Network Specification

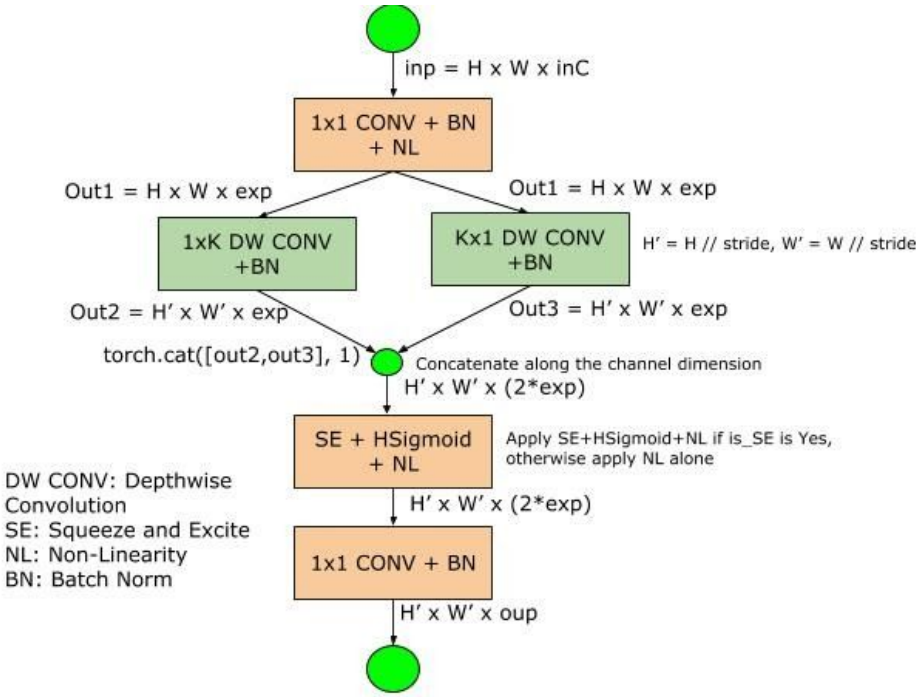
Input (HxWxinC)	Operator	Kernel size (K)	BN	Exp-size (exp)	Out-size (oup)	Squeeze-and-Excite (is_SE)	Non-linearity (NL)	stride
224x224x3	Conv2d	3x3	Yes	-	16	No	H-Swish	2
112x112x16	Fuse	3x1,1x3	Yes	16	16	Yes	ReLU	2
56x56x16	Fuse	3x1,1x3	Yes	72	24	No	ReLU	2
28x28x24	Fuse	3x1,1x3	Yes	88	24	No	ReLU	1
28x28x24	Fuse	5x1,1x5	Yes	96	40	Yes	H-Swish	2
14x14x40	Fuse	5x1,1x5	Yes	240	40	Yes	H-Swish	1
14x14x40	Fuse	5x1,1x5	Yes	240	40	Yes	H-Swish	1
14x14x40	Fuse	5x1,1x5	Yes	120	48	Yes	H-Swish	1
14x14x48	Fuse	5x1,1x5	Yes	144	48	Yes	H-Swish	1
14x14x48	Fuse	5x1,1x5	Yes	288	96	Yes	H-Swish	2
7x7x96	Fuse	5x1,1x5	Yes	576	96	Yes	H-Swish	1
7x7x96	Fuse	5x1,1x5	Yes	576	96	Yes	H-Swish	1
7x7x96	Conv2d	1x1	Yes	-	576	Yes	H-Swish	1
7x7x576	AdaptiveAveratePooling	7x7	No	-	-	-	-	1
1x1x576	Conv2d	1x1	No	-	1024	-	H-Swish	1
1x1x1024	Conv2d	1x1	No	-	100	-	-	1

Note:

1. H, W - Spatial dimensions, inC - number of input channels
2. BN: Batch Normalization - follows a convolution layer if specified (all convolutions in Fuse)
3. Exp-size (exp): The number of intermediate output channels (expansion) for a Fuse block
4. Out-size (oup): The number of output channels (equivalently, number of filters)

5. Squeeze-and-Excite (SE): An additional block, called Squeeze and Excite (Paper) that is conditionally a part of the Fuse block (if is_SE is Yes)
6. Use bias=False for all convolution layers while building the network
7. Pad all layers accordingly such that the input and output spatial dimensions are maintained. (In general, padR = (R - 1) // 2, padS = (S-1) // 2, padding = (padR, padS) for an RxS filter)
8. Add Dropout before the final conv2d layer (linear classifier) with a probability p=0.2
9. You can equivalently use nn.Linear for the final conv2d which is the classifier layer
10. The data-layout is NCHW (N = Batch, C = Channels, H,W = Spatial Dims)
11. Apply stride in Fuse block only for the 1xK and Kx1 DW conv. For the pointwise blocks, stride = 1
12. The above input dimensions are an illustration - for CIFAR-100 the input dimensions will start from 32x32x3. Since PyTorch automatically takes care of dimensions, you need to care only about reducing the dimensions in your code. (For instance, the x-value for AdaptiveAveragePooling(x))

Fuse Block:



Transforms required for CIFAR-100

You are required to add these transforms to the CIFAR-100 dataset before loading them using a DataLoader (check here and here). You are not allowed to do any other augmentations than specified.

```
transform_train = transforms.Compose([
                    transforms.RandomCrop(32, padding=4),
                    transforms.RandomHorizontalFlip(),
                    transforms.ToTensor(),
                    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
                    ])
  transform_test = transforms.Compose([
                    transforms.ToTensor(),
                    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
  ])
```

Weight Initialization:

```
import torch
import torch.nn as nn

 def _initialize_weights(self):
      # weight initialization
      for m in self.modules():
         if isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(m.weight, mode='fan_out')
            if m.bias is not None:
               nn.init.zeros_(m.bias)
         elif isinstance(m, nn.BatchNorm2d):
            nn.init.ones_(m.weight)
            nn.init.zeros_(m.bias)
         elif isinstance(m, nn.Linear):
            nn.init.normal_(m.weight, 0, 0.01)
            if m.bias is not None:
               nn.init.zeros_(m.bias)
```

Squeeze and Excitation:

This layer tries to model the interdependencies between channels of the convolutional features. Squeezing and exciting across channels lets the network learn to selectively emphasize informative features and suppress less important ones.

To make things simple, you can use this code-snippet for implementing Squeeze-and-Excite, H-Sigmoid, and H-Swish Modules

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Hsigmoid(nn.Module):
    def __init__(self, inplace=True):
        super(Hsigmoid, self).__init__()
        self.inplace = inplace

    def forward(self, x):
        return F.relu6(x + 3., inplace=self.inplace) / 6.
```

```
class SEModule(nn.Module):
    def __init__(self, channel, reduction=4):
        super(SEModule, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(
            nn.Linear(channel, channel // reduction, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(channel // reduction, channel, bias=False),
            Hsigmoid()
        )

    def forward(self, x):
        b, c, _, _ = x.size()
        y = self.avg_pool(x).view(b, c)
        y = self.fc(y).view(b, c, 1, 1)
        return x * y.expand_as(x)
```

And this code snippet for HSwish,

```
class Hswish(nn.Module):
    def __init__(self, inplace=True):
        super(Hswish, self).__init__()
        self.inplace = inplace

    def forward(self, x):
        return x * F.relu6(x + 3., inplace=self.inplace) / 6.
```