IS 6733: Deep Learning on Cloud Platforms

Convolutional Neural Network

Copy Right Notice

- Most slides in this presentation are adopted from slides of text book and various sources. The Copyright belong to the original authors. Thanks!

Why CNN for Image

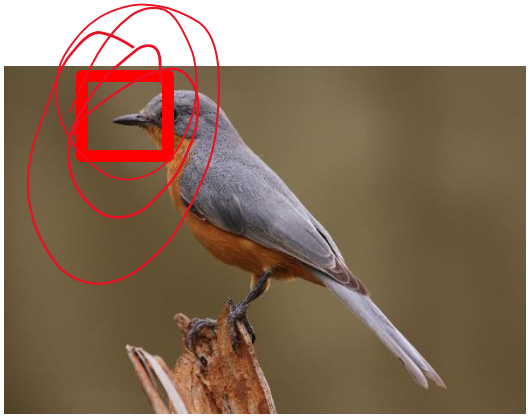
- **Some patterns are much smaller than the whole image**

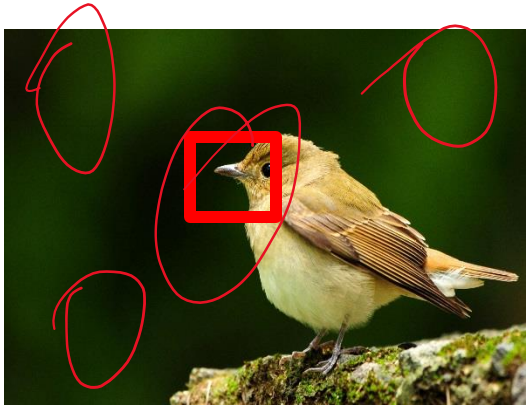
A neuron does not have to see the whole image to discover the pattern.

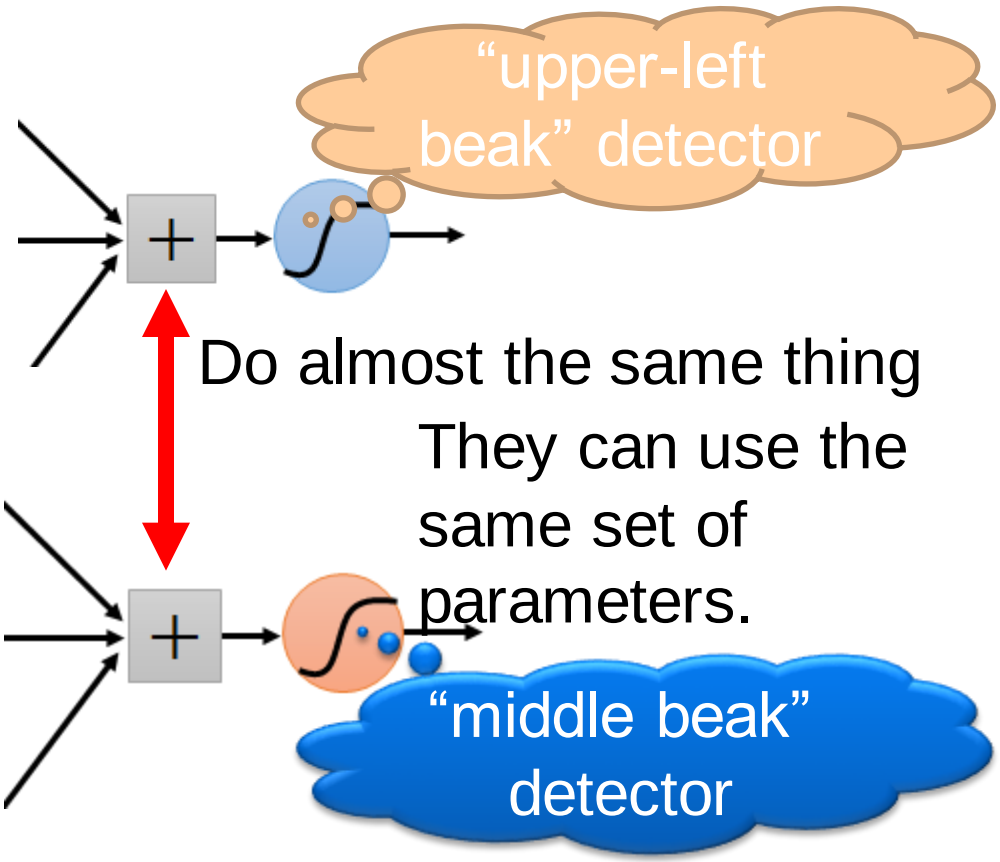
Connecting to small region with less parameters

Why CNN for Image

- **The same patterns appear in different regions.**



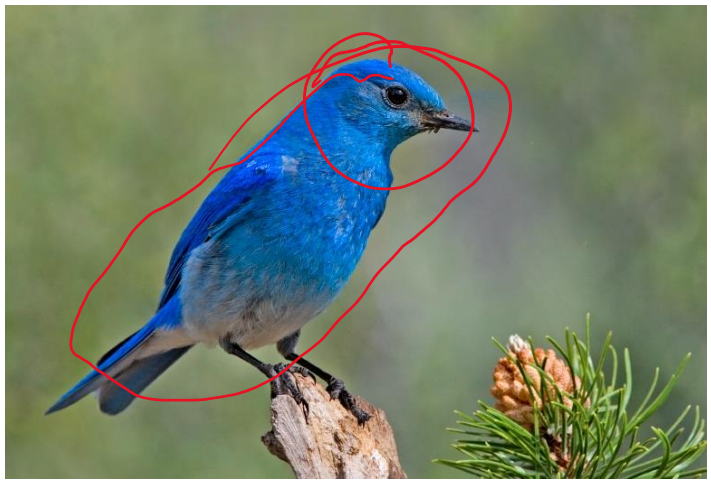




Why CNN for Image

- **Subsampling the pixels will not change the object**

bird



subsampling

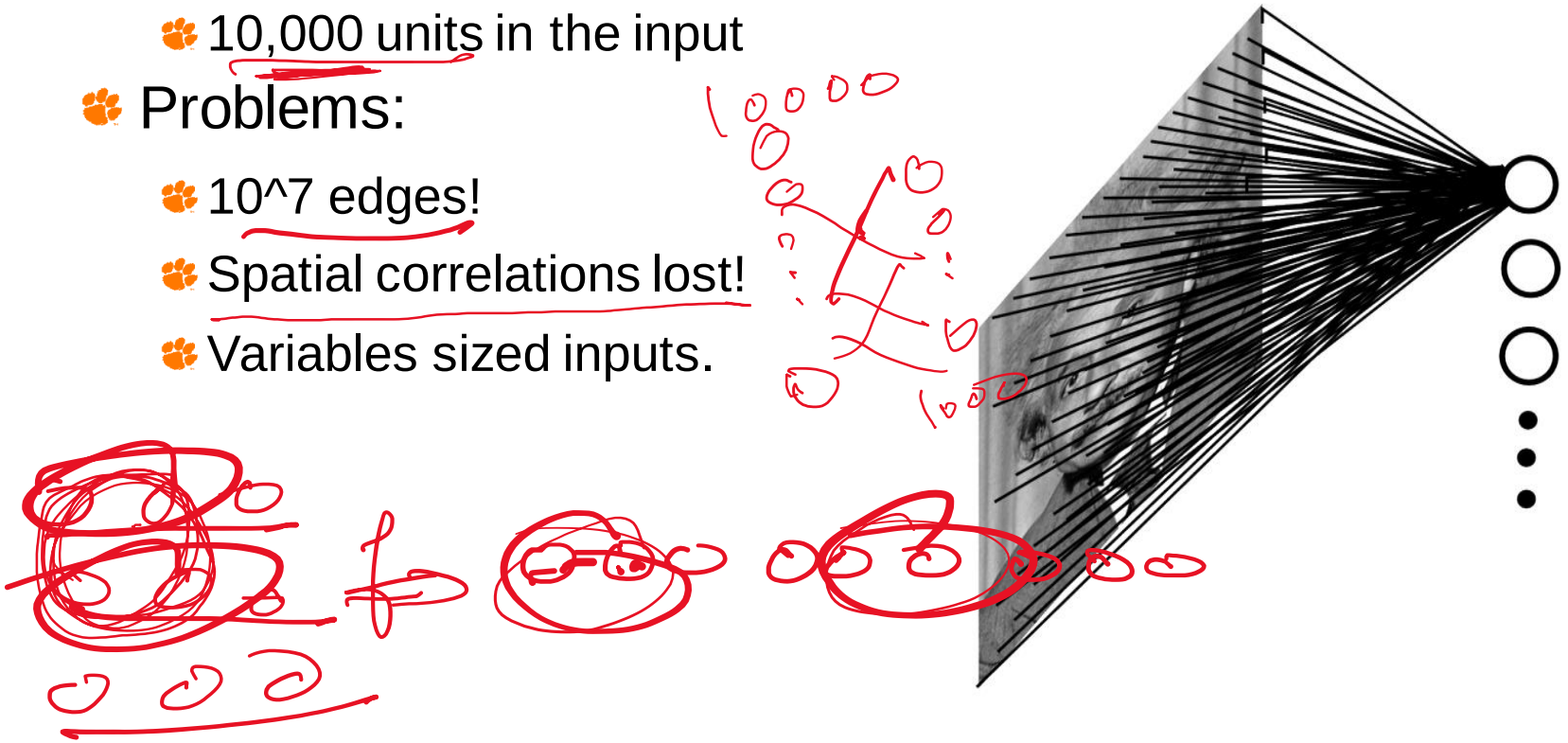
bird

We can subsample the pixels to make image smaller

Less parameters for the network to process the image

Multiple Layer Neural Network

- **A fully connected layer:**
  - Example:
    - 100x100 images
    - 10,000 units in the input
  - Problems:
    - 10^7 edges!
    - Spatial correlations lost!
    - Variables sized inputs.



Convolutional Layer

- **A solution:**
  - **Filters** to capture different patterns in the input space.
    - **Share** parameters across different locations (assuming input is stationary)
    - **Convolutions** with learned filters
  - Filters will be **learned** during training.
  - The issue of variable-sized inputs will be resolved with a **pooling** layer.

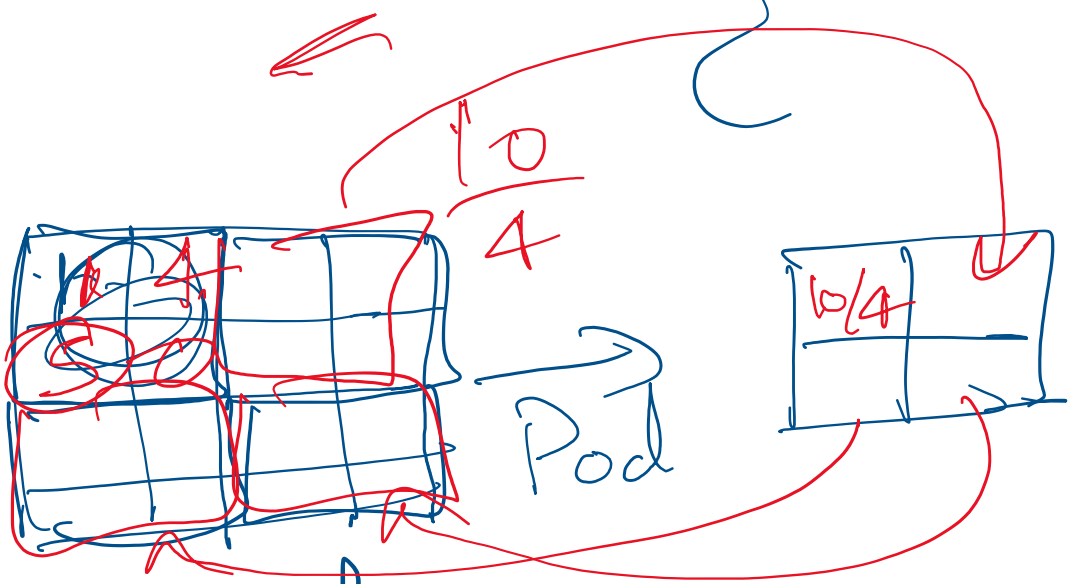
CONVOLUTIONAL NEURAL NETWORK

1. A convolutional neural network (CNN, or ConvNet) is a type of feed-forward artificial neural network.
2. Connectivity pattern between its neurons - inspired by the organization of the animal visual cortex
3. CNN was introduced by Yann LeCun (early 1990s).

LeNet 5

Architecture of CNN

- Convolutional layers
- Sub-sampling layers
- Fully-connected layers



The whole CNN

Convolution

Max Pooling

Convolution

Max Pooling

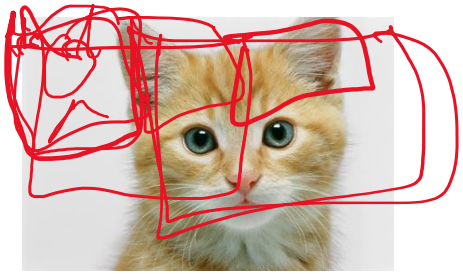
Can repeat many times

Flatten

Fully Connected Feedforward network

cat dog ……

The whole CNN



Property 1

- Some patterns are much smaller than the whole image

Property 2

- The same patterns appear in different regions.

Property 3

- Subsampling the pixels will not change the object

Convolution

Max Pooling

Convolution

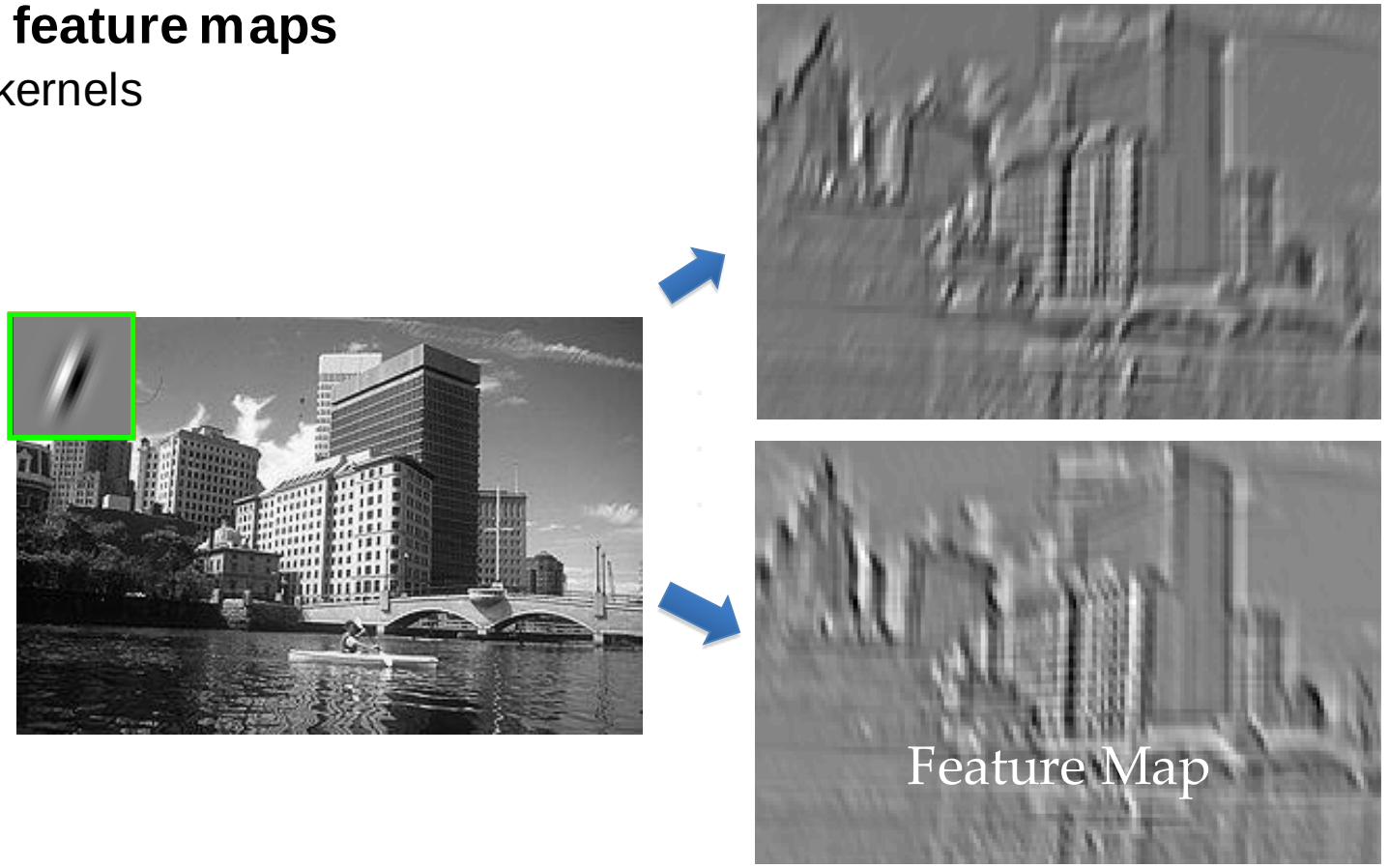
Max Pooling

Can repeat many times

Flatten

Convolution

- **Convolution is a common image processing technique that changes the intensities of a pixel to reflect the intensities of the surrounding pixels. A common use of convolution is to create image filters**
- **Generating feature maps**
  - Different kernels



Convolution Operator

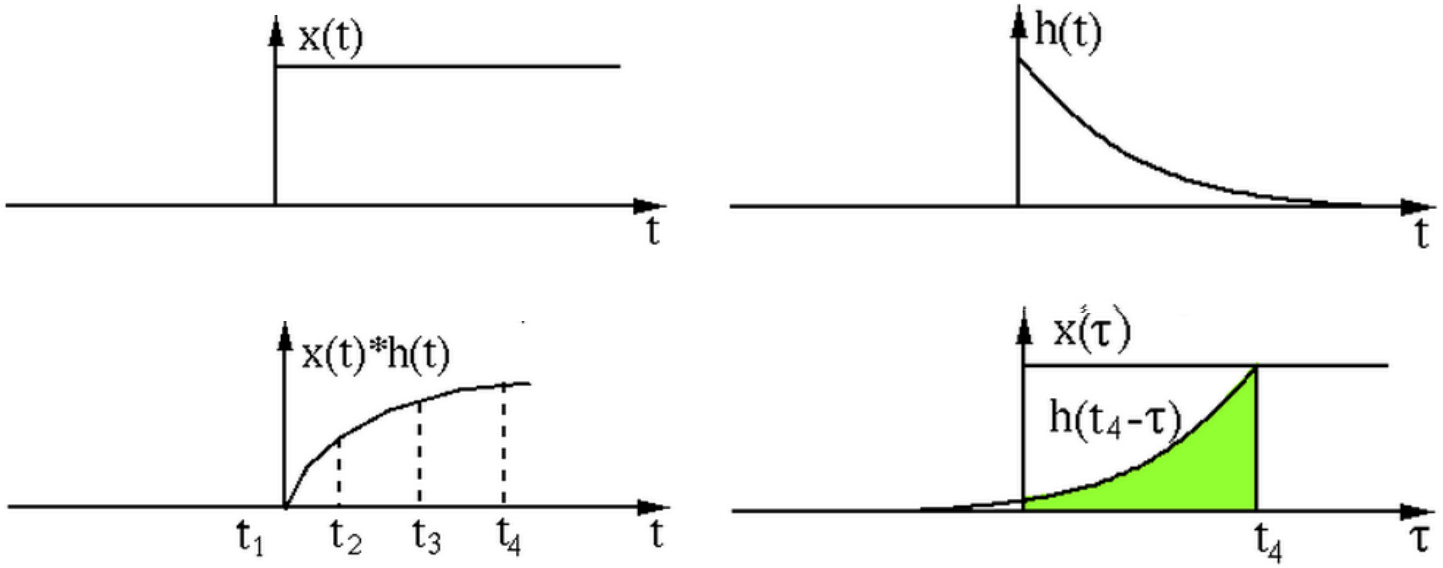
"Convolution" is very similar to "cross-correlation", except that in convolution one of the functions is flipped.

- **Convolution Operator: Take two functions and given another function**

$$(x * h)(t) = \int x(\tau)h(t-\tau)d\tau$$

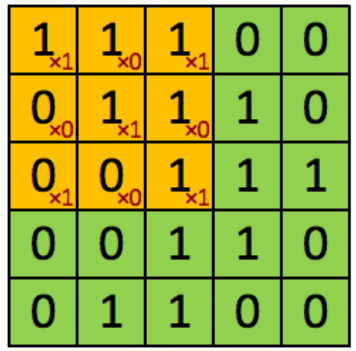
$$(x * h)[n] = \sum_m x[m]h[n-m]$$

- **One dimension:**

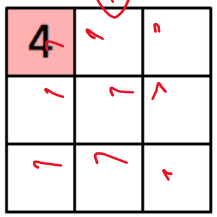


Convolution Operator

- **Convolution in two dimension:**
  - The same idea: flip one matrix and slide it on the other matrix



Image



Convolved Feature

CNN – Convolution

Those are the network parameters to be learned.

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

Matrix

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

Matrix

⋮ ⋮

Property 1

Each filter detects a small pattern (3 x 3).

CNN – Convolution

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

3 -1

CNN – Convolution

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

If stride=2

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

3 -3

We set stride=1 below

CNN – Convolution

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

Property 2

CNN – Conv

-1	1	-1
-1	1	-1
-1	1	-1

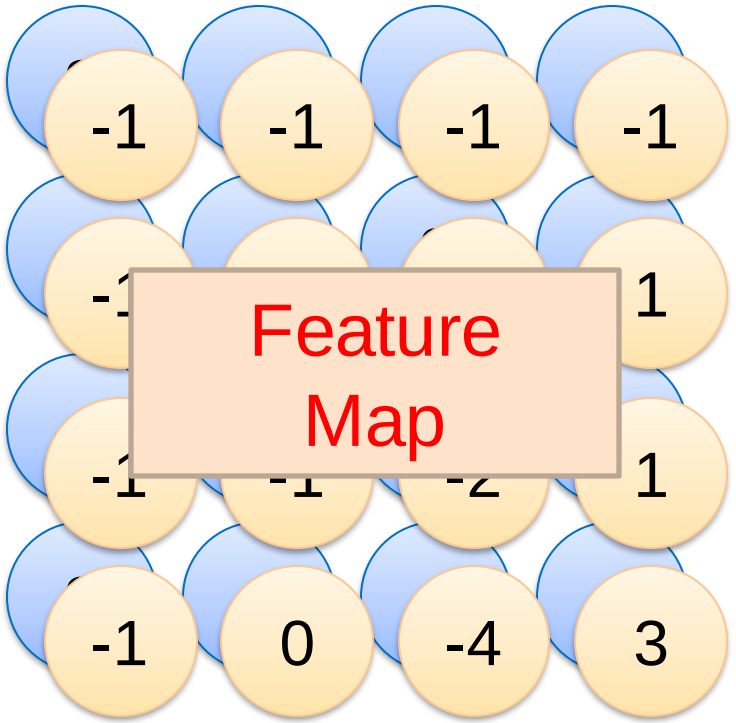
Filter 2

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

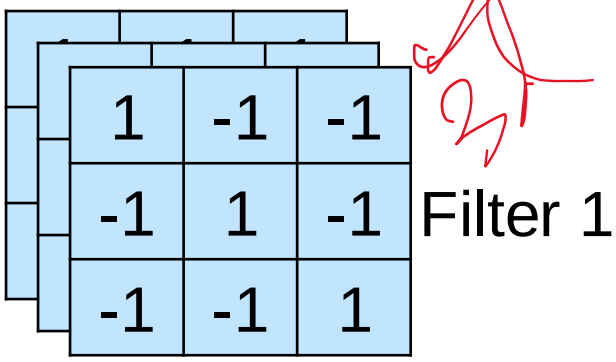
6 x 6 image

Do the same process for every filter

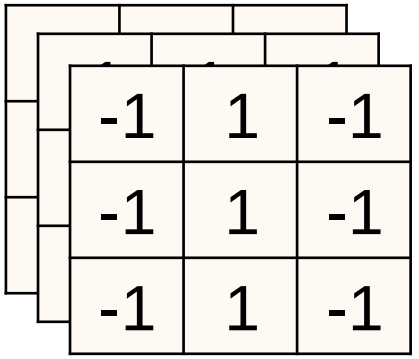


4 x 4 image

CNN – Colorful image

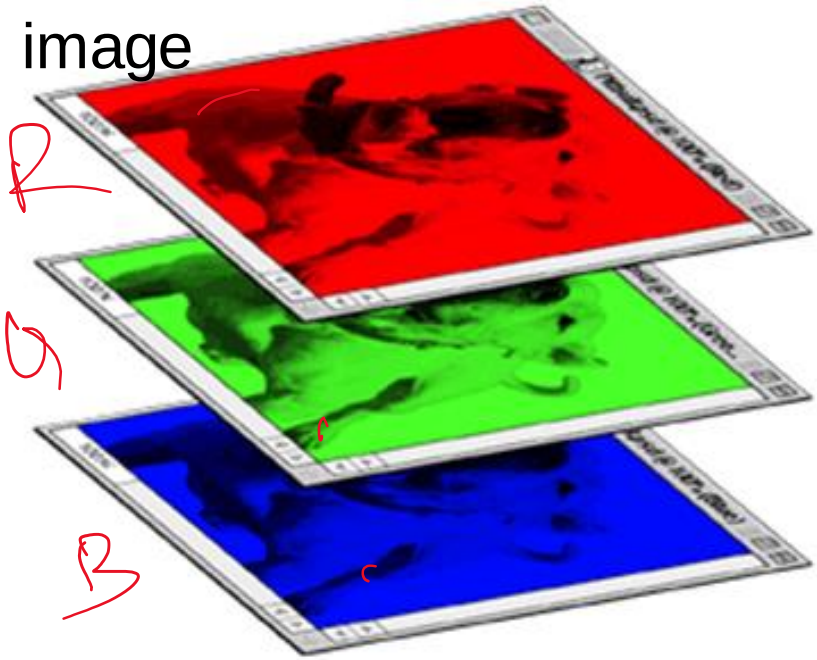


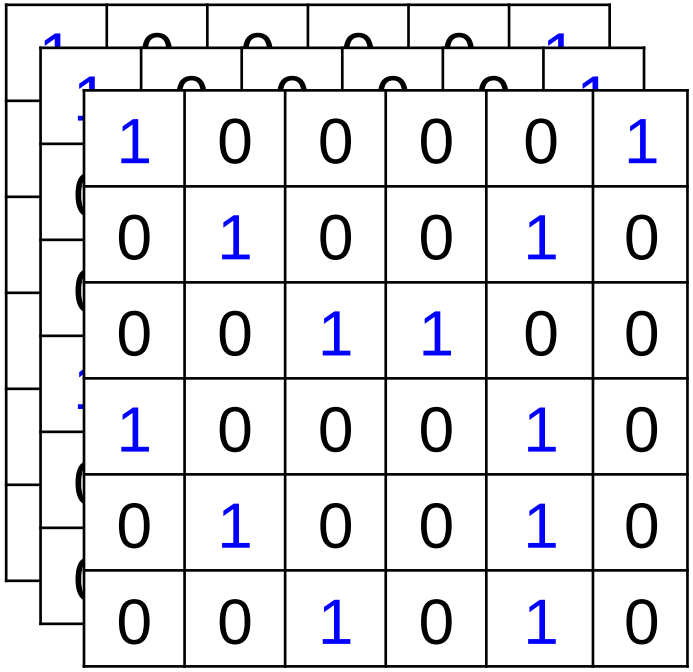
Filter 1



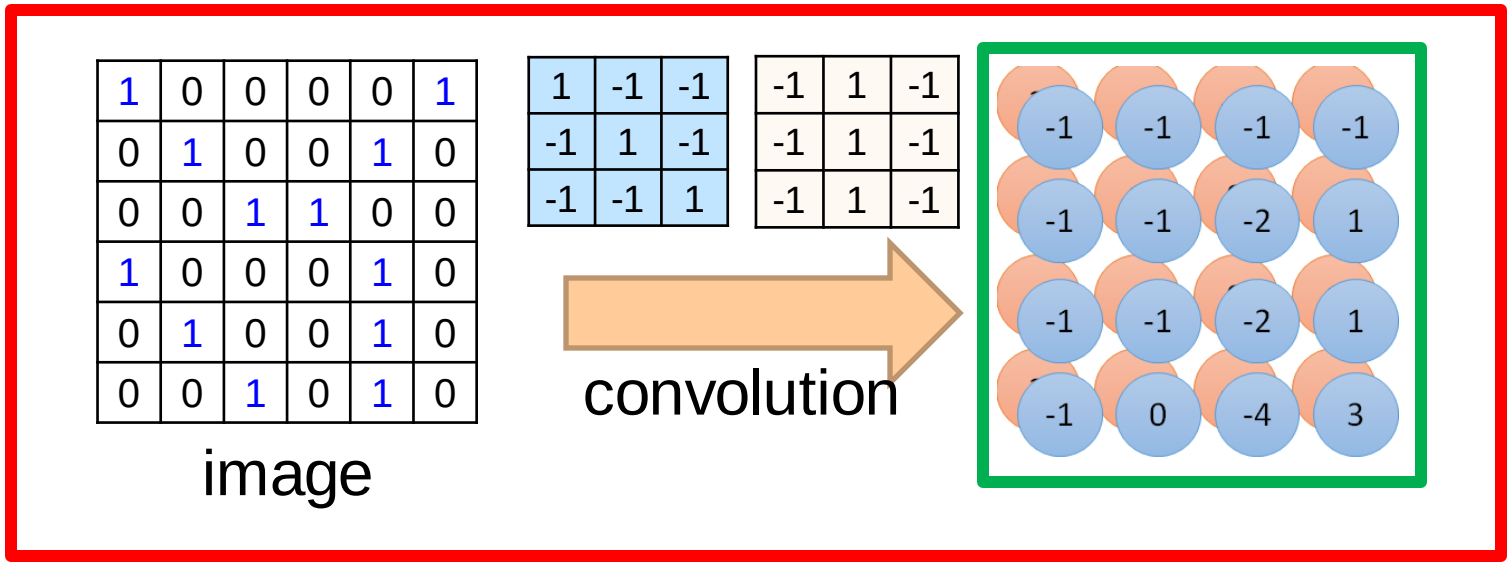
Filter 2

Colorful image

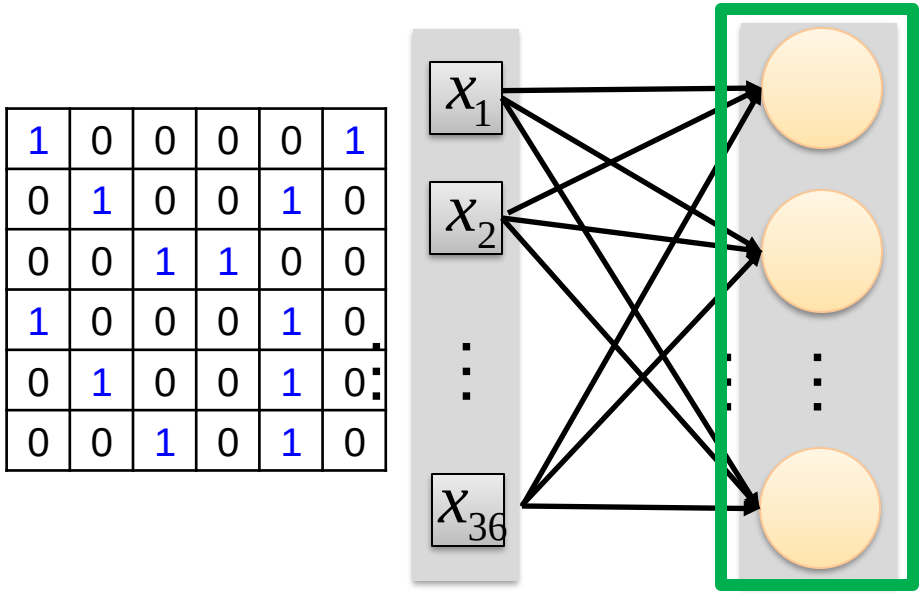


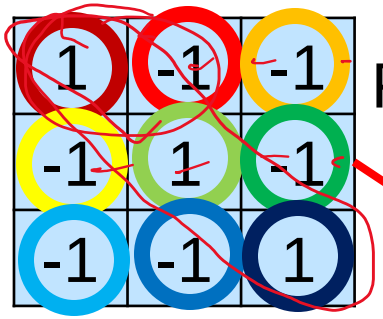


Convolution v.s. Fully Connected



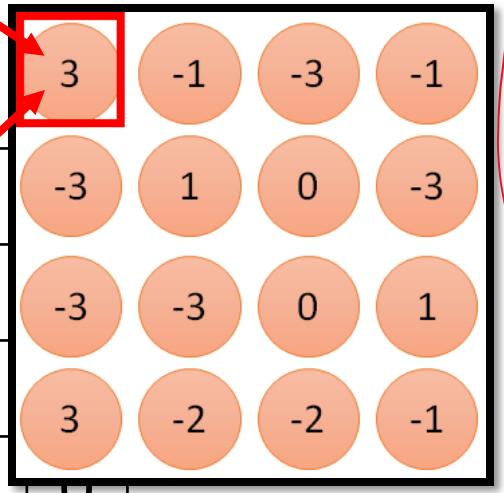
Fully-connected





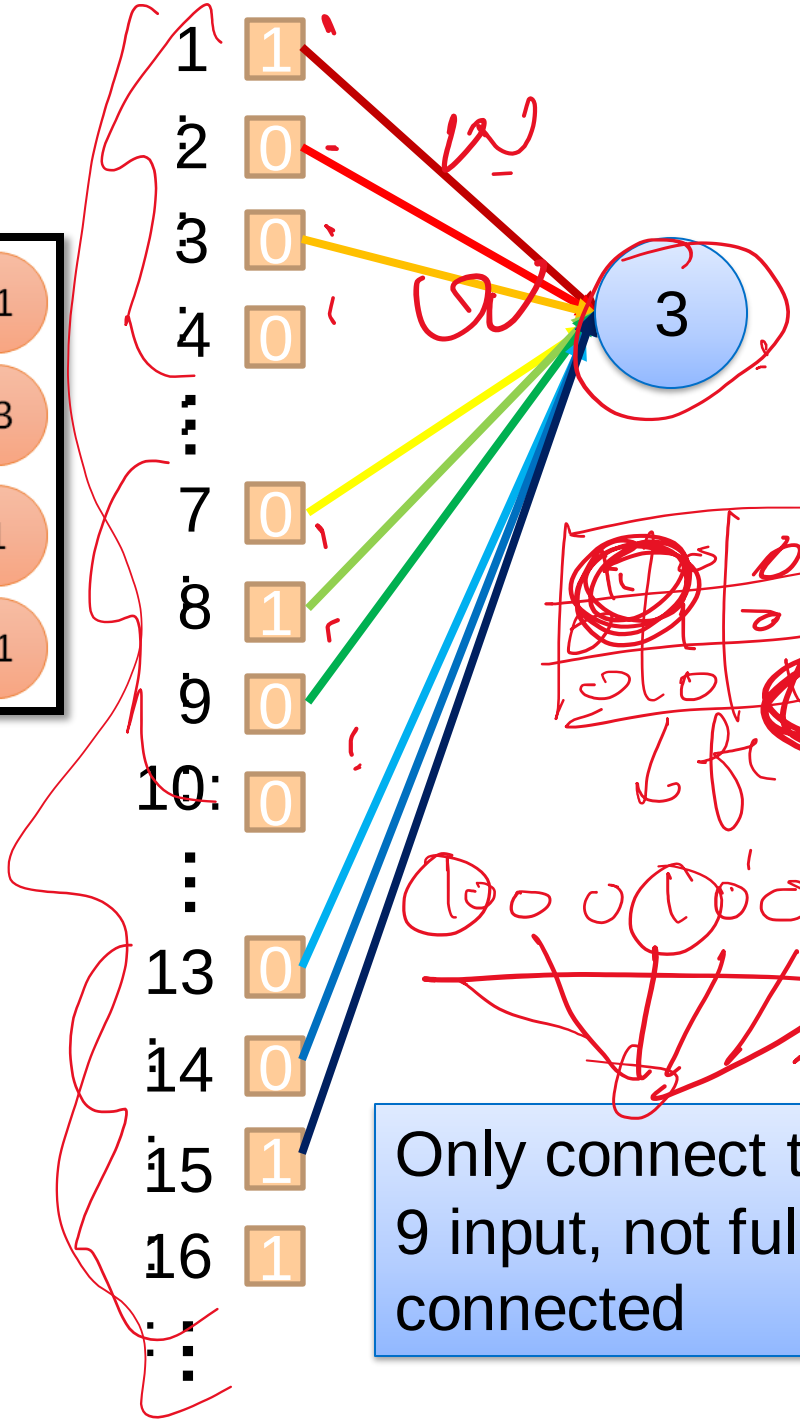
Filter 1

1	0	0	0	0	
0	1	0	0	1	
0	0	1	1	0	
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

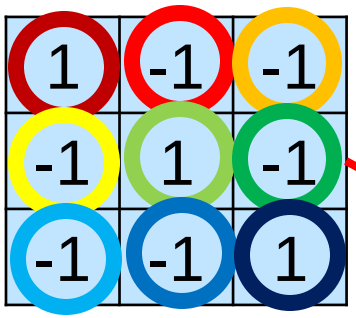


6 x 6 image = 36

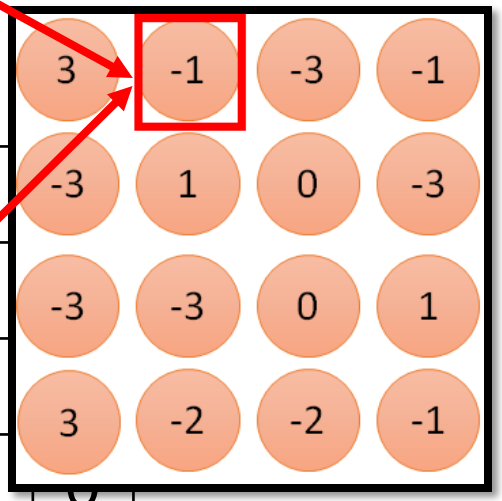
Less parameters!



Only connect to 9 input, not fully connected



Filter 1

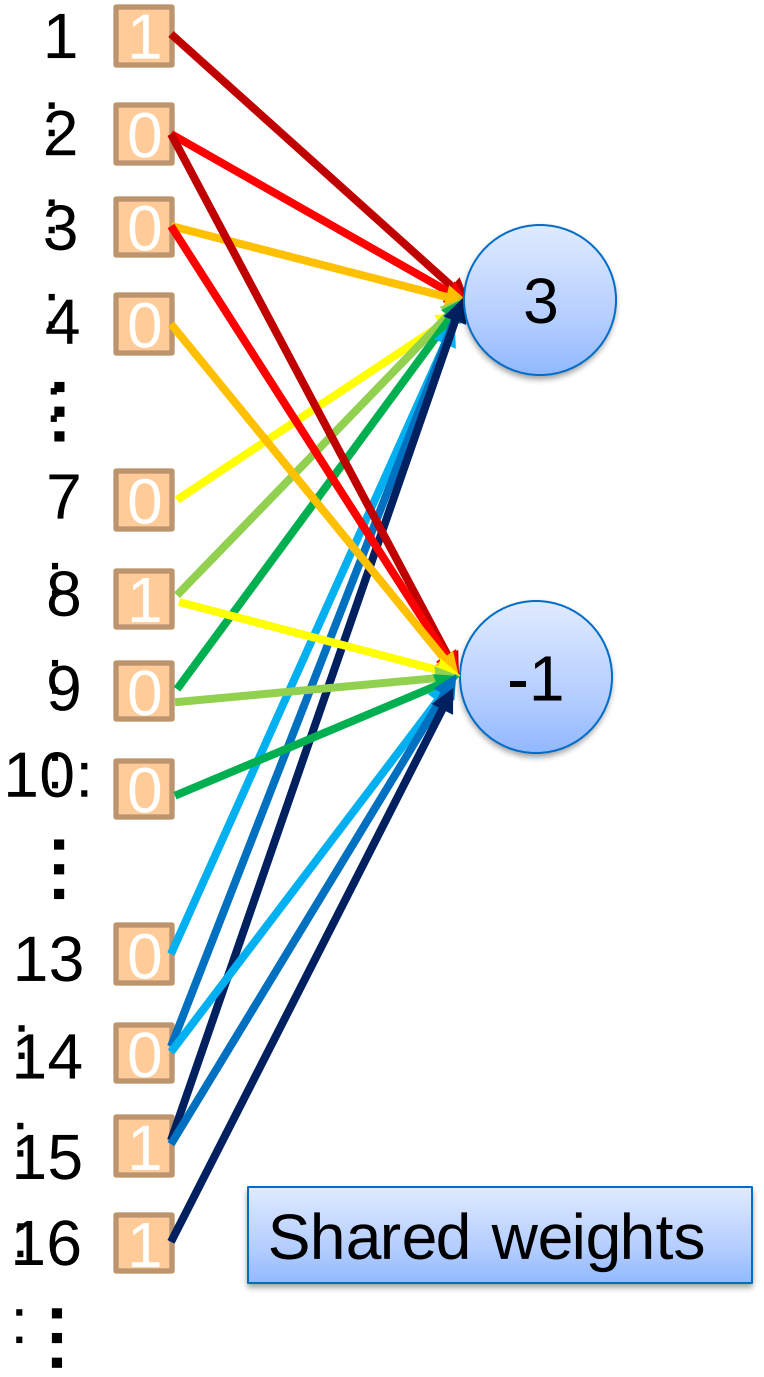


1	0	0	0	0	
0	1	0	0	1	
0	0	1	1	0	
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

Less parameters!

Even less parameters!

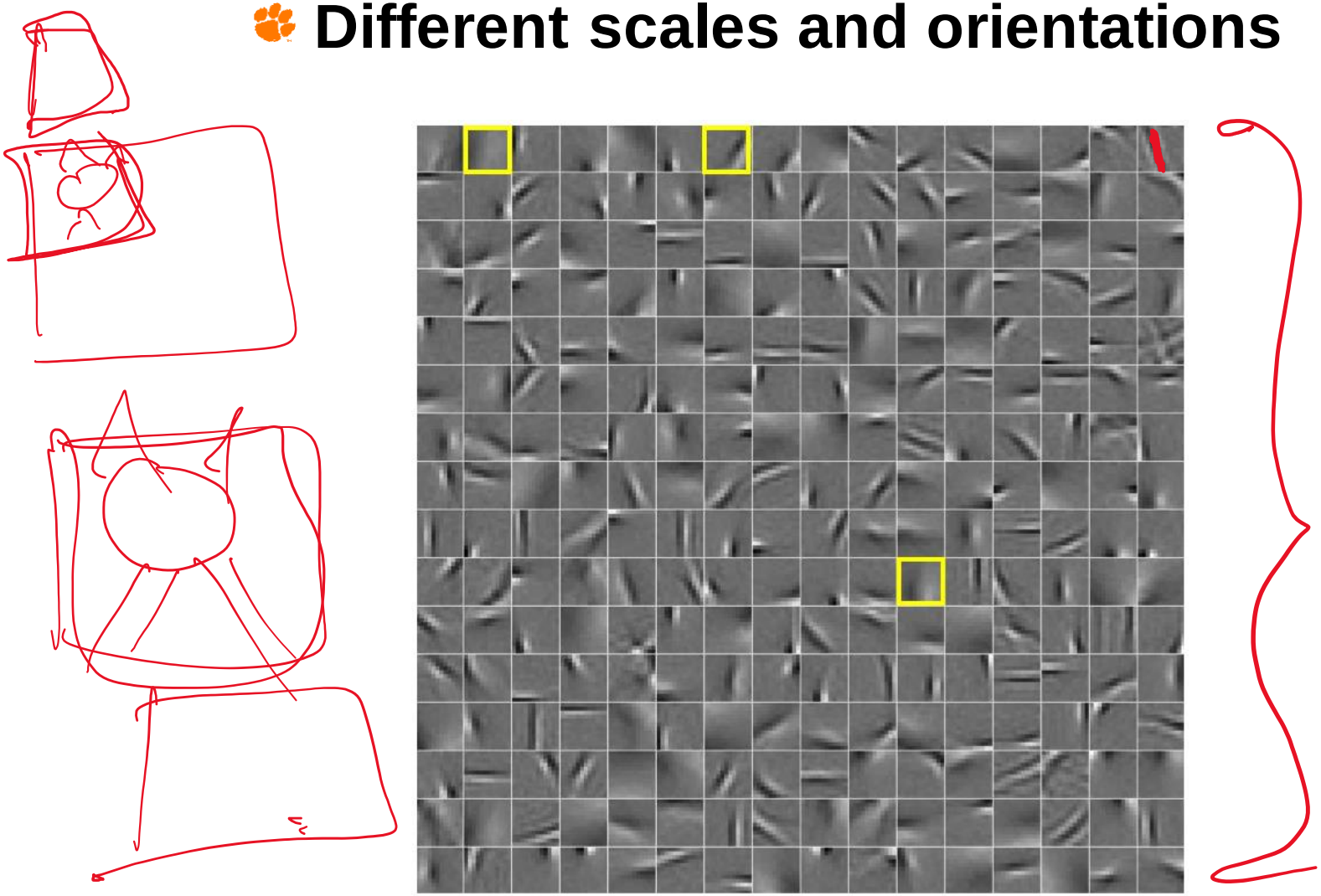


Complexity of Convolution

- **Complexity of convolution operator is** $n\log(n)$, for n inputs.
  - Uses Fast-Fourier-Transform (FFT)
- **For two-dimension, each convolution takes** $MN\log(MN)$ time, where the size of input is MN.

Convolutional kernels

- Different scales and orientations



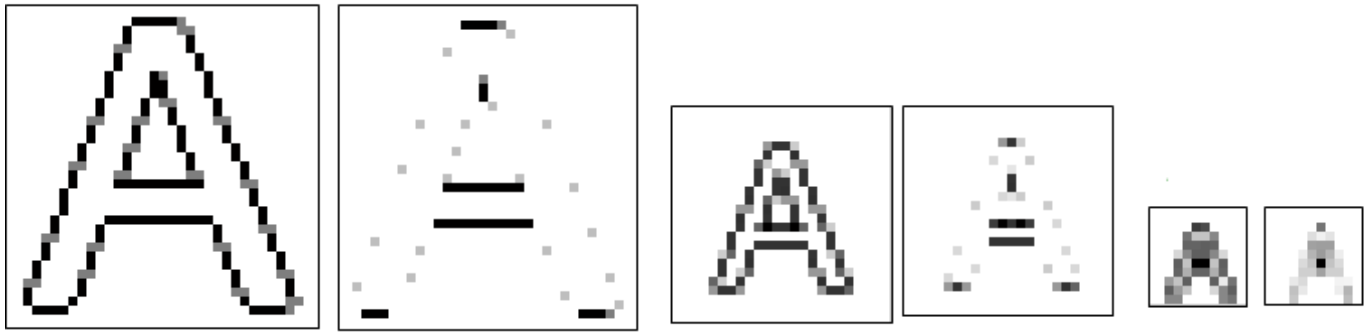
Edge Detector

- **Randomly initialized filter**
  - uniform distribution [-1/fan-in, 1/fan-in]
    - fan-in: the number of inputs to a hidden unit



Sub-Sampling (Pooling) Layer

- **Reduce the spatial resolution of each feature map**
  - A certain degree of shift and distortion invariance is achieved.



Sub-Sampling (Pooling) Layer

- A layer which reduces inputs of different size, to a fixed size.
- Pooling
- Different variations
  - Max pooling

$$h_i[n] = \max_{i \in N(n)} \tilde{h}[i]$$

  - Average pooling

$$h_i[n] = \frac{1}{n} \sum_{i \in N(n)} \tilde{h}[i]$$

  - L2-pooling

$$h_i[n] = \frac{1}{n} \sqrt{\sum_{i \in N(n)} \tilde{h}^2[i]}$$

2	2	4	4
2	4	8	4
4	4	4	1
6	10	3	4

Max pooling

4	8
10	4

Mean Pooling

2.5	5
6	3

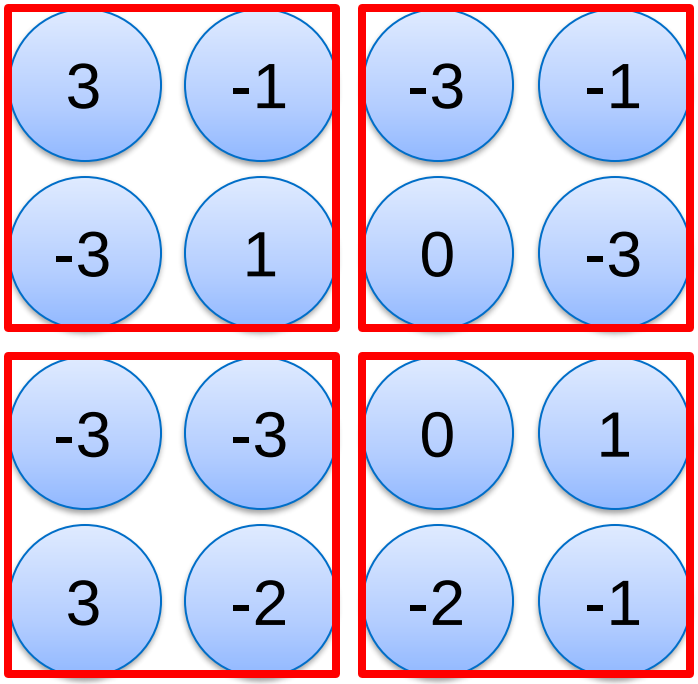
CNN – Max Pooling

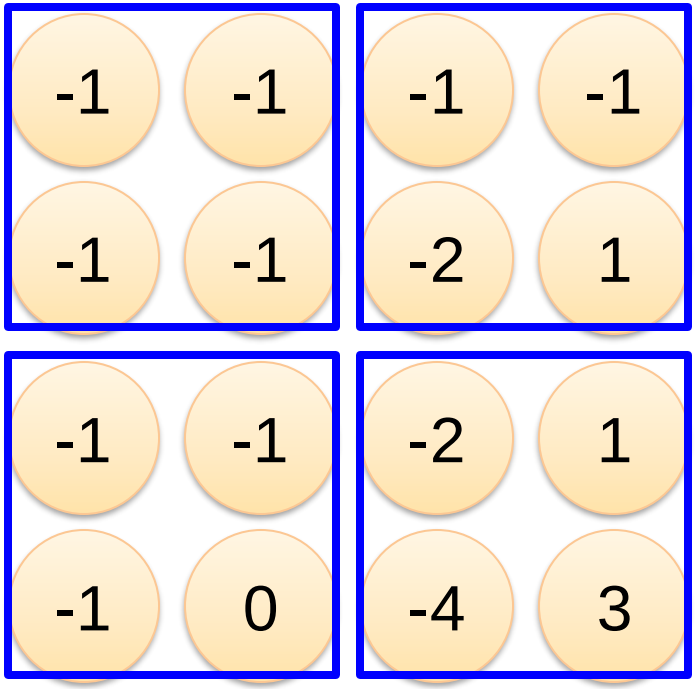
1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

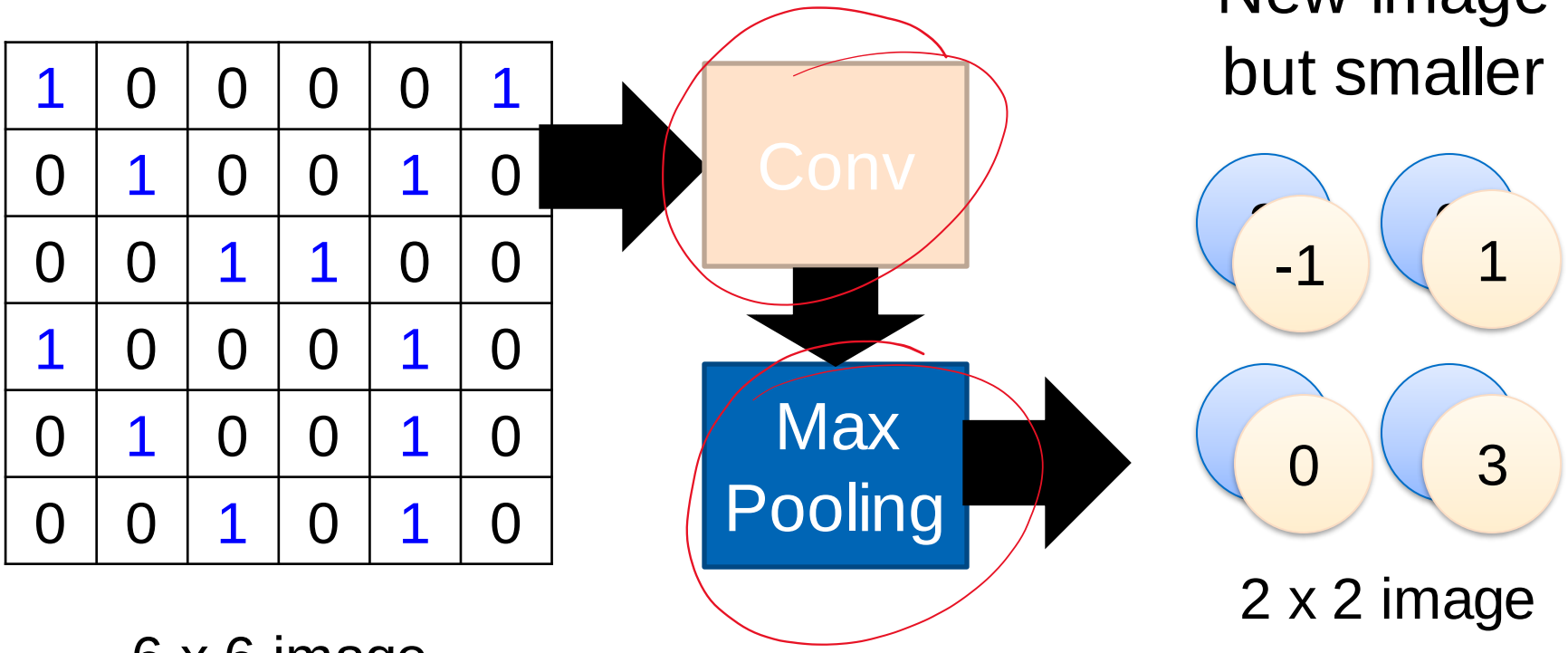
-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

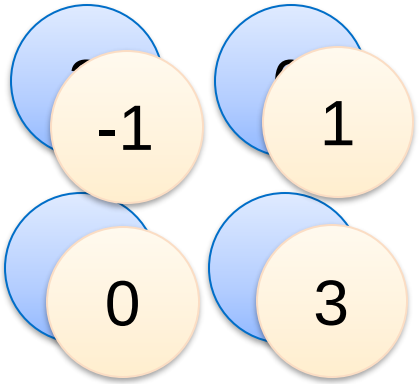




CNN – Max Pooling



The whole CNN



A new image

Convolution

Max Pooling

Convolution

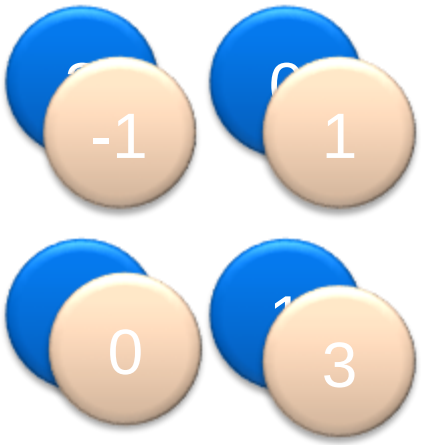
Max Pooling

Can repeat many times

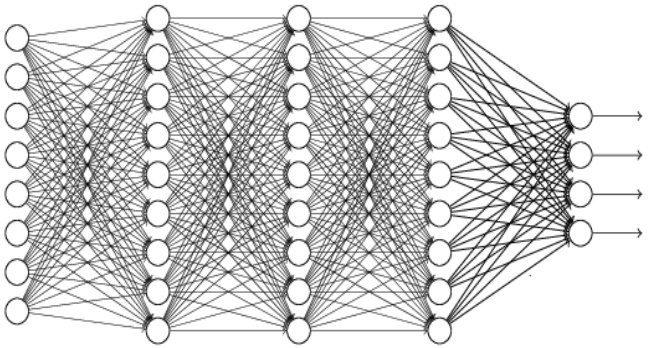
Smaller than the original image

The number of the channel is the number of filters

Flatten



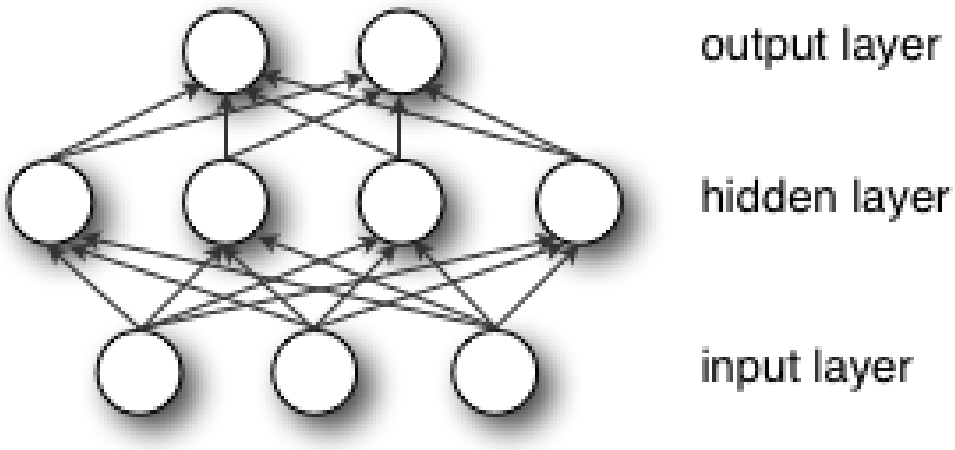
Flatten



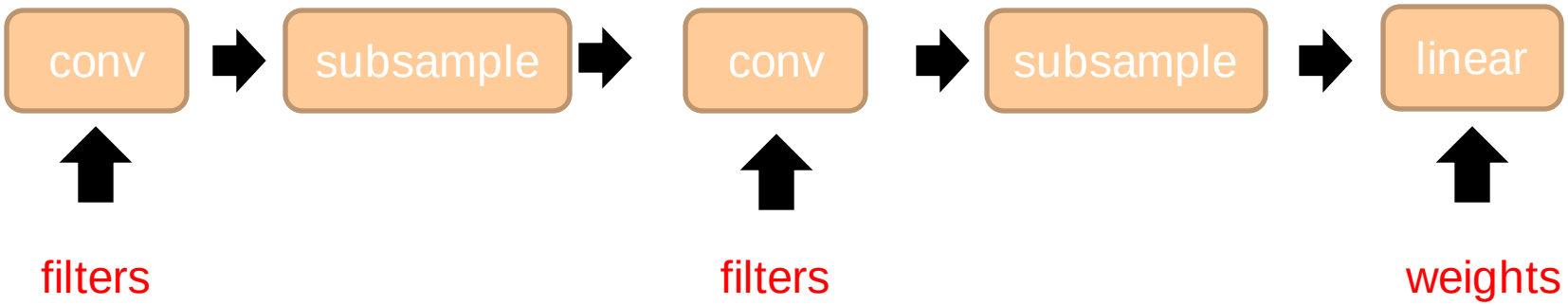
Fully Connected Feedforward network

Fully connected layers

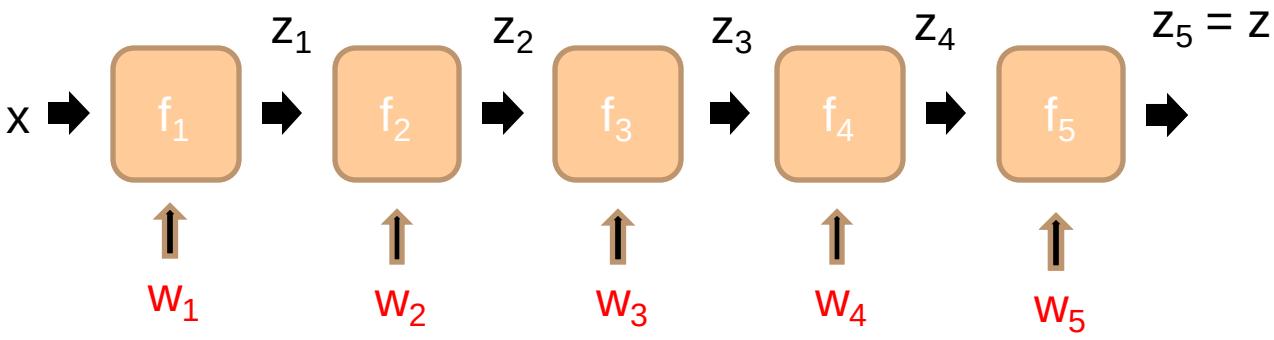
- **Correspond to a traditional MLP**
  - hidden layer + logistic regression



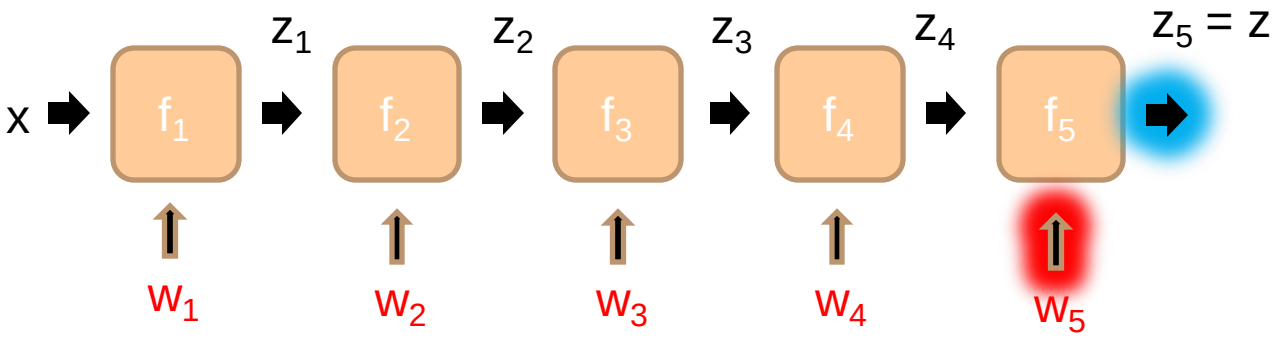
Convolutional networks



The gradient of CNN

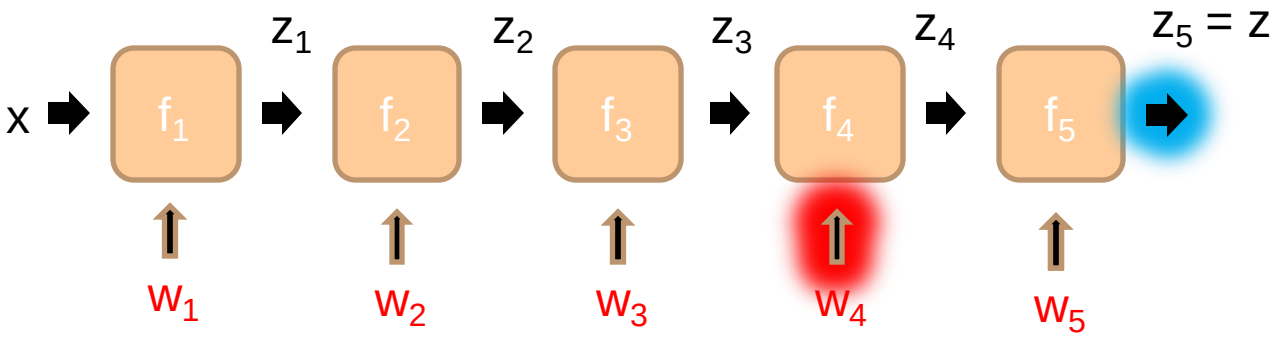


The gradient of CNN



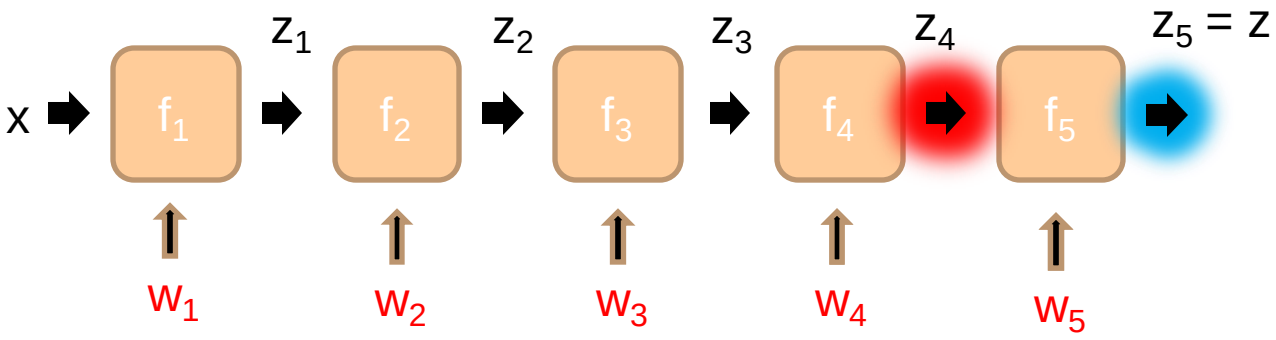
$$\frac{\partial z}{\partial w_5}$$

The gradient of CNN



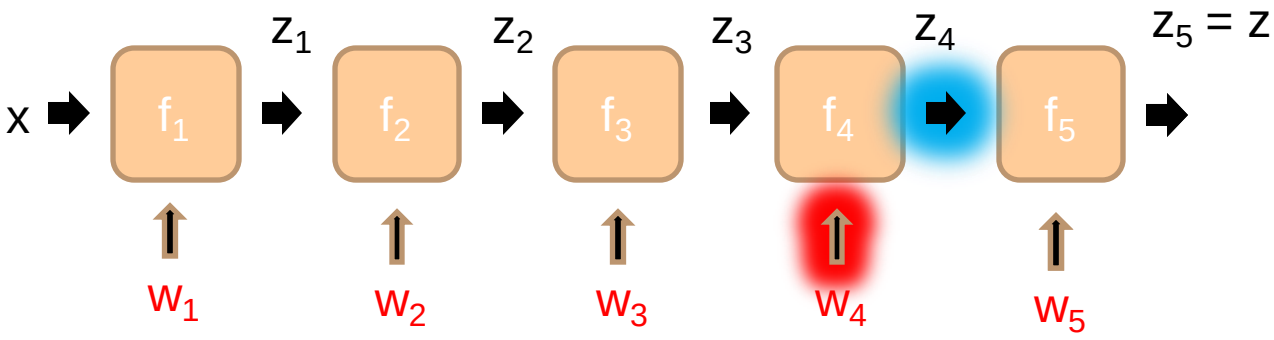
$$\frac{\partial z}{\partial w_4}$$

The gradient of CNN



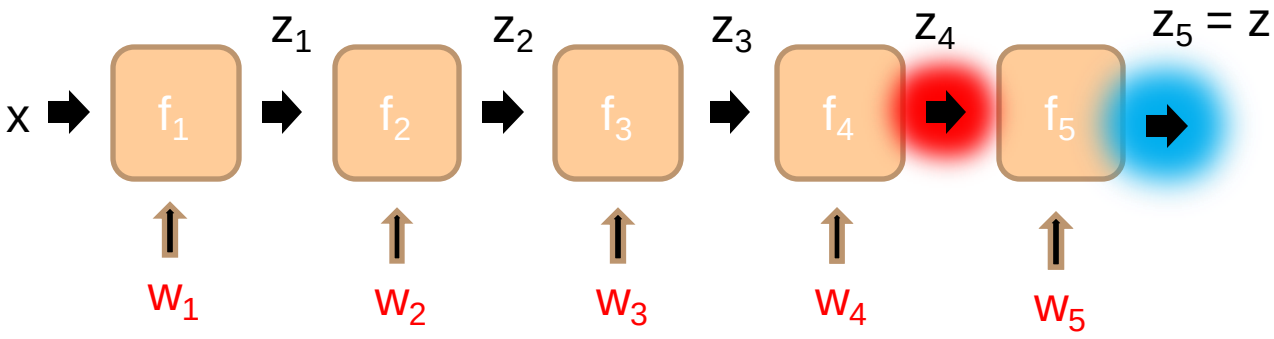
$$\frac{\partial z}{\partial w_4} = \frac{\partial z}{\partial z_4}\frac{\partial z_4}{\partial w_4}$$

The gradient of CNN



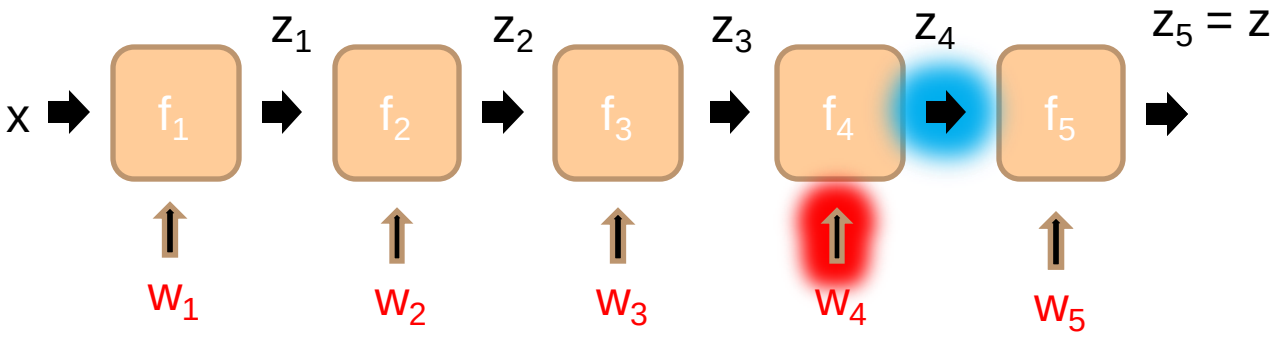
$$\frac{\partial z}{\partial w_4} = \frac{\partial z}{\partial z_4}\frac{\partial z_4}{\partial w_4}$$

The gradient of CNN



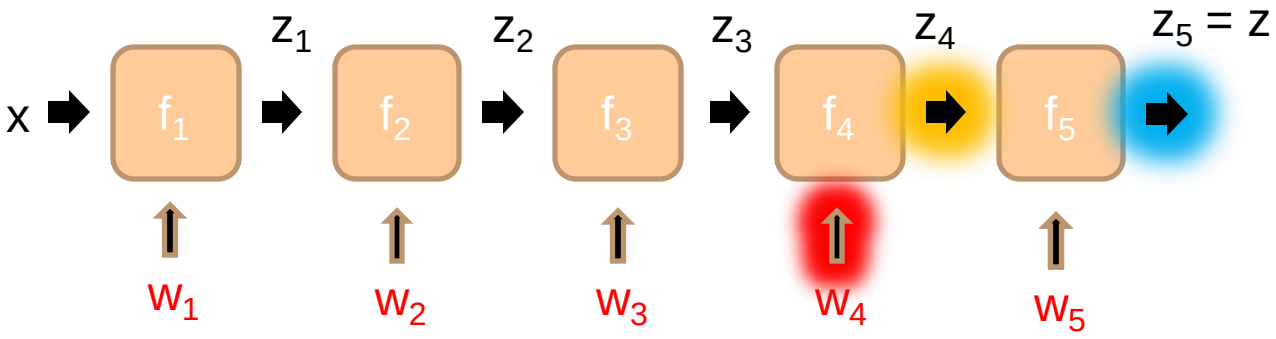
$$\frac{\partial z}{\partial w_4} = \frac{\partial z}{\partial z_4}\frac{\partial z_4}{\partial w_4} = \frac{\partial f_5(z_4, w_5)}{\partial z_4}\frac{\partial f_4(z_3, w_4)}{\partial w_4}$$

The gradient of CNN



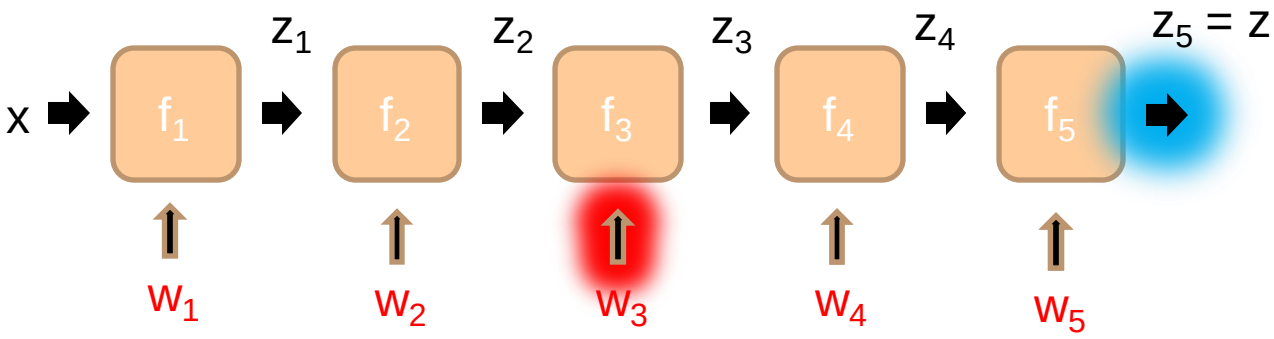
$$\frac{\partial z}{\partial w_4} = \frac{\partial z}{\partial z_4}\frac{\partial z_4}{\partial w_4} = \frac{\partial f_5(z_4, w_5)}{\partial z_4}\frac{\partial f_4(z_3, w_4)}{\partial w_4}$$

The gradient of CNN



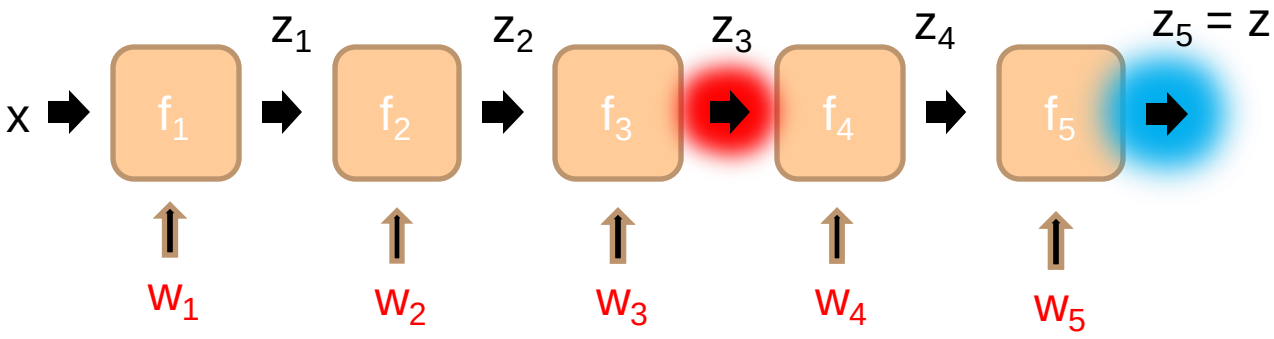
$$\frac{\partial z}{\partial w_4} = \frac{\partial z}{\partial z_4}\frac{\partial z_4}{\partial w_4} = \frac{\partial f_5(z_4, w_5)}{\partial z_4}\frac{\partial f_4(z_3, w_4)}{\partial w_4}$$

The gradient of CNN



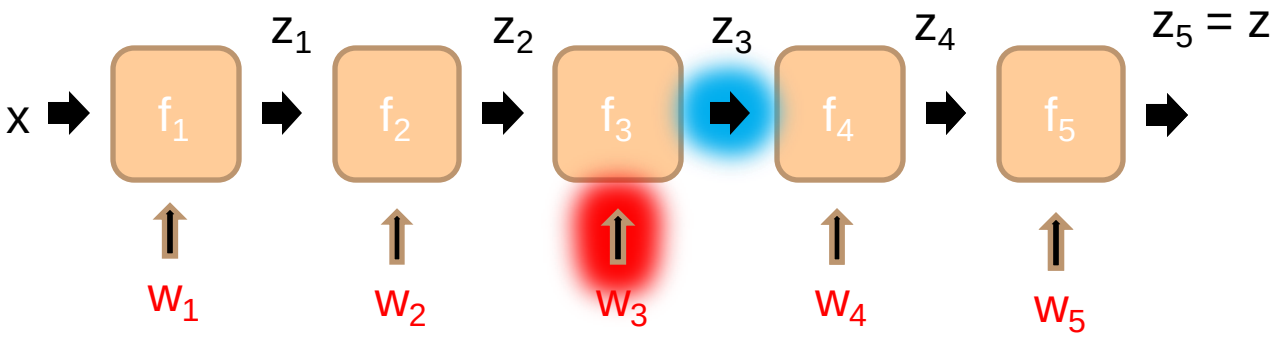
$$\frac{\partial z}{\partial w_3}$$

The gradient of CNN



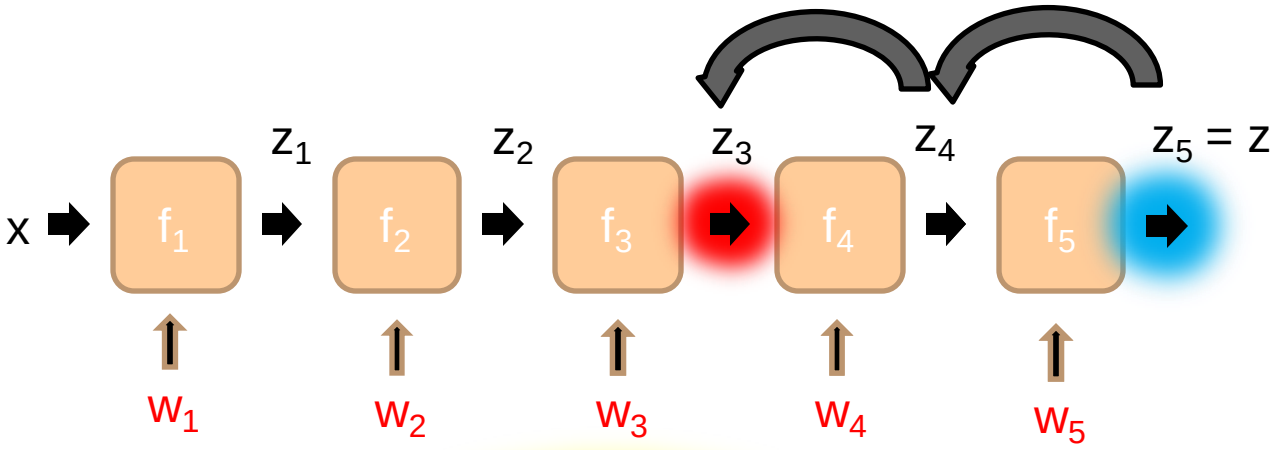
$$\frac{\partial z}{\partial w_3} = \frac{\partial z}{\partial z_3}\frac{\partial z_3}{\partial w_3}$$

The gradient of CNN



$$\frac{\partial z}{\partial w_3} = \frac{\partial z}{\partial z_3}\frac{\partial z_3}{\partial w_3}$$

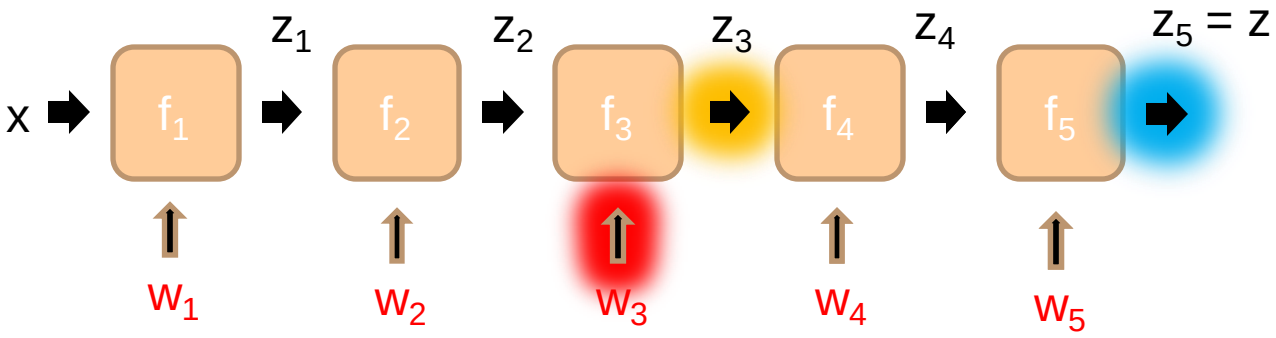
The gradient of CNN



$$\frac{\partial z}{\partial z_3} = \frac{\partial z}{\partial z_4}\frac{\partial z_4}{\partial z_3}$$

$$\frac{\partial z}{\partial w_3} = \frac{\partial z}{\partial z_3}\frac{\partial z_3}{\partial w_3}$$

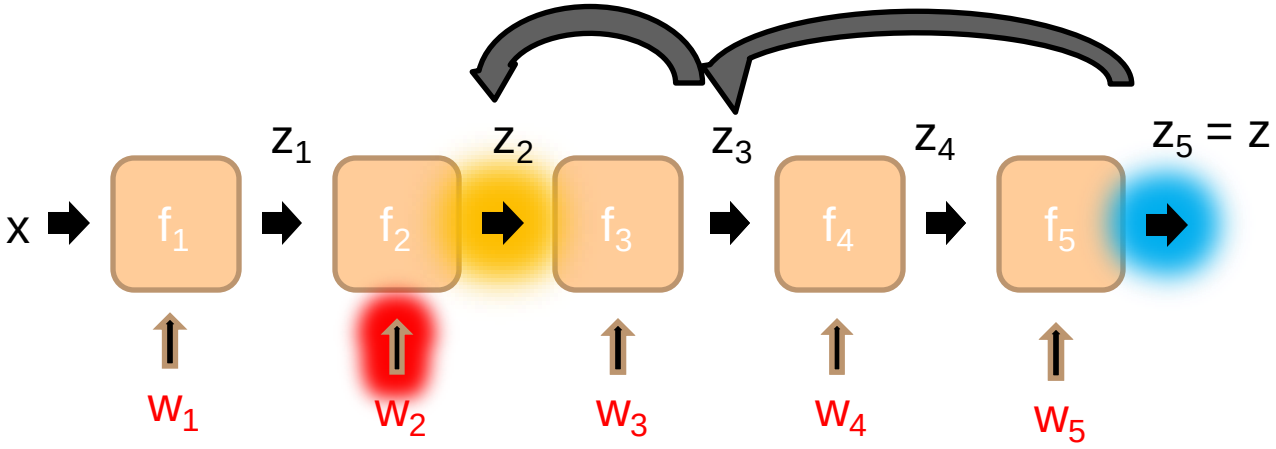
The gradient of CNN



$$\frac{\partial z}{\partial z_3} = \frac{\partial z}{\partial z_4}\frac{\partial z_4}{\partial z_3}$$

$$\frac{\partial z}{\partial w_3} = \frac{\partial z}{\partial z_3}\frac{\partial z_3}{\partial w_3}$$

The gradient of CNN

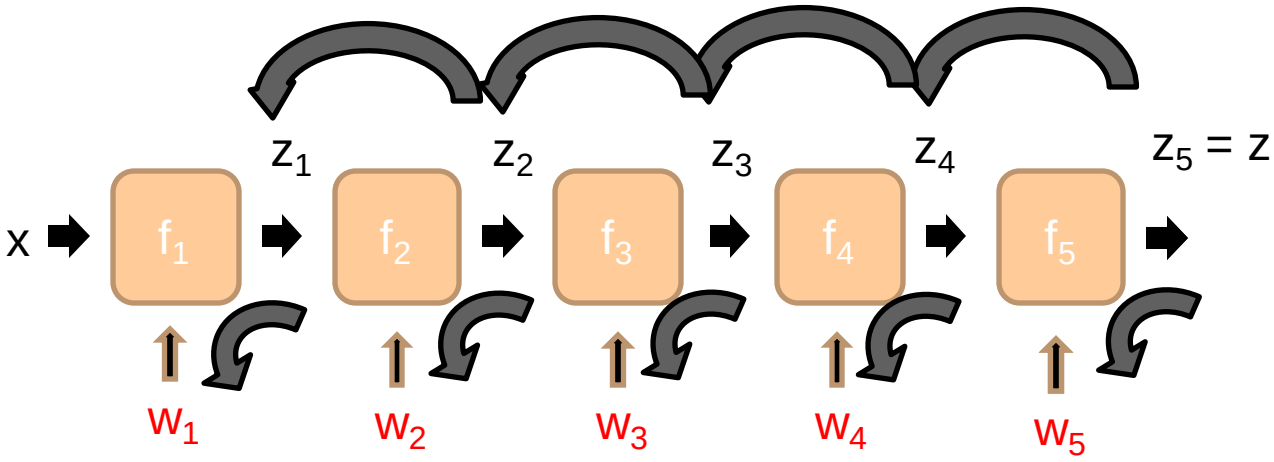


$$\frac{\partial z}{\partial z_2} = \frac{\partial z}{\partial z_3}\frac{\partial z_3}{\partial z_2}$$

$$\frac{\partial z}{\partial w_2} = \frac{\partial z}{\partial z_2}\frac{\partial z_2}{\partial w_2}$$

Recurrence going backward!!

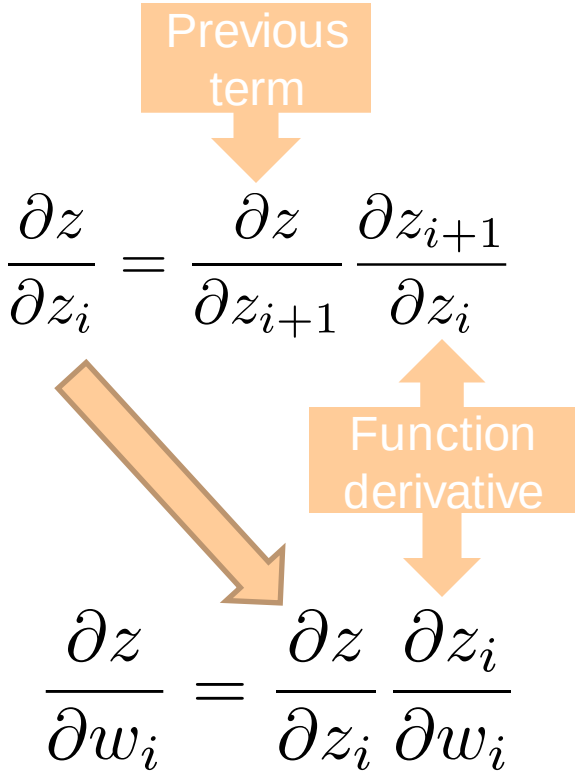
The gradient of CNN



Backpropagation

Backpropagation for a sequence of functions

$$z_i = f_i(z_{i-1}, w_i)$$
$$z_0 = x$$
$$z = z_n$$



Backpropagation for a sequence of functions

$$z_i = f_i(z_{i-1}, w_i) \qquad z_0 = x \qquad z = z_n$$

- **Assume we can compute partial derivatives of each function**

$$\frac{\partial z_i}{\partial z_{i-1}} = \frac{\partial f_i(z_{i-1}, w_i)}{\partial z_{i-1}} \qquad \frac{\partial z_i}{\partial w_i} = \frac{\partial f_i(z_{i-1}, w_i)}{\partial w_i}$$

- **Use $g(z_i)$ to store gradient of z w.r.t z_i, $g(w_i)$ for w_i**
- **Calculate g_i by iterating backwards**

$$g(z_n) = \frac{\partial z}{\partial z_n} = 1 \qquad g(z_{i-1}) = \frac{\partial z}{\partial z_i}\frac{\partial z_i}{\partial z_{i-1}} = g(z_i)\frac{\partial z_i}{\partial z_{i-1}}$$

- **Use gi to compute gradient of parameters**

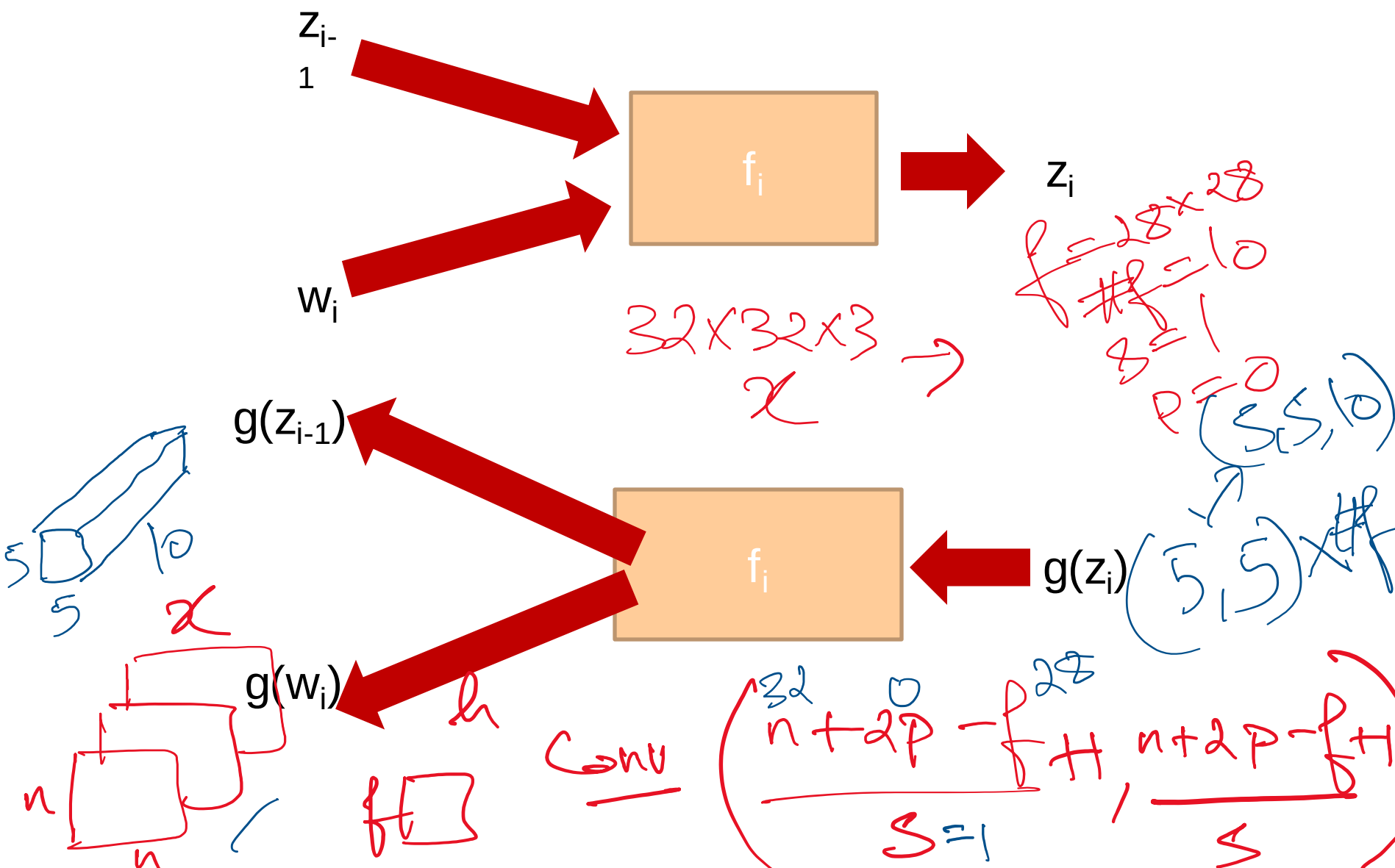
$$g(w_i) = \frac{\partial z}{\partial z_i}\frac{\partial z_i}{\partial w_i} = g(z_i)\frac{\partial z_i}{\partial w_i}$$

Backpropagation for a sequence of functions

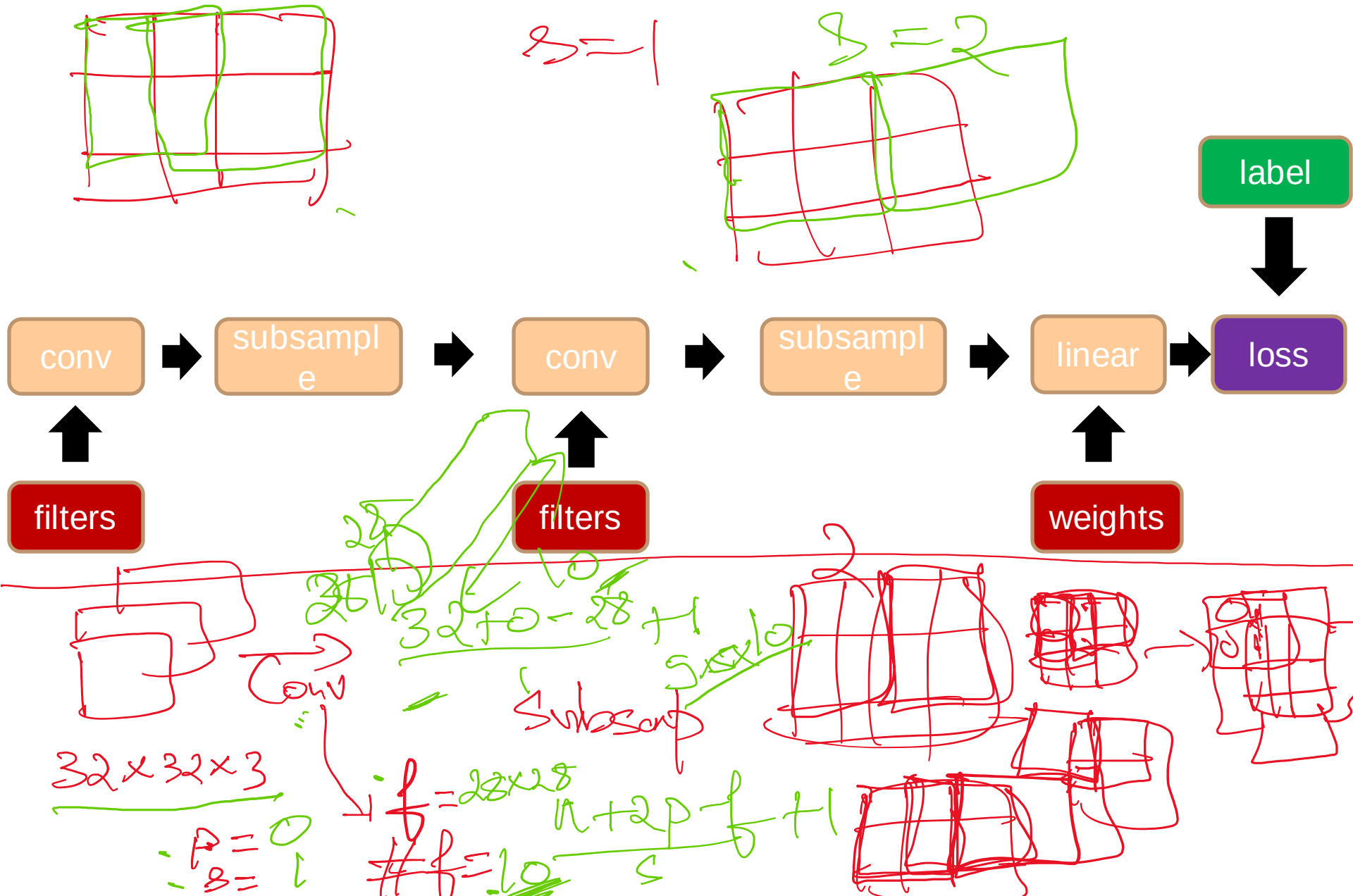
- **Each "function" has a "forward" and "backward" module**
- **Forward module for f_i**
  - takes z_{i-1} and weight w_i as input
  - produces z_i as output
- **Backward module for f_i**
  - takes $g(z_i)$ as input
  - produces $g(z_{i-1})$ and $g(w_i)$ as output

$$g(z_{i-1}) = g(z_i)\frac{\partial z_i}{\partial z_{i-1}} \qquad g(w_i) = g(z_i)\frac{\partial z_i}{\partial w_i}$$

Backpropagation for a sequence of functions



Loss as a function



Loss function

- **Likelihood:**

$$\mathcal{L}(\theta = \{W, b\}, \mathcal{D}) = \sum_{i=0}^{|\mathcal{D}|} \log(P(Y = y^{(i)}|x^{(i)}, W, b)$$

- **Negative log-likelihood:**

$$\ell(\theta = \{W, b\}, \mathcal{D}) = -\mathcal{L}(\theta = \{W, b\}, \mathcal{D})$$

Stochastic gradient descent

- Gradient on single example = unbiased sample of true gradient
- Idea: at each iteration sample single example $x^{(t)}$

$$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \lambda \nabla_{\mathbf{w}} L(h_{\mathbf{w}}(x^{(t)}), y^{(t)})$$

step size

- Con: variance in estimate of gradient → slow convergence, jumping near optimum

Minibatch stochastic gradient descent

- Compute gradient on a small batch of examples
- Same mean (=true gradient), but variance inversely proportional to minibatch size

$$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \lambda \frac{1}{|B^{(t)}|} \sum_{(x,y) \in B^{(t)}} \nabla_{\mathbf{w}} L(h_{\mathbf{w}}(x), y)$$

Momentum

- ***Average*** **multiple gradient steps**
- **Use *exponential averaging***

$$\mathbf{p}^{(t+1)} \leftarrow (1-\mu)\mathbf{p}^{(t)} - \mu \frac{1}{|B^{(t)}|} \sum_{(x,y)\in B^{(t)}} \nabla_{\mathbf{w}} L(h_{\mathbf{w}}(x), y)$$

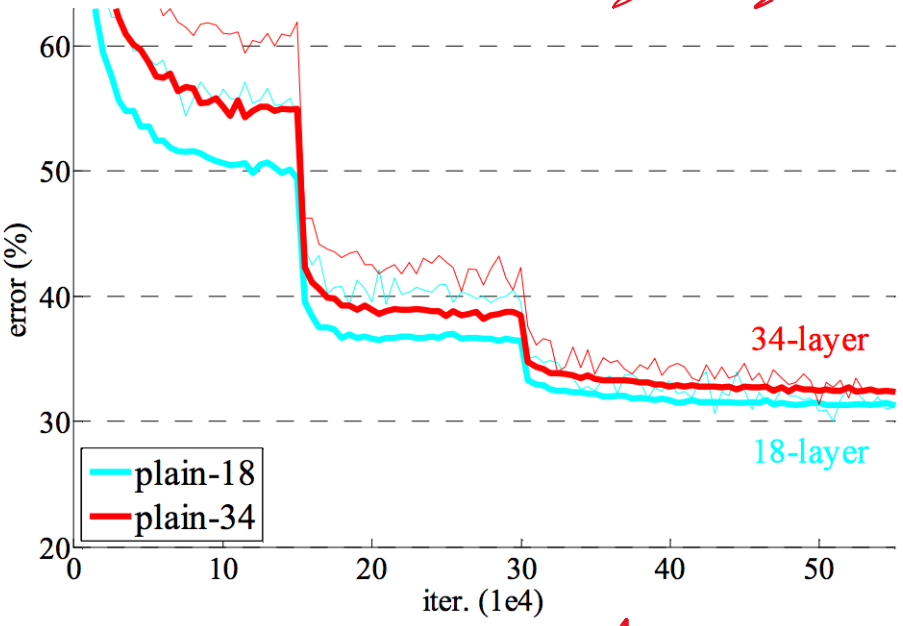
$$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \lambda \mathbf{p}^{(t+1)}$$

Weight decay

- Add $-aw^{(t)}$ to the gradient
- Prevents $w^{(t)}$ from growing to infinity
- Equivalent to L2 regularization of weights

Learning rate decay

- **Large step size / learning rate**
  - Faster convergence initially
  - Bouncing around at the end because of noisy gradients
- **Learning rate must be decreased over time**
- **Usually done in steps**



Convolutional network training

- Initialize network
- Sample *minibatch* of images
- Forward pass to compute loss
- Backpropagate loss to compute gradient
- Combine gradient with momentum and weight decay
- Take step according to current learning rate

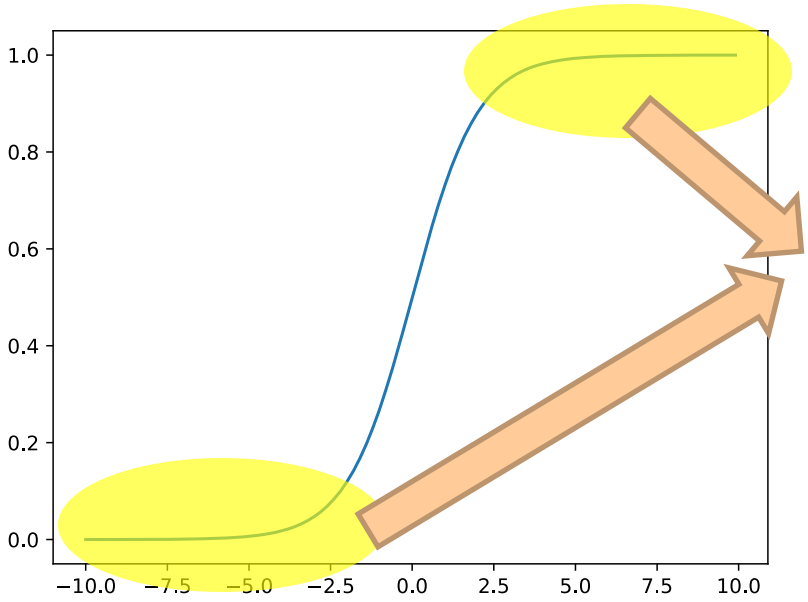
Vagaries of optimization

- **Non-convex**
  - Local optima
  - Sensitivity to initialization
- **Vanishing / exploding gradients**

$$\frac{\partial z}{\partial z_i} = \frac{\partial z}{\partial z_{n-1}} \frac{\partial z_{n-1}}{\partial z_{n-2}} \cdots \frac{\partial z_{i+1}}{\partial z_i}$$

  - If each term is (much) greater than 1 → *explosion of gradients*
  - If each term is (much) less than 1 → *vanishing gradients*

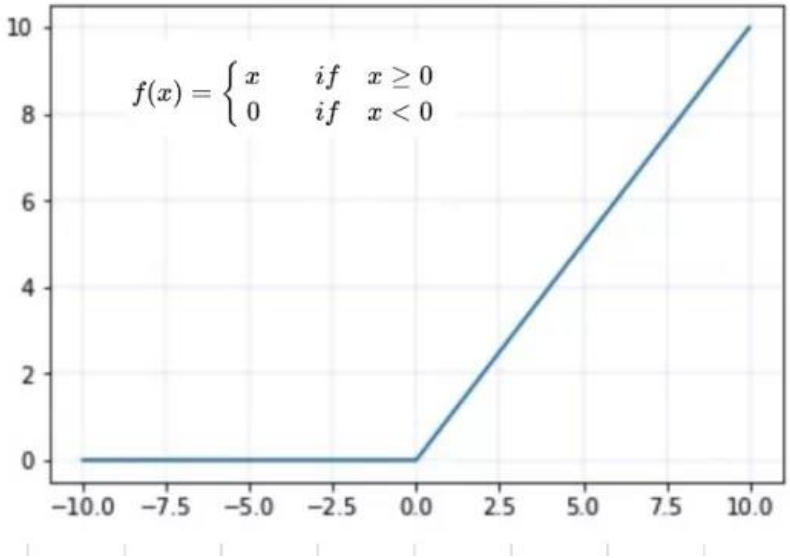
Sigmoids cause vanishing gradients

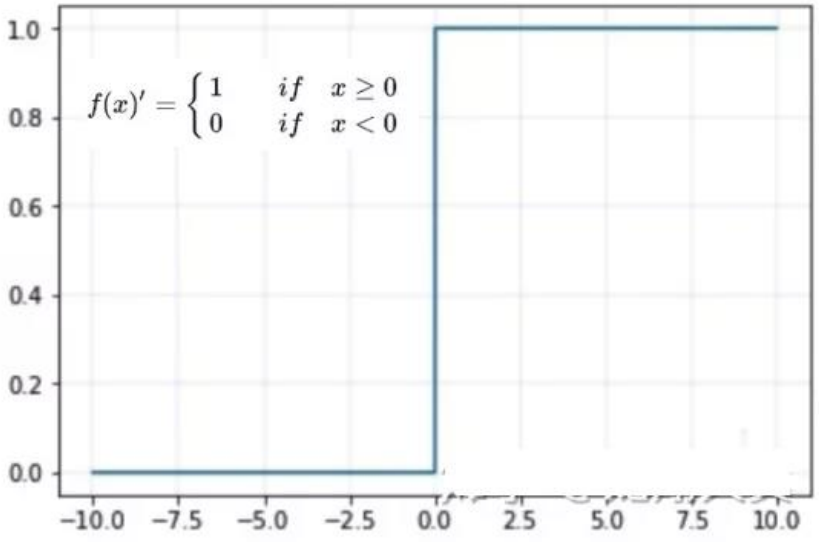


Gradient close to 0

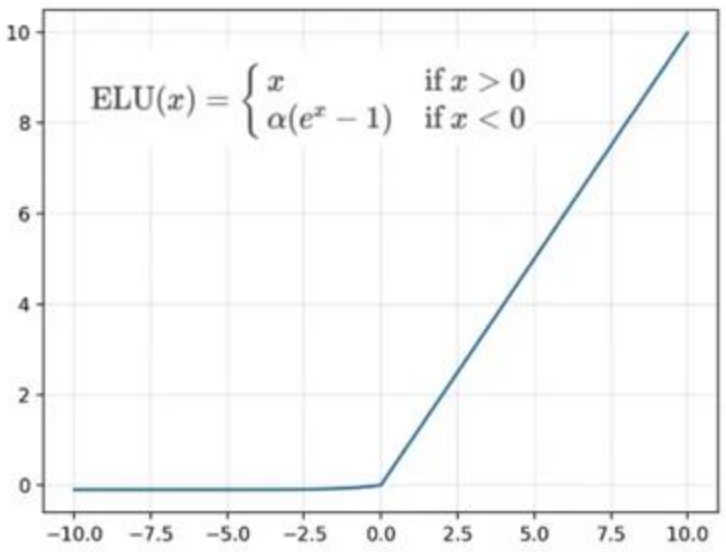
Rectified Linear Unit (ReLU)

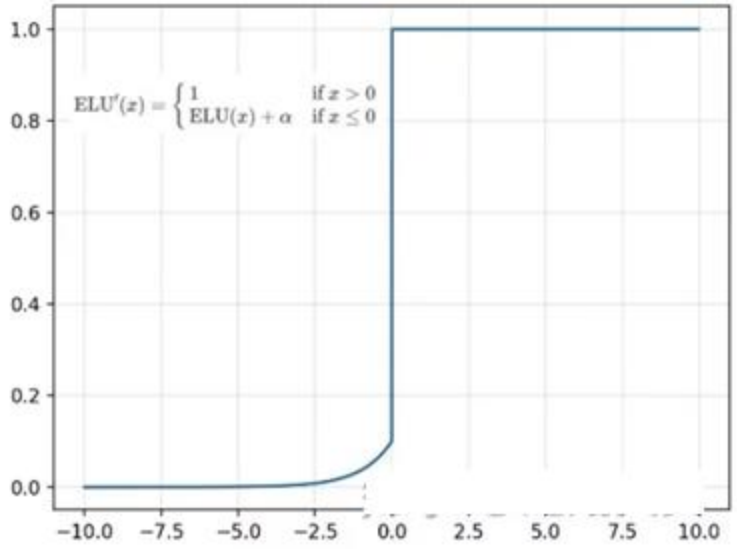
- **max (x,0)**
- **Also called half-wave rectification (signal processing)**
- **Advantage:**
  - Simple, easy to calculate
  - Gradient is 1, can partially solve the gradient vanishing problem.
- **Problem: dying Relu: some part of neural network will never be updated**





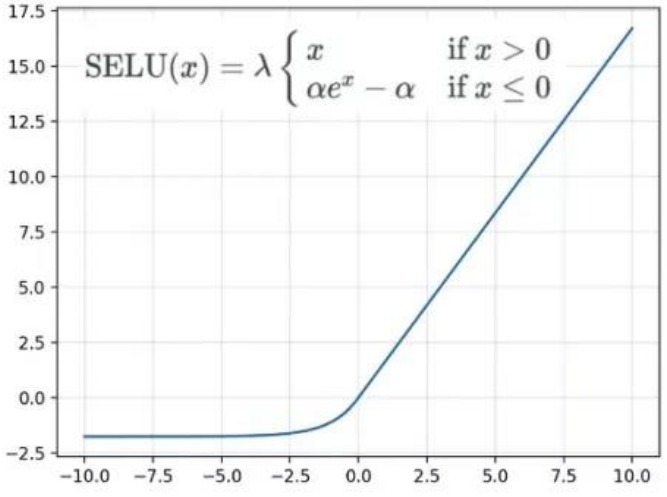
Exponential Linear Unit (ELU)

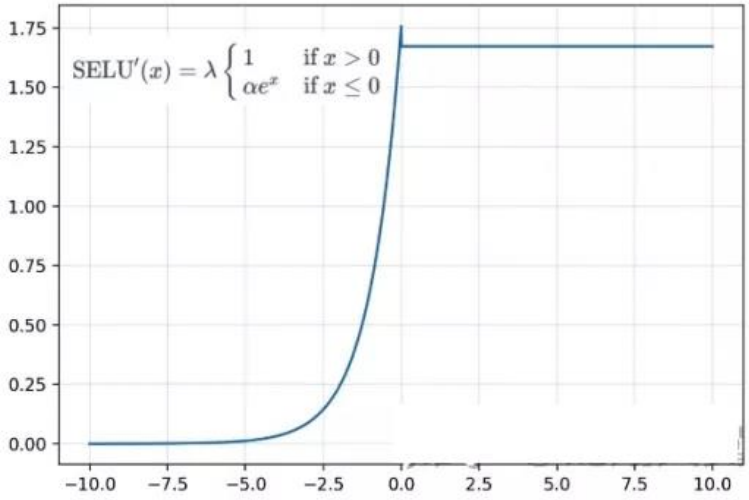




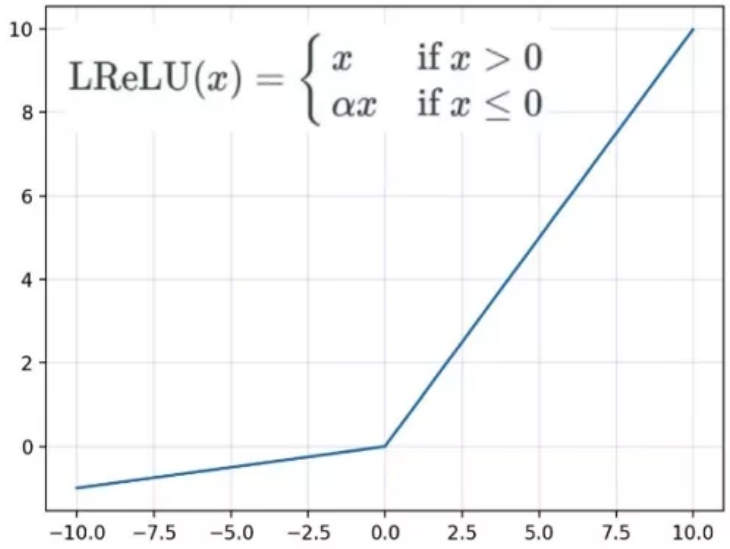
- **Advantage: avoid the dying Relu problem**
- **Disadvantage: high computational cost.**

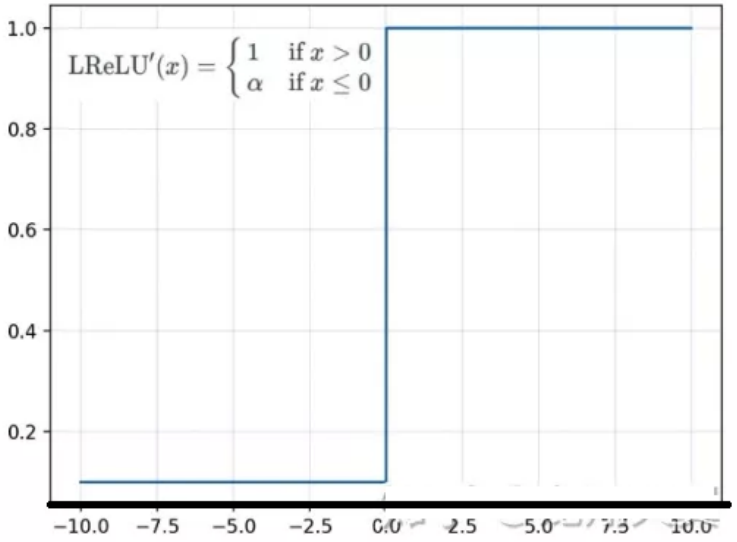
SELU


$\mathrm{SELU}(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leq 0 \end{cases}$
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


$\mathrm{SELU}'(x) = \lambda \begin{cases} 1 & \text{if } x > 0 \\ \alpha e^x & \text{if } x \leq 0 \end{cases}$
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

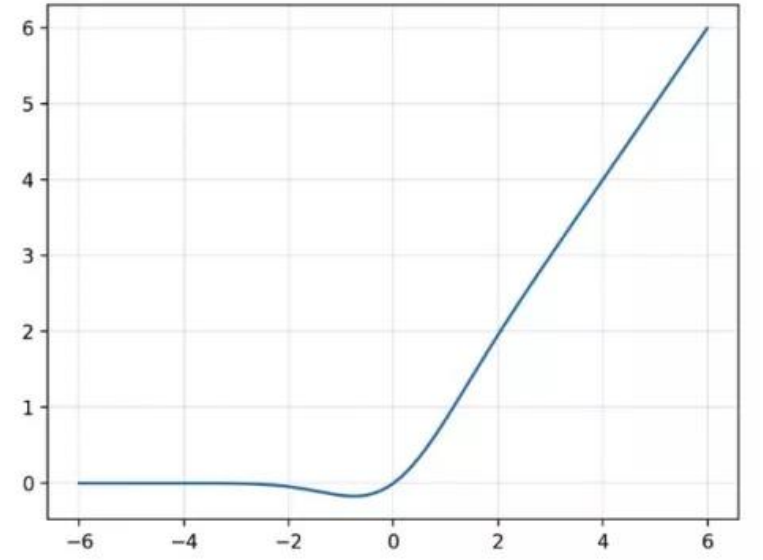
Leaky Relu


$\mathrm{LReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$
10
8
6
4
2
0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

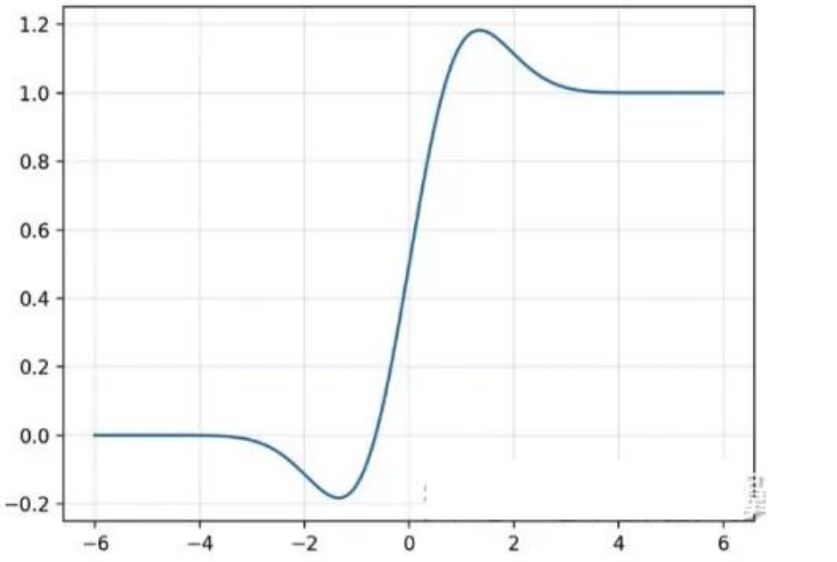

$\mathrm{LReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{if } x \leq 0 \end{cases}$
1.0
0.8
0.6
0.4
0.2
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

Gaussian Error Linear Unit (Gelu)

$$\mathrm{GELU}(x) = 0.5x\left(1 + \tanh\left(\sqrt{2/\pi}(x + 0.044715x^3)\right)\right)$$

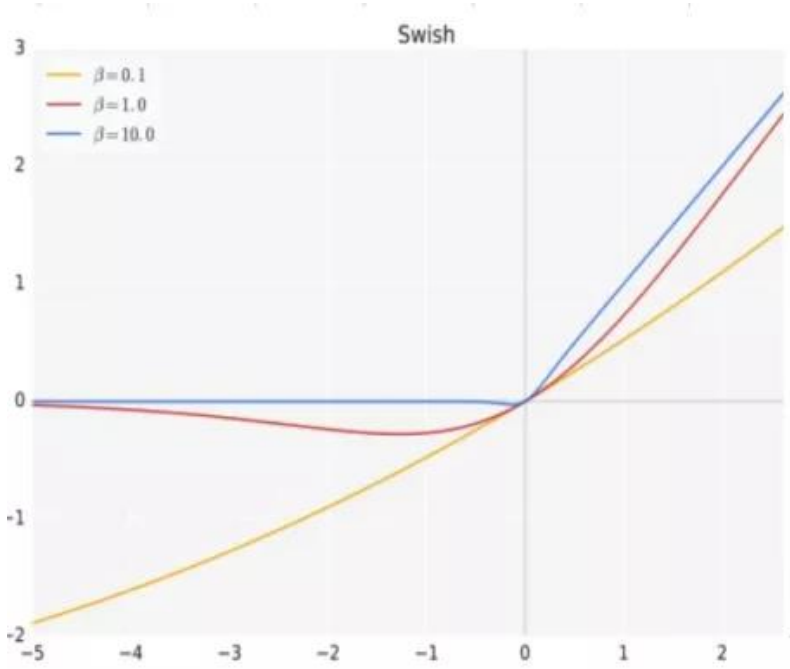


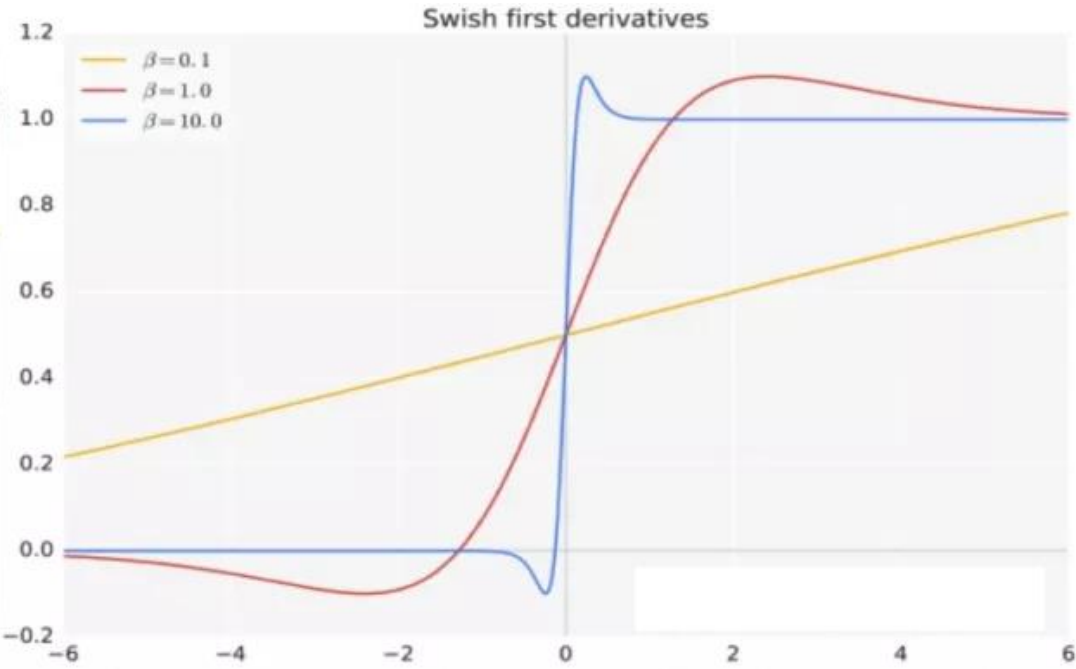
$$GELU(x)' = 0.5 + 0.5 * tanh(\ldots) + 0.5 * x * (1 - tanh(\ldots)^2) * (1 + 3 * 0.044715 * x^2)$$



Swish by Google 2017

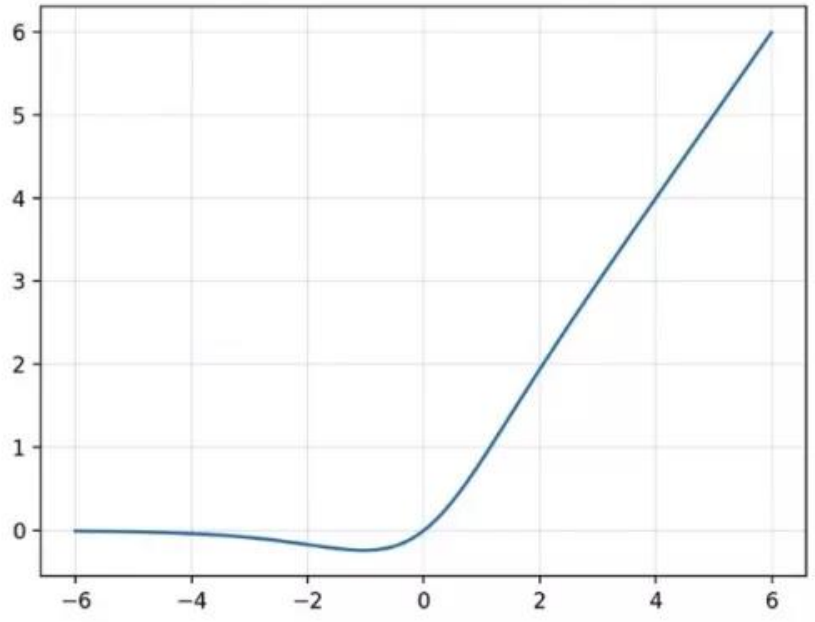
- f(x)=x*sigmoid(βx)
- When β=1.702, it becomes Gelu

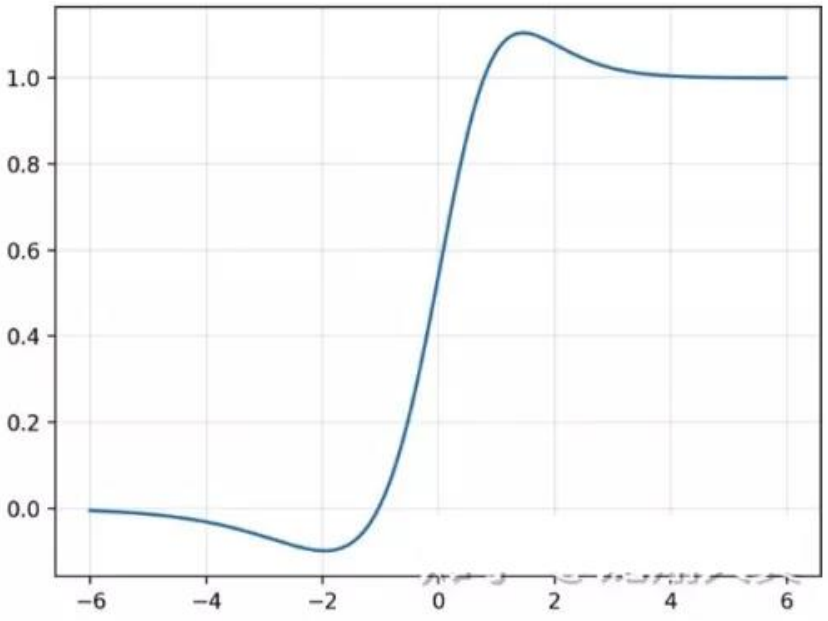




Mish

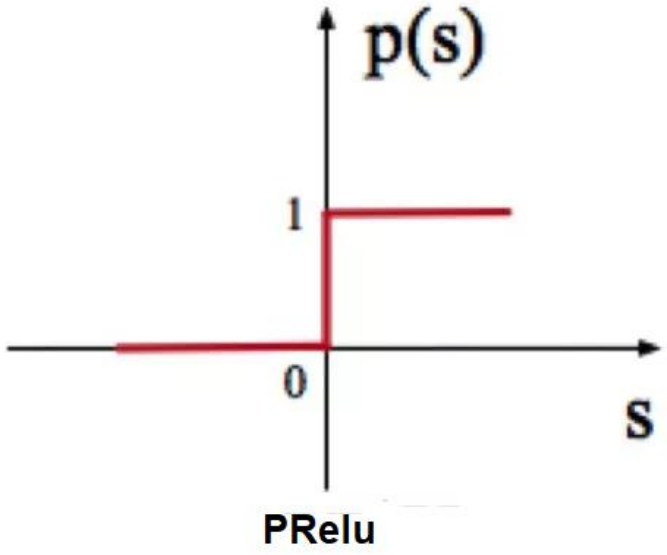
- **Mish=x * tanh(ln(1+e^x))**





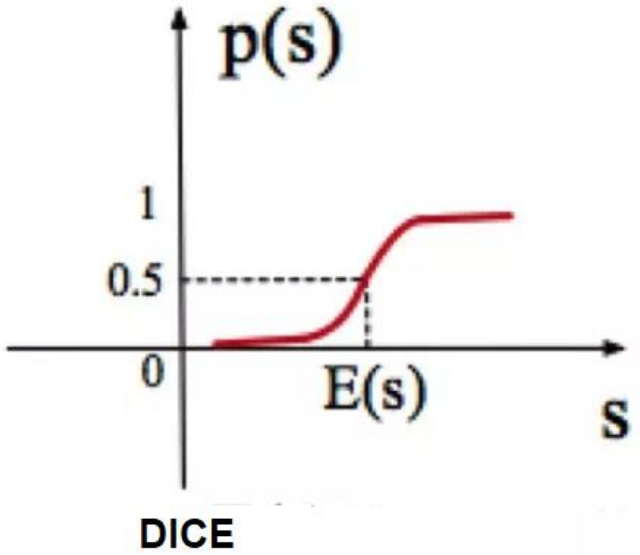
Data Adaptive Activation Function (Dice)

$$f(s) = \begin{cases} s & \text{if } s > 0 \\ \alpha s & \text{if } s \le 0. \end{cases} = p(s) \cdot s + (1 - p(s)) \cdot \alpha s,$$



PRelu

$$f(s) = p(s) \cdot s + (1 - p(s)) \cdot \alpha s, \quad p(s) = \frac{1}{1 + e^{-\frac{s - E[s]}{\sqrt{Var[s] + \epsilon}}}}$$



DICE

Solutions to gradients vanishing

- Careful init
- Batch normalization
- Residual connections

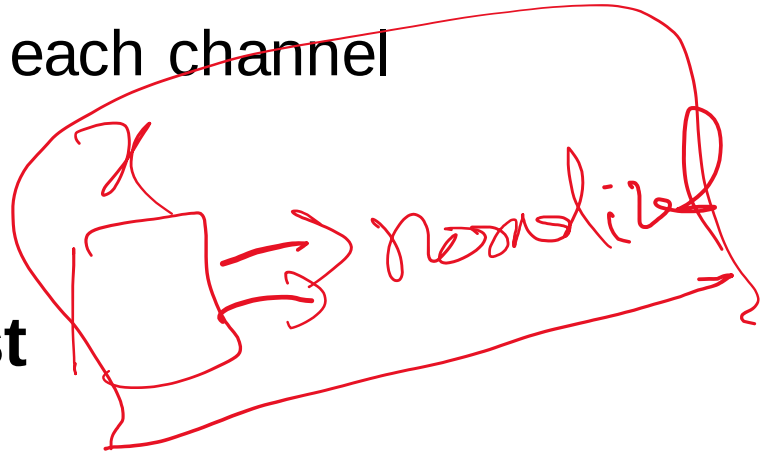
Careful initialization

- **Key idea: want variance to remain approx. constant**
  - Variance increases in backward pass => exploding gradient
  - Variance decreases in backward pass => vanishing gradient
- **“MSRA initialization”**
  - weights = Gaussian with 0 mean and variance = 2/(k*k*d)

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. K. He, X. Zhang, S. Ren, J. Sun

Batch normalization

- **Key idea: normalize so that each layer output has zero mean and unit variance**
  - Compute mean and variance for each channel
  - Aggregate over batch
  - Subtract mean, divide by std
- **Need to reconcile train and test**
  - No ”batches” during test
  - After training, compute means and variances on train set and store



Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. S. Ioffe, C. Szegedy. In *ICML,* 2015.

Residual connections

- In general, gradients tend to vanish
- Key idea: allow gradients to flow unimpeded

$$z_{i+1} = f_{i+1}(z_i, w_{i+1}) \qquad \frac{\partial z_{i+1}}{\partial z_i} = \frac{\partial f_{i+1}(z_i, w_{i+1})}{\partial z_i}$$

$$\frac{\partial z}{\partial z_i} = \frac{\partial z}{\partial z_{n-1}} \frac{\partial z_{n-1}}{\partial z_{n-2}} \cdots \frac{\partial z_{i+1}}{\partial z_i}$$

Residual connections

- In general, gradients tend to vanish
- Key idea: allow gradients to flow unimpeded

$$z_{i+1} = g_{i+1}(z_i, w_{i+1}) + z_i \qquad \frac{\partial z_{i+1}}{\partial z_i} = \frac{\partial g_{i+1}(z_i, w_{i+1})}{\partial z_i} + I$$

$$\frac{\partial z}{\partial z_i} = \frac{\partial z}{\partial z_{n-1}} \frac{\partial z_{n-1}}{\partial z_{n-2}} \cdots \frac{\partial z_{i+1}}{\partial z_i}$$

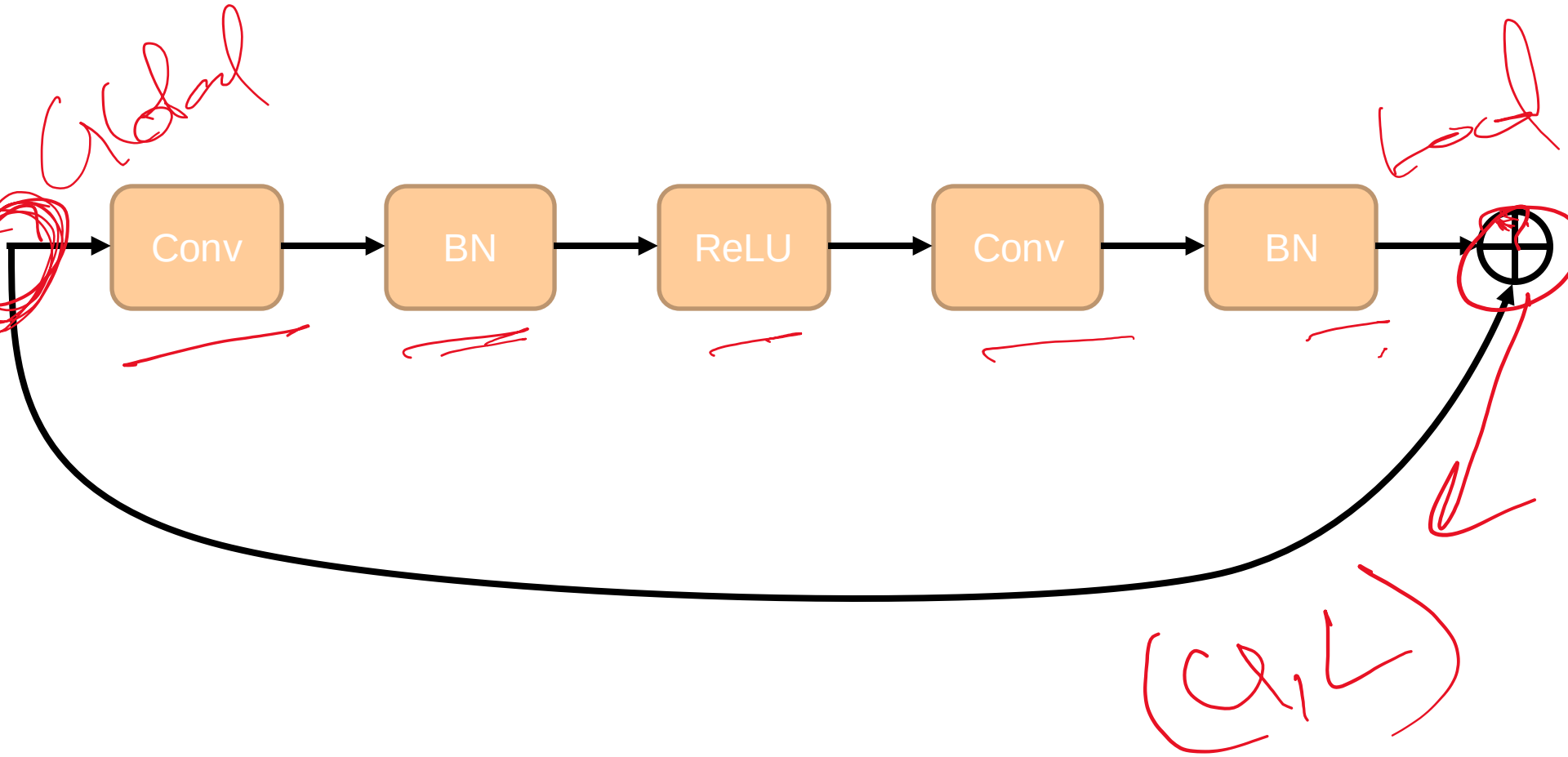
Residual connections

- **Assumes all z_i have the same size**
- **True within a stage**
- **Across stages?**
  - Doubling of feature channels
  - Subsampling
- **Increase channels by 1x1 convolution**
- **Decrease spatial resolution by subsampling**

$$z_{i+1} = g_{i+1}(z_i, w_{i+1}) + \text{subsample}(W z_i)$$

A residual block

- **Instead of single layers, have residual connections over block**



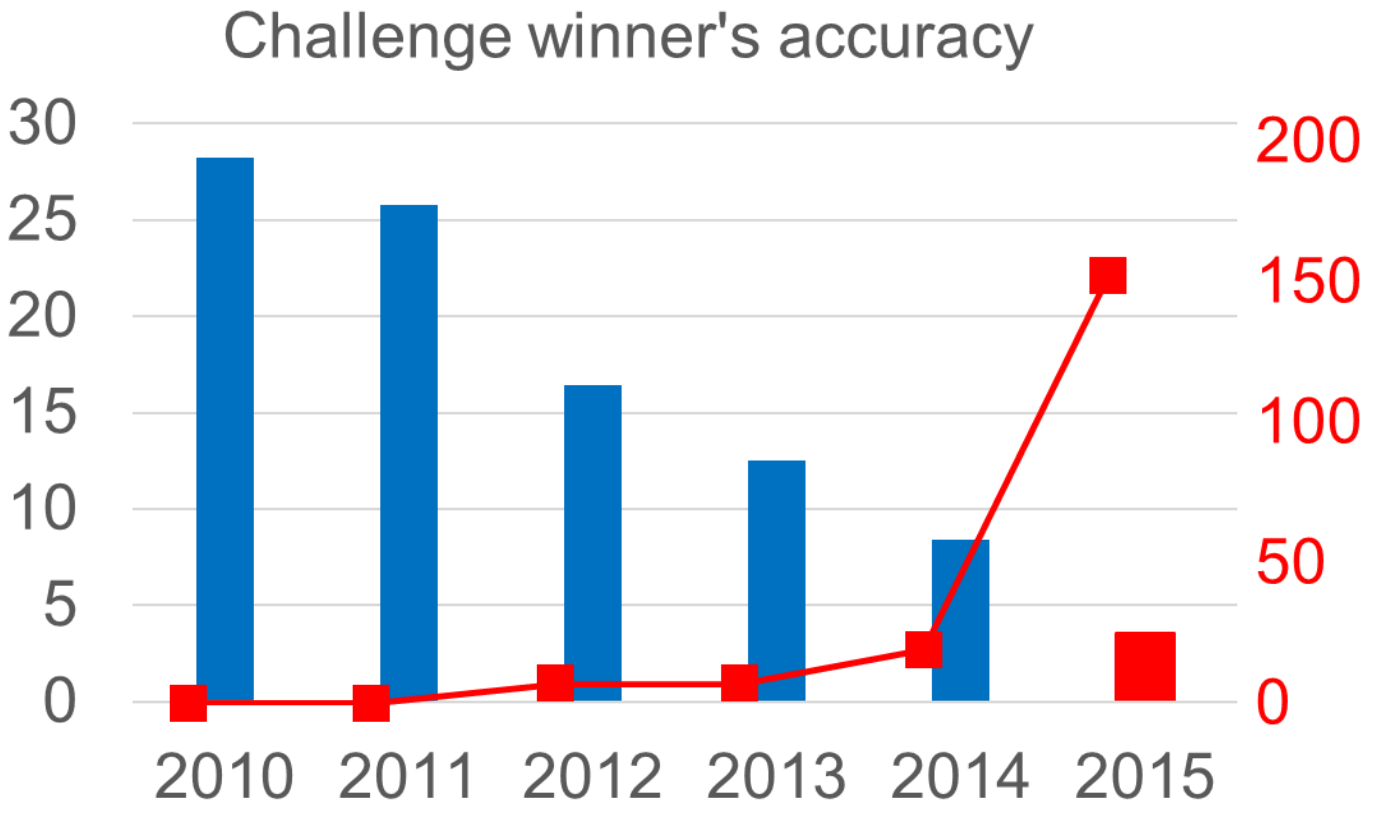
Bottleneck blocks

- **Problem: When channels increases, 3x3 convolutions introduce many parameters**
  - $3x3xc^2$
- **Key idea: use 1x1 to project to lower dimensionality, do convolution, then come back**
  - $cxd + 3x3xd^2 + dxc$

The ResNet pattern

- **Decrease resolution substantially in first layer**
  - Reduces memory consumption due to intermediate outputs
- **Divide into stages**
  - maintain resolution, channels in each stage
  - halve resolution, double channels between stages
- **Divide each stage into residual blocks**
- **At the end, compute average value of each channel to feed linear classifier**

Putting it all together - Residual networks



1x1 convolutions

1x1 convolutions

1	2	3	6	5	8
3	5	5	1	3	4
2	1	3	4	9	3
4	7	8	5	7	9
1	5	3	7	4	8
5	4	9	8	3	5

6X6

*

2

=

2	4	6			

Convolutions Over Channels

- **A convolutional operation is a linear application of a smaller filter to a larger input that results in an output feature map.**
  - Every time, a filter applied to an input image or input feature map always results in a single number.
  - The systematic left-to-right and top-to-bottom application of the filter to the input results in a two-dimensional feature map. One filter creates one corresponding feature map.
- **A filter must have the same depth or number of channels as the input, yet, regardless of the depth of the input and the filter, the resulting output is a single number and one filter creates a feature map with a single channel.**
- **Let's make this concrete with some examples:**
  - If the input has one channel such as a grayscale image, then a 3×3 filter will be applied in 3x3x1 blocks.
  - If the input image has three channels for red, green, and blue, then a 3×3 filter will be applied in 3x3x3 blocks.
  - If the input is a block of feature maps from another convolutional or pooling layer and has the depth of 64, then the 3×3 filter will be applied in 3x3x64 blocks to create the single values to make up the single output feature map.
- **The depth of the output of one convolutional layer is only defined by the number of parallel filters applied to the input.**

Problem of Too Many Feature Maps

- The depth of the input or number of filters used in convolutional layers often increases with the depth of the network, resulting in an increase in the number of resulting feature maps.
- Further, some network architectures, may also concatenate the output feature maps from multiple convolutional layers, which may also dramatically increase the depth of the input to subsequent convolutional layers.
- A large number of feature maps in a convolutional neural network can result in considerably more parameters (weights) and, in turn, computation to perform the convolutional operations (large space and time complexity).
- Pooling layers are designed to downscale feature maps and systematically halve the width and height of feature maps in the network. Nevertheless, pooling layers do not change the number of filters in the model, the depth, or number of channels.
- Deep convolutional neural networks require a corresponding pooling type of layer that can down sample or reduce the depth or number of feature maps.

Down sample Feature Maps With 1x1 convolutions

- **A 1×1 filter will only have a single parameter or weight for each channel in the input**
  - This structure allows the 1×1 filter to act like a single neuron with an input from the same position across each of the feature maps in the input.
  - This single neuron can then be applied systematically with a stride of one, left-to-right and top-to-bottom without any need for padding, resulting in a feature map with the same width and height as the input.
- **The 1×1 filter is a linear weighting or projection of the input. Further, a nonlinearity is used as with other convolutional layers, allowing the projection to perform non-trivial computation on the input feature maps.**
- **This simple 1×1 filter provides a way to usefully summarize the input feature maps.**
- **The use of multiple 1×1 filters, allows the tuning of the number of summaries of the input feature maps to create, effectively allowing the depth of the feature maps to be increased or decreased as needed.**
- **A convolutional layer with a 1×1 filter can, therefore, be used at any point in a convolutional neural network to control the number of feature maps.**

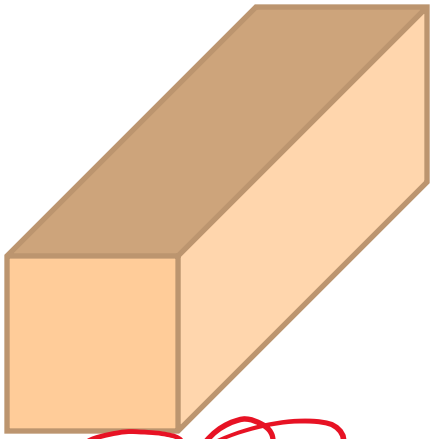
1x1 convolutions

1	2	3	6	5	8
3	5	5	1	3	4
2	1	3	4	9	3
4	7	8	5	7	9
1	5	3	7	4	8
5	4	9	8	3	5

6X6

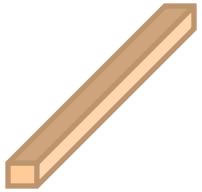
*

2



6X6X32

*

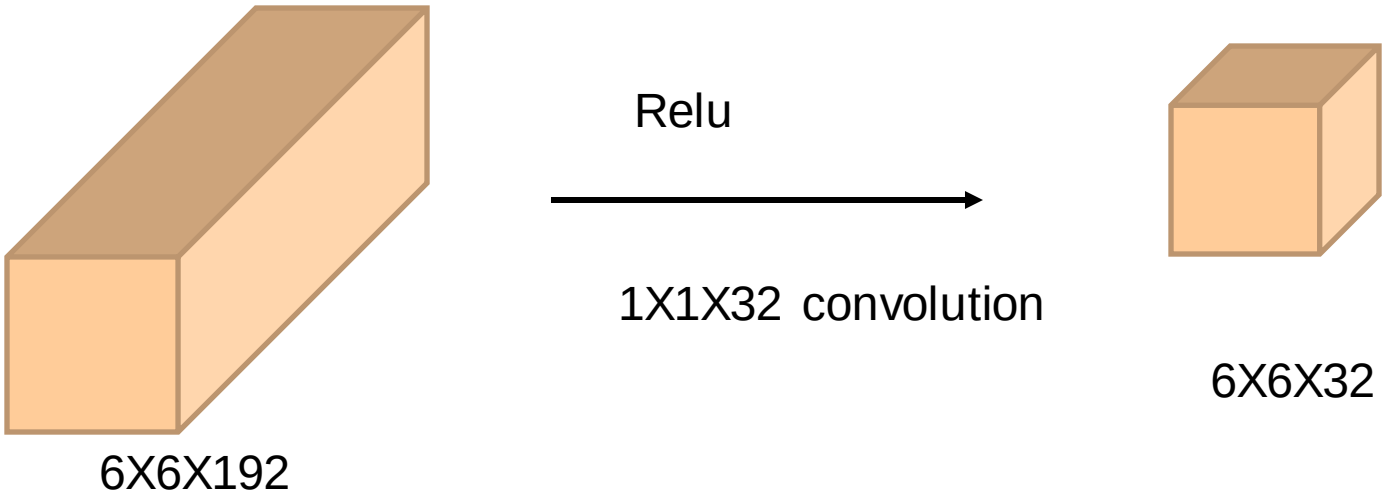


1X1X32

=

6X6X # filters

1x1 convolutions



Examples of How to Use 1×1 Convolutions

- Consider that we have a convolutional neural network that expected color images input with the square shape of 256x256x3 pixels.
- These images then pass through a first hidden layer with 512 filters, each with the size of 3×3 with the same padding, followed by a ReLU activation function.
- The example below demonstrates this simple model.

```
# example of simple cnn model
from keras.models import Sequential
from keras.layers import Conv2D
# create model
model = Sequential()
model.add(Conv2D(512, (3,3), padding='same', activation='relu', input_shape=(256, 2
# summarize model
model.summary()
```

Examples of How to Use 1×1 Convolutions

- Running the example creates the model and summarizes the model architecture.
- There are no surprises; the output of the first hidden layer is a block of feature maps with the three-dimensional shape of 256x256x512.

```
1  _________________________________________________________________
2  Layer (type)                 Output Shape              Param #
3  =================================================================
4  conv2d_1 (Conv2D)            (None, 256, 256, 512)     14336
5  =================================================================
6  Total params: 14,336
7  Trainable params: 14,336
8  Non-trainable params: 0
9  _________________________________________________________________
```

Example of Projecting Feature Maps

- A 1×1 filter can be used to create a projection of the feature maps.
- The number of feature maps created will be the same number and the effect may be a refinement of the features already extracted. This is often called channel-wise pooling, as opposed to traditional feature-wise pooling on each channel. It can be implemented as follows

```
model.add(Conv2D(512, (1,1), activation='relu'))
```

- We can see that we use the same number of features and still follow the application of the filter with a rectified linear activation function.
- The complete example is listed below.

```
# example of a 1x1 filter for projection
from keras.models import Sequential
from keras.layers import Conv2D
# create model
model = Sequential()
model.add(Conv2D(512, (3,3), padding='same', activation='relu', input_shape=(256, 2
model.add(Conv2D(512, (1,1), activation='relu'))
# summarize model
model.summary()
```

Example of Projecting Feature Maps

- Running the example creates the model and summarizes the architecture.
- We can see that no change is made to the width or height of the feature maps, and by design, the number of feature maps is kept constant with a simple projection operation applied.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 256, 256, 512)     14336
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 256, 256, 512)     262656
=================================================================
Total params: 276,992
Trainable params: 276,992
Non-trainable params: 0
_________________________________________________________________
```

Example of Decreasing Feature Maps

- The 1×1 filter can be used to decrease the number of feature maps.
- This is the most common application of this type of filter and in this way, the layer is often called a feature map pooling layer.
- In this example, we can decrease the depth (or channels) from 512 to 64. This might be useful if the subsequent layer we were going to add to our model would be another convolutional layer with 7×7 filters. These filters would only be applied at a depth of 64 rather than 512.

```
model.add(Conv2D(64, (1,1), activation='relu'))
```

- The composition of the 64 feature maps is not the same as the original 512, but contains a useful summary of dimensionality reduction that captures the salient features, such that the 7×7 operation may have a similar effect on the 64 feature maps as it might have on the original 512.
- Further, a 7×7 convolutional layer with 64 filters itself applied to the 512 feature maps output by the first hidden layer would result in approximately one million parameters (weights). If the 1×1 filter is used to reduce the number of feature maps to 64 first, then the number of parameters required for the 7×7 layer is only approximately 200,000, an enormous difference.
- The complete example of using a 1×1 filter for dimensionality reduction is listed below.

```
1 # example of a 1x1 filter for dimensionality reduction
2 from keras.models import Sequential
3 from keras.layers import Conv2D
4 # create model
5 model = Sequential()
6 model.add(Conv2D(512, (3,3), padding='same', activation='relu', input_shape=(256, 2
7 model.add(Conv2D(64, (1,1), activation='relu'))
8 # summarize model
9 model.summary()
```

Example of Decreasing Feature Maps

- Running the example creates the model and summarizes its structure.
- We can see that the width and height of the feature maps are unchanged, yet the number of feature maps was reduced from 512 to 64.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 256, 256, 512)     14336
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 256, 256, 64)      32832
=================================================================
Total params: 47,168
Trainable params: 47,168
Non-trainable params: 0
_________________________________________________________________
```

Example of Increasing Feature Maps

- The 1×1 filter can be used to increase the number of feature maps.
- This is a common operation used after a pooling layer prior to applying another convolutional layer.
- The projection effect of the filter can be applied as many times as needed to the input, allowing the number of feature maps to be scaled up and yet have a composition that captures the salient features of the original.
- We can increase the number of feature maps from 512 input from the first hidden layer to double the size at 1,024 feature maps.

```
model.add(Conv2D(1024, (1,1), activation='relu'))
```

- The complete example is listed below.

```
# example of a 1x1 filter to increase dimensionality
from keras.models import Sequential
from keras.layers import Conv2D
# create model
model = Sequential()
model.add(Conv2D(512, (3,3), padding='same', activation='relu', input_shape=(256, 2
model.add(Conv2D(1024, (1,1), activation='relu'))
# summarize model
model.summary()
```

Example of Increasing Feature Maps

- Running the example creates the model and summarizes its structure.
- We can see that the width and height of the feature maps are unchanged and that the number of feature maps was increased from 512 to double the size at 1,024.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 256, 256, 512)     14336
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 256, 256, 1024)    525312
=================================================================
Total params: 539,648
Trainable params: 539,648
Non-trainable params: 0
_________________________________________________________________
```

Summary: 1x1 convolutions

- The 1×1 filter can be used to create a linear projection of a stack of feature maps.
- The projection created by a 1×1 can act like channel-wise pooling and be used for dimensionality reduction.
- The projection created by a 1×1 can also be used directly or be used to increase the number of feature maps in a model.

Batch Normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want zero-mean unit-variance activations? just make them so."

consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

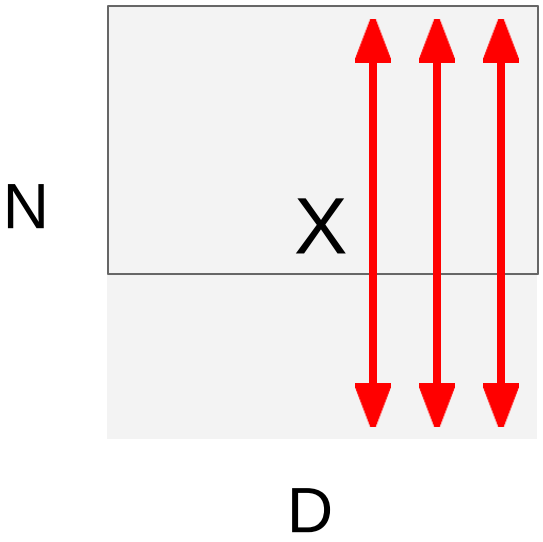
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want zero-mean unit-variance activations? just make them so."



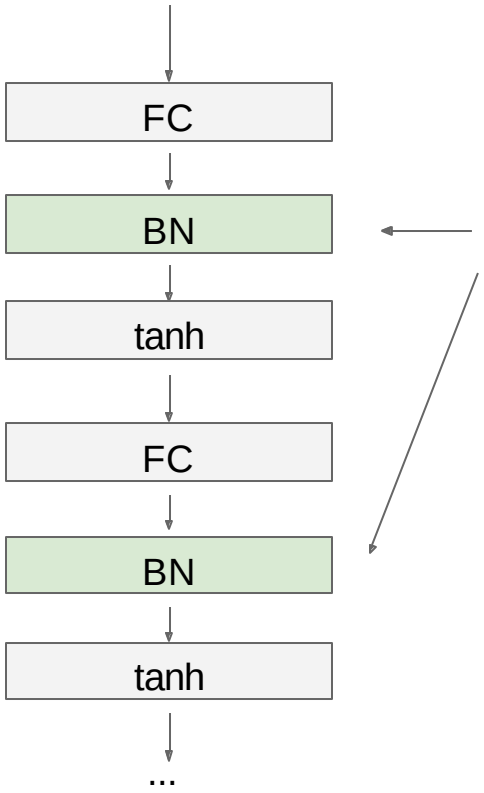
1. compute the empirical mean and variance independently for each dimension.

2. Normalize

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

Batch Normalization

[Ioffe and Szegedy, 2015]

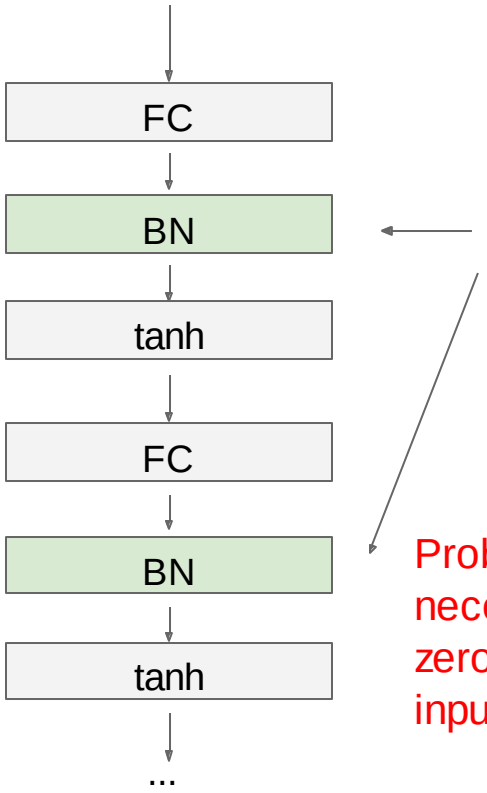


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

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

Batch Normalization

[Ioffe and Szegedy, 2015]



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

Problem: do we necessarily want a zero-mean unit-variance input?

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

Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

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

And then allow the network to squash the range if it wants to:

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

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

to recover the identity mapping.

Batch Normalization: Test Time

[Ioffe and Szegedy, 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

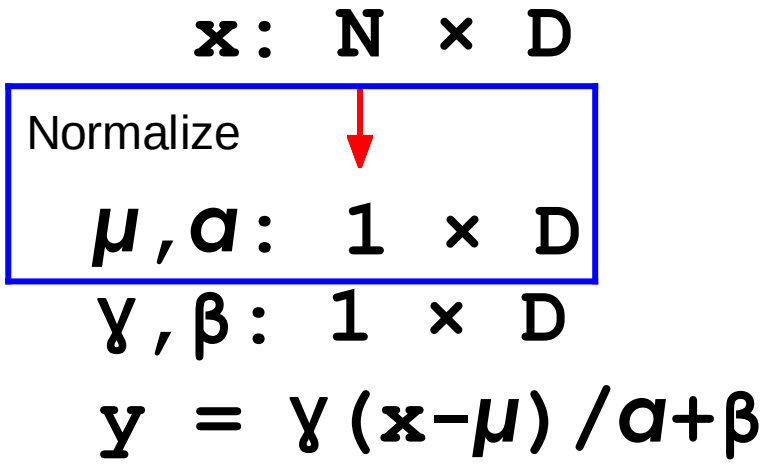
Note: at test time BatchNorm layer functions differently:

The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

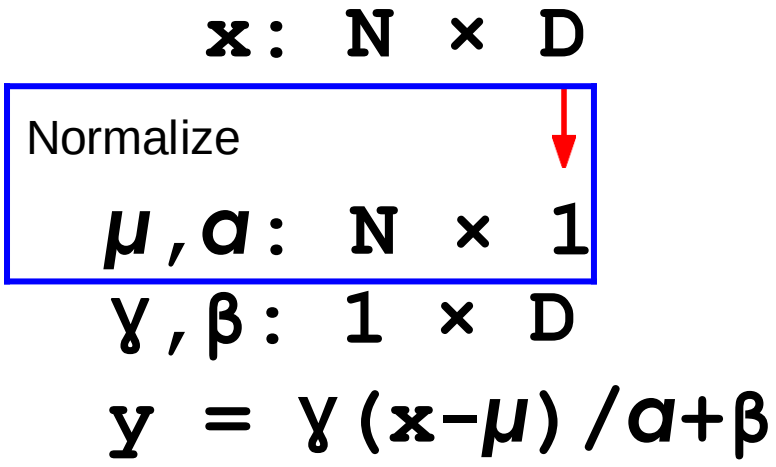
(e.g. can be estimated during training with running averages)

Layer Normalization

Batch Normalization for fully-connected networks



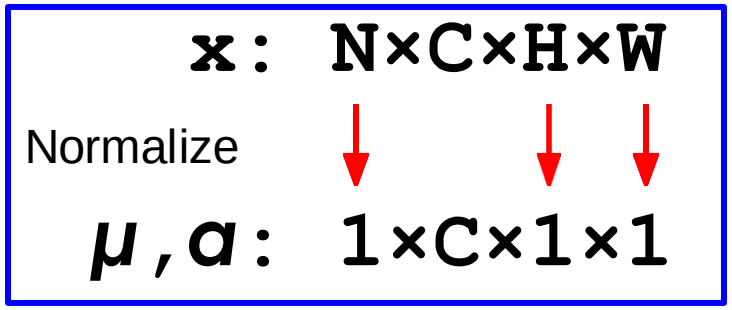
Layer Normalization for fully-connected networks
Same behavior at train and test!
Can be used in recurrent networks



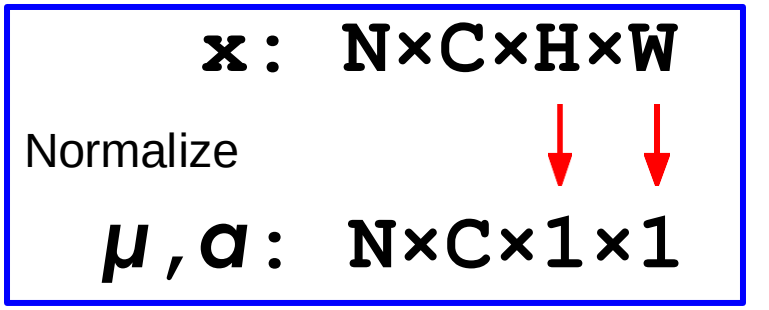
Ba, Kiros, and Hinton, "Layer Normalization", arXiv 2016

Instance Normalization

Batch Normalization for convolutional networks

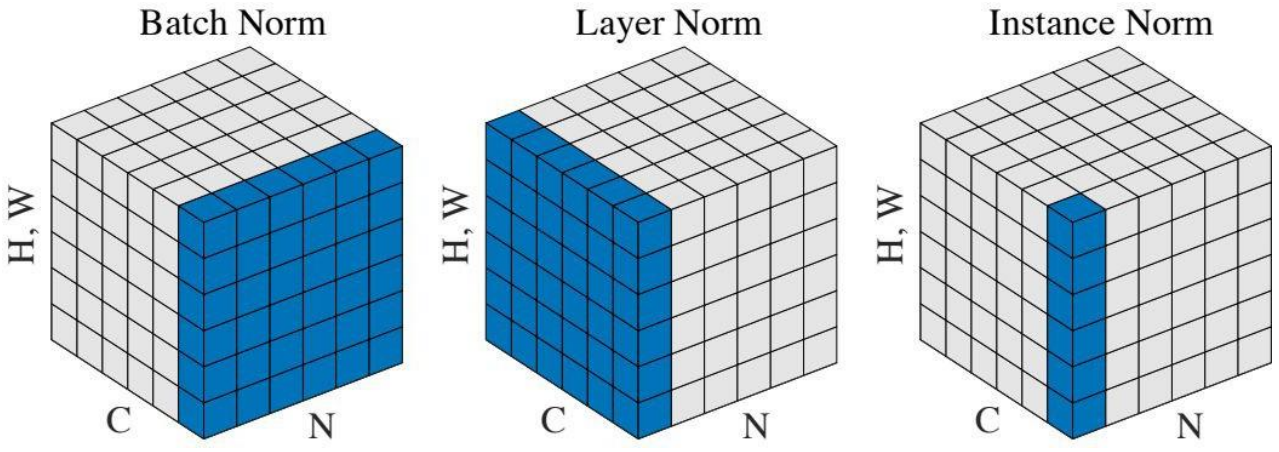


Instance Normalization for convolutional networks
Same behavior at train / test!



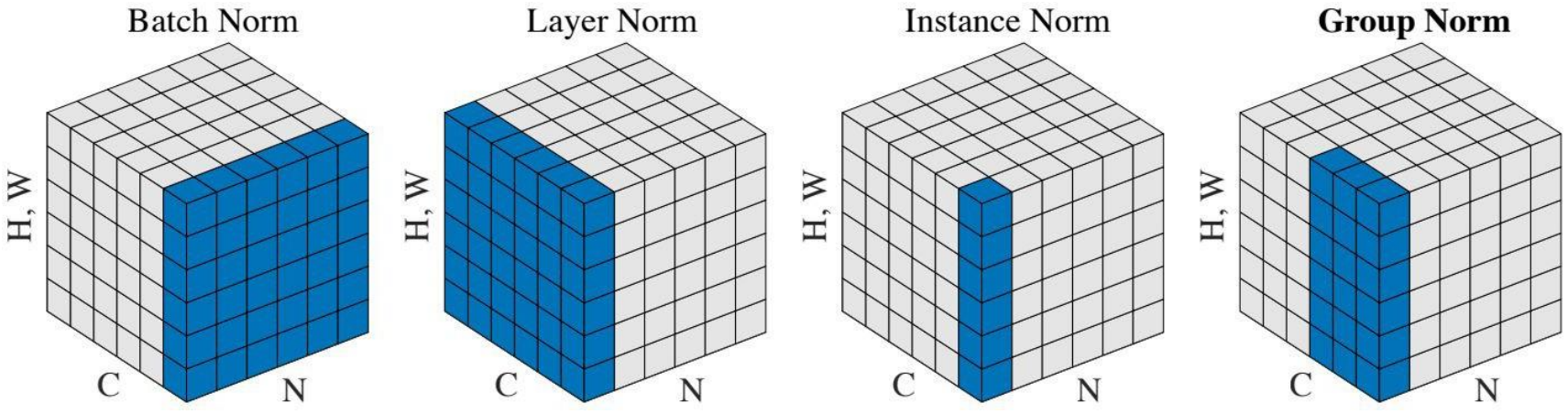
Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017

Comparison of Normalization Layers



Wu and He, “Group Normalization”, arXiv 2018

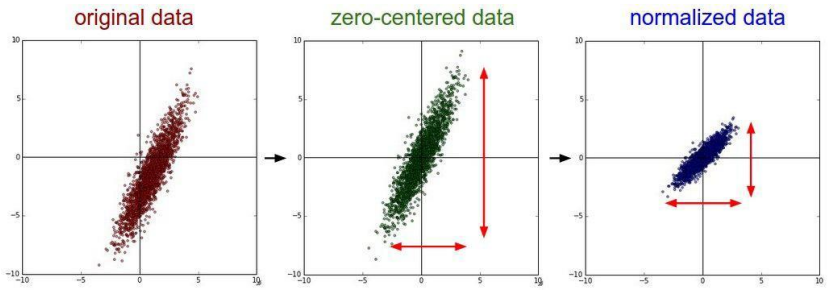
Group Normalization



Wu and He, “Group Normalization”, arXiv 2018 (Appeared 3/22/2018)

Decorrelated Batch Normalization

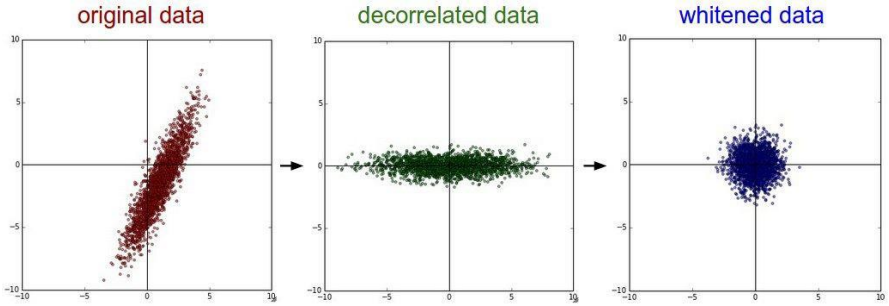
Batch Normalization



$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

BatchNorm normalizes the data, but cannot correct for correlations among the input features

Decorrelated Batch Normalization



$$\hat{x}_i = \Sigma^{-\frac{1}{2}}(x_i - \mu)$$

DBN **whitens** the data using the full covariance matrix of the minibatch; this corrects for correlations

Huang et al, “Decorrelated Batch Normalization”, arXiv 2018 (Appeared 4/23/2018)

Optimization: Adam

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9,
beta2 = 0.999, and learning_rate = 1e-3 or 5e-4
is a great starting point for many models!

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Regularization: Add term to loss

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use:

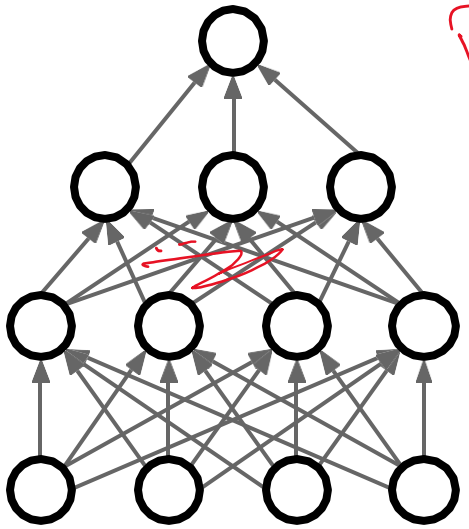
L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)

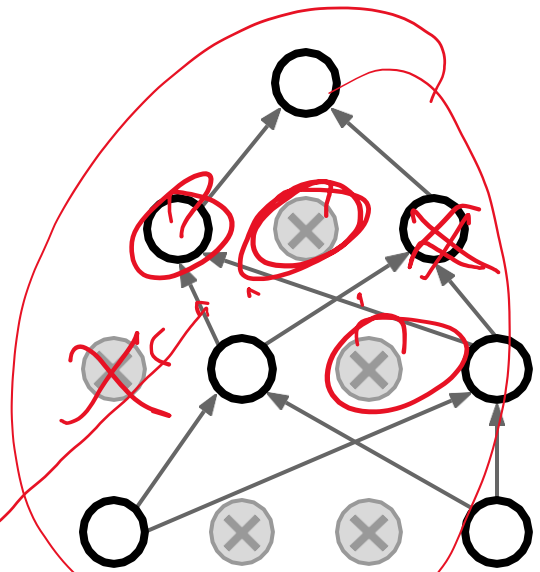
L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Regularization: Dropout

In each forward pass, randomly set some neurons to zero Probability of dropping is a hyperparameter; 0.5 is common

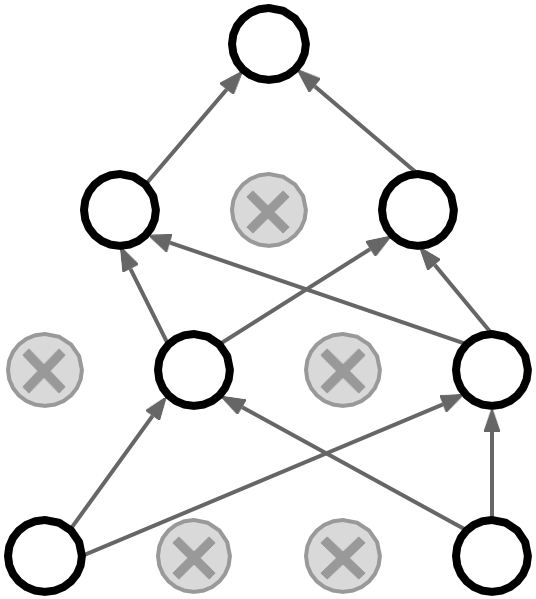




Srivastava et al, "Dropout: A simple way to prevent neural networks from overfitting", JMLR 2014

Regularization: Dropout

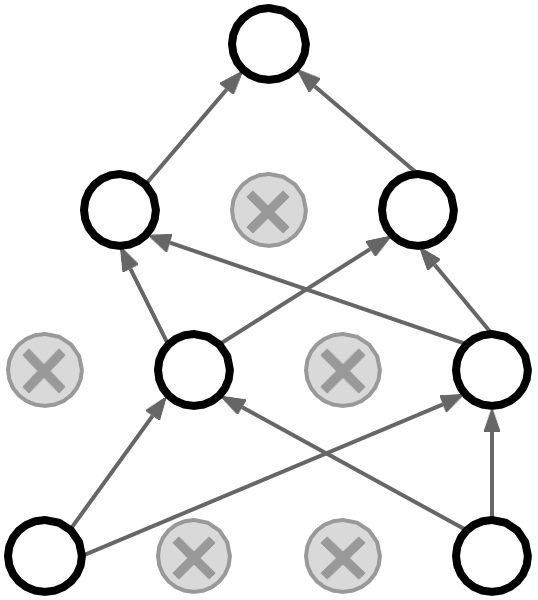
How can this possibly be a good idea?



Forces the network to have a redundant representation;
Prevents co-adaptation of features

Regularization: Dropout

How can this possibly be a good idea?



Another interpretation:

Dropout is training a large **ensemble** of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has
$2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ atoms in the universe...

Dropout as a form of model averaging

- **We sample from 2^H models. So only a few of the models ever get trained, and they only get one training example.**
  - This is as extreme as bagging can get.
- **The sharing of the weights means that every model is very strongly regularized.**
  - It's a much better regularizer than L2 or L1 penalties that pull the weights towards zero.

Dropout: Test time

Dropout makes our output random!

Output (label) Input (image)

$$y = f_W(x, z)$$

Random mask

Want to "average out" the randomness at test-time

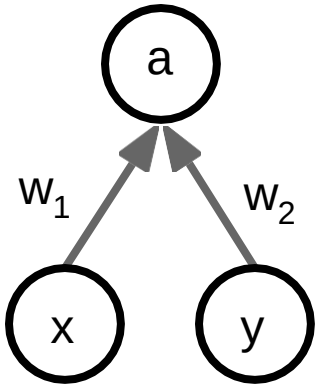
$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$

But this integral seems hard …

Dropout: Test time

Want to approximate the integral

$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$



Consider a single neuron.

At test time we have: $E\big[a\big] = w_1 x + w_2 y$

During training we have:

$$\begin{aligned} E\big[a\big] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

At test time, **multiply** by dropout probability

At test time all neurons are active always

=> We must scale the activations so that for each neuron:

output at test time = expected output at training time

How well does dropout work?

- **The record-breaking object recognition net developed by Alex Krizhevsky uses dropout and it helps a lot.**
- **If your deep neural net is significantly overfitting, dropout will usually reduce the number of errors by a lot.**
  - Any net that uses “early stopping” can do better by using dropout (at the cost of taking quite a lot longer to train).
- **If your deep neural net is not overfitting, you should be using a bigger one!**

Another way to think about dropout

- **If a hidden unit knows which other hidden units are present, it can co-adapt to them on the training data.**
  - But complex co-adaptations are likely to go wrong on new test data.
  - Big, complex conspiracies are not robust.
- **If a hidden unit has to work well with combinatorially many sets of co-workers, it is more likely to do something that is individually useful.**
  - But it will also tend to do something that is marginally useful given what its co-workers achieve.

Regularization: A common pattern

Training: Add some kind of randomness

$$y = f_W(x, z)$$

Testing: Average out randomness (sometimes approximate)

$$y = f(x) = E_z\big[f(x, z)\big] = \int p(z) f(x, z) dz$$

Regularization: A common pattern

Training: Add some kind of randomness

$$y = f_W(x, z)$$

Testing: Average out randomness (sometimes approximate)

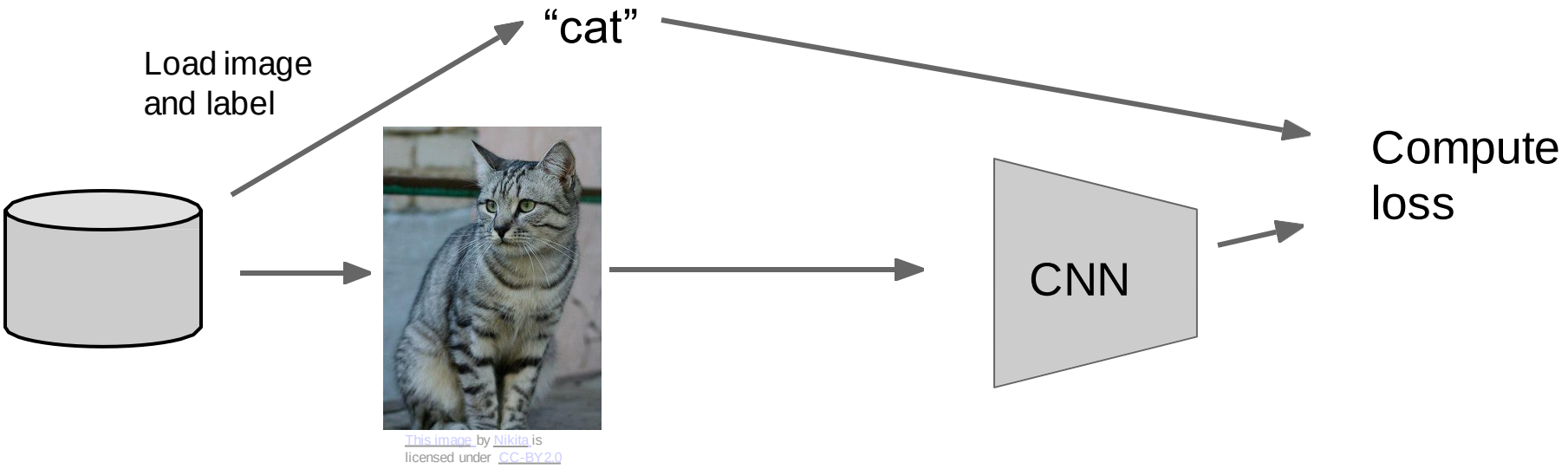
$$y = f(x) = E_z\big[f(x, z)\big] = \int p(z) f(x, z) dz$$

Example: Batch Normalization

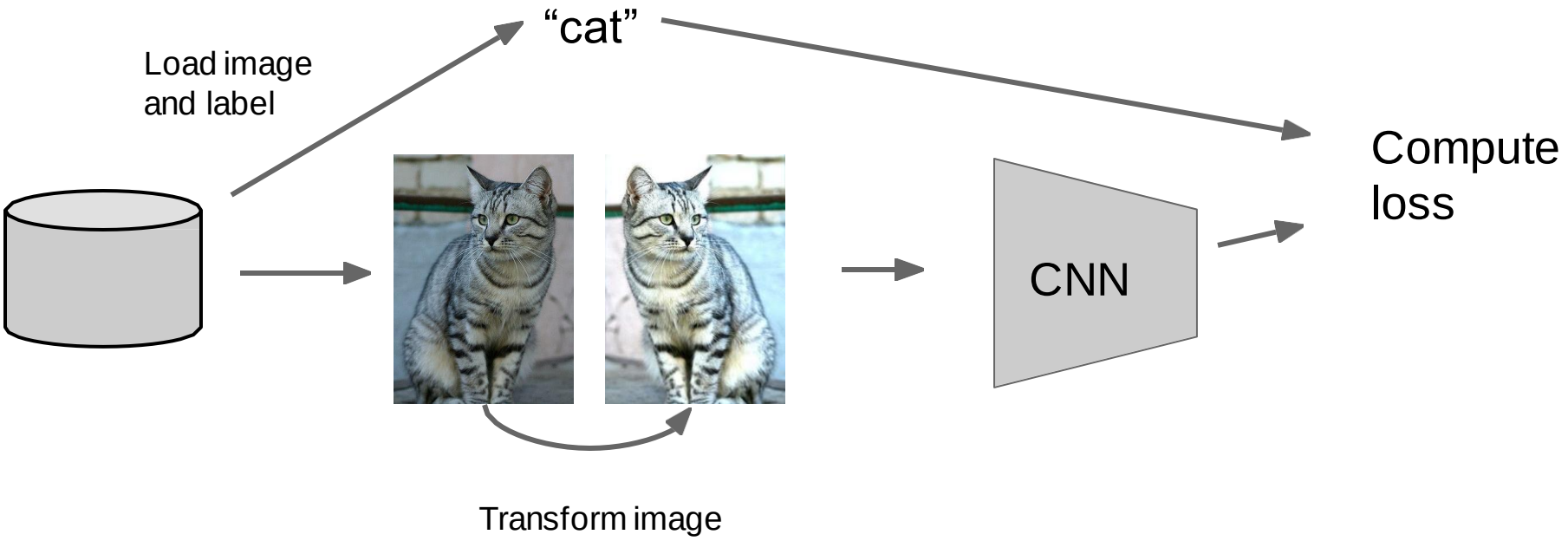
Training: Normalize using stats from random minibatches

Testing: Use fixed stats to normalize

Regularization: Data Augmentation

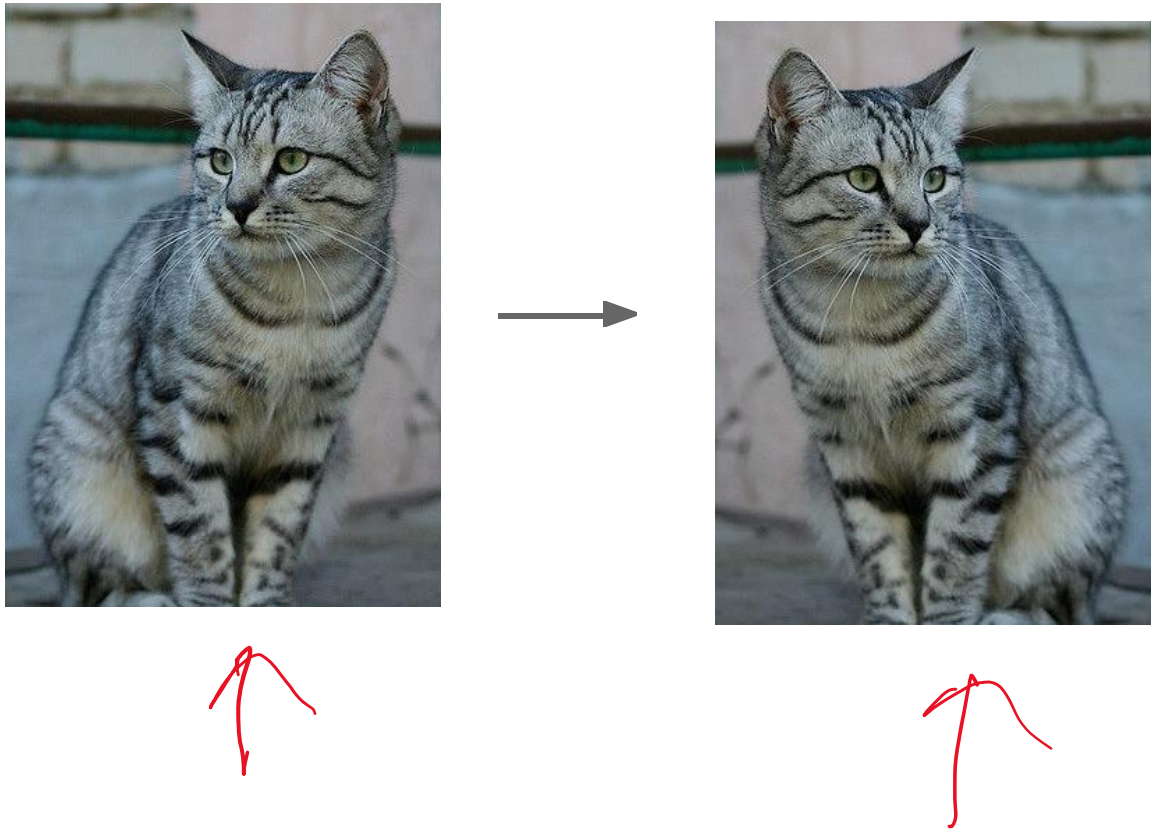


Regularization: Data Augmentation



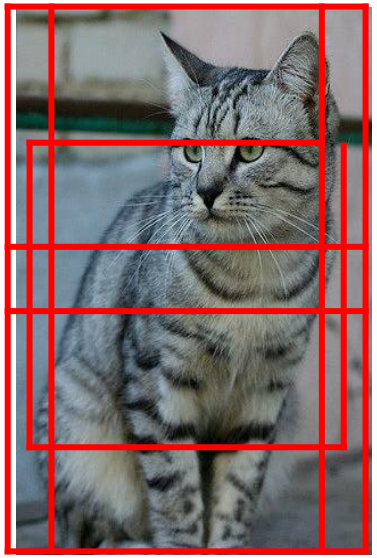
Data Augmentation

Horizontal Flips



Data Augmentation

Random crops and scales



Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch

Testing: average a fixed set of crops

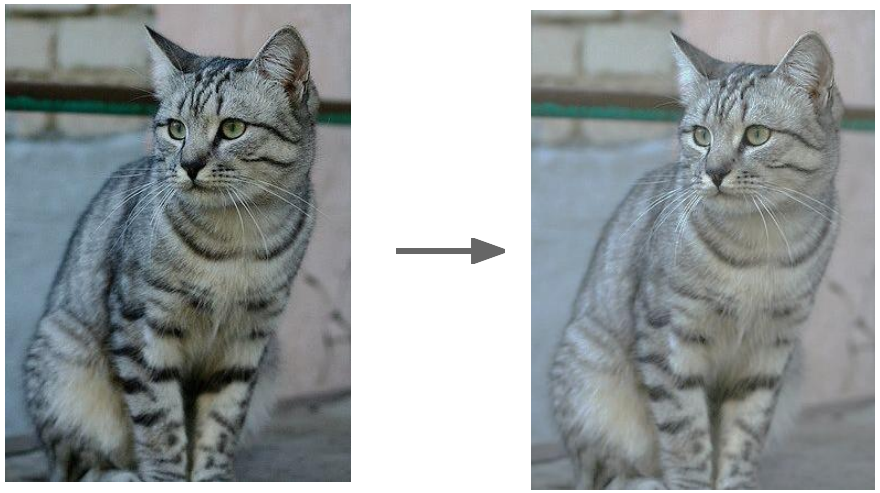
ResNet:

1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

Data Augmentation

Color Jitter

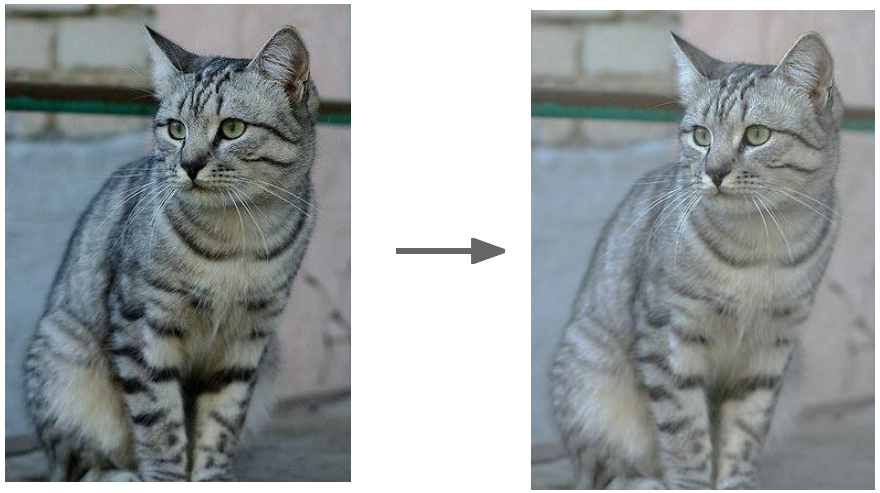
Simple: Randomize contrast and brightness



Data Augmentation

Color Jitter

Simple: Randomize contrast and brightness



More Complex:

- **Apply PCA to all [R, G, B] pixels in training set**
- **Sample a "color offset" along principal component directions**
- **Add offset to all pixels of a training image**

(As seen in *[Krizhevsky et al. 2012],* ResNet, etc)

Data Augmentation

Get creative for your problem!

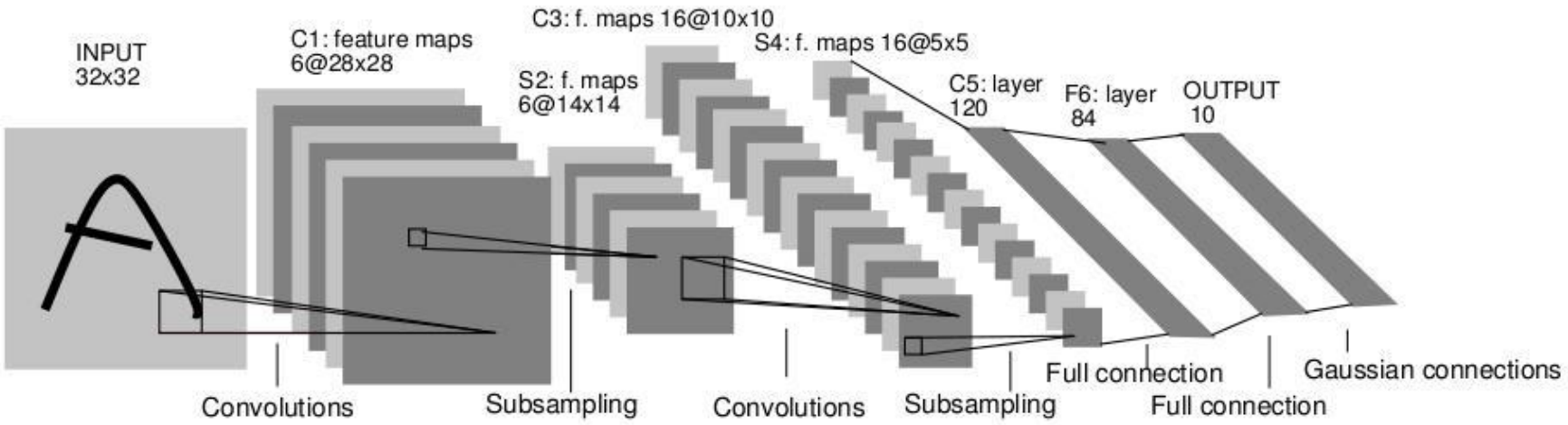
Random mix/combinations of :

- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)

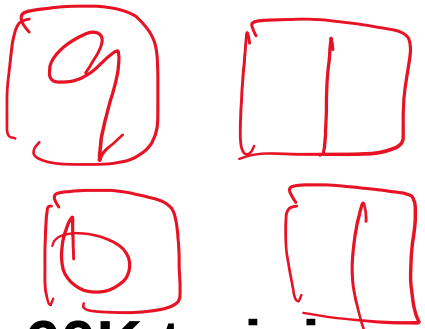
Convolutional Neural Network Architectures

- Convolutional Neural Network Architecture: from LeNet to ResNet

LeNet-5



- **Average pooling**
- **Sigmoid or tanh nonlinearity**
- **Fully connected layers at the end**
- **Trained on MNIST digit dataset with 60K training examples**



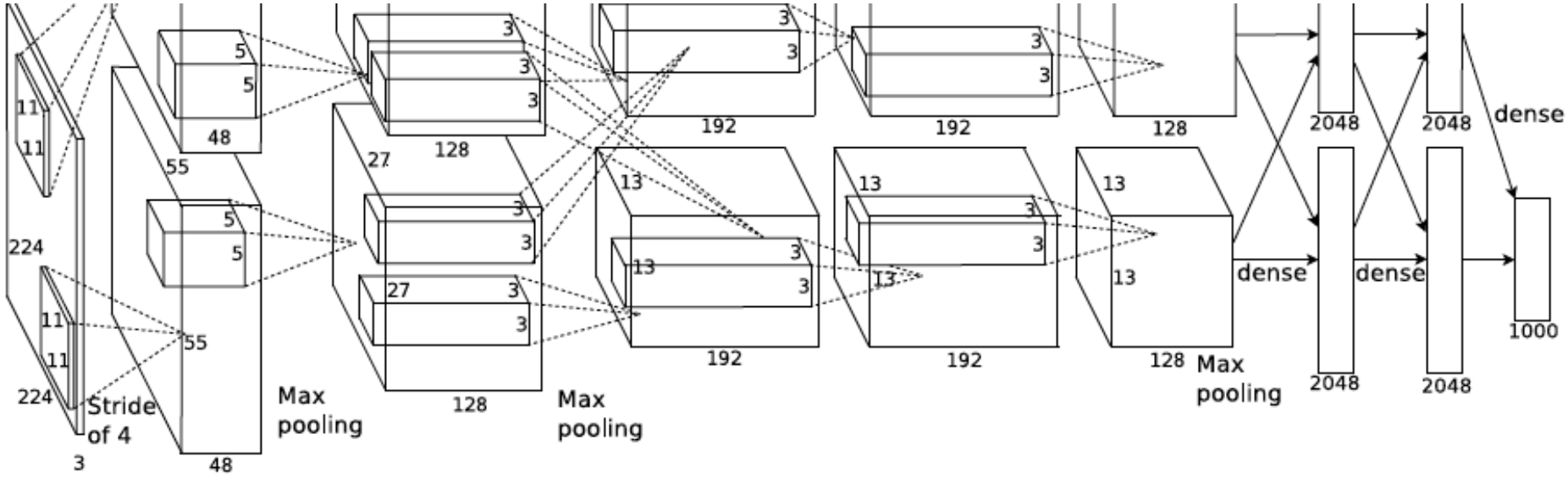
Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE 86(11): 2278–2324, 1998.

Fast forward to the arrival of big visual data…

- ~14 million labeled images, 20k classes
- Images gathered from Internet
- Human labels via Amazon MTurk
- ImageNet Large-Scale Visual Recognition Challenge (ILSVRC): 1.2 million training images, 1000 classes

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

AlexNet: ILSVRC 2012 winner



- Similar framework to LeNet but:
  - Max pooling, ReLU nonlinearity
  - More data and bigger model (7 hidden layers, 650K units, 60M params)
  - GPU implementation (50x speedup over CPU)
    - Trained on two GPUs for a week
  - Dropout regularization

A. Krizhevsky, I. Sutskever, and G. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, NIPS 2012

Clarifai: ILSVRC 2013 winner

- **Refinement of AlexNet**

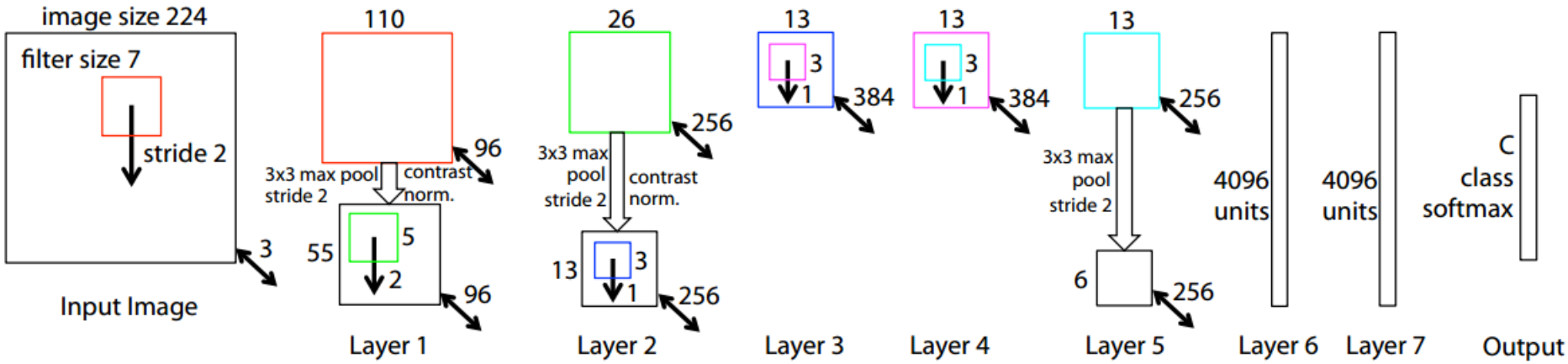


Figure 3. Architecture of our 8 layer convnet model. A 224 by 224 crop of an image (with 3 color planes) is presented as the input. This is convolved with 96 different 1st layer filters (red), each of size 7 by 7, using a stride of 2 in both x and y. The resulting feature maps are then: (i) passed through 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.

M. Zeiler and R. Fergus, Visualizing and Understanding Convolutional Networks, ECCV 2014 (Best Paper Award winner)

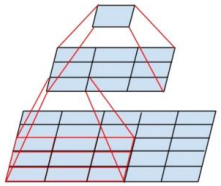
VGGNet: ILSVRC 2014 2nd place

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

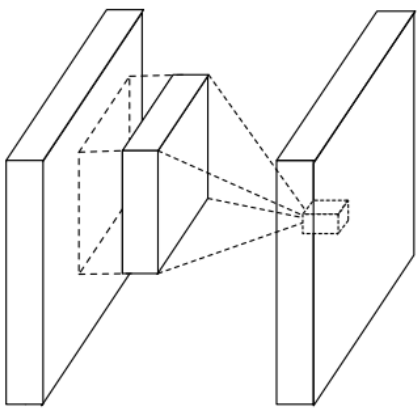
- **Sequence of deeper networks trained progressively**
- **Large receptive fields replaced by successive layers of 3x3 convolutions (with ReLU in between)**



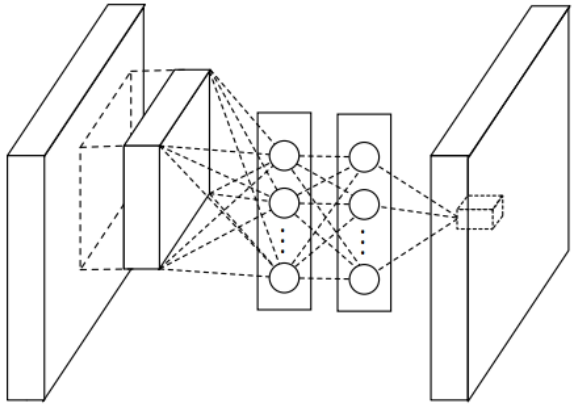
- **One 7x7 conv layer with C feature maps needs $49C^2$ weights, three 3x3 conv layers need only $27C^2$ weights**
- **Experimented with 1x1 convolutions**

K. Simonyan and A. Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition, ICLR 2015

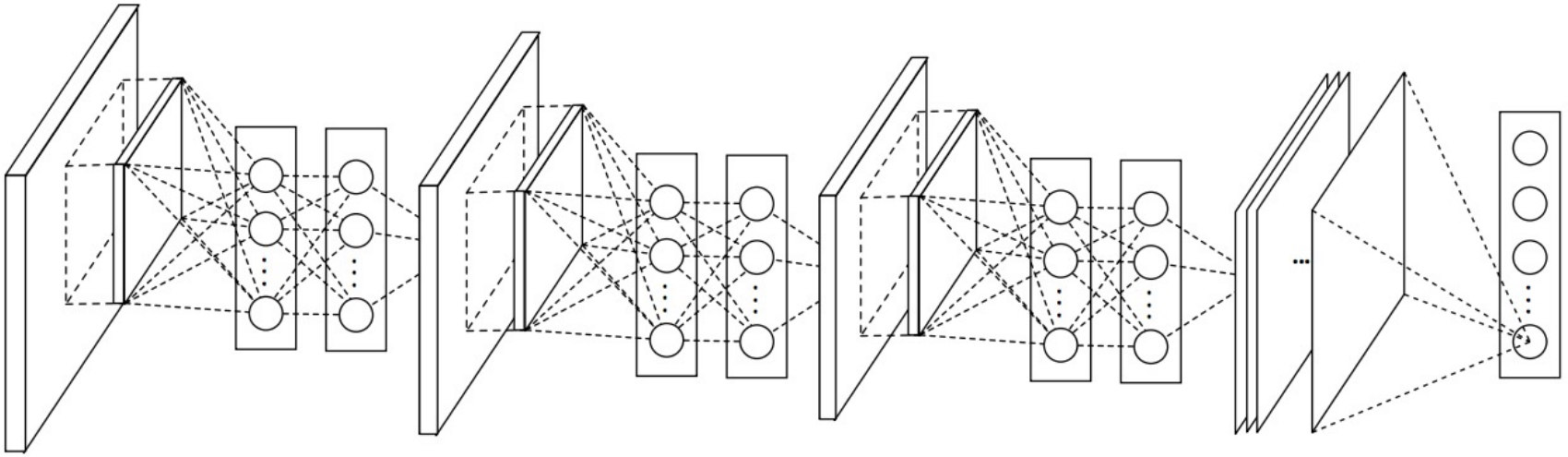
Network in network



(a) Linear convolution layer



(b) Mlpconv layer



M. Lin, Q. Chen, and S. Yan, Network in network, ICLR 2014

GoogLeNet: ILSVRC 2014 winner

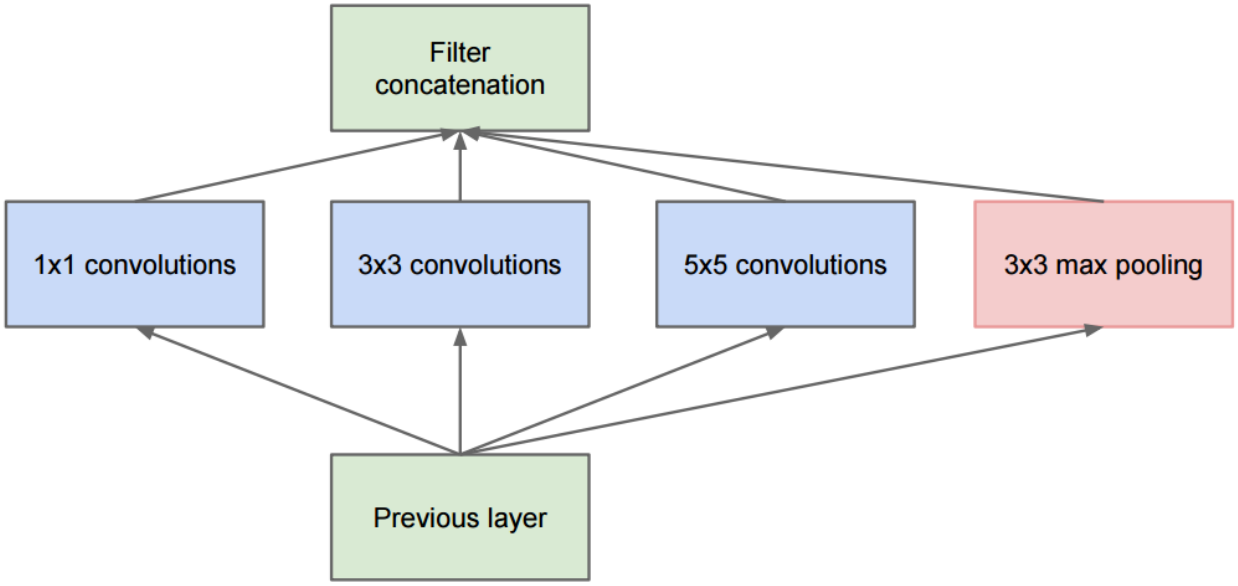
- **The Inception Module**

http://knowyourmeme.com/memes/we-need-to-go-deeper

C. Szegedy et al., Going deeper with convolutions, CVPR 2015

GoogLeNet

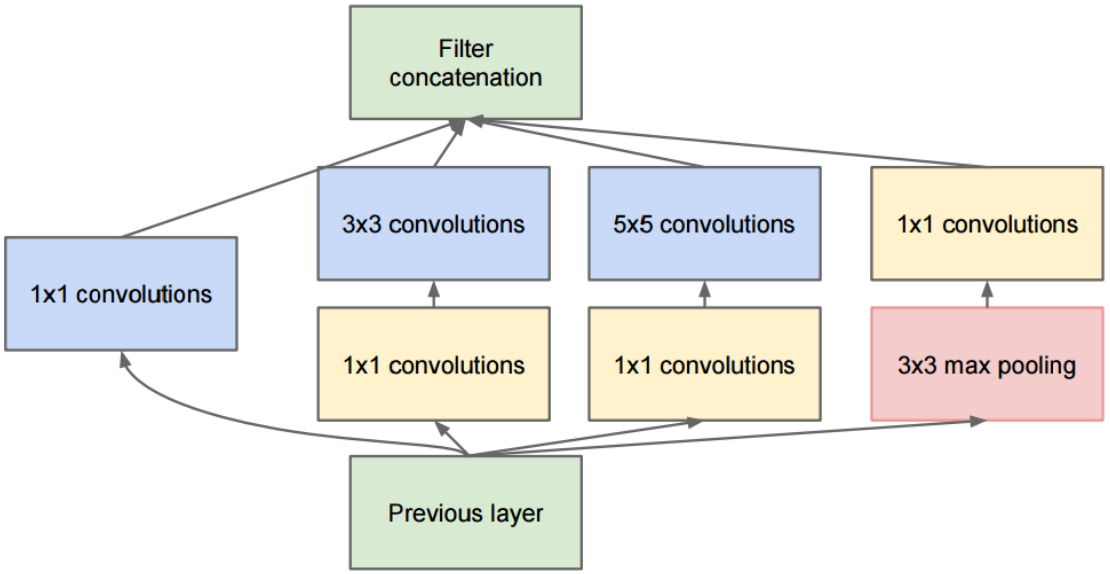
- **The Inception Module**
  - Parallel paths with different receptive field sizes and operations are meant to capture sparse patterns of correlations in the stack of feature maps



C. Szegedy et al., Going deeper with convolutions, CVPR 2015

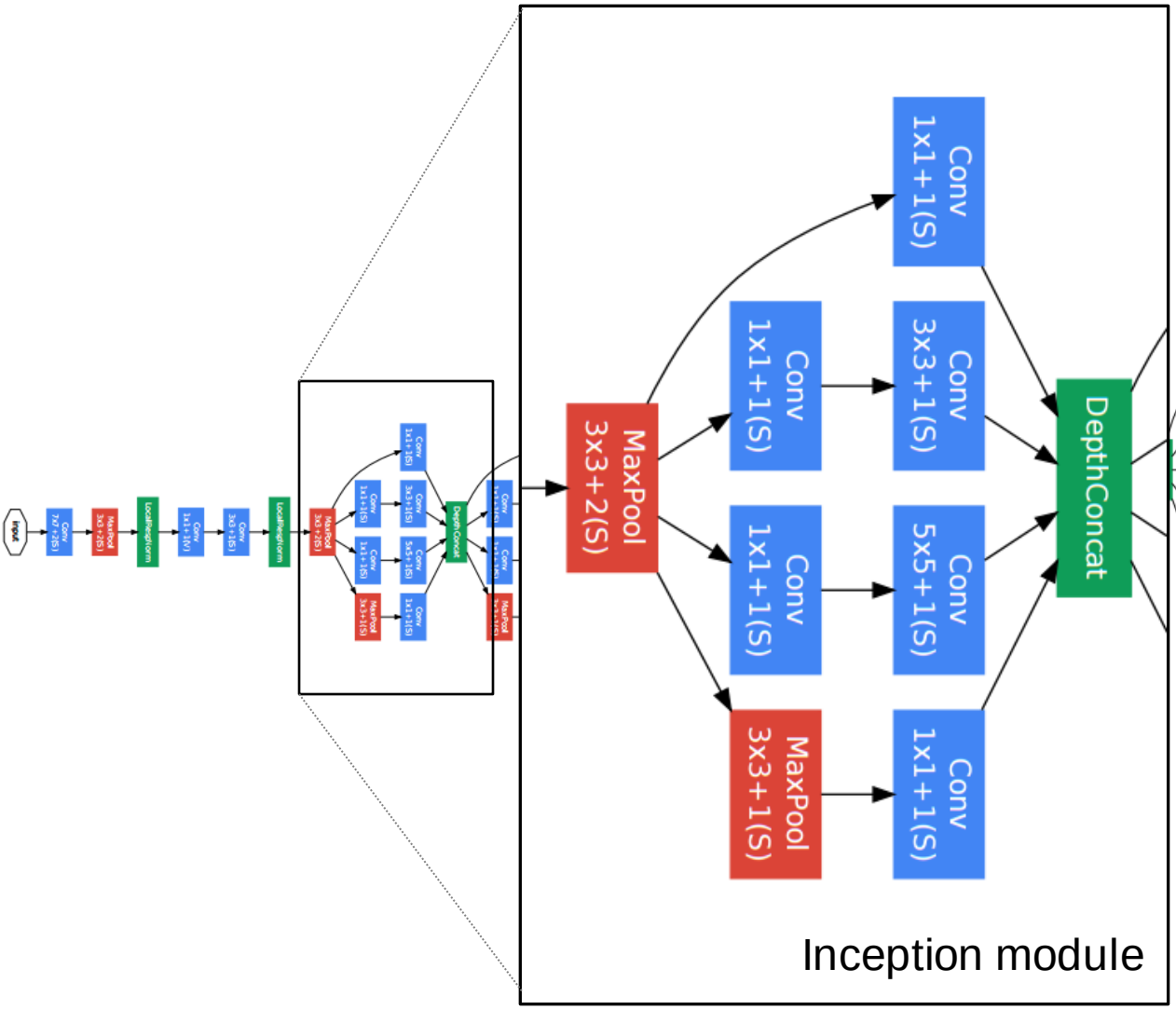
GoogLeNet

- **The Inception Module**
  - Parallel paths with different receptive field sizes and operations are meant to capture sparse patterns of correlations in the stack of feature maps
  - Use 1x1 convolutions for dimensionality reduction before expensive convolutions



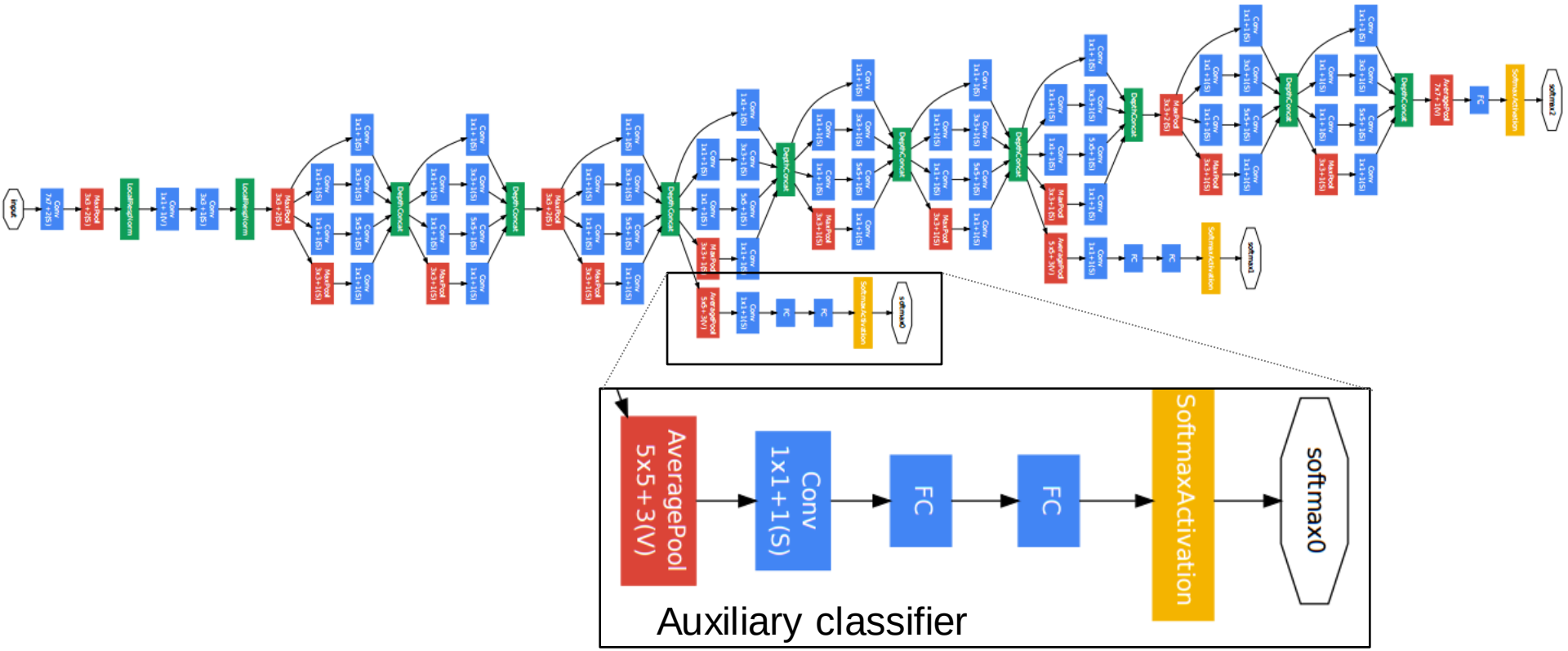
C. Szegedy et al., Going deeper with convolutions, CVPR 2015

GoogLeNet



C. Szegedy et al., Going deeper with convolutions, CVPR 2015

GoogLeNet



C. Szegedy et al., Going deeper with convolutions, CVPR 2015

GoogLeNet

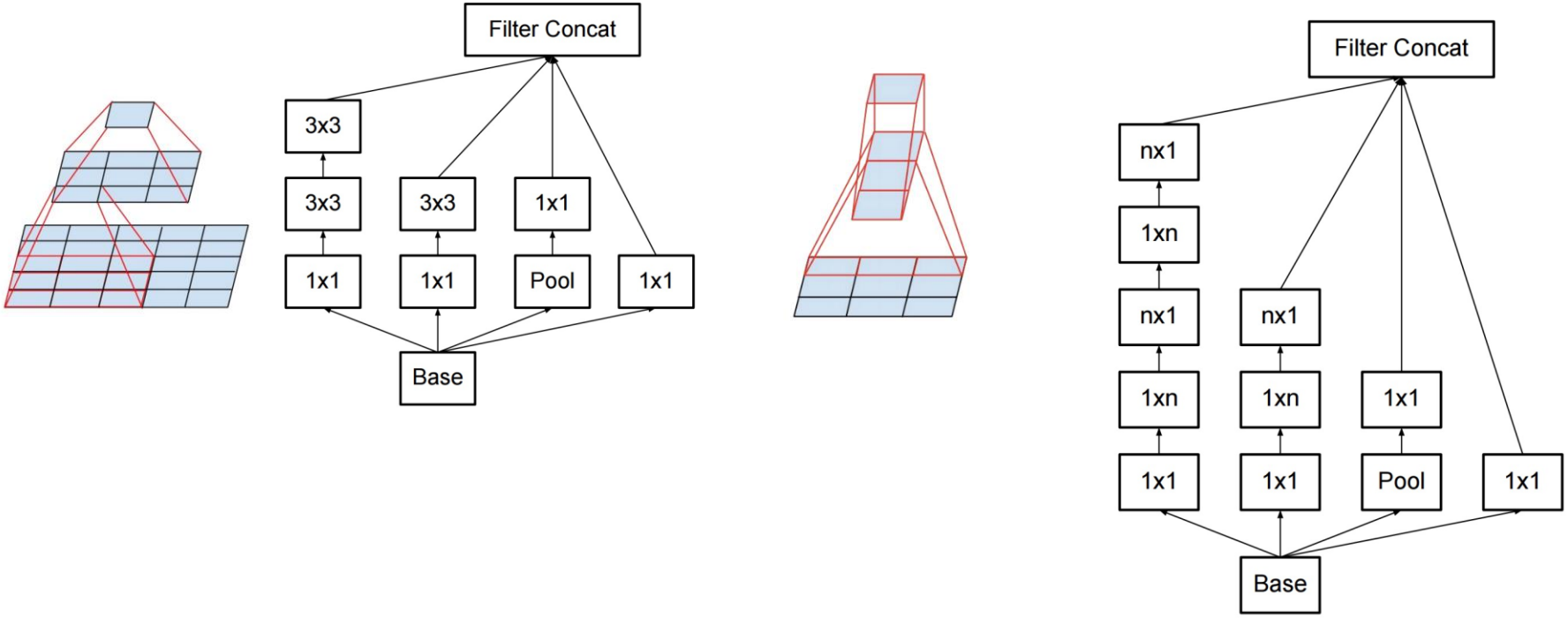
- An alternative view:

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								

C. Szegedy et al., Going deeper with convolutions, CVPR 2015

Inception v2, v3

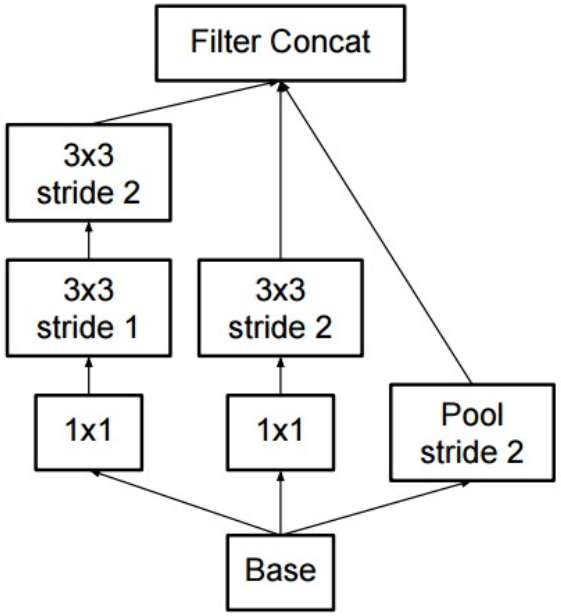
- **Regularize training with batch normalization, reducing importance of auxiliary classifiers**
- **More variants of inception modules with aggressive factorization of filters**



C. Szegedy et al., Rethinking the inception architecture for computer vision, CVPR 2016

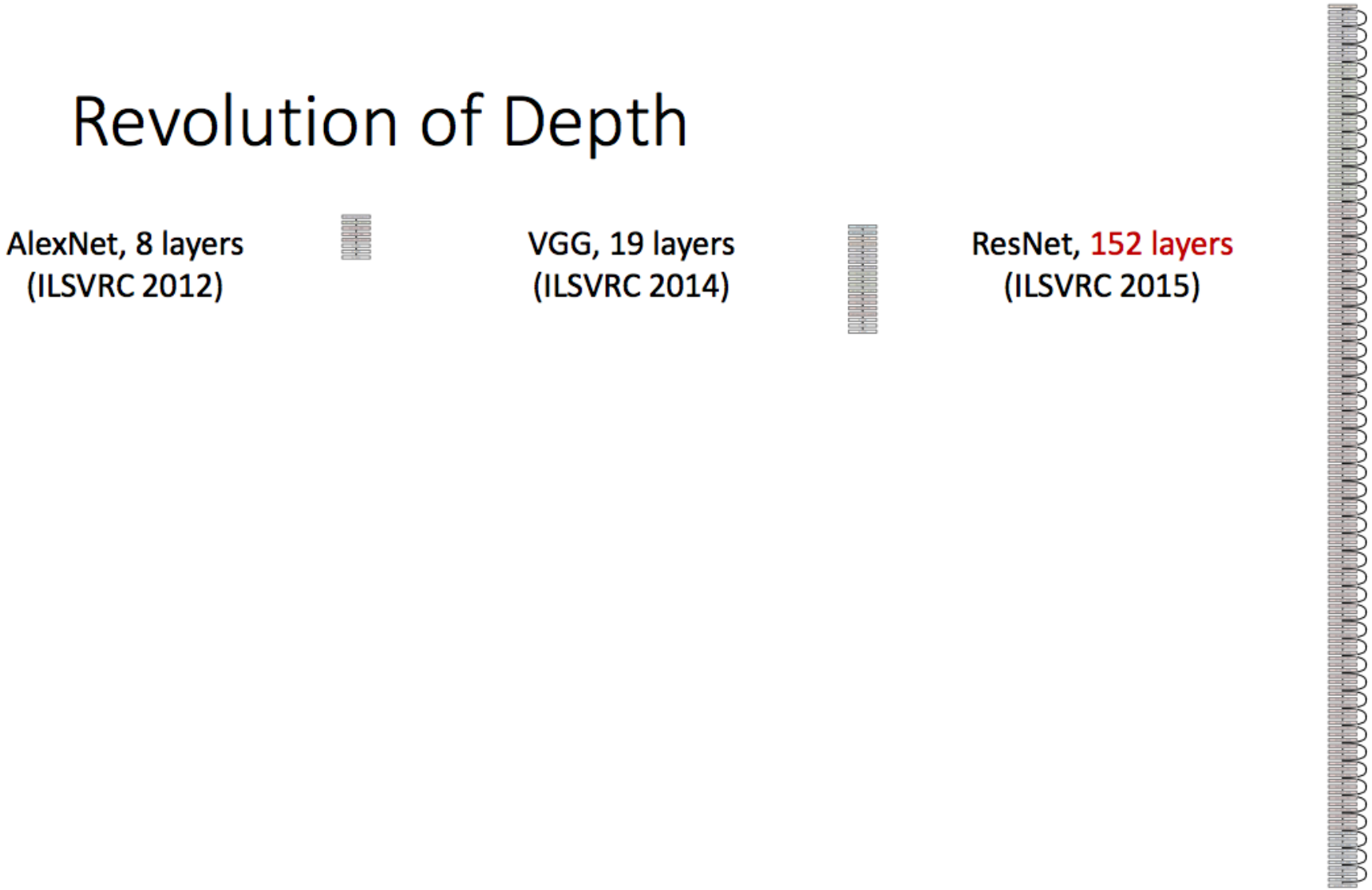
Inception v2, v3

- **Regularize training with batch normalization, reducing importance of auxiliary classifiers**
- **More variants of inception modules with aggressive factorization of filters**
- **Increase the number of feature maps while decreasing spatial resolut**



C. Szegedy et al., Rethinking the inception architecture for computer vision, CVPR 2016

ResNet: ILSVRC 2015 winner



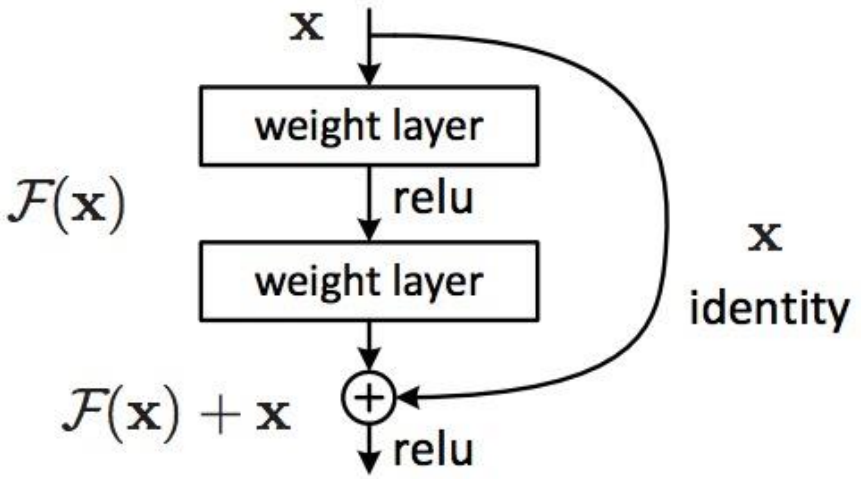
Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, Deep Residual Learning for Image Recognition, CVPR 2016


I WAS WINNING IMAGENET
UNTIL A DEEPER MODEL CAME ALONG
imgflip.com

Source (?)

ResNet

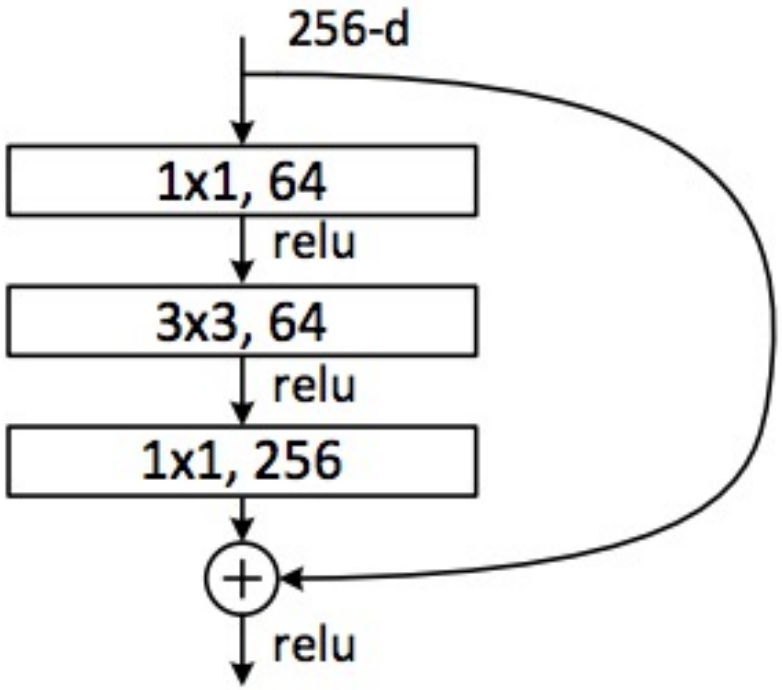
- **The residual module**
  - Introduce *skip* or *shortcut* connections (existing before in various forms in literature)
  - Make it easy for network layers to represent the identity mapping
  - For some reason, need to skip at least two layers



Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, Deep Residual Learning for Image Recognition, CVPR 2016 (Best Paper)

ResNet

Deeper residual module (bottleneck)



- Directly performing 3x3 convolutions with 256 feature maps at input and output:
  256 x 256 x 3 x 3 ~ 600K operations
- Using 1x1 convolutions to reduce 256 to 64 feature maps, followed by 3x3 convolutions, followed by 1x1 convolutions to expand back to 256 maps:
  256 x 64 x 1 x 1 ~ 16K
  64 x 64 x 3 x 3 ~ 36K
  64 x 256 x 1 x 1 ~ 16K
  Total: ~70K

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, Deep Residual Learning for Image Recognition, CVPR 2016 (Best Paper)

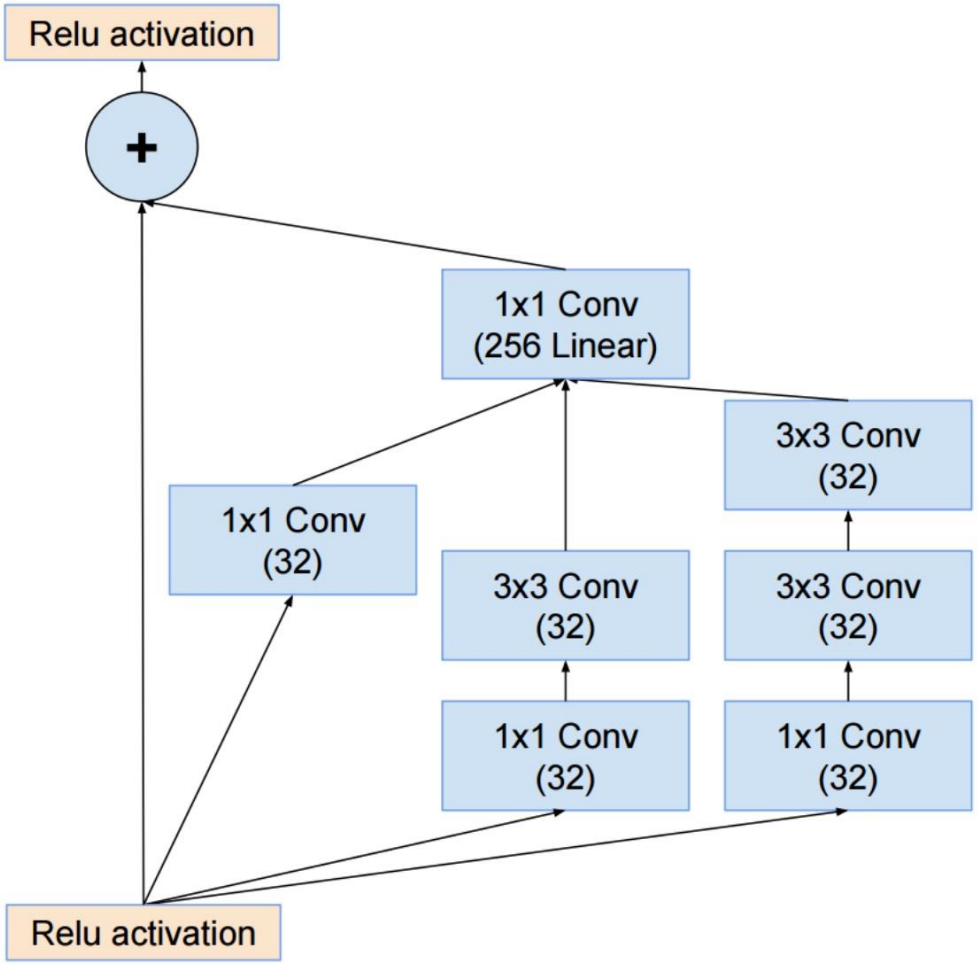
ResNet

- **Architectures for ImageNet:**

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, Deep Residual Learning for Image Recognition, CVPR 2016 (Best Paper)

Inception v4



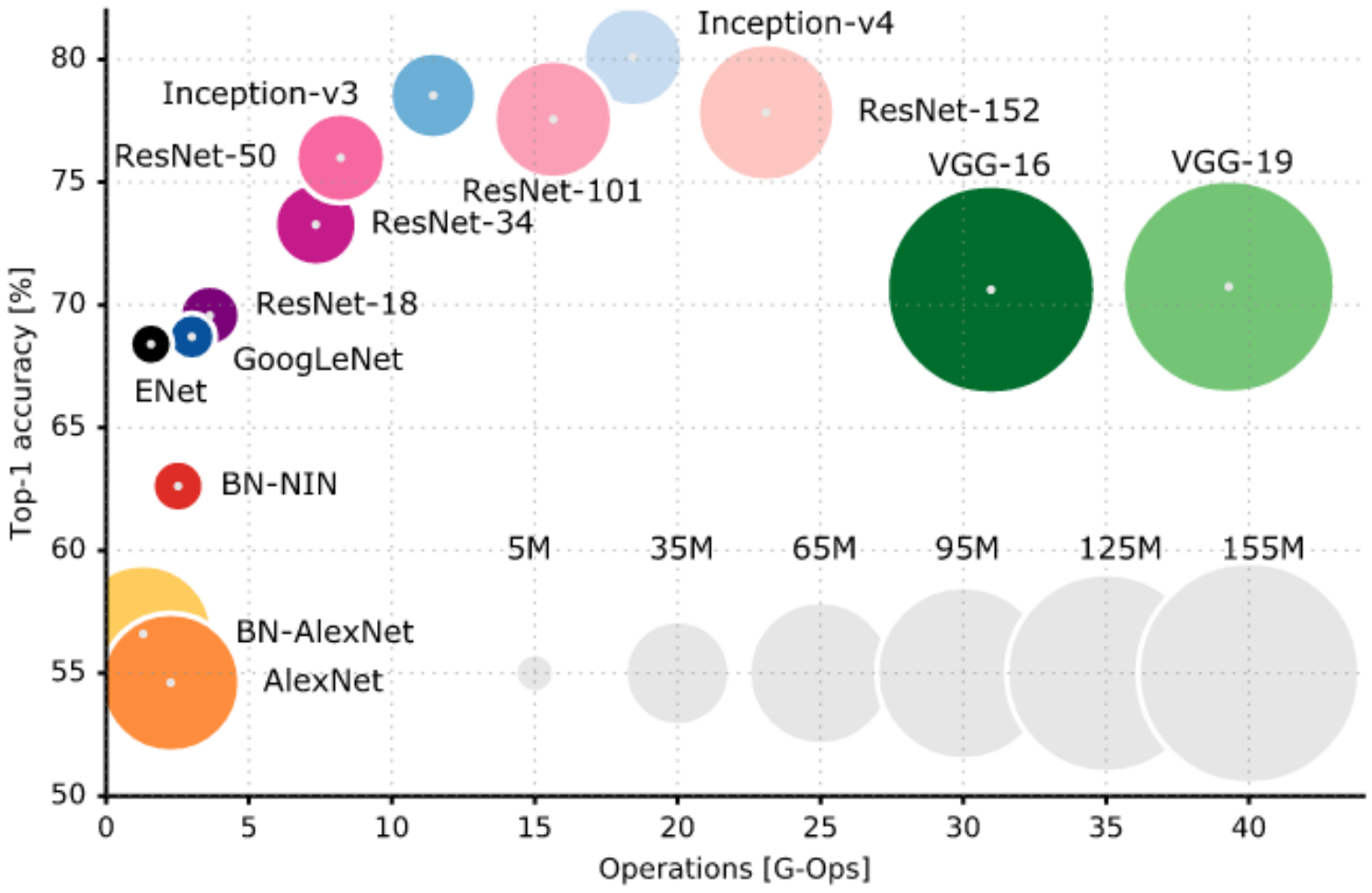
C. Szegedy et al., Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning, arXiv 2016

Summary: ILSVRC 2012-2015

Team	Year	Place	Error (top-5)	External data
SuperVision – Toronto (AlexNet, 7 layers)	2012	-	16.4%	no
SuperVision	2012	1st	15.3%	ImageNet 22k
Clarifai – NYU (7 layers)	2013	-	11.7%	no
Clarifai	2013	1st	11.2%	ImageNet 22k
VGG – Oxford (16 layers)	2014	2nd	7.32%	no
GoogLeNet (19 layers)	2014	1st	6.67%	no
ResNet (152 layers)	2015	1st	3.57%	
Human expert*			5.1%	

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

Accuracy vs. efficiency



https://culurciello.github.io/tech/2016/06/04/nets.html

Design principles

- **Reduce filter sizes (except possibly at the lowest layer), factorize filters aggressively**
- **Use 1x1 convolutions to reduce and expand the number of feature maps judiciously**
- **Use skip connections and/or create multiple paths through the network**

What's missing from the picture?

- **Training tricks and details: initialization, regularization, normalization**
- **Training data augmentation**
- **Averaging classifier outputs over multiple crops/flips**
- **Ensembles of networks**

- **What about ILSVRC 2016?**
  - No more ImageNet classification
  - No breakthroughs comparable to ResNet