概要：SqueezeNet的工作为以下几个方面：

1. **提出了新的网络架构Fire Module，通过减少参数来进行模型压缩**
2. **使用其他方法对提出的SqeezeNet模型进行进一步压缩**
3. **对参数空间进行了探索，主要研究了压缩比和3*3卷积比例的影响**

这篇文章是 SQUEEZENET: ALEXNET-LEVEL ACCURACY WITH 50X FEWER PARAMETERS AND <0.5MB MODEL SIZE 的解读，在精简部分内容的同时补充了相关的概念。如有错误，敬请指正。

论文链接：http://arxiv.org/abs/1602.07360

代码链接：https://github.com/DeepScale/SqueezeNet

ABSTRACT

近来深层卷积网络的主要研究方向集中在提高正确率。对于相同的正确率水平，更小的CNN架构可以提供如下的优势：

（1）在分布式训练中，与服务器通信需求更小

（2）参数更少，从云端下载模型的数据量小

（3）更适合在FPGA等内存受限的设备上部署。

基于这些优点，本文提出SqeezeNet。它在ImageNet上实现了和AlexNet相同的正确率，但是只使用了1/50的参数。更进一步，使用模型压缩技术，可以将SqueezeNet压缩到0.5MB，这是AlexNet的1/510。

1 INTRODUCTION AND MOTIVATION

对于一个给定的正确率，通常可以找到多种CNN架构来实现与之相近的正确率。其中，参数数量更少的CNN架构有如下优势：

（1）更高效的分布式训练

服务器间的通信是分布式CNN训练的重要限制因素。对于分布式 数据并行 训练方式，通信需求和模型参数数量正相关。小模型对通信需求更低。

（2）减小下载模型到客户端的额外开销

比如在自动驾驶中，经常需要更新客户端模型。更小的模型可以减少通信的额外开销，使得更新更加容易。

（3）便于FPGA和嵌入式硬件上的部署

2 RELATED WORK

2.1 MODEL COMPRESSION

常用的模型压缩技术有：

（1）奇异值分解(singular value decomposition (SVD))[1]

（2）网络剪枝（Network Pruning）[2]：使用网络剪枝和稀疏矩阵

（3）深度压缩（Deep compression）[3]：使用网络剪枝，数字化和huffman编码

（4）硬件加速器（hardware accelerator）[4]

2.2 CNN MICROARCHITECTURE

在设计深度网络架构的过程中，如果手动选择每一层的滤波器显得过于繁复。通常先构建由几个卷积层组成的小模块，再将模块堆叠形成完整的网络。定义这种模块的网络为CNN microarchitecture。

2.3 CNN MACROARCHITECTURE

与模块相对应，定义完整的网络架构为CNN macroarchitecture。在完整的网络架构中，深度是一个重要的参数。

2.4 NEURAL NETWORK DESIGN SPACE EXPLORATION

由于超参数繁多，深度神经网络具有很大的**设计空间（design space）**。通常进行设计空间探索的方法有：

（1）贝叶斯优化

（2）模拟退火

（3）随机搜索

（4）遗传算法

3 SQUEEZENET: PRESERVING ACCURACY WITH FEW PARAMETERS

3.1 ARCHITECTURAL DESIGN STRATEGIES

使用以下三个策略来减少SqueezeNet设计参数

（1）使用1*1卷积代替3*3 卷积：参数减少为原来的1/9

（2）减少输入通道数量：这一部分使用squeeze layers来实现

（3）将欠采样操作延后，可以给卷积层提供更大的激活图：更大的激活图保留了更多的信息，可以提供更高的分类准确率

其中，（1）和（2）可以显著减少参数数量，（3）可以在参数数量受限的情况下提高准确率。

3.2 THE FIRE MODULE

Fire Module是SqueezeNet中的基础构建模块，如下定义 Fire Module ：

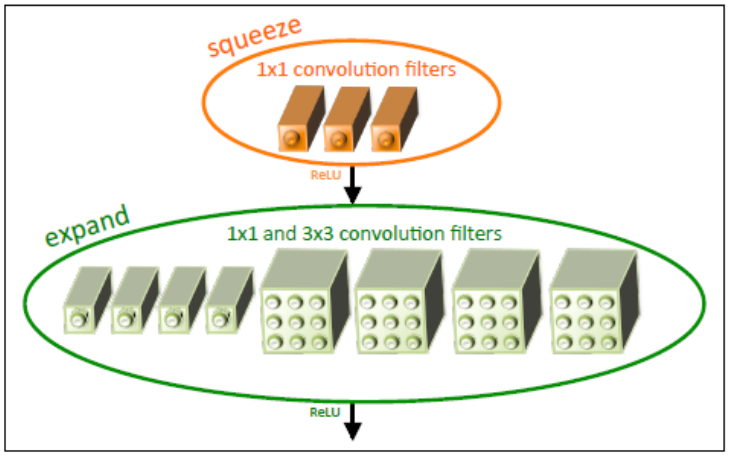
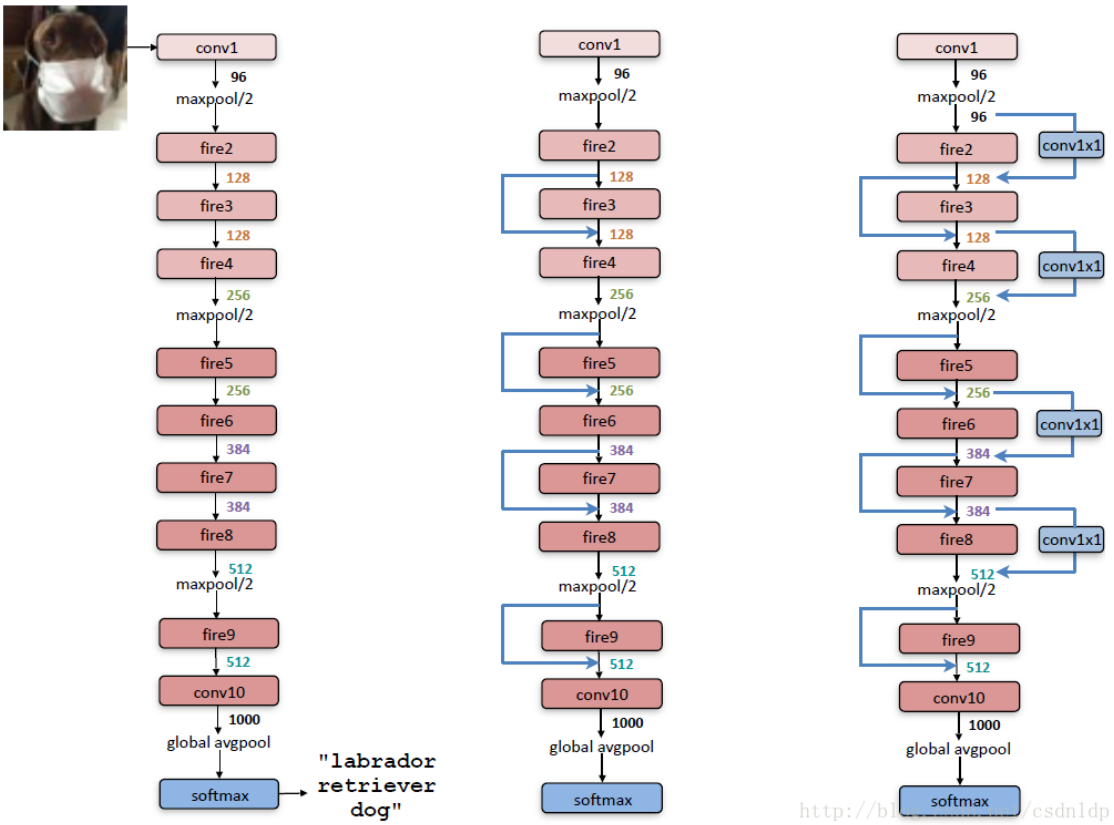


Figure 1: Microarchitectural view: Organization of convolution filters in the **Fire module**. In this example, $s_{1x1} = 3$, $e_{1x1} = 4$, and $e_{3x3} = 4$. We illustrate the convolution filters but not the activations.

1. squeeze convolution layer：只使用1*1 卷积 filter，即以上提到的**策略（1）**
2. expand layer：使用1*1 和3*3 卷积 filter的组合
3. Fire module中使用3ge可调的超参数：s1x1（squeeze convolution layer中1*1 filter的个数）、e1x1（expand layer中1*1 filter的个数）、e3x3（expand layer中3*3 filter的个数）
4. 使用Fire module的过程中，令s1x1 < e1x1 + e3x3，这样squeeze layer可以限制输入通道数量，即以上提到的**策略（2）**

3.3 THE SQUEEZENET ARCHITECTURE

SqueezeNet以卷积层（conv1）开始，接着使用8个Fire modules（fire2-9），最后以卷积层（conv10）结束。每个fire module中的filter数量逐渐增加，并且在conv1，fire4，fire8，和 conv10这几层之后使用步长为2的max-pooling，即将池化层放在相对靠后的位置，这使用了以上的策略（3）。



如上图，左边为原始的SqueezeNet，中间为包含simple bypass的改进版本，最右侧为使用complex bypass的改进版本。在下表中给出了更多的细节。

Table 1: SqueezeNet architectural dimensions. (The formatting of this table was inspired by the Inception2 paper (Ioffe & Szegedy, 2015).)

layer name/type	output size	filter size / stride (if not a fire layer)	depth	s_{1x1} (#1x1 squeeze)	e_{1x1} (#1x1 expand)	e_{3x3} (#3x3 expand)	s_{1x1} sparsity	e_{1x1} sparsity	e_{3x3} sparsity	# bits	#parameter before pruning	#parameter after pruning
input image	224x224x3										-	-
conv1	111x111x96	7x7/2 (x96)	1				100% (7x7)			6bit	14,208	14,208
maxpool1	55x55x96	3x3/2	0									
fire2	55x55x128		2	16	64	64	100%	100%	**33%**	6bit	11,920	5,746
fire3	55x55x128		2	16	64	64	100%	100%	**33%**	6bit	12,432	6,258
fire4	55x55x256		2	32	128	128	100%	100%	**33%**	6bit	45,344	20,646
maxpool4	27x27x256	3x3/2	0									
fire5	27x27x256		2	32	128	128	100%	100%	**33%**	6bit	49,440	24,742
fire6	27x27x384		2	48	192	192	100%	**50%**	**33%**	6bit	104,880	44,700
fire7	27x27x384		2	48	192	192	**50%**	100%	**33%**	6bit	111,024	46,236
fire8	27x27x512		2	64	256	256	100%	**50%**	**33%**	6bit	188,992	77,581
maxpool8	13x12x512	3x3/2	0									
fire9	13x13x512		2	64	256	256	**50%**	100%	**30%**	6bit	197,184	77,581
conv10	13x13x1000	1x1/1 (x1000)	1					**20%** (3x3)		6bit	513,000	103,400
avgpool10	1x1x1000	13x13/1	0									
	activations		parameters				compression info				1,248,424 (total)	**421,098** (total)

因为这是一篇讲解网络压缩的文章，这里顺便提一下参数计算的方法。以上表中的**fire2模块**为例：maxpool1层的输出为55*55*96，一共有96个通道。之后紧接着的Squeeze层有16个1*1*96的卷积filter，注意这里是多通道卷积，为了避免与二维卷积混

淆，在卷积尺寸末尾写上了通道数。这一层的输出尺寸为55*55*16，之后将输出分别送到expand层中的1*1*16（64个）和3*3*16（64个）进行处理，注意这里不对16个通道进行切分。为了得到大小相同的输出，对3*3*16的卷积输入进行尺寸为 1 的zero padding。分别得到55*55*64和55*55*64大小相同的两个feature map。将这两个feature map连接到一起得到55*55*128大小的feature map。考虑到bias参数，这里的参数总数为：

(1*1*96+1)*16+(1*1*16+1)*64+(3*3*16+1)*64＝(1552+1088+9280)=11920

可以看出，Squeeze层由于使用1*1卷积极大地压缩了参数数量，并且进行了降维操作，但是对应的代价是输出特征图的通道数（维数）也大大减少。之后的expand层使用不同尺寸的卷积模板来提取特征，同时将两个输出连接到一起，又将维度升高。但是3*3*16的卷积模板参数较多，远超1*1卷积的参数，对减少参数十分不利，所以作者又针对3*3*16卷积进行了剪枝操作以减少参数数量。从网络整体来看，feature map的尺寸不断减小，通道数不断增加，最后使用平均池化将输出转换成1*1*1000完成分类任务。

3.3.1 OTHER SQUEEZENET DETAILS

以下是网络设计中的一些要点：

（1）为了使 1*1 和 3*3 filter输出的结果又相同的尺寸，在expand modules中，给 3*3 filter的原始输入添加一个像素的边界（zero-padding）。

（2）squeeze 和 expand layers中都是用ReLU作为激活函数

（3）在fire9 module之后，使用Dropout，比例取50%

（4）注意到SqueezeNet中没有全连接层，这借鉴了Network in network的思想

（5）训练过程中，初始学习率设置为0.04，，在训练过程中线性降低学习率。更多的细节参见本项目在github中的配置文件。

（6）由于Caffee中不支持使用两个不同尺寸的filter，在expand layer中实际上是使用了两个单独的卷积层（1*1 filter 和 3*3 filter），最后将这两层的输出连接在一起，这在数值上等价于使用单层但是包含两个不同尺寸的filter。

在github上还有SqueezeNet在其他框架下的实现：MXNet、Chainer、Keras、Torch。

4 EVALUATION OF SQUEEZENET

在表2中，以AlexNet为标准来比较不同压缩方法的效果。

Table 2: Comparing SqueezeNet to model compression approaches. By *model size*, we mean the number of bytes required to store all of the parameters in the trained model.

CNN architecture	Compression Approach	Data Type	Original → Compressed Model Size	Reduction in Model Size vs. AlexNet	Top-1 ImageNet Accuracy	Top-5 ImageNet Accuracy
AlexNet	None (baseline)	32 bit	240MB	1x	57.2%	80.3%
AlexNet	SVD (Denton et al., 2014)	32 bit	240MB → 48MB	5x	56.0%	79.4%
AlexNet	Network Pruning (Han et al., 2015b)	32 bit	240MB → 27MB	9x	57.2%	80.3%
AlexNet	Deep Compression (Han et al., 2015a)	5-8 bit	240MB → 6.9MB	35x	57.2%	80.3%
SqueezeNet (ours)	None	32 bit	4.8MB	**50x**	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	8 bit	4.8MB → 0.66MB	**363x**	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	6 bit	4.8MB → 0.47MB	**510x**	57.5%	80.3%

SVD方法能将预训练的AlexNet模型压缩为原先的1/5，top1正确率略微降低。网络剪枝的方法能将模型压缩到原来的1/9，top1和top5正确率几乎保持不变。深度压缩能将模型压缩到原先的1/35，正确率基本不变。SqeezeNet的压缩倍率可以达到50以上，并且正确率还能有 略微的提升。注意到几时使用未进行压缩的32位数值精度来表示模型，SqeezeNet也比压缩率最高的模型更小，同时表现也更好。

如果将深度压缩（Deep Compression）的方法用在SqeezeNet上，使用33%的稀疏表示和8位精度，会得到一个仅有0.66MB的模型。进一步，如果使用6位精度，会得到仅有0.47MB的模型，同时正确率不变。

此外，结果表明深度压缩不仅对包含庞大参数数量的CNN网络作用，对于较小的网络，比如SqueezeNet，也是有用的。将SqueezeNet的网络架构创新和深度压缩结合起来可以将原模型压缩到1/510。

5 CNN MICROARCHITECTURE DESIGN SPACE EXPLORATION

5.1 CNN MICROARCHITECTURE METAPARAMETERS

在SqueezeNet中，每一个Fire module有3个维度的超参数，即s_{1x1} 、 e_{1x1} 和 e_{3x3}。SqueezeNet一共有8个Fire modules，即一共24个超参数。下面讨论其中一些重要的超参数的影响。为方便研究，定义如下参数：

1. basee：Fire module中expand filter的个数
2. freq：Fire module的个数

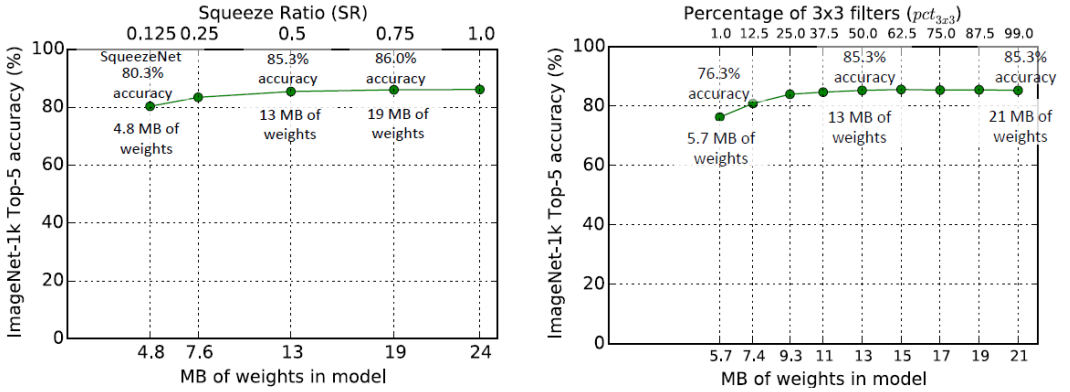
3. incree：在每freq个Fire module之后增加的expand filter个数

4. ei：第i 个Fire module中，expand filters的个数

5. SR：压缩比，即the squeeze ratio ，为squeeze layer中filter个数除以Fire module中filter总个数得到的一个比例

6. pct3x3：在expand layer有1*1和3*3两种卷积，这里定义的参数是3*3卷积个占卷积总个数的比例

下图为实验结果：



(a) Exploring the impact of the squeeze ratio (SR) on model size and accuracy.

(b) Exploring the impact of the ratio of 3x3 filters in expand layers (pct_{3x3}) on model size and accuracy.

Figure 3: Microarchitectural design space exploration.

5.2 SQUEEZE RATIO

Figure 3 中左图给出了压缩比（SR）的影响。压缩比小于0.25时，正确率开始显著下降。

5.3 TRADING OFF 1X1 AND 3X3 FILTERS

Figure 3 中右图给出了3*3卷积比例的影响，在比例小于25%时，正确率开始显著下降，此时模型大小约为原先的44%。超过50%后，模型大小显著增加，但是正确率不再上升。

6 CNN MACROARCHITECTURE DESIGN SPACE EXPLORATION

受ResNet启发，这里探究旁路连接（bypass conection）的影响。在Figure 2中展示了三种不同的网络架构。下表给出了实验结果：

Table 3: SqueezeNet accuracy and model size using different macroarchitecture configurations

Architecture	Top-1 Accuracy	Top-5 Accuracy	Model Size
Vanilla SqueezeNet	57.5%	80.3%	4.8MB
SqueezeNet + Simple Bypass	**60.4%**	**82.5%**	4.8MB
SqueezeNet + Complex Bypass	58.8%	82.0%	7.7MB

注意到使用旁路连接后正确率确实有一定提高。

7 Conclusions

在SqueezeNet提出后，Dense-Sparse-Dense (DSD)[5]使用了新的方法来进行压缩同时提高了精度。

8 Implementation

在这里我们分析SqeezeNet的PyTorch实现，以加深对网络架构的理解。源代码可见：https://github.com/pytorch/vision/blob/master/torchvision/models/squeezenet.py

为方便分析，除去其中不必要的代码。

8.1 Fire module的实现

首先Fire类是对torch.nn.Modules类进行拓展得到的，需要继承Modules类，并实现__init__()方法，以及forward()方法。其中，__init__()方法用于定义一些新的属性，这些属性可以包括Modules的实例，如一个torch.nn.Conv2d，nn.ReLU等。即创建该网络中的子网络，在创建这些子网络时，这些网络的参数也被初始化。接着使用super(Fire, self).__init__()调用基类初始化函数对基类进行初始化。

首先，在Fire类的__init__()函数中，定义了如下几个新增的属性：

1. inplanes：输入向量

2. squeeze：sqeeze layer，由二维1*1卷积组成。其中，参考PyTorch文档，torch.nn.Conv2d 的定义为class torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)，代码中inplanes为输入通道，squeeze_planes为输出通道，卷积模板尺寸为1*1.

3. expand1x1：expand layer中的1*1卷积。

4. expand3x3：expand layer中的3*3卷积。注意，为了使 1*1 和 3*3 filter输出的结果有相同的尺寸，在expand modules中，给3*3 filter的原始输入添加一个像素的边界（zero-padding）

5. 所有的激活函数都选择ReLU。注意inplace=True参数可以在原始的输入上直接进行操作，不会再为输出分配额外的内存，可以节省一部分内存，但同时也会破坏原始的输入。

之后实现foward方法，整个流程如下：

首先，将输入x经过squeeze layer进行卷积操作，再经过 squeeze_activation（） 进行激活，然后将输出分别送到expand1x1和expand3x3中进行卷积和激活操作，最后，使用torch.cat()可以将expand1x1_activation和 expand3x3_activation这两个维度相同的输出张量连接在一起。注意这里的dim=1，即按照列连接，最终得到若干行。

```
class Fire(nn.Module):

    def __init__(self, inplanes, squeeze_planes,
                 expand1x1_planes, expand3x3_planes):
        super(Fire, self).__init__()
        self.inplanes = inplanes
        self.squeeze = nn.Conv2d(inplanes, squeeze_planes, kernel_size=1)
        self.squeeze_activation = nn.ReLU(inplace=True)
        self.expand1x1 = nn.Conv2d(squeeze_planes, expand1x1_planes,
                                   kernel_size=1)
        self.expand1x1_activation = nn.ReLU(inplace=True)
        self.expand3x3 = nn.Conv2d(squeeze_planes, expand3x3_planes,
                                   kernel_size=3, padding=1)
        self.expand3x3_activation = nn.ReLU(inplace=True)

    def forward(self, x):
        x = self.squeeze_activation(self.squeeze(x))
        return torch.cat([
            self.expand1x1_activation(self.expand1x1(x)),
            self.expand3x3_activation(self.expand3x3(x))
        ], 1)
```

以上实现了SqeezeNet中最重要的Fire module，为搭建整体网络做好了准备。接下来定义SqueezeNet类，它同样继承自nn.Module，这里实现了version=1.0和version=1.1两个SqueezeNet版本。区别在于1.0只有 AlexNet的1/50的参数，而1.1在1.0的基础上进一步压缩，参数略微减少，计算量降低为1.0的40%左右。SqueezeNet类定义了如下属性：

1. num_classes：分类的类别个数

2. self.features：定义了主要的网络层，nn.Sequential()是PyTorch中的序列容器（sequential container），可以按照顺序将Modules添加到其中，这也是网络宏观架构实现的重要步骤。要理解这一部分代码，最好结合表1中的细节，并且自己计算一遍卷积操作后的feature map的尺寸。注意表1中的输入图片尺寸是227*227而不是224*224，否则跟之后的输出尺寸对不上。

3. 注意nn.MaxPool2d()函数中ceil_mode=True会对池化结果进行向上取整而不是向下取整。

4. 最后一个卷积层的初始化方法与其他层不同，在接下来的for循环中，定义了不同层的初始化方法，可以看到，最后一层使用了均值为0，方差为0.01的正太分布初始化方法，其余层使用He Kaiming论文中的均匀分布初始化方法。同时这里使用了num_classes参数，可以调整分类类别数目。并且所有的bias初始化为零。

5. self.classifier：定义了网络末尾的分类器模块，注意其中使用了nn.Dropout()

6. forward()方法：可以看到由features模块和classifier模块构成。

```
class SqueezeNet(nn.Module):

    def __init__(self, version=1.0, num_classes=1000):
        super(SqueezeNet, self).__init__()
        if version not in [1.0, 1.1]:
            raise ValueError("Unsupported SqueezeNet version {version}:"
                             "1.0 or 1.1 expected".format(version=version))
        self.num_classes = num_classes
        if version == 1.0:
            self.features = nn.Sequential(
                nn.Conv2d(3, 96, kernel_size=7, stride=2),
                nn.ReLU(inplace=True),
                nn.MaxPool2d(kernel_size=3, stride=2, ceil_mode=True),
                Fire(96, 16, 64, 64),
                Fire(128, 16, 64, 64),
                Fire(128, 32, 128, 128),
                nn.MaxPool2d(kernel_size=3, stride=2, ceil_mode=True),
                Fire(256, 32, 128, 128),
                Fire(256, 48, 192, 192),
                Fire(384, 48, 192, 192),
                Fire(384, 64, 256, 256),
                nn.MaxPool2d(kernel_size=3, stride=2, ceil_mode=True),
                Fire(512, 64, 256, 256),
            )
        else:
            self.features = nn.Sequential(
```

```
            nn.Conv2d(3, 64, kernel_size=3, stride=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2, ceil_mode=True),
            Fire(64, 16, 64, 64),
            Fire(128, 16, 64, 64),
            nn.MaxPool2d(kernel_size=3, stride=2, ceil_mode=True),
            Fire(128, 32, 128, 128),
            Fire(256, 32, 128, 128),
            nn.MaxPool2d(kernel_size=3, stride=2, ceil_mode=True),
            Fire(256, 48, 192, 192),
            Fire(384, 48, 192, 192),
            Fire(384, 64, 256, 256),
            Fire(512, 64, 256, 256),
        )
        # Final convolution is initialized differently form the rest
        final_conv = nn.Conv2d(512, self.num_classes, kernel_size=1)
        self.classifier = nn.Sequential(
            nn.Dropout(p=0.5),
            final_conv,
            nn.ReLU(inplace=True),
            nn.AvgPool2d(13)
        )

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                if m is final_conv:
                    init.normal(m.weight.data, mean=0.0, std=0.01)
                else:
                    init.kaiming_uniform(m.weight.data)
                if m.bias is not None:
                    m.bias.data.zero_()

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x.view(x.size(0), self.num_classes)
```

在之后的代码中定义了完整的squeezenet1_0和squeezenet1_1，如果需要的话，可以使用PyTorch的预训练模型。

```
def squeezenet1_0(pretrained=False, **kwargs):
    r"""SqueezeNet model architecture from the `"SqueezeNet: AlexNet-level
    accuracy with 50x fewer parameters and <0.5MB model size"
    <https://arxiv.org/abs/1602.07360>`_ paper.

    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
    """
    model = SqueezeNet(version=1.0, **kwargs)
    if pretrained:
        model.load_state_dict(model_zoo.load_url(model_urls['squeezenet1_0']))
    return model
```

```
def squeezenet1_1(pretrained=False, **kwargs):
    r"""SqueezeNet 1.1 model from the `official SqueezeNet repo
    <https://github.com/DeepScale/SqueezeNet/tree/master/SqueezeNet_v1.1>`_.
    SqueezeNet 1.1 has 2.4x less computation and slightly fewer parameters
    than SqueezeNet 1.0, without sacrificing accuracy.

    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
    """
    model = SqueezeNet(version=1.1, **kwargs)
    if pretrained:
        model.load_state_dict(model_zoo.load_url(model_urls['squeezenet1_1']))
    return model
```

Refference

1. E.L Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within

convolutional networks for efficient evaluation. In NIPS, 2014. ↩

2. S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural

networks. In NIPS, 2015b. ↩

3. S. Han, H. Mao, and W. Dally. Deep compression: Compressing DNNs with pruning, trained

quantization and huffman coding. arxiv:1510.00149v3, 2015a. ↩

4. Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J

Dally. Eie: Efficient inference engine on compressed deep neural network. International Symposium

on Computer Architecture (ISCA), 2016a. ↩

5. Song Han, Jeff Pool, Sharan Narang, Huizi Mao, Shijian Tang, Erich Elsen, Bryan Catanzaro, John

Tran, and William J. Dally. Dsd: Regularizing deep neural networks with dense-sparse-dense

training flow. arXiv:1607.04381, 2016b. ↩