Intro

- In this video you will learn about one more useful layer of neurons;
- We will build our first fully working neural network for images!

A color image input

Let's say we have a color image as an input, which is $W \times H \times C_{in}$ **tensor** (multidimensional array), where

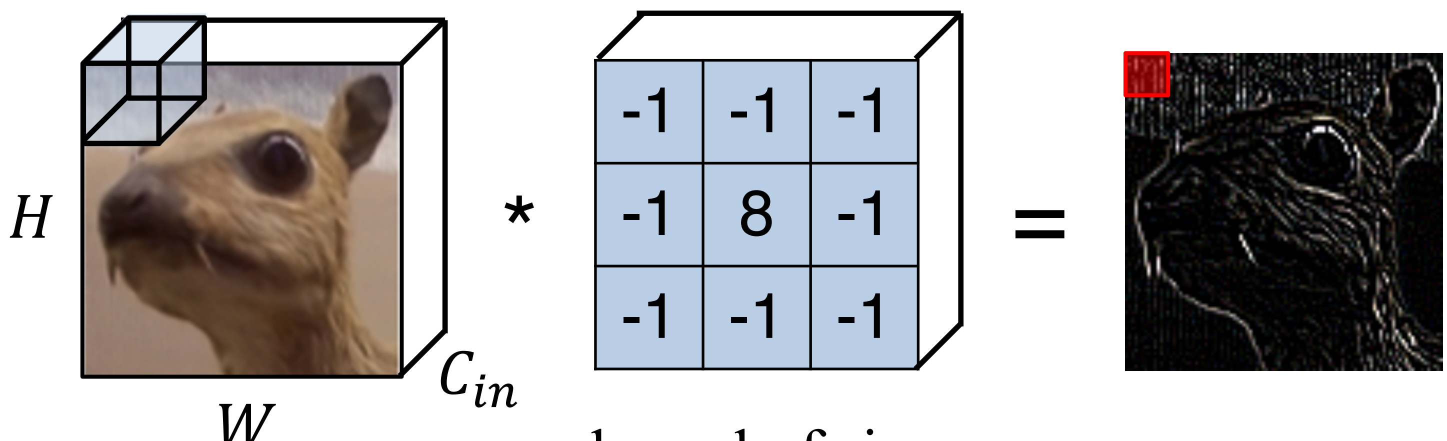
- W – is an image width,
- H – is an image height,
- C_{in} – is a number of input channels (e.g. 3 RGB channels).

A color image input

Let's say we have a color image as an input, which is W$\times H \times C_{in}$ **tensor** (multidimensional array), where

- W – is an image width,
- H – is an image height,
- C_{in} – is a number of input channels (e.g. 3 RGB channels).

We want to incorporate color! So kernel will be 3D as well.



Each patch is called a volumetric patch, as it represents a volume in the image space.

kernel of size $W_k \times H_k \times C_{in}$

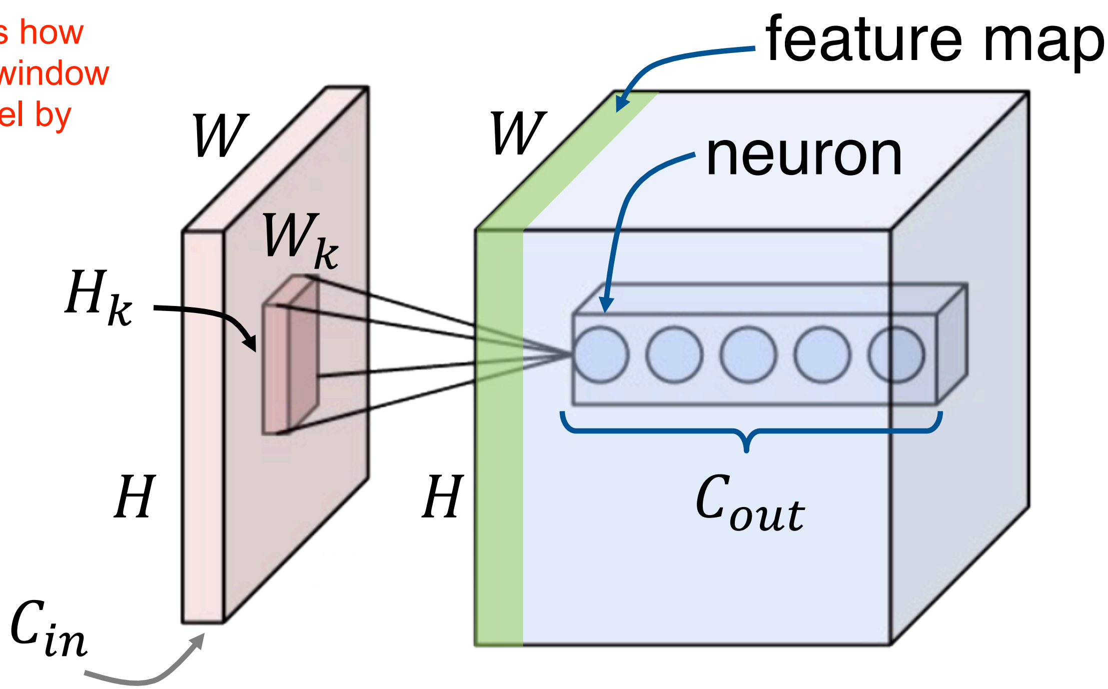
feature map

One kernel is not enough!

We choose Cout by ourselves. Cout is the depth dimension of the feature map.
This makes it enable to get deeper features, presumably.

- We want to train C_{out} kernels of size $W_k \times H_k \times C_{in}$.
- Having a stride of 1 and enough zero padding we can have $W \times H \times C_{out}$ output neurons.

recall that stride is how much we slide the window by. so one is pixel by pixel.



- Using $(W_k * H_k * C_{in} + 1) * C_{out}$ parameters.

Each feature map kernel is a 3D tensor (Wk x Hk x Cin) and we train a bias term for every feature map.

One convolutional layer is not enough!

- Let's say neurons of the 1st convolutional layer look at the patches of the image of size 3x3.
- What if an object of interest is bigger than that?
- We need a 2nd convolutional layer on top of the 1st!

2nd 3x3 conv

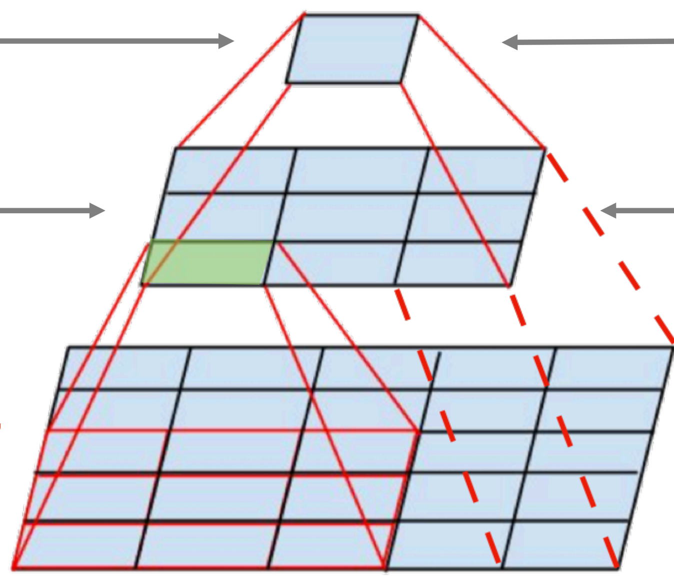
receptive field 5x5

Here, we can see 5 by 5 squares. It is 'receptive' of them.

1st 3x3 conv

receptive field 3x3

See the green square. It can see 3 by 3 squares below.

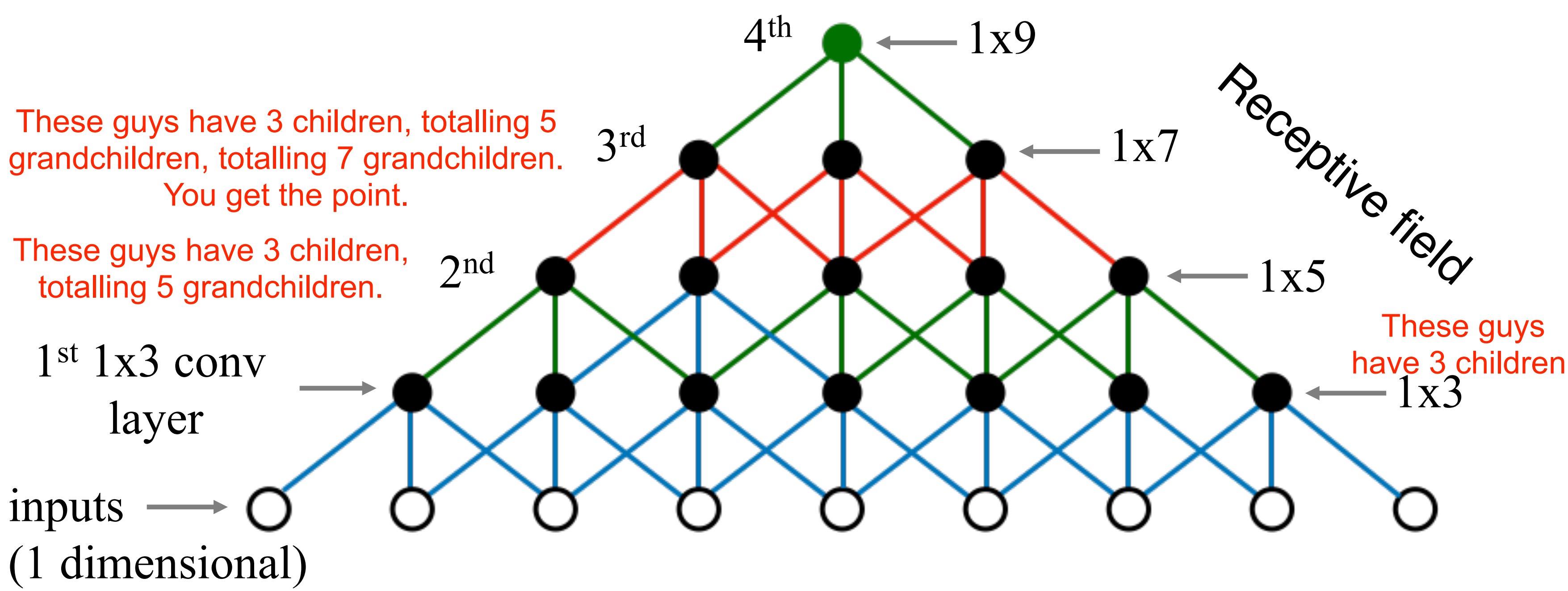


We introduce the concept of a receptive field. In essence, a layer has a receptive field n by n, if for one square in that layer, it can 'see'n by n squares in its children & grandchildren below.

Receptive field after N convolutional layers

KERNEL SIZE 3, STRIDE 1.

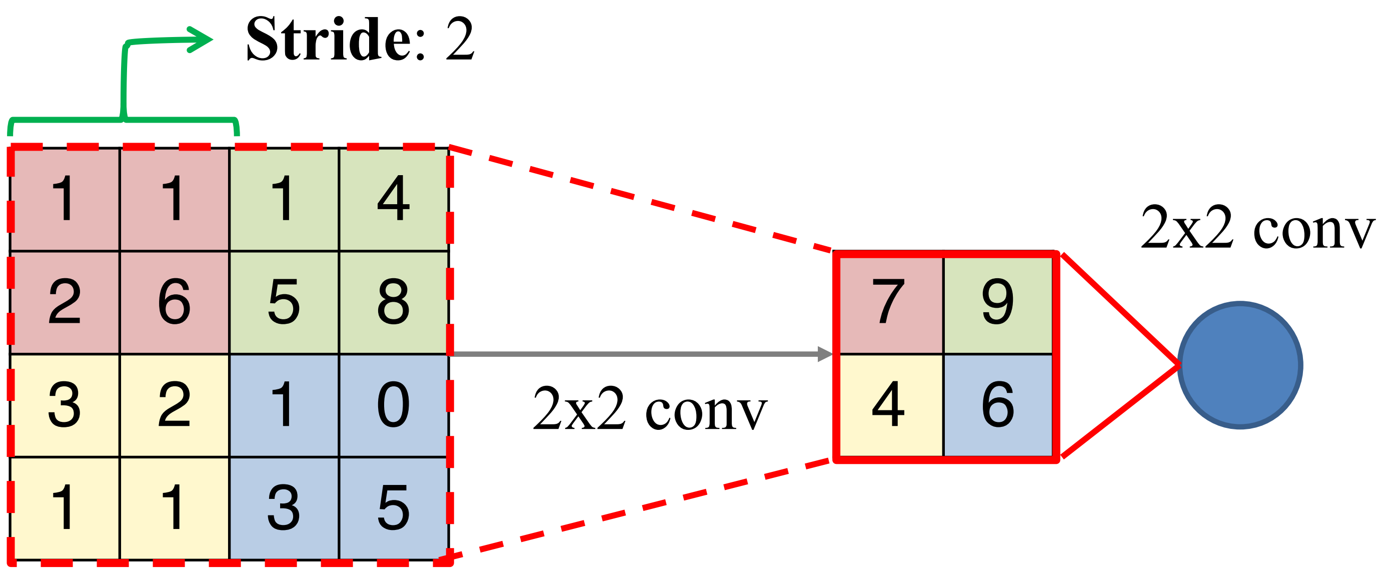
Applying it to a 1D case:



- If we stack N convolutional layers with the same kernel size 3x3 the receptive field on N-th layer will be $\mathbf{2N+1 \times 2N+1}$.
  2N+1 x 2N+1. So for N=100, the receptive field on Nth layer will be (201, 201)
- It looks like we need to stack a lot of convolutional layers!
  To be able to identify objects as big as the input image **300x300** we will need **150** convolutional layers!

We need to grow receptive field faster!

We can increase a **stride** in our convolutional layer to reduce the output dimensions!



Further convolutions will effectively **double** their receptive field!

How do we maintain translation invariance?

recall, the backslash was translated diagonally, yet still has the same sort of 'output'.
We may be afraid that with a larger stride length, we may not have translation invariance.

$$\underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{Input}} * \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}}_{\text{Kernel}} = \underbrace{\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{bmatrix}}_{\text{Output}}$$

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}}_{\text{Input}} * \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}}_{\text{Kernel}} = \underbrace{\begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}}_{\text{Output}}$$

Max = 2

Didn’t change

Max = 2

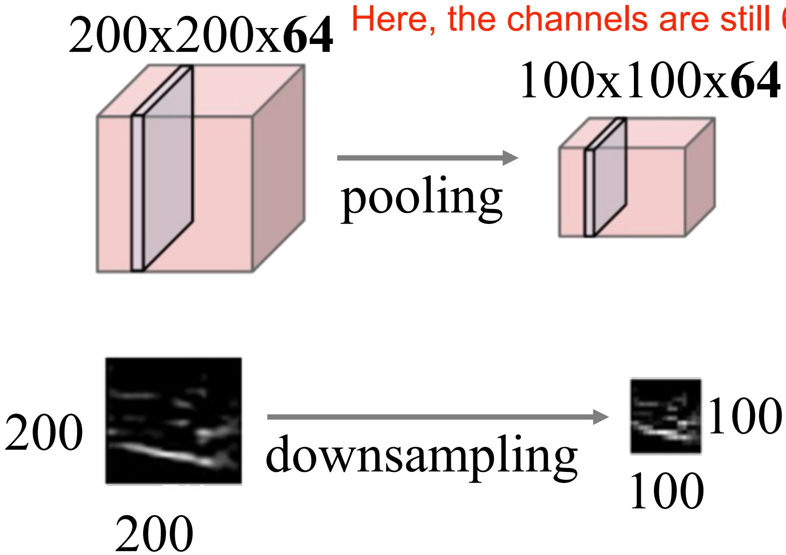
Pooling layer will help!

HELP TRANSLATION INVARIANCE.

This layer works like a convolutional layer but doesn’t have kernel, instead it calculates **maximum** or **average** of input patch values.

AHA. 'pooling'.

Notice we don't change the OUTPUT channels.
Here, the channels are still 64.



Single depth slice

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

→

6	8
3	5

2x2 **max pooling** with stride 2

We apply pooling depthwise.

Backpropagation for max pooling layer

Strictly speaking: maximum is not a differentiable function!

6	8
3	5

→ Maximum = 8

7	8
3	5

→ Maximum = 8

There is no gradient with respect to non maximum patch neurons, since changing them slightly does not affect the output.

6	8
3	5

→ Maximum = 8

7	9
3	5

→ Maximum = 9

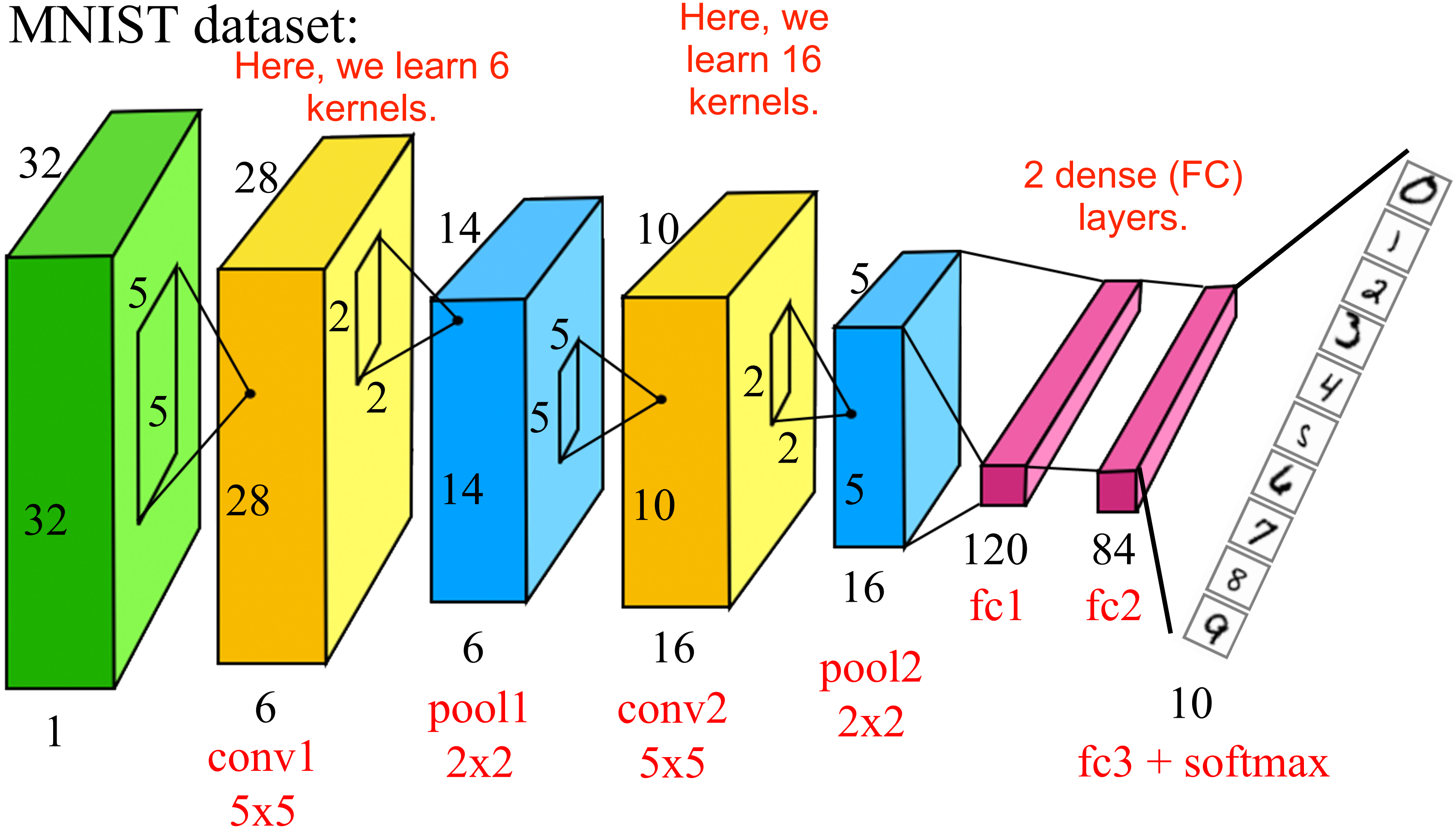
For the maximum patch neuron we have a gradient of 1.

This is a heuristic to make it work.

Putting it all together into a simple CNN

For a kernel of dimension k x k, stride 1, applied to some NxN image, we get N - (k - 1) positions.

LeNet-5 architecture (1998) for handwritten digits recognition on MNIST dataset:



Resolutions:

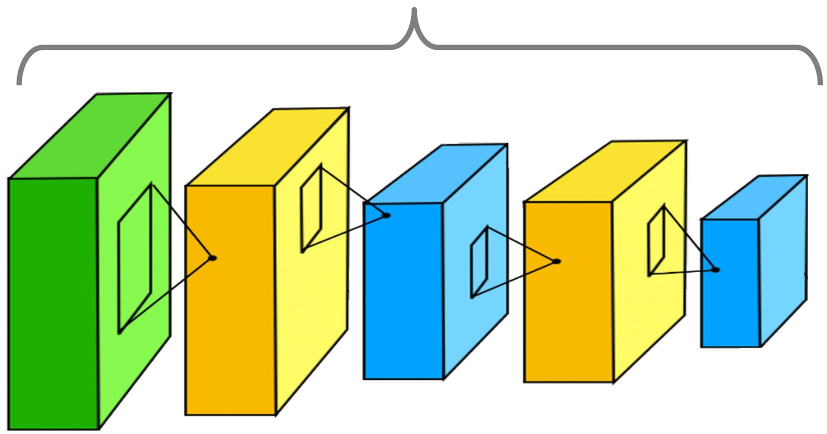
32x32x1 28x28x6 14x14x6 10x10x16 5x5x16

http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf

Learning deep representations

Neurons of deep convolutional layers learn complex representations that can be used as features for classification with MLP. Conv and pooling can be thought of as an automated feature extractor.

Automatic feature extraction



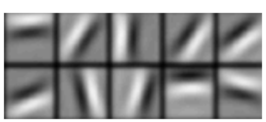
Good features for MLP

Inputs that provide highest activations:

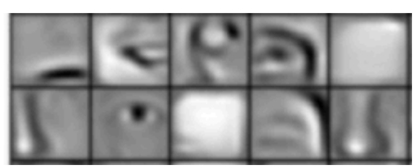
second conv2 learns human features

Conv3 learns full faces.

first conv1 learns edges (low level features)



conv1



conv2

conv3

http://web.eecs.umich.edu/~honglak/icml09-ConvolutionalDeepBeliefNetworks.pdf

Amazing

Summary

- Using convolutional, pooling and fully connected layers we've built our first network for handwritten digits recognition!
- In the next video we'll overview tips and tricks that are utilized in modern neural network architectures.