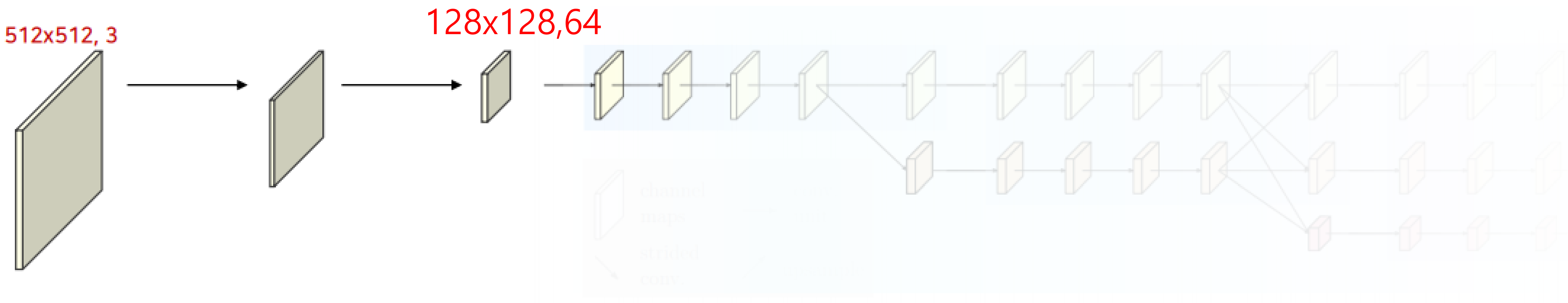
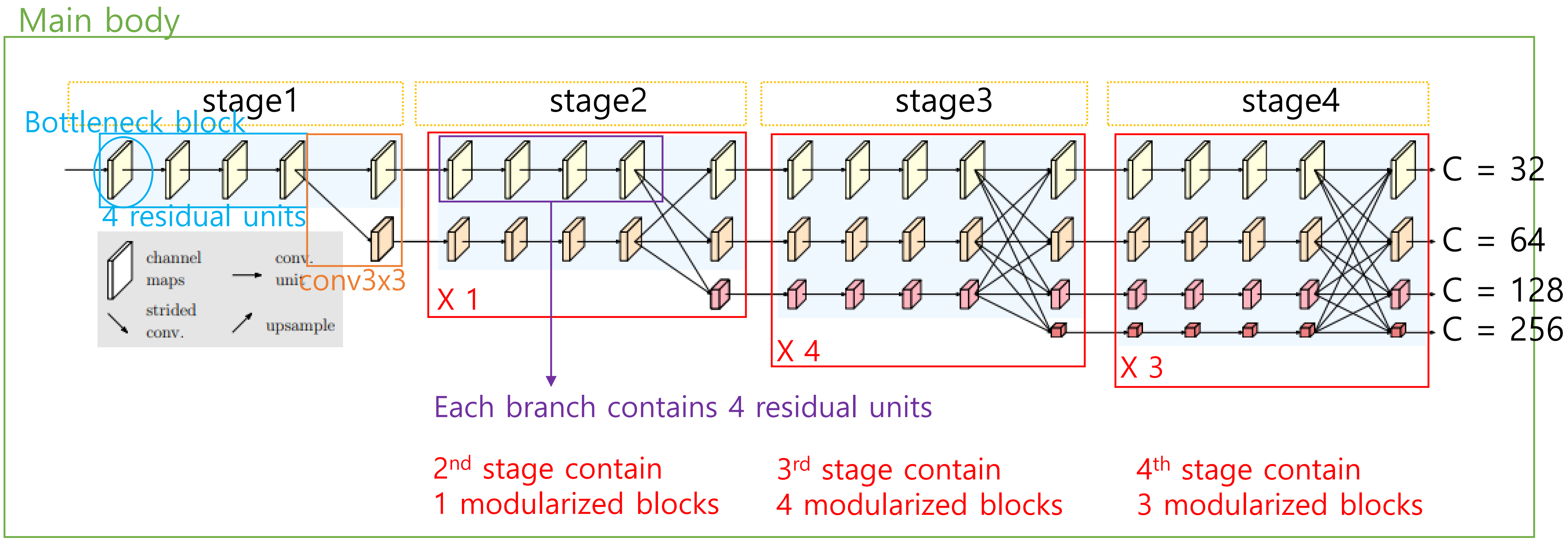
0. Stem network




```
# stem net
self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=2, padding=1, bias=False)
self.bn1 = self.norm_layer(64)
self.conv2 = nn.Conv2d(64, 64, kernel_size=3, stride=2, padding=1, bias=False)
self.bn2 = self.norm_layer(64)
self.relu = nn.ReLU(inplace=True)
```

Stride=2의 convolution을 2번 거쳐서 h/4 x w/4 크기로 만듦

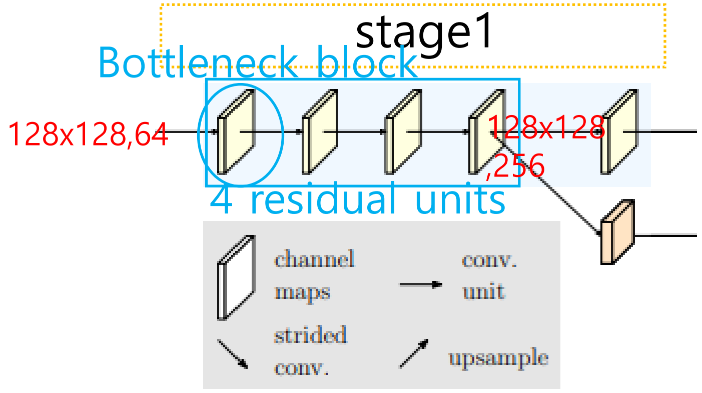


3.4 Instantiation

The main body contains four stages with four parallel convolution streams. The resolutions are 1/4, 1/8, 1/16, and 1/32. The first stage contains 4 residual units where each unit is formed by a bottleneck with the width 64, and is followed by one 3×3 convolution changing the width of feature maps to C. The 2nd, 3rd, 4th stages contain 1, 4, 3 modularized blocks, respectively. Each branch in multi-resolution parallel convolution of the modularized block contains 4 residual units. Each unit contains two 3×3 convolutions for each resolution, where each convolution is followed by batch normalization and the nonlinear activation ReLU. The widths (numbers of channels) of the

Hrnet32
C = 32

1. Main Body – stage1




```
HRNET_32.STAGE1 = CN()
HRNET_32.STAGE1.NUM_MODULES = 1
HRNET_32.STAGE1.NUM_BRANCHES = 1
HRNET_32.STAGE1.NUM_BLOCKS = [4]  # first stage contains 4 residual unit
HRNET_32.STAGE1.BLOCK = 'BOTTLENECK' # where each unit is formed by a bottleneck
HRNET_32.STAGE1.NUM_CHANNELS = [64] #  with width 64
HRNET_32.STAGE1.FUSE_METHOD = 'SUM'
```

Table 14

Resolution	Stage 1	Stage 2	Stage 3	Stage 4
$4\times$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 4 \times 1$	$\begin{bmatrix} 3\times3, C \\ 3\times3, C \end{bmatrix} \times 4 \times 1$	$\begin{bmatrix} 3\times3, C \\ 3\times3, C \end{bmatrix} \times 4 \times 4$	$\begin{bmatrix} 3\times3, C \\ 3\times3, C \end{bmatrix} \times 4 \times 3$
$8\times$		$\begin{bmatrix} 3\times3, 2C \\ 3\times3, 2C \end{bmatrix} \times 4 \times 1$	$\begin{bmatrix} 3\times3, 2C \\ 3\times3, 2C \end{bmatrix} \times 4 \times 4$	$\begin{bmatrix} 3\times3, 2C \\ 3\times3, 2C \end{bmatrix} \times 4 \times 3$
$16\times$			$\begin{bmatrix} 3\times3, 4C \\ 3\times3, 4C \end{bmatrix} \times 4 \times 4$	$\begin{bmatrix} 3\times3, 4C \\ 3\times3, 4C \end{bmatrix} \times 4 \times 3$
$32\times$				$\begin{bmatrix} 3\times3, 8C \\ 3\times3, 8C \end{bmatrix} \times 4 \times 3$

1. Main Body – stage1

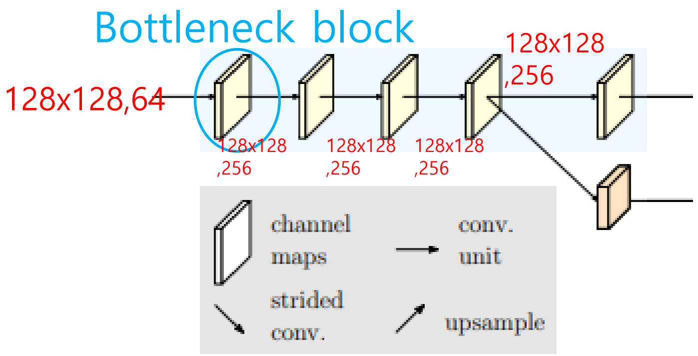
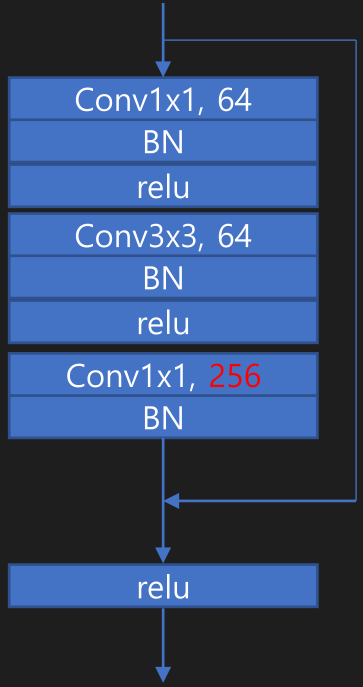


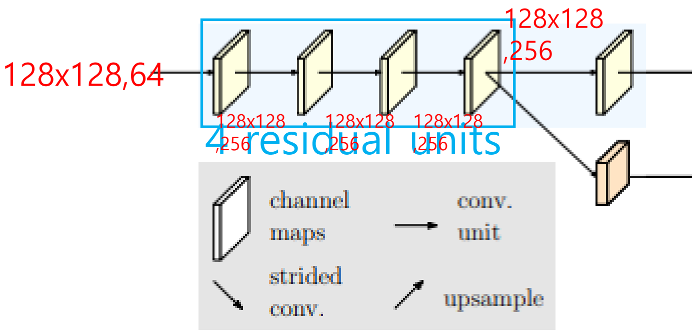
Table 14

Resolution	Stage 1	
$4\times$	$\begin{bmatrix} 1\times 1, 64 \\ 3\times 3, 64 \\ 1\times 1, 256 \end{bmatrix} \times 4 \times 1$	$\begin{bmatrix} 3 \times \\ 3 \times \end{bmatrix}$
$8\times$	Bottleneck block	$\begin{bmatrix} 3 \times \\ 3 \times \end{bmatrix}$
$16\times$		
$32\times$		

```python
class Bottleneck(nn.Module):

    expansion = 4

    def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
                 base_width=64, dilation=1, norm_layer=None):
        super(Bottleneck, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        width = int(planes * (base_width / 64.)) * groups
        # Both self.conv2 and self.downsample layers downsample the input when stride != 1
        self.conv1 = conv1x1(inplanes, width)
        self.bn1 = norm_layer(width)
        self.conv2 = conv3x3(width, width, stride, groups, dilation)
        self.bn2 = norm_layer(width)
        self.conv3 = conv1x1(width, planes * self.expansion)
        self.bn3 = norm_layer(planes * self.expansion)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out
```



1. Main Body – stage1



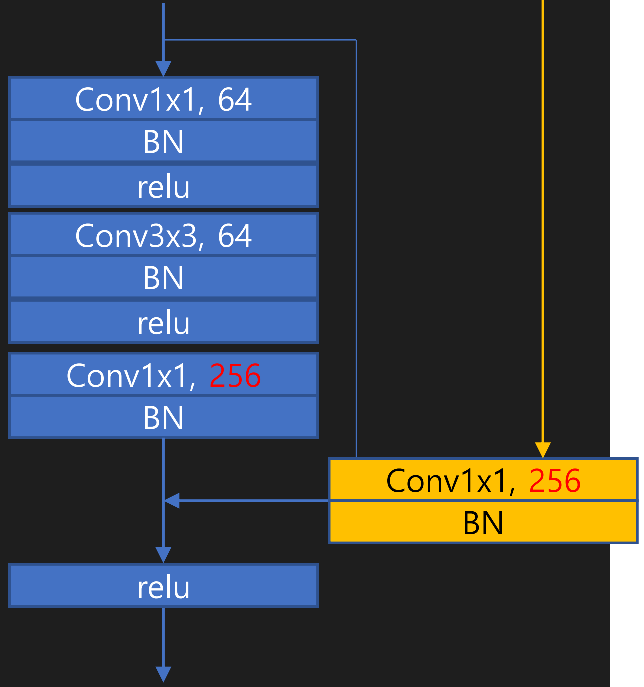

```python
def _make_layer(self, block, inplanes, planes, blocks, stride=1):
    downsample = None
    if stride != 1 or inplanes != planes * block.expansion:
        downsample = nn.Sequential(
            nn.Conv2d(inplanes, planes * block.expansion,
                      kernel_size=1, stride=stride, bias=False),
            self.norm_layer(planes * block.expansion),
        )

    layers = []
    layers.append(block(inplanes, planes, stride, downsample, norm_layer=self.norm_layer))

    inplanes = planes * block.expansion
    for _ in range(1, blocks): # 나머지 3개 unit마다 class Bottleneck(nn.Module) 객체를 만들어서 이어줌
        layers.append(block(inplanes, planes, norm_layer=self.norm_layer))

    return nn.Sequential(*layers)
```

첫 번째 block은 입력 채널(64)과 출력채널(256)을 맞추기 위해 conv1x1 추가

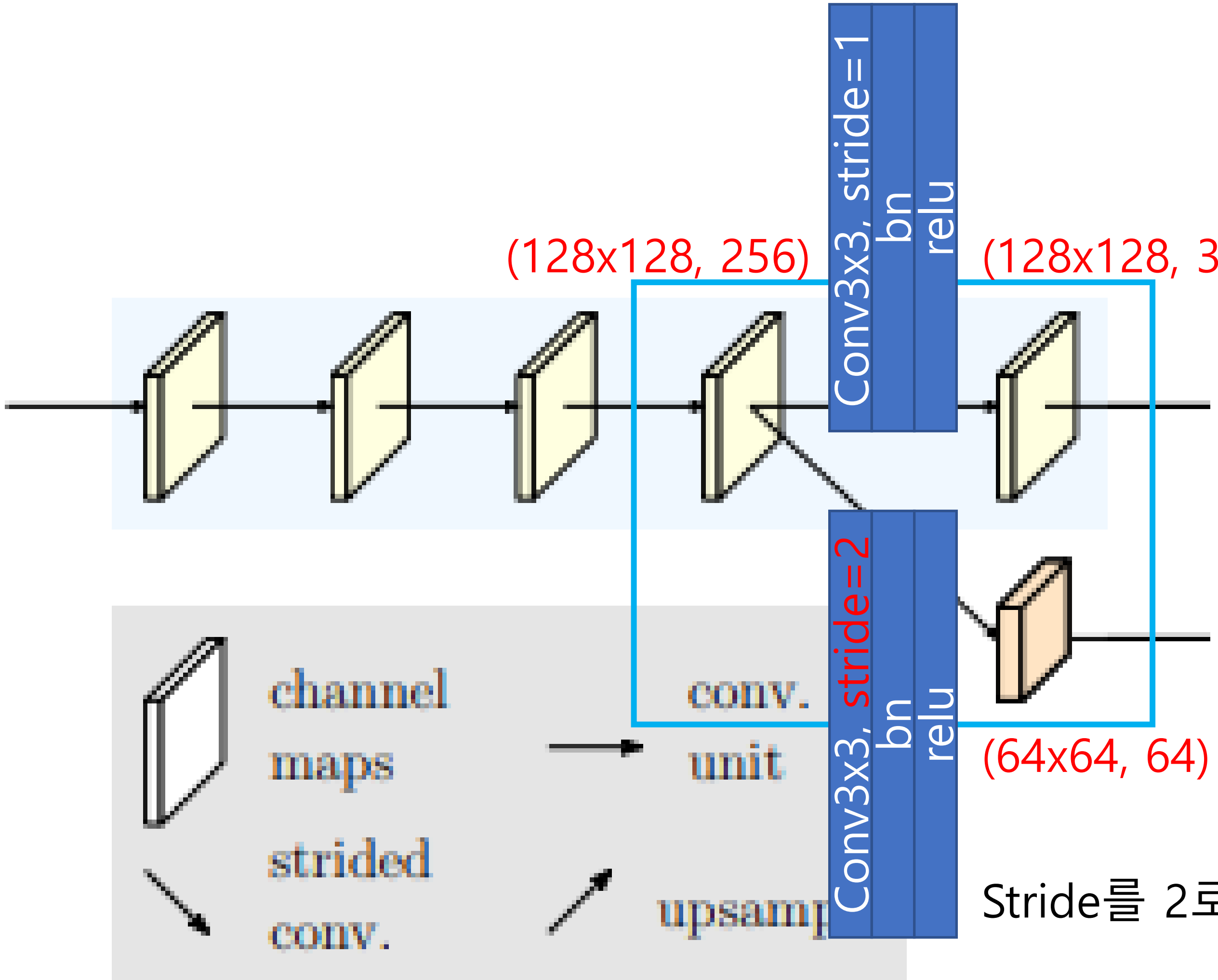
Table 14

Resolution	Stage 1	
4×	$\left[\begin{array}{c} 1\times 1, 64 \\ 3\times 3, 64 \\ 1\times 1, 256 \end{array}\right] \times 4 \times 1$	$\left[\begin{array}{c} 3 \times \\ 3 \times \end{array}\right.$
8×		$\left[\begin{array}{c} 3 \times \\ 3 \times \end{array}\right.$
16×		
32×		

```python
def forward(self, x):
    identity = x

    out = self.conv1(x)
    out = self.bn1(out)
    out = self.relu(out)

    out = self.conv2(out)
    out = self.bn2(out)
    out = self.relu(out)

    out = self.conv3(out)
    out = self.bn3(out)

    if self.downsample is not None:
        identity = self.downsample(x)

    out += identity
    out = self.relu(out)

    return out
```



1. Main Body – stage1



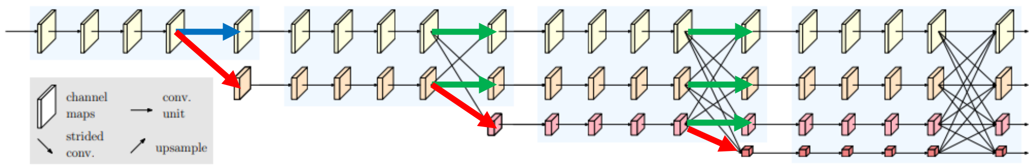
Hrnet32니까 32로 설정

채널 수는 2배 증가

Stride를 2로 줘서 resolution을 반으로 줄임

1. Main Body – stage1

$$\begin{array}{ccccccc} \mathcal{N}_{11} & \rightarrow & \mathcal{N}_{21} & \rightarrow & \mathcal{N}_{31} & \rightarrow & \mathcal{N}_{41} \\ & \searrow & \mathcal{N}_{22} & \rightarrow & \mathcal{N}_{32} & \rightarrow & \mathcal{N}_{42} \\ & & & \searrow & \mathcal{N}_{33} & \rightarrow & \mathcal{N}_{43} \\ & & & & & \searrow & \mathcal{N}_{44}, \end{array}$$



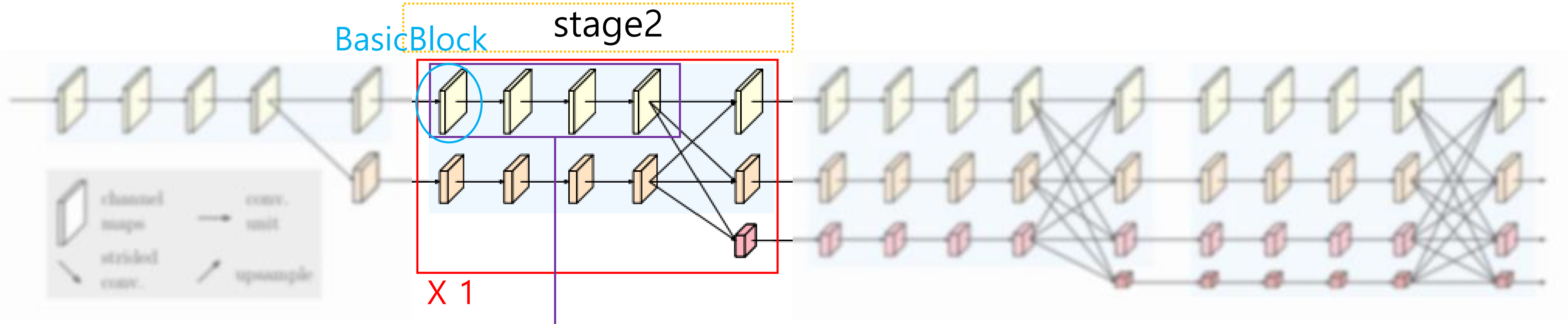

```
def _make_transition_layer(self, num_channels_pre_layer, num_channels_cur_layer):

    num_branches_cur = len(num_channels_cur_layer) # num_channels_cur_layer : [32, 64]
    num_branches_pre = len(num_channels_pre_layer) # num_channels_pre_layer : [64]

    transition_layers = []
    for i in range(num_branches_cur):
        if i < num_branches_pre:
            if num_channels_cur_layer[i] != num_channels_pre_layer[i]:
                transition_layers.append(nn.Sequential(
                    nn.Conv2d(num_channels_pre_layer[i],
                              num_channels_cur_layer[i],
                              3,
                              1,
                              1,
                              bias=False),
                    self.norm_layer(num_channels_cur_layer[i]),
                    nn.ReLU(inplace=True)))
            else:
                transition_layers.append(None)
        else:
            conv3x3s = []
            for j in range(i+1-num_branches_pre):
                inchannels = num_channels_pre_layer[-1]
                outchannels = num_channels_cur_layer[i] \
                    if j == i-num_branches_pre else inchannels
                conv3x3s.append(nn.Sequential(
                    nn.Conv2d(
                        inchannels, outchannels, 3, 2, 1, bias=False),
                    self.norm_layer(outchannels),
                    nn.ReLU(inplace=True)))
            transition_layers.append(nn.Sequential(*conv3x3s))

    return nn.ModuleList(transition_layers)
```

Resolution 그대로
& Channel 수 변환

Resolution 그대로
& Channel 수 그대로

Resolution 절반
& Channel 수 두 배

1. Main Body – stage2

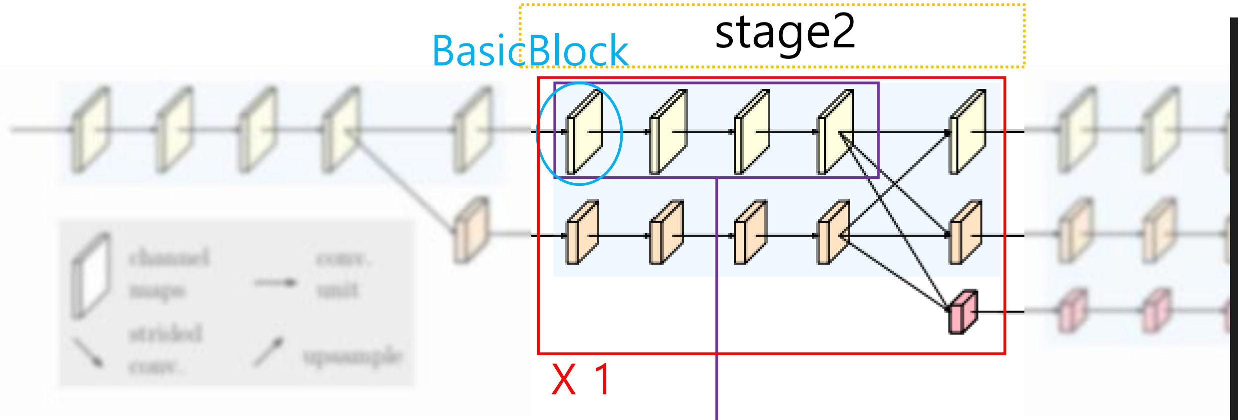


Each branch contains 4 residual units

2nd stage contain
1 modularized blocks

Resolution	Stage 1	Stage 2	Stage 3	Stage 4
4×	[illegible]	$\begin{bmatrix} 3\times3, C \\ 3\times3, C \end{bmatrix} \times 4 \times 1$	[illegible]	[illegible]
8×		$\begin{bmatrix} 3\times3, 2C \\ 3\times3, 2C \end{bmatrix} \times 4 \times 1$	[illegible]	[illegible]
16×			[illegible]	[illegible]
32×				[illegible]

1. Main Body – stage2



Each branch contains 4 residual units

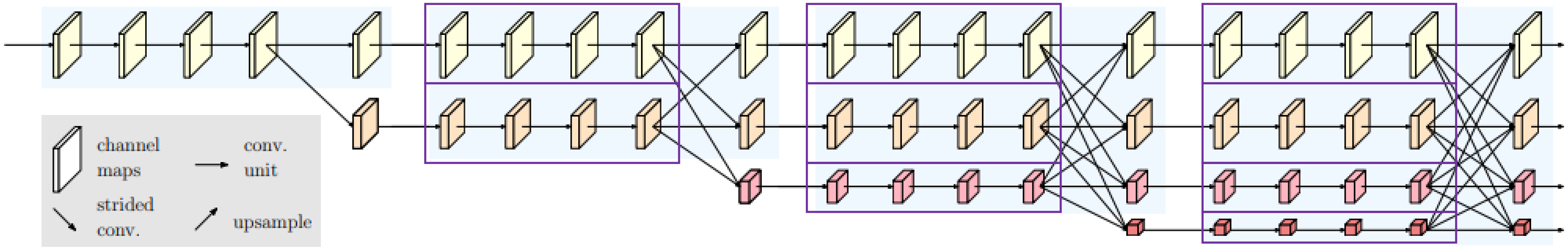
2nd stage contain
1 modularized blocks

Resolution	Stage 2
4×	$\begin{bmatrix} 3\times3, C \\ 3\times3, C \end{bmatrix} \times 4 \times 1$
8×	$\begin{bmatrix} 3\times3, 2C \\ 3\times3, 2C \end{bmatrix} \times 4 \times 1$
16×	
32×	

```
def _make_stage(self, layer_config, num_inchannels,
                multi_scale_output=True):
    num_modules = layer_config['NUM_MODULES']
    num_branches = layer_config['NUM_BRANCHES']
    num_blocks = layer_config['NUM_BLOCKS']
    num_channels = layer_config['NUM_CHANNELS']
    block = blocks_dict[layer_config['BLOCK']]
    fuse_method = layer_config['FUSE_METHOD']

    modules = []
    for i in range(num_modules):
        # multi_scale_output is only used last module
        if not multi_scale_output and i == num_modules - 1:
            reset_multi_scale_output = False
        else:
            reset_multi_scale_output = True

        modules.append(
            HighResolutionModule(num_branches,
                                 block,
                                 num_blocks,
                                 num_inchannels,
                                 num_channels,
                                 fuse_method,
                                 reset_multi_scale_output,
                                 norm_layer=self.norm_layer)
        )
        num_inchannels = modules[-1].get_num_inchannels()

    return nn.Sequential(*modules), num_inchannels
```

Stage2: 1개
Stage3: 4개
Stage4: 3개

1. Main Body – stage2



Branch 2개

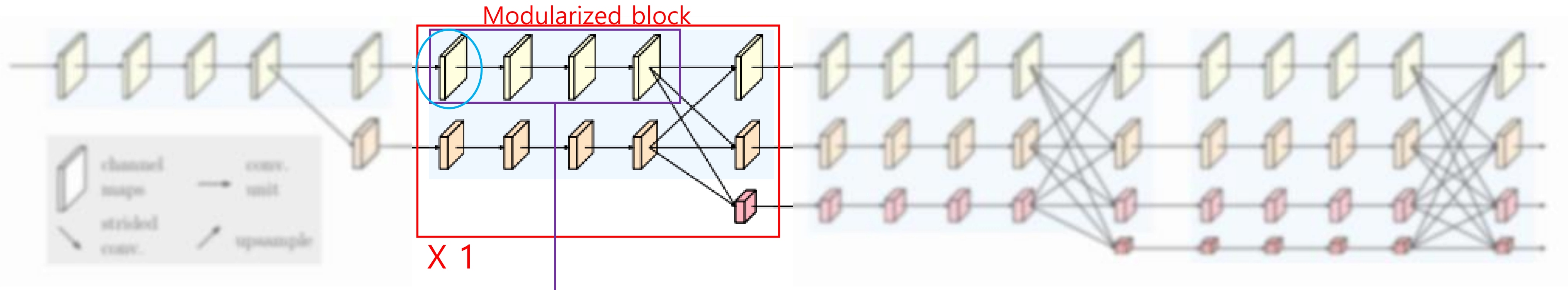
Branch 3개

Branch 4개

stage2에서는 2개,
stage3에서는 3개,
Stage4에서는 4개의 branch를 더함

```
class HighResolutionModule(nn.Module):
    ...
    def _make_branches(self, num_branches, block, num_blocks, num_channels):
        branches = []

        for i in range(num_branches):
            branches.append(
                self._make_one_branch(i, block, num_blocks, num_channels))

        return nn.ModuleList(branches)
```

1. Main Body – stage2



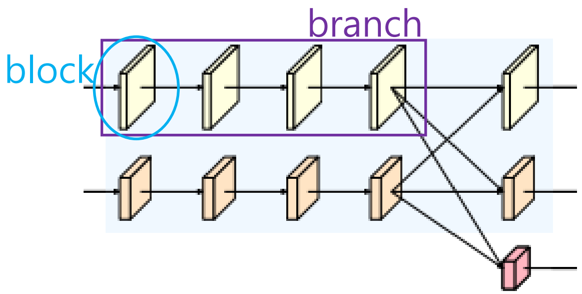
Each branch contains 4 residual units

2nd stage contain
1 modularized blocks

stage2에서는 2개,
stage3에서는 3개,
Stage4에서는 4개의
branch(BasicBlock)을 더해줌

```
class HighResolutionModule(nn.Module):   Modularized block
    ...

    def _make_branches(self, num_branches, block, num_blocks, num_channels):
        branches = []

        for i in range(num_branches):
            branches.append(
                self._make_one_branch(i, block, num_blocks, num_channels))

        return nn.ModuleList(branches)
```

1. Main Body – stage2



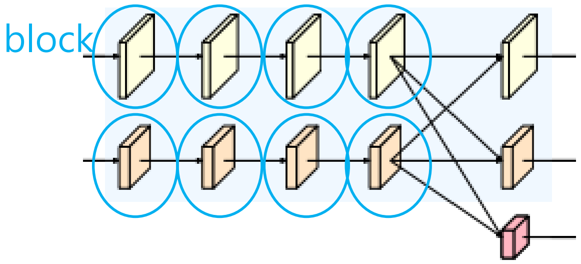

```python
class HighResolutionModule(nn.Module):
    ...
    def _make_one_branch(self, branch_index, block, num_blocks, num_channels,
                         stride=1):
        downsample = None
        if stride != 1 or \
                self.num_inchannels[branch_index] != num_channels[branch_index] * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.num_inchannels[branch_index],
                          num_channels[branch_index] * block.expansion,
                          kernel_size=1, stride=stride, bias=False),
                self.norm_layer(num_channels[branch_index] * block.expansion),
            )

        layers = []
        layers.append(block(self.num_inchannels[branch_index],
                             num_channels[branch_index], stride, downsample, norm_layer=self.norm_layer))
        self.num_inchannels[branch_index] = \
            num_channels[branch_index] * block.expansion
        for i in range(1, num_blocks[branch_index]):
            layers.append(block(self.num_inchannels[branch_index],
                                num_channels[branch_index], norm_layer=self.norm_layer))

        return nn.Sequential(*layers)
```

branch 하나

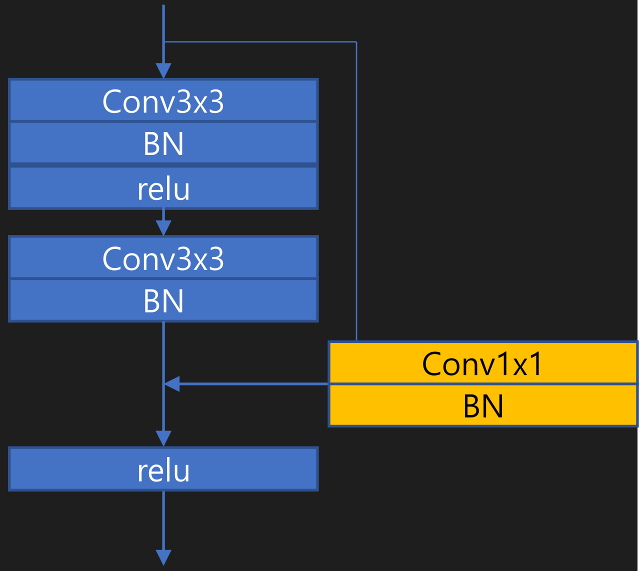
- residual unit (BasicBlock) 4개로 이루어짐
- Each branch in multi-resolution parallel convolution of the modularized block contains **4 residual units**.

1. Main Body – stage2

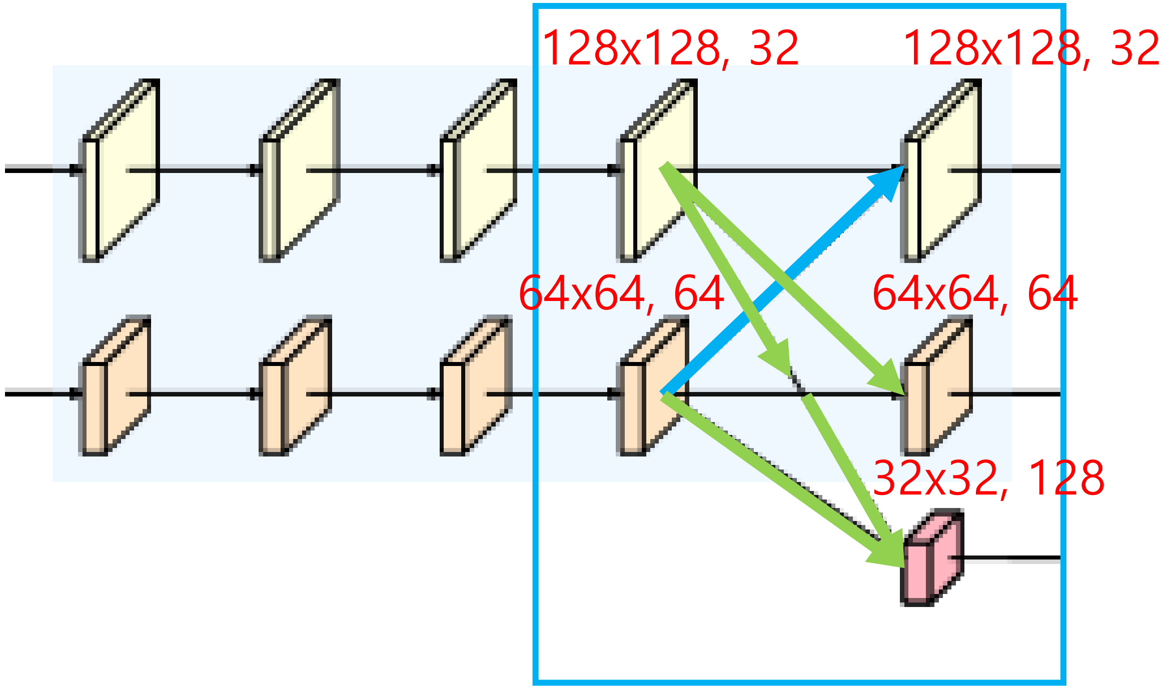
residual unit (BasicBlock)




```python
class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
                 base_width=64, dilation=1, norm_layer=None):
        super(BasicBlock, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        if groups != 1 or base_width != 64:
            raise ValueError('BasicBlock only supports groups=1 and base_width=64')
        if dilation > 1:
            raise NotImplementedError("Dilation > 1 not supported in BasicBlock")
        # Both self.conv1 and self.downsample layers downsample the input when stride != 1
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x

        out = self.conv1(x) # 3x3
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out) # 3x3
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out
```



1. Main Body – stage2



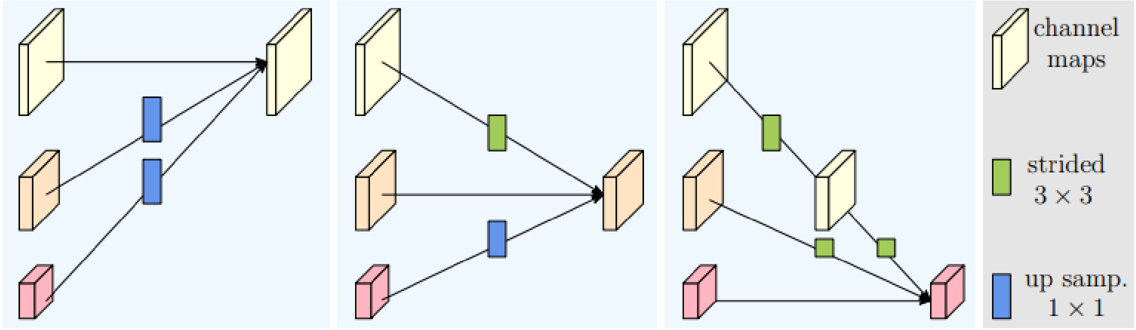


Fig. 3. Illustrating how the fusion module aggregates the information for high, medium and low resolutions from left to right, respectively. Right legend: strided 3×3 = stride-2 3×3 convolution, up samp. 1×1 = bilinear upsampling followed by a 1×1 convolution.

- Strided Convolution으로 하위 stream 생성
- Bilinear upsampling → 1x1 Convolution으로 상위 stream 생성

1. Main Body – stage2

```python
def _make_fuse_layers(self): # high resolution으로 만들때는 bilinear->conv1x1, low resolution으로 만들때는 conv3x3 (stride=2)
    if self.num_branches == 1:
        return None

    num_branches = self.num_branches
    num_inchannels = self.num_inchannels
    fuse_layers = []
    for i in range(num_branches if self.multi_scale_output else 1):
        fuse_layer = []
        for j in range(num_branches):
            if j > i: # up samp. 1 × 1 = bilinear upsampling followed by a 1 × 1 convolution.
                fuse_layer.append(nn.Sequential(
                    nn.Conv2d(num_inchannels[j],
                              num_inchannels[i],
                              1,
                              1,
                              0,
                              bias=False),
                    self.norm_layer(num_inchannels[i])))
            elif j == i:
                fuse_layer.append(None)
            else:# strided 3 × 3 = stride-2 3 × 3 convolution
                conv3x3s = []
                for k in range(i-j):
                    if k == i - j - 1:
                        num_outchannels_conv3x3 = num_inchannels[i]
                        conv3x3s.append(nn.Sequential(
                            nn.Conv2d(num_inchannels[j],
                                      num_outchannels_conv3x3,
                                      3, 2, 1, bias=False),
                            self.norm_layer(num_outchannels_conv3x3)))
                    else:
                        num_outchannels_conv3x3 = num_inchannels[j]
                        conv3x3s.append(nn.Sequential(
                            nn.Conv2d(num_inchannels[j],
                                      num_outchannels_conv3x3,
                                      3, 2, 1, bias=False),
                            self.norm_layer(num_outchannels_conv3x3),
                            nn.ReLU(inplace=True)))
                fuse_layer.append(nn.Sequential(*conv3x3s))
        fuse_layers.append(nn.ModuleList(fuse_layer))

    return nn.ModuleList(fuse_layers)
```

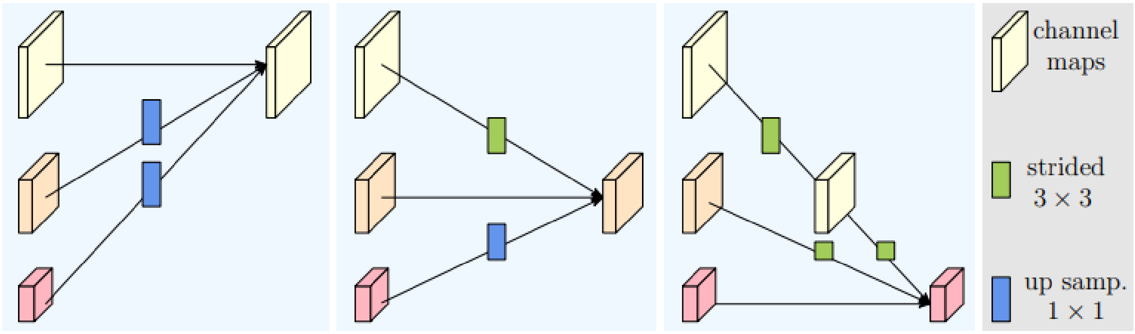
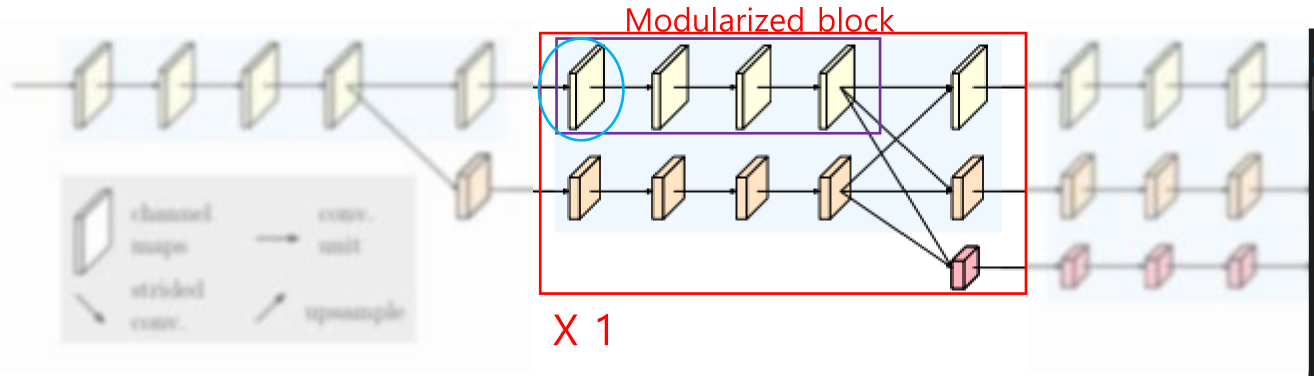


Fig. 3. Illustrating how the fusion module aggregates the information for high, medium and low resolutions from left to right, respectively. Right legend: strided 3×3 = stride-2 3×3 convolution, up samp. 1×1 = bilinear upsampling followed by a 1×1 convolution.

1. Main Body – stage2




```
class HighResolutionModule(nn.Module): Modularized block
    ...
 def forward(self, x):
    if self.num_branches == 1:
        return [self.branches[0](x[0])]

    for i in range(self.num_branches):
        x[i] = self.branches[i](x[i])

    x_fuse = []
    for i in range(len(self.fuse_layers)):
        y = x[0] if i == 0 else self.fuse_layers[i][0](x[0])
        for j in range(1, self.num_branches):
            if i == j:
                y = y + x[j]
            elif j > i:
                width_output = x[i].shape[-1]
                height_output = x[i].shape[-2]
                y = y + F.interpolate(
                    self.fuse_layers[i][j](x[j]),
                    size=[height_output, width_output],
                    mode='bilinear',
                    align_corners=True
                    )
            else:
                y = y + self.fuse_layers[i][j](x[j])
        x_fuse.append(self.relu(y))

    return x_fuse
```

2. Last layer

```python
# Upsampling
x0_h, x0_w = x[0].size(2), x[0].size(3)
x1 = F.interpolate(x[1], size=(x0_h, x0_w), mode='bilinear', align_corners=False)
x2 = F.interpolate(x[2], size=(x0_h, x0_w), mode='bilinear', align_corners=False)
x3 = F.interpolate(x[3], size=(x0_h, x0_w), mode='bilinear', align_corners=False)

x = torch.cat([x[0], x1, x2, x3], 1)

x = self.last_layer(x) # conv1x1(c->c) -> bn -> relu -> conv1x1(c->19)

return x
```

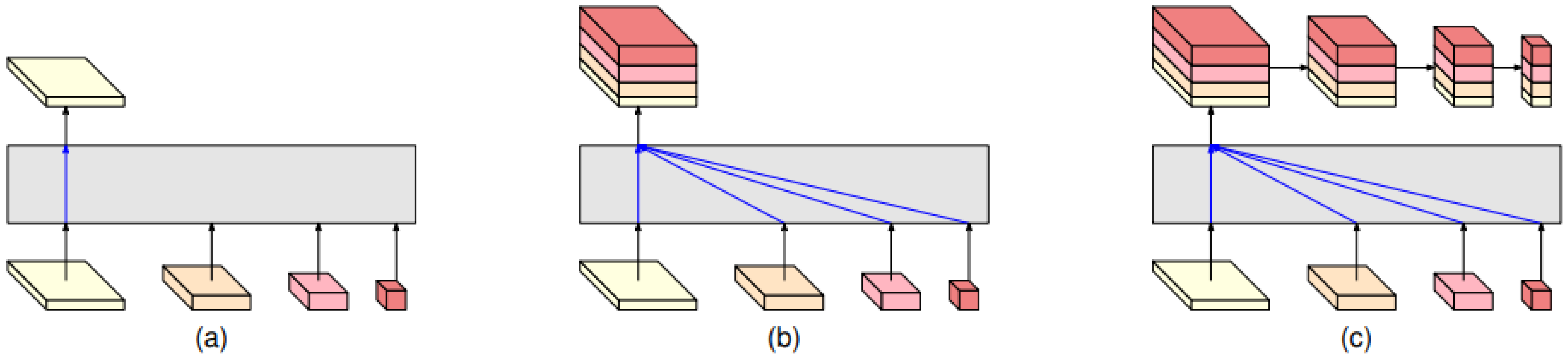


Fig. 4. (a) HRNetV1: only output the representation from the high-resolution convolution stream. (b) HRNetV2: Concatenate the (upsampled) representations that are from all the resolutions (the subsequent 1×1 convolution is not shown for clarity). (c) HRNetV2p: form a feature pyramid from the representation by HRNetV2. The four-resolution representations at the bottom in each sub-figure are outputted from the network in Figure 2, and the gray box indicates how the output representation is obtained from the input four-resolution representations.