Învățare automată pentru matematică simbolică

Adrian Manea, 510 SLA

16 decembrie 2019

Cuprins

1 Preliminarii **2**
1.1 Introducere și motivație 2
1.2 Expresii matematice prin arbori sintactici 2

2 Probleme și soluții **5**
2.1 Generarea funcțiilor și a primitivelor 5
2.2 Generarea soluțiilor EDO I 6

3 Rezultate și concluzii **7**
3.1 Rezultate 7
3.2 Critici 8

Index **9**

Bibliografie **9**

1 Preliminarii

1.1 Introducere și motivație

Articolul își propune să introducă o metodă prin care, folosind învățarea automată, să se genereze expresii matematice de diverse complexități și forme. Utilizarea acestor expresii este, de exemplu, pentru rezolvarea problemelor matematice care necesită, din motive teoretice, căutarea unor soluții într-un spațiu foarte mare.

Mai mult decît atît, însăși problema generării expresiilor simbolice este una complicată pentru rețele neuronale, iar abordarea din articolul [Lample şi Charton, 2019] pe care îl prezentăm este una specifică traducerilor automate sau a procesării limbajului natural (NLP).

Autorii remarcă faptul că tehnicile folosite în NLP pot fi utile și în cazul expresiilor matematice simbolice. Dacă sistemele de tip *computer algebra* de obicei rezolvă probleme matematice prin algoritmi foarte sofisticați, oamenii lucrează prin identificarea tiparelor în expresiile matematice. Nicio rețea neuronală existentă în prezent nu a fost folosită pentru identificarea tiparelor în expresii matematice, însă.

1.2 Expresii matematice prin arbori sintactici

Autorii propun utilizarea unei metode de traducere de tip *sequence to sequence* (seq2seq, pe scurt), descrisă cu tot cu implementare în Python în [Chollet, 2017]. Pentru aceasta, expresiile matematice sînt descompuse folosind arbori de sintaxă, iar reprezentarea lor se face folosind așa-numita *formă poloneză* cu prefix. Astfel, expresia scrisă în mod clasic prin:

$$2 + 3 \cdot (5 + 2)$$

va fi descrisă în notația poloneză prin `[+ 2 * 3 + 5 2]`, iar prin arbore binar în forma din figura 1.1.

Similar se pot coda și expresii mai complicate, precum, de exemplu (v. fig. 1.2):

$$\frac{\partial^2 u}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2}.$$

Cu ajutorul arborilor, înțelesurile expresiilor sînt imediate și în plus, folosind notația poloneză, avem o legătură directă între expresia scrisă în limbaj matematic și arborele său sintactic.

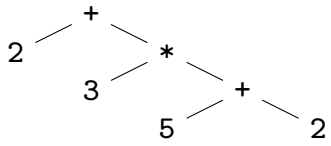

Figura 1.1: Arbore binar pentru expresia `[+ 2 * 3 + 5 2]`

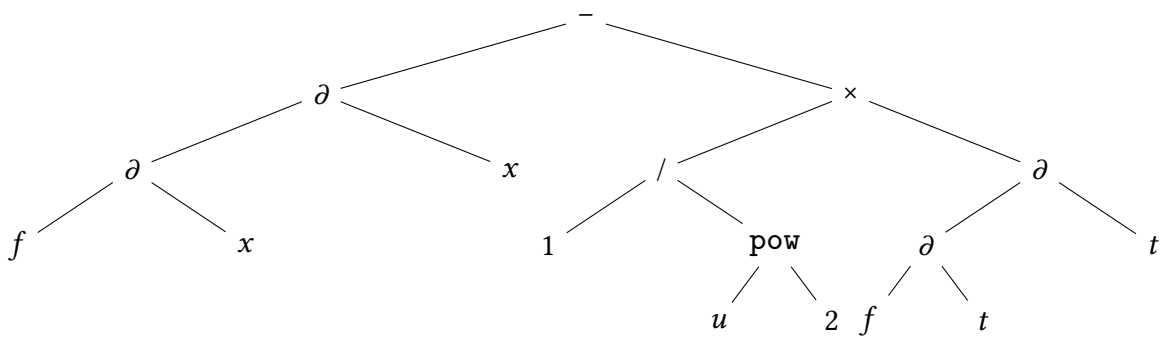

Figura 1.2: Arbore binar pentru expresia $\frac{\partial^2 u}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2}$

Evident, o primă problemă remarcată este că un arbore sintactic nu poate identifica expresiile matematice fără sens, precum împărțirile la zero sau expresii de forma $\log 0$ sau $\sqrt{-\ln(-3)}$.

Însă scopul principal al articolului este să genereze un corpus de funcții și expresii matematice complexe, care apoi pot fi testate dacă sînt soluții pentru probleme complicate, precum ecuații cu derivate parțiale sau ecuații integrale.

Inferența formulelor generate din arbori se face folosind o tehnică specifică, des utilizată împreună cu metoda seq2seq, numită *beam search*, cu mărimi de 1, 10 și 50.

Reluăm pe scurt prezentarea metodei, din [Chollet, 2017]. Cazul general este acela al traducerilor automate, deci cînd se pornește cu o expresie de o anumită lungime și se dorește obținerea unei expresii care poate avea o altă lungime.

Pentru aceasta, avem nevoie de:

- O rețea neuronală recursivă (RNN), care „codează“ secvența de intrare și returnează o stare internă, specifică. Nu vom fi interesați de codarea propriu-zisă, ci de starea specifică returnată, folosită mai departe ca un context;
- O rețea neuronală recursivă de „decodare“, care va fi antrenată să prezică următorul caracter din secvență, dacă i se dau caracterele anterioare.

Pentru *inferență*, adică atunci cînd se dorește să se decodeze secvențe de intrare necunoscute, procedura este cumva diferită:

(1) Se codează secvența de intrare folosind vectori de stare;

(2) Se pornește cu o secvența de lungime 1, alcătuită doar din primul caracter al secvenței de intrare;

(3) Se transmit vectorii de stare și primul caracter către decodor, care va produce predicții pentru caracterul ce urmează;

(4) Se estimează următorul caracter folosind metoda argmax și se adaugă rezultatul la șir;

(5) Se repetă pînă se ajunge fie la ultimul caracter din șir, fie la o lungime prestabilită.

Astfel că, în cazul seq2seq aplicat expresiilor matematice reprezentate ca arbori binari, se generează un arbore de căutare de tip *breadth-first* și se caută succesorii, sortîndu-i după cost, dar păstrînd doar un număr de succesori egal cu mărimea razei de căutare (1, 10, 50 în cazul curent).

De îndată ce s-a stabilit metoda de generare și de înțelegere a expresiilor generate (i.e. folosind arbori binari și *beam search*), se poate analiza exact dimensiunea spațiului soluțiilor, care va conține:

- arbori cu cel mult n noduri, pentru un anume n;
- o multime de operatori unari (e.g. sin, cos, exp, log);
- o mulțime de operatori binari (e.g. $+$, $-$, $\times$, `pow`);
- o multime de frunze care să conțină variabile, constante și numere reale (sau întregi, pentru simplitate).

Mulțimile operatorilor unari, respectiv binari au cardinalele bine stabilite, determinate de numerele Catalan, respectiv Schröder, care se pot găsi, de exemplu, și în [Sloane, 1964].

În continuare, prezentăm problemele abordate și soluțiile propuse, folosind instrumentele teoretice de mai sus.

2 Probleme și soluții

Autorii și-au propus să ajute la rezolvarea ecuațiilor diferențiale (eventual cu derivate parțiale) și a ecuațiilor integrale. Ideea de bază este că astfel de probleme sînt foarte greu de rezolvat folosind instrumente matematice standard, însă operațiile în sine, de derivare și integrare, sînt relativ simple, chiar și într-o formă algoritmică.

De aceea, abordarea propusă este să se genereze seturi foarte mari de funcții și expresii matematice, cu zeci de milioane de membri, iar fiecare element să fie apoi testat dacă satisface ecuația dată. Din nou, ideea de bază este că, în locul rezolvării ecuației propriu-zise, se poate doar deriva sau integra, ambele proceduri fiind relativ ieftine din punct de vedere computațional.

Pe lîngă dificultatea eliminării expresiilor fără sens, apare, totodată, și problema funcțiilor ale căror primitive sînt foarte greu sau imposibil de calculat, precum $\exp(x^2)$ sau $\dfrac{\sin x}{x}$. Aceste neajunsuri sînt asumate de autori.

2.1 Generarea funcțiilor și a primitivelor

Autorii propun trei metode de abordare:

(1) **Generarea directă** (`FWD`): se generează funcții aleatorii cu pînă la n operatori și se calculează primitivele lor folosind software specializat (Computer Algebra System, CAS). Dacă programul CAS nu poate calcula primitiva, se elimină funcția generată din mulțime;

(2) **Generarea inversă** (`BWD`): se generează funcții care se derivează folosind un CAS sau altă metodă, care este mult mai ieftină computațional și se poate calcula mereu. Astfel, se înregistrează rezultatul invers: funcția generată de program va fi primitiva celei care se calculează prin derivare;

(3) **Generare inversă și integrare prin părți** (`IBP`): folosind formula de integrare prin părți:

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

cu notațiile standard $F' = f$ și $G' = g$, putem face o combinație a metodelor de mai sus: se dau două funcții generate F și G. Se calculează derivatele, folosind `BWD` și se verifică dacă

fG se găsește deja în mulțime. Dacă da, știm deja integrala sa (din metoda `FWD`) și atunci putem obține integrala Fg din formula de integrare prin părți. Similar dacă Fg a fost deja generată, îi știm integrala din `FWD` și putem obține integrala fG.

2.2 Generarea soluțiilor EDO I

Se pot genera ecuații diferențiale ordinare de ordinul I în felul următor. Se pornește cu o funcție $F(x, y)$ astfel încît ecuația (problema Cauchy) $F(x, y) = c$ să poată fi rezolvată pentru y, dată o constantă c. Cu alte cuvinte, există o funcție f astfel încît pentru orice (x, c), are loc $F(x, f(x, c)) = c$.

Calculăm derivata în raport cu x a acestei expresii și se obține:

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

unde am notat prin f_c funcția $x \mapsto f(x, c)$. Rezultă că, pentru orice constantă c, funcția f_c este o soluție pentru ecuația diferențială de ordinul întîi:

$$F_x + y' F_y = 0,$$

unde $F_x = \dfrac{\partial F(x, y)}{\partial x}$ și similar pentru F_y.

De exemplu:

(1) Se generează o funcție aleatorie: $f(x) = x \log(c/x)$;

(2) Se rezolvă ecuația $c = x \exp\left(\dfrac{f(x)}{x}\right) = F(x, f(x))$;

(3) Se calculează derivata în raport cu x:

$$\exp\left(\frac{f(x)}{x}\right)\left(1 + f'(x) - \frac{f(x)}{x}\right) = 0;$$

(4) Prin simplificare, ajungem la $xy' - y + x = 0$.

Observaţie 2.1: Se poate proceda similar pentru ecuații diferențiale de ordinul al doilea, rezultat pe care îl omitem, indicînd referința [Lample şi Charton, 2019, §3.3].

3 Rezultate și concluzii

3.1 Rezultate

Modelul pe care l-au folosit autorii a conținut:

- expresii cu cel mult $n = 15$ noduri interne (fără rădăcină și frunze);
- $L = 11$ frunze, cu valori din $\{x\} \cup \{-5, \ldots, 5\} - \{0\}$;
- $p_2 = 4$ operatori binari: adunare, scădere, înmulțire, împărțire;
- $p_1 = 15$ operatori unari: exp, log, $\sqrt{}$, funcțiile trigonometrice directe, inverse, hiperbolice și hiperbolice inverse.

Rezultatele sînt detaliate în [Lample şi Charton, 2019, §4]. Totodată, sînt prezentate și comparații cu software specializat, precum Mathematica, Maple și Matlab, cu care modelul concurează în ce privește complexitatea-timp, abaterea fiind de aproximativ ±10%.

Un rezultat interesant este faptul că modelul a atribuit scoruri identice expresiilor echivalente în unele situații, fără a fi antrenat pentru a o face. Astfel, dacă două expresii reprezintă aceeași funcție, ca soluție a unei ecuații diferențiale sau integrale, modelul a returnat scoruri identice, fără a avea o metodă programată de simplificare a expresiilor.

În ansamblu, principalul merit al articolului este de a fi arătat că se pot folosi metode din procesarea limbajului natural pentru a genera expresii matematice. Pentru acestea, operația de eliminare a expresiilor fără sens poate fi făcută destul de simplu, astfel că metoda poate fi considerată eficientă. Mai mult decît atît, abordarea găsită de autori poate fi folosită pentru rezolvarea unor probleme extrem de complicate din matematică, mai precis din analiza reală. În fine, deși modelul găsit de ei nu este verificat formal, autorii speră ca proiectul SymPy sau altele să producă un cadru de verificare formală a expresiilor matematice, care poate fi folosit apoi nu doar pentru validarea modelelor găsite pînă acum, ci și pentru eficientizarea lor.

3.2 Critici

Cea mai mare parte a criticilor, formulate atît pe platforma OpenReview, cît și în articolul [Davis, 2019], sînt îndreptate spre faptul că modelul găsit în [Lample şi Charton, 2019] (LC, pe scurt), nu „știe matematică“, nici măcar la nivel elementar. Astfel, de exemplu, modelul nu poate face simplificări nici măcar ale unor expresii elementare, precum:

$$\int \sin^2(\exp(\exp(x))) + \cos^2(\exp(\exp(x)))dx,$$

integrand care este egal cu 1, din formula fundamentală a trigonometriei. Evident că exemplul de mai sus poate fi făcut de o complexitate sintactică arbitrară sau să se folosească alte formule matematice elementare, dar subtile, într-o expresie oricît de complicată.

Faptul că modelul LC nu face simplificări nu doar că dăunează timpului de execuție, dar există posibilitatea de a rata soluții elementare. Dată fiind complexitatea sintactică a integrandului, este puțin probabil ca modelul LC să încerce $f(x) = x$ ca o primă variantă de răspuns, deși acesta este cel corect. Chiar mai mult, autorii LC remarcă drept „surprinzător“ faptul că modelul face simplificări fără a fi antrenat să o facă. Deci în mod evident, cel puțin în această etapă a dezvoltării, programul nu poate găsi expresii echivalente cu consecvență.

Autorul [Davis, 2019] remarcă o altă situație cînd simplificarea ar fi crucială: presupunem că pornim cu o funcție pentru care LC a calculat integrala. Apoi facem o mică schimbare în funcție. În multe cazuri, integrala va fi mult mai greu de calculat sau poate chiar nu va exista (e.g. $\exp(x) \longrightarrow \exp(x^2)$). Dar modelul LC este făcut astfel încît să producă mereu un rezultat. Neavînd posibilitatea de simplificare, se va verifica dacă derivata rezultatului este egală cu integrandul inițial. Cu cît expresiile sînt mai complicate, cu atît procedura este mai laborioasă.

Un alt caz în care LC ar fi fost bine să „știe matematică“ este cel al derivării funcțiilor compuse. Dată o expresie de forma $f(g(h(j(x))))$, modelul LC nu are o ordine preferată a produslui care apare în derivată. Acest lucru, coroborat cu faptul că nu face simplificări, poate avea o influență semnificativă asupra performanței.

În fine, o ultimă critică adresată în [Davis, 2019] este că, deși LC a fost comparat cu Mathematica, Matlab și Maple, putem afirma că însăși construcția modelului se bazează pe tehnici deja implementate de ani buni în sistemele CAS. Astfel că o comparație poate să fie utilă, însă o abordare mai eficientă și cu adevărat nouă ar trebui să facă mai mult decît să *implementeze* metode simbolice dezvoltate în ultimii 50 de ani în cadrul învățării automate.

Bibliografie

[Chollet, 2017] Chollet, F. (2017). A ten-minute introduction to sequence-to-sequence learning in Keras. Articol disponibil la https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html, accesat decembrie 2019.

[Davis, 2019] Davis, E. (2019). The use of deep learning for symbolic integration: A review of (Lample and Charton, 2019). *arXiv*. https://arxiv.org/abs/1912.05752.

[Lample şi Charton, 2019] Lample, G. şi Charton, F. (2019). Deep learning for symbolic mathematics. *ICLR 2020*. https://arxiv.org/abs/1912.01412.

[Sloane, 1964] Sloane, N. (1964). The on-line encyclopedia of integer sequences. Disponibil online la https://oeis.org/, accesat decembrie 2019.