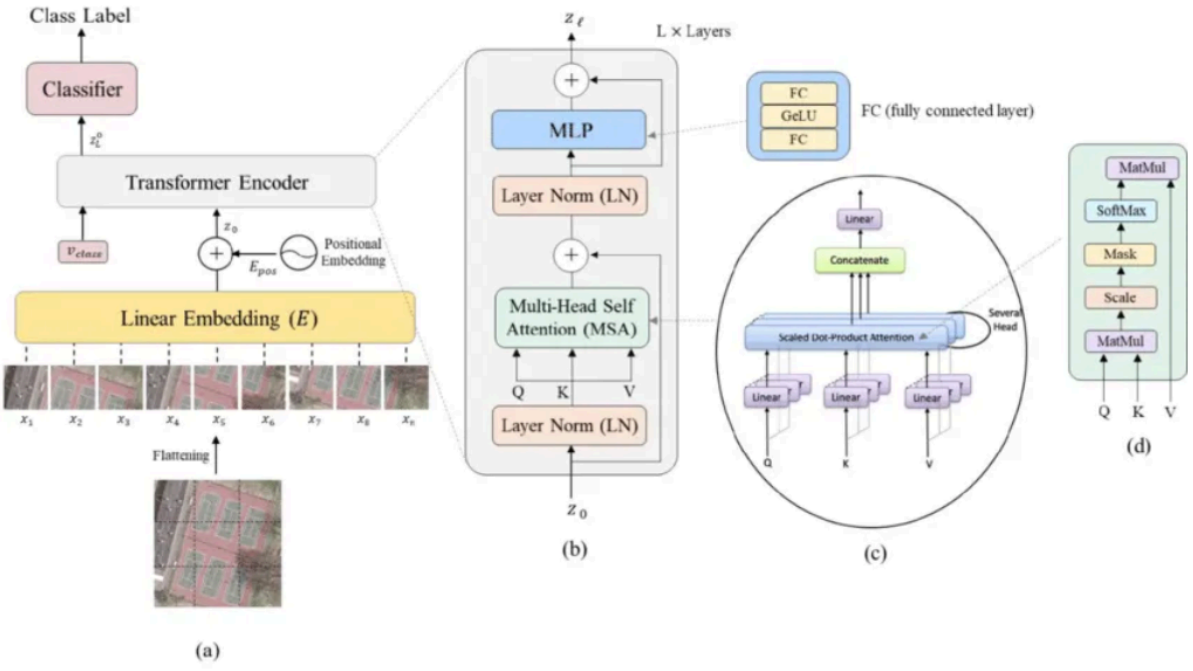
Transformers



Étape 1 : séparer en chunks "flattening" et transformer ces chunks (tokens) en embeddings

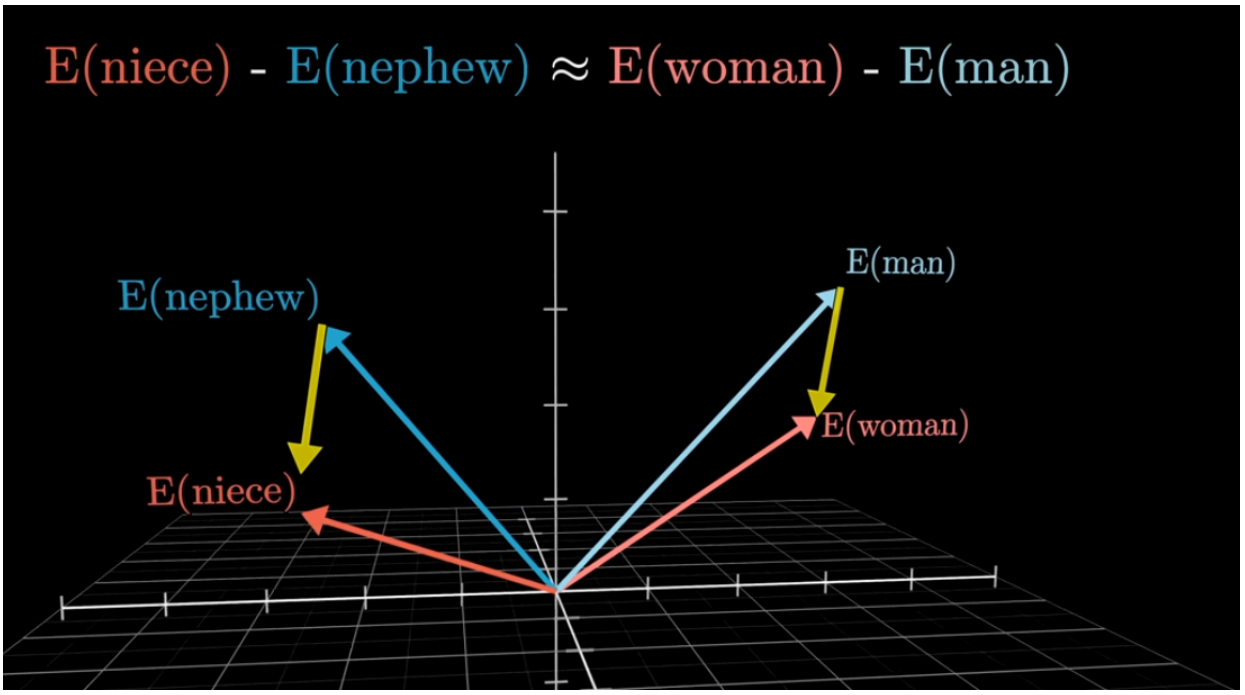
Prendre une image, une phrase, etc. ou autre et "l'aplatir", la transformer en petits chunks (généralement des syllables, mots, etc. ou petites parties d'images).

texte : chunks sont appelés tokens
image : chuncks sont appelés patches

Ensuite, on prend chaque chunk, et on obtient un vecteur à partir de la **embedding matrix** (qui contient déjà un mapping 1-1 entre chaque token et son embedding, c'est une look-up table).

Cette matrice contient donc $n_{\text{dimensions}} * n_{\text{mots}}$ paramètres.

Context size : le nombre de tokens, chunks, que peut prendre le transformer.



Une **direction** dans l'espace correspond à un genre, à la pluralité du mot, la taille par exemple, etc. On parle bien de **direction** et non pas de **dimension**, il n'y a pas une dimension liée à chaque critère !

aux vecteurs d'embeddings sont ajoutés des informations à propos de la position des éléments dans la phrase.

en fait on a des embeddings pour chaque position et on fait embedding mot + embedding position. De même que pour les critères de genre, pluralité, etc., on a une direction pour la position du mot dans la phrase.

Exemple de positional embedding

```
def get_positional_embeddings(sequence_length, d):
 result = torch.zeros(sequence_length, d)
 for i in range(sequence_length):
    for j in range(d // 2):
        angle = i / (10000 ** (2 * j / d))
        result[i, 2 * j] = torch.sin(torch.tensor(angle))
        result[i, 2 * j + 1] = torch.cos(torch.tensor(angle))
return result
```

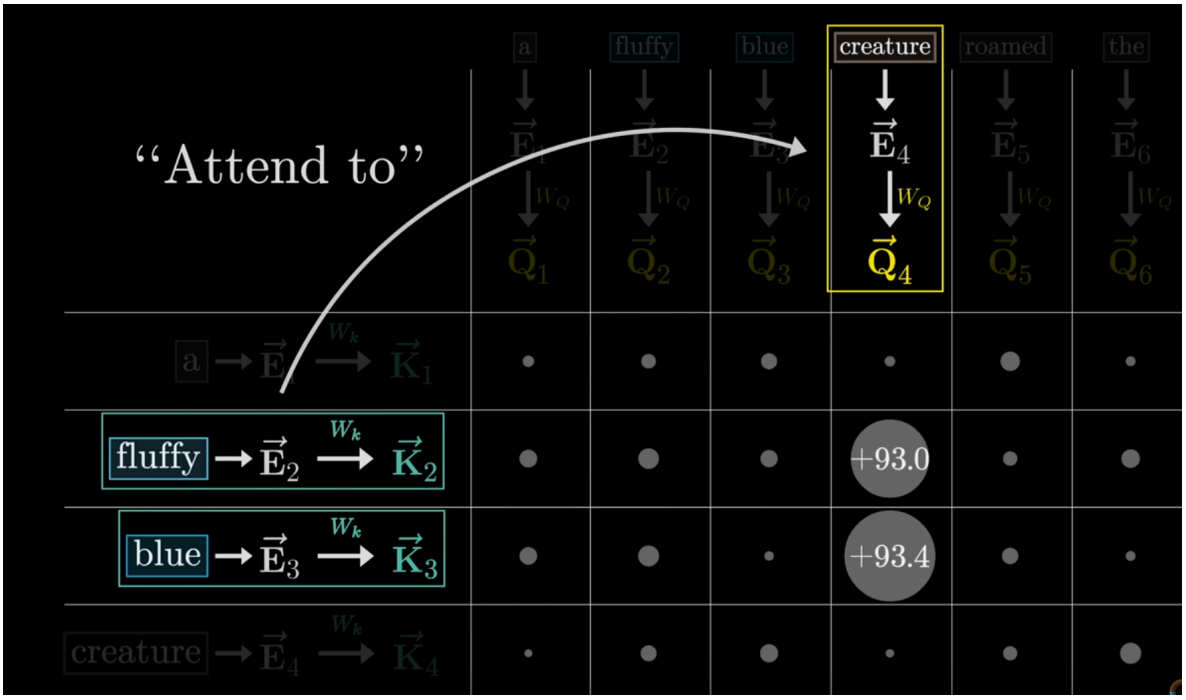
`sequence_length` c'est le nombre de tokens et `d` c'est le nombre de dimensions. Pour chaque embedding, on remplit les dimensions paires avec

Étape 2 : phase d'attention, partage de contexte entre les différents tokens

On veut une façon d'ajuster l'embedding des mots en fonction des autres mots à côté. Par exemple il y a sans doute une position dans l'espace qui correspond exactement à "tour eiffel", et "tour" est mal positionné au début.

Queries : les questions que le token se pose. pour la calculer, on a une matrice W_Q qu'on multiplie à l'embedding.
Keys : une sorte de FAQ proposée par le token. de même, on multiplie W_K avec l'embedding.
Values : les réponses à chaque question mise dans la FAQ (parce que les réponses ne sont pas forcément proches de la question !)

le nombre de dimensions de W_K et W_Q est bien plus petit que celui le nombre de dimensions des embeddings.



On va faire le dot product entre la query de chaque embedding et les keys des autres mots, et si on a un grand nombre alors ces mots sont sans doute intéressant.

On va donc mapper tous ces dot-products alors entre 0 et infinity entre 0 et 1 avec une fonction de softmax (qui s'applique sur chaque colonne parce que la somme de chaque **colonne** doit faire 1), puis les mutiplier par leur V pour obtenir une weighted sums des réponses les plus intéressantes.
Ensuite, on ajoute ce vecteur $\text{softmax}\left(K^T Q\right) V$ à notre embedding original.

masking

Quand on donne une séquence de tokens à notre GPT, et qu'on lui demande de prédire le suivant, en fait il va quand même essayer de prédire chaque token qu'on lui déjà donné (ce qui fait que quand on l'entraîne en lui disant "une créature verte mange une X", il va essayer de prédire "une créature verte X"). pour ça, on veut éviter que le modèle ait accès aux mots suivants "une créature verte" pour éviter qu'on lui spoile la réponse.

C'est pour ça qu'on "masque" les mots suivants, en modifiant, avant d'appliquer softmax sur chaque colonne, le résultat de leur dot product et en le mettant à -infinity (comme ça ça donnera 0).

cross attention

Ce qu'on a vu jusque-là c'était du self attention, le cross attention c'est quand on a deux types de données (par exemple une phrase source en anglais et une traduction en cours en français, ou alors un speech audio, et une transcription en cours). Les queries viennent donc de la transcription, traduction en français, et les keys/values viennent de l'audio ou de la phrase source.

multi-head attention

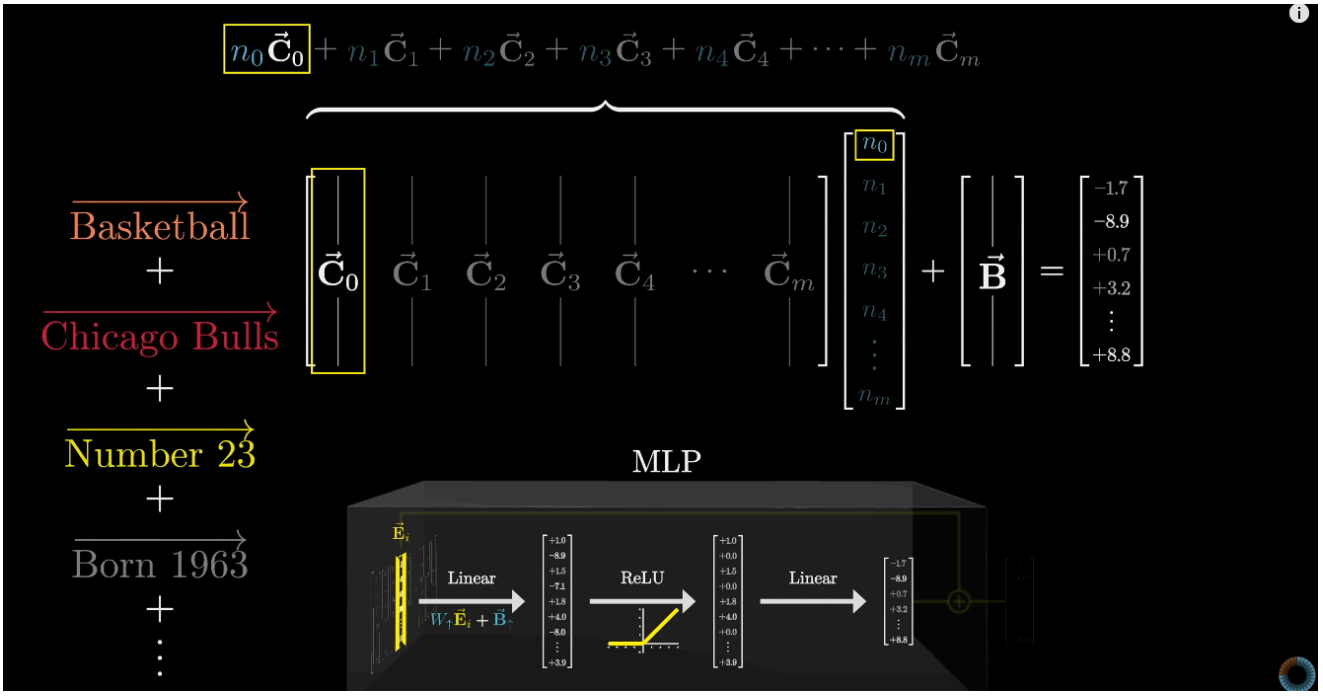
GPT 3 utilise 96 sets de queries, keys and values entraînés pour détecter et prêter attention de façon différentes aux mots autour. Chacun de ces heads proposent une

update de l'embedding de base ΔE, et pour calculer l'embedding final on additionne tous ces changements.

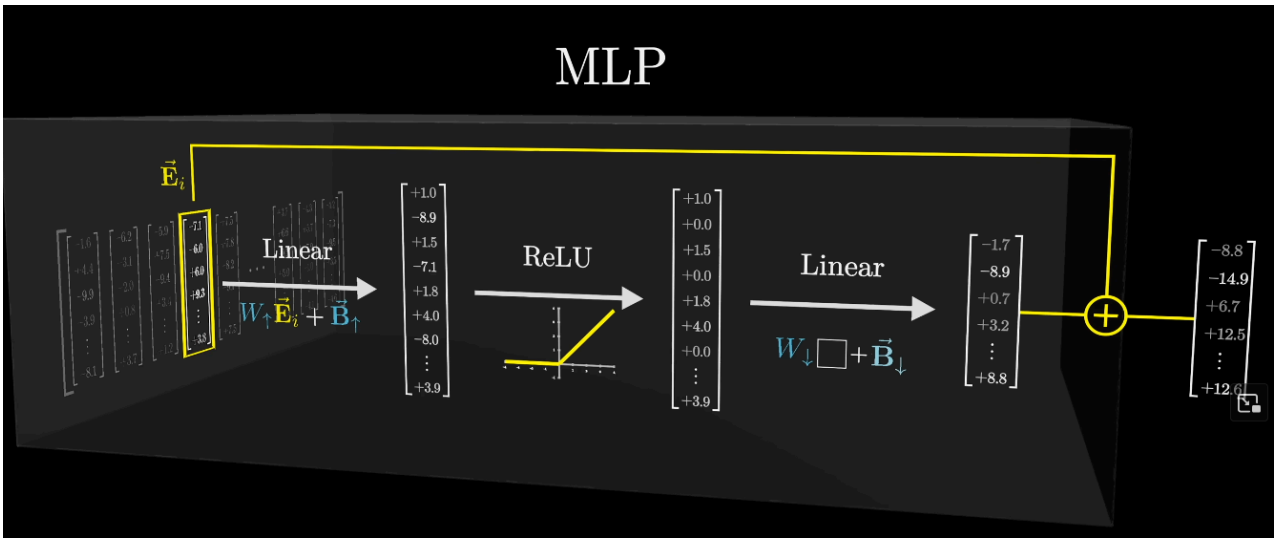
Étape 3 : multi layer perceptron

Dans cette étape, les vecteurs ne se parlent pas entre eux. On peut donc faire cette étape en parallèle pour chaque embedding.

- on multiplie notre embedding avec une matrice qui contient des paramètres (étape **linéaire**). on ajoute aussi un bias, typiquement utile pour que le résultat soit positif si la réponse est oui et négatif sinon. Chaque ligne pose une question sur l'embedding par exemple "est-ce que c'est du code ?", "est-ce que c'est de l'anglais ?", etc.
- le problème c'est que c'est linéaire pour le moment, nous on veut une réponse "oui" ou "non". on applique donc la fonction ReLU (Rectified Linear Unit).
- ensuite, on reconstruit l'embedding en ajoutant ou non des informations en fonction de si le neurone est actif



par exemple ici, si le neurone n_0 est 1 (c'est celui qui a demandé "est-ce que l'embedding parle de Michael Jordan"), alors on ajoute à notre embedding C_0 qui contient les informations à propos de basketball, des chicago bulls, du numéro de maillot de Michael Jordan, etc.



Étape y : on récupère une liste de probabilités pour chaque mot

On prend le dernier vecteur et on utilise un unembedding matrix + softmax pour générer les probabilités pour les mots.
La unembedding matrix est l'inverse de la embedding matrix ?