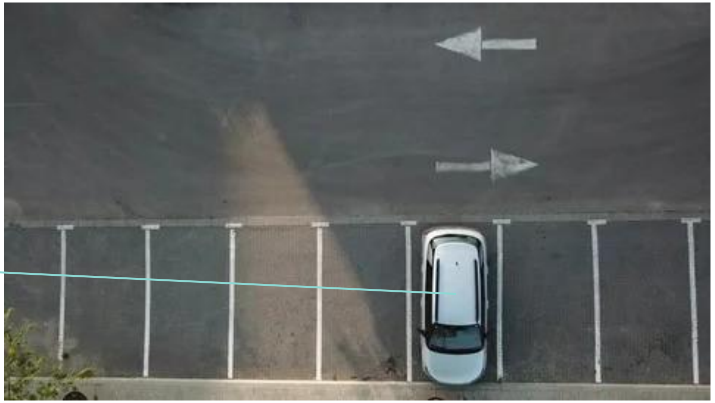
Convolutional neural networks

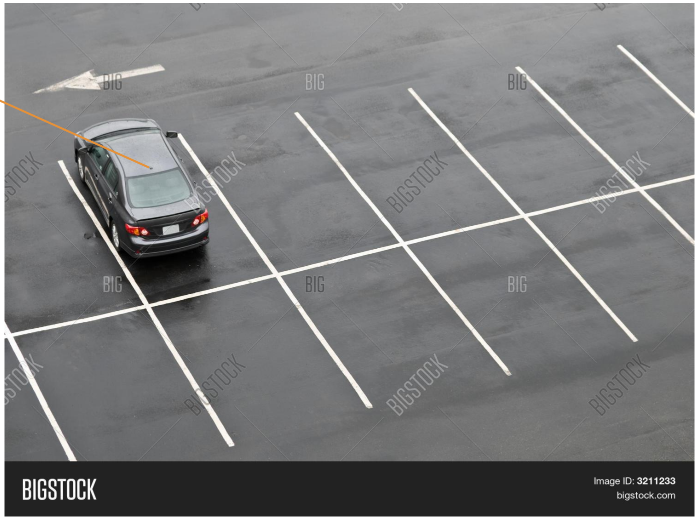
Linear regression on images

- We have one parameter for each pixel of the image
- We want to find out if there's a car in the image: the larger the output value of the model, the higher the chance the image contains a car
- For the image to the left, the bottom right parameters will specialize to detect the car
- The rest of the parameters will end up as zeroes so they don't affect the output



Contradictions

- Here, the top-left parameters will try to specialize such that they recognize the car, and the rest of the parameters will likely end up as zeroes
- If we want our model to work on both this and the previous image, we end up with contradicting goals: top left parameters should be zeroes for the first image, and should specialize in recognizing the car for the second image
- **Inefficient solution**: do multiple linear regressions, where one will detect cars in the upper left corner, other in the bottom right and so on



Convolution = local linear regression

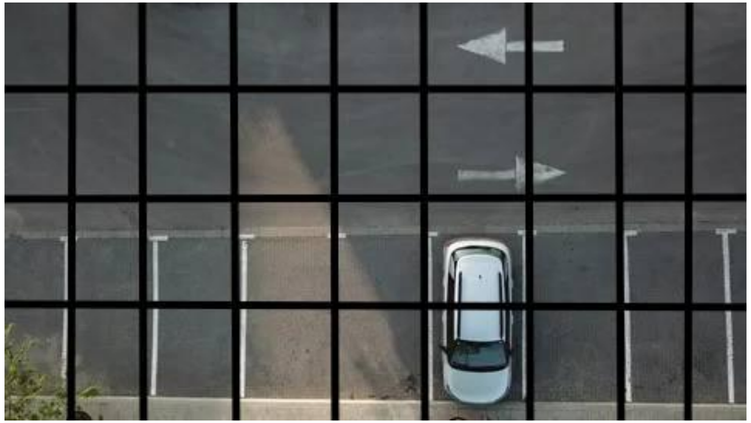
Locality principle

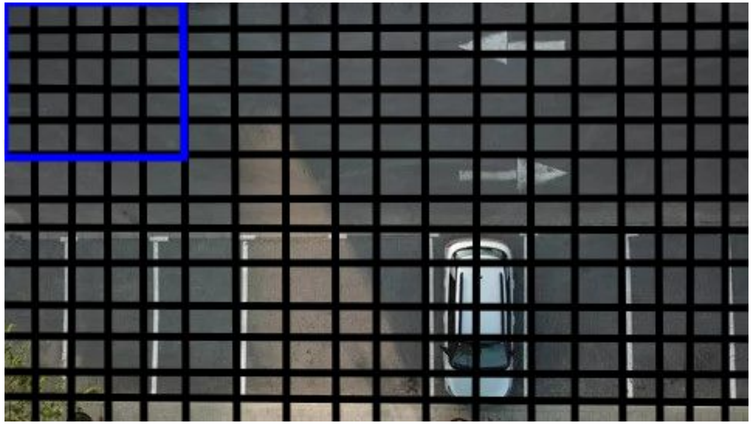
- Split image into small patches
- Apply a linear regression to each patch
- As many outputs as there are patches
- **Advantages**: far less parameters, because we use the same linear regression for each patch
- **Advantages**: we can detect the car even if it's in another part of the image

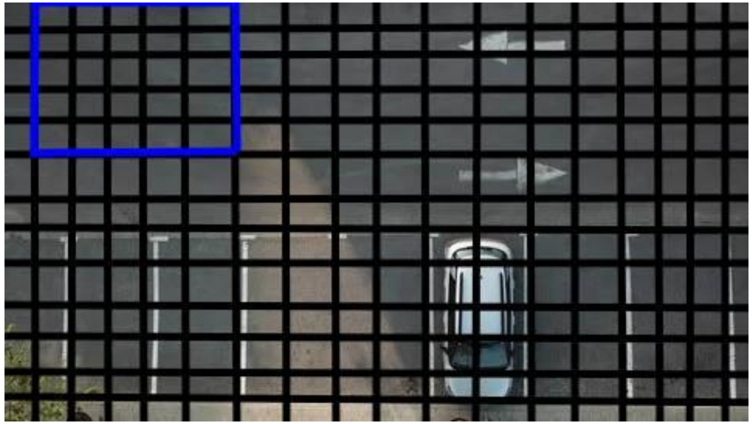
Translation invariance
Translation = moving the car to another part of the image
Invariance = the output doesn't change

Object split in half by gridlines

- If we choose the patch size incorrectly, the object of interest may end up cut in half by the grids
- **Solution**: we apply the linear regression between gridlines

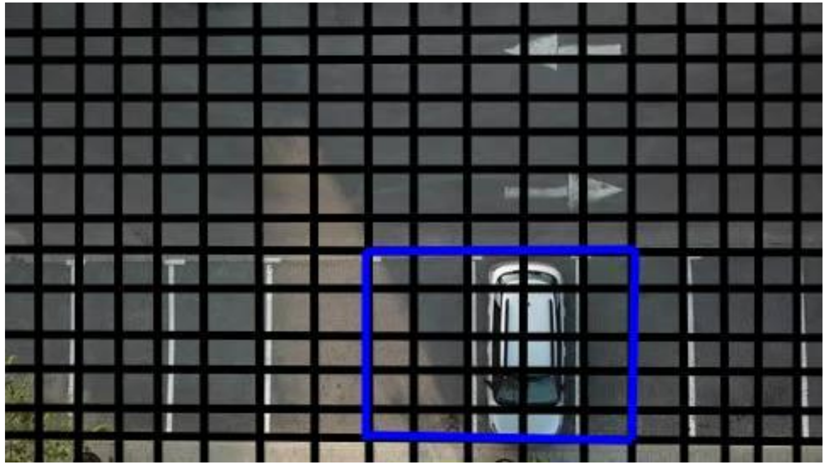






Apply linear regression to top left patch (a 5x5 of “squares”)

Move one “line of squares” to the right and apply linear regression to the 5x5 patch



Eventually we end up in a 5x5 that contains the car

In PyTorch

Will discuss when talking about
multiple input and output channels

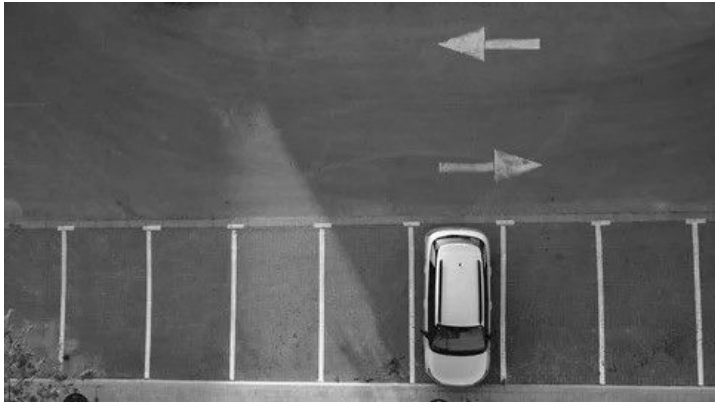
- `model = nn.Conv2d(1, 1, kernel_size = 2)`

We're using 2 x 2 (meaning 4 pixels) **patches**

- `model.weight` and `model.bias` are the parameters of the local linear regression that we mentioned

How powerful is a convolution / local linear regression?

- a convolution can do “simple” image processing tasks such as edge detection
- it can’t detect a whole car by itself

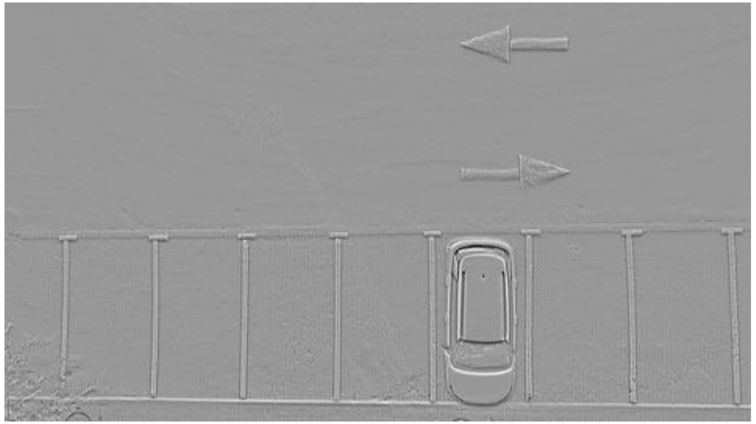


Kernel (parameters of local linear regression):

[[-1, -1, -1]

[-1,   8, -1]

[-1, -1, -1]]



Is one convolution enough?

- **Problem:** not all cars look the same so we may need multiple convolution kernels to detect various types of cars
- **Problem**: one convolution is powerful enough to detect edges, lines, shapes in the image, but usually not powerful enough for a whole car
- **Problem**: the car may be larger than the convolution kernel so maybe the kernel only sees one of the wheels
- We need a way to apply multiple convolution kernels, and a way to compose convolution kernels such that we can detect a car through kernels that detect the wheels, body, doors etc.
- **Note**: the kernel that detects the wheels will likely use kernels that detect parts of the wheel, and those kernels may use kernels that detect simple geometric shapes => **hierarchical approach**

Multiple input / output channels

- If we apply multiple convolution operations to an image, we end up with multiple output “images” => these are called **channels**
- The input can also have multiple channels:
    - input is an RGB image => 3 channels, for red, green and blue
    - input can be the output of a convolution => however many channels the convolution had
- If we have **m** input channels and wish to obtain **n** output channels:
    - step 1: apply a convolution to each of the input channels
    - step 2: add up the results to obtain the first output channel
    - step 3: do step 1 and step 2 until we end up with **n** output channels

Multiple input / output convolutions in PyTorch

- `conv = nn.Conv2d(3, 16, kernel_size = 5)`

Number of input channels

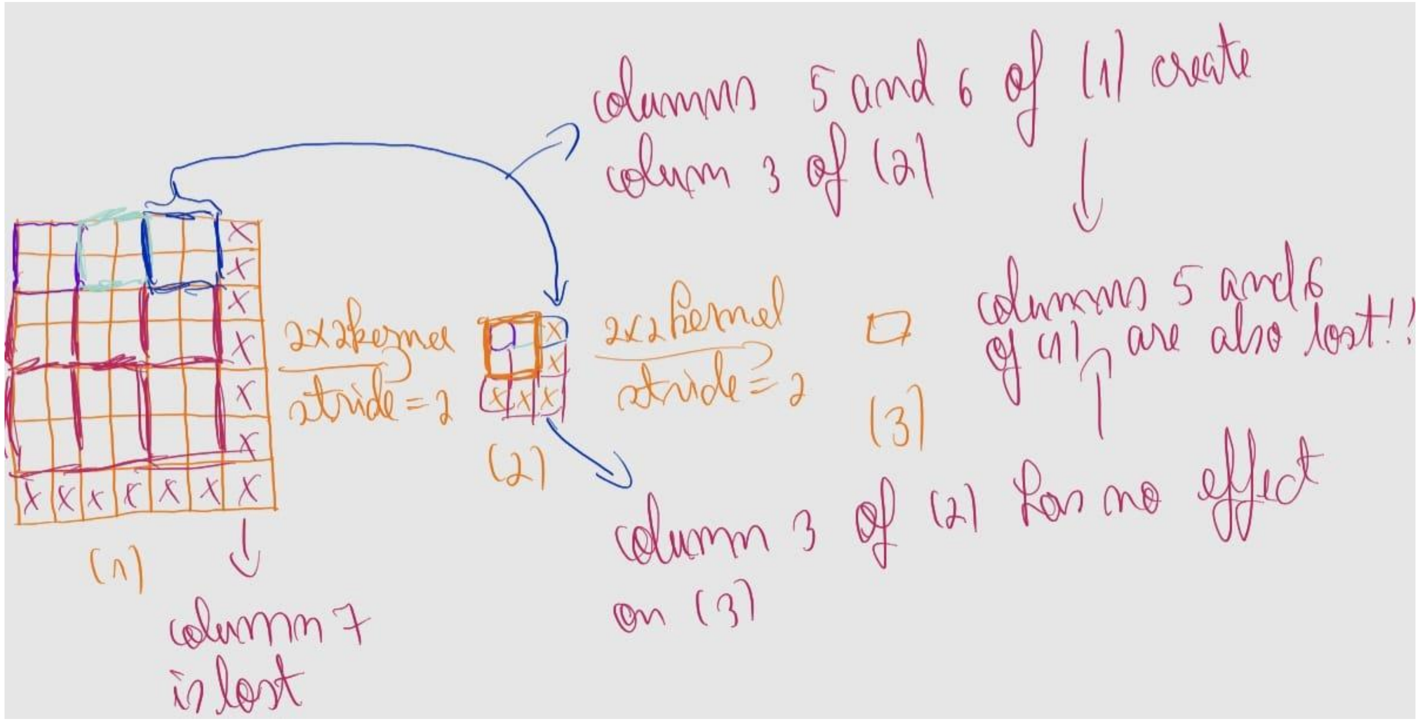
Number of output channels

- `conv.weight.data => 16 x 3 x 5 x 5` tensor: contains 48 kernels of size 5 x 5, one for each pair of input channel - output channel
- `conv.bias.data =>` vector of 16 free terms / biases, one for each output channel

Parameters of a convolution

- `in_channels:` how many input channels do we have (1 for grayscale image, 3 for RGB image, more if the input is the output of another convolution)
- `out_channels:` how many output channels do we have
- `kernel_size`: patch size (i.e. how many pixels do we run the local linear regression on?)
- `padding`: how many lines / columns of zeros we want to add around the input image
- `stride`: how many pixels do we want to jump over?
- *Other parameters (for your own research)*: groups, dilation
- *Other kinds of convolutions (for your own research)*: transposed convolutions, deformable convolutions

Stride, padding and loss of information


columns 5 and 6 of (1) create colum 3 of (2)
2x2 kernel stride = 2
2x2 kernel stride = 2
(1)
(2)
(3)
column 7 is lost
columns 5 and 6 of (1) are also lost!!
column 3 of (2) has no effect on (3)

Overall convolution in PyTorch

```
nn.Conv2d(
    in_channels = 16,
    out_channels = 32,
    kernel_size = 3,
    padding = 1,
    stride = 2
)
```

usually an odd number, most often 3
even number => need padding only on left or right side, not both; convenient to add padding on both sides

1 row/column of zeros on every side of the image
if kernel_size is odd, this is equal to `kernel_size // 2` to keep same input/output shape

default is 1, meaning we don't jump over pixels
stride = 2 means we jump over pixels, reducing image resolution to half

Making things more “global” with pooling operations

- we want to answer the question: is there a car in this image?
- we have looked at a bunch of patches of the image with a local linear regression / convolution
- pooling “summarizes” the results for a bunch of the patches
- functions exactly like a convolution
- apply a summarization operation instead of a local linear regression operation
- usually have `stride` equal to `kernel_size` => output resolution is input resolution divided by `kernel_size`

Pooling in PyTorch

- average pooling: replaces each patch with its mean value
- PyTorch: `nn.AvgPool2d(kernel_size = 3, padding = 1, stride = 3)`

PyTorch uses `kernel_size` for the stride if we don't specify it

- max pooling: replaces each patch with its largest value
- PyTorch: `nn.MaxPool2d(kernel_size = 3, padding = 1, stride = 3)`
- a `2x2` pooling layer followed by a `2x2` convolution layer: each output of the convolution uses a `4x4` of values from the input of the pooling layer => pooling helps increase receptive field of the convolutions that follow it!

CNN for classification: recipe

Convolutional neural network

- convolutions and pooling layers: generally reduce resolution and increase number of channels
- an `nn.Flatten()` layer to transform the “image” to a vector
- linear layers that output as many values as there are classes
- don’t forget activation functions!

LeNet convolutional neural network

Convolutional part
In literature: **feature extractor**

Flatten layer: go from “image” to vector of “pixels”

Linear part
In literature: **classification head / MLP head**

```
net = nn.Sequential(
      nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.Sigmoid(),
      nn.AvgPool2d(kernel_size=2, stride=2),
      nn.Conv2d(6, 16, kernel_size=5), nn.Sigmoid(),
      nn.AvgPool2d(kernel_size=2, stride=2),
      nn.Flatten(),
      nn.Linear(16 * 5 * 5, 120), nn.Sigmoid(),
      nn.Linear(120, 84), nn.Sigmoid(),
      nn.Linear(84, 10))
```

Normalization

- models based on linear regression work better when all inputs have roughly the same scale
- we can force model inputs to have a similar scale, for example by subtracting the mean and dividing by the standard deviation
- **after first layer: scale of data may not be the same anymore!**
- must apply normalization **inside the network**!

Batch normalization

`Mean` becomes `0`, `std` becomes `1`:
Values end up roughly in `[-2, 2]`

Applied at once to all images in the mini-batch

$$x \leftarrow [\frac{x - \mathbf{mean}(x)}{\mathbf{std}(x)}] \cdot \gamma + \beta$$

The new `mean`, decided by the network

one of the channels of the image**s**

The new `std`, decided by the network: can stay `1`, if the network believes `1` to be a good `std`

Batch Normalization in PyTorch

- `bn = nn.BatchNorm2d(num_features)`

$\swarrow$ number of input channels

- `bn.weight` => tensor containing `num_features` elements, representing the new standard deviations (γ)
- `bn.bias` => tensor containing `num_features` elements, representing the new means (β)

Theory Takeaways

- **convolution** applies a small linear regression at all positions in the image
- **pooling** helps further convolutions see more of the input image without increasing kernel size
- **batch normalization** ensures data stays at the same scale throughout the network; it computes the mean and std separately for every channel, then subtracts the mean and divides by the std
- **recipe for classification**: a bunch of convolutions and pooling layers that **reduce resolution** and **increase channel count**; a flatten layer to go from images to vectors (note, the flatten layer doesn’t flatten the batch dimension, 2 x 3 x 10 x 10 becomes 2 x 300, not 600); finally, one or more linear regression layers to produce the output probabilities

PyTorch takeaways

- `nn.Conv2d(in_channels, out_channels, kernel_size, padding, stride)`
- to keep resolution the same, use `padding = kernel_size // 2`
- `nn.Avg/MaxPool2d(in_channels, kernel_size)`
- PyTorch automatically sets `stride = kernel_size` resulting in final resolution being initial resolution divided by kernel size
- `nn.BatchNorm2d(in_channels)`
- applies normalization to each channel, independently