

# A Visual Guide to Mixture of Experts (MoE)

Demystifying the role of MoE in Large Language Models



MAARTEN GROOTENDORST

OCT 07, 2024



207

11

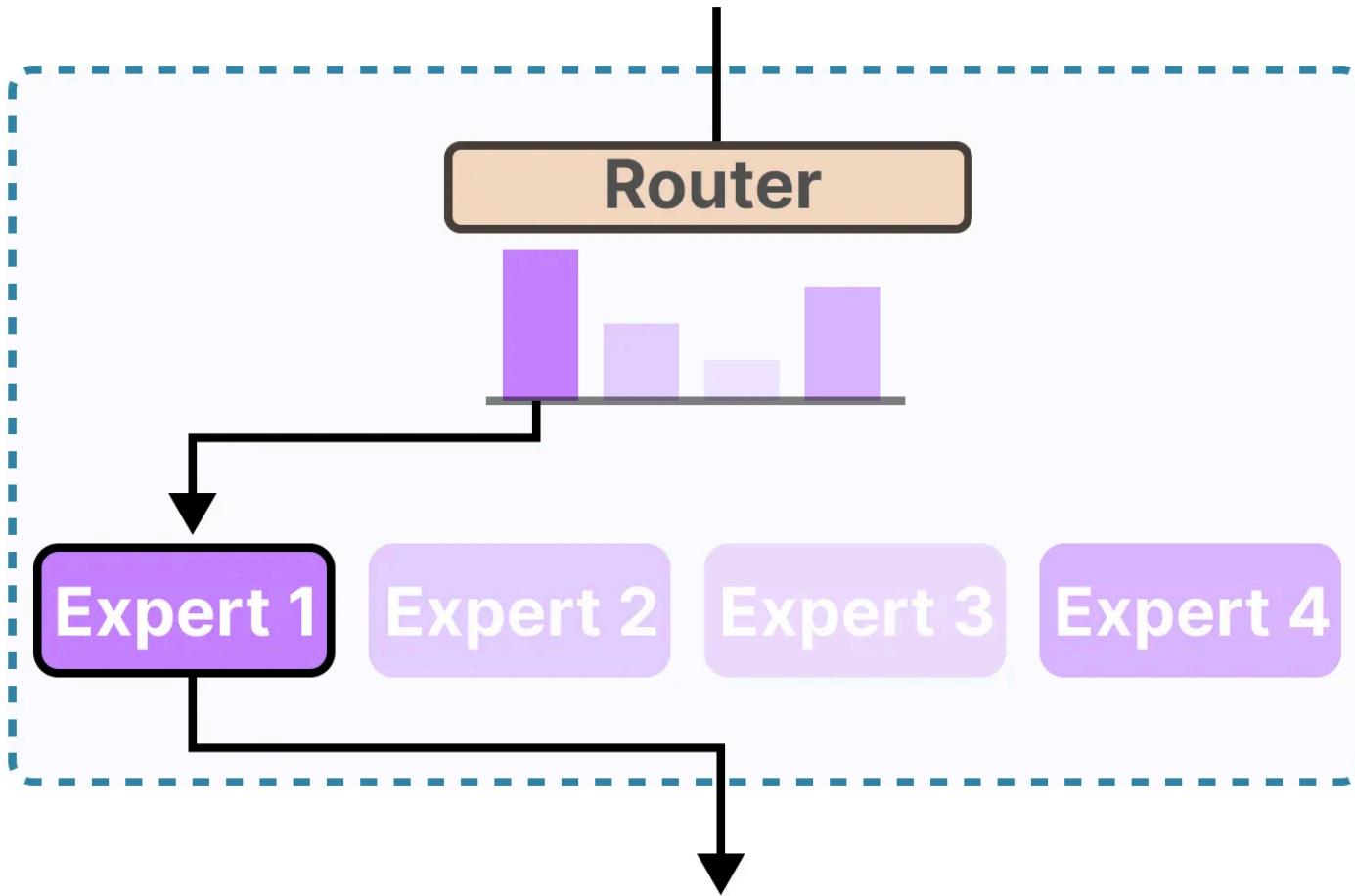
19

Share

Translations - [Korean](#) | Also check out the [YouTube](#) version with lots of animations!

When looking at the latest releases of Large Language Models (LLMs), you will often see “MoE” in the title. What does this “MoE” represent and why are so many LLMs using it?

In this visual guide, we will take our time to explore this important component, *Mixture of Experts* (MoE) through more than 50 visualizations!



In this visual guide, we will go through the two main components of MoE, namely **Experts** and the **Router**, as applied in typical LLM-based architectures.

---

To see a **Table of Contents** (ToC), click on the stack of lines on the left-hand side.

---

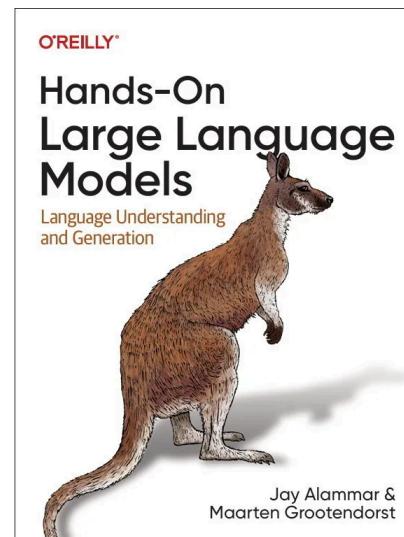
Thanks for reading *Exploring Language Models*!

Subscribe to receive new posts on Gen AI and  
the book: [Hands-On Large Language Models](#)

Type your email...

Subscribe

To see more visualizations related to LLMs and to support this newsletter, check out  
the book I wrote on Large Language Models!



[Official website](#) of the book. You can order the book on [Amazon](#). All code is uploaded to [GitHub](#).

---

*P.S. If you read the book, a [quick review](#) would mean the world—it really helps us authors!*

---

## What is Mixture of Experts?

Mixture of Experts (MoE) is a technique that uses many different sub-models (or “experts”) to improve the quality of LLMs.

Two main components define a MoE:

- **Experts** - Each FFNN layer now has a set of “experts” of which a subset can be chosen. These “experts” are typically FFNNs themselves.
- **Router or gate network** - Determines which tokens are sent to which experts.

In each layer of an LLM with an MoE, we find (somewhat specialized) experts:

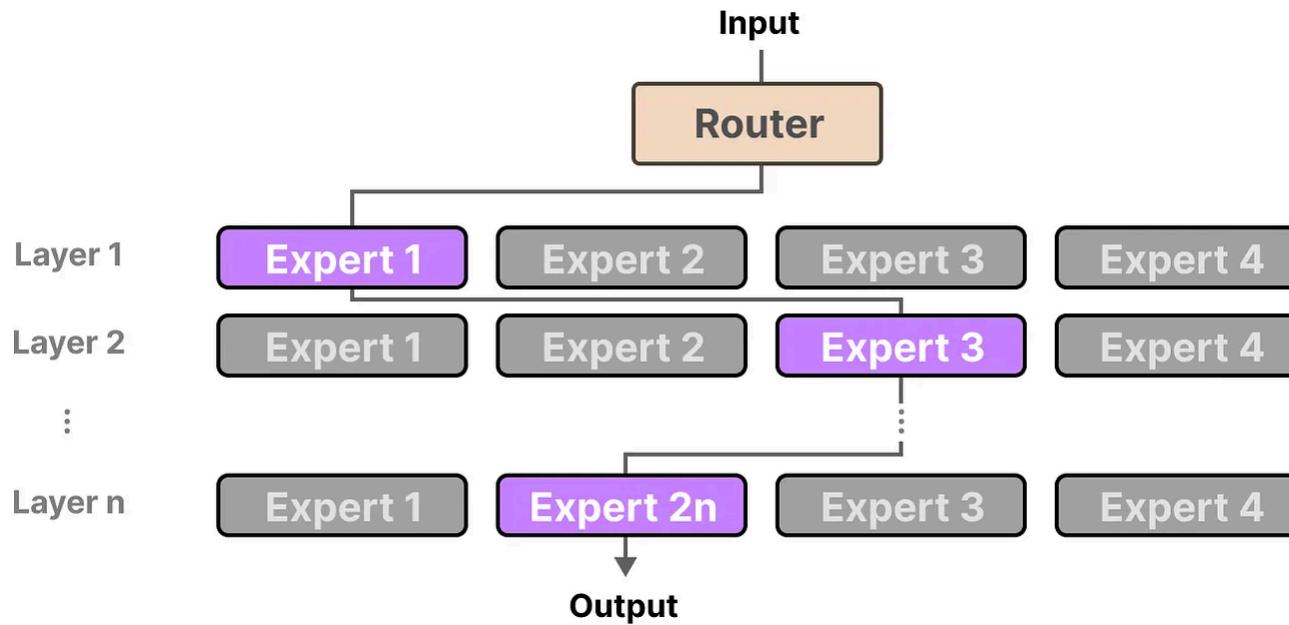


Know that an “expert” is not specialized in a specific domain like “Psychology” or “Biology”. At most, it learns syntactic information on a word level instead:



More specifically, their expertise is in handling specific tokens in specific contexts.

The *router* (gate network) selects the expert(s) best suited for a given input:



Each expert is not an entire LLM but a submodel part of an LLM's architecture.

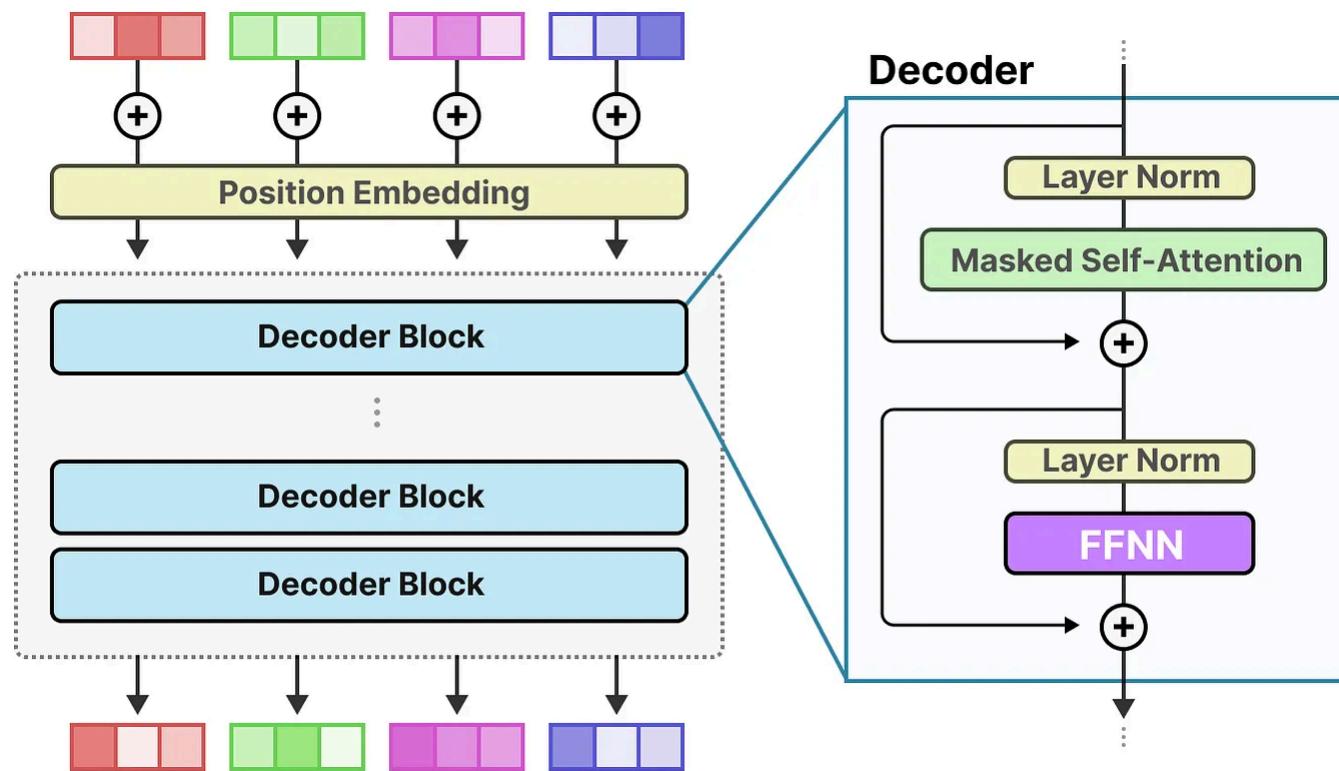
## The Experts

To explore what experts represent and how they work, let us first examine what MoE is supposed to replace; the *dense layers*.

## Dense Layers

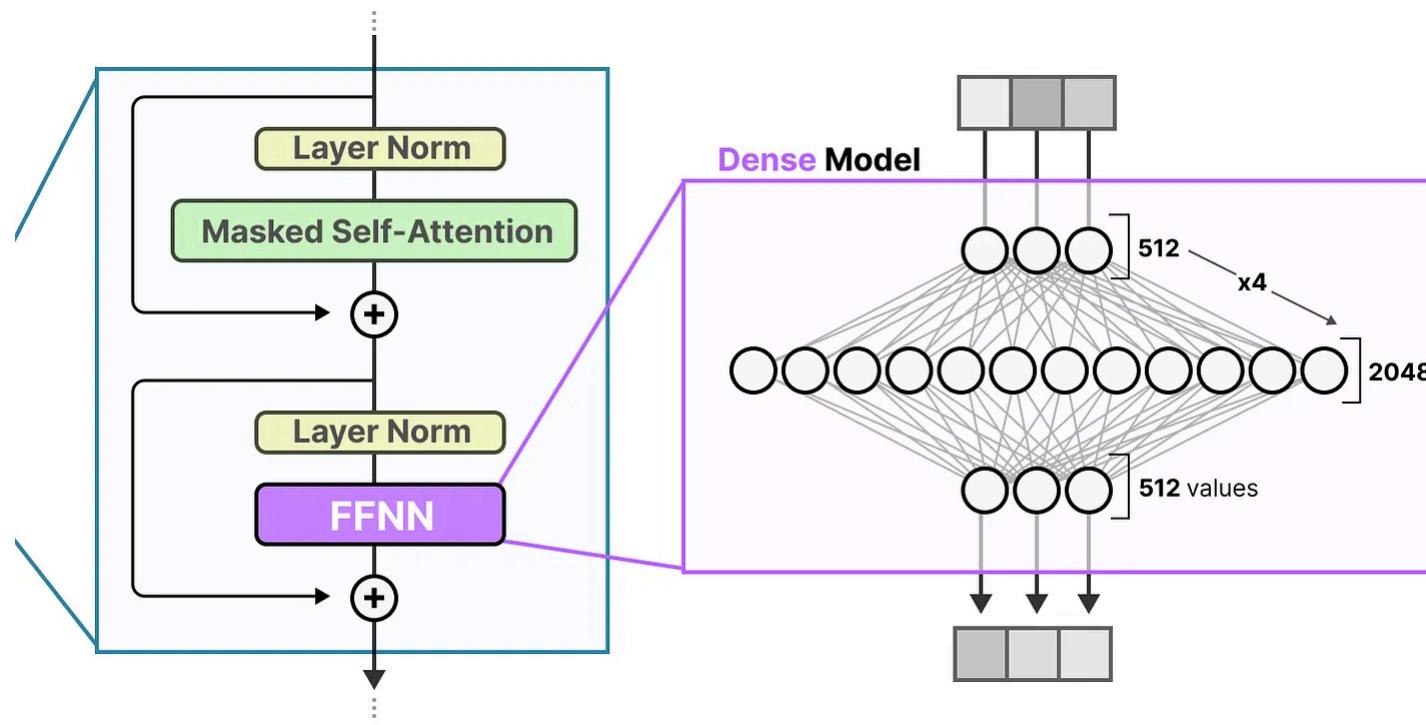
Mixture of Experts (MoE) all start from a relatively basic functionality of LLMs, namely the *Feedforward Neural Network* (FFNN).

Remember that a standard decoder-only Transformer architecture has the FFNN applied after layer normalization:



An FFNN allows the model to use the contextual information created by the attention mechanism, transforming it further to capture more complex relationships in the data.

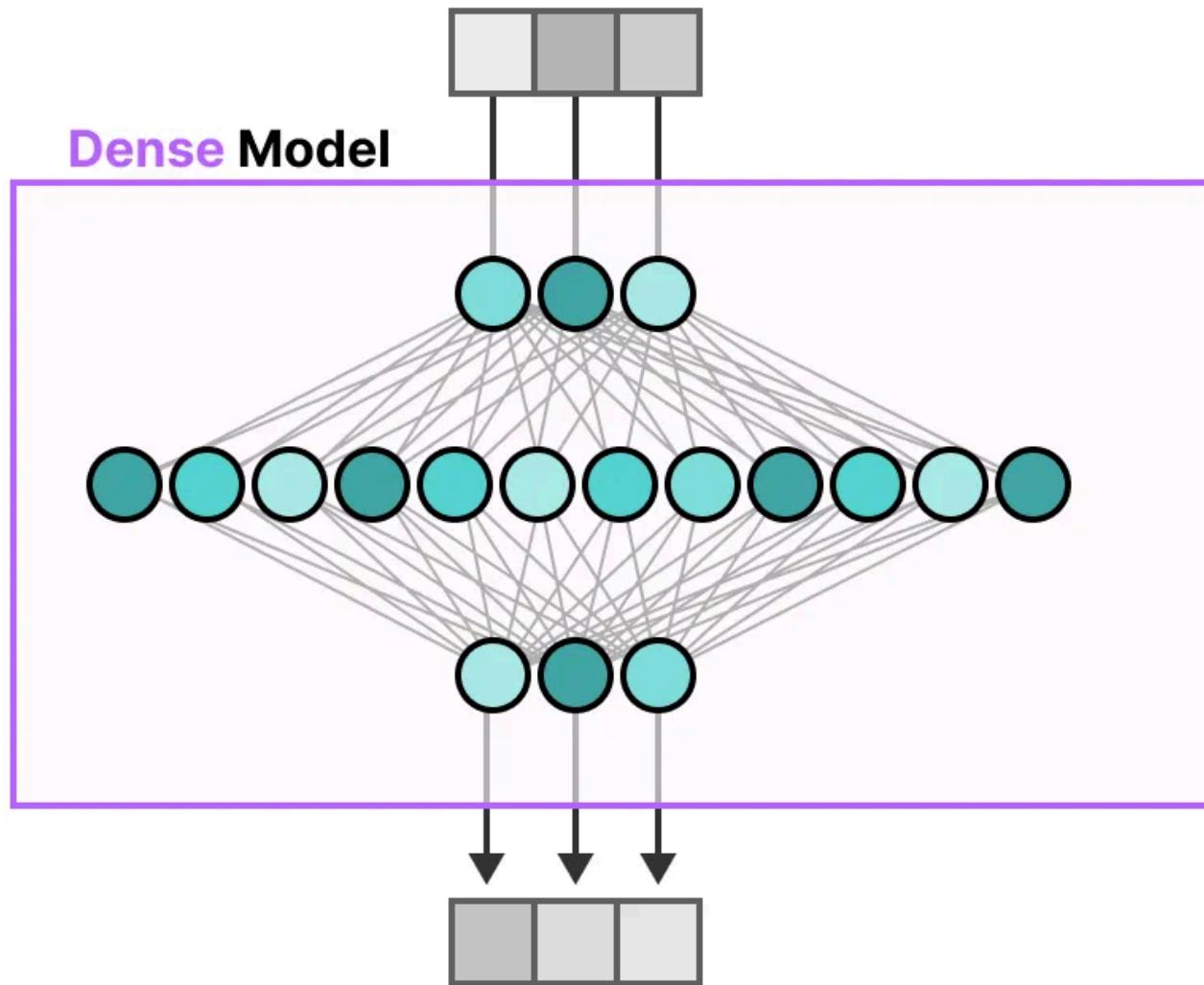
The FFNN, however, does grow quickly in size. To learn these complex relationships, it typically expands on the input it receives:



## Sparse Layers

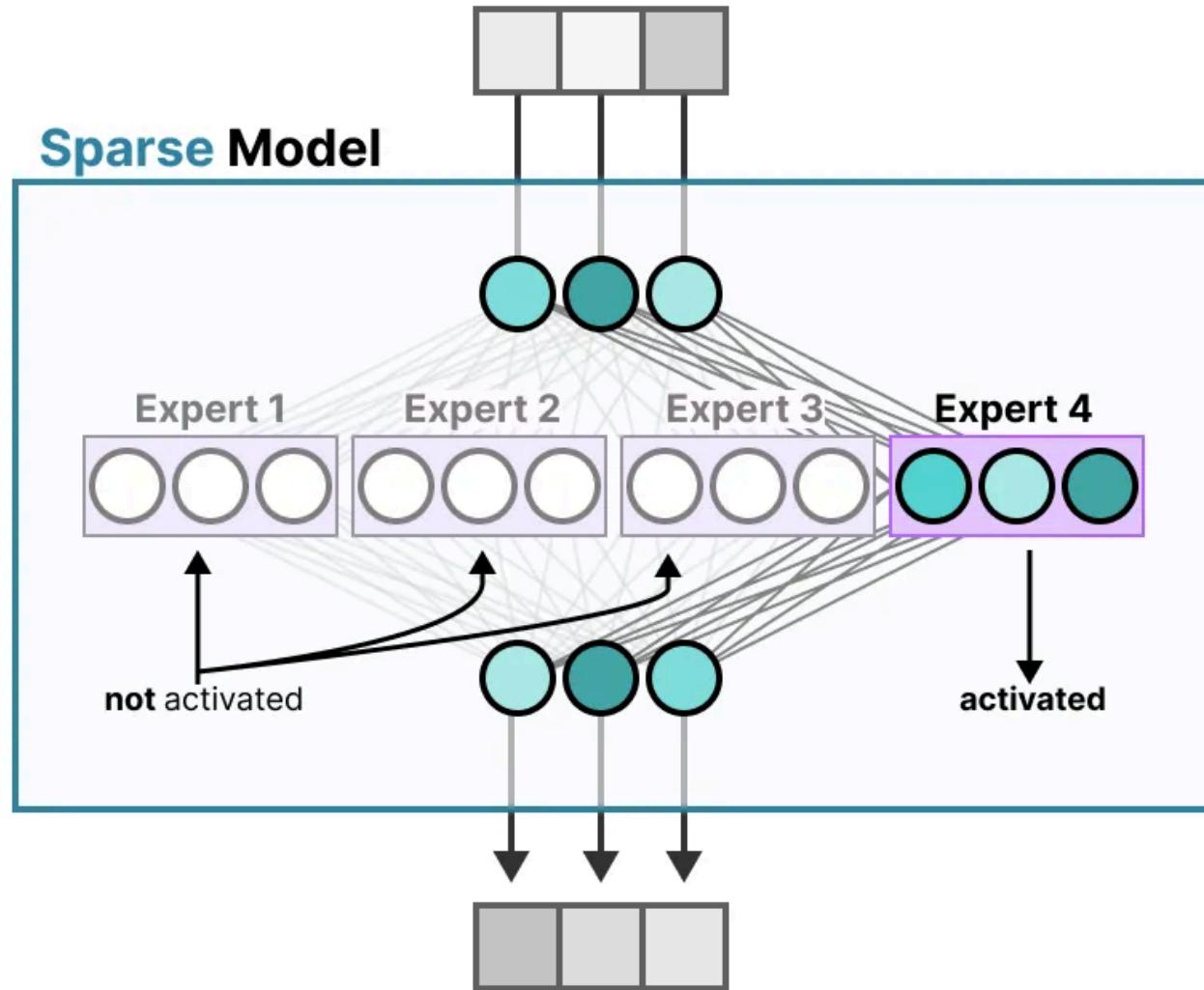
The FFNN in a traditional Transformer is called a **dense** model since all parameters (its weights and biases) are activated. Nothing is left behind and everything is used to calculate the output.

If we take a closer look at the dense model, notice how the input activates all parameters to some degree:



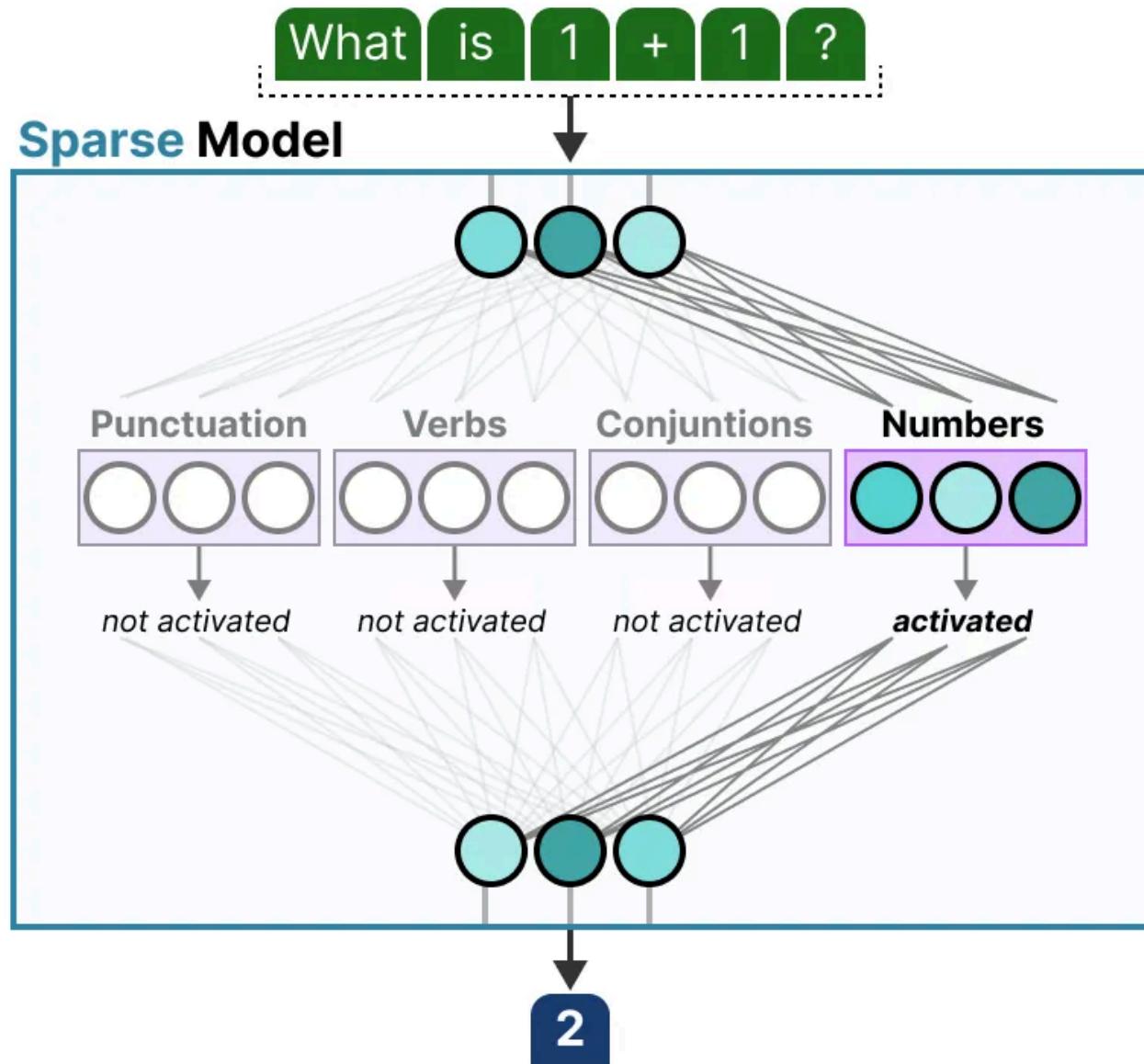
In contrast, **sparse** models only activate a portion of their total parameters and are closely related to Mixture of Experts.

To illustrate, we can chop up our dense model into pieces (so-called experts), retrain it, and only activate a subset of experts at a given time:



The underlying idea is that each expert learns different information during training. Then, when running inference, only specific experts are used as they are most relevant for a given task.

When asked a question, we can select the expert best suited for a given task:



# What does an Expert Learn?

As we have seen before, experts learn more fine-grained information than entire domains.<sup>1</sup> As such, calling them “experts” has sometimes been seen as misleading.

Expert specialization	Expert position	Routed tokens
<b>Punctuation</b>	Layer 2	, . , . . . - . . . , ) . )
	Layer 6	, . , : . , & , & & ? &-,,? „ „ .
<b>Conjunctions and articles</b>	Layer 3	The the the the the the the The...
	Layer 6	a and and and and and and or and ...
<b>Verbs</b>	Layer 1	died falling identified fell closed left posted lost felt left said read miss place struggling falling signed died...
<b>Visual descriptions</b> <i>color, spatial position</i>	Layer 0	her over her know dark upper dark outer center upper blue inner yellow raw mama bright bright over open your dark blue
<b>Counting and numbers</b> <i>written and numerical forms</i>	Layer 1	after 37 19. 6. 27 11 Seven 25 4, 54 1 two dead we Some 2012 who we few lower

Expert specialization of an encoder model in the ST-MoE paper.

Experts in decoder models, however, do not seem to have the same type of specialization. That does not mean though that all experts are equal.

A great example can be found in the [Mixtral 8x7B paper](#) where each token is colored with the first expert choice.

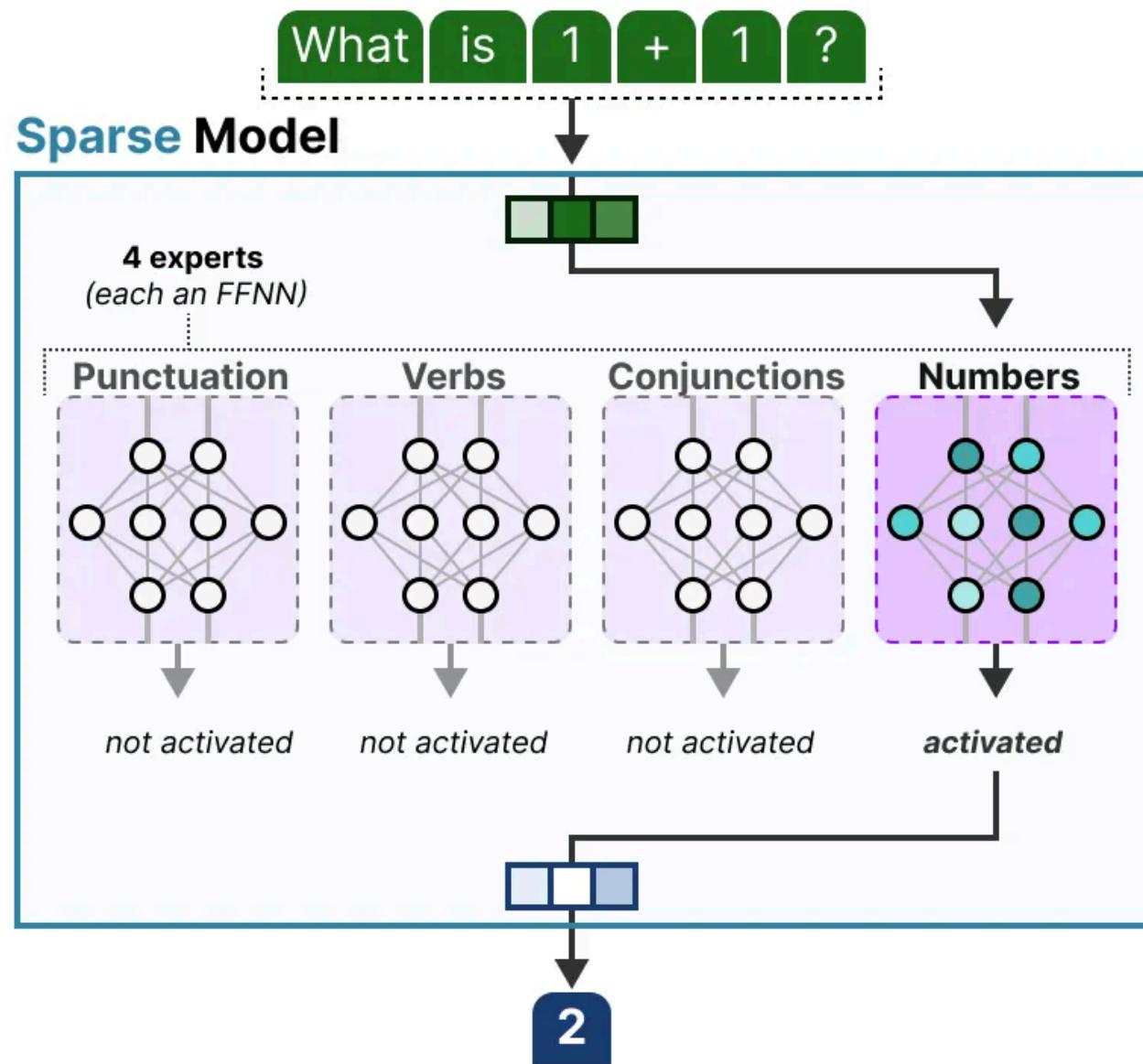
```
class MoeLayer(nn.Module):
    def __init__(self, experts: List[nn.Module],
                 super().__init__()
    assert len(experts) > 0
    self.experts = nn.ModuleList(experts)
    self.gate = gate
    self.args = moe_args
```

This visual also demonstrates that experts tend to focus on syntax rather than a specific domain.

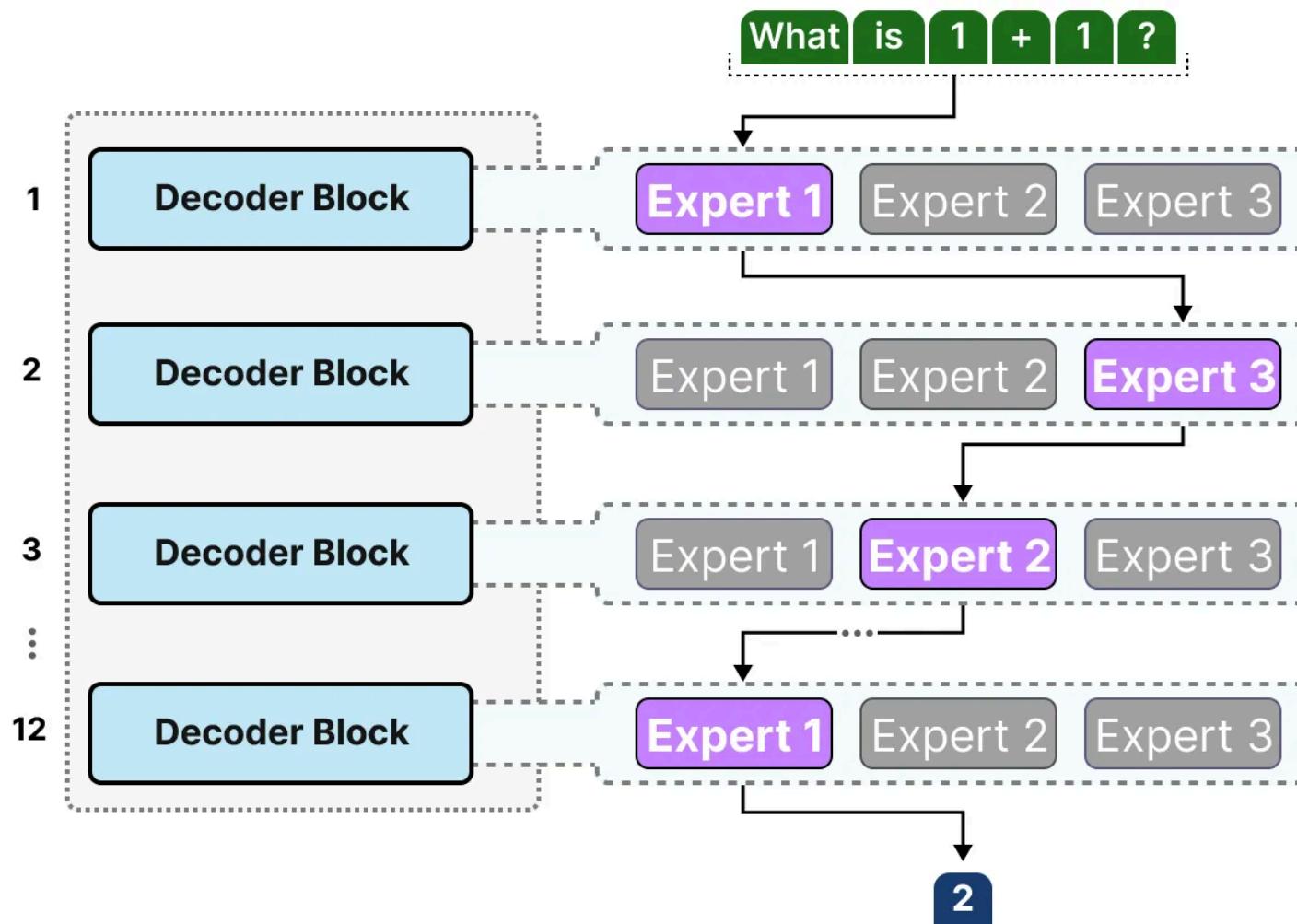
Thus, although decoder experts do not seem to have a specialism they do seem to be used consistently for certain types of tokens.

## The Architecture of Experts

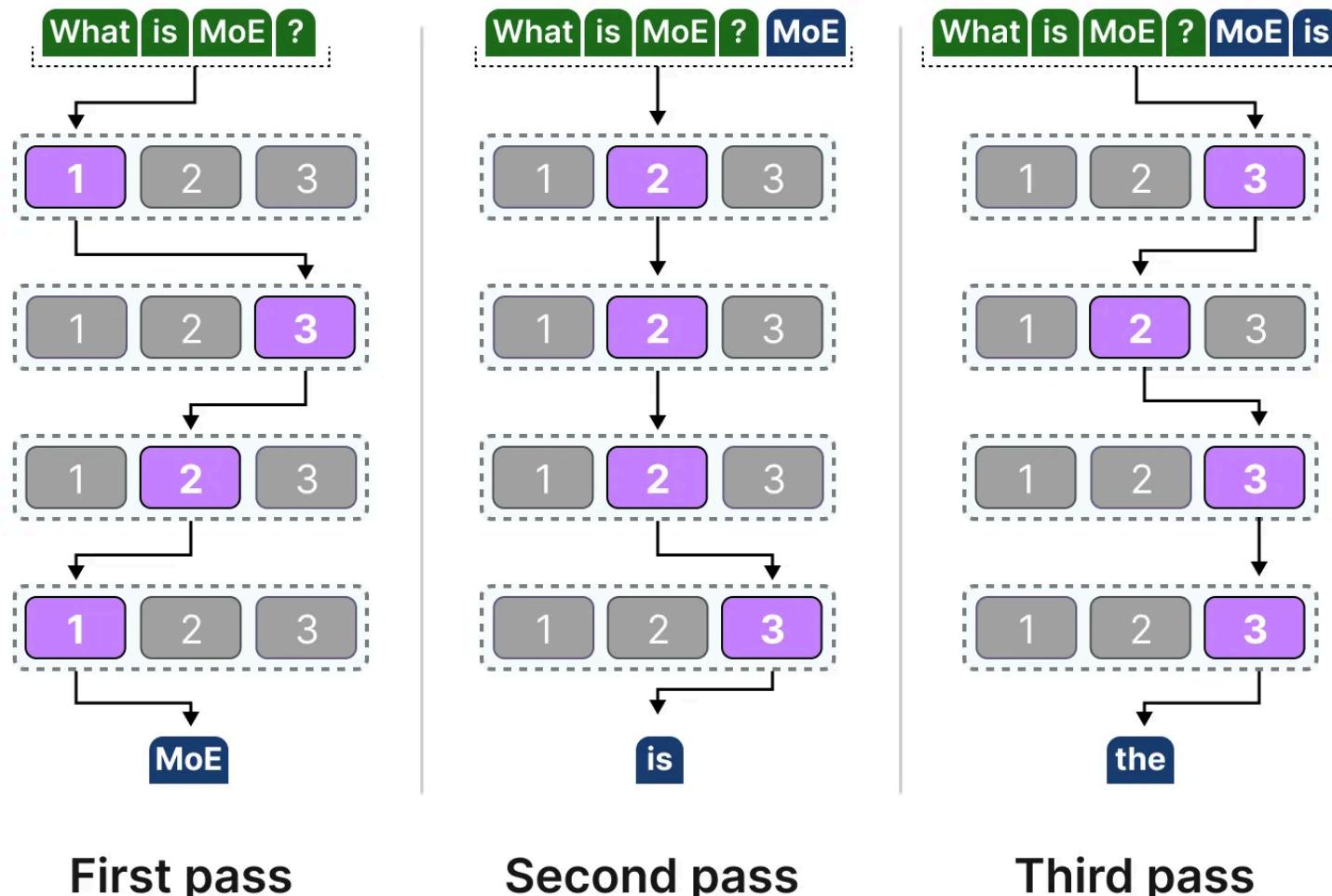
Although it's nice to visualize experts as a hidden layer of a dense model cut in pieces, they are often whole FFNNs themselves:



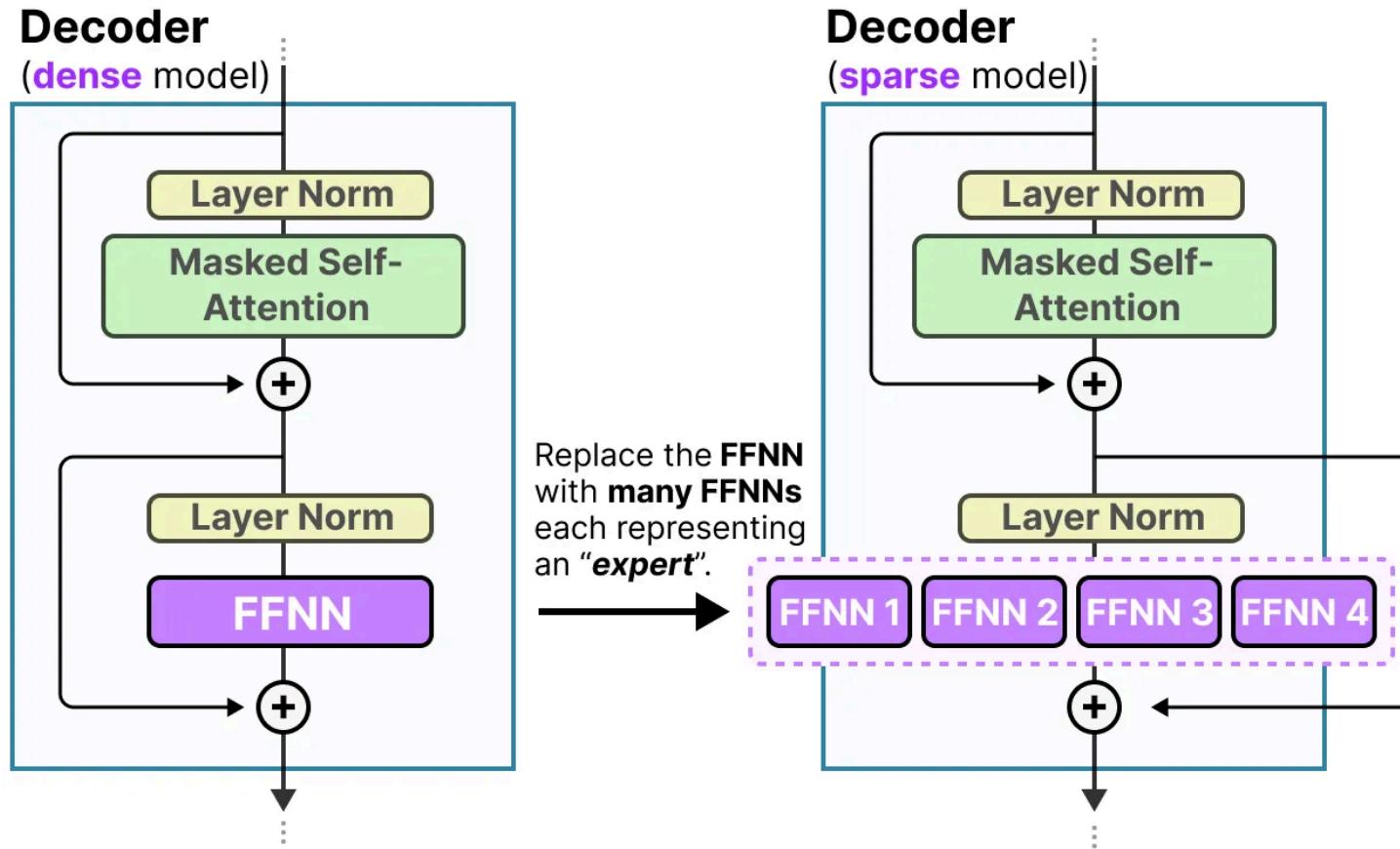
Since most LLMs have several decoder blocks, a given text will pass through multiple experts before the text is generated:



The chosen experts likely differ between tokens which results in different “paths” being taken:



If we update our visualization of the decoder block, it would now contain more FFNNs (one for each expert) instead:



The decoder block now has multiple FFNNs (each an “expert”) that it can use during inference.

# The Routing Mechanism

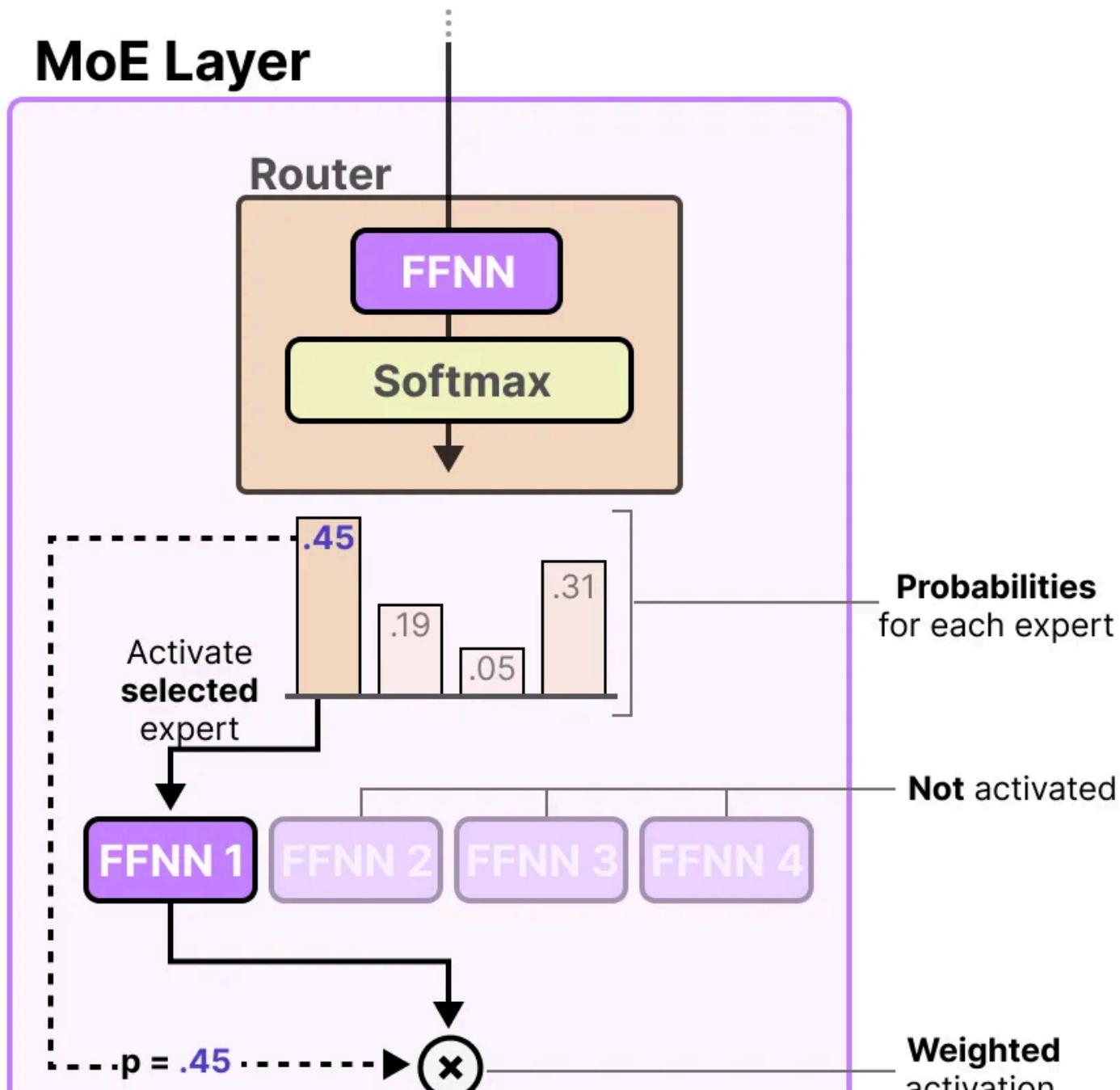
Now that we have a set of experts, how does the model know which experts to use?

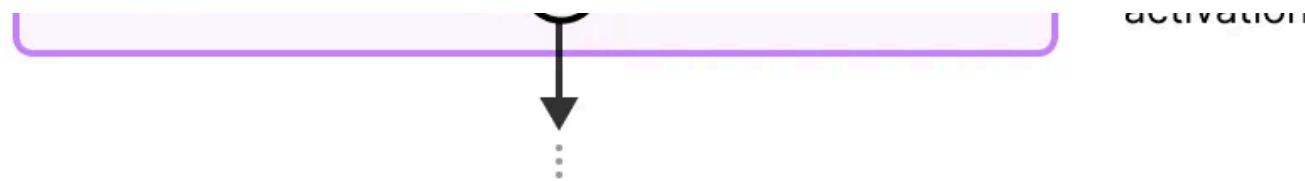
Just before the experts, a **router** (also called a **gate network**) is added which is trained to choose which expert to choose for a given token.

## The Router

The **router** (or **gate network**) is also an FFNN and is used to choose the expert based on a particular input. It outputs probabilities which it uses to select the best matching expert:

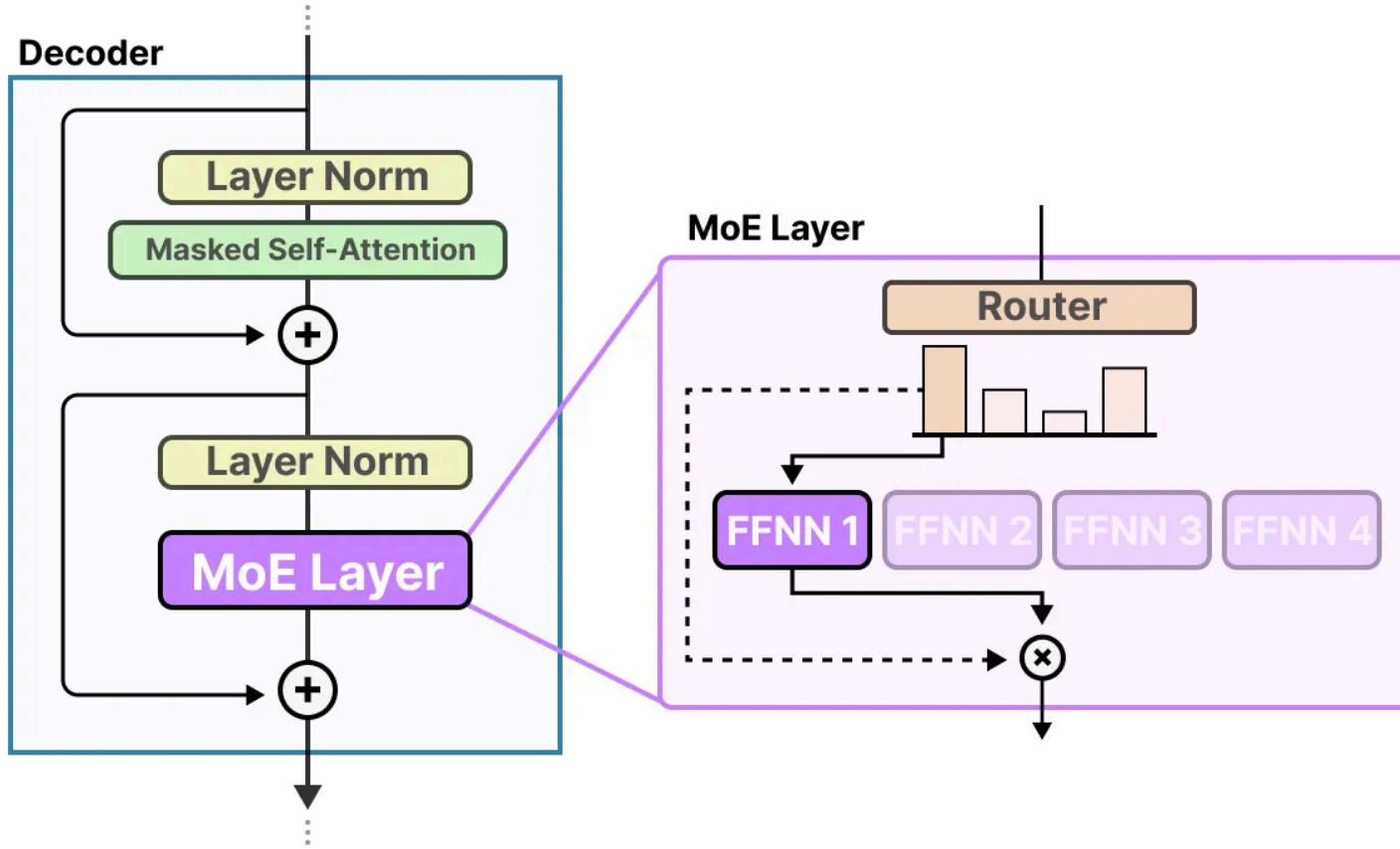
# MoE Layer





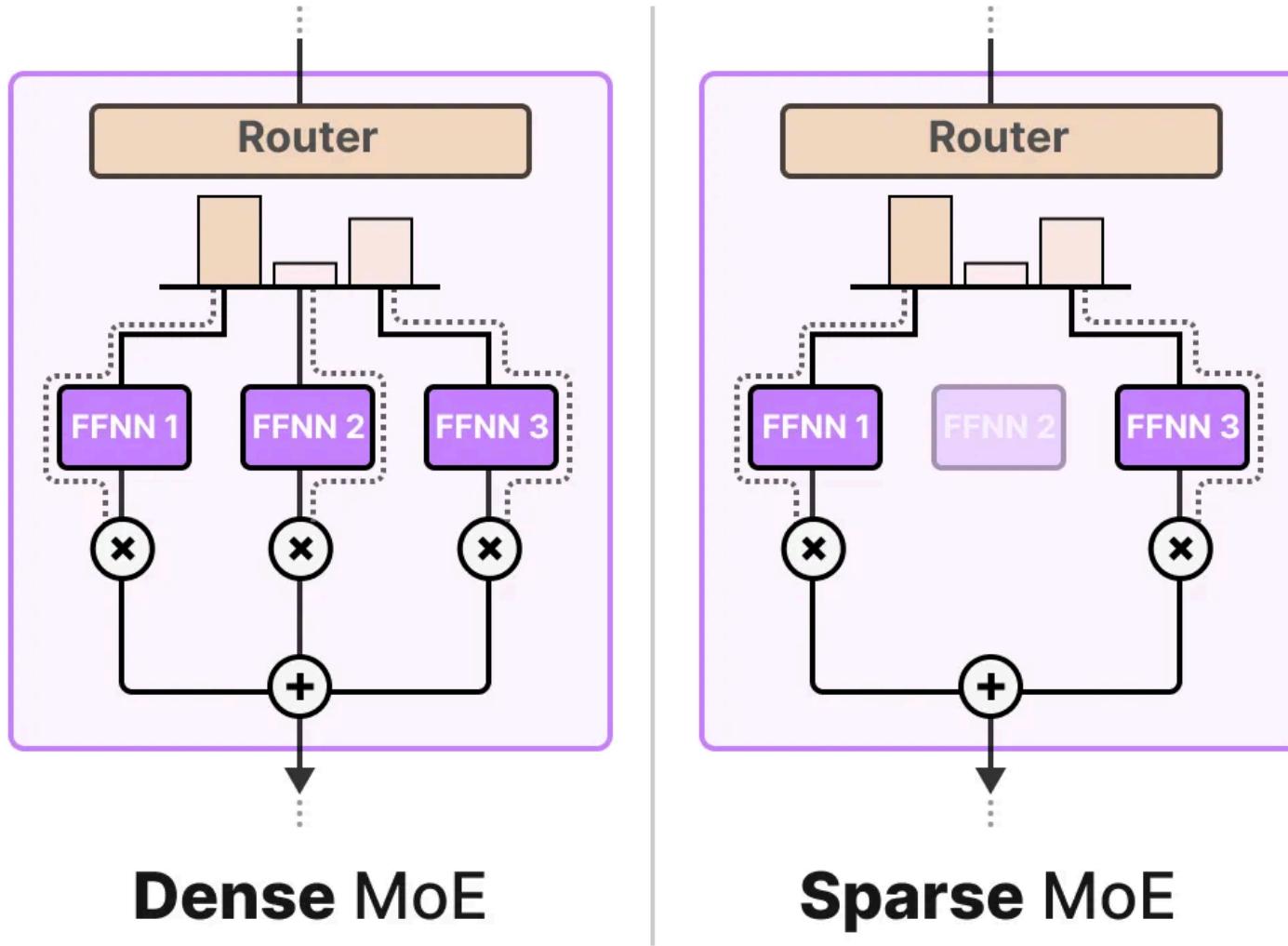
The expert layer returns the output of the selected expert multiplied by the gate value (selection probabilities).

The router together with the experts (of which only a few are selected) makes up the **MoE Layer**:



A given MoE layer comes in two sizes, either a *sparse* or a *dense* mixture of experts.

Both use a router to select experts but a Sparse MoE only selects a few whereas a Dense MoE selects them all but potentially in different distributions.



For instance, given a set of tokens, a MoE will distribute the tokens across all experts whereas a Sparse MoE will only select a few experts.

In the current state of LLMs, when you see a “MoE” it will typically be a Sparse MoE as it allows you to use a subset of experts. This is computationally cheaper which is an important trait for LLMs.

## Selection of Experts

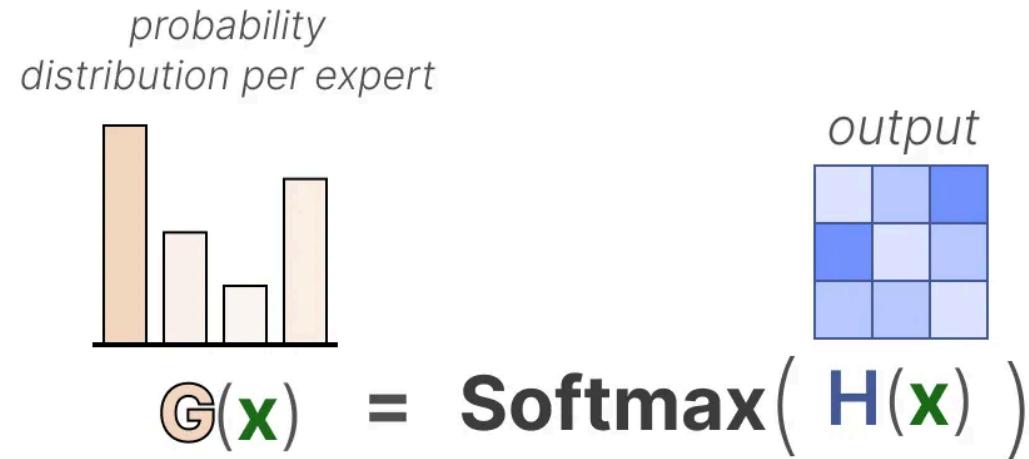
The gating network is arguably the most important component of any MoE as it not only decides which experts to choose during *inference* but also *training*.

In its most basic form, we multiply the input ( $\mathbf{x}$ ) by the router weight matrix ( $\mathbf{W}$ ):

$$\begin{array}{ccc}
 & \text{router weights} & \\
 \text{output} & & \begin{matrix} \text{orange} & \text{light orange} & \text{orange} \\ \text{white} & \text{orange} & \text{orange} \\ \text{orange} & \text{white} & \text{orange} \\ \text{orange} & \text{orange} & \text{white} \\ \text{orange} & \text{white} & \text{orange} \end{matrix} \\
 & \text{input} & \\
 \mathbf{H}(\mathbf{x}) & = & \mathbf{x} * \mathbf{W}
 \end{array}$$

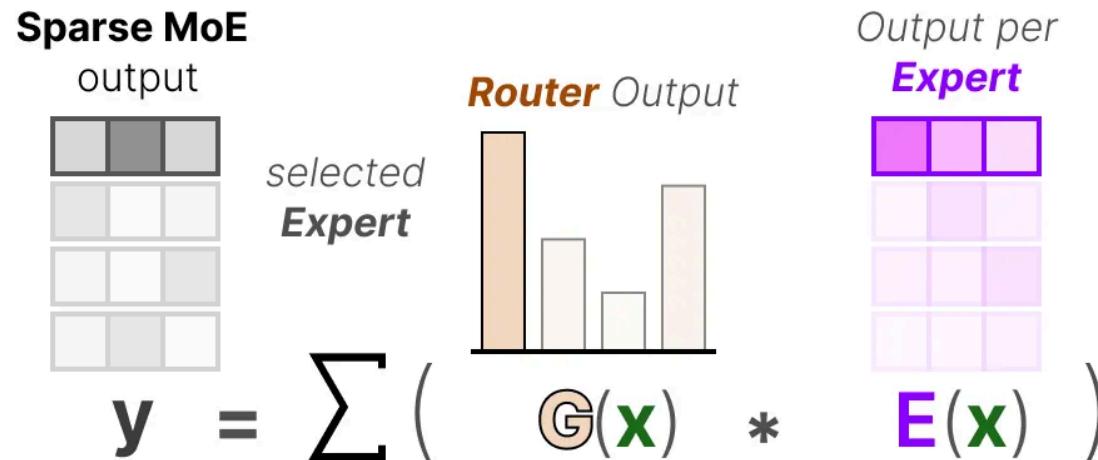
The diagram illustrates the computation of the output matrix  $\mathbf{H}(\mathbf{x})$  from the input matrix  $\mathbf{x}$  and the router weight matrix  $\mathbf{W}$ . The input matrix  $\mathbf{x}$  is a 3x3 grid of green squares. The router weight matrix  $\mathbf{W}$  is a 5x3 grid of orange and white squares. The output matrix  $\mathbf{H}(\mathbf{x})$  is a 3x3 grid of blue squares, where each square's color intensity corresponds to the weighted sum of the input row and the corresponding column in the weight matrix  $\mathbf{W}$ .

Then, we apply a **SoftMax** on the output to create a probability distribution  $G(\mathbf{x})$  per expert:



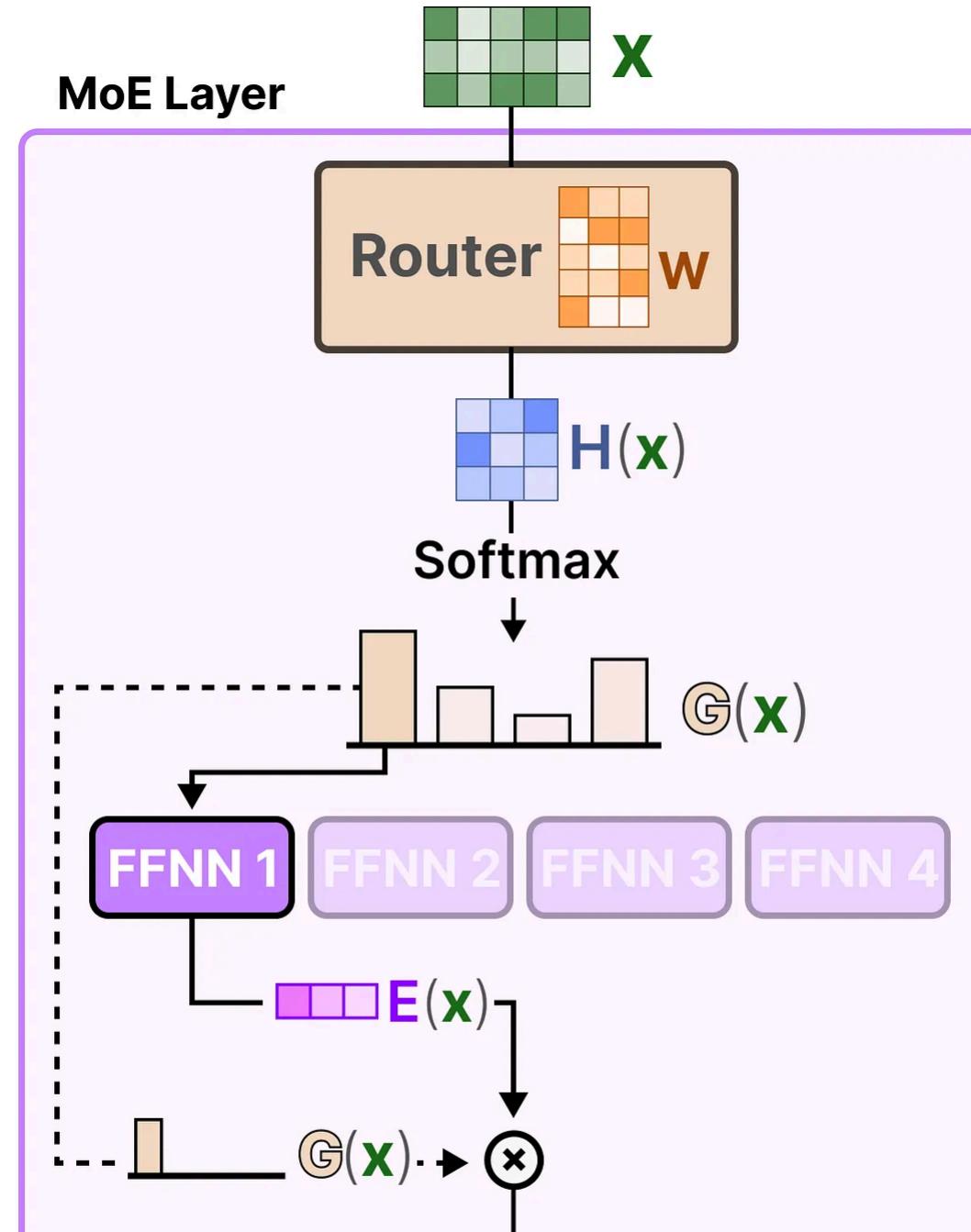
The router uses this probability distribution to choose the best matching expert for a given input.

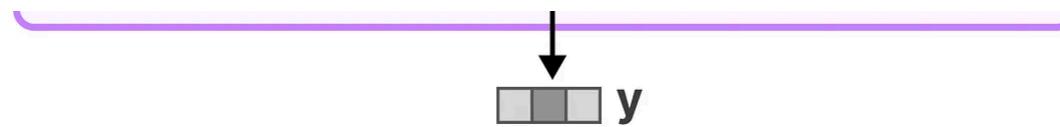
Finally, we multiply the output of each router with each selected expert and sum the results.

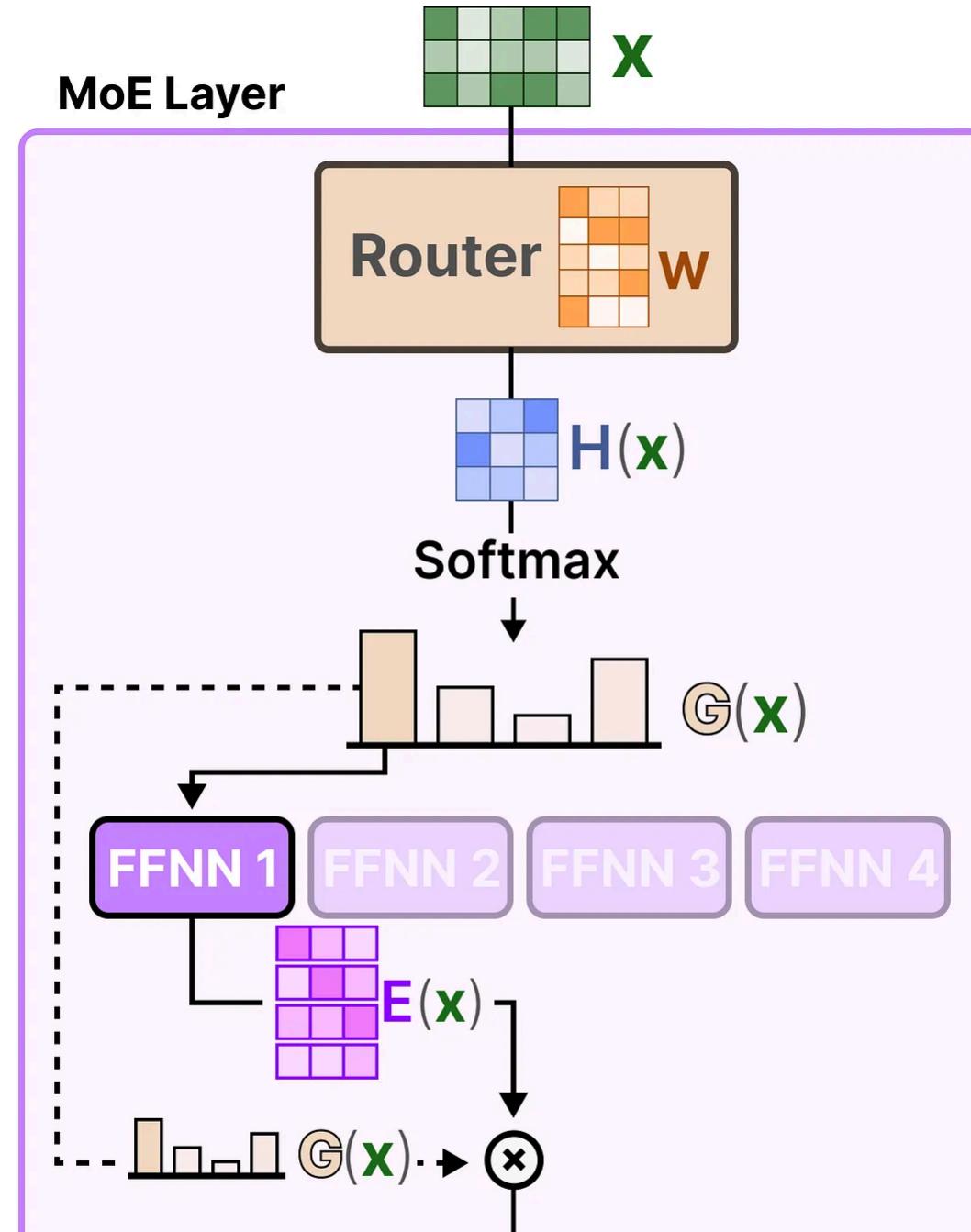


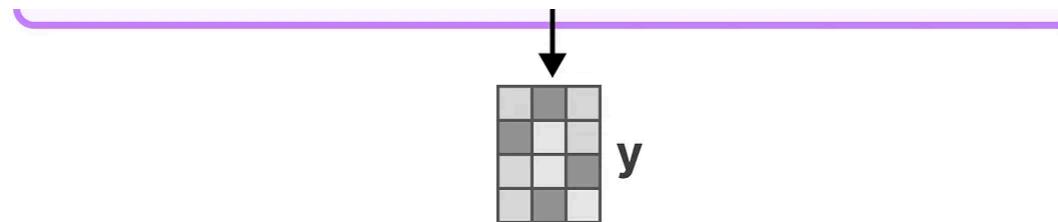
(only **1** expert is chosen in this example)

Let's put everything together and explore how the input flows through the router and experts:



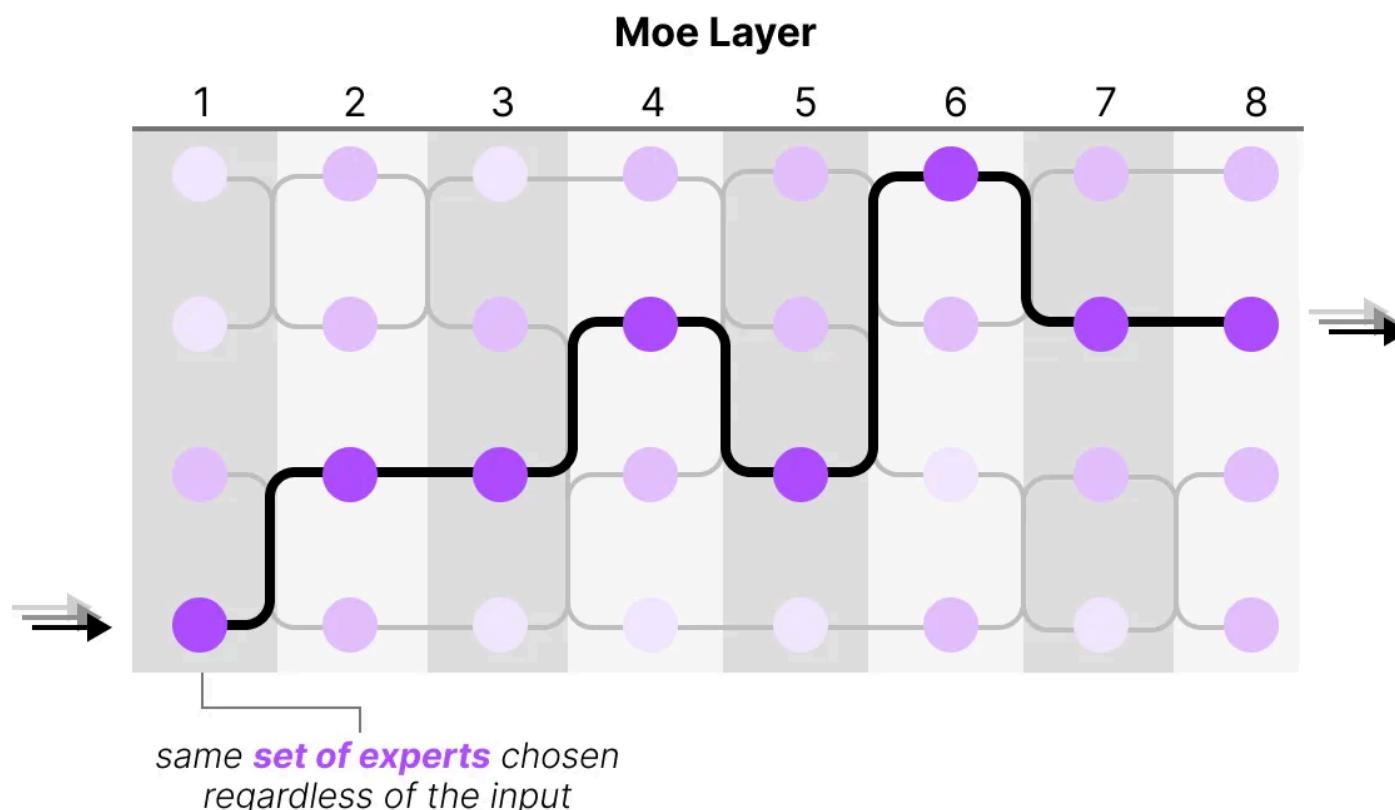






## The Complexity of Routing

However, this simple function often results in the router choosing the same expert since certain experts might learn faster than others:



Not only will there be an uneven distribution of experts chosen, but some experts will hardly be trained at all. This results in issues during both training and inference.

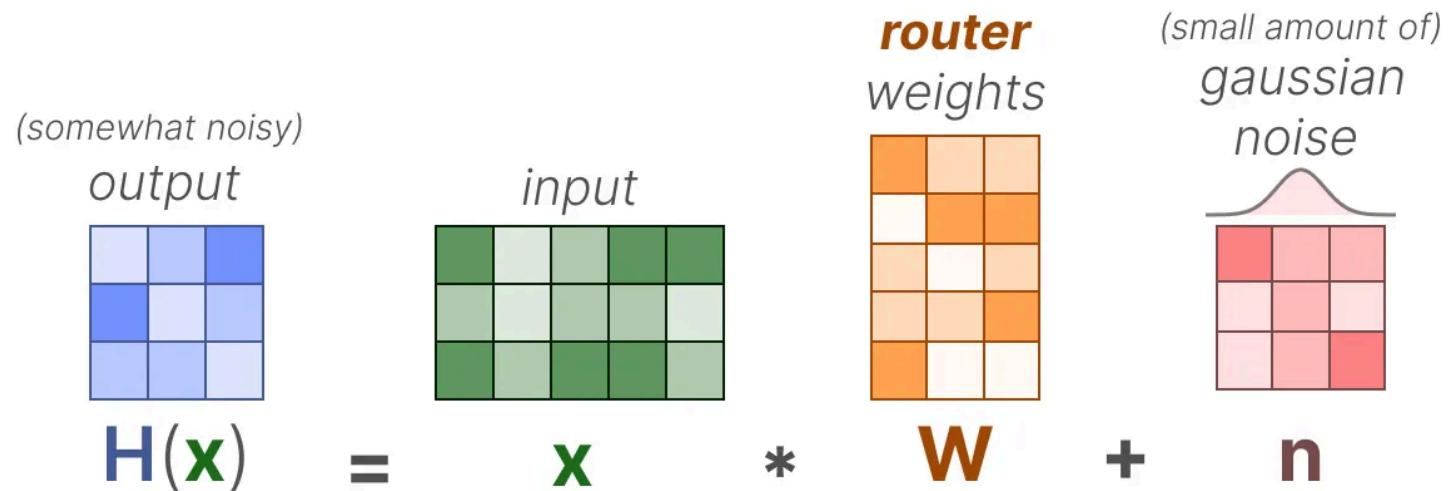
Instead, we want equal importance among experts during training and inference, which we call **load balancing**. In a way, it's to prevent overfitting on the same experts.

## Load Balancing

To balance the importance of experts, we will need to look at the router as it is the main component to decide which experts to choose at a given time.

### KeepTopK

One method of load balancing the router is through a straightforward extension called [KeepTopK](#)<sup>2</sup>. By introducing trainable (gaussian) noise, we can prevent the same experts from always being picked:



Then, all but the top k experts that you want activating (for example 2) will have their weights set to  $-\infty$ :

(updated)  
*output*

-∞	-∞	-∞

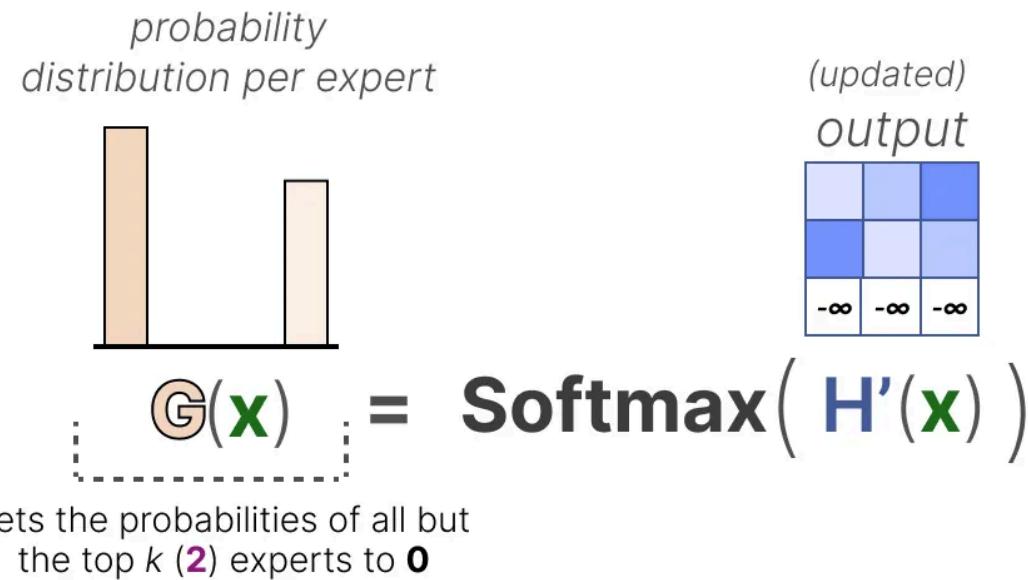
(somewhat noisy)  
*output*

-∞		

$$H'(x) = \text{KeepTopK} ( H(x), 2 )$$

sets the weights of all but  
the top  $k$  (2) experts to  $-\infty$

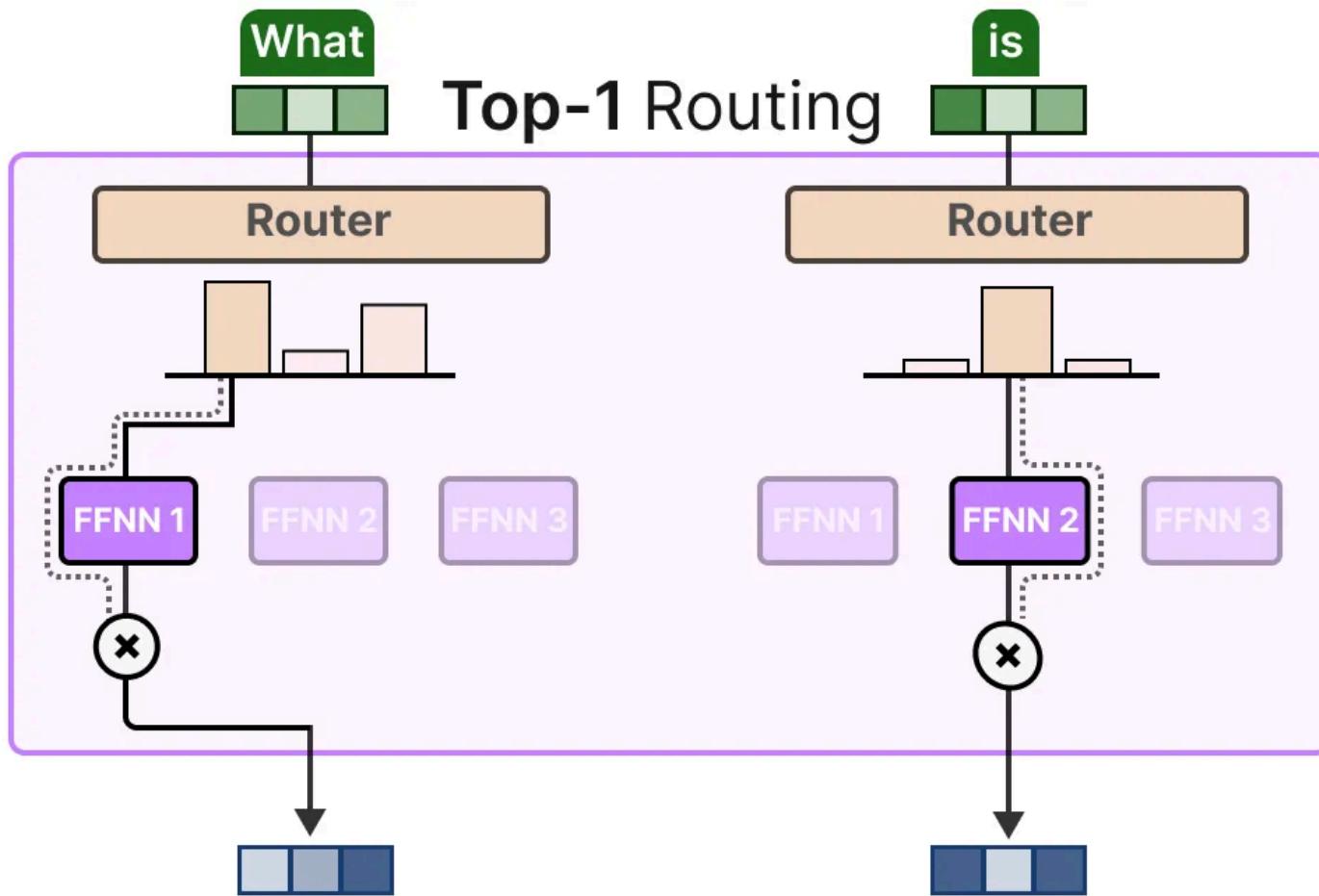
By setting these weights to  $-\infty$ , the output of the SoftMax on these weights will result in a probability of 0:



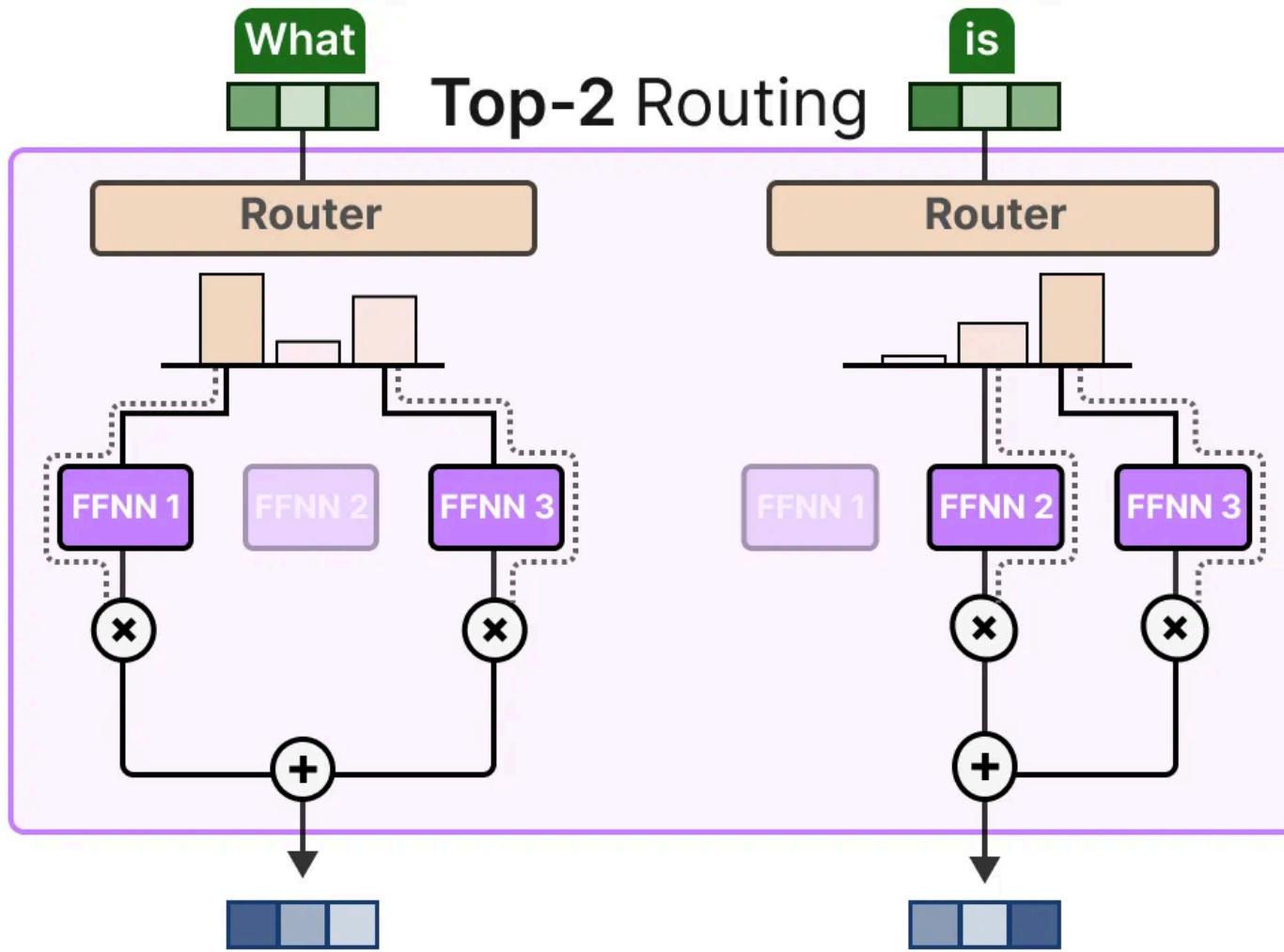
The KeepTopK strategy is one that many LLMs still use despite many promising alternatives. Note that KeepTopK can also be used without the additional noise.

## Token Choice

The KeepTopK strategy routes each token to a few selected experts. This method is called *Token Choice*<sup>3</sup> and allows for a given token to be sent to one expert (*top-1 routing*):



or to more than one expert (top-k routing):



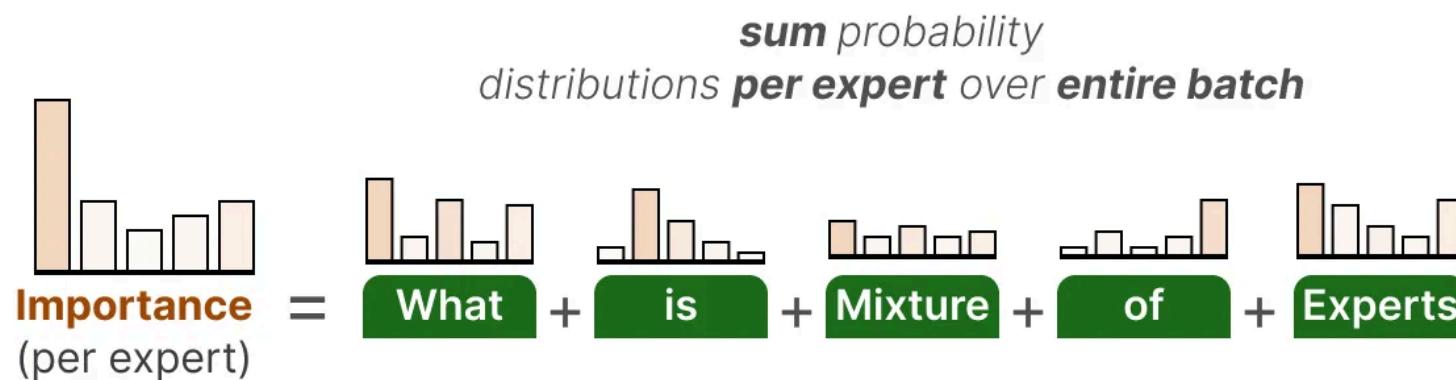
A major benefit is that it allows the experts' respective contributions to be weighed and integrated.

## Auxiliary Loss

To get a more even distribution of experts during training, the auxiliary loss (also called *load balancing loss*) was added to the network's regular loss.

It adds a constraint that forces experts to have equal importance.

The first component of this auxiliary loss is to sum the router values for each expert over the entire batch:

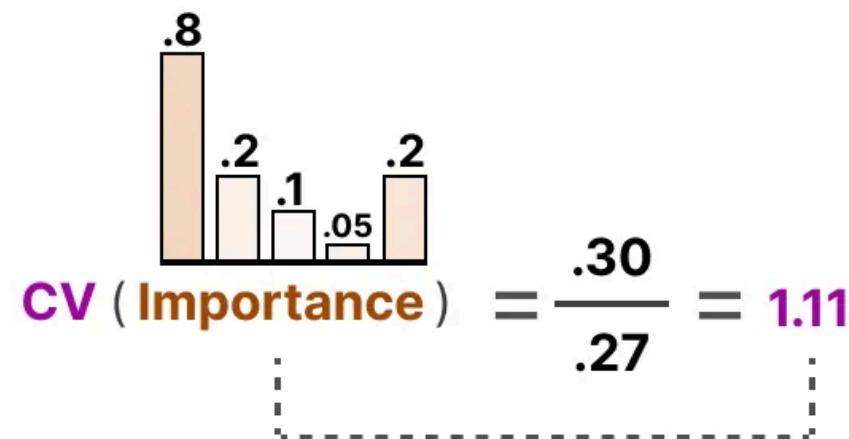


This gives us the *importance scores* per expert which represents how likely a given expert will be chosen regardless of the input.

We can use this to calculate the *coefficient variation (CV)*, which tells us how different the importance scores are between experts.

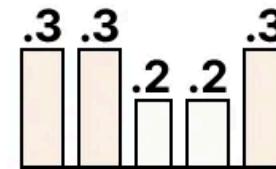
$$\text{Coefficient Variation (CV)} = \frac{\text{standard deviation } (\sigma)}{\text{mean } (\mu)}$$

For instance, if there are a lot of differences in importance scores, the CV will be high:



**High** variance in expert  
**importance** result in **high** CV

In contrast, if all experts have similar importance scores, the CV will be low (which is what we aim for):



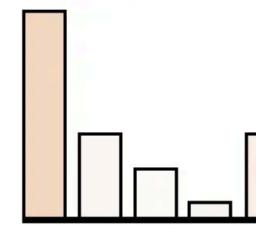
$$CV(\text{Importance}) = \frac{.05}{.26} = .19$$

**Low** variance in expert **importance** result in **low CV**

Using this CV score, we can update the auxiliary loss during training such that it aims to lower the CV score as much as possible (*thereby giving equal importance to each expert*):

$$\text{Auxiliary Loss} = \underbrace{w_{\text{importance}}}_{\substack{\text{(constant)} \\ \text{scaling factor}}} * \text{CV}(\text{Importance})^2$$

A high variance in expert importance (CV) results in high loss and vice versa

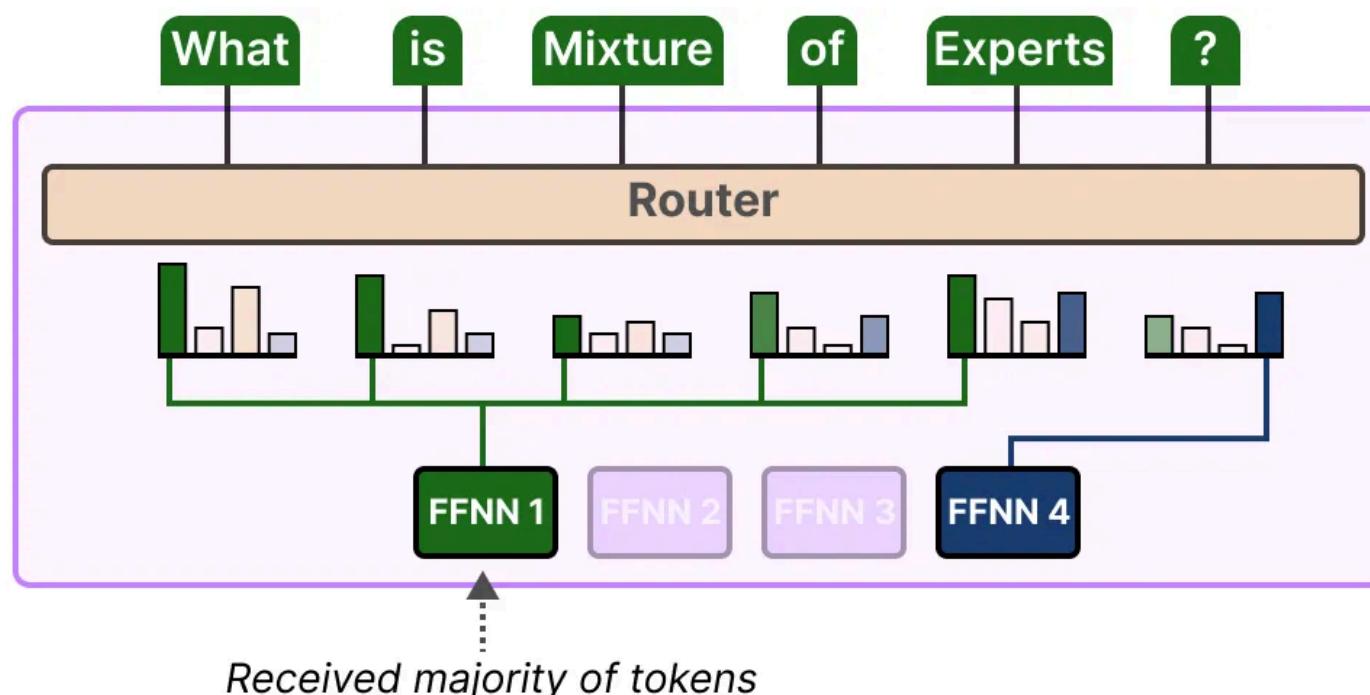


Finally, the auxiliary loss is added as a separate loss to optimize during training.

## Expert Capacity

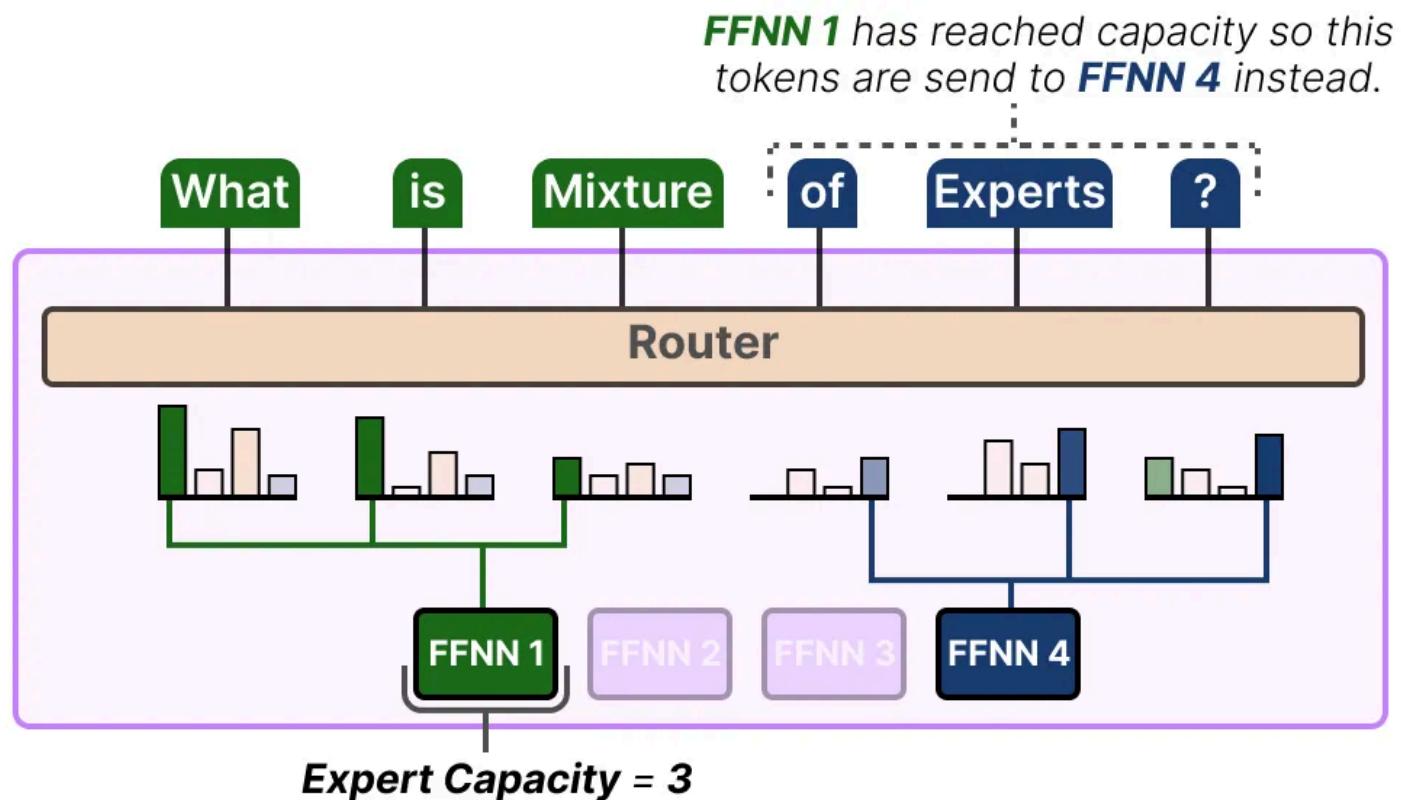
Imbalance is not just found in the experts that were chosen but also in the distributions of tokens that are sent to the expert.

For instance, if input tokens are disproportionately sent to one expert over another then that might also result in undertraining:

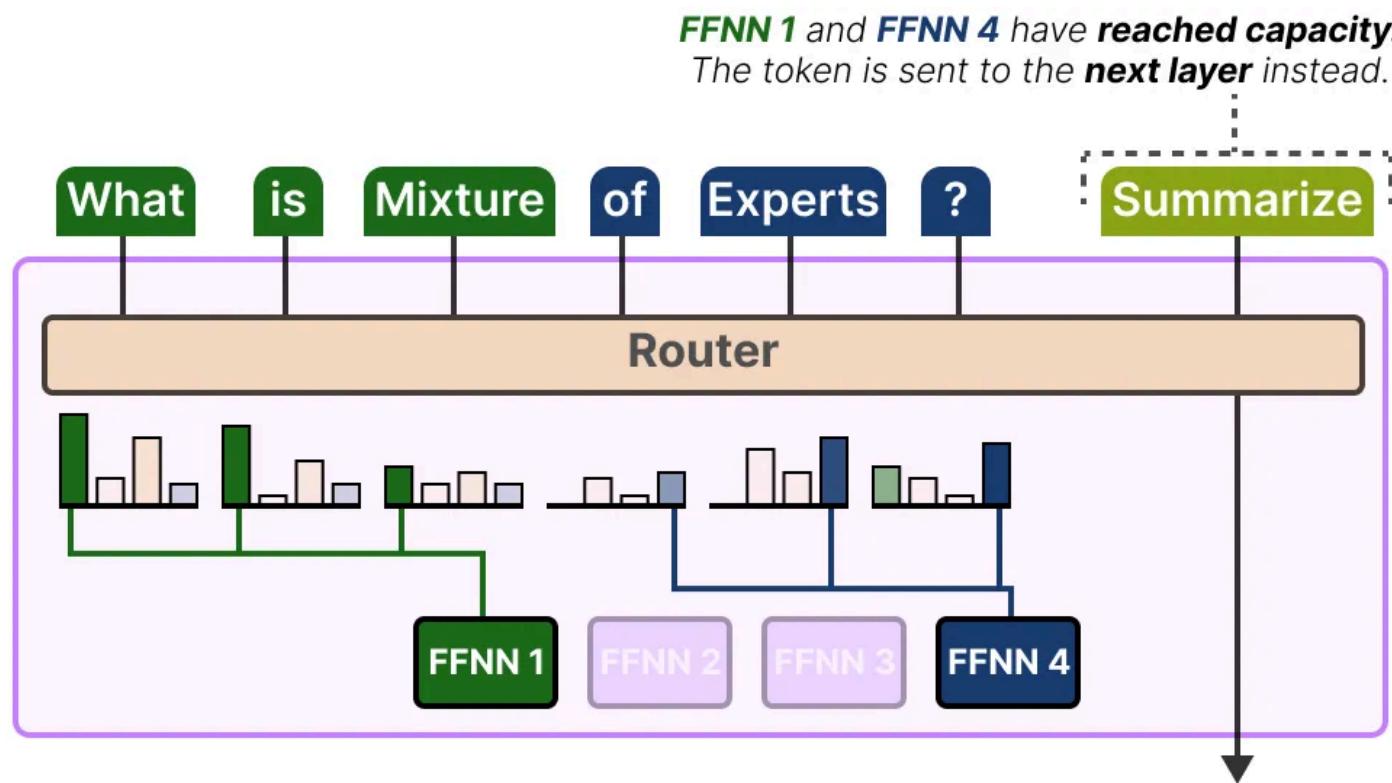


Here, it is not just about which experts are used but **how much** they are used.

A solution to this problem is to limit the amount of tokens a given expert can handle, namely *Expert Capacity*<sup>4</sup>. By the time an expert has reached capacity, the resulting tokens will be sent to the next expert:



If both experts have reached their capacity, the token will not be processed by any expert but instead sent to the next layer. This is referred to as *token overflow*.

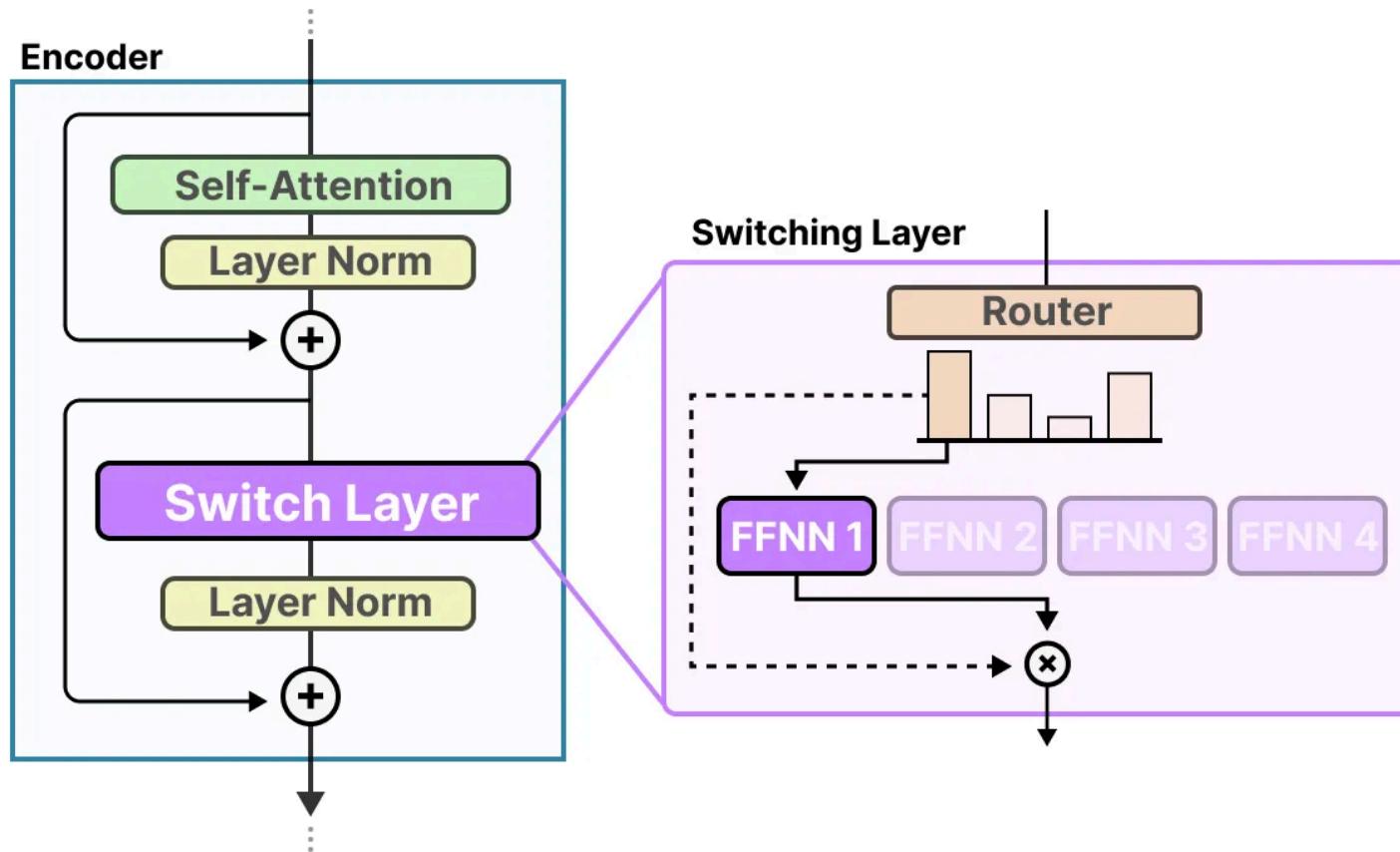


## Simplifying MoE with the Switch Transformer

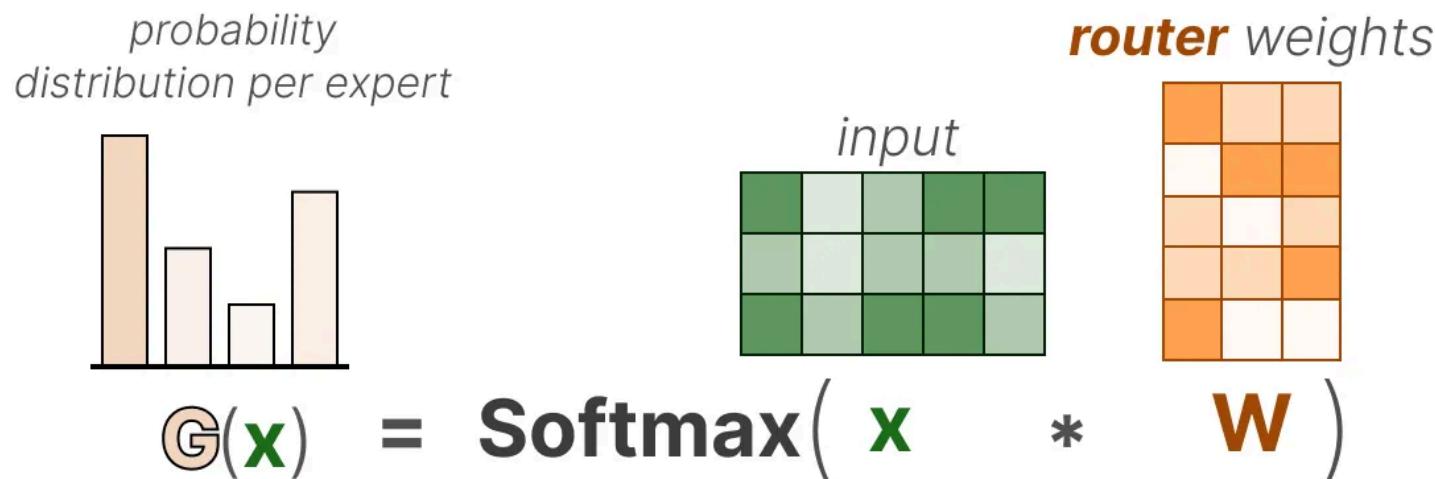
One of the first transformer-based MoE models that dealt with the training instability issues of MoE (such as load balancing) is the Switch Transformer.<sup>5</sup> It simplifies much of the architecture and training procedure while increasing training stability.

# The Switching Layer

The Switch Transformer is a T5 model (encoder-decoder) that replaces the traditional FFNN layer with a Switching Layer. The Switching Layer is a Sparse MoE layer that selects a single expert for each token (*Top-1 routing*).



The router does no special tricks for calculating which expert to choose and takes the softmax of the input multiplied by the expert's weights (same as we did previously).



This architecture (*top-1 routing*) assumes that only 1 expert is needed for the router to learn how to route the input. This is in contrast to what we have seen previously where we assumed that tokens should be routed to multiple experts (*top-k routing*) to learn the routing behavior.

## Capacity Factor

The capacity factor is an important value as it determines how many tokens an expert can process. The Switch Transformer extends upon this by introducing a **capacity factor** directly influencing the expert capacity.

$$\text{expert capacity} = \left( \frac{\text{tokens per batch}}{\text{number of experts}} \right) * \text{capacity factor}$$

The components of expert capacity are straightforward:

*tokens*    What    is    Mixture    of    Experts    ?

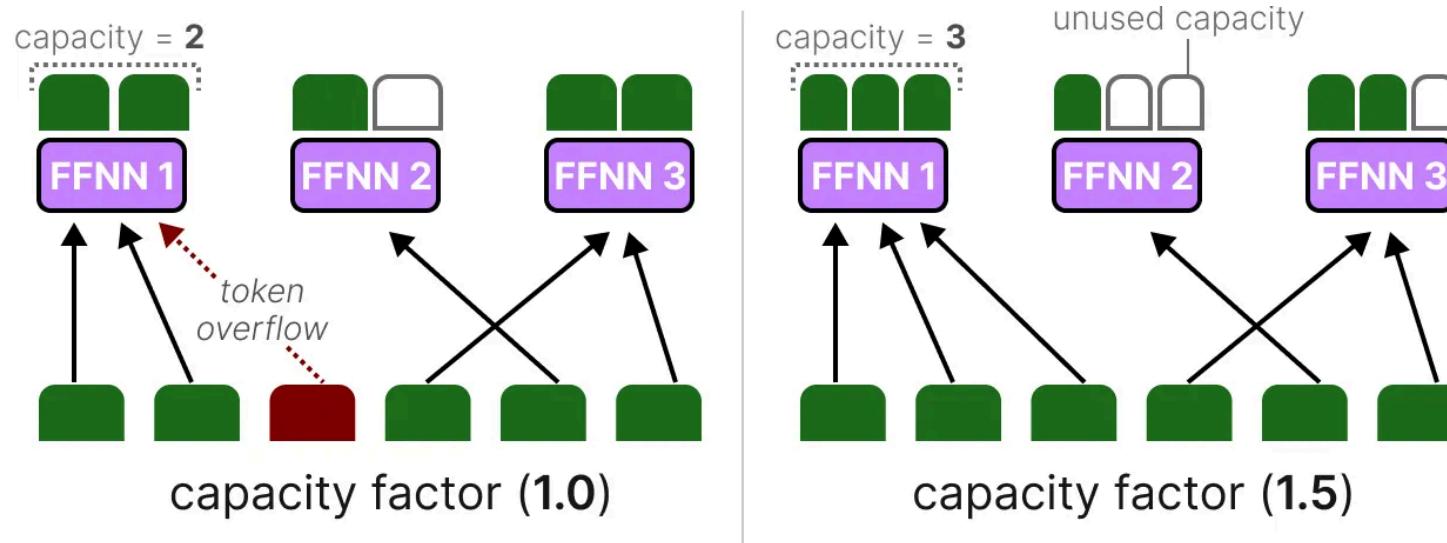
*experts*    FFNN 1    FFNN 2    FFNN 3

*capacity factor*    1.0

---

$$\text{expert capacity} = \left( \frac{6}{3} \right) * 1.0 = 2$$

If we increase the capacity factor each expert will be able to process more tokens.

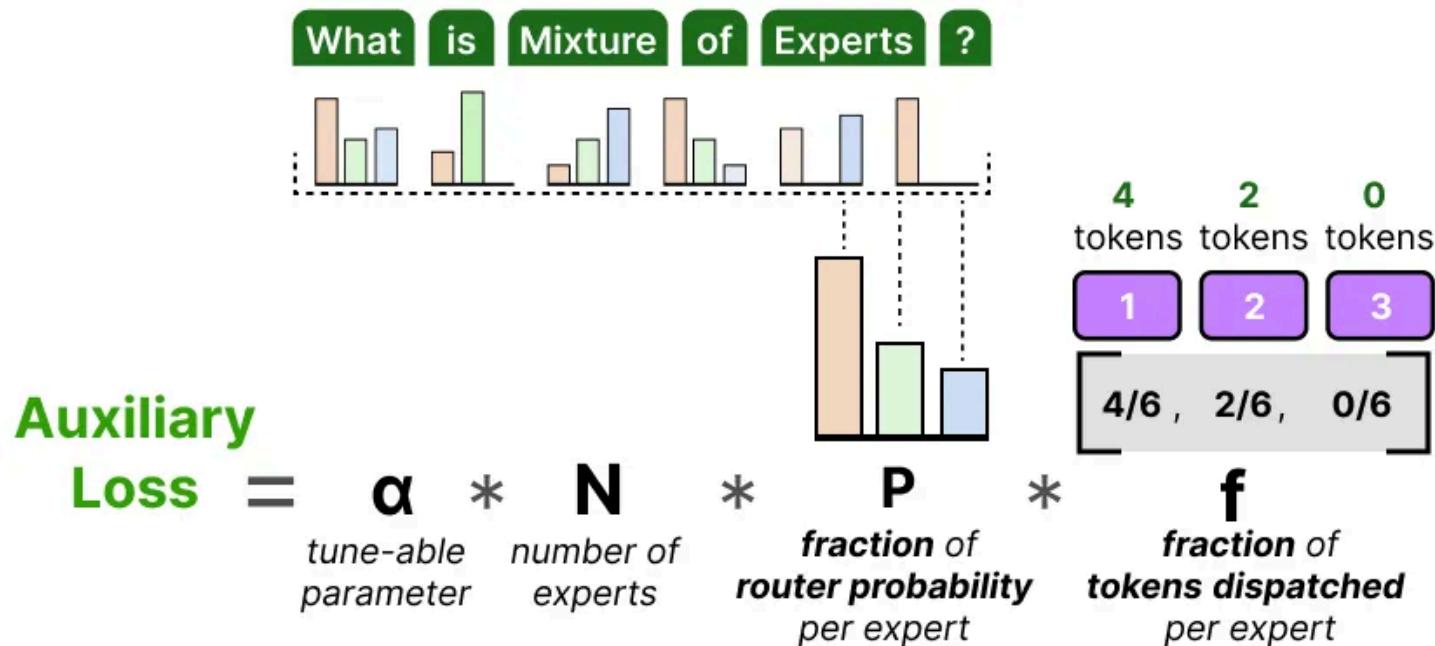


However, if the capacity factor is too large, we waste computing resources. In contrast, if the capacity factor is too small, the model performance will drop due to *token overflow*.

## Auxiliary Loss

To further prevent dropping tokens a simplified version of auxiliary loss was introduced.

Instead of calculating the coefficient variation, this simplified loss weighs the fraction of tokens dispatched against the fraction of router probability per expert:



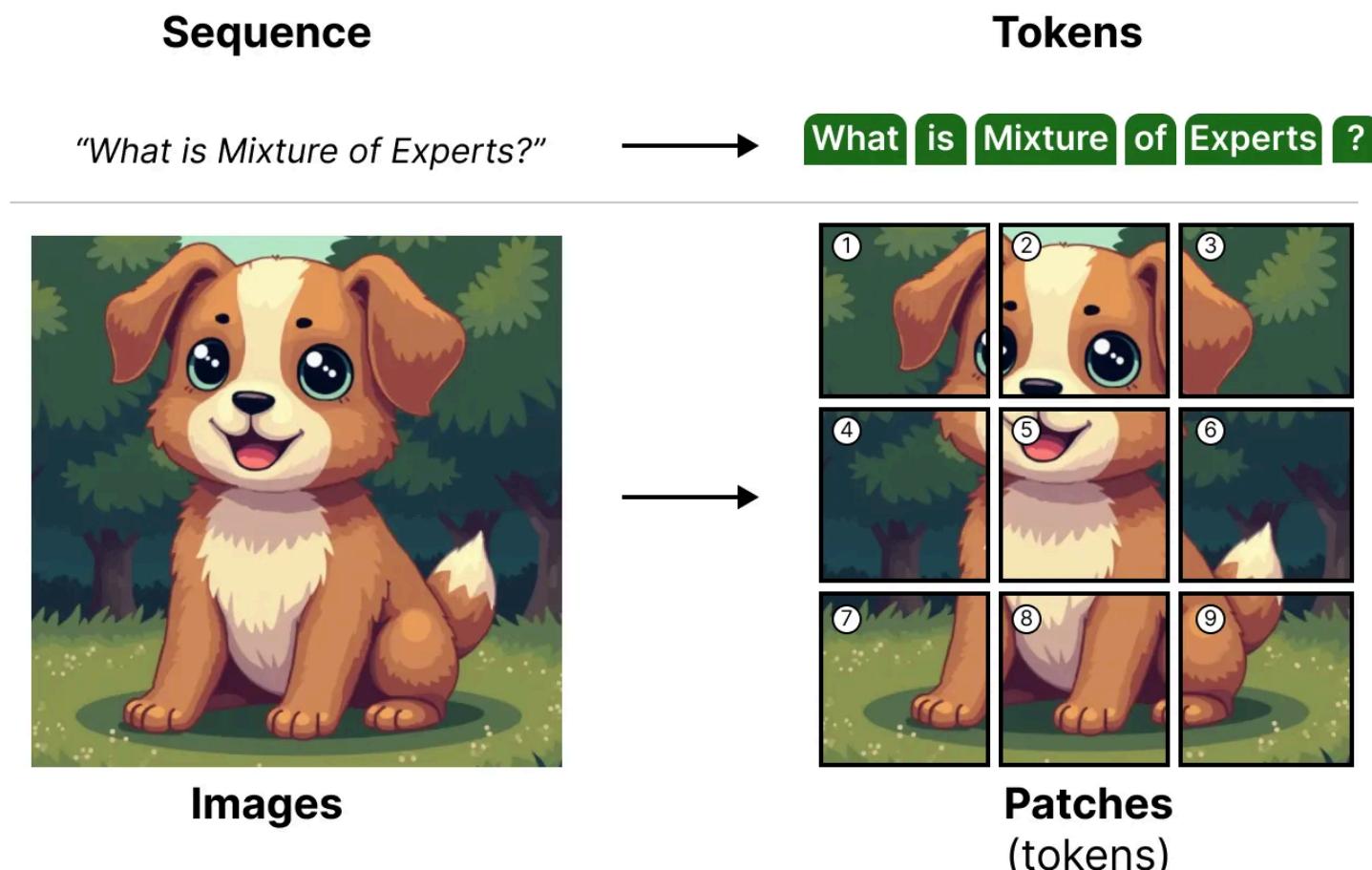
Since the goal is to get a uniform routing of tokens across the  $N$  experts, we want vectors  $P$  and  $f$  to have values of  $1/N$ .

$\alpha$  is a hyperparameter that we can use to fine-tune the importance of this loss during training. Too high values will overtake the primary loss function and too low values will do little for load balancing.

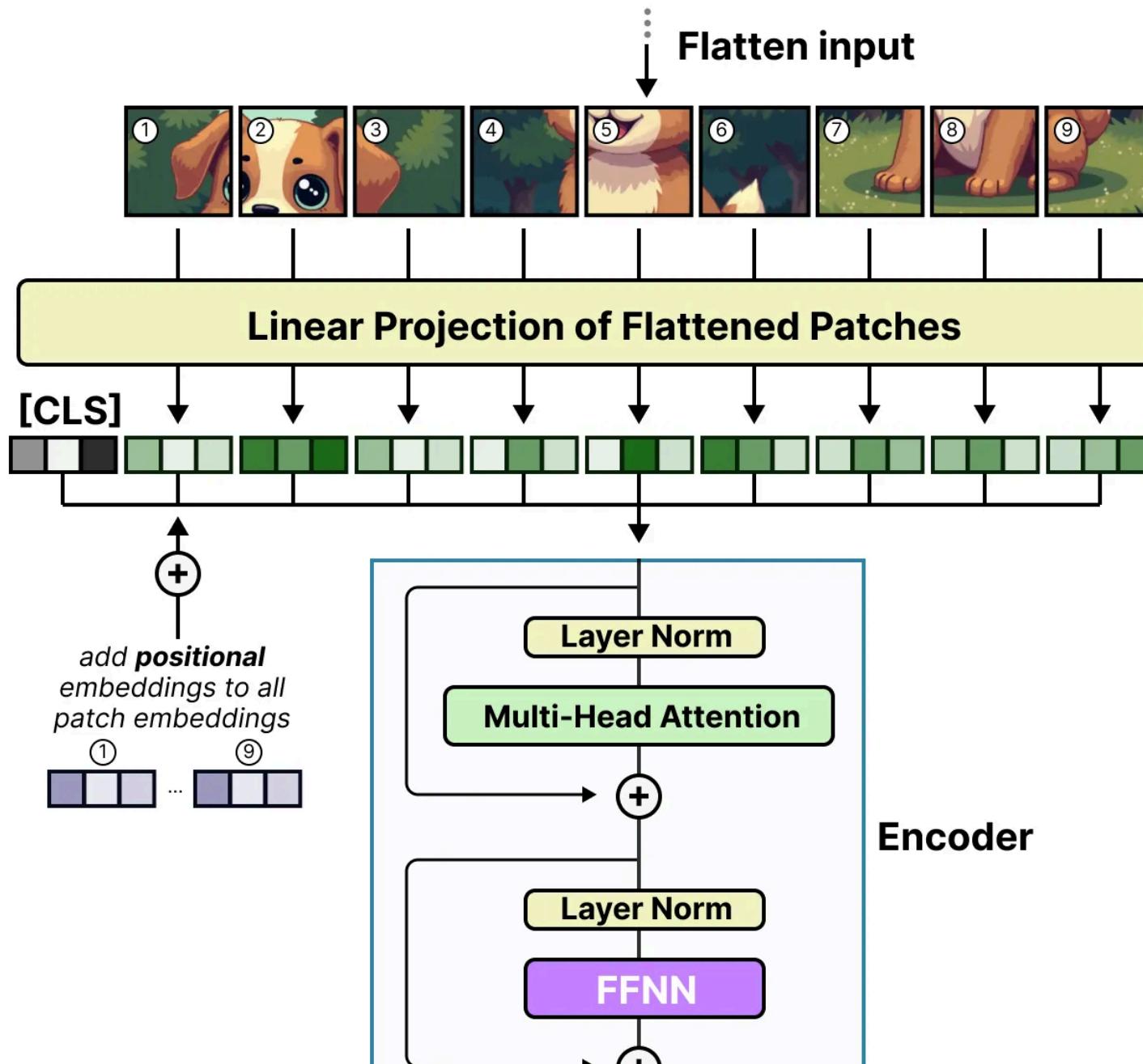
## Mixture of Experts in Vision Models

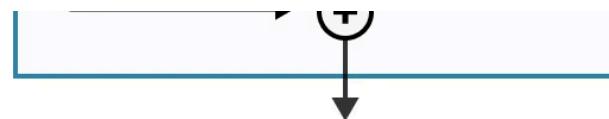
MoE is not a technique that is only available to language models. Vision models (such as ViT) leverage transformer-based architectures and therefore have the potential to use MoE.

As a quick recap, ViT (Vision-Transformer) is an architecture that splits images into patches that are processed similarly to tokens.<sup>6</sup>



These patches (or tokens) are then projected into embeddings (with additional positional embeddings) before being fed into a regular encoder:

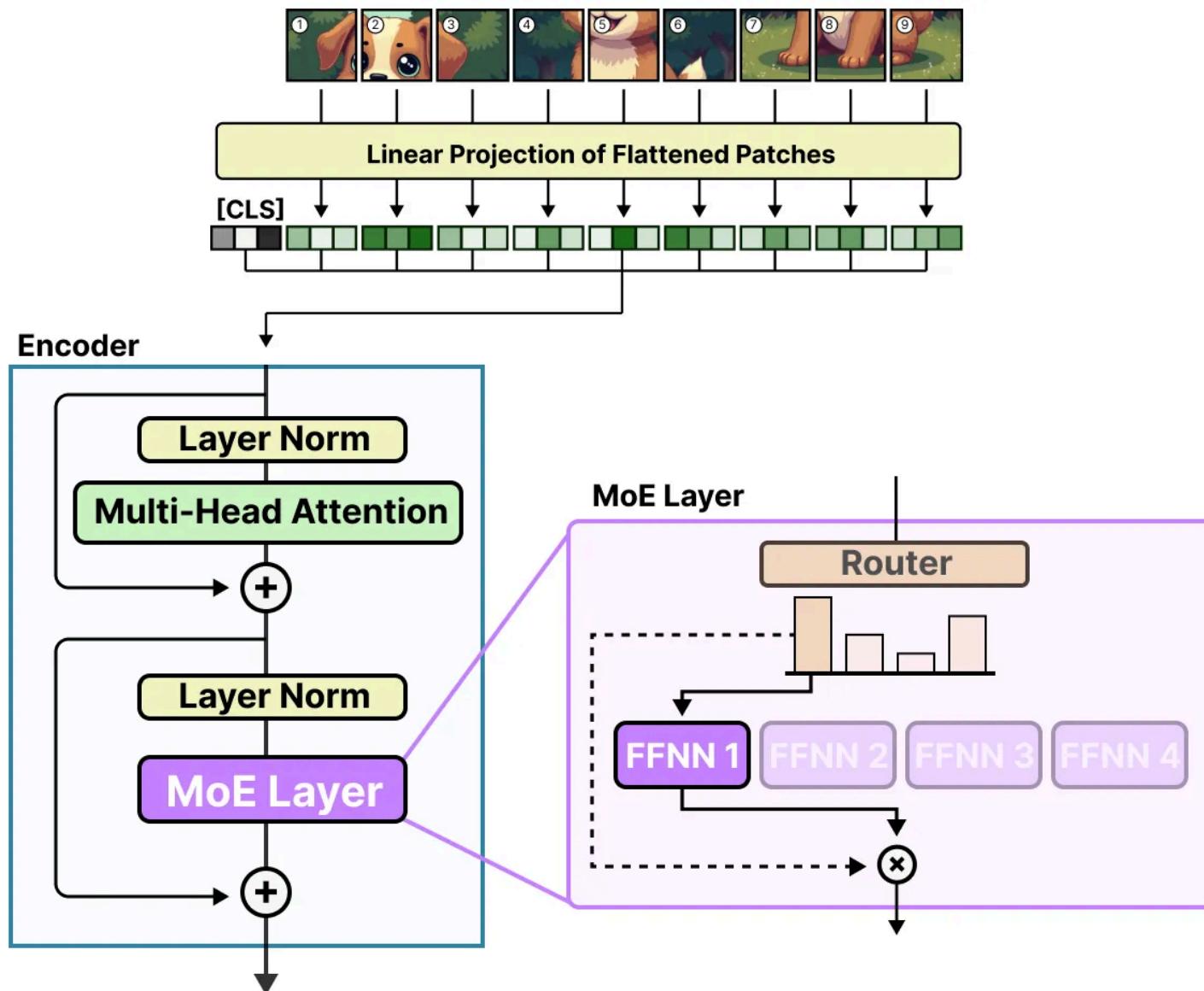




The moment these patches enter the encoder, they are processed like tokens which makes this architecture leverage itself well for MoE.

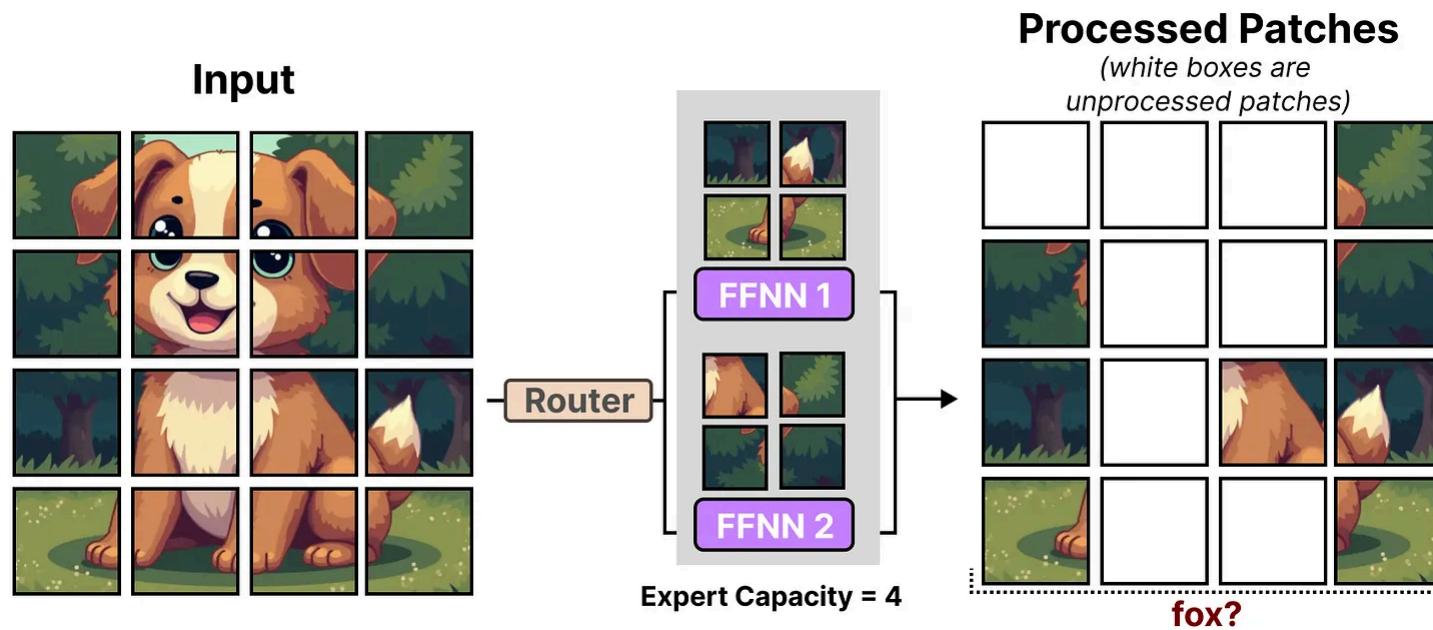
## Vision-MoE

Vision-MoE (V-MoE) is one of the first implementations of MoE in an image model.<sup>7</sup> It takes the ViT as we saw before and replaces the dense FFNN in the encoder with a Sparse MoE.

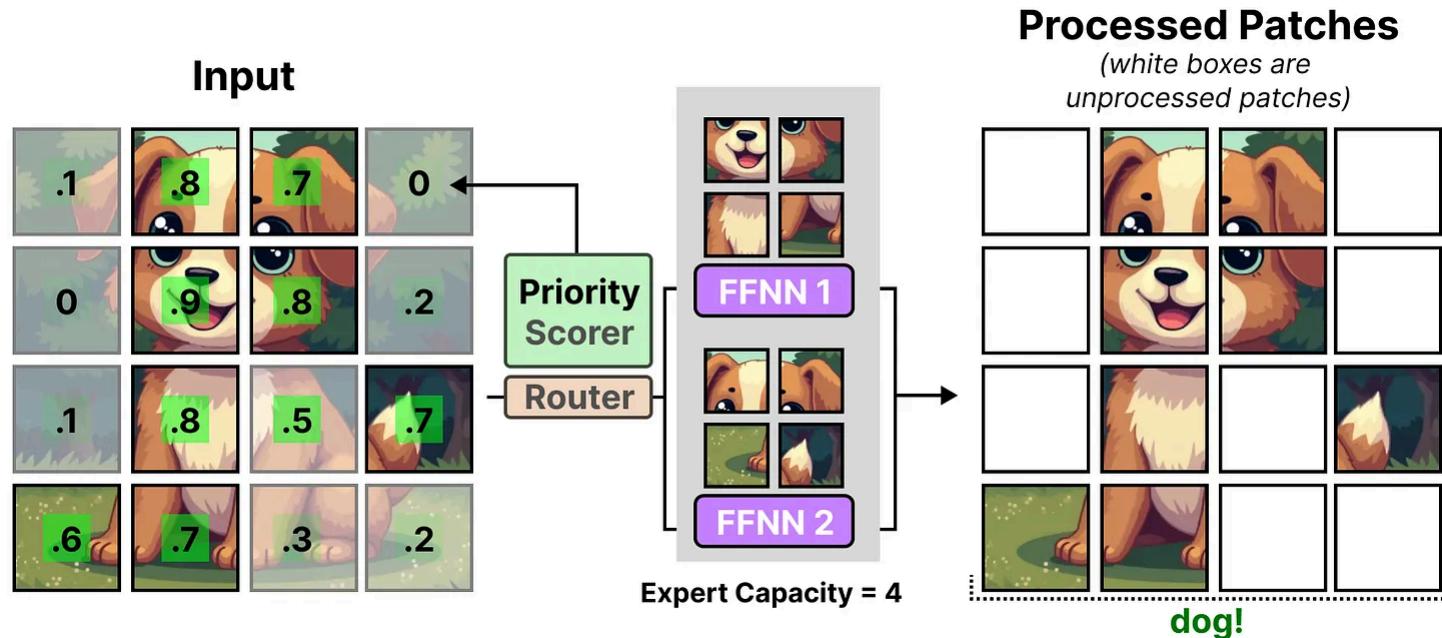


This allows ViT models, typically smaller in size than language models, to be massively scaled by adding experts.

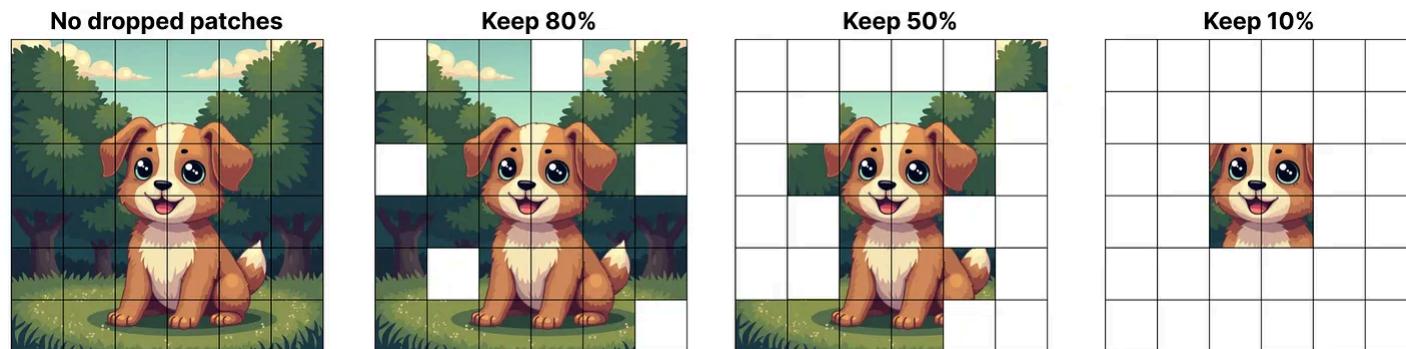
A small pre-defined expert capacity was used for each expert to reduce hardware constraints since images generally have many patches. However, a low capacity tends to lead to patches being dropped (akin to *token overflow*).



To keep the capacity low, the network assigns importance scores to patches and processes those first so that overflowed patches are generally less important. This is called Batch *Priority Routing*.



As a result, we should still see important patches routed if the percentage of tokens decreases.



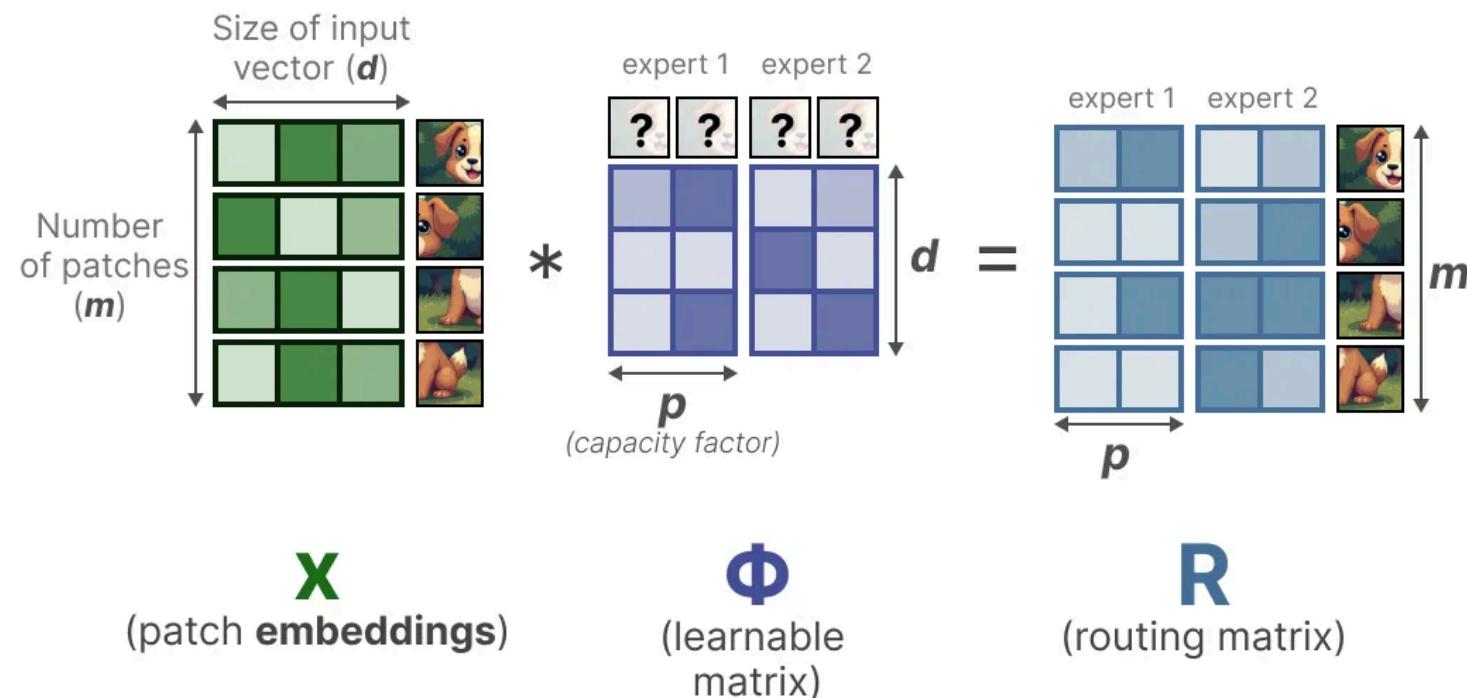
The priority routing allows fewer patches to be processed by focusing on the most important ones.

## From Sparse to Soft MoE

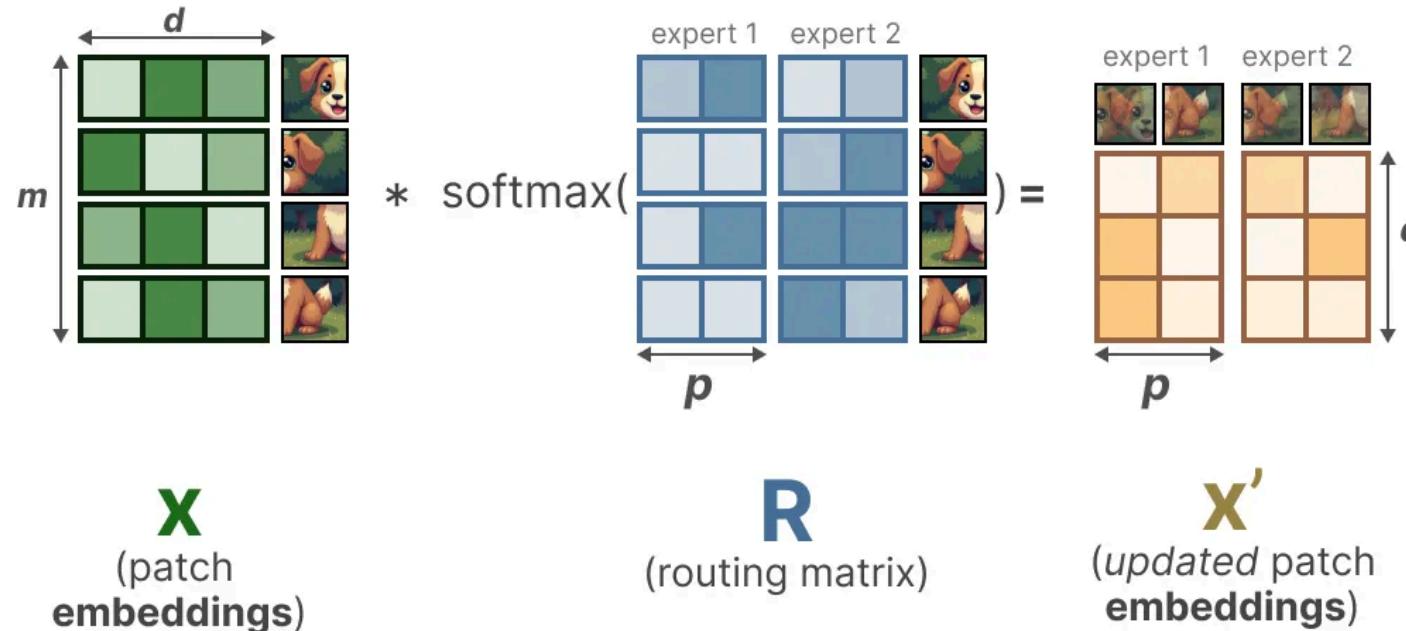
In V-MoE, the priority scorer helps differentiate between more and less important patches. However, patches are assigned to each expert, and information in unprocessed patches is lost.

Soft-MoE aims to go from a discrete to a soft patch (token) assignment by mixing patches.<sup>8</sup>

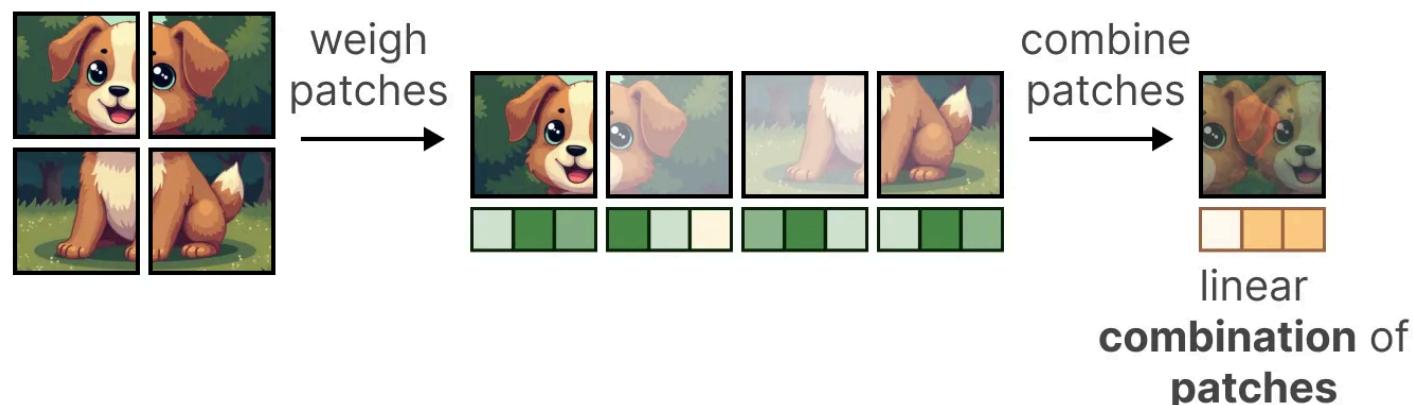
In the first step, we multiply the input  $x$  (the patch embeddings) with a learnable matrix  $\Phi$ . This gives us *router information* which tells us how related a certain token is to a given expert.



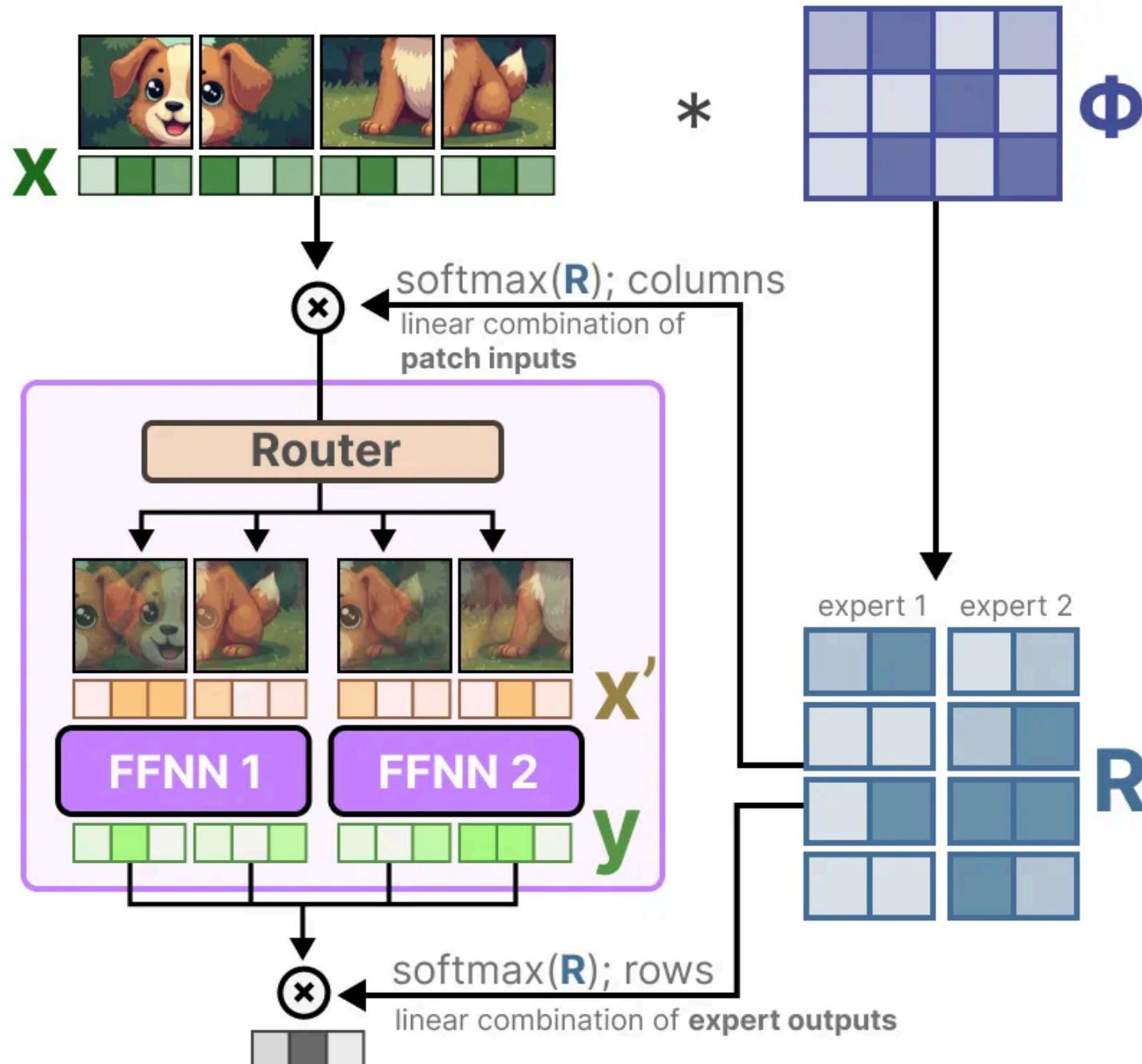
By then taking the softmax of the router information matrix (on the columns), we update the embeddings of each patch.

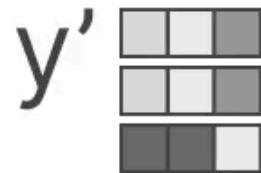


The updated patch embeddings are essentially the weighted average of all patch embeddings.



Visually, it is as if all patches were mixed. These combined patches are then sent to each expert. After generating the output, they are again multiplied with the router matrix.





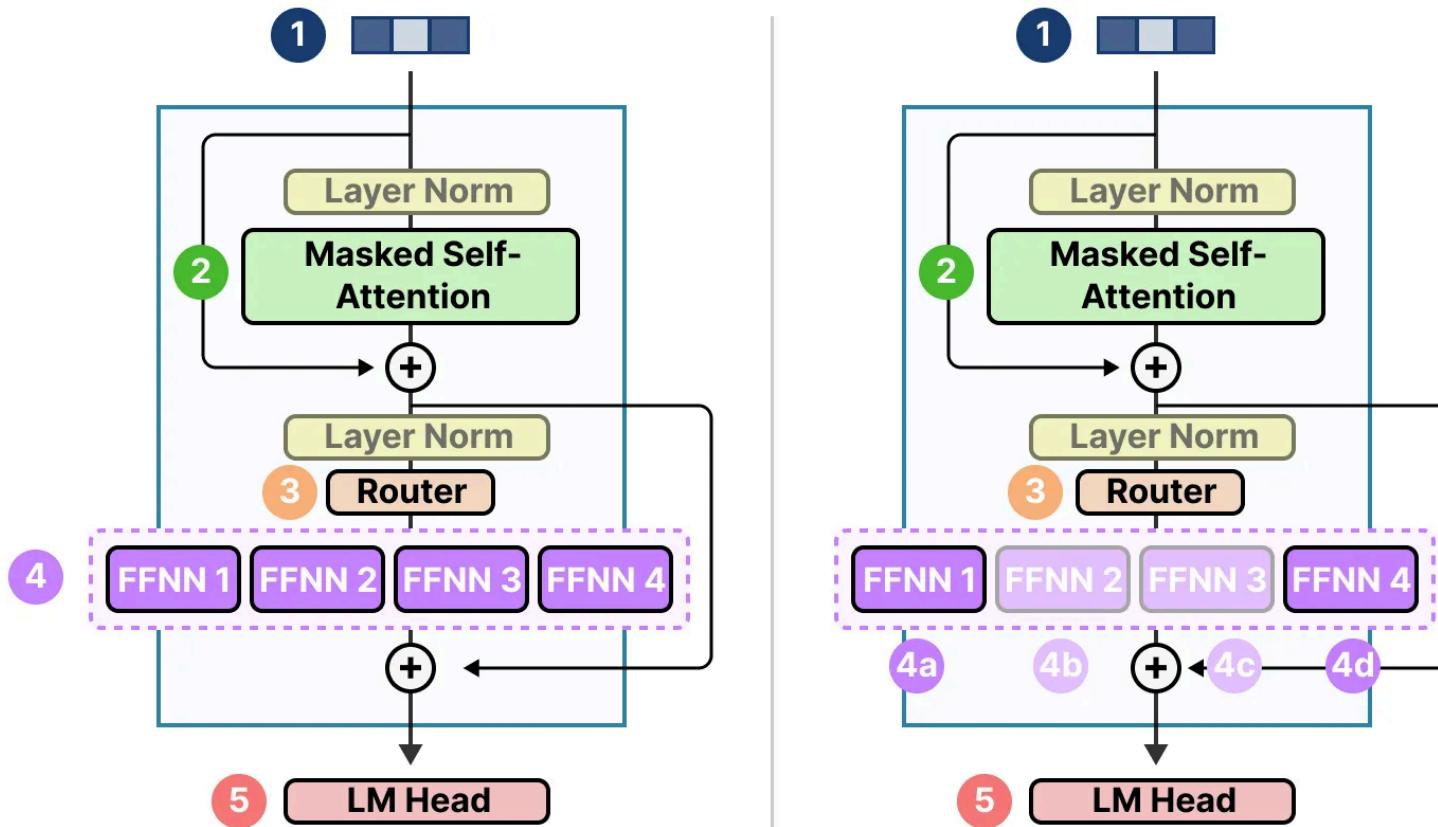
The router matrix affects the input on a token level and the output on an expert level.

As a result, we get “soft” patches/tokens that are processed instead of discrete input.

## Active vs. Sparse Parameters with Mixtral 8x7B

A big part of what makes MoE interesting is its computational requirements. Since only a subset of experts are used at a given time, we have access to more parameters than we are using.

Although a given MoE has more parameters to load (*sparse parameters*), fewer are activated since we only use some experts during inference (*active parameters*).



**Sparse Parameters**  
(loading model)

$$1 + 2 + 3 + 4 + 5$$

$$4a + 4b + 4c + 4d$$

4 experts *loaded*

**Active Parameters**  
(using model)

$$1 + 2 + 3 + 4 + 5$$

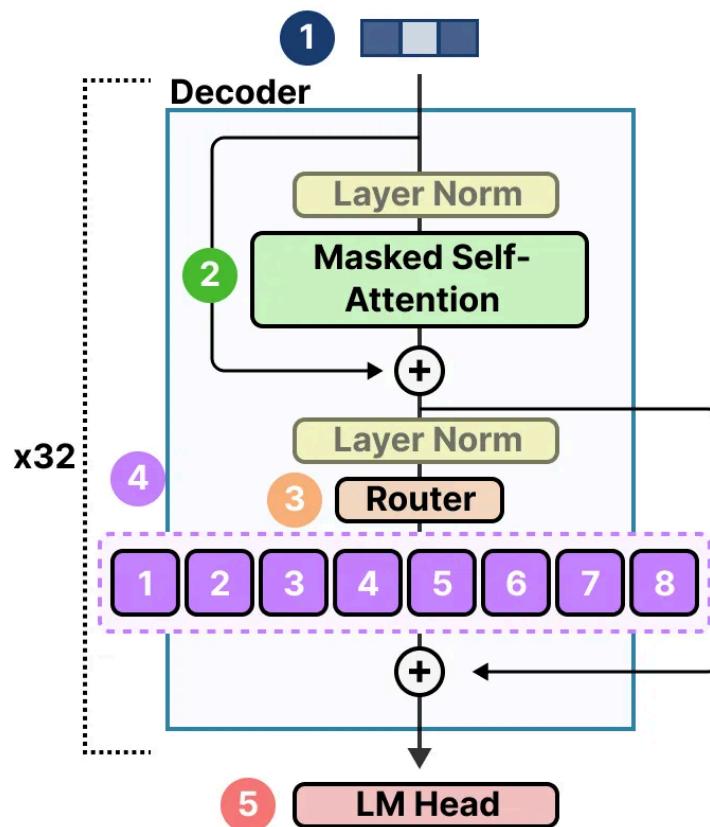
$$4a + 4d$$

2 experts *used*

In other words, we still need to load the entire model (including all experts) onto your device (*sparse parameters*) but when we run inference, we only need to use a subset (*active parameters*). MoE models need more VRAM to load in all experts but run faster during inference.

Let's explore the number of sparse vs active parameters with an example, Mixtral 8x7B.<sup>9</sup>

# Mixtral 8x7B



## 1 Embeddings

$$32000 \times 4096 = 131.072.000$$

embedding size      shared parameters

## 2 Attention

$$32 \times 41.943.040 = 1.342.177.280$$

repeated decoder blocks      (q, k, v)      shared parameters

## 3 Router

$$8 \times 4096 = 32.768$$

# experts      T      shared parameters

## 4 Experts

$$8 \times 5.637144.576 = 45.097.156.608$$

# experts      T      total parameters

$$2 \times 5.637144.576 = 11.274.289.152$$

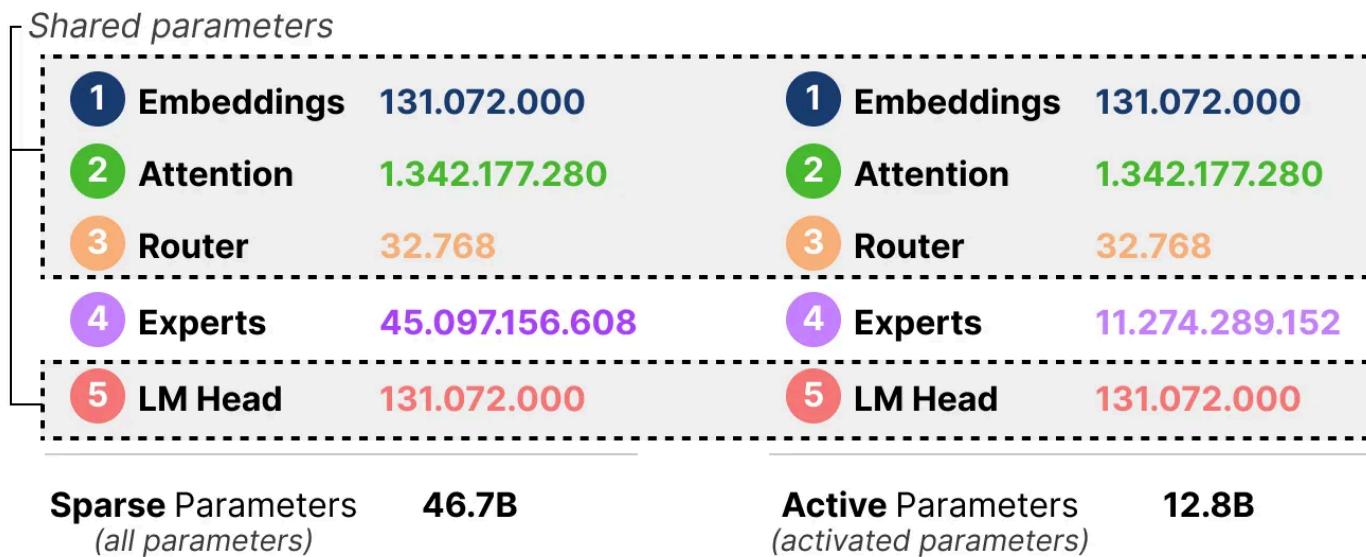
# experts      expert size      active parameters

## 5 LM Head

$$32000 \times 4096 = 131.072.000$$

shared parameters

Here, we can see that each expert is 5.6B in size and not 7B (although there are 8 experts).

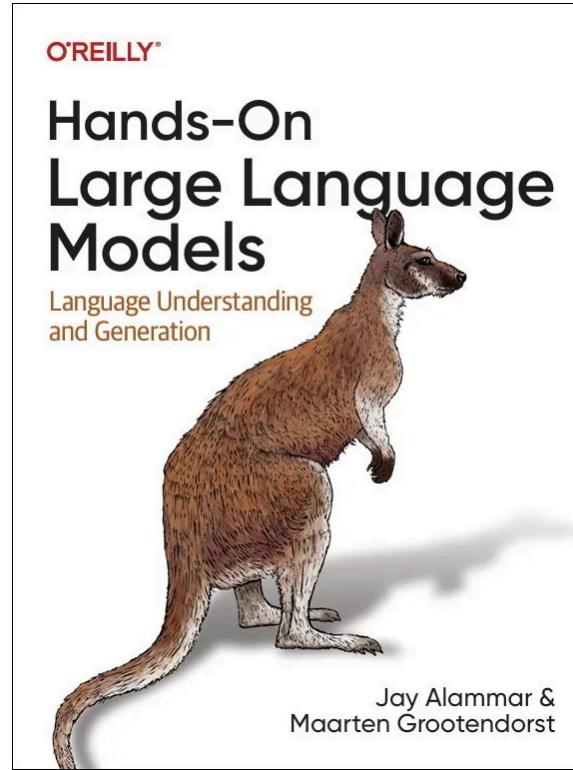


We will have to *load* 8x5.6B (46.7B) parameters (along with all shared parameters) but we only need to use 2x5.6B (12.8B) parameters for *inference*.

## Conclusion

This concludes our journey with a Mixture of Experts! Hopefully, this post gives you a better understanding of the potential of this interesting technique. Now that almost all sets of models contain at least one MoE variant, it feels like it is here to stay.

To see more visualizations related to LLMs and to support this newsletter, check out the book I wrote on Large Language Models!



[Official website](#) of the book. You can order the book on [Amazon](#). All code is uploaded to [GitHub](#).

## Resources

Hopefully, this was an accessible introduction to Mixture of Experts. If you want to go deeper, I would suggest the following resources:

- [This](#) and [this](#) paper are great overviews of the latest MoE innovations.

- The paper on [expert choice routing](#) that has gained some traction.
  - A [great blog post](#) going through some of the major papers (and their findings).
  - A similar [blog post](#) that goes through the timeline of MoE.
- 

- 1 Zoph, Barret, et al. "St-moe: Designing stable and transferable sparse expert models. arXiv 2022." *arXiv preprint arXiv:2202.08906*.
- 2 Shazeer, Noam, et al. "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer." *arXiv preprint arXiv:1701.06538* (2017).
- 3 Shazeer, Noam, et al. "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer." *arXiv preprint arXiv:1701.06538* (2017).
- 4 Lepikhin, Dmitry, et al. "Gshard: Scaling giant models with conditional computation and automatic sharding." *arXiv preprint arXiv:2006.16668* (2020).
- 5 Fedus, William, Barret Zoph, and Noam Shazeer. "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity." *Journal of Machine Learning Research* 23.120 (2022): 1-39.
- 6 Dosovitskiy, Alexey. "An image is worth 16x16 words: Transformers for image recognition at scale." *arXiv preprint arXiv:2010.11929* (2020).

- 7 Riquelme, Carlos, et al. "Scaling vision with sparse mixture of experts." *Advances in Neural Information Processing Systems* 34 (2021): 8583-8595.
- 8 Puigcerver, Joan, et al. "From sparse to soft mixtures of experts." *arXiv preprint arXiv:2308.00951* (2023).
- 9 Jiang, Albert Q., et al. "Mixtral of experts." *arXiv preprint arXiv:2401.04088* (2024).



207 Likes · 19 Restacks

← Previous

## Discussion about this post

Comments

Restacks



Write a comment...



Shanya Chaubey Oct 11

...

♥ Liked by Maarten Grootendorst

The simplicity of the explanation was very helpful.

Thank you for creating this

 LIKE (1)  REPLY

 SHARE



Ruben 4h

...

Hi Maarten, I wonder why is not possible to prune non active parameters of a model with MoE during run time so as to reduce memory requirements. Thanks!

 LIKE  REPLY

 SHARE

1 reply by Maarten Grootendorst

9 more comments...