Deep Learning

Lecture 3

Recap

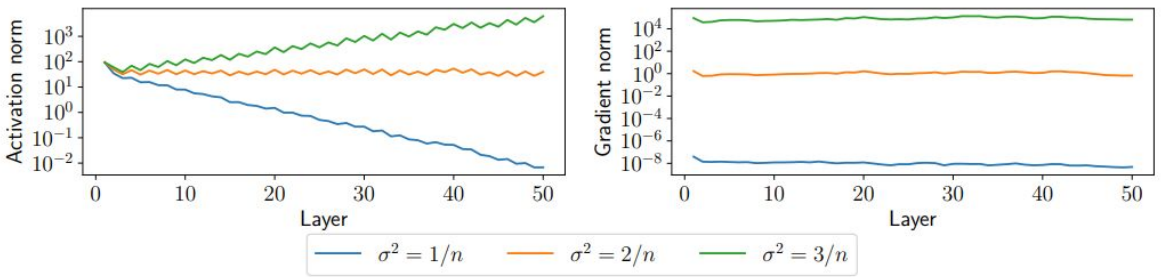
- Gradient descent for neural networks
- Weight decay
- Dropout

Motivation for weight initialization

Suppose we choose $W_i \sim \mathcal{N}(0, \frac{c}{n})$, where (for a ReLU network) $c \neq 2$...

Won't the the scale of the initial weights be "fixed" after a few iterations of optimization?

- No! A deep network with poorly-chosen weights will *never* train (at least with vanilla SGD)



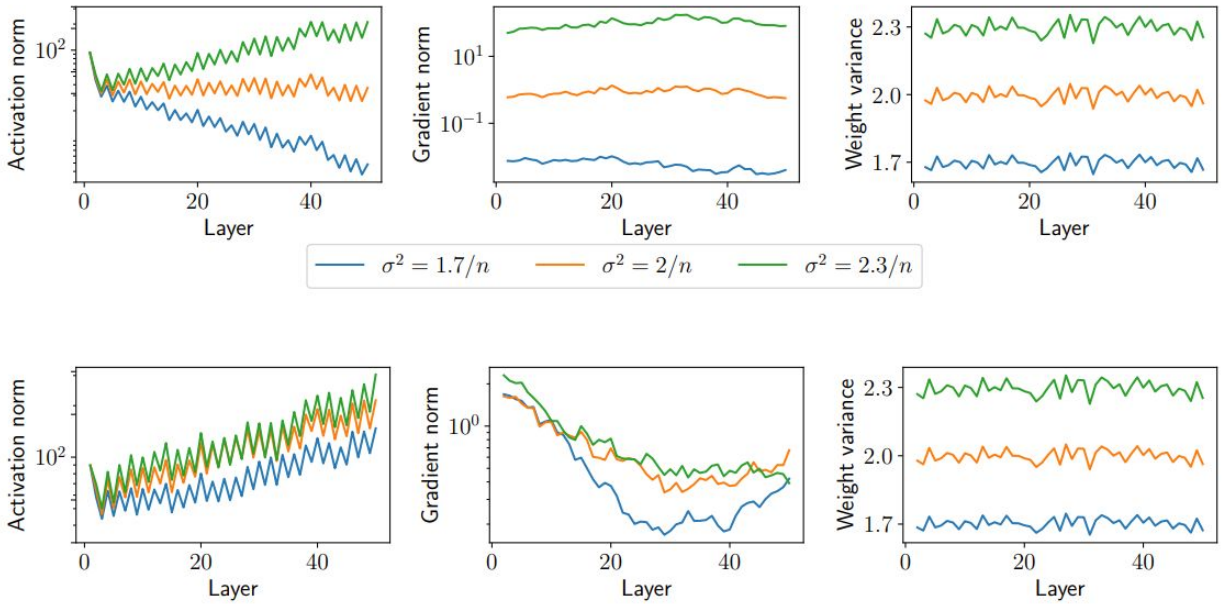
$\sigma^2 = 3/n \Longrightarrow$ NaN

$\sigma^2 = 2/n \Longrightarrow$ Works

$\sigma^2 = 1/n \Longrightarrow$ No progress

Weights after training

The problem is even more fundamental, however: even when trained successfully, the effects/scales present at initialization *persist* throughout training



Train to 5% error on MNIST

Weight Initialization

What kind of initialization can work?

- Zero initialization or constant initialization?
- Random initialization?
  - Too big and too low values
- More smart approaches?

Idea

To prevent the gradients of the network's activations from vanishing or exploding, we will stick to the following rules of thumb:

1. The *mean* of the activations should be zero.
2. The *variance* of the activations should stay the same across every layer.

$$a^{l-1} = g^{l-1}(z^{l-1})$$
$$z^{l} = W^{l}a^{l-1} + b^{l}$$
$$a^{l} = g^{l}(z^{l})$$

Desired condition:

$$\mathbb{E}(a^{l-1}) = \mathbb{E}(a^{l})$$
$$Var(a^{l-1}) = Var(a^{l})$$

https://www.deeplearning.ai/ai-notes/initialization/index.html

Xavier and He initialization

Xavier initialization (for sigmoid and tanh)

$$W^{[l]} \sim \mathcal{N}\left(\mu = 0, \sigma^2 = \frac{1}{n^{[l-1]}}\right) \quad b^{[l]} = 0$$

He initialization (for ReLU)

$$W^{[l]} \sim \mathcal{N}\left(\mu = 0, \sigma^2 = \frac{2}{n^{[l-1]}}\right) \quad b^{[l]} = 0$$

$W^{[l]}$ - weights of l layer

$n^{[l-1]}$ - number of neuron in (l-1) layer

$b^{[l]}$ - bias of l layer

Motivation for normalization

Initialization matters a lot for training, and can vary over the course of training to no longer be “consistent” across layers / networks

But remember that a “layer” in deep networks can be any computation at all…

…let’s just add layers that “fix” the normalization of the activations to be whatever we want!

Layer normalization

First idea: let's normalize (mean zero and variance one) activations at each layer; this is known as layer normalization

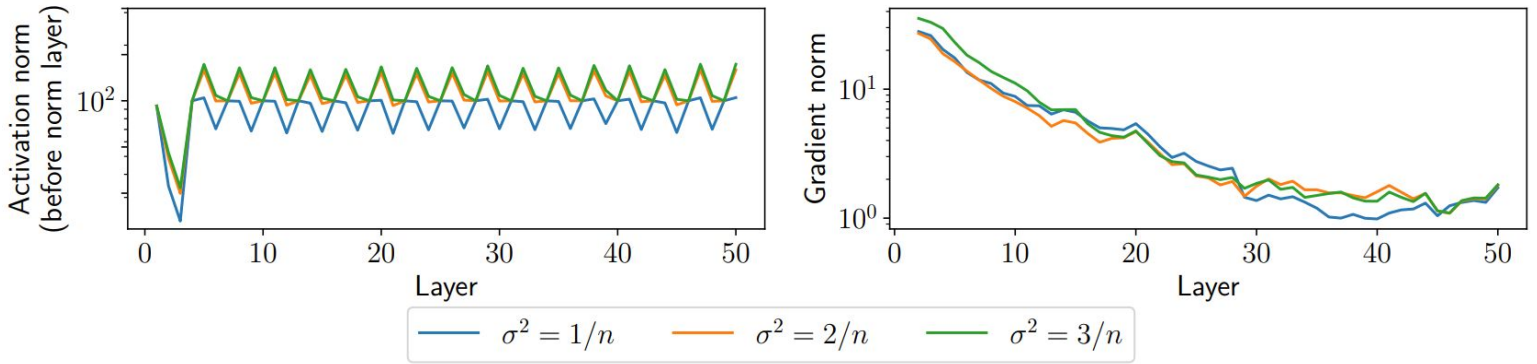
$$\hat{z}_{i+1} = \sigma_i \left(W_i^T z_i + b_i \right)$$

$$z_{i+1} = \frac{\hat{z}_{i+1} - \mathbf{E}\left[\hat{z}_{i+1}\right]}{\left(\mathrm{Var}\left[\hat{z}_{i+1}\right] + \epsilon\right)^{1/2}}$$

Also, common to add a scalar weight and bias to each term (only changes representation e.g., if we put normalization prior to nonlinearity instead)

Layer normalization: Illustration

“Fixes” the problem of varying norms of layer activations (obviously)



In practice, for standard FCN, harder to train resulting networks to low loss
(relative norms of examples are a useful discriminative feature)

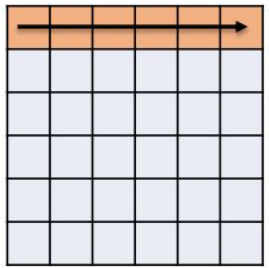
Batch Normalization

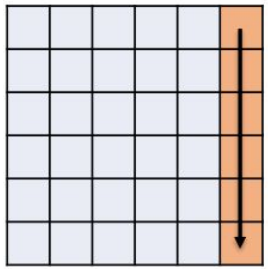
An odd idea: let's consider the matrix form of our updates

$$\hat{Z}_{i+1} = \sigma_i(Z_i W_i + b_i^T)$$

then layer normalization is equivalent to normalizing the *rows* of this matrix

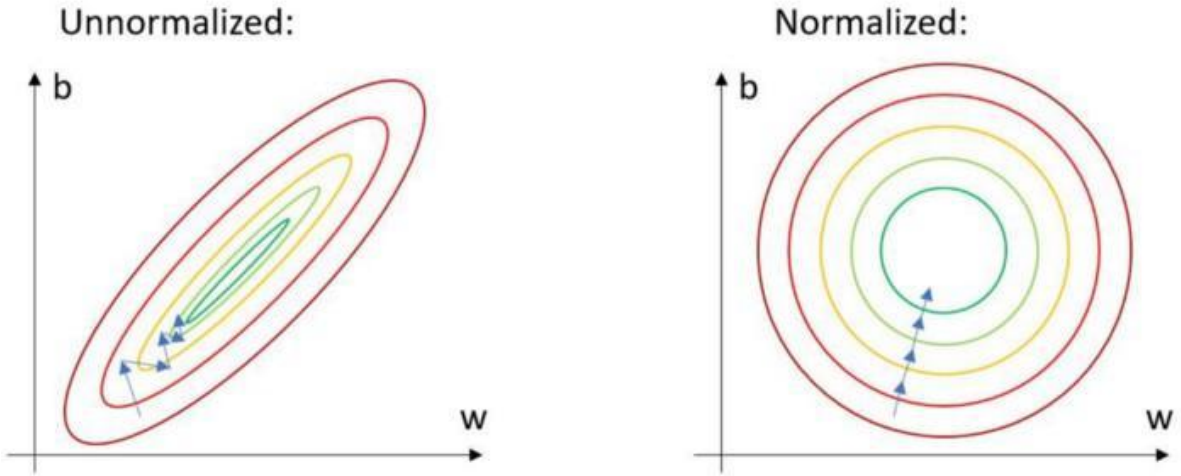
What if, instead, we normalize it's columns? This is called *batch normalization*, as we are normalizing the activations *over the minibatch*

$\hat{Z}_{(i+1)} =$  Normalize (layer norm)

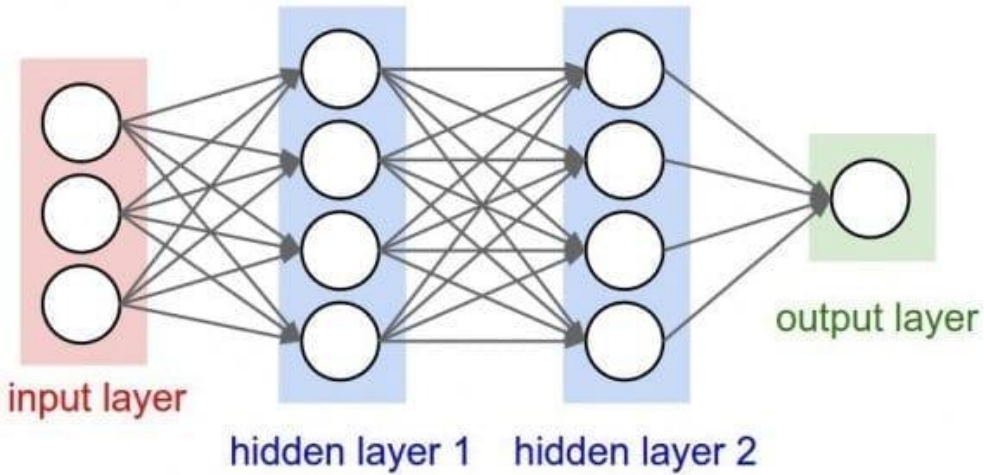
$\hat{Z}_{(i+1)} =$  Normalize (batch norm)

Batch Normalization

- More stable computations
- Better for optimization methods
- Removing covariance shift



What about neural networks?



- Can’t use whole dataset
- Idea: use minibatch

$$\{x_{ij}\}_{i=1,j=1}^{N_{batch},d}$$

$$\mu_j = \frac{\sum_{i=1}^{N_{batch}} x_{ij}}{N_{batch}}$$

$$\sigma_j^2 = \frac{\sum_{i=1}(x_{ij} - \mu_j)^2}{N_{batch}}$$

Batch normalisation

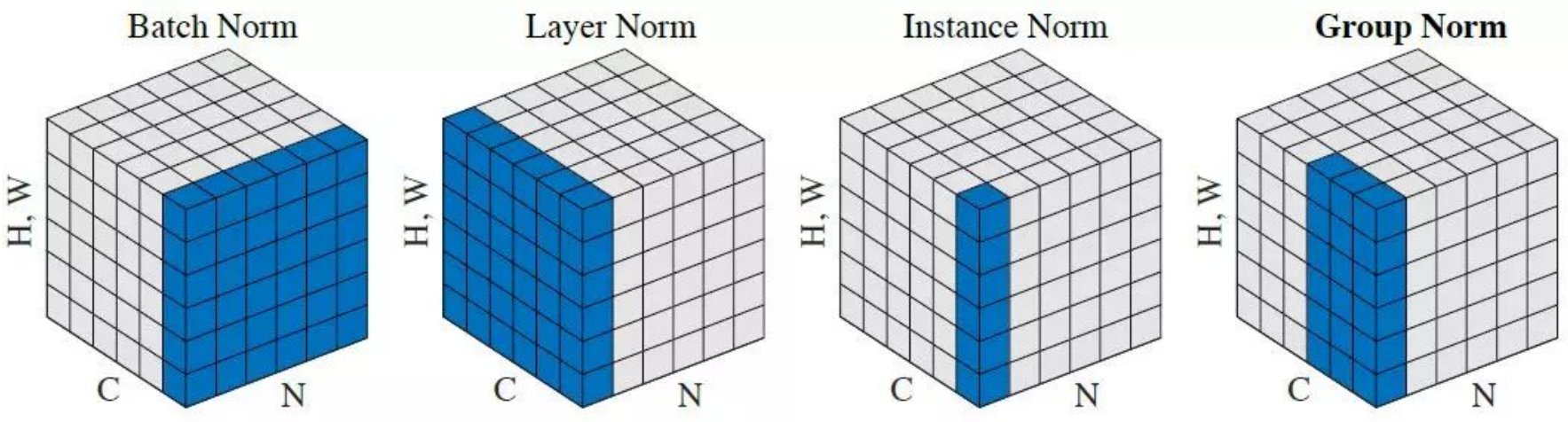
$$\mu_j = \frac{\sum_{i=1}^{N_{batch}} x_{ij}}{N_{batch}}$$

$$\sigma_j^2 = \frac{\sum_{i=1}(x_{ij} - \mu_j)^2}{N_{batch}}$$

$$\hat{y}_{ij} = \frac{x_{ij} - \mu_j}{\sqrt{\sigma_{ij}^2 + \varepsilon}}$$

$$y_{ij} = \gamma_j \hat{y}_{ij} + \delta_j$$

Normalization



Essential in classical architectures

Essential in modern architectures

Used in image generation to balance contrast of images

Used sometimes…

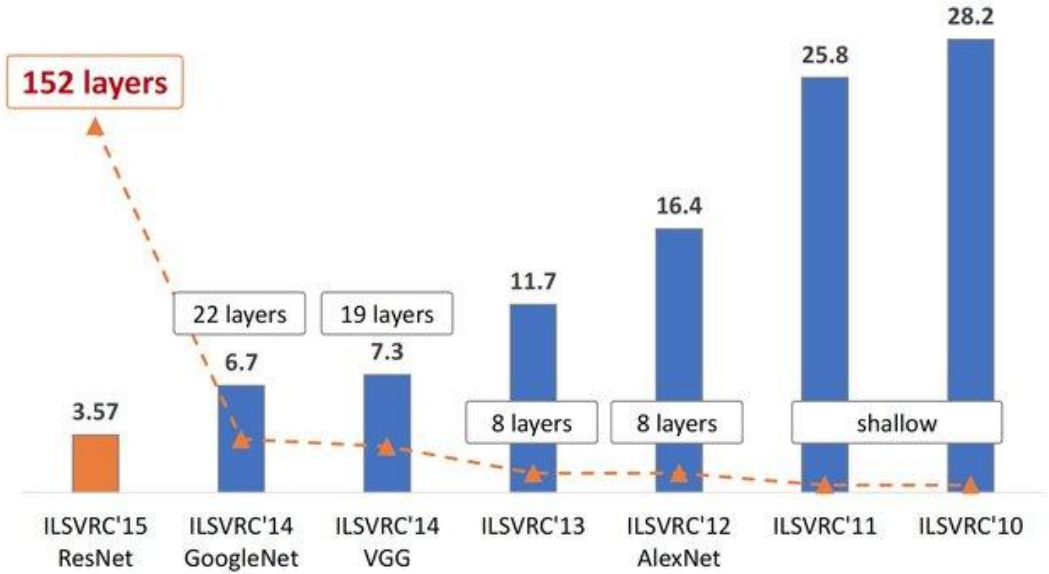
Convolution NN

ImageNet

Everything starts with data!

- The ImageNet dataset contains 14,197,122 annotated images.
- Total number of non-empty WordNet synsets: 21841.
- Since 2010 the dataset is used in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), a benchmark in image classification and object detection.

ImageNet Results

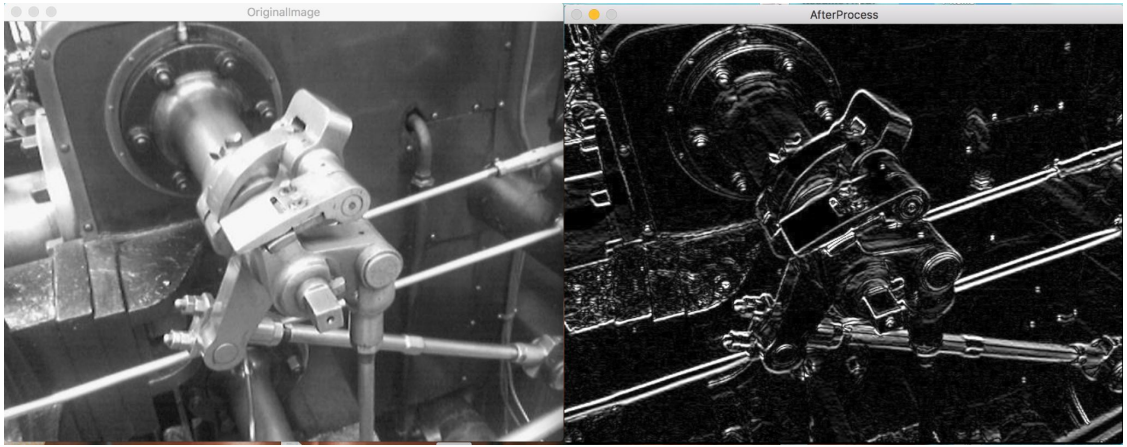


Why we can’t use MLP for images?

- Too many parameters
- Fixed dimension of images
- Features will be too correlated

Sobel and other filters

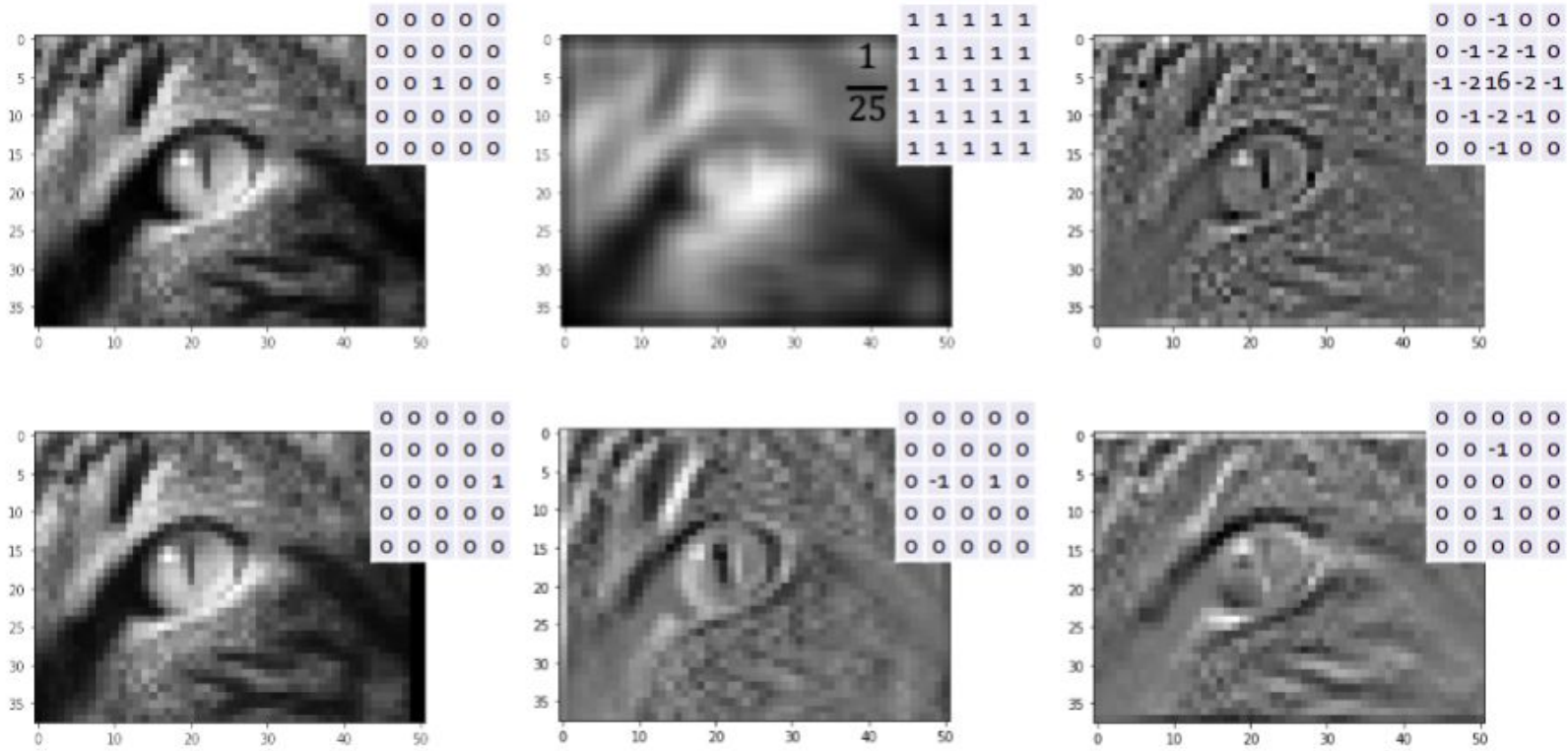
Motivation for application of convolution in computer vision comes from computational imaging tasks. For example, to extract edges from image one can use Sobel filter.



Why are edges important? Because our brain really react on edges

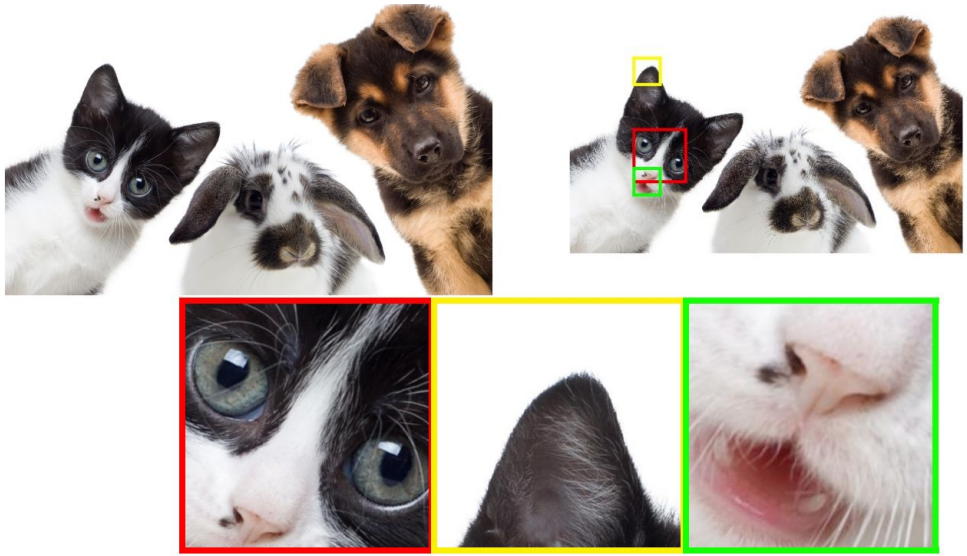
https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm
https://www.researchgate.net/publication/220695992_Machine_Perception_of_Three-Dimensional_Solids

Sobel and other filters



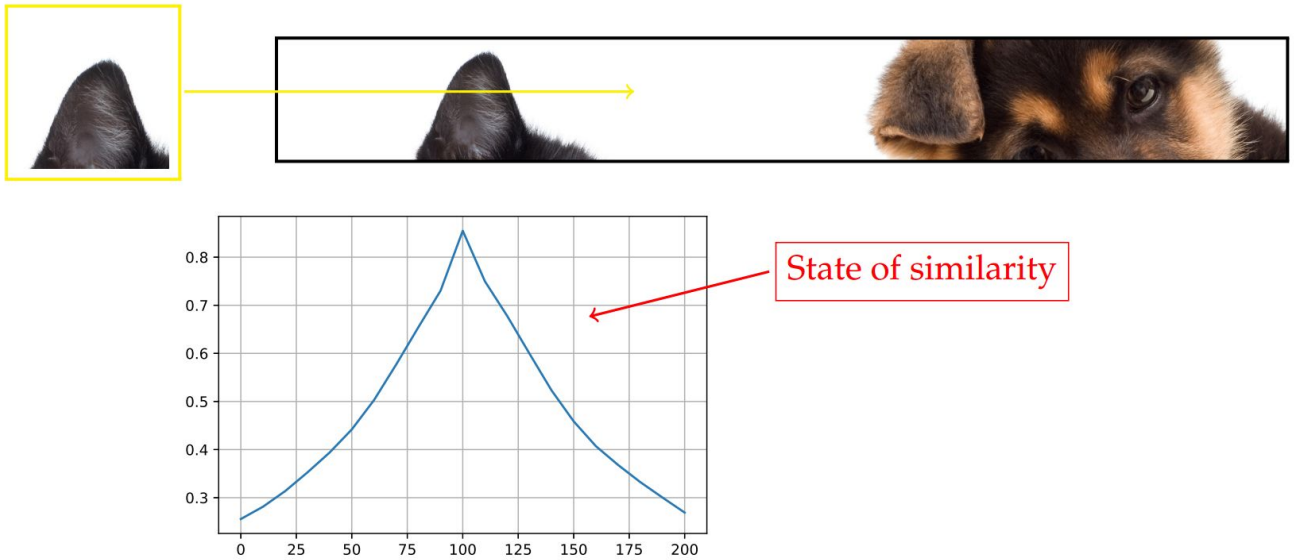
Motivation for convolution

- To perform identification we are seeking for noticeable parts of an object ·
- However, these parts appear at different locations ·
- Want to perform a feature search over the whole picture



Motivation for convolution

Result of visual searches looks like this:

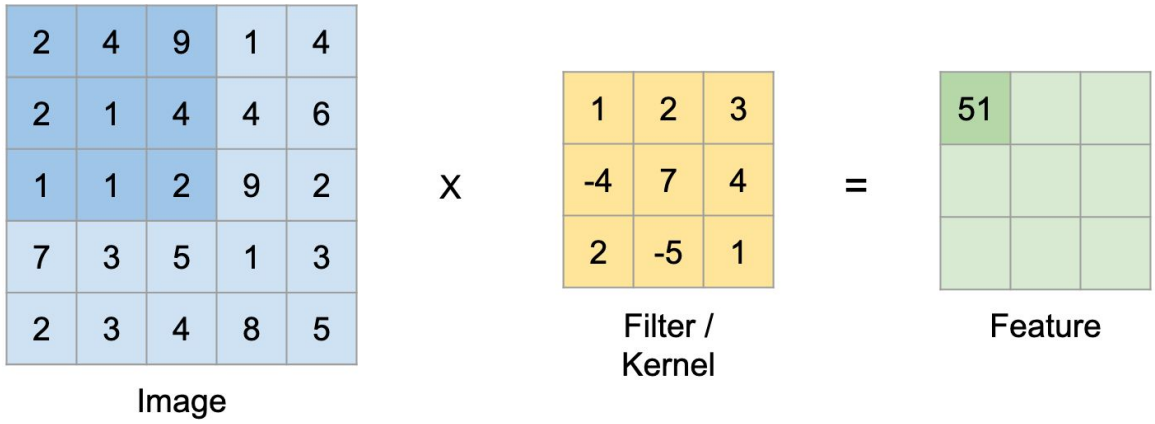


With the high probability we can detect the chosen feature

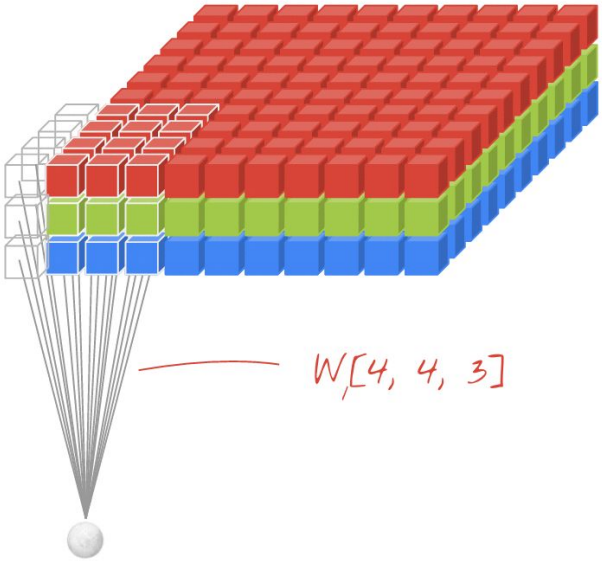
Convolution

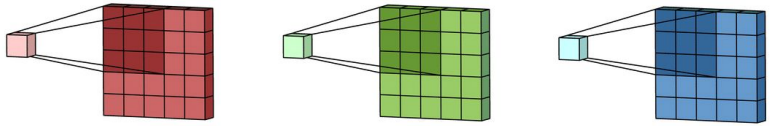
Mathematically, 2D convolution can be written in the following form

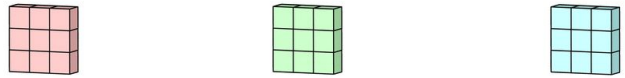
$$Y(i,j) = \sum_{u,v} X(i+u, j+v) K(u,v)$$



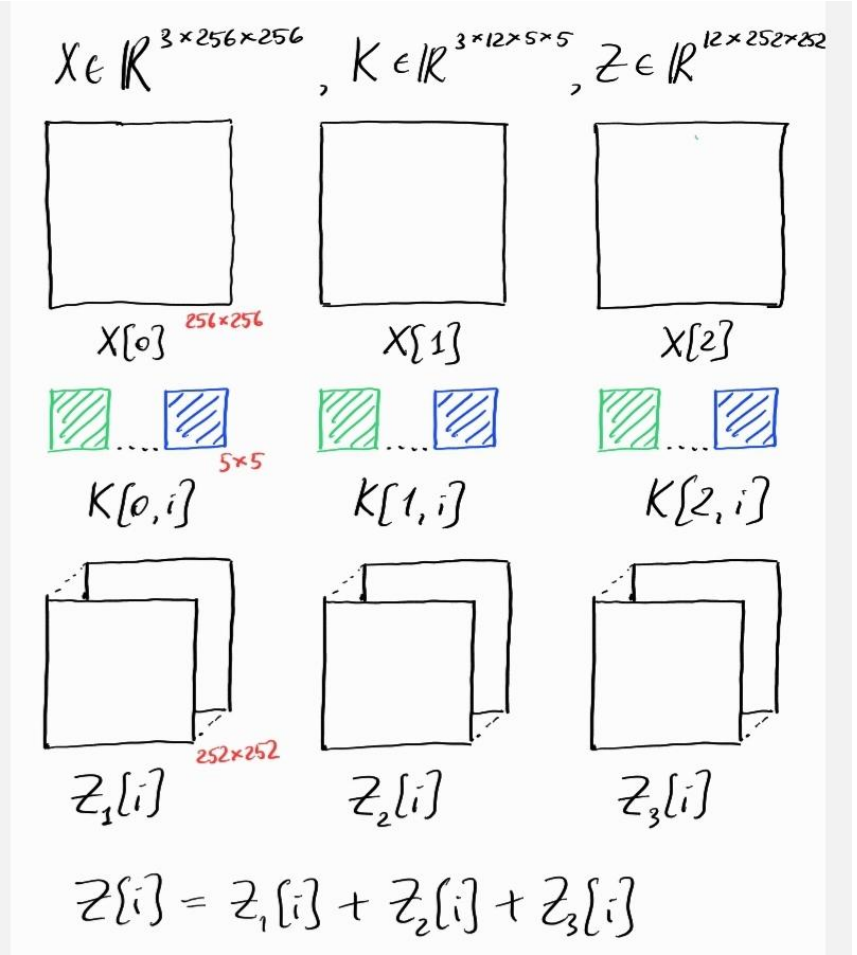
Convolution for 3D tensor







Convolution for 3D tensor



Backpropagation for Conv 1d

Recap: Linear Layer Gradient

1) $Y = X \cdot W$

$dY = X\,dW + dX \cdot W$

$dL = \langle \nabla_Y L, dY \rangle = \langle \nabla_Y L, X\,dW \rangle + \langle \nabla_Y L, dX \cdot W \rangle$

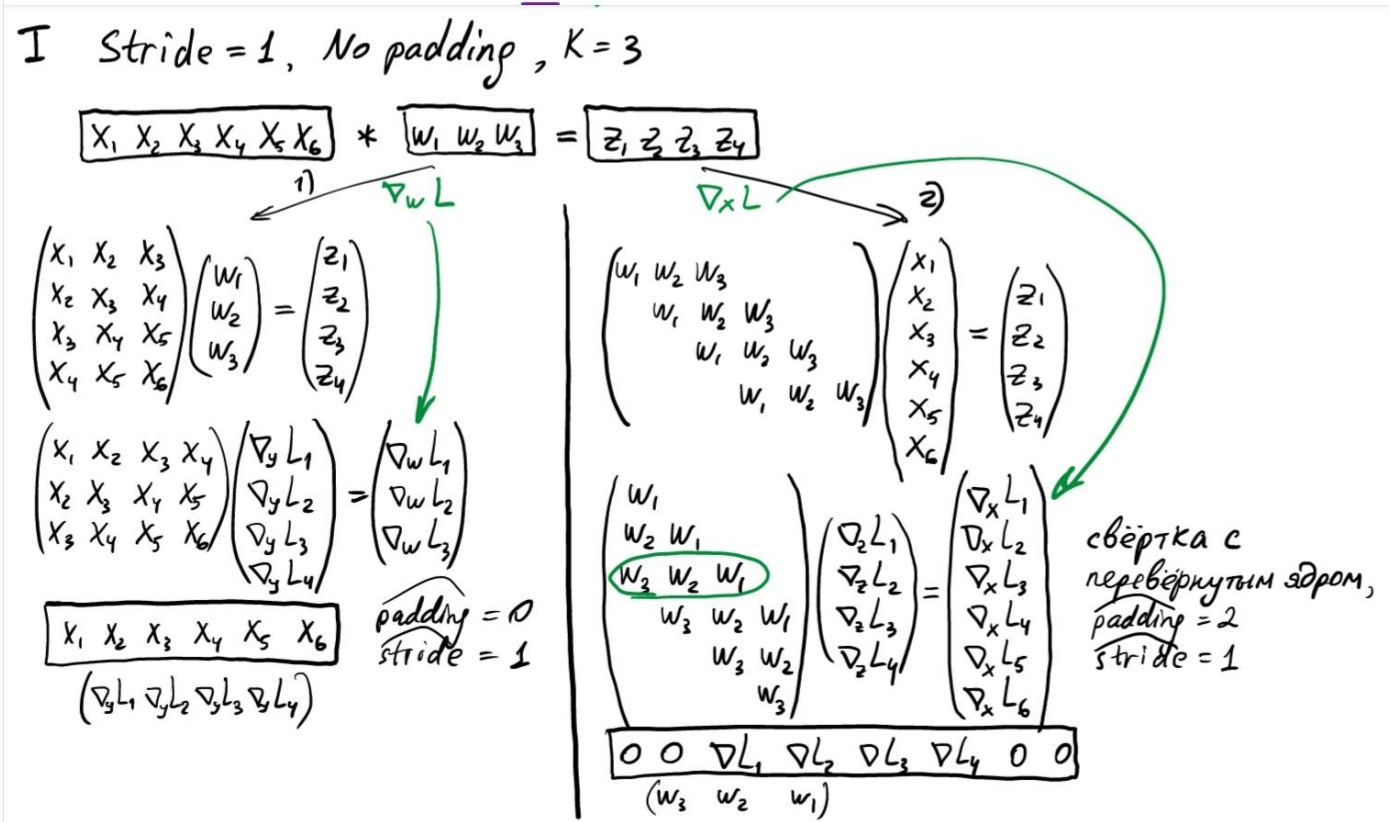
$= \langle \underbrace{X^T \nabla_Y L}_{\nabla_W L}, dW \rangle + \langle \underbrace{\nabla_Y L \cdot W^T}_{\nabla_X L}, dX \rangle$

2) $y = W \cdot x$

$dy = W\,dx + dW \cdot x$

$dL = \langle \nabla_y L, dy \rangle = \langle \underbrace{W^T \nabla_y L}_{\nabla_x L}, dx \rangle + \langle \underbrace{\nabla_y L\, x^T}_{\nabla_W L}, dW \rangle$

Backpropagation for Conv 1d



Backpropagation for Conv 2d

The interesting question is how to calculate a gradient for convolution?

Idea: convolution is a linear operator!

$$\begin{pmatrix} y_1 & y_2 \\ y_3 & y_4 \end{pmatrix} = \begin{pmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{pmatrix} \times \begin{pmatrix} k_1 & k_2 \\ k_3 & k_4 \end{pmatrix}$$

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} k_1 & k_2 & 0 & k_3 & k_4 & 0 & 0 & 0 & 0 \\ 0 & k_1 & k_2 & 0 & k_3 & k_4 & 0 & 0 & 0 \\ 0 & 0 & 0 & k_1 & k_2 & 0 & k_3 & k_4 & 0 \\ 0 & 0 & 0 & 0 & k_1 & k_2 & 0 & k_3 & k_4 \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix}$$

$$y = Kx$$

Wow, we can calculate a gradient, but let's not stop here and go in details!

Backpropagation for Conv 2d

Doing some boring mathematics…

$$y = Kx$$

$$\nabla_y L \rightarrow \nabla_x L, \nabla_K L$$

$$dy = dKx + Kdx$$

$$dL = \nabla_y L^T dy = \nabla_y L^T (dKx + Kdx) = tr(\nabla_K L^T dK) + \nabla_x L^T dx$$

$$\nabla_x L = K^T \nabla_y L \qquad \nabla_K L = \nabla_y L x^T$$

Backpropagation for Conv 2d

The gradient which is passed further in the network can be written in the following form:

$$\nabla_x L = K^T \nabla_y L \qquad \nabla_x L = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & (\nabla_y L)_1 & (\nabla_y L)_2 & 0 \\ 0 & (\nabla_y L)_3 & (\nabla_y L)_4 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} k_4 & k_3 \\ k_2 & k_1 \end{pmatrix}$$

Wow, it’s a convolution again!

Backpropagation for Conv 2d

$$\nabla_K L = \nabla_y L x^T \qquad dK = \begin{pmatrix} dk_1 & dk_2 & 0 & dk_3 & dk_4 & 0 & 0 & 0 & 0 \\ 0 & dk_1 & dk_2 & 0 & dk_3 & dk_4 & 0 & 0 & 0 \\ 0 & 0 & 0 & dk_1 & dk_2 & 0 & dk_3 & dk_4 & 0 \\ 0 & 0 & 0 & 0 & dk_1 & dk_2 & 0 & dk_3 & dk_4 \end{pmatrix}$$

$$dL = tr(\nabla_K L^T dK) = \nabla_k L^T dk \qquad dk = \begin{pmatrix} dk_1 \\ dk_2 \\ dk_3 \\ dk_4 \end{pmatrix}$$

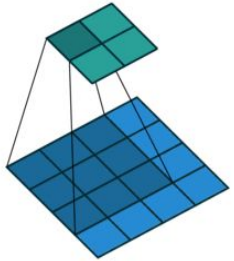
$$(\nabla_k L)_1 = (\nabla_K L)_{11} + (\nabla_K L)_{22} + (\nabla_K L)_{34} + (\nabla_K L)_{45}$$

$$(\nabla_k L)_1 = (\nabla_y L)_1 x_1 + (\nabla_y L)_2 x_2 + (\nabla_y L)_3 x_4 + (\nabla_y L)_4 x_5$$

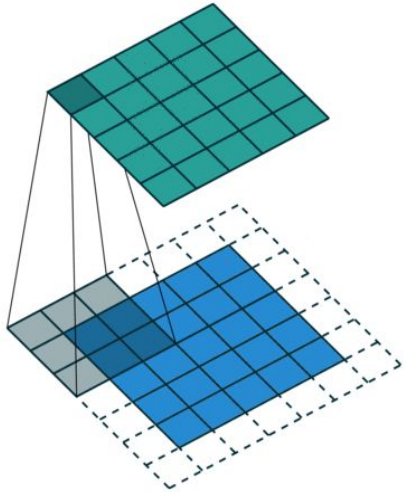
$$\nabla_k L = X \times \nabla_y L$$

Convolution again!

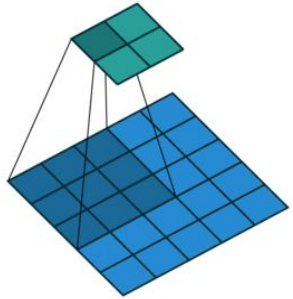
Different types of convolution



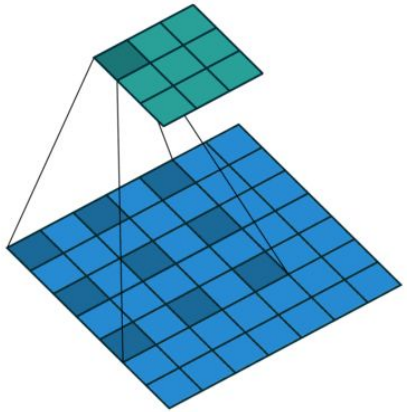
No padding,
no strides



Padding 1,
no strides

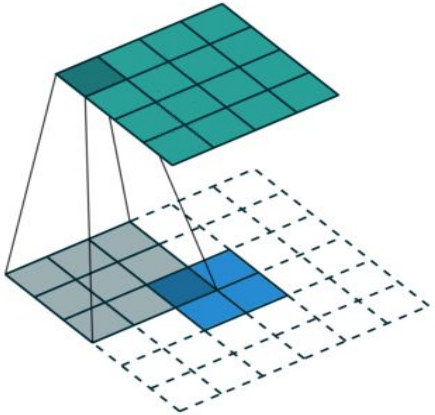


No padding,
stride = 2

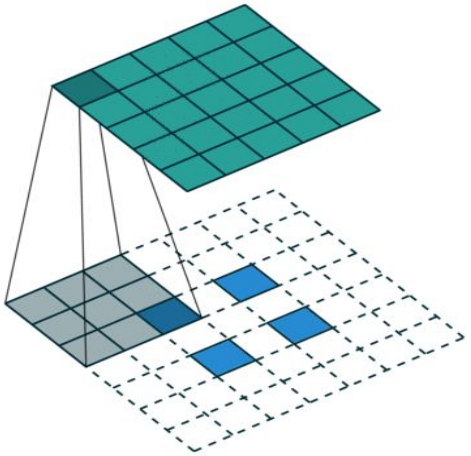


No padding,
no stride,
dilation = 2

Different types of convolution

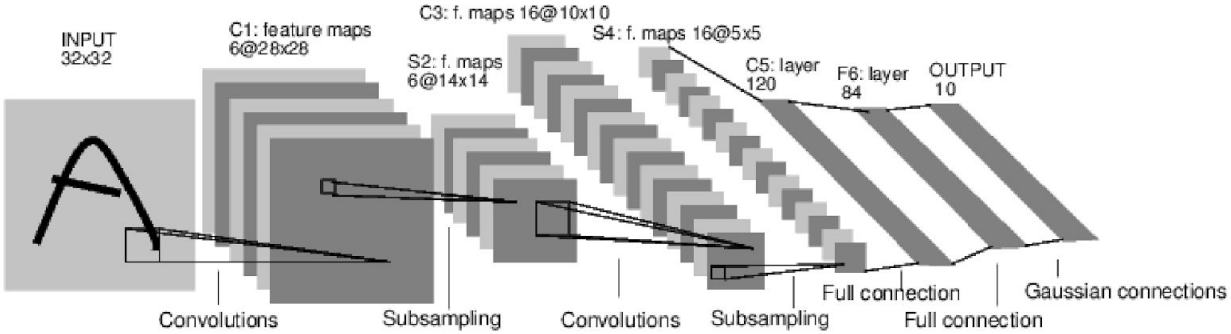


Transposed convolution



Transposed convolution
stride = 2

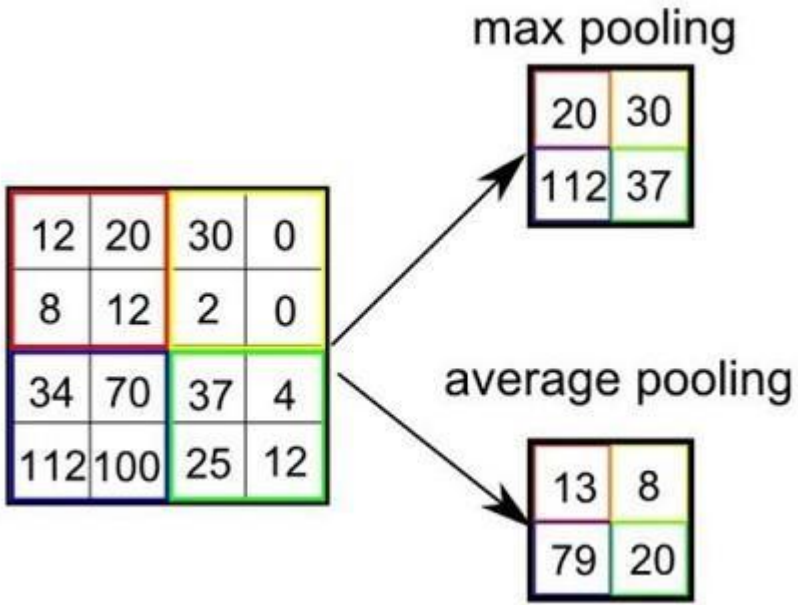
LeNet [LeCun et al., 1998]



Operation types:

- Convolutions
- Nonlinearities
- Pooling
- Full connection layers

What is pooling?



- Usual motivation: adding invariance to small shifts
- Several max-poolings can accumulate invariance to stronger shifts
- No invariance/covariance to scaling, rotation, color changes!

Recap

- Weight initialization
- Batch Normalization
- CNN