Liviu Firca 332

Proiect PS

Introducere

Natural language processing (NLP) consta in a deriva intuitie despre limbajele de zi cu zi, prin a transforma continutul textului in vectori si a procesa datele. Un model foarte dominant este cel de transformer, care a fost introdus in paperul Attention is all you need. Practic, un transformer consta in a aplica un multi-layer perceptron (mlp) de n ori pe acelasi cuvant (token), dar cu caviatul ca inainte sa se aplice mlp, informatia dintre tokenuri este “impartita” intre ele, prin mecanismul de atentie. In primul rand, fiecare vector emite un delta (notat V), si este interpreta ca fiind “cum influenteaza acest token pe altii” , si un vector cheie K, care este interpretat ca fiind un summary la cum ii influenteaza pe altii. Dupa acea, fiecare token emite un vector query Q, care inseamna ce are nevoie acest token, cum se poate modifica. Deci mecanismul de atentie influenteaza continutul tokenilor in acest fel :

$$\mathbf{Softmax}\left(\frac{QK^t}{\sqrt{d}}\right)V$$

Dupa acest calcul (QK^t), pentru fiecare token, ii se maschiaza informatia care vine dupa el, (in cazul in care vrem sa prezicem cuvantul viitor), si se insumeaza toate liniile acestei matrici (matricea softmax . V). La final, pentru tokenul t_i, care are valoarea v_i, pur si simpul ii se adauga acest delta, adica $Sum(A[i])$, unde A este matricea de atentie. Acest delta este foarte important, pentru ca “imparte” informatia intre tokenuri, si mlp care vine dupaia, se aplica doar din token in token, deci informatia se imparte doar in layerul de atentie. Transformerul de astazi este foarte dominant in NLP, pentru ca este foarte usor de paralelizat, operatiile sunt toate logice si usor de explicat si este eficient de antrenat (ie, daca ii se da un vector de n tokenuri, el va da n predictii , care accelereaza antrenarea de n ori mai mult). Totusi , acest layer de atentie poate fi vazut ca scump computational, si de fapt trebuie sa fie: informatia relevanta pentru a schimba un cuvant se poate afla oriunde in propozitie, si din aceasta cauza trebuie sa fie repetat acest produs matricial de fiecare data, in fiecare layer. Acest studiu se intreaba daca nu exista o rearanjare logica a datelor, astfel incat informatia relevanta pentru un token sa fie “aproape” de el, astfel incat sa se trebuiasca doar layere de mapare (mlp sau convolutie). De exemplu, o poza nu are nevoie de mecanism de atentie, pentru ca informatiile relevante pentru un pixel sunt langa el.

Prima idee

Prima idee consta in a schimba informatia noastra intr-o poza, si fiecare pixel reprezinta o activare a unui neuron, lucru destul de comun într-un CNN. Pentru a mapa informatia pe poza, fiecare pixel emite un vector query, la fel ca la un layer de atentie normal, si fiecare token (vector al unui token) emite un vector cheie. Activarea pentru acest pixel cauzata de

token este pur si simplu produsul scalar kq^t . Deci fiecare token produce o imagine de activari, kQ^t, unde Q este toate query-urile de la pixeli.

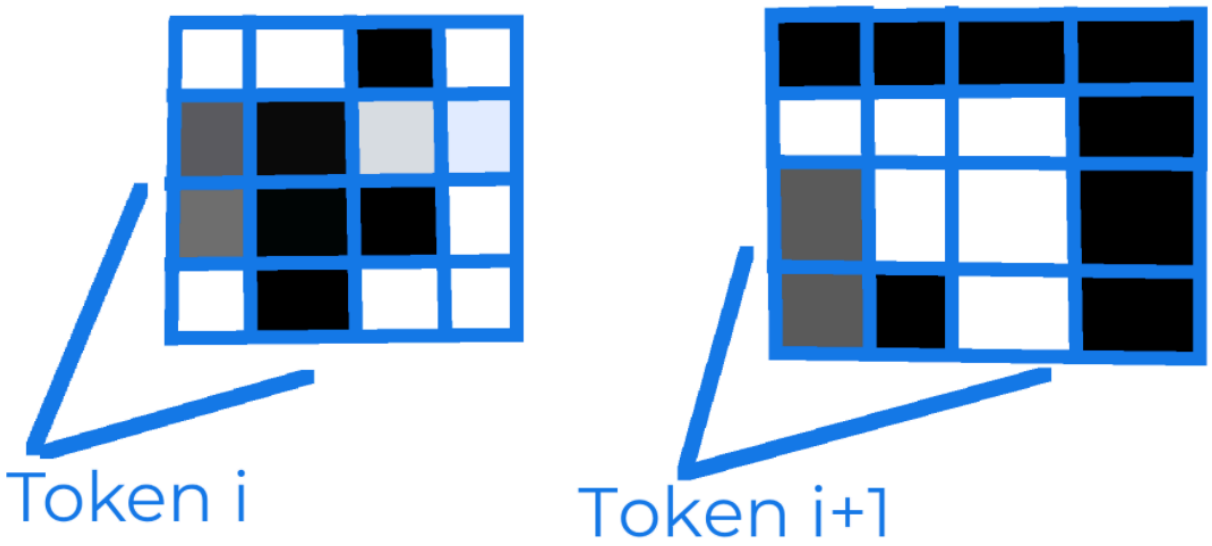


Figura 1: fiecare token activeaza anumite parti din poza

Deci pentru a clarifica : query-ul fiecarui pixel nu se schimba de la token la altul, si fiecare token produce o activare (poza). Aceste activari raman separate pentru pasul urmator. Se prea poate ca un token sa activeze o zona mai mult ca celelalte, de exemplu zona pentru sex, zona pentru concluzie etc.
Este foarte tentant de a rula pur si simplu un CNN pe aceste activari concatenate, sau insumate, dar trebuie sa “masam” datele putin mai mult pentru a garanta un model eficient. Trebuie ca modelul sa produca n predictii pentru un input de lungime n, pentru a avea un model care se antreneaza eficient. Din acest motiv, imaginile de activare produse nu pot sa fie puse la gramada, pentru ca tokenul i sa nu vada informatia de la tokenul i+1. Ne trebuie o operatie care sa imparta informatiile anterioare, si numai cele anterioare. Cea mai usoara operatie posibila este adunarea: act[i+1]+=act[i], unde act este matricea de activare. Deci daca o zona a fost activata in trecut, va fi activata si in prezent. Pentru a simula acest layer, este foarte simplu:

```
For i in range(1,n):
        Act[i] += Act[i-1]
```

In realitate, o sa existe mai multe canale (culori) la aceasta poza, pentru a pareleliza cat mai mult operatiile, si a lasa un cuvant sa influenteze in mai multe moduri aceasta poza.
In sfarsit, se aplca un CNN ales pe fiecare poza de activare, si outputul acestui CNN sunt predictiile. Deci incercam sa rearanjam datele, ca ele sa nu mai trebuiasca sa caute in restul propozitiei, si sa se afle chiar langa ea. Dupa acea, modelul nostru este pur si simplu un CNN.
Arhitectura propusa este urmatoarea :

Positional encoding	Vector embedding

3 layere GPT pentru a incalzi tokenurile

Mapare catre imagine (8 canale)

VGGnet (9 layere, primul de la 8 la 64 de canale si restul de 64)

Daca consultati codul trimis, o sa vedeti ca este implementat un ResNet in loc de VGG, asta este pentru ca libraria pytorch inca mai are multe buguri, si Resnet foloseste operatii putin mai complicate. Deci a trebuit sa schimb cat mai repede codul in compute instance, cand mi-am dat seama ca are buguri pe GPU. Pentru ca tipul de CNN este irelevant pentru acest studiu, nu am stat sa mai modific si codul local.

Antrenare

Pentru antrenare a fost folosit un node de compute, cu 4 rtx4090. Setul de date folosit a fost Openwebtext, care incearca sa copieze setul de date Webtext, folosit pentru antrenarea lui GPT2. Setul de date a fost tokenizat folosit encoderul lui GPT2, care este un byte pair encoder. Block size a fost setat la 1024 tokenuri. Modelul are 50304 tokenuri ca vocabular, deci ne asteptam ca modelul nostru sa aiba un cross entropy loss mai bun decat $ln(50304) \simeq 10.8$ (un model care asigneaza probabilitati egale ar avea acest loss). Lossul in timpul antrenarii a scazut de la 10.8 la 6.46, dupa ce a primit 2 ore de compute. Pentru a pune asta in perspectiva, un GPT cu 12 layere (atentie +MLP) a avut un loss mai mic decat 3.5 pentru acelasi compute.

A doua idee

Problemea cu prima idee este ca avea nevoie de prea mult compute, deca un pixel are un vector, atunci o singura poza de 32*32 cu 8 canale va avea 8 mii de vectori, si asta se va intampla pentru fiecare cuvant. In plus de asta, acesti vectori vor ocupa temporar mult spatiu in memorie, si s-ar putea sa consume prea mult vram. Trebuie sa “extindem” fiecare pixel ca sa minimizam numarul de produse scalare. De exemplu, am putea sa impartim poza in patrate 2*2 si sa lasam produsul scalar sa controleze toata aceasta zona. Dupa aceea trebuie sa concatenam rezultatul, pentru a reconstrui matricea. Totusi, modul cel mai eficient de a face acest lucru este in felul urmator:

- Fiecare poza are o singura activare, si activarea este multiplicata cu restul imaginii
- Poza este generata printr-o aplicatie liniara, de la vectorul token v_i la imaginea act_i.

Din acest lucru se inspira urmatoarea procedure:

- Rulam mecanismul de atentie usual, numai ca fiecare V nu mai reprezinta un delta pentru un token, dar o poza.
- Informatia este transmisa ca in modul usual in atentie, adica : tokenul i nu primeste informatie de la tokenul i+1, in acest fel modelul nostru se antreneaza eficient.

Arhitectura este deci :

Positional encoding	Token embedding

3 layere GPT pentru a incalzi tokenurile

Mapare catre imagine folosing atentie intre imagini (8 canale)

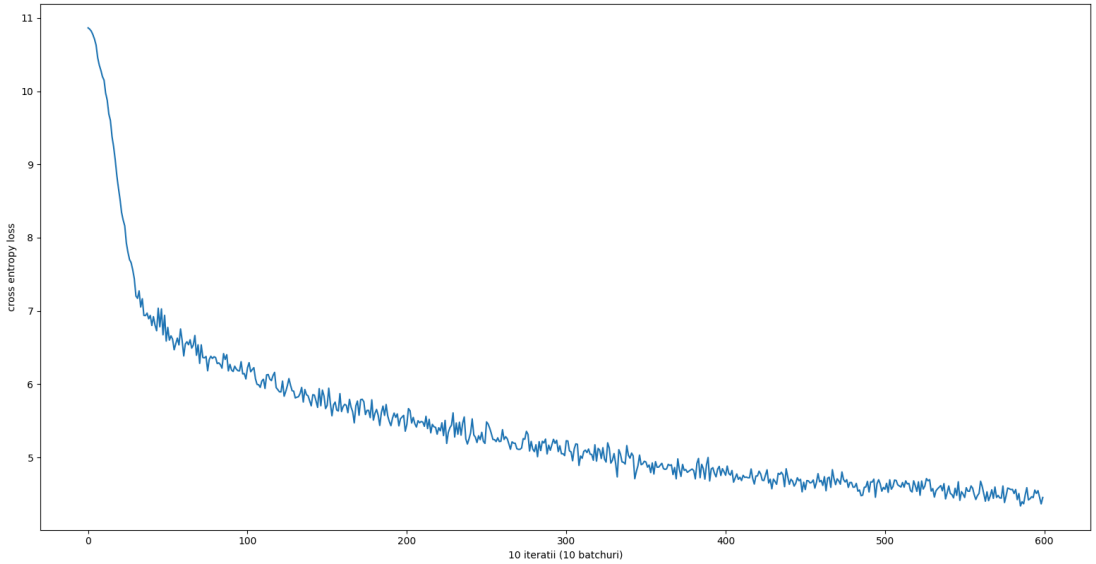
VGGnet (9 layere, primul de la 8 la 64 de canale si restul de 64)

Antrenare

Antrenarea s-a intamplat la fel ca la primul model, numai ca a avut mai mult timp alocat : 3 ore de compute in loc de 2. Pierderea era de 10.8 la inceput si a scazut la 4.73 de data aceasta. M-am gandit ca s-ar putea sa fie prea putine canale pentru vocabular (ultimul layer este o aplicatie liniara de la R^64 la R^50304), deci am facut si o antrenare cu urmatoarea schimbare la reteaua convolutionala :

Layer convolutional de la 8 la 64
3 layere convolutionale de la 64 la 64, si la final unul catre 128 de layere
3 layere convolutionale de la 128 la 128
3 layere convolutionale de la 128 la 128 , si ultimul layer produce 256 canale

Prin urmare, ultimul strat fully connected este o aplicatie liniara de la R^256 la R^50304. A doua oara cand am antrenat modelul, i-am alocat 3 ore si 27 min de compute si a produs acest grafic de pierdere:



Train lossul era de 4.5023 si val loss era de 4.5044, cu 0.2 mai bine decat modelul anterior, (cel cu 64 de canale la final). Totusi acesta este o performanta foarte proasta in comparatie cu un transformer. Ca martor, am antrenat un transformer de 12 layere, timp de 4 ore, si am obtinut un loss de 3.198, deci se pare ca transformari sunt regi inca o data.

Concluzie

Era **foarte** improbabil ca acest model sa fie mai eficient ca un transformer. Toate layerele de convolutie mananca mult vram, si au si ele problemele lor de eficienta. In plus, se poate vedea in repoul git ca ambele modele (prima idee si a doua) au o varianta “chained”, adica se aplica modelul de ML de mai multe ori pe el insusi (la fel ca la un transformer). Nu am stat sa le antrenez si pe astea, pentru ca au avut o performanta prea proasta cele 2 modele, si ar reprezenta prea mult compute pentru ceva mult mai prost decat ce se stie deja in industrie. Intrebarea initiala era daca este posibil sa reorganizam datele, astfel incat sa nu mai trebuiasca sa aplicam mecanismul de antentie, si prin extindere, daca pot convolutiile sau nu sa propage atentia eficient. Concluzia acestui studiu este ca probabil ca nu. Chiar daca acest rezultat a fost negativ, am invatat multe lucruri :

- cum sa fac ssh intrun node de compute
- codul de productie are mereu buguri neprevazute
- O definitie mai eleganta a produsului matricial: Fie doua multimi de vectori de aceasi lungime, produsul matricial reprezinta toate produsele scalare intre cele doua multimi (cu o structura particulara a datelor)
- Mecanismul de atentie reprezinta mai mult un mecanism de taiere decat unul de generare

O sa expandez mai mult ultimul punct pentru ca este cel mai important. Un mecanism de taiere se refera la filtrare : se taie datele nerelevante, pentru a le pastra pe cele importante. In modul in care se fac calculele intr-un transformer, ai putea crede ca mecanismul de atentie genereaza un delta, care schimba continutul vectorului. Dar in realitate se intampla ceva diferit : fiecare cuvant **genereaza** un delta, si mecanismul de atentie le **taie** pe cele

irrelevante (pentru un anumit cuvant). Este o diferenta subtila, dar foarte importanta, pentru ca atentia nu genereaza nimic. Acest lucru este relevant, pentru ca atentia este implicata si in alte modele de ML (unele chiar inainte de transformer), dar si in modul in care functioneaza gandurile noastre. Deci, atentia este un mecanism de a scapa de garbage data, pentru ca modelul este mult mai performant daca primeste doar ce trebuie, pentru a calcula doar ce trebuie. Acest garbage data influentetaza prost modelul.
Pentru ca mecanismul de atentie nu genereaza nimic, s-ar putea ca scopul initial sa fie imposibil : nu poti sa generezi o poza folosind atentia, pentru ca mecanismul nu face asta. Ar trebui un alt mecanism sa faca acest lucru.
Pentru a termina studiul pe o nota buna, am inclus niste sample-uri generate de ultimul model. Este mereu amuzant sa asculti un calculator cum incearca sa vorbeasca. Vectorii reprezenta tokenurile care alcatuiesc propozitia, si 198 este empty token, dat mereu la inceput la LLM. Sample 1 este pentru modelul cu loss de 4.7 (idea 2) si sample2 este pentru modelul cu loss de 4.5 (idea 2, cu mai multi parametrii si compute). Enjoy !