SliceOut: Training Transformers and CNNs faster while using less memory

Pascal Notin\textsuperscript{1}  Aidan N. Gomez\textsuperscript{1,2}  Joanna Yoo\textsuperscript{2}  Yarin Gal\textsuperscript{1}

OATML Group\textsuperscript{1}  Department of Computer Science
University of Oxford

Cohere\textsuperscript{2}  Toronto, Canada

\{pascal.notin,aidan.gomez,yarin\}@cs.ox.ac.uk  joanna@cohere.ai

Abstract

We demonstrate 10–40\% speedups and memory reduction with Wide ResNets, EfficientNets, and Transformer models, with minimal to no loss in accuracy, using SliceOut—a new dropout scheme designed to take advantage of GPU memory layout. By dropping contiguous sets of units at random, our method preserves the regularization properties of dropout while allowing for more efficient low-level implementation, resulting in training speedups through (1) fast memory access and matrix multiplication of smaller tensors, and (2) memory savings by avoiding allocating memory to zero units in weight gradients and activations. Despite its simplicity, our method is highly effective. We demonstrate its efficacy at scale with Wide ResNets & EfficientNets on CIFAR10/100 and ImageNet, as well as Transformers on the LM1B dataset. These speedups and memory savings in training can lead to CO\textsubscript{2} emissions reduction of up to 40\% for training large models.

1 Introduction

Dropout\textsuperscript{[1,2]} is a regularisation technique widely used in large neural networks. However, while dropout does remove information from the representations of the neural network, standard implementations of dropout neither reduce the memory nor the computational complexity of training.

This seemingly missed opportunity is explained by the limitations imposed by contemporary accelerator hardware that is highly optimised for parallel computation of linear algebra primitives, while relatively slow at memory manipulation. This means that dropout’s unstructured nature – consisting of eliminating neurons (network units) uniformly at random (Fig.\textsuperscript{[1a]}) – fits poorly with the current computing paradigm. This problem is especially pronounced with several schemes for performing dropout in convolutional neural networks (CNNs), e.g., SpatialDropout\textsuperscript{[3]}, in which entire convolution filters are set to zero at random. The resulting feature tensors, containing large slices explicitly represented as zeros in memory, are then propagated “as is” to the rest of the network (Fig.\textsuperscript{[2a]}). This results in up to half of the memory of a CNN being assigned the value “zero”, unnecessarily storing data carrying no information, and leads to a large number of useless zero multiplications in both the forward and backward passes.

In this work we present an alternative to the standard formulation of dropout that simultaneously preserves the regularisation effects of the original dropout method, while speeding up computations and reducing the memory requirements of training neural networks. We give three variations of our method, specialised to deep feed-forward architectures, to convolutional neural networks, and to Transformer models.
Our proposed method, SliceOut (§3), introduces structure to dropout by slicing contiguous memory segments, i.e., selecting a contiguous range of neighboring neurons and slicing feature tensors or weight matrices row/column-wise (Fig. 1c), as opposed to selecting neurons uniformly at random. From the computational perspective, this strategy takes advantage of GPU memory layout as the operation requires a single access to contiguous memory. From the memory perspective, the zero units, that would physically remain in memory with standard dropout, are removed from memory overhead by the slicing operation. This implies a smaller memory footprint for weight gradients and activations throughout the network, and also results in matrix multiplications with smaller tensors compared to standard dropout. This in turn allows us to fit larger models in memory than would otherwise be possible, or conversely, to train a model of similar size with fewer computing resources. Lastly, SliceOut helps prevent some of the issues that standard dropout has when applied to CNNs (see §3.4 and Fig. 2).

Our experiments are carried in three settings (§4): the first is a toy setting consisting of relatively small neural networks applied to MNIST and FashionMNIST, demonstrating that training time does not suffer with our new method, but rather that our method converges faster than standard dropout; the second is Wide ResNets & EfficientNets applied to CIFAR-10 and CIFAR-100, demonstrating significant memory and speedup gains due to the large reduction in ops with the CNN’s high dimensional feature vectors; and the final setting is language modelling with Transformers applied to LM1B, demonstrating the applicability of our method beyond vision tasks.

In all our settings we find that SliceOut performs comparatively (or out-performs) standard dropout in terms of test accuracy, while achieving memory and compute savings of 10-40%, depending on the model architecture and dropout rate considered. This can in turn lead to CO$_2$ emissions reduction of up to 40% for training large models, as it would require fewer GPUs to be used for a shorter amount of time (see Appendix E).

Our contributions are as follows:

- We introduce a new structured scheme to perform dropout in neural networks that can achieve significant memory and computation gains
- We derive various sampling schemes for the method which preserve (exactly or approximately) the first and second moments of the layers’ output, allowing for efficient deterministic approximations at inference time
- We implement this new scheme across a diverse set of network architectures - from regular MLPs, to Residual networks and Transformers
- We quantify the relative speedups and memory gains between the different dropout schemes across experimental setups, demonstrating practical gains with SOTA models with minimal to no impact on accuracy

2 Background

Dropout randomly “turns off” neurons of a given layer (and, implicitly, the weights connected to them) during training. This is claimed to help prevent co-adaptation between neurons, and empirically results in improved generalisation in many model architectures and across many tasks [4]. Standard dropout may also be interpreted as sampling a “thinned” architecture from an exponential number of related networks ($2^d$ if the layer width is $d$) during training, and approximately ensembling these architectures at test time through first-moment propagation [5].

Since the seminal dropout paper [1], many alternative dropout schemes have been proposed to improve the efficiency of the technique across a wide range of different neural network architectures. We review the most relevant approaches aimed at memory gains and model speed-up related to our work:

**Standard dropout.** At each training step, the activations from neurons at a layer where dropout is applied are zeroed out with a probability $p$ – the dropout probability for that layer – with the forward and backward passes being then performed as usual (Fig. 1d). During testing, all units of the original architecture are kept to perform the forward pass. Because a fraction $p$ of units are dropped during training, activations need to be renormalised to preserve the expected value of pre-activations of subsequent layers between train and test, preserving the first and second moments of the layer’s output. This normalisation may be performed at test time (“weight scaling inference rule” [3]), or
during training (“inverted dropout”). The latter is the most popular approach used nowadays and consists of dividing each neuron at a layer where dropout is applied by the probability of it being kept (i.e., divided by \((1 - p)\)).

**Controlled dropout.** Controlled dropout \(\text{[7, 8]}\) was suggested to speed up the training of fully connected networks based on the observation that storing zeroed activations throughout the forward and backward pass leads to computational inefficiencies. The authors propose to keep a random subset of rows or columns of the activation tensors by performing a set of ‘gather’ operations (gather ops) on the corresponding network weights (Fig. 1). The gather ops select specific weight rows/columns, and allocate new memory into which these rows/columns are copied, so that subsequent multiplications in the forward and backward passes involve smaller tensors. Although this approach helps avoid unnecessary multiplications, the gather ops’ memory allocations introduce significant overhead. More specifically, the GPU needs to perform a quadratic number of reads and writes in order to create the required reduced tensors. This is not only slow to perform, but also results in duplicating the gathered weight tensors data in memory (Table 1).

**DropBlock & SpatialDropout.** Convolutional neural networks require a different scheme than standard dropout to perform effective regularisation \([3, 9]\). This is both due to the strong correlations between adjacent pixels present in natural images (and preserved in subsequent feature maps) and the fact convolution kernels operate on nearby pixels. Consequently, when a given pixel is zeroed out, information can still propagate through neighboring pixels as if no dropout had been applied. Several schemes have been proposed to circumvent this limitation, for example by zeroing out contiguous regions of the feature maps \([10]\) or zeroing out entire convolution filters \([3]\).

Further parallels may be drawn between SliceOut and Nested Dropout \([11]\), in which coherent nested sets of hidden units are dropped in order to learn ordered representations, and with DropEdge \([12]\), in which a certain number of edges are removed from the input graph at each training epoch.

### 3 SliceOut

SliceOut is a structured dropout scheme aimed at speeding up computations and reducing cached memory footprint, while preserving the regularisation benefits of standard dropout. We first convert the dropout rate into an expected number of nodes that should be kept at a layer where SliceOut is applied, i.e. the “slice width”. During training, we uniformly sample the starting index of the slice (restricting to a subset of eligible positions as explained below), then “slice” (see next paragraph) the relevant rows and columns of the weights and biases that precede / follow the layer(s) where SliceOut is applied (Fig. 1). We then perform the forward and backward passes with the sliced weights and biases, updating the corresponding slice(s) of the original weight matrices in-place. We repeat this end-to-end process, sampling different slices at each step, until convergence (see Algorithm 1). At test time, we use the full network without dropping any weights or biases, similar to standard dropout.
Table 1: Comparison of memory usage & No. of basic operations for different dropout schemes with \( b \) the batch size, \( n \) & \( m \) the No. of neurons in the input & output layers resp. & \( p \) the dropout probability applied to both input and output layers. SliceOut benefits from the same computation savings as Controlled dropout, without the memory reallocation overhead.

| Metric                          | Standard dropout | Controlled dropout | SliceOut        |
|---------------------------------|------------------|--------------------|-----------------|
| No. extra read/writes to manipulate weights | -                | \( O((1 - p)^2 * n * m) \) | \( O(1) \)    |
| Extra memory usage due to weight copy | -                | \((1 - p)^2 * n * m \) | -              |
| No. basic operations for weight multiply | \( O(b * n * m) \) | \( O((1 - p)^2 * n * m * b) \) | \( O((1 - p)^2 * n * m * b) \) |
| Size of output activations tensor | \( m * b \)     | \((1 - p) * m * b \) | \((1 - p) * m * b \) |

3.1 The slice op

Slicing is a fast and memory efficient operation: it selects the tensor elements of interest with a single memory access, and performs tensor operations with the logical tensors in-place [13, 14]. The slice operation (slice op) only changes the logical view into the memory, but not the physical memory. When a GPU matmul or conv kernel (both GPU functions) is called, it only sees the weights within that view, and does its operation with those weights without having to move anything in memory. SliceOut enjoys speed-ups from performing forward and backward passes with smaller tensors; Furthermore, as we now need only keep the smaller sliced activation tensors in memory to perform the backward pass at train time, we save on activation storage.

At test time we use the full network, and therefore there is no difference in memory usage to a network trained with standard dropout. However, the memory bottleneck for large networks is typically at train time since we are required to store intermediate activations to compute gradients on the backward pass.

3.2 Normalisation

After applying dropout, it is necessary to re-normalise activations in order to preserve the moments of their distributions and avoid the network outputs exploding or collapsing to zeros. We experimented with different approaches to normalise activations after dropout, and describe here the two that lead to the best results in experimental settings (Appendix B):

- **Flow normalisation**: We divide activations by the expected proportion of nodes kept at that layer during training (i.e., the ratio of the slice width to the full layer width). Intuitively, this helps keep constant the expected values of pre-activations at subsequent layers.

- **Probabilistic normalisation**: We divide each node by the probability of this specific node being kept during training. This helps ensure that, on average during training, the activations stemming from this particular node are equal to what they would be at test time.

These two normalisations coincide in the standard dropout case, where the expected proportion of nodes kept at a given layer is exactly equal to the probability of each node to be kept during training. This is not the case in SliceOut, as we impose constraints on eligible slices during sampling to avoid memory re-allocations and keep the size of tensors constant throughout training (see Appendix A.1): nodes around the edges are less likely to be selected at a given training step.

3.3 Regularisation and ensembling

While standard dropout samples a “thinned” network from an exponential number of possible architectures, SliceOut samples from a linear or quadratic number of architectures[1]. As a result SliceOut can be seen as a milder regularisation scheme (for a fixed dropout probability value). We

---

[1]: If SliceOut is applied at only one layer, we only take slices row-wise of the corresponding weight vector (and column-wise of the subsequent weight vector), thereby sampling from a linear number of architectures. If
observe in several experimental settings that, beyond a certain dropout probability threshold, the performance drops more sharply in standard dropout than in SliceOut. This increased stability makes SliceOut less sensitive to the chosen dropout probability, enabling higher drop rates.

**Algorithm 1: Slice dropout algorithm - Simple FFN with L hidden layers**

Let $W_l$, with $l \in [1 - L]$, be the weights tensor of the $l^{th}$ hidden layer;

Let $f_l(\cdot)$ be the non-linearity applied at the $l^{th}$ layer;

for training$_{-}$step ← 1 to $T$

Sample mini-batch $(x, y)$;

for layer$ _l$ ← 1 to $L$

Sample slice: Slice$ _l$ = $(start_l, end_l)$;

for layer$ _l$ ← 1 to $L$

Sample weights: $W_l$-slice = $W_l[[start_l, end_l], (start_{l-1}, end_{l-1})]$ – where $(start_0, end_0)$ selects the full input;

Perform forward pass with sliced weights: for layer$ _l$ ← 1 to $L$

$\quad x \leftarrow f_l(norm(W_l$-slice $\cdot x))$ – where norm($\cdot$) is the activation normalisation applied post dropout

Perform backward pass with sliced weights;

3.4 SliceOut and CNNs

Our SliceOut schemes for CNNs (Fig. 2) draw inspiration from the prior dropout schemes tailored to CNNs that we discussed in §2 ([3, 10]):

- **Channel-SliceOut**: slicing contiguous sets of channels for a given convolution kernel
- **Patch-SliceOut**: slicing contiguous 2D chunks of the input activation tensors, and then performing the convolution

Channel-SliceOut builds on the SpatialDropout scheme [3], with the critical difference that we directly slice the convolution kernels instead of zeroing out feature maps of the output activation tensor. This results in smaller output tensors and helps avoiding performing tensor operations for which the outcome will be ultimately be set to zero. Patch-SliceOut can be seen as performing the complement operation to what is done in Cutout [15] (on the input image), or more generally in DropBlock [10], where units in a contiguous region of a feature map are dropped together, except that we slice out zeros instead of carrying them around.

![Figure 2](image)

Figure 2: Comparison of the feature tensor of a convolution layer where different dropout schemes are applied

- **a) SpatialDropout**: randomly sets entire convolution channels to zero.
- **b) Channel SliceOut**: randomly selects a contiguous set of convolution channels, resulting in a more compact feature tensor (other channels are never allocated in memory).
- **c) Patch SliceOut**: selects a contiguous block of the input tensor across feature maps, then performs the convolution.

3.5 SliceOut and Transformers

Transformers [16] represent the state of the art across a host of natural language benchmarks and have seen adoption across academia and industry. One short-coming of the architecture is the considerable memory requirements demanded by the model architecture since Transformers tend to improve their memory requirements for two consecutive layers, we slice the second weight matrix row and column wise, thereby sampling from a quadratic number of architectures (see Appendix C).
Performance dramatically with the number of parameters they are given. This observation has lead
to several strategies to construct larger and better models (a quick overview of the Transformer
architecture is given in Fig. 3).

SliceOut represents a complementary technique to the standard model-scaling measures taken in
the literature (e.g., distributed data and model-parallelism, memory efficiency-focused optimisers [17])
and can be used in conjunction with them.

In our implementation of SliceOut in transformers we do not normalise the queries and keys as in
§3.2. Instead, we modify the temperature value (α) used in the attention weights:

\[ W_{\text{attn}} = \text{softmax} \left( \frac{QK^T}{\sqrt{\alpha}} \right) \]

In a Transformer, α is generally set to the dimensionality of the vectors in the queries and keys, but in
our case, SliceOut changes the dimensionality of those vectors during training, and so we adjust α to
be the new dimensionality of these vectors after SliceOut. We do still perform normalisation (§3.2) on
the values and within the feed-forward networks (Fig. 3). Note: In the figure, normalisation is denoted
“scale” while “norm” refers to layer normalisation, as in the original Transformer paper).

Since there is a dot product taken between each of the queries and keys, it is necessary that the sliced
out indices of those vectors are aligned. That is, SliceOut slices out some contiguous set of elements
from a query vector \( Q_{\text{sliced}} = (q_i, \ldots, q_{q+d}) \); it is, of course, extremely important that these indices
are the same for the sliced keys \( K_{\text{sliced}} = (k_i, \ldots, k_{q+d}) \). Similarly, when slicing weight matrices
we must ensure that the slices made along the leading dimension align with the slices applied to the
incoming activation vector. See the orange lines in Fig. 3 for a pictorial description of indices that
must be aligned.

4 Experimental results

We quantify the benefits of SliceOut across several neural network architectures and application
domains: fully connected networks on MNIST and FashionMNIST datasets (§4.1), Wide ResNets
and EfficientNets on the CIFAR-10 and CIFAR-100 datasets (§4.2 and 4.3), and Transformers on
the LM1B dataset (§4.4). For each experiment, we train the different networks until convergence,
measure speedups based on the train time per epoch, and memory gains via the maximum GPU
memory managed by the caching allocator at each epoch. All details about the training procedure
and hyperparameters used are provided in Appendix F, including comparisons between the different
normalisation schemes and details about their use in the following experiments.

4.1 Fully connected networks

This first set of experiments is performed in a toy setting aimed at studying the properties of our
method with fully connected networks on the MNIST [18] and FashionMNIST [19] datasets.
In the FashionMNIST experiments, we observe that not only do we obtain speedups (up to 15%) and cached GPU memory savings (up to 30%) with SliceOut, we also converge faster and to a higher test accuracy value (Fig. 4) when typical dropout rates are applied (i.e., \( p \leq 0.5 \)). The highest test accuracy obtained with SliceOut across all hyperparameter settings tested was 90.0 ± 0.03% (obtained with \( p = 0.1 \)), while the highest with standard dropout (also for \( p = 0.1 \)) was 89.6 ± 0.08% (no dropout lead to a test accuracy similar to the latter, see Appendix F.1.2).

In the MNIST experiments, we observe similar speedups and memory gains from SliceOut, although there was no statistically significant difference in terms of top test accuracy (Appendix F.1.3).

Across both experiments, controlled dropout was converging to similar test accuracy values, but was systematically slower and less memory efficient than SliceOut (Appendix F).

### 4.2 Wide ResNets

Wide ResNets [20] are a variant of the original ResNet architecture that achieve higher test accuracy while significantly reducing the depth of the network by substantially increasing the number of convolution filters in each residual block. The architecture strings together several “Wide-dropout” blocks, progressively increasing the number of channels and reducing the height & width of the activation tensors. Standard dropout is used critically in each residual block between the two 3x3 convolutions, to prevent potential overfitting resulting from the channel widening.

We remove the standard dropout layer in the original “Wide-dropout” block and experiment with our two SliceOut schemes for CNNs (see Fig. 2 and architecture diagrams in Appendix F.2.1):

- **Channel-SliceOut**: we apply SliceOut on the first 3x3 convolution across all residual blocks. It is critical to ensure that we operate on the same slice at the subsequent convolution layer, and the batch norm in-between;
- **Patch-SliceOut**: we apply Patch-SliceOut on the input tensor to the first 3x3 convolution, across all blocks.

For both schemes, normalisation is performed right after SliceOut is applied. In our CNN experiments, we observe higher final test accuracy when using the Probabilistic normalisation scheme (§3.2).

We test the performance of the SliceOut architectures on the CIFAR-10 & CIFAR-100 datasets (Appendix F.2.1). We obtain similar or higher test accuracy with both schemes compared the original Wide ResNets architecture (Table 2), with compute time and memory savings of 10-30% depending on the dropout probability applied. SliceOut also appears to be more stable in the medium-high dropout regime (0.4-0.5 dropout probability) compared to standard dropout, possibly due to ensembling over a lower number of different architectures.

### 4.3 EfficientNets

EfficientNets [21] achieve state of the art performance on several vision datasets incl. ImageNet [22], while being more compute efficient than prior architectures. Standard dropout is used in EfficientNets.
Table 2: CNNs results. Training time & Max cached GPU memory are resp. the relative % of train time per epoch for a network trained with SliceOut Vs standard dropout, and the maximum cached GPU memory during training. “WRN” refers to “Wide ResNets” and “EN” to “EfficientNets”. Results with EfficientNets were obtained by fine tuning a model trained on ImageNet (without SliceOut). Detailed results in Appendix F.2.

| Dataset  | Architecture | Dropout rate | Test accuracy | Test accuracy | Training time | Max cached memory |
|----------|--------------|--------------|---------------|---------------|---------------|-------------------|
|          |              |              | Standard dropout | SliceOut     |               |                   |
| CIFAR-10 | WRN 40x10    | 0.0          | 96.3%          | -             | -             | -                 |
|          |              | 0.1          | 96.2%          | -             | -             | -                 |
|          |              | 0.5          | 93.9%          | 96.1%         | -39.3%        | -22.9%            |
| CIFAR-10 | EN B2        | 0.0          | 98.4%          | -             | -             | -                 |
|          |              | 0.1          | -              | 98.3%         | -7.8%         | -9.0%             |
|          |              | 0.5          | -              | 95.2%         | -31.8%        | -40.8%            |
| CIFAR-100| WRN 40x10    | 0.0          | 81.5%          | -             | -             | -                 |
|          |              | 0.1          | 81.5%          | 81.8%         | -5.2%         | -7.8%             |
|          |              | 0.5          | 73.8%          | 80.3%         | -33.3%        | -27.8%            |
| CIFAR-100| EN B4        | 0.0          | 89.8%          | -             | -             | -                 |
|          |              | 0.1          | -              | 89.3%         | -4.1%         | -6.0%             |
|          |              | 0.3          | -              | 86.3%         | -18.1%        | -22.8%            |

architecture, but it is only applied on the last fully connected layer and is not present in the residual blocks. A drop in replacement of standard dropout by SliceOut would only yield marginal gains. We use instead SliceOut to operate on the “expand convolution” that performs the width expansion in each MBConv block (i.e., the first convolution layer in each block) – this is typically where the largest tensors are created (see detailed architecture diagram in Appendix F.2.2).

We fine-tune EfficientNets models pre-trained on ImageNet (following the standard experiment setup in the EfficientNet paper) to achieve even higher test accuracy on CIFAR-10/100 [21, 23]. We observe speedups and memory savings of up to 40%, with comparable test accuracy to fine tuning with the standard EfficientNets models, despite the fact we fine tune from models originally trained without SliceOut (Fig. 2). When training similar models from scratch, we observe the same speedups and memory gains, and converge to a higher test accuracy for dropout rates \( \leq 0.4 \) (Appendix F.2.2).

4.4 Transformers

Table 3: Transformer results. We observe speedups and memory gains of \( \sim 10\% \) when using SliceOut, despite the fact in Transformers the performance is dominated by looking up embedding vectors. Although Transformers are typically under-parametrised for language modeling on LM1B, SliceOut is a more effective form of regularization compared to standard dropout or controlled dropout (detailed results in Appendix F.3).

| Width | Dropout rate | Perplexity | Controller Perplexity | SliceOut Perplexity | Training time | Max cached memory |
|-------|--------------|------------|-----------------------|---------------------|---------------|-------------------|
| 1024  | 0.0          | 31.65      | -                     | -                   | -             | -                 |
| 2048  | 0.3          | 45.10      | 45.74                 | -                   | -             | -                 |
|       | 28.07        | 33.71      | -8.4%                 | -9.0%               | -             | -                 |
|       | 0.3          | 88.59      | 53.14                 | 28.12               | -11.0%        | -10.0%            |

In our experiments, we evaluate a vanilla Transformer language model on the popular “One Billion Word Benchmark” [24]. Our results are shown in Table 3, we find that we are able to maintain much of the modelling performance of the baseline Transformer while reducing memory overhead by 9% and reducing steptime by at least 8.4%. These reductions are more modest in comparison to ResNets, and this is primarily due to the fact that, in Transformers, a significant portion of steptime is spent looking up embedding vectors and computing logits over a vocabulary of more than 32,000 elements. Similarly, much of the Transformer’s memory is spent on storing the parameters, which SliceOut does not reduce.

Despite these limitations, the importance of SliceOut grows as the network size relative to the embeddings becomes more comparable, as can be seen in Table 4, where doubling the width improves both the time and memory savings. Similarly, when doubling the model width we can see that
the impact of dropping 30% of the neurons is reduced dramatically and the achieved perplexity is comparable to training without dropout.

5 Conclusion

Training modern deep neural networks in a resource-intensive task. As deep learning-based applications become more pervasive, their impact on our environment is ever increasing [25, 26]. SliceOut is an effective approach for speeding up and reducing the memory requirements of neural networks at train time. We demonstrated in this work how the scheme can be beneficial to a diverse set of network architectures across application domains. Successfully leveraging SliceOut in large neural networks can help curtail the $CO_2$ emissions during training by 10-40% (Appendix E).

6 Broader Impact

SliceOut is a method to train models faster and with a smaller memory footprint, while ultimately preserving their statistical accuracy. When developing the idea, we had the following two objectives in mind.

Objective 1: Train larger models faster with the same hardware.

In the past few years we have observed an unprecedented race to training ever larger neural networks with the objective to squeeze in incremental points of accuracy - the latest example being the newly released GPT-3 model with an impressive 175 billion parameters [27]). Significant progress has also been made towards the ability to train large deep networks very rapidly – with several teams competing to train high accuracy models on ImageNet in a few minutes [28, 29]. At the other end of the spectrum, the democratization of deep learning has led a growing number of practitioners to develop custom models for their applications, albeit with more limited means than the aforementioned research teams.

We believe SliceOut can help with these trends and challenges as follows:

- The memory gains it provides can help train models with a higher number of parameters, or with a higher batch size [30], on the same hardware;
- Its speedups can further accelerate our ability to train large models with high accuracy;
- Its demonstrated ability to perform well across application domains and in different training scenario (e.g., training from scratch, fine tuning) can be beneficial to a large number of practitioners that have to deal with limited computing resources.

Objective 2: Reduce the environmental impact from training machine learning models.

Training large machine learning models relies more and more on power-hungry hardware accelerators like GPUs and TPUs. With deep learning becoming mainstream, its growing carbon footprint and environmental impact cannot be understated [25, 26]. SliceOut can help train an otherwise identical model to the same level of accuracy while reducing the resulting carbon emissions by 10-40%, through increased speed and memory efficiency.
References

[1] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors, 2012.

[2] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15(56):1929–1958, 2014.

[3] Jonathan Tompson, Ross Goroshin, Arjun Jain, Yann LeCun, and Christopher Bregler. Efficient object localization using convolutional networks, 2014.

[4] Alex Labach, Hojjat Salehinejad, and Shahrokh Valaee. Survey of dropout methods for deep neural networks, 2019.

[5] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning, 2015.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. MIT Press, 2016. [http://www.deeplearningbook.org]

[7] ByungSoo Ko, H. Kim, Kyo-Joong Oh, and H. Choi. Controlled dropout: A different approach to using dropout on deep neural network. In 2017 IEEE International Conference on Big Data and Smart Computing (BigComp), pages 358–362, 2017.

[8] B. Ko, H. Kim, and H. Choi. Controlled dropout: A different dropout for improving training speed on deep neural network. In 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pages 972–977, 2017.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[10] Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V. Le. Dropblock: A regularization method for convolutional networks, 2018.

[11] Oren Rippel, Michael A. Gelbart, and Ryan P. Adams. Learning ordered representations with nested dropout, 2014.

[12] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. Dropedge: Towards deep graph convolutional networks on node classification, 2019.

[13] https://numpy.org/doc/1.18/reference/arrays.indexing.html, 2020.

[14] https://pytorch.org/docs/stable/tensor_view.html, 2020.

[15] Terrance DeVries and Graham W. Taylor. Improved regularization of convolutional neural networks with cutout, 2017.

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Advances in neural information processing systems, pages 5998–6008, 2017.

[17] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost, 2018.

[18] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11):2278–2324, 1998.

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

[20] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks, 2016.

[21] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2019.

[22] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge, 2014.

[23] Simon Kornblith, Jonathon Shlens, and Quoc V. Le. Do better imagenet models transfer better?, 2018.

[24] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. arXiv preprint arXiv:1312.3005, 2013.

[25] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. Quantifying the carbon emissions of machine learning, 2019.

[26] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp, 2019.
[27] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[28] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes, 2018.

[29] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour, 2017.

[30] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training, 2018.

[31] Yeming Wen, Dustin Tran, and Jimmy Ba. Batchensemble: An alternative approach to efficient ensemble and lifelong learning, 2020.
Appendix

A SliceOut - Additional implementation details

A.1 Constraints on eligible positions for slicing

Modern GPU kernels have subroutines to select the best algorithm based on the shape of tensors involved in operations (e.g., cudnn.benchmark in Pytorch). These routines typically involve benchmarking different alternative algorithms in the first training step (e.g., FFT, Winograd), then keeping the best algorithm(s) for all subsequent steps, as long as the shape of tensors remains the same (otherwise the subroutine runs at each step). To obtain maximal speed-ups, we ensure that the shape of all tensors is constant throughout training, and consequently prevent re-running these optimisation subroutines. This is achieved by restricting the sampled starting index to a subset of eligible positions, specifically, restricting the starting index to be at a position where $\text{start\_index} + \text{slide\_width} \leq \text{layer\_width}$ (see Fig. 5 for a concrete example).

Figure 5: Depiction of eligible indices for slice start - In the example above, the layer width is 10, and dropout rate 40%. Consequently, slices are contiguous sets of 6 units and the last 5 positions (i.e., indices 5 to 9) are ineligible to be the slice start to enforce strict same-sized slicing of contiguous units.

B Activation normalisation & moment preservation

We show here how the two normalisations schemes described in §3 – exactly or approximately – preserve the moments of the layer output (pre-activation).

Let $x$ be the input tensor at a layer $W$, with $n$ input units & $m$ output units, where the SliceOut function $S(.)$ is applied. We note norm$(.)$ the normalisation operator used in SliceOut (“Flow” or “Probabilistic” normalisation, as described in §3.2), and $\mathbb{1}\{\cdot\}$ the Heaviside step function – where $\mathbb{1}\{\text{unit}_j \text{ kept}\}$ indicates whether unit $j$ in the layer is kept. $\forall j \in [1, m]$, $e_j$ is the canonical basis vector for the $j^{th}$ dimension of the output tensor space.

B.1 First moment preservation

B.1.1 No SliceOut

Using linearity of expectation, we obtain the following formula for the expected value of the output tensor:

$$E(W.x) = E(\sum_{j=1}^{m} \sum_{i=1}^{n} w_{j,i} \ast x_i) e_j = \sum_{j=1}^{m} \sum_{i=1}^{n} w_{j,i} \ast E(x_i) e_j = W.E(x)$$

B.1.2 With SliceOut

Similarly, when SliceOut is applied we obtain the following since the expectation and normalisations operators are both linear and slicing is performed at random:

$$E(S(W.x)) = E(\text{norm}(\sum_{j=1}^{m} \sum_{i=1}^{n} \mathbb{1}\{\text{unit}_j \text{ kept}\} \ast \mathbb{1}\{\text{unit}_i \text{ kept}\} \ast w_{j,i} \ast x_i) e_j))$$

$$= \sum_{j=1}^{m} \sum_{i=1}^{n} P\{\text{unit}_j \text{ kept}\} \ast P\{\text{unit}_i \text{ kept}\} \ast \text{norm\_out}(w_{j,i}) \ast \text{norm\_in}(E(x_i)) e_j$$

$$= \sum_{j=1}^{m} \sum_{i=1}^{n} w_{j,i} \ast E(x_i) e_j = W.E(x)$$
In (2b) \( \text{norm}_\text{out}(\cdot) \) refers to the normalisation of the output, while \( \text{norm}_\text{in}(\cdot) \) refers to the normalisation of the input (relevant only if SliceOut was also applied on the prior layer).

Equality (2c) is exact when using the “Probabilistic normalisation”, since the normalisation scheme consists exactly in dividing activations by the probability that the corresponding unit is kept. It is approximate in the case of the “Flow” normalisation, since we divide by the average probability that a unit is kept – across the layer – which differs from the probability that a given unit is kept (e.g., units near the edges are more likely to be dropped for reasons detailed in Appendix A.1).

B.2 Second moment preservation

Let \( y \) be the output tensor at layer \( W \).

B.2.1 No SliceOut

\[
\forall (j_1, j_2) \in [1, m]^2, \text{Cov}(y_{j_1}, y_{j_2}) = E(y_{j_1} \ast y_{j_2}) - E(y_{j_1}) \ast E(y_{j_2})
\]

(3)

B.2.2 With SliceOut

\[
\forall (j_1, j_2) \in [1, m]^2, \text{Cov}(S(y_{j_1}), S(y_{j_2})) = E(S(y_{j_1}) \ast S(y_{j_2})) - E(S(y_{j_1})) \ast E(S(y_{j_2}))
\]

(4)

Since we have just seen preservation for the second term, we will focus on the first term only:

\[
E(S(y_{j_1}) \ast S(y_{j_2})) = E(S(\sum^n_{i=1} w_{j_1, i} \ast x_i) \ast S(\sum^n_{i=1} w_{j_2, i} \ast x_i))
\]

(5a)

\[
= E(\sum^n_{i=1} \sum^n_{k=1} \mathbb{I}\{\text{unit}_i \text{ kept}\} \ast \mathbb{I}\{\text{unit}_k \text{ kept}\} \ast \mathbb{I}\{\text{unit}_j_1 \text{ kept}\} \ast \mathbb{I}\{\text{unit}_j_2 \text{ kept}\} \ast \text{norm}_\text{out}(w_{j_1, i}) \ast \text{norm}_\text{in}(x_i) \ast \text{norm}_\text{out}(w_{j_2, k}) \ast \text{norm}_\text{in}(x_k))
\]

(5b)

\[
= \sum^n_{i=1} \sum^n_{k=1} \mathbb{P}\{\text{unit}_j_1 \text{ kept}\} \ast \mathbb{P}\{\text{unit}_j_2 \text{ kept}\} \ast \mathbb{P}\{\text{unit}_j_1 \text{ kept}\} \ast \mathbb{P}\{\text{unit}_j_2 \text{ kept}\} \ast \text{norm}_\text{out}(w_{j_1, i}) \ast \text{norm}_\text{in}(x_i) \ast \text{norm}_\text{out}(w_{j_2, k}) \ast \text{norm}_\text{in}(x_k)
\]

(5c)

When considering the “Probabilistic normalisation”, we thus have:

\[
\forall (j_1, j_2) \in [1, m]^2, E(S(y_{j_1}) \ast S(y_{j_2})) = \sum^n_{i=1} \sum^n_{k=1} \mathbb{P}\{\text{unit}_j_1 \text{ kept}\} \ast \mathbb{P}\{\text{unit}_j_2 \text{ kept}\} \ast \text{norm}_\text{out}(w_{j_1, i}) \ast \text{norm}_\text{out}(w_{j_2, k}) \ast \text{norm}_\text{in}(x_k)
\]

(6)

Unlike in standard dropout, for SliceOut we usually have:

\[
\forall (j_1, j_2) \in [1, m]^2, \mathbb{P}\{\text{unit}_j_1 \text{ kept}\} \ast \mathbb{P}\{\text{unit}_j_2 \text{ kept}\} \neq \mathbb{P}\{\text{unit}_j_2 \text{ kept}\}
\]

(7)

due to the structure imposed by slicing contiguous units, and therefore the second moment is not exactly preserved.

However, for dropout rates \( p \leq 0.5 \) we have: \( \forall j_1 \in [\text{slice}_\text{width}, m - \text{slice}_\text{width}], \forall j_2 \in [1, m], \)

\[
\mathbb{P}\{\text{unit}_j_1 \text{ kept}\} \ast \mathbb{P}\{\text{unit}_j_2 \text{ kept}\} = \mathbb{P}\{\text{unit}_j_1 \text{ kept}\} = 1
\]

\[
\mathbb{P}\{\text{unit}_j_2 \text{ kept}\} \ast \mathbb{P}\{\text{unit}_j_1 \text{ kept}\} = \mathbb{P}\{\text{unit}_j_2 \text{ kept}\}.
\]

(8)

Thus we approximately have:

\[
\forall (j_1, j_2) \in [1, m]^2, E(S(y_{j_1}) \ast S(y_{j_2})) \sim \sum^n_{i=1} \sum^n_{k=1} w_{j_1, i} \ast x_i \ast w_{j_2, k} \ast x_k = E(y_{j_1} \ast y_{j_2})
\]

(9)

Similarly, the “Flow normalisation” approximately preserves the second moment, with the same additional approximation discussed in the first moment section.
C  Number of architectures sampled from

Let’s consider a simple feedforward network architecture with 3 hidden layers. We illustrate over a few scenarios the number of model architectures that SliceOut effectively samples from at layer 2, i.e., the number of distinct dropout masks at layer 2 resulting from applying SliceOut (see Fig. 6 for an example of a mask at layer 2). We assume that the first hidden layer has width \( n \) and the second hidden layer has width \( m \). The SliceOut probabilities at these layers are respectively \( p_1 \) and \( p_2 \) (when SliceOut is indeed applied at the corresponding layer).

C.1 Scenario 1: SliceOut applied only on hidden layer 1

In that scenario, we apply SliceOut only on hidden layer 1, which means that we sample a contiguous set of \( n \times (1 - p_1) \) output units from hidden layer 1. As a result, we also need to slice the columns of the weight matrix at layer 2 to match that same slice. There is a linear number of distinct slices of length \( n \times (1 - p_1) \) we can take, therefore we sample from a linear number of architectures at layer 2.

C.2 Scenario 2: SliceOut applied only on hidden layer 2

This scenario is similar to scenario 1 except that we now need to slice the rows of the weight matrix at layer 2 to extract a slice of length \( m \times (1 - p_2) \). There is also a linear number of distinct architectures sampled at layer 2.

C.3 Scenario 3: SliceOut is applied on both hidden layers 1 and 2

In that scenario, we both have to slice the columns and the rows of the weight matrix at layer 2. There are \( n \times (1 - p_1) \times m \times (1 - p_2) = O(n \times m) \) distinct pairs of slices we can take on rows and columns of the weight matrix at layer 2, hence a quadratic number of distinct model architectures sampled from (quadratic in the layer width).

D  Connection to BatchEnsemble

During training, the forward pass for a given layer where SliceOut is applied can be written as: \( y = f((W(x \odot c)) \odot r) \) where \( W \) is the weight tensor, \( x \) is the activations tensor from the prior layer, \( y \) the activations tensor at this layer, and \( c \) and \( r \) are the dropout masks applied at this layer, operating respectively on the columns and rows of \( W \). This can be re-written: \( y = f((W \odot S)x) \), where \( \odot \) is the Hadamard product between \( W \) and the rank-1 matrix \( S = r \otimes c \).

There is a striking parallel with the BatchEnsemble approach \[31\], a parameter-efficient approach to ensembling several models together, with two major differences:

- The parameters \( c \) and \( r \) are learned separately for each ensemble member in the BatchEnsemble case, while they are constant filters in SliceOut.
We are ensembling a sub-quadratic number of architectures in SliceOut, while there is no such restriction in the BatchEnsemble case (although the number of ensemble members is typically much smaller in practice, e.g., 2-10 models)

E CO$_2$ emissions calculation

Given the speedups and memory gains it provides during training, SliceOut can help reduce the energy consumption, and thereby the CO$_2$ emissions, that would result from training a given model in three different ways:

- **Approach 1 - Reduced number of machines needed during training**: Thanks to the memory efficiency provided by SliceOut, a model – that would require a given number of GPUs when trained with standard dropout – could be trained with fewer GPUs with SliceOut without degrading performance. For example, we saw in the Wide ResNets experiments on CIFAR-10 (Appendix F.2.1) that a model with similar accuracy can be trained 39% faster and with 23% lower GPU memory requirements when using SliceOut (with a dropout rate 0.5). As a thought experiment, we compared the relative speed when training that model with SliceOut on 3 GPUs, compared to training the same model without SliceOut on 4 GPUs (keeping the ratio of CPUs per GPU constant); when training with SliceOut on 3 GPUs, the train time per epoch was 1% lower. Therefore, training the same model on 25% fewer GPU for about the same amount of time would result in CO$_2$ emissions that are 25% lower;

- **Approach 2 - Same hardware, higher batch size**: Instead of reducing the number of GPUs to be used during training, we may want instead to further increase the batch size to make the most of the additional GPU memory available. Using the same example as before, while this time keeping the same number of GPUs with and without SliceOut (1 GPU) but increasing the batch size so that the GPU memory footprint is identical with and without SliceOut, we observe new speedups from SliceOut – and thus CO$_2$ emissions reductions – of 41%;

- **Approach 3 - Same hardware, same hyperparameters, just faster training**: In cases where neither reducing the number of machines used during training nor increasing the batch size are desirable, training models with SliceOut can still help reduce CO$_2$ emissions through the speedups it provides, i.e., 10% with Transformers or up to 40% in some of the CNNs architectures we analysed.

To summarise, SliceOut can help train models with comparable accuracy while resulting in CO$_2$ emissions 10%-40% lower, depending on the model architecture, the set of hyperparameters chosen and the hardware used for training.

F Detailed experimental results

In this section we present the comprehensive set of experimental results obtained across our experiments with fully-connected networks, Wide ResNets, EfficientNets and Transformers.

F.1 Fully connected networks experiments

F.1.1 Experimental setting

**Objectives.** In these experiments we aim to study the quality and stability of our SliceOut scheme in a simple setting and quantify the impact on memory savings and computation speedups over standard and controlled dropout. The purpose is not to obtain state-of-the-art performance on these datasets (the architecture leveraged here would not allow us to do that), but rather to confirm the validity and expected benefits of the SliceOut scheme with a simple architecture and commonly used datasets.

**The MNIST and FashionMNIST datasets.** The MNIST [18] and FashionMNIST [19] datasets consist of grayscale images of 28x28 pixels, respectively representing the 10 digits and items from 10 distinct object classes from Zalando’s catalogue (e.g., shoes, bags, dresses). No data transformation is applied on the input. FashionMNIST is available at the following location: https://github.com/zalandoresearch/fashion-mnist/tree/master/data/fashion. MNIST is available at the following location: http://yann.lecun.com/exdb/mnist/. We used the same train/test split as from these sources.

**Model architecture.** For both the MNIST and FashionMNIST experiments we use the same model architecture: a simple fully connected network with 3-hidden layers. The input layer has 28x28 = 784 units (one for each image pixel), each hidden layer has 2048 units, and the output layer has 10 units (one for each class). We apply standard dropout, controlled dropout and SliceOut on each hidden layer, and vary the dropout rate from 0
to 0.9 by increment of 0.1. For SliceOut, we compare the two types of normalisation discussed in § 3.2 – in both cases, normalisation is applied immediately after SliceOut (unlike what we do in CNNs – see Appendix F.2).

Training procedure and hyperparameters. We minimize the cross-entropy loss between predictions and labels via the Adam algorithm. We used Pytorch (https://pytorch.org/docs/stable/index.html) to instantiate and train the different models. All hyperparameters are summarised in table 4, using the Pytorch convention for the hyperparameter names (https://pytorch.org/docs/stable/optim.html).

| Hyperparameter   | Value       |
|------------------|-------------|
| Batch size       | 256         |
| Learning rate    | $10^{-4}$   |
| Beta1            | 0.9         |
| Beta2            | 0.999       |
| Epsilon          | $10^{-8}$   |
| Weight decay     | 0.0         |

Hardware. All results for our FashionMNIST and MNIST experiments were obtained with a single GPU (an Nvidia RTX 2080) and averaged across 4 independent runs.

F.1.2 FashionMNIST experiments results

As discussed in § B.1 we observe speedups (up to 15%), cached GPU memory savings (up to 30%), faster convergence and to a higher test accuracy value with SliceOut, when typical dropout rates are applied (i.e., $p \leq 0.5$). “Flow” normalisation appears to be leading to higher test accuracy in this set of experiments compared to “Probabilistic” normalisation, although the difference is not always statistically significant (see Table 5). We note that SliceOut may actually underperform compared to standard dropout, in the case of extreme dropout (e.g., $p \geq 0.9$) as it would lead to only ensembling under-capacitated networks.

Controlled dropout was more memory efficient than standard dropout only beyond a certain dropout rate ($p = 0.4$), as the gains from storing smaller activations in memory (for the backward pass) become higher than the data duplication overhead resulting from the gather ops. SliceOut always consumed less memory as there is now such duplication.

| Dropout rate | Standard dropout | SliceOut - Flow normalisation | SliceOut - Proba. normalisation | Controlled dropout |
|--------------|------------------|-------------------------------|------------------------------|-------------------|
| 0.0          | 89.6 ± 0.04 %    | -                             | -                            | 89.8 ± 0.05 %     |
| 0.1          | 89.6 ± 0.08 %    | **90.0±0.02 %**               | **90.0±0.06 %**              | 89.7 ± 0.03 %     |
| 0.2          | 89.4 ± 0.03 %    | **90.0±0.04 %**               | 89.7 ± 0.03 %               | 89.7 ± 0.03 %     |
| 0.3          | 89.2 ± 0.02 %    | **90.0±0.07 %**               | 89.6 ± 0.04 %               | 89.6 ± 0.05 %     |
| 0.4          | 89.0 ± 0.05 %    | 89.7 ± 0.05 %                 | 89.5 ± 0.05 %               | 89.4 ± 0.04 %     |
| 0.5          | 88.8 ± 0.03 %    | 89.6 ± 0.05 %                 | 89.5 ± 0.07 %               | 89.3 ± 0.04 %     |
| 0.6          | 88.5 ± 0.03 %    | 89.4 ± 0.03 %                 | 89.2 ± 0.05 %               | 89.1 ± 0.02 %     |
| 0.7          | 88.2 ± 0.04 %    | 89.1 ± 0.02 %                 | 88.6 ± 0.30 %               | 88.7 ± 0.02 %     |
| 0.8          | 87.8 ± 0.01 %    | 88.2 ± 0.01 %                 | 87.7 ± 0.03 %               | 87.4 ± 0.01 %     |
| 0.9          | 86.0 ± 0.02 %    | 72.9 ± 0.17 %                 | 73.0 ± 0.10 %               | 66.8 ± 1.87 %     |

F.1.3 MNIST experiments results

The results of the MNIST experiments are consistent with what we discussed in the FashionMNIST experiments in terms of speedups and memory gains. However, there was no statistically-significant difference in terms of test accuracy, all models converging to the same high test accuracy values ($\sim 98.5\%$).
Table 6: FashionMNIST experiments - Train time per epoch (% of standard dropout)

| Dropout rate | SliceOut - Flow normalisation | SliceOut - Proba. normalisation | Controlled dropout |
|--------------|-------------------------------|---------------------------------|-------------------|
| 0.1          | 98.9 ± 0.22%                 | 98.5 ± 0.76%                   | 153.6 ± 0.83%     |
| 0.2          | 95.3 ± 0.54%                 | 95.4 ± 0.68%                   | 145.8 ± 0.94%     |
| 0.3          | 93.4 ± 0.51%                 | 93.4 ± 0.54%                   | 138.8 ± 0.71%     |
| 0.4          | 90.5 ± 0.65%                 | 91.2 ± 0.84%                   | 132.7 ± 0.75%     |
| 0.5          | 87.3 ± 0.39%                 | 87.8 ± 0.60%                   | 125.9 ± 0.87%     |
| 0.6          | 87.0 ± 0.66%                 | 87.5 ± 0.63%                   | 121.0 ± 0.59%     |
| 0.7          | 86.4 ± 0.85%                 | 86.9 ± 0.76%                   | 115.5 ± 0.84%     |
| 0.8          | 87.1 ± 0.91%                 | 86.7 ± 0.80%                   | 115.2 ± 1.03%     |
| 0.9          | 87.6 ± 0.77%                 | 86.9 ± 0.83%                   | 114.7 ± 0.98%     |

Table 7: FashionMNIST experiments - Max GPU cached memory (% of standard dropout). No confidence interval reported in the table below as the max memory usage was strictly equal across all experiments.

| Dropout rate | SliceOut - Flow normalisation | SliceOut - Proba. normalisation | Controlled dropout |
|--------------|-------------------------------|---------------------------------|-------------------|
| 0.1          | 93.2%                         | 93.2%                           | 148.3%            |
| 0.2          | 87.0%                         | 87.0%                           | 132.1%            |
| 0.3          | 81.5%                         | 81.5%                           | 116.4%            |
| 0.4          | 76.6%                         | 76.6%                           | 103.7%            |
| 0.5          | 73.0%                         | 73.0%                           | 92.3%             |
| 0.6          | 69.1%                         | 69.1%                           | 82.1%             |
| 0.7          | 66.8%                         | 66.8%                           | 76.7%             |
| 0.8          | 65.4%                         | 65.4%                           | 70.4%             |
| 0.9          | 65.4%                         | 65.4%                           | 66.2%             |

Table 8: MNIST experiments - Highest test accuracy across dropout schemes

| Dropout rate | Standard dropout | SliceOut - Flow normalisation | SliceOut - Proba. normalisation | Controlled dropout |
|--------------|------------------|-------------------------------|---------------------------------|-------------------|
| 0.1          | 98.4 ± 0.03%     | 98.5 ± 0.03%                 | 98.5 ± 0.04%                   | 98.5 ± 0.03%     |
| 0.2          | 98.5 ± 0.02%     | 98.5 ± 0.03%                 | 98.5 ± 0.04%                   | 98.6 ± 0.02%     |
| 0.3          | 98.5 ± 0.04%     | 98.4 ± 0.01%                 | 98.4 ± 0.04%                   | 98.6 ± 0.01%     |
| 0.4          | 98.5 ± 0.01%     | 98.5 ± 0.03%                 | 98.4 ± 0.01%                   | 98.6 ± 0.02%     |
| 0.5          | 98.6 ± 0.03%     | 98.5 ± 0.04%                 | 98.4 ± 0.05%                   | 98.6 ± 0.03%     |
| 0.6          | 98.6 ± 0.04%     | 98.5 ± 0.03%                 | 98.4 ± 0.02%                   | 98.6 ± 0.02%     |
| 0.7          | 98.6 ± 0.02%     | 98.4 ± 0.02%                 | 98.3 ± 0.02%                   | 98.4 ± 0.04%     |
| 0.8          | 98.3 ± 0.01%     | 98.0 ± 0.02%                 | 97.8 ± 0.02%                   | 97.6 ± 0.01%     |
| 0.9          | 97.3 ± 0.02%     | 94.8 ± 0.03%                 | 94.4 ± 0.07%                   | 92.6 ± 0.14%     |

F.2 Convolutional neural networks experiments

F.2.1 Wide ResNets

**Objectives.** The main objective of our Wide ResNets experiments was to demonstrate that the benefits of the method are even greater with convolutional neural networks due to the relatively higher footprint of activations in GPU memory (because of weight sharing), and to show that these benefits can be achieved on a larger and more complex set of data. The choice of Wide ResNets [20] was very natural as they achieve high test accuracy
Table 9: MNIST experiments - Train time per epoch (% of standard dropout)

| Dropout rate | SliceOut - Flow normalisation | SliceOut - Proba. normalisation | Controlled dropout |
|--------------|-------------------------------|---------------------------------|--------------------|
| 0.1          | 98.3 ± 0.38 %                 | 97.6 ± 0.69 %                  | 153.9 ± 0.85 %     |
| 0.2          | 94.7 ± 0.52 %                 | 95.4 ± 0.70 %                  | 144.7 ± 1.09 %     |
| 0.3          | 93.1 ± 0.79 %                 | 92.8 ± 0.83 %                  | 138.1 ± 0.71 %     |
| 0.4          | 91.0 ± 0.74 %                 | 90.9 ± 0.93 %                  | 131.4 ± 1.07 %     |
| 0.5          | 88.5 ± 0.70 %                 | 88.6 ± 0.71 %                  | 126.8 ± 0.65 %     |
| 0.6          | 86.7 ± 0.51 %                 | 87.4 ± 0.98 %                  | 119.9 ± 0.58 %     |
| 0.7          | 87.6 ± 1.20 %                 | 86.4 ± 0.60 %                  | 116.0 ± 0.66 %     |
| 0.8          | 88.1 ± 0.86 %                 | 86.0 ± 0.76 %                  | 115.6 ± 0.68 %     |
| 0.9          | 86.0 ± 0.55 %                 | 84.6 ± 0.47 %                  | 113.5 ± 0.80 %     |

Table 10: MNIST experiments - Max GPU cached memory (% of standard dropout). No confidence interval reported in the table below as the max memory usage was strictly equal across all experiments.

| Dropout rate | SliceOut - Flow normalisation | SliceOut - Proba. normalisation | Controlled dropout |
|--------------|-------------------------------|---------------------------------|--------------------|
| 0.1          | 93.2%                         | 93.2%                           | 148.4%             |
| 0.2          | 87.0%                         | 87.0%                           | 132.0%             |
| 0.3          | 81.5%                         | 81.5%                           | 116.2%             |
| 0.4          | 76.6%                         | 76.6%                           | 104.5%             |
| 0.5          | 73.0%                         | 73.0%                           | 92.3%              |
| 0.6          | 69.1%                         | 69.1%                           | 82.3%              |
| 0.7          | 66.8%                         | 66.8%                           | 76.5%              |
| 0.8          | 65.4%                         | 65.4%                           | 70.5%              |
| 0.9          | 65.4%                         | 65.4%                           | 66.2%              |

across many vision tasks, and include dropout in each residual block as a way to mitigate potential overfitting risk due to the channel widening.

The CIFAR-10 and CIFAR-100 datasets. The CIFAR-10 and CIFAR-100 datasets contain 32x32 color images of respectively 10 and 100 distinct object or animal classes (e.g., airplane, automobile, cat, dog). The datasets can be obtained at the following location: https://www.cs.toronto.edu/kriz/cifar.html. We used the same train / test split as from the source website, i.e., 50k images in the training data and 10k images in the test data. Following the data preparation approach of the original Wide ResNet paper [20], our data augmentation consisted only of random horizontal flips and random crops of 224 pixels.

Model architecture. The overall model architecture used in our experiments follows very closely the original Wide ResNets architecture. The only notable difference is the type of dropout used across experiments (standard dropout Vs SliceOut) – see detailed architecture diagram in Fig[7]. When using SliceOut, it was critical to align the sampled slices between the two convolution layers and the batch norm in-between (all items in orange font on Fig[7]).

We experimented with the “Flow” and “Probabilistic” normalisations schemes, and observed systematically higher test accuracy with the latter (see Tables[12] and [13]). "Delaying" the normalisation until after the batch norm and right before the second convolution also helped further increase accuracy.

We additionally compared the performance of the two types of SliceOut described in §3.4 and observed higher test accuracy with “Channel SliceOut” over “Patch SliceOut” (see Table[14]).

For all experiments, we analysed the benefits of SliceOut on the 40x10 Wide ResNet architecture since authors of the original Wide ResNet paper achieved their highest test accuracy on CIFAR-10 and CIFAR-100 with this architecture.
Training procedure and hyperparameters. We also closely followed the training process from the original Wide ResNet paper \cite{20}, i.e. minimising the cross-entropy loss via SGD with momentum over 200 epochs, with an identical learning rate schedule. We summarise all the hyperparameters used in the table below:

**Table 11: Wide ResNets experiments - List of hyperparameters used for training**

| Hyperparameter            | Value                      |
|---------------------------|----------------------------|
| Batch size                | 128                        |
| Initial learning rate     | 0.1                        |
| Learning rate schedule    | Dropped by 0.2 at epochs 60,120 and 160 |
| Momentum                  | 0.9                        |
| Dampening                 | 0.0                        |
| Weight decay              | $5 \times 10^{-4}$         |

Hardware. All our Wide ResNets results were obtained by running experiments (single runs) on a single GPU (an Nvidia Titan RTX).

Figure 7: Sliceout - Wide ResNet Residual block

Experiments results. We observe speedups of up to $\sim 40\%$ and memory gains of up to $\sim 30\%$, with a test accuracy that matches the value obtained for the best baseline Wide ResNet models (with standard dropout). “Channel SliceOut” tends to outperform “Patch SliceOut” across experiments. The “Probabilistic” normalisation delivers higher test accuracy over the “Flow” normalisation for “Channel SliceOut” (see Tables 13 and 12). For “Patch SliceOut”, stable learning was only achieved with the “Flow” normalisation.

F.2.2 EfficientNets

Objectives. The purpose of the EfficientNets experiments was threefold: firstly, we wanted to demonstrate that SliceOut could be leveraged in an architecture achieving state-of-the-art performance on the CIFAR datasets; secondly, we wanted to illustrate that SliceOut can be used more broadly as a tool to achieve memory gains and speedups even if the original architecture did not use (standard) dropout in the first place; thirdly, we also wanted to demonstrate the fact that SliceOut could be used effectively when fine-tuning a model that was originally trained without it.
Table 12: Wide ResNets - Channel SliceOut with “Probabilistic” normalisation. Training time & Max cached GPU memory are resp. the relative % of train time per epoch for a network trained with SliceOut Vs standard dropout, and the maximum cached GPU memory during training.

| Dataset    | Architecture | Dropout rate | Test accuracy (Standard dropout) | Test accuracy (SliceOut) | Training time | Max cached memory |
|------------|--------------|--------------|----------------------------------|--------------------------|---------------|-------------------|
| CIFAR-10   | 40x10        | 0.0          | 96.3%                            | -                        | -             | -                 |
|            |              | 0.1          | 96.2%                            | 96.3% (5%)               | -8%           | -                 |
|            |              | 0.2          | 95.9%                            | 96.4% (18%)              | -13%          | -                 |
|            |              | 0.3          | 95.6%                            | 96.2% (21%)              | -17%          | -                 |
|            |              | 0.4          | 94.5%                            | 96.3% (34%)              | -23%          | -                 |
|            |              | 0.5          | 93.9%                            | 96.1% (39%)              | -23%          | -                 |
| CIFAR-100  | 40x10        | 0.0          | 81.5%                            | -                        | -             | -                 |
|            |              | 0.1          | 81.5%                            | 81.8% (5%)               | -8%           | -                 |
|            |              | 0.2          | 80.7%                            | 81.1% (18%)              | -13%          | -                 |
|            |              | 0.3          | 78.7%                            | 80.8% (20%)              | -17%          | -                 |
|            |              | 0.4          | 75.3%                            | 80.2% (30%)              | -23%          | -                 |
|            |              | 0.5          | 73.8%                            | 80.3% (33%)              | -28%          | -                 |

Table 13: Wide ResNets - Channel SliceOut with “Flow” normalisation. Training time & Max cached GPU memory are resp. the relative % of train time per epoch for a network trained with SliceOut Vs standard dropout, and the maximum cached GPU memory during training.

| Dataset    | Architecture | Dropout rate | Test accuracy (Standard dropout) | Test accuracy (SliceOut) | Training time | Max cached memory |
|------------|--------------|--------------|----------------------------------|--------------------------|---------------|-------------------|
| CIFAR-10   | 40x10        | 0.0          | 96.3%                            | -                        | -             | -                 |
|            |              | 0.1          | 96.2%                            | 95.9% (5%)               | -8%           | -                 |
|            |              | 0.2          | 95.9%                            | 94.9% (18%)              | -13%          | -                 |
|            |              | 0.3          | 95.6%                            | 90.5% (21%)              | -17%          | -                 |
|            |              | 0.4          | 94.5%                            | 74.4% (34%)              | -23%          | -                 |
|            |              | 0.5          | 93.9%                            | 72.2% (39%)              | -23%          | -                 |
| CIFAR-100  | 40x10        | 0.0          | 81.5%                            | -                        | -             | -                 |
|            |              | 0.1          | 81.5%                            | 80.6% (5%)               | -8%           | -                 |
|            |              | 0.2          | 80.7%                            | 77.1% (18%)              | -13%          | -                 |
|            |              | 0.3          | 78.7%                            | 68.0% (20%)              | -17%          | -                 |
|            |              | 0.4          | 75.3%                            | 33.1% (30%)              | -23%          | -                 |
|            |              | 0.5          | 73.8%                            | 38.7% (33%)              | -28%          | -                 |

Datasets. We leveraged the CIFAR-10 and CIFAR-100 datasets, as described in § F.2.1. Regarding data augmentation, we used the following to train both the baseline models and the SliceOut equivalent:

- CIFAR auto-augment policies (as per the original EfficientNet paper [21])
- Random horizontal flips (as per the official tensorflow implementation: https://github.com/tensorflow/tpu/tree/master/models/official/efficientnet)
- Bicubic image interpolation to resize images to the required resolution needed by the different EfficientNet architectures (also as per the official tensorflow implementation)
- Cutout [15]

Model architecture. Although standard dropout is used in the original EfficientNets model architecture, it is only applied on the last fully connected layer and not in any of the residual blocks, offering very limited opportunity to obtain speedups and memory gains if replaced by SliceOut. We instead apply SliceOut on the very first convolution layer in each MBConv block (see Fig. 8) – this is where the channel width expansion is being performed, and typically where the largest tensors are being created. We propagate the sliced tensor throughout the block, up until the final "Project block" where we perform the delayed normalisation (see following paragraph). Similar to what we discussed with Wide ResNets, it is critical to ensure the same slice is being used throughout the block (all elements in orange font in Fig. 8).
Table 14: Wide ResNets - Patch SliceOut with “Flow” normalisation
Training time & Max cached GPU memory are resp. the relative % of train time per epoch for a network trained with SliceOut Vs standard dropout, and the maximum cached GPU memory during training.

| Dataset   | Architecture | Dropout rate | Test accuracy | Test accuracy | Training time | Max cached memory |
|-----------|--------------|--------------|---------------|---------------|---------------|-------------------|
|           |              |              | Standard dropout | SliceOut       |               |                   |
| CIFAR-10  | 40x10        | 0.0          | 96.3%          | -             | -             | -                 |
|           |              | 0.1          | 96.2%          | 96.0%         | -5%           | -8%               |
|           |              | 0.2          | 95.9%          | 95.8%         | -7%           | -13%              |
|           |              | 0.3          | 95.6%          | 95.0%         | -10%          | -18%              |
|           |              | 0.4          | 94.5%          | 94.9%         | -24%          | -23%              |
|           |              | 0.5          | 93.9%          | 92.5%         | -26%          | -24%              |
| CIFAR-100 | 40x10        | 0.0          | 81.5%          | -             | -             | -                 |
|           |              | 0.1          | 81.5%          | 80.5%         | -3%           | -8%               |
|           |              | 0.2          | 80.7%          | 80.0%         | -6%           | -13%              |
|           |              | 0.3          | 78.7%          | 78.4%         | -9%           | -18%              |
|           |              | 0.4          | 75.3%          | 76.9%         | -22%          | -23%              |
|           |              | 0.5          | 73.8%          | 73.6%         | -25%          | -24%              |

Delayed normalisation. One of the key characteristics of the MBConv block used in EfficientNets is the fact that, after the “expand convolution” is performed, each channel is dealt with independently from other channels by the subsequent two blocks (depthwise and squeeze-and-excite blocks). We found experimentally that delaying the (probabilistic) normalisation of activations after the squeeze-and-excite layer, and right before the projection convolution, provided the best stability to learning and ultimately the highest test accuracy.

Training procedure and hyperparameters. We compared two different approaches to train EfficientNet models on CIFAR: fine-tuning models trained on ImageNet (without SliceOut) and training from scratch. Higher accuracy can typically be achieved on CIFAR by fine-tuning a model previously trained on ImageNet [21, 23]. In order to perform the fine-tuning, we use the same procedure as described in [23], i.e., we minimise the cross-entropy loss with SGD over 200 epochs and the hyperparameters listed in Table 15. Optimal values for the learning rates and weight decay were obtained by performing a grid search for the B0 architecture (without SliceOut) over 7 logarithmically spaced learning rates between 0.0001 and 0.1, and 7 logarithmically spaced weight decay to learning rate ratios between $10^{-6}$ and $10^{-3}$, as well as no weight decay (this was also similar to the procedure in [23]; we subsequently used the same hyperparameters for all experiments). We used the same training procedure and hyperparameters when training the models from scratch. In all EfficientNets experiments, we focused on the “Channel SliceOut” scheme with probabilistic normalisation, as it had delivered superior performance in the Wide ResNets experiments.

Table 15: EfficientNet experiments - List of hyperparameters used for training

| Hyperparameter          | Value   |
|-------------------------|---------|
| Batch size              | 128     |
| Initial learning rate   | 0.01    |
| Learning rate schedule  | Cosine annealing |
| Momentum                | 0.9     |
| Dampening               | 0.0     |
| Weight decay            | $10^{-4}$|
| Batch norm momentum     | 0.99    |

Hardware. We conducted all our experiments (single runs) on 1, 2 or 4 Titan RTX GPUs, depending on the network architecture considered (e.g., B0 architecture trained on 1 GPU, B2 on 2 GPUs, B4 trained on 4 GPUs).

Experiments results. In the fine-tuning experiments, we observe significant speedups (up to 30%) and memory gains (up to 33%) with minimal impact on test accuracy up to a dropout rate of 0.4 on CIFAR-10 and up to 0.3 in the CIFAR-100 experiments, despite the fact we fine tune models that were previously trained without SliceOut. Higher dropout rate values degrade performance as they lead to ensembling under-capacitated...
architectures for CIFAR. When training from scratch, we observed similar speedups and memory gains, except the test accuracy obtained with SliceOut is higher than the baseline for all dropout rates $\leq 0.4$.

F.3 Transformers

Objectives. The purpose of our Transformers experiments was to demonstrate the benefits of SliceOut in a different application domain (language modelling), with another model architecture and with a larger dataset.

The LM1B Dataset. The LM1B dataset is a benchmark corpus for measuring progress in statistical language modeling, with about one billion words in the training data. The dataset can be obtained at the following location: http://www.statmt.org/lm-benchmark/. We used the same train / test split as from the source website, with 30,301,028 examples in training data and 306,688 examples in test data.

Model architecture. The model architecture follows very closely the original Transformer architecture [16]. Each model had 6 encoder and decoder layers and the number of attention heads was set to 4. The dimension of feed-forward network was set to 4 times the embedding dimension. We experimented with the normalisation schemes discussed in §3.2 and observed that the “Probabilistic” normalisation performed best.

Training procedure and hyperparameters. We followed the same training process as in the original Transformer paper [16] (e.g., stochastic optimisation with Adam algorithm, same learning rate schedule as in the original paper). We trained the different models with batch sizes of 256 and 1024, and selected the value providing the lowest perplexity for each model. The models with embedding dimension of 1024 were trained for 240,000 steps and the model with embedding dimension of 2048 were trained for 300,000 steps. The models trained with SliceOut were finetuned without SliceOut for the last 1% of the training steps.
Table 16: EfficientNets results. Training time & Max cached GPU memory are resp. the relative % of train time per epoch for a network trained with SliceOut Vs standard dropout, and the maximum cached GPU memory during training. "EN" to "EfficientNets". “FT” refers to results that were obtained by fine tuning a model trained on ImageNet (without SliceOut) and “SC” refers to results obtained when training the same architecture from scratch.

| Dataset  | Architecture | Dropout rate | Test accuracy Standard dropout | Test accuracy SliceOut | Training time | Max cached memory |
|----------|--------------|--------------|---------------------------------|------------------------|---------------|-------------------|
| CIFAR-10 | EN B2 FT     | 0.0          | 98.4%                           | -                      | -             | -                 |
|          |              | 0.1          | -                               | 98.3%                  | -8%           | -9%               |
|          |              | 0.2          | -                               | 97.9%                  | -13%          | -16%              |
|          |              | 0.3          | -                               | 97.8%                  | -21%          | -24%              |
|          |              | 0.4          | -                               | 97.2%                  | -30%          | -33%              |
|          |              | 0.5          | -                               | 95.2%                  | -32%          | -41%              |
| CIFAR-10 | EN B0 FT     | 0.0          | 98.1%                           | -                      | -             | -                 |
|          |              | 0.1          | -                               | 98.0%                  | -5%           | -9%               |
|          |              | 0.2          | -                               | 97.7%                  | -12%          | -17%              |
|          |              | 0.3          | -                               | 97.2%                  | -20%          | -24%              |
|          |              | 0.4          | -                               | 96.5%                  | -27%          | -32%              |
|          |              | 0.5          | -                               | 95.0%                  | -37%          | -41%              |
| CIFAR-10 | EN B0 SC     | 0.0          | 91.8%                           | -                      | -             | -                 |
|          |              | 0.1          | -                               | 92.6%                  | -5%           | -9%               |
|          |              | 0.2          | -                               | 92.4%                  | -12%          | -17%              |
|          |              | 0.3          | -                               | 92.3%                  | -20%          | -24%              |
|          |              | 0.4          | -                               | 91.9%                  | -27%          | -32%              |
|          |              | 0.5          | -                               | 91.0%                  | -37%          | -41%              |
| CIFAR-100| EN B4 FT     | 0.0          | 89.8%                           | -                      | -             | -                 |
|          |              | 0.1          | -                               | 89.3%                  | -4%           | -6%               |
|          |              | 0.2          | -                               | 88.1%                  | -11%          | -14%              |
|          |              | 0.3          | -                               | 86.3%                  | -18%          | -23%              |
|          |              | 0.4          | -                               | 83.0%                  | -19%          | -33%              |
|          |              | 0.4          | -                               | 74.1%                  | -23%          | -42%              |
| CIFAR-100| EN B0 FT     | 0.0          | 87.8%                           | -                      | -             | -                 |
|          |              | 0.1          | -                               | 86.8%                  | -6%           | -9%               |
|          |              | 0.2          | -                               | 85.3%                  | -15%          | -17%              |
|          |              | 0.3          | -                               | 83.3%                  | -23%          | -24%              |
|          |              | 0.4          | -                               | 80.5%                  | -30%          | -32%              |
|          |              | 0.5          | -                               | 73.2%                  | -39%          | -41%              |
| CIFAR-100| EN B0 SC     | 0.0          | 70.1%                           | -                      | -             | -                 |
|          |              | 0.1          | -                               | 71.0%                  | -6%           | -9%               |
|          |              | 0.2          | -                               | 71.5%                  | -15%          | -17%              |
|          |              | 0.3          | -                               | 71.3%                  | -23%          | -24%              |
|          |              | 0.4          | -                               | 71.8%                  | -30%          | -32%              |
|          |              | 0.5          | -                               | 71.7%                  | -39%          | -41%              |

Hardware. Models with embedding dimension of 1024 were trained on 64-core TPUv3 clusters and 2048 or higher were trained on 256-core TPUv3 clusters.

Experiment results. Transformers are typically underparametrised for a dataset as rich and complex as LM1B. Consequently, any type of regularisation on the “smaller” Transformer models we trained (with embedding size 1024) leads to higher perplexity than a model trained with no dropout. We note however that SliceOut outperforms the other types of regularisation we experimented with (standard dropout and controlled dropout) for all dropout rates tested (0.1 to 0.3). As we increase the size of the embedding from 1024 to 2048, the perplexity of all models decreases significantly (down to ~ 28). However, the gap in perplexity between the models trained with SliceOut and without is negligible (~ 0.05 perplexity points), and we are able to preserve the same speedups and memory gains (~ 10%).
Table 17: Transformer results

| Width | Dropout rate | Dropout Perplexity | Controlled Perplexity | Sliceout Perplexity | Training time | Max cached memory |
|-------|--------------|-------------------|-----------------------|--------------------|---------------|------------------|
| 1024  | 0.0          | 31.65             | -                     | -                  | -             | -                |
|       | 0.1          | 32.98             | 33.01                 | 32.39              | 6%            | -1%              |
|       | 0.2          | 43.46             | 36.37                 | 32.88              | -3%           | -5%              |
|       | 0.3          | 45.10             | 45.74                 | 33.71              | -8%           | -9%              |
| 2048  | 0.0          | 28.07             | -                     | -                  | -             | -                |
|       | 0.3          | 88.59             | 53.14                 | 28.12              | -11%          | -10%             |

Perplexity vs No. of steps: 1024 embedding dim.

Perplexity vs No. of steps: 2048 embedding dim.