Computer Vision

Montag, 20. Juli 2020 09:50

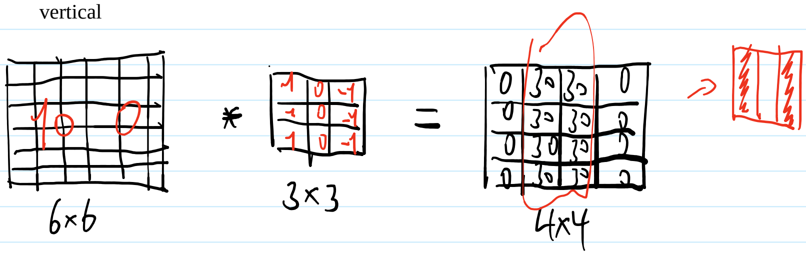
Convolution - filter

-- elementwise multiplication

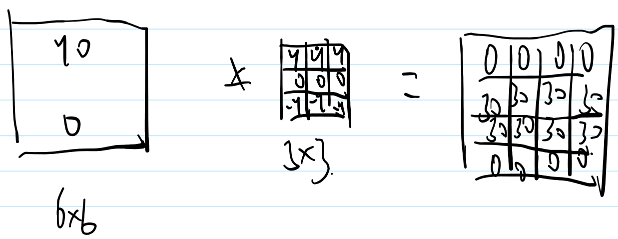
to create filter in: Python -- conv_forward; tensorFlow -- tf.nn.conv2d; Keras -- conv2D

Edge detection:

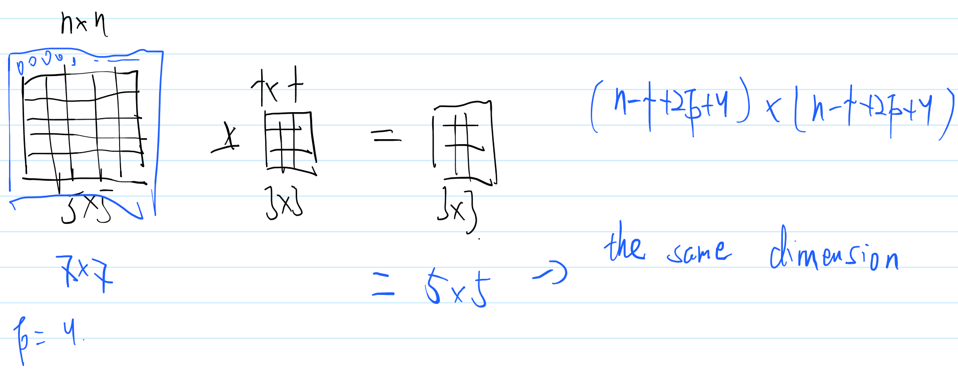
vertical



horizental

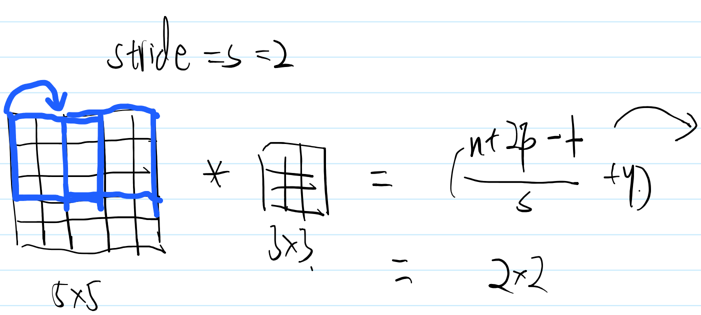


Padding



valid convolution -> no pading
same convolution -> p = (f-1)/2 -> output size == input size

Stride



if not integer:
floor((n+2p-f)/s+1)

Convolution on RGB image -- convolution on each channel

what CNN acturally does

Convolution on RGB image -- convolution on each channel

6x6x3 * 3x3x3 = 4x4

must match

what CNN acturally does

6x6x3 * 3x3x1 = 4x4

3x3x1 = 4x4

2 different filters

4x4x2

$$n \times n \times n_c * f \times f \times n_c = (n-f+1) \times (n-f+1) \times n_c'$$

n_c: # channels

n_c': # filters

for convolutional layer l:

$f^{[l]}$ - filter size

$p^{[l]}$ - padding

$s^{[l]}$ = stride

$n_c^{[l]}$ = # filters

Input $n_H^{[l-1]} \times n_W^{[l-1]} \times n_C^{[l-1]}$

Output $n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$

$$n_H^{[l]} = \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1$$

$a^{[l]}$ activation $\quad A^{[l]} = m * a^{[l]}$

weight $\underbrace{f^{[l]} \times f^{[l]} \times n_c^{[l-1]}}_{\text{dimension of one filter}} \times n_c^{[l]}$

bias $n_c^{[l]} \to (1, 1, 1, \ldots n_c^{[l]})$

$\underbrace{6\times6\times3}$ $a^{[0]}$

* 3x3x3 filter 1 → ReLU $\underbrace{4\times4 + b_1}$ → 4x4

* 3x3x3 filter 2 → ReLU $\underbrace{4\times4 + b_2}$ → 4x4

$\underbrace{\text{filter}}_{\text{"}w^{[1]}\text{"}}$

4x4x2 $a^{[1]}$

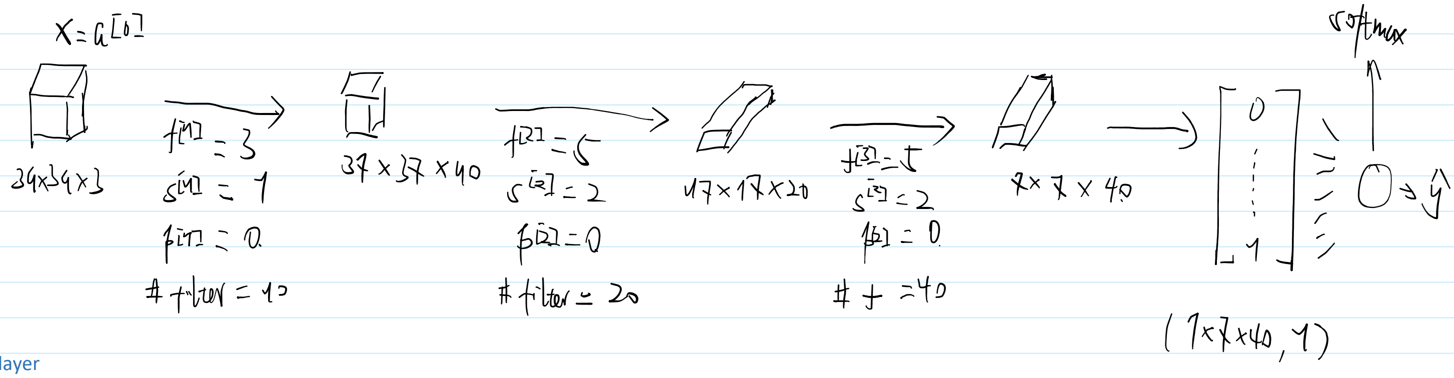
Linear: $W^{[1]} a^{[0]} + b^{[1]} \to z^{[1]}$

One layer, from $a^{[0]} \to a^{[1]}$ with 2 filters

In this layer there are (3*3*3 + 1) * 2 = 56 parameters to learning.

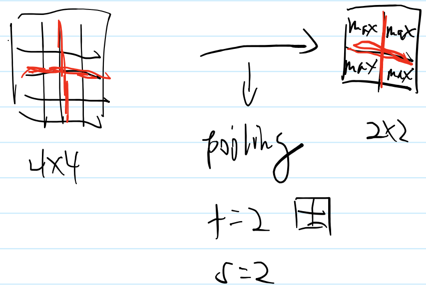
1 filter $\quad$ bias $\quad$ # filters

Example:



Pooling layer

Inituition: reduce the dimension by keeping only the strongest features



Average pooling layer: instead of taking max, taking average.

Why using Conv net instead of convential deep learning network? -> input size (features) is huge with respect to image/computer vision problem. To get same output size for each layer (how many features can be computered), conv net need to learn less parameters. Because, as to image convolution, actually a lot of pixels (information) are used more than once, so using conv net a lot of parameters(learned) are shared. So it is faster.

Parameters to be learned:

$$f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]} + \underbrace{n_c^{[l]}}_{\text{bias}}$$

Take away from Programming Assignment:

Build from scratch with numpy:

- Input shape = (m, n_H_prev, n_W_prev, n_C_prev)
- Output Z shape = (m, n_H, n_W, n_C)
- n_H = int((n_H_prev + 2*pad - f)/stride + 1), same applied to n_W
- Weights shape (W) = (f, f, n_C_prev, n_C)
- biases (b) shape = (1, 1, 1, n_C)
- A[i, h, w, c] = activation(Z[i, h, w, c])
- Pooling layer: reduces the height and width of the input. It helps reduce computation, as well as helps make feature detectors more invariant to its position in the input.

Build with TensowFlow:

- tf.nn.conv2d(X,W, strides = [1,s,s,1], padding = 'SAME'): given an input XX and a group of filters WW , this function convolves WW 's filters on X. The third parameter ([1,s,s,1]) represents the strides for each dimension of the input (m, n_H_prev, n_W_prev, n_C_prev).
- tf.nn.max_pool(A, ksize = [1,f,f,1], strides = [1,s,s,1], padding = 'SAME'): given an input A, this function uses a window of size (f, f) and strides of size (s, s) to carry out max pooling over each window.
- tf.nn.relu(Z): computes the elementwise ReLU of Z
- tf.contrib.layers.flatten(P): given a tensor "P", this function takes each training (or test) example in the batch and flattens it into a 1D vector. (batch_size, k), where k=h×w×ck=h×w×c .
- tf.contrib.layers.fully_connected(F, num_outputs): given the flattened input F, it returns the output computed using a fully connected layer, automatically initilizing the weights on this layer.
- forward_propagation function : CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> ……. -> FLATTEN -> FULLYCONNECTED.
- tf.nn.softmax_cross_entropy_with_logits(logits = Z, labels = Y): computes the softmax entropy loss. This function both computes the softmax activation function as well as the resulting loss. Note that, the input of softmax is not A is Z.
- tf.reduce_mean: computes the mean of elements across dimensions of a tensor. Use this to calculate the sum of the losses over all the examples to get the overall cost.