UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in
Informatica

Elaborato finale

Uso di Machine Learning e Deep Learning per l'individuazione di parafrasi

Caso di Studio - Quora Question Pairs

Supervisore
Brunato Mauro

Laureando
Grigoletto Cesare

Anno accademico 2016/2017

Ringraziamenti

Ai miei amici e alla mia famiglia, perché mi hanno supportato e sopportato, nonostante tutto, permettendomi di arrivare fin qui

Indice

Sommario

Un problema comune a molti siti che si occupano di domande e risposte è l'individuazione di domande duplicate, cioè domande che sono già state poste, magari in una forma leggermente diversa, e che in molti casi hanno già ricevuto risposte esaustive. Attualmente molto spesso sono gli stessi utenti o moderatori a dover marcare le domande come duplicati, oppure i siti ricorrono a metodi per evitare direttamente che le domande duplicate siano poste, per esempio invitando gli utenti a effettuare una ricerca prima di porre la propria domanda.

Il caso di studio preso in esame è il sito Quora[18] che tramite il noto sito di competizioni di data science Kaggle ha indetto un concorso per migliorare il proprio sistema di riconoscimento dei duplicati; al momento della pubblicazione della competizione l'algoritmo in uso da parte del sito era un *Random Forest Classifier*.

Il problema da risolvere rientra nella categoria del NLP cioè *Natural Language Processing*, quella branca dell'informatica che si occupa di sviluppare algoritmi che permettano ai computer di interfacciarsi con il linguaggio umano. Esso è inoltre un esempio di applicazione dell'addestramento supervisionato, in quanto nel dataset di addestramento sono forniti degli esempi già etichettati, da usare come riferimento.

Nel corso della competizione sono state sperimentate varie tecniche di preprocessamento dei dati combinate poi con algoritmi di Machine Learning e Deep Learning. Come vedremo in seguito, il Deep Learning si è dimostrato superiore al Machine Learning classico nello svolgere questo compito.

Il più grosso ostacolo in questo tipo di studi è il fattore tempo, strettamente legato alla potenza di calcolo disponibile: nel caso del Machine Learning è il preprocessing dei dati che necessita di parecchio tempo, nel caso del Deep Learning è invece l'addestramento del modello. Infatti modelli relativamente complessi eseguiti su di un hardware di medio livello hanno tempi di addestramento nell'ordine dei giorni.

Le basi di questo studio sono da ricercarsi nel lavoro svolto per quanto riguarda il parsing delle frasi e la loro rappresentazione sotto forma di vettori n-dimensionali. Questo infatti ha permesso di rappresentare le frasi in modo facilmente interpretabile da una macchina. Tra i maggiori contributori in questo campo troviamo il gruppo dell'università di Stanford che si occupa appunto del Natural Language Processing.[16]

All'interno di questo studio ho scelto di evitare il confronto diretto, almeno inizialmente, con lo stato dell'arte per l'individuazione di parafrasi, ritenendo fosse più proficuo sviluppare autonomamente dei modelli basandomi su ricerche e articoli che trattano più in generale del Natural Language Processing e di sue applicazioni che non siano specificamente l'individuazione di parafrasi. Tale scelta è stata guidata dalla consapevolezza che, con le conoscenze in mio possesso non avrei potuto prendere un modello sviluppato da eminenti studiosi, espressamente per risolvere questo tipo di problemi, e avere la presunzione di poterlo migliorare. Conseguentemente ho scelto un percorso che fosse di crescita personale e che mi permettesse di sperimentare vari approcci al problema, senza essere condizionato eccessivamente da risultati precedenti ottenuti da altri.

Non avendo mai avuto alcuna esperienza nell'uso del Deep Learning o del Natural Language Processing, una prima parte della ricerca è stata dedicata al prendere confidenza con i principali concetti e metodologie relative a tali ambiti. Nello specifico si sono rivelate estremamente utili, le

lezioni tenute dai docenti Chris Manning e Richard Socher dell'università di Stanford[15]

1 Premesse

In questo capitolo vedremo come si configura la sfida proposta dal sito *Kaggle*, con che tipo di dati si andrà a lavorare, quali sono gli obiettivi da raggiungere e il metro di valutazione.

1.1 Il dataset

Il dataset fornito per la competizione è costituito da una parte di training composta da 404290 elementi e da una di test composta da 2345796 elementi; ogni riga del dataset di training contiene sei campi[11]:

- Id: l'id che identifica univocamente la coppia di domande
- Quid1: l'id che identifica univocamente la prima domanda
- Quid2: l'id che identifica univocamente la seconda domanda
- Question1: il testo completo della prima domanda
- Question2: il testo completo della seconda domanda
- Is_duplicate: valore booleano che etichetta la coppia come duplicati (1) o meno (0), che sarà poi il bersaglio della previsione

Il testo delle domande risulta essere in inglese e inoltre l'attribuzione delle etichette risulta essere non accurata al 100% come affermato sulla pagina della competizione Ogni riga del dataset di test è invece composta da soli tre campi:

- Id: l'id che identifica univocamente la coppia di domande
- Question1: il testo completo della prima domanda
- Question2: il testo completo della seconda domanda

Di norma quando si affronta un problema di classificazione si cerca di ottenere dei dataset che siano quanto più possibile uniformi, nello specifico si fa in modo che la percentuale delle classi all'interno dei due dataset sia pressoché la stessa, così da addestrare il modello su un dataset quanto più simile alla situazione di test. In questa competizione tuttavia non ci è dato sapere la distribuzione delle classi all'interno del dataset di test, quindi l'unica cosa di cui ci si può assicurare è che tale distribuzione sia uguale all'interno del dataset di addestramento e di quello di validazione.

1.2 Obiettivo della competizione e metro di valutazione

L'obiettivo della competizione consiste nello stabilire, data una coppia di domande, se si tratti di duplicati oppure no, in sostanza se una sia la parafrasi dell'altra. Viene richiesto di fornire una probabilità che le domande siano effettivamente duplicati, sotto forma di numero reale tra 0 e 1, quindi il problema si configura come un problema di classificazione binaria dove l'output dei modelli da sottoporre al sito per la valutazione finale, sarà la probabilità che la classe a cui appartiene la coppia sia "1". L'accuratezza della previsione sarà valutata usando come metrica la "log loss" [8] dove l'accuratezza di ciascuna previsione è data da:

$$-(y \log(p) + (1 - y) \log(1 - p)) \tag{1.1}$$

Dove:

- y è il valore reale dell'etichetta

- p è il valore predetto dal modello

Per questa ragione, indipendentemente dal modello utilizzato, sarà sempre scelta la *Log Loss* come parametro di riferimento per la valutazione dell'accuratezza degli algoritmi

2 Machine Learning

L'apprendimento automatico (anche chiamato machine learning dall'inglese) rappresenta un insieme di metodi sviluppati a partire dagli ultimi decenni del '900 in varie comunità scientifiche, i quali forniscono ai computer l'abilità di apprendere senza essere stati esplicitamente programmati. Un algoritmo di machine learning sfrutta un insieme di esempi di addestramento per realizzare un sistema che sia in grado di generalizzare quanto appreso anche per nuovi elementi, purchè essi appartengano al contesto su cui è stato addestrato. [4]

2.1 Estrazione delle Features

Al primo approccio con la competizione la prima cosa di cui ci si rende conto è che le frasi, scritte in linguaggio naturale, non sono utilizzabili direttamente come input per un algoritmo di classificazione per vari motivi:

- Le frasi sono di lunghezza variabile, già questo impedisce di usarle per addestrare un classificatore
- Lo spazio delle frasi con cui ci si trova a lavorare è troppo grande e sparso per poter progettare un classificatore efficiente, anche partendo dal presupposto di avere tutte frasi di uguale lunghezza
- La rappresentazioni delle parole e conseguente delle frasi è basata sui singoli caratteri codificati in ascii o UTF-8, tale rappresentazione non tiene minimamente conto del significato, quindi della semantica delle parole e frasi analizzate; per esempio da un punto di vista puramente rappresentativo "cado" e "dado" sono parole molto simili, differiscono infatti per una sola lettera (nel codice ASCII per esempio la lettera c corrisponde al numero 99 e la lettera d al numero 100) La rappresentazione delle due parole in codice binario sarebbe quindi

  - *cado*: 01100011 01100001 01100100 01101111
  - *dado*: 01100100 01100001 01100100 01101111

  Un computer potrebbe quindi classificare le parole come simili, nonostante semanticamente non lo siano affatto.

I classificatori lavorano infatti con numeri, in buona sostanza applicano una funzione ai dati di input, cercando di ottenere il corretto valore di output. In questo caso però, trattare le parole (e quindi le frasi) come array di valori ascii (o qualunque altra codifica), non cattura nessuna carattestica della frase stessa che possa essere utile alla classificazione.

Per questo è necessario effettuare delle operazioni su di esse per renderle utilizzabili da un classificatore, quindi estrarre un egual numero di feature significative da ciascuna coppia: questa fase è detta preprocessing dei dati. In questo caso di studio si è scelto di estrarre varie feature relative sia alla struttura della frase che alla sua semantica.

2.1.1 Features di base

Come prima cosa si estraggono le informazioni riguardanti la lunghezza delle frasi, in particolar modo il numero di parole e il numero di caratteri contenuti in ciascuna di esse, presupponendo che parole di lunghezza molto diversa, difficilmente abbiano lo stesso significato. Al termine di questa fase si hanno a disposizione due feature per ogni frase: il numero di parole e il numero di caratteri contenuti in essa, per un totale di quattro feature per coppia.

Successivamente ho provato a ridurre queste quattro feature a due, considerando la differenza tra il numero di caratteri nelle due frasi, e la differenza tra il numero di parole nelle due frasi, la precisione dell'algoritmo è tuttavia leggermente peggiorata, quindi sono tornato sui miei passi

2.1.2 LCS

Un approccio simile porta alla scelta come parametro della LCS[13], la Longest Common Subsequence tra le due frasi, cioè la lunghezza della sottosequenza di parole più lunga che le due frasi hanno in comune. Per agevolare il calcolo ho prima convertito le frasi in liste di interi associando a ciascuna parola un id univoco, per poi applicare l'algoritmo per l'individuazione della LCS ai vettori così ottenuti

2.1.3 Utilizzo della libreria Spacy

Il passaggio successivo è stato rappresentare le parole in uno spazio vettoriale in modo da poter utilizzare concetti quali la distanza tra esse come metro di valutazione per la similitudine. Esistono varie librerie che permettono di fare ciò, ho trovato la libreria Spacy più efficiente soprattutto dal punto di vista del consumo di memoria e dei tempi di computazione. [22]
Appoggiandomi ad essa ho estrapolato un indice di similarità che consiste nella somma dei valori di similarità delle coppie $(w1, w2)$ dove $w1$ appartiene alla prima domanda e $w2$ alla seconda, inizialmente avevo pensato di normalizzare la similarità rispetto al numero di coppie considerate o alla lunghezza delle frasi prese in esame, ma il valore non normalizzato ha portato a risultati migliori rispetto a quelli normalizzati
Questo parametro è stato poi raffinato calcolando la frequenza di apparizione di ciascuna parola e ignorando il contributo delle parole considerate "comuni", cioè con una frequenza troppo alta.
Proseguendo sul concetto di parole comuni ho poi definito le parole rare, cioè con una frequenza bassa, e definito un ulteriore indice di similarità che prendesse in considerazione solo le parole rare, anche in questo caso, normalizzare il valore riduce la precisione della previsione.
Infine ho applicato nuovamente la funzione di similarità, però invece di applicarla alle coppie di parole, l'ho utilizzata direttamente sulle due frasi rappresentate ciascuna come un unico vettore, ottenendo un terzo parametro di similitudine.

2.1.4 Cosine Similarity - Tf-Idf

Un altro utile parametro per la similarità tra frasi è la Cosine similarity: una tecnica euristica per la misurazione della similitudine tra due vettori effettuata calcolando il coseno tra di loro. Per poterla applicare è quindi necessario convertire ciascuna frase in una qualche rappresentazione vettoriale per poi calcolare la Cosine Similarity tra i due vettori ottenuti. Si noti che anche il terzo parametro ottenuto in 2.1.3 è una cosine similarity calcolata tra le rappresentazioni vettoriali delle due frasi prodotte da *Spacy*

In questo caso specifico ho convertito le frasi in vettori di $tf - idf$ [23]. Il $tf - idf$, *Term Frequency - Inverse Document Frequency* è un valore attribuito a ciascuna parola all'interno di un corpus, che stabilisce la sua "importanza", tale valore cresce in maniera proporzionale al numero di volte che una parola compare all'interno del documento considerato (in questo caso una frase) ma è normalizzato rispetto alla frequenza della parola all'interno dell'intero corpus (in questo caso la coppia di frasi).

La formula per calcolare il valore del $tf - idf$ per una parola t è la seguente:

$$TF(t) = (p/n) \tag{2.1}$$

$$IDF(t) = \log d/d_t \tag{2.2}$$

$$TFIDF(t) = TF(t) * IDF(t) \tag{2.3}$$

dove:

- p: numero di volte che la parola appare nella frase considerata
- n: numero di parole all'interno della frase considerata

- d: numero totale di frasi (in questo caso 2)
- d_t: numero di frasi in cui compare t

2.1.5 Ripulire le frasi

Come ulteriore migliormente all'algoritmo avevo pensato di ripulire il dataset, prima rimuovendo solamente la punteggiatura, successivamente rimuovendo tutte le cosiddette *Stopwords* cioè tutte quelle parole estremamente comuni come articoli e congiunzioni che potrebbero portare rumore nel dataset. Tuttiavia entrambe queste operazioni di preprocessing hanno peggiorato i risultati rispetto all'estrazione delle feature dal dataset non ripulito. Comunque è interessante come l'idea delle *stopwords* sia in qualche modo incorporata nel concetto di parole comuni e rare esposto in 2.1.3

2.1.6 Riassunto del preprocessing

Al termine del preprocessing sono state raccolte le seguenti feature

- *Feature di base*: numero di parole contenute nella prima frase, numero di caratteri contenuti nella prima frase, numero di parole contenute nella seconda frase, numero di caratteri contenuti nella seconda frase
- *LCS*: Lunghezza della sottosequenza comune massimale tra le due frasi
- *Feature Relative alla libreria Spacy*: similarità tra tutte le parole delle due frasi escluse le comuni, similarità tra le parole rare delle due frasi, similarità tra le due frasi considerate come un unico vettore ciascuna
- *Cosine Similarity*: Similarità del coseno calcolata tra le due frasi convertite in vettori $tf - idf$

Vediamo come l'aggiunta di nuove feature ha migliorato la capacità predittiva dell'algoritmo:

Feature	**log_loss**
Feature Base	0.6114
LCS	0.5238
Parole comuni/rare	0.4961
Spacy Sentence Similarity	0.4836
Cosine Similarity tf-idf	0.4648

Di seguito vediamo degli istogrammi che mostrano chiaramente il miglioramento progressivo delle prevesioni: sull'asse delle ascisse di ciascun grafico troviamo la probabilità che le frasi della coppia siano effettivamente duplicate e sull'asse delle ordinate vediamo il numero (normalizzato) di frasi che hanno ottenuto quel valore. In blu sono segnate le coppie non duplicate, quelle in arancione sono invece realmente duplicate.

Notiamo come l'algoritmo diventa progressivamente più preciso nell'individuare le frasi non duplicate, mentre ha difficoltà a classificare con precisione i duplicati

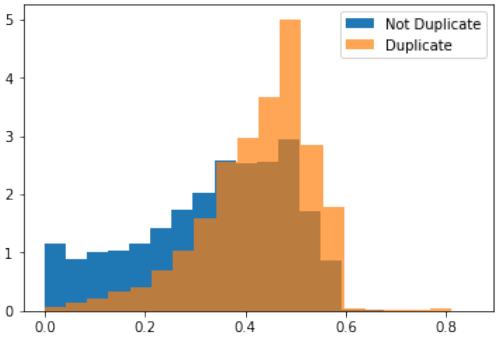

Figura 2.1: Feature di base

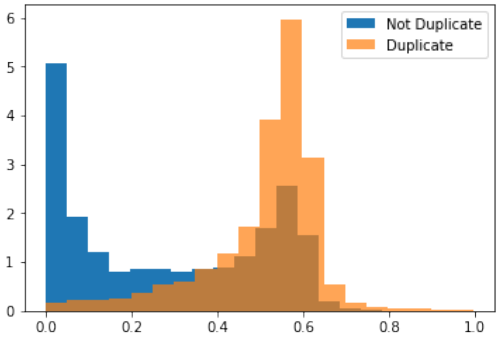

Figura 2.2: LCS

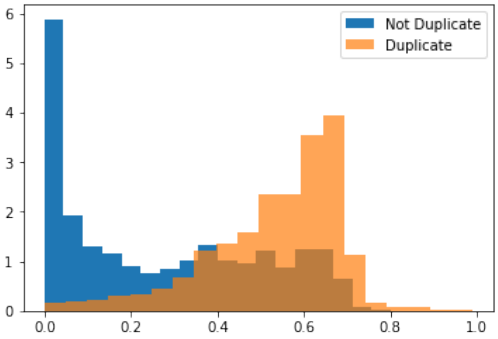

Figura 2.3: Parole Comuni/Rare

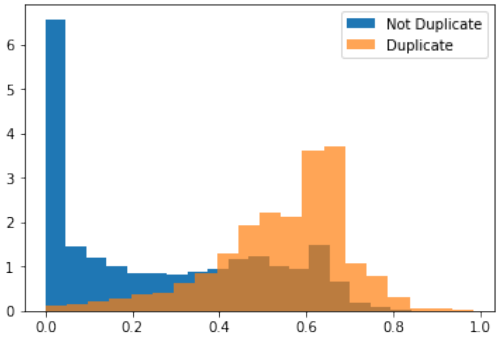

Figura 2.4: Spacy Similarity

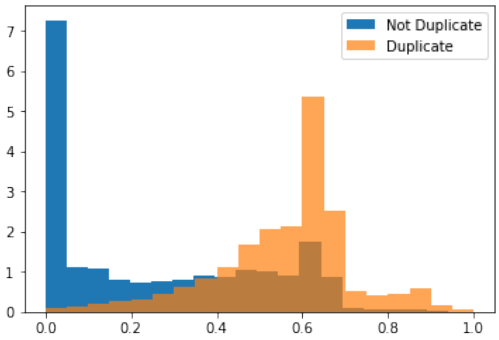

Figura 2.5: Cosine Similarity - tfidf

2.2 L'Algoritmo

Per quanto riguarda l'algortmo, mi sono appoggiato alla libreria Sklearn [21], che permette di istanziare i principali modelli di machine learning classico rapidamente, pur permettendo un buon grado di personalizzazione

In tutte le esecuzioni degli algoritmi ho utilizzato l'80% dei dati per l'addestramento e il 20% per la validazione

2.2.1 Random Forest

Ho voluto testare l'algoritmo utilizzato dallo stesso sito Quora, cioè il Random Forest Classifier, esso crea un insieme di alberi di decisione a partire da sottoinsiemi del dataset di addestramento, scelti casualmente. In fase di test si considerano le predizioni di ogni singolo albero, che vengono poi aggregate per formare la predizione definitiva, per esempio si sceglie a maggioranza.

I più importanti parametri messi a disposizione dalla libreria per generare il classificatore sono i seguenti:

- Il criterio da utilizzare per misurare la qualità di uno split: [*Gini*, *Entropia o information gain*]
- La massima profondità raggiungibile da un albero (se viene selezionato *None* l'algoritmo terminerà quando tutte le foglie saranno pure: [*None*, *numero intero*]
- Il numero massimo di feature da utilizzare per selezionare il miglior split (sqrt(numero di features) o log2(numero di features): [*auto*, *log2*]

Ho utilizzato un algoritmo GridSearch per trovare la combinazione di parametri che restituisce il miglior risultato

mean_test_score	**mean_train_score**	**criterion**	**max_depth**	**max_features**
0.4689	0.4232	gini	13	log2

2.2.2 Confronto con gli altri principali algoritmi

Per completezza ho voluto verificare i risultati che si ottengono, partendo dalle stesse feature ma utilizzando algoritmi di machine learning diversi:

algoritmo	**log_loss**
Naive Bayes Gaussian	1.2554
Quadratic Discriminant	1.0661
Support Vector Classifier	11.3197

Si tenga presente che il risultato relativo al $SupportVectorClassifier$ è stato ottenuto con un $LinearSVC$ che non converge nemmeno dopo 10000 iterazioni (lo standard della libreria sono 1000), l'uso di un $C - SVC$ è invece non raccomandabile in questo caso perchè il tempo di esecuzione è più che quadratico rispetto al numero di esempi

Come si può notare il *Random Forest Classifier* ottiene risultati decisamente migliori rispetto agli altri algoritmi.

2.3 Multithreading

Nelle ultime fasi di sviluppo il preprocessamento è divenuto estremamente costoso in termini di tempo, per questo ho scirtto tutte le funzioni che andavano ad operare sul dataset in modo che sfuttassero il multithreading; avendo quattro core a disposizione nella macchina utilizzata, questo miglioramento ha permesso di ridurre approssimativamente ad un quarto il tempo necessario a processare l'intero dataset, cosa che si è rivelata estremamente utile soprattutto in fase di previsione, quando si è reso necessario effettuare le stesse operazioni sul dataset di test, che è circa cinque volte più grande rispetto a quello di training.

2.4 Considerazioni

Stabilito che il *Random Forest Classifier* sia l'algoritmo di apprendimento migliore da usare in questo caso, il passo successivo sarebbe individuare feature migliori su cui addestrare il classificatore, tuttavia così potremmo davvero parlare di apprendimento? Il miglioramento delle previsioni non sarebbe dovuto alla macchina che apprende di più o in maniera migliore, bensì al mio studio più approfondito di questo problema specifico che mi permetterebbe di estrarre features migliori.

Per questo nel capitolo successivo ho sperimentato tecniche di apprendimento automatico che simulano il funzionamento del cervello umano e che danno la sensazione che la macchina stia effettivamente imparando.

3 Deep Learning

Il Deep Learning è un particolare tipo di Machine Learning che ottiene grande capacità e flessibilità imparando a rappresentare il mondo come una gerarchia nidificata di concetti, dove ogni concetto è definito in relazione a concetti più semplici, e rappresentazioni più astratte sono definite in termini di quelle più semplici

Componente chiave del Deep Learning sono le reti neurali, sistemi di calcolo ispirati al funzionamento del cervello negli esseri viventi: ciascuna rete è basata su un insieme di nodi chiamati neuroni artificiali collegati tra loro in modo simile ai neuroni collegati dalle sinapsi nel cervello. Ogni neurone può emettere un segnale che sarà ricevuto dai neuroni dello strato successivo, ad esso collegati.

Rispetto al semplice Machine Learning il Deep Learning richiede una quantità maggiore di dati su cui addestrare il modello e una potenza di calcolo superiore, tuttavia la capacità predittiva di un algoritmo di Deep Learning risulta solitamente superiore, nel caso in cui tali dati siano disponibili.

Un altro vantaggio fondamentale risiede nella capacità degli algoritmi di imparare autonomamente features complesse, riducendo così il lavoro necessario da parte dello sviluppatore nella fase di estrazione delle features. Se infatti un modello di Machine Learning classico migliora quando il suo sviluppatore "impara" una nuova feature rilevante e la fornisce al modello, nel Deep Learning è il modello stesso che impara e il ruolo dello sviluppatore sta nel progettare la Rete Neurale, cioè il "cervello" che più si adatta a risolvere quel particolare problema

3.1 Multi-Layer Perceptron

Inizialmente ho utilizzato i dati preprocessati durante la fase precedente a cui ho applicato un classificatore basato su di un *Multi Layer Perceptron*, un'architettura di Deep Learning composta da una serie di strati textitfully-connected.

3.1.1 MLPClassifier Sci-kit Learn

Per questo modello mi sono appoggiato nuovamente alla libreria *Sci-kit learn*, che di default genera un'architettura composta da un layer di input, un layer denso contente 100 nodi, e un layer di output.

Per la scelta dell'ottimizzatore e della funzione di attivazione ho optato per l'utilizzo di una GridSearch che variasse i parametri sopracitati tra i seguenti:

- Funzioni di attivazione: [*identity*, *logistic*, *tanh*, *relu*]
- Ottimizzatore: [*lbfgs*, *sgd*, *adam*]

il risultato migliore è stato il seguente

mean_test_score	**mean_train_score**	**activation**	**solver**
0.4801	0.4788	logistic	adam

Successivamente ho rieseguito una GridSearch per cercare di trovare i migliori iperparametri per il modello, l'output della ricerca è stato il seguente

mean_test_score	**mean_train_score**	**alpha**	**batch_size**	**beta**	**epsilon**	**learning_rate_init**
0.4784	0.4767	0.0001	100	0.9	2e-09	0.0005

come possiamo notare, una corretta selezione degli iperparametri riduce la log_loss sul validation set di 0.0017, un valore abbastanza trascurabile

3.1.2 MLP Keras

Da qui in avanti ho abbandonato *scikit-Learn* in favore della libreria *Keras*[6], a sua volta sviluppata sopra alla libreria *Tensorflow*[1] sviluppata da Google.

Ho progettato un Multi Layer Perceptron composto da tre livelli con 200 neuroni ciascuno e ho assegnato ai primi due una funzione di attivazione *f1* e all'ultimo una funzione di attivazione *f2*, in quanto ho notato che, in molti modelli presentati negli articoli o in rete, l'ultimo layer ha una funzione di attivazione diversa dagli altri, che spesso si adatta al tipo di previsione su cui si sta lavorando; ho comunque fatto variare le due funzioni considerando anche il caso in cui tutti gli strati abbiano la stessa funzione di attivazione, in modo da verificare la mia teoria. L'ottimizzatore è Adam e la funzione di perdita rimane Log Loss come nel modello di *scikit Learn*

Ho provato tutte le combinazioni che si ottengono assegnando a f1 e f2 i seguenti valori: [*sigmoid, elu, relu, softmax, softplus, tanh, hard_sigmoid*]

Le migliori combinazioni si sono rivelate essere le seguenti:

f1	f2	Log_Loss	Accuracy
elu	softmax	0.4844	0.7369
relu	sigmoid	0.4857	0.7374
softplus	softmax	0.4879	0.7385

Il risultato conferma la mia idea secondo cui le prestazioni migliorano se il layer di output possiede una funzione di attivazione diversa da quella degli altri Come visto, le tre combinazioni funzioni di attivazione per i layer e il layer di output ottengono risultati simili. Nelle architetture successive ho optato per la combinazione *Relu / Sigmoid* perchè è attualmente la scelta più utilizzata e consolidata.

3.1.3 Aumento della profondità della rete

Vediamo come variano le performance della rete aumentando la profondità e variando il numero di neuroni presenti su ogni livello

Numero di Layer	Neuroni per layer	Log_Loss	Accuracy
10	200	0.4740	0.7466
10	400	0.5036	0.7225
15	200	0.4931	0.7306

Possiamo notare come, dato il ridotto numero di features in input reti troppo complesse perdano di efficacia.

3.1.4 Risultati

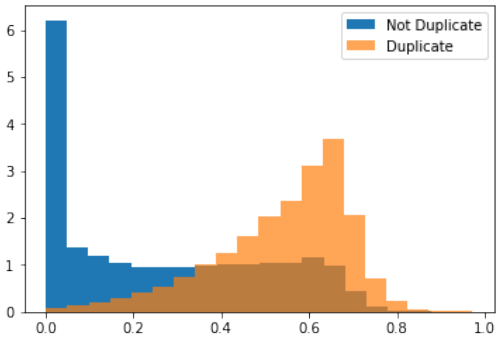

Figura 3.1: Risultati del Multilayer Perceptron

Confrontando i risultati notiamo che un tipo di architettura *Fully Connected* ottiene risultati simili a quelli ottenuti dal *Random Forest Classifier*, del resto lo scopo del Deep Learning è che sia la macchina a imparare autonomamente le feature, d'ora in poi sarà quindi necessario fornire all'algoritmo un input ancora scarsamente elaborato, ma che lui sia in grado di interpretare.

3.2 Matrice di similarità

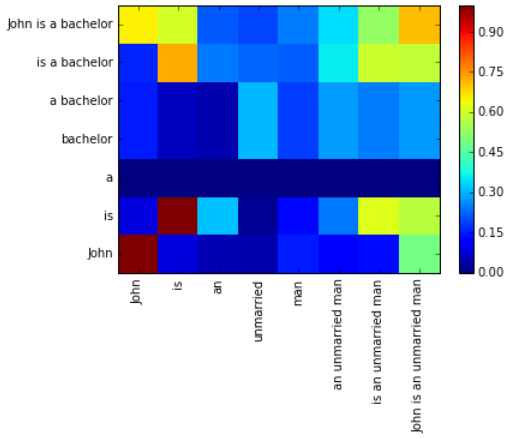

Figura 3.2: Esempio di Matrice di similarità

Prendendo spunto dalla pubblicazione [19] ho pensato di sfruttare il concetto di Matrice di similarità per individuare i duplicati. L'idea è la seguente, per ogni coppia di domande viene generata una matrice N x M dove N è la lunghezza della prima frase e M la lunghezza della seconda, tale matrice avrà in posizione i, j la similarità tra l'i-esima parola della prima frase e la j-esima parola della seconda, calcolata grazie alla libreria *Spacy* Le matrici saranno poi ridimensionate in modo da risultare di dimensione P x P ed essere facilmente sottoponibili ad una rete neurale.

Il valore di P è stato individuato a seguito di varie prove, i cui migliori risultati sono i seguenti:

P	Log_loss
10	0.5536
20	0.5170
30	0.5284

3.2.1 Architettura

L'architettura progettata in questo caso si compone di tre blocchi convoluzionali 2D con funzione di pooling *Max* e funzione di attivazione $ReLU$. Sopra di essi due livelli fully-connected con attivazione $ReLU$ e un livello di output con attivazione sigmoide

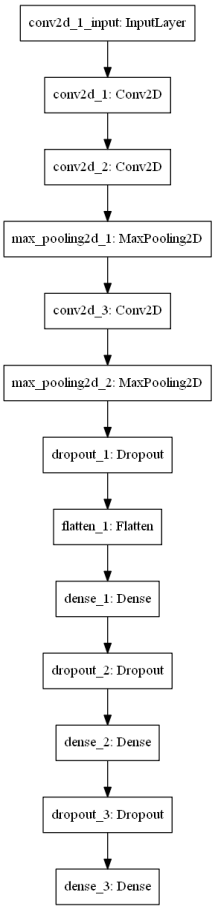

Figura 3.3: Architettura

I risultati ottenuti con questa architettura sono i seguenti:

Log_loss	**Accuracy**
0.5046	0.7384

Da notare che usando come funzione di pooling *Min* oppure *Mean* i valori sono pressoché identici.

3.2.2 Considerazioni

I risultati sono buoni, ma inferiori ai risultati ottenuti con metodi classici. Questo è probabilmente dovuto al mio desiderio di sperimentare, sostituendo il Parser Sviluppato da Stanford e la rappresentazione ottenuta tramite *Recursive AutoEncoder* utilizzati nel paper con il parser e la rappresentazione vettoriale della libreria Spacy. I tempi di preprocessamento e successivo addestramento del modello risultano enormi, ho quindi deciso di accantonare questo metodo in favore di altri, conscio che probabilmente per ottenere il massimo da esso avrei finito con il copiare l'architettura proposta e ciò sarebbe stato tutt'altro che interessante.

3.3 Reti neurali siamesi

Il problema preso in esame consiste nel valutare la similarità tra due frasi, per questo è importante che all'interno della rete il modo in cui le frasi vengono processate risulti equivalente in modo che la rappresentazione interna alla rete di due frasi simili sia effettivamente simile[14]. Per ottenere ciò, all'interno dei modelli presentati da qui in avanti saranno presenti due rami siamesi, che riceveranno in input una frase ciascuno. Tali sotto-reti sono dette siamesi perché sono assolutamente identiche, hanno la stessa configurazione, stessi parametri e stessi pesi, inoltre in fase di addestramento l'aggiornamento dei parametri avviene in maniera uguale per entrambe le reti

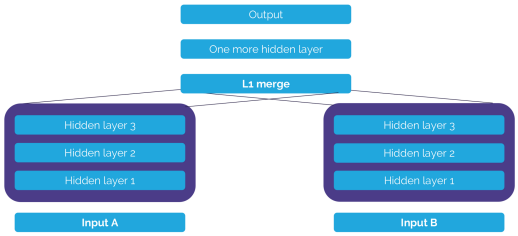

Figura 3.4: Esempio di architettura siamese

Come si nota dalla figura 3.3 i layer utilizzati nelle due componenti siamesi, sono di fatto li stessi, in pratica in un'architettura siamese, lo stesso layer viene utilizzato in entrambe le sezioni siamesi. Per il resto l'addestramento si svolge come per una rete neurale standard.

3.4 Rappresentazioni dei dati

In un algoritmo di Deep Learning è fondamentale scegliere un buon modo di rappresentare i dati in modo che il modello sia in grado di estrarre autonomamente le features più utili e contemporaneamente sia in grado di generalizzare il più possibile. Visto il risultato non ottimo ottenuto dalla matrice di similarità, ho optato per un preprocessamento dei dati ancora più ridotto, adottando il concetto di *Word Vector* e *Sentence Vector* in cui le parole vengono rappresentate come vettori in uno spazio vettoriale n-dimensionale; conseguentemente le frasi vengono rappresentate a loro volta come vettori o matrici.

Nel caso preso in esame sono state sperimentate due diverse rappresentazioni che sfruttano questo principio, queste rappresentazioni sono combinate con vari modelli che sfruttano la struttura di rete siamese presentata in precedenza, in modo che ciascun ramo siamese prenda in input la rappresentazione di una delle frasi della coppia

3.4.1 Spacy Word Vectors

Il primo metodo di rappresentazione è quello utilizzato dalla libreria *Spacy*, già utilizzata in precedenza con l'algoritmo di Machine Learning classico 2.1.3. In questo caso si sfrutta la rappresentazione vettoriale delle frasi fornite da Spacy che viene ottenuta facendo la media dei vettori che rappresentano le singole parole che compongono la frase.

3.4.1.1 Rete convoluzionale 1D

Le reti convoluzionali funzionano bene con le immagini perchè sono in grado di estrarre features da porzioni di esse astraendo ad ogni convoluzione e individuando relazioni tra le parti dell'immagine, Lo stesso principio può essere applicato alle frasi, in quanto è possibile considerare le frasi rappresentate in uno spazio vettoriale, come matrici (immagini) di altezza 1, in questo modo la convoluzione opera su componenti vicine della rappresentazione vettoriale delle frasi. In una rete convoluzionale il blocco base di costruzione della rete si compone tipicamente di quattro elementi principali:

- layer convoluzionale
- layer di attivazione
- layer di pooling
- eventuale layer di *Dropout*

L'architettura proposta si compone di due sottoreti siamesi composte ciascuna da tre blocchi convoluzionali, in cima ad esse due livelli fully-connected con attivazione ReLU e infine il layer di output con attivazione sigmoide In questa architettura ho implementato anche il concetto di *Rete Residuale*, un particolare tipo di architettura che mira a combattere il problema del *Vanishing Gradient*, cioè

il fatto che, se si ha a che fare con reti profonde l'informazione sbiadisce progressivamente mentre attraversa gli strati della rete . Per questo risulta utile suddividere la rete in blocchi e concatenare l'output di ciascun blocco con l'iput del precedente[9] in modo che le informazioni contenute nell'input iniziale vengano mantenute anche negli strati più profondi della rete.

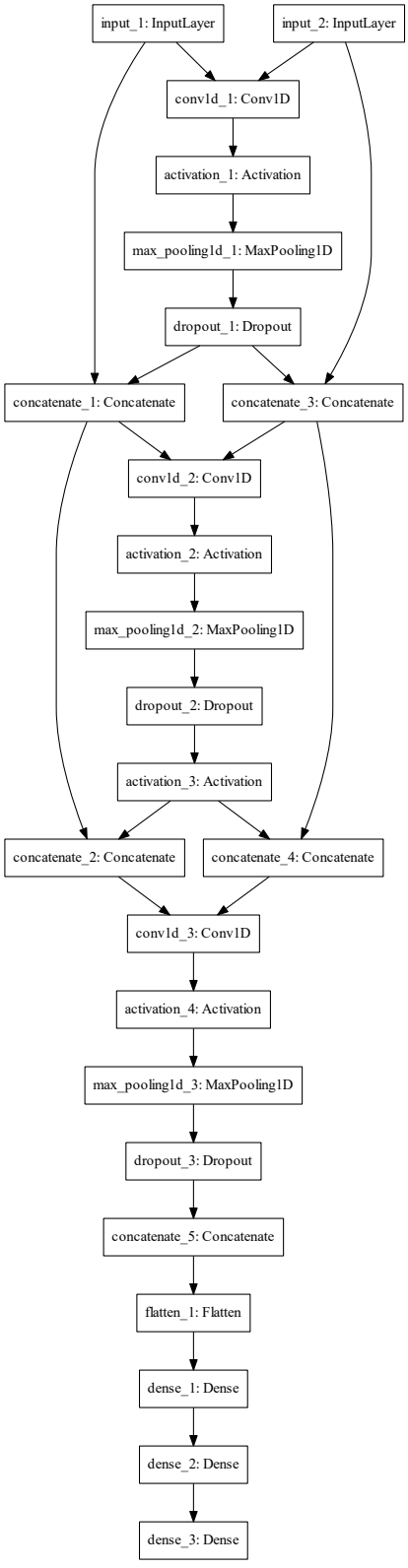

Figura 3.5: Architettura

Un passaggio successivo è stato normalizzare l'input cioè trasformarlo in modo che i valori risultino centrati e con una varianza unitaria

I risultati ottenuti sono i seguenti, si noti come normalizzare l'input migliori il risultato

normalizzato	**Log_loss**	**Accuracy**
no	0.4516	0.7742
si	0.4293	0.7863

3.4.1.2 LSTM

Le reti neurali ricorrenti sono un particolare tipo di architettura che permettono all'informazioni di persistere, cioè l'analisi di una particolare porzione dell'input dipende da ciò che è stato visto in precedenza; questa caratteristica rende questo tipo di reti ottime per lavorare con liste e sequenze, quali appunto le frasi[2] Il problema che caratterizza le reti ricorrenti è che le dipendenze tra le parole diventano progressivamente più deboli man mano che due parole sono distanti tra loro all'interno della frase, in modo simile al problema del *Vanishing Gradient* visto con il modello precedente. Questo rende difficile cogliere molte delle dipendenze all'interno delle frasi, poiché appunto parole molto distanti tra loro potrebbero comunque essere collegate, ad esempio una negazione all'inizio di una frase potrebbe condizionare il significato di parole anche molto distanti da essa.

Qui entrano in gioco le cosiddette reti *Long Short Term Memory*, LSTM: esse sono progettate specificatamente per evitare problemi con dipendenze a lungo termine, il loro scopo è proprio ricordare informazioni per lungo tempo.

L'architettura progettata utilizzando le LSTM è simile alla precedente, dove la componente siamese è data da un singolo layer LSTM, sopra le due reti siamesi abbiamo gli stessi elementi dell'architettura precedente: due livelli fully-connected con attivazione ReLU e un livello di output con attivazione sigmoide

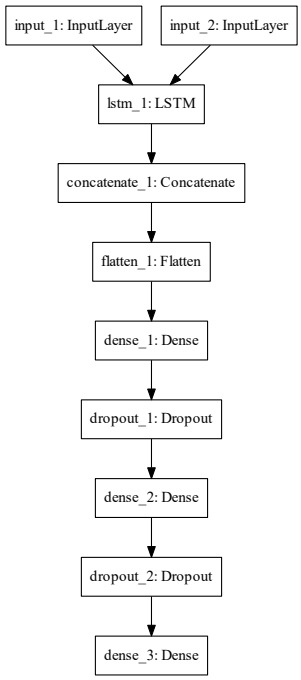

Figura 3.6: Architettura

Tale architettura ha ottenuto i seguenti risultati:

Log_loss	Accuracy
0.4258	0.8141

Se si osserva il grafico che confronta i risultati sul dataset di training e quello di validazione, risulta palese che la rete precedentemente illustrate ha un problema di overfitting

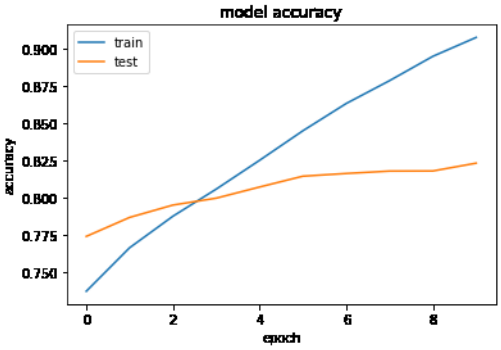

Figura 3.7: Model Accuracy

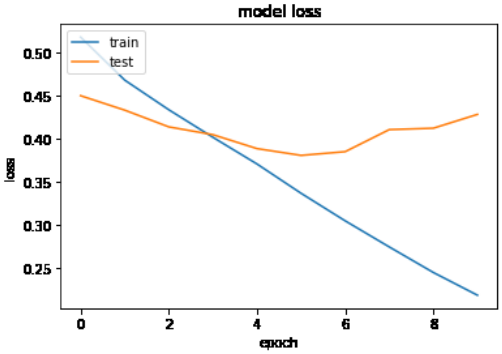

Figura 3.8: Model LogLoss

3.4.1.3 LSTM migliorata

Per cercare di incrementare la capacità di astrazione e generalizzazione del modello ho introdotto uno strato denso nella componente siamese, subito dopo lo strato LSTM e inoltre ho incrementato il Dropout e aggiunto due strati di batch normalization nella componente comune Le curve di LogLoss e Accuracy, seppur con valori migliori, seguono lo stesso andamento delle precedenti, inoltre i risultati ottenuti variano consistentemente al variare della quantità di dati messo a disposizione per l'addestramento

Dropout	Log_loss	Accuracy
0.3	0.3739	0.8255

3.4.2 GloVe Embeddings

Un altro metodo di rappresentazione delle frasi consiste nella tokenizzazione delle stesse per poi procedere con un'operazione di padding che permetta di avere input di uguale lunghezza. Questa rappresentazione però risulta essere ancora poco efficace, per questo in tutte le architetture che hanno tale rappresentazione come input, si inserirà un layer di embedding che converta le frasi in matrici facilmente processabili dalla rete e contenenti tuttavia informazioni significative.

In questo caso piuttosto che addestrare il layer di embedding da zero si è scelto di caricare i pesi al momento dell'istanziazione della rete e poi congelarli in modo che non vengano modificati durante l'addestramento.

I pesi utilizzati sono quelli forniti da *GloVe* [17], un algoritmo di apprendimento non supervisionato atto ad ottenere una rappresentazione vettoriale delle parole, sviluppato dalla *Stanford University*, e diventato uno degli standard in fatto di word embeddings. Il nome *GloVe* sta per *Global Vectors*, in quanto il modello cattura le statistiche globali del Corpus su cui viene addestrato, nello specifico il modello si basa principalmente sul rapporto tra le probabilità di co-occorrenza delle parole. Nello specifico dovendo stabilire la relazione tra due parole a e b si considera il rapporto tra la correlazione di a con parole appartenenti ad un insieme di controllo e la corrlazione di b con le parole del medesimo insieme.

In questo caso di studio è stato scelto di utilizzare i *Word Embeddings* ottenuti dal corpus *Common Crawl*, il file così ottenuto contiene l'embedding di 840 miliardi di parole sottoforma di vettori a 300 dimensioni, ed è il più grande disponibile sul sito ufficiale del progetto

3.4.2.1 Rete convoluzionale 1D

Questa architettura è molto simile a quella utilizzata con la rappresentazione vettoriale di *Spacy*, 3.4.1.1 l'unica differenza è l'aggiunta di un embedding layer alla componente siamese

3.4.2.2 LSTM

Anche in questo caso l'architettura è come quella utilizzata precedentemente con i vettori di Spacy 3.4.1.2, con l'aggiunta dell'embedding layer alla componente siamese

3.4.2.3 LSTM bidirezionale

La LSTM bidirezionale si ottiene concatenando due layer LSTM che leggano l'input in direzioni opposte, quindi il primo layer legge dall'inizio alla fine, il secondo dalla fine verso l'inizio, questo con l'obiettivo di catturare ulteriori relazioni tra le parole all'interno della frase

L'architettura utilizzata in questo caso, risulta equivalente a quella utilizzata al punto precedente, con il layer LSTM della componente siamese sostituito da un layer LSTM bidirezionale

3.4.2.4 Risultati

Modello	Log_loss	Accuracy
Convoluzionale 1D	0.5257	0.7790
LSTM	0.3981	0.8103
LSTM Bidirezionale	0.4029	0.8123

NB: Tutti i risultati sono stati ottenuti usando l'80% del dataset di training per addestrare i modelli e il 20% per validarli

Come si può vedere dai risultati, l'architettura siamese basata su LSTM risulta la migliore tra quelle sperimentate finora, sia per quanto riguarda la rappresentazione di *Spacy* che quella di $GloVe$. Vediamo graficamente i risultati:

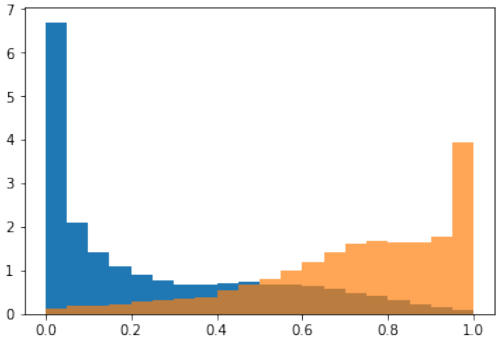

Figura 3.9: Risultati

Nelle sezioni successive si è cercato di migliorare ulteriormente il risultato mantenendo una struttura di base siamese

3.4.2.5 Tecniche di aumento dei dati

Partendo dal presupposto che la relazione di similarità è simmetrica ho pensato si potesse incrementare la quantità di dati su cui addestrare la rete semplicemente invertendo l'ordine delle frasi nelle coppie (in quanto A simile B è equivalente a B simile A). Da questo si ottiene che il dataset raddoppia di dimensione, incrementando quindi i tempi di addestramento, la capacità predittiva tuttavia non migliora

Log_loss	Accuracy
0.4019	0.8172

3.4.2.6 Differenza di risultati tra il validation set e il Test Set

Dopo aver provato a sottoporre l'algoritmo alla valutazione di Kaggle, ci si rende conto che il modello ottiene risultati molto diversi sul validation set rispetto al test set fornito da Kaggle, si tratta di un problema simile all'overfitting seppur in questo caso i risultati di training set e validation set non evidenzino alcun overfitting propriamente detto.

Tale problema è stato arginato riducendo il numero di *Epoch* della sessione di training, questo ha permesso di ottenere anche una riduzione dei tempi di addestramento senza comportare una riduzione eccessiva delle performance rispetto al dataset di validazione

n_epoch	Log_loss	Accuracy	Log_loss_Kaggle
50	0.3981	0.8103	0.4616
10	0.4156	0.7994	0.4403

3.4.2.7 Keras non restituisce come output il miglior modello

Dopo aver stampato dei grafici della sessione di allenamento per verificare l'andamento dei parametri di accuratezza e Log Loss, mi sono reso conto che, a differenza di Sci-kit Learn, Keras non restituisce il modello migliore, ma si limita a restituire il risultato dell'ultimo ciclo di addestramento.

Il metodo standard della libreria per ovviare a questo problema consiste nell'usare una callback per salvare il miglior modello, che sarà poi caricato per eseguire la predizione, tuttavia è possibile usare una funzione di callback costruita ad-hoc per ottenere in ritorno dalla funzione di addestramento il miglior modello, senza dover passare per la fase di salvataggio e caricamento

3.4.2.8 Merged Model

Come passaggio successivo ho voluto combinare i due modelli LSTM basati rispettivamente sulla rappresentazione con *Spacy* e sulla rappresentazione di $GloVe$.

Ho addestrato separatamente i due modelli per poi rimuovere da entrambi il layer di output e congelare tutti i livelli (in modo che nessuno dei due modelli venisse allenato nuovamente nella fase successiva). La libreria Keras non rende comoda questa operazione in quanto con i modelli classici Sequenziali, dove ogni ogni layer è aggiunto sopra il precedente, si possono aggiungere e togliere strati facilmente con apposite funzioni; nel caso di un modello più complesso, per ottenere il risultato sperato si hanno due possibilità:

- Si assegna ciascun layer ad una variabile al momento della costruzione dei due modelli "base" e si richiamano tali variabili al momento della costruzione del modello finale
- si itera attraverso i layer dei modelli preallenati e si costruisce il modello finale all'interno di questi due cicli

La prima opzione risulta macchinosa dovendo avere una variabile per ogni layer, ciò nonostante risulta comunque più comoda rispetto alla seconda in quanto nel secondo caso, si hanno problemi a tenere traccia di quale layer si sta processando.

Si noti che in entrambi i casi è necessario congelare ogni layer singolarmente, in quanto seppur Keras permetta di congelare tutti i layer di un modello in un colpo solo, sarà il modello ad essere congelato e non i layer, quindi in caso di riuso (come nel nostro caso) i layer risulteranno addestrabili seppur il modello da cui sono stati presi risulti congelato

Gli output dei due modelli vengono quindi concatenati e collegati a due strati Fully Connected che saranno gli unici ad essere allenati nella fase di addestramento del modello finale.

Log_loss	Accuracy
0.3685	0.8184

I risultati ottenuti da questo modello sono migliori rispetto a quelli ottenuti fino a questo momento, ma tale miglioramento, se rapportato al tempo necessario ad addestrare l'intero modello (contando i due da addestrare separatamente) fa pensare che non sia conveniente proseguire per questa strada.

3.4.2.9 Rete Convoluzionale 2D

Non completamente soddisfatto dei risultati ottenuti dal *Merged Model* ho deciso di Sperimentare ulteriori modelli: questo in particolare, è costruito sulla base dell'architettura presentata nell'articolo [12]: 3 filtri convoluzionali di larghezza pari alla lunghezza di una parola e altezza corrispondente a 2, 3, e 4 rispettivamente vengono applicati alla frase rappresentata tramite i *GloVe Embeddings*, da ciascuno si ricavano due canali, per un totale di sei. A ciascuno di questi canali viene applicato un pooling *max* e poi vengono concatenati per fare da input ad uno strato denso.

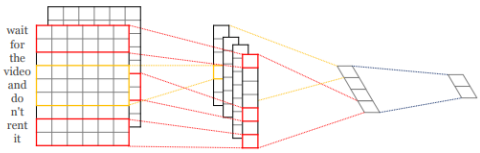

Figura 3.10: Modello CNN

Nel modello qui presentato quell'architettura costituisce la componente siamese della rete che viene quindi applicata ad entrambe le frasi. Gli output vengono concatenati e usati come input per due strati densi con attivazione *RElu* e un ultimo strato denso con attivazione *sigmoide* Questo Modello ha ottenuto i seguenti risultati:

Log_loss	**Accuracy**	**Log_loss Kaggle**
0.2891	0.8808	0.5980

Nonostante il modello CNN abbia ottenuto ottimi risultati sul *Validation Set* una volta eseguito sul dataset di Test ha prodotto un risultato inaspettatamente negativo, ottenendo una LogLoss di 0.5980 dimostrando di avere una scarsa capacità di generalizzazione

3.4.3 Concatenazione delle rappresentazioni Vettoriali delle frasi

Nel corso di questo studio, a seguito della lettura di numerosi articoli sembrava chiaro che l'utilizzo di reti siamesi combinate con un qualche tipo di rappresentazione vettoriale, fosse il metodo più efficace per verificare se due frasi fossero parafrasi oppure no, e che fosse un approccio nuovo e migliore rispetto alla precedente pratica di usare come input la concatenazione dei vettori rappresentanti le due frasi da confrontare.[5]

Ho comunque voluto sperimentare questa tecnica, che ho usato come input per un'architettura $MultilayerPerceptron$ come quella utilizzata in 3.1.2, con l'aggiunta di layer *Dropout* e layer $BatchNormalization$ tra un layer denso e il successivo. Ho ottenuto i seguenti risultati:

input	**Log_loss**	**Accuracy**	Log_loss_kaggle
Spacy	0.4163	0.7975	0.3763
Spacy Normalizzati	0.3927	0.8125	0.3661
Glove	0.4736	0.7869	0.4021

Se i risultati sul validation Set, almeno per quanto riguarda la rappresentazione vettoriale prodotta da *Spacy*, sono comparabili rispetto a quelli ottenuti dal $MergedModel$, ciò che colpisce sono i risultati sul Test Set che sono decisamente migliori. Sembra quindi che, seppur le reti siamesi siano un metodo nuovo, nato con lo scopo di migliorare i risultati portati dalla concatenazione dei vettori,

almeno in questo caso di studio, il vecchio metodo si rivela ancora piuttosto efficace. Altro fatto notevole è che in questo modello, contrariamente a tutti gli altri, la rappresentazione di *Spacy* produce un risultato migliore rispetto quella fornita da $GloVe$.

4 Utilizzo di un algoritmo Genetico

Vista la straordinaria efficacia dell'algoritmo proposto in 3.4.3, soprattutto se rapportata ai suoi tempi di addestramento relativamente brevi, ho pensato che potesse essere un buon candidato per sperimentare l'algoritmo di ottimizzazione cosiddetto *Genetico* o *Evolutivo*

Tale algoritmo è così chiamato perchè trae ispirazioni dalla teoria dell'evoluzione e della selezione naturale sviluppata da Darwin[7] secondo cui gli esseri viventi che proliferano in un dato ambiente, sono quelli che meglio si adattano ad esso, sviluppando caratteristiche utili alla loro sopravvivenza, al contrario le specie che non si adattano finiscono per soccombere.

Similmente in quest'algoritmo viene generata una popolazione iniziale di Reti Neurali, a quelle che si rivelano essere più adatte a risolvere il problema in esame viene data la possibilità di sopravvivere e riprodursi, mentre le reti peggiori vengono eliminate; questo procedimento si ripete per diverse generazioni, arrivando alla fine ad ottenere l'esemplare di Rete Neurale perfetto per il problema da risolvere.

4.1 Scelta delle caratteristiche da ottimizzare

Il primo passo per la realizzazione di un algoritmo di ottimizzazione genetico è la scelta delle caratteristiche che distinguono un *Individuo* della popolazione da un altro e il modo in cui fare variare tali caratteristiche. Prendendo in esame il modello esposto in 3.4.3 le caratteristiche che lo definiscono sono:

- *Numero di layer*: La prima caratteristica è ovviamente la profondità della rete, in questo caso ho scelto di far variare tale profondità tra 2 e 7, perchè volevo esplorare modelli relativamente profondi, ma senza esagerare, memore di quanto visto in 3.1.3 dove modelli troppo profondi perdevano di efficacia. Inoltre si consideri che il blocco fondamentale che compone la rete è un layer *Denso*, seguito da un layer di *Batch Normalization* e uno di *Dropout*, quindi il numero di layer indica in realtà quante volte ripetere tale blocco.

- Numero di neuroni per ogni strato: Il tipo di rete analizzato consta principalmente di una serie di strati densamente connessi quindi la sua seconda caratteristica principale è sicuramente il numero di neuroni che si trovano in ciascun stato, per comodità e per non espandere troppo l'area di ricerca ho preso due decisioni:

  - Il numero di neuroni varia all'interno di una lista di numeri ridotta, che ho riempito considerando i valori che ho visto più spesso nei vari modelli in cui mi sono imbattuto: [100, 200, 500, 1000].
  - All'interno di una determinata rete tutti gli strati avranno lo stesso numero di neuroni, ad eccezione dello strato di output che ne avrà 2 (pari al numero delle classi da predire).

- *Funzione di attivazione*: la funzione di attivazione da utilizzare per i layer, anche in questo caso ho preso delle decisioni al riguardo:

  - La funzione di attivazione varia in un gruppo preselezionato di funzioni selezionate tra quelle che nella mia esperienza hanno funzionato meglio: *Relu, Elu, SoftPlus.*
  - All'interno di una stessa rete tutti gli strati utilizzano la stessa funzione di attivazione, ad esclusione dello strato di output, questa scelta è dovuta a quanto appreso in 3.1.2

- *Funzione di attivazione dello strato di Output*: Come anticipato la funzione di attivazione dell'ultimo strato è diversa da quella degli altri ed è scelta tra le seguenti: *[Sigmoid, Softmax]*, seppur quasi certo che la migliore funzione di output per questo problema sia la sigmoide, in quanto di solito essa viene usata per le classificazioni binarie mente la softmax viene usata per quelle con più classi possibili, ho deciso di dare comunque una possibilità a entrambe.

- *Valore di Dropout*: il valore di *Dropout* da assegnare agli strati di Dropout della rete, anche in questo caso il valore è unico e comune a tutti gli strati di una singola rete. Tale valore varia tra i seguenti: [0.1, 0.2, 0.3, 0.4, 0.5], ho scelto di limitare il valore a un massimo di cinque visto che, utilizzando *Keras* tale valore indica la percentuale di neuroni da scartare in quel punto.

4.2 L'algoritmo

Passeremo ora a vedere nel dettaglio le diverse fasi che compongono l'esecuzione di un algoritmo evolutivo, facendo riferimento a quello implementato in questo caso specifico.

4.2.1 Generazione della popolazione iniziale

La prima fase consiste nel generare una prima popolazione, scegliendo in maniera casuale le caratteristiche di ciascuna *Rete Neurale* (o *Individuo*). Nel mio caso ho optato per una popolazione composta da 20 *individui*

4.2.2 Valutazione della popolazione

Nella seconda fase la popolazione viene valutata rispetto ad un dato parametro e ordinata dall'elemento migliore al peggiore. Nell'istanza dell'algoritmo che stiamo considerando ciascuna *Rete* viene addestrata e validata, la *LogLoss* risultante dalla validazione sarà il parametro di valutazione della *Rete*: gli *Individui* considerati migliori saranno quelli con la *LogLoss* minore.

4.2.3 Selezione degli Individui migliori

In questa fase, sulla base delle valutazioni ottenute al punto precedente 4.2.2, si selezionano gli individui da mantenere e quelli da eliminare. Nel mio caso vengono mantenuti inalterati i 2 migliori individui che saranno i primi componenti della nuova *Popolazione*.

4.2.4 Crossover

In questa fase coppie di *Individui* della popolazione vengono scelte per generare *Individui* ibridi che erediteranno le proprie caratteristiche dai *genitori*. Considerando l'eventualità che dalla combinazione di parametri di reti non efficaci ne nasca una efficente, ho deciso di dare la possibilità a tutti gli individui di essere scelti per generare *ibridi*. Ciascun individuo i ha però una possibilità di essere scelto proporzionale alla sua valutazione calcolata come:

$$p_i = \frac{1 - Valutazione_i}{Valutazione_{popolazione}} \tag{4.1}$$

dove la Valutazione della popolazione è calcolata come:

$$Valutazione_{popolazione} = \sum_{i=1}^{20} 1 - Valutazione_i \tag{4.2}$$

Basandosi su tale probabilità vengono scelti due *Individui*, il loro ibrido erediterà 1 o 2 caratteristiche dal primo genitore e le restanti dal secondo.

Ripetendo questo procedimento 16 volte si ottengono altrettanti membri della nuova popolazione

4.2.5 Mutazione

A questo punto i restanti membri della nuova popolazione vengono ottenuti tramite mutazioni casuali dei membri della popolazione attuale. I restanti 2 membri della nuova popolazione generati dal mio algoritmo sono ottenuti selezionando casualmente 2 membri della popolazione attuale e modificando casualmente uno dei parametri di ognuno.

4.2.6 Ripetizione dell'algoritmo

A questo punto si ha a disposizione una nuova popolazione su cui ripetere le operazioni svolte in precedenza, a partire dal punto 4.2.2. Tali operazioni vengono svolte per un numero n di *Generazioni*, nel mio caso 10

4.3 Risultati

L'algoritmo ottenuto tramite questo processo evolutivo è il seguente

Numero di Layer	Neuroni per Layer	F1	F2	Dropout	Accuracy	Log Loss
3	1000	Relu	Softmax	0.3	0.3785	0.8199

L'algoritmo ha prodotto dei miglioramenti, anche se non molto consistenti, comunque questa tecnica di ottimizzazione si è rivelata interessante e, avendo a disposizione maggiore potenza di calcolo, sarebbe interessante sperimentarla su algoritmi più complessi.

Supponiamo si decida di applicare l'algoritmo al modello LSTM visto in 3.4.2.2, che ha un tempo di addestramento di circa 8 ore; considerato che è necessario addestrare (con la configurazione usata precedentemente) 20 modelli per la prima generazione e 18 per le successive, si otterrebbe un totale di 182 modelli, e il tutto richiederebbe circa 60 giorni per essere eseguito, un tempo che, almeno nell'ambito di questo studio, risulta proibitivo.

4.4 Aumento dei dati

L'algoritmo ottenuto tramite l'ottimizzazione genetica risulta il migliore tra quelli sperimentati finora. Vale dunque la pena cercare di migliorare ulteriormente il risultato applicando la stessa tecnica di raddoppio del dataset utilizzata in 3.4.2.5. Effettivamente in questo caso, contrariamente a quanto avvenuto con la LSTM, si ottiene un buon miglioramento:

Log_loss	**Accuracy**	Log_loss_kaggle
0.3370	0.8446	0.33086

5 MSRP - MSR Paraphrase Corpus

In questo capitolo vedremo qual è l'attuale stato dell'arte[3] per quanto riguarda l'individuazione di parafrasi e lo confronteremo con gli algoritmi siviluppati nel corso di questo studio

5.1 Il dataset

Quando si parla di identificazione di Parafrasi, il dataset di riferimento è il *Microsoft Research Paraphrase Corpus*, fornito dalla *Microsoft Research*. Tale dataset si compone di:

- *Dati di training*: 4,076 coppie di frasi, di cui 2,753 effettivamente parafrasi (67.5%)
- *Dati di test*: 1,725 coppie di frasi, di cui 1,147 effettivamente parafrasi (66.5%)

Si nota subito che le dimensioni del dataset sono decisamente inferiori a quelle del dataset in studio: 4000 elementi di training contro 400000 e 1700 elementi di test contro 2350000; per questo ci si può aspettare che modelli sviluppati avendo disposizione una tale quantità di dati, non ottengano risultati altrettanto buoni se addestrati su un dataset ridotto

Nel valutare la bontà di un modello rispetto al $MSRP$ si utilizza come metrica l'*F1 Score*

5.1.1 F1 Score

L'*F1 Score* è una misura dell'accuratezza di una dato test, in questo caso la previsione dell'algoritmo. Per capire cosa rappresenti questo valore, è necessario comprendere che data una previsione, si potranno quattro eventi distinti:

- *True Positive*: una coppia di frasi etichettate dall'algoritmo come parafrasi sono effettivamente parafrasi.
- *True Negative*: l'algoritmo etichetta due frasi come non duplicati, ed effettivamente non lo sono
- *False Positive*: l'algoritmo etichetta due frasi come duplicati quando in realtà non lo sono
- *False Negative*: l'algoritmo etichetta la coppia di frasi come non duplicati ma in realtà sono effettivamente duplicati

Risulta chiaro che, dopo che l'algoritmo avrà effettuato la sua previsione sul *Validation Set*, (in una condizione di apprendimento supervisionato perchè è necessario confrontare i risultati con i valori reali) avremmo quattro valori a disposizione: il numero di *True Positive (TP)*, il numero di *True Negative (TN)*, il numero di *False Positive (FP)*, e il numero di *False Negative (FN)*. Da questi quattro valori possiamo ricavare degli indicatori riguardo la bontà della previsione:

- *Precision*:
$$Precision = TP/(TP + FP) \tag{5.1}$$
La precision è quindi la misura di quante coppie che l'algoritmo etichetta come duplicate, sono effettivamente duplicate
- *Recall*:
$$Recall = TP/(TP + FN) \tag{5.2}$$
Il recall è la misura di quante coppie duplicate sono etichettate correttamente rispetto al loro numero totale

L'*F1 Score* è una misura che ha lo scopo di considerare entrambe le precedenti e si calcola come loro media armonica.
$$F = 2 * ((Precision * Recall)/(Precision + Recall)) \tag{5.3}$$

5.2 Risultati degli algoritmi

I risultati degli algoritmi sviluppati nel corso di questo studio, rispetto al $MSRP$ sono i seguenti, ordinati per *F1 Score*, sono inseriti come riferimento anche i due migliori risultati in assoluto:

Algoritmo	Accuracy	F1 Score
LSTM - Spacy	0.6730	0.6730
MLP - Spacy	0.6773	0.6753
Merged Model	0.6756	0.6758
LSTM - Glove	0.6859	0.6852
CNN - Glove	0.6988	0.6975
SAMS - RecNN[5]	**0.786**	**0.853**
TF - KLD[10]	**0.804**	**0.859**

5.3 Considerazioni

Come pronosticato, gli algoritmi proposti, pur ottenendo buoni risultati nel dataset di Quora, non ottengono risultati altrettanto buoni se addestrati su un dataset di dimensioni così ridotte e sono abbastanza distanti da quello che è lo stato dell'arte su questo dataset[10] con un *F1 Score* pari a 0.859

Un'altro fattore che incide sul risultato è il fatto che, mentre nel corso della competizione cioè che era richiesto di predire era la probabilità che le domande fossero duplicate, in questo caso, ciò che è richiesto è una decisione precisa tra duplicati e non duplicati. Questo fa si che non si consideri il valore che ha portato a dare tale risposta ma solo la risposta stessa, per fare un esempio pratico, una coppia duplicata, che secondo le previsioni, ha una probabilità dello 0.49% di essere duplicata, viene considerata come non duplicata nel caso del MSRP, mentre nel caso di Quora viene tenuto conto di tale percentuale. Infatti i due metodi di valutazione differiscono proprio in questo aspetto:

- La *Log Loss* di Quora tiene conto della probabilità assegnata alla previsione
- L'*F1 Score* usato dall'MSRP conta solo la correttezza della previsione

6 Conclusioni

Come visto, il Deep Learning ha ottenuto risultati decisamente superiori rispetto ai principali algoritmi di Machine Learning classico. Questo perché una volta rappresentato l'input in un modo comprensibile alla macchina è stata essa stessa, tramite la rete neurale, ad elaborare le informazioni per cercare di ottenere previsioni accurate.

Lo sviluppo hardware degli ultimi anni ha dato una grossa spinta al Deep Learning, permettendo di progettare Reti Neurali sempre più complesse, in grado di cogliere relazioni tra i dati in maniera sempre più efficace.

I modelli presentati in precedenza sono si complessi, ma la loro complessità non si avvicina nemmeno a ciò che si potrebbe progettare avendo a disposizione hardware più performante. Nello specifico durante questo progetto la complessità delle reti era nell'ordine delle centinaia di migliaia di parametri, in alcuni casi milioni di parametri. Raggiunte le decine di milioni di parametri, la memoria risulta insufficiente.

Si prendano ad esempio i quattro modelli che ottengono i risultati migliori:

- La *Rete Siamese LSTM* applicata alle rappresentazioni vettoriali della libreria di Spacy: ha un costo di memoria ridotto ma richiede circa due giorni per essere addestrato.
- La *Rete Siamese LSTM* applicata ai *GloVe Embeddings*: richiede 8 ore per essere addestrato e ha un consumo di memoria nella norma, il layer di embedding contribuisce considerevolmente alla dimensione finale del modello
- Il *Merged Model* applicato alla rappresentazione delle frasi tramite i *Word Embeddings* di *GloVe* e le rappresentazioni vettoriali di *Spacy*: ha un tempo di addestramento considerevole (se si considera l'addestramento dei due modelli che lo compongono) e un consumo di memoria da non sottovalutare
- La *Rete Neurale Convoluzionale 2D*, applicata ai *Word Embeddings* di *Glove*: Ha un consumo di memoria considerevole, e richiede circa 20 ore per essere addestrato
- L'$MLPClassifier$ applicato alle rappresentazioni vettoriali delle frasi ottenute dalla libreria *Spacy*, che vengono normalizzate e concatenate: non presenta eccessivi bisogni in termini di memoria e viene allenato in meno di un'ora.

Se si confrontano i risultati ottenuti sul *Test Set* fornito da *Kaggle* si nota come il modello più semplice dei quattro ottenga risultati migliori degli altri. L'approccio più classico al problema, che consiste nel concatenare le rappresentazioni delle frasi, risulta quindi migliore rispetto agli approcci che sfruttano le Reti Neurali Siamesi. Le rappresentazioni vettoriali fornite dalla libreria *Spacy* danno inoltre risultati migliori rispetto ai $GloVeEmbeddings$ Altra cosa degna di nota è il fatto che quasi tutti i risultati superano di molto quello ottenuto tramite Il *Random Forest Classifier*, a riprova del fatto che, in questo caso, avendo a disposizione una buona quantità di dati, il Deep Learning risulta la tecnica migliore da utilizzare.

Architettura	Log_loss	Accuracy	Log_loss Kaggle
Random Forest	0.4648	0.7456	0.4238
LSTM - Spacy	0.3739	0.8255	0.37815
LSTM - Glove	0.4156	0.7994	0.4403
Merged Model	0.3685	0.8184	0.3747
CNN - Glove	0.2891	0.8808	0.5980
MLP - Vettori concatenati Spacy Normalizzati	0.3370	0.8446	0.33086

Il risultato ottenuto nella competizione permette di posizionarsi in posizione 1240 su 3300, un risultato di tutto rispetto, considerato l'enorme miglioramento compiuto nel lasso di tempo di questo progetto.

Dovendo scegliere una strada verso cui proseguire questa ricerca opterei sicuramente per l'utilizzo delle reti siamesi perché, pur non avendo ottenuto il risultato migliore tra quelli presentati, sono comunque la tecnica che a mio avviso ha maggiori possibilità di sviluppo in futuro, in quanto varie strutture (LSTM, CNN) si prestano ad essere utilizzate come componenti per tale architettura. Non bisogna inoltre dimenticare che il secondo posto in quanto a precisione sul $MSRP$ è stato ottenuto proprio grazie ad un'architettura siamese. Anche le stesse *Capsule Networks*, l'ultima innovazione in fatto di Reti Neurali, si prestano a mio avviso all'utilizzo nell'ambito dell'NLP, esse sono viste infatti come successori delle reti Convoluzionali che abbiamo visto essere facilmente applicabili a questo tipo di problemi [20]. Obiettivo di questo studio era dimostrare le potenzialità del Deep Learning, e come si è visto le Reti Neurali, oltre alle potenzialità, posseggono anche un'enorme versatilità che sta facendo sì che vengano usate ormai nei più svariati ambiti.

Siamo ancora distanti dallo sviluppare un'intelligenza artificiale che sia realmente intelligente, capace di pensiero autonomo, tuttavia la cosiddetta Intelligenza artificiale applicata, cioè l'imitazione del funzionamento del cervello umano, che si focalizza su un compito specifico, sta producendo ottimi risultati, e, come abbiamo dimostrato nel piccolo di questo studio, quando si ha a che fare con grosse moli di dati, è sicuramente la scelta migliore

Bibliografia

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

[2] Abhishek. Approaching (almost) any nlp problem on kaggle. `https://www.kaggle.com/abhishek/approaching-almost-any-nlp-problem-on-kaggle/`. Last Visited, 1/3/2018.

[3] ACL. State of the art in paraphrase detection. `https://aclweb.org/aclwiki/Paraphrase_Identification_(State_of_the_art)`. Last Visited, 6/3/2018.

[4] Roberto Battiti and Mauro Brunato. *The LION way. Machine Learning* plus *Intelligent Optimization*. LIONlab, University of Trento, Italy, 2017.

[5] Jianpeng Cheng and Dimitri Kartsaklis. Syntax-aware multi-sense word embeddings for deep compositional models of meaning. *CoRR*, abs/1508.02354, 2015.

[6] François Chollet et al. Keras. `https://github.com/keras-team/keras`, 2015.

[7] C.R. Darwin. *L'origine della specie*. Grande biblioteca della scienza. Fabbri, 2006.

[8] FastAi. Log loss definition. `http://wiki.fast.ai/index.php/Log_Loss`. Last Visited, 6/3/2018.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[10] Yangfeng Ji and Jacob Eisenstein. Discriminative improvements to distributional sentence similarity. In *EMNLP*, pages 891–896. ACL, 2013.

[11] Kaggle. Quora question pairs - data. `https://www.kaggle.com/c/quora-question-pairs/data`. Last Visited, 6/3/2018.

[12] Yoon Kim. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014.

[13] Alberto Montresor. Programmazione dinamica. Last Visited, 5/3/2018.

[14] Jonas Mueller and Aditya Thyagarajan. Siamese recurrent architectures for learning sentence similarity. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2786–2792. AAAI Press, 2016.

[15] Stanford University School of Engineering. Natural language processing with deep learning, full course. `https://www.youtube.com/watch?v=OQQ-W_63UgQ`. Last Visited, 5/3/2018.

[16] Stanford University School of Engineering. Stanford nlp group, home. `https://nlp.stanford.edu/`. Last Visited, 5/3/2018.

[17] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543, 2014.

[18] Quora. Home. `https://www.quora.com/`. Last Visited, 5/3/2018.

[19] Richard Socher and Eric H. Huang and Jeffrey Pennington and Andrew Y. Ng and Christopher D. Manning. Dynamic Pooling and Unfolding Recursive Autoencoders for Paraphrase Detection. In *Advances in Neural Information Processing Systems 24*. 2011.

[20] Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. Dynamic routing between capsules. *CoRR*, abs/1710.09829, 2017.

[21] Scikit-Learn. Homepage. `http://scikit-learn.org/stable/`. Last Visited, 6/3/2018.

[22] Spacy. Homepage. `https://spacy.io/`. Last Visited, 6/3/2018.

[23] Karen Sparck Jones. Document retrieval systems. chapter A Statistical Interpretation of Term Specificity and Its Application in Retrieval, pages 132–142. Taylor Graham Publishing, London, UK, UK, 1988.