50.039 – Theory and Practice of Deep Learning

Alex

Week 05: some state of the art in plain Classification for vision

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

Takeaway points

at the end of this lecture you should be able to:

- dropout and batchnorm
- VGG – specialties
- googlenet – specialties
- resnet – specialties
- densenet
- inception V3 specialties
- networks used for segmentation and GANs, image captioning and other tasks use similar building blocks (we will go for GANs later)

Takeaway points

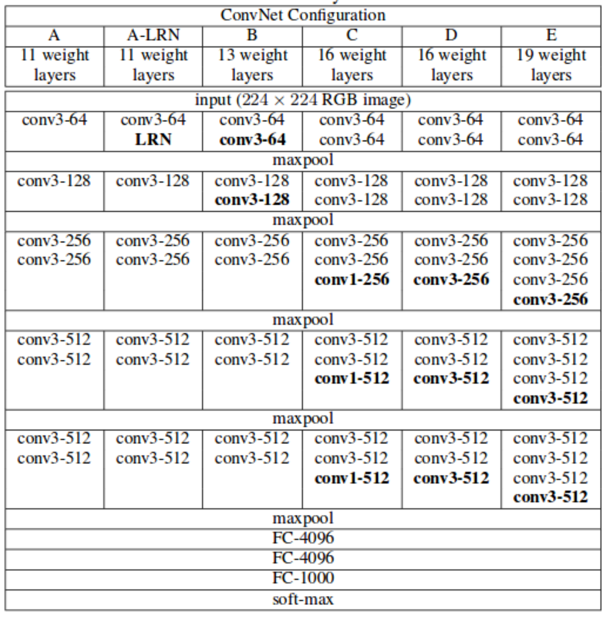
good practices in state of the art models (not only for vision!!!):

- batchnormalization
- residual connections/skip connections
- use an ensemble of models rather than a single model
- stack smaller kernels rather than use a big kernel
- the explanations for why these work well are often rather conceptual (except for ensembles)

1 VGG

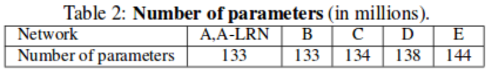
Simonyan & Zisserman, ICLR 2015
`https://arxiv.org/pdf/1409.1556.pdf`

Table 1: **ConvNet configurations** (shown in columns). The depth of the configurations increases from the left (A) to the right (E), as more layers are added (the added layers are shown in bold). The convolutional layer parameters are denoted as "conv⟨receptive field size⟩-⟨number of channels⟩". The ReLU activation function is not shown for brevity.



ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2: **Number of parameters** (in millions).



Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

- contribution: old-style network: repeated blocks of: (convolution-relu)^{*n}-pooling
- only 3x3-convolutions to achieve larger fields of view by stacking
- very large number of parameters: 130 millions!
- 3 fully connected layers contain a large part of the parameters
- dropout layer for less overfitting between fc layers
- 2014 ILSVRC competition second place.

2 Dropout layer

`http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf`
At test time, identity. Has one parameter: keep or drop probability p. At training time: set randomly $1 - p$ of all neurons to zero. A way of adding noise to the learning problem.

2.1 Why does dropout work?

Consider at first bagging (boot strap aggregating), Leo Breiman, 1994:

- Have a dataset of $D_n = \{x_1, \dots, x_n\}$
- train B many models $f_i, i = 1, \dots, n$
  - for each model f_i train on random subset of $d < n$ samples drawn from D_n
  - at test time how to predict?? use average of all B models:

    $$\hat{f}(x) = \frac{1}{B} \sum_{i=1}^{B} f_i(x)$$

  - often used with decision trees / random forest classifiers to prevent them from overfitting.

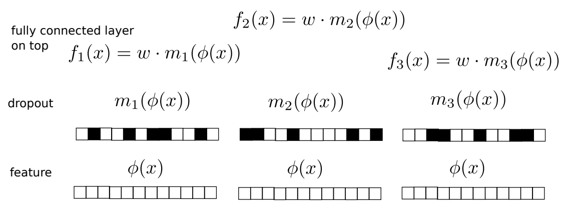
How is dropout different from bagging ??

Looking at subsets of the features of all input sample x, rather than looking at subsets of a dataset.

Lets consider a simple setup (used in a similar way with decisions trees and random forests too) which is similar to dropout

- $\phi(x) = (\phi_1(x), \ldots, \phi_D(x))$ is a the $D-dim$ output of a layer computed over input sample x, to which we want to apply a dropout-like technique
- Let us create B models again, each using a randomly drawn but fixed projection $m_i, i = 1, \ldots, B$ which zeroes out a number of $(1-p)D$ of all the features:

$$m_i(\phi(x)) = \begin{cases} \phi_d(x) & \text{for } d \in S(i) \\ 0 & \text{if } d \notin S(i) \text{ (zeroing out this dimension)} \end{cases}$$



fully connected layer on top
$f_1(x) = w \cdot m_1(\phi(x))$
$f_2(x) = w \cdot m_2(\phi(x))$
$f_3(x) = w \cdot m_3(\phi(x))$
dropout
$m_1(\phi(x))$
$m_2(\phi(x))$
$m_3(\phi(x))$
feature
$\phi(x)$
$\phi(x)$
$\phi(x)$

- we train f_i over the training samples processed by m_i: $\{z_k^{(i)} = m_i(\phi(x_k)), x_k \in$ Training data $\}$
- at test prediction: use average of all B models again

$$\hat{f}(x) = \frac{1}{B}\sum_{i=1}^{B} f_i(x)$$

Why does this work?

- two dimensions of the feature map $\phi_{d_1}(x)$ and $\phi_{d_2}(x)$ may have a correlation which helps to classify sample x on training data

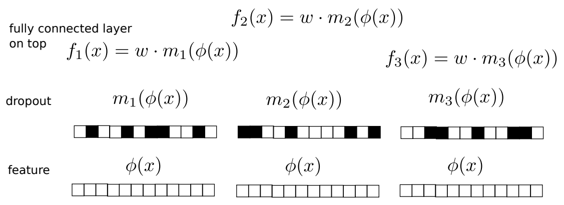
  example: 95% of all the time we have on training data for a pair (x, y) of sample and label:
  $2\phi_{d_1}(x) - \phi_{d_2}(x) > 0$ whenever the label $y > 0$,

  but this correlation is not present in test data – picking up such a correlation results in overfitting.

- setting $\phi_{d_1}(x)$ or $\phi_{d_2}(x)$ to zero, prevents the algorithm from using this non-generalizing correlation

How is dropout different from this setup ?

- at training time: we set of all D neurons $(1-p)D$ of them to zero
- we change the zeroed-out neurons in every minibatch, thus we consider at every minibatch i a different model $f_i(x) = w \cdot m_i(\phi(x))$ rather than a fixed number of B models. Each of the models use the same low level features $\phi(x)$.



fully connected layer on top
$f_1(x) = w \cdot m_1(\phi(x))$
$f_2(x) = w \cdot m_2(\phi(x))$
$f_3(x) = w \cdot m_3(\phi(x))$
dropout
$m_1(\phi(x))$
$m_2(\phi(x))$
$m_3(\phi(x))$
feature
$\phi(x)$
$\phi(x)$
$\phi(x)$

- B fixed deep models is hard to realize, a different model every minibatch is efficiently realizable with deep learning
- at test time we do not have

$$\hat{f}(x) = \frac{1}{B}\sum_{i=1}^{B} f_i(x)$$

instead we know that in expectation every neuron will be active p percent of all the time, so its average output is just $\phi_d(x)$ multiplied with p

$$E[m_i(\phi_d(x))] = p\phi_d(x)$$

We use at test time the expected output to achieve an average of all possible models (including those that were at training time not realized by dropout).

3 Googlenet v1

`https://arxiv.org/abs/1409.4842`
I like to show you in the following that many inventions deal with improving gradient flow.

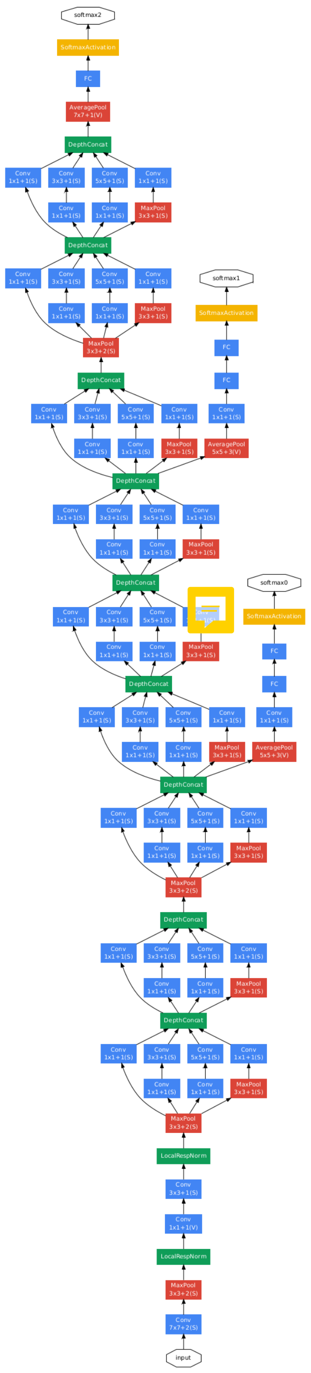
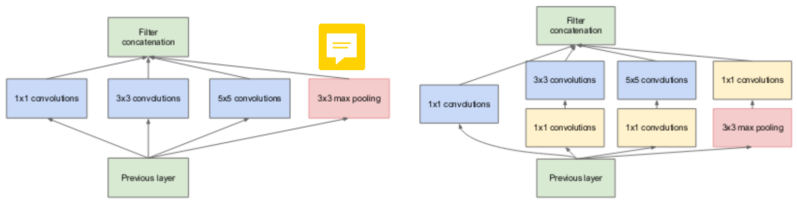


Figure 3: GoogLeNet network with all the bells and whistles

- contribution1: auxiliary output losses – at training time only – for injecting gradient flow in layers close to the bottom.

Auxiliary output is just a separate classification output. At training time a cross entropy loss gets attached. loss to be optimized is weighted sum of main loss and aux losses.

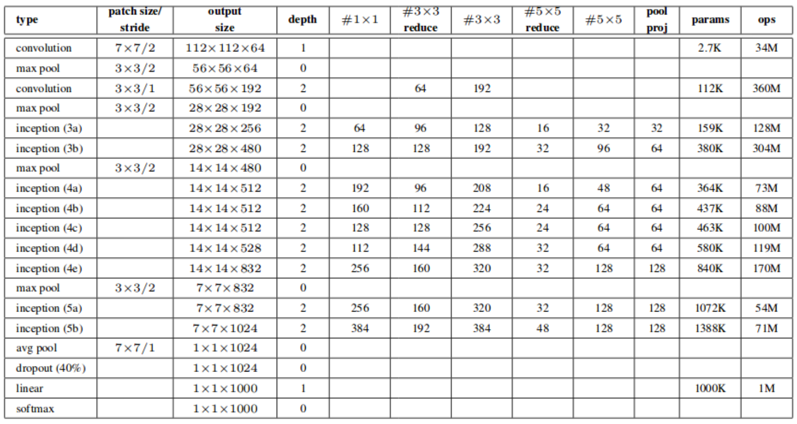
- contribution2: inception module: convolution layers in parallel with different effective kernel sizes – this is classical multi-scale processing



(a) Inception module, naïve version (b) Inception module with dimension reductions

Figure 2: Inception module

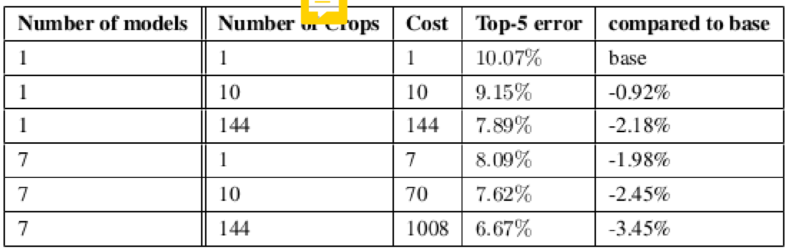
- $1x1$ convolutions to reduce parameters



type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Table 1: GoogLeNet incarnation of the Inception architecture

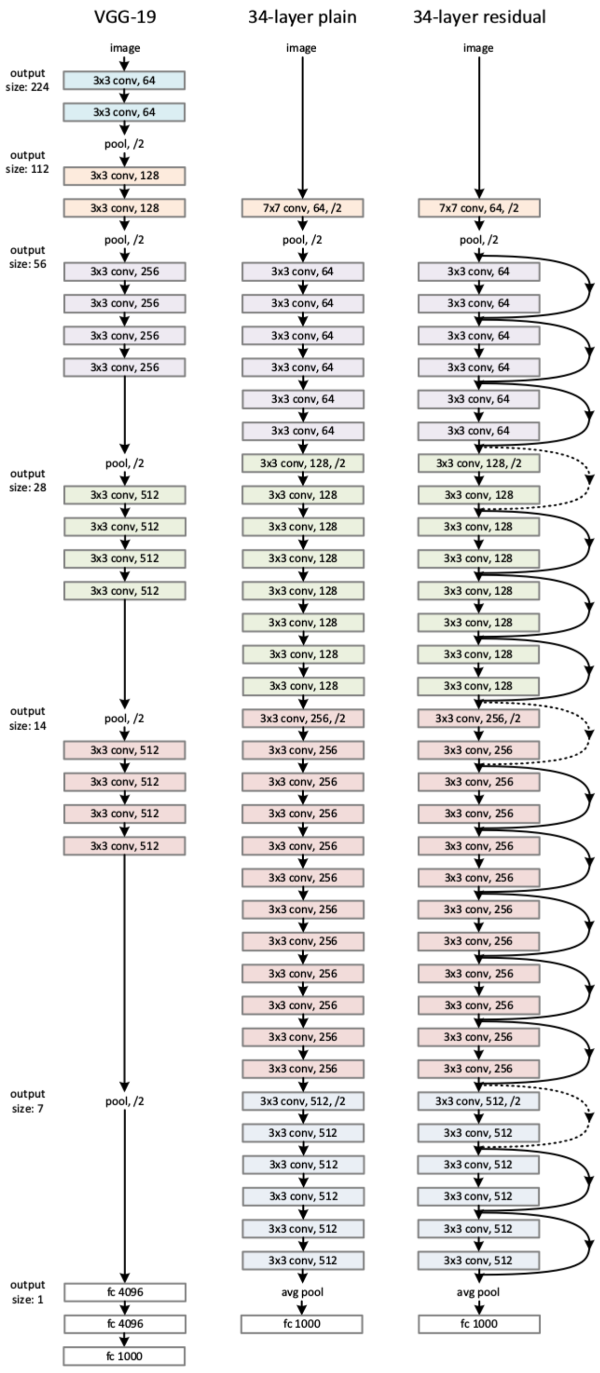
- contribution3: at test time: average over multiple classifiers and massive data augmentation



Number of models	Number of Crops	Cost	Top-5 error	compared to base
1	1	1	10.07%	base
1	10	10	9.15%	-0.92%
1	144	144	7.89%	-2.18%
7	1	7	8.09%	-1.98%
7	10	70	7.62%	-2.45%
7	144	1008	6.67%	-3.45%

Table 3: GoogLeNet classification performance break down

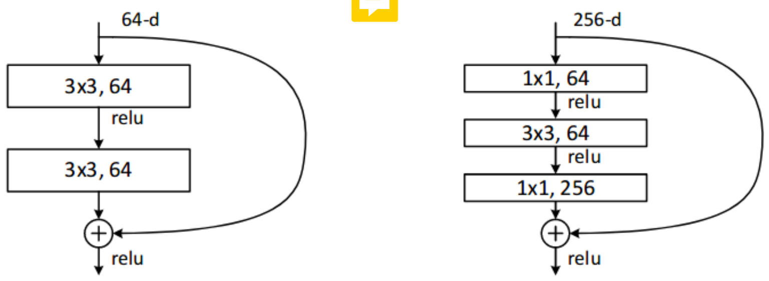
4 Resnets



- only 3x3 and 1x1
- main contribution 1: residual connections – shortcuts across layers (usu-

ally as a linear mapping whenever number of filters changes, not as identity – see option B in Table 4)

- main contribution 2: uses batchnormalization after every convolution
- both contributions are about gradient flow
- in higher layers: once in a while half spatial size of feature maps, double number of filter channels (so that one can a rich set of detectors at higher layers)

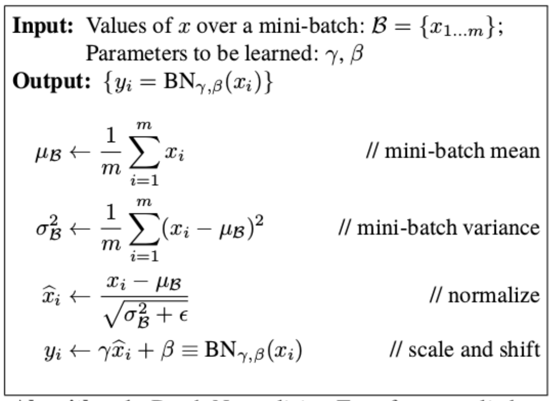


Why do residual connections work ?

- gradient flows as the identity through the shortcut, no vanishing gradient problem
- shortcut in forward pass: inputs feature from previous layer, convolutions across the parallel path can learn additionally non-linear function on top (remember NN lecture on what can be represented...)
- if something stupid was learned during early phases of training, it can be simply undone: set weights of convolution layers to zero, then have only the identity with unhindered information flow forward and backward.
- The latter two effects allow to learn slowly a more complex representation layer by layer: Network can start as: 1 conv layer and one fully connected layer, and (almost) only shortcuts in between. convolution kernels can add layer by layer some nonlinearities.
- later on: compare to the memory cell in LSTM (1998)

5 Batch Normalization

Ioffe and Szegedy, 2015
`https://arxiv.org/pdf/1502.03167.pdf`



```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
       Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                          // mini-batch mean
$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$        // mini-batch variance
$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$                       // normalize
$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$           // scale and shift
```

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Batchnorm at training time performs 2 steps

- take one neuron , normalize its outputs so that – over the all elements in your minibatch have zero mean and standard deviation 1
- in the next step: apply a simple affine transformation on the normalized output $y = ax + b$
- after this output has standard deviation a and mean b
- batchnorm at training time learns to output values which have constant mean and constant standard deviation (computed over the elements in a minibatch)
- convolution layers: compute mean, standard deviation, a, b not for one neuron and all samples in the minibatch but for all elements in the feature map of one channel in a conv layer and all samples in the minibatch – reduces parameters, treats each neuron in the same channel in the same way
- update running mean and running variance
- requires usually a batchsize of 8 at least, better 16 or 32 or more. works not well for less than 8.

Batchnorm at val/test time performs 2 steps:

- take one neuron , normalize its outputs by the running mean and running variance learnt at test time
- apply $y = ax + b$
- important: use `model.eval()` at testing time for your neural network!

Why does batchnorm improve performance?

Equation on page 5 in the paper – Gradient with respect to inputs does not depend on scale of weight anymore. Makes gradient flow more uniform across neurons of a layer.

Group Normalization as newer alternative: Wu & He, ECCV 2018 when cannot use large batchsizes
`http://kaiminghe.com/eccv18gn/group_norm_yuxinwu.pdf`

6 Densenets

Huang, Liu, van der Maaten, Weinberger
`https://arxiv.org/abs/1608.06993`

- resnets to the extreme: within a block of same feature map size ("dense block"), connect each layer with each other layer of the same block with shortcuts.

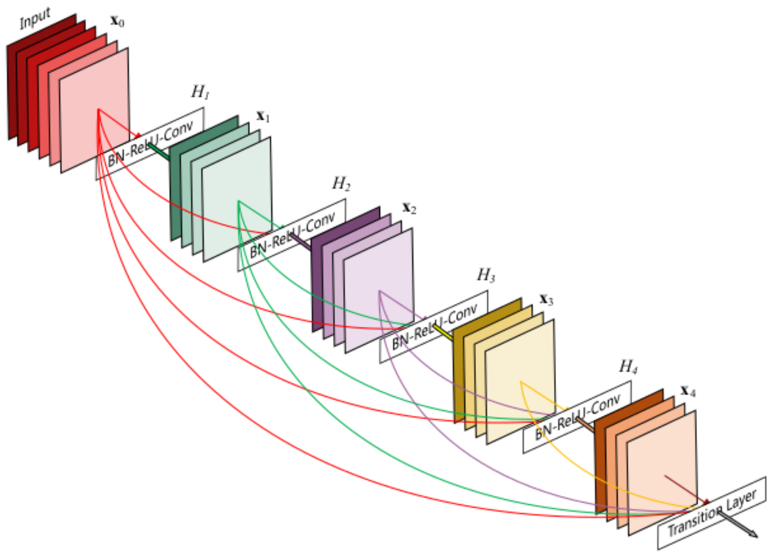


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

The whole net looks like:

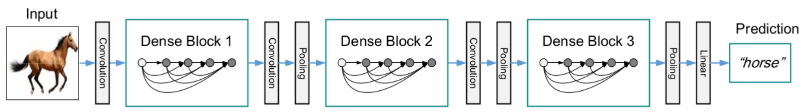
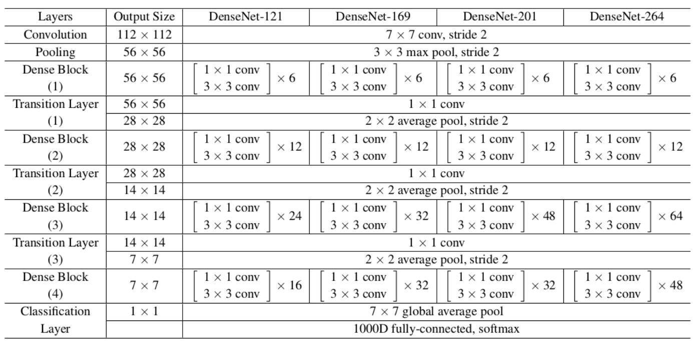


Figure 2: A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature-map sizes via convolution and pooling.



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	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 24$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 48$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 64$
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 16$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 48$
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.

- Important parameter: grow rate - the number of output channels in a convolution layer, typically small like 32
- problem: within a block that starts with k_0 channels, at depth index l one has as inputs $k_0 + (l-1) \cdot growthrate$ many layers. What would be the dimensionality?
- 1x1 convolutions with BN and ReLU before each layer (with $4 \cdot growthrate$ output channels) to reduce parameters in subsequent 3x3 kernels. (Densenet-B)
- in transition layer (where feature map size is downscaled by 1/2): 1x1 conv generates as output channels half the number of incoming maps (Densenet-C)
- low parameter count among the heavier networks, good performance

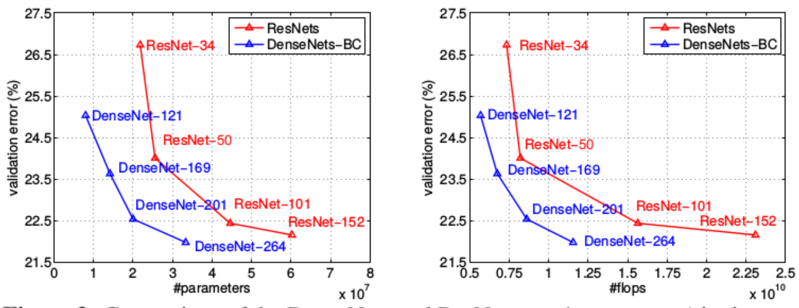


Figure 3: Comparison of the DenseNets and ResNets top-1 error rates (single-crop testing) on the ImageNet validation dataset as a function of learned parameters (*left*) and FLOPs during test-time (*right*).

7 InceptionV3

inception v3: `https://arxiv.org/pdf/1512.00567.pdf`

Some ideas:

- use batchnorm as in resnets

- use one auxiliary loss
- at medium and top layers: factorize filters into low dimensional sequences: instead of a $5x5$ kernel use $3x3 + 3x3$ stacked. New inception module close to the top where one uses $7x1 + 1x7$ convolutions stacked to obtain effectively a $7x7$ field of view

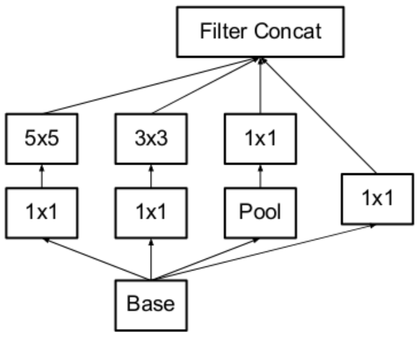


Figure 4. Original Inception module as described in [20].

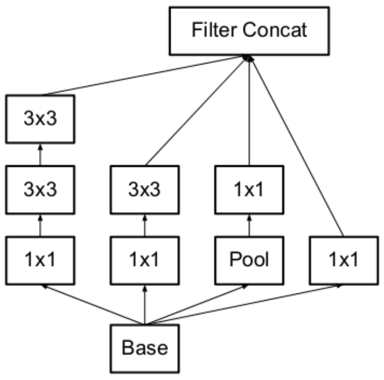
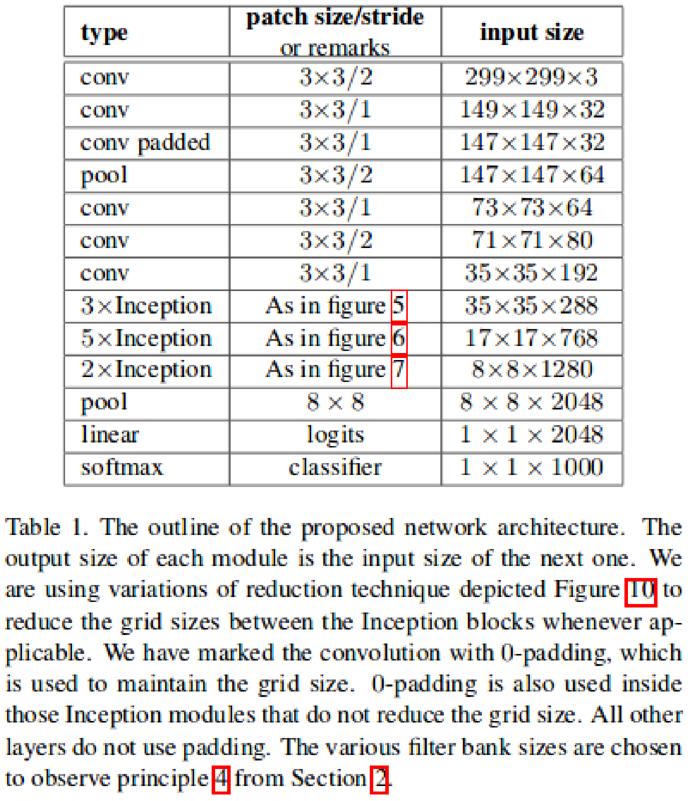


Figure 5. Inception modules where each 5×5 convolution is replaced by two 3×3 convolution, as suggested by principle 3 of Section 2.

- similar idea as in resnets (in higher layers: once in a while half spatial size, double filter channels):



type	patch size/stride or remarks	input size
conv	3×3/2	299×299×3
conv	3×3/1	149×149×32
conv padded	3×3/1	147×147×32
pool	3×3/2	147×147×64
conv	3×3/1	73×73×64
conv	3×3/2	71×71×80
conv	3×3/1	35×35×192
3×Inception	As in figure 5	35×35×288
5×Inception	As in figure 6	17×17×768
2×Inception	As in figure 7	8×8×1280
pool	8 × 8	8 × 8 × 2048
linear	logits	1 × 1 × 2048
softmax	classifier	1 × 1 × 1000

Table 1. The outline of the proposed network architecture. The output size of each module is the input size of the next one. We are using variations of reduction technique depicted Figure 10 to reduce the grid sizes between the Inception blocks whenever applicable. We have marked the convolution with 0-padding, which is used to maintain the grid size. 0-padding is also used inside those Inception modules that do not reduce the grid size. All other layers do not use padding. The various filter bank sizes are chosen to observe principle 4 from Section 2.

keep networks high dimensional closer to the output by having a lot of channels ("disentangled representations") (so that one can a rich set of detectors at higher layers) ... thats why inception modules have so many kernels in parallel

- avoid representation bottlenecks, that is harsh compressions of dimensionality of feature maps $height \times width \times channels$, early in the network (e.g. too much pooling, or too early a 1x1 convolution which does $height \times width \times channels \to height \times width \times 1$).

One consequence is: when one wants to half the feature map size, and double the number of filters $(28, 28, 256) \to (14, 14, 512)$, then do not pool at first (creates a bottleneck).

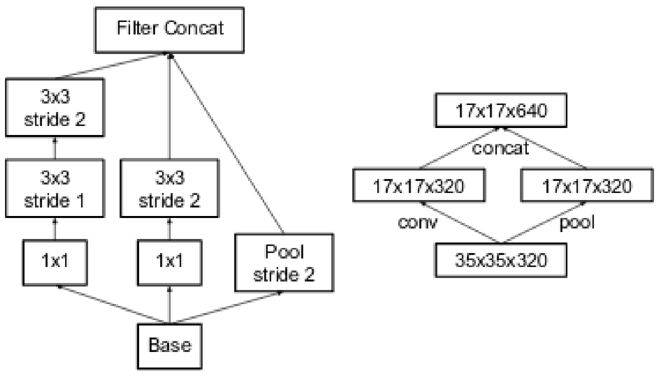


Figure 10. Inception module that reduces the grid-size while expands the filter banks. It is both cheap and avoids the representational bottleneck as is suggested by principle 1. The diagram on the right represents the same solution but from the perspective of grid sizes rather than the operations.

E.g. Fig. 9 left as a "bad" example, Fig. 9 right as better but too costly example, and the solution presented in Fig. 10 as compromise solution

8 out of class: InceptionV4

inception networks styled up as if they are getting married:

v4 is a further modification on v3
`https://arxiv.org/pdf/1602.07261.pdf`

What can be learnt from it:

- residual connections (inceptionresnets) help for a faster convergence
- making layers in the middle and top wider is way to achieve good results without residual connections but it requires much more tuning.

9 out of class: Nasnet

`https://arxiv.org/pdf/1707.07012.pdf`
Neural architecture search.

- First observation: many CNNs have a common structure (see eg. densenet):

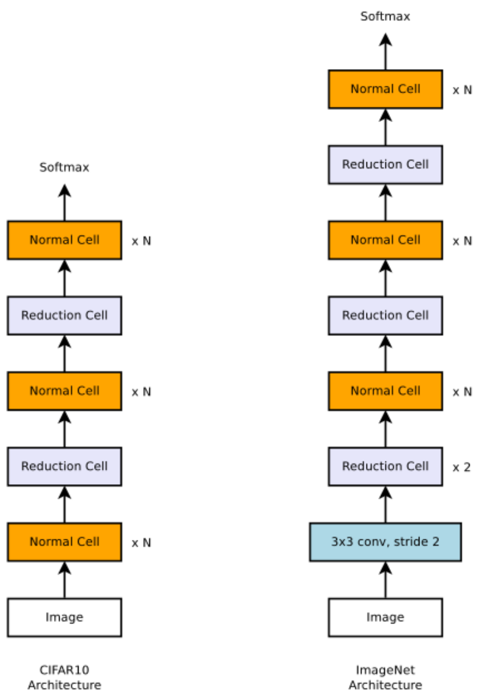


Figure 2. Scalable architectures for image classification consist of two repeated motifs termed *Normal Cell* and *Reduction Cell*. This diagram highlights the model architecture for CIFAR-10 and ImageNet. The choice for the number of times the Normal Cells that gets stacked between reduction cells, N, can vary in our experiments.

– blocks of reduction cells where spatial size of feature maps gets reduced (transition layers in densenet)

– blocks of normal cells (a dense block ... same spatial size of feature maps)

Rough idea: learn to predict the structure of a normal cell and of a reduction cell by reinforcement learning. Once done, predict the number how often these cells are stacked. Then train and observe performance of this model on a validation set.

Reinforcement learning uses real valued rewards instead of ground truth labels to train. Its prediction is: next action given the sequence of past actions and past rewards received. Reward here: performance of the model on a validation data set.
This is slow!

10 Training: learning rate reductions

See eg. the resnet paper. Typically learning rate is reduced by a factor a every K epochs. See pytorch learning rate schedulers to achieve this.

11 other options

other activations if you anyway train from scratch:
sELU `https://arxiv.org/abs/1706.02515` (comes with a modification of dropout),
leaky ReLU activations
a small and clean idea: `https://arxiv.org/pdf/1705.07485.pdf`