# Laborator 5: Arhitecturi de tip Transformer

## 1. Introducere

In timp ce arhitecturilor convolutionale au revolutionat aplicatiile axate pe procesarea informatiei vizuale, alte domenii nu pot beneficia in mod direct de acestea, din cauza naturii informatiei prelucrate. Un astfel de domeniu este cel al Prelucrarii Limbajului Natural (*Natural Language Processing* - NLP), unde se pierd legaturile intrinseci dintre pixeli alaturati, speculate de proprietatile operatiei de convolutie.

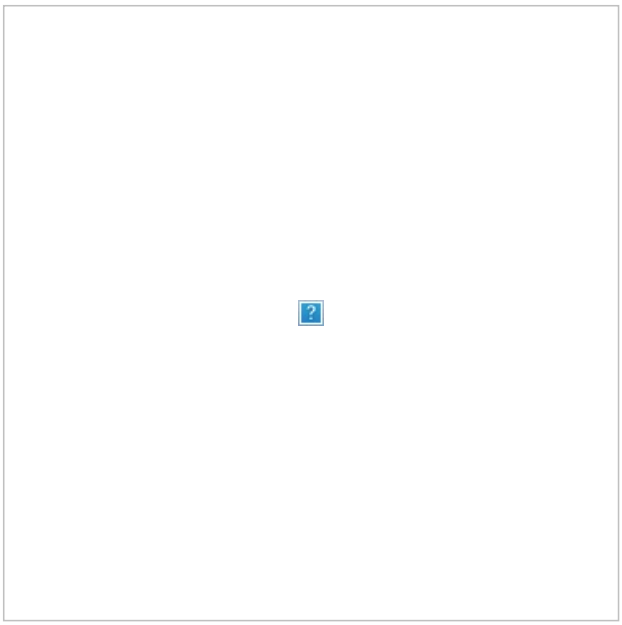
In mod traditional, informatia de tip limbaj natural este procesata utilizand retele recurente, in general sub forma arhitecturilor de tip *Long Short-Term Memory* (LSTM). Aceste solutii sunt avantajoase deoarece sunt proiectate pentru a gestiona intrari de dimensiuni variabile, cum este cazul frazelor, impartite in cuvintele constituente si modelate ca secvente de simboluri de intrare. Pasul echivalent introducerii CNN-urilor in cazul aplicatiilor de limbaj natural a fost reprezentat de aparitia arhitecturilor de tip *Transformer* in anul 2017 [1].

## 2. Structura unui Transformer

### 2.1. Principii fundamentale

In structura generala a unui Transformer se pot delimita doua zone principale de interes:

- o zona de codare a informatiei de intrare (*Encoder*)
- o zone pentru decodare (*Decoder*), in care iesirile primei zone sunt prelucrate pentru a obtine rezultatul dorit.



Structura generala a unei arhitecturi de tip *Transformer*, cu o separare a etajelor de tip Encoder si Decoder. Structurile de tip *Feed Forward* mentionate sunt de tip MLP. Figura preluata din [1].

Dupa cum a fost mentionat anterior, operatiile de tip convolutie 2D nu sunt avantajoase pentru aplicatiile de tip text, stratul cel mai intalnit fiind de tip cel dens conectat. Pe langa utilizarea stratului intr-o maniera asemanatoare celei dintr-un MLP oarecare, inmultirea matriceala este si o parte fundamentala a mecanismului de *atentie* introdus de aceste arhitecturi, ajuns sa fie considerat o operatie standard in bibliotecile moderne de Deep Learning.

### 2.2. Straturile de tip *Attention*

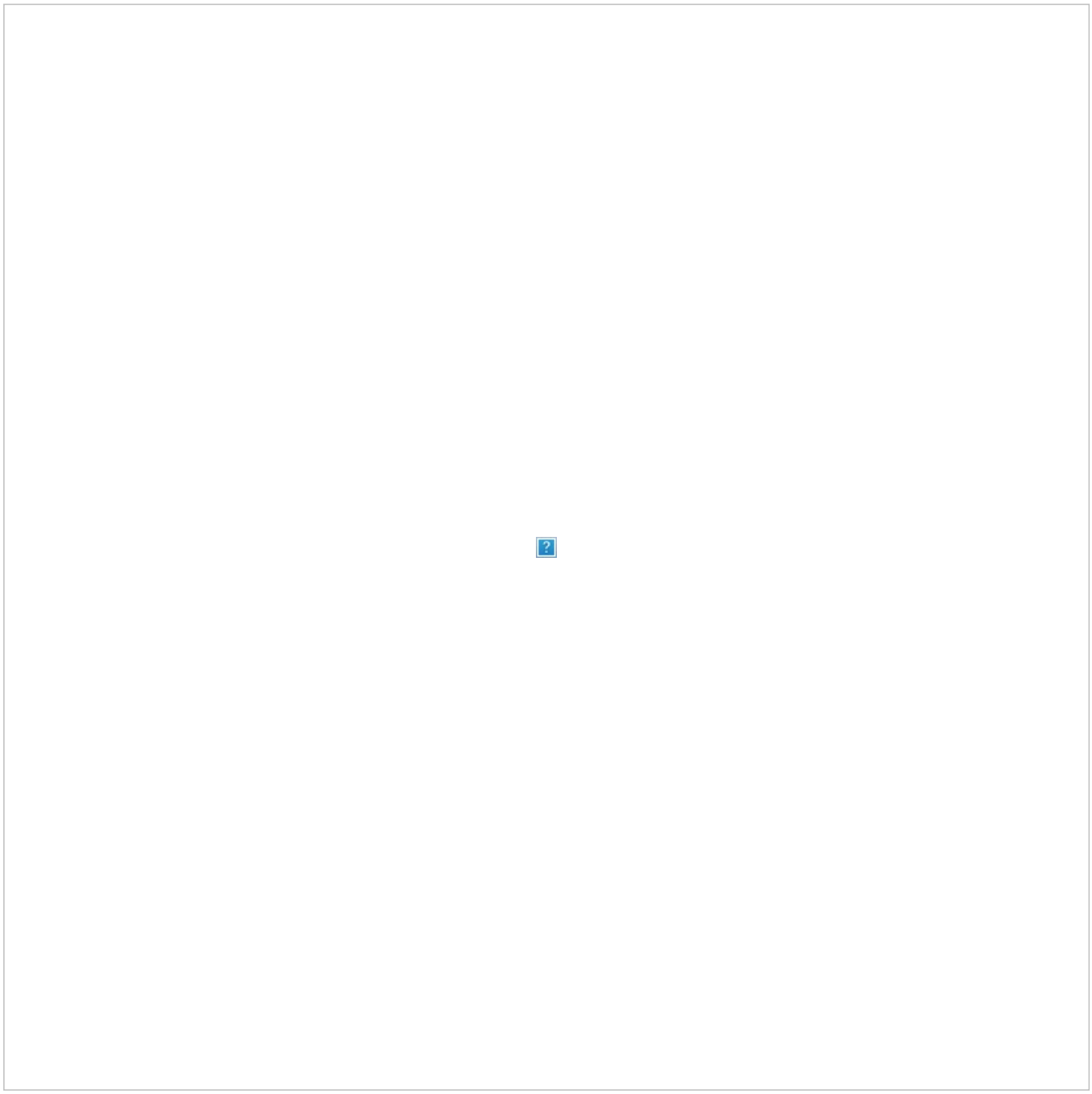
Mecanismele de *atentie* exista in diverse forme de mai multe decenii in literatura de specialitate, desi modelarea lor a suferit transformari puternice de-a lungul timpului. In esenta, scopul ei este de a determina care sunt cele mai importante elemente din secventa procesata in mod curent. Paralela biologica este data de ideea ca oamenii nu prelucreaza in mod simultan toata informatia, ci se concentreaza pe elementele cele mai importante din mesajul receptionat, indiferent de natura acestuia (sunet, imagine, text).

In forma sa actuala, un strat de atentie este proiectat in jurul conceptelor de chei (*Keys* - `K`) si interogari (*Queries* - `Q`) intalnite si in cazul bazelor de date relationale. Scopul final al operatiei este de a determina importanta relativa a valorilor (*Values* - `V`) dintr-o secventa, reprezentate de vectorii din `K` (in general grupati intr-o matrice), raportandu-se la colectia `Q`.

Matematic, `Q`, `K`, `V` sunt vectori calculati prin aplicarea unor straturi liniare asupra informatiei de intrare, pentru a obtine reprezentari noi

ale informatiei. Matricea `Q` este inmultita cu `K`' pentru a determina asemanarea intre reprezentarile de tip *queries* si cele *keys*. Operatia softmax este utilizata pentru a transforma secventele obtinute in probabilitati. Aceste probabilitati se inmultumesc cu matricea de valori, obtinand rezultatul final. Iesirea stratului de atentie este, deci, o colectie de combinatii liniare ale valorilor de intrare, in care ponderile combinatiilor sunt determinate de importanta fiecarui element dintr-o secventa fata de cele ale secventei pentru care se face comparatia.

Daca secventa de reprezentari de intrare pentru `K` si `V` este aceeasi cu cea pentru `Q`, atunci se vorbeste despre operatia de *self-attention*, si este prezenta in zona de *encoder* a unui Transformer. Daca `Q` provine de la alta colectie de intrari, atunci se vorbeste despre *cross-attention*, precum in zona de *decoder*. In mod obisnuit, se presupune ca *self-attention* este utilizat pentru a determina cele mai importante elemente ale unei secvente de intrari, in timp ce *cross-attention* poate fi folosit pentru a vedea legaturile intre colectii diferite de intrari, precum in cazul aplicatiilor de traducere (in care se cauta legaturile dintre cuvinte care provin din limbi diferite).



O vizualizare a operatiei de *self-attention*.
Secventa de 3 elemente (cuvinte in cazul Transformer) este mai intai transformata in reprezentarile Q, K, V. Ponderile legate de prima secventa din Q sunt singurele lasate colorate, pentru evidentiere. Iesirea stratului reprezinta o combinatie liniara a reprezentarilor de tip *Value*.

Mai multe operatii de atentie pot fi aplicate aceleiasi secvente de intrare, pentru a obtine reprezentari mai complexe. Operatia agregata poarta numele de *Multi-Head Attention* si este cea mai utilizata forma in care este intalnita atentia in arhitecturile actuale.

## 2.3. Codarea pozitionala

Solutiile clasice de prelucrare a secventelor de informatii, precum LSTM, sunt construite astfel incat simbolurile de intrare sunt tratate secvential. In cazul Transformer insa, toate elementele sunt prelucrate deodata, fara a se tine cont de ordinea lor de intrare. Din acest motiv,

arhitecturile acestea utilizeaza conceptul de `codare pozitionala` (*positional encoding* sau *positional emedding*) pentru a marca vectorii de intrare, astfel incat sa fie identificabila pozitia lor in secventa totala.

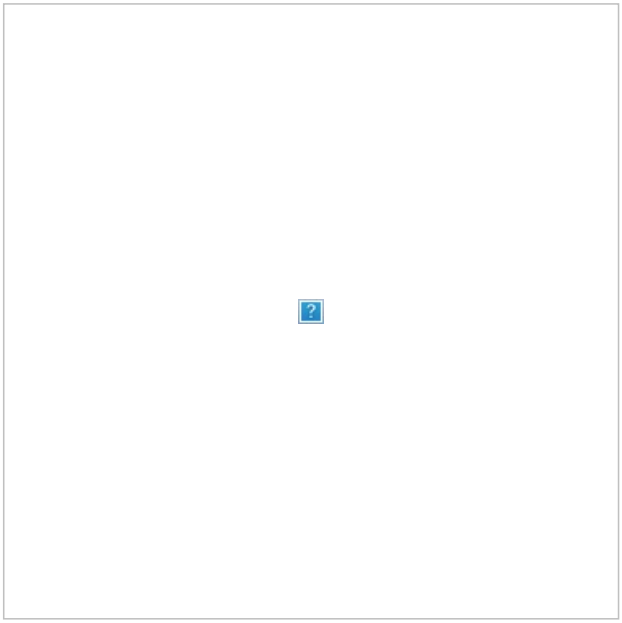
Aceasta marcare suplimentara este realizata prin insumarea simbolurilor de intrare cu un vector a carui valoare este determinata doar de indexul (pozitia) acestuia din secventa de intrare. Exista doua modalitati de a calcula aceste valori suplimentare, prima fiind de a utiliza o formula matematica bazata pe functii trigonometrice. Aceasta varianta este cea folosita in implementarea originala a modelului Transformer. A doua varianta consta in adaugarea unui parametru antrenabil suplimentar, care va fi optimizat odata cu ponderile retelei. Aceasta va fi utilizata in implementarea din lucrarea de fata.

### 2.4. Arhitectura Transformer pentru informatie vizuala

Rezultatele foarte bune ale familiei Transformer a condus la studierea intensa a acestor structuri si aplicarea lor si in alte domenii. Dupa o serie de incercari initiale, *Vision Transformer* (ViT) [2] a fost varianta adoptata uzual pentru sarcinile cu informatie vizuala. Problema principala a unui Transformer obisnuit este data de timpii foarte mari de procesare a imaginilor, fiind structuri de informatie foarte dense.

Solutia ViT a fost de a imparti imaginea intr-o grila cu un numar fix de elemente. Fiecare element al grilei este apoi procesat pentru a obtine o reprezentare intermediara, folosind un strat precum cel liniar, aplicat celulei aplatizate. Colectia de vectori este apoi grupata intr-o matrice si procesata de o structura de tip *encoder* dintr-un Transformer. Iesirea din aceasta zona este considerata ca fiind o reprezentare puternica a spatiului de intrare, precum in cazul iesirii din etajul convolutional al unui CNN.

Dupa iesirea din zona de tip transformer, activarile pot fi procesate obisnuit cu straturi liniare, pentru a obtine raspunsul final al retelei.



Structura generala a unui ViT.
Figura preluata din [2].

In figura de mai sus se poate observa un aspect suplimentar de proiectare a modelului care influenteaza zona de inceput a retelei. Pe pozitia 0 a secventei din figura se adauga un simbol in plus de clasificare. Acest simbol nu contine informatie despre imagine dar, fiindca este prelucrat impreuna cu restul blocurilor, ajunge sa incorporeze contextul global al imaginii de intrare. Atat intrarea cat si iesirea unui Transformer sunt secvente de informatii. Pentru sarcinile de clasificare, care nu necesita iesiri de tip secventa, un astfel de simbol suplimentar este o solutie convenabila fata de alte variante, precum procesarea tuturor blocurilor de iesire concatenate, evitand cresterea efortului computational.

## 3. Implementarea unui Transformer in PyTorch

### 3.1. Pregatirea datelor

Pentru a putea utiliza arhitectura ViT, imaginile de intrare trebuie sa fie rearanjate in structura asteptata de aceasta. Imaginea se imparte in blocuri, care apoi sunt aplatizate. Pentru simplitate, tinand cont ca o imagine din setul MNIST este patrata, se vor genera celule patrate. Metoda noua, regasita mai jos, trebuie adaugata in clasa de tip `Dataset` si apelata in metoda `__getitem__` a ei.

In [ ]:
```
    def image_to_patch(self, img, nr_blocuri=4):
        block_size = int(img.shape[1] // (np.sqrt(nr_blocuri)))
        out_block = np.zeros([nr_blocuri, block_size**2], np.float32)

        current_row = 0
        for i in range(0, img.shape[1], block_size):
            for j in range(0, img.shape[2], block_size):
                out_block[current_row, :] = img[0, i:i+block_size, j:j+block_size].reshape(-1)
```

```
                current_row += 1
        return out_block
```

## 3.2. Clasa modelului

Atat etajul de *encoder* cat si cel *decoder* sunt formate din blocuri repetitive in care sunt prezente operatii de atentie, de transformare liniara, cu straturi dens conectate, si operatii de normalizare. In cazul primei parti a retelei, se folosesc mecanisme de tip *self-attention*. Aceste blocuri pot fi grupate intr-un `nn.Module` separat.

In [ ]:
```
class ViT_Block(nn.Module):
    def __init__(self, nr_neuroni_hidden=128, nr_neuroni_hidden_mlp=128, num_heads=4, p_dropout=0.1):

        super(ViT_Block,self).__init__()

        self.ln1 = nn.LayerNorm(nr_neuroni_hidden, eps=1e-6)
        self.self_attention = nn.MultiheadAttention(nr_neuroni_hidden, num_heads, dropout=p_dropout)

        self.ln2 = nn.LayerNorm(nr_neuroni_hidden, eps=1e-6)

        # MLP
        self.mlp1 = nn.Linear(nr_neuroni_hidden, nr_neuroni_hidden_mlp)
        self.gelu = nn.ReLU()
        self.dropout = nn.Dropout(p_dropout)

        self.mlp2 = nn.Linear(nr_neuroni_hidden_mlp, nr_neuroni_hidden)

    def forward(self, input_batch):
        x = self.ln1(input_batch)
        # In mod implicit, stratul de atentie din PyTorch cere ca dimensiunea corespunzatoare dimensiunii pachetu
        # sa fie a doua. Desi exista un parametru care rezolva aceasta necesitate (batch_first), acesta nu este
        # prezent in toate versiunile de PyTorch, si a fost evitat, pentru compatibilitate.
        x = x.permute(1, 0, 2)
        x, _ = self.self_attention(x, x, x) # Toate 3 intrarile sunt x, deoarece este utilizat self-attention
        x = x.permute(1, 0, 2)
        x = x + input_batch
        y = self.ln2(x)

        x = self.mlp1(y)
        x = self.gelu(x)
        x = self.dropout(x)
        x = self.mlp2(x)

        # Este sarita normalizarea de la final, fiindca a fost introdusa una la inceput.
        out = x + y

        return out
```

In [ ]:
```
from collections import OrderedDict
class ViT(nn.Module):
    def __init__(self, nr_blocuri_vit, nr_neuroni_in, nr_blocuri_imagine, nr_clase, nr_neuroni_hidden=128, nr_neu
        super(ViT,self).__init__()
        self.fc_proj_1 = nn.Linear(nr_neuroni_in, nr_neuroni_hidden) # Pentru o imagine MNIST impartita in 4 bloc

        self.class_token = nn.Parameter(torch.zeros(1, 1, nr_neuroni_hidden)) # Elementul care se adauga la incep
        self.pos_embed = nn.Parameter(torch.randn(1, nr_blocuri_imagine+1, nr_neuroni_hidden)) # Tine cont de sim

        # Inlantuirea blocurilor constituente ViT
        # Pentru a evita bucle in metoda de procesare, se foloseste structura de tip Sequential,
        # care executa automat straturile unei retele in ordinea in care au fost scrise
        blocuri_vit = OrderedDict()
        for i in range(nr_blocuri_vit):
            blocuri_vit[f"bloc_{i}"] = ViT_Block(nr_neuroni_hidden, nr_neuroni_hidden_mlp, num_heads, p_dropout)


        self.blocuri_vit = nn.Sequential(blocuri_vit)
        self.ln = nn.LayerNorm(nr_neuroni_hidden, eps=1e-6)

        # Pentru reducerea numarului de parametri, la iesirea din bloc va fi un singur strat de iesire, in locul
        self.out = nn.Linear(nr_neuroni_hidden, nr_clase)

    def forward(self, input_batch):
        # In prima instanta, simbolul suplimentar de clasificare este repetat pentru a fi utilizat de tot pachetu
        # si adaugat la inceputul secventelor de date, dupa ce sunt prelucrate de primul strat liniar
        class_token = self.class_token.expand(input_batch.shape[0], -1, -1)
        x = self.fc_proj_1(input_batch)
        x = torch.cat([class_token, x], dim=1)

        # Dupa transformarea initiala, este adaugata codarea pozitionala
        x = x + self.pos_embed

        # Partea principala a prelucrarii, prin blocurile de tip Transformer
        x = self.blocuri_vit(x)
        x = self.ln(x)

        # Se pastreaza doar primul element din fiecare secventa, corespunzator
        # simbolului special de clasificare
```

```
        x = x[:, 0]

        # Clasificarea finala
        out = self.out(x)

        return out
```

**!Atentie:** Activarea utilizata in mod obisnuit in ViT este `GELU`, dar a fost pastrat `ReLU` din motive de compatibilitate.

## Exercitiu: Antrenati o arhitectura ViT care sa clasifice cifre, folosind setul MNIST ca sursa de date reale.

Utilizati:

- un singur bloc ViT pentru a accelera antrenarea.
- o impartire a imaginilor din setul MNIST in 4 blocuri.

## Exercitiu: Modificati codul din exercitiul anterior, pentru a schimba numarul de blocuri in care sunt impartite imaginile de intrare.

Bibliografie


1. Vaswani, A. (2017). Attention is all you need. Advances in Neural Information Processing Systems.
2. Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., ... & Houlsby, N. (2020). AN IMAGE IS WORTH 16X16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE. arXiv preprint arXiv:2010.11929.