RICE UNIVERSITY

By

Zichang Liu

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE

Anshumali Shrivastava (Chair)

Αναστασιος Κυριλλιδης (Apr 2, 2024 22:43 CDT)

Anastasios Kyrillidis

Richard G. Baraniuk

HOUSTON, TEXAS

April 2024

ABSTRACT

Dynamic Sparsity for Efficient Machine Learning

by

Zichang Liu

Over the past decades, machine learning(ML) models have delivered remarkable accomplishments in various applications. For example, large language models usher in a new wave of excitement in artificial intelligence. Interestingly, these accomplishments also unveil the scaling law in machine learning: larger models, equipped with more parameters and trained on more extensive datasets, often significantly outperform their smaller counterparts. However, the trends of increasing model size inevitably introduce unprecedented computation resource requirements, creating substantial challenges in model training and deployments.

This thesis aims to improve the efficiency of ML models through algorithmic advancements. Specifically, we exploit the dynamic sparsity pattern inside ML models to achieve efficiency goals. Dynamic sparsity refers to the subset of parameters or activations that are important for a certain data, and different data may have a different dynamic sparsity pattern. We advocate identifying the dynamic sparsity pattern for each data set and focusing computation and memory resources on it.

The first part of this thesis centers around the inference stage. We verify the existence of dynamic sparsity in trained ML models, namely, within the classification layer, attention mechanism, and transformer layers of trained models. Further, we demonstrate that such dynamic sparsity can be cheaply predicted and leveraged

for each data to improve the inference efficiency goals. The subsequent part of the dissertation will shift its focus to the training stage, where dynamic sparsity emerges as a tool to mitigate the problem of catastrophic forgetting or data heterogeneity in federated learning to improve training efficiency.

Acknowledgments

I want to express my gratitude to my advisor, Dr. Anshumali Shrivastava, for all the guidance and support. I joined his lab right after college, when I may have just scratched the surface of research. For the past five years, he has taught me how to become an independent researcher, among which are two particular things that will probably benefit me for the rest of my life. First, regardless of the frustrations or discouragements from outside voices, he has consistently encouraged me to pursue what I am inquisitive and passionate about. Second, by setting an excellent example himself, he has taught me that research is not just about publishing papers but about the potential of making real-world impacts.

I want to thank Dr.Anatasios Kyrilidis and Dr.Richard Baraniuk, members of my Ph.D. committee, for their valuable feedback on my proposal and thesis. I would also like to thank all my mentors and collaborators during my internship: Yifei Ma, Anoop Deoras, Jonas Mueller, Xingjian Shi, Zhe Zhao, Sunny Liu, Yuening Li, etc. I was fortunate to engage with and learn from researchers with diverse expertise and backgrounds.

I must thank all the members of my research group, Ryan Spring, Beidi Chen, Tharun Kumar Reddy, Benjamin Coleman, Gaurav Gupta, Aditya Desai, Zhaozhuo Xu, Tianyi Zhang, and many others. I want to send special thanks to Beidi Chen and Benjamin Coleman, whose mentorship has played a pivotal role in my research journey. Further, I would like to express my gratitude to the peers I met during my internship: Matias Mendieta, Yunhe Gao, Yuji Roh, Chengyu Dong, and Jiayi Chen. Without their support and friendship, I would not have enjoyed my internship as much. I want to send a special thank you to Hui Ye, Xinyu Wu, Yumeng Liu, Yunxi Liu, Zhuang

Wang, Yilong Ju for their friendship. I want to thank my partner, Weitao Wang, for his understanding and support through the good and bad times of my Ph.D. journey.

Last but not least, I would like to extend my gratitude to my parents, Rongxing Liu and Xueli Zhan. Though we couldn't see each other often during the pandemic, they are always there for me with unconditional support and love.

This thesis received generous support from various sources, including the National Science Foundation (IIS-1652131, BIGDATA-1838177), AFOSR-YIP FA9550-18-1-0152, the ONR DURIP Grant, the ONR BRC grant on Randomized Numerical Linear Algebra, and the Ken Kennedy Institute BP fellowship.

Contents

Abstract ii

Acknowledgments iv

List of Illustrations x

List of Tables xv

1 Introduction 1

1.1 Contribution . . . 3

1.1.1 Contextual Sparsity for Efficient LLMs at Inference Time . . . 3

1.1.2 Exploiting the Persistence of Importance Hypothesis for KV Cache Compression at Inference Time . . . 3

1.1.3 Hashing Large Output Space for Cheap Inference . . . 4

1.1.4 Retaining Knowledge for Learning with Dynamic Definition . 5

2 Background 7

2.1 Locality Sensitive Hashing . . . 7

3 Contextual Sparsity for Efficient LLMs at Inference Time 9

3.1 Introduction . . . 9

3.2 Related Work and Problem Formulation . . . 13

3.2.1 LLM Inference Latency Breakdown . . . 14

3.2.2 Problem Formulation . . . 15

3.3 Pre-trained LLMs are Contextually Sparse . . . 16

3.3.1 Contextual Sparsity Hypothesis . . . 17

3.3.2 Token Clustering in Attention Layers . . . 18

3.3.3 Slowly Changing Embeddings across Layers 20
3.4 DeJavu 23
3.4.1 Contextual Sparsity Prediction in MLP Blocks 23
3.4.2 Contextual Sparsity Prediction in Attention Blocks 25
3.4.3 Detailed workflow 26
3.4.4 Reducing Overhead with Asynchronous Execution 26
3.4.5 Hardware-efficient Implementation 28
3.5 Empirical Evaluation 30
3.5.1 End-to-End Result 30
3.5.2 Ablation Results 32

4 Exploiting the Persistence of Importance Hypothesis for KV Cache Compression at Inference Time 36

4.1 Introduction 36
4.2 Problem Description and Related Work 39
4.2.1 LLM Inference Memory Breakdown 40
4.2.2 Efficient Attention 41
4.3 The Persistence of Importance Hypothesis 42
4.3.1 Repetitive Attention Pattern. 42
4.3.2 The Persistence of Importance Hypothesis 43
4.3.3 Attention Weights Decides the Pivotal Tokens 45
4.4 Sequential Token Generation Under budget 47
4.4.1 Budget KV Cache for Single Attention Head 48
4.4.2 Theoretical Analysis. 50
4.5 Empirical Evaluation 53

5 Hashing Large Output Space for Cheap Inference 58

5.1 Introduction 58

5.2 Bridging the Gap Between AMIPS and NN Inference 62
5.2.1 Notation and Formulation 62
5.2.2 The Hardness of Inference by AMIPS 62
5.2.3 The Retrieval Oracle 63
5.2.4 Algorithm Overview 64
5.2.5 Offline Preprocessing: Index Output Layer Neurons 66
5.2.6 Online Inference 69
5.3 Evaluation 70
5.3.1 Main Result 73
5.3.2 Study on *HALOS* for Label Sensitivity 73
5.3.3 Study on *HALOS* for Efficiency 75
5.3.4 Ablation Studies on *HALOS* 76
5.3.5 Study on *HALOS* for Accuracy 78

6 Retaining Knowledge for Learning with Dynamic Definition 82

6.1 Introduction 82
6.2 Related Work 84
6.3 RIDDLE for Learning with Dynamic Definition 86
6.3.1 Architecture 86
6.3.2 Intuition 88
6.3.3 RIDDLE for Learning with Dynamic Definition 88
6.4 RIDDLE is a Universal Function Approximator 91
6.4.1 Weighted LSH Kernel Sums are Universal Function Approximators 93
6.4.2 RIDDLE for Weighted LSH Kernel Sums 94
6.5 Distributed Efficiency Analysis 95
6.5.1 Parallel Models with the Representer Sketch 96

6.5.2 Mergeable Models with the Representer Sketch 97
6.6 Experiments 99
6.6.1 RIDDLE for Learning under Dynamic Definition 99
6.6.2 Main Results 101
6.6.3 Study on Expressiveness and Efficiency 104

7 Conclusion **106**

Bibliography **108**

Illustrations

3.1 (1) LLMs have up to 85% contextual sparsity for a given input. (2) Contextual sparsity has much better efficiency-accuracy trade-offs (up to 7×) than non-contextual sparsity or static sparsity. 10
3.2 DEJAVU uses lookahead predictors to side-step prediction costs: given the input to the attention layer at block k, they (asynchronously) predict the contextual sparsity for the MLP at block k, and given the input to the MLP at block k, they predict the sparsity for the attention head at the next layer. 13
3.3 In Figure (a), we plot the percentage of not-activated attention heads. By only keeping heads that yield large output norms, we can silence over 80% attention heads for a given token. In Figure (b), we plot the average sparsity we impose on MLP layers. We can zero out over 95% of MLP parameters for a given token. 17
3.4 We visualize the attention scores of three different heads for an exemplary sentence. Head 42 and Head 44 give heavy attention scores on particular tokens while Head 43 is more uniform. 20

3.5 **Slowly Changing Embedding.** Figure (a) shows the median cosine similarity between representations at two consecutive layers across all layers for different OPT models. All models show a similarity greater than 95%. Figure (b) shows cosine similarity stays high even a few layers apart. For the residual connection $X' = X + F(X)$ inside each block, we plot the ℓ_2 norm of X and $F(X)$ in Figure (c) and Figure (d). $\|X\|$ is significantly higher than $\|F(X)\|$, which explains the slowly changing embedding. 21

3.6 Detailed diagram on the sparsified computation process of MLP and Attention. Notation refers to Section 3.2.2 27

3.7 **Accuracy Trend for dejavu-OPT-175B**. This figure shows the accuracy of DEJAVU-OPT-175B on language modeling datasets and downstream tasks when we set different sparsity at test time. In general, DEJAVU-OPT-175B incurs no accuracy drop until 75% sparsity. 30

3.8 Average per-token latency (ms) with batch size 1 on 8 A100-80GB with NVLink when generating sequences with prompt lengths 128, 256, 512, and 1024, using FP16. DEJAVU speeds up generation by 1.8-2× compared to the state-of-the-art FT and by 4.8-6× compared to the widely used HF implementation. 31

3.9 Union contextual sparsity with larger batch size. 35

4.1 **Repetitive Attention Pattern**. We plot the attention map at three token positions in a sentence. Only five attention heads are plotted for a clearer presentation. We discretize the attention score such that the high score is dark green, and the low score is light green. In Figure 4.1(a), the token at position 178 pays heavy attention to positions 27, 63, 98, etc. This pattern is also present in the attention maps of position 228 and position 278. 37

4.2 Persistence ratio and the corresponding size of the pivotal token set. The persistence ratio is over 95% in most layers, with decreases at the later layers. Meanwhile, the number of pivotal tokens is considerably smaller than the sequence length. This suggests that the pivotal tokens of later half sentences are almost all included in the set of first halves. 44

4.3 Accuracy trend of SCISSORHANDS on language modeling dataset and downstream tasks with different KV cache compression. In general, SCISSORHANDS incurs no accuracy drop until 5× compression on OPT-66B. 52

4.4 Score between OPT and SCISSORHANDS. 55

4.5 We plot the attention map corresponding to Section 4.3.1 but with a randomly initialized OPT. We observe no repetitive attention for a randomly initialized model. 56

5.1 Top 10 logit values for three randomly sampled testing data from extreme classification dataset Wiki10-31k. Logit dots correspond to label class is marked in yellow. The gap between label logits and non-label is narrow, creating difficulties for AMIPS algorithms 59

5.2 HALOS works in two stages: (1) Offline Processing: We first build hash table with randomly initialize hash functions. Output layer neurons are treated as data, and their indexes are stored in the hash table. Input embedding from last hidden layer is treated as query. Based on current hash table and groud truth information from NN training dataset, we train the randomly initialized hash functions so that input embedding query can find its according label neuron(marked in orange). We rebuild hash tables with the updated hash function for online inference. (2) Online Inference: NN prediction is computed on a subset of neurons retrieved by the input embedding from hash tables rather than the entire last layer neurons. 61

5.3 Plot of Label Recall versus Query Per Second. On all four dataset, the line for *HALOS*(in red) is higher than all AMIPS baselines. Specifically, at every query speed, *HALOS* recalls more label neurons comparing to AMIPS baselines. This validates our arguments that naively applying AMIPS on the output layer is not sensitive to the label class, which may hurt model accuracy as shown in Table 5.1. Wiki-Text2 refers to the LSTN network. 70

5.4 Plot of Recall versus Query Per Second. The lines representing *HALOS*(red) is the highest across datasets. At every recall level, *HALOS* processes more data, leading to better query efficiency comparing to AMIPS baselines. 74

5.5 Collision probability is measured as the probability a pair of inputs hashed in the same bucket. Blue line plots the collision probability between positive pairs. Green line plots the collision probability between negative pairs. The starting point represents the probabilities with randomized initialized hashing functions. It is evident that the proposed learning mechanism pushes positive pairs to land in the same bucket while seperates negative pairs. And since positive pairs are collected as input embedding with its label neurons, *HALOS* is more sensitive to label class and achieve high label recall. 77

6.1 The figure demonstrates a toy RIDDLE model with $L = 5$ and $R = 4$. Given an input x, we calculate its hash code $h(x)$ and access the weights at those locations. We return the average as the output. . . . 83

6.2 RIDDLE can be used as a stand-alone model, as shown in (a). We use this setting for experiments on MNIST-binary and all experiments studying expressiveness and efficiency. RIDDLE can also be combined with existing neural networks, as shown in (b). We replace the classification layer with RIDDLE for experiments on CIFAR10, ImageNet, and News 87

6.3 Accuracy on the original test set while training on the update distribution. We observe that the RIDDLE performance remains nearly constant for all datasets, while baseline methods degrade rapidly with inputs from the new distribution. 98

Tables

3.1 Theoretical breakdown for prompting versus token generation (tensor model parallelism on 8 A100-80G GPUs). 14
3.2 Theoretical breakdown for Attention block versus MLP block in one transformer layer when generating one token (tensor model parallelism on 8 A100-80G GPUs). 15
3.3 Latency breakdown of generating 1 token under the setting of batch size 1 and prompt length 128 on 8 A100-80GB. 15
3.4 Accuracy of zero-shot tasks and language modeling when sparsifying the MLP block and the Attention block separately. The sparsity is set at 85% for MLP-block and 50% for Attention-block. DEJAVU incurs no accuracy drop across the boards. 32
3.5 DEJAVU-OPT66B on zero-shot downstream task. 34
3.6 DEJAVU-BLOOM on zero-shot downstream task. 34
3.7 DEJAVU-OPT-175B with 4-bit quantization. 34

4.1 The memory consumption of model weights and KV cache for three different LLMs at batch size 128 and sequence length 2048 shows that the KV cache dominates the memory consumption. 41
4.2 Maximum batch size before hitting out of memory on a box of 8 A100 80GB GPU when models are deployed with its maximum sequence length. 41
4.3 Perplexity on C4 with different sequence lengths. 54

4.4 Applying 4-bit quantization on top of SCISSORHANDS on Hellaswag. . 54
4.5 Generated examples using OPT-13B with full cache and SCISSORHANDSat different compression ratio. 57

5.1 This table summarizes the performance of HALOS and other baselines on language datasets. 79
5.2 This table summarizes the performance of HALOS and other baselines on recommondation datasets. 80
5.3 Effect of L and K on P@1 and P@5 on Delicious-200K. K is the number of hash functions, L is the number of hash functions. Retrieval Size measures the number of output layer neurons retrieved from hash tables. 81
5.4 This table summarizes the highest accuracy of *HALOS* on four dataset. Bold indicates that *HALOS* is higher than the full accuracy shown in Table 5.1. 81

6.1 This table summarizes the accuracy after training on original dataset, and after training on update dataset. The RIDDLE obtains comparable accuracy on original test set and update test set after trained on corresponding train set. The RIDDLE's accuracy is significantly higher than all baselines on original test set after updating.100
6.2 This table displays the effect of the partition-dependent learning rate. "Global Learning Rate" means that the learning rate is not related to parameter counts. 103
6.3 This table summarizes the accuracy comparison. NN1 and NN2 denote two different deep learning models. RIDDLE achieves higher accuracy than at least half of the baselines on all datasets. 103

6.4 This table summarizes the efficiency comparison between the RIDDLE and a neural network (NN) with similar accuracy. The RIDDLE reduces memory by up to 3.7x, FLOPs by up to 17x and inference time by up to 3.9x. 104

Chapter 1

Introduction

The recent success of machine learning excites the community to tackle ever more challenging tasks and starts a trend of increasing model capacity for unprecedented performance. In natural language processing, practitioners discovered the scaling law; models equipped with more parameters and trained on more extensive datasets often outperform the smaller ones. One exemplary observation on the continuum of Generative Pretrained Transformers(GPTs) models is that with each increase in model size, there is a consistent trend of setting new performance records.

However, the fast growth in model size comes with significant requirements for computing resources. Training a large-scale model requires a stunning number of FLOPs of computing, which not only translates to a long training time but also challenging requirements on the computing system. Continuing with the example on GPT, it is estimated that training the GPT-3 model of 175 billion parameters requires 3.14E23 FLOPS of computing, which will take 355 GPU-year for V100 at its theoretical computing power 28 TFLOPs. And 175 billion parameters are too large to store in a single GPU memory, and along with the computing requirement, system optimization such as a mixture of parallelism is a must. Further, running inference with large-scale models has its own challenges. To meet the application requirements of low latency for a pleasant user experience and scalability for handling a large number of simultaneous application requests, deployment incurs both high upfront hardware costs and high ongoing operation expenses. For example, electricity

itself is considerable: ChatGPT's daily electricity consumption is roughly around 15,000 US households.

This thesis is dedicated to mitigating the computational demands associated with large-scale ML models. We recognize dynamic sparsity as a promising avenue for exploration in this endeavor. In the context of machine learning, sparsity typically characterizes entities such as parameters, where the majority of elements are zero. Sparsity has been leveraged for efficiency purposes in previous works, and one such research direction is pruning, which exploits parameter sparsity and sets neurons or connections in a neural network to zero. This thesis focuses on a specific type of sparsity, namely, dynamic sparsity. We define dynamic sparsity to describe the evolving nature of sparsity patterns, specifically highlighting that the distribution of zero elements is not fixed but rather adjusted based on the current data. The analogy is obvious to draw. Each part of our brain is reactive to certain stimulations only. Dynamic sparsity offers the opportunity to preserve the same model capacity while allocating computing resources to the sub-part of the network that is crucial to the current data.

This thesis investigates the presence of dynamic sparsity within a trained ML model and validates its potential for efficiency gains without touching the model weights or compromising accuracy. Our approach involves efficiently forecasting critical computations or activations for each input, thus reducing the computational load by executing only essential computations or conserving memory by storing only critical activations. Additionally, we delve into the advantages of dynamic sparsity during the training phase, illustrating its impact on mitigating catastrophic forgetting.

1.1 Contribution

1.1.1 Contextual Sparsity for Efficient LLMs at Inference Time

Large language models (LLMs) with hundreds of billions of parameters have sparked a new wave of exciting AI applications. However, they are computationally expensive at inference time. Sparsity is a natural approach to reduce this cost, but existing methods either require costly retraining, have to forgo LLM's in-context learning ability, or do not yield wall-clock time speedup on modern hardware. We hypothesize that contextual sparsity, which are small, input-dependent sets of attention heads and MLP parameters that yield approximately the same output as the dense model for a given input, can address these issues. We show that contextual sparsity exists, that it can be accurately predicted, and that we can exploit it to speed up LLM inference in wall-clock time without compromising LLM's quality or in-context learning ability. Based on these insights, we propose DEJAVU, a system that uses a low-cost algorithm to predict contextual sparsity on the fly given inputs to each layer, along with an asynchronous and hardware-aware implementation that speeds up LLM inference. We validate that DEJAVU can reduce the inference latency of OPT-175B by over 2× compared to the state-of-the-art FasterTransformer and over 6× compared to the widely used HuggingFace implementation without compromising model quality.

1.1.2 Exploiting the Persistence of Importance Hypothesis for KV Cache Compression at Inference Time

Hosting LLMs at scale requires significant memory resources. One crucial memory bottleneck for the deployment stems from the context window. It is commonly recognized that model weights are memory-hungry; however, the size of key-value

embedding stored during the generation process (KV cache) can easily surpass the model size. The enormous size of the KV cache puts constraints on the inference batch size, which is crucial for a high throughput inference workload. Previous methods have shown that the attention mechanism is sparse during training. A large portion of tokens receive attention scores close to zeros, and only a few tokens would sum up to approximately the full attention score.

Inspired by an interesting observation of the attention scores, we hypothesize the persistence of importance: only pivotal tokens, which had a substantial influence at one step, will significantly influence future generations. Based on our empirical verification and theoretical analysis around this hypothesis, we propose Scissorhands, a system that maintains the memory usage of the KV cache at a fixed budget without finetuning the model. In essence, Scissorhands manages the KV cache by storing the pivotal tokens with a higher probability, resulting in a sparse KV cache adjusted based on the inputs. We validate that Scissorhands reduces the inference memory usage of the KV cache by up to 5X without compromising model quality. We further demonstrate that Scissorhands can be combined with 4-bit quantization, traditionally used to compress model weights, to achieve up to 20X compression.

1.1.3 Hashing Large Output Space for Cheap Inference

Efficient inference in large output space is an essential yet challenging task for large-scale ML models, including recommender systems and language models. Previous approaches reduce this problem to Approximate Maximum Inner Product Search (AMIPS) and focus the computation only on the retrieved top candidate, which is based on the observation that the prediction of a given model corresponds to the logit with the largest value. However, models are not perfect in accuracy, and the successful

retrievals of the largest logit may not lead to the correct predictions. We argue that approximate MIPS approaches are sub-optimal because they are tailored for retrieving largest inner products class instead of retrieving the correct class. Moreover, the logits generated from neural networks with large output space lead to extra challenges for the AMIPS method to achieve a high recall rate within the computation budget of efficient inference. In this paper, we propose HALOS, which reduces inference into sub-linear computation by selectively activating a small set of output layer neurons that are likely to correspond to the correct classes rather than to yield the largest logit. Our extensive evaluations show that HALOS matches or even outperforms the accuracy of given models with $21\times$ speed up and 87% energy reduction.

1.1.4 Retaining Knowledge for Learning with Dynamic Definition

Machine learning models are often deployed in settings where they must be constantly updated in response to the changes in class definitions while retaining high accuracy on previously learned definitions. A classical use case is fraud detection, where new fraud schemes come one after another. While such an update can be accomplished by re-training on the complete data, the process is inefficient and prevents real-time and on-device learning. On the other hand, efficient methods that incrementally learn from new data often result in the forgetting of previouslylearned knowledge. We define this problem as Learning with Dynamic Definition (LDD) and demonstrate that popular models, such as the Vision Transformer and Roberta, exhibit substantial forgetting of past definitions. We present a first practical and provable solution to LDD. Our proposal is a hash-based sparsity model RIDDLE that solves evolving definitions by associating samples only to relevant parameters. We prove that our model is a universal function approximator and theoretically bounds the knowledge lost during

the update process. On practical tasks with evolving class definition in vision and natural language processing, RIDDLE outperforms baselines by up to 30% on the original dataset while providing competitive accuracy on the update dataset.

Chapter 2

Background

2.1 Locality Sensitive Hashing

In this section, we first provide a review of Locality Sensitive Hashing(LSH) functions, a essential technique in this thesis. An LSH family is a set of functions that maps similar inputs to the same hash value (with high probability). LSH is an incredibly powerful and well-studied framework, with the following formal definition [1, 2, 3, 4].

Definition 1 (**(S_0, cS_0, p_1, p_2)-sensitive hash family).** *A family $\mathcal{H}$ is called (S_0, cS_0, p_1, p_2)-sensitive with respect to a similarity function $sim(\cdot, \cdot)$ if for any two points $x, y \in \mathbb{R}^D$ and h chosen uniformly from $\mathcal{H}$ satisfies:*

- *If $sim(x, y) \geq S_0$ then $Pr[h(x) = h(y)] \geq p_1$*
- *If $sim(x, y) \leq cS_0$ then $Pr[h(x) = h(y)] \leq p_2$*

The probability $Pr[h(x) = h(y)]$ is known as the collision probability of x and y, where we use the term “collision” to mean that two points map to the same hash value. For the purpose of our arguments, we use a stronger notion of LSH, in which the collision probability is a monotonic increasing function of the similarity between x and y. That is $Pr[h(x) = h(y)] \propto f(\text{sim}(x, y))$.

Under the above conditions, the collision probability $Pr[h(x) = h(y)]$ forms a positive semi-definite radial kernel [5]. A number of useful kernels can be obtained via

LSH, including MinHash for the Jaccard kernel [6], signed random projections for the angular kernel [7], and p-stable LSH for a kernel in Euclidean space [3].

Recent works have shown that LSH can be used for efficient statistical estimation [8, 9, 10, 5, 11, 12, 13, 14], compressive machine learning [15], nearest neighbor search [16, 2, 17] and to accelerate neural network training and inference [18, 19, 20, 21, 22].

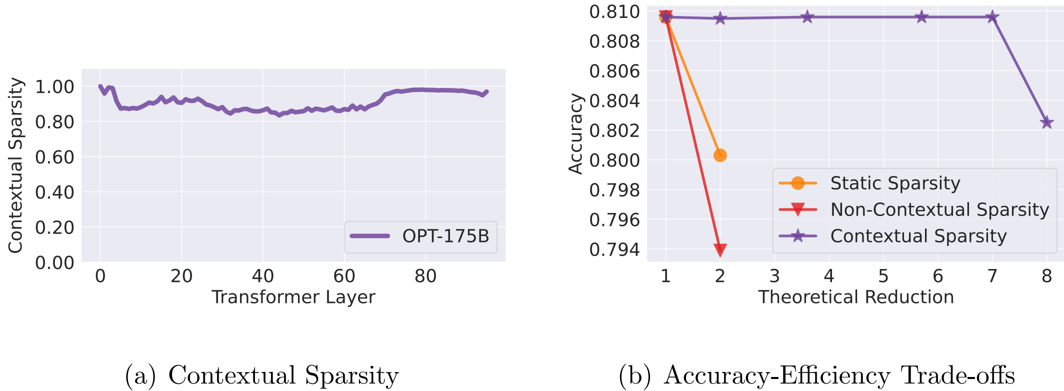
Chapter 3

Contextual Sparsity for Efficient LLMs at Inference Time

3.1 Introduction

Large language models (LLMs), such as GPT-3, PaLM, and OPT have demonstrated that an immense number of parameters unleashes impressive performance and emergent in-context-learning abilities—they can perform a task by conditioning on input-output examples, without updating their parameters [23, 24, 25, 26, 27]. However, they are very expensive at inference time, especially for latency-sensitive applications [28]. An ideal inference-time model should use less computation and memory while maintaining the performance and special abilities of pre-trained LLMs. The simplest and most natural approach is sparsification or pruning, which has a long history before the LLM era [29]. Unfortunately, speeding up inference-time sparse LLMs in wall-clock time while maintaining quality and in-context learning abilities remains a challenging problem.

While sparsity and pruning have been well-studied, they have not seen wide adoption on LLMs due to the poor quality and efficiency trade-offs on modern hardware such as GPUs. First, it is infeasible to retrain or iteratively prune models at the scale of hundreds of billions of parameters. Thus, methods in iterative pruning and lottery ticket hypothesis [30, 31] can only be applied to smaller-scale models. Second, it is challenging to find sparsity that preserves the in-context learning ability



(a) Contextual Sparsity

(b) Accuracy-Efficiency Trade-offs

Figure 3.1 : (1) LLMs have up to 85% contextual sparsity for a given input. (2) Contextual sparsity has much better efficiency-accuracy trade-offs (up to 7×) than non-contextual sparsity or static sparsity.

of LLMs. Many works have shown the effectiveness of task-dependent pruning [32, 33], but maintaining different models for each task conflicts with the task independence goal of LLMs. Lastly, it is hard to achieve wall-clock time speed-up with unstructured sparsity due to its well-known difficulty with modern hardware [34]. For example, recent development in zero-shot pruning like SparseGPT [35] finds 60% unstructured sparsity but does not yet lead to any wall-clock time speedup.

An ideal sparsity for LLMs should (i) not require model retraining, (ii) preserve quality and in-context learning ability, and (iii) lead to speed-up in wall-clock time on modern hardware. To achieve such demanding requirements, we go beyond *static* sparsity in previous works (e.g., structured/unstructured weight pruning). We instead envision *contextual sparsity*, which are small, input-dependent sets of attention heads and MLP parameters that lead to (approximately) the same output as the full model for an input. Inspired by the connections between LLMs, Hidden Markov Models [36, 37], and the classic Viterbi algorithm [38], we hypothesize that for pre-trained LLMs,

contextual sparsity exists given any input.

The hypothesis, if true, would enable us to cut off specific attention heads and MLP parameters (structured sparsity) on the fly for inference-time, without modifying pre-trained models.

However, there are three challenges.

Existence: It is nontrivial to verify if such contextual sparsity exists, and naive verification can be prohibitively expensive.

Prediction: Even if contextual sparsity exists, it is challenging to predict the sparsity for a given input in advance. *Efficiency*: Even if the sparsity can be predicted, it might be difficult to achieve end-to-end wall-clock time speedup. Taking OPT-175B as an example, the latency of one MLP block is only 0.2 ms on an 8×A100 80GB machine. Without a fast prediction and optimized implementation, the overhead can easily increase the LLM latency rather than reduce it.

In this work, we address these challenges as follows:

Existence: Fortunately, we verify the existence of contextual sparsity with a surprisingly simple approach. To achieve essentially the same output, contextual sparsity is on average *85%* structured sparse and thereby potentially leads to a 7× parameter reduction for each specific input while maintaining accuracy (Figure 3.1(a)). During explorations of contextual sparsity, we make important empirical observations and build a theoretical understanding of major components in LLMs that help address the prediction and efficiency challenge.

Prediction: We discover that contextual sparsity depends not only on individual input tokens (i.e., *non-contextual dynamic* sparsity) but also on their interactions (*contextual dynamic* sparsity). Figure 3.1(b) shows that with pure dynamic information, sparsity prediction is inaccurate. Only with token embeddings with sufficient contextual

information can we predict sparsity accurately. Another finding is that *contextual dynamic* sparsity for every layer can be predicted based on the "similarity" between layer parameters (heads/MLP) and the output from the previous layer, which carries the immediate contextual mixture of token embeddings.

Efficiency: Because at inference time, model parameters are static, inspired by the classical nearest neighbor search (NNS) literature and its applications in efficient deep learning, it is possible to formulate the above similarity-based prediction as an NNS problem [39, 40, 41]. However, as mentioned, the overhead might be difficult to overcome as we would need to perform on-the-fly predictions before every layer. Luckily, we exploit a phenomenon of LLM where token embeddings change slowly across layers due to residual connections (well-known in computer vision [42]). Since the inputs to a few consecutive layers are very similar, we can design an asynchronous lookahead predictor (Figure 3.2).

Based on our findings, we present a system, DEJAVU, that exploits contextual sparsity and realizes efficient LLMs for latency-sensitive applications.

- In Section 3.4.1 and Section 3.4.2, we present a low-cost learning-based algorithm to predict sparsity on the fly. Given the input to a specific layer, it predicts a relevant subset of attention (heads) or MLP parameters in the next layer and only loads them for the computation.

- In Section 3.4.4, we propose an asynchronous predictor (similar to classic branch predictor [43]) to avoid the sequential overhead. A theoretical guarantee justifies that the cross-layer design suffices for accurate sparsity prediction.

After integrating hardware-aware implementation of sparse matrix multiply (Section 3.4.5), DEJAVU (written mostly in Python) can reduce latency of open-source

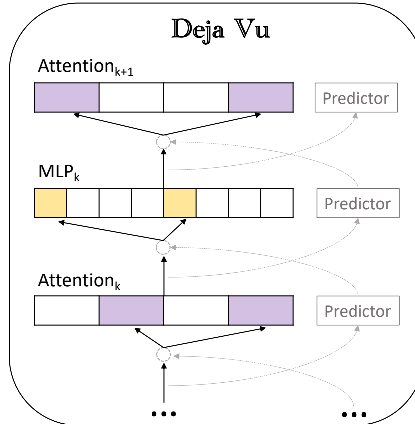


Figure 3.2 : DEJAVU uses lookahead predictors to side-step prediction costs: given the input to the attention layer at block k, they (asynchronously) predict the contextual sparsity for the MLP at block k, and given the input to the MLP at block k, they predict the sparsity for the attention head at the next layer.

LLMs such as OPT-175B by over 2× end-to-end without quality degradation compared to the state-of-the-art library FasterTransformer from Nvidia (written entirely in C++/CUDA), and over 2× compared to the widely used Hugging Face implementation at small batch sizes. Furthermore, we show several ablations on different components of DEJAVU and its compatibility with quantization techniques.

3.2 Related Work and Problem Formulation

We introduce the latency breakdown for LLM inference, and then we provide a formal problem formulation.

3.2.1 LLM Inference Latency Breakdown

The generative procedure of LLMs consists of two phases: (i) the *prompt* phase takes an input sequence to generate the keys and values (KV cache) for each transformer block of LLMs, which is similar to the forwarding pass of LLMs training; and (ii) the *token generation* phase utilizes and updates the KV cache to generate tokens step by step, where the current token generation depends on previously generated tokens.

This paper studies the setting where the token generation phase easily dominates the end-to-end inference time. As shown in Table 3.1, generating a sequence of length 128 takes much longer time than processing a sequence of length 128 as prompt due to I/O latency of loading model parameters. In addition, Table 3.2 shows that attention and MLP are both bottlenecks in LLMs, e.g., in 175B models, loading MLP parameters takes around $\frac{2}{3}$ of the total I/O and attention heads take the other $\frac{1}{3}$. Further, in the tensor-parallel regime, there are two communications between GPUs, one after the attention block, and the other one after the MLP block. As shown in Table 3.3, communication between GPUs takes around 15 % token generation latency. This paper focuses on making attention and MLP more efficient. Communication cost implies that the upper bound of such speed-up is around 6× when skipping all transformer blocks.

Table 3.1 : Theoretical breakdown for prompting versus token generation (tensor model parallelism on 8 A100-80G GPUs).

	TFLOPs	I/O	Compute Latency (ms)	I/O Latency (ms)
Prompting 128	44.6	330 GB	17.87	20.6
Token Generation 128	44.6	41 TB	17.87	2600

Table 3.2 : Theoretical breakdown for Attention block versus MLP block in one transformer layer when generating one token (tensor model parallelism on 8 A100-80G GPUs).

	GFLOPs	I/O (GB)	Compute Latency (ms)	I/O Latency (ms)
Attention Block	1.21	1.12	0.00048	0.07
MLP Block	2.41	2.25	0.00096	0.14

Table 3.3 : Latency breakdown of generating 1 token under the setting of batch size 1 and prompt length 128 on 8 A100-80GB.

All Reduce	MLP Block	Attention Block (ms)	Others
6 ms	19ms	13ms	2ms

3.2.2 Problem Formulation

The goal is to reduce the generation latency of LLMs by exploiting contextual sparsity. In the following, we formally define the sparsified attention and MLP blocks.

Sparsified MLP: There are two linear layers in one MLP block, $W^1, W^2 \in \mathbb{R}^{d \times 4d}$. Denote $y \in \mathbb{R}^{1 \times d}$ as the input to the MLP block in the current generation step. Let each column (the weight of i-th neuron) of linear layers be $W^1_i, W^2_i \in \mathbb{R}^{d \times 1}$. With contextual sparsity, only a small set of them are required for computation. Let $S_M \subseteq [4d]$ denote such set of neurons for input y. The sparsified MLP computation is

$$\mathsf{MLP}_{S_M}(y) = \sigma(yW^1_{S_M})(W^2_{S_M})^\top, \tag{3.1}$$

where σ is the activation function, e.g., ReLU, GeLU. Note that since the computation in the first linear results in sparse activations, the second linear layer is also sparsified.

Sparsified Attention: Let $X \in \mathbb{R}^{n\times d}$ denote the embeddings of all tokens (e.g., prompts and previously generated tokens). Let $y \in \mathbb{R}^{1\times d}$ be the input to the Multi-Head-Attention (MHA) in the current generation step. Suppose there are h heads. For each $i \in [h]$, we use $W_i^K, W_i^Q, W_i^V \in \mathbb{R}^{d\times d_h}$ to denote key, query, value projections for the i-th head, and $W_i^O \in \mathbb{R}^{d_h\times d}$ for output projections. With contextual sparsity, we denote S_A as a small set of attention heads leading to approximately the same output as the full attention for input y. Following the notation system in [44], sparsified MHA computation can be formally written as

$$\mathsf{MHA}_{S_A}(y) = \sum_{i\in S_A} \underbrace{H_i(y)}_{1\times d_h} \underbrace{W_i^O}_{d_h\times d},$$

where $H_i(y) : \mathbb{R}^d \to \mathbb{R}^{d_h}$ and $D_i(y) \in \mathbb{R}$ can be written as

$$H_i(y) := D_i(y)^{-1} \exp(yW_i^Q (W_i^K)^\top X^\top) X W_i^V, \tag{3.2}$$

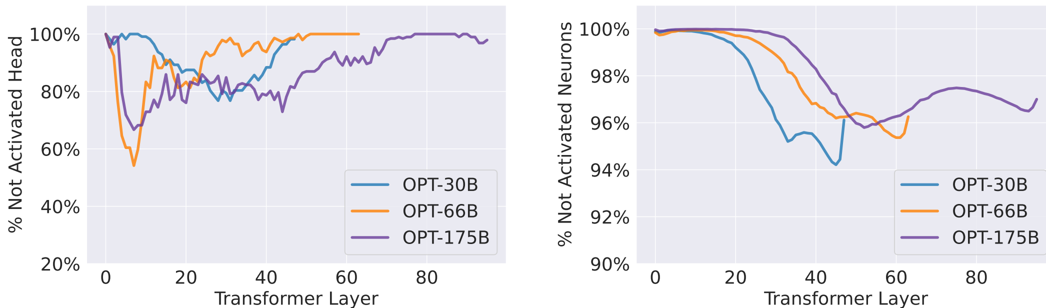
$$D_i(y) := \exp(yW_i^Q (W_i^K)^\top X^\top) \mathbf{1}_n.$$

For both MLP and Attention, given a compute budget, the goal is to find S_M and S_A that minimize the error between the sparse approximation and full computation.

3.3 Pre-trained LLMs are Contextually Sparse

In this section, we present several key observations and theoretical understandings of sparsity in LLMs, upon which the DEJAVU design is based. We first test the contextual sparsity hypothesis and verify that contextual sparsity exists in pre-trained LLMs in

Section 3.3.1. Then, we build an understanding of why contextual sparsity happens naturally even when LLMs are densely trained in Section 3.3.2. Finally, we present an observation on residual connections and explain their relationship to contextual sparsity analytically in Section 3.3.3.



(a) Contextual sparsity in Attention Head (b) Contextual sparsity in MLP Block

Figure 3.3 : In Figure (a), we plot the percentage of not-activated attention heads. By only keeping heads that yield large output norms, we can silence over 80% attention heads for a given token. In Figure (b), we plot the average sparsity we impose on MLP layers. We can zero out over 95% of MLP parameters for a given token.

3.3.1 Contextual Sparsity Hypothesis

Inspired by prior pruning literature [45], we find a surprisingly simple method is sufficient to study and verify our hypothesis. In this section, we describe the testing procedure, observation details, and insights of this study.

Verification: Our test is performed on OPT-175B, 66B, and 30B models and various downstream datasets such as OpenBookQA [46] and Wiki-Text [47]. We find the contextual sparsity for every input example with two forward passes of the model. In the first pass, we record a subset of parameters, specifically which attention

heads and MLP neurons yield large output norms for the input. In the second pass, each input example only uses the recorded subset of parameters for the computation. Surprisingly, these two forward passes lead to similar prediction or performance on all in-context learning and language modeling tasks.

Observation: Figure 3.3 shows that on average, we can impose up to 80% sparsity on attention heads and 95% sparsity on MLP neurons. As mentioned in Section 3.2, OPT-175B model has $2\times$ MLP parameters than those of attention blocks. Therefore total sparsity here is around 85%. Since these are all structured sparsity (heads and neurons), predicting them accurately could potentially lead to $7\times$ speedup.

Insight: It is intuitive that we can find contextual sparsity in MLP blocks at inference time because of their activation functions, e.g., ReLU or GeLU [48]. Similar observations were made by [49]. However, it is surprising that we can find contextual sparsity in attention layers. Note that, finding contextual sparsity in attention is not the same as head pruning. We cross-check that different examples have different contextual sparsity. Although 80% of the parameters are not included in the paths for a given example, they might be used by other examples. Next, we will try to understand why contextual sparsity exists in attention blocks.

3.3.2 Token Clustering in Attention Layers

In the previous section, we have verified that there exists contextual sparsity for a given input in LLMs. In this section, we try to understand the reason for such phenomena, especially in attention layers. We first show an in-depth observation of attention. Then we present a hypothesis that self-attentions are conceptually clustering algorithms. Last we show analytical evidence to support this hypothesis.

Observation: Figure 3.4 shows the attention map of three different heads from

the same layer for an example input. The next token it should predict is "Truck". Darker color represents higher attention scores. We observe that the middle head is a relatively uniform token-mixing head while the top and bottom ones are "heavy hitter" attention heads (with high attention to "like" and "shipping"). Unsurprisingly, only selecting heavy hitter heads but not uniform heads does not affect the prediction, since uniform heads do not model or encode important token interactions. In the next section, we will also explain in detail how the criteria for selecting uniform attention heads and heads with small output norms are highly correlated.

Hypothesis: We hypothesize that the attention head is performing mean-shift clustering [50].

Recall the notation defined in Section 3.2.2. For i-th head at current layer, $X = [x_1, \ldots, x_n]^\top \in \mathbb{R}^{n\times d}$ are the token embeddings in the previous time steps. XW_i^K and XW_i^V are the projection of embedding. For an input embedding y, the output $\widetilde{y}_i = H_i(y)$, where $H_i(y)$ is defined in Eq. 3.2.

For each $i \in [h]$, if we let $K_i(x_j, y) := \exp(yW_i^Q(W_i^K)^\top x_j)$ measure the similarity between x_j and y, and define $m_i(y) := \frac{\sum_j K_i(x_j,y)x_j}{\sum_j K_i(x_j,y)}$, then we have $\widetilde{y}_i = m_i(y)W_i^V$. Further, if we set $W_i^V = I$ and consider the residue connection followed by layer norm, then in the next layer, the embedding $\widehat{y}_i$ of the current token becomes $\widehat{y}_i = \text{Normalize}(y + \widetilde{y}_i) = \text{Normalize}(y + m_i(y))$, which has a fixed point $y = \gamma m_i(y)$ for any scalar γ. This iteration bears a resemblance to mean-shift clustering, which simply performs iteration $y \leftarrow m_i(y)$ until convergence. This has an obvious fixed point $y = m_i(y)$.

Therefore, the self-attention head can be regarded as *one mean-shift step* to push input embeddings of different tokens together, if they are already neighbors in a projection space specified by $W_i^Q(W_i^K)^\top$. Different heads learn different projection

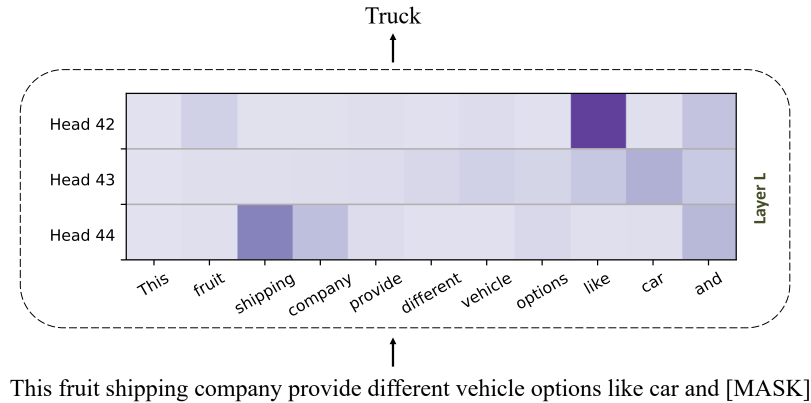


Figure 3.4 : We visualize the attention scores of three different heads for an exemplary sentence. Head 42 and Head 44 give heavy attention scores on particular tokens while Head 43 is more uniform.

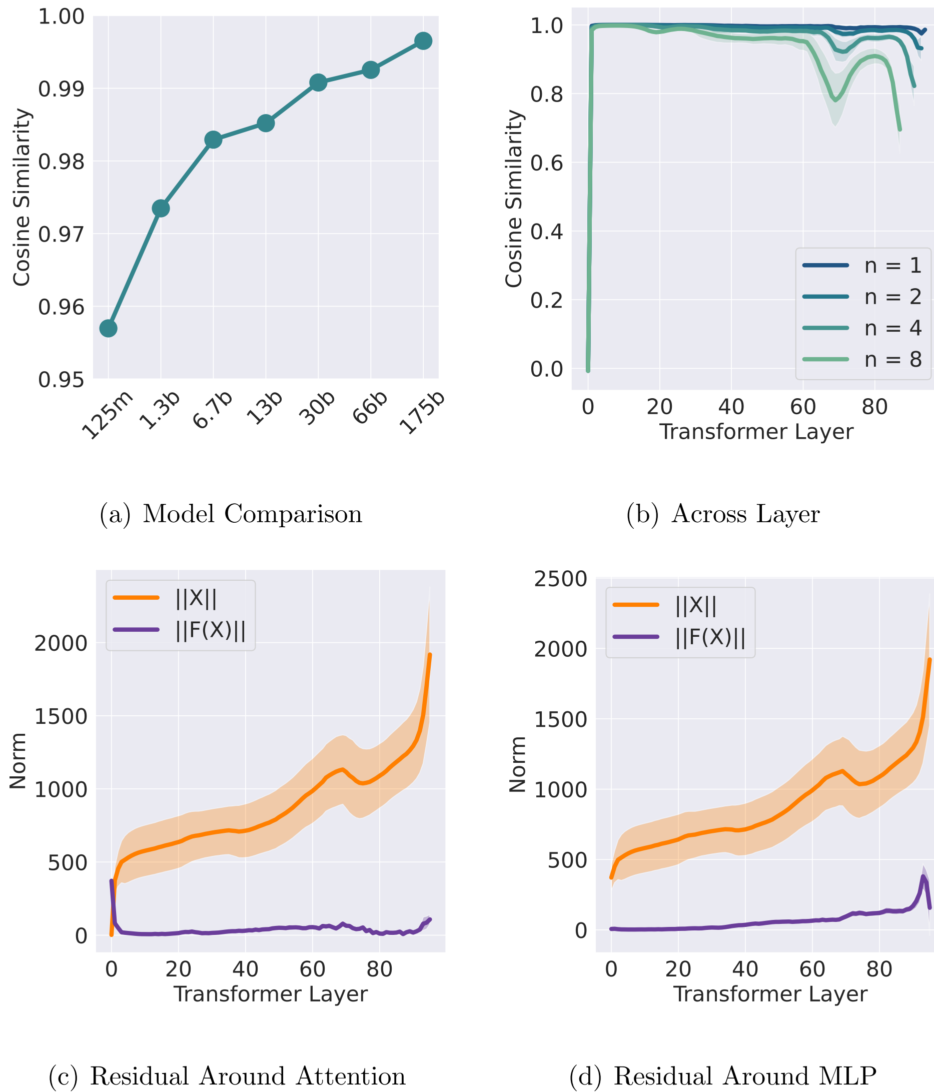
spaces to perform clustering. These dynamics explain the precise reason why token embeddings tend to cluster after going through more layers, resulting in high attention scores among cluster members, and low scores for non-members. Furthermore, the cluster patterns are different at different heads.

The above analysis not only provides an understanding of why contextual sparsity exists naturally in pre-trained LLMs, but also inspires our design of "similarity"-based sparsity prediction for DEJAVU in Section 3.4.

3.3.3 Slowly Changing Embeddings across Layers

We first present our observation that embeddings change slowly across consecutive layers. Then we provide a detailed analysis on the phenomenon. Finally, we show its close connection with contextual sparsity.

High similar embeddings in consecutive layers: In Figure 3.5(a), we show that for the same given input, the cosine similarity between embeddings or activations



(a) Model Comparison

(b) Across Layer

(c) Residual Around Attention

(d) Residual Around MLP

Figure 3.5 : **Slowly Changing Embedding.** Figure (a) shows the median cosine similarity between representations at two consecutive layers across all layers for different OPT models. All models show a similarity greater than 95%. Figure (b) shows cosine similarity stays high even a few layers apart. For the residual connection $X' = X + F(X)$ inside each block, we plot the ℓ_2 norm of X and $F(X)$ in Figure (c) and Figure (d). $\|X\|$ is significantly higher than $\|F(X)\|$, which explains the slowly changing embedding.

in two consecutive layers is exceptionally high on 7 different sizes of OPT models. Specifically, we collect activations from each layer while performing OPT model inference on C4 validation set [51]. Taking OPT-175B as an example, starting from the second layer, the similarity between any two consecutive layers is around 0.99, which indicates that when an input is passed through the model, the direction of its embedding changes slowly. Interestingly, the most drastic change happens in the first layer. Furthermore, we increase the gap and investigate the similarity between the embedding at layer l and at layer $l+n$ shown in Figure 3.5(b). As we increase the gap, the similarity decreases as expected while the differences in cosine similarity between various choices of n are smaller at the shallower layer. We plot the mean similarity, and the standard deviation is indicated by the shading.

Connection to residuals: We verify that the high similarity in embeddings in LLM inference is due to the residual connection. We first dissect the computation graph inside each transformer layer to understand the cause behind this phenomenon. There are two residual connections inside a transformer layer, one around the attention block, and the other one around the MLP block. The residual connection can be written as $X + F(X)$, where F is either the Multi-Head Attention or two MLP Layers. In Figure 3.5(c) and Figure 3.5(d), indeed we can see that $\|X\|$ is significantly greater than $\|F(X)\|$, confirming that embeddings are changing slowly because the residual norm is large.

Connection to Contextual Sparsity: We take a step deeper trying to understand the reason behind the large residual norm with mathematical modeling. We discover that one possible reason for small $\|F(X)\|$ is due to high sparsity. For the MLP Block, high sparsity may contribute to the small norm of $F(X)$ because a large portion of outputs have small norms. Similar reasoning applies to the Attention Block,

and thus a large number of attention heads yield small norm outputs.

Residual Two Sides Bound: Besides empirical reasoning, we formally define the computation of LLMs mathematically. Under our computation model, we can show that a shrinking property which is observed by our practical experiments.

Lemma 1 (Informal)**.** *Let $0 < \epsilon_1 < \epsilon_2 < 1$ be the lower and upper bound of the shrinking factor. Let x be the y be the output. We have the residual connection $y = x + F(x)$. For the MLP block $F(x)$, we have $\epsilon_1 \leq \|y - x\|_2 \leq \epsilon_2$. For the attention block $F(x)$, we have $\epsilon_1 \leq \|y - x\|_2 \leq \epsilon_2$.*

3.4 DeJavu

In this section, we present our framework for inference-time contextual sparsity search for LLMs. We introduce the sparsity predictor for MLPs in Section 3.4.1 and for attention heads in Section 3.4.2. DEJAVU's workflow is shown in Figure 3.2. Section 3.4.4 discusses exploiting our observation on LLMs to avoid the sparse prediction overhead with theoretical guarantees. In Section 3.4.5, we present our optimized implementation that enables end-to-end latency reduction.

3.4.1 Contextual Sparsity Prediction in MLP Blocks

As explained in Section 3.2, MLP blocks are one of the major bottlenecks for the LLM generation ($\frac{2}{3}$ of the FLOPs and IOs). In this section, we discuss how we achieve wall-clock time speed-up with contextual sparsity in the MLP blocks.

Challenge Figure 3.3(b) shows that for a given token, the contextual sparsity of 95% is possible. The contextual sparsity in the MLP block can be identified after computing the activation. However, this only demonstrates the existence of contextual sparsity but brings no benefits in terms of efficiency. A fast and precise prediction is

needed to exploit contextual sparsity for end-to-end efficiency. The naive way is to select a subset of neurons randomly. Unsurprisingly, random selection fails to identify the accurate contextual sparsity, resulting in drastic model degradation.

A Near-Neighbor Search Problem: Recall that we verify the existence of contextual sparsity by recording which neurons yield significant norms. Essentially, given the input, the goal is to search for the neurons that have high inner products with the input, because the activation function "filters" low activation. Thus, we formulate the contextual sparsity prediction of an MLP layer as the classical near-neighbor search problem under the inner product metric.

Definition 2 (Approximate MaxIP in MLP)**.** *Let $c \in (0,1)$ and $\tau \in (0,1)$ denote two parameters. Given an n-vector dataset $W^1 \subset \mathbb{S}^{d-1}$ on a unit sphere, the objective of the (c,τ)-MaxIP is to construct a data structure that, given a query $y \in \mathbb{S}^{d-1}$ such that $\max_{w \in W^1} \langle y, w \rangle \geq \tau$, it retrieves a vector z from W^1 that satisfies $\langle y, z \rangle \geq c \cdot \max_{w \in W^1} \langle y, w \rangle$.*

Remark 2. *Our W^1 (first linear layer) and y (input embedding) in MLP blocks can be viewed as the dataset and query in Definition 2 respectively.*

Design The standard state-of-the-art near-neighbor search methods and implementations slow down the computation. Take OPT-175B where d is 12288 as an example. HNSW [52] requires more than 10ms, and FAISS [53] requires more than 4ms, while the MLP computation is only 0.2ms. The high dimensionality and complications of data structure implementation on GPU make the search time longer than the MLP computation. Therefore, we choose a neural network classifier as our near-neighbor search method to exploit the fast matrix multiplication on GPU. For each MLP block, we train a small two-layer fully connected network to predict contextual sparsity.

Collecting training data is straightforward because we know the contextual sparsity using dense computation. The training algorithm is summarized in Algorithm **??**. The sparsified computation in W^1 has two steps: (1) Given y, the sparsity predictor SP_M predicts a set S_M of important neurons in weights W^1. (2) Compute the sparsified MLP defined in Eq. (3.1). Note here the sparsity in MLP is highly structured.

3.4.2 Contextual Sparsity Prediction in Attention Blocks

Attention blocks take around 30% I/Os in the generation. In this section, we describe how DEJAVU exploits contextual sparsity to speed up the Attention blocks.

Challenge: As discussed in Section 3.3.1, only a few heads perform important computations for a given input token. Similar to the MLP blocks, a fast selection of attention heads without full computation is required to reduce end-to-end latency. Furthermore, one particular challenge of sparse prediction in attention blocks is attention's dependence on previous tokens. On the one hand, it is unclear whether the past token's key and value caches are needed for sparse prediction. On the other hand, it is unclear how to handle the missing KV cache of past tokens for the current token computation at the selected head.

A Near-Neighbor Search Problem: Head prediction can also be formulated as a near-neighbor search problem based on our understanding in Section 3.3.2. Since each head is performing mean-shift clustering, after the first few layers, the current token embedding alone is sufficient for the prediction thanks to the token-mixing nature of the transformer. Therefore, the prediction can be based on the similarity between y and head parameters.

Approach: We design our attention sparse predictor to be the same architecture as the MLP sparse predictor. Each head is regarded as one class and a similar training

process is used. Then, similar to how MLP prediction is performed, the attention sparsity predictor SP_A selects a set S_A of heads H_i (see Eq. (3.2)). To address the problem of missing KV cache for a past token, we exploit the fact that the generation latency is I/O bounded while computation is essentially "free". Specifically, for the predicted attention head of input y, we compute the corresponding keys, and values and store them in the KV cache. But we also save a copy of y for all the other non-selected heads. Then during the future token generation, if there is missing KV cache in the selected heads, we could load stored token embeddings and compute the keys and values together. This requires almost minimal extra memory access (the main cost is loading the weight matrices).

3.4.3 Detailed workflow

Figure 3.6 presents a more detailed workflow of DEJAVU. The left diagram shows how an input y performs the sparse MHA with selected indices $0, 3$, predicted by the head predictor. Similarly, the right diagram shows how an input y performs the sparse MLP with selected indices $0, 2$, predicted by the neuron predictor of that layer.

3.4.4 Reducing Overhead with Asynchronous Execution

Sparse prediction overhead may easily increase the end-to-end latency rather than reduce it despite the reduction in FLOPs. Therefore, we introduce a look-ahead sparse prediction method, inspired by our observations in Section 3.3.3.

Challenge: Denote $y_l \in \mathbb{R}^d$ as the input to transformer layer l. We can write the computation at layer l as $\widetilde{y}_l \leftarrow \mathsf{MHA}^l(y_l), \widehat{y}_l \leftarrow \mathsf{MLP}^l(\widetilde{y}_l)$. With predictors SP_A^l and

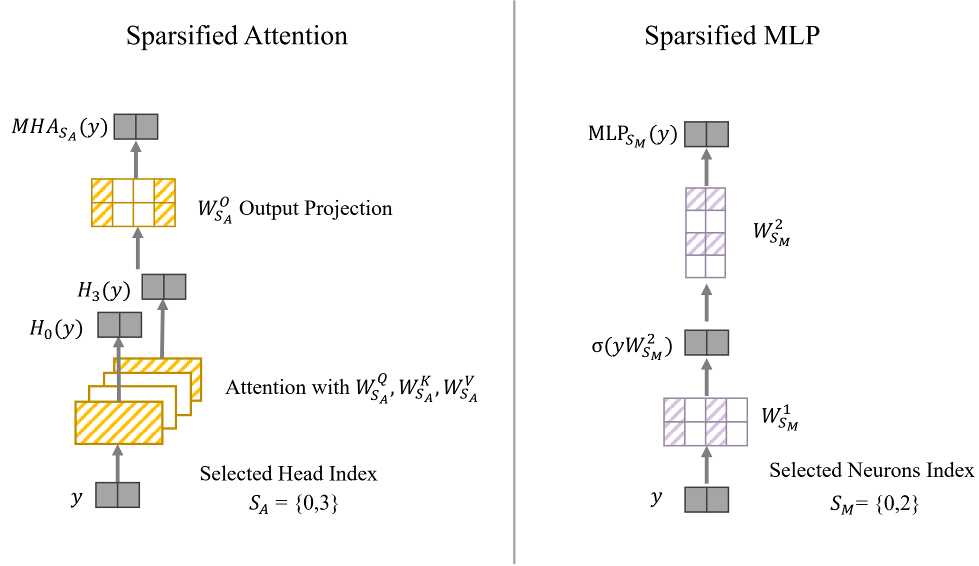


Figure 3.6 : Detailed diagram on the sparsified computation process of MLP and Attention. Notation refers to Section 3.2.2

SP^l_M, the computation at the transformer layer l can be re-written as

$$S^l_A \leftarrow \mathsf{SP}^l_A(y_l), \quad \widetilde{y}_l \leftarrow \mathsf{MHA}^l_{S^l_A}(y_l),$$
$$S^l_M \leftarrow \mathsf{SP}^l_M(\widetilde{y}_l), \quad \widehat{y}_l \leftarrow \mathsf{MLP}^l_{S^l_M}(\widetilde{y}_l)$$

where set S^l_A is the contextual sparsity for the Attention block, and set S^l_M is the contextual sparsity for the MLP block at l-th layer. Note that the computation at Attention and MLP blocks have to wait for the sparse predictor decision. This overhead potentially outweighs the saving from Attention and MLP blocks in terms of latency.

Approach: In Section 3.3.3, we present the slowly evolving embedding phenomenon, which provides opportunities to relax the sequential computation to parallel computation. Along with the observation of low computation intensity during generation, we parallel the sparse prediction with the computation of each block (See

Figure 3.2). The computation can be written as follows:

$$\widetilde{y}_l \leftarrow \mathsf{MHA}^l_{S^l_A}(y_l), \quad \widehat{y}_l \leftarrow \mathsf{MLP}^l_{S^l_M}(\widetilde{y}_l),$$
$$S^{l+1}_A \leftarrow \mathsf{SP}^l_A(y_l), \quad S^{l+1}_M \leftarrow \mathsf{SP}^l_M(y_l),$$

We remark S^{l+1}_A and S^{l+1}_M can be computed in parallel with $\widetilde{y}_l$ or $\widehat{y}_l$, while the previous 4 steps are sequential.

Theoretical guarantee: The sparse predictor can make further cross-layer decisions because of the residual connection. We present an informal lemma statement regarding cross-layer prediction. It is well-known that MaxIP is equivalent to ℓ_2 nearest neighbor search. For convenience, we use MaxIP here.

Lemma 3 (Informal)**.** *Let $\epsilon \in (0, 1)$. Let y_l be input at l-th layer. Let y_{l-1} be the input at $(l-1)$-th layer. Suppose that $\|y_l - y_{l-1}\|_2 \leq \epsilon$. For any parameters c, τ such that $\epsilon < O(c\tau)$. Then we can show that, solving* $\mathsf{MaxIP}(c, \tau)$ *is sufficient to solve* $\mathsf{MaxIP}(0.99c, \tau)$.

3.4.5 Hardware-efficient Implementation

We describe how DEJAVU is implemented in a hardware-efficient manner to realize the theoretical speedup of contextual sparsity. Taking into account hardware characteristics leads to over 2× speedup compared to an optimized dense model, and 4× faster than a standard sparse implementation.

We highlight some hardware characteristics of GPUs:

- Small-batch generation is bottlenecked by GPU memory I/Os [54, 55, 56]. This is because of low arithmetic intensity. For each element loaded from GPU memory, only a small number of floating point operations are performed.

- GPUs are block-oriented devices: loading a single byte of memory takes the same time as loading a block of memory around that same address [57]. The block size is usually 128 bytes for NVIDIA GPUs [58].

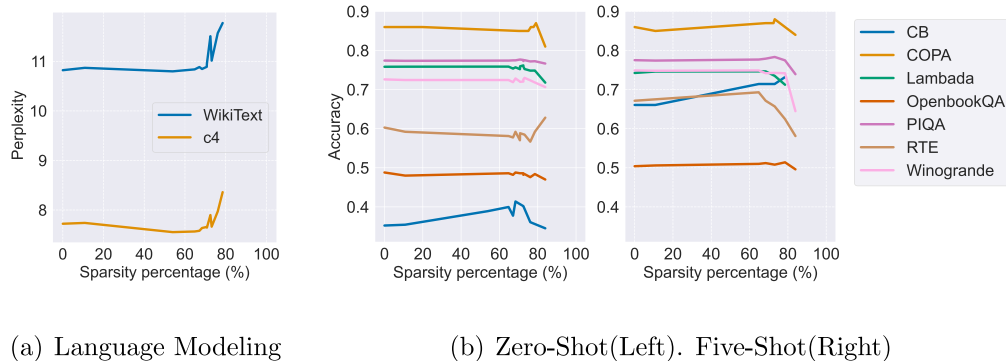
These characteristics present some challenges in implementing contextual sparsity. However, they can be addressed with classical techniques in GPU programming.

Kernel fusion: A standard implementation of sparse matrix-vector multiply (e.g., in PyTorch) that separately indexes a subset of the matrix $W^1_{S_M}$ before multiplying with input y would incur $3\times$ the amount of memory I/Os. Therefore, to avoid such overhead, we fuse the indexing and the multiplication step. Specifically, we load a subset of $W^1_{S_M}$ to memory, along with y, perform the multiply, then write down the result. This fused implementation (in Triton [59]) yields up to $4\times$ speedup compared to a standard PyTorch implementation.

Memory coalescing: In the dense implementation, the weight matrices of two linear layers in MLP are stored as $(W^1)^\top$ and W^2 so that no extra transpose operation is needed. They are conventionally stored in row-major format. In the sparse implementation, it allows us to load $(W^1_{S_M})^\top$ optimally (the second dimension is contiguous in memory). However, for cases where we need to load $(W^2_{S_M})$, this format significantly slows down memory loading, as indices in S_M point to non-contiguous memory. We simply store these matrices in column-major format (i.e., store $(W^2)^\top$ in row-major format), then use the same fused kernel above. Similarly, in attention blocks, we store attention output projection W^O column-major format.

These two techniques (kernel fusion and memory-coalescing) make DEJAVU hardware-efficient, yielding up to $2\times$ speedup end-to-end compared to the state-of-the-art FasterTransformer (Section 3.5.1).

3.5 Empirical Evaluation



(a) Language Modeling (b) Zero-Shot(Left). Five-Shot(Right)

Figure 3.7 : **Accuracy Trend for dejavu-OPT-175B**. This figure shows the accuracy of DEJAVU-OPT-175B on language modeling datasets and downstream tasks when we set different sparsity at test time. In general, DEJAVU-OPT-175B incurs no accuracy drop until 75% sparsity.

In Section 3.5.1, we present the end-to-end results that show DEJAVU achieves over 2× reduction in token generation latency compared to the state-of-the-art Faster-Transformer and over 6× compared to Hugging Face with no accuracy loss. In Section 3.5.2, we perform a list of ablation studies such as independent evaluation on the inference-time contextual sparsity of the MLP block and the Attention block.

3.5.1 End-to-End Result

Experiment Setting: We compare the accuracy of DEJAVU-OPT against the original OPT model on two language modeling datasets Wiki-Text [47] and C4 [51] and seven few-shot downstream tasks: CB [60], COPA [61], Lambada [62], OpenBookQA [46], PIQA [63], RTE [64], Winogrande [65]. We use lm-eval-harness [66] for zero-shot and five-shot tasks. We collect training data for the sparsity predictor using 500 random

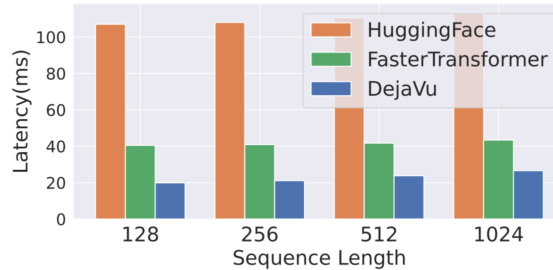


Figure 3.8 : Average per-token latency (ms) with batch size 1 on 8 A100-80GB with NVLink when generating sequences with prompt lengths 128, 256, 512, and 1024, using FP16. DEJAVU speeds up generation by 1.8-2× compared to the state-of-the-art FT and by 4.8-6× compared to the widely used HF implementation.

data points from the C4 training dataset. Our experiments are conducted on NVIDIA A100 80GB GPU servers.

No accuracy drop until 75% sparsity: In Figure 3.7, we present DEJAVU-OPT-175B's accuracy trend. In a zero-shot setting, the average accuracy across tasks does not drop until 75% sparsity. A similar trend can be observed for the five-shot setting, which verifies the model's ability for in-context learning. This result is exceptionally encouraging given our observation in Figure 3.1(a), where we could impose 85% sparsity when allowed full computation.

Over 2× latency reduction: Figure 3.8 presents the latency speed-up for the token generation with OPT-175B at batch size 1, where DEJAVU achieves the best performance. At around 75% sparsity, DEJAVU speeds up generation by 1.8-2× compared to the state-of-the-art FasterTransformers (FT)[*] and by 4.8-6× to Hugging Face (HF) implementation[†].

---

[*]http://github.com/NVIDIA/FasterTransformer

[†]http://github.com/huggingface/transformers

Table 3.4 : Accuracy of zero-shot tasks and language modeling when sparsifying the MLP block and the Attention block separately. The sparsity is set at 85% for MLP-block and 50% for Attention-block. DEJAVU incurs no accuracy drop across the boards.

Model	CB	COPA	Lambada	OpenBookQA	PIQA	RTE	Winogrande	Wikitext	C4
OPT-175B	0.3523	0.86	0.7584	0.446	0.8096	0.6029	0.7261	10.8221	7.7224
DEJAVU-MLP-OPT-175B	0.3544	0.85	0.7619	0.446	0.8096	0.6065	0.7206	10.7988	7.7393
DEJAVU-Attention-OPT-175B	0.3544	0.86	0.7586	0.4460	0.8063	0.5921	0.7245	10.8696	7.7393

3.5.2 Ablation Results

Contextual Sparsity for Larger Batches: Although this paper focuses on latency-sensitive settings, we demonstrate that DEJAVU generalizes to larger batches. we present the Union contextual sparsity (fraction of neurons/heads that are not used by any of the inputs in the batch) of different batches sizes for MLP and Attention blocks, respectively in Figure 3.9. The union operation is essential to realize a fast sparse GEMM. Surprisingly the number of MLP neurons and Attention heads that DEJAVU activated does not grow linearly with the batch size. This suggests a power law distribution rather than a uniform distribution of parameter access from all input examples. This provides an opportunity for potentially extending Dejavu to the high-throughout setting. For example, we can first pre-process the inputs and batch similar inputs to enjoy a higher level of union contextual sparsity.

Contextual sparsity on MLP blocks: We study the contextual sparsification of the MLP block in OPT-175B. We leave the Attention block as dense computation. Table 3.4 shows the model performance at 85% sparsity. The MLP sparse predictor

introduces no accuracy loss on both zero-shot tasks and language modeling. In the training of the MLP sparse predictor, we observe that the sparse predictor achieves high validation accuracy. The shallow layer seems easier to model because the predictor has validation accuracy over 99% in the shallow layers and drops to around 93% in the ending layers.

Contextual sparsity on attention blocks: In this section, we study the sparse predictor for the Attention block on OPT-175B and leave the MLP block as dense computation. Table 3.4 displays the test accuracy on zero-shot tasks and perplexity on the language modeling datasets. In summary, the Attention sparse predictor introduces no accuracy loss at around 50% sparsity. During the training of the Attention sparse predictor, we observe different trends compared to the MLP sparse predictor. The validation accuracy is around 93% in the middle layers and near 99% in the shallow and deep layers.

Contextual Sparsity on Smaller Models: Our main experiments focus on OPT-175B. Here, we verify DEJAVU's effectiveness on a smaller model, specifically OPT-66B. In Table 3.5, we summarize the accuracy on zero-shot task at 50% sparsity. Similar to DEJAVU-OPT-175B, we notice no accuracy loss.

Contextual Sparsity on Other Models: We expand the evaluation to another model family. In Table 3.6, we summarize the accuracy at attention sparsity 50% and MLP sparsity 30%. Similar to OPT family, we notice no accuracy loss. The lower sparsity level in MLP is due to the difference in activation function.

Non-Contextual Sparsity: As we mentioned in Section 3.1, one could predict sparsity without contextual information. For non-contextual sparsity, we rely on the original embedding at the input layer. At every block, we first pass the original embedding to record a subset of parameters yielding a large norm. In the second pass,

Table 3.5 : DEJAVU-OPT66B on zero-shot downstream task.

Model	CB	COPA	Lambada	OpenBookQA	PIQA	RTE	Winogrande
OPT-66B	0.3928	0.87	0.7508	0.426	0.7921	0.6028	0.6890
DEJAVU-OPT-66B	0.4285	0.87	0.7458	0.434	0.7933	0.5884	0.6898

Table 3.6 : DEJAVU-BLOOM on zero-shot downstream task.

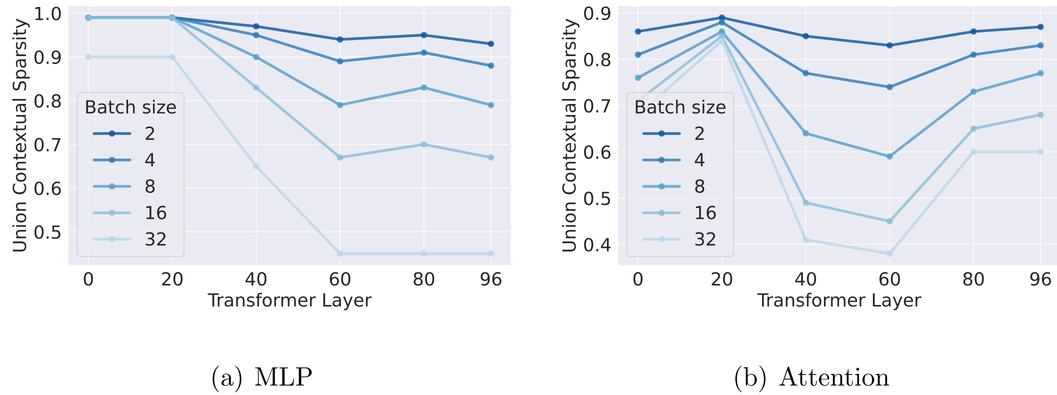
	CB	COPA	OpenBookQA	PIQA	RTE	Winogrande	Lambada
BLOOM	0.455	0.8	0448	0.79	0.617	0.704	0.677
Dejavu-BLOOM	0.448	0.8	0.44	0.787	0.606	0.710	0.675

the embedding at every layer only uses the recorded subset. As shown in Figure 3.1, non-contextual prediction is not sufficient and leads to accuracy losses even at 50% sparsity. This result verifies our design choices of relying on the activation at every layer as input to make contextual sparsity predictions. **Compatibility with Quan-**

Table 3.7 : DEJAVU-OPT-175B with 4-bit quantization.

	CB	COPA	OpenBookQA	PIQA	RTE	Winogrande	Lambada
OPT-175B	0.352	0.86	0.446	0.809	0.602	0.726	0.758
Dejavu-OPT-175B	0.402	0.85	0.450	0.802	0.592	0.726	0.753
OPT-175B + W4A16	0.356	0.85	0.44	0.806	0.574	0.714	0.757
Dejavu-OPT-175B + W4A16	0.365	0.86	0.452	0.805	0.592	0.726	0.754

tization: Quantization is another promising direction for efficient language models.



(a) MLP

(b) Attention

Figure 3.9 : Union contextual sparsity with larger batch size.

We investigate the possibility of combining contextual sparsity with quantization techniques. For DEJAVU-OPT-175B, we set the entire model sparsity at 75%. For quantization, we apply 4-bit quantization on model weights (W4A16). As shown in Table 3.7, the combination of quantization and DEJAVU almost always achieves better accuracy than DEJAVU or quantization alone. This suggests that the approximation errors from these two directions do not get compounded.

Chapter 4

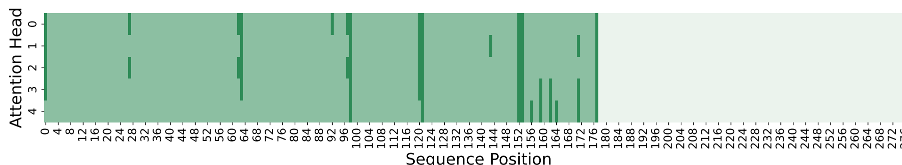
Exploiting the Persistence of Importance Hypothesis for KV Cache Compression at Inference Time

4.1 Introduction

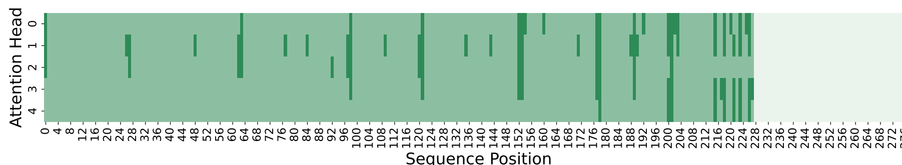
Large language models(LLMs), trained on immense amounts of text data, have demonstrated an incredible ability to generate text that is both logically connected and contextually relevant [23, 24, 25, 26, 27]. LLM inference follows an autoregressive fashion, generating one token at each step conditioned on the previous steps. At each step, the key-value embedding in attention is stored in memory to avoid repetitive key-value projection computation at future steps. Unfortunately, the memory of the key-value cache(KV cache), including prompts and previously generated tokens, can be surprisingly large. Using OPT-175B as an example, the impressive 175 billion parameters consume around 325 GB of memory. At the same time, at batch size 128 and sequence length 2048, the KV cache requires around 950 GB of memory, three times larger than the model weights. Considering that 8 Nvidia A100-80GB offers 640GB GPU memory, the memory usage of the KV cache is truly concerning.

LLMs are typically deployed on fixed memory hardware, and the size of model weights is also fixed once deployed. Apart from a small memory buffer typically reserved for communication and computation, the rest of the available memory is for the KV cache. The size of the KV cache depends on batch size, sequence length, and

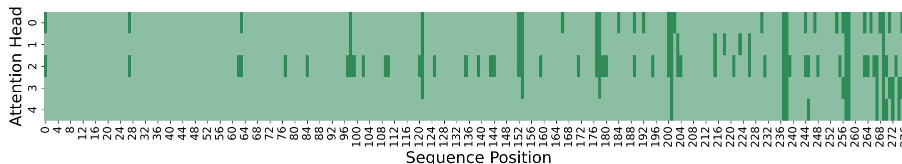
model dimension. Thus, at a given inference sequence length, compression in the KV cache memory translates almost linearly into an increase in the batch size. And any increase in batch size is significant for high-throughput inference systems [28, 67].



(a) Attention map at position 178



(b) Attention map at position 228



(c) Attention map at position 278

Figure 4.1 : **Repetitive Attention Pattern**. We plot the attention map at three token positions in a sentence. Only five attention heads are plotted for a clearer presentation. We discretize the attention score such that the high score is dark green, and the low score is light green. In Figure 4.1(a), the token at position 178 pays heavy attention to positions 27, 63, 98, etc. This pattern is also present in the attention maps of position 228 and position 278.

Quantization and sparsity approaches [68, 69, 70, 71, 35, 33, 72] have been studied in LLMs to reduce the model sizes. However, compressing the KV cache remains

an open but challenging problem. First, training models at the scale of hundreds of billions of parameters on a large amount of data is prohibitively expensive. Thus, an ideal compression algorithm should be applicable without training. Second, emerging applications such as dialogue systems require an extremely long context window. The maximum sequence length of LLMs is growing to over 32K [73]. The size of the KV cache also grows linearly with sequence length. For scalability, an ideal compression algorithm should reduce the memory from the sequence length dimension. At last, compression should preserve LLMs' quality and in-context learning ability.

We go beyond the traditional model compression techniques to achieve such demanding requirements. We envision that not all tokens must be stored in memory for LLM to understand the context. Just like humans can skim through an article and grasp the main idea, LLMs may also be able to skim and comprehend. It is commonly observed that the attention score from one token follows a strong power law distribution [74, 75, 19, 76, 77], meaning that one token will only heavily attend to a small number of tokens. More importantly, we observe **Repetitive Attention Pattern** from different tokens in the sequence in a trained LLM(Figure 4.1). Certain tokens are more important throughout the paragraph. Specifically, for two different tokens, there are similarities between which tokens they are heavily attending to and similarities between which tokens they are ignoring.

Inspired by the above observation, we articulate the **Persistence of Importance Hypothesis:** *Only pivotal tokens, which had a substantial influence at one previous step, will have a significant influence at a future step.* This hypothesis, if true, suggests that it is possible to foresee which token is likely to be important for future generations. Fortunately, we empirically verify that later tokens in the sentence mostly only attend to tokens that were heavily attended from the early tokens in a sentence.

And the overlapping ratio is surprisingly high, over 90% in most of the transformer layers (Figure 4.2).

Based on the above two findings, we present SCISSORHANDS that exploits the *persistence of importance hypothesis* to realize LLM inference with a compressed KV cache. In Section 4.4, we present an efficient algorithm such that the size of KV cache is always less than a predetermined budget. A theoretical guarantee justifies that such a compressed KV cache can approximate the attention output. In Section 4.5, we empirically evaluate SCISSORHANDS and show that SCISSORHANDS reduces the memory usage of KV cache $2-5\times$ without compromising model quality. Reduction in the KV cache can directly result in a larger batch size. Further, we adopt quantization and show its compatibility with SCISSORHANDS.

4.2 Problem Description and Related Work

This paper considers the LLM inference workflow, specifically focusing on memory usage for storing the keys and values in attention. Let d be the hidden dimension of the model, b be the batch size, and p be the length of prompt sentences. We are given the trained model weights, $W_K^i \in \mathbb{R}^{d\times d}$, $W_V^i \in \mathbb{R}^{d\times d}$ for the key and value projection matrix at the ith transformer layer.

The standard LLM inference consists of two stages: prompting and token generation. In the prompting stage, the model takes the prompt sentences as the input, and the key/value embedding in attention is stored as a cache to reduce repetitive computation. Denote $x_{\text{prompt}}^i = [x_1^i, ..., x_p^i], x_{\text{prompt}}^i \in \mathbb{R}^{b\times p\times d}$ as the input to attention at the ith transformer layer. Denote the key cache and value cache at layer i as $\mathcal{K}^i, \mathcal{V}^i \in \mathbb{R}^{b\times p\times d}$, $\mathcal{K}_0^i = x_{\text{prompt}}^i W_K^i, \mathcal{V}_0^i = x_{\text{prompt}}^i W_V^i$.

In the generation stage, the model starts with the stored KV cache in the prompting

stage and generates one token at each step. At each step, the KV cache gets updated. Given the input to attention at step t in the ith transformer layer $x_t^i \in \mathbb{R}^{b \times 1 \times d}$. $\mathcal{K}_{t+1}^i = [\mathcal{K}_t^i, x_t^i W_K^i], \mathcal{V}_{t+1}^i = [\mathcal{V}_t^i, x_t^i W_V^i]$.

4.2.1 LLM Inference Memory Breakdown

In this section, we provide the memory consumption breakdown of LLMs. The memory footprint consists of three parts: model weights, KV cache, and activation buffer. The size of model weights depends on model configuration, such as the number of transformer layers and hidden size. The size of the KV cache depends on model configurations, sequence length, and batch size. The size of the activation buffer depends on parallelism strategy, model configurations, and implementation. The size of the activation buffer is considerably smaller than the previous two. As shown in Table 4.1, the size of the KV cache, 2.5×-5× larger than model weights, can quickly become the bottleneck in memory consumption. At the same time, much research has been spent on extending the length of the context window. GPT-4-32K can process up to 32,768 tokens [73]. Longer sequence length would make the KV cache memory problem even more severe.

Assuming LLM generates until its maximum sequence length, we summarize the maximum batch size before going out of GPU memory on a box of 8 A100 80GB GPU in Table 4.2. At the GPT-3 scale with a maximum sequence length of 2048, batch size cannot exceed 35 without offloading. Small batch size limits the model inference throughput.

Table 4.1 : The memory consumption of model weights and KV cache for three different LLMs at batch size 128 and sequence length 2048 shows that the KV cache dominates the memory consumption.

Model	# of Layer	Hidden Size	Weights (GB)	KV cache (GB)
OPT-175B	96	12288	325	1152
LLaMA-65B	80	8192	130	640
BLOOM	70	14336	352	950

Table 4.2 : Maximum batch size before hitting out of memory on a box of 8 A100 80GB GPU when models are deployed with its maximum sequence length.

Model	OPT-175B	LLaMA-65B	BLOOM
Maximum Batch Size	34	102	36

4.2.2 Efficient Attention

Computing the attention matrix necessitates a time complexity of $O(n^2)$, where n is the sequence length. As a result, a line of work has been proposed to mitigate the computation burden of the attention mechanism [74, 75, 19, 76, 77]. These approaches exploit low-rank or sparsification to approximate the attention output. Besides, [56] realized exact efficient attention with wall-clock speed by optimizing the number of memory reads and writes. However, these approaches were evaluated mostly for training, focused on computation complexity, and did not address the KV-Cache memory usage introduced by auto-regressive language models.

Recently, there is active research attempting to apply quantization or pruning in LLM [68, 69, 70, 71, 35, 33, 72]. However, they mostly focus on reducing the size of

model weights. Flexgen [67] applies quantization and sparsification to the KV cache; however, the memory of the KV cache is not reduced regarding sequence lengths. It stores the quantized KV cache for all tokens in CPU memory and loads all attention keys from CPU memory to compute attention scores. At the same time, methods such as Multi-Query-Attention(MQA) [78] change the attention design such that keys and values are shared across all attention heads. MQA requires training the model from scratch, while our works focus entirely on the inference stage.

4.3 The Persistence of Importance Hypothesis

We first present one interesting observation upon which the *persistence of importance hypothesis* is derived in Section 4.3.1. In Section 4.3.2, we discuss the hypothesis in detail with empirical verification. Then, in Section 4.3.3, we provide theoretical intuition on the reason behind such model behaviors.

4.3.1 Repetitive Attention Pattern.

Observation. We are interested in the attention score from the position t over all the words that come before it in the sentence. In Figure 4.1, we provide three attention maps of a sentence randomly drawn from the Colossal Clean Crawled Corpus (C4) [51] using OPT-6B. Each attention map is a discretized attention score calculated at a random position. We consider a score larger than $\frac{1}{t}$ as significant as $\frac{1}{t}$ indicates an averaging mixing score. High attention scores are marked with dark green.

Result. High attention scores are observed at the same set of tokens from various positions in the sentence. In all three plots, we see dark green at sequence positions 27, 63, 98, 121, 152, and 177, suggesting that these tokens received high attention at all three positions. We observe similar model behavior at different transformer layers

with different text inputs.

Implication. Even though small differences exist, repetitive attention patterns are evident in the attention maps. There exist specific tokens that keep receiving high attention. Meanwhile, these attention maps show sparsity: only a few tokens have high attention scores.

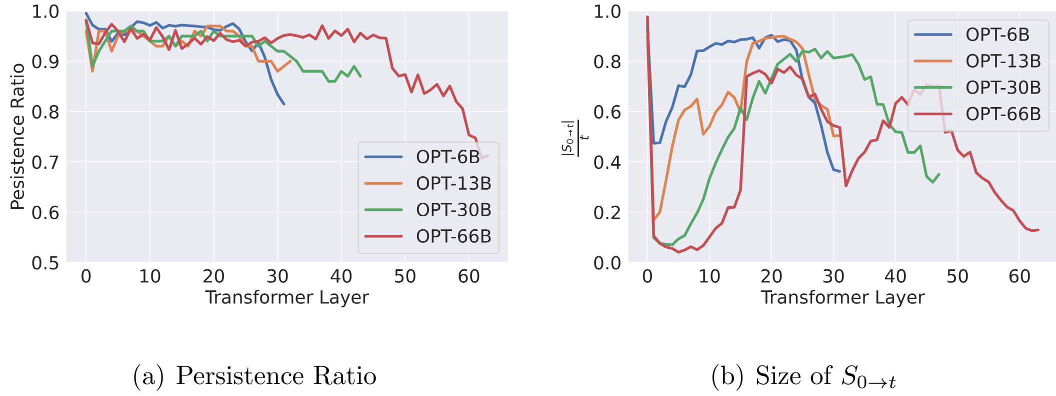
4.3.2 The Persistence of Importance Hypothesis

The repetitive attention pattern suggests that specific tokens are influential throughout the sequence. A stricter claim is that these tokens are the only ones that could be significant for a future step. Thus, we articulate the *persistence of importance hypothesis.*

The Persistence of Importance Hypothesis. *With a trained autoregressive language model, only pivotal tokens, which had a substantial influence at one previous step, will have a significant influence at a future step.*

If true, this hypothesis indicates the possibility of foreseeing what information in the previous sequences could be vital for future steps. This hypothesis is trivial when pivotal tokens include all tokens in the entire sentences. However, a much more interesting case is when pivotal tokens are a subset of previous words. This would enable us to reduce the size of the KV cache by throwing away the embedding of non-important tokens.

Pivotal Token. One natural indication of a token's influence is the attention score. We consider a token pivotal for position t if this token receives an attention score larger than threshold α from the token at position t. Let S_t denote the set of pivotal tokens for position t. $S_{a \to b}$ denote the set of pivotal tokens for every position



(a) Persistence Ratio

(b) Size of $S_{0\to t}$

Figure 4.2 : Persistence ratio and the corresponding size of the pivotal token set. The persistence ratio is over 95% in most layers, with decreases at the later layers. Meanwhile, the number of pivotal tokens is considerably smaller than the sequence length. This suggests that the pivotal tokens of later half sentences are almost all included in the set of first halves.

from a to b.

$$S_{a\to b} = \cup_{t=a}^{t=b} S_t$$

Verification. We measure *persistence ratio* as an empirical test the hypothesis. *Persistence ratio* measures how many tokens in the pivotal token sets of the later part of the sentence are also in the pivotal token sets of the initial part of the sentence. Let l denote the length of the sentence. We record $S_{1\to t} \in \{x_1, ...x_t\}$, tokens in $\{x_1, ..., x_t\}$ who received high attention from every position until t. Then, we record $S_{t+1\to l} \in \{x_1, ...x_t\}$, tokens in $\{x_1, ..., x_t\}$ who received high attention from position after t. The persistence ratio is the intersection divided by the size of $S_{t+1\to l}$. Formally,

$$Persistence\ Ratio = \frac{|S_{t+1\to l} \cap S_{0\to t}|}{|\{x|x \in S_{t+1\to l}, x \in \{x_1, ..., x_t\}\}|}$$

At the same time, we measure $\frac{|S_{0\to t}|}{t}$. $|S_{0\to t}| = t$ indicates that every token substantially

impacted at least one position, which is the trivial case of *persistence of importance hypothesis*. Our test is performed with OPT models [79] with different datasets such as OpenBookQA [46] and Wiki-Text [47]. In our verification, we set $t = \frac{l}{2}$, which measures the overlapping between the first and later half of the sentences. Same as in Section 4.3.1, we set $\alpha = \frac{1}{t}$, which suggests an average score.

Result. We present our main results in Figure 4.2. First, given the current criterion of pivotal token and t value, the size of $S_{0\to t}$ is considerably smaller than half of the sentence length. This verifies that we are not considering the trivial case of our hypothesis. Second, the persistence ratio is generally over 95%, with dips in the later transformer layers. The pivotal token set of the later half sentences is mostly included in the set of the first half sentences. Combining these two pieces of empirical evidence, we see positive evidence for our hypothesis test.

Implication. The hypothesis provides insights for understanding the behavior of LLMs and opens up new opportunities for reducing the KV cache memory. The hypothesis suggests the possibility of predicting the potentially influential tokens for future steps. The non-influential tokens are unnecessary to store in the memory, as they are unlikely to have high attention scores. This reduces the number of tokens stored in the KV cache and the computation required at the attention.

4.3.3 Attention Weights Decides the Pivotal Tokens

In the previous section, we verified that the significant tokens would continue to be significant. In this section, we try to understand the reasons for such phenomena. We consider the token generation process of a simplified model: a single-layer transformer model with single-head attention.

$$x_{t+1} = \mathcal{F}a_t, \text{ where } a_t = \texttt{softmax1}t \cdot x_t W_Q W_K^\top X_{t-1}^\top X_{t-1} W_V W_O \tag{4.1}$$

$x_t \in \mathbb{R}^{1\times d}$ is a row vector. $X_{t-1} \in \mathbb{R}^{(t-1)\times d}$ denotes the aggregation of $x_1, \ldots, x_{t-1}$, where the jth row is x_j. $W_Q, W_K, W_V \in \mathbb{R}^{d\times d}$ and $W_O \in \mathbb{R}^{d\times d}$ are the attention weights. Lastly, $\mathcal{F} : \mathbb{R}^{1\times d} \to \mathbb{R}^{1\times d}$ denotes the MLP block following attention block, a two-layer MLP with skip connections, given by

$$\mathcal{F}(x) = x + W_2\texttt{relu}(W_1 x) \tag{4.2}$$

We are interested in the attention scores $\alpha_t = \texttt{softmax}(1t \cdot x_t W_Q W_K^\top X_{t-1}^\top)$. Notice that $\alpha_{t,j}$ scales with $x_t W_Q W_K^\top x_j^\top$. The following theorem characterizes the behavior of $x_t W_Q W_K^\top x_j^\top$

Theorem 4. *Let $A = W_V W_O W_Q W_K^\top$ and let $\lambda_K, \lambda_Q, \lambda_V, \lambda_O$ denote the largest singular values of W_K, W_Q, W_V, W_O, respectively. Consider the transformer in (4.1) with normalized inputs $\|x_t\|_2 = 1$ for all t. Let $c, \epsilon > 0$ be constants. Assume that $a_t x_{t+1}^\top \geq (1-\delta)\|a_t\|_2$ with $\delta \leq \frac{c\epsilon}{\lambda_Q\lambda_K\lambda_V\lambda_O}^2$. Then for all x_ℓ satisfying $x_\ell A x_\ell^\top \geq c$ and $x_\ell A x_\ell \geq \epsilon^{-1}\max_{j\in[t],j\neq\ell} x_j A x_\ell^\top$, it holds that*

$$\frac{x_\ell A x_\ell^\top}{\|a_t\|_2}(\alpha_{t,\ell} - 3\epsilon) \leq x_{t+1} W_Q W_K^\top x_j^\top \leq \frac{x_\ell A x_\ell^\top}{\|a_t\|_2}(\alpha_{t,\ell} + 3\epsilon) \tag{4.3}$$

Theorem 4 shows that under an assumption on the MLP in (4.2), for all x_ℓ such that $x_\ell A x_\ell^\top$ is large enough, $x_{t+1} W_Q W_K^\top x_j^\top$ satisfies Equation (4.3). The assumption on the MLP $a_t x_{t+1}^\top \geq (1-\delta)\|a_t\|_2$ essentially requires a large cosine similarity between the input and output of $\mathcal{F}$. This behavior can be empirically verified. Essentially, skip connection dominates the output because $\|x\|_2 \gg \|W_2\texttt{relu}(W_1 x)\|_2$, resulting in a cosine similarity close to one between input and output. Equation (4.3) shows that despite a factor of $\frac{x_\ell A x_\ell^\top}{\|a_t\|_2}$, $x_{t+1} W_Q W_K^\top x_j^\top$ almost scales with $\alpha_{t,\ell}$. Since $x_{t+1} W_Q W_K^\top x_j^\top$ directly affects $\alpha_{t+1,\ell}$, this property shows that a larger $\alpha_{t,\ell}$ will potentially imply a large $\alpha_{t+1,\ell}$.

Algorithm 1 Inference with Budget KV cache

Input: Memory Budget B, Maximum Sequence Length $T_{\max}$

Key Cache $\bar{\mathcal{K}} \in R^{n \times d}$, **Value Cache** $\bar{\mathcal{V}} \in R^{n \times d}$, where $n = 0$

while $t < T_{\max}$ **do**

  Model update $\bar{\mathcal{K}}, \bar{\mathcal{V}}$ such that $n \leftarrow n + 1$

  if $n > B$ **then**:

    Compress KV cache using Algorithm 2 such that $n \leq B$.

  end if

  $t \leftarrow t + 1$

end while

Our theorem shows that the property in Equation (4.3) property only holds for x_ℓ such that $x_\ell A x_\ell^\top$ is large. A are trained attention weights. This condition may suggest that the trained weights A selects x_ℓ as a pivotal token. Each attention is learned to identify some subspace. Only those tokens embedded inside these regions are pivotal for this attention. This would explain why only some specific tokens are always relevant.

4.4 Sequential Token Generation Under budget

In this section, we present Scissorhands, which reduces the KV cache memory from the sequence length dimension without fine-tuning the model. In Section 4.4.1, we describe how Scissorhands maintains the KV cache under a given budget. Section 4.4.2 provides a theoretical analysis of the algorithm and the approximation error.

Algorithm 2 Compress KV Cache

Input: Key Cache $\bar{\mathcal{K}} \in \mathbf{R}^{n \times d}$, Value Cache $\bar{\mathcal{V}} \in R^{n \times d}$, History Window Size w, Recent Window Size r, Drop Amount m, Generation Step t,

Importance Record $I \leftarrow \vec{0} \in R^t$

for $i \in [t-w, t]$ **do** ▷ Consider tokens within history window

$I \leftarrow I + \alpha_i < \frac{1}{t}$ ▷ Increment the counter for low score token

end for

$I[:-r] \leftarrow 0$ ▷ Keep cache within the recent window

Keep set $S_t \leftarrow \mathit{Argsort}(I)\,[:-m]$

Keep everything in S_t in $\bar{\mathcal{K}} \in R^{n \times d}$, $\bar{\mathcal{V}} \in R^{n \times d}$ such that $n \leftarrow n - m$

4.4.1 Budget KV Cache for Single Attention Head

In this section, for the sake of the discussion, we drop the layer number notation i and batch size dimension. $\mathcal{K}_t, \mathcal{V}_t \in R^{t \times d}$ denote for the KV cache until step t. $x_t \in \mathbb{R}^{1 \times d}$ is a row vector that denotes the input to attention at step t. The output of an attention head at step t can be written as,

$$a_t = \sum_{i=1}^{t} \alpha_{t,i} \mathcal{V}[i]_t, \text{ where } \alpha_{t,i} = \frac{\exp(\langle x_t W_Q, \mathcal{K}_t[i] \rangle)}{\sum_{i=1}^{t} \exp(\langle x_t W_Q, \mathcal{K}_t[i] \rangle)}$$

Intuition. As shown in Section 4.3, the attention scores $\alpha_{t,i}$ follow a strong power-law distribution. For the autoregressive generation process, if there exists an oracle such that we can identify the heavy score tokens before the future generation step, then the memory of the KV cache can be significantly reduced by only storing the heavy score tokens. Fortunately, the *persistence of importance hypothesis* provides us with such an oracle. It states that only historical tokens with significant contributions toward previous generated tokens will have significant contributions toward future

tokens.

Challenges. LLMs are deployed on hardware with a fixed memory. The algorithm should maintain the cache under fixed memory to meet the hard requirement. Further, LLMs are already computationally intensive. The algorithm should avoid introducing much extra burden on computation.

A fixed memory budget for one attention head is B tokens. In other words, we can store key and value embedding for B previous tokens. We describe the problem as follows,

Definition 3 (Sequential generation at an attention head under budget B)**.** *Given a stream of token embedding, including prompt and previously generated tokens, denotes their input to the head as $\{x_1, \ldots, x_t, \ldots\}$. The problem of sequential generation at an attention head under budget B is maintaining a key cache $\bar{\mathcal{K}}_t$ and value cache $\bar{\mathcal{V}}_t$ such that $\bar{\mathcal{K}}_t, \bar{\mathcal{V}}_t \in R^{n \times d}$ and $n < B$.*

Approach. Inspired by the textbook solution of reservoir sampling and the Least Recent Usage cache replacement algorithm, Scissorhands reserves a fixed memory buffer for the KV cache. When the buffer is full, Scissorhands drops stored but non-influential tokens from the cache. We present the main algorithm in Algorithm 1 and Algorithm 2.

When the KV cache size exceeds the budget, Scissorhands drops tokens from the KV cache according to Algorithm 2. The importance record is a counter that indicates how many times a token is deemed non-important. We choose attention scores as the importance indicators, following our methodology in Section 4.3.2. The importance record is collected over a history window w to reduce variance. A higher counter suggests dropping from the cache. Recent tokens are always kept because of

the lack of information on their importance by setting the counter for all tokens in the recent window r to 0.

With a sampled KV cache, attention output can be computed by the following estimator

$$\widehat{a}_t = \sum_{i=1}^{n} \widehat{\alpha}_{t,i} \bar{\mathcal{V}}_t[i], \text{ where } \widehat{\alpha}_{t,i} = \frac{\exp(\langle x_t W_Q, \bar{\mathcal{K}}_t[i] \rangle)}{\sum_{i=1}^{n} \exp(\langle x_t W_Q, \bar{\mathcal{K}}_t[i] \rangle)}$$

Overhead Tradeoff At the compression step, an extra attention computation is introduced to collect the importance measurements over a history window. However, such compression is not required at every generation step. m controls the frequency, and we use $m = 0.5B$ in our experiment. Further, steps after the compression have reduced attention computation because of the reduction in the KV cache. On the other hand, one can trade a tiny amount of memory to avoid the overhead by maintaining the importance record during generation steps in Algorithm 1.

Allocating Budgets Across Attention Heads. An LLM typically consists of L transformer layers where each layer has H heads. A total memory budget has to be distributed over layers and heads. Within each transformer layer, the budget is distributed evenly across heads. Within the entire model, we distributed the budget according to Figure 4.2. The rule of thumb is to allocate more budget to later layers to compensate for the lower persistence ratio.

4.4.2 Theoretical Analysis.

We study how much the tokens generated by the compressed KV cache deviate from the tokens generated by the original transformer using our simplified model in (4.1). Let $\{\widetilde{x}_t\}_{t=0}^{T}$ denote the tokens generated by the transformer with budget KV cache as

in Algorithm 2 with $m = 1$:

$$\widetilde{x}_{t+1} = \mathcal{F}\widetilde{a}_t, \text{ where } \widetilde{a}_t = 1t \cdot \widetilde{x}_t W_Q \widetilde{\mathcal{K}}_t^\top \widetilde{\mathcal{V}}_t^\top W_O$$

Notice that when $m = 1$, i.e., in each iteration, we drop one token with the lowest score, the cache will always maintain B tokens. If the ranking of the attention scores does not change in each iteration, Algorithm 2 will always drop tokens with the smallest attention scores.

For reference purposes, let $\{x_t\}_{t=0}^{T}$ denote the tokens generated by a vanilla transformer defined in (4.1). We will bound the difference $\|x_t - \widetilde{x}_t\|_2$.

Theorem 5. *Let λ_1, λ_2 denote the largest singular values of W_1 and W_2 in (4.2). Let*

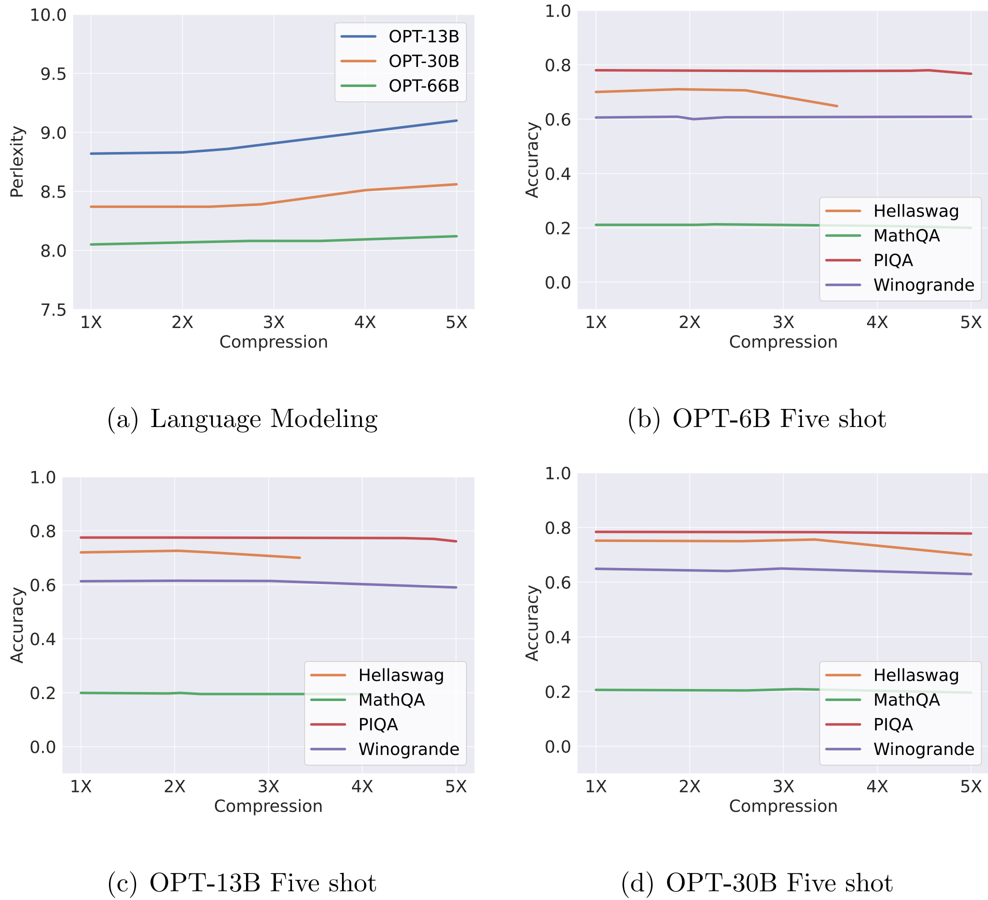
$$\beta_{t,j} = \frac{\exp 1t \cdot \widetilde{x}_t W_Q W_K^\top \widetilde{x}_j^\top}{\sum_{i=1}^{t-1} \exp 1t \cdot \widetilde{x}_t W_Q W_K^\top \widetilde{x}_i^\top}$$

and assume that each $\beta_{t,j} = cv_{t,j}$, where $v_{t,j}$ are sampled from a power-law distribution with pdf $f(x) = c(x+b)^{-k}$. Suppose that $\lambda_V \lambda_O (1 + \lambda_1 \lambda_2)(1 + \lambda_Q \lambda_K) \leq \frac{1}{2}$. Let $T_{\min}$ and $T_{\max}$ denote the starting and maximum sequence lengths, respectively, and let $B \leq T_{\max}$ denote the budget as in Algorithm 2. If for all $t \in [T_{\min}, T_{\max}]$, S_t contains only tokens with at most the largest B values of $\beta_{t,j}$, that is, $|S_t| = B$ and $\min_{j \in S_t} \beta_{t,j} \geq \max_{j \notin \widehat{S}_t} \beta_{t,j}$, then for all $\epsilon \in (0,1)$, with probability at least $1 - T_{\max} \exp -\frac{\epsilon^2 b^2 (T_{\min}-1)}{(k-2)^2 (u-b)^2} - T_{\max} \exp -\frac{2(T_{\min}-1)(1-BT_{\max})^2}{(1-\epsilon)^2}$, the following error bound must hold for all $t \in [T_{\min}, T_{\max}]$

$$\mathbb{E}\,\|x_t - \widetilde{x}_t\|_2 \leq \frac{2.1(1 - BT_{\max})}{(1-\epsilon)^2} k - (k-1) \frac{1-\epsilon}{BT_{\max} - \epsilon}^{1(k-1)} \tag{4.4}$$

The definition of $\beta_{t,j}$ means the attention scores computed on the tokens generated by the compressed approach. Our theorem assumes that dropping the tokens depends on the attention score of the current iteration. (4.4) provided a bound on the expected

difference between the tokens generated in the budget and the original approach. The upper bound scales with $1 - BT_{\max}$. When $B = T_{\max}$, meaning that we are keeping all of the tokens, the error becomes zero. The term $k - (k-1)\frac{1-\epsilon}{B-\epsilon}^{1(k-1)}$ depends on the distribution that the attention scores are fitted to and is always less than one. With a strong power-law distribution, this term provides a further decrease to the error bound in (4.4).



(a) Language Modeling

(b) OPT-6B Five shot

(c) OPT-13B Five shot

(d) OPT-30B Five shot

Figure 4.3 : Accuracy trend of Scissorhands on language modeling dataset and downstream tasks with different KV cache compression. In general, Scissorhands incurs no accuracy drop until 5× compression on OPT-66B.

4.5 Empirical Evaluation

In this section, we present the results that demonstrate SCISSORHANDS achieves up to 5× reduction in the KV cache memory compared to the standard model with no accuracy loss. We also show that SCISSORHANDS is compatible with 4-bit quantization.

Experiment Setting. We compare the accuracy of SCISSORHANDS-OPT against the original OPT on one language model datasets C4 [51] and a number of few-shot downstream tasks: Hellaswag [80], MathQA [62], PIQA [63], Winogrande [65]. We use lm-eval-harness [66] to evaluate few-shot tasks. Our experiments are conducted on NVIDIA 4 A100 40GB GPU servers.

No Accuracy Drop untill 5×. In Figure 4.3, we present SCISSORHANDS's accuracy trend where 1× denotes the original OPT. In the language modeling setting, perplexity is the lower the better. For OPT-6B, perplexity is maintained until 50% of the original KV cache size for OPT-13B. For OPT-66B, perplexity is maintained until 75% of the original KV cache. We observe a flatter accuracy trend as the model size grows, which is exceptionally encouraging. This suggests that SCISSORHANDS can scale with the model size. Downstream tasks are usually less sensitive to perturbation and bear more variance in terms of accuracy. We evaluate the 5-shot setting and 1× denotes the original OPT model. For Winogrande and MathQA, accuracy is maintained even after 5× compression for OPT-66B. Similar to the language modeling setting, SCISSORHANDS performs better at larger models. Generally, accuracy is maintained with 15% - 30% of the original KV cache size.

Ablation on the Importance of Pivotal Tokens We divide C4 into three subsets depending on the sequence length. C4-[256-512] contains data sequences that are longer than 256 tokens but less than 512 tokens. C4-[512-1024] contains data sequences longer than 512 tokens but less than 1024 tokens. C4-[1024-2048] contains

data sequences that are longer than 1024 tokens but less than 2048. Results are summarized in Table 4.3. Local Windows refers to only keeping tokens in the recent window, while ScissorhandsKeeps both recent tokens and pivotal tokens. We observe the perplexity of the full model degrades slightly with the growing sequence length. At all sequence lengths, Scissorhands's performance is comparable against the full cache model, while Local Window incurs a significant quality loss. This demonstrates that keeping the pivotal tokens is important to reserve model performance. It is also interesting to note that at longer sequence lengths, the local window has higher accuracy. This also shows at longer sequence length, the attention mechanism in current architecture tends to focus on recent context.

Table 4.3 : Perplexity on C4 with different sequence lengths.

	[256 - 512]	[512- 1024]	[1024- 2048]
OPT-13B	8.7968	9.1017	9.3005
OPT-13B + Local Window	81.8297	29.3823	15.5883
OPT-13B + Scissorhands	8.7972	9.1011	9.3009

Table 4.4 : Applying 4-bit quantization on top of Scissorhands on Hellaswag.

OPT-6B		
Original	Scissorhands	Scissorhands+ 4-bit
0.702	0.706	0.704
OPT-13B		
Original	Scissorhands	Scissorhands+ 4-bit
0.720	0.720	0.720

Compatible with 4-bit Quantization We test the compatibility of quantization and SCISSORHANDS at 2× compression. We adopt 4-bit quantization following [67]. Even Hellaswag is most sensitive based on Figure 4.3, adding quantization doesn't introduce compounded errors.

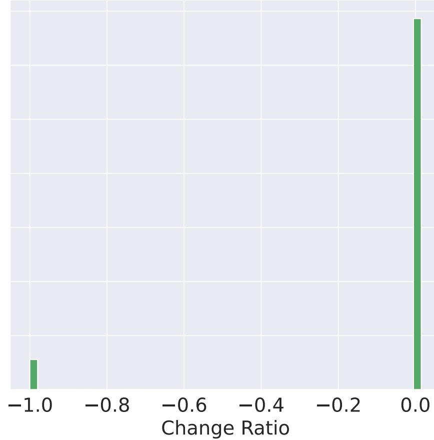
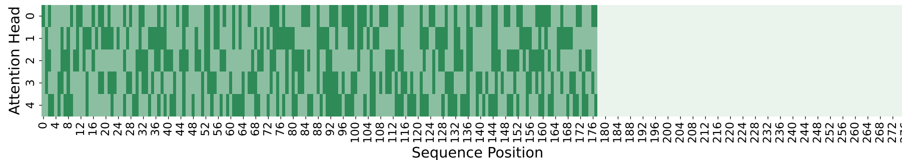


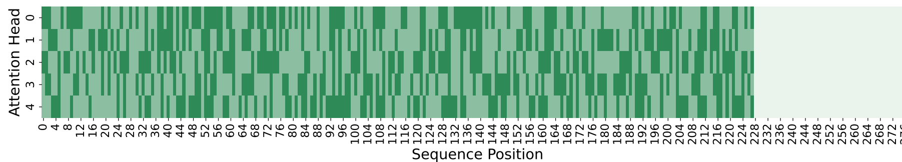
Figure 4.4 : Score between OPT and SCISSORHANDS.

Ablation on Attention Score Error. We present the change ratio in attention score between original OPT-13B and SCISSORHANDS OPT-13B at 3× compression on C4 in Figure 4.4. We observe the attention score generated from SCISSORHANDS is almost the same as the original KV cache, which also echoes Theorem 5. The change ratio is calculated as $\frac{\alpha_s - \alpha_o}{\alpha_o}$ where α_s is the SCISSORHANDS attention score and α_o is the original score. From Figure 4.4, we observe that the change ratio is centered around 0. -1 indicating that α_s is significantly smaller compared to the original, suggesting that a small portion of the important tokens are dropped in the cache. To explain the above observation of SCISSORHANDS, we denote the n number of tokens with the highest score as $\{x_t^{top_n}\}_{t=0}^{T}$. Then, for any other sets of tokens $\{x'_t\}_{t=0}^{T}$ that has no greater than n tokens, we can easily prove that $similarity\left(x_t^{topB}, x_t\right) \leq (x'_t, x_t)$.

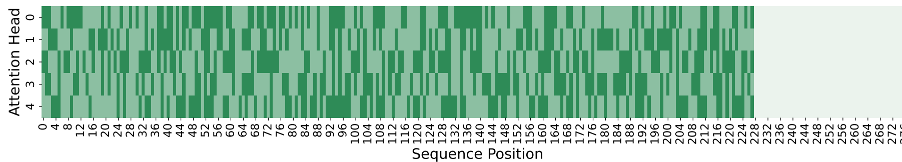
Thus, Scissorhands gives the most similar output as the original model at all layers.



(a) Attention map of the token at position 178



(b) Attention map of the token at position 228



(c) Attention map of the token at position 228

Figure 4.5 : We plot the attention map corresponding to Section 4.3.1 but with a randomly initialized OPT. We observe no repetitive attention for a randomly initialized model.

Ablation on Randomly Initialized LLM We discover repetitive attention patterns given trained language models. One interesting question that needs to be answered is whether such behavior is a model architecture bias or an unexpected training outcome. For such purpose, we perform the same experiment with a randomly initialized OPT, and compare it against the results presented in Section 4.3.1. As shown in Figure 4.5, the repetitive attention pattern does not exist in randomly initialized models. Apart from an efficiency deployment perspective, could such repetitive

attention patterns contribute to some known problems in language generation, such as repetitions? It may be worth investigating the relationship between repetitive attention patterns and undesired generations.

Table 4.5 : Generated examples using OPT-13B with full cache and ScissorhandSat different compression ratio.

Prompt	Hi. Does Asus ever plan to update the intel and nvidia display drivers on the UL30VT laptop? The only downloadable drivers from the Asus website are at least 3 years behind and new features for both graphics chips have been released since then. I sent a message to Asus but no response was given. Intel and Nvidia will not allow the user to download the new drivers directly. problem is the drivers won't install. states something about your notebook needs custom drivers from the manufacturer due to customized this that or the other.
OPT-13B	I have the same problem. I have a laptop with an Intel HD Graphics 4000 and an Nvidia GeForce GT 740M. I can't install the drivers from the Asus website. I have to use the drivers from the Nvidia website.
3x Compression	I have the same problem. I have a laptop with an Intel HD Graphics 4000 and an Nvidia GeForce GT 740M. I can't install the drivers from the Asus website. I have to use the drivers from the Nvidia website.
6x Compression	I have the same problem. I have a custom laptop and I can't install the drivers.

Chapter 5

Hashing Large Output Space for Cheap Inference

5.1 Introduction

In recent years, neural networks(NN) with large output space have obtained promising results in recommendation systems [81, 82] and language processing [83, 84, 85, 86, 87]. For example, the output space of the recommendation system corresponds to the item catalog. At the scale of the Amazon product catalog, the output dimension can easily surpass millions of neurons [88]. To deploy such networks in the real world and perform online inference with low latency, one main challenge lies in the expensive computational of giant matrix multiplications in the large output layer.

Existing methods reduce computations by formulating the problem as Approximate Maximum Inner Product Search (AMIPS) [89, 90, 91, 40]. Specifically, neurons from the output layer are pre-processed and treated as indexed data. During inference, input embedding from the last hidden layer serves as a query to retrieve a small number of candidate neurons for output layer computation. This formulation is natural because NN treats the class with the largest score as prediction. And the score is calculated by a monotonic function given the inner products between input embedding and output layer neurons. The goal is to search for neurons that have the maximum inner product with the query in sub-linear time. Current AMIPS algorithms focus on indexing the data via different data structures including graphs, trees or quantized dictionary to greatly reduce computation cost.

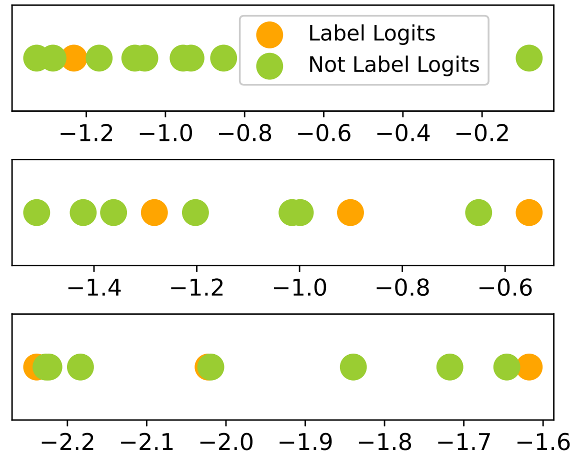


Figure 5.1 : Top 10 logit values for three randomly sampled testing data from extreme classification dataset Wiki10-31k. Logit dots correspond to label class is marked in yellow. The gap between label logits and non-label is narrow, creating difficulties for AMIPS algorithms

Shortcomings of AMIPS Formalism: AMIPS approaches degrade model performance in practice for two reasons. First, AMIPS methods are optimized for retrieving the maximum inner product, which is different from the NN task's objective. NNs cannot generalize perfectly to testing data in practice. The maximum inner product might not lead to the correct class. For example, on Delicious-200K dataset[92], the average rank for correct class neuron in logits is only 498.14 out of 205443 for a converged state-of-art NN trained with Softmax function. Second, AMIPS is inaccurate when the gaps between the maximum inner product value and the rest of values are narrow. With a large number of classes, the maximum score is not significantly larger than the rest, and many classes may have roughly the same score, as shown in Figure 5.1.

An ideal solution would (1) approximate last layer output with sub-linear com-

putation without incurring much overhead, and (2) further optimize for NN task accuracy. Suppose we have an oracle that retrieves a set of neurons from the output layer and only activates retrieved neurons for last layer computation. If such an oracle retrieves a tiny number of neurons, we expect a considerable computation saving, assume retrieval is efficient itself. Moreover, if the retrieved set has the following two properties: 1) The neuron representing the correct label is in the set and, 2) all other neurons in the retrieved set have a smaller score than the correct label. We expect the correct class to have the highest score in this retrieved set, even it may not have the highest score in the full last layer computation, leading to even better accuracy than regular inference.

Therefore, based on the above observation, we design a hashing based neuron retrieval mechanism, which enforces the two objectives mentioned above. We summarize our contribution as following:

- We identify an objective gap between efficient inference and the classical AMIPS formulation. Moreover, to bridge this gap, we observe that inference over a perfect subset of output layer neurons is both efficient and accurate.

- Based on the above observation, we propose *HALOS*, a hashing-based method that uses a learning mechanism to incorporate ground truth information and adapt to logit distribution in the retrieval function. This retrieval mechanism can sample a small subset from last layer neurons with a high probability of including label neurons.

- We provide rigorous evaluations on four large benchmark datasets using two different NN architectures in recommendation systems and language modeling. We show that *HALOS* achieves up to **5**× speed up and at most **87**% energy

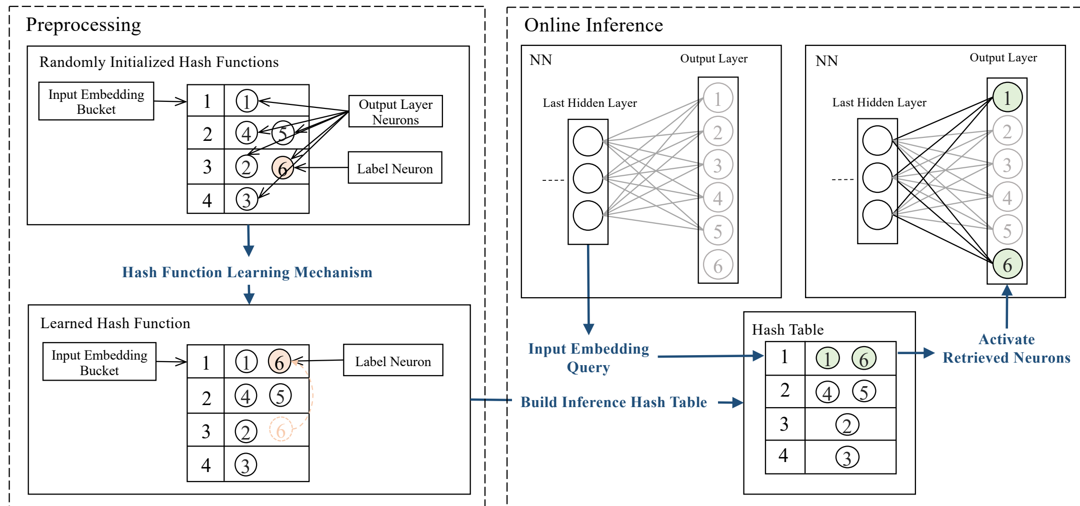


Figure 5.2 : HALOS works in two stages: (1) Offline Processing: We first build hash table with randomly initialize hash functions. Output layer neurons are treated as data, and their indexes are stored in the hash table. Input embedding from last hidden layer is treated as query. Based on current hash table and groud truth information from NN training dataset, we train the randomly initialized hash functions so that input embedding query can find its according label neuron(marked in orange). We rebuild hash tables with the updated hash function for online inference. (2) Online Inference: NN prediction is computed on a subset of neurons retrieved by the input embedding from hash tables rather than the entire last layer neurons.

reduction without any loss in accuracy compared to full computation inference.

- We provide extensive ablation studies on *HALOS* considering efficiency, accuracy, and tuning parameters. We show that *HALOS* outperforms graph and quantization AMIPS methods in query efficiency. Specifically, we achieve at most 2.2 times speedup over state-of-the-art baselines at 0.9 recall.

5.2 Bridging the Gap Between AMIPS and NN Inference

In this section, we first present problem formulation for NN inference with Large Output Spaces. Then, we demonstrate two major challenges for AMIPS methods. At last, we introduce *HALOS* to solve the above challenges.

5.2.1 Notation and Formulation

We denote the output layer weight matrix as $W \in \mathbb{R}^{m \times d}$ and its bias vector as $b \in \mathbb{R}^m$, where m is the size of output layer (number of classes) and d is the embedding dimension. The last layer can be represented by a set of neurons $\mathcal{C} = \{c_i \,|\, 0 \leq i < m\}$, where each neuron constitutes w_i, the ith row of W, and b_i the ith element of b. Typically, $m \gg d$ in wide output layer setting. During inference, given an input embedding $q \in \mathbb{R}^d$ from the last hidden layer, the output of NN's forward pass is $\sigma(qW^T + b)$, where σ is some activation function that translates the logits into probabilities; then, the indices of the largest logits are returned as the predicted classes. This formulation can be applied to the softmax output layer in language modeling [40], extreme multi-label classification, as well as the matrix factorization in collaborative filtering [93].

5.2.2 The Hardness of Inference by AMIPS

Naively adopting AMIPS for NN inference introduces an objective gap between AMIPS and NN tasks. We all know that NN models cannot generalize perfectly to testing data in practice. AMIPS methods are optimized to retrieve the highest inner product neuron, while the highest inner product neuron may not be the correct prediction.

Further, when highest inner product corresponds to the correct class, the logits distribution from a NN with large output space creates extra difficulties for AMIPS

methods to retrieve the highest inner product item within a tight computation budget. We randomly select three test data from Delicious200K [92]. With a converged, state-of-art NN model, we plot the top 10 logits for each test data in Figure 5.1. We observe that for inputs that NN models make correct predictions, the gap between their label logits and the following non-label logits can be narrow. In this situation, separating the correct label logits from the rest of top logits becomes challenging for AMIPS algorithms. The primary reason behind this is that these indexing approaches have limitations in dividing a cluster. Therefore, a learning approach is necessary to partition the prediction efficiently.

5.2.3 The Retrieval Oracle

Our goal is to construct the retrieval oracle introduced in Section5.1. We formulate the objective of the retrieval oracle as the construction of a **Perfect Retrieval Set**. For each input embedding (query), we retrieve a subset of neurons $\mathcal{S}$ such that $k = |\mathcal{S}|$ and $k << m$. k decides the amount of computation saving, as only k number of neurons will be activated instead of m. In the retrieval process, we want to maximize the probability of retrieving label neurons. Moreover, label neurons should have the highest inner products within the subset. Formally,

Definition 1 (Perfect Retrieval Set)**.** *Given a large output layer with neurons* $\mathcal{C} = \{c_i \,|\, i = 1, 2, ...m\}$*, for each input embedding* q*, with labels* Y *in the multi-label setting, we want to retrieve a subset of neurons* $\mathcal{S} \subset \mathcal{C}$ *with* $|\mathcal{S}| \ll m$ *such that,*

$$\arg\max_{c_j \in \mathcal{S}} q^T w_j + b_j \in Y.$$

Algorithm 3 Offline Preprocessing

1: **Input:** Q**,** Y**,** W**,** HT**,** H**,** t_1**,** t_2

2: $P_+ = \{\}$, $P_- = \{\}$

3: **for** $i = 1 : N$ **do**

4: Compute $H(q_i)$.

5: S $= \{\}$

6: **for** $l = 1 : L$ **do**

7: $S = S\cup$ **Query**$(H_l(q_i),\ HT_l)$

8: **end for**

9: $P_+ = P_+ \cup \{(q_i, w_{y_i})|y_i \notin S, q_i^T w_{y_i} > t_1\}$

10: $P_- = P_- \cup \{(q_i, w_j)|w_j \in S \setminus y_i, q_i^T w_j < t_2\}$

11: **end for**

12: **shuffle** P_+, P_-

13: $g = min(|P_+|, |P_-|)$

14: $H^{'} \leftarrow$ **IUL**$(H, Pair_p[: g], Pair_n[: m])$

15: $HT^{'} \leftarrow$ **Rebuild**$(HT, H^{'})$

16: **Return** $HT^{'}$, $H^{'}$

5.2.4 Algorithm Overview

To construct the ideal retrieval oracle, which returns a Perfect Retrieval Set, we introduce *HALOS*. *HALOS* exploits Locality Sensitive Hashing (LSH) along with an efficient hash function learning procedure to approximate the computation in large output layer using label information from training dataset. Note that we choose a particular variant of LSH called SimHash [94], which is parameterized by hyperplanes in the dimensionality of the query. The full workflow is illustrated in Figure 5.2.

Algorithm 4 Online Inference

1: **Input:** q**,** W**,** b**,** $HT^{'}$**,** $H^{'}$

2: Compute $H^{'}(q)$

3: S = {}

4: **for** $l = 1 : L$ **do**

5: $S = S \cup$ **Query**$(H_l^{'}(q),\, HT_t^{'})$

6: **end for**

7: **Return** $\arg\max qW_S^T$

HALOS works in two separate stages: an offline preprocessing stage (Section 5.2.5) and an online inference stage (Section 5.2.6). We summarize the *offline* preprocessing as the following three steps: (1) Given any trained NN, we index last layer weights in hash tables(HT) via hash functions (H), which are randomly generated hyperplanes. (2)We leverage the NN training data to iteratively update H using **Index Updating Loss**(introduce in Section 5.2.5) based on the neurons retrieved from HT by each training data embedding query. (3) We rebuilt HT using new hash functions and store HT for online inference stage. We summarize the *online* inference phase as the following two steps: (1) Given input from the test set, we compute the forward pass until the output layer and get an embedding q. Then, we query the hash tables with q. (2) We set retrieved neurons as "active", and all other neurons as "inactive" for this input. Finally, we perform matrix multiplication on the "active" neurons and the top-ranked neurons (with highest logits) are returned as the prediction.

5.2.5 Offline Preprocessing: Index Output Layer Neurons

Following the standard LSH scheme, we construct L hash tables and each hash table has a capacity of 2^K bucket, where K is the number of binary hash functions per table (Each hash table key is concatenated by the K binary codes together). In total, we have $K \times L$ hash functions. Each neuron c_i can be represented by the concatenation of its weight and bias parameters $c_i = [w_i, b_i]$. For each $[w_i, b_i]$, we generate $K \times L$ hash codes, which constitute L hash table keys, and insert neuron's index i into L hash tables.

For each data in the training set, we collect its output from last hidden layer as embedding q and $[q, 1]$ is used as the embedding query to account for bias. For each embedding query, we generate L hash table keys and retrieve neuron indexes from L hash tables. For simplicity, we omit $[w, b]$, and $[q, 1]$ and directly use w and q in the following sections.

Every binary hash code for an input x is generated by function $f(x) = sign(\theta^T x)$, where θ is drawn i.i.d. from $\mathcal{N}(0, 1)$. This is equivalent to the method used in Simhash [94]. Geometrically, each θ represents a projection hyperplane in $\mathcal{R}^{d+1}$, such that the space is partitioned by K hyperplanes.

In order to possess the properties of a Perfect Retrieval Set, the hash functions should have the following properties: (1) the collision probability between the input embedding query and its ground truth label neuron is high, (2) the collision probability between the input embedding query and its non-label neurons is low (3) neurons are distributed evenly over all buckets for better load-balancing, which leads towards lower query overhead for efficiency purpose. We formally define our ideal hash function as the following:

Definition 2 (**Label Sensitive Hash Family**). *A hash family* $\mathcal{H}$ *is called* $(1, 0, p1, p2)$-*sensitive if for a triplet* $(q, x, y) \in \left(\mathbb{R}^{d+1} \times \mathbb{R}^{d+1} \times \{0, 1\}\right)$, *a hash function* h *chosen uniformly from* $\mathcal{H}$ *satisfies:*

- *if* $y = 1$ *then* $Pr(h(q) = h(x)) \geq p_1$
- *if* $y = 0$ *then* $Pr(h(q) = h(x)) \leq p_2$

We approximate hash functions from such a family using an iterative learning mechanism, which encourages the above three properties through **Index Updating Loss (IUL)** on pairwise data. Inspired by contrastive learning [95], the key to this learning process is the collection of positive and negative pairwise training samples. For each data embedding q from the NN training dataset, we retrieve its corresponding set of neurons $\mathcal{S}$ from the existing L hash tables. Then, pairwise training samples are collected according to the following criterion: (1)Positive pair consists of the input embedding, and its label class neuron which was not retrieved from current hash table. (2) Negative pair consists of the input embedding, and a retrieved, non-label neuron. Formally,

- Positive Pair $P_+ = (q, w_i)$

  if $c_i \in C \setminus S$ and $c_i \in Y$ and $q^T w_i > t_1$
- Negative Pair $P_- = (q, w_i)$

  if $c_i \in S$ and $c_i \notin Y$ and $q^T w_i < t_2$

Index Update Loss (IUL): IUL is based on the classic triplet loss [96] and we customize it specifically for updating hash table bucket assignment. The intuition behind IUL is that positive pairs are encouraged to land in the same bucket while negative pairs are allocated towards different buckets. We use Hamming distance as an approximation of the difference between the hash codes. Since *sign* is a discrete function, we use tanh as a differentiable approximation. We know that

$\text{dist}_{\text{hamming}}(q, w) = \frac{1}{2}(K - q^T w)$ [97].Formally,

$$\mathcal{IUL}(P_+, P_-) = \sum_{q_i, w_i \in P_+} -log(\sigma(\mathcal{K}(w_i)^T \mathcal{K}(q_i))) - \sum_{q_j, w_j \in P_-} log(1 - \sigma(\mathcal{K}(w_j)^T \mathcal{K}(q_j))),$$

where $\mathcal{K}(w) = \tanh(\theta^T w)$, $\mathcal{K}(q) = \tanh(\theta^T q)$, and $\sigma(x) = 1/1 + e^{-x}$.

Positive samples are used to maximize the probabilities of retrieving correct label neurons. Concurrently, it increases the relative ranking of label neurons on the inner product by decreasing the probabilities of retrieving other non-essential neurons. Negative samples are used to maintain a relatively small bucket size by pushing out low inner product neurons from the bucket. Otherwise, all the neurons would ultimately converge to the same bucket in each table. We collect the pairwise training data based on each retrieved set because it directly reflects the circumstances on how last layer neurons are separated by the current hyperplanes. t_1 and t_2 are two inner product ranking thresholds, that control the inner product quality of positive and negative pairs. Usually, we have $t_1 > t_2$ in any valid settings. Otherwise, in the situation that the NN predict a certain label neuron with a small logit, due to the nature of LSH, it would be challenging to train hash functions in the manner that low inner product neurons are retrieved while high inner product neurons are excluded.

Difference from Standard Learning for AMIPS: The positive and negative pair construction is essential. It should be observed that standard learning approaches, focused on AMIPS objective and use every positive and negative pair for training the hash function, potentially solving a harder problem. Instead, we only use the negative pairs arising from the buckets, and positive pairs missed by buckets if they are the correct labels. Overall, our training is aware of the current retrieval mechanism and only enforces what is needed for classification.

5.2.6 Online Inference

In this section, we introduce the online inference process. The NN parameters, hash functions, and hash tables from processing stages are frozen for deployment. For each test data, forward computation until the last layer is computed and the input embedding from last hidden layer is used for hash table query. Specifically, the $K \times L$ hash codes of the input embedding are generated using the stored hash functions. Based the hashcodes, a subset of output layer neurons is retrieved from the store hashed table. Here, we have a smaller output layer consisted of only retrieved neurons. Specifically, instead of performing the full computation over the output layer, we only compute the logits for each retrieved class, and the highest probability class among all retrieved classes are returned as the prediction. Note that unlike pruning, which set weights as zeros, we construct a much smaller output layer weight matrix and perform a smaller matrix multiplication.

Analysis on Computation Efficiency We focus on the output layer computation for the discussion here as computation before the output layer are not changed. With regular NN inference, the computation at output layer can be expressed as $\sigma(qW^T + b)$. Output layer weight matrix W is of dimension $m \times d$ where m is the size of output layer. With *HALOS* inference, the weight matrix is of dimension $k \times d$ where k is the number of neurons retrieved. The computation overhead *HALOS* introduced is the cost to compute $K \times L$ hash codes and hash table look up, which are known to be cheap. *HALOS* reduce inference cost by computing a much smaller matrix multiplication and nonlinear function, and usually, we have $\frac{k}{m} \leq 5\%$.

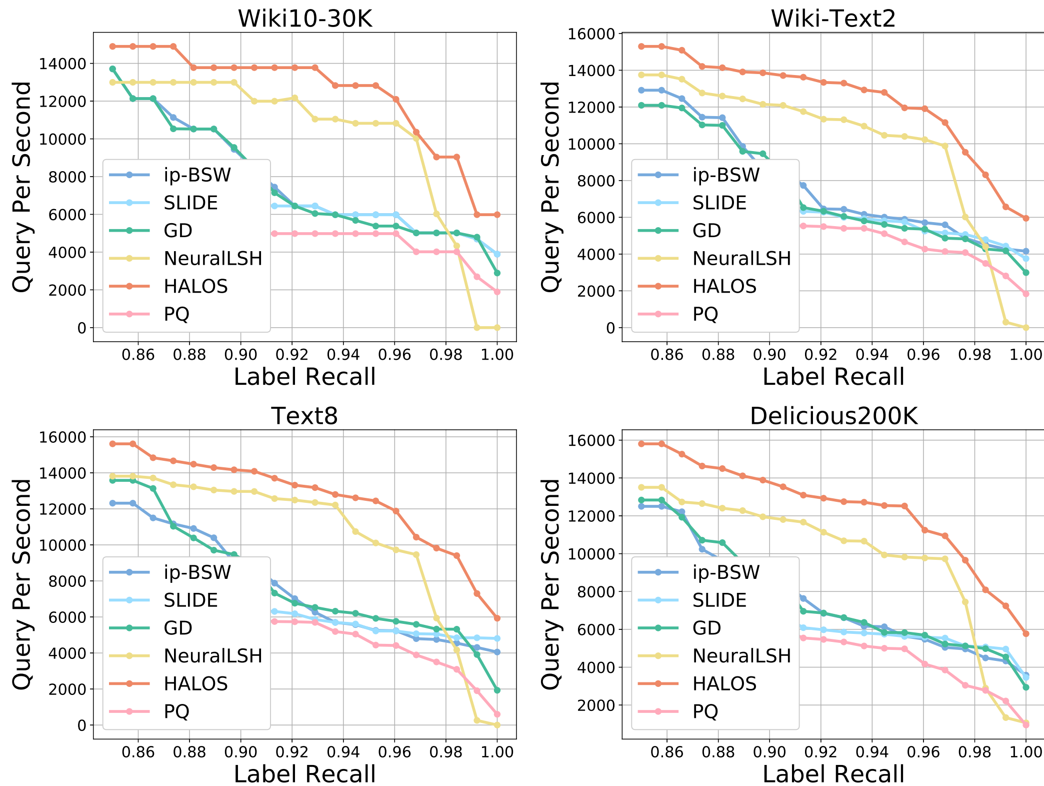


Figure 5.3 : Plot of Label Recall versus Query Per Second. On all four dataset, the line for *HALOS*(in red) is higher than all AMIPS baselines. Specifically, at every query speed, *HALOS* recalls more label neurons comparing to AMIPS baselines. This validates our arguments that naively applying AMIPS on the output layer is not sensitive to the label class, which may hurt model accuracy as shown in Table 5.1. Wiki-Text2 refers to the LSTN network.

5.3 Evaluation

In this section, we evaluate the effectiveness of *HALOS* method for efficient inference with large output spaces on two large scale extreme classification and two language modeling datasets. Specifically, we investigate from the following three perspectives (1) How does *HALOS* perform compared with established efficient inference baselines

in terms of the accuracy and efficiency trade-off? (2) How does *HALOS* perform in terms of optimizing for NN accuracy compared with AMIPS baselines? (3) How does *HALOS* perform compared to AMIPS baselines in terms of both query efficiency and inference efficiency?

Datasets and NN Models As we focus on NN with large output space, we choose two applications with such characteristics: language modeling and extreme classification, which concerns problems in webpage and production categorization, as well as webpage-to-webpage and product-to-product recommendation tasks [88]. For extreme classification, we use a standard fully connected neural network with one hidden layer of size 128. We evaluate on two datasets: Wiki10-31K [98] and Delicious200K [92]. For language modeling, we use a standard fully connected network with one hidden layer of size 128 for Text8 [99]. A two-layer LSTM network with hidden dimension of 200 is used for Wiki-Text2 [100],denotes as Wiki2-LSTM in the results. For mask language modeling, a 6 layer, 4 head Transformer model for Wiki-Text2,(denoted as Wiki2-Trans) and PTB [101]. The datasets are under MIT license. We refer the readers to Section 6.3.1 for more details.

Baselines: We compare *HALOS* with following approaches:(1) **Full** is the regular and paralleled NN inference using all neurons in NN last layer. (2) **SLIDE** [102] is a deep learning system utilizing locality-sensitive hashing for faster training, written in C++. We implement this method for inference. (3) **Graph Decoder** (GD) is an AMIPS method proposed for efficient Softmax inference in [40] that combines the asymmetric transform in [103] with HNSW [52]. Here we use the original implementation of HNSW [104] and pre-process the data according to [40]. (4) **ip-NSW** is a state-of-the-art graph-based AMIPS algorithm proposed in [105]. It belongs to the direct MIPS category and shows performance improvement over GD. (5)

Product Quantization (PQ) [106] is an AMIPS solver with K-means and asymmetric transformation. We implement this method following the popular open-source ANNS platform from Facebook [53]. (6) **NeuralLSH** [107] is a LSH based indexing with partition learning from the k-NN graph.

Implementation and Experiment Setting: Following standard NN inference procedure, we set test batch size as 1. All the experiments are conducted on a machine equipped with two 20-core/40-thread processors (Intel Xeon(R) E5-2698 v4 2.20GHz). The machine is installed with Ubuntu 16.04.5 LTS. HALOS for the output layer is written in C++ and compiled under GCC7 with OpenMP. The **Full** inference baseline is implemented in PyTorch. **GD**, **ip-NSW**, **PQ** are implemented in C++ with OpenMP. All implementation is parallelized with multi-threading with full usage of CPU cores. All baselines report the best results after a hyperparameter search.

Energy Measurement We further investigate efficiency from the energy consumption perspective. We measure the energy usage of each method using a monitoring tool. Command line utility tools, including *s-tui*, were used to monitor the CPU power consumption, in Watts (Joules / second), over the inference times for each dataset and each method, in intervals of 1 second. The base power consumption would be subtracted from the average power of the inference time, in order to measure and compare the energy expenditure of only the inference step of each method.

Evaluation Metric: We compare HALOS against other baselines on the following evaluation metrics: (1) **p@k** for top k accuracy classification tasks. (2) **Label recall** is the times of retrieval set including label neurons divided by total number of testing data.(3) **Time** is measured as the average wall-clock time for passing one testing data through the last layer in milliseconds. (4) **Energy**, measured in Joules, is the average CPU power over the inference period, multiplied by the inference time. It is then

averaged over each testing data.

5.3.1 Main Result

Table 5.1 and Table 5.2 summarize inference accuracy, wall-clock time and energy usage for HALOS and all other baselines. We report the performance in Table 5.1 when each method achieves maximum retrieval efficiency, denoted as the ratio between p@1 and time usage. This selection criterion is chosen to reflect the balance between accuracy and efficiency. We observe that: (1) Comparing to full inference, HALOS achieves up to 20× reduction in time and 8.2× reduction in energy consumption with no or minor accuracy drops. Note that on Text8 and Wiki2-LSTM, HALOS achieves both higher than full-inference accuracy and significant time and energy reduction.(2) HALOS achieves the highest label recall compared to all other baselines on most datasets. (3) HALOS achieves the highest accuracy, lowest time, and energy usage compared to other AMIPS baselines.

Discussion: [40, 108] compared the performances of different methods under a single CPU thread setting, which is not a practical simulation for real-world cloud systems. Moreover, despite their decent performance on a single thread, methods such as ip-NSW or GD are ill-suited to exploit the full parallelization offered by multi-core CPUs and tend to have a large number of irregular memory accesses. This limitation significantly degrades their performance even compared to exact MIPS computation on CPU with the current deep learning framework such as PyTorch.

5.3.2 Study on *HALOS* for Label Sensitivity

Both *HALOS* and baselines have tuning parameters that control the accuracy-efficiency trade-offs. Figure 5.3 plots Label Recall vs. Query Per Second(QPS) for *HALOS*

and AMIPS baselines. At every QPS level, *HALOS* retrieves more label neurons. Meanwhile, from Table 5.1, AMIPS baselines, which target on the largest inner product, have a lower label recall rate on all datasets. This phenomenon validates our arguments regarding the shortcomings of applying AMIPS methods in NN Inference. AMIPS methods struggle in retrieving the label neuron at a given time budget. And failing to retrieve the labels directly leads to incorrect prediction.

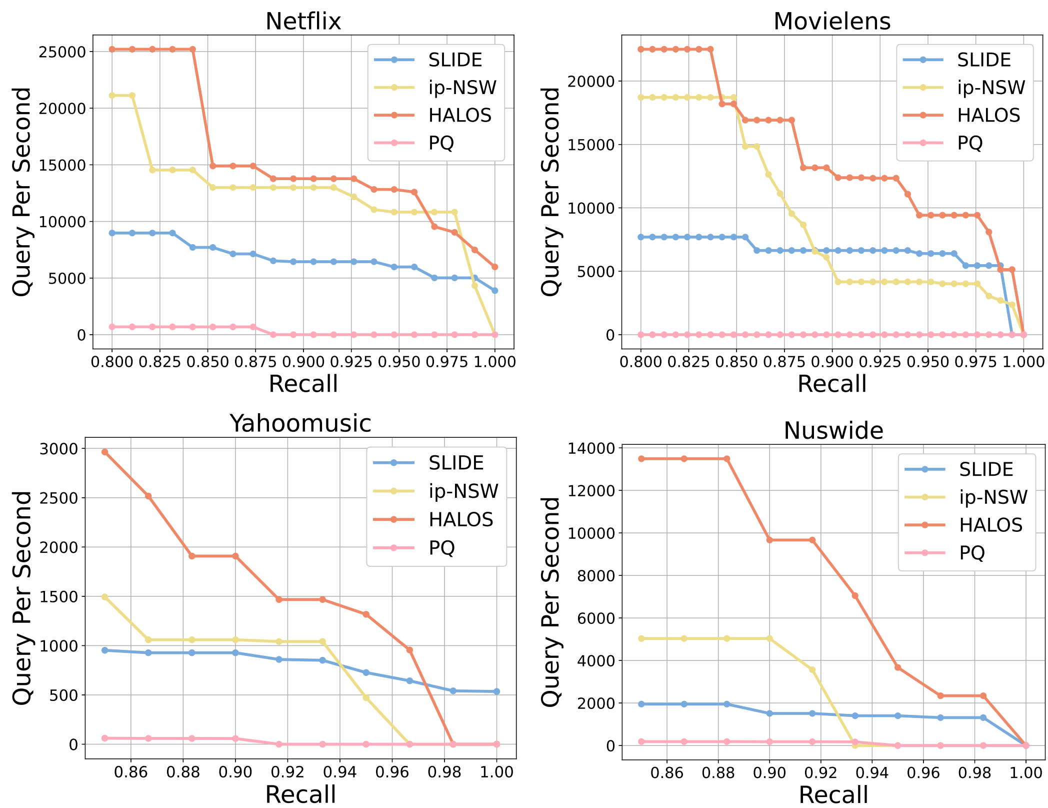


Figure 5.4 : Plot of Recall versus Query Per Second. The lines representing *HALOS*(red) is the highest across datasets. At every recall level, *HALOS* processes more data, leading to better query efficiency comparing to AMIPS baselines.

5.3.3 Study on *HALOS* for Efficiency

The major overhead introduced by *HALOS* in inference efficiency lies in the query time. Large overhead may cancel out the computation savings from smaller matrix multiplication and even result in longer than regular inference time, as we observe in Table 5.1 for some baselines. In this section, we purely focus on *HALOS*'s query efficiency in perfect model settings, where the MIPS results correspond to the label.

Setting: We use four recommendation datasets and the task is to directly retrieving the item that yields the largest inner product between user and item embedding. To generate user and item embedding, we use deep matrix factorization (DMF) [81] for Movielens and Netflix dataset, and alternating least squares (ALS) based MF method [109] for Yahoomusic and Nuswide dataset. Then, the preference of an user towards an item is described by their inner product in the embedding space.

Analysis: Figure 5.4 shows the standard Recall vs. Query Per Second(QPS) plots. The line for *HALOS* lies consistently above other baselines, which means *HALOS* processes more data at every recall level. *HALOS* outperforms Random LSH on all datasets, which validates our arguments that learned hashing functions could better adapt to data distribution and improve query efficiency. *HALOS* also achieves up to $4\times$ QPS compared to ip-NSW and PQ, which validates our choice of hashing tables as indexing data structures. Despite lower time complexity during the preprocessing phase, hash table look-up operations are faster than the greedy walk on tree or graph structures in the query phase. Moreover, hashing methods are more amenable towards less computation and multi-threading [110] compared to MIPS solvers [91] and graph methods [105, 40].

One common concern about the hash table is its memory usage. However, since we only store neuron indices, memory usage is limited. For example, the output

dimension of Delicious-200K is 205,443. Each integer cost 4 bytes, and assuming we set L as 10 (same parameters as our main result), the total memory usage is only 8MB.

5.3.4 Ablation Studies on *HALOS*

Collision Probability: Collision probability denotes the probability that a pair of inputs are hashed to the same bucket. Figure 5.5 shows the collision probability for both positive and negative pairs we collect during hash function training. On Text8 and Delicious200K, collision probability between positive pairs increases and converges to a level above 0.9. Meanwhile, collision probability between negative pairs decreases throughout the training process. On Delicious200K, the collision probability of negative pairs converges to around 0.1. On Text8, it converges close to zero. These plots confirm that the proposed hash functions learning mechanism adjusts the randomized initialized hyperplanes to index input embedding and its label neurons into the same bucket with a much higher probability. This observation explains *HALOS*'s high label recall in Table 5.1.

Influence of Hyper Parameters: K, the number of hashes, and L, the number of hash tables, are two key parameters to balance *HALOS*'s retrieval quality and efficiency. In general, fewer hash tables improve efficiency by reducing query time and retrieval size, while more hash table improves retrieval quality and help with accuracy. Here retrieval size represents the average number of items returned by the hash tables in each query. Fewer hashes decrease hash code computation but increase retrieval size(including unnecessary neurons). More hashes increase hash code computation but decrease retrieval size(missing important neurons with high probability). In Table 5.3, we summarize the effects of varying K and L on Delicious200K. $L = 1$ and $K = 4$

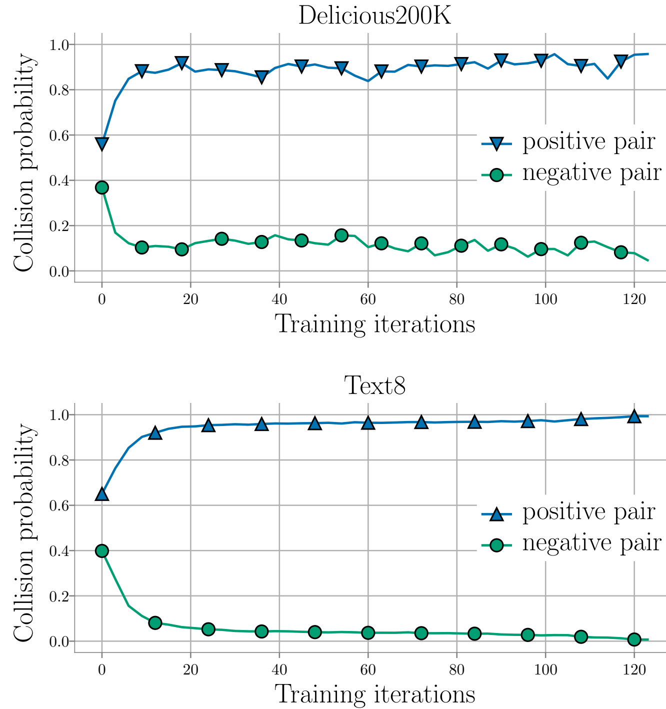


Figure 5.5 : Collision probability is measured as the probability a pair of inputs hashed in the same bucket. Blue line plots the collision probability between positive pairs. Green line plots the collision probability between negative pairs. The starting point represents the probabilities with randomized initialized hashing functions. It is evident that the proposed learning mechanism pushes positive pairs to land in the same bucket while seperates negative pairs. And since positive pairs are collected as input embedding with its label neurons, *HALOS* is more sensitive to label class and achieve high label recall.

(parameters for our main result) have the lowest accuracy but lead to the smallest overhead and the most computation reduction: it requires less hash code computations (determined by K), fewer hash table lookups (determined by L), and smaller last layer matrix multiplication (determined by retrieval size). Different choices of K and L

have limited effect on $P@1$ and $P@5$((up to 3.5% on $P@1$ and 1.3% on $P@5$).

5.3.5 Study on *HALOS* for Accuracy

Our hashing learning mechanism was designed for the purpose of constructing the Perfect Retrievel Set as described in SectionSection 5.2.4. In this section, we deviates from focusing on efficiency, purely investigate the possibility of achieving higher accuracy than regular Softmax inference. Table 5.4 summarizes the highest accuracy *HALOS* achieves during our hyper parameter search. On Delicious200K and Wiki-Text2 for language modeling, *HALOS* outperforms full inference accuracy by up to 2%. The major reason behind this increase is that the probability of predicting the correct class from a subset of neurons is naturally higher than the probability of predicting the correct class from the entire output space. Specifically, we observe a larger retrieval size to surpass regular inference accuracy. A larger retrieval size along with the query overhead may not lead to efficiency gain but may be of the separate interest for accuracy.

Method	Dataset	Accuracy	Label Recall	Avg.Time(ms)	Avg.Energy(10^{-3} J)
Full	Wiki2-LSTM	0.4044	1	0.63	9.31
HALOS.	Wiki-2LSTM	**0.4265**	**1**	**0.36 (1.7x)**	**3.20 (2.9x)**
PQ	Wiki2-LSTM	0.2234	0.6654	10.57	128.76
ip-NSW	Wiki2-LSTM	0.3995	0.8705	1.60	23.92
GD	Wiki2-LSTM	0.1369	0.9215	1.76	27.07
NeuralLSH	Wiki2-LSTM	0.2299	0.8826	1.99	39.98
SLIDE	Wiki2-LSTM	0.3309	0.8695	3.33	45.54
Full	Text8	0.9129	1	1.88	31.99
HALOS.	Text8	**0.9132**	**1**	**0.56 (3.3x)**	**4.98 (6.4x)**
PQ	Text8	0.6138	0.6084	12.99	100.26
ip-NSW	Text8	0.8299	0.8977	2.07	22.66
GD	Text8	0.9129	0.9908	2.09	20.40
NeuralLSH	Text8	0.8990	0.9011	2.55	28.44
SLIDE	Text8	0.6517	0.9799	3.92	29.33
Full	Wiki2-Trans	0.8538	1	5.66	8.77
HALOS.	Wiki2-Trans	**0.8519**	**0.9993**	**0.28(20x)**	**1.66(5.28x)**
PQ	Wiki2-Trans	0.7631	0.8566	7.02	10.33
ip-NSW	Wiki2-Trans	0.8164	0.9661	0.37	4.02
GD	Wiki2-Trans	0.8221	0.9787	0.39	3.94
NeuralLSH	Wiki2-Trans	0.8300	0.9804	0.36	3.98
SLIDE	Wiki2-Trans	0.8056	0.9511	0.49	2.33
Full	PTB	0.8534	1	0.55	7.42
HALOS.	PTB	**0.8531**	**0.9997**	**0.32(1.7x)**	**1.33(5.6x)**
PQ	PTB	0.7442	0.8777	2.67	8.98
ip-NSW	PTB	0.8325	0.9903	0.52	2.34
GD	PTB	0.8401	0.9908	0.54	2.19
NeuralLSH	PTB	0.8231	0.9775	0.51	2.47
SLIDE	PTB	0.7799	0.8994	0.60	2.66

Table 5.1 : This table summarizes the performance of HALOS and other baselines on language datasets.

Method	Dataset	Accuracy	Label Recall	Avg.Time(ms)	Avg.Energy(10^{-3} J)
Full	Delicious200K	0.4391	1	4.16	71.34
HALOS	Delicious200K	0.4245	**0.889**	**0.81 (5.1x)**	**8.7 (8.2x)**
PQ	Delicious200K	0.3547	0.6677	9.33	113.22
ip-NSW	Delicious200K	0.4122	0.6900	2.66	31.66
GD	Delicious200K	**0.4362**	0.7000	2.29	29.05
NeuralLSH	Delicious200K	0.3928	0.6450	3.33	30.21
SLIDE	Delicious200K	0.4024	0.8542984	5.12	37.75
Full	Wiki10-30K	0.8232	1	0.76	10.69
HALOS.	Wiki10-30K	0.8018	0.9779	**0.39 (1.9x)**	**3.53 (3.0x)**
PQ	Wiki10-30K	0.6766	0.8905	4.06	39.28
ip-NSW	Wiki10-30K	0.7703	0.8854	1.65	15.75
GD	Wiki10-30K	0.7636	0.9163	1.69	15.80
NeuralLSH	Wiki10-30K	0.7001	0.9000	0.99	19.77
SLIDE	Wiki10-30K	**0.8079**	**0.9995**	6.22	25.45

Table 5.2 : This table summarizes the performance of HALOS and other baselines on recommondation datasets.

		K = 4	K = 6	K = 8
L = 1	P@1	0.4245	NA	NA
	P@5	0.3473	NA	NA
	Retrieval Size	424	0	0
L = 10	P@1	0.4602	0.4488	0.4408
	P@5	0.3676	0.3733	0.3598
	Retrieval Size	2560	875.53	153.31
L = 50	P@1	0.4405	0.4455	0.4457
	P@5	0.3659	0.3599	0.3615
	Retrieval Size	15568	2122.47	360

Table 5.3 : Effect of L and K on P@1 and P@5 on Delicious-200K. K is the number of hash functions, L is the number of hash functions. Retrieval Size measures the number of output layer neurons retrieved from hash tables.

Dataset	Wiki10-31k	Delicious200k	Text8	Wiki2-LSTM
HALOS p@1	0.82	**0.46**	0.913	**0.427**
HALOS p@5	0.48	**0.37**	0.740	**0.084**
Retrieval Size	2372	2560	965	3071
Full p@1	0.82	0.44	0.913	0.404
Full p@5	0.57	0.36	0.740	0.077

Table 5.4 : This table summarizes the highest accuracy of *HALOS* on four dataset. Bold indicates that *HALOS* is higher than the full accuracy shown in Table 5.1.

Chapter 6

Retaining Knowledge for Learning with Dynamic Definition

6.1 Introduction

Motivation: A common machine learning pipeline is to first define a task objective based on historical experience, then deploy a model trained to optimize the objective over collected data. However, machine learning models rarely operate in such a static environment. Instead, models must be dynamically updated to accommodate changes in class definitions without forgetting previous knowledge. Consider the case of fraud detection, where models must adapt to emerging scam patterns while also providing protection against previously-known patterns. Machine learning models are also used to identify harmful microbes given gene sequences, where it is important to perform well on new variants or newly discovered microorganisms. A similar situation arises when models are used to filter inappropriate content such as hate speech and violence. New phrases or previously-innocuous language may become inappropriate due to world events or attempts by users to circumvent the filter. While the classification task remains the same (fraud or safe, harmful or not, inappropriate or appropriate, etc.), class definitions evolve over time and likely differ significantly from the point of the initial training. We refer to this setting as *Learning under Dynamic Definition* (LDD). It should be noted that this notion is very different for several other notions of distribution shift in the literature, which we review in the next section.

A standard solution is to retrain the model on all available data (original dataset and data for the new definitions). This methodology is inefficient due to data storage and extra computation, and even prohibitive when dealing with large models and streaming data. Furthermore, it prevents real-time and on-device learning. An ideal and efficient solution is to update the trained model incrementally only with data for the new definitions (e.g. using online gradient descent). However, this is a challenging process due to the tendency of machine learning models, in particular neural networks, to forget previous knowledge. This situation, known as collapse/catastrophic forgetting [111], causes performance decay on the old definitions.

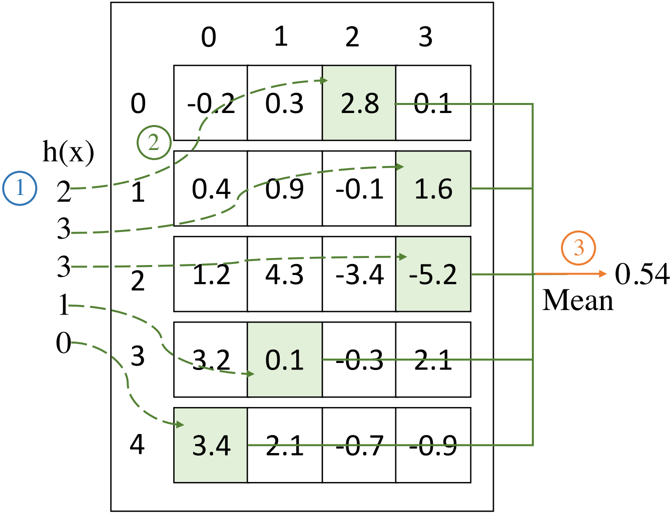


Figure 6.1 : The figure demonstrates a toy RIDDLE model with $L = 5$ and $R = 4$. Given an input x, we calculate its hash code $h(x)$ and access the weights at those locations. We return the average as the output.

Our Proposal: Catastrophic forgetting occurs because gradient updates from training examples in the update process are able to change parameters that are tuned for optimal performance on the original dataset [112]. Inspired by recent connections between random partitions and distribution models [5], we propose RIDDLE (Retaining Information in a Dynamically Defined Learning Environment), a novel architecture with meaningful parameter groupings. RIDDLE works by partitioning the input space

and assigning a set of parameters to each partition. Gradient updates are restricted only to locally relevant parameters, identified by cheap hashing computations. RIDDLE can be viewed as a Mixure-of-Expert-Layer where the gating function is Locality Sensitive Function (LSH) and each expert is just one number. At the same time, RIDDLE can be viewed as a kernel method where the prediction is an efficient weighted kernel density estimation. RIDDLE can be plugged into diffirent neural network architectures, just like fully connect layers, to ease the memory loss while utilizing the learned features, as shown in Figure 6.2. We summarize our technical contributions below.

- We introduce the RIDDLE (in Section 6.3), consisting of a set of parameters indexed by locality-sensitive hash functions. RIDDLE is computationally efficient and requires less memory than popular machine learning models with similar accuracy.
- We prove that RIDDLE is universal function approximator and can therefore represent any function(Section 6.4). We prove a formal connection between parameter usage and distribution similarity in Theorem 13.
- In Section 6.6, we apply RIDDLE in the dynamic definition setting for four real-world tasks. Our method outperforms baselines by up to 30% on the original dataset while achieving competitive accuracy on the new dataset.

6.2 Related Work

Continual Learning: Continual learning refers to learning and remembering multiple tasks from a stream of information. Current approaches can be split into three main categories (1) memory replay to complement neural networks [113, 114], (2) constraints on network updates [115, 116, 117], and (3) dynamic architectural adjustments to

accommodate new information, such as additional layers [118, 119]. Recently, few-shot learning methods are also applied into continue learning to solve sequential learning with limited data [120, 121, 122]. Continual learning methods encounter similar challenges to the ones presented by learning under dynamic distribution, most notably catastrophic forgetting. However, continual learning attempts to solve a sequence of related tasks, often expressed as task embeddings, over data that does not necessarily reflect changes in class definitions.

Train-Test Distribution Shift: Transfer learning and test-time adaptation attempt to remedy misalignments between the training and test distribution. Although such methods may seem relevant in learning with dynamic definition, they have dramatically different goals. Transfer learning attempts to adapt a model from a data-rich source domain to a data-scarce target domain, without regard for the performance on the source domain [123, 124, 125]. While several works on train-test distribution shift focus on robustness, it is from the perspective of out-of-distribution detection [126, 127] or uncertainty estimation [128, 129, 130]. Our objective is to provide models that perform well on both the original dataset and new dataset, where class definitions are different, rather than simply detect or adapt to the distribution change.

Online Learning: Online learning regards optimization as a process where a model is trained by interacting with a stream of data. Online learning algorithms attempt to minimize the *regret*, or difference between the loss of the predictions by the incrementally-trained model and the loss of the globally optimal model [131, 132]. Online learning methods attain minimal regret against adaptive adversaries and could be expected to perform well in the dynamic distribution setting. However, the best performing methods for non-convex online learning require retraining over all previous

data at every time step [133], making them practically unsuitable for the tasks we consider.

6.3 RIDDLE for Learning with Dynamic Definition

Learning with Dynamic Definition: We formally define the problem as follows.

Definition 4. *Given a model $f(x;\theta)$ and a loss $\ell(f(x;\theta),y)$, consider an original dataset $D^o = \{(x_i^o, y_i^o)\}_{i=1}^m$ drawn i.i.d. from a distribution P^o and an update dataset $D^u = \{(x_i^u, y_i^u)\}_{i=0}^n$ drawn i.i.d from distribution P^u. Let θ^o be the parameters that optimize $\mathcal{L}_o(\theta) = \sum_{D^o} \ell(f(x;\theta),y)$. The dynamic definition learning problem is to find a solution minimizing $\mathcal{L}_{o+u}(\theta)$ when we are only given access to θ^o and D^u.*

$$\mathcal{L}_{o+u}(\theta) = \sum_{(x,y)\in D^o} \ell(f(x;\theta),y) + \sum_{(x,y)\in D^u} \ell(f(x;\theta),y)$$

In this paper, we consider the model performance in response to the changes from P^o to P^u, specifically by allowing x to be drawn from different input spaces. Note that Definition 4 generalizes in a straightforward way to sequences of dataset updates by considering D^o to be the union of all prior data. The difficulty of the problem may be expressed in terms of the Kullback-Leibler divergence $D_{KL}(P^o \| P^u)$. Intuitively, small changes in distribution are easier to handle because θ need not change substantially to perform well on D^u.

6.3.1 Architecture

RIDDLE is a 2D parameter array S that contains L rows and R columns $S \in \mathbb{R}^{L\times R}$. The model is indexed by a set of L randomly generated LSH functions $h_l \rightarrow \{1, \dots R\} \in \mathbb{Z}$ for $l = 1, \dots L$. The prediction evaluated at a query x is defined as following:

$$f(x;S,h) = \frac{1}{L}\sum_{l=1}^{L} S[l, h_l(x)]$$

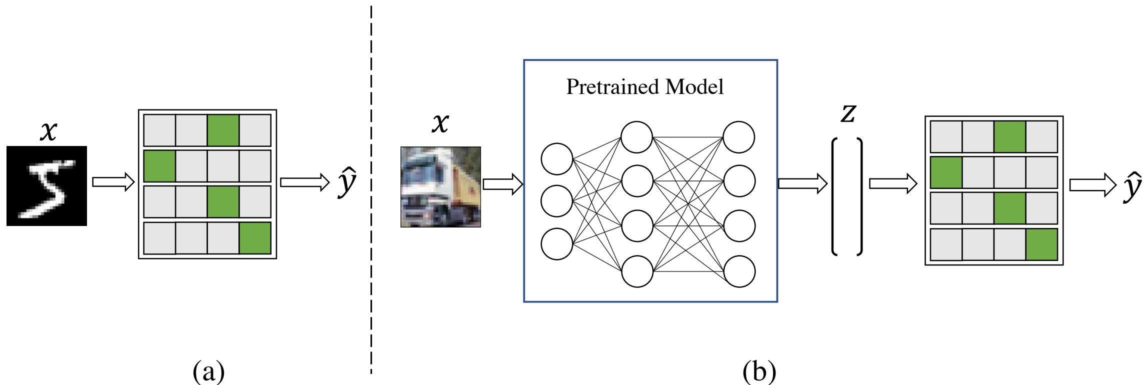


Figure 6.2 : RIDDLE can be used as a stand-alone model, as shown in (a). We use this setting for experiments on MNIST-binary and all experiments studying expressiveness and efficiency. RIDDLE can also be combined with existing neural networks, as shown in (b). We replace the classification layer with RIDDLE for experiments on CIFAR10, ImageNet, and News

Algorithmically, the process consists of three steps: (1) hash code computation, (2) partition look-ups, and (3) output averaging. Specifically, the model output for a query x is an average over the L parameters retrieved from S using the hash codes $h(x)$.

For a classification problem with C classes, the 2D array is repeated C times, resulting in a 3D tensor of parameters $S \in \mathbb{R}^{C \times L \times R}$. The same LSH functions are used for all C repetitions, and the classification prediction for query x is defined as follows:

$$f(x; S, h) = \text{argmax}_{c \in C} \left(\frac{1}{L} \sum_{l=1}^{L} S[c, l, h_l(x)] \right)$$

RIDDLE can be used as a stand alone machine learning model on simple datasets but can also be combined with existing neural networks. Specifically, RIDDLE can use the learned representation from neural network (Figure 6.2). RIDDLE can

also be implemented in standard deep learning frameworks and trained using SGD (Algorithm 5). It should be noted that the parameters of LSH functions $h(x)$ are not trainable. Only the values in $S \in \mathbb{R}^{L \times R}$ will be updated by SGD. The inference process consists only of inexpensive hash evaluations and memory look-ups. We use LSH functions based on sparse projections [134], which compute $h(x)$ via an inner product with parameter vectors randomly drawn from $\{-1, 0, 1\}$ with probability $\{1/6, 2/3, 1/6\}$. With such a hash function, computation can be reduced to addition, subtraction and memory lookup.

6.3.2 Intuition

RIDDLE may seem conceptually different from neural networks at first gasp, however, RIDDLE can be viewed as a combination of Mixture-of-Expert-Layers [135] with Kernel Methods [136]. To understand RIDDLE as a mixture of experts, observe that the gating function for every row in RIDDLE is a randomly initialized hashing function h_i, and the expert is one number stored at $S[i, h_i(x)]$. RIDDLE can be viewed as an ensemble of L such Mixture-of-Expert-Layers. On the other side, RIDDLE is also a kernel density estimation process where the LSH collision probability is the kernel function. The number stored at $S[i, h_i(x)]$ is an efficient estimation of the weighted kernel values between the test sample and training samples.

6.3.3 RIDDLE for Learning with Dynamic Definition

Before we determine the class of functions that are representable using our model, we first demonstrate that this architecture addresses the problem from Definition 4. We consider a RIDDLE model $f(x; S, h)$ following the notation from Section 6.3.1. That is, we are given a model with parameters $\theta = (S, h)$ trained on D^o. We wish to adapt

Algorithm 5 RIDDLE Training

Input: Training Dataset D, $|D| = m$, number of rows L, number of cells R, learning rate η, error function $E(\cdot)$, number of epochs e, batch size b, random seed s.

Output: Trained Model S, Counters C

Initialize: Zero initialize $S \in \mathbb{R}^{L\times R}$; Zero initialize counters $C \in \mathbb{R}^{L\times R}$; Randomly generated L independent LSH functions $h_1, \ldots, h_L$ using random seed s;

for $i = 1 \to e$ **do**

  Shuffle dataset D

  for $j = 0 \to m/b - 1$ **do**

    Take b samples $\{x_{jb}, x_{jb+1}, \ldots, x_{(j+1)b-1}\}$ with labels $\{y_{jb}, y_{jb+1}, \ldots, y_{(j+1)b-1}\}$

    Compute gradient $g = \frac{1}{m}\nabla_S \sum_i E(f_{S,h}(x_i), y_i)$

    Apply update $S \to S - \eta g$

  end for

end for

for every $x_i \in D$ **do**

  Increase counters $C[l, h_l(x_i)] += 1$ *for* $l = 1 \ldots L$

end for

the model on D^u using gradient descent.

Parameter Separation: The RIDDLE partitions parameters in such a way that inputs from different distributions are likely to use different parameters. The parameter update may be written as:

$$S_{t+1} = S_t[l, h_l(x)] + \eta g[l] \text{ for } l = 1 \ldots L, \text{ where } g = \nabla \ell \left(\frac{1}{L} \sum_{l=1}^{L} S[l, h_l(x)], y \right)$$

From this expression, we can see that L parameters are involved in the computation of

$f(x; S, h)$ - specifically, the parameters selected by the hash functions $\{h_1(x)...h_L(x)\}$. Based on the LSH property (Definition 1), we expect unrelated inputs to have different hash values and therefore use different parameters. When P^u is significantly different from P^o, $x^u \in D^u$ and $x^o \in D^u$ are likely to activate different partition and thus use different parameter sets.

We formalize this intuition below by showing that the difference between $f(x; S, h)$ and its initialization is bounded by an estimate of the distribution P^o.

Theorem 6. *Given a dataset D, train a RIDDLE model $f(x; S, h)$ by running Algorithm 5 for e epochs with learning rate η and initialization S_0. Suppose the gradient norms are bounded by G. Then*

$$|\mathbb{E}_h[f(x; S, h) - f(x; S_0, h)]| \leq e\eta G \, \mathrm{KDE}(x, D)$$

where $\mathrm{KDE}(x, D)$ is the kernel density of x over D using the kernel of the LSH function h.

Theorem 13 allows us to prove the following guarantee:

Theorem 7. *Using the notation from Definition 4, let S_o be the sketch that optimizes the loss $\mathcal{L}_o(S)$ over D^o and S_{o+u} be the sketch obtained by training over D^u for e epochs with learning rate η and initialization S_o. If the loss is L-Lipschitz with G-bounded gradients, then:*

$$\mathbb{E}_h[\mathcal{L}_o(S_{o+u}) - \mathcal{L}_o(S_o)] \leq \sum_{x \in D^o} GLe\eta \mathrm{KDE}(x, D^u)$$

The loss difference in Theorem 7 quantifies the amount of information about D^o that is forgotten by the model when trained on D^u. The excess loss on example $x \in D^o$ is bounded by the kernel density estimate $\mathrm{KDE}(x, D^u)$. When D^o and D^u are drawn

from different distributions, which is the characteristic of learning under dynamic distribution, the density becomes small (i.e. as the divergence $D_{KL}(P^o \| P^u) \to \infty$, $\text{KDE}(x, D^u) \to 0$).

Parameter Importance: The RIDDLE allows us to determine the importance level of each parameter for encoding information from the original distribution with simple counting. We may do this by counting the number of inputs from D^o that fall into each partition, using a set of counters $C \in \mathbb{R}^{L \times R}$. Parameters with a large counter c can be regarded as important for the original distribution, as they participate in prediction for a large fraction of examples from D^o. Our intuition is that parameters corresponding to partitions with large count values are critical for performance on D^o and should not be adjusted. We use this idea to improve performance via partition-dependent learning rates. Specifically, we allow the learning rate $\eta(c)$ depend on the partition count c. We find that the following update rule provides good empirical results:

$$\eta(c) \propto \frac{\bar{c} + \alpha}{c \log c + 1}, \text{ where } \alpha \text{ is a constant}$$

6.4 RIDDLE is a Universal Function Approximator

In this section, we analyze the class of functions that can be represented using the RIDDLE. Using techniques from the sketching literature, we prove a probabilistic guarantee.

Intuition: To see why it is reasonable for our model to be a universal function approximator, observe that the function learned by RIDDLE is a piecewise-constant spline function. Splines are an incredibly powerful representation that can approximate any function, given sufficient flexibility in choosing the partitions (or knots) of the spline [137, 138]. For example, many neural networks can be written as compositions

of piecewise-linear splines over a constrained set of partitions[139, 140, 141, 142]. In our case, the partition boundaries are determined by the intersection of LSH functions. With sufficient independent overlapping partitions (large L), our model attains a high-quality representation.

Proof Sketch: We prove that the expected output of our model is a universal function approximator, where the expectation is taken over the choice of LSH functions. We begin with a proof that any continuous and bounded function can be approximated arbitrarily well by a weighted kernel sum over an N point dataset. We then develop a process that constructs a RIDDLE $S \in \mathbb{R}^{L \times R}$ whose expected output is this kernel sum. Using Chernoff bounds, we show that the error between the model and the kernel sum becomes arbitrarily small as $L \to \infty$ (with high probability). Taken together, we have a probabilistic guarantee for the error between the RIDDLE and any continuous and bounded function.

Simplified optimization in practice: Constructing a RIDDLE over a carefully chosen N point dataset can approximate any continuous and bounded function. However, learning both the data and data weights with N being arbitrarily large poses difficulty in optimization. We simplify the optimization by observing that the output of the learning process is not a dataset but a model. Therefore, we consider the easier task of learning the model parameters values directly in practice as shown in Algorithm 5. This is a slight relaxation of the problem, however, we argue that with a row-sum constraint and a reasonable uniqueness assumption about the hash partitions, the two processes are equivalent in practice. Thus, the RIDDLE is a universal function approximator.

6.4.1 Weighted LSH Kernel Sums are Universal Function Approximators

We are interested in the representation capabilities of weighted kernel sums over an N-point dataset $D = \{x_1, ...x_N\}$. We consider the specific case where the kernel is the collision probability of an LSH function. Such kernels are referred to in the literature as *LSH kernels* [5].

$$f(x) = \sum_{i=1}^{N} \alpha_i \mathcal{K}(x, x_i)$$

The crucial property is that LSH kernels are *universal*. Informally, a kernel is universal if a weighted kernel sum over a carefully-chosen point set can approximate any function with arbitrary accuracy [143]. Some restrictions apply; the size of the point set is allowed to be arbitrarily large and the accuracy is computed over any compact subset of $\mathbb{R}^d$ rather than the entire space.

Definition 5. *Universal Kernel [143]: Let $S(\mathcal{X})$ denote the space of bounded and continuous functions on a compact domain $\mathcal{X}$. Then a kernel $\mathcal{K}(x, y)$ is universal if it is continuous and induces a reproducing kernel Hilbert space that is dense in $S(\mathcal{X})$.*

Applied to our problem, a universal kernel is one which can approximate any well-behaved function over compact subsets of $\mathbb{R}^d$ using a linear combination of kernels. The theory of universal kernels allows for a convergence-style proof for weighted LSH kernel sums. We begin by showing that the p-stable LSH functions are universal kernels. We defer proofs to the appendix.

Lemma 8. *The L2 LSH kernel [5] induced by the p-stable LSH function [3] is shift-invariant and universal.*

Theorem 9. *Given a continuous and bounded function $g(q)$ and any $\epsilon > 0$, there*

exists a set of coefficients $\{\alpha_n\}$*, set of points* $\{x_n\}$ *and an integer* N *such that*

$$f_N(q) = \sum_{n=1}^{N} \alpha_N \mathcal{K}(x_n, q) \qquad \|f_N(q) - g(q)\|_{\mathcal{X}} \leq \epsilon$$

where $\mathcal{K}(x_n, q)$ *is the L2 LSH kernel and* $\mathcal{X}$ *is any compact subset of* $\mathbb{R}^d$*.*

6.4.2 RIDDLE for Weighted LSH Kernel Sums

The next step of our proof is to construct a RIDDLE capable of approximating $f_N(q)$ from Theorem 9. We consider a RIDDLE model with $S \in \mathbb{R}^{L\times R}$ constructed from a set of points $\{x_n\}$ with coefficient $\{\alpha_n\}$ in the following way. For every data point $x \in \{x_n\}$, we calculate the indices of x using a collection of L2 LSH functions $h_l(x)$ for $l = 1, \ldots, L$. Then, we increment the value of S at location $(l, h_l(x)$ with the weight α. To query the model, we calculate the index of the query q using the same LSH functions and retrieve the values $S[l, h_l(q)]$ for $l = 1, \ldots, L$, which we aggregate and return. It should be noted that recent algorithms from the sketching literature propose similar constructions [5, 144], but with the weights constrained to $\alpha_n \in \{+1, -1\}$.

One can prove S yields a sharp unbiased estimator for the weighted kernel sum (Proofs in appendix).

Theorem 10 (RIDDLE Estimator)**.** *Given a dataset* $\mathcal{D}$ *of weighted samples* $\{(\alpha_{x_i}, x_i)\}$*, let* $h(x)$ *be an LSH function drawn from an LSH family with collision probability* $\mathcal{K}(\cdot,\cdot)$*. Let* S *be a 2d parameter array constructed using* $h(x)$*. For any query* q*,*

$$\mathbb{E}(S[h(q)]) = \sum_{i}^{|\mathcal{D}|} \alpha_{x_i} \mathcal{K}(x_i, q), \mathrm{var}\left(S[h(q)]\right) \leq \left(\sum_{i}^{|\mathcal{D}|} \alpha_{x_i} \sqrt{\mathcal{K}(x_i, q)}\right)^2$$

The key insight is that each row of S is an unbiased estimator for weighted sums of LSH kernels. By finding the central tendency of the L rows, we can obtain a sharp

concentration around $f_N(q)$. Although we often use the average to do the aggregation in practice, we analyze the median-of-means estimator [145, 9], as it allows us to prove an exponential concentration of the estimate around the weighted KDE. Note that while this simplifies the proof, it does not affect the convergence to $f_N(q)$ as $L \to \infty$. We combine the variance bound from Theorem 10 with the median-of-means guarantee to obtain a relationship between the RIDDLE parameters S, weighted kernel values, and estimation error.

Theorem 11 (Weighted-KDE Estimation Error). *Let $Z(q)$ be the median-of-means estimate constructed using the L unbiased estimators. Then with probability $1-\delta$,*

$$|Z(q) - f_K(q)| \leq \epsilon \qquad \epsilon = 6\frac{\widetilde{f}_K(q)}{\sqrt{L}}\sqrt{\log 1/\delta}$$

where $f_K(q)$ and $\widetilde{f}_K(q)$ are the weighted KDE with kernels $\mathcal{K}(x,q)$ and $\sqrt{\mathcal{K}(x,q)}$, respectively.

The universal approximation property of our model follows by observing that the error ϵ in Theorem 11 goes to zero as $N \to \infty$ and $\epsilon \to 0$ in Theorem 9 as $L \to \infty$.

6.5 Distributed Efficiency Analysis

In this section, we investigate *Representer Sketch* for distributed setting and argue that it is inherently efficient because of the "parallel-ability" and "merge-ability."

Why are neural networks hard to distribute? There are two standard ways to distribute a model's training process: model parallel and data parallel. Model parallel splits a model among multiple devices while each device holds a complete copy of all training data. For neural networks, the previous step's results must be combined for the next step. As a result, devices must communicate activations during

the forward pass and gradients during the backward pass for each layer and every round of gradient descent. Data parallel splits the data among devices, with each device holding a complete copy of the model. Due to the lack of interpretation of neural network parameters and gradient descent training process, neural networks trained on different data source cannot be simply merged together. To obtain one single model, neural networks either require communication during training to sync parameters or the use of ensemble techniques, which requires full computation for every model during inference.

6.5.1 Parallel Models with the Representer Sketch

Consider a *Representer Sketch* with a sketch of L rows and R columns $S \in \mathbb{R}^{L \times R}$, and L LSH functions $h_l \rightarrow \{1, \ldots R\} \in \mathbb{Z}$ for $l = 1, \ldots L$. Following the model parallel setting, we split our model to P computing devices. Every device i holds a sub-model $S_i \in \mathbb{R}^{\frac{L}{P} \times R}$ and $\frac{L}{P}$ hash functions h_i. For example, $S[0 : \frac{L}{P}]$ and $h_0 \ldots h_{\frac{L}{P}-1}$ on device 0; $S[\frac{L}{P} : \frac{2L}{P}]$ and $h_{\frac{L}{P}} \ldots h_{\frac{2L}{P}-1}$ on device 1,etc. Then, Equations **??** can be re-organized into

$$
\begin{aligned}
f(x; S, h) &= \frac{1}{P} \sum_{i=1}^{P} \left(\frac{\sum_{l=1}^{\frac{L}{P}} S_i[l, h_{i_l}(x)]}{L} \right) \\
&= \frac{1}{P} \sum_{i=1}^{P} f(x; S_i, h_i)
\end{aligned}
\tag{6.1}
$$

It is clear that only the final output from each sub-model is required to communicate between devices. Hashing computation and array look-ups happening at every row in the sketch is entirely parallel-able. Thus, the communication cost is independent of model structure, and solely depends on the number of computing devices.

6.5.2 Mergeable Models with the Representer Sketch

We seek to understand the properties of the average of independently-trained *Representer Sketch* models. We begin with an intuitive connection between the averaged sketch and bagging ensembles.

Theorem 12. *Given m datasets $\{D_1, ...D_m\}$, construct m representer sketch models $\{f(x; S_1, h), ...f(x; S_2, h)\}$ with the same hash functions h. Let $f(x; \bar{S}, h)$ be the model obtained by averaging the sketch parameters $\{S_1, ...S_m\}$. Then*

$$f(x; \bar{S}, h) = \frac{\sum_{i=1}^{m} f(x; S_i, h)}{m}$$

The averaged sketch output $f_{(\bar{S},h)}(x)$ is the ensemble average over a bagged ensemble with disjoint bags. One benefit over the traditional ensemble method is that only one final model is required for inference. Note that we are free to use the same hash functions h without affecting model quality, as h is not trainable parameter.

In the more difficult case where the training data on each device are not distributed i.i.d, we require further analysis. We consider query data being outside of training distribution.

Theorem 13. *Given a dataset D, construct a representer sketch model $f_{(S,h)}(x)$ by running Algorithm 5 for e epochs with learning rate η. Suppose the gradient norms are bounded by G. Then*

$$|\mathbb{E}_h[f_{(S,h)}(x)]| \leq e\eta G \, \mathrm{KDE}(x)$$

where $\mathrm{KDE}(x)$ *is the kernel density of x over D using the kernel of the LSH function h.*

Our intuition is that the model outputs values near zero for queries that are out of the training distribution, because these queries will land in array cells that were not

modified during the training process. Theorem 13 proves this behavior by showing that the expected output value is bounded by the density of the query over the dataset. This result provides theoretical justification for averaging *Representer Sketch* models even with non-i.i.d. data. If x is out-of-distribution for device D_i, it will have a low KDE(x) over D_i and thus a low $f(x; S_i, h)$. As a result, S_i contributes little to the ensemble average when queried with low-confidence inputs. By naturally encoding this out-of-distribution information in the sketch, our models are robust to non-i.i.d. distributions.

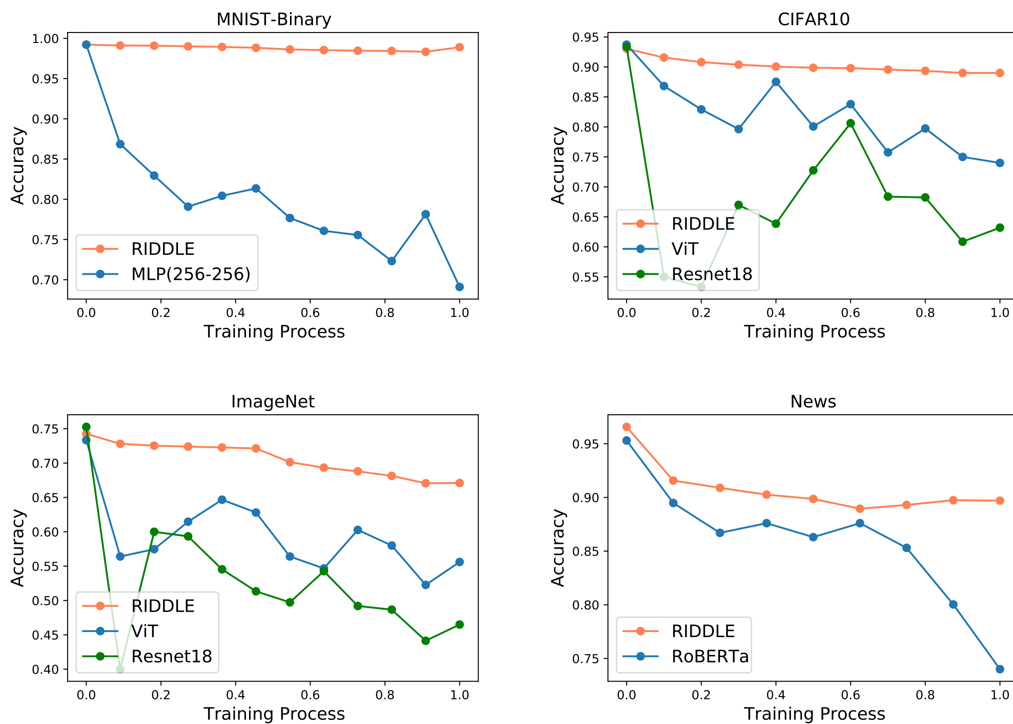


Figure 6.3 : Accuracy on the original test set while training on the update distribution. We observe that the RIDDLE performance remains nearly constant for all datasets, while baseline methods degrade rapidly with inputs from the new distribution.

6.6 Experiments

In this section, we first evaluate RIDDLE on practical dynamic distribution learning task in Section 6.6.1. Further, since we prove RIDDLE as a universal function approximator, we provide an extensive study on the expressiveness and efficiency of RIDDLE in Section 6.6.3.

6.6.1 RIDDLE for Learning under Dynamic Definition

Here, we investigate whether RIDDLE can adapt to new distributions while retaining good performance on the original distributions. We consider four datasets that reflect dynamic distributions. Every dataset is organized into two parts: the original dataset D^o and the update dataset D^u. D^o and D^u have significant semantic differences and do not share any overlapping training data.

MNIST Binary: We randomly partitioned digits into two groups (e.g. $4, 1, 7, 5, 3$ and $9, 0, 8, 6, 2$) and train the model to predict the group assignment of an image. For the original distribution, we randomly choose 3 digits from the first group and 3 digits from the second group such as D^o only consists images of 4, 1, 7, 9, 0, 8. The update dataset consists of images of the remaining digits.

CIFAR10: The task is to predict whether the object in the image is capable of carrying things (e.g. car and horse can carry things, while a frog cannot). The original dataset D^o contains images of 6 out of 10 objects that can and cannot carry things, and the update dataset D^u contains the remaining 4 objects.

ImageNet: The task is to predict whether a person is allergic to the object in the image. We select the image classes overlapping with national allergen directctory [146], and assign images to the allergen and non-allergen classes based on the common allergen group in national allergen directory [146]. The original dataset D^o contains

		Trained on original dataset		Updated on update dataset	
Dataset	Method	Original	Update	Original	Update
MNIST Binary	Update-MLP	99.2	59.1	69.1	98.9
	EWC-MLP	99.2	59.8	90.9	96.7
	RIDDLE	99.2	60.2	**98.3**	98.0
CIFAR10	Update-Resnet18	93.3	69.4	63.22	88.9
	Update-ViT	93.3	72.6	74.4	92.7
	EWC-Resnet	92.0	71.8	75.9	83.9
	EWC-ViT	91.3	75.5	84.5	90.5
	RIDDLE	93.0	75.7	**89.0**	90.9
ImageNet	Update-Resnet18	76.0	45.2	46.53	62.5
	Update-ViT	74.3	45.3	55.6	71.5
	EWC-Resnet	74.0	46.6	52.8	68.2
	EWC-Vit	71.8	41.2	54.5	71.8
	RIDDLE	74.3	45.8	**67.1**	70.5
News	Update-RoBERTa	96.1	54.8	74.0	85.7
	EWC-RoBERTa	96.5	45.5	86.1	87.5
	RIDDLE	96.6	54.6	**89.7**	86.4

Table 6.1 : This table summarizes the accuracy after training on original dataset, and after training on update dataset. The RIDDLE obtains comparable accuracy on original test set and update test set after trained on corresponding train set. The RIDDLE's accuracy is significantly higher than all baselines on original test set after updating.

a subset of allergens and non-allergens, while the update dataset D^u contains the remaining objects. This task was built to mimic the the seasonal addition and sensitization of new allergens.

News: The original task for the News dataset [147] is to predict the news topics

given a caption and short description. We re-task the dataset to predict whether a person would be interested in a new article by assigning some topics as interesting and the rest as not-interesing. The original dataset D^o contains news from a subset of interesting and non-interesting topics, while the update dataset D^u contains the rest. We organize this dataset to reflect changes in a user's interests.

We compare RIDDLE with two updating strategies on different architectures.

Update: We train a neural network to full convergence on the original dataset D^o. Then, for updating, we train models only on the update dataset D^u, where the initialization is the parameters obtained on D^o. This approach does not require data storage nor training computation over the original dataset.

Elastic Weight Consolidation(EWC): [148] was proposed to mitigate catastrophic forgetting by selectively slowing down learning on the weights important for previous tasks using Fisher information.

Experiment Details

For tasks where it is reasonable (CIFAR10, ImageNet and News), we exploit recent advances in large scale pretrained models and replace the final layer with RIDDLE. Pretrained models are usually trained on a diverse corpus of unlabeled data and are likely to provide good featurization across distributions. [149] also observe that pretrained models are more resistant to forgetting.

6.6.2 Main Results

Table 6.1 summarizes the test accuracy of all models after training on the original dataset and after the update. After training on the original dataset, we observe that all models achieve roughly the same accuracy on original test set. Accuracy on

update test dataset is much worse, which is expected since the update distribution is significantly different from the original distribution.

We are interested in the accuracy on original dataset after updating the model on update dataset. We observe a 20% to 30 % accuracy drop on the original accuracy with naive updating. Models such as Resnet18 and ViT display catastrophic forgetting behavior, eventually performing no better than random. EWC is effective in prevent forgetting to some extend, reducing the degradation to around 10% to 20%. The RIDDLE significantly outperforms both baselines combined with all architectures across datasets up to in terms of original test set accuracy. Indeed, the RIDDLE keeps the accuracy degradation below 7 %. For example, the accuracy drop is only 0.9% on Mnist-Binary. On the updated test set, the RIDDLE achieves competitive accuracy (Table 6.1), indicating that it is learning the properties of both distributions.

Figure 6.3 shows the accuracy on the original test set during the update process. Unsurprisingly, the baseline performance decreases as we see more training samples from update test set. Models such as Resnet18 and ViT display catastrophic forgetting behavior, eventually performing no better than random. The RIDDLE model performance, on the other hand, remains nearly constant and lies above all baselines throughout the update process, especially on image datasets.

Note that ViT outperforms Resnet18 on both CIFAR10 and ImageNet, which confirms our intuition that pretrained models are inherently more robust to distribution changes. By switching the final classification layer to the RIDDLE, we increase the accuracy by a large margin. The experiment results strongly support our hypothesis that the RIDDLE retains the ability to adapt to new distributions while being less vulnerable than established methods to knowledge loss.

	MLP	Global Learning Rate	$\eta(c) \propto \frac{\bar{c}}{c \log c+1}$
Test accuracy on original	69.1	94.9	98.3

Table 6.2 : This table displays the effect of the partition-dependent learning rate. "Global Learning Rate" means that the learning rate is not related to parameter counts.

Ablation on Update Rule

There are two main drivers of performance when training RIDDLE over dynamically evolving definitions: the hash function and the parameter-dependently learning rate.

Without a learning rate that depends on parameter importance, the test accuracy on the original MNIST test set after updating is 94.9%, which is 26 % higher than Update-MLP, 4 % higher than EWC-MLP. Using the dynamic learning rate, RIDDLE achieves 98%. There could be smarter or simpler update functions based on the parameter count, which we leave for future work.

Dataset	LightGBM	Random Forest	XGBoost	NN1	NN2	RIDDLE
Susy	0.7903	0.7880	0.7926	0.8019	0.8024	0.7924
HAR	0.8532	0.8387	0.8532	0.9027	0.9123	0.9001
Covtype	0.9565	0.9932	0.9874	0.9836	0.9936	0.9853
Connect4	0.8171	0.8335	0.8171	0.8558	0.8657	0.8586
Fashion MNIST	0.8848	0.8777	0.8848	0.8852	0.8889	0.8852
Boston Housing	1.77	2.05	1.62	2.57	2.16	1.74

Table 6.3 : This table summarizes the accuracy comparison. NN1 and NN2 denote two different deep learning models. RIDDLE achieves higher accuracy than at least half of the baselines on all datasets.

Dataset	Method	Accuracy	Memory(MB)	Flops (M)	Inference Time (μs)
MNIST	NN	0.9815	3.59	0.90	164.177
	RIDDLE	0.9784	1.36(1.6X)	0.15(6.1X)	42.003 (3.9X)
Fashion MNIST	NN	0.8852	2.59	0.65	131.385
	RIDDLE	0.8852	1.92(1.6X)	0.13(5X)	39.228 (3.3X)
HAR	NN	0.9027	2.11	0.53	166.584
	RIDDLE	0.9001	1.18(1.8X)	0.18 (2.9X)	46.567 (3.6X)
Boston Housing	NN	2.16	1.96	0.51	61.338
	RIDDLE	1.74	0.53(3.7X)	0.03 (17X)	21.862 (2.8X)

Table 6.4 : This table summarizes the efficiency comparison between the RIDDLE and a neural network (NN) with similar accuracy. The RIDDLE reduces memory by up to 3.7x, FLOPs by up to 17x and inference time by up to 3.9x.

6.6.3 Study on Expressiveness and Efficiency

In this section, we investigate whether RIDDLE is competitive among popular machine learning models in terms of expressiveness and efficiency. Our goal is to show that RIDDLE provides a good model representation for a variety of tasks. We curated a diverse set of datasets from UCI and Kaggle. These datasets cover a wide range of tasks and domains and are widely used as benchmarks in other works [150, 151]. All experiments are conducted on a machine with 96 24-core/2-thread/2-socket processors (Intel Xeon(R) Gold 5220R 2.20GHz) and 8 Nvidia V100 32GB.

In Table 6.1, we observe that RIDDLE outperforms popular linear methods and is competitive compared to deep learning methods. NN1 and NN2 denote two different neural network architectures(Details in Appendix). Both the architecture and evaluation metric is included in Table 6.3. We also investigate the efficiency

and memory requirement. This is important, as it shows that RIDDLE does not outperform other methods due to an increased parameter size. Table 6.4 shows that RIDDLE attains the same accuracy as baseline methods with a smaller memory footprint, fewer FLOPs and a faster inference speed. Specifically, we use up to 3.7X less memory, use 17X less FLOPs, and achieve 3.9x faster inference speed. The memory improvements likely arise from the meaningful parameter groupings, while the computational improvements are because it only uses addition, subtraction and array lookup operations. This experiment confirms that even though the RIDDLE allocates different parameters to different distributions, we do not require more parameters than existing methods to achieve robustness or accuracy.

Chapter 7

Conclusion

This thesis centers on addressing the computational challenges encountered in large-scale machine learning models. Specifically, out investigations delved into harnessing dynamic sparsity to achieve efficiency objectives.

First, our focus lies on ML models during the inference phase. We verified that within a trained ML model, dynamic sparsity inherently exists, where a subset of parameters and hidden states is necessitated to produce nearly identical outcomes for input data. MMoreover, we establish that predicting the dynamic sparsity pattern for individual inputs can be achieved inexpensively, drawing inspiration from the literature on nearest neighbor search. Subsequently, we demonstrate the feasibility of conducting model inference using predicted sparsity parameters/activations, yielding substantial computational savings without compromising accuracy. Our analysis extends to diverse model components, encompassing the transformer block, classification layer, and key-value cache in large language models (LLMs). These findings hold promising implications for the efficient deployment of models and the sustainable advancement of machine learning. Such methodology could be a step towards making large-scale ML models more accessible to the general community, which could unlock exciting new AI applications.

Subsequently, our exploration extends to the fine-tuning stages, where we introduce a novel architecture intentionally designed to accentuate dynamic sparsity. This design ensures that varying input distributions route to distinct sets of parameters. We

illustrate such a model featuring dynamic sparsity is inherently resilient to forgetting, as gradient updates naturally confine themselves to parameters relevant to the input data.

Bibliography

[1] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, (New York, NY, USA), p. 604–613, Association for Computing Machinery, 1998.

[2] A. Andoni and I. Razenshteyn, "Optimal data-dependent hashing for approximate near neighbors," in *Proceedings of the forty-seventh annual ACM symposium on Theory of computing (STOC)*, pp. 793–801, 2015.

[3] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, (New York, NY, USA), p. 253–262, Association for Computing Machinery, 2004.

[4] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, "Practical and optimal lsh for angular distance," in *Advances in Neural Information Processing Systems (NIPS)*, pp. 1225–1233, Curran Associates, 2015.

[5] B. Coleman and A. Shrivastava, *Sub-Linear RACE Sketches for Approximate Kernel Density Estimation on Streaming Data*, p. 1739–1749. New York, NY, USA: Association for Computing Machinery, 2020.

[6] A. Broder, "On the resemblance and containment of documents," in *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97, (USA),

p. 21, IEEE Computer Society, 1997.

[7] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, (New York, NY, USA), p. 380–388, Association for Computing Machinery, 2002.

[8] R. Spring and A. Shrivastava, "A new unbiased and efficient class of lsh-based samplers and estimators for partition function computation in log-linear models," *arXiv preprint arXiv:1703.05160*, 2017.

[9] M. Charikar and P. Siminelakis, "Hashing-based-estimators for kernel density in high dimensions," in *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 1032–1043, IEEE, 2017.

[10] C. Luo and A. Shrivastava, "Arrays of (locality-sensitive) count estimators (ace): Anomaly detection on the edge," in *Proceedings of the 2018 World Wide Web Conference*, WWW '18, (Republic and Canton of Geneva, CHE), p. 1439–1448, International World Wide Web Conferences Steering Committee, 2018.

[11] X. Li and P. Li, "Random projections with asymmetric quantization," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[12] X. Li and P. Li, "Quantization algorithms for random fourier features," in *International Conference on Machine Learning*, pp. 6369–6380, PMLR, 2021.

[13] X. Li and P. Li, "Quantization algorithms for random fourier features," in *International Conference on Machine Learning*, pp. 6369–6380, PMLR, 2021.

[14] A. Zandieh, N. Nouri, A. Velingker, M. Kapralov, and I. Razenshteyn, “Scaling up kernel ridge regression via locality sensitive hashing,” in *International Conference on Artificial Intelligence and Statistics*, pp. 4088–4097, PMLR, 2020.

[15] X. Li and P. Li, “Generalization error analysis of quantized compressive learning,” *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[16] A. Andoni, P. Indyk, H. L. Nguyen, and I. Razenshteyn, “Beyond locality-sensitive hashing,” in *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pp. 1018–1028, SIAM, 2014.

[17] A. Andoni, T. Laarhoven, I. Razenshteyn, and E. Waingarten, “Optimal hashing-based time-space trade-offs for approximate near neighbors,” in *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 47–66, SIAM, 2017.

[18] B. Chen, T. Medini, and A. Shrivastava, “SLIDE : In defense of smart algorithms over hardware acceleration for large-scale deep learning systems,” *CoRR*, vol. abs/1903.03129, 2019.

[19] B. Chen, Z. Liu, B. Peng, Z. Xu, J. L. Li, T. Dao, Z. Song, A. Shrivastava, and C. Re, “{MONGOOSE}: A learnable {lsh} framework for efficient neural network training,” in *International Conference on Learning Representations*, 2021.

[20] N. Kitaev, Kaiser, and A. Levskaya, “Reformer: The efficient transformer,” 2020.

[21] Z. Liu, Z. Xu, A. B. Ji, J. Li, B. Chen, and A. Shrivastava, “Climbing the WOL: training for cheaper inference,” *CoRR*, vol. abs/2007.01230, 2020.

[22] M. Zhang, X. Liu, W. Wang, J. Gao, and Y. He, "Navigating with graph representations for fast and scalable decoding of neural language models," *arXiv preprint arXiv:1806.04189*, 2018.

[23] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, *et al.*, "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2021.

[24] P. Liang, R. Bommasani, T. Lee, D. Tsipras, D. Soylu, M. Yasunaga, Y. Zhang, D. Narayanan, Y. Wu, A. Kumar, *et al.*, "Holistic evaluation of language models," *arXiv preprint arXiv:2211.09110*, 2022.

[25] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[26] S. Min, X. Lyu, A. Holtzman, M. Artetxe, M. Lewis, H. Hajishirzi, and L. Zettlemoyer, "Rethinking the role of demonstrations: What makes in-context learning work?," *arXiv preprint arXiv:2202.12837*, 2022.

[27] S. C. Chan, A. Santoro, A. K. Lampinen, J. X. Wang, A. K. Singh, P. H. Richemond, J. McClelland, and F. Hill, "Data distributional properties drive emergent in-context learning in transformers," in *Advances in Neural Information Processing Systems*, 2022.

[28] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, A. Levskaya, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *arXiv preprint arXiv:2211.05102*, 2022.

[29] Y. LeCun, J. Denker, and S. Solla, "Optimal brain damage," *Advances in neural information processing systems*, vol. 2, 1989.

[30] N. Lee, T. Ajanthan, and P. H. Torr, "Snip: Single-shot network pruning based on connection sensitivity," *arXiv preprint arXiv:1810.02340*, 2018.

[31] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, OpenReview.net, 2019.

[32] P. Michel, O. Levy, and G. Neubig, "Are sixteen heads really better than one?," *Advances in neural information processing systems*, vol. 32, 2019.

[33] H. Bansal, K. Gopalakrishnan, S. Dingliwal, S. Bodapati, K. Kirchhoff, and D. Roth, "Rethinking the role of scale for in-context learning: An interpretability-based case study at 66 billion scale," *arXiv preprint arXiv:2212.09095*, 2022.

[34] S. Hooker, "The hardware lottery," *Communications of the ACM*, vol. 64, no. 12, pp. 58–65, 2021.

[35] E. Frantar and D. Alistarh, "Massive language models can be accurately pruned in one-shot," *arXiv preprint arXiv:2301.00774*, 2023.

[36] S. M. Xie, A. Raghunathan, P. Liang, and T. Ma, "An explanation of in-context learning as implicit bayesian inference," in *International Conference on Learning Representations*, 2022.

[37] L. E. Baum and T. Petrie, "Statistical inference for probabilistic functions of finite state markov chains," *The annals of mathematical statistics*, vol. 37, no. 6,

pp. 1554–1563, 1966.

[38] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, 1967.

[39] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *STOC*, 1998.

[40] M. Zhang, W. Wang, X. Liu, J. Gao, and Y. He, "Navigating with graph representations for fast and scalable decoding of neural language models," in *Advances in Neural Information Processing Systems*, pp. 6308–6319, 2018.

[41] B. Chen, T. Medini, J. Farwell, C. Tai, A. Shrivastava, *et al.*, "Slide: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 291–306, 2020.

[42] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition(CVPR)*, pp. 770–778, 2016.

[43] J. E. Smith, "A study of branch prediction strategies," in *25 years of the international symposia on Computer architecture (selected papers)*, pp. 202–215, 1998.

[44] J. Alman and Z. Song, "Fast attention requires bounded entries," *arXiv preprint arXiv:2302.13214*, 2023.

[45] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," *arXiv preprint arXiv:1611.06440*, 2016.

[46] T. Mihaylov, P. Clark, T. Khot, and A. Sabharwal, "Can a suit of armor conduct electricity? a new dataset for open book question answering," in *EMNLP*, 2018.

[47] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," 2016.

[48] M. Kurtz, J. Kopinsky, R. Gelashvili, A. Matveev, J. Carr, M. Goin, W. Leiserson, S. Moore, N. Shavit, and D. Alistarh, "Inducing and exploiting activation sparsity for fast inference on deep neural networks," in *Proceedings of the 37th International Conference on Machine Learning* (H. D. III and A. Singh, eds.), vol. 119 of *Proceedings of Machine Learning Research*, pp. 5533–5543, PMLR, 13–18 Jul 2020.

[49] Z. Li, C. You, S. Bhojanapalli, D. Li, A. S. Rawat, S. J. Reddi, K. Ye, F. Chern, F. Yu, R. Guo, and S. Kumar, "Large models are parsimonious learners: Activation sparsity in trained transformers," 2022.

[50] K. G. Derpanis, "Mean shift clustering," *Lecture Notes*, vol. 32, pp. 1–4, 2005.

[51] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *arXiv e-prints*, 2019.

[52] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[53] J. Johnson, M. Douze, and H. Jegou, "Billion-scale similarity search with gpus," *IEEE Transactions on Big Data*, 2019.

[54] NVIDIA, "Gpu performance background user's guide," 2022.

[55] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data movement is all you need: A case study on optimizing transformers," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 711–732, 2021.

[56] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," in *Advances in Neural Information Processing Systems*, 2022.

[57] M. Harris, "How to access global memory efficiently in CUDA C/C++ kernels," *NVIDIA, Jan*, 2013.

[58] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2012.

[59] P. Tillet, H.-T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019.

[60] M.-C. de Marneffe, M. Simons, and J. Tonhauser, "The commitmentbank: Investigating projection in naturally occurring discourse," 2019.

[61] A. Gordon, Z. Kozareva, and M. Roemmele, "SemEval-2012 task 7: Choice of plausible alternatives: An evaluation of commonsense causal reasoning,"

in **SEM 2012: The First Joint Conference on Lexical and Computational Semantics – Volume 1: Proceedings of the main conference and the shared task, and Volume 2: Proceedings of the Sixth International Workshop on Semantic Evaluation (SemEval 2012)*, (Montréal, Canada), pp. 394–398, Association for Computational Linguistics, 7-8 June 2012.

[62] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[63] Y. Bisk, R. Zellers, R. L. Bras, J. Gao, and Y. Choi, "Piqa: Reasoning about physical commonsense in natural language," in *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

[64] D. Giampiccolo, B. Magnini, I. Dagan, and B. Dolan, "The third PASCAL recognizing textual entailment challenge," in *Proceedings of the ACL-PASCAL Workshop on Textual Entailment and Paraphrasing*, (Prague), pp. 1–9, Association for Computational Linguistics, June 2007.

[65] "Winogrande: An adversarial winograd schema challenge at scale," 2019.

[66] L. Gao, J. Tow, S. Biderman, S. Black, A. DiPofi, C. Foster, L. Golding, J. Hsu, K. McDonell, N. Muennighoff, J. Phang, L. Reynolds, E. Tang, A. Thite, B. Wang, K. Wang, and A. Zou, "A framework for few-shot language model evaluation," Sept. 2021.

[67] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, D. Y. Fu, Z. Xie, B. Chen, C. Barrett, J. E. Gonzalez, P. Liang, C. Ré, I. Stoica, and C. Zhang, "High-throughput generative inference of large language models with a single gpu,"

2023.

[68] Z. Yao, R. Y. Aminabadi, M. Zhang, X. Wu, C. Li, and Y. He, "Zeroquant: Efficient and affordable post-training quantization for large-scale transformers," *arXiv preprint arXiv:2206.01861*, 2022.

[69] G. Park, B. Park, S. J. Kwon, B. Kim, Y. Lee, and D. Lee, "nuqmm: Quantized matmul for efficient inference of large-scale generative language models," *arXiv preprint arXiv:2206.09557*, 2022.

[70] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "Llm. int8 (): 8-bit matrix multiplication for transformers at scale," *arXiv preprint arXiv:2208.07339*, 2022.

[71] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," *arXiv preprint arXiv:2210.17323*, 2022.

[72] G. Xiao, J. Lin, M. Seznec, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," *arXiv preprint arXiv:2211.10438*, 2022.

[73] OpenAI, "Gpt-4 technical report," 2023.

[74] N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, OpenReview.net, 2020.

[75] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," *arXiv preprint arXiv:2006.04768*, 2020.

[76] B. Chen, T. Dao, E. Winsor, Z. Song, A. Rudra, and C. Ré, "Scatterbrain: Unifying sparse and low-rank attention," *Advances in Neural Information Processing Systems*, vol. 34, pp. 17413–17426, 2021.

[77] K. M. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlós, P. Hawkins, J. Q. Davis, A. Mohiuddin, L. Kaiser, D. B. Belanger, L. J. Colwell, and A. Weller, "Rethinking attention with performers," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, OpenReview.net, 2021.

[78] N. Shazeer, "Fast transformer decoding: One write-head is all you need," 2019.

[79] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[80] R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi, "Hellaswag: Can a machine really finish your sentence?," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4791–4800, 2019.

[81] H.-J. Xue, X. Dai, J. Zhang, S. Huang, and J. Chen, "Deep matrix factorization models for recommender systems.," in *IJCAI*, pp. 3203–3209, 2017.

[82] M. Fan, J. Guo, S. Zhu, S. Miao, M. Sun, and P. Li, "Mobius: Towards the next generation of query-ad matching in baidu's sponsored search," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2509–2517, 2019.

[83] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic

language model," *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[84] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh annual conference of the international speech communication association*, 2010.

[85] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[86] C. You, N. Chen, and Y. Zou, "MRD-Net: Multi-Modal Residual Knowledge Distillation for Spoken Question Answering," in *IJCAI*, 2021.

[87] C. You, N. Chen, and Y. Zou, "Self-supervised contrastive cross-modality representation learning for spoken question answering," in *Findings of the Association for Computational Linguistics: EMNLP*, 2021.

[88] K. Bhatia, K. Dahiya, H. Jain, A. Mittal, Y. Prabhu, and M. Varma, "The extreme classification repository: Multi-label datasets and code," 2016.

[89] A. Shrivastava and P. Li, "Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips)," in *NIPS*, 2014.

[90] A. Shrivastava and P. Li, "Improved asymmetric locality sensitive hashing (alsh) for maximum inner product search (mips)," in *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence*, UAI'15, (Arlington, Virginia, USA), p. 812–821, AUAI Press, 2015.

[91] R. Guo *et al.*, "Quantization based fast inner product search," in *Artificial Intelligence and Statistics*, pp. 482–490, 2016.

[92] R. Wetzker, C. Zimmermann, and C. Bauckhage, "Analyzing social bookmarking systems: A delicious cookbook," *Proceedings of the ECAI 2008 Mining Social Data Workshop*, pp. 26–30, 01 2008.

[93] J. Xu, X. He, and H. Li, "Deep learning for matching in search and recommendation," in *WWW Tutorials*, 2018.

[94] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *STOC*, 2002.

[95] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *International conference on machine learning*, pp. 1597–1607, PMLR, 2020.

[96] G. Chechik, V. Sharma, U. Shalit, and S. Bengio, "Large scale online learning of image similarity through ranking.," *Journal of Machine Learning Research (JMLR)*, vol. 11, no. 3, 2010.

[97] Z. Cao, M. Long, J. Wang, and P. S. Yu, "Hashnet: Deep learning to hash by continuation," *CoRR*, vol. abs/1702.00758, 2017.

[98] A. Zubiaga, "Enhancing navigation on wikipedia with social tags," *CoRR*, vol. abs/1202.5469, 2012.

[99] M. Mahoney, "Text8." `http://mattmahoney.net/dc/textdata.html`, 2011.

[100] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," *CoRR*, vol. abs/1609.07843, 2016.

[101] T. Mikolov and G. Zweig, "Context dependent recurrent neural network language

model," in *2012 IEEE Spoken Language Technology Workshop (SLT)*, pp. 234–239, IEEE, 2012.

[102] B. Chen, T. Medini, and A. Shrivastava, "Slide: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems," in *Proceedings of the 3rd Conference on Systems and Machine Learning (MLSys)*, 2020.

[103] Y. Bachrach, Y. Finkelstein, R. Gilad-Bachrach, L. Katzir, N. Koenigstein, N. Nice, and U. Paquet, "Speeding up the xbox recommender system using a euclidean transformation for inner-product spaces," in *Proceedings of the 8th ACM Conference on Recommender systems*, pp. 257–264, 2014.

[104] L. Boytsov and B. Naidan, "Engineering efficient and effective non-metric space library," in *Similarity Search and Applications - 6th International Conference, SISAP 2013, A Coruna, Spain, October 2-4, 2013, Proceedings* (N. R. Brisaboa, O. Pedreira, and P. Zezula, eds.), vol. 8199 of *Lecture Notes in Computer Science*, pp. 280–293, Springer, 2013.

[105] S. Morozov and A. Babenko, "Non-metric similarity graphs for maximum inner product search," in *Advances in Neural Information Processing Systems*, pp. 4726–4735, 2018.

[106] J. Johnson, M. Douze, and H. Jegou, "Billion-scale similarity search with gpus," *arXiv preprint arXiv:1702.08734*, 2017.

[107] Y. Dong, P. Indyk, I. Razenshteyn, and T. Wagner, "Learning space partitions for nearest neighbor search," *arXiv preprint arXiv:1901.08544*, 2019.

[108] P. H. Chen, S. Si, S. Kumar, Y. Li, and C.-J. Hsieh, "Learning to screen for

fast softmax inference on large vocabulary neural networks," *arXiv preprint arXiv:1810.12406*, 2018.

[109] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pp. 263–272, Ieee, 2008.

[110] Y. Wang, A. Shrivastava, J. Wang, and J. Ryu, "Flash: Randomized algorithms accelerated over cpu-gpu for ultra-high dimensional similarity search," *arXiv preprint arXiv:1709.01190*, 2017.

[111] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio, "An empirical investigation of catastrophic forgetting in gradient-based neural networks," 2013.

[112] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

[113] G. I. Parisi, J. Tani, C. Weber, and S. Wermter, "Lifelong learning of spatiotemporal representations with dual-memory recurrent self-organization," *Frontiers in neurorobotics*, p. 78, 2018.

[114] N. Kamra, U. Gupta, and Y. Liu, "Deep generative dual memory network for continual learning," *arXiv preprint arXiv:1710.10368*, 2017.

[115] Z. Li and D. Hoiem, "Learning without forgetting," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 12, pp. 2935–2947, 2017.

[116] H. Jung, J. Ju, M. Jung, and J. Kim, "Less-forgetting learning in deep neural networks," *arXiv preprint arXiv:1607.00122*, 2016.

[117] J. Serra, D. Suris, M. Miron, and A. Karatzoglou, "Overcoming catastrophic forgetting with hard attention to the task," in *International Conference on Machine Learning*, pp. 4548–4557, PMLR, 2018.

[118] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell, "Progressive neural networks," *arXiv preprint arXiv:1606.04671*, 2016.

[119] J. Yoon, E. Yang, J. Lee, and S. J. Hwang, "Lifelong learning with dynamically expandable networks," *arXiv preprint arXiv:1708.01547*, 2017.

[120] A. Antoniou, M. Patacchiola, M. Ochal, and A. Storkey, "Defining benchmarks for continual few-shot learning," 2020.

[121] O. Vinyals, C. Blundell, T. Lillicrap, K. Kavukcuoglu, and D. Wierstra, "Matching networks for one shot learning," in *NIPS*, pp. 3630–3638, 2016.

[122] J. Snell, K. Swersky, and R. Zemel, "Prototypical networks for few-shot learning," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, (Red Hook, NY, USA), p. 4080–4090, Curran Associates Inc., 2017.

[123] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, "A comprehensive survey on transfer learning," *Proceedings of the IEEE*, vol. 109, no. 1, pp. 43–76, 2020.

[124] Y. Sun, X. Wang, Z. Liu, J. Miller, A. A. Efros, and M. Hardt, "Test-time training with self-supervision for generalization under distribution shifts," 2020.

[125] D. Wang, E. Shelhamer, S. Liu, B. Olshausen, and T. Darrell, "Tent: Fully test-time adaptation by entropy minimization," 2021.

[126] S. Fort, J. Ren, and B. Lakshminarayanan, "Exploring the limits of out-of-distribution detection," *CoRR*, vol. abs/2106.03004, 2021.

[127] S. Liang, Y. Li, and R. Srikant, "Principled detection of out-of-distribution examples in neural networks," *CoRR*, vol. abs/1706.02690, 2017.

[128] Y. Ovadia, E. Fertig, J. Ren, Z. Nado, D. Sculley, S. Nowozin, J. Dillon, B. Lakshminarayanan, and J. Snoek, "Can you trust your model's uncertainty? evaluating predictive uncertainty under dataset shift," *Advances in neural information processing systems*, vol. 32, 2019.

[129] B. Lakshminarayanan, A. Pritzel, and C. Blundell, "Simple and scalable predictive uncertainty estimation using deep ensembles," *Advances in neural information processing systems*, vol. 30, 2017.

[130] J. Van Amersfoort, L. Smith, Y. W. Teh, and Y. Gal, "Uncertainty estimation using a single deep deterministic neural network," in *International conference on machine learning*, pp. 9690–9700, PMLR, 2020.

[131] A. Resler and Y. Mansour, "Adversarial online learning with noise," *CoRR*, vol. abs/1810.09346, 2018.

[132] S. Pokutta and H. Xu, "Adversaries in online learning revisited: with applications in robust optimization and adversarial training," *CoRR*, vol. abs/2101.11443,

2021.

[133] A. S. Suggala and P. Netrapalli, "Online non-convex learning: Following the perturbed leader is optimal," *CoRR*, vol. abs/1903.08110, 2019.

[134] D. Achlioptas, "Database-friendly random projections: Johnson-lindenstrauss with binary coins," *Journal of Computer and System Sciences*, vol. 66, no. 4, pp. 671–687, 2003. Special Issue on PODS 2001.

[135] W. Fedus, J. Dean, and B. Zoph, "A review of sparse expert models in deep learning," 2022.

[136] T. Hofmann, B. Schölkopf, and A. J. Smola, "Kernel methods in machine learning," *The Annals of Statistics*, vol. 36, jun 2008.

[137] I. J. Schoenberg, "Contributions to the problem of approximation of equidistant data by analytic functions," in *IJ Schoenberg Selected Papers*, pp. 3–57, Springer, 1988.

[138] J. Schmidhuber, "Discovering problem solutions with low kolmogorov complexity and high generalization capability," tech. rep., MACHINE LEARNING: PROCEEDINGS OF THE TWELFTH INTERNATIONAL CONFERENCE, 1994.

[139] R. Balestriero and richard baraniuk, "A spline theory of deep learning," in *Proceedings of the 35th International Conference on Machine Learning* (J. Dy and A. Krause, eds.), vol. 80 of *Proceedings of Machine Learning Research*, pp. 374–383, PMLR, 10–15 Jul 2018.

[140] R. Balestriero, R. Cosentino, H. Glotin, and R. Baraniuk, "Spline filters for end-to-end deep learning," in *Proceedings of the 35th International Conference on Machine Learning* (J. Dy and A. Krause, eds.), vol. 80 of *Proceedings of Machine Learning Research*, pp. 364–373, PMLR, 10–15 Jul 2018.

[141] R. Balestriero and R. G. Baraniuk, "Mad max: Affine spline insights into deep learning," *Proceedings of the IEEE*, vol. 109, no. 5, pp. 704–727, 2021.

[142] Z. Wang, R. Balestriero, and R. Baraniuk, "A MAX-AFFINE SPLINE PERSPECTIVE OF RECURRENT NEURAL NETWORKS," in *International Conference on Learning Representations*, 2019.

[143] C. A. Micchelli, Y. Xu, and H. Zhang, "Universal kernels.," *Journal of Machine Learning Research*, vol. 7, no. 12, 2006.

[144] B. Coleman and A. Shrivastava, "Sub-linear race sketches for approximate kernel density estimation on streaming data," in *Proceedings of The Web Conference 2020*, pp. 1739–1749, 2020.

[145] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," *Journal of Computer and system sciences*, vol. 58, no. 1, pp. 137–147, 1999.

[146] "Asthma and allergies foundation of america, https://www.aafa.org/types-of-allergies/."

[147] R. Misra, "News category dataset," 2022.

[148] J. Kirkpatrick, R. Pascanu, N. C. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis,

C. Clopath, D. Kumaran, and R. Hadsell, "Overcoming catastrophic forgetting in neural networks," *CoRR*, vol. abs/1612.00796, 2016.

[149] V. V. Ramasesh, A. Lewkowycz, and E. Dyer, "Effect of scale on catastrophic forgetting in neural networks," in *International Conference on Learning Representations*, 2022.

[150] Z. Zhou and J. Feng, "Deep forest: Towards an alternative to deep neural networks," *CoRR*, vol. abs/1702.08835, 2017.

[151] N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, and A. Smola, "Autogluon-tabular: Robust and accurate automl for structured data," 2020.