**Week 2 – Notes**

**Case Studies**

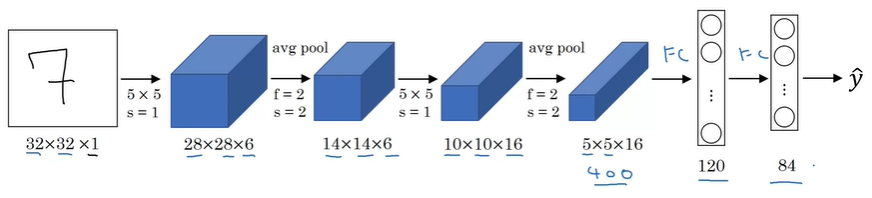
**Why look at case studies?**

There are classic networks, such as: LeNet-5, AlexNet and VGG

More modern networks are the ResNet which has up to 152 layers and Inception

**Classic Networks**

LeNet-5 was published in Yann LeCun in 1998 and it has ~60k parameters



Differences compared to nowadays:

At that time, avg pooling was more popular than the max pooling

Instead of softmax as an activation function of the output layer, another function was used

Sigmoid and tanh were used instead of ReLU

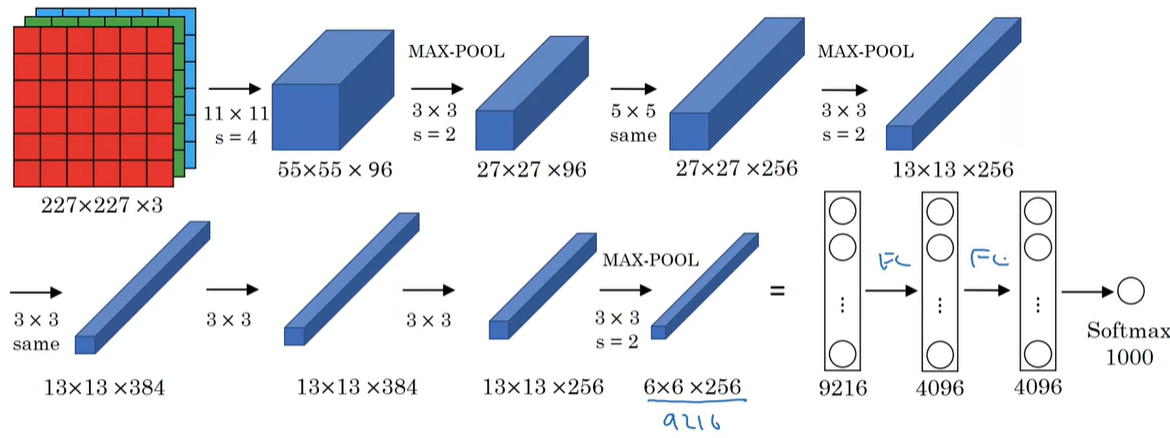
The non-linearity was applied after the pooling layer

Things that are still valid today:

As the data progress through the network, nH and nW decrease and nC increases

The architecture is relevant: conv -> pool -> conv -> pool -> fc -> fc -> output

AlexNet was published in 2012 and it has ~60 million parameters



It uses convolutions with large kernels and sequential convolutions with the same kernel (even with the same padding and stride)

Things that are still valid today:

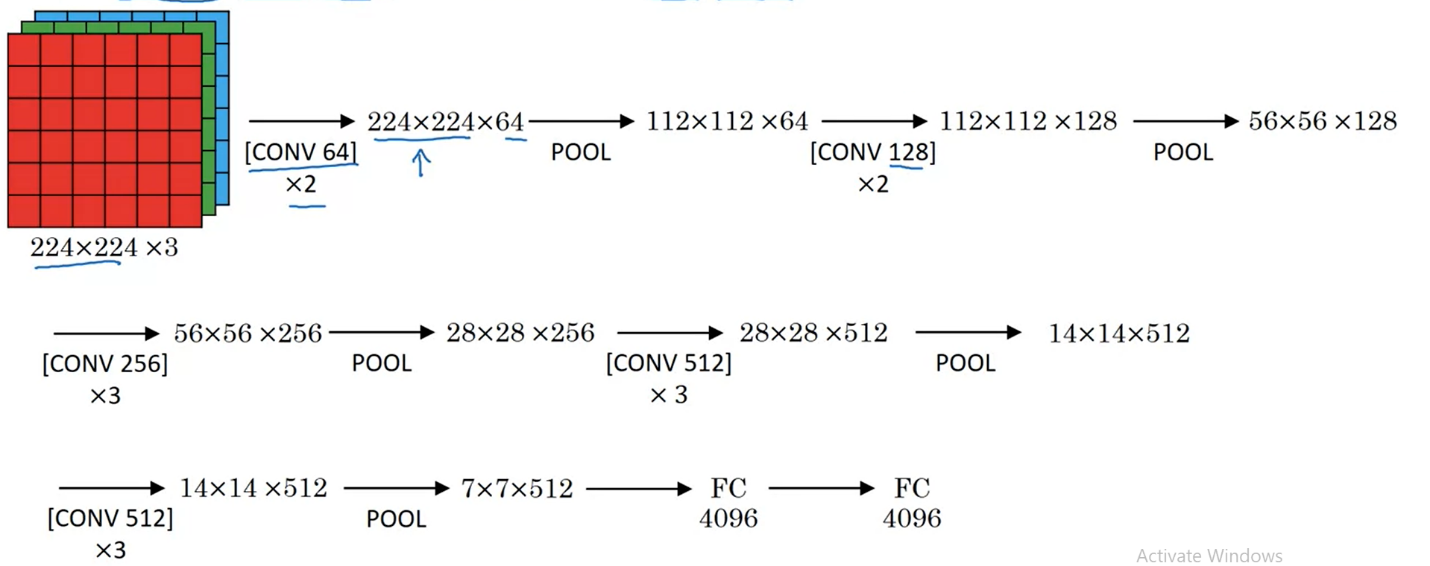
The activation function is ReLU

Differences compared to nowadays:

Used a Local Response Normalization (for each pixel across all channels) – turned out to not be that effective

VGG-16 was published in 2015 and it has ~138 million parameters

There also is a version called VGG-19



This network simplified very much the architecture of CNNs, which became more and more complicated

The authors used only 2 types of layers: Conv2D with 3x3 filters, a stride of 1 and the same padding and max pooling layer with a kernel of 2x2 and a stride of 2

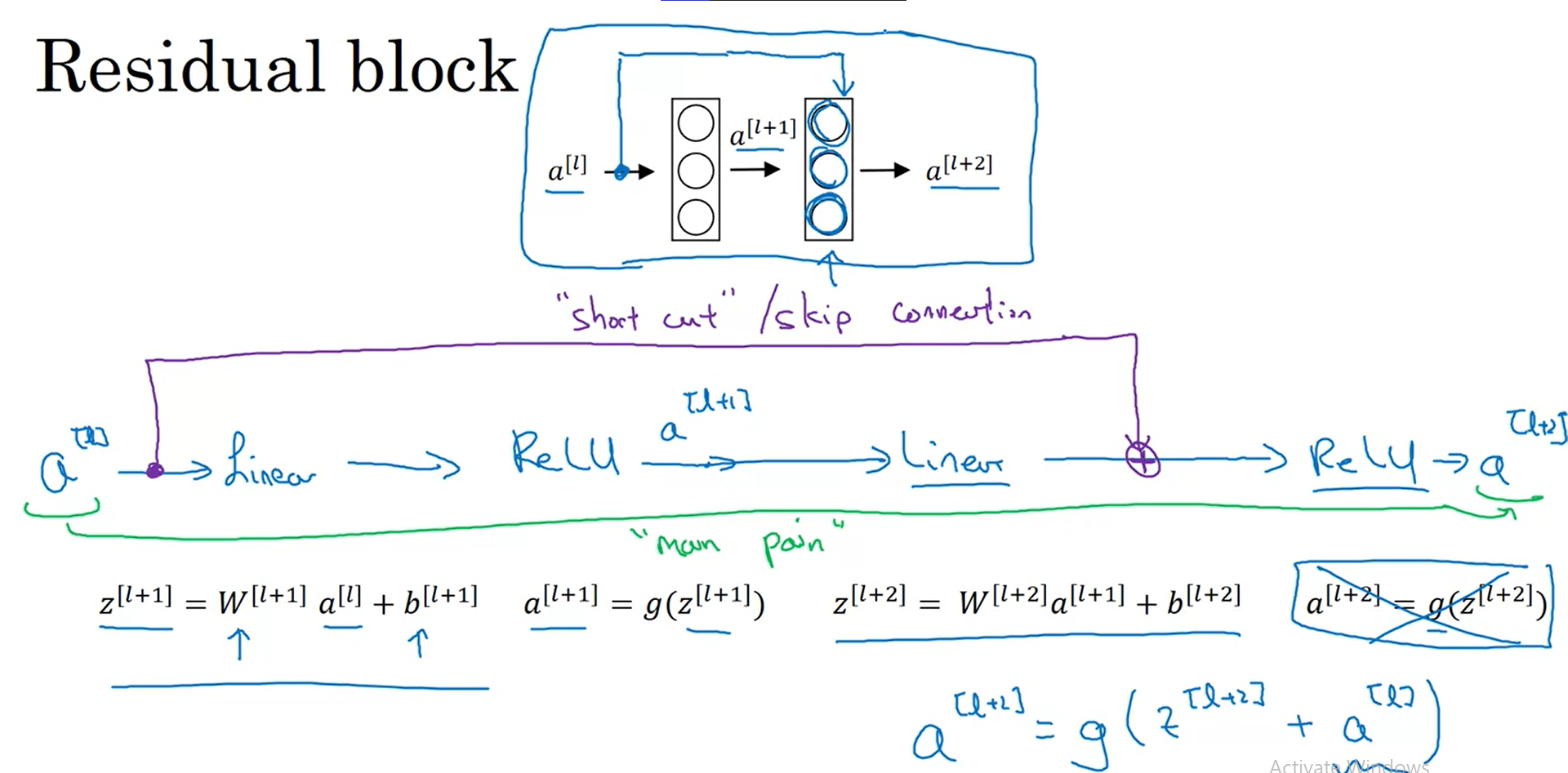
Interesting idea: you can use consecutive identical conv layers

**ResNets**

Deeper networks have problems such as vanishing (tanh or sigmoid activation functions) or exploding gradients (relu activation function), so to solve that we can use short cuts / skip connections

This means we take the outputs of a layer and we add to the output of the layer, but before we apply the activation function (in our case ReLU)

We don’t add after we apply the activation function because we want to avoid the exploding gradients; additionally, we don’t add before the we compute the current Z to not damage the computation of Z of the current layer



A network w/o skip connections is called a plain network

The skip connection is added so that each time we jump over one layer

Plain network: in reality, as we add more layers, after one point the training error increases

ResNets: as we add more layers, the training error decreases (at most there’s a plateau)

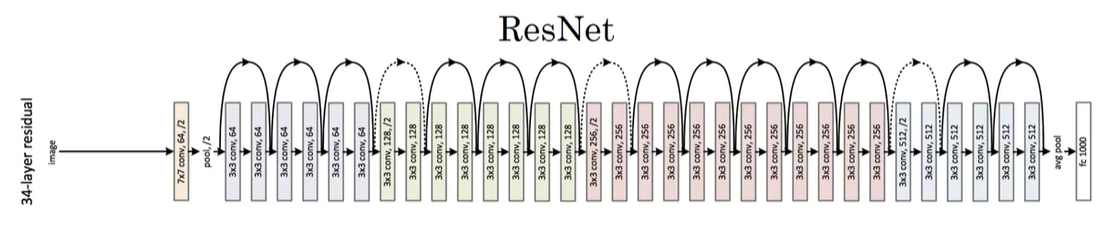
**Why ResNets Work?**

Let’s say you have a big nn that has the output a[l]

If you add 2 new layers and use a skip connection from a[l] to a[l+2] (layer l and l + 2 have outputs with the same size), then in the worst case because you use weight decay (L2 regularization) your network has w[l+2] and b[l+2] as zero values and then you will compute in the l+2 layer g(a[l]) which is equal to a[l] in the case of g = ReLU; thus, there’s an identity function because a[l+2] == a[l]

On the contrary, if you network can learn anything, then the skip connection is useful, but in any case, it doesn’t do any harm to you network

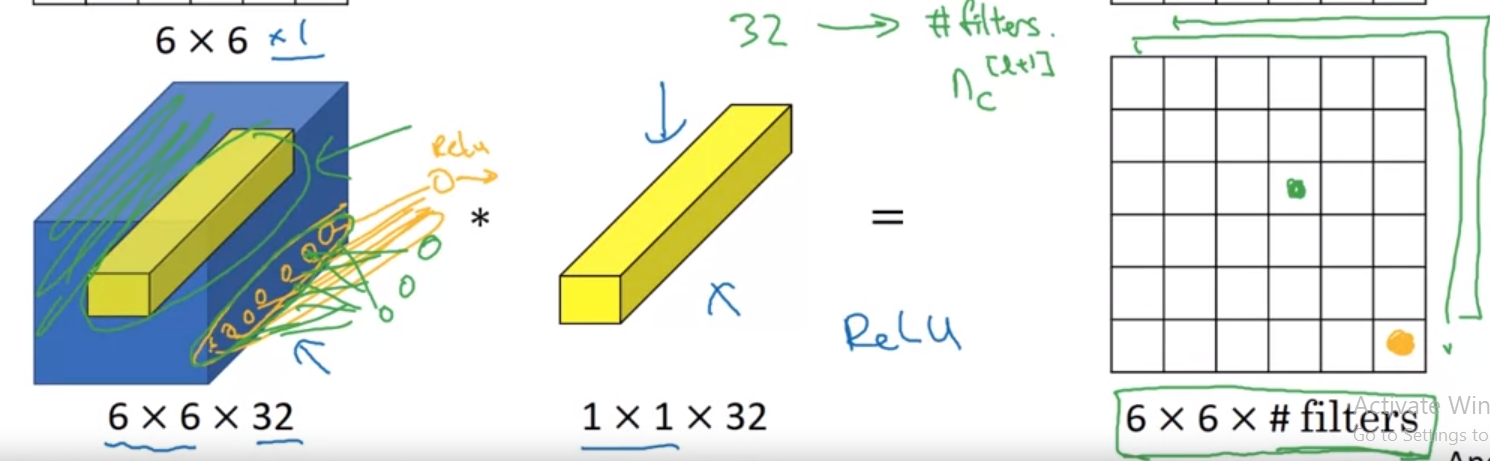
If the size of a[l] and a[l+2] is different, then when adding the skip connection, we multiply a[l] with Ws, so that the Ws \* a[l] has the same shape as a[l+2]; however, mostly same convolutions are used, so that we have outputs with identical shapes



We can see that after applying a pooling layer, the dimensions of the outputs don’t match, so we have to use a skip connection + adjust the output of the forwarded layer

**Networks in Networks and 1x1 Convolutions**

A 1x1 convolution can be used to learn more complicated non-linearities of the network and to decrease number of channels

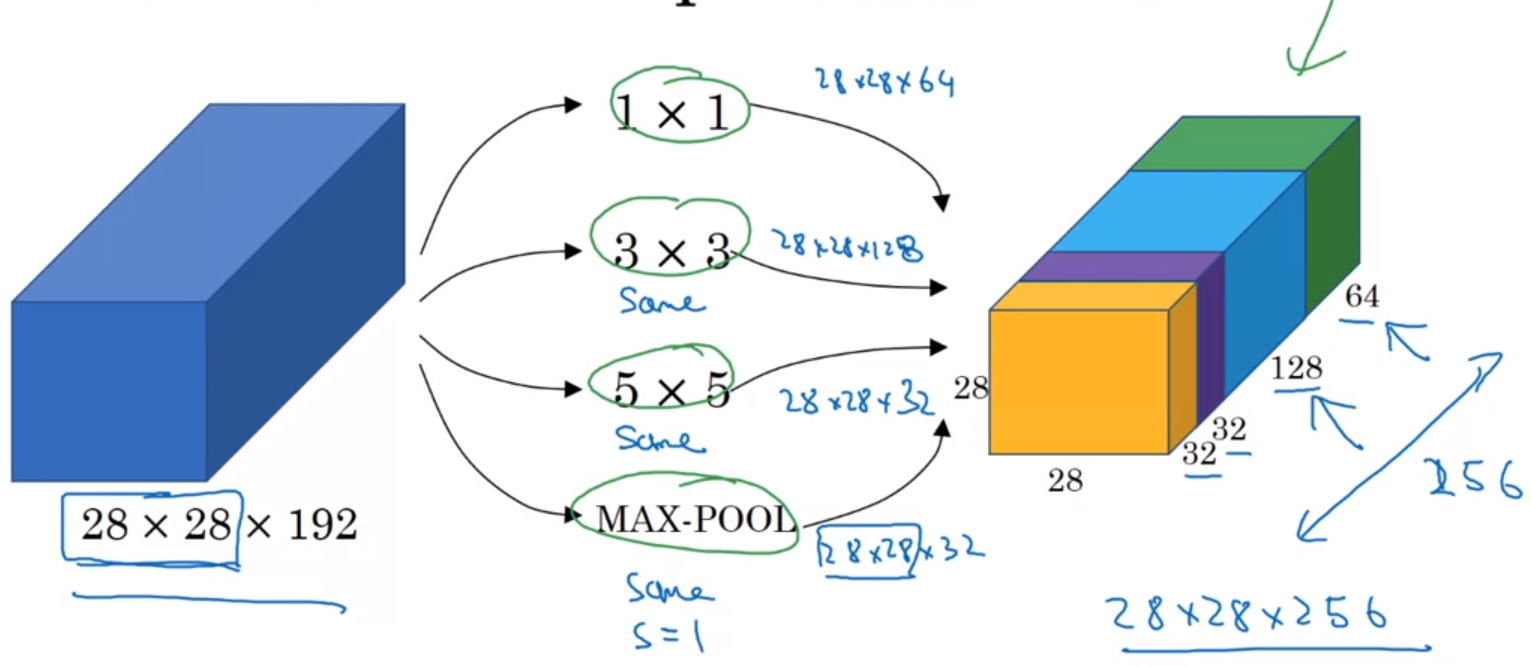
****

This type of convolution can be perceived as a network in network, because one filter produces a feature map with one channel, where each individual value is the output of a simple perceptron, considering that we are multiplying the values of the kernel with the associated values from the input, then we add them and apply the ReLU function as the activation one

We can have as many filters as we want

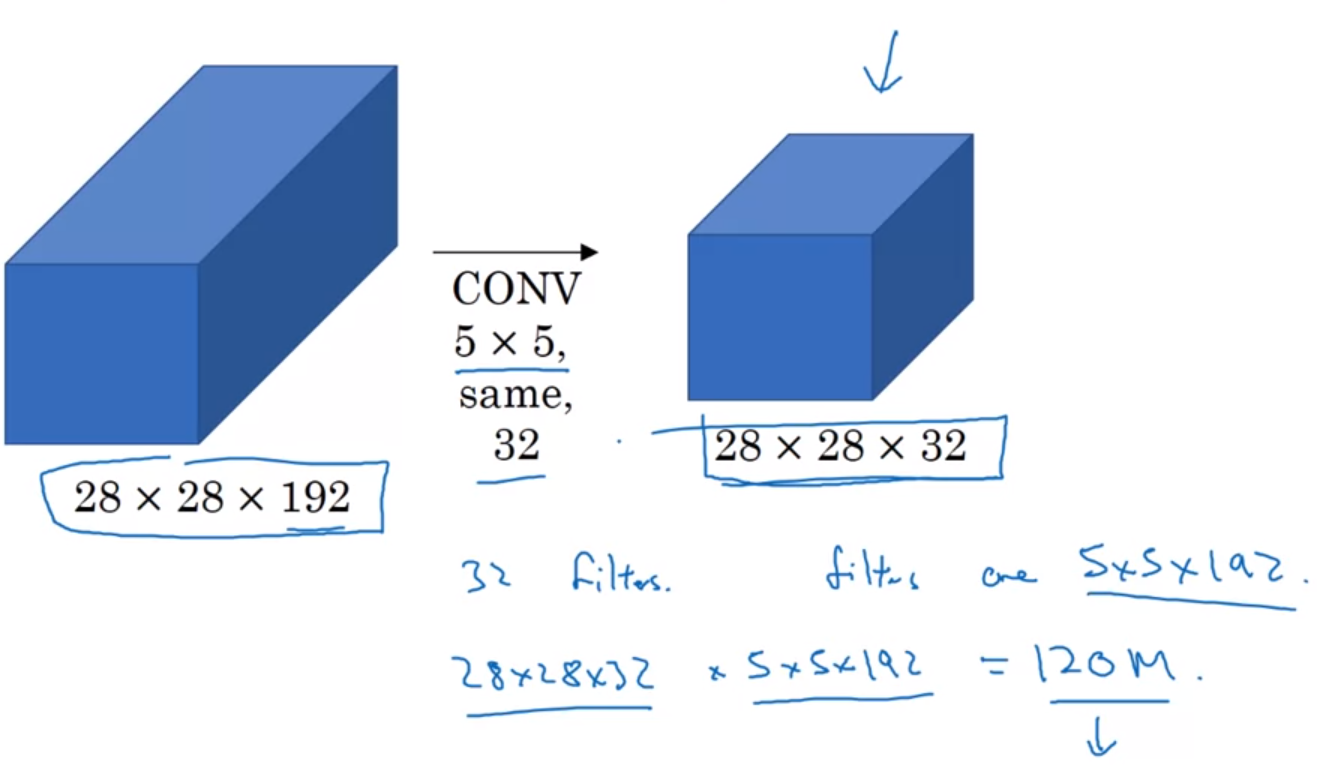
**Inception Network Motivation**

We don’t know every time which type of Convolution to use, or if we have to use conv layers of pooling ones; thus, this network uses inception modules which apply various conv and pool layers, and concatenate the feature maps in a single one

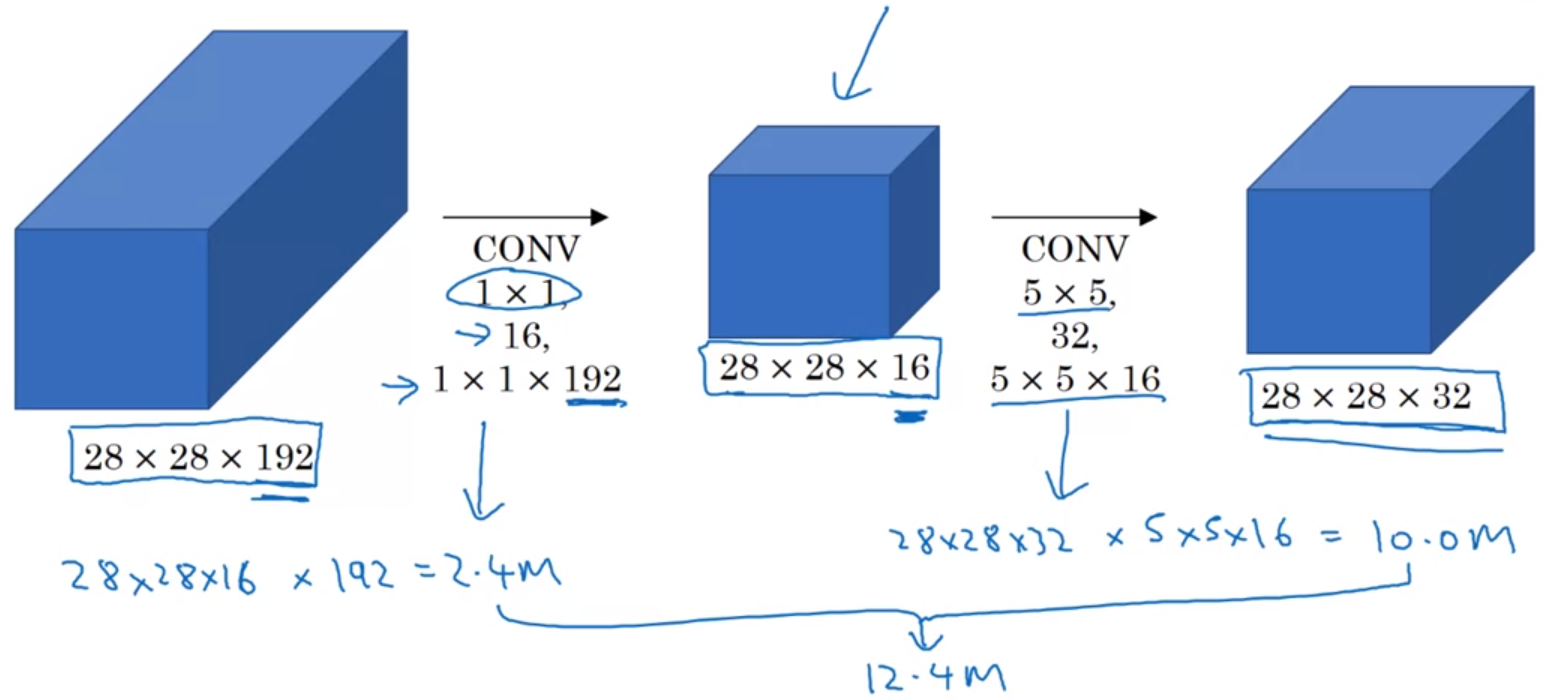


It worth noting that the max-pooling layer is used with padding and a stride of 1, so that each that the output size matches the shape of the input

There is a problem related to that fact that we use many operations: the computational cost

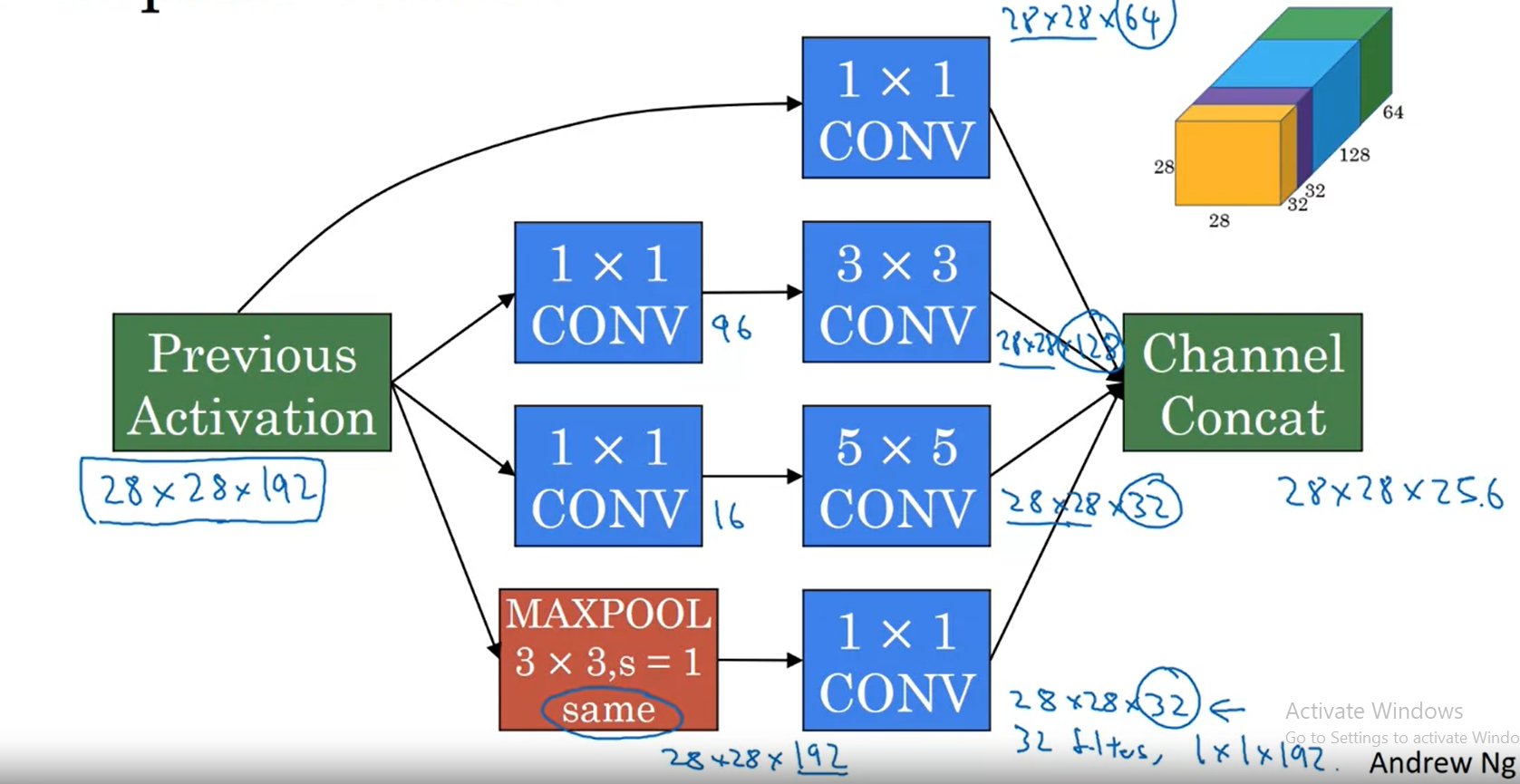
It’s expensive to apply a 5x5 conv directly on the input that has many channels; example: 

The solution is to use the 1x1 convolution before applying a 5x5 convolution. Because the number of channels is lower, the number of operations is significantly reduced; example:



**Inception Network**

It’s created by stacking many inception modules



Between these modules, there are some max pooling layers

It’s interesting that the network has 3 output parts formed by fully connected layers and softmax activation function. That’s because all the inputs are used to compute the loss; in this way the overfitting probability is reduced

This network has 4 versions and the first one is called GoogLeNet

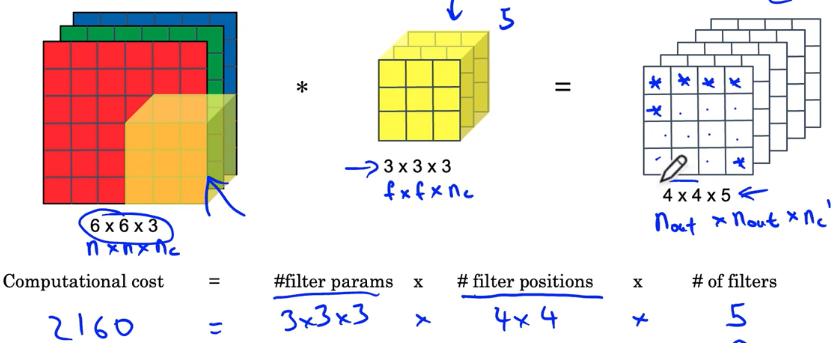
**MobileNet**

This network appeared to be used by mobile devices

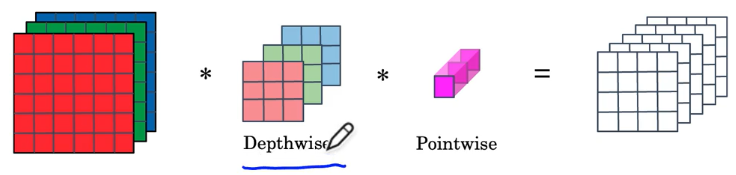
The trend was to create bigger and bigger networks, but there wan’t enough compute power to run them

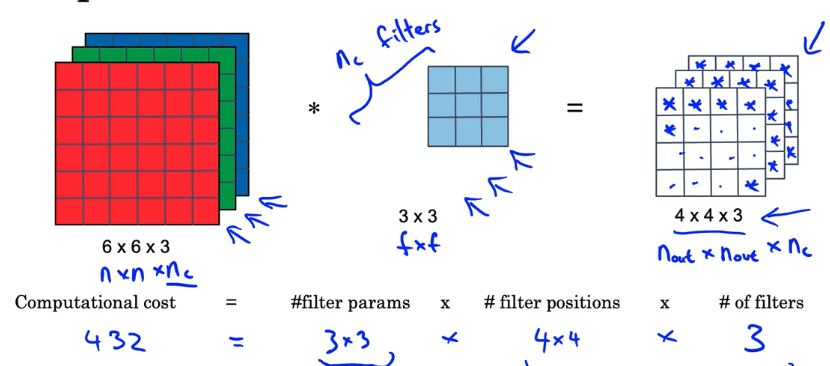
The key idea to reduce the number of operation was to use instead of normal convolutions, the depthwise-separable convolutions

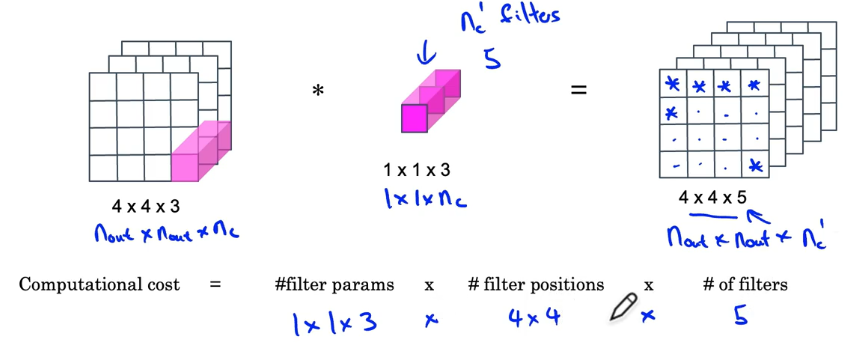
For a normal convolution, the computational cost = # filter params \* # filter positions \* # filters



The depthwise separable convolution reduces the computational cost because there each channel of the kernel is convoled with the corresponding channel of the input and then a pointwise convolutions (1x1 convolution) is used to generate as many output channles as desired (the number of channles of the pointwise convolution has to match the number of output channels)







The cost ratio between the cost of normal convolution and the cost of depthwise separable convolution can be computed by using the following formula:

1 / number of output channels + 1 / kernel size ^ 2

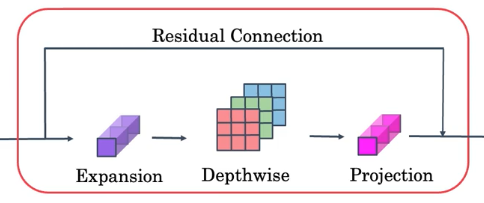
A typical cost reduction is around 3x, 10x and so on

The paper that describes this cost-efficient convolution was published in 2017

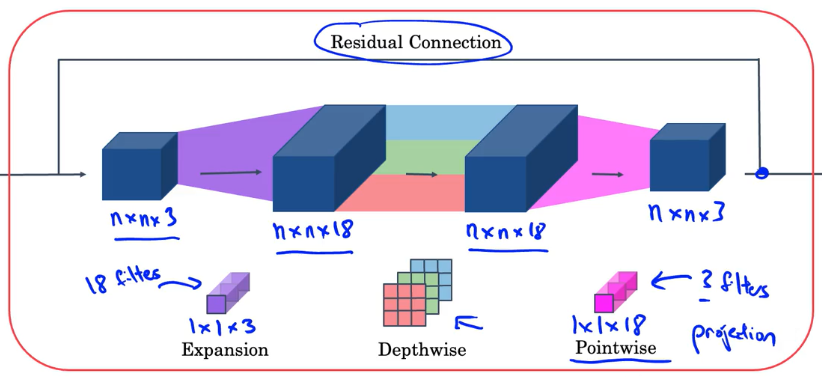
**MobileNet Architecture**

The MobileNet v1 has 13 blocks of depthwise separable convolution, followed by a pooling, a fully connected and a softmax layer

In 2019, MobileNet V2 introduced to the block the expansion and the residual connection



This block, called bottleneck, is repeated 17 times and is followed by a pooling, fully connected and a softmax layer



The idea of the expansion is to increase the number of channels by applying a 1x1 convolution

In this way, the depthwise convolution can extract more info from the data

However, a pointwise convolution reduces the number of channels to the initial value, so that the cost remain efficient

**EfficientNet**

Considering that the mobile devices have different compute power, the baseline networks have to be scaled up or down to match the device capabilities

EfficientNet solved this problem by finding a way to scale the image input size, the depth (number of layers) and the width of the network (number of channels)

They found that the best way is to scale these together, and to avoid the scaling of only the depth / width or input; the scaling should be done with a constant based on the FLOPS of the device

The optimum sizes were found by using NAS (neural architecture search) and improved the performance of known networks and in the same time reduced the number of parameters

**Using Open-Source Implementation**

A good option is to get open-source implementations from GitHub instead of developing them from scratch

**Transfer Learning**

If you don’t have an exceptionally large data set, to train a network from scratch, the best practice is to use transfer learning, which is extremelly adviced

There are 3 different cases:

If you have only a few images, you freeze all the layers and replace only the last layer with its activation function (e.g.: softmax) to match your classes; A trick is to store for each input image the output of the last frozen layer, and then to train a network that has as inputs the stored outputs; in this way you are training a shallow network

If you have a moderate number of images, you can freeze all layers, except the last 2-3 and to train these 2-3 layers and the output layer that now has the activation function modified to match your classes

If you have enough data, you can train the entire network, and to use the weights for initialization

**Data Augmentation**

Computer Vision models require a lot of data and it always seems that more data be useful

In order to increase the data set size, we can use augmentation techniques

1. mirroring, random cropping – widely used; rotation, shearing and local warping – less frequently used

2. color shifting – there is a more advanced technique compared to the random one – PCA color augmentation – described in the AlexNet paper; this technique modifies more the the channels with larger values and changes less the channels that aren’t that impactful in the image; in this way the overall tint of the image is kept

In practice, the images are taken from the hard disk and a CPU thread loads them and distorts them, creating mini-batches; these are used by another CPU thread / GPU for training

The data augmentation process has many hyper-parameters, exactly as a network, that’s why it’s a good idea to use the same techniques as used by researchers (so copy the hyper-parameters from open-source implementations)

**State of Computer Vision**

As there is more and more data, there are used simpler algorithms with less hand-engineering (for example for speech recognition)

However, as we have less data (image recognition or even object detection), there’s the need of using more hand-engineering; in this case the transfer learning technique is really suitable

There are two sources of knowledge: labeled data and hand engineered features / network architecture / other components; if there’s less data, we have to improve the other components

To do well on benchmark or in competitions, people:

1. use ensembling techniques: train 3 -15 networks and then average the outputs

2. multi-crop at test time: run the classifier on multiple versions of test images and average the results; a 10-crop technique is used: take the image and the vertically flipped image and from each you take 5 crops (each corner and the center)



Open source code:

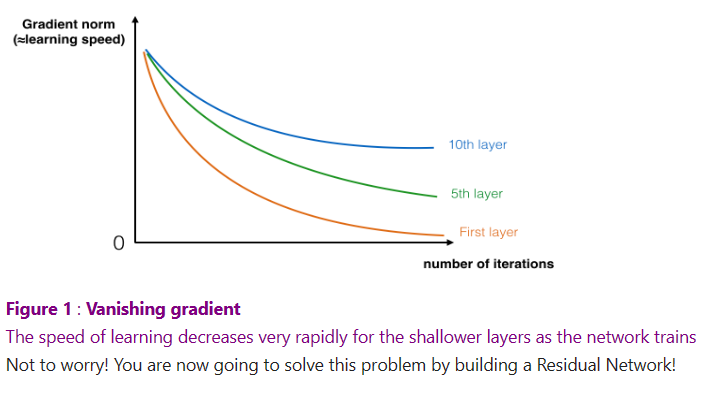
Use architectures of networks published in the literature

Use open source implementations if possible

Use pretrained models and fine-tune on your dataset

**Assignment 1**

You can find known developed models directly in tensorflow.keras.applications

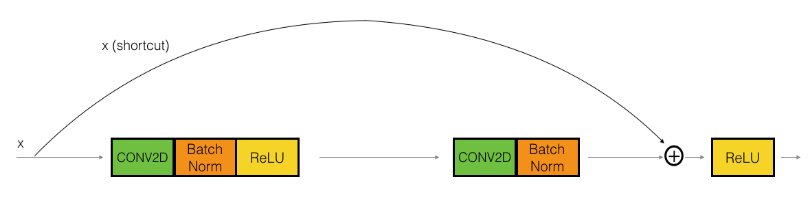
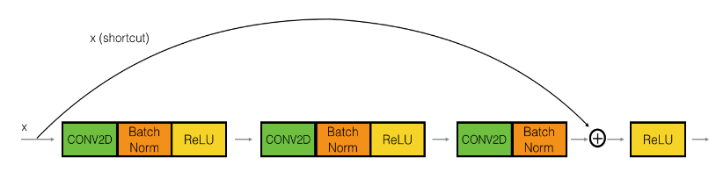


In deeper networks, as we go through the network, gradients are smaller and smaller (not the weights), thus the learning is slower

There are 2 main types of blocks that are used in ResNets: identity block and the convolutional block

In the identity block, the skip connection can “skip” over 2 or 3 layers

This is the standard block and it’s used when the input activation a[l] has the same shape as the a[l+2]



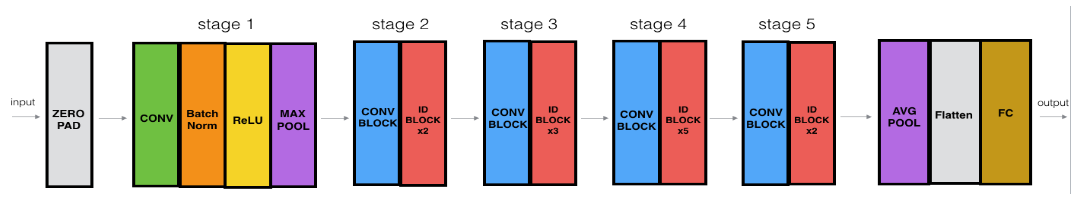
Inside a ResNet block we refer to the skip connection / shortcut and the main path

Batch normalization is applied across the channels

A convolutional block is used when the input and output dimensions aren’t the same

The difference is that on the shortcut there are a Conv2D and a BatchNorm to adjust the input size to match the output of the main path; the Conv2D on the shortcut path doesn’t have any non-linear activation function, its role being only to adjust the input size

Architecture of the ResNet-50:



It’s interesting that the pooling layer is used only in the beginning

**Assignment 2**

Load data from directory with image\_dataset\_from\_directory function and use the same seed for training and validation folds

To avoid data bottlenecks (the model to wait for data), we can prefetch it in an optimal way by using train\_dataset = train\_dataset.prefetch(buffer\_size=tf.data.experimental.AUTOTUNE)

The data augmentation is done with a simple Sequential model that has as layers the preprocessing operations such as RandomFlip and RandomRotation

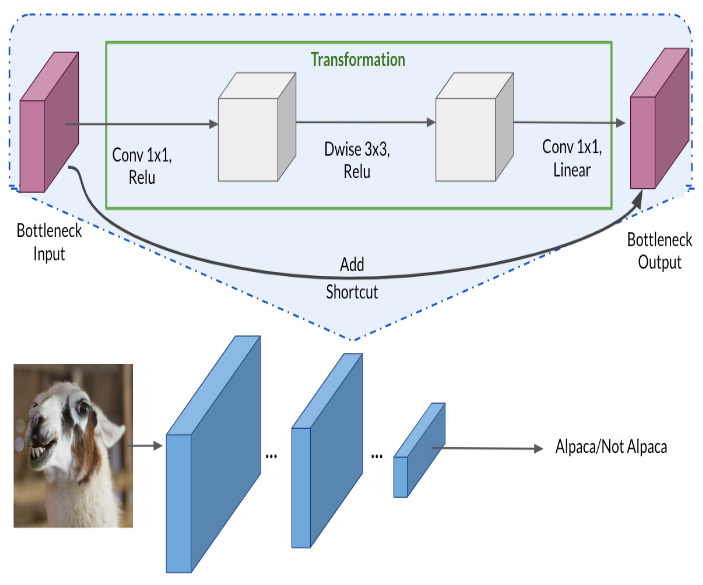
When using a pre-trained model you have to preprocess the data in the same way it was processed for pre-training (in TF there is a method that preprocessed the data in the same way)

MobileNetV2 has 155 layers and 3 main features:

- depthwise separable convolutions

- thing input and output bottlenecks between layers

- shortcut connections between bottleneck layers



When using a pretrained model form tf keras, you can specify:

- input shape, even though it’s different from the one presented in the original architecture

- if you want to include the top layers (the pooling and fully connected layers)

- the pre-trained weights, for example the ones from ImageNet

- the number of classes

Predict with the model:

base\_model(image\_batch)

Decode the outputs of the model with:

tf.keras.applications.mobilenet\_v2.decode\_predictions(pred.numpy(), top=2) (gives for each input the top 2 predictions and the attached probabilities)

Transfer learning - replace last layers – set the include\_top to false, add new final layers and freeze the base model (base\_model.trainable=False and base\_model(x, training=False))

Transfer learning – train the last layers – because the last layers learn finer details, we want to use their pre-trained weights as initialization and then to further learn from the current data, but we’ll use a smaller learning rate that yields better results than using a larger one

You can freeze individual layers by using base\_model.layers[index].trainable = False

You can train a model multiple times and resume the training each time. You have to use in the .fit() method the initial\_epoch = value of the new initial epoch (e.g.: 10, if initially you trained the model for 10 epochs 0 -> 9)