Note on the implementation of a convolutional neural networks.

This post is a follow-up on the second assignment proposed as part of the Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition.

The last part of the assignement deals with the implementation of a convolutional neural network. Among other things, this implies the implementation of forward and backward passes for convolutional layers, pooling layers, spatial-batch normalisations and other non-linearities.

Instead of writing everything on papers and, undoubtedly, lost myself in the jungle of indices, I decided to document my derivation in this post.

Convolutional layer

Convolutional layers are the building blocks of conv-net. They convolve their inputs with learnable filters to extract what are called **activation maps**.

Notation

I am going to follow the notation of the assignment.

In particular,

- Input x (N, C, H, W)
- Weights w (F, C, HH, WW)
- Output y (N,F, Hh, Hw)

and the indices, unless differently specified, are

- n for the different images
- f for the different filters
- c for the different channels
- f, k for the spatial outputs

Forward pass

The first step is the implementation of the forward pass. Instead of jumping into it, let's first look at an example to see what it does.

We are going to consider the convolution of an input x of size $H = W = 5$ with a filter of size $HH = WW = 3$ with stride $S = 2$ and zero padding $P = 0$.

Graphically, x looks like

$$x = \begin{pmatrix} x_{00} & x_{01} & x_{02} & x_{03} & x_{04} \\ x_{10} & x_{11} & x_{12} & x_{13} & x_{14} \\ x_{20} & x_{21} & x_{22} & x_{23} & x_{24} \\ x_{30} & x_{31} & x_{32} & x_{33} & x_{34} \\ x_{40} & x_{41} & x_{42} & x_{43} & x_{44} \end{pmatrix},$$

the filter w

$$x = \begin{pmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \\ w_{20} & w_{21} & w_{22} \end{pmatrix},$$

and the bias is just b as we consider only one filter. The output y is a vector of size $Hh = Hw = 2$ as Hh (and Hw here) is given by

$$Hh = 1 + (H + 2P - HH)/S.$$

During the convolution process, the filter is convolved to the input in a way that is defined by the stride and produces its proper activation map y. For instance, y_{11} resumes to the multiplication element-wise of the lower right part (size 3×3) of x with the filter. Mathematically, it reads

$$\begin{aligned} y_{11} &= x_{22}w_{00} + x_{23}w_{01} + \ldots + x_{43}w_{21} + w_{44}w_{22} \\ &= \sum_{p=2}^{4}\sum_{q=2}^{4} x_{p,q}w_{p-p_0,q-q_0} + b \end{aligned}$$

where the beginning of the sum is given by the index of the element times the stride S, and the size of the sums are given by the size of their respective filter dimensions (HH or WW).

Generalising from this example, we see that the output reads

$$y_{kl} = \sum_{p=p_0}^{p=p_0+\Delta p}\sum_{q=q_0}^{q=q_0+\Delta q} x_{p,q}^{pad} w_{p-p_0,q-q_0} + b$$

where

$$
\begin{aligned}
p_0 &= Sk \\
q_0 &= Sl \\
\Delta p &= HH \\
\Delta q &= WW
\end{aligned}
$$

and x^{pad} is the input padded with the adequate number of zeros (given by P).

With N images, F filters and C channels, the example above easily generalised and the forward pass for the convolutional layer reads

$$
\begin{aligned}
y_{n,f,k,l} &= \sum_{c} \sum_{p=p_0}^{p=p_0+\Delta p} \sum_{q=q_0}^{q=q_0+\Delta q} x^{pad}_{n,c,p,q} w_{f,c,p-p_0,q-q_0} + b_f \\
&= \sum_{c} \sum_{p=0}^{\Delta p} \sum_{q=0}^{\Delta q} x^{pad}_{n,c,p+p_0,q+q_0} w_{f,c,p,q} + b_f \\
&= \sum_{c} \sum_{p=0}^{HH} \sum_{q=0}^{WW} x^{pad}_{n,c,p+kS,q+lS} w_{f,c,p,q} + b_f
\end{aligned}
$$

which nicely translates mathematically that to obtain the specific value indexed by k, l in the f activation map for an input n, select the corresponding subpart of the input of size (HH, WW), multiply it by the filter f and sum all the resulting terms, i.e. the convolution!

In python, it looks like

```
x_pad = np.pad(x, ((0,), (0,), (P,), (P,)), 'constant')
out = np.zeros((N, F, Hh, Hw))
for n in range(N):  # First, iterate over all the images
    for f in range(F):  # Second, iterate over all the kernels
        for k in range(Hh):
            for l in range(Hw):
                out[n, f, k, l] = np.sum(x_pad[n, :, k * S:k * S +
```

Backward pass

During the backward pass, we have to compute $\frac{d\mathcal{L}}{dw}$, $\frac{d\mathcal{L}}{dx}$, $\frac{d\mathcal{L}}{db}$ where each gradient with respect to a quantity contains a vector of size equal to the quantity itself and where we know from the previous pass $\frac{d\mathcal{L}}{dy}$ (see previous post for more details).

Gradient with respect to the weights $\frac{d\mathcal{L}}{dw}$

The gradient of the loss with respect to the weights has the same size as the weights themselves (F,C,HH,WW). Chaining by the gradient of the loss with respect to the outputs y, it reads

$$\frac{d\mathcal{L}}{dw_{f',c',i,j}} = \sum_{n,f,k,l} \frac{d\mathcal{L}}{dy_{n,f,k,l}} \frac{dy_{n,f,k,l}}{dw_{f',c',i,j}}.$$

The expression of $y_{n,f,k,l}$ is derived above and therefore, its derivative with respect to the weight w reads

$$\begin{aligned}
\frac{dy_{n,f,k,l}}{dw_{f',c',i,j}} &= \frac{d}{dw_{f',c',i,j}} \left(\sum_{c} \sum_{p=0}^{p=\Delta p} \sum_{q=0}^{q=\Delta q} x^{pad}_{n,c,p+p_0,q+q_0} w_{f,c,p,q} \right) \\
&= \sum_{c} \sum_{p=0}^{p=\Delta p} \sum_{q=0}^{q=\Delta q} \delta_{c,c'} \delta_{f,f'} \delta_{p,i} \delta_{q,j} x^{pad}_{n,c,p+p_0,q+q_0} \\
&= \delta_{f,f'} x^{pad}_{n,c',i+Sk,j+Sl}
\end{aligned}$$

Injecting this expression back into the gradient of the loss with respect to the weights, we then have

$$\begin{aligned}
\frac{d\mathcal{L}}{dw_{f',c',i,j}} &= \sum_{n,f,k,l} \frac{d\mathcal{L}}{dy_{n,f,k,l}} \delta_{f,f'} x^{pad}_{n,c',i+Sk,j+Sl} \\
&= \sum_{n,k,l} \frac{d\mathcal{L}}{dy_{n,f',k,l}} x^{pad}_{n,c',i+Sk,j+Sl}
\end{aligned}$$

which in python translates

```
dw = np.zeros((F, C, HH, WW))
for fprime in range(F):
    for cprime in range(C):
        for i in range(HH):
            for j in range(WW):
                sub_xpad = x_pad[:, cprime, i:i + Hh * S:S, j:j +
                dw[fprime, cprime, i, j] = np.sum(dout[:, fprime,
```

Gradient with respect to the bias $\frac{d\mathcal{L}}{db}$

The gradient of the loss with respect to the bias is of size (F). Chaining by the gradient of the loss with respect to the outputs y and simplifying, it reads

$$\begin{aligned}\frac{d\mathcal{L}}{db_{f'}} &= \sum_{n,f,k,l} \frac{d\mathcal{L}}{dy_{n,f,k,l}} \frac{dy_{n,f,k,l}}{db_{f'}} \\ &= \sum_{n,f,k,l} \frac{d\mathcal{L}}{dy_{n,f,k,l}} \delta_{f,f'} \\ &= \sum_{n,k,l} \frac{d\mathcal{L}}{dy_{n,f',k,l}}\end{aligned}$$

which, in python, translates

```
db = np.zeros((F))
for fprime in range(F):
    db[fprime] = np.sum(dout[:, fprime, :, :])
```

Gradient with respect to the input $\frac{d\mathcal{L}}{dx}$

As above, we first chain by the gradient of the loss with respect to the output y, which gives

$$\frac{d\mathcal{L}}{dx_{n',c',i,j}} = \sum_{n,f,k,l} \frac{d\mathcal{L}}{dy_{n,f,k,l}} \frac{dy_{n,f,k,l}}{dx_{n',c',i,j}}$$

The second term reads

$$\frac{dy_{n,f,k,l}}{dx_{n',c',i,j}} = \frac{d}{dx_{n',c',i,j}} \left(\sum_{c} \sum_{p=0}^{HH} \sum_{q=0}^{WW} x^{pad}_{n,c,p+kS,q+lS} w_{f,c,p,q} + b_f \right)$$

where we have to handle carefully the fact that y depends on the padded version of x^{pad} and not x itself. In other words, we better first chain with the gradient of y with respect to the padded version of x^{pad} to get

$$\frac{dy_{n,f,k,l}}{dx_{n',c',i,j}} = \sum_{n",c",i",j"} \frac{dy_{n,f,k,l}}{dx^{pad}_{n",c",i",j"}} \frac{dx^{pad}_{n",c",i",j"}}{dx_{n',c',i,j}}.$$

Let's first look at the second term, the gradient of x^{pad} with respect to x. There is a simple relationship between the two which reads

$$x^{pad}_{n",x",i",j"} = x_{n",x",i"-P,j"-P} \mathbb{1}(P \leq i\ "\leq H - P) \mathbb{1}(P \leq j\ "\leq W - P)$$

then,

$$\frac{dx^{pad}_{n",c",i",j"}}{dx_{n',c',i,j}} = \delta_{n",n'} \delta_{c",c'} \delta_{i,i"-P} \delta_{j,j"-P}$$

For the first term,

$$\frac{dy_{n,f,k,l}}{dx^{pad}_{n",c",i",j"}} = \frac{d}{dx^{pad}_{n",c",i",j"}}\left(\sum_{c}\sum_{p=0}^{HH}\sum_{q=0}^{WW} x^{pad}_{n,c,p+kS,q+lS} w_{f,c,p,q} + b_f\right)$$
$$= \sum_{c}\sum_{p=0}^{HH}\sum_{q=0}^{WW} \delta_{n,n"}\delta_{c,c"}\delta_{p+kS,i"}\delta_{q+lS,j"} w_{f,c,p,q}$$
$$= \sum_{p=0}^{HH}\sum_{q=0}^{WW} \delta_{n,n"}\delta_{p+kS,i"}\delta_{q+lS,j"} w_{f,c",p,q}$$

Then, putting both terms together, we find that

$$\frac{dy_{n,f,k,l}}{dx_{n',c',i,j}} = \sum_{n",c",i",j"} \frac{dy_{n,f,k,l}}{dx^{pad}_{n",c",i",j"}} \frac{dx^{pad}_{n",c",i",j"}}{dx_{n',c',i,j}}$$
$$= \delta_{n,n'} \sum_{i",j"}\sum_{p=0}^{HH}\sum_{q=0}^{WW} \delta_{p+kS,i"}\delta_{i,i"-P}\delta_{j,j"-P}\delta_{q+lS,j"} w_{f,c',p,q}$$
$$= \delta_{n,n'} \sum_{p=0}^{HH}\sum_{q=0}^{WW} \delta_{p+kS,i+P}\delta_{q+lS,j+P} w_{f,c',p,q}$$

Hence, the gradient of the loss with respect to the inputs finally reads

$$\frac{d\mathcal{L}}{dx_{n',c',i,j}} = \sum_{n,f,k,l} \frac{d\mathcal{L}}{dy_{n,f,k,l}} \frac{dy_{n,f,k,l}}{dx_{n',c',i,j}}$$
$$= \sum_{f,k,l}\sum_{p=0}^{HH}\sum_{q=0}^{WW} \frac{d\mathcal{L}}{dy_{n',f,k,l}} \delta_{p+kS,i+P}\delta_{q+lS,j+P} w_{f,c',p,q}$$

which in python can be written with 9 beautiful loops ;)

```
 # For dx : Size (N,C,H,W)
dx = np.zeros((N, C, H, W))
for nprime in range(N):
    for cprime in range(C):
        for i in range(H):
            for j in range(W):
                for f in range(F):
                    for k in range(Hh):
                        for l in range(Hw):
                            for p in range(HH):
                                for q in range(WW):
```

```
                                    if (p + k * S == i + P) & (q +
                                        dx[nprime, cprime, i, j] +
```

Though inefficient, this implementation has the advantage of translating point by point the formula. A may be more clever implementation could look like

```
dx = np.zeros((N, C, H, W))
for nprime in range(N):
    for i in range(H):
        for j in range(W):
            for f in range(F):
                for k in range(Hh):
                    for l in range(Hw):
                        mask1 = np.zeros_like(w[f, :, :, :])
                        mask2 = np.zeros_like(w[f, :, :, :])
                        if (i + P - k * S) < HH and (i + P - k * S
                            mask1[:, i + P - k * S, :] = 1.0
                        if (j + P - l * S) < WW and (j + P - l * S
                            mask2[:, :, j + P - l * S] = 1.0
                        w_masked = np.sum(w[f, :, :, :] * mask1 *
                        dx[nprime, :, i, j] += dout[nprime, f, k,
```

which is somewhat still very inefficient ;) If anyone has any idea on how to remove the **i** and **j** loops, please tell me !

Pooling layer

A pooling layer reduces the spatial dimension of its input without affecting its depth.

Basically, if a given input x has a size (N, C, H, W), then the output will have a size (N, C, H_1, W_1) where H_1 and W_1 are given by

$$H_1 = (H - H_p)/S + 1$$
$$W_1 = (W - W_w)/S + 1$$

and where H_p, W_p and S are three hyperparameters which corresponds to

- H_p is the height of the pooling region
- H_w is the width of the pooling region
- S is the stride, the distance between two adjacent pooling region.

Forward pass

The forward pass is very similar to the one of the convolutional layer and reads

$$y_{n,c,k,l} = \max \begin{pmatrix} x_{n,c,kS,lS} & \dots & x_{n,c,kS,W_p+lS} \\ \dots & \dots & \dots \\ x_{n,c,kS+H_p,lS} & \dots & x_{n,c,kS+H_p,lS+W_p} \end{pmatrix}.$$

or, more pleasantly,

$$y_{n,c,k,l} = \max_{0 \leq p < H_p, 0 \leq q < W_p} x_{n,c,p+kS,q+lS}$$

which in python translates to

```
out = np.zeros((N, C, H1, W1))
for n in range(N):
    for c in range(C):
        for k in range(H1):
            for l in range(W1):
                out[n, c, k, l] = np.max(x[n, c, k * S:k * S + Hp,
```

Backward pass

The gradient of the loss with respect to the input x of the pooling layer writes

$$\frac{d\mathcal{L}}{dx_{n',c',i,j}} = \sum_{n,c,k,l} \frac{d\mathcal{L}}{dy_{n,c,k,l}} \frac{dy_{n,c,k,l}}{dx_{n',c',i,j}}.$$

Let's look at the second term. In particular, we are going to assume that the spatial indices of the max in $y_{n,c,k,l}$ are p_m and q_m respectively. Therefore,

$$y_{n,c,k,l} = x_{n,c,p_m,q_m}$$

and

$$\begin{aligned} \frac{d\mathcal{L}}{dx_{n',c',i,j}} &= \sum_{n,c,k,l} \frac{d\mathcal{L}}{dy_{n,c,k,l}} \frac{dy_{n,c,k,l}}{dx_{n',c',i,j}} \\ &= \sum_{n,c,k,l} \frac{d\mathcal{L}}{dy_{n,c,k,l}} \delta_{n,n'} \delta_{c,c'} \delta_{i,p_m} \delta_{j,q_m} \\ &= \sum_{k,l} \frac{d\mathcal{L}}{dy_{n',c',k,l}} \delta_{i,p_m} \delta_{j,q_m} \end{aligned}$$

and we are done! Indeed, in python, find the indices of the max is fairly easy and the lazy compute of the gradient reads

```
dx = np.zeros((N, C, H, W))
for nprime in range(
    for cprime in range(C):
        for i in range(H):
            for j in range(W):
                for k in range(H1):
                    for l in range(W1):
                        x_pooling = x[nprime, cprime, k * S:k * S+
                        maxi = np.max(x_pooling)
                        # Make sure to find the indexes in x and r
                        x_mask = x[nprime, cprime, :, :] == maxi
                        pm, qm = np.unravel_index(x_mask.argmax(),
                        if (i == pm) & (j == qm):
                            dx[nprime, cprime, i,j] += dout[nprime
```

Note here that we are calculating the same **x_pooling** many times. A more clever solution, computationally speaking is the following

```
dx = np.zeros((N, C, H, W))
for nprime in range(N):
    for cprime in range(C):
        for k in range(H1):
            for l in range(W1):
                x_pooling = x[nprime, cprime, k *S:k * S + Hp, l *
                maxi = np.max(x_pooling)

                x_mask = x_pooling == maxi
                dx[nprime, cprime, k * S:k * S + Hp, l * S:l *S +
```

But we are note looking for efficiency here, are we ?? ;)

Spatial batch-normalization

Finally, we are asked to implement a vanilla version of batch norm for the convolutional layer.

Indeed, following the argument that the feature map was produced using convolutions, then we expect the statistics of each feature channel to be relatively consistent both between different images and different locations within the same image. Therefore spatial batch normalization computes a mean and variance for each of the C feature channels by computing statistics over both the minibatch dimension N and the spatial dimensions H and W.

Forward pass

The forward pass is straighforward here

$$y_{nckl} = \gamma_c \hat{x}_{nckl} + \beta_c$$
$$\hat{x}_{nckl} = (x_{nckl} - \mu_c) \left(\sigma_c^2 + \epsilon\right)^{-1/2}$$

where

$$\mu_c = \frac{1}{NHW} \sum_{mqp} x_{mcqp}$$
$$\sigma_c^2 = \frac{1}{NHW} \sum_{mqp} \left(x_{mcqp} - \mu_c\right)^2$$

In four line of python, it resumes to

```
mu = (1. / (N * H * W) * np.sum(x, axis=(0, 2, 3))).reshape(1, C,
var = (1. / (N * H * W) * np.sum((x - mu)**2,axis=(0, 2, 3))).resh


xhat = (x - mu) / (np.sqrt(eps + var))
out = gamma.reshape(1, C, 1, 1) * xhat + beta.reshape(1, C, 1, 1)
```

Backward pass

In the backward pass, we have to find an expression for $\frac{d\mathcal{L}}{d\gamma}$, $\frac{d\mathcal{L}}{d\beta}$, $\frac{d\mathcal{L}}{dx}$ where each gradient with respect to a quantity contains a vector of size equal to the quantity itself.

I spent already an entire post explaining how to derive the gradient of the loss with respect to the centred inputs in a previous post and I just drop the generalized version for the conv-net application here.

Gradient of the loss with respect to β

$$\frac{d\mathcal{L}}{d\beta_{c'}} = \sum_{n,c,k,l} \frac{d\mathcal{L}}{dy_{n,c,k,l}} \frac{dy_{n,c,k,l}}{d\beta_{c'}}$$
$$= \sum_{n,k,l} \frac{d\mathcal{L}}{dy_{n,c',k,l}}$$

```
dbeta = np.sum(dout, axis=(0, 2, 3))
```

Gradient of the loss with respect to γ

$$\frac{d\mathcal{L}}{d\gamma_{c'}} = \sum_{n,c,k,l} \frac{d\mathcal{L}}{dy_{n,c,k,l}} \frac{dy_{n,c,k,l}}{d\gamma_{c'}}$$
$$= \sum_{n,k,l} \frac{d\mathcal{L}}{dy_{n,c',k,l}} \hat{x}_{n,c',k,l}$$

```
dgamma = np.sum(dout * xhat, axis=(0, 2, 3))
```

Gradient of the loss with respect to the input x

$$\frac{d\mathcal{L}}{dx_{n',c',k',l'}} = \sum_{n,c,k,l} \frac{d\mathcal{L}}{dy_{n,c,k,l}} \frac{dy_{n,c,k,l}}{dx_{n',c',k',l'}}$$
$$= \frac{1}{NHW} \gamma_{c'} \left(\sigma_c'^2 + \epsilon\right)^{-1/2} \left(NHW \frac{d\mathcal{L}}{dy_{n',c',k',l'}} - \sum_{n,k,l} \frac{d\mathcal{L}}{dy_{n,c',k,l}} \right)$$
$$- \frac{1}{NHW} \gamma_{c'} \left(\sigma_c'^2 + \epsilon\right)^{-1/2} \left(\left(x_{n',c',k',l'} - \mu_{c'}\right) \left(\sigma_c'^2 + \epsilon\right)^{-1} \sum_{n,k,l} \frac{d\mathcal{L}}{dy_{n,c',k,l}} \left(x_{n,c',k,l} - \right.\right.$$

In python

```
gamma = gamma.reshape(1, C, 1, 1)
beta = beta.reshape(1, C, 1, 1)
Nt = N * H * W
dx = (1. / Nt) * gamma * (var + eps)**(-1. / 2.) * (Nt * dout
        - np.sum(dout, axis=(0, 2, 3)).reshape(1, C, 1, 1)
        - (x -  mu) * (var  + eps)**(-1.0) *  np.sum(dout * (x  -
```

Conclusion

This post focus on the derivation of the forward and backward passes for different building blocks of convolutional neural networks. Namely

- A convolutional layer
- A pooling layer
- Spatial-batch normalization

I also document the corresponding code in **python** for those who want to implement their own convolutional neural networks.

To finish, I'd like to thank all the team from the CS231 Standford class who do a fantastic work in vulgarising the knowledge behind neural networks.

For those who want to take a look to my full implementation of a convolutional neural networks, you can found it here.

Written on February 2, 2016