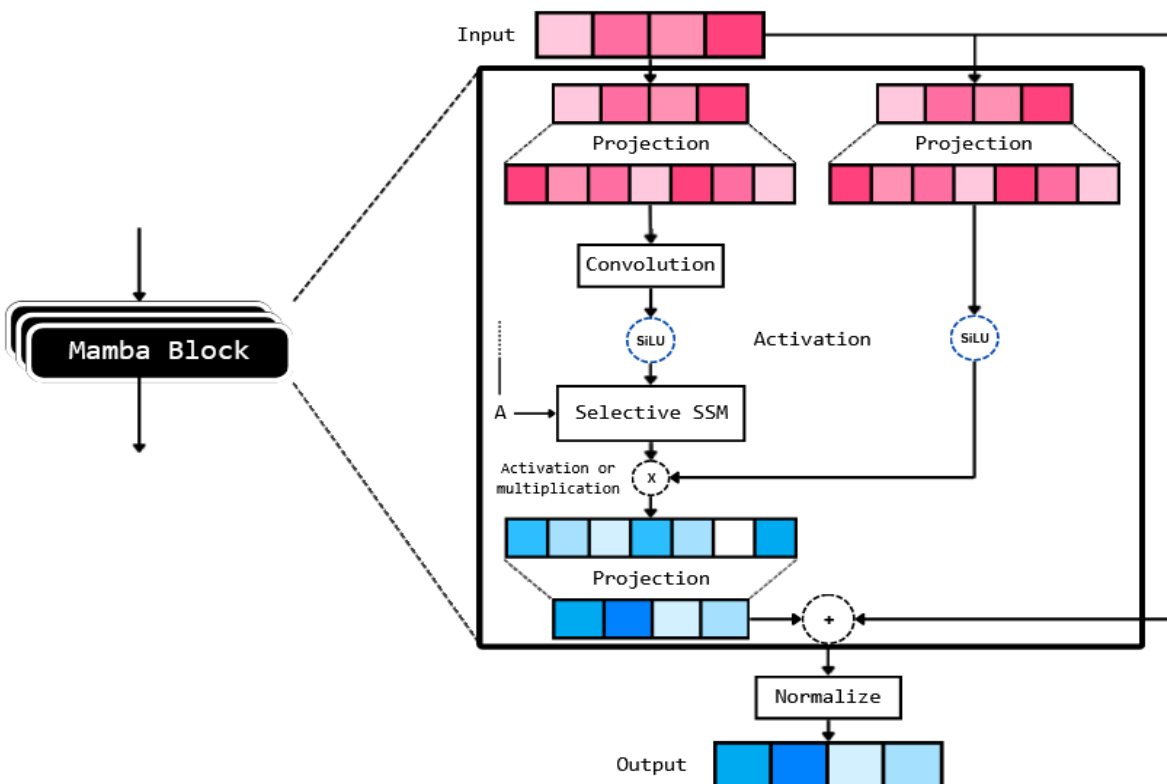


# Mamba from scratch (RUST edition)

Audric HARRIS

## Chapter 1 : Breaking down the problem

It's important that we take our time to fully understand how Mamba works; if we try to code something that we ourselves don't understand, then we aren't going to get far. I will start off by presenting the Mamba model as shown in the picture below and breaking it down step by step what a singular Mamba block does.



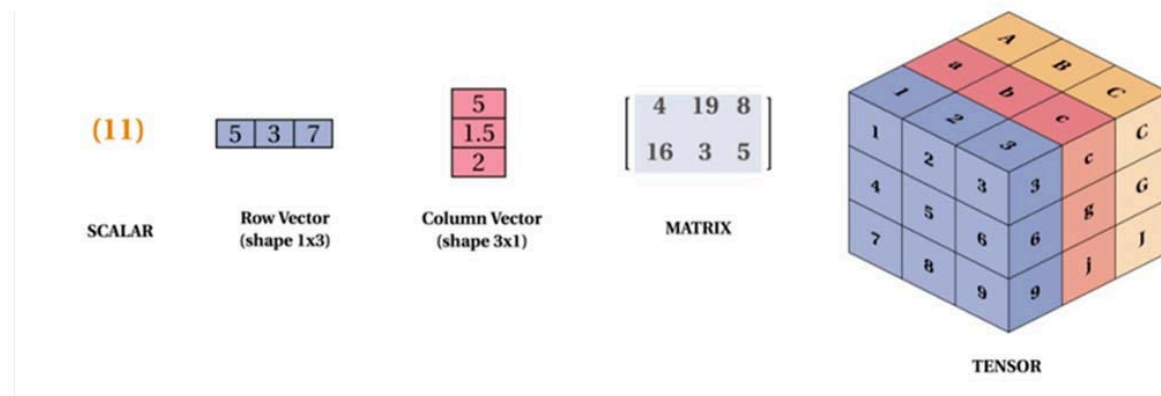
*The diagram shows a singular ResidualBlock also named mamba block.*

Large language models typically have multiple residual blocks; in our case, we will have 10, which are also called layers. But at first, we will only be focusing on a single layer to simplify explanations.

## The inputs :

What type of input does a Mamba block take? You might think it's a word or sentence, but that's incorrect. What a Mamba block takes is a tensor. Every word and sentence will have its own tensor representation, and the Mamba model will be able to predict another tensor based on an existing one. In our case, we'll only be talking about tensors because the words only come into play before and after computation.

Tensor definition : a mathematical object analogous to but more general than a vector, represented by an array of components that are functions of the coordinates of a space.



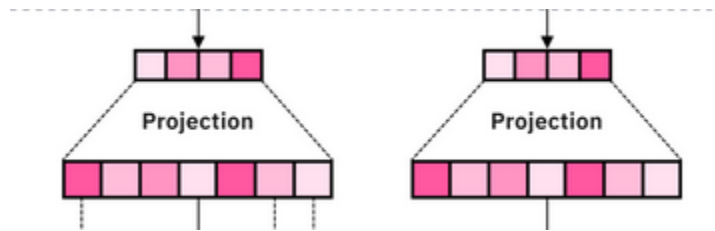
*Diagram of a tensor and similar mathematical objects to help for understanding*

Now that we've fed the tensor to our Mamba block, what does it do with it? It first keeps a copy in memory the larger the tensor, the more information it can store. The downside is that training Mamba will take longer to process these larger tensors and will require more model parameters. (We'll discuss parameters distinct from hyperparameters later in the docs.)

Our model saves one instance of the original tensor as an attribute; typically, we don't store input variables in a struct and instead rely on those passed to the forward function. However, what gets stored in the struct here are two instances of a projection of this original tensor.

## The projections :

In the context of a Mamba block, a linear projection is a fundamental operation that transforms the input tensor (let's call it  $x \times x$ ) into a new representation by applying a learnable linear transformation:  $y = Wx + b$ , where  $W$  is a weight matrix (with dimensions that define the output size) and  $b$  is an optional bias vector. This is essentially a matrix multiplication followed by addition, implemented efficiently in frameworks like Burn via fully connected (dense) layers. The purpose here is to expand or reshape the tensor into a higher-dimensional space suited for the selective state space model (SSM) components for instance, projecting  $x \times x$  to create separate pathways for parameters like the SSM's  $A$ ,  $B$ ,  $C$  matrices and the time-variant  $\Delta$ , allowing the model to selectively attend to sequence information. In our implementation, this projection typically increases the tensor's hidden dimension (e.g., from  $d_{\text{model}}$  to  $2 \times d_{\text{state}}$ ) to provide richer features for the recurrent computation, though later projections (like output ones) might down-project back to the original size for efficiency. During training, the weights  $W$  and biases  $b$  are optimized via backpropagation, enabling the model to learn task-specific transformations without altering the core sequence length.



Now that I explained how linear projection works, how do we do this in rust? Under the burn documentation we have `burn::nn.Linear`.

```
burn::nn
Struct Linear
Settings Help Summary

pub struct Linear<B>
where
  B: Backend,
{
  pub weight: Param<Tensor<B, 2>>,
  pub bias: Option<Param<Tensor<B, 1>>>,
}

▽ Applies a linear transformation to the input tensor.
Should be created with LinearConfig

O = IW + b
```

*Image on how the linear structure is organised*

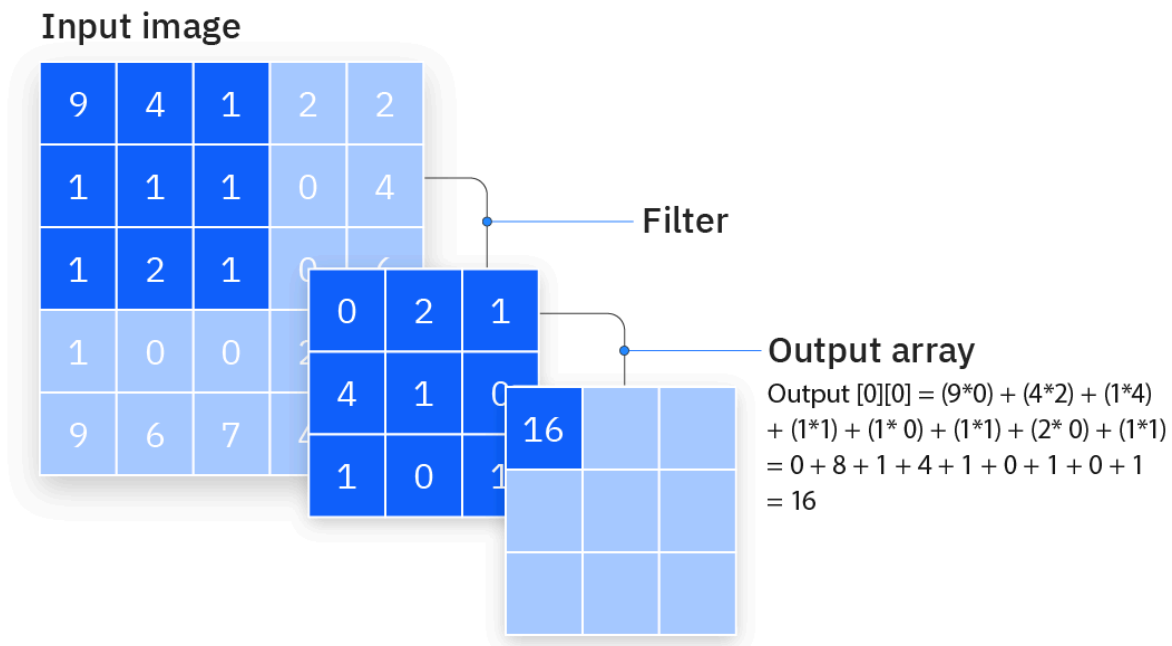
The Linear structure seems to have 2 variables weight that represents W and bias represented by B, with those 2 variables we are able to use this in the linear equation  **$O = IW+B$** .

All the functions for the Linear layer can be found on <https://burn.dev/docs/burn/nn/struct.Linear.html> .

In our mamba block structure we will store the 2 projections that we made.

## The convolution layer :


The diagram shows a layer called the convolutional layer. This layer passes an input tensor through a filter; once the filter has slid over the input, it generates an output tensor.



There are three types of convolutional layers in Rust: 1D, 2D, and 3D. For the Mamba architecture, we will use only the 1D convolutional layer. This layer uses an object called a filter, which is placed over the input tensor to compute the output values. The equation is shown in the image above.

In burn the 1 dimensional convolution layer is coded as the following.

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](#).

We can find out what each variable does using the fields provided in the burn documentation :

### 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.

## Activation Functions :

What are activation functions ? Activation functions are crucial in shaping the output of neural networks. As mathematical equations, they control the output of the network's neurons, impacting both the learning processes and predictions of the network. They achieve this by regulating the signal transmitted to the next layer, ranging from 0% (complete inactivity) to 100% (full activity). This regulation significantly influences the model's accuracy, learning efficiency, and generalization ability on new, unseen data.

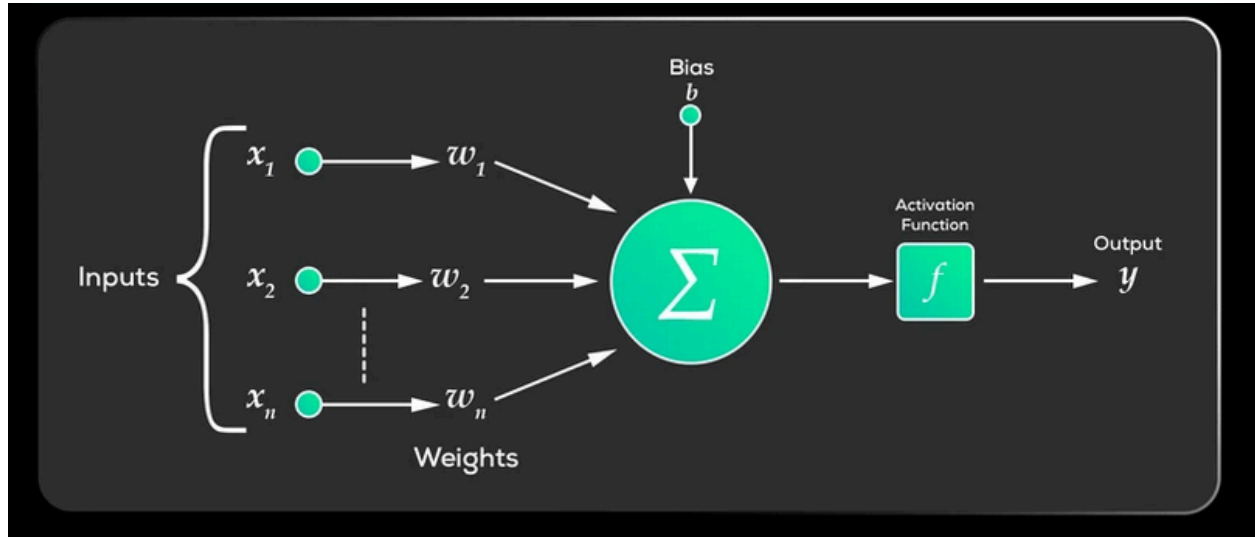
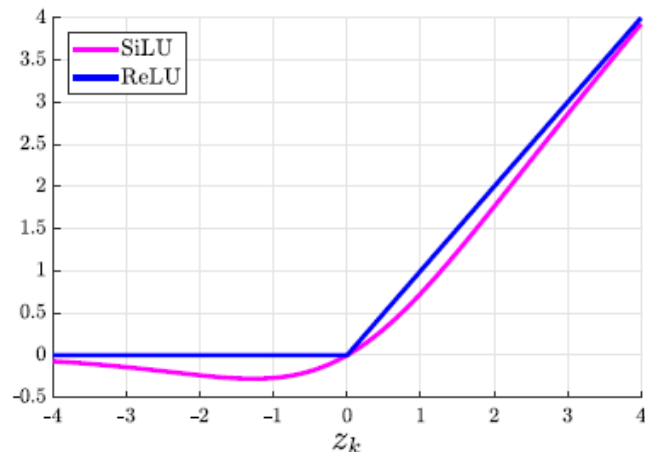


Diagram from [Deppgram.com](https://deppgram.com)

There are multiple activation functions, each with different uses. I usually opt for the ReLU activation function, which is a simple one defined by the equation  $f(x) = \max(0, x)$ . This removes all negative values, leaving only positives. However, in this project, we will use the SiLU activation function, defined by the equation  $\text{SiLU}(x) = x \cdot \sigma(x)$ , where  $\sigma(x)$  is the logistic sigmoid. The reason for choosing SiLU over ReLU for this particular model is that the Mamba paper was designed with SiLU in mind; it enables better performance and preserves gradients during training due to its smooth and non-monotonic nature.




SiLu vs ReLu




In burn we can use the SiLu activation function as shown in the image below.


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

There are a lot of activation functions in burn and in the image below will be all currently available activation functions.

burn::tensor

Module activation 

 Settings  Help  Summary

 The activation module.

### Functions

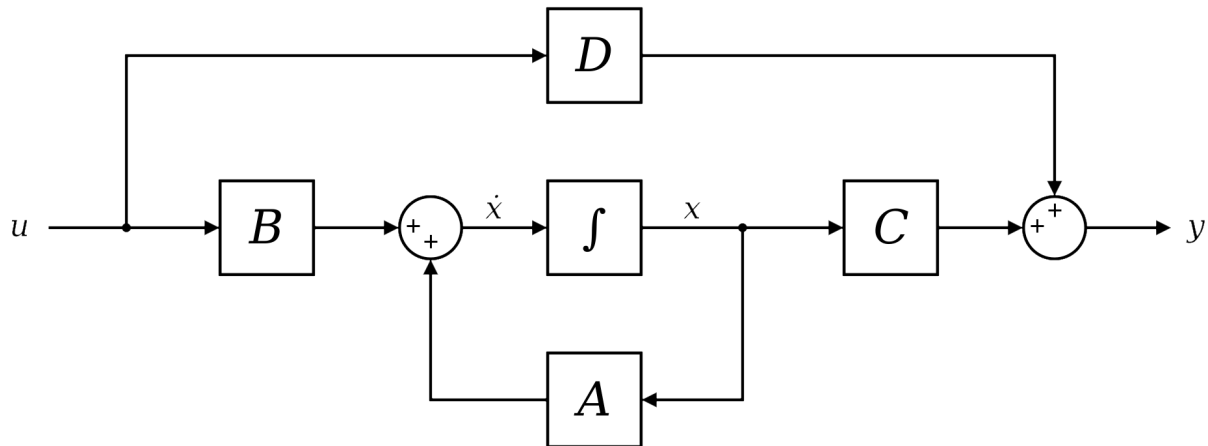
gelu	Applies the Gaussian Error Linear Units function as described in the paper <a href="#">Gaussian Error Linear Units (GELUs)</a> .
hard_sigmoid	Applies the hard sigmoid function element-wise.
leaky_relu	Applies the leaky rectified linear unit function element-wise.
log_sigmoid	Applies the log sigmoid 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 <a href="#">Mish: A Self Regularized Non-Monotonic Neural Activation Function</a> .
prelu	Applies Parametric ReLu activation function as described in the paper <a href="#">Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification</a> .
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 <a href="#">Deep Learning using Rectified Linear Units (ReLU)</a> .
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.

I think it's important to take a look at all activation functions to at least know they exist. It's also interesting to test them out for the same model to see what performs better for the same task.

## Selective State Space Models :

In this section on the selective state space model, we will break it down into two parts. The first part will focus on the state space model: how it works, who developed it, and how we will implement it in Rust. In the second part, we will integrate the selective mechanism using a selective scan.

What is a State space model ? It's a bit of a hard concept to explain just in text so let's start off with a diagram and work out how it works from there.



In the diagram we see that it takes an input  $u$  and returns an output  $y$ .

**A ( $n \times n$  matrix): System matrix** describes how the state evolves on its own (internal dynamics).

**B ( $n \times m$  matrix): Input matrix** shows how inputs affect state changes.

**C ( $p \times n$  matrix): Output matrix** maps states to outputs.

**D ( $p \times m$  matrix): Feedthrough matrix** direct influence of inputs on outputs (often zero for physical systems).

I will provide an example code in python that we will translate in 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)
```

Thanks to this function we are able to calculate the following values  
[ Delta , A , B , C D ] in this example x represents the variable u.



We are working with an SSSM (Selective state space model). One step is missing in our SSM function for it to be fully complete. What we will do is provide x, delta, A, B, C, D through a function called selective scan.

The selective scan implements the discrete-time SSM recurrence.

It selectivity prevents convolution equivalence, so the scan uses a recurrent parallel associative scan (inspired by prefix sums) to compute all  $h_{th\_t}$  in parallel across the sequence. For a batch B, it processes tensors of shape (B,L,D,N) in O(BLDN) FLOPs.

An example of the function coded in python is shown below :

```
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
```

This is one of the hardest concepts to explain and main focus points of the mamba architecture. I have added links allowing for a more pushed explanation.

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

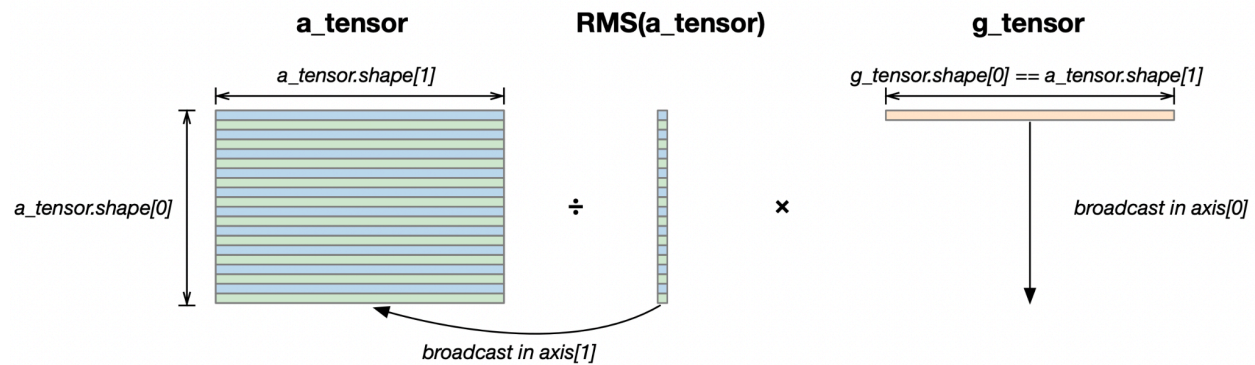
<https://www.youtube.com/watch?v=9dSkvxS2EB0&t>

[https://www.youtube.com/watch?v=8Q\\_tqwpTpVU](https://www.youtube.com/watch?v=8Q_tqwpTpVU)

Warning : The videos are really long.

## Normalizing :

In our case we will be using Root Mean Square Normalization (RMSNorm). This normalization technique is widely used across Transformer models and mamba models.



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:

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

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

Square each  $a_i$  (to focus on magnitude), average them, then take the square root.

Example: If  $a = [3, 4]$ , squares are  $[9, 16]$ , average = 12.5,  $\text{RMS} \approx 3.54$ .

Divide each element by this RMS value.

This scales everything so the "average size" becomes 1 keeps the shape but tames big/small spikes.

In the prototype I made the RMSNorm class was coded as follows :

```
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
```

As you can see the equation feels over the top but in reality it's a bit easier than it looks.

## Conclusion :

We managed to break down the complex diagram into smaller complex problems. We now have to manage the following in our final version in Rust ( The inputs, the projections, the convolution layer, activation functions, the selective state space model and normalizing). Some of these problems will appear multiple times but are quite easy to handle.

There are also different stages in AI development that I didn't precise before

- Creating the model structure
- Gathering Data
- Training the model
- Inference (Launching the AI using it)

In the next documentation page I make I will focus on creating the model structure in rust with the different classes we may possess