DESIGNING EFFECTIVE SPARSE EXPERT MODELS

Barret Zoph[*]
Google Brain

Irwan Bello[*†]
Google Brain

Sameer Kumar
Google

Nan Du
Google Brain

Yanping Huang
Google Brain

Jeff Dean
Google AI

Noam Shazeer[†]
Google Brain

William Fedus[*]
Google Brain

ABSTRACT

Scale has opened new frontiers in natural language processing – but at a high cost. In response, Mixture-of-Experts (MoE) and Switch Transformers have been proposed as an energy efficient path to even larger and more capable language models. But advancing the state-of-the-art across a broad set of natural language tasks has been hindered by training instabilities and uncertain quality during fine-tuning. Our work focuses on these issues and acts as a design guide. We conclude by scaling a sparse model to 269B parameters, with a computational cost comparable to a 32B dense encoder-decoder Transformer (Stable and Transferable Mixture-of-Experts or ST-MoE-32B). For the first time, a sparse model achieves state-of-the-art performance in transfer learning, across a diverse set of tasks including reasoning (SuperGLUE, ARC Easy, ARC Challenge), summarization (XSum, CNN-DM), closed book question answering (WebQA, Natural Questions), and adversarially constructed tasks (Winogrande, ANLI R3). [1]

---

[*]Equal contribution. Correspondence to {`barretzoph,liamfedus`}@google.com.

[†]Work was done while at Google.

[1]Code for our models is available at `https://github.com/tensorflow/mesh/blob/master/mesh_tensorflow/transformer/moe.py`

CONTENTS

1 INTRODUCTION

Sparse expert neural networks showcase the advantage of sheer scale and offer an efficient alternative to the static neural network architectures commonly used today (Raffel et al., 2019; Brown et al., 2020; Rae et al., 2021). Rather than applying the same parameters to all inputs, sparse expert networks dynamically select which parameters to use for each input (Shazeer et al., 2017). This allows for networks to vastly expand their number of parameters, while keeping the FLOPs per token roughly constant. These approaches have yielded state-of-the-art translation models (Lepikhin et al., 2020), 4-7x pre-training speed-ups (Fedus et al., 2021; Artetxe et al., 2021), and GPT-3 level one-shot performance using 1/3 the energy training cost (Du et al., 2021). And despite a shocking number of parameters, sparse models reduce the carbon footprint for training large neural networks by an order of magnitude (Patterson et al., 2021). However, difficulties remain.

Fedus et al. (2021) observed that a sparse 1.6T parameter model achieved a 4x pre-training speed-up over the prior state-of-the-art (Raffel et al., 2019), but lagged smaller models when fine-tuned on common benchmarks like SuperGLUE. Similar gaps were observed in Artetxe et al. (2021) when MoE language models were fine-tuned on out-of-domain data. In response, Switch-XXL, a model with fewer parameters, but a 8x-larger computational footprint (FLOPs approximately equal to the largest T5 model), was proposed and improved quality on natural language understanding tasks. However, necessary pre-training was hampered by training instabilities previously undetected during smaller scale studies. These instabilities were later identified in other sparse models (Du et al., 2021). These results revealed a necessary balance of parameters *and* computation, but left an open question on how to reliably train these types of models.

Our aim in this paper is to increase the *practicality* and *reliability* of sparse models. We study these two issues and pre-train a 269B sparse model that achieves state-of-the-art results when fine-tuned across many competitive NLP benchmarks, including SuperGLUE. We also put forth additional analysis and a design guide (or at least, our heuristics) for sparse expert models. Furthermore, this work emphasizes *jointly* optimizing both the upstream pre-training and the downstream fine-tuning metrics to avoid discrepancies (Tay et al., 2021).

Contributions

1. A large-scale study of the quality-stability trade-offs of stability techniques.

2. An introduction of the *router z-loss* that resolves instability issues, while slightly improving model quality.

3. A fine-tuning analysis of sparse and dense models highlighting different hyperparameter sensitivity to the batch size and learning rate. We show bad hyperparameters result in virtually no fine-tuning gain over dense models, despite large pre-training speed-ups.

4. Architectural, routing and model design principles for designing Pareto efficient sparse models in a distributed setting.

5. A qualitative analysis tracing token routing decisions across expert layers.

6. A 269B sparse model (the Stable Transferable Mixture-of-Experts or ST-MoE-32B) which achieves state-of-the-art performance across a diverse set of natural language benchmarks.

---

2 BACKGROUND

Sparse expert models typically substitute a neural network layer with a set of *experts*, each having unique weights (Jacobs et al., 1991; Jordan and Jacobs, 1994). Typically all the experts within a layer are of the same type and shape (homogeneous), however, varied (heterogeneous) expert-types are possible. Inputs are only processed by a subset of the experts to save computation, so a mechanism must be added to determine where to send each input. Usually a *router* or gating network determines where to send inputs (i.e. words, sentences, image patches, etc.), but alternative schemes have been proposed (Lewis et al., 2021; Roller et al., 2021; Zuo et al., 2021; Clark et al., 2022).

Specifically, in natural language processing, Shazeer et al. (2017) proposed a Mixture-of-Experts (MoE) layer which takes a token representation x as input and routes it to the best matched top-k experts selected out of a set $\{E_i(x)\}_{i=1}^N$ of N experts. The router variable W_r produces logits $h(x) = W_r \cdot x$ which are normalized via a softmax distribution over the available N experts at that layer. The gate-value for expert i is given by

$$p_i(x) = \frac{e^{h(x)_i}}{\sum_j^N e^{h(x)_j}} \tag{1}$$

and the token x is routed to the experts with the highest top-k gate values (set of indices $\mathcal{T}$). The output of the layer is the weighted sum of each expert's computation by the gate value

$$y = \sum_{i \in \mathcal{T}} p_i(x) E_i(x) \tag{2}$$

Originally proposed in LSTMs (Hochreiter and Schmidhuber, 1997), expert layers were later used in the Transformer (Vaswani et al., 2017) by Shazeer et al. (2018) and Lepikhin et al. (2020). Follow-on work by Fedus et al. (2021) simplified the MoE further to route tokens to a single expert (top-1) and reduced other costs to improve training efficiency.

To improve hardware utilization, most implementations of sparse models have static batch sizes for each expert (Shazeer et al., 2017; 2018; Lepikhin et al., 2020; Fedus et al., 2021). The *expert capacity* refers to the number of tokens that can be routed to each expert. If this capacity is exceeded (the router sends too many inputs to that expert) then the overflowed tokens have no computation applied to them and are passed to the next layer through a residual connection.

Terminology	Definition
Expert	An independently-learned neural network with unique weights.
Router	A network that computes the probability of each token getting sent to each expert.
Top-n Routing	Routing algorithm where each token is routed to n experts.
Load Balancing Loss	An auxiliary (aux) loss to encourage each group of tokens to evenly distribute across experts.
Group Size	The global batch size is split into smaller groups, each of size Group Size. Each group is considered separately for load balancing across experts. Increasing it increases memory, computation, and communication.
Capacity Factor (CF)	Each expert can only process up to a fixed number of tokens, which is often set by evenly dividing across experts, $\frac{\text{tokens}}{\text{experts}}$. The capacity factor can expand or contract this amount to CF $\cdot \frac{\text{tokens}}{\text{experts}}$.
FFN	Acronym of Feed Forward Network (FFN) layer of Transformer consisting of linear, activation, linear.
Encoder-Decoder	A Transformer architectural variant that all of our models are based on. Consists of an encoder that does all-to-all attention on the inputs and a decoder that attends to the encoder and to its own inputs in an autoregressive manner.
`allreduce`	Communication primitive which sums a subset of n tensors on n different devices, then broadcasts the summed value to all n devices. This is used in distributed training for gradient accumulation and model parallelism.
`all2all`	Communication primitive where each device sends to every other device a part of its tensor. Used in sparse Transformer models for token routing.
($\uparrow$/$\downarrow$)	Indicates whether higher/lower values are better (e.g. accuracy/train loss).

Table 1: **Terminology used throughout the paper.**

The batch $\mathcal{B}$ of input tokens is broken into $\mathcal{G}$ unique groups across the data-parallelism dimension[2], each with size $\mathcal{B}/\mathcal{G}$. The expert capacity is equal to CF $\cdot$ tokens/experts where CF represents the

---

[2]Our implementation relies on einsums with one-hot tensors for dispatching and combining tensors to/from experts. The size of this one-hot tensor grows quadratically with the number of tokens being routed as a group which motivates breaking the batch into smaller groups. This may be avoided with sparse lookup operations.

capacity factor hyperparameter, experts is the number of experts and tokens is the group size. If the capacity factor is increased, it creates extra buffer so that fewer tokens will be dropped in case of load imbalance. However, increasing the capacity factor also increases the memory and computational costs, so there exists a trade off[3].

Finally, an auxiliary load balancing loss encourages tokens to be roughly evenly distributed across the experts (Shazeer et al., 2017). This improves the hardware efficiency by ensuring that all accelerators are processing significant chunks of data in parallel as mentioned above. The details of the loss are presented in Appendix A. However, alternatives exist: Lewis et al. (2021) and Clark et al. (2022) treats balanced token allocation as an assignment problem and removes the auxiliary loss entirely.

3   STABILIZING TRAINING OF SPARSE MODELS

Sparse models often suffer from training instabilities (Figure 1) worse than those observed in standard densely-activated Transformers.
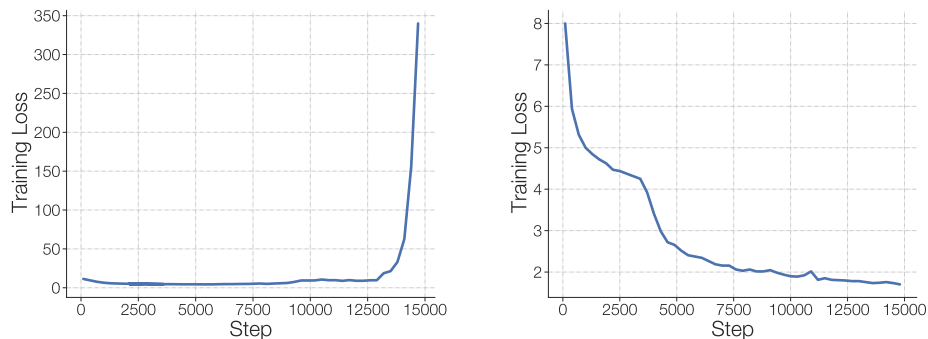
Figure 1: **Training instabilities for sparse models.** We refer to training instabilities as divergences in the training loss. Above are two runs from sparse models FLOP-matched to the T5-XL version (Raffel et al., 2019) each trained with a batch size of 1M tokens using the Adafactor optimizer (Shazeer and Stern, 2018). **(Left)** An unstable training run. **(Right)** A stable training run.

It's straightforward to find changes that improve the *stability*, however, these often come at an untenable expense to model *quality* (for instance, using an arbitrarily small learning rate or using tight gradient clipping). We categorize and examine several approaches to improve stability. The stability techniques span generic fixes to Transformers as well as those specific to sparse models: **(1)** Remove multiplicative interactions **(2)** Inject model noise **(3)** Constrain activations, and gradients. We conclude with our recommendation: a new auxiliary loss, the *router z-loss*, which significantly improves training stability with no quality degradation. This is an adaptation of the z-loss used for final softmax logits in the Mesh Tensorflow codebase (Shazeer et al., 2018).

> **Stabilizing Sparse Models**
>
> **1.** Many methods stabilize sparse models, but at the expense of worse quality.
>
> **2.** The router z-loss stabilizes models without quality degradation.
>
> **3.** Transformer modifications with more multiplicative components (GEGLU, RMS normalization) worsen stability, but boost quality.

Designing a large-scale stability study.   We design a large-scale stability study of sparse models FLOP-matched to the T5-XL version (Raffel et al., 2019) pre-trained on the multilingual corpus mC4 (Xue et al., 2020). Each sparse model has 32 experts and we introduce a sparse MoE layer for

---

[3]See Fedus et al. (2021) for a graphical illustration of how the capacity factor works.

every fourth FFN. The train capacity factor is 1.25 and the eval capacity factor is 2.0. See Table 11 for a more detailed description of models used throughout this paper. For each stability technique, we record the fraction that are stable, the mean quality (negative log perplexity on English), and the standard deviation over seeds.

The primary issue in constructing this study is that small models are rarely unstable but large unstable models are too costly to run for sufficient steps and seeds. We found a sparse model FLOP-matched to T5-XL to be good object of study because it was unstable roughly $1/3$ of the runs, but was still relatively cheap to train. Furthermore, we run our instability experiments on multilingual data since we find this exacerbates model instabilities, allowing us to experiment on slightly smaller models. See Section 9 for more details. Our baseline configuration is trained using six random seeds and each configuration with a stability technique uses three random seeds. We use six seeds for the baseline to better characterize the instability rate and three seeds for the variants to save compute. Each model is pre-trained for 20k steps on mC4 using a masked language modeling objective (Fedus et al., 2018; Devlin et al., 2018).

3.1 Stability and Quality Tradeoffs when Removing Multiplicative Interactions

Some architectural improvements involve more multiplications than additions or do not sum many items at once. For example, a matrix multiplication has one multiplication for each addition and hence we do not refer to it as a "multiplicative" operation. We present and analyze the impact of two instances of multiplicative interactions in Transformers here.

GELU Gated Linear Units (GEGLU). Our first example is the Gated Linear Unit (Dauphin et al., 2017) which is a component-wise product of two linear projections, one of which is first passed through a sigmoid function. Shazeer (2020) extends this to other variants and presents a GELU-Linear (Hendrycks and Gimpel, 2016) FFN layer as a replacement the usual ReLU (Nair and Hinton, 2010) FFN in Transformer.

$$FFN_{GEGLU}(x, W, V, b, c) = GELU(xW + b) \odot (xV + c) \tag{3}$$

This quality gain was corroborated in later work (Narang et al., 2021).

Root Mean Square Scale Parameters. Our second example is the scale parameter in root mean square (RMS) normalization (Zhang and Sennrich, 2019). Within the Transformer, rather than calling layers back-to-back, there is an internal structure (referred to as sublayer calls) which improve gradient propagation and training dynamics. Our sublayer calls match that of Raffel et al. (2019) and consist of: **(1)** RMS normalization, **(2)** layer call (e.g. Self Attention), **(3)** dropout (Srivastava et al., 2014), **(4)** add residual (He et al., 2015). RMS normalization scales the input vector $x \in \mathbb{R}^d$ element-wise per the root-mean-square. It then rescales the output element-wise by multiplying with a learned scale parameter g.

$$y_i = \frac{x_i}{\sqrt{\frac{1}{d} \sum_{i=1}^{d} x_i^2}} \cdot g_i \tag{4}$$

Table 2 shows that both removing GEGLU layers or the RMS scale parameter improves stability, but at a significant loss to model quality. We note that these scale parameters (g) have a disproportionate gain to model quality versus parameters elsewhere (e.g. FFN). In line with our findings, Shleifer et al. (2021) found adding a learned multiplicative scalar to the residual connection in Transformers made them much more unstable.

In Appendix C, we further study the quality impact of adding new multiplicative interactions in expert layers. We find that this operation yields quality improvements with virtually no slow-down in model step time.

3.2 Stability and Quality Tradeoffs when Adding Noise

We next explore a hypothesis that adding noise into the model can improve training stability (Neelakantan et al., 2015). Taleb (2012) argues that certain systems exhibit the property of anti-fragility, where they *improve* through noise. Inspired by the concept and by our observation that fine-tuning

Method	Fraction Stable	Quality ($\uparrow$)
Baseline	4/6	**-1.755** ± 0.02
Remove GEGLU	3/3	-1.849 ± 0.02
Remove RMS Norm. Scale Param	3/3	-2.020 ± 0.06

Table 2: **Removing operations with more multiplicative interactions**. Multiplicative interactions improve quality, but can destabilize training. Individually removing two sources of multiplicative components improves the stability, but worsens quality significantly. When we remove the GEGLU layer, we replace it with with an equivalent Dense-ReLU-Dense layer to match the FLOPs and parameters.

(which injects noise via dropout) was rarely unstable, we examined whether training noise might improve the stability of sparse models. Table 3 shows a stability improvement versus the baseline, but at the expense of lower quality. We also find that input-jitter, introduced by Fedus et al. (2021), diminishes quality at XL-scale, hence we ablate it in our models. Input-jitter multiplies the input logits to the router by a uniform random variable between $[1 - 10^{-2}, 1 + 10^{-2}]$. Dropout in our ablation is applied throughout the Transformer. As seen previously, improvements in small-scale settings may fail to generalize when scaled up and therefore trends should always be monitored and re-assessed at increasing scale (Kaplan et al., 2020).

Method	Fraction Stable	Quality ($\uparrow$)
Baseline	4/6	**-1.755** ± 0.02
Input jitter (10^{-2})	3/3	-1.777 ± 0.03
Dropout (0.1)	3/3	-1.822 ± 0.11

Table 3: **Injecting noise during training**. Both input-jitter and dropout improve stability, but lead to a significant loss of model quality. There is a clear tradeoff with most methods: when one improves stability, it then typically decreases model quality. Our work aims to find methods that fix stability without hurting quality.

3.3 STABILITY AND QUALITY TRADEOFFS WHEN CONSTRAINING ACTIVATIONS AND GRADIENTS

One of the most successful approaches to stabilizing neural networks are constraints on activations, and gradients (Pascanu et al., 2013; Ioffe and Szegedy, 2015; Salimans and Kingma, 2016; Ba et al., 2016). A popular approach consists in the clipping of gradient norms to remedy exploding gradients while backpropagating through deep networks (Pascanu et al., 2013).

In this work, we use the Adafactor optimizer due to its memory efficiency (though recently introduced 8-bit optimizers (Dettmers et al., 2021) may offer better trade-offs). Instead of gradient clipping, Adafactor uses *update clipping*, where the changes to the weights are constrained to be below a certain norm. We experiment with tightening the update clipping to a smaller value.

Next, we study constraints on the logits going into the router. The router computes the probability distribution over the experts in `float32` precision (i.e. selective precision) (Fedus et al., 2021). However, at the largest scales, we find this is insufficient to yield reliable training. To fix this, we introduce the *router z-loss*,

$$L_z(x) = \frac{1}{B} \sum_{i=1}^{B} \left(\log \sum_{j=1}^{N} e^{x_j^{(i)}} \right)^2 \tag{5}$$

where B is the number of tokens, N is the number of experts, and $x \in \mathcal{R}^{B \times N}$ are the logits going into the router. This penalizes large logits into the gating network and Section 3.4 contains a more detailed explanation of why the z-loss before the router is useful.

Table 4 shows that both update clipping and the router z-loss stabilize the model in all 3 runs, but the update clipping significantly hurts the model quality. Therefore we use the z-loss method for fixing our model stability due to improved quality and stability[4].

Method	Fraction Stable	Quality ($\uparrow$)
Baseline	4/6	-1.755 ± 0.02
Update clipping (clip = 0.1)	3/3	-4.206 ± 0.17
Router Z-Loss	3/3	**-1.741** ± 0.02

Table 4: **Constraining weight updates and router logits.** Constraining the update clipping in Adafactor improves stability, but at a catastrophic loss of quality. Looser clipping values did not reliably stabilize training so we exclude them here. The router z-loss stabilizes the model without any quality degradation (in this case, we observe a slight quality boost).

The router z-loss introduces another hyperparameter (c_z), which is the coefficient to weight this as part of the total loss optimized. The total loss is a linearly weighted combination of the cross entropy loss (L_{CE}), the auxiliary load balance loss (L_B), and the router z-loss (L_Z), yielding a total loss

$$L_{tot} = L_{CE} + c_B L_B + c_z L_Z \tag{6}$$

We choose a value of $c_z = 0.001$ based on the best model quality after pre-training with a hyperparameter sweep. Appendix B logs the resulting losses over the course of pre-training.

3.4 SELECTING A PRECISION FORMAT: TRADING EFFICIENCY AND STABILITY

As in most modern distributed Transformers we train with *mixed precision* (Micikevicius et al., 2017) [5]. Weights are stored in `float32` for gradient updates and then converted to `bfloat16` when doing matrix multiplications in the forward and backward pass[6]. Furthermore, all activations are stored and operated on in `bfloat16` and `allreduce` communications can be done in either `bfloat16` or `float32` numerical precision. For the largest model explored in this work (ST-MoE-32B presented later) we find speed-ups halving the numerical precision of the `allreduce`, however this also can destabilize the training so we keep this as `float32` throughout this work.

A lower precision format enables more efficient models by reducing **(a)** communication costs between processors and memory, **(b)** computation costs, **(c)** memory for storing tensors (e.g. activations). However, lower precision formats come at the expense of larger roundoff errors which can lead to irrecoverable training instabilities.

Precision Format: Float32
Exponent (8 bits) Mantissa (23 bits)

Precision Format: BFloat16
Exponent (8 bits) Mantissa (7 bits)

Number Range	Max BFloat16 Roundoff Error	Max Float32 Roundoff Error
[2, 4)	0.01563	2.34x10^(-7)
[32, 64)	0.25	3.81x10^(-6)
[1024, 2048)	8.0	0.00012
[2^20, 2^21)	8192.0	0.125
[2^30, 2^31)	8288608.0	128.0
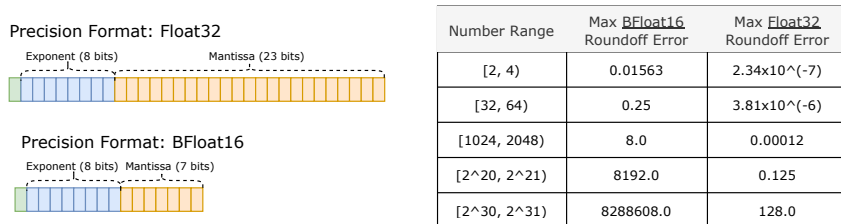
Figure 2: **Numerical precision formats and roundoff errors.** Larger numbers have larger roundoff errors. `bfloat16` has up to 65,536x worse roundoff errors than `float32`. The router z-loss encourages the absolute magnitude of numbers to be small, which doesn't hinder model performance and reduces roundoff errors. The router z-loss is most effective into functions where larger errors can drastically change the relative output (e.g. exponential and sinusoidal functions).

---

[4]We also experimented with adding z-losses onto the attention logits which also improves model instability without hurting model quality.

[5]See Mesh Tensorflow for implementation details: `https://github.com/tensorflow/mesh/blob/master/mesh_tensorflow/`

[6]Matrix multiplications on TPUs perform multiplications in `bfloat16` and accumulations in `float32`.

Understanding precision format and roundoff errors. Figure 2 reviews the properties of different precision formats and their corresponding roundoff errors for different number ranges. Numbers in any range of two consecutive powers of 2 (e.g. [2,4) and [1024, 2048)) are represented by a fixed number of mantissa bits (7 for `bfloat16`, 23 for `float32`). As a result, **(1)** `bfloat16` will have about 65,536x (i.e. $23 - 7 = 16$ additional bits and $2^{16} = 65536$) as large roundoff errors as `float32` and **(2)** larger numbers have larger roundoff errors. Due to the 8 exponent bits, number can get as large as $\approx 3e^{38}$, which leads to even `float32` having some issues with roundoff errors.

Sparse expert models are sensitive to roundoff errors because they have more exponential functions due to the routers. Sparse expert models introduce additional exponential functions – through the router – which can exacerbate roundoff errors[7] and lead to training instabilities. While a roundoff error does not change the ordering of probabilities within a softmax operation, it does impact the routing of the second token in MoE due to relative thresholding (e.g. a token is only routed to its second place expert if the gating probability for the second expert is $1/5$ as large as that of the first expert). Additionally, roundoff errors can drastically change the probability that scales the expert output – which we have found to be important. Finally, we conjecture that the higher stability we observed for decoder-only models (not shown here) was because they had fewer exponential functions. Section 9 contains a more detailed discussion.

An aside on the router z-loss. One might think that the router z-loss is a convoluted method replaceable by clipping logits (Wu et al., 2016). We explain why this is not the case. The goal is to minimize large roundoff errors going into exponential functions. Clipping the logits occurs *after* any roundoff errors – resulting in even larger discontinuities. In one view, clipping in itself is a roundoff error; conversely, the z-loss naturally encourages the model to produce logits that are small in value and thus more accurately modeled. Due to these dynamics, we ensure all exponentiated tensors are cast to `float32`. This hints at the possibility of better number formats for neural networks because of the unused exponent bits when z-losses are added throughout the network (see Section 9).

4 FINE-TUNING PERFORMANCE OF SPARSE MODELS

The best performing language models are usually obtained by (1) pre-training on large amounts of data (e.g. the internet) followed by (2) fine-tuning on a task of interest (e.g. SuperGLUE). Promising new techniques have emerged as an alternative, including few-shot inference (Brown et al., 2020), prefix tuning (Li and Liang, 2021), prompt tuning (Lester et al., 2021), and adapter modules (Houlsby et al., 2019) – however, a quality gap still persists compared to fine-tuning. Because of this, we focus on fine-tuning in this work, but highlight recent successes of sparse models in few-shot settings from Du et al. (2021); Artetxe et al. (2021). Further, we leave as future work techniques that adapt large language models through reinforcement learning (Ouyang et al., 2022)

4.1 HYPOTHESIS: A GENERALIZATION PROBLEM

Sparse models have performed remarkably well in the regime of large datasets, but have sometimes performed poorly when fine-tuning (Fedus et al., 2021; Artetxe et al., 2021). We present evidence for a (not so surprising) hypothesis that sparse models are prone to overfitting. We illustrate this problem through two tasks in SuperGLUE (Wang et al., 2019) – Commitment Bank (De Marneffe et al., 2019) and ReCORD (Zhang et al., 2018). Commitment Bank (CB) has 250 training examples while ReCORD has over 100,000. This significant size discrepancy facilitates a natural study for overfitting on two tasks selected as part of the same benchmark.

In Figure 3, we compare the fine-tuning characteristics of the Dense L and the ST-MoE-L model. Each model was pre-trained on 500B tokens from the C4 corpus (Raffel et al., 2019). The models

---

[7]Exponential functions have the property that a small input perturbation can lead to a large difference in the output. As an example, consider inputting 10 logits to a softmax function with values of 128 and one logit with a value 128.5. A roundoff error of 0.5 in `bfloat16` will alter the softmax output by 36% and incorrectly make all logits equal. The calculation goes from $\frac{\exp(0)}{\exp(0)+10\cdot\exp(-0.5)} \approx 0.142$ to $\frac{\exp(0)}{\exp(0)+10\cdot\exp(0)} \approx 0.091$. This occurs because the max is subtracted from all logits (for numerical stability) in softmax operations and the roundoff error changes the number from 128.5 to 128. This example was in `bfloat16`, but analogous situations occur in `float32` with larger logit values.
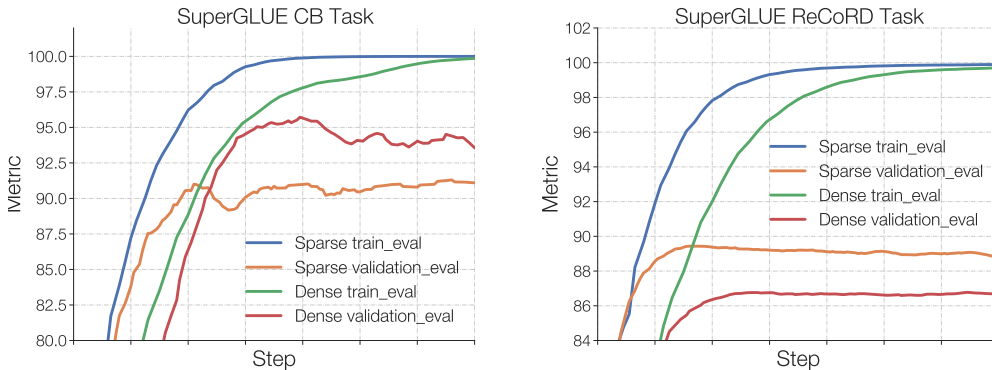
Figure 3: **Sparse models are prone to overfit.** We plot train and validation curves for our ST-MoE-L and a dense-L models fine-tuned on the CB task (250 train sequences) and ReCoRD (138k train sequences). In both cases, the sparse model learns more quickly on the train partition (blue exceeds green line). However, for the smaller CB task, the dense model outperforms the sparse model on the held-out validation set (red vs. orange). In contrast, on the larger ReCoRD task, the sparse model outperforms the dense model by several percentage points.

are designed to be roughly FLOP matched variants of the T5-Large encoder-decoder models from Raffel et al. (2019) with 770M parameters. The ST-MoE models have 32 experts with an expert layer frequency of $1/4$ (every fourth FFN layer is replaced by an MoE layer). The pre-training and fine-tuning train capacity factor is 1.25 and the eval is 2.0. We evaluate performance on the held-out validation and train dataset partitions.

Across both tasks, the sparse model converges faster to 100% train set accuracy supporting that sparse models optimize effectively under a data distribution shift. On the larger task, ReCORD, the validation quality of the sparse model follows the boost in training and significantly exceeds the dense model. However, on the smaller task, CB, the sparse model lags its dense counterpart on held-out data. As per the recommendation of Fedus et al. (2021), we consider increasing the dropout within the expert hidden state (i.e. expert dropout), but find that at this scale, higher values only moderately improve quality (Figure 4). We study further improvements to fine-tuning in Section 4.2 and hyperparameter sensitivity in Section 4.3.
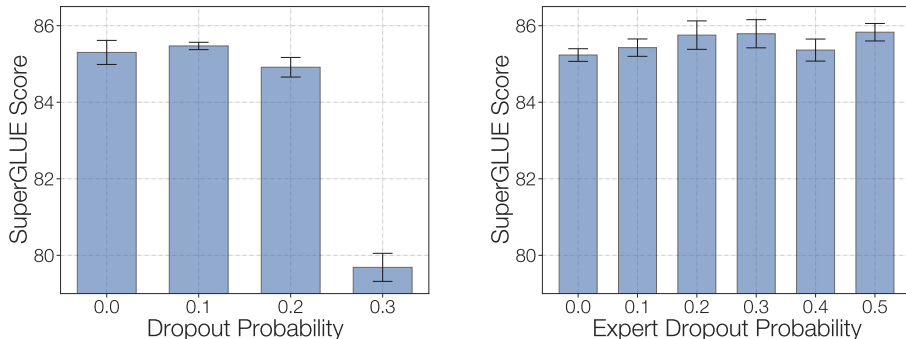
Figure 4: **Regularization studies of sparse models for fine-tuning**. For each setting, we train three random seeds till convergence on SuperGLUE. We find that increased regularization through dropout provides modest boosts. **(Left)** demonstrates peak SuperGLUE fine-tuning quality at a global dropout rate of 0.1. Higher values over-regularize and severely hurt quality. **(Right)** Starting with the best known global dropout rate of 0.1, we selectively increase the expert dropout (an independent dropout rate on the expert hidden activation). This yields further generalization benefits and is in line with the findings of Fedus et al. (2021).

4.2 Fine-Tuning a Subset of Model Parameters to Improve Generalization

To combat overfitting we experiment updating only a subset of models parameters during fine-tuning. Figure 5 measures quality for updating 5 different subsets of parameters: all parameters (All), only non MoE parameters (Non MoE), only MoE parameters (MoE), only the self-attention and enc-dec attention parameters (Attention) and only the non MoE FFN parameters (FFN).
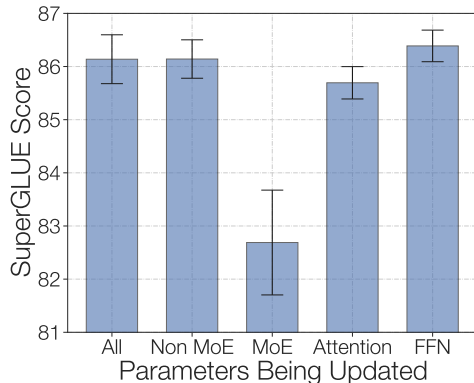
Figure 5: **Updating only a subset of model parameters during fine-tuning**. To improve the generalization of sparse models and combat overfitting, we fine-tune a subset of the model parameters. All results are with the ST-MoE-L model and are an average of 5 different random seeds. We observe that updating $3/5$ of the subsets of parameters appear to work about the same, while fine-tuning only the MoE parameters results in a drastic quality reduction.

We observe that updating the non MoE parameters works about as well as updating all the parameters and updating only the FFN parameters works a bit better. Updating only the MoE parameters significantly degrades fine-tuning performance, which is where $\approx 80\%$ of model parameters are. Only updating the non MoE parameters can be an effective way to speedup and reduce memory for fine-tuning.

We hypothesize that fine-tuning only the MoE parameters leads to bad performance since expert layers only occur every $1/4$ layers and a token will see at most two experts per layer. Therefore, updating the MoE parameters will affect much fewer layers and FLOPs than updating any other subset of the parameters we tried. Updating only the MoE parameters resulted in a much larger training loss than updating the non MoE parameters, even though there are significantly more parameters. We further observe that updating all the non-MoE parameters results in a higher training loss than updating all the parameters, but unfortunately this regularization effect didn't translate to better validation performance.

Further, one regularizer we tried was a dropout variant where entire experts were masked out stochastically during training. However, this failed to improve generalization in our preliminary studies. Appendix J expands on this experiment and contains other negative results.

4.3 Sparse and Dense Models Require Different Fine-Tuning Protocols

How sensitive are sparse and dense models to the fine-tuning protocol? We study two hyperparameters: the batch size and the learning rate. We pretrain a Dense-L and ST-MoE-L on 500B tokens of C4 and then fine-tune on SuperGLUE. Figure 6 summarizes our experiments with the full data presented in Table 20 (Appendix F). Across all hyperparameter settings, the sparse models (orange) outperform the dense (blue) counterparts – however, the best setting for each can materially change results. Sparse and dense models have vastly different performance across different batch sizes and learning rates. Sparse models benefit from smaller batch sizes and a higher learning rate. Consistent with the overfitting hypothesis (Section 4.1), both these changes might improve generalization through higher noise in the fine-tuning process. Finally, we point out the importance of correctly tuning the batch size and learning rate during fine-tuning. Simply using the same fine-tuning hyper-

parameters that worked well for the dense model can mask any pre-training improvements obtained by the sparse model.
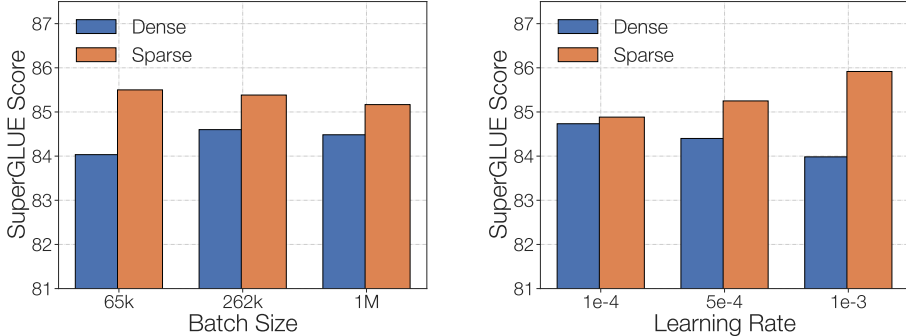

Figure 6: **Batch size and learning rate sensitivity.** We measure differences and sensitivity to fine-tuning protocols between dense (blue) and sparse (orange) models. Each bar is an average across 6 different runs with different hyperparameters. On SuperGLUE, sparse models benefit from noisier hyperparameters including small batch sizes and high learning rates. Dense models behave nearly oppositely. See Appendix F for all data.

4.4 SPARSE MODELS ARE ROBUST TO DROPPED TOKENS DURING FINE-TUNING

Sparse models route tokens to one or more experts at each layer. To make these models efficient in the SPMD paradigm with modern hardware, the expert capacity (the number of tokens each expert processes) needs to be fixed ahead of time (see Section 2 for more details). When an expert receives more tokens than its capacity, the extra tokens are dropped — no computation is applied to those tokens. We again try to prevent this by **(1)** pre-training with an auxiliary loss that promotes equal amounts of tokens getting sent to each expert and **(2)** a capacity factor (a hyperparameter) that adds room for extra tokens at each expert. We experiment with turning off the auxiliary loss during fine-tuning and using different capacity factors. Tables 5 reveals a surprising result that fine-tuning quality is not materially impacted by dropping up to 10-15% of tokens[8]. Studies on ST-MoE-32B corroborate that high capacity factors do not improve fine-tuning quality. This is in-line with findings of Yang et al. (2021) that unequal load balance may not significantly impact model quality.

Model	Train CF	Eval CF	Aux Loss	Percent Tokens Dropped	SuperGLUE ($\uparrow$)
Sparse	0.75	2.0	Yes	10.6%	86.5 ± 0.21
Sparse	1.25	2.0	Yes	0.3%	86.7
Sparse	2.0	3.0	Yes	0.0%	85.8
Sparse	4.0	5.0	Yes	0.0%	86.4
Sparse	0.75	2.0	No	15.6%	85.7
Sparse	1.25	2.0	No	2.9%	85.8
Sparse	2.0	3.0	No	0.4%	85.9
Sparse	4.0	5.0	No	0.0%	86.4

Table 5: **Sparse models are robust to dropped tokens when fine-tuning.** We find the fine-tuning quality on SuperGLUE is not impacted significantly across the values explored. Interestingly, dropping 10-15% of tokens can perform approximately as well as models that drop $< 1\%$. We also observe that load balance losses (Aux Loss) improve fine-tuning. The dropped token percentage corresponds to the fraction of dropped tokens across all expert layers at peak validation accuracy.

---

[8]Token dropping may be a form of regularization and a more extensive study may be an interesting direction for future work.

4.5 Inserting Sentinels Tokens During Fine-Tuning

Sentinel tokens denote masked sequences in the span-corruption objective (Fedus et al., 2018; Devlin et al., 2018). This differs from any fine-tuning task we would likely encounter, leading to a domain mismatch between pre-training and fine-tuning. Table 6 illustrates the difference. We examine whether modifying the fine-tuning task to look more like the pre-training task effects results.

Objective	Inputs	Targets
Span Corruption	I like **<X>** the pool **<Y>** day .	**<X>** going to **<Y>** on a sunny
Fine-Tuning	What is the capital of Illinois ?	Springfield
Fine-Tuning + Sentinels	What is the capital of Illinois ? **<X>**	**<X>** Springfield

Table 6: **Inserting sentinels during fine-tuning mimics the pre-training span objective**. We highlight the typical difference between span corruption and fine-tuning. We propose modifying the fine-tuning task to resemble pre-training by inserting sentinel tokens.

In Table 7 we find that adding sentinel tokens while fine-tuning only improves Grammar Error Correction (GEC) (Rothe et al., 2021), but not SuperGLUE. We tried to further reduce the data distribution shift by inserting multiple sentinel tokens (as would be encountered by the model while pre-training), but again found no universal benefit. However, despite no consistent benefit on held-out data, we find that training convergence is accelerated for both dense and sparse models.

Model	Insert Sentinel Tokens	SuperGLUE ($\uparrow$)	GEC ($\uparrow$)
Dense		84.9 ± 0.33	22.3 ± 0.25
Dense	✓	85.1 ± 0.25	22.1 ± 0.42
Sparse		86.6 ± 0.18	22.2 ± 0.04
Sparse	✓	86.6 ± 0.24	$\mathbf{22.9} \pm 0.09$

Table 7: **Impact of sentinel tokens for fine-tuning**. The addition of sentinel tokens (a similar concept used in Lester et al. (2021)) during fine-tuning has mixed performance on the two tasks we consider. SuperGLUE records the average score and GEC records the exact match. While we find it doesn't improve generalization, sentinel tokens can accelerate training convergence.

5 Designing Sparse Models

The design of dense models has been guided by the foundational work of Kaplan et al. (2020). But sparse models pose a myriad of additional questions: (1) How many experts to use? (2) Which routing algorithm? (3) What value for the capacity factor? (4) How does hardware change these decisions? In this section, we comment on these and offer recommendations for building Pareto efficient sparse models. Concurrently, Clark et al. (2022) provides additional design recommendations including higher layer frequency and top-1 routing as per Fedus et al. (2021).

Designing Sparse Models

1. In our setup, we recommend top-2 routing with 1.25 capacity factor and at most one expert per core.
2. The capacity factor can be changed during evaluation to adjust to new memory/compute requirements.
3. Dense layer stacking and a multiplicative bias can boost quality (Appendix C).

5.1 Setting the Number of Experts

One of the first questions is the number of experts to use. Fedus et al. (2021) presented the scaling-properties of Switch Transformer which yielded monotonic pre-training benefits (on a step basis) on

C4 up to 512-experts, Kim et al. (2021) up to 64-experts and Clark et al. (2022) up to 512-experts. But the incremental benefit quickly diminishes with many experts (>256) or equivalently, with very sparse models (<1% of experts activated).

However, reflecting on the specific hardware system can further guide this choice. The compute-to-memory ratio (operational intensity) can serve as an estimate of the efficiency of different operations (Williams et al., 2009; Shazeer, 2019). A model is *memory bound* if the time to load tensors to the computing core (e.g. ALU/MMU) greatly exceeds the time required to do the computation on the tensors. On modern GPUs and TPUs, increasing this compute to memory ratio improves the efficiency.

Returning to sparse expert models, using more than one expert per core increases memory transfer, potentially hurting efficiency. Increasing the number of experts does not change the computation done (sparse models apply a fixed amount of computation to each input), but increases the memory transfer requirement (additional expert variables must be loaded from device memory). This *decreases* the compute-to-memory ratio[9].

On our TPU system, we recommend to one expert (or less) per core. Our largest models use both data and model parallelism where data parallelism is over "rows" and model-parallelism over "columns" of the logical mesh. We use ≤ 1 expert per data parallelism row to ensure the compute-to-memory ratio is high and to reduce the cores needed for evaluation and inference. Furthermore, using less experts lets us allocate more cores to the model parallelism "column" to have more FLOPs in our model. Appendix H explains our mesh layouts for when we have fewer experts than data parallelism rows.

5.2 Choosing the Capacity Factor and Routing Algorithm

We generalize top-1 routing (Fedus et al., 2021; Roller et al., 2021) and top-2 (Shazeer et al., 2017; Lepikhin et al., 2020) to study top-n routing where each token is processed by at most n experts. In this study, all models are pre-trained for 100k steps with 1M tokens per batch and sparse models have 32 experts and are FLOP matched to T5-Large Raffel et al. (2019). We draw two key conclusions.

First, increasing both the train and eval capacity factors (CF) improves quality as seen by comparing across the segmented blocks of Table 8. For instance, top-1 routing improves by +0.011 neg. log perp. when increasing from $1.0 \rightarrow 1.25$ train CF and top-2 routing improves +0.009 increasing from $1.25 \rightarrow 2.0$ train CF. To provide context for these numbers: tripling the size of a dense model (Dense-L to Dense-XL) yields a +0.090 neg. log perp. boost. Therefore, these CF boosts are $\sim 1/10^{\text{th}}$ of that magnitude. But this comes at a cost. Increasing the capacity factor linearly increases the einsums costs, memory for activations, `all2all` communication costs, and model-parallelism `allreduce` communication costs for expert layers[10].

Second, there are small gains of top-$(n+1)$ over top-n routing given a *fixed* capacity factor (Table 8). For instance, top-2 routing improves +0.004 over top-1 at train CF of 1.25 or about $1/20^{\text{th}}$ the boost of a dense model tripling. This revises an earlier recommendation from Fedus et al. (2021). The primary difference between these experimental setups was scale of compute. Fedus et al. (2021) trained 220M-FLOP matched models for 50B tokens. We find at an 8x larger scale of training (1B-FLOP matched models for 100B tokens) there is instead a small gain to route to more than one expert. Furthermore, at the larger experimental scale, the speed difference of top-n versus top-$(n+1)$ routing is negligible. Speed differences were observed in Fedus et al. (2021) because the router computation was a larger fraction of the total model computation.

---

[9]As an exercise to the reader, verify the operational intensity of the first expert computation is $\frac{b \cdot h}{b + h \cdot e}$ with b batch size, h hidden dimension, e number of experts.

[10]`all2all` and `allreduce` costs depend on the number of devices, batch size, d_{model} and capacity factor, but not on the number of experts.

Algorithm	Train CF	Eval CF	Neg. Log Perp. ($\uparrow$)
Dense-L	—	—	-1.474
Dense-XL	—	—	-1.384
Top-1	0.75	0.75	-1.428
Top-1	0.75	2.0	-1.404
Top-2	0.75	0.75	-1.424
Top-2	0.75	2.0	-1.402
Top-1	1.0	1.0	-1.397
Top-1	1.0	2.0	-1.384
Top-2	1.0	1.0	-1.392
Top-2	1.0	2.0	-1.378
Top-1	1.25	1.25	-1.378
Top-1	1.25	2.0	-1.373
Top-2	1.25	1.25	-1.375
Top-2	1.25	2.0	-1.369
Top-2	2.0	2.0	-1.360
Top-2	2.0	3.0	-1.359
Top-3	2.0	2.0	-1.360
Top-3	2.0	3.0	-1.356

Table 8: **Comparing capacity factors (CF) and routing algorithms.** Increasing both train and eval CF improves performance. Increasing or decreasing the eval CF gives an additional lever if you have more or less compute at eval time. Next, there are smaller gains of top-$(n + 1)$ over top-n routing across capacity factors. Because the quality improves, but the speed slows as the CF increases, the Pareto efficient CF must be determined by the specific hardware system.

The specific hardware-software system will determine the optimal n and capacity factor. For instance, if the system supports fast `all2all` and `allreduce` communications, larger capacity factors and larger n in top-n routing may be optimal. However, if the `all2all` and/or `allreduce` communications are slow, smaller capacity factors may dominate. In our case, the hardware-software stack is the TPU and Mesh Tensorflow. We record the training speed of both our ST-MoE-L and ST-MoE-32B model in Table 9 as we increase the train capacity factor. As the models scale, a higher capacity factor makes the models *increasingly* slower. The ST-MoE-L does not require model parallelism (it fits within accelerators memory, which implies no additional `allreduce` communications) making it better suited for high capacity factors than our ST-MoE-32B model. For our largest model, we therefore continue to use the smaller train capacity factor of 1.25 advocated by Fedus et al. (2021) for Pareto efficiency, differing from other work which use a larger and more expensive 2.0 capacity factor (Lepikhin et al., 2020; Du et al., 2021).

Model	Train CF	Step Time (s) ($\downarrow$)
ST-MoE-L	1.25	2.397
ST-MoE-L	2.0	2.447 (+7%)
ST-MoE-32B	1.25	4.244
ST-MoE-32B	2.0	4.819 (+14%)

Table 9: **Profiling sparse models on TPUs.** Increasing the train capacity factor from 1.25 to 2.0 increases the step-time by +7% for the large (1B) model but by +14% for our 32B model. As the model size increases, we find the small quality gains of higher train capacity factors from Table 8 are more than offset by the significant 14% slow-down. Note: the step time between ST-MoE-L and ST-MoE-32B are not comparable because they used a different number of cores.

Our results in this section focus on top-n routing, but we also experimented with a variety of other routing techniques in Appendix J. We found most performed similarity or worse compared to top-n routing. However we found Batch Prioritized Routing (BPR), introduced in Riquelme et al. (2021), significantly helps performance for capacity factors less than one (Appendix D). We recommend

BPR for larger models where `all2all` and `allreduce` are more expensive and lower capacity factors are optimal.

6 EXPERIMENTAL RESULTS

Given our improvements to training stability, fine-tuning and model design, we start by validating a sparse model approximately FLOP-matched to T5-Large (Raffel et al., 2019). We conclude this section by designing and training a 269B sparse parameter model (FLOP matched to a 32B dense model) which achieves state-of-the-art quality across a wide set of NLP tasks.

We studied the SuperGLUE (Wang et al., 2019) benchmark throughout this work which consists of tasks including sentiment analysis (SST-2), word sense disambiguation (WIC), sentence similarity (MRPC, STS-B, QQP), natural language inference (MNLI, QNLI, RTE, CB), question answering (MultiRC, RECORD, BoolQ), coreference resolution (WNLI, WSC) and sentence completion (COPA) and sentence acceptability (CoLA). We often observe good performance on SuperGLUE to correlate with (but not guarantee) performance across many NLP tasks. We also include a divers set of additional benchmarks. The CNN-DM (Hermann et al., 2015) and BBC XSum (Narayan et al., 2018) datasets are used to measure the ability to summarize articles. Question answering is probed with the SQuAD dataset (Rajpurkar et al., 2016) as well as on grade-school science questions in ARC Easy and ARC Reasoning Challenge (Clark et al., 2018). And as in Roberts et al. (2020), we evaluate the knowledge of our models by fine-tuning on three closed-book question answer datasets: Natural Questions (Kwiatkowski et al., 2019), Web Questions (Berant et al., 2013) and Trivia QA (Joshi et al., 2017). Closed-book simply refers to questions posed with no supplemental reference or context material. To gauge the model's common sense reasoning we evaluate it on the Winogrande Schema Challenge (Sakaguchi et al., 2020). And finally, we test our model's natural language inference capabilities on the Adversarial NLI Benchmark (Nie et al., 2019).

6.1 ST-MoE-L

For simplicity and to cover dozens of tasks easily, we train on *mixtures* of the tasks listed rather than separately fine-tuning a model on each task. However, because the tasks vary in size considerably, equally sampling per the number of examples would over-sample large tasks and under-sample small ones. We therefore mix each task in proportion to the number of examples in its 'train' split (up to some `max_num_examples=65536`) as in Raffel et al. (2019). This means that tasks containing more than 65536 training examples are weighted as if they only contain `max_num_examples`.

Table 10 summarizes the quality of a dense T5-Large (L) model and sparse model with approximately the same number of FLOPs pre-trained for 500k steps with a 1M batch size (524B tokens) on the C4 dataset (Raffel et al., 2019). The sequence length for the encoder was 512 and 114 for the decoder. We observe improvements on the validation (dev) sets across a wide array of tasks examining natural language understanding, question answering, and summarization. As seen in Fedus et al. (2021), striking gains are observed in closed book question answering (Roberts et al., 2020).

Also, in support of the overfitting hypothesis presented in Section 4.1, we observe two of the smallest tasks CB and WSC (250 and 259 training examples, respectively), are the only ones where the sparse model does not yield gains over its dense counterpart. This again suggests that improved forms of regularization for sparse models may unleash greater performance.

6.2 ST-MoE-32B

With quality validated at the scale of T5-Large, we seek to push the capabilities of sparse models through the ST-MoE-32B. When designing this, we sought a balance between FLOPs and parameters. High-FLOP sparse models were previously unstable in Fedus et al. (2021) in our setting (i.e. encoder-decoder models, Adafactor optimizer), but the router z-loss enabled us to proceed. For computational efficiency, we expanded the hidden size of the experts (d_{ff} in Table 11 below)[11]. Finally, we increased the d_{kv} to 128 for better performance on our hardware. The most salient changes are fewer overall parameters and more FLOPs per token relative to both Switch-C and Switch-XXL. Our

---

[11]`allreduce` activation communications introduced through model parallelism are independent of the hidden size, but not the model dimension, making it a good choice to increase.

Name	Metric	Split	Dense-L (↑)	ST-MoE-L (↑)	Gain (%)
SQuADv2	F1	dev	94.0	**94.5**	+1%
SQuADv2	acc	dev	87.6	**88.1**	+1%
SuperGLUE	avg	dev	85.1	**87.4**	+3%
BoolQ	acc	dev	87.1	**88.6**	+2%
Copa	acc	dev	83.0	**91.0**	+10%
RTE	acc	dev	91.0	**92.1**	+1%
WiC	acc	dev	70.4	**74.0**	+5%
MultiRC	F1	dev	83.9	**86.0**	+3%
WSC	acc	dev	**95.2**	93.3	−2%
ReCoRD	acc	dev	85.7	**88.9**	+4%
CB	acc	dev	**100**	98.2	−2%
XSum	ROUGE-2	dev	19.9	**21.8**	+10%
CNN-DM	ROUGE-2	dev	20.3	**20.7**	+2%
WinoGrande (XL)	acc	dev	75.4	**81.7**	+8%
ANLI (R3)	acc	dev	54.3	**57.3**	+6%
ARC-Easy	acc	dev	63.5	**75.4**	+19%
ARC-Challenge	acc	dev	50.2	**56.9**	+13%
Closed Book TriviaQA	acc	dev	28.1	**33.8**	+20%
Closed Book NatQA	acc	dev	27.2	**29.5**	+8%
Closed Book WebQA	acc	dev	30.5	**33.2**	+9%

Table 10: **Fine-tuning performance of FLOP-matched dense and sparse models.** Comparison of the dense-L baseline and the sparse FLOP-matched version (higher numbers better). We observe consistent gains across diverse tasks, using approximately the same amount of computation. The only two tasks without improvement from the sparse model are the two smallest: CB with 250 training examples and WSC with 259.

ST-MoE-32B has "only" 269B parameters and is approximately FLOP-matched to a dense Transformer with 32B parameters. The reduced parameter count from Switch-C and Switch-XXL eases the burden for both serving and fine-tuning. Finally, we use the sparse-dense stacking described in Appendix C.

We pre-train for 1.5T tokens on a mixture of English-only C4 dataset (Raffel et al., 2019) and the dataset from GLaM (Du et al., 2021) summarized in Appendix E. We use 1M tokens per batch, the Adafactor optimizer with default hyperparameters, and a learning rate warm-up of 10k steps followed by inverse square root decay. Our model follows the initialization scheme proposed in Fedus et al. (2021).

Table 12 evaluates our ST-MoE-32B model against previous state-of-the-art approaches using inference-only (zero-shot, one-shot) as well as fine-tuning. On SuperGLUE, our model improves upon the prior state-of-the-art model, achieving an average score of 91.2 on the test server (93.2 validation accuracy) which is over one percentage point beyond estimated human capability. For both summarization datasets, XSum and CNN-DM, our model achieves state-of-the-art without additional changes to training or fine-tuning (Raffel et al., 2019; Liang et al., 2021). ST-MoE-32B improves the current state-of-the-art on the test server submissions for both ARC Easy (92.7 → 94.8) and ARC Challenge (81.4 → 86.5). On two of the three closed book QA tasks, we improve over the prior state-of-the-art. Closed book WebQA achieves a 47.4 accuracy (prior best of 42.8 from Roberts et al. (2020) and exceeds results from the zero-shot performance of the ERNIE 3.0 Titan 260B dense parameter model (Wang et al., 2021)). Closed book NatQA improves to 41.9 accuracy (prior best of 41.5 from Karpukhin et al. (2020)). We find significant improvements on adversarially constructed datasets (ANLI R3 and WinoGrande XL). ANLI R3 (Nie et al., 2019) improves the state-of-the-art to 74.7 (prior best of 53.4).

We note some weaknesses in our model. ST-MoE-32B has lackluster performance on the small SQuAD dataset, with an exact match score of 90.8 which falls short of the older benchmark set by the T5-XXL of 91.3. Furthermore, while setting a new state-of-the-art for SuperGLUE in aggregate,

Model	Parameters	FLOPs/seq	d_{model}	FFN$_{\text{GEGLU}}$	d_{ff}	d_{kv}
Dense-L	0.8B	645B	1024	✓	2816	64
T5-XXL	11.1B	6.3T	4096	✓	10240	64
Switch-XXL	395B	6.3T	4096	✓	10240	64
Switch-C	1571B	890B	2080		6144	64
ST-MoE-L	4.1B	645B	1024	✓	2816	64
ST-MoE-32B	269B	20.2T	5120	✓	20480	128

Model	Num. Heads	Num. Layers	Num. Experts	Expert Layer Freq.	Sparse-Dense
Dense-L	16	27	–	–	
T5-XXL	64	24	–	–	
Switch-XXL	64	24	64	1/4	
Switch-C	32	15	2048	1/1	
ST-MoE-L	16	27	32	1/4	✓
ST-MoE-32B	64	27	64	1/4	✓

Table 11: **Model comparisons.** A comparison of the Dense-L and T5-XXL, the two largest Switch Transformer variants (Switch-XXL and Switch-C), and the ST-MoE-L and ST-MoE-32B. d_{model} refers to the model hiddenstate size and d_{ff} is the internal size of the FFN layer. d_{kv} is the dimension of each attention head. Expert Layer Freq. is the fraction of FFN layers replaced with a sparse layer. Sparse-Dense refers to the architectural variant described in Appendix C.

			Previous Best (↑)			Ours (↑)
Name	Metric	Split	Zero-Shot	One-Shot	Fine-Tune	Fine-Tune
SQuADv2	F1	dev	68.3[e]	70.0[e]	96.2[a]	**96.3**
SQuADv2	acc	dev	62.1[e]	64.6[e]	**91.3**[a]	90.8
SuperGLUE	avg	test	–	–	90.9	**91.2**
BoolQ	acc	dev/test	83.0[e]	82.8[e]	92.0	**92.4**
Copa	acc	dev/test	91.0[d]	92.0[e]	98.2	**99.2**
RTE	acc	dev/test	68.8[e]	71.5[e]	**94.1**	93.5
WiC	acc	dev/test	50.5[e]	52.7[e]	**77.9**	77.7
MultiRC	F1	dev/test	72.9[d]	72.9[d]	88.6	**89.6**
WSC	acc	dev/test	84.9[e]	83.9[e]	**97.3**	96.6
ReCoRD	acc	dev/test	90.3[e]	90.8[e]	**96.4**	95.1
CB	acc	dev/test	46.4[d]	73.2[e]	**99.2**	98.0
XSum	ROUGE-2	test	–	–	24.6[h]	**27.1**
CNN-DM	ROUGE-2	test	–	–	21.6[a]	**21.7**
WinoGrande XL	acc	dev	73.4[e]	73.2[d]	–	96.1
ANLI R3	acc	test	40.9[e]	40.8[e]	53.4	**74.7**
ARC-Easy	acc	test	71.9[e]	76.6[e]	92.7[g]	**95.2**
ARC-Challenge	acc	test	51.4	53.2	81.4[g]	**86.5**
CB TriviaQA	em	dev	68.0[e]	**74.8**[e]	61.6[b]	62.3
CB NatQA	em	test	21.5[e]	23.9[e]	41.5[c]	**41.9**
CB WebQA	em	test	38.0[f]	25.3	42.8[b]	**47.4**

Table 12: **ST-MoE-32B versus previous best for inference-only techniques and fine-tuned models.** A split of "dev/test" refers to dev split for Zero-Shot and One-Shot and test split for Fine-Tune quality. Data not available filled in with "–". Superscript letters denote the result: [a]: Raffel et al. (2019) [b]: Roberts et al. (2020) [c]: Karpukhin et al. (2020), [d]: Brown et al. (2020), [e]: Du et al. (2021), [f]: Wang et al. (2021), [g]: UnifiedQA + ARC MC/DA + IR, [h]: Zhang et al. (2020).

certain tasks, including small ones like CB, WSC, fail to improve. Finally, on closed book Trivia QA, our model improves over the fine-tuned baseline with SSM from Roberts et al. (2020), but fails to produce gains over both GPT-3 and GLAM.

While not the focus of this paper, we present the quality differential between recent advances in inference-only techniques like few-shot learning and fine-tuning on these tasks (GPT-3 (Brown et al., 2020), GLAM (Du et al., 2021) and Gopher (Rae et al., 2021)). As expected and observed previously, fine-tuning outperforms zero/one-shot learning, but has the disadvantage of requiring additional training and different models for each task.

7   TRACING TOKENS THROUGH THE MODEL

Thus far we have presented quantitative measures and performance metrics. We change tack to explore *qualitative* features by visualizing how tokens are routed among the experts. We do so by passing a batch of tokens to the model and manually inspecting token assignment at each layer. We consider our ST-MoE-L model pre-trained either on the monolingual C4 corpus (Raffel et al., 2019) or on the multilingual mC4 corpus (Xue et al., 2020). On both the encoder and the decoder, the model has six sparse layers, each with 32 experts.

Preliminaries

The span corruption objective is to recover spans of variable-length contiguous segments masked out in the inputs. This is formatted as:

Inputs: *I went to <extra_id_0> to buy <extra_id_1>*

Targets: *<extra_id_0> the store <extra_id_1> milk*

In our encoder-decoder architecture, the inputs will be passed to the encoder and targets to the decoder.

Each group of tokens is routed jointly with load balancing across experts incentivized by an auxiliary loss as proposed in Shazeer et al. (2017) (see Appendix A for details). Tokens compete for expert assignment against other tokens in their group, rather than the entire batch, and expert specialization is heavily influenced by the distribution of tokens in each group. The notion of groups is introduced to limit the cost of dispatching and gathering the correct tokens to the correct experts.

7.1   ENCODER EXPERTS EXHIBIT SEMANTIC SPECIALIZATION

Our first observation is that, at each layer, at least one expert specializes in sentinel tokens (mask tokens that represent blanks to fill-in). Additionally, some encoder experts exhibit clear semantic specialization, with some experts primarily operating on punctuation, verbs, proper names, counting, etc. Table 13 presents a few notable example of semantic specialization across encoder experts. And while we find many instances of specialization, these have been specifically extracted from many examples without a clear semantic specialization.

7.2   DECODER EXPERTS LACK SEMANTIC SPECIALIZATION

In contrast, *expert specialization is far less noticeable in the decoder*. Not only are sentinel tokens routed somewhat uniformly across decoder experts (see Table 14), but we also do not observe semantically meaningful specialization in decoder experts.

We hypothesize that this lack of semantically meaningful expert specialization is caused by the *distribution of target tokens induced by the span corruption objective*. In particular, **(a)** a smaller number of tokens are routed jointly in the decoder due to longer sequence lengths in the encoder (e.g. group size is 2048 in the encoder vs 456 in the decoder in our setup) and **(b)** a higher proportion of tokens are sentinel tokens in the decoder. As a result, target tokens in each group typically cover a smaller semantic space (compared to the encoder), perhaps explaining the lack of semantic expert specialization in the decoder. This intricate interplay between the architecture and the train-

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

Table 13: **Notable examples of semantic specialization in encoder experts**. We find experts that specialize in punctuation, conjunctions & articles, verbs, visual descriptions, proper names, counting & numbers. Across all layers (not shown), we observe experts that primarily operate on sentinel tokens (marked as <extra_id_x>). Note that a SentencePiece model (Kudo and Richardson, 2018) will split a token if it doesn't exist in the vocabulary, e.g. Kenneth may become Ken, ne, th.

ing objective invites further research on better leveraging sparsity and expert specialization in the decoder. Alternatively, future work could study simply removing the experts in the decoder layer, which also confers benefits during autoregressive decoding (Kudugunta et al., 2021a).

	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6	Uniform (32-experts)
Encoder	2.2	1.8	1.6	1.7	1.7	1.2	3.5
Decoder	3.4	3.4	3.4	3.4	3.4	3.4	3.5

Table 14: **Entropy of routed sentinel tokens across encoder and decoder layers.** We support our qualitative observation that encoder experts specialize, but decoder expert don't by computing the entropy over the routing for sentinel tokens. The encoder routing entropy is low, but the decoder router is high entropy, and nearly equal to uniform routing. Because each layer has 32-experts, a completely uniform distribution has entropy of 3.5.

7.3 Multilingual Experts Specialize Semantically, But Not by Language

We next consider a *multilingual* sparse model pretrained on a mixture of different languages and inspect the expert specialization in the encoder. As in the monolingual case, we find strong evidence of semantically meaningful expert specialization. Table 15 presents some examples of experts specializing in sentinel tokens, numbers, conjunctions & articles and proper names.

Semantic specialization	Routed tokens
Sentinel tokens	to <extra_id_6>to til <extra_id_9> <extra_id_10>to <extra_id_14><extra_id_17> <extra_id_19><extra_id_20><extra_id_21>...
Numbers	$50 comment .10.2016 ! 20 20 3 ! 5 1. ! 91 ? né ? 2 17 4 17 11 17 8 & 11 & 22:30 02 2016.) iOS
Conjunctions & Articles	of of of their their of any this this your your am von this of Do of of This these our 的的于的在的在的 le les Le la di la sur sur 136 sur ののするのというのし
Prepositions & Conjunctions	For for or for for or for from because https during https 并与和par c Pour à a par trè pour pour pour pour pour c とやのに でででなので- and and + c between and and
Proper names	Life Apple iOS A IGT 众莫HB F HB A K A OPP OK HB A Gia C Gia C P Scand Wi G H Z PC G Z ハイ PC G Ti CPU PC PC A キット OS

Table 15: **Examples of semantic specialization in multilingual experts (encoder)**. Multilingual experts also exhibit semantic specialization, which sometimes spans across different languages (e.g. "for" and "pour"). Experts trained on multilingual mixtures do not exhibit language specialization.

One might expect experts to specialize in languages, which appears as a natural criterion for divvying up batches of data among experts. However, we find no evidence of language specialization (see Table 15). Routers instead pass tokens from English, Japanese, French and Chinese indiscriminately and the experts appear to be *multilingual*. But this lack of language specialization is less surprising when considering the mechanism of token routing and load balancing. Since each group of tokens may only contain one, to at most a few, languages (a group usually consists of 2-4 sequences in our setup), then all experts are encouraged to handle tokens from all languages. We experimented with a global load balance loss, however, this usually results in worse load-balance and worse model performance, so we leave further improving multilingual expert models as an area of open work (Section 9).

Our visualization reveals apparent specialization learned in our models (Tables 13, 15) for the encoder layers. Other expert specializations were also observed in the appendix of Shazeer et al. (2017). However, this leads to an interesting question of how architectures that eliminate learned routing Roller et al. (2021); Zuo et al. (2021) appear to perform well. An extensive study of the scaling properties of learned versus random routing could prove helpful as future work and help guide us to a better understanding of routing behavior.

8 Related Work

Mixture-of-Experts (MoE) date back at least three decade history to the work of Jacobs et al. (1991); Jordan and Jacobs (1994). In initial concepts, the MoE defined the entire neural network akin to ensemble methods. But later Eigen et al. (2013) extended the idea of including MoE as a *component* as part of deeper networks. Shazeer et al. (2017) then scaled this idea to a 137B parameter model to achieve state-of-the-art in machine translation. Most of the later work (including ours) follows this MoE as a component approach.

Scale in natural language processing. The remarkable success of scale in natural language processing (Kaplan et al., 2020; Brown et al., 2020) has reinvigorated MoE research evidenced by a

surge of recent work (Lepikhin et al., 2020; Fedus et al., 2021; Yang et al., 2021; Kim et al., 2021; Du et al., 2021; Artetxe et al., 2021; Zuo et al., 2021; Clark et al., 2022). Sparse expert models have been proposed as a method to achieve the results of large-scale dense models, more efficiently. Fedus et al. (2021) showed a 4x pre-train speed-up over T5-XXL (Raffel et al., 2019) and Du et al. (2021) matched the quality of GPT-3 (Brown et al., 2020) using only $1/3$ of the energy. And in the span of the last twelve months, a milestone of efficiently training trillion parameter deep neural networks has been achieved by multiple groups (Fedus et al., 2021; Yang et al., 2021; Du et al., 2021), and most recently, Lin et al. (2021) introduced techniques to train a 10T parameter model. One side note is that the recent significant successes of sparse expert models have often been in settings with a lot of data and no distribution shift – two examples being language modeling/span corruption and machine translation (Shazeer et al., 2017; Lepikhin et al., 2020; Kim et al., 2021; Fedus et al., 2021). In contrast, discrepancies between strong pre-training quality and poor fine-tuning quality for sparse models have been observed in Fedus et al. (2021); Narang et al. (2021); Artetxe et al. (2021), but we expect advances in regularization techniques to continue to improve downstream quality.

Towards better routing algorithms. BASE layers (Lewis et al., 2021) recasts token routing as a linear assignment problem – removing the need for load balancing auxiliary losses. This work also demonstrated the efficacy of a single expert layer. Clark et al. (2022) studies in depth the scaling properties of a few different routing algorithms and propose their own variant of BASE layers that uses an optimal transport formulation. Yang et al. (2021) introduces the M6-T architecture and expert prototyping which splits experts into different groups and applies k top-1 routing procedures (contrasting with the top-k routing commonly used elsewhere). Hazimeh et al. (2021) proposed a continuously differentiable sparse gate with demonstrated improvements over vanilla top-k gating. Other work (Bengio et al., 2016) considered casting the routing selection as a reinforcement learning problem. More radical versions remove learning the routing entirely. Hash layers (Roller et al., 2021) shows *random* fixed routing (per hash functions) led to competitive performance with learned routing. Zuo et al. (2021) also proposed an algorithm which randomly selects experts during training and inference and found gains of 2 BLEU points over Switch Transformers and competitive scores with the larger models of Kim et al. (2021). Finally, Fan et al. (2021) designs an architecture with explicit language-specific sublayers (rather than allowing arbitrary routing as done in Lepikhin et al. (2020)) to yield gains of +1 BLEU.

Sparse expert models in other modalities. MoE and sparse experts model have also advanced results in modalities aside from language. Riquelme et al. (2021) designed a 15B parameter V-MoE to match state-of-the-art ImageNet (Deng et al., 2009) models with fewer computational resources. Lou et al. (2021) similarly showed a benefit over dense vision models by using MoE layers across both image patch and channel dimensions. Additionally, Automatic Speech Recognition has been improved by the SpeechMoE variants (You et al., 2021a;b). Kumatani et al. (2021) reduced word error rates using MoE models in Sequence-to-Sequence Transformer and Transformer Transducer.

Improving deployment of sparse models. Initial expert designs (including this work) route each token separately to experts at that layer. One issue is that these type of architectures may be burdensome to serve since it requires sufficient memory for storing the parameters. Distillation was shown in Fedus et al. (2021) to be moderately effective, but recent approaches modified the routing to instead route full sentences or tasks (Kudugunta et al., 2021b; Zuo et al., 2021) which then permits extraction of sub-networks at time of serving (e.g. deploy only the network associated with the new task). As an alternative to distillation, Kim et al. (2021) considers directly pruning away experts not essential to the task of interest.

Multitask learning with MoE. We conclude our tour of recent MoE research with successes in multitask settings. Ma et al. (2018) recommended using a separate gating or router network for each task, an idea that may soon be revisited for Transformer architectures. Finally, Gururangan et al. (2021) recommends even greater modularity of language models and conditionally activates experts based on the domain/task label or by an inferred label.

9 DISCUSSION

While this work is on sparse models, these models intersect with many other interesting topics in machine learning such as adaptive computation, low-precision training, scaling principles, and

neural network architecture advances. Our discussion therefore covers a broader range of topics surfaced during this research.

Unpredictable dynamics when pre-training on multilingual data. We often observe that the same model pre-trained on multilingual data will yield smaller pre-training speed-ups and be more unstable. One hypothesis is that this is due to the variance of sequences per group across batches. As a reminder, we encourage tokens *in a group* to be load-balanced. There are usually only 2-8 sequences per group (higher becomes expensive) where each sequence is written in a single language. Therefore, at most 2-8 languages must be balanced across experts – even when training with over 100 languages. This leads to high variance across groups and batches, resulting in chaotic and unpredictable routing. In a follow-up experiment (just highlighted for brevity), we pre-trained on a mixture of English C4 plus a small fraction of a fine-tuning task which similarly resulted in an unstable model.

The robustness of sparse models. Despite a paper focused on the details of sparse model-particulars, zooming out we find them to be robust to a wide set of hyperparameters and architectural changes. Sparse models obtain great performance under a variety of routing algorithms, dropping high fractions of tokens, and different hyperparameters. While we did point out the importance of tuning the batch size and learning rate for fine-tuning, our intuition, in-line with Kaplan et al. (2020), is that the real winner is scale. For instance, Table 8 shows larger gains to be had by simply increasing the capacity factor (i.e. FLOPs) rather than by more sophisticated routing (i.e. algorithms).

Adaptive computation. Sparse models are a subclass of adaptive computation models since each input gets different computation applied to it. In sparse models a token is routed to the expert(s) of its choosing. When capacity factors are less than one, the model learns to not apply computation to certain tokens. This has shown promise in computer vision (Riquelme et al., 2021) and our language experiments (Appendix D). We envision future models expanding this through heterogeneous experts (e.g. each expert applies differing computation). Intuitively, different input examples will likely require different amounts of processing depending on difficulty. Future models in this direction will be efficiently enabled through emerging computing infrastructures (Dean, 2021).

Generalizing findings from small to large scale. A key issue we faced throughout our work was identifying small scale models and training setups that reflect larger scale experiments. This was evident in our stability studies in Section 3 where experiments had to be run with XL sized models to surface relevant dynamics. For our architecture and routing algorithm experiments, we often find improvements vanish, or even reverse, when models are trained for longer or made larger. As one example, the top-n findings of Fedus et al. (2021) were reversed in our 8x larger-scale experiments presented here, which revealed small boosts of top-$(n + 1)$ routing over top-n routing (see Table 8).

Training models with even lower precision. The best method we found to stabilize our models without hurting (and sometimes improving) quality was the router z-loss. This is an auxiliary loss that encourages the model logits to have values smaller in absolute magnitude. Given the max range of numbers `float32` and `bfloat16` can support ($\sim 3e^{38}$), this leads us to believe most of this range is not needed, and compressing it actually might improve model training dynamics. Therefore, future precision formats might take into account more compressed exponential ranges to train certain classes of models.

Designing new operations with more multiplicative interactions. Section 3.1 shows that operations with more multiplicative interactions than additions, or those that don't accumulate over many numbers, improve model performance. We test this further by injecting more multiplicative interactions into expert layers which speedup pre-training by 4% without any change to step-time (Appendix C). We think this hints at promising architectural improvements for models and could be a good design principle. Recently depthwise convolutions, which only accumulate 3-5 elements, have also been shown to greatly improve Transformer performance (So et al., 2021). These operations are especially exciting as elementwise multiplications typically do not introduce any communication overhead when using model parallelism (which makes operations like depthwise convolutions and our multiplicative interactions very efficient). While we did note these methods to increase model instabilities in Section 3.1, using the router z-loss in our models prevented any further instabilities.

Constrain activations to alleviate other undesirable model scaling dynamics. We observed two additional sources of training instability. **(1)** Encoder-decoder models are more unstable than decoder only models (for fixed amount of FLOPs). Encoder-decoder models have a higher ratio of attention layers (e.g. more exponential functions) due to having both self-attention and enc-dec attention layers for each FFN on the decoder. **(2)** Deeper models are more unstable than shallower models for a fixed amount of FLOPs. Deeper models also introduce more exponential functions through additional attention layers. We hypothesize that a contributing factor to both of these observations is simply the increased number of exponential functions found in the network. Future work could look at resolving these training dynamics by adding z-loss penalties to the attention softmaxes for non-sparse models, especially since we observed adding them didn't change model quality.

Dense and sparse models depend differently on hyperparameters. Our fine-tuning analysis in Section 4.3 shows optimal fine-tuning hyperparameters differ significantly between dense and sparse models. In certain settings, fine-tuning hyperparamters that worked well for the dense model masked any improvements from the sparse model (despite large pre-training speedups). For new model classes, we recommend researchers and practitioners to extensively test key hyperparameters before prematurely abandoning a method.

10 CONCLUSION

We temper the over-exuberance for scale in Fedus et al. (2021) by showing how a model with 1/5th the size, but with a better balance of computation (FLOPs) to parameters – is a more effective sparse learner. Furthermore, this improves the usability of sparse models since it can be deployed with less memory overhead. Using our sparse model variant, we achieve SOTA across a wide range of the most competitive public benchmarks. We hope this work shows the power of model sparsity and accelerates the adoption of such models.

REFERENCES

Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, Giri Anantharaman, Xian Li, Shuohui Chen, Halil Akin, Mandeep Baines, Louis Martin, Xing Zhou, Punit Singh Koura, Brian O'Horo, Jeff Wang, Luke Zettlemoyer, Mona Diab, Zornitsa Kozareva, and Ves Stoyanov. Efficient large scale language modeling with mixtures of experts, 2021.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. Conditional computation in neural networks for faster models, 2016.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1533–1544, 2013.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Aidan Clark, Diego de las Casas, Aurelia Guy, Arthur Mensch, Michela Paganini, Jordan Hoffmann, Bogdan Damoc, Blake Hechtman, Trevor Cai, Sebastian Borgeaud, et al. Unified scaling laws for routed language models. *arXiv preprint arXiv:2202.01169*, 2022.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *International conference on machine learning*, pages 933–941. PMLR, 2017.

Marie-Catherine De Marneffe, Mandy Simons, and Judith Tonhauser. The commitmentbank: Investigating projection in naturally occurring discourse. In *proceedings of Sinn und Bedeutung*, volume 23, pages 107–124, 2019.

Jeff Dean. Introducing pathways: A next-generation ai architecture. *Google AI Blog*, 2021.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization, 2021.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Nan Du, Yanping Huang, Andrew M. Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten Bosma, Zongwei Zhou, Tao Wang, Yu Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathy Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc V Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. Glam: Efficient scaling of language models with mixture-of-experts, 2021.

David Eigen, Marc'Aurelio Ranzato, and Ilya Sutskever. Learning factored representations in a deep mixture of experts. *arXiv preprint arXiv:1312.4314*, 2013.

Angela Fan, Shruti Bhosale, Holger Schwenk, Zhiyi Ma, Ahmed El-Kishky, Siddharth Goyal, Mandeep Baines, Onur Celebi, Guillaume Wenzek, Vishrav Chaudhary, et al. Beyond english-centric multilingual machine translation. *Journal of Machine Learning Research*, 22(107):1–48, 2021.

William Fedus, Ian Goodfellow, and Andrew M Dai. Maskgan: Better text generation via filling in the_. *arXiv preprint arXiv:1801.07736*, 2018.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

Suchin Gururangan, Mike Lewis, Ari Holtzman, Noah A. Smith, and Luke Zettlemoyer. Demix layers: Disentangling domains for modular language modeling, 2021.

Hussein Hazimeh, Zhe Zhao, Aakanksha Chowdhery, Maheswaran Sathiamoorthy, Yihua Chen, Rahul Mazumder, Lichan Hong, and Ed H. Chi. Dselect-k: Differentiable selection in the mixture of experts with applications to multi-task learning, 2021.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching machines to read and comprehend. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28, pages 1693–1701. Curran Associates, Inc., 2015. URL https://proceedings.neurips.cc/paper/2015/file/afdec7005cc9f14302cd0474fd0f3c96-Paper.pdf.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

Michael I Jordan and Robert A Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural computation*, 6(2):181–214, 1994.

Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. *arXiv preprint arXiv:1705.03551*, 2017.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Vladimir Karpukhin, Barlas Ouz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. Dense passage retrieval for open-domain question answering, 2020.

Young Jin Kim, Ammar Ahmad Awan, Alexandre Muzio, Andres Felipe Cruz Salinas, Liyang Lu, Amr Hendy, Samyam Rajbhandari, Yuxiong He, and Hany Hassan Awadalla. Scalable and efficient moe training for multitask multilingual models, 2021.

Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*, 2018.

Sneha Kudugunta, Yanping Huang, Ankur Bapna, Maxim Krikun, Dmitry Lepikhin, Minh-Thang Luong, and Orhan Firat. Beyond distillation: Task-level mixture-of-experts for efficient inference. *arXiv preprint arXiv:2110.03742*, 2021a.

Sneha Kudugunta, Yanping Huang, Ankur Bapna, Maxim Krikun, Dmitry Lepikhin, Minh-Thang Luong, and Orhan Firat. Beyond distillation: Task-level mixture-of-experts for efficient inference. *arXiv preprint arXiv:2110.03742*, 2021b.

Kenichi Kumatani, Robert Gmyr, Felipe Cruz Salinas, Linquan Liu, Wei Zuo, Devang Patel, Eric Sun, and Yu Shi. Building a great multi-lingual teacher with sparsely-gated mixture of experts for speech recognition, 2021.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

Mike Lewis, Shruti Bhosale, Tim Dettmers, Naman Goyal, and Luke Zettlemoyer. Base layers: Simplifying training of large, sparse models. *arXiv preprint arXiv:2103.16716*, 2021.

Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021.

Xiaobo Liang, Lijun Wu, Juntao Li, Yue Wang, Qi Meng, Tao Qin, Wei Chen, Min Zhang, and Tie-Yan Liu. R-drop: Regularized dropout for neural networks, 2021.

Junyang Lin, An Yang, Jinze Bai, Chang Zhou, Le Jiang, Xianyan Jia, Ang Wang, Jie Zhang, Yong Li, Wei Lin, Jingren Zhou, and Hongxia Yang. M6-10t: A sharing-delinking paradigm for efficient multi-trillion parameter pretraining, 2021.

Yuxuan Lou, Fuzhao Xue, Zangwei Zheng, and Yang You. Sparse-mlp: A fully-mlp architecture with conditional computation, 2021.

Jiaqi Ma, Zhe Zhao, Xinyang Yi, Jilin Chen, Lichan Hong, and Ed H Chi. Modeling task relationships in multi-task learning with multi-gate mixture-of-experts. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1930–1939, 2018.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.

Sharan Narang, Hyung Won Chung, Yi Tay, William Fedus, Thibault Fevry, Michael Matena, Karishma Malkan, Noah Fiedel, Noam Shazeer, Zhenzhong Lan, et al. Do transformer modifications transfer across implementations and applications? *arXiv preprint arXiv:2102.11972*, 2021.

Shashi Narayan, Shay B Cohen, and Mirella Lapata. Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization. *arXiv preprint arXiv:1808.08745*, 2018.

Arvind Neelakantan, Luke Vilnis, Quoc V Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *arXiv preprint arXiv:1511.06807*, 2015.

Yixin Nie, Adina Williams, Emily Dinan, Mohit Bansal, Jason Weston, and Douwe Kiela. Adversarial nli: A new benchmark for natural language understanding. *arXiv preprint arXiv:1910.14599*, 2019.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. 2022.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. PMLR, 2013.

David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew Johnson, Blake Hechtman, Laura Weidinger, Iason Gabriel, William Isaac, Ed Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Lorrayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling language models: Methods, analysis & insights from training gopher, 2021.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

Carlos Riquelme, Joan Puigcerver, Basil Mustafa, Maxim Neumann, Rodolphe Jenatton, André Susano Pinto, Daniel Keysers, and Neil Houlsby. Scaling vision with sparse mixture of experts. *arXiv preprint arXiv:2106.05974*, 2021.

Adam Roberts, Colin Raffel, and Noam Shazeer. How much knowledge can you pack into the parameters of a language model? *arXiv preprint arXiv:2002.08910*, 2020.

Stephen Roller, Sainbayar Sukhbaatar, Arthur Szlam, and Jason Weston. Hash layers for large sparse models. *arXiv preprint arXiv:2106.04426*, 2021.

Sascha Rothe, Jonathan Mallinson, Eric Malmi, Sebastian Krause, and Aliaksei Severyn. A simple recipe for multilingual grammatical error correction. *arXiv preprint arXiv:2106.03830*, 2021.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8732–8740, 2020.

Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *Advances in neural information processing systems*, 29:901–909, 2016.

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

Noam Shazeer. Glu variants improve transformer, 2020.

Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR, 2018.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, pages 10414–10423, 2018.

Sam Shleifer, Jason Weston, and Myle Ott. Normformer: Improved transformer pretraining with extra normalization. *arXiv preprint arXiv:2110.09456*, 2021.

David R So, Wojciech Mańke, Hanxiao Liu, Zihang Dai, Noam Shazeer, and Quoc V Le. Primer: Searching for efficient transformers for language modeling. *arXiv preprint arXiv:2109.08668*, 2021.

Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014. URL `http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf`.

Nassim Nicholas Taleb. *Antifragile: Things that gain from disorder*, volume 3. Random House Incorporated, 2012.

Yi Tay, Mostafa Dehghani, Jinfeng Rao, William Fedus, Samira Abnar, Hyung Won Chung, Sharan Narang, Dani Yogatama, Ashish Vaswani, and Donald Metzler. Scale efficiently: Insights from pre-training and fine-tuning transformers. *arXiv preprint arXiv:2109.10686*, 2021.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems. In *Advances in Neural Information Processing Systems*, pages 3266–3280, 2019.

Shuohuan Wang, Yu Sun, Yang Xiang, Zhihua Wu, Siyu Ding, Weibao Gong, Shikun Feng, Junyuan Shang, Yanbin Zhao, Chao Pang, et al. Ernie 3.0 titan: Exploring larger-scale knowledge enhanced pre-training for language understanding and generation. *arXiv preprint arXiv:2112.12731*, 2021.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Linting Xue, Noah Constant, Adam Roberts, Mihir Kale, Rami Al-Rfou, Aditya Siddhant, Aditya Barua, and Colin Raffel. mt5: A massively multilingual pre-trained text-to-text transformer. *arXiv preprint arXiv:2010.11934*, 2020.

An Yang, Junyang Lin, Rui Men, Chang Zhou, Le Jiang, Xianyan Jia, Ang Wang, Jie Zhang, Jiamang Wang, Yong Li, Di Zhang, Wei Lin, Lin Qu, Jingren Zhou, and Hongxia Yang. M6-t: Exploring sparse expert models and beyond, 2021.

Zhao You, Shulin Feng, Dan Su, and Dong Yu. Speechmoe: Scaling to large acoustic models with dynamic routing mixture of experts, 2021a.

Zhao You, Shulin Feng, Dan Su, and Dong Yu. Speechmoe2: Mixture-of-experts model with improved routing, 2021b.

Biao Zhang and Rico Sennrich. Root mean square layer normalization. *arXiv preprint arXiv:1910.07467*, 2019.

Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter J. Liu. Pegasus: Pre-training with extracted gap-sentences for abstractive summarization, 2020.

Sheng Zhang, Xiaodong Liu, Jingjing Liu, Jianfeng Gao, Kevin Duh, and Benjamin Van Durme. Record: Bridging the gap between human and machine commonsense reading comprehension. *arXiv preprint arXiv:1810.12885*, 2018.

Simiao Zuo, Xiaodong Liu, Jian Jiao, Young Jin Kim, Hany Hassan, Ruofei Zhang, Tuo Zhao, and Jianfeng Gao. Taming sparsely activated transformer with stochastic experts, 2021.

A   TOKEN LOAD BALANCE DESCRIPTION

The auxiliary load balancing loss from Shazeer et al. (2017) is also used to here to balance tokens across experts. Assume we have N experts indexed by $i = 1$ to N and a batch $\mathcal{B}$ with T tokens. The auxiliary loss is computed as the scaled dot-product between vectors f and P,

$$\text{loss} = \alpha \cdot N \cdot \sum_{i=1}^{N} f_i \cdot P_i \tag{7}$$

where f_i is the fraction of tokens dispatched to expert i,

$$f_i = \frac{1}{T} \sum_{x \in \mathcal{B}} \mathbb{1}\{\text{argmax } p(x), i\} \tag{8}$$

and P_i is the fraction of the router probability allocated for expert i, [2]

$$P_i = \frac{1}{T} \sum_{x \in \mathcal{B}} p_i(x) \tag{9}$$

Since we seek uniform routing of the batch of tokens across the N experts, we desire both vectors to have values of $1/N$. The auxiliary loss of Equation 7 encourages uniform routing since it is minimized under a uniform distribution. The objective can also be differentiated as the P-vector is differentiable, but the f-vector is not. The final loss is multiplied by expert count N to keep the loss constant as the number of experts varies since under uniform routing $\sum_1^N (f_i \cdot P_i) = \sum_1^N (\frac{1}{N} \cdot \frac{1}{N}) = \frac{1}{N}$. Finally, a hyperparameter α is a multiplicative coefficient for these auxiliary losses; throughout this work we use an $\alpha = 10^{-2}$ which was sufficiently large to ensure load balancing while small enough to not to overwhelm the primary cross-entropy objective.

B   ROUTER Z-LOSS TRAINING DYNAMICS

Figure 7 plots the router z-loss from Equation 5 across a coefficient sweep where the best value of $c_z = 0.001$ is plotted in green for the encoder and decoder.
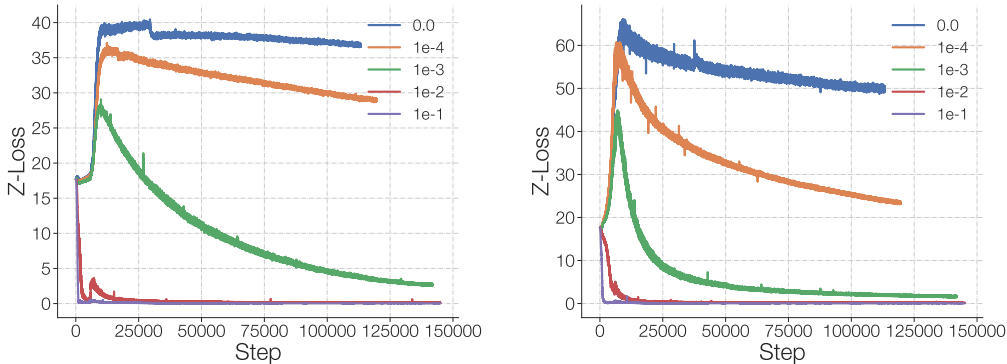
Figure 7: **Sweeping loss coefficient (c_z) for Router Z-Loss.** We plot the router z-losses over the course of pre-training without router z-loss (blue) and with increasing values of c_z (we selected coefficient associated with green curve for all later experiments). With values of 1e-2, or larger, the z-loss shrinks near to zero. The left plot shows an encoder layer and the right plot shows a decoder layer.

---

[2] A potential source of confusion: $p_i(x)$ is the probability of routing token x to expert i. P_i is the probability fraction to expert i across *all tokens* in the batch $\mathcal{B}$.

C IMPROVED ARCHITECTURAL MODIFICATIONS

We consider a few small architecture variations here. The first modification was adding additional FFN layers (feed-forward network, see Table 1 for more details) immediately before or after each MoE layer (referred to as Sparse-Dense). Table 16 reveals the effectiveness of an FFN layer immediately preceding or following each sparse layer and that these extra FFN layers help less when added elsewhere in the network. Guaranteeing all tokens have at least one FFN applied to them between each attention layer appears useful.

Model	Neg. Log Perp. (↑)	Δ
Dense model (baseline)	-1.474	-
Dense model w/ extra FFN layers	-1.452	0.022
Sparse model (baseline)	-1.383	-
Sparse model w/ extra FFN layer *after* each sparse layer	**-1.369**	0.014
Sparse model w/ extra FFN layer *before* each sparse layer	**-1.369**	0.014
Sparse model w/ extra FNN layers placed randomly in the network	-1.376	0.007

Table 16: **A dense FFN immediately before or after each sparse layer improves quality**. Inserting an extra dense FFN immediately before or after each sparse layer improves quality 2x as much as placing the dense layers (randomly) elsewhere in the network. All of the non-baseline models have the same amount of FFN layers added for fair comparisons. Note that improving perplexity becomes harder as the model gets better.

Second, we introduce an additional bias in the expert layers. All our models use the GELU-Linear FFN (Shazeer, 2020), rather than the ReLU FFN:

$$\text{FFN}_{\text{ReLU}}(x) = (\text{ReLU}(xW_1))W_2$$
$$\text{FFN}_{\text{GEGLU}}(x) = (\text{GELU}(xW_{11}) \odot xW_{12})W_2$$

The additive bias is a learned weight (B) added after the first matrix multiplication in the FFN layer of shape $[batch, d_{ff}]$. The multiplicative bias (also referred to as a scale parameter) is a learned weight of the same shape, but does an elementwise multiplication. We initialize the additive bias to zeros and the multiplicative bias to ones.

$$\text{FFN}_{\text{GEGLU}} + \text{Add Bias}(x) = [(\text{GELU}(xW_{11}) \odot xW_{12}) + B]W_2$$
$$\text{FFN}_{\text{GEGLU}} + \text{Mult Bias}(x) = [(\text{GELU}(xW_{11}) \odot xW_{12}) \odot B]W_2$$

Table 17 shows the results of our different methods. Both the additive and multiplicative biases are essentially free: cheap to compute, adds few new parameters, and incurs no additional communication costs with model and expert parallelism. When using our router z-loss from Section 3.1, we observe no instabilities from the multiplicative bias. We do see that the multiplicative interactions improve performance, achieving a 4% speedup in convergence time over our strong sparse baseline. This hints that a promising avenue for future architectural research is finding new ways of adding more multiplicative interactions into networks.

Model	Neg. Log. Perp. (↑)	Δ
Dense Baseline	-1.474	-
Sparse Baseline	-1.369	-
Sparse + Additive Bias	-1.371	-0.002
Sparse + Multiplicative Bias	**-1.361**	0.008

Table 17: **More multiplicative interactions improve sparse model quality.** Both the additive and the multiplicative bias add virtually no parameters or compute.

Finally, motivated by the work of Roller et al. (2021), we explored similar methods, but did not find improvements in our setting. We tried routing using the word embedding exclusively, as well as

an additional input to the layer embedding for routing decisions. We toggled stopping the gradient through the word embedding or allowing it to have gradients propagated from the router. Using only the word embedding hurt quality, while using it in addition to the normal layer hidden activation was initially positive, but after pre-training for 50B+ tokens on models of scale 1B+ dense parameters it had a neutral effect. Appendix J has further details on the experiments with negative results.

D   BATCH PRIORITIZED ROUTING FOR LOWER CAPACITY FACTORS

Surprisingly, top-1 and top-2 routing work well with CF less than 1.0 despite token routing being done in a left to right order over the sequence. If N tokens are sent to an expert with only M spaces then $N > M$ tokens will dropped. The ordering of the dropping is important: we drop tokens going left to right (e.g. tokens earlier in the sentence will be routed first over the end tokens). This is done to avoid the model cheating. If we dropped tokens in another ordering, the model gets information on what tokens are occurring later in the sequence based on if tokens are being dropped or not.

Batch Prioritized Routing (BPR) from Riquelme et al. (2021) was introduced in Vision Transformers (Dosovitskiy et al., 2020) for image classification. Our work explores BPR with top-1 routing in the context of language modeling. BPR aims to have a global view of all tokens to determine which tokens should be dropped instead of the left-to-right ordering. The algorithm works by looking at all N tokens getting sent to Expert i and then only routing the M ones with the highest probabilities from the router. Table 18 shows that BPR top-1 routing improves performance over top-2 routing, especially when capacity factors are less than 1.0. We leave it to future work to try top-n BPR routing, which will hopefully yield larger improvments for higher capacity factors.

Importantly, BPR routing can only be done on the encoder side of the encoder-decoder model. On the encoder side there are not autoregressive predictions and all tokens can see each other. If you use BPR on the decoder, it learns to cheat by using future token information to improve current token predictions.

Algorithm	Train CF	Eval CF	Neg. Log. Perp. (↑)
Dense	—	—	-1.474
Dense-L	—	—	-1.384
BPR Top-1	0.5	0.5	-1.433
BPR Top-1	0.5	2.0	-1.416
Top-1	0.75	0.75	-1.428
Top-1	0.75	2.0	-1.404
Top-2	0.75	0.75	-1.424
Top-2	0.75	2.0	-1.402
BPR Top-1	0.75	0.75	-1.409
BPR Top-1	0.75	2.0	-1.397
Top-1	1.0	1.0	-1.397
Top-1	1.0	2.0	-1.384
Top-2	1.0	1.0	-1.392
Top-2	1.0	2.0	-1.378
BPR Top-1	1.0	1.0	-1.386
BPR Top-1	1.0	2.0	-1.379
Top-1	1.25	1.25	-1.378
Top-1	1.25	2.0	-1.373
Top-2	1.25	1.25	-1.375
Top-2	1.25	2.0	-1.369
BPR Top-1	1.25	1.25	-1.376
BPR Top-1	1.25	2.0	-1.375

Table 18: **Batch Prioritized Top-1 Routing (BPR) performance.** BPR top-1 routing improves quality when capacity factors are ≤ 1. However, once the capacity factor reaches 1.25, the improvements greatly diminish and it underperforms top-2 routing. Future work can try BPR with top-2 routing, which should hopefully further improve the performance.

E PRE-TRAINING DATASET DETAILS

The pre-training dataset used to train our Sparse 32B model is a mix of C4 (Raffel et al., 2019) and the dataset introduced in GLaM (Du et al., 2021).

Dataset	Tokens (B)	Weight in Mixture
Filtered C4	183	0.17
Filtered Webpages	143	0.34
Wikipedia	3	0.05
Conversations	174	0.23
Forums	247	0.02
Books	390	0.17
News	650	0.02

Table 19: **Data and mixture weights in the training set**. We sample from different dataset sources with probability proportional to "weight in mixture". The number of tokens listed are in billions (B). For more details on the C4 corpus see Raffel et al. (2019) and for the other datasets see Du et al. (2021).

F FULL FINE-TUNING SENSITIVITY DATA

Table 20 contains the raw data for Figure 6 measuring the fine-tuning protocol sensitivity. Dense and Sparse are encoder-decoder models FLOP matched to T5-Large that were pre-trained for 500k steps with a batch size of 1M tokens on the C4 corpus.

Model	Learning Rate	Batch Size	Reset Optimizer Slot Vars	SuperGLUE (↑)
Dense	1e-3	1M		84.8
Dense	1e-3	1M	✓	84.3
Dense	5e-4	1M		84.8
Dense	5e-4	1M	✓	84.2
Dense	1e-4	1M		84.0
Dense	1e-4	1M	✓	84.8
Dense	1e-3	262k		84.9
Dense	1e-3	262k	✓	83.7
Dense	5e-4	262k		84.9
Dense	5e-4	262k	✓	84.0
Dense	1e-4	262k		**85.1**
Dense	1e-4	262k	✓	85.0
Dense	1e-3	65k		83.7
Dense	1e-3	65k	✓	82.5
Dense	5e-4	65k		84.4
Dense	5e-4	65k	✓	84.1
Dense	1e-4	65k		84.9
Dense	1e-4	65k	✓	84.6
Sparse	1e-3	1M		**86.9**
Sparse	1e-3	1M	✓	85.9
Sparse	5e-4	1M		86.1
Sparse	5e-4	1M	✓	83.5
Sparse	1e-4	1M		84.3
Sparse	1e-4	1M	✓	84.3
Sparse	1e-3	262k		86.2
Sparse	1e-3	262k	✓	85.2
Sparse	5e-4	262k		85.5
Sparse	5e-4	262k	✓	84.8
Sparse	1e-4	262k		85.1
Sparse	1e-4	262k	✓	85.5
Sparse	1e-3	65k		85.8
Sparse	1e-3	65k	✓	85.5
Sparse	5e-4	65k		86.5
Sparse	5e-4	65k	✓	85.1
Sparse	1e-4	65k		85.6
Sparse	1e-4	65k	✓	84.5

Table 20: **Fine-tuning protocol sensitivity.** We vary the batch size, learning rate and whether we reset the optimizer slot variables for both dense and sparse models. Resetting the optimizer state during fine-tuning hurts performance. We observe a difference in optimal batch size and learning rate for sparse vs. dense models. Certain hyperparameter fine-tuning settings make the sparse and dense models perform almost exactly the same, showing the importance of correctly tuning the hyperparameters.

G OPTIMALLY SETTING THE ROUTING THRESHOLD

> **Top-n Routing Algorithm**
>
> 1. Route each token x to the expert with the highest router probability ($\text{gate}_1(x)$).
>
> 2. Normalize the top-n expert router scores for each token x, so $\text{gate}_i = \frac{\text{gate}_i(x)}{\sum_{i=1}^{n} \text{gate}_i(x)}$.
>
> 3. Route the token to the other n-1 experts (indexed by i) with probability $\min(1.0, \frac{\text{gate}_i(x)}{\text{threshold}})$. Threshold is a predefined hyperparameter that is typically set to 0.2.

We describe the MoE hyperparameters and how they should change as the routing algorithm changes. The MoE top-2 routing algorithm (Shazeer et al., 2017; 2018; Lepikhin et al., 2020) works as follows: first the router finds the expert that is assigned the higher router score (gate_1) and always sends the token to that expert. The token is also sent to its second highest expert with probability $\min(1.0, \text{gate}_2/\text{threshold})$. The threshold is a hyperparameter that is typically set to 0.2, and gate_2 is the token's router probability for the second highest expert. Note that gate_1 and gate_2 get normalized by the sum of their two scores, so they sum to one.

We trivially extend the top-2 algorithm to work for top-n routing here. Take the scores of the top-n experts per token and sum them, then renormalize each expert router score based on that sum. If the specific renormalized expert score has a higher value than the threshold (e.g. 0.2), then the token will be routed, otherwise it will be routed with probability $\frac{\text{score}}{\text{threshold}}$. At a high level this only routes the token to the next n-1 experts if their scores are not too much lower than the highest scored expert.

For top-3 routing vs top-2, the sum that the expert scores are normalized by is larger, therefore we experimented with *decreasing* the threshold. Our experimental results are shown in Table 21. Interestingly, we do observe the top-3 routing to slightly benefit from the lower threshold, while the opposite is true for top-2 routing.

We also experimented with an absolute threshold policy instead of a relative one. This is where the next n-1 tokens will be routed only if their router score is great than some pre-defined value (e.g. 0.2). We found it can achieve as good of performance if the threshold value is tuned.

Algorithm	Train CF	Threshold	Neg. Log. Perp. (↑)
Dense	—	—	-1.474
Dense-L	—	—	-1.384
Top-2	3.0	0.2	-1.354
Top-2	3.0	0.05	-1.356
Top-3	3.0	0.2	**-1.351**
Top-3	3.0	0.05	**-1.349**

Table 21: **Performance of top-2 and top-3 routing with different thresholds.** Top-3 routing does slightly better with lower thresholds than top-2 routing.

H MESH LAYOUT FOR DATA, MODEL AND EXPERT PARALLELISM WITH FEW EXPERTS

We use data and model parallelism partitioning with Mesh-Tensorflow (Shazeer et al., 2018). The partitioning strategy works by first forming a logical 2D mesh of size d x m, with the rows corresponding to the data dimension (d) and the columns as the model dimension (m) and the product equal to the total number of cores, $n = d$ x m. This mesh is only an abstraction. Each logical core must be mapped to a physical core, which is optimized through performance tuning.

As a refresher, each row in the mesh will have its own unique slice of the data and each column will have a unique slice of the model weights. The final gradient `allreduce` communication occurs across each individual column. The model parallelism `allreduce` communications occur across each row in the mesh. One constraint from this approach is that the number of rows must evenly
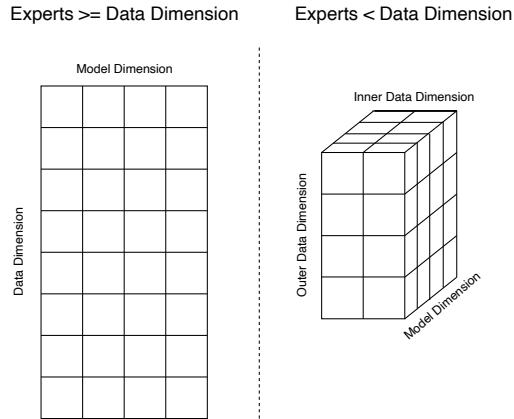
Figure 8: **Data and model parallelism meshes used for distributing models.** In this example there are a total of 32 processors (e.g. $n = 32$). **(Left)** A valid 2D mesh if the number of experts is greater than or equal to the data parallelism dimension. The data dimension has 8 rows (d) and the model dimension has 4 columns (m). **(Right)** A valid 3D mesh when we have fewer experts than the data parallelism dimension. The batch dimension is factorized into two new dimensions: inner data and outer data dimensions. Now we have 1 expert per inner data dimension (i). The 8 data rows in the left figure become 4 in the outer batch (o) and 2 in the inner batch (i) with 2 experts instead of 8.

divide the number of data sequences and the number of columns must evenly divide the model dimensions being partitioned.

But if we have *fewer* than d experts then this layout will not work. To allow for fewer experts than data parallelism rows in our mesh, we factorize the data dimension into two new dimensions: inner (i) and outer (o) where i x $o = d$ and the number of experts equals i. This transforms the logical 2D mesh of shape d x m into a 3D mesh of shape o x i x m. See Figure 8 for a visualization of both meshes [12].

I NOTE ON COMMUNICATION COSTS FOR DISTRIBUTED MODELS

Communication operations (`allreduce` and `all2all`) can significantly impact sparse model training throughput (see Table 1 for a description of the communication operations). `allreduce` calls are executed along model and batch dimensions, typically dominated by the model dimension `allreduce` calls that sum results of partial matrix multiplication operations from the workers. These calls are needed when matrix multiplications are partitioned across multiple cores (e.g. model parallelism). The gradient summation `allreduce` calls can be amortized away by training models with larger batch sizes since the gradient accumulation `allreduce` communication cost is independent of the batch size. To alleviate the memory issues of larger batch sizes, microbatches can be used. Microbatches do this by splitting the batch into n evenly divisible chunks and computing gradients on each sequentially, then summing.

To increase the `allreduce` throughput, more workers may need to be assigned to the model dimension (instead of batch dimension). However, increasing the number of workers may reduce compute per worker resulting in higher communication overheads that cancel some of the gains from higher communication throughput from `allreduce`. For the results in this paper, first we explored various model partitioning strategies. Next the shapes of the pre-training jobs were allocated based on performance benchmarking which showed the lowest cumulative communication overheads in `allreduce` and `all2all`.

---

[12]See Mesh Tensorflow for more details on the inner and outer batch: `https://github.com/tensorflow/mesh/blob/master/mesh_tensorflow/transformer/moe.py`

J Negative Results

We conclude with some ideas that yielded negative results in our setting.

Adding information if tokens were dropped to the router. We experimented with having the expert layer have information of whether the token was routed or dropped in the previous expert layers. We implemented this through counting the number of times a token was routed in all previous expert layers, having embeddings for each possible value and then adding this to the router embedding. We found that this made no difference in performance.

Adding explicit expert positional information. We experimented with adding explicit positional information into the outputs of the expert layer. We wanted to see if it either improved performance or sped up convergence during the beginning of training when expert layers were drastically changing. We did this through adding an embedding corresponding to what expert each token was sent (including an embedding if the token was dropped), but this did not improve performance.

Adding pre-training noise to fix pre-training and fine-tuning discrepancies. To help fix the pre-training perplexity and fine-tuning gap we tried pre-training the sparse models with a variety of different types of noise. The goal was to help pre-training match the fine-tuning conditions where dropout is used and more tokens can be dropped. Some of the noise types we tried adding during pre-training were dropout, dropping out full experts for a batch of tokens, and adding an entropy maximization auxiliary loss to the router. Unfortunately, all of the methods either hurt the pre-training quality too much or didn't end up helping the fine-tuning.

Load balancing in top-n routing over lower n-1 experts. In the standard top-n MoE formalization there is only loading balancing over the top expert a token is sent to. We experimented with adding an auxiliary load balancing term to the other $n-1$ experts in top-n routing, but found this to provide minimal benefits.

Mixing pre-training and fine-tuning data to prevent overfitting. To help combat the overfitting of sparse models during fine-tuning, we tried mixing in pre-training span corruption data at varying amounts (e.g. 1%, 5%, 25%, ...) during fine-tuning. This ended up not helping the fine-tuning performance, but did increase the training loss.