Bài 48 - Mobilenet model

19 Sep 2020 - phamdinhkhanh

Menu

1. Nhu cầu về lightweight model

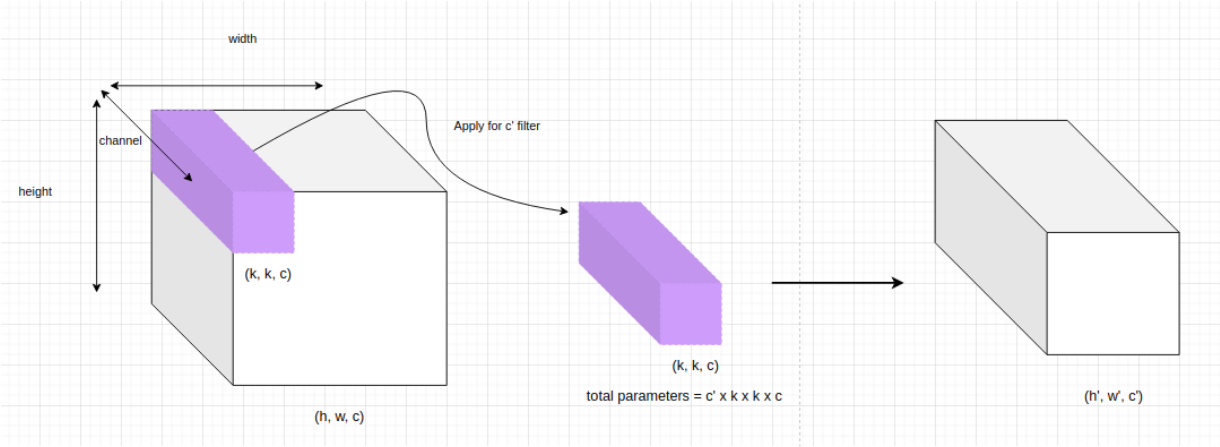
Sự phát triển về mặt học thuật của thị giác máy tính đã tạo ra rất nhiều các models khác nhau với độ chính xác được đo lường trên bộ dữ liệu ImageNet ngày càng được cải thiện. Tuy nhiên không phải toàn bộ trong số chúng đều có thể sử dụng được trên các thiết bị gặp hạn chế về tài nguyên tính toán. Để phát triển được những ứng dụng AI trên các thiết bị như mobile, IoT thì chúng ta cần hiểu về tài nguyên của những thiết bị này để lựa chọn model phù hợp cho chúng. Những mô hình ưa chuộng được sử dụng thường là những model có số lượng tính toán ít và độ chính xác cao. MobileNet là một trong những lớp mô hình như vậy và trong bài viết này chúng ta cùng tìm hiểu về kiến trúc đặc biệt ẩn đằng sau họ các lớp mô hình MobileNet.

2. Các dạng tích chập.

2.1. Tích chập 2 chiều thông thường.

Như chúng ta đã biết tích chập 2 chiều thông thường sẽ được tính toán trên toàn bộ chiều sâu (channel). Do đó số lượng tham số của mô hình sẽ gia tăng đáng kể phụ thuộc vào độ sâu của layer trước đó.

Top



Để thống nhất, ký hiệu shape của các tensor3D theo thứ tự các chiều (height, width, channel).

Chẳng hạn ở trên chúng ta có một đầu vào kích thước $h \times w \times c$, tích chập thông thường sẽ cần $k \times k \times c$ tham số để thực hiện tích chập trên toàn bộ độ sâu của layers. Mỗi một bộ lọc sẽ tạo ra một ma trận output kích thước $h' \times w' \times 1$. Áp dụng c' bộ lọc khác nhau ta sẽ tạo ra đầu ra có kích thước $h' \times w' \times c'$ (các ma trận output khi áp dụng tích chập trên từng bộ lọc sẽ được concatenate theo chiều sâu). Khi đó số lượng tham số cần sử dụng cho một tích chập thông thường sẽ là: $c' \times k \times k \times c$.

Số lượng tham số sẽ không đáng kể nếu độ sâu của input channel c là nhỏ, thường là ở các layers ở vị trí đầu tiên. Tuy nhiên khi độ sâu tăng tiến dần về những layers cuối cùng của mạng CNN thì số lượng tham số của mô hình sẽ là một con số không hề nhỏ. Sự gia tăng tham số này tạo ra những mô hình cồng kềnh làm chiếm dụng bộ nhớ và ảnh hưởng tới tốc độ tính toán. Alexnet và VGGNet là những mô hình điển hình có số lượng tham số rất lớn do chỉ áp dụng những tích chập 2 chiều thông thường.

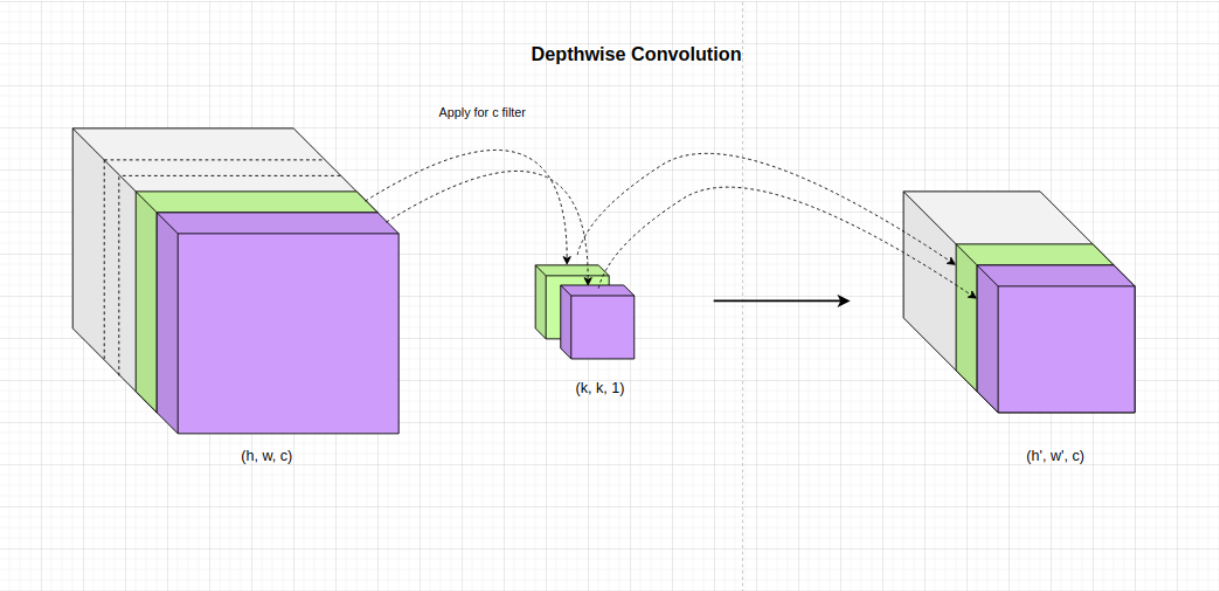
Nếu bạn theo dõi ImageNet Leader Board (https://paperswithcode.com/sota/image-classification-on-imagenet) sẽ thấy rằng những mô hình thuộc họ MobileNet chỉ có vài triệu tham số nhưng độ chính xác tốt hơn AlexNet vài chục triệu tham số. Điểm mấu chốt tạo nên sự khác biệt này đó là MobileNet lần đầu tiên áp dụng kiến trúc tích chập tách biệt chiều sâu. Chúng ta sẽ cùng tìm hiểu về kiến trúc này ở phần tiếp theo.

2.2. Tích chập tách biệt chiều sâu (Depthwise Separable Convolution).

Chúng ta nhận định rằng độ sâu là một trong những nguyên nhân chính dẫn tới sự gia tăng số lượng tham số của mô hình. Tích chập tách biệt chiều sâu sẽ tìm cách loại bỏ sự phụ thuộc vào độ sâu khi tích chập mà vẫn tạo ra được một output shape có kích thước tương đương so với tích chập thông thường. Cụ thể quá trình sẽ được chia thành hai bước tuần tự:

- **Tích chập chiều sâu (Depthwise Convolution)**: Chúng ta sẽ chia khối input tensor3D thành những lát cắt ma trận theo độ sâu. Thực hiện tích chập trên từng lát cắt như hình minh họa bên dưới.

Top

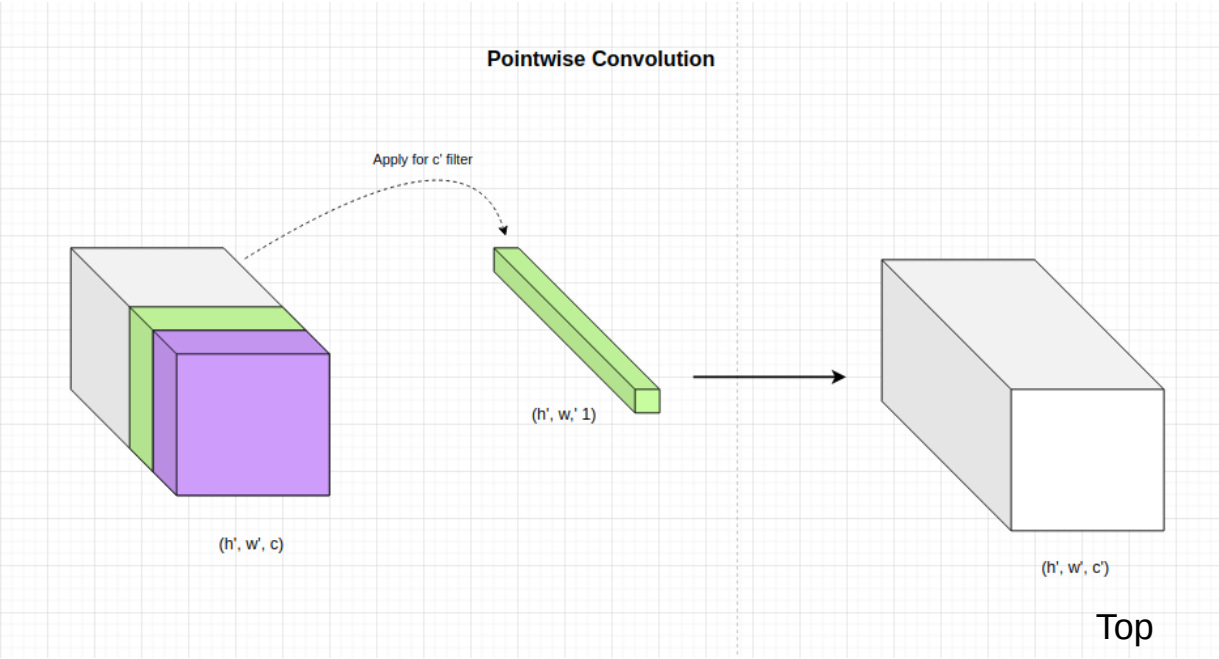


Mỗi một channel sẽ áp dụng một bộ lọc khác nhau và hoàn toàn không chia sẻ tham số. Điều này có ba tác dụng chính cho mô hình:

- Nhận diện đặc trưng: Quá trình học và nhận diện đặc trưng sẽ được tách biệt theo từng bộ lọc. Nếu đặc trưng trên các channels là khác xa nhau thì sử dụng các bộ lọc riêng cho channel sẽ chuyên biệt hơn trong việc phát hiện các đặc trưng. Chẳng hạn như đầu vào là ba kênh RGB thì mỗi kênh áp dụng một bộ lọc khác nhau chuyên biệt.
- Giảm thiểu khối lượng tính toán: Để tạo ra một điểm pixel trên output thì tích chập thông thường cần sử dụng $k \times k \times c$ phép tính trong khi tích chập chiều sâu tách biệt chỉ cần $k \times k$ phép tính.
- Giảm thiểu số lượng tham số : Ở tích chập chiều sâu cần sử dụng $c \times k \times k$ tham số. Số lượng này ít hơn gấp c' lần so với tích chập chiều sâu thông thường.

Kết quả sau tích chập được concatenate lại theo độ sâu. Như vậy output thu được là một khối tensor3D có kích thước $h' \times w' \times c$.

- **Tích chập điểm (Pointwise Convolution)**: Có tác dụng thay đổi độ sâu của output bước trên từ c sang c'. Chúng ta sẽ áp dụng c' bộ lọc kích thước $1 \times 1 \times c$. Như vậy kích thước width và height không thay đổi mà chỉ độ sâu thay đổi.



Top

Kết quả sau cùng chúng ta thu được là một output có kích thước $h' \times w' \times c'$. Số lượng tham số cần áp dụng ở trường hợp này là $c' \times c$.

2.3. So sánh tích chập tách biệt chiều sâu và tích chập thông thường

2.3.1. Số lượng tham số

Như chúng ta đã biết, để tạo thành một shape có kích thước $h' \times w' \times c'$ thì số lượng tham số sử dụng ở tích chập thông thường là: $c' \times k \times k \times c$ Trong khi đó số lượng tham số sử dụng ở *tích chập chiều tách biệt chiều sâu*: $k \times k \times c + c' \times c$ Thông thường c' sẽ lớn hơn c vì càng ở các layer sau thì độ sâu càng lớn. Do đó tỷ lệ: $\frac{c' \times k \times k \times c}{k \times k \times c + c' \times c} = \frac{c' \times k \times k}{k \times k + c'}$ Tỷ lệ này sẽ gần bằng $k \times k$ nếu $c' \gg k \times k$. Đây là một mức giảm khá đáng kể về kích thước mô hình. Bạn đọc đã hiểu lý do vì sao mà MobileNet lại có kích thước nhỏ gấp vài chục lần so với Alexnet rồi chứ ?

2.3.2. Số lượng phép toán cần thực hiện

Để cùng tạo ra một output shape có kích thước $h' \times w' \times c'$ thì tích chập thông thường cần thực hiện: $(h' \times w' \times c') \times (k \times k \times c)$ Trong đó $h' \times w' \times c'$ là số lượng pixels cần tính và $k \times k \times c$ là số phép nhân để tạo ra một pixel.

Tích chập tách biệt chiều sâu chỉ phải thực hiện lần lượt trên:

- Tích chập chiều sâu: $(h' \times w' \times c) \times (k \times k)$ phép nhân.
- Tích chập điểm: $(h' \times w' \times c) \times (h' \times w')$ phép nhân. Tỷ lệ các phép tính giữa tích chập thông thường và tích chập chiều sâu:

$$\frac{h' \times w' \times c' \times k \times k \times c}{h' \times w' \times c \times k \times k + h' \times w' \times c \times h' \times w'} = \frac{c' \times k \times k}{k \times k + h' \times w'}$$

Đây là một tỷ lệ khá lớn cho thấy tích chập chiều sâu tách biệt có chi phí tính toán thấp hơn rất nhiều so với tích chập thông thường. Do đó nó phù hợp để áp dụng trên các thiết bị cấu hình yếu.

2.4. Ví dụ về tích chập chiều sâu.

Bên dưới chúng ta sẽ lấy ví dụ về phương pháp sử dụng tích chập chiều sâu trên cả pytorch và tensorflow.

tensorflow: Trên tensorflow chúng ta có thể thực hiện tích chập chiều sâu tách biệt theo hai bước tuần tự là:

- *tích chập chiều sâu*: Sử dụng hàm `tf.keras.layers.DepthwiseConv2D()` với *depth_multiplier = 1* đại diện cho độ sâu được tích chập.
- *tích chập điểm*: Sử dụng hàm `tf.keras.layers.Conv2D()` với kích thước *kernel_size = 1*.

Top

```
import tensorflow as tf
import numpy as np
x = np.random.randn(10, 15, 15, 20) # Batch_size, W, H, C
x = x.astype('float32')
def depthwise_separable_conv_tf(x):
  dw2d = tf.keras.layers.DepthwiseConv2D(
      kernel_size=3, strides=1, padding='same', depth_multiplier=1
  )
  pw2d = tf.keras.layers.Conv2D(
      filters=50, kernel_size=1, strides=1
  )
  y = dw2d(x)
  y = pw2d(y)
  return y
y = depthwise_separable_conv_tf(x)
y.shape # Batch_size, W, H, C
```

```
TensorShape([10, 15, 15, 50])
```

pytorch:

Một ưu điểm mà mình nghĩ pytorch đang tốt hơn so với tensorflow đó là thiết kế của tích chập 2 chiều là module `torch.nn.Conv2d()` cho phép cấu hình độ sâu của tích chập thông qua tham số *groups*. Kích thước của độ sâu sẽ bằng *input channels/groups*. Mặc định của tham số này là *groups = 1*. Khi đó toàn bộ độ sâu sẽ là một nhóm và tích chập được tính trên toàn bộ độ sâu. Nếu muốn sử dụng tích chập theo chiều sâu thì thiết lập *groups = input channels*. Khi đó độ sâu sẽ được chia thành *input channels* nhóm, mỗi nhóm có kích thước là 1.

Bạn đọc có thể tham khảo thêm tại torch.nn.Conv2D() (https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html).

Bên dưới là code:

```
from torch import nn

class DepthwiseSeparableConv(nn.Module):
    def __init__(self, nin, nout):
        super(Depthwise_separable_conv, self).__init__()
        self.depthwise = nn.Conv2d(nin, nin, kernel_size=3, padding=1
        self.pointwise = nn.Conv2d(nin, nout, kernel_size=1)

    def forward(self, x):
        out = self.depthwise(x)
        out = self.pointwise(out)
        return out


x = torch.randn(10, 20, 15, 15) # Batch_size, C, W, H
y = DepthwiseSeparableConv(nin=20, nout=50)(x)
y.size()
```

```
torch.Size([10, 50, 15, 15])
```

Top

3. MobileNetV2

Kể từ khi ra đời, MobileNetV2 là một trong những kiến trúc được ưa chuộng nhất khi phát triển các ứng dụng AI trong computer vision. Rất nhiều các kiến trúc sử dụng backbone là MobileNetV2 như SSDLite (https://phamdinhkhanh.github.io/2019/10/05/SSDModelObjectDetection.html) trong object detection và DeepLabV3 (https://phamdinhkhanh.github.io/2020/06/18/DeepLab.html) trong image segmentation.

MobileNetV2 có một số điểm cải tiến so với MobileNetV1 giúp cho nó có độ chính xác cao hơn, số lượng tham số và số lượng các phép tính ít hơn mà chúng ta sẽ tìm hiểu ngay sau đây.

3.1. Inverted Residual Block

MobileNetV2 cũng sử dụng những kết nối tắt như ở mạng ResNet (https://phamdinhkhanh.github.io/2020/05/31/CNNHistory.html#46-resnet-50-2015). Các khối ở layer trước được cộng trực tiếp vào layer liền sau. Nếu coi layer liền trước là $\mathbf{x}$, sau khi đi qua các xử lý tích chập hai chiều ta thu được kết quả $\mathcal{F}(\mathbf{x})$ thì output cuối cùng là một residual block có giá trị $\mathbf{x} + \mathcal{F}(x)$.

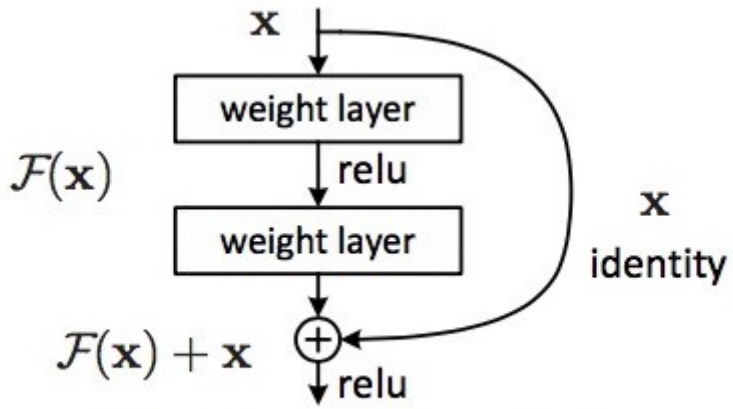


Figure 2. Residual learning: a building block.

Tuy nhiên kết nối tắt ở MobileNetV2 được điều chỉnh sao cho số kênh (hoặc chiều sâu) ở input và output của mỗi block residual được thắt hẹp lại. Chính vì thế nó được gọi là các bottleneck layers (bottleneck là một thuật ngữ thường được sử dụng trong deep learning để ám chỉ các kiến trúc thu hẹp kích thước theo một chiều nào đó).

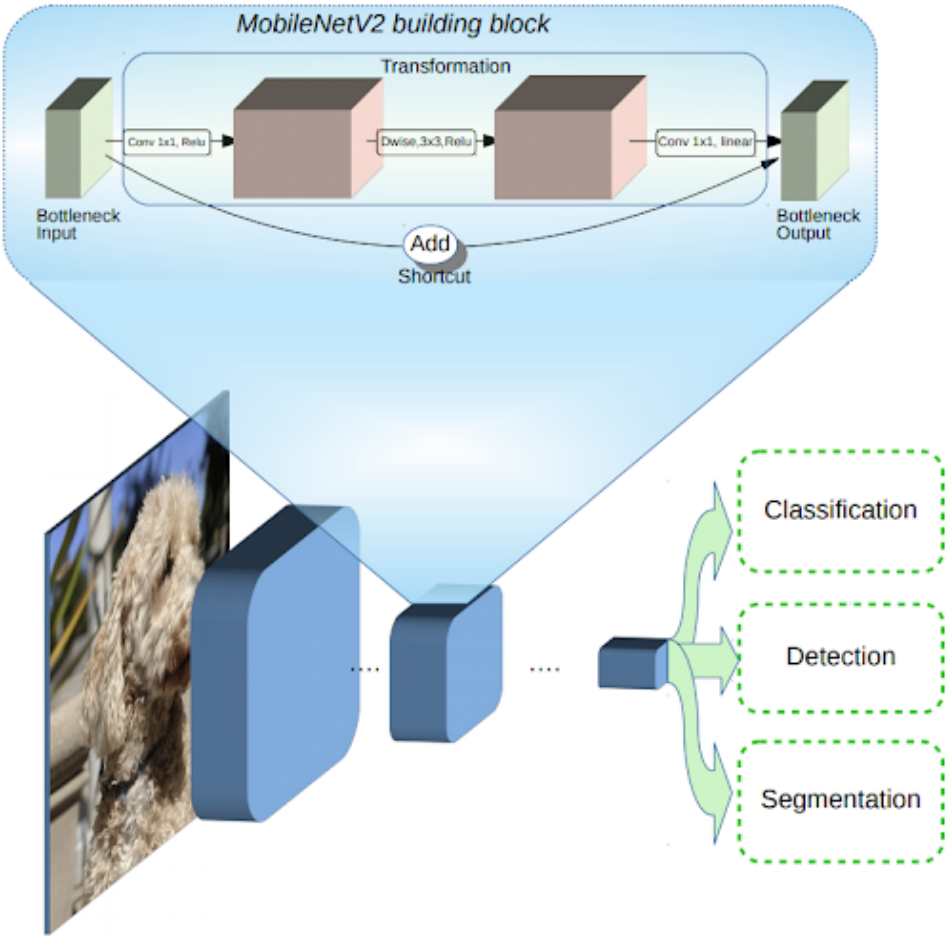
Xin trích dẫn:

```
The MobileNetV2 architecture is based on an inverted residual structure where
the input and output of the residual block are thin bottleneck layers opposite
to traditional residual models which use expanded representations in the input
an MobileNetV2 uses lightweight depthwise convolutions to filter features in the
intermediate expansion layer.
```

Source MobileNetV2 - Mark Sandler, etc (https://arxiv.org/abs/1801.04381).

Cụ thể hơn chúng ta theo dõi hình minh họa bên dưới:

Top



Source: AI google blog (https://ai.googleblog.com/2018/04/mobilenetv2-next-generation-of-on.html)

Kiến trúc residual này ngược lại so với các kiến trúc residual truyền thống vì kiến trúc residual truyền thống có số lượng kênh ở input và output của một block lớn hơn so với các layer trung gian. Chính vì vậy nó còn được gọi là kiến trúc *inverted residual block*.

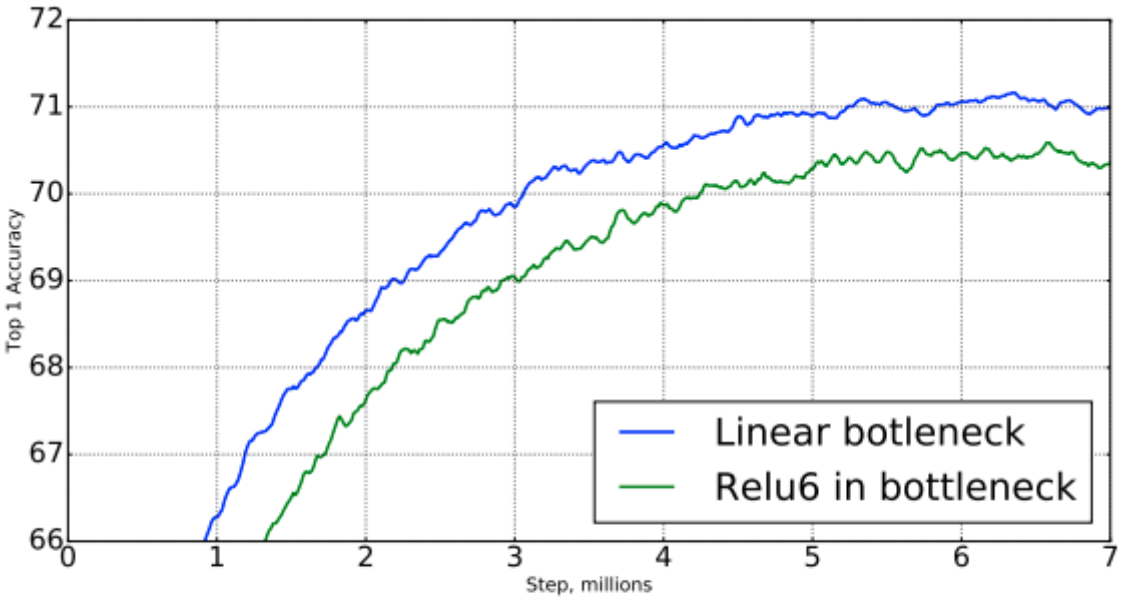
Tác giả cho rằng các layer trung gian trong một block sẽ làm nhiệm vụ biến đổi phi tuyến nên cần dày hơn để tạo ra nhiều phép biến đổi hơn. Kết nối tắt giữa các block được thực hiện trên những bottleneck input và output chứ không thực hiện trên các layer trung gian. Do đó các layer bottleneck input và output chỉ cần ghi nhận kết quả và không cần thực hiện biến đổi phi tuyến.

Ở giữa các layer trong một block *inverted residual block* chúng ta cũng sử dụng những biến đổi *tích chập tách biệt chiều sâu* để giảm thiểu số lượng tham số của mô hình. Đây cũng chính là bí quyết giúp họ các model MobileNet có kích thước giảm nhẹ.

3.2. Loại bỏ non-linear

Một trong những thực nghiệm được tác giả ghi nhận đó là việc sử dụng các biến đổi phi tuyến (như biến đổi qua ReLu hoặc sigmoid) tại input và output của các *residual block* sẽ làm cho thông tin bị mất mát. Cụ thể cùng xem kết quả thực nghiệm bên dưới:

Top



(a) Impact of non-linearity in the bottleneck layer.

Source MobileNetV2 - Figure 6a (https://arxiv.org/pdf/1801.04381.pdf)

Chính vì thế trong kiến trúc của *residual block* tác giả đã loại bỏ hàm phi tuyến tại layer input và output và thay bằng các phép chiếu tuyến tính.

3.3. Khởi tạo Inverted Residual Block

Tensorflow: Chúng ta có thể thực hiện khởi tạo *Inverted Residual Block* trên tensorflow như sau :

Top

```
import tensorflow as tf
import numpy as np

x = np.random.randn(10, 64, 64, 16)
x = x.astype('float32')

def inverted_linear_residual_block(x, expand=64, squeeze=16):
  '''
  expand: số lượng channel của layer trung gian
  squeeze: số lượng channel của layer bottleneck input và output
  '''
  # Depthwise convolution
  m = tf.keras.layers.Conv2D(expand, (1,1), padding='SAME', activatio
  m = tf.keras.layers.DepthwiseConv2D((3,3), padding='SAME', activati
  # Pointwise convolution + Linear projection
  m = tf.keras.layers.Conv2D(squeeze, (1,1), padding='SAME', activati
  opt = tf.keras.layers.Add()([m, x])
  return opt

y = inverted_linear_residual_block(x, expand=64, squeeze=16)
y.shape
```

```
TensorShape([10, 64, 64, 16])
```

Trong kiến trúc này chúng ta sử dụng lần lượt:

- Tích chập 2 chiều với kernel=(1, 1), số kênh của output được mở rộng để tạo thành layer trung gian.
- Tích chập chiều sâu (`DepthwiseConv2D`) với kernel=(3, 3) có tác dụng trích lọc đặc trưng. Lưu ý ở `DepthwiseConv2D` chúng ta sẽ không khai báo số kênh vì tích chập chiều sâu trả ra số kênh bằng với input trước đó.
- Tích chập 2 chiều với kernel=(1, 1) có số kênh giảm xuống để tạo thành output cho block residual.

Pytorch: Code trên pytorch như sau.

Top

```
import torch
from torch import nn

class InvertedLinearResidualBlock(nn.Module):
    def __init__(self, expand=64, squeeze=16):
        '''
        expand: số lượng channel của layer trung gian
        squeeze: số lượng channel của layer bottleneck input và outpu
        '''
        super(InvertedLinearResidualBlock, self).__init__()
        self.conv = nn.Sequential(
            # Depthwise Convolution
            nn.Conv2d(squeeze, expand, kernel_size=3, stride=1, paddi
            nn.ReLU6(inplace=True),
            # Pointwise Convolution + Linear projection
            nn.Conv2d(expand, squeeze, kernel_size=1, stride=1, paddi
        )

    def forward(self, x):
        return x + self.conv(x)

x = torch.randn(10, 16, 64, 64) # Batch_size, C, W, H
y = InvertedLinearResidualBlock(expand=64, squeeze=16)(x)
y.size()
```

```
torch.Size([10, 16, 64, 64])
```

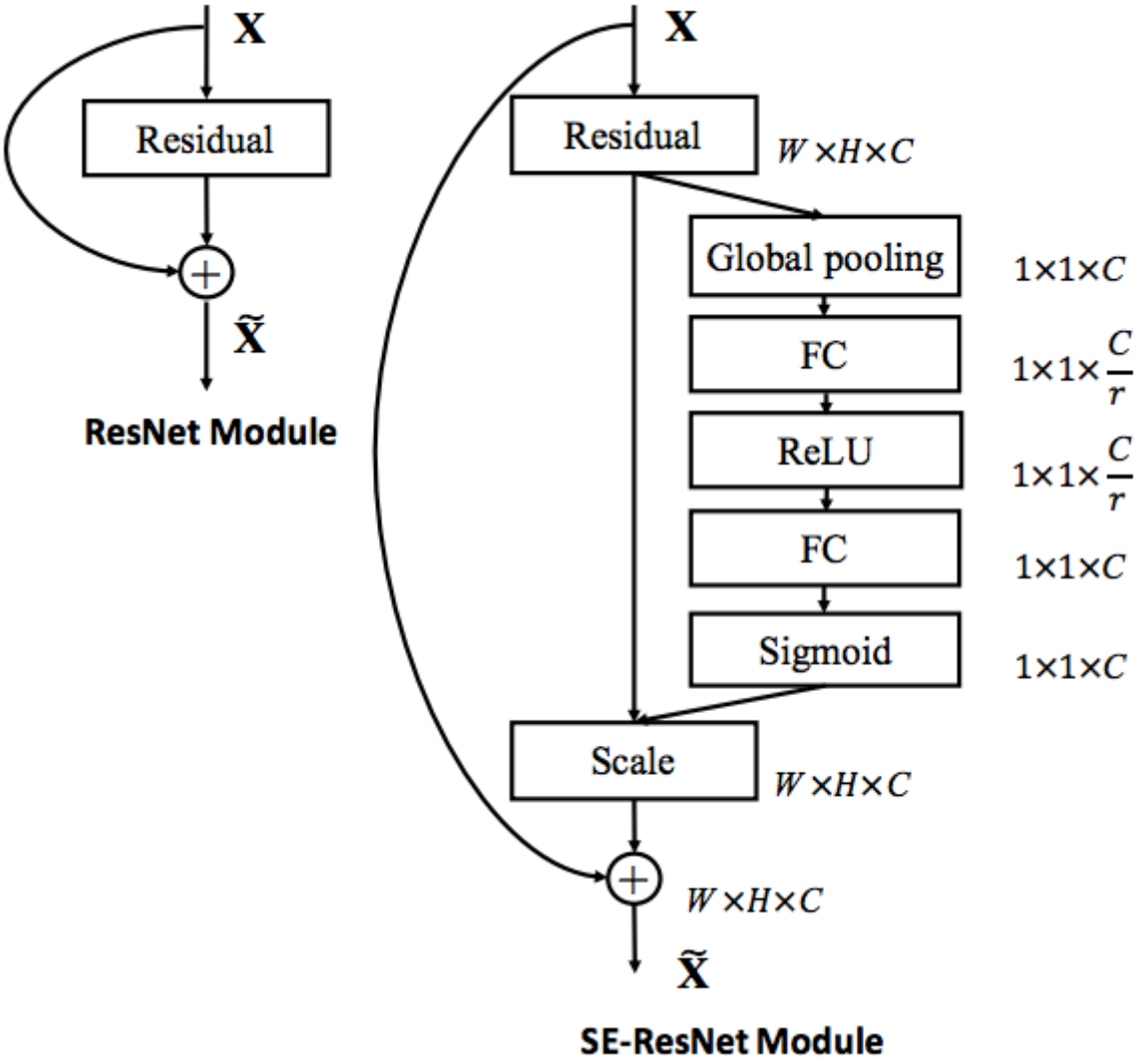
Trên pytorch thì tích chập chiều sâu tách biệt có thể được thực hiện ngay trên chính một layer `Conv2d` thông qua cấu hình tham số groups như đã giải thích ở mục 2.4 phần code pytorch. Mình nghĩ đây là một điểm tốt hơn trong thiết kế của pytorch.

Ngoài ra chúng ta lưu ý các thay đổi so với MobilenetV1 đó là: biến đổi tại layer output đã được thay bằng linear projection và độ sâu của các layer trung gian sâu hơn so với layer input và output.

4. MobileNetV3

Thêm Squeeze and Excitation (SE) vào block Residual để tạo thành một kiến trúc có độ chính xác cao hơn. Chúng ta cùng phân tích hình minh họa bên dưới để thấy sự khác biệt giữa kiến trúc Residual của mạng ResNet thông thường và SE-ResNet (áp dụng thêm Squeeze and Excitation).

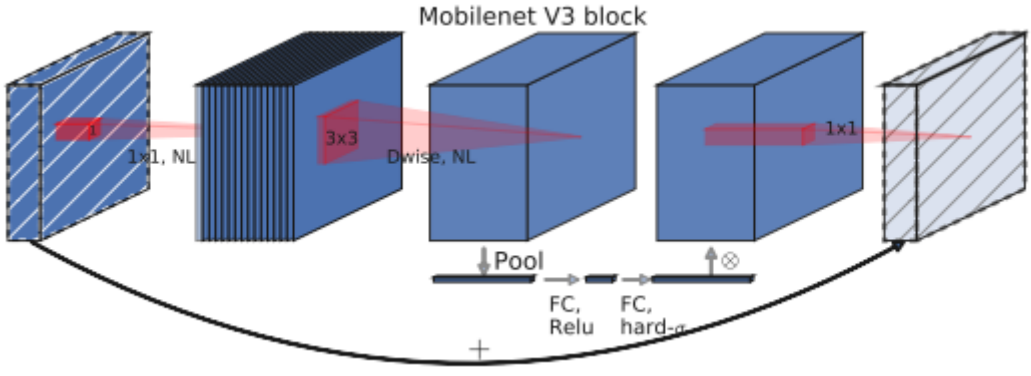
Top



SE-Resnet block. Source: Fig. 3: Squeeze-and-Excitation Networks (https://arxiv.org/pdf/1709.01507.pdf)

SE-ResNet áp dụng thêm một nhánh Global pooling có tác dụng ghi nhận bối cảnh của toàn bộ layer trước đó. Kết quả sau cùng ở nhánh này ta thu được một véc tơ global context được dùng để scale đầu vào X.

Tương tự như vậy SE được tích hợp vào kiến trúc của một residual block trong mobilenetV3 như sau:



Residual block MobilenetV2 + Squeeze and Excitation. Source: Fig 4: MobilenetV3 (https://arxiv.org/pdf/1905.02244.pdf)

Top

Để ý bạn sẽ thấy tại layer thứ 3 có một nhánh Squeeze and Excitation có kích thước (width x height) bằng `1 x 1` có tác dụng tổng hợp global context. Nhánh này lần lượt đi qua các biến đổi `FC -> Relu -> FC -> hard sigmoid` (FC là fully connected layer). Cuối cùng được nhân trực tiếp vào nhánh input để scale input theo global context. Các kiến trúc còn lại hoàn toàn giữ nguyên như MobileNetV2.

Để khởi tạo một inverted residual block trong mobilenetv3 chúng ta sẽ sử dụng lại code của MobileNetV2 và thêm vào một nhánh SE. Phần này mình sẽ không code mà xin dành cho bạn đọc như một bài tập tham khảo. Các bạn có thể đặt câu hỏi thảo luận tại AICode (https://www.facebook.com/groups/3235479620010379).

5. Kết luận

MobileNet là một trong những kiến trúc được ưa chuộng và sử dụng phổ biến bởi độ chính xác và hiệu năng tính toán. Qua bài này mình đã phân tích cho các bạn đặc điểm kiến trúc của một block layer MobileNet. Điểm mấu chốt giúp cho các mô hình MobileNet giảm thiểu số lượng tính toán đó là áp dụng tích chập tách biệt chiều sâu. Đồng thời qua thời gian, nhóm tác giả MobileNet đã lồng ghép các ưu điểm từ những kiến trúc CNN khác vào mô hình của mình để ngày càng cải thiện hơn về độ chính xác và hiệu năng. Xin cảm ơn các bạn đã theo dõi bài viết. Bên dưới là các phần tài liệu tham khảo.

6. Tài liệu

1. Mobilenetv2 next generation of On-Device computer vision networks - googleblog (https://ai.googleblog.com/2018/04/mobilenetv2-next-generation-of-on.html)
2. Mobilenetv2 inverted residuals and linear bottlenecks - (https://towardsdatascience.com/mobilenetv2-inverted-residuals-and-linear-bottlenecks-8a4362f4ffd5)
3. Squeezenet image classification (https://towardsdatascience.com/review-squeezenet-image-classification-e7414825581a)
4. Review squeeze and excitation network (https://towardsdatascience.com/review-senet-squeeze-and-excitation-network-winner-of-ilsvrc-2017-image-classification-a887b98b2883)
5. MobileNetV2: Inverted Residuals and Linear Bottlenecks - Mark Sandler, Andrew Howard (https://arxiv.org/abs/1801.04381)
6. MobilenetV3 - Andrew Howard, Mark Sandler (https://arxiv.org/pdf/1905.02244)

Top