# NLP – Natural Language Processing

* Subdomeniu computer science și AI
* Utilizează MACHINE LEARNING pentru a putea face computerele să înțeleagă și să comunice în limbaj uman

Funcția pipeline()

* Conectează un model cu preprocesarea și postprocesarea
* Putem introduce orice text și să primim un răspuns inteligibil
* ex. : pipeline(„sentiment-analysis”)

Etape pipeline

1. Preprocesare – Convertirea textului în numere cu un tokenizer (tokenizare)
2. Numerele trec prin model = LOGITS
3. Post-procesare = labels + score = output final
4. **Preprocesare**

* Se adaugă tokeni speciali(dacă e cazul)

Ex. Terminatori de șir, CLS, SEP

Apoi se transformă în tokeni:

* Se împarte textul în părți mai mici(tokeni)
* Se mapează fiecare token cu un număr
* Se adaugă input în plus, folositor modelului

Transformers acceptă doar tensori ca input.

Tensor = array în NumPy(scalar, vector, matrice)

Ieșirea modulului Transformer are 3 dimensiuni:

1. **Dimensiunea lotului (batch size)**

* Numărul de secvențe procesat la un moment dat

1. **Lungimea secvențe**

* Lungimea reprezentării numerice

1. **Dimensiunea ascunsă**

* Dimensiunea vectorială a fiecărei intrări

Arhitectura „Transformer” – iunie 2017

1. Pretraining – este necesar un corp de informații, trainingul poate dura câteva săptămână

2. Fine-tuning

- Faza de după pretraining

- Conține un set de date specifice task-ului ce se va implementa

- mai puțin costisitor (date, timp, financiar)

Arhitectura unui transformer

1. Encoder
   1. Bidirecțională
   2. Self-attention

2. Decoder

**Beneficii transformer:**

* Ușor de folosit
* Flexibilitate
* Simplitate
* Un model este compus din două blocuri:

1. Encoder

- preia input și creează o reprezentare a acestuia

2. Decoder

- utizează reprezentarea encoderului pentru a genera secvențele dorite (generează output)

* Există 3 tipuri de modele:

1. Encoder models

2. Decoder models

3. Encoder-decoder models (sequence-to-sequence)

* Attention layer

- layerul semnalează modelului să acorde atenție asupra anumitor cuvinte cheie în momentul în care citește secvențele de date

ex: la traduceri, unele cuvinte de legătură pot dispărea, pot fi incluse în traducerea unui cuvânt sau poate fi necesară introducerea lor

* Arhitectura inițială "transformers"

- inițiali erau folosiți în traduceri / translări

În timpul trainigului se trimitea ca input cu o propoziție și se aștepta output cu aceeași propoziție, tradusă în limba dorită.

În **encoder**, atenție este îndreptată spre toate cuvintele din propoziție (lucrează secvențial) ex: află 3 cuvinte și îl prezice pe al patrulea.

În decoder – primul „Attention layer” din bloc acordă atenție tuturor inputurilor din decoder, dar al doilea utilizează ieșirea din encoder. El folosește toată propoziția pentru a putea prezice cuvântul curent.

**Obs. Este folositor ca în traduceri să se ia în considerare tot contextul.**

* **Arhitectura VS Checkpoints** (puncte de control)

1. Arhitectura

* scheletul unui model
* definiția fiecărui layer și fiecărei operații care se desfășoară fără model

2.Checkpoints

* „weights” adăugate arhitecturii

3. Modelul

* Termen „umbrelă”, care nu este precis ca celelalte două, dar poate însemna ambele

**Modelul encoder**

* Folosesc doar encoderul
* Layerul de atenție accesează toate cuvintele din propoziția inițială
* Au atenție bidirecțională
* Numite și **”auto-encoding models”**

**Pretrainingul** presupune încercarea de a reconstrui o propoziție incompletă.

Este folosit la:

* Înțelegera unor propoziții întregi
* Clasificarea propozițiilor
* Recunoașterea **ner – Named Entity Recognition**
* Răspunsuri la întrebări extractive (extrage răspunsul din secvență)

**Modelul decoder**

* Folosesc doar decoderul
* poate accesa doar cuvinte poziționate înaintea cuvântului la care s-a ajuns in propoziție
* "auto-regresive models"

Folosite la:

* Generarea de text

**Modele encoder-decoder**

* folosește encoder și decoder
* layerul de atenție: la encoder ia toată propoziția; la decoder ia doar cuvintele dinaintea cuvântului dat ca input.

Pretrainingul are aceleași obiective ca modelelel encoder/decoder, uneori mai complex

* ajută la umplerea unor goluri de text

Folosit la:

* generarea de propoziții noi avînd la bază un input dat
* rezumate, translare, răspunde la întrebări generative

***Bias and limitations***

*Unele modele pot returna cuvinte sexiste, rasiste sau supărătoare, motiv pentru care trebuie filtrată informația înainte de a o folosi.*

**Modele**

*Clasa AutoModel*

* se folosește când se dorește ânceperea unui model dintr-un punct de control

**Crearea unui Transformer (ex. Pentru BERT model)**

Ex.

config = BertConfig()

model = BertModel(config)

- model creat cu configurație cu valori random

- trebuie antrenat, însă antrenamentul poate dura foarte mult

-refolosim modele antrenate

*Obs. Configurația conține mai multe detalii folosite la crearea unui model.*

Ca să folosim un model antrenat, folosim metoda **from\_pretrained().**

Ex.

from transformers import BertModel

model = BertModel.from\_pretrained(„bert\_based\_cased”)

În cazul în care codul meu lucrează cu un punct de control, ar trebui să lucreze si cu altul.

Asta se aplică si la arhitecturi diferite atâta timp cât punctul de control a fost antrenat pentru un task asemănător.

Modelul antrenat este initializat cu **weights** punctului de control => poate fi folosit la deducerea sarcinilor pentru care a fost antrenat se poate fi ajustat pentru o sarcina nouă => e mai ușor de gestionat un model preantrenat decât unul antrenat de la 0. Dacă vrem să preantrenăm un model, trebuie trebuie ca punctul de control să facă parte din arhitectura BERT.

* Salvarea modelelor

*model save\_pretrained ("directory\_on\_my\_computer")*

* config json – au date despre arhitectura modelului, atribute, versiuni
* pytouch-model bin - contine "model's weights" = state dictionary

Modelele transformer pot procesa doar numere (generate după tokenizare)

1. Se dau niste secvente (cuvinte)
2. Tokenizers converter cuvintele în indici de vocabular care sunt numiti "input IDS"
3. Fiecare cuvânt are câte o lista de numere (o lista de liste)
4. Convertim matricea într-un tensor

ex: *import torch*

*model\_inputs = torch.tensor (encoded\_sequences)*

output = model (model\_inputs) - folosirea tensorilor ca intrari in model

**Tokenizers**

- printre cele mai importante componente din NLP pipeline

- necesari pentru a transla textele in date ce pot fi procesate de model

- una din probleme apare când avem cuvinte de legătură

**Algoritmi de tokenizare**

1. **Word-based**

Se împarte secvența:

- după spații (split PY) (split)

- după punctuatie etc.

ex: tokenized\_text = "Jim Henson was a puppeteer". split ()

Se scriu în vocabularul modelului toate cuvintele dintr-o limbă. Cuvintele asemanatoare (ex: "dog"si "dogs") vor fi considerate 2 cuvinte diferite, care nu sunt similare.

Alegem si un cuvânt care să reprezinte cuvintele care nu sunt in vocabular => unknown token => [UNK] sau <unk>. Daca apar multe simboluri de acest tip -> antrenamentul a fost incomplet.

**2. Character-based**

- Se transformă textul in caractere

=> vocabular mai mic și mai putine cuvinte "unknown"

**Dezavantaje**

- simbolurile nu au nu au sens de sine statator

**3.Subword tokenization**

Cuvintele folosite mai rar (compuse/ derivate) se descompun în cuvinte de bază (rădăcini) si prefixe/sufixe, care sunt adăugate la vocabular

Ex. Let's </w> do /w> token ization</w> !</w>

Obs. Rezulta mai putine cuvinte in vocabular deoarece multe sufixe / prefixe se repetă.

Pentru incarcare si salvare

1.

from transformers import BertTokenizer

tokenizer = BertTokenizer.from\_pretrained ("bert-base-cased)

au acelasi punct de control ca BERT

SAU

2.

from transformers import AutoTokenizer

tokenizers = AutoTokenizer.from\_pretrained (bert-base-cased) - pot pune orice punct de control.

tokenizer ("Using a Transformer network is simple")

- avem:input\_ids, token-type-ids, attention - mask

Salvam: tokenizer save\_pretrained ("directory\_on\_my\_computer)

1. **Encoding**

* La transfarea din text în numere
* Are 2 pasi

1. Tokenizarea
2. Conversia în input IDs

Etape:

1. Despărțirea textului în cuvinte/ simboluri etc. (tokens)

Obs: Pentru tokenizer se foloseste acelasi tip cu cel al modelului - pentru a fi siguri ca folosim aceleasi reguli cu cele ale modelului preantrenat

2.Convertivea token-ilor în numere, pentru a putea construi tensori.

* tokenizer are un vocabular, acesta este partea caruia îi dăm download când instanțiem metoda " from\_pretrained". Vocabularul trebuie să fie de acelasi tip cu modelul.

|  |  |
| --- | --- |
| Input ID's | [101,2292, 2005, 3046, 2000, 19204, 4697, 906, 10] |
| Special tokens | [[CLS], Let, ’s, try, to, token, ##ize, !, [SEP]] |
| Tokens | [Let, ’s, try, to, token, ##ize, !] |
| Raw text | Let's try to tokenize! |

**1) Tokenizare**

from transformers import AutoTokenizers

tokenizer = AutoTokenizer.from\_pretrained ("bert-base card")

sequence = "Using a Transformer network is simple"

tokens = tokenizer.tokenize (sequence)

print (tokens)

Rezultat: ['Using', 'a', 'transform','##er', network', 'is', 'simple']

* tokenizerul imparte cuvintele până găsește ceva asemanator in vocabularil lui.

Tokens => input IDs

ids = tokenizer.convert\_tokens\_to\_ids (tokens)

1. **Decoding**

* din indici doresc string (metoda "**decode()**")

decoded\_string=tokenizer.decode ([793, 170, 11303..])

* nu doar că face conversia, mai si lipeste "părtile de cuvânt" pentru a putea forma cuvântul dorit.

Obs.

Pentru a putea lucra cu tensori, propozițiile trebuie să aibă aceeasi lungime (când vreau să prelucrez mai multe), deoarece tensorii lucrează doar cu marimi dreptunghiulare.

Pentru a se ajunge la dimensiunile dorite, se adauga un "padding" .

ex:

padding\_id = 100

batched\_id = [200, 200, 200],

[200, 200 padding-id]

* "padding\_id" se gaseste la "tokenizer.pad\_ token\_id"

*Pentru a avea acelasi rezultat atât când incarcam propozitii separat, cât si când le incarcam deodată, folosim layerul de atentie pentru a ignora tokenii de padding.*

* Diferenta apare in momentul in care se contextualizeaza tokenii (se inlocuieste valoarea cu tokenul din propriul vocabular), pe când daca e apelată din tokenizer, au altă valoare.