

Générative pre-training of Transformer pour la génération de prénoms

Louis Thiry, Maître de conférences en mathématiques appliquées
Laboratoire Jacques-Louis Lions, Sorbonne Université

19 octobre 2026

Objectif du cours. Vous avez probablement entendu parler des LLMs (Large Language Models), automates capables de générer du texte et de répondre à des questions. Leur efficacité est surprenante, et la vitesse des progrès dans ce domaine est spectaculaire, à tel point que ces outils, moyennant une puissance de calcul phénoménale, permettent d'avancer sur des conjectures mathématiques majeurs, en exhibant par exemple des solutions de l'équation de Navier-Stokes faisant apparaître des singularités en temps fini. Ils sont tous basés sur un principe simple : la prédiction du prochain token sachant les tokens précédents, et sont tous "pré-entraînés" selon la méthode GPT : "Generative Pre-training of Transformer". Nous allons voir dans ce cours précisément ce qu'est le "Generative Pre-training of Transformer" en se concentrant sur une base de données de texte relativement simple : une base de prénoms. L'algorithme que nous verrons n'est pas spécifique aux prénoms, c'est le même que pour le pré-entraînement de grands modèles de langage. Le code que nous écrirons sera du "pur" Python, c'est à dire du python sans utiliser de bibliothèque de calcul vectoriel comme numpy ou pytorch. En s'y prenant avec soin, le code complet de l'algorithme ne fait pas plus de 300 lignes, tout en étant clair et compréhensible. Nous étudierons l'algorithme brut, et à la fin du cours, vous devriez être capable de réécrire cet algorithme et de l'expliquer.

Notions abordées :

- Algèbre linéaire et probabilités
- Vraisemblance et maximum de vraisemblance
- Feature engineering
- Régression logistique
- Minimisation par descente de gradient stochastique
- Différentiation automatique
- Réseaux de neurones de type Transformers

Table des matières

0	Algèbre linéaire en Python	4
0.1	Notations pour l'algèbre linéaire	4
0.2	Bibliothèques de fonctions de base Python	4
0.3	Représentation des vecteurs	5
0.4	Représentation des matrices	5
0.5	Opérations d'algèbre linéaire sur les vecteurs	6
0.6	Opérations d'algèbre linéaire sur les matrices	6
0.7	Le pivot de Gauss et son application à l'inversion de matrice	6
0.8	Plus grande valeur propre d'une matrice carrée : la méthode des puissances itérées	8
1	Hypothèse de travail et modélisation	9
1.1	Représentation du texte sous forme de vecteurs : les tokens	9
1.2	Hypothèses probabilistes	9
1.3	Théorème de Bayes ou règle des probabilités composées	10
1.4	Modélisation	10
1.4.1	Feature engineering + fonction "simple"	10
1.4.2	Les réseaux de neurones	10
2	Maximum de vraisemblance	11
2.1	La vraisemblance	11
2.2	La log-vraisemblance négative (NLL)	11
2.3	Exemple d'application du maximum de vraisemblance	12
2.4	Cas général	12
3	La régression logistique	12
3.1	Représentation des variables de conditionnement	13
3.2	Un modèle linéaire	13
3.3	La fonction à minimiser : log-vraisemblance négative	14
3.3.1	Pour un token dans un prénom	14
3.3.2	Pour tous les tokens d'un prénom	14
3.3.3	Pour tous les prénoms de la base de données	14
4	Minimisation	15
4.1	Calcul du gradient (loss d'un token)	15
4.1.1	Descente de gradient stochastique	16
4.1.2	L'algorithme ADAM	16
4.2	Convexité de la fonction de coût	17
4.2.1	Pour aller plus loin : méthode de Newton	17
5	Différentiation automatique	18
5.1	Motivation	18
5.2	Cas d'une fonction d'une variable	19
5.3	Cas général : fonctions de d variables	22
5.3.1	Un exemple	22
5.3.2	Règle de la chaîne à plusieurs variables	22
5.3.3	Ordre topologique	23
5.3.4	Coût de la différentiation automatique	26
5.4	Retour sur la régression logistique	26

6	Les transformers	27
6.1	Du feature vector au réseau de neurones	27
6.2	Embedding des tokens et des positions	27
6.3	L'attention	27
6.4	Un tête d'attention	28
6.5	Attention causale	29
6.5.1	Attention multi-têtes	30
6.6	RMSNorm	31
6.7	Le Multi-layer Perceptron (MLP)	32
6.8	Un bloc transformer	32
6.9	L'architecture de type GPT	33
6.10	Génération et KV cache	34
A	Rappels de probabilité	36
A.1	Loi discrète sur un espace fini	36
A.2	Probabilité conditionnelle et règle de Bayes	36
A.3	Indépendance et échantillon i.i.d.	36
A.4	Maximum de vraisemblance et log-vraisemblance	36
B	Rappels de convexité et optimisation convexe	37
B.1	Ensemble et fonction convexes	37
B.2	Caractérisations différentiables	37
B.3	Opérations qui préservent la convexité	37
B.4	Deux fonctions convexes utiles	37
B.5	Pourquoi la convexité aide	37
B.6	Adam comme approximation de la Hessienne	37
B.7	Inégalité de Jensen	38

0 Algèbre linéaire en Python

0.1 Notations pour l'algèbre linéaire

- Nous noterons les vecteurs et matrices en caractères gras, les vecteurs seront en minuscules et les matrices en majuscules. Par exemple, $\mathbf{u}$ est un vecteur de dimension d et $\mathbf{M}$ est une matrice de taille (n, m) .
- Les nombres, entiers ou flottants seront notés en caractère normaux (ni gras ni italique). Par exemple la coordonnée i du vecteur $\mathbf{u}$ sera notée u_i . Le coefficient i, j de la matrice $\mathbf{M}$ sera noté $m_{i,j}$.
- Pour les lignes et colonnes d'une matrice, nous utiliserons la notation $i.$ pour la i -ème ligne et $.j$ pour la j -ème colonne. Par exemple, la i -ème ligne de la matrice $\mathbf{M}$ sera notée $\mathbf{m}_i$. et la j -ème colonne de la matrice $\mathbf{M}$ sera notée $\mathbf{m}_j$. En python, on utilisera les variables `mi_` et `m_j` pour les lignes et colonnes de la matrice `M`, puisque les syntaxes `mi.` et `m.j` ne sont pas valides pour des variables Python.

0.2 Bibliothèques de fonctions de base Python

On n'utilise que deux bibliothèques de la librairie standard : `math` (fonctions mathématiques) et `random` (tirages aléatoires).

```
import math
import random

# listes et tuples en python
l = [1, 2, 3] # mutable, on peut modifier les elements
t = (1, 2, 3) # immutable, on ne peut pas modifier les elements
l[0] = 4 # valide
t[0] = 4 # invalide, raise une erreur

# attention ajouter deux listes les concatener en python
[1,2] + [3,4]
[1, 2, 3, 4]

# calcul de l'exponentielle et le logarithme de a
a = 1
exp_a, log_a = math.exp(a), math.log(a)
exp_a, log_a
(2.718281828459045, 0.0)

# comprehension de liste pour creer une nouvelle liste
v = [1, 2, 3]
log_v = [math.log(vi) for vi in v]
log_v
[0.0, 0.6931471805599453, 1.0986122886681098]

# zip(v1, v2 (, v3)) pour créer la liste des couples (triplets) (v1i, v2i(, v3i))
v1 = [1, 2, 3]
v2 = [4, 5, 6]
v3 = [7, 8, 9]
# couples
v1v2_tuples = list(zip(v1, v2))
v1v2_tuples
[(1, 4), (2, 5), (3, 6)]
# triplets
v1v2v3_tuples = list(zip(v1, v2, v3))
v1v2v3_tuples
```

```

[(1, 4, 7), (2, 5, 8), (3, 6, 9)]

# l'étoile pour unpacker les listes et les donner en argument à une fonction
v1, v2 = zip(*v1v2_tuples)
v1
(1, 2, 3)
v2
(4, 5, 6)

# tirage d'une v.a. gaussienne de moyenne 1 et d'écart type 2
random.seed(42) # reproductibilité des tirages aléatoires
mean, std = 1, 2
x = random.gauss(mean, std)
x
0.7118193408441433

```

0.3 Représentation des vecteurs

On représente un **vecteur** de dimension d comme une simple liste de d nombres flottants (float en Python).

On se dotera des fonctions suivantes, permettant de créer un vecteur nul, un vecteur aléatoire et un vecteur de la base canonique, aussi appelé one-hot vector :

- `zero_vec(d)` qui renvoie un vecteur nul de dimension d .
- `rand_vec(d, std)` qui renvoie un vecteur aléatoire de dimension d dont les coordonnées sont des variables aléatoires gaussiennes centrées d'écart type `std`.
- `one_hot(d, i)` qui renvoie un vecteur de dimension d dont la i -ème coordonnée est égale à 1 et les autres sont égales à 0. En maths, on appelle ce vecteur le i -ème vecteur de la base canonique de $\mathbb{R}^d$.

Solution.

```

zero_vec = lambda d: [0. for _ in range(d)]
rand_vec = lambda d, std=1: [random.gauss(0, std) for _ in range(d)]
one_hot = lambda d, i: [float(j == i) for j in range(d)]

```

0.4 Représentation des matrices

Une **matrice** est vue comme la liste de ses vecteurs *lignes* : une matrice à n lignes et m colonnes est une liste de n vecteurs de dimension m .

Exercice. Écrire les fonctions suivantes :

- `zero_mat(n, m)` qui renvoie une matrice nulle à n lignes et m colonnes.
- `rand_mat(n, m, std)` qui renvoie une matrice aléatoire à n lignes et m colonnes.
- `identity(n)` qui renvoie la matrice identité de taille $n \times n$.

Solution.

```

zero_mat = lambda n, m: [[0. for _ in range(m)] for _ in range(n)]
rand_mat = lambda n, m, std=1: [rand_vec(m, std) for _ in range(n)]
identity = lambda n: [[float(i == j) for j in range(n)] for i in range(n)]

```

0.5 Opérations d'algèbre linéaire sur les vecteurs

Dans tout ce paragraphe :

- $\mathbf{u} = (u_1, u_2, \dots, u_d)$ et $\mathbf{v} = (v_1, v_2, \dots, v_d)$ sont deux vecteurs de même dimension d
 - α est un scalaire
 - $\mathbf{M}$ et $\mathbf{N}$ est une matrice de tailles respectives (m, d) et (d, n)
- Un $\mathbb{R}$ -espace vectoriel normé Euclidien E de dimension finie d est doté :
- d'une loi interne $+$: $(\mathbf{u}, \mathbf{v}) \in E \times E \mapsto \mathbf{u} + \mathbf{v} \in E$
 - d'une loi externe : $(\alpha, \mathbf{u}) \in \mathbb{R} \times E \mapsto \alpha \mathbf{u} \in E$
 - d'un produit scalaire : $(\mathbf{u}, \mathbf{v}) \in E \times E \mapsto \langle \mathbf{u}, \mathbf{v} \rangle \in \mathbb{R}$
 - d'une norme : $\mathbf{u} \in E \mapsto \|\mathbf{u}\| \in \mathbb{R}$

Voici comment les implémenter en Python :

```
add_vec = lambda u, v: [ui + vi for ui, vi in zip(u, v)]
mul_vec = lambda alpha, u: [alpha * ui for ui in u]
dot_prod = lambda u, v: sum(ui * vi for ui, vi in zip(u, v))
norm = lambda u: math.sqrt(dot_prod(u, u))
```

0.6 Opérations d'algèbre linéaire sur les matrices

Les applications linéaires entre deux espaces vectoriels normés Euclidiens de dimension finie m et n sont représentées par des matrices de taille (n, m) . Il nous faut donc implémenter les opérations suivantes :

- le produit matrice-vecteur
- la transposée d'une matrice
- le produit matrice-matrice

En python :

```
mat_vec = lambda M, v: [dot_prod(mi_, v) for mi_ in M]
transpose = lambda M: [list(m_j) for m_j in zip(*M)] # zip(*M) renvoie une list de
tuples que l'on convertit en listes
mat_mat = lambda A, B: [[dot_prod(ai_, b_j) for b_j in zip(*B)] for ai_ in A]
```

0.7 Le pivot de Gauss et son application à l'inversion de matrice

Principe. Le pivot de Gauss transforme une matrice $\mathbf{M}$ de taille (n, m) quelconque en sa forme dite échelonnée réduite. Cette transformation est obtenue par opérations élémentaires sur les lignes :

1. échanger deux lignes,
2. multiplier une ligne par un scalaire,
3. ajouter à une ligne un multiple d'une autre.

L'algorithme se déroule en deux phases :

1. **Échelonnage (ou descente).** On suppose que la première colonne de la matrice a au moins un coefficient non nul. On choisit le pivot comme le plus grand coefficient en valeur absolue dans cette première colonne. On échange donc la première ligne avec la ligne contenant le pivot. En utilisant l'opération élémentaire 3, on annule les coefficients de la première colonne sous le premier pivot. Puis on recommence avec la deuxième colonne : si tous ses coefficients sont nuls, on passe à la troisième colonne. Sinon on échange les lignes pour que le deuxième coefficient de la deuxième ligne soit non nul. Et on itère le processus.
2. **Réduction (ou remontée).** En remontant les pivots, on normalise chaque pivot à 1 puis on annule les coefficients au-dessus des pivots en utilisant l'opération élémentaire 3.

A la fin de l'algorithme, la matrice est échelonnée réduite. Le nombre de pivots est le rang de $\mathbf{M}$. Si la matrice M est carrée et que l'on veut son inverse, on peut appliquer l'algorithme à la matrice augmentée $[\mathbf{M} \mid \mathbf{I}]$. A la fin de l'algorithme, la partie droite de la matrice augmentée est l'inverse de $\mathbf{M}$, si $\mathbf{M}$ est inversible, sinon c'est sa pseudo-inverse.

Exercice. Écrire une fonction `echelonner(M)` qui effectue la phase de descente et renvoie la matrice échelonnée ainsi que la liste des pivots (couples (ligne, colonne)), puis une fonction `reduire(M, pivots)` qui effectue la phase de remontée. Ecrire la fonction `pivot_gauss(M)` qui enchaîne les deux phases. Utiliser `pivot_gauss` pour écrire la fonction `inverse(A)` qui renvoie l'inverse de la matrice carrée $\mathbf{A}$. Ecrire la fonction `rank(M)` qui renvoie le rang de la matrice $\mathbf{M}$.

Solution.

```
def echelonner(M, eps=1e-12):
    """ Les coefficients la matrice inferieurs a eps sont consideres comme nuls. """
    n, m = len(M), len(M[0])
    pivots, row = [], 0
    for col in range(m):
        if row == n:
            break
        # ligne du pivot: cofficient le plus grand en valeur absolue
        p = max(range(row, n), key=lambda i: abs(M[i][col]))
        if abs(M[p][col]) < eps:
            # colonne de zero (ou presque zeros), pas de pivot
            continue
        # on echange la ligne actuelle et celle du pivot
        echanger_lignes(M, row, p)
        M[row], M[p] = M[p], M[row]
        for i in range(row + 1, n):
            # on annule les elements de la colonne en dessous du pivot
            M[i] = sum_vec(M[i], ext_prod(-M[i][col] / M[row][col], M[row]))
        pivots.append((row, col))
        row += 1
    return M, pivots

def reduire(M, pivots):
    for row, col in reversed(pivots):
        M[row] = ext_prod(1 / M[row][col], M[row])
        for i in range(row):
            # on annule les éléments de la colonne au dessus du pivot
            M[i] = sum_vec(M[i], ext_prod(-M[i][col], M[row]))
    return M

def pivot_gauss(M):
    return reduire(*echelonner(M))

def inverse(A):
    n = len(A)
    # matrice augmentée (A/Id)
    A_Id = [list(ai_) + [float(i == j) for j in range(n)] for i, ai_ in enumerate(A)]
    M = pivot_gauss(A_Id)
    A_inv = [mi_[n:] for mi_ in M]
    return A_inv

def rank(M):
    _, pivots = echelonner(M)
    return len(pivots)
```

0.8 Plus grande valeur propre d'une matrice carrée : la méthode des puissances itérées

Principe. On part d'un vecteur aléatoire $\mathbf{v}$ de norme 1 et on répète $\mathbf{v} \leftarrow \mathbf{A}\mathbf{v}/\|\mathbf{A}\mathbf{v}\|$. Chaque multiplication par $\mathbf{A}$ amplifie la composante de $\mathbf{v}$ sur le vecteur propre de plus grand module, donc $\mathbf{v}$ converge vers ce vecteur propre, à vitesse $|\lambda_2/\lambda_1|^k$. La valeur propre est estimée par le quotient de Rayleigh $\lambda \approx \langle \mathbf{v}, \mathbf{A}\mathbf{v} \rangle$. Hypothèses : la valeur propre dominante est unique (en module), et le vecteur initial n'est pas orthogonal au vecteur propre dominant (vrai presque sûrement pour un tirage gaussien).

Exercice. Écrire une fonction `power_iteration(A, n_iter)` qui renvoie une approximation de la plus grande valeur propre (en module) de $\mathbf{A}$ et du vecteur propre associé.

Solution.

```
def power_iteration(A, max_iter=100, tol=1e-8):
    v = rand_vec(len(A))
    v = ext_prod(1 / norm(v), v)          # vecteur initial de norme
    1
    for _ in range(max_iter):
        v_new = mat_vec(A, v)
        v_new = ext_prod(1 / norm(v_new), v_new)    # normalisation du vecteur
        propre
        diff_norm = norm(sum_vec(v_new, ext_prod(-1, v)))
        v = v_new
        if diff_norm < tol:
            break
    return dot_prod(v, mat_vec(A, v)), v      # (valeur propre, vecteur
        propre)
```

En pratique, on utilise `numpy.linalg.eigh` (matrices symétriques) ou `numpy.linalg.eig`, qui calculent toutes les valeurs propres par l'algorithme QR de LAPACK. La méthode des puissances reste utile quand on ne veut que la plus grande valeur propre d'une très grande matrice, car elle n'exige que des produits matrice-vecteur.

On construit une matrice symétrique définie positive $\mathbf{A} = \mathbf{B}^\top \mathbf{B} + \mathbf{I}$ (la méthode des puissances y est fiable), et on vérifie les trois algorithmes.

```
random.seed(0)
d = 5
B = rand_mat(d, d)
BtB = mat_mat(transpose(B), B)
A = [[BtB[i][j] + (i == j) for j in range(d)] for i in range(d)]
b = rand_vec(d)

# pivot de Gauss : A A^{-1} doit etre proche de l'identite
M = pivot_gauss([list(ai_) + [float(i == j) for j in range(d)] for i, ai_ in enumerate
(A)])
A_inv = [mi_[d:] for mi_ in M]
mat_mat(A, A_inv)

# LU : le residu A x - b doit etre proche de 0
perm, L, U = lu(A)
x = lu_solve(perm, L, U, b)
norm(sum_vec(mat_vec(A, x), ext_prod(-1, b)))

# puissances iterees : A v doit etre proche de lambda v
lam, v = power_iteration(A)
norm(sum_vec(mat_vec(A, v), ext_prod(-lam, v)))
```

1 Hypothèse de travail et modélisation

1.1 Représentation du texte sous forme de vecteurs : les tokens

Définition 1. Un token est un morceau unitaire de texte pour un LLM.

On découpe le texte en tokens. Pour notre base de données de prénoms, on choisit comme tokens les caractères de l'alphabet plus un caractère spécial de *stop* (début/fin de mot), soit

$$T = 26 + 1 = 27 \text{ valeurs.}$$

On attribue à chacun un numéro :

$$\mathbf{a} \rightarrow 0, \quad \mathbf{b} \rightarrow 1, \quad \dots, \quad \mathbf{z} \rightarrow 25, \quad \mathbf{stop} \rightarrow 26.$$

Un prénom à n lettres devient une suite de $n+2$ tokens $= (t_0, t_1, t_2, \dots, t_{n+1})$, puisque l'on ajoute deux tokens de stop au début et à la fin du prénom. Par exemple :

$$\begin{aligned} \mathbf{anna} &= (t_0=26, t_1=0, t_2=13, t_3=13, t_4=0, t_5=26), \\ \mathbf{max} &= (t_0=26, t_1=12, t_2=0, t_3=21, t_4=26). \end{aligned}$$

On dit que le prénom est "tokenisé" (tokenized en anglais).

Définition 2. Le *one-hot encoding* d'un token t est le vecteur de dimension T qui vaut 1 en position t et 0 partout ailleurs.

Une fois ce prénom de n lettres tokenisé, on transforme les tokens en une suite de $n + 2$ vecteurs one-hot de dimension T , que l'on peut voir comme une matrice de taille $(n + 2, T)$.

Définition 3. Dans une telle représentation de taille $(n + 2, T)$, $n + 2$ est la *sequence length* et T est la *vocabulary size*.

1.2 Hypothèses probabilistes

On fait essentiellement deux hypothèses probabilistes :

- On suppose que les prénoms sont des échantillons **indépendants et identiquement distribués (i.i.d.)** d'une loi de probabilité sur les prénoms. Ainsi la base des Q prénoms vérifie

$$\mathbb{P}(\text{prénom}_1, \dots, \text{prénom}_Q) = \prod_{q=1}^Q \mathbb{P}(\text{prénom}_q).$$

- Puisqu'un prénom est représenté par ses tokens, la probabilité du prénom q à n_q lettres s'écrit comme la probabilité jointe sur les $n_q + 2$ tokens du prénom :

$$\mathbb{P}(\text{prénom}_q) = \mathbb{P}(t_0, t_1, t_2, \dots, t_{n_q+1})$$

Si l'on introduit les variables aléatoires $T_0, T_1, T_2, \dots, T_{n_q+1}$ pour les tokens du prénom q , la probabilité du prénom q s'écrit comme l'intersection sur les événements $\{T_i = t_i\}_{i=0}^{n_q+1}$:

$$\mathbb{P}(\text{prénom}_q) = \mathbb{P}(\{T_0 = t_0\} \cap \{T_1 = t_1\} \cap \{T_2 = t_2\} \cap \dots \cap \{T_{n_q+1} = t_{n_q+1}\})$$

Cette dernière notation étant plus fastidieuse, on utilisera $\mathbb{P}(t_0, t_1, t_2, \dots, t_{n_q+1})$.

Remarque 1. C'est une modélisation peu naturelle. Quand on pense aux prénoms, on pense à l'étymologie, aux versions masculines et féminines, aux variations d'orthographe,... On aurait envie d'inclure ces informations à priori lorsqu'on modélise les prénoms.

Remarque 2. C'est une loi de probabilité **discrète** sur un espace d'états **fini**. Il « suffit » donc de connaître la probabilité de chacune de ces valeurs finies. Avec $n=10$ lettres maximum par prénom et $T=27$, on a $27^{10} = 205,891,132,094,649$ configurations possibles. Il est clairement impossible de représenter directement cette loi de probabilité discrète sur un ordinateur.

1.3 Théorème de Bayes ou règle des probabilités composées

On factorise la loi jointe sur les tokens une avec la règle des probabilités composées. Pour deux tokens, on a :

$$\mathbb{P}(t_0, t_1) = p(t_0) p(t_1 | t_0)$$

Et pour $n + 1$ tokens, on a :

$$\begin{aligned} \mathbb{P}(t_0, t_1, \dots, t_n) &= \mathbb{P}(t_0) \mathbb{P}(t_1 | t_0) \mathbb{P}(t_2 | t_0, t_1) \cdots \mathbb{P}(t_n | t_0, \dots, t_{n-1}) \\ &= \prod_{l=0}^n \mathbb{P}(t_l | t_0, \dots, t_{l-1}). \end{aligned}$$

Le problème se ramène donc à trouver la loi conditionnelle du token l sachant les précédents (token 0 à $l - 1$).

- **Avantage** : il y a T valeurs possibles pour le token l : la sortie est n'est qu'un vecteur de T probabilités (et non T^{n+1}).
- **Inconvénient** : il y a $n + 1$ probabilités conditionnelles à calculer par prénom (*no free lunch, as Milton Friedman used to say...*).

1.4 Modélisation

Pour représenter numériquement cette loi conditionnelle, on choisit une classe de fonctions paramétriques p_θ pour représenter la probabilité conditionnelle du prochain token sachant les précédents. Pour construire cette fonction paramétrique, il y deux grandes approches en machine learning.

1.4.1 Feature engineering + fonction "simple"

On construit de façon déterministe une représentation vectorielle $\phi(t_0, \dots, t_{j-1})$ explicite à partir des tokens précédents. On l'appelle un **feature vector** dans le jargon. On utilise ensuite une régression logistique multinomiale/multivariée pour modéliser la probabilité conditionnelle du prochain token sachant ce feature vector. Ce feature vector doit contenir les informations "pertinentes" des tokens précédents pour la prédiction du prochain token.

1.4.2 Les réseaux de neurones

On définit directement une classe de fonctions paramétriques p_θ par composition de fonctions linéaires et non linéaires. Une fonction de cet espace de fonction est appelé un **réseau de neurones**, et les paramètres de cette fonction sont appelées les **poids (weights)** du réseau de neurones. Ces réseaux de neurones sont composés d'un certain nombre de **couches**, chaque couche étant un bloc élémentaire, par exemple une opération linéaire suivie d'une non-linéarité. On dit qu'un réseau de neurones est **profond (deep)** si il a un grand nombre de couches (how deep is deep ? that is the question...). Une classe particulière de réseaux de neurones s'est imposée comme le choix de référence pour la génération de texte : les **transformers**. Nous détaillerons cette classe de fonctions dans la suite du cours.

Nous verrons les deux approches dans la suite du cours :

- La methode simple ou nous construisons un feature vector explicite à partir des tokens précédents et nous verrons la régression logistique multinomiale.
- La methode des réseaux de neurones ou nous verrons les transformers.

2 Maximum de vraisemblance

Jusqu'ici nous utilisons la notation $\mathbb{P}$, c'est à dire la "vraie" loi de probabilité. Nous allons nous doter maintenant d'une famille de fonctions paramétriques

$$\{p_\theta, \theta \in \mathbb{R}^p\}$$

pour représenter la probabilité conditionnelle du prochain token sachant les précédents. p_θ est une fonction des tokens $t_0, \dots, t_{l-1}$, et est à valeurs dans $\mathbb{R}^T$, avec comme contraintes la positivité des coordonnées et la somme des coordonnées égale à 1. Au mieux, p_θ sera une approximation de la vraie loi conditionnelle $\mathbb{P}(t_l | t_0, \dots, t_{l-1})$.

Lorsqu'on a choisi une classe de fonctions paramétriques p_θ , il faut :

- définir un critère de sélection du paramètre optimal θ^* : fonction à optimiser.
- choisir une procédure pour trouver ce paramètre optimal θ^* (ou une approximation de ce paramètre) : méthode d'optimisation.

2.1 La vraisemblance

Définition 4. La *vraisemblance* d'un modèle est la probabilité d'observer les données sachant le modèle.

Sur notre exemple d'un jeu de données de Q prénoms, la vraisemblance vaut :

$$p_\theta(\text{prénom}_1, \dots, \text{prénom}_Q).$$

Par indépendance puis par la règle des probabilités composées,

$$p_\theta(\text{prénom}_1, \dots, \text{prénom}_Q) = \prod_{i=1}^Q p_\theta(\text{prénom}_i) = \prod_{i=1}^Q \prod_{j=0}^{n_i} p_\theta(t_j^{(i)} | t_0^{(i)}, \dots, t_{j-1}^{(i)}).$$

Le principe de maximum de vraisemblance s'écrit donc :

$$\theta^* = \prod_{i=1}^Q \prod_{j=0}^{n_i} p_\theta(t_j^{(i)} | t_0^{(i)}, \dots, t_{j-1}^{(i)}).$$

Sur une base de données de 10000 prénoms, ce produit de probabilités est un nombre trop petit pour être représentable en machine. Le critère que nous allons utiliser est la log-vraisemblance négative (NLL).

2.2 La log-vraisemblance négative (NLL)

Définition 5. La *log-vraisemblance négative* (*negative log-likelihood, NLL*) d'un modèle est égale à moins log de la vraisemblance.

Proposition 1. Maximiser la vraisemblance équivaut à minimiser la log-vraisemblance négative :

$$\theta^* = \arg \max_{\theta} p_\theta(\text{prénom}_1, \dots, \text{prénom}_Q) = \arg \min_{\theta} -\log p_\theta(\text{prénom}_1, \dots, \text{prénom}_Q).$$

Démonstration. $x \mapsto -\log(x)$ est une fonction décroissante. □

Le principe de maximum de vraisemblance s'écrit donc :

$$\begin{aligned} \theta^* &= \arg \max_{\theta} \mathcal{L}(\theta) \\ \mathcal{L}(\theta) &= \arg \min_{\theta} - \sum_{i=1}^Q \sum_j \log p_\theta(t_j^{(i)} | t_0^{(i)}, \dots, t_{j-1}^{(i)}). \end{aligned}$$

2.3 Exemple d'application du maximum de vraisemblance

Prenons une pièce de monnaie. On modélise la probabilité de tomber sur pile ou face par une loi binomiale.

$$p_\theta(x) = \theta^x(1 - \theta)^{1-x}$$

où x est une variable binaire qui vaut 1 si on tombe sur pile et 0 si on tombe sur face.

La vraisemblance vaut

$$\prod_{i=1}^Q p_\theta(x_i) = \prod_{i=1}^Q \theta^{x_i}(1 - \theta)^{1-x_i}.$$

C'est une fonction continue strictement positive sur $]0, 1[$, et qui s'annule en 0 et 1, qui atteint donc son maximum en un point de $]0, 1[$.

Pour trouver ce point, on peut calculer la dérivée de la log-vraisemblance négative et chercher le paramètre θ où elle s'annule. La log-vraisemblance négative vaut

$$\mathcal{L}(\theta) = - \sum_{i=1}^Q (x_i \log \theta + (1 - x_i) \log(1 - \theta))$$

et sa dérivée est

$$\mathcal{L}'(\theta) = - \sum_{i=1}^Q \left(x_i \frac{1}{\theta} + (1 - x_i) \frac{-1}{1 - \theta} \right)$$

En ayant lancé la pièce Q fois, on obtient k piles et $Q - k$ faces, de sorte que $\sum_{i=1}^Q x_i = k$ et $\sum_{i=1}^Q (1 - x_i) = Q - k$. La dérivée s'écrit donc

$$\mathcal{L}'(\theta) = -\frac{k}{\theta} + \frac{Q - k}{1 - \theta} = -\frac{k - Q\theta}{\theta(1 - \theta)}$$

et s'annule en $\theta = \frac{k}{Q}$. On a donc trouvé la loi de probabilité de la pièce qui maximise la vraisemblance des données.

2.4 Cas général

Dans le cas général : la log-vraisemblance négative est une fonction de $\mathbb{R}^p$ dans $\mathbb{R}$ si on a p paramètres.

- la log-vraisemblance négative est une fonction de $\mathbb{R}^p$ dans $\mathbb{R}$ si on a p paramètres.
- nous utiliserons uniquement des méthodes de descente de gradient pour optimiser cette fonction de coût, c.a.d. que nous calculerons

$$\nabla_\theta \mathcal{L}(\theta)$$

et nous mettrons à jour les paramètres de la façon suivante :

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta)$$

où η est un pas de la descente de gradient qui varie possiblement au cours des itérations.

3 La régression logistique

Comme mentionné précédemment, la méthode dite classique pour la génération de texte était la **régression logistique multinomiale** sur un feature vector, une représentation vectorielle explicite des tokens précédents. Nous allons détailler ici un choix de feature vector, puis la régression logistique multinomiale sur ce feature vector.

3.1 Représentation des variables de conditionnement

Il nous faut choisir une représentation vectorielle simple (on ne détaillera pas les techniques sophistiquées ici). On convertit chaque token t_k en **vecteur one-hot** de dimension T (le vecteur nul partout sauf un 1 en position t_k), noté $\mathbf{1}_{t_k} \in \mathbb{R}^T$. On résume le contexte $(t_0, \dots, t_{j-1})$ par la **moyenne** de ces vecteurs :

$$\mathbf{v}(t_0, \dots, t_{j-1}) = \frac{1}{j} \sum_{k=0}^{j-1} \mathbf{1}_{t_k} \in \mathbb{R}^T.$$

La coordonnée m du vecteur v est simplement la fréquence du token m dans le contexte.

Exercice. Implémenter la fonction v ci-dessus en Python en utilisant les fonctions d'algèbre linéaire définies dans la section précédente.

Solution.

```
tokens, T = [26, 0, 13, 13, 0, 26], 27 # prenom 'anna' tokenise
def feature_vector(tokens, T):
    sum_tokens = [0] * T
    for token in tokens:
        sum_tokens[token] += 1
    # sum_tokens = sum_vec(sum_tokens, one_hot(token, T)) # other way to do it
    return ext_prod(1 / len(tokens), sum_tokens)
```

3.2 Un modèle linéaire

Ce qu'il nous faut. Une fonction de $\mathbb{R}^T$ dans $\mathbb{R}^T$, sous la contrainte que la sortie soit un **vecteur de probabilités** (coordonnées ≥ 0 , de somme 1).

Une régression linéaire ? On introduit une matrice $\mathbf{M} \in \mathbb{R}^{T \times T}$ et on regarde $\mathbf{M}\mathbf{v}$, un vecteur à T coordonnées. Problème : rien ne garantit que les coordonnées de $\mathbf{M}\mathbf{v}$ soient positives ni qu'elles somment à 1.

Définition 6. La fonction softmax est une fonction de $\mathbb{R}^d$ dans $\mathbb{R}^d$ (d quelconque) définie par :

$$\mathbf{z} \in \mathbb{R}^d \mapsto \text{softmax}(\mathbf{z}) \in \mathbb{R}^d$$
$$\text{softmax}(\mathbf{z})_k = \frac{\exp(z_k)}{\sum_{l=1}^d \exp(z_l)}, \quad k = 1, \dots, d.$$

Remarque 3. La fonction softmax devrait en réalité s'appeler softargmax car c'est une version lisse de la fonction discontinue argmax qui prend en entrée un vecteur de réels et renvoie un vecteur one-hot sur la position du maximum s'il est unique (ou la moyenne des one-hot sur les positions des maxima s'il y en a plusieurs).

Exercice. Implémenter la fonction softmax en Python en utilisant les fonctions d'algèbre linéaire définies dans la section précédente.

Solution.

```
import math
def softmax(z):
    exp_z = [math.exp(zk) for zk in z]
    return ext_prod(1 / sum(exp_z), exp_z)
```

Proposition 2 (Propriétés de softmax). Pour tout $\mathbf{z} \in \mathbb{R}^T$:

- le vecteur $\text{softmax}(z) \in \mathbb{R}^T$ a toutes ses coordonnées **strictement positives** et **somment à 1**
- **invariante par translation** : pour tout $c \in \mathbb{R}$, $\text{softmax}(z + c\mathbf{1}) = \text{softmax}(z)$
- de classe C^∞

La fonction paramétrique de probabilité conditionnelle. Notre fonction paramétrique p_θ a pour paramètres $\theta = M$, c'est à dire les T^2 coefficients de la matrice M .

$$\begin{aligned} p_\theta(t_l \mid t_0, \dots, t_{l-1}) &= p_{t_l}, \\ \mathbf{p} &= \text{softmax}(\mathbf{M}\mathbf{v}_{(t_0, \dots, t_{l-1})}) \\ \mathbf{v}(t_0, \dots, t_{l-1}) &= \frac{1}{l} \sum_{k=0}^{l-1} \mathbf{1}_{t_k} \end{aligned}$$

Remarque 4. On appelle $\mathbf{z} = \mathbf{M}\mathbf{v} \in \mathbb{R}^T$ les logits (*log-probabilités*).

3.3 La fonction à minimiser : log-vraisemblance négative

3.3.1 Pour un token dans un prénom

Soit un prénom a n lettres, donc $n + 2$ tokens (token stop au début et à la fin). Connaissant les tokens $t_0, \dots, t_{l-1}$, on veut prédire le token t_l . La fonction que l'on veut minimiser, la log-vraisemblance négative, et une fonction de $\theta = M$. Elle vaut

$$\begin{aligned} \mathcal{L}_{\text{token}}(M) &= -\log(p_M(t_l \mid t_0, \dots, t_{l-1})) \\ &= -\log(p_{t_l}), \mathbf{p} = \text{softmax}(\mathbf{M}\mathbf{v}) \end{aligned}$$

3.3.2 Pour tous les tokens d'un prénom

Pour un prénom de n lettres, le produit des probabilités conditionnelles devient une somme avec le log. Log-vraisemblance négative est donc la somme des log-vraisemblance négative pour chaque token :

$$\begin{aligned} \mathcal{L}_{\text{prénom}}(M) &= -\sum_{l=0}^n \log(p_M(t_l \mid t_0, \dots, t_{l-1})) \\ &= -\sum_{l=0}^n \log(\text{softmax}(M\mathbf{v}(t_0, \dots, t_{l-1}))_{t_l}) \end{aligned}$$

3.3.3 Pour tous les prénoms de la base de données

Par hypothèse d'indépendance, la log-vraisemblance négative est la somme des log-vraisemblance négative pour chaque prénom :

$$\begin{aligned} \mathcal{L}(M) &= -\sum_{q=1}^Q \sum_{l=0}^{n^{(q)}} \log(p_M(t_l \mid t_0, \dots, t_{l-1})) \\ &= -\sum_{q=1}^Q \sum_{l=0}^{n^{(q)}} \log(\text{softmax}(M\mathbf{v}(t_0, \dots, t_{l-1}))_{t_l}) \end{aligned}$$

Pour des raisons de stabilité numérique, on utilisera la moyenne au lieu des sommes dans toutes les formules ci-dessus.

Remarque 5 (Repère à l'initialisation). Si on prend pour initialisation $M = 0$, la matrice nulle, alors $\mathbf{z} = 0$. La probabilité $\mathbf{p}$ est uniforme ($p_k = 1/T$) pour tous les tokens. Par conséquent, $\ell = \log T$ pour tous les tokens, et la fonction coût globale vaut $\mathcal{L}(M) = \log T$.

4 Minimisation

On veut minimiser la fonction de coût globale $\mathcal{L}(M)$, somme sur tous les prénoms et tous les tokens des log-vraisemblances négatives $\mathcal{L}_{\text{token}}(M)$ introduites à la section précédente. Comme le gradient d'une somme est la somme des gradients, **il suffit de savoir calculer le gradient de la loss d'un seul token**; on l'additionne ensuite sur tous les tokens de tous les prénoms. On procède en deux temps : le **gradient** (pour la descente de gradient), puis la **convexité** (pour savoir ce que l'on minimise).

4.1 Calcul du gradient (loss d'un token)

On se concentre donc sur la loss d'un token. Rappelons les notations de la section précédente : le contexte $(t_0, \dots, t_{l-1})$ donne le feature vector $\mathbf{v}(t_0, \dots, t_{l-1}) \in \mathbb{R}^T$, les logits sont $\mathbf{z} = \mathbf{M}\mathbf{v}$, la distribution prédite est $\mathbf{p} = \text{softmax}(\mathbf{z})$, et la cible est le token t_l . La loss d'un token s'écrit

$$\mathcal{L}_{\text{token}}(M) = -\log \text{softmax}(M\mathbf{v})_{t_l} = \log \left(\sum_{j=1}^T e^{z_j} \right) - z_{t_l}.$$

On calcule son gradient par la règle de la chaîne : d'abord par rapport aux logits z , puis par rapport à la matrice M .

Étape 1 - gradient par rapport aux logits z . On a :

$$\begin{aligned} \frac{\partial}{\partial z_k} \log \left(\sum_l e^{z_l} \right) &= \frac{e^{z_k}}{\sum_l e^{z_l}} = p_k, \\ \frac{\partial}{\partial z_k} (-z_{t_l}) &= -\mathbb{1}[k = t_l]. \end{aligned}$$

et donc la dérivée de la log-vraisemblance négative par rapport au logit z_k vaut :

$$\frac{\partial \mathcal{L}_{\text{token}}}{\partial z_k} = p_k - \mathbb{1}[k = t_l]$$

donc le gradient (dont les coordonnées sont les dérivées partielles) vaut :

$$\nabla_z \mathcal{L}_{\text{token}} = \mathbf{p} - \mathbf{1}_{t_l}$$

où $\mathbf{1}_{t_l}$ est le vecteur one-hot du token t_l à prédire.

Étape 2 - gradient par rapport à M . On a :

$$\begin{aligned} z_k &= \sum_j M_{kj} v(t_0, \dots, t_{l-1})_j, \\ \frac{\partial z_k}{\partial M_{ij}} &= \mathbb{1}[k = i] v(t_0, \dots, t_{l-1})_j. \end{aligned}$$

Par la règle de la chaîne :

$$\begin{aligned} \frac{\partial \mathcal{L}_{\text{token}}}{\partial M_{ij}} &= \sum_k \frac{\partial \mathcal{L}_{\text{token}}}{\partial z_k} \frac{\partial z_k}{\partial M_{ij}} \\ &= (p_i - \mathbb{1}[i = t_l]) v(t_0, \dots, t_{l-1})_j \\ &= \underbrace{((\mathbf{p} - \mathbf{1}_{t_l}) \mathbf{v}(t_0, \dots, t_{l-1}))^\top}_{\text{matrice } \sim(T, T)}_{ij} \end{aligned}$$

Par linéarité du gradient, le gradient total est la moyenne du gradient par token $\nabla_M \mathcal{L}_{\text{token}} = (\mathbf{p} - \mathbf{1}_{t_l}) \mathbf{v}(t_0, \dots, t_{l-1})^\top$ sur tous les tokens de tous les prénoms,

$$\nabla_M \mathcal{L} = \frac{1}{Q} \sum_{\text{prénoms}} \frac{1}{n^{(q)}} \sum_{\text{tokens}} (p - \mathbf{1}_{t_l}) v(t_0, \dots, t_{l-1})^\top,$$

Exercice. Implémenter la fonction $\nabla_M \mathcal{L}$ en Python en utilisant les fonctions d'algèbre linéaire définies dans la section précédente.

Solution.

```
def gradient_nll(M, tokens, target_token, T):
    v = feature_vector(tokens, T)
    z = mat_vec(M, v)
    p = softmax(z)
    p[target_token] -= 1 # p - 1_{t_l}
    p_col = [[p_i] for p_i in p]
    v_row = [v]
    return mat_mat(p_col, v_row)
```

4.1.1 Descente de gradient stochastique

Le gradient exact $\nabla_M \mathcal{L}$ est une *moyenne* sur les Q prénoms de la base. Or Q est grand ($Q \approx 10^4$ dans ce que nous verrons dans le TP), calculer a un coût prohibitif. De plus, de gradient exact est inutilement précis au début de l'entraînement : on n'a pas besoin de la direction exacte pour savoir qu'il faut descendre.

L'idée de la **descente de gradient stochastique** (SGD) est de remplacer la moyenne sur toute la base par une moyenne sur un petit échantillon de b prénoms tirés au hasard (on prendra même $b = 1$ en TP). $\widehat{\nabla_M \mathcal{L}}$ ce gradient approché, et on met à jour

$$M \leftarrow M - \eta \widehat{\nabla_M \mathcal{L}}.$$

L'intérêt est double :

- **Coût par étape** : $Q \rightarrow b$
- **Estimateur sans biais** : si le prénom est tiré uniformément, $\mathbb{E}[\widehat{\nabla_M \mathcal{L}}] = \nabla_M \mathcal{L}$. En moyenne, on suit bien la direction du gradient ; le tirage n'ajoute qu'un *bruit* centré autour de la bonne direction. Ceci peut être bénéfique lorsque la fonctionnelle est non convexe.

4.1.2 L'algorithme ADAM

La SGD « brute » utilise un **même pas η pour tous les paramètres**, ce qui n'est pas robuste, ni optimal. Selon les coordonnées, le gradient peut être systématiquement trop grand auquel cas l'algorithme diverge, ou trop petit auquel cas l'algorithme converge excessivement lentement. De plus, le gradient stochastique est bruité, ce qui peut faire osciller la trajectoire.

ADAM (*Adaptive Moment estimation*) corrige ces deux points en gardant, **coordonnée par coordonnée**, deux moyennes glissantes exponentielles (Exponential Moving Average, EMA) du gradient $\widehat{\nabla_M \mathcal{L}}$ à l'étape t :

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t && \text{(moment d'ordre 1 : la } \textit{moyenne}, \text{ un « élan »),} \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 && \text{(moment d'ordre 2 : l' } \textit{amplitude} \text{ typique, au carré).} \end{aligned}$$

Comme $m_0 = v_0 = 0$, ces moyennes sont biaisées vers 0 au début ; on corrige ce biais par

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t},$$

puis on met à jour chaque paramètre par

$$M \leftarrow M - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}.$$

L'intuition tient en deux points :

- Le numérateur $\widehat{m}_t$ **lisse le bruit** de la SGD (on avance dans la direction moyenne récente, avec de l'inertie).
- La division par $\sqrt{\widehat{v}_t}$ **renormalise chaque coordonnée** par sa propre amplitude de gradient : les coordonnées à fort gradient sont freinées, celles à faible gradient accélérées. Chaque paramètre a ainsi un *pas effectif adapté*, et η n'a plus qu'un rôle d'échelle globale. Le ε (petit, $\sim 10^{-8}$) évite simplement la division par zéro.

Exercice. Implémenter l'algorithme ADAM en Python en utilisant les fonctions d'algèbre linéaire définies dans la section précédente.

4.2 Convexité de la fonction de coût

Proposition 3. La loss d'un token $M \mapsto \mathcal{L}_{\text{token}}(M)$ est **convexe**. Par somme, la fonction de coût globale $\mathcal{L}$ est convexe.

Démonstration. Écrivons $\mathcal{L}_{\text{token}} = g(\mathbf{M}\mathbf{v})$ avec $g(\mathbf{z}) = \log(\sum_l e^{z_l}) - z_{t_l}$.

1. La fonction $z \mapsto \log \sum_l e^{z_l}$ (*log-sum-exp*) est convexe. En effet, sa hessienne vaut

$$\frac{\partial^2}{\partial z_k \partial z_l} \log \sum_m e^{z_m} = p_k \mathbb{1}[k=l] - p_k p_l, \quad \text{soit} \quad H = \text{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top.$$

Pour tout $u \in \mathbb{R}^T$,

$$u^\top H u = \underbrace{\sum_k p_k u_k^2}_{\mathbb{E}_p[u^2]} - \underbrace{\left(\sum_k p_k u_k\right)^2}_{\mathbb{E}_p[u]^2} = \underbrace{\sum_k p_k \left(u_k - \left(\sum_l p_l u_l\right)\right)^2}_{\mathbb{E}_p[(u - \mathbb{E}_p[u])^2]} \geq 0,$$

(inégalité de Jensen avec $\sum_k p_k = 1$ et la fonction $x \mapsto x^2$ convexe). Donc $H \succeq 0$: log-sum-exp est convexe.

2. Le terme $-z_{t_l}$ est linéaire (donc convexe), et la somme de fonctions convexes reste convexe : g est convexe.
3. Enfin $M \mapsto Mv$ est **linéaire**, et la composée d'une fonction convexe avec une application affine est convexe. Donc $\mathcal{L}_{\text{token}} = g(\mathbf{M}\mathbf{v})$ est convexe en $\mathbf{M}$.

□

Remarque 6. On a pas stictte convexite car la hessienne a une vp nulle. La hessienne $\mathbf{H} = \text{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top$ est semi-définie positive mais **singulière** : $H\mathbf{1} = p - p(\mathbf{1}^\top p) = p - p = 0$. La direction $\mathbf{1}$ correspond exactement à l'invariance par translation de softmax. La fonction n'est donc pas strictement convexe, pas d'unicité du minimum global. On peut en effet ajouter une constante c à une matrice $\mathbf{M}$ sans changer la fonction coût. Cela vient du fait que le problème est surparamétré : on a T probabilités, on a besoin en réalité de $T-1$ paramètres libres, la dernière probabilité étant déterminée par le fait que les probas sont positives et somment à 1.

4.2.1 Pour aller plus loin : méthode de Newton

Puisqu'on vient de calculer la **hessienne**, on peut s'en servir. L'idée de la **méthode de Newton** est de remplacer, autour du point courant θ , la fonction par son développement de Taylor à l'ordre 2,

$$\mathcal{L}(\theta + \delta) \approx \mathcal{L}(\theta) + \nabla \mathcal{L}(\theta)^\top \delta + \frac{1}{2} \delta^\top \nabla^2 \mathcal{L}(\theta) \delta,$$

puis de sauter directement au minimum de cette fonction quadratique. Cette fonction quadratique est convexe, elle est minimisée en annulant son gradient, ce qui donne le pas de Newton δ , solution de carré minimum de l'équation

$$\nabla^2 \mathcal{L}(\theta)x = \nabla \mathcal{L}(\theta)$$

d'où la mise à jour

$$\theta \leftarrow \theta - \delta.$$

On peut la voir comme une descente de gradient dont le pas scalaire η est remplacé par la pseudo-inverse de la hessienne $\nabla^2 \mathcal{L}$. L'intérêt est une convergence **quadratique** (le nombre de décimales correctes double à chaque étape) près de l'optimum, bien plus rapide que la convergence linéaire de la descente de gradient.

On peut d'ailleurs relire **Adam** sous cet angle : la division coordonnée par coordonnée par $\sqrt{\widehat{v}_t}$ est une approximation diagonale bon marché de la Hessienne, une version du pauvre du pas de la méthode de Newton en gardant que la diagonale de la hessienne au lieu de la matrice complète (détails en annexe).

5 Différentiation automatique

5.1 Motivation

Pour la régression logistique, nous avons calculé le gradient $\nabla_M \mathcal{L}$ à la main. C'était possible car la fonction $\mathcal{L}$ est relativement simple : on multiplie la matrice $\mathbf{M}$ par le vecteur $\mathbf{v}$, on applique un softmax et on sélectionne la log-vraisemblance négative du token t_l à prédire.

Lorsque l'on choisit une classe de fonction plus compliquée, par exemple un réseau de neurones de type transformer, la fonction $\theta \mapsto \mathcal{L}(\theta)$ est une composition de dizaines voire de centaines d'opérations dites élémentaires comme les produits matrice-vecteur, normalisation de vecteurs, fonctions softmax ou ReLU, ... Calculer le gradient de la fonction de coût est long, source de nombreuses erreurs possible, et tout est à refaire lorsqu'on change de classe de fonction (on dit qu'on change d'architecture de réseau de neurones dans le jargon). Il y a trois façons de faire calculer un gradient par un ordinateur.

- **Différences finies** On approxime numériquement le gradient par différences finies :

$$\frac{\partial \mathcal{L}}{\partial \theta_i} \approx \frac{\mathcal{L}(\theta + h\mathbf{e}_i) - \mathcal{L}(\theta)}{h}$$

C'est simple, mais il faut $p + 1$ évaluations de $\mathcal{L}$ pour p paramètres. De plus, le résultat n'est qu'approché : pour h trop grand on a des erreurs de troncature, pour h trop petit des erreurs d'arrondi, rappelez-vous en python $(1 + 1\text{e-16} - 1) / 1\text{e-16} = 0$.

- **Différentiation symbolique** (comme Mathematica ou sympy) : on calcule une expression formelle de la dérivée.
- **Différentiation automatique** : on décompose le calcul en opérations élémentaires dont on connaît la dérivée, et on applique la règle de la chaîne *numériquement*, opération par opération. Le résultat est exact (à la précision machine près) et le coût de calcul est de l'ordre du coût de calcul d'une évaluation de la fonction $\mathcal{L}$.

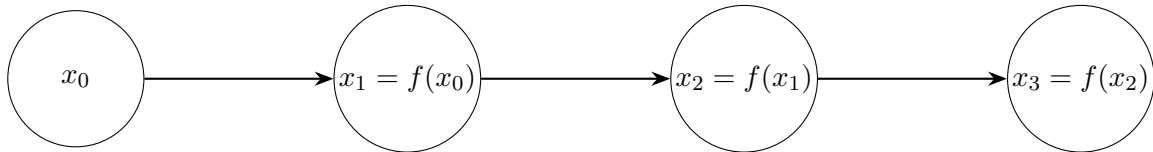
La différentiation automatique est le cœur de toutes les bibliothèques d'apprentissage profond (PyTorch, JAX, TensorFlow). Nous allons dans la suite en voir les principes et en écrire une version minimale inspiré par le micrograd d'Andrej Karpathy.

5.2 Cas d'une fonction d'une variable

Imaginons que nous n'ayons qu'un paramètre x à optimiser et que notre fonction de coût $\mathcal{L}$ soit la composition de trois fonctions

$$\begin{aligned}\mathcal{L}(x) &= f_3(f_2(f_1(x))) = \exp(\sqrt{x^2 + 1}) \\ f_1(x) &= x^2 + 1 \\ f_2(x) &= \sqrt{x} \\ f_3(x) &= \exp(x)\end{aligned}$$

avec comme graphe de calcul :



Par la règle de composition des dérivées, on a

$$\mathcal{L}'(x) = (f_2 \circ f_1)'(x) \times f_3'(f_2(f_1(x))) = f_1'(x) \times f_2'(f_1(x)) \times f_3'(f_2(f_1(x))).$$

On introduit les variables intermédiaires

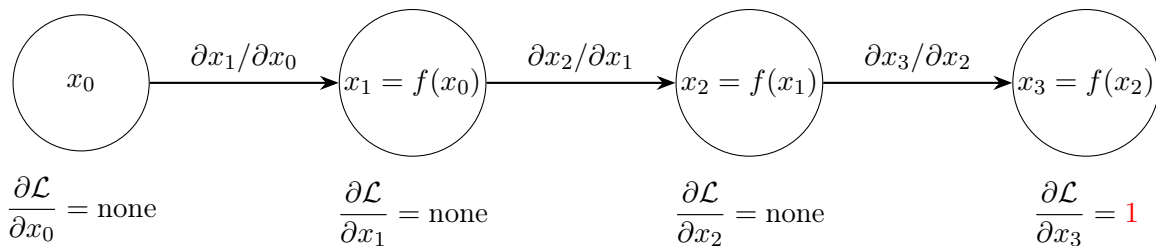
$$x_0 = x, \quad x_1 = f_1(x_0), \quad x_2 = f_2(x_1), \quad x_3 = f_3(x_2)$$

pour obtenir la formule de la dérivée

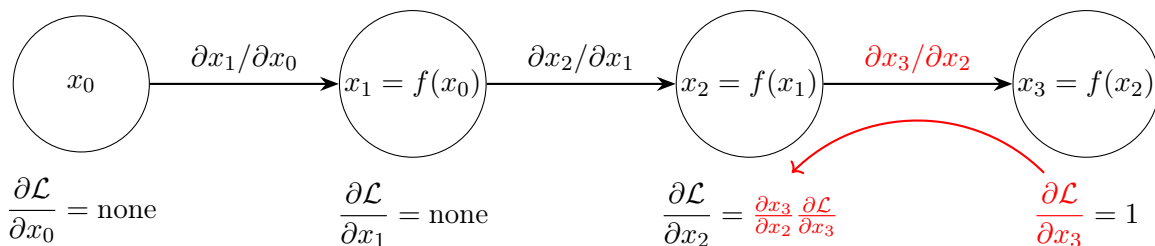
$$\mathcal{L}'(x) = f_1'(x_0) \times f_2'(x_1) \times f_3'(x_2) = \frac{\partial x_1}{\partial x_0} \times \frac{\partial x_2}{\partial x_1} \times \frac{\partial x_3}{\partial x_2}$$

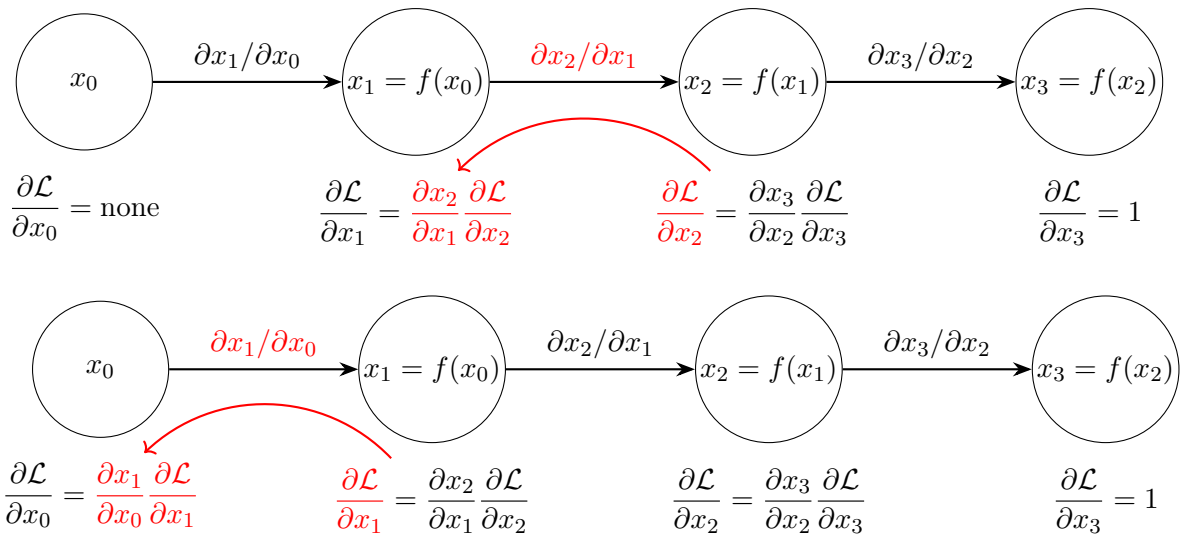
Si lors du calcul initial on stocke :

- sur l'arrête $k \rightarrow k + 1$ du graphe de calcul la dérivées locales $\partial x_k / \partial x_{k-1}$
- dans le noeud x_k la dérivée de la sortie $\mathcal{L}(x)$ par rapport à x_k , initialisée a none sauf pour le dernier noeud x_3 pour lequel la dérivée est 1 car $\mathcal{L}(x) = x_3$



Alors il suffit de remonter le graphe de calcul en multipliant la dérivée de la sortie par la dérivée locale de chaque noeud :





On a donc à la fin du calcul la dérivée de la fonction de coût $\mathcal{L}$ par rapport à tous les x_k , et donc par rapport à x .

Pour une composition de n fonctions :

— On généralise cette formule par récurrence simple et immédiate :

$$\begin{aligned} \mathcal{L}(x) &= f_n(f_{n-1}(\dots(f_1(x))\dots)) \\ x_0 &= x, x_{k+1} = f_{k+1}(x_k) \quad \text{pour } k = 0, \dots, n-1 \\ \mathcal{L}'(x) &= \prod_{k=0}^{n-1} f'_{k+1}(x_k) \end{aligned}$$

- On représente ce calcul par son **graphe de calcul**, c.a.d. un noeud par variable intermédiaire x_k qui contient
 - la valeur de la quantité
 - la dérivée de la loss par rapport cette variable intermédiaire x_k
 - le parent x_{k-1}
 - la dérivée de x_k par rapport au parent
- une fois calculé la loss $\mathcal{L}(x) = x_n$ on appelle la méthode `.backward()` qui
 - initialise $\partial \mathcal{L} / \partial x_n = 1$, puisque $\mathcal{L}(x) = x_n$
 - rétropropage jusqu'à x_0 le gradient avec la formule

$$\frac{\partial \mathcal{L}}{\partial x_{k-1}} = \frac{\partial x_k}{\partial x_{k-1}} \frac{\partial \mathcal{L}}{\partial x_k}$$

Implémentation. Voilà comment cela s'implémente en python avec la classe `Value1D`. Un noeud est un objet Python qui stocke sa valeur (`value`), la dérivée de la loss par rapport à cette valeur (`dloss_dself`), son unique parent et la dérivée de cette valeur par rapport à son parent. Chaque opération mathématique (somme, multiplication, fonction exponentielle ou racine carrée) crée un nouveau noeud (`c` et `n` sont des constantes).

```
import math

class Value1D:
    def __init__(self, value, parent=None, dself_dparent=None, k=0):
        self.value = value                # valeur du noeud (passe avant)
        self.dloss_dself = None           # dérivée de la sortie par rapport au
        # noeud
        self.parent = parent               # l'unique parent (noeud à partir duquel
        # ce noeud est calculé)
```

```

self.dself_dparent = dself_dparent # derivee locale par rapport au parent
self.k = k

def __add__(self, c):
    return Value1D(
        value=self.value + c,
        parent=self,
        dself_dparent=1,
        k=self.k+1)
def __pow__(self, n):
    return Value1D(
        value=self.value**n,
        parent=self,
        dself_dparent=n * self.value**(n-1),
        k=self.k+1)
def sqrt(self):
    return Value1D(
        value=math.sqrt(self.value),
        parent=self,
        dself_dparent=0.5 / math.sqrt(self.value),
        k=self.k+1)
def exp(self):
    return Value1D(
        value=math.exp(self.value),
        parent=self,
        dself_dparent=math.exp(self.value),
        k=self.k+1)

def backward(self):
    self.dloss_dself = 1
    x_k, x_km1 = self, self.parent
    while x_km1 is not None:
        dloss_dxk, dxk_dxkm1 = x_k.dloss_dself, x_k.dself_dparent
        print(f"k={x_k.k}, dloss/dxk={dloss_dxk:.7f}, dxk/dxkm1={dxk_dxkm1:.7f}")
        x_km1.dloss_dself = dxk_dxkm1 * dloss_dxk
        x_k, x_km1 = x_km1, x_km1.parent # on remonte d'un cran

# calcul explicite
x_ = math.sqrt(3)
loss_ = math.exp(math.sqrt(x_**2 + 1))
loss_prime_ = x_ / math.sqrt(x_**2 + 1) * loss_
print(f"loss = {loss_:.7f}, deriv = {loss_prime_:.7f}")
# differentiation automatique
x = Value1D(math.sqrt(3))
loss = (x**2 + 1).sqrt().exp()
loss.backward()
print(f"loss = {loss.value:.7f}, deriv = {x.dloss_dself:.7f}")

```

Sortie :

```

loss = 7.3890561, deriv = 6.3991103
k=4, dloss/dxk=1.0000000, dxk/dxkm1=7.3890561
k=3, dloss/dxk=7.3890561, dxk/dxkm1=0.2500000
k=2, dloss/dxk=1.8472640, dxk/dxkm1=1.0000000
k=1, dloss/dxk=1.8472640, dxk/dxkm1=3.4641016
loss = 7.3890561, deriv = 6.3991103

```

Surcharge d'opérateurs. En Python, `+`, `*`, `**` ... sont des méthodes (`__add__`, `__mul__`, `__pow__` ...) que l'on peut redéfinir pour nos propres objets : c'est la **surcharge d'opérateurs**.

L'expression `x**2 + 1` appelle d'abord `x.__pow__(2)` puis `x.__add__(1)`, et si `x` est un `Value1D`, `x**2` est un nouveau `Value1D` qui mémorise son parent et sa dérivée locale, puis `x**2 + 1` est un nouveau `Value1D` qui mémorise son parent et sa dérivée locale. Ainsi, l'expression `(x**2 + 1).sqrt().exp()` s'évalue de gauche à droite, chaque opération empilant un nœud sur le précédent : le graphe de calcul se construit de façon automatique au fil du calcul. Une fois le calcul fait, le graphe est prêt pour le calcul de la dérivée par rétropropagation avec `backward()`.

5.3 Cas général : fonctions de d variables

5.3.1 Un exemple

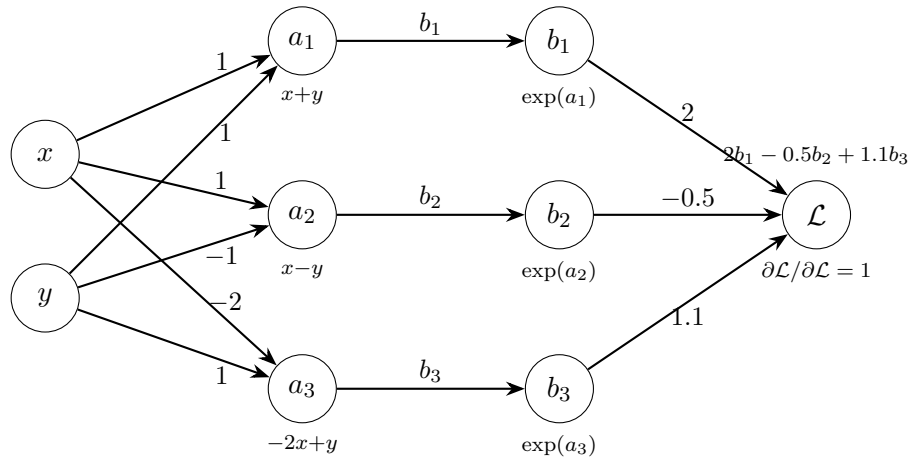
On prend la fonction de $\mathcal{L}$ à deux variables suivante :

$$\mathcal{L}(x, y) = 2 \exp(x + y) - 0.5 \exp(x - y) + 1.1 \exp(-2x + y)$$

On décompose $\mathcal{L}$ en deux “couches” de trois nœuds :

- une première couche cachée de **combinaisons linéaires** a_1, a_2, a_3 ,
- puis une couche de **non-linéarités** $b_i = \exp(a_i)$,

puis la sortie est une combinaison linéaire de (b_1, b_2, b_3) . De la même façon, on représente le graphe de calcul, avec les dérivées du nœud par rapport à son parent sur les arrêtes, et $\partial \mathcal{L} / \partial \mathcal{L}$ initialisée à 1, et toutes les autres à `None` (non-représentés ci-dessous), ce qui donne le graphe suivant :



Contrairement au cas d'une variable, le graphe n'est plus une chaîne : un nœud peut avoir plusieurs parents (a_1 dépend de x et y) et un parent peut avoir plusieurs enfants (x alimente a_1 , a_2 et a_3). C'est cette structure de **DAG** (graphe orienté acyclique) qui rend la règle de la chaîne multi-variable nécessaire.

5.3.2 Règle de la chaîne à plusieurs variables

Théorème 1 (Règle de la chaîne, une entrée, plusieurs intermédiaires). Soit $u \in \mathbb{R}$, $v : \mathbb{R} \rightarrow \mathbb{R}^n$, $u \mapsto (v_1(u), \dots, v_n(u))$ dérivable en u , et $\mathcal{L} : \mathbb{R}^n \rightarrow \mathbb{R}$ différentiable en $v(u)$. Alors $\mathcal{L} \circ v : \mathbb{R} \rightarrow \mathbb{R}$ est dérivable en u et

$$(\mathcal{L} \circ v)'(u) = \sum_{i=1}^n \frac{\partial \mathcal{L}}{\partial v_i}(v(u)) \frac{\partial v_i}{\partial u}(u).$$

Démonstration. Soit $h \neq 0$, $\Delta v_i = v_i(u + h) - v_i(u)$ et $\Delta \mathcal{L} = \mathcal{L}(v(u + h)) - \mathcal{L}(v(u))$.

1. Par dérivabilité des v_i : $\Delta v_i = v'_i(u) h + \varepsilon_i(h) h$ avec $\varepsilon_i(h) \rightarrow 0$. En particulier $\|\Delta v\| \rightarrow 0$, et $\frac{\|\Delta v\|}{|h|} \rightarrow \sqrt{\sum_i v'_i(u)^2}$ reste borné.
2. Par différentiabilité de $\mathcal{L}$: $\Delta \mathcal{L} = \sum_i \frac{\partial \mathcal{L}}{\partial v_i}(v(u)) \Delta v_i + \eta(\Delta v) \|\Delta v\|$ avec $\eta(\Delta v) \rightarrow 0$ quand $\|\Delta v\| \rightarrow 0$.
3. En substituant et en divisant par h :

$$\frac{\Delta \mathcal{L}}{h} = \sum_i \frac{\partial \mathcal{L}}{\partial v_i}(v(u)) v'_i(u) + \underbrace{\sum_i \frac{\partial \mathcal{L}}{\partial v_i}(v(u)) \varepsilon_i(h)}_{\rightarrow 0} + \underbrace{\eta(\Delta v) \frac{\|\Delta v\|}{h}}_{\rightarrow 0 \text{ (borné} \times \rightarrow 0)}.$$

On conclut en faisant tendre h vers 0. □

Le principe est sensiblement le même que dans le cas à une variable, mais on voit que pour pouvoir calculer la dérivée de la loss par rapport à un nœud, il faut d'abord avoir calculé la dérivée de la loss par rapport à tous les nœuds intermédiaires, c.a.d tous les enfants du nœud. Pour cela nous avons besoin de parcourir le graphe de calcul dans un certain ordre appelé ordre topologique inverse.

5.3.3 Ordre topologique

D'après la formule de la règle de la chaîne à plusieurs variables ci-dessus, pour calculer $\partial \mathcal{L} / \partial u$, il faut connaître $\partial \mathcal{L} / \partial v_i$ pour tous les enfants v_i de u . On doit donc traiter les nœuds dans un ordre où chaque nœud vient après tous ses enfants.

Définition 7. Un *ordre topologique* d'un graphe orienté acyclique est une énumération de ses nœuds dans laquelle chaque nœud apparaît après tous ses parents.

Un tel ordre existe toujours pour un graphe acyclique, et s'obtient par un parcours en profondeur : on ajoute un nœud à la liste une fois que tous ses parents y ont été ajoutés. En parcourant cet ordre à l'envers, chaque nœud est traité après tous ses enfants.

Exercice. Construire en python un graphe orienté acyclique correspondant au graphe de calcul de $\mathcal{L}$ ci-dessus, et écrire une fonction `build_topo` qui construit un ordre topologique du graphe.

Solution.

```
class Node:
    def __init__(self, name, parents=()):
        self.name = name
        self.parents = parents
    # pour l'affichage
    def __repr__(self):
        return self.name

    def build_topo(self, topo=None, visited=None):
        if topo is None: topo = []
        if visited is None: visited = set()
        if self not in visited:
            visited.add(self)
            for parent in self.parents:
                parent.build_topo(topo, visited)
            topo.append(self)
        return topo, visited

# graphe correspondant au graphe de calcul de L
x, y = Node("x"), Node("y")
```

```

a1, a2, a3 = Node("a1", (x, y)), Node("a2", (x, y)), Node("a3", (x, y))
b1, b2, b3 = Node("b1", (a1,)), Node("b2", (a2,)), Node("b3", (a3,))
loss = Node("loss", (b1, b2, b3))

topo, visited = loss.build_topo()
print(f"Ordre topologique: {topo}")

```

Sortie :

Ordre topologique: [x, y, a1, b1, a2, b2, a3, b3, loss]

Est-il unique? Non, il y a plusieurs ordres topologiques possibles pour un même graphe, par exemple :

```

[y, x, a1, b1, a2, b2, a3, b3, loss] # echange de x et y
[y, x, a1, a2, a3, b1, b2, b3, loss] # tous les ai avant les bi
...

```

Définition 8. Un *ordre topologique inverse* d'un graphe orienté acyclique est une énumération de ses nœuds dans laquelle chaque nœud apparaît **avant** tous ses parents.

C'est simplement l'ordre topologique inversé, qui s'obtient avec la fonction `reversed` en python sur un ordre topologique.

Algorithme de rétropropagation (backward).

1. On évalue la fonction, ce qui construit le graphe et les dérivées locales des noeuds par rapport a leur parent. La dérivée de la loss par rapport au noeud est initialisée a None.
2. On construit un ordre topologique des nœuds à partir de la sortie, et on met la dérivée de la loss par rapport au noeud a 0 si le noeud est visité. Un noeud non visité n'a pas de dérivée par rapport a la loss, donc la dérivée de la loss par rapport au noeud reste None.
3. On initialise $\partial \mathcal{L} / \partial \mathcal{L} = 1$.
4. Pour chaque nœud v dans l'ordre topologique inverse, et pour chaque parent u de v :

$$\bar{u} += \frac{\partial v}{\partial u} \partial \mathcal{L} / \partial v.$$

Exercice. Implémenter en python la classe `Value` qui implémente la construction du graphe et la rétropropagation. Tester l'exemple de $\mathcal{L}$ ci-dessus

$$\mathcal{L}(x, y) = 2 \exp(x + y) - 0.5 \exp(x - y) + 1.1 \exp(-2x + y)$$

avec $x = 1$ et $y = 2$.

Solution.

```

import math

class Value:
    def __init__(self, value, parents=(), dself_dparents=()):
        self.value = value                # valeur du noeud (passe avant)
        self.dloss_dself = None           # dérivée de la sortie par rapport au
        noeud
        self.parents = parents             # les parents (0, 1 ou 2 dans nos
        operations elementaires)
        self.dself_dparents = dself_dparents # dérivées locales par rapport aux
        parents

        # operations a deux arguments
    def __add__(self, other):

```

```

        other = other if isinstance(other, Value) else Value(other)
        return Value(self.value + other.value, (self, other), (1, 1))

def __mul__(self, other):
    other = other if isinstance(other, Value) else Value(other)
    return Value(self.value * other.value, (self, other), (other.value, self.value))

# operations d'une variable (n est une constante)
def __pow__(self, n): return Value(self.value**n, (self,), (n * self.value**(n-1),))
def exp(self): return Value(math.exp(self.value), (self,), (math.exp(self.value),))
def log(self): return Value(math.log(self.value), (self,), (1 / self.value,))
def relu(self): return Value(max(0, self.value), (self,), (float(self.value > 0),))

# operations derivees des precedentes
def __neg__(self): return self * -1
def __radd__(self, other): return self + other
def __sub__(self, other): return self + (-other)
def __rsub__(self, other): return other + (-self)
def __rmul__(self, other): return self * other
def __truediv__(self, other): return self * other**-1
def __rtruediv__(self, other): return other * self**-1

def build_topo(self, topo=None, visited=None):
    if topo is None: topo = []
    if visited is None: visited = set()
    if self not in visited:
        visited.add(self)
        self.dloss_dself = 0
        for parent in self.parents:
            parent.build_topo(topo, visited)
        topo.append(self)
    return topo, visited

def backward(self):
    topo, visited = self.build_topo()
    self.dloss_dself = 1
    for v in reversed(topo): # enfants traites avant les parents
        for parent, dself_dparent in zip(v.parents, v.dself_dparents):
            parent.dloss_dself += dself_dparent * v.dloss_dself

# test numerique pour x=1, y=2
x_ = 1.0
y_ = 2.0

# calcul explicite de la loss et de ses derivees partielles
loss_ = 2*math.exp(x_ + y_) - 0.5*math.exp(x_ - y_) + 1.1*math.exp(-2*x_ + y_)
dloss_dx_ = 2*math.exp(x_ + y_) - 0.5*math.exp(x_ - y_) - 2.2*math.exp(-2*x_ + y_)
dloss_dy_ = 2*math.exp(x_ + y_) + 0.5*math.exp(x_ - y_) + 1.1*math.exp(-2*x_ + y_)

# loss et ses derivees partielles par differentiation automatique
x = Value(x_)
y = Value(y_)
loss = 2*(x + y).exp() - 0.5*(x - y).exp() + 1.1*(-2*x + y).exp()

```

```
loss.backward()

print(f"loss = {loss.value:.7f} (attendu {loss_:.7f})")
print(f"dL/dx = {x.dloss_dself:.7f} (attendu {dloss_dx:.7f})")
print(f"dL/dy = {y.dloss_dself:.7f} (attendu {dloss_dy:.7f})")
```

Sortie :

```
loss = 41.0871341 (attendu 41.0871341)
dL/dx = 37.7871341 (attendu 37.7871341)
dL/dy = 41.4550136 (attendu 41.4550136)
```

5.3.4 Coût de la différentiation automatique

Proposition 4. Soit E le nombre d'arêtes du graphe de calcul de $\mathcal{L} : \mathbb{R}^p \rightarrow \mathbb{R}$. Le coût de calcul de $\partial \mathcal{L} / \partial x_i$ est $O(E)$. Son coût en mémoire dépend du nombre maximal de parents d'un noeud.

- coût mémoire : $O(E)$, il faut garder en mémoire les dérivées locales des noeuds par rapport à leur parent
- coût computationnel : $O(E)$, la passe arrière effectue une multiplication et une addition par arête, de l'ordre du coût de l'évaluation de la fonction.

Les différences finies, elles, demandent $p + 1$ évaluations. Avec les réseaux de neurones, on a vite $p \approx 10^3 - 10^6$ paramètres, ce qui rend les différences finies inutilisables. La contrepartie est la mémoire : tout le graphe de calcul doit être gardé jusqu'à la passe arrière.

Remarque 7 (Mode direct et mode inverse). Ce que nous avons décrit est le **mode inverse** (reverse mode) de la différentiation automatique : on propage $\bar{u} = \frac{\partial f}{\partial u}$ de la sortie vers les entrées. Il existe aussi un **mode direct** (forward mode) qui propage $\frac{\partial u}{\partial x_j}$ des entrées vers la sortie, une passe par variable d'entrée x_j . Le mode direct est efficace pour les fonctions $\mathbb{R} \rightarrow \mathbb{R}^n$, le mode inverse pour les fonctions $\mathbb{R}^p \rightarrow \mathbb{R}$: c'est notre cas, la fonction de coût étant à valeurs réelles.

5.4 Retour sur la régression logistique

Les fonctions d'algèbre linéaire de la section 0 n'utilisent que $+$, $*$ et `sum` : grâce à la surcharge des opérateurs, elles fonctionnent telles quelles avec des `Value`. Seul le softmax doit être réécrit avec la méthode `.exp()`.

```
def softmax_value(z):
    m = max(zi.value for zi in z)          # stabilite numerique, m est une constante
    exp_z = [(zi - m).exp() for zi in z]
    s = sum(exp_z)
    return [e / s for e in exp_z]
```

Exercice. Calculer, par différentiation automatique, le gradient de la loss d'un prénom par rapport à M , et vérifier qu'il coïncide avec la formule explicite $\nabla_M \mathcal{L}_{\text{token}} = (\mathbf{p} - \mathbf{1}_{t_l}) \mathbf{v}^\top$ moyennée sur les tokens.

Solution.

```
def loss_autodiff(M, tokens):
    losses = []
    for l in range(len(tokens) - 1):
        v = feature_vector(tokens[l + 1], T)    # des float
        p = softmax_value(mat_vec(M, v))        # des Value
        losses.append(-p[tokens[l + 1]].log())
    return sum(losses) * (1 / len(losses))
```

```
M = [[Value(random.gauss(0, 0.5)) for _ in range(T)] for _ in range(T)]
loss = loss_autodiff(M, encode('tommy'))
loss.backward()
grad_M = [[Mij.dloss_dself for Mij in Mi_] for Mi_ in M] # = gradient explicite
```

En TP, on vérifie que les deux gradients coïncident à 10^{-16} près, et que la descente de gradient donne la même matrice M avec les deux méthodes.

6 Les transformers

6.1 Du feature vector au réseau de neurones

Le feature vector de la régression logistique, la moyenne des one-hot du contexte, a deux défauts : il oublie l'ordre des tokens (**anna** et **naan** ont la même représentation), et il donne le même poids à tous les tokens passés, alors que pour prédire la lettre suivante la dernière lettre compte sans doute davantage que la première. Un transformer *apprend* la représentation du contexte : les poids donnés aux tokens passés dépendent eux-mêmes du contexte.

Nous allons construire étape par étape un réseau de neurones de type transformer. Nous verrons qu'à chaque étape, on pourrait utiliser la représentation construite jusque là avec une régression logistique. Il serait intéressant de le faire pour voir la progression de la performance du modèle au fur et à mesure qu'on le complexifie.

Nous décrivons l'architecture GPT-2, simplifiée comme dans le code microgpt de Karpathy : RMSNorm au lieu de LayerNorm, pas de biais, non-linéarité ReLU au lieu de GeLU. On note d la dimension des représentations (**n_embd**, taille des embeddings), T la taille du vocabulaire, et B la longueur maximale du contexte (**block_size**).

6.2 Embedding des tokens et des positions

Deux matrices de paramètres transforment les entrées discrètes (token et position) en vecteurs de $\mathbb{R}^d$:

$$\mathbf{M}_{te} \in \mathbb{R}^{T \times d} \text{ (token embedding), } \quad \mathbf{M}_{pe} \in \mathbb{R}^{B \times d} \text{ (position embedding).}$$

Le token t_l en position l est représenté par la somme des lignes correspondantes :

$$\mathbf{x}_l = \mathbf{M}_{te}[t_l] + \mathbf{M}_{pe}[l] \in \mathbb{R}^d.$$

Remarque 8. $\mathbf{M}_{te}[t] = \mathbf{M}_{te}^\top \mathbf{1}_t$: l'embedding d'un token est une application linéaire appliquée à son one-hot. Dans la première partie, on a fait une régression logistique en calculant l'embedding de chaque token selon cette méthode, puis en faisant la moyenne sur les tokens du contexte. On pourrait donc complexifier notre régression logistique en calculant la moyenne de l'embedding du token plus sa position.

Sans le plongement des positions, l'attention (ci-dessous) serait invariante par permutation du contexte et ne distinguerait pas **ab** de **ba**.

```
embed = lambda tokens, positions, Mtok, Mpos: [sum_vec(Mtok[token], Mpos[pos]) for
    token, pos in zip(tokens, positions)]
```

6.3 L'attention

L'idée principale de l'attention et de calculer des interactions "pairwise" entre les vecteurs représentant chaque token par des simples produits scalaires. L'idée la plus naturelle est de prendre la matrice de Gram $\mathbf{G}$ de taille $n \times n$ des vecteurs représentant les tokens d'une séquence.

Pour la famille de vecteurs $\mathbf{x}_1, \dots, \mathbf{x}_n$ représentant les n tokens d'une séquence, la matrice de Gram de taille $n \times n$ aux coefficients

$$g_{i,j} = \langle \mathbf{x}_i, \mathbf{x}_j \rangle.$$

La matrice d'attention est un peu plus sophistiquée : on utilise des matrices auxiliaires pour le produit scalaire, puis on applique une fonction de softmax aux vecteurs lignes. On obtient ainsi une matrice stochastique, c'est-à-dire une matrice dont les coefficients sont positifs et dont la somme des lignes égale à 1.

6.4 Un tête d'attention

Soient $\mathbf{x}_1, \dots, \mathbf{x}_n$ les représentations des tokens d'une séquence. On voit cette séquence comme une matrice $\mathbf{X}$ à n lignes et d colonnes, chacune des lignes de la matrice représentant un token.

On introduit trois matrices de paramètres de taille $d \times d_h$, d_h étant la dimension de la tête d'attention (d head) :

- $\mathbf{M}_Q$: matrice des queries
- $\mathbf{M}_K$: matrice des keys
- $\mathbf{M}_V$: matrice de values

Avec ces matrices on calcule les queries, keys et values pour chaque position $i \in \{1, \dots, n\}$ de la séquence :

$$\begin{aligned}\mathbf{q}_i &= \mathbf{M}_Q \mathbf{x}_i \in \mathbb{R}^{d_h} \\ \mathbf{k}_i &= \mathbf{M}_K \mathbf{x}_i \in \mathbb{R}^{d_h} \\ \mathbf{v}_i &= \mathbf{M}_V \mathbf{x}_i \in \mathbb{R}^{d_h}.\end{aligned}$$

En écriture matricielle cela s'écrit :

$$\begin{aligned}\mathbf{Q} &= \mathbf{X} \mathbf{M}_Q \sim (n, d_h) \\ \mathbf{K} &= \mathbf{X} \mathbf{M}_K \sim (n, d_h) \\ \mathbf{V} &= \mathbf{X} \mathbf{M}_V \sim (n, d_h).\end{aligned}$$

On calcule une sorte de matrice de Gram normalisée $\mathbf{G}$ de taille $n \times n$ en faisant le produit scalaire entre les vecteurs queries $\mathbf{q}_i$ et les vecteurs keys $\mathbf{k}_j$, que l'on normalise par $\frac{1}{\sqrt{d_h}}$:

$$g_{i,j} = \frac{1}{\sqrt{d_h}} \langle \mathbf{q}_i, \mathbf{k}_j \rangle$$

ou en écriture matricielle :

$$\mathbf{G} = \frac{1}{\sqrt{d_h}} \mathbf{Q} \mathbf{K}^\top = \frac{1}{\sqrt{d_h}} \mathbf{X} \mathbf{M}_Q \mathbf{M}_K^\top \mathbf{X}^\top \sim (n, n)$$

Le facteur $\frac{1}{\sqrt{d_h}}$ est un facteur de normalisation pour que les coefficients de la matrice de Gram soient de l'ordre de 1 en valeur absolue. En effet, si les vecteurs queries et keys sont des vecteurs Gaussiens centrés de variance 1, les produits scalaires $\langle \mathbf{q}_i, \mathbf{k}_j \rangle$ sont des variables aléatoires centrées de variance d_h :

$$\begin{aligned}\mathbb{E}[\langle \mathbf{q}_i, \mathbf{k}_j \rangle] &= 0 \\ \mathbb{E}[\langle \mathbf{q}_i, \mathbf{k}_j \rangle^2] &= d_h\end{aligned}$$

Preuve. On suppose les coordonnées $q_{i,a}$ et $k_{j,b}$ indépendantes, centrées, de variance 1. Par linéarité et indépendance :

$$\mathbb{E}[\langle \mathbf{q}_i, \mathbf{k}_j \rangle] = \sum_{l=1}^{d_h} \mathbb{E}[q_{i,l}] \mathbb{E}[k_{j,l}] = 0.$$

Pour le moment d'ordre 2 :

$$\mathbb{E}[\langle \mathbf{q}_i, \mathbf{k}_j \rangle^2] = \sum_{l,m=1}^{d_h} \mathbb{E}[q_{i,l} q_{i,m}] \mathbb{E}[k_{j,l} k_{j,m}] = \sum_{l,m=1}^{d_h} \delta_{lm} \delta_{lm} = d_h,$$

car $\mathbb{E}[q_{i,l} q_{i,m}] = \delta_{lm}$ (coordonnées indépendantes, centrées, réduites), et de même pour $\mathbf{k}_j$. Ainsi $\text{Var}(g_{i,j}) = \frac{1}{d_h} \cdot d_h = 1$. $\square$

Une fois cette matrice de Gram $\mathbf{G}$ calculée, on applique une fonction de softmax aux vecteurs lignes $\mathbf{g}_{i,\cdot}$ pour obtenir la matrice stochastique $\mathbf{A}$ de taille $n \times n$ des poids d'attention :

$$\mathbf{a}_{i,\cdot} = \text{softmax}(\mathbf{g}_{i,\cdot})$$

On calcule enfin la matrice de sortie $\mathbf{O}$ de taille $n \times d_h$ dont les lignes sont les vecteurs values $\mathbf{v}_i$ pondérés par les poids d'attention $\mathbf{a}_{i,j}$, c'est à dire une combinaison convexe des vecteurs values :

$$\mathbf{o}_i = \sum_{j=1}^n \mathbf{a}_{i,j} \mathbf{v}_j \in \mathbb{R}^{d_h}$$

De façon matricielle, cela s'écrit :

$$\mathbf{O} = \mathbf{A} \mathbf{V}$$

Si l'on adopte la notation $\text{softmax}(M)$ pour appliquer la fonction de softmax à chaque ligne de la matrice M , tout le calcul ci dessus s'écrit de façon concise :

$$\mathbf{O} = \text{softmax}(\mathbf{X} \mathbf{M}_Q \mathbf{M}_K^\top \mathbf{X}^\top) \mathbf{V}$$

Les lignes de la matrice de sortie $\mathbf{O}$ sont de nouvelles représentations des tokens de la séquence. Avec la matrice d'attention, cette nouvelle de représentation d'un token incorpore l'interaction de ce token avec **tous les autres tokens de la séquence**. Le cout de calcul est d'ailleurs quadratique en la longueur de la séquence, c'est à dire $O(n^2)$, caractéristique d'un interaction "pairwise".

En Python, avec nos fonctions de base d'algèbre linéaire, on peut écrire la fonction `attention_head` qui calcule la sortie de l'attention pour un token donné.

```
def attention_head(X, Mq, Mk, Mv):
    d_h = len(Mq) # dimension de la tête d'attention
    scale = 1 / math.sqrt(d_h)
    Q = mat_mat(X, Mq)
    K = mat_mat(X, Mk)
    V = mat_mat(X, Mv)
    G = [[scale * dot_prod(qi_, kj_) for kj_ in K] for qi_ in Q]
    A = [softmax(gi_) for gi_ in G] # softmax line by line
    O = mat_mat(A, V)
    return O
```

6.5 Attention causale

L'attention pleine, telle que définie ci-dessus, n'est pas applicable pour notre problème de prédiction du token suivant. En effet, la probabilité conditionnelle $p_\theta(t_{l+1} | t_0, \dots, t_l)$, dépend seulement des tokens précédents, $t_0, \dots, t_l$, et ne doit pas dépendre des tokens suivants, $t_{l+1}, \dots, t_n$.

Dans l'attention pleine, la représentation d'un token dépend de tous les autres tokens de la séquence, ce qui n'est pas compatible avec la modélisation de la probabilité conditionnelle.

Pour corriger ce problème, on introduit l'attention causale. On ne calcule que la partie **triangulaire inférieure** de la matrice de Gram $\mathbf{G}$, c'est à dire les coefficients $g_{i,j}$ pour $j \leq i$. Et on calcule le softmax uniquement sur les coefficients $g_{i,j}$ pour $j \leq i$:

$$\mathbf{a}_{i,j \leq i} = \text{softmax}(\mathbf{g}_{i,j \leq i}),$$

tous les autres coefficients de la matrice d'attention étant égaux à 0.

On peut directement écrire la fonction `attention_causal` en calculant uniquement la partie triangulaire inférieure de la matrice de Gram, et en appliquant le softmax uniquement sur les coefficients de la partie triangulaire inférieure.

```
def attention_causal(X, Mq, Mk, Mv):
    d_h = len(Mq) # dimension de la tête d'attention
    scale = 1 / math.sqrt(d_h)
    K = mat_mat(X, Mk)
    Q = mat_mat(X, Mq)
    V = mat_mat(X, Mv)
    G_causal = [[scale * dot_prod(qi_, kj_) for j, kj_ in enumerate(K) if j <= i] for
i, qi_ in enumerate(Q)]
    A_causal = [softmax(gi_) for gi_ in G_causal_]
    O_causal = [[dot_prod(ai_, v_j[:len(ai_)]) for v_j in zip(*V)] for ai_ in A_causal
]
    return O_causal
```

Dans le cas où l'on voudrait utiliser des bibliothèques performantes d'algèbre linéaire, on préfère utiliser des formules matricielles pour le calcul de l'attention causale. Il y a deux solutions qui sont très similaires :

- On calcule la matrice de Gram $\mathbf{G}$ comme défini ci-dessus, on conserve la partie triangulaire inférieure, et on met à moins l'infini les coefficients de la partie triangulaire supérieure. Ainsi, lorsqu'on calcule le softmax, l'exponentielle de moins l'infini vaut 0, et donc les coefficients de la partie triangulaire supérieure de la matrice d'attention seront égaux à 0. On introduit donc la matrice $\mathbf{C}$ dont les coefficients sont égaux à moins l'infini pour $j > i$ et 0 pour $j \leq i$, et la sortie de l'attention causale s'écrit :

$$\mathbf{O} = \text{softmax}(\mathbf{G} + \mathbf{C})\mathbf{V}$$

- On calcule la matrice de Gram $\mathbf{G}$ comme défini ci-dessus, on calcule l'exponentielle de chaque coefficient de la matrice de Gram, on met à 0 les coefficients de la partie triangulaire supérieure, puis on normalise les lignes par la somme des coefficients de la ligne. Dans ce cas on introduit la matrice $\mathbf{C}$ dont les coefficients sont égaux à 0 pour $j > i$ et 1 pour $j \leq i$. On calcule l'exponentielle coefficient par coefficient de la matrice de Gram, on multiplie par la matrice $\mathbf{C}$, et on normalise les lignes par la somme des coefficients de la ligne.

$$\mathbf{O} = \text{normalize_lines}(\exp(\mathbf{G})\mathbf{C})\mathbf{V}$$

Attention, dans la formule ci-dessus, $\exp(\mathbf{G})$ est obtenue en appliquant la fonction exponentielle coefficient par coefficient à la matrice de Gram $\mathbf{G}$. Ce n'est pas l'exponentielle matricielle $e^{\mathbf{M}} = \sum_{k=0}^{\infty} \frac{\mathbf{M}^k}{k!}$.

6.5.1 Attention multi-têtes

Dans le papier original "Attention is all you need", les auteurs introduisent l'attention multi-têtes (multi-head attention). Étant donnée une taille d en sortie de l'attention, on découpe cette taille en H blocs de taille $d_h = d/H$, où H est le nombre de têtes d'attention. On calcule alors H

têtes d'attention indépendantes, chacune avec sa propre matrice de paramètres $\mathbf{M}_Q^h, \mathbf{M}_K^h, \mathbf{M}_V^h$, chacun de taille $d \times d_h$. Cela donne H sorties $\mathbf{O}^h$ de taille d_h que l'on concatène pour obtenir la sortie de l'attention multi-têtes $\mathbf{O}$ de taille d :

$$\mathbf{O} = [\mathbf{O}^1; \dots; \mathbf{O}^H].$$

D'un point de vue informatique, on peut représenter H matrices $\mathbf{M}^h$ de taille $d \times d_h$ par une seule matrice $\mathbf{M}$ de taille $d \times d$, puisque $d_h = d/H$. La matrice $\mathbf{M}^1$ correspond aux premières d_h colonnes de la matrice $\mathbf{M}$, la matrice $\mathbf{M}^2$ correspond aux d_h colonnes suivantes, etc. Ainsi le code de l'attention multi-têtes s'écrit simplement en prenant en entrée $\mathbf{X}$, $\mathbf{M}_Q$, $\mathbf{M}_K$ et $\mathbf{M}_V$ et en calculant les H têtes d'attention indépendantes.

```
def multi_head(X, Mq, Mk, Mv, n_head):
    d_h = len(Mq) // n_head
    outputs = []
    for h in range(n_head):
        # on selectionne les colonnes h * d_h:(h + 1) * d_h de chaque matrice
        Mq_h = [Mqi_[h * d_h:(h + 1) * d_h] for Mqi_ in Mq]
        Mk_h = [Mki_[h * d_h:(h + 1) * d_h] for Mki_ in Mk]
        Mv_h = [Mvi_[h * d_h:(h + 1) * d_h] for Mvi_ in Mv]
        outputs.append(attention_head(X, Mq_h, Mk_h, Mv_h))
    return [list(sum(outs)) for outs in zip(*outputs)] # on somme pour la concaté
    nation des listes
```

La version causale du code ci-dessus s'obtient simplement en remplaçant la fonction `attention_head` par la fonction `attention_causal`.

Remarque. On ajoute en général une matrice $\mathbf{M}_O$ de taille $d \times d$ en sortie de l'attention multi-têtes. Dans ce cas la sortie de l'attention multi-têtes s'écrit :

$$\mathbf{O} = [\mathbf{O}^1; \dots; \mathbf{O}^H] \mathbf{M}_O.$$

Les vecteurs de sortie sont alors des combinaisons linéaires des vecteurs de sortie de toutes les têtes d'attention.

6.6 RMSNorm

Dans un réseau de neurones de type transformer, on normalise régulièrement les vecteurs de représentation pour des raisons de stabilité numérique. Sans normalisation, il est possible que la norme des vecteurs devienne trop grande ou trop petite, ce qui peut causer des problèmes de stabilité numérique. On utilise la normalisation par la moyenne quadratique (RMSNorm) pour contrôler l'échelle des représentations, définie simplement par :

$$\text{rmsnorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2 + \varepsilon}}, \quad \varepsilon = 10^{-5}.$$

où ε est un petit paramètre pour éviter la division par zéro.

```
def rmsnorm_vec(x):
    """Input shape: x: (d,)"""
    ms = sum(xi * xi for xi in x) / len(x)
    scale = (ms + 1e-5) ** -0.5
    return [xi * scale for xi in x]

rmsnorm_seq = lambda xs: [rmsnorm_vec(x) for x in xs]
```

6.7 Le Multi-layer Perceptron (MLP)

Un réseau de neurone Multi-layer Perceptron (MLP) est un réseau de neurones qui applique une même transformation non linéaire à chaque vecteur de la séquence en entrée. On y applique à chaque vecteur de la séquence une première matrice $\mathbf{M}_1$ de taille $d \times d_{\text{MLP}}$. On applique une fonction d'activation non-linéaire coordonnée par coordonnée, par exemple la fonction ReLU :

$$\text{ReLU}(x) = \max(0, x).$$

On applique enfin une deuxième matrice $\mathbf{M}_2$ de taille $d_{\text{MLP}} \times d$. La sortie du MLP est donc un vecteur de taille d :

$$\text{MLP}(\mathbf{x}) = \mathbf{M}_2 \text{ReLU}(\mathbf{M}_1 \mathbf{x}) \in \mathbb{R}^d$$

Les paramètres du MLP sont les matrices $\mathbf{M}_1$ et $\mathbf{M}_2$, il y en a $2(d \times d_{\text{MLP}})$. On prendra pour d_{MLP} un multiple de d , par exemple $d_{\text{MLP}} = 4d$, ce qui donne $8d^2$ paramètres.

Remarque. La non-linéarité est nécessaire. Sans elle, on aurait une seule application linéaire dont la matrice équivalente est le produit des deux matrices $\mathbf{M}_1$ et $\mathbf{M}_2$:

$$b\mathbf{x} = \mathbf{M}\mathbf{x}, \mathbf{M} = \mathbf{M}_2\mathbf{M}_1.$$

Une composée d'applications linéaires reste linéaire, il n'y a pas d'intérêt à utiliser deux matrices dans ce cas.

En Python :

```
relu_vec = lambda x: [max(0, xi) for xi in x]
mlp_vec = lambda x, M1, M2: mat_vec(M2, relu_vec(mat_vec(M1, x)))
mlp_seq = lambda xs, M1, M2: [mlp_vec(x, M1, M2) for x in xs]
```

6.8 Un bloc transformer

Un **bloc transformer** un réseau de neurones qui combine les briques de base vues ci dessus :

- en entrée une séquence de vecteurs de représentation $\mathbf{x}_1, \dots, \mathbf{x}_n$.
- on applique une normalisation RMSNorm à chaque vecteur de la séquence.
- on applique l'attention causale à la séquence de vecteurs normalisés, avec une matrice de sortie $\mathbf{M}_O$.
- on ajoute à la sortie de l'attention la séquence en entrée de l'attention (connexion résiduelle).
- on applique à nouveau une normalisation RMSNorm aux vecteurs de la séquence.
- on applique le MLP à la séquence de vecteurs normalisés.
- on ajoute à la sortie du MLP la séquence en entrée du MLP (connexion résiduelle).

Ainsi, le bloc de transformer prend en entrée une séquence de vecteurs de représentation $\mathbf{x}_1, \dots, \mathbf{x}_n$ et renvoie une nouvelle séquence de vecteurs de représentation $\mathbf{x}'_0, \dots, \mathbf{x}'_n$ de même taille.

Connexions résiduelles. Les opérations qui consistent à ajouter la séquence en entrée à la sortie d'un bloc sont appelées connexions résiduelles. Leur but est d'éviter le problème de "vanishing gradients", qui apparait lorsque le gradients de la composition de toutes les couches d'un réseau de neurones devient trop petit, ce qui empeche l'optimisation des paramètres par descente de gradient. En effet, il faut en optimisation éviter que le norme des gradients soit trop petite ou trop grande.

6.9 L'architecture de type GPT

Une architecture de type GPT est un réseau de neurones qui combine les blocs transformer de manière répétée. On empile N blocs transformer, chacun avec ses propres paramètres. Cela nous donne une représentation de la séquence de taille d pour chaque position l de la séquence. On termine par une régression logistique pour obtenir les logits et la loi du prochain token, avec une matrice $\mathbf{M}_{\text{logreg}}$ de taille $d \times T$.

```
T = 27 # nombre de tokens dans le vocabulaire, 27 pour les prénoms
n_layer = 1 # nombre de blocs transformer
d = 16 # width of the network (embedding dimension)
block_size = 16 # maximum context length of the attention window (note: the longest
    name is 15 characters)
n_head = 4 # number of attention heads
d_h = d // n_head # derived dimension of each head
d_mlp = 4 * d # derived dimension of the MLP

matrices = {
    'Mtok': rand_mat(T, d),
    'Mpos': matrix(block_size, d),
    'Mlogreg': matrix(d, T)
}

for i in range(n_layer):
    state_dict[f'layer{i}.attn_Mq'] = rand_mat(d, d)
    state_dict[f'layer{i}.attn_Mk'] = rand_mat(d, d)
    state_dict[f'layer{i}.attn_Mv'] = rand_mat(d, d)
    state_dict[f'layer{i}.attn_Mo'] = rand_mat(d, d)
    state_dict[f'layer{i}.mlp_M1'] = rand_mat(d, d_mlp)
    state_dict[f'layer{i}.mlp_M2'] = rand_mat(d_mlp, d)

params = [p for mat in state_dict.values() for row in mat for p in row] # flatten
    params into a single list[Value]
print(f"num params: {len(params)}")

def gpt(tokens, matrices):
    # calcul des embeddings initiaux
    positions = list(range(len(tokens)))
    sequence = embed(tokens, positions, matrices['Mtok'], matrices['Mpos'])

    for li in range(n_layer):
        sequence_res = sequence # stocker pour la connexion résiduelle
        # normalisation
        sequence = rmsnorm_seq(sequence)
        # attention causale
        Mq, Mk, Mv = matrices[f'layer{li}.attn_Mq'], matrices[f'layer{li}.attn_Mk'],
        matrices[f'layer{li}.attn_Mv']
        sequence = attention_causal(sequence, Mq, Mk, Mv)
        Mo = matrices[f'layer{li}.attn_Mo']
        sequence = mat_mat(sequence, Mo)
        # connexion résiduelle
        sequence = sum_vec(sequence, sequence_res)
        # normalisation
        sequence = rmsnorm_seq(sequence)
        sequence_res = sequence # stocker pour la connexion résiduelle
        # MLP
        sequence = mlp_vec(sequence, matrices[f'layer{li}.mlp_M1'], matrices[f'layer{
```

```
li}.mlp_M2'])
    # connexion résiduelle
    sequence = sum_vec(sequence, sequence_res)

    # regression logistique sur sequence qui est de taille d
    # qui sort la séquence les logits du token suivant pour chaque token de la sé
    quence
    return mat_vec(sequence, matrices['Mlogreg'])
```

6.10 Génération et KV cache

Pour générer du texte, on part de $t_0 = \text{stop}$, on tire $t_1 \sim p_\theta(\cdot | t_0)$, puis $t_2 \sim p_\theta(\cdot | t_0, t_1)$, etc. Si l'on fait naïvement le calcul, on appelle le modèle à l'étape l sur tout le préfixe $t_0, \dots, t_l$ et on ne garde que la dernière sortie. A chaque étape, on recalcule $\mathbf{q}_s, \mathbf{k}_s, \mathbf{v}_s, \mathbf{o}_s$ pour tous les $s < l$, alors qu'ils avaient déjà été calculés à l'étape précédente, et leur valeur ne change pas car elle ne dépend pas des tokens suivants par construction de l'attention causale.

Proposition 5. Dans un réseau de neurones constitué d'une succession de couches d'attention causale, pour toute couche, la représentation $\mathbf{x}_s$ en position s , et donc $\mathbf{k}_s$ et $\mathbf{v}_s$, ne dépendent que des tokens $t_0, \dots, t_s$.

Démonstration. Par récurrence sur les couches. Le plongement $\mathbf{x}_s$ ne dépend que de t_s et s . RMSNorm, le MLP et les connexions résiduelles (ci-dessous) agissent position par position. La sortie de l'attention en position s ne dépend que de $\mathbf{x}_0, \dots, \mathbf{x}_s$ de la couche précédente, qui par hypothèse de récurrence ne dépendent que de $t_0, \dots, t_s$. $\square$

Les clés et valeurs calculées à l'étape s restent donc valables à toutes les étapes suivantes : on les garde en mémoire. C'est le **KV cache** (key-value cache), une liste de clés et une liste de valeurs par couche. À l'étape l , avec le vecteur $\mathbf{x}_l$ en entrée, on ne calcule que $\mathbf{q}_l, \mathbf{k}_l, \mathbf{v}_l$, on ajoute $\mathbf{k}_l, \mathbf{v}_l$ au cache. On calcule ensuite la dernière ligne $\mathbf{g}_l = \mathbf{q}_l \mathbf{K}^\top$ de la matrice de Gram. On applique la fonction de softmax à cette ligne, de qui nous donne la dernière ligne de la matrice d'attention. Et on calcule le vecteur de sortie $\mathbf{o}_l = \sum_{j=1}^l \mathbf{a}_{l,j} \mathbf{v}_j$.

Exercice. Écrire une fonction `attention_head_kvcache` qui prend en entrée $\mathbf{x}_l, \mathbf{M}_Q, \mathbf{M}_K$ et $\mathbf{M}_V$ et les matrices de keys $\mathbf{K}$ et valeurs $\mathbf{V}$ de $\mathbf{x}_1, \dots, \mathbf{x}_{l-1}$, et qui renvoie la sortie de l'attention pour la seule position courante l , et met à jour les matrices de keys et valeurs du cache.

Solution.

```
def attention_head_kvcache(x, Mq, Mk, Mv, K, V):
    """Input shapes:
    - x: (d)
    - Mq: (d, d_h)
    - Mk: (d, d_h)
    - Mv: (d, d_h)
    - K: (l-1, d_h)
    - V: (l-1, d_h)
    """
    d_h = len(Mq) # dimension de la tête d'attention
    scale = 1 / math.sqrt(d_h)
    q = mat_mat(x, Mq)
    k = mat_mat(x, Mk)
    v = mat_mat(x, Mv)
    K.append(k) # shape (l, d_h)
    V.append(v) # shape (l, d_h)
    gl_ = [scale * dot_prod(q, kj_) for kj_ in K] # dernière ligne de la matrice de
    Gram
```

```

al_ = softmax(gl_) # dernière ligne de la matrice d'attention
output = [dot_prod(al_, v_j) for v_j in zip(*V)] # size d_h
return output

```

Coût de la génération de n tokens Prenons un réseau formé d'une couche d'attention. Le fait que l'attention soit causale

- Attention naïve : à l'étape l , $O(l d^2)$ pour les projections et $O(l^2 d)$ pour l'attention, soit au total $\sum_{l \leq n} O(l d^2 + l^2 d) = O(n^2 d^2 + n^3 d)$.
- KV cache : à l'étape l , $O(d^2)$ pour les projections et $O(l d)$ pour l'attention, soit au total $O(n d^2 + n^2 d)$.

On gagne un facteur n . Le prix est la mémoire : le cache contient $2 n d$ nombres par couche. Pour les LLMs, dont les tailles de contexte atteignent des centaines de milliers de tokens, la taille du KV cache est un sujet important pour la consommation mémoire et le temps de calcul.

Remarque 9 (Entraînement). *Pendant l'entraînement, tous les tokens du prénom sont connus à l'avance : on calcule toutes les positions en une seule passe avec l'attention naïve, chaque position n'est calculée qu'une fois et le cache n'apporte rien. Sur GPU, on écrit cette passe sous forme matricielle, $\mathbf{O} = \text{softmax}(\mathbf{Q}\mathbf{K}^\top / \sqrt{d_h} + \mathbf{C}) \mathbf{V}$ (softmax ligne par ligne), avec un masque causal $C_{l,s} = 0$ si $s \leq l$ et $C_{l,s} = -\infty$ sinon. En python, on a bien `math.exp(-math.inf) == 0.0`.*

Nombre de paramètres. $\underbrace{Td}_{\mathbf{W}_{te}} + \underbrace{Bd}_{\mathbf{W}_{pe}} + \underbrace{Td}_{\mathbf{W}_{lm}} + N \left(\underbrace{4d^2}_{\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{O}} + \underbrace{8d^2}_{\mathbf{W}_1, \mathbf{W}_2} \right)$. Avec $T = 27$, $B = 16$,

$d = 16$ et $N = 1$, cela fait 4192 paramètres, à comparer aux $T^2 = 729$ paramètres de la régression logistique.

A Rappels de probabilité

A.1 Loi discrète sur un espace fini

Soit X une variable aléatoire à valeurs dans un ensemble fini $\mathcal{E} = \{e_1, \dots, e_T\}$. Sa loi est entièrement décrite par le vecteur des probabilités $p_k = P(X = e_k)$, qui vérifie

$$p_k \geq 0 \quad \text{pour tout } k, \quad \sum_{k=1}^T p_k = 1.$$

Un tel vecteur vit dans le **simplexe** de $\mathbb{R}^T$. C'est exactement la sortie de softmax : tout apprendre revient à connaître ces T nombres pour chaque contexte.

A.2 Probabilité conditionnelle et règle de Bayes

Pour deux événements A, B avec $P(B) > 0$,

$$P(A | B) = \frac{P(A \cap B)}{P(B)}, \quad \text{d'où} \quad P(A \cap B) = P(B) P(A | B).$$

En itérant (règle des probabilités composées / chaîne), pour des variables $t_0, \dots, t_n$:

$$P(t_0, \dots, t_n) = P(t_0) P(t_1 | t_0) P(t_2 | t_0, t_1) \cdots P(t_n | t_0, \dots, t_{n-1}) = \prod_{j=0}^n P(t_j | t_{<j}),$$

avec la notation $t_{<j} = (t_0, \dots, t_{j-1})$. Cette factorisation est *exacte* et ne suppose aucune indépendance : c'est elle qui justifie qu'on ne modélise qu'une loi conditionnelle « prochain token sachant le passé ».

A.3 Indépendance et échantillon i.i.d.

Des variables $X_1, \dots, X_N$ sont **indépendantes** si leur loi jointe se factorise :

$$P(X_1, \dots, X_N) = \prod_{i=1}^N P(X_i).$$

Elles sont **i.i.d.** (indépendantes et identiquement distribuées) si de plus elles suivent toutes la même loi. On suppose que les N prénoms de la base sont i.i.d. : c'est ce qui transforme la vraisemblance en un simple produit, puis (par $-\log$) en une somme.

A.4 Maximum de vraisemblance et log-vraisemblance

Étant donné des données $x_1, \dots, x_N$ et un modèle p_θ , la **vraisemblance** est $\theta \mapsto \prod_i p_\theta(x_i)$. On préfère la minimiser sous forme de **log-vraisemblance négative** :

$$\theta^* = \arg \max_{\theta} \prod_{i=1}^N p_\theta(x_i) = \arg \min_{\theta} \left(- \sum_{i=1}^N \log p_\theta(x_i) \right),$$

car $x \mapsto -\log x$ est strictement décroissante (l'arg max est préservé) et transforme un produit numériquement instable en une somme.

B Rappels de convexité et optimisation convexe

B.1 Ensemble et fonction convexes

Un ensemble $C \subseteq \mathbb{R}^d$ est **convexe** si pour tous $x, y \in C$ et $\lambda \in [0, 1]$, $\lambda x + (1 - \lambda)y \in C$. Une fonction $f : C \rightarrow \mathbb{R}$ (avec C convexe) est **convexe** si

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y), \quad \forall x, y \in C, \forall \lambda \in [0, 1].$$

Géométriquement : la corde est au-dessus du graphe.

B.2 Caractérisations différentiables

Si f est dérivable, f est convexe si et seulement si son graphe est au-dessus de chacun de ses plans tangents :

$$f(y) \geq f(x) + \nabla f(x)^\top (y - x), \quad \forall x, y.$$

Si f est deux fois dérivable, f est convexe si et seulement si sa **hessienne est semi-définie positive** en tout point :

$$\nabla^2 f(x) \succeq 0 \iff u^\top \nabla^2 f(x) u \geq 0 \quad \forall u.$$

Elle est **strictement** convexe si $\nabla^2 f(x) \succ 0$ (définie positive), ce qui garantit l'unicité du minimiseur.

B.3 Opérations qui préservent la convexité

- **Somme** et combinaison à coefficients positifs : si f, g convexes et $\alpha, \beta \geq 0$, alors $\alpha f + \beta g$ est convexe.
- **Composition avec une application affine** : si f est convexe et $Ax + b$ est affine, alors $x \mapsto f(Ax + b)$ est convexe.
- **Supremum** : un sup (même infini) de fonctions convexes est convexe.

Ces trois règles suffisent à établir la convexité du coût de la §4 : $\ell = g(Mv)$ avec g convexe et $M \mapsto Mv$ linéaire (règle 2), puis somme sur les exemples (règle 1).

B.4 Deux fonctions convexes utiles

- $x \mapsto e^x$ et $x \mapsto -\log x$ sont convexes sur leur domaine.
- La fonction **log-sum-exp** $z \mapsto \log(\sum_{i=1}^T e^{z_i})$ est convexe sur $\mathbb{R}^T$; sa hessienne vaut $\text{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top$ avec $\mathbf{p} = \text{softmax}(\mathbf{z})$ (calcul de la §4.2).

B.5 Pourquoi la convexité aide

Pour une fonction convexe, **tout minimum local est un minimum global**, et il n'existe pas de point-selle piégeant. La descente de gradient (avec un pas assez petit) converge alors vers un optimum global. C'est ce qui rend la régression logistique « facile » à entraîner — propriété que l'on *perd* dès le jour 2, la fonction de coût d'un transformer n'étant pas convexe.

B.6 Adam comme approximation de la Hessienne

On rappelle l'algorithme Adam : ...

Le lien passe par la moyenne glissante v_t des carrés du gradient (la moyenne m_t ne sert qu'à lisser, elle n'intervient pas ici). On note g_t le gradient stochastique à l'étape t , et $\ell(\theta; x) = -\log p_\theta(x)$ la log-vraisemblance négative d'un échantillon.

- v_t estime un moment d'ordre 2 du gradient. La récurrence $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ est une moyenne pondérée des g_s^2 sur les $\approx 1/(1 - \beta_2)$ dernières itérations (≈ 100 pour $\beta_2 = 0.99$). C'est donc, coordonnée par coordonnée, un estimateur de $\mathbb{E}[g_i^2]$, l'espérance portant sur le tirage des échantillons. La moyenne glissante remplace l'espérance, et l'oubli exponentiel permet de suivre le fait que la hessienne change quand θ évolue. Attention, ce n'est pas exactement une « variance » : $\mathbb{E}[g_i^2] = \text{Var}(g_i) + \mathbb{E}[g_i]^2$. C'est une variance seulement lorsque le gradient moyen est proche de 0, c'est-à-dire près d'un minimum.
- le moment d'ordre 2 du gradient est la hessienne (identité de l'information). Si x est tiré selon le modèle p_θ , alors

$$\mathbb{E}[\nabla_\theta \ell \nabla_\theta \ell^\top] = \mathbb{E}[\nabla_\theta^2 \ell].$$

En effet, $\sum_x p_\theta(x) = 1$ donne, en dérivant une fois, $\sum_x p_\theta \nabla \log p_\theta = 0$, puis en dérivant une seconde fois (avec $\nabla p_\theta = p_\theta \nabla \log p_\theta$)

$$0 = \sum_x p_\theta (\nabla \log p_\theta \nabla \log p_\theta^\top + \nabla^2 \log p_\theta) = \mathbb{E}[\nabla \ell \nabla \ell^\top] - \mathbb{E}[\nabla^2 \ell].$$

On peut le vérifier sur notre modèle. Pour les logits, le gradient est $p - \mathbf{1}_t$ avec $t \sim p$, donc

$$\mathbb{E}[(p - \mathbf{1}_t)(p - \mathbf{1}_t)^\top] = pp^\top - pp^\top - pp^\top + \text{diag}(p) = \text{diag}(p) - pp^\top = H,$$

qui est bien la hessienne calculée plus haut. Pour la matrice M , le gradient est $g_{ij} = (p_i - \mathbb{1}[t = i]) v_j$, d'où

$$\mathbb{E}[g_{ij}^2] = \mathbb{E}[(p_i - \mathbb{1}[t = i])^2] v_j^2 = p_i(1 - p_i) v_j^2 = H_{ii} v_j^2,$$

et $H_{ii} v_j^2$ est exactement la dérivée seconde $\partial^2 \mathcal{L}_{\text{token}} / \partial M_{ij}^2$. Ainsi, ce que calcule v_t est (une estimation de) **la diagonale de la hessienne**.

En pratique, la cible t n'est pas tirée selon le modèle mais selon les données. Ce que l'on estime est alors la *Fisher empirique*, qui coïncide avec la hessienne seulement lorsque $p_\theta \approx p_{\text{données}}$, c'est-à-dire près de l'optimum si le modèle est bien spécifié. Ailleurs, c'est une heuristique.

- de la diagonale au pas de Adam. Si l'on ne garde que la diagonale de la hessienne, le pas de Newton $H^{-1}g$ devient, coordonnée par coordonnée, g_i/H_{ii} . Adam utilise $g_i/\sqrt{v_i}$ avec $v_i \approx H_{ii}$, soit $H_{ii}^{-1/2}$ au lieu de H_{ii}^{-1} : on normalise par la courbure seulement « à moitié ». Cela donne un pas plus prudent, ce qui est utile quand l'estimation de la courbure est bruitée ou quand la fonction n'est pas convexe. Les directions très courbes (H_{ii} grand) sont freinées et les directions plates accélérées, comme dans la méthode de Newton.
- Limites. Il s'agit d'une analogie, pas d'une équivalence : Adam ignore les termes hors diagonale de la hessienne (les corrélations entre paramètres), ce qui est précisément ce qui donne à Newton sa convergence quadratique, et il utilise la racine carrée de la courbure.

Nous n'implémenterons pas cette méthode en TP car nous voulons rester sur une implémentation en pur python, et cela nécessiterait d'implémenter la résolution de système linéaire. Ce n'est pas inintéressant, mais cela sort du cadre de ce cours.

B.7 Inégalité de Jensen

Aussi appelée la plus chic des inégalités fonctionnelles. Si f est convexe et X une variable aléatoire intégrable, alors

$$f(\mathbb{E}[X]) \leq \mathbb{E}[f(X)].$$