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

Tesi di Laurea

TITOLO ITALIANO

TITOLO INGLESE

ELIA MERCATANTI

Relatore: *Donatella Merlini*
Correlatore: *Correlatore*

Anno Accademico 2020-2021

INDICE

ELENCO DELLE FIGURE

"Inserire citazione"
— Inserire autore citazione

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 app 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'83% 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 è bloccato 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 abbastanza vicine a quelle corrette 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 branchia 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 rappresentare qualsiasi cosa come lettere, parole o serie temporali, e restituisce 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.

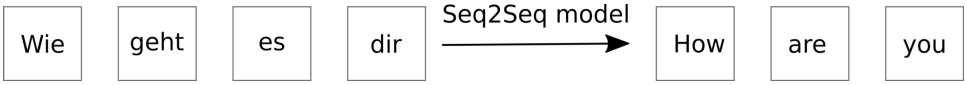

Figura 1: Sequenza di input in tedesco trasformata in una sequenza di output tradotta in inglese.

In particolare i due ricercatori 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. La struttura ad albero 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

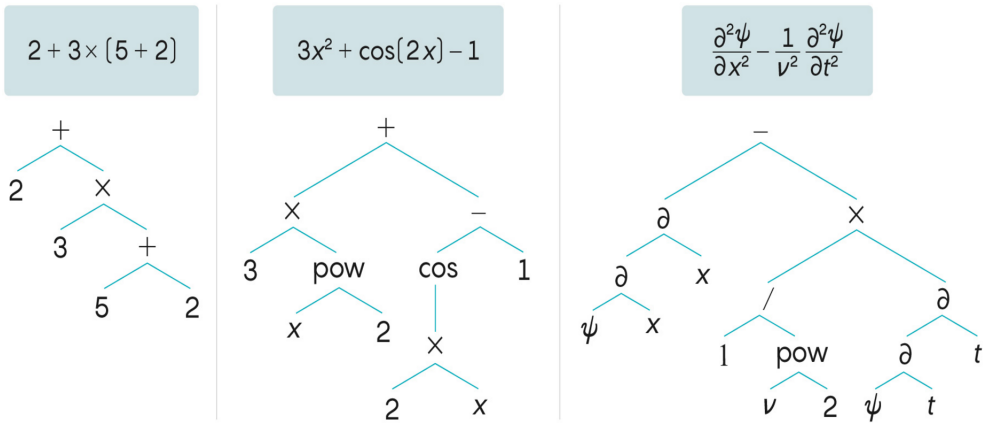

Figura 2: Espressioni matematiche in forma di albero

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 avendo un po' meno successo nelle equazioni differenziali ordinarie, mantenendo però anche in questo caso accuratezze abbastanza alte.

Per quasi tutti i problemi, il programma ha impiegato meno di 1 secondo per generare soluzioni corrette e sui problemi di integrazione, ha superato 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].

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 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 sono 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 $\frac{\partial^2\psi}{\partial^2 x^2}-\frac{1}{v^2}\frac{\partial^2\psi}{\partial^2 t^2}$:

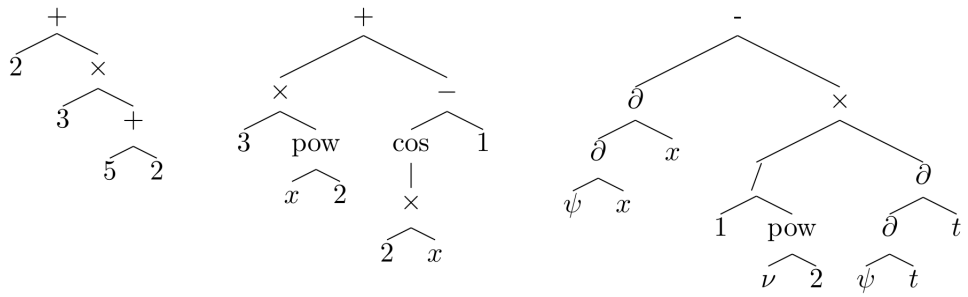

Figura 3: Espressioni matematiche in forma di albero.

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:

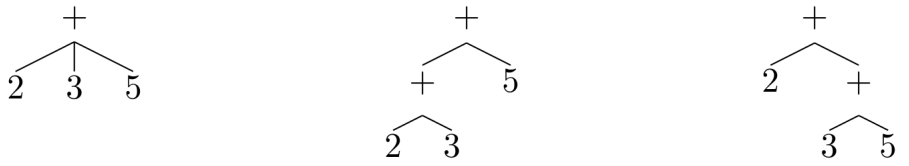

Figura 4: Possibili varianti di alberi ottenuti dall'espressone $2 + 3 + 5$.

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 $-x$:

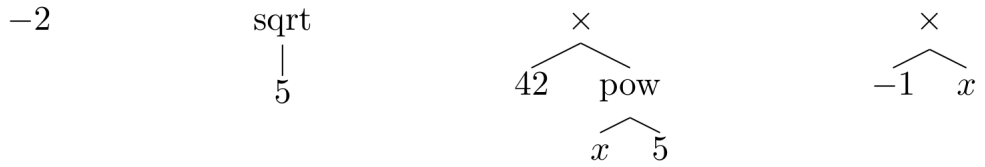

Figura 5: Alberi per le espressioni -2, $\sqrt{5}$, $42x^5$, e $-x$.

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.*, 2015 [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. 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.

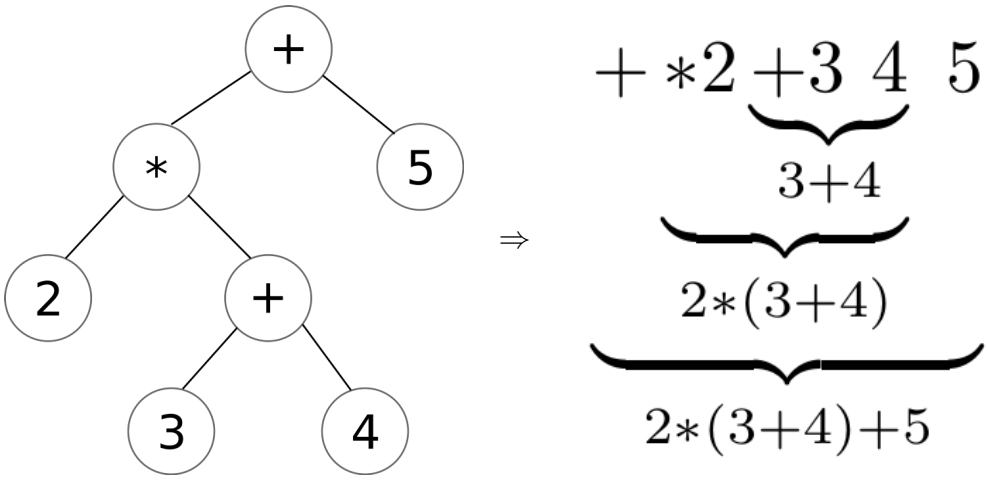

Figura 6: Da albero a sequenza in notazione prefissa per $2 * (3 + 4) + 5$.

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 tutti 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 che vogliamo generare con la stessa probabilità.

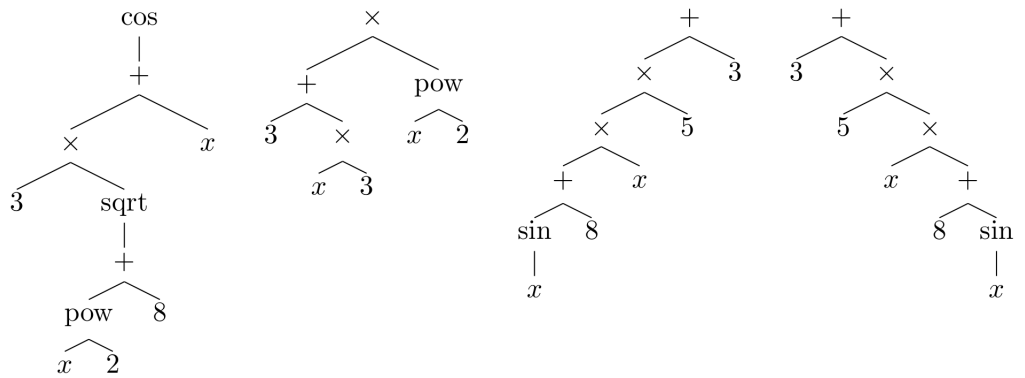

Figura 7: Tipi di alberi differenti che vogliamo generare.

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 potrebbe 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 i problemi accennati prima.

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 calcole 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 nella 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 si 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 ad eccezione 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 i valori 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, attraverso le dimensioni dei dataset e 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 equazione differenziali soddisfatte da esse.

3

NATURAL LANGUAGE PROCESSING E MODELLI SEQUENCE TO SEQUENCE

4

IL MODELLO TRANSFORMER

5

VERIFICHE SPERIMENTALI

BIBLIOGRAFIA

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

[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 page 10.)

[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 page 18.)

[14] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *In International Conference on Learning Representations (ICLR)*, 2015. (Cited on page 18.)

[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.)