TorchSLIDE

LSH based Sub-Linear Neural Networks on PyTorch

E Santhosh Kumar (CS16B107)

Prof. Pratyush Kumar (Guide)

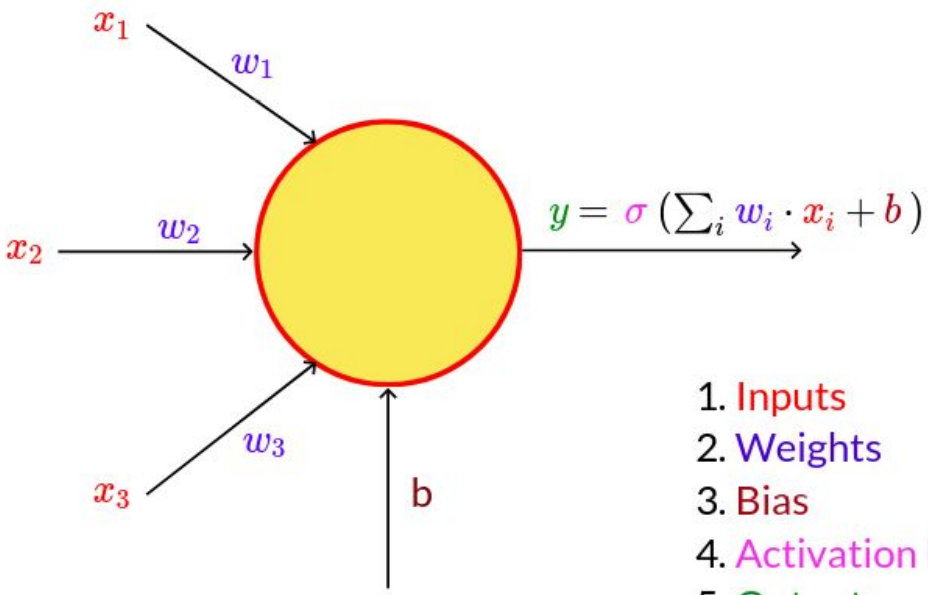
DDP 2 Presentation - 25 Jun 2021

Outline

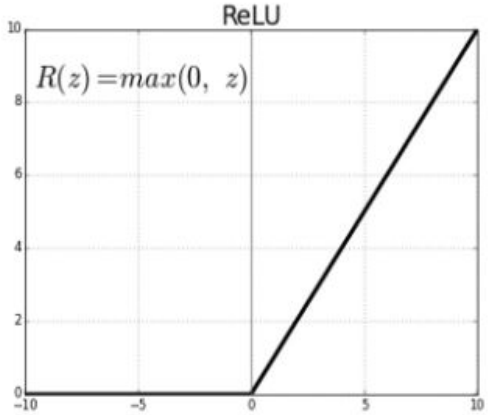
- Background
  - SLIDE (Chen et al.,)
- Motivation
- TorchSLIDE Implementation
  - Sparse Multiplication
  - Hash Tables
  - Buckets Tables
  - Cross Entropy Loss
- Evaluations
- Profiling Experiments
- Next Steps

Background

Neuron - The Building Block



1. Inputs
2. Weights
3. Bias
4. Activation Function
5. Output



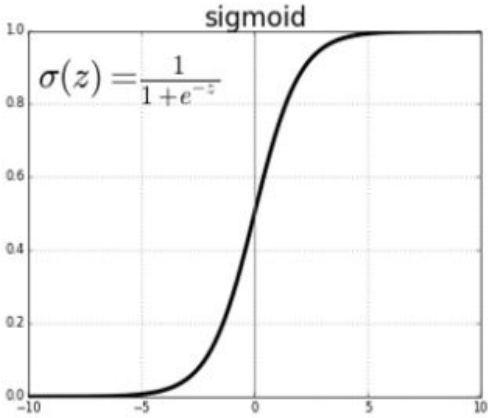
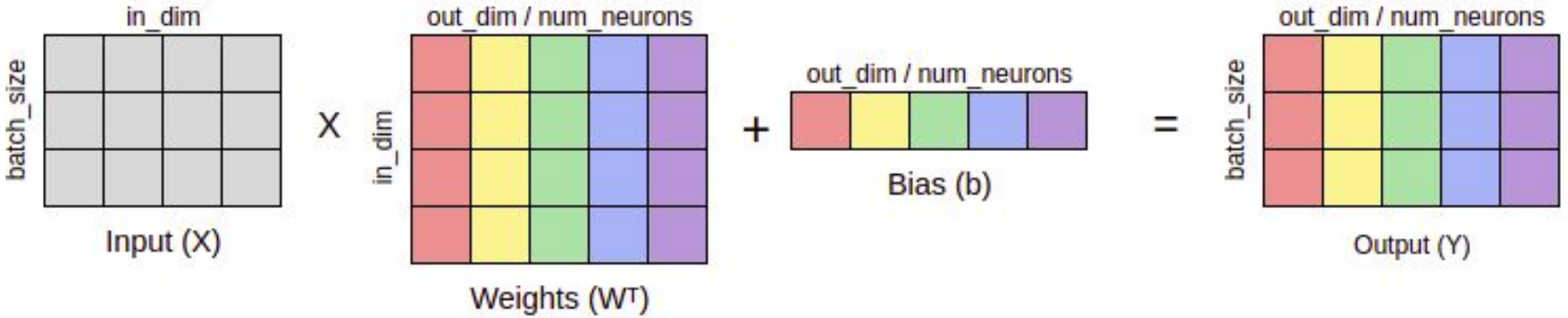


Image Sources: [1] , [2]

Fully Connected Layer



forward pass: $Y = X \times W^T + b$

backpropagation: $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \times W$ $\frac{\partial L}{\partial W} = \left(\frac{\partial L}{\partial Y}\right)^T \times X$ $\frac{\partial L}{\partial b} = \sum_{batch} \frac{\partial L}{\partial Y}$

- Specialized hardware accelerators (GPUs, TPUs) designed to efficiently perform exact matrix multiplications.

- Do we need all values in Output matrix? No, we typically require only the outputs with high activation
  - Sampling few neurons in proportion of activations is enough (Adaptive Dropouts, Ba et al.)
  - Output layer in extreme classification tasks
  - self attention over long sequences (e.g, Reformer, Kitaev et al.)

- Can we identify and compute only the outputs with high activation?

- Can this be exploited to achieve the performance of hardware accelerators on CPUs with modest multi-core parallelism?

Locality Sensitive Hashes (LSH)

- An LSH family of functions has the property that similar inputs in the domain have a higher probability of collision of their corresponding outputs.
- Sufficient Condition: a hash family H is an LSH family for a similarity metric Sim if

$$Pr_H(h(x) = h(y)) = f(Sim(x, y))$$

where h~H is a randomly chosen hash function, x and y are inputs and f is a monotonic function.

- An LSH family is sufficient for efficient nearest-neighbor search queries in sub-linear time (Indyk & Motwani, 1998)

e.g, SimHash

$$h_r(x) = \text{sign}\left(\sum_i x_i r_i\right) \quad , \text{where } x, r \in \Re^d$$

$$\textit{When } r_i \sim \text{N}(0,1) \text{ and } \rho = \rho(x,y) = \text{cosine similarity between } x \text{ and } y,$$

$$\text{collision prob.} = \Pr\left(h_r(x) = h_r(y)\right) = 1 - \frac{1}{\pi} cos^{-1}(\rho)$$

Collision probability is monotonic in cosine similarity

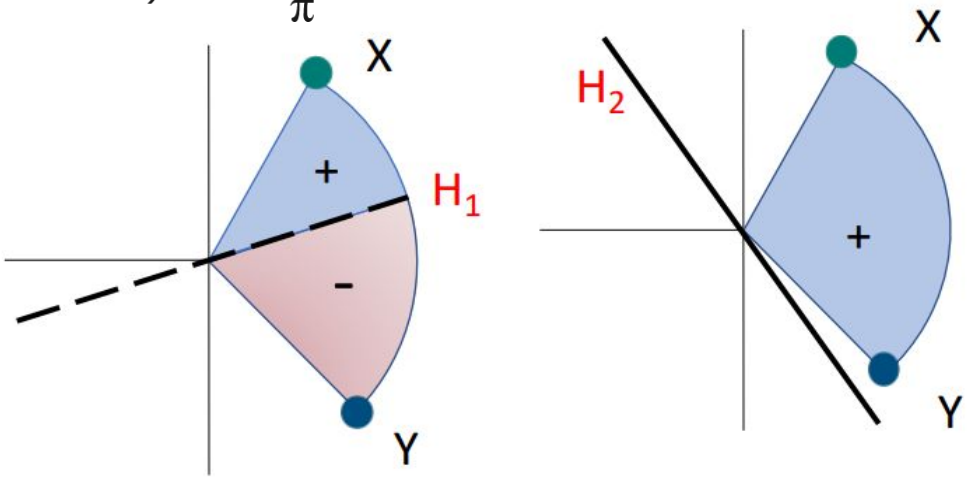


Image Sources: [3]

Sampling Nearest Neighbours using LSH

- Output of K different hash functions concatenated to form a meta-hash function.
- Each point x_i in sample space is put in K independent tables of buckets, where the bucket index of insertion is the output of meta-hash function.
- For a given query vector q, each x_i is sampled with probability

$$const \times \left(1 - \left(1 - Pr_{collision}\left(x_i, q \right)^K \right)^L \right)$$

(const determined by capacity of buckets)

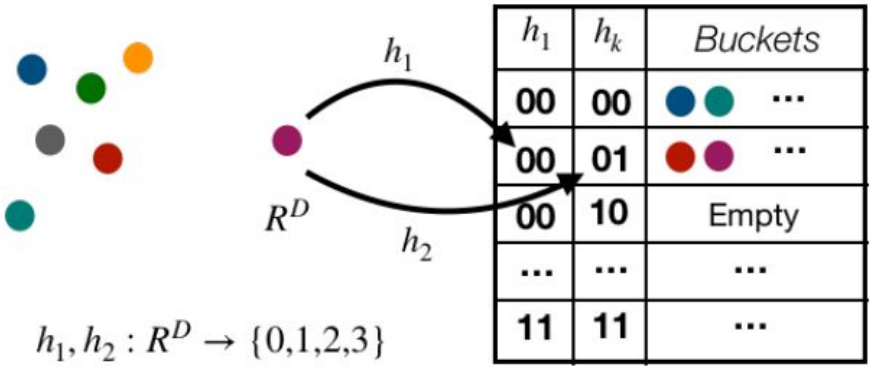


Image Sources: [3]

SLIDE: Sub-LInear Deep learning Engine (Chen et al.,)

For each input in batch, uses LSH to sample and compute only the outputs of neurons with potentially high activation

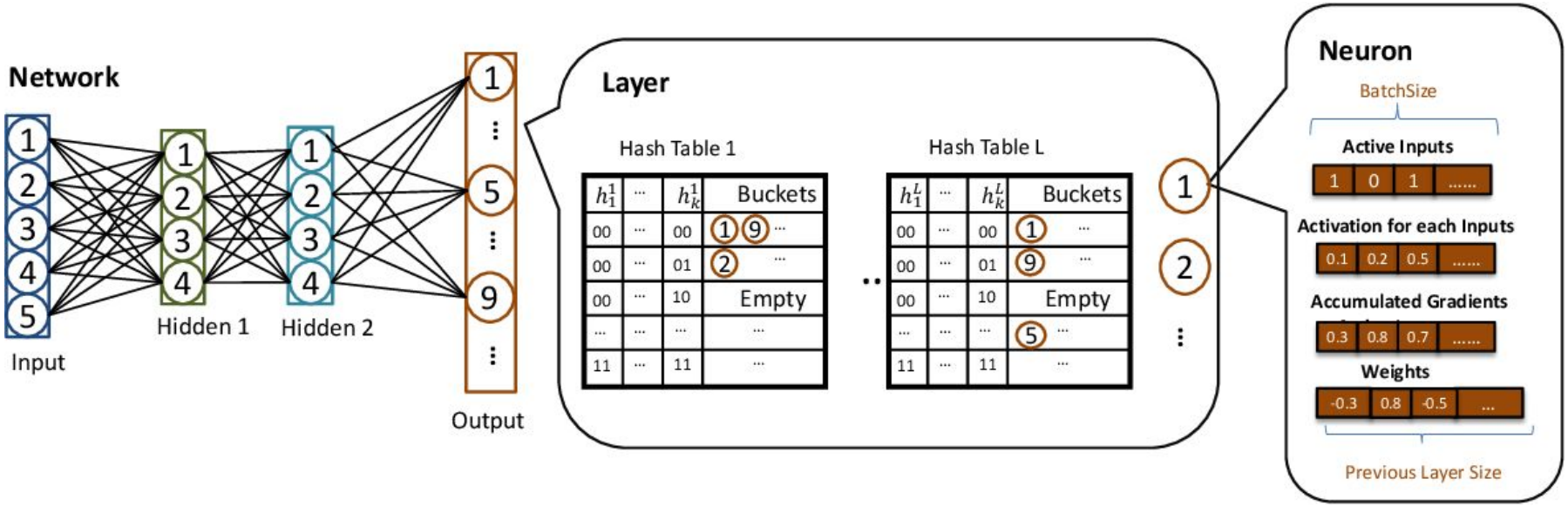
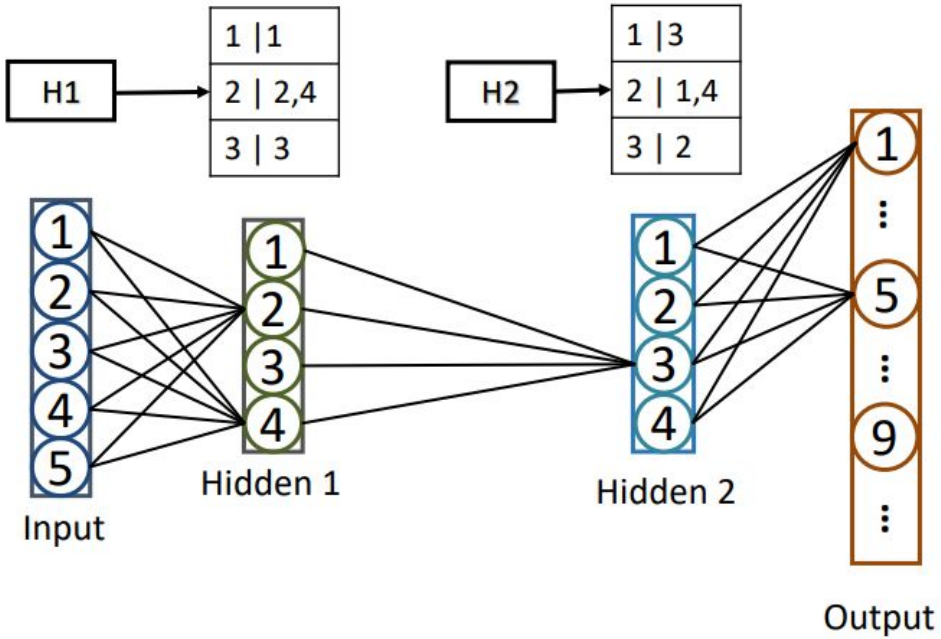


Image Sources: [3]

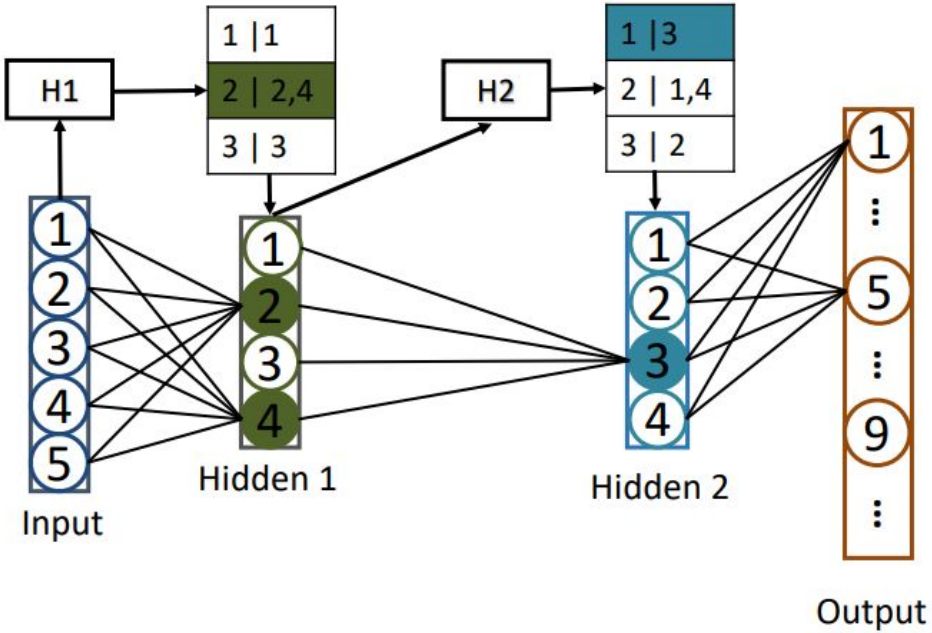
Pre-Process Phase



- Fill indices of neurons into bucket tables based on meta-hash function value of their weight vectors.
- Performed during initialization, and after every few iterations of weight updates.
- Replacement policies to handle overflow of fixed size buckets.
- Complexity linear in no. of neurons

Image Sources: [3]

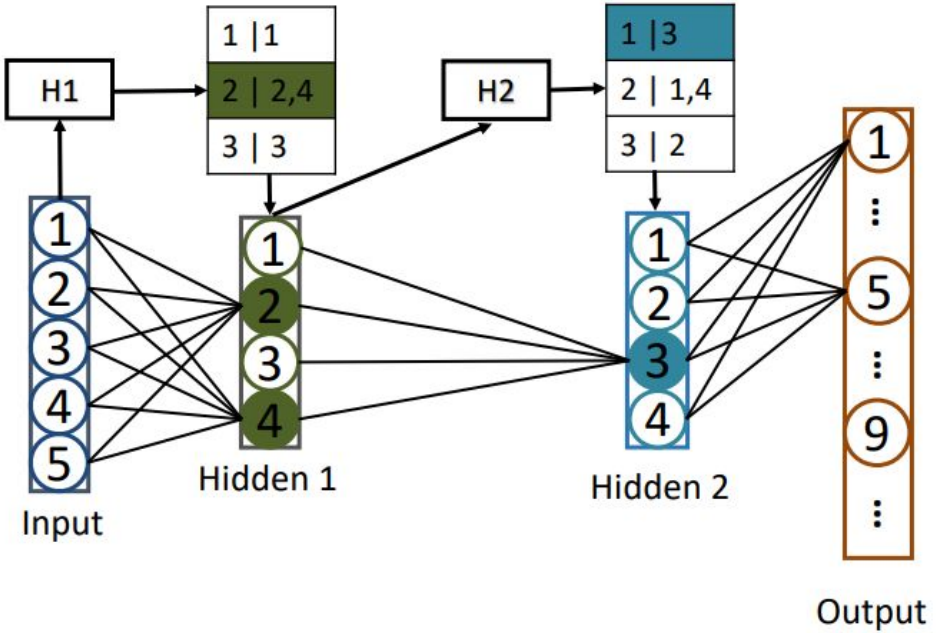
Sparse Forward Pass



- Compute each meta-hash function (bucket index) for each input in batch.
- Sample union of neurons present in chosen buckets.
- Compute activations only for sampled neurons
- Parallelize over batch. Each input processed independently.
- Complexity sub-linear in no. of neurons

Image Sources: [3]

Sparse Backpropagation



- Partial gradients propagated only through sampled neurons.
- Weights connecting sampled neurons in consecutive layers are updated.
- Parallelized over batch. High sparsity ensures minimum collision in weight updates (HogWild)
- Complexity sub-linear in no. of neurons

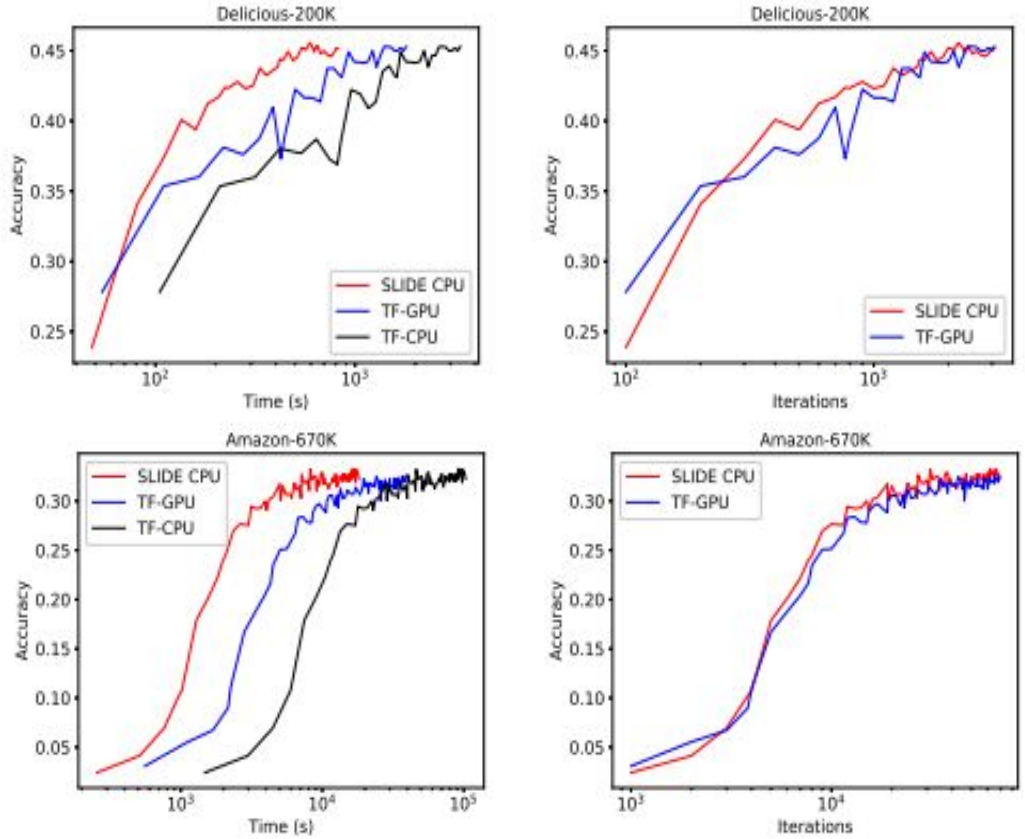
Image Sources: [3]

Results

Dataset	Feature Dimension	avg. Feature Sparsity	Label Dimension	avg. True Label Sparsity
Delicious-200K	782,585	297.4 (0.038%)	205,443	75.5 (0.037%)
Amazon-670K	135,909	74.7 (0.055%)	670,091	5.4 (0.0008%)

- Network Architecture: Fully Connected NN with 128 neuron hidden layer
- CPU: 22-core/44-thread Intel Xeon E5-2699A v4 2.40GHz
- GPU: NVIDIA Tesla V100 Volta 32GB
- SLIDE: C++ with OpenMP parallelism (Hugepages, Intel-AVX instructions)
- Tensorflow v1.12 Baselines

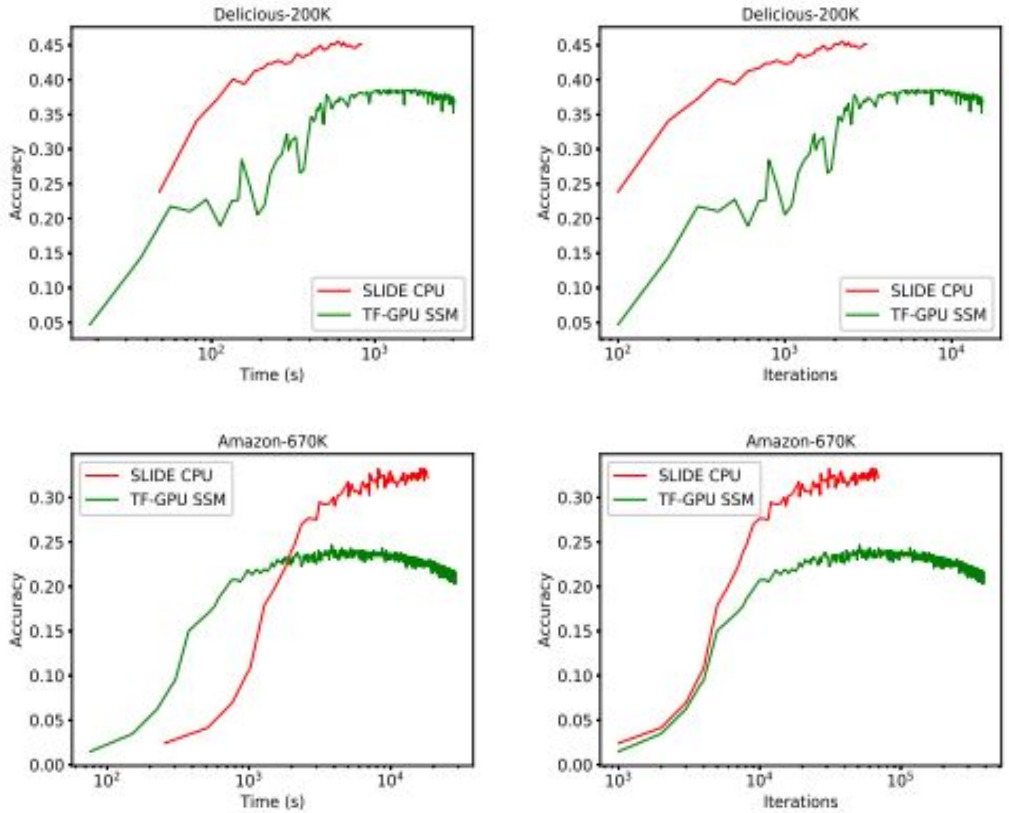
Results - Comparing with TF baselines



- SLIDE up to 3.5x faster than TF-GPU
- SLIDE up to 10x faster than TF-CPU
- matches convergence behaviour of baselines
- speedup increases with larger dimensions and more sparsity.

Image Sources: [4]

Results - Comparing with Sampled Softmax



SLIDE sampling 0.5% neurons on average outperforms TF-GPU Sampled Softmax even when sampling 20% neurons

Image Sources: [4]

Motivation

SLIDE is implemented from scratch in C++ (github)

- Forced to re-implement standard ML machinery
- Difficult to apply/extend to other models. Needs non-transferable re-implementation around a specific repo
- Does not take advantage of optimized ML libraries
- compilation required for every change

torchSLIDE

- PyTorch CPU implementation of SLIDE (github)
- Offers an easy-to-use API. Acts as a drop-in replacement for regular linear layer
- Replaces SLIDE logic with optimized PyTorch operations wherever possible.
- Components
  - Sparse Multiplication
  - Hash Tables
  - Buckets Tables
  - Cross Entropy Loss

Implementation

1. Sparse Multiplication

- **Input (potentially sparse) ⇒ Output (with activations for sampled neurons)**
- Backpropagation for the same

PyTorch's built-in sparse multiplication is not usable here

Dense Representation

	0	1	2	3	4
0	0	0.1	0	0.2	0
1	0.3	0	0	0.4	0
2	0	0	0.5	0	0.6
3	0.7	0.8	0	0	0

COO Representation

values	indices
0.1	0 1
0.2	0 3
0.3	1 0
0.4	1 3
0.5	2 2
0.6	2 4
0.7	3 0
0.8	3 1

SLIDE Representation

values		indices	
0.1	0.2	1	3
0.3	0.4	0	3
0.5	0.6	2	4
0.7	0.8	0	1

(assumes equal num of non-zeroes in each row, else use padding)

1. Sparse Multiplication

TorchSLIDE	SLIDE
Uses batched input/output. Each layer parallelised independently.	End-to-end parallelisation over whole network. Each input processed independently.
Indexes used only for sparse vectors. Requires equal sparsity for all inputs in batch.	Indexes used for all vectors. Inputs in batch need not have equal sparsity.
Specialized implementation for each configuration ● Dense Input Dense Output (DIDO) ● Sparse Input Dense Output (SIDO) ● Dense Input Sparse Output (DISO) ● Sparse Input Sparse Output (SISO)	Common implementation for all configurations.
Only configurations with Sparse Output exposed to weight/bias update collisions during backprop.	All configurations exposed to weight/bias update collisions during backprop.

Sparse Multiplication

Implementation Options Considered

1. Directly using Pytorch Python's slicing operations - not used because of data copy overhead in advanced indexing

```
def diso(in_values, active_out_indices, weights, bias):
    # dense input sparse output
    # in_values = (batch x in_dim)
    # active_out_indices = (batch x active_out_dim)
    # out_values = (batch x active_out_dim)

    active_weights = weights[[active_out_indices]]
    active_biases = bias[[active_out_indices]]
    out_values = torch.bmm(active_weights, in_values.unsqueeze(-1)).squeeze(-1) + active_biases
    return out_values
```

1. Sparse Multiplication

Implementation Options Considered

2. Using Python's multi-threading library - not used due to extreme inefficiencies from GIL and data copy

3. Using Pytorch support for custom C++ extensions (chosen method for building most components). It offers
   - parallel for loops
   - data element accessors
   - data-type based dispatches

   https://pytorch.org/tutorials/advanced/cpp_extension.html

1. Sparse Multiplication - Configurations

Differences in TorchSLIDE compared to SLIDE

	Forward Pass	**Backprop**
DIDO	uses PyTorch Matrix Multiplication	uses PyTorch Matrix Multiplication | No Collisions
SIDO	similar to SLIDE, except no indexes for output	Weight gradients and input gradients computation parallelized separately | No Collisions
DISO	similar to SLIDE, except no indexes for input	similar to SLIDE
SISO	similar to SLIDE	similar to SLIDE

Example pseudo-code follows

```
--------------------------------------------------------------------------------
// 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]
```

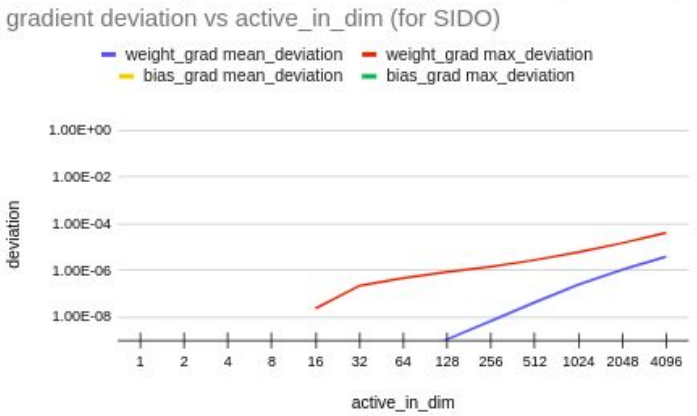
1. Sparse Multiplication - Collisions in Gradient Update

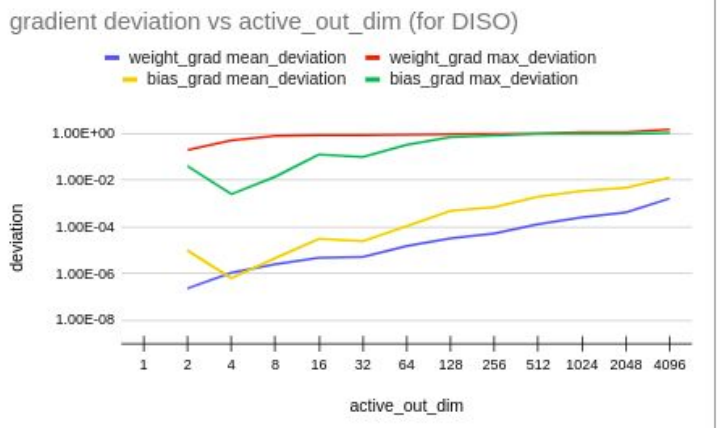
- using float32

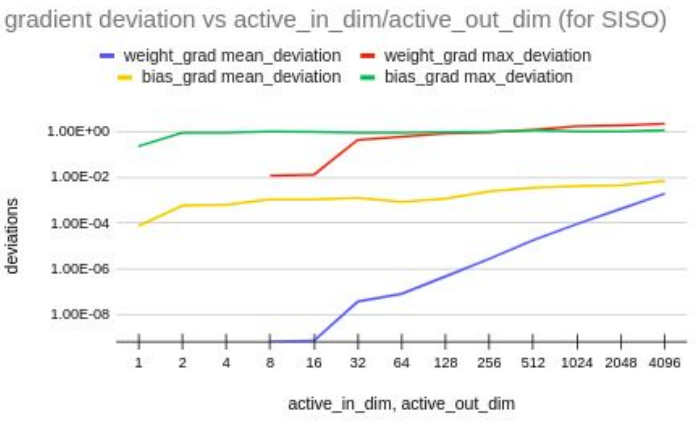
in_dim 4096
out_dim 4096
batch_size 128
num_threads 48

for DIDO

weight_grad mean_deviation	weight_grad max_deviation	bias_grad mean_deviation	bias_grad max_deviation
9.134E-07	1.526E-05	0.000E+00	0.000E+00







1. Sparse Multiplication - Collisions in Gradient Update

(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

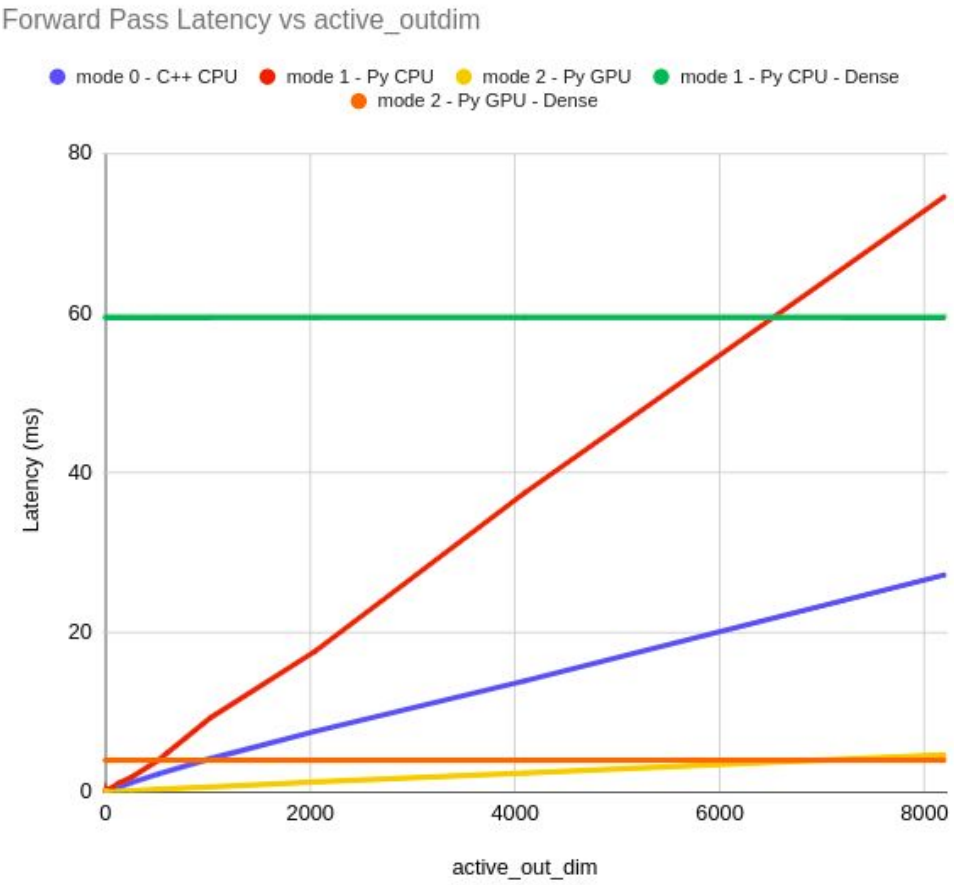
mean deviations are small enough to indicate the applicability of HogWild

Sparse Multiplication Latency

	Latency (ms)					
	forward pass			forward pass + backpropagation		
active_out_dim	mode 0 - C++ CPU	mode 1 - Py CPU	mode 2 - Py GPU	mode 0 - C++ CPU	mode 1 - Py CPU	mode 2 - Py GPU
dense / DIDO	354.228	59.480	4.012	4066.937	176.837	22.489
1	0.061	0.121	0.082	38.586	36.374	3.943
2	0.066	0.121	0.082	38.216	36.174	3.753
4	0.077	0.165	0.082	38.286	40.687	3.993
8	0.096	0.408	0.082	38.855	39.130	3.828
16	0.136	0.249	0.083	38.333	40.203	3.759
32	0.213	0.377	0.083	38.520	41.327	3.807
64	0.358	0.576	0.085	39.006	42.995	3.835
128	0.644	1.138	0.125	39.887	63.651	4.076
256	1.210	1.894	0.205	41.687	65.271	3.971
512	2.250	3.979	0.392	44.716	68.793	4.282
1024	4.206	9.317	0.671	50.929	83.418	5.002
2048	7.632	17.664	1.293	62.137	112.037	7.109
4096	13.948	37.514	2.391	81.188	184.144	10.989
8192	27.208	74.616	4.665	120.152	256.083	18.696

- DISO Latencies
- in_dim = 128
- out_dim = 524288
- batch_size = 128
- num_threads = 48
- GPU: NVIDIA GeForce GTX 108

Sparse Multiplication Latency



2. Hash Tables

- **Input:** Batched Input (potentially sparse)
- **Method:** For each input in batch
    - Compute L*K hash functions
    - Compute L meta-hash function from above hash values

For SimHash

- **Hash Function:** $h(x, r) = \text{sign}(\sum_i x_i r_i)$, where $x,r \in R^d$
- **Meta-hash Function:** $f(x)$ = K-bit integer b where $b_i = 1$ iff $h(x, r_i) < 0$

2. Hash Tables - SimHash

Comparing Hash-Function computation logic (for dense inputs)

SLIDE logic	proposed alternative logic
• Sparse Projection: chosen r_i has sparsity $0<\beta<1$. Complexity is $O(\beta d)$ and not $O(d)$. • Non-zero components of r_i can ± 1. Replaces multiply operations with addition.	• Direct matrix multiplication of input batch and hash matrix. • No need for sparsity • Components can take any values

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 II: Simhash Implementations Comparison. Results for K=9, L=50, d=128

(Pre-Process Phase)

2. Hash Tables Latency

(SimHash)

	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

3. Buckets Tables - PreProcess Phase

- **Input:** L bucket indices (meta-hash functions) for each neuron. (Range of bucket indices is $[0,2^K-1]$ for Simhash)
- **Method:** For each neuron, for each of the L tables, add the neuron's index to the appropriate bucket
- **Replacement Policy**
    - FIFO: mild bias for larger neuron indexes during overflow
    - Reservoir Sampling: Slightly slower than FIFO

TorchSLIDE	SLIDE
Parallelizes over L. Each thread iterates over all neurons. No Collisions	Parallelizes over num_neurons. Each thread iterates over L

3. Buckets Tables - Forward Pass

- **Input:** L bucket indices (meta-hash functions) for each input in batch
- **Output:** For each input, sample active neurons from its L chosen buckets.

Differences in Vanilla Sampling Implementation

TorchSLIDE	SLIDE
Samples a fixed no. of neurons for all inputs in batch	Only specifies a minimum no. of neurons to be sampled for each input in batch
Probes buckets only till sample size criterion is not met	Always probes all neurons in all L buckets
Uses C++ STL unordered set (hash map). No sort	Uses C++ STL map to sort sampled neurons
Complexity truly order O(sample_size) for each input	Complexity more than O(sample_size)

3. Buckets Tables Latency

- bucket_size = 128

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

4. Cross Entropy Loss

- For loss functions, our vector representations are such that we can directly use simple wrappers over PyTorch's built-in **softmax_cross_entropy_with_logits** and **binary_cross_entropy_with_logits** functions.

Cumulative Latency

- Total Pre-Process Latency for TorchSLIDE (K = 12, L = 50, num_neurons = 524288) ≈250ms
- Total Reported Pre-Process Latency for SLIDE (K = 9, L = 50, num_neurons = 205443) ≈18s
- Total Latency per Training Iteration of TorchSLIDE layer (sample_size = 4096) ≈88.617ms
- Total Latency per Training Iteration of Dense linear layer (CPU) ≈176.837ms

Evaluations

- Infrastructure
  - 48 threads of a Intel Xeon Platinum 8160 2.10GHz CPU
  - NVIDIA Geforce GTX 1080 Ti GPU
  - Ubuntu 18.04.5 LTS system installed with PyTorch 1.6
  - without HugePages or bfloat16 optimizations

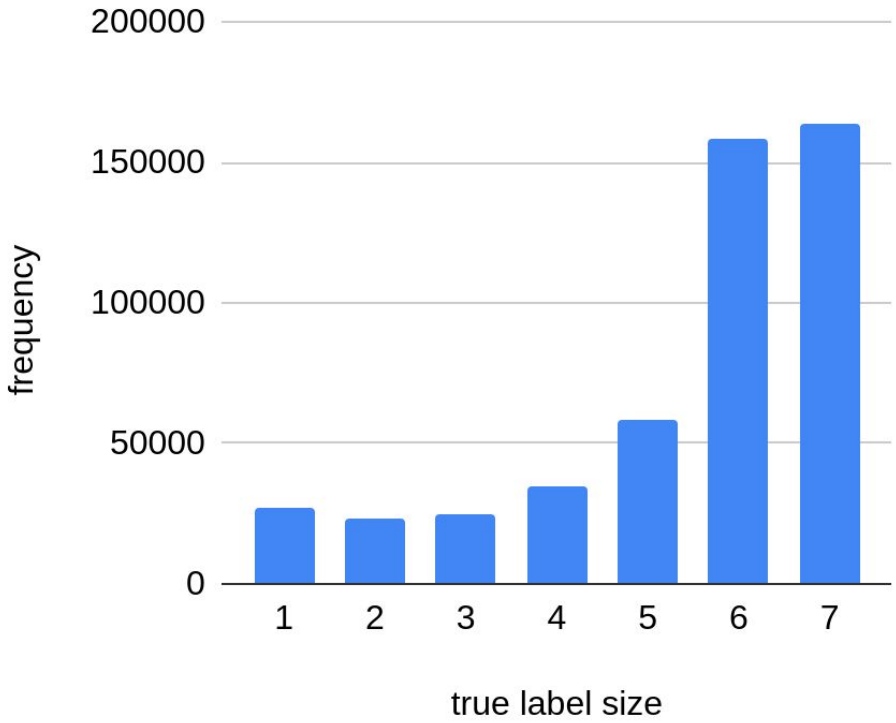
- Common Network Hyperparameters
  - 3 layer fully connected network with 128 dim hidden layer
  - ADAM optimizer. learning rate = 0.0001
  - LSH sampling only in output layer, with rehashing frequency = 50 iterations
  - For TorchSLIDE, num_label_samples fixed to match SLIDE.

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.

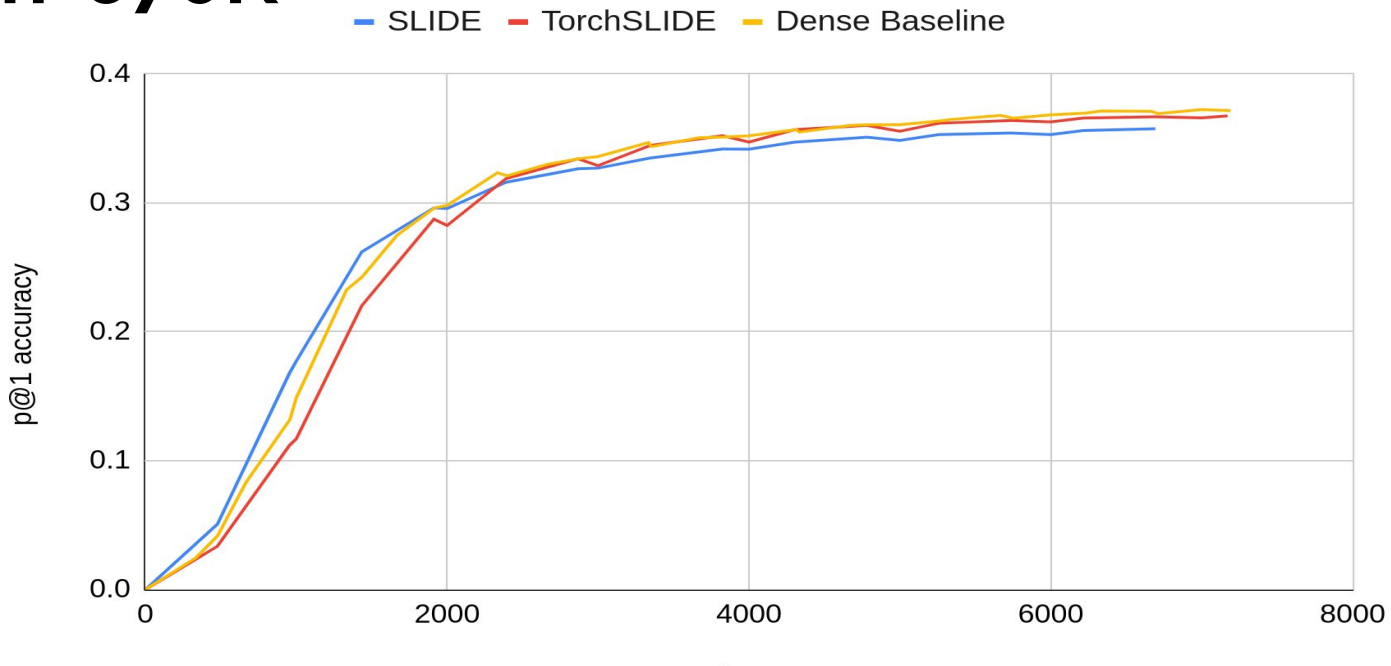
Amazon-670K

- Feature Dim = 135,909
- Label Dim = 670,091
- Train Dataset Size = 490,449
- Test Dataset Size = 153,025

- For all models
  - Batch Size = 128
- For all SLIDE/TorchSLIDE models
  - K=6, L=400
  - Winner-Takes-All (WTA) Hash
- For TorchSLIDE model
  - num_label_samples = 32,768



Amazon-670K



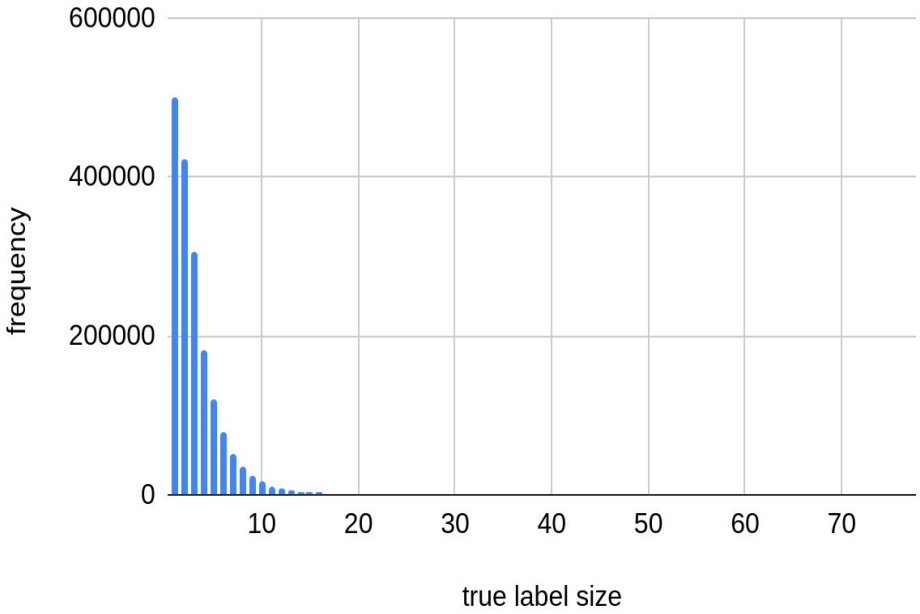
PyTorch-CPU	PyTorchGPU	SLIDE	New SLIDE	TorchSLIDE
550 s	71.8 s (trained using gradient accumulation)	1062.9 s	326.1s	397.6 s

latency per 100 iters

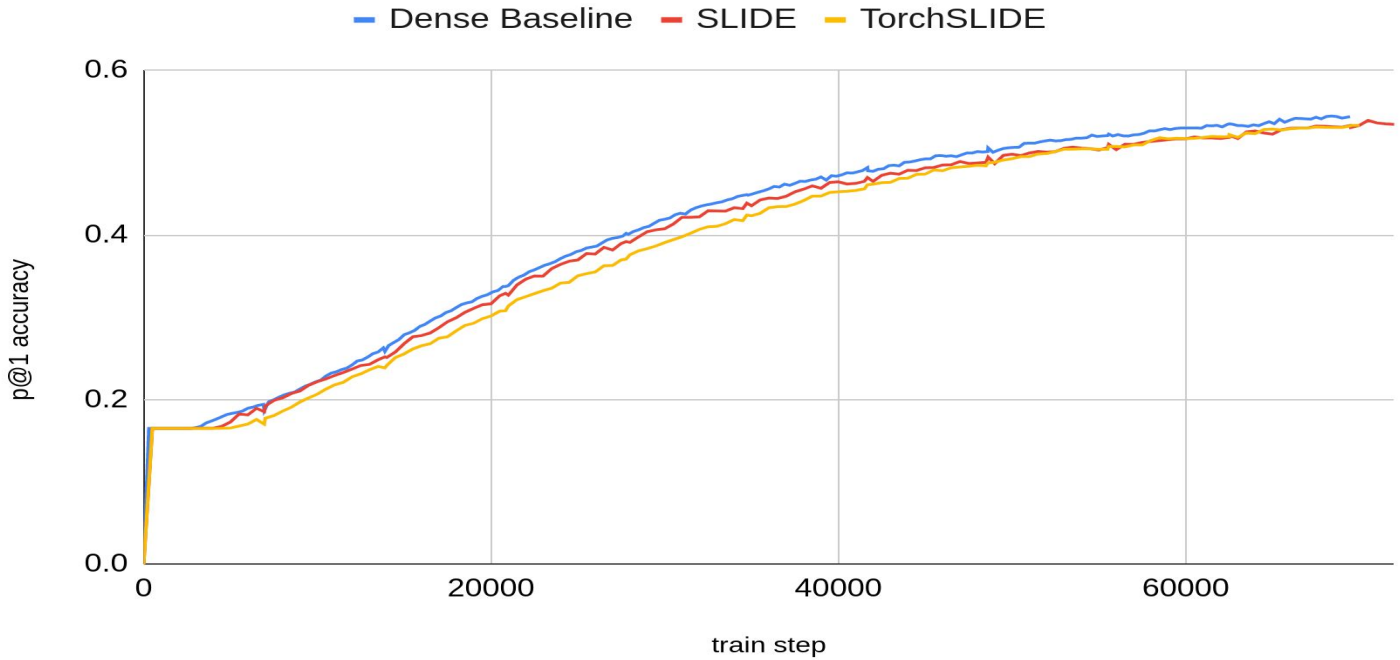
WikiLSHTC-325K

- Feature Dim = 1,617,899
- Label Dim = 325,056
- Train Dataset Size = 1,778,351
- Test Dataset Size = 587,084

- For all models
  - Batch Size = 256
- For all SLIDE/TorchSLIDE models
  - K=5, L=350
  - Winner-Takes-All (WTA) Hash
- For TorchSLIDE model
  - num_label_samples = 40,000



WikiLSHTC-325K



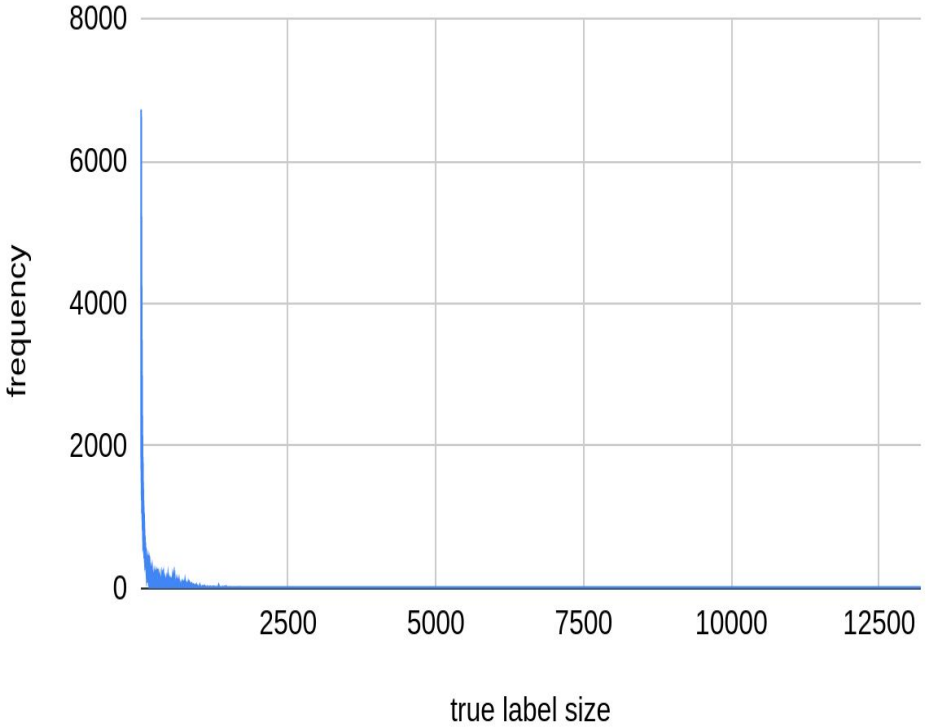
PyTorch-CPU	PyTorchGPU	SLIDE	New SLIDE	TorchSLIDE
167.9 s	35.7 s	229.4 s	67.5 s	156.9 s

latency per 100 iters

Delicious-200K

- Feature Dim = 782,585
- Label Dim = 205,443
- Train Dataset Size = 196,606
- Test Dataset Size = 100,095

- For all models
  - Batch Size = 128
- For all SLIDE/TorchSLIDE models
  - K=9, L=50
  - SimHash
- For TorchSLIDE model
  - num_label_samples = 2,048



Delicious-200K

Latency Profiling Experiments

batch size	128
input dim	131072
non zero input dim	128
hidden dim	128
output dim	524288
K	6

	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
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

Conclusion & Future Directions

1. End-to-End Evaluation of a Simple Network (Extreme Classification tasks) (till Jan 31)

- Perform experiments in Chen et al., Compare performance against
  - Tensorflow baselines - CPU, GPU
  - sampled softmax - CPU, GPU
  - SLIDE
- Compare against similar PyTorch baselines - CPU, GPU
- Using datasets from The Extreme Classification Repository with larger feature/label dimension (order millions)

2. Application to Attention-based Models

- Literature review about efficient transformers and transformer training/inference on CPU.
- Replacing output linear layer with TorchSLIDE when output dimension is large (e.g, vocabulary size)
- Modifying TorchSLIDE layers into self-attention layers for transformer models. Helps decrease order of computation from $O(n^2)$ to O(const*n). Ideal for long sequences.
- Similar modification for cross-attention layers to bring down per token computation from O(n) to sub-linear.
- Analyzing trade-off between sampling sparsity and accuracy
- Interpretation of chosen tokens by the hard attention mechanism

- Drop-in replacement for the linear layer module. Can conveniently be introduced in existing PyTorch models with little code change.
- Optimized using PyTorch's built-in functions wherever possible
- Avoids gradient collisions on a case-by-case basis.
- Matches convergence behaviour of regular dense networks.
- Speedup of up to 2.6x over SLIDE

- GPU based implementation of TorchSLIDE. Can use vectorization insights from new SLIDE.
- Application to attention based models like Transformers
  - Replacing output linear layer with TorchSLIDE when output dimension is large (e.g, vocabulary size)
  - Modifying TorchSLIDE layers into self-attention layers for transformer models. Helps decrease order of computation from $O(n^2)$ to O(const*n). Ideal for long sequences.
  - Similar modification for cross-attention layers to bring down per token computation from O(n) to sub-linear.
  - Interpretation of chosen tokens by the hard attention mechanism

References

- B. Chen, T. Medini, J. Farwell, S. Gobriel, C. Tai, and A. Shrivastava, “Slide : In defense of smart algorithms over hardware acceleration for large-scale deep learning systems,” 2020.
- B. Recht, C. Re, S. Wright, and F. Niu, “Hogwild!: A lock-free approach to parallelizing stochastic gradient descent,” Advances in neural information processing systems, vol. 24, pp. 693–701, 2011.
- Ba, Jimmy, and Brendan Frey. "Adaptive dropout for training deep neural networks." Advances in neural information processing systems 26 (2013): 3084-3092.
- Indyk, Piotr, and Rajeev Motwani. "Approximate nearest neighbors: towards removing the curse of dimensionality." Proceedings of the thirtieth annual ACM symposium on Theory of computing. 1998.
- Tay, Yi, et al. "Efficient transformers: A survey." arXiv preprint arXiv:2009.06732 (2020).
- N. Kitaev, Ł. Kaiser, and A. Levskaya, “Reformer: The efficient transformer,” arXiv preprint arXiv:2001.04451, 2020.
- K. Bhatia, K. Dahiya, H. Jain, A. Mittal, Y. Prabhu, and M. Varma, “The extreme classification repository: Multi-label datasets and code,” 2016. [Online]. Available: http://manikvarma.org/downloads/XC/XMLRepository.html

THANK

~YOU~