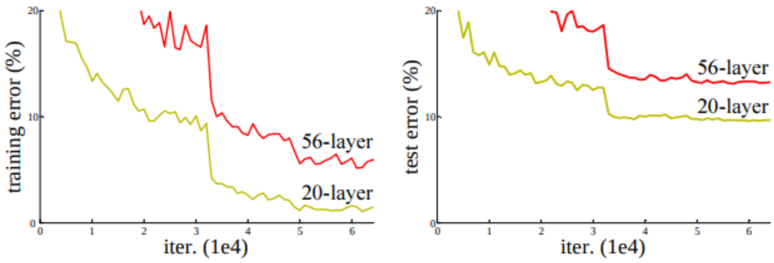
Niko Gamulin
Nov 1, 2020 · 4 min read · Listen

ResNet Implementation with PyTorch from Scratch

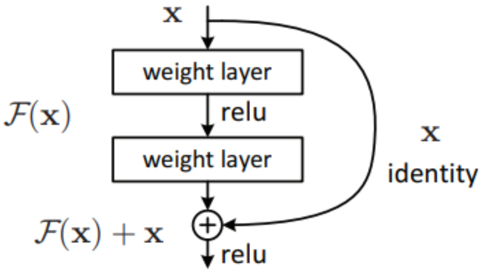
In the past decade, we have witnessed the effectiveness of convolutional neural networks. Khrichevsky's seminal ILSVRC2012-winning convolutional neural network has inspired various architecture proposals. In general, the deeper the network, the greater is its learning capacity.

While increasing the network depth, however, the accuracy gets saturated and then degrades rapidly. By observing the training errors of various network depths, it is evident that the degradation is not caused by overfitting (in case of overfitting, the training error decreases, and the test error increases). The training accuracy degradation indicates that the deeper network architectures are harder to optimize due to vanishing/exploding gradients.



Comparison of training error(left) and test error (right) using convolutional neural networks without skip connections.

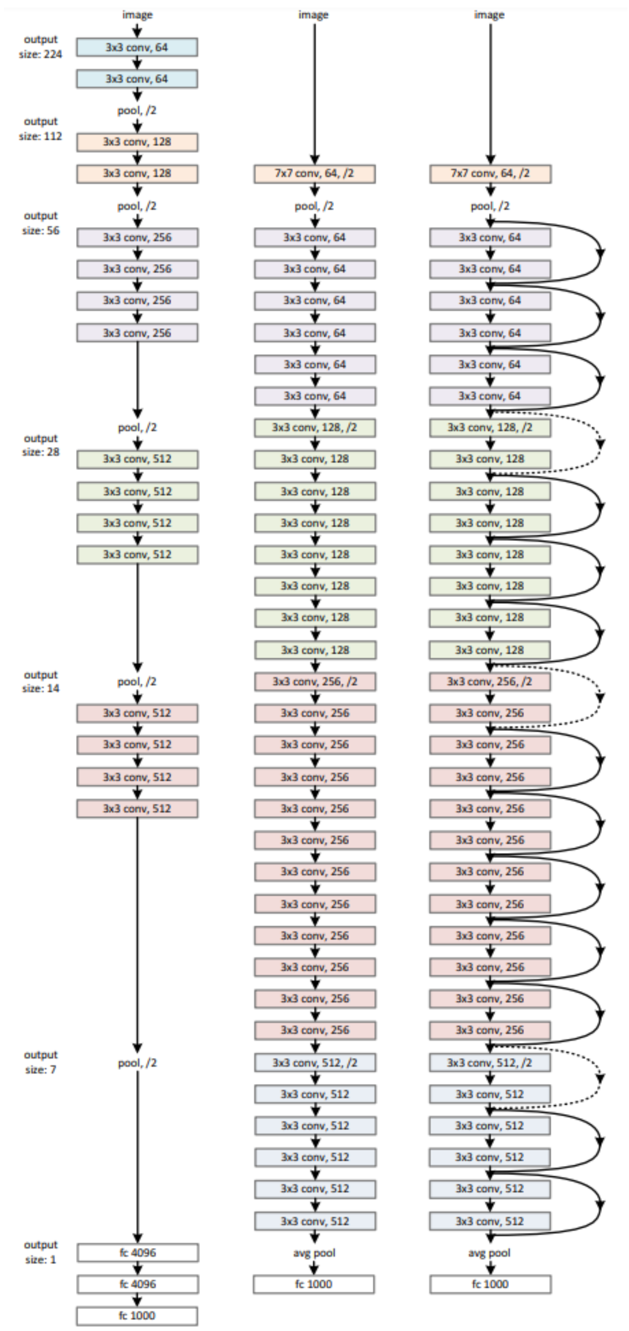
The network, presented in "Deep Residual Learning for Image Recognition", addressed the degradation problem by implementing skip connections. The authors hypothesize that it is easier to optimize the network with skip connections than to optimize the original, unreferenced mapping. To the extreme, if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers.



Residual block

. . .

Network Implementation



left: VGG19, middle: a plain network with 34 parameter layers, right: a residual network with skip connections.

Translation of tabular representation to code

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				
		[3×3, 64 3×3, 64] ×2	[3×3, 64 3×3, 64] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3
conv3_x	28×28	[3×3, 128 3×3, 128] ×2	[3×3, 128 3×3, 128] ×4	[1×1, 128 3×3, 128 1×1, 512] ×4	[1×1, 128 3×3, 128 1×1, 512] ×4	[1×1, 128 3×3, 128 1×1, 512] ×8
conv4_x	14×14	[3×3, 256 3×3, 256] ×2	[3×3, 256 3×3, 256] ×6	[1×1, 256 3×3, 256 1×1, 1024] ×6	[1×1, 256 3×3, 256 1×1, 1024] ×23	[1×1, 256 3×3, 256 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512 3×3, 512] ×2	[3×3, 512 3×3, 512] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3
	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

representation of residual networks with 18, 34, 50, 101, and 152 layers.

conv1

The first layer is a convolution layer with 64 kernels of size (7 x 7), and stride 2. the input image size is (224 x 224) and in order to keep the same dimension after convolution operation, the padding has to be set to 3 according to the following equation:

```
n_out = ((n_in + 2p - k) / s) + 1
n_out - output dimension
n_in - -input dimension
p - padding
s - stride
```

maxpool1

The second layer is a max-pooling layer with kernel size (3x3) and stride 2. In order to get the size (56 x 56) at the output, the padding has to be set to 1

Convolutional Blocks

all the architectures consist of 4 convolutional groups of blocks. In the case of ResNet18, there are [2, 2, 2, 2] convolutional blocks of 2 layers, and the number of kernels in the first layers is equal to the number of layers in the second layer. Similarly, in the case of ResNet34, there are [3, 4, 6, 3] blocks of 2 layers and the numbers of kernels of the first and second layers are the same.

In the case of ResNet50, ResNet101, and ResNet152, there are 4 convolutional groups of blocks and every block consists of 3 layers.

Conversely to the shallower variants, in this case, the number of kernels of the third layer is three times the number of kernels in the first layer.

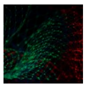
The convolutional block is defined as the following class:

```
class Block(nn.Module):
    def __init__(self, num_layers, in_channels,
out_channels, identity_downsample=None, stride=1):
        assert num_layers in [18, 34, 50, 101, 152],
"should be a a valid architecture"
        super(Block, self).__init__()
        self.num_layers = num_layers
        if self.num_layers > 34:
            self.expansion = 4
        else:
            self.expansion = 1
        # ResNet50, 101, and 152 include additional layer
of 1x1 kernels
        self.conv1 = nn.Conv2d(in_channels, out_channels,
kernel_size=1, stride=1, padding=0)
        self.bn1 = nn.BatchNorm2d(out_channels)
        if self.num_layers > 34:
            self.conv2 = nn.Conv2d(out_channels,
out_channels, kernel_size=3, stride=stride, padding=1)
        else:
            # for ResNet18 and 34, connect input directly
to (3x3) kernel (skip first (1x1))
            self.conv2 = nn.Conv2d(in_channels,
out_channels, kernel_size=3, stride=stride, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.conv3 = nn.Conv2d(out_channels, out_channels *
self.expansion, kernel_size=1, stride=1, padding=0)
        self.bn3 = nn.BatchNorm2d(out_channels *
self.expansion)
        self.relu = nn.ReLU()
        self.identity_downsample = identity_downsample

    def forward(self, x):
        identity = x
        if self.num_layers > 34:
            x = self.conv1(x)
            x = self.bn1(x)
            x = self.relu(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = self.relu(x)
        x = self.conv3(x)
        x = self.bn3(x)

        if self.identity_downsample is not None:
            identity = self.identity_downsample(identity)

        x += identity
        x = self.relu(x)
        return x
```

Putting all together

the whole network is defined as the following class:

```
class ResNet(nn.Module):
    def __init__(self, num_layers, block, image_channels,
num_classes):
        assert num_layers in [18, 34, 50, 101, 152],
f'ResNet{num_layers}: Unknown architecture! Number of
```


Scratching Linear Regression using PyTorch - Part 1

Pre-processing Market1501 Person Reid Dataset for PyTorch

```
layers has ' \
                                                     f'to
be 18, 34, 50, 101, or 152 '
        super(ResNet, self).__init__()
        if num_layers < 50:
            self.expansion = 1
        else:
            self.expansion = 4
        if num_layers == 18:
            layers = [2, 2, 2, 2]
        elif num_layers == 34 or num_layers == 50:
            layers = [3, 4, 6, 3]
        elif num_layers == 101:
            layers = [3, 4, 23, 3]
        else:
            layers = [3, 8, 36, 3]
        self.in_channels = 64
        self.conv1 = nn.Conv2d(image_channels, 64,
kernel_size=7, stride=2, padding=3)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=3,
stride=2, padding=1)

        # ResNetLayers
        self.layer1 = self.make_layers(num_layers, block,
layers[0], intermediate_channels=64, stride=1)
        self.layer2 = self.make_layers(num_layers, block,
layers[1], intermediate_channels=128, stride=2)
        self.layer3 = self.make_layers(num_layers, block,
layers[2], intermediate_channels=256, stride=2)
        self.layer4 = self.make_layers(num_layers, block,
layers[3], intermediate_channels=512, stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * self.expansion,
num_classes)

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

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.reshape(x.shape[0], -1)
        x = self.fc(x)
        return x

    def make_layers(self, num_layers, block,
num_residual_blocks, intermediate_channels, stride):
        layers = []

        identity_downsample =
nn.Sequential(nn.Conv2d(self.in_channels,
intermediate_channels*self.expansion, kernel_size=1,
stride=stride),

nn.BatchNorm2d(intermediate_channels*self.expansion))
        layers.append(block(num_layers, self.in_channels,
intermediate_channels, identity_downsample, stride))
        self.in_channels = intermediate_channels *
self.expansion # 256
        for i in range(num_residual_blocks - 1):
            layers.append(block(num_layers,
self.in_channels, intermediate_channels)) # 256 -> 64, 64*4
(256) again
        return nn.Sequential(*layers)
```

jupyter notebook is available here

. . .

References

ImageNet Classification with Deep Convolutional Neural Networks

Deep Residual Learning for Image Recognition — original paper