Indian Institute of Technology Bombay

OBJECT DETECTION IN CIFAR10

In-Semester Project

Harsh Maheshwari

170020037@iitb.ac.in

Supervised by, *Prof. P Balamurugan, IEOR Department, IIT Bombay*

15 October, 2018

Abstract

This paper is a re implementation of the ResNet architecture for developing learning skills and methodologies for object recognition in images. Here CIFAR10 data set is used to understand and develop an implementation of Residual Networks of different number of layer starting from 18 then 34, 50, 101 and 152. Change in the accuracy based on position of activation layer was studied.

1 Introduction

CIFAR-10 is an established computer-vision dataset used for object recognition. It is a subset of the 80 million tiny images dataset and consists of 60,000 32x32 color images containing one of 10 object classes, with 6000 images per class. It was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The label classes in the dataset are:

1. airplane
2. automobile
3. bird
4. cat
5. deer
6. dog
7. frog
8. horse
9. ship
10. truck

Deeper neural networks are more difficult to train. In residual learning framework to ease the training of networks that are substantially deeper than other previous networks ResNet explicitly reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning un-referenced functions. These residual networks are easier to optimize, and can gain accuracy from their considerably increased depth. The depth of representations is of central importance for many visual recognition tasks. Solely due to our extremely deep representations When deeper networks are able to start converging, a degradation problem starts. With the network depth increasing, accuracy gets saturated which is not very surprising today and then degrades rapidly.Such degradation is not caused by overfitting, and adding more layers to a suitably deep model leads to higher training error. There exists a solution by construction to the deeper model: the added layers are identity mapping, and the other layers are copied from the learned shallower model. The existence of this constructed solution indicates that a deeper model should produce no higher training error than its shallower counterpart. Instead of hoping each few stacked layers directly fit a desired underlying mapping, Resnet

architecture explicitly let these layers fit a residual mapping. Formally, denoting the desired underlying mapping as H(x), and lets the stacked nonlinear layers fit another mapping of
F(x) := H(x) − x. The original mapping is recast into F(x)+x. The main concept is that it is easier to optimize the residual mapping 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.

2 Residual Learning

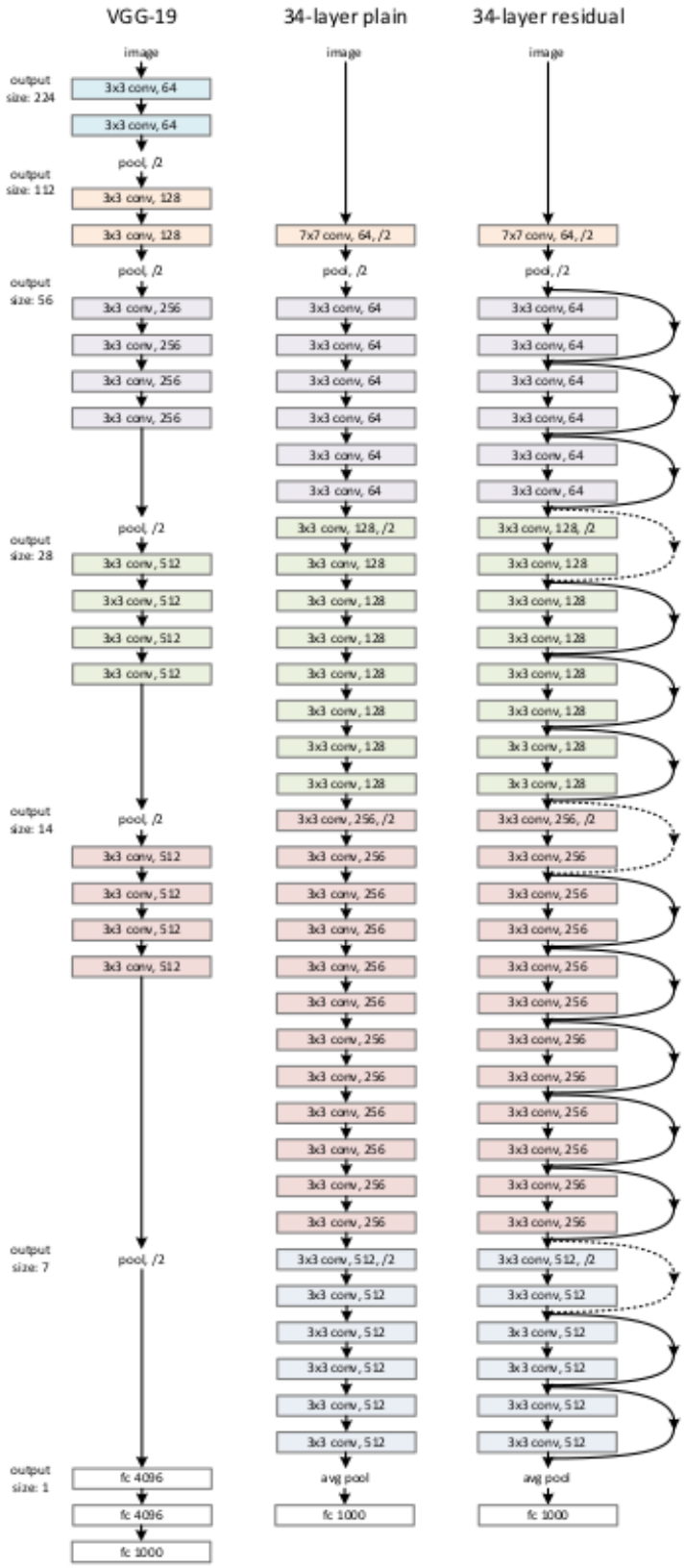
H(x) is an underlying mapping to be fit by a few stacked layers (not necessarily the entire net), with x denoting the inputs to the first of these layers. Thus if multiple nonlinear layers can asymptotically approximate complicated functions 2 , then it is equivalent to hypothesize that they can asymptotically approximate the residual functions, i.e.,
H(x) − x (assuming that the input and output are of the same dimensions). So rather than expect stacked layers to approximate H(x), we explicitly let these layers approximate a residual function F(x) := H(x) − x. The original function thus becomes F(x)+x. Although both forms should be able to asymptotically approximate the desired functions (as hypothesized), the ease of learning might be different.
The equation below is used as building block for the residual network

$$\mathbf{y} = \mathcal{F}\left(\mathbf{x}, \{W_i\}\right) + W_s\mathbf{x} \tag{1}$$

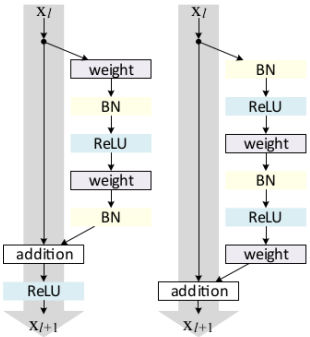
Here x and y are the input and output vectors of the layers considered. The function $\mathcal{F}\left(\mathbf{x}, \{W_i\}\right)$ represents the residual mapping to be learned. The dimensions of $\mathbf{x}$ and $\mathcal{F}$ must be equal,for this perform a linear projection W_s by the shortcut connections to match the dimensions

Figure 1 (a) describes the layers of a simple VGG-19 model and then a plain 34 parameter layer Deep neural network and then the residual network with 34 parameter layers. The curve arch in the residual networks represent the shortcut in the system.

Figure 1 (b) describes a newer interesting concept of pre-activation that is the use of Relu activation before any weights are introduced this feature is also experimented in this paper.



(a) Source : Directly taken from Deep Residual Learning Paper
Left: the VGG-19 model, not so deep network
Middle: a plain network with 34 parameter layers
Right: a residual network with 34 parameter layers
The dotted shortcuts increase dimensions



(a) original (b) proposed

(b) Source : Directly taken from Identity Mappings in Deep Residual Networks paper original is the Residual layer and proposed is the Pre-activation Residual layer

3 Implementation of Code

The code is entirely implemented in python using libraries like PyTorch Torch, Torchvision. First a general model development file is created which can be used to run different architectures from the same file by just changing the argument 'model'. Then the file resnet.py describes the complete model architecture consisting of basic blocks and bottleneck classes and a file class resnet to combine all other miscellaneous layers.

resnet.py file is described below :

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


class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_planes, planes, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes,
        kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes,
        kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion*planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion*planes,
                kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion*planes)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out


class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, in_planes, planes, stride=1):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, 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, self.expansion*planes,
        kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(self.expansion*planes)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion*planes:
```

```
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion*planes,
                kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion*planes)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = F.relu(self.bn2(self.conv2(out)))
        out = self.bn3(self.conv3(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out


class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        super(ResNet, self).__init__()
        self.in_planes = 64

        self.conv1 = nn.Conv2d(3, 64,
        kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.layer1 = self._make_layer(
        block, 64,num_blocks[0], stride=1)
        self.layer2 = self._make_layer(
        block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(
        block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(
        block, 512, num_blocks[3], stride=2)
        self.linear = nn.Linear(
        512*block.expansion, num_classes)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes * block.expansion
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out


def ResNet18():
    return ResNet(BasicBlock, [2,2,2,2])

def ResNet34():
    return ResNet(BasicBlock, [3,4,6,3])

def ResNet50():
    return ResNet(Bottleneck, [3,4,6,3])

def ResNet101():
    return ResNet(Bottleneck, [3,4,23,3])
```

```
def ResNet152():
    return ResNet(Bottleneck, [3,8,36,3])
```

4 Results

After analysing CIFAR10 dataset using residual networks with post activation (original) and pre activation (modified) i could derive the results that pre activation using relu works better than the older version without pre activation. 95.11% accuracy was obtained from the pre-activation resent18 while 93.02% accuracy was obtained from simple resnet18 architecture.

5 References

1. Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009, http://www.cs.toronto.edu/ kriz/learning-features-2009-TR.pdf
2. Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun Deep Residual Learning for Image Recognition. arXiv:1512.03385
3. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun Identity Mappings in Deep Residual Networks arXiv:1603.05027v3
4. K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In ICLR, 2015.