Convolutional Layers in PyTorch

Defining Filters:

```
filter_vals = np.array( [-1,-1,1,1],
                        [-1,-1,1,1],
                        [-1,-1,1,1],
                        [-1,-1,1,1])
filters = np.array([
            filter_vals,
           -filter_vals,
            filter_vals.T,
           -filter_vals.T])
```

Defining Convolutional Layer:

```
class MyNN(nn.Module):

  ★ def __init__(self, weights):
        super(MyNN, self).__init__()
        h, w = weight.shape[2:]

        self.Conv = nn.Conv2d(1, 4,
                              Kernel_size=(h,w),
                              bias=False)
```

input depth

output depth

define a Convolutional layer...

...that has four filters...

...all of which are grayscale.

```
        self.Conv.weight = torch.nn.Parameter(weight)
        self.pool = nn.MaxPool2d(2,2)


  ★ def forward(self, x):
        Conv_x = self.conv(x)
        activated_x = F.relu(conv_x)
        Pooled_x = self.pool(activated_x)
        return Conv_x, activated_x, Pooled_x
```

→ Compute output of conv. layer.

pre-activation

post-activation

```
weight = torch.from_numpy(filters).unsqueeze(1).
                              type (torch.TensorFloat)

model = MyNN(weight)
```

```
nn.Conv2d(in_channels,      → input depth
          out_channels,     → output depth
          Kernel_size,      → commonly 3
          Stride = 1 ,
          padding = 0)
```

Input depth { RGB = 3
BW = 1

```
nn.MaxPool2d(Kernel_size, Stride)
↳ forward :
      x = F.relu(self.Conv1(x))
      x = self.pool(x)
```

Sequential Models

There is another way in pytorch to create CNNs ; Sequential wrapper

```
def __init__(self):
        super(ModelName, self).__init__()
        self.features = nn.Sequential(
                nn.Conv2d(1,16, 2, Stride=2,
        ①       nn.MaxPool2d(2,2),
                nn.ReLU(True),
                nn.Conv2d(16, 32,3, padding=1),
        ②       nn.MaxPool2d(2,2),
                nn.ReLU(True)
        )
```