Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Sviluppo di un tool configurabile per il training di un Adversarial Autoencoder

Relatore:
Chiar.mo Prof.
Andrea Asperti

Presentata da:
Sara Brolli

Sessione II
Anno Accademico 2016/2017

Indice

Introduzione **ii**

1 Deep Learning **7**

1.1 Introduzione alle reti neurali 7

1.1.1 Nodi 8

1.1.2 Connessioni e pesi 9

1.1.3 Livelli 10

1.2 Principali reti neurali 10

1.2.1 Perceptron 10

1.2.2 Multilayer Perceptron 12

1.3 Convolutional Neural Network 15

1.3.1 Convolutional layer 16

1.3.2 Pooling layer 19

1.3.3 Fully connected layer 20

2 Autoencoding e generazione di immagini **21**

2.1 Autoencoder 22

2.2 Variational Autoencoder 23

2.2.1 Ridurre lo spazio di z 26

2.2.2 Massimizzare P(x) 28

2.3 Generative Adversarial Network 31

2.4 Adversarial Autoencoder 34

3 Sviluppo di advae: un tool a linea di comando **39**

3.1 Comandi advae train e advae generate 39

3.2 Struttura della rete neurale 40

3.3 Implementazione supporto a linea di comando 46

4 Risultati e conclusioni **49**

4.0.1 Sviluppi futuri 50

Bibliografia **53**

Introduzione

L'idea di computer intelligenti ha da anni affascinato e forse spaventato l'uomo. Già nel 1968 lo scrittore Philip K. Dick si chiedeva "Do Androids Dream of Electric Sheep?" ma mai come oggi il mondo spinge verso il progresso dell'intelligenza artificiale. Algoritmi che la implementano vengono utilizzati quotidianamente, per esempio comuni applicazioni sono l'elaborazione di previsioni sul mercato finanziario, la generazione di consigli per gli acquisti in base a specifiche preferenze o semplicemente il riconoscimento automatico di volti nelle foto sui social network.

Larghissima parte di tali algoritmi utilizza metodi di Machine Learning. Il termine risale al 1959 e fu coniato da Arthur Samuel[31], il quale scrisse il primo algoritmo in grado di migliorare la propria capacità di giocare a dama, giocando numerose volte contro se stesso.

Alla base delle tecniche di Machine Learning è l'idea di poter descrivere tutto ciò che ci circonda con una funzione matematica sufficientemente complessa. Spesso accade che tale funzione sia talmente complicata che per un uomo risulta impossibile, oppure semplicemente scomodo, codificarla nella sua interezza. Tuttavia, utilizzando tecniche di Machine Learning, è possibile permettere a un computer di calcolare la funzione desiderata. Invece di costruire algoritmi che codifichino ogni regola logica al fine di risolvere un problema, l'elaborazione viene fondata su decisioni fatte dal computer sulla base di esempi forniti dall'uomo.
La macchina deve quindi imparare quali decisioni intraprendere non sulla base di regole logiche ma su esempi concreti.

Cruciale per il progresso dell'intelligenza artificiale fu il contributo di Frank Rosenblatt, che nel 1957 inventò la prima rete neurale, costituita da un solo nodo: il *perceptron*[28]. Esso è un classificatore binario capace di dividere i dati di input in due classi linearmente separabili. Nel loro libro del 1969, Marvin Minsky e Seymour Papert[25] dimostrarono che è impossibile implementare utilizzando un perceptron la funzione logica XOR.

La soluzione ai problemi del perceptron arrivò con il *multilayer perceptron*[5], una rete neurale costituita da più nodi e caratterizzata dall'utilizzo di una funzione non lineare.

La ricerca andò avanti e nel 2006 fu coniato il termine "Deep Learning" con riferimento a reti neurali "profonde" cioè costituite da numerosi livelli di nodi e quindi capaci di imparare funzioni molto complesse. Gli algoritmi aventi reti di questo tipo iniziarono a mostrare prestazioni superiori ad altre tecniche di Machine Learning nella soluzione di numerosi problemi.

Tra i motivi che hanno permesso al Deep Learning di essere così largamente diffuso, nonostante l'innegabilmente elevata mole di computazioni implicata, possiamo identificare: il miglioramento delle tecniche di inizializzazione delle reti, l'uso di funzioni di attivazione più efficienti (la funzione ReLU si sostituisce alla sigmoide), l'utilizzo di tecniche di regolarizzazione come quelle di dropout e la continua ricerca di tecniche per velocizzare le computazioni, per esempio vale la pena ricordare l'utilizzatissimo stochastic gradient descent[22].

Ai progressi riguardanti le tecniche e le strutture utilizzate in campo di Deep Learning si affiancano i seguenti fattori, anch'essi decisivi per la diffusione di tali algoritmi:

1. l'ampia disponibilità di big data, che ha permesso di avere una quantità senza precedenti di dati da analizzare,

2. lo sviluppo di ricerche a proposito di parallel computation, le quali hanno consentito di velocizzare notevolmente l'elaborazione di queste grandi quantitá di informazioni,

3. lo sviluppo dell'architettura CUDA di Nvidia, la quale ha permesso di affiancare alla CPU la capacità di calcolo della GPU.

Le prime applicazioni di queste reti sono state in campo di *image classification*, cioè cercavano di far capire a un computer quale sia il soggetto di una data immagine, oppure di *image recognition*, cioè puntavano a sviluppare la capacità di riconoscere oggetti o soggetti specifici all'interno di un'immagine[7] ma come già accennato in questi anni la quantità di problemi affrontati con l'utilizzo di algoritmi di Deep Learning è cresciuta fortemente.

Modelli generativi

Una interessante applicazione delle reti neurali è stata l'utilizzo delle stesse come modelli generativi: algoritmi capaci di replicare la distribuzione dei dati in input allo scopo di poter poi generare nuovi valori a partire da tale distribuzione.

Solitamente viene analizzato un dataset di immagini e si cerca di imparare la distribuzione associata ai pixel delle immagini stesse, in modo da produrre figure simili a quelle di partenza.

Una delle prime reti neurali capaci di imparare la distribuzione dell'input è chiamata Boltzmann Machine e prende il nome dalla distribuzione di Boltzmann[1]. A partire da essa si sono costruiti altri modelli come quello delle Restricted Boltzmann Machine (RBM), originariamente chiamate Harmonium e inventate da Paul Smolensky nel 1986, le Deep Belief Networks, ideate da Geoffrey E. Hinton nel 2009[11], e le Deep Boltzmann Machine introdotte da Salakhutdinov and Hinton sempre nel 2009[30].

Questa tesi analizza due modelli particolarmente promettenti emersi negli ultimi anni: i Variational Autoencoder [17] e le Generative Adversarial Network[10].

I **Variational Autoencoder** o VAE si ispirano al concetto di Autoencoder: un modello costituito da due reti neurali chiamate encoder e decoder. La prima cerca di codificare il suo input in una forma compressa, mentre la seconda a partire da tale codifica cerca di ricostruire l'input di partenza.

I VAE utilizzano la stessa struttura per la generazione di nuove immagini, simili a quelle appartenenti al training set. L'encoder in questo caso non produce direttamente una codifica per un dato input ma calcola media e varianza di una distribuzione normale. Si preleva un valore da tale distribuzione ed esso viene fatto decodificare dal decoder. Il training consiste nel modificare i parametri di encoder e decoder in modo che il risultato della decodifica così effettuata sia il più simile possibile all'immagine di partenza. Alla fine del training si ha che a partire dalla distribuzione normale con media e varianza prodotte dall'encoder, il decoder sarà capace di produrre immagini simili a quelle appartenenti al training set.

Le **Generative Adversarial Networks** o GAN sono un modello generativo costituito da due reti che vengono allenate congiuntamente, chiamate generator e discriminator.

Le dinamiche tra queste due reti sono come quelle tra un falsario e un investigatore. Il falsario cerca di produrre imitazioni fedeli ad opere d'arte autentiche mentre l'investigatore cerca di distinguere i falsi dagli originali. In questa metafora il falsario rappresenta il generator e l'investigatore rappresenta il discriminator. Il generator accetta in input valori appartenenti ad una distribuzione fissata e da essi cerca di produrre immagini simili a quelle del dataset. Il discriminator cerca di distinguere i dati creati dal generator da quelli appartenenti al dataset. Queste due reti vengono allenate conguintamente nel senso che il discriminator cerca di restituire in output 1 avendo un input appartenente al dataset e di restituire 0 nel caso in cui il suo input sia stato generato dal generator. Quest'ultimo invece cerca di massimizzare la possibilità che il discriminator si sbagli.

VAE e GAN, sebbene molto diversi, possono essere combinati in vari modi[20], [24].

In questa tesi si analizza una loro possibile combinazione: gli **Adversarial Autoencoder** (AAE)[23]. Tale modello si compone di tre reti: encoder, decoder e discriminator.

Il training si compone di due fasi. Nella prima l'encoder e il decoder cercano di minimizzare l'errore di ricostruzione dell'input. Similmente a quanto accade nei Variational Autoencoder l'encoder cerca di produrre codifiche appartenenti a una distribuzione fissata $p(z)$. Nella seconda fase il discriminator cerca di distinguere valori selezionati randomicamente da $p(z)$ da quelli prodotti dall'encoder, mentre l'encoder viene allenato a produrre le codifiche in modo da "ingannare" il discriminator.

In definitiva l'encoder nella prima parte del training si comporta come nei VAE mentre nella seconda si comporta come se fosse il generator di una GAN.

Lo scopo di questo lavoro è quello di fornire un tool facilmente utilizzabile e personalizzabile che implementi una rete neurale di questo tipo.

Nel primo capitolo si introducono le reti neurali, la loro struttura e il loro funzionamento. Inoltre si analizzano i modelli: Perceptron, Multilayer Perceptron e Convolutional Neural Network.

Il secondo capitolo tratta di modelli generativi, in particolare GAN, VAE e AAE con un accenno al concetto di Autoencoder perchè da esso traggono ispirazione VAE e AAE.

Il terzo capitolo espone l'implementazione di un tool a linea di comando che permette il training di un AAE e la generazione di immagini a partire da un modello già allenato, con anche la possibilità di personalizzare le caratteristiche delle reti neurali utilizzate.

Capitolo 1

Deep Learning

Il termine *reti neurali* ovviamente si ispira al mondo biologico e alle sinapsi del cervello, tuttavia le reti neurali artificiali risultano essere maggiormente strutturate rispetto a quelle biologiche: esse si compongono di nodi o neuroni posizionati in livelli, ogni livello riceve l'input dal precedente e manda l'output al successivo, formando così reti più o meno profonde. Da qui il termine *Deep Learning*.

In questo capitolo si espone il concetto di rete neurale analizzandone le componenti (nodi, livelli e connessioni), dopodichè si presentano le reti neurali "di base" e i meccanismi che permettono loro di imparare, infine si introducono le Convolutional Neural Network, reti più complesse e ampiamente usate negli algoritmi di Deep Learning.

1.1 Introduzione alle reti neurali

Le reti neurali sono la componente fondamentale degli algoritmi di Deep Learning. I primi modelli computazionali erano ad ispirazione biologica cioè cercavano di mimare il cervello umano e le sinapsi[2]. Purtroppo, ad oggi, riusciamo a comprendere il funzionamento di tali meccanismi solo in parte, per cui si è passati a delineare una struttura più semplice e ordinata, chiamata

appunto **Artificial Neural Network** (ANN). Essa è composta da neuroni o nodi, connessi da archi e strutturati in livelli ordinati.

Matematicamente una rete neurale può essere vista come una funzione $f : X \to Y$, definita come composizione di altre funzioni $g_i(x)$, le quali possono essere ulteriormente decomposte.
Una composizione comunemente usata è la *nonlinear weighted sum*:

$$f(x) = K\left(\sum_i w_i g_i(x)\right)$$

dove K è qualche funzione fissata, solitamente chiamata *funzione di attivazione*.

Il processo di apprendimento, chiamato anche *trainig* della rete, consiste nella modifica dei pesi w_i in modo da produrre un output coerente con gli esempi forniti in input.

Tutto ciò verrà ulteriormente approfondito nei paragrafi seguenti.

1.1.1 Nodi

I nodi o neuroni sono l'unità elementare su cui si compongono le reti neurali. Ogni nodo riceve uno o più input e produce un solo output, ad eccezione dei nodi di input che non hanno predecessori e i nodi di output che non hanno successori.
Gli input ricevuti hanno associato un peso, ad indicare la diversa importanza di ciascuno nella creazione del risultato finale. Ciò che fa ogni neurone è sommare il prodotto tra i propri input e i rispettivi pesi, dopodichè applica una funzione di attivazione e infine dà il risultato in output.
Matematicamente possiamo vedere un neurone nel seguente modo:

$$y = f(\sum_{i=1}^{n} x_i \cdot w_i + b)$$

Dove:

- x è il vettore di dimensione n dato in input al neurone,

- $w_i \in \mathbb{R}$, con i=1..n, sono i pesi associati ad ogni componente di x,
- $f : \mathbb{R} \to I \subseteq \mathbb{R}$ è la funzione di attivazione,
- b $\in \mathbb{R}$ è il termine di bias o soglia, esso è una costante e può essere utilizzato per modificare il comportamento della funzione di attivazione.

La scelta della funzione di attivazione varia a seconda del compito che il neurone e la rete neurale sono destinati a svolgere. Sono esempi di funzioni di attivazione:

- **sigmoide**
$$f(x) = \frac{1}{1 + e^{-x}}$$
- **gradino**
$$f(x) = \begin{cases} 0 & \text{se } x \leq 0 \\ 1 & \text{se } x > 0 \end{cases}$$
- **tangente iperbolica**
$$tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$
- **ReLu** (Rectified Linear Unit)
$$f(x) = max(0, x)$$

1.1.2 Connessioni e pesi

Una rete neurale si configura come un insieme di connessioni tra neuroni come quelli descritti in precedenza. Ogni connessione trasferisce l'output del neurone i-esimo come input del neurone j-esimo, in questo modo si crea una struttura ordinata e si più dire che il nodo i è il predecessore del nodo j e che il nodo j è il successore del nodo i. Ogni connessione ha associato un peso $w_{i,j}$, il quale verrà modificato durante il training della rete.

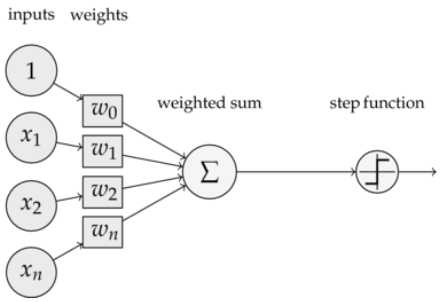

Figura 1.1: Il perceptron: una rete neurale costituita da un solo nodo.

1.1.3 Livelli

I nodi sono organizzati in livelli, ciascun livello può differire dal precedente in termine di numero di nodi.

Supponiamo di avere una rete con n+1 livelli, ciascuno con K_l neuroni. Chiamiamo $z^{(0)}$ il primo livello e K_0 il numero di neuroni presenti in esso. Definiamo i livelli successivi come:

$$z^{(l)} = f(u^{(l)})$$

con $u_j^{(l)} = \sum_{i=1}^{K_{l-1}} w_{i,j}^{(l)} z_i^{(l-1)}$.
Come si può notare ogni livello è costruito a partire dagli output del livello precedente applicando su di essi i pesi e la funzione di attivazione f. L'output della rete sarà quindi $y = z^{(n)}$, con $|y| = K_l$.

1.2 Principali reti neurali

1.2.1 Perceptron

Il perceptron è un algoritmo che permette di dividere i dati in input in due classi linearmente separabili.[9] É stato uno dei primi algoritmi riconducibili al concetto di Artificial Neural Network ed è stato ideato nel 1957 al Cornell Aeronautical Laboratory da Frank Rosenblat.[28]

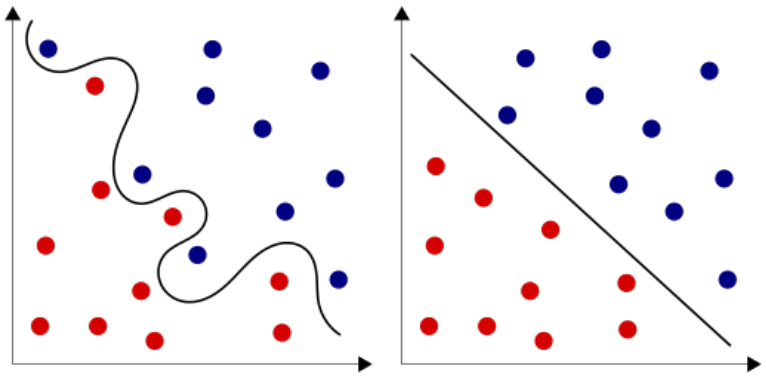

Figura 1.2: A sinistra: classi non linearmente separabili. A destra: classi linearmente separabili

Si tratta di una rete neurale costituita da un solo neurone che calcola la seguente funzione di attivazione

$$f(x) = \begin{cases} 1 & \text{se } w \cdot x + b > 0 \\ 0 & \text{altrimenti} \end{cases}$$

All'inizio tutti i neuroni hanno pesi e bias randomici, dopo alcune iterazioni pesi e bias vengono gradualmente modificati per far sì che l'output della rete si avvicini all'output desiderato.
Per esempio avendo come input un insieme di punti $x^{(i)}$ con $i = 1...n$ e sapendo per ognuno di essi la rispettiva classe di appartenenza $y^{(i)}$, si cerca di minimizzare l'errore:

$$E(w, b) = \sum_{i=1}^{n} (f(x^{(i)}) - y^{(i)})^2$$

facendo in modo che non superi una soglia stabilita in precedenza. Questo tipo di algoritmo è utile in quei problemi dove si richiede di suddividere i dati di input in due sottoinsiemi distinti a patto che essi siano linearmente separabili, infatti l'equazione $y = w \cdot x + b$ definisce un iperpiano.

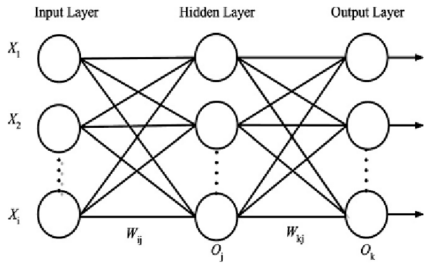

Figura 1.3: Una rete di tipo MLP con un solo hidden layer

1.2.2 Multilayer Perceptron

Le limitazioni dei perceptron possono essere superate costruendo a partire da essi un altro tipo di ANN chiamato Multilayer Perceptron (MLP). Essa consiste in una rete di almeno tre livelli in cui tutti i neuroni, a parte quelli di input, usano una funzione di attivazione non lineare. Se la funzione fosse lineare l'utilizzo di più livelli sarebbe inutile: combinazioni lineari di altre combinazioni lineari sono ancora combinazioni lineari.

Si tratta di una rete completamente connessa, ciò significa che ogni nodo i di un livello è collegato con un certo peso $w_{i,j}$ a tutti i nodi j del livello successivo.

Ora analizziamo il training di un MLP su un dataset composto da n coppie $(x_{(i)}, y_{(i)})$, dove il primo elemento rappresenta l'input e il secondo l'output desiderato corrispondente. Il trainig si compone di due fasi: la *forwward phase* e la *backpropagation phase*.

Forward phase

I dati in questa prima fase fluiscono dal livello di input a quello di output passando in maniera ordinata per gli hidden layer, cioè i livelli intermedi.
Si moltiplica (naturalmente stiamo parlando di prodotto scalare) ogni $x^{(i)}$ dei nodi dell'input layer per il rispettivo peso $w_{(i,j)}$ andando verso il nodo j del livello successivo, dopodichè si applica la funzione di attivazione non lineare

associata al livello di arrivo. Questo processo viene iterato procedendo di livello in livello, fino ad arrivare a quello di output.

Algorithm 1 Forward propagation

1: $z^{(0)} \leftarrow x$
2: **for** $l \leftarrow 1...n$ **do**
3: **for** $j \leftarrow 1...K_l$ **do**
4: $u_j^{(l)} \leftarrow \sum_{i=1}^{K_{l-1}} w_{(i,j)}^{(l)} z_i^{(l-1)}$
5: $z_j^{(l)} \leftarrow f(u_j^{(l)})$
6: **end for**
7: **end for**
8: $y \leftarrow z^{(n)}$

Probabilmente l'output prodotto dalla rete neurale dopo una sola forward phase risulterà diverso da quello previsto. Vogliamo trovare dei valori ottimali per i pesi per fare in modo che dato qualsiasi vettore in input la rete riuscirà a produrre il corretto output. Vogliamo cioè minimizzare l'errore:

$$E = \sum_{i=1}^{n} (y^{(i)} - y_{output}^{(i)})^2$$

Si procede quindi con la fase successiva: la backward phase o backpropagation.

Backward phase o backpropagation

Se volessimo rappresentare in un grafico come l'errore E si comporta rispetto al vettore dei pesi w, otteremmo un grafo come quello in figura 1.4

Come è facile notare, con pesi di valore troppo alto o troppo basso l'errore è elevato, per cui si devono cercare valori ottimali per i pesi in modo da avere l'errore minimo. Per fare questo bisogna spostare w nella direzione verso il punto di minimo della curva. Questa direzione è data dall'opposto del gradiente e questo procedimento viene chiamato *gradient descent*.

Siccome una rete neurale può essere vista come una grande funzione composta, si procede calcolando il gradiente della funzione dell'errore rispetto

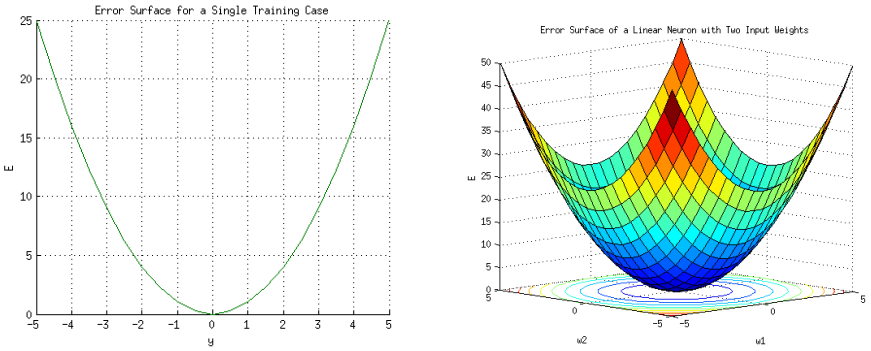

Figura 1.4: Funzione di errore rispetto a w. A sinistra: con w $\in \mathbb{R}$. A destra: con w $\in \mathbb{R}^2$

Algorithm 2 Backpropagation

for $i \leftarrow 1...K_{n-1}$ **do**
  $\delta_j^{(n)} \leftarrow (y_j - \overline{y}_j) f'(u_j^{(n)})$
  for $j \leftarrow 1...K_n$ **do**
    $\Delta w_{i,j}^{(n)} \leftarrow \delta_j^{(n)} z_i^{(n-1)}$
  end for
end for
for $l \leftarrow (n-1)...1$ **do**
  for $j \leftarrow 1...K_l$ **do**
    $\delta_j^{(l)} \leftarrow \sum_{h=1}^{K_l+1} \delta_h^{(l+1)} f'(u_j^{(l)})$
    for $i \leftarrow 1...K_{l-1}$ **do**
      $\Delta w_{i,j}^{(l)} \leftarrow \delta_j^{(l)} z_i^{(l-1)}$
    end for
  end for
end for

a ogni singolo peso $w_{(i,j)}$, utilizzado la regola della derivata della funzione composta (in inglese *chain rule*).

Per la modifica dei pesi si procede ricorsivamente sommando al vettore w il gradiente moltiplicato per un parametro η detto *learning rate*.

Esistono diverse varianti dell'algoritmo di gradient descent [29], per esem-

pio per velocizzare il processo di apprendimento si può stimare il gradiente su un sottoinsieme di elementi invece che sull'intero training set, questo sottoinsieme viene detto *minibatch*. Al passaggio successivo il minibatch considererà un diverso sottoinsieme di elementi in modo tale da considerarli tutti man mano che il trainig va avanti. Una iterazione di questo procedimento viene detta epoca e questo metodo viene detto *Stochastic Gradient Descent*.

Algorithm 3 Stochastic Gradient Descent

$\eta \leftarrow$ learning rate
$n \leftarrow$ numero epoche
$m \leftarrow$ dimesione minibatch
for $i \leftarrow 1...n$ **do**
    Partiziona il training set T in minibatch $B^{(j)}$ di dimensione m
    for $j \leftarrow 1...(|T|/m)$ **do**
        Calcola il gradiente $\widetilde{g}$ su $B^{(j)}$ tramite backpropagation
        Aggiorna i parametri w $\leftarrow$ w - $\eta\widetilde{g}$
    end for
end for

1.3 Convolutional Neural Network

Le reti neurali convoluzionali (CNN o ConvNet) sono variazioni dei percettroni multistrato (MLP) quindi anch'esse si compongono di un livello di input, uno di output e uno o più hidden layer. Ciò che le caratterizza è il fatto che gli hidden layer non sono tutti necessariamente completamente connessi ma possono essere di 3 tipi diversi:

- convolutional,
- pooling,
- fully connected.

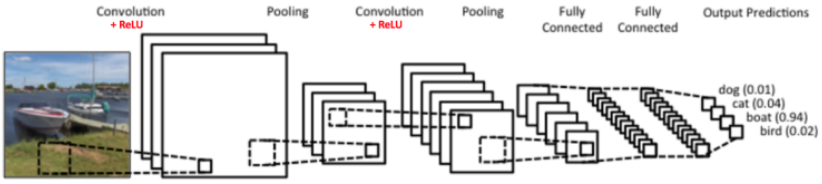

Figura 1.5: Una semplice architettura di tipo CNN

A questi tre tipi di livelli corrispondono tre diverse operazioni che vengono svolte dalla CNN a cui si aggiunge una quarta: l'applicazione di una funzione non lineare. Una semplice CNN si può vedere in figura 1.5

Possiamo ora analizzare queste quattro operazioni e i rispettivi livelli.

1.3.1 Convolutional layer

I livelli convoluzionali stanno alla base del funzionamento delle CNN. Come parametri hanno una serie di filtri che vengono applicati all'immagine, i quali vengono modificati con lo sviluppo del training.

Innanzitutto ricordiamo che un'immagine non è altro che una matrice di pixel di tre dimensioni: larghezza, altezza e profondità. La terza dimensione fa riferimento ai colori, anche chiamati canali e più precisamente sono tre: rosso, verde e blu. I filtri possono avere dimensioni diverse dall'immagine per quanto riguarda larghezza e altezza ma hanno lo stesso numero di canali.

Durante la forward phase, si fa scorrere ogni filtro lungo la lunghezza e l'altezza dell'input e si calcola il prodotto scalare tra le componenti del filtro e l'input stesso. Man mano che questa operazione (chiamata appunto *convoluzione*) procede, viene prodotta una matrice di due dimensioni chiamata *feature map*, contenente il risultato dell'applicazione di un filtro a ogni singolo pixel dell'input di partenza. Come vedremo possono essere applicati anche più filtri nello stesso livello e in questo caso la feature map avrà più di due dimensioni.

Una CNN impara i valori per i filtri durante il training anche se dobbiamo comunque specificare preventivamente parametri quali il numero di filtri da

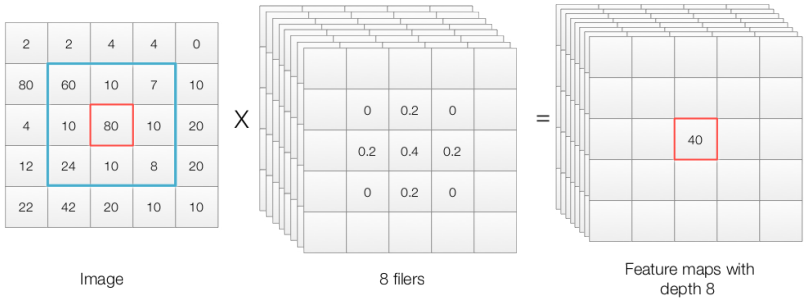

Figura 1.6: Depth: l'applicazione di 8 filtri produce una feature map con depth 8

applicare, la loro dimensione e l'architettura della rete. Più filtri abbiamo, più features potranno essere estratte e migliore sarà la rete a riconoscere immagini mai viste prima.

La dimensione della feaure map (chiamata anche convolved feature) è dipendente da tre parametri, da specificare prima che lo step convoluzionale inizi:

- **depth** : corrisponde al numero di filtri usati in un'operazione di convoluzione. Se per esempio si sceglie di applicare otto filtri allora la feature map avrà profondità 8. (Figura 1.6)

- **stride** : corrisponde al numero di pixel di cui facciamo scorrere il filtro sulla matrice in input. Per esempio se lo stride è 1 allora il filtro si muove un pixel alla volta. Maggiore sarà lo stride e minore sarà la grandezza della feature map. (Figura 1.7)

- **zero-padding** : corrisponde al numero di zeri aggiunti lungo il bordo della matrice di input ed è utile per regolare la dimensione della feature map. Quando lo zero-padding non viene utilizzato si parla di *narrow convolution*, altrimenti di *wide convolution*[15]. (Figura 1.8)

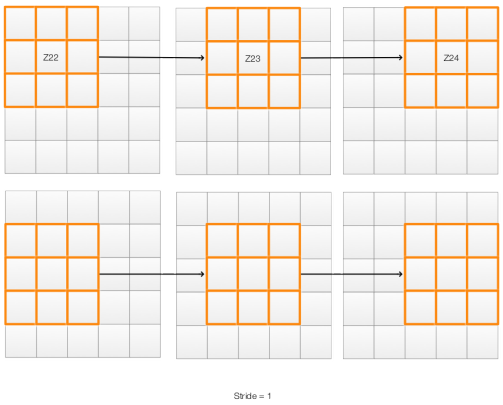

Figura 1.7: Stride: spostamento del filtro con stride pari a uno

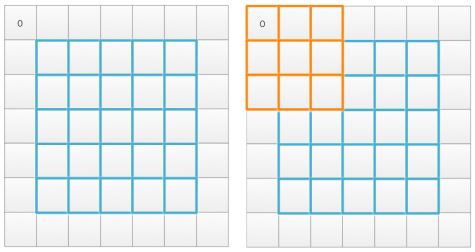

Figura 1.8: Padding: la matrice di input in azzurro ha un bordo (zero-padding) di un pixel

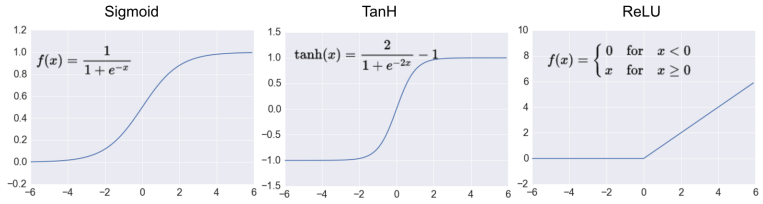

Figura 1.9: Funzioni comunemente usate per l'introduzione della non linearità: sigmoide, tangente iperbolica e ReLU

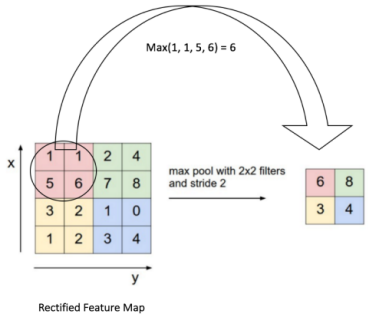

Figura 1.10: Un esempio di max pooling

Non linearità

Alla fine di ogni convoluzione si procede ad applicare una funzione non lineare sulla feature map. L'output di questa funzione è chiamato *rectified feature map*.

1.3.2 Pooling layer

Lo scopo di questo tipo di livello è di diminuire la dimensione delle feature map mantenendo le informazioni più importanti. Esistono vari modi di fare questa trasformazione ma nella pratica il metodo più comunemente usato è il *max pooling* che consiste nel suddividere la feature map in sottoinsiemi e mantenere solo il massimo elemento nel sottoinsieme. Vi è un esempio in figura 1.10.

Questo livello è utile sotto diversi aspetti:

- rende la rappresentazione delle feature più piccola e maneggevole,
- riduce il numero di parametri e di computazioni,
- rende la rete indifferente a modifiche minime dell'input,
- aiuta ad arrivare ad una rappresentazione per l'immagine quasi del tutto equivariante rispetto alle trasformazioni da essa subite (rotazioni, traslazioni etc.)[33].

1.3.3 Fully connected layer

Quest'ultimo tipo di livello si comporta come una rete di tipo MLP, ogni neurone di un livello è connesso a tutti i neuroni del livello successivo. Il suo scopo è quello di classificare l'input in base alle feature map generate nei livelli precedenti.

Capitolo 2

Autoencoding e generazione di immagini

I **modelli generativi** hanno come scopo l'apprendimento di una certa distribuzione $p(x)$, definita su un insieme di datapoint X appartenenti a qualche spazio χ.
Le immagini, per esempio, sono un tipico input per questi modelli: ogni datapoint (immagine) consiste in una moltitudine di pixel e lo scopo del modello è quello di apprendere le relazioni tra essi, come ad esempio che pixel adiacenti hanno spesso lo stesso colore.

Grazie all'apprendimento della distribuzione dei dati è possibile costruirne nuovi con caratteristiche simili a quelle degli originali. Per fare questo possiamo vedere i nostri esempi x come se appartenessero a una distribuzione $p_{data}(x)$ e il nostro obiettivo è quello di imparare un'altra distribuzione $\tilde{p}$ sufficientemente simile a p_{data}.

Molti dei primi approcci al problema si basavano su metodi di inferenza statistica computazionalmente costosi come le Catene di Markov Monte Carlo[1]. Più recentemente è stata dimostrata l'utilità delle reti neurali nell'approssimare funzioni attraverso i meccanismi di backpropagation[18] per cui si è cercato di utilizzare tali meccanismi anche per scopi generativi.

In questo capitolo si analizzano tre modelli:

- i Variational Autoencoder o VAE[17], [27],
- le Generative Adversarial Network o GAN[10],
- gli Adversarial Autoencoder o AAE[23].

Prima di introdurre i modelli sopraccitati si espone il concetto di autoencoder, modello da cui VAE e AAE traggono ispirazione.

2.1 Autoencoder

Un autoencoder è una rete neurale il cui scopo è trovare una codifica di dimensioni ridotte per il suo input e da quella codifica riuscire a ricostruire l'input stesso.
Gli autoencoder sono costituiti dall'unione di due sottoreti:

- l'**encoder**, che calcola la funzione:

$$z = q(x)$$

  Dato un input x l'encoder lo *codifica* in una variabile z, chiamata anche variabile latente. z solitamente ha dimensioni molto inferiori a quelle di x.

- il **decoder**, che calcola la funzione:

$$\bar{x} = p(z)$$

  Sia z la codifica di x prodotta dall'encoder, si cerca di fare in modo che il decoder la *decodifichi* in modo che $\bar{x}$ sia simile a x.

Il training degli autoencoder ha lo scopo di minimizzare l'errore quadratico medio tra l'input e la ricostruzione, in caso di input di dimensione n:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x}_i)^2$$

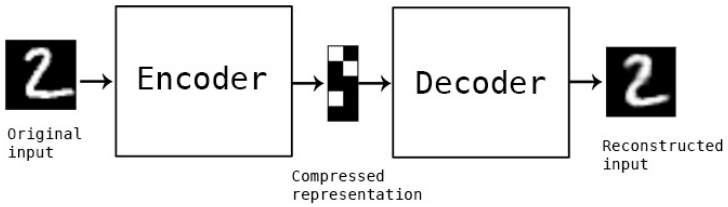

Figura 2.1: L'encoder codifica l'input in una rappresentazione compressa e il decoder restituisce a partire da essa una ricostruzione dell'input. L'unione di queste due reti costituisce un autoencoder.

Lo scopo degli autoencoder non è semplicemente quello di effettuare una sorta di compressione dell'input oppure di imparare una approssimazione della funzione identità ma esistono tecniche che permettono, a partire da un hidden layer di dimensioni ridotte, di forzare il modello a dare la priorità ad alcune proprietà dei dati, dando così origine a diverse rappresentazioni per essi.

In particolare l'idea di autoencoder si è sviluppata nel corso dei decenni[5], [12], [21]: tradizionalmente essi venivano usati per la riduzione delle dimensioni dell'input e l'apprendimento di feature nello stesso, per esempio è stato dimostrato che un autoencoder avente un hidden layer costituito da soli 30 neuroni riesce a ottenere ricostruizioni migliori rispetto a quelle ottenute utilizzando algoritmi di analisi delle componenti principali (PCA)[13]. Più recentemente questi modelli si sono dimostrati efficaci anche nella riduzione del rumore nelle immagini[32]. Invece negli ultimi anni sono stati utilizzati con scopi generativi[4], anche grazie alle analogie tra autoencoder e modelli statistici [17].

2.2 Variational Autoencoder

I variational autoencoder o VAE[17], [27] sono dei modelli il cui funzionamento ricorda quello degli autoencoder classici in quanto anch'essi si strutturano in due parti: encoder e decoder. Diciamo quindi che l'input x

viene *codificato* in una *variabile latente* z (in letteratura chiamata latent vector o latent variable) e che questa rappresentazione viene *decodificata* nello stesso spazio su cui era definito x.

Tuttavia matematicamente il funzionamento dei VAE risulta molto differente da quello dei semplici autoencoder. I variational autoencoder non solo permettono di effettuare la codifica/decodifica dell'input ma anche di **generare nuovi dati** e per fare questo trattano sia la codifica z che la ricostruzione/generazione $\tilde{x}$ come se appartenessero a una certa distribuzione di probabilità. In particolare i VAE sono il risultato della combinazione di Deep Learning e inferenza bayesiana, nel senso che sono costituiti da una rete neurale allenata con l'algoritmo di backpropagation modificato con una tecnica chiamata *riparametrizzazione*. Mentre il Deep Learning si è dimostrato molto efficace nell'approssimazione di funzioni complesse, la statistica bayesiana permette di gestire l'incertezza derivante da una generazione randomica sotto forma di probabilità.

Definiamo:

- l'input alla rete $x \in \chi$ appartenente a un dataset X,
- il vettore z definito su un certo spazio Z,
- la funzione di densità di probabilità $P(z)$ definita su Z,
- $f(z;\theta)$ una famiglia di funzioni deterministiche parametrizzate sul vettore θ definito in uno spazio fissato Θ, dove $f : Z \times \Theta \to \chi$.

f è deterministica ma con z random e θ fissato, $f(z;\theta)$ calcola una variabile randomica appartenente a χ. Vogliamo modificare i parametri θ in modo che, dato $z \sim P(z)$, il risultato di $f(z;\theta)$ sia simile ai dati in X.
Vogliamo cioè massimizzare la probabilità $P(x)$ di ogni x nel training set di essere generato in relazione a z, usando il teorema della probabilità assoluta:

$$P(x) = \int f(z;\theta)P(z)dz \tag{2.1}$$

Si rende esplicita la dipendenza di x da z:

$$P(x) = \int P(x|z;\theta)P(z)dz \tag{2.2}$$

Nei VAE la scelta della distribuzione dell'output spesso ricade sulla Gaussiana, per cui fissiamo:

$$P(x|z;\theta) = N(X|f(z;\theta), \sigma^2 * I)$$

Si mostrerà in seguito che questo permette di usare gradient descent per incrementare $P(x)$ facendo avvicinare $f(z;\theta)$ a x per qualche valore di z.

Innanzitutto si analizza come codificare x in z, in modo che tale codifica catturi le caratteristiche salienti dell'input. Per esempio se x fosse una fotografia rappresentante un volto vorremmo codificare informazioni quali la presenza di un sorriso, il colore della pelle o la forma degli occhi. Queste informazioni possono essere correlate, per esempio se la persona sorride probabilmente avrà gli occhi un po' più chiusi. I variational autoencoder assumono che non debba esistere per esse una codifica fissata ma che z debba appartenere a certa una distribuzione di probabilità. Una funzione comunemente utilizzata è la normale standard $N(0, I)$ dove I è la matrice identità.

Si fissa quindi $P(z) = N(0, I)$.

Questo è possibile perchè ogni distribuzione su d dimensioni può essere generata a partire da un insieme di d variabili distribuite su $N(0, I)$ e mappate attraverso una funzione sufficientemente complicata. Tale funzione, nei VAE, viene imparata durante il training.

Metodo dell'inversione L'adozione della Gaussiana o di altre funzioni continue nei VAE è possibile effettuando un'estensione del metodo dell'inversione, noto anche come trasformazione integrale di probabilità.[8] È stato dimostrato che variabili casuali appartenenti a una distribuzione continua fissata possano essere mappate in variabili casuali aventi una distribuzione uniforme.

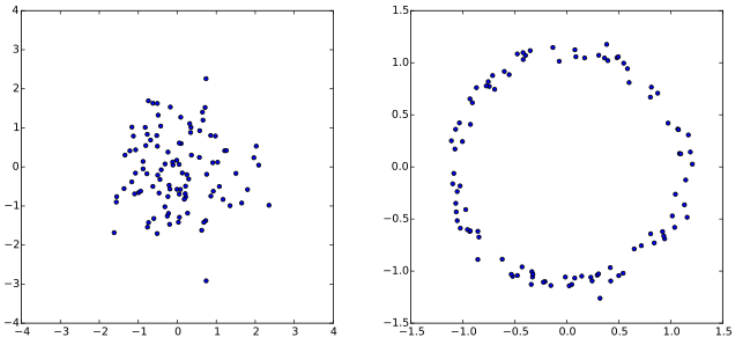

Figura 2.2: Data una variabile random x appartenente a una distribuzione fissata possiamo mapparla su distribuzioni completamente differenti. A sinistra: punti appartenenti a una distribuzione normale. A destra: gli stessi punti mappati attraverso la funzione $g(x) = x/10 + x/||x||$.

Formalmente: avendo una variabile casuale x, con funzione di ripartizione F_x, si definisce la variabile y come $y = F_x(x)$. Allora y ha distribuzione uniforme.
In altre parole, data una variabile casuale uniformemente continua y in $[0, 1]$ e una funzione di ripartizione invertibile F, la variabile casuale $x = F^{-1}(y)$ è distribuita secondo F.

Grazie a questo risultato data una variabile random x appartenente a una distribuzione fissata possiamo mapparla su distribuzioni completamente differenti (figura 2.2). Per distribuzioni su più di una dimensione si calcola la distribuzione marginale su una dimensione e poi si procede con la distribuzione condizionale per ogni dimensione aggiuntiva.

2.2.1 Ridurre lo spazio di z

Osservando l'equazione 2.2 si nota che il calcolo dell'integrale dovrebbe prevedere tutte le possibili configurazioni delle variabili latenti z. Per ovviare al problema della grande complessità di tale operazione si vogliono considerare solo i valori di z che hanno la probabilità più alta di aver generato x e

calcolare $P(x)$ considerando solo essi.

Fissiamo una distribuzione arbitraria $Q(z)$ per z e studiamo la relazione tra $E_{z\sim Q}P(x|z)$ e $P(x)$ a partire dalla divergenza di Kullback-Leiber tra $P(z|x)$ e $Q(z)$.

Definizione 2.1 (Divergenza di Kullback-Leibler[19])**.** Date due distribuzioni continue p e q si definisce **divergenza di Kullback-Leibler**:

$$D(p||q) = \int_{-\infty}^{\infty} p(x) log \frac{p(x)}{q(x)} dx$$

La divergenza di KL permette di misurare la quantità di informazioni persa quando q è usata per approssimare p[6].

Possiamo utilizzarla per definire la differenza tra $P(z|x)$ e $Q(z)$ (per ora la dipendenza di Q da x non ci interessa):

$$D[Q(z)||P(z|x)] = E_{z\sim Q}[logQ(z) - logP(z|x)] \tag{2.3}$$

Si introducono $P(x|z)$ e $P(x)$ applicando la regola di Bayes su $P(z|x)$:

$$D[Q(z)||P(z|x)] = E_{z\sim Q}[logQ(z) - logP(x|z) - logP(z)] + P(x)$$

$$D[Q(z)||P(z|x)] - P(x) = E_{z\sim Q}[logQ(z) - logP(x|z) - logP(z)]$$

$$D[Q(z)||P(z|x)] - P(x) = E_{z\sim Q}[-logP(x|z)] + D[Q(z)||P(z)]$$

$$P(x) - D[Q(z)||P(z|x)] = E_{z\sim Q}[logP(x|z)] - D[Q(z)||P(z)]$$

Supponiamo che i valori di z con alta probabilità di generare x appartengano a una distribuzione $Q(z|x)$, il cui spazio speriamo sia inferiore a quello di $P(z)$.
Introduciamo la dipendenza di Q da x nell'equazione precedente:

$$P(x) - D[Q(z|x)||P(z|x)] = E_{z\sim Q}[logP(x|z)] - D[Q(z|x)|P(z)] \tag{2.4}$$

Nella parte destra dell'equazione 2.4 si può notare la somiglianza con il concetto di autoencoder: è come se Q "codificasse" x in z e P "decodificasse" z in x.

2.2.2 Massimizzare P(x)

Ricordiamo che il nostro obiettivo è quello di massimizzare la probabilità $P(x)$ di ogni x nel training set di essere generato in relazione a z.

Osservando l'equazione 2.4 si nota che nella parte sinistra vogliamo massimizzare $logP(x)$ e nel contempo si vuole definire Q tale che $D[Q(z|x)|P(z)]$ sia minima. $P(z|x)$ non è calcolabile analiticamente ma all'aumentare della complessità di $Q(z|x)$, quindi all'avvicinarsi di quest'ultima a $P(z|x)$, la divergenza di Kullback-Leiber tende a zero. Se accadesse $P(z|x)$ sarebbe calcolabile perchè basterebbe usare $Q(z|x)$ al suo posto. Per massimizzare $logP(x)$ si utilizza gradient descent sulla parte destra dell'equazione 2.4. Perchè questo sia possibile vincoliamo $Q(z|x)$ ad essere una Gaussiana:

$$Q(z|x) = N(z|\mu(X;\theta), \Sigma(X;\theta))$$

μ e Σ sono implementate da reti neurali, sono funzioni deterministiche di parametro θ, il quale viene imparato durante il training. Inoltre per motivi computazionali Σ viene vincolato ad essere una matrice diagonale. Utilizzando la formula per la divergenza di Kullback-Leiber tra due Gaussiane con la stessa dimensione, in questo caso k:

$$D[N(\mu_0, \sigma_0)||N(\mu_1, \sigma_1)] = \frac{1}{2}(tr(\Sigma_1^{-1}\Sigma_0)+(\mu_1-\mu_0)^T\Sigma_1^{-1}(\mu_1-\mu_0)-k+log(\frac{det\Sigma_1}{det\Sigma_0}))$$

si ha che:

$$\begin{aligned} D[Q(z|x)|P(z)] &= D[N(\mu(x), \Sigma(x)||N(0,1)] \\ &= \frac{1}{2}(tr(\Sigma(x)) + (\mu(x))^T(\mu(x)) - k + log(det\Sigma(x))) \end{aligned} \tag{2.5}$$

Continuando a ragionare sull'equazione 2.4, per quanto riguarda $E_{z\sim Q}[logP(x|z)]$, per avere un valore atteso attendibile sarebbe necessario considerare molti valori di z. Per essere più efficienti si fissa un valore per z e si tratta $P(x|z)$ come se fosse un'approssimazione per $E_{z\sim Q}[logP(x|z)]$.

Considerando il fatto che x appartiene al dataset X, l'equazione che veramente vogliamo ottimizzare è:

$$E_{x\sim X}[logP(x)-D[Q(z|x)||P(z|x)]] = E_{x\sim X}[E_{z\sim Q}[logP(x|z)]-D[Q(z|x)||P(z)]] \tag{2.6}$$

Per calcolare il gradiente dell'equazione precedente, è possibile prelevare un singolo valore per x e un singolo valore per z dalla distribuzione $Q(z|x)$ e, utilizzando l'uguaglianza 2.5, calcolare il gradiente di:

$$logP(x|z) - D[Q(z|x)||P(z)] \tag{2.7}$$

Si ripete il calcolo del gradiente di questa funzione per molti valori di x e z e se ne fa la media. All'aumentare del numero di valori considerati il gradiente convergerà a quello dell'equazione 2.6.

Backpropagation e riparametrizzazione

Se non si aggiungesse altro, la rete sarebbe come quella descritta nella parte sinistra della figura 2.3. Con un modello di questo tipo non è possibile fare backpropagation. Un presupposto del gradient descent è che, avendo i parametri della rete fissati, a un certo input corrisponda sempre lo stesso output. È quindi necessario eliminare il campionamento casuale di z.

I variational autoencoder differiscono dagli autoencoder classici anche per questo: l'encoder non produce un vettore di numeri reali. La codifica dell'input produce in questo caso due vettori: uno per la media e uno per la varianza di $Q(z|x)$. Questi parametri dipendono deterministicamente da x e la casualità necessaria per la generazione di immagini inedite viene raggiunta calcolando z nel modo seguente:

$$z = \mu(x) + \Sigma(x) * \epsilon$$

dove $\epsilon \sim N(0, I)$. Per cui la funzione di cui effettivamente si calcola il gradiente è:

$$E_{x\sim X}[E_{\epsilon\sim N(0,I)}[logP(x|z = \mu(x) + \Sigma(x) * \epsilon)] - D[Q(z|x)||P(z)]] \tag{2.8}$$

Si noti che nessuno dei valori attesi dipende dai parametri del modello per cui dati x e ϵ fissati questa funzione è deterministica e continua nei parametri di P e Q. Questo meccanismo è noto come *reparametrization trick* [17] e grazie ad esso è possibile calcolare il gradiente e usare gradient descent per il training.

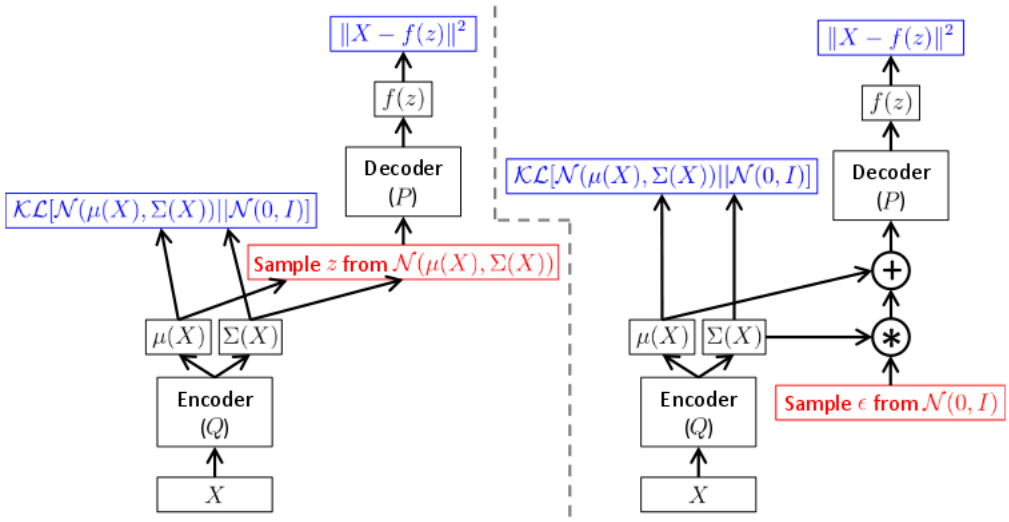

Figura 2.3: Il training di un Variational Autoencoder. $P(x|z)$ è una Gaussiana. In rosso le operazioni che non sono differenziabili, in blu il calcolo dell'errore (*loss*). A sinistra: senza riparametrizzazione. A destra: con riparametrizzazione.

2.3 Generative Adversarial Network

Le Generative Adversarial Network o GAN [10] sono un modello generativo costituito da due diverse reti: alla rete neurale generativa vera e propria si contrappone una rete avversaria. Per comprendere il funzionamento di questa struttura si può immaginare che la rete generatrice sia un falsario e la rete avversaria sia la polizia. L'obiettivo del falsario è quello di produrre contraffazioni che siano indistinguibili dagli originali per la polizia e lo scopo di quest'ultima invece è quello di identificare i falsi. Si vuole far imparare queste due reti l'una con l'altra in modo che il falsario alla fine produca contraffazioni quasi indistinguibili dagli originali.

Per semplicità si tratta il modello fissando le due reti avversarie ad essere Multilayer Perceptron, tuttavia la stessa struttura può essere utilizzata con reti più profonde [26].

Per generare nuove immagini, invece di campionare dati da una distribuzione complessa, l'approccio usato in queste reti è quello di partire da valori appartenenti a una distribuzione semplice oppure da valori randomici. Successivamente essi vengono mappati attraverso una seconda distribuzione che verrà imparata durante il training. Si definisce $G(z;\theta_g)$ una funzione differenziabile calcolata dalla rete generatrice, solitamente chiamata *generator*, con parametri θ_g. Definiamo anche una funzione $D(x;\theta_d)$, avente parametri θ_d e calcolata dalla rete MLP avversaria, solitamente chiamata *discriminator*. $D(x;\theta_d)$ restituisce un solo scalare e rappresenta la probabilità che x provenga dai dati di input piuttosto che dalla rete generatrice. Nella trattazione seguente si omettono θ_g e θ_d per maggiore chiarezza.

Il training del modello ha l'obiettivo di massimizzare la probabilità del discriminator di assegnare 1 a valori provenienti dal training set, invece 0 a quelli prodotti dal generator. D'altra parte si vuole insegnare al generator a minimizzare $log(1-D(G(z)))$. Il training avviene quindi applicando gradient

descent alla seguente espressione:

$$\min_G \max_D V(D,G) = E_{x\sim p_{data(x)}}[logD(x)] + E_{z\sim p_z(z)}[log(1-D(G(z)))] \quad (2.9)$$

Dato che l'errore di ciascuna rete dipente dai parametri dell'altra ma entrambe possono controllare solo i propri parametri questo scenario si configura più come un *gioco* che come una ottimizzazione vera e propria, in particolare come un algoritmo minimax. Gli algoritmi di questo tipo adottano la strategia di minimizzare la massima possibile perdita risultante dalla scelta di un giocatore. Si definisce *equilibrio di Nash* un profilo di strategie, per ciascun giocatore, rispetto al quale nessuno ha l'interesse di essere l'unico a cambiare [14]. Per cui lo scopo dei GAN è trovare un equilibrio di Nash tra G e D.

Con riferimento all'equazione 2.9 il discriminator cerca di agire in modo che $D(x)$ si avvicini a 1 (perchè $x \sim p_{data}$ nel primo valore atteso) e che $D(G(z))$ si avvicini a 0 (perchè $G(z)$ è il risultato del generator). D'altra parte il generator cerca di minimizzare la funzione e fare in modo che $D(G(z))$ si avvicini a 1, cioè è come se volesse indurre il discriminator a credere che $G(z) \sim p_{data}$. Una rappresentazione grafica di questo approccio si può osservare in figura 2.4 mentre l'algoritmo 4 descrive il metodo precisamente.

Il training consiste nell'alternarsi di *gradient ascent* (letteralmente ascesa del gradiente, si procede modificando i pesi andando nella direzione del gradiente) per i parametri del discriminator:

$$\max_{\theta_d}[E_{x\sim p_{data(x)}} logD_{\theta_d}(x) + E_{z\sim p_z(z)} log(1-D_{\theta_d}(G_{\theta_g}(z)))] \quad (2.10)$$

e di *gradient descent* per i parametri del generator:

$$\min_{\theta_g}[E_{z\sim p_z(z)} log(1-D_{\theta_d}(G_{\theta_g}(z)))] \quad (2.11)$$

L'equazione 2.11 potrebbe non permettere a G di imparare abbastanza velocemente. Empiricamente si verifica che il gradiente ha poca pendenza quando il generator produce immagini facilmente distinguibili dalle originali (per esempio nelle prime iterazioni dell'algoritmo) ed è più alto quando

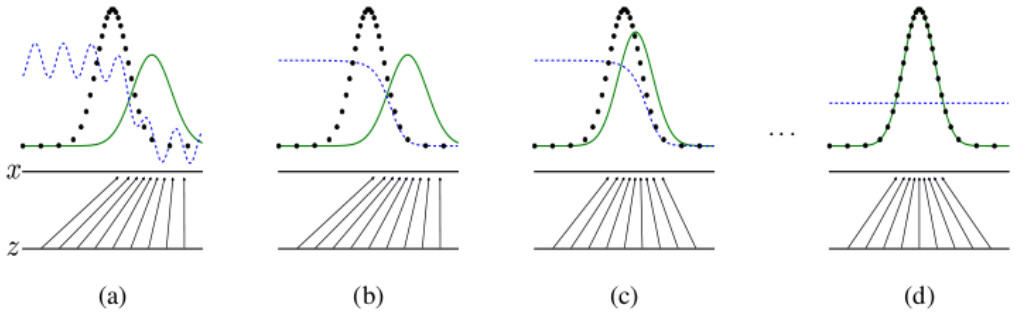

Figura 2.4: Le GAN hanno l'obiettivo di imparare la distribuzione dei dati in input p_{data}, rappresentata come una linea puntinata nera in figura, in modo da generare nuove immagini simili a quelle appartenenti a tale distribuzione. La distribuzione p_g del generator è rappresentata come una linea verde continua e la distribuzione p_d del discriminator come una linea tratteggiata blu. La riga orizzantale nera più in basso rappresenta i valori di z, provenienti da qualche distribuzione p_z, in questo caso una distribuzione uniforme. Questi valori vengono mappati con l'operazione $x = G(z)$. G contrae i valori dove p_g è più densa e li allontana dove p_g ha densità minore. (a) In figura p_g si sta avvicinando a p_{data} e D è un classificatore solo parzialmente accurato. (b) Grazie al ciclo interno dell'algoritmo 4 D migliora, convergendo a $\frac{p_{data}(x)}{p_{data}(x)+p_g(x)}$(si veda la dimostrazione in [10]). (c) Dopo la modifica di G, il gradiente di D ha spostato $G(z)$. (d) Dopo molte iterazioni, se G e D sono sufficientemente complesse, raggiungeranno un punto in cui nessuna delle due può ulteriormente migliorare perchè $p_g = p_{data}$. Il discriminator in questo caso non è più capace di distinguere tra le due distribuzioni.

$D(G(z))$ è vicino a 1, quindi quando il generator è già abbastanza efficace a ingannare il discriminator. Per ovviare al problema, invece di minimizzare la probabilità del discriminator di essere corretto, si massimizza la probabilità che si sbagli. Si mantiene così l'obiettivo di ingannare il discriminator però si ha un gradiente alto quando le immagini prodotte sono molto diverse da quelle appartenenti a p_{data}. Per il training del generator, non si utilizza la funzione 2.11 ma si effettua gradient ascent sulla seguente:

$$\max_{\theta_g}[E_{z\sim p_z(z)} log(D_{\theta_d}(G_{\theta_g}(z)))] \tag{2.12}$$

Nell'algoritmo 4 si alternano k passi in cui si ottimizza D a un solo passo in cui si ottimizza G e questo consente a D di mantenere una posizione vicina alla sua soluzione ottima dato che G cambia abbastanza lentamente. La scelta di k è arbitraria, in [10] è stato usato $k = 1$, ricerche più recenti [3] hanno provato a variare l'algoritmo eliminando la necessità di specificare k.

2.4 Adversarial Autoencoder

Gli adversarial autoencoder (AAE) sono modelli generativi prodotti dall'unione di VAE e GAN. Definiamo:

- x l'input di un autoencoder e z la codifica prodotta a partire da x,
- $p(z)$ la distribuzione che vogliamo imporre alle codifiche,
- $q(z|x)$ la distribuzione imparata dall'encoder e $p(x|z)$ quella imparata dal decoder,
- p_{data} la distribuzione dei dati e $p(x)$ quella del modello.

Si considera la funzione dell'encoder $q(z|x)$ come una distribuzione a posteriori di $q(z)$, la quale viene definita come segue:

$$q(z) = \int_x q(z|x)p_{data}(x)dx \tag{2.13}$$

Si cerca di imporre al modello l'uguaglianza $q(z) = p(z)$.

Algorithm 4 Training di una GAN. Il numero k di iterazioni da applicare al discriminator è un iperparametro.

for numero di iterazioni **do**
  for k iterazioni **do**
    campiona minibatch di m valori $\{z^{(1)}...z^{(m)}\}$ dalla priori $p_g(z)$
    campiona minibatch di m valori $\{x^{(1)}...x^{(m)}\}$ dalla distribuzione dei dati $p_{data}(x)$
    modifica i parametri del discriminator ascendendo il gradiente di:

$$\bigtriangledown_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [logD_{\theta_d}(x^{(i)}) + log(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)})))]$$

  end for
  campiona minibatch di m valori $\{z^{(1)}...z^{(m)}\}$ dalla priori $p_g(z)$
  modifica i parametri del generator ascendendo il gradiente di:

$$\bigtriangledown_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$$

end for

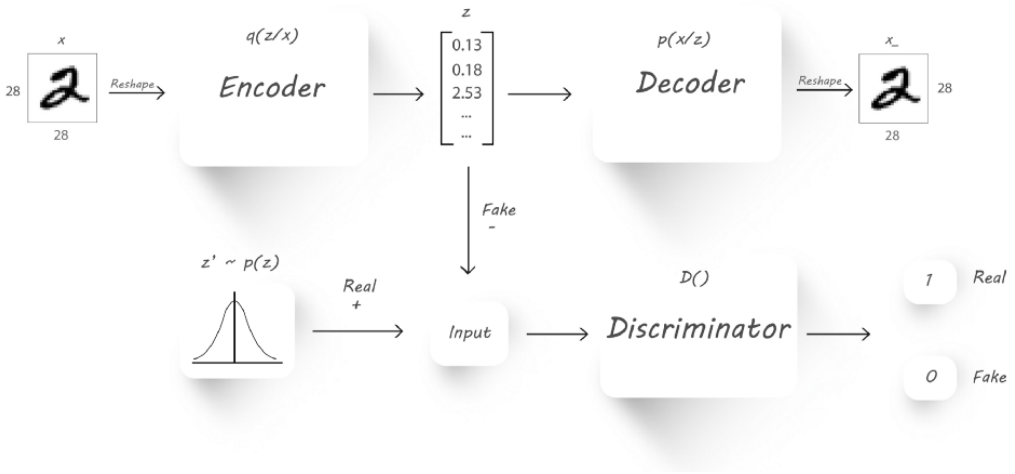

Figura 2.5: Architettura di un adversarial autoencoder. Fonte: https://github.com/Naresh1318/Adversarial_Autoencoder

Ciò che differenzia questa struttura da un VAE è il fatto che ciò che guida $q(z)$ verso $p(z)$ è una rete avversaria. Si considera l'encoder del VAE come il generator di una GAN per cui si può usare un discriminator che cerca di distinguere dati appartententi a $q(z)$ da quelli provenienti da $p(z)$. L'architettura è schematizzata in figura 2.5.

Il training della rete avversaria e dell'autoencoder avviene congiuntamente, usando stochastic gradient descent. In particolare su ogni minibatch si eseguono due fasi:

- la fase di *ricostruzione*, in cui l'encoder e il decoder cercano di minimizzare l'errore di ricostruzione dell'input (l'errore quadratico medio tra il dato di partenza e quello ricostruito). In questa fase il discriminator non viene considerato;

- la fase di *regolarizzazione*, in cui vengono modificati i parametri del discriminator in modo da identificare i dati generati dall'encoder da quelli appartenenti a $p(z)$. Dopodichè, avendo fissato i parametri del discriminator, si aggiornano i parametri del generator/encoder per fare

in modo di "ingannare" il discriminator. In questa fase il decoder non viene considerato.

Dopo il training, il decoder sarà un modello generativo che mappa la priori $p(z)$ su p_{data}.

Si possono scegliere vari tipi di funzioni per l'encoder.

Per esempio sia $q(z|x)$ una funzione deterministica in x. Allora l'encoder si comporta come quello di un autoencoder standard e l'unica fonte di casualità in $q(z)$ proviene da p_{data}.

Invece sia $q(z|x)$ una Gaussiana con media e varianza prodotte dall'encoder ($z_i \sim N(\mu_i(x), \sigma_i(x))$. Allora è possibile utilizzare il *reparametrization trick* [17] per fare backpropagation sull'encoder.

La scelta di diversi tipi di $q(z|x)$ dà luogo a modelli diversi, con tipi di training differenti. I risultati si sono dimostrati simili per ogni tipo di $q(z|x)$ quindi d'ora in avanti si considera $q(z|x)$ una funzione deterministica.

Confronto con VAE I variational autoencoder usano la divergenza di Kullback-Leiber per imporre una distribuzione sulla codifica prodotta dall'encoder. Dall'equazione 2.6 si ottiene:

$$E_{x\sim p_{data}}[-log\, p(x)] < E_{x\sim p_{data}}[E_{z\sim q(z|x)}[-log\, p(x|z)] + D[q(z|x)||p(z)]]$$

$$E_{x\sim p_{data}}[-log\, p(x)] < E_{x\sim p_{data}}[E_{z\sim q(z|x)}[-log\, p(x|z)]] + E_{x\sim p_{data}}[D[q(z|x)||p(z)]]$$

Possiamo considerare $E_{x\sim p_{data}}[E_{z\sim q(z|x)}[-log\, p(x|z)]]$ come l'errore di ricostruzione di un autoencoder e $E_{x\sim p_{data}}[D[q(z|x)||p(z)]]$ come termine di regolarizzazione, senza il quale il modello sarebbe un autoencoder standard. I VAE imparano, grazie a questo secondo termine, a produrre valori per z compatibili con la $p(z)$ scelta. Negli adversarial autoencoder si utilizza invece una procedura di training che incoraggia la distribuzione a posteriori $q(z)$ ad avvicinarsi alla distribuzione $p(z)$ grazie a una rete avversaria.

Confronto con GAN Nelle generative adversarial network la distribuzione p_{data} viene imposta ad ogni pixel dell'output nel generator, invece gli adversarial autoencoders sfruttano il training dell'autoencoder per avvicinare man mano il proprio output ad essa.

Capitolo 3

Sviluppo di advae: un tool a linea di comando

Si presenta un tool a linea di comando per il training di un'adversarial autoencoder. Esso è stato sviluppato interamente in Python, con l'utilizzo di librerie quali Keras[1] e keras_adversarial[2] per l'implementazione delle reti neurali. Per il supporto a linea di comando è stata utilizzata la libreria Click[3].

3.1 Comandi advae train e advae generate

A partire dal comando `advae` si possono eseguire due operazioni: il training di una rete neurale, corrispondente al sottocomando `advae train`, oppure la generazione di immagini a partire da una rete già allenata, corrispondente al sottocomando `advae generate`.

Train Il comando train richiede due argomenti: il path della directory dove si trova il dataset che si vuole dare in input alla rete neurale e il numero di immagini da usare per il training. In caso uno o entrambi questi parametri

[1] https://keras.io/
[2] https://github.com/bstriner/keras-adversarial
[3] http://click.pocoo.org/5/

fossero omessi verrebbe usato il dataset "The Database of Faces" di AT&T ed il numero di immagini utilizzate sarebbe 300.

Si possono specificare anche varie opzioni, le quali modificano il comportamento e la struttura della rete:

`--output-path DIRECTORY` per specificare la directory in cui, alla fine del training, verranno memorizzati l'output della rete e i modelli. Default: `'/output/data_ora'`, dove data e ora sono quelle del momento in cui inizia il training.

`--shape INTERO` per specificare le dimensioni dell'immagine. La rete neurale implementata assume di lavorare con immagini quadrate di lato *shape*. Default: 64.

`--color-channels INTERO` per specificare il numero di canali di colore delle immagini. Default: 3.

`--latent-width INTERO` per specificare la dimensione del risultato dell'encoding (nei capitoli precedenti chiamato z o variabile latente). Default: 256.

`--batch INTERO` dimensione dei minibatch in termini di numero di immagini. Default: 64.

`--epoch INTERO` numero di epoche di training. Default: 200.

Generate Il comando `advae generate` permette di generare nuove immagini a partire da un adversarial autoencoder già allenato in precedenza. Come parametri richiede la directory dov'è salvato il modello da utilizzare e opzionalmente la directory dove salvare l'output.

3.2 Struttura della rete neurale

L'adversarial autoencoder implementato si compone di 3 modelli di base:

- encoder
- decoder

- discriminator

come rappresentato in figura 2.5.

L'**encoder** si compone di quattro livelli convolutivi che applicano 126, 256, 256, 512 filtri 5x5 rispettivamente. Dopo ogni livello convolutivo viene eseguito max pooling con filtri 2x2 e viene applicata la funzione di attivazione:

$$LeakyReLu(x, \alpha) = \begin{cases} x\alpha & \text{se } x < 0 \\ x & \text{se } x \geq 0 \end{cases}$$

con $\alpha = 0.2$. Dopodichè l'output delle convoluzioni viene appiattito in una sola dimensione. A partire dal vettore così ottenuto si costruiscono due diversi livelli completamente connessi, ciascuno avente come output un vettore di 256 elementi (se non specificato diversamente con l'opzione `--latent-width`). I due vettori così ottenuti rappresentano μ e σ di una distribuzione normale. A partire da essi l'output dell'encoder, della stessa dimensione di μ e σ, viene costruito come: $z = \mu + \epsilon \cdot e^{\frac{\sigma}{2}}$

Il **decoder** riceve in input l'output dell'encoder e lo processa con un livello completamente connesso producendo un vettore di 8192 elementi. Esso viene ridimensionato a 4x4x512 e successivamente viene applicata la funzione di attivazione $LeakyReLu$ con $\alpha = 0.2$. Poi si ripetono per tre volte:

- un livello convolutivo[4],
- la funzione $LeakyReLu$ con $\alpha = 0.2$,
- un livello che raddoppia larghezza e altezza (upsampling).

Se la dimensione dell'immagine da costruire è 64x64 allora i precedenti 3 passaggi vengono ripetuti un'ulteriore volta e in questo caso il livello convolutivo applica 64 filtri di dimensione 5x5.

Si conclude con una convoluzione con 3 filtri 5x5 e l'applicazione della sigmoide come funzione di attivazione.

[4] Si applicano 256 filtri di dimensione 5x5 la prima volta, 256 la seconda e 128 la terza.

Il **discriminator** accetta un input della dimensione specificata con l'opzione `--latent-width`). Si compone di:

- un livello completamente connesso, con output di dimensione 256 e funzione di attivazione $LeakyReLu$ con $\alpha = 0.2$,
- un livello completamente connesso, con output di dimensione 128 e funzione di attivazione $LeakyReLu$ con $\alpha = 0.2$,
- un livello completamente connesso, con output di dimensione 128 e funzione di attivazione $LeakyReLu$ con $\alpha = 0.2$,
- un livello completamente connesso con output di dimensione 1 e funzione di attivazione sigmoide.

I tre modelli appena descritti vengono combinati utilizzando la classe **"AdversarialModel"** di keras_adversarial, la quale rappresenta un modello composto da n reti avversarie. In questo caso gli avversari sono due, per semplicità li chiameremo A e D.

A ha come parametri la somma di quelli di encoder e decoder, invece D ha come parametri quelli del discriminator.

Siano:

- z l'output dell'encoder corrispondente all'input x e $zreal$ un valore random appartenente alla normale, della stessa dimensione di z,
- $xpred$ l'output del decoder con input z,
- $yreal$ l'output del discriminator con input $zreal$
- $yfake$ l'output del discriminator con input z.

Per ottimizzare $xpred$ si minimizza l'errore quadratico medio tra $xpred$ e x. Per quanto riguarda $yreal$ e $yfake$ viene utilizzata la binary crossentropy:

$$crossentropy(t, o) = -(t \cdot log(o) + (1 - t) \cdot log(1 - 0))$$

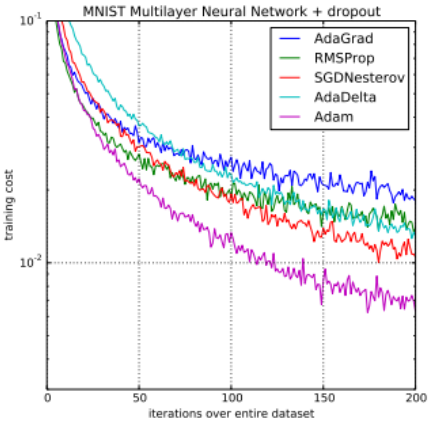

Figura 3.1: Confronto tra Adam e altri metodi di ottimizzazione per il training di un Multilayer Perceptron sul dataset MNIST.

con $t = [1, 0]$ e $o = [yreal, yfake]$).

Per l'aggiornamento dei parametri delle reti avversarie non si utilizza il classico stochastic gradient descent ma un algoritmo più recente, chiamato Adam [16]. Il nome Adam non è un acronimo ma deriva da "adaptive moment estimation". Esso ha la caratteristica di non mantenere lo stesso learning rate per tutta la durata del training ma per ogni parametro della rete viene mantenuto un diverso learning rate che viene moficato man mano che il training procede.

Questo algoritmo viene utilizzato per l'aggiornamento dei parametri di entrambe A e D. In particolare viene fissato un learning rate iniziale di 0.0003 e un learning rate decay di 0.0004 dopo ogni update. È stato scelto questo algoritmo perchè si è dimostrato migliore rispetto ad altri metodi di ottimizzazione per reti neurali. Per esempio in figura 3.1 si può notare la superiorità di Adam nel training di un Multilayer Perceptron.

Encoder		
Tipo livello	Output Shape	N.Param
Input	(64, 64, 3)	0
Convoluzionale	(64, 64, 128)	9728
Max Pooling	(32, 32, 128)	0
LeakyReLu	(32, 32, 128)	0
Convoluzionale	(32, 32, 256)	819456
Max Pooling	(16, 16, 256)	0
LeakyReLu	(16, 16, 256)	0
Convoluzionale	(16, 16, 256)	1638656
Max Pooling	(8, 8, 256)	0
LeakyReLu	(8, 8, 256)	0
Convoluzionale	(8, 8, 512)	3277312
LeakyReLu	(8, 8, 512)	0
Flatten	32768	0
mu	256	8388864
sigma	256	8388864
z	256	0

Tabella 3.1: Struttura dell'encoder in caso di immagini $64 \times 64 \times 3$. Numero totale di parametri: 22,522,880.

Decoder		
Tipo livello	Output Shape	N.Param
Completamente Connesso	8192	2105344
Reshape	(4, 4, 512)	0
LeakyReLu	(4, 4, 512)	0
Convoluzionale	(4, 4, 256)	3277056
LeakyReLu	(4, 4, 256)	0
Upsampling	(8, 8, 256)	0
Convoluzionale	(8, 8, 256)	1638656
LeakyReLu	(8, 8, 256)	0
Upsampling	(16, 16, 256)	0
Convoluzionale	(16, 16, 128)	819328
LeakyReLu	(16, 16, 128)	0
Upsampling	(32, 32, 128)	0
Convoluzionale	(32, 32, 64)	204864
LeakyReLu	(32, 32, 64)	0
Upsampling	(64, 64, 64)	0
Convoluzionale	(64, 64, 3)	4803
Funzione di attivazione	(64, 64, 3)	0

Tabella 3.2: Struttura del decoder in caso di immagini $64 \times 64 \times 3$. Numero totale di parametri: 8,050,051.

Discriminator		
Tipo livello	Output Shape	N.Parametri
Input	256	0
Completamente Connesso	512	131584
LeakyReLu	512	0
Completamente Connesso	256	131328
LeakyReLu	256	0
Completamente Connesso	256	65792
LeakyReLu	256	0
Completamente Connesso	1	257

Tabella 3.3: Struttura del discriminator in caso di immagini $64 \times 64 \times 3$. Numero totale di parametri: 328,961.

3.3 Implementazione supporto a linea di comando

L'implementazione del supporto a linea di comando è stata fatta utilizzando Click, un pacchetto Python. Click permette di creare e organizzare comandi (anche nidificati) con i relativi argomenti ed opzioni, inoltre, per ognuno, produce in automatico una *help page*.

Il comando `advae` è stato definito nel file setup.py con l'uso di `setuptools`:

```
from setuptools import setup

setup(
    name='advae',
    packages=['adversarialAE'],
    # Altre informazioni...
    entry_points={
        'console_scripts': [
            'advae = adversarialAE.cli_tool:initcli'
        ],
    },
```

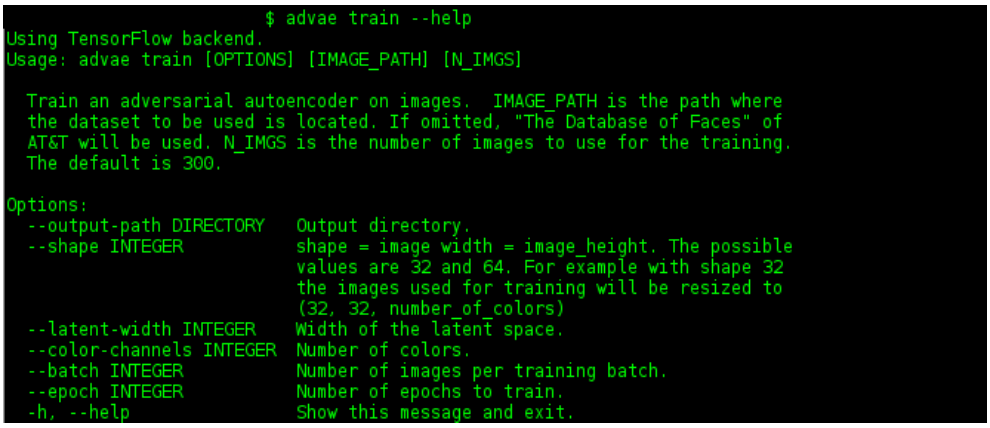

Figura 3.2: L'help page del comando advae train, generata in automatico da Click.

```
)
```

Nel file cli_tool.py i sottocomandi di `advae` vengono definiti con:

```
@click.group(context_settings={'help_option_names':['-h','--help']},
    help='A tool for training an adversarial autoencoder')
def initcli():
   pass
initcli.add_command(train)
initcli.add_command(generate)
```

E per esempio, il sottocomando `train` viene definito nel seguente modo:

```
@click.command()
@click.option('--output-path', default='',
    type=click.Path(resolve_path=False, file_okay=False,
    dir_okay=True), help='Output directory.')
@click.option('--shape', default=64, type = int,
help='shape = image width = image_height. The possible values are
    32 and 64. For example with shape 32 the images used for
    training will be (32, 32, number_of_colors)')
@click.option('--latent-width', default=256, type=int, help="Width
```

```
    of the latent space.")
@click.option('--color-channels', default=3, type=int,
    help='Number of colors.')
@click.option('--batch', default=64, type=int, help="Number of
    images per training batch.")
@click.option('--epoch', default=200, type=int, help="Number of
    epochs to train.")
@click.argument('image-path', default='olivetti',
    type=click.Path(resolve_path=False, file_okay=False,
    dir_okay=True))
@click.argument('n_imgs', default=500)
def train(output_path, shape, latent_width, color_channels, batch,
    epoch, image_path, n_imgs):
   '''Train an adversarial autoencoder on images.
   IMAGE_PATH is the path where the dataset to be used is located.
      If omitted, "The Database of Faces" of AT&$
   N_IMGS is the number of images to use for the training. The
      default is 300.'''
```

A tutto ciò corrisponde la help page mostrata in figura 3.2.

Capitolo 4

Risultati e conclusioni

È stato presentato un tool di facile utilizzo per il training di un Adversarial Autoencoder. Grazie alle opzioni del comando `advae` è possibile personalizzare caratteristiche delle reti che compongono il modello senza intervenire sul codice. Le reti così strutturate non sono vincolanti per il funzionamento del programma, anzi i modelli stessi sono stati posizionati in un modulo a parte in modo che sia facile la loro ristrutturazione.

Per la costruzione dei modelli di encoder, decoder e discriminator sono state utilizzate le API di Keras. Grazie ad esse è possibile salvare modelli già allenati con i rispettivi parametri e ricaricare tali modelli successivamente per continuarne il training oppure per sfruttarne le funzioni. Utilizzando tali API il comando `advae train` produce in output non solo le immagini generate dal decoder ad ogni epoca ma salva anche i modelli che sono stati allenati in file HDF5. In questo modo essi potranno venire riutilizzati in seguito, mantenendo i valori che avevano i pesi alla fine del training. Questo meccanismo è utilizzato da `advae generate`, il quale carica il file corrispondente al decoder e ne analizza il livello iniziale e quello finale. In questo modo si riesce a risalire alla dimensione del latent space e a quella delle immagini da ricostruire. Dopodichè la costruzione di nuove immagini viene fatta a partire da valori delle stesse dimensioni del latent space, appartenenti alla normale standard.

I risultati del programma sono esposti nelle figure seguenti.
Per il training sono stati usati i dataset *Celeba*[1] (in particolare nella versione con immagini tagliate e volti allineati) e *The Database of Faces* di AT&T [2]. Entrambi sono costituiti da immagini di dimensione 64×64. Celeba risulta molto più complicato di The Database of Faces in quanto è costituito da immagini a colori ed alcune di queste presentano sfondi variopinti o addirittura distorti. Il training è stato fatto utilizzando 300 immagini per il training per 500 epoche per entrambi i dataset.

Nelle figure 4.2 e 4.1 in alto è mostrato il risultato prodotto dal decoder a partire da una codifica prodotta dall'encoder. Le immagini appartenenti al dataset sono mostrate nella prima colonna mentre nelle successive si trovano 9 possibili decodifiche. Queste si differenziano le une dalle altre perchè solamente μ e σ prodotti negli ultimi livelli dell'encoder dipendono direttamente dall'input. La codifica z dipende anche dal valore randomico ϵ appartenente alla normale standard, il quale cambia ad ogni esecuzione dell'algoritmo. In basso invece viene mostrato il risultato della generazione di nuove immagini corrispondenti a z non provenienti dall'encoder ma appartenenti alla distribuzione normale standard.

I risultati a partire da Celeba sono lontani dall'essere perfetti, tuttavia quelli a partire da The Database of Faces sono interessanti, soprattutto per quanto riguarda le immagini generate da valori appartenenti alla normale.

4.0.1 Sviluppi futuri

Si può intervenire sul modulo *nets.py* e modificare la struttura delle sottoreti encoder, decoder e discriminator e cercare di ottenere risultati migliori.

Alternativamente si potrebbe ampliare la quantità di reti disponibili per utilizzatori del tool, per esempio permettendo di scegliere il tipo di modello da applicare, creando una nuova opzione usando Click. Sarebbe possibile uti-

[1] http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html
[2] http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html

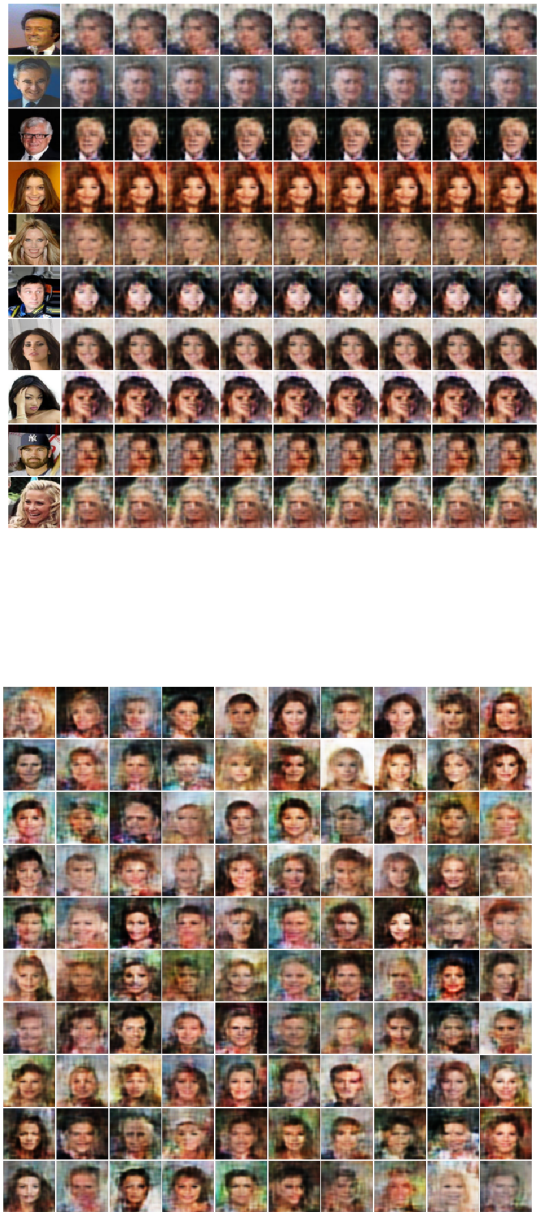

Figura 4.1: L'output dell'AAE descritto in precedenza su immagini appartenenti al dataset Celeba. Sono state eseguite 500 epoche usando 300 immagini per il training. In alto: le immagini appartenenti al training set sono posizionate nella prima colonna. In quelle successive c'è il risultato dell'autoencoding. In basso: immagini generate a partire da valori appartenenti alla distribuzione normale standard.

Figura 4.2: L'output dell'AAE descritto in precedenza su immagini appartenenti al dataset The Database of Faces. Sono state eseguite 500 epoche usando 300 immagini per il training. In alto: le immagini appartenenti al training set sono posizionate nella prima colonna. In quelle successive c'è il risultato dell'autoencoding. In basso: immagini generate a partire da valori appartenenti alla distribuzione normale standard.

lizzare tale opzione per permettere all'utente di scegliere quale rete applicare senza che egli debba intervenire sul codice.

Infine si potrebbe permettere il download di dataset a partire da `advae`, per esempio creando un nuovo sottocomando `advae download`.
Esistono vari dataset scaricabili semplicemente utilizzando funzioni di Keras, come per esempio MNIST e Cifar10[3]. Implementare `advae download` sarebbe semplice, basterebbe chiedere all'utente di fornire il nome di un dataset (da scegliere tra quelli disponibili da Keras) e poi scaricarlo utilizzando le funzioni di Keras stesso.

[3]https://keras.io/datasets/

Bibliografia

[1] D. H. Ackley, G. E. Hinton e T. J. Sejnowski, «A learning algorithm for boltzmann machines», *Cognitive Science*, vol. 9, n. 1, pp. 147–169, 1985.

[2] M. A. Arbib, cur., *The Handbook of Brain Theory and Neural Networks*. MIT Press, 2002.

[3] M. Arjovsky, S. Chintala e L. Bottou, «Wasserstein GAN», *CoRR*, vol. abs/1701.07875, 2017.

[4] Y. Bengio, L. Yao, G. Alain e P. Vincent, «Generalized denoising auto-encoders as generative models», in *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States.*, 2013, pp. 899–907.

[5] H. Bourlard e Y. Kamp, «Auto-association by multilayer perceptrons and singular value decomposition», *Biological Cybernetics*, vol. 59, n. 4, pp. 291–294, 1 set. 1988.

[6] K. P. Burnham e D. R. Anderson, *Model Selection and Multimodel Inference, A Practical Information-Theoretic Approach*. Springer-Verlag New York, 2002.

[7] D. C. Cireşan, U. Meier, J. Masci, L. M. Gambardella e J. Schmidhuber, «Flexible, high performance convolutional neural networks for image classification», in *Proceedings of the Twenty-Second Internatio-*

nal Joint Conference on Artificial Intelligence - Volume Volume Two, AAAI Press, 2011, pp. 1237–1242.

[8] L. Devroye, *Non-Uniform Random Variate Generation*. Springer-Verlag, 1986.

[9] Y. Freund e R. E. Schapire, «Large margin classification using the perceptron algorithm», *Machine Learning*, vol. 37, n. 3, pp. 277–296, 1 dic. 1999.

[10] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville e Y. Bengio, «Generative adversarial networks», *CoRR*, vol. abs/1406.2661, 2014.

[11] G. E. Hinton, «Deep belief networks», *Scholarpedia*, vol. 4, n. 5, p. 5947, 2009.

[12] G. E. Hinton e R. S. Zemel, «Autoencoders, minimum description length and helmholtz free energy», in *Advances in Neural Information Processing Systems 6*, 1994, pp. 3–10.

[13] R. R. Hinton Geoffrey E.; Salakhutdinov, «Reducing the dimensionality of data with neural networks», *Science*, vol. 313, n. 5786, pp. 504–507, lug. 2006.

[14] J. F. N. Jr., «Equilibrium points in n-person games», *Proceedings of the National Academy of Sciences of the United States of America*, vol. 36, n. 1, pp. 48–49, gen. 1950.

[15] N. Kalchbrenner, E. Grefenstette e P. Blunsom, «A convolutional neural network for modelling sentences», *CoRR*, vol. abs/1404.2188, 2014.

[16] D. P. Kingma e J. Ba, «Adam: A method for stochastic optimization», *CoRR*, vol. abs/1412.6980, 2014.

[17] D. P. Kingma e M. Welling, «Auto-encoding variational bayes», *CoRR*, vol. abs/1312.6114, 2013.

[18] A. Krizhevsky, I. Sutskever e G. E. Hinton, «Imagenet classification with deep convolutional neural networks», in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, 2012, pp. 1106–1114.

[19] S. Kullback e R. A. Leibler, «On information and sufficiency», vol. 22, n. 1, pp. 79–86, mar. 1951.

[20] A. B. L. Larsen, S. K. Sønderby e O. Winther, «Autoencoding beyond pixels using a learned similarity metric», *CoRR*, vol. abs/1512.09300, 2015. indirizzo: http://arxiv.org/abs/1512.09300.

[21] Y. LeCun, «Modeles connexionistes de l'apprentissage», 1987.

[22] Y. LeCun, L. Bottou, G. B. Orr e K. Müller, «Efficient backprop», in *Neural Networks: Tricks of the Trade - Second Edition*, 2012, pp. 9–48. DOI: 10.1007/978-3-642-35289-8_3. indirizzo: https://doi.org/10.1007/978-3-642-35289-8_3.

[23] A. Makhzani, J. Shlens, N. Jaitly e I. J. Goodfellow, «Adversarial autoencoders», *CoRR*, vol. abs/1511.05644, 2015.

[24] L. M. Mescheder, S. Nowozin e A. Geiger, «Adversarial variational bayes: unifying variational autoencoders and generative adversarial networks», *CoRR*, vol. abs/1701.04722, 2017.

[25] M. Minsky e S. Papert, *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969.

[26] A. Radford, L. Metz e S. Chintala, «Unsupervised representation learning with deep convolutional generative adversarial networks», *CoRR*, vol. abs/1511.06434, 2015.

[27] D. J. Rezende, S. Mohamed e D. Wierstra, «Stochastic backpropagation and approximate inference in deep generative models», in *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, 2014, pp. 1278–1286.

[28] F. Rosenblatt, *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[29] S. Ruder, «An overview of gradient descent optimization algorithms», *CoRR*, vol. abs/1609.04747, 2016.

[30] R. Salakhutdinov e G. E. Hinton, «Deep boltzmann machines», in *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics, AISTATS 2009, Clearwater Beach, Florida, USA, April 16-18, 2009*, 2009, pp. 448–455.

[31] A. L. Samuel, «Some studies in machine learning using the game of checkers», *IBM Journal of Research and Development*, vol. 3, n. 3, pp. 210–229, 1959.

[32] P. Vincent, H. Larochelle, Y. Bengio e P.-A. Manzagol, «Extracting and composing robust features with denoising autoencoders», 2008, pp. 1096–1103.

[33] D. E. Worrall, S. J. Garbin, D. Turmukhambetov e G. J. Brostow, «Harmonic networks: deep translation and rotation equivariance», *CoRR*, vol. abs/1612.04642,