Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *ACM Computing Surveys (CSUR)*, 2020.

Trieu H. Trinh and Quoc V. Le. A simple method for commonsense reasoning. *ArXiv*, abs/1806.02847, 2018.

Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *ArXiv*, abs/1706.03762, 2017.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *CoRR*, abs/2206.07682, 2022. doi: 10.48550/arXiv.2206.07682. URL https://doi.org/10.48550/arXiv.2206.07682.

Guillaume Wenzek, Marie-Anne Lachaux, Alexis Conneau, Vishrav Chaudhary, Francisco Guzm'an, Armand Joulin, and Edouard Grave. Ccnet: Extracting high quality monolingual datasets from web crawl data. In *LREC*, 2020.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: open pre-trained transformer language models. *CoRR*, abs/2205.01068, 2022a. doi: 10.48550/arXiv.2205.01068. URL https://doi.org/10.48550/arXiv.2205.01068.

Zhengyan Zhang, Yankai Lin, Zhiyuan Liu, Peng Li, Maosong Sun, and Jie Zhou. MoEfication: Transformer feed-forward layers are mixtures of experts. In *Findings of the Association for Computational Linguistics: ACL 2022*, pp. 877–890, Dublin, Ireland, May 2022b. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-acl.71. URL https://aclanthology.org/2022.findings-acl.71.

Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Y. Zhao, Andrew M. Dai, Zhifeng Chen, Quoc Le, and James Laudon. Mixture-of-experts with expert choice routing. *CoRR*, abs/2202.09368, 2022. URL https://arxiv.org/abs/2202.09368.

Yukun Zhu, Ryan Kiros, Richard S. Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 19–27, 2015.

A Training Setting

A.1 Hyperparameters

Table 6 specifies shared hyperparameters across all experiments, in which Table 5a contains ones for training data, optimizer, and efficient infrastructure techniques; and Table 5b for architecture. Then, Table 6a describes the hyperparameters specifically for Switch, Table 6b for LoRKM, Table 6c for PKM,

A.2 Data

Here is a detailed description of our pretraining corpus.

- **BookCorpus** (Zhu et al., 2015) consists of more than 10K unpublished books (4GB);
- **English Wikipedia**, excluding lists, tables and headers (12GB);
- **CC-News** (Nagel, 2016) contains 63 millions English news articles crawled between September 2016 and February 2019 (76GB);
- **OpenWebText** (Gokaslan & Cohen, 2019), an open source recreation of the WebText dataset used to train GPT-2 (38GB);

Table 5: Shared configuration

(a) Shared configuration for data, optimizer, and efficient infrastructure

Name	Values
#Tokens for training	60e9
#Tokens for warmpup	$375 \cdot 1024^2$
#Tokens per batch	$0.5 \cdot 1024^2$
#Tokens per sample	2048
#GPU	32
GPU	NVIDIA Tesla V100 32GB
Optimizer	Adam(βs $= (0.9, 0.98)$, $\epsilon = 1e-8$)
Weight Decay	0.01
Peak Learning Rate	3e-4
Learning Rate Scheduler	`polynomial_decay`
`clip_norm`	0.0
DistributedDataParallel backend	FullyShardedDataParallel
`memory-efficient-fp16`	True
`fp16-init-scale`	4
`checkpoint-activations`	True

(b) Shared configuration for architecture.

Name	Values
Objective	Causal Language Model (CLM)
Activation function(f)	GeLU
Model dimension (d)	1024
d_m of non-S-FFN	$4 \cdot 1024$
#Attention Head	16
#Layer	24
Dropout Rate	0.0
Attention Dropout Rate	0.0
`share-decoder-input-output-embed`	True

Table 6: Specific architecture configuration

(a) Switch

Name	Values
`moe-gating-use-fp32`	True
`moe-gate-loss-wt`	0.01
	i.e. CLM loss + $0.01\cdot$ auxiliary loss (Fedus et al., 2021)
Divide expert gradients by	$\sqrt{\text{\# Expert}} = \sqrt{B}$

(b) LoRKM

Name	Values
d_ℓ	128
BatchNorm after $\mathbf{x} \cdot \mathbf{D}$	False

(c) PKM

Name	Values
d_ℓ	128
# key table (§2.2.2)	1
BatchNorm after $\mathbf{x} \cdot \mathbf{D}$	True

- **CC-Stories** (Trinh & Le, 2018) contains a subset of CommonCrawl data filtered to match the story-like style of Winograd schemas (31GB);

- **English CC100** (Wenzek et al., 2020), a dataset extracted from CommonCrawl snapshots between January 2018 and December 2018, filtered to match the style of Wikipedia (292GB).

B BLOCK SIZE

B.1 VANILLAM WITH BLOCK SIZE $g > 1$

For VanillaM with block size $g > 1$, we also tried three other simple aggregation function, but they all under-perform Average. We show their results in Table 7.

Table 7: VanillaM with different simple aggregators

Selection method	g	#Parameters (Entire Model)	Train ZFLOPs	Aggregator	Out-of-Domain (22 domains; Avg. $\pm$ Std.)	In-Domain Train	In-Domain Val.
Dense Baseline	1	354.7M	0.212	N/A	16.96 ± 5.20	19.60	17.16
VanillaM	4096	858.3M	0.333	Avg$(\cdot)$	$\mathbf{15.56 \pm 4.62}$	18.33	**15.87**
				Avg$(\lvert\cdot\rvert)$	15.67 ± 4.66	—	15.94
				Max$(\cdot)$	16.11 ± 4.86	—	16.33
				Min$(\cdot)$	94.86 ± 57.63	—	17.08

B.2 ANALYSIS OF SMALLER BLOCK SIZES

We first quantify the intuition —"usage of model memory is more spread out" by number of activated memory cells shared between two random tokens — $\mathbb{E}[r]$. We define this quantity to be average of every S-FFN layer, to reflect the overall behavior — $\mathbb{E}[r] = \frac{1}{L_{\text{S-FFN}}} \sum_\ell \mathbb{E}[r_\ell]$, where $L_{\text{S-FFN}}$ is the number of S-FFN. Because block selection usually depends on a contextualized token embedding, it's hard to draw tokens in an i.i.d. fashion. Therefore, we estimate the the quantity by evaluating the model on a validation set. We sample N token pairs from each sequence for estimation:

$$\mathbb{E}[r_\ell] = \frac{1}{|\text{val}| \cdot N} \sum_{s \in \text{val}} \sum_{i=0:(x,y)_i \sim \text{Uniform}(s \times s)}^{N-1} |\mathcal{I}_x \cap \mathcal{I}_y| \cdot g \tag{6}$$

where $\mathcal{I}_x$ is the indices of selected memory block for token at position x, and similarly for $\mathcal{I}_y$.

RandHash, though, is an exception where uniform sampling is used. Therefore, $\mathbb{E}[r]$ could also be analytically calculated for various g, *when assuming tokens are also uniformly distributed*.

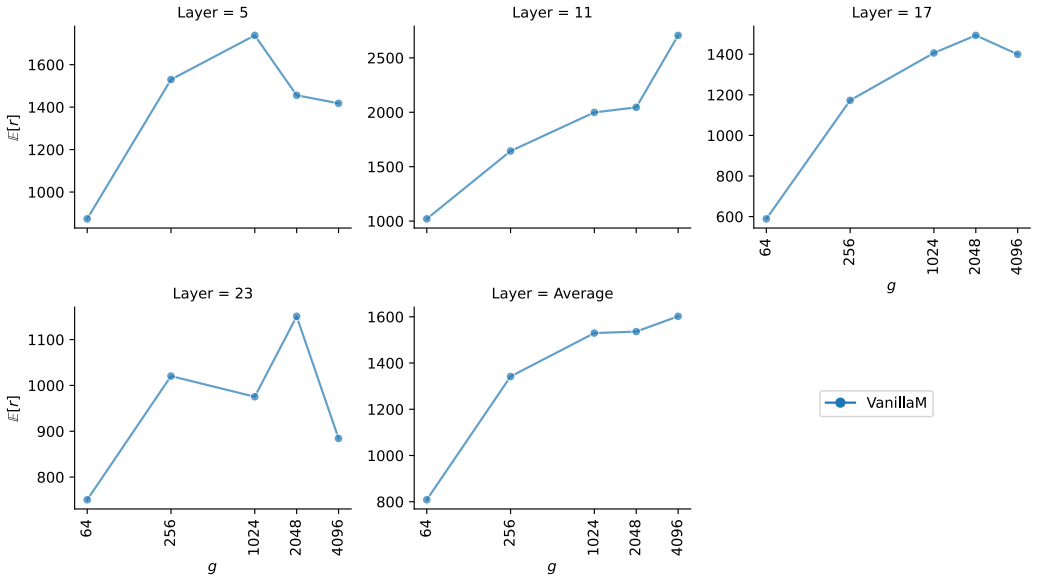
$$\mathbb{E}[r] = \frac{1}{L_{\text{S-FFN}}} \cdot L_{\text{S-FFN}} \cdot \mathbb{E}[r_\ell] = \sum_{i=1}^{b} \underbrace{\binom{b}{i}}_{\substack{\text{No. of such} \\ \text{block assignments}}} \cdot \underbrace{\prod_{j=0}^{i-1} \frac{b-j}{B-j}}_{\substack{\text{Probability of} \\ i \text{ overlaps}}} \cdot \underbrace{\prod_{k=0}^{b-i-1} \frac{B-b-k}{B-k}}_{\substack{\text{Probability of} \\ j \text{ non-overlaps}}} \cdot \underbrace{i \cdot g}_{\substack{r \text{ cells} \\ \text{in an overlap}}} \tag{7}$$

In Fig. 4a, 4b, we evaluate our model with $E = 16$ on our validation subset and calculate the estimations across various g. **It is observed that less sharing happens as block size decreases.** However, the empirical estimation for RandHash are relatively constant across granularity. We suspect this is due to the Zipf's law of tokens. Also, we note that the magnitude of $\mathbb{E}[r]$ are different for different methods. We defer the reason of this phenomena to future work.

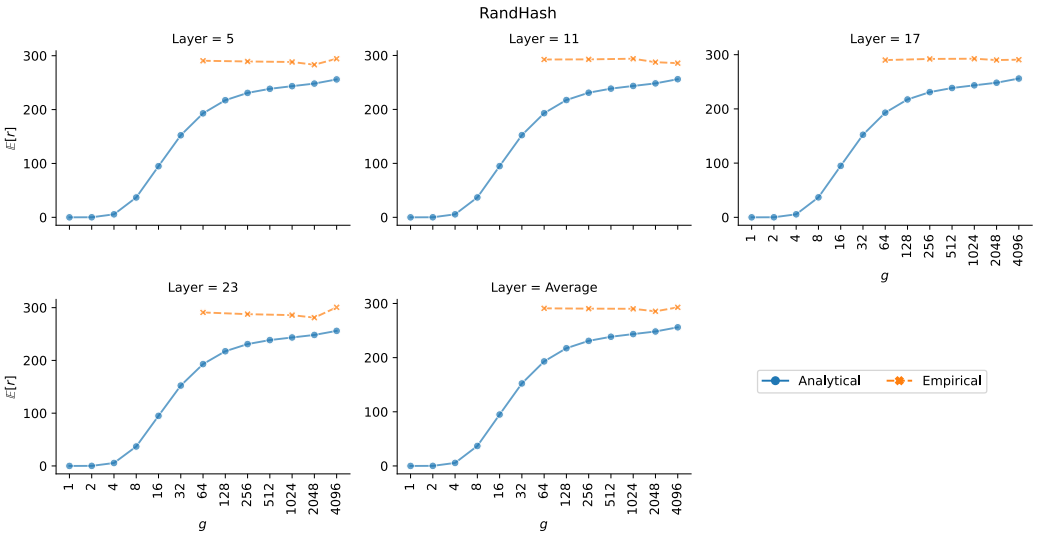
B.3 COST OF SMALLER BLOCK SIZES

B.3.1 COST OF GATE

RandHash is efficient for computation because a hash table theoretically has time complexity $O(1)$. In contrast, a conventional learned gate 2.2.1 has an d-dimensional embedding for each memory block. Therefore, with total of B memory blocks, it has the time complexity of $O(d \cdot B)$. In Table



(a) VanillaM: empirical estimation from sampling



(b) RandHash: both analytical value from close form calculation and empirical value from sampling

Figure 4: Expected of shared memory cells across various block size g

8 we show how the FLOPs percentage of learned gate in a single forward-backward computation changes with respect to the change in memory block size, where we assume setup in §4 is adopted.

C AVG-K

C.1 RATIONALE TO USE AVG IN AVG-K

We heavily base our choice on experiments with aggregators in VanillaM (in Table 7). From the experiments with average absolute value (after GeLU), we hypothesized that a positive feature is good at predicting the value of a label/token against all others. In contrast, a negative value is good at negating the prediction of a single token. As such, positive features are more predictive than negative

Table 8: FLOPs percentage of learned gate increases when memory block size g decreases

TFLOPs of	Memory block size (g)								
	4096	2048	1024	512	256	128	64	32	1
4 learned gates (across 24 layers)	0.275	0.552	1.10	2.20	4.40	8.80	17.6	35.2	1124
Entire model	1850	1850	1860	1860	1860	1860	1870	1890	2980
%	0.0149	0.0298	0.0595	0.118	0.237	0.473	0.941	1.86	37.718

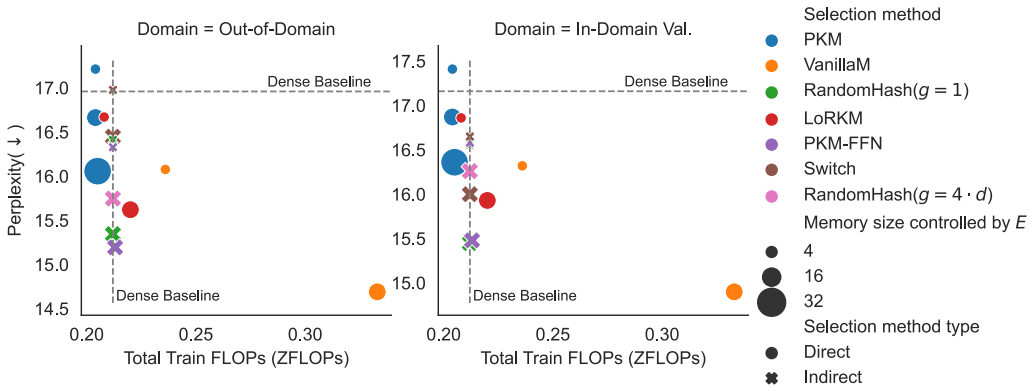


Figure 5: FLOPs-Perplexity trade-off different models where direct/indirect methods are further distinguished by model name.

ones. Although the situation might be different for **Avg-K** (before GeLU), we expect the selection will only be affected more because of the larger impact of negative value.

Also, we consider the experiment with max-pooled hidden states (i.e., Max(·)). This experiment shows that a memory block hardly has a single key-value cell that dominates over others since Max(·) underperforms Avg(·) and Avg(| · |). What makes it worse, the max operation will overlook lots of hidden states at selection, but the overlooked hidden states still contribute to the computation. In contrast, the performance increases when we consider the average (or average of the absolute values) where every hidden state contributes to the decision. Although the situation is slightly different in **Avg-K**, the "max-pooled" version of **Avg-K** will only overestimate the hidden states information even more, and the aggregated value won't be indicative of the hidden states used for computation.

The last consideration we have is that the average function is linear. When we select experts, we use the dot product between input and averaged keys. Due to the linearity, this value is equivalent to taking the dot product between the input and every key and taking the average (See Appendix C.2). Thus, using this design choice saves a great amount of computation compared with VanillaM, while keeping the neural memory analogy.

C.2 Avg-K analysis through comparison with VanillaM

Avg-K essentially applies an average pooling to the unfactorized $\mathbf{K}^g$ to create representation of each block. Due to the linearity of averaging, the operation $\mathbf{e}_i \cdot \mathbf{x}$ is equivalent to calculate the average of dot products within a block before GeLU and select blocks with the average of dot products:

$$\mathbf{e}_i \cdot \mathbf{x} = \left(\frac{1}{g} \cdot \sum_{j=0}^{g-1} \mathbf{k}_j^{(i)} \right) \cdot \mathbf{x} = \frac{1}{g} \cdot \sum_{j=0}^{g-1} \left(\mathbf{k}_j^{(i)} \cdot \mathbf{x} \right) = \texttt{Avg} \left(\mathbf{x} \cdot \left(\mathbf{K}^{(i)} \right)^\top, \texttt{dim=0} \right) \quad (\textbf{Avg-K}) \tag{8}$$

In contrast, VanillaM uses average *after* GeLU(§5.1):

$$\frac{1}{g}\sum_{j=0}^{g-1}\text{GeLU}\left(\mathbf{k}_j^{(i)}\cdot\mathbf{x}\right) \quad \text{(VanillaM)}$$

Because GeLU is a non-linear function, average from **Avg-K** could be shared across tokens. In contrast, VanillaM can't, and thus making **Avg-K** efficient.

In Fig 6, we experiment both methods with various g. We observe when g decreases from 4096, the perplexity of **Avg-K** drops more drastically than VanillaM. We believe this observation highlights the impact of GeLU. Because $\lim_{x\to-\infty}\text{GeLU}(x)=0$, it protects the average in VanillaM from some very negative values. Thus, **Avg-K** with larger g included more and potentially very negative values to average over, and thus leads to worse choices than ones made by VanillaM. On the other hand, when g decreases, this "negative value" problem is mitigated. When there are more blocks available for selection (smaller g), because negative dot products affects **Avg-K** more, it prefers blocks with more or very positive dot products; whereas, VanillaM is protected from negative value so it fails to detect those blocks. Therefore, **Avg-K** with $g \leq 256$ could achieve an even better perplexity.

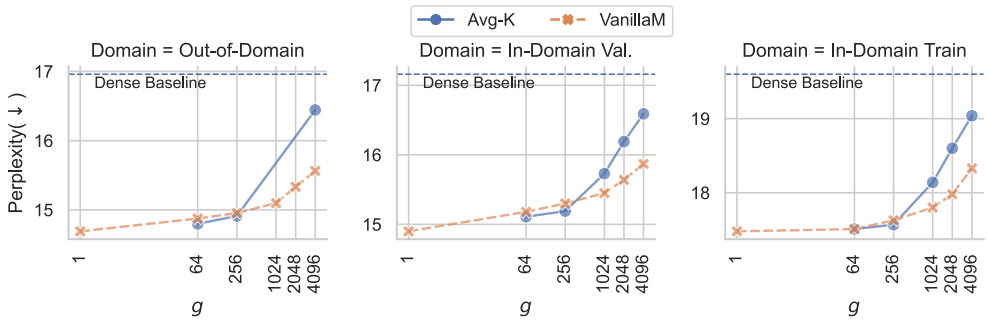


Figure 6: Perplexity performance (lower the better) of **Avg-K** and VanillaM across various g. We observe large drop in perplexity when g decreases in **Avg-K** and less so in VanillaM; and **Avg-K** slightly outperform VanillaM with $g \leq 256$.

C.3 **Avg-K** LOAD BALANCING ANALYSIS

On the same validation set as used in §B, we also conduct a load balancing analysis of memory blocks. Fig. 7 shows that **Avg-K** and VanillaM disproportionally used some memory blocks.

D PRELIMINARY STUDY FOR RELATED WORK

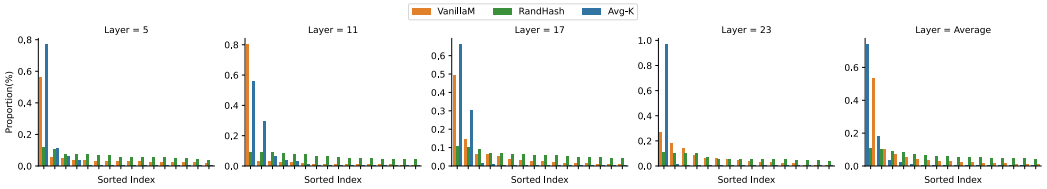
D.1 TERRAFORMER ANALYSIS

Controller in Terraformer Jaszczur et al. (2021) uses a controller to score all memory cells and pre-select a subsets — Controller($\mathbf{x}$) — for computation.

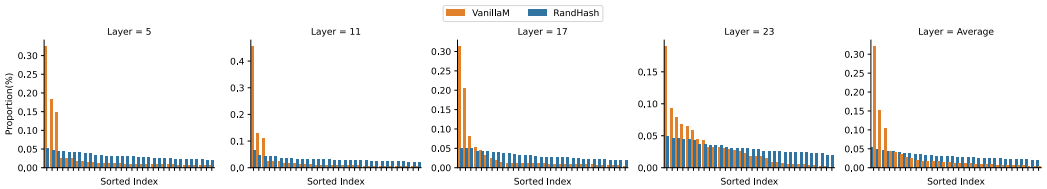
$$\mathbf{y} = \sum_{i\in\text{Controller}(\mathbf{x})} f(\mathbf{x}\cdot\mathbf{k}_i)\cdot\mathbf{v}_i \tag{9}$$

This is closest to our PKM-FFN, since their controller is essentially a gate with low-rank key table in LoRKM— $\mathbf{g}(\mathbf{x}) = (\mathbf{x}\cdot\mathbf{D})\cdot(\mathbf{K}')^\top$, where $\mathbf{D}\in\mathbb{R}^{d\times d_\ell}$, $\mathbf{K}\in\mathbb{R}^{d_m\times d_\ell}$, and $d_\ell \ll d$. The difference is that they additionally assume the estimation from gate (and memory) could be seen as chunked into blocks and only select top-1 memory cell scored by the controller from each blocks:

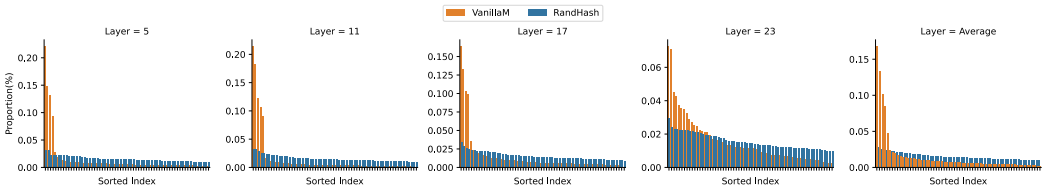
$$\mathbf{y} = \sum_{i=0}^{B-1}\mathbf{g}(\mathbf{x})_{j^*}^{(i)}\cdot f(\mathbf{x}\cdot\mathbf{k}_{j^*}^{(i)})\cdot\mathbf{v}_{j^*}^{(i)}, \text{ where } j^* = \arg\max_j \mathbf{g}(\mathbf{x})_j^{(i)}$$



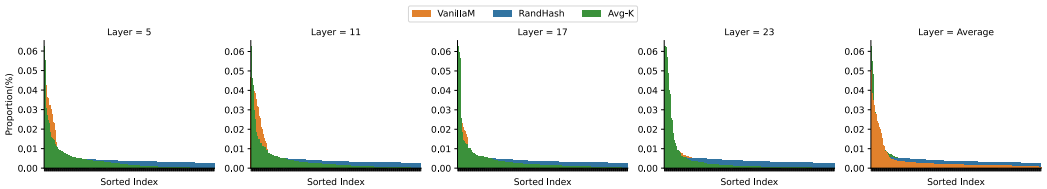
(a) $g = 4096$



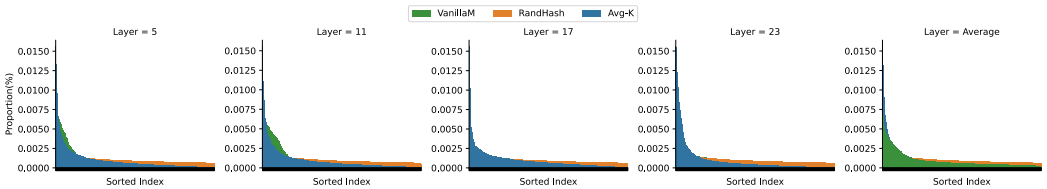
(b) $g = 2048$



(c) $g = 1024$



(d) $g = 256$



(e) $g = 64$

Figure 7: Load balancing of VanillaM, RandHash, **`Avg-K`**. The height of bar represents the proportion of memory block usage with which the memory block are sorted (in descending order).

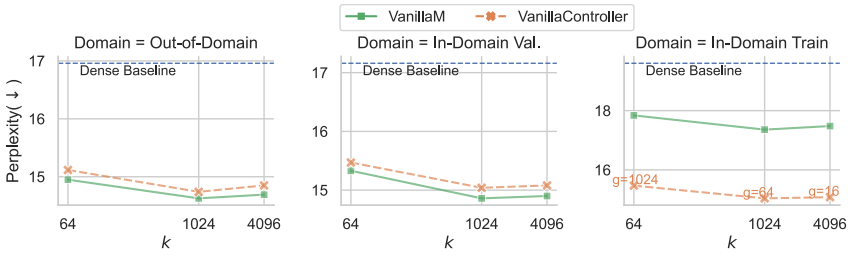
Therefore, their number of active memory cells k is equal to d_m/g.

Similar to our contrastive pair of PKM-FFN and VanillaM, we hypothesize a "vanilla" version of their methods. Memory is chunked into blocks of size g — $\mathbf{K}^g = [\mathbf{K}^{(0)}; \cdots; \mathbf{K}^{(B-1)}]$ and similarly for $\mathbf{V}^g$. Then, one chooses the top-1 with $\mathbf{x} \cdot (\mathbf{K}^{(i)})^\top$. We call it **VanillaController**.

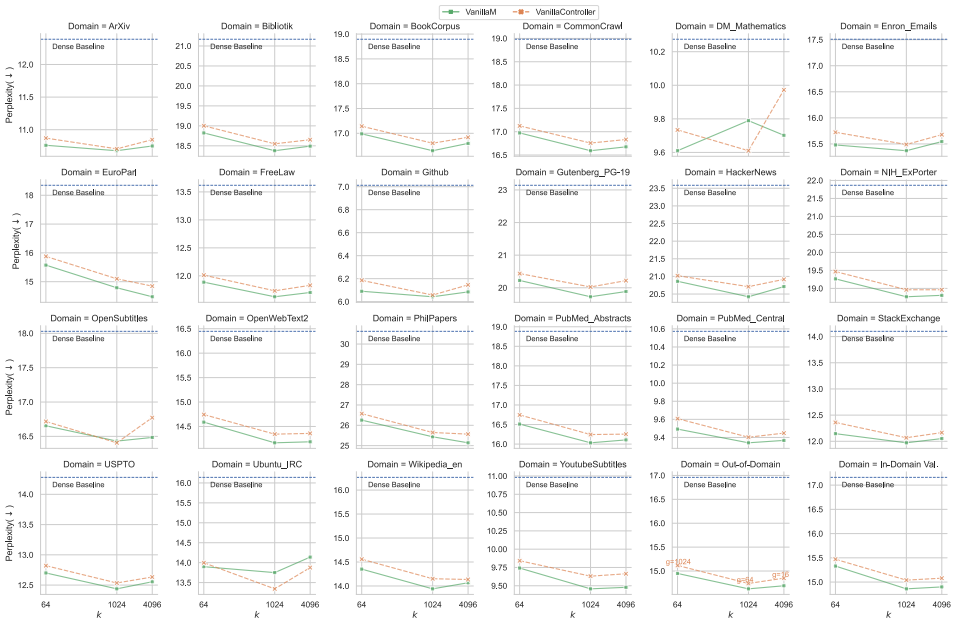
$$\mathbf{y} = \sum_{i=0}^{B-1} f(\mathbf{x} \cdot \mathbf{k}_{j^*}^{(i)}) \cdot \mathbf{v}_{j^*}^{(i)}, \text{ where } j^* = \arg\max_j \mathbf{x} \cdot (\mathbf{K}^{(i)})^\top$$

In Fig. 8, we compare VanillaController to VanillaM with $g = 1$, because the *actual* section is at the level of $g = 1$. We set k in VanillaM to the one determined by equation above. We observe VanillaM

outperforms VanillaController. Although the controller design as a gating function is justified (§5.2), the decision choice of "chunking memory but only select the best memory cells" seems unmotivated. Thus, we exclude this design setup from our analysis.



(a) Average perplexity performance (lower the better)



(b) Performance on individual domain in PILE perplexity (lower the better)

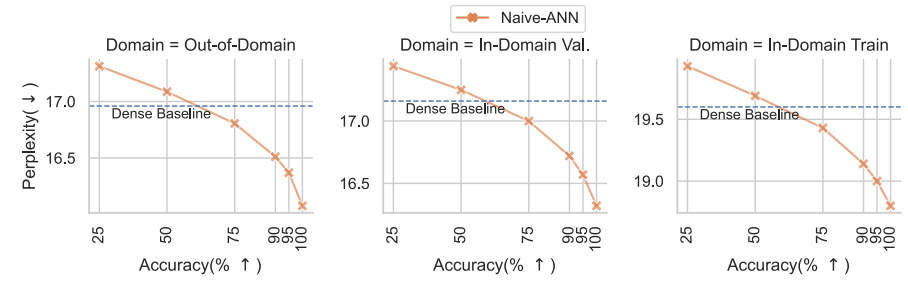
Figure 8: Perplexity performance (lower the better) of VanillaM (g=1) and VanillaController with $E = 16$.

D.2 ANN

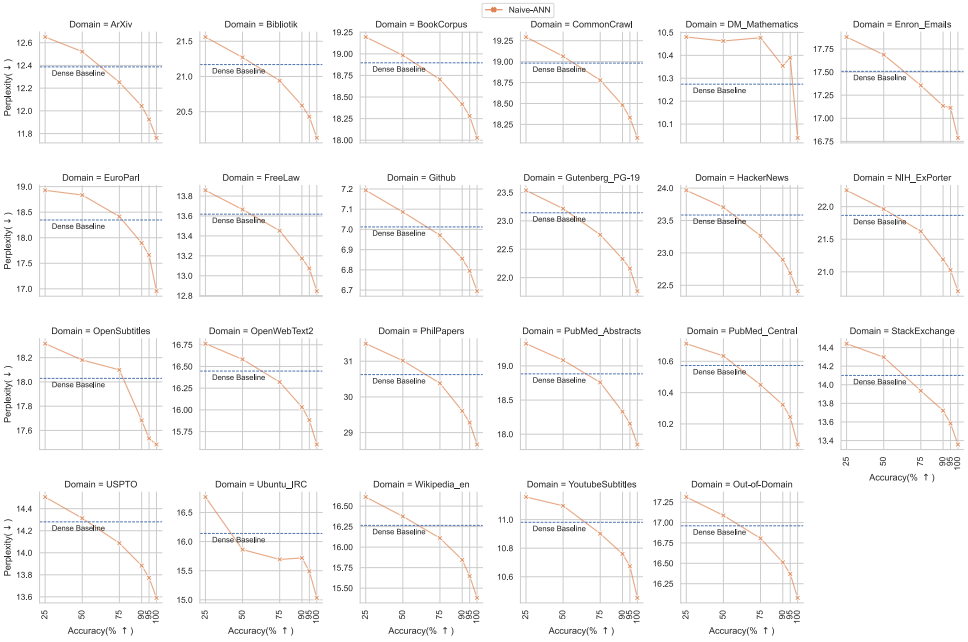
Since ANN is an approximation to exact search, we propose to randomly sabotage VanillaM, which uses the exact search. Given a k, we randomly swap $n\%$ of the top-k of memory coefficient $\mathbf{m}$ (exact search results) with non-top-k values (during training and validation), and has accuracy $(100 - n)\%$ We call it **Naive-ANN**. This is meant to set up a random baseline for ANN, because different ANN techniques might make systematic mistakes, rather than a random one. However, we believe this could still serve as a proxy and shed light on how it affects performance. As we see in Fig. 9, the model quality is sensitive to the quality of ANN.

In our preliminary study, we found building data structure after every update is expensive. This leads to some critical drawback when we apply the techniques to model parameter. Although one could amortize the cost by periodically building, the outdated data structure will lead to lower accuracy. If one chooses a hyperparameter that leads to higher quality, the cost of preprocessing and the corresponding search will be even higher. What makes it worse, the current ANN methods' search

either don't support speedup by using GPU, or is not very well-integrated with GPUs — slower than calculating the exact dot product with CUDA kernel.



(a) Average perplexity performance (lower the better)



(b) Performance on individual domain perplexity in PILE (lower the better)

Figure 9: Perplexity performance (lower the better) of Naive-ANN with $E = 4$.

Table 9: Detailed out-of-domain perplexity for Table 3. Best two performance on each domain is in bold. Relative ranking on each domain generally follows the relative ranking by averaged performance (i.e. last row).

	Selection method Type		Direct				Indirect	
	Selection method	Dense Baseline	PKM			VanillaM	PKM-FFN	RandHash
	E	1	16	32	32	16	16	16
	k	4096	4096	4096	8192	4096	4096	4096
Out-of-Domain	ArXiv	12.39	12.20	11.82	11.89	**10.75**	**11.05**	11.22
	Bibliotik	21.17	20.82	20.15	20.25	**18.49**	**19.11**	19.33
	BookCorpus	18.90	18.61	18.09	18.19	**16.80**	**17.26**	17.49
	CommonCrawl	18.98	18.68	18.09	18.20	**16.68**	**17.23**	17.40
	DM_Mathematics	10.27	10.34	10.05	10.28	**9.70**	**9.72**	9.91
	Enron_Emails	17.51	17.23	16.67	16.68	**15.55**	**15.90**	16.18
	EuroParl	18.35	17.79	17.01	17.05	**14.48**	15.50	**15.03**
	FreeLaw	13.62	13.34	12.84	12.93	**11.70**	**12.11**	12.29
	Github	7.01	6.91	6.67	6.68	**6.08**	**6.25**	6.37
	Gutenberg_PG-19	23.14	22.61	21.83	22.03	**19.88**	**20.74**	21.07
	HackerNews	23.58	23.35	22.44	22.62	**20.71**	**21.36**	21.76
	NIH_ExPorter	21.87	21.45	20.59	20.77	**18.81**	**19.48**	19.69
	OpenSubtitles	18.03	17.99	17.46	17.44	**16.48**	**16.84**	17.10
	OpenWebText2	16.45	16.15	15.60	15.68	**14.19**	**14.73**	14.74
	PhilPapers	30.63	30.02	28.50	28.74	**25.14**	**26.44**	26.60
	PubMed_Abstracts	18.88	18.50	17.71	17.92	**16.11**	**16.75**	16.90
	PubMed_Central	10.57	10.40	10.08	10.14	**9.37**	**9.66**	9.71
	StackExchange	14.10	13.85	13.37	13.45	**12.05**	**12.46**	12.72
	USPTO	14.28	14.07	13.58	13.68	**12.55**	**12.96**	13.09
	Ubuntu_IRC	16.14	15.40	14.95	15.08	**14.14**	**14.35**	14.62
	Wikipedia_en	16.26	16.03	15.33	15.48	**14.07**	**14.51**	14.59
	YoutubeSubtitles	10.98	10.86	10.41	10.41	**9.48**	9.87	**9.77**
	Average	16.96	16.66	16.06	16.16	**14.69**	**15.19**	15.35

Table 10: Detailed out-of-domain perplexity for Table 4. Best performance on each domain is in bold. Relative ranking on each domain generally follows the relative ranking by averaged performance (i.e. last row).

	Selection method	Dense Baseline	RandHash		Switch	PKM-FFN	**Avg-K**		
	g	1	4096	1	4096	1	4096	256	64
Out-of-Domain	ArXiv	12.39	11.42	11.22	12.32	11.05	12.09	10.99	**10.85**
	Bibliotik	21.17	19.84	19.33	19.87	19.11	20.49	18.70	**18.61**
	BookCorpus	18.90	17.94	17.49	17.85	17.26	18.33	16.86	**16.82**
	CommonCrawl	18.98	17.84	17.40	17.70	17.23	18.42	16.89	**16.81**
	DM_Mathematics	10.27	10.22	9.91	10.63	9.72	10.25	9.62	**9.61**
	Enron_Emails	17.51	16.70	16.18	17.36	15.90	17.20	15.65	**15.55**
	EuroParl	18.35	15.55	15.03	19.63	15.50	17.38	15.32	**15.13**
	FreeLaw	13.62	12.56	12.29	12.80	12.11	13.20	11.84	**11.77**
	Github	7.01	6.51	6.37	6.96	6.25	6.89	6.18	**6.12**
	Gutenberg_PG-19	23.14	21.55	21.07	21.81	20.74	22.36	20.07	**19.98**
	HackerNews	23.58	22.30	21.76	22.59	21.36	22.95	20.82	**20.65**
	NIH_ExPorter	21.87	20.22	19.69	20.99	19.48	21.09	19.09	**19.01**
	OpenSubtitles	18.03	17.33	17.10	17.31	16.84	17.74	**16.62**	16.66
	OpenWebText2	16.45	15.16	14.74	15.51	14.73	15.87	14.48	**14.39**
	PhilPapers	30.63	27.53	26.60	30.84	26.44	29.51	25.90	**25.72**
	PubMed_Abstracts	18.88	17.39	16.90	18.36	16.75	18.24	16.34	**16.33**
	PubMed_Central	10.57	9.94	9.71	10.28	9.66	10.30	9.50	**9.45**
	StackExchange	14.10	13.04	12.72	13.78	12.46	13.73	12.23	**12.13**
	USPTO	14.28	13.41	13.09	13.54	12.96	13.89	12.73	**12.62**
	Ubuntu_IRC	16.14	14.95	14.62	14.78	14.35	15.50	14.26	**13.59**
	Wikipedia_en	16.26	14.98	14.59	15.67	14.51	15.73	14.23	**14.12**
	YoutubeSubtitles	10.98	10.06	9.77	11.25	9.87	10.59	9.73	**9.67**
	Average	16.96	15.75	15.35	16.45	15.19	16.44	14.91	**14.80**