Deformable Convolutional Neural Networks

Janos and Julian

Universität Bonn,

Abstract. We implement a deformable convolutional neural network module, introduced by Dai et al. in [1]. Deformable CNNs enhance usual CNNs by making the sampling area of the convolution kernels "deformable". The spatial sampling area can be learned by backpropagation in an unsupervised manner. They add little computational overhead and can therefore easily replace CNN layers in deep neural networks. We detail how to efficiently implement a deformable CNN on a CPU using the Torch framework.

Keywords: cnn, im2col, deformable, torch

1 Introduction

We start by considering a traditional convolution layer. Given a three dimensional image X with dimension $\dim X = C_1 \times H \times W$, the convolution layer samples the input feature map with a kernel K of a fixed geometric shape, usually rectangular. More precisely, if we want to compute C_2 output features (so $\dim K = C_2 \times C_1 \times kH \times kW$) and the sampling region is parameterized by $\mathcal{R} = \{(0,0), (0,1), (1,0), \dots, (kH, kW)\}$, then the convolution operation of X and K at a point p_o in the c_2'th output feature can be expressed more succinctly as

$$Y(c_2, p_o) = b(c_2) + \sum_{c_1=1}^{C_1} \sum_{p_n \in \mathcal{R}} X(c_1, p_o + p_n) K(c_2, c_1, p_n), \tag{1}$$

where each output feature maintains a scalar bias $b(c_2)$ which is added to the respective output map after the convolution. To overcome the geometric limitation of the rectangular sampling region, Dai et al. introduce $kH \cdot kW$ offsets Δp_n for each point p_o in the ouput. So a deformed convolution is computed as

$$Y(c_2, p_o) = b(c_2) + \sum_{c_1=1}^{C_1} \sum_{p_n \in \mathcal{R}} X(p_o + p_n + \Delta p_n) K(c_2, c_1, p_n). \tag{2}$$

Remark: Note that, in the above notation, Δp_n does not only depend on p_n, but also on p_o.
Since the offsets Δp_n do not need to be integral, Dai et al. interpolate bilineraly. For any $p \in \mathbb{R}^2$, they set

$$X(c_1, p) = \sum_q G(q, p) X(c_1, q) = \sum_q (1 - |q^y - p^y|)(1 - |q^x - p^x|) X(c_1, q). \tag{3}$$

A deformable convolution layer consists of two modules (see also figure 1): The first module is the offset predictor (OP), which generates offsets by feeding the input through a traditional convolutional layer. It computes $2 \cdot kH \cdot kW$ output features, which correspond to $kH \cdot kW$ two-dimensional offsets for each sampling region. In particular, the same offsets are used across all input channels. The second module performs the deformed convolution (DC) specified by equation (2).

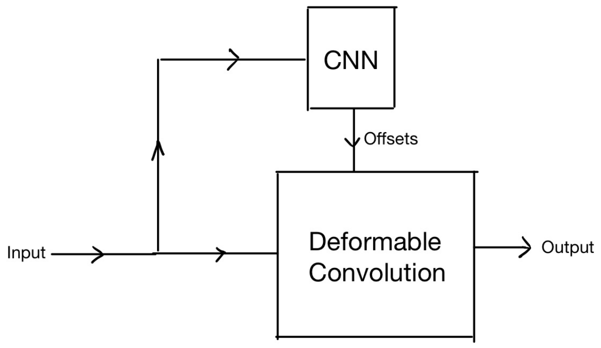

Fig. 1. Architecture of a deformable convolution layer

2 Implementaion

In order to implement a deformable convolution module in Torch, we need to define three functions:

1. A forwarding routine which takes an input and feeds it through the layer,
2. a function that takes the gradient $\frac{\partial E}{\partial Y(c_2,p_o)} =: \delta(c_2,p_o)$ of an error function E with respect to the ouput of the forwarding routine, propagates it back through the layer and computes the gradient of E w.r.t. the input, and
3. a method that, given the gradients $\delta(c_2,p_o)$, calculates the gradient of E w.r.t. all parameters of the module.

We derive the corresponding formulas and explain how to implement them efficiently.

2.1 Forward Pass

To compute a forward pass through a traditional CNN, an im2col routine is typically used, which transforms the feature maps into a two-dimensional matrix

A (see also figure 2). For simplicity, we assume that there is no padding and that stride size is set to one. Then, the output feature maps have height $H_{out} = H - kH + 1$ and width $W_{out} = W - kW + 1$. With this notation, the matrix A contains for each point p_o in the output feature map a column whose entries are $X(p_o + p_n)$ for $p_n \in \mathcal{R}$. If we consider K as a $C_2 \times C_1 \cdot kH \cdot kW$ matrix, then we can express the convolution operation as a matrix product of K and A.

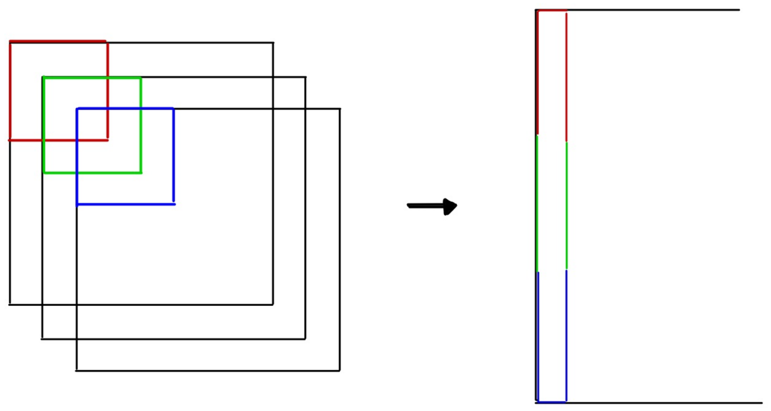

Fig. 2. Schematic representation of im2col. The three-dimensional patches are flattened into columns.

Similarly, we can write the deformed convolution as a matrix product, if, instead of $X(p_o + p_n),\ p_n \in \mathcal{R}$, we take the columns of A to be $X(p_o + p_n + \Delta p_n),\ p_n \in \mathcal{R}$, see also figure 3.
Note that when computing $X(p_o + p_n + \Delta p_n)$ using the formula in (3), the term $G(q, p)$ is nonzero only for the (at most) four lattice points q enclosing p. Let $a = (a^y, a^x)$ be the upperleft of these four points. To propagate the gradients efficiently through the deformable convolution layer, we save the coordinates of the point a, the channel index c_2 and the four values

$$\begin{aligned} w_0 &= 1 - (p^y - a^y), & w_1 &= 1 - (p^x - a^x), \\ w_2 &= 1 - ((a^y + 1) - p^y), & w_3 &= 1 - ((a^x + 1) - p^x) \end{aligned}$$

in the two tensors BufferIndices and BufferWeights. These seven values together with the input feature maps fully determine the value of $X(p_0 + p_n + \Delta p_n)$. If the point $p_o + p_n + \Delta p_n$ does not lie within the spatial dimensions of the input map, we project it onto it before using the interpolation formula.

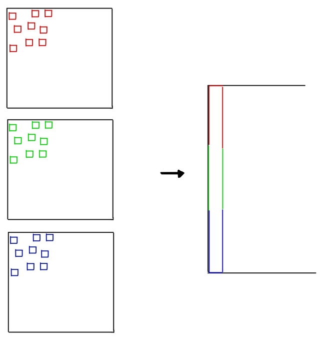

Fig. 3. Adapted version of im2col for deformable convolutions.

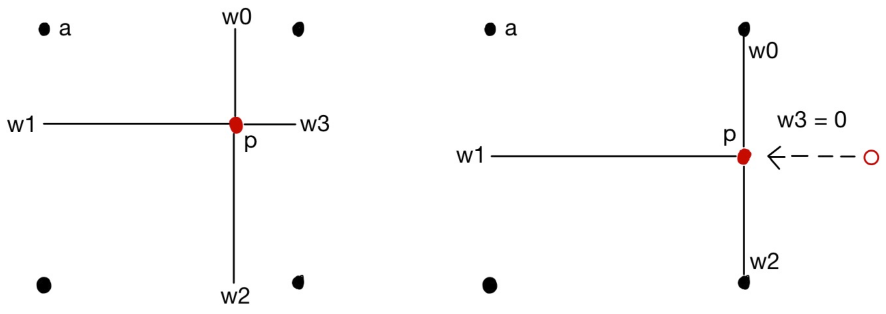

Fig. 4. Left: During interpolation, we save the seven values $w_0, \ldots, w_3$ and $c_1, a = (a^y, a^x)$ which determine $X(p)$. Right: Degenerate case. If the point lies outside of the input map, it is projected onto the image border.

2.2 Gradient w.r.t. Paramters

We now proceed to compute the gradient of the error function E with respect to the kernel of the deformable convolution. As usual, we use the chain rule.

$$\frac{\partial E}{\partial K(c_2, c_1, p_n)} = \sum_{p_o} \frac{\partial E}{\partial Y(c_2, p_o)} \frac{\partial Y(c_2, p_o)}{\partial K(c_2, c_1, p_n)}$$

Substituting the formula for Y from equation (2) yields

$$\frac{\partial E}{\partial K(c_2, c_1, p_n)} = \sum_{p_o} \delta(c_2, p_o) X(c_1, p_o + p_n + \Delta p_o).$$

So we can compute $\frac{\partial E}{\partial K(c_2, c_1, p_n)}$ as the deformed convolution of δ with X at the point p_n. We can use the saved values from the forward pass, to quickly compute $X(c_1, p_n + p_0 + \Delta p_o)$.
The gradient of E w.r.t. the biases $b(c_2)$ can easily be computed using formula (2) and the chain rule:

$$\frac{\partial E}{\partial b(c_2)} = \sum_{p_o} \frac{\partial E}{\partial Y(c_2, p_o)} \frac{\partial Y(c_2, p_o)}{\partial b(c_2)} = \sum_{p_o} \delta(c_2, p_o).$$

2.3 Gradient w.r.t. Input

We now compute the derivative of E w.r.t. the input X. Recall that E depends on the input X and the offsets $OP(X)$ and therefore

$$\frac{\partial E(X, OP(X))}{\partial X} = \partial_1(E)(X, OP(X)) + \partial_2(E)(X, OP(X)) \cdot D(OP)(X).$$

Here, with $D(OP)$ we denote the Jacobian of the offset predictor, and with $\partial_1(E)$ we denote the first $\#(X) = C_1 \cdot H \cdot W$ partial derivatives of E w.r.t. X; similarly, $\partial_2(E)$ denotes the remaining partial derivatives. We start by computing the first summand. We do this in two steps; first, we derive the gradient of E with respect to $A := \text{im2col}(X)$ using the chain rule:

$$\frac{\partial E}{\partial A(p_c)} = \sum_{c_2=1}^{C_2} \sum_{p_o} \frac{\partial E}{\partial Y(c_2, p_o)} \frac{\partial Y(c_2, p_o)}{\partial A(p_c)}.$$

We remember that Y was computed as the matrix product of K and A, so $Y(c_2, p_o) = \sum_i K(c_2, i) A(i, p_o)$. Thus the derivative of $Y(c_2, p_o)$ w.r.t. $A(p_c)$ is given by $K(c_2, i^*)$ for some i^*, p_o^* satisfying $p_c = (i^*, p_o^*)$. Hence

$$\begin{aligned} \frac{\partial E}{\partial A(p_c)} &= \sum_{c_2=1}^{C_2} \delta(c_2, p_c^*) K(c_2, i^*) \\ &= (K^t \cdot \delta)(i^*, p_o^*) \\ &= (K^t \cdot \delta)(p_c). \end{aligned}$$

Here, δ has to be interpreted as a matrix by combining the last two dimensions. Finally, we see that, since

$$\frac{\partial E}{\partial X(c_1, p_i)} = \sum_{p_c} \frac{\partial E}{\partial A(p_c)} \frac{\partial A(p_c)}{\partial X(c_1, p_i)},$$

we can compute $\partial_1(E)(X, OP(X))$ using the following algorithm:

```
Input  : BufferIndices, BufferWeights and $\frac{\partial E}{\partial A}$
Output: $\partial_1(E)(X, OP(X))$
for $p_c$ in $A$ do
    $c_1, a \leftarrow$ BufferIndices($p_c$);
    $w_0, w_1, w_2, w_3 \leftarrow$ BufferWeights($p_c$);
    $\frac{\partial E}{\partial X(c_1, a^y, a^x)} \longleftarrow \frac{\partial E}{\partial X(c_1, a^y, a^x)} + \frac{\partial E}{\partial A(p_c)} \cdot w_0 \cdot w_1;$
    $\frac{\partial E}{\partial X(c_1, a^y + 1, a^x)} \longleftarrow \frac{\partial E}{\partial X(c_1, a^y + 1, a^x)} + \frac{\partial E}{\partial A(p_c)} \cdot w_1 \cdot w_2;$
    $\frac{\partial E}{\partial X(a^y + 1, a^x + 1)} \longleftarrow \frac{\partial E}{\partial X(c_1, a^y + 1, a^x + 1)} + \frac{\partial E}{\partial A(p_c)} \cdot w_2 \cdot w_3;$
    $\frac{\partial E}{\partial X(c_1, a^y, a^x + 1)} \longleftarrow \frac{\partial E}{\partial X(c_1, a^y, a^x + 1)} + \frac{\partial E}{\partial A(p_c)} \cdot w_3 \cdot w_0;$
end
```

The second summand, $\partial_2(E)(X, OP(X)) \cdot D(OP)(X)$, is obtained by calculating the gradient $\partial_2(E)(X, OP(X)) = \frac{\partial E}{\partial \Delta p_n}$ and propagating it back through the offset predictor. Since the OP is a standard convolution layer, this is easy to do.

To work out $\partial_2(E)(X, OP(X))$, we use

$$\begin{aligned}\frac{\partial E}{\partial \Delta p_n} &= \sum_{c_2=1}^{C_2} \sum_{p_o} \frac{\partial E}{\partial Y(c_2, p_o)} \frac{\partial Y(c_2, p_o)}{\partial \Delta p_n} \\ &= \sum_{c_2=1}^{C_2} \frac{\partial E}{\partial Y(c_2, p_o)} \frac{\partial Y(c_2, p_o)}{\partial \Delta p_n}\end{aligned}$$

where only one p_o contributes to the inner sum because each Δp_n belongs to a unique p_o (when writing p_o in the next formulas, we refer to that unique p_o). The gradient of the output w.r.t. the offset can be written as

$$\begin{aligned}\frac{\partial Y(c_2, p_o)}{\partial \Delta p_n} &= \sum_{c_1=1}^{C_1} \sum_{p_m \in \mathcal{R}} K(c_2, c_1, p_m) \frac{\partial X(c_1, p_o + p_m + \Delta p_m)}{\partial \Delta p_n} \\ &= \sum_{c_1=1}^{C_1} K(c_2, c_1, p_n) \frac{\partial X(c_1, p_o + p_n + \Delta p_n)}{\partial \Delta p_n}.\end{aligned}$$

Note that $\Delta p_n = (\Delta p_n^y, \Delta p_n^x)$ is actually two-dimensional and the above calculations hold for both components. Taking the partial derivative of X w.r.t. the corresponding component and using formula (3), we get

$$\begin{aligned}\frac{\partial X(c_1, p_o + p_n + \Delta p_n)}{\partial \Delta p_n^y} =& - w_1 X(c_1, (a^y, a^x)) - w_3 X(c_1, (a^y, a^x + 1)) \\ &+ w_1 X(c_1, (a^y + 1, a^x)) + w_3 X(c_1, (a^y + 1, a^x + 1)), \\ \frac{\partial X(c_1, p_o + p_n + \Delta p_n)}{\partial \Delta p_n^x} =& - w_0 X(c_1, (a^y, a^x)) + w_0 X(c_1, (a^y, a^x + 1)) \\ &- w_2 X(c_1, (a^y + 1, a^x)) + w_2 X(c_1, (a^y + 1, a^x + 1)).\end{aligned}$$

For this calculation, we can use the values which we saved during the forward step.
Note that these equations only apply if $p_o + p_n + \Delta p_n$ has not been projected onto the map; if it was projected onto the map in y-direction, $\frac{\partial X(c_1, p_o + p_n + \Delta p_n)}{\partial \Delta p_n^y}$ has to be set to 0, and analogously for the x-direction.

2.4 Testing

To test our implementation, we used the Jacobian module provided by Torch. For each parameter θ of DC and OP, the Jacobian module computes the gradient numerically using the formula

$$\frac{DC(\theta + \varepsilon) - DC(\theta - \varepsilon)}{2\varepsilon}.$$

This was really helpful, since we had a convenient and fast way to check if the gradients were computed correctly.

3 Results

As a baseline, we chose the LeNet-5 architecture, introduced 1998 by Y. LeCun at al in [2], using the official Torch implementation of convolution layers.

We exchanged the convolution layers for our implementation of deformable convolution layers. In the following, we will call the resulting network LeNet-5-DC. The performance of these two nets were not quiet comparable, but, taking into account that the Torch code multithreads, not too bad. All perfomance tests were conducted on a Intel Core i7-940 processor.

	Deformable Convolution	Torch Spatial Convolution
Forward	$115.85 \cdot 10^{-4}$s	$2.95 \cdot 10^{-4}$s
UpdateGradInput	$101.87 \cdot 10^{-4}$s	$9.34 \cdot 10^{-4}$s
AccGradParameters	$61.57 \cdot 10^{-4}$s	$2.62 \cdot 10^{-4}$s

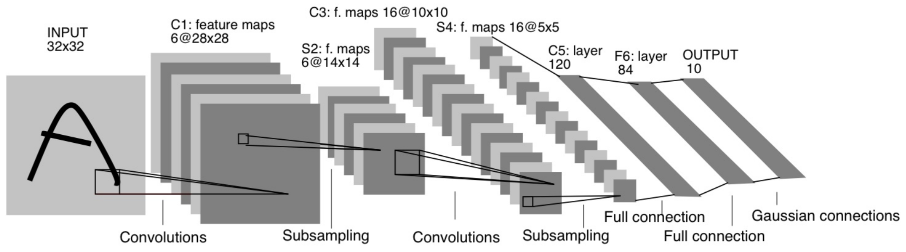

Fig. 5. Architecture of LeNet-5

We conclude that the Torch implementation of a convolution layer is approximately $40, 10$ and 40 respectively times faster than our implementation of a deformable convolution layer. We first tested LeNet-5 and LeNet-5-DC on the CIFAR-10 dataset. LeNet-5-DC was trained for 29 epochs, then we trained only the offset predictor for two further epochs.

	LeNet-5-DC, 29+2 epochs	LeNet-5, 31 epochs
accuracy on testset	54.34	48.3
error	0.5	0.31

To visualize the spatial sampling area of LeNet-5-DC we split the network after the second deformable convolution layer. Using the first of the two resulting networks, we calculated the input gradient of a single output point.

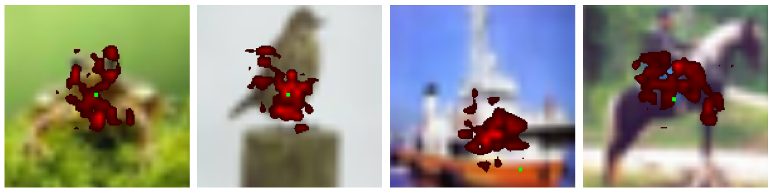

Fig. 6. The red area is the spatial sampling area of the green point. The higher the intensity the more it contributed.

The second trainingset we tried was the MNIST dataset. We first trained for three epochs. Then we fixed the parameters of the deformable convolution and only trained the offsets for two more epochs.

	LeNet-5-DC 3+2 epochs	LeNet-5-DC 3 epochs	LeNet-5 5 epochs
accuracy on testset	98,87	98,35	98.49
error	0.034	0.057	0.035

We used the same technique as above to visualize the spatial sampling area.

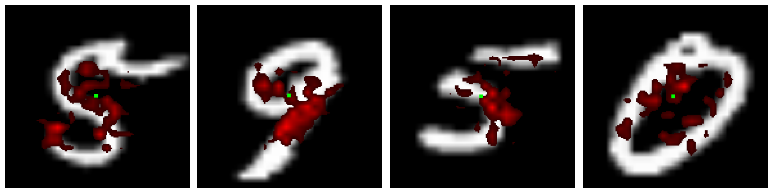

Fig. 7. The first two images were taken after three epochs of training. The third and fourth after two additional offset training epochs.

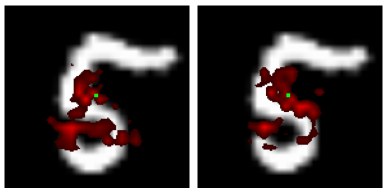

Fig. 8. Here we can see how the sampling area changed after the two additional epochs.

Notes and Comments. The implementation can be found on github. In the first days of the lab we wrote a standard convolution module, mimicking the official Torch implementation. The code can also be found in the same repository.

Outlook. While our implementation can be used for classifying datasets with small images, it is still too slow for larger images. For large images, multithreading would speed up the module significantly. As one can see in the benchmark, our implementaion loses a lot of time in the Forward and AccGradParameters functions. We think that rearranging the buffer dimensions carefully would speed these routines up due to less jumps in the memory.
Furthermore, mechanisms like padding and strides could be added to make the module more flexible.

References

1. Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Li, Guodong Zhang, Han Hu, Yichen Wei: Deformable Convolutional Networks
arXiv:1703.06211v3[cs.CV]
2. Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner: Gradient-Based Learning Applied to Document Recognition
Proceedings of the IEEE 86.11 (Nov. 1998), pp. 2278-2324