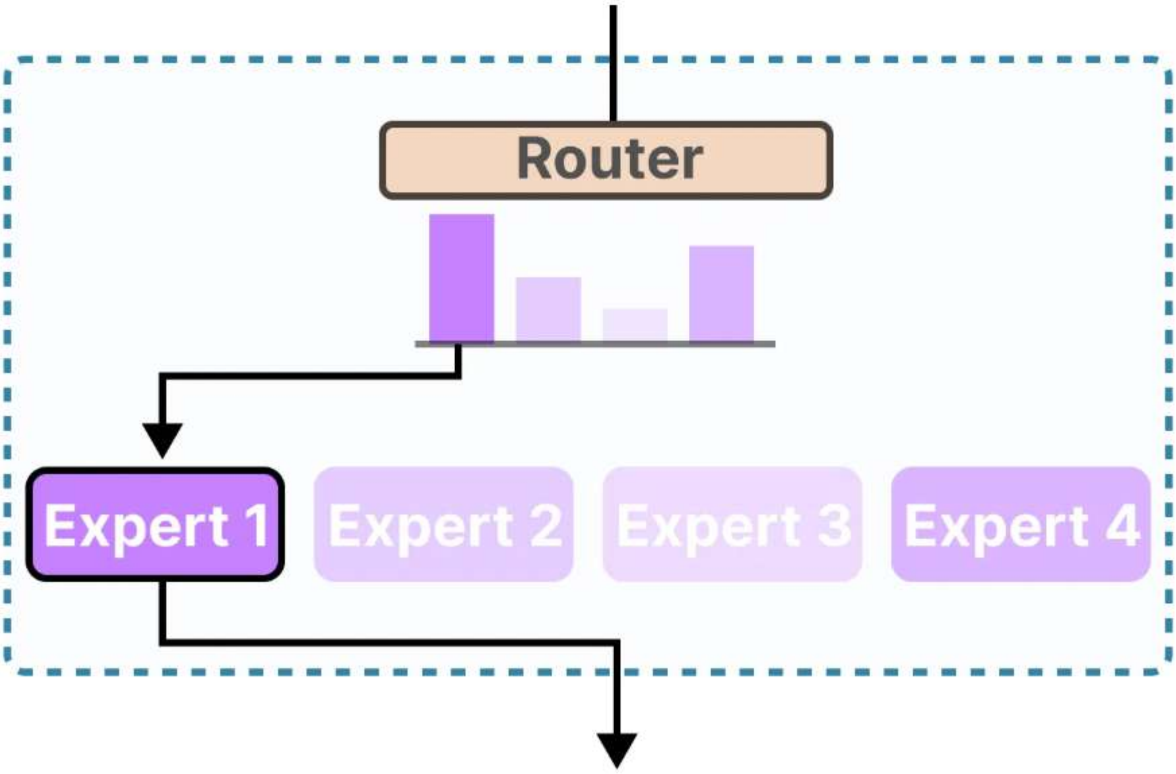

Router
Expert 1
Expert 2
Expert 3
Expert 4

What is Mixture of Experts?

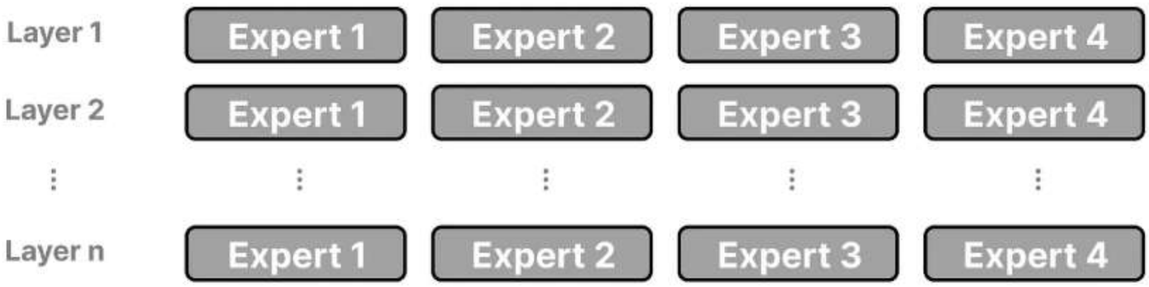
Mixture of Experts (MoE) is a technique that uses many different sub-models (or “experts”) to improve the quality of LLMs.

Two main components define a MoE:

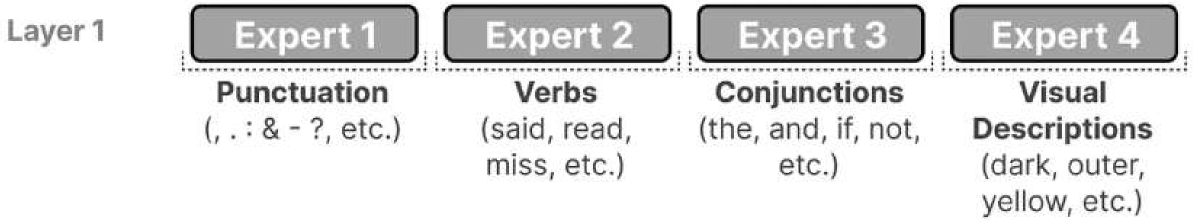
- **Experts** - Each FFNN layer now has a set of “experts” of which a subset can be chosen. These “experts” are typically FFNNs themselves.

- **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:

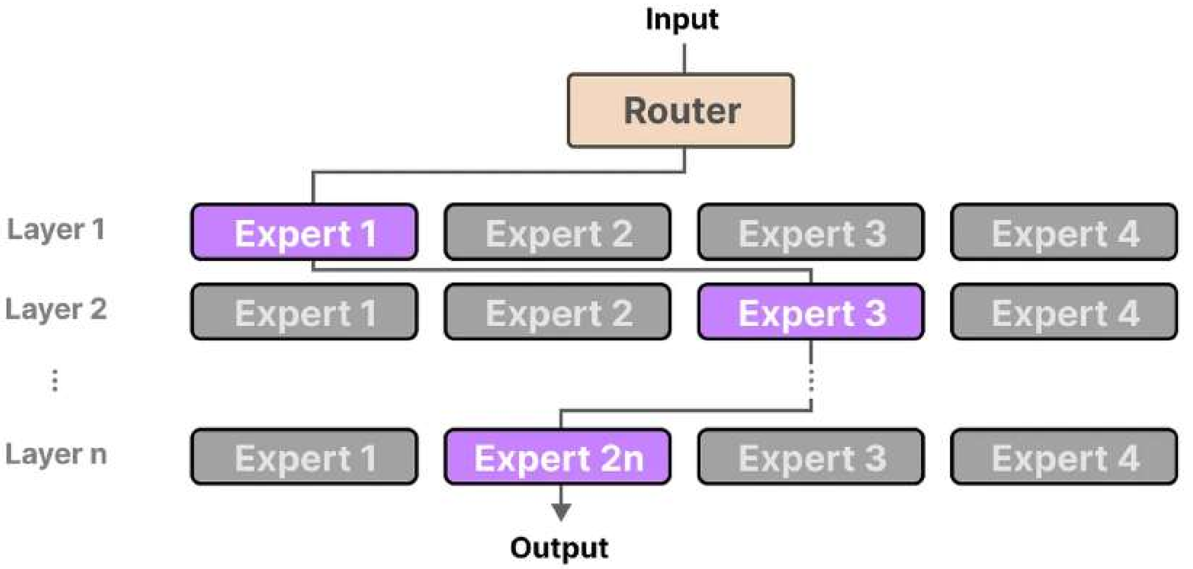


Know that an “expert” is not specialized in a specific domain like “Psychology” or “Biology”. At most, it learns syntactic information on a word level instead:



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:



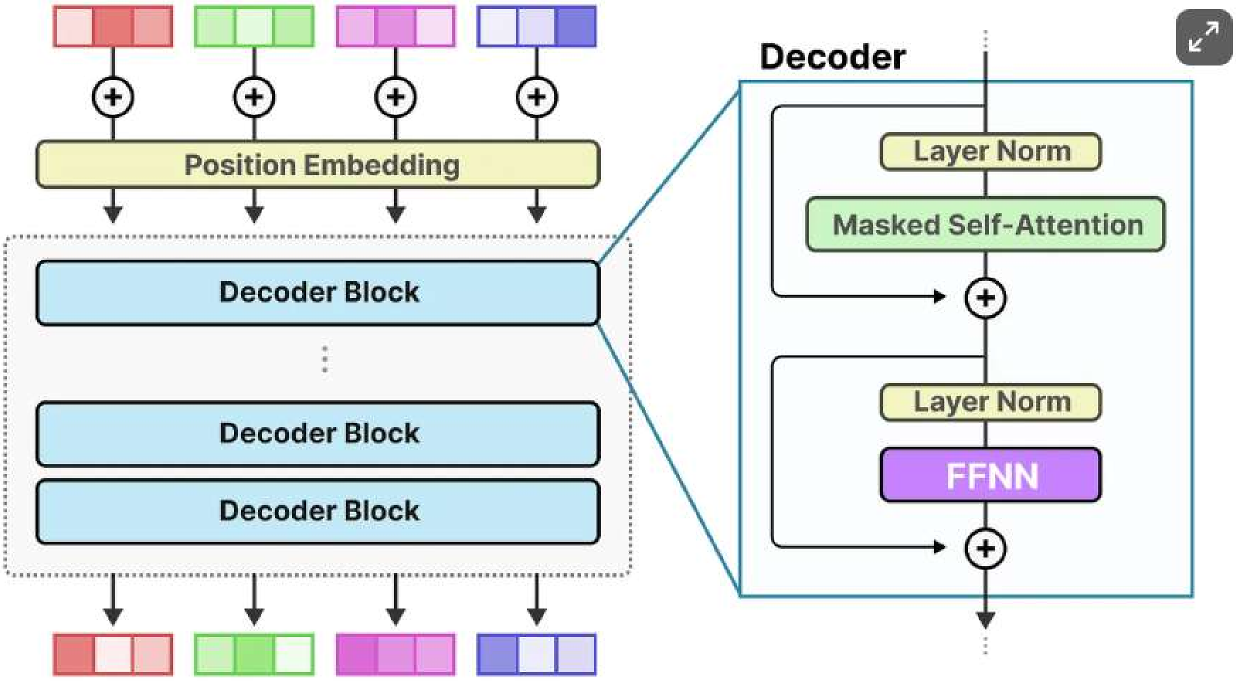
The Experts

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

Dense Layers

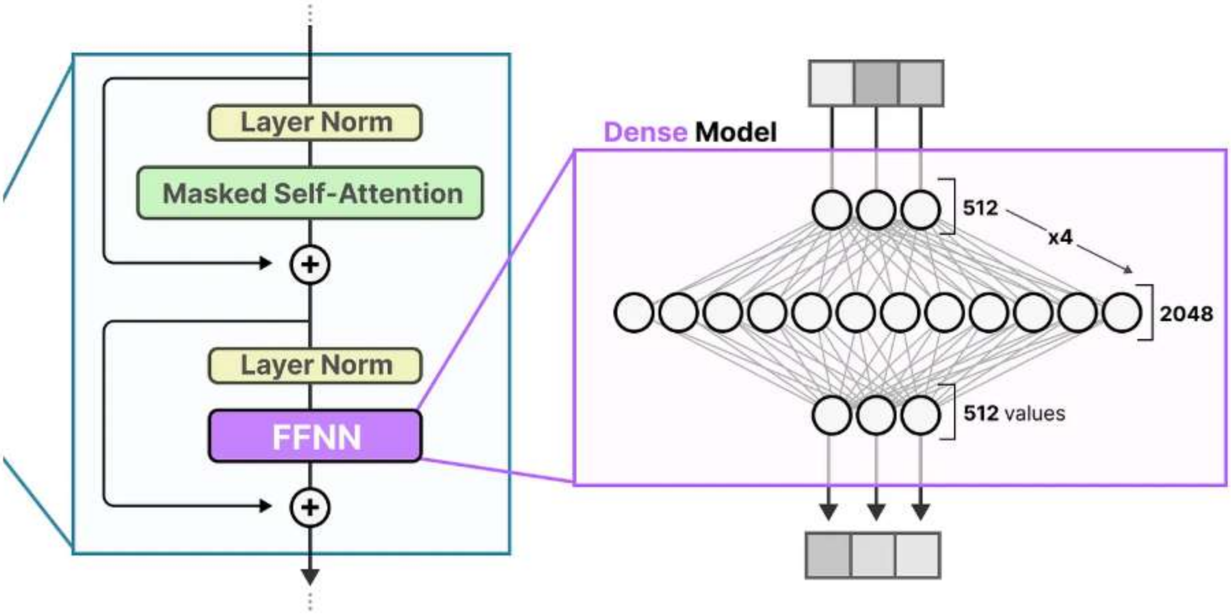
Mixture of Experts (MoE) all start from a relatively basic functionality of LLMs, namely the *Feedforward Neural Network* (FFNN).

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.

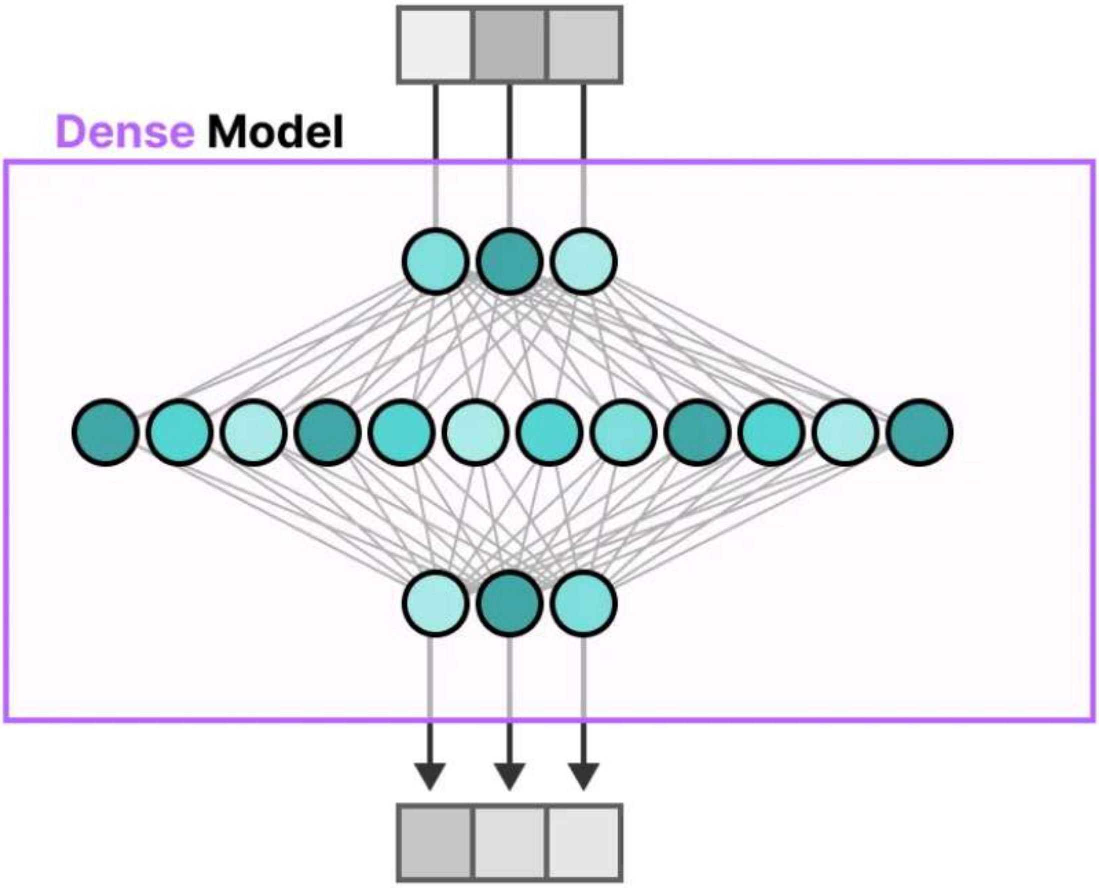
The FFNN, however, does grow quickly in size. To learn these complex relationships, it typically expands on the input it receives:



Sparse Layers

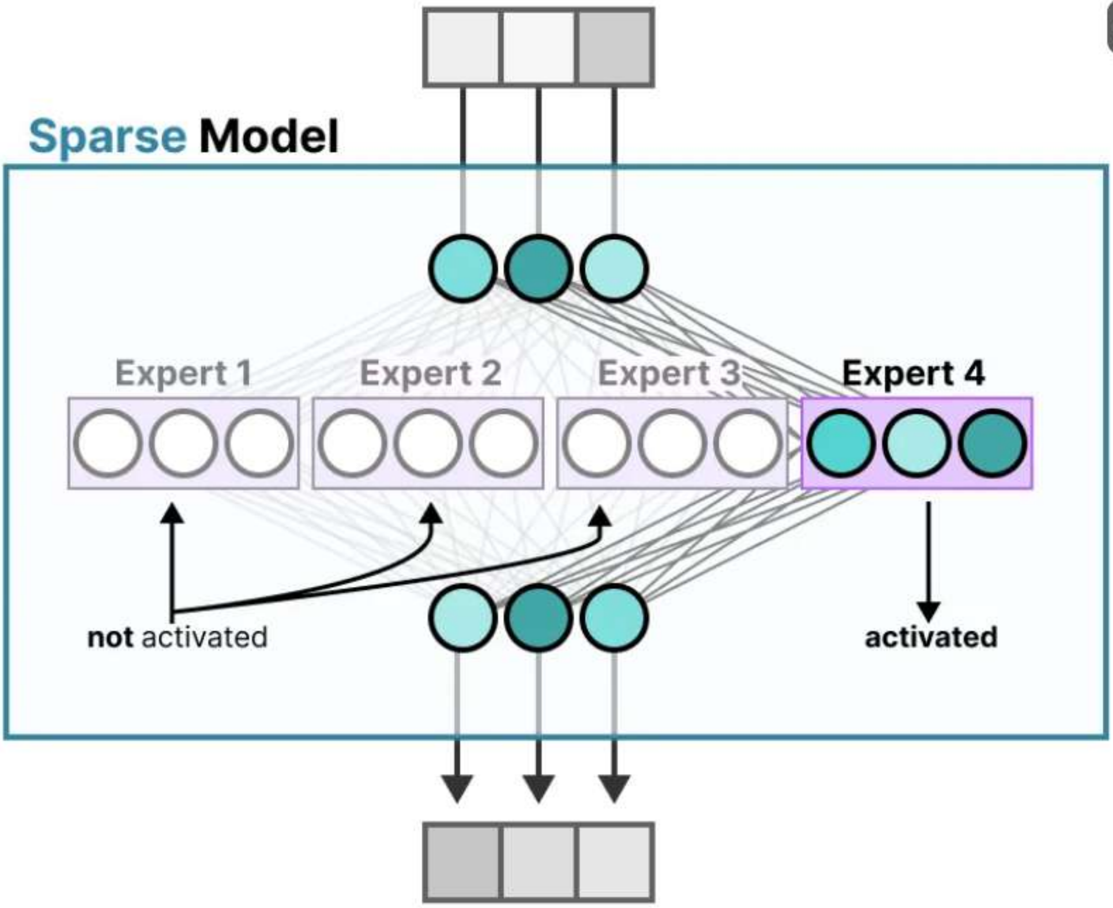
The FFNN in a traditional Transformer is called a **dense** model since all parameters (its weights and biases) are activated. Nothing is left behind and everything is used to calculate the output.

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



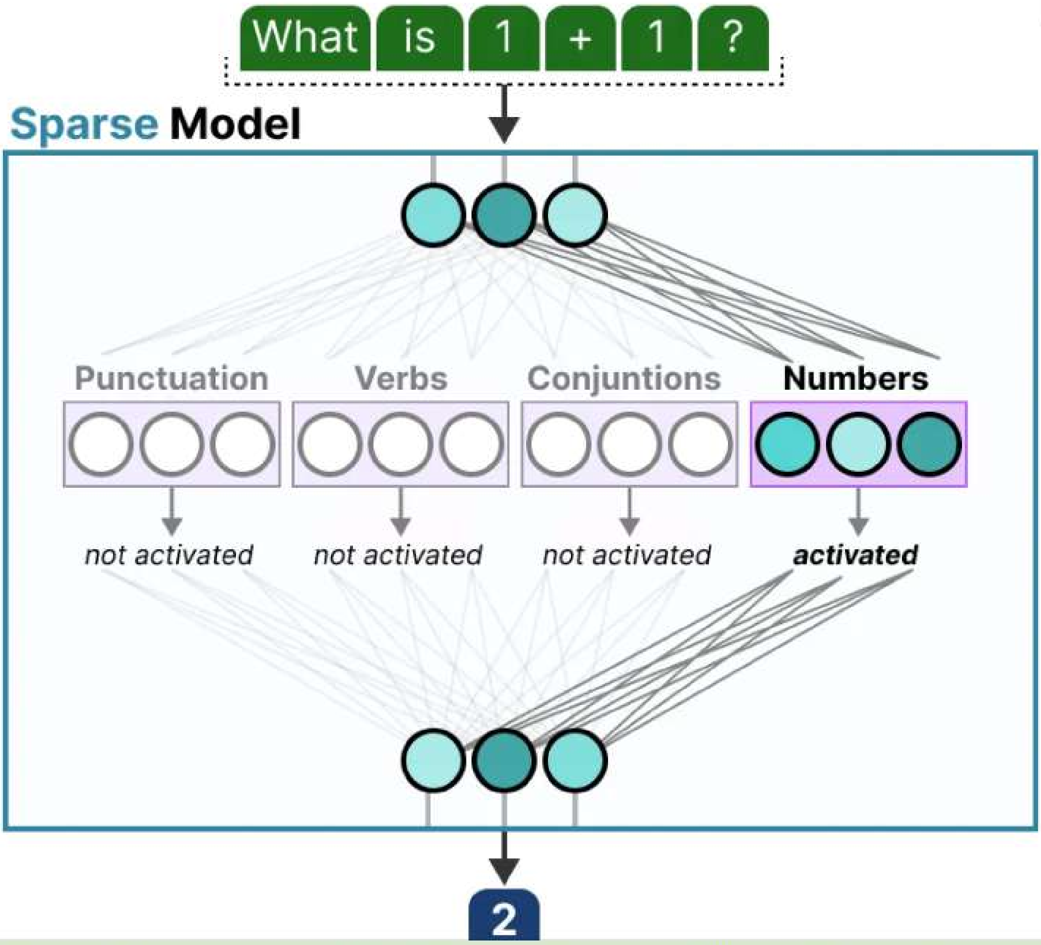
In contrast, **sparse** models only activate a portion of their total parameters and are closely related to Mixture of Experts.

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:



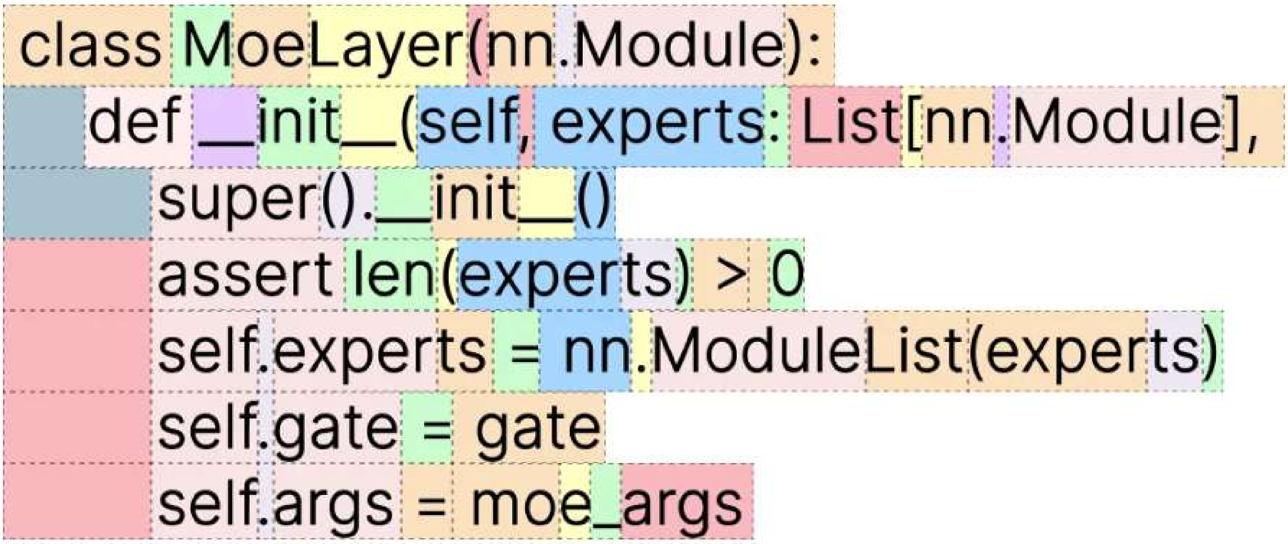
The underlying idea is that each expert learns different information during training. Then, when running inference, only specific experts are used as they are most relevant for a given task.

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



ressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fce63e5cc-9b82-45b4-b3dc-9db0cac47da3_880x748.png

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



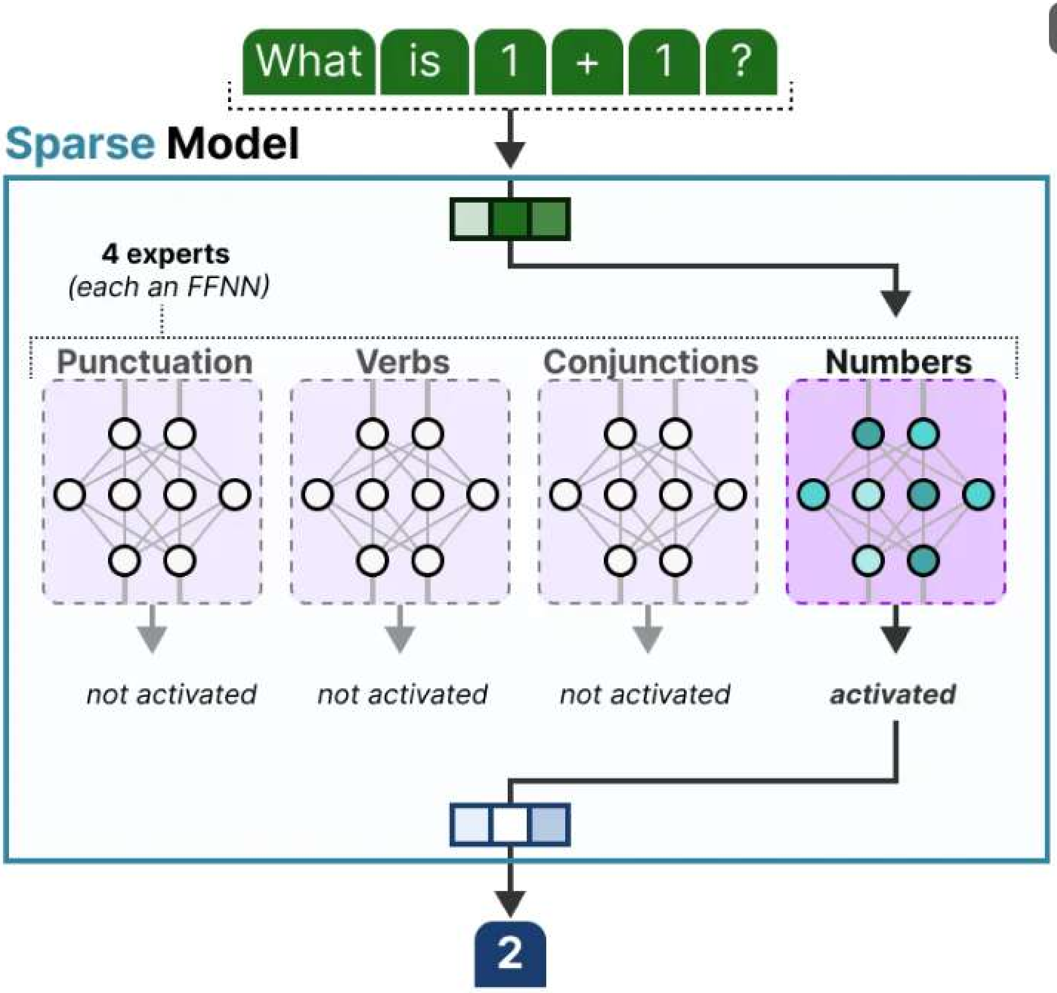
```
class MoeLayer(nn.Module):
    def __init__(self, experts: List[nn.Module],
        super().__init__()
        assert len(experts) > 0
        self.experts = nn.ModuleList(experts)
        self.gate = gate
        self.args = moe_args
```

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

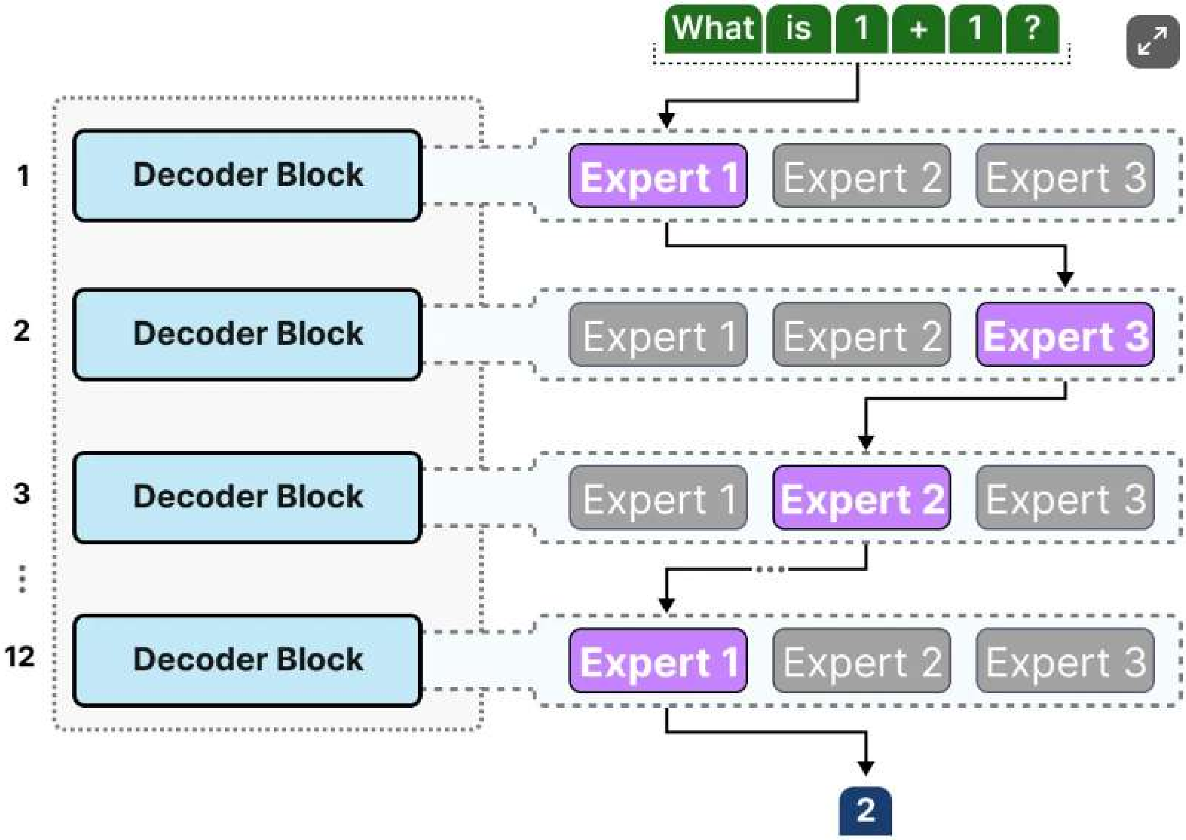
Thus, although decoder experts do not seem to have a specialism they do seem to be used consistently for certain types of tokens.

The Architecture of Experts

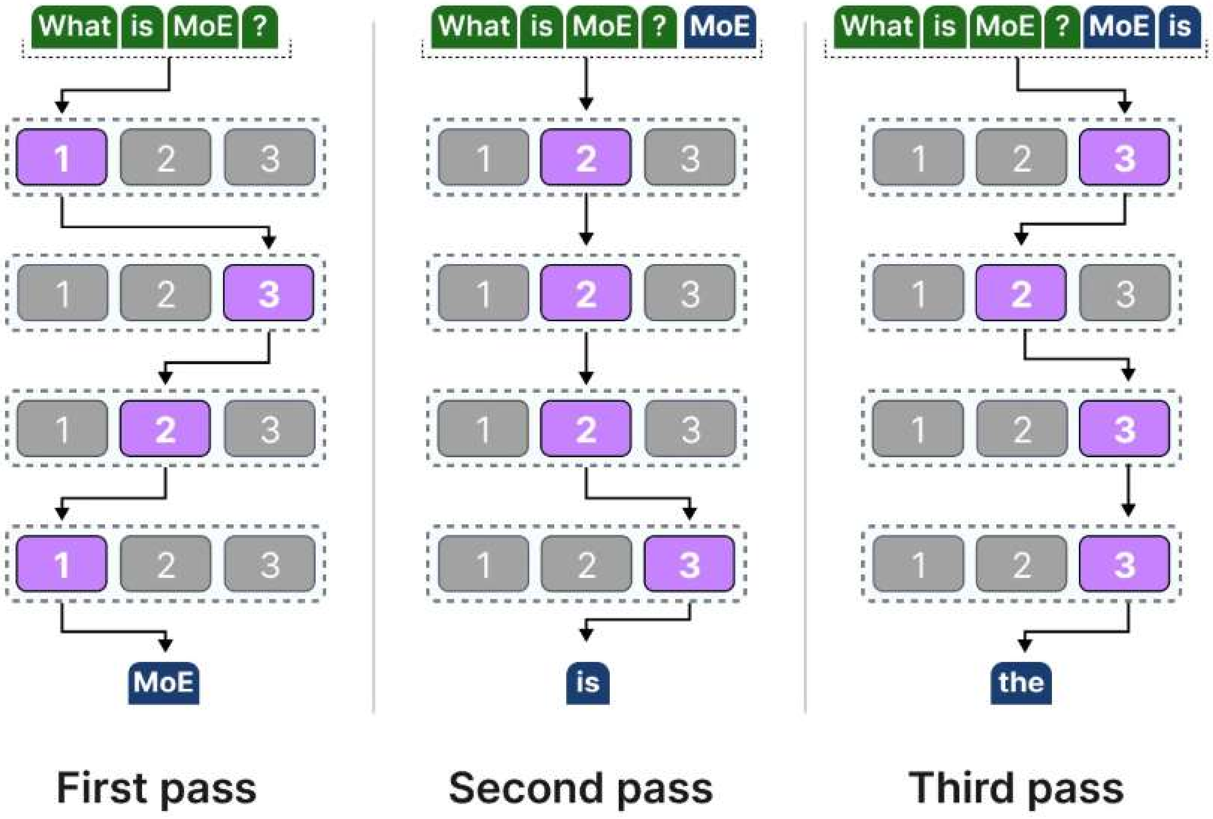
Although it's nice to visualize experts as a hidden layer of a dense model cut in pieces, they are often whole FFNNs themselves:



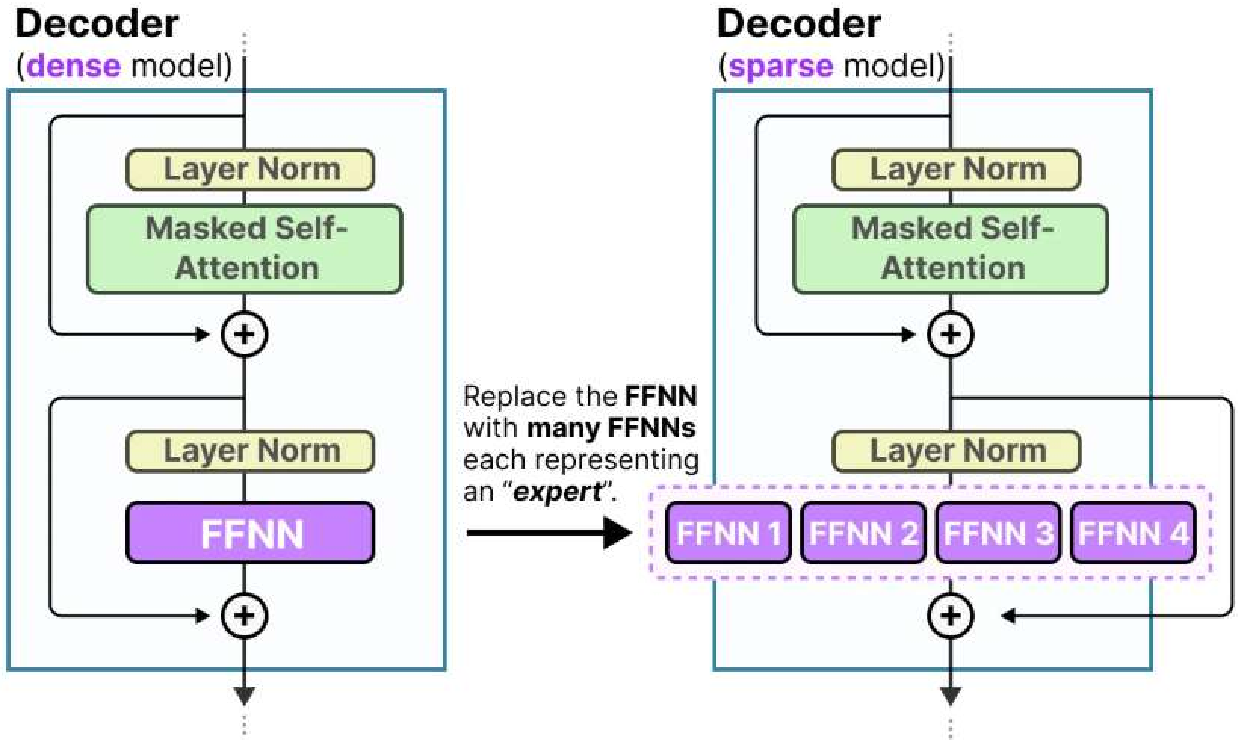
Since most LLMs have several decoder blocks, a given text will pass through multiple experts before the text is generated:



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.

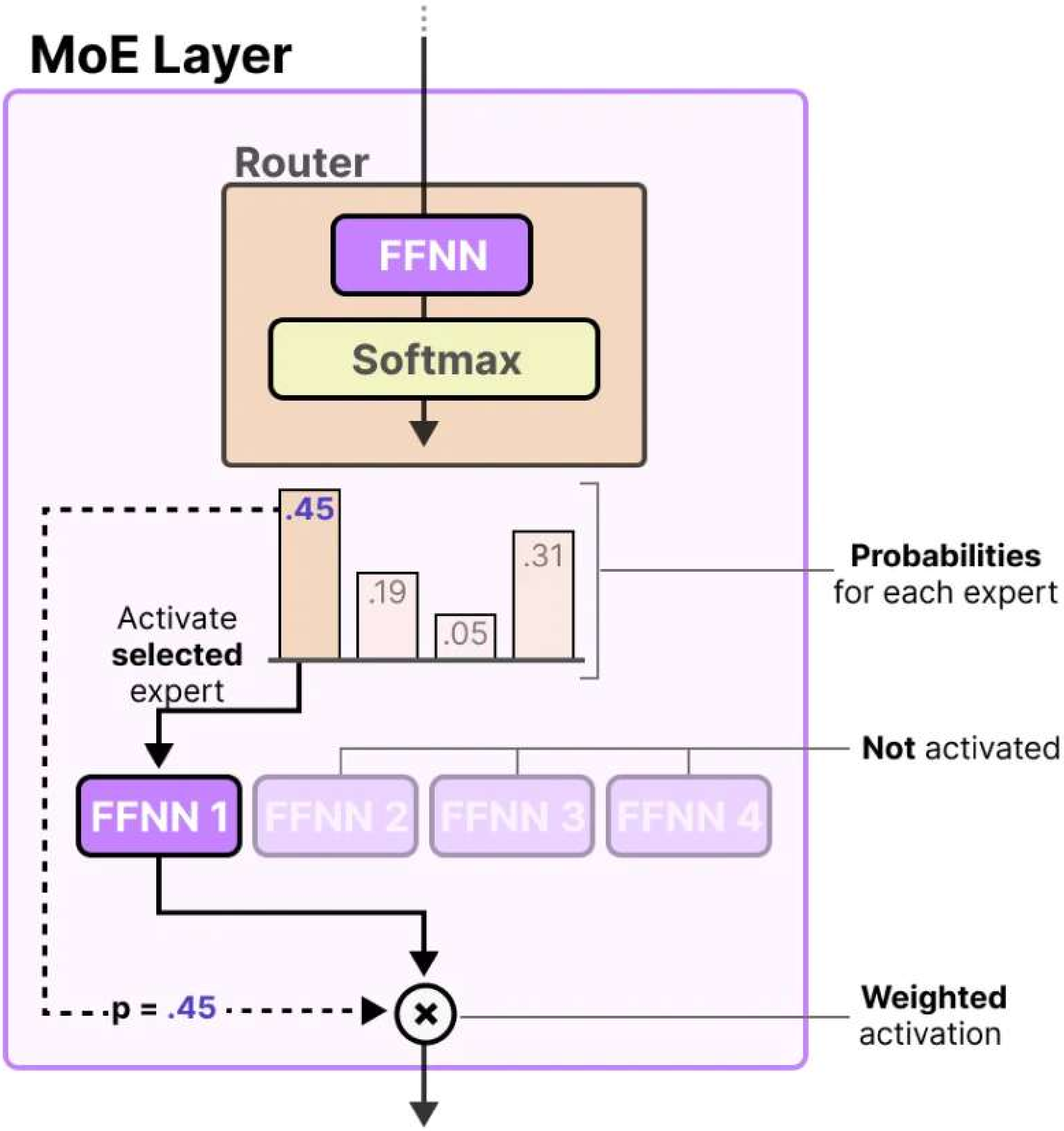
The Routing Mechanism

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.

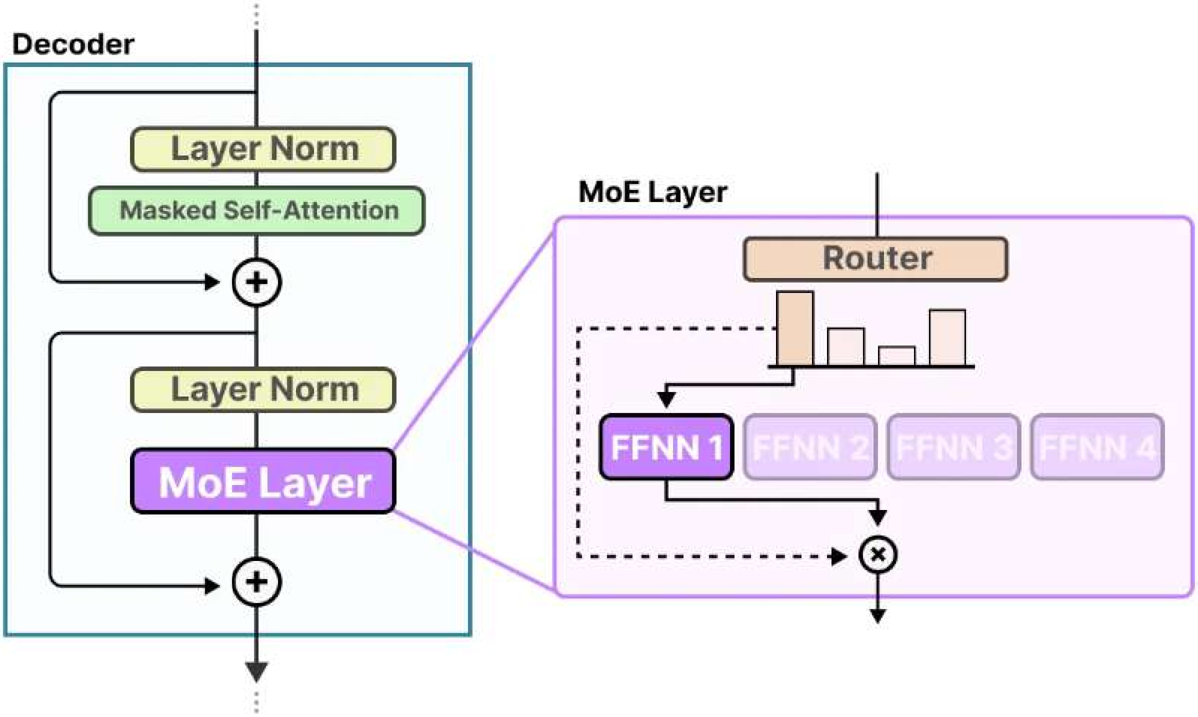
The Router

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
Router
FFNN
Softmax
.45
.19
.05
.31
Probabilities for each expert
Activate selected expert
Not activated
FFNN 1
FFNN 2
FFNN 3
FFNN 4
p = .45
Weighted activation

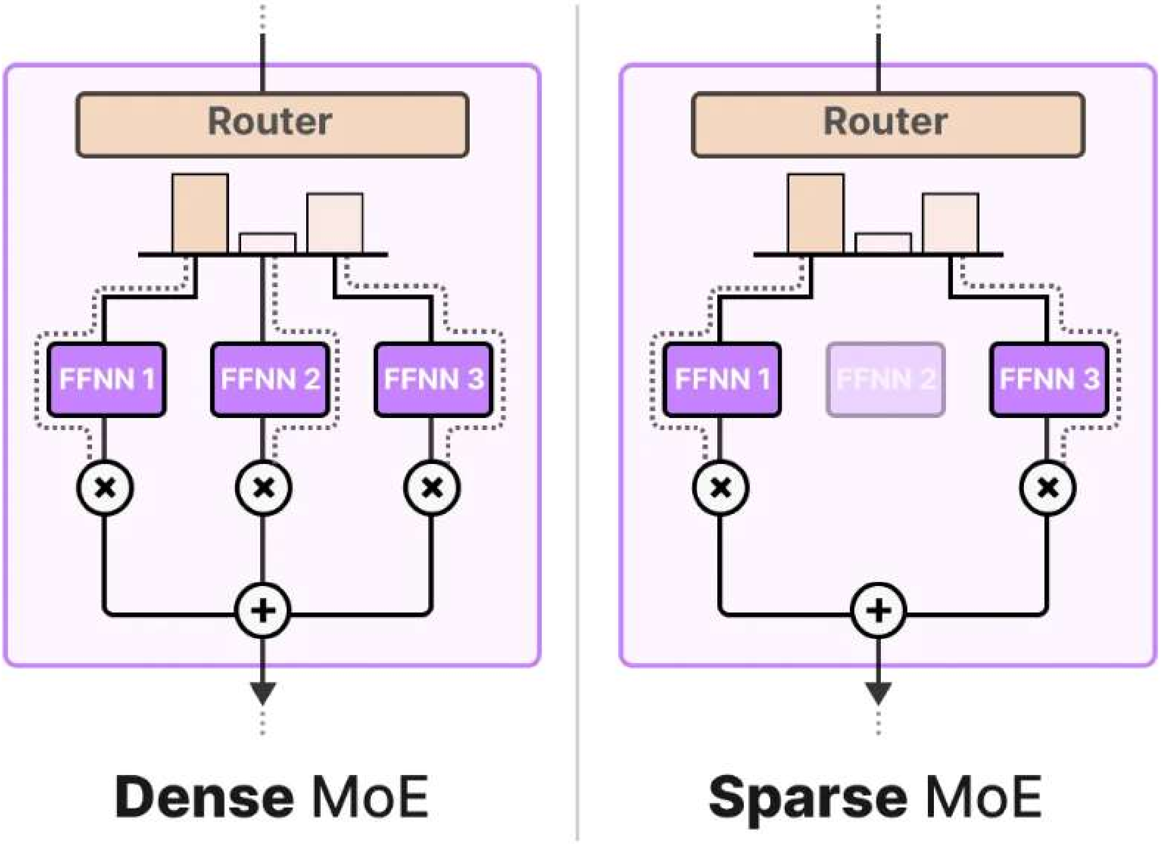
The expert layer returns the output of the selected expert multiplied by the gate value (selection probabilities).

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.



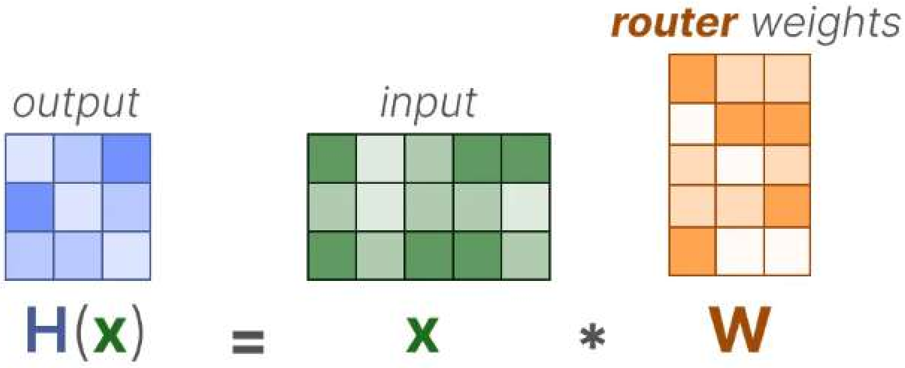
For instance, given a set of tokens, a MoE will distribute the tokens across all experts whereas a Sparse MoE will only select a few experts.

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.

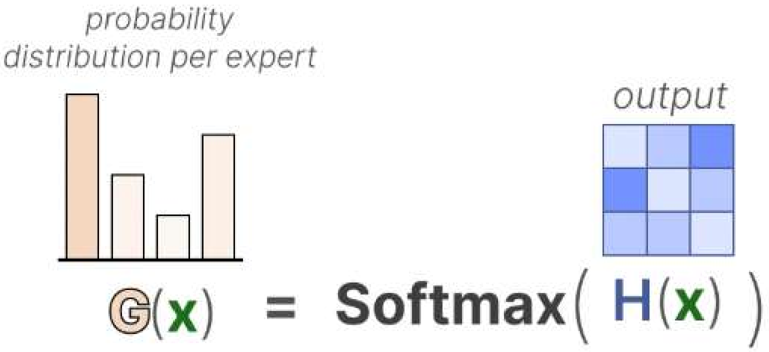
Selection of Experts

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 (***x***) by the router weight matrix (***W***):

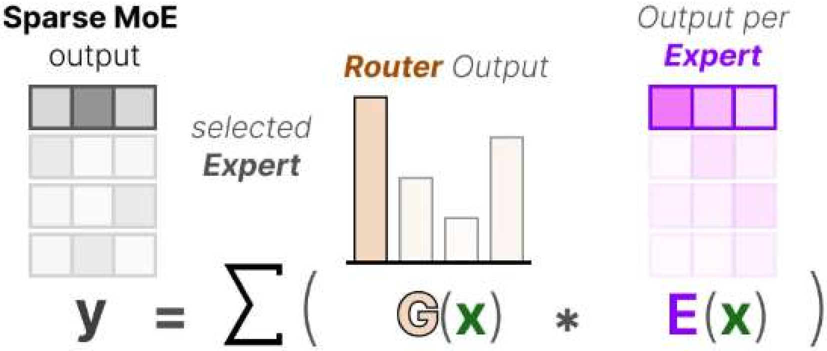


Then, we apply a **SoftMax** on the output to create a probability distribution **G**(***x***) per expert:



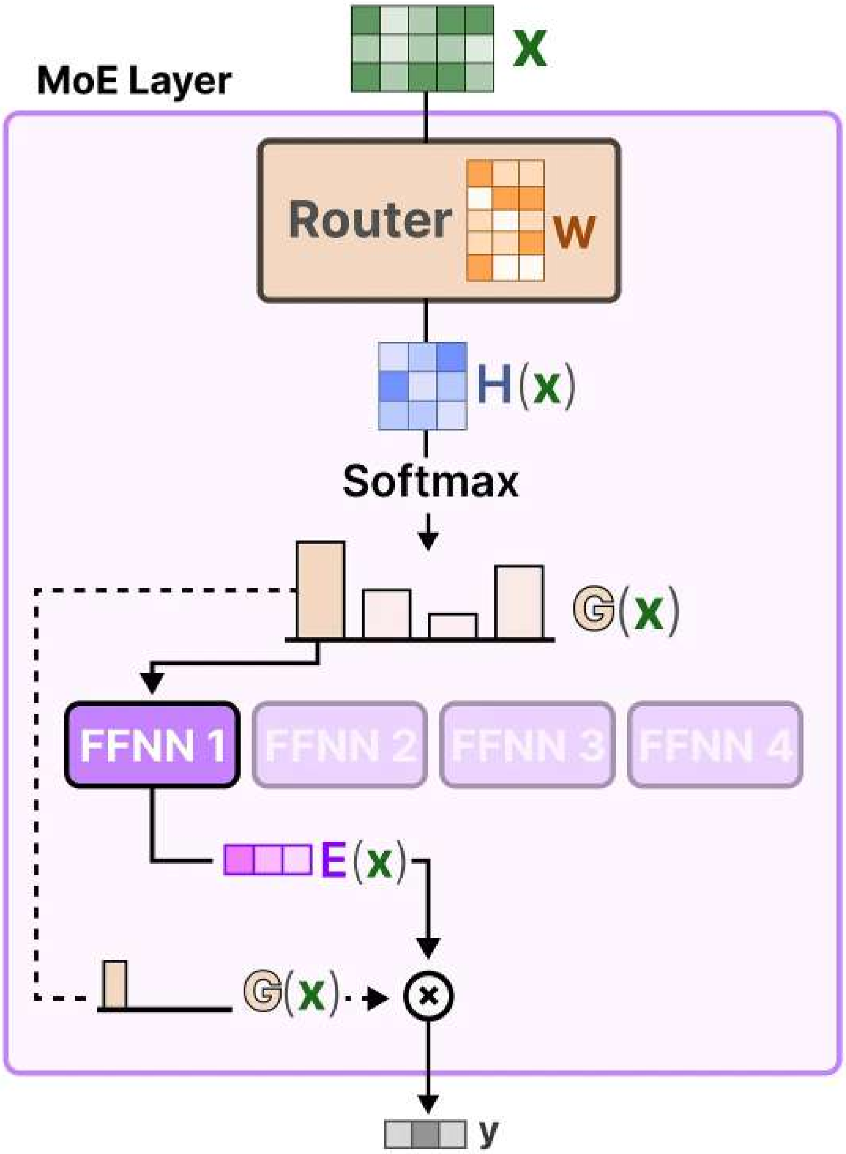
The router uses this probability distribution to choose the best matching expert for a given input.

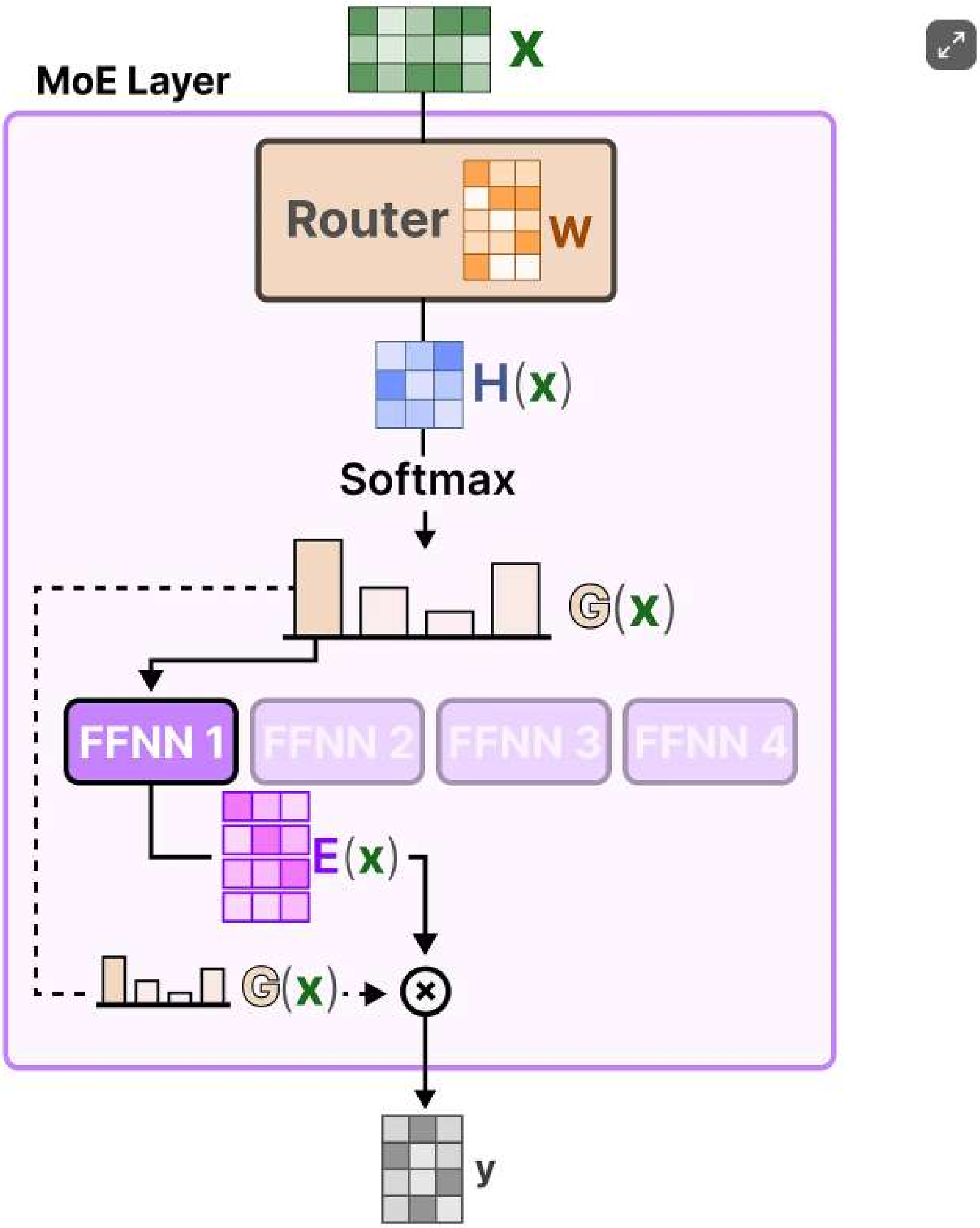
Finally, we multiply the output of each router with each selected expert and sum the results.



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

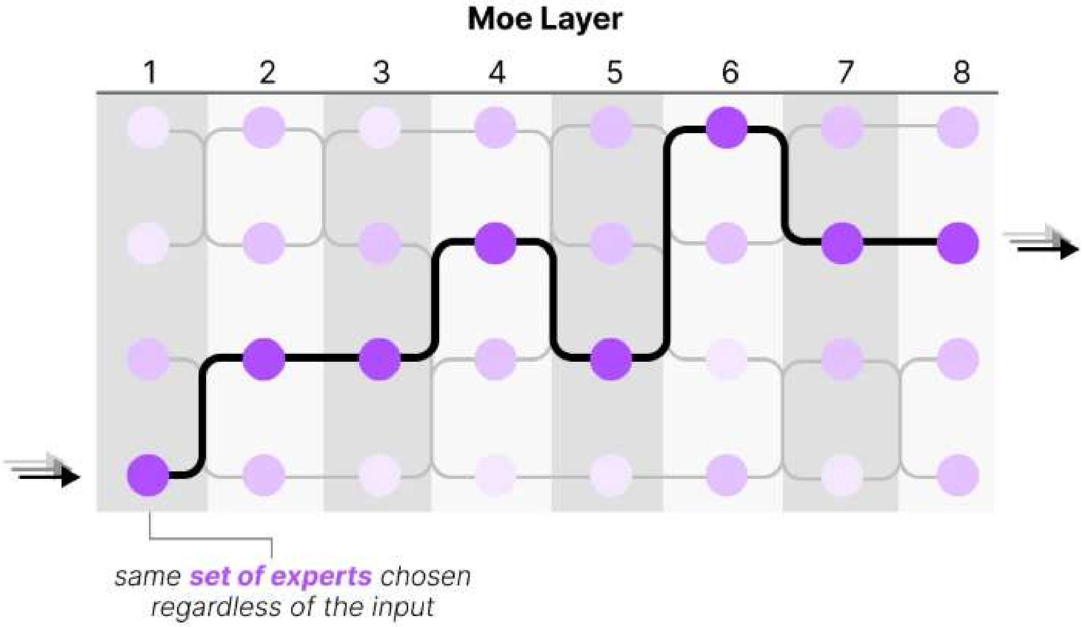
Let's put everything together and explore how the input flows through the router and experts:




x
MoE Layer
Router
W
H(x)
Softmax
G(x)
FFNN 1
FFNN 2
FFNN 3
FFNN 4
E(x)
G(x)
y

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.

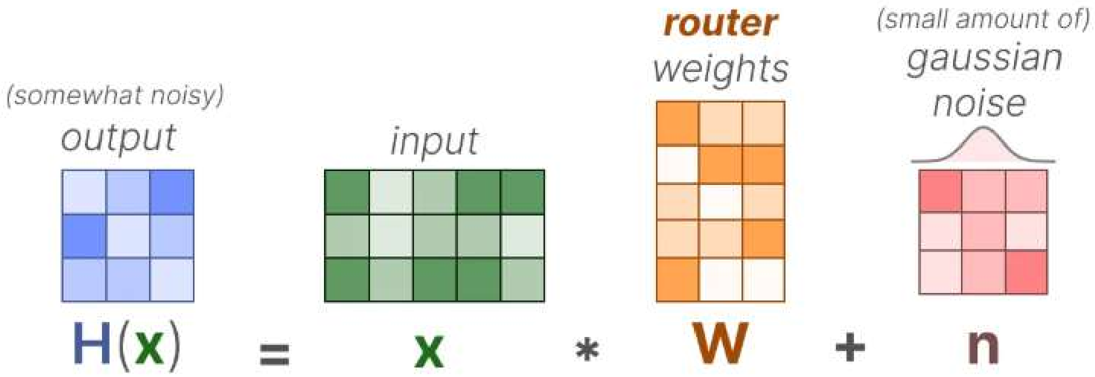
Instead, we want equal importance among experts during training and inference, which we call **load balancing**. In a way, it's to prevent overfitting on the same experts.

Load Balancing

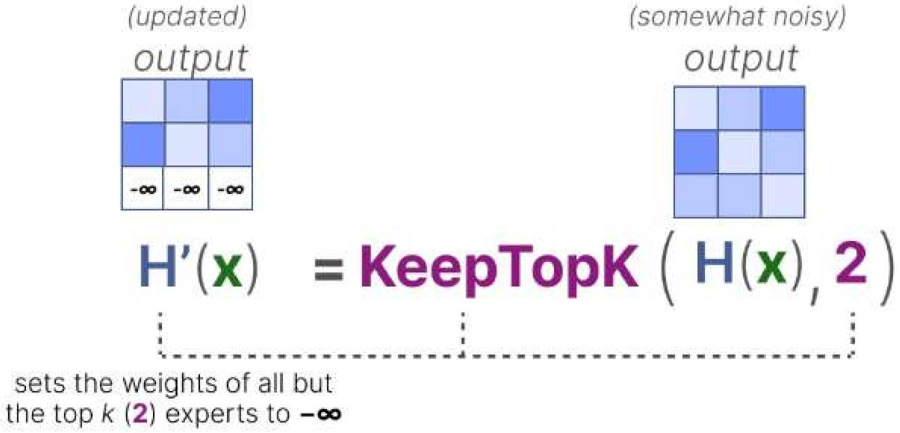
To balance the importance of experts, we will need to look at the router as it is the main component to decide which experts to choose at a given time.

KeepTopK

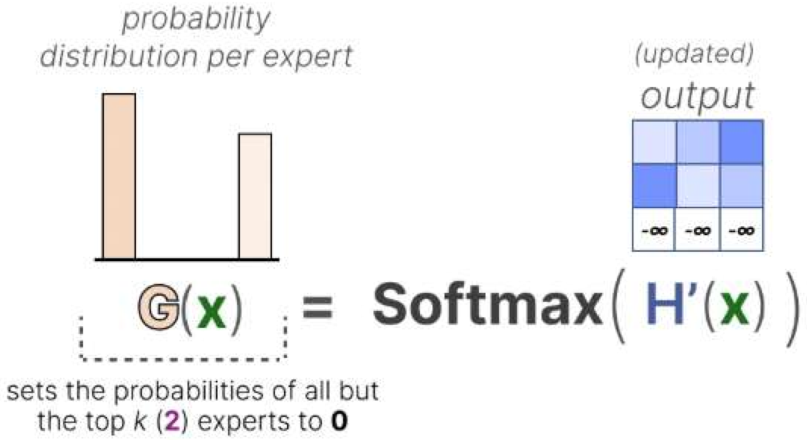
One method of load balancing the router is through a straightforward extension called KeepTopK [2]. By introducing trainable (gaussian) noise, we can prevent the same experts from always being picked:



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


(updated)
output
-∞ -∞ -∞
(somewhat noisy)
output
H'(x) = KeepTopK (H(x), 2)
sets the weights of all but the top k (2) experts to −∞

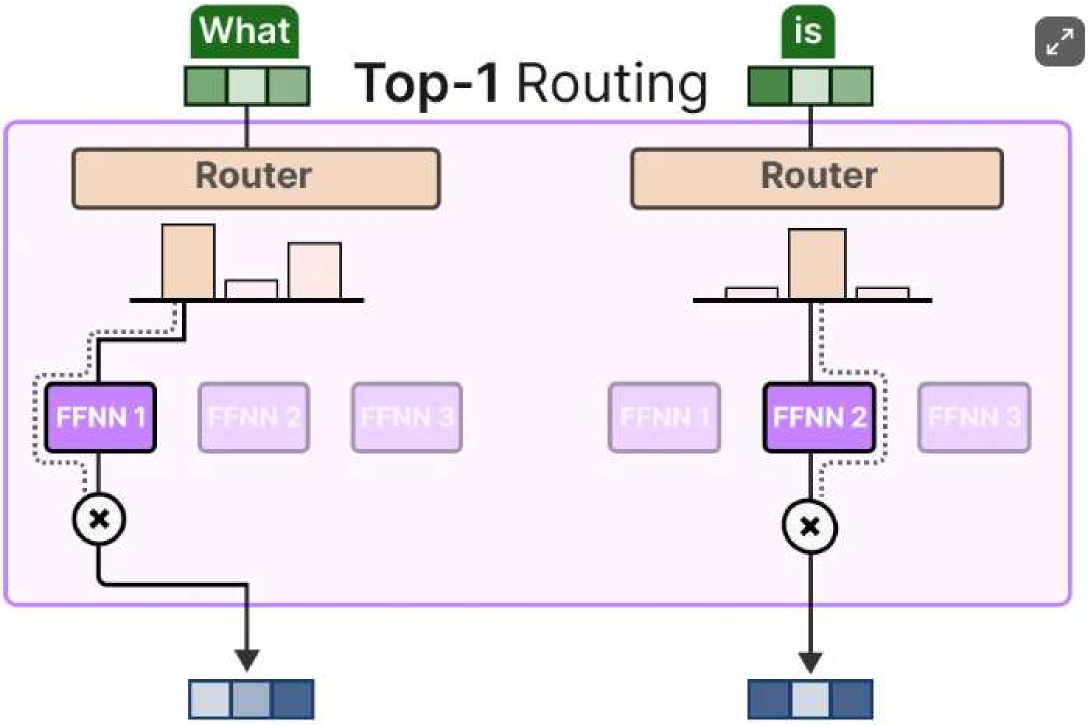
By setting these weights to -∞, the output of the SoftMax on these weights will result in a probability of **0**:


probability distribution per expert
(updated)
output
-∞ -∞ -∞
G(x) = Softmax(H'(x))
sets the probabilities of all but the top k (2) experts to 0

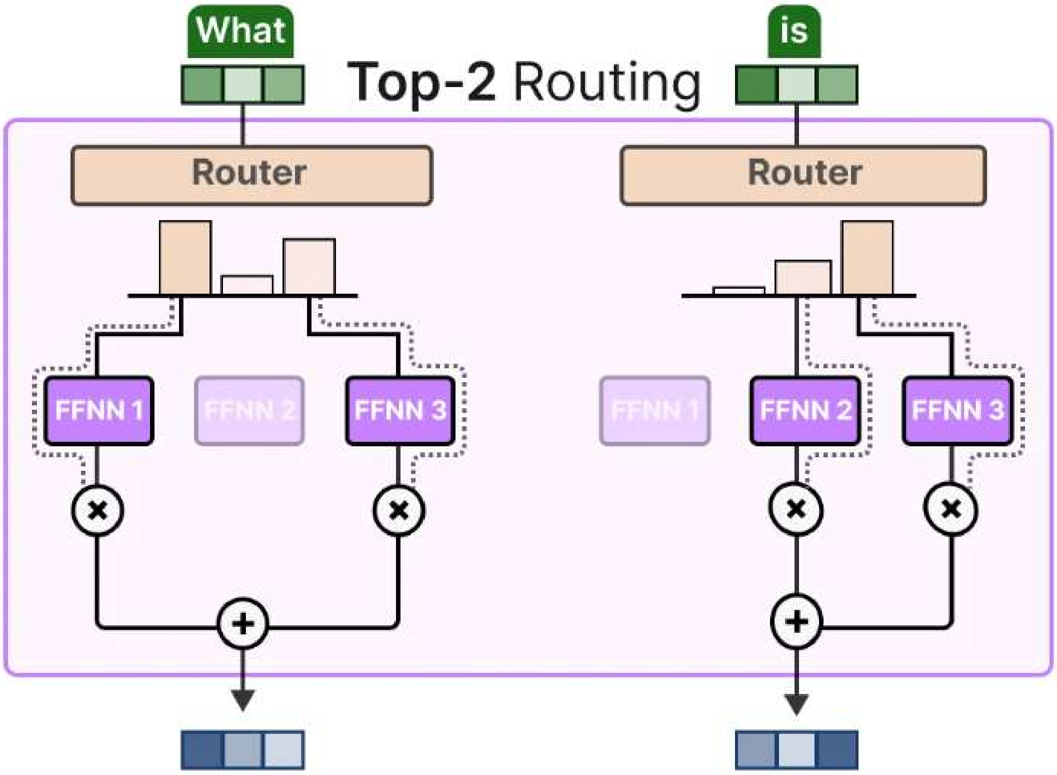
The KeepTopK strategy is one that many LLMs still use despite many promising alternatives. Note that KeepTopK can also be used without the additional noise.

Token Choice

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



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



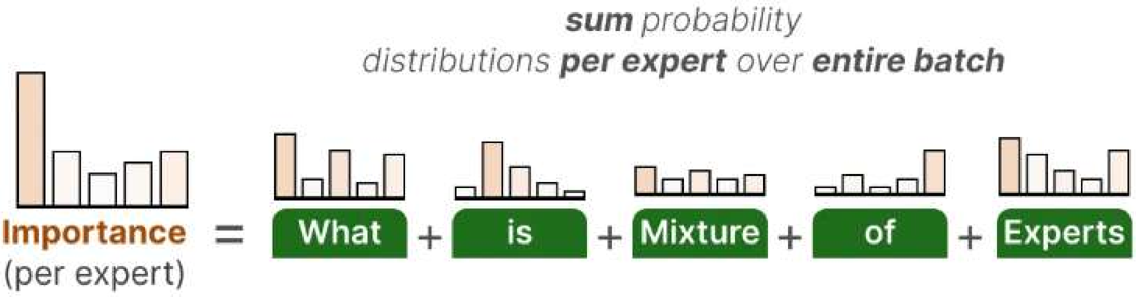
A major benefit is that it allows the experts' respective contributions to be weighed and integrated.

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:

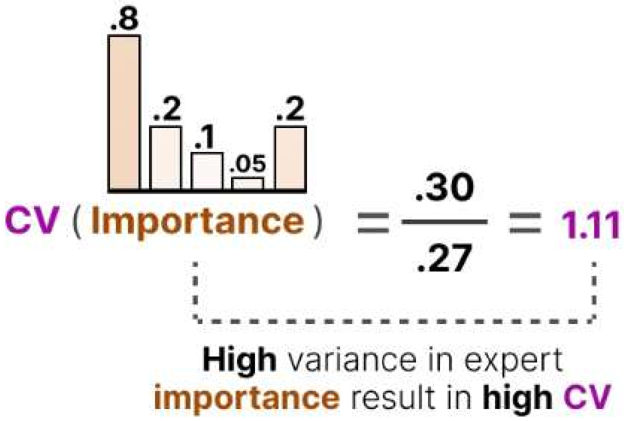


This gives us the *importance scores* per expert which represents how likely a given expert will be chosen regardless of the input.

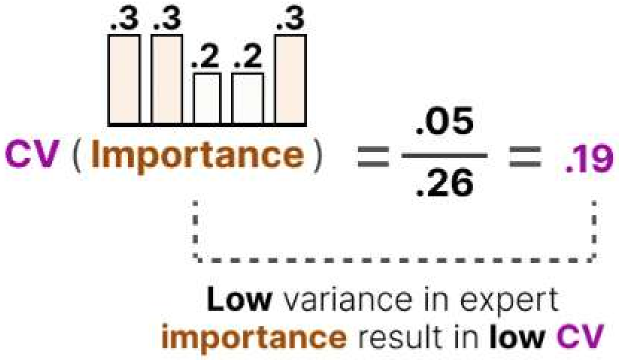
We can use this to calculate the *coefficient variation* (CV), which tells us how different the importance scores are between experts.

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

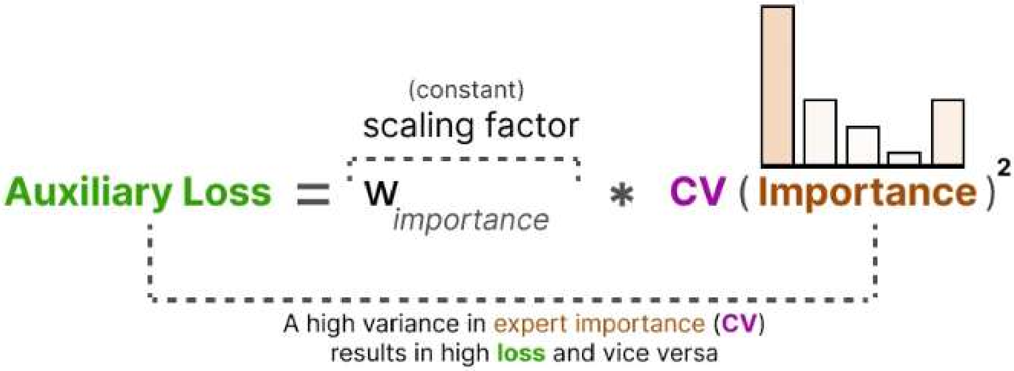
For instance, if there are a lot of differences in importance scores, the CV will be high:


.8
.2
.1
.05
.2
CV (Importance) = .30 / .27 = 1.11
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):


.3 .3
.2 .2
.3
CV (Importance) = .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*):

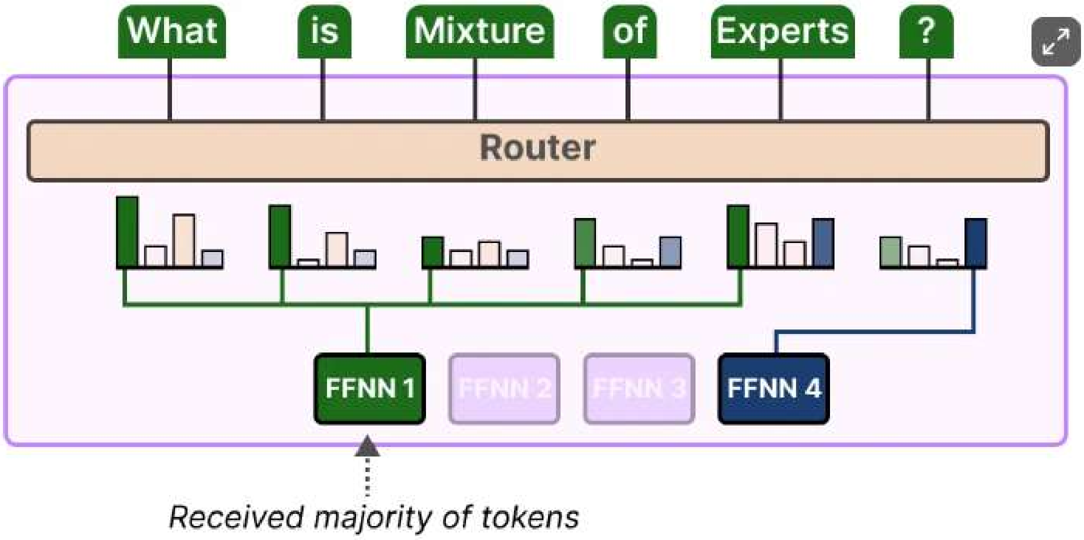

(constant)
scaling factor
Auxiliary Loss = w
importance
* CV (Importance)
2
A high variance in expert importance (CV)
results in high loss and vice versa

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

Expert Capacity

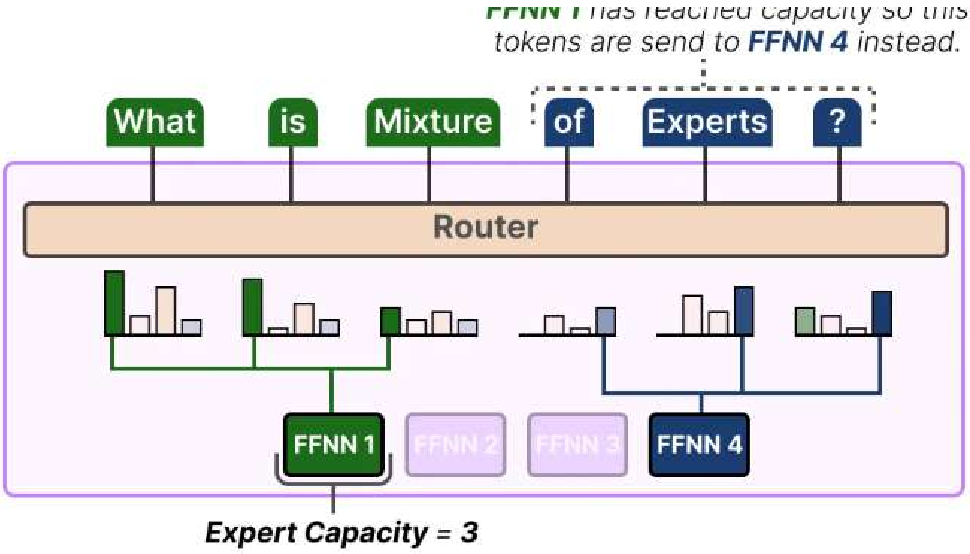
Imbalance is not just found in the experts that were chosen but also in the distributions of tokens that are sent to the expert.

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



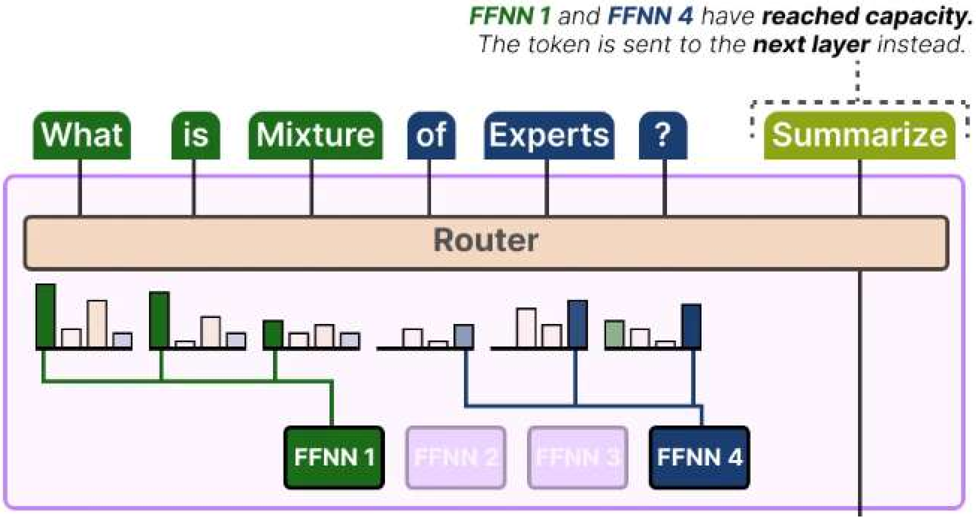
Here, it is not just about which experts are used but **how much** they are used.

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


FFNN 1 has reached capacity so this tokens are send to FFNN 4 instead.

Expert Capacity = 3

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*.

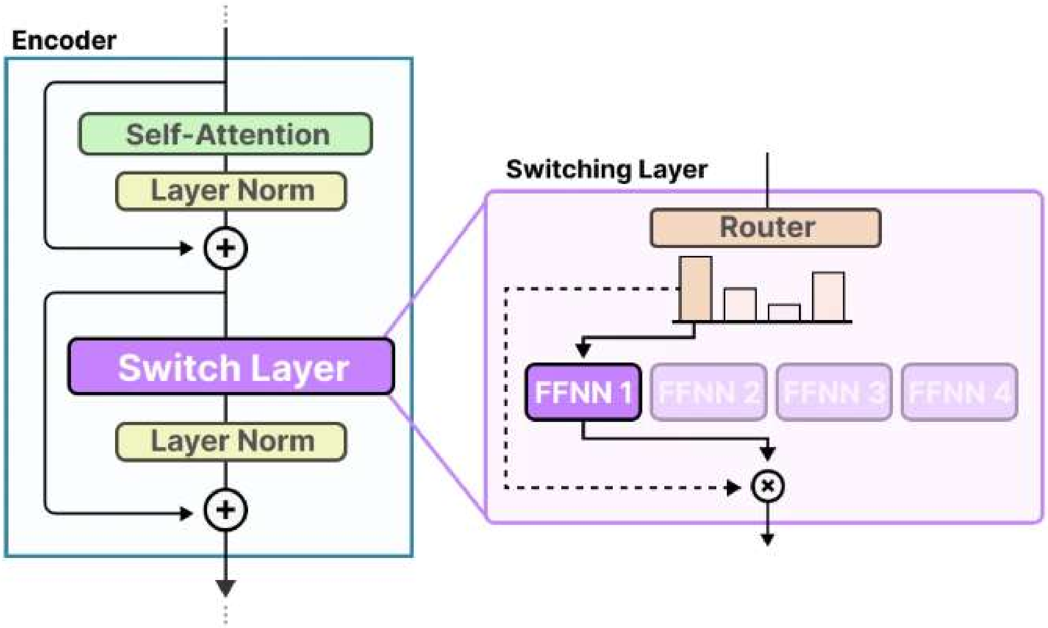

FFNN 1 and FFNN 4 have **reached capacity.** The token is sent to the **next layer** instead.

Simplifying MoE with the Switch Transformer

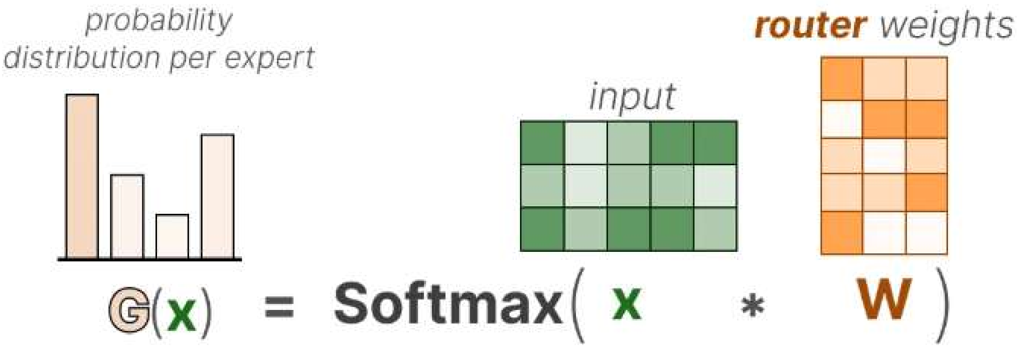
One of the first transformer-based MoE models that dealt with the training instability issues of MoE (such as load balancing) is the Switch Transformer.[5] It simplifies much of the architecture and training procedure while increasing training stability.

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).



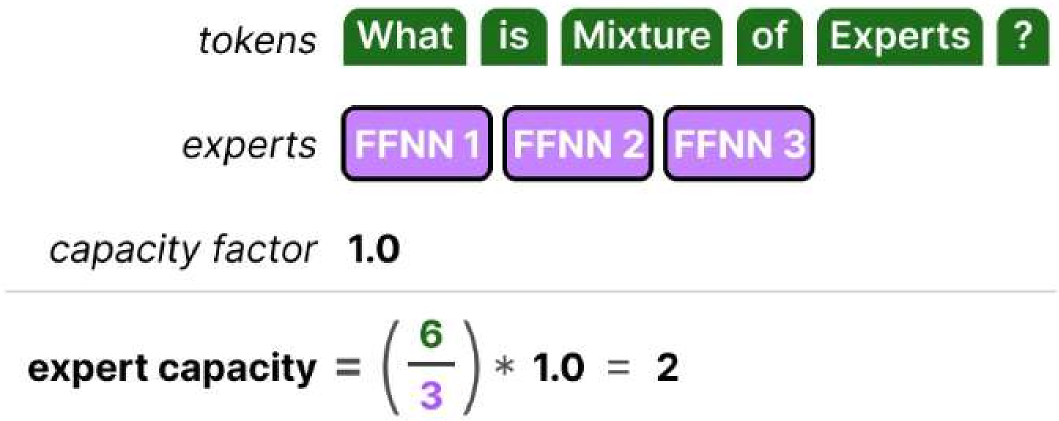
This architecture (*top-1 routing*) assumes that only 1 expert is needed for the router to learn how to route the input. This is in contrast to what we have seen previously where we assumed that tokens should be routed to multiple experts (*top-k routing*) to learn the routing behavior.

Capacity Factor

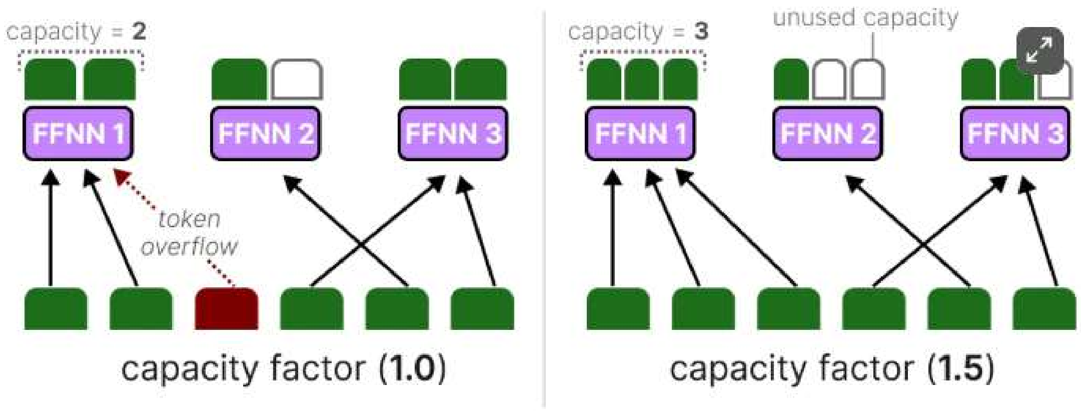
The capacity factor is an important value as it determines how many tokens an expert can process. The Switch Transformer extends upon this by introducing a **capacity factor** directly influencing the expert capacity.

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

The components of expert capacity are straightforward:



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

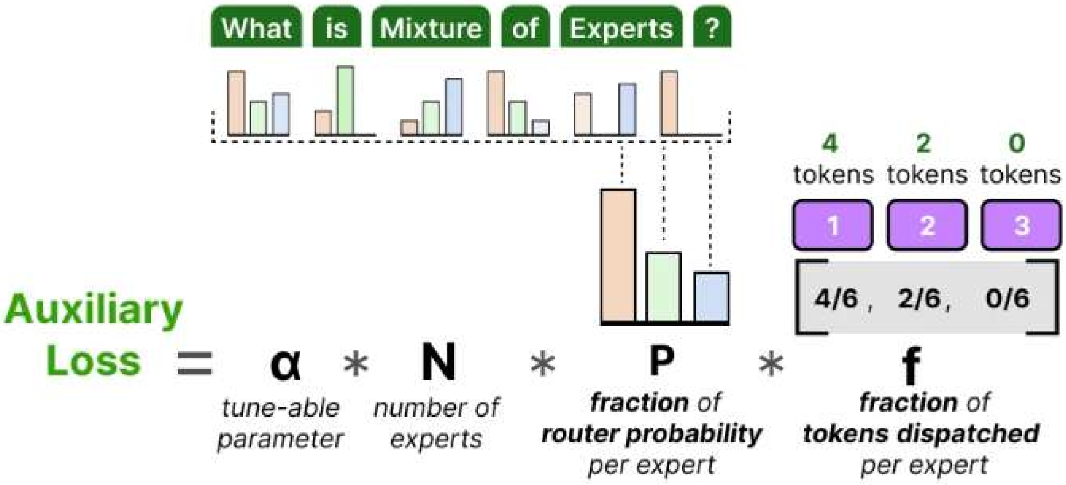


However, if the capacity factor is too large, we waste computing resources. In contrast, if the capacity factor is too small, the model performance will drop due to *token overflow*.

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:


What
is
Mixture
of
Experts
?
4 tokens
2 tokens
0 tokens
1
2
3
4/6 , 2/6 , 0/6
Auxiliary Loss = α * N * P * f
tune-able parameter
number of experts
fraction of router probability per expert
fraction of tokens dispatched 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**.

α 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.