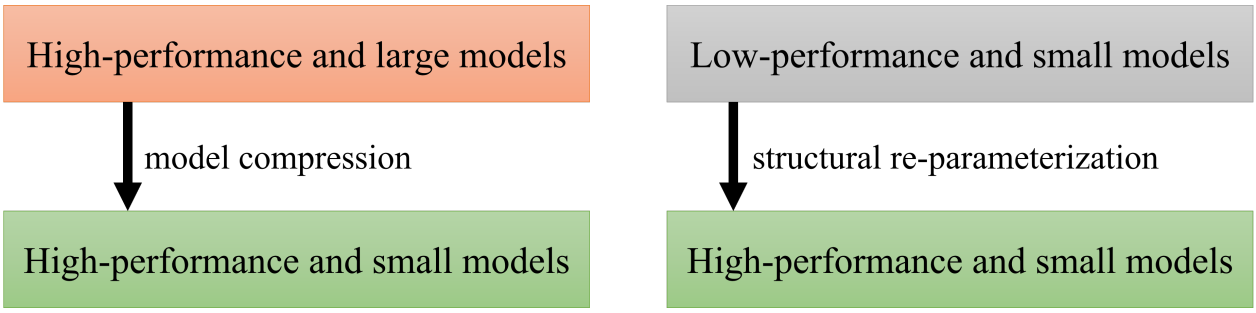
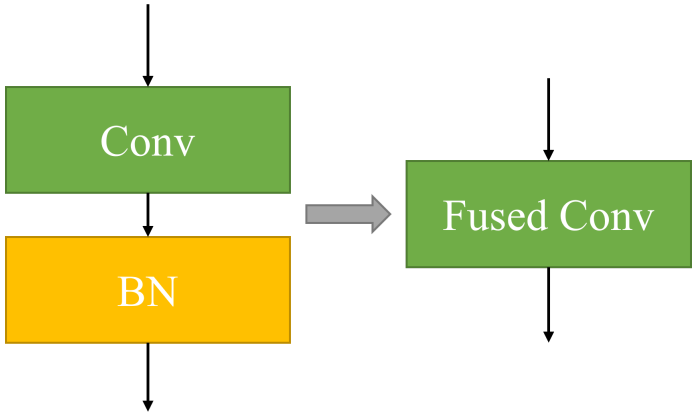
An Introduction to Structural Re-parameterization

A structure A corresponds to a set of parameters X, and a structure B corresponds to a set of parameters Y. If we can equivalently convert X to Y, we can equivalently convert structure A to B. This phenomenon inspired the birth of **Structural Re-parameterization**. Starting from a simple model, a higher-performance model can be trained by replacing its simple structures with re-parameterizable structures. After the training is completed, the re-parameterizable structures in the model will be equivalently converted back to the original simple structures. Thus, the inference burden of the model will not be increased. Structural re-parameterization might constitute a dual approach with traditional model compression methods, such as network pruning and quantization, as shown in the figure below.



This document comprehensively introduces various structural re-parameterization paradigms, including common paradigms, auxiliary paradigms, and other interesting ones for ConvNets.

1. A conv layer and its subsequent BN layer



Define the input feature map as $I \in \mathbb{R}^{C \times H_I \times W_I}$, the output feature map as $O \in \mathbb{R}^{D \times H_O \times W_O}$, the weight of the conv layer as $F \in \mathbb{R}^{D \times C \times K_1 \times K_2}$, the bias of the conv layer as $B \in \mathbb{R}^D$ (when the conv layer precedes the BN layer, it is generally necessary to disable the bias of the conv layer, because the presence of two biases at the same time is redundant, we disable the bias of the conv layer by default when deriving formulas and keep this bias when giving code to facilitate your understanding and to enable you to use more error-tolerant code). Define the running mean, running var, weight, and bias of the BN layer as μ, σ, γ, and β. The forward propagation before structural re-parameterization can be described as:

$$O = (F * I - \mu) \times \frac{\gamma}{\sigma} + \beta,$$

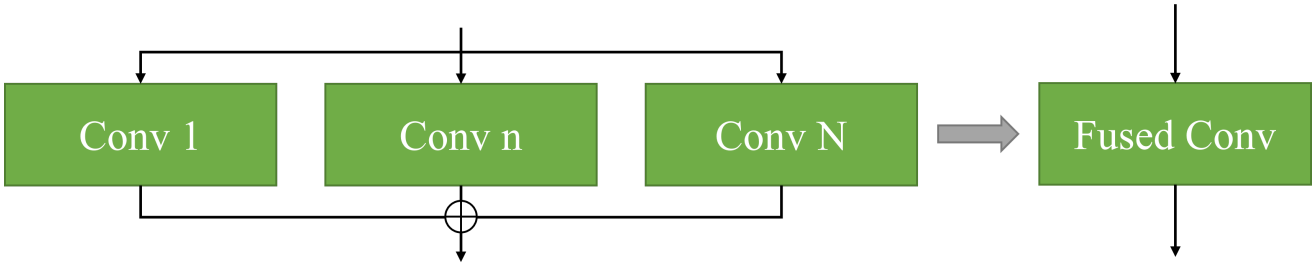
where $*$ represents the convolutional operation. According to the associative axiom of the convolutional operation, the above equation can have the following transformations:

$$O = (F \times \frac{\gamma}{\sigma}) * I - \mu \times \frac{\gamma}{\sigma} + \beta.$$

Thus a conv layer and its subsequent BN layers can be merged into a single conv layer, whose forward propagation can be expressed as $O = \hat{F} * I + \hat{B}$, where $\hat{F} = F \times \frac{\gamma}{\sigma}$ and $\hat{B} = -\mu \times \frac{\gamma}{\sigma} + \beta$. The code for merging a conv layer and its subsequent BN layer is as follows.

```
def fuse_conv_bn_weights(conv_w, conv_b, bn_rm, bn_rv, bn_eps, bn_w, bn_b):
    r"""
    Args:
        conv_w: conv weight
        conv_b: conv bias
        bn_rm: bn running mean
        bn_rv: bn running var
        bn_eps: bn eps
        bn_w: bn wright
        bn_b: bn bias
    Return: fused conv weight; fused conv bias
    """
    if conv_b is None:
        conv_b = bn_rm.new_zeros(bn_rm.shape)
    bn_var_rsqrt = torch.rsqrt(bn_rv + bn_eps)
    conv_w = conv_w * (bn_w * bn_var_rsqrt).reshape([-1] + [1] * (len(conv_w.shape) - 1))
    conv_b = (conv_b - bn_rm) * bn_var_rsqrt * bn_w + bn_b
    return conv_w, conv_b
```

2. Parallel conv layers for branch addition



Define the input feature map as $I \in \mathbb{R}^{C \times H_I \times W_I}$, the output feature map as $O \in \mathbb{R}^{D \times H_O \times W_O}$, the weight of the ith conv layer as $F_i \in \mathbb{R}^{D \times C \times K_1 \times K_2}$, the bias of the ith conv layer as $B_i \in \mathbb{R}^D$. For N parallel conv layers for branch addition, the forward propagation before structural re-parameterization can be described as:

$$O = \sum_{i=1}^{N} (F_i * I + B_i).$$

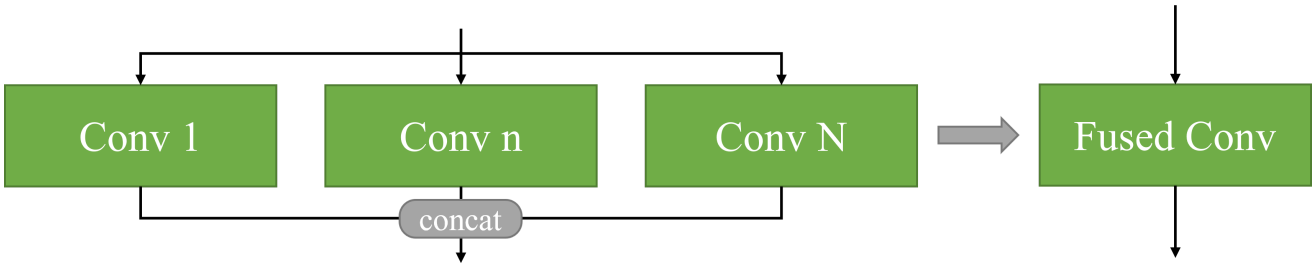
According to the associative axiom of the convolutional operation, the above equation can have the following transformations:

$$O = \sum_{i=1}^{N} F_i * I + \sum_{i=1}^{N} B_i.$$

Thus N parallel conv layers for branch addition can be merged into a single conv layer, whose forward propagation can be expressed as $O = \hat{F} * I + \hat{B}$, where $\hat{F} = \sum_{i=1}^{N} F_i$ and $\hat{B} = \sum_{i=1}^{N} B_i$. The code for merging N parallel conv layers for branch addition is as follows.

```
def branch_add(conv_weights, conv_biases):
    r"""
    Args:
        conv_weights: several kernel weights (array)
        conv_biases: several kernel biases (array)
    Return: fused conv weight; fused bias weight
    """
    assert len(conv_weights) == len(conv_biases)
    fused_weight, fused_bias = conv_weights[0], conv_biases[0]
    for i in range(1, len(conv_weights)):
        fused_weight += conv_weights[i]
        fused_bias += conv_biases[i]
    return fused_weight, fused_bias
```

3. Parallel conv layers for depth concatenation



Define the input feature map as $I \in \mathbb{R}^{C\times H_I\times W_I}$, the output feature map as $O \in \mathbb{R}^{N\cdot D\times H_O\times W_O}$, the weight of the ith conv layer as $F_i \in \mathbb{R}^{D\times C\times K_1\times K_2}$, the bias of the ith conv layer as $B_i \in \mathbb{R}^D$. For N parallel conv layers for depth concatenation, the forward propagation before structural re-parameterization can be described as:

$$O = \mathrm{cat}(F_1 * I + B_1, F_2 * I + B_2, \ldots, F_N * I + B_N),$$

where cat represents the depth concatenation operation. According to the associative axiom of the convolutional operation, the above equation can have the following transformations:

$$O = \mathrm{cat}(F_1, F_2, \ldots, F_N) * I + \mathrm{cat}(B_1, B_2, \ldots, B_N).$$

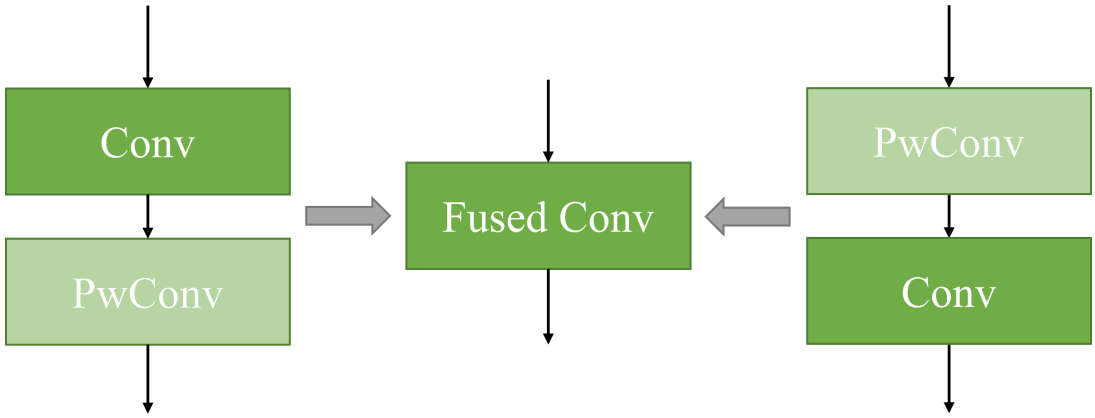
Thus N parallel conv layers for depth concatenation can be merged into a single conv layer, whose forward propagation can be expressed as $O = \hat{F} * I + \hat{B}$, where $\hat{F} = \mathrm{cat}(F_1, F_2, \ldots, F_N)$ and $\hat{B} = \mathrm{cat}(B_1, B_2, \ldots, B_N)$. The code for merging N parallel conv layers for depth concatenation is as follows.

```
def depth_concat(conv_weights, conv_biases):
    r"""
    Args:
        conv_weights: several kernel weights (array)
        conv_biases: several kernel biases (array)
    Return: fused conv weight; fused bias weight
    """
    assert len(conv_weights) == len(conv_biases)
    return torch.cat(conv_weights, dim=0), torch.cat(conv_biases)
```

4. Parallel conv layers of different sizes

The way for merging parallel conv layers of different sizes is the same as Paradigm 2 and Paradigm 3 in principle. Still, the only difference is that all kernels need to be padded with zero to the same size as the largest. Use `torch.nn.functional.pad(input, pad, mode, value)` to facilitate this operation.

5. A pointwise conv layer and a standard conv layer



Consider first the case where two conv layers are adjacent to each other. Define the input feature map as $I \in \mathbb{R}^{C \times H_I \times W_I}$, the output feature map as $O \in \mathbb{R}^{E \times H_O \times W_O}$, the weight of the first conv layer as $F_1 \in \mathbb{R}^{D \times C \times K_1 \times K_2}$, the bias of the first conv layer as $B_1 \in \mathbb{R}^D$, the weight of the second conv layer as $F_2 \in \mathbb{R}^{E \times D \times K_1 \times K_2}$, the bias of the second conv layer as $B_2 \in \mathbb{R}^E$, the forward propagation before structural re-parameterization can be described as:

$$O = (I * F_1 + B_1) * F_2 + B_2.$$

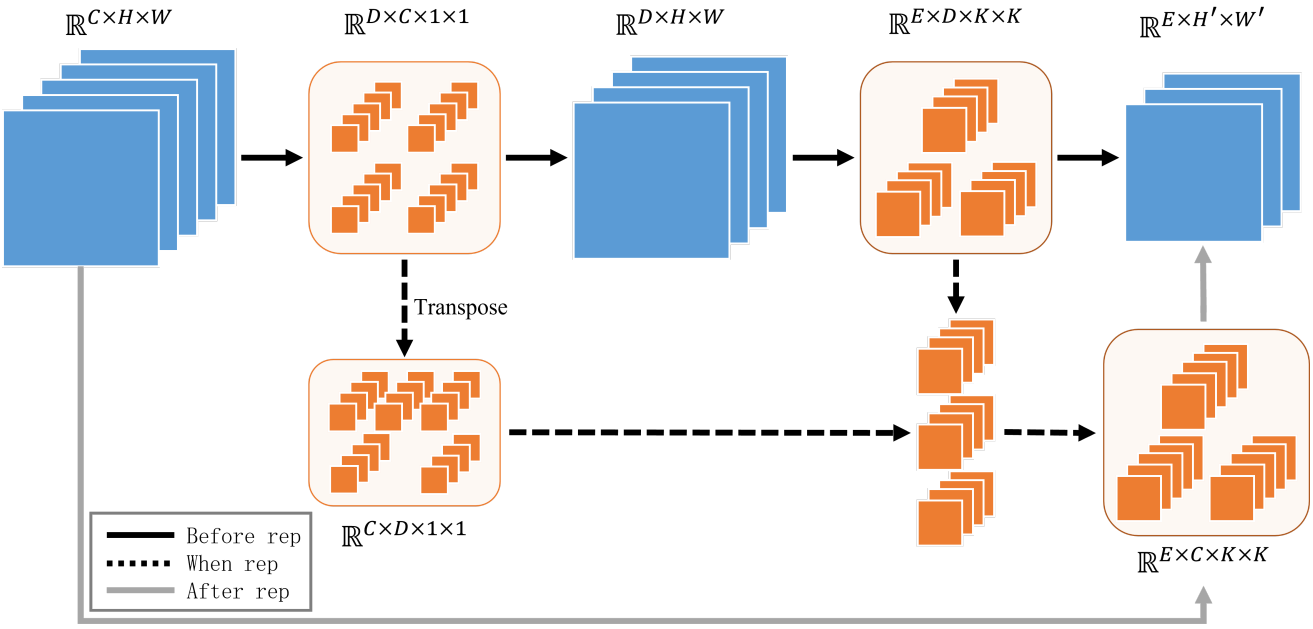
According to the associative axiom of the convolutional operation, the above equation can have the following transformations:

$$O = I * (F_1 * F_2) + B_1 * F_2 + B_2.$$

We can use $\hat{F} = F_1 * F_2$ and $\hat{B} = B_1 * F_2 + B_2$ directly as weights and biases for the individual conv layers after structural re-parameterization. However, merging two $K \times K$ ($K > 1$) kernels will yield a larger kernel, which seems to defeat our original intention of not increasing the inference cost, e.g., Google uses two 3×3 conv layers to replace the 7×7 conv layer, which has reduced the FLOPs and the inference cost. In summary, what is attractive is to merge a kernel of size K and a kernel of size 1 into a kernel of size K. This needs to be discussed in two cases: the pointwise conv layer comes before the standard conv layer, and the pointwise conv layer comes after the standard conv layer.

a. The pointwise conv layer is in front of the standard conv layer

The structure re-parameterization for the case where the pointwise conv layer is in front of the standard conv layer is shown below. For convenience, we make the width and height of the kernels equal, and all the bias is reflected in the code instead of in the figures.



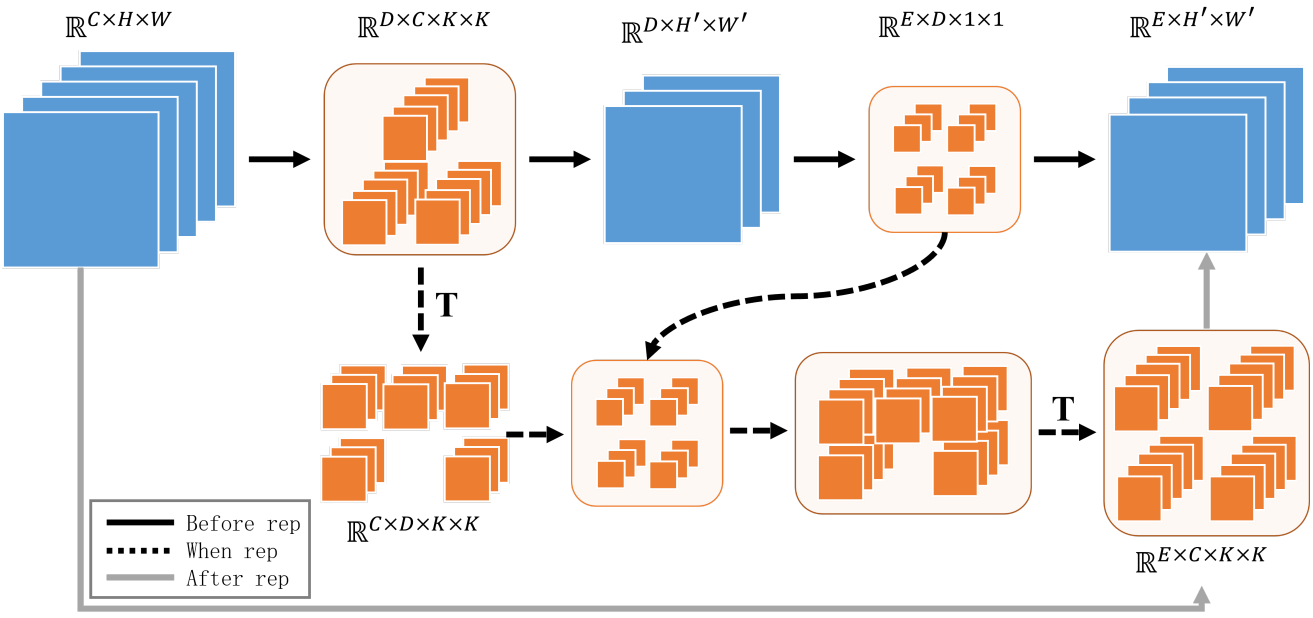
When kernels become the receptor of the convolutional operation, we prefer the kernel of size 1 to act as the conv layer and the kernel of size K to act as the input feature map, which reduces the padding operation. To get a single kernel of final size $E \times C \times K \times K$, it is straightforward to find that the pointwise conv needs to be transposed and perform a convolutional operation on the kernel of size K. The code for merging a pointwise conv layer and its subsequent standard conv layer is as follows.

```
def fuse_1x1_kxk_conv(conv_1x1_weight, conv_1x1_bias, conv_kxk_weight, conv_kxk_bias,
groups=1):
    r"""
    Args:
        conv_1x1_weight: weight of the pointwise conv layer
        conv_1x1_bias: bias of the pointwise conv layer
        conv_kxk_weight: weight of the standard conv layer
        conv_kxk_bias: bias of the standard conv layer
        groups: The number of groups if the kxk conv layer is a groupwise conv layer
    Return: fused kxk conv weight; fused kxk conv bias
    """
    if groups == 1:
        k = F.conv2d(conv_kxk_weight, conv_1x1_weight.permute(1, 0, 2, 3))
        b_hat = (conv_kxk_weight * conv_1x1_bias.reshape(1, -1, 1, 1)).sum((1, 2, 3))
    else:
        k_slices = []
        b_slices = []
        k1_group_width = conv_1x1_weight.size(0) // groups
        k2_group_width = conv_kxk_weight.size(0) // groups
        k1_T = conv_1x1_weight.permute(1, 0, 2, 3)
        for g in range(groups):
            k1_T_slice = k1_T[:, g * k1_group_width:(g + 1) * k1_group_width, :, :]
            k2_slice = conv_kxk_weight[g * k2_group_width:(g + 1) * k2_group_width, :, :,
:]
            k_slices.append(F.conv2d(k2_slice, k1_T_slice))
            b_slices.append((k2_slice * conv_1x1_bias[g * k1_group_width:(g + 1)
                             * k1_group_width].reshape(1, -1, 1, 1)).sum((1, 2, 3)))
        k, b_hat = re-parameterizer.depth_concat(k_slices, b_slices)
    if conv_kxk_bias is None:
        return k, b_hat
    return k, b_hat + conv_
```

Note that the padding operation must be attached to the first conv layer. Otherwise, there will be information loss after re-parameterization. In the code, we provide a re-parameterization method when using groupwise conv. However, we do not recommend using it because the single conv layer obtained after re-parameterization will be a standard conv layer instead of a groupwise conv layer, which may introduce unnecessary computational effort.

b. The standard conv layer is in front of the pointwise conv layer

The structure re-parameterization for the case where the standard conv layer is in front of the pointwise conv layer is shown below. For convenience, we make the width and height of the kernels equal, and all the bias is reflected in the code instead of in the figures.



Similar to the previous case, to get a single kernel of final size $E \times C \times K \times K$, it is straightforward to find that the pointwise conv needs to perform a convolutional operation on the transposed kernel of size K. After the convolutional operation, the obtained feature map also needs to be transposed to get the correct shape. The code for merging a standard conv layer and its subsequent pointwise conv layer is as follows.

```
def fuse_kxk_1x1_conv(conv_kxk_weight, conv_kxk_bias, conv_1x1_weight, conv_1x1_bias,
groups=1):
    r"""
    Args:
        conv_kxk_weight: weight of the standard conv layer
        conv_kxk_bias: bias of the standard conv layer
        conv_1x1_weight: weight of the pointwise conv layer
        conv_1x1_bias: bias of the pointwise conv layer
        groups: The number of groups if the kxk conv layer is a groupwise conv layer
    Return: fused kxk conv weight; fused kxk conv bias
    """
    if groups == 1:
        k = F.conv2d(conv_kxk_weight.permute(1, 0, 2, 3), conv_1x1_weight).permute(1, 0,
2, 3)
        b_hat = (conv_1x1_weight * conv_kxk_bias.reshape(1, -1, 1, 1)).sum((1, 2, 3))
        return k, b_hat + conv_1x1_bias
    k_slices = []
    b_slices = []
```

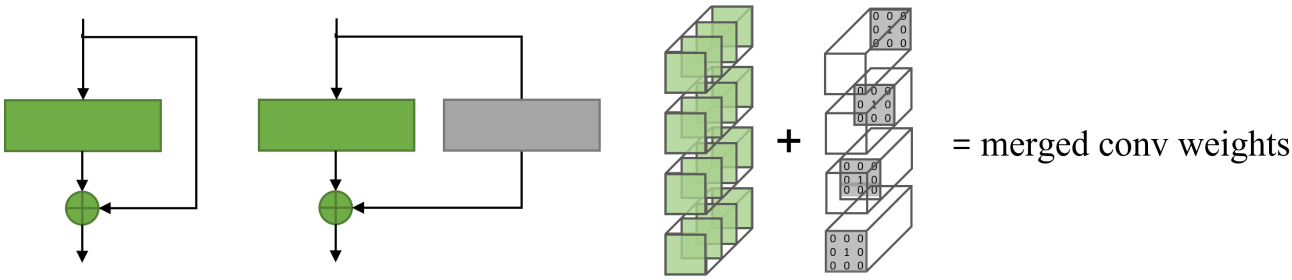
```
    k1_group_width = conv_1x1_weight.size(0) // groups
    k2_group_width = conv_kxk_weight.size(0) // groups
    k3_T = conv_kxk_weight.permute(1, 0, 2, 3)
    for g in range(groups):
        k1_slice = conv_1x1_weight[:, g * k2_group_width:(g + 1) * k2_group_width, :, :]
        k2_T_slice = k3_T[:, g * k2_group_width:(g + 1) * k2_group_width, :, :]
        k_slices.append(F.conv2d(k2_T_slice, k1_slice))
        b_slices.append((conv_1x1_weight[g * k1_group_width:(g + 1) * k1_group_width, :,
:, :]
                         * conv_kxk_bias.reshape(1, -1, 1, 1)).sum((1, 2, 3)))
    k, b_hat = re-parameterizer.depth_concat(k_slices, b_slices)
    if conv_1x1_bias is None:
        return k.permute(1, 0, 2, 3), b_hat
    return k.permute(1, 0, 2, 3), b_hat + conv_1x1_bias
```

6. Shortcut re-parameterization

The re-parameterization of shortcut branches covers various cases and should be discussed separately when the corresponding main branch contains different components.

a. One conv layer in the main branch

If there is only one conv layer on the main branch corresponding to the shortcut branch, this can be considered a structure with parallel conv layers for branch addition. It can be considered that there exists a dirac-initialized conv layer on the shortcut branch (the shape of this conv layer is not restricted), so this case can be easily re-parameterized as Paradigm 2.

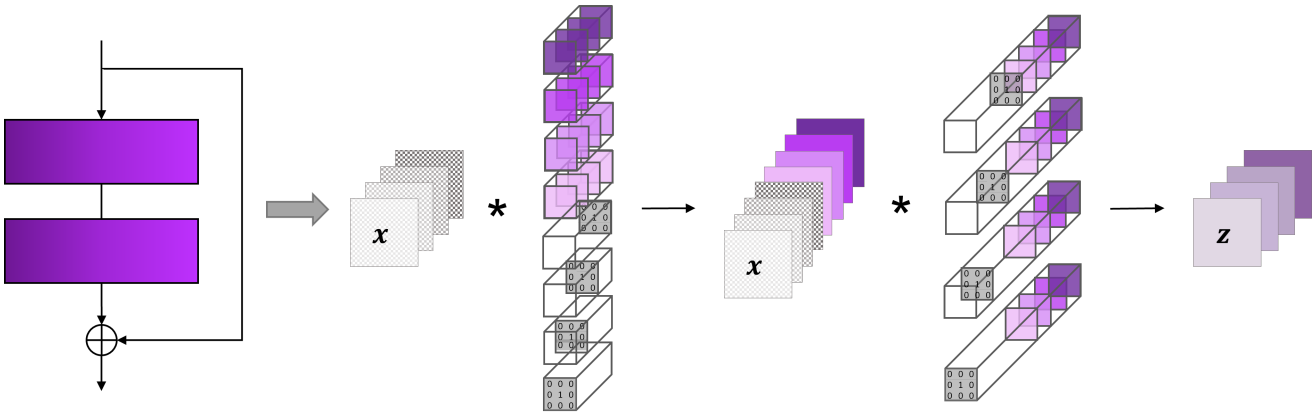


Define a conv kernel as $K \in \mathbb{R}^{O\times I\times H\times W}$, the dirac initialization for it can be described as follows.

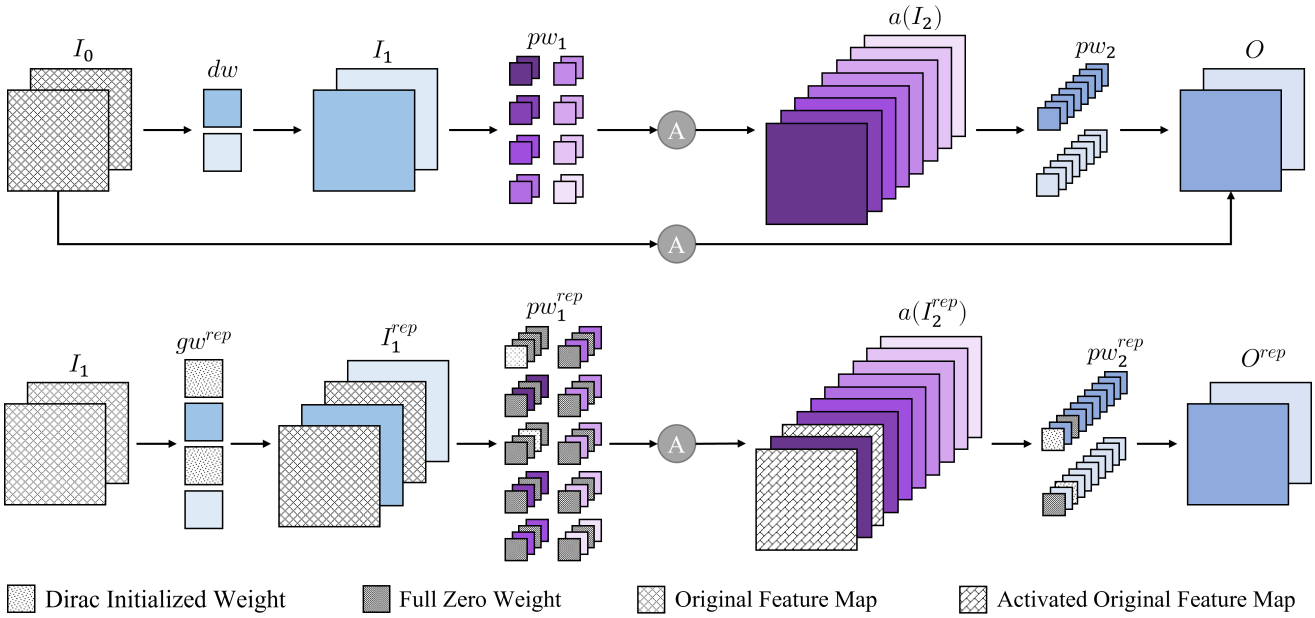
$$K_{o,i,h,w} = \begin{cases} 1 & \text{if } h = \frac{\mathrm{H}}{2}, w = \frac{\mathrm{W}}{2}, o = i, \\ 0 & \text{elsewise.} \end{cases}$$

b. Multiple conv layers in the main branch

If there are more conv layers on the main branch corresponding to the shortcut branch, we need to be more subtle in dirac initialization to ensure input-output consistency. In this case, the input or output dimension of the conv layers of the main branch needs to be increased. The added output dimension keeps the input feature map in the output feature map in the form of concatenation. The added input dimension ensures that the original feature map can be correctly added to the output feature map through the main branch. The weights corresponding to the added dimensions are dirac initialized.



However, when designing networks, we generally use components such as groupwise conv layers, activation layers, and inconsistent input and output dimensions. Therefore a complex example is necessary. The shortcut re-parameterization process for a RepNeXt block is shown below.



Non-all-zero weights and full-zero weights after dirac initialization are marked with different fill textures in the figure. The output dimension of all conv layers except the last one is increased, and the corresponding increased weights are dirac initialized. Thus, the original feature map can appear in the output feature map in a concatenated form. The input dimension of the last conv layer is increased, and the increased weights are also dirac initialized. Thus, the element-wise addition of the original feature map and the output feature map of the main branch can be equivalently converted into a convolutional operation. Some concrete instructions are as follows: **1)** Dirac-initialized kernels need to be inserted into the depthwise conv layer one after another to form a groupwise conv layer (group=dim). This is because the group conv always uses adjacent kernels as a group. **2)** To accommodate instruction 1, dirac-initialized kernels also need to be inserted into the first pointwise conv layer one after another. Some additional all-zero-initialized weights need to be introduced in the input dimension, as pw_1^{rep} demonstrates. **3)** To adapt shortcut re-parameterization to various activation layers, we add an activation layer on the shortcut branch that is identical to the main branch. **4)** The consistency of O and O^{rep} can be proved as follows.

Define the kernels associated with structural re-parameterization as $pw_2^{\text{rep}} = [pw_2^{\text{C}\times 4\text{C}}, E^{\text{C}\times\text{C}}]^{\text{C}\times 5\text{C}}$; $B(pw_2^{\text{rep}}) = B^{\text{C}}(pw_2)$; $pw_1^{\text{rep}} = [[pw_1^{4\text{C}\times\text{C}}, 0^{\text{C}\times\text{C}}]^{5\text{C}\times\text{C}}, [0^{4\text{C}\times\text{C}}, E^{\text{C}\times\text{C}}]^{5\text{C}\times\text{C}}]^{5\text{C}\times 2\text{C}}$; $B(pw_1^{\text{rep}}) = [B^{4\text{C}}(pw_1), 0^{\text{C}}]^{5\text{C}}$; $gw^{\text{rep}} = [dw^{\text{C}\times 1}, E^{\text{C}\times 1}]^{2\text{C}\times 1}$; $B(gw^{\text{rep}}) = [B^{\text{C}}(dw), 0^{\text{C}}]^{2\text{C}}$, where $[\cdot,\cdot]$ means the depth concatenation of the two elements, E means the dirac-initialized weights. For ease of understanding, the dimensions of noticeable elements have been marked in their upper right. I_1^{rep} can be derived as:

$$
\begin{aligned}
I_1^{\text{rep}} &= gw^{\text{rep}} * I_0^{\text{C}} + B(gw^{\text{rep}}) \\
&= [dw^{\text{C}\times 1}, E^{\text{C}\times 1}]^{2\text{C}\times 1} * I_0^{\text{C}} + [B^{\text{C}}(dw), 0^{\text{C}}]^{2\text{C}} \\
&= [dw^{\text{C}\times 1} * I_0^{\text{C}} + B^{C}(dw), E^{\text{C}\times 1} * I_0^{\text{C}} + 0^{\text{C}}] \\
&= [I_1^{\text{C}}, I_0^{\text{C}}]^{2\text{C}}.
\end{aligned}
$$

Using the results of I_1^{rep}, $\text{a}(I_2^{\text{rep}})$ can be calculated as:

$$
\begin{aligned}
\text{a}(I_2^{\text{rep}}) &= \text{a}(pw_1^{\text{rep}} * I_1^{\text{rep}} + B(pw_1^{\text{rep}})) \\
&= \text{a}([[pw_1^{4\text{C}\times\text{C}}, 0^{\text{C}\times\text{C}}]^{5\text{C}\times\text{C}}, [0^{4\text{C}\times\text{C}}, E^{\text{C}\times\text{C}}]^{5\text{C}\times\text{C}}]^{5\text{C}\times 2\text{C}} * [I_1^{\text{C}}, I_0^{\text{C}}]^{2\text{C}} + [B^{4\text{C}}(pw_1), 0^{\text{C}}]^{5\text{C}}) \\
&= \text{a}([pw_1^{4\text{C}\times\text{C}}, 0^{\text{C}\times\text{C}}]^{5\text{C}\times\text{C}} * I_1^{\text{C}} + [0^{4\text{C}\times\text{C}}, E^{\text{C}\times\text{C}}]^{5\text{C}\times\text{C}} * I_0^{\text{C}} + [B^{4\text{C}}(pw_1), 0^{\text{C}}]^{5\text{C}}) \\
&= \text{a}([pw_1^{4\text{C}\times\text{C}} * I_1^{\text{C}}, 0^{\text{C}}]^{5\text{C}} + [0^{4\text{C}}, I_0^{\text{C}}]^{5\text{C}} + [B^{4\text{C}}(pw_1), 0^{\text{C}}]^{5\text{C}}) \\
&= \text{a}([pw_1^{4\text{C}\times\text{C}} * I_1^{\text{C}} + 0^{4\text{C}} + B^{4\text{C}}(pw_1), 0^{\text{C}} + I_0^{\text{C}} + 0^{\text{C}}]) \\
&= [\text{a}^{4\text{C}}(I_2), \text{a}^{\text{C}}(I_0)]^{5\text{C}},
\end{aligned}
$$

where a can be any kind of activation. Finally, we use the result of $\text{a}(I_2^{\text{rep}})$ to calculate O^{rep}, and you will find that it is equivalent to the original output O.

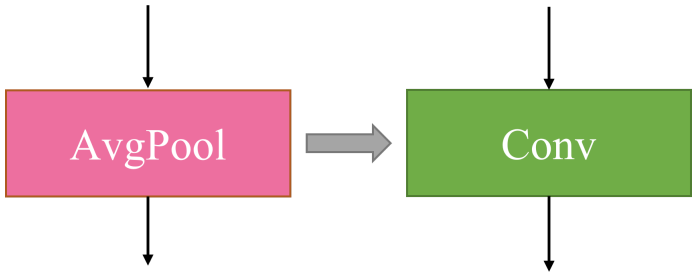
$$
\begin{aligned}
O^{\text{rep}} &= pw_2^{\text{rep}} * \text{a}(I_2^{\text{rep}}) + B(pw_2^{\text{rep}}) \\
&= [pw_2^{\text{C}\times 4\text{C}}, E^{\text{C}\times\text{C}}]^{\text{C}\times 5\text{C}} * [\text{a}^{4\text{C}}(I_2), \text{a}^{\text{C}}(I_0)]^{5\text{C}} + B^{\text{C}}(pw_2) \\
&= pw_2^{C\times 4C} * a^{4C}(I_2) + E^{C\times C} * a^{C}(I_0) + B^{C}(pw_2) \\
&= pw_2^{\text{C}\times 4\text{C}} * \text{a}^{4\text{C}}(I_2) + B^{\text{C}}(pw_2) + \text{a}^{\text{C}}(I_0) \\
&= O.
\end{aligned}
$$

The code for the re-parameterization for shortcut branches in a RepNeXt block is as follows.

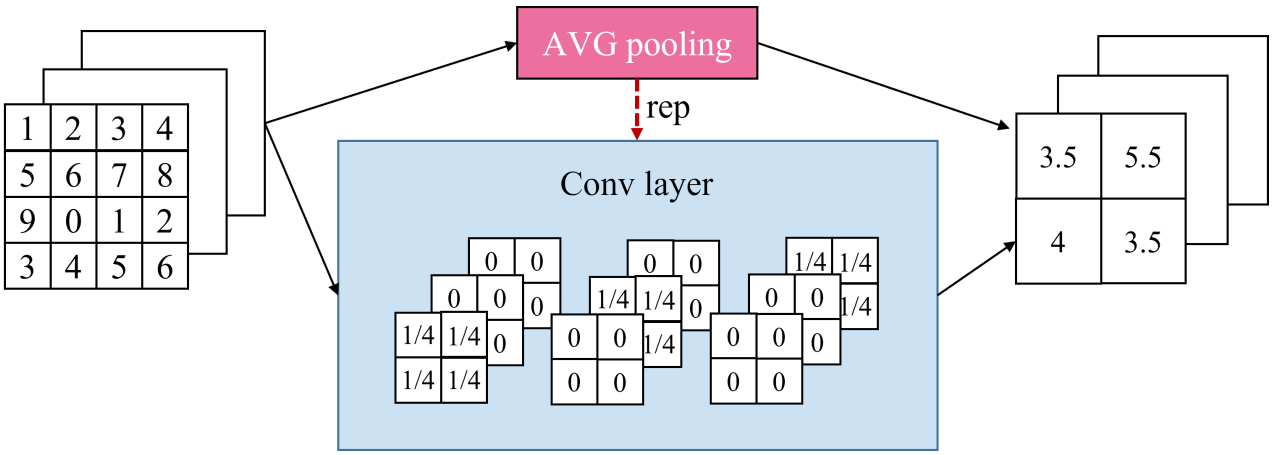
```
def switch_to_deploy(self):
    id_feature = nn.Conv2d(self.dim, self.dim * 2, kernel_size=7, padding=3,
groups=self.dim, bias=True).eval()
    for i in range(id_feature.in_channels):
        nn.init.dirac_(id_feature.weight.data[2 * i: 2 * i + 1])
    id_feature.bias.data[::2] = 0
    id_feature.weight.data[1::2] = self.feature.conv_repara.weight.data
    id_feature.bias.data[1::2] = self.feature.conv_repara.bias.data

    id_pwconv1 = nn.Conv2d(self.dim * 2, self.dim * 5, kernel_size=1, bias=True).eval()
    nn.init.dirac_(id_pwconv1.weight.data[:self.dim * 2:2, ::2])
    nn.init.zeros_(id_pwconv1.weight.data[1:self.dim * 2:2, ::2])
    nn.init.zeros_(id_pwconv1.weight.data[:self.dim * 2:2, 1::2])
    id_pwconv1.bias.data[:self.dim * 2:2] = 0
    id_pwconv1.bias.data[1:self.dim * 2:2] = self.pwconv1.bias.data[:self.dim]
    id_pwconv1.weight.data[1:self.dim * 2:2, 1::2] = self.pwconv1.weight.data[:self.dim]
    nn.init.zeros_(id_pwconv1.weight.data[self.dim * 2:, ::2])
    id_pwconv1.weight.data[self.dim * 2:, 1::2] = self.pwconv1.weight[self.dim:]
    id_pwconv1.bias.data[self.dim * 2:] = self.pwconv1.bias[self.dim:]

    id_pwconv2 = nn.Conv2d(self.dim * 5, self.dim, kernel_size=1, bias=True).eval()
    nn.init.dirac_(id_pwconv2.weight.data[:, :self.dim * 2:2])
    id_pwconv2.weight.data[:, 1:self.dim * 2:2] = self.pwconv2.weight[:, :self.dim]
    id_pwconv2.weight.data[:, self.dim * 2:] = self.pwconv2.weight[:, self.dim:]
    id_pwconv2.bias.data = self.pwconv2.bias

    self.deploy = True
    self.block = nn.Sequential(id_feature, id_pwconv1, self.act, id_pwconv2)
```

```
        self.__delattr__("feature")
        self.__delattr__("pwconv1")
        self.__delattr__("pwconv2")
        self.__delattr__("act")
        self.__delattr__("drop_path")
```

7. Switch a pooling layer into a conv layer



An average pooling with kernel size K and stride S applied to C channels can be equivalently converted into a conv layer with the same K and S, whose re-parameterization process is shown in the figure below.



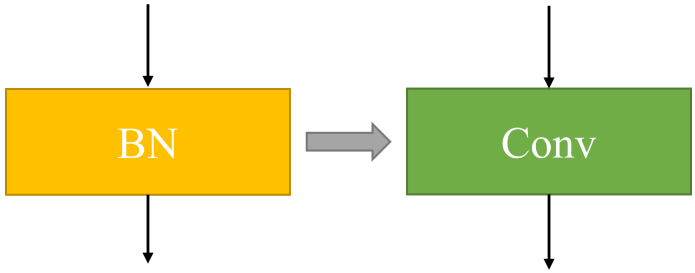
The input and output dimensions of the conv layer obtained by re-parameterization are the same, whose parameters are initialized as follows.

$$F_{d,c,:,:} = \begin{cases} \frac{1}{K \times K} & \text{if } d = c, \\ 0 & \text{elsewise.} \end{cases}$$

Since channels where d is not equal to c are all-zero initialized, you can generalize this to depthwise conv or groupwise conv, and the code is as follows.

```
def trans_avg_2_conv(channels, kernel_size, groups=1):
    r"""
    Args:
        channels: number of channels received
        kernel_size: the kernel size of the AVG pooling layer
        groups: The number of groups you want
    Return: the weight of the transformed kxk conv layer
    """
    input_dim = channels // groups
    k = torch.zeros((channels, input_dim, kernel_size, kernel_size))
    k[np.arange(channels), np.tile(np.arange(input_dim), groups), :, :] = 1.0 /
kernel_size ** 2
    return k
```

8. switch a BN layer into a conv layer



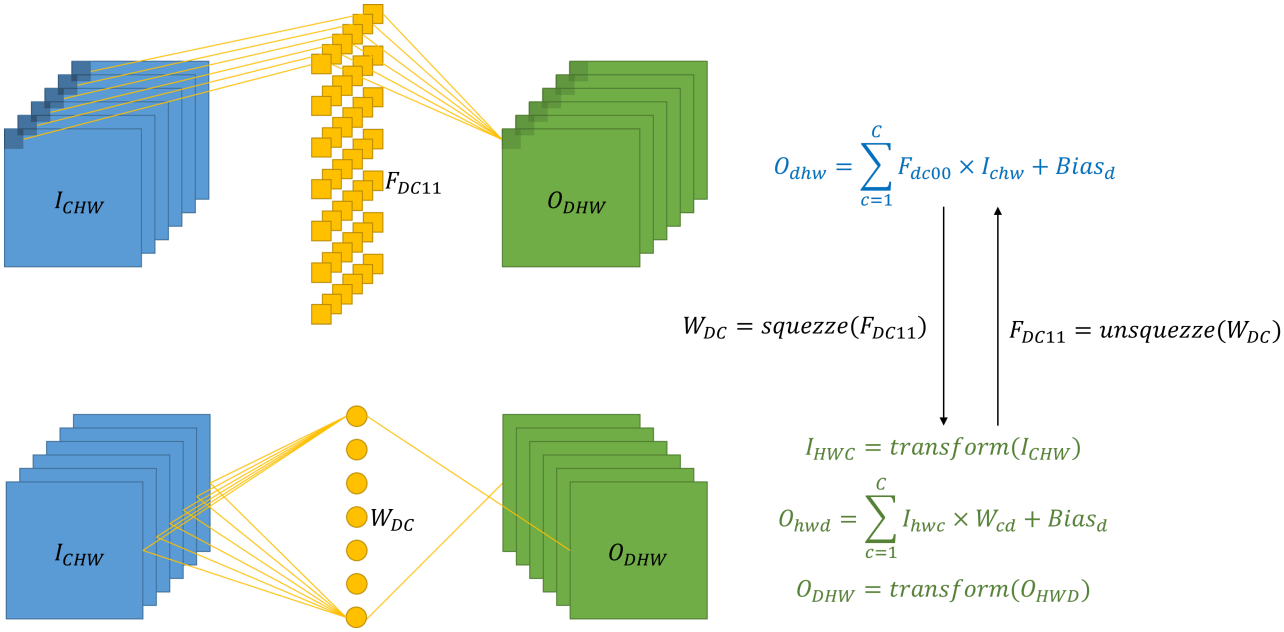
When converting a BN layer to a conv layer, just imagine that there exists a conv layer whose weights are dirac-initialized in front of that BN layer so that the process of switching a BN layer to a conv layer is converted into the process of merging of a conv and a BN layer, see Paradigm 1. Interestingly, the hypothetical conv layer is not limited by the kernel size, and it can be a depthwise conv layer or a groupwise conv layer, and the code is as follows.

```
@staticmethod
def trans_bn_2_conv(bn, channels, kernel_size, groups=1):
    """
    Args:
        bn: the bn layer for transformation
        channels: the number of receiving channels in the bn layer
        kernel_size: the kernel size you want
        groups: the number of groups you want
    Return: fused conv weight; fused conv bias
    """
    assert kernel_size % 2 == 1
    input_dim = channels // groups
    kernel_value = np.zeros((channels, input_dim, kernel_size, kernel_size),
dtype=np.float32)
    for i in range(channels):
        kernel_value[i, i % input_dim, kernel_size//2, kernel_size//2] = 1
    kernel = torch.from_numpy(kernel_value).to(bn.weight.device)
    running_mean = bn.running_mean
    running_var = bn.running_var
    gamma = bn.weight
    beta = bn.bias
    eps = bn.eps
    std = (running_var + eps).sqrt()
    t = (gamma / std).reshape(-1, 1, 1, 1)
    return kernel * t, beta - running_mean * gamma / std
```

9. Interconversion of a pointwise conv and a fully connected layer

The fully connected layer and the pointwise conv layer are interconvertible. However, it is generally seldom considered to re-parameterize the two with each other because the principles of the two are the same. Nevertheless, because fully connected and conv layers differ in their implementations, they can have some differences in performance and efficiency. Experience (cf., Einops) suggests that the fully connected layer may perform better than the pointwise conv layer, coupled with the fact that the efficiency of the conv layer will be higher on some devices that are more adequately optimized for convolutional operations, the implications of switching from a fully connected layer to a pointwise conv layer seem striking. The following

figure illustrates the principle of the interconversion of a pointwise conv and a fully connected layer (The dimensions of the important components are displayed in the lower right of the formula).



As shown in the figure, the interconversion between a pointwise conv and a fully connected layer is straightforward, requiring only a shape transformation, and the code is shown below.

```
def lc2pwconv(lc_weight, lc_bias):
    """
    Args:
        lc_weight: the weight of the fully connected layer
        lc_bias: the bias of the fully connected layer
    Return: new conv weight (OxIx1x1), new conv bias (I)
    """
    out_shape, in_shape = lc_weight.shape
    return lc_weight.reshape((out_shape, in_shape, 1, 1)), lc_bias

def lc2pwconv(conv_weight, conv_bias):
    """
    Args:
        conv_weight: the weight of the pointwise conv layer
        conv_bias: the bias of the pointwise conv layer
    Return: new weight of the fully connected layer (OxI), new bias of the fully
connected layer (I)
    """
    out_shape, in_shape, _, _ = conv_weight.shape
    return conv_weight.reshape((out_shape, in_shape)), conv_bias
```