← Back to Articles

# Mixture of Experts Explained



Published December 11, 2023

Update on GitHub



osanseviero
Omar Sanseviero


lewtun
Lewis Tunstall


philschmid
Philipp Schmid


smangrul
Sourab Mangrulkar


ybelkada
Younes Belkada


pcuenq
Pedro Cuenca


With the release of Mixtral 8x7B (announcement, model card), a class of transformer has become the hottest topic in the open AI community: Mixture of Experts, or MoEs for short. In this blog post, we take a look at the building blocks of MoEs, how they're trained, and the tradeoffs to consider when serving them for inference.

Let's dive in!

## Table of Contents

## TL;DR

MoEs:

- Are **pretrained much faster** vs. dense models
- Have **faster inference** compared to a model with the same number of parameters
- Require **high VRAM** as all experts are loaded in memory
- Face many **challenges in fine-tuning**, but recent work with MoE **instruction-tuning is promising**

Let's dive in!

## What is a Mixture of Experts (MoE)?

The scale of a model is one of the most important axes for better model quality. Given a fixed computing budget, training a larger model for fewer steps is better than training a smaller model for more steps.

Mixture of Experts enable models to be pretrained with far less compute, which means you can dramatically scale up the model or dataset size with the same compute budget as a dense model. In particular, a MoE model should achieve the same quality as its dense

counterpart much faster during pretraining.

So, what exactly is a MoE? In the context of transformer models, a MoE consists of two main elements:

- **Sparse MoE layers** are used instead of dense feed-forward network (FFN) layers. MoE layers have a certain number of “experts” (e.g. 8), where each expert is a neural network. In practice, the experts are FFNs, but they can also be more complex networks or even a MoE itself, leading to hierarchical MoEs!
- A **gate network or router**, that determines which tokens are sent to which expert. For example, in the image below, the token “More” is sent to the second expert, and the token "Parameters” is sent to the first network. As we’ll explore later, we can send a token to more than one expert. How to route a token to an expert is one of the big decisions when working with MoEs - the router is composed of learned parameters and is pretrained at the same time as the rest of the network.

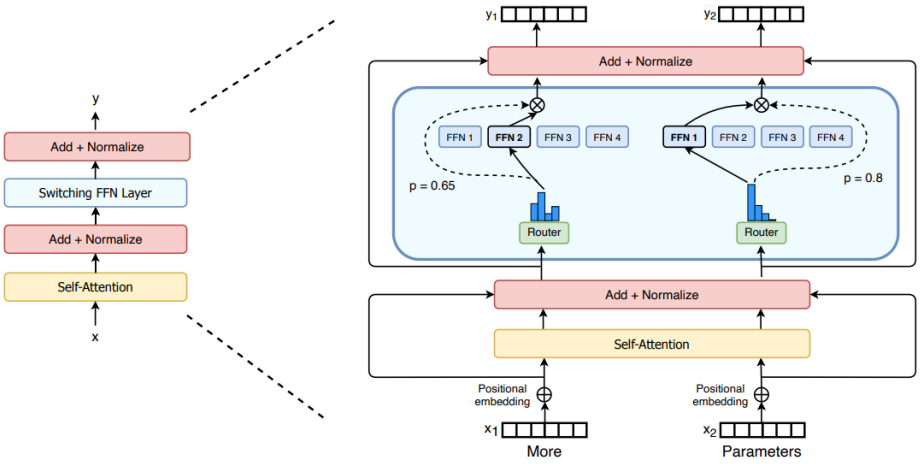



Figure 2: Illustration of a Switch Transformer encoder block. We replace the dense feed forward network (FFN) layer present in the Transformer with a sparse Switch FFN layer (light blue). The layer operates independently on the tokens in the sequence. We diagram two tokens ($x_1$ = “More” and $x_2$ = “Parameters” below) being routed (solid lines) across four FFN experts, where the router independently routes each token. The switch FFN layer returns the output of the selected FFN multiplied by the router gate value (dotted-line).

MoE layer from the [Switch Transformers paper](https://arxiv.org/abs/2101.03961)

So, to recap, in MoEs we replace every FFN layer of the transformer model with an MoE layer, which is composed of a gate network and a certain number of experts.

Although MoEs provide benefits like efficient pretraining and faster inference compared

to dense models, they also come with challenges:

- **Training:** MoEs enable significantly more compute-efficient pretraining, but they've historically struggled to generalize during fine-tuning, leading to overfitting.
- **Inference:** Although a MoE might have many parameters, only some of them are used during inference. This leads to much faster inference compared to a dense model with the same number of parameters. However, all parameters need to be loaded in RAM, so memory requirements are high. For example, given a MoE like Mixtral 8x7B, we'll need to have enough VRAM to hold a dense 47B parameter model. Why 47B parameters and not 8 x 7B = 56B? That's because in MoE models, only the FFN layers are treated as individual experts, and the rest of the model parameters are shared. At the same time, assuming just two experts are being used per token, the inference speed (FLOPs) is like using a 12B model (as opposed to a 14B model), because it computes 2x7B matrix multiplications, but with some layers shared (more on this soon).

Now that we have a rough idea of what a MoE is, let's take a look at the research developments that led to their invention.

## A Brief History of MoEs

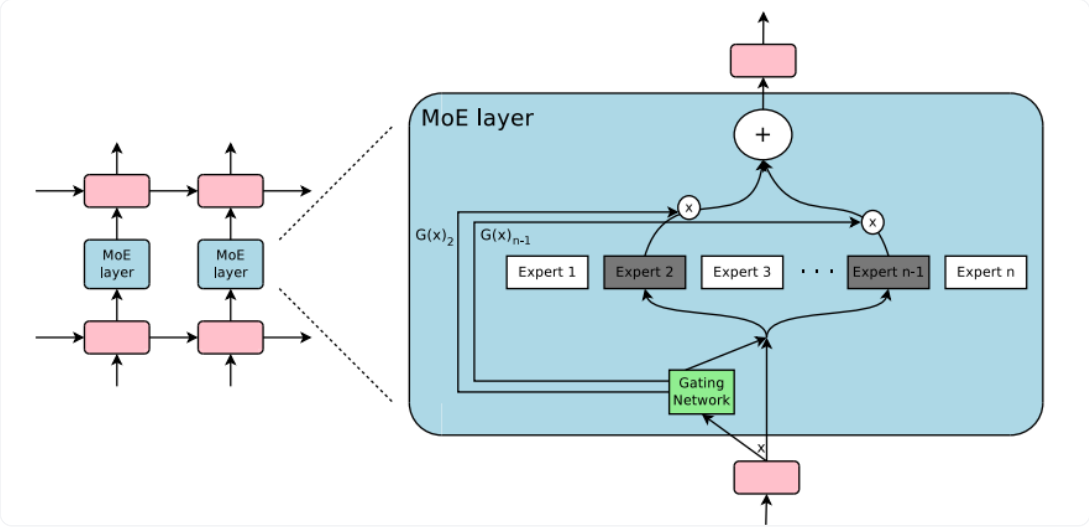
The roots of MoEs come from the 1991 paper Adaptive Mixture of Local Experts. The idea, akin to ensemble methods, was to have a supervised procedure for a system composed of separate networks, each handling a different subset of the training cases. Each separate network, or expert, specializes in a different region of the input space. How is the expert chosen? A gating network determines the weights for each expert. During training, both the expert and the gating are trained.

Between 2010-2015, two different research areas contributed to later MoE advancement:

- **Experts as components**: In the traditional MoE setup, the whole system comprises a gating network and multiple experts. MoEs as the whole model have been explored in SVMs, Gaussian Processes, and other methods. The work by Eigen, Ranzato, and Ilya explored MoEs as components of deeper networks. This allows having MoEs as layers in a multilayer network, making it possible for the model to be both large and efficient simultaneously.

- **Conditional Computation**: Traditional networks process all input data through every layer. In this period, Yoshua Bengio researched approaches to dynamically activate or deactivate components based on the input token.

These works led to exploring a mixture of experts in the context of NLP. Concretely, Shazeer et al. (2017, with "et al." including Geoffrey Hinton and Jeff Dean, Google's Chuck Norris) scaled this idea to a 137B LSTM (the de-facto NLP architecture back then, created by Schmidhuber) by introducing sparsity, allowing to keep very fast inference even at high scale. This work focused on translation but faced many challenges, such as high communication costs and training instabilities.




MoE layer from the Outrageously Large Neural Network paper

MoEs have allowed training multi-trillion parameter models, such as the open-sourced 1.6T parameters Switch Transformers, among others. MoEs have also been explored in Computer Vision, but this blog post will focus on the NLP domain.

## What is Sparsity?

Sparsity uses the idea of conditional computation. While in dense models all the parameters are used for all the inputs, sparsity allows us to only run some parts of the whole system.

Let's dive deeper into Shazeer's exploration of MoEs for translation. The idea of conditional computation (parts of the network are active on a per-example basis) allows one to scale the size of the model without increasing the computation, and hence, this led to thousands of experts being used in each MoE layer.

This setup introduces some challenges. For example, although large batch sizes are usually better for performance, batch sizes in MOEs are effectively reduced as data flows through the active experts. For example, if our batched input consists of 10 tokens, **five tokens might end in one expert, and the other five tokens might end in five different experts, leading to uneven batch sizes and underutilization**. The Making MoEs go brrr section below will discuss other challenges and solutions.

How can we solve this? A learned gating network (G) decides which experts (E) to send a part of the input:

$$y = \sum_{i=1}^{n} G(x)_i E_i(x)$$

In this setup, all experts are run for all inputs - it's a weighted multiplication. But, what happens if G is 0? If that's the case, there's no need to compute the respective expert operations and hence we save compute. What's a typical gating function? In the most traditional setup, we just use a simple network with a softmax function. The network will learn which expert to send the input.

$$G_\sigma(x) = \text{Softmax}(x \cdot W_g)$$

Shazeer's work also explored other gating mechanisms, such as Noisy Top-k Gating. This gating approach introduces some (tunable) noise and then keeps the top k values. That is:

1. We add some noise

$$H(x)_i = (x \cdot W_{\text{g}})_i + \text{StandardNormal}() \cdot \text{Softplus}((x \cdot W_{\text{noise}})_i)$$

2. We only pick the top k

$$\text{KeepTopK}(v, k)_i = \begin{cases} v_i & \text{if } v_i \text{ is in the top } k \text{ elements of } v, \\ -\infty & \text{otherwise.} \end{cases}$$

3. We apply the softmax.

$$G(x) = \text{Softmax}(\text{KeepTopK}(H(x), k))$$

This sparsity introduces some interesting properties. By using a low enough k (e.g. one or two), we can train and run inference much faster than if many experts were activated. Why not just select the top expert? The initial conjecture was that routing to more than

one expert was needed to have the gate learn how to route to different experts, so at least two experts had to be picked. The Switch Transformers section revisits this decision.

Why do we add noise? That's for load balancing!

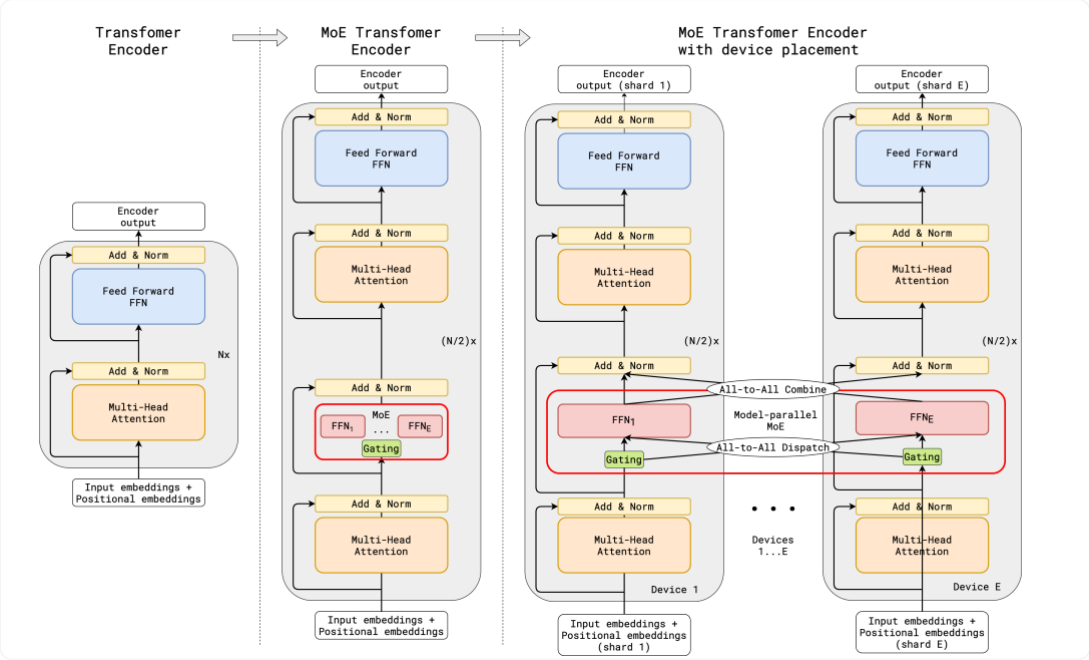
## Load balancing tokens for MoEs

As discussed before, if all our tokens are sent to just a few popular experts, that will make training inefficient. In a normal MoE training, the gating network converges to mostly activate the same few experts. This self-reinforces as favored experts are trained quicker and hence selected more. To mitigate this, an **auxiliary loss** is added to encourage giving all experts equal importance. This loss ensures that all experts receive a roughly equal number of training examples. The following sections will also explore the concept of expert capacity, which introduces a threshold of how many tokens can be processed by an expert. In `transformers`, the auxiliary loss is exposed via the `aux_loss` parameter.

## MoEs and Transformers

Transformers are a very clear case that scaling up the number of parameters improves the performance, so it's not surprising that Google explored this with GShard, which explores scaling up transformers beyond 600 billion parameters.

GShard replaces every other FFN layer with an MoE layer using top-2 gating in both the encoder and the decoder. The next image shows how this looks like for the encoder part. This setup is quite beneficial for large-scale computing: when we scale to multiple devices, the MoE layer is shared across devices while all the other layers are replicated. This is further discussed in the "Making MoEs go brrr" section.




MoE Transformer Encoder from the GShard Paper

To maintain a balanced load and efficiency at scale, the GShard authors introduced a couple of changes in addition to an auxiliary loss similar to the one discussed in the previous section:

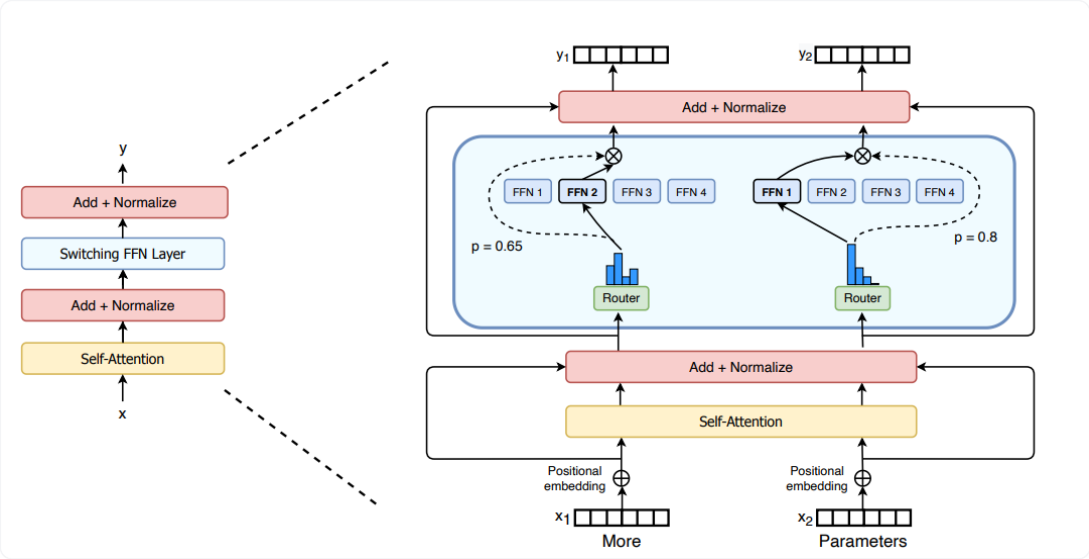
- **Random routing**: in a top-2 setup, we always pick the top expert, but the second expert is picked with probability proportional to its weight.
- **Expert capacity**: we can set a threshold of how many tokens can be processed by one expert. If both experts are at capacity, the token is considered overflowed, and it's sent to the next layer via residual connections (or dropped entirely in other projects). This concept will become one of the most important concepts for MoEs. Why is expert capacity needed? Since all tensor shapes are statically determined at compilation time, but we cannot know how many tokens will go to each expert ahead of time, we need to fix the capacity factor.

The GShard paper has contributions by expressing parallel computation patterns that work well for MoEs, but discussing that is outside the scope of this blog post.

**Note:** when we run inference, only some experts will be triggered. At the same time, there are shared computations, such as self-attention, which is applied for all tokens. That's why when we talk of a 47B model of 8 experts, we can run with the compute of a 12B dense model. If we use top-2, 14B parameters would be used. But given that the attention operations are shared (among others), the actual number of used parameters is 12B.

# Switch Transformers

Although MoEs showed a lot of promise, they struggle with training and fine-tuning instabilities. Switch Transformers is a very exciting work that deep dives into these topics. The authors even released a 1.6 trillion parameters MoE on Hugging Face with 2048 experts, which you can run with transformers. Switch Transformers achieved a 4x pre-train speed-up over T5-XXL.




Switch Transformer Layer of the Switch Transformer paper

Just as in GShard, the authors replaced the FFN layers with a MoE layer. The Switch Transformers paper proposes a Switch Transformer layer that receives two inputs (two different tokens) and has four experts.

Contrary to the initial idea of using at least two experts, Switch Transformers uses a simplified single-expert strategy. The effects of this approach are:

- The router computation is reduced
- The batch size of each expert can be at least halved
- Communication costs are reduced
- Quality is preserved

Switch Transformers also explores the concept of expert capacity.

$$\text{Expert Capacity} = \left(\frac{\text{tokens per batch}}{\text{number of experts}}\right) \times \text{capacity factor}$$

The capacity suggested above evenly divides the number of tokens in the batch across the number of experts. If we use a capacity factor greater than 1, we provide a buffer for when tokens are not perfectly balanced. Increasing the capacity will lead to more expensive inter-device communication, so it's a trade-off to keep in mind. In particular, Switch Transformers perform well at low capacity factors (1-1.25)

Switch Transformer authors also revisit and simplify the load balancing loss mentioned in the sections. For each Switch layer, the auxiliary loss is added to the total model loss during training. This loss encourages uniform routing and can be weighted using a hyperparameter.

The authors also experiment with selective precision, such as training the experts with `bfloat16` while using full precision for the rest of the computations. Lower precision reduces communication costs between processors, computation costs, and memory for storing tensors. The initial experiments, in which both the experts and the gate networks were trained in `bfloat16`, yielded more unstable training. This was, in particular, due to the router computation: as the router has an exponentiation function, having higher precision is important. To mitigate the instabilities, full precision was used for the routing as well.

| Model (precision) | Quality (Neg. Log Perp.) (↑) | Speed (Examples/sec) (↑) |
|---|---|---|
| Switch-Base (float32) | -1.718 | 1160 |
| Switch-Base (bfloat16) | -3.780 [*diverged*] | **1390** |
| Switch-Base (Selective precision) | **-1.716** | 1390 |

Using selective precision does not degrade quality and enables faster models

This notebook showcases fine-tuning Switch Transformers for summarization, but we suggest first reviewing the fine-tuning section.

Switch Transformers uses an encoder-decoder setup in which they did a MoE counterpart of T5. The GLaM paper explores pushing up the scale of these models by training a model matching GPT-3 quality using 1/3 of the energy (yes, thanks to the lower amount of computing needed to train a MoE, they can reduce the carbon footprint by up to an order of magnitude). The authors focused on decoder-only models and few-shot and one-shot evaluation rather than fine-tuning. They used Top-2 routing and much larger capacity factors. In addition, they explored the capacity factor as a metric one can change during training and evaluation depending on how much computing one wants to use.

## Stabilizing training with router Z-loss

The balancing loss previously discussed can lead to instability issues. We can use many methods to stabilize sparse models at the expense of quality. For example, introducing dropout improves stability but leads to loss of model quality. On the other hand, adding more multiplicative components improves quality but decreases stability.

Router z-loss, introduced in ST-MoE, significantly improves training stability without quality degradation by penalizing large logits entering the gating network. Since this loss encourages absolute magnitude of values to be smaller, roundoff errors are reduced, which can be quite impactful for exponential functions such as the gating. We recommend reviewing the paper for details.

## What does an expert learn?

The ST-MoE authors observed that encoder experts specialize in a group of tokens or shallow concepts. For example, we might end with a punctuation expert, a proper noun expert, etc. On the other hand, the decoder experts have less specialization. The authors also trained in a multilingual setup. Although one could imagine each expert specializing in a language, the opposite happens: due to token routing and load balancing, there is no single expert specialized in any given language.

| Expert specialization | Expert position | Routed tokens |
|---|---|---|
| **Sentinel tokens** | Layer 1 | been <extra_id_4><extra_id_7>floral to<br><extra_id_10><extra_id_12><extra_id_15><br><extra_id_17><extra_id_18><extra_id_19>... |
| | Layer 4 | <extra_id_0><extra_id_1><extra_id_2><br><extra_id_4><extra_id_6><extra_id_7><br><extra_id_12><extra_id_13><extra_id_14>... |
| | Layer 6 | <extra_id_0><extra_id_4><extra_id_5><br><extra_id_6><extra_id_7><extra_id_14><br><extra_id_16><extra_id_17><extra_id_18>... |
| **Punctuation** | Layer 2 | , , , , , , , , , - , , , , , ). ) |
| | Layer 6 | , , , , , : . : , & , & & ? & - , , ? , , , . <extra_id_27> |
| **Conjunctions and articles** | Layer 3 | The the the the the the the the the The the the<br>the the the The the the the |
| | Layer 6 | a and and and and and and and or and a and .<br>the the if ? a designed does been is not |
| **Verbs** | Layer 1 | died falling identified fell closed left posted lost felt<br>left said read miss place struggling falling signed died<br>falling designed based disagree submitted develop |
| **Visual descriptions**<br>*color, spatial position* | Layer 0 | her over her know dark upper dark outer<br>center upper blue inner yellow raw mama<br>bright bright over open your dark blue |
| **Proper names** | Layer 1 | A Mart Gr Mart Kent Med Cor Tri Ca Mart<br>R Mart Lorraine Colin Ken Sam Ken Gr Angel A<br>Dou Now Ga GT Q Ga C Ko C Ko Ga G |
| **Counting and numbers**<br>*written and numerical forms* | Layer 1 | after 37 19. 6. 27 I I Seven 25 4, 54 I two dead we<br>Some 2012 who we few lower each |

Table from the ST-MoE paper showing which token groups were sent to which expert.

## How does scaling the number of experts impact pretraining?

More experts lead to improved sample efficiency and faster speedup, but these are diminishing gains (especially after 256 or 512), and more VRAM will be needed for inference. The properties studied in Switch Transformers at large scale were consistent at small scale, even with 2, 4, or 8 experts per layer.

## Fine-tuning MoEs

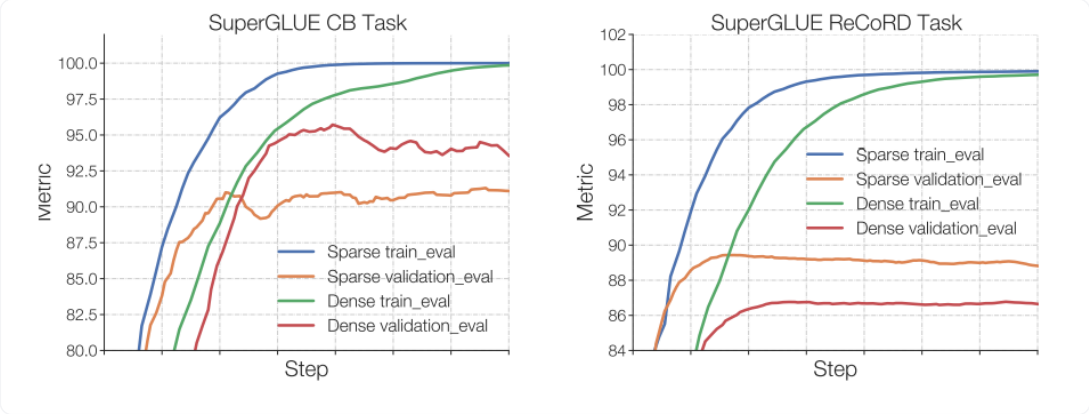
> *"Mixtral is supported with version 4.36.0 of transformers. You can install it with* `pip install transformers==4.36.0 --upgrade`*"*

The overfitting dynamics are very different between dense and sparse models. Sparse

models are more prone to overfitting, so we can explore higher regularization (e.g. dropout) within the experts themselves (e.g. we can have one dropout rate for the dense layers and another, higher, dropout for the sparse layers).

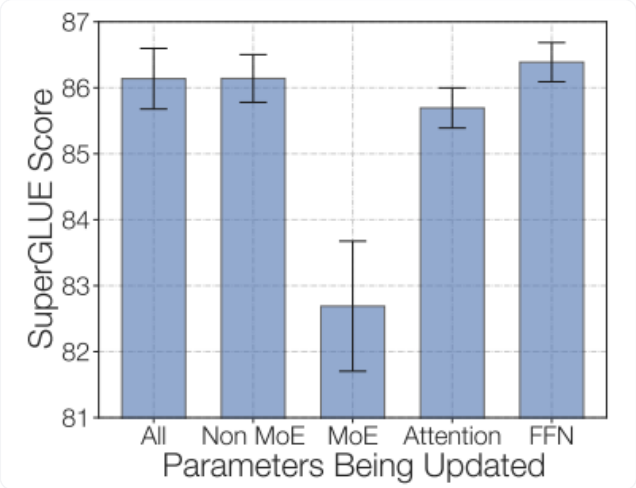
One question is whether to use the auxiliary loss for fine-tuning. The ST-MoE authors experimented with turning off the auxiliary loss, and the quality was not significantly impacted, even when up to 11% of the tokens were dropped. Token dropping might be a form of regularization that helps prevent overfitting.

Switch Transformers observed that at a fixed pretrain perplexity, the sparse model does worse than the dense counterpart in downstream tasks, especially on reasoning-heavy tasks such as SuperGLUE. On the other hand, for knowledge-heavy tasks such as TriviaQA, the sparse model performs disproportionately well. The authors also observed that a fewer number of experts helped at fine-tuning. Another observation that confirmed the generalization issue is that the model did worse in smaller tasks but did well in larger tasks.



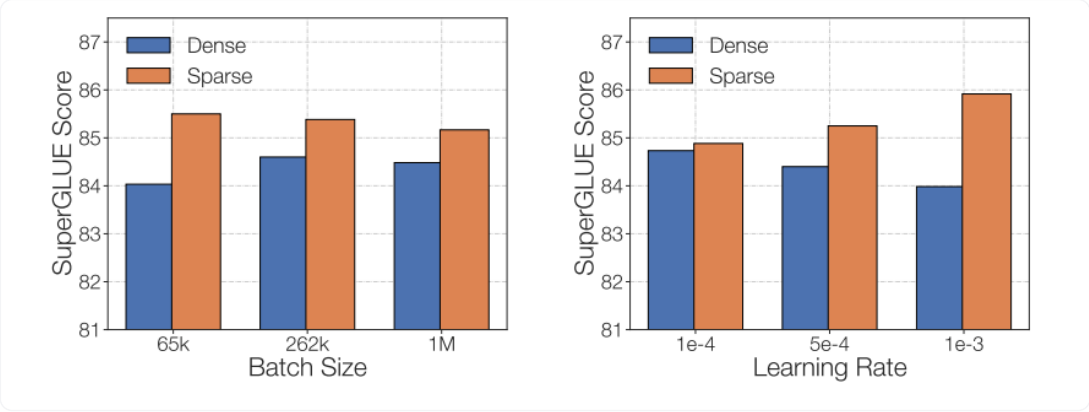

In the small task (left), we can see clear overfitting as the sparse model does much worse in the validation set. In the larger task (right), the MoE performs well. This image is from the ST-MoE paper.

One could experiment with freezing all non-expert weights. That is, we'll only update the MoE layers. This leads to a huge performance drop. We could try the opposite: freezing only the parameters in MoE layers, which worked almost as well as updating all parameters. This can help speed up and reduce memory for fine-tuning. This can be somewhat counter-intuitive as 80% of the parameters are in the MoE layers (in the ST-MoE project). Their hypothesis for that architecture is that, as expert layers only occur every 1/4 layers, and each token sees at most two experts per layer, updating the MoE parameters affects much fewer layers than updating other parameters.




By only freezing the MoE layers, we can speed up the training while preserving the quality. This image is from the ST-MoE paper.

One last part to consider when fine-tuning sparse MoEs is that they have different fine-tuning hyperparameter setups - e.g., sparse models tend to benefit more from smaller batch sizes and higher learning rates.



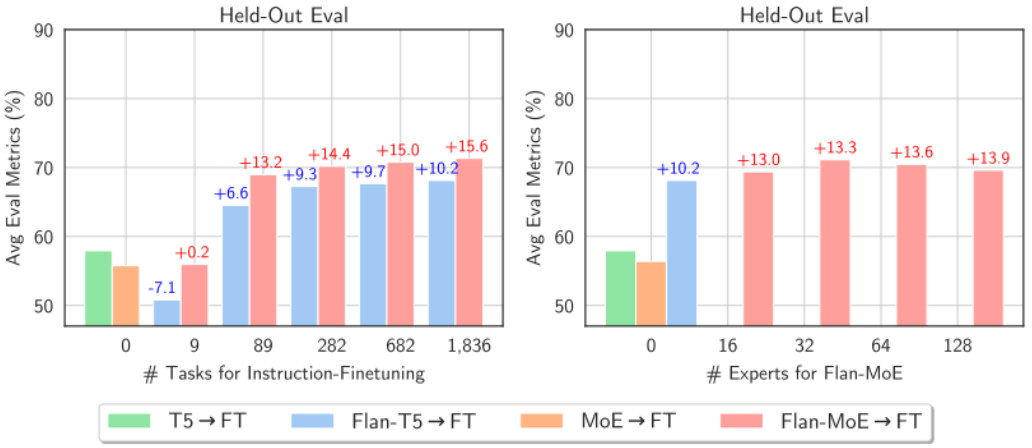

Sparse models fine-tuned quality improves with higher learning rates and smaller batch sizes. This image is from the ST-MoE paper.

At this point, you might be a bit sad that people have struggled to fine-tune MoEs. Excitingly, a recent paper, MoEs Meets Instruction Tuning (July 2023), performs experiments doing:

- Single task fine-tuning
- Multi-task instruction-tuning
- Multi-task instruction-tuning followed by single-task fine-tuning

When the authors fine-tuned the MoE and the T5 equivalent, the T5 equivalent was better. When the authors fine-tuned the Flan T5 (T5 instruct equivalent) MoE, the MoE

performed significantly better. Not only this, the improvement of the Flan-MoE over the MoE was larger than Flan T5 over T5, indicating that MoEs might benefit much more from instruction tuning than dense models. MoEs benefit more from a higher number of tasks. Unlike the previous discussion suggesting to turn off the auxiliary loss function, the loss actually prevents overfitting.




Sparse models benefit more from instruct-tuning compared to dense models. This image is from the MoEs Meets Instruction Tuning paper

## When to use sparse MoEs vs dense models?

Experts are useful for high throughput scenarios with many machines. Given a fixed compute budget for pretraining, a sparse model will be more optimal. For low throughput scenarios with little VRAM, a dense model will be better.

**Note:** one cannot directly compare the number of parameters between sparse and dense models, as both represent significantly different things.

## Making MoEs go brrr

The initial MoE work presented MoE layers as a branching setup, leading to slow computation as GPUs are not designed for it and leading to network bandwidth becoming a bottleneck as the devices need to send info to others. This section will discuss some existing work to make pretraining and inference with these models more practical. MoEs go brrrrr.

## Parallelism

Let's do a brief review of parallelism:

- **Data parallelism:** the same weights are replicated across all cores, and the data is partitioned across cores.
- **Model parallelism:** the model is partitioned across cores, and the data is replicated across cores.
- **Model and data parallelism:** we can partition the model and the data across cores. Note that different cores process different batches of data.
- **Expert parallelism**: experts are placed on different workers. If combined with data parallelism, each core has a different expert and the data is partitioned across all cores

With expert parallelism, experts are placed on different workers, and each worker takes a different batch of training samples. For non-MoE layers, expert parallelism behaves the same as data parallelism. For MoE layers, tokens in the sequence are sent to workers where the desired experts reside.

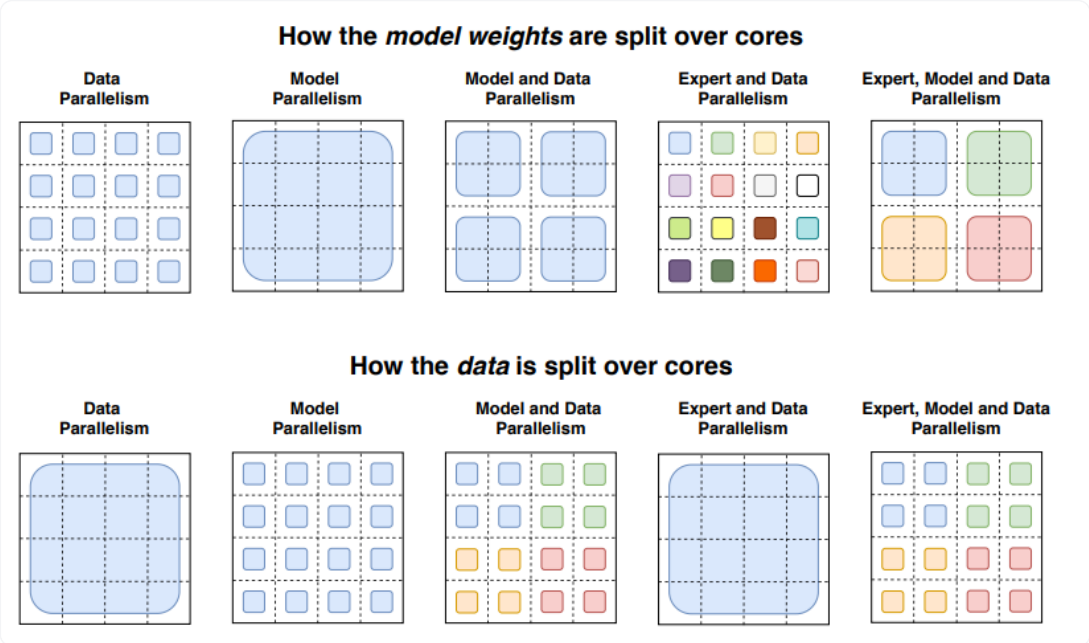



Illustration from the Switch Transformers paper showing how data and models are split over cores with different parallelism techniques.

## Capacity Factor and communication costs

Increasing the capacity factor (CF) increases the quality but increases communication costs and memory of activations. If all-to-all communications are slow, using a smaller capacity factor is better. A good starting point is using top-2 routing with 1.25 capacity factor and having one expert per core. During evaluation, the capacity factor can be changed to reduce compute.

## Serving techniques

> *"You can deploy mistralai/Mixtral-8x7B-Instruct-v0.1 to Inference Endpoints. "*

A big downside of MoEs is the large number of parameters. For local use cases, one might want to use a smaller model. Let's quickly discuss a few techniques that can help with serving:

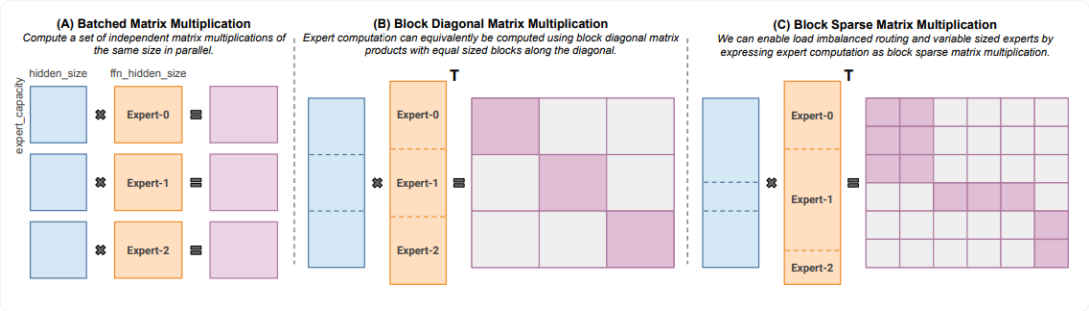
- The Switch Transformers authors did early distillation experiments. By distilling a MoE back to its dense counterpart, they could keep 30-40% of the sparsity gains. Distillation, hence, provides the benefits of faster pretaining and using a smaller model in production.
- Recent approaches modify the routing to route full sentences or tasks to an expert, permitting extracting sub-networks for serving.
- Aggregation of Experts (MoE): this technique merges the weights of the experts, hence reducing the number of parameters at inference time.

## More on efficient training

FasterMoE (March 2022) analyzes the performance of MoEs in highly efficient distributed systems and analyzes the theoretical limit of different parallelism strategies, as well as techniques to skew expert popularity, fine-grained schedules of communication that reduce latency, and an adjusted topology-aware gate that picks experts based on the lowest latency, leading to a 17x speedup.

Megablocks (Nov 2022) explores efficient sparse pretraining by providing new GPU kernels that can handle the dynamism present in MoEs. Their proposal never drops tokens and maps efficiently to modern hardware, leading to significant speedups. What's the trick? Traditional MoEs use batched matrix multiplication, which assumes all experts have the same shape and the same number of tokens. In contrast, Megablocks expresses

MoE layers as block-sparse operations that can accommodate imbalanced assignment.




Block-sparse matrix multiplication for differently sized experts and number of tokens (from [MegaBlocks](https://arxiv.org/abs/2211.15841)).

## Open Source MoEs

There are nowadays several open source projects to train MoEs:

- Megablocks: https://github.com/stanford-futuredata/megablocks
- Fairseq: https://github.com/facebookresearch/fairseq/tree/main/examples/moe_lm
- OpenMoE: https://github.com/XueFuzhao/OpenMoE

In the realm of released open access MoEs, you can check:

- Switch Transformers (Google): Collection of T5-based MoEs going from 8 to 2048 experts. The largest model has 1.6 trillion parameters.
- NLLB MoE (Meta): A MoE variant of the NLLB translation model.
- OpenMoE: A community effort that has released Llama-based MoEs.
- Mixtral 8x7B (Mistral): A high-quality MoE that outperforms Llama 2 70B and has much faster inference. A instruct-tuned model is also released. Read more about it in the announcement blog post.

## Exciting directions of work

Further experiments on **distilling** a sparse MoE back to a dense model with less parameters but similar number of parameters.

Another area will be quantization of MoEs. QMoE (Oct. 2023) is a good step in this direction by quantizing the MoEs to less than 1 bit per parameter, hence compressing the 1.6T Switch Transformer which uses 3.2TB accelerator to just 160GB.

So, TL;DR, some interesting areas to explore:

- Distilling Mixtral into a dense model
- Explore model merging techniques of the experts and their impact in inference time
- Perform extreme quantization techniques of Mixtral

## Some resources

- Adaptive Mixture of Local Experts (1991)
- Learning Factored Representations in a Deep Mixture of Experts (2013)
- Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer (2017)
- GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding (Jun 2020)
- GLaM: Efficient Scaling of Language Models with Mixture-of-Experts (Dec 2021)
- Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity (Jan 2022)
- ST-MoE: Designing Stable and Transferable Sparse Expert Models (Feb 2022)
- FasterMoE: modeling and optimizing training of large-scale dynamic pre-trained models(April 2022)
- MegaBlocks: Efficient Sparse Training with Mixture-of-Experts (Nov 2022)
- Mixture-of-Experts Meets Instruction Tuning:A Winning Combination for Large Language Models (May 2023)
- Mixtral-8x7B-v0.1, Mixtral-8x7B-Instruct-v0.1.

## Citation

```
@misc {sanseviero2023moe,
```

```
    author       = { Omar Sanseviero and
                     Lewis Tunstall and
                     Philipp Schmid and
                     Sourab Mangrulkar and
                     Younes Belkada and
                     Pedro Cuenca
                   },
    title        = { Mixture of Experts Explained },
    year         = 2023,
    url          = { https://huggingface.co/blog/moe },
    publisher    = { Hugging Face Blog }
}
```

```
Sanseviero, et al., "Mixture of Experts Explained", Hugging Face Blog, 2023.
```

## More Articles from our Blog


### SegMoE: Segmind Mixture of Diffusion Experts

By Warlord-K February 3, 2024 guest • △ 7


### Welcome Mixtral - a SOTA Mixture of Experts on Hugging Face

By lewtun December 11, 2023 • △ 11