ResNet in PyTorch

Constructing ResNet from scratch is quite simple in PyTorch (thanks to its `nn.Module` and all kinds of `nn.functional`).

Since blocks' structures are identical in ResNet and we just simply repeatedly call same blocks to make to, say, 152 Layers, we start by first defining `BasicBlock` (for resnet < 50) and `Bottleneck` block(for resnet > 50).

```
import torch
import torch.nn as nn
import torch.utils.model_zoo
```

```
print(torch.cuda.get_device_name(0))
```

```
Quadro M2200
```

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2 $\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

`planes`: same as Channels

```
def conv3x3(inplanes, outplanes, stride=1):
    # basic conv op in resnet
    return nn.Conv2d(inplanes, outplanes, kernel_size=3, stride=stride,
                     padding=1, bias=False)
#### For 3x3 conv, padding=1 can preserve h,w --> input 100x100 output 100x100
```

```
def conv1x1(inplanes, outplanes, stride=1):
    return nn.Conv2d(inplanes, outplanes, kernel_size=1, stride=1,
                     bias=False)

##### For 1x1 conv, target is to modify #channels!
####                and 1x1 no need to padding.
```

`downsample()`:
is just a conv1x1 + BN to modify #channels when going from Layer[i] --> Layer[i+1]
e.g. layer[0] (64 channels) to layer[1] (128 channels). btw, inside each layer, when data are pass from Block to next Block, #channels will be the same, so only need to downsample() from layer to next layer.

- `BasicBlock()` : For resnet18 and resnet34

18-layer	34-layer
$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$
$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$
$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$
$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$

- `Bottleneck()` : For resnet50, 101, 152

50-layer	101-layer	152-layer
7×7, 64, stride 2		
3×3 max pool, stride 2		
$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$

Basic Block for ResNet < 50

```
class BasicBlock(nn.Module):
    #### define layers in __init__
    def __init__(self, inplanes, planes, stride=1, downsample=None):

        super(BasicBlock).__init__()

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

    def forward(self, x):
        # The Identity Path (Shortcut Path)
        identity = x
```

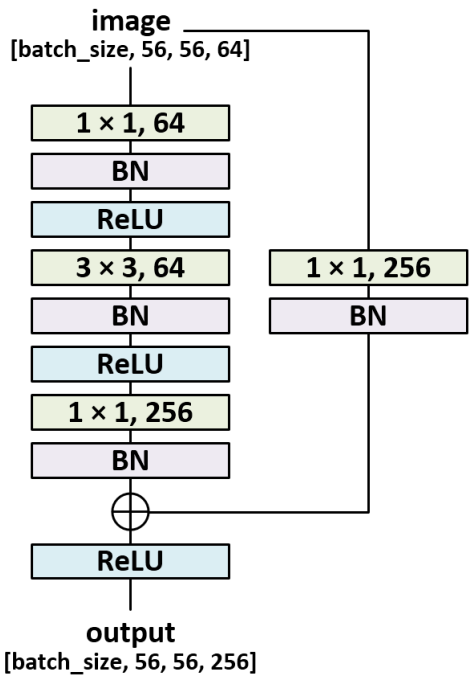
```
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

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

        # [in ResNet] when Layer to Next Layer, check if necessary to modify # channels
        if self.downsample is not None:
            out = self.downsample(out)

        # MERGE !!!
        out = out + identity
        out = self.relu(out)
        return out
```

Bottleneck Block for ResNet>50




```
class Bottleneck(nn.Module):
    expansion = 4  ## Use to do bottle neck!
    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(Bottleneck, self).__init__()
        self.conv1 = conv1x1(inplanes, planes)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = conv3x3(planes, planes, stride)
        self.bn2 = nn.BatchNorm2d(planes)
        self.conv3 = conv1x1(planes, planes * self.expansion)
```

```
        self.bn3 = nn.BatchNorm2d(planes * self.expansion)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):

        # The Identity Path (Shortcut Path)
        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)

        #### Do conv1x1 to expand the Identity's # channels!!!!!!
        ####    in order to do addition with the convol_path
        if self.downsample is not None:
            identity = self.downsample(x)

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

        return out
```

```
class ResNet(nn.Module):

    def __init__(self, block, layers, num_classes=1000, zero_init_residual=False):

        super(ResNet, self).__init__()

        #### First few conv ops are same for any ResNet
        self.inplanes = 64
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        #### Different ResNet have diff number of each layer
        #### (check below the helper: `_make_layer()` )
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * block.expansion, num_classes)
```

```
        #### Initialize weights and biases, according to its type (Conv layer or BN
layer)
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

        # Zero-initialize the last BN in each residual branch (ShortCut Path),
        # so that the residual branch starts with zeros, and each residual block behaves
like an identity.
        # This improves the model by 0.2~0.3% according to
https://arxiv.org/abs/1706.02677
        if zero_init_residual:
            for m in self.modules():
                if isinstance(m, Bottleneck):
                    nn.init.constant_(m.bn3.weight, 0)
                elif isinstance(m, BasicBlock):
                    nn.init.constant_(m.bn2.weight, 0)

    def forward(self, x):
        # first few layers
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        # Layer by Layer (#channels: 64 -> 128 -> 256 -> 512)
        x = self.layer1(x)  # 64
        x = self.layer2(x)  # 128
        x = self.layer3(x)  # 256
        x = self.layer4(x)  # 512
        # final. The last layer is FC (for classification)
        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x

    def _make_layer(self, block, planes, blocks, stride=1):
        # block = BasicBlock or Bottleneck ;
        # planes = this layer's Channel (or planes) 64, 128 ...
        # blocks = [#layer0, #layer1, #layer2, #layer3]
        downsample = None

        #### notice, only layer[0] has stride==1
        if stride != 1 or self.inplanes != planes * block.expansion:
            # if last_layer's # channels != next_layer's # channels
            #    need to downsample to match their channels.
            downsample = nn.Sequential(
                conv1x1(self.inplanes, planes * block.expansion, stride),
                nn.BatchNorm2d(planes * block.expansion),
```

```
            )

        layers = []
        # the first block in this Layer !!
        layers.append(block(self.inplanes, planes, stride, downsample))

        self.inplanes = planes * block.expansion  # Update the self.inplanes (initially
64, then will be 128, 256, 512)
        # For the rest of the Blocks, self.inplanes == planes (e.g. self.inplanes ==
planes == 128)
        for _ in range(1, blocks):
            layers.append(block(self.inplanes, planes))

        return nn.Sequential(*layers)
```

Instantiate Model

```
__all__ = ['ResNet', 'resnet18', 'resnet34', 'resnet50', 'resnet101',
           'resnet152']


model_urls = {
    'resnet18': 'https://download.pytorch.org/models/resnet18-5c106cde.pth',
    'resnet34': 'https://download.pytorch.org/models/resnet34-333f7ec4.pth',
    'resnet50': 'https://download.pytorch.org/models/resnet50-19c8e357.pth',
    'resnet101': 'https://download.pytorch.org/models/resnet101-5d3b4d8f.pth',
    'resnet152': 'https://download.pytorch.org/models/resnet152-b121ed2d.pth',
}
```

resnet18 and 34 use Basicblock

```
def resnet18(pretrained=False, **kwargs):
    # resnet18 and resnet 34 ----> BasicBlock!!  (for resnet>50 ---> BottleNeck Block!!!)
    model = ResNet(BasicBlock, layers=[2, 2, 2, 2], **kwargs)  #### 2 layer[0](with 64
channels); 2 layer[1](with 128 channels) ....
    # If pretrained, then download pretrained weights and biases on PyTorch.org
    if pretrained:
        model.load_state_dict(model_zoo.load_url(model_urls['resnet18']))
    return model
```

```
def resnet34(pretrained=False, **kwargs):
    model = ResNet(BasicBlock, [3, 4, 6, 3], **kwargs)
    if pretrained:
        model.load_state_dict(model_zoo.load_url(model_urls['resnet34']))
    return model
```

resnet>50 use Bottleneck Block

```
def resnet50(pretrained=False, **kwargs):
    # usd Bottleneck block
    model = ResNet(Bottleneck, layers=[3, 4, 6, 3], **kwargs)
    if pretrained:
        model.load_state_dict(model_zoo.load_url(model_urls['resnet50']))
    return model
```

```
def resnet101(pretrained=False, **kwargs):

    model = ResNet(Bottleneck, [3, 4, 23, 3], **kwargs)
    if pretrained:
        model.load_state_dict(model_zoo.load_url(model_urls['resnet101']))
    return model
```

```
def resnet152(pretrained=False, **kwargs):

    model = ResNet(Bottleneck, [3, 8, 36, 3], **kwargs)
    if pretrained:
        model.load_state_dict(model_zoo.load_url(model_urls['resnet152']))
    return model
```