Table of Contents

[MOE PAPER REVIEWS 2](#_Toc162110115)

[Understanding MoE 2](#_Toc162110116)

[Learning Factored Representations in a Deep Mixture-of-Experts 2](#_Toc162110117)

[Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer 5](#_Toc162110118)

[MoE articles 7](#_Toc162110119)

[Towards Understanding MoE 9](#_Toc162110120)

[OpenMoE: An Early Effort on Open Mixture-of-Experts Language Models 11](#_Toc162110121)

[Routing & Architecture 15](#_Toc162110122)

[BASE Layers: Simplifying Training of Large, Sparse Models 15](#_Toc162110123)

[DSelect-k: Differentiable Selection in the Mixture of Experts with Applications to Multi-Task Learning 17](#_Toc162110124)

[Hash Layers for Large Sparse Models 18](#_Toc162110125)

[Mixture-of-Experts with Expert Choice Routing 22](#_Toc162110126)

[From Sparse to Soft Mixtures of Experts 28](#_Toc162110127)

[Mixture-of-Tokens: Efficient LLMs Through Cross-Example Aggregation 34](#_Toc162110128)

[StableMoE: Stable Routing Strategy for Mixture of Experts 38](#_Toc162110129)

[Mixtral of Experts (+Mistral 7B) 41](#_Toc162110130)

[DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models 45](#_Toc162110131)

[Scaling & Stability 54](#_Toc162110132)

[GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding 55](#_Toc162110133)

[Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity 59](#_Toc162110134)

[GLaM: Efficient Scaling of Language Models with Mixture-of-Experts 67](#_Toc162110135)

[ST-MoE: Designing Stable and Transferable Sparse Expert Models 71](#_Toc162110136)

[Unified Scaling Laws For Routed Language Models 76](#_Toc162110137)

[Efficient Large Scale Language Modeling with Mixtures of Experts 80](#_Toc162110138)

[Task/Domain-Level MoE 83](#_Toc162110139)

[Beyond Distillation: Task-level Mixture-of-Experts for Efficient Inference 83](#_Toc162110140)

[Expert Gate: Lifelong Learning with a Network of Experts 89](#_Toc162110141)

[DEMix Layers: Disentangling Domains for Modular Language Modeling 97](#_Toc162110142)

[Branch-Train-Merge: Embarrassingly Parallel Training of Expert Language Models 104](#_Toc162110143)

[Scaling Expert Language Models with Unsupervised Domain Discovery 113](#_Toc162110144)

[Exploring the Benefits of Training Expert Language Models over Instruction Tuning 121](#_Toc162110145)

[MoE Efficiency 126](#_Toc162110146)

[MegaBlocks: Efficient Sparse Training with Mixture-of-Experts 126](#_Toc162110147)

[Fast Feedforward Networks + Exponentially Faster Language Modeling 127](#_Toc162110148)

[Parameter-Efficient Mixture-of-Experts Architecture for Pre-Trained Language Models 131](#_Toc162110149)

[Pushing Mixture-of-Experts to the Limit: Extremely Parameter Efficient MoE for Instruction Tuning 134](#_Toc162110150)

[Parameter-Efficient Sparsity Crafting from Dense to Mixture-of-Experts for Instruction Tuning on General Tasks 138](#_Toc162110151)

[QMoE: Practical Sub-1-Bit Compression of Trillion-Parameter Models 142](#_Toc162110152)

[Fast-Inference of Mixture-of-Experts Language Models with Offloading 145](#_Toc162110153)

[Hybrid Approaches 148](#_Toc162110154)

[Sparse Upcycling: Training Mixture-of-Experts From Dense Checkpoints 148](#_Toc162110155)

[EvoMoE: An Evolutional Mixture-of-Experts Training Framework via Dense-To-Sparse Gate 150](#_Toc162110156)

[Soft Merging of Experts with Adaptive Routing 154](#_Toc162110157)

[MoE-Mamba: Efficient Selective State Space Models with Mixture of Experts 158](#_Toc162110158)

[BlackMamba: Mixture of Experts for State-Space Models 161](#_Toc162110159)

[Mixture-of-Experts Meets Instruction Tuning: a Winning Combination for Large Language Models 164](#_Toc162110160)

[Multimodal MoE 170](#_Toc162110161)

[Multimodal Contrastive Learning with LIMoE: the Language-Image Mixture of Experts 170](#_Toc162110162)

[MoE-Llava: Mixture of Experts for Large Vision-Language Models (+ Visual Instruction Tuning aka Llava) 170](#_Toc162110163)

[Llava-Phi: Efficient Multi-Modal Assistant with Small Language Model 170](#_Toc162110164)

# **MOE PAPER REVIEWS**

## **Understanding MoE**

### Learning Factored Representations in a Deep Mixture-of-Experts

Main Idea:

* To apply stacked layers of mixture-of-experts, so to have multiple sets of (gating, experts). This allows multiple combinations of experts to be called while keeping a modest model size.

The problem they are trying to solve for is that deep neural networks are expensive to compute at inference time since all the neurons are used.

The solution proposed is to implement stacked MoE layers, where multiple expert combinations are possible, and the gating mechanism ensures only useful neurons for that input are used (experts on the specific input space). This gives better computational efficiency at inference, allowing for a model that is both large and efficient.

Approach:

* The input is first passed through the first MoE layer (represented by z1):
  + where and represent the gating probability and expert output for expert *i* at layer 1, respectively.
    - both the gating mechanism and the expert function use a non-linearity (ReLU)
  + The outputs of the first layer (z1) are then passed as an input to the next MoE layer z2, which replaces x with z1.
  + z2 is then passed through a final layer (f3) and a softmax is applied (in the context of classification)

The network is trained with SGD with a caveat to help balance the training through the experts:

* The mean of all experts’ total assignment is compared to each expert’s running total assignment. If an expert is found to have a running total assignment significantly higher than the mean, its training is paused temporarily to allow for the training of other experts.
* This strategy is found to mostly be useful in early stages of training, where the experts have not yet specialized significantly on a part of the input space. After some training, the experts are expected to have some specialization, and thus this constraint can be lifted.

This paper makes use of conditional computation, although the details about this are not shown in-depth.

Results:

* The stacked MoE layer showed promising results, as it came close to fully dense networks in terms of performance while having significant inference pros due to conditional computation.
* Experiments in specific tasks also showed that different experts indeed did specialize in different clusters of the data.

My takeaways:

* This paper is revolutionary in terms of the idea presented in terms of stacking MoE layers in a deep neural network and trying to find a way to balance the load between experts.
* Introduces the idea that MoE can have improved performance when stacked, paving the way for adding this as a modular component that can be added to other architectures.
* This strategy is still not sparse (top-k), but it opens the field to the idea that a top-k strategy is possible as a future line of research.

### Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer

Main Idea:

* To propose a way to improve model capacity, training time and model quality through a conditional computation approach that alternates between dense LSTM and MoE blocks.

Approach:

* Introduces a new neural network component (a new block/layer) which consists of:
  + n experts, each a feed-forward neural network
  + a trainable gating network, which selects a sparse combination of these experts to process each input token given.
* The gating network presented is an improvement over the standard approach, which trains a weight matrix to give score to an input x and pass that to a softmax (gating output . The gating mechanism proposed is called noisy top-k routing, which adds noise and sparsity:
  + Gaussian noise is added before taking the softmax to help with load balancing between experts during training.
  + Sparsity is added by taking only the top k scores given by the gating mechanism.
    - If not in the top k, H(x) becomes -inf so it is not considered in the final output.
* To balance expert utilization, an auxiliary term to the loss is added, which works by being computed at a batch level.
  + For each expert and the training batch X, take the expert’s importance in the batch:
    - Importance(X)e = sum of all the expert’s G(x) for the batch
  + The term Limportance is added to the loss (which will be computed at the batch level) to encourage all experts to have equal importance:
    - is a hand-tuned scaling factor and V is the coefficient of variation squared.
* The final network consists of alternating LSTM blocks with these new MoE blocks.

My takeaways:

* This approach means that for the first time MoE was used as a network component and not as the network itself, providing a method to integrate it with dense layers.
* Introduced top-k routing.
* Experiments showed that experts tend to become specialized on syntax and semantics, which is an important follow-up to the findings of the “Learning Factored Representations…” paper which hinted that different experts specialize in different clusters of the data.
* This paper also provides advancements in load balancing, crafting an auxiliary loss term for load balancing that seems much more effective than the previous method of pausing the training of highly utilized experts.

### MoE articles

The original MoE had 3 components:

* Experts, specialized models which are either regressors or classifiers.
* Manager (router), gate mechanism (like a softmax, for example) which decides in which area(s) of the input space each expert is trustworthy.
* Probabilistic model, which combines the expert and the manager. It joins the experts’ Gaussian distributions (outputs) together based on the probability given by the manager. Y = summation of pi (probability given to expert I by the manager) \* yi (output of the expert), for all experts.

This forms a fully differentiable dense ensemble of all experts with no inference speedup, as no expert computation is discarded.

Large dense neural networks are not efficient scaling in terms of training costs. Conditional computation models (sparse models) can provide advantages, but have their downfalls, such as the computational limitations of training such models (GPUs and TPUs are optimized for large matrix-matrix multiplication).

“Sparsely-Gated MoE Layer” tries to propose a solution to MoE’s computation issues. When training an MoE model the deep learning way, the input is passed through the router the same way as the original MoE method, however, the router only sends the input signal through to the top-k selected experts (a discrete choice, not fully differentiable), and uses the scores given by the router as weights of each expert’s output on the final output. The final output is then a combination of the top-k experts’ outputs weighted by their respective router score.

This deep learning approach has numerous potential problems:

* If one expert gets ahead and generalizes well fast, the router might send most of the data to this expert, overfitting and undertraining others while not specializing on anything. Therefore, training between experts needs to be somewhat uniform.
  + Common approaches to fix this are adding random noise to the router’s probabilities (scores given to experts) in order to create some randomness in the selection of experts’ process, especially in early stages of training (although we don’t want this to be fully random, since it will prevent specialization) to ensure that worse performing experts are still randomly picked for updates; adding a penalty term for uneven router choice to the loss function so the router has motivation to distribute its picks in a more uniform manner. This means the loss would look like: loss = cross-entropy loss + auxiliary loss, where auxiliary loss represents the penalty term for uneven distribution.

This sparse approach is promising in some ways as it provides computational efficiency for inference (only the selected expert weights are a part of the computation). So given 8 experts of 100M parameters each and a dense model of 800M parameters, a forward pass on the MoE model using k=2 would only trigger 2\*100M=200M parameters, while the dense model would always activate all 800M parameters (in reality, shared parameters should be accounted as well in MoE, but this is not mentioned here for simplicity). In theory, the quality of these 2 models should be roughly the same since they both have the same number of total parameters available (800M).

On another hand, due to the need to balance loads through the router function, MoE can be a bit slower to train. That is, the random noise and auxiliary loss to help with router uniformity between experts can slow down training due to data being sent and updated on suboptimal places. Due to its parameter efficiency, MoE has the potential to provide significant speed ups on training steps, but due to challenges such as load balancing and communication costs incurred by MoE, the cost of each step tends to be larger, so each training step takes longer. Therefore, when comparing training speed-ups between sparse and dense models, it is important to consider both training steps and training time.

### Towards Understanding MoE

An MoE layer contains many experts that share the same network architecture and are trained by the same algorithm, with a gating/routing function that routes individual inputs to a few experts among all the candidates.

The number of experts used for an input can be a hyperparameter choice called top-k (usually 1 or 2). The computation (inference) cost will only be the computation cost for the top-k expert(s) used.

In practice, all experts are initialized with the same weight distribution, optimization configuration, and the router is configured to distribute the data evenly between experts (traditionally through random noise and/or an auxiliary load balancing loss). This makes it unclear how this leads to specialization of each expert, instead of collapsing into a single model.

Key findings:

* MoE with linear experts cannot be trained to find a good classifier efficiently. An MoE with non-linear experts trained with gradient descent from random initialization can accomplish this. The gating mechanism, however, can be linear, since it only needs to differentiate between input clusters.
* The study shows that adding random noise to the router’s choice in soft routing (before the discrete choice) helps distribute the data across experts.
* For nonlinear MoE with non-linear expert functions, experts will diverge at the end of the exploration stage. At the end of the exploration stage, an expert will achieve low error in a specific cluster, but high error on the other clusters.
* There is a potential load unbalancing issue when training MoE, with the probability of each input being routed to the same few experts being high. This is a self-fulfilling prophecy, as it will lead to more training of these few experts, resulting in a bigger imbalance. Normalized gradient descent can help with this issue, as well as adding a penalty term to the loss function (auxiliary load balancing loss) or random noise to the router.
* The advantage of MoE over dense models in terms of performance depends on the task and the cluster structure of the data.

My takeaway(s):

* In MoE, the router specializes in dividing the input space into n parts/clusters (where n is the number of experts). Each expert then becomes a specialist on a specific cluster of the input space (as divided by the router).
* The router’s task can be performed linearly, as it only needs to learn how to divide the input space into clusters, while the expert’s task is more challenging, benefitting from non-linearities.
* It is important to employ load balancing strategies to ensure that this clustering is done correctly, especially at early stages of training when the clusters are not yet clear. If this is not done, it can lead to generalization (some experts being assigned to large areas of the input space while others are assigned to too small areas).
* The advantages of MoE will, therefore, depend on the input space of the data – if the data can be clustered into “specialization” areas, MoE will perform better, otherwise if the task benefits from a generalized knowledge of the input space, a dense model will outperform MoE.

### OpenMoE: An Early Effort on Open Mixture-of-Experts Language Models

Main Idea: an open-source project, OpenMoE analyzes decoder-only MoE LLMs from 650M to 34B total parameters and trained on up to 1T tokens (the largest version – 34B – was only trained on 200B tokens, 1T training tokens were used in the 8B version). The findings and recommendations of these experiments are shared in the paper.

The 34B version (largest one trained) has 6B active parameters per input and 32 experts per MoE layer. 5 intermediate checkpoints for the 8B model (every 200B training tokens) were released, and a Chat version of this 8B model was also trained (instruction-tuned).

Design

* Inspired by the facts that including code data in the pre-training dataset boosts performance and that code is always precise (contrary to text), which leads the authors to think that LLMs would more easily understand it, leading to better training, code data is aggressively sampled during pre-training (excessively/to a fault).
* Generally, follows the ST-MoE architecture and routing design. The reason for this is ST-MoE’s focus on training stability, a characteristic OpenMoE aims to achieve.
* Top-2 routing used during the entire training process.
* An MoE layer is inserted every 6 Transformer blocks, so most Transformer blocks do not have an MoE layer.
* Use UL2 method for the training objective (mix of span corruption and prefix language modeling).
* SFT for instruction-tuning is done on a dataset of 58k conversations, each with 1.8 turns on average, to analyze alignment (although this is not a big focus of this work).

Analysis

* MoE experts did not seem to specialize at the domain or at the task levels, but at the token level.
  + This is intuitive and rather obvious since the routing is done at the token-level.
* Context-independent specialization
  + MoE routing is done based on token ID and independent of the context around that token. This means that the routing is not really done based on semantics (context) but on syntax (the token being routed).
* Experts cluster tokens together, that is, they seem to specialize on a specific cluster of the token input space (the raw token’s embeddings without regard to context). Similar tokens are routed to the same expert.
* The token routing is learned at very early stages of training and remains fixed throughout the rest of training.
* Drop-Towards-the-End
  + Due to this fixed routing characteristic, something like instruction-tuning can lead to issues. This is because instruction-tuning data is out-of-domain, presenting a distribution shift from the pre-training data. Since the routing is learned from the pre-training data and is fixed, the distribution shift from instruction-tuning data will lead to overloaded experts, subsequently leading to token dropping in later rounds of the conversation (assuming multi-turn chat).

Takeaways/Recommendations

* The amount of code present in the pre-training data of over 50% was too aggressive (around 30% is recommended instead) and hurt the performance of the model in text tasks.
* The finding that MoE routing is fixed and established at early stages of training indicates that the router can be frozen after a warmup stage.
* The Context-Independent Specialization of experts indicates that the FFN (expert) computation can be done independently from the attention layer, thus an approach that would compute the expert FFN and the attention layers in parallel would make sense, bringing a speedup in training and inference.
  + Future research proposition.
* To alleviate the Drop-Towards-the-End issue, mixing instruction-tuning data into the pre-training data mix while the routing is being learned (the warmup stage) can be effective. This would allow the router to learn the instruction-tuning data distribution, so the token dropping issue experienced in later rounds of multi-chat conversation would be somewhat mitigated.

My takeaways:

* 5 checkpoints for the 8B OpenMoE model were released. This could potentially add to the routing analysis project I have planned.
* The conclusion that experts specialize on a specific cluster of the token input space seems to be inconsistent with the Hash Layers paper comparison of cluster-based hashing vs the opposite.
* The conclusion that token routing is fixed at very early stages of training seems to be inconsistent with the analysis done in the StableMoE paper.

## **Routing & Architecture**

### BASE Layers: Simplifying Training of Large, Sparse Models

Main Idea: introduces a new routing approach that approaches the problem as a linear assignment. This ensures load balancing without the need for auxiliary losses or adjusting CF. BASE also shows that a single expert/MoE layer can be effective.

* Makes use of top-1 routing like Switch.
* The linear assignment problem is designed to maximize token-expert affinities and has the constraint of balanced loads.

BASE Algorithm

1. Compute token-expert score for all experts.
2. Solve the linear assignment problem.
   1. Goal - Maximize token-expert affinity.
   2. Constraint – ensure balanced loads to experts at a batch-level.
3. Route tokens to experts.
4. Compute the expert scores as a weighted sum based on the routing weights.
   1. Top-2 routing is used at training.
5. Return the output to the original worker.

This approach is only used during training, as during test time the strategy of top-1 routing without load balancing is taken.

Results

* Having a single BASE layer in the network can be effective.
* Expert layers are robust to changes in the expert-shared parameters ratio and the position(s) of the layer in the network.
* Exploration of which inputs are assigned to each expert shows the same specialization patterns of other works: experts specialize on simple input patterns related to semantics and syntax.

### DSelect-k: Differentiable Selection in the Mixture of Experts with Applications to Multi-Task Learning

Main Idea: sparse gates commonly used for MoE, like top-k, are not smooth (continuously differentiable), which can lead to performance issues in gradient-based methods. Dselect-k presents a fully differentiable sparse gate for MoE.

Differentiability – a function that is differentiable is a function which has a defined derivative at every point. This is a requirement for gradient-based methods.

Continuous Differentiability – a function is continuously differentiable if it is fully differentiable AND the corresponding derivative is continuous. Continuous meaning smooth, with no abrupt changes and bumps. Continuous differentiability is not a requirement but optimizes performance of gradient-based methods.

* Top-k routing is not continuously differentiable due to the router’s hard selection of experts. This hard routing leads to it being possible for small changes in the input score to have large changes in the expert weights, which is not ideal.
* Dselect-k achieves continuous differentiability through smoothing techniques.

My takeaways:

* Although Dselect-k in theory should perform better than top-k, this technique has not been applied much in practice. This can be attributed to the increased computational complexity it brings, as well as to the simplicity and proven practical use of top-k.
  + Some other recently proposed continuous differentiability methods for MoE routing, check soft MoE (optimized for vision) and Mixture-of-Tokens (optimized for text generation).

### Hash Layers for Large Sparse Models

Main Idea: this paper experiments with using a hash router in an MoE architecture and compares it to other methods like top-k (Switch) and BASE.

Hashing

* Hashing refers to using a function that converts the input into a fixed size output that is unique for each input.
* Hashing does not need to be learned and there is no need for a load balancing loss.
* In a setting where the goal is to scale a model as much as possible (MoE), it is not realistic to try to optimize every hyperparameter and modeling decision (extensive tuning at this scale is too expensive). Hashing does not require this.
* Hashing is also a fixed mapping function, meaning it does not suffer from the issue from routing fluctuation/inconsistency during training (this was explored in the StableMoE paper).
* Some hashing functions used:
  + Clustered hashes – hash training inputs based on k-means clustering.
  + Dispersed hashes – assume the opposite of clustered hashes, that similar inputs need a more fine-grained distinction and should be assigned to different experts (closer inputs should be routed to different experts).
  + Random hashing.
  + Balanced assignment hashing.
  + Oracle future hash – obtains a hash to route token t based on token t+1 (the next token).
* This paper also experiments with what they call MultiHashLayer, which consists of using different hashing strategies in the same network, as to not rely on a single hashing strategy.

Models used in Experiments

* Baseline is a 222M parameter dense Transformer.
* Wider dense Transformer of 755M parameters.
* Deeper dense Transformer of 755M parameters.
* To compare to BASE, a 4.5B total parameter architecture with balanced assignment hashing is used.

Results

* When using a single MoE layer in a Transformer architecture (all other FFs remain the same), balanced Hash slightly outperforms Switch (using 64 experts in a model of 751M total parameters).
* Deep dense models of the same size as MoE (in terms of total parameters) outperform MoE, showing good dense models make better use of each parameter. At the same inference speed (active parameters), MoE performs better.
* Increasing the number of experts (from 64 to 128) leads to a better increase in Hash over Switch.
  + This indicates that the more experts there are in a layer, the less important it is to learn to route.
* As with BASE layers, adding a single Hash layer to a Transformer is shown to work better at later layers of the network.
* Increasing the number of sparse layers (in a setting where dense FFs and MoE layers are alternated) (5 sparse layers with 16 experts each) leads to better Switch performance over Hash.
  + Increasing the number of experts per layer might change this.
* Fine-tuning trends are consistent with pre-training trends.
  + The only part that can be frozen without hurting performance are the sparse layers.

Analysis/Results of Hashing Strategies

* Random and balanced hashing have similar performance (but balanced hashing has training advantages over distributed training schemes).
* Random hashing outperforms clustered hashes.
  + Proves the hypothesis that if tokens are like each other, a more fine-grained distinction is needed, and the tokens need to be routed differently.
* Dispersed hashing (opposite of clustered hashing) performs slightly better than random hashing.
* Learned routing (like BASE or Switch) generally provide clustered expert modules, which could be a disadvantage based on the results obtained during this research.
* Bigram and previous token hashing perform worse than just relying on the current token.
  + This indicates that using the previous token to help with routing is harmful.
* Increasing the dictionary size used for tokenization (thus increasing the number of possible hashes) leads to a decrease in performance against Switch.
  + This indicates that Hash might be better suited for scenarios where the dictionary size is small (so there are less possible hashes), while Switch is better suited to large dictionary size scenarios.
* Oracle future token hashing essentially solves the task.
  + This is expected since the hashing is performed on the target token (the answer).
* Increasing the diversity of hashing strategies (MultiHashLayer) seems to help.
* A learned routing based on the current token (and not on the hidden state, as Switch routing works) leads to small improvements.
  + This is a mix between the hashing strategy and Switch.
* When comparing Hash vs BASE, Hash outperforms BASE in every training step. BASE also shows instabilities at late stages of training, while Hash’s performance consistently improves (due to fixed routing).

Conclusion

Hash shows that there are lots of room for improvement in learned routing strategies. Hash should be used as a baseline for improving learned strategies in future work.

My takeaways:

* Why is it that random routing outperforms clustered hashes and dispersed hashing performs even better? Shouldn’t clustered hashing make more sense since we want experts to specialize in specific clusters of the input space? These results seem to indicate the opposite.

### Mixture-of-Experts with Expert Choice Routing

Main Idea: the main goal of this paper is to tackle the limitations in the MoE architecture caused by imperfect load balancing, which leads to under-training some experts and over-training others, as well as dropping tokens, through a novel approach called Expert Choice (EC).

How Traditional MoE (Token-Choice) Works

* Pass inputs into a gating mechanism which selects the most relevant k expert(s), in a process that relies on each individual token selecting the most relevant expert (token-choice). This leads to training inefficiency as the tokens are unevenly distributed.
  + To help with this, an auxiliary loss is commonly used and added to the loss function, but this still leads to some imbalance.
* The issue of expert capacity is also prevalent, since for efficient computation, usually each expert has a fixed block size to work with in terms of token assignment. A capacity factor can be increased to minimize dropped tokens, but this leads to more memory inefficiency.
* In token-choice, each input is also assigned a fixed compute, regardless of its complexity and/or task.

Expert Choice (EC)

At a high-level, EC routing has the expert models picking the most relevant input tokens instead of the other way around.

, where n is the number of tokens in a batch, c is the capacity factor hyperparameter and e is the number of experts. (c can also be thought of as the average number of experts assigned to each token). Expert capacity is the maximum number of tokens that can be assigned to each expert at a batch-level.

, where x is the input token representation and is expert’s g embedding.

G = matrix with weights given to each expert.

I = index matrix where I[I, j] specifies the j-th selected token of the i-th expert.

The gating input to each expert is then determined by

W1, W2 = parameters of the experts.

OBS: EC routing has no constraints for the number of experts assigned to each token.

OBS2: The capacity factor in EC is the equivalent to top-k in token-choice -> it is the average experts assigned to each token.

Results

* EC with a capacity factor of 2 should be computationally equivalent to top-2 routing. EC-CF2 has training convergence 2x faster than GShard top-2 routing.
* Scaling the number of experts during pre-training, given the same expert size, leads to better results, as expected (more total parameters = more specialized model = better quality).
* EC-CF2 performs better than Switch top-1 and GShard top-2 in all settings, but given a fixed expert size of 100M, increasing the number of experts seems to lead to worse fine-tuning results (opposite to pre-training results).
* Capping the number of experts to be assigned to each token leads to worse fine-tuning results. This shows that allowing variable number of experts per token is indeed helpful.
* EC learns to allocate a variable number of experts per token.

My takeaways:

* Understanding the routing mechanism as an unsupervised clustering method
  + At the early stages of training a model with MoE layers, the routing mechanism (assuming it is a token-choice method and that it is learned) is random, that is, it does not have information regarding of the area it will specialize in on the embedding input space. Without load balancing, the risk is of a specific expert being disproportionately chosen at these early stages, and thus taking up a large area of the input space for itself.
    - In other words, as an expert is picked by the routing mechanism on inputs of a specific cluster that it performs well on in relation to other experts, it will gain abilities that can be generalized to other clusters that other experts still do not have, due to the lack of tokens being assigned to them. This will create a feedback loop that results in a single expert taking up more and more input space, due, again, to the generalization abilities that it picks up along the way, which will end up averaging a single dense model, since the tendency is for this over-generalized expert to take up the entire input space area for itself.
  + The addition of an auxiliary load balancing loss is added to prevent this. To visualize this, we can think of an expert trying to grow its input space area but quickly reverting to a smaller area because of penalization effects.
    - Although this is helpful, there is still the risk of non-perfect clusters being assigned to each expert, especially at a batch level, which leads to other issues like token dropping.
  + In Expert Choice, this auxiliary load balancing loss is not needed, as the experts themselves will pick the tokens that are more relevant to them at a batch level (and not the other way around). If an expert has already reached full capacity, the 2nd expert that wanted that token the most will be chosen, etc.
    - I can imagine this leading to other problems. For example, some batches will contain tons of tokens that are part of the cluster of a specific expert, but the expert won’t be able to choose them because it has reached full capacity. In a token-choice scenario, this might lead to token dropping, which has the negative consequence of certain tokens not being used for inference (loss of information). In EC, this is not felt, but a new consequence may arise: tokens from the cluster of an expert will be given to another expert. Due to this impacting the next update, it can lead to nearby experts fighting for the input space of other nearby experts. Although this can be suboptimal at a batch level, training for many batches might neglect this effect (?).
  + Thought: Training an MoE model using token-choice and the strategy of MegaBlocks seems to be the ideal way to train a MoE model. This would get rid of the token dropping of token-choice, and not suffer from the negative consequence created by EC. The only assumption we’d have to make is that the load balancing loss and random noise penalties are a reliable way to find optimal token-expert assignments, given that token dropping is not an issue.
  + Future work idea – visualize the gating mechanism process and how it routes training inputs based on the clusters of each input embedding. Perhaps this can be done by using the checkpoints of the OpenMoE model (12 checkpoints available at HF I believe)?
* By not enforcing a constraint on the number of experts that can choose each token, EC creates a way for experts to determine how much compute will be used for each input. The idea is that the experts will learn complex and trivial inputs, maybe the intuition for this can be that complex inputs are in more complex/gray areas of the input space. With complex inputs, more experts will choose the token, leading to more computation being assigned to it. With trivial inputs that do not affect the output, no expert will choose the token, leading to no compute being applied to the token (token is dropped).
  + The difference between this token-dropping and token-choice’s token dropping is that this token dropping is learned, and not forced by lack of expert capacity.
  + This is shown to be helpful to fine-tuning performance.
* The result of increasing the number of experts helping in pre-training but hurting fine-tuning performance matches the findings of previous papers already discussed here.

### From Sparse to Soft Mixtures of Experts

Main Idea: Soft MoE presents a strategy to combat the issue of a traditional MoE of not having the property of continuous differentiability. By making a discrete choice (hard routing) to obtain sparsity, traditional MoE introduces training instabilities, as small changes in the input may lead to large changes in the model’s output, since this small change may end up changing the expert(s) chosen. The soft MoE architecture is compatible with certain tasks such as image classification in Vision or machine translation in Language, but it is not compatible with Natural Language Generation (NLG). An equivalent approach that could be compatible with language generation was proposed as “Mixture-of-Tokens” (MoT) (in a different paper), but this MoT architecture seemed to also bring significant challenges that remain unsolved. The continuous differentiability property of Soft MoE is, therefore, only able to be applied to a limited set of tasks.

Traditional MoE Routing

In traditional MoE, each input is routed to its corresponding expert in a hard manner (SoftMax is converted to 1 or 0) and the available slots are then occupied by a single token at a time (each slot gets 1 token). This means the experts will be updated solely based on that token.

OBS: slot refers to each inference run supported by the expert until it reaches its maximum capacity.

Soft MoE Algorithm

* Each slot pi of each expert has learnable parameters.
* The input tokens X are passed through each slot and a SoftMax is applied at the column level.
  + This means that the input slots to be passed to each expert are simply a weighted linear combination of all the input tokens with the respective slot’s learnable parameters.
* We then obtain the output slots by passing each input slot to a corresponding expert.
* The output slots are then merged through some combine weights, which are the inputs passed through the slot’s learnable parameters but now softmaxed at the row level (per token).
* As explained in the paper’s figure:

A diagram of a software

Description automatically generated

* + Soft MoE first computes scores or logits for every pair of input token and slot, based on some learnable per-slot parameters.
  + These logits are then normalized per slot (columns)
    - So now we have, for each slot, a weight to give to each input token, which sum up to one per slot.
  + And every slot computes a linear combination of all the input tokens based on these weights.
    - The tokens’ weights/embeddings are adjusted based on the weights/importance assigned to them per slot.
  + Each expert (an MLP) then processes its slots.
    - Now we have the experts’ outputs.
  + Finally, the same original logits are then normalized per token (by row) and used to combine all the slot outputs, for every token.
    - To get the final output for each token, we then obtain the softmaxed weights now normalized per token (instead of the slot’s weights sum up to 1, each token’s weights sum up to 1) and combine the expert’s outputs with those weights accordingly.

Intuition for softmaxes

* By slot (column)
  + Leads to scores being given for each token by the slot, used to measure the importance which should be given to each token for a specific slot (how much should the slot consider each token).
* By token (row)
  + Leads to scores being given for each slot (by the token), used to measure the importance which should be given to each slot for a specific token (to help determine the final output for each token) (how much should the token consider each slot).

Properties of Soft MoEs

* Usually to get past the token-expert assignment problem, MoE architectures resort to hard assignment methods such as top-k token-choice or expert-choice. These measures are discrete in nature, and thus non-differentiable. Soft MoE, on the other hand, is fully differentiable and continuous.
* Soft MoE does not suffer from token dropping or expert imbalance.
* Soft MoEs adjust better to hardware accelerators than “hard” MoE methods, mainly due to avoiding top-k/sorting routing operations (these are not well suited for hardware accelerators). Therefore, Soft MoEs are fast.
* Soft MoEs are neither sparse (since every token is a weighted average for all input tokens) nor dense (since every expert only processes a subset of the slots, and not all input tokens).
* Traditional MoE models are not so predictable at the sequence-level since inputting a single sequence may force the router to use every expert to balance the load and thus minimize the loss. This can lead to too generalist experts. Traditional MoEs are more predictable at the batch-level (more tokens) since a small number of tokens can fight for the same expert at the sequence level, but this risk is smaller at the batch level. Since in soft MoEs all tokens are grouped together and every expert handles tokens from every input, this risk is not present – leading to more deterministic/predictable and faster inference.

OBS: the number of slots in a soft MoE is a hyper-parameter (must be equal or greater than the number of experts).

Limitation in NLG:

* Soft MoE was only experimented with in an image classification scenario. Translating this method to an NLG setting is not so straightforward.
  + This is because soft MoE uses all input tokens to compute all output tokens at once. In NLG, each input token is generated at a time/separately (one-by-one) and is used as a part of the context to predict the next token. It is possible to use causal masking techniques to only take one token at a time, but this can lead to a bias in training (correlation between token position and a slot).
  + The sequential nature of token generation thus complicates the application of the Soft MoE architecture to language generation tasks.
* More research is needed to translate Soft MoE into an NLG setting.

Memory Consumption

Soft MoE works best when each expert is assigned to one slot only. Therefore, many experts need to be trained and stored, which comes with big costs in terms of memory.

Experiments (Image Classification only)

* Soft MoE is compared to other MoE methods – token-choice and expert-choice – and a dense setting and outperforms all of them in all hyperparameter scenarios.
* With cheaper training and inference costs, Soft MoE outperforms Vision Transformers at a large scale for a given compute budget in both pre-training and fine-tuning.
* Soft MoE scales the number of experts well (more experts = better). Additionally, scaling the number of experts in Soft MoE doesn’t really change training time, while this can have a tremendous negative effect in training time with token-choice and expert-choice.

My takeaways:

* Soft MoE seems to instead of routing each token individually, to route all tokens to each expert. This means that the expert will choose how much importance to give to each input token. The weighted average of the experts is then summed up based on the weights given to each token (normalized per token) to get the final output for each token.

### Mixture-of-Tokens: Efficient LLMs Through Cross-Example Aggregation

Main Idea: traditional sparse MoE has some drawbacks - the router decision is discrete, making it not fully differentiable for training, which can cause training instabilities; the load balancing between experts is also not guaranteed, which leads to the need to apply methods such as using an auxiliary loss or adding random noise to training inputs, which do not guarantee solving this challenge. MoT tries to improve on the traditional MoE architecture, providing a fully differentiable strategy which automatically results in load balancing.

OBS: MoT is compatible with both masked and causal LLM training and inference (fill missing tokens and autoregressive language modeling).

Issues with MoE

* The router is discrete, which causes training instabilities since small changes in the input may cause big changes in the gradient (if the small change in the input results in a different router selection). This makes the training process not fully differentiable. Using a weighted average of the selected experts to form the outputs seems to help with this but is not an optimal solution.
* There is no guarantee that the MoE will distribute loads evenly among experts. A capacity factor (CF) can be set for minimizing token dropping, but this does not help with load balancing and increases memory requirements. This prompts the use of an auxiliary loss, which is, again, not ideal.
* Most studies done with MoE are not compatible with autoregressive decoding (take soft MoE for example).

MoT Algorithm

1. The first step is to pass the input tokens (all of them) through a router/controller (linear layer) and apply a SoftMax to get the token importance scores for each expert.
   * 1. Where is a matrix with each input token as a row and each expert as a column.
     2. Each column sums up to 1, so each expert has its own designated router.
2. Then, the tokens are mixed by their importance weights, forming a mix of tokens for each expert.
   1. So, the token mix passed to expert i is ->
3. After having the mix of tokens for each expert, the next step is to pass the expert’s mix of tokens through its respective FFN.
4. To obtain the final output for a specific expert, we need to scale the expert output based on the importance for each token in its mix:
   1. Final\_output (for token t and a given expert) = expert output \* imp\_weight for token t
5. The final output for a given token t is then the sum of all the final outputs of each expert for that token t.

When doing this process for decoding, having to recompute each token multiple times seems inefficient, so a strategy to group tokens needs to be employed.

* The authors group tokens according to their position in a sequence (1st tokens grouped together, 2nd tokens grouped together, etc.). This way, for a given batch, each sequence can be computed in parallel, token-by-token.

Experiments

* The authors compare a GPT-like model to a MoT model (Transformer architecture with all feed-forward layers replaced with MoT layers).
* The MoT model shows promising pre-training results, achieving the vanilla Transformer’s final loss in 1/4th of the training steps and 1/3 of the training time.

My takeaways:

* Intuition about the MoT algorithm:
  + Calculating the importance vector of the input tokens is done at the expert level. This means that the input tokens are passed through the router for expert n, which will give the importance weights of each token for that expert. This is calculating how the mix of tokens which is passed to each expert will be weighted.
  + MoT sounds like Expert Choice Routing in terms of the expert choosing the importance to give to each token (in Expert Choice, however, the method used to determine if the token will be sent to an expert is given by the affinity or importance weight given by the expert, while in MoT every token is considered by every expert).
  + To get a final output, each token looks at the output of each expert, and considers how much importance to give to each expert’s output based on the importance the expert gave it.
* Although it is possible to do natural language generation with this approach, it seems to be very inefficient since generation of tokens cannot be done in parallel for all input tokens in the same sequence, while this approach takes all input tokens in the same sequence in consideration during inference and performs a forward pass in all experts.
  + Highly impractical in its given form for language generation. The design presented only works at a batch-level.
  + These limitations create the need for future research to make this approach practical.
* For now, this is only a training strategy, but does not work for inference.

### StableMoE: Stable Routing Strategy for Mixture of Experts

Main Idea: the goal of this paper is to solve the sample efficiency issue of training MoEs. The expert selection for a specific input may change during training, causing the weights of experts to be updated that will not be using it in inference – suboptimal training with experts being updated based on an input space that is not attributed to them during inference (routing fluctuation problems).

Problem

By observing the routing fluctuation issue when using BASE layers, it was observed that:

* 40.9% of tokens are unstable (inconsistent in routing) after 20% of the training steps.
* this number decreases to 29.1% after 50% of training, and to 15.4% after 80% of training.

Solution

Split training into 2 parts:

* Stage 1
  + start by training a router (with a new balance loss introduced – not much different, simply penalizes the loss in the case of expert overloading) and using sigmoid instead of SoftMax (sigmoid is thought to propagate the signal better) for determining the assigned expert’s weight.
  + During stage 1 of training, the router is distilled. This distillation process is accounted for in the training loss:
    - Total loss = task loss + balance loss + distillation loss.
    - The components that are important for this distillation are the experts’ centroids and the routing feature of the token t (distilled through a word embedding).
  + At the end of training stage 1, the parameters for the distilled router (which were being trained synchronously) (these parameters are the word embeddings for the tokens and the experts’ centroids) are frozen and kept frozen for the remainder of training (which consists of stage 2).
* Stage 2
  + In stage 2 of training, the router is distilled and stable, so only the task loss is needed. The sigmoid gate is kept so the gating signal is still being trained (I believe this is only for the actual weights given to each expert at inference). Everything else remains the same.

Results

The StableMoE method is compared to a dense Transformer, a Base MoE, a Hash Layer MoE and Switch Transformer at a base and a large setting (454M and 3.22B total parameters, respectively).

* StableMoE outperforms all others in all settings and shows robustness in scaling both model parameters and number of experts.
* Models improve perplexity with a higher number of experts (tested up to 64), given the same model size.
* Stacking MoE layers in-between Transformer blocks was shown to have the best results in comparison to sticking them in other positions.

My takeaways:

* At first glance, it seems logical that the routing fluctuation issue presented will result in suboptimal training, so traditional MoEs leave room for improvement in terms of training efficiency, especially in early stages of training.
* The part which seems to help the most is the routing distillation. The idea is to learn parameters to learn optimal expert centroids and token embeddings. Once this is learned, the router can be frozen to keep stability during training.
* The paper provides evidence that scaling the number of experts with StableMoE leads to improved performance not only in pre-training but also in downstream tasks like multilingual machine translation, as evidenced by higher average test BLEU scores compared to other models. This indicates that the advantages of scaling are not confined to pre-training. However, the paper doesn't provide an extensive evaluation on a variety of downstream tasks or fine-tuning with different amounts of data, which would be valuable for comprehensively understanding the scalability and efficiency of the model in varied contexts.

### Mixtral of Experts (+Mistral 7B)

Main Idea: Mixtral is a recent MoE model that is based on the Mistral architecture (Mistral is a dense model). The difference between these models is that each Mixtral layer consists of sparse FFNs, when these are dense in Mistral, with each sparse layer containing 8 experts each and being the equivalent of a 7B Mistral model.

Mistral 7B

Mistral uses grouped-query-attention (GQA) for accelerated inference speed and reduced memory requirements (allowing larger batch sizes) and sliding window attention (SWA) for handling longer sequences at a lower computational cost. The goal of Mistral is to provide an open-source model that beats other existing open-source models of similar size while improving on inference speed and memory/computational requirements, with a focus on practical use of the model and ease of fine-tuning.

Sliding Window Attention (SWA)

In regular attention, each token in a sequence attends to every other token, resulting in a complexity of O(n^2) with respect to the sequence length. In SWA, the tokens attended are limited by a sliding window, which masks tokens that are farther away from the current token than a pre-defined distance. This changes the complexity to O(n\*w), where w is the maximum number of tokens to be attended (maximum window size).

* SWA reduces computational complexity and memory usage – the longer the sequences the bigger the improvement.
* SWA, due to the fixed window size, allows for a rolling buffer cache (this increases efficiency).
* SWA also allows for pre-fill and chunking for more efficient inference.

Results

* Mistral is compared to Llama 2 7B/13B, Llama 1 34B and Code-Llama 7B.
* Compared to Llama 2 7B/13B and Llama 1 34B, Mistral performs significantly better in complex reasoning areas (code, math, reasoning) and comes close to Code-Llama 7B in coding tasks.
* On knowledge tasks, Mistral also tended to perform better but the gap observed was not as significant as in complex reasoning tasks.
* Instruction fine-tuning was performed using publicly available data to show the straightforwardness of fine-tuning on Mistral 7B.
  + This resulted in comparable performance to 13B instruct models.

Mixtral

* Mixtral uses top-2 token-choice routing.
* Mixtral excels at math, code generation and multilingual benchmarks (consistent with Mistral).
* A Mixtral-Instruct model (performed SFT and DPO) is also provided and surpasses GPT 3.5-Turbo.
* The context length of Mixtral is 32k.
* The gating mechanism of Mixtral takes the SoftMax of the top-2 expert scores and weights the expert’s outputs based on these weights.
  + The final output is then a weighted average of the sum of the two selected experts’ outputs.
* Mixtral seems to be robust to long-range contexts.
  + Perhaps due to Mistral’s SWA?
  + Experiments showed that up to a context length of 30k tokens, information can accurately be retrieved, and the perplexity of Mixtral decreases with an increase in context length.

The name Mixtral 8-7B might induce the thought of the architecture having 56B total parameters (8\*7), but it consists of around 47B parameters due to shared parameters between experts across the embedding, attention and normalization layers (7B is the full size of each expert if converted to a dense model). Likewise, the inference cost is not the equivalent of running 14B parameters (7\*2), but around 13B parameters due to these shared parameters.

In terms of routing analysis, it was shown that experts seem to be selected based on syntax rather than on specific domains – experts specialize in semantics and syntax, not on tasks. This is logical due to the token-choice routing. If routing is done on a token granularity, the experts are expected to specialize on token-level areas. With domain or task-routing (done at a sequence level), experts can be expected to specialize in domain/task-level areas.

My takeaways:

* The goal of Mistral 7B is to provide an open-source model with an optimal performance and efficiency balance.
  + Performance meaning quality, efficiency meaning inference speed and computational requirements.
* Sliding Window Attention seems to sacrifice the context length capacity in return of higher inference speed. The assumption taken for this not to hurt performance seems to be that the more you move away from a token, the lower the odds of it having meaningful dependencies to the current token.
  + Large context lengths are possible under SWA, but each individual token will not use the full context length for inference if the input is larger than the maximum window size.
* Perhaps the idea for Mixtral came after analyzing Mistral’s results? Since Mistral performs significantly better on reasoning tasks but the improvement in knowledge tasks is not so big, it would make sense to try to apply a MoE architecture to this model, with the idea being to retain the reasoning abilities while improving knowledge abilities. This makes sense because other studies seem to show that MoE, due to additional model capacity added, tend to perform very well on knowledge tasks (weakness of Mistral) but the performance on reasoning and fine-tuning tasks (strength of Mistral) leaves room for improvement (although MoE was shown to benefit from instruction-tuning in a more significant way than dense models).

### DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models

Main Idea: traditional top-k MoE experts acquire non-overlapping and focused knowledge, creating challenges for expert specialization. DeepSeekMoE plans on architectural changes to enforce expert specialization through expert segmentation and isolating experts as shared ones to capture common/overlapping knowledge.

3 versions of DeepSeekMoE are trained (in total # of parameters):

* 2B
* 16B
  + Can run on a single GPU with 40GB of memory.
  + Experimented with SFT to create an instruction-tuned chat model.
* 145B
  + Largest model trained.

2 Potential Issues of Traditional (top-k) MoE:

* Knowledge hybridity
  + Current MoE models have a low number of experts (8 or 16). This division assigns each expert to a diverse part of the data, so the parameters are not used so efficiently (there is more sparsity in the data possible than reflected by the number of experts).
* Knowledge redundancy
  + Experts may benefit from common knowledge, but since they are isolated, some experts might end up learning the same information, causing redundancy in their parameters.

Solutions Proposed by DeepSeekMoE

* Fine-grained Expert Segmentation
  + Segment experts into a finer grain by splitting the FFN hidden dimension. More experts are also activated (increase the number of experts while maintaining the number of total and active parameters).
  + More flexibility on which parameters of the experts to use – introduce sparsity within each expert – while keeping computational costs constant.
* Shared expert isolation
  + Isolate certain experts to serve as shared experts, which are always activated. The goal is for these experts to retain the common knowledge between experts, avoiding parameter redundancy.
    - Leads to parameter-efficiency + increased specialization.

Architecture

* As mentioned above, DeepSeekMoE incorporates two new strategies on top of the generic MoE architecture:
  + Fine-grained expert segmentation
    - Not just simply adding more experts but keeping the number of active and the number of total parameters the same while doing so.
    - A small number of experts combined with a low number of activated experts per input makes experts learn a diverse amount of knowledge when what we want is specialization.
    - To solve this, DeepSeekMoE divides the expert’s weights (more specifically, the FFN hidden dimension) into m segments, creating another level of experts. This allows for a scaling of m in the number of experts (if m is 8, the total number of experts will be scaled by 8, for example).
      * mN possible expert combinations vs N possible combinations.
    - This allows for a more flexible combination of experts, since the router will not only pick specific experts, but specific segments within experts.
    - This allows for a greater number of experts to be activated without increasing computational costs.
  + Shared expert isolation
    - Experts in conventional MoE are isolated. This means that if experts have overlap in knowledge in the data fed to them, this will be learned independently, so repeated parameters will exist for the same information, bringing parameter inefficiency.
    - DeepSeekMoE has shared experts – experts that are always activated – which have the goal of capturing this common knowledge so there is no parameter redundancy.
    - The number of shared experts is Ks. To keep computational costs, the number of routed experts will then decrease to mN-Ks and the nonzero gates (segment activations) will be mK-Ks.
  + Balance loss
    - An expert-level and a device-level balance loss are used, with more emphasis/weight on the device-level loss.

Experiments (2B parameter model)

* Substitute all FFNs by an MoE layer.
* 9 Transformer blocks with hidden dimension of 1280.
* Random initialization.
* 16 experts with 4 segments each (64 total expert segments), with 1 shared segment.
* Computation equivalent of top-k with k=2.
* 2B parameter model, 0.3B active parameters.
* Training of 100B tokens with 2k batch size.
* No dropout due to abundance of data used.
* Baselines:
  + Dense – equivalent to top-1 routing (~0.2B active parameters)
  + Switch – equivalent to top-1 routing (~0.2B active parameters)
  + Hash Layer – equivalent to top-1 routing (~0.2B active parameters)
  + GShard
* Results
  + Switch and Hash Layer perform better than Dense (with same number of active parameters but more total parameters).
  + GShard performs slightly better than Switch (with more active parameters).
  + DeepSeekMoE performs significantly better than GShard, with the same number of active and total parameters.
  + DeepSeekMoE closely aligns with the upper bound of MoE models (dense with same number of total parameters) (at least on the 2B total parameters scale when training with 100B tokens).
    - DeepSeekMoE 2B performs comparably to GShard 2.9B (1.5x the expert size) (the advantages increase when scaling to 13.3B and 19.8B, respectively).
    - DeepSeekMoE 2B achieves comparable performance to Dense with FFNs scaled by 16 (same number of total parameters, 16 is number of experts per layer used).
  + Ablation studies reassure the positive effects brought by fine-grained expert segmentation and shared expert isolation.
    - Additionally, the number of shared experts (1,2 and 4 tested with 64 total experts) did not seem to make much difference. A ratio of 1:3 (shared/total activated experts) is used when scaling the architecture.
  + Expert specialization
    - DeepSeekMoE was more sensitive to disabling the top-k experts, showing that there is less common knowledge between experts, thus less redundancy.
    - Shared experts are irreplaceable in DeepSeekMoE, that is, substituting a shared expert by a not-shared expert results in a significant drop in performance.
    - DeepSeekMoE can acquire knowledge more accurately and efficiently. Even using only 4 active experts (equivalent to top-1 routing), DeepSeekMoE performs similarly to GShard.
      * When using this setting of 4 active experts at training time, and not only at inference time, DeepSeekMoE outperforms GShard even with half of the number of active expert parameters.

DeepSeekMoE 16B

* Scaling up of the architecture to a model with 16B total parameters, trained on 2T tokens (same number of training tokens as Llama2-7B).
* 28 Transformer blocks, all FFNs are substituted by an MoE layer except for the first one (because the first layer takes longer to converge if the FFN is substituted by an MoE layer).
* Each MoE layer has 2 shared and 64 routed experts. Each FFN is divided into 4 experts.
* 8 experts per layer activated per input (2 shared, 6 routed), corresponding to 2.8B active parameters.
* Similar training setting to DeepSeekMoE 2B.
* Compared to DeepSeek 7B (its dense counterpart):
  + DeepSeekMoE 16B, with around 40% of active computation at inference, performs comparably to DeepSeek 7B.
  + DeepSeekMoE 16B performs especially well in language modeling tasks.
    - This indicates that scaling up the total FFN parameters helps with memorization.
  + DeepSeekMoE 16B does not perform well in multiple-choice questions.
    - A possible explanation for this can be due to the attention parameters. The number of attention parameters are thought of as being crucial for MC tasks, and the MoE version has around 5x less attention parameters than its dense counterpart (0.5B vs 2.5B).
* Compared to Llama2-7B:
  + DeepSeekMoE 16B, with about 40% of Llama2-7B activations at inference, outperforms it at most baselines (MC tasks like MMLU are the exceptions).
  + DeepSeekMoE 16B is stronger at math and reasoning tasks (strengths of Llama2-7B) probably due to the distribution of the dataset used for training.
  + Despite being trained on less English text, DeepSeekMoE 16B achieves equal or better performance at English understanding and knowledge-intensive tasks.
    - Consistent with MoE’s advantage in memorization due to increase total parameter count compared to dense.
  + On Hugging Face’s Open LLM leaderboard (collection of evaluation tasks), DeepSeekMoE 16B significantly outperforms models of the same size in terms of active parameters and achieves comparable performance to Llama2-7B.

Chat Alignment for DeepSeekMoE 16B (SFT/Instruction-Tuning)

3 models are compared in this section, all trained on the same data:

* Llama2 SFT 7B – Llama 2 instruction-tuned independently from its chat version, to control for the training data.
* DeepSeek Chat 7B.
* DeepSeekMoE 16B Chat – has around 40% of active computations compared to the other models used in this section.

Results:

* The MoE variant achieves comparable performance to the dense models in language understanding and reasoning, machine reading comprehension, and mathematical and knowledge-intensive tasks.
* The MoE variant performs significantly better at code generation.
* The gap in multiple-choice questions still exists but is narrowed.

Scaling DeepSeekMoE to 145B Total Parameters

* Trained on 245B tokens (will probably be scale dup in the future, so this can be seen more as a baseline).
* 62 Transformer blocks, all FFNs substituted by an MoE layer except the first one.
* 4 shared experts and 128 routed experts per MoE layer.
* Each expert is 1/8th the size of a standard FFN (different than the ratio used for the smaller 2B and 16B models, which was 1/4th).
* At inference, the 4 shared experts and 12 routed experts are activated.
* Around 22.2B active parameters.

Results:

* 3 additional models were trained for comparison, using the same training corpus and hyperparameters:
  + DeepSeek 67B (dense)
  + GShard 137B (GShard architecture trained on the same data)
  + DeepSeekMoE 142B (half-activated)
    - Uses half of the activations of DeepSeekMoE 145B – 2 shared experts, 6 routed experts.
* With similar number of active and total parameters, the MoE 145B variant significantly outperforms GShard.
* With only 28.5% of its active computations, the 145B MoE model reaches comparable performance to DeepSeek 67B.
  + Exhibits strong performance in language understanding and knowledge-intensive tasks but struggles in multiple-choice (consistent with the 16B MoE model performance).
* Despite having only half of the activated parameters, the 142B version is not too far behind from the 145B fully activated version, still matches the performance of DeepSeek 67B (with around 18.2% of its computations at inference) and easily beats GShard 137B.

My takeaways:

* DeepSeekMoE has its 16B version with 7 checkpoints released to HF. This could add to a potential exploration of how experts in MoE specialize.

## **Scaling & Stability**

### GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding

Main Idea:

GShard looks to make improvements on different challenges of training MoE models, particularly related to scaling. A 600B MoE model (2048E, 36L) is successfully trained, while the authors fail to reach a stable 1T parameter model (2048E, 60L) due to issues with training stability caused by reduced precision (bfloat32 to bfloat16). The improvements made were in the following topics:

* Computation costs when scaling.
* Ease of programming when scaling.
* Efficient scaling implementation on parallel devices.

GShard modifies the traditional Transformer architecture by alternating between a self-attention and a MoE layer with top-2 routing. To scale, the model is stretched vertically (increase in number of layers in each expert) and/or horizontally (increase in the number of experts per MoE layer).

For dealing with load balancing:

* A hyperparameter for a maximum threshold for the number of tokens to be sent to each expert per batch is set (expert capacity, set to N/E – N=# of tokens in the batch; E=# of experts).
  + Extra tokens (that couldn’t ‘t make it due to the expert capacity being reached) are overflown/discarded.
* Training tokens are distributed evenly into G groups to take advantage of parallelism. Expert capacity is evaluated in a group basis – local group dispatching.
  + Experts are divided into groups that are optimized for communication (communication between experts in the same group is faster than between experts in other groups). Local communication (which is optimized) are used more between experts instead of global communication.
* Addition of a load balancing term to the loss function based on the mean number of token assignment to all experts compared to the token assignment for each expert (calculated at the group level).
* Random routing is employed to help with the expert capacity constraint. Top-2 routing requires a capacity factor of 2. To help with this, some tokens which have a low gating weight for the 2nd-best expert are not propagated through this expert (becoming top-1 routing). These 2nd-best experts are dropped randomly in proportion to the gating weight they were assigned (if assigned a score of 0.2, it would have a higher chance of being dropped than if it was assigned a score of 0.3).

Results:

* Scaling the number of layers (vertical scaling) leads to consistent gains.
* Increasing the number of experts used has diminishing returns.
* Increasing the number of experts helps with high-resource tasks (which have more data), while dense models adjust better to low-resource tasks (low amount of data).
* In terms of training efficiency:
  + Scaling with conditional computation is more practical and efficient than with dense models.
  + Deeper models are more sample efficient (converge faster with fewer examples). That is, increasing the number of layers in a model leads to an almost proportional speed up in terms of training steps to reach a certain loss (a 3x increase in number of layers would lead to ~3x speed up in training steps to reach a certain loss).
  + As mentioned previously, scaling the number of experts per-layer has diminishing returns.

My takeaways:

* GShard is the first attempt of massively scaling MoE in a Transformer architecture. It does so by optimizing the technical implementation of MoE for communication costs and parallelism.
* Highlights:
  + The techniques used for balancing dropping tokens by adjusting the expert capacity (needs to be higher for a higher k) as well as randomly dropping the 2nd-best expert are interesting.
  + The local group dispatching technique to minimize communication overhead costs also seems interesting and deserves a deeper look/understanding.
* Based on my analysis of this paper, I was left with a few questions/thoughts:
  + Are MoE layers robust to dropped tokens? As each expert is assigned to a specific input space, my first thought is that if the experts are of modest size and enough experts are employed per layer (making the specialized input space for each expert smaller), the MoE architecture should be robust to dropping tokens.
  + Are different experiments employed at future research works regarding MoE performing better at high-resource tasks and dense models performing better at low-resource tasks? This seems to mean that MoE’s gains over dense models come at the expense of a bigger amount of data being needed.
  + The fact that increasing the number of experts per-layer leads to diminishing returns makes sense –> as each expert specializes in a certain area of the input space, increasing the number of experts will decrease the size of that area, allowing the experts to specialize further. However, at some point the experts will become redundant (too many experts for a small input space area/cluster), leading to diminishing returns due to these redundant experts having the same specialization.
    - Based on this logic, it should be possible to balance the expert size with the number of experts in each layer. The logic is that decreasing the expert size would lead to each expert only being able to handle smaller input spaces/clusters of the data, so more experts would be needed.
  + The fact that deeper models are more sample efficient hints that the scaling laws for MoE should be like those of dense models. This makes sense as adding more layers is adding computation to the model.

### Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity

Main Idea: the idea for this paper is based on the “Scaling Laws for Neural Language Models”, which states that larger models are more sample-efficient, and thus advises that the optimal allocation of a fixed compute budget should prioritize increasing the number of model parameters while decreasing the number of training steps. This created the motivation to scale MoE models, which allow for an increase in parameter count while keeping FLOPs constant. The main issues to be addressed related to scaling MoEs are:

* Complexity
* Communication costs
* Training instability

Top-1 Routing

Sparsely-Gated MoE had claimed that top-k routing had to have k > 1 to have non-trivial gradients to the routing function (the routers were thought to not train properly if they didn’t have at least two experts to compare results with). Switch challenges this idea and successfully uses top-1 routing. This introduces advantages such as reduced computation, reduced batch size (top-2 routing requires an expert capacity factor of 2, which is not needed in top-1 routing) and reduced communication costs.

Expert Capacity

Each expert has an expert capacity, which is the computation it can perform on each batch. Due to the dynamic nature of routing (load balancing in a batch is not guaranteed), this expert capacity can lead to memory overflow issues (where the overflown tokens in a batch are skipped). This can be managed by setting a capacity factor to the experts (keep some buffer to each expert’s machine).

Expert capacity = (tokens per batch/number of experts) \* capacity factor

Although this helps with memory overflow and the issue of skipped tokens, it results in increased computation and memory costs.

Load Balancing Loss

For the auxiliary loss, Switch introduces a differentiable load balancing loss that considers both the fraction of tokens assigned to each expert and the probability given to each expert by the router (sum of the probabilities given to each expert when it was selected).

Auxiliary loss = alpha \* N \* summation over all experts (fi \* pi), where N is the number of experts, fi is the fraction of tokens in the batch dispatched to expert I and pi is the fraction of the router probability allocated to expert i.

This loss ensures load balancing by leveraging the fact that the product fi \* pi is minimized under a uniform distribution, where both fi and pi are equal or close to 1/N for each expert, corresponding to a balanced load. The sum of fi and pi is constrained to 1 across all experts, highlighting the zero-sum nature of resource distribution. The non-linear impact of the product fi \* pi in the loss function means that the sum of these products across experts is minimized when the load (dispatched tokens) and router probabilities are evenly distributed. This minimization drives the model toward a uniform distribution, promoting load balance by ensuring that no single expert is disproportionally favored in terms of load or router’s allocation.

This loss is a complement to the cross-entropy loss -> total loss = cross-entropy + auxiliary loss.

T5 (dense) vs MoE (top-2 routing) vs Switch (top-1 routing)

Models were trained on a masked language modeling objective with 15% token dropout (for MoE and Switch).

The same computation per token is applied (equal FLOPs) for each model. However, MoE has more active parameters since it uses top-2 routing.

* Switch outperforms T5 and MoE in terms of speed-quality (fixed amount of computation and wall-clock time)
* Switch has a lower computational footprint -> increasing its size to match the speed of MoE leads to outperforming MoE and T5 on a per step basis (since MoE is slower than Switch due to higher number of active parameters)
* Switch performs better at lower capacity factors (1, 1.25)

Training and Fine-Tuning Techniques

Instability in MoE comes mainly from the hard-switching routing strategy. This makes it challenging to train in lower precision. To combat this, a few tricks are used:

* Selective precision with large sparse models
  + Selective casting to float32. More specifically, the router input is casted to float32 within the body of the router function (local computations) but back to float16 at the end of the routing function when the results are dispatched for the selection of the router computation (between devices). This optimizes the router stability while keeping the communication costs low.
* Smaller parameter initialization for stability
  + Simple initialization changes (especially reducing the normal initialization scale of a Transformer by 10) drastically helps with stability.
  + A popular initialization strategy is used -> weights randomly initialized from a distribution with mean of 0 and st dev of sq root(s/n), where s is a scale hyper-parameter and n is the number of input units in the weight tensor.
* Regularizing large sparse models
  + Since MoE models have much more parameters than regular dense Transformers, they can be more prone to overfitting when fine-tuned in small downstream tasks.
    - Switch proposes increasing dropout in expert layers while keeping a smaller dropout rate in other layers. This is shown to lead to improvements in fine-tuning.

Scaling properties

* When keeping the FLOPs per token fixed, having more total parameters (increase in number of experts) speeds up training (although at a cost in memory) in a per-step basis (training is more sample-efficient).
* MoE models have higher communication costs than dense models. So even though Switch is more efficient in a per-step basis, this can fail to hold in a time basis.
  + With a fixed training duration and computational budget, Switch achieves a 7x speedup in training compared to T5 (Switch achieves the same loss 7x faster).
  + Switch shows improvements in both per-step and time basis during pre-training over T5 even when compared to T5-Large (3.5x increase in FLOPs).

Fine-tuning

With an increase in dropout rate (0.4 vs 0.1), Switch was shown to have improved fine-tuning results over T5-Base and T5-Large in a FLOP-matched basis in NLP tasks, including reasoning and knowledge-heavy tasks.

Distillation

When distilling a large sparse model into a small dense model, it is found that reducing the model to 1/20th of its original parameter count still retains 30% of the Switch gains over T5. This is a sign that not all gains are due to increased parameter count, indicating that some part of the gains can be due to other reasons related to the MoE capturing parameters more efficiently.

Parallelism (Data, Model, Experts)

Data parallelism – data is shared over all cores available, while keeping a copy of the model in each core (model is replicated over each core). Each core (model) only needs to communicate at the end of each batch to perform an update on the model’s parameters.

Model parallelism – model is distributed over all cores, while passing all tokens through each core. This method leads to high communication costs between cores since each token needs to be passed from core to core to produce a label.

Model and data parallelism – model is split through m cores and data is split through n cores (mix of pure model parallelism and pure data parallelism.

Expert and data parallelism – the model is distributed by having one expert in each core while sharding the data over all cores. This sharding is done by the routing function, assuming the auxiliary loss will help with load balancing to prevent the token overflow issue.

Expert, model and data parallelism – more complex method where each expert is distributed through multiple cores (in case a single expert does not fit in a single core, which can happen if we want to increase the number of FLOPs – this leads to a decreased batch size because more memory is needed for the experts and the communication costs between cores, leading to less memory available for the data). This needs to consider the communication costs between the routing function distributing the data and the model/expert sharding.

Increasing the number of experts does not seem to lead to instability in training (as seen in training the 1.6T model). What caused instability is increasing the number of FLOPs (increasing the size of each expert).

My takeaways:

* The claim made on Sparsely-Gated MoE that k > 1 is needed for top-k routing initially seems to make sense. This would help the gradient to differentiate between good and bad experts for that input. For example, with k = 2, the router can compare the gradients that come from each expert, and therefore learn which expert was more useful to the final output. With k = 1, this property is not present.
  + Top-1 routing, even if it works, would not benefit from overlaps in the clusters that each expert specializes in. Coupled with reduced computation (less parameters used during inference), it seems that this would lead to efficiency gains but with a loss in performance.
* It is true that increasing the model parameters makes the model more prone to overfitting, especially when there is not enough data available (the more data, the less the risk of overfitting), which is more likely during fine-tuning. Increasing regularization (dropout in this case) is logical when it comes to helping with that. Remember that dropout will randomly drop training samples, allowing the model to go through the data more times.
* Switch is shown to perform significantly better than dense models in pre-training (in both a per-step and per-time basis).
* Increasing regularization during fine-tuning shows promise for MoE models. However, MoE architectures are not as well suited as dense models for fine-tuning due a higher amount of data being needed to prevent overfitting.
  + Results from Switch show that it outperformed T5 in fine-tuning tasks such as GLUE and SQuAD. However, these seem like tasks that have enough data to prevent the issue of overfitting in Switch. It would be interesting to see how this holds when fine-tuning on tasks with less data available.
* Distillation results from Switch show great promise, as it hints that models can be pre-trained in a MoE architecture and then distilled while still performing better than just pre-training on a dense architecture.
* When training an MoE model, it would make sense to use expert parallelism in the scenario where a single expert fits into a core, and to use expert, model and data parallelism in the case of a single expert not fitting into a core.
* The observation that increasing the size of each expert (and not the number of experts) is what causes instability is interesting as it shows that this perhaps leads to experts that are too complex for the clusters they are assigned to (although this should be true for an increase in the number of experts – more experts = smaller clusters for each expert). From intuition, it seems that a balance between number of experts and expert size is needed.

### GLaM: Efficient Scaling of Language Models with Mixture-of-Experts

Main Idea: with the improvement of language models mainly coming from scaling the number of parameters in a dense setting, training these models requires more and more compute and resources. GLaM looks to explore sparse language models (MoE) to reach comparable or superior performance to dense models while decreasing training costs. During evaluation, GLaM focuses on zero-shot and few-shot learning capabilities. The importance of data quality during pre-training is also analyzed.

The largest GLaM model has:

* 1.2T total number of parameters.
* 96.6B active parameters.
  + For comparison, GPT-3 is a 175B parameter dense model.
* 64 experts per MoE layer.

GLaM seems to have been the first study to use a decoder-only MoE on a model of comparable size to GPT-3. Switch, for example, had only around 1B active parameters (compared to 96.6B of GLaM) per input and was an encoder-decoder model.

The training dataset used to train GLaM was highly filtered to ensure that low-quality content was not prominent (although a small collection of low-quality training data is kept to prevent systematic biases).

Architecture

* Alternate between FF (dense) and MoE (sparse) layers.
* Regular top-2 routing, with the output being a weighted average based on the scores given by the routing.
* Auxiliary load balancing loss.

Evaluation Setting

* Mainly focuses on zero-shot, one-shot and few-shot performances of the models being evaluated.
  + This is different to Switch, which focuses on fine-tuning performance.
  + This is consistent with new capabilities shown by scaling LMs as shown by GPT-3.

Results

* GLaM (64B/64E) (96.6B active parameters) has consistent gains in zero, one and few-shot performances over GPT-3, while requiring roughly only half of the compute FLOPs at inference (96.6B vs 175B).
* Improved performance (over GPT-3) on the challenging TriviaQA domain indicates that the additional capacity of GLaM plays a crucial role in its performance gains.
  + On GPT-3’s paper, GPT-3 was shown to consistently improve on this task (TriviaQA) given an increase in parameters, which was attributed to its ability to retain more knowledge with an increase in parameters.
* Using a small model (GLaM (1.7B/64E)), it was shown that the quality of the pretraining data plays a crucial role in determining the quality of the model.
  + The impact of data quality was bigger in NLG tasks compared to NLU tasks.
* MoE models can be scaled in two ways:
  + Increasing the number of experts
    - Keeps the number of active parameters (and thus the compute FLOPs at inference) constant.
    - Increasing the number of experts generally resulted in better performance up to 64 experts (there was a decline in performance in further increases after 64).
  + Increasing the size of experts
    - Leads to an increase in inference costs.
    - Results in improved performance.
* GLaM MoE models perform consistently better than GLaM dense models for similar effective FLOPs per token.
  + MoE models perform similarly to dense at smaller scales but obtain an advantage when scaling the model.
* In terms of data efficiency, GLaM MoE models require significantly less data than dense models of comparable FLOPs.
  + When the same amount of data is used for training, MoE models perform much better, and the difference in performance becomes larger when training up to 630B tokens, so this advantage increases with scale.
* In terms of computational efficiency and energy consumption, sparse models take much less computational resources to achieve the same performance.
  + GLaM (64B/64E) has around 1/3 of training costs of GPT-3, while also halving the inference cost and using 1/6th of the energy costs.
    - These gains can be attributed to the MoE architecture’s superior training efficiency.

My takeaways:

* In terms of architecture, GLaM does not seem to provide any significant advancements in MoE. The main quality of this research was to analyze how the MoE architecture would perform at a large scale (especially of number of active parameters) in a decoder-only model for NLG.

### ST-MoE: Designing Stable and Transferable Sparse Expert Models

Main Idea(s): this paper provides a thorough study on MoEs. It tackles the biggest challenges presented to MoE models at the time of its release, with those being instabilities in training and poor fine-tuning performance. Its main goal is therefore to improve the practicality and reliability of sparse models.

* Trains a 269B sparse encoder-decoder model.
* Introduces router z-loss to resolve instability issues.

Stabilizing Training of Sparse Models

Transformer models today are normally trained by using float32 to compute gradients and float16 to compute the forward and backward pass. Sparse models contain several exponential functions (like softmax), which can lead to large values flowing through the network. Float16 does not handle large numbers well, as the larger the number, the larger its resulting rounding error. It is proposed that this abundance of exponential functions in MoE is what causes training instability. Router z-loss is a trick to penalize large values from flowing through the network, thus improving stability.

* Router z-loss is a function that stabilizes the training of MoE models without degradation in model quality by penalizing large values from flowing through the network.
  + Stability is referred to as constant/smooth decrease in the training loss.
  + Router z-loss is a complement to the overall loss function, as cross-entropy and auxiliary load-balancing loss are also used.
  + So total loss = cross-entropy + auxiliary load-balancing loss + router z-loss.

Fine-Tuning Sparse Models

Model characteristics:

* Dense and sparse models both pre-trained on 500B tokens.
* Both roughly match T5 (encoder-decoder), which has 770M parameters.
* Sparse model has 32 experts and a sparse layer every 4 layers.
* Train capacity factor = 1.25, 2.0 at eval time.
* Fine-tuned on 2 SuperGLUE tasks, one with 100,000 training examples and the other with 250 to analyze overfitting of sparse models during fine-tuning.

Sparse models are thought of to be more prone to overfitting during fine-tuning, especially when there is little data to work with (the more data, the lesser the risk of overfitting). This is observed on the smaller task (250 training examples), where the sparse model performs better against the training set but worse in the evaluation set (classic overfitting). This does not happen in the larger task (which has 100k training examples), where the sparse model performs better than the dense one on both the training and evaluation sets. This leads us to the conclusion that sparse models have fine-tuning advantages if enough data is available to prevent the model from overfitting. Increasing regularization did not seem to have much effect at the small-scale fine-tuning, showing that small amounts of data are hard to overcome in this scenario.

To explore fine-tuning MoEs further, the authors experiment with exclusively updating a few layers while keeping the remaining layers frozen. They test this for different combinations.

* Most combinations yield similar results, except one -> only updating sparse layers, which resulted in degraded performance.
  + This indicates that the overfitting comes from sparse layers (although updating all parameters leads to better performance than updating all non-MoE parameters only).
  + Another explanation for this can be the frequency of sparse layers being too sparse (only 1 sparse layer for every 3 dense layers), so the number of parameters being updated is not large enough.

Another fine-tuning aspect analyzed was the batch size and learning rate of dense vs sparse models. Experiments showed that they do not respond the same way to changes in these training decisions.

* Sparse models benefit from smaller batch sizes and larger learning rates, while the opposite is observed for dense models.
  + The range of batch sizes used was 65K to 1M, and the range of learning rates used was 1e-4 to 1e-3.
  + This is consistent with the overfitting hypothesis proposed for MoE models, as smaller batch sizes have less accurate gradient updates (the higher the number of inputs used for an update, the more accurate the update). This reduced accuracy can be thought of as added noise, which serves as regularization, helping with overfitting.

Load balancing is seen as a key challenge for effectively training and fine-tuning sparse models to optimize for modern hardware and prevent token dropping (expert overflow). Experiments conducted on this topic, however, contradict this (for fine-tuning):

* The percentage of tokens dropped (up to 15%) did not seem to have a significant impact in fine-tuning. So, token dropping in fine-tuning does not seem like a problem.
* High capacity factors used during fine-tuning do not seem to have an impact on model quality.
* The addition of an auxiliary load balancing loss seems to have very little impact on fine-tuning.

In terms of the number of experts to choose, works like Switch Transformers show that a large number of experts (up to 512) can improve model quality if designed correctly (although at a diminishing rate). However, to ensure hardware efficiency, an important constraint is necessary: each GPU/TPU core available should have a maximum of 1 expert to minimize memory transfer costs. The main reason for this can be attributed to modern hardware not being optimized for loading parameters to memory. If more than 1 expert is present in a core, whenever the other expert gets called, all experts in that core need to be loaded, leading to inefficiencies.

Deciding on CF (capacity factor) and k in top-k routing should depend on memory and computational resources, as it leads to performance boosts but at the expense of increased costs.

Results

The ST-MoE 32B (32B active parameters, 369B total parameters) becomes the new SOTA in the SuperGLUE tasks (it was trained on all the tasks concurrently).

A 4.1B sparse model designed to match the FLOPs of T5-L (800M active parameters) shows improved performance on all fine-tuning tasks except the two with fewer training examples (around 250 each) -> more signs of MoEs being prone to overfitting.

Finally, an observation was made that upon analysis, the encoder layers generally show specialization in areas such as punctuation, verbs, numbers, names, etc. While decoder expert layers do not show specialization.

My takeaways:

* Router z-loss seems to be a helpful loss function in terms of stabilizing pre-training of MoE models in a mixed-precision environment.
* As shown by the experiments with fine-tuning MoEs, the performance issues seem to come from the scenario where there is a lack of data to use for fine-tuning. If there is enough data available, fine-tuning MoEs obtains better performance than fine-tuning dense models.
  + This can perhaps be explained to the distribution shift fine-tuning data brings in comparison to pre-training data. Naturally, a few experts will be more suited to the fine-tuning data, and thus will be used more than others, leading to overfitting of the more commonly used experts and underfitting of others.
* Although the fact that only updating sparse parameters during fine-tuning leads to worse performance, it would be worth exploring further if this can be caused by the lack of MoE layers present (only one for every 3 dense layers).
* It is also interesting how MoEs benefit from smaller batch sizes, which add a regularizing effect (adding strength to the claim that MoEs are prone to overfitting).
* Experiments done for this paper show that load balancing is not an issue for fine-tuning, which makes sense since the router should already be fully trained. This seems to indicate that routers can be frozen during fine-tuning.
* It is explained how, although having many experts may lead to performance boosts, a balance needs to be achieved depending on the number of GPU/TPU cores available. Loading more than 1 expert per core leads to inefficiencies.
* Deciding on CF (capacity factor) and k in top-k routing should depend on memory and computational resources, as it leads to performance boosts but at the expense of increased costs.

### Unified Scaling Laws For Routed Language Models

Main idea: this paper investigates the scaling behaviors of routing networks, more specifically in the axis of parameter count (in terms of total number of parameters) and computational requirements (total number of active parameters).

Routing:

It experiments with 3 different routing techniques:

* An approach based on BASE (linear programming).
  + This represents a more traditional learned algorithm for routing. BASE in specific approaches routing as a linear programming problem, which naturally distributes tokens evenly through experts (no load balancing issues). The algorithm experimented with has slight modifications to BASE to be more efficient in accelerated hardware (they call it Sinkhorn-BASE).
* A non-parametric approach (hash layer).
  + HASH layers approaches routing as a fixed function of the input, meaning it does not have learnable parameters.
* A Reinforcement Learning approach.

Results:

* Although routing (sparse) performs better than no routing (dense) on all sizes experimented with (up to 1.3B active parameters, up to 512 experts – biggest model has around 200B parameters), the sparse gains over dense are diminishing with scale (BASE is more robust than other routing techniques).
* Scaling the number of experts when the number of active? parameters is fixed improves the validation loss during pre-training.
* Effective Parameter Count (EPC) is created to compare the performance of dense against sparse models based on an equation that considers the total number of parameters and the active parameters of a model.

Main takeaways (as listed in the paper):

* Routing improves performance across all model sizes and routing strategies (compared to dense aka no routing).
* RL routing is more effective than expected, although BASE is the best performer.
* Performance can be described by scaling the number of experts and dense model size.
* Development of an effective parameter count mapping for performance vs scaling.

Recommendations:

* Use routing when training any model with N (parameter count of base model) <= 1.3B.
* Sinkhorn-BASE is a good default routing algorithm.
* Although more experts lead to improved performance, it is recommended to use between 64 and 128 experts due to diminishing returns above that range.
* It is recommended to use k=1 experts.

My takeaways:

* Shows that learned routing (represented through BASE) is the best routing strategy.
  + Non-parametric routing can be used in cases where there might not be enough data to train specific experts (for example, on task/domain-level MoE like DEMix where we are not certain if the training load for each expert will be similar, which will lead to load balancing issues that cannot be solved through traditional auxiliary loss or adding noise – this might not happen at token-level routing)
  + RL routing performs worse than BASE but looks to not be too far off
* To describe performance, the number of experts and dense model size (number of active parameters for each forward pass) are the most relevant features. This is logic as the number of experts represents the horizontal scale of the model while the dense model size represents the vertical scale of the model. (dense model size corresponds to vertical scaling, does number of experts as mentioned here correspond to an increase in the number of experts with the same total parameter count or is this accounting for an increase in the total parameter count coming from the added experts).
* Sparse models seem to be the most useful at small scales, with diminishing returns over dense with an increase in the scale of active parameters, but this can be prevented to a certain extent by choosing a robust routing strategy.
* The result arrived at that scaling the number of experts when the number of active parameters is fixed is logical as this scales the model horizontally. However, my intuition in this is that scaling the number of experts might make things difficult for fine-tuning (more data will be needed to update all experts while not overfitting on others). Therefore, a balance is needed. (authors recommend between 64 and 128 experts due to diminishing returns in increasing the number of experts). -> how does fine-tuning performance change with differing number of experts and in respect to more training data to use for fine-tuning (explore how scaling the number of experts while keeping the active parameter count constant impacts fine-tuning performance)?
* The EPC equation seems to be useful for practitioners looking to train a MoE model from scratch. This would help with design choices in number of active parameters and number of total parameters.
* Interesting how the authors recommend MoE in scenarios of training smallish models (up to 1.3B). I believe that this is because that was the bigger dense model studied, so it is not saying that dense models perform better when scaled above 1.3B, but just that a bigger dense model was not used in the experiments. It is important to note that the experiments showed diminishing returns for routing models -> did any other papers dive into this question?
* It is also interesting to note that the authors concluded that k=1 experts is the ideal number for k.

### 

### Efficient Large Scale Language Modeling with Mixtures of Experts

Main Idea: this paper has the goal of comparing how the traditional MoE architecture from “Sparsely-Gated MoE”, using top-2 routing, scales in relation to dense models.

Model sizes trained for this experiment range from (in total number of parameters):

* 125M to 13B (in a dense setting).
* 15B to 1.1T (in a MoE setting).

The maximum number of experts used was 512, and the capacity factor used for MoE models was 2 (to support top-2 routing).

Dense and sparse models were compared on a FLOPs-matching basis (models with the same FLOPs are comparable). The dense baseline used was GPT-3.

Evaluations done:

* Perplexity (from next-token predictions).
* Performance on downstream tasks (benchmarks, zero-shot, few-shot).
* MoE speedup factor – how much more efficient MoEs are at achieving a specific performance level relative to dense models (how many training FLOPs are needed to reach a certain performance goal).

Results:

* MoE outperforms dense in all evaluation datasets, although at a different scale depending on the dataset’s domain and model size.
* MoEs are the most efficient (highest speedup factor in in-domain tasks), reaching an 8x to 16x speedup (8x-16x less compute needed for the same performance)
  + This speedup decreases to a 2x-4x speedup in out-of-domain tasks.
* The speedup advantages of MoE decrease at scale, especially in in-domain tasks.
* The closer the data used for evaluation is to the training corpus, the larger the speedup obtained by MoE.
* On downstream zero-shot task evaluation, MoE also outperforms the dense model (which performs on par with GPT-3), but this gain is, again, diminishing at scale.
* In a few-shot setting, MoE still outperforms dense, but the MoE improvements over zero-shot are smaller than dense. This indicates that although MoE still outperforms dense in a few-shot setting, dense models benefit more from few-shot examples.
* In terms of fine-tuning, dense models (as expected) always incur substantial gains. Although this is true in some cases for MoE, fine-tuning MoE models on some domains/datasets leads to worse performance. More research is needed to determine why. Perhaps this comes from fine-tuning MoEs not being deeply explored yet, with an alternative approach needed to obtain good results (the same setting as dense was used for fine-tuning after all).

My takeaways:

* The results from this paper’s experiments show that the traditional MoE architecture does indeed provide speedups over a dense setting. The results from the speedup provided by MoE are bigger the closer the evaluation domains are from the training domains. This seems to indicate that the biggest gains from MoE come from memorization. Generalization gains provided by MoE over dense are not as apparent, although there still are gains (MoE still provides a speedup when evaluated in out-of-domain tasks).
  + The diminishing gains from MoE at scale are more apparent in out-of-domain tasks, as they stay relatively constant when training domains (or close to) are used for evaluation.
* It is interesting to note that few-shot has a bigger effect on dense performance than on MoE performance (dense benefits more), although MoE outperforms dense in this scenario.
* A previous work, ST-MoE, concludes that sparse models benefit from smaller batch sizes and larger learning rates during fine-tuning, while the opposite is observed for dense models. ST-MoE also concludes that MoEs are significantly more prone to overfitting during fine-tuning compared to dense. The fine-tuning results from this paper can be replicated and analyzed with these two aspects in mind as future research.

## **Task/Domain-Level MoE**

### Beyond Distillation: Task-level Mixture-of-Experts for Efficient Inference

Main idea: the goal of this work is to find an alternative method to distillation to store MoE models. It experimented with token-level, task-level and sentence-level routing. MoE solves the issue of training efficiency when compared to dense models (since only a subset of the network is activated at a time) (tradeoff of a few more communication costs due to experts’ communication and routing but less parameters needing to be updated per forward pass compared to a dense model of the same size in terms of total parameters) but still leaves room for improvement in inference efficiency due to the requirement of storing the model across many devices, adding to communication costs and idle resources for calling small batches (since in small batches, most machines will not be used since the respective expert is not needed). This paper’s main goal is to improve inference efficiency for sparse MoE models. Distillation is a possible solution but tends to lead to loss in quality. The task used for experiments was a multilingual machine translation task.

Approach

* Trained a routing strategy to leverage global task-level information to route all tokens corresponding to a particular task collectively to the same set of experts.
  + Decode different tasks separately and only load the subset of experts associated with the corresponding task during inference.
* Task-level routing strategy showed gains over a dense model trained from scratch and a distilled model (student) trained from learning through a token-level MoE teacher model.
* Comparable quality to token-MoE model (not distilled) while achieving significant inference gains (1.9x peak throughput and 6.3% of the decoder size).
* Top-2 routing mechanism used.

Routing Strategies Experimented With

* Token-level.
  + Traditional MoE where each token is routed independently.
* Sentence-level.
  + Route tokens by sentence, determined by the expert with the highest average token weight in the sentence.
  + First thought is that this won’t work well due to the average token weight per expert is used (this is proven to be correct by experiments done later in the paper). A better sentence-level approach could be to use sentence embeddings, which would also only need to call the router once per sentence.
* Task-level.
  + Route tokens based on a task. In the multilingual translation task, this can be determined by either the target language or the language pair.

Inference Implications

* The token-level and sentence-level approach makes inference costly. To help with the challenge of needing to have all experts ready and loaded to the server at inference, these approaches can have experts be dynamically loaded based on the routing decision or model parallelism can be employed (the server often needs to load all experts). Both incur high communication costs.
  + This needs to be done for every then, hence the high cost.
* Task-level routing only need to pre-load the top-k experts for the given input sequence. This is done by determining which task most resembles the input sequence and using the top-k experts for that task only for all tokens.
  + Loading experts only needs to be done once for each input sequence.

Results

* Sentence-level MoE did not perform well.
* The best encoder-decoder model used had a token-MoE in the encoder and a task-MoE in the decoder.
* The best decoder-only model was the task-MoE decoder.
* Statically determining the task through a deterministic approach did not work very well (experts are deterministically allocated to tasks).
* Task-level MoE has higher throughput (tokens/sec), uses less decoder parameters and has less communication overhead (or none) compared to token-level MoE.
* Task-level MoE performs better than models distilled from token-level MoE.

Additionally, analysis of the routing decisions shows that at a task level, the experts called in the encoder do not change much, but experts in the decoder seem to naturally specialize in tasks, giving a possible explanation why the decoder-only task MoE performed well.

My takeaways:

* In MoE, there is a tradeoff in training costs compared to dense models. MoE provides less communication costs overall:
  + There is a partial increase in communication costs due to the communication that needs to be done between activated experts and between these activated experts and the router.
  + Overall, however, MoE is more efficient at training due to only a subset of parameters needing to be updated per forward pass (on the MoE layers, where the bulk of parameters are located). This allows MoE to scale the total number of parameters in an easier way.
* The inspiration for the approach used comes from trying to decrease the cost of storing experts during inference.
  + This is a necessary step as all experts need to be ready to be called during inference, which leads to idle resources (no experts being used for some batches but needing to be stored and ready).
  + Distillation is (was) the most common approach for this, but distilling experts tends to lead to significant loss in quality.
    - Distillation consists of training a small dense model (student) from a large MoE expert (teacher).
* The gains obtained from inference efficiency do not come from calling less parameters at inference (number of active parameters), but from the number of experts being loaded (number of total parameters available).
  + The idea seems to be to predict the most relevant experts that will be needed on a task level, so only those need to be loaded and ready during inference.
  + The meat of this approach is to correctly predict the experts needed. If this prediction is correct, the model will have good quality, otherwise it won’t.
    - The approach seemed to work since the quality of the resulting model was comparable to token-MoE.
  + This ends up reducing the latency costs since the experts used only need to be loaded once per input sequence, and not for every token.
* The task-level approach seems to be useful in some scenarios but not possible in others. For example, if an out-of-domain task is shown at testing that is different than the training tasks, my intuition tells me that the router won’t be able to select the most relevant experts very well (and the experts won’t be prepared for this situation), thus leading to the model underperforming a token-level approach, which I believe would be more robust to these situations.
  + This approach sounds interesting in a scenario where there are predefined tasks that we want the model to perform well on, and it does not necessarily need to perform so well on out-of-domain tasks.
  + This should be considered when choosing between the task-level MoE and a distilled student model (the student model, in theory, would perform better in terms of generalization – not as good in a few tasks, but good in everything -, while task-level MoE would probably perform better in specific tasks scenarios – especially good at a few tasks (depends on training)).

### Expert Gate: Lifelong Learning with a Network of Experts

Main idea: this paper experiments with a novel approach for using MoE in a multi-task setting. More specifically, it focuses on the gating mechanism used. Expert Gate also focuses on scalability, as it is a lifelong learning approach (can be scaled with time). A lifelong learning approach means that:

* Models are trained sequentially.
* No need to store the data used for training, only the models.

Expert Gate is trained on image classification and video prediction problems, but could technically also be used in an NLP/LLM setting (but was not experimented with)

Advantages of Expert Gate

The meat of this method is in the autoencoder gating mechanism used. This mechanism solves problems as:

* Data storage, since the models can be trained sequentially, so keeping all training data is not necessary.
  + Later the paper will show that storing training data used previously is not necessary.
* Catastrophic forgetting, which is an issue other models suffer with. For example, continuously training and fine-tuning the same model on new tasks will lead to this issue.
* Task biases when fine-tuning which can lead to suboptimal local minima.
  + If a model is trained on a task and fine-tuned on a widely different task, it can lead to suboptimal results due to the biases inferred in the initial task being different.
* Memory efficiency, as only one expert needs to be loaded into memory at a time.
* Task relatedness, which can be measured by the autoencoder’s results and then be used to figure out how to initialize the expert’s parameters for a new class and either to use fine-tuning or learning-without-forgetting (LwF) for training the new expert.

LwF vs Fine-tuning

When two tasks are sufficiently related (above a certain task relatedness in threshold), it is beneficial to train a new expert with LwF based on an old task, otherwise the best approach is to fine-tune the expert for the similar (existing) task on the training data from the new task.

* Fine-tuning
  + Based on an existing model, simply continue training using a new dataset
  + The result of this fine-tuning on an existing expert will be a brand-new expert, while the existing expert that it was based on will remain unchanged.
    - So, this process starts with 1 expert and ends up with 2 experts.
* LwF
  + Technique used to prevent catastrophic forgetting when training an existing model on new data. LwF uses soft targets (outputs of the old model) to help retain knowledge from old tasks.
  + As with fine-tuning, this results in 2 experts.

Autoencoder Mechanism – Expert Gate Inner Working

Goals:

* To select an expert based on input data.
* To measure task relatedness to figure out optimal parameters to initialize an expert (based on most related task) and training strategy (fine-tuning or LwF – LwF in the case of task relatedness being above a certain threshold).

The Inner Workings of the Autoencoder

* It follows a regular encoder-decoder architecture.
* Encoder , maps the input x to a code h(x).
* Decoder , maps the encoder’s code (h(x)) to a reconstruction of the input.
* The autoencoder simply uses an encoder-decoder architecture to deconstruct the input (done by the encoder) and reconstruct it (done by the decoder).
* The loss function is simply the reconstruction error.
* The encoder learns, through a hidden layer, a lower dimensional representation (undercomplete autoencoder) or a higher dimensional representation (overcomplete autoencoder) of the input data.
* The lower dimensional subspace learned by one of the undercomplete autoencoders will be maximally sensitive to variations observed in the task data but insensitive to changes orthogonal to the manifold (it represents only the variations that are needed to reconstruct relevant samples)
  + The autoencoder of a domain/task should be better at reconstructing the data of the task it was trained on better than the other autoencoders.
    - The reconstruction error for each autoencoder then allows the input to be routed to the expert of the task of the autoencoder with the lowest reconstruction error for that input (or multiple, in the case of multiple very good autoencoders for that input).
    - The reconstruction error then acts like a score (all reconstruction errors are passed through a SoftMax to determine a normalized score).
* The task relatedness between two tasks is also measured through the autoencoder’s reconstruction error through the following formula:
  + - = new task. = old task.
    - is the relatedness between task k and task a.
    - is the reconstruction error of the autoencoder for task a in the data for task k.
    - is the reconstruction error of the autoencoder for task k on its own data.
    - How can the reconstruction error of the autoencoder for task k on its own data be computed before the expert (and thus its autoencoder) is trained, since its initialization method relies on this task relatedness computation? This seems redundant.

Experiments Results

* Expert Gate was compared with and outperformed (on image classification):
  + Single fine-tuned model (sequentially fine-tuned on each task).
    - One would think that this would result in severe catastrophic forgetting.
  + Single LwF model (sequentially trained on each task).
    - One would think that you can’t train the same model with LwF forever on many different tasks without running into catastrophic forgetting issues.
* Expert Gate performed on-par with:
  + Joint training (assumes all is always available for re-training).
  + Multiple fine-tuned models (fine-tuned on each task separately)
    - This assumes an oracle gate, that is, a gate that knows perfectly how to route each input to the corresponding expert.
  + Multiple LwF models (trained on each task separately).
    - Also assumes an oracle gate.
* Expert Gate vs Discriminative Classifier (neural net trained on all the data available for gating decisions – a routing mechanism).
  + Without ever having simultaneous access to the data of different tasks, Expert Gate based on autoencoders manages to assign test samples to the relevant tasks equally accurately as a discriminative classifier (which assumes all training data is available).
* Task relatedness analysis
  + Expert Gate succeeds in predicting when a task could help another in the LwF framework and when it cannot (LwF vs fine-tuning decision).

My takeaways:

* This is an interesting point to take note of when thinking of a problem related to fine-tuning, especially when fine-tuning MoE.
  + Task biases when fine-tuning which can lead to suboptimal local minima.
    - If a model is trained on a task and fine-tuned on a widely different task, it can lead to suboptimal results due to the biases inferred in the initial task being different (think that the pre-training distribution shift can lead to local minima that is optimal for that distribution, but distribution of new tasks can be different and gain from other local minima that are unreachable due to the pre-training local minima – imagine the gradient descent valley)
* Expert Gate seems like DEMix.
  + Expert Gate focuses on the LwF or fine-tuning decision when being presented a new task, DEMix focuses more on the modularity of each expert.
  + Expert Gate focuses on task-level experts while DEMix focuses on domain-level experts.
  + Expert Gate experiments on computer vision tasks while DEMix focuses on NLP tasks.
* Both LwF and fine-tuning lead to the existing expert that was further trained with LwF or fine-tuning remaining unchanged while also creating a new expert. So, 2 experts are a result of this process (one old, one new).
* Both the routing to determine the similarity of an input with the tasks reflected in the existing experts and the task relatedness are determined by an autoencoder mechanism which is independent for each expert (it is trained as the expert is trained).
* The LwF method seems to be fine-tuning with a twist – instead of only fine-tuning with hard targets from the new data, fine-tune is done by considering the new data and soft targets given by the existing expert.
* Run through the methodology:
  + This method is a task-level MoE – it has the advantage of only routing the input sequence once. Since this is done at the beginning of inference, the selected task experts can be pre-loaded to memory and the routing does not need to be performed again, saving on memory costs of loading different experts for every new token.
  + Each task expert consists of the expert itself and an autoencoder, which is used for two things:
    - Determine the similarity of an input sequence to the task (how well does the task expert fit into the input sequence).
    - Determine the task relatedness between different tasks to help training of new experts.
  + Training new experts can be done in one of two ways:
    - LwF, which uses soft targets of the existing/old model to train a new model based on the new task’s data.
    - Fine-tuning, which fine-tunes an existing/old expert with new data, resulting in a new expert.
  + Expert Gate also has the advantage of not all data needing to be stored on the same place at once for training. Since training can be done sequentially, training data can be used and sequentially discarded, saving on storage costs.
* The autoencoder is simply a function that deconstructs and attempts to reconstruct the input. The logic is that the closer the input is to the training data used to train that task’s expert, the better the autoencoder will be at reconstructing the input.
* In computing the relatedness between two tasks, how can the reconstruction error of the autoencoder for task k on its own data be computed before the expert (and thus its autoencoder) is trained, since its initialization method relies on this task relatedness computation? This seems redundant.

### DEMix Layers: Disentangling Domains for Modular Language Modeling

Main Idea: DEMix, which stands for domain-expert mixture, is a type of architecture that encourages domain specialization. It looks to train multiple feedforward networks that are each specialized in a specific domain, and similarly to MoE, pick one to run during inference, depending on the input space. DEMix layers are modular, meaning they can be mixed, added, removed or used to initialize other layers after initial training. DEMix aims to achieve domain specialization in the sparse layer, while retaining generalization knowledge with shared parameters.

Motivation

Dense training consists of updating all parameters to minimize loss on all the data. This means that it assumes that the model will be able to learn/fit different domains equally. In practice, domains are skewed to domains that are more prevalent in the training data, so models have a hard time generalizing to other domains. Fine-tuning these large dense models can also be expensive and lead to catastrophic forgetting – worsening performance on pre-training domains not represented in the fine-tuning data – since all weights need to be updated. Finally, managing unwanted behavior in dense models is also a challenge.

To help with these issues, a DEMix (modular) architecture is proposed. That is, an LLM with different components that can be modified during inference.

Some Characteristics/Highlights of DEMix

* DEMix is a substitute for an FF layer in the Transformers architecture (every FF layer is replaced by a DEMix layer) and can be conditioned on the input text in cases where the domain is previously known, as well as when the input domain is not known.
* The router used for DEMix is parameter-free and thus not learned, depending on the natural segmentation of the data.
  + Parameter-free probabilistic approach to dynamically estimate a weighted mixture of domains during inference, which is used for novel domains (when it is not clear/known in advance where the input is from, or it is from a brand-new domain).
* Mixing (like using top-k > 1) experts is shown to improve performance in novel domains as well as training domains during test time (probably due to overlap between domains that the shared parameters are not enough to capture).
* The modularity of DEMix offers flexibility by enabling the removal or addition of new domains at inference, thus allowing the ability of choosing what is forgotten. Catastrophic forgetting is also not an issue since a new domain expert can be initialized or an existing one can be further specialized without modifying the model’s behavior on other domains.

Data

* 8 training domains
* 8 testing domains
  + Used to test robustness of mixing experts to data distribution shifts not seen during training.

DEMix vs Traditional MoE

* While in traditional MoE the routing function is learned through training at a token-level, DEMix routing is done at the document (sequence) level and only needs to be performed once per input (all tokens in an input sequence are routed the same way).
  + Token-level routing has been shown to specialize experts in token-level areas, such as semantics and syntax. Document-level routing should enable experts to specialize in specific tasks/domains.
* Because of this characteristic in specializing in domains compared to semantics, the experts are more flexible in terms of addiction and subtraction to the network and provide an ease of interpretation that traditional MoEs don’t have (they are more of a black box).

Training

* During training, each domain expert is assigned to a single GPU (similarly to how it is done in traditional MoE).
* Each mini batch sends k domain examples to each expert (a balanced load is easy to achieve since we know each input’s domain for training).
* Distributed data parallelism is used (expert is replicated through the number of GPUs available for that specific expert, since there were more GPUs available than experts)
  + This is efficient because only globally shared parameters are synced through all GPUs, while domain expert parameters are only synced between the GPUs assigned to that expert.
    - Reduced communication costs due to a decrease in alltoall computations.

Evaluation

* In-domain performance
  + 4 variations used:
    - DENSE – regular dense model with no conditioning on domain.
    - DENSE (balanced) – dense model with equal amount of data used for each domain.
    - +DOMAIN-TOKEN – variation of DENSE (balanced) with a prepended token on every input sequence to indicate its domain.
      * The motivation behind this is to add info about the domain of the input to the context to try to create a dense oracle gate.
    - DEMix – DEMix architecture with known domain for each input.
      * Uses top-1 routing for in-domain experts based on the already known domain of the input.
  + Adding domain info (DENSE (balanced) and +DOMAIN-TOKEN) is shown to help the dense baseline.
    - The smaller the model, the more helpful this is.
  + Heterogeneous domains (diverse domains like WEBTEXT and REDDIT) have more overlap with other training domains, and thus don’t really benefit from DEMix vs a dense baseline.
* Unknown domain performance – mixing experts at inference time.
  + Routing approach
    - In practice, the domain of an input is not always known. In this case, it makes more sense to use a soft choice for routing (top-2 routing), as it was proposed for cases where the domain was known.
    - To not increase training costs with a learned routing approach (more communication costs), a probabilistic routing score based on Bayes’ Rule was used (this is parameter-free).
    - Probabilistic Routing Score:
      * The main part of this is calculating the domain posterior – the probability that the input is from a certain domain d.
      * This approach is very inefficient (the input needs to go through each expert, so the routing is useless in practice) and is improved in future work.
    - They propose 3 variations on the posterior calculation:
      * Uniform - each domain is estimated to be equally likely.
      * Updating - weighted moving average of the posteriors from the previous timesteps.
      * Cached – fixed prior estimated from the test data (100 test sequences used)
  + The estimates of posteriors for both the training and the novel domains is shown to be sparse, justifying the top-1 and top-2 routing selections (so not all experts need to be used, aka sparsity is justified).
  + Ensembling DEMix experts (mixing) using the cached approach performs better than all models analyzed.
    - Compared to DENSE, this is beneficial at smaller scales, while the dense models can catch up as the parameter count increases.
      * Perhaps more data is needed when increasing the DEMix parameters?
    - Ensembling/mixing is also shown to lead to improvements on training domains, especially more heterogeneous ones (more diverse domains).

DEMix-DAPT

DEMix-DAPT consists of adopting existing experts to new domains.

* Previously, experiments were made to evaluate the performance of DEMix in novel domains (domains not seen during training). DEMix-DAPT is different in the sense that it applies new domain data to existing domain experts to create a new expert.
* The new expert is initialized with the parameters of the closest existing domain expert. So, the new expert is a fine-tuned version of an existing domain expert.
  + How close each domain expert is from each other is calculated from the router’s domain posterior.
* In DEMix-DAPT, only the expert parameters are trained. Shared parameters are kept frozen.
* For inference, the cached posterior approach is taken.

Results (DEMix-DAPT)

* DEMix-DAPT is compared to Dense-DAPT, which is a dense version of adapting to a new domain.
* As expected, it is shown that Dense-DAPT suffers from the issue of catastrophic forgetting. This is apparent when seeing how training a Dense-DAPT model in a novel domain leads to degraded performance on (original) training domains.
* As expected, adding experts through DEMix-DAPT significantly improves performance on those novel domains.

In this paper, it was also shown how removing an expert from an unwanted domain (for example, due to hate speech or leaking of private data), leads to similar performance on that domain compared to DEMix models not trained on that domain. This shows that expert domains can be removed from DEMix, if desirable. This also shows that most domain specialization comes from the DEMix layers.

### Branch-Train-Merge: Embarrassingly Parallel Training of Expert Language Models

Main Idea: This paper serves as a continuation to DEMix, focusing more on the aspect of employing techniques to train these modular models more efficiently. Due to the modularity of DEMix, Branch-Train-Merge (BTM) shows that it is possible to train these domain experts independently, saving on multi-node synchronization costs commonly required in the training of LLMs. BTM also explores scaling up the number of experts to 64 (DEMix only trained up to 16).

BTM trains an ELMForest (Expert Language Models for Efficient Sparse Training), which are embarrassingly parallel, that is, different parts of the model are independently trained on different subsets of the data, with no need for multi-node training or inference.

Each ELM is specialized in a different domain with no shared parameters (contrary to DEMix).

ELMs can be added or removed to the model at any time, or parameter-averaged to collapse back into a single LM.

Branch-Train-Merge Algorithm

* The BTM algorithm consists of repeatedly expanding the ELMForest (combination of experts) by adding experts in an embarrassingly parallel manner. There are two possible scenarios: when we are first building the forest (creating the first expert) and when we already have at least one expert created, which makes the process of initializing other experts easier.
* The addition of a new expert is done by:
  + Branch – initializing a new LM with an average of the parameters of the most relevant of the currently existing experts.
  + Train – train this recently initialized expert on new domain data.
  + Merge – merge the trained expert into the ELMForest.
* The first step (branch) needs to be done in a different manner when training the first expert since there are no experts to initialize this expert to. The training of the initial expert is done by training on heterogeneous (diverse) data.

This approach is shown to outperform dense and DEMix when used as an ensemble or when parameter-averaging the weights of the experts. This shows that there are inherent gains from training using the BTM approach.

Overall, BTM shows an efficient way of scaling LLMs without having to train extremely large models. Instead, an ensemble of domain experts, or even a parameter-average, outperforms the dense version. (the models were compared based on GPU training time; the parameter-averaged model is also compute-matched to dense).

In this work, the domains are defined by provenance (source). This is suboptimal and improved in later work.

Like DEMix, BTM has the advantages of fully adding and removing experts, if desired. Since each expert is trained on their own specific data split and there are no parameters shared, this means that removing an expert will lead to complete removal of that domain from the model. The only caveat is if other domain experts were initialized from an undesired domain. In this scenario, simply removing the undesired domain may not be sufficient.

Ensembling and Averaging ELMs

* Ensembling and averaging ELMs keeps the inference cost constant regardless of the number of experts added.
* Ensembling leads to higher inference costs (due to multiple expert results needed), however, results show that top-k routing should be possible.
  + The expert routing (for top-k) or score for parameter-averaging are done through the same domain posterior method from DEMix (with a cached prior, more specifically).

BTM Approach (in more detail)

* BTM can be done asynchronously, that is, multiple new ELMs can be trained in parallel. This can be thought of as having multiple BTM training rounds, each initializing its new experts based on the existing experts at the beginning of the training round.
* Step 0
  + The first ELM needs to be initialized differently, since there are no existing ELMs yet to obtain parameters to initialize an expert from.
  + For this, an initial ELM is trained on heterogeneous (diverse) data.
  + Once this initial ELM is trained, its parameters can be used to initialize the weights of the first batch of the ELMs.
* Branch
  + Refers to adding a new ELM (Expert Language Model).
  + Idea is to initialize the new ELM to be a parameter-average of the current ELMForest (all existing domain experts).
  + The best approach for initialization was to perform a weighted average of existing ELM parameters based on their domain posterior or the new domain data (finding the closest domains to the new domain and only use the parameters of the most relevant experts for this new domain).
* Train
  + After initializing the weights of the new ELM (branching), the ELM is trained independently on its domain data.
* Merge
  + Once the new ELM is fully trained on its domain data, it can be added to the ELMForest.
* It would make sense that the more ELMs exist, the less time new ELMs need to be trained for, since more ELMs means more specialized ELMs, and that the data distribution of the new domain will probably be closer to the distribution of existing domains (since there are more domains to pick from).

Initial Results

* Setup
  + ELMForest trained on 8 domains, one trained at step 0 and the remaining 7 were trained in parallel from the initial domain (only one BTM cycle done).
  + Models compared at a compute-matched basis at training.
  + 3 models used:
    - Dense Transformer - where the data from each domain is balanced.
    - DEMix – domain specialized layer (domain-level MoE).
    - ELMForest – full domain models (ELMs).
* ELMForest provides the best performance on all sizes (up to 1.3B dense), and these hold with scale.
  + However, a full ELMForest ensemble has an increased inference cost.
* ELMForest provides speedups during training (more updates per second).
  + This is justified by the reduction in cross-GPU communication for parameter synchronization (no alltoall operations needed).
* To match inference costs with dense, the ELMForest weights can be averaged. This is experimented through 3 strategies:
  + Uniform – each ELM is given the same weight.
  + Argmax – use only the ELM that is closer to the target data, equivalent to top-k with k=1.
  + Posterior – weighted average between all domains based on the domain posterior score.
  + Uniform performs worse than all other strategies, even dense.
  + Argmax performs better than dense in training domains, but worse in evaluation domains.
    - This is expected since evaluation domains (out-of-domain performance) benefit more from using shared knowledge/parameters.
  + Posterior performs better than all strategies (including dense) except for the smallest model (dense is the best in that scenario).
    - With enough training, Posterior top-k can outperform dense at the 125M scale.
* Even though Posterior parameter-averaging is promising due to improved performance over dense at the same training and inference cost, a full ensemble still provides the best results.
  + The significantly reduced inference cost from Posterior parameter-averaging makes this much more practical.

Further Analysis

* Ablations are made to compare the traditional BTM model with:
  + A random ensemble - same setup but each ELM is trained on a random data split, not on a specific domain. This results in an ensemble of general experts instead of an ensemble of specialized experts.
  + An ELMForest where all ELMs are randomly initialized. This should take away the effect of optimizing the initialization of new experts.
  + These 2 variations led to worse performances, so the ELMForest performance is not simply the result of ensembling parameters.
* Ablations were done to decide on how much compute should be given to the seed training (step 0) – these ablations explain and fix the underperformance of ELMForest compared to dense at the 125M scale:
  + In the initial setup used (8 training domains), the optimal amount of deed training, in relation to the total training budget, was from 40%-60%.
    - For the parameter-averaging approach, the ideal is 60%-70%, and randomly initialized ELMs (0% seed training) do not work well at all (they perform very poorly) in this setup.
  + Although not optimal, reducing seed training down to 10% of the total budget results in gains over dense and randomly initialized ELMs.
    - This shows that ELMForest performance is robust to a wide range of seed LM training compute allocations.
  + More seed training is especially useful for evaluation domains (out-of-domain performance).
* Further ablations were done using different datasets for seed training (using a 50% compute allocation to seed training).
  + The more heterogeneous (diverse) the seed data is, the better.
  + However, performance is robust to the choice of seed training corpus.
    - Even using only JavaScript code for seed training led to better performance than dense.
  + Removal of unwanted ELM domains is also robust to the seed training corpus.
    - Performance on removed domains degrades significantly when such domain is removed.

Scaling the ELMForest to 64 Domains

* 64 domains used for training and 16 for evaluation (80 total).
  + 4 BTM cycles are done, 16 training domains for each cycle/batch.
* The dense Transformer used for comparison:
  + 1.3B parameter model.
  + Trained for 6144 total GPU hours (using 128 GPUs).
* The 64-domain ELMForest:
  + Uses seed training of 75%.
  + 4 GPUs per ELM (4x16 = 64 GPUs used concurrently).
  + For BTM cycles/batches 2 and 3, 40 GPU hours were used for each domain, and for batch 4 20 GPU hours per domain were used.
  + The total training cost of training this 64-domain ELMForest was 2565 hours, significantly lower than training the dense model.
* Using only 40% of the dense Transformer’s computational budget for training, the ELM full ensemble (not parameter-averaged) performs comparably to the Transformer (although at an increased inference cost).
  + ELMForest is especially better for training domains since the parameters for each training domain is not updated and thus not forgotten.
* Analysis shows that sparsity in the ELMForest posterior, suggesting a top-k approach can be taken to reduce inference costs:
  + Top-8 routing performs similarly to a full ensemble.
    - This means that the ensemble can be reduced to 8 experts chosen at inference without a loss in quality.
  + Top-1 routing still performs better than the dense Transformer if the Transformer was given the same amount of training.
    - Parameter-averaging performs significantly better than top-1, and almost as well as top-2 (top-2 has double the inference costs).
    - Anywhere from averaging the parameters to condense them into a single LM or using top-2 to top-8 routing seems optimal, depending on the compute available.

My takeaways:

* Future research can include:
  + How to improve the weights taken for parameter averaging of the ELMForest?
    - There is a hot area of research, so different techniques exist.
  + Best practices for scaling and coordinating the training of ELMForests.
  + Combining ELMForests with adapters to scale into smaller domains.
  + Unsupervised domain assignment.
    - A small sampling of training data is required for calculating the domain posterior. Unsupervised assignment would get rid of this.
    - Topic of the next research paper that gives continuation from the research work by the University of Washington – “Scaling Expert Language Models with Unsupervised Domain Discovery”.
  + Recipes for leveraging ELMForests for user safety.

### Scaling Expert Language Models with Unsupervised Domain Discovery

Main Idea: this research work picks up where BTM left off, adding a caveat to the framework – instead of classifying domains based on provenance (source), this is done in an unsupervised manner, assigning domain data based on clusters. The new framework is named c-BTM (cluster Branch-Train-Merge), and it still holds the embarrassingly parallel characteristic of the original BTM.

Pros of Unsupervised vs Provenance-based Domain Classification

* Not all datasets are able to be grouped based on provenance (like internet crawls).
* Groups created by provenance cannot be easily merged or divided, so one ELM is needed for each group. This is not flexible in terms of adjusting the size and number of ELMs.
* Instead of a domain posterior routing approach, which comes with many disadvantages, routing in c-BTM is done based on the distance of a document’s vector to a cluster’s center, a simpler and more effective approach for routing.

c-BTM vs MoE

* c-BTM routes sequences instead of tokens. This allows for different specialization in domains/tasks instead of specializing in semantics/syntax because of the routing being done at a sentence/document level, not at a token level.
* c-BTM uses offline balanced clustering (size of the clusters can be adjusted to achieve load balancing) compared to online load balancing.
* c-BTM has no shared parameters, which leads to savings in communication costs and results in a highly efficient framework for training domain experts.
* c-BTM has more interpretable experts.

OBS: c-BTM is directly compared to sparse upcycling, which mirrors how c-BTM initializes experts but instead of training ELMs, sparse upcycling substitutes the feedforward networks in the dense checkpoints by MoE layers.

c-BTM Algorithm

* OBS: this paper only focuses on using 1 iteration/cycle for c-BTM, meaning training all ELMs from the seed ELM (no cycles trained based on existing specialized ELMs.
* Step 0: Cluster
  + K-means clustering, with enforced balanced clusters (during training, not inference), is used during training.
  + Tf-idf is used since it was the best performing embedding approach.
* Step 1: Branch (from seed LM)
  + The seed LM is trained on a diverse corpus – experiments can be found at the BTM paper.
  + Done the same way as in BTM.
* Step 2: Train
  + Done the same way as in BTM.
* Step 3: Merge
  + Done the same way as in BTM.

Inference

* Sparse ensemble of outputs of existing ELMs - router chooses top-k ELMs and a weighted average of those ELMs is used.
* The router is fixed at inference and does not need to be updated after training.

Experimental Setup

* OPT is used as the seed LM (the dense checkpoint).
  + Both the 1.3B and the 6.7B versions of OPT were experimented with.
* K between 2 and 128 for k-means clustering was experimented with to evaluate the optimal number of ELMs.
* Dropout of 0.1 is used for all layers except the embedding layer.
* Using only the second half of a document from the embedding-based routing is shown to not result in a performance drop while leading to faster inference.
* Models are compared against each other in a compute basis. Using total training parameters is said to be misleading for sparse models, so it is not used.
  + Total GPU-time is used to evaluate training efficiency.
  + GPU-time needed for inference and latency for end-users are used to evaluate inference efficiency.

Results

* Controlling for total training tokens:
  + Using a single cluster (dense) is always the worst setup.
  + Increasing cluster count in c-BTM improves language modeling performance for a fixed compute budget (up to 16 clusters experimented with).
  + The advantage of c-BTM only increases with an increase in the number of total training tokens.
  + There is a range of optimum number of training clusters and this increases with an increase in total training tokens.
    - It is cheaper to train on more clusters in parallel, so there could be a tradeoff there.
    - This is consistent to scaling up the size of each ELM.
* Comparing training time:
  + Due to c-BTM models with higher cluster counts using fewer GPUs per expert under a fixed budget and having no communication costs between experts, c-BTM models with more clusters can be exposed to more data for the same amount of time as dense models.
    - The more clusters, the faster the training updates.
  + Training on more clusters, due to the small number of GPUs per ELM and the fact that no communication is needed between ELMs, results in a much more robust training setup to GPU failure.
  + Models trained with more clusters have faster updates as we increase the total compute.
    - Opposite is true for MoE due to communication costs between experts.
* Controlling for inference costs via parameter count:
  + The largest training budget used was 168B tokens.
  + c-BTM with top-1 routing (same inference cost as dense) outperforms dense.
    - The more clusters, the better the top-1 routing performance.
  + Top-2 to top-4 routing (of c-BTM) pretty much matches the performance of a full c-BTM ensemble (better parameter efficiency than regular c-BTM, which was top-2 to top-8).
    - Top-2 to top-4 routing sometimes even perform better than a full ensemble.
  + These conclusions hold true even when scaling the ELM count to large values (128).
* Comparing to a larger dense model:
  + 6.7B parameter dense model vs 1.3B parameter ELM c-BTM with 16 clusters and top-4 routing (5.2B inference cost) (1.3B latency cost – since the parameters of the ELMs can be run in embarrassingly parallel fashion).
  + c-BTM has a 3.5x speedup in training (using 168B training tokens).

Downstream Task Results (few-shot results)

* 6 classification tasks experimented with; 8-shot evaluations used. c-BTM trained on the C4 dataset.
* 16-cluster c-BTM always outperforms its 1-cluster version (1.3B ELM size).

Comparing to MoE (sparse upcycling aka MoE from a dense checkpoint)

* Sparse upcycling was shown to be unstable with a high number of experts (64 and 128). With 32 experts, it was shown to have stable training and perform better than the higher expert-count stable runs.
  + Based on this, the sparse upcycling model used here was a 32-expert MoE with top-2 routing.
* MoE has more expensive training due to top-2 routing.
  + Shared parameters need to communicate with each other, resulting in slower training.
* Sparse upcycling performs better at small compute budgets but performs much worse at larger budgets, even performing worse than dense models.
  + Authors speculate this could be due to distribution shifts after pretraining, which might increase the instability of sparse upcycling.

Final Analysis

* Clustering is important as random clustering underperforms it significantly.
* The load balancing constraint in k-means is shown to be useful.
  + This becomes more important with an increase in the number of clusters.
* Using the tf-idf approach, an analysis of important terms in clusters point to cluster specialization. Further analysis also shows that ELMs successfully specialize in their own cluster.
* It is shown that metadata may not correspond to the most optimal segmentation of the corpus (although it is more interpretable).
  + Since c-BTM performs better than regular BTM, with the only significant change being how the segmentation of data is done.
* Future research may investigate improving the technique to merge model weights (as this is a hot area of research).

My takeaways:

* Regarding the relatively low dropout used for the training of ELMs, I believe that these are more robust to overfitting than traditional MoEs due to ELMs being full networks, and thus having more parameters than a single expert FF.
  + The fact that k-means has a constraint to ensure the loads are balanced between ELMs at training time might also help with overfitting.
  + On a similar note, ELMs seem to benefit from larger batch sizes. This is also a sign that ELMs would be more robust to overfitting since the opposite is true for MoEs.
    - Larger batch sizes = more accurate updates (less noise) = less regularization effect.

### Exploring the Benefits of Training Expert Language Models over Instruction Tuning

Main Idea: this paper looks to explore the author’s discovery that training an expert LM fine-tuned on a single task can outperform a multi-task (MT) LM trained on hundreds of tasks (more specifically regarding multi-task performance). This goes against other findings that claim that scaling the number of tasks in MT-LMs is key to making them have stronger performance. Referring to LMs fine-tuned on a single task means a system of multiple Expert Language Models (ELMs), each fine-tuned on a single task, not a single LM trained on a single task.

OBS: Instruction-tuning -> fine-tuning LMs with instructions (prompts).

ELM Framework

* Training experts – two types of experts are trained:
  + Prompt Experts
    - Trained via PEFT through an adapter (an adapter layer is trained on top of the pre-trained LLM, with the pre-trained LLM’s weights kept frozen).
    - Trained to perform well on a single prompt specific to the task.
  + Dataset Experts
    - Trained via regular fine-tuning of the pre-trained LLM’s weights on a single task (all weights are updated).
    - Idea is to train an expert that will perform well to different prompts, so it can merge with other experts.
* Routing mechanism – Retrieval-of-Experts
  + Consists of constructing an Expert Library and training a dense retriever.
    - Each row in the Expert Library corresponds to an expert and consists of keys of S random training instances of that expert and a corresponding expert id.
      * S used was 100.
    - The dense retriever is a Sentence Transformer, and it also assumes that Q examples of the target task are available. It takes the embeddings of the input task and chooses the most relevant expert(s) based on each expert’s similarity to this input task (based on the training instances stored for each expert and the target task instances).
      * Q used was 32.
* Merging of experts
  + The merging of Dataset Experts is also explored, retrieving more than one expert for an unseen task.
    - Merging does not make sense with Prompt Experts, since they were trained to perform well on a single prompt, therefore they would not be performant at this setting (at merging).
  + The merged LM ends up being created at the parameter level. It is a weighted-average (parameter-average) of the selected experts.
    - Since the parameters are merged, the inference cost will be the same as the inference cost of the single MT-LM trained on hundreds of tasks.

Experimental Setup

* 296 Prompt Experts, 36 Dataset Experts (on around 8 prompts each) trained.
* 50,000 samples used for training each classification task. 10,000 for each generative task.
  + On top of the pre-trained T5 model.
* 5 epochs used for training with a constant learning rate of 1e-4.
* Rouge-L score used for evaluating generative tasks.

Results – Prompt Experts

1. A single Prompt Expert significantly outperforms its dense MT-LM baseline (trained on hundreds of tasks).
   1. The single Prompt Expert that achieved this was trained on CosmosQA.
   2. Perhaps this means that the dataset being diverse is more important than the number of tasks trained?
2. The Retrieval-of-Experts (ROE) method with an oracle gate significantly outperforms all other models, including T0-11B (the base LLM used for the adapters was T5-3B) and GPT-3.
   1. This shows that improving the retrieval method is a promising area of future research.
3. A simple ROE approach outperforms T0-3B (the MT-LM baseline) on classification tasks, but not on generative tasks.
   1. A better ROE method reverses this.
   2. Using more diverse data (in quantity) seems to help seems to help generative tasks (perhaps due to the higher complexity in text generation compared to classification?).

Results – Dataset Experts

1. There was negative task transfer when merging the adapter experts (Prompt Experts).
   1. Merging Prompt Experts results in worse performance – does not work.
2. Merging the fully fine-tuned experts (Dataset Experts) resulted in positive task transfer.
   1. Merging resulted in improved performance (merged capabilities > individual capabilities).

* The 3 datasets that show the best performance on unseen tasks (when training on a single task) are all commonsense reasoning datasets (for both merging and not merging).
  + Points to models trained on commonsense reasoning having higher generalization abilities to unseen tasks – commonsense reasoning data is higher quality data.
* Retrieval of the correct expert(s) seems important as the best expert on unseen generative tasks performed poorly on unseen classification tasks.

Benefits of Expert LMs over MT-LMs

* ELMs are less susceptible to negative task transfer on seen tasks (the tasks used for training).
* ELMs have continual learning abilities on new tasks without needing access to all the data at the same time.
* ELMs allow for merging experts on compositional instructions (merging of task prompts).

Limitations of ELMs over MT-LMs

* The method explored assumes a batch of the target task is available for RoE, which is not always a realistic assumption.
* MT-LMs bigger than 11B parameters, which might not suffer from negative task transfer due to increased capacity, were not analyzed.
* For some tasks, merging experts on compositional instructions may not be so simple.

My takeaways:

* A system of ELMs outperforming a single LM in a multi-task setting seems to show that the benefits of specialization outweigh the benefits of shared knowledge between tasks.
  + An ELM system also allows for choosing an expert trained on a task that resembles the target data – ensemble of closely-related experts sounds, in theory, better than a single LM fine-tuned on multiple tasks (that could be both relevant and irrelevant to the target task).
* More exploration is needed in the Retrieval-of-Experts (routing mechanism used) to alleviate the constraint of having training and target instances stored, as well as to appropriate it to scenarios where we do not have examples of the target task available since this task would be unknown.

## **MoE Efficiency**

### MegaBlocks: Efficient Sparse Training with Mixture-of-Experts

Main Idea(s): MegaBlocks aims to improve the challenges of load imbalance and token dropping in MoE architecture using block sparse matrices. The idea is to present a router that dynamically handles the token allocation to experts. While in a regular MoE architecture each expert is assigned to a single GPU in a fixed allocation system (each expert gets the same amount of compute), having to drop tokens in the case of overflow to a specific expert/GPU, while at the same time padding tokens to compensate for idle computational resources in experts which were not assigned enough tokens in a batch, MegaBlocks makes this allocation dynamically from the start, so the computational resources assigned to an expert is variable, being adjusted on a per-batch basis based on the tokens assigned to the expert on that specific batch.

OBS: Tutel, a previous work, used a similar strategy, by implementing a dynamic CF (capacity factor) for each expert, but this leads to computational inefficiencies.

MegaBlocks is possible by making use of block-sparse matrix multiplication as opposed to batched matrix multiplication. This approach maps efficiently to hardware accelerators and allows for variable expert size and allocation.

MegaBlocks leads to training speedups, which is logical since it makes optimum use of computational resources at each update.

My takeaways:

* MegaBlocks is an approach for maximizing computing efficiency when training MoE models. It dynamically adjusts how much compute to be given to each expert at every batch, preventing token dropping and idle resources. Although this is interesting, per the experiments of ST-MoE, this seems to only be useful at pre-training, as load balancing does not seem to affect fine-tuning much.

### Fast Feedforward Networks + Exponentially Faster Language Modeling

Main Idea: the goal of this work is to introduce a new MoE architecture to improve inference time (up to 6x faster than other MoEs). They also claim that FFF (Fast FeedForward) has better training properties due to noiseless conditional execution (no randomness in the gating function).

* Traditional MoEs scale down inference time but remain linear in the width of the feedforward layer (increase in expert parameters). These models also rely on noisy gating for load balancing, which complicates training.
* FFF, on the other hand, uses a binary tree-like structure to improve on these challenges.

Method

* FFF uses nodes to aid the routing mechanism and leaves for the experts.
* The input representation goes through a first node (which is a common MLP layer). The node’s output is then passed to a sigmoid to give a probability p. This probability p is used to route the input representation into the next node (left or right branch, as in a binary tree). This process is repeated until a leaf node (expert) is reached)
  + The number of nodes the input goes through (in case of hard routing) corresponds to the depth d of the network.
* MoE chooses an expert width e (size of expert) and trains n separate expert blocks by the partially randomized output of a gating network of width g = [w/e]. The target is then predicted based on the mixture of the k best scoring experts.
  + MoE cost of inference is k\*e neurons plus the gating overhead g (g tends to be small).
  + FFFs of depth d learn a tree partition R1, … , R2^d of the input space determined by their nodes, and 2^d small leaf feedforward networks (experts) of width l.
  + FFFs uses a soft routing approach to training, meaning that backpropagation is done by considering the soft routing probabilities p, so training a FFF is more costly than even a feedforward network. However, a hard routing approach at inference (routing only happens through the most relevant nodes) ensures an inference gain over MoE.
    - This soft to hard routing transition is referred to as hardening.
* The FFF routing is more efficient than regular MoE routing.
  + In MoE, a gating network for each expert is needed to calculate the suitability of the specific expert to the input.
  + In FFF, the input is passed through d nodes. Since each node halves the number of experts (leaves) to be considered in future routings for the same input, and because the left/right decision is simpler and thus requires less parameters than a normal MoE gating function, FFF provides a logarithmic routing improvement over MoE in terms of computational overhead at inference. This is especially significant when scaling the number of experts.
* The strategy of soft routing during training comes with the idea that as the leaves specialize, the nodes will be more confident in the routing, leading to probabilities closer to 1 (to the correct path). This process is referred to as hardening. If hardening does not occur at the expected rate, the hard routing required for inference might not work as well. In those cases, a hardening loss is used.
* Localized overfitting can occur with a high number of leaves, with each leaf being responsible for a very small part of the input space. To diminish this, one can add random child transpositions (flip the p scores given by a node to its child nodes randomly), which ensures the gradients are more diversely distributed, and exposing different nodes and leaves to areas of the input space they otherwise wouldn’t see.
  + Hardening can also lead to a shrinking batch problem, mitigated by using larger batch sizes, gradient accumulation and smaller learning rates.

FFFs Applied to NLP

A variant of BERT, deemed UltraFastBERT, is developed, where the feedforward layers are replaced with FFFs.

FFFs provide a forward pass speedup over regular FFs of O(log^2n) compared to O(n), a logarithmic improvement (where n is neurons). This improvement comes from FFF’s balanced binary tree structure, which only executes one branch of the tree conditionally on the input.

UltraFastBERT has 4095 neurons (leaves + nodes) and is compared to a 3072-neuron BERT.

UltraFastBERT only uses 1/341 of its neurons for inference while BERT uses all its 3072 neurons.

* This leads to a 78x speedup (not a 341x speedup, as would be expected) due to hardware optimization for matrix multiplication favoring FFs.

Results

* UltraFastBERT performs on-par with BERT on fine-tuning in downstream tasks, with a 78x inference speedup.
* UltraFastBERT shows that only a fraction of parameters of feedforward networks needs to be applied at inference.
* The concept of FFFs can technically be applied to decoder-only models as well.

My takeaways:

* The efficiency gains are a result of instead of passing the input to a routing mechanism which considers all experts, having the router only decide between two experts (sending the input to a specific side of the binary tree.
  + While traditional routing expects the router to choose the specific part of the input space of each expert, this binary tree approach has the router dividing the input space in half at every decision, eventually leading to the desired input space.
* FFF routing seems to be theoretically less expensive, but not allow parallelization (each node decision needs to be performed sequentially), so gains might not be as significant as expected.

### Parameter-Efficient Mixture-of-Experts Architecture for Pre-Trained Language Models

Main Idea: proposes an architecture to make more efficient use of parameters in MoE models by sharing information among experts. Mainly uses matrix product operator (MPO), a tensor decomposition approach from quantum physics to reconstruct the expert layer, then shares parameters from the central tensor (core information) between experts while maintaining specificity through auxiliary tensors (complementary to the central tensor). The intuition behind this approach is to solve MoE’s issue of expert redundancy (different experts learning common knowledge, leading to parameter-inefficiency).

Approach – MPOE

* Core idea is to share the central tensors from the expert layers and enable specificity via expert-specific auxiliary tensors based on the matric decomposition strategy.
  + The final MoE layer would consist of a shared central tensor (looks the same for each expert) and small auxiliary tensors (unique to each expert).
* The central tensor acts like a global parameter – is the same for each expert in a layer.
  + Less total parameters are then needed in total since each expert layer will contain a globally shared tensor for all experts (the central tensor) while retaining expert specificity through auxiliary tensors specific to each expert.
  + Idea is to capture the shared knowledge between experts in the central tensor, and the specialized expert knowledge in the auxiliary tensors.
* In theory, MPOE leads to suboptimal optimization since central tensors are always updated. To stabilize the optimization process, a gradient mask strategy is used:
  + The central tensor is not always updated (determined randomly).
  + Equivalent to a gradient dropout, employed in the central tensor of each MoE layer.
* MPOE is employed on already pre-trained language models (for the matrix decomposition to make sense, the models need to already have been pre-trained, having knowledge to decompose).

Experiments

* GPT-2 (decoder-only) and T5 (encoder-decoder) are used as base models for MPOE.
* 8 experts per MoE layer are generally used.
* Adding MPOE to fine-tune pre-trained LMs in downstream tasks leads to better performance than Switch with a 27.2x parameter reduction.
  + MPOE is especially better at low-resource tasks, indicating that MPOE’s parameter-sharing leads to positive task transfers.
  + The caveat is that MPOE needs an already pre-trained LLM.
* Adding more experts (and thus having more auxiliary tensors) leads to improved MPOE performance.
* MPOE can also potentially work well in a multi-task setting (with task-level routing).

My takeaways:

* DeepSeekMoE is a recent model that was also trained with the idea of improving parameter efficiency by sharing weights of experts to capture common knowledge.
* Also is like sparse upcycling and parameter-efficient sparsity crafting in the sense that it takes a pre-trained LLM and modifies its architecture to have the advantages of MoE.
* This approach is compatible with distillation techniques to further improve inference time.

### Pushing Mixture-of-Experts to the Limit: Extremely Parameter Efficient MoE for Instruction Tuning

Main Idea: this work looks to answer the question “can we leverage MoEs for instruction fine-tuning?”, especially in the context of PEFT methods such as LoRA.

PEFT -> addresses the challenges associated with updating many parameters by restricting weight updates to a limited number of parameters.

Traditional PEFT experimented with were (IA)^3 and LoRA (both add a small number of parameters to the existing model.

* (IA)^3
  + Adds 3 rescaling vectors Vk, Vv and Vff which rescale the keys and values in the self-attention mechanism, and the feed-forward parameters. During finetuning, only the 3 scaling vectors are updated.
* LoRA
  + Optimizes low-rank decomposition of dense layers in LLMs. LoRA modifies the original weights of a layer in the LLM by adding a low-rank approximation of the change required for adaptation.
    - –> normal fine-tuning
    - –> LoRA
  + So, the change Wx to the original weights (Wo) is approximated through low-rank matrices B and A, and the original weights are kept frozen during the fine-tuning process, only requiring updating B and A.
    - The specific rank to use for these low-rank matrices is a hyperparameter.

Extremely Parameter-Efficient MoE

* Leverages lightweight adapters as experts on top of a pretrained model.
* Router used is simply a trainable dense layer that outputs a softmaxed score for each expert based on the input.
* Adds expert layers only for finetuning, and each expert is a PEFT adapter ((IA)^3 or LoRA). The dense model parameters are kept frozen during fine-tuning, so at this stage, only the expert adapters and router will be trained. The experts learn to adapt the dense Transformer layers during fine-tuning.
* Since LoRA and (IA)^3 adapters are linear functions, it is possible to apply soft merging of experts (as in Soft MoE).
* Very efficient in terms of training (fine-tuning) and inference.

Experiments

* Baselines used for comparison:
  + Fully fine-tuned dense model.
  + Standard PEFT methods ((IA)^3 and LoRA with rank=4).
* Ablations:
  + Using sentence embeddings for the router (all tokens in the same sentence activate the same expert) vs token embeddings (experts are activated based on individual tokens).
    - Token routing is better than sentence routing at all levels.
  + Soft routing (output is a weighted mixture of all experts) vs discrete routing (top-k experts with k=1 or k=2 with use of an auxiliary load balancing loss).
    - Soft merging shows significantly better performance in a PEFT MoE setting than top-k routing.
      * Perhaps due to its continuous differentiability characteristics?
      * Not compatible with NLG.
* Results
  + How does PEFT MoE compare to traditional PEFT models?
    - Base model used was T5-3B.
    - PEFT MoE provides a significant performance boost.
      * Performs on-par with full-finetuning, with the largest PEFT MoEs even surpassing it.
      * These effects are shown to be true with scale.
    - Given the same parameter budget, MoV (based on (IA)^3) outperforms MoLoRA (based on LoRA) at large model sizes, but the opposite is true at small model sizes.
  + Increasing the number of experts in MoV and MoLoRA generally improves performance on all model sizes experimented with (700M, 3B and 11B), however, this should be taken with a grain of salt when the number of experts is high (>30), since increasing the number of experts in that case can lead to worse performance.
  + Evaluation of the routing for the last experts’ layer shows that experts are activated at different magnitudes based on the input task for both seen and unseen tasks (during training). This shows that different experts learn different skills (or that different tasks have different data distributions in terms of tokens?).
  + The larger the batch size used in training, the more likely the MoE is to converge to a single expert (the larger the batch size, the less stable the training).
    - Based on PEFT MoEs degrading performance after 5k steps to lie close to the dense PEFT models.
  + A smaller learning rate stabilizes training and leads to improved performance (3e-4 was used, and the range 3e-3 to 6e-4 was tested).

My takeaways:

* Important to note that this is a method to help in the process of fine-tuning, and it is not compatible with pre-training.
* Also seems like sparse upcycling and parameter-efficient sparse crafting.
* Soft MoE is compatible with the approach used in this research because NLG tasks are not explored, only text-to-text tasks and encoder-decoder models are explored.

### Parameter-Efficient Sparsity Crafting from Dense to Mixture-of-Experts for Instruction Tuning on General Tasks

Main Idea: introduces parameter-efficient sparsity crafting (PESC), a technique consisting of transforming a dense LLM into an MoE architecture for added model capacity and making use of adapters to differentiate experts without altering their original weights. This work focuses specifically on instruction-tuning.

The motivation for this work comes mainly from leveraging the idea of sparse upcycling (converting dense into MoE) to improve LLMs’ performance on instruction-tuning, since “Mixture-of-experts meets instruction-tuning: A winning combination for large language models” showed how the MoE architecture is highly effective for instruction-tuning tasks.

Method:

* Sparse upcycling:
  + From a dense pre-trained LLM, transform the FFN layers in each Transformer block into a mixture-of-experts by replicating (copying) the FFN layer n times (n being the number of experts per layer).
  + The other layers of the Transformer block (embedding, attention, normalization) remain the same.
  + Continue pre-training on this sparse architecture.
* PESC:
  + Generally the same as sparse upcycling with a few caveats.
    - The dense to sparse transformation is similar, but instead of replicating the actual FFN layer, PESC initializes an adapter to represent each expert, while the FFN remains the same for each expert.
    - PESC does not continue normal pre-training as sparse upcycling, it only performs instruction-tuning.
    - PESC does not update all experts’ parameters/weights, but only each expert’s adapter instead for parameter-efficiency.
      * This means that we don’t need n copies of the FFN parameters, but instead the equivalent of n copies of the adapter.
    - For constructing the adapter, PESC uses QLoRA.
    - Top-2 routing and auxiliary load balancing loss were used.

Parameter Efficiency Gains

While in sparse upcycling we are trying to optimize Fi (Theta(o)), where this represents the objective function in respect to all experts’ parameters, in PESC we are optimizing expert adapters to approximate Fi(Theta(o)) through ~Fi(Theta(o), w(o)), where w(o) represents the adapters’ weights.

This provides more efficiency in:

* Training costs, since w(o) is significantly smaller than Theta(o).
* Memory costs, since instead of replicating a full FFN layer for each expert, we are replicating an adapter for each expert, which is significantly smaller.
  + Original FFN weights are shared between experts, so only one copy per MoE layer is needed.

Experiments

* The largest PESC model trained was Camelidae-8x34B-pro (38B total parameters, ~34B activated parameters).
* Strong performance of Camelidae-8x34B-pro on benchmarks analyzed when compared to other SOTA chat models (Mixtral-8x7B-Instruct, GPT 3.5, Llama-2-70B-Chat).
  + Especially strong in knowledge and reasoning, math and coding.
  + Comparable overall performance to GPT 3.5.
* Dense vs sparse variations
  + Significant advantage of Camelidae-8x7B over Llama2-7B-Chat and Vicuna-7B, especially in more complex areas (coding and math).
  + Advantages are only amplified in the 10-20B range with Camelidae-8x13B.
  + Strong performance continues in the 30-50B range, with Cameliade-8x34B-pro outperforming the leading sparse model Mixtral-8x7B-Instruct (47B total parameters, 13B active parameters).
  + PESC effectively mitigates the knowledge forgetting issue observed in the instruction-tuning process of Camelidae’s dense counterpart Camel.
  + Increasing the number of experts in the MoE layers significantly improves the model’s performance.
    - Experimented with relatively low number of experts per MoE layer, from 4 to 16.
  + Increasing the number of experts in this approach seems way less costly than with a regular MoE, since we would need to add m more adapters and not m more FFNs.

My takeaways:

* The sparsity crafting idea seems to pretty much be parameter-efficient sparse upcycling applied to instruction-tuning.
* Sounds possible to practically apply this to TinyLlama-1B? Or another model in the 1-3B range.

### QMoE: Practical Sub-1-Bit Compression of Trillion-Parameter Models

Main Idea: this paper presents a technique to quantize/compress large MoE models. More specifically, it compresses Switch Transformer (1.6T parameter model, 3.2 TBs of memory needed) 20x, to only 160 GB (0.8 bits per parameter compared to 16 without any compression). This quantization technique at the Switch scale can be done in less than a day on a single GPU and results in a minor accuracy loss.

MoE: faster inference with the tradeoff of higher memory cost.

Usually, post-training compression techniques only reduce parameters to around 3 or 4 bits. This would not be enough to make MoE practical – giving inspiration to QMoE.

MoE Quantization

* It might suffice only quantizing MoE layers (not FFs).
* Large dense models are more resistant to quantization, so MoE models can be a good target for it (due to increase in scale seeming to relate to better teachers).
* MoE training might be highly resistant to noise.

Main Challenges:

* Memory
  + The quantization process requires data. In the case of MoEs, the data needed is much larger than with dense models, due to the potential large number of experts. This means that it is even more important to have data that represents different parts of the distribution, so all experts are represented.
* GPU utilization
  + Large-scale quantization had previously been applied to dense models, which consists of applying it to single massive individual layers, which is fast and efficient on GPUs. This can be challenging for MoEs as instead of single massive layers there can potentially be many experts.

QMoE Method

For dense part of the model:

* Fetch one sample X, containing a few hundreds of tokens, from CPU to GPU.
* Pass it through the corresponding dense layers to obtain the result Y.
* Calculate and store expert assignments for tokens in Y.
* Send Y back to CPU and overwrite X in B (large buffer).

For sparse part of the model (expert FFs):

* Fetch all individual tokens in B that have been assigned to expert E, denoted by Xe, from CPU to GPU.
* Use these tokens to produce compressed expert E’ (for example, with GPTQ).
* Run Xe through E’ to get Ye’.
* Send Ye’ back to CPU and overwrite Xe in B.

In sum:

* The dense part consists of passing a set of samples X from CPU to GPU, performing a forward pass through only the dense layers, calculating the expert assignments of each input in X to know which experts were used and then storing the outputs of the forward pass and expert assignments in Y, passing it back to the CPU (in QMoE, the dense parts of the model are left uncompressed).
* The sparse part consists of performing a loop through each expert. For each expert, from buffer B, all the tokens in X assigned to that expert are taken, forming Xe. A quantized version of the expert is also formed (through a technique like GPTQ), resulting in E’. Then, Xe is passed through E’, with the outputs forming Ye’. Finally, the results Ye’ are stored in the buffer B, replacing the input tokens Xe.

The method described provides the main compression gains from QMoE but is not sufficient to achieve the goal of 1 bit per parameter established. To achieve this, the authors adopt GPTQ optimizations for the MoE case, GPU decoding optimizations, and more.

Results

* MoEs are shown to be highly robust to quantization as vanilla rounding with ternary precision does not lead to a model collapse.
* Using data-dependent quantization in MoE (method explained) allows 2-bit and ternary quantization with minimal accuracy loss.

My takeaways:

* Due to my lack of expertise in quantization methods like GPTQ, I did not find it relevant to go into this topic more in-depth.

### Fast-Inference of Mixture-of-Experts Language Models with Offloading

Main Idea: this paper investigates a strategy to run large MoE in consumer hardware with limited accelerator memory. It succeeds in running Mixtral 8x7B in a free-tier Google Colab instance.

Method for MoE Generative Inference

1. Encoding the input prompt.
   1. Done in parallel (layer-by-layer).
2. Generate tokens conditioned on the input prompt.
   1. Done sequentially (token-by-token and layer-by-layer).

In other words, step 1 is easy to optimize since we can simply pass all tokens in parallel layer-by-layer. During token generation this is not possible since we need to pass one token at a time, making the offloading challenging to optimize for.

Improvements from this approach

* Caching experts
  + To exploit the fact that previous work shows that activated experts tend to be active for more than one token at a time (common for them to stay active for 2-4 tokens in a row), the experts activated from the previous token can simply be stored in a GPU cache.
* Prefetching
  + With dense models, offloading is simple due to the fixed order of layers to load. This is not true in MoE, so future layers cannot be pre-loaded since they are usually selected based on the previous layer’s output. To help with this, a speculative loading technique is developed based on the heuristic that the previous layer’s hidden state can be a good proxy for the next hidden state (since these hidden states are only updated and not recomputed from scratch). This allows us to predict the next layer’s experts before knowing its hidden state (in case of wrong guesses, the gains are lost since we must load the experts while no computations are being done).
* In terms of quantization, HQQ (data-free) is used for convenience, however, other techniques such as GPTQ could also work. (QMoE was experimented with on Mixtral 8x7B, but loss in quality was too significant due to the 1-bit quantization).
  + Found that ideally experts can be quantized to 3 or even 2 bits and that attention layers should be kept at a larger bit width (16 or minimum 4 bits).
* For expert offloading, a cache of 2 experts per layer is used with 12 GB GPUs and of 4 for 16 GB GPUs. Additionally, 1 or 2 experts per layer are loaded speculatively as soon as the previous layer’s hidden states are available.

Results (on Mixtral 8x7B)

* On the free Google Colab tier, inference speed is of around 2 tokens per second.
* In terms of cache hits, the accuracy to guess the next expert goes from around 0.2 with cache size of 1 to around 0.6-0.7 for cache size of 4.
* For speculative loading the results are even better and show that active experts can be estimated even when 10 layers ahead (that is, using the hidden state of the 10th hidden layer behind it).

My takeaways:

* Able to use this method to experiment with Mixtral in Google Colab.

## **Hybrid Approaches**

### Sparse Upcycling: Training Mixture-of-Experts From Dense Checkpoints

Main Idea: the paper aims to provide an efficient way to train an MoE model from a dense checkpoint (a pre-trained dense transformer) to minimize training costs, that is, provide an MoE training strategy that is cheaper than training from scratch.

* The paper shows that training a MoE from a dense checkpoint outperforms continued dense training.
* Expert-choice routing (with CF of 2) is generally used for the encoder and top-k (with k=2) is used for the decoder.
  + The T5 encoder-decoder model is used as the dense checkpoint.
* Each expert’s weights are initialized as the exact MLP of the dense checkpoint, and the router needs to be trained from scratch.
* The layer-norm, attention, embedding and output layers are copied to the new model from the dense checkpoint.

Results:

* When continuing pre-training, the larger the training continues after the checkpoint, the bigger the advantage obtained by the upcycle model vs a dense model.
  + The continued pre-training is referred to as sparse upcycling.
* When sparse upcycling for language, there are two comparisons made:
  + Upcycle vs dense – upcycle performs better, with continued dense pre-training giving inconsistent results.
  + Upcycle vs MoE – upcycle generally performs better for small computational budgets. When enough computational budget is given (>100% of the initial pre-trained dense computational budget), MoE can catch up and perform better than upcycled models.
* Sparse upcycling is also shown to perform better than warm starting (“dense upcycling”).

My takeaways:

* It sounds like the approach studied takes T5 (encoder-decoder model) and stretches its feedforward layers horizontally (in other words, transforms them in MoE layers). All other layers remain static – assuming the sparse upcycling is only done on the new MoE layers and routing mechanism, while other layers remain frozen during this process.
* The main takeaway of this paper is that it indicates that with enough training computing budget, it is more efficient to train an MoE model than a dense one, and when not much training computing budget is given, the best-performing approach is to train a sparse upcycled model from a dense checkpoint.

### EvoMoE: An Evolutional Mixture-of-Experts Training Framework via Dense-To-Sparse Gate

Main Idea: EvoMoE is a proposed end-to-end framework for training MoE models. The focus of EvoMoE is to deal with the issues of immature experts and unstable sparse gates (instabilities related to early stages of training, the same issue explored in StableMoE), which come from the traditional MoE framework and are harmful to convergence performance. This issue from traditional MoE is thought to come from training a sparse gate from scratch, with randomly initialized weights for both experts and router – impossible to not have router instabilities with this setup. To solve this, EvoMoE proposes starting training with a single expert, and gradually evolving that into a large and sparse MoE structure.

In sum:

* EvoMoE allows the model to warm-up before dividing it into experts.
* The gate starts as dense and gradually sparsifies, allowing it to better understand how to route inputs to experts before it reaches a high degree of sparsity.

Method

2 stages:

1. Expert-Diversify – can be seen as an improved initialization technique.
   1. Start by training a single expert (so the early stages of training are the equivalent of training a dense Transformer architecture).
   2. After T training steps, the single expert is replicated N times to initialize all experts. The initialization of experts from the initial expert can be done in multiple ways: adding random noise to each expert, randomly masking the initial expert’s weights, etc.
      1. EvoMoE adopts the random masking strategy for initializing experts from an original warmed-up expert.
   3. Once all experts are initialized, EvoMoE goes into a standard MoE period with a Dense-to-Sparse (DTS) Gate.
      1. The training of the DTS Gate is what the next stage is all about.
2. Gate-Sparsify – training the router.
   1. The router starts as a dense gate which routes the input to most experts. The idea is that at early stages of routing, the gate is not so good at its task, so would benefit from more dense routing so it can analyze the relevant experts more thoroughly, gaining more information about which experts work better from each input, instead of just using 1 or 2 experts at a time.
   2. As more training steps are done with the router, the better it becomes, so the sparser it can be. So DTS-Gate gradually becomes sparser.
   3. This stage uses an auxiliary load balancing loss.

Experiments

* Baselines
  + Switch - top-1 routing.
  + BASE – linear assignment routing.
  + Hash Layer – hashing-based routing.
  + DSelectK – differentiable routing achieved through smoothing techniques.
  + StableMoE – gate distillation and freezing for routing consistency during training.
* Evaluated on (all with 355M active parameters)
  + Machine translation - encoder-decoder setup.
  + Masked language modeling - encoder-only setup.
  + Language modeling - decoder-only setup.
* Every other FFN layer is replaced by an MoE layer (EvoMoE alternates between dense and sparse FFNs.
* EvoMoE beats other variants on all architectures (encoder, decoder, encoder-decoder) and provides training speedups.
* Both the expert-diversify and gate-sparsify stages are shown to be useful, per ablation studies.
* Compared to GPT-MoE, EvoMoE can provide a 2x training speedup (2x less training samples needed to achieve the same perplexity) as well as a 1.42x speedup in terms of FLOPs efficiency (1.42x less training FLOPs needed to achieve the same perplexity).
  + The sample efficiency and FLOPs efficiency speedups are different because EvoMoE’s routing is dense during some of the gate-sparsify stage, which requires more FLOPs per training sample.
* With increasing number of experts per layer, EvoMoE shows consistent improvements.
* With increasing number of MoE layers (replacing denser FFNs by MoE layers than in the initial setup), EvoMoE shows better performance while maintaining inference FLOPs (although with a higher memory cost – more total parameters).

My takeaways:

* Research Idea – it might not be efficient to enforce load balancing due to some areas of the input space being more common than others (load balancing could cause undesired overlap in clusters at the token-level). Perhaps there could be some synergy between early-stage stability and MegaBlocks (for stable gating + no necessary load balancing at the batch level). Could also explore how custom compute depending on the complexity of the input could be implemented, and how this would perform.
* Overall, EvoMoE shows promising results. The challenge to this framework is the dense routing stage of training, which incurs high compute costs, but is a part of the trade-off for achieving better routing stability and sample efficiency.

### Soft Merging of Experts with Adaptive Routing

Main Idea: develop a technique called SMEAR (Soft Merging of Experts with Adaptive Routing) – single merged expert constructed via a weighted average of all the experts’ parameters - to address the non-differentiability issue of discrete routing in MoE, hypothesizing that this lack of differentiability is what causes instabilities and underperformance in MoE.

Past research points that stable task/domain-level learned experts are possible (like in the DEMix line of work), but this is harder to achieve at the token-level. A few works showing the challenges of learned MoE at the token-level:

* Hash layer (random routing based on a fixed heuristic) achieves comparable results through a fixed random strategy.
* Switch and the Scaling Laws paper find that increasing the active parameters and the number of experts provides a predictable performance improvement, but this is not the same when just scaling the total number of parameters (this shows limited returns).
  + This can perhaps be explained by suboptimal routing.

With SMEAR, the authors hypothesize that these inefficiencies in MoE are caused by gradient estimation issues. First, they explore if fixed heuristic routing can overperform learned routing, and then compare that to SMEAR (which is fully differentiable).

SMEAR

* In traditional MoE routing, the router training needs to resort to gradient estimation techniques. The goal of SMEAR is to develop an architecture that enables end-to-end gradient-based training (fully differentiable, no gradient estimation) without an increase in computational costs.
* Ensemble routing
  + Would allow for an end-to-end gradient-based training but with a significant increase in computational costs.
* Merging of Experts
  + Recent work has shown that averaging the parameters of models that share a common architecture can often produce an aggregate model that shares the capabilities of the individual models.
* SMEAR
  + Constructs a single merged expert whose parameters are computed as the weighted average of the experts within a routing block.
    - Each expert’s set of weights is set by the corresponding routing probability generated by the router.
  + Instead of only taking the top-k experts selected by the router, which is the discrete step in the strategy, SMEAR weighs each expert’s parameters according to the weight given by the router and merges them into a single expert.
    - Allows updating each expert in each forward pass in a fully differentiable manner.
    - Almost equivalent (slightly higher due to the cost of merging) cost of top-1 routing at inference but more expensive training costs (due to having to backpropagate through each expert after each forward pass).

Experimental Setup

* Main question to be answered is if SMEAR can outperform heuristic routing strategies.
* Use T5 fine-tuned on GLUE for NLP tasks, while also conducting computer vision experiments based on ResNet.
* Used a “tag routing” strategy as one of the baselines, which is a routing strategy based on metadata (oracle routing).
* Add experts to existing pre-trained network (models are not trained from scratch and are based off pre-trained dense models).
  + Similarly to adding adapters for fine-tuning (all pre-trained parameters are kept frozen).
* Router is a simple linear classifier.
* Each layer has 8 experts.
* No balance loss was used.

Results

* Models using learned routing strategies learned through gradient estimation (thus not fully differentiable) often underperform heuristic routing strategies.
* SMEAR outperforms every routing strategy (heuristic or learned) in both NLP and Vision settings, including tag routing (determined by metadata) and a parameter-matched (in terms of total parameters) dense baseline.
  + Consistent with DEMix line of research, which says that a good learned routing strategy should be better than routing determined by metadata.
* SMEAR performs comparably to a fully active MoE ensemble (especially in T5-GLUE), which is seen as the upper bound of this approach.
* In terms of inference, SMEAR performs comparably to the top-1 routing strategy.
* Doubling the number of experts (from 8 to 16) in SMEAR led to a slight performance boost in Vision but no notable difference in T5-GLUE.
* Significant sparsity observed when visualizing the router’s distribution, suggesting expert specialization.

My takeaways:

* SMEAR offers a novel training framework that might set a precedent for future MoE models by mitigating the non-differentiability issue common in discrete routing decisions, thereby leading to more stable and efficient learning.
* The gradual diversification from a single expert to a full MoE configuration in SMEAR could inspire new initialization techniques for complex neural networks, ensuring a smoother transition to specialized expert utilization.
* Given SMEAR’s performance improvements and computational efficiency, it would be worthwhile to investigate how it could be adapted to real-world tasks requiring modularity and efficiency, such as personalized recommendation systems or multi-domain language models.

### MoE-Mamba: Efficient Selective State Space Models with Mixture of Experts

Main Idea: the goal of this work is to compare vanilla MoE (Transformer MoE) with vanilla Mamba and MoE-Mamba to explore if these architectures are compatible with each other. The main highlight of this paper is that MoE-Mamba outperforms both Transformer-MoE and vanilla-Mamba, reaching the same performance of vanilla-Mamba with 2.2x less training steps, while preserving Mamba inference gains over the Transformer. This shows that MoE results in performance gains when combined with the Mamba architecture, similarly to when applied to Transformers. In theory, this should result in easier scaling for Mamba, with even more inference gains due to the sparsity of MoE.

Mamba

Mamba is an SSM (State-Space Model) architecture (SSM meaning that it is based on hidden states that update and drop/forget irrelevant information) like RNNs, GRUs and LSTMs.

* Mamba is an improvement over previous SSM architectures because it is optimized for GPUs and can make use of parallelism.
* Mamba is an improvement over Transformers because the characteristic of dropping irrelevant info of SSM architectures allows for a much lesser complexity as the input size increases. In theory, this should result in increased quality and reduced inference costs for Mamba compared to Transformers when scaling the context length.
  + Transformers’ complexity increases quadratically with an increase in input size (O(n^2)). Mamba does not impose this constraint.

MoE-Mamba Architecture

* MoE-Mamba makes use of a similar architecture to Switch Transformer.
  + Token-choice routing (top-k) with k=1 (one expert used per token)
  + Every other Mamba layer is replaced with an FF MoE (each block alternates between dense (Mamba) and sparse (Mamba MoE) layers).
* The active parameters of the models experimented with were ~26M per token.
  + The total number of parameters of the biggest MoE-Mamba model used was 416M parameters (32 experts).
* MoE-Mamba scales well with an increase in the number of experts (expert size was constant, so increasing the number of experts means increasing the number of total parameters while keeping the number of active parameters constant). The largest number of experts experimented with was 32.
  + MoE-Mamba needed at least 8 experts to improve over vanilla Mamba.

My takeaways:

* Mamba’s main advantage over Transformers seems to be of the handling of large context lengths due to SSM architectures inherently having the ability to drop irrelevant info from token to token. This is not true for the attention process in the Transformer architecture, which has an exponential increase in complexity with an increase in the context length.
  + The main questions about Mamba’s legitimacy today are:
    - How will Mamba scale in terms of increasing parameter size and data?
    - Will Mamba work given huge context lengths (tens/hundreds of thousands of tokens)?
  + More research on the Mamba architecture is needed on my end.
* More research on Mamba-MoE needs to be done at increased parameter scales. A 416M parameter model with 26M active parameters per token is too small. Thus, the results of this paper should be seen as a mere indication and be taken with a grain of salt.

### BlackMamba: Mixture of Experts for State-Space Models

Main Idea: this work looks to combine the Mamba with the MoE architecture. Each of these architectures have unique advantages: Mamba has linear time and memory complexity to increases in context length (is robust to long-range context), while MoE has the advantage of allowing for scaling model’s parameters while keeping inference costs fixed at the expense of a larger memory footprint. BlackMamba (Mamba + MoE fusion) is then expected to have the long-range context robustness of Mamba while having the inference efficiency of MoE.

The models experimented with are larger than the previous work done (Mamba-MoE) but could be scaled much more (the models are, in [ACTIVE PARAMETERS/TOTAL PARAMETERS], 340M/1.5B and 630M/2.8B)

Expected Advantages (Synergies) of BlackMamba vs Dense Transformer

* From Mamba
  + Linear computational complexity with respect to input sequence length for both training and inference.
  + Autoregressive generation in constant time and memory.
* From MoE
  + Inference latency and training FLOPs of the equivalent smaller dense base model, while preserving model quality close to an equivalent dense model in terms of total parameters.

MoE Details

* MoE top-k routing is used.
* MoE is compared/evaluated based on:
  + (Forward pass or active parameters) / total parameters ratio
* Similarly to Mixtral8x7B, a relatively small number of experts is used in BlackMamba (even though scaling laws show promise in having many experts) to balance the inference FLOPs and memory cost of MoE (more experts = more memory costs).

Architecture

* BlackMamba consists of replacing a few layers in the Transformer architecture:
  + The MLP/FF layers are replaced by sparse MoE layers.
  + The Attention layers are replaced by Mamba layers.
* BlackMamba was trained on 300B tokens. This is consistent with the scaling done in this paper compared to the previous work trying to combine these architectures (MoE-Mamba was trained on 10B tokens and had significantly smaller model size).
  + 340M/1.5B and 630M/2.8B sized models trained (active parameters/total parameters).
  + 8 experts used per MoE layer.
  + Found a slight advantage in using sequential versus parallel blocks, so prioritized a sequential setup.
    - This is equivalent to depth vs width.
  + Used top-1 routing with the Sinkhorn algorithm to ensure load balancing between experts.
    - Sinkhorn was the same algorithm used in BASE routing. It makes routing more efficient in accelerated hardware (GPUs).
    - A novel version of Sinkhorn was developed, which has faster convergence.
  + Used the Megatron-LM framework for distributed training.
  + Trained using bf16 precision.

Results

* For the same number of active parameters (equal at inference) and the same amount of training FLOPs (equal amount of training), BlackMamba performed significantly better than the Transformer, Transformer-MoE and Mamba equivalents.
* As expected, BlackMamba also showed significant latency improvements over the other architectures. These latency improvements increase with an increase in context length.
  + This indicates that the synergy between Mamba and MoE works.
* In terms of expert balance, most layers show this happens successfully. However, later layers show a clear transition towards expert imbalance.
  + Perhaps this is due to numerical instabilities that show as we get deeper into the network?
  + This pattern of instability in later MoE layers was also shown in the “Faster-MoE” paper.
* BlackMamba leaves room for future work in terms of the Mamba + MoE fusion:
  + Few-shot performance.
  + Quantization and PEFT performance.
  + Fine-tuning, instruction-tuning and DPO performance.
  + Are the expert’s specialization dynamics in BlackMamba the same as in Transformer MoEs?

My takeaways:

* The checkpoints of BlackMamba were released, so perhaps some investigation can be done in terms of exploring the expert’s specialization dynamics in the BlackMamba architecture and compare it to regular Transformer MoEs.

### Mixture-of-Experts Meets Instruction Tuning: a Winning Combination for Large Language Models

Main Idea: this study aims to measure the impact of instruction-tuning in MoE models compared to its impact in dense models.

Instruction-tuning is related to fine-tuning as fine-tuning is training a pre-trained model on a specific task, while instruction-tuning consists of training a language model in a supervised manner to perform well in a dialogue setting. This means for the model to perform well on the task of predicting p(answer | question) instead of the pre-training objective of predicting p(word | context).

Three different scenarios were evaluated:

* Direct finetuning on individual tasks (no instruction tuning).
* Instruction tuning followed by in-context learning (no direct fine-tuning)
* Instruction tuning followed by further finetuning on individual tasks.

The conclusion of this paper was that MoE models outperform dense models of equivalent computational capacity on direct finetuning, but significantly outperform dense models on instruction tuning scenarios. Let’s understand how they reached this conclusion.

Setup

Two dense models were considered: T5 and PaLM.

Four MoE architectures were considered:

* Switch Transformers
* GShard
* Expert-Choice
* ST-MoE

All instruction tuning was done using the FLAN dataset.

Results

* A base MoE architecture outperforms a dense architecture (T5) after instruction-tuning across all scales.
* Scaling the number of experts helps when fine-tuning on challenging tasks but saturates when fine-tuning on easier tasks (more experts is not always better as it might confuse the gating algorithm).
* As expected, increasing k in top-k routing improves performance at an increase in the inference cost.
* Overperformance of MoE compared to dense models when instruction-tuning only exacerbates with scale (the bigger the models, the bigger the performance gain of MoE over dense).
* Expert-choice outperforms GShard (token-choice) in an instruction-tuning scenario, however, this difference is bridged by incorporating advanced auxiliary loss (router z-loss) and pre-training strategy as employed in ST-MoE (also token-choice).
* Even though FLAN-PaLM62B (dense instruction-tuned model) has 3x the number of FLOPs per token than FLAN-ST32B (largest MoE instruction-tuned model trained for this work) at inference, FLAN-ST32B significantly outperforms FLAN-PaLM (57.6 vs 63.6 average score).
* Different auxiliary losses gave different results:
  + Z-loss worked better than balance-loss in FLAN-ST
  + Balance-loss worked better than z-loss in FLAN-EC
* Freezing certain parts of the MoE layers during fine-tuning was evaluated to investigate how to prevent overfitting in MoE fine-tuning:
  + Freezing the gate led to small improvements.
  + Freezing any other areas resulted in worse performance.

My takeaways:

* First thought is that instruction-tuning should work better in dense models than in MoE models based on the difficulties in obtaining good fine-tuning performance with MoE. This may not hold since the instruction-tuning process can be thought of a very specific type of fine-tuning.
  + This is shown to be false, as MoE significantly outperforms dense models when it comes to instruction-tuning. This is even more interesting when showed that this advantage of MoE over dense in the task of instruction-tuning only increases with scale.
  + MoE results after instruction-tuning are quite promising. For some reason, MoE captures the instruction-tuning task much more efficiently than dense models.
* More experts do not guarantee better performance with fine-tuning. In fact, on easier tasks, more experts result in worse fine-tuning performance.
  + What was the size of the datasets used for fine-tuning? Perhaps easier tasks are more prone to overfitting, explaining the underperformance of fine-tuning MoE on easier datasets. If this was the case, these tasks would require more regularization -> how much regularization to use might depend on the difficulty of the task.
    - This makes sense to the overall MoE theory as easier tasks have less complex data distributions The less complex data distribution will lead to less of the experts being called consistently, causing them to overfit. In a complex task, the data distribution will result in a more distributed load balancing due to more semantic/syntax patterns being in place, thus using more experts, preventing overfitting.
  + There might be router issues leading to this difficulty in fine-tuning on easier tasks as well.
* Expert-choice seems to be better than regular token-choice routing. However, ST-MoE, which has improvements over traditional token-choice routing, surpasses expert-choice.
  + Why did Mixtral decide to not use Expert-Choice and seems to use a routing strategy that resembles GShard more, even though it underperforms both Expert Choice and ST-MoE’s routing strategies? Maybe they started training before this paper came out? (investigate if Mixtral’s routing strategy resembles more GShard than ST-MoE).
* Z-loss is better for token-choice, but balance-loss is better for Expert-Choice?
* The routing learned during pre-training is thought to already have a good estimate of data distributions at a semantic and syntactic level, therefore more specialization is not needed during fine-tuning. The idea is that the semantics and syntax at fine-tuning domains are not new, what changes is their distribution. Therefore, the routing algorithm does not to be updated -> gating/routing should be kept frozen during fine-tuning (this is not the first research work to come to this conclusion).
* MoE models are prone to overfitting, so often underperform dense models on single-task fine-tuning. MoE works better when scaling the number of tasks, that is, fine-tuning on more than just one domain. However, instruction-tuning seems to bring a reversal to this trend, with FLAN-MoE performing better than FLAN-T5 in single task fine-tuning.
  + Perhaps a reason for this is how FLAN does not have a single task per-say, it instead has data from many different domains with the common aspect being the structure how it is presented (in a dialogue format).

## **Multimodal MoE**

### Multimodal Contrastive Learning with LIMoE: the Language-Image Mixture of Experts

### MoE-Llava: Mixture of Experts for Large Vision-Language Models (+ Visual Instruction Tuning aka Llava)

### Llava-Phi: Efficient Multi-Modal Assistant with Small Language Model

* <https://arxiv.org/pdf/2209.01667.pdf> - Survey on MoE paper (Sept 2022)
* <https://arxiv.org/pdf/2302.11529.pdf> - Modular Deep Learning Survey (Jan 2024)
* <https://arxiv.org/pdf/2311.08692.pdf> - Routing to the Expert paper
* <https://openreview.net/pdf?id=pEKJl5sflp> - Scalable Modular Network paper
* <https://arxiv.org/pdf/2311.13171.pdf> - ComPEFT paper
* <https://openreview.net/pdf?id=nUBSQKeROV> - Can Dense Pre-Trained Transformers Benefit from Modularity paper (under review)
* <https://arxiv.org/pdf/2310.01334.pdf> - Merge, Then Compress paper
* <https://openreview.net/pdf?id=uWvKBCYh4S> - MoLE paper
* <https://arxiv.org/pdf/2309.13850.pdf> - Statistical Perspective of Top-k Sparse Softmax Gating MoE paper
* <https://arxiv.org/pdf/2306.16788.pdf> - Sparse Model Soups paper
* <https://arxiv.org/pdf/2310.01542.pdf> - Fusing Models with Complementary Expertise paper
* <https://arxiv.org/pdf/2401.10191.pdf> - Divide and Forget paper
* <https://arxiv.org/pdf/2403.07816.pdf> - Branch-Train-MiX paper