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

Tesi di Laurea

DEEP LEARNING PER LA MATEMATICA SIMBOLICA

DEEP LEARNING FOR SYMBOLIC MATHEMATICS

ELIA MERCATANTI

Relatore: *Donatella Merlini*

Anno Accademico 2020-2021

INDICE

ELENCO DELLE FIGURE

"L'uomo deve perseverare nell'idea che l'incomprensibile sia comprensibile;
altrimenti rinuncerebbe a cercare."
— J. W. Goethe

1

INTRODUZIONE

Più di 70 anni fa, i ricercatori in prima linea nella ricerca sull'intelligenza artificiale hanno introdotto le reti neurali come un modo rivoluzionario di pensare a come funziona il nostro cervello. Nel cervello umano, reti di miliardi di neuroni collegati tra loro danno un senso ai dati sensoriali inviati dal corpo permettendoci col tempo di imparare dall'esperienza. Le reti neurali artificiali e in particolare il *deep learning* cercano allo stesso modo di imitare questo comportamento filtrando enormi quantità di dati attraverso dei *layer* connessi per effettuare previsioni e riconoscere schemi, seguendo regole che imparano da sole elaborando i dati forniti.

Le persone trattano ormai le reti neurali come una sorta di panacea per l'intelligenza artificiale, in grado di risolvere sfide tecnologiche che possono essere riformulate come problemi legati al riconoscimento di schemi (*pattern recognition*). Risolvono ad esempio problemi di traduzione linguistica del linguaggio naturale, le applicazioni di fotografia le usano per riconoscere e classificare i volti ricorrenti nella nostra raccolta di foto o separarle da quelle che raffigurano animali, aiutano nella guida automatica dei veicoli per capire quando e come la macchina deve sterzare, oppure riescono a sconfiggere anche i migliori giocatori al mondo in giochi come Go e scacchi.

Tuttavia, le reti neurali sono sempre rimaste indietro in un'area cospicua: risolvere difficili problemi di matematica simbolica. Questi includono quesiti classici dei corsi di calcolo e analisi matematica, come integrali o equazioni differenziali ordinarie. Gli ostacoli nascono dalla natura stessa della matematica, che richiede soluzioni ben precise e spesso anche uniche rispetto invece ai problemi classici su cui vengono utilizzate le reti neurali. Esse infatti tendono ad eccellere più sulle probabilità o sulle approssimazioni, ovvero, in genere per ogni dato input, è ritenuto accettabile ottenere una certa gamma di output. Ad esempio per la classificazione delle immagini non importa se l'alta confidenza del modello è espressa con l'80% o con l'85% di accuratezza, oppure nel campo delle

traduzioni in genere ci possono essere più trasposizioni valide per una singola frase da poter ritenere corrette. In generale dunque le reti neurali imparano a riconoscere degli schemi ed a generarne poi di nuovi, come ad esempio quale traduzione spagnola suona meglio partendo da un testo in italiano o che aspetto ha il nostro viso.

La situazione è cambiata quando *Guillaume Lample* e *François Charton*, una coppia di scienziati informatici che lavorano nel gruppo di ricerca sull'intelligenza artificiale di *Facebook* a Parigi, hanno svelato un primo approccio di successo per risolvere problemi di matematica simbolica con le reti neurali, presentato per la prima volta in *Deep Learning for Symbolic Mathematics* (2019) [1]. Il loro metodo non prevede l'elaborazione di numeri o approssimazioni numeriche, ma gioca sui punti di forza delle reti neurali, riformulando i quesiti di matematica in termini di un problema che ormai è stato ampiamente studiato e praticamente quasi risolto: la traduzione linguistica di testi.

Il programma di *Lample* e *Charton* produce soluzioni precise ad integrali ed equazioni differenziali, inclusi alcuni che non possono essere risolti dai più popolari pacchetti software di matematica simbolica che usano algoritmi con regole esplicite per la risoluzione dei problemi.

Il nuovo programma ideato sfrutta uno dei maggiori vantaggi delle reti neurali: sviluppare e apprendere le proprie regole implicite, di conseguenza non c'è separazione tra le regole e le eccezioni. In pratica, questo significa che il programma non si blocca sugli integrali o le equazioni differenziali più difficili. In teoria dunque, questo tipo di approccio potrebbe derivare "regole" non convenzionali che potrebbero permettere dei grandi progressi su problemi che sono attualmente irrisolvibili, da una persona o da una macchina, ad esempio problemi matematici come scoprire nuove dimostrazioni matematiche o comprendere la natura delle stesse reti neurali [2].

1.1 INSEGNARE AD UN COMPUTER A PARLARE LA MATEMATICA

I computer sono sempre stati estremamente abili ad elaborare numeri. I vari sistemi di calcolo simbolico in genere combinano dozzine o centinaia di algoritmi programmati con istruzioni preimpostate. In genere seguono regole rigorose progettate per eseguire un'operazione specifica ma incapaci poi di accogliere o gestire eccezioni. Per molti problemi simbolici, questi sistemi producono soluzioni numeriche esatte per applicazioni ingegneristiche e fisiche.

Le reti neurali sono invece diverse, non hanno regole fisse. Quest'ultime

si allenano su grandi set di dati, più grandi sono e meglio è, e usano varie statistiche per ottenere approssimazioni molto buone, in questo processo imparano cosa produce i migliori risultati. I programmi di traduzione linguistica si distinguono particolarmente: invece di tradurre parola per parola, traducono le frasi nel contesto dell'intero testo. I ricercatori di *Facebook* hanno dunque visto questo aspetto come un vantaggio per risolvere problemi di matematica simbolica, non come ostacolo, concedendo al programma una sorta di libertà per la risoluzione di tali problemi.

Questa libertà è particolarmente utile per alcuni problemi come l'integrazione. In genere infatti per trovare la derivata di una funzione è necessario solo seguire alcuni passaggi ben definiti, ma calcolare un integrale spesso richiede molto più impegno, serve in genere qualcosa che sia più vicino ad un'intuizione che al mero calcolo.

Il gruppo di ricercatori di *Facebook* ha pensato che questa intuizione potesse essere approssimata usando il riconoscimento di *pattern*, dato che l'integrazione ad esempio è uno dei problemi matematici più simili al riconoscimento di schemi. Quindi, anche se la rete neurale potrebbe non capire cosa facciano le funzioni o cosa significano le variabili, sviluppa una sorta di istinto che le permette di iniziare a percepire cosa funziona anche senza sapere perché. Ad esempio, un matematico a cui viene chiesto di integrare un'espressione come $yy'(y^2+1)^{-\frac{1}{2}}$ penserà intuitivamente che la primitiva, cioè l'espressione che è stata differenziata per dare origine all'integrale, contenga qualcosa che assomigli a $\sqrt{y^2+1}$.

Per consentire ad una rete neurale di elaborare espressioni come un matematico, *Charton* e *Lample* hanno utilizzato un'architettura di *deep learning* basata sui modelli *Sequence to Sequence* (*Seq2Seq*) che hanno ottenuto molto successo in attività come la traduzione automatica dei linguaggi naturali, il riepilogo del testo, il riconoscimento vocale e in generale tutti quei problemi che rientrano nella branca del *Natural Language Processing* (NLP) ma anche nella didascalia di immagini e video. Ad esempio, sono stati utilizzati per sviluppare applicazioni come *Google Translate*. Questi tipi di modelli, osservandoli nel loro insieme, non fanno altro che prendere in input una sequenza di qualunque lunghezza, che in generale può contenere qualsiasi cosa come lettere, parole o serie temporali, e restituiscono in output un'altra sequenza di lunghezza arbitraria. Già da questa descrizione si può intuire come essi siano particolarmente adatti a problemi di traduzione.

Un esempio viene mostrato in Figura 1 dove possiamo notare come può essere eseguita una traduzione dal tedesco all'inglese della frase "come stai" utilizzando un modello *Seq2Seq*. Come si può osservare la

frase viene tradotta cercando di mantenere il senso originale, senza per forza eseguire una traduzione parola per parola e andando contro alle regole della lingua di traduzione.

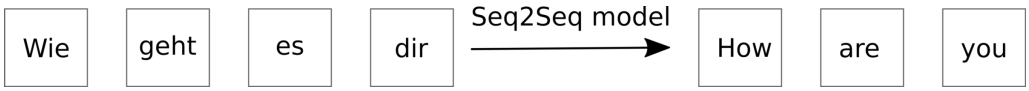

Figura 1: Sequenza di input in tedesco trasformata in una sequenza di output tradotta in inglese.

In particolare *Lample* e *Charton* hanno utilizzato il modello *Transformer*, presentato per la prima volta da alcuni ricercatori di *Google* in *Attention Is All You Need* (2017) [3], uno speciale tipo di modello *Seq2Seq* che sfrutta il meccanismo dell'*Attention* (letteralmente dell'attenzione), che permette al modello di concentrarsi maggiormente e solo sulle parti della sequenza più importanti al fine di ricavare i vari elementi della giusta sequenza di output.

L'idea alla base del lavoro dei due ricercatori è stata dunque quella di cercare di generare e poi successivamente trasformare le espressioni matematiche di integrali ed equazioni differenziali in sequenze adatte ad un modello *Seq2Seq*, per poi lasciare a quest'ultimo il compito di estrapolare dagli enormi dataset forniti le informazioni per ricavare le regole implicite per generare delle soluzioni corrette, rappresentate anch'esse tramite sequenze. La sequenza di input conterrà i simboli che definiscono l'espressione di input e la sequenza di output sarà rappresentata dai simboli che definiscono l'espressione della soluzione.

Il loro lavoro è iniziato traducendo espressioni matematiche in forme più utili e comprensibili. Hanno dunque deciso di reinterpretarle come alberi, un formato simile nello spirito a una frase schematizzata. Operatori matematici come l'addizione, la sottrazione, la moltiplicazione e la divisione diventano i nodi dell'albero e allo stesso modo anche operazioni come l'elevazione a potenza o le funzioni trigonometriche. Gli argomenti invece, come variabili e numeri, diventano le foglie dell'albero. Questo tipo di struttura cattura infatti il modo in cui le operazioni possono essere nidificate all'interno delle varie espressioni matematiche. In Figura 2 vengono mostrati alcuni esempi.

Quando osserviamo una grande funzione, possiamo notare come spesso è composta da funzioni più piccole e di conseguenza abbiamo una certa intuizione su quale possa essere la soluzione. Il modello basato su reti neurali proposto dai due ricercatori cerca in modo simile di trovare indizi per la soluzione nei simboli contenuti nell'espressione matematica

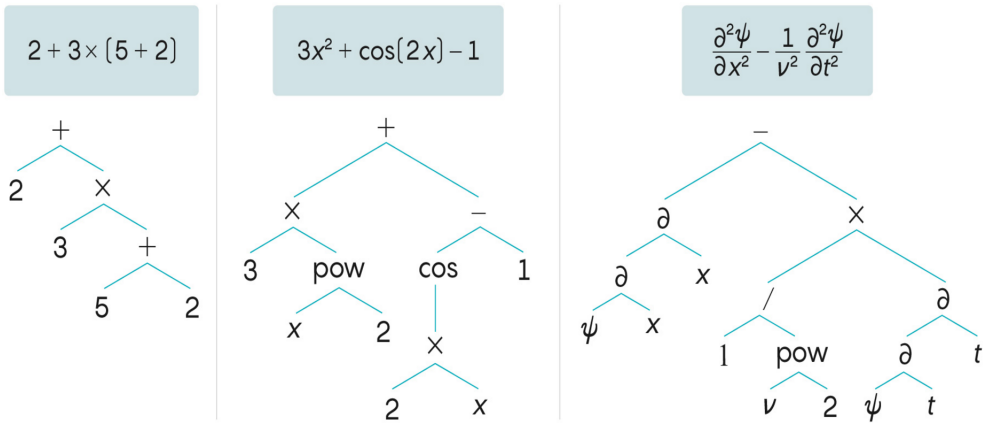

Figura 2: Espressioni matematiche in forma di albero

passata. Questo processo può essere rapportato al modo in cui le persone risolvono gli integrali, e in realtà a tutti i problemi di matematica, ovvero riducendoli a sottoproblemi riconoscibili che hanno già visto e risolto in precedenza. Infine, una volta ottenuta un'espressione matematica in forma di albero non resta che ricavarne una sequenza, ad esempio utilizzando la notazione prefissa, per darla in pasto al modello *Seq2Seq*.

Dopo aver ideato questa architettura, i ricercatori hanno utilizzato un gruppo di funzioni elementari per generare diversi dataset di addestramento per un totale di circa 250 milioni di equazioni e soluzioni, sfruttando la rappresentazione ad albero. Hanno successivamente "alimentato" la rete neurale con tali dati, in modo che potesse apprendere la forma delle soluzioni di questi problemi e come ricavarle, ed infine hanno testato come la rete si comporta su dati mai visti prima. I due ricercatori hanno fornito alla rete una serie di *test set* da 5.000 equazioni, questa volta senza soluzioni, e la rete neurale è passata a pieni voti, riuscendo ad ottenere le giuste soluzioni nella stragrande maggioranza dei problemi. La rete eccelle particolarmente nell'integrazione, risolvendo quasi il 100% dei problemi di test, ma ha un po' meno successo nelle equazioni differenziali ordinarie, mantenendo però anche in questo caso accuratezze abbastanza alte.

Per tutti i problemi, il programma ha impiegato meno di 1 secondo per generare soluzioni corrette, e sui problemi di integrazione ha superato anche alcuni risolutori dei più popolari pacchetti software di matematica simbolica come *Mathematica* e *Matlab* in termini di velocità e precisione. Il team di *Facebook* ha inoltre scoperto che la rete neurale produce soluzioni a problemi che nessuno di questi risolutori commerciali potrebbe affrontare [2].

L'obbiettivo principale di questa tesi è stato dunque quello di presentare ed approfondire il lavoro svolto dai due ricercatori *Lample* e *Charton*, spiegando l'idea alla base del loro studio, come sono stati generati i dataset richiesti per l'addestramento dei modelli, che tipo di architetture sono state utilizzate e come sono riusciti a rendere accurate ed estremamente efficienti delle reti neurali anche su problemi di matematica simbolica, dimostrando che il *deep learning* può essere sfruttato anche su questo tipo di compiti. Sono stati approfonditi i modelli *Sequence to Sequence*, descrivendone il loro comportamento, i loro pregi e difetti, ed in generale le tecniche utilizzate nel campo del *Natural Language Processing* (NLP) che sfruttano queste architetture per risolvere problemi legati all'elaborazione dei linguaggi. E' stato studiato ed approfondito come funziona il modello *Transformer*, la vera architettura utilizzata dai ricercatori per creare i vari modelli, e come può essere sfruttato per risolvere integrali ed equazioni differenziali. Inoltre, i modelli proposti dal team di ricercatori sono stati testati per verificarne la loro accuratezza e il loro comportamento su diversi dataset e anche su alcuni problemi pratici, cercando di analizzare i loro maggiori punti di forza.

In particolare, nel Capitolo 2 ci concentreremo sul descrivere i concetti e le idee di base del lavoro di ricerca di *Lample* e *Charton*, ovvero, come i quesiti di matematica sono stati associati ai problemi sui linguaggi naturali, come le espressioni matematiche sono state trasformate nelle sequenze richieste dai modelli Seq2Seq e come sono stati generati i dataset per le integrazioni e per le equazioni differenziali basandosi su generatori di espressioni matematiche casuali. Nel Capitolo 3 descriveremo in generale di cosa si occupa il campo della *Natural Language Processing* (NLP), parleremo di alcuni concetti di base sulle reti neurali e sul *deep learning*, concentrandoci su come possono essere applicati a problemi dell'NLP, ed inoltre approfondiremo come funzionano i modelli Seq2Seq, quali sono le loro caratteristiche e i loro punti di forza. Nel Capitolo 4 invece descriveremo e approfondiremo come funziona il modello *Transformer*, che tipo di architettura usa e come utilizza ogni suo componente per produrre delle sequenze di output. Infine, nel Capitolo 5 andremo a testare vari modelli *Transformer* addestrati su dataset di integrali ed equazioni differenziali per valutare le loro accuratezze su questi problemi e che tipo di soluzioni restituiscono.

2

DEEP LEARNING PER LA MATEMATICA SIMBOLICA

Le reti neurali hanno dimostrato di essere estremamente efficaci nel *pattern recognition* statistico e sono ormai in grado di ottenere prestazioni all'avanguardia su una vasta gamma di problemi che spaziano dalla visione artificiale, dal riconoscimento vocale, l'elaborazione del linguaggio naturale (NLP, *natural language processing*), ecc. Tuttavia, il successo delle reti neurali nel calcolo simbolico è ancora estremamente limitato: riuscire a combinare il ragionamento simbolico con le rappresentazioni continue è ancora una delle grosse sfide da affrontare nel campo del *machine learning*. Le reti neurali hanno la reputazione di lavorare al meglio nella risoluzione di problemi statistici o in generale su quei problemi dove è concesso un certo grado di approssimazione, questa loro natura le mette dunque in difficoltà nel momento in cui devono risolvere problemi di puro calcolo o lavorano su dati simbolici matematici, dove sono richieste soluzioni ben precise.

2.1 STUDI PRECEDENTI

Solo pochi studi hanno investigato sulla capacità delle reti neurali di trattare oggetti matematici e salvo un piccolo numero di eccezioni la maggior parte di questi lavori si concentra su compiti aritmetici come l'addizione e la moltiplicazione di numeri interi. In questi compiti, gli approcci neurali tendono a funzionare male e richiedono l'introduzione nel modello di componenti che tendono fortemente al compito da svolgere.

I computer sono stati utilizzati per la matematica simbolica a partire dalla fine degli anni '60 ma negli ultimi decenni per risolvere una grande varietà di compiti matematici vengono in genere utilizzati i cosiddetti *Computer Algebra Systems* (CAS), ad esempio software come *Matlab*, *Mathematica*, *Maple*, *PARI* o *SAGE*. Questi software in genere utilizzano metodi moderni per l'integrazione simbolica che sono basati sull'algoritmo di *Risch* (Risch, 1970) [4], un algoritmo che trasforma il problema

dell'integrazione in un quesito algebrico. È basato sulla forma della funzione integrata e sui metodi per integrare funzioni razionali, irrazionali, logaritmiche ed esponenziali, ed utilizza un metodo per decidere se una funzione ha una corrispettiva funzione elementare come integrale indefinito. Tuttavia, la descrizione completa dell'algoritmo di *Risch* richiede più di 100 pagine e non è completamente implementato negli attuali frameworks matematici.

In letteratura possiamo trovare che le reti di *deep learning* sono state ad esempio utilizzate per semplificare espressioni matematiche con rappresentazione ad albero in *Zaremba et al.* (2014) [5], utilizzando reti neurali ricorsive, ma fornendo al modello informazioni relative al problema, ovvero possibili regole per la semplificazione dove la rete neurale è addestrata per selezionare la regola migliore. *Allamanis et al.* (2017) [6] hanno proposto un framework chiamato "reti di equivalenza neurale" per apprendere rappresentazioni semantiche di espressioni algebriche. In genere in questo caso, viene addestrato un modello per mappare espressioni diverse ma equivalenti alla stessa rappresentazione. Tuttavia, sono state considerate solo espressioni booleane e polinomiali. Più recentemente, *Arabshahi et al.* (2018) [7] hanno utilizzato reti neurali strutturate ad albero per verificare la correttezza di determinate entità simboliche e per prevedere gli elementi mancanti in equazioni matematiche incomplete, dimostrando anche che queste reti potrebbero essere utilizzate per prevedere se un'espressione è una soluzione valida di una data equazione differenziale [8].

La maggior parte dei tentativi di utilizzare reti profonde per la matematica si è concentrata sull'aritmetica applicata ad interi, a volte su polinomi a coefficienti interi. Ad esempio, *Kaiser & Sutskever* (2015) [9] hanno proposto l'architettura *Neural-GPU* e hanno addestrato delle reti neurali per eseguire addizioni e moltiplicazioni di numeri utilizzando le loro rappresentazioni binarie, mostrando che un modello addestrato su numeri fino a 20 bit può essere applicato a numeri molto più grandi in fase di test, preservando una precisione perfetta.

Saxton et al. (2019) [10] hanno utilizzato invece delle reti neurali LSTM, *Long Short-Term Memory*, (*Hochreiter & Schmidhuber*, 1997) [11], e reti *Transformers* su un'ampia gamma di problemi, dall'aritmetica alla semplificazione delle espressioni formali. Tuttavia, hanno considerato solo funzioni polinomiali e il problema della differenziazione, che risulta essere un compito significativamente più semplice dell'integrazione. *Trak et al.* (2018) [12] hanno invece proposto delle unità logiche aritmetiche neurali, un nuovo modulo progettato per apprendere il calcolo numerico

sistematico e che può essere utilizzato all'interno di qualsiasi rete neurale e come *Kaiser & Sutskever* (2015) [9], mostrando durante la fase di inferenza che il loro modello può generalizzare su numeri di ordini grandezza maggiori di quelli osservati durante la fase di addestramento.

2.2 LA MATEMATICA COME IL LINGUAGGIO NATURALE

Due ricercatori di *Facebook*, *Guillaume Lample* e *François Charton*, nel loro lavoro *Deep Learning for Symbolic Mathematics* (2019) [1] sono però riusciti a mostrare come queste difficoltà sui problemi di matematica simbolica possono in alcuni casi essere superate, sopratutto quando si richiede loro di risolvere compiti matematici molto più elaborati come ad esempio l'integrazione simbolica e la risoluzione di equazioni differenziali, dimostrando come possano essere sorprendentemente performanti.

Il team nel loro articolo di ricerca propone inizialmente una sintassi per rappresentare problemi matematici e alcuni metodi per generare grandi dataset che possono essere utilizzati per addestrare modelli *Sequence to Sequence* (*Seq2Seq*), dopodiché mostrano gli ottimi risultati ottenuti evidenziando come siano riusciti a superare anche i sistemi commerciali di calcolo simbolico come *Mathematica* o *Matlab*.

L'idea principale da cui i due ricercatori sono partiti nel loro articolo originale, è stata quella di considerare la matematica, e in particolare i calcoli simbolici, come problemi associabili alla traduzione dei linguaggi naturali, e dunque risolvibili tramite modelli pensati per l'NLP. Più precisamente, l'idea di base è stata quella di usare modelli *Sequence to Sequence* su due problemi di matematica simbolica: l'integrazione di funzioni e la risoluzione di equazioni differenziali ordinarie (ODE) partendo dal fatto che entrambi risultano essere problemi difficili da risolvere in molti casi sia da matematici esperti sia da software per computer. Nel caso dell'integrazione, agli studenti in genere viene insegnata una serie di regole (integrazione per parti, cambio di variabile, ecc.), che non garantiscono il successo in ogni caso possibile, e anche i *Computer Algebra Systems* utilizzano algoritmi molto complessi che esplorano un gran numero di casi specifici con regole ben precise, come ad esempio il già citato algoritmo di *Risch* (Risch, 1970) [4].

Tuttavia, se analizziamo la struttura che hanno le funzioni da integrare e le loro soluzioni possiamo notare come in realtà strategie basate sul riconoscimento di *pattern* potrebbero essere molto utili su questo tipo di problemi. Ad esempio, quando noi osserviamo un'espressione del tipo $yy'(y^2+1)^{-\frac{1}{2}}$ possiamo subito intuire che la sua primitiva conterrà

sicuramente un termine simile a $\sqrt{y^2+1}$ per le regole di integrazione che abbiamo studiato e per l'esperienza che guadagniamo e assimiliamo nella risoluzione di integrali simili. Rilevare questo tipo di *pattern* può essere facile per piccole espressioni di y, ma diventa sempre più difficile quando il numero di operatori in y aumenta. Un discorso simile può essere fatto anche per la risoluzione di equazioni differenziali. Tuttavia ad oggi esistono solo pochissimi studi che hanno indagato le capacità delle reti neurali di rilevare *pattern* nelle espressioni matematiche.

In questo capitolo ci concentreremo dunque su come siano stati rappresentati i problemi matematici proposti dai due ricercatori e come poi siano stati generati gli enormi dataset necessari per l'addestramento dei modelli, rimandando ai capitoli successivi l'analisi del funzionamento dei modelli *Seq2Seq* e in particolare del modello *Transformer*, il vero specifico modello utilizzato dal team per la risoluzione di integrali ed equazioni differenziali.

2.3 ESPRESSIONI MATEMATICHE IN FORMA DI ALBERI

Le espressioni matematiche possono essere rappresentate in forma di alberi, usando operatori e funzioni come nodi interni, gli operandi come nodi figli, e numeri, costanti e variabili come nodi foglia. I seguenti alberi mostrati in Figura 3 rappresentano alcuni esempi di espressioni come $2+3\times(5+2)$, $3x^2+\cos(2x)-1$ e $\frac{\partial^2\psi}{\partial^2 x^2}-\frac{1}{v^2}\frac{\partial^2\psi}{\partial^2 t^2}$:

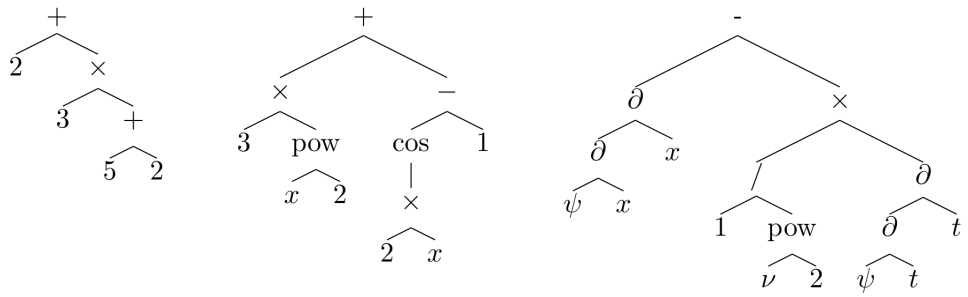

Figura 3: Espressioni matematiche in forma di albero.

Uno dei vantaggi nell'utilizzo di strutture ad albero è sicuramente rendere non ambiguo l'ordine delle operazioni, essi infatti possono gestire in modo semplice proprietà o regole come la precedenza e l'associatività delle operazioni ed eliminano il bisogno e l'uso delle parentesi. Un altro vantaggio chiave è la rimozione del bisogno dell'aggiunta di simboli privi

di significato come spazi, punteggiatura o parentesi ridondanti, che non aggiungono nessun tipo di informazione utile alle espressioni.

In generale inoltre è facile intuire come ad ogni espressione diversa corrisponda un albero diverso come risultato, ma il team di ricercatori ha cercato il più possibile di creare una mappatura uno ad uno tra espressioni e alberi per ottenere una rappresentazione il più efficiente possibile, con alcune assunzioni che discutiamo qui di seguito. Ad espressioni diverse risulteranno sempre alberi differenti, ma per permettere che valga anche il contrario, dobbiamo occuparci di alcuni casi speciali.

Innanzitutto, espressioni come somme e prodotti possono corrispondere a diversi alberi. Ad esempio, l'espressione $2 + 3 + 5$ può essere rappresentata come uno qualsiasi degli alberi mostrati in Figura 4:

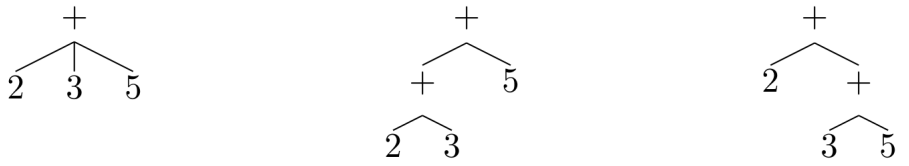

Figura 4: Possibili varianti di alberi ottenuti dall'espressone $2 + 3 + 5$.

Per far valere la mappatura uno ad uno viene dunque assunto che tutti gli operatori abbiano al massimo due operandi e che, in caso di dubbio, siano associativi verso destra. L'espressione $2 + 3 + 5$ corrisponderà quindi sempre all'albero più a destra della Figura 4.

In secondo luogo, la distinzione tra nodi interni (operatori) e foglie (oggetti matematici primitivi) è alquanto arbitraria. Ad esempio, il numero -2 potrebbe essere rappresentato come un oggetto di base singolo, o attraverso l'operatore unario meno $(-)$ applicato al numero 2. Allo stesso modo, ci sono diversi modi per rappresentare $\sqrt{5}$, $42x^5$, o la funzione $\log_{10}$. Per semplicità è stato deciso di considerare come possibili foglie di un albero solo numeri, costanti e variabili, evitando dunque l'uso del meno unario. In particolare, espressioni come $-x$ sono rappresentate come $-1 \times x$. In Figura 5 vengono mostrati gli alberi per -2, $\sqrt{5}$, $42x^5$, e $-x$:

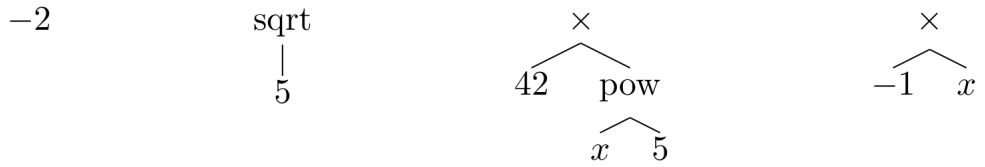

Figura 5: Alberi per le espressioni -2, $\sqrt{5}$, $42x^5$, e $-x$.

Infine gli interi sono rappresentati tramite notazione posizionale, ovvero, con un segno seguito da una sequenza di cifre (da 0 a 9 in base 10). Ad esempio, 1234 e -78 sono rappresentati come $+1\ 2\ 3\ 4$ e $-\ 7\ 8$. Per lo zero invece, viene scelta un'unica rappresentazione tra $+0$ o -0.

Dato che esiste una corrispondenza biunivoca tra alberi ed espressioni, l'uguaglianza tra espressioni si rispecchierà sull'equivalenza dei loro alberi associati: poiché $3 + 5 = 8 = 10 - 2 = 1 \times 8$, i quattro alberi corrispondenti a queste espressioni sono equivalenti.

In sostanza dunque le espressioni matematiche vengono considerate come sequenze di simboli matematici. Espressioni come $3+5$ e $5+3$ sono ritenute diverse, come lo sono $\sqrt{9}x$ e $3x$, e saranno quindi rappresentate da alberi diversi. Inoltre bisogna considerare che seppur la maggior parte delle espressioni rappresentano oggetti matematici significativi, alcune di loro come ad esempio $x/0$, $\sqrt{-2}$ o $\log(0)$ sono espressioni che sono sicuramente legittime anche solo dal punto di vista sintattico ma che non hanno necessariamente un senso matematico.

Molti problemi della matematica formale possono essere riformulati come operazioni su espressioni o alberi. Ad esempio, la semplificazione di un'espressione matematica può essere rapportata a trovare una rappresentazione più corta di un albero ma equivalente. Il team di ricercatori come già accennato si è però concentrato su due problemi: l'integrazione simbolica e le equazioni differenziali, ed entrambi se ci pensiamo bene in fin dei conti possono essere ridotti alla trasformazione di un'espressione in un'altra, ad esempio a mappare l'albero di un'equazione all'albero della sua soluzione, dunque tutti e due possono essere considerati come un caso particolare di traduzione, e nello specifico possiamo trattarli come compiti da far risolvere ad un modello per la *machine translation* (traduzione automatica).

2.4 DAGLI ALBERI ALLE SEQUENZE

I sistemi di *machine translation*, che sono un sottocampo della linguistica computazionale che indaga l'uso del software per tradurre testo o discorsi da una lingua all'altra, in genere operano su sequenze (*Sutskever et al.*, 2014 [13]; *Bahdanau et al.*, 2014 [14]), come approfondiremo poi anche nei capitoli successivi. In letteratura sono stati proposti anche approcci alternativi che utilizzano direttamente gli alberi, come i *Tree-LSTM* (*Tai et al.*, 2015 [15]) o *le Recurrent Neural Network Grammars* (RNNG) (*Dyer et al.*, 2016[16]). Tuttavia, i modelli *Tree to Tree* sono molto più lenti delle loro controparti *Sequence to Sequence*, sia in fase di addestramento sia in fase

di inferenza. Per semplicità ed efficienza dunque i due ricercatori hanno puntato ai modelli *Seq2Seq*, che hanno già dimostrato in passato di essere efficaci nella generazione di alberi, come ad esempio nel contesto del *constituency parsing* (*Vinyals et al.*, 2015 [17]), dove il compito è prevedere un albero di analisi sintattica delle frasi di input.

Se vogliamo dunque utilizzare modelli *Seq2Seq* sfruttando le strutture ad albero dobbiamo trovare un modo per mappare gli alberi a delle sequenze. Per raggiungere questo scopo i due ricercatori hanno scelto di usare la notazione prefissa, nota anche come notazione polacca, un tipo di notazione matematica in cui gli operatori precedono i loro operandi, in contrasto con la più comune notazione infissa, in cui gli operatori sono posizionati tra gli operandi.

La notazione prefissa può essere infatti facilmente utilizzata come sintassi per le espressioni matematiche garantendo una rappresentazione uno ad uno tra sequenze prefisse e gli alberi. Se vogliamo trasformare un albero rappresentante un'espressione matematica in una sequenza prefissa che la descriva, basterà scrivere ogni nodo prima dei suoi figli, elencandoli da sinistra a destra, eseguendo dunque una visita in ordine anticipato dell'albero. Ad esempio, l'espressione aritmetica $2 * (3 + 4) + 5$ è rappresentata dalla sequenza prefissa $[+ \ * \ 2 \ + \ 3 \ 4 \ 5]$, come mostrato in Figura 6. In contrasto con la più comune notazione infissa $2 * (3 + 4) + 5$, le sequenze prefisse non hanno bisogno di parentesi finché ogni operatore ha un numero fisso di operandi e sono quindi di conseguenza più brevi. Infine all'interno di ogni sequenza, gli operatori, le funzioni o le variabili sono rappresentati da specifici *token* (simboli), e i numeri interi invece da sequenze di cifre precedute da un segno.

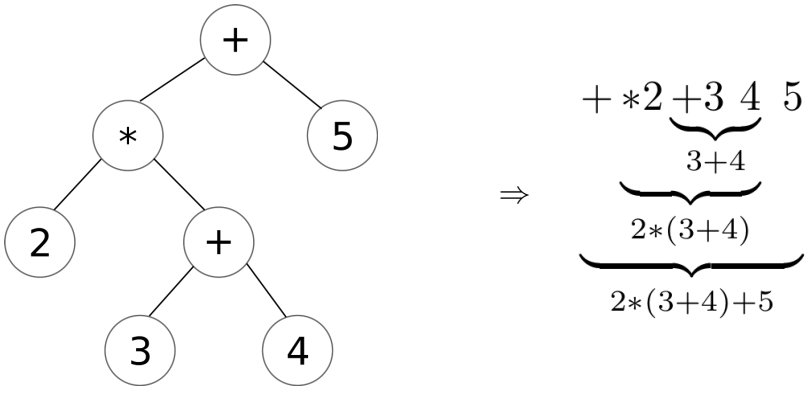

Figura 6: Da albero a sequenza in notazione prefissa per $2 * (3 + 4) + 5$.

2.5 GENERARE ESPRESSIONI MATEMATICHE CASUALI

Per addestrare correttamente delle reti di *deep learning*, ma in particolare anche per i modelli *Seq2Seq*, è necessario avere a disposizione enormi quantità di dati, in genere a seconda del problema anche nell'ordine di milioni, come vedremo anche nei nostri esperimenti finali. Non avendo a disposizione dei set di dati contenenti espressioni matematiche adatte al problema dell'integrazione o delle equazioni differenziali, e sopratutto di dimensioni adeguate, il team di ricercatori ha scelto di generare tramite algoritmi dei grandi insiemi di espressioni matematiche casuali per i due problemi scelti, sfruttando le rappresentazioni ad albero. Generare uniformemente espressioni partendo da alberi con n nodi interni non è però un compito semplice. Esistono ad esempio molti algoritmi *naive* che sfruttano strategie come i metodi ricorsivi o tecniche che utilizzano probabilità fisse per scegliere come generare i vari nodi di un albero, ovvero, decidere se dovranno essere foglie, oppure unari o binari nei loro figli. Queste strategie però tendono a favorire alberi profondi rispetto ad alberi larghi, o inclinati a sinistra rispetto ad alberi inclinati a destra. Inoltre i due ricercatori, come vedremo anche in seguito nel capitolo di sperimentazione, si sono limitati a generare espressioni matematiche con operatori binari e funzioni unarie, questo dunque richiederà di generare solo alberi di tipo binario. In generale però quello che vogliamo è far si che tutte le possibili forme di albero vengano generate con la stessa probabilità, in modo tale da ottenere poi delle espressioni matematiche il più possibile eterogenee, garantendo dunque la creazione di un dataset che riesca a generalizzare bene sul problema a cui punta. In Figura 7 vengono mostrati degli esempi di alberi strutturalmente diversi ma con lo stesso numero di nodi che vogliamo generare con la stessa probabilità.

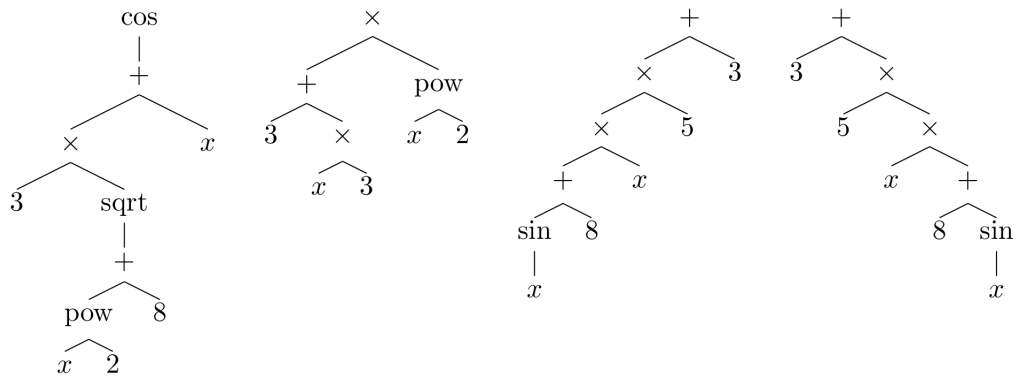

Figura 7: Tipi di alberi differenti che vogliamo generare.

Di seguito presenteremo due algoritmi per generare alberi con n nodi interni ed espressioni matematiche casuali, dove i quattro esempi di alberi in Figura 7 possono essere tutti generati con la stessa probabilità.

Per raggiungere questo obiettivo vengono generati alberi casuali che sono poi "addobbati" con simboli o funzioni matematiche selezionando casualmente i loro nodi e foglie. Iniziamo però con il caso più semplice, ovvero la generazione di alberi binari.

2.5.1 *Generazione di Alberi Binari*

Per generare un albero binario casuale con n nodi interni, viene utilizzata la seguente procedura a singolo passaggio. Partendo da un nodo radice vuoto, viene determinato ad ogni passo la posizione dei successivi nodi interni da creare tra i nodi vuoti e questo procedimento viene ripetuto fino a quando tutti gli n nodi interni sono stati allocati, come viene mostrato nell'Algoritmo 1.

Algorithm 1: Generare un albero binario casuale

```
Inizia con un nodo vuoto, imposta e = 1
while n > 0 do
    Campiona una posizione k da K(e, n)
    Campiona i k nodi vuoti successivi come foglie
    Viene scelto un operatore, crea due nodi figli vuoti
    e = e − k + 1
    n = n − 1
```

Indichiamo con e il numero di nodi vuoti, con $n > 0$ il numero di operatori ancora da generare, e con $K(e, n)$ la distribuzione di probabilità della posizione (partendo da 0) del prossimo nodo interno da allocare.

Per calcolare $K(e, n)$, definiamo $D(e, n)$, come il numero di diversi sottoalberi binari che possono essere generati da e elementi vuoti, con n nodi interni da generare. Avremo dunque che:

$$D(0, n) = 0 \tag{2.1}$$

$$D(e, 0) = 1 \tag{2.2}$$

$$D(e, n) = D(e - 1, n) + D(e + 1, n - 1) \tag{2.3}$$

La prima equazione 2.1 dichiara che nessun albero può essere generato con zero nodi vuoti e $n > 0$ operatori. La seconda equazione 2.2 afferma

che se nessun operatore deve essere assegnato, i nodi vuoti devono consistere tutti in nodi foglia e in questo caso può esistere un solo albero. L'ultima equazione 2.3 dichiara invece che se abbiamo $e > 0$ nodi vuoti, il primo deve essere o un nodo foglia, e ci sono $D(e-1, n)$ alberi di questo tipo, oppure un nodo interno, dove esisteranno $D(e+1, n-1)$ alberi. Questo ci permette di calcolare $D(e, n)$ per ogni e ed n.

Per calcolare la distribuzione $K(e, n)$, si può osservare che tra i $D(e, n)$ alberi con e nodi vuoti ed n operatori, $D(e+1, n-1)$ alberi hanno un nodo binario nella loro prima posizione. Perciò:

$$P(K(e, n) = 0) \;=\; \frac{D(e+1, n-1)}{D(e, n)}.$$

Dei restanti $D(e-1, n)$ alberi, $D(e, n-1)$ hanno un nodo binario nella loro prima posizione, cioè:

$$P(K(e, n) = 1) \;=\; \frac{D(e, n-1)}{D(e, n)}.$$

Per induzione su k possiamo infine ottenere la formula generale:

$$P(K(e, n) = k) \;=\; \frac{D(e-k+1, n-1)}{D(e, n)}.$$

2.5.2 *Generazione di Alberi Unari-Binari*

Nel caso generale, i nodi interni degli alberi che vengono generati possono essere di due tipi: unari o binari. Possiamo adattare il precedente algoritmo considerando la distribuzione di probabilità bidimensionale $L(e, n)$ della posizione (partendo da 0) e dell'arietà del prossimo nodo interno da allocare, ovvero $P(L(e, n)) = (k, a)$ rappresenterà la probabilità che il prossimo nodo interno scelto sia in posizione k e abbia arietà pari ad a. L'Algoritmo 2 mostra la strategia che viene utilizzata.

Per calcolare $L(e, n)$, anche in questo caso ci ricaviamo $D(e, n)$, il numero di sottoalberi con n nodi interni che possono essere generato da e nodi vuoti. Avremo dunque che, per tutti gli $n > 0$ ed e:

$$D(0, n) = 0 \tag{2.4}$$

$$D(e, 0) = 1 \tag{2.5}$$

$$D(e, n) = D(e-1, n) + D(e, n-1) + D(e+1, n-1) \tag{2.6}$$

La prima equazione 2.4 afferma che nessun albero può essere generato con zero nodi vuoti e $n > 0$ operatori.

Algorithm 2: Generare un albero unario-binario casuale

```
Inizia con un nodo vuoto, imposta e = 1
while n > 0 do
    Campiona una posizione k e un'arità a da L(e, n)
    Campiona i k nodi vuoti successivi come foglie
    if a = 1 then
        Campiona un operatore unario
        Crea un nodo figlio vuoto
        e = e − k
    else
        Campiona un operatore binario
        Crea due nodi figli vuoti
        e = e − k + 1
    n = n − 1
```

La secondo equazione 2.5 dichiara che se nessun operatore deve essere allocato, i nodi vuoti devono consistere tutti in nodi foglia e in questo caso può esistere un solo albero. La terza equazione 2.6 afferma che con $e > 0$ nodi vuoti, il primo deve essere o un nodo foglia, e ci sono $D(e-1, n)$ alberi di questo tipo, un operatore unario, e dunque avremo $D(e, n-1)$ alberi possibili, o un operatore binario, dove esisteranno $D(e+1, n-1)$ alberi.

Per ricavare $L(e, n)$, possiamo osservare che tra i $D(e, n)$ sottoalberi con e nodi vuoti ed n nodi interni da generare, $D(e, n-1)$ alberi hanno un operatore unario in posizione zero, e $D(e+1, n-1)$ hanno un operatore binario in posizione zero. Di conseguenza, abbiamo che:

$$P(L(e, n) = (0, 1)) = \frac{D(e, n-1)}{D(e, n)}$$

$$P(L(e, n) = (0, 2)) = \frac{D(e+1, n-1)}{D(e, n)}.$$

Come nel caso binario, possiamo generalizzare per induzione queste probabilità a tutte le posizioni k in $\{0, \ldots, e-1\}$, ottenendo le formule:

$$P(L(e, n) = (k, 1)) = \frac{D(e-k, n-1)}{D(e, n)}$$

$$P(L(e, n) = (k, 2)) = \frac{D(e-k+1, n-1)}{D(e, n)}.$$

2.5.3 *Campionare Espressioni Matematiche*

Per ottenere espressioni matematiche adatte ai nostri scopi, possiamo dunque generare degli alberi binari casuali o unari-binari utilizzando gli algoritmi discussi precedentemente, per poi selezionare casualmente i loro nodi interni e nodi foglia "decorandoli" con elementi presi da una lista di possibili operatori o entità matematiche (interi, variabili o costanti) selezionati casualmente.

I nodi e le foglie dei vari alberi possono essere selezionati in modo uniforme o secondo una probabilità a priori scelta dall'utente. Ad esempio, gli interi tra -10 e 10 potrebbero essere campionati in modo tale che i valori assoluti piccoli siano più frequenti di quelli grandi. Per gli operatori invece, l'addizione e la moltiplicazione potrebbero ad esempio essere scelte con più frequenza rispetto alla sottrazione e alla divisione.

2.6 GENERARE DEI DATASET

Dopo aver definito una sintassi per problemi matematici e delle tecniche per generare espressioni casuali, siamo ora in grado di costruire i set di dati su cui possiamo addestrare i modelli. Ci concentreremo su i due problemi di matematica simbolica scelti dai ricercatori: l'integrazione di funzioni e la risoluzione di equazioni differenziali ordinarie (ODE) del primo e del secondo ordine.

Per addestrare i modelli di reti neurali, abbiamo bisogno di grandi dataset contenenti input e soluzioni relativi ai problemi scelti. Idealmente, l'obiettivo è quello di generare dei campioni che siano rappresentativi dell'intero spazio dei problemi selezionati, ovvero dobbiamo cercare di generare casualmente funzioni da integrare di ogni tipo e varietà ed equazioni differenziali di primo e secondo grado di ogni forma possibile. Sfortunatamente, molto spesso le soluzioni a problemi generati casualmente non esistono, ad esempio gli integrali di $f(x) = e^{x^2}$ o $f(x) = \log(\log(x))$ non possono essere espressi con funzioni matematiche standard, o non possono essere facilmente derivate. Nelle sezioni successive, verranno dunque analizzate le tecniche che sono state utilizzate per generare i grandi dataset di addestramento e per superare questi problemi.

2.6.1 *Integrazioni*

Per l'integrazione sono stati proposti tre approcci per generare coppie di funzioni con i loro integrali associati.

Forward Generation (FWD) - Generazione in Avanti

Un approccio semplice consiste nel generare funzioni casuali con n operatori, usando le strategie e gli algoritmi della Sezione 2.5, e calcolando i loro integrali con un *Computer Algebra System*. Nel caso di questo studio dato che il codice utilizzato è stato sviluppato attraverso il linguaggio di programmazione *Python*, viene utilizzata SymPy [18], un'ottima libreria per la matematica simbolica scritta completamente in *Python* molto semplice da usare e facilmente estendibile.

Questo tipo di generatore va a creare dunque un insieme di coppie (f, F), che rispettivamente rappresentano la funzione da integrare e la sua soluzione, ovvero la sua primitiva. Le funzioni che il sistema non è in grado di integrare vengono invece scartate. Di conseguenza questo approccio genera solo un campione rappresentativo del sottoinsieme dello spazio del problema delle integrazioni che può essere risolto con successo da un qualsiasi framework di matematica simbolica.

Backward Generation (BWD) - Generazione a Ritroso

Un problema con l'approccio in avanti è che il set di dati generato conterrà sicuramente solo funzioni che i framework di matematica simbolica possono risolvere, dato che in determinati casi questi sistemi non riescono a calcolare correttamente l'integrale di funzioni integrabili. Inoltre, l'integrazione di grandi espressioni matematiche richiede tempo, il che rende il metodo in avanti complesso e particolarmente lento.

L'approccio a ritroso invece genera una funzione casuale f, calcola la sua derivata f' e aggiunge la coppia (f', f) al dataset, anche in questo caso dunque rispettivamente la funzione da integrare e la sua soluzione. A differenza dell'integrazione, la differenziazione è sempre possibile ed estremamente veloce anche per espressioni molto grandi. Inoltre, al contrario dell'approccio in avanti, questo metodo non dipende da un sistema di integrazione simbolica esterno.

Backward Generation con l'uso dell'Integrazione per Parti (IBP)

Un problema con l'approccio a ritroso è che risulterà molto improbabile che venga generato l'integrale di funzioni semplici come $f(x) = x^3 \sin(x)$. Il suo integrale infatti, $F(x) = -x^3 \cos(x) + 3x^2 \sin(x) + 6x \cos(x) - 6 \sin(x)$, una funzione con 15 operatori, ha un bassa probabilità di essere generata casualmente. Inoltre, l'approccio all'indietro tende a generare esempi in cui l'integrale (la soluzione) sarà più corta della derivata (il problema), mentre l'approccio in avanti favorisce l'opposto, ovvero funzioni corte e soluzioni lunghe.

Per affrontare questo problema, viene dunque sfruttato il metodo dell'integrazione per parti: date due funzioni F e G generate casualmente, vengono calcolate le loro rispettive derivate f e g. Se fG appartiene già al dataset generato tramite il calcolo delle derivate con l'approccio a ritroso, conosciamo di conseguenza anche il suo integrale e possiamo quindi calcolare l'integrale di Fg sfruttando il metodo di integrazione per parti:

$$\int Fg = FG - \int fG$$

Allo stesso modo, se Fg è già presente nel dataset, possiamo dedurre l'integrale di fG con la stessa strategia. Ogni volta dunque che viene scoperto l'integrale di una nuova funzione, esso viene aggiunto al dataset. Se nessuno tra fG o Fg è già presente nel dataset, vengono generate semplicemente nuove funzioni F e G con le rispettive derivate. Con questo approccio, possiamo generare gli integrali di funzioni come $x^{10} \sin(x)$ senza ricorrere ad un sistema di integrazione simbolica esterno.

2.6.2 *Equazioni Differenziali del Primo Ordine (ODE 1)*

Presentiamo ora il metodo usato per generare equazioni differenziali del primo ordine e le loro soluzioni. Partiamo da una funzione bivariata $F(x, y)$, tale che l'equazione $F(x, y) = c$, dove c è una costante, può essere risolta analiticamente in y. In altre parole, esiste una funzione bivariata f che soddisfa $\forall(x, c)$, $F(x, f(x, c)) = c$. Derivando F rispetto a x, avremo che $\forall(x, c)$:

$$\frac{\partial F(x, f_c(x))}{\partial x} + f_c'(x) \frac{\partial F(x, f_c(x))}{\partial y} = 0$$

dove $f_c = x \mapsto f(x, c)$. Di conseguenza, per ogni costante c, f_c è la soluzione della seguente equazione differenziale del primo ordine:

$$\frac{\partial F(x, y)}{\partial x} + y' \frac{\partial F(x, y)}{\partial y} = 0. \tag{2.7}$$

Con questo approccio, possiamo usare i metodi descritti nella Sezione 2.5 per generare funzioni arbitrarie $F(x, y)$ risolvibili analiticamente in y e creare così un dataset di equazioni differenziali con le loro soluzioni.

Invece di generare una funzione casuale F, possiamo generare una soluzione $f(x, c)$ e determinare un'equazione differenziale che viene soddisfatta da essa. Se $f(x, c)$ è risolvibile in c, calcoliamo F tale che $F(x, f(x, c)) = c$. Usando l'approccio mostrato prima, possiamo notare che per ogni costante c, $x \mapsto f(x, c)$ è una soluzione dell'Equazione Differenziale 2.7. Infine, l'equazione differenziale risultante viene fattorizzata e vengono rimossi tutti i fattori positivi dall'equazione.

Una condizione necessaria affinché questo approccio possa funzionare è che le funzioni generate $f(x, c)$ siano risolvibili rispetto alla variabile c. Ad esempio, la funzione $f(x, c) = c \times \log(x + c)$ non può essere risolta analiticamente in c, ovvero la funzione F che soddisfa $F(x, f(x, c)) = c$ non può essere scritta con funzioni matematiche standard. Dal momento che tutti gli operatori e le funzioni che sono state considerate in questo studio sono invertibili, una semplice condizione per garantire la risolvibilità in c è ad esempio garantire che c appaia solo una volta nelle foglie della rappresentazione ad albero di $f(x, c)$. Un strategia semplice quindi per generare un'opportuna $f(x, c)$ è campionare una funzione casuale $f(x)$ con i metodi descritti nella Sezione 2.5, e sostituire una delle foglie nella sua rappresentazione ad albero con c.

Di seguito usiamo un esempio per riportare i passaggi necessari per l'intero processo:

1. Viene generata una funzione casuale: $f(x) = x \log(c/x)$.
2. La funzione viene risolta in c: $c = xe^{\frac{f(x)}{x}} = F(x, f(x))$.
3. La funzione viene differenziata in x: $e^{\frac{f(x)}{x}}(1 + f'(x) - \frac{f(x)}{x}) = 0$.
4. Il risultato viene semplificato: $xy' - y + x = 0$.

2.6.3 *Equazioni Differenziali del Secondo Ordine (ODE 2)*

Il metodo per generare equazioni differenziali del primo ordine può essere esteso al secondo ordine, considerando funzioni di tre variabili $f(x, c_1, c_2)$ che sono risolvibili in c_2. Come nel precedente metodo, deriviamo una funzione di tre variabili F tali che $F(x, f(x, c_1, c_2), c_1) = c_2$.

Differenziando rispetto a x otteniamo un'equazione differenziale del primo ordine:

$$\frac{\partial F(x, y, c_1)}{\partial x} + f'_{c_1,c_2}(x)\frac{\partial F(x, y, c_1)}{\partial y}\bigg|_{y=f_{c_1,c_2}(x)} = 0$$

dove $f_{c_1,c_2} = x \mapsto f(x, c_1, c_2)$. Se questa equazione può essere risolta in c_1, possiamo dedurre un'altra funzione G in tre variabili tale che $\forall x$, $G(x, f_{c_1,c_2}(x), f'_{c_1,c_2}(x)) = c_1$. Differenziando una seconda volta rispetto a x otteniamo la seguente equazione:

$$\frac{\partial G(x, y, z)}{\partial x} + f'_{c_1,c_2}(x)\frac{\partial G(x, y, z)}{\partial y} + f''_{c_1,c_2}(x)\frac{\partial G(x, y, z)}{\partial z}\bigg|_{\substack{y=f_{c_1,c_2}(x)\\ z=f'_{c_1,c_2}(x)}} = 0$$

Pertanto, per ogni costante c_1 e c_2, f_{c_1,c_2} è la soluzione dell'equazione differenziale del secondo ordine seguente:

$$\frac{\partial G(x, y, y')}{\partial x} + y'\frac{\partial G(x, y, y')}{\partial y} + y''\frac{\partial G(x, y, y')}{\partial z} = 0$$

Usando questo approccio, possiamo creare coppie di equazioni differenziali del secondo ordine con le loro soluzioni, purché sia possibile generare una funzione $f(x, c_1, c_2)$ che sia risolvibile in c_2, e che la corrispondente equazione differenziale del primo ordine sia risolvibile in c_1. Per garantire la risolvibilità in c_2, possiamo usare lo stesso approccio usato per le equazioni differenziali del primo ordine, ad esempio viene creata f_{c_1,c_2} in modo tale che c_2 sia presente solo in una foglia della sua rappresentazione ad albero. Per c_1, viene invece utilizzato un approccio più semplice in cui viene scartata l'equazione corrente su cui si sta lavorando se non possiamo risolverla in c_1. Sebbene sia una strategia *naive*, i ricercatori hanno osservato che le equazioni ottenute dalle operazioni di derivazione risultano essere risolvibili in c_1 nel 50% circa dei casi, dunque questo procedimento riesce comunque a non essere eccessivamente lento e pesante dal punto di vista computazionale, nel caso si debba scartare un'equazione e tentare con una nuova generazione successiva.

Anche in questo caso di seguito usiamo un esempio per riportare i passaggi necessari per l'intero processo:

1. Viene generata una funzione casuale: $f = c_1 e^x + c_2 e^{-x}$.
2. La funzione viene risolta in c_2:
   $c_2 = f(x)e^x - c_1 e^{2x} = F(x, f(x), c_1)$.

3. La funzione viene differenziata in x:
   $e^x(f'(x) + f(x)) - 2c_1e^{2x} = 0$.
4. La funzione viene risolta in c_1:
   $c_1 = \frac{1}{2}e^{-x}(f'(x) + f(x)) = G(x, f(x), f'(x))$.
5. La funzione viene differenziata in x: $0 = \frac{1}{2}e^{-x}(f''(x) - f(x))$.
6. Il risultato viene semplificato: $y'' - y = 0$.

2.6.4 *Pulizia dei Dataset*

Utilizzando i metodi presentati nelle sezioni precedenti per generare dataset di espressioni matematiche è comunque possibile ottenere delle espressioni non ottimali o che non hanno proprio un significato matematico, per questo, durante la generazione di quest'ultime vengono eseguite alcune verifiche o correzioni per far sì che i dataset risultanti siano il più puliti e utili possibile. In pratica vengono eseguite tre operazioni di pulizia sulle espressioni generate: la semplificazione delle equazioni, la semplificazione dei coefficienti e la rimozione delle espressioni non valide. Di seguito approfondiamo le azione che vengono intraprese in ogni caso citato.

Semplificazione delle Equazioni

In pratica, vengono semplificate le espressioni generate dai metodi della Sezione 2.6 per ridurre il numero di possibili equazioni uniche nel dataset e per ridurre la lunghezza delle sequenze. Inoltre, in questo modo si evita di addestrare un modello per prevedere $x + 1 + 1 + 1 + 1 + 1$ quando può semplicemente considerare $x + 5$. Di conseguenza, le sequenze $[+\ 2\ +\ x\ 3]$ e $[+\ 3\ +\ 2\ x]$ saranno entrambe semplificate in $[+\ x\ 5]$ poiché ambedue rappresentano l'espressione $x + 5$. Allo stesso modo, l'espressione $\log(e^{x+3})$ sarà semplificata in $x + 3$, mentre $\cos^2(x) + \sin^2(x)$ sarà semplificata in 1. Al contrario invece, $\sqrt{(x-1)^2}$ non sarà snellita in $x - 1$ poiché non viene fatta alcuna ipotesi sul segno di $x - 1$.

Semplificazione dei Coefficienti

Nel caso di equazioni differenziali del primo ordine, le espressioni generate vengono modificate con espressioni equivalenti a meno di un cambio di variabile. Ad esempio, $x + x\tan(3) + cx + 1$ sarà semplificata in $cx + 1$,

poiché una scelta particolare della costante c rende queste due espressioni identiche. Allo stesso modo, $\log(x^2) + c\log(x)$ diventerà $c\log(x)$.

Una tecnica simile viene applicata anche per le equazioni differenziali del secondo ordine, sebbene in questo caso la semplificazione risulti a volte un po' più complicata per la presenza delle due costanti c_1 e c_2. Ad esempio, $c_1 - c_2 x/5 + c_2 + 1$ viene semplificata in $c_1 x + c_2$, mentre $c_2 e^{c_1} e^{c_1 x e - 1}$ viene espressa con $c_2 e^{c_1 x}$.

Vengono anche eseguite delle trasformazioni che non sono strettamente equivalenti, purché siano valide sotto specifiche ipotesi. Ad esempio, $\tan(\sqrt{c_2}x) + \cosh(c_1 + 1) + 4$ viene semplificata con $c_1 + \tan(c_2 x)$, sebbene il termine costante possa essere negativo nella seconda espressione ma non nella prima. In modo simile $e^3 e^{c_1 x} e^{c_1 \log(c_2)}$ viene trasformata in $c_2 e^{c_1 x}$.

Espressioni non Valide

Dai vari dataset vengono rimosse anche le espressioni matematiche non valide. Ad esempio, espressioni come $\log(0)$ o $\sqrt{-2}$. Per scovarle, viene calcolato nell'albero delle espressioni il valore dei sottoalberi che non dipendono da x. Se un sottoalbero non restituisce un numero reale finito, ad esempio $-\infty$, $+\infty$ o un numero complesso, l'espressione viene scartata.

2.6.5 *Confronto tra i Diversi Metodi di Generazione*

Nella Tabella 1 vengono riassunte le principali differenze tra i tre metodi di generazione rilevate dai due ricercatori nel loro studio, attraverso le dimensioni dei dataset generati e le lunghezze (in termini di *tokens*) delle espressioni contenute in essi.

	Forward (FWD)	**Backward (BWD)**	**Integrazione per Parti (IBP)**	**ODE 1**	**ODE 2**
Dimensione Dataset	20 M	40 M	20 M	40 M	40 M
Lunghezza Input	18.9 ± 6.9	70.2 ± 47.8	17.5 ± 9.1	123.6 ± 115.7	149.1 ± 130.2
Lunghezza Output	49.6 ± 48.3	21.3 ± 8.3	26.4 ± 11.3	23.0 ± 15.2	24.3 ± 14.9
Ratio Lunghezza	2.7	0.4	2.0	0.4	0.1
Max Lunghezza Input	69	450	226	508	508
Max Lunghezza Output	508	75	206	474	335

Tabella 1: Dimensioni dei dataset di training e lunghezza delle loro espressioni.

Per quanto riguarda l'integrazione, il metodo FWD (in avanti) tende a generare problemi brevi con soluzioni lunghe. L'approccio BWD (a ritroso), al contrario, tende a generare problemi lunghi con soluzioni

brevi. Mentre il metodo IBP (BWD con integrazione per parti) genera dei set di dati paragonabili al metodo FWD, dunque problemi brevi e soluzioni lunghe. Una combinazione di dati generati da BWD e IBP dovrebbe quindi fornire una migliore rappresentazione dello spazio del problema dell'integrazione, senza ricorrere a strumenti esterni.

I generatori per le equazioni differenziali (ODE 1 e 2) invece tendono a produrre soluzioni molto più brevi delle loro equazioni, questa caratteristica deriva dalla strategia dei loro generatori di partire direttamente dalla creazione delle soluzioni per poi ricavarsi con i vari passaggi le equazioni differenziali soddisfatte da esse.

3

NATURAL LANGUAGE PROCESSING E MODELLI SEQUENCE TO SEQUENCE

Il ramo del *Natural Language Processing* (NLP) è un campo dell'intelligenza artificiale che fornisce un ponte tra i linguaggi naturali e le macchine, aiutando quest'ultime a comprendere, elaborare e analizzare il linguaggio umano. Racchiude una grande varietà di argomenti che coinvolgono l'elaborazione computazionale e la comprensione delle lingue naturali, ad esempio si richiede alle macchine di eseguire attività simili a quelle umane come comprendere il significato di un testo, tradurre una lingua in un'altra, riconoscere il parlato e convertirlo in azioni significative, generare e riassumere un testo o trovare il senso di un argomento. Lo sviluppo di applicazioni NLP non risulta però un compito semplice dato che gli approcci tradizionali sono spesso poco performanti perché ignorano molte situazioni complesse da gestire, come il contesto dell'argomento, i dialetti o i gerghi usati nelle conversazioni. I dati dunque diventano più significativi attraverso una più profonda comprensione del loro contesto, che a sua volta ne facilita l'analisi [19, 21].

Dagli anni '80, questo campo si è sempre più basato su calcoli che coinvolgono la statistica, la probabilità e il *machine learning*. Recentemente, grazie all'aumento della potenza di calcolo e alla parallelizzazione concessa dalle moderne *Graphical Processing Unit* (GPU), è possibile sfruttare anche in questo campo le tecniche di *deep learning* (apprendimento profondo) che utilizzano alla base le reti neurali artificiali, a volte con miliardi di parametri da addestrare. Il *deep learning* ha già dimostrato di raggiungere prestazioni superiori in campi adiacenti come la *Computer Vision* e lo *Speech Recognition*, ed è in grado di superare in prestazioni e velocità anche l'uomo nella maggior parte dei casi. Inoltre, la contemporanea disponibilità di dataset sempre più grandi, ha facilitato il sofisticato processo di raccolta dei dati, consentendo il corretto addestramento di tali architetture profonde [19, 20].

Negli ultimi anni, ricercatori e professionisti nel campo dell'NLP hanno

dunque sfruttato la potenza delle moderne reti neurali ottenendo risultati vantaggiosi e l'uso del *deep learning* è aumentato considerevolmente. Questo ha portato a progressi significativi sia nelle aree centrali dell'NLP sia nelle aree in cui viene applicato direttamente per ottenere risultati pratici ed obiettivi utili. Tra tutte le applicazioni dell'NLP, i modelli basati sul *deep learning* come gli algoritmi *Sequence to Sequence* (Seq2Seq), e che comprendono le *Reccurent Neural Networks* (RNN), hanno fatto progressi significativi negli ultimi tempi nella traduzione automatica e hanno superato di gran lunga gli approcci tradizionali [19, 21].

In questo capitolo ci concentreremo dunque sui moderni approcci all'NLP basati sul *deep learning*, in particolare analizzando i principi e gli approcci di base utilizzati con le reti neurali e il funzionamento dei modelli Seq2Seq, che sono poi stati usati anche dal team di ricercatori di *Facebook* per risolvere i problemi di matematica simbolica discussi nel capitolo precedente.

3.1 NATURAL LANGUAGE PROCESSING

Il campo del *Natural Language Processing*, noto anche come linguistica computazionale, coinvolge l'ingegneria dei modelli e dei processi computazionali impiegati per risolvere problemi pratici sulla comprensione dei linguaggi naturali. Il lavoro dell'NLP può essere diviso in due ampie sotto-aree: aree centrali e applicative, anche se a volte è difficile distinguere chiaramente a quali aree appartengono i problemi che si possono considerare.

Le aree centrali affrontano quesiti fondamentali come la modellazione linguistica, che sottolinea e quantifica le associazioni che possono esserci tra le parole naturali di un testo, l'elaborazione morfologica, che si occupa della segmentazione di componenti significativi delle parole e identifica dunque le parti da cui possono essere formate, l'elaborazione sintattica, o *parsing*, che permette di costruire diagrammi di frasi, oppure l'elaborazione semantica, che tenta di distillare il significato di parole, frasi e componenti di livello superiore in un testo.

Il aree applicative coinvolgono invece argomenti come l'estrazione di informazioni utili da un testo, ad esempio le entità e le relazioni coinvolte, la traduzione di testi in lingue diverse, la sintesi di opere scritte, la risposta automatica a determinate domande deducendone le risposte, oppure la classificazione e il raggruppamento di documenti. Molto spesso per risolvere questi problemi pratici con successo è necessario gestire uno o più dei problemi delle aree centrali e applicare poi queste idee alle

procedure implementate per la risoluzione dei quesiti.

Attualmente, l'NLP è un campo basato principalmente su tecniche che utilizzano calcoli statistici e probabilistici insieme al *machine learning* e al *deep learning*. In passato, venivano ampiamente usati approcci di apprendimento automatico come i classificatori *Naive Bayes*, i *k-nearest neighbors*, i modelli di *Markov* nascosti, gli alberi decisionali, i *random forests* e le *Support Vector Machines* (SVM). Tuttavia, negli ultimi anni, c'è stata una trasformazione totale e questi approcci sono stati completamente sostituiti, o almeno potenziati, dai modelli basati su reti neurali e *deep learning*, come discuteremo di seguito [19].

3.2 RETI NEURALI E DEEP LEARNING

Le reti neurali sono modelli computazionali composti da "neuroni" artificiali, ispirati vagamente dalla semplificazione di una rete neurale biologica, come il nostro cervello. Sono composte da nodi interconnessi, chiamati appunto neuroni, ciascuno dei quali riceve un certo numero di input e fornisce un dato output. Ogni connessione, come le sinapsi in un cervello biologico, può trasmettere un segnale ad altri neuroni. Un neurone artificiale riceve un "segnale" che viene elaborato e può essere inviato ai neuroni ad esso collegati. Il segnale in una connessione è in realtà un numero reale, e l'output di ciascun neurone è calcolato da una funzione non lineare basata sulla somma ponderata dei suoi ingressi, chiamata anche *activation function*. I neuroni e le connessioni in genere hanno associato un peso (*weight*) che si adatta man mano che l'apprendimento della rete procede. Il peso aumenta o diminuisce la potenza del segnale in corrispondenza di una connessione. Tipicamente, i neuroni sono aggregati in *layers*, che possono eseguire trasformazioni diverse sui loro input. I segnali dunque viaggiano dal primo *layer*, chiamato *layer* di input, all'ultimo *layer*, denominato *layer* di output, possibilmente dopo aver attraversato gli altri vari *layer* di mezzo più volte, chiamati anche *hidden layers* [19]. In Figura 8 viene mostrato un esempio di una semplice rete neurale artificiale.

Una rete neurale per apprendere ed adattarsi al meglio al compito a cui ambisce considera le osservazioni campione che le vengono proposte in input, un vero e proprio dataset di addestramento. L'apprendimento comporta la regolazione dei pesi e delle soglie (*bias*) opzionali della rete per migliorare l'accuratezza del risultato finale, l'output della rete. Questo viene fatto riducendo al minimo gli errori osservati sul set di addestramento. L'apprendimento della rete viene considerato comple-

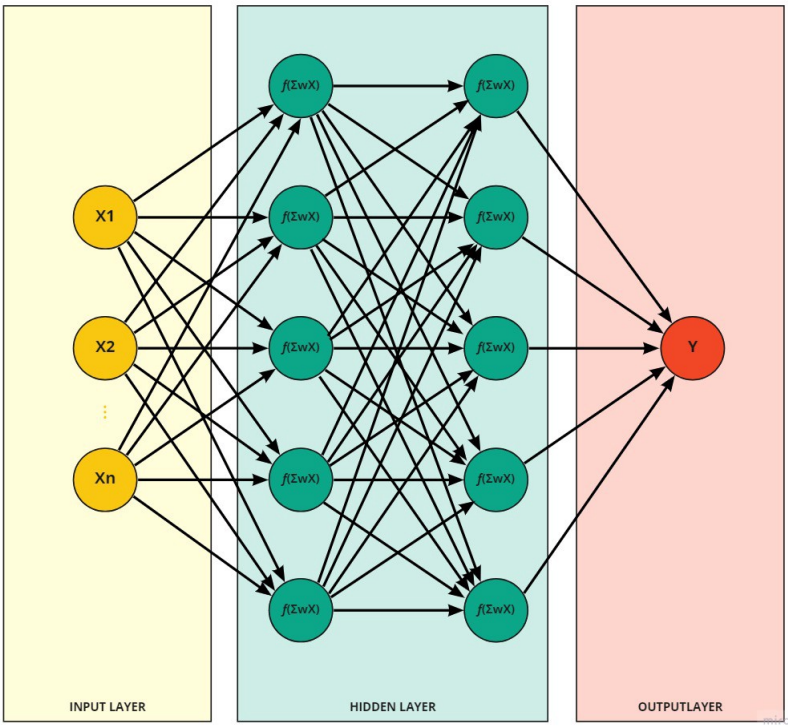

Figura 8: Esempio di una semplice rete neurale artificiale.

tato quando l'esame delle osservazioni aggiuntive non riduce in modo significativo il tasso di errore. Anche dopo l'apprendimento, il tasso di errore in genere non raggiunge lo zero e se dopo l'addestramento, il tasso di errore è ancora troppo alto, la rete deve essere in genere riprogettata. Per calcolare il tasso di errore viene in pratica definita una funzione di costo che viene valutata periodicamente durante l'apprendimento. Finché il risultato della funzione di costo diminuisce, l'apprendimento viene portato avanti. Il costo è spesso definito come una statistica il cui valore può essere solo approssimato. Gli output di tale funzione sono in realtà numeri, quindi quando l'errore è basso, la differenza tra l'output che la rete restituisce e la risposta corretta presa dal set di addestramento sarà di conseguenza piccola. La maggior parte dei modelli di apprendimento può essere vista come un'applicazione diretta della teoria dell'ottimizzazione e della stima statistica. Le correzioni ai pesi della rete che vengono utilizzati per calcolare il suo output sono dunque effettuati in risposta ai singoli errori o alle perdite che le rete presenta ai nodi di uscita in base alla funzione di costo. Tali correzioni nelle reti moderne sono solitamente realizzate utilizzando il metodo della discesa stocastica del gradiente, che

considera le derivate degli errori ai nodi della rete, usando un approccio chiamato *backpropagation*. Il *backpropagation* consiste nel calcolare in modo estremamente efficiente il gradiente della funzione di costo rispetto ai pesi della rete per ogni singolo esempio input-output, calcolando il gradiente un *layer* alla volta ed iterando all'indietro dall'ultimo *layer* per evitare calcoli ridondanti. Questa efficienza rende possibile l'utilizzo di metodi come la discesa stocastica del gradiente per l'addestramento delle reti, aggiornando i pesi per ridurre al minimo l'errore. Viene scelto inoltre anche un tasso di apprendimento (*learning rate*) che definisce la dimensione dei passaggi correttivi che il modello adotta per correggere gli errori in ogni osservazione. Un alto *learning rate* riduce il tempo di addestramento, ma al costo di una precisione finale della rete inferiore, mentre un tasso di apprendimento minore richiede un tempo di addestramento più lungo, ma con il potenziale per una maggiore precisione [19].

Il *deep learning* si riferisce all'applicazione di reti neurali profonde a enormi quantità di dati per apprendere una procedura finalizzata alla gestione di un compito. Quest'ultimo può spaziare da un semplice problema di classificazione fino ad arrivare a problemi di ragionamento complessi. In altre parole, il *deep learning* è un insieme di meccanismi in grado di derivare idealmente una soluzione ottimale a qualsiasi problema dato come input un set di dati sufficientemente ampio e rilevante. Qui, l'intelligenza artificiale e il *deep learning* si incontrano. Un'obiettivo o l'ambizione dietro l'intelligenza artificiale è quello di consentire ad una macchina di superare le prestazioni di un cervello umano. Il *deep learning* è un mezzo per raggiungere questo scopo. Anche se non c'è un chiaro consenso su cosa esattamente definisca una rete neurale profonda, generalmente vengono considerate reti di *deep learning* quelle reti che sono formate da più *hidden layers*, e quelle formate da molti *layers* sono considerate molto profonde [19, 20].

Esistono numerose architetture di *deep learning* che sono state sviluppate in diverse aree di ricerca, ed in particolare per distinguere tra loro i diversi tipi di reti neurali profonde possiamo considerare due aspetti: il modo in cui sono collegati i nodi della rete e il numero di *layers* usati. Di seguito ci concentreremo sulla descrizione di alcuni tra i modelli essenziali che vengono utilizzati sopratutto nel campo dell'NLP per poi approfondire in particolare le reti che compongono i modelli *Sequence to Sequence*.

3.2.1 *Multi Layer Perceptron e Feed-Forward Neural Networks*

Una delle reti neurali più semplici è rappresentata dal *Multi Layer Perceptron* (MLP), formata da almeno tre *layers* (*input*, *hidden* e output *layer*). Nell'architettura MLP, i neuroni in un singolo *layer* non sono connessi fra di loro e, ad eccezione del *layer* di input, utilizzano ognuno un'*activation function* non lineare. Ogni nodo all'interno di un *layer* si connette a tutti i nodi del successivo *layer*, creando dunque una rete completamente connessa [20].

I *Multi Layer Perceptron* sono il tipo più semplice di reti neurali *Feed-Forward* (FNN). Le FNN rappresentano una categoria generale di reti neurali in cui le connessioni tra i nodi non creano alcun ciclo, ovvero, dove tutti i nodi possono essere organizzati in *layer* sequenziali, con ogni nodo che riceve gli input solo dai nodi nei *layer* precedenti, e dunque in una FNN non c'è un flusso ciclico di informazioni [19, 20].

La rete mostrata in Figura 8 rappresenta un esempio di rete MLP e quindi di conseguenza anche FNN.

3.2.2 *Convolutional Neural Networks*

Le reti neurali convoluzionali (CNN), la cui architettura si ispira alla corteccia visiva umana, sono una sottoclasse di reti neurali *feed-forward*. Le CNN prendono il nome dall'operazione matematica di convoluzione, che fornisce una misura dell'interoperabilità delle sue funzioni di ingresso. Le CNN usano inoltre diverse funzioni, note come filtri, che consentono l'analisi simultanea di diverse caratteristiche nei dati. Le reti neurali convoluzionali sono solitamente impiegate in situazioni in cui i dati sono o devono essere rappresentati con una mappa dati 2D o 3D, dunque sono ampiamente utilizzate nell'elaborazione di immagini e video ed in generale nel campo della *Computer Vision* ma anche dell'NLP. Nella rappresentazione di una mappa dati, la vicinanza tra punti dati di solito corrisponde alla loro correlazione informativa [20].

Nelle reti neurali convoluzionali in cui l'input è un'immagine, ad esempio, i vari pixel dell'immagine sono altamente correlati ai pixel adiacenti. Di conseguenza, i *layer* convoluzionali hanno tre dimensioni: larghezza, altezza e profondità. Una CNN prende un'immagine rappresentata come un array di valori numerici e dopo aver eseguito specifiche operazioni matematiche, l'immagine viene rappresentata in un nuovo spazio di output. Questa operazione è chiamata anche estrazione di funzionalità e aiuta a catturare e rappresentare il contenuto dell'immagine. In Figura 9 viene

mostrata un esempio di rete CNN per la classificazione di immagini. Le caratteristiche estratte possono essere utilizzate per ulteriori analisi e per compiti diversi, dove spesso non è importante capire esattamente dove certe caratteristiche si verificano, ma piuttosto se appaiono o meno in zone particolari. Un esempio è la classificazione delle immagini, che mira a raggruppare le figure in base ad alcune classi predefinite da ricercare. Altri esempi includono determinare quali oggetti sono presenti in un'immagine e dove sono situati nell'immagine [19, 20].

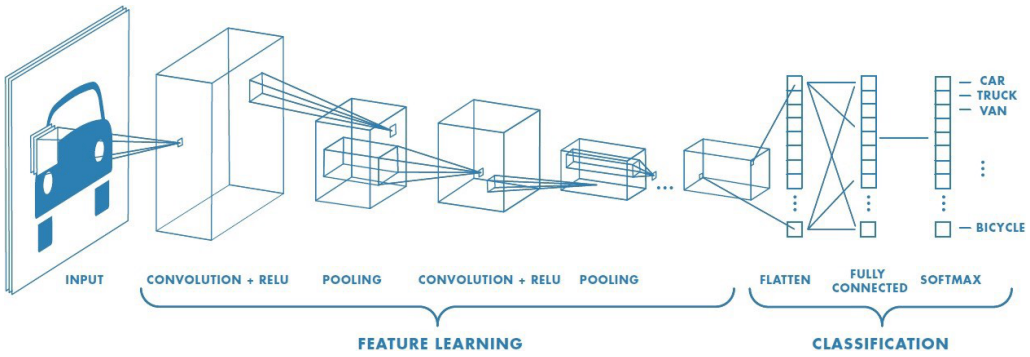

Figura 9: Esempio di una rete neurale convoluzionale per la classificazione di immagini.

Nel caso di utilizzo delle CNN per l'NLP, gli input forniti a tali reti sono in genere frasi o documenti rappresentati come matrici. Ogni riga della matrice è associata a un elemento del linguaggio come una parola o un personaggio. Una grande varietà di questo tipo di reti sono state utilizzate in vari compiti di classificazione come l'analisi del *sentiment*, che si occupa di costruire sistemi per l'identificazione ed estrazione di opinioni da un testo, la categorizzazione degli argomenti o l'estrazione e la classificazione delle relazioni [20].

3.2.3 *Recurrent e Long Short-Term Memory Neural Networks*

Le reti neurali ricorrenti sono tra i modelli di *deep learning* più utilizzati nelle attività dell'NLP. In questo tipo di reti, i neuroni possono ammettere dei cicli e possono essere interconnessi anche a neuroni di un *layer* precedente, ovvero, le connessioni tra i nodi formano un grafo orientato lungo una sequenza temporale, questo consente loro di esibire un comportamento temporale dinamico. Come le FNN, i *layers* in una RNN possono essere classificati come *layer* di input, *hidden* e di output. In tempi successivi, vengono passati in input alla rete delle sequenze di vettori, un vettore

alla volta. Dopo aver passato in input un *batch* (gruppo) di vettori e dopo aver condotto alcune operazioni su di essi insieme all'aggiornamento dei pesi, verrà inviato alla rete il prossimo *batch*.

In Figura 10 viene mostrata un esempio di una rete neurale ricorrente, dove a sinistra viene raffigurata in versione compatta mentre a destra in versione espansa. Con $t-1$, t e $t+1$ vengono rappresentati i passi temporali, x indica l'input della rete, in genere vettori di parole se abbiamo a che fare con un problema dell'NLP, h indica l'*hidden layer* e "o" l'output della rete, in genere la distribuzione di probabilità dell'output su un vocabolario di parole ad ogni passo temporale t, mentre U, V e W rappresentano i pesi della rete. Ciascuno degli *hidden layers* h ha un certo numero di neuroni, ognuno dei quali esegue un'operazione lineare matriciale sui propri input seguita da un'operazione non lineare. Ad ogni passo temporale t, vengono passati all'*hidden layer* due input: l'output del *layer* precedente h_{t-1}, chiamato anche *hidden state*, e l'input a quel passo temporale x_t. Il primo input viene moltiplicato con la matrice dei pesi V e il secondo con la matrice dei pesi U per produrre i parametri di output h_t, che sono moltiplicati con la matrice dei pesi W e il risultato viene applicato ad una *activation function* (di solito la funzione *softmax* [22]) sul vocabolario delle parole per ottenere un output di previsione o_t della parola successiva da generare. Infine dunque, i parametri dell'*hidden layer* h_t corrente, ovvero il suo *hidden state*, viene passato come input per il passo temporale successivo [23].

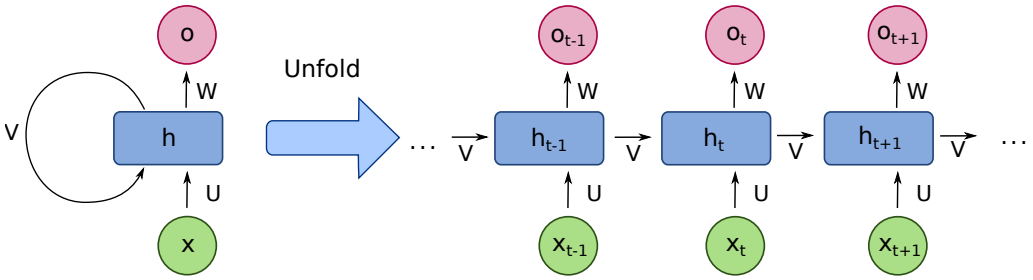

Figura 10: Esempio di una rete neurale ricorrente con versione espansa a destra.

Durante l'esecuzione di attività legate all'NLP, in genere ci si aspetta di processare una sequenza di parole, ad esempio nel caso delle traduzioni, la previsione della parola successiva da tradurre è fortemente influenzata dal precedente insieme di parole analizzate in una determinata frase. Inoltre nella gran parte dell'NLP risulta fondamentale conoscere l'ordine delle parole o altri elementi come fonemi o frasi, è dunque molto utile avere memoria degli elementi precedenti durante l'elaborazione

di nuovi dati. Gli *hidden layers* nelle reti neurali ricorrenti possono trasportare informazioni del passato, o in altre parole, rappresentano una sorta di memoria di quello che è stato analizzato precedentemente, è quindi proprio per questa caratteristica che le RNN sono particolarmente utili per applicazioni che trattano con una sequenza di input come per l'elaborazione del linguaggio naturale. Se ad esempio consideriamo la frase: "Michael Jackson era un cantante, alcune persone lo considerano il re del pop", è molto facile per una persona identificare che il termine "lo" sia riferito a Michael Jackson. Il pronome "lo" compare sei parole dopo Michael Jackson, catturare questa dipendenza è uno dei vantaggi delle RNN, dove gli *hidden layers* fungono da unità di memoria.

A volte, le parole che ricorrono nell'ultima parte del testo sono importanti anche per predire la parola corrente ed in generale la corretta elaborazione di alcune parole può dipendere da quelle che seguono. Pertanto, risulta spesso importante analizzare queste frasi sia all'indietro sia in avanti, utilizzando due *layer* RNN e combinando i loro output. Questa forma di gestione delle parole è chiamata RNN bidirezionale.

Tuttavia, le RNN hanno un grave svantaggio, ovvero riuscire a memorizzare parole precedenti in una frase che sono molto lontane dalla parola corrente che si sta analizzando. Questo problema è dovuto al fatto che durante la fase di addestramento della rete, i valori di aggiornamento dei pesi ricavati tramite la procedura del *backpropagation* tendono a diminuire sempre di più ad ogni passo per ripercorrere all'indietro la rete, un problema conosciuto anche come *vanishing gradient problem* [24].

Una delle architetture più potenti usate per l'NLP che deriva dalle RNN è la rete neurale *Long Short-Term Memory* (LSTM). Nelle reti LSTM, i nodi ricorsivi sono composti da diversi neuroni individuali collegati e progettati in modo tale da conservare, dimenticare o esporre informazioni specifiche. Le RNN standard con singoli neuroni che alimentano se stessi tecnicamente hanno qualche ricordo sui risultati dal lungo passato, quest'ultimi però vengono diluiti eccessivamente ad ogni successiva iterazione. Spesso è importante ricordare le informazioni dal lontano passato della rete, mentre, allo stesso tempo, altre informazioni molto più recenti potrebbero non essere importanti. Utilizzando reti LSTM, queste importante informazioni possono essere conservate molto più a lungo mentre quelle irrilevanti saranno dimenticate. Negli ultimi anni una variante un po' più semplice delle LSTM, chiamata *Gated Recurrent Unit* (GRU), ha dimostrato di funzionare altrettanto bene o meglio dello standard LSTM in molti compiti [19, 20, 21].

3.2.4 *Attention Mechanisms e il Modello Transformer*

Per compiti come la traduzione automatica, il riassunto di un testo o la creazione di didascalie, l'output che si vuole ottenere è in forma testuale ed in genere questi obiettivi vengono raggiunti attraverso l'utilizzo di modelli strutturati a coppie di moduli, un *encoder* ed un *decoder*, come approfondiremo anche in seguito quando parleremo dei modelli *sequence to sequence*. In particolare come mostrato in Figura 11, viene utilizzata una rete neurale di codifica (l'*encoder*) per cercare di stipare tutte le informazioni rilevanti nel testo di input in un vettore di lunghezza fissa, chiamato anche *context vector* (vettore del contesto), e viene utilizzata successivamente una rete neurale di decodifica (il *decoder*) per restituire il testo di output con una lunghezza variabile basato sul *context vector*. In particolare, questi due moduli sono in genere costruiti a partire da reti neurali ricorrenti.

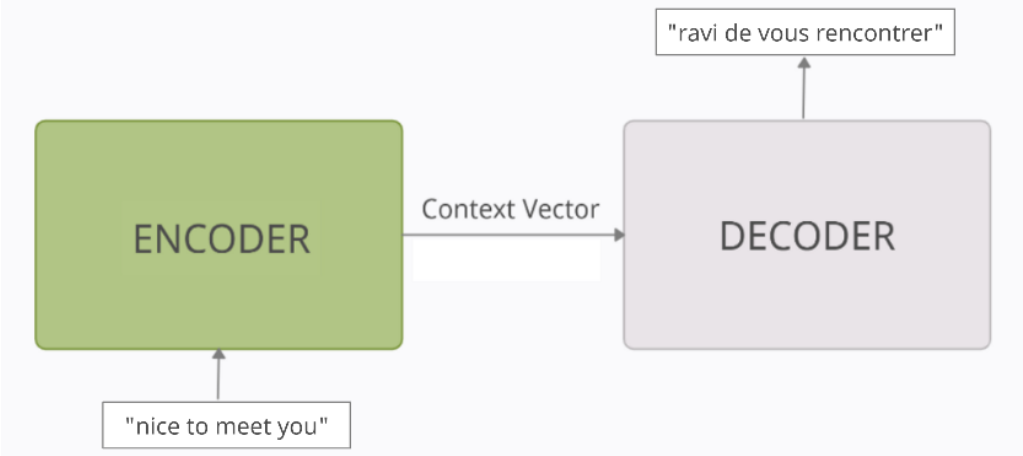

Figura 11: Esempio di modello *Encoder-Decoder* per la traduzione di un testo.

Uno degli svantaggi di questo schema è dato dal fatto che le reti che lo implementano sono costrette a codificare un'intera sequenza di testo in un vettore di lunghezza finita, indipendentemente dal fatto che alcune parti dell'input possano rivelarsi più importanti di altre nei compiti che questi tipi di modelli cercano di risolvere [19].

Una soluzione robusta a questo problema è l'utilizzo all'interno di questi modelli dell'*attention mechanism*. Quando leggiamo la frase "la palla è in campo", non assegniamo la stessa importanza a tutte le cinque parole ma prendiamo principalmente in considerazione le parole "palla", "in" e "campo", ovvero concentriamo la nostra attenzione sulle parole che sono più "importanti" e che racchiudono il significato della frase. Allo

stesso modo, *Bahdanau et al.* (2014) [14], introducendo per la prima volta il concetto di *attention*, hanno notato che usare solo l'*hidden state* finale di una RNN come singolo *context vector* per modelli *sequence to sequence* rappresenti un difetto di queste reti, dato che spesso, diverse parti di un input hanno diversi livelli di significatività ed in particolare le RNN tradizionali hanno grandi difficoltà nel mantenere fino ai passi finali di codifica del *context vector* il significato di un testo molto lungo. Inoltre, diverse parti dell'output per essere generate possono anche considerare svariate ed "importanti" parti dell'input. Ad esempio, nella traduzione di un testo, la prima parola di output è solitamente basata sulle prime poche parole dell'input, ma anche l'ultima parola da tradurre probabilmente deve essere basata sulle ultime poche parole dell'input. Gli *attention mechanisms* fanno uso di questa osservazione fornendo alla rete del *decoder* uno sguardo sull'intera sequenza di input ad ogni fase di decodifica. Il *decoder* può quindi decidere quali sono le parole di input più importanti in qualsiasi momento.

Un modello basato sull'*attention* impara dunque ad assegnare un significato di importanza a diverse parti dell'input per ogni fase dell'output. Nel contesto della traduzione, l'*attention* può essere anche pensata come una sorta di "allineamento". *Bahdanau et al.* (2014) [14] sostengono che i punteggi di *attention* α_{ij} generati al passo i di decodifica evidenziano le parole nella frase di origine che si allineano con la parola i di output. Notando questo, i punteggi di *attention* ottenuti in fase di addestramento della rete possono anche essere utilizzati per costruire una tabella di allineamento, che mappa le parole nella sorgente di input alle parole corrispondenti nella frase di output [25].

Il principale vantaggio dei modelli basati sull'*attention* è la loro capacità di tradurre in modo efficiente frasi lunghe. All'aumentare della dimensione dell'input infatti, i modelli che non usano l'*attention* mancheranno di informazioni e precisione se usano solo il *context vector* finale generato dall'*encoder*. L'*attention* è un modo intelligente per risolvere questo problema e vari studi confermano effettivamente questa intuizione, inoltre questo meccanismo è stato applicato in varie applicazioni NLP come il riassunto di un testo, la traduzione di linguaggi naturali, la classificazione del *sentiment* o legate ad attività con domande e risposte, provocando un miglioramento significativo nei risultati. Un altro vantaggio dell'uso di questo meccanismo è anche l'aiuto che esso da alla comprensione dei modelli di reti neurali che lo sfruttano. Tradizionalmente infatti, la comprensione dei modelli basati su reti neurali è molto difficile e talvolta purtroppo anche impossibile, costringendo a considerare tali mo-

delli a scatola nera, il meccanismo dell'*attention* aiuta a risolvere questo problema fino ad una certa misura [21, 25].

Nel tempo sono state ideate anche molte varianti dell'*attention mechanism*, una tra quelle più popolari è la cosiddetta *self-attention*. La *self-attention* si basa sul mettere in relazione diverse posizioni di una singola sequenza per calcolarne una sua rappresentazione. Ad esempio, nel caso di una frase di un linguaggio naturale la *self-attention* cercherà di estrapolare la correlazione che esiste tra le parole della stessa frase man mano che vengono processate durante la fase di codifica, oppure nel corso della fase di decodifica può cercare di trovare una correlazione tra le parole che sono già state prodotte in precedenza e la prossima parola di output.

La *self-attention* in particolare, è stata ampiamente utilizzata in un modello *encoder-decoder* considerato ormai lo stato dell'arte per le applicazioni dell'NLP e che sta guadagnando sempre più popolarità negli ultimi anni, ovvero il modello *Transformer*, proposto per la prima volta da *Vaswani et al.* in *Attention Is All You Need* (2017) [3] e utilizzato anche dai due ricercatori *Lample* e *Charton* in *Deep Learning for Symbolic Mathematics* (2019) [1] per risolvere i problemi di matematica simbolica discussi nel capitolo precedente. Il modello *Transformer*, di cui parleremo meglio nel capitolo successivo, sfrutta un numero di *encoders* e *decoders* impilati uno sopra l'altro ed usa il meccanismo di *self-attention* in ciascuna di queste unità in una elaborazione parallela, evitando dunque l'uso di ricorrenze o convoluzioni e garantendo una fase di addestramento molto più rapida rispetto ai modelli concorrenti [19, 21].

3.2.5 *Connessioni Residue e Dropout*

Nelle reti profonde, addestrate tramite la *backpropagation*, i gradienti utilizzati per correggere gli errori della rete spesso tendono a svanire o esplodere, causando il cosiddetto *vanishing gradient problem* [24]. Questo comportamento negativo può essere mitigato scegliendo delle apposite *activation function*, come il *Rectified Linear Unit* (ReLU), che non presenta regioni che sono particolarmente ripide o con gradienti eccessivamente piccoli. Per risolvere questo problema vengono spesso utilizzate anche delle connessioni residue all'interno delle reti neurali. Tali connessioni permettono semplicemente ai segnali inviati nella rete di saltare uno o più *layers*. Se utilizzate in alternanza od ogni *layer*, questo comporterà il dimezzamento del numero di *layer* che il gradiente dovrà attraversare tramite la *backpropagation*. Una rete di questo tipo è anche chiamata come rete residuale (ResNet).

Un altro metodo molto importante utilizzato nell'addestramento delle reti neurali è il *dropout*. In questo approccio, alcune connessioni e spesso anche diversi nodi della rete vengono disattivati, di solito in modo casuale, per ogni *batch* di addestramento (un piccolo set di dati), variando quali nodi verranno disattivati ad ogni *batch*. Questo costringe la rete a distribuire la sua memoria attraverso più percorsi, aiutando dunque con la generalizzazione del modello e riducendo la probabilità di *overfitting* sui dati di addestramento [19].

3.2.6 *Word Vectors*

Uno dei più importanti fattori che hanno in comune tutti i compiti dell'NLP è il modo in cui vengono rappresentate le parole come input per uno dei qualsiasi modelli di questo campo. Per svolgere bene la maggior parte delle attività legate all'NLP è necessario avere qualche nozione di somiglianza e differenza tra parole. Con l'uso dei *word vectors*, possiamo facilmente codificare questa capacità nei vettori stessi utilizzando varie tecniche e strategie.

I linguaggi naturali sono in genere composti da decine di milioni di *tokens*, tra parole e simboli, ma non tutti sono completamente incorrelati fra loro. Ad esempio le parole felino e gatto, oppure hotel e motel hanno sicuramente una forte correlazione dato che si riferiscono ad entità simili. L'obbiettivo ideale è quello dunque di codificare ogni *token* in un apposito vettore che rappresenta un punto in una sorta di spazio delle "parole". Questo è fondamentale per una serie di motivi, ma il motivo più intuitivo è che probabilmente esiste effettivamente uno spazio N-dimensionale che sia sufficiente per codificare tutta la semantica di una particolare linguaggio naturale. Ogni dimensione codificherebbe un distinto significato che vogliamo trasmettere usando una parola. Ad esempio, le dimensioni semantiche potrebbero indicare il tempo (passato, presente o futuro), un conteggio (singolare o plurale) oppure un genere (maschile o femminile). In questo contesto parole che hanno un significato simile avranno di conseguenza anche una rappresentazione vettoriale molto affine. I *word vectors* sono dunque utilizzati per la rappresentazione di parole per l'analisi del testo, in genere sotto forma di un vettore con valori reali, che codificano il significato delle parole in modo tale che le parole più vicine nello spazio vettoriale avranno anche un significato simile. Esistono molte tecniche e approcci per ricavare questi vettori, di seguito vengono riportate alcune di queste strategie.

Il *one-hot vector* è probabilmente uno dei metodi più semplici. In questo

caso ogni parola viene rappresentata come un vettore $\in \mathbb{R}^{|V|\times 1}$ contenente tutti 0 e un solo 1 nella posizione di quella parola nel vocabolario ordinato del linguaggio a cui fa riferimento. In questa notazione, $|V|$ è la dimensione del vocabolario. Con questo metodo rappresentiamo ogni parola come un'entità completamente indipendente e il forte svantaggio è quello di non riuscire a dare una qualsiasi nozione di somiglianza tra le parole.

Un altra classe di metodi è data dall'utilizzo di tecniche basate sulla *Singular Value Decomposition* (SVD), una forma di fattorizzazione. Con questi approcci prima di tutto viene analizzato in ciclo un enorme dataset di testo per accumulare dei conteggi di co-occorrenza di parole in una qualche forma di matrice X, e poi viene eseguita la *Singular Value Decomposition* su X per ottenere una decomposizione USV^T. Infine vengono utilizzate le righe di U come *word vectors* per tutte le parole del dizionario scelto.

Altri metodi sono basati invece sull'ottenere degli *word embeddings*, ovvero una classe di tecniche in cui vengono ricavati sempre dei *word vectors* ma dove i valori del vettore vengono appresi in modi che si avvicinano molto ad una rete neurale, e quindi tali metodi sono spesso raggruppati nel campo del *deep learning*. L'idea chiave di questi approcci è quello di utilizzare una rappresentazione distribuita densa per ogni parola. I metodi di *word embeddings* apprendono una rappresentazione vettoriale a valori reali basata su un vocabolario di dimensioni fisse predefinito partendo da un testo di addestramento. Spesso questi vettori hanno decine o centinaia di dimensioni, rispetto alle migliaia o milioni di dimensioni richieste per rappresentazioni sparse, come gli *one-hot vector*.

Un tecnica usata per apprendere gli *word embeddings* consiste nell'uso di un *embedding layer*, ovvero, tramite l'aggiunta di un vero e proprio *layer* di input nel modello di rete neurale per un'applicazione NLP. Questo permette dunque di ricavare gli *word embeddings* insieme all'addestramento della rete. La dimensione dello spazio vettoriale è specificata come parte del modello, ad esempio 50, 100 o 300 dimensioni, ed i vettori sono inizializzati con piccoli numeri casuali. L'*embedding layer* viene utilizzato dunque nella parte iniziale di una rete neurale e viene addestrato in modo supervisionato utilizzando l'algoritmo di *backpropagation*. Questo approccio all'apprendimento di un *embedding layer* richiede molti dati di addestramento e può risultare lento, ma ha il vantaggio di apprendere dei *word vectors* calibrati sia sui dati specifici del testo di addestramento sia sull'attività NLP scelta [26, 27].

3.3 MODELLI SEQUENCE TO SEQUENCE

La maggior parte dei framework nelle applicazioni NLP si basano su modelli *Sequence to Sequence* (Seq2Seq) in cui sia l'input che l'output sono rappresentati tramite sequenze di lunghezza potenzialmente variabile. Tali sequenze sono composte da *tokens* che in generale possono rappresentare qualsiasi elemento a seconda del compito che vogliamo risolvere, come ad esempio parole, lettere o simboli.

Questi modelli sono molto utilizzati in varie applicazioni dell'NLP, incluse per esempio:

- **Traduzioni**: prendere una frase in una lingua naturale come input e produrre la stessa frase in un'altra lingua.
- **Conversazioni**: prendere un'affermazione o una domanda come input e generare una risposta.
- **Riassunti**: prendere un testo molto lungo come input e generare un suo riassunto.

I modelli *Sequence to Sequence* basati sulle tecniche di *deep learning* si sono rivelati estremamente efficaci su questi tipi di problemi e negli ultimi anni sono diventati il nuovo standard, in particolare per la traduzione automatica [20, 25].

3.3.1 *Approcci Precedenti*

In passato, i sistemi di traduzione erano basati su modelli probabilistici costruiti a partire da due componenti:

- Un **modello di traduzione**: che ha il compito di valutare in cosa sia più probabile tradurre una parola o frase presa da un linguaggio.
- Un **modello linguistico**: che ha il compito di valutare quanto sia probabile che si presenti una data parola o frase.

Questi componenti sono stati utilizzati per costruire sistemi di traduzione basati su parole o frasi. Come ci si potrebbe aspettare, un sistema *naive* basato solo sulle parole non riuscirebbe completamente a catturare le differenze di sintassi o l'ordine degli elementi tra lingue, ad esempio capire dove vanno le parole di negazione, la posizione del soggetto e dei verbi in una frase, e altre caratteristiche di questo tipo.

I sistemi basati sulle frasi invece erano molto più comuni prima dei modelli Seq2Seq. Un sistema di traduzione di questo tipo può considerare input e output in termini di sequenze di frasi e può gestire sintassi più complesse rispetto ai sistemi basati su parole. Tuttavia, le dipendenze a lungo termine sono anche in questo caso abbastanza difficili da catturare usando dei sistemi basati su frasi.

Il principale vantaggio che i modelli Seq2Seq hanno portato, soprattutto con l'uso di reti LSTM, è stata la possibilità per i moderni sistemi di traduzione di generare sequenze di output arbitrarie dopo aver analizzato l'intero input. Inoltre, questi sistemi sono anche in grado di concentrarsi su parti specifiche dell'input automaticamente per facilitare la generazione di una traduzione corretta e utile [25].

3.3.2 *Nozioni di Base*

I modelli *Sequence to Sequence*, o "Seq2Seq", sono un paradigma relativamente nuovo di architetture per l'NLP, presentati per la prima volta da *Sutskever et al.* (2014) [13] per la traduzione di testi dall'inglese al francese. La maggior parte dei framework per applicazioni NLP si basano su modelli Seq2Seq in cui non solo l'input ma anche l'output è rappresentato come una sequenza di *tokens*. Questi modelli sono molto frequenti in varie applicazioni, incluse la traduzione automatica, il riassunto del testo e la sintesi vocale.

In generale un modello *sequence to sequence* è composto da due moduli che sono rappresentati da due reti neurali ricorrenti:

- Un **encoder**: che accetta l'input del modello, ovvero una sequenza di *tokens*, e successivamente lo codifica in un *context vector* (vettore del contesto) di dimensione fissa.
- Un **decoder**: che utilizza il *context vector* generato dall'*encoder* come "seme" da cui generare una sequenza di output.

Per questo motivo, i modelli Seq2Seq vengono spesso definiti come modelli *encoder-decoder* [20, 25]. In Figura 12 viene mostrato un esempio di questa architettura. Di seguito analizzeremo in modo separato i dettagli di queste due reti ricorrenti.

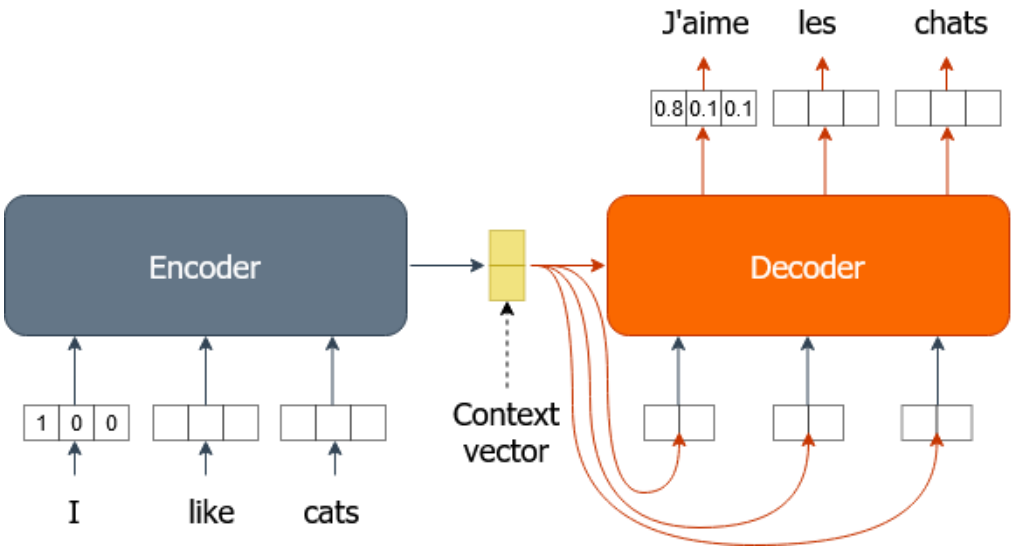

Figura 12: Esempio di modello *Sequence to Sequence* per la traduzione di un testo.

3.3.3 *L'Encoder*

Il compito della rete neurale dell'*encoder* è leggere la sequenza di input $X = \{x_1, x_2, \ldots, x_T\}$ di lunghezza T del nostro modello Seq2Seq, dove $x_t \in V = \{1, \ldots, |V|\}$ è la rappresentazione di un singolo *token* proveniente dal vocabolario V, e successivamente generare un *context vector* C di dimensione fissa per tale sequenza. Per fare ciò, l'*encoder* utilizzerà una cella di rete neurale ricorrente, di solito una LSTM, per leggere i *token* di input uno alla volta. L'*hidden state* finale h_t della cella diventerà quindi il *context vector* C. Tuttavia, dato che è molto difficile comprimere una sequenza di lunghezza arbitraria in un singolo vettore di dimensione fissa, specialmente per compiti complessi come la traduzione, l'*encoder* sarà solitamente costituito da LSTM impilate fra loro, ovvero una serie di *layers* LSTM in cui gli output di ciascun *layer* sono la sequenza di input del *layer* successivo. L'*hidden state* h_t del *layer* LSTM finale verrà utilizzato come il *context vector* C.

Gli *encoder* Seq2Seq fanno spesso però qualcosa di "strano": elaborano la sequenza di input al contrario. Questo in realtà è un comportamento ragionato, dato che l'idea è fare in modo che l'ultimo *token* che vedrà l'*encoder* sia approssimativamente corrispondente al primo *token* che il modello restituisce. Questo infatti rende più facile al *decoder* generare i primi output, e questo permette di conseguenza ad esso di generare una sequenza di output il più corretta possibile. Nel contesto della traduzione, viene permesso alla rete di tradurre le prime parole dell'input non appena

le vede, ed una volta che le prime parole sono state tradotte correttamente, è molto più facile continuare a costruire una frase corretta che rifarlo da zero. In Figura 13 viene mostrato un esempio di come una rete *encoder* potrebbe essere strutturata. In questo esempio il modello viene utilizzato per tradurre la frase inglese "what is your name" (come ti chiami) in francese e si può notare come i *token* di input vengano letti dal modello al contrario. Inoltre la rete è rappresentata in versione espansa, ovvero ogni colonna rappresenta un diverso passo temporale e ogni riga rappresenta un singolo *layer* LSTM, dunque le frecce orizzontali corrispondono agli *hidden states* passati in input alla stessa cella LSTM ai passi temporali successivi e quelle verticali rappresentato invece gli input o gli output delle celle LSTM [20, 25].

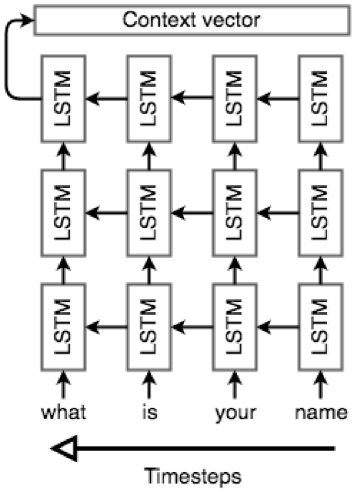

Figura 13: Esempio di rete *Encoder* formata da celle LSTM.

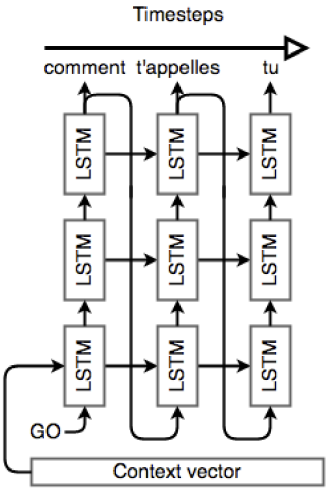

Figura 14: Esempio di rete *Decoder* formata da celle LSTM.

3.3.4 *Il Decoder*

Anche il *decoder* è in genere una rete LSTM, ma il suo utilizzo è un po' più complesso rispetto alla rete dell'*encoder*. In sostanza, viene usato come modello linguistico "consapevole" dell'input e delle parole che sono state generate fino ad un dato momento. A tal fine, anche in questo caso vengono usate varie LSTM impilate fra loro, in maniera simile all'*encoder*, ma inizializzando l'*hidden state* del primo *layer* LSTM con il *context vector* C generato dall'*encoder*. Il *decoder* infine farà letteralmente uso del contesto della sequenza di input per generare una sequenza $Y' = \{y'_1, y'_2, \ldots, y'_L\}$ di output di lunghezza L.

Una volta che il *decoder* è impostato con il suo contesto, gli viene passato in input uno speciale *token* per indicare l'inizio della generazione dell'output. In letteratura, di solito è usato un token <EOS> (*end of sequence*) aggiunto alla fine dell'input, e ne viene inserito uno anche alla fine dell'output. Successivamente, vengono eseguiti tutti i *layer* LSTM, uno dopo l'altro, seguiti dall'applicazione della funzione *softmax* [22] sull'output del *layer* finale per generare il primo *token* di output. Infine, questo *token* viene passato al primo *layer* LSTM e viene poi ripetuta questa procedura di generazione, ottenendo così ad ogni passo un nuovo *token* di output. Con questa strategia facciamo sì che le varie reti LSTM agiscano come un modello linguistico. In Figura 14 viene mostrato un esempio di *decoder* per un modello Seq2Seq. In questo caso il *decoder* sta analizzando il *context vector* per tradurre la frase inglese "what is your name" nella sua versione in francese "comment t'appeles tu". Osservando la figura possiamo notare che il *token* speciale "GO" viene utilizzato all'inizio della generazione per comunicare al *decoder* di iniziare a produrre i *token* di output, e che la loro generazione è effettuata in avanti in modo opposto all'input che viene invece letto al contrario come visto nell'*encoder*. Inoltre con questa strategia l'input e l'output non devono essere per forza della stessa lunghezza.

Una volta ottenuta la sequenza di output, viene utilizzata la stessa strategia di apprendimento classica delle reti neurali. Viene definita una funzione di costo, in genere per problemi di traduzione viene applicata la funzione di *cross-entropy* sulla sequenza di previsione, mentre l'errore viene minimizzato con un algoritmo di discesa del gradiente e utilizzando la tecnica di *backpropagation*. Sia l'*encoder* che il *decoder* sono addestrati nello stesso momento, in modo che entrambi apprendano la stessa rappresentazione del *context vector*. In diverse applicazioni, il *decoder* può anche trarre vantaggio da più informazioni come l'uso dei vettori di *self-attention* per generare risultati migliori.

Per addestrare il modello Seq2Seq è possibile usare anche strategie diverse. Uno degli approcci di addestramento più diffusi per questi modelli è chiamato *Teacher Forcing*, che consiste nell'utilizzare direttamente i *token* corretti della soluzione, ripresi dal dataset di addestramento, come input alle reti LSTM del *decoder*, anziché usare come input la previsione di output del *decoder* presa da un passo temporale precedente. Questo approccio garantisce una convergenza di addestramento più rapida ed evita che il modello accumuli troppi errori e che il suo apprendimento venga rallentato, sopratutto nelle fasi iniziali. Nella fase di inferenza invece, non avendo a disposizione le sequenze corrette a cui il modello

deve puntare, verrà utilizzata la strategia descritta precedentemente di alimentare il *decoder* con le sue previsioni precedenti. In questo approccio esiste dunque una discrepanza tra fase di addestramento e fase di inferenza [20, 25].

3.3.5 *Riepilogo ed un Piccolo Esempio*

Analizzando la descrizione precedente dei modelli Seq2Seq si può notare come non esista alcun collegamento tra le lunghezze delle sequenze di input e quelle di output, ovvero è possibile passare a questi modelli un input di qualsiasi lunghezza e allo stesso modo potranno generare un output di lunghezza arbitraria. Tuttavia, è noto in letteratura che i modelli Seq2Seq perdono di efficacia su sequenze di input estremamente lunghe, una conseguenza che ereditano dei limiti pratici delle reti LSTM.

Per ricapitolare il funzionamento di queste architetture, possiamo analizzare come si comporta un modello Seq2Seq per tradurre la frase inglese "How are you ?" nella sua corrispettiva frase in francese "Comment allez-vous ?". Per prima cosa, iniziamo con quattro vettori *one-hot* per l'input. Questi input, sopratutto nel caso di problemi di traduzione, possono anche essere incorporati in una rappresentazione vettoriale densa, utilizzando per esempio un *embedding layer* di input per ricavarsi i quattro *word embeddings* durante l'addestramento della rete. Successivamente, una rete formata da LSTM impilate, l'*encoder*, legge la sequenza di input al contrario, un *token* alla volta, e la codifica in un *context vector*. Questo vettore di contesto, dato che si basa sulla frase inglese "How are you ?", corrisponderò ad una rappresentazione spaziale vettoriale della nozione di chiedere a qualcuno come sta e viene utilizzato per inizializzare il primo *layer* di un altra rete di LSTM impilate, il *decoder*. Viene eseguito un passo di ogni *layer* di quest'ultima rete, dopodiché viene applicata la funzione *softmax* sull'output dell'ultimo *layer* e il risultato viene utilizzato per selezionare la prima parola di output. Questa parola viene reimmessa nella rete come input per il *decoder*, e il resto della frase "Comment allez-vous ?" viene decodificata seguendo lo stesso procedimento. Durante la *backpropagation*, i pesi della rete LSTM dell'*encoder* vengono aggiornati in modo che apprendano una migliore rappresentazione dello spazio vettoriale per le frasi, mentre i pesi della rete LSTM del *decoder* vengono addestrati per consentirgli di generare frasi grammaticalmente corrette e che si basano sul *context vector* [25]. In Figura 15 viene mostrato il modello Seq2Seq dell'esempio appena descritto.

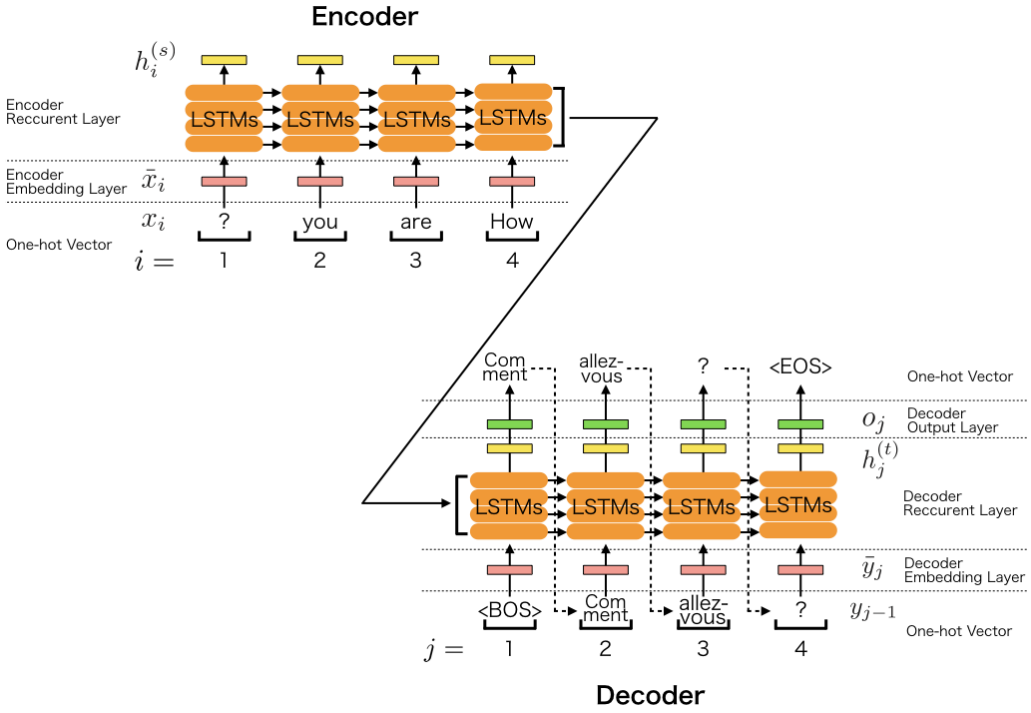

Figura 15: Esempio di modello Seq2Seq con reti LSTM.

3.3.6 *Migliorare la Fase di Inferenza: Beam Search*

Nelle maggior parte delle applicazioni dell'NLP, è possibile cercare di migliorare l'output generato dal *decoder* usando degli approcci che derivano dalla traduzione automatica statistica. Consideriamo un modello che calcoli la probabilità $\mathbb{P}(\bar{s}|s)$ di una traduzione $\bar{s}$ data la frase originale s. Vogliamo dunque scegliere la traduzione $\bar{s}*$ che ha la probabilità più alta, in altre parole, vogliamo $\bar{s}* = \arg\max_{\bar{s}}(\mathbb{P}(\bar{s}|s))$. Poiché lo spazio di ricerca può essere enorme quando cerchiamo di risolvere ad esempio un problema di traduzione su un testo molto lungo, dato che il numero totale di ipotesi che dobbiamo verificare risulta pari a $|V|^n$, dove $|V|$ rappresenta la dimensione del vocabolario del linguaggio ed n il numero di *token* nel testo da tradurre, non possiamo trovare la soluzione esatta, perché cercare di ottenerla diventerebbe impraticabile dal punto di vista computazionale, sopratutto se viene utilizzato un modello Seq2Seq. Pertanto, dobbiamo trovare una soluzione approssimata per cercare di ridurre le dimensioni di ricerca. Nei modelli Seq2Seq i *decoder* possono dunque utilizzare varie strategie per generare gli output.

Nel caso di un modello Seq2Seq standard, come descritto nelle sezioni precedenti, viene adoperato un approccio avido, chiamato *greedy search*,

dove ad ogni passo temporale, viene scelto il *token* con la probabilità più alta, generata dalla funzione *softmax*. Questa tecnica risulta sicuramente efficiente e naturale, tuttavia permette di esplorare solo una piccola parte dello spazio di ricerca. Infatti, nel momento in cui viene commesso un errore di traduzione ad un particolare passo temporale, il resto della frase che verrà generato successivamente potrebbe essere pesantemente influenzato da questo sbaglio, dato che la scelta del *token* migliore fatta al passo corrente non porterà necessariamente in seguito alla generazione della migliore sequenza possibile.

Un metodo alternativo che possiamo utilizzare è la tecnica di *beam search*. Invece di scegliere ad ogni passo un solo *token* con la probabilità più alta, ovvero l'output migliore in quel momento, e generando infine una singola sequenza di output, nel caso della *beam search* vengono selezionati ad ogni passo temporale K sequenze di output con la probabilità più alta, solo queste sequenze verranno poi continuate al passo successivo, generando quindi K *tokens* per ognuna di esse, e scegliendo successivamente sempre le K sequenze di output con la probabilità più alta. Infine seguendo questo procedimento verranno dunque generati K percorsi diversi per la sequenza di output e dunque K sequenze di output totali. Aumentando il numero di candidati K, otterremo una maggior precisione e troveremo delle soluzioni asintoticamente esatte. Tuttavia, il miglioramento non è monotono e dovremo dunque impostare un valore di K che faccia ottenere sia prestazioni ragionevoli che una giusta efficienza computazionale. Per questi motivi, la *beam search* è una della tecniche più utilizzate dell'NLP [20, 25].

4

IL MODELLO TRANSFORMER

Le reti neurali ricorrenti, ed in particolare le reti LSTM, negli ultimi anni si sono saldamente affermate come ottimi strumenti per i modelli *sequence to sequence* e su problemi come la modellazione del linguaggio e la traduzione automatica. Ancora oggi però vengono fatti numerosi sforzi per continuare a spingere i confini dei modelli linguistici ricorrenti e delle architetture *encoder-decoder*.

I modelli ricorrenti tipicamente portano avanti la loro computazione scorrendo le posizioni dei *tokens* presenti nelle sequenze di input ed output che analizzano. Ad ogni loro passo temporale elaborano un *token* di tali sequenze, generando una serie di *hidden states* h_t in funzione del precedente *hidden state* h_{t-1} e dell'input in posizione t. Questa natura intrinsecamente sequenziale preclude la possibilità di parallelizzare la computazione tra i dati di addestramento, e diventa sempre più problematica all'aumentare della lunghezza delle sequenze, poiché i vincoli di memoria limitano il *batching* tra i dati, ovvero il numero di campioni da far elaborare alla rete prima di aggiornare i suoi pesi e parametri. In passato sono stati raggiunti dei miglioramenti significativi nell'efficienza computazionale attraverso alcuni trucchi di fattorizzazione o l'uso del calcolo condizionale che hanno portato a leggeri miglioramenti sulle prestazioni dei modelli, ma il vincolo sull'elaborazione sequenziale è rimasto.

Gli *attention mechanisms* sono diventati parte integrante dei modelli Seq2Seq in vari ambiti, consentendo la modellazione delle dipendenze tra i *tokens* all'interno delle sequenze in modo indipendente rispetto alla loro distanza, sia per sequenze di input che di output. Precedentemente in quasi tutti i casi, tuttavia, tali *attention mechanisms* venivano utilizzati solo in combinazione con una rete neurale ricorrente.

Il modello *Transformer*, presentato per la prima volta in *Attention Is All You Need* (2017) [3] da un gruppo di ricercatori di *Google*, è stata una delle prime architetture ad evitare la ricorrenza delle reti precedenti ed

a basarsi interamente su un *attention mechanisms*, la *self-attention*, per ricavare le dipendenze globali tra input e output, scartando dunque l'uso di reti convoluzionali o di reti neurali ricorrenti impilate fra loro. Il *Transformer* consente una parallelizzazione significativamente maggiore e negli ultimi anni è diventato il nuovo stato dell'arte soprattutto per problemi legati alla traduzione [3].

In questo capitolo analizzeremo dunque come funziona il modello *Transformer*, descrivendone la sua architettura e come sfrutta il meccanismo di *self-attention* per ottenere dei risultati estremamente competitivi.

4.1 ARCHITETTURA DEL MODELLO

La maggior parte dei modelli Seq2Seq più performanti hanno un'architettura *encoder-decoder*. Queste reti neurali usano l'*encoder* per mappare una sequenza di *tokens* di input $x = (x_1, \ldots, x_n)$ in una sequenza di rappresentazioni continue $z = (z_1, \ldots, z_n)$, il *context vector*. Dato z, il *decoder* genera quindi una sequenza di output $y = (y_1, \ldots, y_m)$ di *tokens*, producendo un elemento alla volta. Ad ogni passo questi modelli sono auto-regressivi, ovvero, consumano i *tokens* generati in precedenza come input aggiuntivo durante la generazione del *token* successivo.

Anche il *Transformer* in generale segue questo tipo di architettura, ma utilizza in più una serie di *layers* di *self-attention* impilati fra loro per mettere in relazione diverse posizioni o *tokens* di una singola sequenza e adopera nella sua rete *layers* completamente connessi, sia per l'*encoder* che per il *decoder*. In Figura 16 viene mostrata l'architettura generale del *Transformer*. In particolare possiamo osservare come anch'esso sia formato da un *encoder*, raffigurato dal blocco grigio a sinistra, ed un *decoder*, raffigurato dal blocco grigio a destra [3].

4.1.1 *Input ed Output Embedding*

Poiché il testo scritto non può essere utilizzato direttamente come input per una qualsiasi rete neurale, dato che esse elaborano e comprendono solo matrici e numeri, è necessario trasformare ogni *token* o posizione della sequenza di input in una rappresentazione numerica. Ad ogni *token* di input viene dunque associato un numero intero, che viene in genere preso dall'indice numerico associato ad ogni parola o simbolo del vocabolario del testo che deve essere elaborato. Ogni sequenza di input quindi, prima di venire data in pasto alla rete, verrà trasformata in una

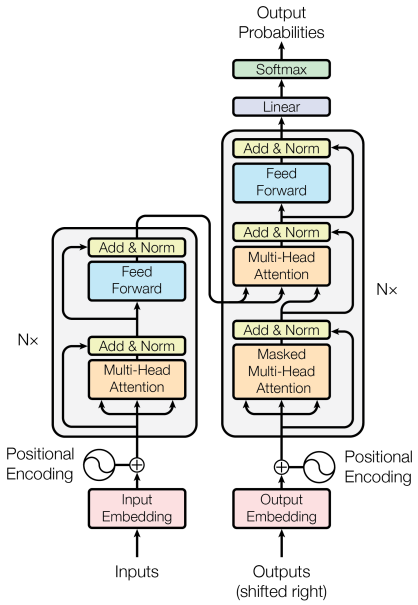

Figura 16: Architettura del modello *Transformer*.

sequenza di numeri interi (o indici).

Successivamente la sequenza viene passata all'*embedding layer* del *Transformer*, che durante l'addestramento della rete avrà il compito di ricavare una rappresentazione vettoriale densa di ogni *token* o posizione di tale sequenza, producendo dei veri e propri *word embeddings* come discusso nella Sezione 3.2.6. In particolare, analogamente anche ad altri modelli Seq2Seq, gli *embeddings* appresi vengono usati per convertire i *token* di input in vettori di dimensione pari a $d_{model} = 512$. Inizialmente questi vettori verranno inizializzati con numeri casuali, mentre in seguito durante la fase di addestramento verranno aggiornati costantemente per aiutare e migliorare il modello ad eseguire il compito a cui punta.

Lo stesso procedimento viene eseguito anche per l'input che sarà passato al *decoder* della rete, ovvero, la sequenza di output generata fino ad un dato momento. Come nel caso dei modelli Seq2Seq classici infatti, i *tokens* di output restituiti dal *decoder* ai passi precedenti vengono passati in input ad esso per permettere al *decoder* di predire con più accuratezza i *tokens* di output successivi. Ogni posizione di questa sequenza di input viene però prima spostata di una posizione verso destra.

Questa accortezza viene eseguita proprio perché il *decoder* viene addestrato in modo tale da essere in grado di prevedere il *token* successivo della sequenza dati i precedenti *tokens* generati. Se tale spostamento non viene effettuato infatti, quando la rete tenta di generare il primo *token*

di output, avrà grandi difficoltà a farlo dato l'assenza di *token* generati precedentemente. Pertanto, ogni posizione della sequenza di output viene spostata verso destra e al suo inizio viene inserito un *token* speciale "BOS" (*Beginning of Sentence*). A questo punto, quando la rete dovrà prevedere il primo *token*, il simbolo BOS diventerà il *token* precedente della sequenza di output [3, 28].

In Figura 17 viene mostrato a sinistra l'input dell'*encoder*, mentre a destra quello del *decoder*.

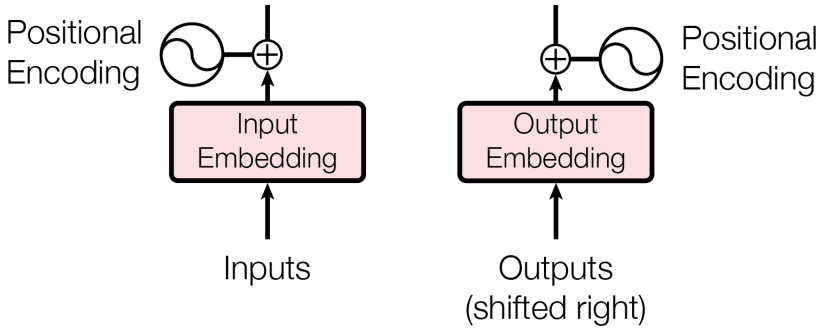

Figura 17: Input del *Transformer*, *embedding layers* e *positional encoding*.

4.1.2 *Positional Encoding*

Poiché il *Transformer* non contiene ricorrenze o convoluzioni, non conosce l'ordine dei *tokens* di input che elabora. I modelli *encoder-decoder* basati su reti ricorrenti LSTM ad esempio, prendono come input, in modo sequenziale ed a ogni passo, un solo *word embedding* di un particolare *token*. Sfruttando la loro natura ricorrente e la loro memoria conoscono perfettamente l'ordine di ciascuno *token* nella sequenza di input, ma ottengono questa informazione al costo di una esecuzione molto lenta. Il *Transformer* invece, sfruttando anche la sua facile parallelizzazione, prende in input tali *tokens* tutti insieme. Questo consente al modello di essere più veloce nell'elaborazione delle sequenze ma gli fa perdere completamente l'informazione sull'ordine dei *tokens* nella sequenza, privandolo dunque di dati utili per la generazione delle corrette sequenze di output. Affinché il modello possa apprendere ed utilizzare in modo esplicito le posizioni di ciascuno *tokens* in una sequenza, queste informazioni devono essere iniettate nelle rappresentazioni che il modello ha di essi, ovvero i loro *word embeddings* ricavati dagli *embedding layers*. In particolare, nelle loro rappresentazioni dovranno essere inserite informazioni sulla loro posizione relativa o assoluta nella sequenza. A tal fine, nel *Transformer*

viene inserito un *layer* di *positional encoding* che aggiunge queste informazioni agli *word embeddings* degli input, sia in corrispondenza della parte inferiore dell'*encoder* che del *decoder*, come mostrato in Figura 17. I *positional encoding* hanno la stessa dimensione d_{model} degli *embeddings* di input, in modo tale che questi due vettori possano essere semplicemente sommati.

In generale ci sono molte scelte di codifiche posizionali che possono essere impiegate, tramite una fase di addestramento oppure fisse. Nel caso del *Transformer*, per calcolare le varie dimensioni dei vettori di *positional encoding* associati ad ogni *token* sono state usate le funzioni seno e coseno di diverse frequenze:

$$PE_{(pos,\ 2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE_{(pos,\ 2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

dove *pos* indica una posizione all'interno della sequenza, ovvero un numero che va da 0 alla lunghezza di quest'ultima, che rappresenta dunque anche quale vettore di codifica posizionale stiamo calcolando, ed i rappresenta una dimensione di tale vettore (l'indice di una sua dimensione). In pratica, ogni dimensione del vettore di *positional encoding* corrisponde ad una sinusoide e le lunghezze d'onda formano una progressione geometrica da 2π a $10000 \cdot 2\pi$. Gli autori del *Transformer* hanno scelto proprio queste funzioni perché hanno ipotizzato che avrebbero permesso al modello di imparare facilmente ad ottenere le posizioni relative tra i *tokens* della sequenza, poiché per ogni offset fisso k, PE_{pos+k} può essere rappresentato come una funzione lineare di PE_{pos} [3, 28].

4.1.3 *L'Encoder*

L'*encoder* del *Transformer* è composto da una pila di $N = 6$ *layers* identici. Ogni *layer* è composto a sua volta da due *sub-layers*. Il primo è un *multi-head self-attention layer* che applica l'*attention mechanism* alla sequenza di input, che aiuta l'*encoder* a confrontare tra loro i vari *tokens* di quest'ultima per ricavarne le correlazioni, mentre il secondo è una rete *feed-forward* semplice e completamente connessa applicata indipendentemente ad ogni posizione della sequenza di input. Nelle sezioni successive analizzeremo questi due *sub-layers* più nel dettaglio.

Vengono utilizzate inoltre due connessioni residue attorno a ciascuno dei due *sub-layers*, che sostanzialmente permettono all'*encoder* di sommare l'output di un *sub-layer* con il suo input facilitando l'addestramento del *Transformer*, seguite da un *layer normalization*, che invece normalizza la rappresentazione vettoriale di ciascun *token* in *batch*, in sostanza normalizzando ognuna di esse con media zero e varianza unitaria, per meglio controllare il "flusso" di dati verso il *layer* successivo. La *layer normalization* migliora inoltre la stabilità della convergenza e talvolta anche la qualità dell'output del modello.

In pratica, l'output di ogni *sub-layers* corrisponde a $\text{LayerNorm}(x + \text{Sub-Layer}(x))$, dove $\text{Sub-Layer}(x)$ è la funzione implementata dal *sub-layer*. Inoltre, per facilitare le connessioni residue, tutti i *sub-layers* nel modello, così come gli *embedding layers*, producono un output di dimensione $d_{\text{model}} = 512$. In generale dunque, ogni *multi-head self-attention layer* prende in ingresso la codifica della sequenza di input restituita dall'*encoder* precedente nella pila, ad eccezione del primo *encoder* che prende in ingresso gli *input embeddings*, e stima le correlazioni reciproche tra i *tokens* per aggiornare a sua volta la codifica di tale sequenza. Ogni rete neurale *feed-forward* elabora ulteriormente questa codifica per ogni *token* e successivamente passa quest'ultima come input all'*encoder* consecutivo nella pila. Infine, l'ultimo *encoder* nella pila genererà la codifica finale della sequenza di input, un vero e proprio *context vector*, da passare poi al *decoder*. In Figura 18 viene mostrata la pila degli *encoder* con i loro relativi *sub-layers*.

Come vedremo meglio in seguito, il blocco dell'*encoder* in realtà esegue solo un mucchio di moltiplicazioni tra matrici che sono seguite da trasformazioni applicate ad ogni posizione della sequenza di input. Queste operazioni rendono il *Transformer* estremamente efficiente, in quanto la computazione può essere facilmente parallelizzata. Impilando queste trasformazioni l'una sull'altra viene creata una rete molto potente [3, 28].

4.1.4 *Il Decoder*

Anche il *decoder* è composto da una pila di $N = 6$ *layers* identici. Oltre ai due *sub-layers* presenti anche in ogni *encoder* in pila, il *decoder* inserisce un terzo *sub-layer*, che esegue un *multi-head self-attention* sull'output della pila di *encoder*, ovvero la codifica della sequenza di input. Analogamente all'*encoder*, vengono utilizzate delle connessioni residue attorno a ciascuno dei *sub-layers*, seguite da un *layer normalization*. In Figura 19 viene mostrata la pila dei *decoder* con i loro relativi *sub-layers*.

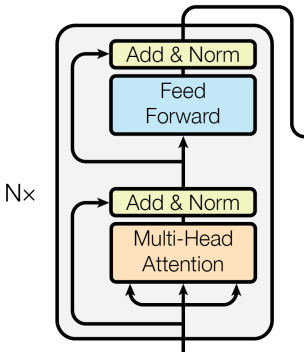

Figura 18: Pila degli *encoder* del *Transformer*.

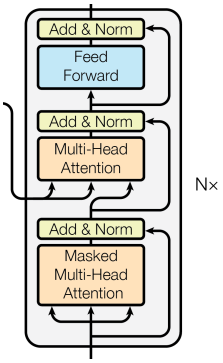

Figura 19: Pila dei *decoder* del *Transformer*.

Il primo *sub-layer* per la *self-attention* in ogni *decoder* in pila viene inoltre leggermente modificato per evitare che in fase di addestramento della rete il *decoder* prenda in input i *tokens* futuri della sequenza di output, che il modello potrebbe ottenere dal dataset su cui si sta addestrando. Infatti, mentre l'*encoder* riceve tutti i *tokens* contemporaneamente e può osservare ognuno di essi nella sequenza di input, il *decoder* genera un *token* alla volta e durante la fase di inferenza non conosce quali *tokens* verranno generati in futuro, mentre in fase di addestramento si vuole evitare che il *decoder* sbirci sui futuri *tokens* che dovrebbe generare in seguito. Inoltre, se al *decoder* viene passata l'intera sequenza di output dal dataset tutta in una volta, il modello tenderà a ripete semplicemente la sequenza non imparando nulla. Questo mascheramento dunque, combinato al fatto che gli *output embeddings* sono sfalsati di una posizione in avanti, assicura che le previsioni per la posizione i della sequenza di output possono dipendere solo dalle previsioni note per le posizioni inferiori a i.

Il *decoder* funziona pertanto in modo simile all'*encoder*, ma in ogni suo componente della pila usa un *multi-head self-attention layer* che ricava informazioni rilevanti dalla codifica generata dalla pila di *encoder*. In modo simile al primo *encoder*, il primo *decoder* accetta gli *output embeddings* come input. In generale invece, ogni *masked multi-head self-attention layer* prende in ingresso l'output del precedente *decoder* della pila, stima le correlazioni reciproche tra i soli *tokens* generati in output fino a quel momento e passa queste informazioni al *sub-layer* successivo. Ogni *multi-head self-attention layer* elabora ulteriormente questi dati insieme all'output ricevuto dall'*encoder*, una sorta di *context vector*, ed il risultato viene passato alla rete *feed-forward* per la lavorazione finale. Infine, questo risultato finale viene passato al *decoder* successivo nella pila. L'ultimo

decoder della pila passerà poi l'output finale agli ultimi due *layers* del *Transformer* per il calcolo delle probabilità finali per la generazione dei *tokens* di output, ovvero, un *linear layer* e un *softmax layer*, di cui parleremo meglio nelle sezioni successive [3, 28].

4.1.5 *Self-Attention*

La *self-attention* è una delle componenti chiave del modello. La differenza tra *attention* e *self-attention* è che quest'ultima opera tra rappresentazioni della stessa natura, ad esempio, tutte le parole in un particolare testo. La *self-attention* è dunque la parte del modello in cui i *tokens* interagiscono tra loro. Ogni *token* "osserva" altri *tokens* nella sequenza con un *attention mechanism*, raccoglie il contesto e aggiorna la precedente rappresentazione di se stesso.

Formalmente, questa intuizione viene implementata con una *query-key-value attention*. Ogni *token* di input, applicando la *self-attention*, riceve tre rappresentazioni corrispondenti ai ruoli che può svolgere:

- **query**: chiedere informazioni;
- **key**: comunicare che ha alcune informazioni;
- **value**: consegnare le informazioni.

La *query* viene utilizzata quando un *token* osserva gli altri, ovvero nel momento in cui sta cercando le informazioni per comprendersi meglio. La *key* viene utilizzata per rispondere alla richiesta di una *query*, ed è dunque usata per calcolare i pesi dell'*attention*. Infine, il *value* viene utilizzato per calcolare l'output dell'*attention* fornendo informazioni ai *tokens* che "comunicano" di averne bisogno, ossia ai *tokens* che hanno assegnato dei grandi pesi al *token* che stiamo considerando.

In generale quindi la *self-attention* usata dal modello può essere descritta come una funzione che mappa una *query* e un insieme di coppie *key-value* ad un output, dove le *queries*, le *keys*, i *values* e gli output sono tutti dei vettori. L'output viene calcolato come somma ponderata dei *values*, dove il peso assegnato a ciascun *value* è calcolato da una funzione di compatibilità applicata alla *query* e alla *key* corrispondente [3, 28].

Scaled Dot-Product Attention

La tipologia di *self-attention* utilizzata dal *Transformer* viene chiamata in particolare *Scaled Dot-Product Attention*, mostrata in Figura 20. L'input è

costituito da *queries* e *keys* di dimensione d_k e *values* di dimensione d_v. Inizialmente vengono calcolati i prodotti scalari delle *queries* con tutte le *keys*, divisi ciascuno per $\sqrt{d_k}$, e successivamente il risultato, viene applicato ad una funzione *softmax* per ottenere i pesi dei *values*.

In pratica, la funzione di *attention* viene calcolata su un insieme di *queries* contemporaneamente, raggruppate in una matrice Q. Le *keys* e i *values* sono anch'essi impacchettati insieme rispettivamente nelle matrici K e V [3]. La matrice di output viene dunque calcolata come segue:

$$\textbf{Attention}(Q,\ K,\ V) = \textbf{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

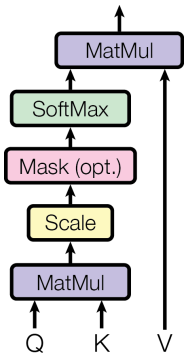

Figura 20: *Scaled Dot-Product Attention* del *Transformer*.

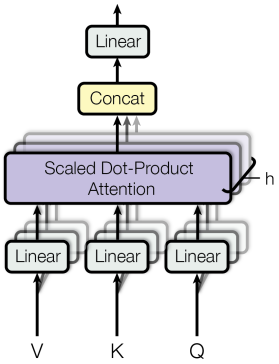

Figura 21: *Multi-Head Attention* del *Transformer*.

Multi-Head Attention

Di solito, capire il ruolo di una parola in una frase richiede la comprensione di come essa è correlata alle diverse parti della frase. Questo è importante non solo nell'elaborazione della frase sorgente, ma anche nella generazione dell'output che vogliamo ottenere. Ad esempio, in alcune lingue, i soggetti definiscono la flessione del verbo, come l'accordo di genere, oppure le coniugazioni dei verbi definiscono i loro soggetti. Ogni parola dunque può fa parte di molte relazioni.

Pertanto, il modello deve avere la possibilità di concentrarsi su cose diverse, ed è proprio questa la motivazione che sta dietro all'uso della *Multi-Head Attention*. Invece di avere un singolo *attention mechanism*, la *Multi-Head Attention* sfrutta diverse "teste" che funzionano in modo indipendente. Invece di eseguire una singola funzione di *attention* con *keys*, *values* e *queries* di d_{model} dimensioni, gli autori del *Transformer* hanno

trovato utile proiettare linearmente le *queries*, le *keys* e i *values* h volte con diverse proiezioni lineari, ricavate durante la fase di addestramento della rete, alle dimensioni d_k, d_k e d_v, rispettivamente. Su ciascuna di queste versioni proiettate delle *queries*, delle *keys* e dei *values* viene dunque poi eseguita la funzione di *attention* in parallelo, ottenendo i valori di output con d_v dimensioni. Quest'ultimi verranno poi concatenati e ancora una volta proiettati, risultando nei valori finali, come mostrato in Figura 21. La *Multi-Head Attention* consente dunque al modello di prestare congiuntamente attenzione alle informazioni provenienti da diverse rappresentazioni di sottospazi in posizioni differenti. In pratica, viene calcolata la seguente matrice:

$$\begin{aligned}\textbf{MultiHead}(Q,\ K,\ V) &= \textbf{Concat}(\text{head}_1,\ \ldots,\ \text{head}_h)W^O \\ \text{dove head}_i &= \textbf{Attention}(QW_i^Q,\ KW_i^k,\ VW_i^V)\end{aligned}$$

dove le proiezioni sono le matrici dei pesi $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ e $W^O \in \mathbb{R}^{hd_v \times d_{model}}$, ricavate in fase di addestramento della rete [3, 28].

Applicazione dell'Attention nel Transformer

Il *Transformer* utilizza la *Multi-Head Attention* in tre modi diversi.

Nei *layers* di *attention* centrali presenti in ogni *decoder* in pila, le *queries* provengono dal precedente *layer* del *decoder*, mentre le *keys* ed i *values* provengono dall'output dell'ultimo *encoder* in pila, come mostrato in Figura 19. Questo permette ad ogni posizione osservata dal *decoder* di analizzare tutte le altre posizioni nella sequenza di input e di concentrarsi maggiormente in fase di addestramento sulle parti più appropriate.

Nei *layers* di *self-attention* degli *encoder* in pila invece, tutte le *keys*, i *values* e le *queries* provengono dall'output del precedente *encoder* in pila, ad eccezione del primo *encoder* dove il *Multi-Head Attention layer* ricava i suoi ingressi dagli *input embeddings*, come mostrato in Figura 18. Ogni posizione osservata dall'*encoder* può dunque analizzare tutte le posizioni della sequenza basandosi sul precedente *layer* dell'*encoder*.

In modo simile, i *Masked Multi-Head Attention layers* nei *decoder* in pila consentono a ciascuna posizione della sequenza generata dal *decoder* di essere confrontata solo con tutte le altre posizioni precedenti generate, fino a se stessa inclusa. Un vero e proprio mascheramento che viene impiegato per prevenire nel *decoder* un flusso di informazioni verso sinistra per preservare la proprietà auto-regressiva del modello. Questo viene implementato all'interno della *Scaled Dot-Product Attention* impostando

a $-\infty$ tutti i valori nell'input del *softmax layer* che corrispondono a posizioni illegali. In Figura 20 viene mostrato anche questo mascheramento opzionale [3].

4.1.6 *Position-wise Feed-Forward Networks*

Oltre ai *sub-layers* di *attention*, ciascuno dei vari *encoder* e *decoder* in pila contiene una rete *feed-forward* completamente connessa, che viene applicata a ciascuna posizione della sequenza separatamente e in modo identico, ovvero la stessa rete neurale viene applicata ad ogni singola rappresentazione vettoriale di ogni *token*. Questa rete consiste in due trasformazioni lineari (due *linear layer*) con una funzione di attivazione ReLU in mezzo, ovvero:

$$\mathrm{FFN}(x) = \max(0,\ xW_1 + b_1)W_2 + b_2.$$

Sebbene le trasformazioni lineari applicate siano le stesse sia negli *encoder* che nei *decoder*, esse utilizzano parametri diversi che cambiano per ogni *layer*. La dimensionalità dell' input e dell'output di questa rete è pari a $d_{\text{model}} = 512$ e il layer interno ha una dimensionalità pari a $d_{ff} = 2048$. In Figura 22 viene mostrato un esempio di questa rete [3].

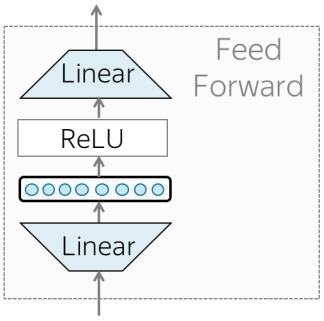

Figura 22: *Position-wise Feed-Forward Network* del *Transformer*.

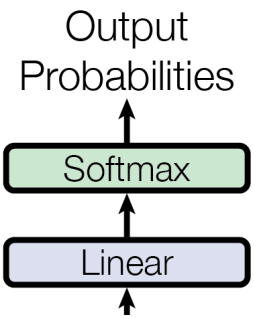

Figura 23: Layers di output del *Transformer*.

4.1.7 *Linear e Softmax Layer per l'Output*

Per convertire l'output della pila dei *decoder* in una probabilità per il prossimo *token* da generare viene usato un *linear layer* e un *softmax layer*.

Il *linear layer* è una semplice rete neurale completamente connessa che tramite una trasformazione lineare proietta il vettore prodotto dalla

pila dei *decoder*, ottenuto appiattendo su una singola riga la matrice di output del *decoder*, in un vettore molto più grande chiamato *logits vector* di dimensione pari alla grandezza del vocabolario $|V|$ del testo di input. Questo *logits vector* conterrà in ogni cella un punteggio corrispondente ad un unico *token* del vocabolario.

Il *softmax layer* trasformerà poi questi punteggi in probabilità, tutte positive e la cui somma risulti pari ad 1. Infine, viene scelta la cella con la probabilità più alta e il *token* associato ad essa viene prodotto come output per questa fase temporale. In Figura 23 vengono mostrati questi due ultimi *layers* del *Transformer*.

Nel *Transformer* inoltre, tra i due *embedding layers* e il *linear layer* precedente al *softmax layer* viene condivisa la stessa matrice dei pesi. Negli *embedding layers*, in particolare, questi pesi vengono moltiplicati per un fattore pari a $\sqrt{d_{model}}$ [3].

4.2 OPTIMIZER E REGOLARIZZAZIONE

Per aggiornare iterativamente i pesi della rete in base ai dati di addestramento il *Transformer* utilizza l'algoritmo ADAM [29], un algoritmo di ottimizzazione (*optimizer*) creato come estensione della discesa stocastica del gradiente, e che recentemente ha visto un'adozione sempre più ampia per le applicazioni di *deep learning* nella visione artificiale e nell'NLP. A differenza della discesa stocastica del gradiente che mantiene un unico *learning rate* per tutti gli aggiornamenti ai pesi che rimane fisso durante tutto l'addestramento, l'algoritmo ADAM può usare un *learning rate* diverso per ogni peso della rete che viene adattato separatamente man mano che l'apprendimento del modello avanza. In generale, l'algoritmo ADAM risulta essere molto efficiente dal punto di vista computazionale e richiede poca memoria, inoltre è particolarmente adatto per problemi di grandi dimensioni in termini di dati o parametri e per problemi con gradienti molto rumorosi o sparsi.

Nel *Transformer* viene usato anche un meccanismo di regolarizzazione durante l'addestramento della rete per evitare l'*overfitting* del modello. In particolare, viene applicato il *dropout* all'output di ogni *sub-layers*, prima che venga sommato all'input dello stesso *sub-layer* e normalizzato. Inoltre, il *dropout* viene applicato anche alle somme tra gli *embeddings* e i *positional encoding* sia per l'input alla pila degli *encoder* che dei *decoder*. Per il modello *Transformer*, viene usato in genere un tasso di *dropout* pari a $P_{drop} = 0.1$ [3].

5

VERIFICHE SPERIMENTALI

In questo capitolo, andremo a mostrare diversi test effettuati su alcuni modelli *Transformer* addestrati sui dataset ottenuti dai generatori presentati nella Sezione 2.6, per la risoluzione di integrali ed equazioni differenziali del primo e del secondo ordine. Valuteremo le loro prestazioni in termini di accuratezza su questi tipi di problemi e riporteremo anche un confronto effettuato con alcuni dei più utilizzati framework di matematica simbolica per la risoluzione di questi compiti. In particolare, inizialmente, descriveremo meglio che tipo di dataset sono stati utilizzati, il tipo di impostazioni e configurazioni usate per i vari modelli *Transformer*, il tipo di hardware utilizzato per effettuare i test e il metodo di valutazione usato per i modelli esaminati. Successivamente analizzeremo le accuratezze e la generalizzazione dei modelli sui due problemi matematici presi in considerazione, discuteremo di come i modelli si comportano all'aumentare dei dati di addestramento forniti, che tipo di soluzioni questi modelli sono in grado di ottenere ed approfondiremo alcuni esempi pratici di come è possibile sfruttare i vari modelli per ottenere soluzioni valide per i quesiti richiesti, fornendo e commentando anche il codice utilizzato. Infine riporteremo un confronto effettuato dai due ricercatori *Lample* e *Charton* tra i modelli *Transformer* proposti e i software di calcolo simbolico *Mathematica*, *Matlab* e *Maple* per mostrare come i modelli *Transformer* riescano ad ottenere spesso prestazioni sensibilmente superiori ai software più comuni e utilizzati per la risoluzione di integrali ed equazioni differenziali.

5.1 DATASET UTILIZZATI ED IMPLEMENTAZIONE DEL MODELLO

L'implementazione del modello *Transformer* per la risoluzione di integrali ed equazioni differenziali e i dataset necessari per i test di questo capitolo sono stati ripresi dal quelli forniti dagli autori nel loro lavoro di ricerca [1]. In particolare, possono essere trovati nel proget-

to GitHub originale "*SymbolicMathematics*" dei due ricercatori: `https://github.com/facebookresearch/SymbolicMathematics`.

Nel loro progetto, *Lample* e *Charton* forniscono un implementazione del *Transformer*, interamente sviluppata nel linguaggio *Python*, e basata sulla libreria *PyTorch*, un framework *open source* molto utilizzato per il *machine learning* e il *deep learning* sviluppato dal *Facebook's AI Research Lab*. Come framework di matematica simbolica è stata invece scelta la libreria Python *SymPy*, utilizzata sia in fase di generazione dei dataset sia in fase di valutazione dei modelli. Nel codice fornito gli autori permettono di generare nuovi dataset seguendo le strategie discusse nella Sezione 2.6, la creazione e l'addestramento di nuovi modelli *Transformer* per la risoluzione dei problemi considerati, e la valutazione dell'accuratezza di un modello tramite *greedy search* o *beam search*. Inoltre, vengono messi a disposizione anche diversi dataset per ogni quesito considerato, divisi in *train*, *valid* e *test set*, e alcuni modelli già addestrati su diverse combinazioni di tali dataset.

Per tutte le attività considerate, i dataset generati dagli autori sono stati creati utilizzando i metodi presentati nella Sezione 2.6, scegliendo in particolare:

- espressioni con un massimo di $n = 15$ nodi interni e con una lunghezza massima pari a 512 *tokens*;
- valori per i nodi foglia compresi in $\{x\} \cup \{-5, \ldots, 5\} \setminus 0$;
- quattro operatori binari: $+$, $-$, $\times$, $/$;
- quindici operatori unari: exp, log, sqrt, sin, cos, tan, $\sin^{-1}$, $\cos^{-1}$, $\tan^{-1}$, sinh, cosh, tanh, $\sinh^{-1}$, $\cosh^{-1}$, $\tanh^{-1}$.

Dataset	Dimensione Train Set	Dimensione Validation/Test Set
Integrazioni - FWD	45 Milioni	10000
Integrazioni - BWD	88 Milioni	9000
Integrazioni - IBP	23 Milioni	8000
Equazioni Differenziali - ODE 1	65 Milioni	8000
Equazioni Differenziali - ODE 2	32 Milioni	9000

Tabella 2: Dimensioni dei dataset per integrazioni ed equazioni differenziali.

Le statistiche sui vari dataset generati dagli autori sono presentate nella Tabella 1 e come già discusso nella Sezione 2.6.5, possiamo osservare come ogni approccio genera delle espressioni di input e di output molto

diverse dal punto di vista della lunghezza. In particolare, per i nostri test sono stati utilizzati i dataset forniti nella pagina *GitHub* del progetto, le cui dimensioni vengono riportate nella Tabella 2.

5.2 CONFIGURAZIONE DEL MODELLO E HARDWARE UTILIZZATO

Per tutti i modelli *Transformer* utilizzati nei vari di test di questo capitolo sono state utilizzate 8 *attention heads* e 6 *layers* di *encoders* e *decoders* in pila. I modelli forniti dagli autori, e dunque già addestrati, utilizzano una dimensionalità pari $d_{model} = 1024$ mentre per i nuovi modelli creati per alcuni test come quelli relativi allo studio dei loro comportamenti al variare della dimensione dei dataset è stata utilizzata una dimensionalità pari $d_{model} = 512$. Come riportato dai due ricercatori *Lample* e *Charton* nel loro lavoro di ricerca [1] l'utilizzo di modelli più grandi non porta in genere a prestazioni migliori, e dunque per evitare un aumento eccessivo del costo computazionale non sono stati addestrati modelli con dimensionalità o con un numero di *layers* superiori. Inoltre, tutti i modelli sono stati addestrati tramite l'algoritmo di ottimizzazione Adam [29], con un *learning rate* pari a 10^{-4} e le espressioni con più di 512 *tokens* sono state rimosse.

Durante la fase di decodifica, è facile intuire come non ci siano vincoli che impediscono ai *decoder* di un modello *Transformer* di generare un'espressione matematica con una notazione prefissa non valida, ad esempio come $[+\ 5\ *\ 8]$. In tali casi, dato che le espressioni generate dai modelli risultano essere quasi sempre valide viene permesso comunque al modello di restituire l'espressione non valida e quando questo accade essa viene semplicemente considerata come una soluzione errata e viene dunque ignorata.

Durante la fase di inferenza, le soluzioni ipotizzate vengono generate tramite *beam search* e gli *scores* di tali ipotesi, delle *log-likelihood* che misurano quanto il modello è sicuro che siano soluzioni valide, vengono normalizzate rispetto alla lunghezza della sequenze. I test in generale sono stati eseguiti con *beam search* di dimensioni 1 (corrispondente ad una *greedy search*) e 10 [1].

Per generare gli enormi dataset richiesti nei test e per addestrare i vari modelli *Transformer*, nel loro studio originale i due ricercatori hanno sfruttato un serie di computer distribuiti equipaggiati con diverse GPU ciascuno, permettendo loro di elaborare fino a 256 equazioni per *batch*. Nel nostro caso, avendo a disposizione una sola macchina equipaggiata con una singola GPU, una GTX 1080 TI di NVIDIA, per riprodurre alcuni

risultati è stato necessario ridurre il numero di equazioni per *batch* ad un massimo di 32, ed in alcuni casi anche a 16, data la limitata memoria della GPU a nostra disposizione.

5.3 METODO DI VALUTAZIONE

Alla fine di ogni epoca di addestramento, i modelli vengono valutati sul *validation* e sul *test set* del dataset su cui si stanno allenando per misurare la loro capacità di prevedere le soluzioni sulle equazioni fornite. Nella traduzione automatica, le ipotesi generate dal modello sono in genere confrontate con i riferimenti a disposizione scritti da traduttori umani, tipicamente con metriche come il punteggio BLEU che misura la corrispondenza tra le ipotesi fatte dal modello ed i riferimenti. Valutare la qualità delle traduzioni è un problema molto difficile e diversi studi hanno dimostrato che un punteggio BLEU superiore non è necessariamente assegnato ad una prestazione migliore secondo la valutazione umana. Nel caso della matematica simbolica, tuttavia, possiamo facilmente verificare la correttezza dei modelli confrontando semplicemente le espressioni generate con le loro soluzioni di riferimento.

Ad esempio, per l'equazione differenziale $xy' - y + x = 0$ che ha come soluzione di riferimento $x \log(c/x)$, dove c è una costante, un modello può generare l'ipotesi $x \log(c) - x \log(x)$ ed utilizzando un framework di matematica simbolica come *SymPy*, possiamo verificare che queste due soluzioni sono uguali sebbene siano scritte in modo diverso.

Tuttavia, i modelli possono anche generare $xc - x \log(x)$ che risulta anch'essa una valida soluzione, dato che è equivalente alla precedente per una diversa scelta della costante c. In tal caso, sostituiamo y nell'equazione differenziale con la soluzione ipotizzata dal modello. Se $xy' - y + x = 0$, concludiamo che l'ipotesi è una soluzione valida. Nel caso del calcolo integrale, possiamo invece semplicemente differenziare l'ipotesi del modello e confrontarlo con la funzione da integrare, se risultano uguali l'ipotesi è una soluzione valida.

Poiché possiamo facilmente verificare la correttezza delle espressioni generate, vengono considerate tutte le ipotesi restituite dalla *beam search*, e non solo quella con lo *score* più alto. Viene verificata dunque la correttezza di ogni ipotesi e se una di esse è corretta allora viene approvato che il modello ha risolto con successo l'equazione di input. Di conseguenza, i risultati relativi ad una *beam search* di dimensione 10 indicano che almeno una delle 10 ipotesi era corretta [1].

5.4 ACCURATEZZA E GENERALIZZAZIONE DEI MODELLI

In questa sezione vengono riportati i risultati ottenuti sulla nostra configurazione sulle percentuali di accuratezza dei modelli addestrati e forniti dai due ricercatori, sia per l'integrazione che per le equazioni differenziali. Nel caso dell'integrazione, oltre ai modelli addestrati sui dataset della Tabella 2 sono stati utilizzati anche alcuni modelli allenati su diverse combinazioni dei tre dataset originali, ripresi sempre dalla pagina *GitHub* del progetto, per meglio verificare la generalizzazione dei tre rispettivi generatori. Ogni modello è stato testato con l'approccio *beam search*, ed in particolare i vari modelli per l'integrazione sono stati tutti valutati su ognuno dei tre dataset di integrazione originali (FWD, BWD e IBP). In tutti i casi sono stato usati i vari *test set* dei dataset come riferimento. I risultati ottenuti sono mostrati nella Tabelle 3 e 4, rispettivamente per le integrazioni e per le equazioni differenziali.

Per l'integrazione, possiamo osservare come i vari modelli addestrati sui corrispettivi dataset dei tre generatori raggiungono delle prestazioni prossime al 95-100% di accuratezza sia con *beam search* di dimensione 1 (*greedy search*) che con dimensione 10. I modelli per le equazioni differenziali invece, ottengono in generale delle accuratezze inferiori, sopratutto il modello con *greedy search* per quelle del secondo ordine. Utilizzando una *beam search* di dimensione 10 possiamo però osservare come i due modelli migliorano sensibilmente. I due ricercatori riportano inoltre che aumentando la dimensione della *beam search* a 50 si ottengono delle accuratezze ancora maggiori per tutti i modelli, in particolare per il modello per le equazioni differenziali del primo ordine che si avvicina alle prestazioni ottimali ottenute per le integrazioni. Questo possibile aumento delle accuratezze tramite l'uso di *beam search* più grandi è una conseguenza prevedibile proprio per il fatto che all'aumento delle dimensioni della *beam search* i vari modelli avranno l'opportunità di generare più ipotesi e dunque una maggiore possibilità di restituire la soluzione giusta.

Integrazioni	**Forward (FWD)**	**Backward (BWD)**	**Integrazione per Parti (IBP)**
Beam Search 1	95.6	98.4	97.7
Beam Search 10	97.2	99.5	99.4

Tabella 3: Percentuali di accuratezza dei modelli per l'integrazione.

Nella Tabella 5, vengono invece riportate le accuratezze ottenete sulla nostra configurazione sui dataset FWD, BWD e IBP utilizzando modelli di integrazione addestrati con diverse combinazioni dei tre dataset originali.

Equazioni Differenziali	ODE 1	ODE 2
Beam Search 1	89.5	75.0
Beam Search 10	96.8	85.5

Tabella 4: Percentuali di accuratezza dei modelli per equazioni differenziali.

Quando i modelli vengono testati sui dataset su cui sono stati in parte addestrati le accuratezze sono molto alte. Ad esempio i tre modelli FWD, FWD+BWD e FWD+BWD+IBP, ottengono un accuratezza superiore al 95% se testati sul dataset FWD, sia con *beam search* 1 che 10. Il modello addestrato su BWD invece, anche nel caso di una *beam search* di dimensione 10 raggiunge solo il 31.4% di accuratezza sul dataset FWD, e viceversa il modello addestrato su FWD raggiunge solo il 15,7% di accuratezza sul dataset BWD. Questo deriva dal fatto che i dataset ottenuti dai generatori FWD e BWD sono molto diversi, come si puo vedere anche dalla Tabella 1 che riassume le caratteristiche di tali generatori.

Modelli \ Dataset	Forward (FWD)		Backward (BWD)		Integrazione per Parti (IBP)	
	Beam 1	Beam 10	Beam 1	Beam 10	Beam 1	Beam 10
FWD	95.6	97.2	11.9	15.7	88.1	89.1
BWD	26.4	31.4	98.4	99.5	50.4	60.0
IBP	45.8	55.2	63.6	85.4	97.7	99.4
FWD + BWD	94.6	96.8	98.7	99.5	85.7	86.0
BWD + IBP	47.7	56.8	98.3	99.3	97.6	99.3
FWD + BWD + IBP	93.0	95.6	98.3	99.5	98.3	99.6

Tabella 5: Accuratezza dei modelli per l'integrazione su tutti i dataset.

In generale infatti, un modello addestrato su un dataset FWD apprende che l'integrazione tende ad allungare le espressioni risultanti delle soluzioni, una proprietà che non vale per i dati BWD, dove avviene l'esatto contrario. Un modello FWD riesce invece abbastanza bene a predire le soluzioni per una dataset IBP, ottenendo un accuratezza pari all'89.1%, questo perché i due generatori FWD e IBP creano dei dataset abbastanza simili, con espressioni di input corte e soluzioni lunghe. Aggiungere più varietà ai dati di addestramento può dunque migliorare i risultati di un modello. Ad esempio, l'aggiunta di dati ripresi dal dataset IBP al modello addestrato su BWD aumenta la precisione sul dataset FWD dal 31,4% al 56,8% e con ulteriori dati di addestramento presi dal dataset FWD il modello raggiunge il 95,6% di accuratezza. Infine, l'addestramento di un

modello su tutti i dataset consente ad esso di ottenere ottimi risultati su tutti i dataset originali, garantendo delle accuratezze superiori al 95%.

5.5 COMPORTAMENTO DEI MODELLI ALL'AUMENTO DEI DATI

Abbiamo voluto inoltre testare come alcuni modelli reagiscono all'aumentare dei dati di addestramento forniti, per verificare come aumentano le accuratezze sulle predizioni e in che modo invece gli errori diminuiscono, in particolare sui modelli BWD, ODE 1 e FWD+BWD+IBP. Tutti i test effettuati sono stati eseguiti addestrando nuovamente i vari modelli sui loro corrispettivi dataset ma fornendo loro solo un sottoinsieme di dati su cui allenarsi, incrementando questo carico ad ogni nuovo test. In generale, ogni tipo di modello è stato addestrato su 1000, 10000, 100000 e 1000000 di dati, ripresi dai corrispettivi dataset a cui i modelli fanno riferimento, e la valutazione è stata eseguita rispetto ai loro *test set* e *validation set*, anch'essi ridotti a soli 500 dati ma fissi per ogni test. In ognuno dei test, l'addestramento è stato portato avanti per 50 epoche utilizzando 32 equazioni per *batch* e un $d_{model} = 512$. Nelle Figure 24, 25 e 26 vengono mostrati i risultati ottenuti rispettivamente sui modelli addestrati con i dataset BWD, ODE 1 e FWD+BWD+IBP.

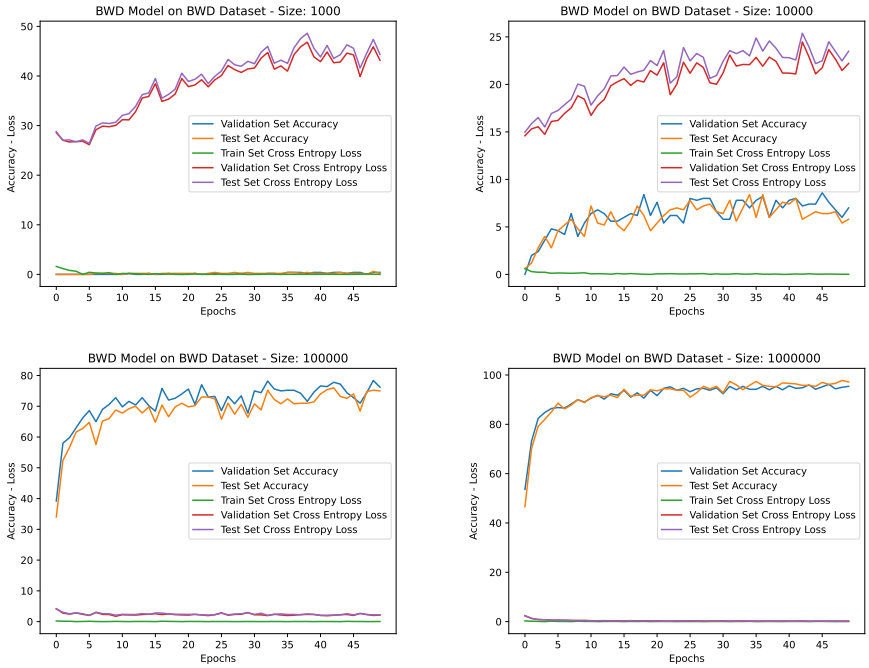

Figura 24: Prestazioni del modello BWD all'aumentare dei dati.

Osservando i grafici in Figura 24 per il modello BWD possiamo notare che nel caso si fornisca al modello solo 1000 dati quest'ultimo non riesce a predire correttamente quasi nessuna soluzione, le accuratezze misurate sul *test set* e sul *validation set* sono infatti prossime allo 0% per tutte le epoche di addestramento. Gli errori calcolati tramite la funzione di costo (una *cross-entropy*) sul *validation* e sul *test set* tendono in generale ad aumentare man mano che si prosegue con l'addestramento mentre l'errore sul *train set* tende invece a diminuire e rimane basso, confermando dunque che il modello è in *overfitting* e non riesce a generalizzare su dati mai visti prima. La situazione comincia a migliorare leggermente nel momento in cui al modello vengono forniti 10000 dati, in questo caso infatti le accuratezze sul *validation* e il *test set* iniziano a salire per poi stabilizzarsi intorno all'8%. Gli errori sui vari set invece mantengono lo stesso comportamento precedente ma si abbassano di un 50% circa. Fornendo al modello 100000 dati le accuratezze sui set si alzano parecchio raggiungendo e stabilizzandosi su buoni risultati intorno al 75% di precisione già dopo 20 epoche, mentre gli errori si mantengono sempre molto bassi fin dall'inizio su tutti i set. Infine se passiamo al modello da 1 milione di dati otteniamo finalmente delle accuratezze in linea con i test precedenti, intorno dunque al 95% di precisione, e mantenendo degli errori molto bassi sui vari set. Possiamo dunque affermare che un modello BWD per poter predire con un alta accuratezza le soluzioni di un set ristretto di dati (inferiori a 500 dati) ha bisogno di addestrarsi su un *train set* di almeno 1 milione di dati, mentre per ottenere le stesse prestazioni su un numero molto più grande di esempi saranno richiesti almeno diverse decine di milioni di dati.

Per quanti riguarda i risultati sul modello ODE 1, riportate in Figura 25, le prestazioni su 1000, 10000 e 100000 sono abbastanza simili al modello BWD, con la differenza che con 100000 dati ODE 1 riesce a raggiungere solo circa un 23% di accuratezza sul *validation* e il *test set*. Con 1 milione di dati invece si ottengono delle accuratezze che si avvicinano ai test nella sezione precedente, con una precisione che si aggira intorno al 75%. Anche qui vediamo dunque che il modello fa un pò più di fatica nel risolvere equazione differenziali ma aumentando i dati forniti per l'addestramento si possono leggermente migliorare le prestazioni.

Infine osservando i risultati sul modello FWD+BWD+IBP, mostrati in Figura 26, anche qui possiamo notare che per 1000 e 10000 dati forniti il modello si comporta in maniera molto simile a quello per il BWD, ottenendo però con 10000 dati delle accuratezze superiori fino a circa il 17% di precisione e con degli errori sul *validation* e sul *test set* più bassi.

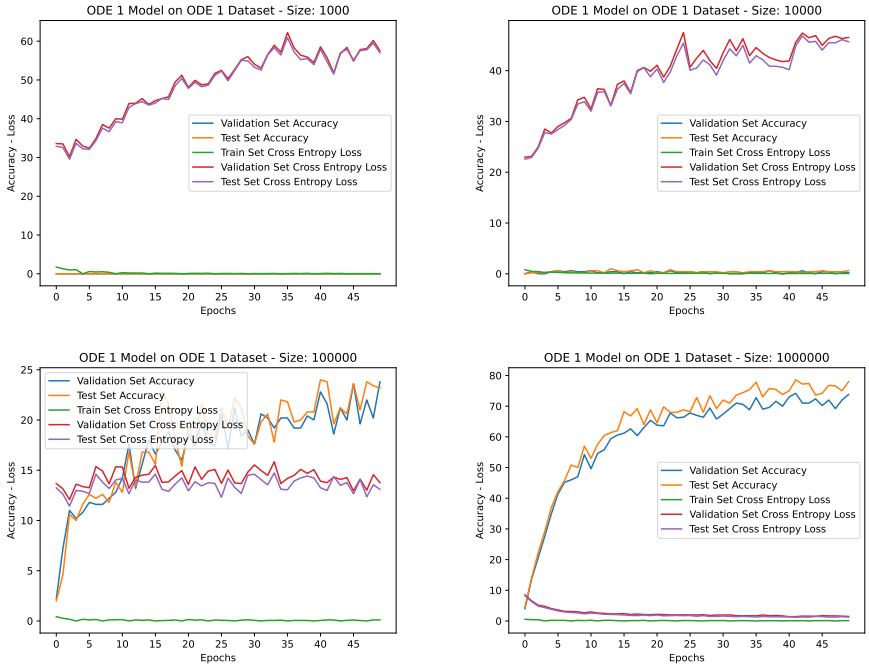

Figura 25: Prestazioni del modello ODE 1 all'aumentare dei dati.

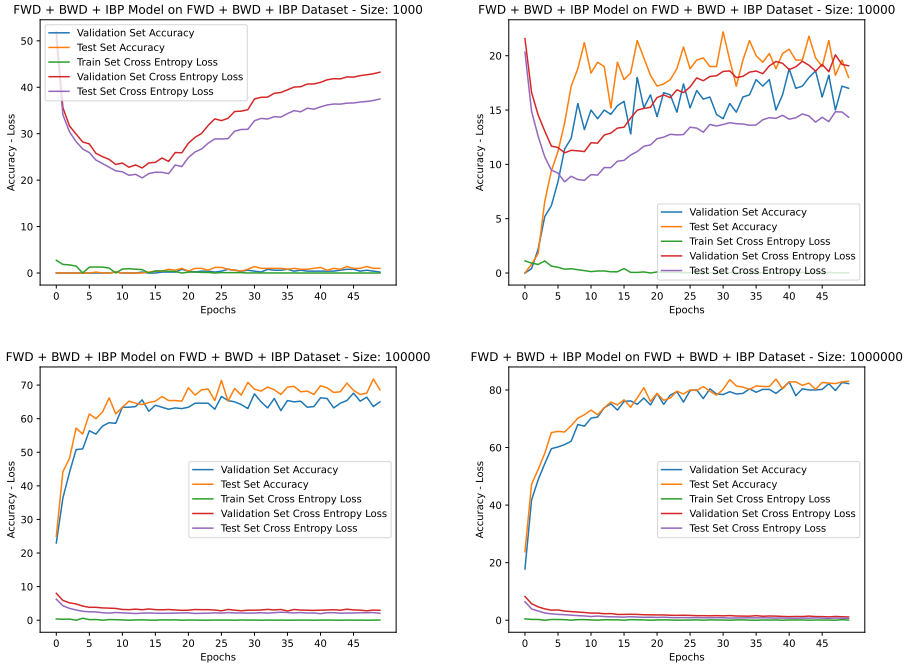

Figura 26: Prestazioni del modello FWD+BWD+IBP all'aumentare dei dati.

Con 100000 ed 1 milione di dati le accuratezze salgono rispettivamente a circa un 65% e un 80% di precisione, con degli errori che si mantengono sempre bassi. A differenza del modello BWD, riesce con più difficoltà a raggiungere delle accuratezze vicine al 100% con 1 milione di dati, questo perché il dataset fornito ad esso per l'addestramento deve contenere dati sufficienti per tutti e tre i tipi di generatore per l'integrazione e non di un solo tipo come avviene nel caso del modello BWD. Aumentando il numero di dati di addestramento dunque aumenteranno anche le prestazioni ed in generale anche in questo caso valgono le stesse considerazioni fatte per il modello BWD.

5.6 SOLUZIONI EQUIVALENTI

Una proprietà interessante dei modelli per le integrazioni e per le equazioni differenziali testati con il metodo *beam search*, è la loro capacità di generare diversi output di soluzioni equivalenti, ma espresse spesso in modi diversi. Ad esempio, consideriamo la seguente equazione differenziale del primo ordine, insieme a una delle sue soluzioni:

$$y' - y - xe^x = 0 \qquad \Longrightarrow \qquad y = e^x\left(\frac{x^2}{2} + c\right)$$

N.	Score	Hypothesis	N.	Score	Hypothesis
1	-0.083924	$x\left(\frac{c}{x} + \frac{x}{2}\right)e^x$	6	-0.194016	$\frac{x^2e^x}{2} + e^{c+x}$
2	-0.089934	$\left(c + \frac{x^2}{2}\right)e^x$	7	-0.209069	$\frac{\left(c+x^2\right)e^x}{2} + e^x$
3	-0.129691	$ce^x + \frac{x^2e^x}{2}$	8	-0.262188	$x\left(\frac{c}{x} + \sinh\left(\log\left(x\right)\right)\right)e^x$
4	-0.143644	$\frac{\left(c+x^2\right)e^x}{2}$	9	-0.275931	$x\left(\frac{c}{x} + \cosh\left(\log\left(x\right)\right)\right)e^x$
5	-0.164454	$\frac{x\left(\frac{c}{x}+x\right)e^x}{2}$	10	-0.309132	$\left(c + x\cosh\left(\log\left(x\right)\right)\right)e^x$

Tabella 6: Ipotesi di soluzioni per l'equazione differenziale: $y' - y - xe^x = 0$.

Nella Tabella 6 vengono riportate le 10 ipotesi, insieme ai punteggi assegnati ad ognuna di loro, che il modello ODE 1 restituisce se viene incaricato di risolvere l'equazione precedente, utilizzando dunque una *beam search* di dimensione 10. Possiamo osservare come tutte le ipotesi generate sono in realtà soluzioni valide di tale equazione, anche se sono

state espresse in modo molto diverso, a volte perché non è stato calcolato un polinomio o raccolto un fattore, oppure perché sono state generate in una forma non semplificata, come ad esempio la funzione $\sinh(\log(x))$ che può essere semplificata in $\frac{x^2-1}{2x}$. Lo stesso comportamento viene riscontrato anche nel caso delle integrazioni. Questa capacità dei modelli di riuscire a generare soluzioni con espressioni equivalenti ma diverse nella forma, senza essere stati esplicitamente addestrati a farlo, è sicuramente un comportamento interessante.

5.7 ALCUNI ESEMPI PRATICI SU PROBLEMI REALI

In questa sezione riportiamo invece alcuni esempi pratici di come i modelli per le equazioni differenziali e per le integrazioni si comportano su alcuni problemi noti. In particolare abbiamo testato il modello ODE 1 sulla risoluzione di alcune equazioni differenziali che devono essere risolte per ricavare la funzione generatrice dei coefficienti binomiali centrali $\binom{2n}{n}$, dei numeri di Catalan $\frac{1}{n+1}\binom{2n}{n}$, del numero medio di confronti e scambi nel Quicksort e del numero di foglie negli alberi di ricerca binaria con n nodi. Mentre per le integrazioni abbiamo testato il modello FWD+BWD+IBP su alcuni integrali ripresi da un libro di matematica. Tutti i test sono stati eseguiti utilizzando una *beam search* di dimensione 50 e tutti e due i modelli sono risultati molto veloci nel generare tutte le ipotesi delle soluzioni, impiegando meno di un secondo in ogni test. Inoltre, vengono riportati anche alcuni esempi del codice Python che è stato utilizzato per impostare e far funzionare i vari modelli nelle predizioni delle soluzioni per i problemi considerati. Il codice riportato e i vari esempi qui discussi possono essere trovati anche nella pagina GitHub di questa tesi: `https://github.com/elia-mercatanti/deep-learning-symbolic-mathematics`, dove oltre al codice originale degli autori sono presenti anche alcuni *notebooks*, scritti durante il lavoro su questa tesi, che descrivono in modo approfondito i test che sono stati eseguiti. Di seguito, discuteremo solo dei test più significativi.

5.7.1 *Test su Equazioni Differenziali*

Per quanto riguarda i coefficienti binomiali centrali $\binom{2n}{n}$, l'obiettivo era risolvere tramite il modello ODE 1 la seguente equazione differenziale

del primo ordine, per ottenere una sua valida soluzione, riportata di seguito a destra.

$$(1-4x)\,y' - 2y = 0 \qquad \Longrightarrow \qquad y = \frac{1}{\sqrt{1-4x}}$$

Per ottenere le soluzioni della precedente equazione per prima cosa è necessario caricare il modello *Transformer* addestrato sulle equazioni differenziali del primo ordine (ODE 1) ed indicare tutta una serie di parametri per specificare come è stato allenato tale modello, come ad esempio il tipo di operatori utilizzati, la lunghezza massima che deve avere una sequenza, oppure i parametri con cui è stato impostato il modello *Transformer* addestrato, come il numero degli *encoder* e dei *decoder* in pila o la dimensione d_{model} utilizzata per i *layers* del modello. Dopodiché, passati questi parametri si possono ottenere e richiedere i due moduli del *Transformer*, il blocco degli *encoder* e dei *decoder*. Questo per permetterci in seguito di codificare la nostra equazione differenziale tramite l'*encoder*, e per ottenere le ipotesi delle soluzioni tramite il *decoder* e il *context vector* restituito dall'*encoder*. Nel Listato 5.1 viene riportato il segmento di codice che ci permette di fare questo e di impostare la dimensione per la *beam search*.

Listing 5.1: Caricamento del modello e recupero dell'*encoder* e del *decoder*

```
model_path = "../models/differential-equations/ode1.pth"

params = AttrDict({
# Environment Parameters
...
# Model Parameters
...
})

env = build_env(params)

modules = build_modules(env, params)
encoder = modules['encoder']
decoder = modules['decoder']

beam_size = 50
```

Successivamente, l'equazione differenziale da risolvere viene dichiarata in forma simbolica tramite la libreria *SymPy* e le sue espressioni polinomiali vengono espanse. Quest'ultima operazione risulta molto utile per aiutare il modello a ipotizzare delle soluzioni corrette, questo perché il dataset ricavato dal generatore delle equazioni differenziali su cui il modello ODE 1 è stato addestrato tende a generare delle equazioni con espressioni molto lunghe le cui soluzioni sono invece corte, come mostra-

to nella Tabella 1 e come già discusso in precedenza. Il modello ODE 1 dunque si aspetta di risolvere equazioni differenziali che hanno queste caratteristiche e potrebbe dunque avere qualche difficoltà nel momento in cui si trovi a dover risolvere equazioni con caratteristiche diverse. In seguito, l'equazione viene trasformata in notazione prefissa, ovvero in

[add, mul, INT-, 2, Y, add, mul, INT-, 4, mul, x, Y', Y'],

dato che il modello è stato addestrato solo su equazioni con tale tipo di notazione. Nel Listato 5.2 viene mostrato il codice usato per queste operazioni.

Listing 5.2: Dichiarazione in forma simbolica, espansione e forma prefissa

```
diff_eq_infix = "(1-4*x)*diff(f(x),x)-2*f(x)"
diff_eq_sympy = sympy.sympify(diff_eq_infix, locals=env.local_dict)
diff_eq_sympy = sympy.expand(diff_eq_sympy)
diff_eq_prefix = env.sympy_to_prefix(diff_eq_sympy)
x1_prefix = env.clean_prefix(diff_eq_prefix)
```

Successivamente l'equazione in notazione prefissa viene trasformata in una sequenza di indici

[0, 33, 54, 72, 83, 78, 33, 54, 72, 85, 54, 12, 79, 79, 0]

con una lunghezza pari alla sequenza prefissa più due *tokens* speciali EOS (*End of Sentence*) per segnalare al modello l'inizio e la fine di tale sequenza. Gli indici vengono ripresi dal vocabolario di tutti i simboli matematici possibili, seguendo la procedura spiegata nel capitolo del *Transformer* e questa sequenza viene poi passata all'*encoder* per essere codificata e ricavarne il *context vector*. Di seguito viene mostrato una parte del *context vector* generato, che ha una dimensione pari a 15×1024 (numero di tokens nella sequenza $\times$ d_{model}).

$$\begin{bmatrix} -0.0489 & -0.0070 & 0.0301 & \dots & 0.0379 & -0.0180 & -0.0294 \\ -0.0186 & -0.0205 & -0.0187 & \dots & -0.0069 & -0.0009 & 0.0130 \\ 0.0160 & -0.0436 & 0.0814 & \dots & 0.0111 & -0.0693 & -0.0459 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ -0.0763 & 0.0338 & 0.0812 & \dots & 0.0214 & -0.0443 & 0.0031 \\ -0.2014 & 0.0209 & 0.0040 & \dots & -0.0047 & -0.0458 & 0.1541 \\ -0.1745 & 0.0363 & -0.0411 & \dots & -0.0154 & -0.0977 & 0.0942 \end{bmatrix}$$

Dopodiché l'output dell'*encoder* viene usato come input per il *decoder* per generare le ipotesi delle soluzioni tramite *beam search*. Questo procedimento viene mostrato nel Listato 5.3.

Listing 5.3: Uso dell'*encoder* e del *decoder* per ricavare le ipotesi delle soluzioni

```
x1 = torch.LongTensor(
  [env.eos_index] +
  [env.word2id[w] for w in x1_prefix] +
  [env.eos_index]
).view(-1, 1)

len1 = torch.LongTensor([len(x1)])
x1, len1 = to_cuda(x1, len1)

with torch.no_grad():
  encoded = encoder('fwd', x=x1, lengths=len1, causal=False).transpose(0, 1)

with torch.no_grad():
  _, _, beam = decoder.generate_beam(encoded, len1, beam_size=beam_size, length_penalty=1.0,
      early_stopping=1, max_len=params.max_len)

hypotheses = beam[0].hyp
```

Infine le ipotesi generate dal modello vengono valutate attraverso la libreria *SymPy* per verificarne la validità. In particolare, nel caso delle equazioni differenziali, ogni ipotesi di soluzione restituita, ovvero sostanzialmente una funzione f, viene sostituita nell'equazione differenziale originale che vogliamo risolvere al posto di y e l'equazione viene valutata con *SymPy*. Se il risultato è pari a 0 la soluzione sarà valida altrimenti viene segnata come incorretta. Le varie ipotesi che il modello restituisce vengono poi stampate in una tabella assieme ai punteggi che il modello ha assegnato loro. Nel Listato 5.4 viene riportato il codice utilizzato per questo procedimento.

Listing 5.4: Validazione delle ipotesi del modello e stampa in tabella

```
rows = numpy.arange(1, beam_size + 1)
columns = ['Score', 'Solution Hypothesis', 'Valid']
results = []

for score, sequence in sorted(hypotheses, reverse=True):
  ids = sequence[1:].tolist()
  hyp_prefix = [env.id2word[word_id] for word_id in ids]

  try:
    hyp_infix = env.prefix_to_infix(hyp_prefix)
    hyp_sympy = env.infix_to_sympy(hyp_infix)

    validation = "YES" if simplify(diff_eq_sympy.subs(f(x), hyp_sympy).doit(), seconds=1) ==
        0 else "NO"

    hyp_expr = "$" + sympy.latex(env.infix_to_sympy(hyp_infix)) + "$"

  except (InvalidPrefixExpression, ValueErrorExpression):
    validation = "INVALID PREFIX EXPRESSION"
    hyp_expr = hyp_prefix

  results.append([score, hyp_expr, validation])

```

```
23 pandas.set_option('max_colwidth', None)
24 pandas.DataFrame(results, index=rows, columns=columns).style.set_properties(**{'text-align':
       'center'})
```

Per l'equazione differenziale per il calcolo della funzione generatrice dei coefficienti binomiali centrali $\binom{2n}{n}$ le ipotesi di soluzioni che il modello ODE 1 restituisce sono mostrate in Tabella 7, dove vengono riportate solo le prime 10. Possiamo notare come la prima ipotesi proposta è esattamente nella stessa forma della soluzione che avevamo riportato in precedenza, se scegliamo $c = 1$ per il valore della costante, mentre tutte le altre sono comunque tutte soluzioni valide rappresentate in forme diverse, spesso semplicemente non semplificate. In questo caso il modello riesce dunque senza fatica a trovare diverse soluzioni per la stessa equazione.

N.	Score	Hypothesis	Valid	N.	Score	Hypothesis	Valid
1	-0.094559	$\frac{c}{\sqrt{1-4x}}$	YES	6	-0.166589	$\frac{c}{\sqrt{\frac{1}{4}-x}}$	YES
2	-0.108193	$\sqrt{\frac{cx}{-4x^2+x}}$	YES	7	-0.190779	$\frac{c}{\sqrt{\frac{1}{2}-2x}}$	YES
3	-0.137775	$\sqrt{\frac{c}{1-4x}}$	YES	8	-0.198530	$\frac{x}{\sqrt{cx(-4x^2+x)}}$	YES
4	-0.159204	$\frac{cx}{\sqrt{x(-4x^2+x)}}$	YES	9	-0.200983	$x\sqrt{\frac{c}{x(-4x^2+x)}}$	YES
5	-0.165491	$c\sqrt{\frac{x}{-4x^2+x}}$	YES	10	-0.206921	$\frac{c}{\sqrt{\frac{-4x^2+x}{x}}}$	YES

Tabella 7: Ipotesi di soluzioni per l'equazione differenziale: $(1-4x)\,y' - 2y = 0$.

Per calcolare invece la funzione generatrice dei numeri di Catalan $\frac{1}{n+1}\binom{2n}{n}$ dovremo risolvere la seguente equazione differenziale, di cui riportiamo di seguito anche una sua soluzione.

$$y' + \frac{1-2x}{x-4x^2}y - \frac{1}{x-4x^2} = 0 \qquad \Longrightarrow \qquad y = \frac{1-\sqrt{1-4x}}{2x}$$

In questo caso, per aiutare il modello ODE 1 a trovare delle soluzioni valide, è stato necessario anche rimuovere i vari denominatori dell'equazione, allungandone così la sua espressione. I risultati ottenuti sono stati riportati nella Tabella 8, dove sono mostrati solo alcuni risultati chiave dei 50 ottenuti tramite *beam search*. Possiamo osservare come in questo caso il modello ha un pò più difficoltà nel trovare una soluzione valida. Infatti la prima che viene trovata compare solo nel sedicesimo output della *beam search*, successivamente però ne vengono trovate altre altrettanto valide e con una forma molto simile. Anche in questo caso sono spesso soluzione non semplificate.

N.	Score	Hypothesis	Valid	N.	Score	Hypothesis	Valid
1	-0.087689	$\frac{c\sqrt{1-4x}+x}{x}$	NO	16	-0.213739	$\frac{c\sqrt{1-4x}+1}{2x}$	YES
2	-0.134191	$\frac{c\sqrt{4x-1}+x}{x}$	NO	23	-0.241732	$\frac{\sqrt{c(1-4x)}+1}{2x}$	YES
3	-0.149671	$\frac{c\sqrt{\frac{1}{4}-x}+x}{x}$	NO	29	-0.260164	$\frac{\sqrt{c(4x-1)}+1}{2x}$	YES
4	-0.172542	$\frac{x+\sqrt{c(1-4x)}}{x}$	NO	36	-0.270136	$\frac{c\sqrt{4x-1}+1}{2x}$	YES
5	-0.173510	$\frac{c\sqrt{\frac{-4x^2+x}{x}}+x}{x}$	NO	46	-0.308814	$\frac{c\sqrt{x-\frac{1}{4}}+1}{2x}$	YES

Tabella 8: Ipotesi di soluzioni per l'equazione: $y' + \frac{1-2x}{x-4x^2}y - \frac{1}{x-4x^2} = 0$.

Riportiamo infine nella Tabella 9 anche le soluzioni ipotizzate per l'equazione differenziale che deve essere risolta per il calcolo della funzione generatrice per il numero di foglie negli alberi di ricerca binaria con n nodi, riportata qui di seguito con una sua soluzione.

$$y' - \frac{2}{1-x}y - 1 = 0 \qquad \Longrightarrow \qquad y = \frac{1}{3}\frac{1}{(1-x)^2} + \frac{1}{3}(x-1)$$

N.	Score	Hypothesis	Valid	N.	Score	Hypothesis	Valid
1	-0.062180	$\frac{c}{(1-x)^2} + \frac{x}{3} - \frac{1}{3}$	YES	6	-0.112487	$\frac{c}{x(x-2)+1} + \frac{x}{3}$	NO
2	-0.086932	$\frac{cx^2}{\left(-x^2+x\right)^2} + \frac{x}{3} - \frac{1}{3}$	YES	7	-0.117153	$\frac{cx^2}{\left(-x^3+x^2\right)^2} + \frac{x}{3}$	NO
3	-0.094559	$\frac{cx^2}{\left(-x^2+x\right)^2} + \frac{x}{3}$	NO	8	-0.119727	$\frac{c}{x^2-2x+1} + \frac{x}{3}$	NO
4	-0.097033	$\frac{c}{(1-x)^2} + \frac{x}{3}$	NO	9	-0.124708	$\frac{cx^2}{\left(-x^2+x\right)^2} + \frac{x}{3} - \frac{1}{2}$	NO
5	-0.106211	$\frac{c}{(1-x)^2} + \frac{x}{3} - \frac{1}{2}$	NO	10	-0.136849	$\frac{cx^2}{\left(x^2-x\right)^2} + \frac{x}{3} - \frac{1}{3}$	YES

Tabella 9: Ipotesi di soluzioni per l'equazione differenziale: $y' - \frac{2}{1-x}y - 1 = 0$.

Anche in questo caso il modello ODE 1, rimuovendo i denominatori dell'equazione per allungarne l'espressione, riesce a trovare subito due prime soluzioni valide per l'equazione precedente, diverse fra loro nella forma solo per il fatto che nella seconda non è stata raccolta una x nel denominatore della prima frazione. Le altre ipotesi si avvicinano molto come forma ad una soluzione corretta ma non sono valide, ad eccezione della decima ipotesi che invece è giusta.

Solo nel caso dell'equazione differenziale per il calcolo della funzione generatrice del numero medio di scambi del Quicksort, $y' - \frac{2}{1-x}y -$

$\frac{x^2(3-x)}{6(1-x)^3} = 0$, il modello ODE 1 non è stato in grado nei nostri test di trovare una soluzione valida con una *beam search* di dimensione 50, anche espandendo l'espressione matematica dell'equazione per facilitare il modello nelle sue ipotesi.

5.7.2 *Test su Integrali*

Nei nostri test abbiamo voluto verificare anche come si comportasse il modello per l'integrazione FWD+BWD+IBP su alcuni integrali complessi ripresi da un libro di matematica. Per impostare il modello e per utilizzarlo nella generazione delle ipotesi di soluzioni per integrali si possono utilizzare in modo simile le stesse procedure viste nel caso delle equazioni differenziali della sezione precedente, con l'unica differenza data dal modo in cui vengono validate le soluzioni restituite dal modello. Nel caso degli integrali infatti, ogni ipotesi di soluzione viene derivata rispetto a x, e successivamente viene eseguita la differenza tra il risultato ottenuto e la funzione originale che volevamo integrare. Se questa differenza è uguale a 0 allora la soluzione sarà valida altrimenti viene segnata come incorretta. Anche in questo caso ogni ipotesi che il modello restituisce viene poi stampata in una tabella assieme al punteggio che il modello le assegna. Riportiamo di seguito due esempi dei vari test che abbiamo effettuato.

N.	Score	Hypothesis	Valid	N.	Score	Hypothesis	Valid
1	-0.002065	$\operatorname{acosh}(x) - \operatorname{asinh}(x)$	YES	6	-0.366785	$\frac{(1+\log(5))(-\operatorname{acosh}(x)+\operatorname{asinh}(x))}{-\log(5)-1}$	YES
2	-0.358701	$\frac{(1+e^2)(-\operatorname{acosh}(x)+\operatorname{asinh}(x))}{-e^2-1}$	YES	7	-0.367217	$\left(1+\sqrt{2}\right)\left(\operatorname{acosh}(x)+\sqrt{2}\operatorname{acosh}(x)-\sqrt{2}\operatorname{asinh}(x)\right)$	NO
3	-0.359802	$\frac{(1+\sqrt{2})(-\operatorname{acosh}(x)+\operatorname{asinh}(x))}{-\sqrt{2}-1}$	YES	8	-0.367492	$\frac{(1+\sqrt{5})(-\operatorname{acosh}(x)+\operatorname{asinh}(x))}{-\sqrt{5}-1}$	YES
4	-0.363234	$\frac{(1+e^5)(-\operatorname{acosh}(x)+\operatorname{asinh}(x))}{-e^5-1}$	YES	9	-0.367903	$\frac{(\log(2)+1)(-\operatorname{acosh}(x)+\operatorname{asinh}(x))}{-1-\log(2)}$	YES
5	-0.363832	$\frac{(1+e^4)(-\operatorname{acosh}(x)+\operatorname{asinh}(x))}{-e^4-1}$	YES	10	-0.371546	$\frac{(1+\log(4))(-\operatorname{acosh}(x)+\operatorname{asinh}(x))}{-\log(4)-1}$	YES

Tabella 10: Ipotesi di soluzioni per l'integrale: $\int \frac{\sqrt{x^2+1}-\sqrt{x^2-1}}{\sqrt{x^4-1}}\,dx$.

N.	Score	Hypothesis	Valid	N.	Score	Hypothesis	Valid
1	-0.026890	$-x+(x+\log(x))\log(\lvert x\rvert)-\frac{\log(x)^2}{2}$	YES	6	-0.116486	$-x+\frac{(x+\log(x))\log(x^4)}{4}-\frac{\log(x)^2}{2}$	YES
2	-0.054778	$-x+\frac{(x+\log(x))\log(x^2)}{2}-\frac{\log(x)^2}{2}$	YES	7	-0.137718	$-x+(x+\log(x))\log(\lvert x\rvert)-\frac{(\log(x)-1)^2}{2}-\log(x)$	YES
3	-0.091177	$-x+(x+\log(x)+1)\log(\lvert x\rvert)-\frac{\log(x)^2}{2}-\log(x)$	YES	8	-0.143895	$-x-(x+\log(x))\log(\lvert x\rvert)-\frac{\log(x)^2}{2}$	NO
4	-0.112607	$-x-(x+\log(x))\log\left(\frac{1}{\lvert x\rvert}\right)-\frac{\log(x)^2}{2}$	YES	9	-0.154724	$-x+\frac{(x+\log(x))\log(x^6)}{6}-\frac{\log(x)^2}{2}$	YES
5	-0.113841	$-x-\frac{(x+\log(x))\log\left(\frac{1}{x^2}\right)}{2}-\frac{\log(x)^2}{2}$	YES	10	-0.158175	$\frac{x\log(x^2)}{2}-x+\frac{\log(\lvert x\rvert)^2}{2}$	NO

Tabella 11: Ipotesi di soluzioni per l'integrale: $\int \frac{(x+1)\log(|x|)}{x}\,dx$.

Nella Tabella 10 e 11 vengono riportate rispettivamente le prime 10 ipotesi del modello per la risoluzione dei seguenti due integrali.

$$\int \frac{\sqrt{x^2+1}-\sqrt{x^2-1}}{\sqrt{x^4-1}}\,dx \qquad \text{e} \qquad \int \frac{(x+1)\log(|x|)}{x}\,dx$$

Osservando le tue tabelle possiamo notare come il modello FWD+BWD+IBP in entrambi i casi riesce senza problemi a trovare diverse soluzioni dei precedente integrali, in particolare, per il primo integrale la prima ipotesi generata dal modello (la soluzione che ritiene più probabile) corrisponde anche alla forma più compatta della soluzione corretta. Anche per tutti gli altri test effettuati su altri tipi di integrali il modello ha sempre trovato almeno una soluzione nelle prime 50 ipotesi.

5.8 COMPARAZIONE CON FRAMEWORKS DI MATEMATICA

I due ricercatori *Lample* e *Charton* nel loro lavoro di ricerca hanno confrontato i loro modelli anche con tre popolari framework di matematica simbolica: *Mathematica* (versione 12.0.0.0), *Maple* (2019) e *Matlab* (R2019a). Per fare questo, le sequenze in notazione prefissa nei vari *test set* dei dataset sono state riconvertite nelle loro rappresentazioni infisse e date così in input ai tre framework. Per ogni specifico input, i tre *computer algebra systems* possono restituire una soluzione, possono non fornire alcuna soluzione (o in alcuni casi restituiscono una soluzione con integrali o funzioni speciali), oppure, nel caso di *Mathematica*, possono terminare il tempo a disposizione per la ricerca di una soluzione dopo un determinato *delay* impostato dall'utente. Nei test eseguiti con *Mathematica* dunque, nel momento in cui il tempo a disposizione selezionato è scaduto, è stato concluso che tale framework non è stato in grado di calcolare una soluzione, sebbene *Mathematica* avrebbe forse potuto trovare una soluzione nel caso in cui gli fosse stato concesso più tempo. Per quanto riguarda l'integrazione, le prestazioni sono state valutate solo sul *test set* del dataset BWD, dato che i dati FWD ad esempio, per costruzione, sono costituiti solo da integrali generati da un framework di matematica simbolica (*SymPy* in questo caso), rendendo dunque il confronto su tale dataset poco interessante. I dataset per le equazioni differenziali sono invece stati utilizzati entrambi per i confronti con i tre *computer algebra systems*. Nella Tabella 12, vengono riportate la accuratezze per i modelli BWD, ODE 1 e ODE 2 con diverse dimensioni di *beam search*, e per i tre framework di matematica simbolica, dove per *Mathematica* è stato scelto un *delay* di timeout di 30 secondi. Dato che anche utilizzando dei

limiti di timeout la valutazione avrebbe richiesto troppo tempo sui *test set* originali di ogni dataset, il team di ricercatori ha scelto di valutare le prestazioni su dei *test set* più piccoli contenenti 500 equazioni ciascuno, su cui ogni modello è stato rivalutato.

	Integrazione (BWD)	**ODE 1**	**ODE 2**
Mathematica (30 s)	84.0	77.2	61.6
Matlab	65.2	-	-
Maple	67.4	-	-
Beam Search 1	98.4	81.2	40.8
Beam Search 10	99.6	94.0	73.2
Beam Search 50	99.6	97.0	81.0

Tabella 12: Confronto dei modelli con *Mathematica*, *Maple* e *Matlab*.

Sia sulle integrazioni che nella risoluzione di equazioni differenziali, possiamo osservare che i modelli *Transformer* addestrati superano in modo significativo *Mathematica*. Per quanto riguarda l'integrazione, il modello BWD ottiene un accuratezza pari al 99.6%, mentre *Mathematica* raggiunge solo l'84% di precisione. Sulle equazioni differenziali del primo ordine, *Mathematica* è quasi alla pari del modello ODE 1 quando viene utilizza una *beam search* di dimensione 1, dunque una *greedy search*. Tuttavia, utilizzando una *beam search* di dimensione 50, l'accuratezza del modello ODE 1 passa dall'81,2% al 97,0%, una precisione che supera ampiamente *Mathematica*. Delle osservazioni simili possono essere fatte anche per le equazioni differenziali del secondo ordine, dove l'utilizzo della *beam search* risulta ancora più importante, poiché il numero di soluzioni equivalenti risulta in questo caso più grande. In media, *Matlab* e *Maple*, sui problemi che sono stati testati, hanno invece prestazioni leggermente inferiori rispetto a *Mathematica*. La Tabella 13 mostra alcuni esempi di problemi che i modelli *Transformer* possono risolvere, ma su cui *Mathematica* e *Matlab* non trovano una soluzione.

Inoltre è stato osservato che anche il framework di matematica simbolica *SymPy* viene superato in prestazioni dai modelli *Transformer*, in particolare dal modello FWD. Il generatore del dataset FWD crea infatti un insieme di coppie (f, F) di funzioni f con i loro integrali F ed esso si basa proprio sul framework *SymPy* per calcolare l'integrale delle funzioni generate casualmente, come spiegato anche nella Sezione 2.6. *SymPy* però non è un software perfetto, e non riesce a calcolare l'integrale di molte

Problema	Soluzione
$\int \frac{1-x^4}{(1+x^2+x^4)\sqrt{1+x^4}}\,dx$	$\arctan\left(\frac{x}{\sqrt{1+x^4}}\right)$
$3xy\cos(x) - \sqrt{9x^2\sin(x)^2+1}y' + 3y\sin(x) = 0$	$y = c\exp(\sinh^{-1}(3x\sin(x)))$
$4x^4yy'' - 8x^4y'^2 - 8x^3yy' - 3x^3y'' - 8x^2y^2 - 6x^2y' - 3x^2y'' - 9xy' - 3y = 0$	$y = \frac{c_1+3x+3\log(x)}{x(c_2+4x)}$

Tabella 13: Problemi risolvibili con i modelli *Transformer* ma non con *Mathematica* e *Matlab*.

funzioni integrabili. In particolare, i due ricercatori hanno osservato che l'accuratezza di *SymPy* sul *test set* del dataset BWD raggiunge solo il 30% di precisione. Mentre il modello addestrato sul dataset FWD, come abbiamo mostrato nella Tabella 5, ottiene solo un'accuratezza del 15,7% sul dataset BWD. Tuttavia, *Lample* e *Charton* hanno osservato che il modello addestrato sul dataset FWD a volte è in grado di calcolare l'integrale di funzioni che *SymPy* non può risolvere. Ciò significa che addestrandosi solo sulle funzioni che *SymPy* può integrare, il modello FWD è stato in grado di generalizzare su funzioni che *SymPy* non può integrare. Nella Tabella 14 vengono riportati alcuni esempi di questi integrali [1].

Integrale	Soluzione
$\int x^2(\tan^2(x)+1) + 2x\tan(x) + 1\,dx$	$x^2\tan(x) + x$
$\int 1 + \frac{2\cos(2x)}{\sqrt{\sin^2(2x)+1}}\,dx$	$x + \operatorname{asinh}(\sin(2x))$
$\int \frac{x\tan(x) + \log(x\cos(x)) - 1}{\log(x\cos(x))^2}\,dx$	$\frac{x}{\log(x\cos(x))}$

Tabella 14: Integrali risolvibili dal modello FWD ma non da *SymPy*.

6

CONCLUSIONI

Il primo scopo di questa tesi, è stato quello di approfondire il lavoro dei ricercatori *Lample* e *Charton* [1] sul cercare di applicare le reti neurali, e in particolare le tecniche di *deep learning*, nella risoluzione di problemi legati alla matematica simbolica, mostrando come i modelli *Sequence to Sequence* possono essere impiegati per quesiti di matematica che sono a volte difficili da risolvere come le integrazioni o la risoluzione di equazioni differenziali. Sono stati presentati dunque diversi approcci per generare dei dataset arbitrariamente grandi, formati da insiemi di equazioni con le loro relative soluzioni. E' stato mostrato come diversi modelli *Transformer* addestrati su questi tipi di dataset possono funzionare molto bene sia per il calcolo degli integrali, sia per la risoluzione di equazioni differenziali, con prestazioni che superano anche i framework di matematica simbolica classici come *Matlab* e *Mathematica* che si basano su un gran numero di algoritmi ed euristiche per ottenere le loro soluzioni. Inoltre, i risultati mostrano anche che i modelli *Transformer* sono in grado di ottenere diverse soluzioni valide rappresentate con espressioni anche molto differenti ma equivalenti tra loro e restituendo le varie ipotesi in meno di un secondo data l'efficiente parallelizzazione dei modelli *Transformer*. Questi risultati sono sicuramente molto sorprendenti anche per via della nota difficoltà dei modelli basati su reti neurali di eseguire compiti matematici più semplici come l'addizione o la moltiplicazione di numeri.

Tuttavia, le ipotesi e le soluzioni proposte possono essere a volte errate e spesso è necessario concedere ai modelli la possibilità di generare più ipotesi per ottenere almeno una soluzione valida. Nel caso delle equazioni differenziali i modelli *Transformer* proposti hanno una leggera difficoltà in più nel generare delle soluzioni valide rispetto alle integrazioni e spesso in questi casi è necessario aiutare i modelli andando ad espandere il più possibile le espressioni delle equazioni che andranno ad elaborare. Oltre a ciò, la correttezza di una soluzione non è fornita dal modello

stesso, ma è affidata ad un framework di matematica simbolica esterno, nel caso di studio la libreria *SymPy*. Infine, per il corretto addestramento dei modelli e per poter ottenere delle accuratezze elevate sono richiesti dataset estremamente grandi composti da almeno decine di milioni di dati e che devono essere basati su generatori che siano in grado di rappresentare correttamente l'intero spazio dei problemi che si vogliono risolvere. I risultati però dimostrano certamente che in futuro i framework standard di matematica simbolica potrebbero trarre un grande vantaggio dall'integrazione nei loro risolutori di componenti basati su modelli di reti neurali e sul *deep learning*.

BIBLIOGRAFIA

[1] Guillaume Lample, François Charton. Deep Learning for Symbolic Mathematics. *arXiv preprint arXiv:1912.01412*, 2019. (Cited on pages 8, 15, 44, 67, 69, 70, 86, and 87.)

[2] Stephen Ornes. Symbolic Mathematics Finally Yields to Neural Networks. *Quanta Magazine*. `https://www.quantamagazine.org/symbolic-mathematics-finally-yields-to-neural-networks-20200520/`. (Cited on pages 8 and 11.)

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *In Advances in Neural Information Processing Systems*, pp. 6000–6010, 2017. (Cited on pages 10, 44, 55, 56, 58, 59, 60, 62, 63, 64, 65, and 66.)

[4] Robert H. Risch. The solution of the problem of integration in finite terms. *Bull. Amer. Math. Soc.*, 76(3):605–608, 05 1970. (Cited on pages 13 and 15.)

[5] Wojciech Zaremba, Karol Kurach, and Rob Fergus. Learning to discover efficient mathematical identities. *In Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'14, pp. 1278–1286, Cambridge, MA, USA, 2014. MIT Press. (Cited on page 14.)

[6] Miltiadis Allamanis, Pankajan Chanthirasegaran, Pushmeet Kohli, and Charles Sutton. Learning continuous semantic representations of symbolic expressions. *In Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pp. 80–88. JMLR.org, 2017. (Cited on page 14.)

[7] Forough Arabshahi, Sameer Singh, and Animashree Anandkumar. Combining symbolic expressions and black-box function evaluations for training neural programs. *In International Conference on Learning Representations*, 2018a. (Cited on page 14.)

[8] Forough Arabshahi, Sameer Singh, and Animashree Anandkumar. Towards solving differential equations through neural programming. 2018b. (Cited on page 14.)

[9] Lukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *CoRR*, abs/1511.08228, 2015. (Cited on pages 14 and 15.)

[10] David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. *In International Conference on Learning Representations*, 2019. (Cited on page 14.)

[11] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. (Cited on page 14.)

[12] Andrew Trask, Felix Hill, Scott E Reed, Jack Rae, Chris Dyer, and Phil Blunsom. Neural arithmetic logic units. *In Advances in Neural Information Processing Systems*, pp. 8035–8044, 2018. (Cited on page 14.)

[13] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *In Advances in Neural Information Processing Systems*, pp. 3104–3112, 2014. (Cited on pages 18 and 48.)

[14] D. Bahdanau, K. Cho, and Y. Bengio, Neural machine translation by jointly learning to align and translate, *arXiv preprint arXiv:1409.0473*, 2014. (Cited on pages 18 and 43.)

[15] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *In Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1556–1566, 2015. (Cited on page 18.)

[16] Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A Smith. Recurrent neural network grammars. *In Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 199–209, 2016. (Cited on page 18.)

[17] Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. *In Advances*

in neural information processing systems, pp. 2773–2781, 2015. (Cited on page 19.)

[18] Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A, Ivanov S, Moore JK, Singh S, Rathnayake T, Vig S, Granger BE, Muller RP, Bonazzi F, Gupta H, Vats S, Johansson F, Pedregosa F, Curry MJ, Terrel AR, Roučka Š, Saboo A, Fernando I, Kulal S, Cimrman R, Scopatz A. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3:e103 `https://doi.org/10.7717/peerj-cs.103` (Cited on page 25.)

[19] D. W. Otter, J. R. Medina and J. K. Kalita, A Survey of the Usages of Deep Learning for Natural Language Processing, *in IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 2, pp. 604-624, Feb. 2021, doi: 10.1109/TNNLS.2020.2979670. (Cited on pages 33, 34, 35, 37, 38, 39, 41, 42, 44, and 45.)

[20] Torfi, A., Shirvani, R.A., Keneshloo, Y., Tavvaf, N., & Fox, E.A., Natural Language Processing Advancements By Deep Learning: A Survey. *ArXiv*, abs/2003.01200, 2020. (Cited on pages 33, 37, 38, 39, 41, 47, 48, 50, 52, and 54.)

[21] Vedantam, Vamsi. The Survey - Advances in Natural Language Processing using Deep Learning, *Turkish Journal of Computer and Mathematics Education*, Vol. 12 No. 4, 2021. (Cited on pages 33, 34, 41, and 44.)

[22] Goodfellow Ian, Bengio Yoshua, Courville Aaron, 6.2.2.3 Softmax Units for Multinoulli Output Distributions, Deep Learning, *MIT Press*, pp. 180–184, 2016, ISBN 978-0-26203561-3. (Cited on pages 40 and 51.)

[23] Christopher Manning, Richard Socher, Milad Mohammadi, Rohit Mundra, Richard Socher, Lisa Wang, Amita Kamath, Language Models, RNN, GRU and LSTM, *CS224n: Natural Language Processing with Deep Learning, Lecture Notes: Part V*, Winter 2019, `https://web.stanford.edu/class/cs224n/readings/cs224n-2019-notes05-LM_RNN.pdf`. (Cited on page 40.)

[24] Hochreiter S., Bengio Y., Frasconi P., Schmidhuber J, Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, In Kremer, S. C., Kolen, J. F. (eds.), *A Field Guide to Dynamical Recurrent*

Neural Networks, IEEE Press, 2001, ISBN 0-7803-5369-2. (Cited on pages 41 and 44.)

[25] Christopher Manning, Richard Socher, Guillaume Genthial, Lucas Liu, Barak Oshri, Kushal Ranjan, Neural Machine Translation, Seq2seq and Attention, *CS224n: Natural Language Processing with Deep Learning, Lecture Notes: Part VI*, Winter 2019, https://web.stanford.edu/class/cs224n/readings/cs224n-2019-notes06-NMT_seq2seq_attention.pdf. (Cited on pages 43, 44, 47, 48, 50, 52, and 54.)

[26] Christopher Manning, Richard Socher, Francois Chaubard, Michael Fang, Guillaume Genthial, Rohit Mundra, Richard Socher, Word Vectors I: Introduction, SVD and Word2Vec, *CS224n: Natural Language Processing with Deep Learning, Lecture Notes: Part I*, Winter 2019, https://web.stanford.edu/class/cs224n/readings/cs224n-2019-notes01-wordvecs1.pdf. (Cited on page 46.)

[27] Jason Brownlee, What Are Word Embeddings for Text?, *Deep Learning for Natural Language Processing - Machine Learning Mastery*, Ottobre 2017, https://machinelearningmastery.com/what-are-word-embeddings/. (Cited on page 46.)

[28] Elena Voita, Sequence to Sequence (seq2seq) and Attention, *NLP Course | For You*, Settembre 2020, https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html. (Cited on pages 58, 59, 60, 62, and 64.)

[29] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *In ICLR*, 2015. (Cited on pages 66 and 69.)