TorchSLIDE: Drop-in PyTorch Extensions for SLIDE Deep Learning

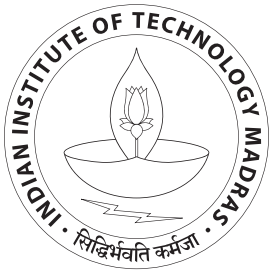
A Project Report

submitted by

E SANTHOSH KUMAR

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

June 2021

THESIS CERTIFICATE

This is to certify that the thesis entitled **TorchSLIDE: Drop-in PyTorch Extensions for SLIDE Deep Learning**, submitted by **E Santhosh Kumar**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Pratyush Kumar
Research Guide
Assistant Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 19 June 2021

ACKNOWLEDGEMENTS

Foremost, I would like to thank my guide Prof. Pratyush Kumar for his continued guidance, motivation and support throughout the thesis. I would also like to thank Vinod Ganesan for his insightful comments. Finally, I would like to thank my friends and family - my father M Elangovan, my mother Pirainuther Selvi and my sister Pavithra, for their love and support. They all kept me going and this thesis would not have been possible without them.

ABSTRACT

KEYWORDS: Extreme Classification, Locality Sensitive Hashing, PyTorch, Sparse Matrix Multiplication, SLIDE

Deep Learning (DL) models have proved to be the state-of-the-art solutions for several machine learning tasks in recent years. With continued increase in size of available training data, researchers have managed to train large neural networks with hundreds of millions of parameters. This is especially true for extreme classification tasks, where the objective is to predict a small subset of true labels among a vast set of available labels (as many as 100 million). The volume and nature of computation required for the training and inference of these huge models has led to the ascent of specialized hardware accelerators (GPUs, TPUs) designed to efficiently perform these operations.

In view of the risks associated with specialized hardware being less effective for future models, Chen *et al.* [2020] had proposed an algorithmic approach to achieve speedup in DL models. Their proposed SLIDE architecture uses Locality Sensitive Hashes (LSH) to take advantage of the high sparsity present in large matrix multiplication operations. Their provided C++ code is reported to show training time on a 44 core CPU that is more than 3.5 times than when using Tensorflow (TF) on a Tesla V100 GPU, and over 10x faster than when using TF on the same CPU.

Given the success of their prototype, we propose TorchSLIDE, a PyTorch im-

plementation of SLIDE. The proposed module is implemented to take advantage of PyTorch's built-in functionality, and works as a drop-in replacement for a linear layer. In the first part of this work, we describe the implementation choices choices behind TorchSLIDE and how they compare against SLIDE. We then evaluate TorchSLIDE on various extreme classification datasets to show that it matches the convergence properties of SLIDE, and achieves a speedup of up to 2.6X over it. Finally, we perform an ablation Study of Latency of TorchSLIDE's components

While we were working on TorchSLIDE, Daghaghi et al., Daghaghi *et al.* [2021] have released an improvement to SLIDE using memory optimizations and AVX-512 vectorization for modern CPUs. Our experiments compare against this implementation too (referred to as New SLIDE).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

LSH	Locality Sensitive Hash
WTA Hash	Winner-Takes-All Hash
DWTA Hash	Densified Winner-Takes-All Hash
DIDO	Dense Input Dense Output
DISO	Dense Input Sparse Output
SIDO	Sparse Input Dense Output
SISO	Sparse Input Sparse Output
TF	Tensorflow

CHAPTER 1

INTRODUCTION

Matrix multiplication operations lie at the heart of most DL models. As the need for fast multiplication of larger and larger matrices has increased over time, so has the investment in specialized hardware capable of performing this efficiently. These hardware devices (GPUs, TPUs, etc) try to push the limits of parallel computation, producing exact output values for all elements in the matrix product. However, in many use cases, we observe that the output dimension of the matrix product is quite large, but only a small fraction of the outputs are expected to have significant positive values. For instance, in the extreme classification task of product recommendation (e.g., Amazon-670K dataset), of the 670K available product labels, only a mere 6 labels correspond to true labels on average. Yet another example could be attention based networks, where among the several indices in a sequence, each index tends to attend most to a few select indices. So, if there is a way to efficiently identify and compute only these significant values in the matrix product, one might be able to replicate/exceed the performance of specialized hardware, even on CPUs having a much smaller degree of parallelization.

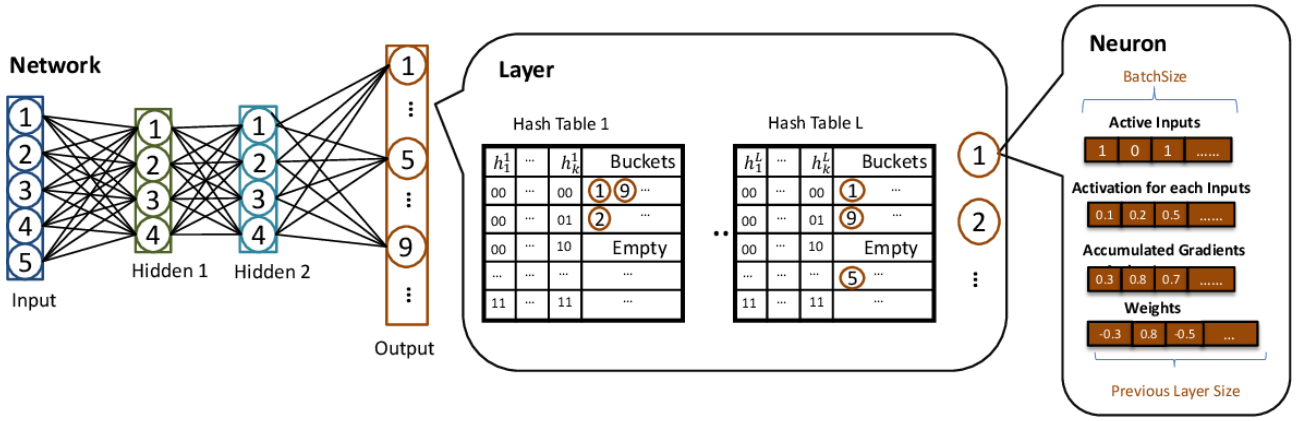


Figure 1.1: SLIDE Architecture (Chen *et al.* [2020])

1.1 Background: Sub-Linear Deep Learning Engine (SLIDE)

SLIDE (Sub-LInear Deep learning Engine) proposed by Chen *et al.* [2020] follows this direction by taking inspiration from 3 key ideas - LSH, sampled softmax and HogWild updates. LSH is a family of hash functions with the property that similar inputs in the domain have a higher probability of colliding in the range space than dissimilar ones. They help identify the nodes that are significantly activated for a given input. Unlike sampled softmax (an efficient approximation to full softmax provided by frameworks like TF) where the negative labels are chosen at random, negative labels in SLIDE's softmax are also adaptively chosen by the LSH. This allows for approximation using a much higher degree of sparsity. Finally, given the high degree of sparsity, the probability of collisions in gradient updates to weights among samples in a batch is very low. This allows for HOGWILD (Recht *et al.* [2011]) style parallel gradient updates that ensures convergence despite the occurrence of rare conflicting updates. A brief description of the SLIDE architecture and its key results follow.

- Every network layer has a list of N indexed neurons and a set of L hash

tables/meta-hash functions. Each hash table is in turn made of K random hash functions drawn independently from a chosen LSH family. Each hash table also holds a set of B fixed size buckets that store neuron indexes. The meta-hash function takes a d (hidden size) dimensional vector as input, and returns a bucket index in 0,1,...,B-1. This corresponds to the bucket in that hash table that the input belongs to. An LSH family of functions has the property that similar inputs in the domain have a higher probability of collision of their corresponding outputs, where the collision probability usually increases monotonically with the measure of similarity. Thus vectors present in the same bucket are expected to have a high similarity. Increasing K makes the nearest neighbour search stricter, making buckets sparse and reducing the number of false positives. Since the queries consider a union of elements in each chosen bucket, increasing L decreases the number of false negatives.

- **Pre-processing Phase**: Each of the N neurons in the layer are hashed into the corresponding buckets of each of the L hash tables. This operations is performed after every few training iterations (update frequency is a hyperparam and can also be exponentially decayed over time), using the current weight vectors of each neuron. Since the buckets are of fixed size, overflows are handled using replacement policies like FIFO or reservoir sampling. Also, the chosen random hash functions may also be reinitialized before bucketing. The computation involved in this step is linear in the number of neurons, and can be parallelized easily.

- **Sparse Feed-Forward Pass with Hash Table Sampling**: During the feed-forward step for an input instance, the incoming vector to the layer is used to query the hash tables. The buckets chosen by this vector contain samples of indexes of neurons that are expected to produce high activations for this input. We consider the union of neurons obtained from all L tables. The bucketing mechanism only requires that we probe one fixed-size bucket per table, instead of having to probe all neurons in the layer. Further, we only compute activations for these potentially significant neurons, and assume zero activation for the rest. Thus the forward pass computation, that happens during every train step, is sub-linear in the number of neurons. Further, it is parallelizable along the batch dimension with no adverse consequence. This process is followed for the final output layer as well, with the only modification (during training) being that true labels of the input instance are always sampled by default for better convergence.

- **Sparse Backpropagation**: Once the forward-pass is completed and loss is computed, gradients are backpropagated layer by layer as usual. Partial gradients are propagated only through the active neurons using a message passing implementation. Additional storage is used to ensure that partial gradients w.r.t neuron activations are maintained independently for each input in the batch. Thus even though backpropagation for the inputs happen

in parallel, the effects of weight update collisions are restricted to each layer, and are not carried backward. Partial gradients w.r.t weight parameters are accumulated (possibly with collisions) only for active weights that connect 2 active neurons. If the fraction of active neurons in each layer is s, the fraction of weight gradient accumulation operations required is s^2. Smaller this fraction, rarer the collisions and hence better for HogWild updates.

- Experiments in the paper were performed on simple fully connected networks with a single hidden layer, trained on extreme classification datasets (Delicious-200K, Amazon-670K). The majority of computations in these tasks is in the last layer. Results report that SLIDE's training time on a 44 core CPU could be up to 3.5 times lower than TF baselines on a Tesla V100 GPU, and up to 10 times lower than TF baselines on the same CPU. Further, SLIDE was observed to match the per-iteration convergence behaviour of the TF baselines. This confirms that adaptive sampling of neurons and asynchronous Stochastic Gradient Descent does not lead to poor convergence.

1.2 Motivation for TorchSLIDE

SLIDE's desirable convergence behaviour and lower per iteration training time results in its overall faster convergence time. Hence, it is a natural direction to experiment with it as a component of other larger models where it might be useful. However, the existing SLIDE repository is a standalone code-base written in pure C++. It is a self-contained experiment for training a SLIDE fully connected neural network. Apart from the core components of SLIDE described above, it also has to re-implement standard ML machinery such as optimizers, linear layers, softmax, etc. This means that any further experimentation, say using SLIDE in a transformer network, would require rebuilding the whole transformer model around a specific C++ repository. This is not suitable for rapid experimentation.

Our proposed solution for this is TorchSLIDE, a SLIDE layer module for the popular ML library PyTorch, meant to work as a drop-in replacement for the reg-

ular linear layer. It offers an easy to use API that can be used on existing PyTorch models with little modification. Our implementation stays true to SLIDE's original implementation, while also taking advantage of PyTorch's highly optimized built-in operations wherever possible. Further, unlike SLIDE's implementation that parallelizes (along the batch dimension) over the entire forward pass and backprop in a single block, our implementation naturally follows PyTorch's way of parallelizing each operation/layer independently. Our implementation is available at https://github.com/iitm-sysdl/TorchSLIDE

CHAPTER 2

TorchSLIDE Implementation

We list the major modules in our implementation and the key design choices behind them. We specifically focus on how they differ from SLIDE and New SLIDE.

2.1 Sparse Multiplication

This component performs the forward and backward propagation of a sparse linear layer. It takes in a batched matrix of input vectors (potentially sparse) and produces a batched matrix of output vectors (potentially sparse). The output matrix contains the activations of the neurons that were sampled for that input.

While PyTorch provides built-in support for sparse tensors and sparse multiplications, we cannot use them here as the sparse multiplication doesn't allow specification of the required output indexes. Also, our represenation of sparse tensors, unlike PyTorch's COO representation (refer Fig A.1 in Appendix), is more suitable for our use case. Whenever our input/output tensor batches are sparse, we use a tuple of index and value matrices for the non-zero indices. It allows us to easily parallelize over the batch and hidden dimensions. We also deviate from the original SLIDE representation by enforcing that each vector in the batch has the same number of non-zero indices. This allows for best use of PyTorch's tensor objects, and also offers better uniformity of workload for the parallel threads. In

specific cases where this cannot be ensured (like input layer), we can use padding. Further, the PyTorch matrices by default have no memory fragmentation - an optimization introduced in New SLIDE.

SLIDE represents all vectors as a tuple of a non-zero index vector and a value vector. It also uses the same vector-matrix multiplication logic irrespective of whether input, or expected output, is sparse or dense. Since it is reasonable to expect most practical use-cases to be sparse only for at most one of input or output, we use different implementations for all 4 combinations of dense/sparse input/output (refer Table 2.1). This lets us save memory and computation required to store and use redundant indexes for dense matrices. It also allows us to use different backpropagation algorithms (based on dimension used for parallelization) for the different cases. Unlike SLIDE, our implementation has no gradient collisions when the output is dense (DIDO and SIDO) (refer Figure A.2 in Appendix for pseudo code for the SIDO implementation). Note that the New SLIDE implementation also handles cases with dense input/output differently, but for the purpose of using vectorized instructions.

Name	Mode	Implementation
DIDO	Dense Input, Dense output	using PyTorch's built-in efficient matrix multiplication for both forward and backprop
SIDO	Sparse Input, Dense output	unlike SLIDE, we parallelize along output dimension during gradient update of weights and bias. This eliminates collision
DISO	Dense Input, Sparse output	Similar to SLIDE
SISO	Sparse Input, Sparse output	Similar to SLIDE

Table 2.1: Matrix Multiplication Modes

The sparse components of TorchSLIDE are implemented using C++ extensions

after comparing against the following alternatives which were found to be slower.

- Performing sparse multiplication using Pytorch Python's slicing operations leads to high memory overheads. This is because views of tensors in PyTorch (and other tensor processing libraries) are specified using offsets, strides and limits for each dimension. Thus the irregular advanced indexing used by us ends up requiring data copies of selected rows and columns in the weight matrix.

- Multi-threaded implementations using Python's threading library were also extremely inefficient due to Python's Global Interpreter Lock (GIL) and also data copies.

2.1.1 Collision Induced Deviations in Gradient Update

This section analyses the expected deviations in gradient update for the 4 forward pass modes - DIDO, SIDO, DISO, SISO. Minimal collisions in gradient values are required for HogWild updates and it is especially important to track this for TorchSLIDE. This is because while SLIDE parallelizes along the batch dimension over the full the end-to-end forward and backward pass, TorchSLIDE parallelizes over each layer separately duting both forward and backward pass. The synchronization at the end of each layer imply that TorchSLIDE could potentially face more collisions. The gradients used in this section were computed by calculating the output values, computing a weighted sum of this (weighted by random numbers from $U(0, 1)$), and then computing gradients w.r.t the weighted sum. We then get absolute deviations of gradients computed by our final TorchSLIDE implementation, compared against our high level python based implementation (mode 1). We use float32 in all experiments.

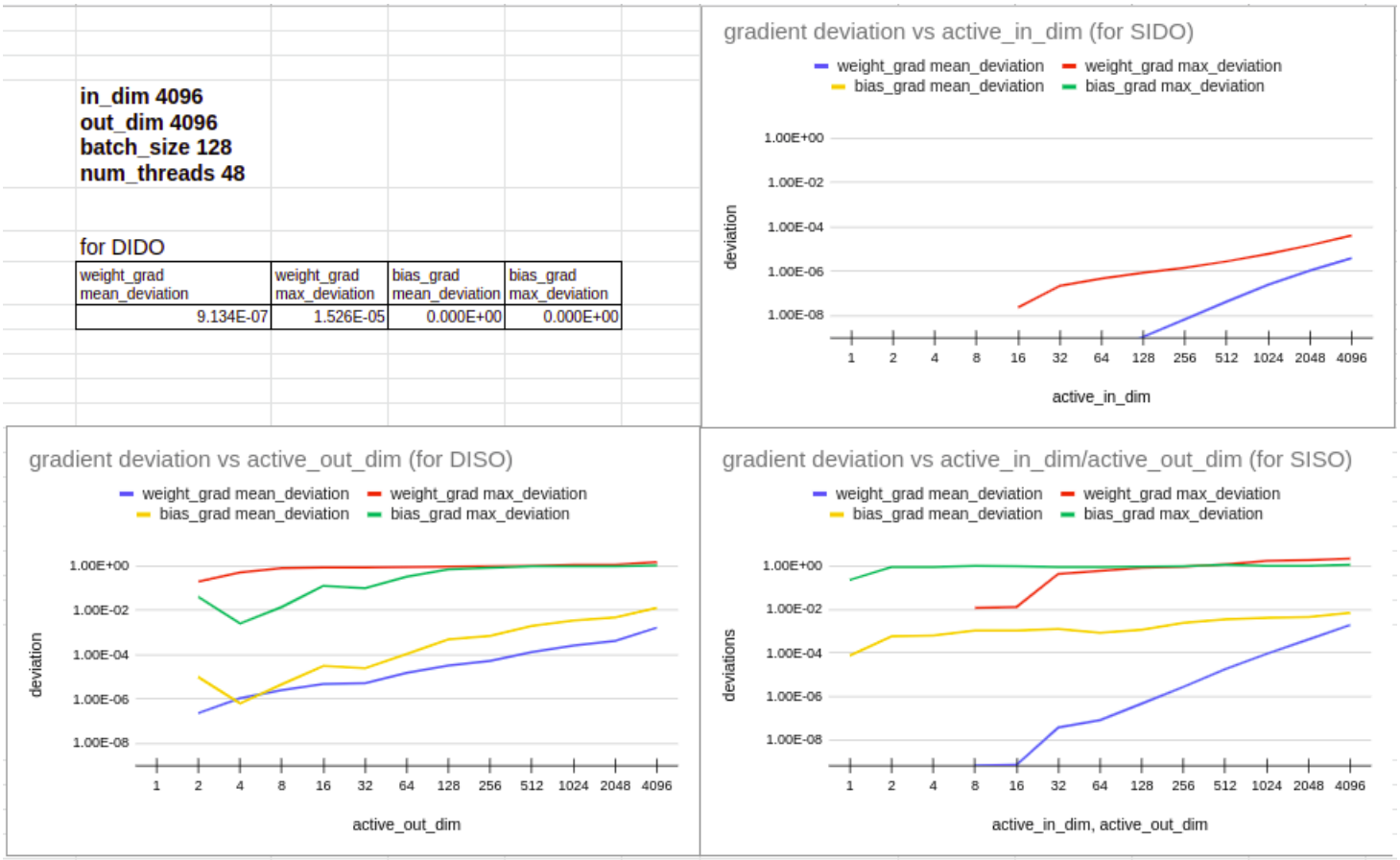


Figure 2.1: Variation of Collision Induced Gradient Deviations with Active Dimension

Figure 2.1 shows mean and max deviations of weight and bias gradients for the configuration $in\_dim = 4096$, $out\_dim = 4096$, $batch\_size = 128$ and $num\_threads = 48$. We vary applicable active dimension values over powers of 2 and plot the graphs with log scale on both axes. Any unplotted point on the graphs are because of zero deviations. The deviations for DIDO are for our naive $O(n^3)$ matrix multiplication and are included for reference. Deviation in gradient w.r.t input values is not shown as this is theoretically zero for both SLIDE and TorchSLIDE (as the computation of this is done by parallelizing along batch dimension).

We observe max weight_grad deviation of $1.53E-05$ for DIDO. This is expected since a single operation on float32 can cause deviation of up to 10^{-6} and several

of these operations are accumulated to produce each gradient value. The same applies to SIDO where our implementation, unlike SLIDE, has no gradient update collisions. For DISO and SISO on the other hand, we see that mean deviations can increase significantly if the value of *active_dim* is not small enough. Figure 2.2 shows deviations for DISO for the same layer (with *out_dim* = 524288) that was used for the previous latency experiments, for a *active_out_dim* value of 16384, much higher than our required 4096. While the max deviations are quite large, the mean deviations are small enough to indicate the applicability of HogWild.

(for DISO) **in_dim 128** **out_dim 524288** **active_out_dim 16384** **batch_size 128** **num_threads 48**	weight_grad mean_deviation	weight_grad max_deviation	bias_grad mean_deviation	bias_grad max_deviation
	5.681E-06	9.559E-01	1.427E-05	8.964E-01

Figure 2.2: Collision Induced Gradient Deviations for DISO

2.2 Hash Tables

This component initializes and applies randomly generated hash functions from a LSH family. It takes in a batched matrix of input vectors (potentially sparse) as input, and returns a batched matrix with L bucket indices for each input. The L indices of each input correspond to the application of the L meta hash functions on that input. We have currently implemented 2 commonly used hash functions - Simhash, WTA hash (described in Appendix), while keeping the API open for adding others in the future. Our Simhash implementation does not use sparse binary projection vectors as used in SLIDE because our experiments showed regular dense projections to be much faster. We also use optimized PyTorch matrix multiplication operations (over custom C++ extensions) wherever possible.

We report the latency overhead associated with the hash table computations by considering simhash during the Pre-Process and Query (Feed-Forward) phases. Figure 2.3 shows these latencies (in ms) for a linear layer with in_dim = 128 and out_dim = 524288, for various combinations of K and L. Since the Pre-Process phase occurs only every few training iterations, we normalize this overhead (over 50 iterations) to get the per iteration latency. We observe that the Pre-Process hash table overhead still significantly dominates the query phase overhead.

	Hash Tables Computation Latency (ms)				
	Dense Input in_dim = 128				Sparse Input in_dim = 128 active_in_dim = 64
	K=12 L=50	K=9 L=50	K=6 L=50	K=6 L=25	K=6 L=50
PreProcess Phase Latency (for 524288 neurons)	203.149	156.287	111.993	61.892	456.904
PreProcess Phase Latency (for 524288 neurons) normalized over 50 iter	4.063	3.126	2.240	1.238	9.138
Query Phase Latency (batch size 128)	0.094	0.097	0.086	0.088	0.183

Figure 2.3: Hash Tables Computation Latency

2.3 Buckets Tables

This component (implemented as a C++ extension) allows for storing and sampling neuron indices in a set of $L \times bucket\_index\_range$ fixed size buckets (e.g, $bucket\_index\_range = 2^K$ for SimHash). During the Pre-Process phase, it takes as input a matrix with L bucket indexes for each neuron, and "fills" the neurons in their corresponding buckets. Similar to SLIDE, we offer both FIFO and reservoir sampling policies to handle overflow in the fixed size buckets. Since we iterate over the neuron indexes in sorted order, FIFO replacement, unlike reservoir sampling, has a mild bias for larger neuron indexes during overflow. The reservoir

sampling policy on the other hand is generally slightly slower.

While the SLIDE implementation parallelizes over the number of neurons for the filling operation, we parallelize along the L dimension. This does not lead to much latency increase (if any) because the L value usually tends to be large (New SLIDE's authors use L=400 in their experiments). On the other hand, our implementation avoids any possible collisions because of two neurons trying to enter the same bucket at the same time.

During the feed-forward phase, the buckets table takes as input a matrix with L bucket indexes for each instance in the batch, and returns a matrix with the sampled neuron indexes for the instances that were taken from these buckets. Among the 3 sampling policies offered by SLIDE, they report that Vanilla Sampling is the fastest, with negligible convergence performance compared to the other modes.This method randomly goes over the L buckets one by one till we have the required number of samples (say n_s). We have thus currently implemented Vanilla Sampling, with flexibility to include the other policies if needed in the future.

SLIDE's implementation of vanilla sampling varies in two ways from the algorithm reported in their paper. First, they always iterate over all L buckets and include their contents (they also include other random neurons if the number of bucket sampled neurons is lower than a minimum threshold). Our implementation on the other hand tries to sample a fixed number of neurons, and only probes buckets till this is satisfied (we too resort to using random neurons in case the buckets are exhausted before we have the required count). Secondly, SLIDE actu-

ally uses a C++ STL map structure that internally sorts the indices. They require this as their matrix multiplication operations expect index vectors to be sorted. Our implementation on the other hand uses the unsorted C++ STL unordered_set (hash map). Thus our implementation is truly order $O(n_s)$.

Figure 2.4 shows the latency overheads (in ms) for filling neurons into buckets during Pre-Process Phase and for Querying samples during Feed-Forward Phase. The results are for the same linear layer with *out_dim* = 524288, for various combinations of K and L. We use *bucket_size* = 128 for the fixed size buckets, and the pre-process latencies are once again normalized over 50 iterations. We observe that the sampling overhead is the mildly dominating overhead for the Buckets Tables operations. Also, as expected, the observed latency increases with L as expected (accounting for the parallelization over L dimension), and is truly order $O(sample\_size)$. Further, as reported by Chen *et al.* [2020], choice of replacement policy does not affect latency much, especially since hash computations are the major overhead during pre-process phase.

2.4 Cross Entropy Loss

We implement simple wrappers for softmax_cross_entropy_with_logits and binary_cross_entropy_with_logits over Pytorch Python's built-in loss functions. These are compatible with our representation of sparse output logits. We use softmax_cross_entropy as used in SLIDE and assumes equal probability for all true classes.

Buckets Tables Latency		K=12 L=50	K=9 L=50	K=6 L=50	K=6 L=25
PreProcess Phase Latency (for 524288 neurons) (ms)	fill_buckets_FIFO	45.301	37.706	40.584	12.028
	fill_buckets_reservoir_sampling	41.565	43.333	50.366	12.523
PreProcess Phase Latency (for 524288 neurons) normalized over 50 iter (ms)	fill_buckets_FIFO	0.906	0.754	0.812	0.241
	fill_buckets_reservoir_sampling	0.831	0.867	1.007	0.250
Query Phase Latency (for batch_size 128) (ms)	vanilla sampling sample_size 1024	0.586	0.616	0.617	0.629
	vanilla sampling sample_size 2048	1.168	1.201	1.237	1.250
	vanilla sampling sample_size 4096	2.366	2.444	2.433	2.092
	vanilla sampling sample_size 8192	3.762	3.943	3.985	2.472

Figure 2.4: Buckets Tables Computation Latency

CHAPTER 3

Evaluations

In this section we compare the latency and convergence behaviour of TorchSLIDE against various implementations like Pytorch baselines, SLIDE and New SLIDE. We perform experiments on 3 extreme classification datasets - Delicious200K, Amazon-670K, WikiLSHTC-325K from the Extreme Classification Repository (Bhatia *et al.* [2016]). The statistics of the datasets are included in Table 3.1. We train simple fully connected networks on these datasets with one hidden layer. Given the wide output layer of these networks, most of the computation involved is in the last layer.

Dataset	Feature Dim	Label Dim	Train Size	Test Size
Delicious-200K	782,585	205,443	196,606	100,095
Amazon-670K	135,909	670,091	490,449	153,025
WikiLSHTC-325K	1,617,899	325,056	1,778,351	587,084

Table 3.1: Statistics of the datasets

3.1 Infrastructure

All experiments are conducted on a server, utilizing 48 threads of a Intel Xeon Platinum 8160 2.10GHz processor, and one NVIDIA Geforce GTX 1080 Ti GPU. The server has an Ubuntu 18.04.5 LTS system installed with PyTorch 1.6. None of

our experiments use the hugepages optimization mentioned in Chen *et al.* [2020]. Also, our processor is based on the skylake architecture, and hence we do not use the BF16 optimization used in new SLIDE.

3.2 Hyperparameters

For all 3 datasets, we use fully connected networks with a hidden layer size of 128. We train them using the Adam optimizer, with an initial learning rate of 0.0001. For all variants of SLIDE models, sparse operations are used only for the output layer, with a rehashing frequency of 50 iterations. For all TorchSLIDE variants, the number of labels to sample is set equal to the average number of labels sampled during corresponding SLIDE runs.

3.3 Amazon-670K

Figure 3.1 and Table 3.2 show the accuracy w.r.t iterations and latency per 100 iterations for various models. We use batch_size=128, and K=6, L=400 with DWTA hash for the SLIDE variants. For TorchSLIDE, we use num_label_samples = 32768. Our experiments show that TorchSLIDE matches the convergence of dense training. It is faster than SLIDE by 2.6X and slower than New SLIDE by 1.2X. The usage of PyTorch over Tensorflow, and the difference in hardware used, leads to speedup values w.r.t dense baselines being different from what was reported in the original papers.

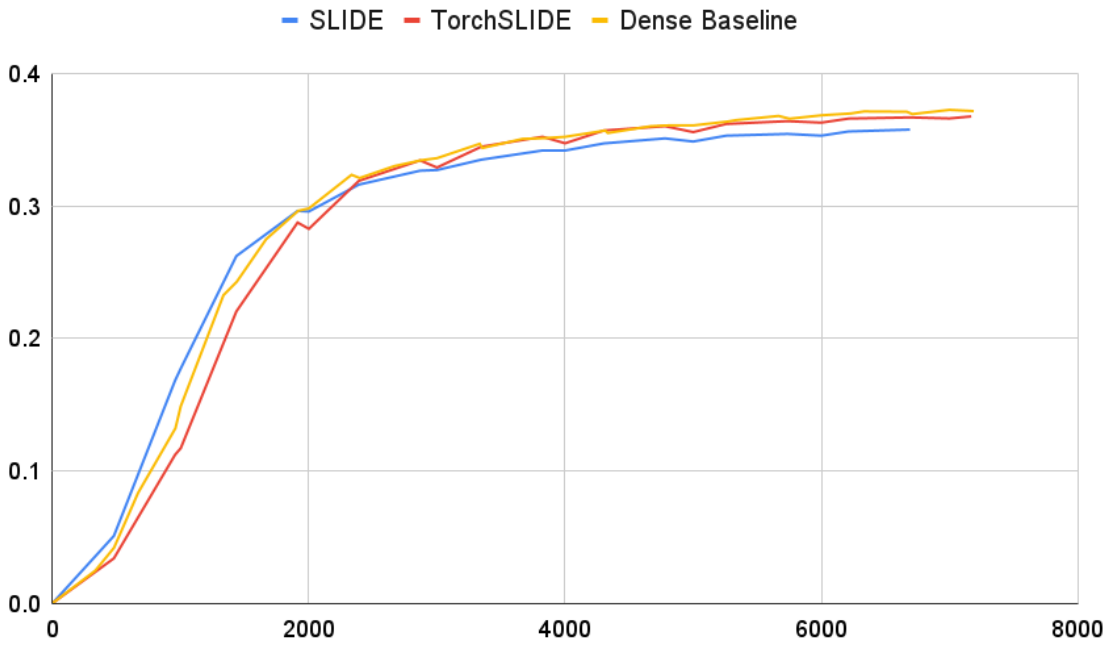


Figure 3.1: Amazon670K - p@1 Accuracy vs Train Step

PyTorch-CPU	Pytorch-GPU	SLIDE	New SLIDE	Torch SLIDE
550 s	71.8 s (trained using gradient accumulation)	1062.9 s	326.1 s	397.6 s

Table 3.2: Amazon670K - Latency (s) per 100 train iterations

3.4 WikiLSHTC-325K

Figure 3.2 and Table 3.3 show similar results for the WikiLSHTC-325K dataset. We use batch_size=256, and K=5, L=350 with DWTA hash for the SLIDE variants. For TorchSLIDE, we use num_label_samples = 40000. Once again, our experiments show that TorchSLIDE matches the convergence of dense training. However, we see that the speedups vary significantly based on the hyperparameters used for the dataset. TorchSLIDE is faster than SLIDE by 1.5X, but slower than New SLIDE by 2.3X.

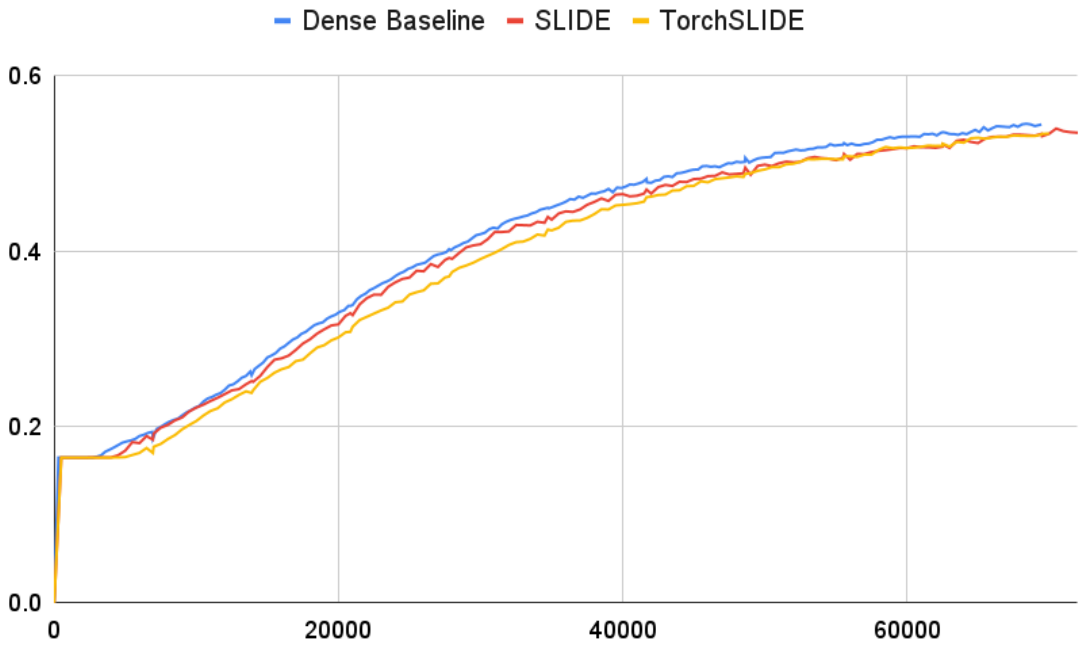


Figure 3.2: WikiLSHTC325K p@1 Accuracy vs Train Step

PyTorch-CPU	Pytorch-GPU	SLIDE	New SLIDE	Torch SLIDE
167.9 s	35.7 s	229.4 s	67.5 s	156.9 s

Table 3.3: WikiLSHTC325K Latency (s) per 100 train iterations

3.5 Delicious-200K

Figure 3.3 shows the convergence plot for Delicious200K dataset. We infer that the poorer convergence of SLIDE and TorchSLIDE in this dataset is a consequence of its distribution of true labels. Unlike Amazon670K and WikiLSHTC325K that have a uniformly small number of true labels for each sample, the true label counts of samples in this dataset ha a long tail distribution as shown in 3.4.

.

Figure 3.3: Delicious200K - p@1 Accuracy vs Train Step

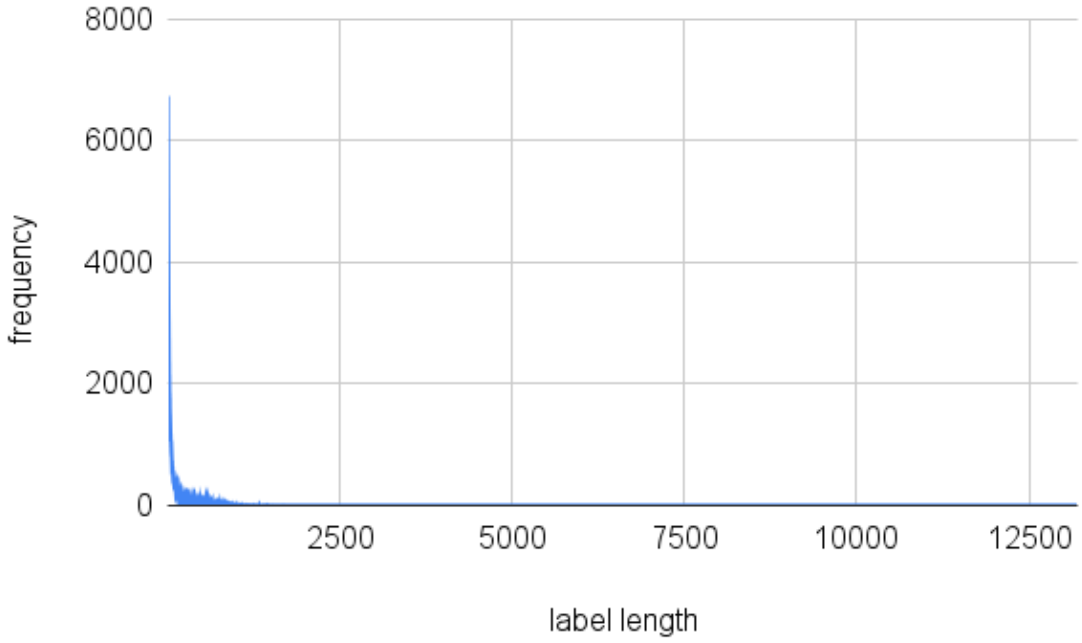


Figure 3.4: Frequency of True Label Lengths (Train Data - Delicious 200k)

CHAPTER 4

Profiling Experiments

In this section we perform latency profiling experiments on the different components of a simple 3 layer network (dense vs torchSLIDE) during training. Figure 4.1 shows the latency profile for different components of a TorchSLIDE network, for various hyperparameter configurations. The default TorchSLIDE configuration used is similar to the fully connected network used for Amazon670K in [1] - batch_size = 128, input_dim = 131072, non_zero_input_dim = 128, hidden_layer_dim = 128, output_dim = 524288, nonzero_output_dim = 4096, last_layer_K = 6, last_layer_L = 50, bucket_size = 128, rehash_freq = 50. Each subsequent configuration varies exactly one parameter compared to the default. For each of these configurations, Figure 4.2 shows the profile for a corresponding regular dense PyTorch baseline (with the same applicable parameters) without torchSLIDE.

In order to track the latency of each line of code, we use the Profiler APIs provided in the new PyTorch v1.7.1. We use the stack trace option, sort the latencies of top level events by cpu_time_total and get the cumulative latency for each line of code. We then attribute the line to the corresponding component of TorchSLIDE. The reported latencies (s) are for 500 iterations. We only report components with non-negligible latencies.

- Comparison of the first layer's forward pass between torchslide and dense (b5,b6 vs t1,t2) shows that torchslide in SIDO mode is much faster than the baselines using PyTorch's built-in sparse_coo_tensors for the same.

	Operator Description	Total profiled time	first slide layer forward	second slide layer forward	Softmax Loss Computation	gradient computation and parameter update
Expt No. TorchSLIDE	**Config Changes from Default TorchSLIDE**					
t1	default	135.273s	2.609	30.713	1.064	89.719

t2	# varying in_dim, active_in_dim (maintaining in sparsity)
	input_dim = 524288
	non_zero_input_dim = 512
		236.415s	10.027	32.547	6.135	153.835

t3	# varying hidden_layer_dim
	hidden_layer_dim = 512
		551.455s	9.723	113.386	2.717	376.098

t4	# varying output_dim, active_output_dim (maintaining out sparsity)
	output_dim = 2097152
	nonzero_output_dim = 16384
		445.794s	2.16	119.771	5.385	302.846

t5	# varying active_output_dim (varying sparsity)
	nonzero_output_dim = 1024
		126.908s	2.064	15.432		98.551

t6	# varying active_output_dim (varying sparsity)
	nonzero_output_dim = 16384
		211.406s	1.945	103.959	4.86	90.964

t7	# varying active_out_dim (sparsity) 3
	nonzero_output_dim = 131072
		749.518s	2.273	192.534	53.373	94.138

t8	# varying active_out_dim (sparsity) 4
	nonzero_output_dim = -1 # dense
		323.606s	2.504	44.642	74.283	98.565

t9	# varying L (num_hashes) 1
	last_layer_L = 25
		140.647s	2.509	27.649	2.311	96.936

t10	# varying L (num_hashes) 2
	last_layer_L = 100
		130.232s	2.545	27.117		89.358

Figure 4.1: TorchSLIDE Profiled Component Latencies (in seconds)

- Comparing b5, b7, b8 with t1, t3, t4 respectively, we see that the second DISO layer of torchslide has latency similar/slightly lower than the corresponding dense layer, for the sparsity of 4096/524288 (0.78%). Further, the advantage of torchslide seems to increase as the output_dim is increased to 2M.
- Comparing b5, b8 with t1, t4 respectively, the softmax loss computation for torchslide (using sparse outputs) is over 100x faster than for the dense case. This is a direct result of the approx 1% sparsity, and leads to the most speedup.
- t1, t5, t6, t7 use the same output_dim and vary the sparsity (nonzero_output_dim). As the sparsity decreases, we see that the softmax loss computation becomes more prominent compared to the forward layer.
- t1, t9, t10 differ in the number of hash functions computed. This doesn't seem to affect the second layer forward times or rehash times (which is negligible anyway) by much.

	Component Description	Total profiled time	first layer forward	second layer forward	softmax loss computation	gradient computation anf parameter update
Expt No. Dense Baseline	**Config Changes from Default Dense Baseline**					
b5	default	334.013s	80.585	35.117	172.803	83.138
b6	input_dim = 524288 non_zero_input_dim = 512	535.905s	360.882	36.008	171.087	136.432
b7	hidden_layer_dim = 512	776.383s	335.032	97.293	172.713	315.206
b8	output_dim = 2097152	1129.053s	83.74	141.471	615.825	305.292

Figure 4.2: Dense PyTorch Baseline Profiled Component Latencies (in seconds)

CHAPTER 5

Conclusions and Future Work

We provide TorchSLIDE, an easy to use PyTorch implementation of the LSH based sparse approximation for the linear layer, that was introduced in SLIDE Chen *et al.* [2020] . Our module is a drop-in replacement for the linear layer module, and can conveniently be introduced in existing PyTorch models with little code change. We have improved on SLIDE using PyTorch's built-in optimized functions wherever possible, resulting in noticeable speedup of up to 2.6x. On the other hand, our implementation does not compromise on convergence behaviour. While TorchSLIDE could potentially be slower than the optimized New SLIDE depending on the configuration, it is still significantly easier to use as a component of a larger model.

One future direction is to design a GPU version of TorchSLIDE. The other direction we look at is the application of TorchSLIDE to attention based models like Transformers. One direct way of doing this is to replace the final output linear layer with TorchSLIDE when the output dimension is large. The other direction is to modify our implementation to decrease the order $O(n^2)$ computation required in self-attention. Each index in the sequence can attend to only a few other indices in the sequence determined by LSH. Apart from being a computational advantage for long sequence tasks, the hard attention mechanism may also lead to insights for interpretation.

APPENDIX A

Sparse Multiplication

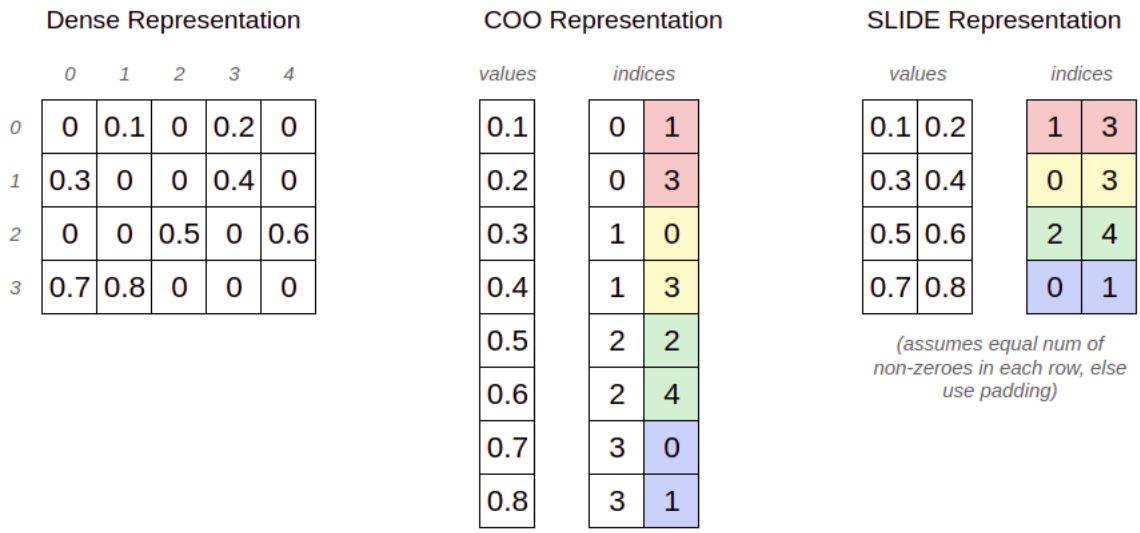


Figure A.1: Representations of Sparse Matrices

```
// Algorithm 1: SIDO forward pass
Inputs:
    in_values           // float[batch_size][active_in_dim]
    active_in_indices   // int[batch_size][active_in_dim]
    weights             // float[out_dim][in_dim]
    bias                // float[out_dim]
Outputs:
    out_values          // float[batch_size][out_dim]
Procedure:
    parallel_for (i = 1 to batch_size)
        for (j = 1 to out_dim)
            out_values[i][j] = bias[j]
            for (k = 1 to active_in_dim)
                index = active_in_indices[i][k]
                out_values[i][j] += in_values[i][k]*weights[j][index]
```

```
// Algorithm 2: SIDO backward pass (representative of algo used by SLIDE)
// This has update collisions for grad_weights and grad_bias
Inputs:
    grad_out_values     // float[batch_size][out_dim]
    in_values           // float[batch_size][active_in_dim]
    active_in_indices   // int[batch_size][active_in_dim]
    weights             // float[out_dim][in_dim]
Outputs:
    grad_in_values      // float[batch_size][active_in_dim]
    grad_weights        // float[out_dim][in_dim]
    grad_bias           // float[out_dim]
Procedure:
    parallel_for (i = 1 to batch_size)
        for (j = 1 to out_dim)
            for (k = 1 to active_in_dim)
                index = active_in_indices[i][k]
                grad_in_values[i][k] += grad_out_values[i][j]*weights[j][index]
                grad_weights[j][index] += grad_out_values[i][j]*in_values[i][k]
            grad_bias[j] += grad_out_values[i][j]
```

```
// Algorithm 3: SIDO backward pass (used in TorchSLIDE)
// This does not have update collisions for grad_weights and grad_bias
Inputs:
    grad_out_values     // float[batch_size][out_dim]
    in_values           // float[batch_size][active_in_dim]
    active_in_indices   // int[batch_size][active_in_dim]
    weights             // float[out_dim][in_dim]
Outputs:
    grad_in_values      // float[batch_size][active_in_dim]
    grad_weights        // float[out_dim][in_dim]
    grad_bias           // float[out_dim]
Procedure:
    parallel_for (i = 1 to batch_size)
        for (j = 1 to out_dim)
            for (k = 1 to active_in_dim)
                index = active_in_indices[i][k]
                grad_in_values[i][k] += grad_out_values[i][j]*weights[j][index]
    parallel_for (j = 1 to out_dim)
        for (i = 1 to batch_size)
            for (k = 1 to active_in_dim)
                index = active_in_indices[i][k]
                grad_weights[j][index] += grad_out_values[i][j]*in_values[i][k]
            grad_bias[j] += grad_out_values[i][j]
```

Figure A.2: SIDO Forward and Backward Pseudo Code

APPENDIX B

Locality Sensitive Hash Functions

B.1 SimHash

SimHash is a popular LSH for the cosine similarity measure. A single Simhash function uses a d dimensional (hidden dimension) random pre-generated vector (let v_j) with components taking only three values +1, 0, 1. Let the meta-hash function be made of K such functions. Then, for a given input vector x, the corresponding returned bucket index is a K-bit integer, where the j^{th} bit is 1 if and only if $v_j \bullet x < 0$. The components only take +1, 0, 1 so that dot product can be obtained using just addition instead of multiplication. Another common optimization is to use sparse random projection, where only a fraction of the indices in the random vector (chosen as $d/3$ in SLIDE) are non-zero +1 or -1. This is the approach used in SLIDE.

We tried implementing C++ extensions for computing the hash functions and meta-hash functions (from the hash values) as done in SLIDE. We also evaluated direct PyTorch Python based implementations for the same, where the hash function computation is performed simply as a matrix multiplication of the input batch matrix and the matrix of random vectors of the hashes (without sparse projections and without assuming components to be in +1, 0, -1). Table B.1 compares the latency for the two approaches for $K = 9, L = 50, d = 128$, for different batch sizes. The larger batch sizes (65536, 524288) correspond to hashing the batch of neuron

weights during the Pre-Process phase, and the batch size 128 corresponds to the input batch during feed-forward pass phase. Based on these latencies, we use PyTorch matrix multiplication for computing hash functions and a C++ extension for computing meta-hash functions from the output hash values. In case of sparse inputs, we compute both using the C++ extension as before.

Batch Size	Hash Function Latency (ms)		Meta-Hash Function Latency (ms)	
	py (matrix mul)	c++ (sparse projection)	py	c++
128	0.078	0.451	0.293	0.0331
65536	41.595	221.000	64.725	12.384
524288	129.728	1617.764	500.689	75.199

Table B.1: Simhash Implementations Comparison. Results for K=9, L=50, d=128

B.2 WTAHash

Winner Take All (WTA) hash (Yagnik *et al.* [2011]) is a sparse embedding method that transforms the input feature space into binary codes such that Hamming distance in the resulting space closely correlates with rank similarity measures. For a given d dimensional input vector, a WTA Hash picks a random pre-generated m-dimensional permutation ($m << d$) of its elements. The value of the hash for this input is the index of the largest element in the permutation, within the permutation. K such hash values are then bit-wise concatenated to get the corresponding meta-hash value (bin index) as in SimHash.

REFERENCES

Bhatia, K., **K. Dahiya**, **H. Jain**, **A. Mittal**, **Y. Prabhu**, and **M. Varma** (2016). The extreme classification repository: Multi-label datasets and code. URL `http://manikvarma.org/downloads/XC/XMLRepository.html`.

Chen, B. (*a*). Slide github repository. URL `https://github.com/keroro824/HashingDeepLearning`.

Chen, B. (*b*). Slide mlsys presentation. URL `https://mlsys.org/media/Slides/mlsys/2020/balla(02-10-30)-02-10-55-1410-slide__in.pdf`.

Chen, B., **T. Medini**, **J. Farwell**, **S. Gobriel**, **C. Tai**, and **A. Shrivastava** (2020). Slide : In defense of smart algorithms over hardware acceleration for large-scale deep learning systems.

Daghaghi, S., **N. Meisburger**, **M. Zhao**, and **A. Shrivastava** (2021). Accelerating slide deep learning on modern cpus: Vectorization, quantizations, memory optimizations, and more. *Proceedings of Machine Learning and Systems*, **3**.

Kitaev, N., **Ł. Kaiser**, and **A. Levskaya** (2020). Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*.

Recht, B., **C. Re**, **S. Wright**, and **F. Niu** (2011). Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems*, **24**, 693–701.

Yagnik, J., **D. Strelow**, **D. A. Ross**, and **R.-s. Lin**, The power of comparative reasoning. *In 2011 International Conference on Computer Vision*. IEEE, 2011.