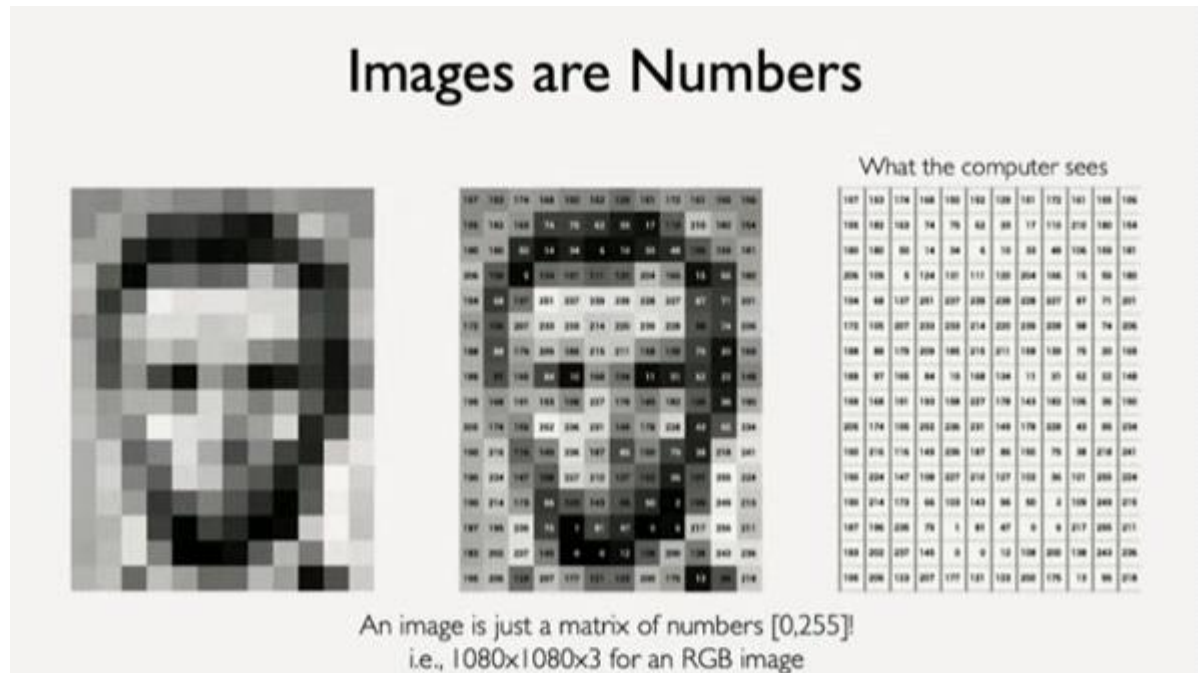


Intro to deep learning 3

Wednesday, August 20, 2025 12:46 PM

Lecture 3: Convolutional neural network - deep computer vision

How can a computer process an image? Images are simply array of numbers in 2D (times 3 for colors). We will focus on **regression** (to output a continuous value) and **classification** (to output a class label through a set of probabilities) based on images input.



We need models able to distinguish unique features. This is the task of **features detection**

High Level Feature Detection

Let's identify key features in each image category



Nose,
Eyes,
Mouth



Wheels,
License Plate,
Headlights



Door,
Windows,
Steps

Defining features by hand is basically impossible because of variations in features, angle of view, ...
Our models need to be invariant to those modifications in the feature space.

Manual Feature Extraction



ML tries to define the features while DL tries to detect them starting from data to defining the features implicitly and not manually

Learning Feature Representations

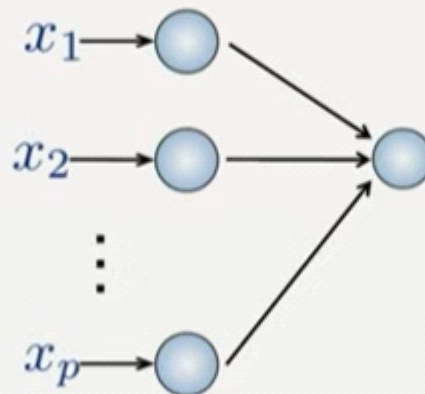
Can we learn a **hierarchy of features** directly from the data instead of hand engineering?



If we wanted to use the fully connected NN from lecture 1, we can do it as follows. But we have problems. For example the 2D input must be flattened and the spatial information is lost. Because of the full connection between layers, there is also a LOT of parameters

Fully Connected Neural Network

- Input:**
- 2D image
 - Vector of pixel values

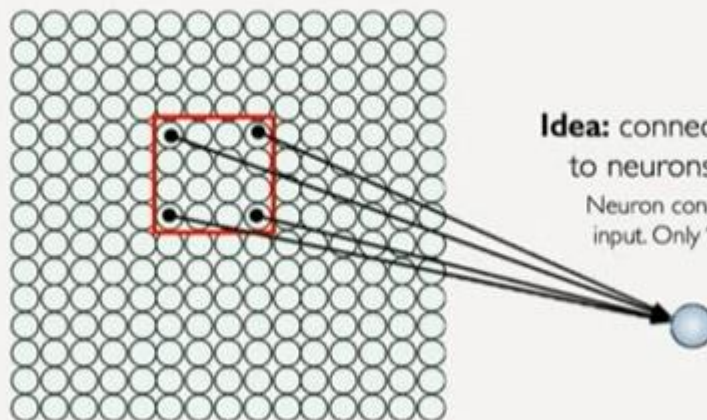


- Fully Connected:**
- Connect neuron in hidden layer to all neurons in input layer
 - No spatial information!
 - And many, many parameters!

We want instead to preserve spatial information native to the data to inform the architecture. We represent the 2D image as 2D array of pixels. We connect patches of pixels to neurons in a sliding fashion.

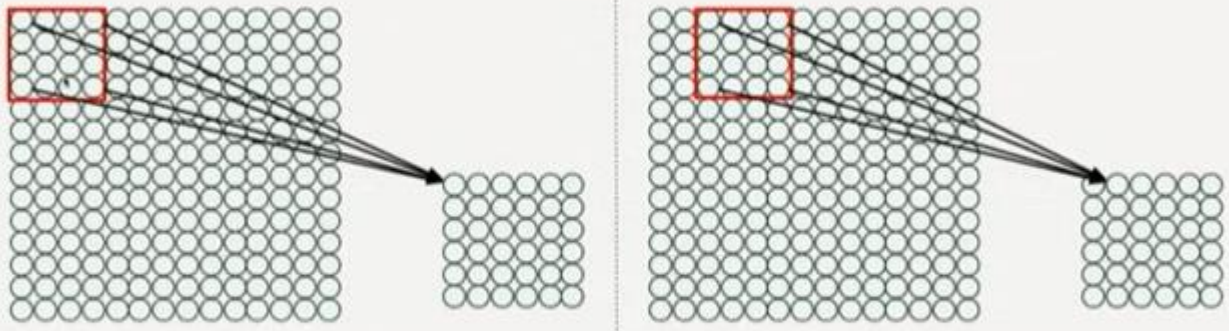
Using Spatial Structure

Input: 2D image.
Array of pixel values



Idea: connect patches of input to neurons in hidden layer.
Neuron connected to region of input. Only "sees" these values.

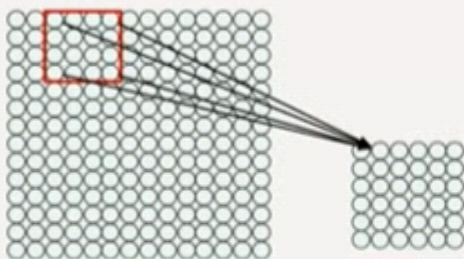
Using Spatial Structure



Connect patch in input layer to a single neuron in subsequent layer.
Use a sliding window to define connections.
How can we **weight** the patch to detect particular features?

Features are extracted with convolution and each patch is multiplied pointwise by a patch-size set of weights called "filter" and then sum everything.

Feature Extraction with Convolution



- Filter of size 4x4 : 16 different weights
- Apply this same filter to 4x4 patches in input
- Shift by 2 pixels for next patch

This "patchy" operation is **convolution**

- 1) Apply a set of weights – a filter – to extract **local features**
- 2) Use **multiple filters** to extract different features
- 3) **Spatially share** parameters of each filter

How is this convolution able to define the feature information?

Example:

X or X?

?

=

=

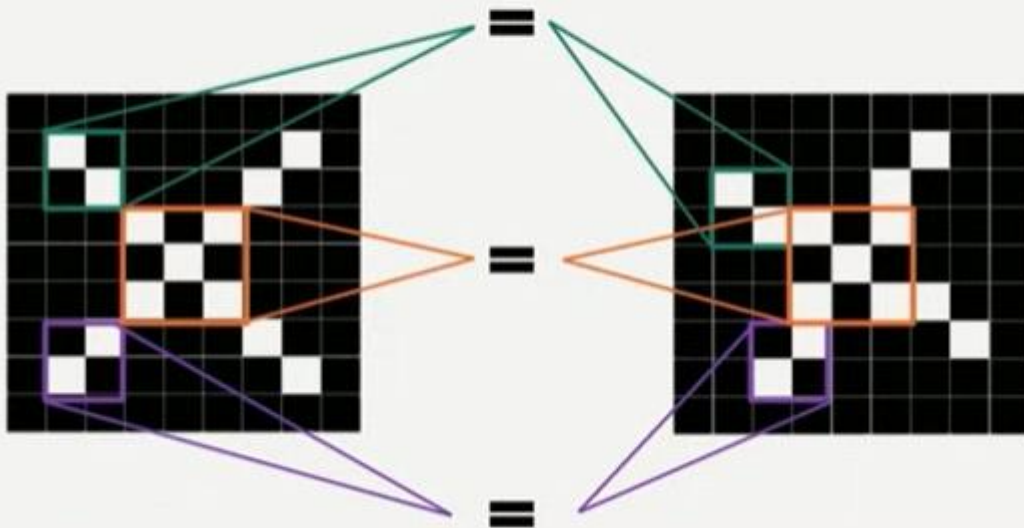
-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

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

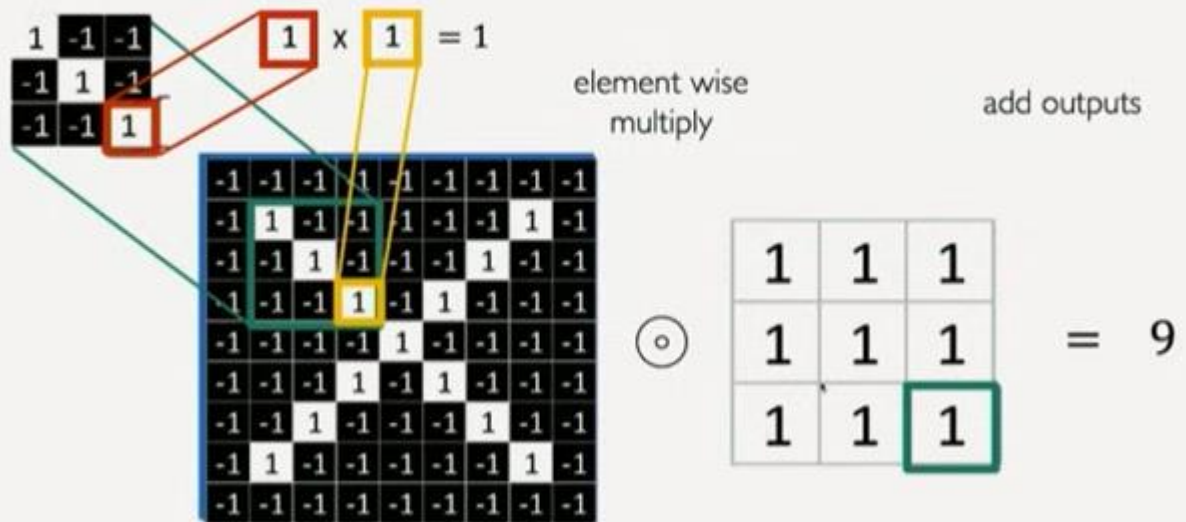
Image is represented as matrix of pixel values... and computers are literal!
We want to be able to classify an X as an X even if it's shifted, shrunk, rotated, deformed.

A patchwise comparison seems already a better alternative than just looking at the whole picture for comparison

Features of X



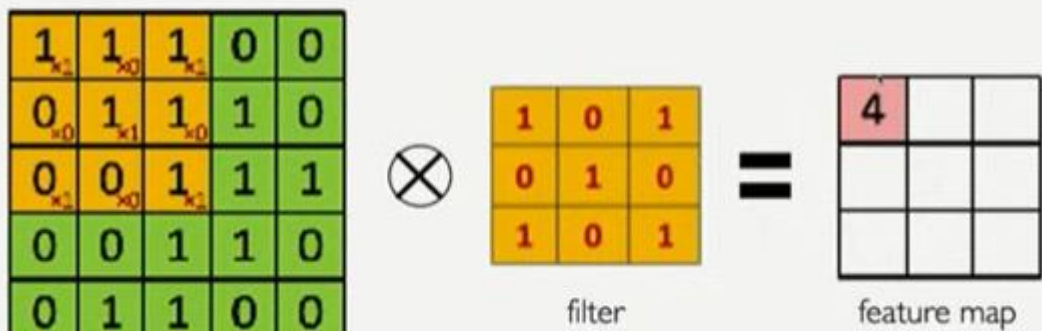
The Convolution Operation



Another example of convolution for fixing the idea

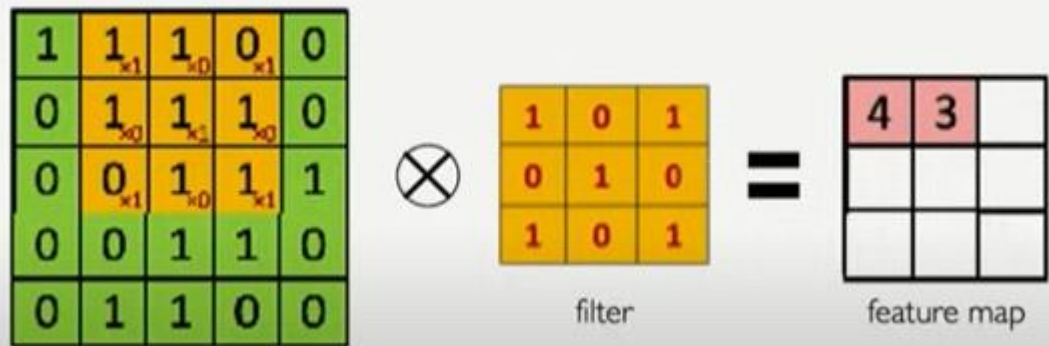
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



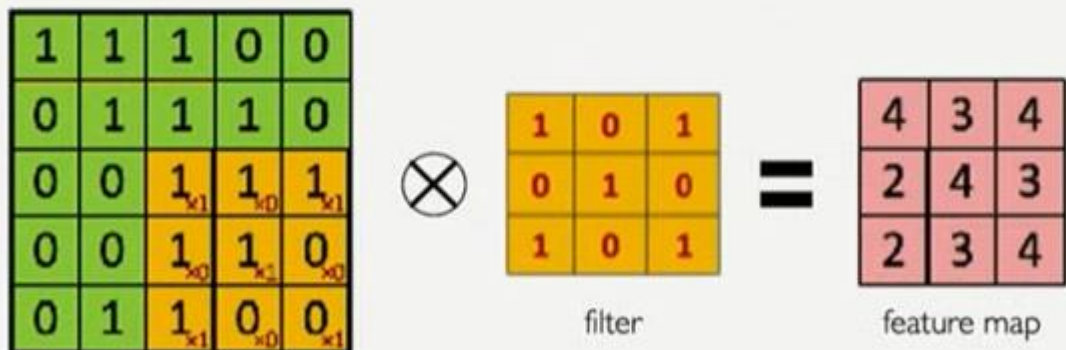
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



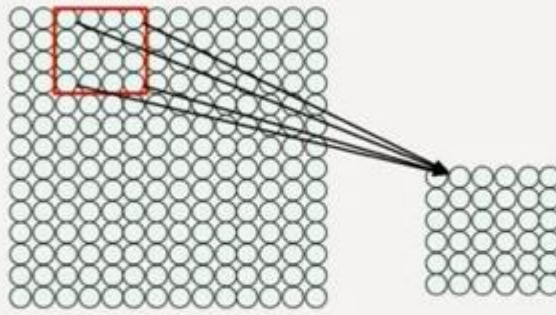
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



Which filter you choose is fundamental for the extraction of features

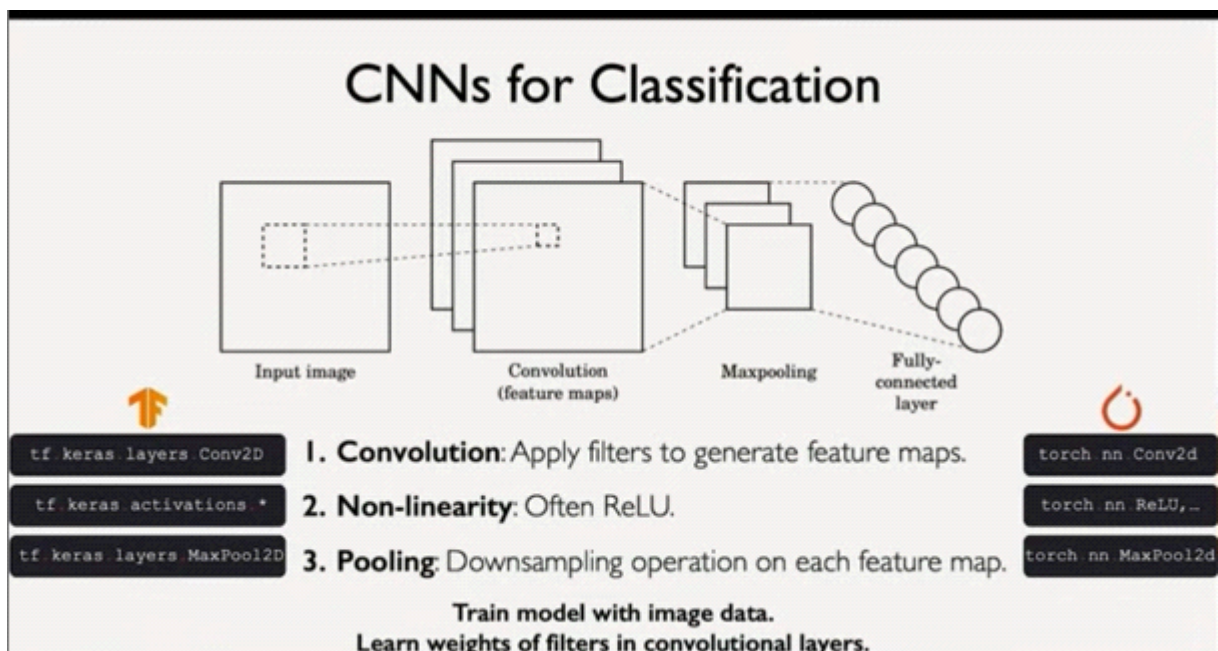
Feature Extraction with Convolution



- 1) Apply a set of weights – a filter – to extract **local features**
- 2) Use **multiple filters** to extract different features
- 3) **Spatially share** parameters of each filter

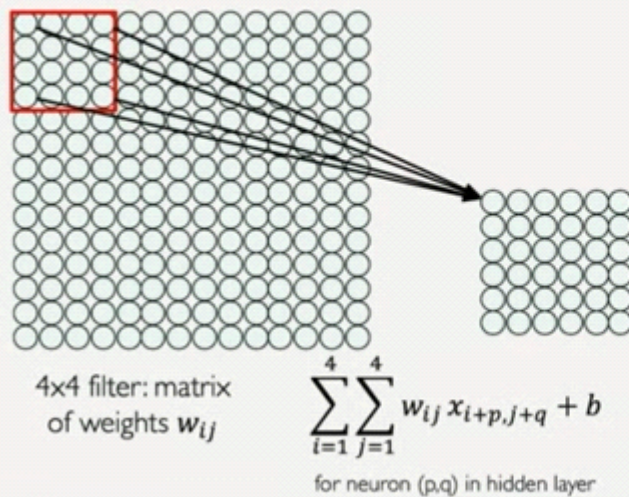
Convolutional Neural Networks


In CNNs we have the convolutions happening on the patches of an image with the filter. Then we do pooling downsampling the feature maps from convolutions.



More in depth in the convolution note each pixel sees only its patch in the operation.

Convolutional Layers: Local Connectivity



 `tf.keras.layers.Conv2D`

 `torch.nn.Conv2d`

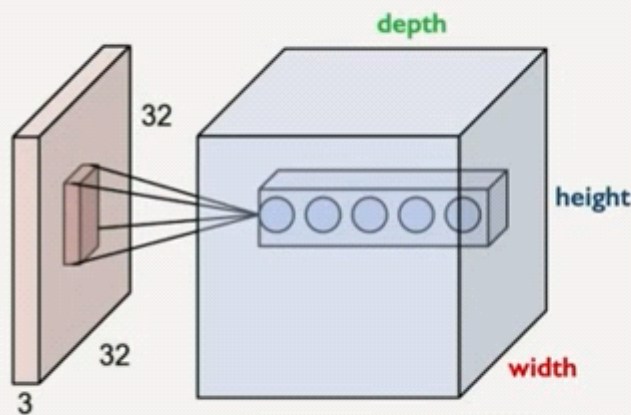
For a neuron in hidden layer:

- Take inputs from patch
- Compute weighted sum
- Apply bias

- 1) applying a window of weights
- 2) computing linear combinations
- 3) activating with non-linear function

Now, when you do convolution, you do not have only ONE filter, but you have multiple ones, so the feature extraction gives a VOLUME of features.

CNNs: Spatial Arrangement of Output Volume



Layer Dimensions:

$h \times w \times d$


where h and w are spatial dimensions
d (depth) = number of filters


Stride:

Filter step size

Receptive Field:

Locations in input image that
a node is path connected to

 `tf.keras.layers.Conv2D(filters=d, kernel_size=(h,w), strides=s)`

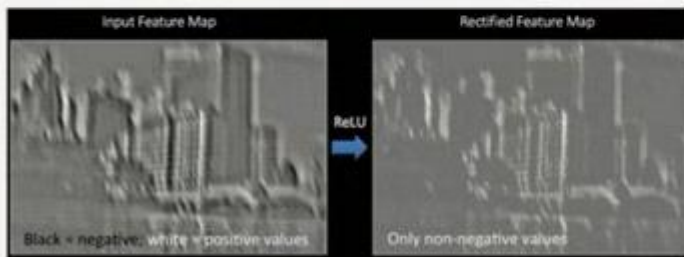
 `torch.nn.Conv2d(in_channels=3, out_channels=d, kernel_size=(h,w), stride=s)`

*Need to specify
input dimensionality!*

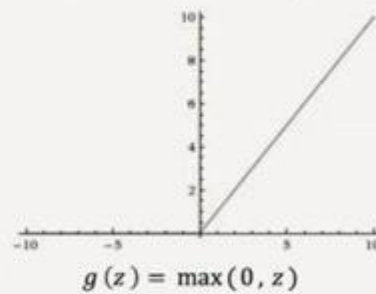
After convolution we need to apply a ReLU (linear rectifier) which puts to 0 all negative values. It works like a threshold and is used in computer vision. Remember it is applied on the feature maps.

Introducing Non-Linearity

- Apply after every convolution operation (i.e., after convolutional layers)
- ReLU: pixel-by-pixel operation that replaces all negative values by zero. **Non-linear operation!**



Rectified Linear Unit (ReLU)



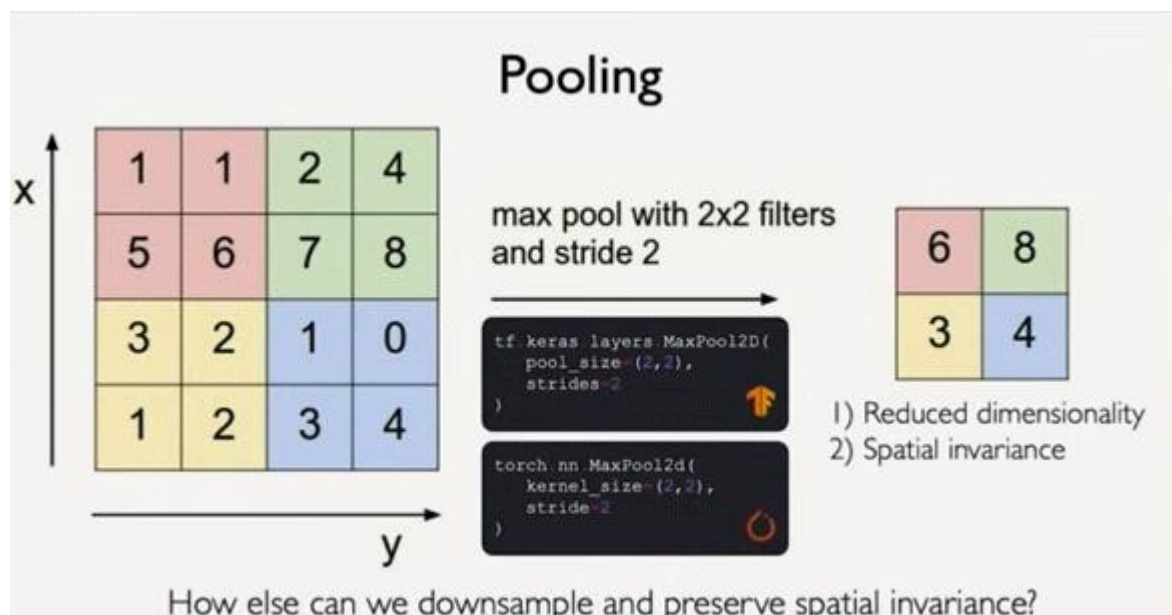
`tf.keras.layers.ReLU`

`torch.nn.ReLU`

Note from questions: pooling is used because you stack multiple convolutions and you are able to detect features at different scales, which you then combine when you apply your network to an input after training.

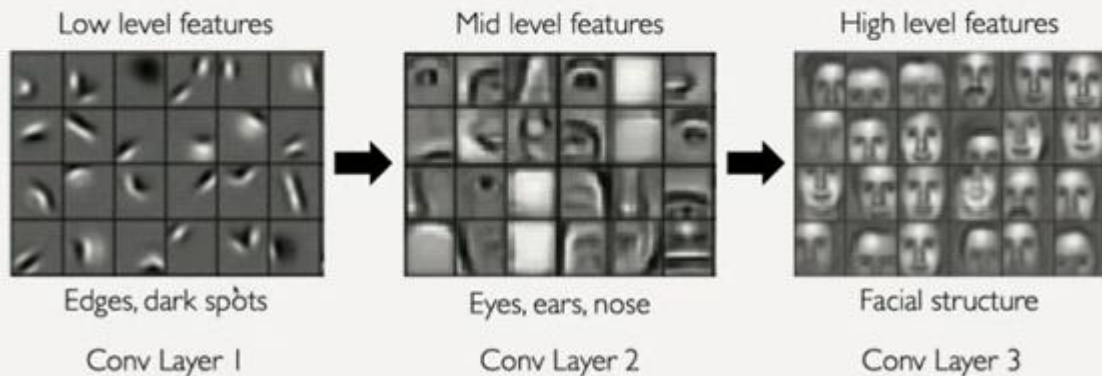
Pooling

Often pooling is done with max pooling taking the max in 2x2 filters because it can change a gradient a lot in optimization (mean pooling would make very stable gradients).



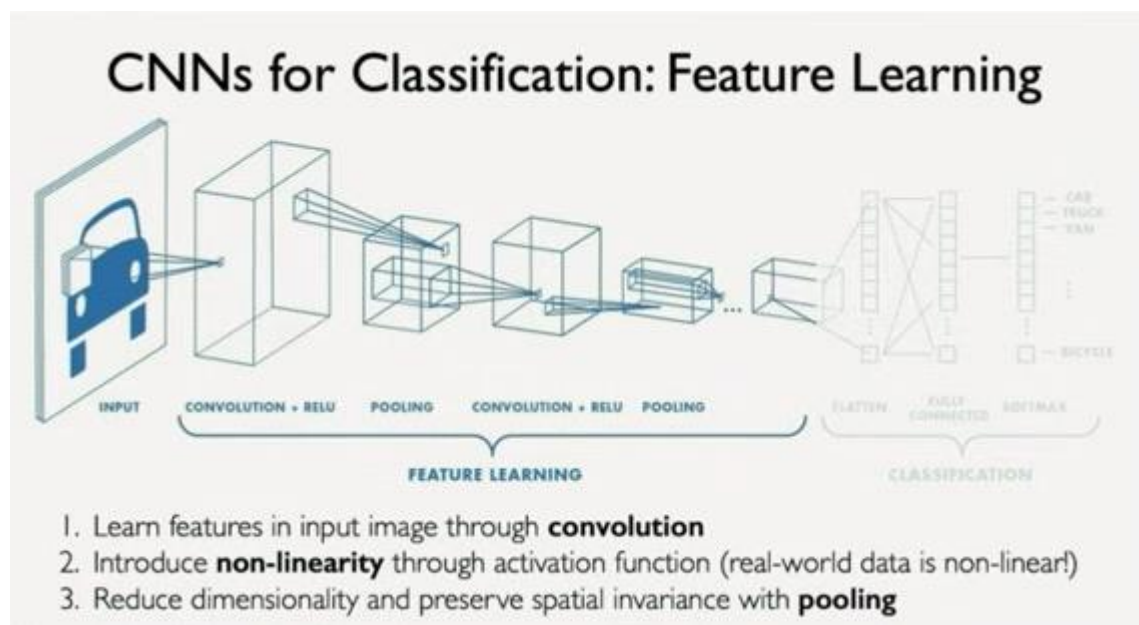
In the end you learn the features without defining them and those features will act at different scales.

Representation Learning in Deep CNNs



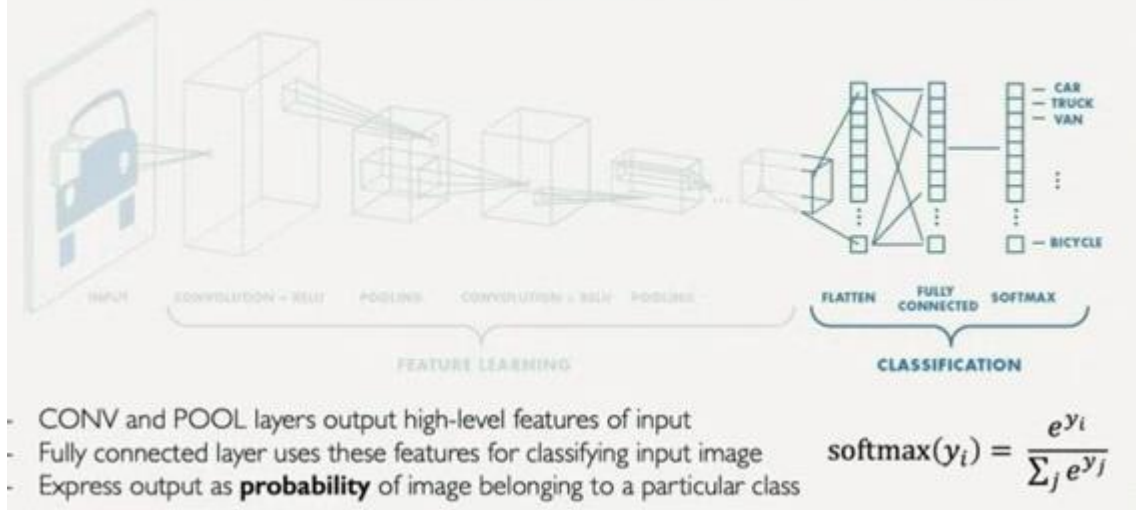
Classification

Now we handled the feature learning



We need to then classify. We use a softmax function to create a probability distribution on the flatten output of the feature extractions going through fully connected layers.

CNNs for Classification: Class Probabilities



How to choose features? Start quite small, but upsample through layers). Also, depending on image size, some resolutions might not make sense for too few pixels, so it depends on the image size, but also on the application type.

```
import torch

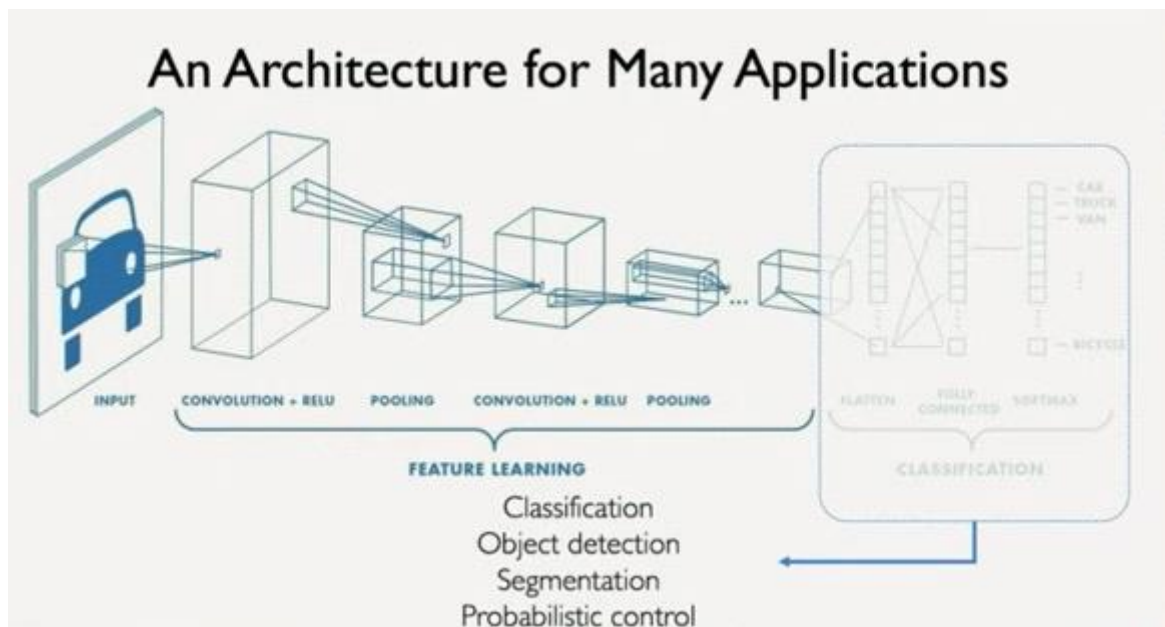
def generate_model():
    model = nn.Sequential([
        # first and second convolutional layer
        torch.nn.Conv2d(in_channels=3, out_channels=32, filter_size=3),
        torch.nn.ReLU(),
        torch.nn.MaxPool2d(kernel_size=2, stride=2),

        torch.nn.Conv2d(in_channels=32, out_channels=64, filter_size=3),
        torch.nn.ReLU(),
        torch.nn.MaxPool2d(kernel_size=2, stride=2),

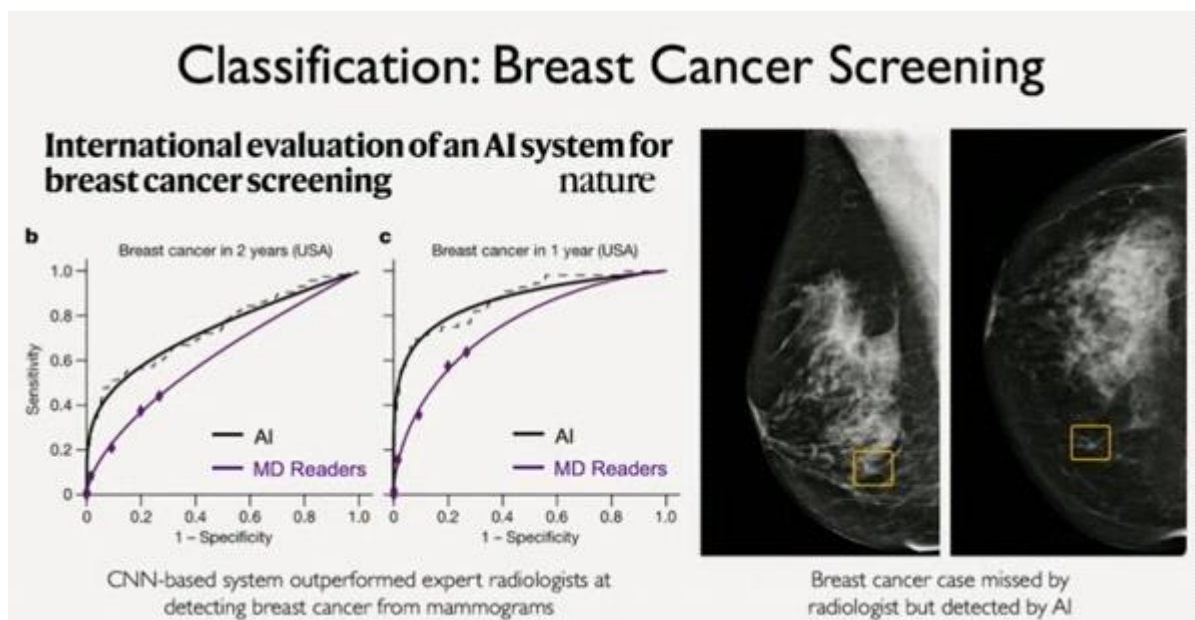
        # fully connected classifier
        torch.nn.Flatten(),
        torch.nn.Linear(64*6*6, 1024), # flattened dim after 2 conv layers
        torch.nn.ReLU(),
        torch.nn.Linear(1024, 10), # 10 outputs
    ])
    return model
```

Possible convolutional applications

Now, the convolutional part of this network can be applied to many tasks beyond classification

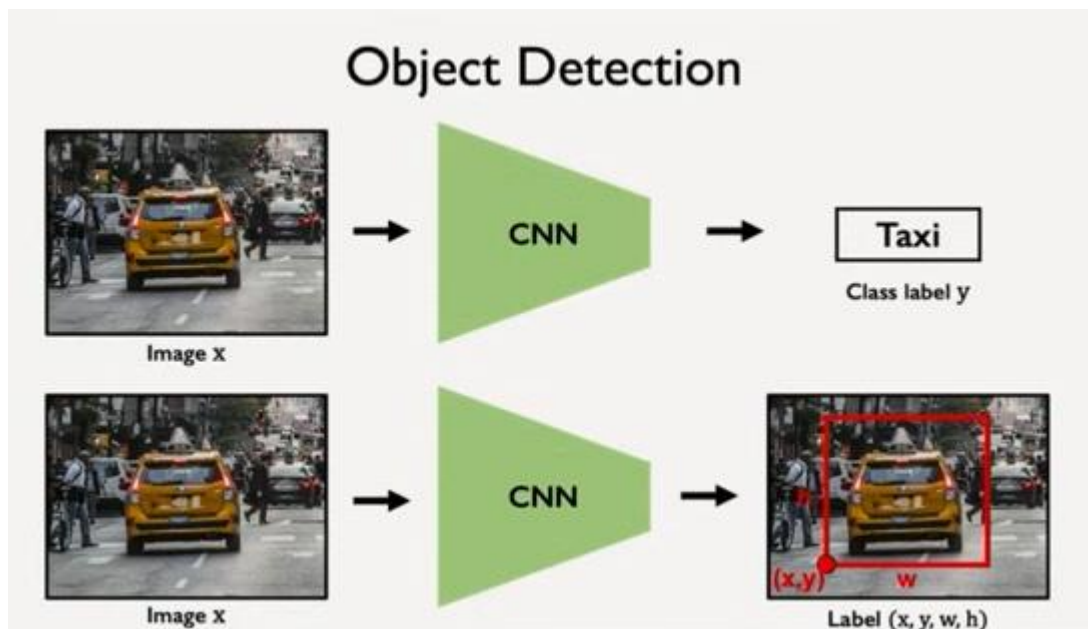


Example



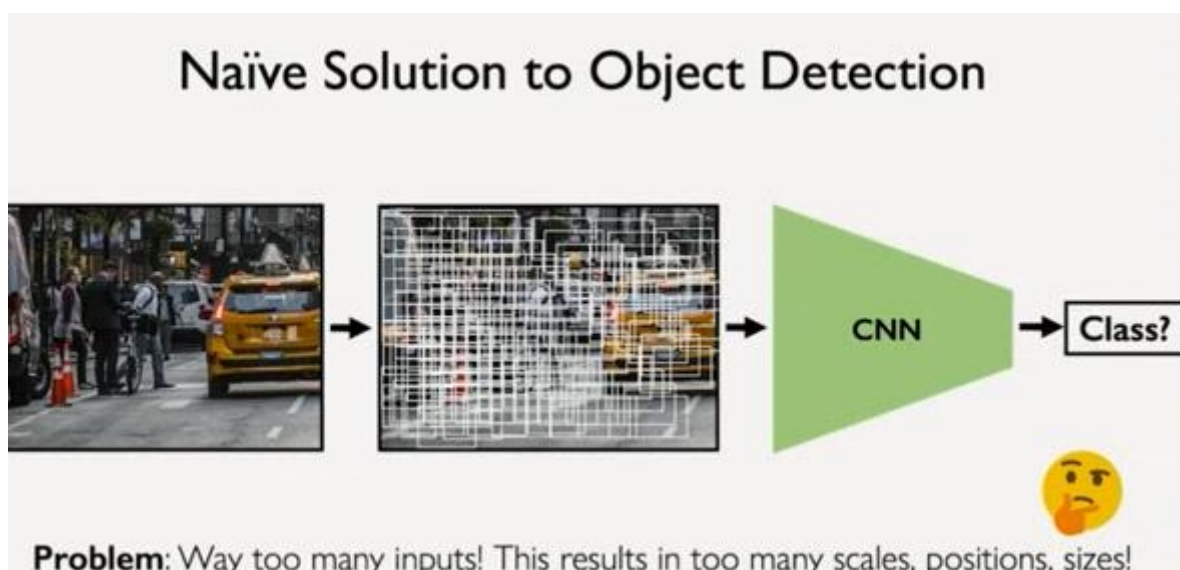
Example

Object detection predicting not only an object, but location, class and bounding boxes. The same for every other object.



Object detection

Look at a naïve solution working on random boxes through CNNs and keep boxes with classes

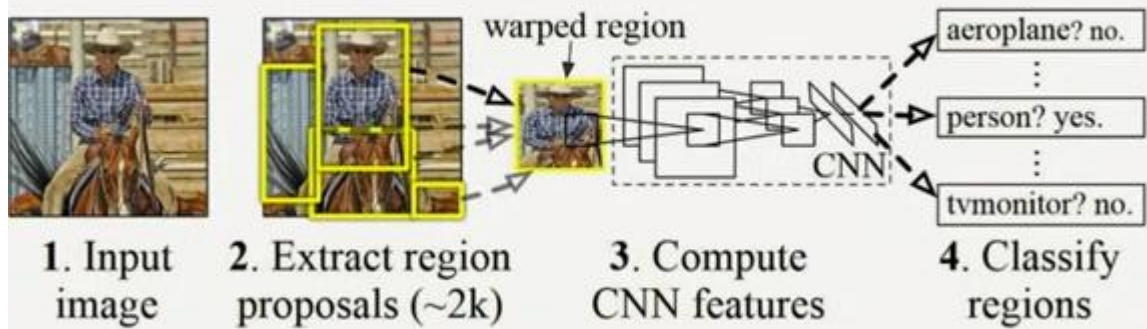


R-CNN

You can use heuristic suggesting regions, but it disconnects region proposal to the rest of the model and it is slow

Object Detection with R-CNNs

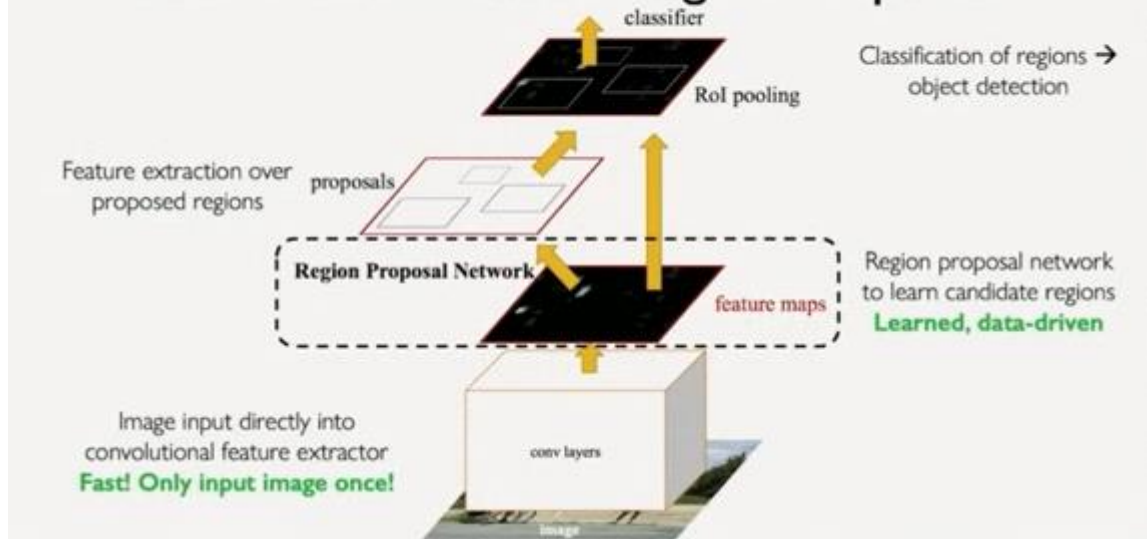
R-CNN algorithm: Find regions that we think have objects. Use CNN to classify.



Problems: 1) Slow! Many regions; time intensive inference.
2) Brittle! Manually defined region proposals.

Faster R-CNN learns those regions within the same network as the classifier, so feature extraction happens on those regions

Faster R-CNN Learns Region Proposals

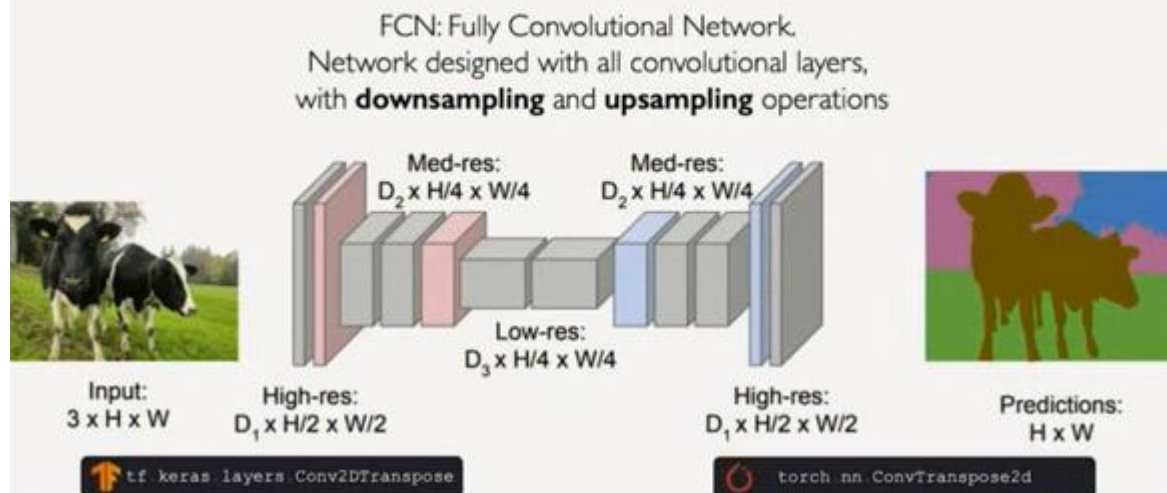


Segmentation

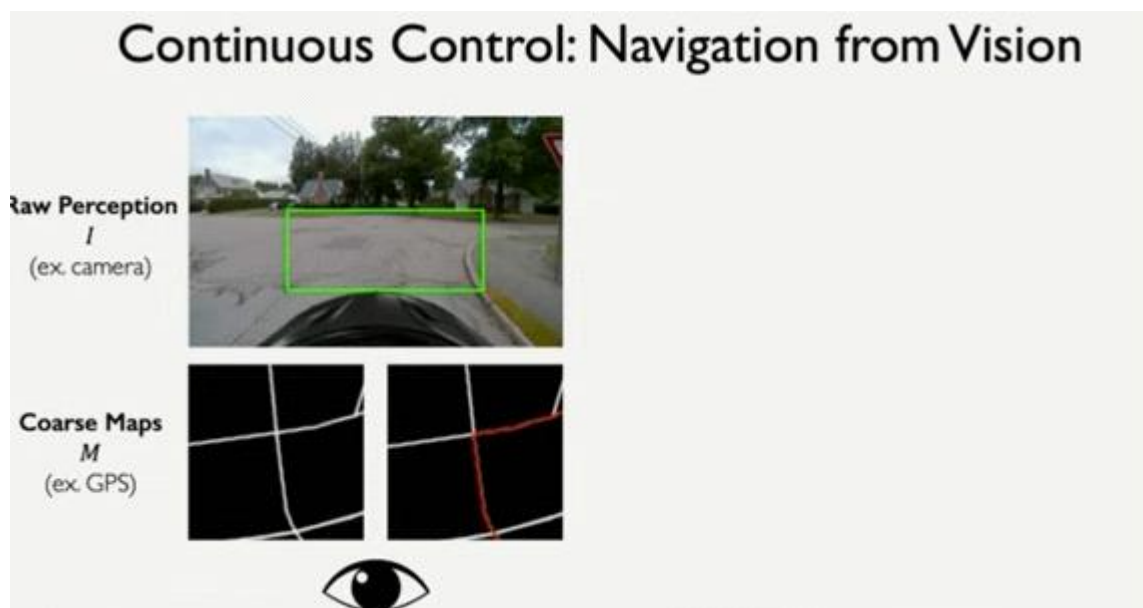
For every single pixel we want to learn another picture classifying every single pixel.

On the left we have convolution, pooling and non-linearity. On the right we have convolutions sampling up to the full image size again.

Semantic Segmentation: Fully Convolutional Networks



Automatic driving




We want to learn a continuous prob distribution of all possible control commands

Continuous Control: Navigation from Vision

Raw Perception I
(ex. camera)

Coarse Maps M
(ex. GPS)

Possible Control Commands



The diagram illustrates the process of navigation from vision. It shows two input components on the left: 'Raw Perception I (ex. camera)' and 'Coarse Maps M (ex. GPS)'. These inputs are processed to generate 'Possible Control Commands', which is shown as a camera view of a road intersection with red lines indicating potential paths. An eye icon is positioned below the input components, and a steering wheel icon is positioned below the output component. A large blue arrow points from the inputs to the output.

End-to-End Framework for Autonomous Navigation

Entire model is trained end-to-end **without any human labelling or annotations**

Diagram illustrating the End-to-End Framework for Autonomous Navigation. The framework processes four inputs (three camera views and a map) through a series of convolutional, fusion, and fully connected (fc) layers to produce a Routed Map. The Routed Map is then processed by a conv layer, followed by a fusion layer, and then a concatenate layer, leading to a final fc layer. The final output is a Probabilistic Control Output, which is a map showing a red path. The Probabilistic Control Output is also used to calculate the Deterministic Control, which is a map showing a red path. The Deterministic Control is also used to calculate the Optional output, which is a map showing a red path.

Probabilistic Control Output

$$P(\theta|I, M) = \sum_{i=1}^R \phi_i \mathcal{N}(\mu_i, \sigma_i^2)$$

Deterministic Control

Optional output if routed map is provided as input

$$L = -\log(P(\theta|I, M))$$

$$L = -\log(P(\theta|I, M))$$