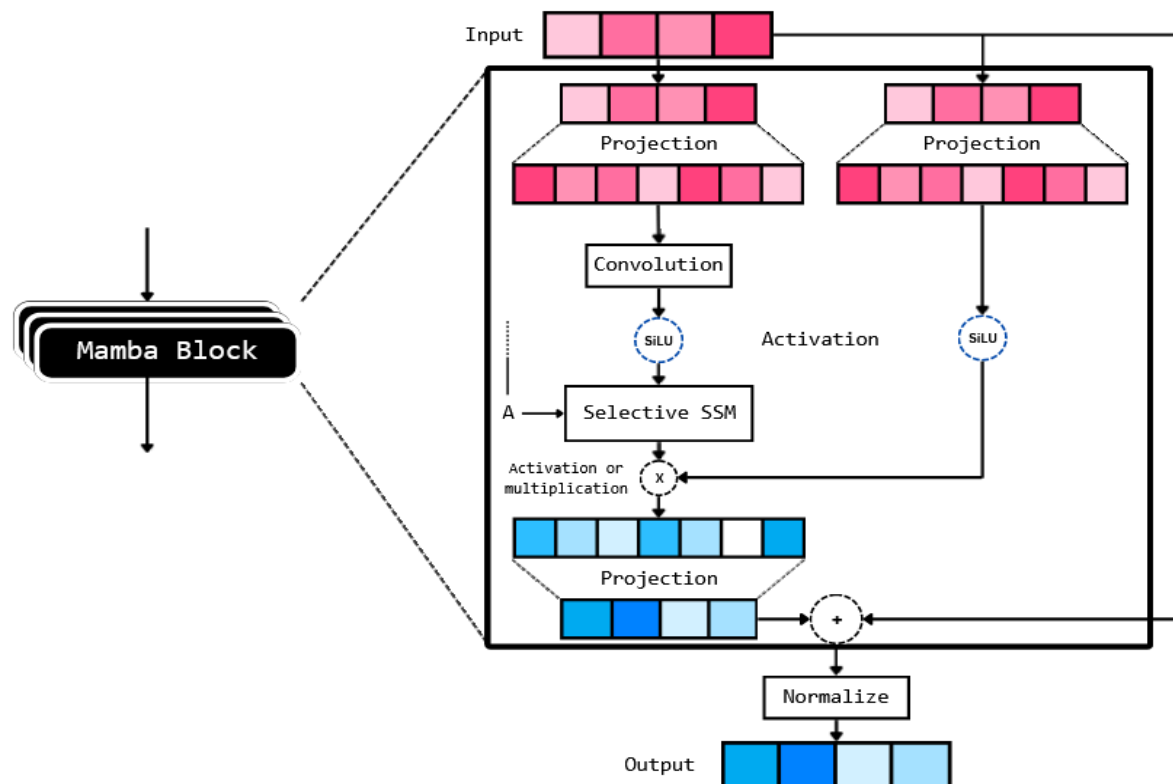


Mamba à partir de zéro (édition RUST)

Audric HARRIS

Chapitre 1 : Décomposer le problème

Il est important de prendre le temps de bien comprendre le fonctionnement de Mamba ; si nous essayons de coder quelque chose que nous ne maîtrisons pas nous-mêmes, nous n'irons pas loin. Je commencerai par présenter le modèle Mamba, comme illustré ci-dessous, et je détaillerai étape par étape le fonctionnement de chaque bloc Mamba.



Le diagramme montre un bloc résiduel singulier également appelé bloc mamba.

Les grands modèles de langage comportent généralement plusieurs blocs résiduels ; dans notre cas, nous en aurons dix, également appelés couches. Dans un premier temps, nous nous concentrerons sur une seule couche afin de simplifier les explications.

Les entrées :

Quel type d'entrée un bloc Mamba utilise-t-il ? On pourrait penser qu'il s'agit d'un mot ou d'une phrase, mais c'est faux. Un bloc Mamba utilise un tenseur. Chaque mot et chaque phrase possède sa propre représentation tensorielle, et le modèle Mamba est capable de prédire un autre tenseur à partir d'un tenseur existant. Dans notre cas, nous ne parlerons que de tenseurs, car les mots n'interviennent qu'avant et après le calcul.

Définition de tenseur : objet mathématique analogue mais plus général qu'un vecteur, représenté par un tableau de composantes qui sont des fonctions des coordonnées d'un espace.

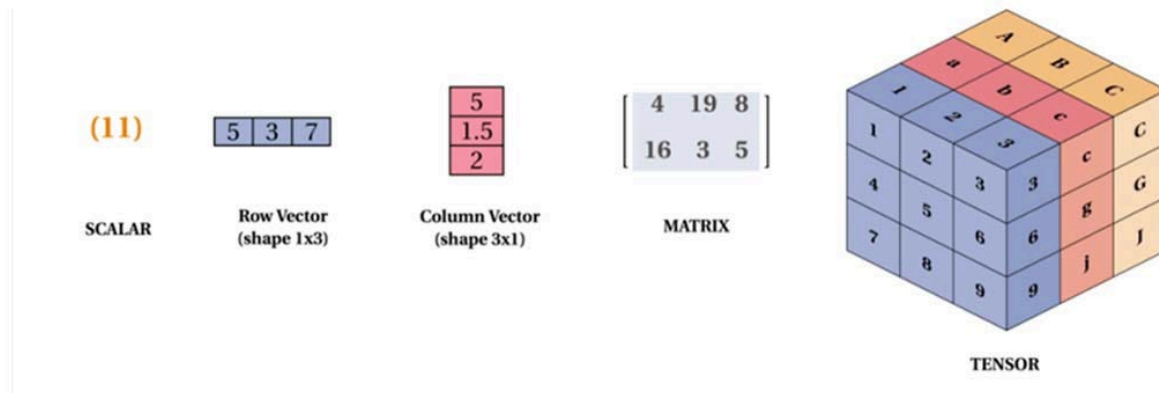


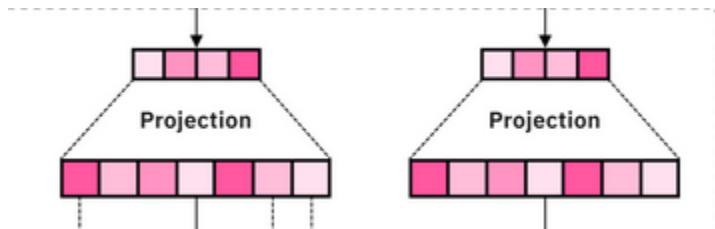
Diagramme d'un tenseur et d'objets mathématiques similaires pour aider à la compréhension

Maintenant que nous avons alimenté le tenseur dans notre bloc Mamba, que fait-il ? Il en conserve d'abord une copie en mémoire : plus le tenseur est grand, plus il peut stocker d'informations. L'inconvénient est que l'entraînement de Mamba prendra plus de temps pour traiter ces tenseurs plus volumineux et nécessitera davantage de paramètres de modèle. (Nous aborderons les paramètres distincts des hyperparamètres plus loin dans la documentation.)

Notre modèle enregistre une instance du tenseur d'origine comme attribut ; en général, nous ne stockons pas les variables d'entrée dans une structure, mais nous nous appuyons sur celles transmises à la fonction forward. Cependant, la structure stocke ici deux instances d'une projection de ce tenseur d'origine.

Les projections :

Dans le contexte d'un bloc Mamba, une projection linéaire est une opération fondamentale qui transforme le tenseur d'entrée (appelons-le $x \times x$) en une nouvelle représentation en appliquant une transformation linéaire apprenable : $y = Wx + b$ $y = Wx + b$ $y = Wx + b$, où W W W est une matrice de pondération (dont les dimensions définissent la taille de sortie) et b b b un vecteur de biais optionnel. Il s'agit essentiellement d'une multiplication matricielle suivie d'une addition, implémentée efficacement dans des frameworks comme Burn via des couches entièrement connectées (denses). L'objectif est d'étendre ou de remodeler le tenseur dans un espace de plus grande dimension adapté aux composants du modèle sélectif d'espace d'état (SSM), par exemple en projetant $x \times x$ pour créer des chemins distincts pour des paramètres tels que les matrices A , B et C du SSM et le Delta variant dans le temps, permettant au modèle de traiter sélectivement les informations de séquence. Dans notre implémentation, cette projection augmente généralement la dimension cachée du tenseur (par exemple, de d_{model} d_{model} d_{model} à $2 \times d_{\text{state}}$ $2 \times d_{\text{state}}$ $2 \times d_{\text{state}}$) afin de fournir des fonctionnalités plus riches pour le calcul récurrent, bien que les projections ultérieures (comme celles de sortie) puissent être ramenées à la taille d'origine pour plus d'efficacité. Lors de l'entraînement, les pondérations W W W et les biais b b b sont optimisés par rétropropagation, ce qui permet au modèle d'apprendre des transformations spécifiques à la tâche sans modifier la longueur de la séquence principale.



Maintenant que j'ai expliqué le fonctionnement de la projection linéaire, comment la réaliser en Rust ? Dans la documentation de burn, nous trouvons `burn::nn.Linear`.



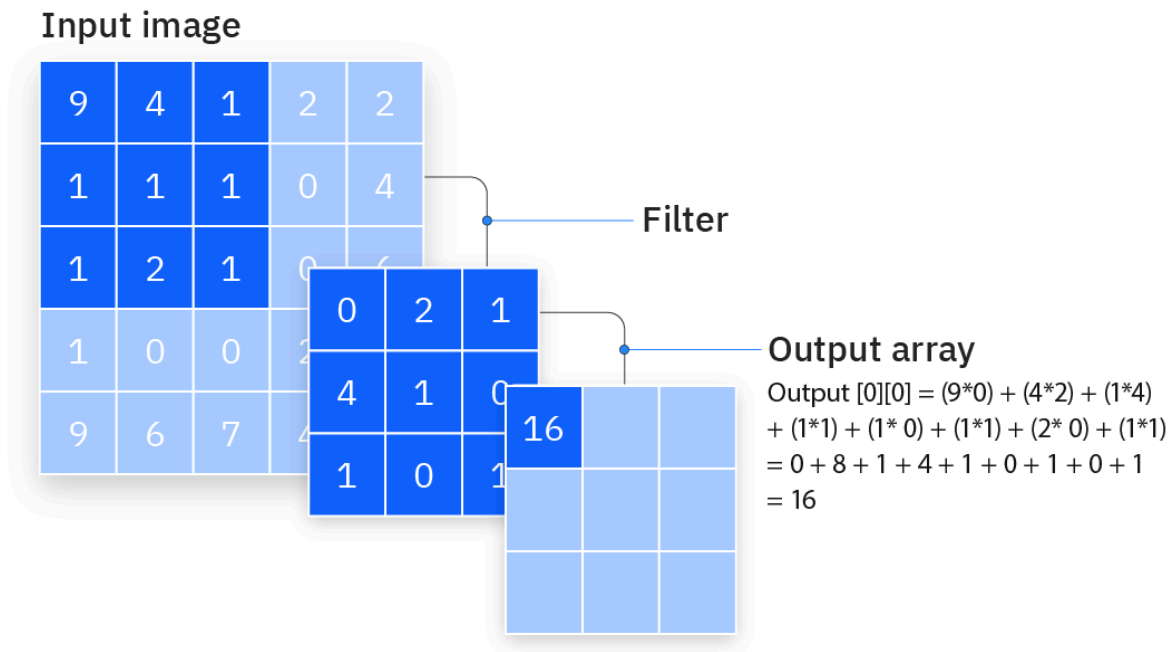
Image sur la façon dont la structure linéaire est organisée

La structure linéaire semble avoir 2 variables : le poids qui représente W et le biais représenté par B, avec ces 2 variables nous pouvons les utiliser dans l'équation linéaire **$O = IW + B$** .

Toutes les fonctions de la couche Linéaire se trouvent sur <https://burn.dev/docs/burn/nn/struct.Linear.html> .

Dans notre structure de blocs mamba, nous stockerons les 2 projections que nous avons réalisées.

La couche de convolution :



Le schéma montre une couche appelée couche convolutive. Cette couche fait passer un tenseur d'entrée à travers un filtre ; une fois le filtre passé sur l'entrée, il génère un tenseur de sortie.

Il existe trois types de couches convolutives dans Rust : 1D, 2D et 3D. Pour l'architecture Mamba, nous utiliserons uniquement la couche convolutive 1D. Cette couche utilise un objet appelé filtre, placé sur le tenseur d'entrée pour calculer les valeurs de sortie. L'équation est illustrée dans l'image ci-dessus.

Dans burn, la couche de convolution unidimensionnelle est codée comme suit.

burn::nn::conv

Struct **Conv1d**

Settings Help Summary

```
pub struct Conv1d<B>
where
  B: Backend,
{
  pub weight: Param<Tensor<B, 3>>,
  pub bias: Option<Param<Tensor<B, 1>>>,
  pub stride: usize,
  pub kernel_size: usize,
  pub dilation: usize,
  pub groups: usize,
  pub padding: Ignored<PaddingConfig1d>,
}
```

✓ Applies a 1D convolution over input tensors.

Should be created with [Conv1dConfig](#).

Nous pouvons découvrir ce que fait chaque variable en utilisant les champs fournis dans la documentation de burn :

Fields

weight: `Param<Tensor<B, 3>>`
Tensor of shape `[channels_out, channels_in / groups, kernel_size]`

bias: `Option<Param<Tensor<B, 1>>>`
Tensor of shape `[channels_out]`

stride: `usize`
Stride of the convolution.

kernel_size: `usize`
Size of the kernel.

dilation: `usize`
Spacing between kernel elements.

groups: `usize`
Controls the connections between input and output channels.

padding: `Ignored<PaddingConfig1d>`
Padding configuration.

Fonctions d'activation :

Que sont les fonctions d'activation ? Les fonctions d'activation sont cruciales pour façonner la sortie des réseaux de neurones. En tant qu'équations mathématiques, elles contrôlent la sortie des neurones du réseau, impactant à la fois les processus d'apprentissage et les prédictions du réseau. Elles y parviennent en régulant le signal transmis à la couche suivante, de 0 % (inactivité complète) à 100 % (pleine activité). Cette régulation influence significativement la précision du modèle, l'efficacité de l'apprentissage et sa capacité de généralisation à des données nouvelles et inédites.

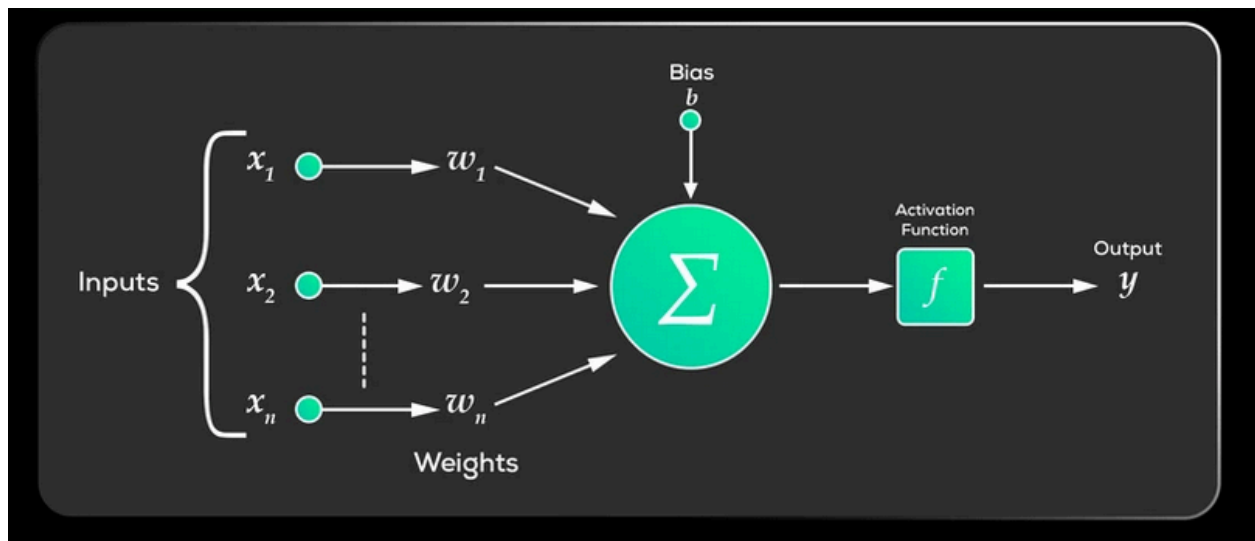
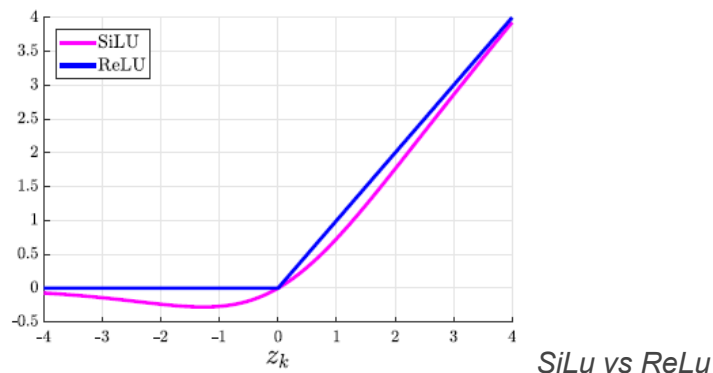


Diagramme de [Deppgram.com](https://deppgram.com)

Il existe plusieurs fonctions d'activation, chacune ayant des utilisations différentes. J'opte généralement pour la fonction d'activation ReLU, une fonction simple définie par l'équation $f(x) = \max(0, x)$. Celle-ci supprime toutes les valeurs négatives et ne laisse que les positives. Cependant, dans ce projet, nous utiliserons la fonction d'activation SiLU, définie par l'équation $\text{SiLU}(x) = x \cdot \sigma(x)$, où $\sigma(x)$ est la sigmoïde logistique. Le choix de SiLU plutôt que ReLU pour ce modèle particulier s'explique par le fait que l'article Mamba a été conçu en tenant compte de SiLU ; il offre de meilleures performances et préserve les gradients pendant l'apprentissage grâce à sa nature lisse et non monotone.







Dans Burn, nous pouvons utiliser la fonction d'activation SiLu comme indiqué dans l'image ci-dessous.

```
pub fn silu<const D: usize, B>(tensor: Tensor<B, D>) -> Tensor<B, D>
where
    B: Backend,
```

Il existe de nombreuses fonctions d'activation dans Burn et dans l'image ci-dessous seront toutes les fonctions d'activation actuellement disponibles.

burn::tensor

Module activation 

 Settings  Help  Summary

▼ The activation module.

Functions

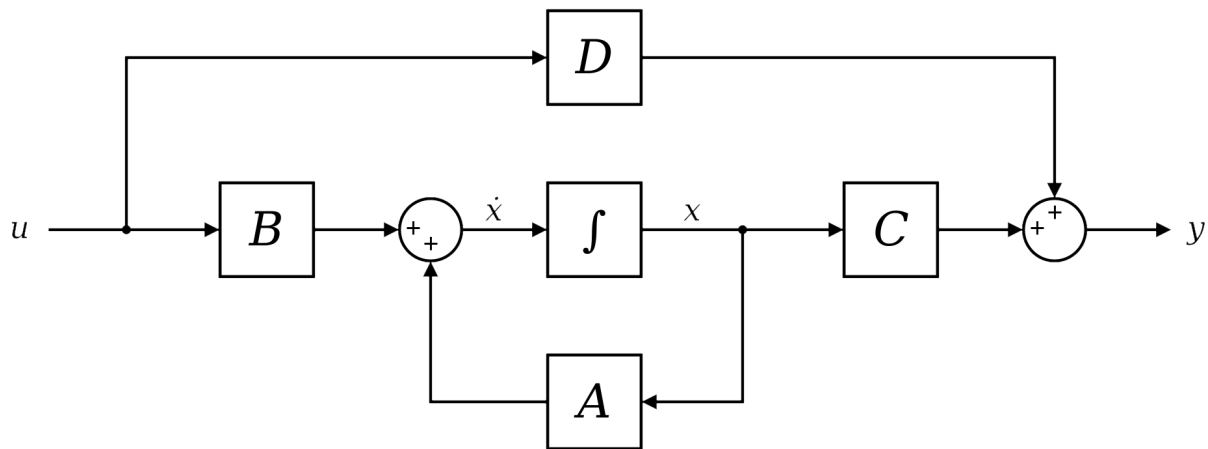
gelu	Applies the Gaussian Error Linear Units function as described in the paper Gaussian Error Linear Units (GELUs) .
hard_sigmoid	Applies the hard sigmoid function element-wise.
leaky_relu	Applies the leaky rectified linear unit function element-wise.
log_sigmoid	Applies the log sigmoid function element-wise.
log_softmax	Applies the log softmax function on the input tensor along the given dimension.
mish	Applies the Mish function as described in the paper in Mish: A Self Regularized Non-Monotonic Neural Activation Function .
prelu	Applies Parametric ReLu activation function as described in the paper Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification .
quiet_softmax	Applies the “quiet softmax” function on the input tensor along the given dimension.
relu	Applies the rectified linear unit function element-wise as described in the paper Deep Learning using Rectified Linear Units (ReLU) .
sigmoid	Applies the sigmoid function element-wise.
silu	Applies the SiLU function (also known as the swish function) element-wise.
softmax	Applies the softmax function on the input tensor along the given dimension.
softmin	Applies the softmin function on the input tensor along the given dimension.
softplus	Applies the SoftPlus function element-wise.
tanh	Applies the tanh function element-wise.

Je pense qu'il est important d'examiner toutes les fonctions d'activation pour au moins savoir si elles existent. Il est également intéressant de les tester sur le même modèle afin de déterminer ce qui est le plus performant pour la même tâche.

Modèles d'espace d'état sélectifs :

Dans cette section consacrée au modèle d'espace d'état sélectif, nous le décomposons en deux parties. La première se concentrera sur le modèle d'espace d'état : son fonctionnement, son développeur et son implémentation en Rust. La seconde partie intégrera le mécanisme de sélection à l'aide d'une analyse sélective.

Qu'est-ce qu'un modèle d'espace d'État ? C'est un concept difficile à expliquer simplement par écrit. Commençons donc par un schéma pour comprendre son fonctionnement.



Dans le diagramme, nous voyons qu'il prend une entrée u et renvoie une sortie y .

A (matrice $n \times n$) : La matrice du système décrit l'évolution de l'état (dynamique interne).

B (matrice $n \times m$) : La matrice d'entrée montre comment les entrées affectent les changements d'état.

C (matrice $p \times n$) : La matrice de sortie associe les états aux sorties.

D (matrice $p \times m$) : La matrice de traversée influence directement les entrées sur les sorties (souvent nulle pour les systèmes physiques).

Je vais fournir un exemple de code en python que nous traduirons en rust :

```
def ssm(self, x):
    (d_in, n) = self.A_log.shape
    A = -torch.exp(self.A_log.float()) # shape (d_in, n)
    D = self.D.float()
    x_dbl = self.x_proj(x) # (b, 1, dt_rank + 2*n)
    (delta, B, C) = x_dbl.split(split_size=[self.args.dtRank, n, n], dim=-1)
    delta = F.softplus(self.dt_proj(delta)) # (b, 1, d_in)
```

Grâce à cette fonction, nous pouvons calculer les valeurs suivantes [Delta , A , B , C , D] dans cet exemple x représente la variable u.

Nous travaillons avec un SSSM (modèle sélectif d'espace d'état). Il manque une étape à notre fonction SSM pour qu'elle soit pleinement fonctionnelle. Nous allons fournir x, delta, A, B, C et D via une fonction appelée balayage sélectif.

L'analyse sélective implémente la récurrence SSM à temps discret

Sa sélectivité empêche l'équivalence de convolution ; l'analyse utilise donc une analyse associative parallèle récurrente (inspirée des sommes de préfixes) pour calculer tous les hth_tht en parallèle sur la séquence. Pour un lot B, elle traite les tenseurs de forme (B, L, D, N) en O(BLDN) FLOPs.

Un exemple de fonction codée en python est présenté ci-dessous :

```
def selective_scan(u, delta, A, B, C, D):
    dA = torch.einsum('bld,dn->bldn', delta, A)
    dB_u = torch.einsum('bld,bld,bln->bldn', delta, u, B)

    dA_cumsum = torch.nn.functional.pad(dA[:, 1:], (0, 0, 0, 0, 1, 1, 0, 0))[:, 1:, :, :]

    dA_cumsum = torch.flip(dA_cumsum, dims=[1])

    dA_cumsum = torch.cumsum(dA_cumsum, dim=1)

    dA_cumsum = torch.exp(dA_cumsum)
    dA_cumsum = torch.flip(dA_cumsum, dims=[1])

    x = dB_u * dA_cumsum
    x = torch.cumsum(x, dim=1) / (dA_cumsum + 1e-12)

    y = torch.einsum('bldn,bln->bld', x, C)

    return y + u * D
```

C'est l'un des concepts les plus difficiles à expliquer et l'un des points clés de l'architecture Mamba. J'ai ajouté des liens pour une explication plus détaillée. (Je reviendrai sur le sujet)

<https://www.youtube.com/watch?v=N6Piou4oYx8&t>

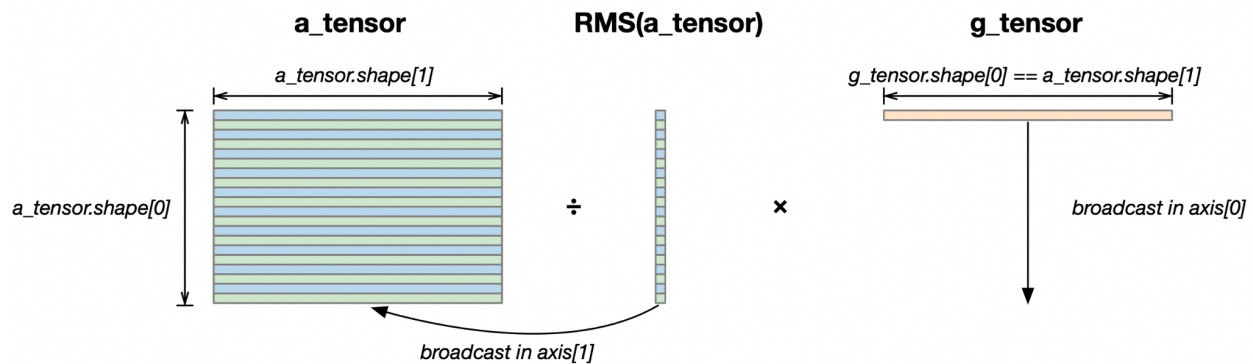
<https://www.youtube.com/watch?v=9dSkvxS2EB0&t>

https://www.youtube.com/watch?v=8Q_tqwpTpVU

Warning : Les vidéos sont un peu longue.

Normalisation :

Dans notre cas, nous utiliserons la normalisation par la moyenne quadratique (RMSNorm). Cette technique de normalisation est largement utilisée dans les modèles Transformer et Mamba.



Start with input: You have some data x (like a vector of numbers from the model). Multiply it by a weight matrix W to get $a = Wx$. This is just transforming the input a bit (common in neural nets).

Compute the RMS: RMS stands for Root Mean Square it's like the "average size" of the values in a , ignoring signs.

The formula:

$$\begin{aligned} \text{RMSNorm} : \\ a &= Wx \\ \text{RMS}(a) &= \sqrt{\frac{1}{n} \sum_{i=1}^n a_i^2} \\ \overline{a_i} &= \frac{a_i}{\text{RMS}(a)} \end{aligned}$$

n is the length of a (number of elements).

Élevez au carré chaque a_i (pour se concentrer sur la grandeur), faites la moyenne, puis prenez la racine carrée.

Exemple : Si $a = [3, 4]$, les carrés sont $[9, 16]$, moyenne = 12,5, RMS $\approx 3,54$.

Divisez chaque élément par cette valeur RMS. Cela met tout à l'échelle, de sorte que la « taille moyenne » passe à 1, préserve la forme, mais atténue les pics importants/petits.

Dans le prototype que j'ai réalisé, la classe RMSNorm était codée comme suit :

```
class RMSNorm(nn.Module):
    def __init__(self, dModel: int, eps: float = 1e-5):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dModel))

    def forward(self, x):
        output = x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps) * self.weight
        return output
```

Comme vous pouvez le voir, l'équation semble exagérée, mais en réalité, elle est un peu plus simple qu'il n'y paraît.

Conclusion :

Nous avons réussi à décomposer le diagramme complexe en problèmes plus petits et plus complexes. Il nous reste maintenant à gérer les éléments suivants dans notre version finale en Rust : les entrées, les projections, la couche de convolution, les fonctions d'activation, le modèle d'espace d'état sélectif et la normalisation. Certains de ces problèmes se présenteront plusieurs fois, mais sont assez faciles à résoudre.

Il existe également différentes étapes de développement de l'IA que je n'avais pas précisées auparavant :

- Création de la structure du modèle
- Collecte des données
- Entraînement du modèle
- Inférence (lancement de l'IA à l'aide de celui-ci)

créerai, je me concentrerai sur la création de la structure du modèle en rouille avec les différentes classes que nous pouvons posséder