UNIVERSITÀ DEGLI STUDI DI PERUGIA
Dipartimento di Matematica e Informatica

CORSO DI LAUREA MAGISTRALE IN INFORMATICA

Tesi di Laurea Magistrale

Reti Convolutive e a Capsule per riconoscimento di post incitanti all'odio

Laureando
Lorenzo Ferri

Relatori
Valentina Poggioni
Alina-Elena Baia

Anno Accademico 2018/2019

Indice

Introduzione

Il termine *Social Network* indica un servizio informatico on-line il cui scopo è quello di facilitare la gestione dei rapporti sociali tra persone consentendo la comunicazione e la condivisione di contenuti digitali mediante diverse forme multimediali come frasi scritte, brani musicali, immagini o video. Sono delle vere e proprie realtà telematiche, dove persone di diversa provenienza, cultura, religione, sesso, età, opinioni politiche condividono tra loro pensieri, preferenze, gusti. Proprio come accadeva prima della nascita dei social, quando le persone si intravano al bar o in paese per condividere del tempo insieme. Ma mentre in questo caso stiamo parlando di una comunicazione che avviene tra persone fisicamente presenti, con una vera identità e volto, nel caso dei social invece ci ritroviamo a parlare con un nickname e un'immagine di profilo davanti ad uno schermo. Una differenza che abbatte qualsiasi possibile legame empatico e di responsabilità tra i due interlocutori. Con *deresponsabilizzazione* si intende infatti un comportamento che porta a evitare l'assunzione di responsabilità e a tutelare solo il proprio interesse come se fosse un diritto, senza tener conto di un bene collettivo, in una società dove ognuno rivendica solo i propri diritti dimenticando i doveri che il vivere sociale prevede, in una sorta di totale irresponsabilità. É il caso dei cosiddetti *leoni da tastiera*, termine coniato per riferirsi a utenti del web che scrivono in modo aggressivo, insultando, offendendo, screditando o minacciando altri utenti approfittando di questo anonimato che i social offrono. Secondo i dati ISTAT, il *cyberbullismo* (discriminazione e violenza che avviene per vie telematiche) ha colpito il $22,2\%$ delle vittime totali di bullismo e, con l'internet aperto a tutti dalle infinite soluzioni informatiche utilizzabili da tutte le fasce di età, è un dato costantemente in crescita. Questa minaccia ha costretto a correre ai ripari, cercando una soluzione per evitare il verificarsi di situazioni del genere, come ad esempio riconoscere se un determinato dato è offensivo o meno, eliminandolo. In ogni istante milioni e milioni di dati (immagini, testo, video) vengono inviati on-line e questo numero eccessivo rende impossibile delegare tale compito a un gruppo di persone che riconoscano ed eliminino contenuti offensivi.

La comunità scientifica invece ha pensato bene di dover automatizzare il riconoscimento di strani contenuti web, assegnando questa responsabilità ad un calcolatore che possa in poco tempo analizzare un enorme quantità di dati, discriminando quelli offensivi da quelli non. Questo proposito ha trovato applicazione nel *Machine Learning*, ossia lo studio di algoritmi e modelli satistici che un computer usa per risolvere un determinato problema senza usare istruzioni esplicite, basandosi invece su schemi e condizioni che il calcolatore stesso riesce ad imparare dai dati di esempio forniti per allenarlo nella comprensione. Il **NLP**, in particolare, è quella parte del Machine Learning che si interessa dello studio e analisi di dati in forma testuale (o vocale) tramite intelligenza artificiale, applicato a diversi problemi tra i quali appunto il riconoscimento di contenuti offensivi. Visto il grande interesse da parte della comunità scientifica e l'effettiva utilità che porterebbe in presenza di problemi dovuti ai social network finora trattati, il NLP vanta di numerose applicazioni per molteplici task, suscitando la curiosità anche di programmatori che giorno dopo giorno offrono nuove soluzioni e strumenti per ottenere risultati ancora migliori di quelli che già riusciamo ad ottenere. L'unico problema è che la maggior parte della documentazione, strumenti, programmi sono programmati per lavorare solo su testo in lingua inglese, risultando inapplicabili su altre lingue. La comunità scientifica quindi preferisce continuare con l'aggiornamento e sviluppo di NLP applicato alla lingua inglese, lasciando le altre sprovviste di toolkit, documentazioni ed esempi che possano permettere di ottenere gli stessi risultati.

EVALITA[1] è una campagna che, sottomettendo periodicamente competizioni in ambito *NLP* su lingua italiana, vuole promuovere lo sviluppo informatico del *Machine Learning* su testo italiano, offrendo così soluzioni alternative ai molteplici modelli progettati esclusivamente per la lingua inglese. La competizione a cui ho preso parte insieme a due colleghi universitari (*Bianchini Giulio, Giorni Tommaso*) sotto il nome di *VulpeculaTeam* è **HaSpeeDee** [2] (abbreviazione di Hate Speech Detection) organizzato da **Evalita 2018** e che ha portato poi alla pubblicazione dell'articolo [1]. Nel progetto relativo alla competizione abbiamo creato un modello *MLP* di rete neurale in grado di riconoscere una frase contenente odio o meno, dopo aver effettuato una delicata e complicata fase di *preprocessing* con successiva estrazione di features. Partendo da tale progetto, sarebbe stato interessante confrontare i risultati ottenuti con quelli invece di reti neurali più sofisticate utilizzando lo stesso dataset per il medesimo task. Lo scopo di questa tesi

[1]Evalita: `http://www.evalita.it/`

[2]HaSpeeDe: `http://www.di.unito.it/~tutreeb/haspeede-evalita18/index.html`

quindi è la realizzazione di un modello che sia in grado di riconoscere se una frase contiene odio oppure no con un'accuratezza soddisfacente, utilizzando *modelli avanzati di reti neurali* come **Reti Neurali Convolutive (CNN)** e **Reti Neurali a Capsule (Capsule Network)** e relativo confronto con i risultati ottenuti nel progetto *HaSpeeDee* di EVALITA.

In questa tesi inizieremo con una piccola sintesi al progetto EVALITA con descrizione del dataset utilizzato seguito dalle varie fasi di preparazione dei dati e risultati ottenuti (Capitolo 1). Seguirà un capitolo di introduzione ai *modelli avanzati* (CNN e Capsule Network) utilizzati in questa tesi (Capitolo 3). Successivamente sarà presentata l'effettiva applicazione delle due reti, ovvero il lavoro concreto effettuato in questo percorso di laurea (Capitolo 4). Per finire, descrizione dei risultati ottenuti e confronto tra i tre modelli applicati all'Hate Speech Detection con il medesimo dataset(Capitolo 5).

Capitolo 1

Hate Speech Detection

Il **Natural Language Processing** (NLP) è una branca della scienza che tratta le tecniche computazionali utilizzate per l'analisi, comprensione e rappresentazione di testi a uno o più livelli di analisi linguistica con lo scopo di ottenere un'elaborazione del linguaggio che possa al meglio rappresentare quello utilizzato all'uomo. I dati elaborati possono essere di qualsiasi natura, lingua, modalità, genere, orali o scritti. L'unico requisito è che siano testi rappresentati in una lingua usata dagli umani per comunicare tra loro [3].

Il NLP ha trovato molte applicazioni nella scienza in generale, venendo applicato in diversi campi come la *linguistica*, che lavora sui modelli formali e strutturali del linguaggio, nella *psicologia cognitiva*, analizzando il linguaggio per studiare i processi cognitivi umani, e la *computer science*, per poter rappresentare i dati testuali in modo che siano computabili da un calcolatore. É in quest'ultimo ambito che il NLP ha trovato maggior interesse nella comunità scientifica. Molteplici sono le applicazioni sviluppate per l'analisi di testo tramite un elaboratore: sintesi e acquisizione di documenti, libri, articoli di eccessiva grandezza; sviluppo di *assistenti personali intelligenti* in grado di interloquire con gli utenti (Alexa, Siri, Google Assistant); *Named-entity recognition* per il riconoscimento e associazione di tag alle varie parole di un testo; *Sentimental Analysis* per il riconoscimento di stati d'animo (felicità, tristezza, odio) che caratterizzano un testo; *Text Classification* per la classificazione di testi in base alla categoria che trattano (sport, medicina, cucina).

La comprensione del linguaggio naturale comunque è considerata ancora oggi un problema *IA-completo* poiché il riconoscimento dei testi richiede una conoscenza estesa del mondo e una grande capacità di manipolarlo, qualità sviluppata dagli uomini in tutta la loro storia e per questo difficile se non impossibile da assegnare ad un elaboratore.

L'**Hate Speech Detection** è una delle tante applicazioni del NLP in tema di *Sentimental Analysis* e riguarda tutte quelle tecniche di analisi ed elaborazione del testo usate per rendere automatico il riconoscimento di testo offensivo, utilizzando l'intelligenza artificiale. Si definisce *testo offensivo* qualsiasi comunicazione (vocale o testuale) volta a denigrare, danneggiare, sminuire una persona o un gruppo di persone in base alla razza, colore, etnia, genere, orientamento sessuale, nazionalità, religione e pensiero, con la chiara intenzione di colpire la vittima nelle proprie caratteristiche intime e individuali.

L'analisi testuale effettuata da una *rete neurale* avviene su diversi livelli di linguaggio che vanno dalla sintassi della frase (distinzione tra nome, verbo, aggettivo) fino alla comprensione semantica della stessa, classificandola in base alla richiesta (ironia, sarcasmo, sentimento, sport, geografia, medicina...).

1.1 Stato dell'Arte

L'*Hate Speech Detection (HSP)* ha trovato vasta applicazione nel *Machine Learning*, suscitando la curiosità di gran parte della comunità scientifica nel trovare soluzioni sempre più efficienti ad un problema attuale e in costante crescita. Come spiegato nell'articolo [7], le tecniche più utilizzate finora per l'HSP si possono dividere in due categorie:

- **Metodi classici** che richiedono una fase di estrazione manuale delle features dai dati iniziali per poi utilizzare questi vettori di features con algoritmi come *Support Vector Machine (SVM), Naive Bayes, Multilayer Perceptrons (MLP)* e *Logistic Regression* per la classificazione. Esistono diversi tipi di features usate in campo HSP: *Simple surface features* come la posizione delle parole o *n-grammi* della frase; features calcolate da *Sentiment Analysis* come per esempio la polarità di un'espressione (se negativa, neutra o positiva); feature che *Lexical resources* per individuare parole negative come insulti, parolacce; *Linguistic features* che si basano su informazioni sintattiche come Part of Speech (PoS) e relazioni tra le altre features; *Meta-information* come frequenza di brutte parole, lunghezza della frase, parole in CAPS-LOCK;

- **Metodi basati sul deep learning** che utilizzano le reti neurali per calcolare le features automaticamente dai dati in input attraverso dei livelli di analisi sequenziali. Le reti più utilizzate per questo scopo sono *Convolutional Neural Network(CNN), Recurrent Neural Network*

(RNN) e *Short-Term Memory network (LSTM)*. Questi modelli di deep neural network devono essere seguiti da classificatori classici (come SVM, MLP) per poter classificare il record in base alle features calcolate dalla deep network stessa.

1.1.1 Dataset

Esistono diversi dataset e collezioni di documenti in lingua inglese usati per HSD, usati dalla maggior parte dei ricercatori del settore, fra cui anche nell'articolo [7] : **DT** contiene 24 783 commenti di odio e non senza un determinato argomento ma collezionati secondo un vocabolario di brutte parole; **RM** è composto da 2 435 tweets di odio e non riguardo rifugiati e comunità musulmana; **WZ** contiene 18 593 record che trattano di razzismo, sessismo e non odio; **WZ-S.amt** 6 579 commenti raccolti da persone amatoriali mentre **WZ-S.exp** 6 559 commenti raccolti da esperti; **WZ-S.gb** raccoglie insieme *WZ-S.amt* e *WZ-S.exp* selezionati sia dagli amatoriali che esperti; **WZ.pj** combina insieme *WZ* e *WZ-S.exp*.

Mentre per la lingua inglese troviamo una grande quantità sia di tool, applicazioni, esempi che di dataset molto grandi per poter ottenere dei modelli efficienti, in lingua italiana invece queste risorse sono decisamente esigue, limitando futuri sviluppi di soluzioni per il testo italiano. Questo è un problema perchè non fa altro che circoscrivere il continuo aggiornamento del Machine Learning solamente sul versante inglese, non permettendo ad altre lingue di ottenere la stessa attenzione.

Capitolo 2

Hate Speech Detection con Reti Neurali MLP

Il modello di rete che abbiamo deciso di utilizzare. in base alle conoscenze di allora, è un *Multilayer Perceptron*. L'architettura e i parametri di questa rete sono stati scelti testando la stessa su un set fisso di costanti per i vari parametri (numero di nodi livello, funzione di attivazione utilizzata, ottimizzatore), effettuando un numero uguale di run con relativa media delle accuratezze. L'architettura che ci ha permesso di ottenere migliori risultati è presentata nella Sezione 2.6. L'intero sistema è organizzato in quattro parti, iniziando da una fase di pulizia dei dati (preprocessing - Sezione 2.3), successiva codifica e preparazione dei dati per la rete (encoding - Sezione 2.5), allenamento della rete per decidere i pesi W_{ij} ottimali (training) e infine verifica del modello (testing). L'intero ciclo di vita è rappresentabile con il diagramma di flusso in figura 2.1:

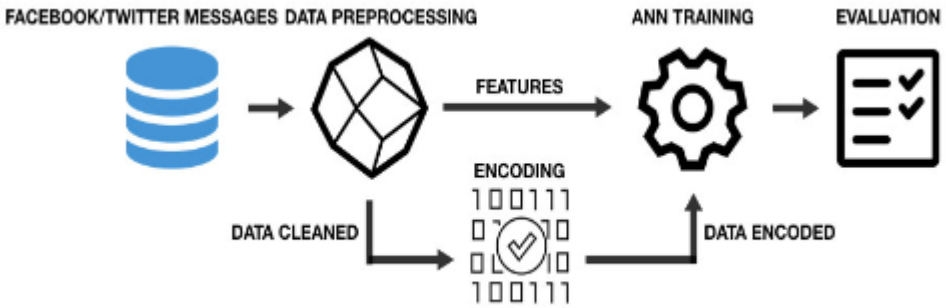


Figura 2.1: Ciclo di vita del nostro software utilizzato per HaSpeeDee di EVALITA 2018

2.1 Modello di rete Multilayer Perceptron

Un multilayer perceptron è un modello di rete neurale *feedfoward* (le cui connessioni tra i vari nodi non formano cicli), la cui unità è rappresentata dal **neurone**, chiamato *nodo*. I nodi sono organizzati in più *livelli* divisibili in tre tipi:

1. **Input Layer**: è il livello che riceve i record del dataset. Il numero di nodi presenti è pari alla dimensionalità dell'input (intesa come numero di features) che vogliamo inviare alla rete;

2. **Hidden Layers**: sono i livelli intermedi tra input e output dove avviene l'allenamento della rete. Sono chiamati "hidden" (nascosti) perchè i nodi si passano dati che solo la rete conosce e sa interpretare;

3. **Output Layer**: l'ultimo livello della rete che si occupa della classificazione del record. Contiene tanti nodi quante sono le categorie che il task richiede. In questo livello è molto importante la scelta della funzione di attivazione da utilizzare, in modo da valutare in modo corretto le varie probabilità distribuite nelle classi e scegliere quella migliore.

I dati dal livello di input vengono passati al primo hidden layer, per poi essere manipolati da tutti gli altri hidden fino al livello di output, dove avviene l'assegnamento della classe al record.
Essendo un modello *fully connected*, ogni nodo è connesso a tutti i nodi del livello successivo ed è caratterizzato da una *funzione di attivazione* non lineare.

Una funzione di attivazione è una funzione non lineare che riceve in input la somma pesata dei neuroni del livello precedente. Questo output totale può variare da $-\infty$ e $+\infty$ quindi è necessario limitarlo ad un intervallo chiuso (alcune funzioni utilizzano l'intervallo $[0; 1]$ altre $[-1; 1]$) e fissare una soglia in base alla quale il neurone viene attivato oppure no. Le funzioni di attivazione garantiscono anche la non linearità all'output di un neurone, migliorando l'apprendimento della rete nel creare pattern meno specifici ma più generali. La funzione di attivazione più utilizzata è la **ReLU**, spiegata nella sezione 3.1.2.

Ogni singola connessione è caratterizzata da un certo peso w_{ij} che verrà via via aggiornato durante l'allenamento in relazione ai risultati ottenuti.

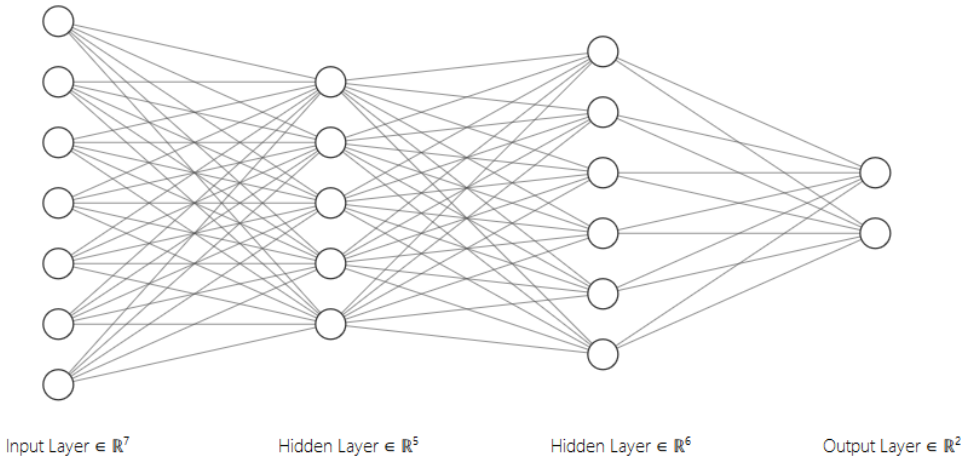


Tabella 2.1: Topologia di un Multilayer Perceptron

Ogni nodo inivia il proprio output moltiplicato per il peso della relativa connessione al nodo del layer successivo. Quest'ultimo nodo somma tra loro tutti gli output dei nodi precedenti e li utilizza come input della funzione di attivazione del layer a cui appartiene. L'input totale che arriva ad un nodo è espresso secondo l'equazione 2.1:

$$Y_j = k(\sum_i w_{ij} X_i) \tag{2.1}$$

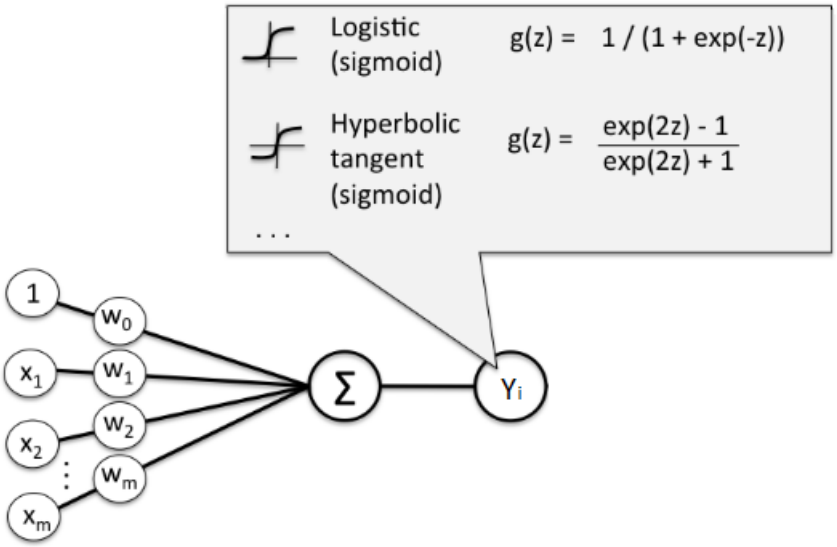


Figura 2.2: Struttura base di un nodo

2.1.1 Algoritmo di Backpropagation

La **Backpropagation** è un algoritmo di apprendimento utilizzato per aggiornare i pesi w_{ij}, inizialmente impostati a valori random e successivamente aggiornati durante l'allenamento della rete, permettendo la creazione di un modello generale in base al training set utilizzato. Trattandosi di un apprendimento supervisionato (cioè che per il training vengono utilizzati record già etichettati), ad ogni step dell'allenamento viene confrontato l'output calcolato dalla rete con il valore reale dello stesso record. Questo confronto avviene secondo una determinata **funzione di errore** scelta dal programmatore. Di funzioni ce ne sono diverse e ognuna si applica meglio ad una certa categoria di problemi (es. problemi binari) rispetto che ad altre.

Una volta valutata la funzione in base all'etichetta calcolata e quella reale, rappresentante la distanza che c'è tra i due risultati e quindi quanto ancora la rete deve migliorare, la backpropagation fa utilizzo dell'algoritmo di **Discesa del Gradiente** applicato alla stessa funzione di loss per cercarne il minimo. Ad ogni step ci si muove nella funzione di un certo passo (chiamato *learning rate*). Viene calcolato il gradiente in quel punto tramite la derivata parziale (indicante la pendenza della curva), per decidere se cercare il minimo o a sinistra o a destra del punto dove ci eravamo fermati nello step prima. Ogni passo porta con se un aggiornamento dei pesi w_{ij}, aggiornamento comunque

pesato in base all'output della funzione di loss in modo da cercare ad ogni step di ottenere una rete migliore, che minimizza sempre più l'errore.
La convergenza di questo algoritmo può essere molto lenta e non è garantita: il calcolo del gradiente potrebbe fermarsi su di un falso minimo che non permetterà di ottimizzare la rete. Spesso si è soliti utilizzare un learning rate adattivo in modo che l'algoritmo regoli ad ogni epoca lo spostamento da effettuare per la ricerca del minimo, evitando salti sproporzionati dove la convessità della funzione è eccessiva.

Algorithm 1 Back-propagation

```
1: Inizializza tutti i pesi W_ij con valori casuali;
2: repeat
3:     for each record t del training-set do
4:         Dai in input t alla rete e calcola l'output relativo;
5:         Calcola l'errore della rete secondo la loss-function scelta;
6:         for each neurone k di output do
7:             δ_k = y_k(1 − y_k)(C_k − y_k)
8:         end for
9:         for each neurone h degli hidden-layers do
10:            δ_h = y_h(1 − y_h) Σ_k w_hk δ_k
11:        end for
12:        for each peso w della rete do
13:            w_i,j = w_i,j + η δ_j x_i,j
14:        end for
15:    end for
16: until non è soddisfatta la condizione di terminazione stabilita
```

La condizione di terminazione può essere di due tipi: o dopo un numero fissato di epoche di allenamento della rete; oppure dopo aver raggiunto una certa soglia di errore nella funzione di loss, ma in questo caso non è garantita la convergenza.

2.2 Dataset

Il dataset utilizzato è stato offerto da **EVALITA** per la competizione *HaSpeeDe* ed è diviso in due parti, entrambe contenenti 4000 record l'una (3000 per il *training set* e 1000 per il *test set*):

- La prima è una raccolta di commenti Facebook creata nel 2016 da un gruppo di Pisa, lavoro presentato nell'articolo [2];

- La seconda invece è un corpus di commenti Twitter creato nel 2018 da un gruppo di Torino, contenuto negli articoli [4] e [6].

Ogni frase dei due set è inoltre dotata di una label (**1** o **0**) che indica rispettivamente se vi sono contenuti d'odio o meno. In questa tesi ho voluto creare un modello allenato solo sul dataset di Facebook utilizzando solo i primi 4000 record.

I record sono commenti e post grezzi raccolti da pagine Facebook e Twitter a tema politico. Dato che il dataset proviene direttamente dai social, i singoli record sono frasi che riguardano un'utenza di fascia medio-bassa, che utilizza quindi un linguaggio povero e volentieri sbagliato, ricco di imperfezioni. Questo ha reso necessaria un'approfondita fase di **Preprocessing** del dataset per eliminare tutto ciò che è superfluo alla semantica del testo e nel renderlo più comprensibile da una rete neurale. In figura 2.3 vengono riportati tre esempi di record con le relative label (**1** se c'è odio e **0** se non) per rendere l'idea del tipo di dati in questione forniti da *EVALITA*.

Io voterò no😉😉😉Renzi deve andare a casa☹☹☹	0
.....AZZI.....LORO !!!!!!!!!!!!!!!!!!!!!!!!!!!	1
Malpezzi ma vaiiiiiiiii finiscila x piacere sei ridicolaaaaaaaaaa	1

Figura 2.3: Tre esempi di record del dataset EVALITA

2.3 Preprocessing

I dataset forniti da EVALITA comprendono record raccolti direttamente da Facebook e Twitter, senza che sia stata applicata alcuna modifica al dato originale. I social non richiedono l'utilizzo di un linguaggio pulito e formale, e per questo i record presentano svariati errori, imperfezioni, ambiguità alle quali dovremo ricorrere eliminandole o trovando una soluzione che permetta di mantenere la semantica della frase rendendo comunque questa più comprensibile da un computer. Aggiungiamo pure la presenza di vocaboli inutili (congiunzioni, preposizioni, ...) e le numerose coniugazioni di uno stesso verbo ed è chiaro come la fase di preprocessing sia un'operazione assolutamente necessaria, delicata e in alcuni aspetti complessa dalla quale dipende il risultato finale di una rete.

Le operazioni di preprocessing effettuate nel progetto HaSpeeDe sono state:

- Eliminazione delle stopwords (congiunzioni, preposizioni, articoli...);

- Sostituzione di caratteri speciali (@ → a, & → e, ú → u);
- Eliminazione della punteggiatura;
- Sostituzione delle emoticons con la relativa descrizione in inglese (☺ → ”smilingface”);
- Separazione degli hashtag (#vivaItaliaLibera → ”viva Italia Libera”);
- Riconoscimento di parolacce occultate con caratteri e simboli all'interno della parola (”s7uP.....1d0” → ”stupido”);
- Riconoscimento di errori ortografici (inteso come parole non contenute in un vasto vocabolario) e sostituzione con la parola (contenuta sempre nello stesso vocabolario) che presenta la più alta percentuale di similarità;
- Lemmatizzazione delle singole parole per eliminare eventuali alterazioni (diminutivo, maggiorativo, vezzeggiativo, dispregiativo) e sostuire le molteplici coniugazioni di uno stesso verbo con la relativa forma al tempo infinito (”gioco” → ”giocare” , ”giocando” → ”giocare”).

2.4 Estrazione features

Per poter allenare un *Multilayer Perceptron* è necessario dover calcolare delle features relative al problema da affrontare. In tutto abbiamo calcolato ben nove features per ogni record:

1. Percentuale di parolacce nella singola frase;
2. Numero di parolacce nella singola frase;
3. Polarità di un commento utilizzando il tool *SentiWordNet*, assegnando un valore da -1 a 1 (-1 → commento negativo, 0 → commento neutro, 1 → commento positivo);
4. Soggettività della frase utilizzando il tool *TextBlob*;
5. Numero di proposizioni all'interno di un record
6. Numero di '?' e '!';
7. Numero di '.' e ',' ;
8. Percentuale di parole scritte in *CAPS-LOCK*;
9. Lunghezza della frase (intesa come numero di parole).

2.5 Word2Vec

Dopo il preprocessing è richiesta una fase di **Embedding**, ossia di traduzione delle parole in numeri reali in modo che siano più riconoscibili e calcolabili da un computer. Questa è la parte relativa all'encoding dei dati puliti.

Il *word embedding* permette di memorizzare le informazioni sia semantiche che sintattiche delle parole, costruendo uno spazio vettoriale in cui i vettori delle singole parole sono più vicini se queste sono riconosciute come semanticamente più simili.

Si può effettuare in diversi modi ma il modello che ha dato migliori risultati e che utilizziamo è il modello **Word2Vec**. Si tratta di una rete neurale a 2 livelli allenabile con un corpus testuale (preferibilmente non lo stesso dataset del task ma un corpus molto più grande). Dopo aver specificato la dimensionalità dell'output che corrisponde a quanto deve essere grande il vettore di reali rappresentante la singola parola, viene creato lo spazio vettoriale di tutte le parole contenute nel corpus di input. Il risultato di questa fase iniziale è un file dove è riassunto lo spazio vettoriale, espresso come dizionario *parola* $\rightarrow$ *vettore*.

Una volta costruito il modello e generato lo spazio vettoriale, non ci resta che tradurre le parole del dataset nel relativo vettore di reali corrispondente nel dizionario.

"Matteo"	$\rightarrow$	[	2.3	0.5	-1.0]
"sei"	$\rightarrow$	[	0.1	-1.6	0.0]
"grande"	$\rightarrow$	[	1.7	-4.0	1.0]

Se ogni parola ha una sua rappresentazione vettoriale, allora una frase può essere vista come un insieme di vettori, ognuno corrispondente alle parole contenute nella frase stessa.

"Matteo sei grande"

$$\downarrow$$

$$[\ [\ 2.3\ \ 0.5\ \ -1.0\ ],\ [\ 0.1\ \ -1.6\ \ 0.0\ ],\ [\ 1.7\ \ -4.0\ \ 1.0\ ]\ ]$$

Nel progetto HaSpeeDe è stata utilizzata come ulteriore feature sia la somma che la media dei vettori delle parole di una singola frase, in modo da

poter rappresentare con un vettore di **n** valori non più ogni singola stringa bensì l'intero record. In questo modo riusciamo a dare un peso alla frase totale e non più alle singole parole, rendendo possibile il confrontro tra interi record. La potenza semantica e sintattica di una frase viene riassunta in un vettore di **n** valori reali.

MEDIA "Matteo sei grande" → [1.4 -1.7 0.0]

SOMMA "Matteo sei grande" → [4.1 -5.1 0.0]

Il nostro modello *Word2Vec* è stato allenato usando **220.000** commenti scaricati da varie pagine Facebook a tema politico con l'aggiunta dei record offerti da EVALITA. La dimensione scelta per ogni vettore è **128**.

2.6 Architettura rete

Il tipo di rete utilizzato è un *Multilayer Perceptron* formato dai seguenti livelli:

Name	**# Nodes**	**Function**	**Dropout**
Input Layer	265	relu	0.45
1° Hidden Layer	128	sigmoid	0.45
2° Hidden Layer	56	relu	0.45
Output Layer	2	softmax	-

L'input layer, avendo il compito di ricevere i singoli record in input e inviarli agli hidden layer, dovrà avere un numero di nodi pari alle features scelte. Le features calcolate nella sezione 2.4 sono **9** mentre quelle relative al *word embedding* sono 2 vettori da **128** valori reali (corrispondenti alla *somma* e *media* dei vettori delle parole). Per questo il numero di nodi input è $9 + 128 + 128 =$ **265**.

L'ottimizzatore utilizzato è **Adagrad** mentre per quanto riguarda la funzione di loss, trovandosi davanti ad un problema binario con output **0** o **1** quella più idonea al task è la **binary cross-entropy**.
L'intera struttura è riassumibile nel seguente schema:

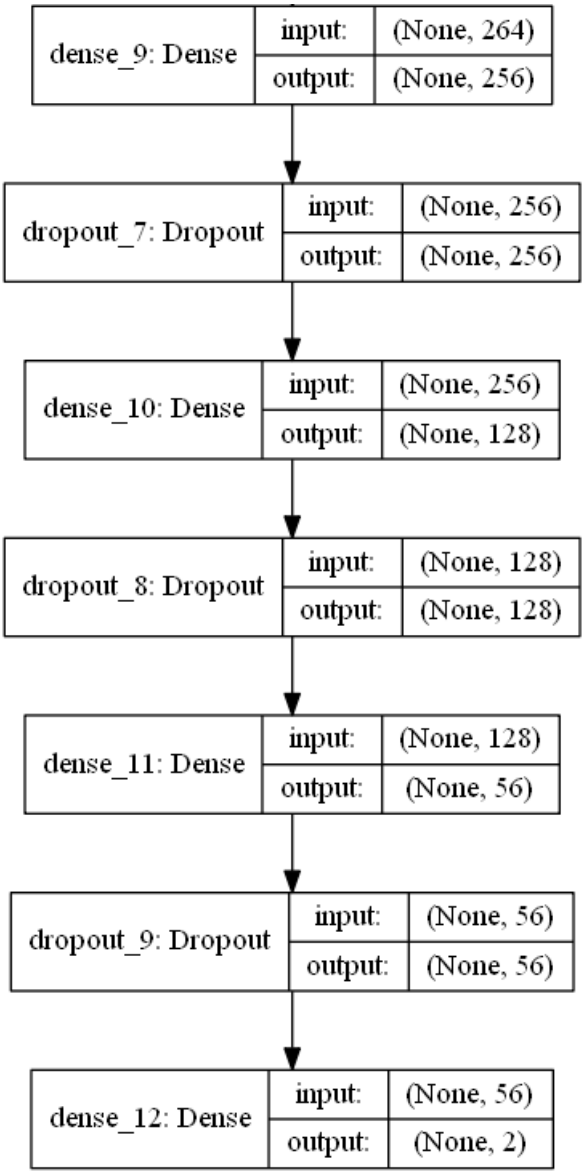


Figura 2.4: Architettura del Multilayer Perceptron usato per la competizione *HaSpeeDe*

Capitolo 3

Modelli avanzati di reti neurali

I risultati riportati nel capitolo 5.1 dimostrano come un *MultiLayer Perceptron*, sebbene sia un modello di rete neurale semplice, riesca ad ottenere una buona percentuale di accuratezza anche se applicato a piccoli dataset. Decisiva sicuramente è stata la fase di *estrazione features*, senza la quale una rete come questa non avrebbe dati a sufficienza per riuscire a costruire un pattern adeguato che generalizzi efficientemente il dataset. É la fase più delicata e minuziosa, questo perchè le features da calcolare devono essere relative al problema posto (es. numero di parolacce per calcolare quanto odio contiene una frase) e soprattutto richiedono di essere calcolate ed estrapolate manualmente dal programmatore stesso. Questo porta con se altri due problemi: se tutte le features calcolate sono utili, cioè permettono alla rete di generalizzare meglio il problema, e se le ha calcolate nel miglior modo e con la giusta unità di misura. Il programmatore dovrà chiedersi se è meglio utilizzare come feature il numero di parolacce all'interno di una frase (rappresentato quindi da un intero) oppure la percentuale di parolacce (un numero reale), con realtivi problemi di normalizzazione e dati su diverse scale di misura. Il programmatore si troverà quindi a dover testare molte volte la stessa rete, variando il set di features ad ogni test per capire qual è l'insieme migliore con cui la rete riesce ad apprendere meglio.

Se da una parte il Multilayer Perceptron è una rete dal semplice utilizzo e costruzione ma che presenta difficoltà nel ricavare autonomamente i dati lasciando al programmatore questo delicato compito, dall'altra invece esistono altri *modelli avanzati* di reti neurali, più complessi del primo, che fanno dell'estrazione di features una loro grande qualità.

In questa tesi ho voluto confrontare tra loro queste tipologie di reti, cercando di evidenziarne sia i pro che i contro in modo da comprendere il perchè

preferire l'una rispetto che l'altra davanti ad un determinato task.

I modelli avanzati di rete che questa tesi tratterà sono le **Reti Neurali Convolutive (CNN)** e **le Reti Neurali a Capsule (CapsNet)**.

3.1 Reti Neurali Convolutive (CNN)

Le CNN sono un modello di rete neurale dove in almeno uno dei livelli viene applicata l'operazione di convoluzione tra matrici. Un livello convolutivo consiste nell'applicazione di determinati filtri (tramite appunto l'operazione di convoluzione) per poter estrarre features e modelli molto più elaborati e instrinsechi ai dati input che invece, nelle precedenti reti neurali, il programmatore avrebbe dovuto calcolare a mano. L'effettiva innovazione portata dalle CNN è l'introduzione di **informazioni spaziali** tra le features di uno stesso dato. Prendiamo come esempio un record contenente i dati x, y, z. Per una MLP, qualsiasi sia l'ordine con cui questi si presentano (xyz, xzy, yzx, ...) non inciderà nell'output della rete, ottenendo sempre lo stesso risultato. Nelle reti convolutive invece vengono calcolate anche le informazioni spaziali dei dati vicini tra loro, ottenendo features diverse in base all'ordine in cui i tre dati si presentano. É proprio l'utilizzo dei filtri nei livelli convolutivi che permette di dare ai dati questa importanza spaziale, analizzando non solo un dato singolarmente ma l'insieme di più informazioni ricoperte dal filtro stesso.

Questa qualità ha permesso alle CNN di ottenere per molti anni il primato nell'ambito dell'image-processing, dove i dati rappresentati da matrici rendono necessaria l'analisi spaziale tra loro e le relazioni tra pixel vicini localmente.

L'image-processing vede così le CNN applicate a molti problemi, ottenendo sempre ottimi risultati rispetto a quelli ottenuti con altri modelli di rete: riconoscimento facciale e di oggetti, data mining, auto intelligenti, diagnosi di malattie, analisi/statistica e molto altro.

Richiedono una fase di preprocessing minore e meno accurata perchè la rete apprende i filtri che nella MLP invece erano progettati a mano. Questa indipendenza dalle precedenti conoscenze e dallo sforzo umano nella progettazione delle caratteristiche è un grande vantaggio.

3.1.1 Operazione di Convoluzione

Il nome "*convolutiva*" indica che la CNN impiega la **convoluzione**, un'operazione matematica lineare tra due funzioni di una variabile che consiste nell'integrare il prodotto tra la prima e la seconda traslata di un certo valore. In pratica con la convoluzione analizziamo il cambiamento della prima funzione al variare della seconda.

$$(f * g)(x) = \int_{\infty}^{\infty} f(y)g(x-y)dy \tag{3.1}$$

La convoluzione viene utilizzata molto nell'*image processing*, in particolare nell'applicazione di filtri. Un filtro è una matrice (solitamente di dimensione dispari 3x3, 5x5, ...) che si muove ad ogni step di una certa lunghezza (chiamata **stride**) per tutta l'immagine a cui è applicato, dal primo pixel in alto a sinistra fino all'ultimo in basso a destra.

Dopo che il filtro ha scansionato tutta l'immagine, il risultato ottenuto è una **feature map**. Quest'ultima, come dice il nome stesso, evidenzia una determinata feature nell'immagine, come può essere il rilevamento degli angoli, la presenza o meno di determinati colori e forme.

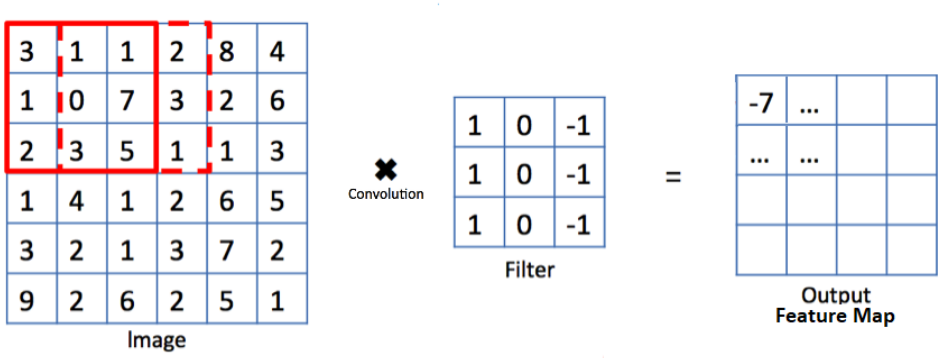


Figura 3.1: Operazione di convoluzione con stride=1, immagine input 6x6, filtro random 3x3

É importante notare dalla figura 3.1 come tale operazione non sia una classica moltiplicazione matriciale *righe*×*colonne* tra matrici, bensì una moltiplicazione *element-wise* come spiegato nell'esempio 3.2.

$$(f * g) = (3 \times 1) + (0 \times 1) + (1 \times -1) + (1 \times 1) + (0 \times 0) + (7 \times -1) + (2 \times 1) + (3 \times 0) + (5 \times -1) = -7 \tag{3.2}$$

La feature map di output è una matrice che conterrà tutte le informazioni necessarie a descrivere una determinata feature, calcolate direttamente dalla CNN.

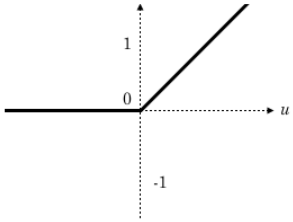
Le reti convolutive quindi sono reti neurali (come la MLP) che però utilizzano l'operazione di convoluzione tra matrici in almeno uno dei livelli della sua architettura. Il numero di filtri applicati in un determinato livello viene specificato dal programmatore e ad ogni filtro corrisponde una feature map in output. All'inizio tutte le matrici-filtro vengono inizializzate con valori random, ma che comunque seguono una determinata distribuzione di probabilità, aggiornati poi durante il training dalla Backpropagation in base ai risultati ottenuti confrontati con il test set secondo una determinata funzione di loss. La logica di inizializzazione e di aggiornamento è identica a quella utilizzata nella MLP, soltanto che mentre nell'ultima riguarda i singoli pesi delle connessioni w_{ij}, nelle CNN invece comprende intere matrici-filtro.

3.1.2 Design

La struttura base di una CNN si divide in tre parti:

- **Input Layer**: come nella MLP, è il livello che prende l'input della rete e lo invia ai livelli successivi. La forma dell'input è una matrice di dimensioni pari alle dimensioni delle immagini contenute nel dataset;
- **Convolutional Part**: è la parte della CNN dove vengono applicati i filtri e calcolate le features tramite convoluzione. É divisa in strati e ognuno è diviso a sua volta in tre componenti principali:
  - **Convolutional Layer**: è il livello che applica effettivamente la convoluzione;
  - **ReLU Layer**: la ReLU è una funzione non lineare molto utilizzata nelle reti neurali e soprattutto nelle CNN per assicurare la non linearità delle immagini filtrate. Le immagini originali che noi acquisiamo infatti sono irregolari ma, una volta nella rete, sono oggetto di operazioni lineari come la convoluzione (somma e moltiplicazione). Per mantenere questa proprietà di irregolarità, la ReLU rimpiazza tutti i valori negativi di un immagine con il valore **0**.

$$f(x) = max(0, x) \tag{3.3}$$



– **Pooling Layer**: Il *pooling* è un processo di discretizzazione applicato alle feature maps estratte, con lo scopo di ridurre la grandezza delle stesse ma conservando comunque i dati e le caratteristiche più importanti. La feature map viene divisa in celle (chiamate *window*) e per ognuna di queste si sceglie quale tra i diversi metodi di pooling utilizzare, tra cui quelli più comuni sono:
  * **Average Pooling**: viene calcolato il valore medio contenuto in ogni window;
  * **Max Pooling**: viene preso il valore più grande all'interno della window.

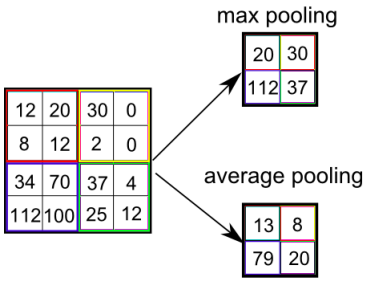
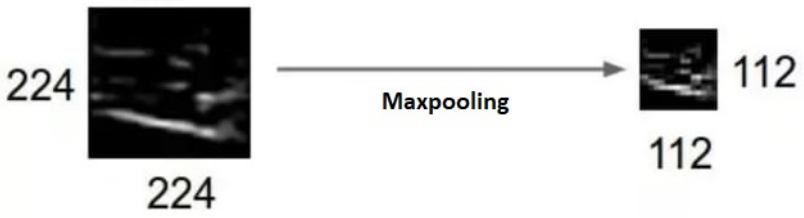
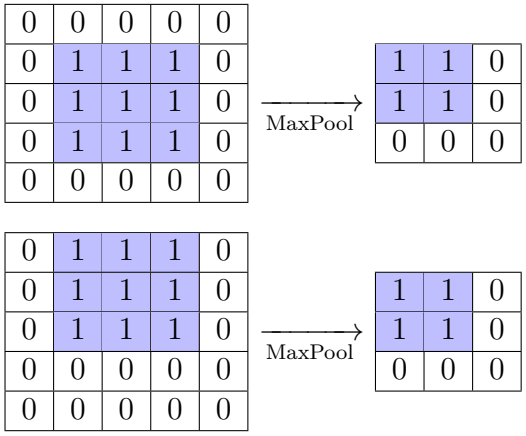


Figura 3.2: Applicazione di max e average pooling con window 2x2 ad una feature-map 4x4

Come risultato otteniamo una feature map decisamente più piccola e semplice, ma che contiene comunque le informazioni necessarie calcolate in quella input.

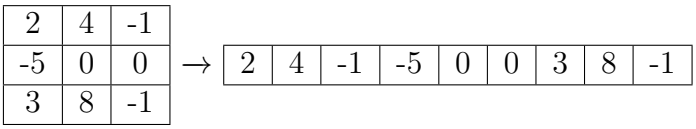


Il pooling layer quindi riduce il numero di parametri e, di conseguenza, di calcoli che la rete deve effettuare, migliorando l'efficienza della stessa e riducendo il pericolo di overfitting. Questa è una delle principali differenze tra le CNN e MLP.
Il processo di pooling garantisce la *translation invariance*, cioè riuscirà a riconoscere un oggetto in due immagini diverse anche se questo è stato spostato di qualche pixel.



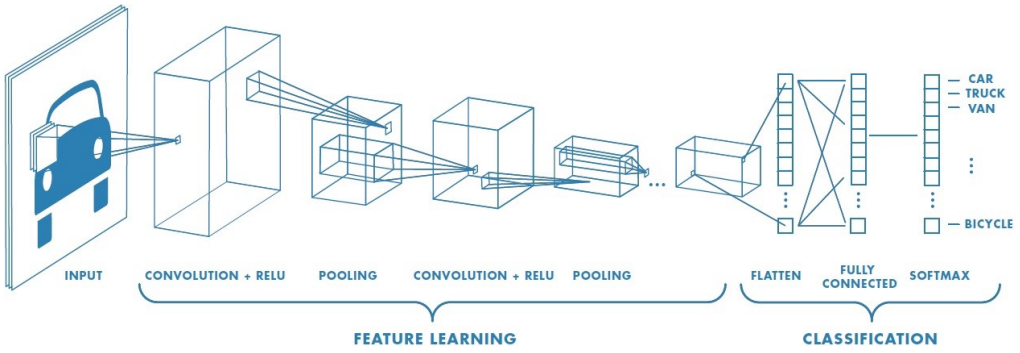
L'esempio sopra riporta i due output dopo il *max pooling* applicato a due immagini con un oggetto spostato in alto (rappesentato dai pixel di valore 1). Come è possibile notare, anche se le due immagini hanno un contenuto informativo differente, gli output sono uguali, permettendo quindi la translation invariance. Se non utilizzassimo il MaxPooling la CNN sarebbe utile a classificare solo immagini uguali a quelle contenute nel training set.

- **Fully Connected Network**: Si tratta di una MLP a tutti gli effetti, con un input layer, n hidden layers e un output layer con un numero di nodi pari alle categorie. L'input layer dovrà avere un numero di nodi uguale alla grandezza della matrice output dell'ultimo livello convolutivo della rete. In particolare a questo punto è necessario ridimensionare tale output, passando da una matrice (risultato dei livelli convolutivi) ad un vettore (input della MLP).



Più livelli convolutivi e filtri aggiungiamo, più è approfondita l'analisi delle immagini passate come input e il riconoscimento delle varie componenti.

La CCN ha appunto una complessità crescente in base a quanti *Convolutional Layers* aggiungiamo: il primo layer riuscirà solamente ad individuare angoli, vertici, linee; il secondo, grazie all'aiuto del primo, può riconoscere forme geometriche; il terzo invece anche parti intere dell'immagine (occhi, naso); fino ad arrivare all'ultimo che riuscirà a riconoscere anche interi oggetti nel loro complesso. Più la rete è topologicamente complessa, maggiore è il numero di parametri da allenare e quindi la complessità computazionale della rete stessa, ma dall'altro lato aumenta il grado di rappresentazione del modello.



3.1.3 MLP vs CNN

La principale differenza tra le CNN e MLP sta nell'estrazione delle features. Nella MLP è compito del programmatore calcolare le features manualmente, andando incontro a diversi problemi:

- Dati non adeguatamente normalizzati;
- Scelta delle unità di misura da utilizzare;
- Come dover rappresentare e calcolare i dati;
- Scelta del migliore set di features con cui la rete riesce ad apprendere meglio;

Questi pericoli potrebbero causare un' incompatibilità tra le features stesse, un enorme problema per il fatto che la rete non riesce a generalizzare (portando così all'overfitting) e perchè il programmatore potrebbe non accorgersi di un problema banale come può essere un'unità di misura non adeguata.

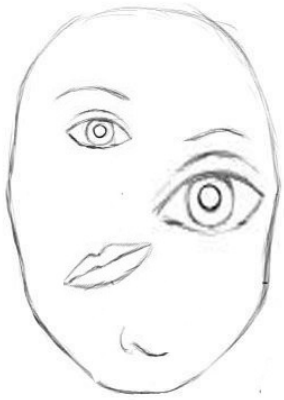
Nelle *CNN* invece è la rete stessa che calcola le features nei Convolutional Layers. Tramite l'applicazione di filtri la rete individua dati *intrinsechi* al

record in input invisibili ad un programmatore ma che sono indispensabili all'apprendimento della stessa. Inoltre non c'è bisogno di porsi eventuali problemi di normalizzazione o unità di misura. É possibile anche che sia il programmatore che la CNN abbiano calcolato le stesse medesime features, ma comunque la CNN le avrà calcolate e rappresentate in una forma più idonea e matematicamente coerente rispetto al primo. Questo ha un enorme vantaggio sia sulla convergenza dell'allenamento che sul problema dell'overfitting, generando modelli meno precisi ma più studiati, complessi e generici.

Inoltre le MLP non sono reti appropriate da utilizzare nella classificazione delle immagini: poichè le immagini hanno una rappresentazione a matrice fa di loro un tipo di dato inadeguato da utilizzare in una rete dove la struttura dati utilizzata nei calcoli è il vettore. Passare in input un'immagine trasformata in vettore richiederebbe un numero veramente eccessivo di nodi input e quindi di più connessioni e parametri, complicando esponenzialmente il problema. Informazioni riguardo la posizione dei pixel e la relazione con quelli vicini vengono perse, rendendo le MLP incapaci di lavorare con immagini o parti di queste traslate o rotate.

3.2 Capsule Networks

La *Rete a Capsule* è un modello di rete neurale introdotto per la prima volta nel 2017 da *Geoffrey Hinton* nell'articolo [5]. L'idea nasce con lo scopo di trovare una soluzione al **Picasso Problem** dovuto all'impossibilità nelle CNN di mantenere le relazioni spaziali tra le vari componenti dell'immagine. La causa è il **MaxPooling**: durante questa fase di discretizzazione, molte rilevanti informazioni spaziali vengono perse. Come è spiegato nella sezione 3.1.2 riferita ai *Pooling Layer*, una feature map viene ridotta di dimensionalità mantenendo comunque quelle caratteristiche che la rendono ancora riconoscibile, ma al tempo stesso comporta un'eccessiva perdita del proprio contenuto informativo.



L'immagine sopra contiene tutte le componenti di una faccia, ma effettivamente non è un volto reale. La CNN è molto efficiente nell'individuare le singole componenti di un oggetto ma non riesce a metterle in relazione spaziale tra loro. Quindi sa che una faccia è composta da due occhi, un naso e una bocca, ma non le singole relazioni tra loro come *"gli occhi devono essere alla stessa altezza", "il naso deve stare sotto gli occhi e sopra alla bocca", "la bocca si trova sotto il naso e nella parte bassa della faccia"*, tutte informazioni necessarie per discriminare un volto reale da un volto stile Picasso. Per questo una CNN classificherà l'immagine sopra come una *faccia*, mentre una CapsNet riesce a riconoscere l'incoerenza spaziale delle varie componenti non classificandola allo stesso modo.

Con l'introduzione delle CapsNet si vuole garantire il concetto di **equivarianza**: variare la posizione, la rotazione, la grandezza di una feature in un'immagine cambierà le sue proprietà geometriche (coordinate, angolo di rotazione) ma non la probabilità di fare parte di un insieme o meno.

3.2.1 Capsule

Come spiega Hinton, una capsula è un insieme di neuroni il cui output è un *activity vector* caratterizzato da due grandezze:

- *lunghezza*: rappresenta la probablità di un oggeto di esistere (valore compreso tra 0 e 1);
- *direzione*: codifica i parametri geometrici che descrivono la sua *pose* (posizione, rotazione, grandezza).

Un livello *i* può essere composto da 1 o più capsule e ognuna di queste è connessa a tutte le capsule del livello *i+1*. Ogni arco che collega la coppia è caratterizzato da:

- un *coefficiente di accoppiamento* c_{ij};
- una *matrice peso* W_{ij}.

Come nelle CNN, anche nelle CapsNet troviamo una struttura gerarchica dei livelli: capsule appartenenti a livelli inferiori operano su features di più basso livello rispetto a quelle superiori. Una capsula di livello i riconosce i particolari di un'immagine (come un naso, una bocca) mentre una capsula *i+1* riesce a identificare l'intera faccia nel suo complesso.

La logica di allenamento di questo nuovo routing si basa sulle **affinità** (*agreement*) tra capsule di livelli diversi, cioè di come i loro activity vector concordano nell'identificare un determinato oggetto. Una capsula del livello L si interessa di una determinata feature X, calcolandone l'activity vector e i relativi parametri. Quest'ultimo, moltiplicato per la matrice peso W_{ij} di ogni connessione, verrà trasformato in un *prediction vector* (di cui parleremo dopo) con lo scopo di prevedere quale delle features calcolate nel livello L+1 è più affine alla feature X da lei stessa calcolata. Il prediction vector quindi sarà inviato a tutte le capsule del livello L+1 e, una volta confrontato con l'*activity vector* di queste, sarà inviato un *top-down feedback* che incrementerà o meno il coupling-coefficient c_{ij} in base all'affinità della capsula L con le capsule L+1, in modo da ottenere una distribuzione di probabilità. Tale distribuzione indica a quali capsule-parent è più simile la capsula-child.

Per esempio, il livello L+1 è formato da due capsule che si interessano una della feature Y e l'altra della feature K. La capsula del livello L si accorge che la feature X da lei calcolata ha grande affinità con la feature Y rispetto che con K, così il coefficiente c_{xy} aumenta mentre c_{xk} diminuisce. La capsula di L quindi, sentendosi più affine alla capsula di L+1 che calcola Y, preferisce scambiare i dati di allenamento maggiormente con questa piuttosto che con l'altra capsula che calcola K. I percorsi delle varie attivazioni rappresentano la *gerarchia* delle features ($occhio \rightarrow viso \rightarrow persona$).

Poichè le capsule sono entità **indipendenti** tra loro, tramite questo routing se due capsule concordano molto tra loro allora la probabilità che la feature sia stata calcolata correttamente è molto alta e attendibile.

3.2.2 Coefficiente di Accoppiamento

Ogni connessione tra due capsule è rappresentata da un certo *coefficiente di accoppiamento* c_{ij}, parametro che indica la distribuzione di probabilità

(intesa come affinità) di una capsula del livello k a tutte le capsule del livello *k+1*. Più una capsula di k è affine ad una capsula k+1, più questo coefficiente è alto, dalla cui grandezza dipende l'importanza delle informazioni inviate dalla prima alla seconda. Essendo una distribuzione di probabilità, data una capsula i la somma dei coefficienti di tutte le sue connessioni dovrà essere 1:

$$\sum_{j=1}^{n} c_{ij} = 1 \tag{3.4}$$

Ogni c_{ij} è un parametro allenabile della rete aggiornato dall'algoritmo di routing, impostato inizialmente a 0 ma poi incrementato o decrementato a seconda di come la capsula-figlia concorda con la capsula-padre. Questa affinità padre-figlio, come spiegato nella sezione 3.2.1, dipende da come la predizione del figlio riguardo il futuro output del padre (rappresentata dal prediction vector $\hat{u}_{j|i}$) rispecchi effettivamente l'output della capsula-padre, ossia l'activity vector di quest'ultimo che chiameremo v_j. Avendo questi due dati alla mano, possiamo quindi calcolare l'affinità tra le due capsule, rappresentata matematicamente dall'equazione:

$$a_{ij} = v_j \cdot \hat{u}_{j|i} \tag{3.5}$$

Infine è necessario trasformare questi n valori reali c_{ij} in una distribuzione di probabilità, la cui sommatoria darà sempre risultato **1**. A questo scopo viene utilizzata la funzione **softmax** che prende in input un vettore di n valori reali e lo trasforma in un vettore di n valori reali ma compresi nell'intervallo $[0, 1]$.

$$c_{ij} = \frac{\exp(b_{ij})}{\sum_n \exp(b_{in})}$$

dove b_{ij} è la somma degli agreements a_{ij} tra una capsula del livello i e una capsula del livello j dopo t iterazioni:

$$b_{ij} = \sum_t a_{ijt}$$

Ad ogni step del training ogni b_{ij} sarà aggiornato in base all'affinità a_{ij} calcolata:

$$b_{ij} = b_{ij} + a_{ij} \tag{3.6}$$

Tramite questa funzione otteniamo quindi la distribuzione di probabilità richiesta, garantendo appunto che la somma di tutti i c_{ij} di una capsula i sia **1**. Questo tipo di aggiornamento fa si che le connessioni tra capsule affini tra loro diventino sempre più importanti (con un alto valore c_{ij}). Dopo alcune

iterazioni, i coefficienti vengono sempre aggiornati e diventano sempre più alti quelli relativi a genitori più concordanti con la feature calcolata dal figlio, indicando appunto che la presenza della capsula-figlio implica la presenza della capsula-padre.

3.2.3 Matrice peso

Oltre ai singoli coefficienti di accoppiamento ogni connessione è caratterizzata da una matrice peso W_{ij} contenente le informazioni riguardo la relazione spaziale tra la feature della capsula i (es. naso) e la feature della capsula j (es. faccia). Anche questo è un parametro aggiornato dalla rete ad ogni step.

3.2.4 Routing-by-agreement

L'algoritmo di routing proposto da Hinton viene sostituito al maxpooling delle convolutive. Quest'ultimo infatti garantisce solamente la *Translation Invariance* (classificando l'immagine input nello stesso modo indipendentemente da come le informazioni dentro questa sono spostate) non riuscendo però a calcolare le informazioni spaziali relative tra le features ne informazioni come rotazione e grandezza di un immagine. Così, oltre alla translation invariance, tramite il *routing-by-agreement* viene assicurata anche l'**equivarianza**, riconoscendo una faccia qualsiasi sia il suo angolo di rotazione o posizione.

Indichiamo con u_i l'*activity vector* della capsula i. Come detto prima, questo vettore output viene moltiplicato con la matrice peso W_{ij} di una determinata connessione child-parent. Essendo un prodotto *righe*×*colonne*, l'output sarà sempre un vettore $\hat{u}_{j|i}$ chiamato *prediction vector*. Questo vettore rappresenta la prediction della capsula-child riguardo l'output futuro della capsula-parent :

$$\hat{u}_{j|i} = W_{ij} \cdot u_i \tag{3.7}$$

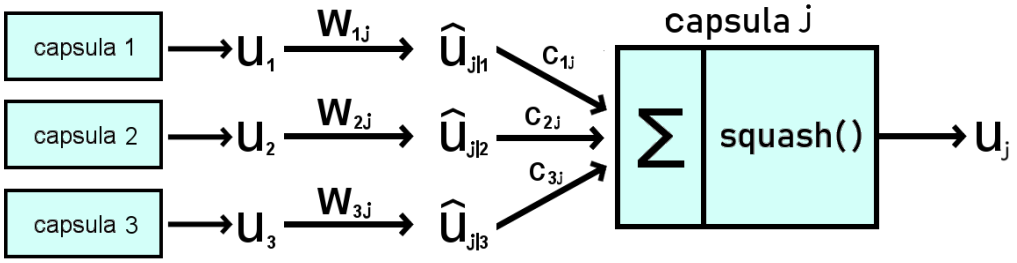
dove: **i** è indice della capsula-child; **j** è indice della capsula-parent; W_{ij} è la matrice peso tra capsula i e j; $\hat{u}_{j|i}$ è il prediction vector.

L'input totale che arriva ad una capsula di un livello superiore è la somma pesata di tutti i *prediction vector* $\hat{u}_{j|i}$ output delle capsule del livello prima:

$$s_j = \sum_i c_{ij} \cdot \hat{u}_{j|i} \tag{3.8}$$

Da questa equazione è evidente l'importanza dei vari c_{i_j} nel routing da una capsula all'altra: questi pesi faranno in modo che dati provenienti da capsule

precedenti affini (e quindi con un valore di c_{ij} alto) siano più importanti e rilevanti per la nuova capsula j rispetto invece ad altre capsule il cui prediction vector è più basso, e quindi non molto rilevanti.
s_j sarà l'input con il quale la nuova capsula j calcolerà nuove features più complesse rispetto al livello precedente i.



3.2.5 Funzione Squashing

La lunghezza dell'*activity vector* di ogni capsula indica la probabilità della feature (calcolata dalla stessa capsula) di esistere nell'immagine passata in input, per questo dovrà avere un valore $\in [0, 1]$. Viene così introdotta una nuova funzione di attivazione non lineare chiamata **Squash**, che schiaccia il vettore a cui è applicata ad un valore minore di 1, mantenendo comunque la sua **direzione**. É importante garantire quest'ultima proprietà perchè contiene informazioni (rotazione, posizione, grandezza) della feature necessarie per le capsule. Il nuovo activity vector v_j sarà dato dalla formula:

$$v_j = \frac{\left\| s_j^2 \right\|}{1 + \|s_j\|^2} \frac{s_j}{\|s_j\|} \tag{3.9}$$

Di seguito è presentato lo pseudocodice dell'algoritmo Routing-by-agreement proposto da Hinton:

Algorithm 2 Routing-by-agreement

function RouteByAgree($\hat{u}_{j|i}$, r, l)
    Per tutte le capsule i nel livello l e per tutte le capsule j nel livello *(l+1)*: $b_{i_j} \leftarrow 0$;
    for r iterazioni **do**
        per tutte le capsule i nel livello l: $c_i \leftarrow softmax(b_i)$;
        per tutte le capsule j nel livello *(l+1)*: $s_j \leftarrow \sum_i c_{ij} \cdot \hat{u}_{j|i}$;
        per tutte le capsule j nel livello *(l+1)*: $v_j \leftarrow squash(s_j)$;
        per tutte le capsule i nel livello l e per tutte le capsule j nel livello *(l+1)*: $b_{i_j} \leftarrow b_{i_j} + \hat{u}_{j|i} \cdot v_j$;
    end for
return v_j
end function

3.2.6 Loss Function

Hitnon presenta il modello di rete a capsule applicandolo al database **MNIST**, un dataset di 70.000 immagini di cifre scritte a mano usato per l'addestramento di reti per l'image-processing. Il livello di output della rete è composto da dieci nodi, uno per ogni possibile cifra. Alla fine di ogni step di allenamento verrà calcolata la funzione di loss utilizzando l'activity vector v_c, output di ciascuna classe. Verranno calcolati quindi 10 valori diversi, sommati poi insieme per ottenere il valore di *loss totale* che determinerà l'aggiornamento della rete:

$$tot = \sum_c L_c \tag{3.10}$$

Il singolo value-loss L_c di ogni classe c ad ogni step è dato dalla formula:

$$L_c = T_c max(0, m^+ - \|v_c\|)^2 + \lambda(1 - T_C) max(0, \|v_c\| - m^-)^2 \tag{3.11}$$

L'indice $\boldsymbol{c}$ si riferisce alle singole classi dell'ultimo livello, una per ogni cifra. T_c può assumere solo due valori: 1 se stiamo calcolando la loss della classe giusta (ossia la classe corrispondente alla label dell'immagine input), 0 se invece stiamo calcolando la loss delle altre nove classi errate.

La formula è possibile dividerla in due parti esclusive tra loro: se stiamo calcolando la loss della classe giusta viene valutata soltanto la parte sinistra; negli altri nove casi (corrispondenti alle classi errate) viene valutata invece la parte destra.

$m^+ = 0.9$ e $m^- = 0.1$ sono costanti utili a fissare una soglia di accuratezza della predizione. m^+ è relativa solo alla classe giusta, m^- alle altre classi.

Consideriamo le due parti separatamente:

$$T_c max(0, m^+ - \|v_c\|)^2 \tag{3.12}$$

Se la decisione della rete, ossia la probabilità dell'immagine di appartenere alla categoria c ($\|v_c\|$) è molto sicura, con una probabilità $\|v_c\| \geq m^+ = 0.9$, allora il valore di tutta questa equazione è 0.

$$\lambda(1 - T_C) max(0, \|v_c\| - m^-)^2 \tag{3.13}$$

Stesso discorso vale per la seconda parte dell'equazione. Se la rete riconosce che una determinata classe non è giusta con una probabilità molto bassa ($\|v_c\| \leq m^- = 0.1$), allora il valore di loss per quella determinata categoria c è 0.

$\lambda = 0.5$ è un coefficiente utilizzato nella formula per evitare che durante la fase iniziale del training la lunghezza dei vettori v_c venga ridotta troppo. Questo perchè nel calcolo della loss ad ogni step devo sommare dieci distinti valori L_c (uno relativo alla classe giusta e nove per quelle errate). Quindi, essendo alta questa differenza (1 vs 9), è possibile che il valore dato dalle classi sbagliate influenzi troppo l'allentamento. Per questo tramite λ il loro valore viene dimezzato.

Se per la classe giusta otteniamo una classificazione con valore $\|v_c\| \geq m^+ = 0.9$, logicamente per tutte le altre classi avremo che $\sum_c \|v_c\| \leq 0.1$. Per questo il valore di loss totale sarà 0 e quindi in questo step non verrà applicato nessun aggiornamento dei pesi.

Se per ciascun record del dataset abbiamo che per la classe giusta$\|v_c\| \geq m^+ = 0.9$ e tutte le altre $\|v_c\| \leq m^- = 0.1$, allora la rete è sicura di se ed efficiente nella scelta; la loss function totale sarà 0 ad ogni step e la rete non verrà mai aggiornata perchè ritenuta ottima così com'è.

Discorso opposto avviene quando $\|v_c\| \leq m^+ = 0.9$ per la classe giusta e $\|v_c\| \geq m^- = 0.1$ per le classi sbagliate. Ciò vuol dire che la rete ha ancora qualche incertezza nelle decisioni, non raggiungendo quella determinata soglia di accuratezza specificata da m^+ e m^- e necessita di ulteriori aggiornamenti. In questo caso il valore totale della loss non sarà mai 0 e quindi ciò comporta un aggiornamento delle W_{ij} e c_{ij}.

3.2.7 CNN vs CapsNet

L'allenamento delle CNN è molto più costono e difficile: per poter garantire l'*equivarianza* è necessario comprendere nel training set diversi esempi di rotazioni di questa e di tutte le altre informazioni spaziali che altrimenti la rete non riuscirebbe a rilevare da sola. Le CapsNet al contrario non hanno bisogno di ulteriori esempi di una stessa immagine per poterla riconoscere in tutte le sue varianti. L'*activity vector* contiene tutte le informazioni necessarie per rendere possibile ciò (di quanti gradi è ruotato un oggetto, le sue coordinate...), riducendo notevolmente il numero di record necessari per ottenere un modello adattivo ed efficiente.

Capitolo 4

CNN e CapsNet per Hate Speech Detection

Il lavoro svolto in questa tesi riguarda l'applicazione di due modelli di reti neurali avanzate (*Reti Convolutive* e *Reti a Capsule*) a problemi di Hate Speech Detection. L'idea è nata dalla curiosità di voler confrontare i risultati ottenuti precedentemente con una rete MLP (rete neurale utilizzata per la competizione) con i risultati ottenuti invece con queste reti avanzate, utilizzando lo stesso dataset. Sebbene le reti convolutive siano applicabili maggiormente a problemi di image-processing, quindi dati rappresentati a matrice (come immagini appunto), trovano una discreta applicazione ad altri problemi di diversa natura, tra cui il Natural Language Processing, come descritto nella sezione 1.1. La difficoltà nel passaggio da MLP a Reti Convolutive per NLP è quella di dover rappresentare le frasi in una forma adeguata all'input richiesto da un livello convolutivo e renderle utilizzabili nell'operazione di convoluzione.

4.1 Reti Neurali Convolutive per Natural Language Processing

La più grande innovazione che le reti convolutive hanno portato nel campo del Machine Learning è stata quella di riuscire a calcolare informazioni spaziali tra le features di uno stesso record tramite l'operazione di convoluzione, cosa impossibile da implementare nelle MLP per via della loro architettura. La convoluzione, presentata nella sezione 3.1.1, prevede uno spostamento del kernel in tutta la matrice-immagine con conseguente moltiplicazione element-wise ad ogni step. Se ci trovassimo quindi davanti ad un problema di image-processing sicuramente non avremo alcun problema nell'inviare i

dati alla rete ed elaborarli, perchè le immagini sono matrici di pixel e per questo hanno una forma adeguata al tipo di rete che utilizziamo. Discorso diverso avviene per i problemi di Natural Language Processing, dove il dato principale da analizzare è una frase invece che un'immagine: è necessario dover adattare il tipo di dato (stringa) a quello compatibile con l'operazione di convoluzione (matrice). Per fare ciò, abbiamo bisogno di alcuni passaggi preliminari prima di arrivare alla creazione della matrice-frase, presentati nel capitolo 5.2.

L'operazione di convoluzione viene applicata alla matrice-frase diversamente da come avviene in una matrice-immagine: il kernel adesso non deve calcolare informazioni spaziali tra pixel locali bensì tra intere parole vicine tra loro. Per questo si tratta di una convoluzione che non avviene in due dimensioni come nell'image-processing (dal primo pixel in alto a sinistra fino all'ultimo in basso a destra) ma in una sola dimensione, scansionando le parole che compongono la frase (dalla prima parola fino all'ultima). Il kernel per questo viene adattato alla forma di rappresentazione usata per le frasi: dovendo comprendere intere parole (tutte rappresentate con un vettore di lunghezza fissa) avrà una *width* fissa appunto ma con una *height* variabile in base alle scelte del programmatore. Da quest'ultimo parametro infatti dipenderanno le informazioni spaziali calcolate tra le singole parole: se *height=2*, allora ad ogni step il kernel calcola features spaziali relative a due parole consecutive; se invece *height=3*, le parole consecutive analizzate insieme saranno tre.

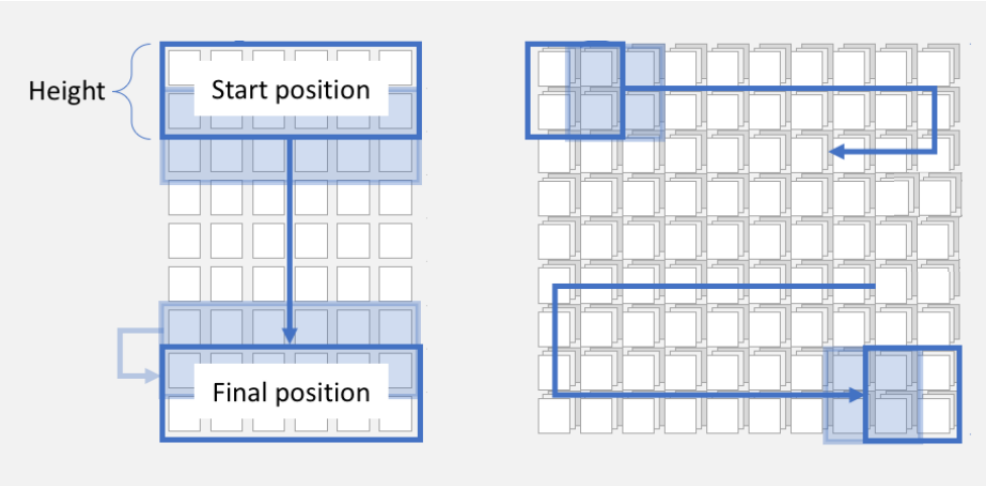
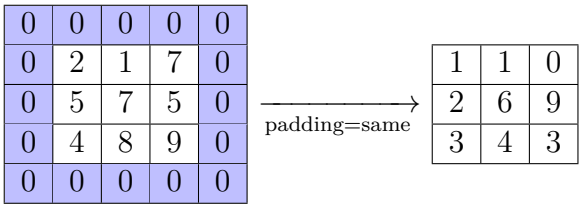


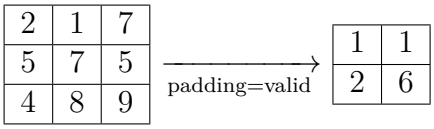
Figura 4.1: Confronto tra la convoluzione 1D impiegata nei problemi NLP (sinistra) e convoluzione 2D per image-processing (destra)

4.2 Struttura base di una CNN

La struttura base di una rete neurale convolutiva è formata da questi layer, componendo i quali possiamo creare reti più o meno complesse:

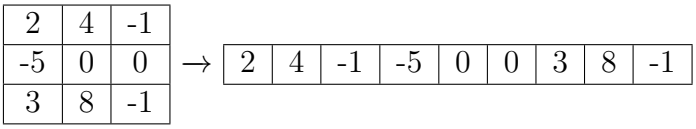
- **Embedding Layer**: è il livello di input con lo scopo di creare la matrice-frase e inviarla al primo layer convolutivo;
- **1D Convolutional Layer**: è il layer dove avviene l'operazione di convoluzione su una sola dimensione rappresentato dai seguenti parametri:
  - *filters*: numero di filtri applicati all'input tramite convoluzione, corrispondente alla dimensione dell'output di quel determinato livello. Ogni filtro restituisce una feature-map che sarà inviata al layer successivo;
  - *kernel_size*: specifica l'altezza della window applicata alla matrice dove avviene la convoluzione ad ogni step. Ricordiamo che si parla solo di altezza della window e non di larghezza perchè quest'ultima sarà pari alla dimensione scelta per rappresentare tutte le parole;
  - *strides*: specifica la lunghezza dello spostamento della window, ossia di quanto quest'ultima si muove nella matrice ad ogni step;
  - *activation*: funzione di attivazione usata nel layer;
  - *padding*: durante l'operazione di convoluzione tra un'immagine e un filtro è possibile che la feature map risultante abbia una dimensione diversa dall'immagine di partenza. Questo perchè i pixel ai bordi sono soggetti alla convoluzione meno volte rispetto agli altri pixel in mezzo, con conseguenza perdita di informazioni ai bordi dell'immagine. Il parametro padding permette appunto di gestire questo problema in base alle nostre necessità. Se *padding=same*, ai bordi dell'immagine verranno aggiunti tanti pixel di valore 0 quanto basta per ottenere una feature map della stessa dimensione. Se *padding=valid* invece l'immagine input viene lasciata così com'è, senza aggiunta di bordi (pad) aggiuntivi, fornendo in output una feature map di dimensioni diverse e ridotte.





- **1D MaxPooling Layer**: livello dove viene applicato il maxpooling spiegato nella sezione 3.1.2 e relativa riduzione di dimensionalità delle feature-maps. Questo layer segue tutti i tre livelli convolutivi sopra citati;

- **Flatten Layer**: è il livello di passaggio dalla parte convolutiva alla rete MLP. Ha il compito di rimodellare l'output della CNN (feature-maps in forma di matrice) trasformandolo in un vettore input adatto alla *fully connected network*.



Subito dopo il Flatten Layer troviamo una MLP con il compito di classificare il record di input in base alle features calcolate dai livelli convolutivi.

4.2.1 Learning Rate Scheduler

All'intera rete viene implementato un **Learning Rate Scheduler**, una funzione utile a ridurre la grandezza del *learning rate* all'avanzare delle epoche. Ai primi passi di allenamento di una rete ci troviamo ancora lontani dal minimo della funzione di loss, quindi non è un problema se vengono effettuati grandi spostamenti nella funzione stessa. Andando avanti con le epoche invece il modello diventa sempre più efficiente, quindi teoricamente si dovrebbe avvicinare sempre più al minimo, ma se utilizzassimo lo stesso learning rate iniziale è possibile che passiamo da una parte e l'altra del minimo, senza trovarlo o senza avvicinarsi, soprattutto in presenza di funzioni con elevato grado di pendenza.

Figura 4.2: Learning rate fisso VS Learning rate adattivo

Gli spostamenti dovrebbero essere attenuati con l'aumentare delle epoche così che, essendo più piccoli, è molto più probabile trovare il minimo della funzione di loss, senza rischiare di girargli intorno senza trovarlo o di fermarsi su minimi locali. Esistono diverse tecniche per implementare learning rate adattivi; i più noti sono *step decay* e *exponentional decay*.

Nel caso dello **step decay** il learning rate viene ridotto secondo la formula:

$$lr = lr0 \cdot \lambda^{floor(\frac{1+n}{r})} \tag{4.1}$$

dove **lr** è il nuovo learning rate aggiornato; **lr0** è il learning rate quando l'aggiornamento viene richiesto; λ indica di quanto il learning rate viene diminuito (0.5 corrisponde alla metà); **n** è il numero di epoche passate; **r** la frequenza di aggiornamento, ossia quante epoche devono passare tra un aggiornamento e l'altro; **floor()** è la funzione che riduce a 0 i valori più piccoli di 1: se *r=10*, per le prime nove epoche *floor()* darà come risultato 0, quindi non è richiesto alcun aggiornamento perchè *lr=lr0*; all'epoca numero dieci invece l'output di *floor()* sarà 1, con conseguente aggiornamento di lr0.

Nel caso dell' **exponential decay** invece il learning rate viene aggiornato secondo la funzione:

$$lr = lr0 \cdot e^{-\lambda n} \tag{4.2}$$

In quest'ultimo caso la diminuzione del learning rate segue una funzione esponenziale perciò è molto più veloce rispetto allo step decay che invece prevede una diminuzione lineare.

4.3 Modello CNN multichannel

Il modello CNN sopra citato è un modello *single-channel* dove i dati entrano nella rete attraverso un solo canale di input. Durante il lavoro di tesi è stato testato anche un modello **multi-channel** che, contrariamente al primo, prevede più ingressi di input alla rete che seguiranno un proprio flusso con architetture di rete indipendenti, le quali poi uniranno i loro output in uno solo, comprendendoli tutti. Questo modello permette di utilizzare diversi tipi di rete (CNN, MLP, ...) in una stessa architettura, consentendo di gestire più flussi di analisi per lo stesso input.

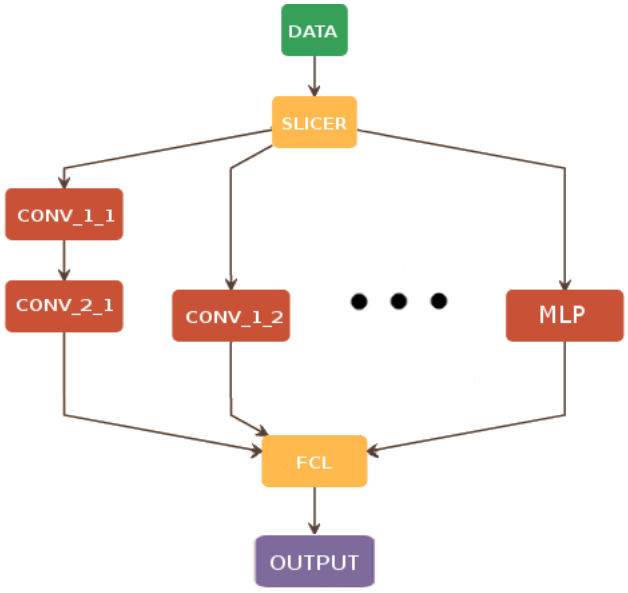


Figura 4.3: Esempio di architettura multichannel

Questa nuova architettura ha permesso di testare la rete CNN aggiungendo le features del progetto HaSpeeDe oltre quelle già presenti nel modello single-channel calcolate dalla CNN stessa. Tale implementazione è stata eseguita per testare se raccogliere insieme **features CNN + features Hand-Crafted** può portare miglioramenti alla rete oppure no. In tutto sono stati effettuati tre test con lo stesso set di features per ogni record ma con diversa normalizzazione:

- **Hand-Normalization**: le features sono state modificate a mano in base a come tali modifiche si prestavano meglio alla rete, come nel progetto HaSpeeDe. Questo perchè i valori sono di diverse scale di misura (interi, reali, percentuali, ...). Si tratta di una pseudo-normalizzazione ma che nella competizione ha portato guadagno alla rete;
- **Nessuna normalizzazione**: le modifiche citate nel punto precedente non sono state applicate e i valori delle features sono stati passati alla rete così come si presentano nel dataset. Anche questo è un approccio sbagliato al problema perchè essendo dati di diversa natura e unità di misura richiedono necessariamente una normalizzazione;
- **Normalizzazione min-max**: è il tipo di normalizzazione più utilizzato e consiste nello scalare i valori di un vettore nel range $[0; 1]$ o $[-1; 1]$ secondo la formula:

$$x^{'} = \frac{x - min(x)}{max(x) - min(x)} \tag{4.3}$$

4.4 Capsule Network

Il modello di Capsule Network presentato in questa tesi è quello base introdotto da Hinton nell'articolo [5] ma, invece che applicato all'image-processing del dataset MNIST, è stato adattato al NLP. Sebbene sia un modello diverso dalla CNN presentata sopra, la parte iniziale di encoding del testo è la stessa: partendo da frasi variabili e tutte diverse tra loro in termini di lunghezza, vogliamo uniformarle in modo che seguano una determinata forma per riuscire poi a creare le matrici con cui la convolutiva lavorerà, come spiegato nella sezione 4.1.

Il modello di Hinton presenta la seguente architettura:

- **Embedding Layer**: come nella CNN, è il livello iniziale che prevede la preparazione dei vari dati in input alla rete, trasformando la frase da sequenza di parole in matrice;
- **Convolutional Layer**: dopo il livello di input, le singole frasi formattate in forma matriciale vengono passate ad un livello convolutivo. Si trova prima del Capsule Layer perchè il compito di quest'ultimo è combinare tra loro le features di uno stesso record e non di calcolarle, rendendo necessaria questa fase preliminare di raccolta delle informazioni con le quali poi le capsule potranno operare. Le features calcolate tramite convoluzione saranno quindi l'input del primo Capsule Layer;

- **Dropout Layer**: è un livello di dropout;
- **Primary Capsule Layer**: è un livello convolutivo a capsule. Riceve in input le features calcolate dal *Convolutional Layer*. In questo livello viene applicata la funzione *squash* per garantire sia la non-linearità dei dati, sia la lunghezza dell'activity-vector compresa nell'intervallo $[0, 1]$ (perchè rappresenta una probabilità). Il numero di capsule utilizzato in questo livello dipende dal dataset utilizzato (per conservare le informazioni spaziali più importanti Hinton usa 32 capsule per il dataset MNIST);
- **Output Capsule Layer**: è il livello di output dove ogni classe del task è rappresentata da una singola capsula. Nel paper di Hinton sono utilizzate in tutto dieci capsule-output corrispondenti alle dieci cifre del dataset MNIST. Ogni capsula del *Primary Capsule Layer* invia il proprio activity vector a tutte le capsule di questo livello. É in questa parte di rete che viene applicata l'innovazione introdotta da Hinton: il *routing-by-agreement* (3.2.4). Questo infatti può essere applicato solamente tra due capsule, quindi nel passaggio tra un capsule-layer e l'altro, come appunto tra *Primary Capsule Layer* e *Output Capsule Layer*.
- **Lenght Layer**: rimpiazza ogni capsula del livello output con la relativa lunghezza. Come spiegato in 3.2.1, l'output di una capsula è un *activity vector* la cui lunghezza corrisponde alla probabilità di quella features di esistere nell'immagine. Quindi, ottenute le capsule di output dal livello *Output Capsule Layer*, per completare la classificazione è necessario ottenere le probabilità di tutte le capsule/classi, scegliendo poi quella con probabilità maggiore. Viene così calcolata la *magnitudine* dell'activity vector di ogni capsula, corrispondente alla propria lunghezza e, quindi, probabilità di appartenenza ad una classe rispetto che altre. La classe *Lenght* non fa altro che applicare la formula 4.4 all'activity vector v per il calcolo della relativa magnitudine:

$$\|v\| = \sqrt{v_1^2 + v_2^2 + ... + v_n^2} \tag{4.4}$$

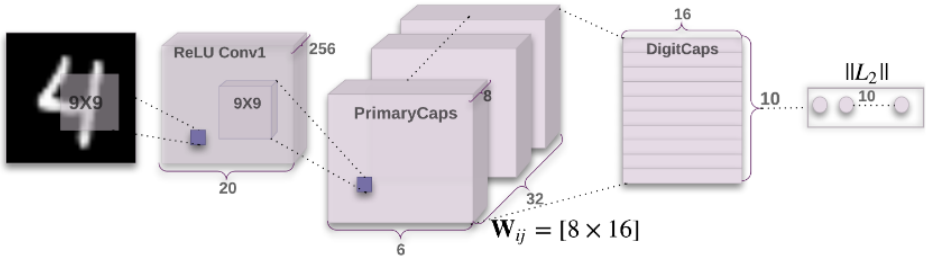


Figura 4.4: Capsule Network presentata da Hinton nell'articolo [5]

Capitolo 5

Risultati

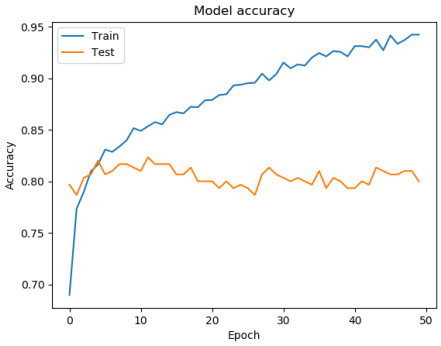
Come metodo di confronto tra i modelli presentati in questa tesi è stato scelto di testare le reti su **30 epoche** calcolando l'accuratezza nel test-set ogni cinque per analizzare meglio la convergenza della rete. Questo è stato ripetuto per tre volte e come risultato finale viene considerata la media delle accuratezze. Ogni test è stato eseguito sia con una **batch-size** di **64** che **128**.

5.1 Risultati MLP

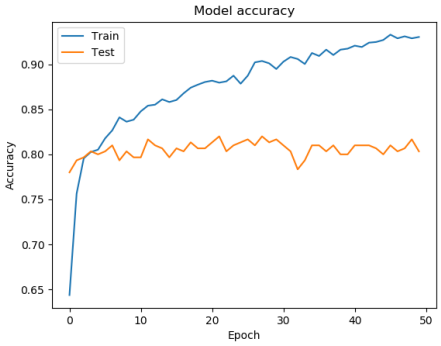
Per prima consideriamo la rete MLP proposta per la competizione HaSpeeDe, i cui risultati hanno rappresentato una *baseline* per lo sviluppo degli altri modelli. L'architettura della MLP è presentata nella sezione 2.6 e per un miglior confronto con le altre reti è stata allenata esclusivamente con il dataset di Facebook (3600 commenti per il training e 400 per il testing) secondo le dinamiche spiegate all'inizio di questo capitolo. I risultati ottenuti sono riassumibili nella seguente tabella:

	64						128					
	5	10	15	20	25	30	5	10	15	20	25	30
MLP	0.815	0.819	0.813	0.816	0.81	0.816	0.811	0.809	0.819	0.819	0.818	0.811

Per una migliore lettura delle performance dei due modelli vengono presentate le relative curve di convergenza: con il colore arancione è indicato l'accuratezza del modello calcolata sul test-set in funzione delle epoche; con il blu invece l'accuratezza durante il training.



Batch-size = 64



Batch-size = 128

Il modello MLP riesce a raggiungere un'accuratezza pari a **0.819** sebbene vi sia presenza di *overfitting*, problema dovuto al povero dataset di soli 4000 commenti a nostra disposizione: quando il numero di record è molto piccolo rispetto alla complessità della rete (e quindi alla quantità di parametri delle connessioni e nodi), accade che con così tanti parametri e pochi record la rete riesce ad adattarsi al training set, imparandolo a memoria e quindi non riuscendo a generalizzare adeguatamente il problema per poter classificare dati mai visti prima. Dalla curva di convergenza l'overfitting si nota controllando i comportamenti delle curve (training e test) in relazione tra loro: l'accuratezza nel training continua a salire linearmente con il numero delle epoche mentre quella del test rimane in un intervallo intorno al **0.80**, senza salire. Questo vuol dire che la rete sta imparando a memoria i dati in input e non riesce a generalizzare il problema, migliorando nel training ma rimanendo inalterata nel test.

Inoltre sempre dai grafici di convergenza si può dedurre che la rete MLP costruita è un modello instabile poichè le due curve variano molto da un'accuratezza e l'altra, non riuscendo a stabilizzarsi in un piccolo intervallo. Il modello con *batch=128* sembra comunque leggermente più accurato e più stabile rispetto al modello con *batch=64*.

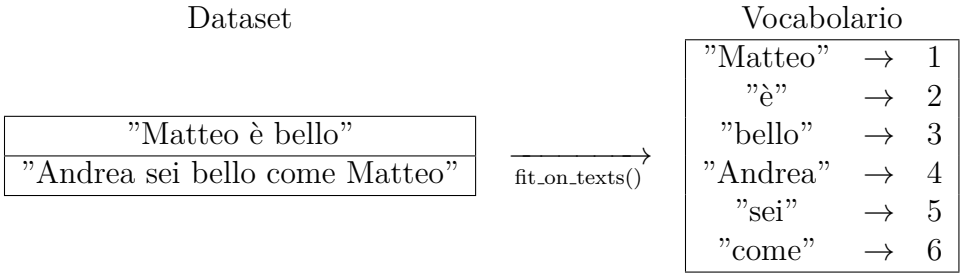
Questi risultati sono stati presi come punto di riferimento (baseline) per valutare i risultati ottenuti con modelli di rete più avanzati.

5.2 Risultati CNN

Prima di passare all'effettiva creazione e applicazione delle reti convolutive è necessario dover convertire le frasi (stringhe) in matrici in modo che possa essere usata l'operazione di convoluzione. Sia nel modello di *Rete Convolutiva* che *Rete a Capsule* tale operazione è stata effettuata utilizzando la classe **Tokenizer** di **Keras**.

5.2.1 Tokenizer e creazione della matrice-frase

Dopo aver diviso il training-set dal test-set, tramite la classe *Tokenizer* ad ogni frase viene associata una rappresentazione vettoriale.
Per prima cosa, tramite la funzione *tokenizer.fit_on_texts()* generiamo un vocabolario *word* → *index*, associando ad ogni parola del dataset un indice intero univoco.



Una volta indicizzata ogni parola, è il momento di vettorializzare le singole frasi con la funzione *tokenizer.texts_to_sequences()*: presa una frase, ogni parola contenuta in questa viene sostituita dall'indice corrispondente nel vocabolario creato nel passo precedente.

"Matteo è bello" → [1 2 3]
"Andrea sei bello come Matteo" → [4 5 3 6 1]

Le singole frasi ottenute fino a questo punto non sono altro che vettori di interi di diversa lunghezza in base al numero di parole che contengono. Il prossimo passo sarà quello di uniformare la grandezza delle frasi in modo che tutte siano rappresentabili da vettori della stessa dimensione. Ciò è reso possibile dalla funzione *tokenizer.pad_sequences()*: fissata la massima lunghezza che le frasi devono avere (indicata con **maxlen**), ognuna di queste sarà rappresentata da un vettore di maxlen dimensionalità. Se la frase ha un numero di parole minore a maxlen, le posizioni rimanenti saranno riempite da 0; se invece è maggiore, verrà troncata in modo che possa essere contenuta in un vettore lungo maxlen.

"Matteo è bello" → [1 2 3] → [1 2 3 0]
"Andrea sei bello come Matteo" → [4 5 3 6 1] → [4 5 3 6]

Alla fine di questi passaggi avremo una rappresentazione di tutte le parole utilizzate nel dataset secondo vettori di numeri interi di dimensione uguale, corrispondenti all'indice delle parole contenuto nel vocabolario word→index. Tutto ciò è necessario per poter utilizzare le frasi sotto forma di matrice e quindi per poterle analizzare con una rete convolutiva.

La conversione da frase a matrice è resa possibile grazie al livello **Embedding()** di Keras. Viene utilizzato come primo livello di una rete consentendo di inviare in input vettori invece che singoli valori reali.

```
keras.layers.Embedding(
                        input_dim = vocab_size,
                        output_dim = embedding_dim,
                        input_length = maxlen,
                        weights = [embedding_matrix]
                        )
```

Il parametro **input_dim** indica la grandezza del vocabolario creato dal Tokenizer; **output_dim** è la dimensione utilizzata nell'embedding, ossia la dimensione del vettore che rappresenta la singola parole (tale valore corrisponde al numero di *colonne* della matrice-frase); **input_lenght** indica la lunghezza della frase. Le frasi devono essere tutte uniformate ad una lunghezza standard, evitando la presenza di frasi più o meno lunghe. Nel nostro progetto la lunghezza standard associata alle frasi è rappresentata dalla variabile *maxlen* e corrisponde al numero di *righe* della matrice-frase.
Infine **weights** è la matrice dei pesi in base alla quale vengono sostituite le parole nella frase. Questa fase di sostituzione prevede due varianti:

- Se non viene passata alcuna matrice dei pesi sarà utilizzato l'embedding di Keras, dove la singola frase è data da un vettore di reali calcolato dallo stesso Embedding Layer come mostrato in figura 5.1:

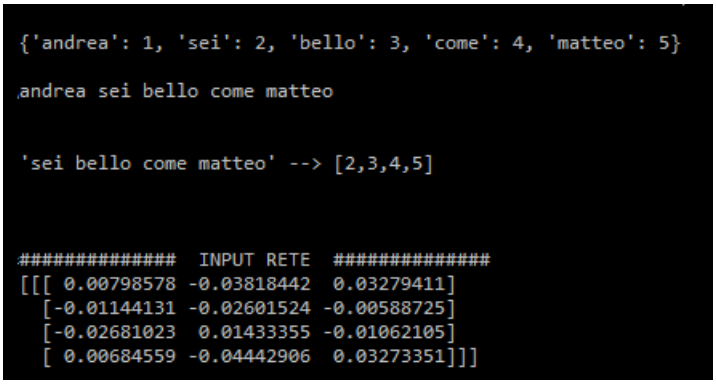


Figura 5.1: Creazione matrice-frase con embedding di Keras

- Viene passata una matrice dei pesi precedentemente creata con un modello di embedding (*Word2Vec*, *Glove*, ...). Ogni parola viene sostituita dal relativo vettore di reali.

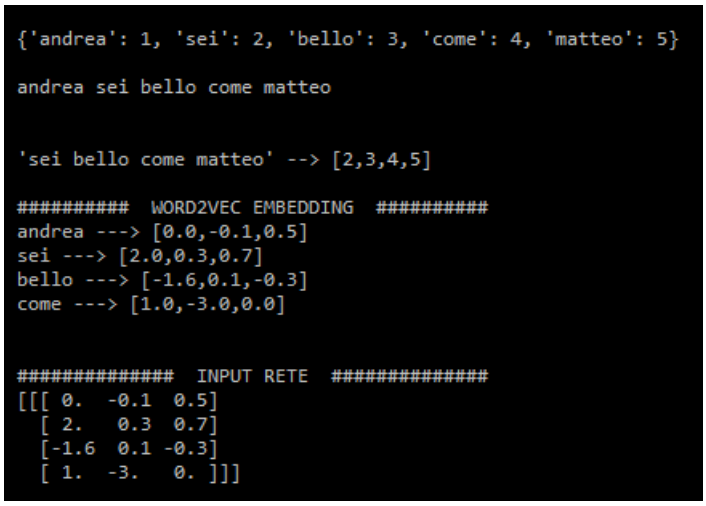


Figura 5.2: Creazione matrice-frase con embedding Word2Vec

Prendendo come esempio la figura 5.2, nella prima riga di output viene mostrato il vocabolario $word \rightarrow index$ creato dal Tokenizer. Sotto si trova la frase passata in input e la frase dopo l'embedding con il relativo vettore di indici con il quale viene tradotta (manca la prima parola "Andrea" perchè il maxlen nell'esempio è stato impostato a 4 ed essendo una frase di cinque parole è stata troncata all'inizio). I vettori-peso del Word2Vec sono mostrati vicino alle parole corrispondenti e infine viene mostrata la matrice-frase creata dall' *Embedding Layer* per poi

essere inviata al primo livello della rete convolutiva.

La matrice avrà un numero di righe pari al maxlen=4 scelto per il dataset, mentre il numero di colonne sarà *output_dim*, la dimensionalità scelta per i vettori Word2Vec (in questo caso di dimensionalità 4). Ogni riga della matrice quindi corrisponde ad una parola della frase passata in input.

Completata questa fase di conversione delle frasi in matrici, siamo pronti ad inviarle come input alla rete convolutiva per l'allenamento.

5.2.2 Architettura proposta

Il modello CNN sviluppato è composto dai seguenti livelli per la parte convolutiva:

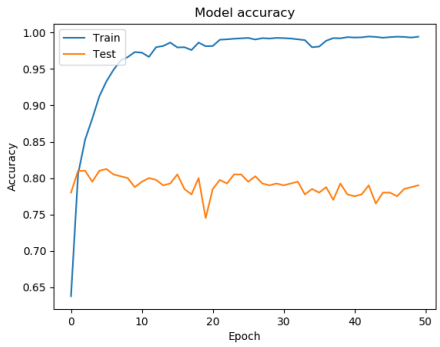
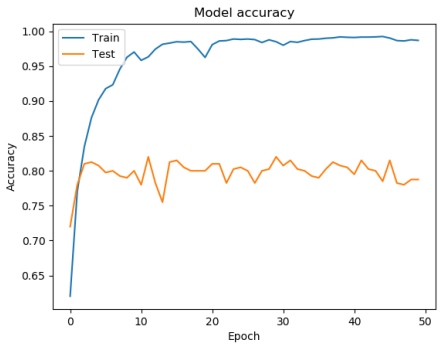
Layer	**# Filtri**	**Filter size**	**ActFuncion**	**Stride**	**Padding**
1D Convolutional Layer	64	2	relu	1	same
1D Convolutional Layer	512	2	relu	1	same
1D Convolutional Layer	64	2	relu	1	same

Per la parte di MLP invece è stata scelta la seguente configurazione:

Name	**# Nodes**	**Function**	**Dropout**
Input Layer	265	relu	0.45
2° Hidden Layer	56	sigmoid	0.45
Output Layer	1	sigmoid	-

I risultati ottenuti da questo modello (nominato **CNN**), messi a confronto con la rete MLP di prima, sono:

	64						128					
	5	10	15	20	25	30	5	10	15	20	25	30
MLP	0.815	0.819	0.813	0.816	0.81	0.816	0.811	0.809	0.819	0.819	0.818	0.811
CNN	0.78	0.788	0.784	0.789	0.781	0.775	0.805	0.793	0.811	0.799	0.8	0.813

Curve di convergenza per modello CNN a 3 livelli	
batch_size = 64	batch_size = 128
	

I risultati tabellari dimostrano che la rete MLP raggiunge livelli di accuratezza maggiori rispetto la CNN per quasi tutti gli intervalli di epoche.

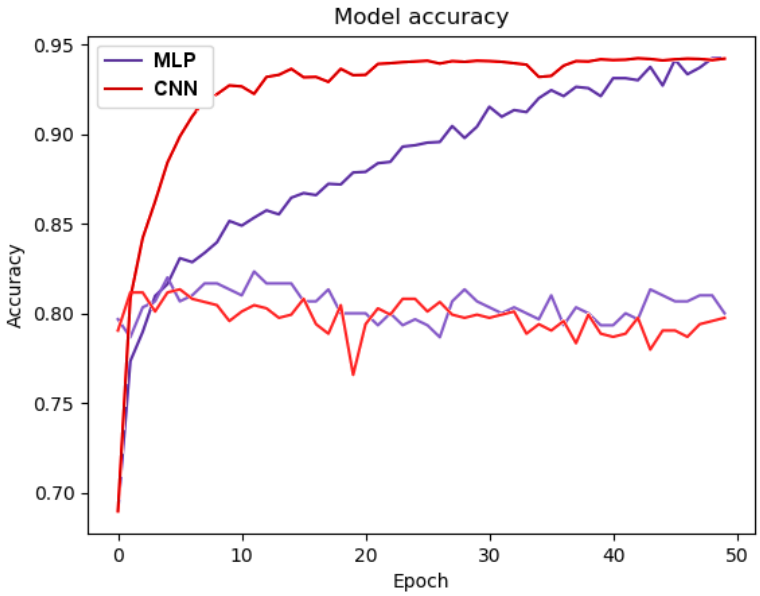


Figura 5.3: Confronto curve di covergenza tra MLP (colore blu) e CNN (colore rosso)

Dalla figura 5.5 è evidente che l'apprendimento della MLP (accuratezza nel training) è molto lento raggiungendo il valore **0.90** verso le 25 epoche, aumentando sempre in maniera lineare. La CNN invece ha una convergenza più veloce, raggiungendo lo stesso traguardo già nelle prime 5 epoche, per

poi stabilizzarsi e rimanere più o meno costante. Subisce una crescita esponenziale nelle prime 10 epoche di allenamento, cosa inceve che non avviene nella MLP. Da ciò possiamo dedurre quindi che la CNN ha bisogno di un numero minore di epoche per raggiungere una buona accuratezza e stabilità, rispetto ad una MLP che presenta un apprendimento lineare.

Osservazione diversa per la test-accuracy (le due funzioni inferiori), dove entrambe le reti sembrano seguire lo stesso andamento molto variabile, senza stabilizzarsi.

Anche qui siamo di fronte anche ad un caso di overfitting dimostrabile dalla curva della funzione di loss:

Figura 5.4: Curva-loss della rete CNN a tre livelli con batch_size=64

Come si vede in figura 5.4, mentre la loss nel training set diminuisce fino a valori molto prossimi allo zero, la loss nel test invece aumenta sempre più con l'avanzare delle epoche, prova del fatto che il modello riconosce bene i dati nel training-set ma non riesce a generalizzare i nuovi dati del test-set. Più impara a riconoscere i dati del train, più il modello è specifico per questi (train-loss diminuisce) e quindi incompatibile e non adattabile a quelli mai visti (test-loss aumenta).

5.2.3 Regolarizzatori di kernel e bias

Per ridurre l'overfitting, stimolando la rete a generalizzare sempre più e quindi essere più performante davanti a nuovi dati mai visti prima, viene introdotta la *weight regularization*. Si tratta di una regolarizzazione applicata alla loss function, alla quale vengono aggiunte delle penalità forzando i pesi delle connessioni nel tendere a 0. Questo perchè una rete neurale con matrici-peso più piccole costruisce modelli più semplici ma generali, mentre un modello troppo complesso è più soggetto a overfitting. La regolarizzazione scelta in questo progetto è la **L2** che prevede l'aggiunta della sommatoria dei vari pesi al quadrato alla loss function scelta:

$$NewErrorFunction = LossFunction + \frac{\lambda}{2m} \cdot \sum \|w\|^2 \tag{5.1}$$

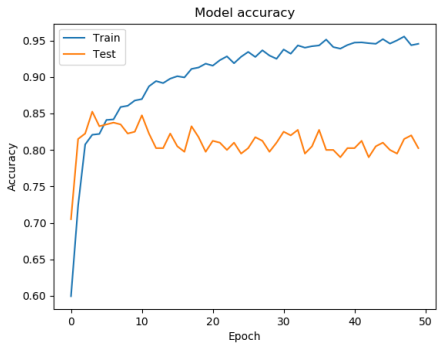
dove λ è un iperparametro dal quale dipende la complessità o meno del modello creato. Non c'è una misura standard per ogni tipo di task, ma dovrà essere scelta in base al dataset a disposizione. In questo progetto è stato scelto $\lambda = 0.5$.

Oltre ai regolarizzatori dei pesi, esistono pure quelli per il termine **bias** della rete. I regolarizzatori possono essere aggiunti in qualsiasi livello della rete, come fatto in questo progetto di tesi: considerando la struttura della sola parte convolutiva della CNN, ognungo dei tre livelli è provvisto sia di **kernel_regularizer = l2(0.05)** che **bias_regularizer = l2(0.05)**. I risultati di questo modello (**CNNreg**) sono aggiunti alla tabella:

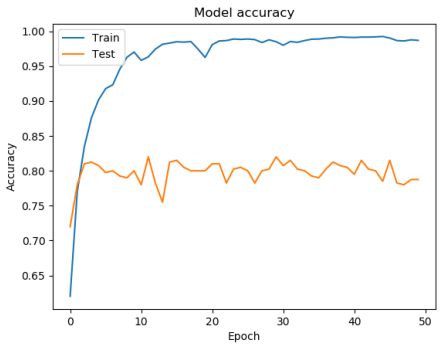
	64						128					
	5	10	15	20	25	30	5	10	15	20	25	30
MLP	0.815	0.819	0.813	0.816	0.81	0.816	0.811	0.809	0.819	0.819	0.818	0.811
CNN	0.78	0.788	0.784	0.789	0.781	0.775	0.805	0.793	0.811	0.799	0.8	0.813
CNNreg	0.836	0.831	0.828	0.82	0.812	0.823	0.831	0.816	0.838	0.81	0.822	0.82

Curve di convergenza per modello CNN a 3 livelli + regolarizzatori kernel e bias

batch_size = 64

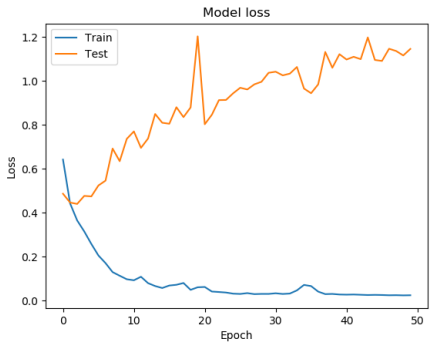


batch_size = 128

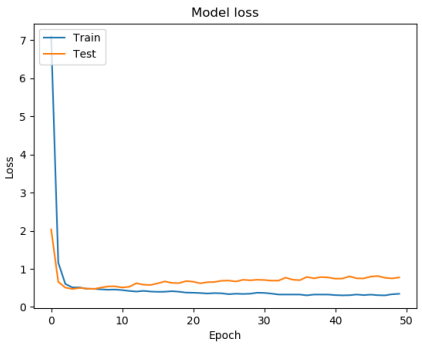


Curva di loss

Modello CNN a 3 livelli senza regolarizzatori



Modello CNN a 3 livelli con regolarizzatori



Sebbene la curva-loss del nuovo modello con i regolarizzatori presenta sempre una piccola crescita con l'aumentare delle epoche, comunque è una crescita decisamente più attenuata rispetto al modello senza regolarizzatori, arrivando anche ad essere costante nelle ultime epoche del test. Inoltre presenta un comportamento molto più regolare e lineare, dimostrando che tramite l'aggiunta dei regolarizzatori abbiamo trovato un miglioramento della rete e del modello, riducendo molto l'overfitting.

5.2.4 Learning Rate Adattivo

Per quanto riguarda l'instabilità della rete dovuta a continui salti del valore di accuratezza, come è possibile vedere dalle curve di convergenza finora presentate mostrando funzioni spezzate e irregolari, la causa principale è dovuta al learning rate costante, problema descritto nella sezione 4.2.1. Ad ogni step viene calcolata la funzione di loss confrontando i risultati ottenuti con quelli

reali del dataset, e in base a quanto sono distanti tra loro questi due valori vengono aggiornati i pesi delle connessioni dalla Backpropagation. Lo scopo è di trovare il minimo della funzione di loss, muovendosi su questa tramite l'algoritmo di *Discesa del gradiente* ogni volta di un certo spazio chiamato *learning rate*. Però, se quest'ultimo ha un valore costante, è molto probabile che il minimo non venga mai trovato e quindi il modello non venga ottimizzato, mostrando comportamenti irregolari come appunto nella curva di convergenza.

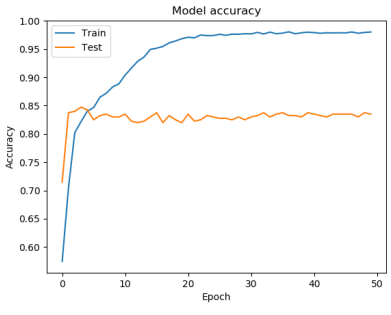
La soluzione adottata è quella di un **learning rate adattivo esponenziale**, spiegato sempre nella sezione 4.2.1. Questo nuovo modello (**CNNtot**) ottiene i seguenti risultati:

	64						128					
	5	10	7	20	25	30	5	10	7	20	25	30
MLP	0.815	0.819	0.813	0.816	0.81	0.816	0.811	0.809	0.819	0.819	0.818	0.811
CNN	0.78	0.788	0.784	0.789	0.781	0.775	0.805	0.793	0.811	0.799	0.8	0.813
CNNreg	0.836	0.831	0.828	0.82	0.812	0.823	0.831	0.816	0.838	0.81	0.822	0.82
CNNtot	0.837	0.839	0.842	0.842	0.839	0.836	0.845	0.845	0.835	0.838	0.841	0.844

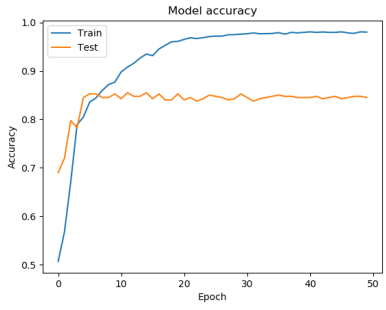
Il modello finale con i regolarizzatori e il learning rate adattivo (**CNNtot**) raggiunge un livello di accuratezza più alto rispetto agli altri modelli, presentando anche un comportamento più lineare e costante, senza troppe variazioni di accuratezza avanzando con il numero di epoche.

Curve di convergenza per modello CNN a 3 livelli + regolarizzatori kerel e bias + adaptive learning rate

batch_size = 64



batch_size = 128



Mentre prima, in presenza di un learning rate costante, la rete non riusciva evidentemente a trovare il minimo della funzione di loss portando ad una curva di convergenza molto irregolare, adesso invece, grazie ad una ricerca del minimo adattiva, la rete riesce ad avvicinarsi meglio al minimo, generando

così un modello più stabile e con una curva molto più regolare e lineare rispetto la prima.

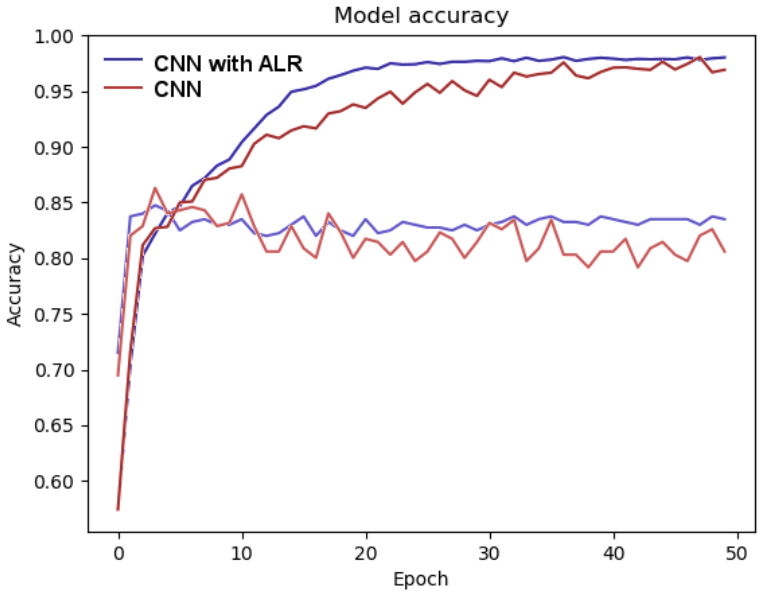
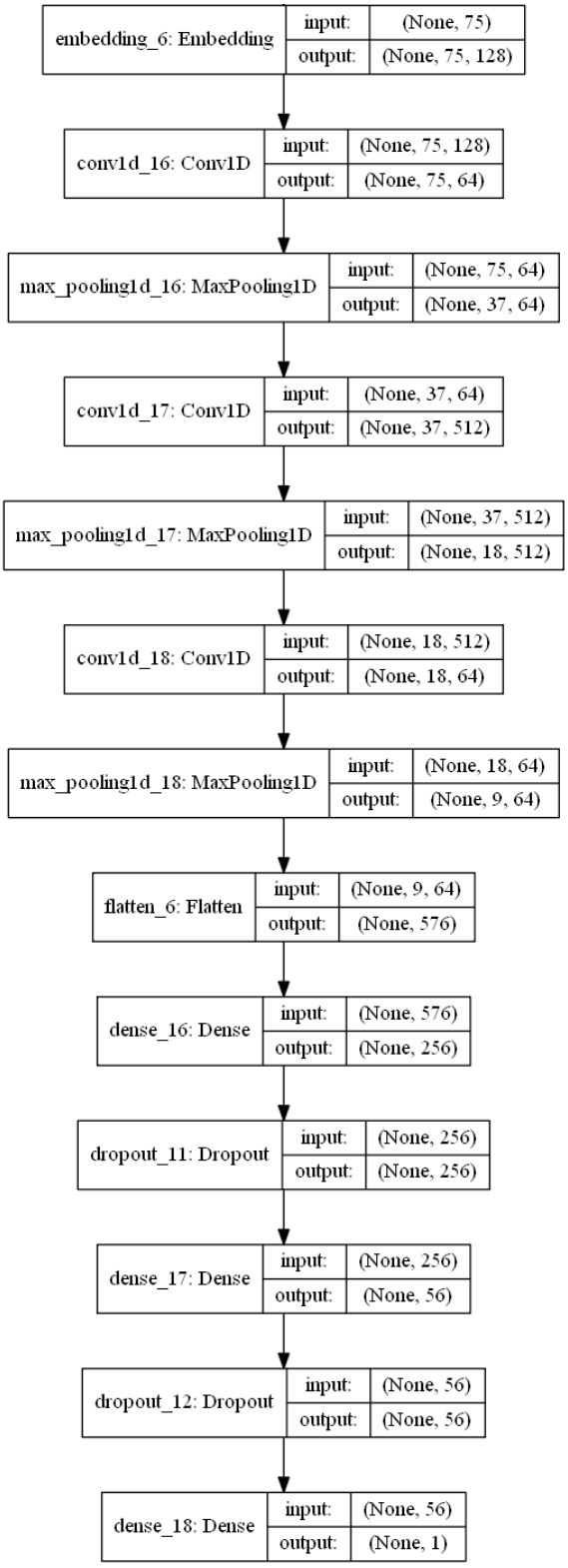


Figura 5.5: Confronto curve di covergenza con adaptive learning rate (colore blu) e senza (colore rosso)

L'architettura del modello CNN a tre livelli convolutivi finale, con regolarizzatori kernel/bias e learning rate adattivo (*CNNtot*) è la seguente:


embedding_6: Embedding
input: (None, 75)
output: (None, 75, 128)
conv1d_16: Conv1D
input: (None, 75, 128)
output: (None, 75, 64)
max_pooling1d_16: MaxPooling1D
input: (None, 75, 64)
output: (None, 37, 64)
conv1d_17: Conv1D
input: (None, 37, 64)
output: (None, 37, 512)
max_pooling1d_17: MaxPooling1D
input: (None, 37, 512)
output: (None, 18, 512)
conv1d_18: Conv1D
input: (None, 18, 512)
output: (None, 18, 64)
max_pooling1d_18: MaxPooling1D
input: (None, 18, 64)
output: (None, 9, 64)
flatten_6: Flatten
input: (None, 9, 64)
output: (None, 576)
dense_16: Dense
input: (None, 576)
output: (None, 256)
dropout_11: Dropout
input: (None, 256)
output: (None, 256)
dense_17: Dense
input: (None, 256)
output: (None, 56)
dropout_12: Dropout
input: (None, 56)
output: (None, 56)
dense_18: Dense
input: (None, 56)
output: (None, 1)

5.3 Risultati CNN multichannel

Nella sezione 4.3 è stato presentato un modello di CNN multichannel che prevede più ingressi di input alla stessa rete, permettendo di utilizzare più tipologie di rete per uno stesso task. Il modello multichannel è stato utilizzato in questa tesi per testare se l'aggiunta delle features del progetto HaSpeeDe, in aggiunta a quelle calcolate dalla CNN, porta profitto alla rete. Le features aggiunte in questione sono elencate nella sezione 2.3.

Oltre a questi tre modelli è stata testata anche un'ulteriore rete composta da tre canali differenti di input.

5.3.1 Multichannel con features

In tutto le varianti dello stesso modello multichannel + features sono:

- **Hand-Normalization**: le features sono state normalizzate a mano, come nel progetto HaSpeeDe (**CnnHN**);
- **Nessuna normalizzazione**: le features sono passate alla rete senza alcuna normalizzazione (**CnnNN**);
- **Normalizzazione min-max**: normalizzazione min-max eseguita utilizzando *Sklearn*, una libreria python che offre algoritmi di classficazione, regressione, clustering e molti altri tool per il Machine Learning. Tramite la funzione *normalize()* viene normalizzato il vettore delle features di ogni record. Le features da normalizzare sono quelle usate nel modello *CnnNN* (**CnnSN**).

I risultati sono confrontati con quelli finora ottenuti dai precedenti modelli sviluppati nella seguente tabella:

	64						128					
	5	10	15	20	25	30	5	10	15	20	25	30
MLP	0.815	0.819	0.813	0.816	0.81	0.816	0.811	0.809	0.819	0.819	0.818	0.811
CNN	0.78	0.788	0.784	0.789	0.781	0.775	0.805	0.793	0.811	0.799	0.8	0.813
CNNreg	0.836	0.831	0.828	0.82	0.812	0.823	0.831	0.816	0.838	0.81	0.822	0.82
CNNtot	0.837	0.839	0.842	0.842	0.839	0.836	0.845	0.845	0.835	0.838	0.841	0.844
CnnHN	0.7	0.696	0.689	0.688	0.688	0.688	0.701	0.753	0.763	0.769	0.769	0.766
CnnNN	0.637	0.666	0.659	0.659	0.657	0.663	0.647	0.671	0.677	0.676	0.678	0.678
CnnSN	0.832	0.833	0.827	0.831	0.82	0.828	0.806	0.803	0.82	0.821	0.828	0.826

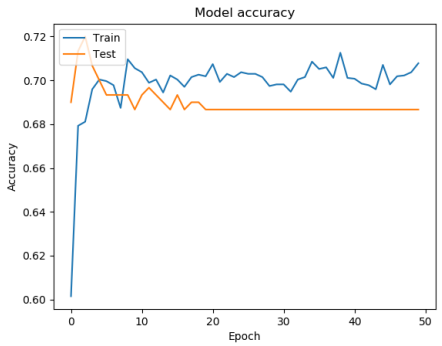
Nel modello **CnnHN** con le features normalizzate a mano (come nella MLP del progetto HaSpeeDe) il massimo valore di accuratezza raggiunto è

di **0.769**. Quindi è stato testato il modello con le features non normalizzate (**CnnNN**) che raggiunge solamente il **0.678** di accuratezza, uno dei risultati più bassi ottenuti tra tutti i modelli. Questo è la dimostrazione che, sebbene la hand-normalizzazione sia una modifica fatta a mano adattando i valori in base a come si prestavano meglio con la rete (e quindi non una vera normalizzazione come la min-max), comunque è necessaria alla rete per poter comprendere meglio i dati e raggiungere una migliore accuratezza.

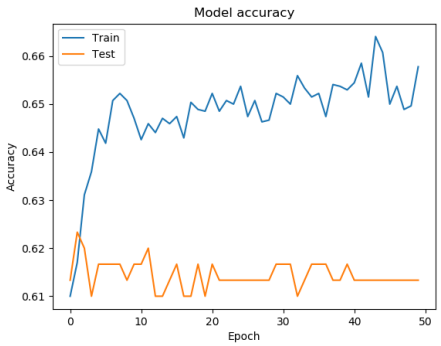
Come prova finale, è stato testato invece il modello dove viene eseguita una normalizzazione min-max standard delle features di ogni record tramite la funzione *normalize* della libreria *Sklearn* (**CnnSN**). Tale normalizzazione permette di ottenere un vettore con valori più coerenti tra loro e con le altre features calcolate dalla CNN, alle quali poi verranno aggiunti per l'ingresso nella MLP della convolutiva. Questa compatibilità tra features permette di ottenere un'accuratezza di **0.833**, di gran lunga superiore alle accuratezze dei precedenti due modelli.

Curve di convergenza per modello CNN multichannel + features MLP con batch_size=64

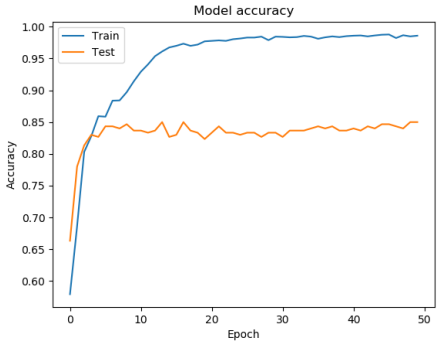
Hand Normalization (CnnHN)



No Normalization (CnnNN)



Sklearn Normalization (CnnSN)



Dalle curve di convergenza possiamo notare come cambia il comporta-

mento della stessa rete a seconda della normalizzazione applicata alla features aggiunte della MLP. I risultati peggiori riguardano il modello senza alcuna normalizzazione applicata, presentando un comportamento molto irregolare sia sul training che sul testing, sebbene ad ogni livello siano stati aggiunti i regolarittatori di kernel e bias: i dati non normalizzati non presentano alcun pattern calcolabile e/o utile alla rete, per questo il modello ottenuto non riesce a generalizzare ne i dati del training (che solitamente sono quelli su cui la rete si comporta meglio) ne del testing.

Un netto miglioramento è ottenuto dalla normalizzazione a mano del modello **CnnHN**. Anche se non si tratta di una vera normalizzazione, comunque sia queste modifiche forniscono alla rete un migliore contenuto informativo rispetto a features non normalizzate. Bisogna ricordare anche che essendo un multichannel le features vengono poi aggiunte insieme a quelle calcolate dalla parte convolutiva della CNN, per questo sicuramente tale normalizzazione permette di avere una migliore compatibilità tra i due tipi di informazioni. Dal grafico della stessa CnnHN è interessante notare come la curva di convergenza rispetto il testing (arancione) raggiunga una stabilità verso l'epoca 15, rimanendo costante per le epoche successive. L'accuratezza nel testing ancora rimane instabile.

Per finire, il modello **CnnSN** con la normalizzazione min-max è indubbiamente quello migliore tra i tre. Oltre a raggiungere un'accuratezza molto più altra rispetto agli altri due, le curve sia del testing che training assumono la forma classica di un modello ben allenato ed efficiente. Sebbene ci sia ancora presenza di overfitting, le curve si presentano abbastanza regolari.

5.3.2 Multichannel con tre canali input

Il modello multichannel è stato utilizzato anche nella costruzione di una CNN con tre diversi canali di input, permettendo analisi diverse per gli stessi dati da comprendere poi in un unica MLP finale per la classificazione. I tre diversi input corrispondono a tre flussi indipendenti tra loro:

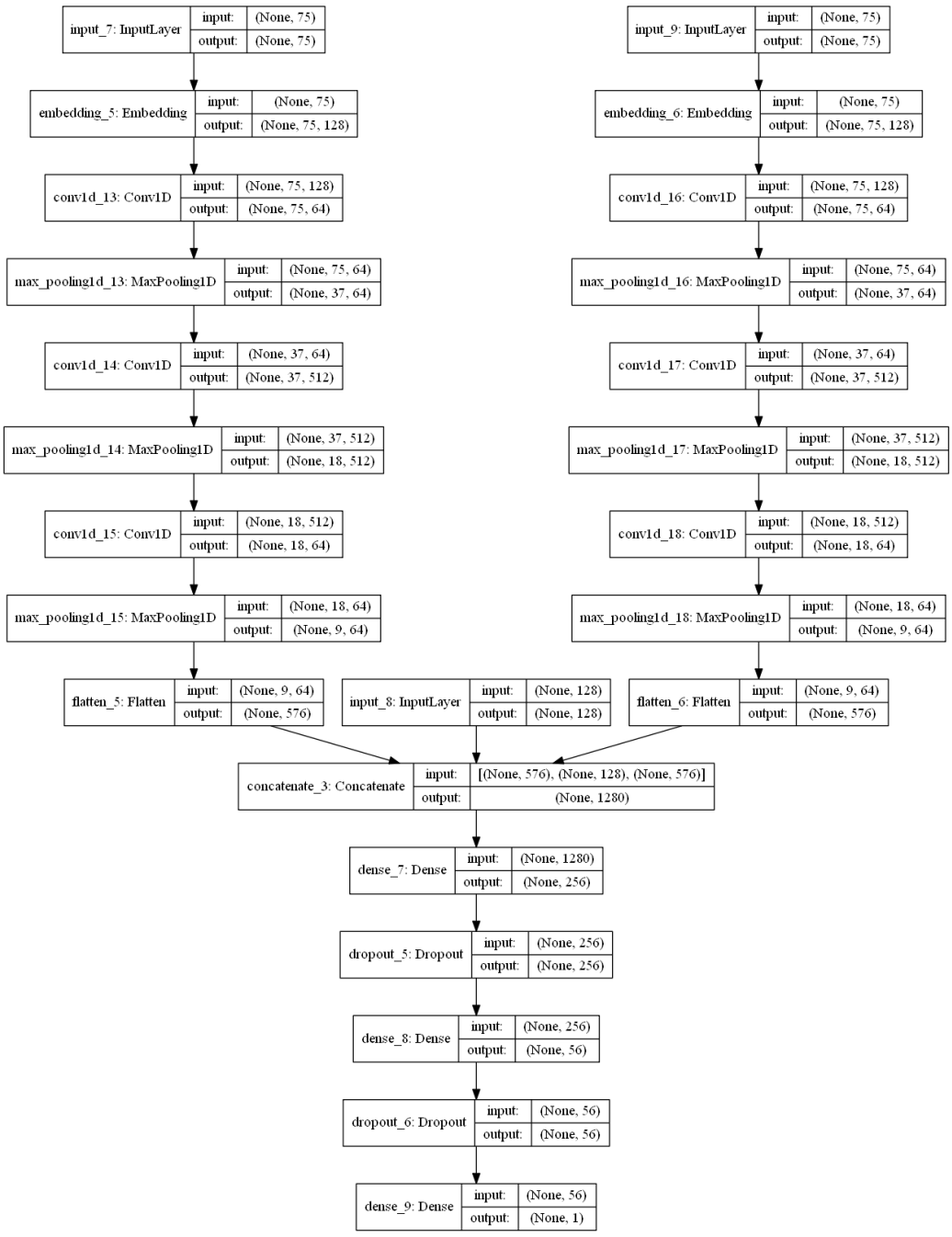
1. Sottorete con la stessa architettura del modello CNN sopra descritto con un kernel di *height=2*. Questa permette di analizzare i singoli record calcolando le informazioni spaziali tra 2 parole consecutive;

2. Sottorete con la stessa architettura del modello CNN sopra descritto con un kernel di *height=3*. Le informazioni spaziali sono relati-

ve a 3 parole consecutive, analizzate insieme durante l'operazione di convoluzione;

3. Sottorete che riceve in input la rappresentazione *Word2Vec* dell'intera frase. Mentre nei primi due esempi viene inviata la matrice dei pesi (ottenuta sempre con Word2Vec) al livello di Embedding con il quale verranno sostituite tutte le parole, in questo caso invece effettuiamo la sostituzione dell'intera frase, sommando tra loro i vettori delle parole che la compongono. Questa procedura alternativa di embedding della frase è spiegata nell'ultima parte della sezione 2.5. In questo caso non viene utilizzato un livello di Embedding perchè i dati devono essere passati alla rete tramite vettore e non matrice.

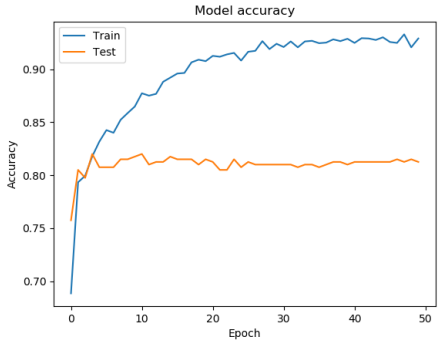
Per una migliore comprensione dell'architettura di questa rete a tre canali input viene presentata la figura seguente:


input_7: InputLayer
input: (None, 75)
output: (None, 75)
input_9: InputLayer
input: (None, 75)
output: (None, 75)
embedding_5: Embedding
input: (None, 75)
output: (None, 75, 128)
embedding_6: Embedding
input: (None, 75)
output: (None, 75, 128)
conv1d_13: Conv1D
input: (None, 75, 128)
output: (None, 75, 64)
conv1d_16: Conv1D
input: (None, 75, 128)
output: (None, 75, 64)
max_pooling1d_13: MaxPooling1D
input: (None, 75, 64)
output: (None, 37, 64)
max_pooling1d_16: MaxPooling1D
input: (None, 75, 64)
output: (None, 37, 64)
conv1d_14: Conv1D
input: (None, 37, 64)
output: (None, 37, 512)
conv1d_17: Conv1D
input: (None, 37, 64)
output: (None, 37, 512)
max_pooling1d_14: MaxPooling1D
input: (None, 37, 512)
output: (None, 18, 512)
max_pooling1d_17: MaxPooling1D
input: (None, 37, 512)
output: (None, 18, 512)
conv1d_15: Conv1D
input: (None, 18, 512)
output: (None, 18, 64)
conv1d_18: Conv1D
input: (None, 18, 512)
output: (None, 18, 64)
max_pooling1d_15: MaxPooling1D
input: (None, 18, 64)
output: (None, 9, 64)
max_pooling1d_18: MaxPooling1D
input: (None, 18, 64)
output: (None, 9, 64)
flatten_5: Flatten
input: (None, 9, 64)
output: (None, 576)
input_8: InputLayer
input: (None, 128)
output: (None, 128)
flatten_6: Flatten
input: (None, 9, 64)
output: (None, 576)
concatenate_3: Concatenate
input: [(None, 576), (None, 128), (None, 576)]
output: (None, 1280)
dense_7: Dense
input: (None, 1280)
output: (None, 256)
dropout_5: Dropout
input: (None, 256)
output: (None, 256)
dense_8: Dense
input: (None, 256)
output: (None, 56)
dropout_6: Dropout
input: (None, 56)
output: (None, 56)
dense_9: Dense
input: (None, 56)
output: (None, 1)

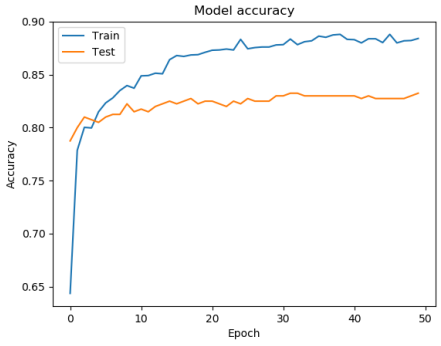
	64						128					
	5	10	15	20	25	30	5	10	15	20	25	30
MLP	0.815	0.819	0.813	0.816	0.81	0.816	0.811	0.809	0.819	0.819	0.818	0.811
CNN	0.78	0.788	0.784	0.789	0.781	0.775	0.805	0.793	0.811	0.799	0.8	0.813
CNNreg	0.836	0.831	0.828	0.82	0.812	0.823	0.831	0.816	0.838	0.81	0.822	0.82
CNNtot	0.837	0.839	0.842	0.842	0.839	0.836	0.845	0.845	0.835	0.838	0.841	0.844
CnnHN	0.7	0.696	0.689	0.688	0.688	0.688	0.701	0.753	0.763	0.769	0.769	0.766
CnnNN	0.637	0.666	0.659	0.659	0.657	0.663	0.647	0.671	0.677	0.676	0.678	0.678
CnnSN	0.832	0.833	0.827	0.831	0.82	0.828	0.806	0.803	0.82	0.821	0.828	0.826
CNN3inp	0.825	0.84	0.842	0.84	0.842	0.846	0.818	0.827	0.831	0.831	0.834	0.834

Curve di convergenza per modello CNN multichannel a 3 input

batch_size = 64



batch_size = 128



I risultati ottenuti mostrano che il modello multichannel con tre diverse analisi del testo raggiunge un'accuratezza pari a quella del modello single-channel. Sebbene si comportino in modo uguale, bisogna ricordare anche che presenta un'architettura decisamente più complessa rispetto al single-channel: nel single-channel abbiamo solamente un flusso di analisi dell'input, mentre nel multichannel ne abbiano tre diversi, richiedendo anche la traduzione delle singole frasi in vettori Word2Vec. Per questo, a parità di accuratezza, è consigliabile utilizzare comunque il modello single-channel.

5.4 Capsule Network

L'ultimo modello presentato nella sezione 4.4 è la *Capsule Network* di Hinton. Il modello implementato per il Natural Language Processing è quello base utilizzato per il database MNIST dallo stesso Hinton nell'articolo [5]: l'unica differenza è che mentre quest'ultimo utilizza un livello di Convoluzione 2D perchè lavora con immagini di cifre scritte a mano, in questa tesi è stato utilizzato un livello di Convoluzione 1D perchè è stato necessario adattare il modello non più all'image-processing ma al NLP.

Per questo anche qui troviamo un livello di Embedding al quale però non inviamo la matrice dei pesi calcolata con Word2Vec ma optiamo per un embedding di Keras. Nei vari allenamenti della rete infatti è stato notato che utilizzando un embedding Word2Vec la rete non riesce assolutamente ad imparare dai dati forniti, rimanendo ferma ad un valore di accuratezza pari a circa **0.50**. Invece, utilizzando l'embedding di Keras, i risultati sono decisamente migliori.

Appena dopo il livello di Embedding si trova il *Convolutional Layer*, il livello convolutivo caratterizzato dai seguenti parametri:

filters	kernel_size	strides	padding	activation
64	2	1	same	relu

Il successivo livello è il *Primary Capsule Layer*:

dim_vector	n_channels	kernel_size	strides	padding
8	32	9	2	same

dim_vector è la dimensione dell'activity-vector di ogni capsula; **n_channels** è il numero di capsule contenute nel relativo livello.

E infine abbiamo il livello di output *Output Capsule Layer*:

num_capsule	dim_vector	num_routing
2	16	3

num_capsule indica il numero delle classificazioni possibili per il task: nel paper di Hinton sono utilizzate in tutto dieci capsule-output corrispondenti alle dieci cifre del dataset MNIST, mentre nel progetto di questa tesi, avendo a che fare con una classificazione binaria, abbiamo solo due capsule; **dim_vector** specifica la dimensione di ogni capsula: nel nostro caso, ogni capsula-output avrà dimensione 16; **num_routing** è il numero di iterazioni del *routing-by-agreement*, solitamente 3.

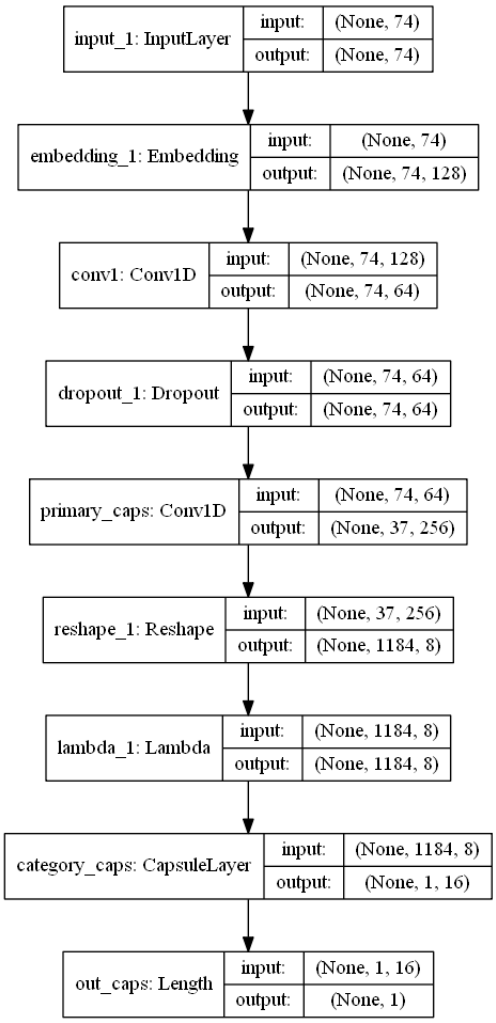


Figura 5.6: Architettura della CapsuleNet proposta da Hinton

Dello stesso modello base sono state presentate quattro varianti:

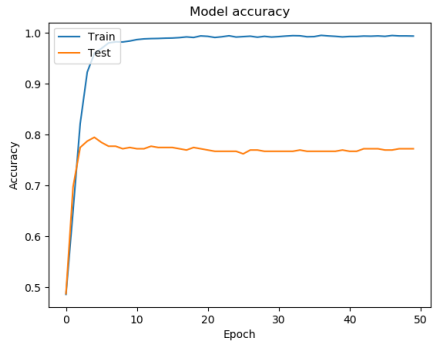
- 1 capsula di output e function-loss di Hinton presentata e descritta nella sezione 3.2.6 (**CapNet_1_H**);
- 1 capsula di output e function-loss crossentropy (**CapNet_1_CE**);
- 2 capsule di output e function-loss di Hinton (**CapNet_2_H**);
- 2 capsule di output e function-loss crossentropy (**CapNet_2_CE**).

	64						128					
	5	10	15	20	25	30	5	10	15	20	25	30
CapNet_1_H	0.81	0.798	0.798	0.801	0.798	0.798	0.812	0.805	0.801	0.8	0.8	0.798
CapNet_1_CE	0.778	0.772	0.772	0.769	0.77	0.768	0.785	0.775	0.773	0.768	0.768	0.768
CapNet_2_H	0.269	0.327	0.325	0.332	0.343	0.358	0.435	0.614	0.569	0.558	0.582	0.579
CapNet_2_CE	0.829	0.823	0.821	0.825	0.825	0.822	0.829	0.827	0.821	0.821	0.823	0.821

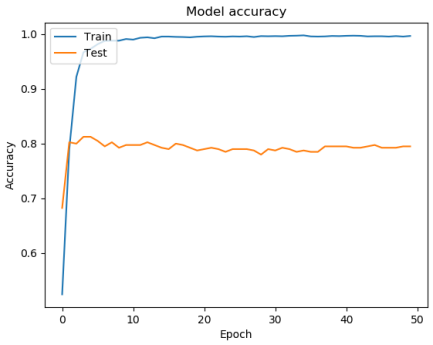
La rete che raggiunge l'accuratezza migliore è la **CapNet_2_CE**, ossia *Capsule Network base di Hinton* con *2 nodi output*, utilizzando la *crossentropy* come funzione di loss.

Curve di convergenza per modello Capsule Network con batch_size=64

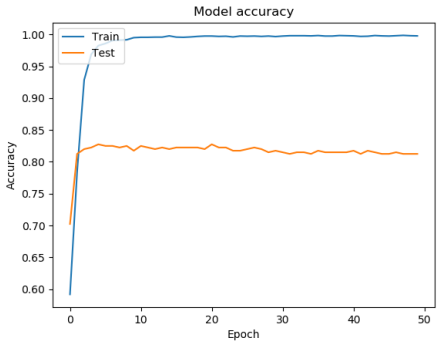
CapNet_1_CE



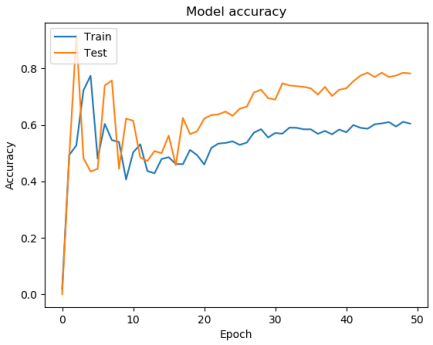
CapNet_1_H



CapNet_2_CE



CapNet_2_H



Tramite la *crossentropy* riusciamo ad ottenere un modello molto stabile a differenza della funzione di Hilton alla quale corrisponde una curva di convergenza nel caso di 1 output leggermente variabile ma che, aggiungendo 2 output, confonde completamente il modello, generando una curva disordinata e molto variabile.

Sia dai dati tabellari che dai grafici si può notare come la CapsuleNet riesca a raggiungere un valore alto di accuratezza già alle prime epoche dell'allenamento: in tre casi su quattro infatti, già nelle prime 5 epoche la curva

ha raggiunto un livello di accuratezza che il modello mantiene più o meno costante nelle epoche successive.

Capitolo 6

Conclusioni

I modelli che hanno ottenuto risultati soddisfacenti per il task *Hate Speech Detection* sono i seguenti:

- **MLP**: la Multilayer Perceptron impiegata nella competizione HaSpeeDe;
- **CNNtot**: rete convolutiva a tre livelli con *regolarizzatori (kernel e bias)* e *adaptive learning-rate*;
- **CnnSN**: rete convolutiva multichannel a due input: uno per le frasi e uno per le features calcolate nella competizione HaSpeeDe e normalizzate tramite *Sklearn*;
- **CNN3inp**: rete convolutiva multichannel a tre input: uno per l'analisi della frase con *kernel=2*; uno per l'analisi della frase con *kernel=3*; uno per l'inserimento della rappresentazione vettoriale dell'intera frase;
- **CapNet_2_CE**: modello base di rete a capsule con *due capsule di output* e *crossentropy* come funzione di loss.

	64						128					
	5	10	15	20	25	30	5	10	15	20	25	30
MLP	0.815	0.819	0.813	0.816	0.81	0.816	0.811	0.809	0.819	0.819	0.818	0.811
CNNtot	0.837	0.839	0.842	0.842	0.839	0.836	0.845	0.845	0.835	0.838	0.841	0.844
CnnSN	0.832	0.833	0.827	0.831	0.82	0.828	0.806	0.803	0.82	0.821	0.828	0.826
CNN3inp	0.825	0.84	0.842	0.84	0.842	0.846	0.818	0.827	0.831	0.831	0.834	0.834
CapNet_2_CE	0.829	0.823	0.821	0.825	0.825	0.822	0.829	0.827	0.821	0.821	0.823	0.821

	MLP	CNNtot	CnnSN	CNN3inp	CapNet_2_CE
accuracy	0.819	0.845	0.833	0.846	0.829
parametri	108 074	310 257	312 305	712 305	1 107 008
secondi	6	143	103	331	802

La differenza principale tra MLP e CNN è sicuramente l'estrazione delle features: mentre nella MLP è il programmatore che deve calcolarle e rappresentarle, nella CNN invece è la rete stessa (tramite l'operazione di convoluzione) che riesce a calcolare i pattern intrinsechi ai dati, ottenendo un contenuto informativo più completo e coerente.

Il modello **CNNtot** sicuramente è quello migliore sia in termini di accuratezza che di complessità. Grazie all'implementazione dei regolarizzatori kernel e bias siamo riusciti a limitare gli effetti dell'overfitting sebbene la grandezza del dataset fosse decisamente insufficiente (solo 4000 record) per ottenere un modello che generalizzasse adeguatamente i dati. Tramite l'adaptive learning-rate è stato possibile regolarizzare la convergenza della rete, stabilizzando l'accuratezza sia nel train che test, cosa che fino a quel momento (sia su MLP che altri modelli di CNN) si presentava molto instabile.
L'utilizzo di una CNN è sicuramente la scelta da fare qualora ci trovassimo di fronte ad un problema che nella MLP comporterebbe un eccessivo lavoro nel dataset, come preprocessing e successiva estrazione di features. Nelle reti convolutive infatti non è richiesto un minuziono preprocessing dei dati input, per non parlare dell'estrazione delle features, compito svolto direttamente alla rete stessa e da nessuno sforzo umano.

Sempre in tema di features abbiamo voluto testare se, oltre le informazioni calcolate dalla CNN (che solo lei sa leggere e interpretare), l'aggiunta delle features estratte nella MLP avesse potuto portare un ulteriore contenuto informativo utile a generalizzare meglio il dataset. Questo modello (*CnnSN*) si avvicina molto in performance alla *CNNtot*, ma non riesce a superarla. Inoltre, richiede anche una maggiore complessità della rete, con l'implementazione di un modello multichannel e l'inserimento nella rete di vettori di feature. Quindi, la scelta ricade sempre nella *CNNtot* sia per l'accuratezza che per la complessità nella costruzione della rete, risultando più pulita e semplice.

É stato testato anche un modello a tre input diversi (**CNN3inp**) per poter eseguire tre diverse e indipendenti analisi del dataset (kernel=2, ker-

nel=3, Word2Vec della frase) ottenendo dei risultati quasi equivalenti a quelli ottenuti nella *CNNtot*. Anche in questo caso comunque ci troviamo di fronte ad un modello che complica decisamente la struttura della rete e quindi l'allenamento, portando alla conclusione che per la rete *CNNtot* una semplice architettura e una singola analisi dei dati (kernel=2) è sufficiente per ottenere ottimi risultati a differenza di modelli più complicati.

Per finire abbiamo la **CapNet_2_CE**, il modello innovativo di Hinton. Sebbene sia il modello base presentato nell'articolo [5] e adattato al NLP, riesce comunque ad ottenere delle buone accuratezze anche se inferiori alla *CNNtot*. Le CapsNet sono un argomento ancora nuovo e in continuo aggiornamento, sicuramente pronte a superare le classiche convolutive: basti pensare come il modello base abbia quasi eguagliato la *CNNtot* a tre livelli convolutivi. Grazie al nuovo *routing-by-agreement* sostituito al *pooling* delle CNN, è possibile ottenere un molto più vasto contenuto informativo per i vari dati, come le varie relazioni spaziali tra diversi oggetti della stessa immagine, garantendo sia la *translation invariance* che *equivariance*. Queste qualità rendono le Capsule molto performanti anche su dataset piccoli rispetto alle CNN che, non riuscendo a garantire l'equivarianza, richiedono più immagini dello stesso oggetto con diverse rotazioni per poterlo riconoscere in tutte le sue varianti.

6.1 Lavori Futuri

Sicuramente le premesse per i lavori futuri riguardano soprattutto le Capsule Network, un modello di rete innovativo presentato nel 2017 e per questo oggetto di continuo sviluppo e aggiornamento. Gli esempi disponibili sono pochi e comunque riguardano maggiormente il modello base presentato da Hinton, senza proporre modelli alternativi e più complessi. Come mostrato dai risultati ottenuti in questa tesi, anche un modello semplice di Capsule Network riesce ad ottenere un buon livello di accuratezza eguagliando quelli più complessi di Reti Convolutive; per questo riuscire a costruire reti più complesse e più studiate potrebbe portare un'importante evoluzione nel Machine Learning.

Inerente al lavoro svolto in questa tesi, sarebbe interessante capire i motivi per i quali l'embedding Word2Vec risulta essere incompatibile con l'allenamento delle CapsuleNet, costringendo all'utilizzo di un embedding di Keras

per poter ottenere un accettabile livello di accuratezza.

Nella sezione 5.3.1 è stato testato un modello CNN multichannel con l'aggiunta delle features estratte e utilizzate nella competizione HaSpeeDe, inviate alla rete come vettore di features tramite un ulteriore canale di input. Oltre a questo metodo che richiede la creazione di un modello multichannel, ci sarebbe un altra tecnica per utilizzare delle features aggiuntive a quelle già calcolate dai livelli convolutivi: il metodo **out-of-the-box** si basa sull'utilizzo dei canali di un'immagine per aggiungere ulteriori informazioni al dato originale. Per ogni feature che vogliamo utilizzare, viene aggiunto un ulteriore canale all'immagine di partenza, con la stessa dimensione di quest'ultima. Il nuovo canale sarà composto di un solo valore pari alla relativa feature. Sarebbe curioso quindi sviluppare un modello che faccia utilizzo di quest'ultimo metodo e confrontare le performance con le reti presentate in questa tesi.

Bibliografia

[1] Giulio Bianchini, Lorenzo Ferri e Tommaso Giorni. "Text Analysis for Hate Speech Detection in Italian Messages on Twitter and Facebook." In: *EVALITA@ CLiC-it*. 2018.

[2] Fabio Del Vigna12 et al. "Hate me, hate me not: Hate speech detection on facebook". In: (2017).

[3] Elizabeth D Liddy. "Natural language processing". In: (2001).

[4] Fabio Poletto et al. "Hate speech annotation: Analysis of an italian Twitter corpus". In: *4th Italian Conference on Computational Linguistics, CLiC-it 2017*. Vol. 2006. CEUR-WS. 2017, pp. 1–6.

[5] Sara Sabour, Nicholas Frosst e Geoffrey E Hinton. "Dynamic routing between capsules". In: *Advances in neural information processing systems*. 2017, pp. 3856–3866.

[6] Manuela Sanguinetti et al. "An italian twitter corpus of hate speech against immigrants". In: *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*. 2018.

[7] Ziqi Zhang e Lei Luo. "Hate Speech Detection: A Solved Problem? The Challenging Case of Long Tail on Twitter". In: *CoRR* abs/1803.03662 (2018). arXiv: 1803.03662. URL: http://arxiv.org/abs/1803.03662.