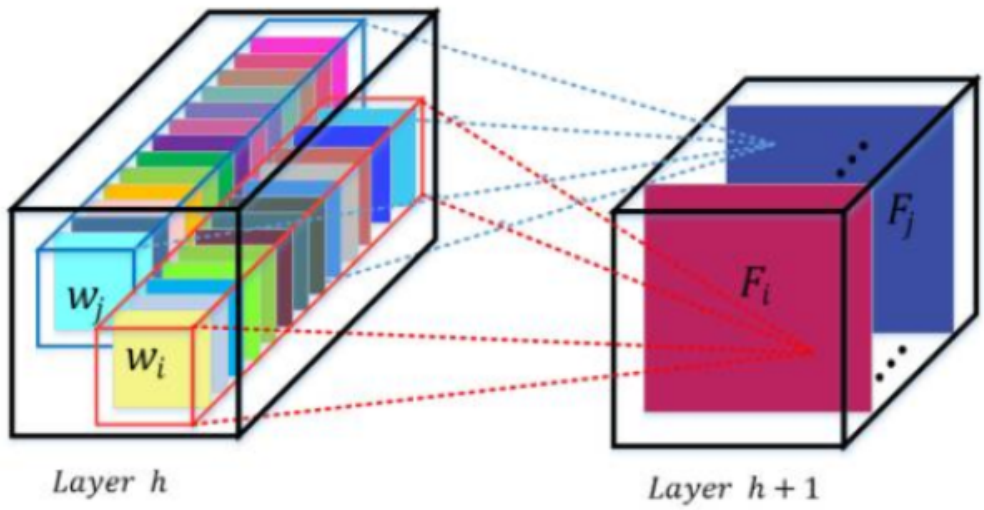
Squeeze-and-Excitation Networks

Introduction

In the field of deep learning, the development of CNN has play a great role in many computer vision tasks, such as classification and segmentation because many of them take CNN as backbone. VGG, ResNet, Inception, DenseNet and many other models have sprung up and achieved great preformance. Here we want to introduce an effectively and simple model named SENet, Sequeeze-and-Excitation Networks, which won ImageNet-2017 champion. What's more important, SENet can be plugged into other CNN models, like Inception, ResNet. In this lecture, we will teach you why we need squeeze and excitation and how it works.

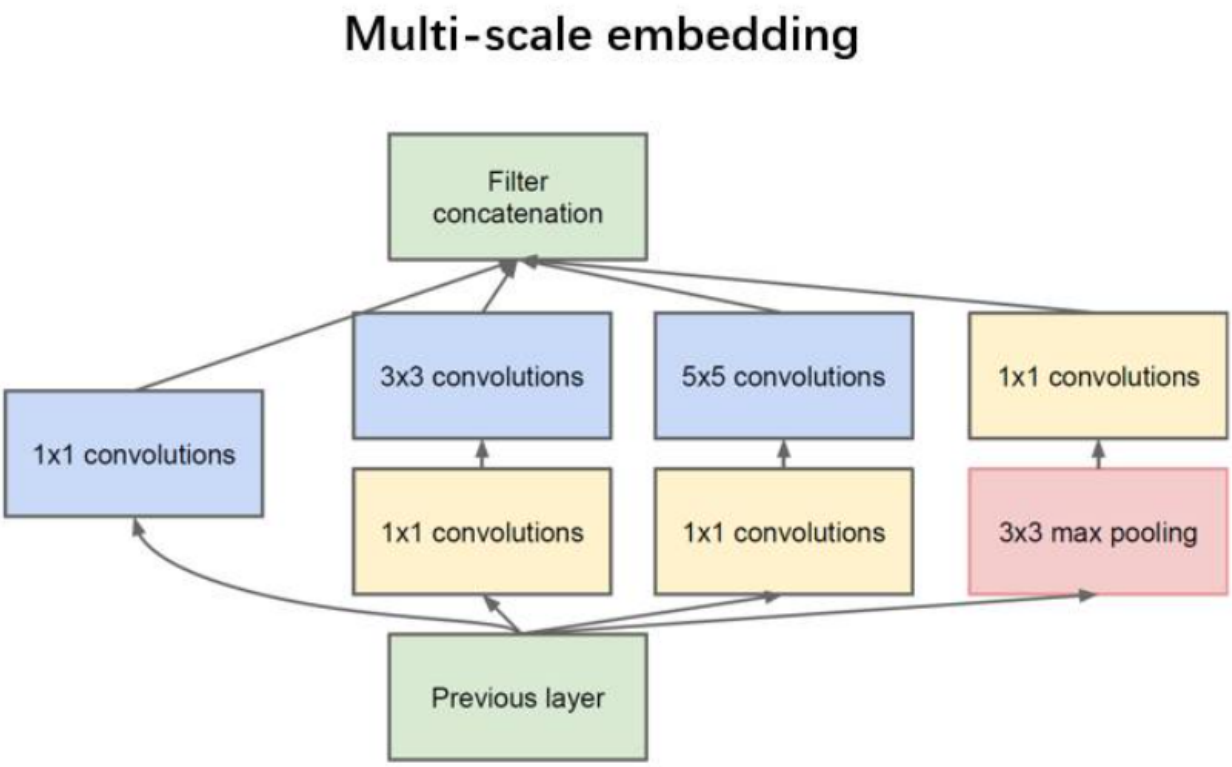
I believe everyone is familiar with convolution (if not, please refer to [CNN LINK]). ***A convolution filter is expected to be an informative combination, which fuse channel-wise and spatial information within local receptive fields.***



Convolution neural networks are composed of convolution filters, non-linear layers and down-sampling layers so that it can capture the image features from the global

receptive field. Global information is of great importance for many tasks, such as classfication.

So, how to enhance the capture of spatial information becomes a hot topic. Like inception, its structure is embedded with multi-scale information and fusing features in different receptive fields to improve the performance. Or introduce attention or consider the temporal information (previous frames and last frames)



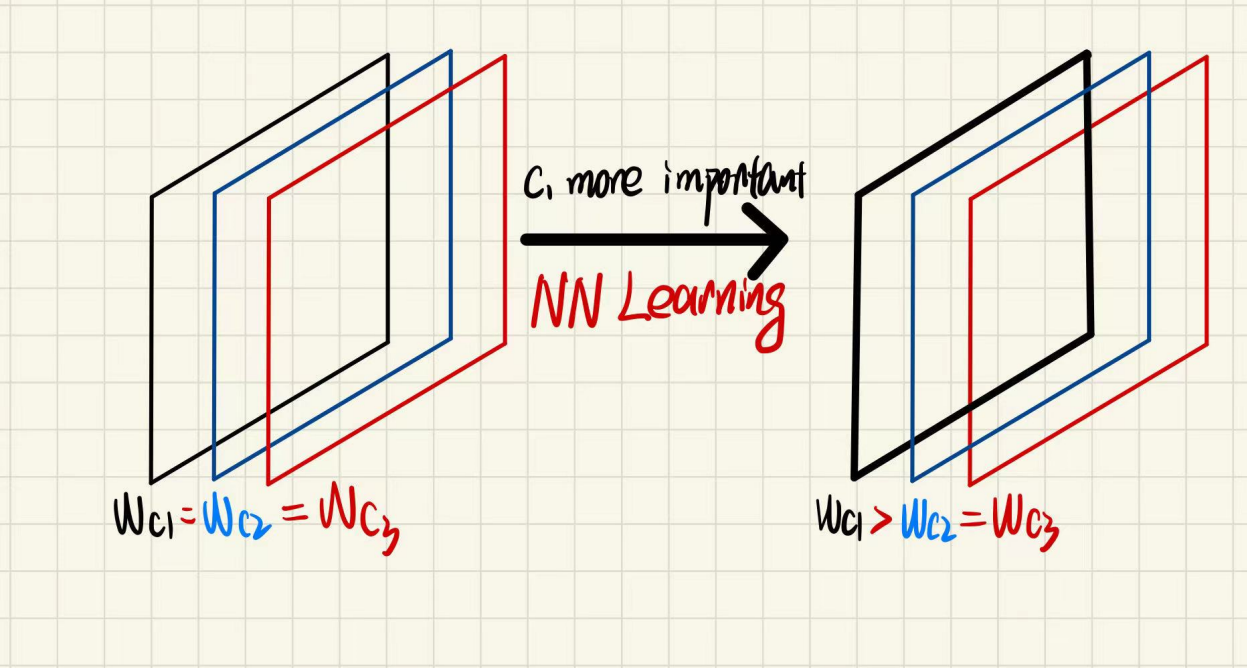
Squeeze-and-Excitation Networks

As we can see from some previous excellent work, they all consider how to improve the performance in the spatial field. But we can ask us one question, is there any fields that we can consider? Whether a network can be enhanced from the aspect of channel relationship?

If you can think of this, congratulations, you are as good as the authors of SENet!

But where does the idea come from? What is the motivation? In previous convolution filters, each channel has equal weights, which means make equal contribution to the final output. However, even from our intuition, different channels ought to focus on different things.

For example, the first channel is more important than the other channels. So, we need to increase the weight of first channel and meanwhile decrease the others. How can we do this? Neural network is designed for this! Use learning method to get the weights or importance of respective channels, then we can promote useful features and suppress the useless ones.



Make a conclusion for the motivation:

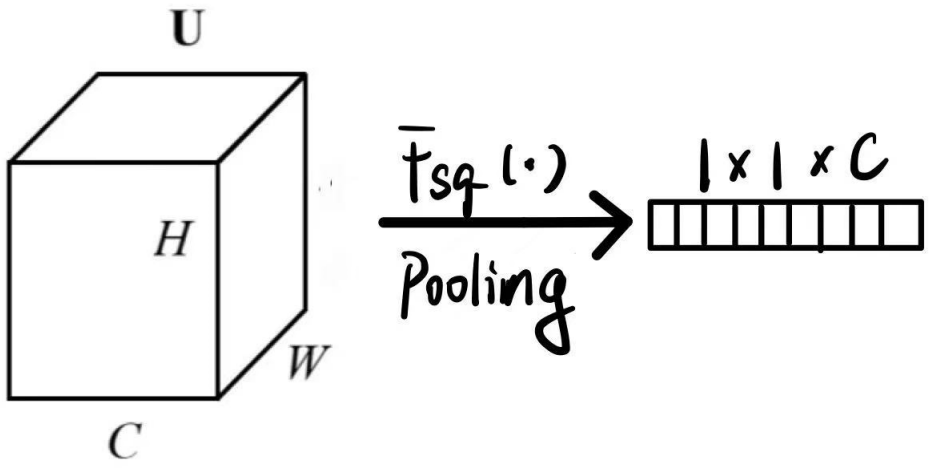
- Explicitly model channel-interdependencies within modules
- Feature recalibration

  ▼ Selectively enhance useful features and suppress less useful ones

Squeeze

Right now we have got the central idea of SENet: learn the weights of channels! But how can we do this? Let's think of pooling layers. It is designed for increase receptive

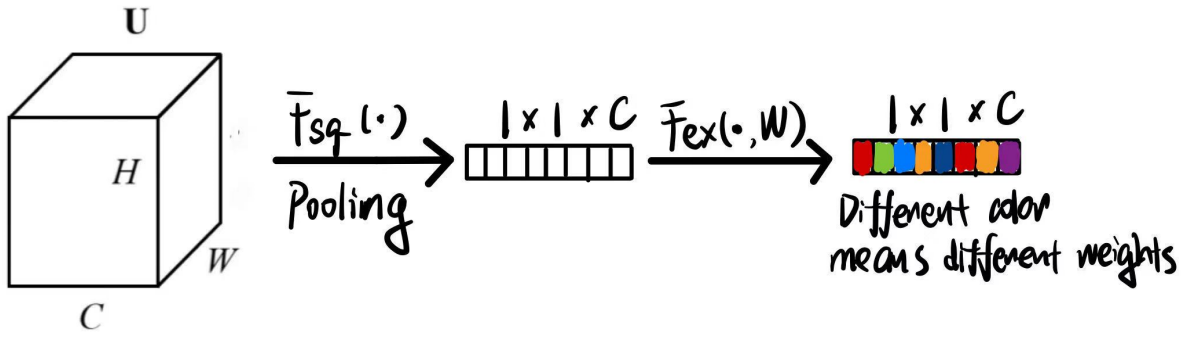
field and help network get more global information. How about we use pooling layers on the whole feature maps? So that we get 1 x 1 x C (C means the channels) matrix and each channel has all the information in some degree. That's "Squeeze" doing. ***Shrink feature maps*** $\in R^{w \times h \times c}$ ***through spatial dimensions*** $(w \times h)$ ***and get global distribution of channel-wise responses.***



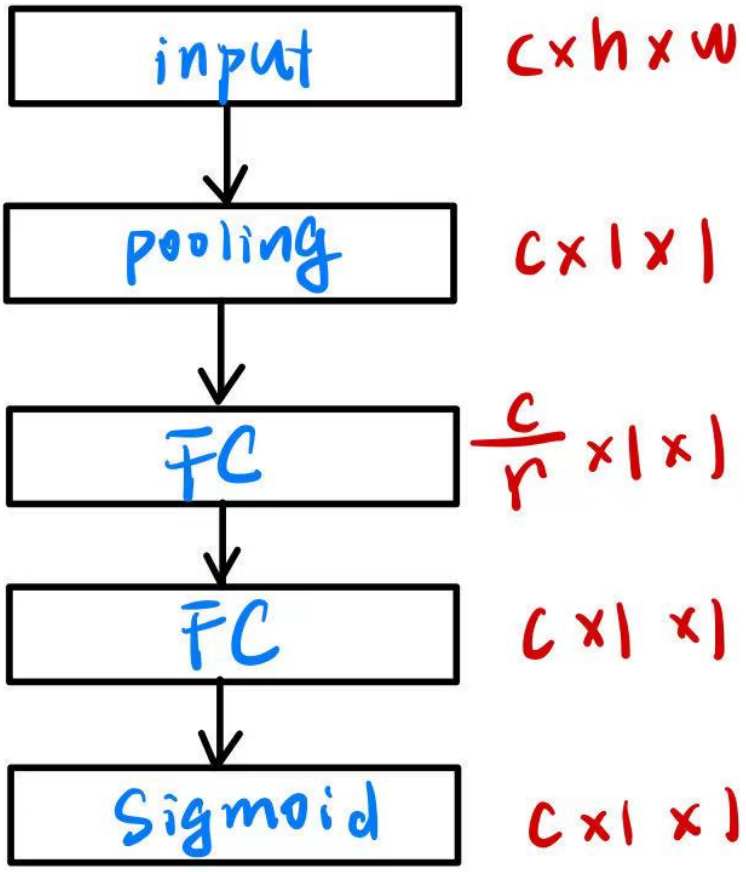

```
class SELayer(nn.Module):
    def __init__(self, channel, reduction=16):
        super(SELayer, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(output_size=1)

    def forward(self, x):
        b, c, _, _ = x.size()
        y = self.avg_pool(x).view(b, c)
        return y
# TEST
senet = SELayer()
input = torch.randn((10, 20, 5, 5))
output = senet.forward(input)
assert(output.size == (10, 20, 1, 1))
```

Excitation

After squeeze, we get the channel-wise responses and then we need to learn the weights. Here comes excitation. ***It can learn*** $W \in R^{c \times c}$ ***to explicitly model channel-association and what's more, excitation kinds of like the gating mechanism in RNN, to produce channel-wise weights.***



We need to use two fully connected layers to learn the channel-wise association. Here we introduce one hyperparameters named **reduction.** Based on this, we decrease the channel which can greatly decrease the number of parameters and improve computation performance. Use two FC layers instead of one can help model get more non-linearity and simulate the complex association between channels. Finally, going through one sigmoid gate to get 0 - 1 weights.




```
class SELayer(nn.Module):
    def __init__(self, channel, reduction=16):
        super(SELayer, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(output_size=1)
        self.fc = nn.Sequential(
            nn.Linear(channel ,channel // reduction, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(channel // reduction, channel, bias=False),
            nn.Sigmoid()
        )

    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 y

# TEST
senet = SELayer()
input = torch.randn((10, 20, 5, 5))
output = senet.forward(input)
assert(output.size == (10, 20, 1, 1))
```

Scale

Finally, we can think of the output of excitation as the importance of each channel after the feature choosing procedure. Then reweight the feature maps by channel-wise multiplication to the previous feature, which can be directly fed into subsequent layers. It can be called ***feature recalibration.***

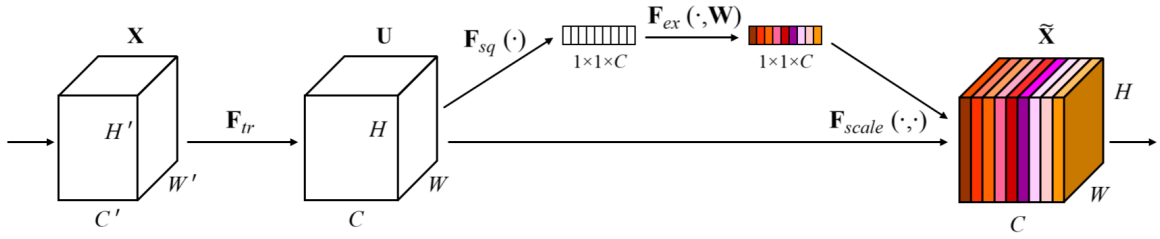
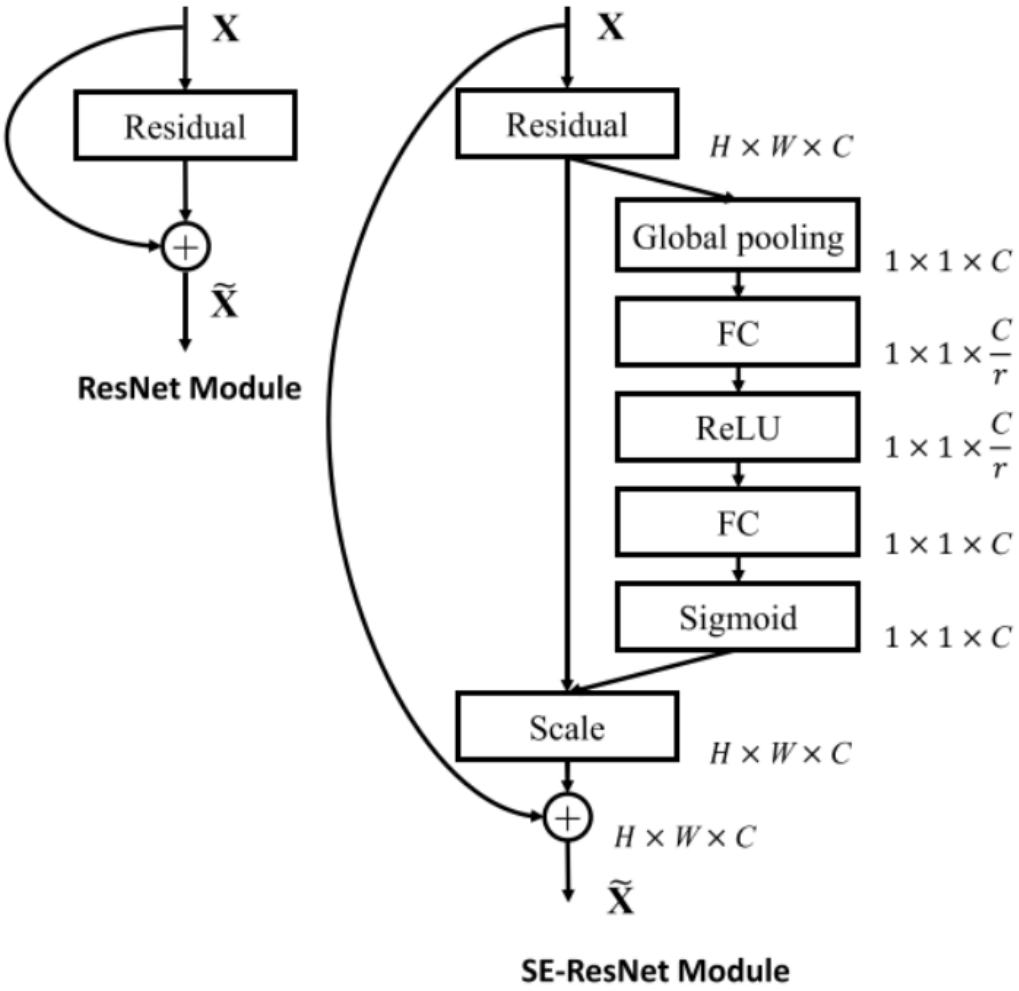


Figure 1: A Squeeze-and-Excitation block.

```
class SELayer(nn.Module):
    def __init__(self, channel, reduction=16):
        super(SELayer, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(output_size=1)
        self.fc = nn.Sequential(
            nn.Linear(channel ,channel // reduction, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(channel // reduction, channel, bias=False),
            nn.Sigmoid()
        )

    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)

# TEST
senet = SELayer()
input = torch.randn((10, 20, 5, 5))
output = senet.forward(input)
assert(output.size == (10, 20, 5, 5))
```

SE-ResNet

SENet can easily embed into skip connections module, like resnet and inception. Then we get a novel model. named SE-ResNet, SE-Inception, etc.




```
import torch.nn as nn
from se_layers import SELayer
from resnet import ResNet


def conv3x3(in_planes, out_planes, stride=1):
    return nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride, padding=1, bias=False)
```

```
class SEBasicBlock(nn.Module):
    """
    Basic block for resnet18 or resnet34
    expansion = 1
    structure: two 3x3 convolutions
    """

    expansion = 1

    def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
                 base_width=64, dilation=1, norm_layer=None, *, reduction=16):
        super(SEBasicBlock, self).__init__()

        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes, 1)
        self.bn2 = nn.BatchNorm2d(planes)
        self.se = SELayer(planes, reduction)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

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

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

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

        return out



class SEBottleneck(nn.Module):
    """
    Basic block for resnet50 and resnet with more layers
    expansion = 4
    structure:  1x1 convolution + 3x3 convolution + 1x1 convolution
    """

    expansion = 4

    def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
                 base_width=64, dilation=1, norm_layer=None,
                 *, reduction=16):
```

```
        super(SEBottleneck, self).__init__()
        self.conv1 = nn.Conv2d(inplanes, planes, kernel_size=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=stride,
                               padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)
        self.conv3 = nn.Conv2d(planes, planes * 4, kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(planes * 4)
        self.relu = nn.ReLU(inplace=True)
        self.se = SELayer(planes * 4, reduction)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        residual = 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)
        out = self.se(out)

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

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

        return out


def SEResNet18(n_classes):
    model = ResNet(SEBasicBlock, [2, 2, 2, 2], num_classes=n_classes)
    model.avgpool = nn.AdaptiveAvgPool2d(1)
    return model

def SEResNet34(n_classes):
    model = ResNet(SEBasicBlock, [3, 4, 6, 3], num_classes=n_classes)
    model.avgpool = nn.AdaptiveAvgPool2d(1)
    return model


def SEResNet50(n_classes):
    model = ResNet(SEBasicBlock, [3, 4, 6, 3], num_classes=n_classes)
    model.avgpool = nn.AdaptiveAvgPool2d(1)
    return model

def SEResNet101(n_classes):
    model = ResNet(SEBasicBlock, [3, 4, 23, 3], num_classes=n_classes)
    model.avgpool = nn.AdaptiveAvgPool2d(1)
```

```
    return model

def SEResNet152(n_classes):
    model = ResNet(SEBasicBlock, [3, 8, 36, 3], num_classes=n_classes)
    model.avgpool = nn.AdaptiveAvgPool2d(1)
    return model
```

Model and Computational Complexity

SE-ResNet-50 vs ResNet-50

- Parameters: 2%~10% additional parameters
- Computation cost: < 1% additional computation (theoretical)
- GPU inference time: 10% additional time
- CPU inference time: < 2% additional time

SENet is a quite simple model with excellent performance and it can easily be embedded without adding more function or layers. Besides the performance we mentioned above, if we remove the SENet in the last three stages of SE-ResNet, the additional parameters decrease greatly from 10% to 2%.

For details, please refer to the papers: ***Squeeze-and-Excitation Networks***

https://openaccess.thecvf.com/content_cvpr_2018/papers/Hu_Squeeze-and-Excitation_Networks_CVPR_2018_paper.pdf