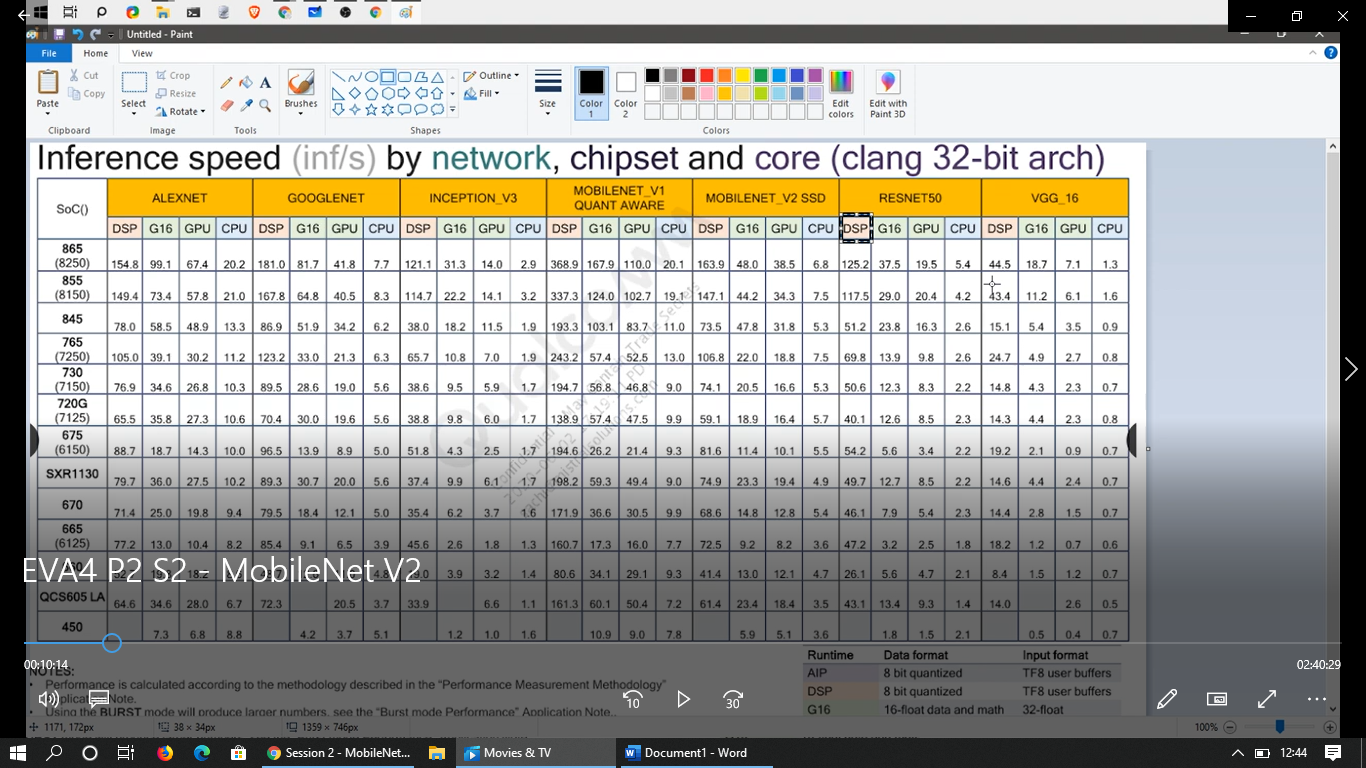
**Session 2 - Embedded Devices**

In this session we will look at models for embedded devices.

Mobile (embedded) devices need special “Mobile Neural Networks”

|  |  |  |
| --- | --- | --- |
| Mobile (embedded) devices need special "Mobile Neural Networks"  Differences between Cloud NN and Mobile NN: | | |
| **Neural Network Type** | **Cloud NN** | **Mobile NN** |
| Runtime | Fast on GPU, Slow on CPU | Superfast on GPU, Fast on CPU |
| Energy Usage | High | Low |
| Size | 100-800 MB | 1-5 MB |
| Training | Slow | Fast |
| Update | Direct | Over the Air |
| Accuracy | High | Lower |

1. Mobile NN is fast on CPU too because of very low size.
2. Energy usage means what amount of battery it is consuming. Its low in Mobile NN whereas High in Cloud NN.
3. Comparison chart



Here DSP means Digital Signal Processing., which will be inside mobile phones. And these what we discussed here are Ultra modern - Qualcom Processors. DSP is main part of processor which allows Image Processing. Here in Resnet50 its 125.2 when compared to CPU its high as these processors will have **neural compute engine inside**. – dedicated harware

When we compare here VGG and Resnet, speed is very low in VGG as the distinction between these two mainly because of fully connected layers specification in VGG. But as we are compressing original models into Mobile NN its going to give very less accuracy when compared to Cloud NN.

**Simple Old Tricks:**

These are very simple tricks which they considered when we have to move for embedded devices like Mobile.

In VGG and AlexNet 90% parameters or more than is there because of last few fully connected layers. That has been converted to GAP mechanism.

GAP after FC is going to give more capacity and number of classes here.

5\*5 and 7\*7 are replaced by 3\*3 because

1. fewer parameters (regularization effect!)
2. less compute (faster)
3. more non-linearity (better)

these points were considered good at the year 2014-15, but now the third point is not so true. It means converting from 7\*7 to 5\*5 and then 3\*3 we need 3 times adding 3\*3 Convolution to it. Means adding 3 times ReLu to it (non linearity). Its good thing. But actually its not good thing. We will see why.

**Squeezenet:**

We can get Deep Learning or Deep neural networks in Embedded devices is **SqueezeNet**.

Squeeze & Expand!

**Fire Block** Architecture:

Squeeze Blocks:

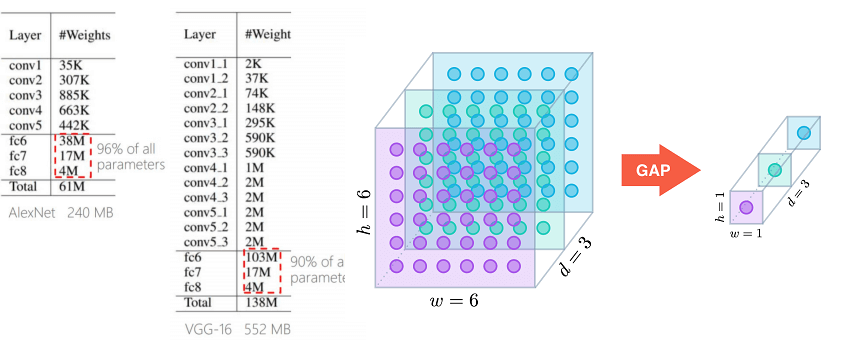
* Squeeze number of channels to reduce the computation in expanding the block
* use 1x1 filters instead of 3x3 (9x computation reduction) example when to convert 512 to 64, we can use 1\*1 instead 3\*3 filters.
* apply ReLU

Expand Block:

* do convolution using 1x1 and 3x3 filters
* apply ReLU

**SqueezeNet base vs AlexNet**

* 4.8MB vs 240 MB (50x smaller)
* Paper was titled "AlexNet level accuracy with 50x fewer parameters and 0.5mb model size"
* Same accuracy
* Slower than AlexNet in runtime
  + Downsampling (pooling) done late in the network (to increase accuracy)
  + Big memory Footprint. Like resnet 50 and resnet 34 max pooling to be done late at the layers because want to keep resolution same to preserve channel size and weights.



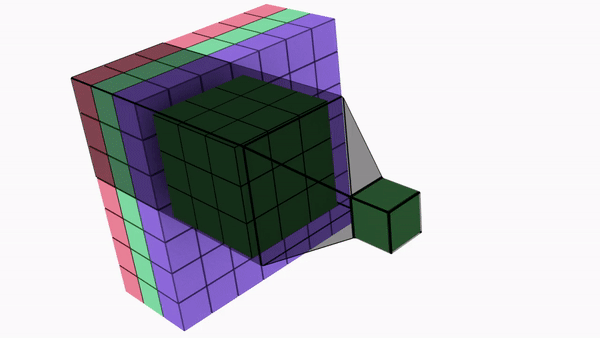
**Mobilenet V1: Coming from Google**

[MAIN PAPER - *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*, Howard et al, **2017**](https://arxiv.org/pdf/1704.04861.pdf)

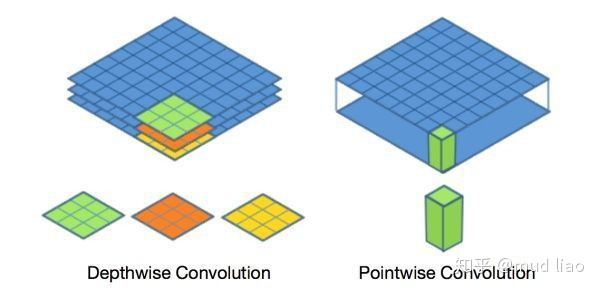
* a class of efficient models for mobile and embedded vision applications
* based on depth-wise separable convolutions

Author: For MobileNets the depthwise convolution applies a single filter to each input channel. The pointwise convolution then applies a 1×1 convolution to combine the outputs of the depthwise convolution. 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 reducing computation and model size.

**Normal Convolution**

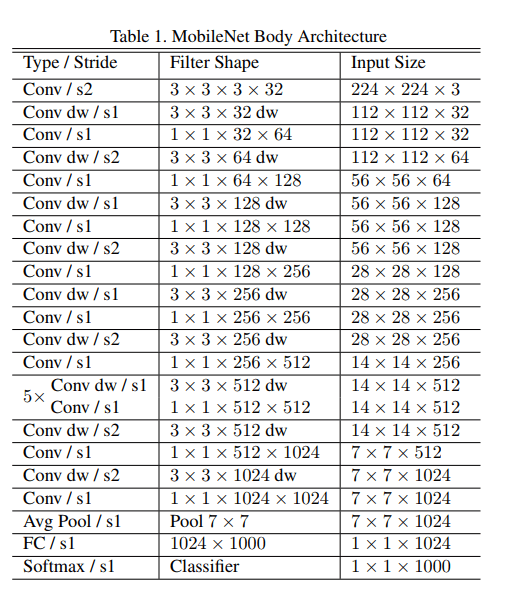


**Depthwise Seperable Convolution**:



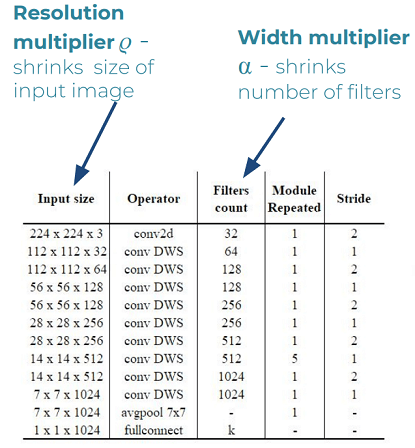
The overall architecture of Mobilenet is as follows:

1. 30 layers in total (we are solving ImageNet)
2. Convolution Layers with a stride of 2 (immediate increase in "jump" parameter) (here we are trying to have receptive field as high as possible)
3. Depthwise Layer followed by 1x1 to double the number of channels
4. Depthwise Layer with Stride of 2, followed by 1x1 to double the number of channels



Here after GAP, that’s Global average pooling, we add FC 1024, we will gather information every minute detail of a class. So, we crystallize and say the specific class of each.

As we going to use models in mobile phones, it has to be faster that’s why we use strategy **Resolution multiplier** (i.e) we are reducing the input size. One more strategy is **Width multiplier (i.e) shrinks number of filters by 0.5 every time.**



**Grid search means we are going to have right value of chi and alpha values here(resolution and width).**

**Optimized (read grid-search) architectures**

**(Computations) (Parameters)**

|  |  |  |
| --- | --- | --- |
| **Layer/Modification** | **Million Mult-Adds** | **Million Parameters** |
| **Convolution** | **462** | **2.36** |
| **Depthwise Separable Convs** | **52.3** | **0.27** |
| **α = 0.75** | **29.6** | **0.15** |
| **ρ = 0.714** | **15.1** | **0.15** |

**Example input 224 \* 0.714 = 160 as input channel size**

**And resolution 1024 \* 0.75 gives 768 kernels.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **ImageNet Accuracy** | **Million Mult-Adds** | **Million Parameters** |
| **0.5 MobileNet-160** | **60.2%** | **76** | **1.32** |
| **SqueezeNet** | **57.5%** | **1700** | **1.25** |
| **AlexNet** | **57.2%** | **720** | **60** |

Why we have 1700 million computations here in Squeezenet is that we are not using max pooling. So it gives original computations much.

**Quantization** – instead of floating point units comparison in speeds between different architectures we try to do in ints.

Xnor.ai is best example for binary convolutional neural networs. 🡪 xnor.net in google. (paper)

Quantization needs fully connected layers to work with.- older one.

Range & Step:

* + - get layer weights min & max
    - map min & max to desired type min & max
    - example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Min** | **Max** | **Zero** | **step** |
| float32 | -20 | 40 | 0 | ~0.235 |
| int8 | -127 | 127 | -42 | 1 |

(20+40) = 60 and 127+127 = 256 (60/256 = `0.235) step size.

-127 + (20/60) \* 256 = -41.6667 = ~ -42 (zero)

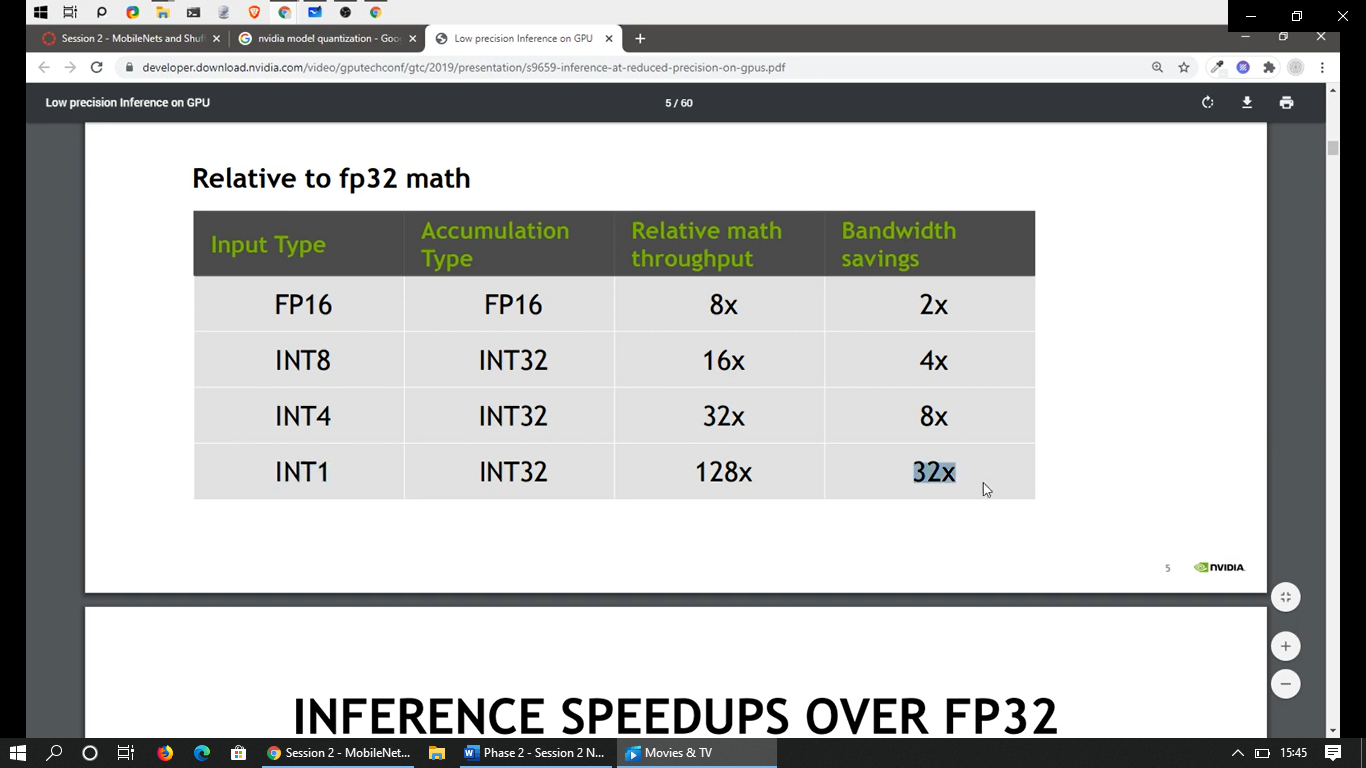
**We want to quantize here the channels not the kernels. Kernels going to quantize but the range here is for channels.**

**Quantization of Trained Model**

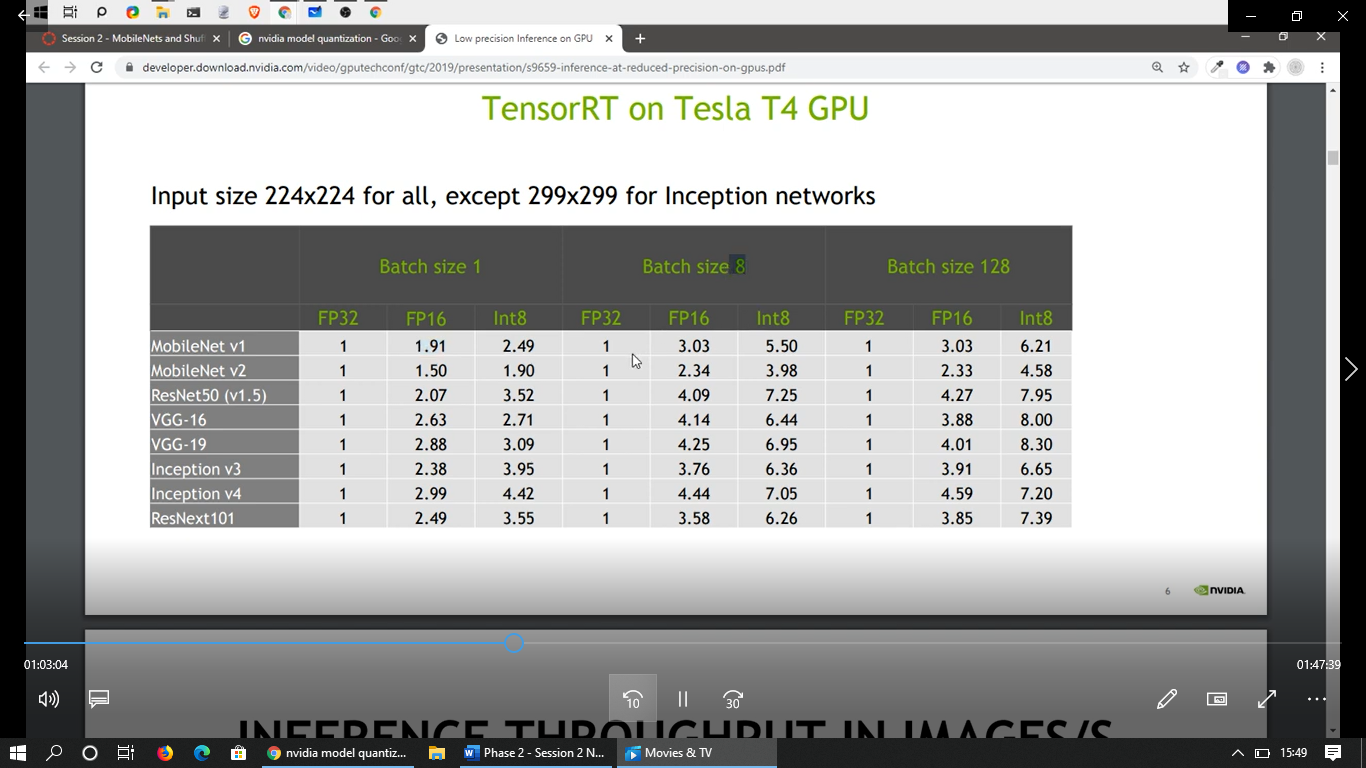
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Model** | **Million MACs** | **Million Parameters** | **Top-1 Accuracy** | **Top-5 Accuracy** | **Size [MB}** |
| **MobileNetV1** | **569** | **4.24** | **70.9** | **89.9** | **16.9** |
| **MobileNetV1 Quantized** | **569** | **4.24** | **69.7** | **89.5** | **4.3** |

Quantization while training. So when you quantize float 16 🡪 int 8 🡪 int 4🡪 int 1 we are going to save bandwidths. This is related to nvidia model quantization.





According to batch size increase, quantize from one floating point to other and int, speed increases too.



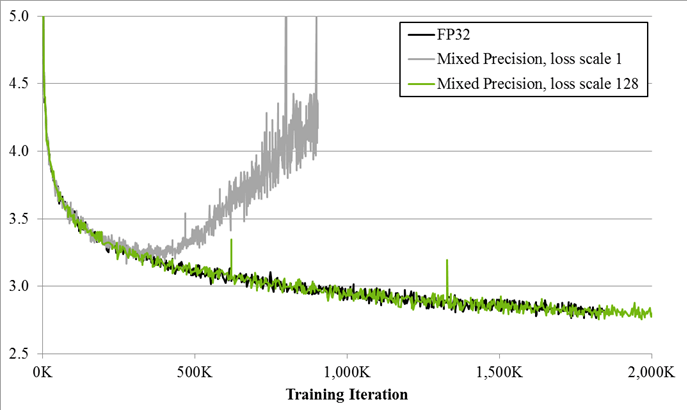
**Training with Quantization:**

* + - learn ranges during training
    - forward pass - quantized values
    - backward pass - float values

if training with quantization has to work, have to do lot of optimization here.

**Mixed Precision Training Iteration:**

1. **Make an FP16 copy of the weights**
2. Forward propagate using FP16 weights and activations
3. **Multiply the resulting loss by the scale factor S – why we doing this** (look at below figure)
4. Backward propagate using FP16 weights, activations, and their gradients
5. **Multiply the weight gradients by 1/S**
6. Optionally process the weight gradients (gradient clipping, weight decay, etc.)
7. Update the master copy of weights in FP32



When its of FP 32, according to training iteration loss was converging down. But when loss was scaled to factor 1 that’s constant, it was diverging entirely. But the same applied loss scale by 128, it was again converging back.

0.0003 << F32, 0.03 << F16

If 1/10000 is a gradient and 4 is loss value, it backpropagates it as 0.0004 value and will be in FP32. Now 0.0004 multiplied by scale factor 100 its 0.04 so will come in FP16.

Then again multilply the value by 1/s that is 0.04 / 100 = 0.0004 back to FP 32 weights.

Pytorch article on Apr 9 2020 - <https://analyticsindiamag.com/pytorch-mixed-precision-training/>

Initialising mixed precision in PyTorch using one line of code:

***model, optimizer = amp.initialize(model, optimizer, opt\_level="O1")***

***01 just indicates mixed precision.***

**Quiz:**

1. Mobile DNNs are lower in accuracy compared to cloud DNNs. Why?

* Because we have made them that way.( with respect to processing power and other numbers)

1. Delaying down sampling of the images can increase accuracy – True.
2. Depthwise Convolutions must be faster on Mobile phones. Actual answer is who knows because mobile phones are not designed for AI still. Its faster in Iphones and qualcom series 800 series phones.
3. If our float range was between -2.45 to 1.45 and we quantized it to integers, the 0.0 float point will be equal to which integer?

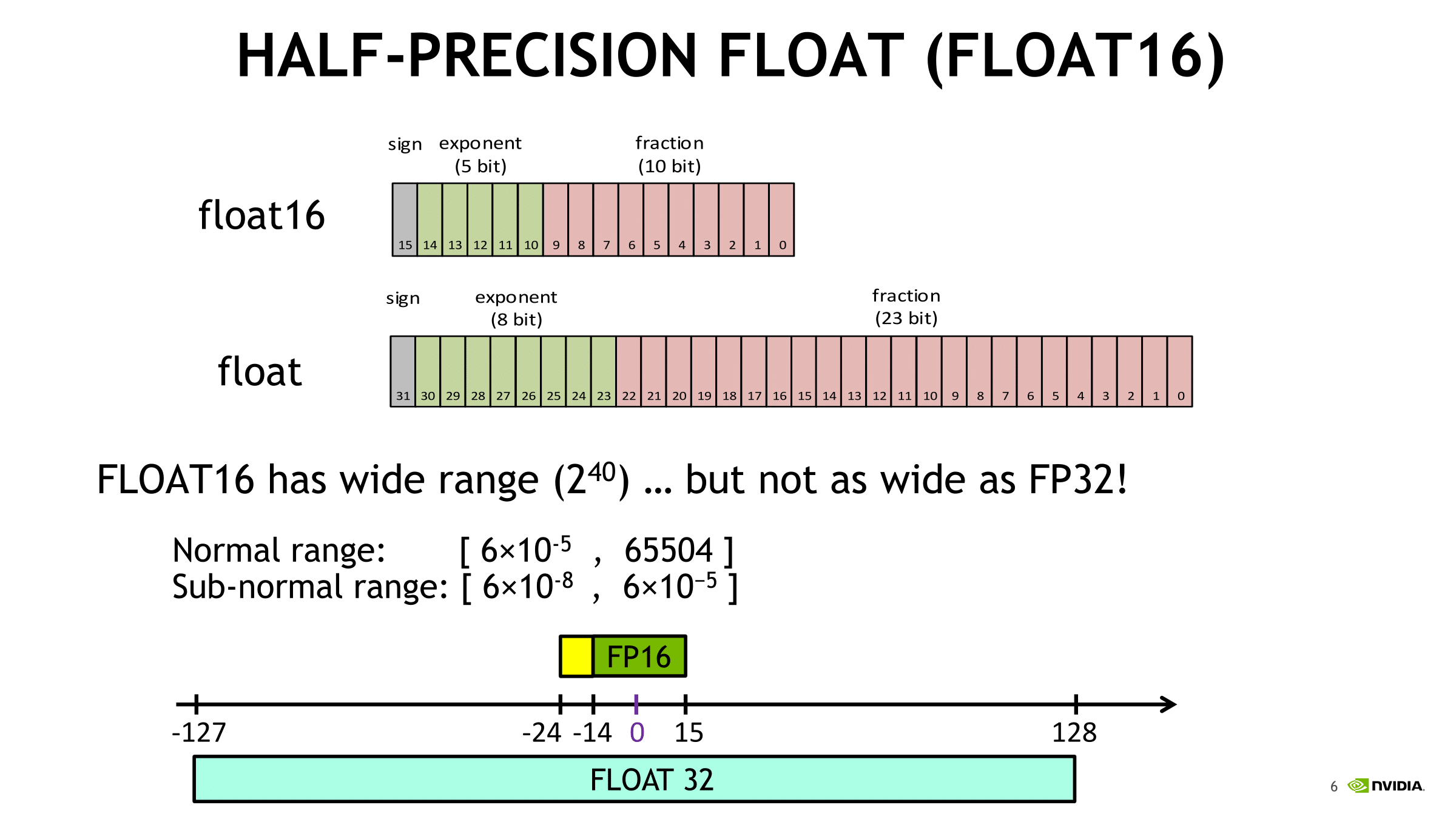
-127+ (2.45)/(2.45+1.45) \* 255 = 33.192 = ~ 34 (Approx answer)

1. what we need to do with the weight gradients in mixed precision training, if we are dividing the loss by S?

If we multiply the loss by S, we need to divide weight gradients by 1/S, therefore here we are dividing the loss by S, so multiplying the weight gradients by S.

Ans. Multiply gradients by S

**Half Precision Float (Float 16):**



Here the red blocks are the important numbers which matters more. For example 0.000000456789 here 000000 after decimal point will come under Green line, but after the zeroes what numbers are there in red line matters more. To make it precised we can push and make it important ans convert using the technique of Mixed precision training.

**ShuffleNet:**

**An extremely efficient Convolutional Neural Network for Mobile Devices.**

Megvii Inc (a.k.a Face++) introduces an extremely computation-efficient CNN architecture named ShuffleNet, which is designed especially for mobile devices with very limited computing power (e.g. 10-150 MFLOPs).

Why we are going to discuss about shufflenet is just one important and interesting concept in Shufflenet is **Shuffleblock, which efficiently utilizes the purpose of channels or how channels can be handled.**

* The new architecture utilizes two new operations, pointwise group convolution and **channel shuffle, to greatly reduce computation cost while maintaining accuracy.**
* Group convolution is 56\*56\*300 means we separate into 3 groups so, 3\* 56\*56\*100
* Pointwise group convolution means 3 \*1\*1\*100\*100, we will see where we going to use it.
* Experiments on ImageNet classification and MS COCO object detection demonstrate the superior performance of ShuffleNet over other structures, e.g. lower top-1 error (absolute 7.8%) than MobileNet on ImageNet classification task, under 40MFLOPs. ShuffleNet is 13x faster than AlexNet while maintaining comparable accuracy.
* Mainly mobile networks cannot have pointwise convolution as its very dense. So better to follow group pointwise convolution to avoid complexity in smaller networks.
* **Idea**:
* Xception and ResNeXt architectures are awesome but become less efficient in extremely small networks because of the costly dense (1x1) convolutions. **They proposed using pointwise group convolutions to reduce the computation complexity of 1x1 convolutions**.

Example : 4 \* 56\*56\*128 | **4 \* 1 \* 1 \* 128 \* 32** (Cheaper)

1. To overcome the side effects brought by group convolutions, they come up with a novel **channel shuffle** operation to help the information flowing across feature channels.
2. Hence, ShuffleNet allows more feature map channels, which helps to encode more information and is especially critical to the performance of very small networks.
3. **For each residual unit in ResNeXt, the pointwise convolutions occupy 93.4% multiplication-adds**(cardinality 32).

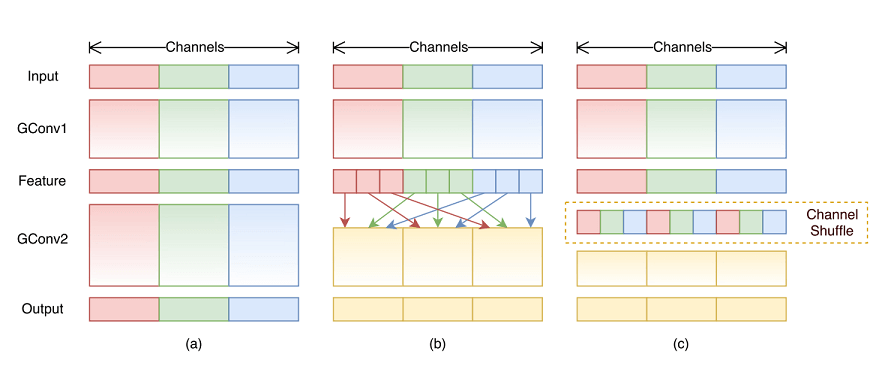


Figure 1(a) illustrates a situation of two stacked group convolution layers, which has twofold properties, blocking the information flow between channel groups and weakening representations.

Figure 1(b) shows that the group convolution is allowed to obtain input data from different groups. Notice that the input channels are fully related to the output ones.

Figure 1(c) sets up the feature map from the previous group layer, which is then implemented by a channel shuffle operation. Channel shuffle operation is able to construct more powerful structures with multiple group convolutional layers.

But there is a drawback, output from a certain channel is only derived from a small fraction of input channels. **How do we fix this problem?**

* If we allow group convolutions to obtain input data from different groups (b), the input and output channels will be fully related. Specifically, for the feature map generated from the previous group layer, we can first divide the channels in each group into several subgroups, then feed each group in the next layer with different groups. This can be efficiently and elegantly implemented by a **channel shuffle**operation.
* Suppose a convolutional layer with *g* groups whose output has *g*×*n* channels; we first **reshape** the output channel dimension into (*g*, *n*), **transposing,** and then **flattening** it back as the input of the next layer.

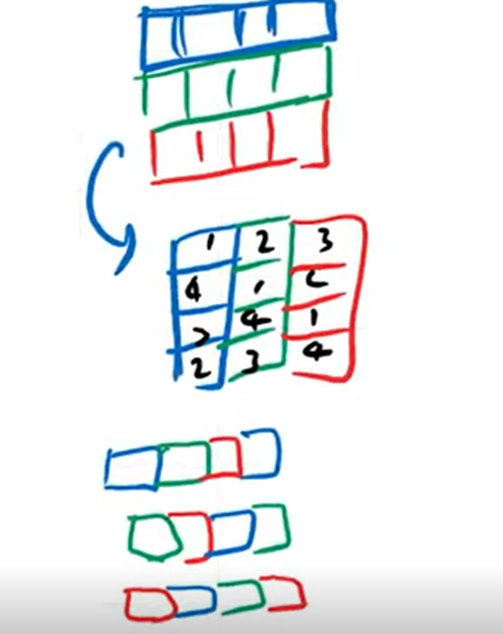
**Channel Shuffle:**

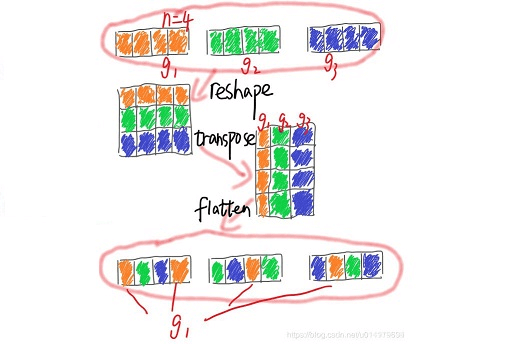
Let's understand, by example:

1. Let Gr = 3,  and N = 4 (channels in each group)
   1. RRRR GGGG BBBB
2. Reshape to (Gr, N)
   1. RRRR
   2. GGGG
   3. BBBB
3. Transpose to (N, Gr)
   1. RGB
   2. RGB
   3. RGB
   4. RGB
4. Flatten (not x, y, but channels, don't forget)
   1. RGB RGB RGB RGB
5. Convert back to Gr = 3 and N = 4
   1. RGBR GBRG BRGB

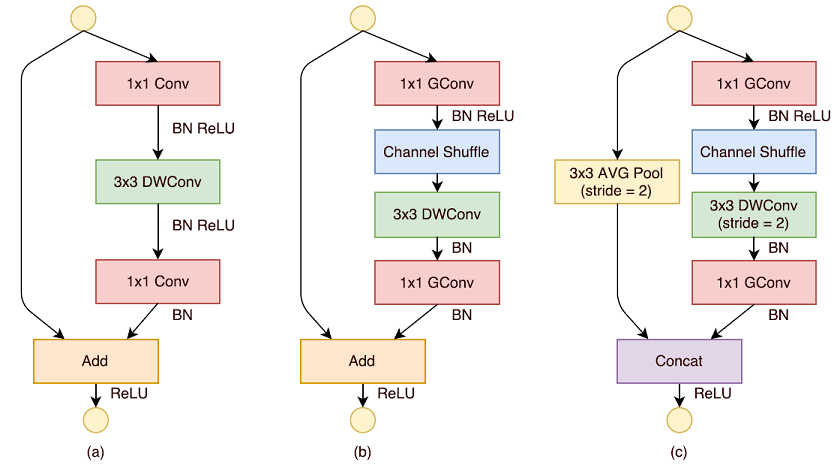
56 \* 56 \* 12 | 3 \* 3\* 3\* 4 |

**56 \* 56 \* 12 | 1 \* 1 \* 12 \* 12 | 56 \* 56 \* 12**



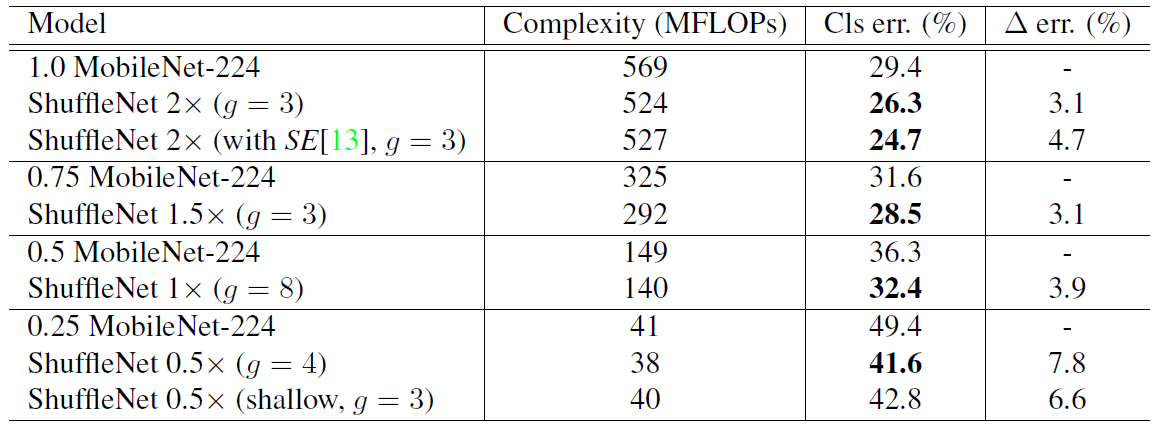


**Shufflenet Unit: (a) bottleneck unit with depthwise convolution (DWConv), (b) ShuffleNet unit with pointwise group convolution (GConv) and channel shuffle, (c) ShuffleNet unit with stride = 2.**



* + - (a) Bottleneck Unit: This is a standard residual bottleneck unit, but with depthwise convolution used. (Depthwise convolution is used in [MobileNetV1 (Links to an external site.)](https://towardsdatascience.com/review-mobilenetv1-depthwise-separable-convolution-light-weight-model-a382df364b69).) With 1×1 then 3×3 DW then 1×1 convolutions used, it can be also treated as a bottleneck type of depthwise separable convolution used in [MobileNetV2 (Links to an external site.)](https://towardsdatascience.com/review-mobilenetv2-light-weight-model-image-classification-8febb490e61c).
    - (b) ShuffleNet Unit: The first and second 1×1 convolutions are replaced by group convolutions. A channel shuffle is applied after the first 1×1 convolution.
    - (c) ShuffleNet Unit with Stride=2: When stride is applied, a 3×3 average pooling on the shortcut path is added. Also, the element-wise addition is replaced with channel concatenation, which makes it easy to enlarge channel dimension with little extra computation cost.
    - ReLU is not applied to 3x3 DWConv
    - Given the input *c*×*h*×*w*, and bottleneck channels *m*, [ResNet (Links to an external site.)](https://towardsdatascience.com/review-resnet-winner-of-ilsvrc-2015-image-classification-localization-detection-e39402bfa5d8" \t "_blank) unit requires *hw*(2*cm*+9*m*²) FLOPs and [ResNeXt (Links to an external site.)](https://towardsdatascience.com/review-resnext-1st-runner-up-of-ilsvrc-2016-image-classification-15d7f17b42ac" \t "_blank) requires *hw*(2*cm*+9*m*²/*g*) FLOPs, while ShuffleNet only requires *hw*(2*cm*/*g*+9*m*) FLOPs where g is the number of group convolutions.
    - In other words, given a computational budget, ShuffleNet can use wider feature maps. We find this is critical for small networks, as tiny networks usually have an insufficient number of channels to process the information.

**Comparison with**[**MobileNetV1 (Links to an external site.)**](https://towardsdatascience.com/review-mobilenetv1-depthwise-separable-convolution-light-weight-model-a382df364b69)



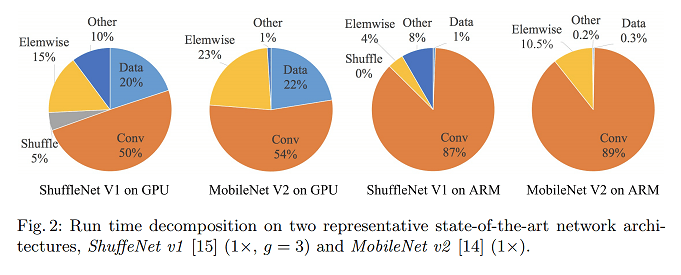
* + - ShuffleNet models are superior to [MobileNetV1 (Links to an external site.)](https://towardsdatascience.com/review-mobilenetv1-depthwise-separable-convolution-light-weight-model-a382df364b69) for all the complexities.
    - Though the ShuffleNet network is specially designed for small models (< 150 MFLOPs), it is still better than [MobileNetV1 (Links to an external site.)](https://towardsdatascience.com/review-mobilenetv1-depthwise-separable-convolution-light-weight-model-a382df364b69) for higher computation cost, e.g. 3.1% more accurate than [MobileNetV1 (Links to an external site.)](https://towardsdatascience.com/review-mobilenetv1-depthwise-separable-convolution-light-weight-model-a382df364b69) at the cost of 500 MFLOPs.
    - The simple architecture design also makes it easy to equip ShuffeNets with the latest advances such as Squeeze-and-Excitation (SE) blocks. (Hope I can review SENet in the future.)
    - ShuffleNets with SE modules boosting the top-1 error of ShuffleNet 2× to 24.7%, but are usually 25 to 40% slower than the “raw” ShuffleNets on mobile devices, which implies that actual speedup evaluation is critical on low-cost architecture design.

**Flops are Not Reliable:**

To measure the computation complexity, a widely used metric is the number of float-point operations or FLOPS. However, FLOPs are an indirect metric. It is an approximation of, but usually not equivalent to the direct metric that we rarely care about, such as the speed of latency. Such discrepancy has been noticed in many networks.

 There are three reasons for this discrepancy:

* First, several important factors that have a considerable effect on speed are not taken into account by FLOPs. One such factor is memory access cost (MAC). Such cost constitutes a large portion of run time in certain operations like group convolutions. It could be a bottleneck on devices with strong computation power (like GPUs).
* Another one is the **degree of parallelism**. A model with a high degree of parallelism could be much faster than another one with low.
* Finally, operations with the same FLOPs could have different running times, depending on the platform.

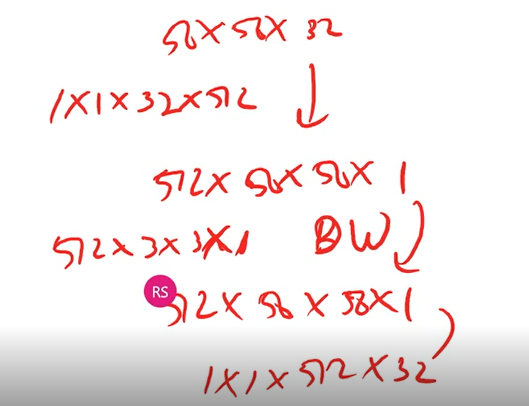


With these observations, we propose that two principles should be considered for effective network architecture design. First, the direct metric (e.g., speed) should be used instead of the indirect ones (e.g., FLOPs). Second, such metrics should be evaluated on the target platform. In **this** work, we follow the two principles and propose a more effective network architecture.

**MobileNet V2:**

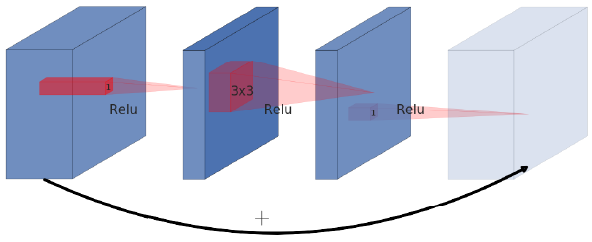
56 \* 56 \* 512 | (1 \* 1 \* 512 \* 32) > 56 \* 56 \* 32 (3 \* 3 \* 32 \* 32) | 1 \*1 \* 32 \* 512

Inverted residuals concept:

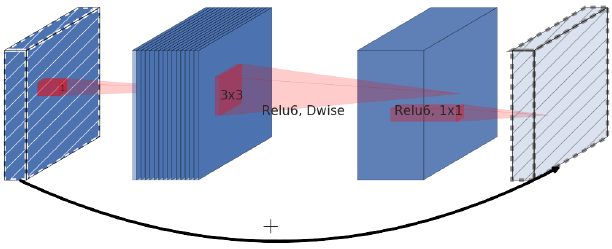


* Inverted Residuals
* Linear Bottlenecks
* Smaller Memory Footprint
* Faster
* Smaller
* Uses ReLU6

Residuals



Inverted Residuals

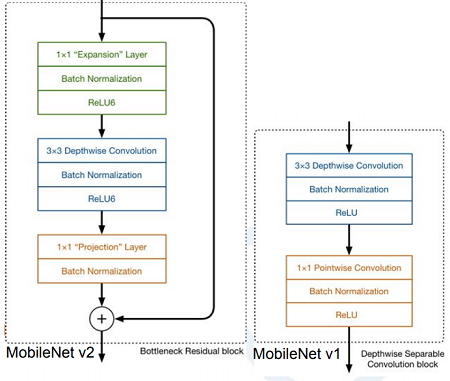


the inverted block has far fewer parameters

**Linear Bottlenecks**

The reason we use non-linear activation functions in neural networks is that multiple matrix multiplications cannot be reduced to a single numerical operation. It allows us to build neural networks that have multiple layers. At the same time the activation function ReLU, which is commonly used in neural networks, discards values that are smaller than 0. This loss of information can be tackled by increasing the number of channels in order to increase the capacity of the network.

With inverted residual blocks, we do the opposite and squeeze the layers where the skip connections are linked. This hurts the performance of the network. The authors introduced the idea of a linear bottleneck where the last convolution of a residual block has a linear output before it’s added to the initial activations.

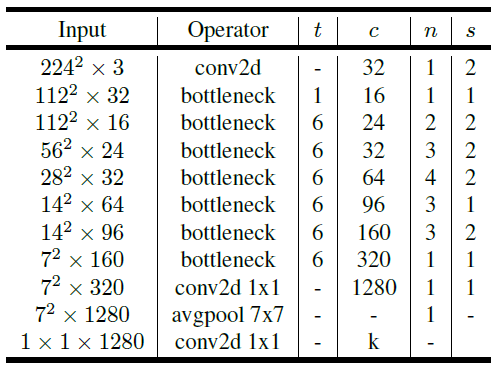


**ReLU6**  
The snippet above shows the structure of a convolutional block that incorporates inverted residuals and linear bottlenecks. If you want to match MobileNetV2 as closely as possible there are two other pieces you need. The first aspect simply adds Batch Normalization behind every convolutional layer as you’re probably used to by now anyways.

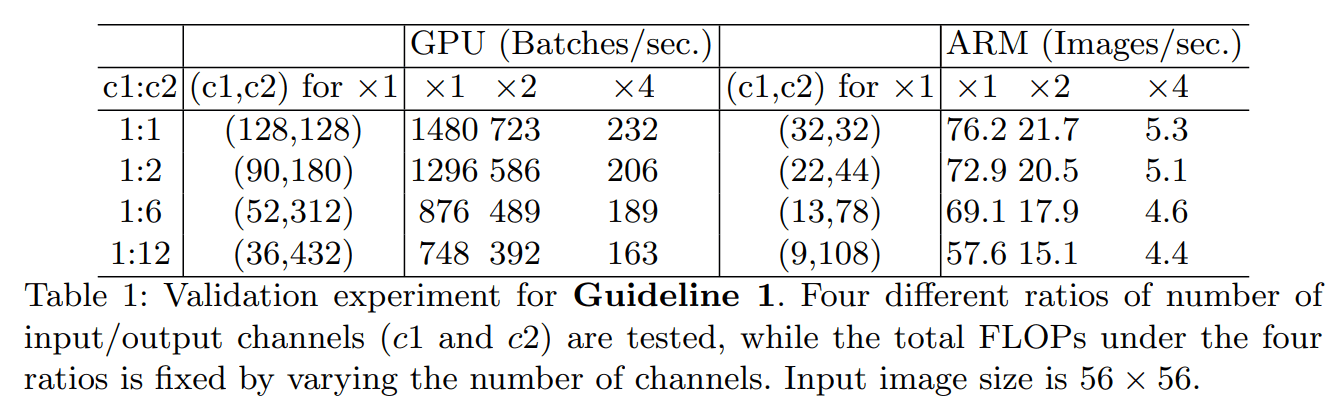
The second addition is not quite as common. The authors use ReLU6 instead of ReLU, which limits the value of activations to a maximum of…well…6. The activation is linear as long as it’s between 0 and 6.



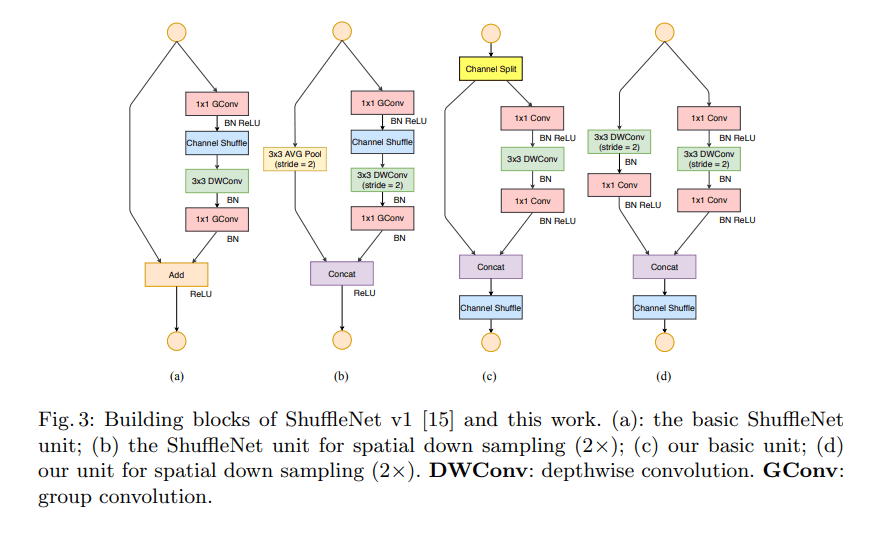
* Channels in the architecture increases like:
  + 224-3 >> 112-16 >> 56-24 >> 28-32 >> 14-64 >> 14-96 >> 7-160 >> 7-320 >> 1-1280
* But inside there may be an expansion factor of 6!
* so 24 could be:
  + 24 >> 144 >> 144>> 24



**Equal channel with minimizes memory access cost**



Shufflenet V2:



Channels are split into 2 branches, one goes as identity, another with 1-3-1 route, and then they are concatenated (unlike ResNet). Total channels are kept the same before and after.

**Prepared By: G Uday Kiran & Srilakshmi V**