Mixture of Experts (MoEs) (8x7B)

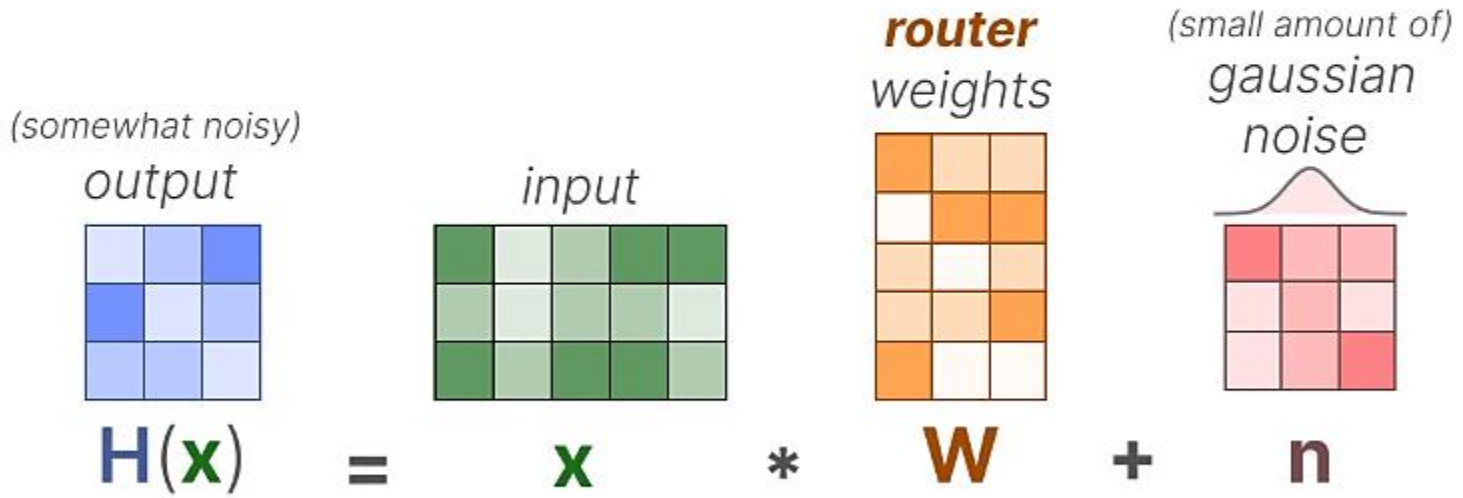
Load Balancing

In standard MoE, routers can develop a bias where only a few experts are frequently selected while others remain underutilized. This results in:

- **Bottlenecks**: Some experts become overloaded.
- **Wasted Capacity**: Some experts are rarely used.
- **Poor Generalization**: The model does not effectively utilize all experts.

KeepTopK

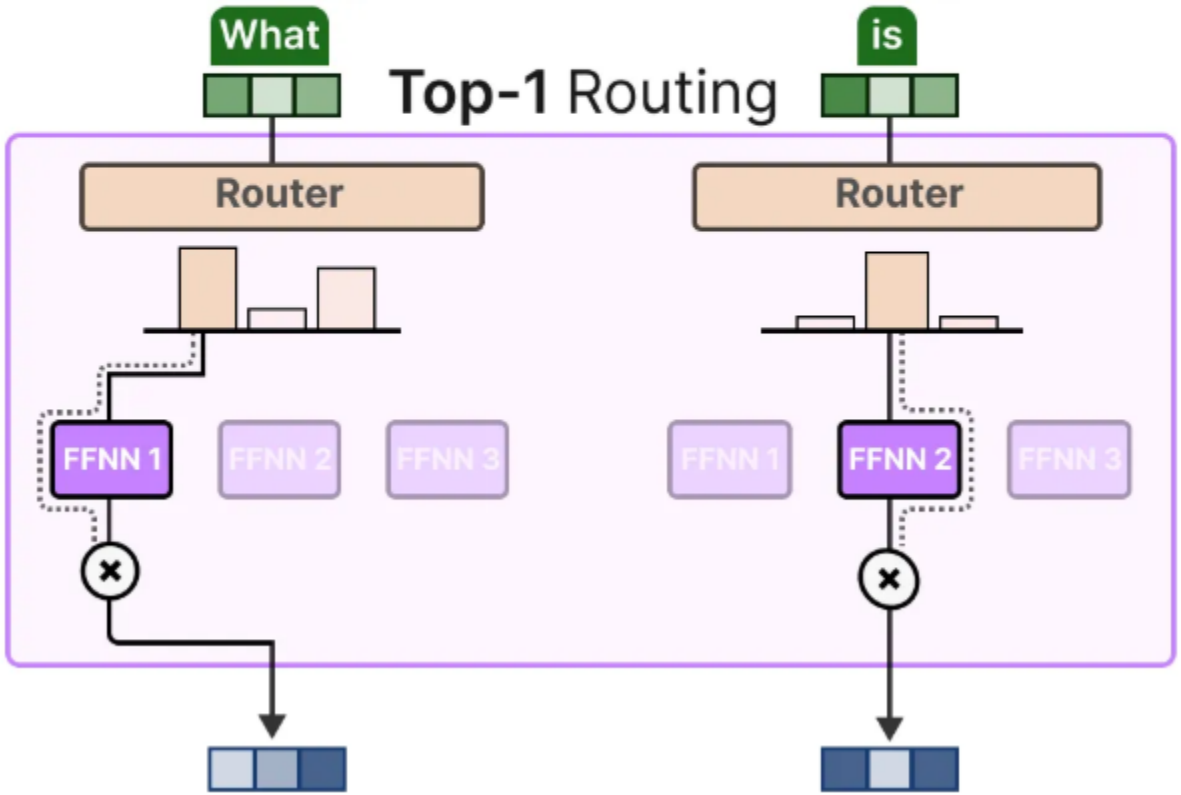
One method of load balancing the router is through a straightforward extension called KeepTopK [2]. By introducing trainable (gaussian) noise, we can prevent the same experts from always being picked:

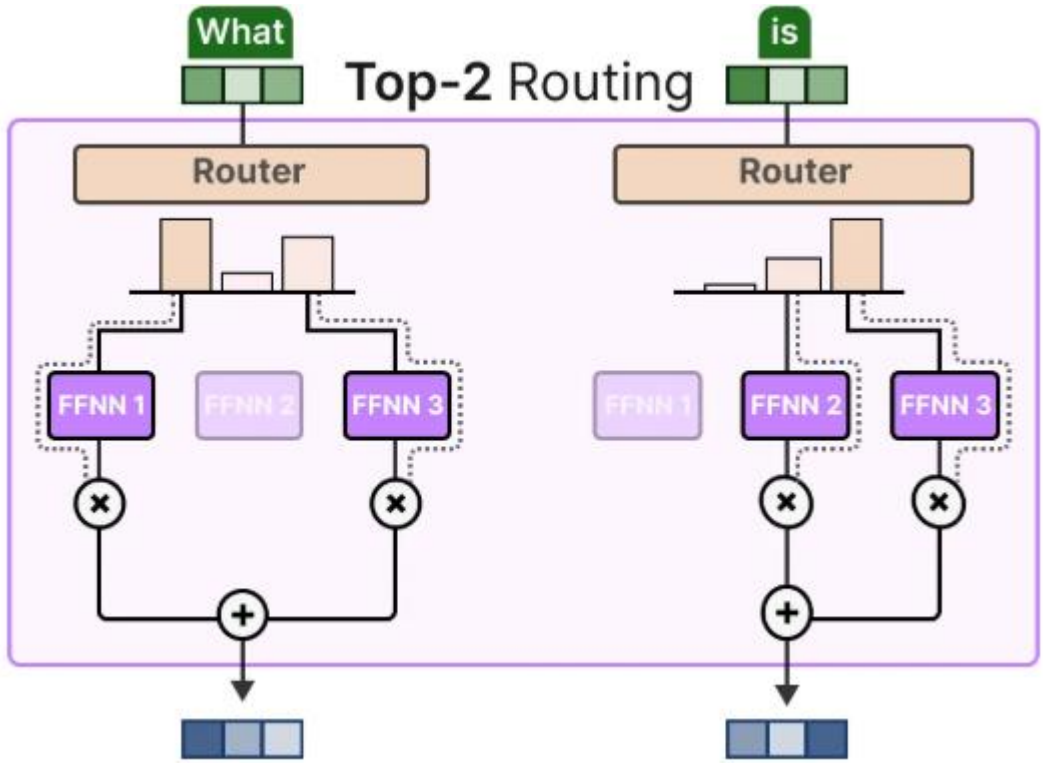


Then, all but the top k experts that you want activating (for example 2) will have their weights set to -∞:

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*):





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

Load Balancing Loss (Auxiliary Loss)

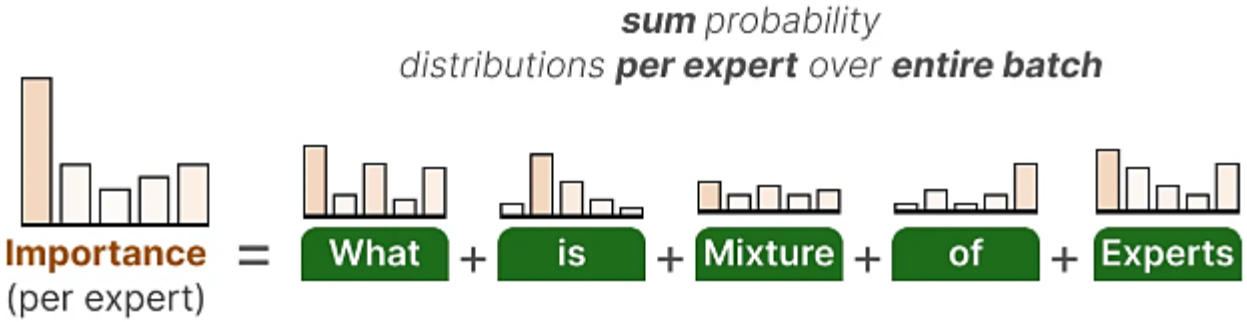
- To get a more even distribution of experts during training, the **auxiliary loss** (also called *load balancing loss*) was added to the network's regular loss.
- It adds a constraint that **forces experts** to have equal importance.
- **Step 1:** Compute Token-wise Outputs Using Expert Probabilities

Let's assume we have **three tokens** in a batch:

- **Token1** → Expert1 (0.6), Expert2 (0.4)
- **Token2** → Expert1 (0.7), Expert3 (0.3)
- **Token3** → Expert2 (0.5), Expert3 (0.5)

Each token's final output is computed as:

- **Token1**: $0.6 \times \text{Output}(E1) + 0.4 \times \text{Output}(E2)$
- **Token2**: $0.7 \times \text{Output}(E1) + 0.3 \times \text{Output}(E3)$
- **Token3**: $0.5 \times \text{Output}(E2) + 0.5 \times \text{Output}(E3)$



Step 1: Compute Token-wise Outputs Using Expert Probabilities

$$\text{Final Output}(t_i) = p_a \cdot \text{Output}(E_a) + p_b \cdot \text{Output}(E_b)$$

Step 2: Sum Over the Entire Batch to Compute Expert Importance

Importance(E1)=0.6+0.7=1.3

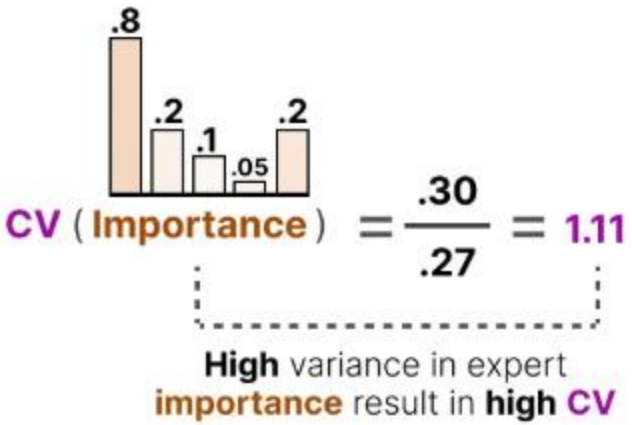
Importance(E2)=0.4+0.5=0.9

The **importance score** tells us how frequently an expert is chosen.

We can use this (from step 2) to calculate the *coefficient variation* (**CV**), which tells us how different the importance scores are between experts.

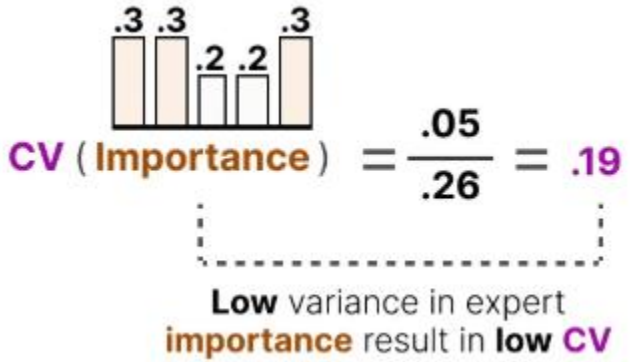
$$\text{Coefficient Variation (CV)} = \frac{\text{standard deviation } (\sigma)}{\text{mean } (\mu)}$$

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

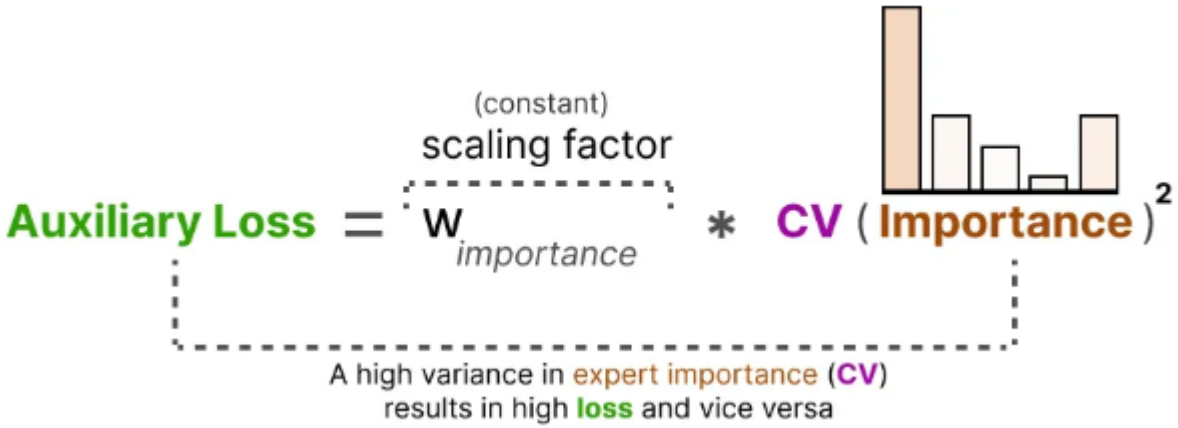


If CV is high → large loss → stronger penalty.
If CV is low → small loss → minimal penalty.

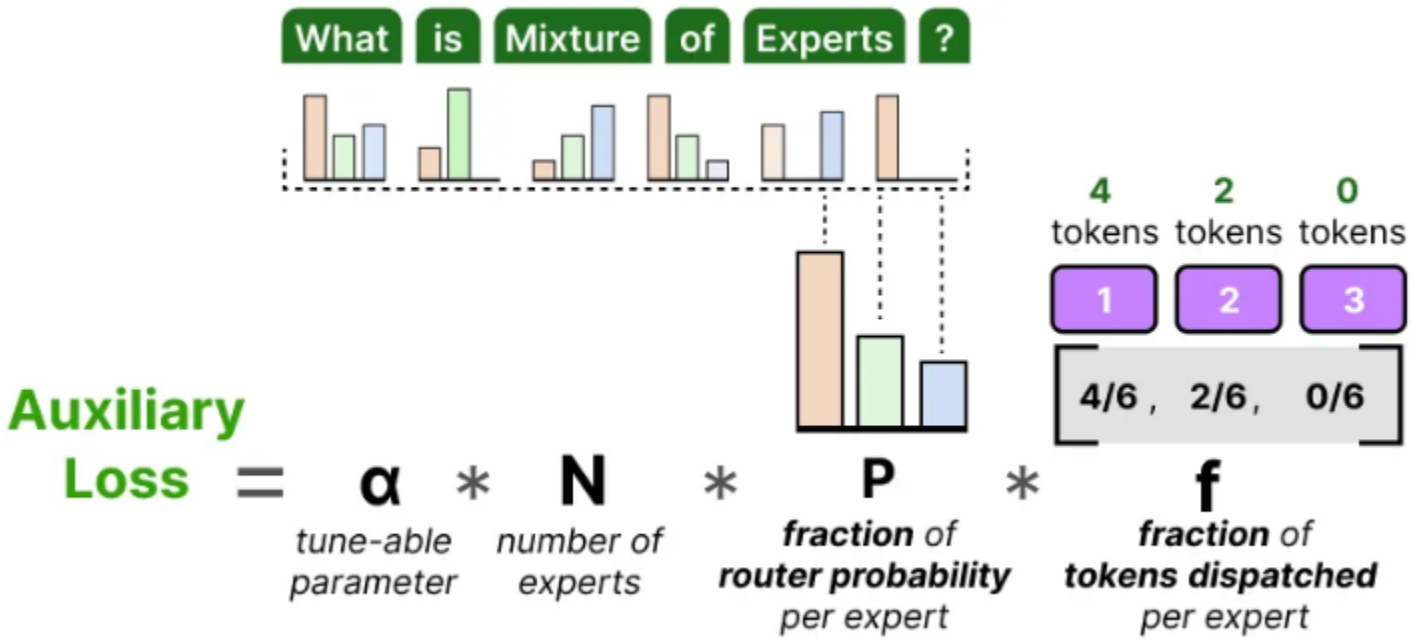
In contrast, if all experts have similar importance scores, the **CV** will be low (which is what we aim for):



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*):



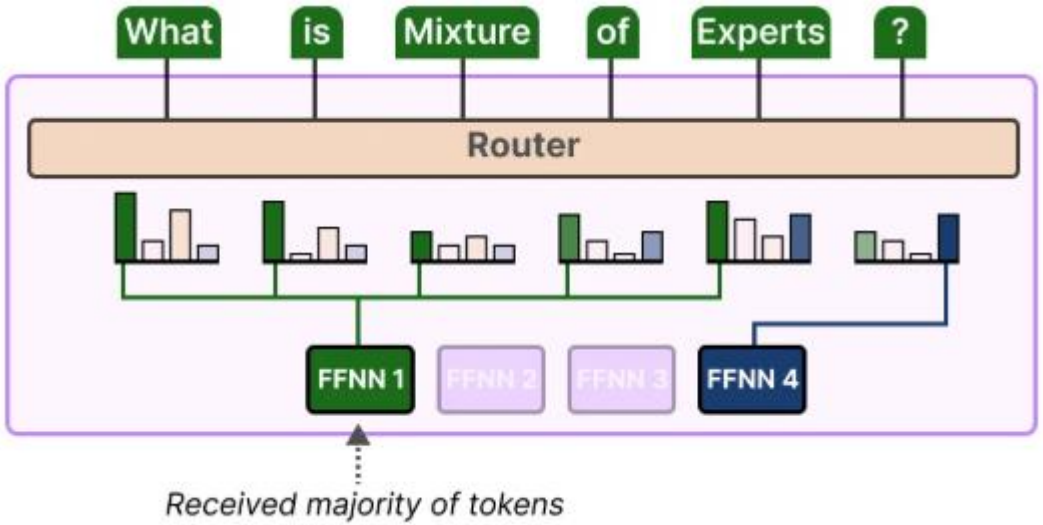
- Since the goal is to get a uniform routing of tokens across the **N** experts, we want vectors **P** and **f** to have values of 1/**N**.
- **α** is a hyperparameter that we can use to fine-tune the importance of this loss during training. Too high values will overtake the primary loss function and too low values will do little for load balancing.



1. **Initial Tokens:** When you input a prompt, the initial tokens of the prompt are indeed assigned to different experts by the gating network of the first MoE layer.
2. **Parallel Processing (within a layer):** The experts within a *single MoE layer* process their assigned tokens in parallel. This is where the parallelism of MoE models comes into play.
3. **Combining Expert Outputs (within a layer):** After the experts in a layer have processed their tokens, their outputs are combined using a **weighted average**, as determined by the gating network's probabilities. This weighted average produces a single output for *each token* at that MoE layer.
4. **Sequential Layers:** The combined output from the first MoE layer is then passed to the next layer in the model (which might be another MoE layer or a standard transformer layer). This process continues sequentially through the layers of the model.
5. **Autoregressive Text Generation:** The final output of the model is used to predict the next token in the sequence. This is done autoregressively, meaning one token at a time. The model generates one token, then uses that generated token as input to predict the next token, and so on.

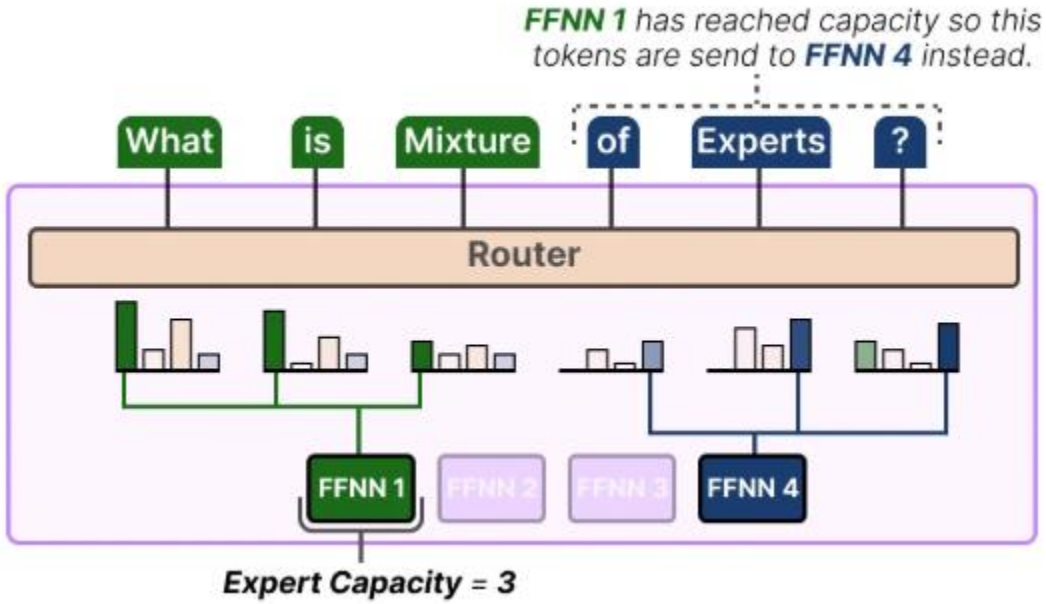
Expert Capacity

- Imbalance is not just found in the experts that were cho
  but also in the **distributions of tokens** that are sent to tl
  expert.
- For instance, if input tokens are **disproportionally** sent t
  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***.
- 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***.

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:

tokens What | is | Mixture | of | Experts | ?

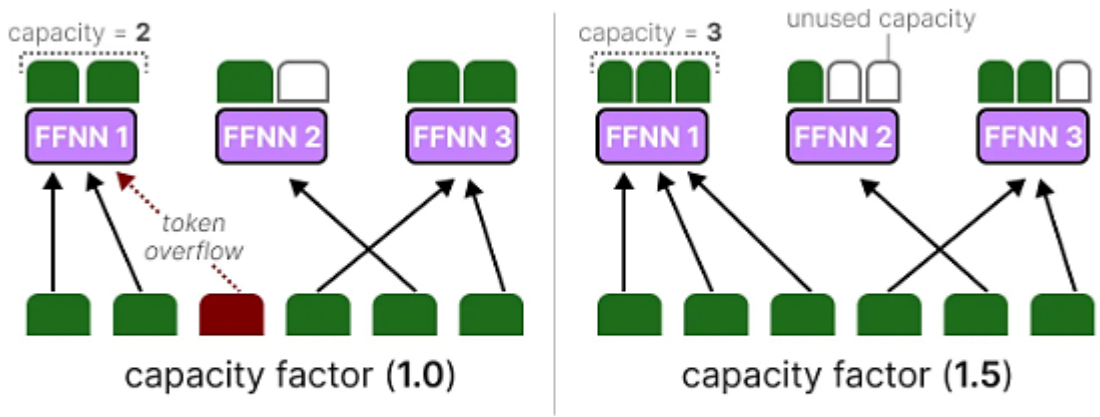
experts FFNN 1 | FFNN 2 | FFNN 3

capacity factor **1.0**

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

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

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*.

How does each expert get gradients?

- Each expert is a separate feed-forward network (FFN).
- The weighted sum of expert outputs contributes to the final model output, which is used for loss computation.
- Gradient flows backward from the loss function through the combined MoE output to each expert's parameters.
- Since each expert's output was weighted by the gating network's probability, only the selected experts get significant gradients.

- **How does the gating network receive gradients?**
- The gating network computes softmax probabilities, which determine how much each expert contributes.
- The weighted sum of **expert-outputs** flows into the loss function.
- Gradient flows back from the loss into the gating network's softmax scores because:
    - Softmax scores directly influence which experts are chosen.
    - If an incorrect expert was chosen, the softmax needs to be updated to select better experts.