ChatGPT – partie 1

Comment est-il possible pour un ordinateur de générer un texte aussi cohérent que des phrases écrites par un humain ? Nous commençons par expliquer le principe général. Puis, partant de statistiques sur les mots d'un texte, nous expliquons les concepts de token et de plongement.

1. Qu'est-ce que *ChatGPT* ?

Commençons par demander à *ChatGPT* (version 3.5) de « Faire une courte introduction sur le principe de fonctionnement de *ChatGPT* avec l'accent sur les aspects mathématiques. » Voici sa réponse :

> « *ChatGPT fonctionne grâce à une architecture de réseau de neurones appelée GPT (Generative Pre-trained Transformer). Cette architecture utilise des opérations linéaires et non linéaires, notamment des couches d'attention, pour traiter l'information. Le modèle est massivement pré-entraîné sur des données textuelles, utilisant des techniques mathématiques avancées telles que la rétropropagation du gradient et l'optimisation par descente de gradient.* »

C'est une réponse très impressionnante. Il y a plein de termes que nous connaissons : réseau de neurones, rétropropagation, descente de gradient. Il y a d'autres concepts qu'il va falloir étudier, ce travail va être divisé en deux parties :

- la première partie est consacrée aux données textuelles avec les notions de *token* et de *plongement*,
- la seconde partie sera consacrée au type de réseau de neurones adapté aux données textuelles avec les notions de *transformer* et d'*attention*.

Le domaine d'étude de la génération automatique de texte s'appelle le *traitement automatique du langage naturel* (NLP, *Natural Language Processing*). Nous allons nous concentrer sur les LLM (*Large Language Model*) qui ont eu un grand succès ces dernières années grâce à *ChatGPT* et ses variantes. *ChatGPT* est un robot conversationnel basé sur le modèle de langage *GPT3*. Dans ces chapitres nous allons en expliquer le principe général. Pour les exemples, on s'appuiera sur *GPT2* (dont les paramètres sont publiquement accessibles) et sur un autre modèle appelé *BERT*. On construira aussi nous-mêmes des générateurs de textes élémentaires. Ces techniques étant nouvelles et essentiellement développées outre-atlantique nous ne chercherons pas à traduire tous les termes. Par exemple le mot *token*, se traduirait dans notre contexte par *portion* ou bien *tronçon*. De plus, nos exemples linguistiques seront basés sur l'anglais car les ressources sont beaucoup plus facilement accessibles.

1.1. Un exemple de génération de texte

Demandons à *ChatGPT* (plus précisément à *GPT2*) de continuer la phrase suivante :

This dog is ...

Le mot le plus probable est « **a** ». On demande donc ensuite à la machine de compléter la phrase

This dog is a ...

Le mot le plus probable est « **great** » :

This dog is a great ...

On continue ainsi jusqu'à obtenir une phrase complète :

This dog is a great companion
This dog is a great companion for
This dog is a great companion for me
This dog is a great companion for me.

En fait, le modèle calcule pour chacun des 50 000 tokens la probabilité que ce soit le mot suivant. Pour compléter notre phrase **This dog is ...** voici les cinq mots les plus probables :

This dog is	**a**	11.8 %
	very	4.7 %
	not	3.6 %
	so	2.9 %
	an	2.2 %

Une fois que le « **a** » a été choisi, on calcule les nouvelles probabilités pour le mot suivant :

This dog is a	**great**	4.5 %
	very	4.0 %
	good	3.0 %
	bit	2.8 %
	little	1.8 %

Pour construire des phrases plus variées, on utilise un paramètre appelé *température* qui contrôle la variabilité de la complétion. Prenons par exemple à chaque itération, le troisième mot le plus probable, on obtient :

This dog is not going away.

La variante de *GPT* la plus célèbre est bien sûr *ChatGPT* qui répond à vos questions. Nous ne discuterons pas spécifiquement de *ChatGPT*, mais répondre à une question c'est comme compléter un texte.

1.2. Principe général

Voici les grandes étapes qui seront étudiées dans ce chapitre ainsi que le suivant afin de compléter une phrase.

- **Tokenisation.** Il s'agit de découper un texte en une suite de mots ou de parties de mots, appelés *tokens*. Chaque token est codé par un numéro. (Il y a $N = 50\,257$ tokens pour *GPT2*.)
- **Plongement.** (*Embedding.*) Chaque token (ou chaque mot si vous préférez) est transformé en un vecteur de très grande taille. (Pour *GPT2* un vecteur token v est un vecteur de $\mathbb{R}^n$ avec $n = 768$.) De plus, on fait en sorte que dans ce vecteur soit aussi encodée la position du token dans la phrase. Ainsi une phrase (composée de K tokens) est codée par une liste de vecteurs $(v_1, v_2, \ldots, v_K)$, c'est-à-dire par une matrice $M \in M_{n,K}$ (n lignes, K colonnes).
- **Retranscription.** (*Unembedding.*) À la fin, en sortie du réseau, on obtient un vecteur $w \in \mathbb{R}^n$. On pourrait chercher parmi les $N = 50\,257$ vecteurs tokens, lequel est le plus proche (la distance entre deux vecteurs est définie par le produit scalaire ou la *similarité cosinus*). On préfère attribuer une probabilité de correspondance à chacun des N tokens et, par exemple, choisir le mot suivant parmi les 10 plus probables.
- **Transformeur.** (*Transformer.*) L'architecture des réseaux de neurones pour les grands modèles de langages (LLM) se nomme transformeur. En son cœur on trouve deux types de couches. Le premier type est simplement composé de couches denses de neurones (MLP, *Multi Layer Perceptron*, sans convolution) comme on a déjà rencontrées dans les chapitres précédents.

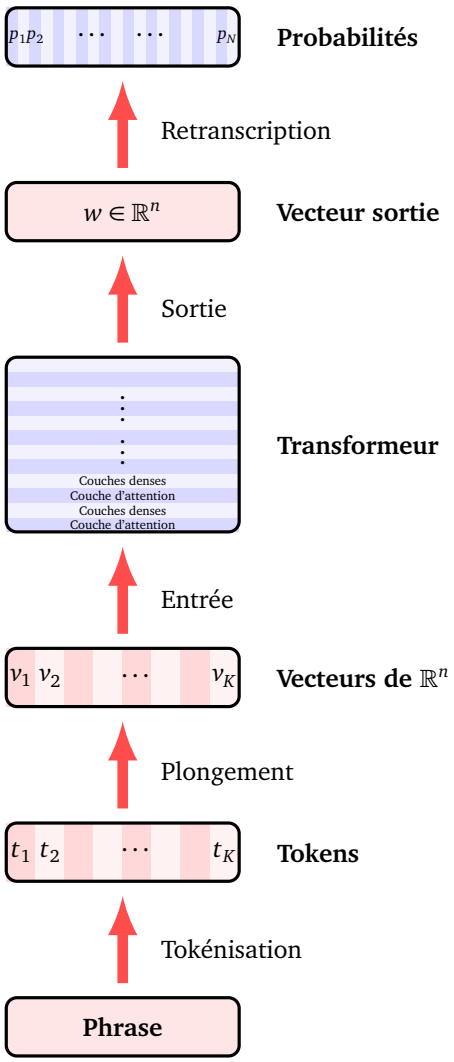
- **Attention.** Les couches de second type sont les couches d'attention qui ont fait le succès des LLM après l'article fondateur *Attention is all you need* par une équipe de *Google*. Dans une phrase tous les mots n'ont pas la même importance pour comprendre son sens. L'attention consiste à attribuer des poids aux mots en fonction de leur pertinence dans la phrase. Par exemple, en oubliant les articles ou les adjectifs inutiles, afin de mettre en évidence la structure sujet/verbe/complément. En fait à chaque mot/token de la phrase on associe une liste des poids aux autres mots de la phrase qui explique le sens du mot dans cette phrase. On détaillera le mécanisme d'attention dans le chapitre suivant.
- **Réseau.** Le réseau est gigantesque. Par exemple, le modèle *GPT2-Small* que l'on étudiera possède 117 millions de poids. Il est composé de 24 couches : 12 couches de neurones et 12 couches d'attention (chacune avec 12 têtes d'attention). Mais surtout ce réseau est entraîné sur un corpus gigantesque basé sur plusieurs gigaoctets de documents (livres, articles, pages web, forum…) principalement en anglais. Le paramétrage du réseau nécessite énormément de calculs, il est donc indispensable de posséder des algorithmes performants, des ordinateurs puissants travaillant en parallèle avec en leur cœur des processeurs graphiques (GPU) adaptés à ce type de calculs.
- **Flux résiduel.** En plus de l'attention, il y a une différence importante avec les réseaux des chapitres précédents : le flux résiduel (*residual stream*). L'idée est de garder la mémoire de ce que chaque couche a appris afin que toutes les couches suivantes (même les couches éloignées) aient un accès direct aux résultats importants déjà obtenus. Les données envoyées d'une couche à l'autre sont déterminées par les coefficients de matrices qui sont des paramètres du réseau. Le flux résiduel remédie en particulier à un problème des réseaux ayant beaucoup de couches pour lesquels la rétropropagation modifie très lentement les poids des premières couches.
- **Dimensions.** Une des particularité de LLM est de travailler avec des espaces de grandes dimensions (par exemple dans $\mathbb{R}^{768}$ pour *GPT2*), cela permet d'avoir « de la place », le réseau peut ainsi se réserver des sous-espaces pour envoyer des informations d'une couche à l'autre via le flux résiduel. La contrepartie est que le nombre de poids du réseau devient gigantesque.

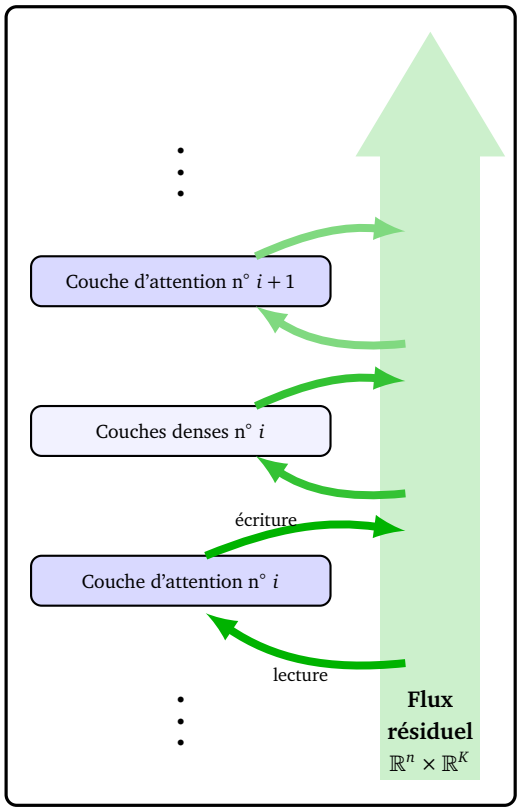
1.3. Architecture du réseau

Voici l'architecture de *GPT2-Small* dont le modèle est publiquement accessible ainsi que ses poids après entraînement.

- Nombre de tokens $N = 50\,257$.
- Dimension de plongement $n = 768$.
- Une phrase est découpée en $K = 1024$ tokens maximum.
- Nombre de couches d'attention : 12 (avec chacune 12 têtes d'attention).
- Nombre de couches denses : 24 (12 fois une paire de couches).
- Nombre total de poids du réseau : 117 millions.
- Entraîné sur un corpus de plusieurs dizaines de gigaoctets de textes variés.

Ci-dessous à gauche l'architecture globale du réseau, et à droite plus de détails pour la partie « transformeur » qui est le cœur du réseau.





Transformeur

Depuis, les progrès ont été constants. Les réseaux sont de plus en plus grands (avec plusieurs milliards de poids), entraînés dans toutes les langues possibles (y compris les langages informatiques). Ils peuvent être spécialisés (*fine tuning*) pour répondre à une tâche spécifique (assistance internet, classification de documents, génération de code *Python*…). Nul doute qu'ils vont encore beaucoup progresser.
Notre modeste but est de comprendre le principe de fonctionnement global en donnant un nombre raisonnable de détails. Le travail s'étend sur deux chapitres. Il suppose la connaissance des couches denses de neurones.

2. Statistique et linguistique

Nous allons voir comment l'étude des fréquences des mots dans les textes permet de générer des phrases qui ressemblent à des phrases cohérentes.

2.1. La loi de Zipf-Mandelbrot

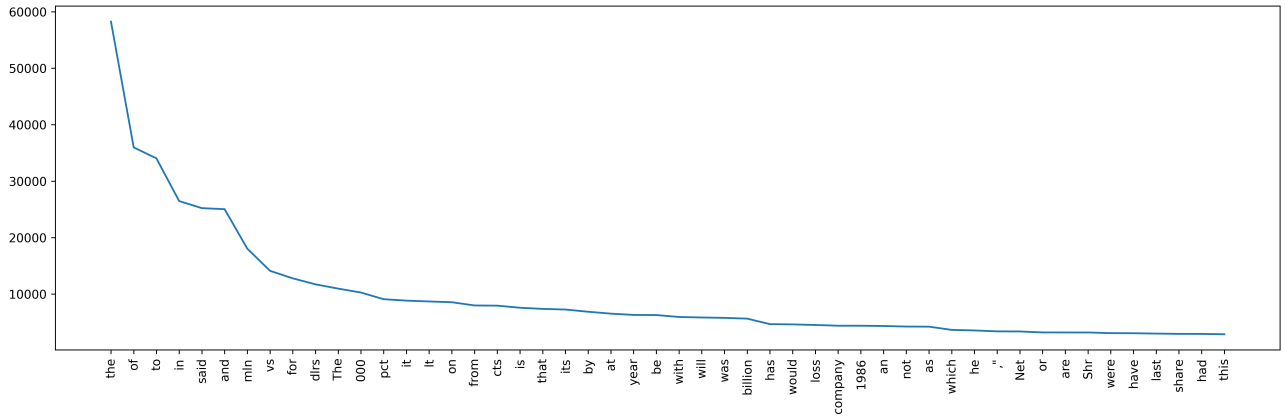
Commençons par étudier la fréquence des mots dans un (grand) texte. Prenons un ensemble de dépêches de l'agence *Reuters* qui fournit des informations économiques et politiques. Ce *corpus* est constitué de plus de 10 000 dépêches en anglais qui forment un total de plus 1.7 millions de mots. Voici un extrait :

The country's oil import bill, however, fell 23 pct in the first quarter due to lower oil prices.

Voici les 10 mots (de deux lettres ou plus) les plus fréquents et leur nombre d'occurrences :

the	58251	**and**	25043
of	35979	**mln**	18037
to	34035	**vs**	14120
in	26478	**for**	12785
said	25224	**dlrs**	11730

Voici le graphique des nombres d'occurrences des 50 mots les plus fréquents.



Quelle formule régit ce nombre d'occurrences ? Une règle simple que l'on constate est que si le mot le plus fréquent apparaît N fois, alors le deuxième mot le plus fréquent apparaît $N/2$ fois, le troisième mot le plus fréquent $N/3$… Ainsi la répartition des occurrences en fonction du rang se fait selon les termes :

$$1 \qquad \frac{1}{2} \qquad \frac{1}{3} \qquad \frac{1}{4} \qquad \dots$$

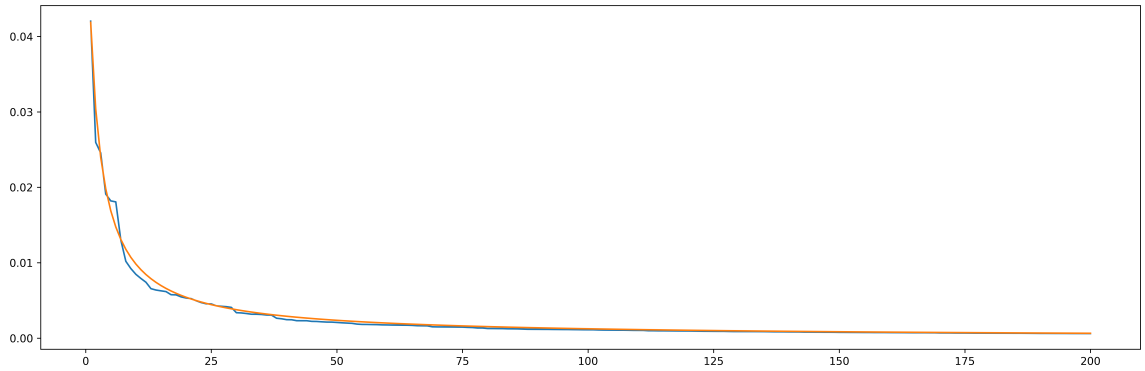
Cette observation expérimentale se retrouve dans n'importe quel grand texte, quelle que soit sa langue. Elle se décline avec plus de précision dans la ***loi de Zipf-Mandelbrot*** :

$$f_i = \frac{a}{(i+b)^c}$$

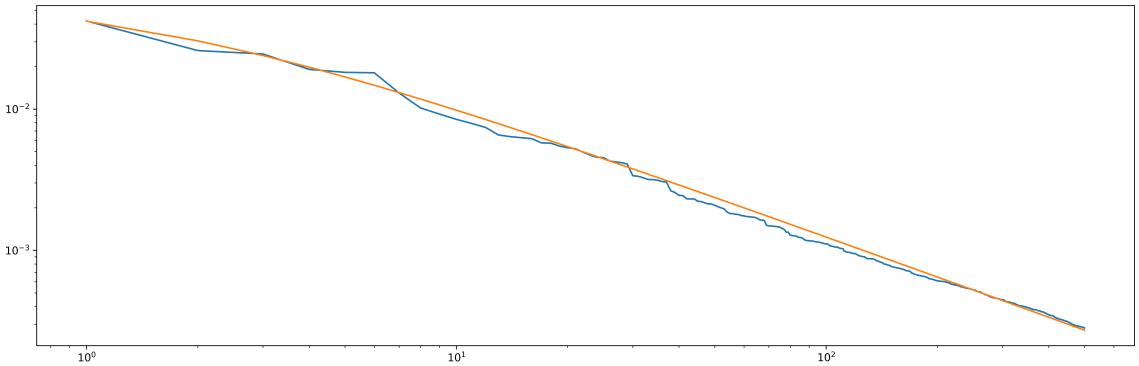
où :

- $i \geqslant 1$ est le rang du mot concerné,
- f_i est sa fréquence (c'est-à-dire le nombre d'occurrences du mot divisé par le nombre total de mots),
- a, b, c sont des constantes réelles à déterminer (avec $c \simeq 1$).

Pour nos dépêches, voici le graphique des 200 premières occurrences ainsi que le graphe de la fonction $x \mapsto \frac{a}{(b+x)^c}$ avec ici $a = 0.1$, $b = 1.5$, $c = 0.95$:



Il est plus facile de visualiser l'ajustement en utilisant une échelle logarithmique (ici sur les 500 mots les plus fréquents).



En effet si $y = \frac{a}{(b+x)^c}$ alors $\ln(y) = \ln(a) - c\ln(b+x)$, en posant $Y = \ln(y)$ et $X = \ln(b+x)$ on trouverait l'équation $Y = \ln(a) - cX$ d'une droite. Comme nous traçons $\ln(y)$ en fonction de $\ln(x)$ (et pas de $\ln(b+x)$) nous n'obtenons pas exactement le tracé d'une droite.
Retenons que les mots d'un texte n'apparaissent pas tous avec la même fréquence et que la loi de Zipf-Mandelbrot est une formule empirique qui modélise ces fréquences.

2.2. Co-occurrence

La fréquence des mots d'une langue ne permet pas de générer des phrases qui paraissent naturelles. Par contre, compter les paires de mots consécutifs va nous permettre de générer des phrases pseudo-naturelles.
La *co-occurrence* d'une paire de mot (mot1, mot2) est le nombre de fois où mot1 est immédiatement suivi de mot2 dans un texte. Plus généralement la co-occurrence de (mot1, mot2, …, mot$n$) est le nombre de fois où ces mots apparaissent en se suivant et dans cet ordre.
Par exemple dans les dépêches précédentes, voici les co-occurrences les plus fortes lorsque le premier mot est **the** :

(**the**, **company**)	1941
(**the**, **dollar**)	878
(**the**, **first**)	810
(**the**, **year**)	704
(**the**, **government**)	620

Cela nous donne une méthode simple permettant de générer une phrase : partant d'un premier mot mot1, on choisit pour deuxième mot celui qui a la plus grande co-occurrence (mot1, mot2). Puis on choisit le mot suivant réalisant la plus grande co-occurrence (mot2, mot3)… On s'arrête lorsque la phrase atteint la longueur voulue.
En commençant par **the** on obtient la phrase :

the company said it has been made by the company

2.3. Softmax et température

Pour générer des phrases variées et naturelles nous allons ajouter un peu d'aléatoire lors de leur construction. Au lieu de choisir comme mot suivant le plus probable, on peut choisir le mot suivant au hasard, disons parmi les 10 plus probables.
La façon la plus évidente de le faire est de calculer les 10 co-occurrences les plus fortes et de choisir un mot au hasard, mais en tenant compte des fréquences. De sorte que, par exemple, après **the**, le mot **company** a deux fois plus de chance d'être choisi que **dollar**.
Voici quelques phrases commençant par **the** que l'on obtient par ce processus aléatoire pondéré :

the end of the company also said it raised to the end assets to acquire
the United States to the world prices will meet the company said it agreed

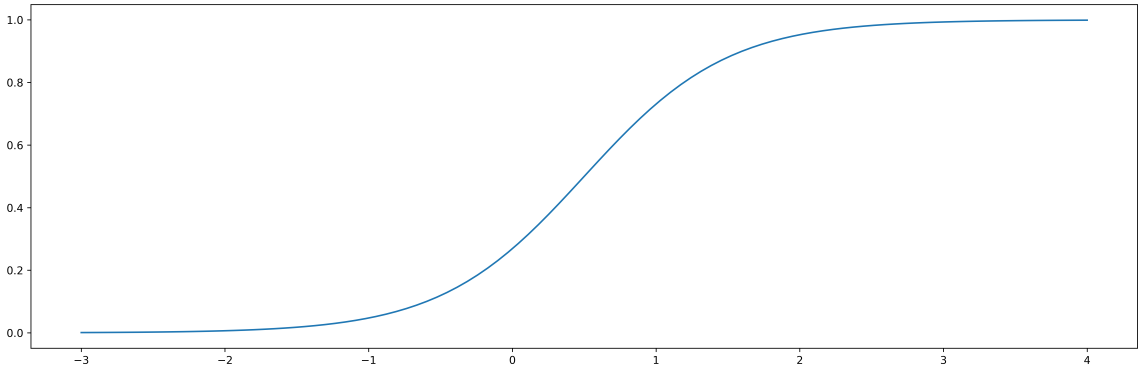
the same month earlier reported 1986 results include nonrecurring costs

Ces phrases n'ont pas de sens mais elles possèdent quand même une tonalité familière.

Nous allons voir une façon de procéder un peu plus sophistiquée. Revenons d'abord sur la fonction softmax. Pour $X = (x_1, x_2, \ldots, x_k)$ on note :

$$\sigma_i = \frac{e^{x_i}}{e^{x_1} + e^{x_2} + \cdots + e^{x_k}}.$$

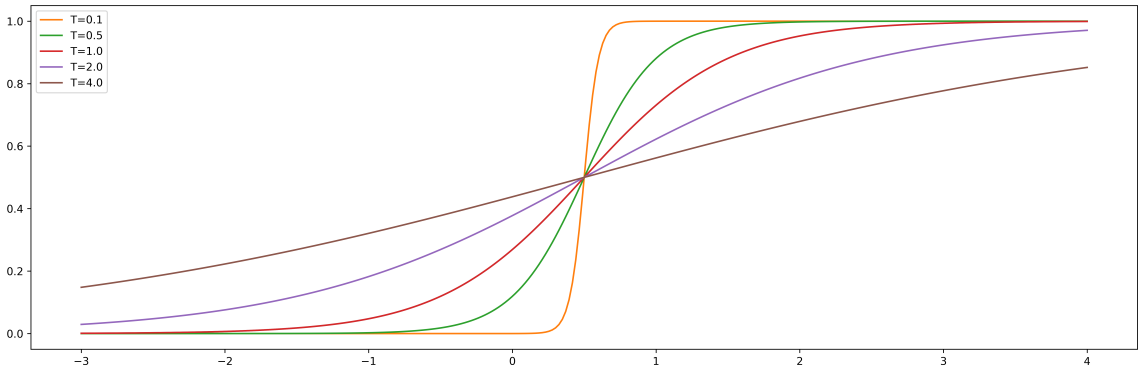
On a $\sigma_1 + \sigma_2 + \cdots + \sigma_k = 1$. On peut interpréter les σ_i comme des probabilités. La valeur de σ_i la plus grande est obtenue pour la valeur la plus grande de x_i. Il s'agit d'une version « lisse » de la fonction argmax (on rappelle que la fonction argmax renvoie le rang pour lequel le maximum est atteint).
Voici le graphe de $\sigma_1(x) = \frac{e^x}{e^x + e^{1-x}}$ pour $X = (x, 1-x)$ et $x \in \mathbb{R}$.



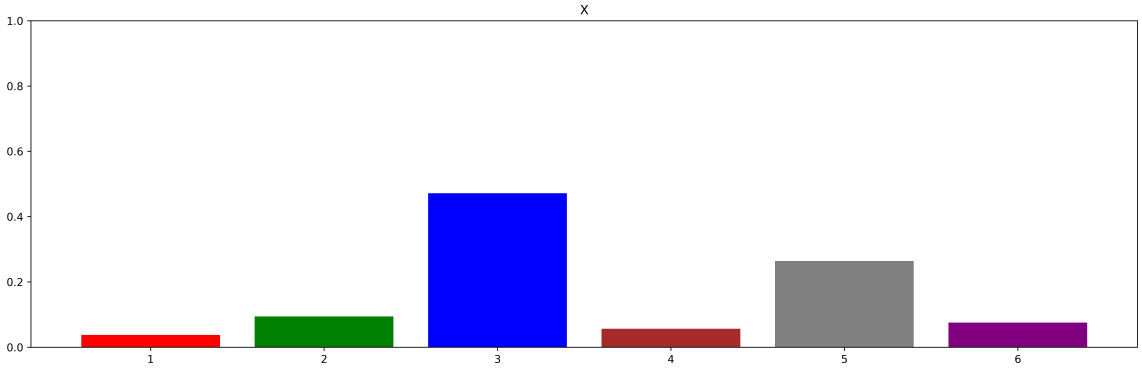
Nous allons rajouter un paramètre, appelé ***température***, à la fonction softmax. Pour $X = (x_1, x_2, \ldots, x_k)$ et $T > 0$, on note :

$$\sigma_{T,i} = \frac{e^{x_i/T}}{e^{x_1/T} + e^{x_2/T} + \cdots + e^{x_k/T}}.$$

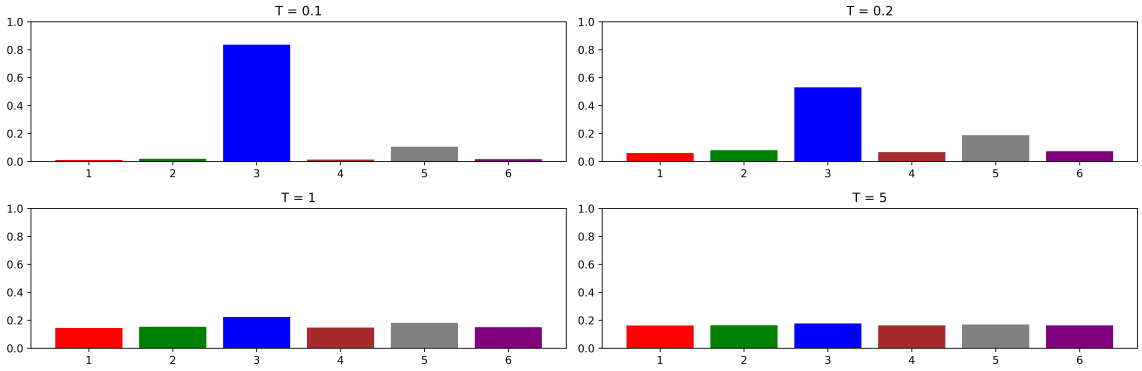
On a de nouveau $\sigma_{T,1} + \sigma_{T,2} + \cdots + \sigma_{T,k} = 1$.
Voici les graphes de $\sigma_{T,1}(x) = \frac{e^{x/T}}{e^{x/T} + e^{1-x/T}}$ pour $X = (x, 1-x)$ et quelques valeurs de T.



Plus la température est proche de 0, plus la fonction softmax se rapproche de ce que ferait une fonction argmax, c'est-à-dire que $\sigma_{T,i}$ vaut 1 là où x_i est le plus grand et 0 ailleurs. Plus la température est élevée, plus la fonction softmax se rapproche d'une distribution uniforme, c'est-à-dire $\sigma_{T,i} \simeq \frac{1}{k}$ (quelles que soient les valeurs $x_1, \ldots, x_k$).
Voici un exemple de distribution aléatoire $X = (x_1, x_2, \ldots, x_6) = \frac{1}{53}(2, 5, 25, 3, 14, 4)$.



Voici les valeurs renvoyées par $\sigma_{T,i}(X)$ pour différentes températures. Pour $T = 0.1$ la valeur x_i la plus élevée est renforcée. Pour $T = 0.2$ on retrouve ici à peu près la distribution originale. Pour $T = 1$ ou $T = 5$ les valeurs retournées correspondent à des distributions quasi-uniformes.



Le vocable « température » est issu de la physique statistique. Des particules dans un environnement proche du zéro absolu sont quasi-figées. Plus la température augmente plus elles atteignent des vitesses élevées. Pour nous, le paramètre T permet de prendre plus ou moins de liberté pour le choix du mot suivant. Avec une température très proche de 0, on prend à coup sûr pour mot suivant celui le plus fréquent, avec une température élevée, on prend pour mot suivant un mot au hasard parmi les k plus fréquents avec une probabilité $1/k$.
Voici des exemples de phrases commençant par **the** obtenues pour différentes températures (le choix se fait à chaque fois parmi les 20 mots suivants les plus probables). Pour $T = 0.001$ c'est toujours la même successions de mots :

the company said it has been made by the company

Voici trois phrases obtenues avec $T = 0.05$:

the company said the next week ended March 31
the company said it said it acquired by the second quarter
the end of the company said it will be held by the dollar

$T = 0.5$:

the dollar down slightly short notice about 18 pct this level
the world trade deficit and Trade Representative to help reduce debt

$T = 10$:

the new business for delivery to an attempt was due for comment
the year earlier it expects oil reserves are in January 28

2.4. Trigrammes

On appelle ***bi-gramme*** une suite (mot1, mot2) de deux mots consécutifs. On appelle ***tri-gramme*** une suite de 3 mots consécutifs et plus généralement un n-gramme une suite de n mots consécutifs. La co-occurrence, c'est exactement compter les bi-grammes. Ce que nous avons fait précédemment, c'est utiliser ces valeurs pour choisir le mot2 qui doit suivre le mot1. Améliorons notre méthode en comptant d'abord les tri-grammes, c'est-à-dire en comptant le nombre d'occurrences de tous les triplets (mot1, mot2, mot3). Une fois ce travail effectué, étant donné une paire (mot1, mot2) on peut décider quels sont les mot3 les plus probables.
Voici des phrases commençant par **the** puis **company** et complétées par cette méthode (la température est $T = 1$, le mot3 qui suit les deux mots précédents est choisi parmi les 20 les plus probables) :

the company expects its overall retail income figures
the company reported earnings from many countries to open the Japanese Government
the company with land holdings and production has become definitive

Les phrases obtenues sont un peu plus naturelles que précédemment. Nous avons compris une idée clé de la génération de texte par ordinateur : à partir d'un début de phrase, l'algorithme décide du mot suivant, puis recommence à partir de cette nouvelle liste de mots, jusqu'à obtenir une phrase complète. Il s'agit maintenant d'améliorer la qualité des phrases obtenues.

3. Tokens

3.1. Vocabulaire

Commençons par établir la terminologie :

- Le ***corpus*** est l'ensemble des textes utilisés pour établir le modèle de prédiction. Ce corpus doit être le plus grand possible (plusieurs millions ou milliards de mots) mais aussi de bonne qualité puisque ce sont les statistiques issues de ces textes qui permettront de générer de nouvelles phrases.
- Les textes sont découpés en ***phrases*** (que l'on peut définir en disant qu'elles commencent par une majuscule et finissent par un point). Les phrases sont composées de ***mots*** (les mots sont séparés par des espaces). Chaque mot est formé de ***caractères*** (par exemple, les lettres **a**, **b**,…, **z** mais aussi les chiffres, la ponctuation).

Pour compléter un début de phrase, on va prédire une suite probable. La prédiction se fait mot par mot. Nous allons à l'avenir utiliser un découpage différent en utilisant des *tokens*.
Un ***token*** (une ***portion*** ou un ***tronçon***) est une suite de caractères qui correspond à une partie de mot. Chaque modèle de langage fait son choix de tokens. L'ensemble des tokens choisis constitue le ***vocabulaire***.
Le modèle *GPT2* utilise 50 257 tokens, chacun est numéroté. En voici quelques uns :

1009	**ution**	1015	**ave**
1010	**ters**	1016	**␣going**
1011	**␣take**	1017	**␣sl**
1012	**␣Cl**	1018	**ug**
1013	**␣conf**	1019	**␣Americ**
1014	**way**		

Noter que dans ce modèle les majuscules et minuscules sont distinguées et que l'espace (ici notée par ␣) est prise en compte pour signifier que le token est un début de mot. Par exemple **Man**, **man**, **␣Man** et **␣man** sont quatre tokens différents pour *GPT2*.

3.2. À quoi servent les tokens ?

- Les tokens servent à transformer une suite de mots en une liste de nombres. Par exemple la phrase :

  mathematics is the queen of the sciences

  se transforme en la liste de tokens :

  [**mat**, **hemat**, **ics**, **␣is**, **␣the**, **␣queen**, **␣of**, **␣the**, **␣sciences**]

  qui à son tour se transforme en une liste d'entiers (les *tokens id*) :

  [6759, 10024, 873, 318, 262, 16599, 286, 262, 19838]

  Il s'agit d'un encodage sans perte d'information. À partir de la liste des tokens id, on retrouve les tokens et donc la phrase originale.
- Les tokens permettent de maîtriser la taille du vocabulaire. Par exemple le *English Oxford Dictionnary* qui fait référence en langue anglaise, contient plus de 250 000 entrées. Chaque mot fréquemment utilisé sera codé par un unique token, alors que les mots plus rares sont découpés en deux tokens ou plus. Il faut interpréter les tokens comme un intermédiaire entre un caractère et un mot complet :

  caractère $\preccurlyeq$ token $\preccurlyeq$ mot
- Les tokens peuvent faciliter un apprentissage logique des termes, par exemple en séparant la racine d'un mot et sa terminaison. Ce n'est cependant pas la motivation principale. Voici des exemples (fictifs) :

  learning = learn + ing
  surfing = surf + ing
  learner = learn + er
  surfer = surf + er

GPT2 a été entraîné sur un corpus d'environ 8 milliards de tokens (en anglais uniquement), *GPT3.5* avec 500 milliards de tokens (dans plus de 100 langues différentes).

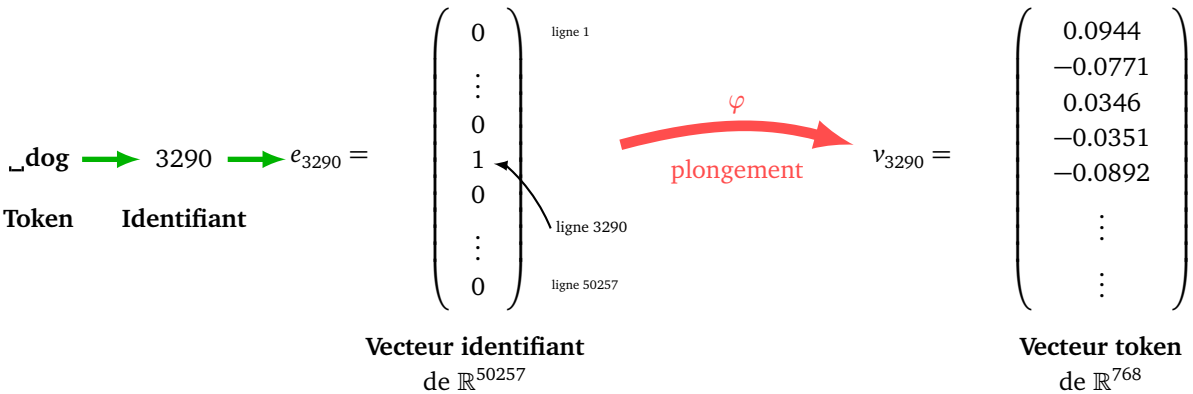
3.3. Différents codages

Nous allons résumer ici les différentes façons de coder un token parmi les N tokens :

- ***token*** : une suite de caractères,
- ***l'identifiant token*** (*token id*) : l'entier i avec $1 \leqslant i \leqslant N$ correspondant au rang du token dans la liste des tokens,
- ***vecteur identifiant token*** (*token id vector*) : le vecteur e_i de la base canonique de $\mathbb{R}^N$ (où i est l'identifiant du token).

Il ne faut pas confondre le vecteur identifiant e_i avec son image $v_i = \varphi(e_i) \in \mathbb{R}^n$ obtenue par plongement (voir plus loin). C'est ce vecteur image v_i qui sera très important pour la suite, il sera appelé le ***vecteur token*** (*token vector/hidden state vector/embedding vector*).

Exemple.
Dans le modèle *GPT2* on a $N = 50\,257$ et $n = 768$.



Pour le token **␣cat**, dont l'identifiant est 3797, le vecteur token débute par : $v_{3797} = j \begin{pmatrix} 0.0099 \\ 0.0365 \\ 0.1640 \\ -0.2185 \\ 0.0285 \\ \vdots \end{pmatrix}$.

3.4. Algorithme de tokenisation

Compression. La tokenisation a été inventée afin de compresser un texte sans perte d'information. On considère qu'en moyenne un token regroupe 4 caractères. Et, également en moyenne, 4 tokens forment 3 mots.

Exemple.
Si on considère un texte de 1000 caractères codés en ASCII, alors la taille mémoire nécessaire est 1000 octets. La tokenisation transforme ce texte en une liste d'environ 250 tokens, chaque token est codé sur 2 octets (pour stocker son identifiant entre 0 et 50 257). Ce qui donne une utilisation de 500 octets, soit une compression de 50% sans perte d'information.

Pour nous, la tokenisation a pour principal but de fixer le nombre de mots du vocabulaire à une valeur souhaitée. Les mots rares ou inconnus sont alors découpés en plusieurs tokens.

Expliquons l'algorithme BPE (*Byte Pair Encoding*) qui permet d'obtenir une liste de tokens de taille voulue. Les données initiales sont :
- un texte (qui est le texte à compresser ou qui est le texte des données d'apprentissage),
- un vocabulaire initial qui est généralement la liste des caractères utilisés,
- un nombre F de fusions (*merge*) souhaitées.

L'algorithme renvoie le vocabulaire (la liste de tous les tokens) et le texte compressé (une liste de tokens).

Algorithme.
- Transformer le texte en une liste de caractères.
- Initialiser le vocabulaire à l'ensemble des caractères du texte.
- Répéter (au plus) F fusions :
  — Chercher la paire la plus fréquente de deux éléments du vocabulaire à des positions consécutives dans le texte.
  — Ajouter cette paire fusionnée au vocabulaire et transformer le texte en utilisant cette fusion.
- Renvoyer le vocabulaire et le texte.

Exemple.
Considérons le texte :

ab ab abc dbac bacde

- Ce texte est formé de 5 mots, on représente chaque mot comme une liste de caractères.
  $$\texttt{texte} = [\mathbf{a}, \mathbf{b}], [\mathbf{a}, \mathbf{b}], [\mathbf{a}, \mathbf{b}, \mathbf{c}], [\mathbf{d}, \mathbf{b}, \mathbf{a}, \mathbf{c}], [\mathbf{b}, \mathbf{a}, \mathbf{c}, \mathbf{d}, \mathbf{e}]$$
  Le vocabulaire initial est formé des caractères :
  $$\texttt{vocab} = [\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}]$$
- **Fusion 1.** On cherche dans les mots du texte les occurrences de paires de caractères. La paire la plus fréquente est la paire (**a**, **b**) car **ab** apparaît 3 fois dans les mots (suivi de la paire **ba** et de la paire **ac** ayant chacune 2 occurrences). On ajoute **ab** au vocabulaire et on ajuste le texte par fusion :
  $$\texttt{vocab} = [\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{ab}]$$

$$\texttt{texte} = [\mathbf{ab}], [\mathbf{ab}], [\mathbf{ab}, \mathbf{c}], [\mathbf{d}, \mathbf{b}, \mathbf{a}, \mathbf{c}], [\mathbf{b}, \mathbf{a}, \mathbf{c}, \mathbf{d}, \mathbf{e}]$$

- **Fusion 2.** On cherche dans les mots du texte les occurrences de paires de mots du nouveau vocabulaire. On fusionne ensuite **b** et **a** (2 occurrences de **ba**) :

$$\texttt{vocab} = [\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{ab}, \mathbf{ba}]$$
$$\texttt{texte} = [\mathbf{ab}], [\mathbf{ab}], [\mathbf{ab}, \mathbf{c}], [\mathbf{d}, \mathbf{ba}, \mathbf{c}], [\mathbf{ba}, \mathbf{c}, \mathbf{d}, \mathbf{e}]$$

- **Fusion 3.** La paire du vocabulaire la plus fréquente est maintenant la paire (**ba**, **c**) avec 2 occurrences. On les fusionne :

$$\texttt{vocab} = [\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{ab}, \mathbf{ba}, \mathbf{bac}]$$
$$\texttt{texte} = [\mathbf{ab}], [\mathbf{ab}], [\mathbf{ab}, \mathbf{c}], [\mathbf{d}, \mathbf{bac}], [\mathbf{bac}, \mathbf{d}, \mathbf{e}]$$

- Dans cet exemple, il n'y a plus de paires doublées, on s'arrête là.

Les tokens sont les éléments du vocabulaire que l'on numérote ici de 1 à 7. Le texte tokenisé est alors codé sous la forme :

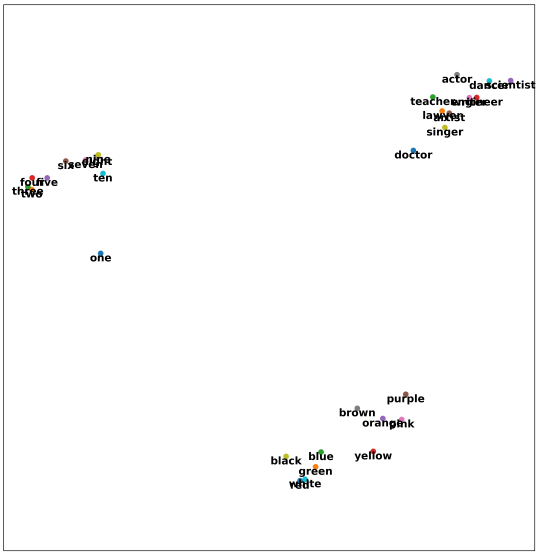
$$\texttt{texte} = [6], [6], [6,3], [4,7], [7,4,5]$$

Pour *GPT2* le vocabulaire initial est l'ensemble des 256 caractères ASCII plus un caractère spécial (EOS, *End Of Sentence*). Le nombre de fusions est $F = 50\,000$. L'algorithme va fournir un total de 50 257 tokens. Il n'y a pas de raison de chercher à augmenter ce nombre de tokens. En effet, pour une langue donnée le nombre de mots utilisés est borné.

4. Plongement

Le but d'un plongement est de regrouper les mots par catégories. Voici un exemple de regroupement que l'on peut obtenir pour les mots suivants :

one, **two**, **three**, **four**, **five**, **six**, **seven**, **eight**, **nine**, **ten**
doctor, **lawyer**, **teacher**, **engineer**, **scientist**, **artist**, **writer**, **actor**, **singer**, **dancer**
red, **green**, **blue**, **yellow**, **orange**, **purple**, **pink**, **brown**, **black**, **white**



On note bien le regroupement par catégories : les chiffres, les professions et les couleurs. Cependant cette représentation dans le plan ne donne qu'une idée partielle du résultat car chaque mot sera en fait représenté par un point (plus exactement par un vecteur) dans un espace de grande dimension.

4.1. Un mot est un vecteur

Plongement. Nous allons transformer un mot (ou plus exactement un token) en un vecteur. Cette opération s'appelle le plongement. Mathématiquement cela correspond à une fonction (une application linéaire) :

$$\begin{array}{rccc} \varphi : & \mathbb{R}^N & \longrightarrow & \mathbb{R}^n \\ & e_i & \longmapsto & v_i \end{array}$$

- N est le nombre total de tokens (par exemple $N = 50\,257$ pour *GPT2* et environ 100 000 pour *GPT3.5* et *GPT4*),
- n est la ***dimension de plongement***, par exemple $n = 768$ pour *GPT2* (et entre 1024 et 2048 pour les modèles plus récents).
- e_i est le vecteur identifiant le token numéro i (avec $1 \leqslant i \leqslant N$), c'est le i-ème vecteur de la base canonique de $\mathbb{R}^N$.
- $v_i = \varphi(e_i)$ est le ***vecteur token***.

Une fois les tokens définis (voir la section précédente) ce plongement est calculé une fois pour toutes.

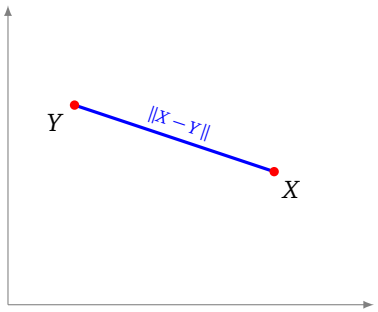
Matrice. Comme φ est une application linéaire, elle est définie par une matrice $A \in M_{n,N}$ ayant n lignes et N colonnes. La i-ème colonne de la matrice A est le vecteur v_i. Cette matrice s'appelle la ***matrice de plongement***. Ainsi, en termes de vecteurs, nous avons simplement :

$$v_i = Ae_i.$$

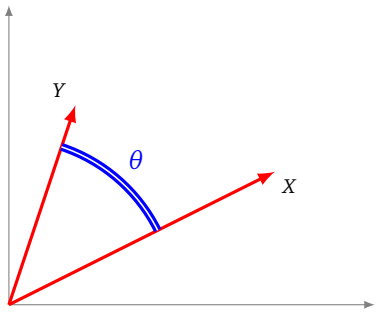
Remarques mathématiques : φ est une application linéaire. Comme on a défini φ sur la base canonique de $\mathbb{R}^N$, φ s'étend par linéarité sur tout $\mathbb{R}^N$, mais nous n'en aurons pas vraiment besoin. D'autre part, le mot plongement n'est pas ici un plongement au sens mathématique usuel.

Point ou vecteur ? On peut à la fois considérer v_i comme un point ou bien comme un vecteur de $\mathbb{R}^n$. Si on considère les éléments de $\mathbb{R}^n$ comme des points, alors la distance entre deux points X et Y est par exemple la ***distance euclidienne*** définie par :

$$\|X - Y\| = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}.$$



Distance euclidienne
$\|X - Y\|$



Similarité cosinus
$\cos(\theta)$

Mais en fait nous allons considérer les éléments de $\mathbb{R}^n$ comme des vecteurs. La ***similarité cosinus*** entre X et Y est le cosinus de l'angle entre ces deux vecteurs :

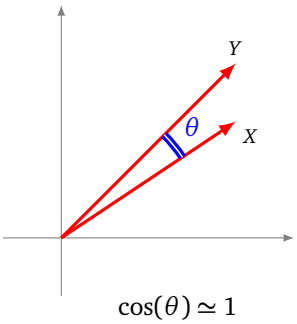
$$S_{\cos}(X, Y) = \cos(\theta) = \frac{\langle X \mid Y \rangle}{\|X\| \, \|Y\|}.$$

On a $\cos(\theta) \in [-1, 1]$, ce n'est pas une distance au sens mathématique usuel.
Cette formule provient d'une formule qui relie le produit scalaire à l'angle entre deux vecteurs :

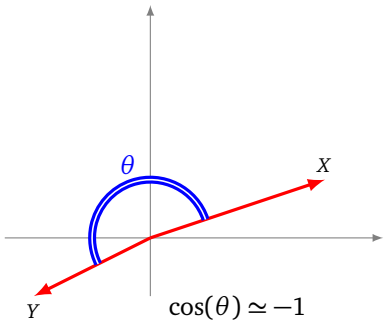
$$\langle X \mid Y \rangle = \|X\| \, \|Y\| \cos(\theta)$$

où on rappelle que :

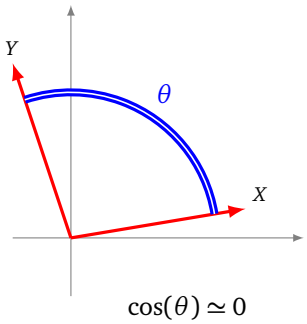
- $\langle X \mid Y \rangle = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$ est le produit scalaire des vecteurs X et Y,
- $\|X\| = \sqrt{x_1^2 + \cdots + x_n^2}$ et $\|Y\| = \sqrt{y_1^2 + \cdots + y_n^2}$ sont les normes de X et Y,
- θ est l'angle entre les vecteurs X et Y.
- Plus $\cos(\theta)$ est proche de 1, plus l'angle est proche de 0 et alors les vecteurs ont presque la même direction et le même sens,
- plus $\cos(\theta)$ est proche de -1, plus l'angle est proche de π et alors les vecteurs ont presque la même direction mais des sens contraires,
- si $\cos(\theta)$ est proche de 0 alors les vecteurs sont presque orthogonaux.



$\cos(\theta) \simeq 1$

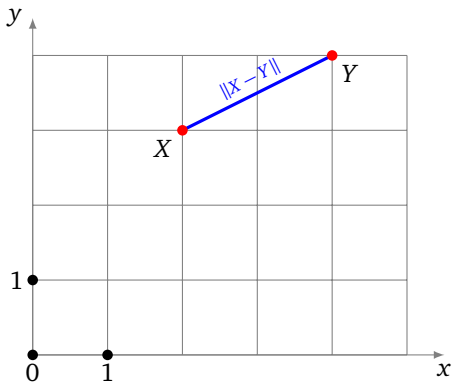


$\cos(\theta) \simeq -1$

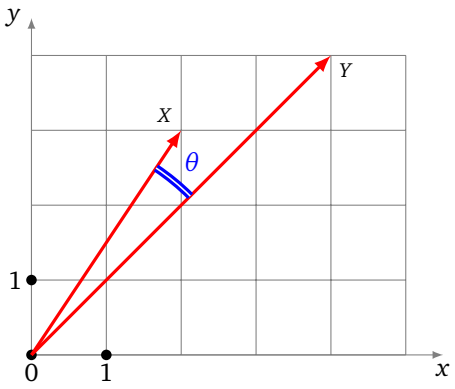


$\cos(\theta) \simeq 0$

Exemple.
Soient $X = \begin{pmatrix} 2 \\ 3 \end{pmatrix} \in \mathbb{R}^2$ et $Y = \begin{pmatrix} 4 \\ 4 \end{pmatrix} \in \mathbb{R}^2$.



Distance euclidienne
$\|X - Y\|$



Similarité cosinus
$\cos(\theta)$

- Distance euclidienne $\|X - Y\| = \sqrt{(2-4)^2 + (3-4)^2} = \sqrt{5} \simeq 2.24$.
- Similarité cosinus : $S_{\cos}(X, Y) = \frac{2 \cdot 4 + 3 \cdot 4}{\sqrt{13}\sqrt{32}} = \frac{5}{\sqrt{26}} \simeq 0.98$. Le cosinus est proche de 1 donc l'angle θ est proche de 0.

Exemple.
Dans le modèle *GPT2* les tokens **␣dog** et **␣cat** ont pour vecteurs plongés :

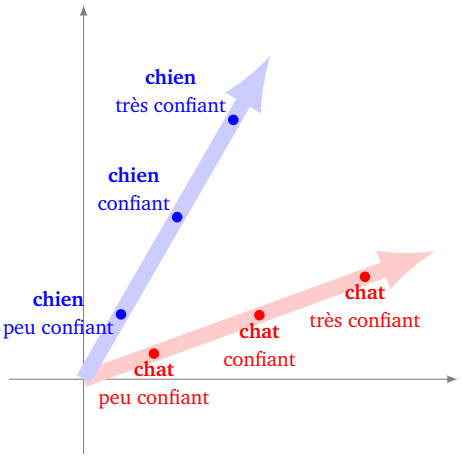
$$v_{\text{␣dog}} = \begin{pmatrix} 0.0944 \\ -0.0771 \\ 0.0346 \\ -0.0351 \\ -0.0892 \\ \vdots \end{pmatrix} \in \mathbb{R}^{768} \qquad v_{\text{␣cat}} = \begin{pmatrix} 0.0099 \\ 0.0365 \\ 0.1640 \\ -0.218 \\ 0.0285 \\ \vdots \end{pmatrix} \in \mathbb{R}^{768}$$

- La similarité cosinus $S_{\cos}(v_{\text{␣dog}}, v_{\text{␣cat}}) \simeq 0.55$ soit un angle $\theta \simeq 57°$. Ainsi le mots **␣dog** et **␣cat** sont bien distingués.
- Le token le plus proche de **␣dog** est **␣dogs** (au pluriel) avec une similarité de 0.79 (soit $\theta \simeq 38°$). Parmi les 50 257 tokens, les plus similaires à **␣dog** sont dans l'ordre :
  ␣dogs, **␣Dog**, **Dog**, **canine**, **dog**, **␣Dogs**, **␣puppy**, **␣pet**, **␣cat**, …
- Le token le moins similaire est **␣Stephenson** avec une similarité de 0.08 (soit $\theta \simeq 85°$).

On retient que si la similarité cosinus est proche de 1, les mots ont des significations proches. Selon les modèles, une similarité cosinus proche de -1 peut indiquer que les mots ont des significations opposées et si la similarité cosinus est proche de 0 les mots n'ont pas de lien entre eux.

Remarque importante : la similarité cosinus $S_{\cos}(X, Y)$ ne dépend que de la direction de X et de Y, pas de la longueur des vecteurs.

Exemple.
Dans notre situation linguistique, la longueur du vecteur correspond à la certitude que l'on a dans notre résultat.



Pourquoi utiliser la similarité cosinus ? Voici une explication sous la forme d'un exemple (fictif).

Exemple.
Chacun de nos axes de $\mathbb{R}^n$ correspond à une catégorie, par exemple un axe pour **homme**, un axe pour **femme**, un axe pour **enfant**, un axe pour **royauté**… (évidemment ce ne sera pas aussi simple que cela). Dans ces coordonnées (**homme**, **femme**, **enfant**, **royauté**) de $\mathbb{R}^4$ imaginons que le mot **roi** ait pour direction le vecteur $X_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$ (c'est autant un homme qu'un symbole de royauté), la mot **reine** a pour direction le vecteur $X_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$ et **princesse** $X_3 = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$.

	roi	**reine**	**princesse**
homme	1	0	0
femme	0	1	1
enfant	0	0	1
royaute	1	1	1

Si dans une phrase on trouve trois fois le mot **homme** et aussi les mots **couronne** et **château** (deux symboles de royauté), on va associer à ce texte le vecteur $Y = \begin{pmatrix} 3 \\ 0 \\ 0 \\ 2 \end{pmatrix}$ obtenu en comptant les occurrences

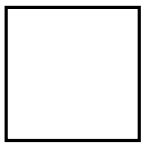
de chaque catégorie.
Calculons la similarité cosinus entre Y et les X_i :

$$S_{\cos}(X_1, Y) \simeq 0.98 \qquad S_{\cos}(X_2, Y) \simeq 0.39 \qquad S_{\cos}(X_3, Y) \simeq 0.32$$

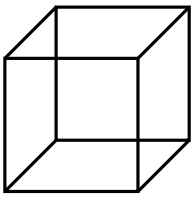
La similarité cosinus entre Y et X_1 (**roi**) est plus forte que celle entre Y et X_2 (**reine**) et entre Y et X_3 (**princesse**). Il est donc naturel d'associer à notre phrase le mot **roi**.
Noter qu'en terme de distance euclidienne X_1 (pour **roi**) et Y (pour notre phrase) sont éloignés.

4.2. Projections

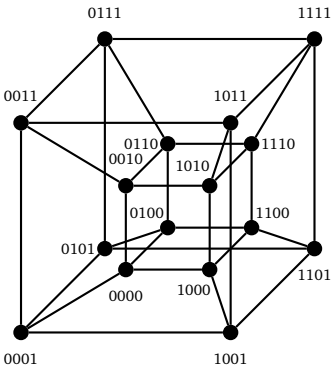
Projections. Les mots/tokens vont être plongés dans un espace $\mathbb{R}^n$ de grande dimension, par exemple avec $n = 768$ ou $n = 1024$. Notre perception est limitée aux dimensions 2 (le plan) et 3 (l'espace), on va donc se ramener à cette situation pour nos illustrations graphiques.
On se représente bien un carré (à gauche) et aussi un cube de l'espace projeté sur un plan (au centre), mais il est plus difficile de bien visualiser un hyper-cube de $\mathbb{R}^4$ (à droite avec les coordonnées de chaque sommet).



Carré



Cube



Hypercube

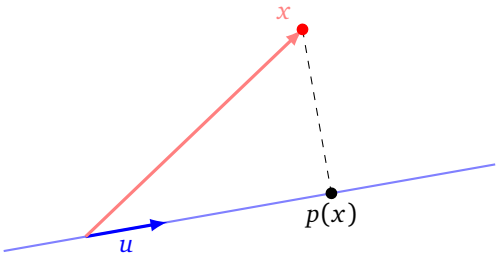
Nous allons projeter les éléments de $\mathbb{R}^n$ dans le plan ou l'espace. Nous notons π cette projection :

$$\pi : \mathbb{R}^n \to \mathbb{R}^2 \qquad \text{ou} \qquad \pi : \mathbb{R}^n \to \mathbb{R}^3$$

Projections sur une droite. Voici la formule pour projeter orthogonalement un vecteur $x \in \mathbb{R}^n$ sur une droite dirigée par un vecteur $u \in \mathbb{R}^n$ de norme 1 :

$$p(x) = \langle x \mid u \rangle u$$

On obtient une fonction à valeurs réelles en ne retenant que le produit scalaire : $\pi : \mathbb{R}^n \to \mathbb{R}$ définie par $\pi(x) = \langle x \mid u \rangle$.



On peut bien sûr définir le vecteur u à la main, mais on peut aussi choisir pour u un vecteur calculé par le plongement comme dans l'exemple suivant. Nous allons utiliser le modèle *BERT* qui a pour avantage d'avoir beaucoup de mots entiers qui sont des tokens, ce qui rend les exemples plus compréhensibles. *BERT* compte

30 522 tokens, parmi ceux-ci plus de 20 000 sont des mots complets. Le plongement de *BERT* associe à chaque token un vecteur de $\mathbb{R}^{768}$.

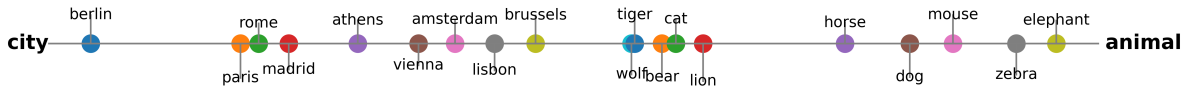
Exemple.
Considérons des noms d'animaux et des noms de villes :

cat, **dog**, **mouse**, **elephant**, **lion**, **tiger**, **bear**, **wolf**, **horse**, **zebra**
paris, **berlin**, **madrid**, **rome**, **lisbon**, **brussels**, **amsterdam**, **vienna**, **athens**

On souhaite visualiser si le plongement du modèle (ici *BERT*) distingue bien ces mots. On pourrait définir à la main un vecteur $u \in \mathbb{R}^{768}$ et projeter chaque vecteur v_i sur la droite dirigée par u, mais il n'est pas facile de faire un choix judicieux pour u.
Il est plus malin de calculer d'abord les plongements $v_{\textbf{animal}}$ et $v_{\textbf{city}}$ des tokens **animal** et **city**. Puis on définit pour u le vecteur :

$$u = v_{\textbf{animal}} - v_{\textbf{city}}$$

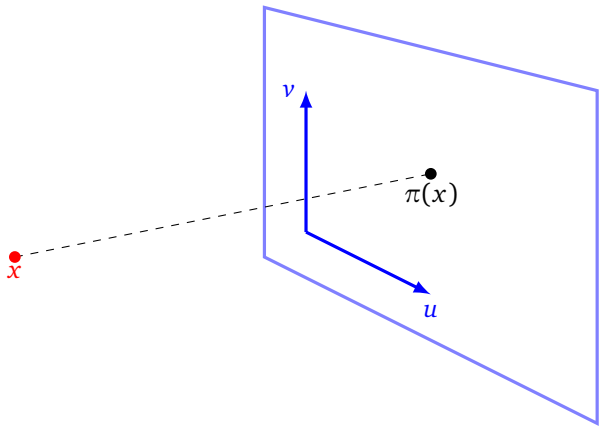
(que l'on peut rendre unitaire si besoin). Ensuite on considère la projection sur la droite dirigée par u : pour chaque vecteur token v_i on calcule le produit scalaire $\langle v_i \mid u \rangle$. Si notre modèle est correct alors plus ce produit scalaire est haut, plus le token correspond à la catégorie « animaux », plus le produit scalaire est bas, plus le token correspond à la catégorie « villes ».
Voici le tracé des nos mots obtenus par cette méthode :



On constate que le modèle sépare correctement les villes (à gauche) des animaux (à droite).

Projection sur un plan ou sur un espace. La formule précédente s'étend naturellement en dimension 2 et 3. Considérons les vecteurs $u, v \in \mathbb{R}^n$ formant une base orthonormée (c'est-à-dire $\|u\| = 1$, $\|v\| = 1$ et $\langle u \mid v \rangle = 1$), la projection orthogonale sur le plan défini par (u, v) est l'application $\pi : \mathbb{R}^n \to \mathbb{R}^2$ définie par

$$\pi(x) = \big(\langle x \mid u \rangle, \langle x \mid v \rangle\big).$$

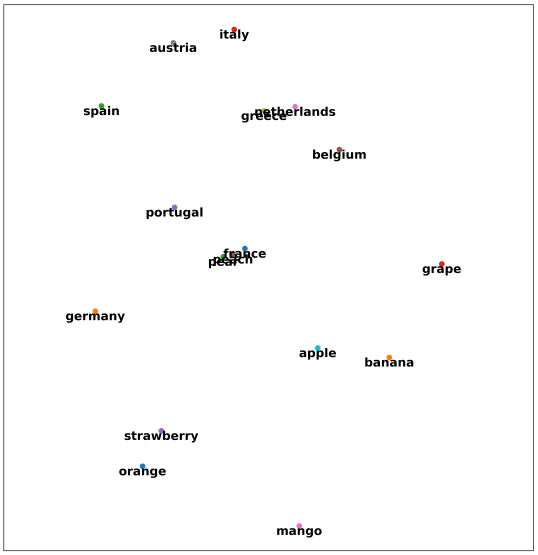


De même pour $u, v, w \in \mathbb{R}^n$ formant une base orthonormée, $\pi : \mathbb{R}^n \to \mathbb{R}^3$ est définie par

$$\pi(x) = \big(\langle x \mid u \rangle, \langle x \mid v \rangle, \langle x \mid w \rangle\big).$$

Exemple.
Voici la projection d'une liste de tokens de fruits et de pays sur le plan de $\mathbb{R}^{768}$ engendré par les vecteurs

$$u = f_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ \vdots \end{pmatrix} \text{ et } v = f_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \end{pmatrix}.$$



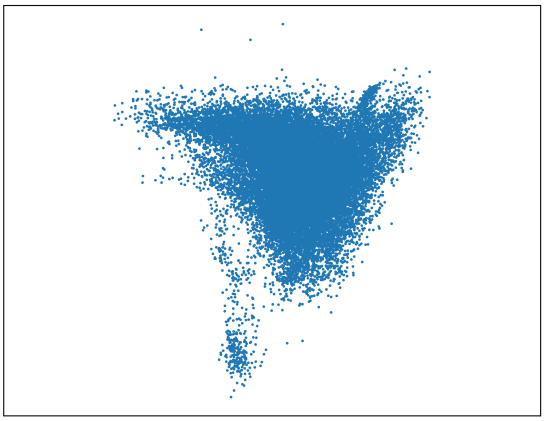
Noter que globalement la séparation entre les fruits et les pays est correcte, sauf pour les tokens **pear**, **peach**, **france** regroupés au centre. Cela ne signifie pas que le modèle sépare mal ces tokens, mais juste que notre projection ne montre pas correctement cette séparation.

Revenons sur le dernier point soulevé par cet exemple avec deux analogies. Lorsque vous observez le ciel étoilé, ce n'est pas parce que deux étoiles vous semblent proches dans le ciel, qu'elles le sont réellement. L'une peut être beaucoup plus éloignée de la Terre que l'autre. Une autre analogie est la suivante : lorsque vous regardez une radio d'un corps humain il est difficile de reconstituer la situation 3D, c'est d'ailleurs pour cela que l'on fait souvent des radios suivant plusieurs points de vue. Ici, il est illusoire de vouloir comprendre un espace de dimension 768 ou plus en n'étudiant seulement quelques projections. Cependant nous allons voir comment obtenir des projections plus pertinentes que d'autres.

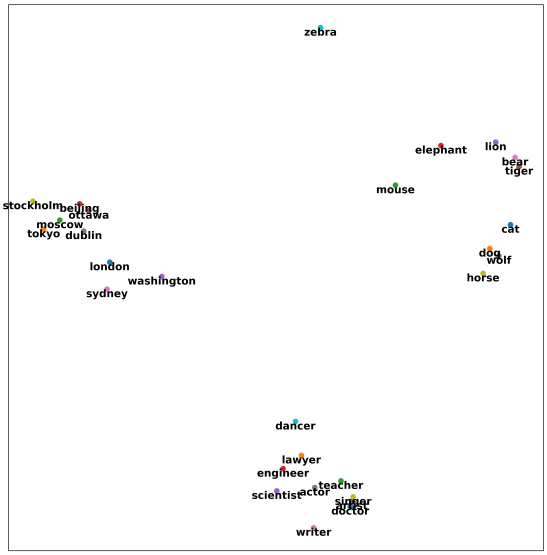
Analyse en composantes principales. Comment trouver une bonne projection qui mette le mieux en évidence l'étalement de nos données ? Il existe une technique statistique appelée analyse en composantes principales (*pca*, *principal component analysis*). Nous ne cherchons pas expliquer cette méthode un peu sophistiquée, mais qui s'écrit en quelques lignes en *Python*. Ici `X` est une liste `numpy` de vecteurs de $\mathbb{R}^n$, et cette fonction renvoie la projection de ces vecteurs dans le plan ($k = 2$) ou l'espace ($k = 3$).

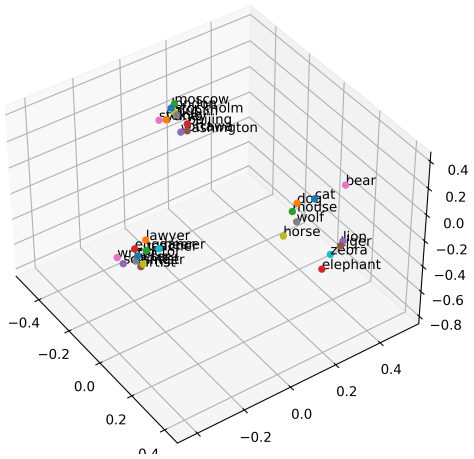
```
def pca(X, k=2):
    Xmean = X.mean(axis=0)                                  # Moyenne
    XX = X - Xmean                                  # Centrer les données
    C = np.dot(XX.T, XX) / (XX.shape[0] - 1)  # Matrice de covariance
    d, u = np.linalg.eigh(C)        # Décomposition en valeurs propres
    idx = np.argsort(d)[::-1]                # Tri des valeurs propres
    u = u[:, idx]                           # Tri des vecteurs propres
    U = u[:, :k]      # Projection sur les k premiers vecteurs propres
    return np.dot(XX, U)
```

Exemple.
En reprenant l'exemple précédent avec la projection obtenue par notre nouvelle méthode, on constate que cette fois les fruits sont clairement séparés des pays (ci-dessous à gauche). Sur la figure de droite on a projeté tous les 30 522 tokens de *BERT*.



Exemple.
On considère des mots/tokens de trois thèmes différents : « animaux », « métiers » et « villes ». Les mots sont clairement séparés selon leur catégorie, cela se voit sur la projection 2D et la projection 3D.



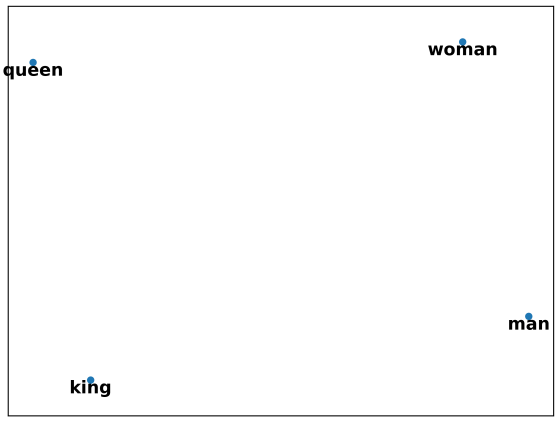


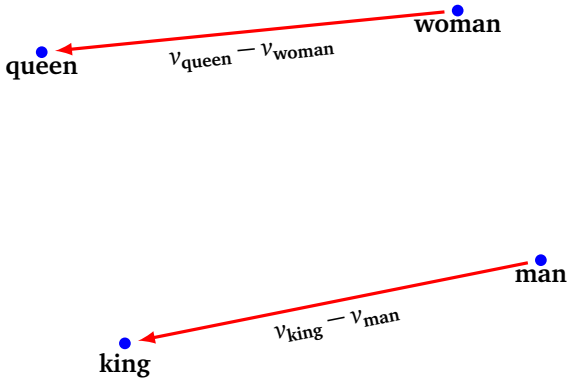
King - Man + Woman = Queen. Nous avons vu que les tokens/mots sont plongés dans l'espace en respectant une certaine logique. Voyons à quel point le plongement des tokens permet de comprendre la logique linguistique. Il existe un phénomène fascinant mais qui relève pour partie d'un mythe, c'est la formule :

$$\textbf{king} - \textbf{man} + \textbf{woman} = \textbf{queen}$$

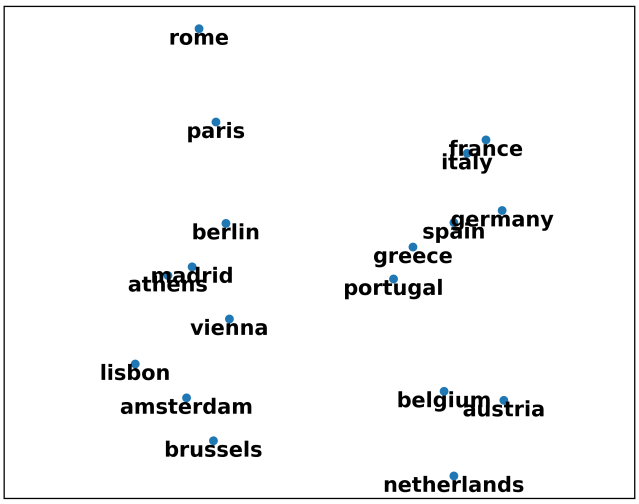
Il faut comprendre cette identité sur les vecteurs tokens obtenus par le plongement de chaque mot. Observons la figure de gauche ci-dessous qui est la projection (pca) des plongements des mots **man**, **king**, **woman**,

queen. On observe en effet que le vecteur qui va de **man** à **king** (c'est donc la projection du vecteur $v_{\text{king}} - v_{\text{man}}$) est presque égal au vecteur qui va de **woman** à **queen** (c'est donc la projection du vecteur $v_{\text{queen}} - v_{\text{woman}}$).





On pourrait donc croire que le plongement contient un certaine logique : si pour le terme **king** on remplace le caractère **man** par **woman**, on obtient **queen**. On retrouve le même phénomène avec les pays et leur capitale. Ci-dessous on passe d'une capitale à son pays par un vecteur à peu près horizontal.



En fait ce que l'on voit sur les projections ne reflète pas tout à fait la réalité de ce qui se passe en grande dimension. Si on calcule le vecteur

$$v = v_{\text{king}} - v_{\text{man}} + v_{\text{woman}}$$

et que l'on cherche quel vecteur token v_i parmi tous les tokens possibles est le plus proche de v (au sens de la similarité cosinus) les tokens les plus proches sont dans l'ordre :

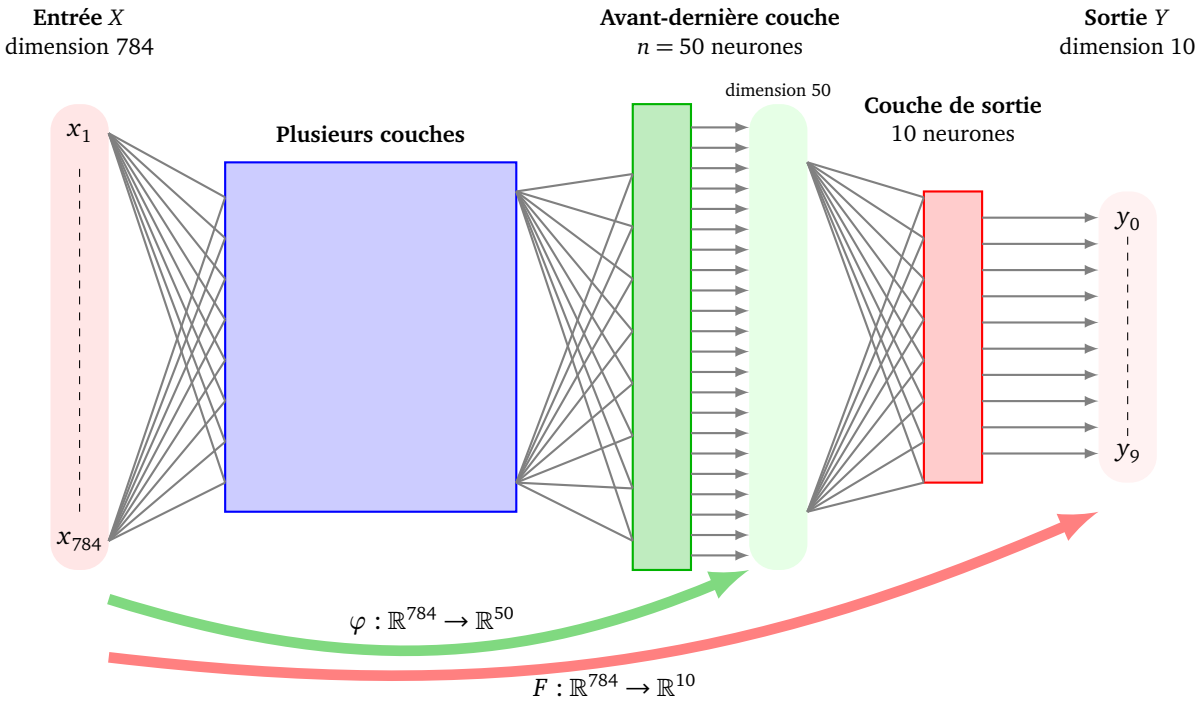
king, **queen**, **woman**, **princess**, **kings**, **queens**, **monarch**

Ainsi, le résultat souhaité, **queen**, n'arrive qu'en deuxième position. Dans ce genre de problème il faut généralement exclure des résultats les termes de départ (ici on exclurait **king**, **man**, **woman**).

5. Réalisation d'un plongement

5.1. Plongement des chiffres

Expliquons le principe de la création d'un plongement à l'aide de l'exemple des chiffres de la base MNIST. Rappelons que la base MNIST contient des images 28×28 en niveaux de gris, qu'il faut identifier par une valeur de 0 à 9. Il s'agit donc de trouver une fonction $F : \mathbb{R}^{784} \to \mathbb{R}^{10}$ qui à un vecteur image associe une liste de probabilités (voir la chapitre « Python : tensorflow avec keras - partie 2 »). Pour déterminer une telle fonction F nous construisons un réseau de neurones comme ci-dessous.



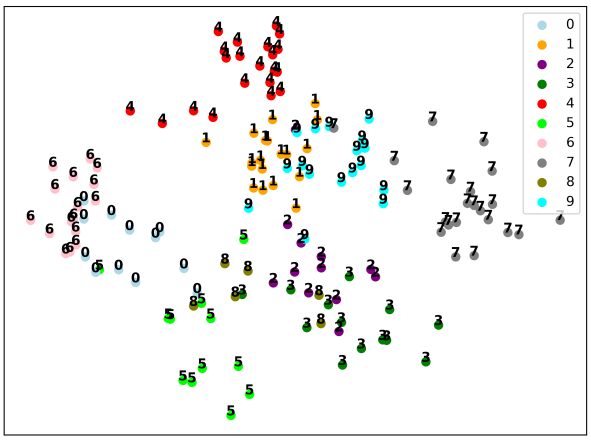
- En entrée, nous avons un vecteur de taille $N = 784$.
- Au milieu, nous avons un réseau de neurones non détaillé (avec ou sans couche de convolution), la partie non-détaillée se termine par une couche dense de n neurones, avec par exemple $n = 50$.
- La dernière couche (couche de sortie) est composée de 10 neurones, la fonction d'activation est softmax qui est adaptée pour renvoyer des probabilités correspondant au chiffre perçu. On pourrait considérer que $F : \mathbb{R}^N \to \mathbb{R}^{10}$ est notre plongement mais ce n'est pas le rôle de la couche de sortie qui est de décider d'une valeur entre 0 et 9.

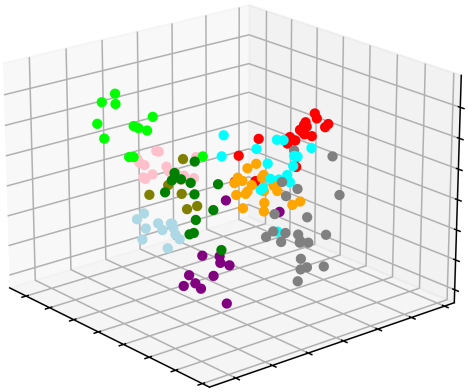
Une fois le réseau entraîné, il détecte correctement les chiffres (disons avec une précision $\geqslant 99\%$). Il est légitime de considérer que cette reconnaissance efficace n'est pas due qu'à la dernière couche et que l'essentiel du travail de catégorisation a déjà eu lieu lors des couches précédentes. Ainsi les valeurs sortantes de l'avant-dernière couche (qui servent d'entrée à la couche de sortie) ont déjà une certaine pertinence pour déterminer le chiffre. On considère donc la fonction

$$\varphi : \mathbb{R}^N \to \mathbb{R}^n$$

qui à une image associe les valeurs de sortie de l'avant-dernière couche. C'est cette fonction qui sera notre plongement. Noter que n peut être choisi à la valeur que l'on souhaite, en changeant le nombre de neurones de l'avant-dernière couche.

Voyons l'effet de ce plongement sur une centaine d'images de la base MNIST, après avoir appliqué une analyse en composantes principales pour obtenir une projection 2D et 3D.





On visualise bien les différentes catégories de chiffres, il y a cependant des superpositions dans nos projections. Par exemple, dans la projection 2D, les catégories des chiffres 2 et 3 s'entremêlent mais on constate que dans la projection 3D ces catégories sont assez bien séparées. On rappelle qu'une projection 2D ou 3D ne reflète que très partiellement la réalité d'un espace de grande dimension, ici $n = 50$.

5.2. Co-occurrence

Nous expliquons maintenant comment réaliser un plongement des tokens à partir d'un corpus. Nous allons réaliser un plongement simple mais tout de même efficace.
La première partie consiste à reprendre nos statistiques linguistiques des sections précédentes. À partir du corpus, on définit N tokens. Ensuite on définit la matrice des co-occurrences. C'est une matrice de taille $N \times N$ dans laquelle à la position (colonne i, ligne j) on insère le nombre de co-occurrences de (mot$i$, mot$j$) c'est-à-dire mot$i$ suivi de mot$j$.
Voici un exemple minimaliste avec les $N = 4$ tokens **le**, **et**, **chat**, **chien** (dans cet ordre). Imaginons que la matrice de co-occurrence soit :

	$i = 1$ **le**	$i = 2$ **et**	$i = 3$ **chat**	$i = 4$ **chien**
$j = 1$ **le**	0	10	1	0
$j = 2$ **et**	0	0	8	5
$j = 3$ **chat**	6	2	0	0
$j = 4$ **chien**	4	3	1	0
Total	*10*	*15*	*10*	*0*

Chaque occurrence de « **le chat** » dans le texte correspond à la paire (**le**, **chat**) et contribue à +1 à la place $(i = 1, j = 3)$ de la matrice de co-occurrence. Ici dans le texte il y a eu 6 occurrences de « **le chat** ».
Pour obtenir la matrice des probabilités, il faut pour chaque mot$i$ calculer la probabilité que le mot suivant soit mot$j$. Cette probabilité p_{ij} (à la colonne i, ligne j) se calcule par :

$$p_{ij} = \frac{\text{nombre d'occurrences de la paire (mot}i\text{, mot}j\text{)}}{\text{nombre d'occurrences de mot}i}.$$

Pour notre exemple minimaliste la matrice serait :

$$P = \begin{pmatrix} \frac{0}{10} & \frac{10}{15} & \frac{1}{10} & \frac{0}{5} \\ \frac{0}{10} & \frac{0}{15} & \frac{8}{10} & \frac{5}{5} \\ \frac{6}{10} & \frac{2}{15} & \frac{0}{10} & \frac{0}{5} \\ \frac{4}{10} & \frac{3}{15} & \frac{1}{10} & \frac{0}{5} \end{pmatrix}.$$

Exemple.
Reprenons le corpus *Reuter*. Parmi tous les mots (en minuscules, de longueur > 1), on travaille sur le

vocabulaire des $N = 500$ mots les plus fréquents qui seront nos tokens.
Les cinq premiers mots les plus fréquents sont :

the, **of**, **to**, **in**, **and**.

La matrice de co-occurrence est une matrice de taille $N \times N$, donc ici 500×500 :

$$M = \begin{pmatrix} 20 & 6834 & 2482 & 6757 & 1575 & \dots \\ 0 & 11 & 0 & 1 & 31 & \dots \\ 0 & 2 & 8 & 2 & 211 & \dots \\ 2 & 4 & 6 & 1 & 132 & \dots \\ 0 & 1 & 11 & 23 & 5 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}.$$

Par exemple, la deuxième colonne correspond au nombre d'occurrences du mot qui suit **of**. Il y a par exemple 6834 occurrences de la paire (**of**, **the**). La matrice P des probabilités (de même taille) s'en déduirait facilement.

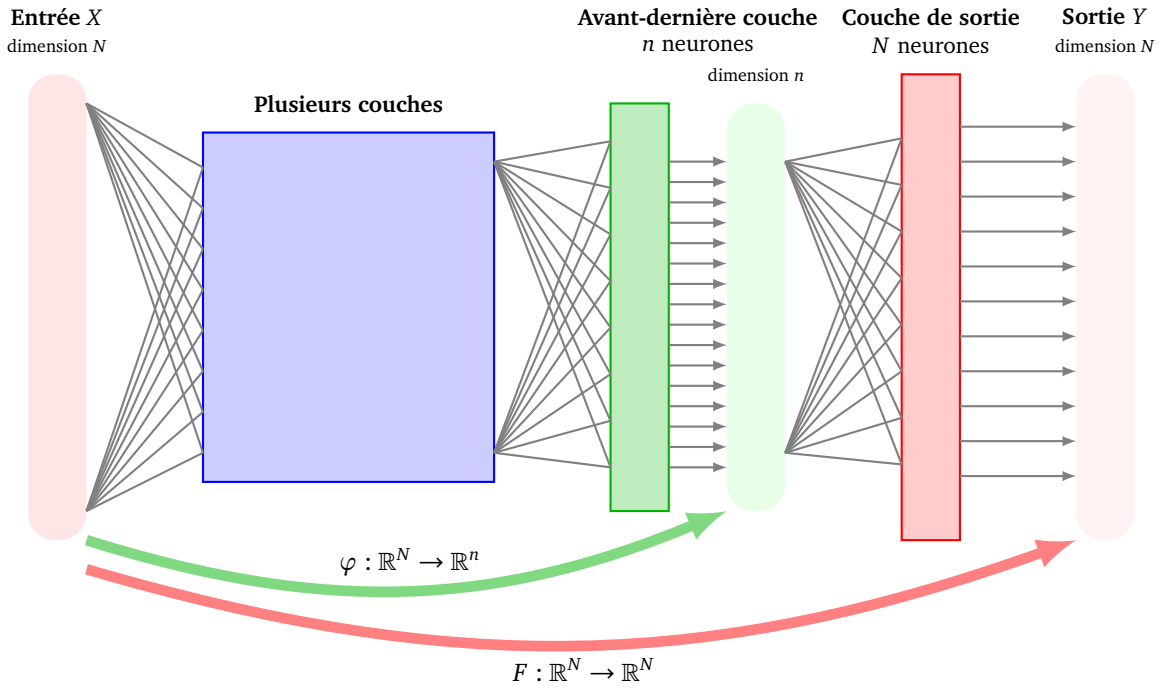
Ce que l'on retient, c'est que la colonne P_i de la matrice P contient la liste des probabilités pour chaque mot à la suite de mot$i$. Autrement dit, comme $P_i = Pe_i$, P est la matrice de l'application linéaire G définie par :

$$\begin{array}{rccc} G : & \mathbb{R}^N & \longmapsto & \mathbb{R}^N \\ & e_i & \longmapsto & P_i \end{array}$$

5.3. Plongement

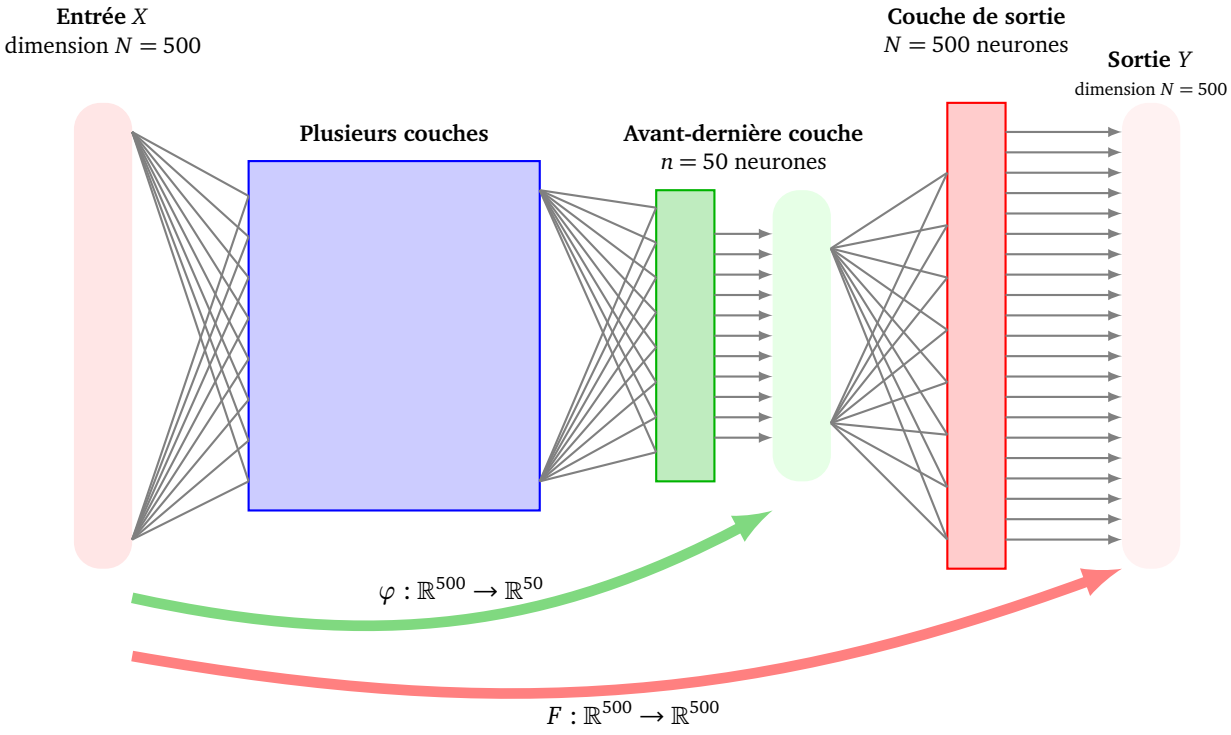
Nous allons maintenant réaliser un plongement $\varphi : \mathbb{R}^N \to \mathbb{R}^n$ à partir de la matrice P des probabilités. Construisons un réseau de neurones :

- En entrée nous avons un vecteur de taille N.
- Ensuite nous avons un réseau de neurones (plus ou moins complexe), ce réseau se termine par une couche dense de n neurones, avec par exemple $n = 768$ ou $n = 1024$.
- On ajoute enfin une couche de sortie composée de N neurones (autant que la dimension d'entrée) associés à la fonction d'activation softmax.

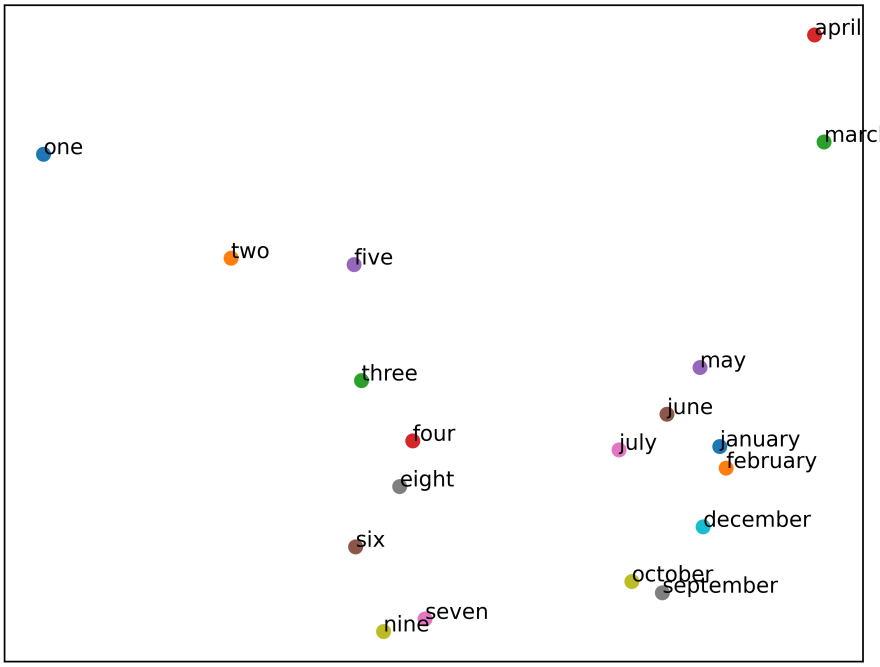


Ce réseau réalise donc une fonction $F : \mathbb{R}^N \to \mathbb{R}^N$. On entraîne ce réseau afin que F soit le plus proche possible de G, c'est-à-dire que pour chaque i on ait $F(e_i) \simeq P_i$. Ce réseau est donc adapté à prédire le mot suivant.
Une fois ce réseau entraîné, on considère la fonction $\varphi : \mathbb{R}^N \to \mathbb{R}^n$ qui à un vecteur e_i de la base canonique associe les valeurs en sortie de la couche dense de n neurones. C'est cette fonction qui sera notre plongement. (Une fois φ définie sur la base canonique, elle se prolonge par linéarité à tout $\mathbb{R}^N$.)
Comme pour le plongement des chiffres, le fait que le réseau soit entraîné à reconnaître le successeur d'un mot signifie que les valeurs en sortie de l'avant-dernière couche de n neurones sont déjà une certaine représentation des mots.

Exemple.
Finalisons le plongement issu des dépêches *Reuter*. On crée un réseau de neurones avec $N = 500$ entrées, puis une ou deux couches denses ou de convolutions, suivies ensuite d'une couche dense de $n = 50$ neurones (l'avant-dernière couche), puis une couche de sortie de $N = 500$ neurones avec la fonction d'activation softmax.



On entraîne ce réseau pour que $F(e_i) \simeq P_i$ pour chaque i. En récupérant les valeurs en sortie de la couche des n neurones, on définit notre plongement $\varphi : \mathbb{R}^N \to \mathbb{R}^n$.
On teste notre plongement sur des catégories de mots et on projette le résultat sur le plan. Voici un exemple avec les mois de l'année et des nombres.



Les mots sont clairement séparés en deux catégories « mois » et « nombres » sans que l'on ait eu à définir une catégorie « mois » ou « nombres » (et sans même avoir construit le réseau pour cela).
Les plongements de *BERT* ou *GPT* sont évidemment bien plus performants. Notre modèle est ici très simple, on le limite à un vocabulaire de $N = 500$ mots, notre corpus n'est pas gigantesque et on ne tient compte que des paires de mots consécutifs. Mais même ainsi, on obtient des résultats intéressants.