Memory-Efficient Implementation of DenseNets

Geoff Pleiss*
Cornell University
geoff@cs.cornell.edu

Danlu Chen*
Fudan University
dc688@cornell.edu

Gao Huang, Tongcheng Li
Cornell University
{gh349,tl486}@cornell.edu

Laurens van der Maaten
Facebook AI Research
lvdmaaten@fb.com

Kilian Q. Weinberger
Cornell University
kwq4@cornell.edu

Abstract

The DenseNet architecture [9] is highly computationally efficient as a result of feature reuse. However, a naïve DenseNet implementation can require a significant amount of GPU memory: If not properly managed, pre-activation batch normalization [7] and contiguous convolution operations can produce feature maps that grow quadratically with network depth. In this technical report, we introduce strategies to reduce the memory consumption of DenseNets during training. By strategically using shared memory allocations, we reduce the memory cost for storing feature maps from *quadratic* to *linear*. Without the GPU memory bottleneck, it is now possible to train extremely deep DenseNets. Networks with $14M$ parameters can be trained on a single GPU, up from $4M$. A 264-layer DenseNet ($73M$ parameters), which previously would have been infeasible to train, can now be trained on a single workstation with 8 NVIDIA Tesla M40 GPUs. On the ImageNet ILSVRC classification dataset, this large DenseNet obtains a state-of-the-art single-crop top-1 error of 20.26%.

1 Introduction

The DenseNet architecture [9] is highly efficient, both in terms of parameter use and computation time. On the ImageNet ILSVRC-2012 dataset [13], a 201-layer DenseNet achieves roughly the same top-1 classification error as a 101-layer Residual Network (ResNet) [6], while using half as many parameters ($20M$ vs $44M$) and half as many floating point operations ($80B$/image vs $155B$/image). Each DenseNet layer is explicitly connected to all previous layers within a pooling region, rather than only receiving information from the most recent layer. These connections promote feature reuse, as early-layer features can be utilized by all other layers. Because features accumulate, the final classification layer has access to a large and diverse feature representation.

This inherent efficiency makes the DenseNet architecture a prime candidate for very-high capacity networks. Huang et al. [9] report that a 161-layer DenseNet (with $k = 48$ features per layer and $29M$ parameters) achieves a top-1 single-crop error of 22.2% on the ImageNet ILSVRC classification dataset. It is reasonable to expect that larger networks would perform even better. However, with most existing DenseNet implementations, model size is currently limited by GPU memory.

Each layer only produces k feature maps (where k is small – typically between 12 and 48), but uses all previous feature maps as input. This causes the number of *parameters* to grow quadratically with network depth. It is important to note that this quadratic dependency of parameters w.r.t. depth is *not* an issue, as networks with more parameters perform better and in that respect DenseNets are more competitive than alternative architectures, such as e.g. ResNets. Most naïve implementations of DenseNets do however also have a quadratic memory dependency with respect to *feature maps*. This growth is responsible for the vast majority of the memory consumption, and as we argue in this report, it is implementation issue and not an inherent aspect of the DenseNet architecture.

*Authors contributed equally.

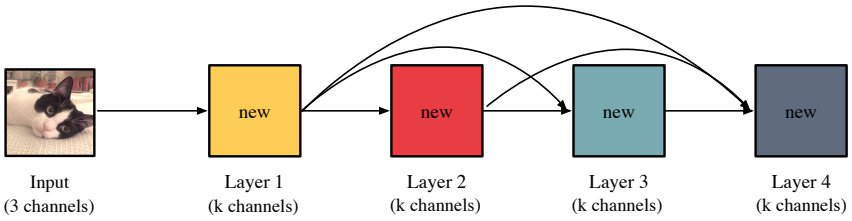

Figure 1: High-level illustration of the DenseNet architecture.

The quadratic memory dependency w.r.t. feature maps originates from intermediate feature maps generated in each layer, which are the outputs of batch normalization and concatenation operations. Intermediate feature maps are utilized both during the forward pass (to compute the next features) and during back-propagation (to compute gradients). If they are not properly managed, and are naïvely stored in memory, training large models can be expensive – if not infeasible.

In this report, we introduce a strategy to substantially reduce the training-time memory cost of DenseNet implementations, with a minor reduction in speed. Our primary observation is that *the intermediate feature maps responsible for most of the memory consumption are relatively cheap to compute*. This allows us to introduce Shared Memory Allocations, which are used by all layers to store intermediate results. Subsequent layers overwrite the intermediate results of previous layers, but their values can be re-populated during the backward pass at minimal cost. Doing so reduces feature map memory consumption *from quadratic to linear*, while only adding $15 - 20\%$ additional training time. This memory savings makes it possible to train *extremely large* DenseNets on a reasonable GPU budget. In particular, we are able to extend the largest DenseNet from 161 layers ($k = 48$ features per layer, $20M$ parameters) to 264 layers ($k = 48$, $73M$ parameters). On ImageNet, this model achieves a single-crop top-1 error of 20.26%, which (to the best of our knowledge) is state-of-the-art.

Our memory-efficient strategy is relatively straightforward to implement in existing deep learning frameworks. We offer implementations in Torch[2] [5], PyTorch[3] [1], MxNet[4] [2], and Caffe[5] [11].

2 The DenseNet Architecture

At a high-level, a DenseNet explicitly connects all layers with matching feature size. Huang et al. refer to a block of directly connected layers as a *dense block*, which is typically followed by a pooling layer (which reduces feature map sizes) or the final classifier. Layers in traditional neural networks only use the most recent features; in a DenseNet, a layer has access to all preceding features within its dense block. Mathematically, $\mathbf{x}_\ell$ – the features produced by layer ℓ – are computed as

$$\mathbf{x}_\ell = H_\ell\left([\mathbf{x}_1, \ldots, \mathbf{x}_{\ell-1}]\right),$$

where H_ℓ denotes the operations of layer ℓ and $[\cdot]$ represents the concatenation operator. Huang et al. [9] define H_ℓ to be a composite of three operations: batch normalization [10], a rectified linear unit (ReLU), and convolution – in that order. (H_ℓ may also include additional operations, such as a "bottleneck"; see [9].) In most deep learning frameworks, each of these three operations produce intermediate feature maps.

Given a dense block with m layers, the input to its final layer is $[\mathbf{x}_1, \ldots, \mathbf{x}_{m-1}]$ – all previous convolutional features. Because the number of convolutional features grows linearly with network depth, storing these in memory does not impose a significant memory burden. However, if the network stores the intermediate feature maps as well (e.g. the batch normalization output), GPU memory may become a limited resource. This is due to the fact that the intermediate features are computed for each input feature map, thus incurring $O(m^2)$ memory usage *if they are stored*. Many deep learning frameworks keep these intermediate feature maps allocated in GPU memory for use during back-propagation. The gradients of convolutional features (and parameters) typically are functions of the intermediate outputs, and therefore the intermediate outputs must remain accessible

[2] Torch implementation: `https://github.com/liuzhuang13/DenseNet/tree/master/models`
[3] PyTorch implementation: `https://github.com/gpleiss/efficient_densenet_pytorch`
[4] MxNet implementation: `https://github.com/taineleau/efficient_densenet_mxnet`
[5] Caffe implementation: `https://github.com/Tongcheng/DN_CaffeScript`

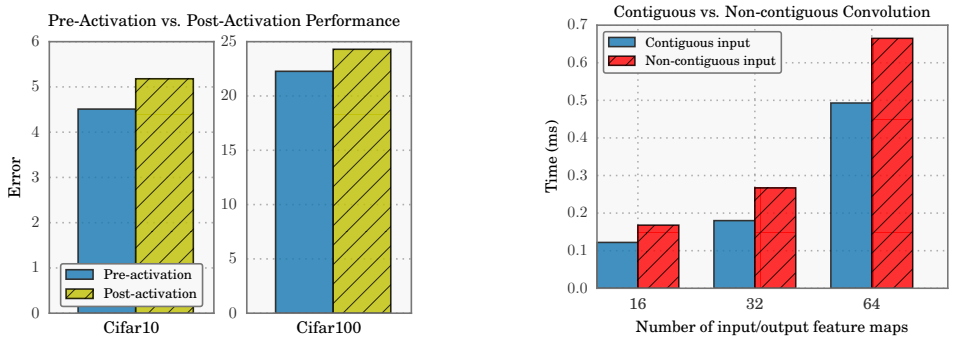

Figure 2: **Left:** Comparison of pre-activation vs post-activation DenseNet architectures (100-layer DenseNet-BC, $k = 12$). Pre-activations offer significant error reductions, but generate a quadratic number of outputs. **Right:** Speed of contiguous vs non-contiguous convolution operations (measured on a NVIDIA GTX 980). Each operation applies 3-by-3 filters to 32×32 feature maps (minibatch size of 64). Contiguous operations are preferred, but generate multiple copies of each feature.

for the backward pass. In DenseNets, there are two operations which are responsible for this quadratic growth: pre-activation batch normalization and contiguous concatenation.

Pre-activation batch normalization. The DenseNet architecture, as described in [9], utilizes *pre-activation* batch normalization [7]. Unlike conventional architectures, pre-activation networks apply batch normalization and non-linearities *before* the convolution operation rather than after. Though this might seem like a minor change, it makes a big difference in DenseNet performance. Batch normalization applies a scaling and a bias to the input features. If each layer applies its own batch normalization operation, then each layer applies a *unique* scale and bias to previous features. For example, the Layer 2 batch normalization might scale a Layer 1 feature by a positive constant, while Layer 3 might scale the same feature by a negative constant. After the non-linearity, Layer 2 and Layer 3 extract opposite information from Layer 1. This would not be possible if all layers shared the same batch normalization operation, or if normalization occurred after convolution operations. Without pre-activation, the CIFAR-10 error of a 100-layer DenseNet-BC grows from 4.51 to 5.18. On CIFAR-100, the error grows from 22.27 to 24.30 (Figure 2 left).

Given a DenseNet with m layers, pre-activations generate up to m normalized copies of each layer. Because each copy has different scaling and bias, naïve implementations in standard deep learning frameworks typically allocate memory for each of these $(m-1)(m-2)/2$ duplicated feature maps.

Contiguous concatenation. Convolution operations are most efficient when all input data lies in a *contiguous block of memory*. Some deep learning frameworks, such as Torch, explicitly require that all convolution operations are contiguous. While CUDNN, the most common library for low-level convolution routines [4], does not have this requirement, using non-contiguous blocks of memory adds a computation time overhead of $30-50\%$ (Figure 2 right).

To make a contiguous input, each layer must copy all previous features into a contiguous memory block. Given a network with m layers, each feature may be copied up to m times. If these copies are stored separately, the concatenation operation would also incur quadratic memory cost.

It is worth noting that we cannot simply assign filter outputs to a pre-allocated contiguous block of memory. Features are represented as tensors in $\mathbb{R}^{n \times d \times w \times h}$ (or $\mathbb{R}^{n \times w \times h \times d}$), where n is the number of mini-batch samples, d is the number of feature maps, and w and h are the width and height. GPU convolution routines, such as from the CUDNN library, assume that feature data is stored with the minibatch as the outer dimension. Assigning two features next to each other in a contiguous block therefore concatenates along the minibatch dimension, and not the intended feature map dimension.

3 Naïve Implementation

In modern deep learning libraries, layer operations are typically represented by edges in a computation graph. Computation graph nodes represent intermediate feature maps stored in memory. We show the computation graph for a naïve implementation of a DenseNet layer (without bottleneck operations) in

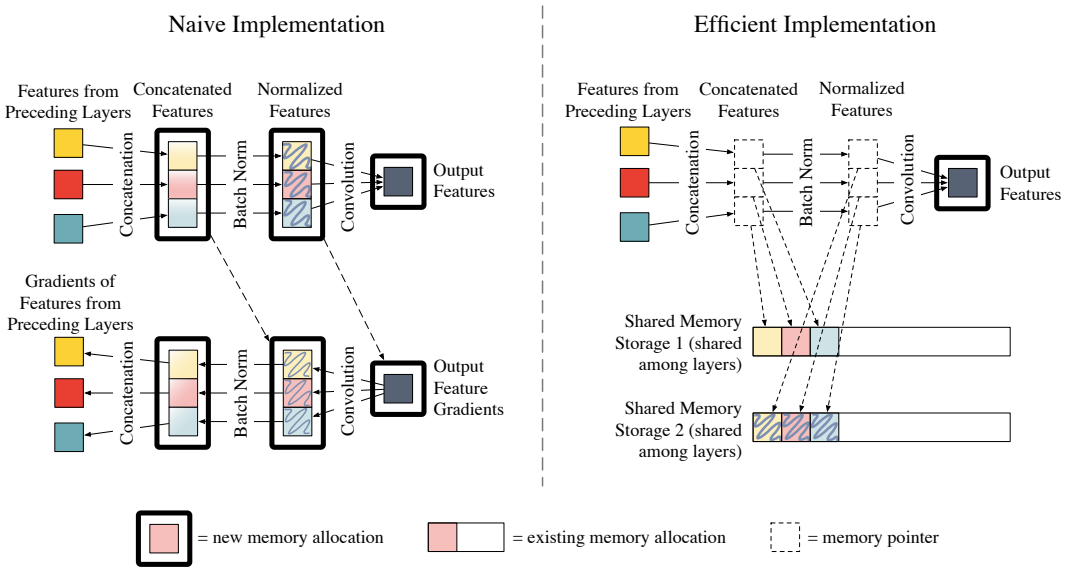

Figure 3: DenseNet layer forward pass: original implementation (**left**) and efficient implementation (**right**). Solid boxes correspond to tensors allocated in memory, where as translucent boxes are pointers. Solid arrows represent computation, and dotted arrows represent memory pointers. The efficient implementation stores the output of the concatenation, batch normalization, and ReLU layers in temporary storage buffers, whereas the original implementation allocates new memory.

Figure 3 (top left).[6] The inputs to each layer are features from previous layers (the colored boxes). As these features originate from different layers, they are not stored contiguously in memory. The first operation of the naïve implementation therefore *copies* each of these features and concatenates them into a contiguous block of memory (center left). If each of the ℓ previous layers produces k features, the contiguous block of memory must accommodate $\ell \times k$ feature maps. These concatenated features are input to the batch normalization operation (center right), which similarly allocates a new $\ell \times k$ contiguous block of memory. (The ReLU non-linearity occurs in-place in memory, and therefore we choose to exclude it from the computation graph for simplicity.) Finally, a convolution operation (far right) generates k new features from the batch normalization output. From Figure 3 (top left), it is easy to visualize the quadratic growth of memory. The two intermediate operations require a memory block which can fit all $O(\ell k)$ previously computed features. By comparison, the output features require only a constant $O(k)$ memory per layer.

In some deep learning frameworks, such as LuaTorch, even more memory may be allocated during back-propagation. Figure 3 (bottom left) displays a computation graph for gradients. The output feature gradients (far right) and the normalized feature maps from the forward pass (dotted line) are used to compute the batch normalization gradients (center right). Storing these gradients requires an additional $\ell \times k$ feature maps worth of memory allocation. Similarly, a $\ell \times k$ memory allocation is necessary to store the concatenated feature gradients.

4 Memory-Efficient Implementation

To circumvent this issue we exploit that concatenation and normalization operations are computationally extremely cheap. We propose two pre-allocated *Shared Memory Storage* locations to avoid the quadratic memory growth. During the forward pass, we assign all intermediate outputs to these memory blocks. During back-propagation, we recompute the concatenated and normalized features on-the-fly as needed.

This recomputation strategy has previously been explored on other neural network architectures. Chen et al. [3] exploit recomputation to train $1,000$-layer ResNets on a single GPU. Additionally,

[6] The computation graph is based roughly on the original implementation: `https://github.com/liuzhuang13/DenseNets/`.

Chen et al. [2] have developed recomputation support for arbitrary networks in the MxNet deep learning framework. In general, recomputing intermediate outputs necessarily trades-off memory for computation. However, we have discovered that this strategy is very practical for DenseNets. The concatenation and batch normalization operations are responsible for most memory usage, yet only incur a small fraction of the overall computation time. Therefore, recomputing these intermediate outputs yields substantial memory savings with little computation time overhead.

Shared storage for concatenation. The first operation – feature concatenation – requires $O(\ell k)$ memory storage, where ℓ is the number of previous layers and k is the number of features added each layer. Rather than allocating memory for each concatenation operation, we assign the outputs to a memory allocation shared across all layers ("Shared Memory Storage 1" in Figure 3 right). The concatenation feature maps (center left) are therefore pointers to this shared memory. Copying to pre-allocated memory is significantly faster than allocating new memory, so this concatenation operation is extremely efficient. The batch normalization operation, which takes these concatenated features as input, reads directly from Shared Memory Storage 1. Because Shared Memory Storage 1 is used by all network layers, its data is not permanent. When the concatenated features are needed during back-propagation, we assume that they can be recomputed efficiently.

Shared storage for batch normalization. Similarly, we assign the outputs of batch normalization (which also requires $O(m)$ memory) to a shared memory allocation ("Shared Memory Storage 2"). The convolution operation reads from pointers to this shared storage (center right). As with concatenation, we must recompute the batch normalization outputs during back-propagation, as the data in Shared Memory Storage 2 is not permanent and will be overwritten by the next layer. However, batch normalization consists of scalar multiplication and addition operations, which are very efficient in comparison with convolution math. Computing Batch Normalized features accounts for roughly 5% of one forward-backward pass, and therefore is not a costly operation to repeat.

Shared storage for gradients. The concatenation, batch normalization, and convolution operations each produce gradients tensors during back-propagation. In LuaTorch, it is straight forward to ensure that this tensor data is stored in a single shared memory allocation. This strategy prevents gradient storage from growing quadratically. We use a memory-sharing scheme based on of Facebook's ResNet implementation.[7] The PyTorch and MxNet frameworks share gradient storage out-of-the-box.

Putting these pieces together, the forward pass (Figure 3 right) is similar to the naïve implementation (Figure 3 top left), with the exception that intermediate feature maps are stored in Shared Memory Storage 1 or Shared Memory Storage 2. The backward pass requires one additional step: we first re-compute the concatenation and batch normalization operations (center left and center right) in order to re-populate the shared memory storage with the appropriate feature map data. Once the shared memory storage contains the correct data, we can perform regular back-propagation to compute gradients. In total, this DenseNet implementation only allocates memory for the output features (far right), which are constant in size. Its overall memory consumption for feature maps is *linear* in network depth.

5 Results

We compare the memory consumption and computation time of three DenseNet implementations during training. The **naïve implementation** is based on the original LuaTorch implementation[8] of Huang et al. [9]. As described in Section 3, this implementation allocates memory for the concatenation and batch normalization outputs. Additionally, each of these operations uses additional memory to store gradients of the intermediate features. Therefore, this implementation has four operations with quadratic memory growth (two forward-pass operations and two backward-pass operations).

We then test two memory-efficient implementations of DenseNets. The first implementation **shares gradient storage** so that there are no new memory allocations during back-propagation. All gradients are instead assigned to shared memory storage. The LuaTorch gradient memory-sharing code is based on Facebook's ResNet implementation[9]. PyTorch automatically performs this optimization

[7] `https://github.com/facebook/fb.resnet.torch`

[8] `https://github.com/liuzhuang13/DenseNet`

[9] `https://github.com/facebook/fb.resnet.torch`

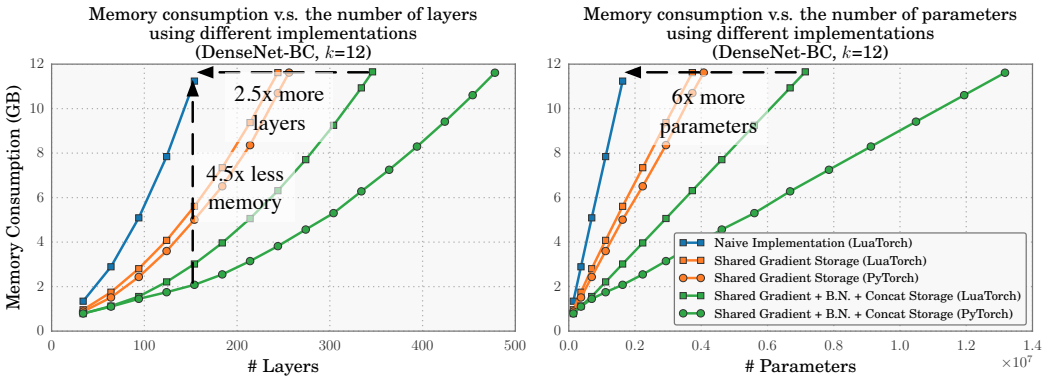

Figure 4: GPU memory consumption as a function of network depth. Each model is a bottlenecked Densenet (Densenet-BC) with $k = 12$ features added per layer. The efficient implementations enables training much deeper models with less memory.

out-of-the-box. The second implementation includes all optimizations described in Section 4: shared storage for batch normalization, concatenation, and gradient operations (**shared gradient + B.N. + concat storage**). The concatenated and normalized feature maps are recomputed as necessary during back-propagation. The only tensors which are stored in memory during training are the convolution feature maps and the parameters of the network.

Memory consumption of the three implementations (in LuaTorch and PyTorch) is shown in Figure 4. (There is no PyTorch naïve implementation because PyTorch automatically shares gradient storage.) With four quadratic operations, the naïve implementation becomes memory intensive very quickly. The memory usage of a 160 layer network ($k = 12$ features per layer, $1.8M$ parameters) is roughly 10 times as much as a 40 layer network ($k = 12$, $160K$ parameters). Training a larger network with more than 160 layers requires over 12 GB of memory, which is more than a typical single GPU. While sharing gradients reduces some of this memory cost, memory consumption still grows rapidly with depth. On the other hand, using all the memory-sharing operations significantly reduces memory consumption. In LuaTorch, the 160-layer model uses 22% of the memory required by the Naïve Implementation. Under the same memory budget (12 GB), it is possible to train a 340-layer model, which is $2.5\times$ as deep and has $6\times$ as many parameters as the best naïve implementation model.

It is worth noting that the *total* memory consumption of the most efficient implementation does not grow linearly with depth, as the number of parameters is inherently a quadratic function of the network depth. This is a function of the architectural design and part of the reason why DenseNets are so efficient. The memory required to store the parameters is far less than the memory consumed by the feature maps and the remaining quadratic term does not impede model depth.

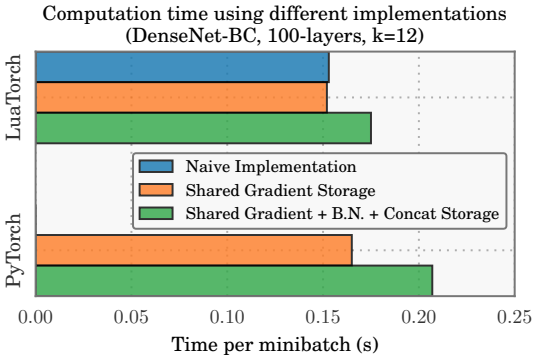

Figure 5: Computation time (measured on a NVIDIA Maxwell Titan-X).

Finally, we see in Figure 4 that PyTorch is more memory efficient than LuaTorch. Using the efficient PyTorch implementation, we can train DenseNets with nearly 500 layers ($13M$ parameters) on a single GPU. The "autograd" library in PyTorch performs memory optimizations during training, which likely contribute to this implementation's efficiency.

Training time is not significantly affected by the memory optimizations. In Figure 5 we plot the time per minibatch of a 100-layer DenseNet-BC ($k = 12$) on a NVIDIA Maxwell Titan-X. Shared gradient storage does not incur any time cost. Sharing batch normalization and concatenation storage adds roughly 15% time overhead on LuaTorch, and 20% on PyTorch. This extra cost is

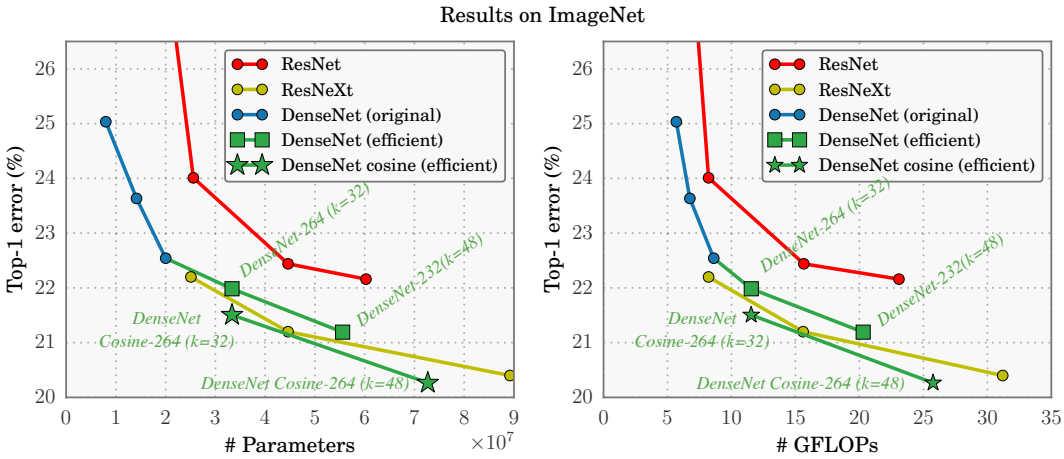

Figure 6: Top-1 classification error on ImageNet. The DenseNet models were trained on 8 NVIDIA Tesla M40 GPUs. Results in stars were not possible to train without the efficient implementation.

a result of recomputing operations during back-propagation. For DenseNets of any size, it makes sense to share gradient storage. If GPU memory is limited, the time overhead from sharing batch normalization/concatenation storage constitutes a reasonable trade-off.

ImageNet results. We test the new memory efficient LuaTorch implementation (shared memory storage for gradients, batch normalization, and concatenation) on the ImageNet IVLSRC classification dataset [13]. The deepest model trained by Huang et al. [9] using the original LuaTorch implementation was 161 layers ($k = 48$ features per layer, $29M$ parameters). With the efficient LuaTorch implementation however, it is possible to train a 264-layer DenseNet ($k = 48$, $73M$ parameters) using 8 NVIDIA Tesla M40 GPUs.

We train two new **DenseNet** models with the efficient implementation, one with 264 layers ($k = 32$, $33M$ parameters) and one with 232 layers ($k = 48$, $55M$ parameters).[10] These models are trained for 100 epochs following the same procedure described in [9]. Additionally, we train two DenseNets models with a cosine learning-rate schedule (**DenseNet cosine**), similar to what was used by [12] and [8]. The model is trained for 100 epochs, with the learning rate of epoch t set to $0.05 \cos(t(\pi/100)) + 1$. Intuitively, this learning rate schedule starts off with a large learning rate, and quickly (but smoothly) anneals the learning rate to a small value. Both of the cosine DenseNets have 264 layers – one with $k = 32$ ($33M$ parameters) and one with $k = 48$ ($73M$ parameters). Given a fixed GPU budget, these DenseNet models can only be trained with the efficient implementation.

We display the top-1 error performance of these DenseNets in Figure 6. The models trained with the efficient implementation are denoted as green points (squares for standard learning rate schedule, stars for cosine). We compare the performance of these models to shallower DenseNets trained with the original implementation. Additionally, we compare against **ResNet** models introduced in [6] and **ResNeXt** models introduced in [14]. Results were obtained with single centered test-image crops.

The new DenseNet models (standard training procedure) achieve nearly 1 percentage point better error than the deepest ResNet model, while using fewer parameters. Additionally, the deepest cosine DenseNet achieves a top-1 error of 20.26%, which outperforms the previous state-of-the-art model [14]. It is clear from these results that DenseNet performance continues to improve measurably as more layers are added.

6 Conclusion

In this report, we describe a new implementation strategy for DenseNets. Previous DenseNet implementations store all intermediate feature maps during training, causing feature map memory usage to grow quadratically with depth. By employing a shared memory buffer and recomputing

[10] There are four dense blocks in this model. In the 264-layer model, the dense blocks have 6, 32, 64, and 48 layers respectively. In the 232-layer model, the dense blocks have 6, 32, 48, and 48 layers.

some cheap transformations, models utilize significantly less memory, with only a small increase in computation time. With this new implementation strategy, memory no longer impedes training extremely deep DenseNets. As a result, we are able to double the depth of prior models, which results in a measurable drop in top-1 error on the ImageNet dataset.

Acknowledgments

The authors are supported in part by the III-1618134, III-1526012, and IIS-1149882 grants from the National Science Foundation, the Office of Naval Research DOD (N00014-17-1-2175), and the Bill and Melinda Gates Foundation.

References

[1] Pytorch. `https://github.com/pytorch`. Accessed: 2017-06-09.

[2] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[3] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[4] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[5] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[6] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[7] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *ECCV*, 2016.

[8] G. Huang, Y. Li, G. Pleiss, Z. Liu, J. E. Hopcroft, and K. Q. Weinberger. Snapshot ensembles: Train 1, get m for free. In *ICLR*, 2017.

[9] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten. Densely connected convolutional networks. In *CVPR*, 2017.

[10] S. Ioffe and C. Szegedy. batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

[11] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[12] I. Loshchilov and F. Hutter. Sgdr: stochastic gradient descent with restarts. In *ICLR*, 2017.

[13] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[14] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. In *CVPR*, 2017.