Convolutional Networks with Dense Connectivity

Gao Huang, Zhuang Liu, Geoff Pleiss, Laurens van der Maaten and Kilian Q. Weinberger

Abstract—Recent work has shown that convolutional networks can be substantially deeper, more accurate, and efficient to train if they contain shorter connections between layers close to the input and those close to the output. In this paper, we embrace this observation and introduce the Dense Convolutional Network (DenseNet), which connects each layer to every other layer in a feed-forward fashion. Whereas traditional convolutional networks with L layers have L connections—one between each layer and its subsequent layer—our network has $\frac{L(L+1)}{2}$ direct connections. For each layer, the feature-maps of all preceding layers are used as inputs, and its own feature-maps are used as inputs into all subsequent layers. DenseNets have several compelling advantages: they alleviate the vanishing-gradient problem, encourage feature reuse and substantially improve parameter efficiency. We evaluate our proposed architecture on four highly competitive object recognition benchmark tasks (CIFAR-10, CIFAR-100, SVHN, and ImageNet). DenseNets obtain significant improvements over the state-of-the-art on most of them, whilst requiring less parameters and computation to achieve high performance.

Index Terms—Convolutional neural network, deep learning, image classification

1 INTRODUCTION

CONVOLUTIONAL neural networks (CNNs) have become the dominant machine learning approach for visual object recognition. Although they were originally introduced over 20 years ago [1], improvements in computer hardware and network structure have enabled the training of truly deep CNNs only recently. The original LeNet5 [2] consisted of 5 layers, VGG featured 19 [3], and thanks to the skip/shortcut connections, Highway Networks [4] and Residual Networks (ResNets) [5] have surpassed the 100-layer barrier.

As CNNs become increasingly deep, a new research problem emerges: information about the input or gradient that passes through many layers it can vanish and "wash out" by the time it reaches the end (or beginning) of the network. Many recent publications address this problem. For example, Rectified Linear Unites (ReLU) [6] avoid gradient saturation, batch-normalization [7] reduces covariate shift across layers by re-scaling the outputs of its previous layer. ResNets [5] and Highway Networks [4] bypass signal from one layer to the next via identity connections. Stochastic depth [8] shortens ResNets by randomly dropping layers during training to allow better information and gradient flow. FractalNets [9] repeatedly combine several parallel layer sequences with different number of convolutional blocks to obtain a large nominal depth, while maintaining many short paths in the network. Although these different approaches vary in network topology and training procedure, they all share a key characteristic: they create short paths from early layers to later layers.

In this paper, we propose an architecture that distills this insight into a simple connectivity pattern: to ensure maximum information flow between layers in the network, we connect

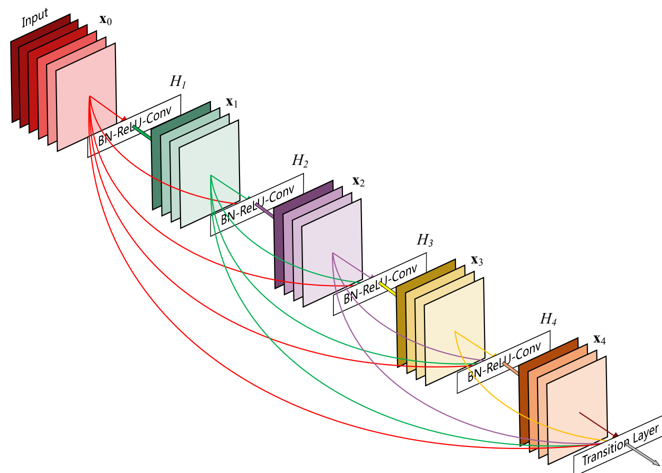

Fig. 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

all layers (with matching feature-map sizes) directly with each other. To preserve the feed-forward nature, each layer obtains additional inputs from all preceding layers and passes on its own feature-maps to all subsequent layers. Fig. 1 illustrates this layout schematically. Crucially, in contrast to ResNets, we never combine features through summation before they are passed into a layer; instead, we combine features through concatenations. Hence, the ℓth layer has ℓ inputs, consisting of the feature-maps of all preceding convolutional blocks. Its own feature-maps are passed on to all $L - \ell$ subsequent layers. This introduces $\frac{L(L+1)}{2}$ connections in an L-layer network, instead of just L, as in traditional architectures. Because of its dense connectivity pattern, we refer to our approach as *Dense Convolutional Network (DenseNet)*.

A possibly counter-intuitive effect of this dense connectivity pattern is that it requires *fewer* parameters than traditional convolutional networks, as there is no need to re-learn redundant feature maps. Traditional feed-forward architectures can be viewed as algorithms with a state, which is passed on from layer to layer.

- *Gao Huang is with the Department of Automation, Tsinghua University, Beijing, 100084. (E-mail: gaohuang@tsinghua.edu.cn).*
- *Geoff Pleiss and Kilian Q. Weinberger are with the Department of Computer Science, Cornell University, Ithaca, NY, 14850. (E-mail: kwq4@cornell.edu, geoff@cs.cornell.edu).*
- *Zhuang Liu is with Berkeley Artificial Intelligence Research, UC Berkeley, Berkeley, CA 94704. (E-mail: zhuangl@berkeley.edu).*
- *Laurens van der Maaten is with Facebook AI Research. (E-mail: lvdmaaten@fb.com).*

©2019 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1109/TPAMI.2019.2918284

Each layer reads the state from its preceding layer and writes to the subsequent layer. It changes the state but also passes on information that needs to be preserved. ResNets [5] make this information preservation explicit through additive identity transformations. Recent variations of ResNets [8] show that many layers contribute very little and can in fact be randomly dropped during training. This makes the state of ResNets similar to (unrolled) recurrent neural networks [10], but the number of parameters of ResNets is substantially larger because each layer has its own weights. Our proposed DenseNet architecture explicitly differentiates between information that is *added* to the network and information that is *preserved*. DenseNet layers are very narrow (*e.g.*, 12 feature-maps per layer), adding only a small set of feature-maps to the "collective knowledge" of the network and keeping the remaining feature-maps unchanged—enables the final classifier to base its decision on all feature-maps in the network.

Besides better parameter efficiency, another big advantage of DenseNets is that they are easier to train, due to their improved information flow and gradients throughout the network. Each layer has direct access to the gradients from the loss function and the original input signal, facilitating an implicit form of deep supervision [11]. Finally, dense connections create many short paths in the network, which have a strong regularizing effect and reduce overfitting on smaller training sets.

We evaluate DenseNets on four highly competitive benchmark datasets (CIFAR-10, CIFAR-100, SVHN, and ImageNet). On these, our models exhibit their superior parameter efficiency, and the benefits thereof, in two ways: 1. they tend to require far fewer parameters when compared against alternative algorithms with comparable accuracy; 2. they outperform the current state-of-the-art results on most of the benchmark tasks as the number of model parameters is increased.

The main results of this paper were published originally in its conference version[1]. However, this longer article provides a more comprehensive analysis and a deeper understanding of the novel DenseNet architecture, *e.g.* hyper-parameters (Section 5) and design choices(Section 6). In addition, we also provide detailed instructions on how to implement the model in a memory efficient way (Section 3.2 and 4.6).

2 Related Work

The exploration of network architectures has been an integral part of neural network research since their initial discovery. The recent resurgence in popularity of neural networks has also revived this research domain. The increasing number of layers in modern networks amplifies the differences between architecture types and motivates the exploration of different connectivity patterns as well as the revisiting of old research ideas.

A cascade structure similar to our proposed dense connectivity has already been studied in the neural networks literature in the 1980s [12]. Their pioneering work focuses on fully connected multi-layer perceptrons trained in a layer-by-layer fashion. More recently, Wilamowski and Yu [13] proposed fully connected cascade networks to be trained with batch gradient descent. Although effective on small datasets, this approach only scales to networks with a few hundred parameters. Several recent publications [14,15,16,17], have found the utilization of multi-level features in CNNs through skip-connections effective for various vision tasks. Parallel to our work, [18] derived a purely theoretical framework for networks with cross-layer connections similar to ours.

Highway Networks [4], with their gating units and bypassing paths, were amongst the first architectures that provided a means to effectively train end-to-end networks with more than 100 layers. The bypassing paths are presumed to be the key factor that eases the training of these very deep networks, which is further supported by ResNets [5], in which pure identity mappings are used as bypassing paths. ResNets have achieved impressive, record-breaking performance on many challenging image recognition, localization, and detection tasks, such as ImageNet and COCO object detection [5]. Recently, *stochastic depth* was proposed as a way to successfully train a 1202-layer ResNet [8]. Stochastic depth improves the training of deep residual networks by dropping layers randomly during training. This shows that not all layers may be needed and highlights that there is a great amount of redundancy in deep (residual) networks. Our paper was partly inspired by that observation.

An orthogonal approach to making networks deeper (*e.g.*, with the help of skip connections) is to increase the network *width*. The GoogLeNet [19,20] uses an "Inception module" which concatenates feature maps produced by filters of different sizes. In [21], a variant of ResNets with wide generalized residual blocks was proposed. In fact, simply increasing the number of filters in each layer of ResNets can improve its performance provided the depth is sufficient [22]. FractalNets also achieve competitive results on several benchmark datasets using a wide network structure [9].

Instead of drawing representational power from extremely deep or wide architectures, DenseNets exploit the potential of the network through *feature reuse*, yielding condensed models that are easy to train and highly parameter-efficient. Concatenating feature maps learned by *different layers* increases variation in the input of subsequent layers and improves efficiency. This constitutes a major difference between DenseNets and ResNets. Compared to Inception networks [19,20], which also concatenate features from different layers, DenseNets are simpler and more efficient.

There are other notable network architecture innovations which have yielded competitive results. The Network in Network (NIN) [23] structure includes micro multi-layer perceptrons into the filters of convolutional layers to extract more complicated features. In Deeply Supervised Network (DSN) [11], internal layers are directly supervised by auxiliary classifiers, which can strengthen the gradients received by earlier layers. Ladder Networks [24,25] introduce lateral connections into autoencoders, producing impressive accuracies on semi-supervised learning tasks. In [26], Deeply-Fused Nets (DFNs) were proposed to improve information flow by combining intermediate feature representations of different base networks.

Since the original publication of the conference version of this paper, many extensions of DenseNet have been proposed. These include the Dual Path Network [27], which is a hybrid network architecture combining the DenseNet and ResNet; the autoencoder DenseNet [28] proposed for semantic segmentation tasks; and 3D-DenseNet used for volume data segmentation [29]. Recently, a number of novel network building modules have been proposed, e.g., group convolution [30], learned group convolution [31], depth-separable convolution [32] and squeeze and excitation [33], which are orthogonal innovations to our work and could potentially or have already been shown to be helpful for further improving DenseNet.

1. http://openaccess.thecvf.com/content_cvpr_2017/papers/Huang_Densely_Connected_Convolutional_CVPR_2017_paper.pdf

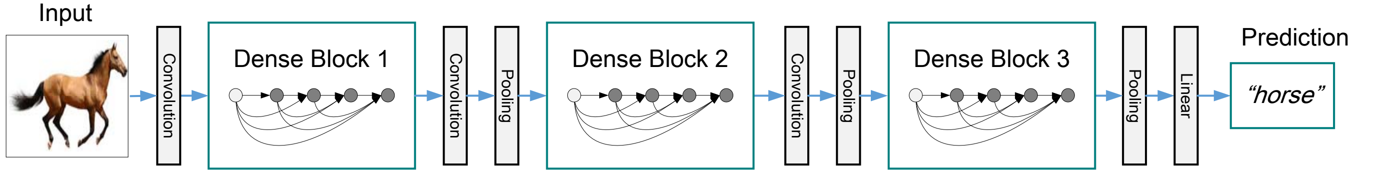

Fig. 2: A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature map sizes via convolution and pooling.

3 DENSENETS

In this section, we first describe the basic design methodology of DenseNet [2], then we discuss how to implement properly for memory efficient training.

3.1 The DenseNet Architecture

Consider a single image $\mathbf{x}_0$ that is passed through a convolutional network. The network comprises L layers, each of which implements a non-linear transformation $H_\ell(\cdot)$, where ℓ indexes the layer. $H_\ell(\cdot)$ can be a composite function of operations such as Batch Normalization (BN) [7], rectified linear units (ReLU) [6], Pooling [2], or Convolution (Conv). We denote the output of the ℓth layer as $\mathbf{x}_\ell$.

ResNets. Traditional convolutional feed-forward networks connect the output of the ℓth layer as input to the $(\ell+1)^{th}$ layer [30], which gives rise to the following layer transition: $\mathbf{x}_\ell = H_\ell(\mathbf{x}_{\ell-1})$. ResNets [5] add a skip-connection that bypasses the non-linear transformations with an identity function:

$$\mathbf{x}_\ell = H_\ell(\mathbf{x}_{\ell-1}) + \mathbf{x}_{\ell-1}. \quad (1)$$

An advantage of ResNets is that the gradient flows directly through the identity function from later layers to the earlier layers. However, the identity function and the output of H_ℓ are combined by summation, which may cancel out useful features due to the fact the one *cannot* exactly recover the two input features from the summation.

Dense connectivity. To further improve the information flow between layers we propose a different connectivity pattern: we introduce direct connections from any layer to all subsequent layers. Fig. 1 illustrates the layout of the resulting DenseNet schematically. Consequently, the ℓth layer receives the feature-maps of all preceding layers, $\mathbf{x}_0, \ldots, \mathbf{x}_{\ell-1}$, as input:

$$\mathbf{x}_\ell = H_\ell([\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{\ell-1}]), \quad (2)$$

where $[\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{\ell-1}]$ refers to the concatenation of the feature-maps produced in layers $0, \ldots, \ell-1$. Because of its dense connectivity we refer to this network architecture as *Dense Convolutional Network (DenseNet)*. The recursive concatenation may create massive redundant features in GPU memory during training if not implemented properly. We will discuss this issue in Section 3.2.

Composite function. Following [3,5,20], we define $H_\ell(\cdot)$ as a composite function of three types of operations: batch normalization (BN) [7], rectified linear unit (ReLU) [6] and convolution (Conv). Specifically, each $H_\ell(\cdot)$ corresponds to the sequence: BN-ReLU-Conv(1×1)-BN-ReLU-Conv(3×3). Here, the 1×1 convolution is introduced as a *bottleneck* layer to reduce the number of input feature-maps. This design choice is adopted by many other architectures to improve computational efficiency, and we find it especially effective for DenseNet. Unless otherwise specified, each bottleneck layer reduces the input to 4 times the number of feature-maps produced by the subsequent 3×3 convolutional layer. To avoid confusion, we call each H_ℓ a *basic layer*, to distinguish it from a single convolutional layer.

Pooling layers. The concatenation operation used in Eq. (2) is not viable when the size of feature-maps changes. However, an essential part of convolutional networks is pooling layers that change the size of feature-maps. To facilitate pooling in our architecture we divide the network into multiple densely connected *dense blocks*; see Fig. 2. We refer to layers between blocks as *transition layers*, which do convolution and pooling. The transition layers used in our experiments consist of a batch normalization layer and a 1×1 convolutional layer followed by a 2×2 average pooling layer. Note that the transition layers are much "wider" compared to other basic layers, and it is inefficient to use the expensive 3×3 convolution with stride 2 to perform down-sampling in a DenseNet.

Growth rate. If each function H_ℓ produces k feature-maps as output, it follows that the ℓth layer has $k\times(\ell-1)+k_0$ input feature-maps, where k_0 is the number of channels in the input of that dense block. To prevent the network from growing too wide and to improve the parameter efficiency we limit k to a small integer, *e.g.*, $k = 12$. We refer to the hyper-parameter k as the *growth rate* of the network. We show in Section 4 that a relatively small growth rate is sufficient to obtain state-of-the-art results on the datasets that we tested on. One explanation for this is that each layer has access to all the preceding feature-maps in its block and, therefore, to the network's "collective knowledge". One can view the feature-maps as the global state of the network. Each layer adds k feature-maps of its own to this state. The growth rate regulates how much new information each layer can contribute to the global state. The global state, once written, can be accessed from everywhere within the network and, unlike in traditional network architectures, there is no need to replicate it from layer to layer.

Compression. To further improve model compactness, we can reduce the number of feature-maps at transition layers. If a dense block contains m feature-maps, we let the following transition layer generate $\lfloor \theta m \rfloor$ output feature-maps, where $0 < \theta \leq 1$ is referred to as the compression factor. When $\theta = 1$, the number of feature-maps across transition layers remains unchanged. We set $\theta = 0.5$ in our experiments unless otherwise specified.

3.2 Implementation Details

DenseNet is a novel architecture that most deep-learning frameworks have not yet been optimized for. If implemented naively, deep-learning frameworks may copy feature maps in every concatenation, producing many redundant copies of the same feature maps. In turn, this may lead to prohibitively high memory

2. Note that unlike its conference version, this paper only presents the DenseNet-BC architecture, i.e., DenseNet with bottleneck layer and transition layer with compression. For notation simplicity, we use the term DenseNet to denote this architecture.

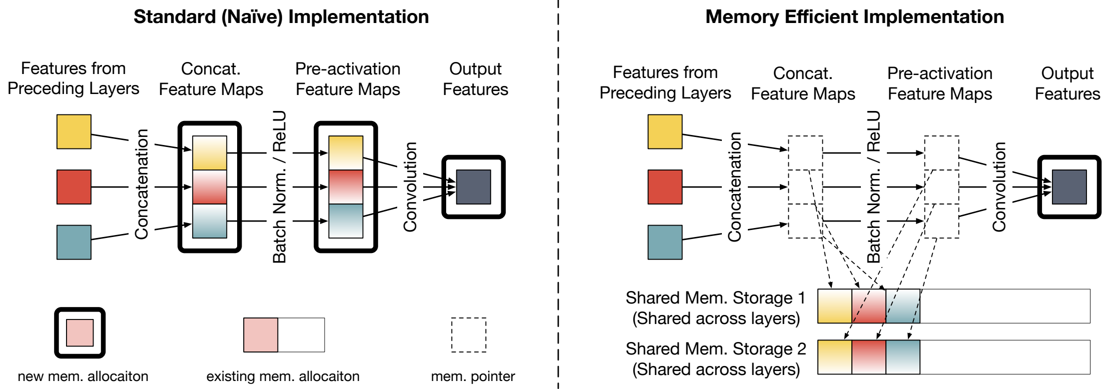

Fig. 3: DenseNet layer forward pass: original implementation (**left**) and efficient implementation (**right**). Solid boxes correspond to tensors allocated in memory, where as translucent boxes are pointers. Solid arrows represent computation, and dotted arrows represent memory pointers. The efficient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

consumption on GPU during training. In this subsection, we discuss how to implement DenseNet correctly, and demonstrate that good implementations of DenseNets are in fact very memory-efficient.

The DenseNet computation graph is illustrated in the left-hand side of Fig. 3 (left) for a simplified (pre-activation, no bottleneck) layer H_ℓ during training. This computation graph shows the components of the layer's composite function: 1) previous features are concatenated, 2) a pre-activation batch normalization/ReLU non-linearity are applied, and then 3) a convolution operation produces the next output feature. (The ReLU non-linearity is computed in-place, and therefore we combine it with batch normalization for simplicity.) We refer to the outputs of the concatenation and pre-activation batch normalization as the *intermediate feature maps* and the new convolutional feature as the *output feature*. Each basic layer H_ℓ produces k feature maps, and an M-layer dense block has $k \times M$ output features.

3.2.1 Memory-Efficient Implementation of DenseNets

During *inference*, an M-layer dense block stores $\mathcal{O}(M)$ output features in memory for use by the transition layer or final classifier. The intermediate feature maps are only used to compute their respective output feature and do not need to be stored. The GPU memory usage is therefore *linear* in the depth of the network. Although DenseNet needs to store the output features of multiple layers during inference, it still requires less memory during inference as its layers are very "narrow". For example, inferencing a ResNet-50 needs to store 256 feature maps of size 56×56 at each layer in its first stage; while the 121-layer DenseNet needs to store at most $64/2 + 6 \times 32 = 224$ feature maps of the same size [3].

During *training*, the network needs the intermediate feature maps not only for computing output features but also for computing parameter gradients. Most deep learning libraries will store all the intermediate feature maps in GPU memory until the forward and backward passes are complete. Obviously, if new space is allocated to store the concatenated features at each layer, the outputs of the lth layer have $M - l + 1$ copies in the memory, leading to a rapid growth in memory consumption.

3. This corresponds to the first dense block. Later dense blocks require less memory as their feature maps are of lower resolution, although the number of channels is larger.

To avoid this redundancy, we can pre-allocate a single memory buffer that will ultimately contain all the output feature maps of a dense block. The computation of a single layer involves reading the relevant feature maps from the shared memory buffer, computing the outputs of the layer, and storing those outputs in a consecutive part of the memory buffer. In tensor libraries that support operations on strided tensors (such as cudnn), all these operations can be performed in-place, which leads to a memory-efficient implementation of the feature-maps logic without requiring the complex memory management that is required to make other convolutional network architectures memory-efficient. We illustrate the memory pattern in Fig. 3. We implemented it in LuaTorch, and found it allows us to reduce the memory usage by more than $2\times$ compared to a naive implementation.

3.2.2 Further Reducing Memory Usage

Although sharing the memory of concatenated features could avoid saving redundant output features, the pre-activation batch normalization at each layer still needs store a normalized copy of all the previous output features. This also accounts for a quadratic memory consumption with respect to the network depth. Using post-activation batch normalization could circumvent this problem, while it generally leads to significant worse results (see Section 6.4).

Fortunately, the batch normalization layer (and the subsequent ReLU) is much cheaper to compute compared to convolution. Instead of storing all feature maps for the backward pass, one could *recompute* normalized feature maps on-the-fly when they are needed for gradient computations. Therefore, we only need to allocate a global memory that are shared by all the batch normalization layers (later BN layers simply override the outputs of earlier ones). This strategy can be generally applied to other architectures (as shown in [34]), while it is especially useful for DenseNet, as it allows us to train DenseNets with very small memory consumption. With this optimization, we are able to train three times larger models using the same amount of memory, while introducing little computation time overhead. Additional results are presented in Section 4.6.

3.3 Network Configurations

On the CIFAR and SVHN datasets, the DenseNets used in our experiments have 3 dense blocks that each have an equal numbers

TABLE 1: DenseNet architectures for ImageNet. The growth rate for all the networks is $k = 32$. Note that each "conv" layer shown in the table corresponds to the sequence BN-ReLU-Conv, except that the first"conv" layer with filter size 7×7 corresponds to Conv-BN-ReLU.

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-265
Convolution	112×112	7×7 conv, stride 2			
Pooling	56×56	3×3 max pool, stride 2			
Dense Block (1)	56×56	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer (1)	56×56	1×1 conv			
	28×28	2×2 average pool, stride 2			
Dense Block (2)	28×28	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer (2)	28×28	1×1 conv			
	14×14	2×2 average pool, stride 2			
Dense Block (3)	14×14	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 24$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 64$
Transition Layer (3)	14×14	1×1 conv			
	7×7	2×2 average pool, stride 2			
Dense Block (4)	7×7	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 16$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$
Classification Layer	1×1	7×7 global average pool			
		1000D fully-connected, softmax			

of layers. Before entering the first dense block, a 3×3 convolution with $2\times k$ filters is performed on the input images. By varying the number of basic layers (M) in each block, we can create models with different depth (L)[4]. It is easy to see that $L = 6 \times M + 4$. For convolutional layers with kernel size 3×3, each side of the inputs is zero-padded by one pixel to keep the feature-map size unchanged. We use 1×1 convolution followed by 2×2 average pooling as transition layers between two contiguous dense blocks. At the end of the last dense block, a global average pooling is performed and then a softmax classifier is attached. The feature-map sizes in the three dense blocks are $32\times$ 32, 16×16, and 8×8, respectively. We experimented with growth rate and layer configurations $\{L = 100, k = 12\}$, $\{L = 250, k = 24\}$ and $\{L=190, k=40\}$.

On ImageNet, we use a structure with 4 dense blocks on 224×224 input images. The initial convolution layer comprises $2\times k$ convolutions of size 7×7 with stride 2; the number of feature-maps in all other layers is a function of k. The exact network configurations we used on ImageNet are shown in Table 1.

4 Results

We empirically demonstrate the effectiveness of DenseNet on several benchmark datasets and compare it with state-of-the-art network architectures, especially with ResNet and its variants.

4.1 Datasets

CIFAR. The two CIFAR datasets [37] consist of colored natural scene images, with 32×32 pixels each. CIFAR-10 (C10) consists of images drawn from 10 and CIFAR-100 (C100) from 100 classes. The train and test sets contain 50,000 and 10,000 images respectively and we hold out 5,000 training images as a validation set. We adopt a standard data augmentation scheme that is widely used for this dataset [4,5,8,9,11,23,35,38]: the images are first zero-padded with 4 pixels on each side, then randomly cropped to again produce 32×32 images; half of the images are then horizontally mirrored. We denote this augmentation scheme by a "+" mark at the end of the dataset name (*e.g.*, C10+). For data preprocessing, we normalize the data using the channel means and standard deviations. We evaluate our algorithm on all four datasets: C10, C100, C10+, C100+. For the final run we use all 50,000 training images and report the final test error at the end of training.

SVHN. The Street View House Numbers (SVHN) dataset [39] contains 32×32 colored digit images coming from Google Street View. The task is to classify the central digit into the correct one of the 10 digit classes. There are 73,257 images in the training set, 26,032 images in the test set, and 531,131 images for additional training. Following common practice [8,11,23,40,41] we use all the training data without any data augmentation, and split a validation set with 6,000 images from the training set. We select the model with the lowest validation error during training and report the test error. We follow [22] and divide the pixel values by 255 so they are in the $[0, 1]$ range.

ImageNet. The ILSVRC 2012 classification dataset [42] consists 1.2 million images for training, and 50,000 for validation. Each image is associated with a label from $1,000$ predefined classes. We adopt the same data augmentation scheme for the training images as in [5,36,43], and apply a 224×224 center crop to images at test time. Following common practice [5,8,36], we report classification errors on the validation set.

4.2 Training

All the networks are trained using SGD. On CIFAR and SVHN we train using mini-batch size 64 for 300 and 40 epochs, respectively. The initial learning rate is set to 0.1, and is divided by 10 at 50% and 75% of the total number of training epochs. On ImageNet, we train models for 90 epochs[5] with a mini-batch size of 256. The learning rate is set 0.1 initially, and is lowered by a factor of 10 after epoch 30 and epoch 60.

Following [43], we use a weight decay of 10^{-4} and a Nesterov momentum [44] of 0.9 without dampening. We adopt the weight initialization introduced by [45]. For the three datasets without data augmentation, *i.e.*, C10, C100 and SVHN, we add a dropout layer [46] after each convolutional layer (except the first one) and

4. Following existing works, network depth corresponds to the number of layers with trainable weights, *e.g.*, convolutional layers and fully connected layers. However, batch normalization layers are not counted.

5. Training for more epochs on ImageNet generally leads to higher accuracy. But for a fair comparison with ResNets, we limit the training of DenseNets to 90 epochs.

TABLE 2: Error rates (%) on CIFAR and SVHN datasets. L denotes the network depth and k its growth rate. Results that surpass all competing methods are **bold** and the overall best results are **blue**. "+" indicates standard data augmentation (translation and/or mirroring). $*$ indicates results run by ourselves. All the results of DenseNets without data augmentation (C10, C100, SVHN) are obtained using Dropout. DenseNets achieve lower error rates while using fewer parameters than ResNet. Without data augmentation, DenseNet performs better by a large margin.

Method	Depth	Params	C10	C10+	C100	C100+	SVHN
Network in Network [23]	-	-	10.41	8.81	35.68	-	2.35
All-CNN [35]	-	-	9.08	7.25	-	33.71	-
Deeply Supervised Net [11]	-	-	9.69	7.97	-	34.57	1.92
Highway Network [4]	-	-	-	7.72	-	32.39	-
FractalNet [9]	21	38.6M	10.18	5.22	35.34	23.30	2.01
with Dropout/Drop-path	21	38.6M	7.33	4.60	28.20	23.73	1.87
ResNet [5]	110	1.7M	-	6.61	-	-	-
ResNet (reported by [8])	110	1.7M	13.63	6.41	44.74	27.22	2.01
ResNet with Stochastic Depth [8]	110	1.7M	11.66	5.23	37.80	24.58	1.75
	1202	10.2M	-	4.91	-	-	-
Wide ResNet [22]	16	11.0M	-	4.81	-	22.07	-
	28	36.5M	-	4.17	-	20.50	-
with Dropout	16	2.7M	-	-	-	-	1.64
ResNet (pre-activation) [36]	164	1.7M	11.26^*	5.46	35.58^*	24.33	-
	1001	10.2M	10.56^*	4.62	33.47^*	22.71	-
DenseNet ($k = 12$)	100	0.8M	**5.92**	4.51	**24.15**	22.27	1.76
DenseNet ($k = 24$)	250	15.3M	**5.19**	**3.62**	**19.64**	**17.60**	1.74
DenseNet ($k = 40$)	190	25.6M	-	**3.46**	-	**17.18**	-

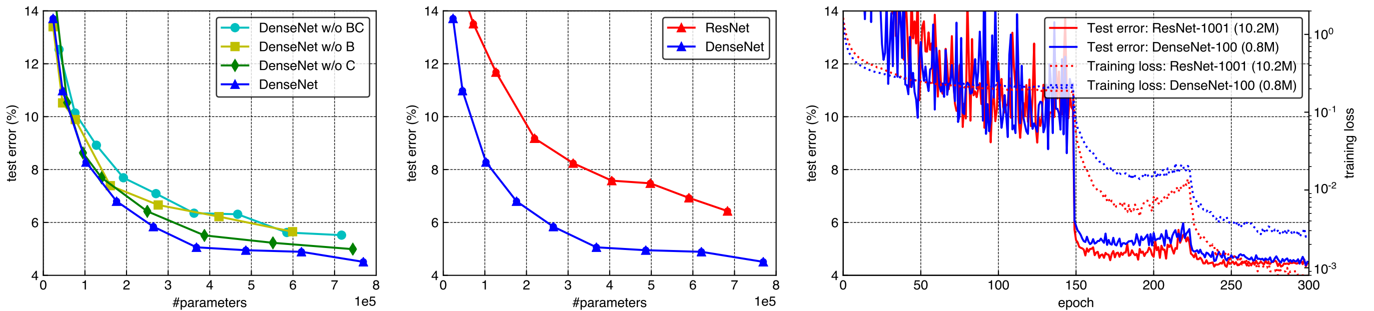

Fig. 4: *Left:* Comparison of the parameter efficiency on C10+ between DenseNet variations. *Middle:* Comparison of the parameter efficiency between DenseNet and (pre-activation) ResNets. DenseNet requires about 1/3 of the parameters as ResNet to achieve comparable accuracy. *Right:* Training and testing curves of the 1001-layer pre-activation ResNet [36] with more than 10M parameters and a 100-layer DenseNet with only 0.8M parameters.

set the dropout rate to 0.2. The test errors were only evaluated once for each task and model setting.

4.3 Classification Results on CIFAR and SVHN

We train DenseNets with different depths, L, and growth rates, k. The main results on CIFAR and SVHN are shown in Table 2. To highlight general trends, we mark all results that outperform the existing state-of-the-art in **boldface** and the overall best result in **blue**.

Accuracy. Possibly the most noticeable trend is observable in the bottom row of Table 2, which shows that DenseNet with $L = 190$ and $k = 40$ outperforms the existing state-of-the-art consistently on *all* the CIFAR datasets. Its error rates of 3.46% on C10+ and 17.18% on C100+ are significantly lower than the error rates achieved by wide ResNet architecture [22]. Our best results on C10 and C100 (without data augmentation) are even more encouraging: both are close to 30% lower than FractalNet with drop-path regularization [9]. On SVHN, with dropout, the DenseNet with $L = 100$ and $k = 24$ also surpasses the current best result achieved by wide ResNet. However, the 250-layer DenseNet does not further improve the performance significantly over its shorter counterpart. This may be explained by the fact that SVHN is a relatively easy task and very deep models tend to overfit to the training set.

Capacity. Without compression or bottleneck layers, there is a general trend that DenseNets perform better as L and k increase. We attribute this primarily to the corresponding growth in model capacity. This is best demonstrated by the column of C10+ and C100+. On C10+, the error drops from 4.51% to 3.62% and finally to 3.46% as the number of parameters increases from 0.8M, over 15.3M to 25.6M. On C100+, we observe a similar trend. This suggests that DenseNets can utilize the increased representational power of bigger and deeper models. It also indicates that DenseNets do not suffer from overfitting or the optimization difficulties of residual networks [5].

Parameter Efficiency. The results in Table 2 indicate that DenseNets utilize parameters more effectively than alternative model architectures (in particular, ResNets). The DenseNet with bottleneck structure and compression at transition layers is particularly parameter-efficient. For example, our deepest model only has 15.3M parameters, but it consistently outperforms other models such as FractalNet and Wide ResNets with more than 30M parameters. We also highlight that DenseNet with $L = 100$ and $k = 12$ achieves comparable performance (*e.g.*, 4.51% vs 4.62%

Model	top-1	top-5
DenseNet-121	25.02 / 23.61	7.71 / 6.66
DenseNet-169	23.80 / 22.08	6.85 / 5.92
DenseNet-201	22.58 / 21.46	6.34 / 5.54
DenseNet-264	22.15 / 20.80	6.12 / 5.29

TABLE 3: The top-1 and top-5 error rates on the ImageNet validation set, with single-crop / 10-crop testing.

error on C10+, 22.27% vs 22.71% error on C100+) as the 1001-layer pre-activation ResNet using 9× fewer parameters. Fig. 4 (right panel) shows the training loss and test error of these two networks on C10+. The 1001-layer deep ResNet converges to a lower training loss value but a similar test error. We analyze this effect in more detail below.

Overfitting. One positive side-effect of the more efficient use of parameters is a tendency of DenseNets to be less prone to overfitting. We observe that on the datasets without data augmentation, the improvements of DenseNet architectures over prior work are particularly pronounced. On C10, the improvement denotes a 29% relative reduction in error from 7.33% to 5.19%. On C100, the reduction is about 30% from 28.20% to 19.64%.

4.4 Classification Results on ImageNet

We evaluate DenseNet with different depths and growth rates on the ImageNet classification task, and compare it with state-of-the-art ResNet architectures. To ensure a fair comparison between the two architectures, we eliminate all other factors such as differences in data preprocessing and optimization settings by adopting the publicly available LuaTorch implementation for ResNet [43][6]. We simply replace the ResNet model with the DenseNet network and keep all the experimental settings unchanged to match those used for ResNets. All results were obtained with single centered test-image crops. Fig. 5 shows the validation errors of DenseNets and ResNets on ImageNet as a function of the number of parameters (left) and flops (right). The results presented in the figure reveal that DenseNets perform on par with the state-of-the-art ResNets, whilst requiring significantly fewer parameters and computation to achieve comparable performance. For example, a DenseNet-201 with 20M parameters model yields similar validation error as a 101-layer ResNet with more than 40M parameters. Similar trends can be observed from the right panel, which plots the validation error as a function of the number of FLOPs: a DenseNet that requires as much computation as a ResNet-50 performs on par with a ResNet-101, which requires twice as much computation. It is also clear from these results that DenseNet performance continues to improve measurably as more layers are added.

It is worth noting that our experimental setup implies that we use hyperparameter settings that are optimized for ResNets but not for DenseNets. It is conceivable that more extensive hyper-parameter searches may further improve the performance of DenseNet on ImageNet.

4.5 Discussion

Superficially, DenseNets are quite similar to ResNets: Eq. (2) differs from Eq. (1) only in that the inputs to $H_\ell(\cdot)$ are concatenated instead of summed. However, the implications of this seemingly small modification lead to substantially different behaviors of the two network architectures.

Model compactness. As a direct consequence of the input concatenation, the feature maps learned by any of the DenseNet layers can be accessed by all subsequent layers. This encourages feature reuse throughout the network, and leads to more compact models.

The left two plots in Fig. 4 show the result of an experiment that aims to compare the parameter efficiency of the various variants of DenseNets (left) and also a comparable ResNet architecture (middle). We train multiple small networks with varying depths on C10+ and plot their test accuracies as a function of network parameters. In comparison with other popular network architectures, such as AlexNet [30] or VGG-net [3], ResNets with pre-activation use fewer parameters while typically achieving better results [36]. Hence, we compare DenseNet ($k = 12$) against this architecture. The training setting for DenseNet is kept the same as in the previous section.

The graph shows that DenseNet with bottleneck layer structure and transition layer compression is consistently the most parameter efficient among these variants. Further, to achieve the same level of accuracy, DenseNet only requires around 1/3 of the parameters of ResNets (middle plot). This result is in line with the results on ImageNet we presented in Fig. 5. The right plot in Fig. 4 shows that a DenseNet with only 0.8M trainable parameters is able to achieve comparable accuracy as the 1001-layer (pre-activation) ResNet [36] with 10.2M parameters.

Implicit Deep Supervision. One explanation for the improved accuracy of dense convolutional networks may be that individual layers receive additional supervision from the loss function through the shorter connections. One can interpret DenseNets to perform a form of "deep supervision". The benefits of deep supervision have previously been shown in deeply-supervised nets (DSN; [11]), which have classifiers attached to every hidden layer, enforcing the intermediate layers to learn discriminative features.

DenseNets perform a similar deep supervision in an implicit fashion: a single classifier on top of the network provides direct supervision to all layers through at most two or three transition layers. However, the loss function and gradient of DenseNets are substantially less complicated, as the same loss function is shared between all layers.

Stochastic vs. deterministic connection. There is an interesting connection between dense convolutional networks and stochastic depth regularization of residual networks [8]. In stochastic depth, layers in residual networks are randomly dropped, which creates direct connections between the surrounding layers. As the pooling layers are never dropped, the network results in a similar connectivity pattern as DenseNet: There is a small probability for any two layers, between the same pooling layers, to be directly connected—if all intermediate layers are randomly dropped. Although the methods are ultimately quite different, the DenseNet interpretation of stochastic depth may provide insights into the success of this regularizer.

Feature Reuse. By design, DenseNets allow layers access to feature maps from all of its preceding layers (although sometimes through transition layers). We conduct an experiment to investigate if a trained network takes advantage of this opportunity. We first train a DenseNet on C10+ with $L = 40$ and $k = 12$. For each convolutional layer ℓ within a block, we compute the average (absolute) weight assigned to connections with layer s. Fig. 6 shows a heat-map for all three dense blocks. The average absolute weight serves as a surrogate for the dependency of a convolutional layer on its preceding layers. A red dot in position (ℓ, s) indicates

6. https://github.com/facebook/fb.resnet.torch

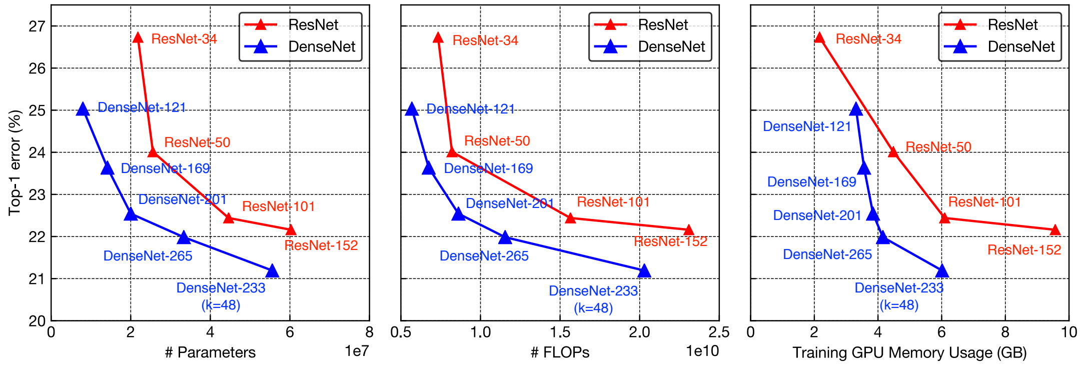

Fig. 5: Comparison of the DenseNet and ResNet Top-1 (single model and single-crop) error rates on the ImageNet classification dataset as a function of learned parameters (*left*), flops (*middle*, and GPU memory footprint at training time(*right*). Training GPU memory measured using the efficient LuaTorch DenseNet implementation with a batch size of 64.

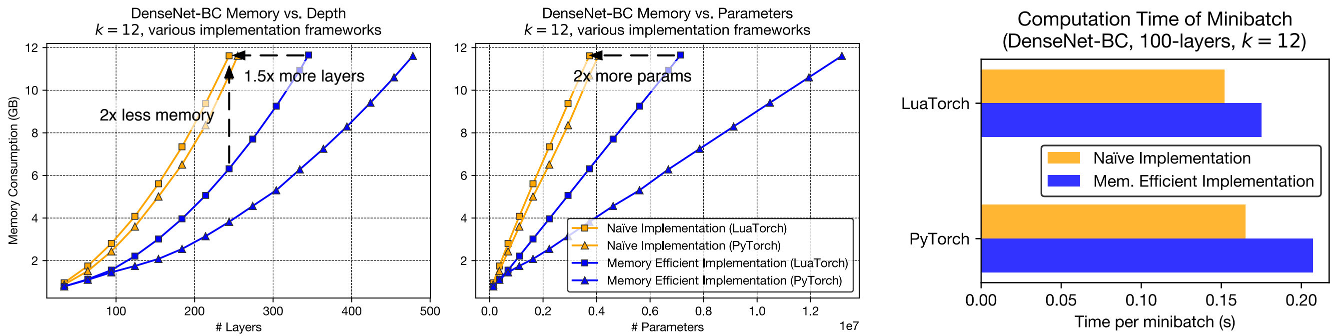

Fig. 7: *Left and Middle:* GPU memory consumption as a function of network depth/number of parameters. Each model is a DenseNet with $k = 12$ features added per layer. The efficient implementation can train much deeper models with less memory. *Right:* Computation time (on a NVIDIA Maxwell Titan-X).

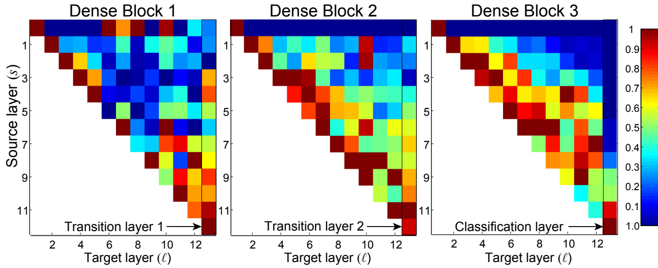

Fig. 6: The average absolute filter weights of convolutional layers in a trained DenseNet. The color of pixel (s, ℓ) encodes the average $L1$ norm (normalized by the number of input feature maps) of the weights connecting convolutional layer s to layer ℓ within a dense block. The three columns highlighted by black rectangles correspond to the two transition layers and the classification layer. The first row encodes those weights connected to the input layer of the dense block.

that the layer ℓ makes, on average, strong use of feature maps produced s-layers before. Several observations can be made from the plot:

1) All layers spread their weights over many inputs within the same block. This indicates that features extracted by very early layers are, indeed, directly used by deep layers throughout the same dense block.
2) The weights of the transition layers also spread their weight across all layers within the preceding dense block, indicating information flow from the first to the last layers of the DenseNet through few indirections.
3) The layers within the second and third dense block consistently assign the least weight to the outputs of the transition layer (the top row of the triangles), indicating that the transition layer outputs many redundant features (with low weight on average). This is in keeping with the strong results of DenseNet where exactly these outputs are compressed.
4) Although the final classification layer, shown on the very right, also uses weights across the entire dense block, there seems to be a concentration towards final feature-maps, suggesting that there may be some more high-level features produced late in the network.

4.6 Memory Efficient Implementation Results

We compare the memory consumption and computation time of three DenseNet implementations during training. The naïve implementation allocates memory for every pre-activation batch normalization operation. We compare this against a memory-efficient implementation of DenseNets, which includes all optimizations described in Section 3.2. The concatenated and normalized feature maps are recomputed as necessary during back-propagation. The only tensors stored in memory during training are the convolution feature maps and the parameters of the network. We test these two

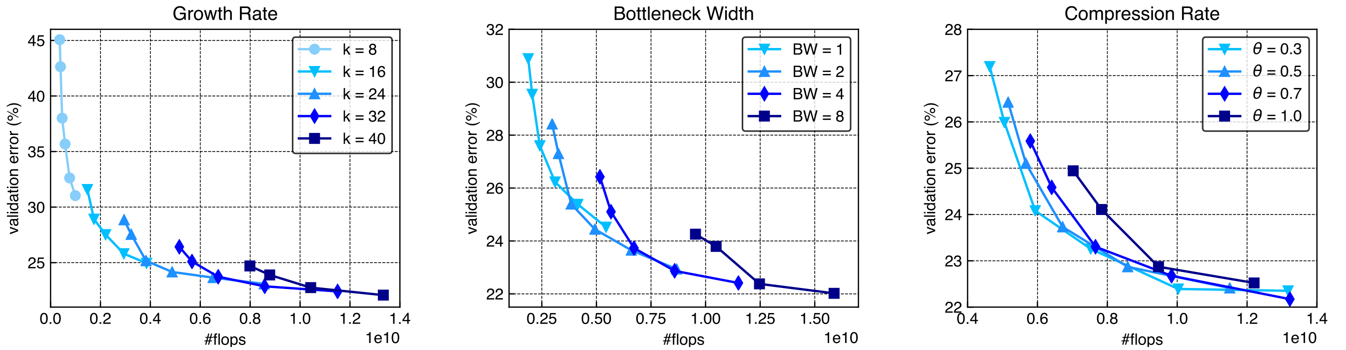

Fig. 8: Top-1 validation error on ImageNet as a function of the computational cost (measured by flops) of different DenseNets, with varying growth rate (*Left*), varying width of the bottleneck layers (*Middle*) and varying compression ratio at transition layers (*Right*).

implementations in both the LuaTorch and PyTorch[7] deep learning frameworks.

Memory consumption. We train networks of various depth on the CIFAR-10 dataset. All networks have a growth rate of $k = 12$ and are trained with a batch size of 64. In Fig. 7, we see that the naïve implementation becomes memory intensive very quickly in both LuaTorch and PyTorch. The memory usage of a 160 layer network ($1.8M$ parameters) is roughly 10 times as much as a 40 layer network ($160K$ parameters). Training a larger network with more than 220 layers requires over 12 GB of memory, pushing the memory limits of a typical single GPU. On the other hand, using all the memory-sharing operations significantly reduces memory consumption. In LuaTorch, the 220-layer model uses 50% 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 $1.5\times$ as deep and has $2\times$ as many parameters as the best naïve implementation model. With the PyTorch efficient implementation, we can train 500 layer networks ($13M$ parameters) on a single GPU. The "autograd" library in PyTorch performs memory optimizations during training, which likely contribute to this implementation's efficiency.

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.

Training time. In the right panel of Fig. 7, we plot the time per minibatch of a 100-layer DenseNet ($k = 12$) on a NVIDIA Maxwell Titan-X. The efficient implementation adds roughly 15% time overhead on LuaTorch, and 20% on PyTorch. This extra cost is a result of recomputing the intermediate feature maps during back-propagation. 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 on the ImageNet classification dataset. The deepest model trained using the naïve implementation was 201 layers ($20M$ parameters). With the efficient LuaTorch implementation however, we are able to train two deeper DenseNet models with the efficient implementation, one with 265 layers ($k = 32$, $33M$ parameters) and one with 233 layers ($k = 48$, $55M$ parameters).[8]

5 Architecture Hyperparameters

We perform a series of analytical experiments to study how hyperparameter choices associated with aspects of the network architecture, affect the performance of DenseNets. Specifically, we examine three hyperparameters: the growth rate k, the bottleneck width (number of filters in the $1{\times}1$ bottleneck layer) and the compression rate at transition layers θ.

As larger models tend to yield higher accuracy, we incorporate the computational cost whenever we compare two models directly. Therefore, we always train multiple DenseNets with varying depth for each hyperparameter setting, and compare different configurations on the error *v.s.* compute (number of *flops*) plot. The depth of the models ranges from 101 layers to 329 layers, with the number of basic layers for the four dense blocks selected from the set $\{[6, 12, 18, 12]$, $[6, 12, 24, 16]$, $[6, 12, 32, 32]$, $[6, 12, 48, 32]$, $[6, 12, 64, 48]$, $[6, 12, 80, 64]\}$.

Growth rate k. The growth rate determines the *width* of each layer, i.e., the number of feature maps produced by H_ℓ as given in Eq.(2). We experiment with various growth rates from the set $\{8, 16, 24, 32, 40\}$.

The results are shown in the left panel of Fig. 8. We can observe that due to the dense connectivity, even DenseNets with very *narrow* layers (e.g., $k = 8$) can be trained effectively. Although each layer only produces 8 feature maps (light dotted blue curve), the model still results in highly competitive results. In fact, a small growth rate is essential for DenseNets to achieve high computational efficiency. For example, to achieve a 24% validation error, a DenseNet with growth rate 24 requires about $0.50{\times}10^{10}$ flops; while a similar architecture with growth rate 40 requires $0.88{\times}10^{10}$ flops. However, as the networks grow deeper, larger growth rates seem to show greater potential. This indicates that to achieve high efficiency for a DenseNet, we should ensure its depth and width are compatible. It is noteworthy that wider convolutional layers can be more efficiently computed on GPUs due to better parallelism. Therefore, a larger growth rate may be preferred if one is more concerned about wall time efficiency during training.

Bottleneck layer width. The convolution layer with filter size $1{\times}1$ introduced to the transformation H_ℓ significantly improves

7. http://github.com/torch/torch7/, http://github.com/pytorch/pytorch/

8. In the 233-layer model, the four dense blocks have 6, 12, 48, and 48 layers, respectively.

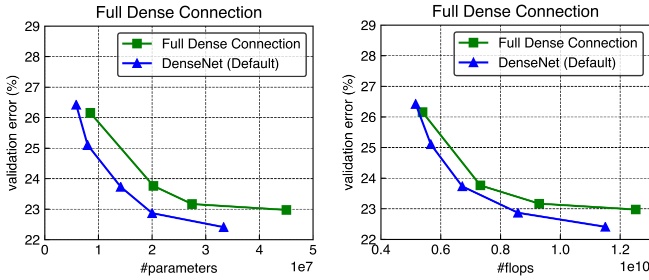

Fig. 9: Comparison of DenseNet and its variant with full dense connectivity, in terms of parameter efficiency (*Left*) and computational efficiency (*Right*).

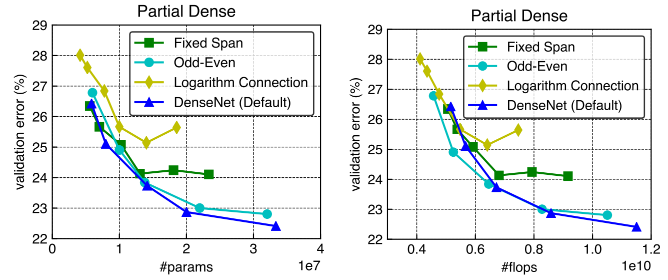

Fig. 10: Comparison of DenseNet and its variants with partial dense connectivity, in terms of parameter efficiency (*Left*) and computational efficiency (*Right*).

the parameter efficiency of DenseNets. It performs dimension reduction on the concatenated feature maps before passing them to the more expensive 3×3 convolution layer. In all our previous experiments, we fix the width of these bottleneck layers to $m \times k$, where $m = 4$, and k is the growth rate. To understand how the performance of DenseNets is affected by m, we train multiple DenseNets with m selected from the set $\{1, 2, 4, 8\}$. All the other hyperparameters are set to their default value, except that we vary the depth of the networks to make the error *v.s.* flops plot.

We show the results in the middle panel of Fig. 8. Here, the trend is similar to what we have observed in the experiment with varying growth rates: wider bottleneck layers (e.g., $m = 8$) yield lower computational efficiency on smaller networks. Therefore, the bottleneck layer width should also be compatible with the network depth in order to maximize the parameter efficiency of DenseNets.

Compression factor. In previous experiments with DenseNets, each of the transition layers between two *dense blocks* halves the number of channels, i.e. we have set $\theta = 0.5$ throughout. Here we conduct an experiment to study how sensitive the model performance is to the compression factor θ.

The results are shown in the right panel of Fig. 8. In general, the parameter efficiency of DenseNets is quite insensitive to the compression rate. There seems to be no significant difference between the three curves with $\theta = 0.3$, $\theta = 0.5$ and $\theta = 0.7$, when the flops is greater than $0.8{\times}10^{10}$. However, we do observe that DenseNets with smaller compression factor consistently outperform those with larger compression factor when model size is small. This is to some extent counterintuitive, as we would expect a small compression factor (more reduction) to be more beneficial in larger models, which tend to create more redundancy in the feature maps due to the quadratically growing connections. A possible explanation is that low level features in deep networks are indeed actively reused by deeper layers, and keeping more information about the input is helpful when the network has more capacity to process.

6 DENSENET VARIANTS

Multiple variants of the DenseNet architecture are possible, which we discuss and experiment with briefly.

6.1 Full Dense Connectivity

In a DenseNet, as described so far, layers in different dense blocks are only indirectly connected via a transition layer. This transition layer is necessary because layers in different dense blocks have incompatible feature map sizes. One possible variant of the DenseNet layout is to obtain *full dense connectivity* (FDC), and truly connect each layer with every other layer. We can achieve such a configuration by incorporating a downsampling step via pooling directly into connections between layers with different feature map sizes. This is also equivalent to simplifying the transition layer to a simple pooling operation without 1×1 convolution and compression.

We give a comparison of the network with FDC and the original DenseNet in Fig. 9. It can be observed that DenseNet with transition layers performs significant better in terms of both parameter and computational efficiency. These results suggest that the 1×1 convolution in transition layers may be helpful in compressing redundant features.

6.2 Partial Dense Connection

The DenseNet architecture introduces a quadratic number of connections to the network. It is a fair question to ask if all of these are indeed necessary. In this subsection, we experiment with some variants of DenseNet, which only keep part of the dense connections. Specifically, we consider three settings: (1) each layer is connected to the most recent M layers; (2) an odd (even) layer is connected to all previous even (odd) layers in that dense block; and (3) each layer is connected to the 2^ith layer before it[9] , $i = 0, 1, 2, \ldots$. Similar to the setting in last subsection, we train multiple networks with varying depth for each architecture and use parameter and computational efficiency as the main performance criteria.

Fig. 10 shows the results of these three architectural variants, as well as the error rate of the standard DenseNet structure. The first variant has dense connections with a span of less than 4, 8 or 12 layers. As shown by the green curve in 10, this connectivity pattern tends to yield higher validation error than standard DenseNets. When the depth becomes large, the network cannot be trained effectively and the validation error stops decreasing as the network becomes deeper (the training error follows a similar trend). These results demonstrate that earlier features are indeed utilized and necessary for deeper layers in a DenseNet, and the long range connections are critical for effectively training very deep models. The second variant features only half of the number of connections of standard DenseNets. From the cyan curve in 10, one can observe that this connectivity pattern leads to slightly worse parameter efficiency, while it has a higher computational efficiency when depth is not very large. For deeper models this advantage disappears. The third variant networks also under-perform standard DenseNets, especially with larger model

9. A similar design choice was also investigated in [47].

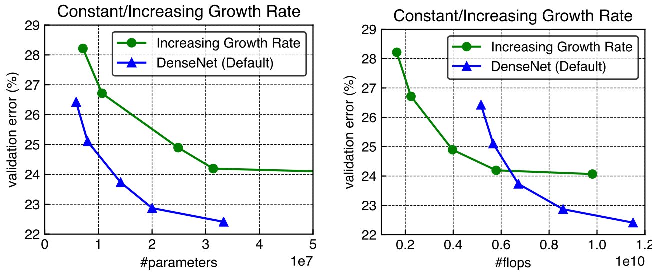

Fig. 11: Comparison of DenseNet and its variant with increasing growth rate, in terms of parameter efficiency (*Left*) and computational efficiency (*Right*).

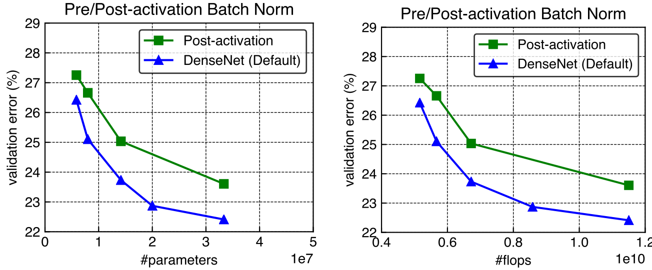

Fig. 12: Comparison of pre- and post- activation BN in DenseNets, in terms of parameter efficiency (*Left*) and computational efficiency (*Right*).

sizes. This set of experiments further support our hypothesis that dense connections strengthen the information/gradient flow in the network, and enable training of deep models effectively.

6.3 Exponentially Increasing Growth Rate

The standard DenseNet use a constant growth rate throughout the whole network. Compared to other network architectures, e.g., the VGG [3] and ResNet [5], which doubles the number of feature channels after each down-sampling layer, DenseNets/DenseNets tend to allocate fewer parameters towards processing low resolution feature maps. Such a design is oriented more towards parameter efficiency than computational efficiency, as pointed out by [48].

In Fig. 11 we compare the standard DenseNet structure with a variant that doubles the growth rate after each transition layer. Standard DenseNets have a constant growth rate of 32, while for the latter network, the growth rate in the jth dense block has a growth rate of $k_0 \times 2^{j-1}$ ($j = 1, 2, 3, 4$). According to the results, the DenseNet with exponentially increasing growth rate has lower efficiency in terms of parameter efficiency (left panel of Fig. 11), while it is much more competitive on the error $v.s.$ flops plot (right panel of Fig. 11). This suggests that using larger growth rate for deeper dense blocks should be preferred in scenarios where computation is the major concern.

6.4 Post-activation Batch Normalization

The DenseNet architecture utilizes *pre-activation* batch normalization. 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 constant large than 1, while Layer 3 might scale the same feature by a small positive constant. This provides Layer 2 and Layer 3 with the flexibility to up-weight and down-weight a same feature independently. Note that this would not be possible if all layers shared the same batch normalization operation, or if normalization occurred after convolution operations. Fig. 12 provides a comparison between DenseNets with pre- and post- activation batch normalization. There is a clear trend that the former leads to significantly higher efficiency. However, pre-activation batch normalization generally leads to substantially higher memory footprint during training. If GPU memory is limited, we can apply the memory optimization strategy described in Section 3.2.

7 Conclusion

We proposed a new convolutional network architecture, which we refer to as Dense Convolutional Network (DenseNet). It introduces direct connections between any two layers with the same feature-map size. Whilst following a simple connectivity rule, DenseNets naturally integrate the properties of identity mappings, deep supervision, and diversified depth. They allow feature reuse throughout the network and can consequently learn more compact and, according to our experiments, more accurate models.

We showed that DenseNets scale naturally to hundreds of layers, while exhibiting no optimization difficulties. In our experiments, DenseNets tend to yield consistent improvement in accuracy with growing number of parameters, without any signs of performance degradation or overfitting. Under multiple settings, DenseNets achieve state-of-the-art results across several highly competitive datasets. Moreover, DenseNets require substantially fewer parameters and less computation than prior work at comparable accuracy levels.

Acknowledgments

The authors are supported in part by the NSF III-1618134, III-1526012, IIS-1149882, IIS-1724282, the Office of Naval Research Grant N00014-17-1-2175, the Bill and Melinda Gates foundation, SAP America Inc., and the NSF TRIPODS Award #1740822 (Cornell TRIPODS Center for Data Science for Improved Decision Making). We thank Danlu Chen, Daniel Sedra, Tongcheng Li and Yu Sun for many insightful discussions.

References

[1] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, “Backpropagation applied to handwritten zip code recognition,” *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[3] 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*, vol. 115, no. 3, pp. 211–252, 2015.

[4] R. K. Srivastava, K. Greff, and J. Schmidhuber, “Training very deep networks,” in *NIPS*, 2015.

[5] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *CVPR*, 2016.

[6] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” in *AISTATS*, 2011.

[7] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *ICML*, 2015.

[8] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger, "Deep networks with stochastic depth," in *ECCV*, 2016.

[9] G. Larsson, M. Maire, and G. Shakhnarovich, "Fractalnet: Ultra-deep neural networks without residuals," in *ICLR*, 2017.

[10] Q. Liao and T. Poggio, "Bridging the gaps between residual learning, recurrent neural networks and visual cortex," *arXiv preprint arXiv:1604.03640*, 2016.

[11] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu, "Deeply-supervised nets," in *AISTATS*, 2015.

[12] S. E. Fahlman and C. Lebiere, "The cascade-correlation learning architecture," in *NIPS*, 1989.

[13] B. M. Wilamowski and H. Yu, "Neural network learning without backpropagation," *IEEE Transactions on Neural Networks*, vol. 21, no. 11, pp. 1793–1803, 2010.

[14] B. Hariharan, P. Arbeláez, R. Girshick, and J. Malik, "Hypercolumns for object segmentation and fine-grained localization," in *CVPR*, 2015.

[15] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *CVPR*, 2015.

[16] P. Sermanet, K. Kavukcuoglu, S. Chintala, and Y. LeCun, "Pedestrian detection with unsupervised multi-stage feature learning," in *CVPR*, 2013.

[17] S. Yang and D. Ramanan, "Multi-scale recognition with dag-cnns," in *ICCV*, 2015.

[18] C. Cortes, X. Gonzalvo, V. Kuznetsov, M. Mohri, and S. Yang, "Adanet: Adaptive structural learning of artificial neural networks," *arXiv preprint arXiv:1607.01097*, 2016.

[19] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *CVPR*, 2015.

[20] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *CVPR*, 2016.

[21] S. Targ, D. Almeida, and K. Lyman, "Resnet in resnet: Generalizing residual architectures," *arXiv preprint arXiv:1603.08029*, 2016.

[22] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv:1605.07146*, 2016.

[23] M. Lin, Q. Chen, and S. Yan, "Network in network," in *ICLR*, 2014.

[24] A. Rasmus, M. Berglund, M. Honkala, H. Valpola, and T. Raiko, "Semi-supervised learning with ladder networks," in *NIPS*, 2015.

[25] M. Pezeshki, L. Fan, P. Brakel, A. Courville, and Y. Bengio, "Deconstructing the ladder network architecture," in *ICML*, 2016.

[26] J. Wang, Z. Wei, T. Zhang, and W. Zeng, "Deeply-fused nets," *arXiv preprint arXiv:1605.07716*, 2016.

[27] Y. Chen, J. Li, H. Xiao, X. Jin, S. Yan, and J. Feng, "Dual path networks," in *NIPS*, 2017, pp. 4470–4478.

[28] S. Jégou, M. Drozdzal, D. Vazquez, A. Romero, and Y. Bengio, "The one hundred layers tiramisu: Fully convolutional densenets for semantic segmentation," in *Computer Vision and Pattern Recognition Workshops (CVPRW)*. IEEE, 2017, pp. 1175–1183.

[29] X. Li, H. Chen, X. Qi, Q. Dou, C.-W. Fu, and P. A. Heng, "H-denseunet: Hybrid densely connected unet for liver and liver tumor segmentation from ct volumes," *arXiv preprint arXiv:1709.07330*, 2017.

[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.

[31] G. Huang, S. Liu, L. van der Maaten, and K. Q. Weinberger, "Condensenet: An efficient densenet using learned group convolutions," in *CVPR*, 2018.

[32] F. Chollet, "Xception: Deep learning with depthwise separable convolutions."

[33] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7132–7141.

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

[35] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," *arXiv preprint arXiv:1412.6806*, 2014.

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

[37] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," *Tech Report*, 2009.

[38] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, "Fitnets: Hints for thin deep nets," in *ICLR*, 2015.

[39] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning, 2011," in *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[40] I. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout networks," in *ICML*, 2013.

[41] P. Sermanet, S. Chintala, and Y. LeCun, "Convolutional neural networks applied to house numbers digit classification," in *ICPR*. IEEE, 2012, pp. 3288–3291.

[42] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *CVPR*, 2009.

[43] S. Gross and M. Wilber, "Training and investigating residual nets," 2016. [Online]. Available: http://torch.ch/blog/2016/02/04/resnets.html

[44] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *ICML*, 2013.

[45] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *ICCV*, 2015.

[46] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting." *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[47] H. Hu, D. Dey, A. Del Giorno, M. Hebert, and J. A. Bagnell, "Log-densenet: How to sparsify a densenet," *arXiv preprint arXiv:1711.00002*, 2017.

[48] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.