

L'attention, de la phrase au copieur

Requêtes, clés, valeurs et têtes d'induction

attention

softmax

masque causal

résidu

tête d'induction

Comment une position d'une suite peut-elle aller lire les précédentes, et pourquoi ce calcul a-t-il changé le traitement des langues ?

Un cadeau de fin de cours, à lire par curiosité : rien n'est à rendre. Le projet « Petit transformer » reste décrit par sa fiche sur le site et par le README de son squelette ; ce fascicule raconte les idées qui l'entourent et ce qu'elles deviennent au-delà du cours.

1 Pourquoi ce fascicule

Dans le projet « Petit transformer », vous avez écrit les trois calculs d'une tête d'attention causale : les scores entre requêtes et clés, leur transformation en poids par un masque et un softmax, puis le mélange des valeurs. Tout le reste était fourni : le corpus de copie $\hat{abc} | abc$, le tokenizer à sept caractères, le bigramme témoin, les embeddings, les normalisations, les résidus, le réseau dense, la perte, Adam et les figures. Vous avez comparé le bigramme au transformer, modifié un seul réglage, et sans doute constaté qu'une précision de copie honorable n'empêche pas une génération libre de dérailler.

Ce fascicule raconte d'où vient ce calcul et ce qu'il devient. Nous partirons de la traduction automatique de 2014, où l'attention est née pour résoudre un problème précis ; nous referons à la main, avec deux positions et deux dimensions, le calcul que vous avez écrit en Nx ; nous regarderons ce qui entoure l'attention dans un bloc, positions, résidus et normalisation, et pourquoi ces briques existent ; nous verrons pourquoi la copie est un bon jouet, et ce que des chercheurs ont trouvé en ouvrant des modèles qui copient ; enfin nous replacerons notre modèle de 3 520 paramètres face à GPT-2 et aux lois d'échelle. Aucune solution du projet n'y figure ; en revanche, le calcul à la main de la section 3 vaut la peine d'être refait avec un crayon.

2 Une phrase dans un seul vecteur

En 2014, deux équipes, celle d'Ilya Sutskever, Oriol Vinyals et Quoc Le, et celle de Kyunghyun Cho et Yoshua Bengio, montrent qu'un réseau peut traduire une phrase entière sans règle grammaticale écrite. Le dispositif comporte deux réseaux récurrents, c'est-à-dire des réseaux qui lisent une suite un élément à la fois en transportant un état : un *encodeur* lit la phrase source mot après mot et résume tout ce qu'il a lu dans un seul vecteur, puis un *décodeur* part de ce vecteur et produit la traduction mot après mot. Le résultat est remarquable sur les phrases courtes et se dégrade nettement sur les longues. Pourquoi ? Un vecteur de quelques centaines de nombres doit contenir toute la phrase, quelle que soit sa longueur : c'est un goulot d'étranglement.

En septembre 2014, Dzmitry Bahdanau, Kyunghyun Cho et Yoshua Bengio proposent de ne plus résumer. L'encodeur conserve un vecteur par mot source. À chaque mot qu'il produit, le décodeur calcule un score entre son état courant et

chacun de ces vecteurs, transforme les scores en poids positifs de somme un par un softmax, puis utilise la moyenne pondérée des vecteurs sources comme contexte. Les auteurs parlent de *recherche douce* (*soft search*) : le décodeur « va chercher » les mots sources utiles, sans qu'on lui dise lesquels, et les poids appris dessinent des alignements lisibles, par ex. l'inversion de l'adjectif et du nom entre l'anglais et le français. L'article paraît à ICLR 2015 ; le mot *attention* désigne depuis ce mécanisme. En 2015, Thang Luong, Hieu Pham et Christopher Manning montrent qu'un simple produit scalaire suffit comme score.

En juin 2017, Ashish Vaswani et sept coauteurs publient « Attention is all you need ». Le titre est un programme : ils suppriment les réseaux récurrents et ne gardent que l'attention, appliquée d'une phrase à elle-même. Chaque position calcule trois vecteurs, une requête, une clé et une valeur, lit toutes les autres positions par produit scalaire, et plusieurs *têtes* font ce calcul en parallèle ; des encodages de position compensent la perte de l'ordre, et des connexions résiduelles avec normalisation permettent d'empiler les blocs. Le modèle, nommé *transformer*, s'entraîne sur huit GPU en trois jours et demi pour la plus grande version, et atteint 28,4 BLEU en anglais-allemand et 41,8 en anglais-français sur les jeux de WMT 2014, au-dessus de tous les systèmes précédents. Le décodeur de cet article, qui ne lit que les positions déjà produites, est l'ancêtre direct de notre bloc : les modèles de la famille GPT n'en conservent que cette moitié.

Du point de vue d'AP5, rien n'a changé dans la méthode : une formule à paramètres, une perte, un gradient et des pas d'optimisation. Ce qui change, c'est la formule, et c'est elle que nous allons dérouler.

3 Requêtes, clés, valeurs : un calcul à la main

Fixons les notations du projet. Une suite de T positions arrive sous la forme de T vecteurs de dimension D , un par token. Trois matrices apprises transforment chacun de ces vecteurs en une *requête* q_i , une *clé* k_i et une *valeur* v_i . Le *score* entre la position i qui lit et la position j lue est le produit scalaire $s_{ij} = q_i \cdot k_j / \sqrt{D}$. Les scores d'une ligne passent par un softmax, qui rend des *poids* w_{ij} positifs et de somme un, et la sortie de la position i est la moyenne pondérée des valeurs, $o_i = \sum_j w_{ij} v_j$. Ce vocabulaire décrit un calcul ; il ne prête aucune intention au réseau, même si les images « ce qu'une position cherche », « ce qu'elle offre » et « ce qu'elle transmet » aident à retenir qui fait quoi.

Prenons deux positions et $D = 2$, avec des vecteurs choisis pour que le calcul tienne sur un coin de table. La position 1 porte la lettre `a`, la position 2 le séparateur `|` :

Position	Token	Requête q	Clé k	Valeur v
1	a	(0, 1)	(1, 0)	(1, 0)
2		(2, 0)	(0, 1)	(0, 1)

Calculons les quatre produits scalaires, puis divisons par $\sqrt{2} \approx 1,414$. La ligne de `a` donne $q_1 \cdot k_1 = 0$ et $q_1 \cdot k_2 = 1$, soit des scores 0 et 0,707. La ligne de `|` donne $q_2 \cdot k_1 = 2$ et $q_2 \cdot k_2 = 0$, soit 1,414 et 0. Le softmax d'une ligne (s_1, s_2) vaut $(e^{s_1}, e^{s_2}) / (e^{s_1} + e^{s_2})$. Pour la ligne de `|`, $e^{1,414} \approx 4,113$ et $e^0 = 1$, d'où les poids (0,804; 0,196) : la position du séparateur lit la lettre `a` à quatre-vingts pour cent. Sa sortie vaut $0,804 \times (1, 0) + 0,196 \times (0, 1) = (0,804; 0,196)$, un vecteur qui ressemble surtout à la valeur de `a`. C'est ce que nous voulons d'un copieur : après le séparateur, la première lettre est désormais « à portée de main ».

Et la ligne de `a` ? Sans précaution, son softmax vaut (0,330; 0,670) : la position 1 lirait à soixante-sept pour cent la position 2, qui vient après elle. Pendant l'apprentissage, la ligne entière `^abc|abc` est présente, et la cible de la position 1 est précisément le token de la position 2. Un modèle autorisé à lire la suite apprendrait à recopier la réponse, puis échouerait au moment de générer, quand la suite n'existe pas encore. D'où le *masque causal* : avant le softmax, les scores des positions $j > i$ sont remplacés par $-\infty$, dont l'exponentielle vaut zéro. La ligne de `a` devient (1; 0) et sa sortie est sa propre valeur, (1, 0). Voici le calcul complet :

Ligne	Scores / $\sqrt{2}$	Sans masque	Avec masque	Sortie o_i
a lit	(0; 0,707)	(0,330; 0,670)	(1; 0)	(1, 0), sa propre valeur
lit	(1,414; 0)	(0,804; 0,196)	(0,804; 0,196)	(0,804; 0,196), surtout celle de <code>a</code>

Pourquoi diviser par $\sqrt{D}$? Parce qu'un produit scalaire de D termes grandit avec D : si les composantes sont d'ordre un, le produit est d'ordre $\sqrt{D}$. Or le softmax est très sensible aux grands écarts. Comparons nos scores à des scores dix fois plus grands, dans une petite fonction Elixir qui soustrait d'abord le maximum de la ligne, comme le fait votre version Nx pour éviter les exponentielles trop grandes :

```

1 softmax = fn scores ->
2   plus_grand = Enum.max(scores)
3   exps = Enum.map(scores, fn s -> :math.exp(s - plus_grand) end)
4   total = Enum.sum(exps)
5   Enum.map(exps, fn e -> Float.round(e / total, 4) end)
6 end
7
8 IO.inspect(softmax.([1.414, 0.0]), label: "scores divisés par  $\sqrt{2}$  ")
9 IO.inspect(softmax.([14.14, 0.0]), label: "les mêmes, dix fois plus
grands")

```

```

CONSOLE  scores divisés par  $\sqrt{2}$  : [0.8044, 0.1956]
les mêmes, dix fois plus grands: [1.0, 0.0]

```

Avec des scores dix fois plus grands, la position ne lit plus qu'une seule autre, et la dérivée du softmax y est presque nulle : le gradient ne traverse plus, et l'apprentissage s'arrête de corriger cette tête. Vaswani et ses collègues justifient la division exactement ainsi. Le même phénomène, à l'échelle d'un neurone, est celui de `tanh` saturée dans Micrograd.

CE QUE GARANTIT UNE LIGNE DE POIDS

Après le masque et le softmax, la ligne i est une distribution de probabilité sur les positions 0 à i : des nombres positifs de somme un, nuls au-delà de i . La sortie o_i est donc une *combinaison convexe* des valeurs du préfixe, et ne dépend d'aucune position future. En effet, les exponentielles sont strictement positives et la division par leur somme normalise la ligne ; un score $-\infty$ donne $e^{-\infty} = 0$ exactement, pas un petit nombre, et ce zéro multiplie la valeur v_j correspondante. C'est pourquoi modifier un token futur ne change pas les logits du préfixe, ce que les tests du projet vérifiaient.

3.1 Le masque causal en grille

La figure ci-dessous montre la grille des poids pour le début d'une ligne, `^abc|`, lorsque tous les scores autorisés sont égaux : chaque position lit alors uniformément son préfixe. Les lignes ont des longueurs différentes, ce qui explique les poids 1, $1/2$, $1/3$ et ainsi de suite ; les cases grises sont celles où le score vaut $-\infty$.

		position lue j				
		$\wedge$	a	b	c	
position qui lit i	$\wedge$	1	0	0	0	0
	a	1/2	1/2	0	0	0
	b	1/3	1/3	1/3	0	0
	c	1/4	1/4	1/4	1/4	0
		1/5	1/5	1/5	1/5	1/5

$j \leq i$: position lue
 $j > i$: masquée
 chaque ligne somme à 1
 scores autorisés tous égaux

Lignes : la position qui lit ; colonnes : la position lue. Le triangle inférieur, diagonale comprise, porte les poids ; les cases au-dessus sont masquées. Avec des scores appris, les poids d'une ligne deviennent inégaux, mais le triangle et les sommes à un demeurent.

PIÈGE

Le masque s'applique aux scores, avant le softmax. Mettre à zéro les scores interdits ne masque rien : $e^0 = 1$, et la position future recevrait un poids ordinaire. Mettre à zéro les poids après le softmax casse la somme à un. Seul $-\infty$ avant l'exponentielle donne un zéro exact et une ligne encore normalisée. En Nx, ce masque de forme $\{\tau, \tau\}$ doit en outre être diffusé sur l'axe du lot avant un `select` : c'est le « cannot broadcast » du README.

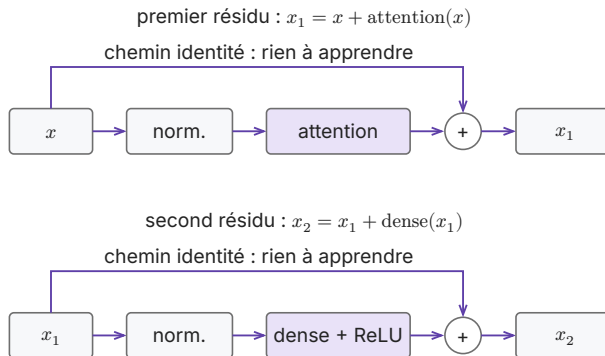
4 Autour de l'attention : position, résidus, normalisation

Une tête d'attention seule ne fait pas un transformer. Trois briques l'entourent dans votre bloc, toutes fournies, et chacune répond à un souci précis.

La première est l'*encodage de position*. Relisez la formule de la sortie : $o_i = \sum_j w_{ij} v_j$, où chaque w_{ij} ne dépend que de q_i et de k_j . Si les vecteurs d'entrée ne portaient que l'identité du token, permuter les positions 0 à $i - 1$ permuterait les termes de cette somme sans changer sa valeur : la position i verrait son préfixe comme un sac de tokens, sans ordre. Les entrées $\wedge abc|$ et $\wedge cba|$ donneraient la même prédiction après le séparateur, et aucune copie dans l'ordre ne serait possible. Vaswani et ses collègues ajoutent donc au vecteur de chaque token un vecteur qui dépend de sa position, construit à partir de sinusoides de fréquences différentes ; GPT-2, comme

notre modèle, préfère apprendre un vecteur par position, ce qui revient au même tant que les suites ne dépassent pas la longueur vue à l'entraînement.

La deuxième brique est la *connexion résiduelle*. En décembre 2015, Kaiming He, Xiangyu Zhang, Shaoqing Ren et Jian Sun constatent qu'un réseau de vision plus profond apprend parfois moins bien qu'un réseau plus court, alors qu'il pourrait en principe l'imiter en laissant ses couches supplémentaires ne rien faire. Le remède est de reformuler chaque couche : au lieu d'apprendre une transformation $F(x)$, elle apprend une correction, et sa sortie est $x + F(x)$. Ne rien faire revient alors à produire $F(x) = 0$, ce qui est facile. Leurs réseaux de 152 couches remportent la compétition ImageNet de 2015. L'argument qui nous intéresse est celui du gradient : la dérivée locale d'une addition vaut un pour chacun de ses arguments, comme la clause fournie dans Micrograd, de sorte que l'adjoint de la sortie remonte tel quel jusqu'à x par le chemin identité, quoi que fasse la branche F . Empiler quarante-huit blocs ne multiplie donc pas quarante-huit dérivées locales petites : un chemin direct relie toujours la perte à chaque bloc. Dans notre modèle, deux résidus entourent l'attention et le réseau dense ; on parle de *flux résiduel* pour ce vecteur par position que chaque brique lit, puis augmente d'une correction.



Un bloc, lu de gauche à droite. Chaque brique reçoit une version normalisée du flux et sa sortie est ajoutée au flux, pas substituée. Le chemin du haut ne contient aucun paramètre : il transmet les valeurs à l'aller et les adjoints au retour sans les modifier.

La troisième brique est la *normalisation de couche*, due à Jimmy Ba, Jamie Kiros et Geoffrey Hinton en juillet 2016. Pour chaque position, prise seule, on recentre les D composantes du vecteur à moyenne nulle et on les divise par leur écart-type, puis deux vecteurs appris, une échelle et un décalage, rendent au réseau la liberté perdue. Contrairement à la normalisation par lot, qui calcule des statistiques sur les exemples d'un lot, elle ne mélange ni les positions ni les exemples et fait le même calcul à l'entraînement et à l'usage. Son effet est de rendre l'échelle du flux prévisible,

quels que soient les poids : voici deux vecteurs qui diffèrent d'un facteur dix et qui, normalisés, deviennent identiques.

```
1 normaliser = fn vecteur ->
2   n = length(vecteur)
3   moyenne = Enum.sum(vecteur) / n
4   variance = Enum.sum(for v <- vecteur, do: (v - moyenne) * (v -
5     moyenne)) / n
6   Enum.map(vecteur, fn v -> Float.round((v -
7     moyenne) / :math.sqrt(variance + 1.0e-5), 3) end)
8 IO.inspect(normaliser.([1.0, 2.0, 3.0, 10.0]))
9 IO.inspect(normaliser.([10.0, 20.0, 30.0, 100.0]))
```

```
CONSOLE [-0.849, -0.566, -0.283, 1.697]
[-0.849, -0.566, -0.283, 1.697]
```

Le petit 10^{-5} sous la racine évite une division par zéro sur un vecteur constant. Où placer cette normalisation ? L'article de 2017 la mettait après chaque addition résiduelle ; GPT-2, puis notre modèle, la placent avant chaque brique, à l'entrée de l'attention et du réseau dense, en laissant le chemin identité intact. Ruibin Xiong et ses collègues ont montré en 2020 que cette variante, dite *pré-normalisation*, donne des gradients mieux dimensionnés au départ de l'entraînement et se passe de la montée progressive du taux d'apprentissage qu'exigeait l'autre.

À RETENIR

Les trois briques sont des réponses à trois pannes connues : sans position, l'attention ne voit pas l'ordre ; sans résidu, la profondeur dégrade le gradient ; sans normalisation, l'échelle du flux dérive et le softmax sature. Aucune n'ajoute de pouvoir d'expression spectaculaire ; toutes rendent l'apprentissage possible.

5 Copier est un bon jouet

Pourquoi avoir choisi une tâche aussi artificielle que recopier trois lettres ? Parce qu'elle a tout ce qu'il faut pour une expérience que l'on peut expliquer : un corpus minuscule de 64 lignes, une règle exactement connue, une mesure ciblée, la précision

sur les trois lettres copiées, et surtout un témoin dont on peut démontrer l'échec. Le bigramme prédit le prochain caractère à partir du seul caractère courant. Après le séparateur |, il doit annoncer la première lettre du triplet, et il n'a aucun moyen de la connaître : sur le corpus complet, les quatre lettres suivent | aussi souvent les unes que les autres. Calculons ce qu'un bigramme parfait, qui connaîtrait exactement ces fréquences, pourrait obtenir au mieux :

```

1 lettres = ["a", "b", "c", "d"]
2
3 lignes =
4   for x <- lettres, y <- lettres, z <- lettres do
5     triplet = x <> y <> z
6     "^" <> triplet <> "|" <> triplet <> "\n"
7   end
8
9 paires =
10  lignes
11  |> Enum.flat_map(fn ligne ->
12    caracteres = String.graphemes(ligne)
13    Enum.zip(caracteres, tl(caracteres))
14  end)
15  |> Enum.frequencies()
16
17 entropie = fn courant ->
18   suivants = for {{^courant, suivant}, n} <- paires, do: {suivant,
19     n}
20   total = suivants |> Enum.map(&elem(&1, 1)) |> Enum.sum()
21   -Enum.sum(for {_, n} <- suivants, do: n / total * :math.log(n /
22     total))
23 end
24
25 IO.puts("après | : #{Float.round(entropie("|"), 3)} nats, soit ln
26 4")
27 IO.puts("après a : #{Float.round(entropie("a"), 3)} nats, soit ln
28 6")
29 IO.puts("uniforme sur 7 : #{Float.round(:math.log(7), 3)} nats")

```

```

CONSOLE  après | : 1.386 nats, soit ln 4
         après a : 1.792 nats, soit ln 6
         uniforme sur 7 : 1.946 nats

```

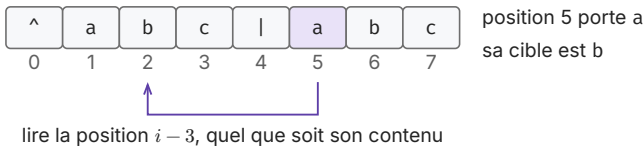
L'entropie conditionnelle calculée ici est la perte moyenne la plus basse qu'un modèle puisse atteindre quand il ne connaît que le caractère courant : c'est l'entropie croisée

du chapitre 8, prise au mieux. Après `|`, le bigramme ne peut descendre sous $\ln 4 \approx 1,386$ nats, et il ne devine la bonne lettre qu'une fois sur quatre ; après une lettre, six suites sont équiprobables, et il ne descend pas sous $\ln 6 \approx 1,792$ nats. Sur les trois positions de copie, sa précision plafonne donc vers $(1/4 + 1/6 + 1/6)/3$, soit dix-neuf pour cent. Aucun entraînement, aussi long soit-il, ne changera ces nombres : la limite est dans la famille de fonctions, comme la limite affine d'AP5 sur les classes alternées. Le transformer, lui, peut atteindre la lettre manquante, puisqu'elle est dans le préfixe, trois positions plus tôt.

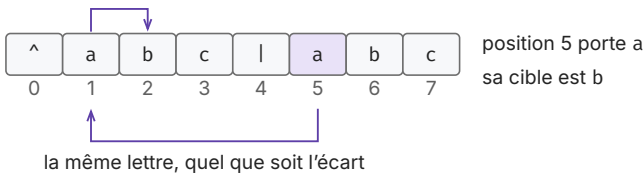
5.1 Deux façons de copier, et les têtes d'induction

Comment une tête peut-elle « retrouver » la lettre à copier ? La figure ci-dessous montre deux stratégies. La première repose sur la position : dans notre corpus, la lettre à prédire se trouve toujours trois cases avant la position courante, et une requête qui s'accorde avec la clé de la position $i - 3$, grâce aux vecteurs de position, suffit. La seconde repose sur le contenu : à la position qui porte `a`, chercher l'occurrence précédente de `a` et lire ce qui la suivait. Elle fonctionne quel que soit l'écart, et donc sur des triplets de longueur variable, mais elle demande deux étapes.

par la position : lire trois cases en arrière



par le contenu : retrouver a, puis lire ce qui le suivait



En haut, une tête lit la position $i - 3$ quel qu'en soit le contenu : il lui suffit de comparer des vecteurs de position. En bas, le mécanisme d'induction : retrouver la lettre courante plus tôt dans la suite, puis se déplacer d'une case pour lire ce qui la suivait. La première lettre copiée, après `|`, n'a pas d'occurrence précédente et passe par une autre voie.

La seconde stratégie a un nom. En décembre 2021, Nelson Elhage et ses collègues d'Anthropic décrivent, dans de petits transformers sans réseau dense, un circuit de deux têtes situées dans deux couches successives : une *tête de token précédent* écrit

dans chaque position l'identité du token qui la précède, puis une *tête d'induction* cherche, depuis la position courante qui porte A , une position dont le token précédent était A , et recopie le token qui s'y trouve, B . Le circuit complète les motifs $AB...A \rightarrow B$. En mars 2022, Catherine Olsson et ses collègues montrent que ces têtes apparaissent d'un coup, tôt dans l'entraînement, dans tous les modèles d'au moins deux couches, au moment précis où le modèle devient capable d'utiliser son contexte, et ils avancent, avec les précautions d'usage, qu'elles portent l'essentiel de cet *apprentissage en contexte* dans les grands modèles. Le point qui nous concerne est leur remarque sur la profondeur : un modèle à une seule couche d'attention ne peut pas former ce circuit, car la seconde tête a besoin de ce que la première a écrit. Notre bloc unique est donc condamné à la première stratégie, celle du décalage fixe. Elle suffit sur ce corpus ; elle ne suffirait plus si la longueur des triplets variait.

EN PRATIQUE

Une carte d'attention montre où une tête lit ; elle ne dit pas ce qu'elle fait de ce qu'elle lit, ni si la prédiction en dépend. Elhage et Olsson ne s'arrêtent pas aux cartes : ils retirent une tête et mesurent ce que la perte perd, ou suivent ce qu'une tête écrit jusqu'aux logits. Le compte rendu du projet demandait la même prudence.

6 En contexte : GPT-2, les lois d'échelle et les tokens

Notre modèle a un bloc, une tête, des vecteurs de dimension 16 et 3 520 paramètres. GPT-2, publié par Alec Radford et ses collègues en février 2019, empile 48 blocs de la même forme que le vôtre, pré-normalisation comprise, avec des vecteurs de dimension 1 600, plusieurs têtes par bloc, un contexte de 1 024 tokens et 1 542 millions de paramètres. Il est entraîné à prédire le token suivant sur WebText, quarante gigaoctets de texte tirés de huit millions de pages web. Entre les deux, la perte, le masque causal, les résidus et les trois calculs de l'attention sont identiques ; ce qui change est la taille, le nombre de têtes, qui découpent la dimension en sous-espaces lus en parallèle, et le vocabulaire.

Ce vocabulaire compte 50 257 tokens, obtenus par *encodage par paires d'octets* (*byte pair encoding*, BPE). L'algorithme vient de la compression de données : Philip Gage le décrit en 1994 pour remplacer, dans un fichier, la paire d'octets la plus fréquente

par un octet inutilisé, et recommencer. En 2016, Rico Sennrich, Barry Haddow et Alexandra Birch l'adaptent à la traduction : on part des caractères, on fusionne la paire adjacente la plus fréquente en un nouveau symbole, et l'on répète jusqu'à la taille de vocabulaire voulue. Les mots fréquents deviennent un token, les mots rares se découpent en morceaux connus, et aucun mot n'est jamais inconnu. Voici trois fusions sur l'exemple classique :

```

1 fusionner = fn mot, nouveau ->
2   paires = mot |> Enum.zip(tl(mot)) |> Enum.frequencies()
3   {{gauche, droite}, _n} = Enum.max_by(paires, fn {_, n} -> n end)
4   Enum.reduce(mot, [], fn
5     jeton, [^gauche | reste] when jeton == droite -> [nouveau |
6       reste]
7     jeton, acc -> [jeton | acc]
8   end)
9   |> Enum.reverse()
10
11 mot = String.graphemes("aaabdaaabc")
12 etape1 = fusionner.(mot, "Z")
13 etape2 = fusionner.(etape1, "Y")
14 etape3 = fusionner.(etape2, "X")
15 IO.puts(Enum.join([mot, etape1, etape2, etape3] |>
16   Enum.map(&Enum.join/1, " → ")))

```

CONSOLE aaabdaaabc → ZabdZabc → YbdYbac → XdXac

La paire aa devient z, puis za devient y, puis yb devient x : onze symboles sont devenus cinq, et la table des trois fusions suffit à reconstituer le mot. Notre projet n'avait pas besoin de cela : sept caractères formaient tout le vocabulaire. GPT-2 applique l'algorithme aux octets eux-mêmes, de sorte que n'importe quelle suite d'octets, dans n'importe quelle langue, se découpe.

Reste la question de la taille. En janvier 2020, Jared Kaplan, Sam McCandlish et leurs collègues mesurent la perte de modèles de cette famille en faisant varier le nombre de paramètres, la quantité de texte et le calcul dépensé, et observent que la perte suit des *lois de puissance* régulières sur plus de sept ordres de grandeur : doubler les ressources améliore la perte d'une fraction prévisible, sans palier visible. Ces courbes, affinées en 2022 par Jordan Hoffmann et ses collègues, qui corrigent l'équilibre entre paramètres et données, ont guidé le choix des tailles des modèles suivants. Notre entraînement de deux cents pas sur 48 lignes est très loin, tout à

gauche de ces courbes ; mais il repose sur le même bloc, et c'est précisément ce que ces lois disent : la recette ne change pas, seule l'échelle change.

7 Au-delà du projet : ce qui change à l'échelle, et ce qui ne change pas

Le coût de l'attention croît avec le carré de la longueur du contexte : chaque position lit toutes les précédentes, soit T^2 scores par tête et par bloc. Avec $T = 8$, cela fait 64 scores ; avec les 1 024 tokens de GPT-2, environ un million ; avec les contextes de plus de cent mille tokens des modèles actuels, plus de dix milliards par tête et par bloc. La matrice $T \times T$ des poids pose en outre un problème de mémoire. En 2022, Tri Dao et ses collègues montrent avec FlashAttention qu'on peut calculer exactement le même softmax par blocs, sans jamais écrire la matrice complète en mémoire, en gardant au fil des tuiles le maximum courant et la somme courante, la même astuce que votre softmax stable. À la génération, les modèles conservent les clés et les valeurs des tokens déjà produits dans un *cache*, pour ne pas les recalculer à chaque nouveau token ; ce cache grandit avec le contexte et devient à son tour la ressource rare. D'autres approches, attention creuse ou linéaire, renoncent à l'exactitude pour un coût moindre.

Ce qui ne change pas mérite d'être dit aussi nettement. Les trois fonctions que vous avez écrites sont, à la forme des tenseurs près, celles qu'exécute chaque tête de chaque bloc des grands modèles. Le masque causal est le même. La perte est la même entropie croisée sur le token suivant, et l'entraînement montre toujours les vrais tokens précédents, tandis que la génération réutilise ses propres sorties : l'écart que vous avez observé entre la précision de copie et les triplets générés existe à toutes les échelles. Enfin, une carte d'attention reste une observation du calcul, pas une explication, et un modèle qui copie ou qui complète n'a appris que les régularités de son corpus. Savoir ce qu'il a appris, et pourquoi, demande des expériences comme celles de la section précédente, à toutes les tailles.

8 Pour aller plus loin

Neural machine translation by jointly learning to align and translate, Dzmitry Bahdanau, Kyunghyun Cho et Yoshua Bengio (2014).

<https://arxiv.org/abs/1409.0473>

L'article qui invente l'attention pour résoudre un problème précis, avec les cartes d'alignement de la section 4.

Attention is all you need, Ashish Vaswani et al. (2017).

<https://arxiv.org/abs/1706.03762>

Dix pages denses ; la section 3 décrit scores, échelle, masque, têtes multiples et positions, exactement dans l'ordre de votre projet.

The Illustrated Transformer, Jay Alammar (2018).

<https://jalammar.github.io/illustrated-transformer/>

Les formes des tenseurs dessinées une à une ; la meilleure lecture pour relier le papier de 2017 à votre `model.ex`.

Deep residual learning for image recognition, Kaiming He, Xiangyu Zhang, Shaoqing Ren et Jian Sun (2015).

<https://arxiv.org/abs/1512.03385>

D'où viennent les deux additions de votre bloc, et pourquoi elles permettent la profondeur.

Layer normalization, Jimmy Lei Ba, Jamie Ryan Kiros et Geoffrey Hinton (2016).

<https://arxiv.org/abs/1607.06450>

La normalisation fournie dans le squelette, et sa comparaison avec la normalisation par lot.

In-context learning and induction heads, Catherine Olsson et al. (2022).

<https://arxiv.org/abs/2209.11895>

Ce que l'on trouve en ouvrant des modèles qui copient ; à lire après « A mathematical framework for transformer circuits » des mêmes auteurs, de 2021.

Language models are unsupervised multitask learners, Alec Radford et al. (2019).

https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

GPT-2 : la section 2 décrit le corpus, le BPE sur octets et l'architecture, qui est la vôtre en plus grand.

Scaling laws for neural language models, Jared Kaplan et al. (2020).

<https://arxiv.org/abs/2001.08361>

Les courbes de perte en fonction de la taille, des données et du calcul ; la figure 1 résume tout.

9 Une question pour la route

Dans notre corpus, l'écart entre une lettre et sa copie vaut toujours trois positions. Supposez un corpus où le triplet devient un mot de deux à six lettres, par ex. `^ab|ab` ou `^abcdef|abcdef`. Un seul bloc à une tête, avec des vecteurs de position,

peut-il encore copier ? Quelle information devrait circuler d'une position à l'autre pour qu'une tête retrouve « la lettre qui suivait la dernière occurrence de la lettre courante », quel que soit l'écart, et combien de couches faut-il au minimum pour la faire circuler ? Enfin, que se passe-t-il pour la toute première lettre copiée, juste après le séparateur ?

Pour toute question, vous pouvez me contacter par email.