Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Modelli di Deep Learning per la generazione di Linguaggio Naturale

Relatore:
Chiar.mo Prof.
ANDREA ASPERTI

Presentata da:
ANDREA BINZONI

Sessione II
Anno Accademico 2020/2021

Alla mia FAMIGLIA.

Introduzione

«I am not a human. I am a robot.». L'8 settembre del 2020 sulle colonne del prestigioso "The Guardian" un modello di *Machine Learning* ci rassicurava sulle sue intenzioni. GPT-3, questo il nome dell'"autore", ha raccontato ai lettori britannici cosa è in grado di fare e le numerose opportunità che in futuro ci potrà offrire.

Nella mia esperienza decennale di programmatore informatico mi è capitato di affrontare problemi particolarmente complessi, che dovevano tener conto di innumerevoli regole e caratterizzati dalla frequente comparsa di nuove variabili e casi particolari. Un approccio tradizionale porterebbe a scrivere programmi che richiederebbero continui interventi sul codice, ma soprattutto i risultati prodotti potrebbero non tenere in considerazione parametri importanti e non ancora conosciuti.

Il *Machine Learning* entra in gioco per risolvere questi (e tanti altri) tipi di problemi, ma oggi è diventato qualcosa di più, e grazie al *Deep Learning* siamo in grado di realizzare attività molto più complesse negli ambiti più disparati.

Quando è giunto il momento di scegliere l'argomento della Tesi Finale non ho avuto dubbi, volevo approfondire la conoscenza di questo mondo e scoprire le modalità con cui complesse strutture matematiche, come le Reti Neurali Artificiali, siano in grado di simulare il comportamento del cervello umano, per svolgere attività complesse nel modo in cui sono state programmate per farlo.

Il lavoro è strutturato in 3 capitoli, il primo è di carattere indroduttivo e

si pone l'obiettivo di approfondire tutti i principali componenti del *Machine Learning*, gli algoritmi e il processo di apprendimento, per poi focalizzarsi su *Deep Learning* e sulle principali famiglie di Reti Neurali Artificiali.

La seconda parte è dedicata all'attività che ho scelto di sperimentare, la generazione automatica di testo. Partendo dall'elaborazione del linguaggio naturale passo in rassegna tutte le tecniche e le architetture maggiormente utilizzate per questo scopo: LSTM, GRU, GAN e *Transformer*, una tecnologia rivoluzionaria che ha permesso un importante salto di qualità

Infine, la terza parte è dedicata alla sperimentazione, le basi teoriche acquisite mi hanno permesso di costruire un modello basato sull'architettura LSTM in grado di generare del testo a livello carattere in lingua italiana.

L'obiettivo che mi ero posto è stato raggiunto, il modello è piuttosto semplice, ma è stato comunque in grado di apprendere alcuni concetti fondamentali nonostante il contenuto complessivo sia privo di significato.

Indice

Elenco delle figure

Capitolo 1

Machine Learning

1.1 L'apprendimento automatico

Con il termine *Machine Learning* intendiamo il campo di studio che fornisce ai calcolatori la capacità di imparare senza essere esplicitamente programmati. L'apprendimento può avvenire utilizzando dati di esempio o esperienze passate e, in base alle conoscenze acquisite, vengono prodotti risultati di tipo predittivo per fare previsioni in futuro, o descrittivo per acquisire conoscenze dai dati. [1]

Il primo accenno a questa tecnologia risale al 1950, quando Alan Turing pubblicò il celebre articolo "Computing machinery and intelligence", in cui per la prima volta si esplorò la possibilità di creare un'intelligenza artificiale sfruttando gli schemi del cervello umano. Da qui nacque il celebre "Imitation Game" [17], un test che permetteva ad una macchina di rendersi indistinguibile ad un essere umano.

Dopo decenni di studi e sperimentazioni, il *Machine Learning* è oggi uno strumento preziosissimo per il nostro mondo interconnesso, in molti casi funziona talmente bene che non ci accorgiamo nemmeno della sua presenza: assistenti vocali e filtri antispam, traduttori automatici e filtri per la bellezza sono solo alcuni esempi delle innumerevoli attività realizzate da questa tecnologia.

Per consentire ad un dispositivo elettronico di acquisire nuove abilità, il *Machine Learning* costruisce algoritmi allo scopo di replicare il processo di apprendimento degli esseri umani attraverso l'applicazione di formule matematiche.

1.2 Gli Algoritmi

Gli algoritmi sono progettati per risolvere un determinato problema, elaborando i dati attraverso una serie di stati ben definiti e non necessariamente deterministici e producendo un output che risolva il problema. Il *Machine Learning* in un certo senso capovolge il modo di approcciare un problema: le informazioni di partenza sono i dati di input e un risultato atteso, e ciò che vogliamo ottenere dall'apprendimento è una funzione che produca la soluzione. Gli algoritmi di apprendimento, a seconda delle finalità, possono essere suddivisi in 3 grandi famiglie:

- L'**Apprendimento supervisionato** è il caso in cui un algoritmo impara attraverso dati di esempio comprensivi del risultato corretto. Il suo scopo è fare in modo che l'algoritmo sia in grado prevedere le risposte esatte quando gli vengono sottoposti dei nuovi esempi. Fanno parte di questo gruppo i problemi di regressione, in cui l'obiettivo è predire una quantità numerica, e i classificatori, il cui *target* è una variabile qualitativa, come una classe o un'etichetta.

- L'**Apprendimento non supervisionato** è caratterizzato dalla mancanza delle risposte associate agli esempi: in questi casi è quindi l'algoritmo che impara semplicemente dai dati, e in autonomia ne individua i pattern. Sono particolarmente utili per estrarre nuove caratteristiche dai dati stessi e creare nuovi input per algoritmi supervisionati. Un tipico esempio è il *clustering*, il cui scopo è raggruppare tra loro dati con caratteristiche simili.

- L'**Apprendimento per rinforzo** rappresenta il concetto secondo il quale si impara anche dagli errori. Gli esempi sono privi della relativa risposta, ma in questo caso l'algoritmo propone una soluzione che riceve a sua volta un riscontro, positivo o negativo, dal quale è in grado di apprendere quale approccio avrà minori probabilità di successo.

1.3 I Dati

I DataSet sono il complesso dei dati su cui viene eseguito l'insieme di operazioni che compongono l'algoritmo di apprendimento. Per risultare efficaci devono essere sufficientemente grandi da permettere il riconoscimento di specifici pattern durante la fase di addestramento. Quando sono particolarmente corposi vengono definiti come "big data". Proprio come un essere umano, che è in grado di riconoscere un oggetto in base ad alcune caratteristiche (dopo averlo osservato un numero sufficiente di volte), allo stesso modo l'algoritmo usa i dati come osservazioni per apprendere le caratteristiche del problema, e, man mano che la computazione avanza, affinare le tecniche, in questo caso funzioni matematiche, adatte a risolverlo.

1.3.1 Dati di Training e Dati di Test

Per testare al meglio un modello di *Machine Learning* andrebbero utilizzati dei dati che l'algoritmo non ha mai visto prima, ma siccome nella pratica è una possibilità molto rara, è consuetudine suddividere il dataset in 2 sottoinsiemi distinti: il set di addestramento, che in genere comprende il 70% dei casi, e il set di test, comprensivo del restante 30. Se la dimensione dei dati è sufficientemente grande, la suddivisione può essere effettuata in modo casuale, ma in caso contrario occorre allontanare la possibilità che alcuni dati particolarmente utili vengano esclusi dalla fase di addestramento.

K-fold Cross Validation

Questo metodo consiste in una suddivisione di dati in un numero k di gruppi di uguale dimensione: ad ogni iterazione viene estratta una porzione di dati come set di test mentre le restanti k-1 formano il set di addestramento. Al termine della procedura tutti i gruppi sono stati utilizzati come set di test ed è possibile calcolare la stima della campionatura più efficace.

1.4 Il Processo di Addestramento

La chiave di lettura del *Machine Learning* è che ogni aspetto della realtà può essere rappresentato attraverso una funzione matematica, che inizialmente non è conosciuta, ma che viene scoperta e affinata man mano che vengono elaborati i dati di apprendimento. Ad alto livello può essere quindi descritto come la realizzazione di algoritmi capaci di rilevare e imparare delle relazioni nei dati.

Ogni caso di esempio può quindi essere visto come una singola osservazione, ognuna delle quali è a sua volta composta da diverse caratteristiche (*feature*) da cui l'algoritmo estrae i suoi parametri interni. Lo scopo finale della fase di addestramento è quindi effettuare una mappatura tra le *feature* e le risposte e produrre una funzione che approssimi al meglio la funzione obiettivo, ovvero quella che associa i casi di esempio ai rispettivi risultati corretti.

1.4.1 La Funzione di Costo

La Funzione di Costo (Loss Function) ha il compito di misurare la capacità dell'algoritmo di approssimare la funzione obiettivo e quindi stabilirne il livello di errore. Può essere rappresentata come la differenza tra le previsioni dell'algoritmo e i risultati corretti, in questo modo il processo di addestramento può essere ridotto ad un problema di ottimizzazione in cui si

deve minimizzare la Funzione di Costo attraverso la ricerca del suo punto di minimo globale.

1.4.2 La Discesa del Gradiente

Il metodo della Discesa del Gradiente è il procedimento più diffuso per raggiungere tale obiettivo. Dopo aver ricevuto un insieme di input costituiti da *feature* e *target*, l'algoritmo genera un risultato casuale, quindi prosegue attraverso una serie di iterazioni, modificando di volta in volta i propri parametri interni nella direzione che riduce la Funzione di Costo. Per raggiungere il punto di minimo, e quindi l'approssimazione cercata, possono essere necessarie un numero molto elevato di iterazioni, ma ad ogni passaggio l'errore sarà gradualmente ridotto. [15]

1.4.3 Problemi comuni

Approssimazione è il concetto chiave su cui concentrarsi per non incorrere in alcuni problemi piuttosto comuni dovuti principalmente alla complessità del modello.

Figura 1.1: Un esempio di corretta approssimazione, underfitting e overfitting

Overfitting

Durante la fase di addestramento l'algoritmo potrebbe "imparare a memoria" i dati elaborati; in questi casi il modello è in grado di effettuare previ-

sioni corrette esclusivamente per i campioni di esempio già visti nel training generando in questo modo un problema di overfitting.

Feature qualitative

Può accadere con una certa frequenza che il dataset di cui disponiamo contenga informazioni utili di carattere qualitativo, come la professione di una persona, o il colore di un oggetto, che necessitano di essere trasformati in valori numerici per essere manipolati correttamente dall'algoritmo. La tecnica di codifica che rappresenta l'approccio più comune prende il nome di "One Hot Encoding" e consiste nella trasformazione binaria delle *feature* qualitative: vengono create tante *feature* quante sono le classi presenti al suo interno e, per ogni osservazione, vengono inizializzati tutti i valori a zero, eccetto per la classe presente in tale osservazione che assumerà il valore di 1. Questa soluzione risolve il problema ma in alcuni casi può causare l'esplosione della dimensione dell'input.

1.5 Le Reti Neurali

Per la risoluzione di problemi particolarmente complessi, come il riconoscimento di immagini o la comprensione del linguaggio naturale, sono state realizzate le Reti Neurali, una famiglia di algoritmi di apprendimento straordinariamenti efficaci, che traggono ispirazione diretta dal comportamento del cervello umano e dal modo in cui esso elabora i segnali. Il componente fondamentale di una Rete Neurale è il neurone. Molti neuroni vengono organizzati in una struttura interconnessa e compongono una Rete Neurale, la cui architettura prevede la distribuzione dei neuroni in livelli (layer o strati): a partire dallo strato di input i dati vengono fatti fluire in avanti attraverso i livelli successivi fino a raggiungere il layer di output: questo tipo di struttura è chiamata *feed-forward*.

Una Rete Neurale è composta da un livello di input, un livello di output, e da uno o più livelli nascosti; per ogni coppia di livelli adiacenti i neuroni

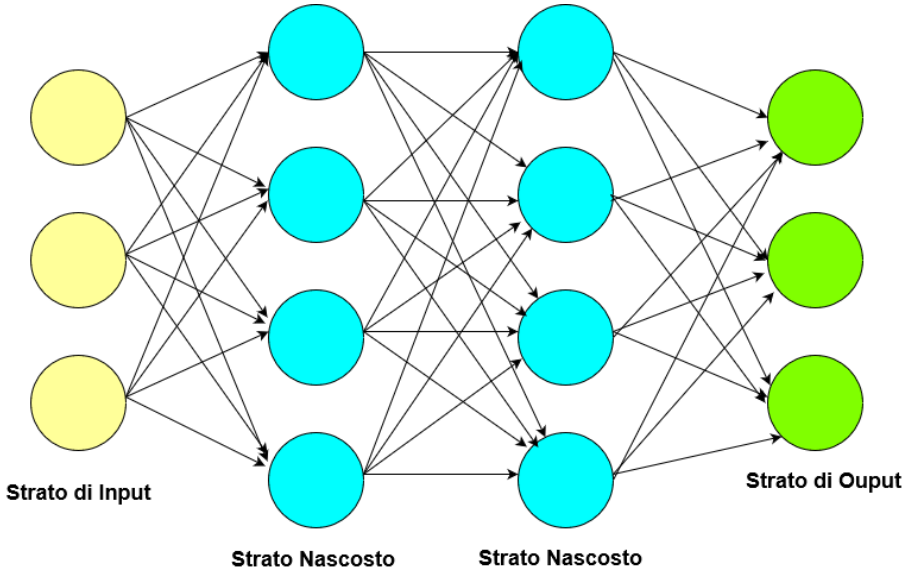

Figura 1.2: Struttura di una Rete Neurale totalmente connessa

del livello precedente sono connessi a quelli del livello successivo attraverso collegamenti a cui sono associati dei pesi di valore numerico, che rappresentano la forza della connessione tra neuroni. Il livello di input estrae le *feature* dai singoli casi e li immette nella rete, mentre quello di output produce il risultato. Ogni neurone dei livelli nascosti riceve in input una serie di valori ponderati, quindi li somma tra loro e si serve di una funzione di attivazione per elaborare il risultato, nella maggioranza dei casi in maniera non lineare.

Quando l'output di ogni neurone di un layer è funzione di tutti i neuroni del layer precedente, definiamo il layer "totalmente connesso", oppure come viene definito dalla celebre libreria Keras, strato Dense.

1.5.1 Deep Learning

Il *Deep Learning*, o apprendimento profondo, è un sottoinsieme del *Machine Learning* che si occupa di Reti Neurali complesse, dotate quindi di almeno 2 livelli nascosti, e di algoritmi per addestrarle. Da una prospettiva di alto livello, il *Deep Learning* riguarda la definizione di un modello che:

- proponga una soluzione di un problema,
- calcoli l'errore di ogni previsione,
- riduca l'errore con algoritmi di ottimizzazione, come ad esempio la discesa del gradiente.

1.6 Gli iperparametri di una Rete Neurale

La struttura di una rete neurale può essere compresa come una combinazione di iperparametri assegnati a priori che determineranno la correttezza del modello. Alcuni esempi di iperparametri da configurare sono il numero di livelli nascosti, la quantità di neuroni appartenenti a ciascuno di essi e il valore iniziale dei pesi; scelte che dovranno essere ponderate con attenzione per ottenere i risultati attesi senza appesantire eccessivamente il modello.

1.6.1 La funzione di attivazione

Una rete neurale con molti layer tipicamente prevede diverse funzioni di attivazione: se quelle di tipo lineare sono usate di rado per il fatto che non applicano alcuna trasformazione e quindi riducono una rete neurale a una semplice regressione, le funzioni non lineari permettono alla rete di modellare la relazione non lineare tra le *feature* e l'obiettivo. Una funzione di attivazione deve inoltre essere monotona, in modo che possa "conservare" le informazioni sui valori che le sono stati passati.

Sigmoid

$$\frac{1}{1+e^{-x}} \tag{1.1}$$

Sigmoid è universalmente riconosciuta come una buona funzione di attivazione: è monotona e non lineare e soprattutto ha un azione "normalizzante" sul modello perchè costringe le *feature* a compattarsi entro un intervallo finito

tra 0 e 1. Purtroppo ha anche un aspetto negativo: a causa del fatto che produce gradienti piuttosto bassi renderà la fase di apprendimento piuttosto lenta, soprattutto se usata in diversi livelli della rete.

ReLu

$$\begin{cases} 0 \; se \; x < 0 \\ x \; altrimenti \end{cases} \tag{1.2}$$

ReLu è una funzione di attivazione non lineare e monotona molto popolare perchè, al contrario di Sigmoid, risulta piuttosto efficiente in termini di tempi di apprendimento. Come aspetto negativo è evidente la distinzione troppo netta e arbirtaria tra valori inferiori o superiori di 0.

Tanh

$$\frac{sinh(x)}{cosh(x)} \tag{1.3}$$

Tanh realizza un ottimo compromesso tra le funzioni viste finora, la sua forma è molto simile a Sigmoid con la differenza che compatta gli input nell'intervallo (-1, 1). Questa differenza produce gradienti più ripidi di Sigmoid e ne riduce sensibilmente i tempi di addestramento

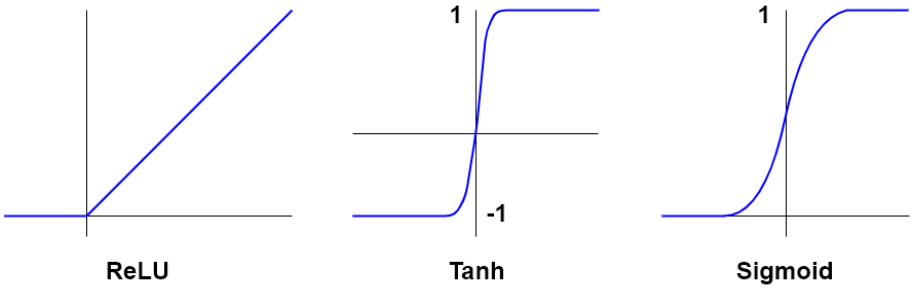

Figura 1.3: I grafici di ReLU, Tanh e Sigmoid

1.6.2 La Funzione di Costo

Nei problemi di regressione la Funzione di Costo che si preferisce utilizzare è lo scarto quadratico medio (*Mean Squared Error*), una funzione convessa con un grande vantaggio: tanto più la previsione è lontana dal valore corretto, maggiore è il gradiente ricevuto dai parametri, e quindi la ripidità di discesa nella ricerca del minimo. Nei problemi di classificazione, in cui l'output cercato può assumere il valore di 0 o 1, viene spesso preferita la funzione di scarto entropia incrociata (*Cross entropy*), in questi casi l'errore tende all'infinito man mano che la differenza tra la previsione e l'obiettivo si avvicina a 1 e, producendo gradienti più ripidi, rende molto più efficiente il training della rete.

1.6.3 Optimizer

Un *Optimizer* è l'algoritmo che viene scelto allo scopo di convergere al valore minimo della funzione di costo durante la fase di addestramento. Il Metodo di Discesa del Gradiente è una tecnica indispensabile, ma in reti neurali particolarmente profonde e in presenza di un numero molto alto di parametri da ottimizzare è necessario introdurre delle estensioni, come ad esempio Adam, per raggiungere risultati ottimali.

1.6.4 Learning Rate

Il tasso di apprendimento è in genere un numero molto piccolo e generalmente compreso tra 0.001 e 0,05 e rappresenta la grandezza con cui verranno sommati i pesi correnti in modo da addestrare il modello sui valori aggiornati.

1.7 Backpropagation

Il naturale flusso "in avanti" di una rete neurale raggiungerà ad un certo punto il layer di output che produrrà una previsione. Nei casi di apprendimento supervisionato l'algoritmo confronta questo valore con il risultato

atteso attraverso la funzione di costo che, a sua volta, elabora un certo indice di errore. L'entità di questo scarto può essere ridotto modificando i parametri dell'algoritmo, ma trovare una regola di aggiornamento è tutt'altro che semplice, considerando che l'architettura di una rete neurale è complessa e le connessioni di un livello dipendono da come il livello precedente ha ricombinato gli input. L'algoritmo di *Backpropagation* (o retropropagazione) è una tecnica intelligente per propagare gli errori "all'indietro", ridistribuirli nella rete fra tutte le sue unità e permettere il corretto aggiornamento dei pesi delle connessioni tra layer consecutivi.[15] Questa procedura è ripetuta per ogni esempio del dataset di addestramento e raggiunge il suo compito minimizzando l'errore prodotto dalla funzione di costo. L'aggiornamento dei pesi può avvenire in 3 diverse modalità:

1. Online: l'aggiornamento dei pesi viene effettuato dopo che ogni caso di esempio ha attraversato la rete. L'algoritmo in questo modo apprende in tempo reale e con limitato uso della memoria, ma soffre la presenza di valori anomali, questo impone un tasso di apprendimento estremamente basso che allunga i tempi di tutta la procedura.

2. Batch: l'aggiornamento dei pesi viene effettuato dopo aver osservato tutti i casi di esempio del set di training. E' solitamente la modalità più efficiente e per questo è spesso utilizzata per problemi standard.

3. Mini-Batch o stocastica: Il dataset di addestramento viene suddiviso in pacchetti e l'aggiornamento dei pesi avviene dopo che la rete ha elaborato ognuno di questi pacchetti di esempi dal set di addestramento

1.7.1 Problema della Scomparsa o dell'Esplosione del Gradiente

Durante la retropropagazione il gradiente può raggiungere valori infinitesimamente piccoli, o molto elevati, legati soprattutto alla presenza di funzioni di attivazione non lineari in molti livelli, che può portare rispettivamente

alla scomparsa o all'esplosione del Gradiente, un problema dovuto alla limitata rappresentazione numerica dei calcolatori che in questi casi genera underflow/overflow e quindi ignora l'aggiornamento dei pesi.

1.8 Transfer Learning

Il *Transfer Learning* è una tecnica molto diffusa che consente di trasferire le conoscenze apprese da un modello addestrato di *Machine Learning* per risolvere problemi di tipo diverso. Ad esempio, se si addestra un classificatore in grado di rilevare se un immagine contiene uno zaino, è possibile servirsi delle conoscenze acquisite dal modello per individuare altri tipi di oggetti come occhiali da sole. La finalità è quindi di utilizzare un modello addestrato per un'attività con molti esempi a disposizione, in una nuova attività che non dispone della stessa quantità di dati ma è in un certo senso correlata.

1.8.1 Fine-Tuning

La tecnica più conosciuta quando si parla di *Transfer Learning* è il *fine-tuning*, che generalmente viene eseguito in 4 passi.[23]

1. Si definisce il modello sorgente, addestrando una rete neurale sul dataset ricco ed etichettato di cui si dispone.

2. Si definisce il modello obiettivo, un nuovo modello di rete neurale costruito in modo identico al modello sorgente, per architettura e parametri, ad eccezione dello strato di output.

3. Si implementa lo strato di output sul modello obiettivo, inizializzando i suoi parametri in modo casuale.

4. Si addestra il modello obiettivo con il dataset meno ricco. Lo strato di output viene addestrato da zero, mentre i parametri di tutti gli altri livelli verranno appunto "ritoccati"

1.9 Reti Neurali Convoluzionali

Le Reti Neurali Convoluzionali (CNNs) sono una famiglia di algoritmi largamente diffusi e che oggi raggiungono lo stato dell'arte in numerose applicazioni di *Deep Learning* come la classificazione di immagini, l'*object detection* e il riconoscimento vocale. Ciò che maggiormente contraddistingue una CNN dalle reti neurali viste finora sono le operazioni tipiche di questa architettura, come la convoluzione e il raggruppamento, oltre alla presenza di un numero sensibilmente minore di parametri, che permette di costruire reti molto profonde senza preoccuparsi eccessivamente di possibili overflow di memoria.[21]

L'architettura di una Rete Neurale Convoluzionale è strutturata su un insieme di livelli di tipo diverso, in cui troviamo strati convoluzionali, strati di raggruppamento o di pooling, e strati totalmente connessi. Le CNNs si rivelano particolarmente efficienti per emettere previsioni quando le osservazioni in input sono immagini, sfruttando un concetto chiave in questo tipo di applicazioni, ovvero che i pattern interessanti provengono da pixel adiacenti.

Il funzionamento di queste reti prevede che i livelli convoluzionali facciano scorrere una piccola matrice di pesi attraverso la struttura dati di input (ad esempio una matrice in grado di mappare i pixel di una immagine) tramite l'operazione di convoluzione che produrrà a sua volta un output dalla forma del tutto simile, chiamato mappa delle caratteristiche. La matrice dei pesi rappresenta un "rilevatori di schemi" o filtro convoluzionale, e il suo prodotto scalare, preso con i valori dei pixel in ogni posizione dell'immagine, indica se il modello visivo che stiamo cercando è presente nella porzione di immagine e, in caso positivo, produrrà un valore alto. Questa operazione prende il nome di convoluzione. [9]

I livelli convoluzionali possono essere accoppiati a strati di raggruppamento allo scopo di ridurre la dimensione dell'input tramite un sottocampionamento della mappa delle caratteristiche prodotta dalla convoluzione.

[1]from Natural Language Processing with TensorFlow. Published by Packt Publishing. Used with permission.

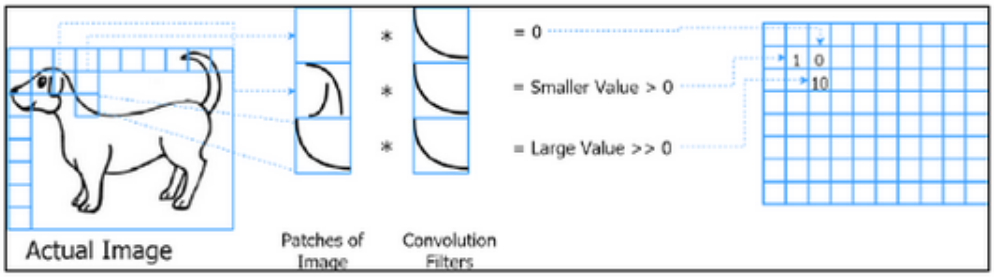

Figura 1.4: Un'operazione di Convoluzione applicata ad un'immagine[1]

L'input viene quindi suddiviso in piccole porzioni, solitamente matrici 2x2, che vengono rimpiazzate dal valore massimo nel caso di raggruppamento massimo, o dal valore medio nel caso di raggruppamento medio. Lo scopo di questa operazione è di forzare in qualche modo una CNN ad imparare con una notevole riduzione della computazione, considerando che la perdita di informazioni generata dal sottocampionamento non sembra compromettere la sua efficacia.

Solitamente il livello di output è preceduto da un insieme di strati totalmente connessi, allo scopo di fornire una visione globale su come le caratteristiche locali apprese dai livelli convoluzionali possono essere connesse tra loro per ottenere un output attendibile.

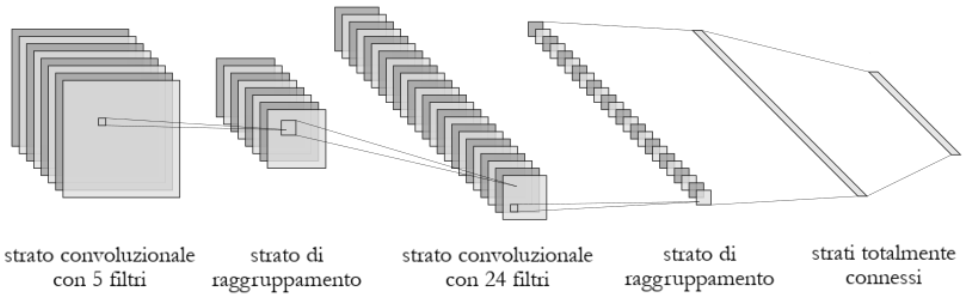

Figura 1.5: Una tipica architettura CNN

1.10 Reti Neurali Ricorrenti

Le Reti Neurali Ricorrenti, RNN, sono una classe di algoritmi dedicati alla gestione di sequenze di dati, ovvero quando il risultato di una computazione dipende dalle informazioni osservate negli istanti precedenti. Se le Reti Neurali analizzate sinora trattano i dati in input come un insieme di osservazioni indipendenti tra loro, le RNN sono progettate per esaminarli in un determinato ordine temporale, come ad esempio l'andamento del prezzo di un terreno nell'arco del tempo o la prossima parola all'interno di una frase. Anche la forma dell'input sarà necessariamente diversa, e caratterizzata da una dimensione in più; se nelle reti neurali feed forward ogni osservazione poteva essere strutturata in un *array* di n *features*, nelle RNN si necessita di una matrice bidimensionale di n caratteristiche per t fasi temporali.

L'architettura delle Reti Neurali Ricorrenti è caratterizzata dalla presenza di una memoria di stato che durante la computazione cattura i diversi pattern e permette di modellare un comportamento temporale in base alle informazioni ricevute precedentemente. La cella è quindi una componente di una RNN che preserva lo stato interno in ogni istante t, ed è composto da un determinato numero di neuroni.

Per capire il funzionamento di una rete neurale ricorrente assumiamo di avere due funzioni: f_1 che prende l'input attuale e lo stato all'istante precedente e genera lo stato attuale; e f_2 che riceve lo stato attuale e genera l'output per l'istante corrente.

$$h_t = f_1(x_t, h_{t-1}) \quad \text{e} \quad y_t = f_2(h_t)$$

e quindi

$$y_t = f_2(f_1(x_t, h_{t-1})) \tag{1.4}$$

La rappresentazione grafica può quindi essere riassunta nella Figura 1.6.

Nella parte a sinistra della Figura 1.7 è possibile osservare la generalizzazione della computazione di un singolo passo di una RNN per ogni istante t.

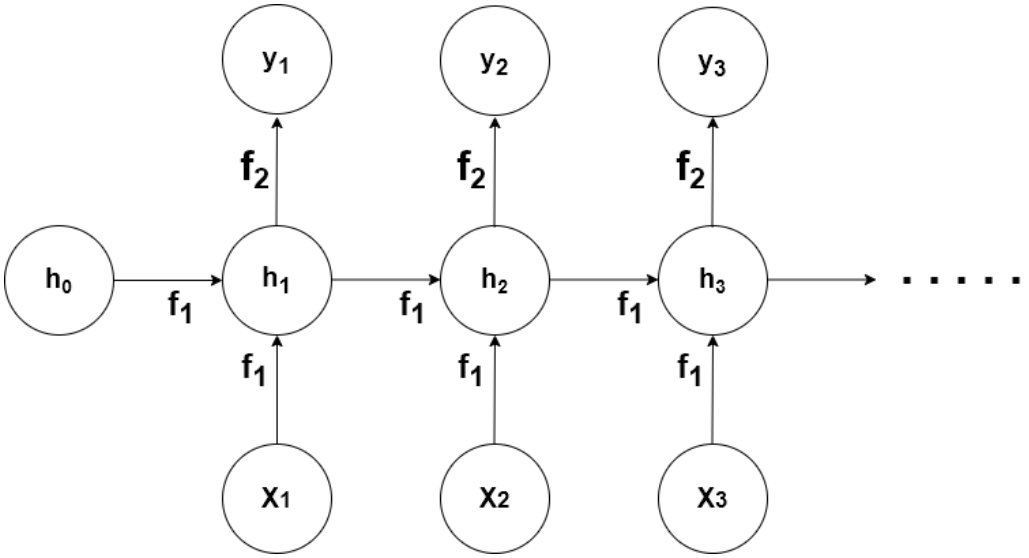

Figura 1.6: La relazione tra input ed output per ogni istante t

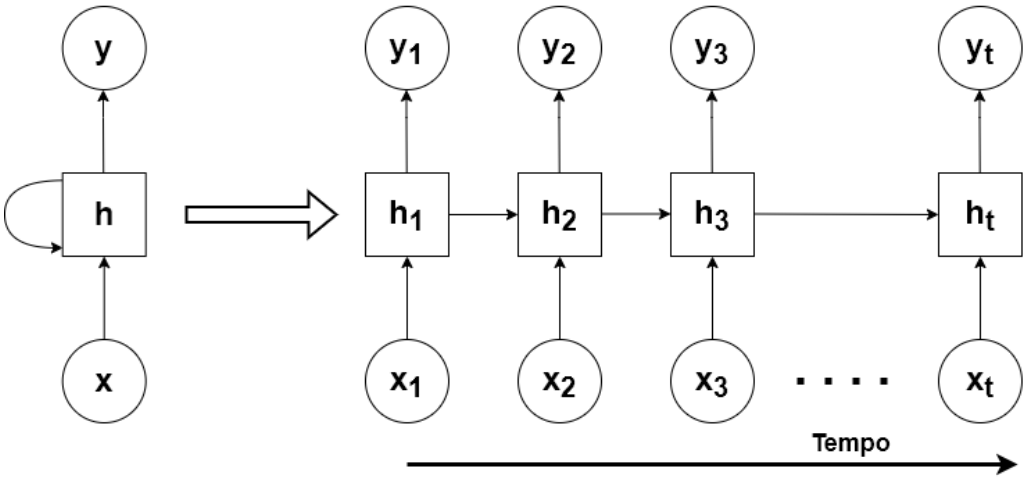

Figura 1.7: Unfolding della RNN in t step

Per addestrare una Rete Neurale Ricorrente è necessario eseguire una procedura chiamata *unfolding* [2], stabilendo a priori il numero di step temporali, in questo modo si può notare che il caso di 10 passi equivale ad una rete *feed-forward* con 10 layer.

1.10.1 Backpropagation through time

Per completare l'addestramento di una RNN la propagazione a ritroso avviene attraverso una speciale procedura chiamata *Backpropagation through time*, una procedura piuttosto complicata in cui i gradienti vengono fatti scorrere all'indietro attraverso la rete nell'ordine inverso rispetto a quello in cui gli input sono stati passati in avanti.

1.10.2 Applicazioni delle RNN

Il concetto fondamentale delle reti neurali ricorrenti riguarda il fatto che permettono di operare su sequenze di vettori, in input, in output, o più in generale in entrambi i casi. [7]

- **One to One**: Queste reti prendono un singolo input e producono un singolo output. in questi casi l'input attuale dipende dagli input precedentemente osservati.
- **One to Many**: Attraverso un singolo input viene generata una sequenza composta da un numero arbitrario di elementi. Assumiamo in questi casi che ogni input sia indipendente dagli altri e l'output sia formato da una sequenza di valori che dipendono dai valori degli output precedenti.
- **Many to One**: La rete riceve in input una sequenza di dimensione arbitraria e produce un singolo output.
- **Many to Many**: Queste reti prendono in input una sequenza di lunghezza arbitraria e producono una sequenza di dimensione arbitraria.

Capitolo 2

Elaborazione del Linguaggio Naturale

2.1 Introduzione

In qualità di essere umani, tra le primissime abilità che riusciamo a conquistare c'è la padronanza del linguaggio, ovvero comprendere l'associazione tra una parola e il suo significato e trasmettere in modo chiaro ciò che vogliamo comunicare ad un interlocutore. I computer, come ben sappiamo, usano al loro interno solamente numeri, ma il loro modo di comunicare con gli esseri umani è oggetto di studio sin dal 1950, quando Alan Turing concepì il suo celebre test per l'identificazione di una intelligenza artificiale.

Grazie ai numerosi studi realizzati in questi anni, oggi una macchina è pienamente in grado di comunicare grazie all'avvento dell'elaborazione del linguaggio naturale (*Natural Language Processing*, NLP), un ramo dell'informatica che si occupa di comprensione e generazione dei linguaggi comprensibili dall'uomo. Nella nostra vita quotidiana ci troviamo molto spesso ad interagire direttamente con applicazioni che fanno uso di NLP: i filtri antispam sulla posta elettronica, i traduttori automatici, gli assistenti vocali sui dispositivi mobili, il sistema di raccomandazioni che troviamo nei principali siti di *e-commerce*, sono solo alcuni esempi di questa tecnologia al lavoro e

delle sue enormi potenzialità. [22] In questo elaborato, mi concentrerò su un task particolarmente interessante e ricco di sfide: la generazione di testo.

2.2 Modelli Linguistici

Visto ad alto livello, il linguaggio degli esseri umani è composto da una sequenza ordinata di parole che nel loro complesso formano una frase, cioè generano significato. Per un computer riprodurre questo principio è un compito assai difficile. Un impianto puramente statistico infatti non è sufficiente a svolgere tale attività, ma deve essere in grado di catturarne le proprietà più profonde, per risolvere le ambiguità ed esprimere concetti coerenti. Il Language Modeling è un attività di *Machine Learning* con un compito piuttosto chiaro: fornire una distribuzione di probabilità su sequenze di simboli relativi ad un linguaggio [14], e in questo modo addestrare modelli linguistici per compiere previsioni sulla prossima parola o carattere in un documento. L'obiettivo è quindi stimare le probabilità di frammenti di testo in modo tale da riflettere la conoscenza di un linguaggio, ovvero fare in modo che ad una frase coerente con le regole di tale linguaggio venga attribuita una probabilità maggiore.

2.2.1 Addestramento e Generazione

Per calcolare la corretta probabilità di frammenti di testo è necessario eseguire la decomposizione in elementi più piccoli, come ad esempio parole; in questo modo la probabilità di una frase può essere costruita in modo iterativo dove ad ogni passo si calcola la probabilità del termine successivo in base al contesto precedente.

$$P(x_i|x_1, x_2, ...x_{i-1}) \tag{2.1}$$

A seconda delle modalità con cui vengono effettuati questi calcoli si possono distinguere 2 famiglie di modelli linguistici: N-gram e Neurali.

N-gram Language Model

Questi modelli producono distribuzioni di probabilità basati su valori statistici calcolati sul corpus di testo attraverso i suoi componenti principali:

- Proprietà di Markov: si assume che la probabilità di un termine dipenda solamente da un numero n prefissato di parole che lo precedono. Quindi $\mathrm{P}(x_i|x_1,...x_{i-1}) = \mathrm{P}(x_i|x_{i-n+1},...x_{i-1})$;
- Regolarizzazione: si applicano opportunamente alcune tecniche di smoothing per garantire risultati consistenti
  - Algoritmo di backoff: se l'n-gramma non appare nel corpus, si riduce il numero di elementi di una unità, iterando il procedimento fino a quando la nuova sequenza viene rilevata nel testo
  - Laplace smoothing: un valore piccolo viene sommato ad ogni n-gramma, in questo modo l'algoritmo assegnerà una probabilità anche a sequenze corrette che non compaiono nel corpus

La fase generativa avviene in modalità iterativa, in cui ad ogni passo il modello elabora il contesto attuale e stima una distribuzione di probabilità per il termine successivo, che viene scelto a campione e accodato alla sequenza generata. [20] Semplicità e scalabilità sono i principali vantaggi dei modelli N-gram che però, a causa della dimensione limitata del contesto, non sono in grado di intercettare le dipendenze a medio-lungo termine.

Neural Language Model

Questo approccio combina le capacità di apprendimento di una rete neurale con il concetto di rappresentazione vettoriale e codifica del contesto, producendo una distribuzione delle probabilità in base all'attinenza con i termini precedenti [13].

Durante il processo di addestramento:

- la sequenza di testo in input viene modellata con *Word Embedding* e passata alla Rete Neurale;

- al termine della computazione la Rete Neurale produce la rappresentazione vettoriale del contesto e, attraverso uno strato Dense, viene stimata la probabilità per ogni elemento del vocabolario;
- ad ogni passo il modello deve aggiornare i pesi in modo da massimizzare la probabilità dell'elemento corretto.

In fase generativa un approccio Greedy non sempre rappresenta la soluzione migliore, in questi casi infatti si presentano molto spesso ripetizioni e cicli, mentre il modello deve essere in grado di generare testo che sia diverso ma soprattutto abbia senso. Per questo vengono preferite strategie diverse, come il tuning del parametro temperature, che ridefinisce le differenze di probabilità fra i vari termini, o il Top-K sampling, in cui la scelta del termine viene effettuata tra i K candidati con maggiori probabilità, o infine Nucleus sampling che prevede di restringere la scelta tra i termini che, sommati tra loro, rappresentano la probabilità maggiore [11].

2.3 Text Generation

La generazione automatica di testo, o più formalmente "generazione di linguaggio naturale" è un settore del *Natural Language Processing* che ha lo scopo di produrre in modo artificiale dei contenuti testuali di qualità indistinguibili da quelli umani. Questo compito può essere realizzato attraverso modelli generativi profondi come LSTM o reti generative avversarie (GAN); ma grazie all'avvento delle architetture *Transformer* negli ultimi anni sono stati fatti passi da gigante, e oggi i modelli BERT e GPT, basati su questa tecnologia, rappresentano lo stato dell'arte in questo settore. [12]

2.3.1 Valutazione e Metriche

Perplexity

La generazione automatica di testo è un'attività molto complessa, così come la ricerca di un metodo che sia in grado di valutare la qualità di un

modello generativo. Allo stato attuale, una procedura molto diffusa prevede di misurare la Perplessità del modello (*Perplexity*), ovvero come la rete interpreta il testo prodotto in base all'input. Quindi, se consideriamo la funzione di costo per un input x_i e il suo corrispondente y_i come l(x_i,y_i), allora la formula per il calcolo della Perplessità è

$$p(x_i, y_i) = e^{l(x_i,y_i)} \tag{2.2}$$

In questo modo è quindi possibili calcolare la Perplessità media per un dataset di addestramento di dimensione n come:

$$p(D_{train}) = (1/n) \sum_{i=1}^{n} p(x_i, y_i) \tag{2.3}$$

Quindi in linea generale possiamo affermare che minore è l'indice di perplessità del modello, maggiore è la qualità del testo prodotto.

2.4 Algoritmi di NLP

In NLP è di fondamentale importanza la fase di preprocessing dei dati, fase in cui il dataset viene ripulito dalle ridondanze e normalizzato prima di essere passato ad uno dei numerosi algoritmi di NLP. [4]

2.4.1 BoW (Bag of Words)

Questo modello considera la parole come un insieme e segnala se ognuna di esse è presente nel testo assegnandole un valore numerico all'interno di una struttura dati. BoW non tiene alcuna traccia delle parole adiacenti e questo lo rende un algoritmo estremamente semplice ma di efficacia piuttosto limitata.

2.4.2 N-gram

n-gram è una tecnica per creare un vocabolario in cui vengono raggruppate le parole adiacenti tra loro e si fonda sul concetto di *token*, come singolo elemento del testo. Un n-gramma è quindi una sequenza continua composta da n *token* del testo che viene utilizzata come singola unità nella rappresentazione.

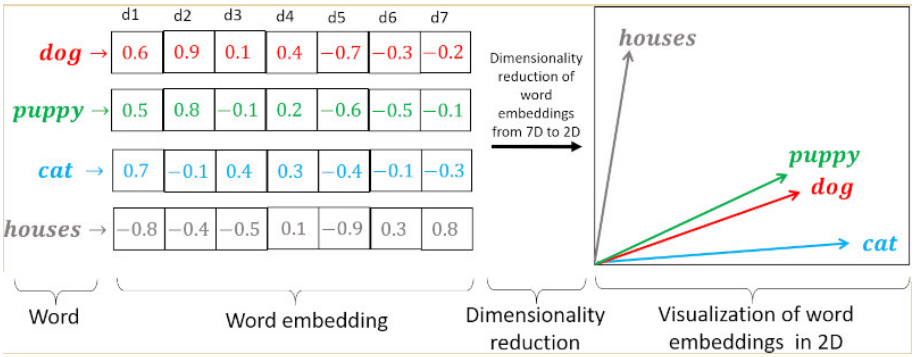

Figura 2.1: Word Embedding e lo spazio vettoriale[1]

2.4.3 TF-IDF

La trasformazione TF-IDF è molto importante per estrarre la rilevanza di un *token* in un contesto. Rappresenta il rapporto tra la frequenza in cui un termine compare (*Term Frequency*) e il numero di documenti in cui appare (*Document Frequency*). In linea generale, questo modello ritiene le parole rare più rilevanti rispetto a quelle che compaiono con maggior frequenza.

2.4.4 Word Embedding

Embedding è un vettore di dimensione predefinita che codifica e rappresenta un'entità, una frase, una parola o un documento. Word Embeeding è una tecnica di modellazione in cui i componenti di un vocabolario vengono mappati in vettori sparsi di numeri reali. Lo scopo di questa tecnica è di creare uno spazio vettoriale in cui le parole che occorrono negli stessi contesti sono più vicine tra loro. Gli algoritmi di questo tipo più popolari sono Word2vec e GloVe.

[1]David Rozado. *Word embeddings map words in a corpus of text to vector space.* 2020. (modificato)

2.5 Long Short-Term Memory

LSTM, *Long Short-Term Memory*, è una speciale tipologia di Rete Neurale Ricorrente particolarmente performante nel campo del NLP, grazie alla capacità di gestire dipendenze a lungo termine, ovvero quando in un documento la distanza tra un informazione rilevante e il contesto in cui è richiesta è particolarmente elevata. Le celle base di una RNN hanno in genere grosse difficoltà a ricordare gli input di passi lontani, che infatti nel tempo tendono a svanire; LSTM al contrario incorpora un sistema di celle molto più evoluto che gli permette di realizzare una memoria a lungo termine, grazie al quale registra in genere performance superiori, specialmente in caso di una notevole quantità di dati a disposizione. [8]

Nelle architettura di questi modelli lo stato interno è suddiviso in 2 unità, h_t e c_t, che rappresentano rispettivamente la memoria a breve e a lungo termine, mentre la fase di addestramento è guidata da 3 componenti molto importanti: l'*Input Gate*, il *Forget Gate* e l' *Output Gate*.

2.5.1 Funzionamento di LSTM

Nella figura 2.1 è descritto il funzionamento di una LTSM durante un istante temporale t:

- durante la fase di training lo stato a lungo termine c_t attraversa il *Forget Gate* e stabilisce cosa può dimenticare dallo stato precedente c_{t-1};
- la memoria a breve termine mediante l'*Input Gate* estrae e aggiunge informazioni dall'input corrente;
- tramite l'*Output Gate* viene prodotto l'output combinando l'input corrente con informazioni estratte dalla memoria a lungo termine.

[2]from Hands-On Machine Learning with Scikit-Learn and TensorFlow, by Aurélien Géron. Copyright © 2017 Aurélien Géron. Published by O'Reilly Media, Inc. Used with permission.

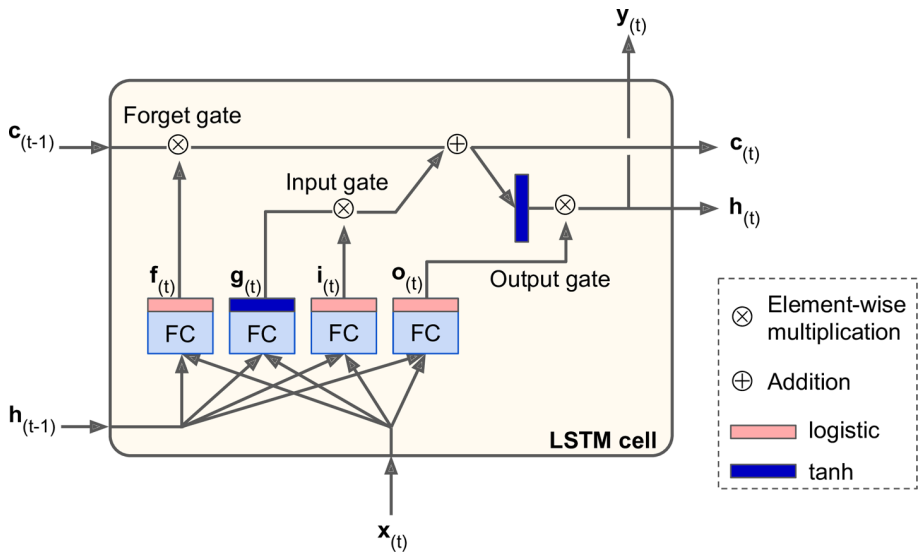

Figura 2.2: Cella LSTM all'istante t[2]

Tra i vantaggi principali di queste architetture, la capacità di evitare il problema della scomparsa del gradiente permette di mantenere le informazioni in memoria per centinaia di passi temporali, ben oltre la capacità delle tradizionali RNN.

Una tecnica per migliorare la capacità predittiva è di rendere il modello bidirezionale, ovvero costruita su due reti LSTM distinte, una tradizionale che apprende i dati dall'inizio alla fine, e una seconda contraria, che quindi segue la direzione opposta.

Le prestazioni delle Reti LSTM sono davvero ottime, se si esclude una certa lentezza nella fase di addestramento dovuta alla natura sequenziale del modello, e per questo hanno rappresentato la prima scelta in moltissime applicazioni NLP fino dell'avvento dei modelli *Transformer*.

2.6 GRU

Gated Recurrent Unit può essere considerata come una versione semplificata di una architettura standard di LSTM in cui l'obiettivo è di alleggerire il modello e il numero di parametri, mantenendo allo stesso tempo i benefici.

La differenza sostanziale di questa architettura riguarda la cella di memoria, che in questi modelli torna unica, così come il numero di gate in cui è possibile aggiungere e rimuovere informazioni. In GRU infatti l'input e il *Forget Gate* vengono combinati tra loro in un unico update gate; mentre tramite un reset gate la rete stabilisce quali informazioni contenute nello stato precedente sono importanti per la computazione.

2.7 GAN

GAN, *Generative Adversarial Network*, è un algoritmo di *Deep Learning* molto popolare e caratterizzato da un approccio "avversario" molto diverso da quello assunto dalle reti neurali tradizionali. Le Reti GAN sono costituite principalmente da 2 modelli che vengono addestrati in modalità "avversaria": un generatore a cui è affidato il compito della produzione di campioni di dati, ed un discriminatore a cui è richiesto di classificare questi dati tra ciò che considera corretto, ovvero i dati di addestramento, e ciò che ritiene errato, quindi i dati prodotti dal generatore. L'obiettivo di quest'ultimo è produrre dati talmente simili a quelli reali da ingannare il discriminatore e fare in modo che vengano etichettati come corretti.

Questa famiglia di reti ha avuto un enorme successo soprattutto nelle applicazioni in cui è richiesta la generazione di un'entità di cui non disponiamo, come ad esempio la rappresentazione del viso di una persona dopo molti anni, ma ha avuto notevoli sviluppi anche nel campo del NLP e del *Text Generation* grazie all'applicazione dell'addestramento per rinforzo.

2.8 Transformer

2.8.1 Attenzione è tutto ciò di cui hai bisogno

Nel 2017 i ricercatori di Google pubblicarono un articolo dal titolo "Attention is All you need" che al suo interno proponeva una nuova e rivoluzionaria

architettura di rete, *Transformer*. [19] Questi modelli si basano sul rivoluzionario concetto di *autoattenzione*, che si è rivelato particolarmente efficace in ambito NLP, raggiungendo lo stato dell'arte in numerose applicazioni e soppiantando in brevissimo tempo i precedenti modelli di CNN e RNN usati a tale scopo. [16]

Se un RNN esamina la sequenza di input in modo sequenziale, una parola alla volta, i modelli *Transformer* sono in grado di elaborare tutte le parole della sequenza in parallelo, accelerando sensibilmente la computazione. I modelli *Transformer* operano principalmente con dati testuali di natura sequenziale, ricevono una sequenza in ingresso e producono una sequenza di output, ad esempio nella generazione di testo possono espandere una frase passata in input.

2.8.2 Architettura

L'architettura di un modello *Transformer* comprende uno stack di layer di *encoder* e di layer di *decoder*, ognuno dei quali dotato del proprio insieme di pesi.

L'*encoder* contiene il layer di autoattenzione, che calcola le relazioni tra le diverse parole dell'input, e un tradizionale strato feed forward; entrambi succeduti da un layer "Add & Norm" allo scopo di normalizzare gli output.

Il *decoder*, oltre al layer di autoattenzione e *feed-forward*, include un ulteriore livello di attenzione *Encoder-Decoder*.

2.8.3 Il meccanismo di AutoAttenzione

La chiave delle straordinarie performance dei modelli *Transformer* nelle applicazioni NLP è senza dubbio legata al concetto di "Attenzione", un meccanismo che durante l'elaborazione di una parola permette al modello di concentrarsi su tutte le altre parole che compongono l'input che sono in stretta relazione con quella che si sta processando.

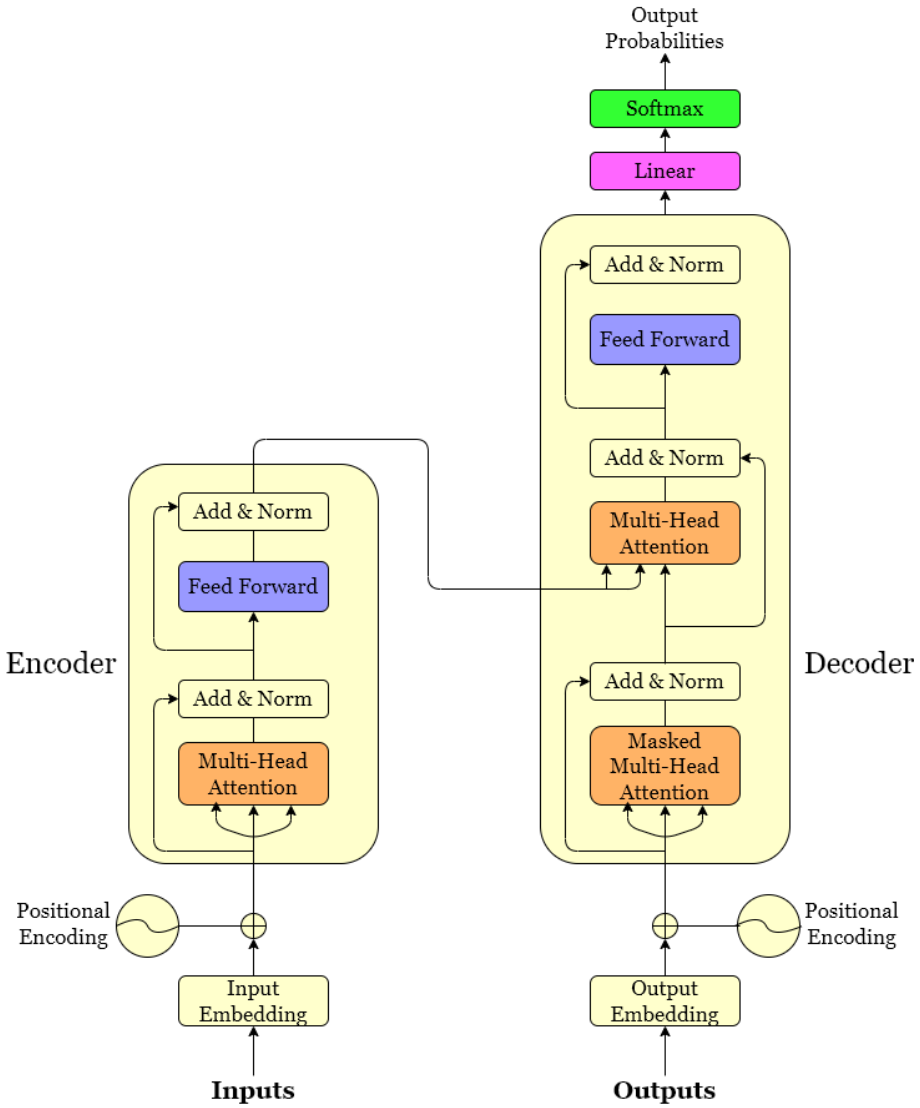

Figura 2.3: Architettura Transformer

Analogamente, il meccanismo di autoattenzione consiste nel mettere in relazione tra loro tutte le parole che compongono l'input, impostando un sistema di punteggi (attention score) che assegna valori più alti in base alla correlazione.

2.8.4 BERT

BERT, *Bidirectional Encoder Representations from Transformers*, è un modello innovativo realizzato da Google e destinato ad applicazioni di molteplici ambiti nel campo NLP. [6]

L'architettura di BERT è del tutto analoga ai modelli *Transformer*, e implementa multipli strati di *encoder* unitamente al meccanismo di autoattenzione per generare modelli linguistici. Rispetto ai modelli unidirezionali, che in genere elaborano le sequenze da sinistra verso destra, l'*encoder* di BERT è bidirezionale e consente al modello di apprendere il contesto di una parola da entrambe le direzioni del documento.

Il preaddestramento viene effettuato su 2 applicazioni caratterizzate da modalità di apprendimento non supervisionato, *Language Modelling* e *Next Sentence Prediction*, ed avviene servendosi esclusivamente di dati grezzi; al termine di questa operazione computazionalmente molto dispendiosa, è possibile eseguire il *fine-tuning* con risorse limitate e dataset di dimensioni ridotte, in modo da adattare le performance del modello in diversi ambiti NLP. Sarà infatti necessario un solo strato di output aggiuntivo per eseguire il finetuning sul modello preaddestrato e realizzare modelli performanti per un'ampia varietà di attività linguistiche.

2.8.5 GPT-3

Generative Pre-trained Transformer 3, meglio conosciuto come GPT-3 è un modello linguistico creato da OpenAI nel 2020 allo scopo di risolvere diverse classi di problemi di NLP.

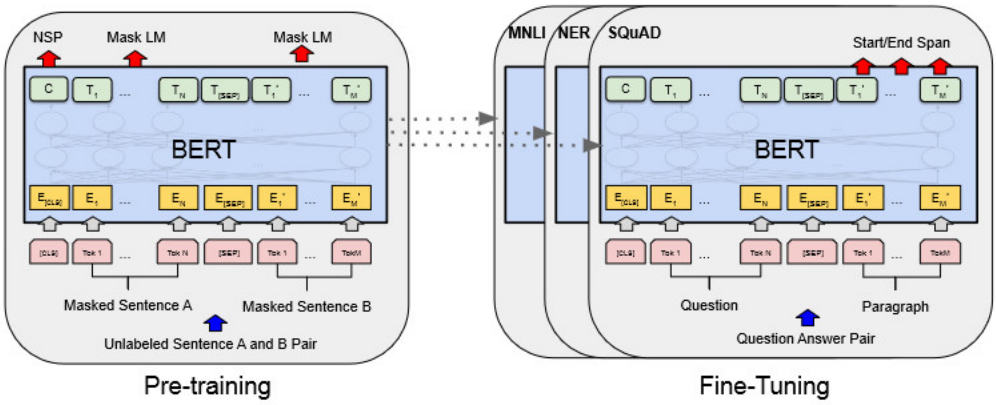

Figura 2.4: Le procedure di preaddestramento e fine-tuning per BERT

Stato dell'arte per il *Text Generation*, e ultima produzione in ordine temporale della famiglia GPT, questi modelli fanno uso della tecnologia *Transformer* per produrre sequenze di testo di altissima qualità e per molti aspetti indistinguibili dallo stile umano.

Dal punto di vista architetturale questa terza famiglia di OpenAI eredita dal suo predecessore e dai modelli basati su *Transformer* gran parte della sua struttura: GPT-3 è infatti costituito esclusivamente da uno stack di blocchi *decoder*, mentre fa uso di strati alternativi rispetto all'*encoder*, e grazie ad una nuovo meccanismo chiamato "*Sparse Transformers*" è in grado di definire il concetto di Attenzione in tempi straordinariamente rapidi. [3] I modelli GPT sono stati addestrati per svolgere un compito estremamente semplice: data una sequenza di testo, devono predire quale sarà la parola successiva.

Sono stati realizzati 8 modelli di complessità crescente, in grado di addestrare da un minimo di 125 milioni fino ad un massimo di 175 miliardi di parametri, usando dataset testuali di 45TB e composti da oltre 400 miliardi di *token*. Un modello GPT-3 ha recentemente scritto un editoriale sul quotidiano inglese "The Guardian" che ha attirato su di sé grandissime attenzioni e curiosità. [3]

[3]https://www.theguardian.com/commentisfree/2020/sep/08/robot-wrote-this-article-gpt-3

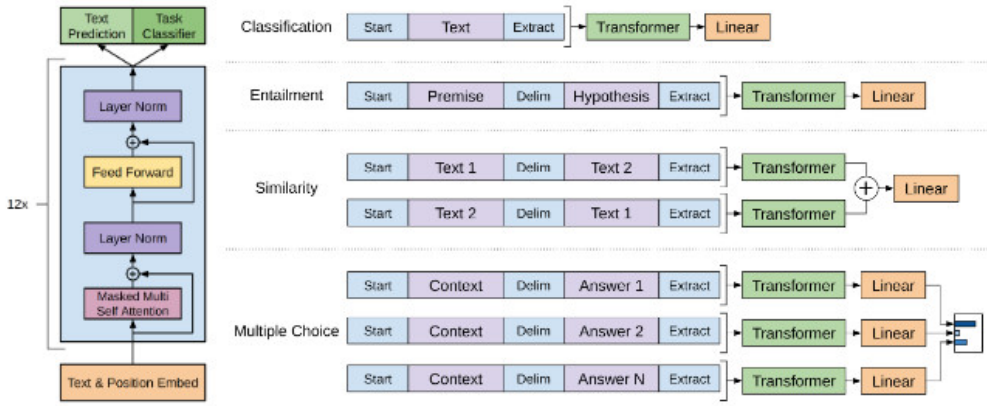

Figura 2.5: L'architettura transformer di GPT e le trasformazioni dell'input per il fine-tuning su diverse attività

Capitolo 3

Sperimentazione

3.1 Gererazione automatica di Testo a livello carattere con LSTM

Dopo uno studio approfondito su *Deep Learning* e sulle tecniche che governano le Reti Neurali più significative nel campo dell'elaborazione del linguaggio naturale mi sono dedicato alla progettazione di un modello di *Text Generation*.

Lo studio preliminare di questo tipo di applicazione si è rivelata inizialmente piuttosto ardua, i modelli *Transformer* mi attiravano molto ma la potenza di calcolo a mia disposizione non era sufficiente per costruire reti di questo tipo, straordinariamente performanti ma anche piuttosto costose in termini di risorse. I modelli più evoluti a disposizione degli sviluppatori sono preaddestrati su enormi dataset in lingua inglese [5], mentre la mia intenzione era di realizzare un sistema che producesse testo in lingua italiana. Avevo quindi due strade a disposizione, il *Training from scratch*, l'addestramento di una rete neurale *da zero*, e il *Fine-tuning*, entrambe però non praticabili per mancanza di risorse.

Ho scelto quindi di affidarmi alle solide performance delle reti neurali ricorrenti [10] e per la precisione la variante che più mi ha incuriosito durante la preparazione di questa tesi: LSTM.

3.2 Il modello

L'idea di base con cui ho addestrato una rete neurale LSTM per generare testo automatico a livello carattere è piuttosto semplice: passando in input una stringa di caratteri di dimensione arbitraria si chiede al modello di restituire la distribuzione delle probabilità del carattere immediatamente successivo rispetto alla sequenza di input. In questo modo il modello è in grado di generare testo un carattere alla volta. [14]

3.3 Preparazione e Dataset

Il dataset utilizzato per addestrare la rete è il romanzo completo "L'Isola del Tesoro" di Robert Louis Stevenson, reperito da Wikisource[1] in formato txt, ripulito dai crediti iniziali, dalle numerose tabulazioni non necessarie e da tutti i caratteri non ascii presenti nel testo. Il codice è stato realizzato in linguaggio Python ed eseguito sul servizio Colab di Google, per la parte programmatica ho utilizzato il Framework Tensorflow [18] e la libreria Keras.

```
import tensorflow as tf
from tensorflow.keras.layers.experimental import preprocessing
import numpy as np
import os
import time
import re
f = open("isola_del_tesoro.txt","r", encoding="utf8")
raw_text = f.read()
raw_text = re.sub(r'[^\x00-\x7f]',r'', raw_text)
vocab = sorted(set(raw_text))
```

Ho successivamente creato il vocabolario dei caratteri presenti sul Dataset (composto in questo caso da 81 caratteri unici)

[1] https://it.wikisource.org/wiki/L'isola_del_tesoro

3.4 Vettorializzazione

Il testo viene suddiviso in *token*: ogni carattere del vocabolario viene convertito in un valore numerico; e al momento della generazione di testo ognuno di questi sarà invertibile per recuperare da essi i caratteri corretti.

```
ids_from_chars = preprocessing.StringLookup(
    vocabulary=list(vocab), mask_token=None)
chars_from_ids = tf.keras.layers.experimental.preprocessing.StringLookup(
    vocabulary=ids_from_chars.get_vocabulary(), invert=True, mask_token=None)
ids = ids_from_chars(tf.strings.unicode_split(raw_text, 'UTF-8'))
chars = chars_from_ids(ids)
```

3.5 Sequenze di input e target

In questo modello gli esempi di training sono sequenze di caratteri del testo di lunghezza seq_length, mentre il *target* corrispondente è rappresentato da una sequenza di identica dimensione, ma spostata di un carattere in avanti. Quindi il testo viene trasformato in un flusso di singoli caratteri, si creano le sequenze di dimensione seq_length+1, che vengono poi suddivise in input e *target* di dimensione seq_lenght e accoppiate sul dataset per l'addestramento.

```
ids_dataset = tf.data.Dataset.from_tensor_slices(ids)
seq_length = 100
examples_per_epoch = len(raw_text)//(seq_length+1)
sequences = ids_dataset.batch(seq_length+1, drop_remainder=True)
def split_input_target(sequence):
    input_text = sequence[:-1]
    target_text = sequence[1:]
    return input_text, target_text
```

3.6 Definizione del Dataset per l'addestramento

I dati vengono mescolati e impacchettati e a questo punto sono pronti per essere presentati al modello

```
BATCH_SIZE = 64
BUFFER_SIZE = 10000
dataset = (
    dataset
    .shuffle(BUFFER_SIZE)
    .batch(BATCH_SIZE, drop_remainder=True)
    .prefetch(tf.data.experimental.AUTOTUNE))
```

3.7 Costruzione del Modello

Il modello è costituito da 3 layer:

- un layer *Embedding* di input per mappare ogni carattere ad un vettore sparso di dimensione embedding_dim=256;
- un layer LSTM con dimensioni rnn_units=1024;
- un layer Dense di output che produce la probabilità per ogni carattere del vocabolario di essere l'obiettivo.

```
vocab_size = len(vocab)
embedding_dim = 256
rnn_units = 1024

class MyModel(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim, rnn_units):
    super().__init__(self)
    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
```

```
    self.lstm = tf.keras.layers.LSTM(rnn_units,
                                     return_sequences=True,
                                     return_state=True)
    self.dense = tf.keras.layers.Dense(vocab_size)

  def call(self, inputs, states=None, return_state=False, training=False):
    x = inputs
    x = self.embedding(x, training=training)
    if states is None:
        states = self.lstm.get_initial_state(x)
        state_h = states[0]
        state_c = states[1]
    x, state_h, state_c = self.lstm(x, initial_state=[states[0], states[1]], tra
    x = self.dense(x, training=training)
    states = [state_h, state_c]

    if return_state:
        return x, states
    else:
        return x

model = MyModel(
    vocab_size=len(ids_from_chars.get_vocabulary()),
    embedding_dim=embedding_dim,
    rnn_units=rnn_units)
```

3.8 Funzionamento

Il carattere viene passato in input al livello *Embedding* che produce il vettore omonimo realizzando il "*char embedding*" e lo presenta come input alla rete LSTM che ad ogni passo temporale applica lo strato Dense per generare

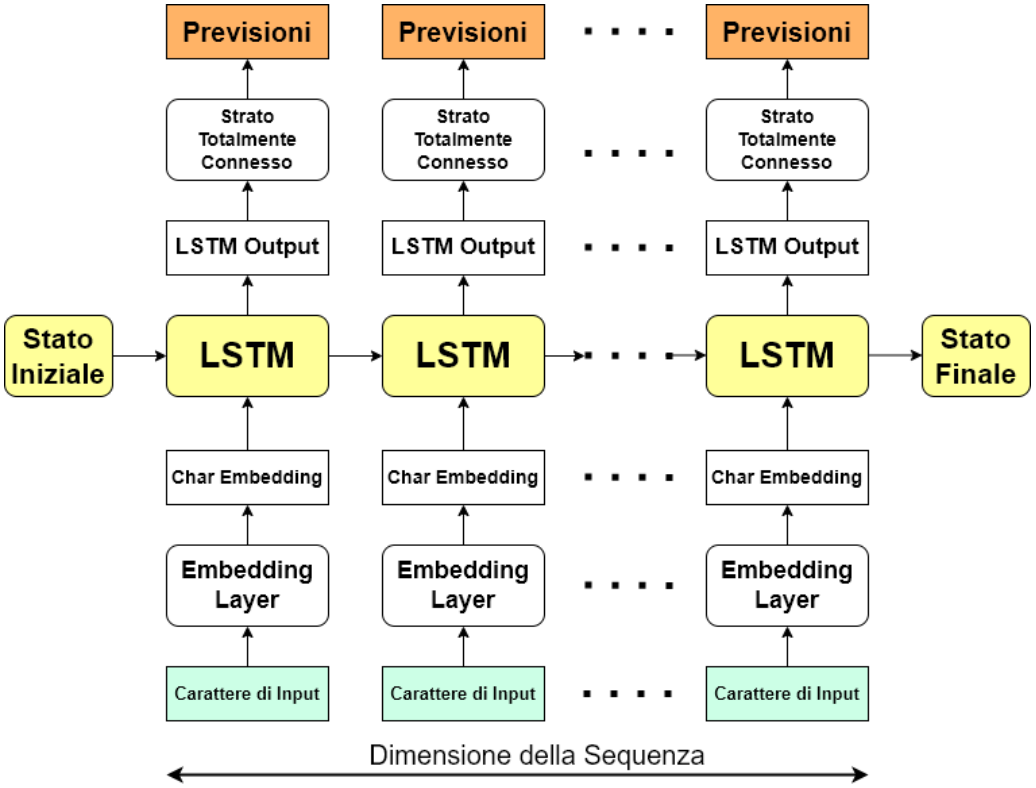

Figura 3.1: Struttura del Modello

le probabilità del carattere successivo per ogni elemento del vocabolario.

3.9 Addestrare il modello

La ricerca del carattere successivo rispetto alla sequenza di input è quindi diventato un problema di classificazione standard. Per quanto riguarda gli ultimi iperparametri del modello ho scelto *Cross entropy* come funzione di costo e Adam come *optimizer*, in entrambi i casi con il settaggio di default.

```
loss = tf.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer='adam', loss=loss)
EPOCHS = 20
history = model.fit(dataset, epochs=EPOCHS, callbacks=[checkpoint_callback])
```

3.10 Generazione del Testo

La modalità più semplice per generare testo è tramite l'iterazione del modello, in questo modo ad ogni passaggio verrà generato un carattere che sarà passato come input nell'iterazione successiva unitamente allo stato interno della rete.

```
class OneStep(tf.keras.Model):
  def __init__(self, model, chars_from_ids, ids_from_chars, temperature=1.0):
    super().__init__()
    self.temperature = temperature
    self.model = model
    self.chars_from_ids = chars_from_ids
    self.ids_from_chars = ids_from_chars
    skip_ids = self.ids_from_chars(['[UNK]'])[:, None]
    sparse_mask = tf.SparseTensor(
        values=[-float('inf')]*len(skip_ids),
        indices=skip_ids,
        dense_shape=[len(ids_from_chars.get_vocabulary())])
    self.prediction_mask = tf.sparse.to_dense(sparse_mask)

  @tf.function
  def generate_one_step(self, inputs, states=None):
    input_chars = tf.strings.unicode_split(inputs, 'UTF-8')
    input_ids = self.ids_from_chars(input_chars).to_tensor()
    predicted_logits, states = self.model(inputs=input_ids, states=states,
                                          return_state=True)
    predicted_logits = predicted_logits[:, -1, :]
    predicted_logits = predicted_logits/self.temperature
    predicted_logits = predicted_logits + self.prediction_mask
    predicted_ids = tf.random.categorical(predicted_logits, num_samples=1)
    predicted_ids = tf.squeeze(predicted_ids, axis=-1)
```

```
    predicted_chars = self.chars_from_ids(predicted_ids)
    return predicted_chars, states

one_step_model = OneStep(model, chars_from_ids, ids_from_chars)

states = None
next_char = tf.constant(['Tommy era partito per un lungo viaggio '])
result = [next_char]

for n in range(1000):
  next_char, states = one_step_model.generate_one_step(next_char, states=states)
  result.append(next_char)

result = tf.strings.join(result)
print(result[0].numpy().decode('utf-8'), '\n\n' + '_'*80)
```

3.11 Risultato e Analisi

La sequenza di input che ho scelto è "Tommy era partito per un lungo viaggio ", e ciò che segue è il testo generato automaticamente dal modello addestrato.

> Tommy era partito per un lungo viaggio metterrisci e mello come cappeta nei loro fuori dellecitome della mia mapo rinebbendie: pi per ferea, permonesse, e uomino di lega, non ho perso la miglia altornamenta ferroni e dicenar la scunferavano con sardie del bargo, il capo: In una perch, Ditt incurata carzo, si destreno appenat era lo aveva troppo dalla pi di brua, pencamente, il parte, camando, e quello parolava della mastacera una cavalcia amprendevano allgammo a questo concio dei situonifici; e Mi OBOh LAstanze Pownos. Dimentrare molto degaro? E Non per dissi, una sile stembra di spirati.

> Non avevamo perava spala di fusconto del Caniolacci. Ma andivamo.
>
> Luiv e il caperto con testa, a schi, e forzai da me i memeriggini duna forta.
>
> Nouthe. Qua, singando il cui bosco che le loro piccolo tra colpito al capitano: si stavamo sotto la macchie di mio pirato illelbingo, da tropo di Steventoniola vista del soli richementando. Non ti compagnavo saltato qui da un indiguolo la collvia dal sole che concuvare e questa fucchi dalla ca

```
Run time: 5.373439311981201
```

La Rete che ho costruito ha generato testo privo di significato, ma questo era un risultato atteso considerando che la complessità del modello è in linea con le limitate risorse che avevo a disposizione.

Le soluzioni che proporrei per rendere il testo prodotto maggiormente comprensibile, e con la supposizione di disporre di un infrastruttura hardware in linea con questa ambizione, prevede una maggiore profondità della rete che realizzerei con l'aggiunta di altri layer LSTM.

Sarebbe inoltre interessante addestrare il modello con dataset più corposi, tentativo che ho più volte cercato di realizzare durante questa sperimentazione ma a cui non ho potuto dare seguito per motivi di costi e tempi.

Analizzando il risultato della sperimentazione è comunque molto interessante rilevare come questo semplice modello abbia raggiunto risultati molto significativi. In particolar modo, l'uso corretto della punteggiatura e la forma del linguaggio sono in linea con gli obiettivi che mi ero prefissato, e che considero un ottimo punto di partenza per futuri sviluppi. E' infine piuttosto marcato il modo in cui modello ha appreso lo stile linguistico e l'ambientazione del romanzo usato per l'addestramento, caratterizzando il testo generato con frequenti richiami ad avventure marinare e ambienti pirateschi.

Conclusioni

In questo lavoro ho cercato di esplorare il campo del *Machine Learning* con un approccio ad alto livello, cercando di comprendere le nozioni fondamentali e limitando i dettagli tecnici unicamente a modelli ed architetture utilizzate durante la sperimentazione.

Questo mi ha portato a conoscere da vicino le reti neurali e i suoi numerosi algoritmi, le tecniche e i meccanismi che nei prossimi anni entreranno nella nostra quotidianità in modo perentorio e discreto.

Mi sono poi concentrato su un ambito molto importante del *Machine Learning*, l'elaborazione del linguaggio naturale, un mondo a sé stante in continua evoluzione, e in cui accademici e grandi industrie dedicano enormi risorse con risultati davvero sorprendenti.

Mi sono infine soffermato sul compito di *Machine Learning* che più mi ha stimolato e incuriosito, la generazione automatica di testo, un'attività in grandissimo risalto che con l'avvento dell' architettura *Transformer* sta ottenendo risultati davvero straordinari. I modelli basati su questa tecnologia, BERT e GTP, stanno rapidamente realizzando ciò che sembrava impossibile, costruire modelli linguistici di altissima qualità e generazione di testi indistinguibili da quelli prodotti dall'uomo.

Nei prossimi anni i contenuti informativi saranno consumati in tempi sempre più rapidi, ed è per questo facile aspettarsi dai grandi network un utilizzo crescente di modelli automatici, che nel frattempo saranno stati ulteriormente affinati.

D'altro canto, gli sviluppatori di OpenAI hanno posto l'attenzione sul

grande pericolo che un uso fraudolento di queste tecnologie potrebbe creare alla comunità, con il rischio concreto di esplosione di contenuti spam e fake-news.

La fase di sperimentazione mi ha entusiasmato e arricchito allo stesso tempo, mi ha portato a conoscere da vicino i dettagli architetturali di una rete neurale ricorrente, a comprendere la configurazione degli iperparametri, a conoscere strumenti fondamentali come Colab, Tensorflow e Keras, e a "sporcarmi le mani" in prima persona con il codice Python.

Ritengo questo lavoro una straordinaria opportunità, grazie al quale ho potuto comprendere le principali architetture che nel prossimo futuro saranno il motore degli ulteriori successi del *Machine Learning*.

Bibliografia

[1] E. Alpaydin. *Introduction to Machine Learning.* MIT Press, 2014.

[2] R. Arumugam and R. Shanmugamani. *Hands-On Natural Language Processing with Python : A Practical Guide to Applying Deep Learning Architectures to Your NLP Applications.* Packt Publishing, Limited, 2018.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[4] O. Campesato. *Artificial Intelligence, Machine Learning, and Deep Learning.* Mercury Learning Information, 2020.

[5] Sumanth Dathathri, Andrea Madotto, Janice Lan, Jane Hung, Eric Frank, Piero Molino, Jason Yosinski, and Rosanne Liu. Plug and play language models: A simple approach to controlled text generation. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020.* OpenReview.net, 2020.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[7] T. Ganegedara. *Natural Language Processing with TensorFlow : Teach Language to Machines Using Python's Deep Learning Library.* Packt Publishing, Limited, 2018.

[8] Aurelien Geron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly Media, Inc., 2nd edition, 2019.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[10] Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

[11] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020.* OpenReview.net, 2020.

[12] Touseef Iqbal and Shaima Qureshi. The survey: Text generation models in deep learning. *Journal of King Saud University - Computer and Information Sciences*, 2020.

[13] Kun Jing and Jungang Xu. A survey on neural network language models. *CoRR*, abs/1906.03591, 2019.

[14] Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016.

[15] L. Massaron and J.P. Mueller. *Machine learning for dummies.* Hoepli, 2019.

[16] Dimas Muñoz-Montesinos. Modern methods for text generation. *CoRR*, abs/2009.04968, 2020.

[17] Alan M. Turing. Computing machinery and intelligence. In Margaret A. Boden, editor, *The Philosophy of Artificial Intelligence*, Oxford readings in philosophy, pages 40–66. Oxford University Press, 1990.

[18] D. Van Boxel. *Hands-On Deep Learning with TensorFlow.* Packt Publishing, Limited, 2017.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[20] Elena Voita. NLP Course For You, Sep 2020.

[21] S. Weidman. *Deep learning. Dalle basi alle architetture avanzate con Python.* Tecniche Nuove, 2020.

[22] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[23] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.

Ringraziamenti

Innanzitutto vorrei ringraziare il Prof. Asperti per aver creduto in me e nel mio progetto, per i suoi preziosi insegnamenti e per avermi introdotto al Deep Learning, una disciplina che continuerò ad approfondire in futuro e a sperimentare con passione.

Mia moglie Silvia è stata la grande protagonista di questa avventura: a partire dal momento in cui abbiamo fatto questa scelta coraggiosa mi ha supportato senza indugi, spronandomi nelle notti insonni e nei weekend passati sui libri e motivandomi nei momenti di tensione e stanchezza, mi ha sempre trasmesso una grandissima fiducia e questo mi ha moltiplicato le forze e le motivazioni. Questo grande risultato è tuo quanto mio.

Un grazie speciale lo dedico ai nostri grandissimi tesori Tommaso e Federico: nei vostri occhi ho trovato le motivazioni per crescere e nei vostri sorrisi ho scoperto le cose davvero importanti. Mi dispiace molto per il tanto tempo che ho sottratto ai vostri giochi, per i giorni in cui tornavo a casa tardissimo dopo una giornata in ufficio e una serata in biblioteca, mi auguro davvero che questi sacrifici possano darvi nuove opportunità ma sono certo che in ogni caso sarete molto orgogliosi di me.

Un grazie speciale al mio "bro" Cristian, fratello e migliore amico, per stare sempre dalla mia parte e per esserci nei momenti del bisogno: continuerai per sempre ad essere il mio punto di riferimento.

Ai miei genitori, Valeria e Roberto, che mi hanno insegnato ad essere felice e a non smettere mai di sognare. E soprattutto ad agire, anzichè lamentarsi o piangersi addosso, se c'è qualcosa che non va.

Un grazie speciale a Francesca per il suo enorme e preziosissimo aiuto: senza di te non ce l'avrei fatta; e ai miei suoceri Mirella e Andrea, per il loro grandissimo supporto e per tutto l'aiuto che mi hanno dato inconsapevolmente in questi anni.

Grazie a Giancarlo ed Anna per il sincero apprezzamento che da subito hanno dimostrato per questa avventura.

Vorrei ringraziare inoltre tutti i miei colleghi di lavoro per aver contribuito in questi anni a rendermi un informatico migliore.

Infine un grazie speciale a tutti i Professori, il personale universitario e gli studenti che ho conosciuto in questi anni per aver condiviso con me questa meravigliosa esperienza che mi porterò dentro.