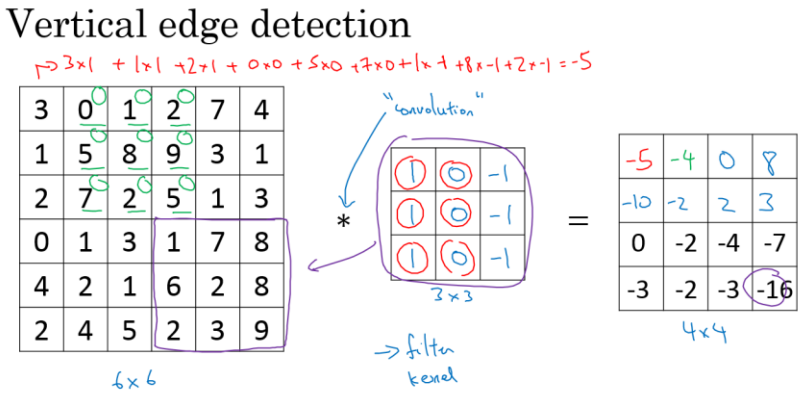
Understanding conv matrix-Edge detection

The convolution operation is one of the fundamentals blocks of a CNN. One of the examples about convolution is the image edge detection operation.

Early layers of CNN might detect edges then the middle layers will detect parts of objects and the later layers will put the these parts together to produce an output.

example of convolution operation to detect vertical edges:



- The vertical edge detection filter will find a 3x3 place in an image where there are a bright region followed by a dark region.
- If we applied this filter to a white region f ollowed by a dark region, it should find the edges in between the two colors as a positive value. But if we applied the same filter to a dark region followed by a white region it will give us negative values. To solve this we can use the abs function to make it positive.

Horizontal edge detection

Filter would be like: $\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$

Many different filtters for vertical:

Sobel filter (vertical):	**Scharr** filter
$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix}$

we don't need to hand craft these numbers, we can treat them as weights and then learn them. It can learn horizontal, vertical, angled, or any edge type automatically rather than getting them by hand.

Shrinking output calculation:

Input-(n,n) | filter-(f,f) | output-(n-f+1,n-f+1)

Padding

Problem:

When convoluting our output matrix shrinks(6x6 matrix convolved with 3x3->4x4 matrix). In order to to use deep neural networks we really need to use **paddings**(We want to apply convolution operation multiple times without getting (1,1) output matrix).

Solution:

To solve these problems we can pad the input image before convolution by adding some rows and columns to it. We will call the padding amount `P` the number of row/columns that we will insert in top, bottom, left and right of the image.

Size with padding:

Input-(n,n) | filter-(f,f) | output-(n+2p-f+1,n+2p-f+1)

To maintain the same size for input and output:

- `P = (f-1) / 2`

Strided convolution

When we are making the convolution operation we used `S(strides)` to tell us the number of pixels we will jump when we are convolving filter/kernel. The last examples we described S was 1.

Size with padding and strides:

Input-(n,n) | filter-(f,f) | strides=s | output-$\left(\frac{n+2p-f}{s}+1, \frac{n+2p-f}{s}+1\right)$

Note: In math textbooks the conv operation is filpping the filter before using it. What we were doing is called cross-correlation operation but the state of art of deep learning is using this as conv operation.

Same convolutions:

convolution with a padding so that output size is the same as the input size.

$$P = \frac{ns - n + f - s}{2}$$

Convolutions over volumes

We will convolve an image of height, width, # of channels with a filter of a height, width, same # of channels. Hint that the image number channels and the filter number of channels are the same.

Example one filter over RGB image:

- Input image: `6x6x3`
- Filter: `3x3x3`
- Result image: `4x4x1|p=0,s=1`

Example many filter over RGB image:

- Input image: `6x6x3`
- 10 Filters: `3x3x3`
- Result image: `4x4x10|p=0,s=1`

One Layer of a Convolutional Network

Steps(p=0, s=1):

1. Input image: (6,6,3) # a0 | 10 Filters: (3,3,3) #W[1]
2. Result image: (4,4,10) #W[1]*a0
3. Add b (bias)- (10,1) ->#z[1] (4,4,10)= W[1]a0 + b
4. Apply RELU #A1(4,4,10) = RELU(W1a0 + b)

number of parameters: #w+b=(3x3x3x10) + 10 = 280

Notations and sizes in a conv layer:

```
f[l] = filter size
p[l] = padding        # Default is zero
s[l] = stride
nc[l] = number of filters

Input:  (nH[l-1] , nW[l-1] ,nc[l-1])
Output: (nH[l], nW[l] , nc[l])
Where n_H = (n_H+2p[l]−f[l])/s[l] + 1

Each filter is: (f[l], f[l], nc[l-1])

Activations: a[l]= (nH[l], nW[l], nc[l])
             A[l]= (m, nH[l], nW[l], nc[l])#batch mode

Weights: ((f[l], f[l]), nc[l-1]-number of channels, nc[l]-number filters)

bias:  (1, 1, 1, nc[l])
```

A simple convolution network example

Types of layer in a convolutional network:

- Convolution. #Conv
- Pooling #Pool
- Fully connected #FC

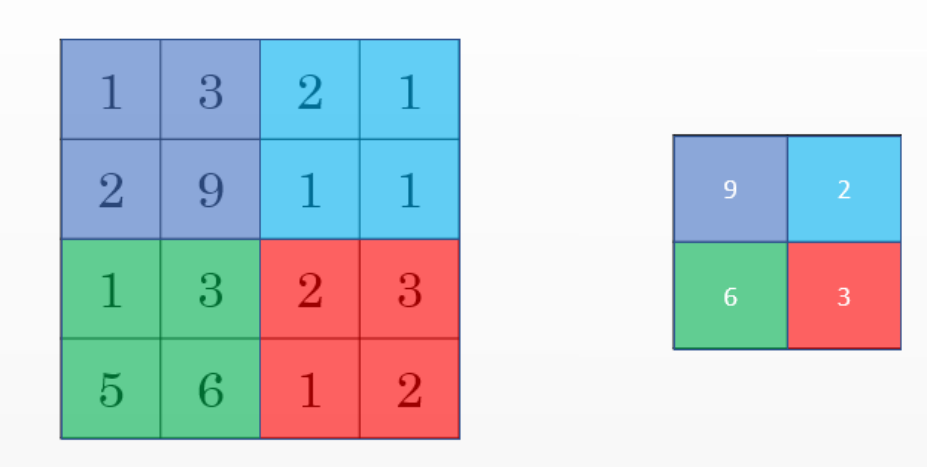
Lets build a big example:

- Input Image are: `a0 = (39,39,3)` | `n0 = 39 ,nc0 = 3`
- First layer (Conv layer):
  - `f1 = 3, s1 = 1,` and `p1 = 0 ,number of filters = 10`
  - Then output are `a1 =(37,37,10)`| `n1 = 37 , nc1 = 10`
- Second layer (Conv layer):
  - `f2 = 5, s2 = 2, p2 = 0, number of filters = 20`
  - The output are `a2 = (17,17,20)` | `n2 = 17, nc2 = 20`
- Third layer (Conv layer):
  - `f3 = 5, s3 = 2, p3 = 0, number of filters = 40`
  - The output are `a3 = (7,7,40)` | `n3 = 7, nc3 = 40`
- Forth layer (Fully connected Softmax)
  - `a3 = 7,7,40 = 1960` as a vector..

Pooling layers #pool

Other than the conv layers, CNNs often uses pooling layers to reduce the size of the inputs, speed up computation, and to make some of the features it detects more robust.

- Max pooling example:



*This example has `f = 2, s = 2,` and `p = 0` hyperparameters

Max pooling -keep the highest number in the region. Max pooling has no parameters to learn.

Average pooling-taking the averages of the values instead of taking the max values.

Example of Max pooling on 3D input:

Input: `(4,4,10)` → `Max pooling size = 2 ,stride = 2` → Output: `2x2x10`

Why convolutions?

- Two main advantages of Convs are:
  - Parameter sharing.
    - A feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image.
  - sparsity of connections.
    - In each layer, each output value depends only on a small number of inputs which makes it translation invariance.
  - Fully connected layers has a lot of parameters

Example: like LeNet-5 [Yann Lecun]

	Activation shape	Activation Size	# parameters
Input:	(32,32,3)	3,072 $a^{[0]}$	0
CONV1 (f=5, s=1)	(28,28,8)	6,272	208
POOL1	(14,14,8)	1,568	0
CONV2 (f=5, s=1)	(10,10,16)	1,600	416
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48,001
FC4	(84,1)	84	10,081
Softmax	(10,1)	10	841

- Usually the input size decreases over layers while the number of filters increases.
- **Fully connected layers has the most parameters in the network.**

Deep convolutional models: case studies

Learn about the practical tricks and methods used in deep CNNs straight from the research papers.

Why look at case studies?

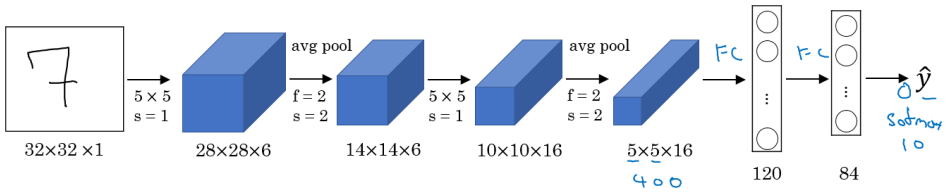
- We learned about Conv layer, pooling layer, and fully connected layers. It turns out that computer vision researchers spent the past few years on how to put these layers together.
- To get some intuitions you have to see the examples that has been made.
- Some neural networks architecture that works well in some tasks can also work well in other tasks.
- Here are some classical CNN networks:
    - **LeNet-5**
    - **AlexNet**
    - **VGG**
- The best CNN architecture that won the last ImageNet competition is called **ResNet** and it has 152 layers!
- There are also an architecture called **Inception** that was made by Google that are very useful to learn and apply to your tasks.
- Reading and trying the mentioned models can boost you and give you a lot of ideas to solve your task.

Classic networks

LeNet-5(published in 1998)

[LeCun et al., 1998. Gradient-based learning applied to document recognition]

The goal for this model was to identify handwritten digits in a `32x32x1` gray image. Here are the drawing of it:



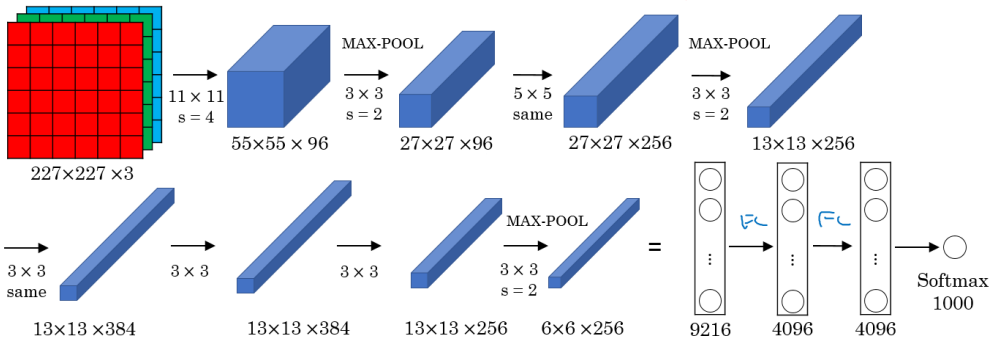
- The dimensions of the image decreases as the number of channels increases.
- `Conv ==> Pool ==> Conv ==> Pool ==> FC ==> FC ==> softmax` this type of arrangement is quite common.

.

AlexNet(published in 2012)

[Krizhevsky et al., 2012. ImageNet classification with deep convolutional neural networks]

The goal for the model was the ImageNet challenge which classifies images into 1000 classes. Here are the drawing of the model:



Summary:

- `Conv => Max-pool => Conv => Max-pool => Conv => Conv => Conv => Max-pool ==> Flatten ==> FC ==> FC ==> Softmax`

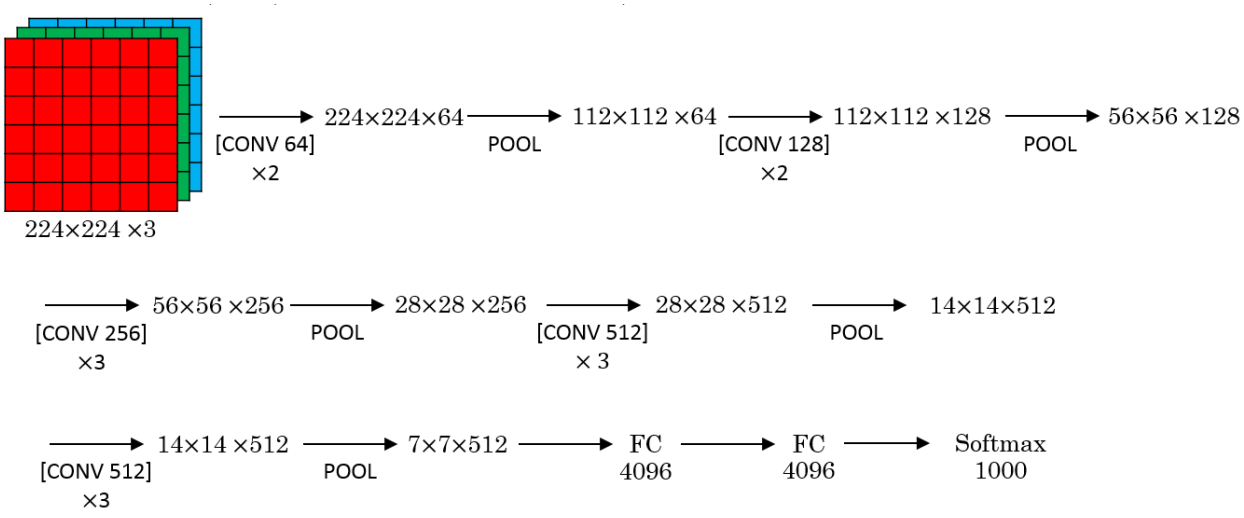
Notes:

- Similar to LeNet-5 but bigger.

- Has **60 Million parameter** compared to 60k parameter of LeNet-5.
- It used the RELU activation function.
- The original paper contains Multiple GPUs and Local Response normalization (RN).
    - Multiple GPUs were used because the GPUs were not so fast back then.
    - Researchers proved that Local Response normalization doesn't help much so for now don't bother yourself for understanding or implementing it.
- This paper convinced the computer vision researchers that deep learning is so important.

VGG-16(published in 2015)

[Simonyan & Zisserman 2015. Very deep convolutional networks for large-scale image recognition]



Properties:

- Focus on having only these blocks repeating:

    CONV = 3 X 3 filter, s = 1, same

MAX-POOL = 2 X 2 , s = 2

- A modification for AlexNet.
- Number of filters increases from 64 to 128 to 256 to 512. 512 was made twice
- Pooling was the only one who is responsible for shrinking the dimensions.
- It has a total memory of 96MB per image for only forward propagation!
- This network is large even by modern standards. It has around **138 million parameters**
- VGG paper is attractive it tries to make some rules regarding using CNNs.
- Most of the parameters are in the fully connected layers.
- Most memory are in the earlier layers.

Notes:

- There are another version called **VGG-19** which is a bigger version. But most people uses the VGG-16 instead of the VGG-19 because it does the same.

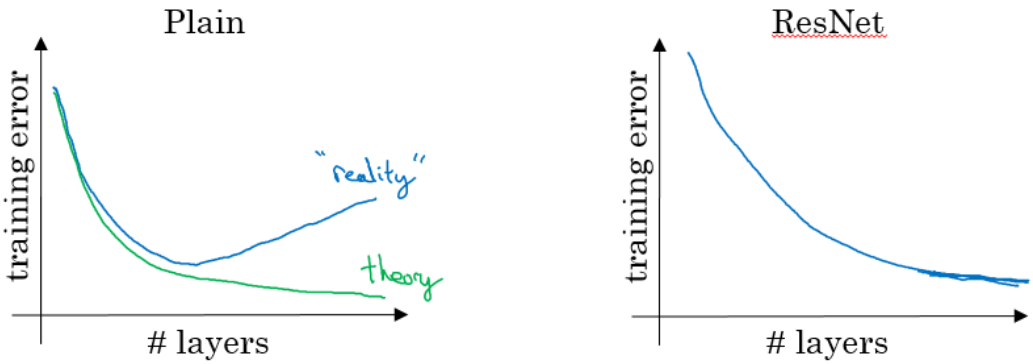
Residual Networks (ResNets)[published in 2015]

[He et al., 2015. Deep residual networks for image recognition]

Skip connection:

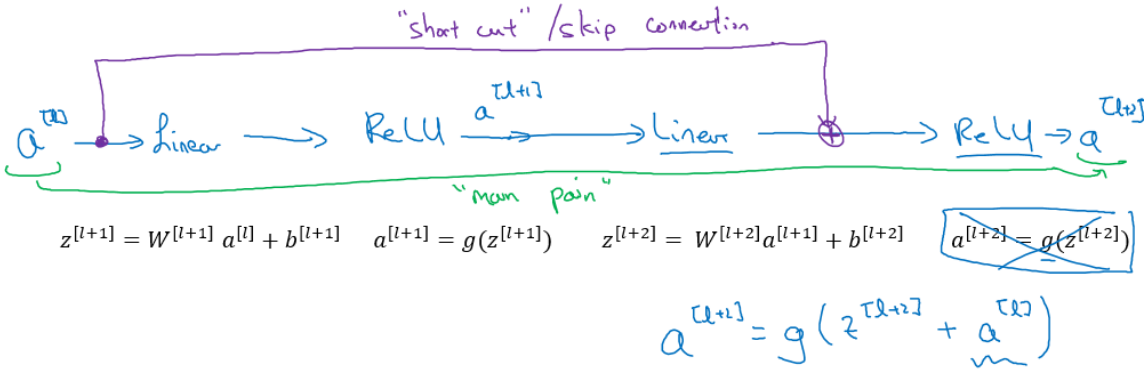
Why?

Very, very deep NNs are difficult to train because of vanishing and exploding gradients problems.



skip connection which makes you take the activation from one layer and suddenly feed it to another layer even much deeper in NN which allows you to train large NNs even with layers greater than 100.

How?



Why it is at least better?

`a[l] --> Layer1 --> Layer2 --> a[l+2]` | `a[l]` has a direct connection to `a[l+2]`

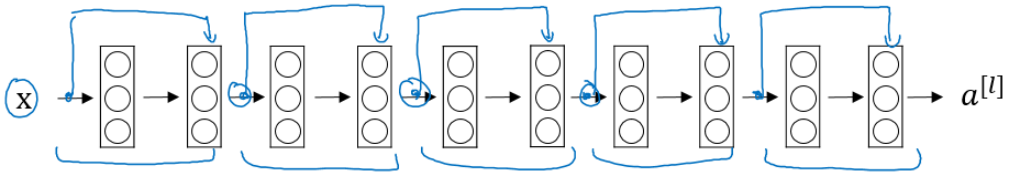
```
a[l+2] = g(z[l+2] + a[l] ) = g( W[l+2] a[l+1] + b[l+2] + a[l] )
```

We can take b[l+2] and W[l+2] to 0 and get same result as like not have connections:
T `a[l+2] = g( a[l] ) = a[l]`

This show that identity function is easy for a residual block to learn. And that why it can train deeper NNs.

Residual Network

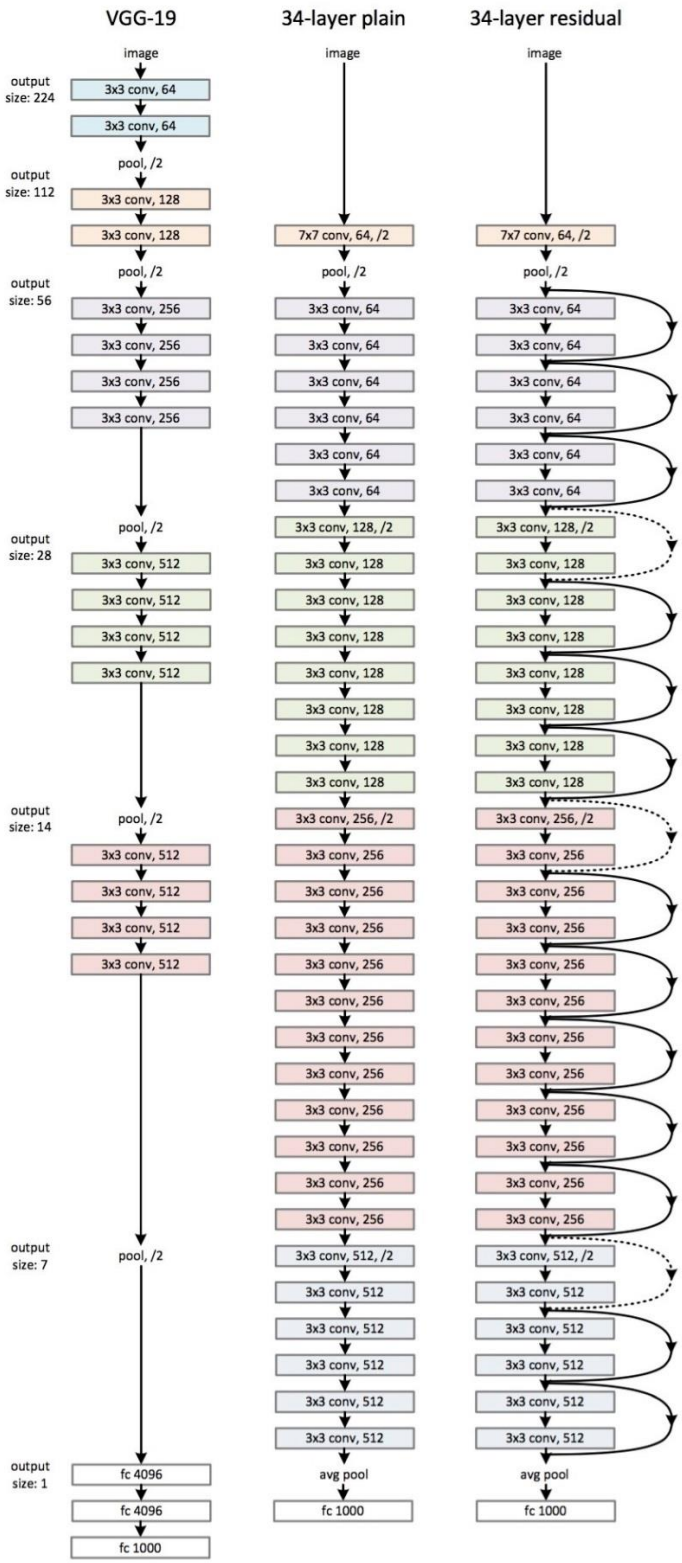
Are a NN that consists of some Residual blocks.



Properties:

- These networks can go deeper without hurting the performance. In the normal NN - Plain networks - the theory tell us that if we go deeper we will get a better solution to our problem, but because of the vanishing and exploding gradients problems the performance of the network suffers as it goes deeper. Thanks to Residual Network we can go deeper as we want now.
- On the left is the normal NN and on the right are the ResNet. As you can see the performance of ResNet increases as the network goes deeper.
- In some cases going deeper won't effect the performance and that depends on the problem on your hand.
- Some people are trying to train 1000 layer now which isn't used in practice.

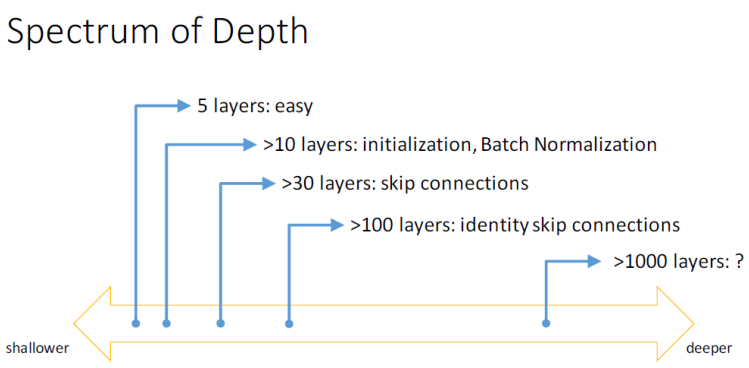
ResNets 34| VGG-19 architecture:



Notes:

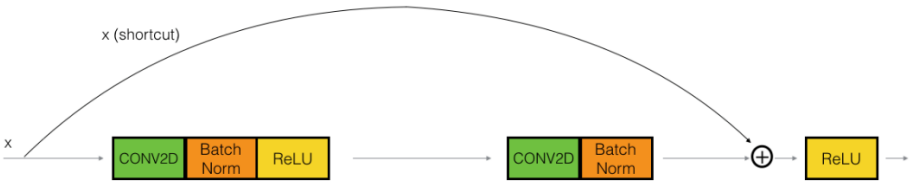
- All the 3x3 Conv are same Convs.
- Keep it simple in design of the network.
- spatial size /2 => # filters x2
- No FC layers, No dropout is used.
- Two main types of blocks are used in a ResNet, depending mainly on whether the input/output dimensions are same or different. You are going to implement both of them.
- The dotted lines is the case when the dimensions are different. To solve then they down-sample the input by 2 and then pad zeros to match the two dimensions. There's another trick which is called bottleneck which we will explore later.

- Useful concept (**Spectrum of Depth**):



The convolutional ResNet block:

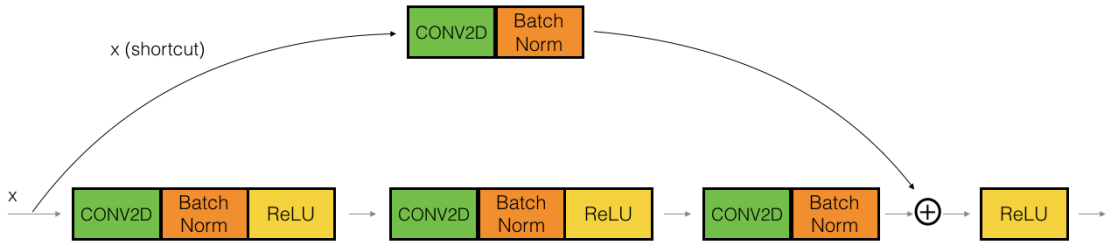
(1) None convolutional block:



- Hint the conv is followed by a batch norm BN before RELU. Dimensions here are same.

- This skip is over 2 layers. The skip connection can jump n connections where n>2

(2) The convolutional block:



The conv can be bottleneck 1 x 1 conv

Network in Network and 1 X 1 convolutions

[Lin et al., 2013. Network in network]

What does a 1 X 1 convolution do? Isn't it just multiplying by a number?

Lets first consider an example:

- Input: `6x6x1`
- Conv: `1x1x1` one filter. `# The 1 x 1 Conv`
- Output: `6x6x1`

Another example:

- Input: `6x6x32`
- Conv: `1x1x32` 5 filters. `# The 1 x 1 Conv`
- Output: `6x6x5`

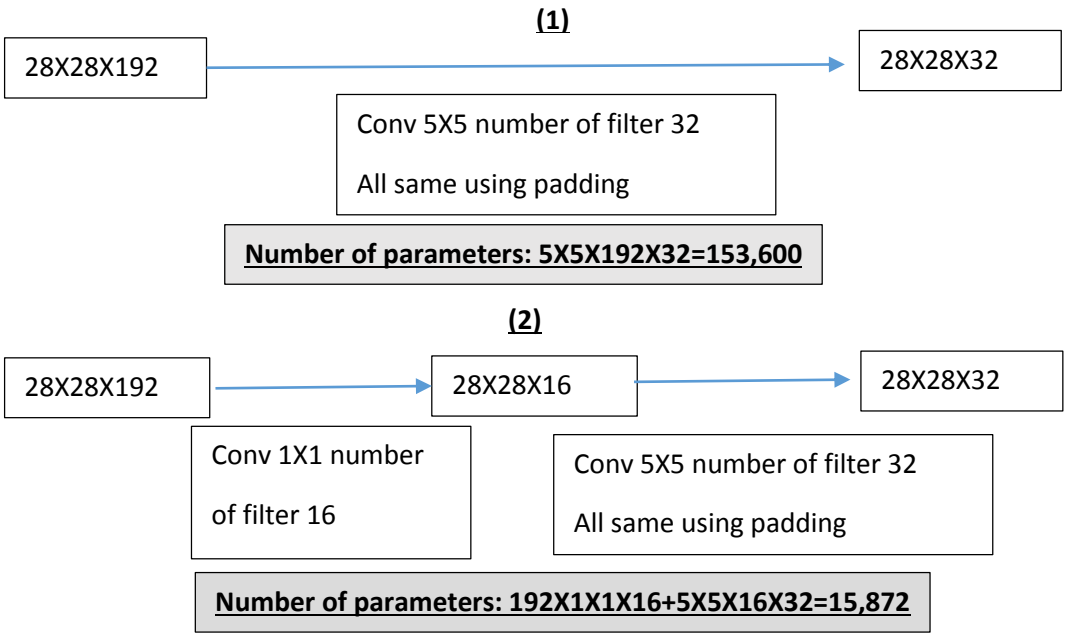
A 1 x 1 convolution is useful when:

- We want to shrink the number of channels. We also call this feature transformation(32 to 5 channels in second example).

- The 1 x 1 Conv will add non linearity and will learn non linearity operator(if we put relu or other non linear function after the filter). This is a smart way to make the net deeper without adding a lot of parameters.
- The 1X1 filter act similar to a perceptron(the number of filter are equivalent to the number of Hidden units) on the depth of the input. Example input 6X6X4 filter 1X1X3:

  Output cell 1x1x1=(1x1x1*w1)+(1x1x2*w2)+(1x1x3*w3)

- reduce computational cost example:



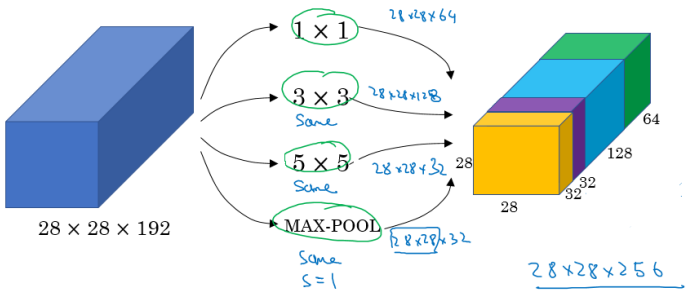
Reduce number of parameter in Order of magnitude

Inception network:

[Szegedy et al. 2014. Going deeper with convolutions]

Motivation: When you design a CNN you have to decide all the layers yourself. Will you pick a 3 x 3 Conv or 5 x 5 Conv or maybe a max pooling layer. You have so many choices.

Practice: Why not use all of them at once?

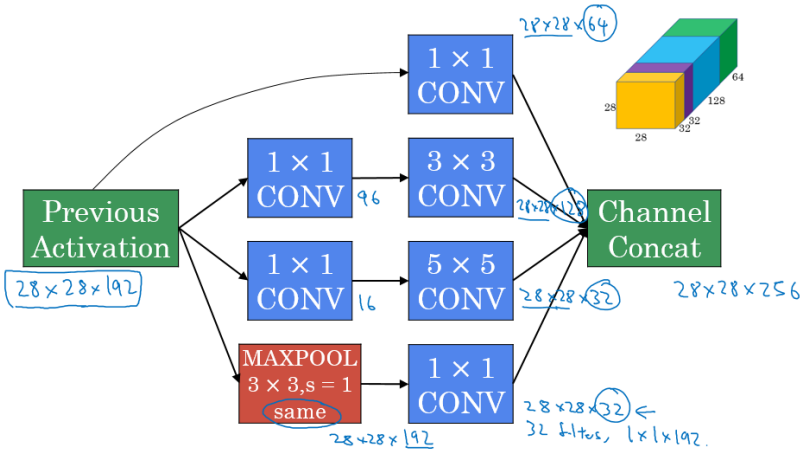


- Input to the inception module are 28 x 28 x 192 and the output are 28 x 28 x 256
- We have done all the Convs and pools we might want and will let the NN learn and decide which it want to use most.

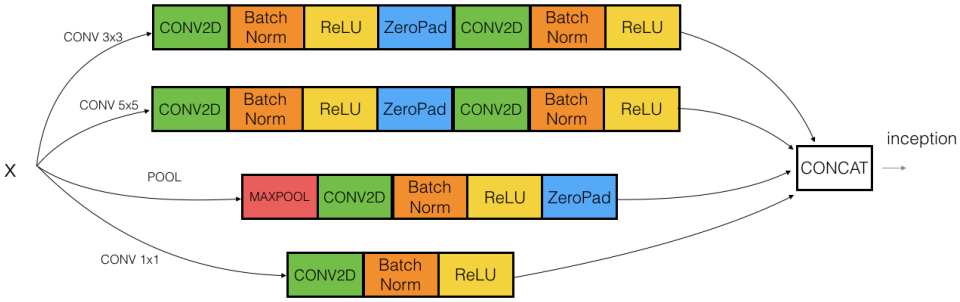
problem of computational cost in Inception model:

- The total number of multiplications needed here are:
  - Number of outputs * Filter size * Filter size * Input dimensions
  - Which equals: `28 * 28 * 32 * 5 * 5 * 192 = 120 Mil`
  - 120 Mil multiply operation still a problem in the modern day computers.
- Using a 1 x 1 convolution we can reduce 120 mil to just 12 mil. Lets see how.

- **Inception module**, dimensions reduction version:

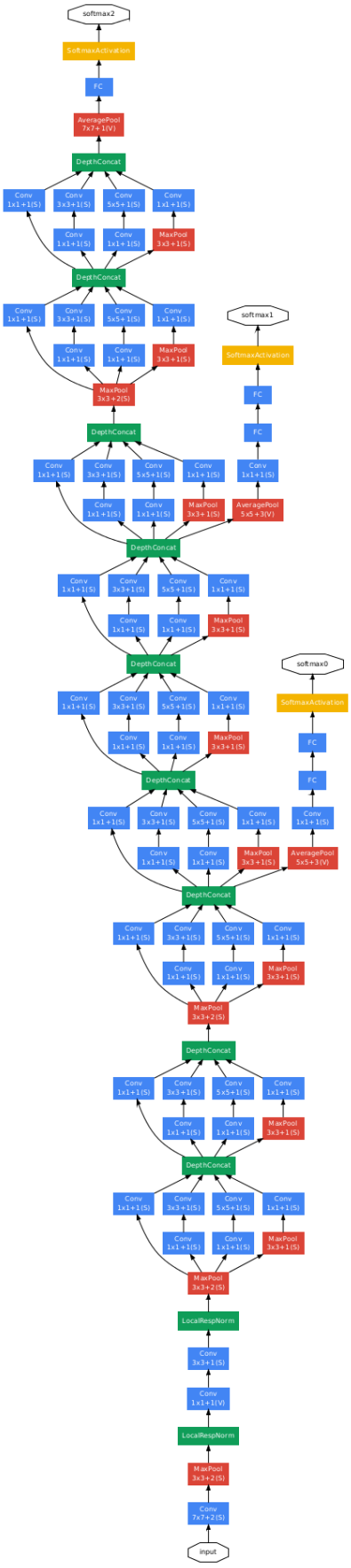


- Example of inception model in Keras:



Inception network (GoogleNet)

- The inception network consist of concatenated blocks of the Inception module.
- The name inception was taken from a *meme* image which was taken from **Inception movie**
- Here are the full model:


softmax2
SoftmaxActivation
FC
AveragePool 7x7+1(V)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
MaxPool 3x3+1(S)
MaxPool 3x3+2(S)
softmax1
AveragePool 5x5+3(V)
softmax0
LocalRespNorm
Conv 7x7+2(S)
Conv 1x1+1(V)
input

Notes:

- Sometimes a Max-Pool block is used before the inception module to reduce the dimensions of the inputs.
- There are a 3 Sofmax branches at different positions to push the network toward its goal. and helps to ensure that the intermediate features are good enough to the network to learn and it turns out that softmax0 and sofmax1 gives regularization effect.
- Since the development of the Inception module, the authors and the others have built another versions of this network. Like inception v2, v3, and v4. Also there is a network that has used the inception module and the ResNet together.

Using Open-Source Implementation

- We have learned a lot of NNs and ConvNets architectures.
- It turns out that a lot of these NN are difficult to replicated. because there are some details that may not presented on its papers. There are some other reasons like:
    - Learning decay.
    - Parameter tuning.
- A lot of deep learning researchers are opening sourcing their code into Internet on sites like Github.
- If you see a research paper and you want to build over it, the first thing you should do is to look for an open source implementation for this paper.
- Some advantage of doing this is that you might download the network implementation along with its parameters/weights. The author might have used multiple GPUs and spent some weeks to reach this result and its right in front of you after you download it.

Transfer Learning

- If you are using a specific NN architecture that has been trained before, you can use this pretrained parameters/weights instead of random initialization to solve your problem.
- It can help you boost the performance of the NN.
- The pretrained models might have trained on a large datasets like ImageNet, Ms COCO, or pascal and took a lot of time to learn those

parameters/weights with optimized hyperparameters. This can save you a lot of time.

example:

Lets say you have a cat classification problem which contains 3 classes Tigger, Misty and neither.You don't have much a lot of data to train a NN on these images.

Andrew recommends to go online and download a good NN with its weights, remove the softmax activation layer and put your own one and make the network learn only the new layer while other layer weights are fixed/frozen.

Frameworks have options to make the parameters frozen in some layers using `trainable = 0` or `freeze = 0`

One of the tricks that can speed up your training, is to run the pretrained NN without final softmax layer and get an intermediate representation of your images and save them to disk. And then use these representation to a shallow NN network. This can save you the time needed to run an image through all the layers.

Its like converting your images into vectors.

Another example:

- If you have enough data, you can fine tune all the layers in your pretrained network but don't random initialize the parameters, leave the learned parameters as it is and learn from there.

Data Augmentation

If data is increased, your deep NN will perform better. Data augmentation is one of the techniques that deep learning uses to increase the performance of deep NN.

data augmentation methods that are used for computer vision tasks includes:

- Mirroring.
- Random cropping.

        - The issue with this technique is that you might take a wrong crop.
        - The solution is to make your crops big enough.
    - Rotation.
    - Shearing.
    - Local warping.
    - Color shifting.
        - For example, we add to R, G, and B some distortions that will make the image identified as the same for the human but is different for the computer.
        - In practice the added value are pulled from some probability distribution and these shifts are some small.
        - Makes your algorithm more robust in changing colors in images.
        - There are an algorithm which is called ***PCA color augmentation*** that decides the shifts needed automatically.

Note:

- You can use a different CPU thread to make you a distorted mini batches while you are training your NN.

Data Augmentation has also some hyperparameters. A good place to start is to find an open source data augmentation implementation and then use it or fine tune these hyperparameters. Scikit-image library has some augmentation tool.

Computer Vision insights

- If your problem has a large amount of data, researchers are tend to use:
    - Simpler algorithms.
    - Less hand engineering.
- If you don't have that much data people tend to try more hand engineering for the problem "Hacks". Like choosing a more complex NN architecture.
- Because we haven't got that much data in a lot of computer vision problems, it relies a lot on hand engineering.
- We will see in the next chapter that because the object detection has less data, a more complex NN architectures will be presented.

Tips for competition in Computer Vision

- Tips for doing well on benchmarks/winning competitions:
    - ensembles.
        - Train several networks independently and average their outputs. Merging down some classifiers.
        - After you decide the best architecture for your problem, initialize some of that randomly and train them independently.
        - This can give you a push by 2%
        - But this will slow down your production by the number of the ensembles. Also it takes more memory as it saves all the models in the memory.
        - People use this in competitions but few uses this in a real production.
    - Multi-crop at test time.
        - Run classifier on multiple versions of test versions and average results.
        - There is a technique called 10 crops that uses this.
        - This can give you a better result in the production.
- Use 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.