OBProx-SG paper-recurrence-note

Zhenwei Lin

Date: November 5, 2020

1 Model theories

1.1 MobileNetV1

Many papers on small networks focus on only on size but do not consider speed.

A different approach for obtaining small networks is shrinking, facotrizing or compressing pretained networks. Another method for training small networks is distillation which uses a larger net-work to teach a smaller network.

We then describe the MobileNet network structure and conclude with descriptions of the two model shrinking hyper-parameters width multiplier and resolution multiplier.

1.1.1 MobileNet Architecture

A standard convolution both filters and combines inputs into a new set of outputs in one step.

The depthwise separable convolution splits this into two layers, a separate layer for filtering and a separate layer for combining.**This factorization has the effect of drastically reducting computation and model size**

Now we analysis the different computational cost between the two convolution methods.

Input dimension is $D_F \times D_F \times M$,where D_F is spatial width and height of a square input feature map, M is the number of input channels

A standard convolutional layer is parameterized by convolution kernel K of size $D_K \times D_K \times M \times N$ where D_k is the spatial dimension of the kernel assumed to be square and M is number of input channels and N is the number of output channels as defined previously.

The output feature map for standard convolution assuming stride 1 and padding is computed as :

$$G_{k,l,n} = \sum_{i,j,m} K_{i,j,m,n} \cdot F_{k+i-1,l+j-1,m}$$

so standard convolutions have the computational cost of :

$$D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F$$

Now consider the second method is to use depthwise convolutions adn pointwise convolutions.

Step1: Depthwise convolution with one filter per input channel can be written as:

$$\hat{G}_{k,l,m} = \sum_{i,j} \hat{K}_{i,j,m} \cdot F_{k+i-1,l+j-1,m}$$

where $\hat{K}$ is the depthwise convolutional kernel of size $D_K \times D_K \times M$,so the depthwise convolution has a computational cost of

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F$$

Depthwise convolution is extremely efficient relative to standard convolution.**However it only filters input channels, it does not combine them to create new features.**

Step2: Depthwise separable convolution. Using N 1×1 convolution kernels. And the cost is

$$M \cdot N \cdot D_F \cdot D_F$$

By expressing convolution as two step process of filtering and combining we get a reduction in computation of :

$$\frac{D_k \cdot D_k \cdot M \cdot D_F \cdot +M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F} = \frac{1}{N} + \frac{1}{{D_K}^2}$$

1.1.2 Network Structure and Training

The MobileNet structure is built on depthwise separable convolutions as metioned in the previous section except for **first layer which is a full convolution.**

The MobileNet architecture is defined in Table 1.

All layers are followed by a batchnorm and ReLU nonlineraity with the exception of the final fully connected layer which has no nonlinearity and feeds into a softmax layer for classification.

Figure 2contrasts a layer with regular convolutions, batchnorm and ReLU nonlinearity to the factorized layer with depthwise convolution, 1×1 pointwise convolution as well as batchnorm and ReLU after each convolutional layer.

Batchnorm: is a method used to make artificial neural networks faster and more stable through normalization of the input layer by recentering and re-scaling. It was believed that it can mitigate the problem of internal covariate shift, where parameter initialization and changes in the distribution of the inputs of each layer affect the learning rate of the network.

Motivation of Batchnorm: the phenomenon of internal covariate shift

Each layer of a neural network has inputs with a corresponding distribution, which is affected during the training process by the randomness in the parameter initialization and the ranodmness in the input data. The effect of these sources of randomness on the distribution of the inputs to interval layers during training is described as internal covariate shift.

(An important hypothesis in Machine learning: training data has the same distribution with test data; Then, refine to each layer of the neural network, the distribution is inconsistent in each round of training.

Then the relative training effect cannot be guaranteed, so it is called covariate shift between layers.)

Batch Normalizing Transform:

Use B to denote a mini-batch of size m o fthe entire training set. The empirical mean and variance of B could thus be denoted as

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \quad \text{and} \quad \sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

For a layer of the network with d-dimension input, $x = (x^{(1)}, \ldots, x^{(d)})$,each dimension of its input is then normalized

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_B^{(k)^2} + \epsilon}}$$

where $k \in [1, d]$ and $i \in [1, m]$, ϵ is added in the denominator for numerical stability and is and arbitrarily small constant. The resulting normalized activation $\hat{x}^{(k)}$ have zero mean and unit variance, if ϵ is not taken into account.

To restore the representation power of the network, a transformation step then follows as:

$$y_i^{(k)} = \gamma^{(k)}\hat{x}_i^{(k)} + \beta^{(k)}$$

where the parameter $\gamma^{(k)}$ and $\beta^{(k)}$ are subsequently learned in the optimization process.

Formally, the operation that implements batch normalization is a transform $BN_{\gamma^{(k)},\beta^{(k)}}(x^{(k)}) : x_{1\ldots m}^{(k)} \rightarrow y_{1\ldots m}^{(k)}$ called batchnormalizing transform.

The reason of restoring the representation power of the network is a normalization make most of the activated values fall into the linear region of the nonlinear funciton. However, this method equals to replace nonlinear function with linear function, so the representation power of network is decreased.

Down sampling is handled with strided convolution in the depthwise convolutions as well as in the first layer. A final average pooling reduces the spatial resolution to 1 before the fully connected layer.

Our model structure puts nearly all of the computation into dense 1×1 convolutions. This can be implemented with highly optimized general matrix multiply(GEMM) functions.

1.1.3 Width Multiplier: Thinner Models

Although the base MobileNet architecture is already small and low latency, many times a specific use case or application may require the model to be smaller and faster.

Width multiplier α the role of the width multiplier α is to thin a network **uniformly** at each layer. For a given layer and width multiplier α the number of input channels M becomes αM and the output channels N becomes αN

The computational cost of a depthwise separable convolution with width multiplier α is:

$$D_K \cdot D_K \cdot \alpha M \cdot D_F \cdot D_F + \alpha M \cdot \alpha N \cdot D_F \cdot D_F$$

1.2 ResNet18

1.2.1 Introduction

Driven by the significace of depth, a questiion arises: Is learning better networks as easy as stacking more layers?

An obstacle to answering this question was the notorious problem of vanishing/exploding gradients, which hamper convergence from the beginning. This problem, however, has been largely addressed by normalized initialization and itermediate normalizationlayers, which enable networks with tens of layers to start converging for stochastic gradient descent with back propagation.

When deeper networks are able to start converging, a degradation problem has been exposed: with the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly. Unexpectedly, such degradation is not caused by overfitting, and adding more layers to a suitably deep model leads to higher training error.

Let us consider a shallower architecture and its deeper counterpart that adds more layers onto it. There existence of this constructed solution indicates that a deeper model should produce no higher training error than its shallower counterpart.**But experiments show that our current solvers on hand are unable to find solutions that are comparably good or better than the constructed solution, or unable to do so in feasible time.**

In this paper, we address the degradation problem by introducing a deep residual learning framwork. Instead of hoping each few stacked layers directly **fit a desired underlying mapping**, we explicitly let these layers fit a residual mapping.

Formally, denoting the desired underlying mapping as $\mathcal{H}(x)$ we let the stacked nonlinear layers fit another mapping of

$$\mathcal{F}(x) := \mathcal{H}(x) - x$$

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.

The formulation of $\mathcal{F}(x) + x$ can be realized by feedforward neural network with "shortcut connections"(as being shown in figure 3)

1.2.2 Deep Residual Learning

If the added layers can be constructed as identity mappings, a deeper model should have training error no greater than its shallower counter-part. The degradation problem suggests that the solvers might have difficulties in approximating identity mappings by multiple nonlinear layers.

With the residual learning reformulation, if identity mappings are optimal, the solvers may simply drive the weights of the multiple nonlinear layers toward zero to approach identity mappings.

In real cases, it is unlikely that identity mapping are optimal, but our reformulation may help to precondition the problem. The experiments show that the learned residual functions in general have small responses, suggesting that identity mappings provide reasonable preconditioning.

Identity Mapping by shortcuts:

We consider a building block defined as:

$$y = \mathcal{F}(x, \{W_i\}) + x$$

where x and y are the input and output vectors of the layers considered. The function $\mathcal{F}(x, \{W_i\})$ represents the residual mapping to be learned. The operation $\mathcal{F} + x$ is performed by a shortcut connection and element-wise addition. The shortcut connections introduce neither extra parameter nor computation complexity. This is not only attractive in practice but also important in our comparisons between plain and residual networks. that simultaneously have the same number of parameters, depth, and computational cost.

The dimension of x and $mathcalF$ must be equal, in this is not the case(e.g., when changing the input/output channels), we can perform a **linear projection** W_s by the shortcut connections to match the dimensions:

$$y = \mathcal{F}(x, \{W_i\}) + W_s x$$

We can also use a square matrix W_s, but we will show by experiments that the identity mapping is sufficient for addressing the degradation problem and is economical and **thus W_s is only used when matching dimensions.** The function $\mathcal{F}(x, \{W_i\})$ can represent multiple convolutional layers. The element-wise addition is performed on two feature maps, channel by channel.

1.2.3 Network Architectures

We have tested various plain/residual nets, and have observed consistent phenomena.

Plain Network: inspired by the philosophy of VGG net. The convolutional layers mostly have 3×3filter and follow two simple design rules:

1. for the same output feature map size, the layers have the same number of filters;
2. if the feature map size is halved, the number of filters is doubled so as to preserve the time complexity per layer. We perform downsampling directly convolutional layers that have a stride of 2.

Residual Network: Based on above plain network, we insert shortcut connections as shown in Figure 4 right which turn the network into its counterpart residual version.

The identity can be directly used when the input and output are of the same dimensions (solid line shortcuts in Figure 4) When the dimension increase (dotted line shortcuts in Figure 4),we consider two options(This is very important and the second option was used in Orthant based's code)

1. The shortcut still performs identity mapping, with extra zero entries padded for increasing dimensions. This option introduces no extra parameter.
2. The projection shortcut in is used to match dimensions (done by 1×1 convolutions)

For both options, when the shortcuts go across feature maps of two sizes, they are performed with a stride of 2.

ImageNet Classification

Plain Network: We first evaluate 18-layer and 34-layer plain nets . The 34-layer plain net is in Figure 4 See Figure 5 for detailed architectures.

Residual Networks: Next we evaluate 18-layer and 34-layer residual nets(ResNets). The baseline architectures are the same as the above plain nets, expect that a shortcut connection is added to each pair of 3×3 filters as in Figure 4. We use identity mapping

CIFAR-10 and Analysis

The plain/residual hitectures follow the form in Figure 4 (middle/right). The network inputs are 32×32 images, with the per-pixel mean subtracted.

2 DC approximation approaches: consistency results

We focus on the sparse optimization problem with l_0-norm in the objective function, called the l_0-problem, that takes the form

$$min \{F(x, y) = f(x, y) + \lambda \|x\|_0 : (x, y) \in K\} \tag{1}$$

where λ is a positive parameter.K is a convex set in $\mathbb{R}^n \times \mathbb{R}^m$ and f is a finite DC function on $\mathbb{R}^n \times \mathbb{R}^m$. Suppose that f has a DC decomposition

$$f(x, y) = g(x, y) - h(x, y) \quad \forall (x, y) \in \mathbb{R}^n \times \mathbb{R}^m$$

where g, h are finite convex funtion on $\mathbb{R}^n \times \mathbb{R}^m$.

Define the step functions: $\mathbb{R} \to \mathbb{R}$ by $s(t) = 1$ for $t \neq 0$ and $s(t) = 0$

$$s(t) = \begin{cases} 1 & t \neq 0 \\ 0 & otherwise \end{cases}$$

Then $\|x\|_0 = \sum_{i=1}^n s(x_i)$, the idea is to replace the discontinuous step function by a continuous approximation r_θ, where $\theta > 0$ is a parameter controling the tightness of approximation.

Assumption:$\{r_\theta\}_{\theta>0}$ is a family of funciton $\mathbb{R} \to \mathbb{R}$ satifying the folling properities:

1. $\lim_{\theta \to +\infty} r_\theta(t) = s(t), \forall t \in \mathbb{R}$
2. $\forall \theta > 0$,r_θ is even, i.e.$r_\theta(|t|) = r_\theta(t)(\forall t \in \mathbb{R})$ and r_θ is a increasing on $[0, +\infty)$
3. $\forall \theta > 0$, r_θ is a DC function which can be represented as $r_\theta(t) = \varphi_\theta(t) - \psi_\theta(t), t \in \mathbb{R}$,where $\varphi_\theta, \psi_\theta$ are finite convex functions on $\mathbb{R}$
4. $t\mu \geq 0, \forall t \in \mathbb{R}, \mu \in \partial r_\theta(t)$, where $\partial r_\theta(t) = \{u - v : u \in \partial r_\theta(t), v \in \partial \psi_\theta(t)\}$
5. $\forall a \leq b$ and $0 \notin [a, b]$,then $\lim_{\theta \to \infty} sup \{|z| : z \in \partial r_\theta(t), t \in [a, b]\} = 0$

First of all, we observe that by assumption 2 above, we get another equivalent form of

$$\min_{(x,y,z)\in\Omega_1} \bar{F}_{r_\theta}(x,y,z) := f(x,y) + \lambda \sum_{i=1}^{n} r_\theta(z_i) \tag{2}$$

where $\Omega_1 = \{(x,y,z) : (x,y) \in K, |x_i| \le z_i, \forall i = 1,...,n\}$

Indeed,1 and 2 are equivalent in the following sense.

Proposition 2.1. *A point $(x^*, y^*) \in K$ is a global(resp.local) solution of the problem 1 if and only if $(x^*, y^*, |x^*|)$ is a global (resp.local) solution of the problem 2. Moreover , if (x^*, y^*, z^*) is a global solution of 2,then (x^*, y^*) is a global solution of 1*

Proof. There are many problem in the proof process. Since r_θ is an increasing function on $[0, +\infty)$, we have

$$\bar{F}_{r_\theta}(x,y,z) \ge \bar{F}_{r_\theta}(x,y,|x|) = F_{r_\theta}(x,y)$$

the inequality by $\Omega_1 = \{(x,y,z) : (x,y) \in K, |x_i| \le z_i\}$ □

Now we show the link between the original problem and the approximate problem. This result gives a mathmatical foundation of approximation methods.

Theorem 2.2. *Let $\mathcal{P}, \mathcal{P}_\theta$ be the solution sets of the problem 1 and 2 respectively.*

1. *Let $\{\theta_k\}$ be a sequence of nonnegative numbers such that $\theta_k \to +\infty$ and $\left\{(x^k, y^k)\right\}$ be a sequence such that $(x^k, y^k) \in \mathcal{P}_{\theta_k} \forall k$. If $(x^k, y^k) \to (x^*, y^*)$,then $(x^*, y^*) \in \mathcal{P}$*
2. *If K is compact, then $\forall \epsilon > 0$, there is $\theta(\epsilon) > 0$,such that $\mathcal{P}_\theta \subset \mathcal{P} + B(0,\epsilon), \forall \theta \ge \theta(\epsilon)$*
3. *If there is a finite set S such that $\mathcal{P}_\theta \bigcap S \ne 0 \forall \theta > 0$, then there exits $\theta_0 \ge 0$ such that*

$$\mathcal{P}_\theta \bigcap S \subset \mathcal{P} \quad \forall \theta \ge \theta_0$$

Proof. Let (x,y) be arbitrary in K. For any k, since $(x^k, y^k) \in \mathcal{P}_{\theta_k}$, we have

$$f(x,y) + \lambda \sum_{i=1}^{n} r_{\theta_k}(x_i) \ge f(x^k, y^k) + \lambda \sum_{i=1}^{n} r_{\theta_k}(x_i^k)$$

v because $\mathcal{P}_{\theta_k}$ is the solution sets of problem 2

By assumption 2: $\forall \theta > 0$,r_θ is even, i.e.$r_\theta(|t|) = r_\theta(t)(\forall t \in \mathbb{R})$ and r_θ is a increasing on $[0, +\infty)$

If $x_i^* = 0$, we have $\liminf_{k\to+\infty} r_{\theta_k}(x_i^k) \ge \liminf_{k\to+\infty} r_{\theta_k}(0) = 0$. (so the limit point (x^*, y^*) is minimum)

If $x_i^* \ne 0$, there exists $a_i \le b_i$ and $k_i \in \mathbb{N}$ such that $0 \notin [a_i, b_i]$ and $x_i^k \in [a_i, b_i]$ for all $k \ge k_i$. Then we have

$$\left|r_{\theta_k}(x_i^k) - s(x_i^*)\right| \le max \left\{\left|r_{\theta_k}(a_i) - s(a_i)\right|, \left|r_{\theta_k}(b_i) - s(b_i)\right|\right\} \forall k \ge k_i$$

(concave funciton substract 1, and get its absolute value, so need to get maximum)

Since $\lim_{k\to\infty} r_{\theta_k}(a_i) = s(a_i)$ and $\lim_{k\to\infty} r_{\theta_k}(b_i) = s(b_i)$,we have $\lim_{k\to\infty} r_{\theta_k}(x_i^k) = s(x_i^*)$

Note that f is continuous, takeing liminf of both side of $f(x,y) + \lambda \sum_{i=1}^{n} r_{\theta_k}(x_i) \ge f(x^k, y^k) + \lambda \sum_{i=1}^{n} r_{\theta_k}(x_i^k)$ we have

$$f(x,y) + \lambda \sum_{i=1}^{n} s(x_i) \ge f(x^*, y^*) + \liminf_{k\to\infty} r_{\theta_k}(x_i^k) = f(x^*, y^*) + \lambda \sum_{i=1}^{n} s(x_i^*)$$

Thus, $F(x,y) \geq F(x^*,y^*)\forall(x,y) \in K$ or $(x^*,y^*) \in \mathcal{P}$ □

Proof. Consider its contradiction is $\exists\epsilon > 0, \forall\theta(\epsilon) > 0, \exists\theta > \theta(\epsilon)$ such that $\mathcal{P}_\theta \nsubseteq \mathcal{P} + B(0,\epsilon)$

so there $\exists\epsilon$ without loss of generality,$\forall\theta_1(\epsilon) > 0$ $\exists\theta_2(\epsilon) > \theta_1(\epsilon)$ such that $P_{\theta_2} \nsubseteq P + B(0,\epsilon)$ .let θ_2 replace θ_1, then we can get θ_3 meet the above conditions. So we can get a subsequence converge in out of $P + B(0,\epsilon)$, but this is an infinity sequence, so its limit point is should converges to compact set K(P_{θ_k}) which is a contradiction. □

Proof. Consider its contradiction is $\forall\theta_0$, there $\exists\theta > \theta_0$ such that $P_\theta \bigcap S \not\subset P$.

Next let θ_1 replace θ, so it can generate a infinity sequence by this method, so there exists $(x_i, y_i) \in P_{\theta_i} \bigcap S$ and $\notin P$ but $\left\{(x_i, y_i)_{i=1,2,3,\ldots}\right\} \subseteq S$,

because $\{(x_i, y_i)\}$ is a infinity sequence and S is a finite set, so the sequence would converge to a point $(\bar{x}, \bar{y})$, by 1 ,this point should $\notin P$ which contradict to the $(x^*, y^*) \in P$. □

3 Programming Syntax

Now we introduce the Syntax by Adam algorithm

Algorithm 1 Outline of Adam

```
1: procedure ADAM($\alpha, \beta_1, \beta_2$)
   Input: $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$ where $\beta_1, \beta_2$ are exponential decay rates for the moment estimates
2:     Initialize: $\theta_0, m_0, v_0, t$
   where $\theta_0$ is parameter vector; $m_0$ 1-st moment vector; $v_0$ 2-nd moment vector; t is timestep.
3:     while $\theta_t$ not converged do
4:         $t \leftarrow t + 1$
5:         $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
6:         $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
7:         $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t \bigodot g_t$
8:         $\hat{m}_t \leftarrow m_t/(1 - \beta_1^t)$(Compute bias-corrected first moment estimate, t power)
9:         $\hat{v}_t \leftarrow v_t/(1 - \beta_2^t)$(Compute bias-corrected second raw moment estimate, t power)
10:        $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)$(Update parameters)
11:    return $\theta_t$(Resulting parameters)
```

Some basic starters

The **params** are model's parameters, and the parameter **defaults** need to update in the algorithm.

3.1 About initializing parameter

```
def __init__(self,params,lr,betas = (0.9,0.99)):
   if not 0.0 <= lr:
      raise ValueError("Invalid learning")
   defaults = dict(lr = lr,betas = betas)
   super(algorithm_name,self).__init__(params,defaults)
```

Key is step function, and it need to be decorated by **torch.no_grad()**

```
@torch.no_grad()
def step(self, closure = None):
   '''
   closure(callable,optional): A closure that reevaluates the model and return the loss
   '''
   loss = None #first put loss to None, and it will recalculate.
   if closure is not None:
      with torch.enable_grad():
         loss = closure()
```

Some basic formulas. Note that there is "is None"

```
@torch.no_grad()
def step(self, closure = None):
   '''
   closure(callable,optional): A closure that reevaluates the model and return the loss
   '''
   loss = None #first put loss to None, and it will recalculate.
   if closure is not None:
      with torch.enable_grad():
         loss = closure()
   for group in self.param_groups:
      '''
      There are some hyperparameters you can get here which is from outside.
      '''
      lr = group['lr']
      beta1,beta2 = group['betas']
      for p in group['params']:
         '''Next is about how to change p which is model's parameter'''
         if p.grad is None:
            continue
```

And we will find the iteration will using the information from the last iteration. So you need to use container to hold it, like **state**

```
state = self.state[p]

#State initialization
if len(state) == 0:
   state['step'] = 0
   #Exponential moving average of gradient values,which is $m_t$ in the formula
   state['exp_avg'] = torch.zeros_like(p,memory_format = torch.preserve_format)
   #Exponential moving average of squared gradient values, which is $v_t$ in the formula
   state['exp_avg_sq'] = torch.zeros_like(p,memory_format = torch.preserve_format)
```

```
import math
import torch
from .optimizer import Optimizer
class adam(Optimizer):
    def __init__(self,params,lr = 1e-3, betas = (0.9,0.999),eps = 1e-8):
        if not 0.0 <= lr:
            raise ValueError('Invalid learning rate: {}'.format(lr))
        if not 0.0 <= eps:
            raise ValueError('Invalid epsilon value: {}'.format(eps))
        if not 0.0 <= betas[0]<1.0:
            raise ValueError('Invalid beta parameter at index 0: {}'.format(betas[0]))
        if not 0.0 <= betas[1]<1.0:
            raise ValueError('Invalid beta parameter at index 0: {}'.format(betas[1]))
        defaults = dict(lr = lr,betas = betas,eps = eps)
        super(adam,self).__init__(params, defaults)

    @torch.no_grad()
    def step(self,closure = None):
        loss = None
        if closure is not None:
            with torch.enable_grad():
                loss = closure()

        for group in self.param_groups:
            lr = group['lr']
            beta1,beta2 = group['betas']
            eps = group['eps']
            for p in group['params']:
                if p.grad is None:
                    continue
                grad = p.grad

                state = self.state[p]
                if len(state) == 0:
                    state['step'] = 0
                    state['m'] = torch.zeros_like(p,memory_format=torch.preserve_format)
                    state['v'] = torch.zeros_like(p.memory_format=torch.preserve_format)

                m = state['m']
                v = state['v']
                # t+1
                state['step'] += 1
                m.mul_(beta1).add_(grad,alpha = 1-beta1)
                # addcmul performs the element-wise multiplication of tensor1 by tensor2,
                    multiply the result by the scalar value and add it to input.
                # out_i = input_i + value*tensor1_i*tensor2_i
                v.mul_(beta2).addcmul_(grad,grad,value = 1-beta2)

                bias_correction1 = 1- beta1**state['step']
                bias_correction2 = 1- beta2**state['step']

                hat_m = torch.div(m,bias_correction1)
                hat_v = torch.div(v,bias_correction2)

                denom = hat_v.sqrt().add_(eps)
                p.addcdiv(hat_m,denom,value = -lr)
        return loss
```

And you will find the code is different from official code, maybe official code is optimized in memory resource **which is needed to confirmed with the teacher.**

Next I consider a algorithm from the paper A proximal difference of convex algorithm with extrapolation and the penalty function is $r_{exp} = 1 - \exp\{-\theta\,|t|\}$ and $P_1 = \theta\,|t|$, $P_2 = \theta\,|t| + exp\{-\theta\,|t|\} - 1$

Algorithm 2 Proximal difference of convex algorithm with extrapolation ($pDCA_e$) for r_{exp}

```
1: procedure $pDCA_e$(L = lr)
   Input: $\theta$(which is to the degree of controling approximation)
   L(lr:learning rate),
   $x_0 \in domP_1$
   $\{\beta_t\} \subset [0, 1)$ with $\sup_t \beta_t < 1$,
   set $x^{-1} = x^0$
2:     Initialize:$k_{-1} = k_0 = 1$
3:     while $x_t$ is not convergence do
4:         $\xi^t = sign(x^t)\theta(1 - exp\left\{-\theta \left|x^t\right|\right\})$
5:         $\beta_t = (k_{t-1} - 1)/k_t$
6:         $k_{t+1} = (1 + \sqrt{1 + 4k_t^2})/2$
7:         $y_t = x_t + \beta_t(x_t - x_{t-1})$
8:
           $$x_{t+1} = \begin{cases} -\frac{1}{L}(\nabla f(y_t) - \xi^t + \theta) + y_t & , y_t > \frac{1}{L}(\nabla f(y^t) - \xi^t + \theta) \\ 0 & , \text{otherwise} \\ -\frac{1}{L}(\nabla f(y_t) - \xi^t - \theta) + y_t & , y_t < \frac{1}{L}(\nabla f(y_t) - \xi^t - \theta) \end{cases}$$
9:         $t \leftarrow t + 1$
```

Figure 1: MobileNet Body Architecture

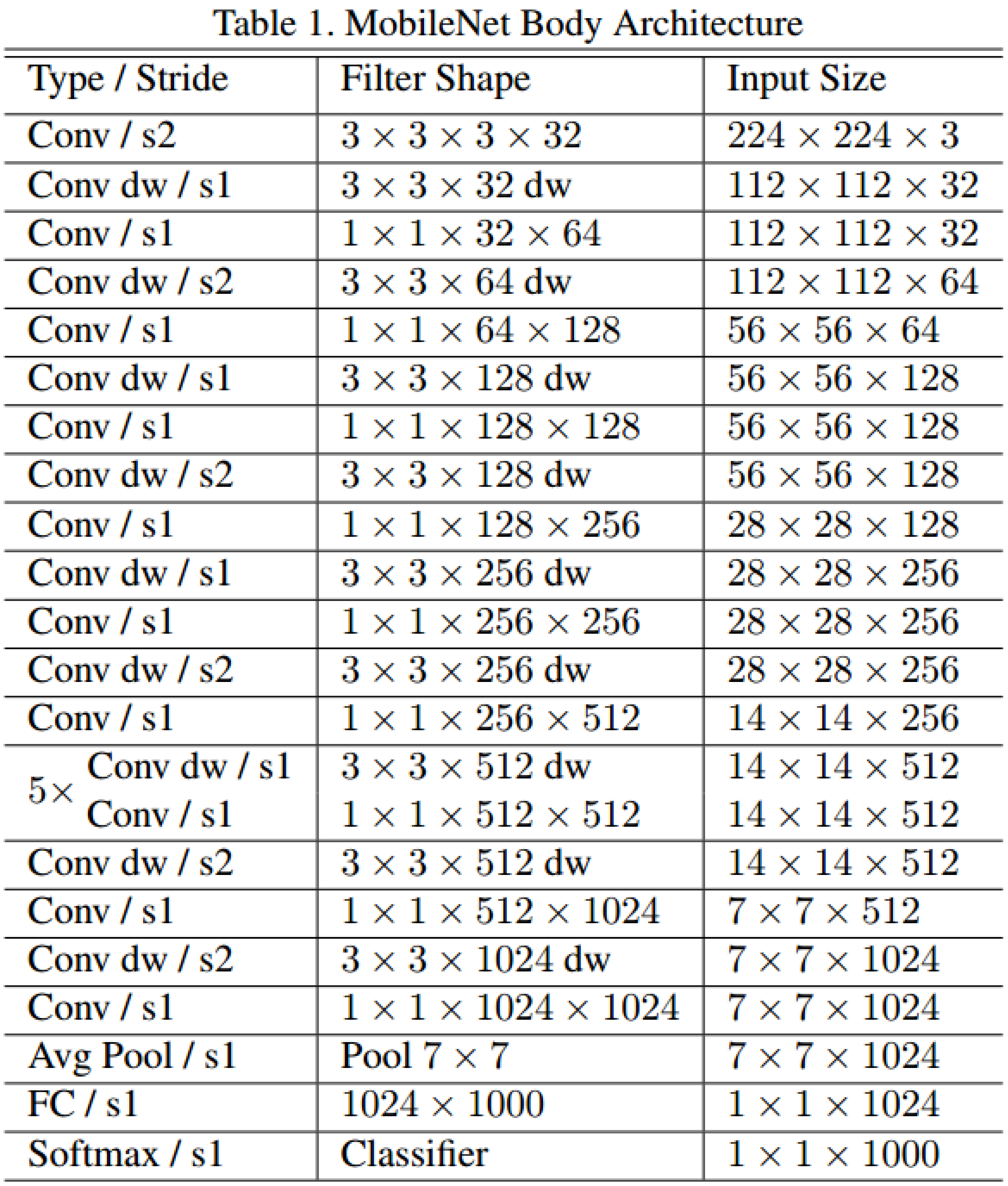


Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Figure 2: comparison of A and B

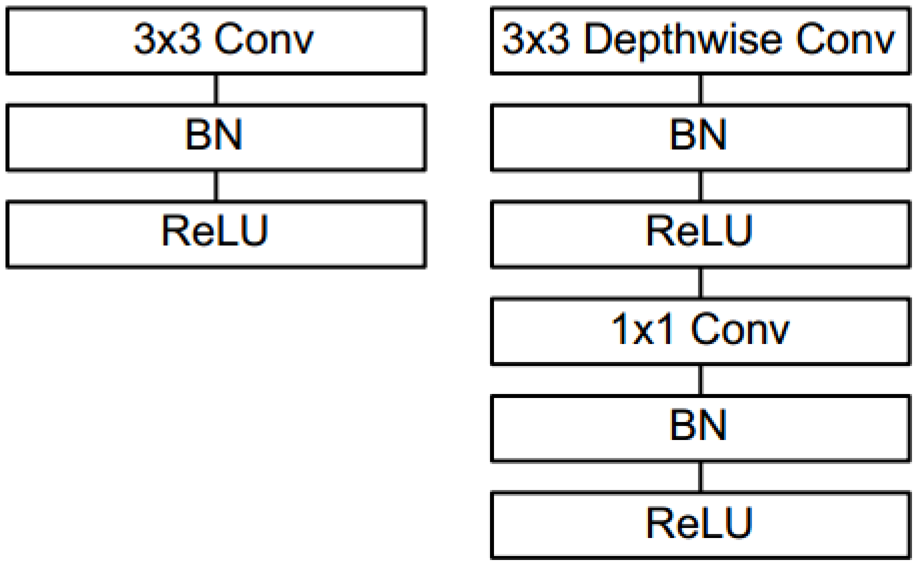


Figure 3. Left: Standard convolutional layer with batchnorm and ReLU. Right: Depthwise Separable convolutions with Depthwise and Pointwise layers followed by batchnorm and ReLU.

Figure 3: Residual learning: a building block

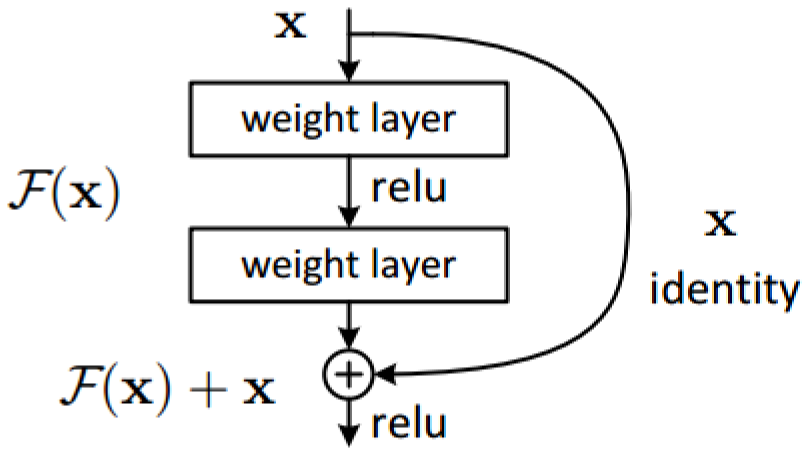


Figure 2. Residual learning: a building block.

Figure 4: example network architectures for ImageNet

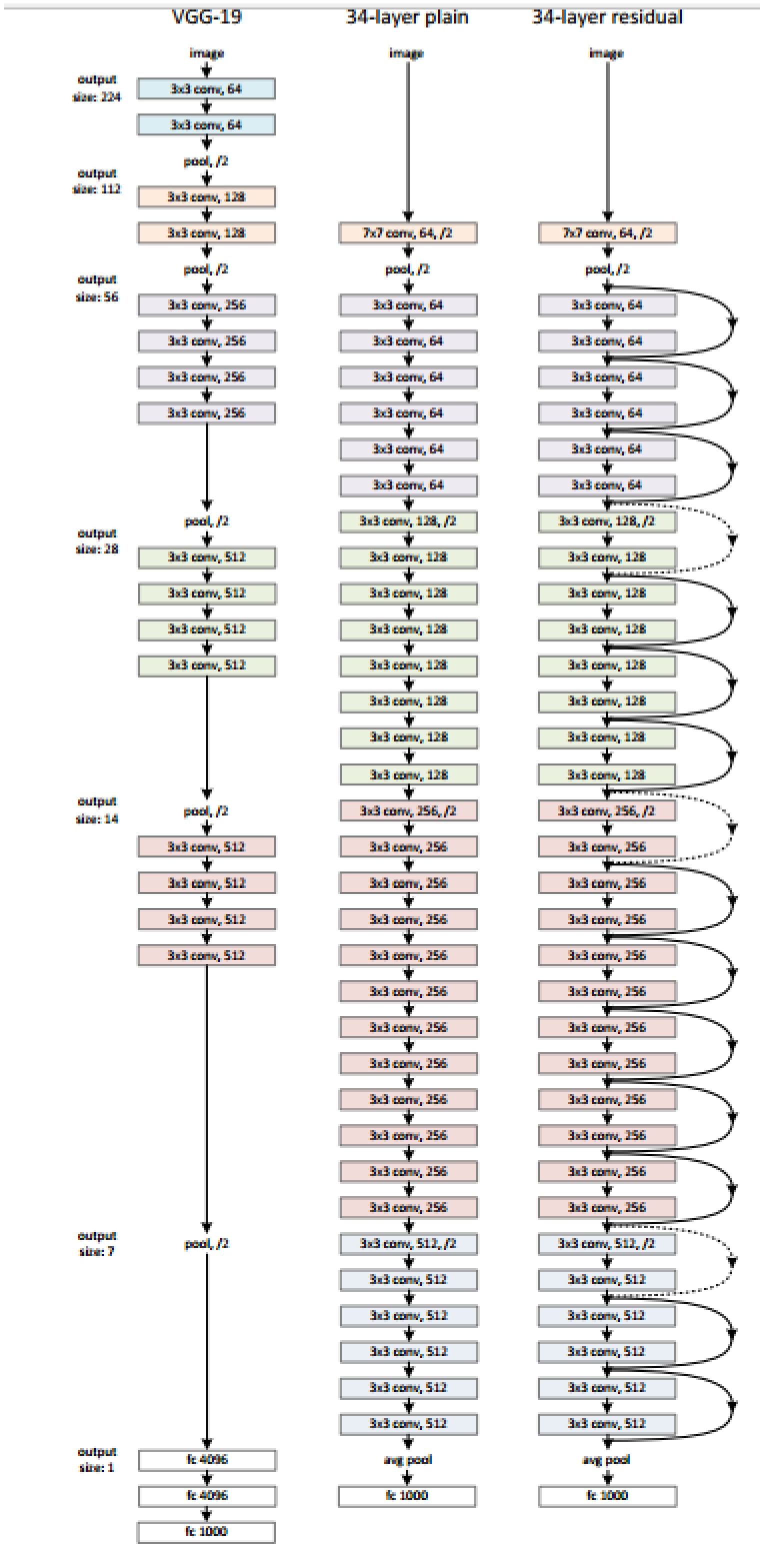
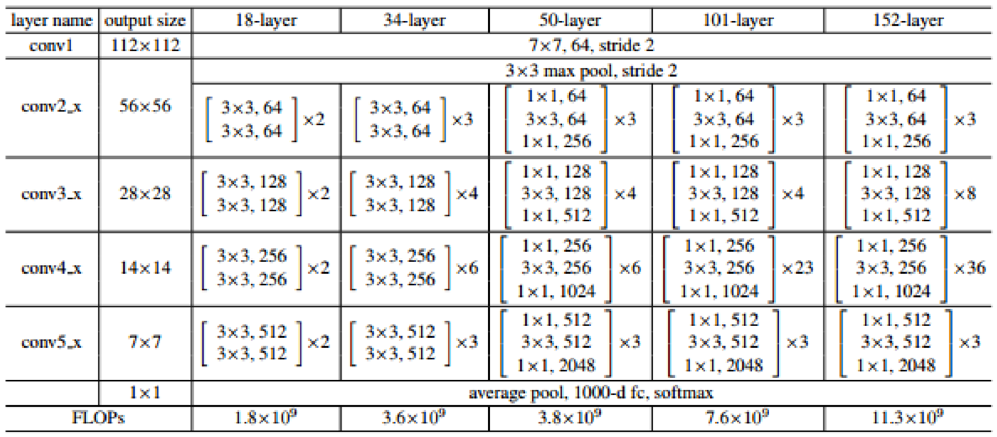


Figure 3. Example network architectures for ImageNet. **Left**: the VGG-19 model [40] (19.6 billion FLOPs) as a reference. **Middle**: a plain network with 34 parameter layers (3.6 billion FLOPs). **Right**: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. **Table 1** shows more details and other variants.

Figure 5: plain network detailed architectures



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

Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Down-sampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.