M6-T: Exploring Sparse Expert Models and Beyond

Anonymous ACL submission

Abstract

Sparse expert models can achieve promising results with outrageous large amount of parameters but constant computation cost, and thus it has become a trend in model scaling. Still, it is a mystery how Mixture-of-Experts (MoE) layers leveraging the parameters with sparse activation bring quality gains. In this work, we investigate several key factors in sparse expert models. We find that load imbalance may not be a significant problem affecting model quality, and auxiliary balancing loss can be removed without significant performance degrade. We further discover that larger number of sparsely activated experts k may not necessarily benefit the performance on the time basis, and we observe diminishing marginal utility that the performance gap gradually narrows with the increase in k We take a step forward to propose a simple method called expert prototyping that splits experts into different prototypes and applies top-k routing for each prototype in parallel. Our experiments demonstrate that the prototyping strategy improves the model quality, in comparison with further increasing to a larger k with comparable computation cost to prototyping. Furthermore, we conduct an exploration on training extremely large-scale models, and we figure out that the strategy shows greater effectiveness in training larger models. Notably, we push the model scale to over 1 trillion parameters on solely 480 NVIDIA V100-32GB GPUs. The proposed giant model M6-T with expert prototyping achieves substantial speedup in convergence over the same-size baseline.

1 Introduction

Large-scale pretraining has been demonstrating tremendous success across several fields, especially natural language processing (Devlin et al., 2019; Radford et al., 2019; Shoeybi et al., 2019; Raffel et al., 2020; Brown et al., 2020). Recent studies have shown that scaling model size can bring significant quality gains in downstream task performance (Shoeybi et al., 2019; Raffel et al., 2020; Brown et al., 2020), and the model quality scales as a power law with the data size, model scale, and amount of computation (Kaplan et al., 2020). This can be extended to the field of multimodal representation learning (Lu et al., 2019; Chen et al., 2020), where models with outrageous numbers of parameters (Ramesh et al., 2021; Lin et al., 2021) can achieve outstanding performance in cross-modal understanding and generation. However, training dense models is computationally expensive and it is hard to train them on super large-scale data with limited computational resources.

Inspired by the success of Mixture-of-Experts (MoE) (Shazeer et al., 2017; Ramachandran and Le, 2019), recent studies have focused on training large-scale sparse expert models with high training efficiency (Lepikhin et al., 2021; Fedus et al., 2021; Lin et al., 2021; He et al., 2021). An MoE layer consists of multiple experts and thus has a large model capacity. Each forward computation routes a token to k experts from N, where $k \ll N$. Such routing mechanism allows the combination of data parallelism and expert parallelism. Previous studies (Lepikhin et al., 2021; Fedus et al., 2021) show that it can achieve obvious performance speedup with the same computational costs. However, training such large-scale MoE models can be extremely difficult owing to multiple factors, e.g. system challenges of communication and load imbalance, and algorithmic challenges of training instabilities, etc.

In this work, we conduct an analysis of the recent MoE models to figure out which factors influence the model quality and training efficiency. We investigate several factors, including load balance, top-k routing strategies, etc. Our analysis demonstrates that load imbalance is not a significant problem affecting model quality, and the auxiliary balancing loss can be removed without significant performance drop. We further observe that the number of sparsely activated experts k in top-k routing make a difference in this context. Basically, increasing k from top-1 gating, namely switch gating (Fedus et al., 2021), contributes to better model performance. However, larger k for top-k routing will significantly degrades the training efficiency as well as convergence efficiency. Diminishing marginal utility can be discovered that there will be few performance gains when k is substantially large.

In this scenario, we propose a simple expert prototyping strategy called Expert Prototyping. Expert prototyping splits the experts into different groups and applies top-k routing for each prototype in a parallel schedule. The parallelism maintains substantially higher training efficiency in comparison with the FLOP-matched baselines, and the speed advantage turns out larger with the increase in k. We observe that the proposed models achieves obviously lower convergence on the time basis in comparison with the baseline models. Moreover, with the merits in training efficiency, the models with expert prototyping can still achieve comparable or even superior performance over the baselines with similar computation FLOPs, according to our experiments for upstream and downstream perplexity evaluation.

For further exploration, we extend the experiments to large-scale models with over 100 billion parameters respectively. Our findings and proposals are applicable to extremely large-scale models, and results show that expert prototyping has more significant advantages in training larger models. To go even further, we push the model scale to over 1 trillion parameters and successfully implement it on solely 480 NVIDIA V100-32GB GPUs. We show that the 1-trillion-parameter model M6-T with expert prototyping outperforms the baseline of the identical model scale, and achieves around 5 times of speedup in training convergence.

To summarize, our contributions are as follows:

- We explore key factors inside MoE models, and find that auxiliary balancing loss is not necessary, and the number of sparsely activated experts k can significantly impact the model performance.

- We propose a simple method called expert prototyping, which splits experts into k prototypes and applies top-k activation inside each prototype. The strategy maintains high training efficiency while keeping comparable or superior performance over the FLOP-matched baselines.

- We advance our models to 1 trillion parameters and successfully implement it on solely 480 NVIDIA V100-32GB GPUs. M6-T with expert prototyping outperforms the same-scale baseline and achieves substantial speedup in convergence.

In the following, we introduce background knowledge about sparse expert models in Section 2, and we demonstrate our exploration and proposal in Section 3 and further present our extreme-scale practice in Section 4. Finally, we present the related work in Section 5, and we draw the conclusion in Section 6.

2 Sparse Expert Models

Sparse expert models are regarded as a promising method for model scaling with high training efficiency. Training dense models requires extremely high computational costs, while sparse expert models can converge significantly faster as it iterates on more data with far higher efficiency on a time basis. Such mode of large-scale model training is also more environmentally friendly. In this setting, a large amount of the model weights are distributed to different workers owing to the architecture of MoE layers, and MoE allows the increase in parameters while keeping constant computational costs (Lepikhin et al., 2021; Fedus et al., 2021).

Mixture-of-Experts is essentially a routing algorithm that routes tokens to k specified experts from N (where $k \ll N$) for forward computation. It enables expert parallelism so that experts process input tokens specified by gating functions simultaneously. Expert networks, each of which is a multi-layer perceptron, are distributed across workers. We use a gating function which specifies k from N experts for an input token representation x. The chosen experts process the representation with forward computation and reduce their results with weighted summation based on the gating values:

$$\begin{aligned}\tilde{x} &= \sum_{i}^{k} p_i E_i(x) \\ p &= softmax(g) \\ g &= topk(norm(W_g x))\end{aligned} \tag{1}$$

where E_i refers to the i-th expert. The most typical MoE algorithms for large-scale sparse expert models are Switch Transformer and GShard (Fedus et al., 2021; Lepikhin et al., 2021). Their key

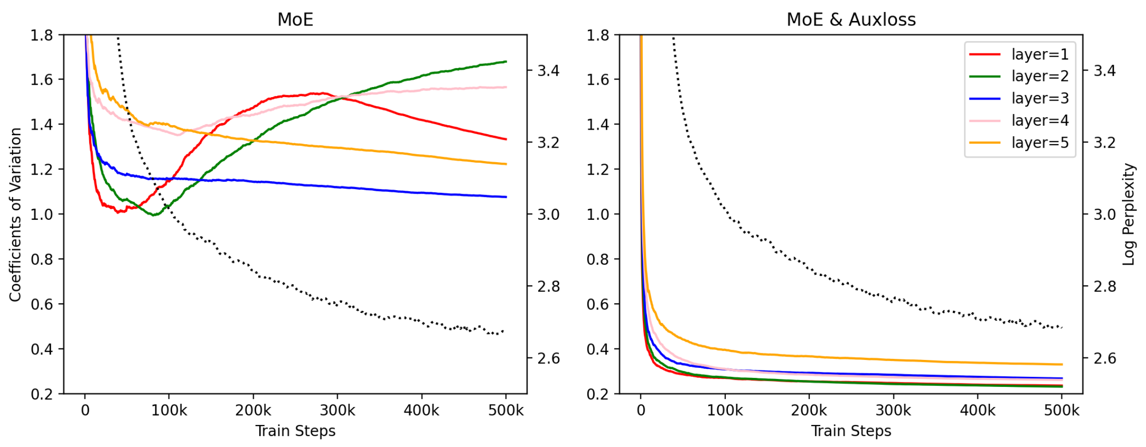


Figure 1: **Curves of the developments of coefficients of variation c_v at different layers.** Here we demonstrate the development of c_v of baseline and auxiliary loss at all layers. We also demonstrate their curves of training log perplexity (see the black dotted curve). Auxiliary loss helps the model gain highly balanced compute loads at every layer, but the higher balance has not been translated to higher model quality. On the contrary, the behaviors of load balance in the vanilla MoE model are peculiar. Though c_v at all layers drop at the beginning, yet some of them even increase to a high value afterward.

difference lies in their choice of top-k selection, which applies top-1 and top-2 selection respectively. Switch Transformer (Fedus et al., 2021) noted that routing a token to 1 expert only is effective in preserving model quality and reducing computation complexity, contrary to the idea of Shazeer et al. (2017) that k should be larger than 1 so that there are non-trivial gradients to the routing functions.

What makes a difference in the performance and model quality is the actual implementation of distributed experts. Model parallelism allows the partitioning of a large tensor across workers, and our implementation even enables multiple experts on an identical worker, instead of one expert per worker. Due to the dynamic nature of top-k routing, it may cause low efficiency if severe load imbalance happens. A standard implementation to tackle the problem is the setting of expert capacity (Lepikhin et al., 2021; Fedus et al., 2021; Shazeer et al., 2018), which is defined as:

$$C = \frac{k \cdot T}{N} \cdot \gamma, \tag{2}$$

where T refers to the number of tokens in a batch and γ refers to the capacity factor which is commonly larger than 1.0. A larger capacity factor can provide more buffer for expert capacity. Tokens are distributed to experts with all-to-all dispatching operations. The token representations processed by selected experts are combined with all-to-all communication to their original workers. On the condition of exceeded capacity, token representations skip forward computation by residual connection. For experts still having space in their capacities after dispatching, we add padding tokens to fill in. Thus the computations and communications costs are positively related to the number of experts N and the expert capacity C. Low expert capacity can cause a significant amount of dropped tokens if there is load imbalance on experts, but increasing expert capacity will correspondingly enhance the computation and communication costs.

3 Exploration of Sparse Expert Models

To investigate the MoE models, we conduct experiments to observe the effects of several factors, including auxiliary balancing loss for load balance, the value of k for sparse activation, etc. Following Lin et al. (2021), we pretrain the models on the M6-corpus and we observe their performance on a held-out training set for upstream evaluation and an image captioning dataset, E-commerce IC in MUGE benchmark[1], for downstream evaluation. We provide more experimental details in Appendix A.

3.1 Development of Load Balance

Recent studies pointed out the significance of balanced routing (Fedus et al., 2021; Lepikhin et al., 2021; Lewis et al., 2021), and illustrated the importance of balancing methods such as auxiliary expert balancing loss. We first conduct experiments on

[1]https://tianchi.aliyun.com/muge

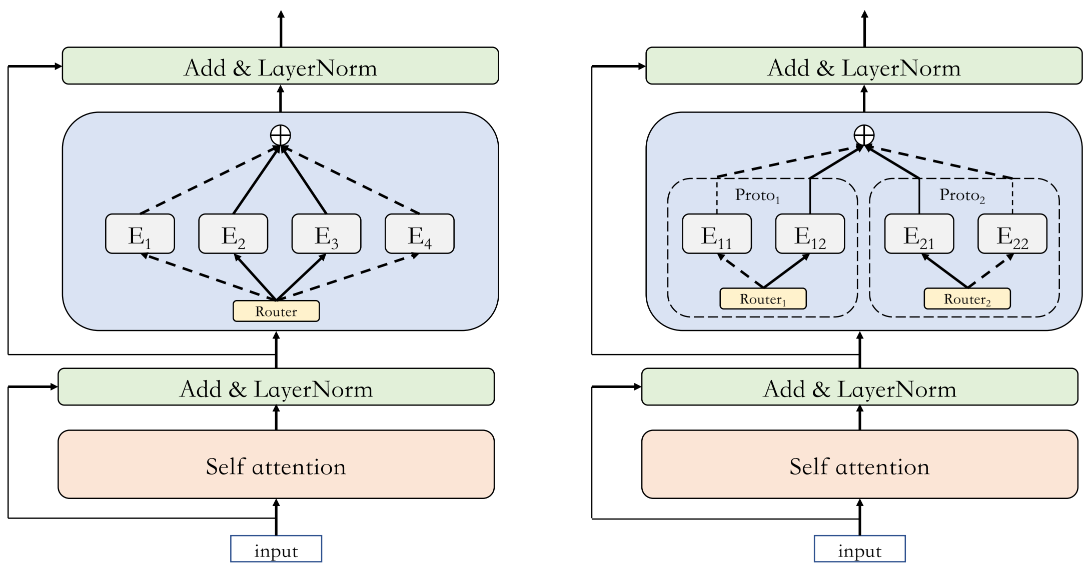


Figure 2: **A demonstration of conventional top-2 routing and expert prototyping with 2 top-1 routing.** In top-2 routing, the router selects the top-2 from all the experts and sends the token representation to those experts. In 2 top-1 routing for expert prototyping, experts are first grouped into 2 prototypes, and there is a router for each prototype. Each router chooses the top-1 expert and sends the token representation through it. The output from each prototype is summed up element-wisely for the final output.

the MoE models with and without auxiliary load balancing loss respectively. We pretrain both models for $500k$ steps and compare their performance in the Perplexity (PPL) evaluation. Experimental results show that both models achieve similar performance, and the one with auxiliary loss even performs slightly worse in the evaluation of training log perplexity (2.694 vs. 2.645). These observations intrigue us to further investigate the relationship between load balance and model quality.

We evaluate the degree of compute load balance of experts at every layer. Following Shazeer et al. (2017), we use the coefficient of variation for the evaluation of load balance. We define the coefficient of variation for effective compute loads as that of the number of real tokens computed by the experts $c_v = \frac{\sigma(\mathcal{T})}{\mu(\mathcal{T})}$, where $\mathcal{T}$ refers to tokens computed by experts. This metric reflects the degree of uniformity of token assignment. We observe the development of c_v in the training process to evaluate the change of load assignment.

We demonstrate the results in Figure 1. For all layers, significant load imbalance exists at the initial stage according to the high values of c_v. Notably, the value is generally higher at the top layers. For the model trained with auxiliary expert load balancing loss, c_v at all layers drop drastically at the initial stage to a low value around 0.3. This denotes that highly balanced compute loads,[2] and they become stable in the following. However, compute loads are quite different for the MoE model without auxiliary loss. Though c_v at all layers drop at the beginning, yet they fail to reach a small value, which denotes highly balancing compute loads. Except for that, some even increase to a high value afterward. These phenomena reflect the existence of load imbalance. Though auxiliary loss is advantageous in expert load balancing, such an advantage has not been translated to performance, as mentioned above.

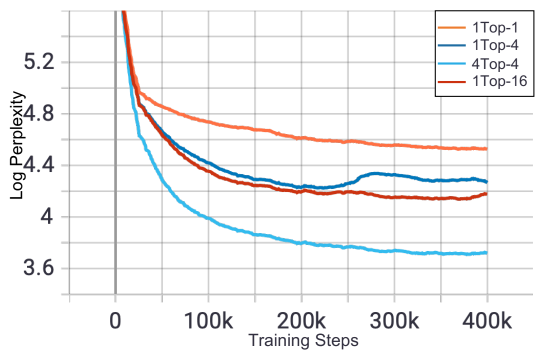
3.2 The Effects of Top-k Sparse Activation

Table 1: **Speed of models with different top-k routing strategies (ms/step).** We report the training speed of models with different routing strategies. The comparison between different strategies are conducted under various model sizes.

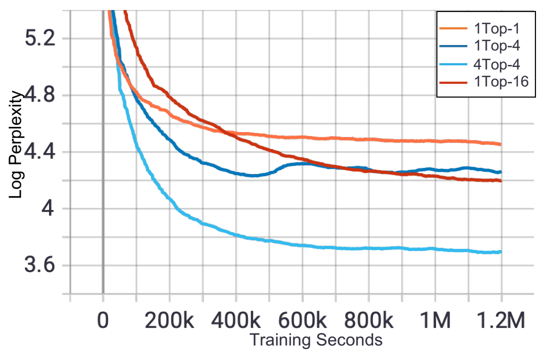
Model Size	Top-1	Top-4	Top-16	4 Top-4
3-layer	98	199	600	203
16-layer	436	975	3100	980

Previous work (Fedus et al., 2021) pointed out that top-1 is sufficient for high model quality in

[2]We manually observe the number of tokens that experts receive and find that compute loads are highly balanced.

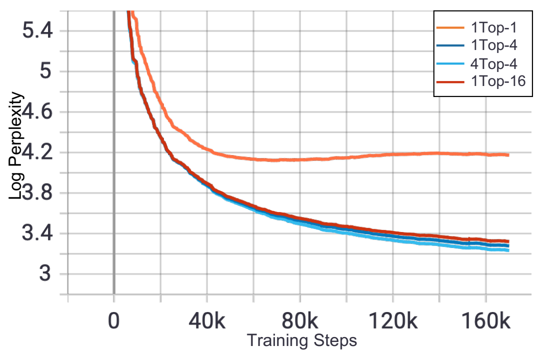


(a) step-basis curve

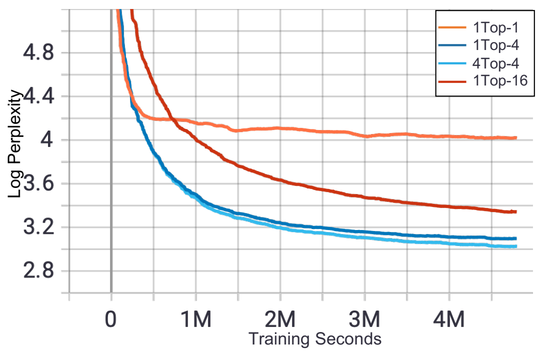


(b) time-basis curve

Figure 3: **Model performance with different routing strategies under the 3-layer setup.** (a) demonstrates the performance on the step basis and (b) demonstrates that on the time basis. From both figures, increasing k from 1 to 4 significantly improves convergence. However, further changing the routing strategy to 1 top-16 brings marginal benefit. In comparison, the model with expert prototyping, namely 4 top-4, achieves a substantially lower perplexity.[3]



(a) step-basis curve



(b) time-basis curve

Figure 4: **Model performance with different routing strategies under the 16-layer setup.** (a) demonstrates the performance on the step basis and (b) demonstrates that on the time basis. Similar to the results of 3-layer models, the 4 top-4 model achieves the best performance and it gains a significant advantage on the time-basis evaluation over the 1 top-16 baseline.

comparison with top-k selection with $k > 1$, while it has significant advantages in computational efficiency, but the experimental results in Lewis et al. (2021) show that top-2 still outperforms the top-1 selection. Here we conduct experiments to further investigate how different top-k methods influence the model quality. We first examine whether larger k can help the model achieve better performance. Specifically, we choose $k \in \{1, 4, 16\}$ for the evaluation, and we implement them on models of different scales, where $l \in \{3, 16\}$. More experimental details are illustrated in Appendix A.2. We demonstrate their performance of upstream log perplexity from different views, including the step-basis[4] and time-basis performance.

Figure 3(a) and Figure 4(a) show that on the step basis models with larger values of k can reach more superior performance. We also observe that the performance gap between the top-4 routing and top-16 routing is small especially when the model is large with 16 layers. This phenomenon demonstrates diminishing returns with the increase in k, and the problem becomes more salient when the scale of model is large.

However, the computation costs for quality is actually quite large, as training efficiency drops drastically with the increase in k. Table 1 reports the training speed of models of different scales with different routing strategies. It illustrates that larger k will greatly decrease the training speed,

[4]We use the same batch size so that the step basis is essentially equal to sample basis.

Table 2: **Evaluation of perplexity (lower is better) on upstream pretraining and downstream E-commerce IC task** (Lin et al., 2021). Following Radford et al. (2019), we do not perform any fine-tuning for any of these results. Within each model size, the strategies taken here for comparison are pretrained on almost same budget of training time and GPU devices.

Model	Training Time	Steps	Training Samples	Upstream PPL	Downstream PPL
			3-layer model		
1 top-4	2.33 days	1010K	64.6M	64.07	54.74
1 top-16	2.36 days	340K	21.8M	59.74	51.66
4 Top-4	2.33 days	1010K	64.6M	**37.71**	**33.48**
			16-layer model		
1 top-4	6.21 days	550K	352M	23.10	25.07
1 top-16	6.28 days	175K	112M	27.66	28.78
4 top-4	6.24 days	550K	352M	**20.91**	**24.21**

e.g., for the 16-layer model, top-16 routing is almost 7 times slower than top-1 routing. The complexity of top-k gating comes from the fact that token-to-expert multiple assignments are hard to parallel. Therefore, we hope to figure out a solution to alleviate this problem of serial computing.

3.3 Expert Prototyping

Previous analysis indicates that k experts play different roles and outcompete a single expert. However, the looping "argmax" operation in top-k routing incurs computation inefficiency. To tackle the issue of inefficiency, we propose a simple method called expert prototyping. Expert randomly splits experts into k prototypes. In each forward computation, each token is first sent to the k prototypes. Inside each prototype, it is processed by the expert selected by top-k routing. Parallel routing is performed across prototypes. Then the outputs of prototypes are combined linearly as shown below:

$$y = \sum_{i=1}^{k} \sum_{j=1}^{m} p_{ij} E_{ij}(x), \tag{3}$$

where m refers to the number of experts inside a group. This method avoids the looping argmax operation. Instead, it generates k outputs in a parallel fashion and it does not incur training inefficiency. From Table 1, it can be found that the 4 top-4 has similar performance in training efficiency compared with the 1 top-4 (980 vs. 975 ms/step). Expert prototyping makes the gating independent within each group, which is easy to parallel on modern devices. Under the parallel optimization and the same number of experts, the complexity of conventional top-k routing is $O(K)$, while the complexity of prototyping top-k is $O(K/P)$, where P is the number of prototyping groups. Consequently, expert prototyping reduces the gating time linearly while keeping the same number of experts.

We demonstrate the performance of expert prototyping model on the step and time basis in Figure 3 and Figure 4. From Figure 3, we find that for smaller models the 4 top-4 routing model can greatly outperform all the baselines with a large gap in log perplexity, on both step and time basis. From Figure 4, for larger models with around 10 billion parameters, we find that 4 top-4 still achieve the best performance on the step basis according to Figure 4(a). However, on the time basis according to Figure 4(b), we observe that 4 top-4 with an obvious speed advantage can significantly surpass the 1 top-16 baseline. This reflects that our proposed expert prototyping can help the model maintain high training efficiency but bring quality gains.

3.4 Evaluation

We demonstrate both the upstream and downstream PPL of the examined models in Table 2. We also provide the training time, step, and samples for better comparison. It can be found that under the same computation budget, the proposed expert prototyping model with 4 top-4 routing can outperform both the 1 top-4 and 1 top-16 baselines in both upstream and downstream evaluation with obvious advantages, no matter when the model scale is small (3 layers) or large (16 layers). Furthermore, the 4 top-4 has similar computation efficiency in comparison with the 1 top-4. Therefore, it can be trained with more samples than the 1 top-16 and it naturally achieves superior performance.

4 Rocketing to Trillion Parameters

In this section, we demonstrate that our findings and proposals are also applicable to large-scale pretrained models, and we finally advance the model

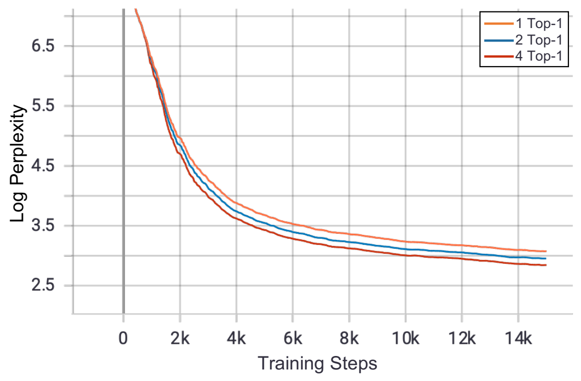


Figure 5: **Performance of 100B models with different routing strategies.** In both cases, expert prototyping performs better than the MoE baseline, and larger values of k further benefit model qualities.

scale to 1 trillion parameters.

We validate our findings on extremely large-scale models. We first scale the model size up to 100 billion parameters respectively. For simplicity, we validate expert prototyping on the 100B-parameter models. We report the training log perplexity for the model performance. As Figure 5 demonstrates, expert prototyping still have advantages over the MoE baseline, and similarly in both contexts larger k for expert prototyping can further benefit the model quality. This shows the effectiveness of expert prototyping in training models of a much larger scale.

Based on the aforementioned findings and proposals, we move forward to build an extremely large-scale model with over 1 trillion parameters. Due to limited computational resources, we attempt to figure out solutions to implement a 1-trillion-parameter model on solely 480 NVIDIA V100-32GB GPUs.

To be more specific, we implement our model on a cluster of workers connected by RDMA networks with a bandwidth of 100Gb. To save memory usage, we instead turn to Adafactor (Shazeer and Stern, 2018) for optimization in concern of its sublinear memory costs. However, there are a series of sporadic issues concerning training instabilities. Through trials and errors, we find that such model training is highly sensitive to learning rates, especially when being trained with Adafactor. We did not use the default one 0.01 due to divergence, but instead, we use 0.005 to strike a balance between training stability and convergence speed. Also, we find that it is essential to lower the absolute values of initialized weights, which is also illustrated in Fedus et al. (2021). We specifically reduce the BERT initialization, a truncated normal distribution with $\mu = 0$ and $\sigma = 0.02$, by a factor of 10.

We first evaluate the quality of models with different parameters but similar computation FLOPs by observing training log perplexity. We compare the performance of MoE baseline models with 100 billion, 250 billion parameters, and 1 trillion parameters, and we observe that the results prove the scaling law that models with larger capacity performs better, as demonstrated in Figure 6. Then we implement both 1-trillion-parameter MoE baseline and our expert prototyping MoE model.[5] Still, from Figure 6 we can figure out the proposal has a strong advantage over the compared model with 1 trillion parameters. We observe a substantial speedup in convergence, where our method is around 5 times faster than the baseline. However, both models have similar computational FLOPs, which demonstrates that our method strikes a far better balance between computational efficiency and model quality.

5 Related work

Pretraining has achieved great success in these years, and it has recently become a common practice in natural language processing (Peters et al., 2018; Devlin et al., 2019; Radford et al., 2018; Yang et al., 2019; Liu et al., 2019; Dong et al., 2019). In the field of cross-representation learning, pretraining has also become significant and pushed the limit of model performance in downstream tasks (Lu et al., 2019; Su et al., 2020; Lu et al., 2020; Chen et al., 2020; Gan et al., 2020; Li et al., 2020; Yu et al., 2021; Li et al., 2021; Zhang et al., 2021). Recent studies (Kaplan et al., 2020) demonstrate the power law of model scale and performance. With the rapid development in distributed training and parallelism (Shoeybi et al., 2019; Rajbhandari et al., 2020; Ren et al., 2021; Rajbhandari et al., 2021), we have witnessed the burst of studies in extremely large scale pretraining in both natural language processing (Brown et al., 2020; Shoeybi et al., 2019) and multimodal pretraining (Ramesh et al., 2021; Lin et al., 2021) and also new state-of-the-art performance in the recent two years. Though extremely large-scale dense models are highly effective especially in the

[5]Due to limited computational resources and instabilities in systems and hardware, the trillion-parameter expert prototyping model has been trained for only $30k$ steps.

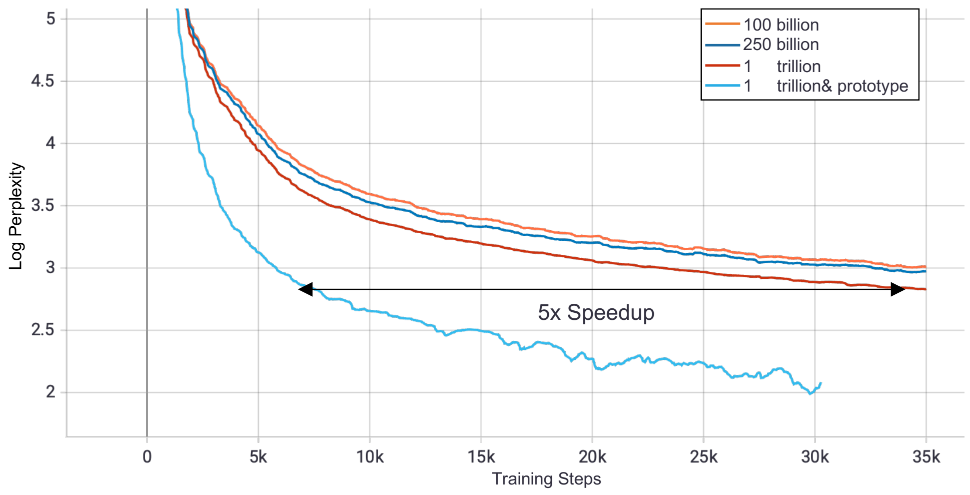


Figure 6: **Performance of baseline models with** 100 **billion,** 250 **billion, and** 1 **trillion parameters, as well as** 1**-trillion-parameter model with expert prototyping.** The curves reflect the scaling law, and also demonstrate the advantage of expert prototyping for giant models.

context of few-shot learning (Brown et al., 2020), some researchers have turned to sparse expert models for efficient large-scale pretraining. Inspired by the success of Mixture-of-Experts (Shazeer et al., 2017; Ramachandran and Le, 2019; Shazeer et al., 2018), recent studies (Lepikhin et al., 2021; Fedus et al., 2021) expand the model size to over trillion parameters and fully utilize the advantages of TPUs to build sparse expert models with Mesh-Tensorflow (Shazeer et al., 2018). They demonstrate that sparse expert models can perform much better than dense models with the same computational FLOPs but their computational costs are similar. A series of the following work successfully implement sparse expert models on NVIDIA GPU (Lin et al., 2021; Lewis et al., 2021). In this work, we follow the practice of Lin et al. (2021) and implement our models on the distributed learning framework Whale (Wang et al., 2020).

6 Conclusion

In this work, we explore the factors inside sparse expert models and investigate how they influence the model quality and computational efficiency. We find out that load imbalance may not be a significant issue affecting model quality, and the auxiliary balancing loss can be removed without significant performance drop. We observe that the number of activated experts k play a significant role in training MoE models, where larger k can help the model achieve better performance. However, the increase will enhance computation complexity and incur training inefficiency. Therefore, we propose a simple solution called expert prototyping. The method splits experts into different prototypes and applies top-k routing. With extensive experiments, we show that expert prototyping can help maintain high training efficiency but significantly improve the model performance in both upstream and downstream evaluation. Furthermore, to evaluate its effects in large-scale training, we extend the experiments to large-scale models with over 100 billion parameters and demonstrate the effectiveness. Finally, we push the scale to 1 trillion parameters and successfully implement the 1-trillion parameter model M6-T on solely 480 NVIDIA V100-32GB GPUs. We show that our simple method can effectively improve the performance of M6-T over the same-scale baseline, and M6-T gains a 5-time speedup in convergence.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. Tensorflow: A system for large-scale machine learning. In OSDI 2016, pages 265–283.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In NeurIPS 2020.

Yen-Chun Chen, Linjie Li, Licheng Yu, Ahmed El Kholy, Faisal Ahmed, Zhe Gan, Yu Cheng, and Jingjing Liu. 2020. UNITER: universal image-text representation learning. In ECCV 2020, pages 104–120.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In NAACL-HLT 2019, pages 4171–4186.

Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. 2019. Unified language model pre-training for natural language understanding and generation. In NeurIPS 2019, pages 13042–13054.

William Fedus, Barret Zoph, and Noam Shazeer. 2021. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. CoRR, abs/2101.03961.

Zhe Gan, Yen-Chun Chen, Linjie Li, Chen Zhu, Yu Cheng, and Jingjing Liu. 2020. Large-scale adversarial training for vision-and-language representation learning. In NeurIPS 2020.

Jiaao He, Jiezhong Qiu, Aohan Zeng, Zhilin Yang, Jidong Zhai, and Jie Tang. 2021. Fastmoe: A fast mixture-of-expert training system. CoRR, abs/2103.13262.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In CVPR 2016, pages 770–778.

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

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2021. Gshard: Scaling giant models with conditional computation and automatic sharding. In ICLR 2021.

Mike Lewis, Shruti Bhosale, Tim Dettmers, Naman Goyal, and Luke Zettlemoyer. 2021. BASE layers: Simplifying training of large, sparse models. CoRR, abs/2103.16716.

Wei Li, Can Gao, Guocheng Niu, Xinyan Xiao, Hao Liu, Jiachen Liu, Hua Wu, and Haifeng Wang. 2021. UNIMO: towards unified-modal understanding and generation via cross-modal contrastive learning. In ACL/IJCNLP 2021, pages 2592–2607.

Xiujun Li, Xi Yin, Chunyuan Li, Pengchuan Zhang, Xiaowei Hu, Lei Zhang, Lijuan Wang, Houdong Hu, Li Dong, Furu Wei, Yejin Choi, and Jianfeng Gao. 2020. Oscar: Object-semantics aligned pre-training for vision-language tasks. In ECCV 2020, pages 121–137.

Junyang Lin, Rui Men, An Yang, Chang Zhou, Ming Ding, Yichang Zhang, Peng Wang, Ang Wang, Le Jiang, Xianyan Jia, Jie Zhang, Jianwei Zhang, Xu Zou, Zhikang Li, Xiaodong Deng, Jie Liu, Jinbao Xue, Huiling Zhou, Jianxin Ma, Jin Yu, Yong Li, Wei Lin, Jingren Zhou, Jie Tang, and Hongxia Yang. 2021. M6: A chinese multimodal pretrainer. CoRR, abs/2103.00823.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized BERT pretraining approach. CoRR, abs/1907.11692.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In ICLR 2019.

Jiasen Lu, Dhruv Batra, Devi Parikh, and Stefan Lee. 2019. Vilbert: Pretraining task-agnostic visiolinguistic representations for vision-and-language tasks. In NeurIPS 2019, pages 13–23.

Jiasen Lu, Vedanuj Goswami, Marcus Rohrbach, Devi Parikh, and Stefan Lee. 2020. 12-in-1: Multi-task vision and language representation learning. In CVPR 2020, pages 10434–10443.

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In NAACL-HLT 2018, pages 2227–2237.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. OpenAI blog, 1(8):9.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. J. Mach. Learn. Res., 21:140:1–140:67.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: memory optimizations toward training trillion parameter models. In SC 2020, page 20.

Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. Zero-infinity: breaking the GPU memory wall for extreme scale deep learning. In SC '21, pages 59:1–59:14.

Prajit Ramachandran and Quoc V. Le. 2019. Diversity and depth in per-example routing models. In ICLR 2019.

Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. 2021. Zero-shot text-to-image generation. In ICML 2021, pages 8821–8831.

Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. Zero-offload: Democratizing billion-scale model training. In ATC 2021, pages 551–564.

Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake A. Hechtman. 2018. Mesh-tensorflow: Deep learning for supercomputers. In NeurIPS 2018, pages 10435–10444.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In ICLR 2017.

Noam Shazeer and Mitchell Stern. 2018. Adafactor: Adaptive learning rates with sublinear memory cost. In ICML 2018, pages 4603–4611.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. CoRR, abs/1909.08053.

Weijie Su, Xizhou Zhu, Yue Cao, Bin Li, Lewei Lu, Furu Wei, and Jifeng Dai. 2020. VL-BERT: pre-training of generic visual-linguistic representations. In ICLR 2020.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In NeurIPS 2017, pages 5998–6008.

Ang Wang, Xianyan Jia, Le Jiang, Jie Zhang, Yong Li, and Wei Lin. 2020. Whale: A unified distributed training framework. CoRR, abs/2011.09208.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. In NeurIPS 2019, pages 5754–5764.

Fei Yu, Jiji Tang, Weichong Yin, Yu Sun, Hao Tian, Hua Wu, and Haifeng Wang. 2021. Ernie-vil: Knowledge enhanced vision-language representations through scene graphs. In AAAI 2021, pages 3208–3216.

Pengchuan Zhang, Xiujun Li, Xiaowei Hu, Jianwei Yang, Lei Zhang, Lijuan Wang, Yejin Choi, and Jianfeng Gao. 2021. Vinvl: Revisiting visual representations in vision-language models. In CVPR 2021, pages 5579–5588.

A Appendix

A.1 Multimodal Pretraining and Downstream Evaluation

In practice, we follow Lin et al. (2021) that employs an extremely large-scale multimodal pretrained model with MoE architecture in Chinese. Specifically, we pretrain a model on the image-text pairs from the dataset M6-Corpus (Lin et al., 2021). In multimodal pretraining, the pretrained model receives the inputs of a pair of related image and text as the input and generates the high-level representations with layers of Transformer (Vaswani et al., 2017). In our experiments, we first transform an input image to patch features by splitting it into 4×4 patches and extracting patch features with a trained ResNet (He et al., 2016). We flatten the patch features of the input image to a sequence of representations and concatenate them with the word embeddings of the text sequence shorter than 128 words. Then we build a feature extractor with multiple layers of transformer consisting of self attention and feed-forward neural networks (FFN). Notably, in order to integrate MoE to the model architecture, we replace the FFN with MoE, where FFN as experts are distributed across workers. We pretrain the model with the task of image captioning, where the model learns to generate words autoregressively based on the previous context including the patch features.

To comprehensively evaluate the performance of the methods, we conduct experiments on image captioning in Chinese, and we follow Lin et al. (2021) to use the E-commerce IC dataset in MUGE benchmark[6]. We focus on the capability of language modeling of the pretrained model, and thus we use teacher forcing and evaluate the performance by perplexity (PPL).

A.2 Experimental Setups

For the exploration, we investigate different setups for both models. Here we point out key configurations of our experimental setups and we demonstrate the details in Table 3. Following BERT-Chinese (Devlin et al., 2019), we use the same vocabulary with 21128 subwords. For the initialization, we use the BERT initialization with $\mu = 0$ and $\sigma = 0.02$ for most cases, and we use an initialization with a smaller standard deviation of 0.002 for the $1T$ model. As to the expert capacity, we

[6] https://tianchi.aliyun.com/muge

Table 3: **Hyperparameters for pretraining the MoE models.**

Hparam	**3-layer**	**16-layer**	**100B**	**1T**
Hidden size	128	128	1024	1024
Intermediate size	512	512	4096	21248
Number of layers	3	16	24	24
Number of attention heads	8	8	16	16
Attention head size	16	16	64	64
Initializer range	0.02	0.02	0.02	0.002
Number of experts	4096	4096	512	960
Number of GPUs	8	8	128	480
Optimizer	AdamW	AdamW	AdamW	Adafactor
Learning rate	1e-4	1e-4	8e-5	5e-3
Mixed precision	✓	✓	✓	×
FP16 communication	✓	✓	✓	×
Params	1.6B	8.6B	103.2B	1002.7B

Table 4: **Notation table for Pseudo code.**

Variable	**Definition**
D	Number of workers
d	Number of GPUs per worker (d=1 in this paper)
E	Number of total experts
e	Number of experts per worker (e*D=E)
C	Capacity per expert
M	Model size (same as hidden size, same as embedding size)
I	Intermediate size
B	Batch Size per GPU
L	Sequence Length
T	Number of tokens (T=B*L)
Z	Number of prototypes
F	Number of expert per prototype (Z*F=E)

generally use a capacity factor of $\gamma = 1.25$ for more buffer. The batch size per GPU is 8 and the total batch size is equal to the product of the batch size per GPU and the number of GPUs. We use AdamW optimizer (Loshchilov and Hutter, 2019) for optimization except for the $1T$ model where we use Adafactor (Shazeer and Stern, 2018) instead. For Adafactor, we set the learning rate to $5e-3$. We use warmup schedule with a warmup step of 500. The dropout rate for FFN and attention is 0.1. We use mixed precision training for FP16 communication for all models except the $1T$ one due to the issue of training instability.

We implement our experiments on Tensorflow 1.15 (Abadi et al., 2016). Different from the original implementation of Switch and GShard with Mesh-Tensorflow (Shazeer et al., 2018), we implement the multimodal pretrained model with the framework Whale (Wang et al., 2020), which enables data, model, and expert parallelism on NVIDIA GPU.

A.3 Pseudo Code for Expert Prototyping

The pseudo codes for MoE layer and proposed expert prototyping in Whale are provided in Figure 7 and Figure 8 respectively. Table 4 illustrates the notations of specific tensor dimensions.

Amount of All-to-All Communication There are two operations of all-to-all communication in each MoE FFN layer in a forward propagation process (one for *dispatch_inputs* and the other for *outputs* in the pseudo code). During the communication, each entry of the communicated tensor passes to a worker once. Thus, the total amount of communication, which is $O(EdCM) + O(eDCM) =$

$O(EdCM) = O(ECM)$, depends on the number of experts, capacity and model size.

Amount of Computation For the 1T-scale MoE model, the total amount of computation in the MoE FFN layer is mainly dominated by the two matrix multiplications, which transform the input tensor from the hidden size to the intermediate size and then vice versa. The total computation of these two matrix multiplications is $O(DeCMI) + O(DeCIM) = O(ECMI)$. For 1T model, these two operations hold around 98% total forward FLOPs of the MoE FFN layer.

```
import whale as wh
import tensorflow as tf


def MoE_feed_forward(inputs, prototype_num, num_experts, expert_capacity):
    """ MoE Layer FeedForward.  """
    # inputs (BLM): Each example is typically a vector of size model_dim,
    # representing embedded token or an element of Transformer layer output
    orig_batch_dim, orig_seq_length, model_dim = inputs.size()
    total_token_num = orig_batch_dim * orig_seq_length
    # Flatten input tokens.
    reshaped_inputs = tf.reshape(inputs, [1, total_token_num, model_dim])  # dTM
    # Moe Gating.
    # combine_tensor (dTEC): used for combining expert outputs and scaling with probabilities.
    # dispatch_mask (dTZFC): used for dispatching input tokens to the correct expert.
    combine_tensor, dispatch_mask, aux_loss = prototype_gating(reshaped_inputs, prototype_num,
                                                               num_experts, expert_capacity)

    # Expand inputs for different prototypes.
    reshaped_inputs = tf.broadcast_to(tf.expand_dims(reshaped_inputs, axis=2),
                                      [1, total_token_num, prototype_num, model_dim])  # dTZM

    # Prepare to dispatch tokens to the correct expert.
    dispatch_inputs = tf.einsum("dTZFC,dTZM->ZFdCM", dispatch_mask, reshaped_inputs,
                                name="dispatch_inputs")

    dispatch_inputs = tf.reshape(
        dispatch_inputs,
        [num_experts, 1, -1, model_dim])  # ZFdCM -> EdCM

    # Standard forward.
    # Whale is able to infer an efficient parallel strategy automatically within the split scope.
    # It will insert an appropriate all-to-all communication operator in the necessary position.
    with wh.split():
        # All-to-All communication.
        # Inputs are split across the experts dimension (1st) and dispatched to correct experts.
        # Workers gather tokens sent by other workers along the workers dimension (2nd).
        # dispatch_inputs: EdCM -> eDCM
        # inter experts forward.
        # inter_expert_weights (eMI): Each expert has its own unique set of weights.

        intermediate = tf.einsum('eDCM,eMI->eDCI', dispatch_inputs, inter_expert_weights,
                                 name="dispatched_inter_outputs")
        activated_inters = activation_fn(intermediate)  # eDCI

        # Output experts forward.
        # out_expert_weights (eIM): Each expert has its own unique set of weights.
        outputs = tf.einsum(
            'eDCI,eIM->eDCM', activated_inters, out_expert_weights, name="dispatched_outputs")

        # All-to-All communication.
        # Outputs are split across the workers dimension (2nd) and switched back to experts.
        # Workers gather outputs sent by other workers along the experts dimension (1st).
        # outputs: eDCM -> EdCM
        # Multiply outputs of experts by the routing probability.
        combined_outputs = tf.einsum(
            'dTEC,EdCM->dTM', combine_tensor, outputs, name="combined_outputs")

    # Convert the outputs back to input shape.
    outputs = tf.reshape(combined_outputs,
                         [orig_batch_dim, orig_seq_length, model_dim])  # dTM -> BLM
    return outputs, aux_loss
```

Figure 7: Pseudo code of the MoE Transformer layer in Whale.

```python
import tensorflow as tf


def prototype_gating(inputs, prototype_num, num_experts, expert_capacity):
    """
        Produce the combine and dispatch tensors used for dispatching and
        receiving tokens from their highest probability expert in each prototype.
    """

    _, total_token_num, model_dim = inputs.size()
    inputs = tf.broadcast_to(tf.expand_dims(inputs, axis=2),
                             [1, total_token_num, prototype_num, model_dim])  # dTZM
    # gating weights (MZF): weights for each expert, shared across experts.
    logits = tf.einsum('dTZM,MZF->dZTF', inputs, gating_weights)

    # Probabilities and indices for each token of what expert
    # it should be sent to in each prototype.
    raw_gates = tf.nn.softmax(logits)  # along expert dim, dZTF
    _, expert_index = tf.math.top_k(raw_gates, k=1)  # dZTk k=1
    expert_index = tf.squeeze(expert_index, [3])  # dZT
    expert_mask = tf.one_hot(expert_index, num_experts // prototype_num,
                             dtype=inputs.dtype)  # dZTF
    density_proxy = raw_gates  # dZTF
    importance = tf.ones_like(expert_mask[:, :, :, 0])  # dZT
    gate = tf.einsum('dZTF,dZTF->dZT', raw_gates, expert_mask)  # dZT

    # We compute cumulative sums of assignment indicators for each expert
    # index i \in 0..F-1 for each prototype independently.
    # First occurrence of assignment indicator is excluded.
    position_in_expert = tf.cumsum(expert_mask, exclusive=True, axis=2)  # dZTF

    # density[:, :, i] represents assignment ratio (num assigned / total) to
    # expert i as top expert without taking capacity into account.
    density_denom = tf.reduce_mean(importance, axis=2)[:, :, tf.newaxis] + 1e-6
    density = tf.reduce_mean(expert_mask, axis=2) / density_denom
    # density_proxy[:, :, i] represents mean of raw_gates for expert i, including
    # those of examples not assigned to i with top_k.
    density_proxy = tf.reduce_mean(density_proxy, axis=2) / density_denom

    with tf.name_scope('aux_loss'):
        # The MoE paper (https://arxiv.org/pdf/1701.06538.pdf) uses an aux loss of
        # reduce_mean(density_proxy * density_proxy). Here we replace one of
        # the density_proxy with the discrete density following mesh_tensorflow.
        aux_loss = tf.reduce_mean(density_proxy * density)  # element-wise
        aux_loss *= (num_experts // prototype_num) *
                    (num_experts // prototype_num) * loss_coef

    # Make sure that not more than expert capacity tokens can be dispatched to each expert.
    capacity = tf.cast(expert_capacity, dtype=position_in_expert.dtype))
    expert_mask *= tf.cast(tf.less(position_in_expert, capacity, dtype=expert_mask.dtype)
    position_in_expert = tf.einsum('dZTF,dZTF->dZT', position_in_expert, expert_mask)
    mask_flat = tf.einsum('dZTF->dZT', expert_mask)
    gate *= mask_flat

    # Construct combine tensor and dispatch mask.
    b = tf.one_hot(tf.cast(position_in_expert, dtype=tf.int32), expert_capacity,
                   dtype=inputs.dtype)  # dZTC
    a = tf.expand_dims(gate * mask_flat, -1) *
        tf.one_hot(expert_index, num_experts // prototype_num,  dtype=inputs.dtype)  # dZTF
    combine_tensor = tf.einsum('dZTF,dZTC->dTZFC', a, b, name='combine_tensor')  # dTZFC

    dispatch_mask = tf.cast(tf.cast(combine_tensor, tf.bool), inputs.dtype,
                            name='dispatch_mask')  # dTZFC
    dispatch_mask = tf.reshape(dispatch_mask,
        [1, total_token_num, prototype_num, -1, expert_capacity])  # dTZFC

    combine_tensor = tf.reshape(combine_tensor,
                               [1, total_token_num, num_experts, expert_capacity])  # dTEC

    return combine_tensor, dispatch_mask, aux_loss
```

Figure 8: Pseudo code of the Expert Prototyping (an example on top-1 gating).