Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

RIPRODUZIONE ED ANALISI DI UN MODELLO GENERATIVO PER LA CREAZIONE DI IMMAGINI VETTORIALI

REPRODUCTION AND ANALYSIS OF A GENERATIVE MODEL FOR VECTOR IMAGES CREATION

GIULIANO GAMBACORTA

Relatore: *Paolo Frasconi*

Anno Accademico 2016-2017

INDICE

ELENCO DELLE FIGURE

"What I cannot create, I do not understand."
— Richard Feynman

1

INTRODUZIONE

1.1 PREFAZIONE

Lo scopo di questo elaborato è la riproduzione e lo studio di *sketch-rnn* [1]: una rete neurale in grado di generare schizzi di semplici oggetti, composti da sequenze di tratti, appresi da un dataset di disegni creati da esseri umani. Il dataset è costantemente ampliato tramite *Quick Draw!* [2], un gioco online in cui agli utenti viene chiesto di riprodurre alcuni oggetti entro 20 secondi, che al momento attuale costituisce la più vasta collezione di disegni al mondo. In questo lavoro viene proposto un prototipo in *Keras* [3], un framework che a sua volta poggia su *tensorflow* [4], che è la libreria software utilizzata per il lavoro originale.

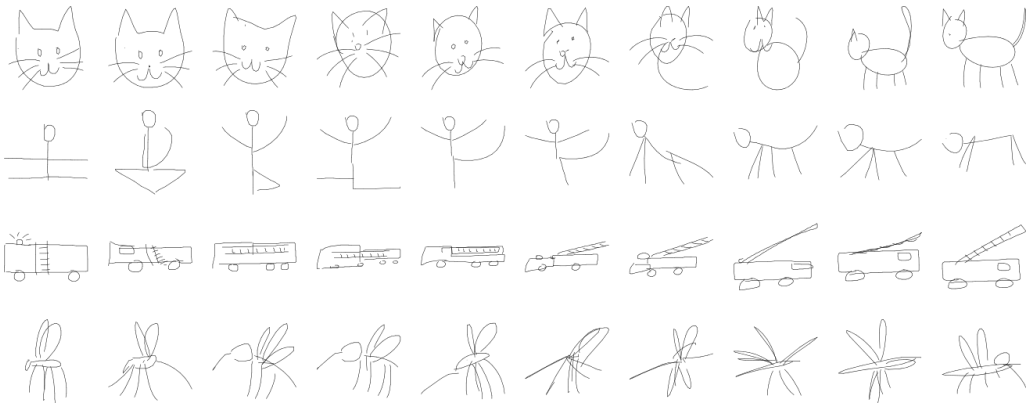


Figura 1: Interpolazioni nello spazio di latenza di immagini vettoriali generate dal modello.

1.2 STATO DELL'ARTE

Negli ultimi anni la generazione di immagini attraverso l'uso di reti neurali ha avuto ampia diffusione, fra i modelli più importanti possiamo

citare: *Generative Adversarial Networks (GANs)* [5], *Variational Inference (VI)* [6] e *Autoregressive Density Estimation (AR)* [7]. Il limite della maggior parte di questi algoritmi consiste nel fatto che lavorano con figure in pixel a bassa risoluzione, a differenza degli animali complessi che, piuttosto che vedere il mondo come una griglia di pixel, astraggono concetti per rappresentare ciò che osservano. Allo stesso modo degli esseri umani, che fin da piccoli imparano a riportare le proprie idee attraverso una sequenza di tratti su un foglio, questo modello generativo apprende da, e produce, immagini vettoriali.

L'obbiettivo è di addestrare una macchina a riprodurre ed astrarre concetti, in maniera analoga a come farebbe una persona. Ciò può avere numerose applicazioni in campo didattico come artistico, ad esempio assistendo il processo creativo, così come l'analisi della rappresentazione prodotta può offrire spunti di ricerca.

1.3 RETI NEURALI

Per spiegare le reti neurali e il Deep Learning si possono usare diversi approcci: si può ad esempio seguire il corso storico, introducendo il concetto di *Percettrone*, passando ai Percettroni Multi-Strato ed ai primi metodi di ottimizzazione che furono applicati a questi modelli. Un'altra possibilità consiste nel partire da un punto di vista stocastico, definendo una regressione logistica che implica in modo naturale la minimizzazione di una *loss function*. Ciò permette la definizione del concetto stesso di loss function e di come modificare dei parametri per ottenere una soluzione migliore, cosa che si ricollega perfettamente al concetto di Percettrone Multi-Strato come classificatore di una regressione logistica.

Di seguito verrà proposta la spiegazione di alcuni modelli fondamentali del Deep Learning, che saranno considerati come "mattoni" costitutivi della rete neurale studiata in questo progetto. Ciò aiuterà a comprendere agevolmente l'implementazione realizzata nel codice, seguendo un punto di vista coerente con le piattaforme più diffuse.

1.4 RETI DENSAMENTE CONNESSE

Un neurone artificiale consiste in una funzione matematica che costituisce l'unità computazionale di base di una rete neurale artificiale.

Come si nota dalla figura 2 il neurone artificiale riceve n input pesati $(w_1x_1, ..., w_nx_n)$ e un bias b. Successivamente li somma e applica una fun-

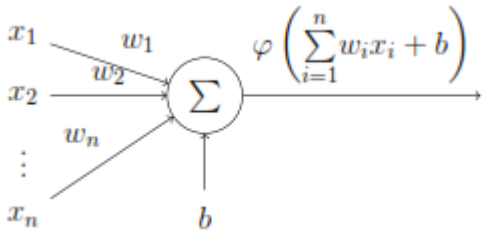


Figura 2: Rappresentazione di un neurone artificiale.

zione non lineare nota come *activation function* φ per generare l'output. In sostanza calcolare l'output di un singolo neurone corrisponde a formare una combinazione lineare dei suoi input pesati e poi passarla ad una funzione di attivazione.

Per ragioni storiche, nel caso particolare in cui la funzione di attivazione consista di una funzione con una soglia lineare (e output binario), il neurone è detto *Percettrone*.

Da qui in poi, quando ci riferiremo ad un *layer* (strato) di una rete neurale staremo parlando di un raggruppamento di neuroni che formano, nello specifico, una colonna nel grafo della figura 2, ovvero neuroni che si trovano allo stesso livello di profondità nella rete. Inoltre, ogni layer che si trova fra quello di input e quello di output verrà chiamato *hidden layer*.

Come detto in precedenza, un Percettrone Multi-Strato (MLP da Multi-Layer Perceptron) può essere visto come un classificatore a regressione logistica dove l'input è trasformato utilizzando una trasformazione non lineare appresa $h(\mathbf{x})$ che costituisce l'hidden layer della nostra rete neurale (come in figura 2). Questa trasformazione proietta l'input in uno spazio dove diventa linearmente separabile.

Questa computazione eseguita da una rete neurale multi-Strato, con un singolo hidden layer con una funzione di attivazione non lineare per elementi φ_i sul vettore di input $\mathbf{x}$ e l'hidden output può essere scritta, in forma matriciale, come $\mathbf{y} = \varphi_2(W_2\varphi_1(W_1\mathbf{x} + \mathbf{b_1}) + \mathbf{b_2})$ dove $W_i, \mathbf{b_i}$ sono la matrice dei pesi e il vettore del bias dell'i-esimo layer.

Seguendo il teorema di approssimazione universale [11] possiamo affermare che una rete neurale con un singolo hidden layer contenente un numero finito di neuroni (ovvero un MLP) è sufficiente per approssimare qualunque funzione continua su un sottoinsieme compatto di $\mathbb{R}^n$.

Alcuni sinonimi rilevanti utilizzati al posto di reti neurali Multi-Strato sono *dense layers*, *fully-connected layers* o, meno diffuso, *inner product layers*.

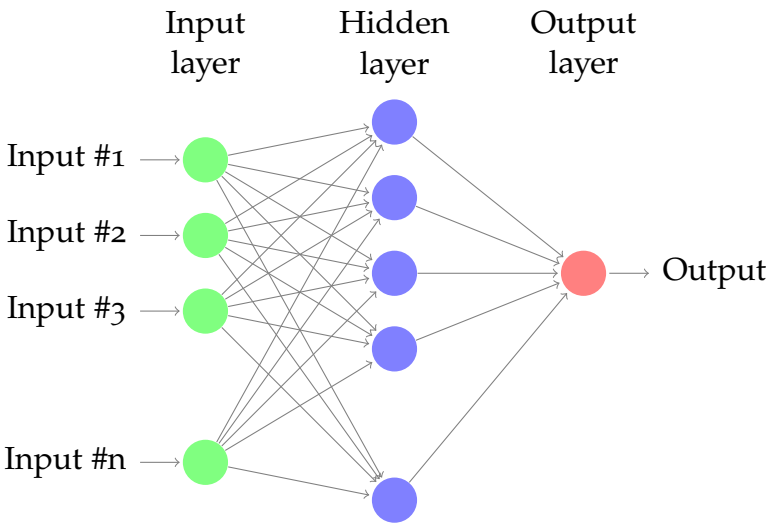


Figura 3: Rappresentazione di un modello Multi-Strato.

1.5 RETI RICORRENTI

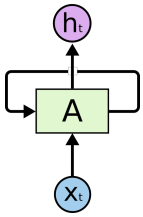


Figura 4: Una semplice RNN (Recurrent Neural Network).

Il cervello umano, nello specifico il lobo frontale, è in grado di elaborare conseguenze future risultanti da azioni nel presente, ha la capacità di selezionare fra buone e cattive azioni (o fra migliori e ideali) e può determinare somiglianze e differenze fra oggetti ed eventi.

Molti problemi per essere risolti necessitano di un certo grado di conoscenza pregressa. Un esempio può riguardare le variazioni della luce di un semaforo: se, per esempio, si osserva che nel momento attuale la luce accesa è quella gialla, lo scopo sarebbe quello di sapere quale sarà la prossima ad accendersi. Le diverse posizioni delle luci in un semaforo sono irrilevanti: ciò che interessa sapere è quale colore apparirà, sapendo che quello appena apparso è il giallo. Nella maggior parte delle città è noto che la risposta sarebbe il rosso. Per rispondere a questa domanda è venuta in aiuto l'esperienza ma se si provasse a risolvere il problema utilizzando una rete neurale come quella proposta in precedenza non

si otterrebbe una risposta soddisfacente. Ciò accade perché la soluzione presenta una dipendenza temporale, che corrisponde al metodo attraverso cui un essere umano apprende a risolvere problemi: analizzando sequenze di eventi.

Per trovare una soluzione a questo problema, com'è uso nel campo dell'Intelligenza Artificiale, si parte ancora una volta dai modelli basati sui principi biologici per elaborare una classe di reti neurali artificiali, in cui le connessioni fra le unità formano un ciclo orientato, dette Reti Neurali Ricorrenti (RNN da Recurrent Neural Networks, Fig. 4). Queste connessioni creano uno stato interno della rete che le permette di esibire un comportamento dinamico nel tempo.

Per quanto riguarda le reti neurali non ricorrenti sono già state presentate molte conoscenze utili allo scopo di ottenere buoni parametri, di conseguenza si rende necessario trovare un modo di trasferire le potenzialità già discusse anche su questo nuovo tipo di reti neurali. Un'idea primitiva sarebbe quella di elaborare la rete ricorrente attraverso copie molteplici di una singola rete, come si vede in fig. 5, trasferendo le informazioni dall'hidden layer A alla copia successiva per realizzare una forma di memorizzazione.

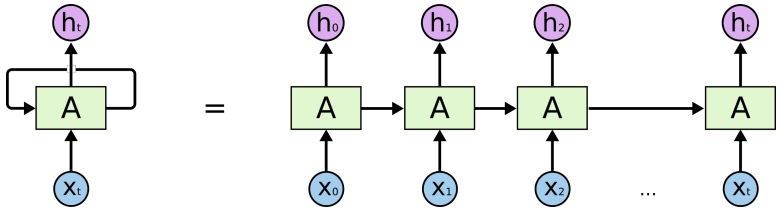
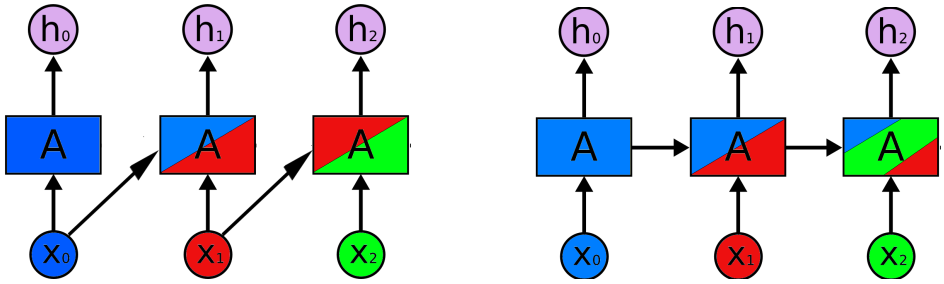


Figura 5: Una RNN dispiegata lungo la linea temporale.

Tuttavia ci potrebbero essere diversi modi di intendere la memoria e di combinare l'input del *time-step* attuale con le informazioni ottenute dal precedente. Si prendono in considerazione due esempi: nel primo scegliamo di ottenere le informazioni precedenti conservando l'input, nel secondo l'hidden layer del time-step passato, ottenendo risultati completamente diversi. in fig. 6 sono riportati i due esempi sopra descritti, dove ogni colore rappresenta gli effetti sulla memoria dell'hidden layer A.

Come si può vedere in fig. 6(a), usando la ricorrenza dell'input viene ricordato solo l'attuale input e il precedente, invece nel caso della ricorrenza dell'hidden layer (fig. 6(b)) viene ricordata una mistura di tutti gli input precedenti. Per comprendere perché la seconda ipotesi è migliore si può usare questo esempio: si immagini di voler dedurre la parola



(a) Visualizzazione della memoria usando la ricorrenza dell'input

(b) Visualizzazione della memoria usando la ricorrenza dell'hidden layer

Figura 6: Due diversi metodi di implementazione della memoria in una RNN

seguente a *"I love"* ("io amo") e che il testo contega le affermazioni *"I love you"* ("ti amo") e *"I love carrots"* ("amo le carote"). Se lo scopo è predire la decisione che verrà presa fra queste due opzioni e non sono note altre informazioni al di là dell'input (nel caso della ricorrenza dell'input), la rete neurale non avrà abbastanza informazioni per decidere. Viceversa, se la rete neurale possiede informazioni riguardanti il contesto, ad esempio se precedentemente si è parlato di cibo o di pasti, la sceltà diventerà più chiara. In teoria la ricorrenza dell'hidden layer può essere interpretata come un tentativo di ricordare ogni informazione con cui la rete neurale è entrata in contatto ma in pratica vige un limite tutt'al più di qualche passo.

Un'altra caratteristica delle RNN, che le avvantaggia ulteriormente rispetto alle MLN, è la versatilità. Le RNN, infatti, sono in grado di operare su sequenze di vettori, a differenza delle MLN o delle CNN (che non saranno trattate in questo elaborato) che operano su vettori di dimensioni fissate, così come fissato è il numero di passi computazionali (limitato ad esempio dal numero di hidden layers). Si possono elaborare sequenze sia in input che in output (o entrambi), in fig. 7 si distinguono cinque casi.

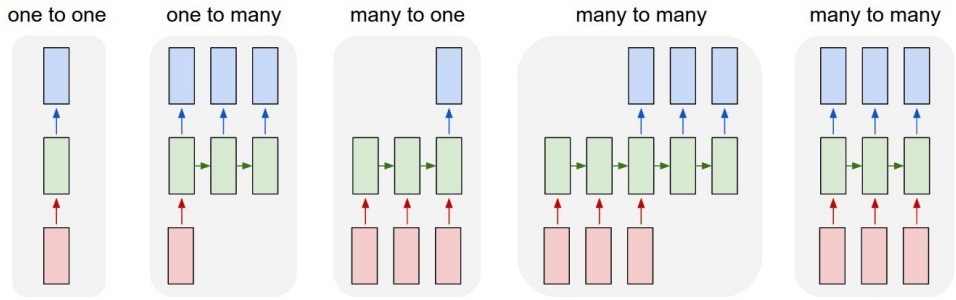


Figura 7: Combinazioni di sequenze vettoriali. [12]

Uno a uno: il caso più semplice, in cui non vi è ricorrenza, ad esempio nella classificazione di immagini.

Uno a molti: il caso in cui è presente una sequenza in output. Tipicamente usato nella creazione di sottotitoli, dove ad una sola immagine corrisponde una frase.

Molti ad uno: in questo caso la sequenza è presente solo in input, è la struttura della *sentiment analysis*, dove da una frase viene estratta la sensazione positiva/negativa contenuta in essa.

Molti a molti: qui abbiamo una sequenza sia in input che in output, si potrebbe trattare di un modello di traduzione che assegna ad una frase in una lingua la frase corrispondente in un'altra.

Molti a molti (sinc.): come nell'esempio precedente ma l'output è in sincronia con l'input. Utilizzato ad esempio nella classificazione video, in cui un'etichetta va assegnata ad ogni frame in tempo reale.

1.5.1 *Dipendenze a lungo termine*

Nel progettare una RNN va presa in considerazione la possibilità di incontrare la necessità di fornire un vasto numero di informazioni dal contesto per risolvere un problema. Tornando al compito della previsione di parole in un testo, si supponga di dover completare la frase "sto per andare a nuotare in", le informazioni a breve termine suggeriscono che la parola successiva sia un luogo dove sia possibile nuotare e sapendo che la frase precedente è "la spiaggia è molto assolata" si potrebbe dedurre che la parola da predire sia "mare". Il problema in questione è che si incontrano spesso difficoltà nell'identificare informazioni rilevanti. La formulazione di RNN data finora, in teoria, è perfettamente in grado di

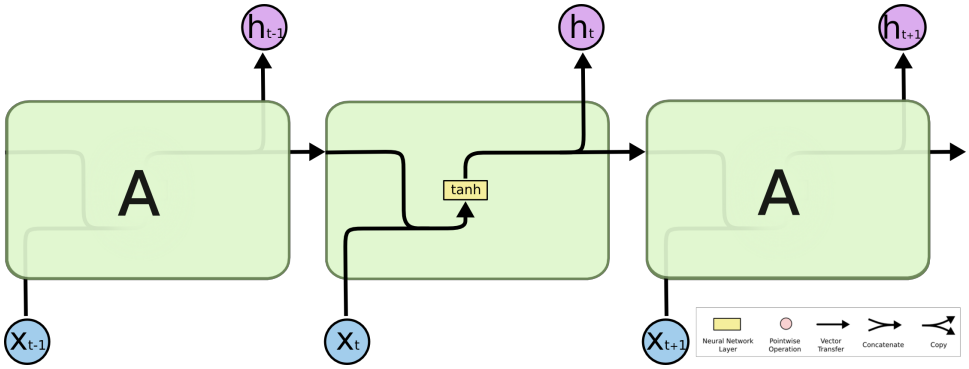


Figura 8: Struttura interna di una RNN standard con un singolo hidden layer.

gestire dipendenze a lungo termine. In pratica, come spesso accade, si tratta di una prova tipicamente banale per una mente umana ma che una semplice implementazione (come in fig. 8) non sarebbe in grado di risolvere efficacemente. Uno degli ostacoli più frequenti da risolvere è il cosiddetto problema del gradiente evanescente (*vanishing gradient* [13]), per superarlo si rendono necessarie varianti più elaborate della RNN semplice, come le LSTM.

1.5.2 *Long Short Term Memory*

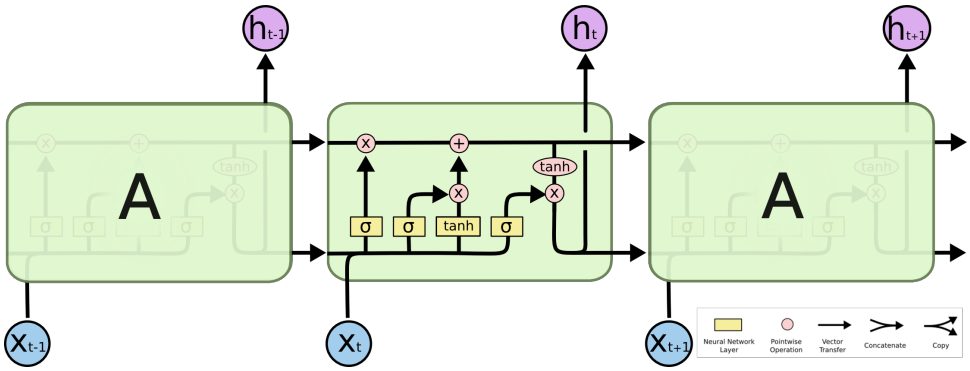


Figura 9: Struttura interna di una lstm che evidenzia i quattro strati interni di un layer

Per ottenere una RNN in grado di apprendere dipendenze a lungo termine, Hochreiter and Schmidhuber [13] introdussero, nel 1997, una versione detta *Long Short Term Memory (LSTM)*.

A differenza di una RNN standard, che si limita ad una semplice struttura ripetuta, le LSTM operano una computazione complessa per

ottenere lo stato interno. Invece della sola funzione di attivazione, una LSTM possiede strumenti per rimuovere, aggiungere o lasciar passare intatta l'informazione attraverso lo stato corrente del layer, detti *gates* (cancelli). Questi gates sono composti da layer con attivazione sigmoide e operazioni di somma o moltiplicazione punto a punto, che definiscono quanto lasciar passare di ogni componente in ingresso.

Seguendo il diagramma in fig. 9, vediamo in basso a sinistra il primo layer sigmoide, detto *forget gate layer*, che si occupa di decidere quale informazione scartare dallo strato precedente o dall'input. Successivamente abbiamo il secondo layer sigmoide, detto *input gate layer*, che decide quali valori saranno aggiornati, tramite l'output di un layer a tangente iperbolica. Questa combinazione va a modificare lo stato interno della LSTM, che attraversa la linea orizzontale superiore, che verrà poi trasferito allo stato successivo. Infine, dopo aver attraversato un'altra tangente iperbolica (per standardizzare i valori fra -1 e 1), un'altra sigmoide opera da gate per regolare l'output. [1]

1.5.3 *L'informazione futura*

Per come sono state illustrate finora, le RNN si possono ritenere in grado di prendere in considerazione tutta l'informazione ricevuta fino al time frame corrente, dove la struttura specifica della rete e l'algoritmo utilizzato per l'apprendimento definiscono quanta di questa informazione verrà effettivamente sfruttata.

Talvolta accade che, allo scopo di migliorare la previsione corrente, si renda utile conoscere anche una parte dell'informazione successiva a quella dell'attuale time frame (ad esempio il genere del soggetto di una frase, nel determinare la traduzione di un articolo da una lingua con articoli neutri ad un'altra). La generica RNN potrebbe ottenere questo risultato aggiungendo un ritardo sull'output di un certo numero M di time frames, per includere l'input fino a $\mathbf{x}_{t_c+M}$ allo scopo di predire $\mathbf{y}_{t_c}$. In teoria M potrebbe essere scelto largo abbastanza da includere tutto l'input restante, in pratica empiricamente è noto che la bontà della previsione diminuisce drasticamente per un M troppo grande. Una possibile spiegazione a questo potrebbe essere che al crescere di M le capacità predittive della rete si vadano sempre più concentrando nel ricordare

1 Esiste un'ampia varietà di modelli basati sulle LSTM, si suggeriscono: LSTM con "peephole connections" [14] (connessioni a spioncino), Gated Recurrent Units (GRU) [15] e Depth Gated RNNs [16]

l'input fino a x_{t_c+M}, lasciando sempre meno potenza di elaborazione per combinare le conoscenze da diversi vettori.

Nonostante quest'operazione di ritardo di alcuni frames sia stata concretamente sfruttata con successo per migliorare i risultati in un sistema di riconoscimento vocale [20], successo confermato anche da esperimenti indipendenti [17], il ritardo ottimale dipende dal compito specifico ed è stato ottenuto con prove ed errori sul validation set. Sarebbe naturalmente preferibile un approccio più elegante.

Per sfruttare tutta l'informazione disponibile, è possibile usare due diverse reti (una per ogni direzione) e in qualche modo combinare i loro risultati. Entrambe le reti possono essere considerate esperte del problema specifico su cui sono state addestrate. Un modo per combinare le *"opinioni di esperti"* è di assumerne l'indipendenza, che comporta l'uso della media aritmetica per la regressione e della media geometrica (o, alternativamente, la media aritmetica nel dominio logaritmico) per la classificazione. Queste procedure sono dette *linear opinion pooling* e *logarithmic opinion pooling* rispettivamente [18], [19]. Nonostante la semplice combinazione degli output sia stata applicata con buoni risultati [21], non è chiaro, in generale, come effettuarla efficacemente, dal momento che reti diverse addestrate sugli stessi dati non possono essere considerate effettivamente indipendenti.

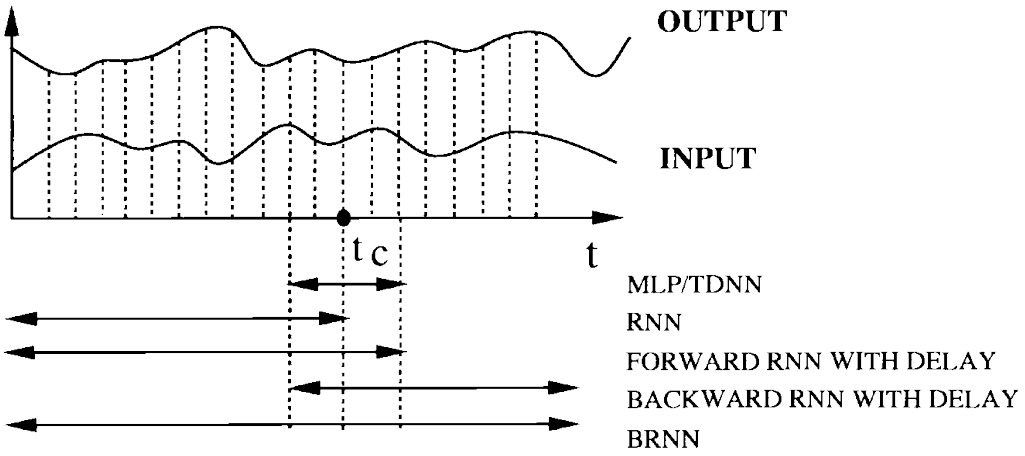


Figura 10: Confronto sull'utilizzo dell'input in diverse reti neurali.

1.5.4 *Reti bidirezionali*

Allo scopo di superare le limitazioni di una generica RNN, esposte nella sezione precedente, è stata ideata una struttura detta *Bidirectional*

Recurrent Neural Network (BRNN, rete neurale ricorrente bidirezionale) che può essere addestrata con tutte le informazioni in input, passate e future, di ogni time frame.

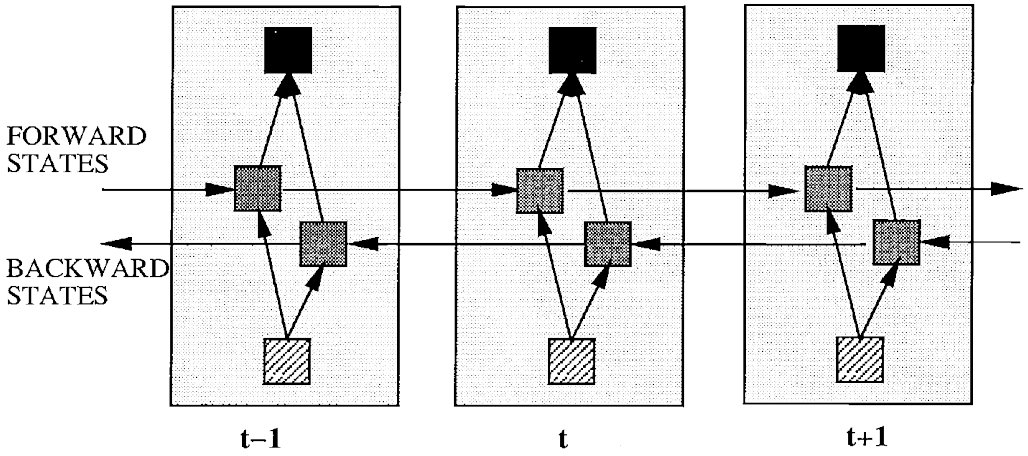


Figura 11: Struttura generica di una BRNN, svolta lungo tre time steps.

L'idea alla base della BRNN è quella di suddividere lo stato interno di un elemento di una RNN regolare in due parti: una responsabile per la dimensione temporale positiva (forward), una per quella negativa (backward). Gli output dagli stati forward non sono connessi agli input di quelli backward e viceversa. Ciò conduce alla struttura che si può vedere in fig. 11. Si può notare come eliminando gli stati backward, questa struttura diventa analoga a quella di una generica RNN come in fig. 5. Rimuovendo gli stati forward, invece, otteniamo una RNN con l'asse temporale invertito. Prendere in considerazione entrambe le direzioni temporali, rende possibile utilizzare direttamente tutta l'informazione passata e futura rispetto al time step corrente, senza alcun bisogno di ricorrere a ritardi.

Una BRNN può essere addestrata con gli stessi algoritmi con cui si addestrerebbe una RNN semplice, dal momento che non vi sono interazioni fra le due tipologie di stati e, di conseguenza, può essere svolta in una generica rete ad avanzamento semplice. Tuttavia se, ad esempio, viene utilizzata una forma qualunque di *back propagation through time* (BPTT), la procedura del passo forward e backward diventa leggermente più complessa, dal momento che l'aggiornamento dello stato e dell'output non può più essere effettuato uno per volta. In questo caso, i passi forward e backward lungo la BRNN svolta lungo il tempo vengono effettuati all'incirca allo stesso modo che per un MLP regolare. Alcuni

accorgimenti particolari sono richiesti solo all'inizio e alla fine dei dati di addestramento[2].

1.6 AUTOENCODER

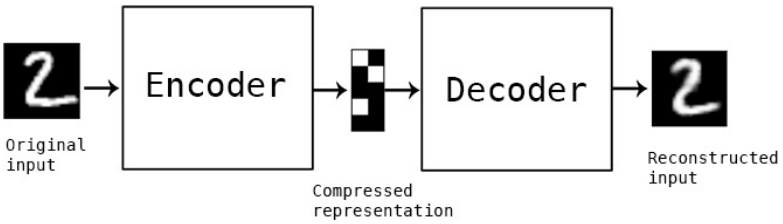


Figura 12: Struttura di un generico AutoEncoder, da [22]

Il processo di *autoencoding* indica il risultato di un algoritmo di compressione, dove le funzioni di compressione e decompressione presentano le seguenti caratteristiche: sono specifiche dei dati, presentano perdite e sono apprese automaticamente attraverso gli esempi, piuttosto che codificate a mano da un programmatore. L'ultima di queste caratteristiche rimanda immediatamente all'uso di una rete neurale, che infatti è l'implementazione tipica di un autoencoder.

A differenza degli algoritmi di compressione genericamente utilizzati, ad esempio MPEG-2 Audio Layer III (comunemente detto MP3) per l'audio, che si prestano ad un utilizzo ampio, un autoencoder è limitato dal dataset su cui viene addestrato. Le prestazioni su un autoencoder addestrato su di un certo suono o su fotografie di volti crollerebbero drasticamente, su suoni diversi o su fotografie di alberi. Allo stesso modo di alcuni degli algoritmi più comuni, l'output dopo la decompressione presenta una diminuzione di qualità rispetto all'input originale. In compenso è facile addestrare istanze specifiche dell'algoritmo, che presentino buoni risultati su un particolare input, in quanto non sono generalmente necessari nuovi interventi di ingegneria del software ma solo dati appropriati.

Per assemblare un autoencoder sono necessarie tre componenti fondamentali: una funzione che operi la codifica, una che operi la decodifica e una che misuri la distanza che occorre fra la rappresentazione compressa e quella ottenuta dalla ricostruzione, in termini di perdita di dati (ovvero una loss function). Decoder ed encoder (fig 12) sono tipicamente

2 Si rimanda a [17] per approfondimenti e varianti.

funzioni parametriche (reti neurali), differenziabili rispetto alla funzione di distanza in modo da essere ottimizzabili per minimizzare la perdita in ricostruzione, usando lo *Stochastic Gradient Descent* (SGD).

1.7 MODELLI GENERATIVI

All'inizio di questo elaborato è stato citato brevemente il concetto di *modello generativo*, poi ripreso in alcuni esempi nella sezione riguardante le reti ricorrenti.

Gli algoritmi finora elencati, nella loro implementazione più semplice, formano modelli che tipicamente stabiliscono i confini che intercorrono fra le varie classi a cui possono appartenere i dati in input. Per questa ragione sono detti *modelli discriminativi*, in quanto non si preoccupano di fare assunzioni sull'origine dei dati forniti ma si limitano a categorizzarli nel modo più efficace. Lo scopo di un modello generativo, invece, è quello di apprendere come sono stati generati i dati ottenuti, producendo una rappresentazione delle classi a cui appartengono attraverso l'analisi delle caratteristiche rilevate nell'input.

Da un punto di vista matematico, un modello discriminativo cerca di apprendere la distribuzione di probabilità condizionata rispetto ai dati, in formule:

$$f(\mathbf{x}) = \arg\max_{y} p(\mathbf{y}|\mathbf{x}) \tag{1.1}$$

Per contro, un modello generativo cerca di apprendere la distribuzione di probabilità congiunta, ovvero, usando la regola di Bayes (e liberandoci di $p(x)$, in quanto si massimizza secondo y):

$$f(\mathbf{x}) = \arg\max_{y} p(\mathbf{y}|\mathbf{x})p(\mathbf{y}) \tag{1.2}$$

Che corrisponde a $p(x, y)$. Come si può notare intuitivamente, ad un modello generativo è richiesta una complessità superiore che si può tradurre in un calo di prestazioni su compiti meno elaborati, rispetto ai modelli discriminativi, ad esempio nella classificazione. D'altra parte un modello discriminativo non è in grado di cogliere caratteristiche e relazioni complesse fra i dati in input e le variabili target, né è in grado di produrre dati originali analoghi a quelli appresi, proprietà che rendono preferibile un modello generativo per compiti di apprendimento non supervisionato e che sono fondamentali per generare astrazioni come quelle ricercate nella riproduzione di disegni a mano.

1.8 VARIABILI LATENTI

Una variabile latente è una variabile casuale che condiziona la generazione dell'output di un modello. Se, ad esempio, si volessero generare immagini di cifre scritte a mano (database MNIST), se nella metà sinistra della cifra è presente metà del carattere *5*, non si può accettare che nella metà destra compaia metà del carattere *0*. In questo caso, la variabile latente permette di stabilire in precedenza quale carattere generare, prima di assegnare valori ai pixel. La variabile è detta *latente* poiché, dato il carattere prodotto dal modello, non si possiede necessariamente l'informazione su quale assetto della variabile l'abbia generato.

Prima di poter dire che un modello è in grado di rappresentare il dataset considerato per ogni punto x, ci si deve assicurare che esista un assetto della variabile latente che permette al modello di generare un punto molto simile ad esso. Formalmente si supponga di avere un vettore di variabili latenti z, in uno spazio multidimensionale Z che si può facilmente campionare in accordo ad una qualche funzione di densità di probabilità (PDF - Probability Density Function) $P(z)$ definita su Z. Successivamente, si supponga di avere una famiglia di funzioni deterministiche $f(z, \theta)$, parametrizzate rispetto ad un vettore θ su di un qualche spazio Θ, dove $f : Z \times \Theta \to \chi$. f è deterministica ma, se z è casuale e θ è fissato, allora $f(z; \theta)$ è una variabile casuale nello spazio χ. Si desidera ottimizzare θ in modo da poter campionare z da $P(z)$ e, con alta probabilità, $f(z; \theta)$ sarà analoga alle x del dataset considerato.

In modo matematicamente rigoroso, si può affermare che l'obbiettivo è massimizzare la probabilità di ogni x nel training set lungo tutto il procedimento generativo, in accordo a:

$$P(x) = \int P(x|z; \theta) P(z) dz \tag{1.3}$$

Qui $f(z, \theta)$ è stato rimpiazzato da $P(x|z; \theta)$, che permette di rendere esplicita la dipendenza di x da z attraverso la regola delle probabilità totali. L'intuizione dietro a questo framework, detto *"maximum likelihood"*, è che se il modello è in grado di riprodurre esempi del training set, sarà probabilmente in grado di generare esempi analoghi e con poca probabilità produrrà risultati molto diversi.

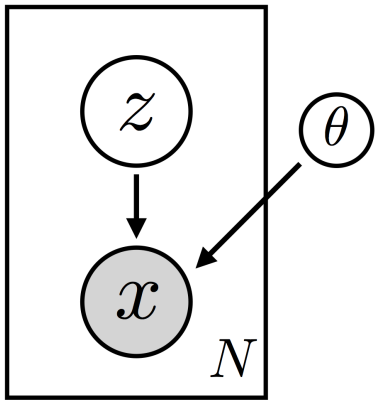


Figura 13: Modello di un VAE, da [24]

1.9 VARIATIONAL AUTOENCODER

I variational autoencoder [23] realizzano l'implementazione di un modello generativo attraverso una struttura analoga a quella degli autoencoder tradizionali, seppur fondandosi su principi matematici diversi, per la presenza di un encoder ed un decoder nella propria struttura. Si tratta di una classe di modelli che si basano sulla generazione condizionata da una variabile latente.

Nei VAE, la scelta della distribuzione di output è spesso gaussiana, ovvero della forma:

$$P(\mathbf{x}|\mathbf{z};\boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}|f(\mathbf{z};\boldsymbol{\theta}), \sigma^2 * \mathbf{I}) \tag{1.4}$$

Il che significa che ha media $f(\mathbf{z};\boldsymbol{\theta})$ e covarianza pari alla matrice identità moltiplicata per un qualche scalare σ (che è un iperparametro). In ogni caso ciò non è obbligatorio, ad esempio, se $\mathbf{x}$ fosse binario, allora $P(\mathbf{x}|\mathbf{z})$ potrebbe essere una distribuzione di Bernoulli parametrizzata da $f(\mathbf{z};\boldsymbol{\theta})$. La proprietà fondamentale è che $P(\mathbf{x}|\mathbf{z})$ sia computabile e continua in $\boldsymbol{\theta}$.

La necessità della variabile latente è quella di rappresentare il più accuratamente possibile l'informazione latente. Dovendo ad esempio rappresentare delle cifre scritte a mano (MNIST) dovrebbe non solo determinare quale cifra rappresentare ma anche l'angolo con cui disegnarla, lo spessore del tratto ed altre proprietà prettamente stilistiche. A complicare le cose c'è il fatto che queste proprietà potrebbero presentare dipendenze: una cifra maggiormente inclinata potrebbe essere correlata ad una scrittura più frettolosa e, di conseguenza, ad un tratto più sottile. Idealmente si vorrebbe evitare di stabilire manualmente quali informazioni verranno

codificate in ogni dimensione di z, inoltre sarebbe da evitare anche la descrizione specifica delle dipendenze fra le dimensioni. L'approccio sfruttato dai VAE è inusuale: partono dall'assunto che non ci sia un'interpretazione semplice delle dimensioni, affermando che campioni di z possano essere estratti piuttosto da una distribuzione semplice, ovvero $\mathcal{N}(0, \mathbf{I})$. La chiave di questa logica sta nell'osservare che una distribuzione qualunque in d dimensioni, può essere generata attraverso un set di d variabili casuali con distribuzione normale, mappante tramite una funzione sufficientemente complicata[3]. In generale non c'è da porsi il problema di assicurarsi che esista la struttura latente, se questa struttura aiuta il modello a riprodurre accuratamente (ovvero, massimizzare la verosimiglianza) i dati del training set allora il modello la imparerà in qualche layer.

Per massimizzare l'equazione 1.3, come genericamente si fa nel machine learning, si cerca una formula computabile per $P(\mathbf{x})$, se ne prende il gradiente e con esso si ottimizza il modello tramide SGA (stochastic gradient ascent). Una computazione approssimata di $P(\mathbf{x})$ risulta abbastanza semplice: è sufficiente campionare una vasta quantità di valori di z, $\{z_1, ..., z_n\}$ e calcolare $P(\mathbf{x}) \approx \frac{1}{n} \sum_i P(\mathbf{x}|z_i)$. Il problema sorge per spazi a molte dimensioni, n potrebbe dover essere estremamente ampio prima di ottenere una stima accurata di $P(\mathbf{x})$. La soluzione dei VAE è di evitare la definizione di una misura di somiglianza più accurata, che potrebbe essere di difficile implementazione in domini complessi quali le immagini. La scelta ricade piuttosto sull'alterare la procedura di campionamento, per renderla più rapida senza dover modificare la metrica. In sostanza l'idea è di definire una nuova funzione $Q(z|\mathbf{x})$ che cerca di campionare valori di z che potrebbero aver prodotto $\mathbf{x}$. La speranza è che lo spazio generato da Q sia minore dello spazio di ogni z che sottostà a $P(z)$. Ciò permette di calcolare facilmente $E_{z\sim Q} P(\mathbf{x}|z)$ a patto che questa sia legata a $P(\mathbf{x})$.

Per integrare questo concetto all'interno della computazione della rete neurale, potendo quindi effettuare SGD su tutti i passaggi, altrimenti non differenziabili, si compie un'operazione detta *reparametrization trick*. Si sceglie la funzione di codifica come $Q(z|\mathbf{x}) = \mathcal{N}(z|\mu(\mathbf{x}; \Theta), \sigma(\mathbf{x}; \Theta))$, dove

3 In una dimensione si può utilizzare l'inversa della funzione a distribuzione cumulativa (CDF - cumulative distribution function) della distribuzione desiderata, composta con la CDF di una Gaussiana, in estensione al principio di *"inverse transform sampling"*. Per dimensioni multiple si può applicare il medesimo processo cominciando con la distribuzione marginale di una dimensione e ripetendo con quella condizionale di ogni dimensione aggiuntiva. Si veda "inversion method" e "conditional distribution method" in Devroye et al. [26]

μ e $\hat{\sigma}$ sono funzioni deterministiche arbitrarie (ovvero implementabili con una rete neurale) con parametro Θ che può essere appreso dai dati. Ciò permette di definire una loss function per la rete neurale che non è altro che la somma fra l'errore di ricostruzione nel riprodurre i dati a partire dalla variabile latente z e la *KL divergence* (divergenza di Kullback - Leibler) fra $Q(z|x)$ e $P(z)$:

$$l = -E_{z\sim Q}[\log(P(x|z))] + KL(Q(z|x)||P(z)) \tag{1.5}$$

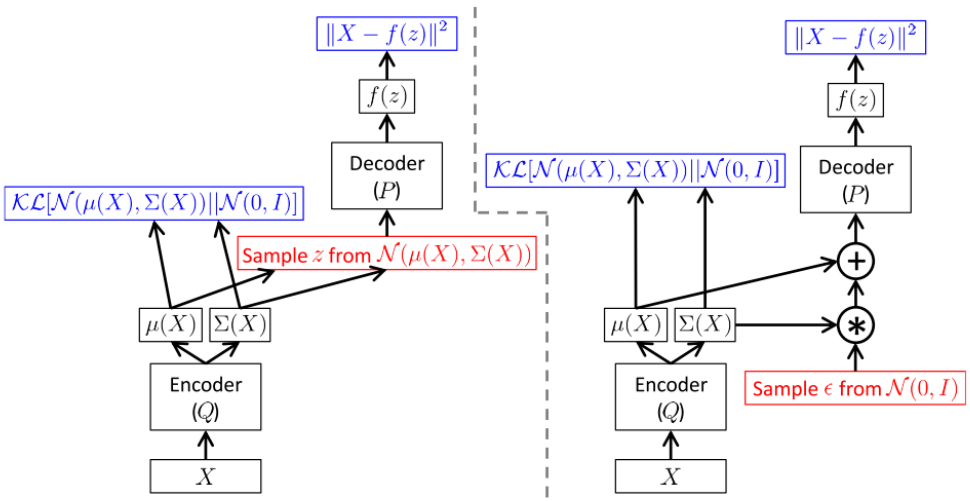


Figura 14: Un VAE per intero, a sinistra senza il "reparametrization trick", a destra con. In rosso le operazioni non differenziabili, in blu le loss. Queste due varianti differiscono per il fatto che la retropropagazione si può applicare solo sulla rete a destra.

1.10 PROBLEMI INVERSI

Molte applicazioni potenziali delle reti neurali ricadono nella categoria dei problemi inversi. Alcuni esempi includono il controllo di impianti industriali, l'analisi di dati spettrali, le ricostruzioni tomografiche e la cinematica dei robot. Per questi problemi esiste un ben definito problema *diretto* caratterizzato da una mappatura funzionale (ovvero a valore singolo). Di solito ciò corrisponde alla causalità nei sistemi fisici.

Si consideri un semplice esempio di un problema inverso che riguarda la mappatura fra una variabile unitaria in input e una in output, rispettivamente t e x, definito da:

$$x = t + 0.3\sin(2\pi t) + \epsilon \tag{1.6}$$

Dove ϵ è una variabile casuale con distribuzione uniforme nell'intervallo (-0.1, 0.1). La mappatura da t a x costituisce un esempio di problema diretto. In assenza del termine di disturbo ϵ, questa mappatura è a valore singolo, ovvero ogni valore di t dà origine ad un singolo valore di x. In fig. 15 si può notare un dataset di mille punti generati dal campionamento1.6 ad intervalli uguali di t nel raggio (0.0, 1.0). Allo stesso modo è mostrata la mappatura rappresentata da un MLP standard addestrato su questo dataset. La rete prende un valore in input, possiede cinque unità con funzione d'attivazione *tanh* (tangente iperbolica) e restituisce un output lineare. L'addestramento è stato eseguito con mille cicli completi dell'algoritmo quasi-Newtoniano BFGS [25]. Si può notare che la rete, che approssima la media condizionale dei dati target, dà un'ottima rappresentazione della distribuzione alla base dei dati. Questo risultato è indipendente alla scelta della struttura della rete, al valore iniziale dei pesi e ai dettagli della procedura di training.

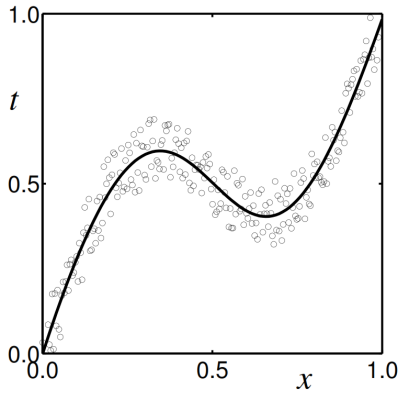


Figura 15: Un semplice esempio di problema diretto: sono evidenziati 1000 punti corrispondenti ai dati (i cerchi) generati dalla mappatura $x = t + 0.3\sin(2\pi t) + \epsilon$ dove ϵ è una variabile casuale. La curva rappresenta il risultato dell'addestramento di un MLP con cinque unità e somma di quadrati come funzione d'errore. La rete approssima la media condizionale del target, che dà una buona rappresentazione del generatore dei dati.

Si consideri adesso il corrispondente problema inverso, dove verrà utilizzato lo stesso dataset ma sarà tentata la mappatura dalla variabile x

alla variabile t. Il risultato dell'addestramento di un'analoga rete neurale è mostrato in fig. 16.

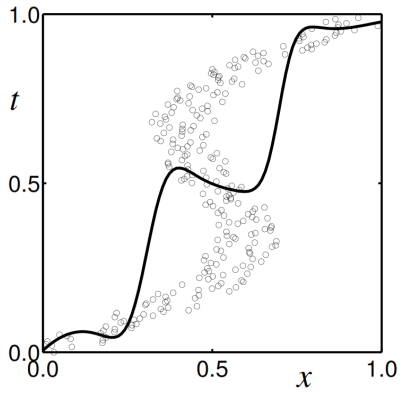


Figura 16: Quest'immagine mostra esattamente lo stesso dataset della figura 15, invertendo input e variabili target. La curva mostra nuovamente il risultato dell'addestramento di una MLP con funzione *sum-of-squares*. Stavolta la rete ottiene un pessimo risultato, continuando a tentare di rappresentare la media condizionale.

Di nuovo, la rete tenta di approssimare la media condizionale dei dati di target ma stavolta ottiene una rappresentazione inadeguata del procedimento che ha generato i dati. La modalità di mappatura della rete adesso è molto più sensibile alla struttura, all'inizializzazione dei pesi, ecc... Di quanto non fosse nel caso del problema diretto. Il risultato ottenuto in figura è il migliore che sia stato possibile ricavare dopo attenta ottimizzazione (ovvero la generica soluzione trovata è stata spesso di qualità inferiore a questa). La rete di questo esempio è composta da venti unità ed è stata addestrata per mille cicli col medesimo algoritmo. Evidentemente una MLP convenzionale, addestrata con la somma di quadrati come funzione d'errore, non è in grado di dare una buona rappresentazione della distribuzione di questi dati.

1.11 MIXTURE DENSITY NETWORK

Come spiegato da Bishop [27], se si assume che la distribuzione di probabilità condizionata dei dati di target sia Gaussiana, si può derivare la tecnica dei minimi quadrati utilizzando la massima verosimiglianza. Questo motiva l'idea di rimpiazzare la distribuzione Gaussiana nella densità condizionale del vettore target con un *mixture model* (modello a mistura) [28], che ha la flessibilità per modellare completamente funzioni

di distribuzione generiche. La densità di probabilità dei dati target è quindi rappresentata come combinazione lineare di funzioni kernel della forma:

$$p(\mathbf{x}|\mathbf{y}) = \sum\_i = 1^m \alpha_i(\mathbf{x}\theta_i(\mathbf{y}|\mathbf{X})) \tag{1.7}$$

Dove *m* è il numero di componenti nella mistura e $\alpha_i(\mathbf{x})$ sono detti *mixing coefficients* (coefficienti di mescolamento).

Sempre in accordo a Bishop, le funzioni della mistura saranno Gaussiane della forma:

$$\theta_i(\mathbf{y}|\mathbf{x}) = \frac{1}{(2\pi)^{\frac{c}{2}}\sigma_i(\mathbf{x})^c} \exp\left\{ -\frac{\|\mathbf{y} - \boldsymbol{\mu}_i(\mathbf{x}\|^2}{2\sigma_i(\mathbf{x})^2} \right\} \tag{1.8}$$

Dove $\boldsymbol{\mu}_i$ rappresenta il centro dell'i-esima funzione. L'autore assume che le componenti del vettore di output sono statisticamente indipendenti rispetto ad ogni componente della distribuzione e possono essere descritte dalla varianza $\sigma_i(\mathbf{x}$. Questa assunzione può essere rilassata introducendo matrici complete di covarianza per ogni Gaussiana, al costo di un formalismo più complesso. In linea di principio, tuttavia, questa complicazione non è necessaria poiché un *Gaussian Mixture Model* (modello a mistura gaussiana), con componenti date da 1.8, può approssimare qualunque funzione di densità di probabilità data con precisione arbitraria, data la corretta selezione dei coefficienti di mistura e dei parametri delle Gaussiane (media e varianza)[4]. Di conseguenza la rappresentazione data da 1.7 e 1.8 è completamente generale e, in particolare, non assume necessariamente che le componenti di $\mathbf{t}$ siano statisticamente indipendenti[5].

Come si può vedere in fig. 17, dato un vettore di input $\mathbf{x}$, la MDN fornisce un formalismo generale per modellare una funzione di densità di probabilità condizionata $p(\mathbf{y}|\mathbf{x})$. Questa combinazione fra reti neurali tradizionali e modelli a mistura è resa possibile utilizzando la verosimiglianza logaritmica della combinazione lineare fra le distribuzioni Gaussiane come funzione di loss per la rete.

Assemblare una MDN incrementa il numero di elementi di output della rete neurale di partenza: dati *c* parametri, con la MDN questi salgono a $(c + 2) * m$, dove *m* è il numero di misture del modello.

Bishop suggerisce alcune restrizioni che i parametri della rete devono soddisfare:

4 Ottenuti tramite le variabili addestrabili della rete neurale sottostante.

5 Al contrario alla rappresentazione con singola Gaussiana.

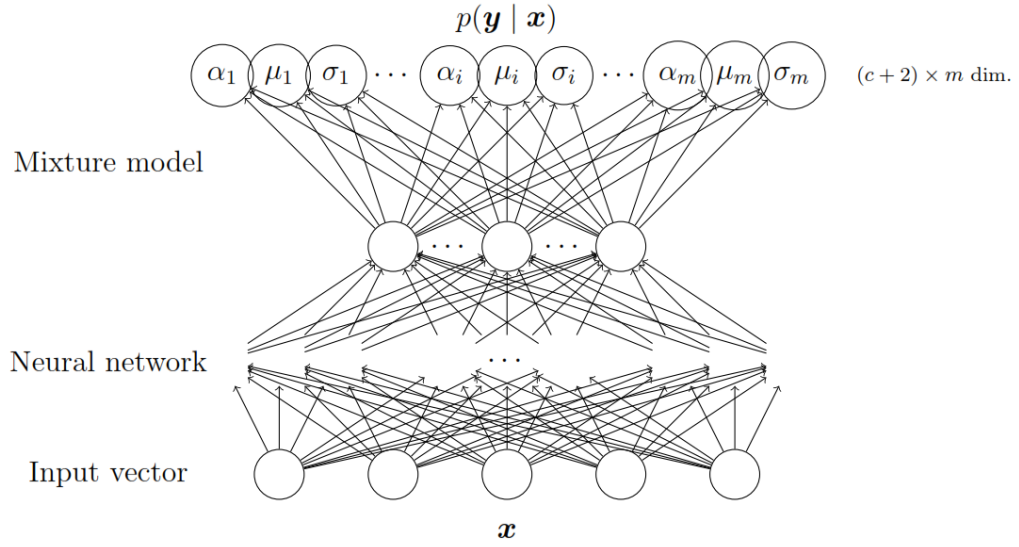


Figura 17: Rappresentazione di una MDN. L'output della rete neurale determina i parametri di una mistura. Di conseguenza, il modello rappresenta la funzione di densità di probabilità condizionale delle variabili target condizionate all'input $\mathbf{x}$ della rete.

- Come richiesto per le probabilità, è necessario che i coefficienti di mistura α_i soddisfino il vincolo $\sum_{i=1}^{m} = 1$. Per ottenere questa restrizione, in linea di principio, è sufficiente una funzione di attivazione *softmax* per il nodo corrispondente ad α_i.

$$\alpha_i = \frac{\exp(z_i^{\alpha})}{\sum_{j=1}^{m} \exp(z_j^{\alpha})} \tag{1.9}$$

  dove z_i^{α} rappresenta l'output corrispondente. Questa restrizione assicura che le quantità α_i siano comprese in $(0, 1)$ e sommino ad uno.

- Le varianze σ_i rappresentano parametri di scala, di conseguenza è conveniente rappresentarle in termini dell'esponenziale dell'output corrispondente

$$\sigma_i = \exp(z_i^{\sigma}) \tag{1.10}$$

  che, in un framework Bayesiano, corrisponderebbe alla scelta di una distribuzione a priori non informativa, assumendo che l'output corrispondente z_i^{σ} abbia una distribuzione di probabilità uniforme. Questa rappresentazione ha il beneficio addizionale di evitare configurazioni "patologiche" in cui una o più varianze vanno a zero.

- I parametri centrali μ_i rappresentano parametri di posizione, prendendo in considerazione l'idea che la distribuzione a priori sia non informativa, è conveniente rappresentare direttamente questi parametri dagli output della rete, ovvero:

$$\mu\_i, k = z^{\mu}_{i,k} \tag{1.11}$$

A questo punto è possibile costruire una funzione di verosimiglianza usando la densità condizionale del vettore di target completo. Dopodiché, per definire una funzione di errore (da usare come loss per la rete), si può utilizzare l'approccio standard del metodo di massima verosimiglianza, che richiede la massimizzazione della funzione di verosimiglianza logaritmica o, equivalentemente, la minimizzazione del logaritmo negativo della verosimiglianza. Di conseguenza, la funzione di errore per una MDN è:

$$\log \mathbb{L}(\mathbf{y}|\mathbf{x}) = -\log(p(\mathbf{y}|\mathbf{x})) = -\log\left(\sum_{i=0}^{m} \alpha_i(\mathbf{x})\theta_i(\mathbf{y}|\mathbf{x})\right) \tag{1.12}$$

2

STRUMENTI

2.1 LIBRERIE SOFTWARE

Il campo del Deep Learning è in costante e vertiginosa evoluzione. Ciò rende difficile, se non impossibile, essere costantemente aggiornati su ogni nuovo sviluppo del settore.

Al momento attuale le librerie software maggiormente diffuse sono *TensorFlow* sviluppata da google, *Torch* sviluppata da facebook, *Theano* sviluppata dall'università di Montreal e *Caffe* Sviluppata dall'università di Berkeley. Senza scendere in ulteriori dettagli per le altre si prende in considerazione la più apprezzata (stando a GitHub), ovvero TensorFlow.

2.2 TENSORFLOW

Secondo la pagina ufficiale[4], TensorFlow è una libreria software open-source per il calcolo numerico tramite grafi di flussi di dati. I nodi di un grafo rappresentano operazioni matematiche, mentre gli archi rappresentano i tensori comunicanti fra loro. Può essere sfruttato in molti modelli di Machine Learning, al di là del Deep Learning. Uno dei vantaggi più importanti di TensorFlow, rispetto alle altre librerie, è che possiede un'architettura flessibile che permette agli sviluppatori di svolgere i calcoli su una o più CPU o GPU con una singola API.

L'estrema versatilità delle API, inoltre, facilita la possibilità di interfacciarsi a TensorFlow anche da parte di altre librerie, quali Keras.

2.2.1 *Keras*

Secondo la pagina ufficiale[3], Keras è una libreria di alto livello per sviluppare reti neurali, sviluppata da François Chollet, scritta in Python ed in grado di interfacciarsi sia a TensorFlow che a Theano.

Questa libreria è stata sviluppata a seguito del progetto di ricerca noto come ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System), ed è da considerarsi indipendente da istituzioni, organizzazioni o compagnie. Si tratta probabilmente della più supportata ed utilizzata dopo TensorFlow. Le ragioni del successo di Keras sono molteplici:

- La modularità, la semplicità e l'estendibilità garantiscono la facilità d'uso e la rapidità della prototipazione.
- Supporta l'implementazione di reti ricorrenti e convoluzionali, permettendo la combinazione fra di esse, inoltre è possibile definire layer personalizzati per integrare strutture complesse in una rete (ad esempio MDN).
- Come TensorFlow (appoggiandosi ad esso), può eseguire la computazione su CPU e/o GPU senza accorgimenti particolari.

2.2.2 *Recurrent Shop*

Secondo la documentazione[34], Recurrent Shop è un framework adatto alla costruzione di reti ricorrenti complesse su Keras. Librerie come Keras permettono una facile prototipazione di layer e modelli, tramite l'iterazione immediata fra implementazioni di architetture diverse, presentano però alcune carenze nella definizione di reti ricorrenti personalizzate. Una delle più rilevanti è l'implementazione di celle ricorrenti riutilizzabili. In Keras, come in molte altre librerie, sono presenti layer (come LSTM, GRU ecc...) che possono essere utilizzati solo così come vengono implementati, senza poterli sfruttare per includerli all'interno di una RNN complessa. A sua volta implementare la logica di una RNN in un layer personalizzato può risultare un'operazione onerosa, ad esempio informazioni riguardo gli stati (forma e inizializzazione) necessitano di essere aggiornate tramite metodi diversi `get_initial_states` e `reset_states`. Molte architetture presentano quindi implementazioni non banali, quali:

- Sincronizzazione degli stati in tutti i layers di una pila di RNN.
- Readout, ovvero il passaggio dell'output di un layer di una pila di RNN come input del time step successivo.
- Decoder: RNN che vedono tutta la sequenza (o il vettore) di input ad ogni time step.

- *Teacher forcing*: l'utilizzo del dato reale al tempo t-1 per la previsione al tempo t durante il training.
- RNN nidificate.
- Inizializzazione degli stati tramite distribuzioni diverse.

Recurrent Shop facilita l'implementazione di queste strutture, permettendo la scrittura di RNN di complessità arbitraria, attraverso le API di Keras. In sostanza è possibile costruire modelli standard di Keras in cui vengano definite le logiche dei singoli time step e convertirli in layer ricorrenti attraverso Recurrent Shop[1].

2.3 DATASET

alarm clock	ambulance	angel	ant	barn	basket	bee
bicycle	book	bridge	bulldozer	bus	butterfly	cactus
castle	cat	chair	couch	crab	cruise ship	dolphin
duck	elephant	eye	face	fan	fire hydrant	firetruck
flamingo	flower	garden	hand	hedgehog	helicopter	kangaroo
key	lighthouse	lion	map	mermaid	octopus	owl
paintbrush	palm tree	parrot	passport	peas	penguin	pig
pineapple	postcard	power outlet	rabbit	radio	rain	rhinoceros
roller coaster	sandwich	scorpion	sea turtle	sheep	skull	snail
snowflake	speedboat	spider	strawberry	swan	swing set	tennis racquet
	Monna Lisa	toothbrush	truck	whale	windmill	

Tabella 1: Le 75 classi inserite inizialmente in `QuickDraw`

Come già citato nell'introduzione 1.1, `QuickDraw` è un dataset di immagini vettoriali ottenuti attraverso *Quick, Draw!*[2], un gioco online dove agli utenti è chiesto di disegnare oggetti appartenenti ad alcune classi entro 20 secondi. `QuickDraw` attualmente consiste di centinaia di categorie, ognuna delle quali è un dataset di almeno settantamila immagini di training e duemilacinquecento immagini di validazione e test, in tabella 1 sono elencate le 75 classi che hanno costituito il nucleo iniziale del dataset.*Quick, Draw!* continua a fornire dati quotidianamente e, di tanto in tanto, nuove tipologie di oggetti vengono aggiunte. Questo ha portato ad ampliare considerevolmente il volume di partenza, arrivando in alcune classi a più che raddoppiare il training set.

Le immagini nel dataset sono rappresentate in un formato che le codifica come sequenze di tratti di penna, detto *stroke-3* [2]. L'evento binario

1 Questa libreria è stata usata nel prototipo di riproduzione in Keras visibile in appendice.
2 in estensione al formato utilizzato in [29].

che rappresenta lo stato della penna è poi codificato come evento multi-stato, in un formato detto *stroke-5*, attraverso lo *one hot encoding* delle possibili configurazioni, prima di essere passato alla rete. Le coordinate assolute iniziali del disegno sono poste all'origine degli assi, lo schizzo è una lista di punti, dove ogni punto è un vettore che consiste di 5 elementi: $(\Delta x, \Delta y, p_1, p_2, p_3)$. I primi due elementi sono le variazioni posizionali della penna, rispetto alla posizione precedente, in direzione x e y. I successivi tre elementi rappresentano la codifica dei possibili stati della penna: p_1 indica che nell'arco dello spostamento indicato dai primi due elementi la penna sarà poggiata sul foglio, p_2 indica invece che la penna è sollevata dal foglio nel punto corrente e perciò non traccerà alcuna linea. p_3 infine indica che il disegno è concluso e che eventuali punti seguenti, compreso il corrente, non saranno considerati.

I tratti, dopo la codifica, vengono semplificati attraverso l'algoritmo *Ramer-Douglas-Peucker*[30], che riduce il numero di punti necessari a visualizzare una determinata curva, con un parametro ϵ che è stato stabilito essere pari a 2. I dati originariamente sono salvati come immagini in pixel, la conversione in liste di punti viene poi ottenuta attraverso un fattore di scala, che è calcolato per imporre che la variazione standard delle distanze nel training set sia pari ad 1. Per comodità non è stato scelto di portare gli offset a media zero, dato che tipicamente le medie sono già relativamente piccole.

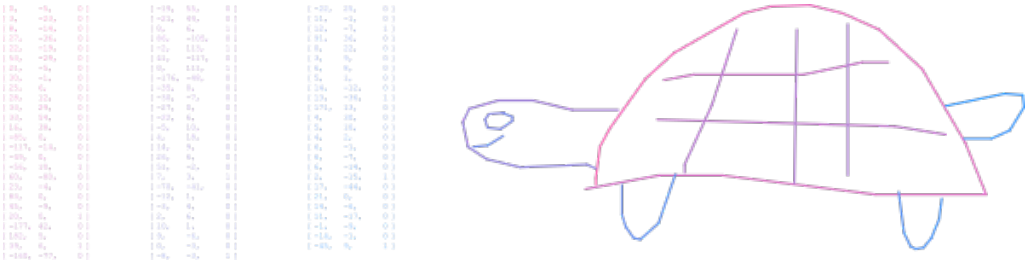


Figura 18: Uno sketch e la lista di tratti corrispondenti, in formato *stroke-5*, i colori corrispondono all'ordine della sequenza.

2.4 MODELLO

In accordo a [1], il modello di sketch-rnn è un VAE 1.9 composto da reti ricorrenti 1.5, che formano uno schema "molti a molti" 1.5. L'encoder è una RNN bidirezionale 1.5.4 che prende in input uno schizzo e come output genera un vettore di latenza di dimensione N_z. Nello specifico,

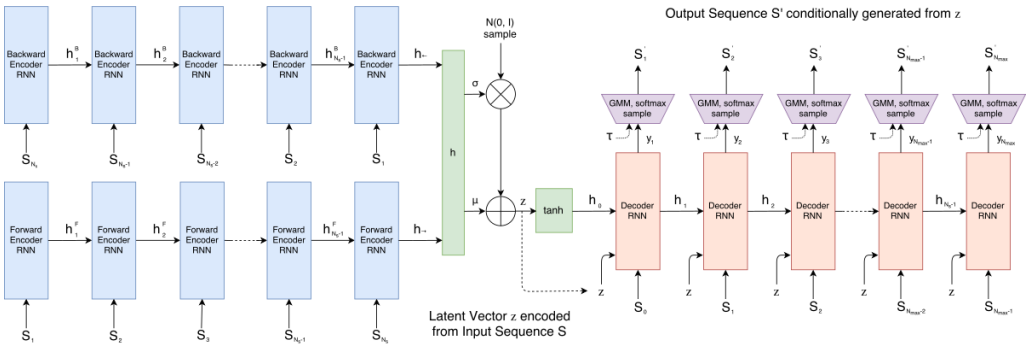


Figura 19: Sketch-rnn

secondo la definizione di rete bidirezionale, l'input[3] viene passato alla rete anche invertito, dopodiché i due stati finali risultanti vengono concatenati in uno stato $\mathbf{h} = [\mathbf{h}_{\rightarrow}; \mathbf{h}_{\leftarrow}]$. L'output $\mathbf{h}$ viene poi proiettato su due vettori, rispettivamente $\boldsymbol{\mu}$ e $\hat{\boldsymbol{\sigma}}$, entrambi di dimensione N_z, attraverso un layer densamente connesso 1.4. $\hat{\boldsymbol{\sigma}}$ viene convertito in un parametro di deviazione standard (non negativo) $\boldsymbol{\sigma}$ attraverso un'operazione esponenziale, $\boldsymbol{\mu}$ e $\boldsymbol{\sigma}$ vengono poi utilizzati, insieme a $\mathcal{N}(0, I)$, che è un vettore di variabili gaussiane identicamente distribuite di dimensione N_z, per costruire un vettore latente $\boldsymbol{z} \in \mathbb{R}^{N_z}$ 1.9:

$$\boldsymbol{\mu} = W_{\mu}\mathbf{h} + b_{\mu}, \hat{\boldsymbol{\sigma}} = W_{\hat{\sigma}}\mathbf{h} + b_{\hat{\sigma}}, \boldsymbol{\sigma} = \exp(\frac{\hat{\boldsymbol{\sigma}}}{2}), \boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \mathcal{N}(0, I) \quad (2.1)$$

Attraverso questo schema di codifica, il vettore di latenza $\boldsymbol{z}$ risulta essere una variabile casuale condizionata rispetto al disegno in input ($Q(\boldsymbol{z}|\mathbf{x})$).

Il decoder è una RNN (potenzialmente autoregressiva[4]) che genera in output degli schizzi condizionati ad un vettore latente $\boldsymbol{z}$ dato. Lo stato iniziale $\mathbf{h}_0$ e, se disponibile 1.5.2, lo stato delle celle $\mathbf{c}_0$ del layer nel decoder viene inizializzato da un layer densamente connesso, con una tangente iperbolica come funzione d'attivazione[5]: $[\mathbf{h}_0; \mathbf{c}_0] = \tanh(W_z\boldsymbol{z} + b_z)$. Ad ogni passo, il decoder prende in input il punto precedente $S_{i\text{-}1}$ e il vettore di latenza $\boldsymbol{z}$, che vengono concatenati come un vettore $\mathbf{x}_i$, dove S_o è definito come $(0, 0, 1, 0, 0)$. L'output di ogni time-step è costituito dai parametri di una distribuzione di probabilità per il prossimo punto nei dati S_i.

$$p(\Delta x, \Delta y) = \sum_{j=1}^{M} \Pi_j \mathcal{N}(\Delta x, \Delta y | \mu_{x,j}, \mu_{y,j}, \sigma_{x,j}, \sigma_{y,j}, \rho_{xy,j}), \text{ dove } \sum_{j=1}^{M} \Pi_j = 1$$

3 Si ricorda che ogni sketch in input non è altro che una tabella di sequenze di tratti.
4 L'output ad ogni time-step viene riportato come input per il time-step successivo.
5 Questo garantisce che i valori siano compresi fra -1 e 1.

(2.2)

Nell'equazione 2.2 è mostrato come la rete modella $(\Delta x, \Delta y)$: attraverso un modello a mistura Gaussiana 1.11 (GMM - Gaussian mixture model), con M distribuzioni normali [27]. (q_1, q_2, q_3) invece, sono presi come distribuzione categorica allo scopo di modellare i dati reali (p_1, p_2, p_3), dove $(q_1 + q_2 + q_3 = 1)$[6][7].

$\mathcal{N}(\Delta x, \Delta y | \mu_{x,j}, \mu_{y,j}, \sigma_{x,j}, \sigma_{y,j}, \rho_{xy,j})$ è la funzione di distribuzione di probabilità per una distribuzione normale bivariata. Ognuna delle M distribuzioni normali bivariate consiste di cinque parametri: $(\mu_x, \mu_y, \sigma_x, \sigma_y, \rho_{xy})$ dove μ_x, μ_y sono le medie, σ_x, σ_y sono le deviazioni standard e ρ_{xy} è il corrispondente parametro di correlazione. Il vettore Π di lunghezza M, a sua volta considerato come una distribuzione categorica, corrisponde ai pesi delle distribuzioni nel GMM. Come conseguenza di questa struttura, si deduce che l'output del decoder debba avere dimensione $5M + M + 3$.[8]

Il successivo hidden state della RNN nel decoder, è proiettato nel vettore di output y_i attraverso uno strato densamente connesso:

$$x_i = [S_{i-1}; z], [h_i; c_i] = forward(x_i, [h_{i-1}; c_{i-1}]), y_i = W_y h_i + b_y, y \in \mathbb{R}^{6M+3} \tag{2.3}$$

Il vettore y_i è poi suddiviso nei parametri della distribuzione di probabilità per il prossimo punto nei dati:

$$[(\hat{\Pi}_1, \mu_x, \mu_y, \hat{\sigma}_x, \hat{\sigma}_y, \hat{\rho}_{xy})_1, ..., (\hat{\Pi}_M, \mu_x, \mu_y, \hat{\sigma}_x, \hat{\sigma}_y, \hat{\rho}_{xy})_M (\hat{q}_1, \hat{q}_2), \hat{q}_3)] = y_i \tag{2.4}$$

Per assicurarsi che la deviazione standard non risulti negativa e che il valore di correlazione sia nell'intorno (-1, 1), si applicano gli operatori esponenziali ai $\hat{\sigma}$ e tangente iperbolica ai $\hat{\rho}$:

$$\sigma_x = \exp(\hat{\sigma}_x), \sigma_y = \exp(\hat{\sigma}_y) \implies \sigma_x, \sigma_y > 0 \tag{2.5}$$

$$\rho_x y = \tanh(\hat{\rho}_x y) \implies \rho_x y \in (-1, 1) \tag{2.6}$$

6 Come in [31] e [32].

7 Si ricorda che la sequenza generata è condizionata ad una variabile latente z campionata dall'encoder.

8 dove il primo termine indica i parametri di ogni distribuzione, il secondo la lunghezza di Π e il terzo corrisponde ai logit per generare (q_1, q_2, q_3)

Le probabilità per le distribuzioni categoriche, invece, sono calcolate attraverso un'operazione di *softmax*:

$$q_k = \frac{\exp(\hat{q}_k)}{\sum_{j=1}^{3} \exp(\hat{q}_j)}, k \in 1,2,3, \implies q_k \in (0,1), \sum_j q_k = 1 \tag{2.7}$$

$$\Pi_k = \frac{\exp(\hat{\Pi}_k)}{\sum_{j=1}^{M} \exp(\hat{\Pi}_j)}, k \in 1,...,M \implies \Pi_k \in (0,1), \sum_j \Pi_k = 1 \tag{2.8}$$

```
def get_mixture_coef(output):
    out_pi = output[:, :20]
    out_mu_x = output[:, 20:40]
    out_mu_y = output[:, 40:60]
    out_sigma_x = output[:, 60:80]
    out_sigma_y = output[:, 80:100]
    out_ro = output[:, 100:120]
    pen_logits = output[:, 120:123]
    # use softmax to normalize pi and q into prob distribution
    out_pi = K.exp(out_pi)
    normalize_pi = 1 / (K.sum(out_pi, axis=1, keepdims=True))
    out_pi = normalize_pi * out_pi
    out_q = K.exp(pen_logits)
    normalize_q = 1 / (K.sum(out_q, axis = 1, keepdims = True))
    out_q = normalize_q * out_q
    out_ro = K.tanh(out_ro)
    # use exponential to make sure sigma is positive
    out_sigma_x = K.exp(out_sigma_x)
    out_sigma_y = K.exp(out_sigma_y)
    return out_pi, out_mu_x, out_mu_y, out_sigma_x, out_sigma_y,
        out_ro, pen_logits, out_q
```

Listing 2.1: Implementazione in Keras del metodo per l'estrazione e la normalizzazione dei parametri della distribuzione

Una volta ottenuti i parametri appropriati, diventa possibile calcolare le distribuzioni normali bivariate, tramite:

$$\mathcal{N}(\Delta x, \Delta y | \mu_x, \mu_y, \sigma_x, \sigma_y, \rho_{xy}) = \frac{\exp(\frac{-Z}{2(1-\rho_{xy}^2)})}{2\pi\sigma_x\sigma_y\sqrt{1-(\rho_{xy})^2}} \tag{2.9}$$

con:

$$Z = \frac{(\Delta x - \mu_x)^2}{\sigma_x^2} + \frac{(\Delta y - \mu_y)^2}{\sigma_y^2} - \frac{\rho_{xy}((\Delta x - \mu_x)(\Delta y - \mu_y))}{\sigma_x\sigma_y} \tag{2.10}$$

```
def tf_bi_normal(x, y, mu_x, mu_y, sigma_x, sigma_y, ro):
    norm1 = x_ - mu_x
    norm2 = y_ - mu_y
    sigma = sigma_x * sigma_y
    z = (K.square(norm1 / (sigma_x + 1e-8)) + K.square(norm2 /
        (sigma_y + 1e-8)) - (2 *
         ro * norm1 * norm2) / (sigma + 1e-8) + 1e-8)
    ro_opp = 1 - K.square(ro)
    result = K.exp(-z / (2 * ro_opp + 1e-8))
    denom = 2 * np.pi * sigma * K.square(ro_opp) + 1e-8
    result = result / denom + 1e-8
    return result
```

Listing 2.2: Implementazione in Keras del calcolo della normale bivariata

Sempre in accordo a [1], un problema chiave dell'apprendimento sta nello stabilire quando il modello debba smettere di disegnare. Le probabilità dei tre tipi di tratto [9] sono molto sbilanciate e ciò rende il modello difficile da addestrare. La probabilità di un evento p_1 sono molto più alte di quelle di un evento p_2 e l'evento p_3 avviene una volta sola per disegno. L'approccio seguito in alcuni lavori[10] è stato quello di pesare differentemente ogni evento della penna nel calcolo dell'errore, ad esempio imponendo manualmente i valori (1, 10, 100). In sketch-rnn è stato scelto un approccio più robusto e funzionale: tutte le sequenze sono generate dal modello fino ad N_{max}, che è la lunghezza del disegno più lungo del training set. Dato che la lunghezza del generico sketch S è tipicamente minore di N_{max}, S_i è impostato a (0, 0, 0, 0, 1) per ogni $i > N_s$2.3.

Dopo il training è possibile campionare disegni dal modello, questo procedimento è effettuato utilizzando il decoder in modo autoregressivo: ad ogni time-step vengono generati i parametri sia del GMM che della distribuzione categorica, tramite i quali si ricava un punto S'_i, questo viene poi concatenato all'input[11] del time step seguente. Il procedimento prosegue finché $p_3 = 1$ o quando viene raggiunto $i = N_{max}$.

Come per l'encoder, l'output ottenuto in questo modo non è deterministico, si tratta di una sequenza casuale condizionata al vettore di latenza.

9 Rispettivamente: penna sul foglio, penna sollevata e fine del disegno.
10 Vedere [31] e [32].
11 Una variabile casuale di dimensione N_z.

Il livello di casualità puà essere controllato introducendo un parametro di temperatura τ:

$$\hat{q}_k \to \frac{\hat{q}_k}{\tau}, \hat{\Pi}_k \to \frac{\hat{\Pi}_k}{\tau}, \sigma_x^2 \to \sigma_x^2\tau, \sigma_y^2 \to \sigma_y^2\tau \tag{2.11}$$

Questo valore può essere utilizzato sui parametri delle softmax della distribuzione categorica e sulle deviazioni standard della normale bivariata, il parametro è tipicamente scelto fra 0 e 1, nel caso in cui $\tau = 0$ il modello diventa deterministico e i punti generati risulteranno essere i punti più probabili della funzione di densità di probabilità.

2.4.1 *training*

La procedura di training segue l'approccio di un VAE [23], dove la funzione di loss è composta dalla somma di due termini: *Reconstruction Loss* (l'errore di ricostruzione), L_R, e la divergenza di Kullback-Leibler, L_{KL}. il termine dell'errore di ricostruzione 2.14 massimizza la verosimiglianza logaritmica della distribuzione di probabilità generata dalla rete, nel descrivere l'oggetto di training S. Si può calcolare l'errore di ricostruzione utilizzando i parametri generati dalla PDF e il dato di training, in questo modo otteniamo una descrizione dell'errore attraverso due componenti: la somma del logaritmo degli errori sui termini di distanza, L_s 2.12, e la somma logaritmica degli errori sugli stati della penna (p_1, p_2, p_3), L_p 2.13

$$L_s = -\frac{1}{N_max} \sum_{i=1}^{N_s} \log\left(\sum_{j=1}^{M} \Pi_{j,i} \mathcal{N}(\Delta x_i, \Delta y_i | \mu_{x,j,i}, \mu_{y,j,i}, \sigma_{x,j,i}, \sigma_{y,j,i}, \rho_{xy,j,i})\right) \tag{2.12}$$

$$L_p = -\frac{1}{N_max} \sum_{i=1}^{N_max} \sum_{k=1}^{3} p_{k,i} \log(q_{k,i}) \tag{2.13}$$

$$L_R = L_s + L_p \tag{2.14}$$

Si nota che, nel calcolo della sommatoria sull'errore di offset, l'indice si ferma a N_S, scartando tutti i valori successivi, mentre L_p è calcolata usando tutti i parametri che modellano (p_1, p_2, p_3) fino a N_{max}. Questo

metodo di calcolo della perdita risulta più robusto e permette al modello di apprendere quando smettere di disegnare, a differenza del metodo citato precedentemente di soppesare diversamente i valori p_i 2.4.

```
def get_lossfunc(out_pi, out_mu_x, out_mu_y, out_sigma_x,
    out_sigma_y, out_ro, out_q, x, y, logits):
    # L_r loss term calculation, L_s part
    result = tf_bi_normal(x, y, out_mu_x, out_mu_y, out_sigma_x,
        out_sigma_y, out_ro)
    result = result * out_pi
    result = K.sum(result, axis=1, keepdims=True)
    result = -K.log(result + 1e-8)
    fs = 1.0 - logits[:, 2]
    fs = K.reshape(fs, (-1, 1))
    result = result * fs
    # L_r loss term, L_p part
    result1 = K.categorical_crossentropy(out_q, logits, from_logits
        = True)
    result1 = K.reshape(result1, (-1, 1))
    result = result + result1
    return K.mean(result)
```

Listing 2.3: Implementazione in Keras del calcolo dell'errore di ricostruzione, suddiviso fra i termini dell'errore nell'offset (L_s) e l'errore degli stati della penna (L_p)

Il termine di loss della divergenza KL misura la differenza fra la distribuzione del vettore latente z e un vettore di distrubizioni Gaussiane IID con media zero e varianza unitaria. Ottimizzare secondo questo termine permette di minimizzare questa differenza. Sempre secondo i risultati di [23]:

$$L_{KL} = -\frac{1}{2N_z}(1 + \hat{\sigma} - \mu^2 - \exp(\hat{\sigma})) \tag{2.15}$$

La funzione di loss complessiva non è altro che la somma pesata dei sopracitati termini L_R e L_{KL}, ovvero:

$$\text{Loss} = L_R + w_{KL} L_{KL} \tag{2.16}$$

Esiste un compromesso fra l'ottimizzazione secondo un termine rispetto all'altro: per $w_{KL} \to 0$, il modello si avvicina a quello di un autoencoder puro, sacrificando l'abilità di forzare una distribuzione a priori sullo spazio di latenza, ottenendo migliori risultati nella ricostruzione. Si noti che nella generazione non condizionale, dove il modello è costituito dal

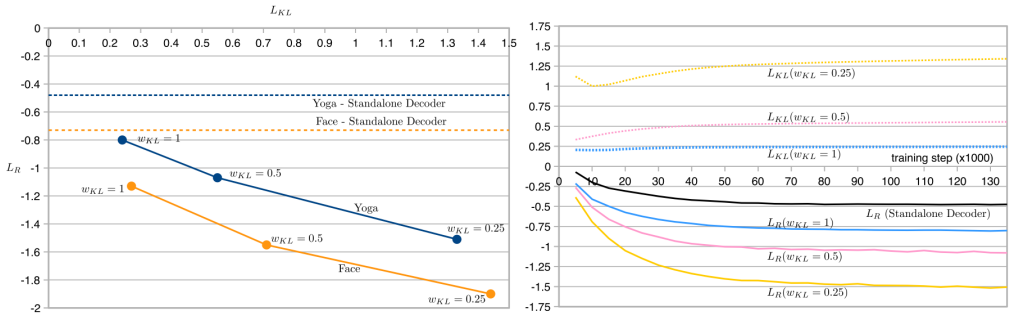


Figura 20: Compromesso fra L_R e L_{KL}, su due modelli addestrati su dataset con singola classe(sinistra). Il grafo della *Validation Loss* per modelli addestrati sulla classe Yoga, al variare del peso w_{KL}, da [1]

solo decoder, non sarà presente il termine L_{KL}. In fig. 20 è illustrato il compromesso, ottenuto al variare di w_{KL}, fra l'errore di ricostruzione e la divergenza KL sul test set, confrontato (a destra) con un modello costituito dal solo decoder. Dato che un modello non condizionale non riceve informazioni precedenti riguardo al disegno che deve generare, la metrica L_R dei modelli a decoder è utilizzata come limite superiore per i modelli condizionali a variabile latente.

3

ESPERIMENTI

3.1 INTRODUZIONE

In questo elaborato si è scelto di riprodurre ed analizzare i risultati presentati da Ha e Eck attraverso tre esperimenti: nel primo è stata scelta la configurazione standard del modello, con un VAE completo e layer LSTM sia nell'encoder che nel decoder, la rete è stata addestrata su due dataset: *cat.npz* (gatti) e *flying_saucer.npz* (dischi volanti). Nel secondo è stata scelta la configurazione con il solo decoder, utilizzato come modello autoregressivo non condizionato ad una variabile latente (con i pesi inizializzati a zero), in questo caso il layer utilizzato è HyperLSTM[1], che eccelle nella generazione di sequenze, il dataset è stato addestrato sul solo dataset *owl.npz* (gufi), infine è stato addestrato un modello con *Layer Normalization* nell'encoder e HyperLSTM nel decoder[2], per poter gestire un ampio training set composto da tre categorie: *elephant* (elefanti), *hat* (cappelli) e *snake* serpenti[3]. Di seguito verranno illustrate e confrontate le reti neurali prodotte da queste configurazioni, inoltre verranno mostrate alcune immagini generate tramite diversi approcci che mostreranno la capacità del modello di concettualizzare le proprietà di un disegno.

1 Questo layer non è stato approfondito nell'introduzione di questo elaborato, si rimanda a [33] per ulteriori informazioni.

2 Soluzione suggerita in `https://github.com/tensorflow/magenta/tree/master/magenta/models/sketch_rnn`

3 In omaggio a [35]

```
num_steps=10000000,              # Total number of training set. Keep large.
save_every=500,                  # Number of batches per checkpoint creation.
dec_rnn_size=512,                # Size of decoder.
dec_model='lstm',                # Decoder: lstm, layer_norm or hyper.
enc_rnn_size=256,                # Size of encoder.
enc_model='lstm',                # Encoder: lstm, layer_norm or hyper.
z_size=128,                      # Size of latent vector z. Rec. 32, 64 or 128.
kl_weight=0.5,                   # KL weight of loss. Recommend 0.5 or 1.0.
kl_weight_start=0.01,            # KL start weight when annealing.
kl_tolerance=0.2,                # Level of KL loss at which to stop optimizing
batch_size=100,                  # Minibatch size. Recommend leaving at 100.
grad_clip=1.0,                   # Gradient clipping. Recommend leaving at 1.0.
num_mixture=20,                  # Number of mixtures in Gaussian mixture model.
learning_rate=0.001,             # Learning rate.
decay_rate=0.9999,               # Learning rate decay per minibatch.
kl_decay_rate=0.99995,           # KL annealing decay rate per minibatch.
min_learning_rate=0.00001,       # Minimum learning rate.
use_recurrent_dropout=True,      # Recurrent Dropout without Memory Loss.
recurrent_dropout_prob=0.90,     # Probability of recurrent dropout keep.
use_input_dropout=False,         # Input dropout. Recommend leaving False.
input_dropout_prob=0.90,         # Probability of input dropout keep.
use_output_dropout=False,        # Output droput. Recommend leaving False.
output_dropout_prob=0.90,        # Probability of output dropout keep.
random_scale_factor=0.15,        # Random scaling data augmention proportion.
augment_stroke_prob=0.10,        # Point dropping augmentation proportion.
conditional=True,                # If False, use decoder-only model.
```

Listing 3.1: Iperparametri standard di sketch-rnn

In questa sezione di codice sono mostrati gli iperparametri del modello, coi loro valori di default. Si può notare che $N_z = 128$ e che decoder e encoder sono simmetrici[4]. A partire da questi valori, viene assemblata la rete neurale mostrata in tabella 1.

Nel caso della seconda rete la dimensione del layer ricorrente è sempre 512 ma, per la maggior complessità dei parametri interni, si ottiene una rete com 2218363 variabili addestrabili. L'ultimo modello, infine, presenta 24515195 variabili addestrabili con la dimensione dell'output del decoder pari a 2048 e quella dell'encoder pari a 512 (1024)[5].

Con queste reti sono stati condotti vari esperimenti, in particolare: nei modelli condizionali sono state svolte interpolazioni nello spazio di latenza, sia all'interno di una singola classe sia fra classi diverse, inoltre è stato osservato l'esito della variazione del parametro di temperatura sia su interpolazioni che su generazioni condizionali semplici ed in special modo nel modello autoregressivo, sul quale ha un impatto rilevante. Di seguito i test effettuati saranno illustrati in dettaglio e commentati.

4 Si ricordi che l'encoder è un layer bidirezionale, di conseguenza la dimensione dello stato finale, dopo la concatenazione, risulta essere 512.

5 Si omette la tabella, riportando solo le dimensioni fondamentali, per la complessità di lettura derivante dalla struttura dell'HyperLSTM.

Layer	Output shape	Variabili addestrabili
Input	(?, 250, 5)	0
Forward LSTM	(?, 256)	268288
Backward LSTM	(?, 256)	268288
Mu	(?, 128)	65664
Log Sigma	(?, 128)	65664
State initializer	(?, 1024)	132096
Decoder LSTM	(?, 512)	1323008
Output	(?, 123)	63099
Numero di variabili totale: 2186107		

Tabella 2: I layer ottenuti attraverso la configurazione standard.

3.1.1 *Generazione condizionale*

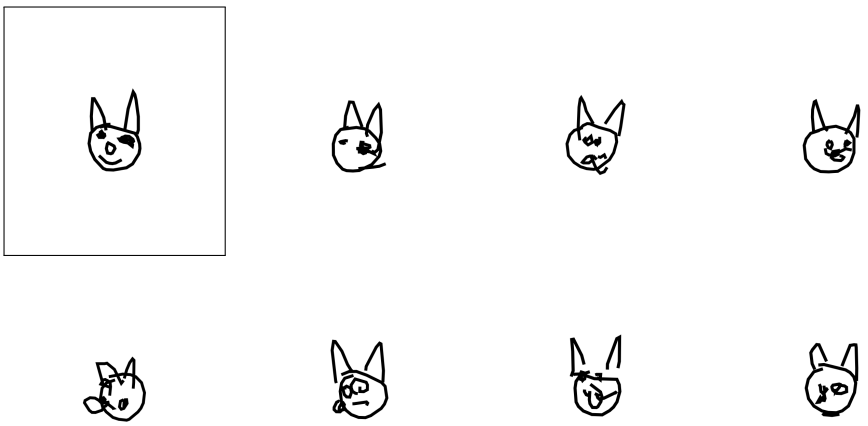


Figura 21: La prima immagine è estratta dal dataset, le altre sono immagini originali generate dal modello standard

Questo primo esempio 21 illustra un risultato semplice, quanto interessante: la figura nel riquadro è stata passata come input alla prima rete neurale, le successive sono immagini generate dalla rete attraverso un parametro di temperatura di 0.8, condizionate allo stesso input. L'ampiezza del parametro di temperatura permette alla rete una maggiore

libertà 2.4, aumentando la varianza sull'offset dei punti[6], il che mostra le potenzialità del modello nel creare sketch simili ma unici ed originali, a partire da un singolo input.

3.1.2 *Generazione non condizionale*

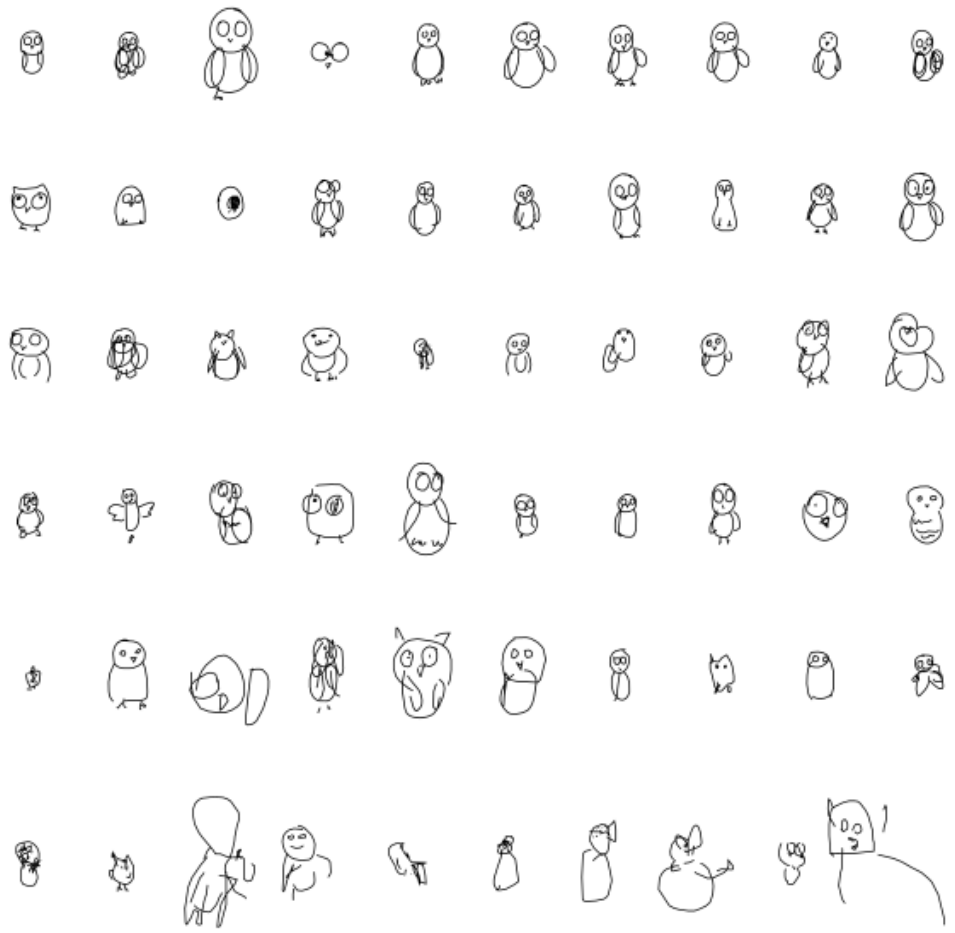


Figura 22: Immagini generate dal modello autoregressivo al variare del parametro di temperatura

6 Al prezzo di immagini maggiormente confuse.

4

CONCLUSIONI

4.1 LAVORI CORRELATI

4.2 SVILUPPI FUTURI

BIBLIOGRAFIA

[1] David Ha, Douglas Eck - *A Neural Representation of Sketch Drawings* - arXiv:1704.03477, 2017. (Cited on pages 4, 7, 32, 36, and 39.)

[2] J. Jongejan, H. Rowley, T. Kawashima, J. Kim, and N. Fox-Gieg. - *The Quick, Draw! - A.I. Experiment.* - `https://quickdraw.withgoogle.com/`, 2016. (Cited on pages 7 and 31.)

[3] Chollet, François and others - *Keras* - GitHub, `https://github.com/keras-team/keras`. (Cited on pages 7 and 29.)

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. - *TensorFlow: Large-scale machine learning on heterogeneous systems* - `tensorflow.org`, 2015. (Cited on pages 7 and 29.)

[5] I. Goodfellow. - *NIPS 2016 Tutorial: Generative Adversarial Networks.* - ArXiv e-prints, Dec. 2017. (Cited on page 8.)

[6] D. P. Kingma and M. Welling. - *Auto-Encoding Variational Bayes.* - ArXiv e-prints, Dec. 2013. (Cited on page 8.)

[7] S. Reed, A. van den Oord, N. Kalchbrenner, S. Gómez Colmenarejo, Z. Wang, D. Belov, and N. de Freitas. - *Parallel Multiscale Autoregressive Density Estimation.* - ArXiv e-prints, Mar. 2017. (Cited on page 8.)

[8] Dustin Tran and Alp Kucukelbir and Adji B. Dieng and Maja Rudolph and Dawen Liang and David M. Blei - *Edward: A library for probabilistic modeling, inference, and criticism* - arXiv preprint arXiv:1610.09787, 2016.

[9] Axel Brando - *Mixture Density Networks (MDN) for distribution and uncertainty estimation* - `https://github.com/axelbrando/Mixture-Density-Networks-for-distribution-and-uncertainty-estimation/blob/master/ABrando-MDN-MasterThesis.pdf`, 2017.

[10] Colah - *Understanding lstm networks, colah's blog.* - `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[11] G. Cybenko - *Approximation by Superpositions of a Sigmoidal Function* - 1989. (Cited on page 9.)

[12] A. Karpathy - *The Unreasonable Effectiveness of Recurrent Neural Networks* - `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`, 2015. (Cited on pages 3 and 13.)

[13] S. Hochreiter and J. Schmidhuber *Long short-term memory* - Neural computation, 9 1997, pp. 1735. (Cited on page 14.)

[14] F. Gers, J. Schmidhuber, et al. - *Recurrent nets that time and count* - Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on, vol. 3, IEEE, 2000, pp. 189194. (Cited on page 15.)

[15] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio - *Learning phrase representations using rnn encoder-decoder for statistical machine translation* - arXiv preprint arXiv:1406.1078, 2014. (Cited on page 15.)

[16] K. Yao, T. Cohn, K. Vylomova, K. Duh, and C. Dyer - *Depth-gated recurrent neural networks* - arXiv preprint arXiv:1508.03790, 2015. (Cited on page 15.)

[17] M. Schuster, K. K. Paliwal - *Bidirectional recurrent neural networks.* - IEEE Transactions on Signal Processing, 1997 (Cited on pages 16 and 18.)

[18] J. O. Berger - *Statistical Decision Theory and Bayesian Analysis.* - Berlin, Germany: Springer-Verlag, 1985. (Cited on page 16.)

[19] R. A. Jacobs - *Methods for combining experts' probability assessments* - Neural Comput., vol. 7, no. 5, pp. 867–888, 1995. (Cited on page 16.)

[20] A. J. Robinson - *An application of recurrent neural nets to phone probability estimation* - IEEE Trans. Neural Networks, vol. 5, pp. 298–305, Apr. 1994. (Cited on page 16.)

[21] *Improved phone modeling with recurrent neural networks* - Proc. IEEE Int. Conf. Acoust., Speech, Signal Process., vol. 1, 1994, pp. 37–40. (Cited on page 16.)

[22] Francois Chollet - *Building Autoencoders in Keras* - `https://blog.keras.io/building-autoencoders-in-keras.html`, 2016. (Cited on pages 3 and 18.)

[23] . P. Kingma and M. Welling. - *Auto-Encoding Variational Bayes.* - ArXiv e-prints, Dec. 2013. (Cited on pages 21, 37, and 38.)

[24] C. Doersch - *Tutorial on Variational Autoencoders* - arXiv preprint arXiv:1606.05908v2, 2016. (Cited on pages 3 and 21.)

[25] Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery - *Numerical Recipes in C: The Art of Scientific Computing* - (1992), (Second ed.). Cambridge University Press. (Cited on page 24.)

[26] Luc Devroye - *Sample-based non-unifmro random variate generation* - Springer-Verlag, New York, 1986. (Cited on page 22.)

[27] C. M. Bishop. - *Mixture density networks. Technical Report* - 1994. (Cited on pages 25 and 34.)

[28] K. E. B. McLachlan G. J, - *Mixture models: Inference and applications to clustering* - (1988). (Cited on page 25.)

[29] A. Graves. - *Generating sequences with recurrent neural networks.* - arXiv:1308.0850, 2013 (Cited on page 31.)

[30] D. H. Douglas and T. K. Peucker. - *Algorithms for the reduction of the number of points required to represent a digitized line or its caricature.* - Cartographica: The International Journal for Geographic Information and Geovisualization, 10(2):112–122, Oct. 1973. (Cited on page 32.)

[31] D. Ha. *Recurrent Net Dreams Up Fake Chinese Characters in Vector Format with TensorFlow* - 2015. (Cited on pages 34 and 36.)

[32] X. Zhang, F. Yin, Y. Zhang, C. Liu, and Y. Bengio. *Drawing and Recognizing Chinese Characters with Recurrent Neural Network.* - CoRR, abs/1606.06539, 2016. (Cited on pages 34 and 36.)

[33] D. Ha, A. M. Dai, and Q. V. Le. - *HyperNetworks.* - In ICLR, 2017. (Cited on page 41.)

[34] Fariz Rahman - *Recurrent Shop* - `https://github.com/farizrahman4u/recurrentshop`, GitHub, 2017. (Cited on page 30.)

[35] Antoine de Saint-Exupéry - *Il piccolo principe* - ISBN 978-88-909042-4-0. (Cited on page 41.)

4.3 LISTA DI ACRONIMI

MLN	Multi-Layer Network
MLP	Multi-Layer Perceptron
RNN	Recurrent Neural Network
LSTM	Long Short Term Memory
BRNN	Bidirectional Recurrent Neural Network
BPTT	Back Propagation Through Time
SGD	Stochastic Gradient Descent
SGA	Stochastic Gradient Ascent
MNIST	Modified National Institute of Standards and Technology
PDF	Probability Density Function
CDF	Cumulative Distribution Function
KL	Kullback-Leibler Divergence
VAE	Variational Auto Encoder
GMM	Gaussian Mixture Model
MDN	Mixture Density Network
API	Application Programming Interface