Problem3

1

Calculate the number of parameters in Alexnet. You will have to show calculations for each layer and then sum it to obtain the total number of parameters in Alexnet. When calculating you will need to account for all the filters (size, strides, padding) at each layer. Look at Sec. 3.5 and Figure 2 in Alexnet paper (see reference). Points will only be given when explicit calculations are shown for each layer. (5)

In [16]:
```
import torch
import torchvision.models.alexnet as alexnet
model = alexnet()
```

In [17]:
```
model
```

Out[17]:
```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode
=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode
=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mod
e=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Note: The PyTorch Version Alexnet is different from the paper version!

In [13]:
```
from thop import profile
input = torch.randn(1, 3, 224, 224)
macs, params = profile(model, inputs=(input, ))
print(macs, params)
```

```
[INFO] Register count_convNd() for <class 'torch.nn.modules.conv.Conv2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.activation.ReLU'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.pooling.MaxPool2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.container.Sequentia
l'>.
[INFO] Register count_adap_avgpool() for <class 'torch.nn.modules.pooling.Ad
aptiveAvgPool2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.dropout.Dropout'>.
[INFO] Register count_linear() for <class 'torch.nn.modules.linear.Linear'>.
1429383808.0 61100840.0
```

In [7]:
```
macs
```

Out[7]:
```
714691904.0
```

For Conv Layer: (n*m*l+1)*k n: width of the input feature map m: height of the input feature map l: channel of the input feature map k: width of the kernel

Table for Alexnet Parameters Counts

For FC Layer: (n+1)*k n: number of input neurons k: number of output neurons

Layer	Filters	Kernel Size	Stride	Padding	Output Size	Parameter Count	Note
Conv 1	96	11 x 11	4		55 x 55 x 96	(11 x 11 x 3 x 96) + 96 = 34848	
Max Pool 1	-	3 x 3	2		27 x 27 x 96		
Conv2	128	5 x 5	1	2	27 x 27 x 256	((5 x 5 x 48 x 128) + 128) x 2 = 307456	Two GPU
Max Pool 2	-	3 x 3	2		13 x 13 x 256		
Conv3	384	3 x 3	1	1	13 x 13 x 384	3 x 3 x 256 x 384+384 = 885120	Cross Connection
Conv4	192 x 2	3 x 3	1	1	13 x 13 x 384	(3 x 3 x 192 x 192+192) x 2 = 663936	Only Connect to the Same GPU
Conv5	128 x 2	3 x 3	1	1	13 x 13 x 256	(3 x 3 x 192 x 128+128) x 2 = 442624	Only Connect to the Same GPU
Max Pool 3		3 x 3	2				
FC6						(6 x 6 x 128 x 2) x 4096+4096 = 37752832	
FC7						4096 x 4096 + 4096= 16781312	
FC8						4096 x 1000 + 1000 = 4097000	

Total Number

34848 + 307456 + 885120 + 663936 + 442624 + 37752832 + 16781312 + 4097000 = 60965128

Something we need to note there is that, for the original paper, there are two parts of the network in two different GPUs, so the Connection between that two parts is corssed in conv3 but seperated in conv4 and conv5.

2

VGG (Simonyan et al.) has an extremely homogeneous architecture that only performs 3x3 convolutions with stride 1 and pad 1 and 2x2 max pooling with stride 2 (and no padding) from the beginning to the end. However VGGNet is very expensive to evaluate and uses a lot more memory and parameters. Refer to VGG19 architecture on page 3 in Table 1 of the paper by Simonyan et al. You need to complete Table 1 below for calculating activation units and parameters at each layer in VGG19 (without counting biases). Its been partially filled for you. (6)

In [21]:
```
from torchvision.models.vgg import vgg19
```

VGG19

In [23]:
```
model = vgg19(pretrained=False)
```

In [24]:
```
model
```

Out[24]:
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode
=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode
=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (15): ReLU(inplace=True)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (17): ReLU(inplace=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
e=False)
    (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (22): ReLU(inplace=True)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (26): ReLU(inplace=True)
    (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
e=False)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (31): ReLU(inplace=True)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (33): ReLU(inplace=True)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (35): ReLU(inplace=True)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
e=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
```

```
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

We give the source table and finished table below.

Layer ID	Type	Patch Size (Height)	Patch Size (Width)	Stride (Height)	Stride (Width)	Output Size (Height)	Output Size (Width)	Output Size (Channel)	Memo
0	Input					224	224	3	150,528
1	Conv1_1	3	3	1	1	224	224	64	3,211,2(
2	Conv1_2	3	3	1	1	224	224	64	3,211,2(
3	Pool1	2	2	2	2	112	112	64	802,81(
4	Conv2_1	3	3	1	1	112	112	128	1,605,6
5	Conv2_2	3	3	1	1	112	112	128	1,605,6
6	Pool2	2	2	2	2	56	56	128	401,40(
7	Conv3_1	3	3	1	1	56	56	256	802,81(
8	Conv3_2	3	3	1	1	56	56	256	802,81(
9	Conv3_3	3	3	1	1	56	56	256	802,81(
10	Conv3_4	3	3	1	1	56	56	256	802,81(
11	Pool3	2	2	2	2	28	28	256	200,70(
12	Conv4_1	3	3	1	1	28	28	512	401,40(
13	Conv4_2	3	3	1	1	28	28	512	401,40(
14	Conv4_3	3	3	1	1	28	28	512	401,40(
15	Conv4_4	3	3	1	1	28	28	512	401,40(
16	Pool4	2	2	2	2	14	14	512	100,35(
17	Pool5_1	3	3	1	1	14	14	512	100,35(
18	Pool5_2	3	3	1	1	14	14	512	100,35(
19	Pool5_3	3	3	1	1	14	14	512	100,35(
20	Pool5_4	3	3	1	1	14	14	512	100,35(
21	Pool6	2	2	2	2	7	7	512	25,088
22	FC1					1	1	4096	4,096
23	FC2					1	1	4096	4,096
24	FC3					1	1	1000	1,000
Total									16,542,

VGG19

Layer	Number of Activations (Memory)	Parameters (Compute)
Input	224*224*3=150K	0
CONV3-64	224*224*64=3.2M	(3*3*3)*64=1,728
CONV3-64	224*224*64=3.2M	(3*3*64)*64=36,864

Layer	Number of Activations (Memory)	Parameters (Compute)
POOL2	112*112*64=800K	0
CONV3-128	112*112*128=1568K	(3*3*64)*128=73728
CONV3-128	112*112*128==1568K	(3*3*128)*128=147456
POOL2	56*56*128=400K	0
CONV3-256	56*56*256=800K	(3*3*128)*256=294912
CONV3-256	56*56*256=800K	(3*3*256)*256=589824
CONV3-256	56*56*256=800K	(3*3*256)*256=589824
CONV3-256	56*56*256=800K	(3*3*256)*256=589824
POOL2	56*56*128=401408	0
CONV3-512	28*28*512=400K	(3*3*256)*512=1179648
CONV3-512	28*28*512=400K	(3*3*512)*512=2359296
CONV3-512	28*28*512=400K	(3*3*512)*512=2359296
CONV3-512	28*28*512=400K	(3*3*512)*512=2359296
POOL2	28*28*256=200,704	0
CONV3-512	14*14*512=100k	(3*3*512)*512=2359296
CONV3-512	14*14*512=100k	(3*3*512)*512=2359296
CONV3-512	14*14*512=100k	(3*3*512)*512=2359296
CONV3-512	14*14*512=100k	(3*3*512)*512=2359296
POOL2	7*7*512=25088	0
FC	4096	25088*4096=102,760,448
FC	4096	4096*4096=16,777,216
FC	1000	4096*1000=4096,000
TOTAL	16,542,184	143,652,544

3

VGG architectures have smaller filters but deeper networks compared to Alexnet (3x3 compared to 11x11 or 5x5). Show that a stack of N convolution layers each of filter size F × F has the same receptive field as one convolution layer with filter of size (NF – N + 1) × (NF – N + 1). Use this to calculate the receptive field of 3 filters of size 5x5. (4)

We have

$$RF_{i+1} = RF_i + (k-1) \times S_i \tag{1}$$

$$S_i = \prod_{i=1}^{i} \text{Stride}_i \tag{2}$$

RF_{i+1} is the current receptive field and RF_i is the previous receptive field. k is the kernel size and S_i is the product of previous strides.

The stacked 3 filter convolution layer has the same receptive field as the single filter convolution layer with a filter size of (5*3-3+1)x(5*3-3+1) = (13 x 13).

4

The original Googlenet paper (Szegedy et al.) proposes two architectures for Inception module, shown in Figure 2 on page 5 of the paper, referred to as naive and dimensionality reduction respectively.

<!--

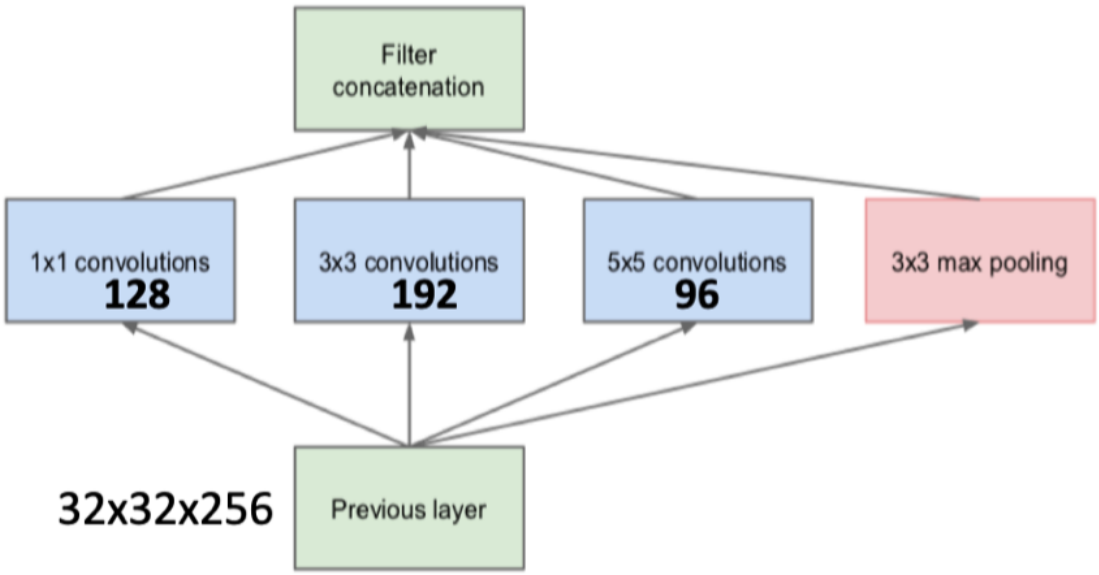
- @Author: Xiang Pan
- @Date: 2022-03-27 05:34:31
- @LastEditTime: 2022-03-27 09:17:50
- @LastEditors: Xiang Pan
- @Description:
- @FilePath: /HW3/Problem3.ipynb
- @email: xiangpan@nyu.edu --> ### (a) What is the general idea behind designing an inception module (parallel convolutional filters of different sizes with a pooling followed by concatenation) in a convolutional neural network ? (3)

Multi-branch structure: Each inception module is divided into four branches, and concatenation is performed on the output of all branches when outputting. 1*1conv makes the caculation faster.

Heterogeneous branch structure: the structure of each branch is different, mainly in the depth and kernel size of the branch. Objects of the same category may have different sizes in different pictures, so features generated by kernels of different sizes should be integrated within the same module, which is beneficial for CNN to identify objects of different sizes.

(b)

Assuming the input to inception module (referred to as ”previous layer” in Figure 2 of the paper) has size 32x32x256, calculate the output size after filter concatenation for the naive and dimensionality reduction inception architectures with number of filters given in Figure 1. (4)



(a) Inception module, naïve version

In [31]:
```
import torch
import torch.nn as nn

class NaiveInception(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1x1 = nn.Conv2d(256, 128, kernel_size=1)
        self.conv3x3 = nn.Conv2d(256, 192, kernel_size=3, padding=1)
        self.conv5x5 = nn.Conv2d(256, 96, kernel_size=5, padding=2)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)

    def forward(self, x):
        x1 = self.conv1x1(x)
        x2 = self.conv3x3(x)
        x3 = self.conv5x5(x)
        x4 = self.maxpool(x)
        print(x1.shape, x2.shape, x3.shape, x4.shape)
        return torch.cat([x1, x2, x3, x4], dim=1)

model = NaiveInception()
model(torch.randn(1, 256, 224, 224)).shape
```

```
torch.Size([1, 128, 224, 224]) torch.Size([1, 192, 224, 224]) torch.Size([1,
96, 224, 224]) torch.Size([1, 256, 224, 224])
```

Out[31]:
```
torch.Size([1, 672, 224, 224])
```

The shape is (672, 224, 224)

(c)

Next calculate the total number of convolutional operations for each of the two inception architecture again assuming the input to the module has dimensions 32x32x256 and number of filters given in Figure 1. (4)

In [42]:
```
class InceptionWithDimReduction(nn.Module):
    def __init__(self):
        super().__init__()
```

```
        self.conv1x1 = nn.Sequential(nn.Conv2d(256, 128, kernel_size=1))
        self.conv3x3 = nn.Sequential(nn.Conv2d(256, 128, kernel_size=1),
                                     nn.Conv2d(128, 192, kernel_size=3, p
        self.conv5x5 = nn.Sequential(nn.Conv2d(256, 32, kernel_size=1),
                                     nn.Conv2d(32, 96, kernel_size=5, pad
        self.poolbranch = nn.Sequential(nn.MaxPool2d(kernel_size=3, stride=1
                                     nn.Conv2d(256, 64, kernel_size=1, pa
    def forward(self, x):
        x1 = self.conv1x1(x)
        x2 = self.conv3x3(x)
        x3 = self.conv5x5(x)
        x4 = self.poolbranch(x)
        print(x1.shape, x2.shape, x3.shape, x4.shape)
        return torch.cat([x1, x2, x3, x4], dim=1)
model = InceptionWithDimReduction()
model(torch.randn(1, 256, 224, 224)).shape
```

```
torch.Size([1, 128, 224, 224]) torch.Size([1, 192, 224, 224]) torch.Size([1,
96, 224, 224]) torch.Size([1, 64, 224, 224])
```

Out[42]:
```
torch.Size([1, 480, 224, 224])
```

The **NaiveInception**

Conv Ops for **NaiveInception**:

[conv1x1 branch, conv1x1 128] 128 *224* 224 *1* 1 * 256 = 1644167168

[conv3x3 branch, conv3x3 192] 192 *224* 224 *3* 3 * 256 = 22196256768

[conv5x5 branch, conv5x5 96] 96 *224* 224 *5* 5 * 256 = 30828134400

Total Conv Ops for **NaiveInception**: 1644167168 + 22196256768 + 30828134400 = 54668558336

The **InceptionWithDimReduction** output is (480, 224, 224). 224*224*480.

Conv Ops for InceptionWithDimReduction:

[conv1x1 branch, conv1x1 128] 224 *224* 128 *1* 1 * 256 = 1644167168

[conv3x3 branch, conv1x1 128] 224 *224* 128 *1* 1 * 256 = 1644167168

[conv3x3 branch, conv3x3 192] 224 *224* 192 *3* 3 * 128 = 11098128384

[conv5x5 branch, conv1x1 32] 224 *224* 32 *1* 1 * 256 = 411041792

[conv5x5 branch, conv5x5 96] 224 *224* 96 *5* 5 * 32 = 3853516800

Total Conv Ops for InceptionWithDimReduction: 1644167168 + 1644167168 + 11098128384 + 411041792 + 3853516800 = 18651021312

(d)

Based on the calculations in part (c) explain the problem with naive architecture and how dimensionality reduction architecture helps (Hint: compare computational complexity). How much is the computational saving ? (2+2)

The dimension reduction architecture reduces the dimensionality for each branch, thus the complex convolutional operations with less input dimension, the total convolutional operations are reduced.

The naive architecture has a total of 54668558336 convolutional operations. The dimensionality reduction architecture has a total of 18651021312 convolutional operations. The dimensionality reduction architecture reduces the number of convolutional operations by a factor of 3

$$\text{ops-naive} \approx 3 * \text{ops-DimReduction} \tag{3}$$

5

Faster-RCNN is a CNN based architecture for object detection which is much faster that Fast-RCNN.

Read about Faster-RCNN in Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks and answer the following questions:

(a)

What is the main difference between Fast-RCNN and Faster-RCNN that resulted in faster detection using Faster-RCNN? (2)

Fast-RCNN: selective search for region selection, which runs in CPU and slow (but fast than multi-stage RCNN).

Faster-RCNN: Rigion Proposal Network (RPN) for region selection, CNN for feature extraction, and RoI pooling for object detection.

The main difference is using RPN for region selection and RPN shared CNN feature mapping.

(b)

What is Region Proposal Network (RPN)? Clearly explain its architecture. (2)

The input of RPN is a feature map, and the output has two branches, one is the classification (cls) layer, which is used to determine whether there is a target, and the other is the regression (reg) layer, which is used to generate a series of target proposal boxes.

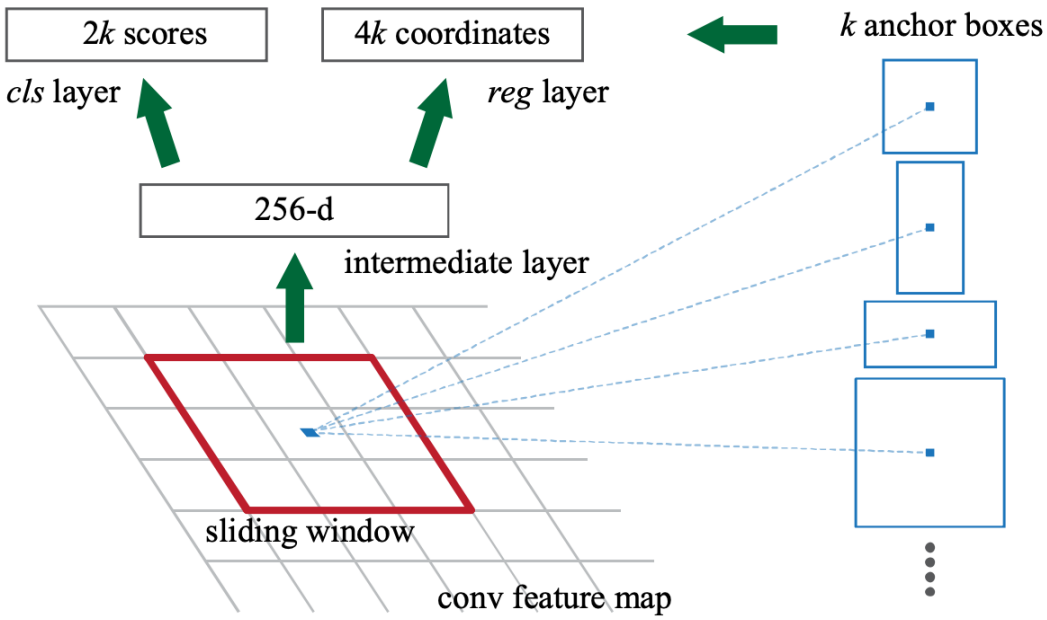
In RPN, we will use an nxn sliding window to traverse the entire feature map. Each sliding window will generate K different scales and different proportions of proposed regions, which we call Anchors. Each An anchor will generate 2k scores (target and non-target) in the clf layer, and 4k position information in the reg layer.

(c)

Explain how are region proposals generated from RPN using an example image.(3)

We use the figure 3 from Faster-RCNN paper to illustrate the RPN. CNN maps the image to a feature map, and then we use the feature map to generate a series of anchors. Each anchor is a sliding window with a different scale and different aspect ratio. The anchors are then used to generate a series of target and non-target proposals with coordinates difference.

Figure:

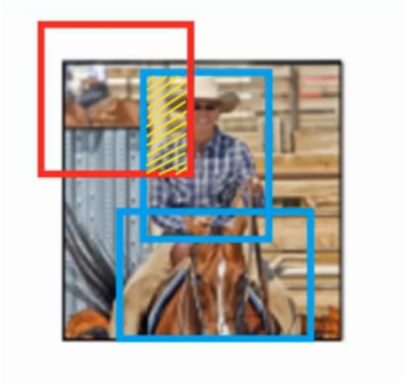


We have k types of anchors, each with different scales and different ratios. The sliding window will traverse the entire feature map, and for each sliding window, we will generate k anchors. For each anchor, we will generate 2k scores (target and non-target) in the clf layer, and 4k position information in the reg layer.

(4)

There is a lot of overlap in the region proposals generated by RPN. What technique is used in Faster-RCNN to reduce the number of proposals to roughly 2000? Explain how does this technique work using an example. (3)

Ignore Cross-boundary anchors: During training, they ignore all cross-boundary anchors so they do not contribute to the loss.



The red anch