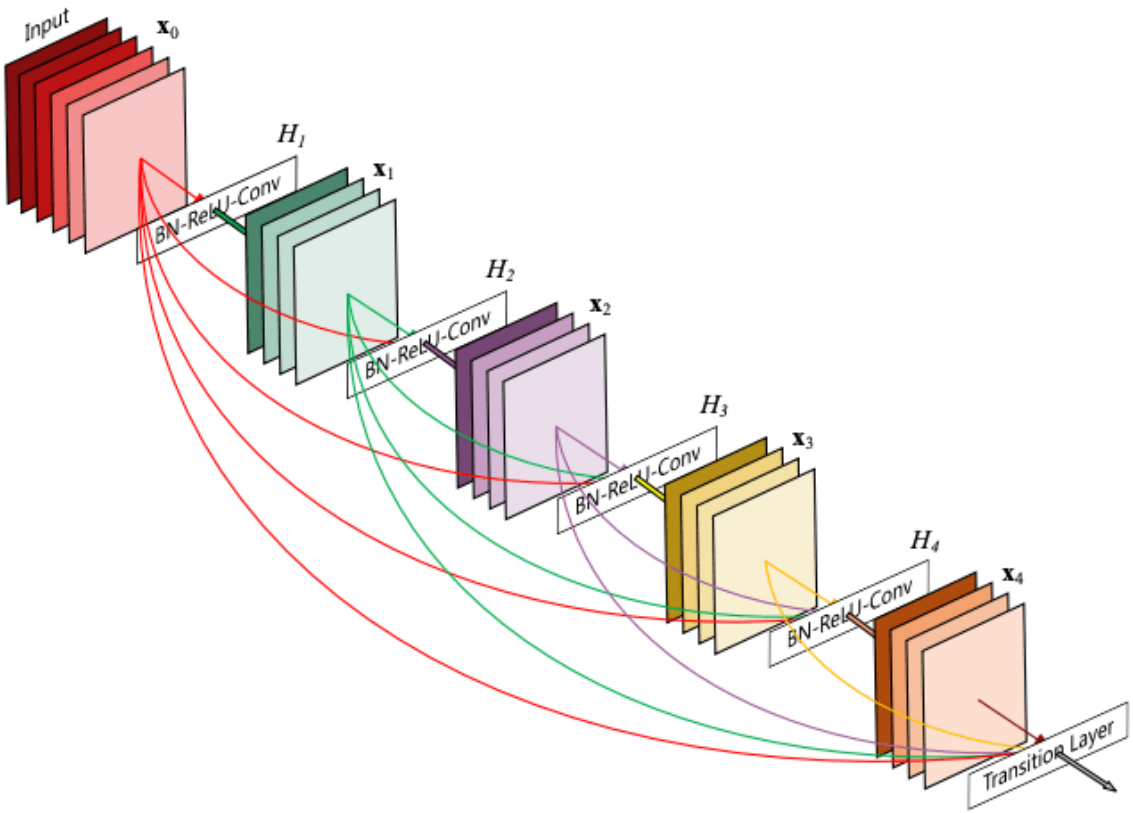
Session 16 & Assignment

Re-submit Assignment

Due Sep 1 by 11:59pm **Points** 600 **Submitting** a website url **Available** after Aug 25 at 2pm

DENSENETs

maximum information flow between layers in the network



We know that CNNs can be substantially deeper, mode accurate and efficient to train if they contain shorter connections between layers close to the input and those close to the output. DenseNet embrace this observation and creates a network which connects each layer to other layer in a feed-forward fashion.

Traditional networks with L layers have L connections, DenseNet has L*(L + 1)/2 direct connections.

For each layer, the feature-maps of all preceding layers are used as inputs, and it's own feature-maps are ued as inputs into all subsequent layers.

DenseNet:

- alleviate the vanishing gradient problem,
- strengthen feature propagation
- encourage feature reuse
- substantially reduce the number of parameters
- in contrast to ResNets, it never combine features through summation before they are passed into a layer, instead it combine features by concatenating them
- achieves state of art on CIFAR10, CIFAR100, ImageNet and others.

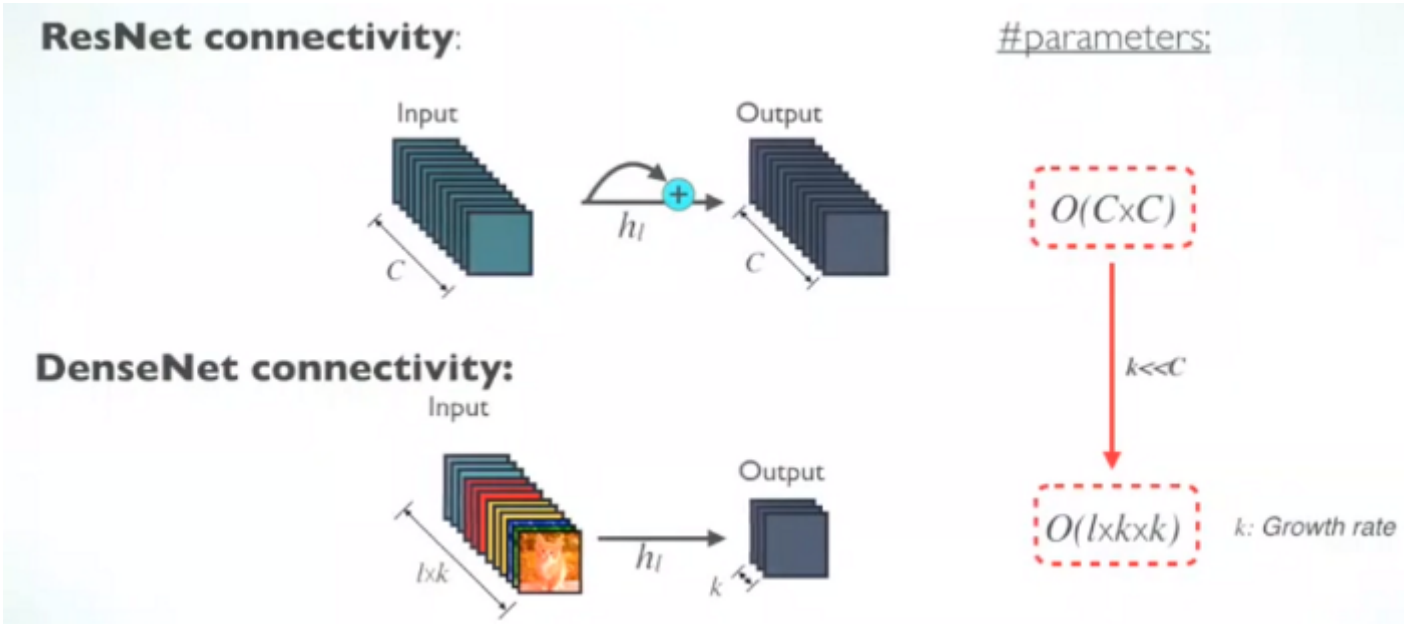
Method	Depth	Params	C10	C10+	C100	C100+	SVHN
Network in Network [22]	-	-	10.41	8.81	35.68	-	2.35
All-CNN [31]	-	-	9.08	7.25	-	33.71	-
Deeply Supervised Net [20]	-	-	9.69	7.97	-	34.57	1.92
Highway Network [33]	-	-	-	7.72	-	32.39	-
FractalNet [17]	21	38.6M	10.18	5.22	35.34	23.30	2.01
with Dropout/Drop-path	21	38.6M	7.33	4.60	28.20	23.73	1.87
ResNet [11]	110	1.7M	-	6.61	-	-	-
ResNet (reported by [13])	110	1.7M	13.63	6.41	44.74	27.22	2.01
ResNet with Stochastic Depth [13]	110	1.7M	11.66	5.23	37.80	24.58	1.75
	1202	10.2M	-	4.91	-	-	-
Wide ResNet [41]	16	11.0M	-	4.81	-	22.07	-
	28	36.5M	-	4.17	-	20.50	-
with Dropout	16	2.7M	-	-	-	-	1.64
ResNet (pre-activation) [12]	164	1.7M	11.26*	5.46	35.58*	24.33	-
	1001	10.2M	10.56*	4.62	33.47*	22.71	-
DenseNet ($k = 12$)	40	1.0M	**7.00**	5.24	**27.55**	24.42	1.79
DenseNet ($k = 12$)	100	7.0M	**5.77**	**4.10**	**23.79**	**20.20**	1.67
DenseNet ($k = 24$)	100	27.2M	**5.83**	**3.74**	**23.42**	**19.25**	**1.59**
DenseNet-BC ($k = 12$)	100	0.8M	**5.92**	4.51	**24.15**	22.27	1.76
DenseNet-BC ($k = 24$)	250	15.3M	**5.19**	**3.62**	**19.64**	**17.60**	1.74
DenseNet-BC ($k = 40$)	190	25.6M	-	**3.46**	-	**17.18**	-

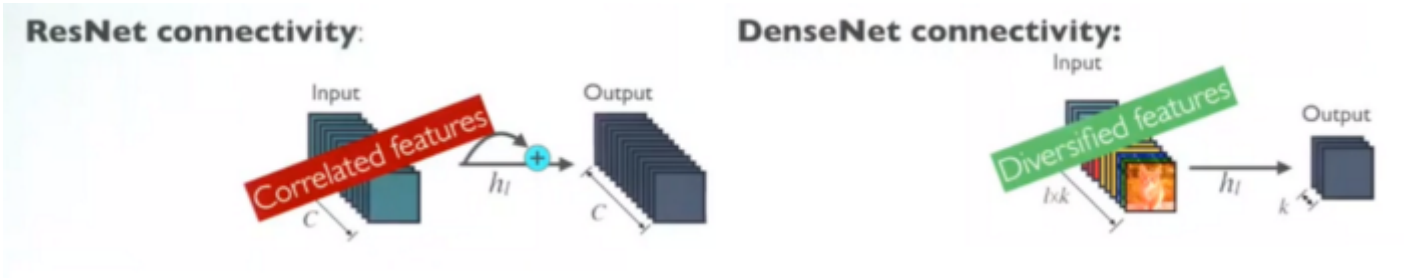
DenseNet needs lesser parameters

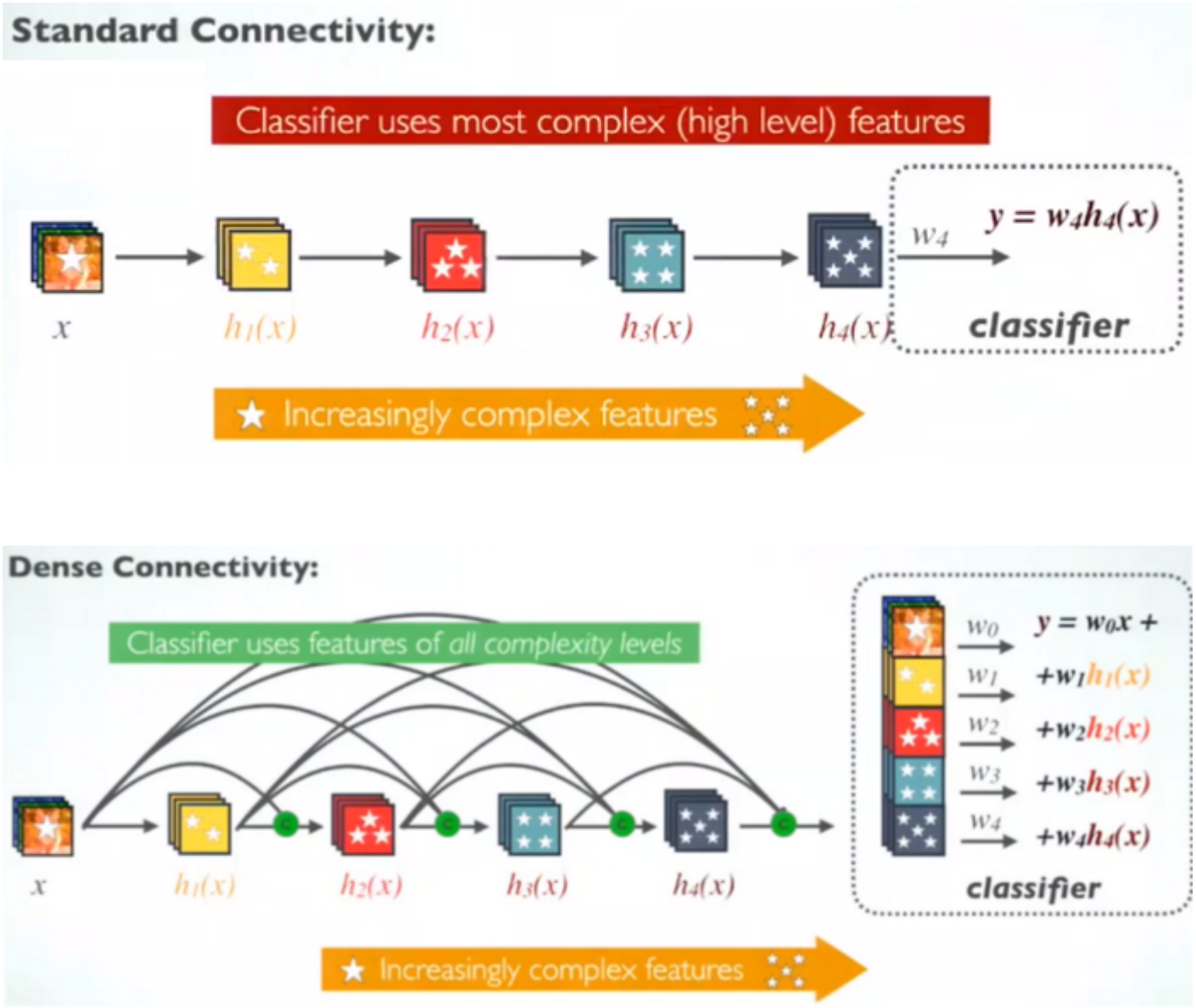
Counter-intuitive?

It needs fewer parameters than traditional convnets.

- Traditional feed-forward architectures can be viewed as algorithms with a state which is passed on from layer to layer
- 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
- ResNet make this information presentation explicit through additive identity transformations
- **Recent variations of ResNets show that:**
  - **many layers contribute very little and**
  - **can in fact be randomly dropped during training**
  - **this makes the state of ResNets similar to an unrolled recurrent neural network**
  - **but the number of params of ResNets is substantially larger because each layer has its own weight**
- DenseNet layers are very narrow (e.g. 12 filters per layer), adding only a small set of feature-maps to the "collective knowledge" of the network and keep the remaining maps unchanged
- Final classifier makes a decision based on all features-maps in the network
- this improved flow of information and gradients throughout the network, makes it easier to train as each layer has direct access to the gradients from the loss function
- DenseNets have also shown to reduce over-fitting on tasks with smaller training set sizes


ResNet connectivity:
Input
Output
h_l
C
C
#parameters:
O(CxC)
k<<C
O(lxkxk)
k: Growth rate
DenseNet connectivity:
Input
Output
lxk
h_l
k


ResNet connectivity:
Input
Correlated features
Output
h_l
C
C
DenseNet connectivity:
Input
Diversified features
Output
lxk
h_l
k



DenseNet Architecture

ResNet Architecture

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

DenseNet Architecture

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

where H_L is a **composite function** of operations such as Batch Normalization, ReLU, Pooling or Convolutions

Composite function consists of three consecutive operations:

1. Batch Normalization
2. ReLU
3. 3x3 Convolution

Of course this is inspired from pre-activated networks.

Pooling Layers. Concatenation operation is not viable when the size of the feature maps changes (down-samples). Since down-sampling is essential, DenseNet is divided into multiple connected dense blocks and transition blocks.

Between each Dense Block, we have a transition block, which does convolution and pooling operation. Transition layers used consists of BN and a 1x1 followed by a 2x2 average pooling layers

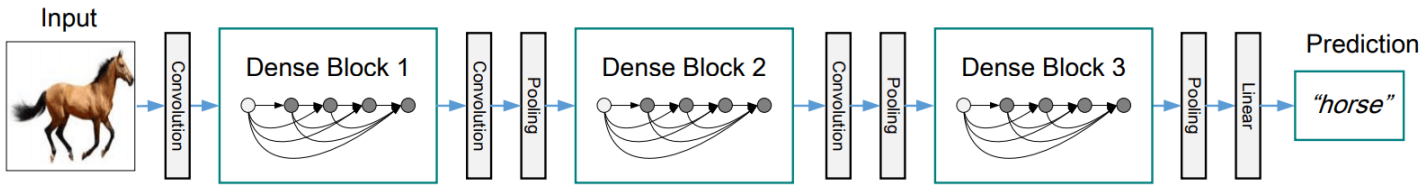


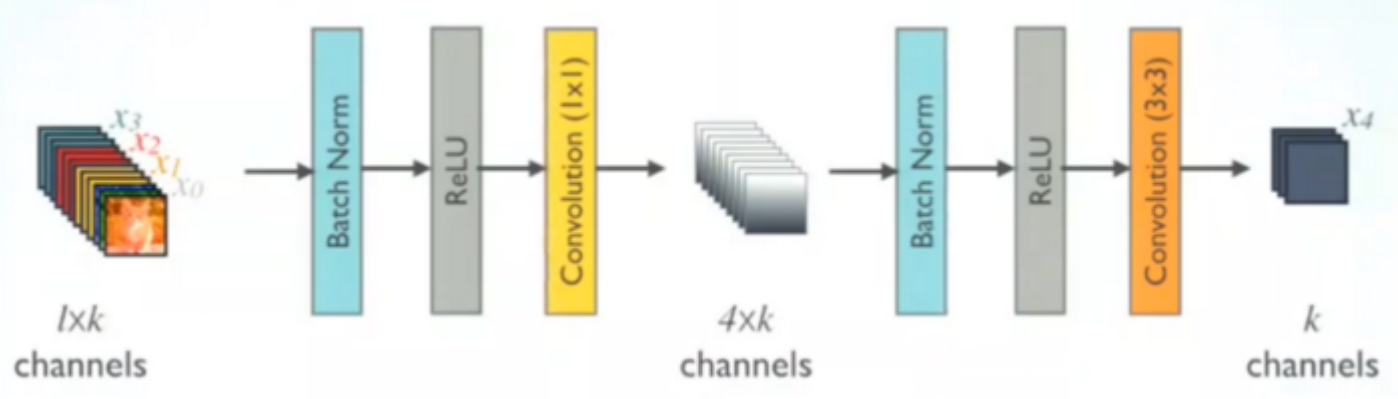
Figure 2: A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature-map sizes via convolution and pooling.

Growth Rate: If each function H_L produces K features, it follows that the L^{th} layer has K_o + K *(L - 1) input feature maps, where K_o is the number of channels in the input layer.

Bottleneck Layer: Although each layer only produces k output feature maps, it typically has many more inputs. We know that a 1x1 convolution can be introduced as a *bottleneck* layer before each 3x3 convolution to reduce the number of input feature maps, and thus to improve

computational efficiency.

Bottleneck Layer in DenseNet is: BN-ReLU-Conv(1x1)-BN-ReLU-Conv(3x3) version of H_L, as DenseNet-B (another network). **In DenseNet, each 1x1 produces 4k feature-maps.**



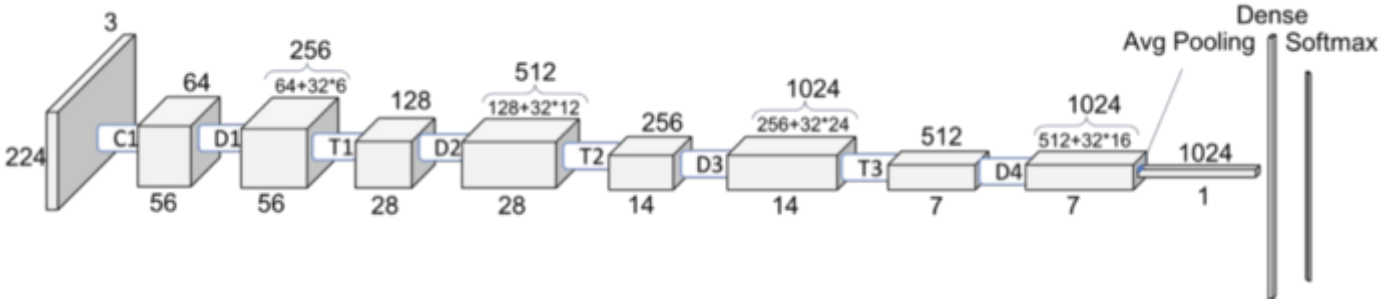
Compression: To further improve model compactness, it further reduces the number of feature maps at the transition layers. If a dense block contains m feature-maps, we can compress it to be between 0 and 1. Network with θ = 0.5 is called DenseNet-C

When both Bottleneck and transition layers are present, it is called DenseNet-BC.

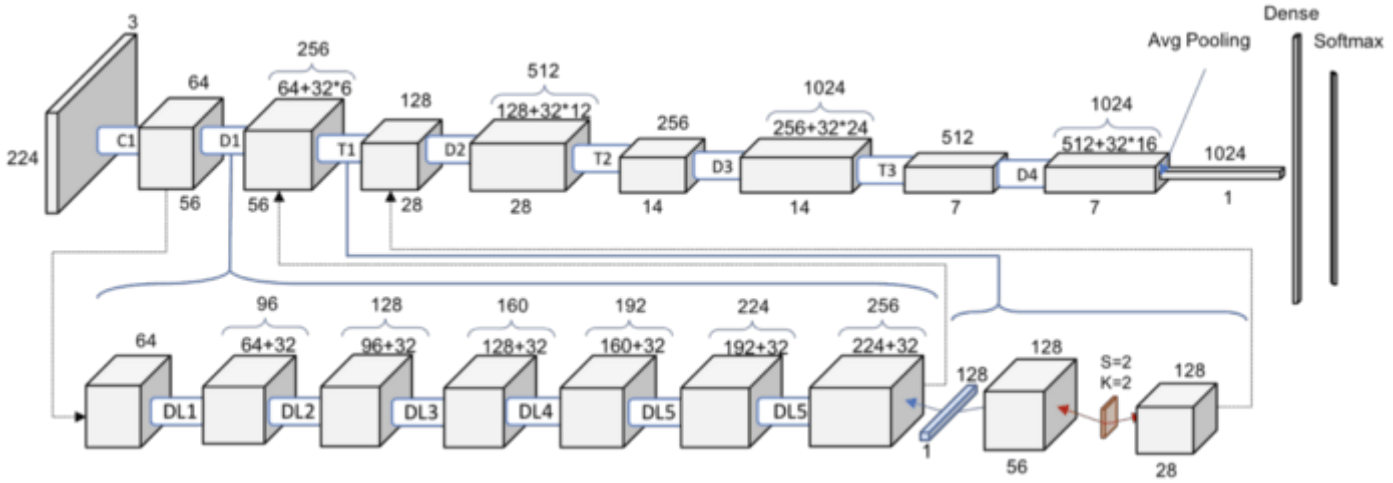
Growth Rate: Since we are concatenating feature maps, the channel dimensions is increasing at every layer. If we make H_L to produce k feature maps every-time, then we can generalize for the l^{th} layer as:

$$k_l = k_0 + k * (l - 1)$$

This hyper-parameter k is the growth rate.



The volume after ever Dense Block increase by the growth rate times the number of Dense Layers within that Dense Block.



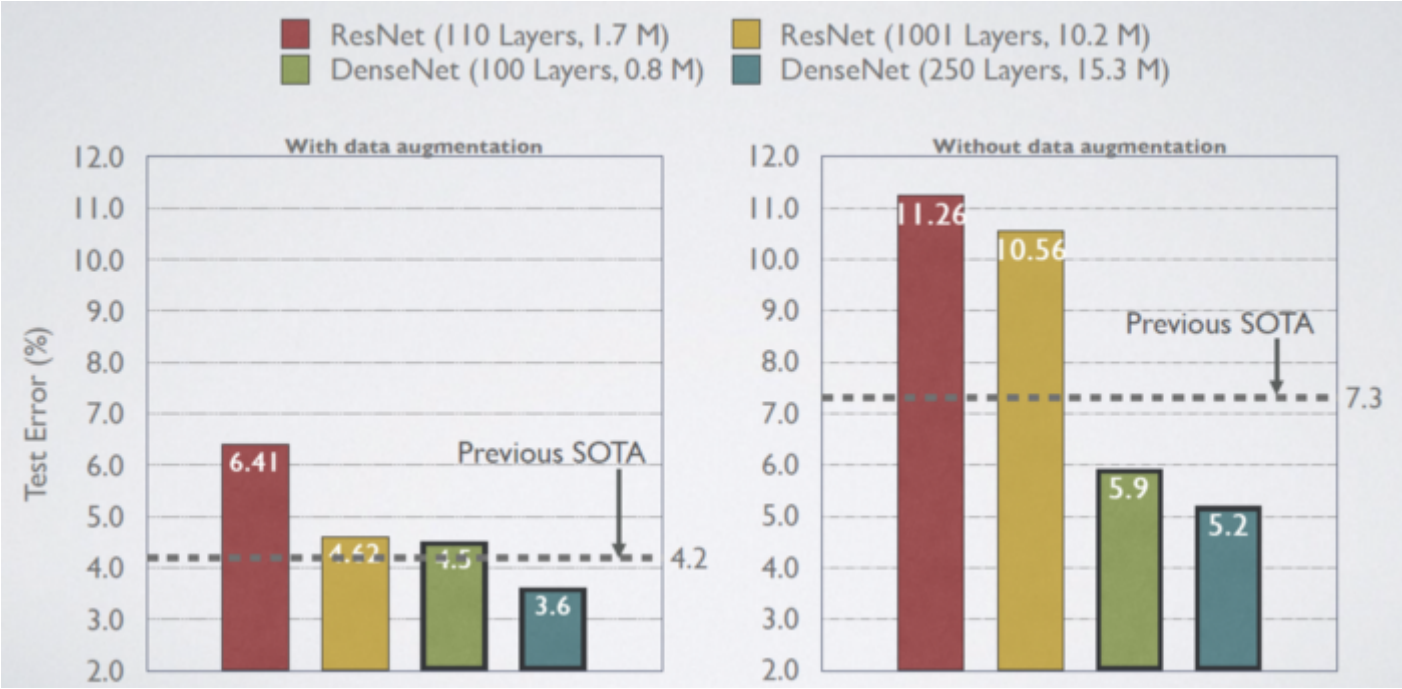
Implementation:

- Except on ImageNet, all other networks has three dense blocks that has an equal number of layers.
- Before entering the first dense block, a convolution with 16 (or twice the growth rate for DenseNet-BC), output channels is performed on the input images
- For conv layers with 3x3 kernels, each side of the input is zero-padded by one-pixel to keep the feature map size fixed.
- 1x1 convolution is followed by 2x2 average pooling in the 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 classified is attached
- For ImageNet:
  - DenseNet-BC is used
  - it has 4 blocks
  - initial layer consists of 2k convoluitions with 7x7 kernel with stride of 2

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
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			

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 the sequence BN-ReLU-Conv.

- CIFAR10
  - it is trained with a batch size of 64, and for 300 epochs
  - Initial learning rate is set to 0.1, and is divided by 10 at 50% and 75% of the total training epochs.
  - Of course SGD is used
- On imagenet,
  - it is trained for 90 epochs with a batch size of 256
  - initial learning rate is set to 0.1, and is lowed by 10 times at epoch 30 and 60.
- weight decay of 10^{-4} and Nesterov momentum of 0.9 is used.



Why is it not used?

- did you notice the batch size?
- Reddit user: Unfortunately DenseNets are extremely memory hungry. On my Titan-X Pascal the best DenseNet model I can run achieves 4.51% accuracy on CIFAR-10 and has only 0.8M parameters, while a 36M Wide ResNet consumes around the same of my card's memory (even though it uses 128 batch size instead of 64), achieving 3.89%. There are some Wide ResNet variants that get to ~3.6% in less than 8 hours training time, while the 3.62% DenseNet (depth = 250, k = 24) requires 2 Titan-Xs to run and even then would take longer to train.
- Understanding why is pretty straightforward. Backprop requires storing all layer's outputs, therefore the number of such layers (and their respective output sizes) is the main culprit for memory bottlenecks. The 37M Wide ResNet (the one that achieves 3.89%) has only 28 conv layers, against 100/250/190 of DenseNets. Things get even worse if you analyze DenseNets with the residual interpretation (shortcut connections to all previous layers instead of concatenation with the previous).
    - **RESPONSE FROM THE AUTHORS ON REDDIT:** One of the reasons why DenseNet is less memory/speed-efficient than Wide ResNet, is that in our paper, we mainly aimed to compare the connection pattern between DenseNets (dense connection) and ResNets (residual connection), so we build our DenseNet models in the same "deep and thin" fashion as the original ResNets (rather than Wide ResNets). Based on our experiments, on GPUs, those "deep and thin" models are usually more parameter-efficient, but less memory/speed-efficient,

compared with "shallow and wide" ones. The reason is already explained by give_me_tensors. If one wants to use a more memory/speed-efficient DenseNet, I recommend trying a "Wide DenseNet", by making it shallow (set the depth to be smaller) and wide (set the growthRate k to be larger).

- **https://www.reddit.com/r/MachineLearning/comments/67fds7/d_how_does_densenet_compare_to_resnet_and/ (https://www.reddit.com/r/MachineLearning/comments/67fds7/d_how_does_densenet_compare_to_resnet_and/)**

Lets look at the code

```
# Dense Block
def add_denseblock(input, num_filter = 12, dropout_rate = 0.2):
    global compression
    temp = input
    for _ in range(l):
        BatchNorm = BatchNormalization()(temp)
        relu = Activation('relu')(BatchNorm)
        Conv2D_3_3 = Conv2D(int(num_filter*compression), (3,3), use_bias=False ,padding='same')(relu)
        if dropout_rate>0:
          Conv2D_3_3 = Dropout(dropout_rate)(Conv2D_3_3)
        concat = Concatenate(axis=-1)([temp,Conv2D_3_3])

        temp = concat

    return temp
```

```
def add_transition(input, num_filter = 12, dropout_rate = 0.2):
    global compression
    BatchNorm = BatchNormalization()(input)
    relu = Activation('relu')(BatchNorm)
    Conv2D_BottleNeck = Conv2D(int(num_filter*compression), (1,1), use_bias=False ,padding='same')(relu)
    if dropout_rate>0:
      Conv2D_BottleNeck = Dropout(dropout_rate)(Conv2D_BottleNeck)
    avg = AveragePooling2D(pool_size=(2,2))(Conv2D_BottleNeck)

    return avg
```

```
def output_layer(input):
    global compression
    BatchNorm = BatchNormalization()(input)
    relu = Activation('relu')(BatchNorm)
    AvgPooling = AveragePooling2D(pool_size=(2,2))(relu)
    flat = Flatten()(AvgPooling)
    output = Dense(num_classes, activation='softmax')(flat)

    return output
```

TFRecords:

tf.data: Fast, flexible, and easy-to-use input pipelines (Tens...

How to load a custom dataset with tf.data [Tensorflow]

Assignment:

- Refer this code: **LINK** **(https://colab.research.google.com/drive/1pFn0wvWOKj93A4_-pjMxsxR96VDyN-NF)**
- Add a file called createTFRecords.py in a folder called "HEHEHAHAHA"
- you need to use this file to create TFRecords data and only use this data.
- you must call a function from this file and pass "CIFAR10" as the input. Later we will expand this to add other data sets
- Submit.
- Total score is 600

VIDEO: