Efficient Sparse Training with Structured Dropout

Andy Lo
Department of Computer Science
University of Cambridge
`cyal4@cam.ac.uk`

Abstract

Dropout is a common technique for regularising neural networks to improve generalisation. Even though it introduces sparsity and thus potential for higher throughput, it usually cannot bring speed-ups on GPUs due to its unstructured nature. In this project, I experiment with SPARSEDROP, a structured variant of dropout, and realise its theoretical speed-up by implementing custom sparse CUDA kernels. Even at low sparsity levels, SPARSEDROP achieves speed-ups against its dense counterpart. I empirically demonstrate that SPARSEDROP provides similar, or sometimes even better, regularisation properties as standard dropout. This suggests its potential as a drop-in replacement to standard dropout with faster training speeds. The source code is available at `https://github.com/andylolu2/flash-dropout`.

1 Introduction

Designing machine learning algorithms that can be executed efficiently on modern hardware is core to the success of many prior efforts [15, 19, 22]. A popular area of research is to utilise sparsity to improve training and/or inference efficiency by skipping over unnecessary computation [14, 20, 21, 23]. While sparse computation has successfully brought substantial speed-ups on CPUs [20], its typically unstructured nature makes it difficult to execute efficiently on highly parallelised hardware such as GPUs. To tackle this issue, structured pruning is an emerging method for accelerating inference on GPUs. Unlike most prior works, this paper focuses on the use of structured sparsity for *GPU training*.

Dropout is a standard regularisation technique and has been shown to improve generalisation across many domains [32, 35]. The use of dropout is often motivated by performance, neglecting the fact that it introduces sparsity hence also interesting from a computational perspective. Unfortunately, standard dropout is unstructured which means its theoretical floating point operations (FLOPs) reductions cannot translate to faster execution speeds. To overcome this issue, this paper proposes SPARSEDROP, a simple, structured, and hardware-friendly variant of dropout that can benefit from sparsity on GPUs. Concretely, the previous and/or next operation of the dropout can now become a sparse operator, effectively fusing the dropout and the previous/next layer. In this paper, I will only focus on optimising the specific pattern of 'dropout followed by matrix multiplication', leaving other patterns as future work.

Achieving the theoretical benefits of SPARSEDROP is a challenging task as it requires low-level control over GPU computation and memory. Such fine-grained control over the hardware is not exposed by common deep learning libraries such as TensorFlow [1], PyTorch [29], or JAX [5]. Instead, I implemented several sparse matrix multiplication kernels in CUDA whose execution time decreases linearly with the sparsity level. Section 3 describes the important implementation details to achieve similar or better throughput compared to dense matrix multiplication kernels. In Section 4, I experimentally compare the effectiveness of SPARSEDROP and standard dropout in terms of training efficiency and evaluation performance.

2 Background

This section aims to provide sufficient background knowledge for understanding the design choices made in this project. Sections 2.1 and 2.2 introduce the basics of dropout and efficient matrix multiply and Section 2.3 discusses how this paper relates to several prior works.

2.1 Dropout

Dropout is a stochastic regularisation that helps to learn more robust features and has been shown to improve generalisation. The standard dropout applied to an activation matrix $\mathbf{X} \in \mathbb{R}^{M \times K}$,[1] randomly drops each value with hyper-parameter probability p. Since dropout is only used during training and becomes a no-op during inference, additional scaling is applied to un-dropped values to avoid train-inference mismatch. Concretely, its forward and backward pass is defined as:

$$\mathbf{m} \sim \text{Bernoulli}(1-p)^{M \times K} \qquad \bar{\mathbf{X}} = \frac{1}{1-p}\mathbf{X} \odot \mathbf{m} \qquad \frac{\partial L}{\partial \mathbf{X}} = \frac{1}{1-p}\frac{\partial L}{\partial \bar{\mathbf{X}}} \odot \mathbf{m}$$

where $\odot$ denotes element-wise multiplication.

The reasons why dropout is a good regularisation method are beyond the scope of this project, hence the theoretical background is omitted here. This project will instead take an empirical approach and test the effectiveness of dropout by performance.

2.2 Matrix multiplication on GPU

Matrix multiplication (also known as GEMM[2]) is one of the most foundational operations in deep learning, commonly referred to as a 'Dense' layer in neural networks. Consequently, many years of hardware and software engineering efforts have been put into making GEMM as efficient as possible. Here, we cover the fundamentals of implementing efficient dense GEMM which will help later as we implement performant sparse GEMM.

Efficient GEMM is designed to maximise the use of hardware features. GPUs thrive at parallel computation hence it is necessary to partition the GEMM problem into smaller, independent sub-problems that can be solved concurrently. The highest level of parallelism is across *threadblocks*, representing a group of threads that execute on the same *streaming multiprocessor*. Consider the problem $\mathbf{C} = \mathbf{AB}$ where $\mathbf{A} \in \mathbb{R}^{M \times K}$, $\mathbf{B} \in \mathbb{R}^{K \times N}$, and $\mathbf{C} \in \mathbb{R}^{M \times N}$. We say this is a problem of size (M, N, K). This problem is partitioned in the M and N dimension by block sizes M_{blk} and N_{blk} respectively[3]:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 \\ \vdots \\ \mathbf{A}_{M/M_{blk}} \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_0 & \dots & \mathbf{B}_{N/N_{blk}} \end{bmatrix} \quad \mathbf{C} = \begin{bmatrix} \mathbf{A}_0\mathbf{B}_0 & \dots & \mathbf{A}_0\mathbf{B}_{N/N_{blk}} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{M/M_{blk}}\mathbf{B}_0 & \dots & \mathbf{A}_{M/M_{blk}}\mathbf{B}_{N/N_{blk}} \end{bmatrix}$$

where $\mathbf{A}_i \in \mathbb{R}^{M_{blk} \times K}$ and $\mathbf{B}_j \in \mathbb{R}^{K \times N_{blk}}$. This gives $(M/M_{blk}) \cdot (N/N_{blk})$ independent sub-problems of size (M_{blk}, N_{blk}, K), each of which is computed by one threadblock. For a large K, it is not possible to load $\mathbf{A}_i$ or $\mathbf{B}_j$ into lower-level memory entirely. A typical implementation will thus also split the K dimension into blocks of size K_{blk} and accumulate over the partial sums:

$$\mathbf{A}_i = \begin{bmatrix} \mathbf{A}_{i,0}, \dots, \mathbf{A}_{i,K/K_{blk}} \end{bmatrix} \qquad \mathbf{B}_j = \begin{bmatrix} \mathbf{B}_{0,j} \\ \vdots \\ \mathbf{B}_{K/K_{blk},j} \end{bmatrix} \qquad \mathbf{A}_i\mathbf{B}_j = \sum_{k=0}^{K/K_{blk}} \mathbf{A}_{i,k}\mathbf{B}_{k,j}$$

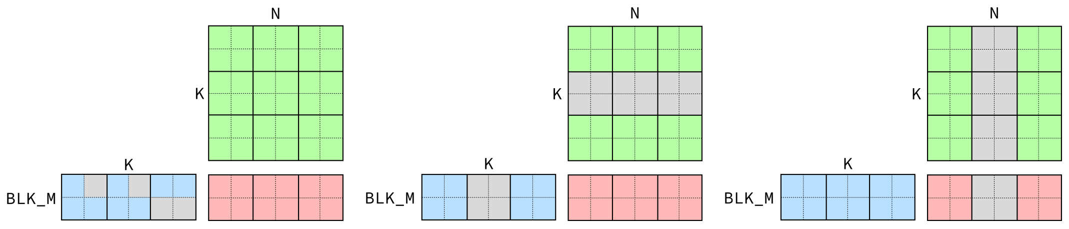
where $\mathbf{A}_{i,k} \in \mathbb{R}^{M_{blk} \times K_{blk}}$ and $\mathbf{B}_{k,j} \in \mathbb{R}^{K_{blk} \times N_{blk}}$.

To implement an efficient kernel, we must first understand the typical bottlenecks of GPU computation. One important property of is that global memory (VRAM) is comparatively slow, meaning that most operations, including GEMM, are *global memory bound*. To alleviate the problem, $\mathbf{A}_{i,k}$ and $\mathbf{B}_{k,j}$

[1] We assume 2D activations as typically we have one batch dimension and one hidden dimension.

[2] GEMM is the acronym for GEneral Matrix-to-matrix Multiply

[3] We assume the block sizes divide the problem sizes in this paper to simplify the implementation.



(a) Unstructured `dsd_matmul`. Since data is accessed in blocks, all blocks still needs to be loaded.

(b) Structured `dsd_matmul`. The second block of the first input are zeros, thus the entire second row-block of the second input can be ignored.

(c) Structured `sdd_matmul`. The second block of the output are zeros, thus the entire second column-block of the second input can be ignored.

Figure 1: Standard versus block-wise sparse GEMM for one M-block with $M_{blk} = N_{blk} = K_{blk} = 2$. The blue matrix represents a fragment of input $\mathbf{A}$, the green matrix is input $\mathbf{B}$ and the red matrix is the output $\mathbf{C}$. Solid lines represent the block-wise access granularity by the GPU. Ignored elements are coloured in gray. The sparsity level is 33% in all cases but only the structured variants can benefit from it by skipping over entire blocks.

are first loaded into the *shared memory* for each threadblock. Subsequent reads of the data are then served by the shared memory directly, improving data reuse and thus reducing the load on the global memory. Nonetheless, global memory remains the main bottleneck to faster GPU algorithms. More performant algorithms are only possible if they reduce global memory accesses [8, 13].

I would like to emphasise that efficient GEMM is a complicated algorithm and even achieving on-par performance with standard libraries such as PyTorch (which uses CuBLAS) is a challenging task. To reduce the complexity of the implementation, libraries like CUTLASS [33] exist to make high-performance GEMM more customisable. The high ceiling of CUTLASS comes at the cost of a steep learning curve which adds to the engineering difficulty of this project.

2.3 Related work

Structured dropout have been explored in prior works [6, 7, 12, 38]. However, all of them are motivated by performance (e.g., accuracy). For example, DropBlock [12] identifies that standard dropout performs poorly when used with convolutional neural networks [22] and demonstrates that better generalisation can be achieved if consecutive pixels (patches) of an image are dropped together. This makes it an exciting line of research since structured dropout can be favourable from both a performance and, as discussed later, computational perspective.

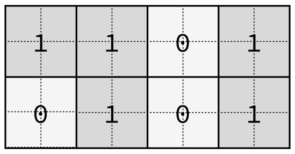
There is also a long lineage of work on training sparse neural networks [21]. However, most work focuses on faster or lower-memory inference rather than speeding up training. In fact, it is common to use *more* training resources for sparse networks. For example, all of [14, 20, 23, 24, 26, 31] first fully train a dense network and then apply pruning with re-training to obtain a performant sparse model. [4] is the most similar to the idea presented in this project, suggesting to use block-sparse patterns to accelerate GPU training. However, the implementation[4] is rather out-of-date and the authors were only able to achieve speedups at high sparsity levels ($\geq 75\%$).

3 Method

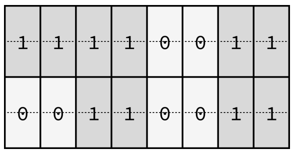
3.1 Fusing dropout with GEMM

To benefit from sparsity introduced by dropout, we must replace dense operators before/after the dropout layer with their sparse counterparts. As discussed earlier, we will limit our scope to optimising the ‘dropout + linear layer‘ pattern. To further simplify the implementation, we will only consider

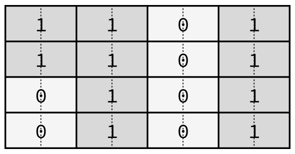
[4]`https://github.com/bengioe/condnet/`



(a) Logical mask



(b) Retiled mask ($p = 2, q = 1$)



(c) Retiled mask ($p = 1, q = 2$)

Figure 2: Example of block splitting. The logical block size is 2×2 but mask (b) operates on 2×1 while (c) operates on 1×2. The semantics is the same across all three masks.

linear layers without the bias term.[5] The forward and backward formulae are:

$$\mathbf{m} \sim \text{Bernoulli}(1-p)^{M \times K}$$

$$\mathbf{Y} = \frac{1}{1-p}(\mathbf{X} \odot \mathbf{m})\mathbf{W} \tag{1}$$

$$\frac{\partial L}{\partial \mathbf{X}} = \frac{1}{1-p}\left(\frac{\partial L}{\partial \mathbf{Y}}\mathbf{W}^\top\right) \odot \mathbf{m} \tag{2}$$

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{1}{1-p}(\mathbf{X} \odot \mathbf{m})^\top \frac{\partial L}{\partial \mathbf{Y}} \tag{3}$$

where $\mathbf{X} \in \mathbb{R}^{M \times K}$ and $\mathbf{W} \in \mathbb{R}^{K \times N}$. We see that Eq. (1) and Eq. (3) is a GEMM (with scaling) that produces a $\underline{\text{d}}$ense output from one $\underline{\text{s}}$parse input and one $\underline{\text{d}}$ense input, commonly abbreviated as a `dsd_matmul`. Similarly, Eq. (2) produces a $\underline{\text{s}}$parse output from two $\underline{\text{d}}$ense inputs, abbreviated as a `sdd_matmul`. As shown in Fig. 1a, the unstructured sparsity pattern defined by $\mathbf{m}$ makes it difficult to accelerate in hardware in the general case.

3.2 SparseDrop

To make the sparse GEMM fast in hardware, I propose to *alter the semantics to match the hardware requirements*. In particular, SparseDrop uses a block-sparse masking matrix $\mathbf{m'}$ in place of the per-element $\mathbf{m}$ while keeping Eqs. (1) to (3) the same. The important implementation detail is to *choose the same block sizes (i.e. M_{blk} and K_{blk}) for $\mathbf{m'}$ and the matrix multiplication algorithm*. As a result, the block sizes affect both the semantics (the masking granularity) and the GEMM efficiency. Empirically, I found that it is important to tune the block sizes to ensure optimal throughput.

Fig. 1b depicts how the block-sparse mask allows us to accelerate the `dsd_matmul` kernel. For each threadblock, instead of iterating through all K/K_{blk} blocks from the inputs, we can now skip over entire blocks that have been logically masked by $\mathbf{m'}$. Crucially, such implementation allows us to *reduce the global memory bottleneck* since blocks that are not required do not need to be read from global memory at all, meeting the critical requirement for faster GPU kernels as discussed in Section 2.2.

Similarly, the `sdd_matmul` kernel can be accelerated by only computing the output blocks that have not been masked (assuming the output matrix has been initialised to zeros). A simple example is shown in Fig. 1c.

3.3 Block splitting

One limitation of choosing the same block sizes for $\mathbf{m'}$ and GEMM is that it ties the choice of block sizes for the forward and backward pass. We denote the GEMM algorithm for a (M, N, K) problem with block sizes of $(M_{blk}, N_{blk}, K_{blk})$ as $\text{GEMM}(M, N, K, M_{blk}, N_{blk}, K_{blk})$. If the forward pass Eq. (1) is implemented as $\text{GEMM}(M, N, K, M_{blk}, N_{blk}, K_{blk})$, Eq. (3) then necessarily needs to be a $\text{GEMM}(K, N, M, K_{blk}, N'_{blk}, M_{blk})$. This is problematic since the optimal choices of M_{blk}, N_{blk}, K_{blk} for the forward pass might be sub-optimal for the backward pass. I observe that this can sometimes lead to 2-10$\times$ worse throughput.

[5]There is some evidence suggesting that the bias term is not always necessary for good performance. [17]

To alleviate this problem, I propose to perform *block splitting*. Given a logical block-wise mask with block sizes (M_{blk}, K_{blk}), we can retile the mask with a smaller block size $(M_{blk}/p, K_{blk}/q)$ and repeat each entry p times horizontally and q times vertically to obtain a logically equivalent mask, assuming p and q divides M_{blk} and K_{blk} respectively. An illustration is shown in Fig. 2. This now allows us to implement Eq. (1) as $\text{GEMM}(M, N, K, M_{blk}, N_{blk}, K_{blk}/2)$ (by choosing $p = 1, q = 2$) and Eq. (3) as $\text{GEMM}(K, N, M, K_{blk}, N'_{blk}, M_{blk}/2)$ (by choosing $p = 2, q = 1$) which has proven to provide sufficient flexibility to recover most of the performance.

3.4 Implementation details

To ensure optimal GEMM performance, most of the typical dense GEMM optimisations need to be implemented. These include the use of tensor cores [2], vectorised memory accesses [25], threadblock swizzling [3] and shared memory layout swizzling [27]. I empirically observed that each of such optimisations provided at least 20% improvement in throughput. To simplify the implementation, I specialise the kernel for NVIDIA Turing architecture (specifically RTX 2060 Max-Q), ignoring any cross-platform or cross-generation compatibility concerns. With the help of CUTLASS, the `dsd_matmul` and `sdd_matmul` kernels can be implemented in ≈ 650 lines of CUDA code.

I also found that a naive PyTorch (Python) implementation of sampling $\mathbf{m}'$ incurs significant overhead. Using the PyTorch profiler, I examined the stack trace for problems of various sizes and discovered that for small-to-medium sized problems (e.g., $M, N, K \leq 1024$), most of the time is spent on generating the mask $\mathbf{m}'$ rather than GEMM. To close the performance gap, I re-implemented the mask generation in C++ with a Python binding. This custom implementation also allowed me to efficiently pack the mask bits as 64-bit integers which further reduced the overhead of the sparse kernel[6].

3.5 Performance benchmarking

To test the performance of the CUDA sparse GEMM implementation, we compare it against several baseline implementations. They are dense GEMM (**Dense**), standard dropout followed by dense GEMM (**Dropout + Dense**), and block-wise dropout followed by dense GEMM (**Block dropout + Dense**, implemented in PyTorch). We denote the CUDA implementation described in the previous section as SPARSEDROP. Note that **Block dropout + Dense** generates the mask with PyTorch while SPARSEDROP uses the custom C++ implementation. [7]

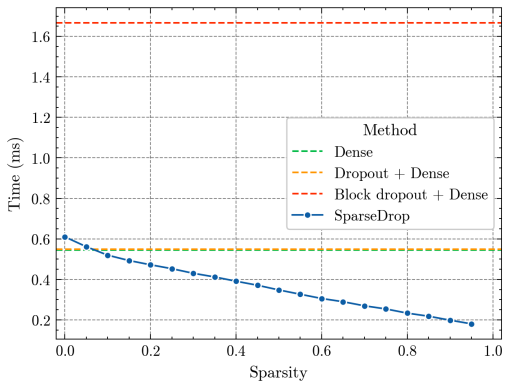
Fig. 3 demonstrates that SPARSEDROP can achieve speed-up over the baseline methods even at low sparsity levels. Fig. 3a shows that **Block dropout + Dense** is significantly slower than the rest of the methods, highlighting the bottleneck from naively generating and applying the mask using PyTorch. At sparsity levels $> 5\%$, SPARSEDROP already executes faster than **Dense** and **Dropout + Dense** and the execution time decreases linearly with the sparsity level. This is a significant improvement against other sparsity schemes as many only aim to achieve speed-ups with high sparsity [37]. Fig. 3b show that SPARSEDROP can maintain a relatively high throughput across many sparsity levels. Counter-intuitively, the FLOPS *increases* slightly with small amounts of sparsity ($\leq 30\%$) as seen in Fig. 3b. I fail to find a simple explanation for this phenomenon and leave further investigation as future work.

4 Experiments

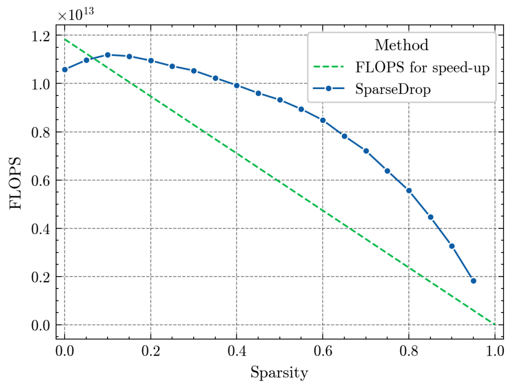
This section aims to test the empirical effectiveness of SPARSEDROP. Specifically, we want to understand whether SPARSEDROP can aid generalisation similar to standard dropout and evaluate its practicality in real-world training. To do so, I apply SPARSEDROP to Multi-Layer Perceptron (MLP) and Transformer [35] and evaluate its generalisation performance across both computer vision and natural language tasks in Section 4.1. Across all tasks, the block size of SPARSEDROP is fixed to $M_{blk} = 128$, $K_{blk} = 128$ as it gives good throughput at input size ≈ 1024.

[6]PyTorch currently does not have a bit-packing feature, so a naive implementation would require one global memory read per inner iteration.

[7]At the time of writing, PyTorch has an experimental feature for block-sparse GEMM. However, the backward pass for block-sparse GEMM has not been implemented yet thus it has not been considered as a baseline.



(a) Total time (forward + backward) for $M = N = K = 1024$ at various sparsity levels.



(b) FLOPS achieved by SPARSEDROP for $M = N = K = 1024$ at various sparsity levels. The dotted line represents the minimum FLOPS required to achieve speed-up over **Dense**.

Figure 3: Benchmark of SPARSEDROP against baseline methods, measured on RTX 2060 Max-Q with GPU clock locked at 1200MHz and clearing the L2 cache between each measurement.

The empirical results shown in Section 4.2 suggest that SPARSEDROP is competitive against standard dropout both semantically and computationally. In some tasks, it even leads to better generalisation while also training faster.

4.1 Tasks and methodology

The following sections give a brief overview of each task and the experiment setup. The full description for each experiment can be found in Appendix A.

4.1.1 MLP

An MLP model is applied to solve image classification tasks. To test the effects of dropout, each linear layer in the standard MLP is replaced with either **Dense** (equivalent to removing the bias term), **Dropout + Dense** or SPARSEDROP. The model has one input layer, two hidden layers and one output layer, uses ReLU activations, and has hidden dimension 1024.

As dropout is typically the most effective when a model is prone to overfitting, we only focus on a simple dataset, namely MNIST [9]. To exaggerate the effects of overfitting, only 16,384 images are used from the training set. The raw images are resized to 32×32 to simplify the implementation.

I perform a hyper-parameter search for $p \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7\}$ to find the dropout rate for the best generalisation. Each model is trained until the validation accuracy does not increase for 5 consecutive validation checkpoints or up to 100 epochs. Table 1 shows the best p found for each method according to the validation accuracy, averaged across 3 runs for each p.

4.1.2 Vision Transformer

This section aims to test the effectiveness of SPARSEDROP for image classification with a more capable, closer to state-of-the-art architecture: Vision Transformers (ViT) [10]. On a high level, ViT group patches of pixels into 'vision tokens' and apply a standard transformer model to process to token sequence. Similar to Section 4.1.1, each linear layer in ViT is substituted with either **Dense**, **Dropout + Dense** or SPARSEDROP. The model uses patch size 2×2 and has two layers with hidden dimension 1024.

The models are tested on Fashion MNIST [36] and CIFAR-10 [18] with the same pre-processing steps as Section 4.1.1[8]. The other experimental details are the same as Section 4.1.1 except that, due to limited compute, the experiments are not repeated.

[8]For CIFAR-10, the early-stopping patience is increased to 10 consecutive validation steps as I observe higher variability in the validation accuracy/loss.

Table 1: Best dropout rate for each method. The accuracy and loss are calculated from the best checkpoint throughout training. Note that the training time is the end-to-end time until early-stopping, thus measuring both the throughput and how quickly the models converge.

Model	Dataset	Method	Best p	Val accuracy	Val loss	Training time (minutes)
MLP	MNIST	Dense	-	96.96±0.62	0.131±0.046	**1.94±0.10**
		Dropout + Dense	0.5	**97.82±0.24**	**0.078±0.015**	2.78±0.51
		SPARSEDROP	0.3	97.61±0.20	0.090±0.022	3.18±0.13
ViT	Fashion MNIST	Dense	-	85.69	0.418	**20.15**
		Dropout + Dense	0.5	86.84	0.394	38.08
		SPARSEDROP	0.2	**87.04**	**0.367**	22.62
	CIFAR-10	Dense	-	48.71	1.466	**12.3**
		Dropout + Dense	0.4	53.05	1.428	24.73
		SPARSEDROP	0.4	**56.79**	**1.264**	45.05
GPT	Shakespeare	Dense	-	-	1.643	**6.48**
		Dropout + Dense	0.6	-	1.448	46.8
		SPARSEDROP	0.5	-	**1.430**	42.6

4.1.3 Language modelling

This section aims to test SPARSEDROP on a different domain: Natural language processing. I focus on language modelling using a GPT-style, decoder-only transformer model. The experiments use standard architecture choices following [30] and [17] with the exception of the linear layer substitutions. The model consists of 4 layers with hidden dimension 1024.

A character-level model is used to learn the Shakespeare dataset [16] with 1,115,394 characters. To reduce experiment time, only the first 524,288 tokens are used for training. Since this is not a classification task, the early stopping is performed according to the validation loss.

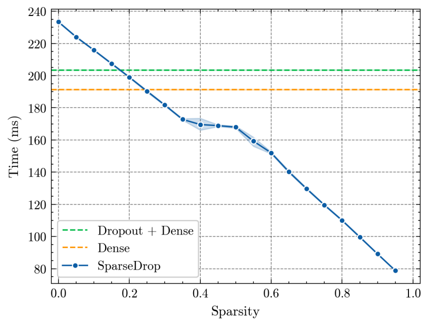
4.2 Results

The experiment results shown in Table 1 clearly demonstrate that SPARSEDROP has the same regularisation effects as standard dropout. Across all three tasks, SPARSEDROP provides a significant improvement over **Dense**, proving that it can indeed improve generalisation. When paired with the transformer architecture, SPARSEDROP gives *better* generalisation than **Dropout + Dense**. This is likely because transformers preserve locality information, hence, similar to how DropBlock [12] performs better with convolutional neural networks, SPARSEDROP can more effectively regularise transformers. The end-to-end training time of SPARSEDROP is also lower for the Fashion MNIST and Shakespeare tasks. In such cases, SPARSEDROP is both *better and faster than standard dropout*. I observe that the per-step convergence speed is similar between SPARSEDROP and **Dropout + Dense**, thus most of the speed-up comes from the faster step time of SPARSEDROP.

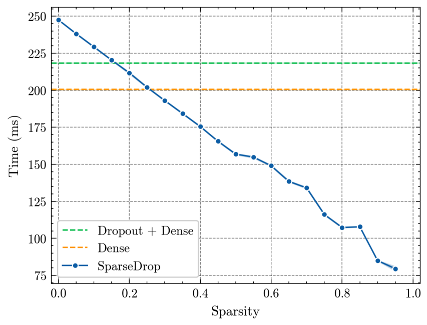
At the same dropout rate p, SPARSEDROP is a stronger regulariser than standard dropout. This is reflected in that the best p found for SPARSEDROP is lower than that for **Dropout + Dense** across all three tasks. Additionally, I observe that the training loss is higher for SPARSEDROP than standard dropout for the same p. This matches our expectation as two consecutive entries of data are more likely to be correlated than two random entries, particularly when dealing with sequential/spatial inputs. As a result, dropping out consecutive blocks of data should destroy more information than randomly chosen items, leading to a stronger regularisation effect. From a computational perspective, this is a negative result as the preference for a smaller p means there is less room to exploit sparsity.

4.3 Real-world throughput

Fig. 4 shows the practical run times of full models during training. Similar to that observed in Fig. 3a, the latency decreases linearly as the sparsity increases. With sparsity $\geq 20\%$, SPARSEDROP already outperforms **Dropout + Dense**. Despite not fully sparsifying the models, the overall speed can still more than double that of **Dense**, highlighting the potential of sparse operators for speeding up training.



(a) ViT on Fashion MNIST (batch size 64).



(b) GPT on Shakespeare (sequence length 128, batch size 32).

Figure 4: Total time (forward + backward) of models at various sparisty levels. Measured on RTX 2060 Max-Q (GPU clock locked at 1200MHz) with automatic mixed precision.

5 Limitations and future directions

5.1 Performance tuning

While SparseDrop appears to be computationally efficient in an isolated benchmark (Fig. 3), I observe inconsistent timing behaviour when used in MLP. It is unclear what the source of the issue is though the expected linear-scaling trend is recovered as the model size increases. Further engineering efforts are required to identify the cause of the issue.

There is still room for improvement to further optimise the GEMM performance. For example, all of the experiments use a block size of 128×128, even across different p values. I hypothesise that as the sparsity increases, smaller block sizes might be more optimal as it is effectively solving a smaller problem. Sections 3.5 and 4.3 also reveal that there is still some performance gap between SparseDrop and **Dense** when the sparsity is zero. I hypothesise that with more engineering efforts (e.g., implementing software pipelining [28]) the gap can be completely closed.

5.2 Hardware and software lock-in

The implementation of SparseDrop is heavily reliant on CUDA and is only developed for Turing architecture GPUs. This makes it difficult to deploy SparseDrop on other accelerators (e.g., TPUs), or to achieve the same performance guarantees on a different generation of NVIDIA GPUs. Additionally, CUDA is a low-level programming language that takes considerable engineering effort to use. The implementation in this project is thus not easily extensible to different hardware.

A higher-level kernel programming language like OpenAI Triton [34] might be preferred as it provides better cross-platform support. In fact, SparseDrop is initially implemented in Triton. Unfortunately, I was unable to optimise the Triton implementation to match dense GEMM performance, possibly because Triton mainly targets Ampere or newer architectures. It is also not possible to inject custom C++ logic which is important for removing the mask generation bottleneck. Future work might want to revisit the Triton implementation as the Triton language matures.

5.3 Future directions

While dropout is mostly only used during training, it is occasionally also utilised for inference. A common application is Bayesian Dropout [11], where dropout serves as a source of stochasticity for Bayesian inference. SparseDrop might be suitable for accelerating both the training and inference of such models.

This project only focused on accelerating the 'dropout followed by dense' pattern. While this has been proven to be successful in the experiments, it is more common to employ structured dropout for convolutional neural networks. There is potentially more gains to optimise for the 'dropout followed by convolution' pattern, which in principle can be implemented in the same way as SparseDrop with the implicit GEMM algorithm [39].

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[2] Jeremy Appleyard and Scott Yokim. Programming tensor cores in cuda 9, Oct 2017. URL `https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/`.

[3] Louis Bavoil. Optimizing compute shaders for l2 locality using thread-group id swizzling, Jul 2020. URL `https://developer.nvidia.com/blog/optimizing-compute-shaders-for-l2-locality-using-thread-group-id-swizzling/`.

[4] Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. Conditional computation in neural networks for faster models. *arXiv preprint arXiv:1511.06297*, 2015.

[5] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

[6] Liyan Chen, Philip Gautier, and Sergul Aydore. Dropcluster: A structured dropout for convolutional networks. *arXiv preprint arXiv:2002.02997*, 2020.

[7] Zuozhuo Dai, Mingqiang Chen, Xiaodong Gu, Siyu Zhu, and Ping Tan. Batch dropblock network for person re-identification and beyond. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 3691–3701, 2019.

[8] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[9] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[10] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[11] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059. PMLR, 2016.

[12] Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V Le. Dropblock: A regularization method for convolutional networks. *Advances in neural information processing systems*, 31, 2018.

[13] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

[14] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

[15] Sara Hooker. The hardware lottery. *Communications of the ACM*, 64(12):58–65, 2021.

[16] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks, 2015. URL `https://karpathy.github.io/2015/05/21/rnn-effectiveness/`.

[17] Andrej Karpathy. nanogpt. `https://github.com/karpathy/nanoGPT`, 2023.

[18] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research). URL `http://www.cs.toronto.edu/~kriz/cifar.html`.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[20] Eldar Kurtic, Denis Kuznedelev, Elias Frantar, Michael Goin, and Dan Alistarh. Sparse finetuning for inference acceleration of large language models. *arXiv preprint arXiv:2310.06927*, 2023.

[21] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.

[22] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[23] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 806–814, 2015.

[24] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE international conference on computer vision*, pages 2736–2744, 2017.

[25] Justin Luitjens. Cuda pro tip: Increase performance with vectorized memory access, Dec 2013. URL `https://developer.nvidia.com/blog/cuda-pro-tip-increase-performance-with-vectorized-memory-access/`.

[26] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378*, 2021.

[27] NVIDIA. Parallel thread execution isa version 8.3, Nov 2023. URL `https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#swizzling-modes`.

[28] NVIDIA. Efficient gemm in cuda, 2023. URL `https://github.com/NVIDIA/cutlass/blob/main/media/docs/efficient_gemm.md#pipelining`.

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

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

[31] Suraj Srinivas, Akshayvarun Subramanya, and R Venkatesh Babu. Training sparse neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 138–145, 2017.

[32] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[33] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. Cutlass, 1 2023. URL `https://github.com/NVIDIA/cutlass/tree/v3.0.0`.

[34] Philippe Tillet, Hsiang-Tsung Kung, and David 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*, pages 10–19, 2019.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[36] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[37] Takuma Yamaguchi and Federico Busato. Accelerating matrix multiplication with block sparse format and nvidia tensor cores, March 2021. URL `https://developer.nvidia.com/blog/accelerating-matrix-multiplication-with-block-sparse-format-and-nvidia-tensor-cores/`.

[38] Yiren Zhao, Oluwatomisin Dada, Xitong Gao, and Robert D Mullins. Revisiting structured dropout. *arXiv preprint arXiv:2210.02570*, 2022.

[39] Yangjie Zhou, Mengtian Yang, Cong Guo, Jingwen Leng, Yun Liang, Quan Chen, Minyi Guo, and Yuhao Zhu. Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 214–225. IEEE, 2021.

A Experiment details

A.1 MLP MNIST

The full training configuration is:

```
{
    "data": {
        "name": "mnist",
        "val_size": 4096,
        "train_size": 16384,
        "val_batch_size": 1024,
        "train_batch_size": 1024
    },
    "seed": 1,
    "model": {
        "dropout": {
            "p": 0.5,
            "variant": "blockwise[cuda]",
            "block_size": [
                128,
                128
            ]
        },
        "hidden_dim": 1024,
        "num_layers": 2,
        "output_dim": 10
    },
    "train": {
        "log_every": 10,
        "early_stop": {
            "mode": "max",
            "monitor": "Valiation accuracy",
            "patience": 5
        },
        "eval_every": 50,
        "max_epochs": 100
    },
    "wandb": {
        "mode": "online",
        "notes": null,
        "project": "flash-dropout-mlp"
    },
    "fabric": {
        "precision": "16-mixed",
        "accelerator": "auto"
    },
    "optimizer": {
        "lr": 0.001
    }
}
```

A.2 ViT Fashion MNIST

The full training configuration is:

```
{
    "data": {
        "name": "fashion_mnist",
        "val_size": 4096,
        "train_size": 16364,
        "val_batch_size": 64,
        "train_batch_size": 64
    },
    "seed": 0,
    "model": {
        "n_head": 8,
        "dropout": {
            "p": 0.5,
            "variant": "blockwise[cuda]",
            "block_size": [
                128,
                128
            ]
        },
        "n_embed": 1024,
        "n_layers": 2,
        "block_size": [
            8,
            8
        ],
        "patch_size": [
            2,
            2
        ]
    },
    "train": {
        "log_every": 50,
        "early_stop": {
            "mode": "max",
            "monitor": "Valiation accuracy",
            "patience": 5
        },
        "eval_every": 200,
        "max_epochs": 50
    },
    "wandb": {
        "mode": "online",
        "notes": null,
        "project": "flash-dropout-vit"
    },
    "fabric": {
        "precision": "16-mixed",
        "accelerator": "auto"
    },
    "optimizer": {
        "lr": 0.0001
    }
}
```

A.3 ViT CIFAR-10

The full training configuration is:

```
{
    "data": {
        "name": "cifar10",
        "val_size": 4096,
        "train_size": 16364,
        "val_batch_size": 64,
        "train_batch_size": 64
    },
    "seed": 0,
    "model": {
        "n_head": 8,
        "dropout": {
            "p": 0.4,
            "variant": "blockwise[cuda]",
            "block_size": [
                128,
                128
            ]
        },
        "n_embed": 1024,
        "n_layers": 2,
        "block_size": [
            8,
            8
        ],
        "patch_size": [
            2,
            2
        ]
    },
    "train": {
        "log_every": 50,
        "early_stop": {
            "mode": "max",
            "monitor": "Valiation accuracy",
            "patience": 10
        },
        "eval_every": 200,
        "max_epochs": 50
    },
    "wandb": {
        "mode": "online",
        "notes": null,
        "project": "flash-dropout-vit"
    },
    "fabric": {
        "precision": "16-mixed",
        "accelerator": "auto"
    },
    "optimizer": {
        "lr": 0.0001
    }
}
```

A.4 LLM Shakespeare

The full training configuration is:

```
{
    "data": {
        "name": "shakespeare",
        "length": 128,
        "val_size": 1024,
        "cache_dir": "data/shakespeare",
        "train_size": 4096,
        "val_batch_size": 64,
        "train_batch_size": 32
    },
    "seed": 0,
    "model": {
        "n_head": 8,
        "dropout": {
            "p": 0.4,
            "variant": "blockwise[cuda]",
            "block_size": [
                128,
                128
            ]
        },
        "n_embed": 1024,
        "n_layer": 4,
        "context_length": 128
    },
    "train": {
        "log_every": 50,
        "early_stop": {
            "mode": "min",
            "monitor": "Valiation loss",
            "patience": 5
        },
        "eval_every": 200,
        "max_epochs": 100
    },
    "wandb": {
        "mode": "online",
        "notes": null,
        "project": "flash-dropout-llm"
    },
    "fabric": {
        "precision": "16-mixed",
        "accelerator": "auto"
    },
    "optimizer": {
        "lr": 0.0003,
        "weight_decay": 0.1
    }
}
```