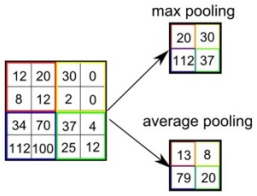
Assignment 3 (Li Yonghao)

1-2 MaxPool2d AvgPool2d

The idea is to scale the data by using kernels. These 2 functions are similar, the only difference is max is to calculate the max value of a kernel scanned area, avg is to calculate the mean. I check my results are the same as torch.




```python
class MyMaxPool2D(nn.Module):
    def __init__(self, kernel_size=(2, 2), stride=1):
        super(MyMaxPool2D, self).__init__()
        self.stride = stride
        self.kernel_size = kernel_size
        self.k_height = kernel_size[0]
        self.k_width = kernel_size[1]

    def forward(self, x):
        in_batch = x.size(0)
        in_channel = x.size(1)
        in_height = x.size(2)
        in_width = x.size(3)

        out_height = int((in_height - self.k_height) / self.stride) + 1
        out_width = int((in_width - self.k_width) / self.stride) + 1
        out = torch.zeros((in_batch, in_channel, out_height, out_width))

        for a in range(in_batch):
            for k in range(in_channel):
                for i in range(out_height):
                    for j in range(out_width):
                        start_i = i * self.stride
                        start_j = j * self.stride
                        end_i = start_i + self.k_height
                        end_j = start_j + self.k_width
                        out[a, k, i, j] = torch.max(x[a, k, start_i: end_i, start_j: end_j])
        return out
```

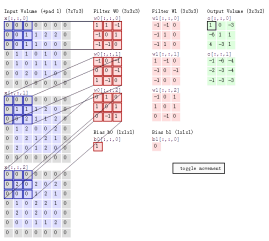
```python
class MyAveragePool2D(nn.Module):
    def __init__(self, kernel_size=(2, 2), stride=1):
        super(MyAveragePool2D, self).__init__()
        self.stride = stride
        self.kernel_size = kernel_size
        self.k_height = kernel_size[0]
        self.k_width = kernel_size[1]

    def forward(self, x):
        in_batch = x.size(0)
        in_channel = x.size(1)
        in_height = x.size(2)
        in_width = x.size(3)

        out_height = int((in_height - self.k_height) / self.stride) + 1
        out_width = int((in_width - self.k_width) / self.stride) + 1
        out = torch.zeros((in_batch, in_channel, out_height, out_width))

        for a in range(in_batch):
            for k in range(in_channel):
                for i in range(out_height):
                    for j in range(out_width):
                        start_i = i * self.stride
                        start_j = j * self.stride
                        end_i = start_i + self.k_height
                        end_j = start_j + self.k_width
                        out[a, k, i, j] = x[a, k, start_i: end_i, start_j: end_j].mean()
        return out
```

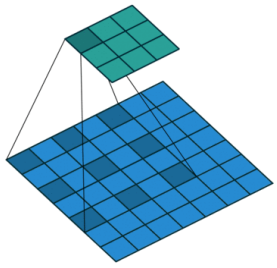
```python
print(torch.all(torch.abs(my_out - torch_out) <= 1e-6))
```

```
tensor(True)
```

3-4. Conv2d

The difference of 2 question is 4 care about dilation=2.





```python
class My_conv2d(nn.Module):
    def __init__(self, in_channels=3, out_channels=6, kernel_size=(3,3), stride=1, bias=bias):
        super(My_conv2d, self).__init__()
        self.stride = stride
        self.bias = bias
        self.kernel_size = kernel_size
        self.k_height = kernel_size[0]
        self.k_width = kernel_size[1]
        self.in_channels = in_channels
        self.out_channels = out_channels

    def forward(self, x):
        in_batch = x.size(0)
        in_channel = x.size(1)
        in_height = x.size(2)
        in_width = x.size(3)

        out_height = int((in_height - self.k_height) / self.stride) + 1
        out_width = int((in_width - self.k_width) / self.stride) + 1
        out = torch.zeros((in_batch, self.out_channels, out_height, out_width))

        for a in range(in_batch):
            for ker in range(self.out_channels):
                for k in range(in_channel):
                    for i in range(out_height):
                        for j in range(out_width):
                            start_i = i * self.stride
                            start_j = j * self.stride
                            end_i = start_i + self.k_height
                            end_j = start_j + self.k_width
                            out[a, ker, i, j] += (x[a, k, start_i: end_i, start_j: end_j]*kernels_data[ker][k]).sur
        for i in range(self.out_channels):
            out[0][i] = out[0][i]+bias[i]
        return out
```

```python
class My_conv2d(nn.Module):
    def __init__(self, in_channels=3, out_channels=6, kernel_size=(5,5), stride=2, bias=bias, dilation=2):
        super(My_conv2d, self).__init__()
        self.stride = stride
        self.bias = bias
        self.kernel_size = kernel_size
        self.k_height = kernel_size[0]+dilation-1
        self.k_width = kernel_size[1]+dilation-1
        self.h = kernel_size[0]
        self.w = kernel_size[1]
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.dilation = dilation

    def forward(self, x):
        dilation = self.dilation
        in_batch = x.size(0)
        in_channel = x.size(1)
        in_height = x.size(2)
        in_width = x.size(3)

        out_height = int((in_height - self.k_height) / self.stride) + 1
        out_width = int((in_width - self.k_width) / self.stride) + 1
        out = torch.zeros((in_batch, self.out_channels, out_height, out_width))

        kernel = torch.zeros((self.out_channels, in_channel, self.k_height, self.k_width))
        for a in range(self.out_channels):
            for k in range(in_channel):
                for i in range(self.h):
                    for j in range(self.w):
                        kernel[a, k, i*dilation, j*dilation] = kernels_data[a,k,i,j]

        for a in range(in_batch):
            for ker in range(self.out_channels):
                for k in range(in_channel):
                    for i in range(out_height):
                        for j in range(out_width):
                            start_i = i * self.stride
                            start_j = j * self.stride
                            end_i = start_i + self.k_height
                            end_j = start_j + self.k_width
                            out[a, ker, i, j] += (x[a, k, start_i: end_i, start_j: end_j]*kernel[ker][k]).sum()
        for i in range(self.out_channels):
            out[0][i] = out[0][i]+bias[i]
        return out
```

```python
print(torch.all(torch.abs(my_out - torch_out) <= 1e-6))
```

```
tensor(True)
```

5. ConvTranspose2d:

```
class My_ConvTranspose2d(nn.Module):
    def __init__(self, in_channels=3, out_channels=4, kernel_size=(3,3), stride=2, bias=bias):
        super(My_ConvTranspose2d, self).__init__()
        self.stride = stride
        self.bias = bias
        self.kernel_size = kernel_size
        self.k_height = kernel_size[0]
        self.k_width = kernel_size[1]
        self.in_channels = in_channels
        self.out_channels = out_channels

    def forward(self, x):
        in_batch = x.size(0)
        in_channel = x.size(1)
        in_height = x.size(2)
        in_width = x.size(3)

        out_height = in_height + self.k_height - 1
        out_width = in_width + self.k_width -1
        out = torch.zeros((in_batch, self.out_channels, out_height, out_width))

        for a in range(in_batch):
            for ker in range(self.out_channels):
                for k in range(in_channel):
                    for i in range(in_height):
                        for j in range(in_width):
                            out[a, ker, i:i+self.k_height, j:j+self.k_width] += x[a, k, i, j] * kernel[k][ker]
        for i in range(self.out_channels):
            out[0][i] = out[0][i]+bias[i]
        return out
```

4. batch norm

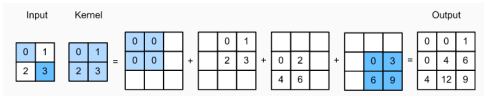
```
def my_batch_norm(input=input, running_mean=mean, running_var=var, momentum=0.1, eps=1e-05):
    mean = running_mean
    var = running_var
    mean, var = mean[:,None,None],var[:,None,None]

    return (input-mean) / torch.sqrt(var+eps)
```

```
my_out = my_batch_norm(3)
my_out
```

```
tensor([[[[-0.3766,  0.5735,  1.5725,  ..., -1.4475,  0.9262,  2.0847],
          [ 1.2692,  0.1266,  0.6571,  ...,  0.5287,  0.8677,  0.6809],
          [ 0.3547,  1.0930, -0.4347,  ...,  1.2764,  0.1840, -0.0931],
          ...,
          [-0.0596, -0.4301, -0.9783,  ...,  0.7476, -0.9466,  0.9225],
          [ 1.7942, -0.8501,  1.1719,  ...,  1.6483,  1.4476,  0.8597],
          [ 0.6685,  1.2018, -1.6216,  ..., -0.3274, -0.8482, -0.5354]],

         [[ 1.1680,  1.0046,  0.3377,  ..., -1.1417,  0.7837,  0.0826],
          [ 1.0036,  0.4145, -0.2393,  ..., -1.2774, -1.6438,  0.3282],
          [ 0.9128, -0.5835,  0.1598,  ...,  0.6143,  1.1052,  0.7001],
          ...,
          [ 0.4275, -0.6680,  0.6898,  ...,  0.1746, -0.1167,  0.2368],
          [-0.5590, -0.3695,  0.6708,  ...,  0.5934, -0.1099,  0.1781],
          [-0.2277, -0.8610, -0.1733,  ...,  1.4653, -0.5360, -0.2459]],

         [[-0.8072,  2.2740, -0.3678,  ..., -0.3133, -0.2430, -0.0277],
          [ 1.0278, -1.0807, -1.2142,  ..., -0.0095, -1.8551,  0.2439],
          [ 0.6980, -0.3608,  0.5830,  ...,  1.3056, -0.5819, -0.4674],
          ...,
          [ 0.9788,  0.9750, -0.1387,  ..., -0.7537, -1.5301, -0.7639],
          [ 0.2928, -1.4455, -1.2412,  ...,  0.4073,  0.7824,  0.4496],
          [-0.2625,  0.5905,  0.0306,  ..., -0.9942, -0.4371,  0.5810]]]])
```

```
print(torch.all(torch.abs(my_out - torch_out) <= 1e-6))
```

```
tensor(True)
```



1. flatten

```
torch_out = torch.flatten(input, start_dim=0, end_dim=-1)
torch_out
```

```
tensor([-0.4018,  0.5637,  1.5790,  ..., -1.0221, -0.4624,  0.5604])
```

```
def my_flatten(data, start, end):
    index = list(data.numpy().shape)
    if end<0:
        end = end + len(index)
    temp = 1
    for i in range(start,end+1):
        temp = temp*index[i]
    for j in range(end,start,-1):
        index.pop(j)
    index[start] = temp
    return data.reshape(index)
```

```
my_out = my_flatten(input, 0, -1)
my_out
```

```
tensor([-0.4018,  0.5637,  1.5790,  ..., -1.0221, -0.4624,  0.5604])
```

```
torch.equal(my_out, torch_out)
```

```
True
```

2. sigmoid

```
def my_sigmoid(x):
    return 1 / (1+torch.exp(-x))
```

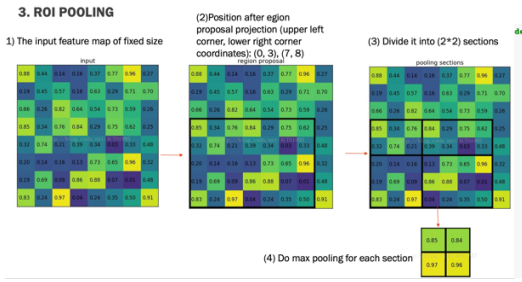
```
my_out = my_sigmoid(input)
my_out
```

```
tensor([[[[0.4009, 0.6373, 0.8291,  ..., 0.1839, 0.7155, 0.8909],
          [0.7809, 0.5274, 0.6567,  ..., 0.6267, 0.7032, 0.6621],
          [0.5845, 0.7487, 0.3868,  ..., 0.7821, 0.5419, 0.4716],
          ...,
          [0.4801, 0.3879, 0.2663,  ..., 0.6771, 0.2727, 0.7147],
          [0.8587, 0.2926, 0.7635,  ..., 0.8397, 0.8103, 0.7015],
          [0.6593, 0.7689, 0.1588,  ..., 0.4129, 0.2929, 0.3628]],

         [[0.7557, 0.7244, 0.5745,  ..., 0.2356, 0.6783, 0.5114],
          [0.7242, 0.5932, 0.4315,  ..., 0.2121, 0.1573, 0.5722],
          [0.7057, 0.3499, 0.5306,  ..., 0.6403, 0.7440, 0.6598],
          ...,
          [0.5963, 0.3310, 0.6575,  ..., 0.5343, 0.4617, 0.5498],
          [0.3555, 0.3999, 0.6532,  ..., 0.6355, 0.4634, 0.5352],
          [0.4343, 0.2898, 0.4477,  ..., 0.8356, 0.3608, 0.4299]],

         [[0.3028, 0.9056, 0.4030,  ..., 0.4163, 0.4335, 0.4872],
          [0.7329, 0.2480, 0.2239,  ..., 0.4918, 0.1316, 0.5552],
          [0.6633, 0.4047, 0.6370,  ..., 0.7839, 0.3525, 0.3792],
          ...,
          [0.7231, 0.7224, 0.4594,  ..., 0.3142, 0.1736, 0.3120],
          [0.5673, 0.1861, 0.2192,  ..., 0.5953, 0.6820, 0.6055],
          [0.4287, 0.6388, 0.5019,  ..., 0.2646, 0.3864, 0.6365]]]])
```

```
torch.equal(my_out, torch_out)
```

```
True
```

3. ROI Pool: my ppt example, the result is same as torch




```
def my_roi_pool(input=input, boxes=boxes, output_size=(7,7)):

    roi_start_h = int(boxes[0][1])
    roi_start_w = int(boxes[0][2])
    roi_end_h = int(boxes[0][3])
    roi_end_w = int(boxes[0][4])

    roi_width = (roi_end_w - roi_start_w + 1)
    roi_height = (roi_end_h - roi_start_h + 1)

    bin_size_w = roi_width / output_size[0]
    bin_size_h = roi_height / output_size[1]

    self_result = []
    for k in range(input.shape[1]):
        temp = []
        for j in range(output_size[0]):
            sub_res = []
            for i in range(output_size[1]):
                sub_res.append(input[..., int(0+bin_size_h*(j)):math.ceil(0+bin_size_h*(j+1)),
                                     int(0+bin_size_w*(i)):math.ceil(0+bin_size_w*(i+1))][0][k].max())

            temp.append(sub_res)
        self_result.append(temp)
    res = torch.Tensor(self_result)
    res = res.unsqueeze(0)
    return res
```

5. cross_entropy

```
def softmax(x):
    res = torch.zeros(x.shape)
    for i in range(len(x)):
        exp = torch.exp(x[i])
        sum = torch.sum(exp, dim=0)
        softmax = exp/sum
        res[i] = -torch.log(softmax)
    return res

def nn_loss(input, target):
    loss_list = torch.zeros([len(input)])
    for K in range(len(input)):
        loss = 0
        for i in range(target.shape[1]):
            for j in range(target.shape[2]):
                index = target[K, i, j]
                num = input[K, index, i ,j]
                loss += num
        loss_list[k] = loss / target.shape[1] / target.shape[2]
    return loss_list

def cross_entropy(input, target):
    log_softmax = softmax(input)
    loss = nn_loss(log_softmax, target)
    return loss.mean()
```

```
my_out = cross_entropy(input, target)
my_out
```

```
tensor(1.3738)
```

```
print(torch.all(torch.abs(my_out - torch_out) <= 1e-6))
```

6. mse_loss

It is the mean-square error, the formula is below:

It is the mean-square error, the formula is below. This is the result of my coding, the output is the same as torch.

```
def mse_loss(input, target):
    square = (input - target)**2
    return square.mean()
```

```
my_out = mse_loss(input, target)
my_out
```

```
tensor(2.7353)
```

```
print(torch.all(torch.abs(my_out - torch_out) <= 1e-6))
```

```
tensor(True)
```

To sum up, I calculate all the torch_out and my_out, the result are the same, it can be checked in my code.