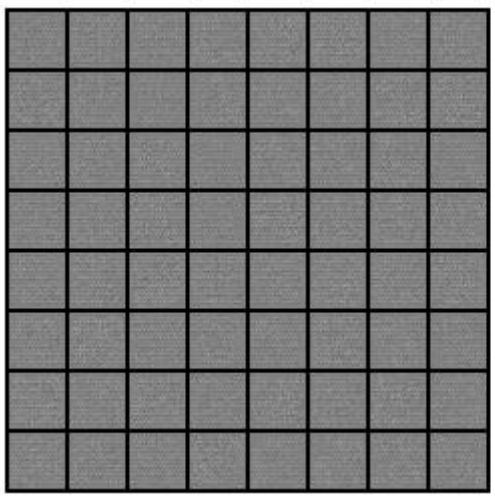
Epoca 25

- `batch_size=256`

  L’andamento delle funzioni di loss mostra, con l’avanzamento delle epoche, una possibile intersezione:



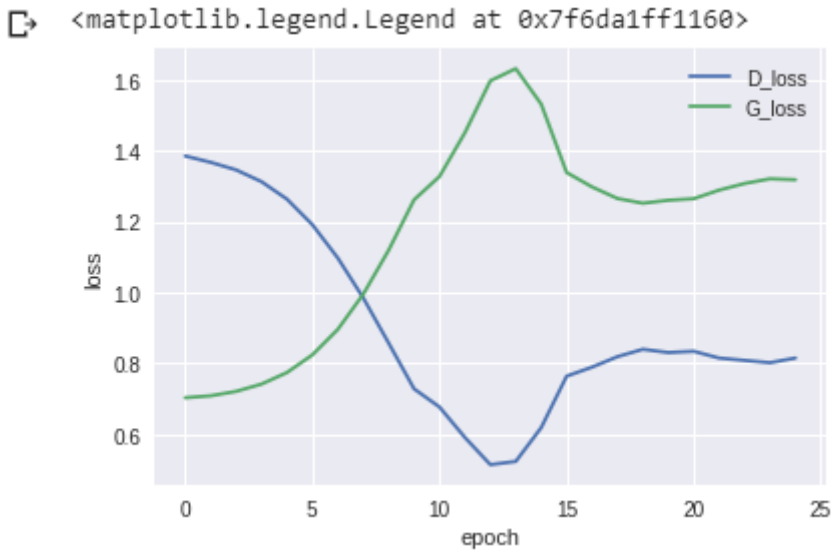
  Infatti, la loss del Generatore si incrementa, dando luogo a immagini falsificate molto simili al rumore iniziale:



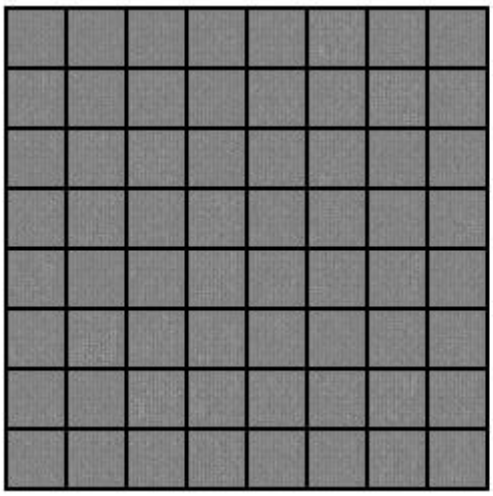
Epoca 25

- `batch_size=64`

  Il grafo mostra un comportamento insolito:



In effetti, il Generatore fa fatica, inizialmente, a realizzare immagini verosimili:



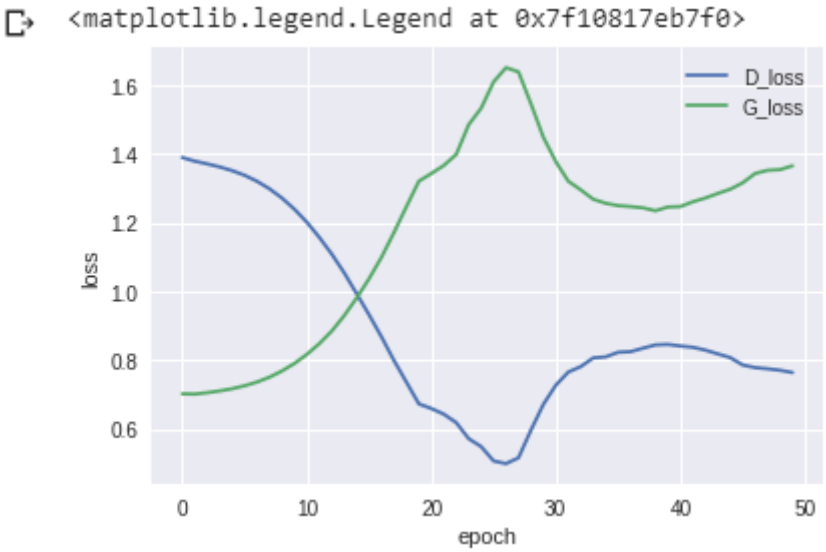
Epoca 10

Alla venticinquesima epoca, sebbene la loss del Generatore sia più alta di quella del Discriminatore, l'addestramento si avvicina ad un buon risultato. Probabilmente, con un numero maggiore di epoche, sarebbe riuscito a perfezionare l'output:

Epoca 25

50 epoche

Reimpostando il *batch_size* a 128 ed il numero di epoche a 50, il comportamento delle funzioni nel grafico assomiglia molto all'esperimento precedente con numero di epoche pari a 25 e `batch_size=64`:

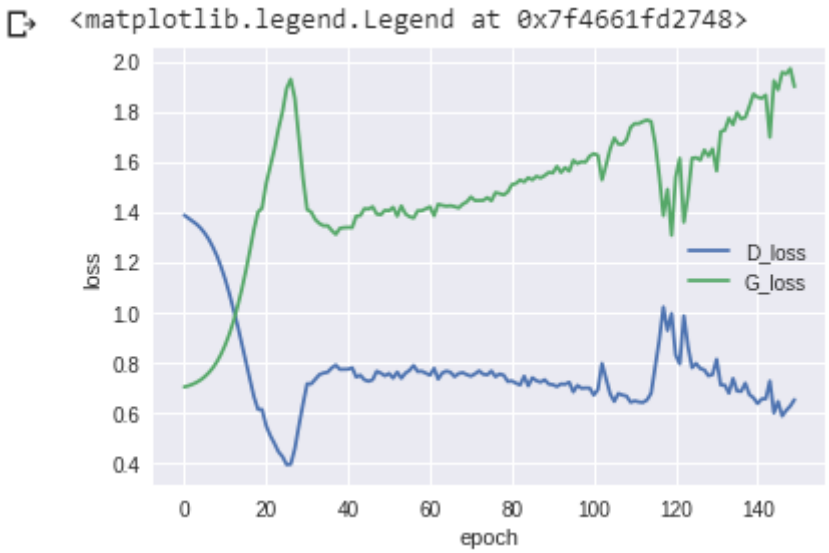


Infatti, anche in questo caso, le immagini falsificate dell'output necessitano di maggiore addestramento, in quanto non molto definite:

Epoca 50

150 epoche

Impostando il parametro *batch_size* a 128 ed aumentando notevolmente il numero di epoche, il Discriminatore inizialmente presenta un tasso di errore abbastanza elevato rispetto al solito, sintomo del fatto che non è in grado di distinguere le immagini vere da quelle false. L'andamento prosegue subendo un notevole decremento, fino a stabilizzarsi. Il comportamento della funzione di loss del Generatore è quasi speculare:



L’output si presenta in tal modo:

Epoca 150

Prove con ulteriori dataset

In questa sezione si presentano gli output relativi alle sperimentazioni con altri dataset.

fashionMNIST

I parametri utilizzati sono i seguenti:

```
# hyperparameters
batch_size = 128
lr = 0.002
z_dim = 62
num_epochs = 25
sample_num = 16
log_dir = './logs'
```

La DCGAN in output stampa immagini di nuovi abiti:

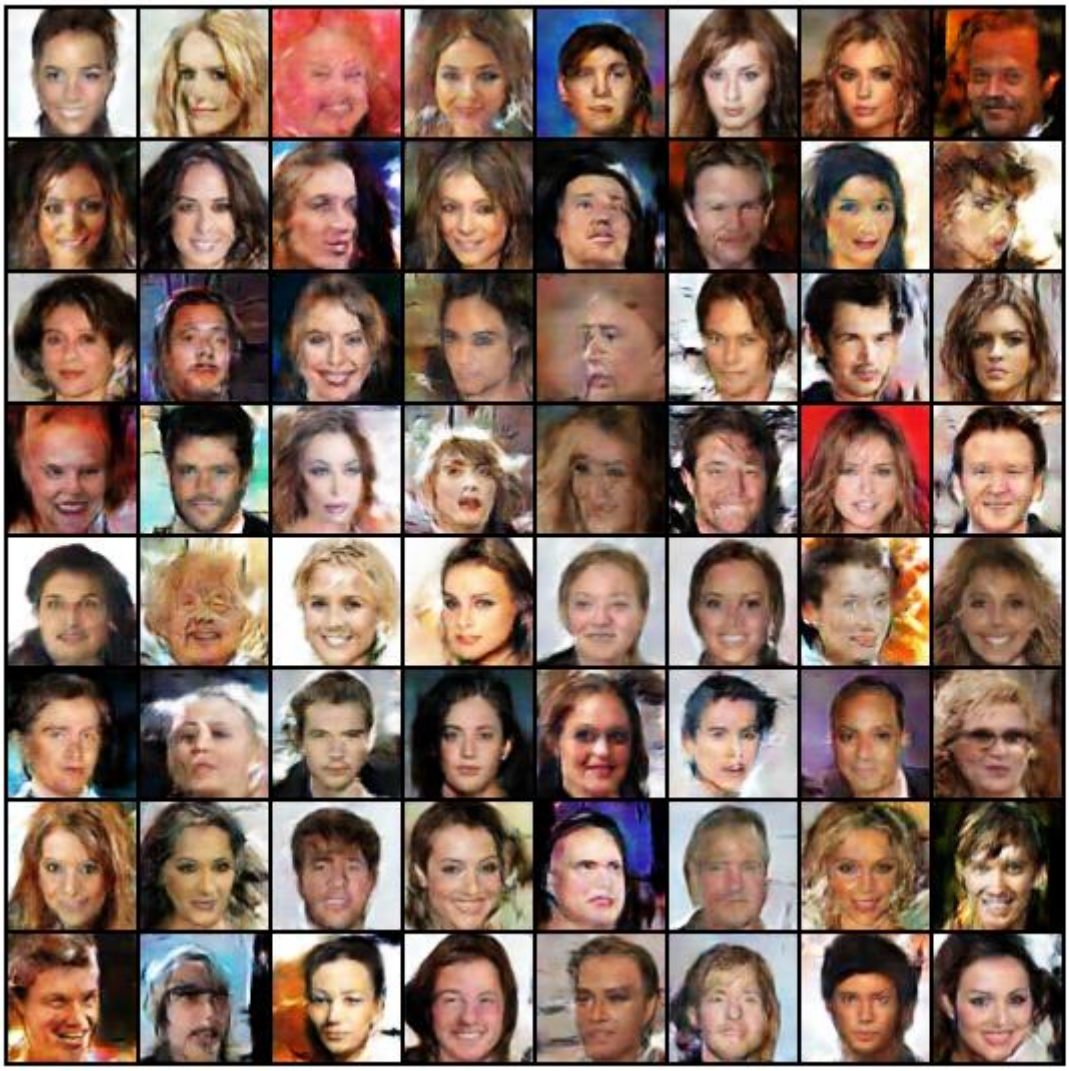
Epoca 1

Epoca 25

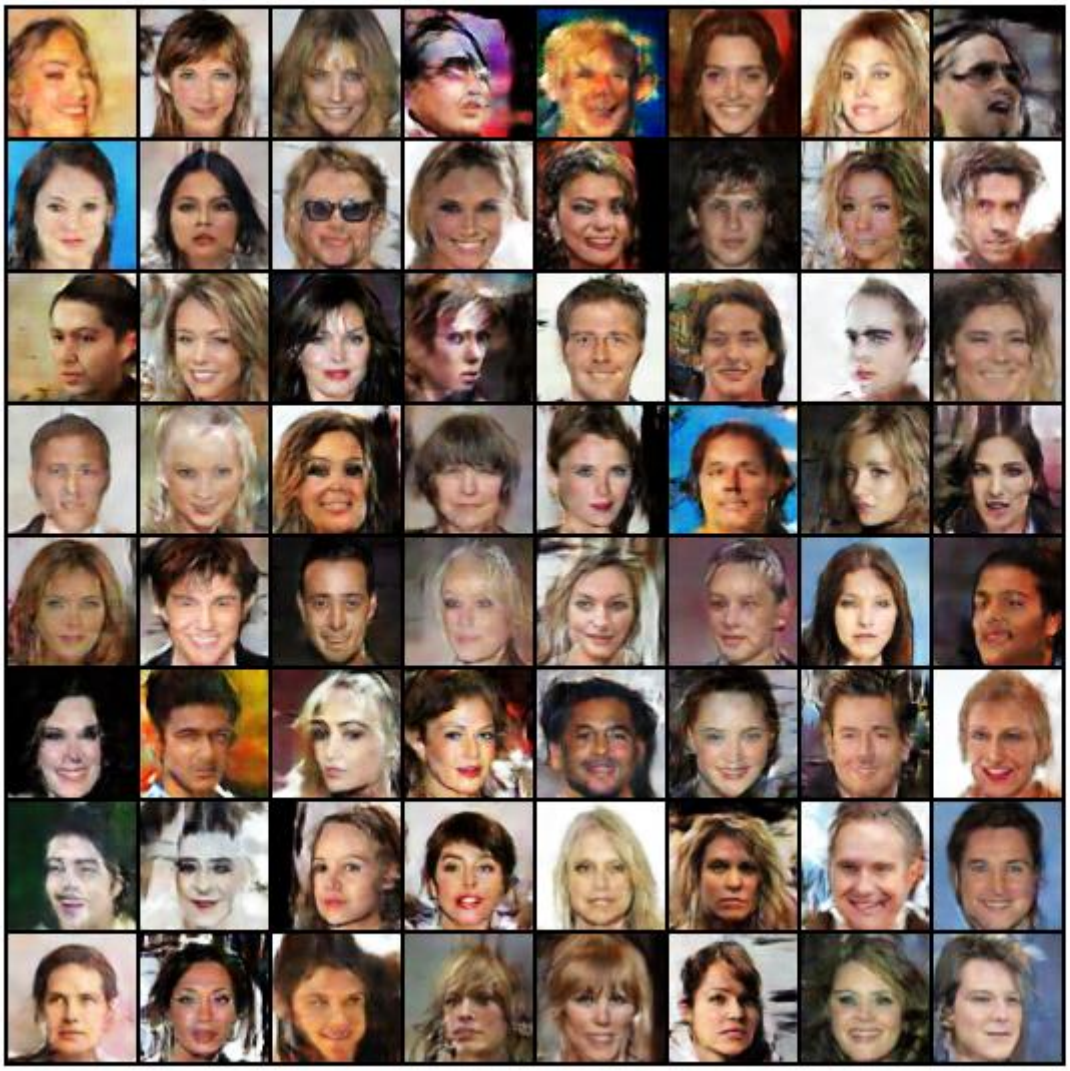
celebA

Si utilizzano i medesimi parametri con il dataset *CelebA*, che contiene immagini a colori di volti di celebrità. Si ottengono dei risultati molto interessanti, nonostante l'elevato tempo impiegato per il training di sole 20 epoche:

Epoca 1



Epoca 10



Epoca 20

Problemi

Molti modelli di GAN soffrono di questi problemi, alcuni dei quali si sono manifestati nella sperimentazione:

- *Non-convergence*: i parametri del modello oscillano e non convergono mai;
- *Mode collapse*: il Generatore collassa producendo una varietà limitata di esempi;
- *Diminished Gradient*: il Discriminatore ha talmente successo che il Generatore non impara nulla.

Osservazioni DCGAN

Le sperimentazioni effettuate sulle Deep Convolutional Generative Adversarial Network (DCGAN) mostrano un'ampia varietà di risultati ottenuti: partendo dalla modifica del *batch size* fino alla variazione delle epoche, si è notata una sensibile variazione del comportamento della rete, che ha evidenziato come il Generatore ed il Discriminatore abbiano ad un certo punto "combattuto" alla pari, sebbene il Discriminatore abbia avuto la meglio.

In particolare, le funzioni di loss ottenute impiegando la funzione di ottimizzazione Adam nella rete mostrano un andamento differente rispetto a quelle ottenute con lo Stochastic Gradient Descent. Difatti, in queste ultime si nota un'intersezione delle funzioni di loss, indice del fatto che entrambi gli "avversari" abbiano appreso con lo stesso tasso di errore.

Inoltre, si nota come utilizzando la funzione SGD, il grafico delle loss mostra per le due curve un'inversione del valore di partenza, in cui la loss del Discriminatore assume inizialmente valori più alti rispetto ai bassi valori di partenza riscontrati utilizzando la funzione Adam.

Conclusioni

In realtà, le funzioni di loss sono molto poco intuitive. Principalmente, Generatore e Discriminatore sono in competizione l'uno contro l'altro, quindi il miglioramento di uno comporta l'aumento della loss dell'altro, finché quest'ultimo non impara a migliorarsi grazie alla loss ricevuta; questo va a rovinare le prestazioni del concorrente, e così via.

Un comportamento che dovrebbe presentarsi abbastanza spesso (dipendentemente dai dati e dall'inizializzazione) è che sia le loss del Discriminatore, sia quelle del Generatore convergano verso alcuni valori permanenti. Spesso le funzioni di loss presentano delle leggere oscillazioni, ma non è un problema: è la prova che il modello sta cercando di migliorarsi.

La convergenza delle loss normalmente significa che la GAN ha trovato un ottimo, ovvero non può migliorare più di così; questo dovrebbe implicare anche che il modello ha avuto un apprendimento corretto. Si noti, inoltre, che i valori numerici di per sé non forniscono molte informazioni riguardo alle prestazioni.

Le analisi svolte in questo progetto hanno portato a diverse conclusioni:

- Se la funzione di loss non converge, non significa necessariamente che il modello non abbia imparato nulla; infatti, se si controllano i campioni generati, spesso sono ben formati;
- Se il modello converge, c'è comunque bisogno di controllare le immagini di output: a volte il Generatore trova solamente uno o pochi esempi che il Discriminatore non riesce a distinguere dai dati originali. Il problema, in questo caso, è che vengono restituite poche immagini, andando a creare un numero limitato di nuovi campioni: *mode collapse*;
- Le GAN Vanilla solitamente sono abbastanza instabili, perciò è preferibile utilizzare le versioni DCGAN, le quali contengono alcune funzionalità (come gli strati convoluzionali e la batch normalization) che contribuiscono alla stabilità della convergenza.
- In ogni caso, come nella maggior parte delle strutture di reti neurali, cambiare i parametri o l'architettura permette di adattare il modello alle proprie esigenze, o ai propri dati. Questo aggiornamento può portare sia ad un miglioramento delle prestazioni del modello, che ad un peggioramento.

-

Fonti

freeCodeCamp – T. Silva (2018), “An intuitive introduction to Generative Adversarial Networks (GANs)” [https://medium.freecodecamp.org/an-intuitive-introduction-to-generative-adversarial-networks-gans-7a2264a81394]

I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio (2014), “Generative Adversarial Networks” [https://arxiv.org/abs/1406.2661]

PyTorch (2018), “PyTorch Documentation” [https://pytorch.org/docs/stable/index.html]

A. Radford, L. Metz, S. Chintala (2016), “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks” [https://arxiv.org/abs/1511.06434]