UNIVERSITÀ
DEGLI STUDI
FIRENZE

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 2017-2018

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 proposta un'implementazione in *Keras* [3], un framework che a sua volta poggia su *tensorflow* [4], che è la libreria software utilizzata per il lavoro originale.

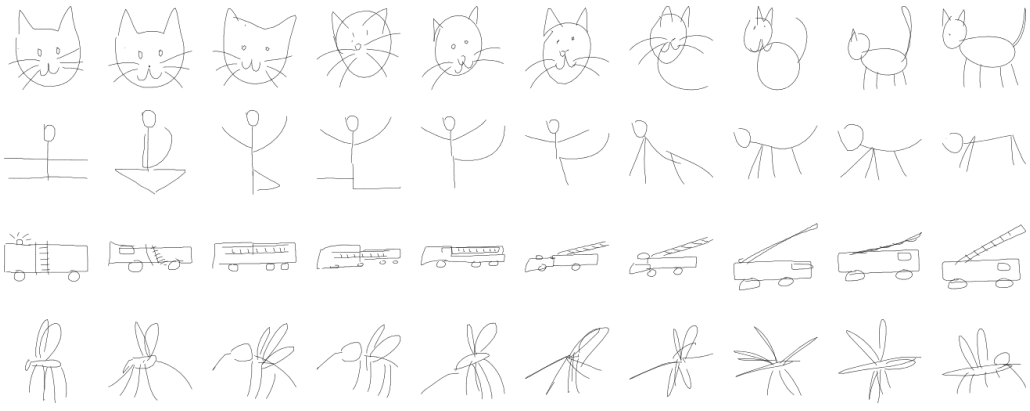


Figura 1: Interpolazioni nello spazio di latenza di immagini vettoriali generate dal modello.

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 LAVORI CORRELATI

1.4 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.

1.5 RETI DENSAMENTE CONNESSE

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

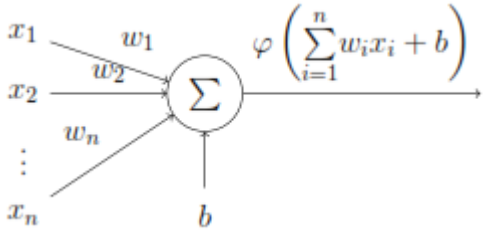


Figura 2: Rappresentazione di un neurone artificiale.

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 funzione 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 $\mathbb{R}^n$.

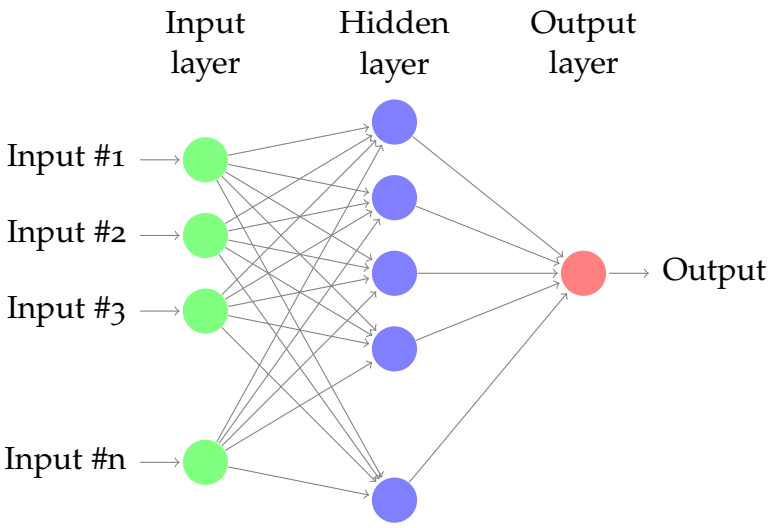


Figura 3: Rappresentazione di un modello Multi-Strato.

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

1.6 RETI RICORRENTI

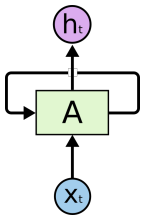


Figura 4: Una semplice RNN (Recurrent Neural Network).

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.

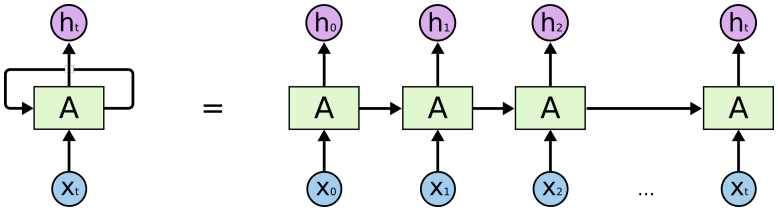
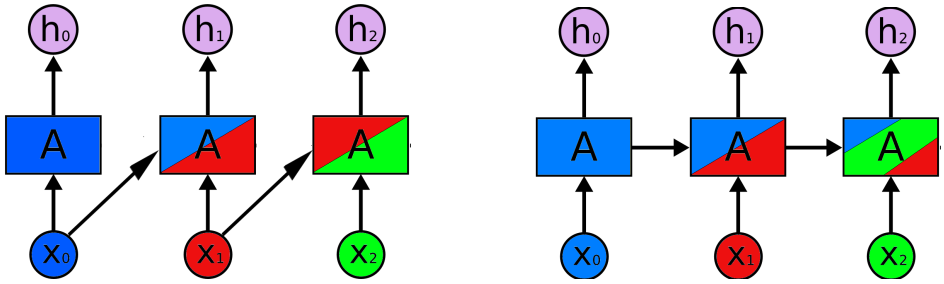


Figura 5: Una RNN dispiegata lungo la linea temporale.

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 ricor-



(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

renza 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 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.

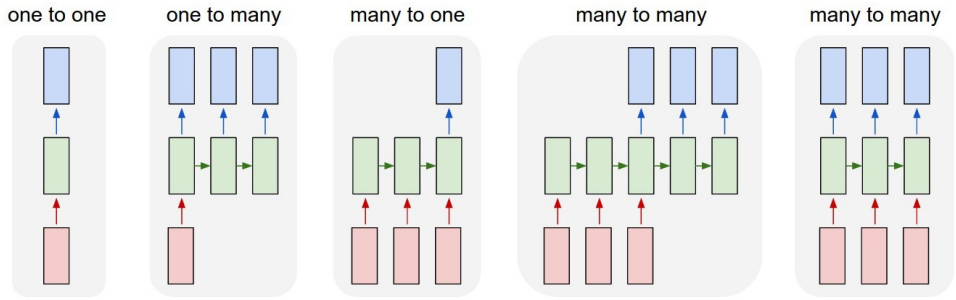


Figura 7: Combinazioni di sequenze vettoriali. [12]

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.

1.6.1 *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, 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

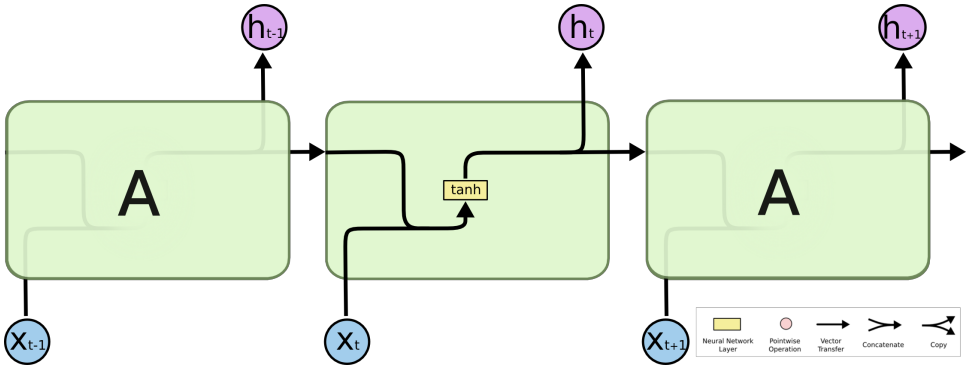


Figura 8: Struttura interna di una RNN standard con un singolo hidden layer.

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.

1.6.2 *Long Short Term Memory*

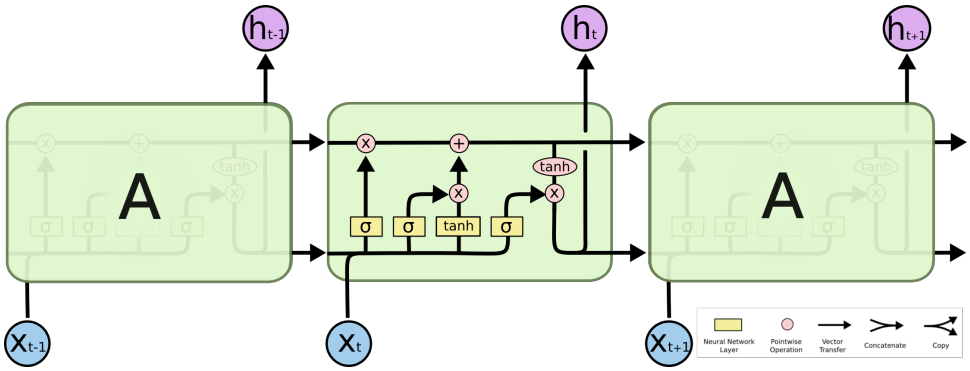


Figura 9: Struttura interna di una lstm che evidenzia i quattro strati interni di un layer

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]

1.6.3 *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.

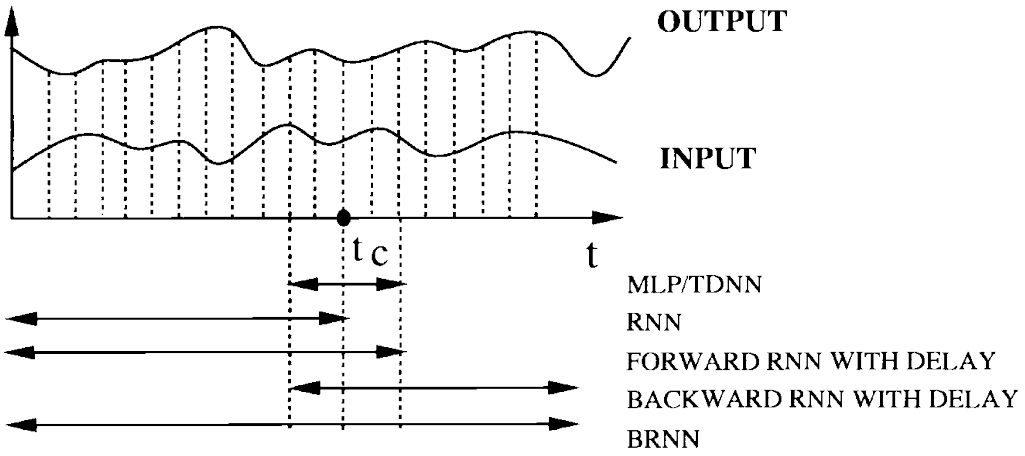


Figura 10: Confronto sull'utilizzo dell'input in diverse reti neurali.

1.6.4 *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.

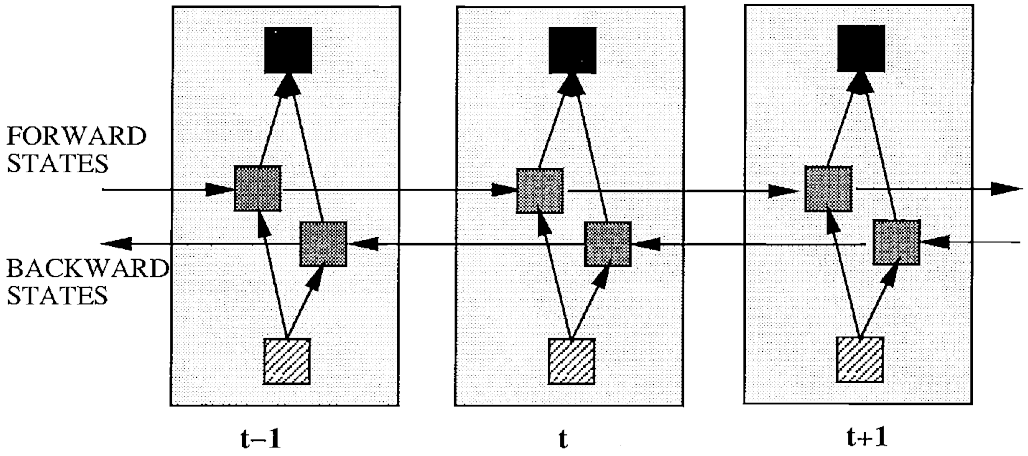


Figura 11: Struttura generica di una BRNN, svolta lungo tre time steps.

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].

1.7 AUTOENCODER

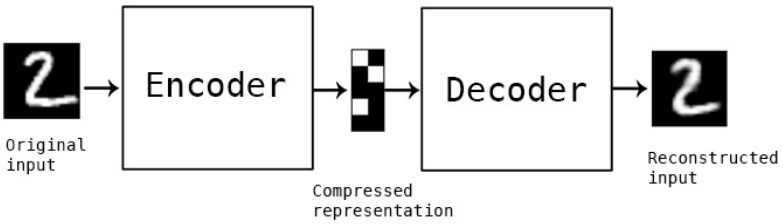


Figura 12: Struttura di un generico AutoEncoder, da [22]

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).

1.8 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 massimiziamo 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.

1.9 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 $f : Z x \Theta \rightarrow \chi$. 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.

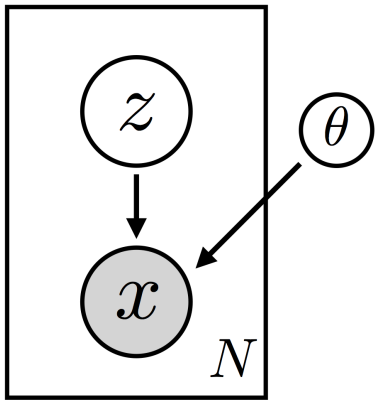


Figura 13: Modello di un VAE

1.10 VARIATIONAL AUTOENCODER

I variational autoencoder 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) 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 $Q(z|\mathbf{x}) = \mathcal{N}(z|\boldsymbol{\mu}(\mathbf{x}; \Theta), \Sigma(\mathbf{x}; \Theta))$, 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. [25]

μ e Σ 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 *divergenza di Kullback - Leibler* (KL divergence) fra $Q(z|x)$ e $P(z)$:

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

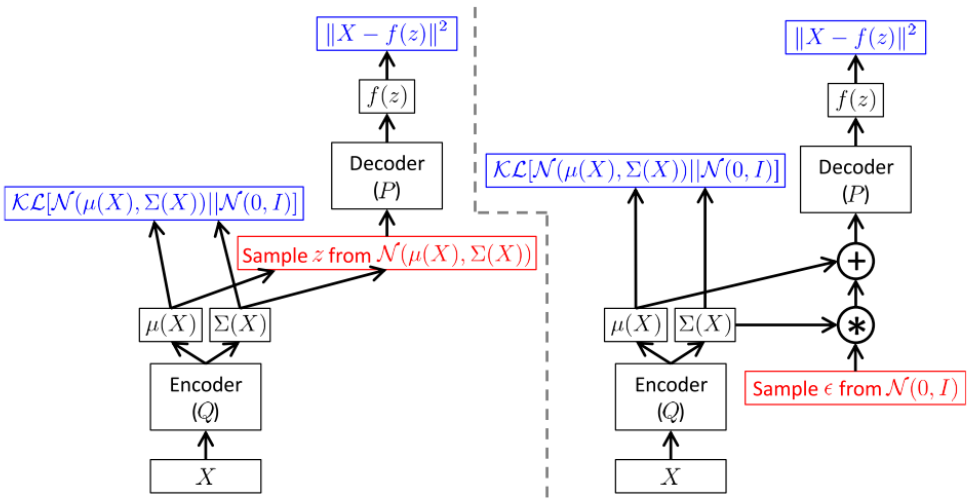


Figura 14: Un VAE per intero, a sinistra senza il "reparametrization trick", a destra con. In rosso le operazioni non differenziabili, in blu le loss. Queste due varianti differiscono per il fatto che la retropropagazione si può applicare solo sulla rete a destra.

1.11 MDN

1.11.1 *Misture gaussiane*

2

STRUMENTI

2.1 LIBRERIE SOFTWARE

2.1.1 *Keras*

2.1.2 *Recurrentshop*

2.2 DATASET

3

ESPERIMENTI

3.1 KERAS SKETCH-RNN

```
class MixtureDensity(Layer):
    def __init__(self, kernelDim, numComponents, **kwargs):
        self.kernelDim = kernelDim
        self.numComponents = numComponents
        super(MixtureDensity, self).__init__(**kwargs)

    def build(self, inputShape):
        self.inputDim = inputShape[1]
        self.outputDim = self.numComponents * (1 + self.kernelDim)
            + 3
        self.Wo = K.variable(np.random.normal(scale=0.5, size =
            (self.inputDim, self.outputDim)))  # TODO check if
            random.normal is a meaningful choice
        self.bo = K.variable(np.random.normal(scale=0.5, size =
            self.outputDim))

        self.trainable_weights = [self.Wo, self.bo]

    def call(self, x, mask=None):
        output = K.dot(x, self.Wo) + self.bo
        return output

    def compute_output_shape(self, inputShape):
        return inputShape[0], self.outputDim
```

Listing 3.1: Il layer personalizzato che implementa l'output per la mistura gaussiana

```
class Vae:
 max_len = 250
 input = Input((max_len, 5,))
 decoder_input = Input(shape = (133,))
 h_in = Input(shape = (512,))
 c_in = Input(shape = (512,))
 readout_in = Input(shape = (133,))
 enc_1 = Bidirectional(LSTM(256))
 enc_mean = Dense(128)
 enc_log_sigma = Dense(128)
 h_init = Dense(1024)
 dec_1 = Dense(5)
 dec_2 = Dense(123)
 dec_3 = LSTMCell(512)
 mdn = MixtureDensity(5, 20)

 def __init__(self, generate = False):
     self.h_out = None
     self.c_out = None

     if not generate:
         self.encoder = self.build_encoder()
         self.mean, self.log_sigma = self.encoder(self.input)
         self.z = Lambda(self.sampling)([self.mean,
             self.log_sigma])
     else:
         self.z = Input(shape = (128,))

     if self.h_out is None:
         _h = self.h_init(self.z)

         _h = Reshape((512, 2,))(_h)

         _h_1 = Lambda(lambda x: x[:, :, 0])(_h)
         _c_1 = Lambda(lambda x: x[:, :, 1])(_h)
     else:
         _h_1 = self.h_out
         _c_1 = self.c_out

     z_ = Reshape((1, 128,))(self.z)
     z_ = Lambda(self.tile)(z_)
```

```
        z_ = concatenate([z_, self.input], axis = 2)

        dec_out, h, c = self.dec_3([self.decoder_input, self.h_in,
            self.c_in])

        dec_out = self.mdn(dec_out)

        rnn = RecurrentModel(input = self.decoder_input,
                             initial_states = [self.h_in,
                                 self.c_in],
                             output = dec_out, final_states = [h,
                                 c],
                             readout_input = self.readout_in,
                             return_sequences = True, return_states
                                 = True)

        out, self.h_out, self.c_out = rnn(z_, initial_state =
            [_h_1, _c_1])

        if generate:
            self.vae = Model([self.z, self.input], out)
        else:
            self.vae = Model(self.input, out)

    def tile(self, tensor):
        return K.tile(tensor, [1, self.max_len, 1])

    def build_encoder(self):
        a = self.enc_1(self.input)
        mean = self.enc_mean(a)
        log_sigma = self.enc_log_sigma(a)
        encoder = Model(self.input, [mean, log_sigma])
        return encoder

    def sampling(self, args):
        z_mean, z_log_sigma = args
        epsilon = K.random_normal(shape = (128,), mean = 0.,
                                  stddev = 1.0)
        return z_mean + K.exp(z_log_sigma) * epsilon
```

Listing 3.2: Il modello completo del VAE, composto da LSTM

```
  class Compiler:

   batch_size = 100
   original_dim = 250
   epochs = 3

   def __init__(self, names, generate = False):
       self.vae = vae.Vae(generate)
       if not generate:
           self.z_mean = self.vae.mean
           self.z_log_sigma = self.vae.log_sigma
       self.x_train = None
       self.x_valid = None
       self.x_test = None
       self.load_dataset(names)

   def load_dataset(self, names):
       for name in names:
           dataset = np.load("Dataset/" + name + ".full.npz",
               encoding = 'bytes')
           if self.x_train is None:
               self.x_train = dataset["train"][:2500]
               self.x_valid = dataset["valid"]
               self.x_test = dataset["test"]
           else:
               self.x_train = np.concatenate((self.x_train,
                   dataset["train"][:2500]))
               self.x_valid = np.concatenate((self.x_valid,
                   dataset["valid"]))
               self.x_test = np.concatenate((self.x_test,
                   dataset["test"]))

       self.x_train = dl.DataLoader(self.x_train, self.batch_size)
       normal_scale_factor =
           self.x_train.calculate_normalizing_scale_factor()
       self.x_train.normalize(normal_scale_factor)

       self.x_valid = dl.DataLoader(self.x_valid, self.batch_size)
       self.x_valid.normalize(normal_scale_factor)

       self.x_test = dl.DataLoader(self.x_test, self.batch_size)
       self.x_test.normalize(normal_scale_factor)
```

```
def vae_loss(self, x, x_decoded):
    val_x = x[:, :, 0]
    val_y = x[:, :, 1]
    pen = x[:, :, 2:]
    rec_loss = mdn_loss(val_x, val_y, pen, x_decoded)
    try:
        kl_loss = - 0.5 * K.mean(1 + self.z_log_sigma -
            K.square(self.z_mean) -
                                 K.exp(self.z_log_sigma), axis
                                     = -1)
        return rec_loss + 0.5 * kl_loss
    except AttributeError:
        print("no encoder")
        return rec_loss

def set_batches(self):
    batches = None
    val_batches = None
    for i in range(50):
        a, b, c = self.x_train.get_batch(i)
        if batches is None:
            batches = b
        else:
            batches = np.append(batches, b, axis = 0)

    for i in range(10):
        a, b, c = self.x_valid.get_batch(i)
        if val_batches is None:
            val_batches = b
        else:
            val_batches = np.append(val_batches, b, axis = 0)

    return batches, val_batches

def load_weights(self):
    try:
        self.vae.vae.load_weights("vae_model")
        print("Loaded weights from file")
    except IOError:
        print("Weights not found")
```

```
    def compile_fit(self):
        self.vae.vae.compile(loss = self.vae_loss, optimizer =
            'adam',
                                 metrics = ['accuracy'])

        self.load_weights()

        batches, val_batches = self.set_batches()

        print(batches.shape[:])
        print(val_batches.shape[:])

        self.vae.vae.fit(batches, batches, shuffle = False,
                         batch_size = self.batch_size, epochs =
                             self.epochs,
                         validation_data = (val_batches,
                             val_batches))
        self.vae.vae.save_weights("vae_model", True)
```

Listing 3.3: La classe che si occupa della compilazione della rete e del fitting dei dati

```
   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]
    # out_mu = K.reshape(out_mu, [-1, numComonents*2, outputDim])
    out_mu_x = K.permute_dimensions(out_mu_x, [0, 2, 1])
    out_mu_y = K.permute_dimensions(out_mu_y, [0, 2, 1])
    # use softmax to normalize pi and q into prob distribution
    max_pi = K.max(out_pi, axis = 1, keepdims=True)
    out_pi = out_pi - max_pi
    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.softmax(pen_logits)
    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, out_q


def tf_bi_normal(x, y, mu_x, mu_y, sigma_x, sigma_y, ro):
    norm1 = subtract([x, mu_x])
    norm2 = subtract([y, mu_y])
    norm1 = K.permute_dimensions(norm1, [0, 2, 1])
    norm2 = K.permute_dimensions(norm2, [0, 2, 1])
    sigma = multiply([sigma_x, sigma_y])
    z = (K.square(norm1 / (sigma_x + 1e-8)) + K.square(norm2 /
        (sigma_y + 1e-8)) - 2 *
         multiply([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 * multiply([sigma, K.square(ro_opp)]) + 1e-8
    result = result / denom + 1e-8
    return result
```

```
# TODO sort dimensions out
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 = multiply([result, out_pi])
    result = K.permute_dimensions(result, [0, 2, 1])
    result = K.sum(result, axis=1, keepdims=True)
    result = -K.log(result + 1e-8)
    fs = 1.0 - logits[:, 2]
    fs = Reshape((-1, 1))(fs)
    result = multiply([result, fs])
    result = K.permute_dimensions(result, [0, 2, 1])
    # L_r loss term, L_p part
    result1 = K.categorical_crossentropy(logits, out_q, from_logits
        = True)
    result1 = Reshape((-1, 1))(result1)
    result = result + result1
    return K.mean(result)
```

Listing 3.4: I metodi per il calcolo dell'errore di ricostruzione

4

CONCLUSIONI

BIBLIOGRAFIA

[1] David Ha, Douglas Eck - *A Neural Representation of Sketch Drawings* - arXiv:1704.03477, 2017. (Cited on page 7.)

[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 page 7.)

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

[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 page 7.)

[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.

[24] C. Doersch - *Tutorial on Variational Autoencoders* - arXiv preprint arXiv:1606.05908v2, 2016.

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