CS 6890: Deep Learning

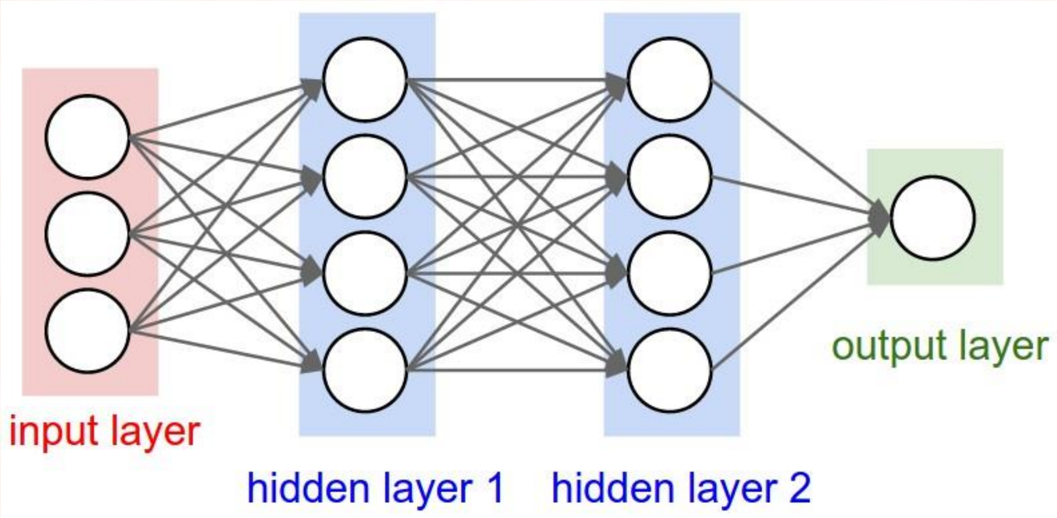
Convolutional Neural Networks

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

Fully Connected Networks



- Problematic when the input is large:
  - MNIST: 28x28
  - CIFAR-10: 32x32
  - STL-10: 64x64 and 96x96
  - ImageNet: 224x224

Fully Connected Networks

- Consider a network with 3 layers:
  1. Input layer: 96x96 pixels
  2. Hidden layer: 100 filters (features).
  3. Output layer: 10 classes softmax.

- Total number of parameters:
  – About 10^6 parameters.
    - 10 times more than MNIST 28x28.

  ⇒ slower feedforward and backpropagation.

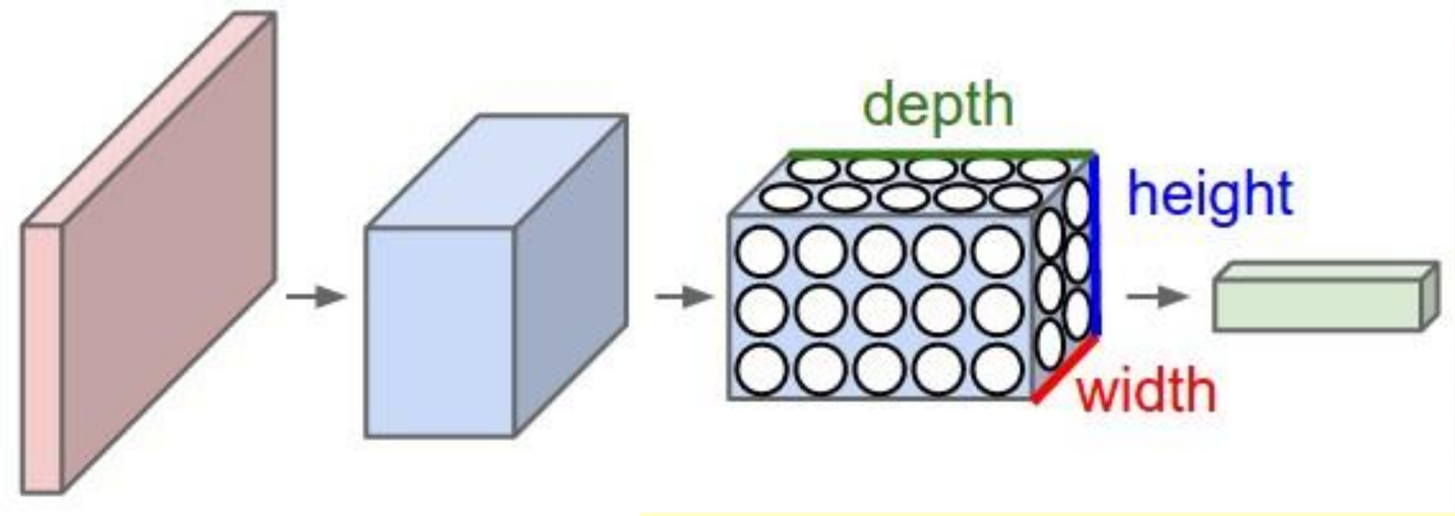
  ⇒ harder to train without overfitting.

Locally Connected Networks

- Restrict the connections between the hidden units and the input units:
  - For **images**, each hidden unit will connect to only a small contiguous (e.g. square) region of pixels in the input.
    - Neurons in the visual cortex have localized **receptive fields**.
  - For **audio**, and **time series** in general, a hidden unit might be connected to only the input units corresponding to a certain time span (e.g. segment).

Locally Connected Networks: Images

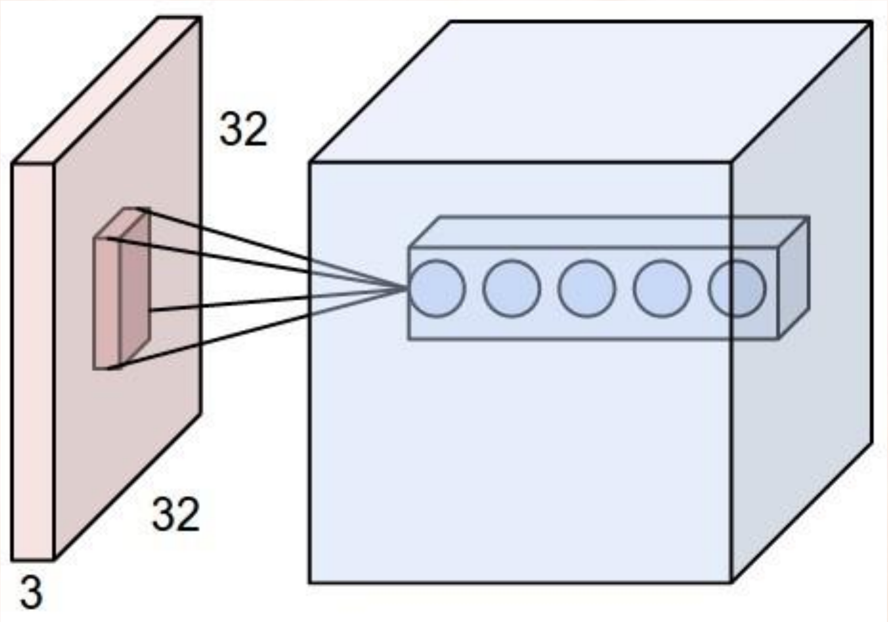
- Neurons in each layer are arranged in 3D:
  - **width** and **height** and **depth**.
  - CIFAR-10: Input layer has W = H = 32, D = 3 (color channels).



http://cs231n.github.io/convolutional-networks/

Locally Connected Networks: Images

- Each neuron is connected only to a local region in the **input volume** spatially, but to the full depth.

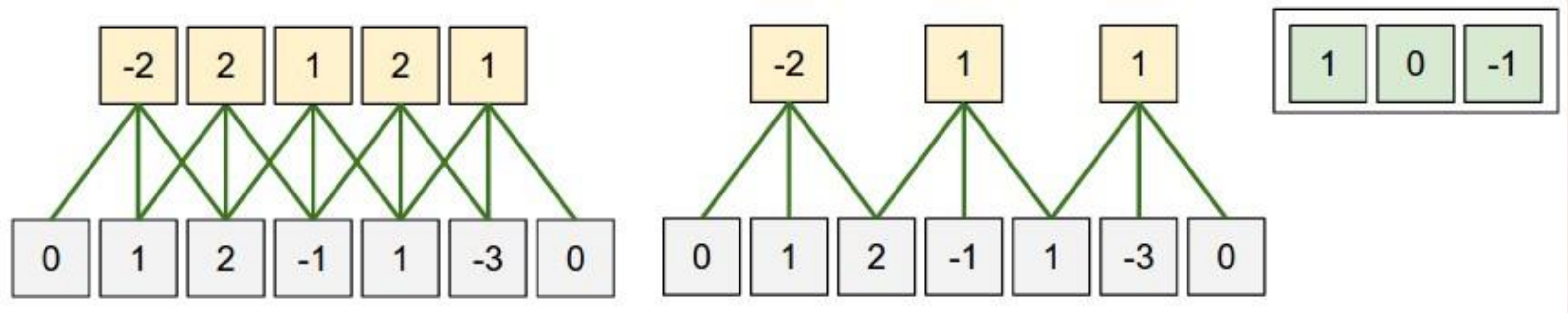


Terminology & Hyperparameters

- Three hyperparams control the size of the **output volume**:
  - **Depth** is the number of filters (features) that are computed on the same region in the input.
  - **Stride** (S) is the step with which we slide the **receptive field** window over the input.
    - When the stride is 1, we move the filters one pixel at a time.
    - When the stride is 2, we move the filters two pixels at a time.
  - **Zero-padding** (P) refers to the number of zeros used to pad around the border of the input volume.
    - Allows to control the *spatial size* of the output volumes.
      - Common is to preserve the width and height of the input.

Locally Connected Networks: Audio

- Neurons in each layer are arranged in 2D:

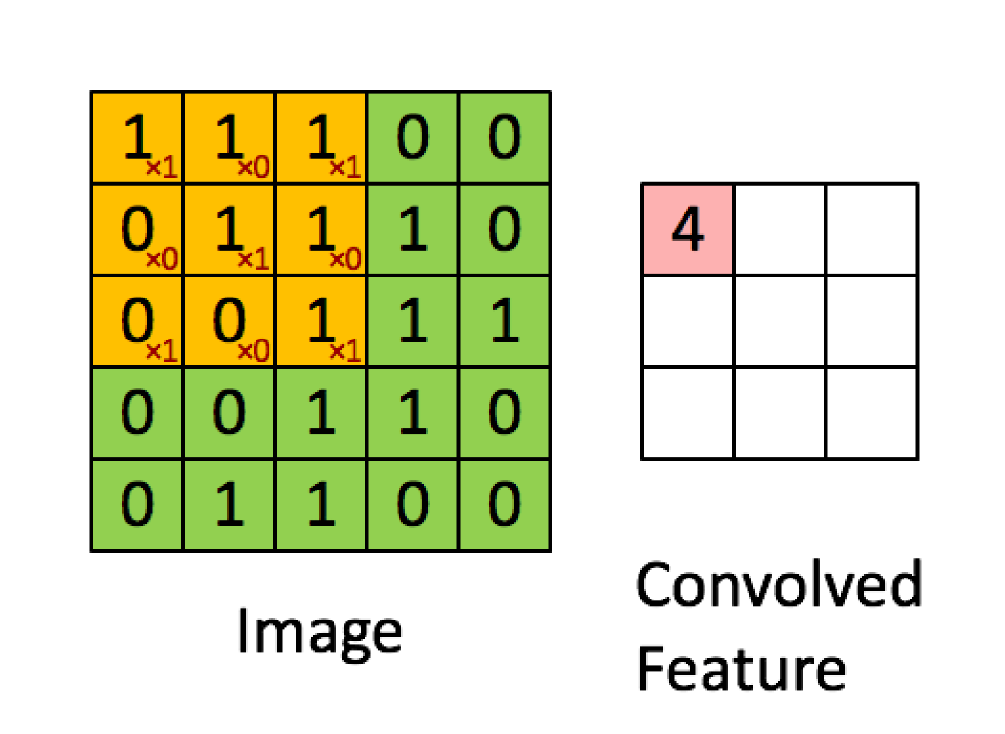


stride S = 1
padding P = 1
receptive field F = 3
depth = 1

stride S = 2
padding P = 1
receptive field F = 3
depth = 1

Locally Connected Networks: Images

- Neurons in each layer are arranged in 3D:
  - Example: **field** F = 3, **depth** = 1, **stride** S = 1, and **padding** P=0



$1_{\times 1}$	$1_{\times 0}$	$1_{\times 1}$	0	0
$0_{\times 0}$	$1_{\times 1}$	$1_{\times 0}$	1	0
$0_{\times 1}$	$0_{\times 0}$	$1_{\times 1}$	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved Feature

Locally Connected Networks: Still too many parameters

- AlexNet architecture (won ImageNet challenge in 2012):
  - Images of size 227 x 227 x 3 (W = 227).
  - First hidden layer has receptive filed size F = 11, stride S = 4, no padding is used P = 0.
    - (W − F + 2P) / S + 1 = (227 − 11) / 4 + 1 = 55 => output volume has **spatial area** of 55 x 55.
    - Depth, i.e. # of filters, is = 96.

    ⇒ **output volume** has size 55 x 55 x 96 = 290,400 neurons.

    - Each neuron is connected to a region of size 11 x 11 x 3 in the input volume => 363 weights + 1 bias.

    => If each neuron had separate params, the first layer would need 290,400 * 364 > 100 million parameters!

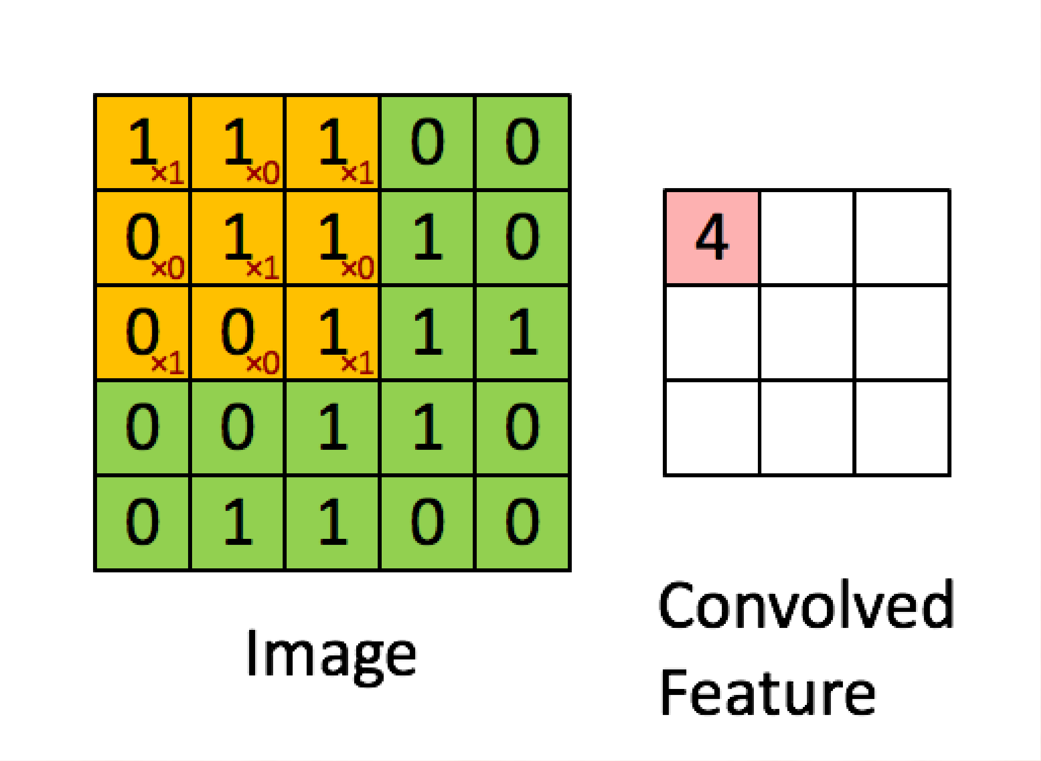
Locally Connected Networks: Parameter Sharing in Convolutional Layers

- Natural images have the property of being **stationary**:
  - The statistics in one part of the image are the same as of any other part.
  - Thus, we can use the same features at all locations.
    - Constrain the neurons in each depth slice to use the same params => run the same **filter** or a **kernel** over all receptive field windows, i.e. **convolve** the filter with the input image.

- AlexNet example:
  - **Output volume** has size 55 x 55 x 96 = 290,400 neurons.
    - There are 96 depth slices (96 filters), each with 55 x 55 neurons:
      - all 55 x 55 have the same 11 x 11 x 3 + 1 = 364 params.

⇒ only 96 * 364 = 34,944 params => a dramatical reduction from 10^8!

Convolution Demo: 1 Channel, 1 Filter



1 ×1	1 ×0	1 ×1	0	0
0 ×0	1 ×1	1 ×0	1	0
0 ×1	0 ×0	1 ×1	1	1
0	0	1	1	0
0	1	1	0	0

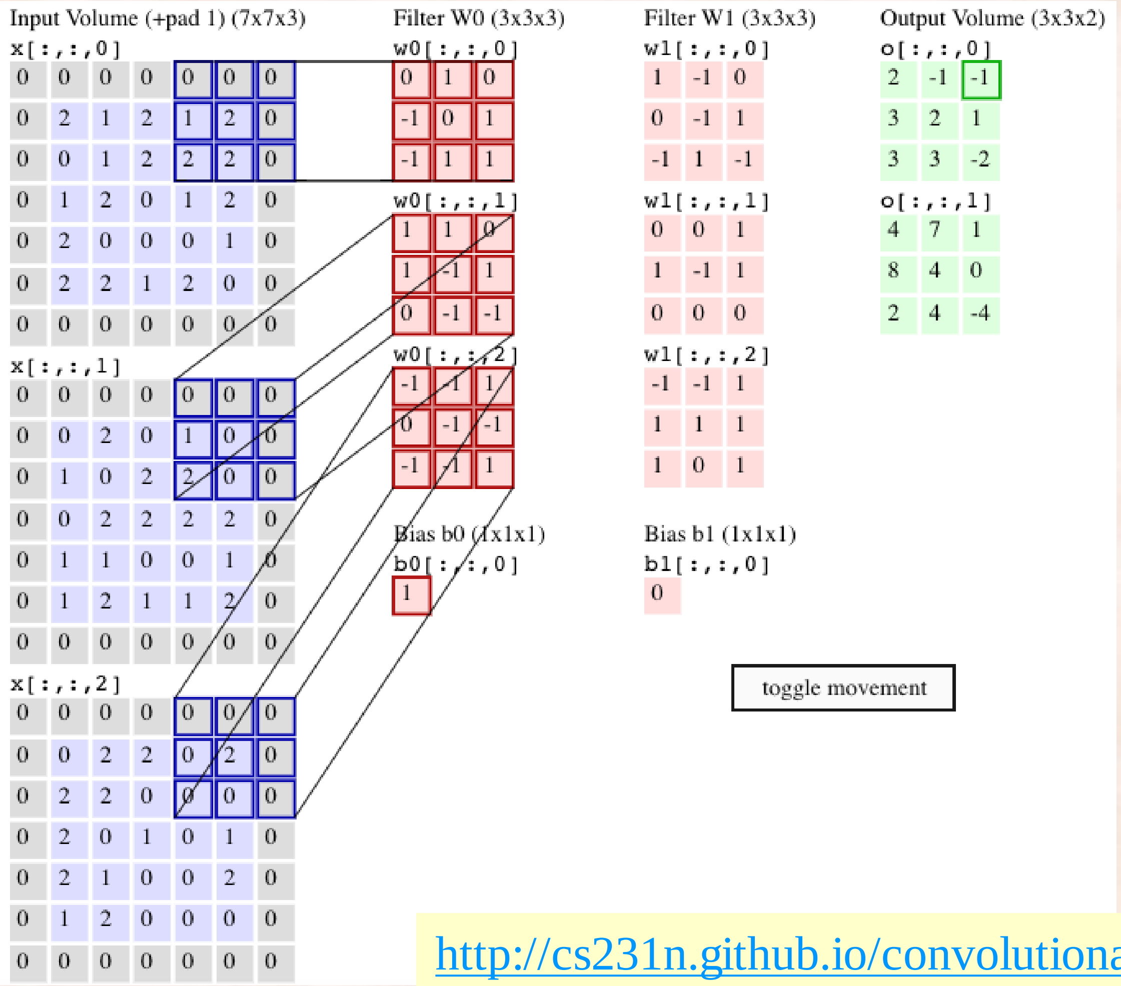
Image

4		

Convolved Feature

http://ufldl.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

Convolution Demo: 3 Channels, 2 Filters



Input Volume (+pad 1) (7x7x3)

x[:,:,0]

0	0	0	0	0	0	0
0	2	1	2	1	2	0
0	0	1	2	2	2	0
0	1	2	0	1	2	0
0	2	0	0	0	1	0
0	2	2	1	2	0	0
0	0	0	0	0	0	0

x[:,:,1]

0	0	0	0	0	0	0
0	0	2	0	1	0	0
0	1	0	2	2	0	0
0	0	2	2	2	2	0
0	1	1	0	0	1	0
0	1	2	1	1	2	0
0	0	0	0	0	0	0

x[:,:,2]

0	0	0	0	0	0	0
0	0	2	2	0	2	0
0	2	2	0	0	0	0
0	2	0	1	0	1	0
0	2	1	0	0	2	0
0	1	2	0	0	0	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

w0[:,:,0]

0	1	0
-1	0	1
-1	1	1

w0[:,:,1]

1	1	0
1	-1	1
0	-1	-1

w0[:,:,2]

-1	-1	1
0	-1	-1
-1	-1	1

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

1	-1	0
0	-1	1
-1	1	-1

w1[:,:,1]

0	0	1
1	-1	1
0	0	0

w1[:,:,2]

-1	-1	1
1	1	1
1	0	1

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

o[:,:,0]

2	-1	-1
3	2	1
3	3	-2

o[:,:,1]

4	7	1
8	4	0
2	4	-4

http://cs231n.github.io/convolutional-networks/

Mathematical Convolution

- Convolution $f * g$ of two functions f and g is:
  - Continuous case:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(t - \tau) g(\tau) d\tau$$

  - Discrete case:

$$(f * g)(t) = \sum_{-\infty}^{+\infty} f(t - \tau) g(\tau) d\tau$$

The weight / importance of value of f computed at:

- τ steps in the past ($\tau \geq 0$)
- τ steps in the future ($\tau < 0$)

Mathematical Convolution

- Discrete case:

$$(f * g)(t) = \sum_{-\infty}^{+\infty} f(t-\tau)g(\tau)d\tau$$

- Examples:

1) Moving average of *f* over the past *K* values:

$$g(\tau) = \begin{cases} \frac{1}{K} & 0 \leq \tau < K \\ 0 & elsewhere \end{cases}$$

2) Exponential moving average of *f* :

$$g(\tau) = \begin{cases} \alpha(1-\alpha)^{\tau} & \tau \geq 0 \\ 0 & elsewhere \end{cases}$$

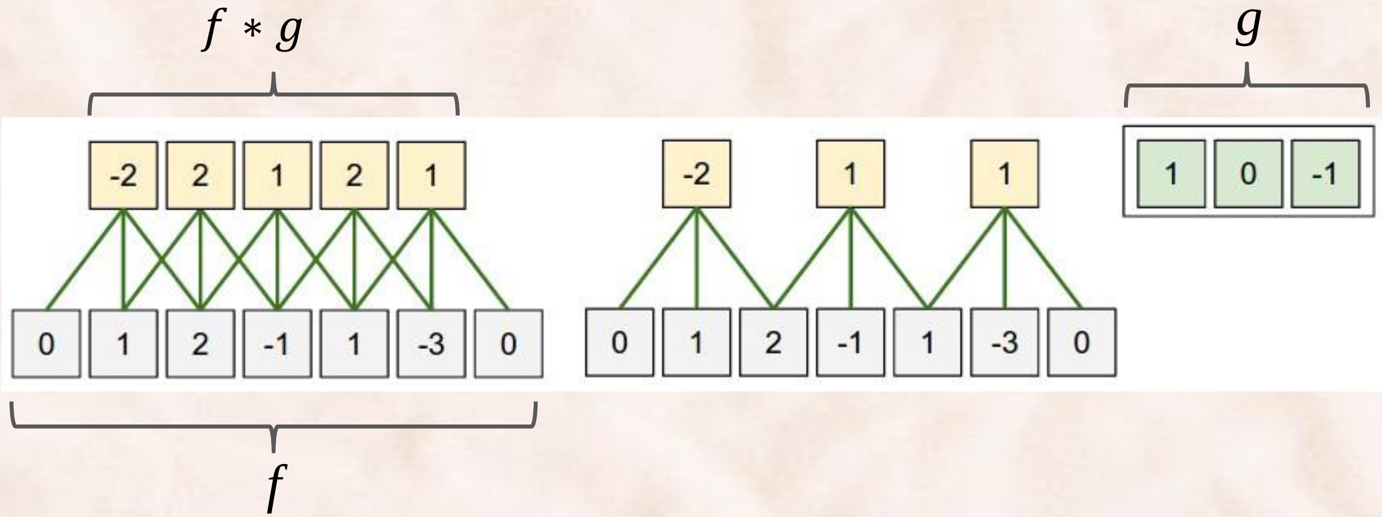
From Mathematical Convolution to CNNs

- Discrete convolution:

$$(f * g)(t) = \sum_{-\infty}^{+\infty} f(t-\tau)g(\tau)d\tau$$

- Assume g is non-zero only within $[-K, K]$:

$$(f * g)(t) = \sum_{-K}^{+K} f(t-\tau)g(\tau)d\tau$$

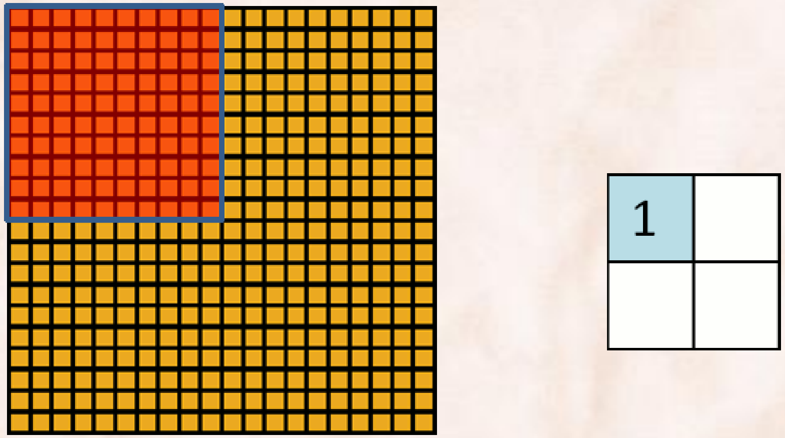


Downsampling with Pooling

- Pooling layers can be inserted between Convolution layers in Deep CNNs.
  - The most common form is a max-pooling layer with a max filter of size 2 x 2 (F = 2) applied with a stride of S = 2:
    - It downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations.
    - The depth dimension remains unchanged.
  - Another common pooling F = 3, S = 2 (overlapping).

- Pooling reduces the spatial size of each depth slice in the output => fewer parameters in higher levels in the network.

Pooling Demo: 10x10 Filters and Stride 10

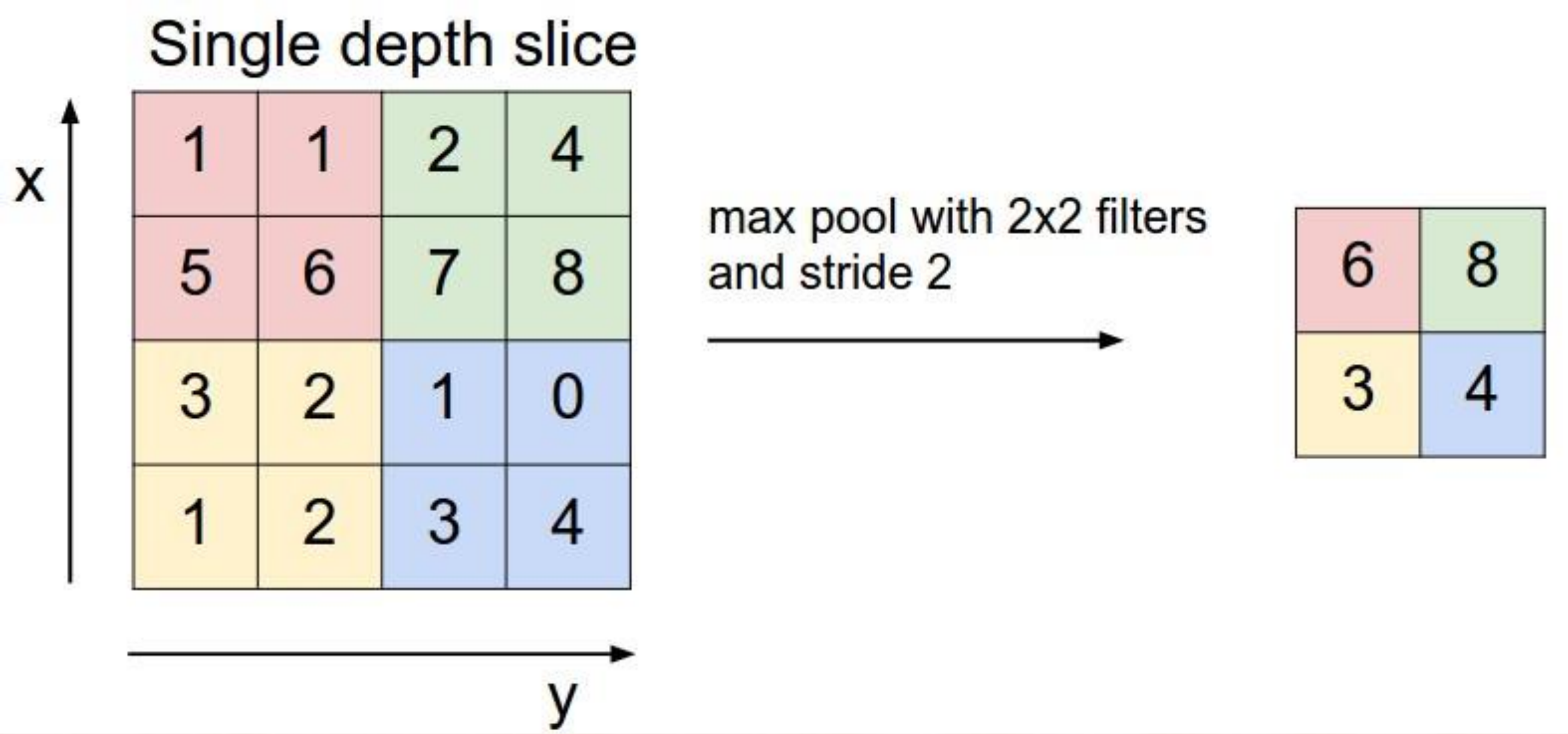


Convolved feature

Pooled feature

http://ufldl.stanford.edu/wiki/index.php/Pooling

Pooling Demo: 2x2 Filters and Stride 2



Single depth slice

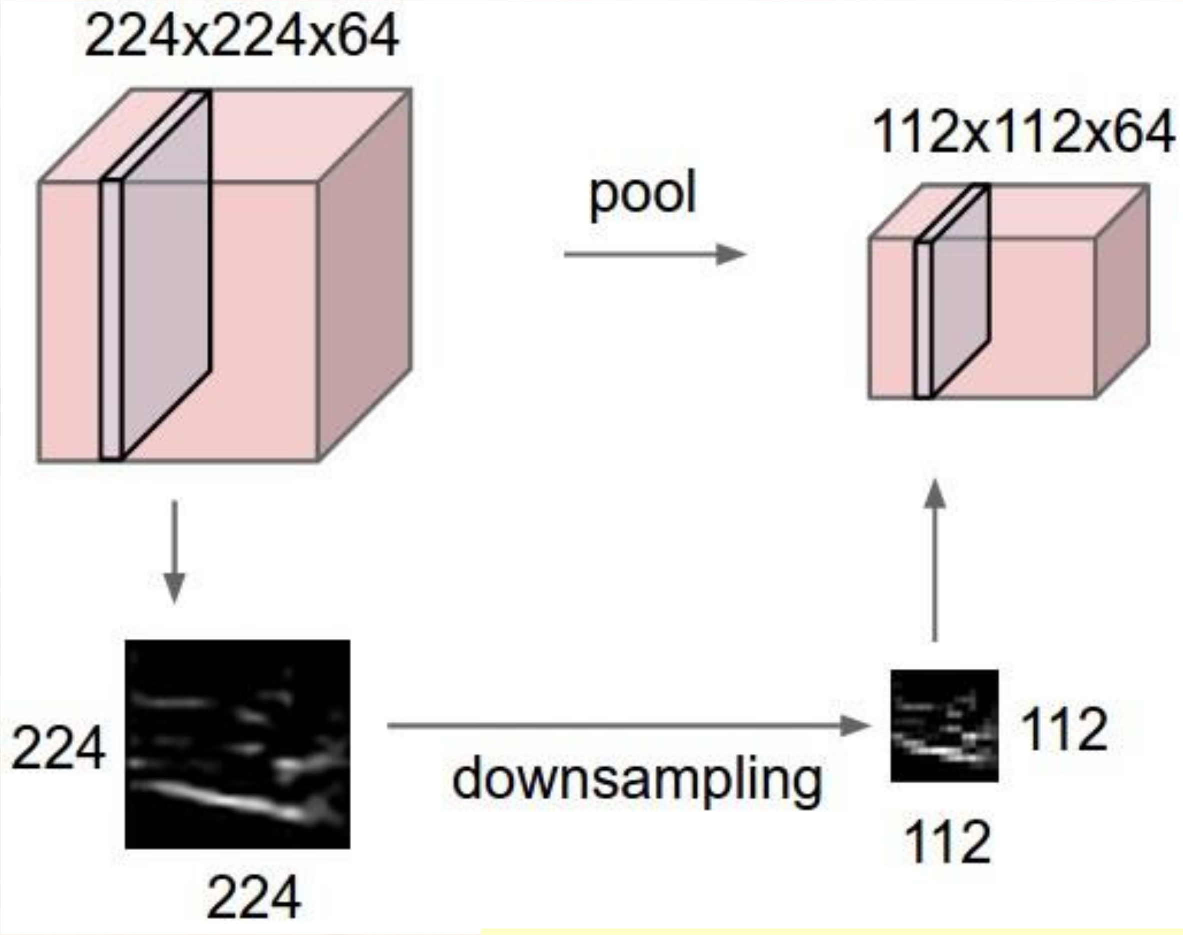
1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

max pool with 2x2 filters and stride 2

6	8
3	4

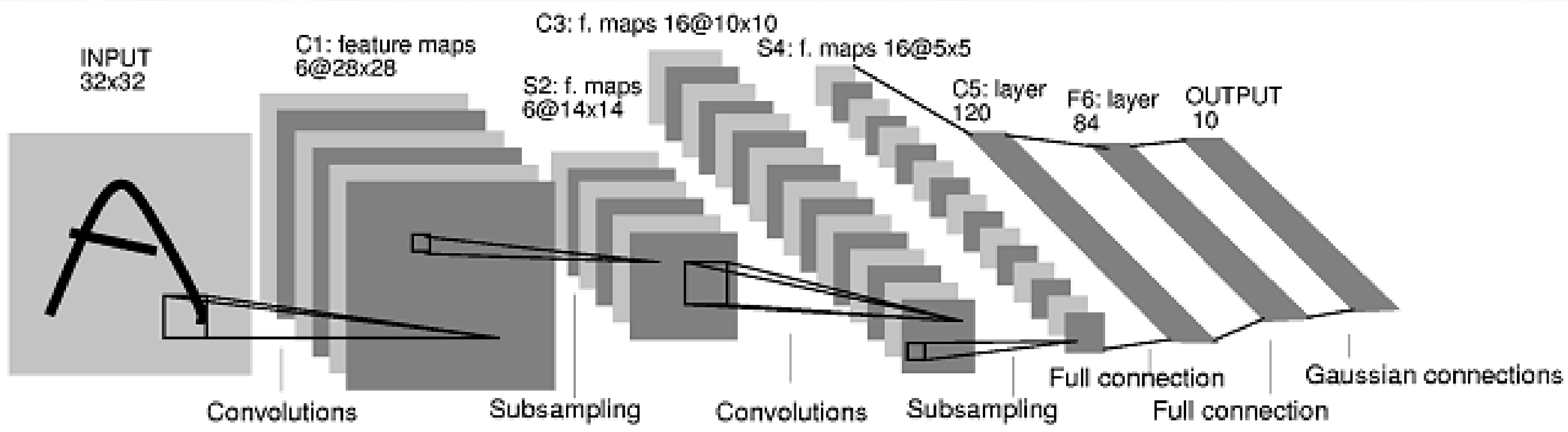
http://cs231n.github.io/convolutional-networks/

Pooling Demo



http://cs231n.github.io/convolutional-networks/

LeNet (1998)



- Average pooling • Sigmoid or tanh nonlinearity • Fully connected layers at the end • Trained on MNIST digit dataset.

Backpropagation Algorithm: FCNs

1. For softmax layer, compute:

$$\delta^{(n_l+1)} = (\mathbf{a}^{(n_l+1)} - \mathbf{y})$$

one-hot label vector

2. For $l = n_l, n_l{-}2, n_l{-}3, \ldots, 2$ compute:

$$\delta^{(l)} = \left(\left(W^{(l)}\right)^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$

3. Compute the partial derivatives of the cost $J(W, b, x, y)$

$$\nabla_{W^{(l)}} J = \delta^{(l+1)} \left(a^{(l)}\right)^T$$

$$\nabla_{b^{(l)}} J = \delta^{(l+1)}$$

CNNs as FCNs

https://arxiv.org/pdf/1603.07285.pdf

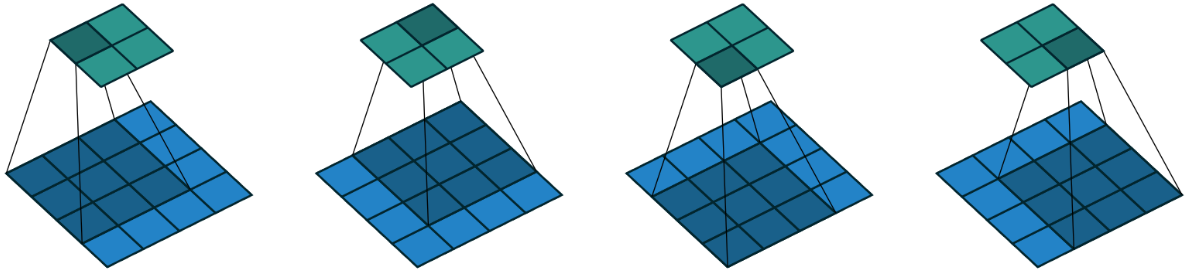


Figure 2.1: (No padding, unit strides) Convolving a 3 × 3 kernel over a 4 × 4 input using unit strides (i.e., $i = 4$, $k = 3$, $s = 1$ and $p = 0$).

- Flatten the input as a 16-dim vector and produces a flattened 4-dim output vector.
- Correspondingly, 3x3 convolution kernel is represented as a sparse matrix **C** where non-zero elements are the elements $w_{i,j}$ of the kernel:

$$\begin{pmatrix} w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 & 0 \\ 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} \end{pmatrix}$$

- Error is backpropagated by multiplying with $\mathbf{C}^T$.

Backpropagation Algorithm: CNNs

- The δ terms (2D) are computed similarly, one for each kernel k:

$$\delta_k^{(l)} = \left(\left(W^{(l)}\right)^T \delta_k^{(l+1)}\right) \bullet f'\left(z_k^{(l)}\right)$$

the shape of the feature map for filter k

- When pooling is used, need to also *upsample*:
  - Propagate the error through the pooling layer by calculating the error w.r.t to each unit incoming to the pooling layer.
    - Mean pooling: uniformly distributes the error for a single pooling unit among the units which feed into it in the previous layer.
    - Max pooling: the unit which was chosen as the max receives all the error since very small changes in input would perturb the result only through that unit.

$$\delta_k^{(l)} = \text{upsample}\left(\left(W^{(l)}\right)^T \delta_k^{(l+1)}\right) \bullet f'\left(z_k^{(l)}\right)$$

Backpropagation Algorithm: CNNs

- The gradient for filter k is computed by convolving the previous layer activations $a^{(l)}$ with the *flipped* error map δ_k:

$$\nabla_{W_k^{(l)}} J = a^{(l)} * flip\left(\delta_k^{(l+1)}\right) \qquad \nabla_{b_k^{(l)}} J = \sum_{i,j} \left(\delta_k^{(l+1)}\right)_{i,j}$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \xrightarrow{flip} \begin{pmatrix} 7 & 4 & 1 \\ 8 & 5 & 2 \\ 9 & 6 & 3 \end{pmatrix}$$

- Need convolution (sum) because same weight is used on multiple inputs:
  - Example: $a^{(l)}$ is 4 x 4, receptive field is 3 x 3:
    - What is the dimensionality of $a^{(l+1)}$ and $W_k^{(l)}$
      - What if $a^{(l)}$ has more than 1 channel?
    - Show contribution of $\left(W_k^{(l)}\right)_{1,1}$

Batch Normalization: Reducing Internal Covariate Shift [Ioffe & Szegedy, ICML'15]

- **Internal Covariate Shift**:
  - The distribution of each layer's inputs changes during training:
    - because the parameters of the previous layers change.
  - This slows down the training:
    - requires lower learning rates and careful param initialization.
    - notoriously hard to train models with saturating nonlinearities.

- **Batch Normalization**:
  - Normalize the input for each layer, for each training minibatch:
    - Allows for much higher learning rates, init. less important.
    - Acts as a rgularizer, eliminating the need for Dropout.

Batch Normalization

[Ioffe & Szegedy, ICML'15]

1. Normalize each activation *x* using minibatch μ and σ.

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

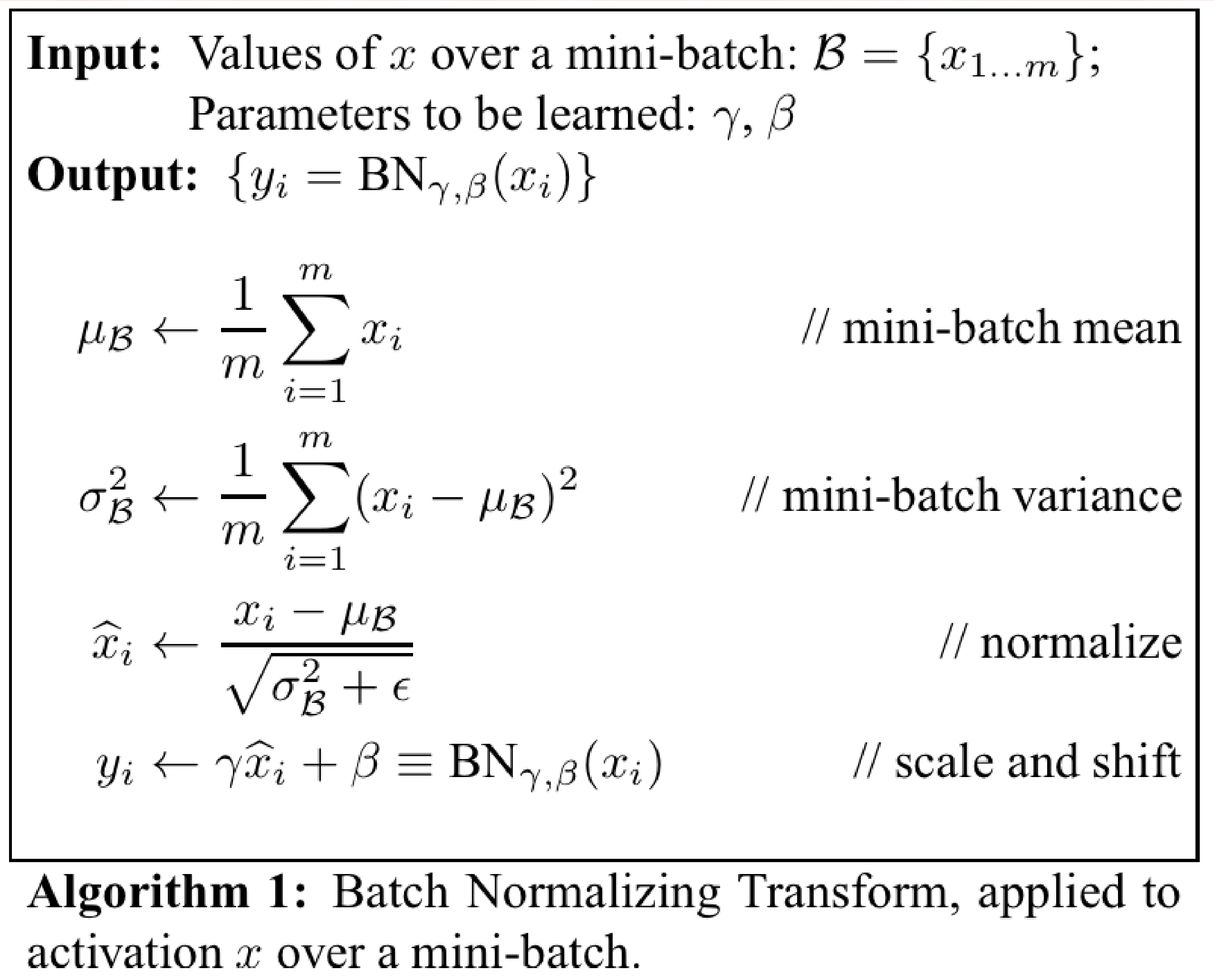
2. Train parameters that scale (γ) and shift (β) the normalized value:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

   – Thus allow the overall transformation to represent the identity transform.

Batch Normalization

[Ioffe & Szegedy, ICML'15]



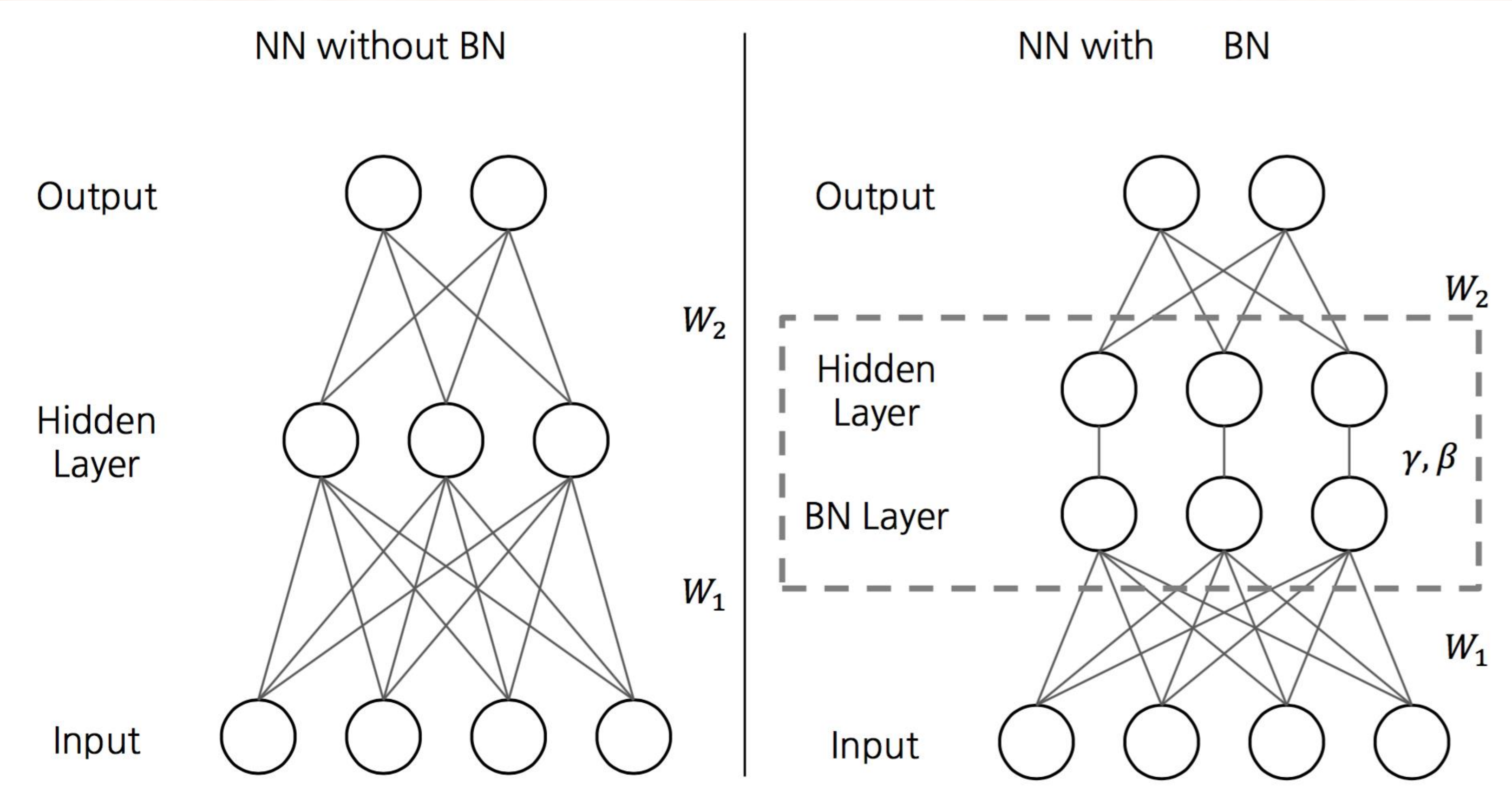
```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1 \ldots m}\}$;
       Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

    $\mu_\mathcal{B} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i$    // 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
    $y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$    // scale and shift
```

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

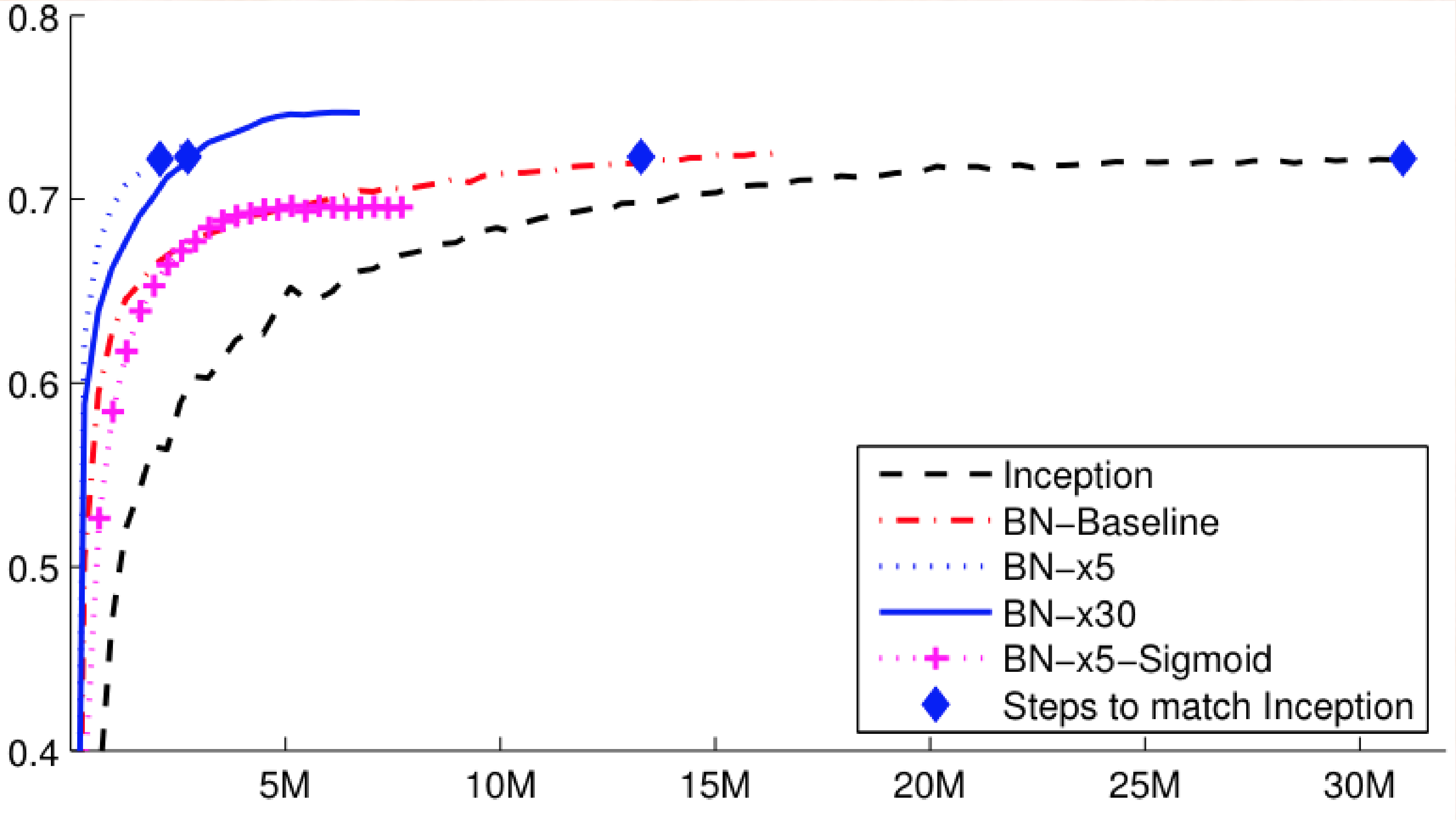
Batch Normalization

[Ioffe & Szegedy, ICML'15]



Batch Normalization

[Ioffe & Szegedy, ICML'15]



Layer Normalization

[Ba, Kiros & Hinton, 2016]

- **Batch Normalization**:
  - The effect of is dependent on the mini-batch size.
  - Not obvious how to apply it to recurrent neural networks.

- **Layer Normalization**: Fix the mean and the variance of the summed inputs within each layer:
  - Compute the mean and variance used for normalization from all of the summed inputs to the neurons in a layer on a single training example.
  - [Like BN] Give each neuron its own adaptive bias and gain which are applied after the normalization but before the non-linearity.
  - [Unlike BN] Layer normalization performs exactly the same computation at training and test times.

Layer Normalization

[Ba, Kiros & Hinton, 2016]

1. Compute layer normalization statistics over all hidden units in the same layer:

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$$

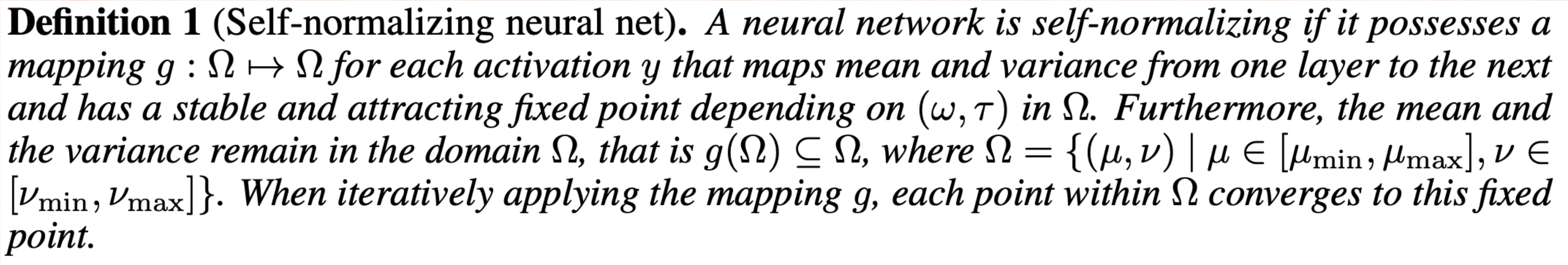
2. Learn an adaptive bias *b* and gain *g* for each neuron after the normalization:

$$h_i = f\left(\frac{g_i}{\sigma_i}\left(a_i - \mu_i\right) + b_i\right)$$

3. BN better than LN for CNNs, LN works well for RNNs.

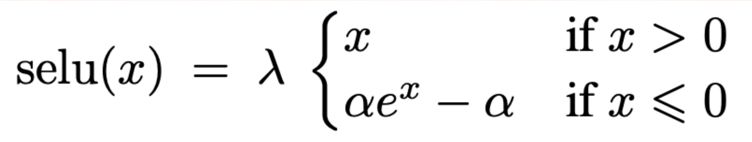
Self-Normalizing Neural Nets

[Klambauer et al., NIPS 2017]



Definition 1 (Self-normalizing neural net). *A neural network is self-normalizing if it possesses a mapping $g : \Omega \mapsto \Omega$ for each activation y that maps mean and variance from one layer to the next and has a stable and attracting fixed point depending on (ω, τ) in Ω. Furthermore, the mean and the variance remain in the domain Ω, that is $g(\Omega) \subseteq \Omega$, where $\Omega = \{(\mu, \nu) \mid \mu \in [\mu_{\min}, \mu_{\max}], \nu \in [\nu_{\min}, \nu_{\max}]\}$. When iteratively applying the mapping g, each point within Ω converges to this fixed point.*

- Use Scaled Exponential Linear Units (SELU) to make an FNN self-normalizing:



$$\mathrm{selu}(x) \; = \; \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^{x} - \alpha & \text{if } x \leqslant 0 \end{cases}$$

- Use 0 and 1 as the first and second order moment for distribution to initialize weights in higher layer.

Self-Normalizing Neural Nets

[Ba, Kiros & Hinton, 2016]

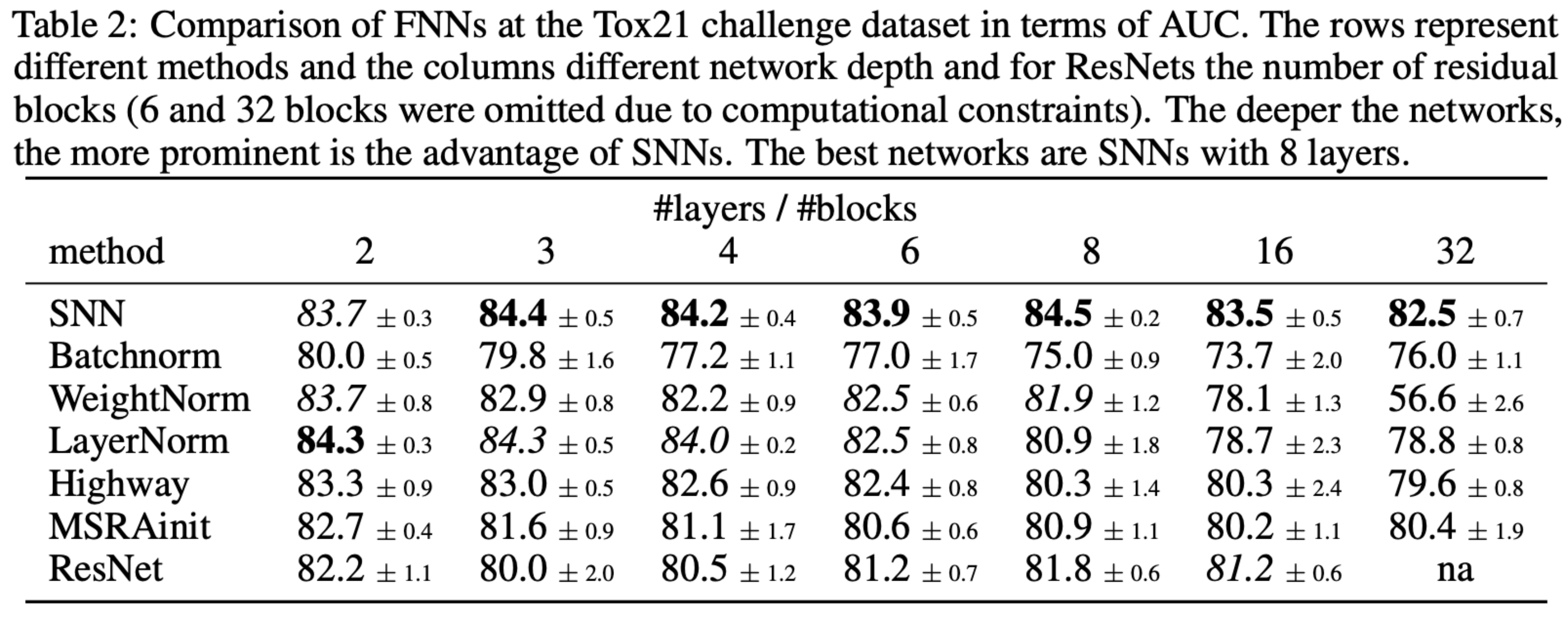


Table 2: Comparison of FNNs at the Tox21 challenge dataset in terms of AUC. The rows represent different methods and the columns different network depth and for ResNets the number of residual blocks (6 and 32 blocks were omitted due to computational constraints). The deeper the networks, the more prominent is the advantage of SNNs. The best networks are SNNs with 8 layers.

	#layers / #blocks						
method	2	3	4	6	8	16	32
SNN	*83.7* ± 0.3	**84.4** ± 0.5	**84.2** ± 0.4	**83.9** ± 0.5	**84.5** ± 0.2	**83.5** ± 0.5	**82.5** ± 0.7
Batchnorm	80.0 ± 0.5	79.8 ± 1.6	77.2 ± 1.1	77.0 ± 1.7	75.0 ± 0.9	73.7 ± 2.0	76.0 ± 1.1
WeightNorm	*83.7* ± 0.8	82.9 ± 0.8	82.2 ± 0.9	*82.5* ± 0.6	*81.9* ± 1.2	78.1 ± 1.3	56.6 ± 2.6
LayerNorm	**84.3** ± 0.3	*84.3* ± 0.5	*84.0* ± 0.2	*82.5* ± 0.8	80.9 ± 1.8	78.7 ± 2.3	78.8 ± 0.8
Highway	83.3 ± 0.9	83.0 ± 0.5	82.6 ± 0.9	82.4 ± 0.8	80.3 ± 1.4	80.3 ± 2.4	79.6 ± 0.8
MSRAinit	82.7 ± 0.4	81.6 ± 0.9	81.1 ± 1.7	80.6 ± 0.6	80.9 ± 1.1	80.2 ± 1.1	80.4 ± 1.9
ResNet	82.2 ± 1.1	80.0 ± 2.0	80.5 ± 1.2	81.2 ± 0.7	81.8 ± 0.6	*81.2* ± 0.6	na

CNN Architectures

- CNNs are commonly made up of 3 layer types:
  - Convolution.
  - [Max/Avg Pooling]
    - *We find that max-pooling can simply be replaced by a convolutional layer with increased stride without loss in accuracy …* [Striving for Simplicity: The All Convolutional Net, Springenberg et al., ICLR 2015]
  - Fully Connected.

- Common architecture:
  1. Stack a few Conv-ReLU layers [followed by a Pool layer].
  2. Repeat pattern until image is reduced to a small representation.
  3. Transition to one or more FC-ReLU layers.
  4. The last FC layer computes the output.

Data Augmentation

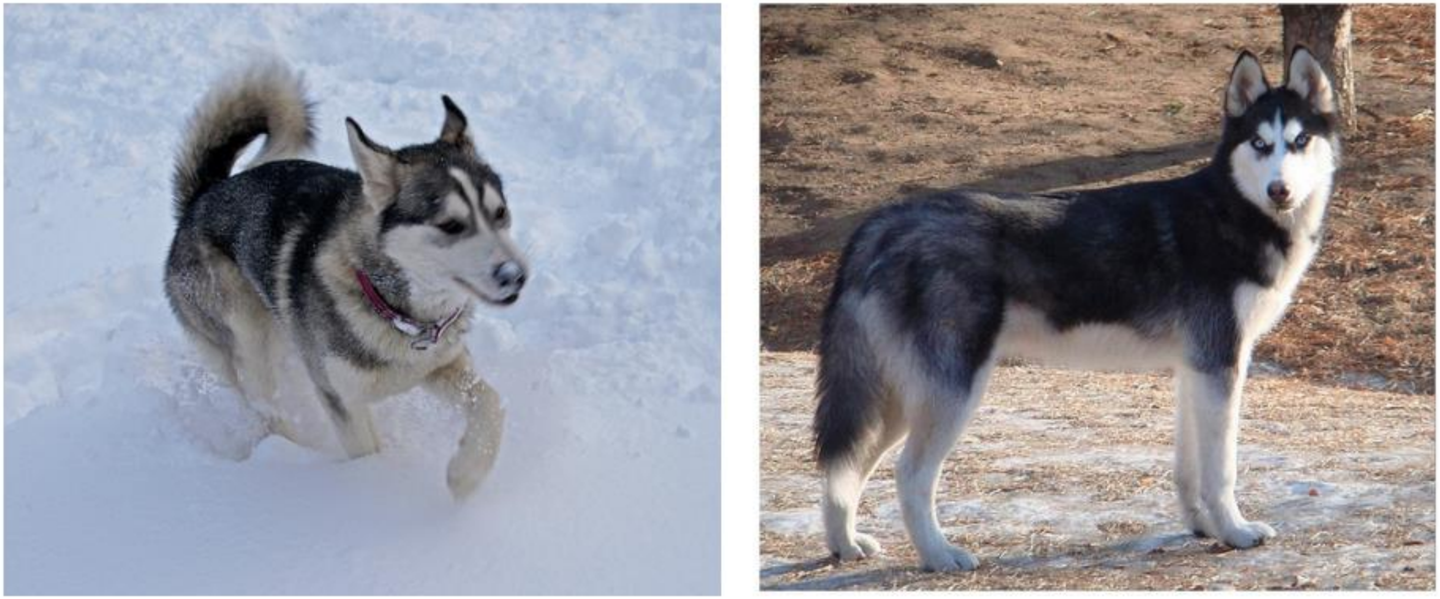
- Apply a series of (random) distortions to artificially increase the data set size:
  - Randomly flip the image from left to right.
  - Randomly distort the image brightness.
  - Randomly distort the image contrast.
  - Displace each training image by a single pixel, either:
    - up one pixel, down one pixel, left one pixel, or right one pixel.

ImageNet ILSVRC:
Large-Scale Visual Recognition Challenge

http://www.image-net.org/challenges/LSVRC

- 1.28 training images, in 1000 object categories.
- Human top-5 accuracy in the 5-10% range.

http://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet



(a) Siberian husky

(b) Eskimo dog

Figure 1: Two distinct classes from the 1000 classes of the ILSVRC 2014 classification challenge.

AlexNet (2012)

[Krizhevsky et al., NIPS’12]

- Top 5 error of 16% compared to runner-up with 26% error!
  - Sparked current commercial interest in deep learning.

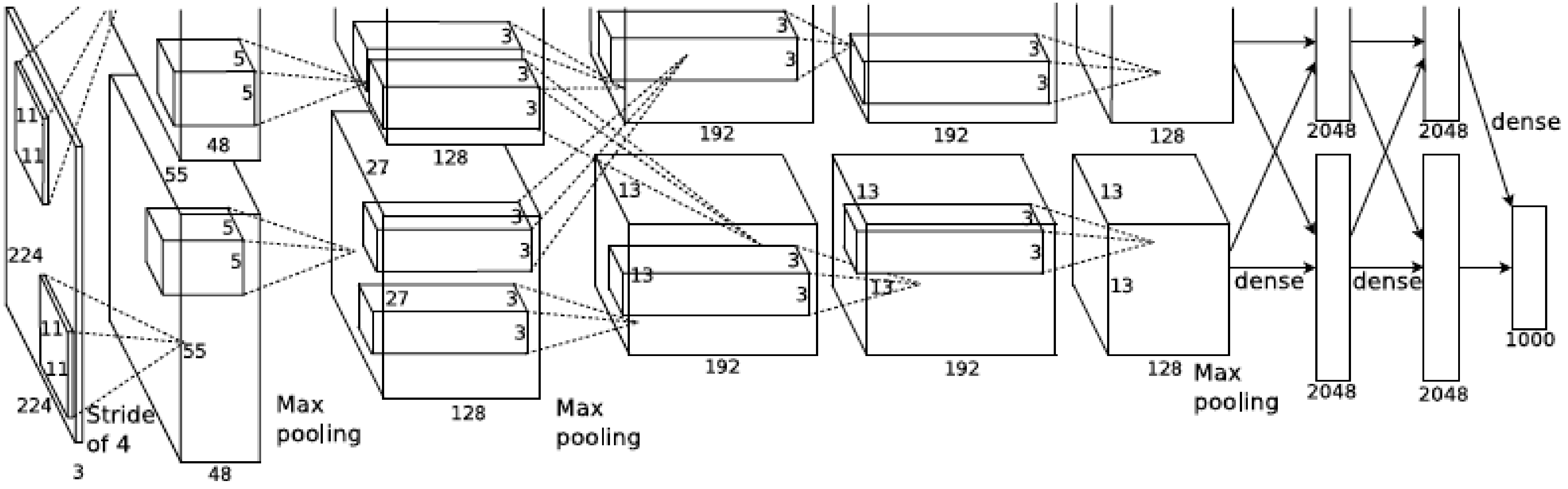
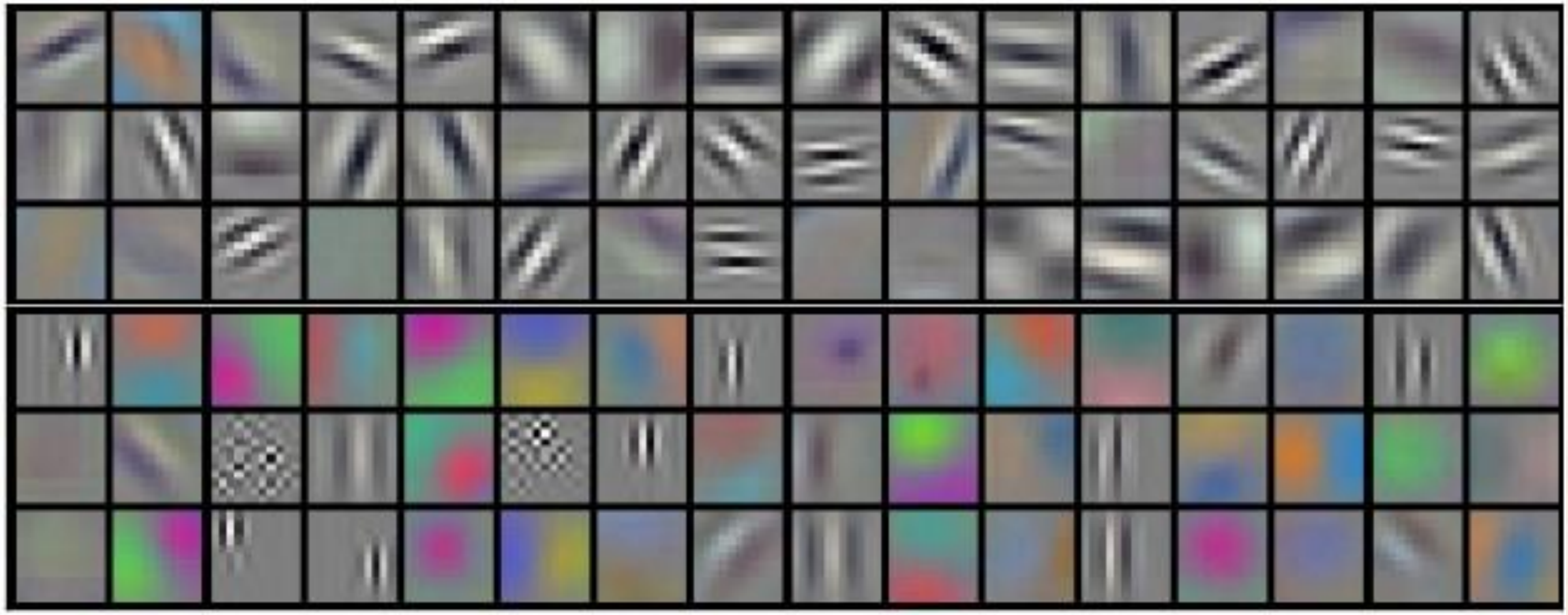


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network’s input is 150,528-dimensional, and the number of neurons in the network’s remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

AlexNet Filters (96)



ZFNet (2013)

[Zeiler & Fergus, ECCV’14]

- Fine tuning of the previous AlexNet structure.
- Shows how to visualize the filters using **DeconvNet**.

Visualizing and Understanding Convolutional Networks

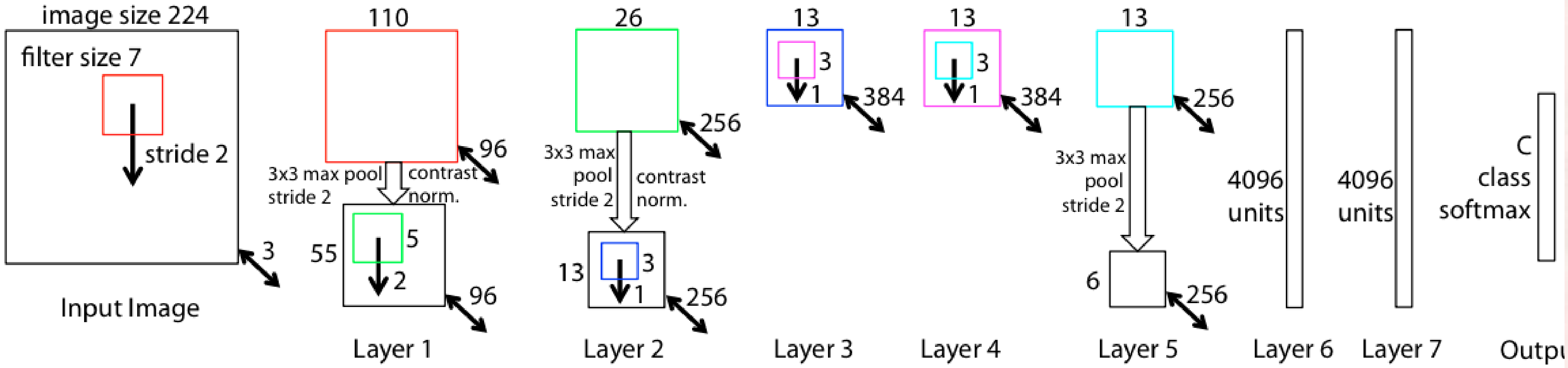
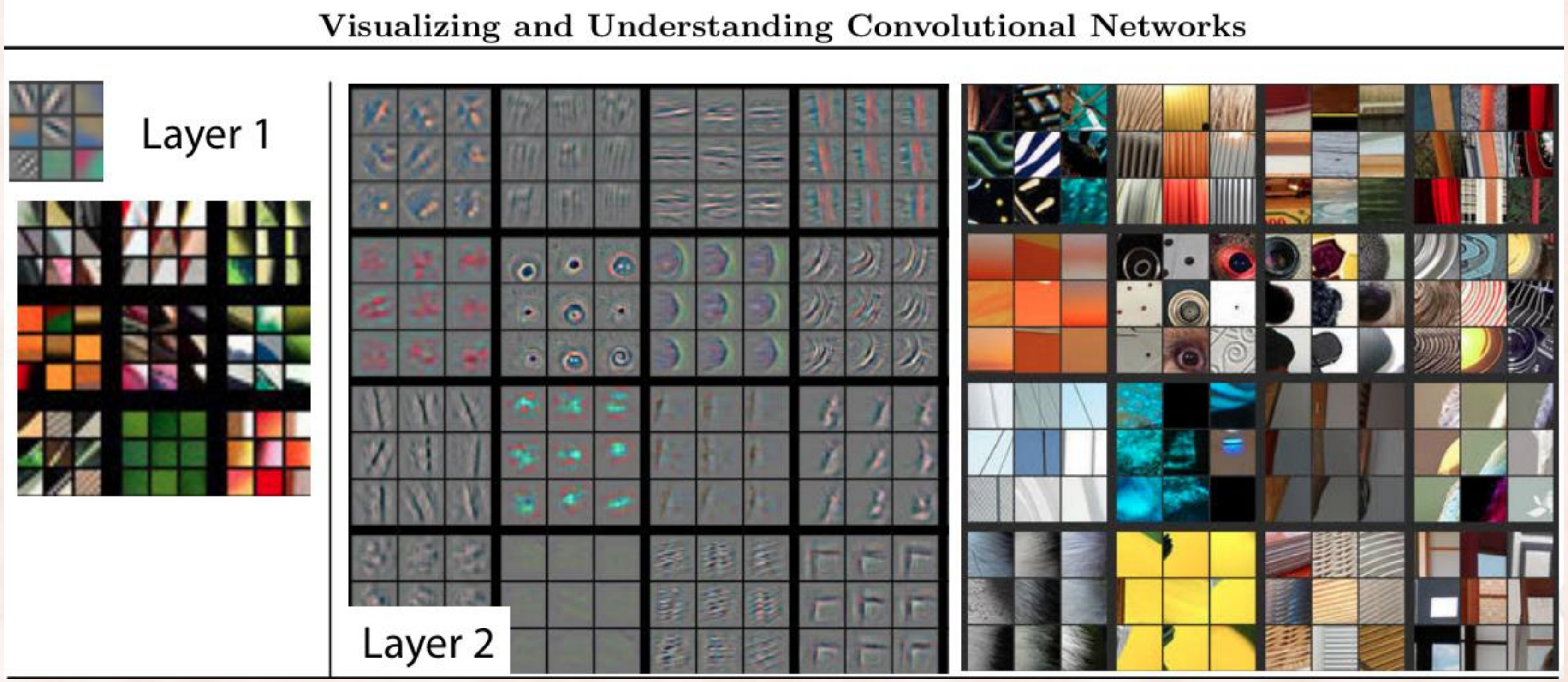


Figure 3. Architecture of our 8 layer convnet model. A 224 by 224 crop of an image (with 3 color planes) is presented as the input. This is convolved with 96 different 1st layer filters (red), each of size 7 by 7, using a stride of 2 in both x and y. The resulting feature maps are then: (i) passed through a rectified linear function (not shown), (ii) pooled (max within 3x3 regions, using stride 2) and (iii) contrast normalized across feature maps to give 96 different 55 by 55 element feature maps. Similar operations are repeated in layers 2,3,4,5. The last two layers are fully connected, taking features from the top convolutional layer as input in vector form ($6 \cdot 6 \cdot 256 = 9216$ dimensions). The final layer is a C-way softmax function, C being the number of classes. All filters and feature maps are square in shape.

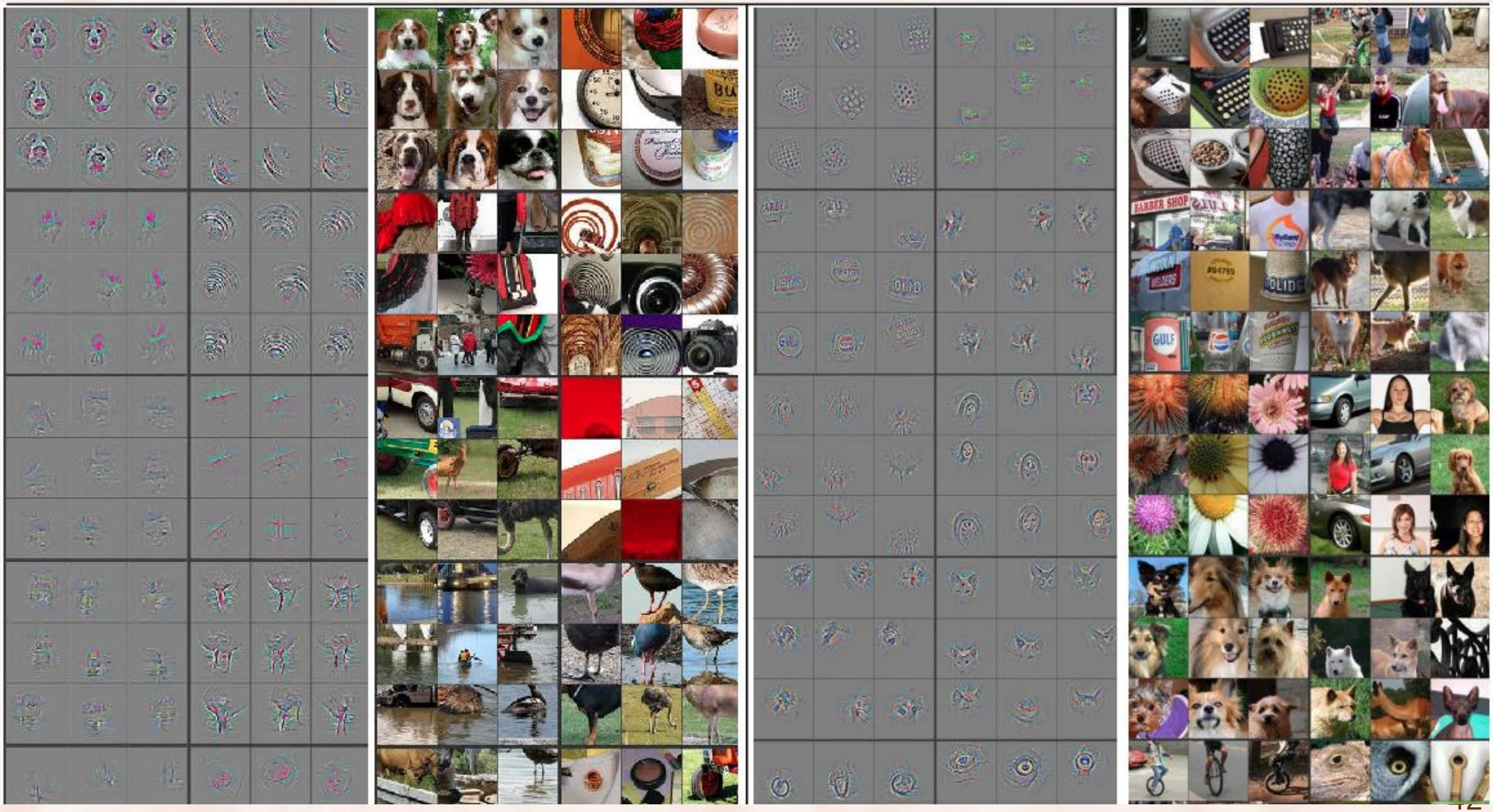
ZFNet (2013)

[Zeiler & Fergus, ECCV’14]



ZFNet (Layer 4)

[Zeiler & Fergus, ECCV’14]



VGGNet and GoogLeNet (2014)

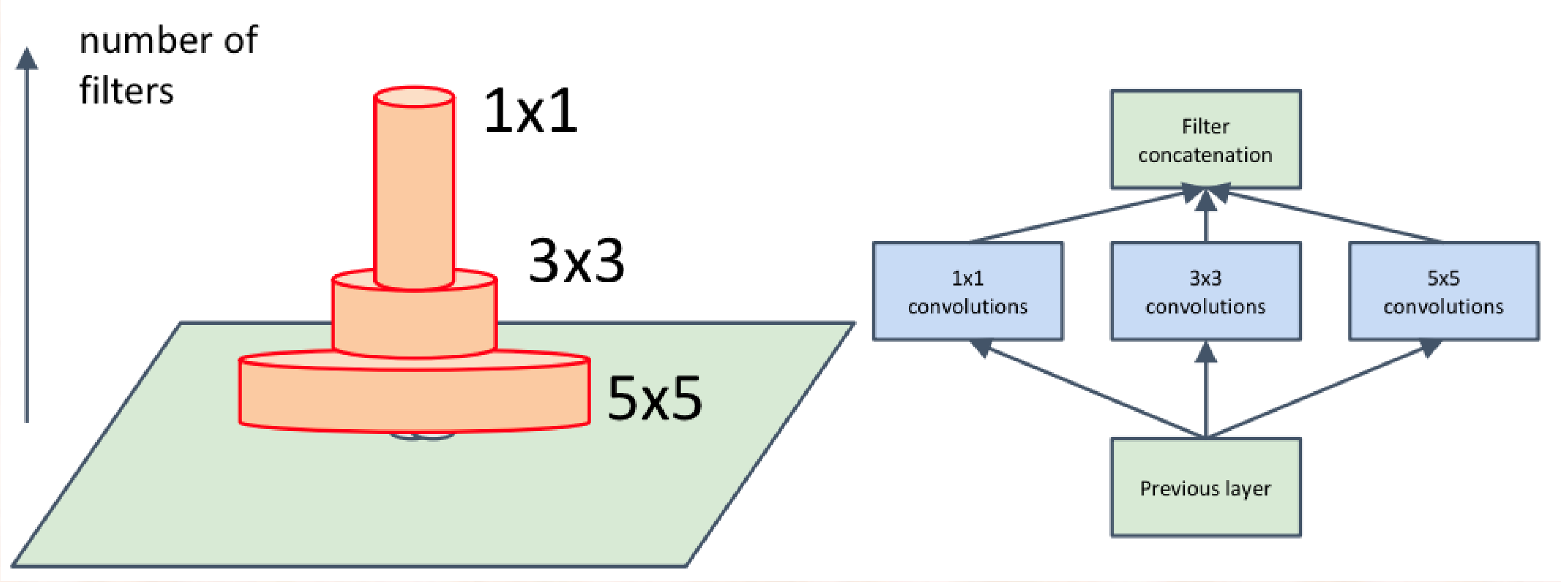
- **VGGNet** (7.3%) was runner-up in ImageNet 2014, showed depth of the network is critical for good performance:
  - Best configuration has 16 CONV/FC layers.
  - Only 3x3 convolutions and 2x2 pooling in all layers.
  - Pretrained model is available in Caffee.

- **GoogLeNet** (6.7%) was the winner in ImageNet 2014:
  - An Inception module dramatically reduced the number of parameters.
  - Used average pooling instead of fully connected layers.
  - 22 layers deep!
  - Inception-v3 reaches 3.46% top 5 error rate.

GoogLeNet

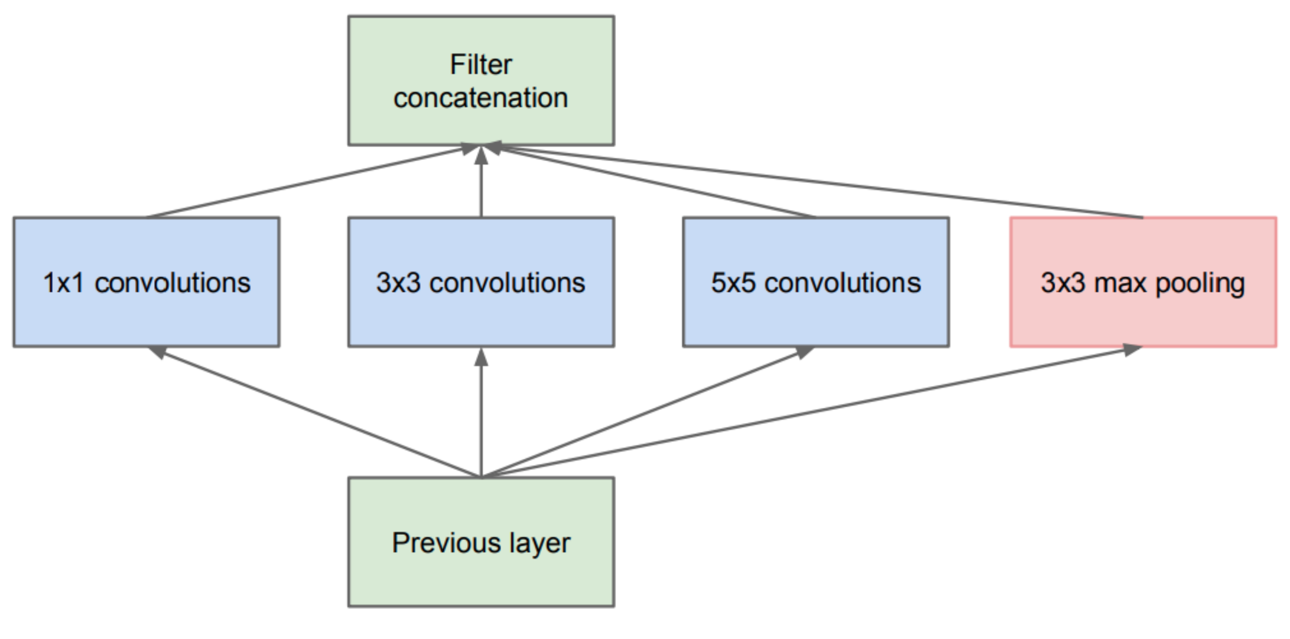
[Szegedy et al., CVPR’15]

- Use a diverse set of convolutions:
  - Filters capture invariances at different scales.



Inception Module

[Szegedy et al., CVPR'15]

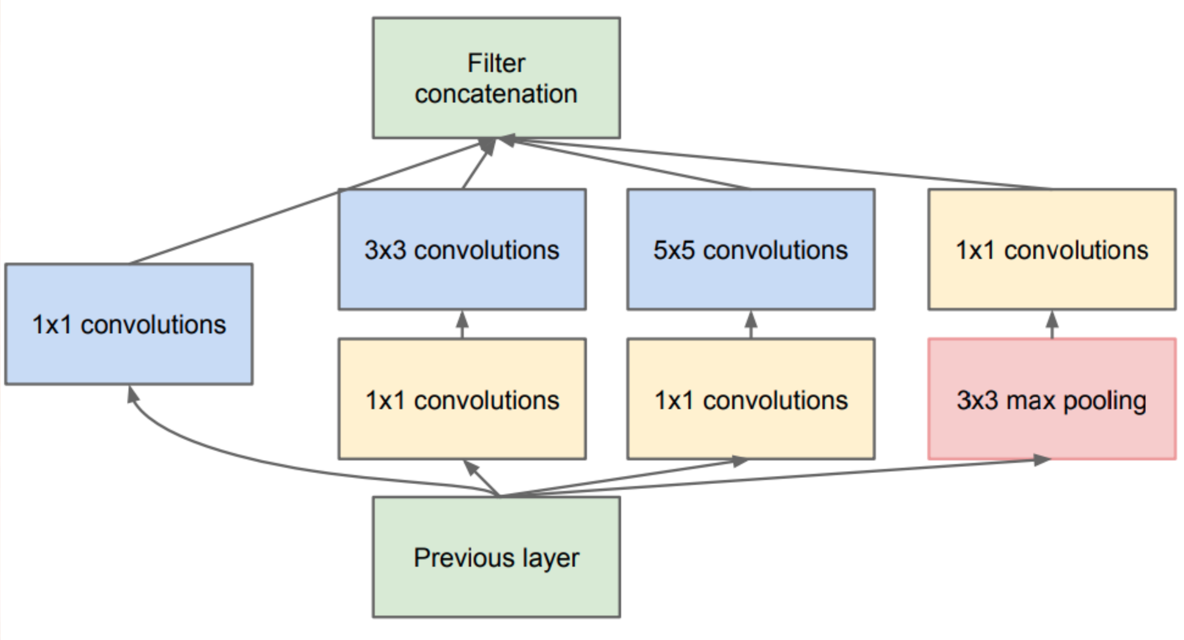


(a) Inception module, naïve version

- Even a modest number of 5×5 convolutions can be prohibitively expensive on top of a convolutional layer with a large number of filters.

Inception Module

[Szegedy et al., CVPR'15]



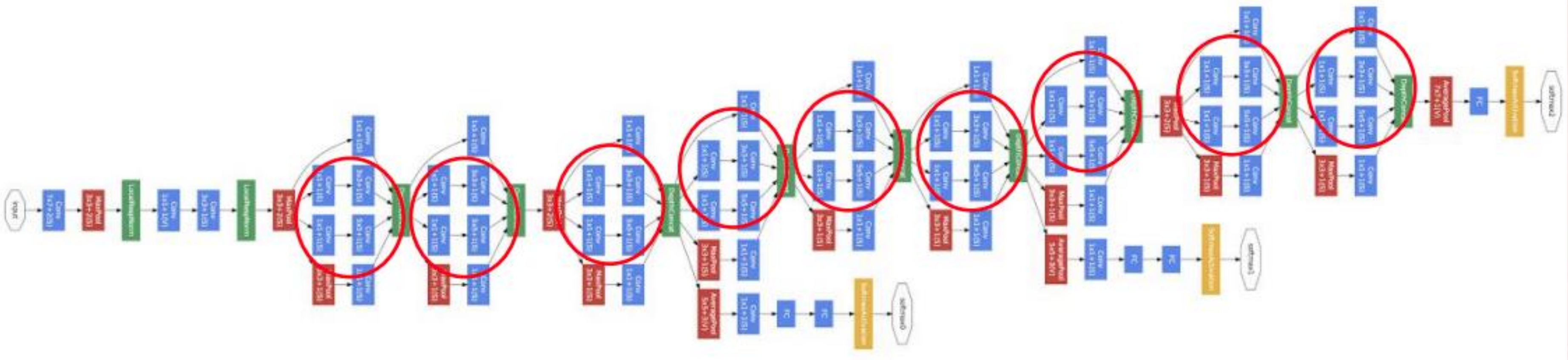
(b) Inception module with dimension reductions

- 1×1 convolutions are used to compute reductions before the expensive 3×3 and 5×5 convolutions.

Inception Network

[Szegedy et al., CVPR'15]

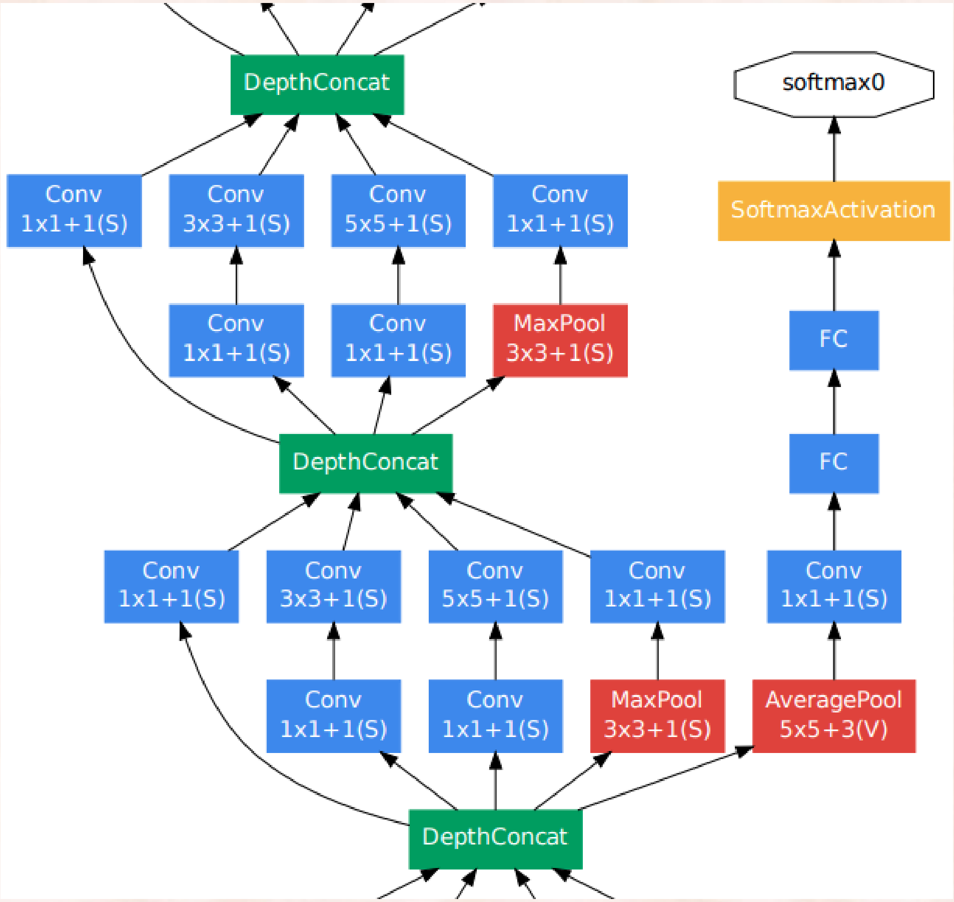
- Inception network is a network consisting of (9) Inception modules stacked upon each other:
  - Occasional max-pooling layers with stride 2 to halve the resolution of the grid.
  - For memory efficiency during training, use Inception modules only at higher layers, keeping the lower layers traditional convolutional.



Inception Network

[Szegedy et al., CVPR'15]

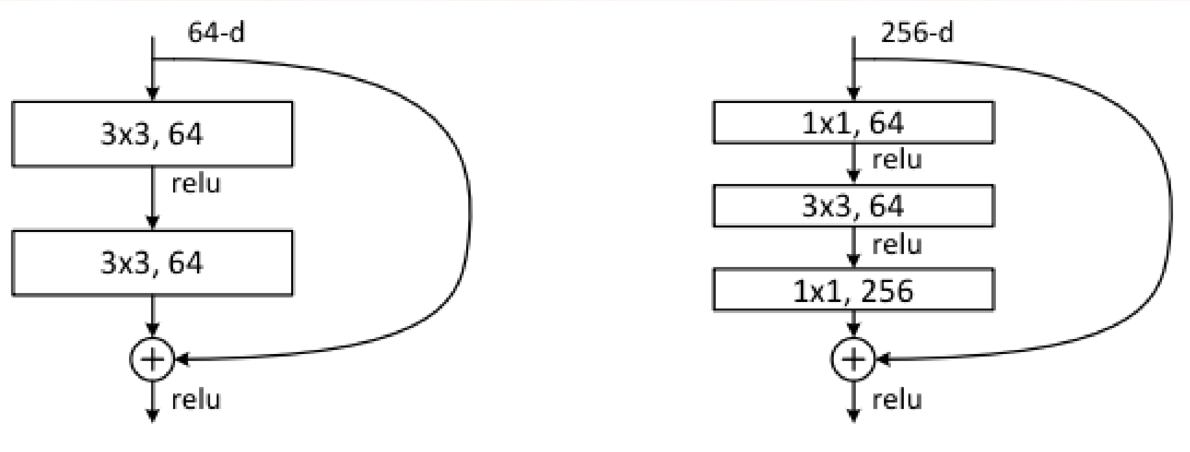
- Softmax outputs in the middle, same labels as at the top:
  - Encourage the network to learn features that are useful for classification.



Microsoft ResNet (2015)

[He at al, CVPR 2016]

- The winner in ImageNet 2015:
  - **Ultra-deep**: from 34 to 152 layers.
  - **Batch normalization** after each convolution, before activation.
  - First layer is (7x7 conv, 64 kernels, S=2), and (3x3 pool, S=2).
  - A **shortcut connection** is added for every block of:
    - Two (3x3,64;relu) layers (34 deep).
    - Three (1x1,64;relu, 3x3,64;relu, 1x1,256;relu) layers (152 deep).



Training/Testing Error Increases with Very Large Depth

[He at al, CVPR 2016]

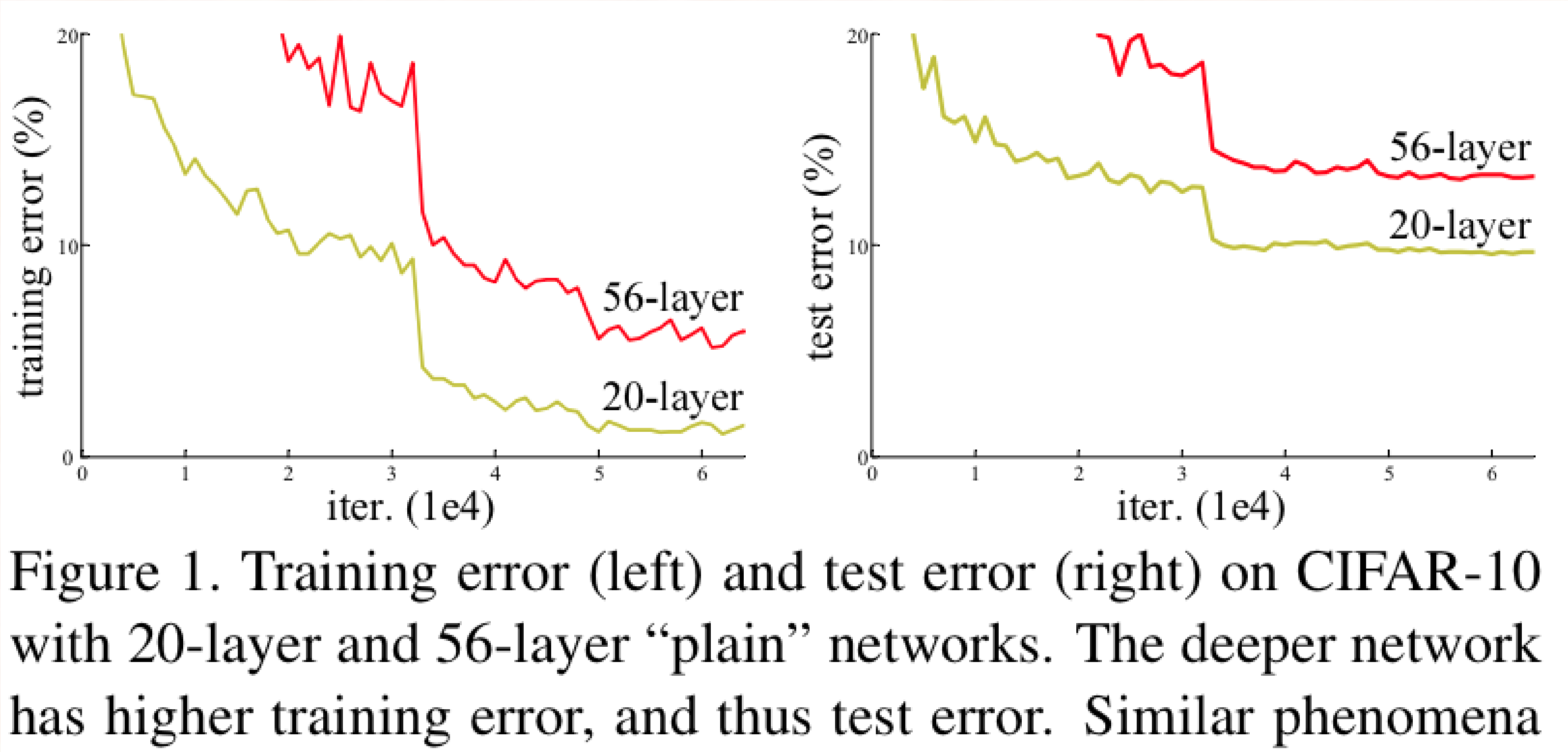


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena

Residual Connections

[He at al, CVPR 2016]

- Make it easier for the network to learn identity mappings.
- **Shortcut connections**:
  - Make identity mappings trivial (all weights 0).
  - Addition operation distributes the gradient.

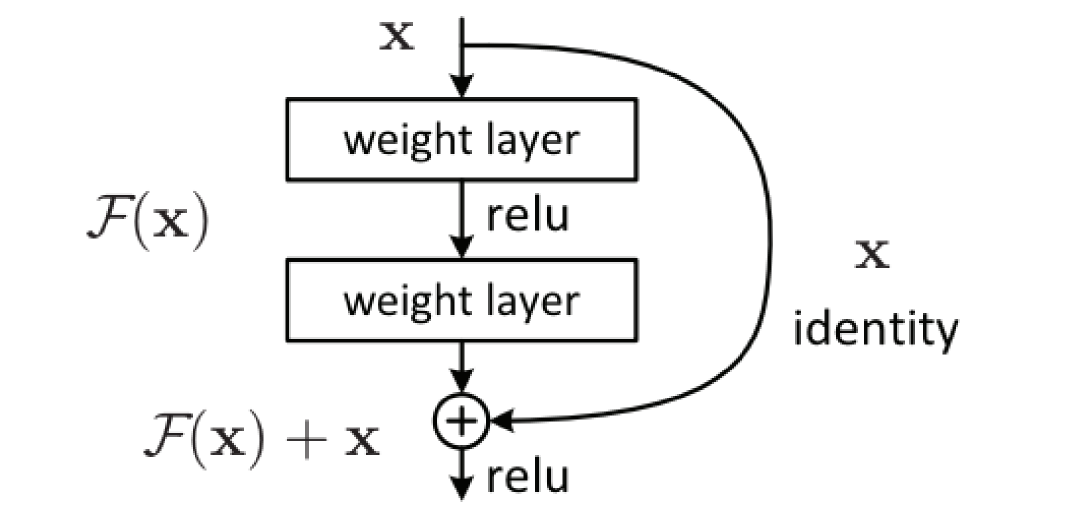
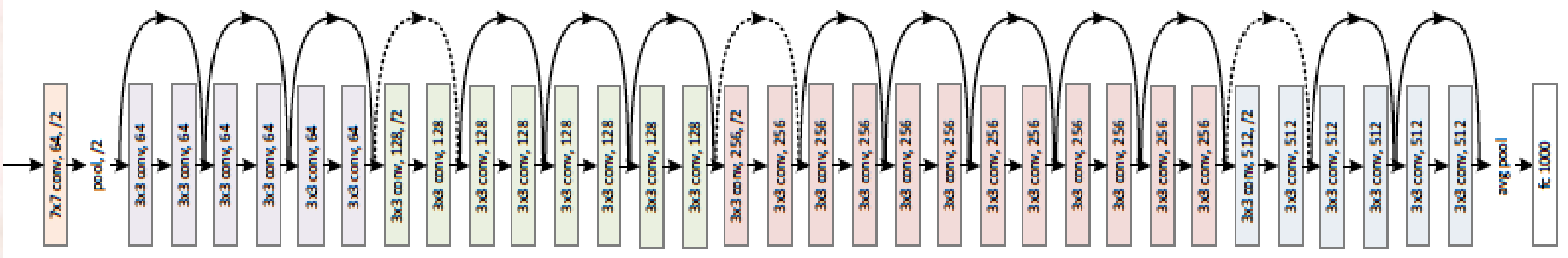


Figure 2. Residual learning: a building block.

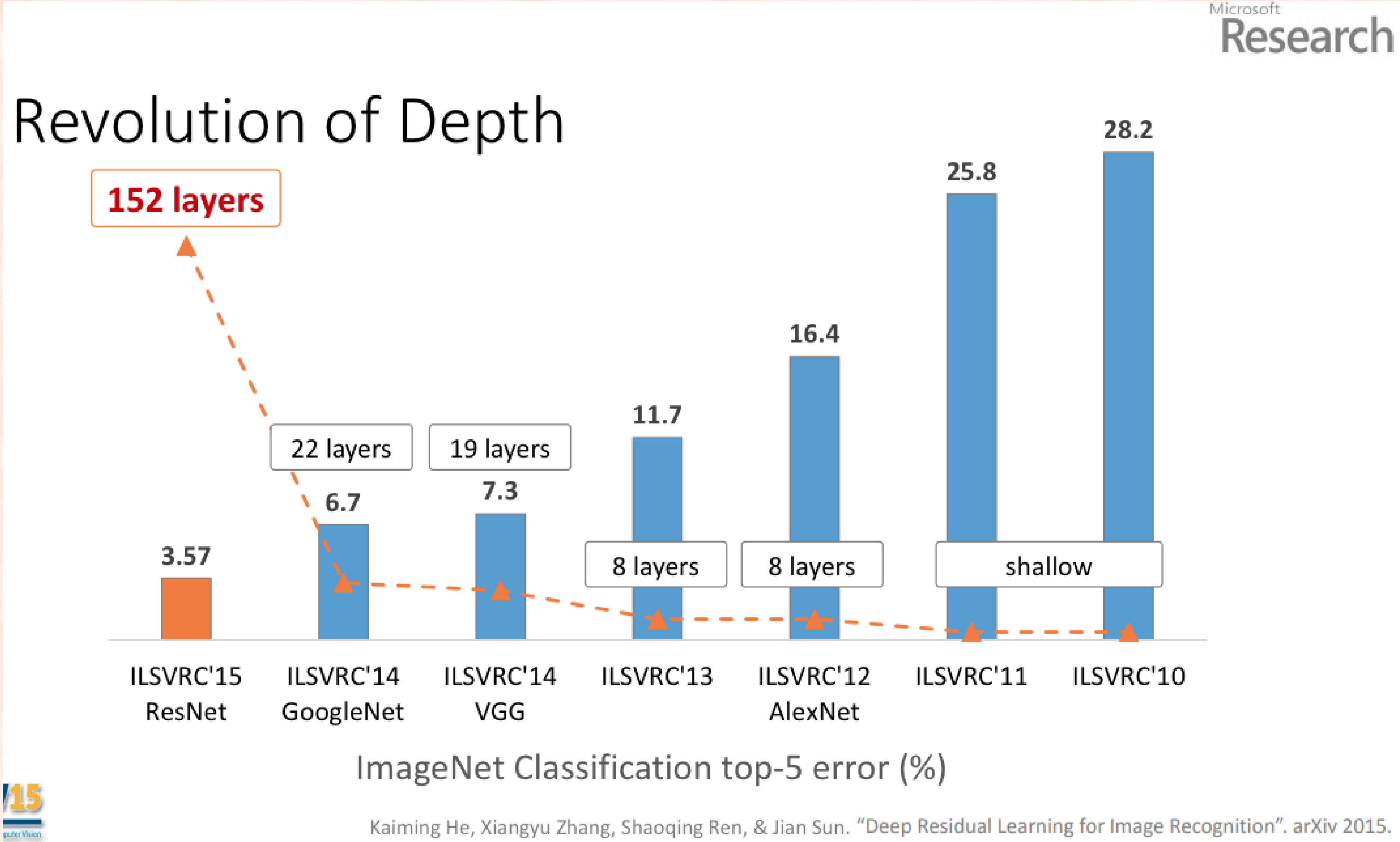
Microsoft ResNet

[He at al, CVPR 2016]



- SGD with minibatch of 256, momentum of 0.9, decay of 0.0001.
- Learning rate = 0.1 divided by 10 when error plateaus, 6x10^5 epochs.
- Depth vs. error rate on validation:
  - 34 depth: top5 = 7.4%, top1 = 24.2%.
  - 152 depth: top5 = **5.7**%, top1 = **21.4**%

http://image-net.org/challenges/talks/ilsvrc2015_deep_residual_learning_kaiminghe.pdf



Microsoft Research

Revolution of Depth

ImageNet Classification top-5 error (%)

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. “Deep Residual Learning for Image Recognition”. arXiv 2015.

Visualization of CNNs

- Activation Maximization [Erhan et al., 2009]:
  - Works well for first layer (see Homework 2).
  - For higher layers => non-convex optimization problem:
    - Use gradient ascent, need careful initialization.
    - Does not give info about unit's invariances (translation, scaling, etc.)
- Data Gradient [Symonian et al., 2014]:
  - Same idea as activation maximization, but applied to ImageNet.
  - Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps
- DeconvNet [Zeiler & Fergus, 2013]:
  - Visualizing and Understanding Convolutional Networks
- Guided Backpropagation [Springenberg et al., 2015]:
  - Striving for Simplicity: The All Convolutional Net

Data Gradient: Deep Inside CNNs

https://arxiv.org/pdf/1312.6034.pdf

- **Class model visualization**:
  - Find the L2-regularized image I such that its score $S_C(I)$ for class C is maximized:

    $$\arg\max_{I} S_C(I) - \lambda \|I\|_2^2$$

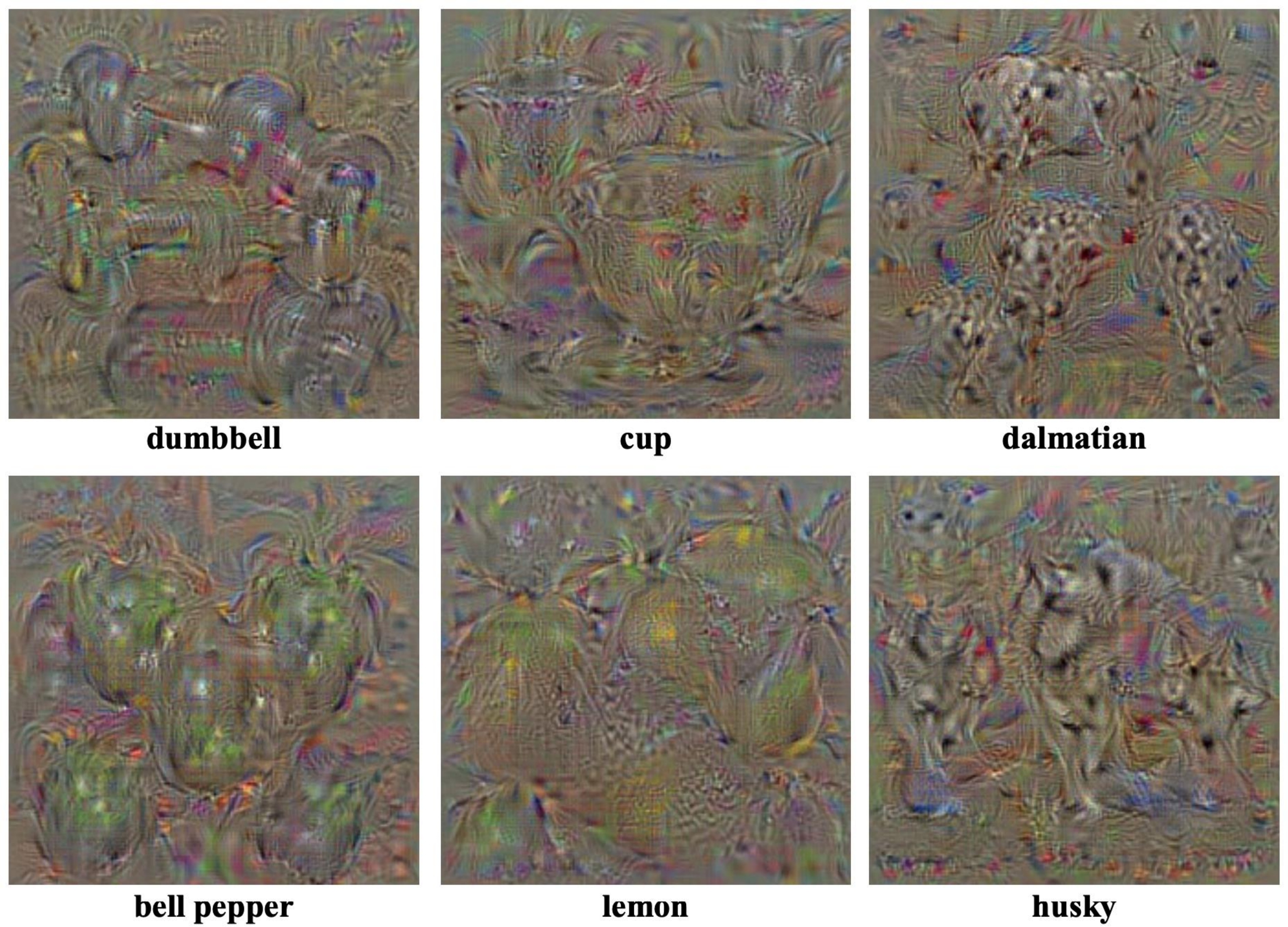
  - Use Backpropagation and the logit scores for S_C:
    - Because maximizing P(C|I) could be achieved by minimizing the logit scores of the other classes.
- **Image-specific class saliency visualization**:
  - Given image I_0 and class C, rank the input pixels based on their influence on the score $S_C(I)$.
    - Linearize $S_C(I)$ around I_0 using 1st order Taylor approximation:

$$S_C(I) \approx \left(\underbrace{\frac{\delta S_C}{\delta I}(I_0)}\right)^T I + b$$

Use Backpropagation, this is the pixel-level ranking score.

Data Gradient: Deep Inside CNNs

https://arxiv.org/pdf/1312.6034.pdf



dumbbell

cup

dalmatian

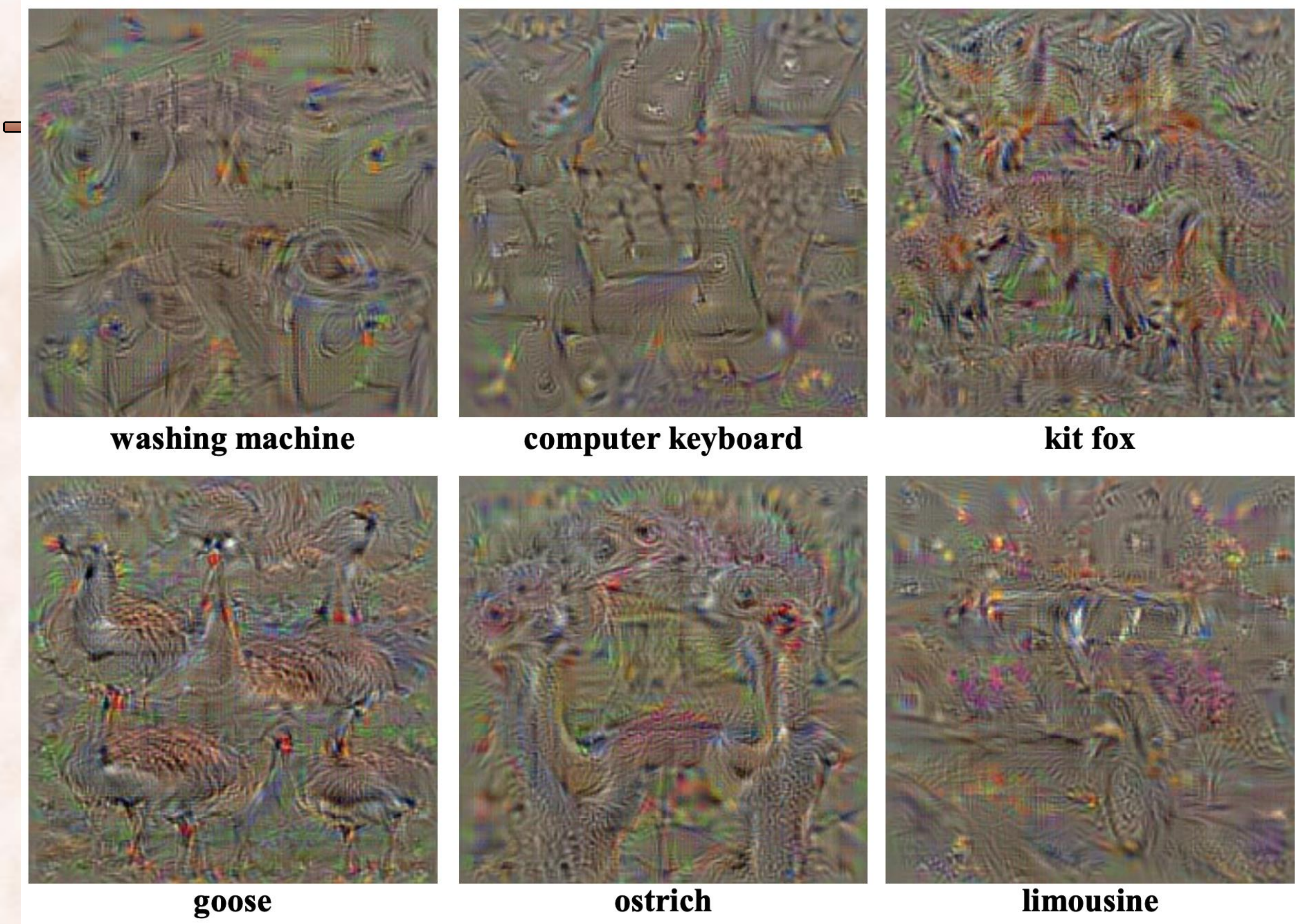
bell pepper

lemon

husky

Data Gradient: Deep Inside CNNs

https://arxiv.org/pdf/1312.6034.pdf

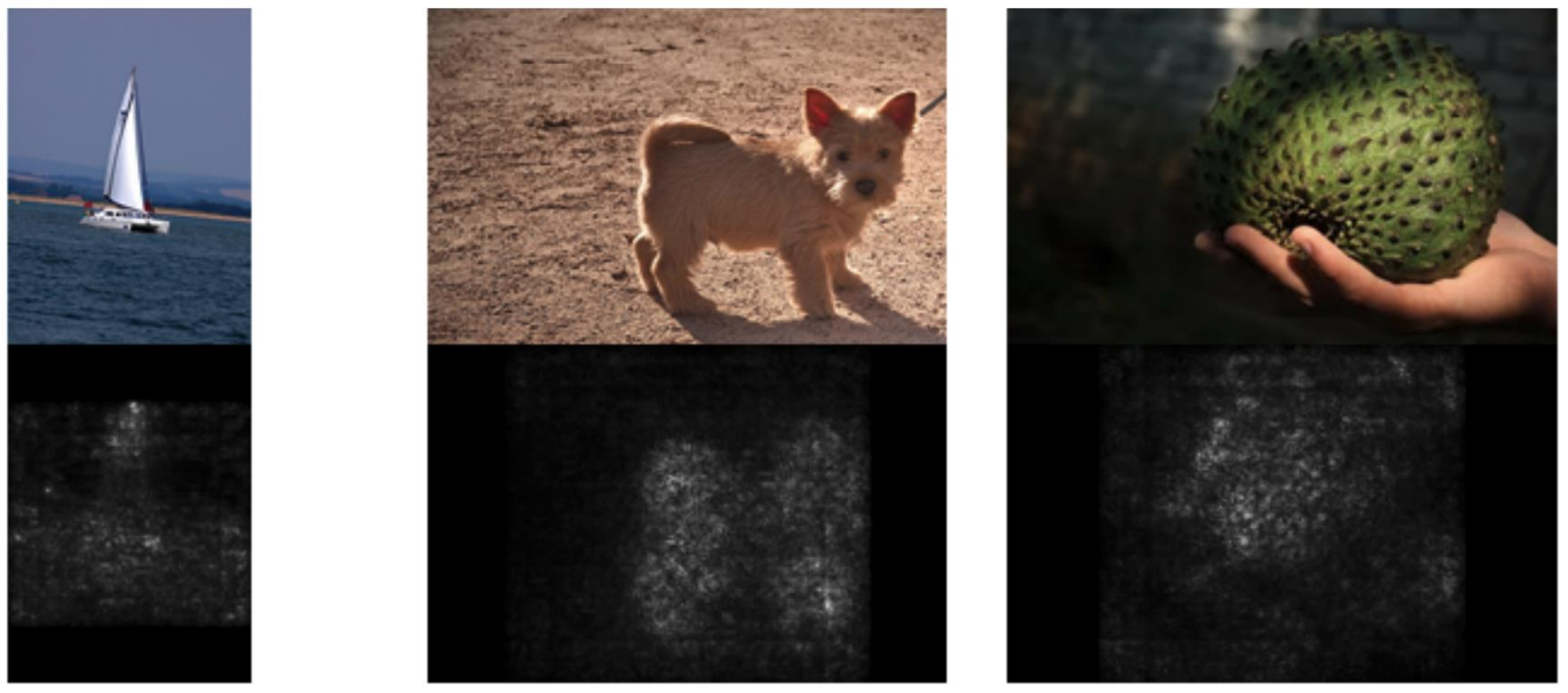


washing machine computer keyboard kit fox

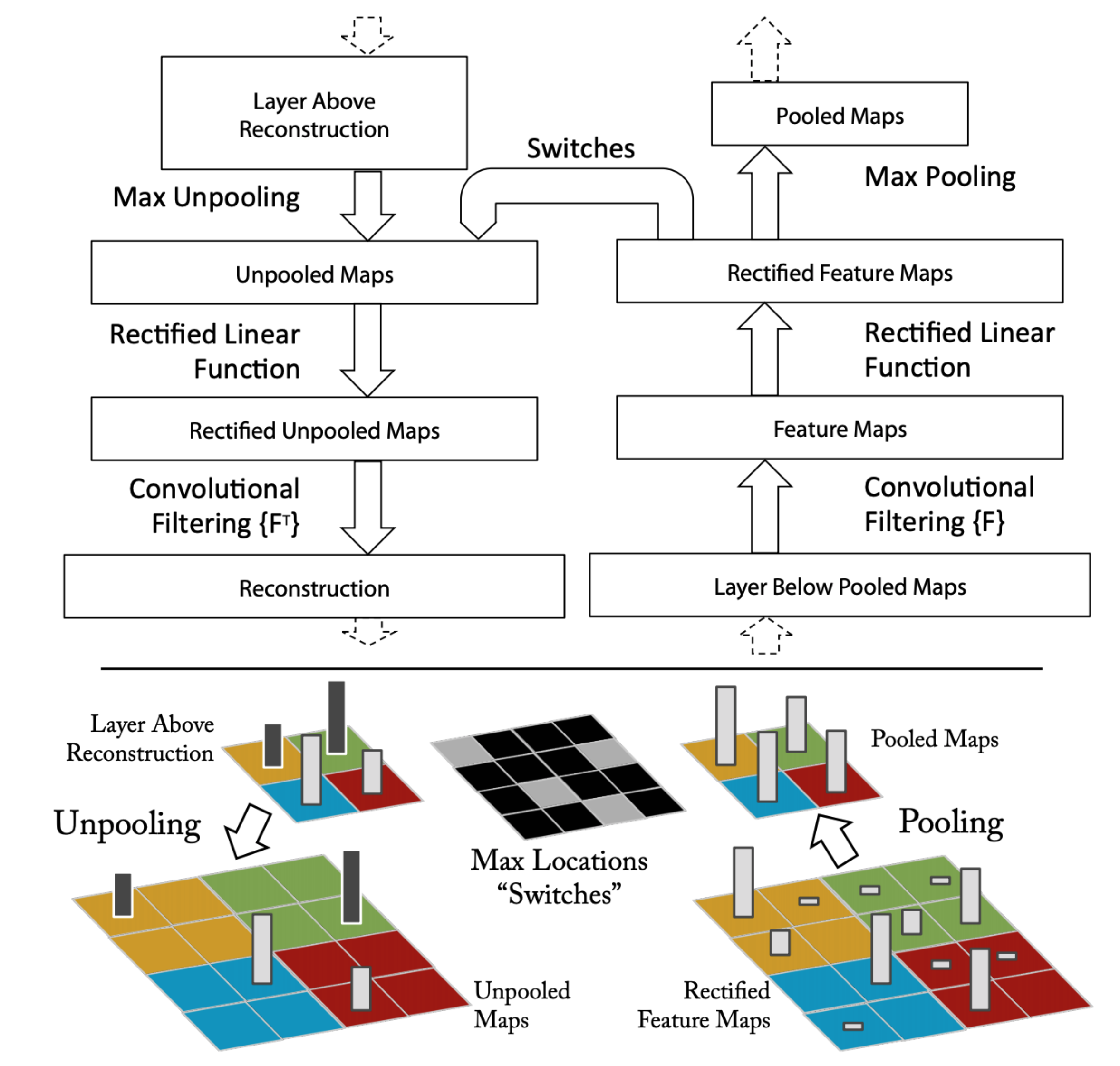
goose ostrich limousine

Data Gradient: Deep Inside CNNs

https://arxiv.org/pdf/1312.6034.pdf



DeconvNet: Visualizing and Understanding CNNs



DeconvNet: Visualizing and Understanding CNNs

- A "deconvnet" layer is attached to each convnet layer.
  - It will reconstruct an approximate version of the convnet features from the previous layer.
    - FF to compute features.
    - Record the locations of each maxima in a pooling region using *switches*.
    - For each activation in a feature map:
      1. **Unpooling**: use switches to place the reconstructions from the layer above into appropriate locations.
      2. **Rectification**: Pass reconstructed signal through *ramp*.
      3. **Filtering**: Apply the "transposed" filter:
         - » actually, the flipped filter (rotate right 90° twice).

DeconvNet: Feature Visualization

https://arxiv.org/pdf/1311.2901.pdf

- Trained and evaluated using AlexNet architecture.
- For layers 2 to 5, show top 9 activations in a random subset of feature maps (*figure 2, page 4*):
  - For each layer, also show the corresponding image patches.

  1. Strong grouping within each reconstructed feature map.
  2. Greater invariance at higher layers:
     - Translation and Scaling, but not Rotation / Viewpoint:
       - Capsule Networks are more robust to affine transforms.
         - » https://arxiv.org/pdf/1710.09829.pdf
  3. Exageration of discriminative parts of images:
     - Image patches have greater variation.
       - Reconstructed features focus on discriminative structure.

DeconvNet: Occlusion Sensitivity

https://arxiv.org/pdf/1311.2901.pdf

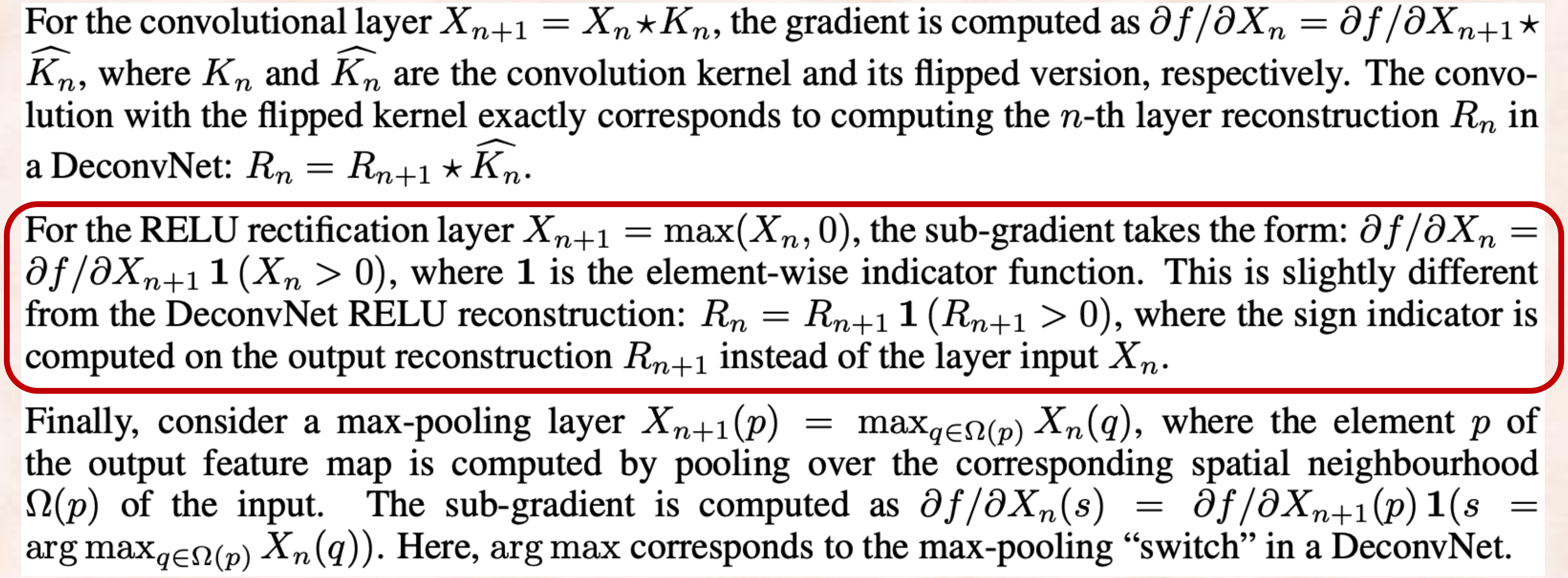
- Is the model truly identifying the location of the object in the image, or just using the surrounding context?

- Occlude different portions of the input image with a grey square, and monitor the output of the classifier (*figure 7, page 7*):
  - Build heat maps to show:
    - How the probability of the correct class changes as the occlusion box moves over the input image.
    - Strongest feature map in layer 5 (strongest over the unoccluded image).
  - Also show projection of strongest feature map.

DeconvNet vs. Data Gradient

https://arxiv.org/pdf/1312.6034.pdf

- Apart from the ReLU layer, computing the approximate feature map reconstruction R_n using a **DeconvNet** is equivalent to computing the **Data Gradient** $\partial f / \partial X_n$ using Backpropagation:



For the convolutional layer $X_{n+1} = X_n \star K_n$, the gradient is computed as $\partial f / \partial X_n = \partial f / \partial X_{n+1} \star \widehat{K_n}$, where K_n and $\widehat{K_n}$ are the convolution kernel and its flipped version, respectively. The convolution with the flipped kernel exactly corresponds to computing the n-th layer reconstruction R_n in a DeconvNet: $R_n = R_{n+1} \star \widehat{K_n}$.

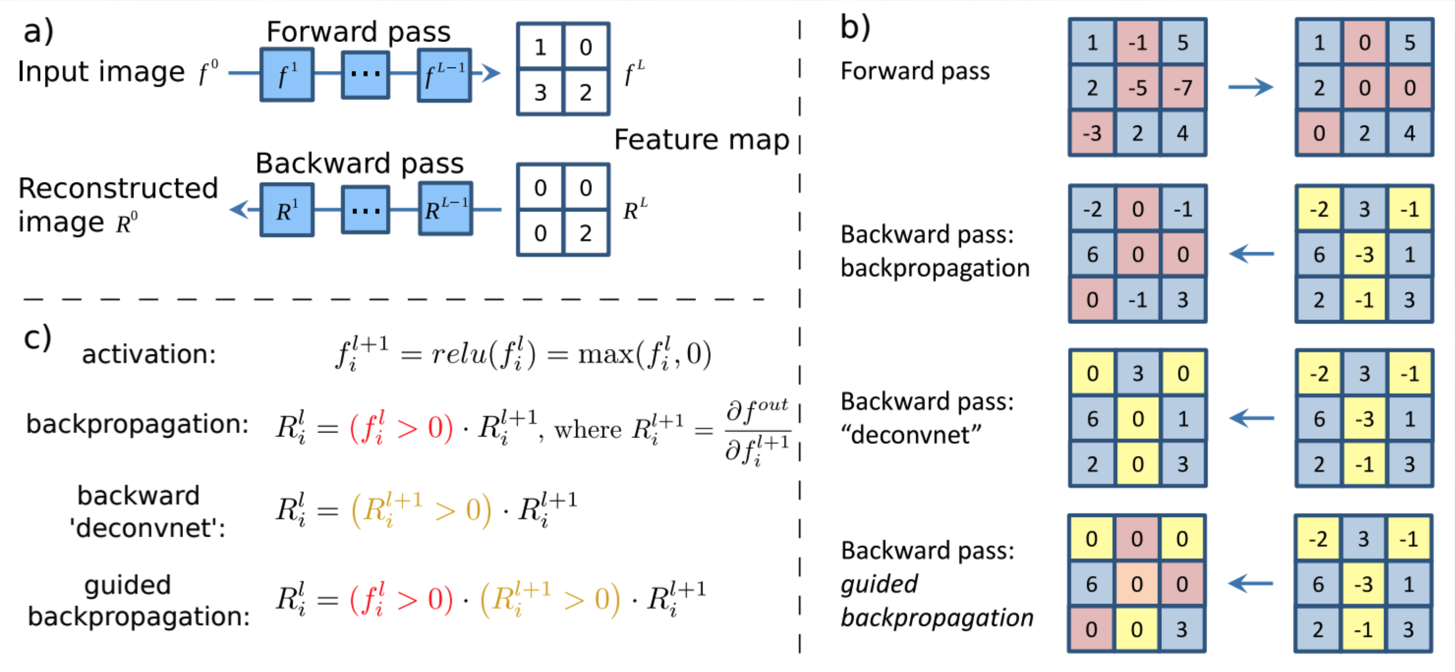
For the RELU rectification layer $X_{n+1} = \max(X_n, 0)$, the sub-gradient takes the form: $\partial f / \partial X_n = \partial f / \partial X_{n+1} \, \mathbf{1}\left(X_n > 0\right)$, where $\mathbf{1}$ is the element-wise indicator function. This is slightly different from the DeconvNet RELU reconstruction: $R_n = R_{n+1} \, \mathbf{1}\left(R_{n+1} > 0\right)$, where the sign indicator is computed on the output reconstruction R_{n+1} instead of the layer input X_n.

Finally, consider a max-pooling layer $X_{n+1}(p) = \max_{q \in \Omega(p)} X_n(q)$, where the element p of the output feature map is computed by pooling over the corresponding spatial neighbourhood $\Omega(p)$ of the input. The sub-gradient is computed as $\partial f / \partial X_n(s) = \partial f / \partial X_{n+1}(p) \, \mathbf{1}(s = \arg\max_{q \in \Omega(p)} X_n(q))$. Here, arg max corresponds to the max-pooling "switch" in a DeconvNet.

The All Convolutional Net

https://arxiv.org/pdf/1412.6806.pdf

- DeconvNet and Backprop differ only in how they treat the ReLU.
- Combine the two methods:
  - Rather than masking out values corresponding to negative entries of the top gradient ('deconvnet') or bottom data (backprop), mask out the values for which at least one of these values is negative.



1	0
3	2

0	0
0	2

Forward pass

1	-1	5	→	1	0	5
2	-5	-7		2	0	0
-3	2	4		0	2	4

Backward pass: backpropagation

-2	0	-1	←	-2	3	-1
6	0	0		6	-3	1
0	-1	3		2	-1	3

Backward pass: "deconvnet"

0	3	0	←	-2	3	-1
6	0	1		6	-3	1
2	0	3		2	-1	3

Backward pass: *guided backpropagation*

0	0	0	←	-2	3	-1
6	0	0		6	-3	1
0	0	3		2	-1	3

activation: $f_i^{l+1} = relu(f_i^l) = \max(f_i^l, 0)$

backpropagation: $R_i^l = (f_i^l > 0) \cdot R_i^{l+1}$, where $R_i^{l+1} = \frac{\partial f^{out}}{\partial f_i^{l+1}}$

backward 'deconvnet': $R_i^l = (R_i^{l+1} > 0) \cdot R_i^{l+1}$

guided backpropagation: $R_i^l = (f_i^l > 0) \cdot (R_i^{l+1} > 0) \cdot R_i^{l+1}$

The All Convolutional Net

https://arxiv.org/pdf/1412.6806.pdf

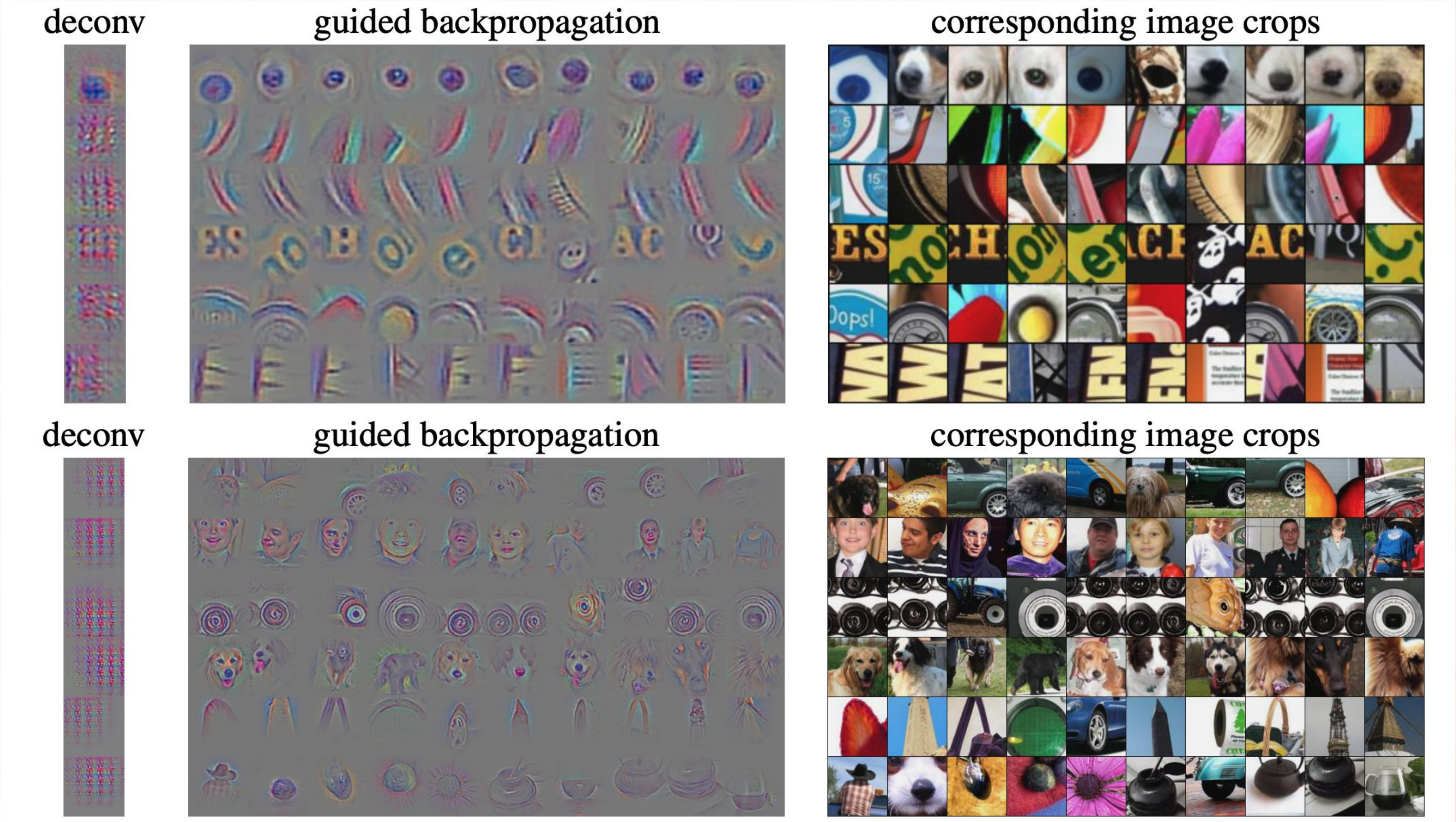


Figure 3: Visualization of patterns learned by the layer conv6 (top) and layer conv9 (bottom) of the network trained on ImageNet. Each row corresponds to one filter. The visualization using “guided backpropagation” is based on the top 10 image patches activating this filter taken from the ImageNet

The All Convolutional Net

https://arxiv.org/pdf/1412.6806.pdf

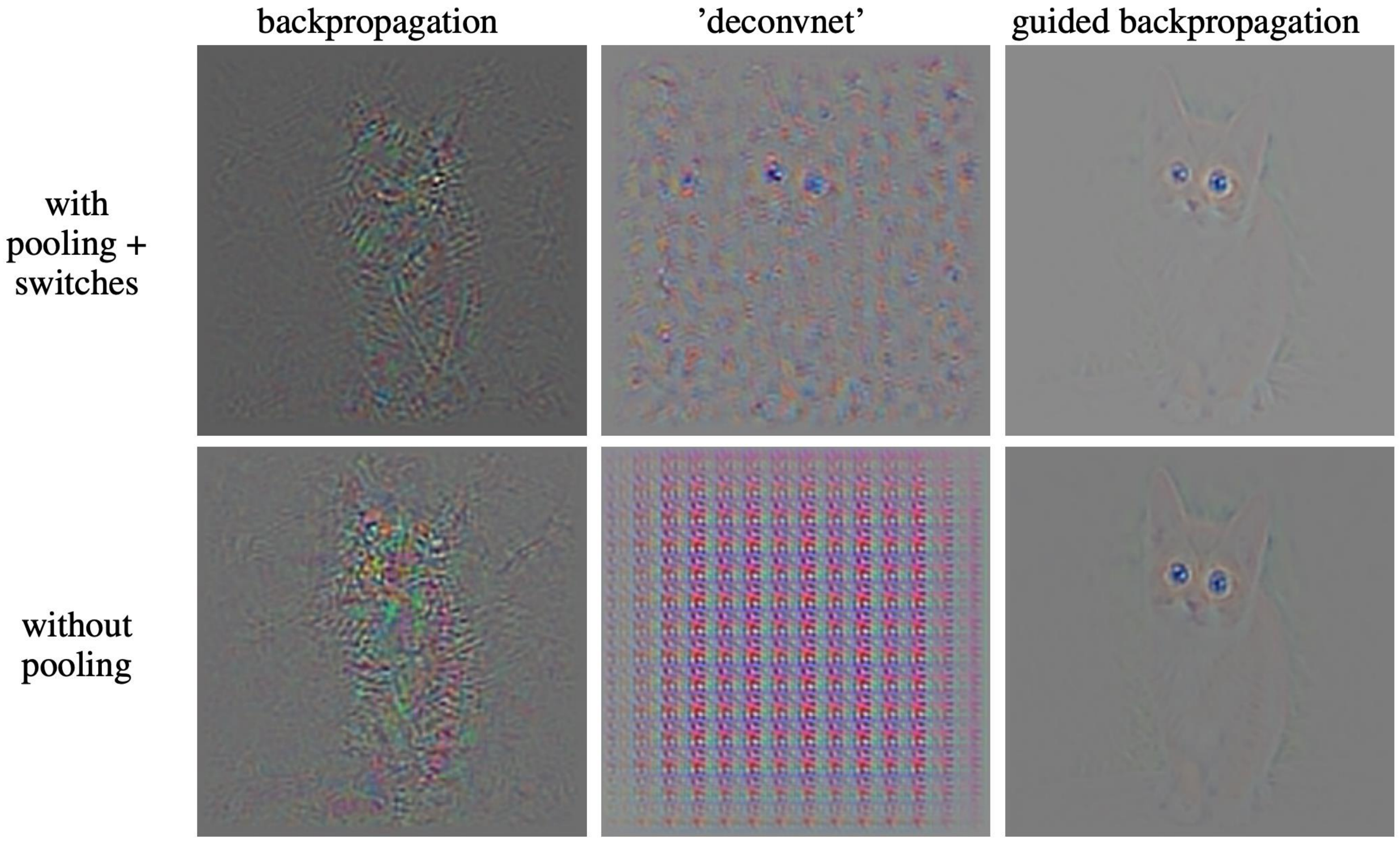


Figure 5: Visualization of descriptive image regions with different methods from the single largest activation in the pre-softmax layer global_pool of the network trained on ImageNet.

Local Response Normalization

[Krizhevsky et al., NIPS’12]

- **Lateral inhibition** (neurobiology):
  - – Capacity of an excited neuron to subdue its neighbors.
    - Create contrast (significant peak), increase sensory perception.
- Let $a^i_{x,y}$ be the activity of a neuron:
  - – Computed by applying kernel i at position (*x*, *y*) and then ReLU.
- Compute the response-normalized activity:
  - – Sum runs over *n* adjacent kernel maps at the same spatial position:

$$b^i_{x,y} = a^i_{x,y} \Big/ \left(k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} \left(a^j_{x,y}\right)^2 \right)^{\beta}$$