Convnet: Implementing Convolution Layer with Numpy

Convolutional Neural Network or CNN or convnet for short, is everywhere right now in the wild. Almost every computer vision systems that was recently built are using some kind of convnet architecture. Since the AlexNet's (http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf) groundbreaking result in ImageNet 2012 challenge, every year, it was convnet's year.

There are many factors that contribute to convnet's success. One of them is the effectiveness of the convolution layer; the heart of convnet.

Architecture wise, convnet is just a usual feed forward net, put on top of convolution layer(s). So really, convolution layer is a kind of feature extractor that can effectively learn the optimal features, which makes the linear classifier put on top of it looks good.

Nowadays, to build a convnet model, it's easy: install one of those popular Deep Learning libraries like TensorFlow, Torch, or Theano, and use the prebuilt module to rapidly build the model.

However, to understand the convnet better, it's essential to get our hands dirty. So, let's try implementing the conv layer from scratch using Numpy!

Conv layer

As we've already know, every layer in a neural net consists of forward and backward computation, because of the backpropagation. Conv layer is no different, as essentially, it's just another neural net layer.

Before we get started, of course we need some theoritical knowledge of convnet. For this, I'd direct you to the excellent CS231n class (http://cs231n.github.io/convolutional-networks/).

Having already understand the theories, it's time for us to implement it. First, the forward computation!

Conv layer forward

As stated in the CS231n class (http://cs231n.github.io/convolutional-networks/), we can think of convolutional operation as a matrix multiplication, as essentially at every patch of images, we apply a filter on it by taking their dot product. What nice about this is that we could think about conv layer as feed forward layer (your usual neural net hidden layer) to some extent.

The implication is very nice: we can reuse our knowledge and thought process when implementing conv layer! We will see that later, especially in the backward computation.

Alright, let's define our function:

```
def conv_forward(X, W, b, stride=1, padding=1):
    pass
```

Our conv layer will accept an input in `X: DxCxHxW` dimension, input filter `W: NFxCxHFxHW`, and bias `b: Fx1`, where:

- `D` is the number of input
- `C` is the number of image channel
- `H` is the height of image

- `W` is the width of the image
- `NF` is the number of filter in the filter map `W`
- `HF` is the height of the filter, and finally
- `HW` is the width of the filter.

Another important parameters are the stride and padding of the convolution operation.

As stated before, we will approach the conv layer as kind of normal feed forward layer, which is just the matrix multiplication between the input and the weight. To do this, we will use a utility function called `im2col`, which essentially will stretch our input image depending on the filter, stride, and width. `im2col` utilities could be found in the second assigment files of CS231n (http://vision.stanford.edu/teaching/cs231n/winter1516_assignment2.zip).

Let's say we have a single image of `1x1x10x10` size and a single filter of `1x1x3x3`. We also use stride of 1 and padding of 1. Then, naively, if we're going to do convolution operation for our filter on the image, we will loop over the image, and take the dot product at each `3x3` location, because our filter size is `3x3`. The result is a single `1x1x10x10` image.

But, what if we don't want to do the loop? Or is there a nice way to do it?

Yes, there is!

What we need is to gather all the possible locations that we can apply our filter at, then do a single matrix multiplication to get the dot product at each of those possible locations. Hence, with the above setup, we will have 100 possible locations, the same as our original input, because doing `3x3` convolution with 1 stride and 1 padding will preserve the input dimension.

At every those 100 possible location, there exists the `3x3` patch, stretched to `9x1` column vector that we can do our `3x3` convolution on. So, with `im2col`, our image dimension now is: `9x100`.

To make the operation compatible, we will arrange our filter to `1x9`. Now, if we do a matrix multiplication over our stretched image and filter, we will have `1x100` image as a result, which we could reshape it back to `10x10` or `1x1x10x10` image.

Let's see the code for that.

```
# Let this be 3x3 convolution with stride = 1 and padding = 1
# Suppose our X is 5x1x10x10, X_col will be a 9x500 matrix
X_col = im2col_indices(X, h_filter, w_filter, padding=padding, stride=stride)
# Suppose we have 20 of 3x3 filter: 20x1x3x3. W_col will be 20x9 matrix
W_col = W.reshape(n_filters, -1)

# 20x9 x 9x500 = 20x500
out = W_col @ X_col + b

# Reshape back from 20x500 to 5x20x10x10
# i.e. for each of our 5 images, we have 20 results with size of 10x10
out = out.reshape(n_filters, h_out, w_out, n_x)
out = out.transpose(3, 0, 1, 2)
```

That basically it for the forward computation of the convolution layer. It's similar to the feed forward layer with two additions: `im2col` operation and thinkering about the dimension of our matrices.

It's definitely harder to implement, mainly because thinking in multidimension isn't that nice. We have to be careful with the dimension manipulation operations like the reshape and transpose as it's tricky to work with.

Conv layer backward

One of the trickiest part of implementing neural net model from scratch is to derive the partial derivative of a layer. Conv layer is no different, it's even more trickier as we have to deal with different operation (convolution instead of just affine transformation) and higher dimensional matrices.

Thankfully because we're using the `im2col` trick, we at least could reuse our knowledge on implementing the feed forward layer's backward computation!

First let's compute our bias gradient.

```
db = np.sum(dout, axis=(0, 2, 3))
db = db.reshape(n_filter, -1)
```

Remember that the matrix we're dealing with, i.e. `dout` is a `5x20x10x10` matrix, similar to the output of the forward computation step. As the bias is added to each of our filter, we're accumulating the gradient to the dimension that represent of the number of filter, which is the second dimension. Hence the sum is operated on all axis except the second.

Next, we will compute the gradient of the the filters `dW`.

```
# Transpose from 5x20x10x10 into 20x10x10x5, then reshape into 20x500
dout_reshaped = dout.transpose(1, 2, 3, 0).reshape(n_filter, -1)
# 20x500 x 500x9 = 20x9
dW = dout_reshaped @ X_col.T
# Reshape back to 20x1x3x3
dW = dW.reshape(W.shape)
```

It's similar with the normal feed forward layer, except with more convoluted (ha!) dimension manipulation.

Lastly, the input gradient `dX`. We're almost there!

```
# Reshape from 20x1x3x3 into 20x9
W_reshape = W.reshape(n_filter, -1)
# 9x20 x 20x500 = 9x500
dX_col = W_reshape.T @ dout_reshaped
# Stretched out image to the real image: 9x500 => 5x1x10x10
dX = col2im_indices(dX_col, X.shape, h_filter, w_filter, padding=padding, stri
```

Again, it's the same as feed forward layer with some careful reshaping! At the end though, we're getting the gradient of the stretched image (recall the use of `im2col`). To undo this, and getting the real image gradient, we're going to de-im2col that. We're going to apply the operation called `col2im` to the stretched image. And now we have our image input gradient!

Full source code

Here's the full source code for the forward and backward computation of the conv layer.

```
def conv_forward(X, W, b, stride=1, padding=1):
    cache = W, b, stride, padding
    n_filters, d_filter, h_filter, w_filter = W.shape
    n_x, d_x, h_x, w_x = X.shape
    h_out = (h_x - h_filter + 2 * padding) / stride + 1
    w_out = (w_x - w_filter + 2 * padding) / stride + 1

    if not h_out.is_integer() or not w_out.is_integer():
        raise Exception('Invalid output dimension!')

    h_out, w_out = int(h_out), int(w_out)

    X_col = im2col_indices(X, h_filter, w_filter, padding=padding, stride=stri
    W_col = W.reshape(n_filters, -1)

    out = W_col @ X_col + b
    out = out.reshape(n_filters, h_out, w_out, n_x)
    out = out.transpose(3, 0, 1, 2)

    cache = (X, W, b, stride, padding, X_col)

    return out, cache


def conv_backward(dout, cache):
    X, W, b, stride, padding, X_col = cache
    n_filter, d_filter, h_filter, w_filter = W.shape

    db = np.sum(dout, axis=(0, 2, 3))
    db = db.reshape(n_filter, -1)

    dout_reshaped = dout.transpose(1, 2, 3, 0).reshape(n_filter, -1)
    dW = dout_reshaped @ X_col.T
    dW = dW.reshape(W.shape)

    W_reshape = W.reshape(n_filter, -1)
    dX_col = W_reshape.T @ dout_reshaped
    dX = col2im_indices(dX_col, X.shape, h_filter, w_filter, padding=padding,

    return dX, dW, db
```

Also check out the complete code in my repository: https://github.com/wiseodd/hipsternet (https://github.com/wiseodd/hipsternet)!

Conclusion

As we can see, conv layer is just an extension of the normal feed forward layer, with some additions. Those are the `im2col` operation and dimension manipulation, as convnet, particularly for computer vision task assumes our input and weight to be arranged as 2d or 3d images, because we're trying to capture the spatial information in our data.

Dealing with multidimensional matrices as we will always encounter in convnet is tricky. A lot of careful dimension manipulation need to be done. But it's definitely a great exercise to visualize them in our mind!

References

- http://cs231n.github.io/convolutional-networks/ (http://cs231n.github.io/convolutional-networks/)
- http://vision.stanford.edu/teaching/cs231n/winter1516_assignment2.zip (http://vision.stanford.edu/teaching/cs231n/winter1516_assignment2.zip)
- http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf (http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf)

← PREVIOUS POST (/TRAVEL/2016/07/13/ISE/)

NEXT POST → (/TECHBLOG/2016/07/18/CONVNET-MAXPOOL-LAYER/)

(/feed.xml) (https://github.com/wiseodd)