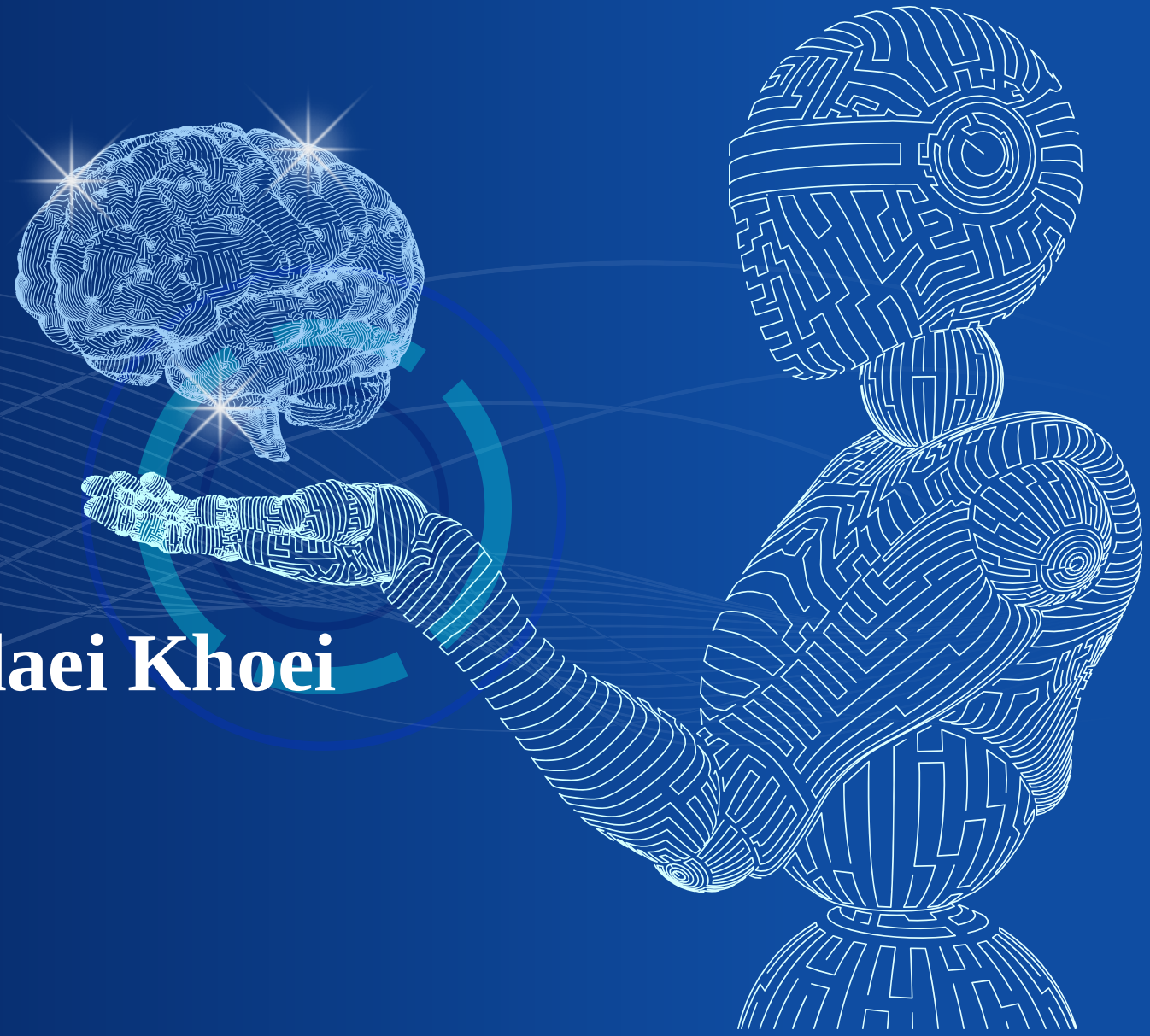
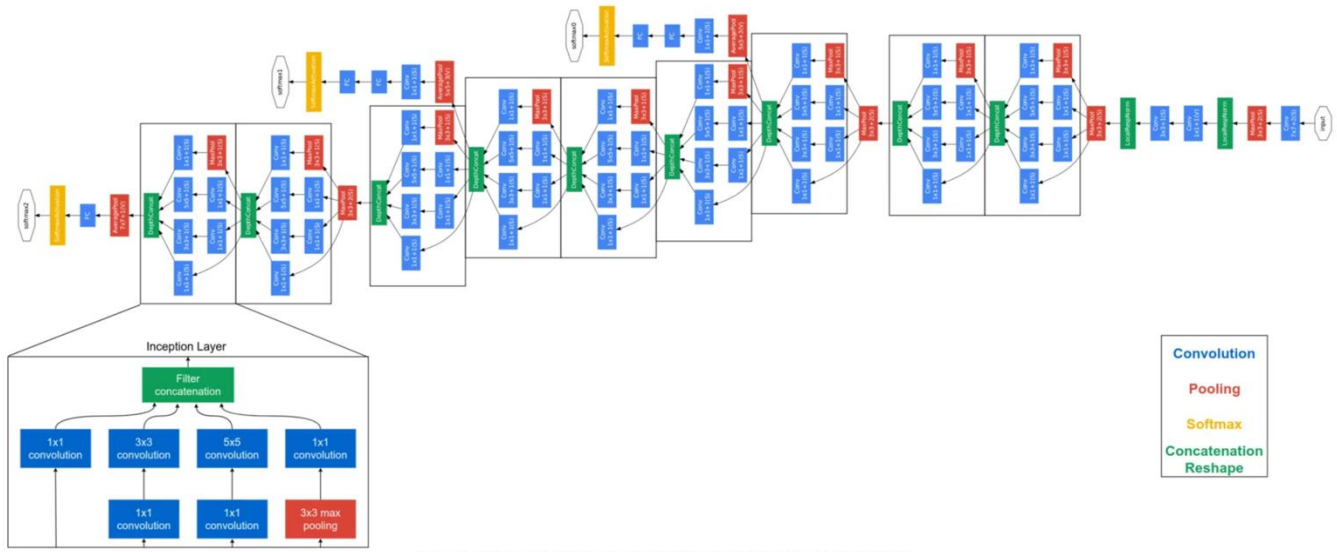
CS 7150: Deep Learning



Presented by: Dr. Tala Talaei Khoei

Introduction to GoogLeNet

- The paper Going Deeper with Convolutions introduces the first version of Inception model called GoogLeNet.
- It has around 6.7977 million parameters (without auxilaries layers) which is 9x fewer than AlexNet) and 20x fewer than its competitor VGG-16.
- In most of the standard network architectures, the intuition is not clear why and when to perform the max-pooling operation, when to use the convolutional operation. For example, in AlextNet we have the convolutional operation and max-pooling operation following each other whereas in VGGNet, we have 3 convolutional operations in a row and then 1 max-pooling layer.
- Thus, **the idea behind GoogLeNet is to use all the operations at the same time**. It computes multiple kernels of different size over the same input map in parallel, concatenating their results into a single output. This is called an **Inception module**.



- Suppose we have an image with a dog can be either of the following, as shown below. The area occupied by the dog is different in each image.

From left: A dog occupying most of the image, a dog occupying a part of it, and a dog occupying very little space (Images obtained from Unsplash).

- Because of this huge variation in the location of the information, choosing the **right kernel size** for the convolution operation becomes tough. A **larger kernel** is preferred for information that is distributed more **globally**, and a **smaller kernel** is preferred for information that is distributed more **locally.**

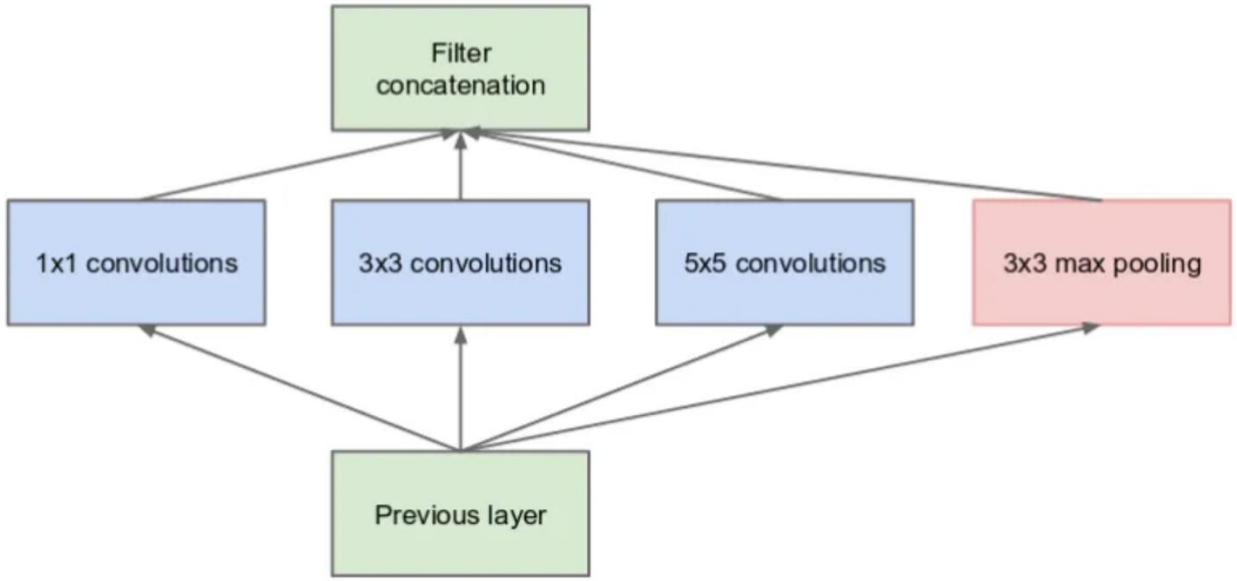
- Very deep networks are prone to overfitting. It also hard to pass gradient updates through the entire network.

- Naively stacking large convolution operations is computationally expensive.

Solution

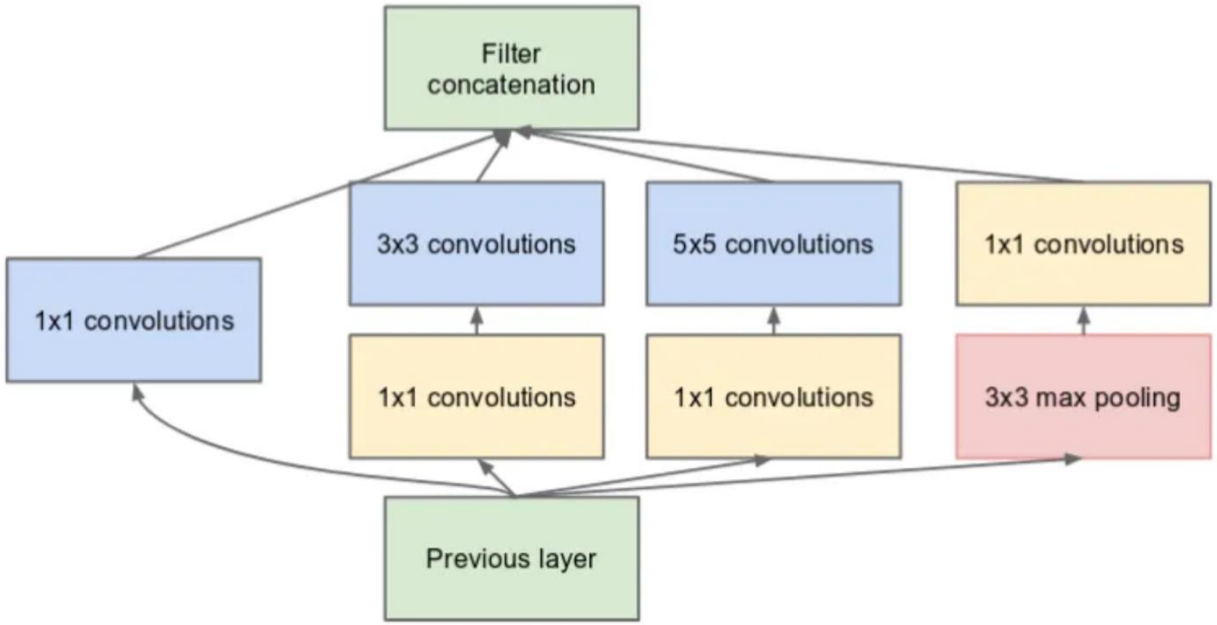
- Why not have filters with **multiple sizes** operate on the **same level**? The network essentially would get a bit "**wider**" rather than "deeper". The authors designed the inception module to reflect the same.

- The below image is the "naive" inception module. It performs **convolution** on an input, with **3 different sizes of filters** (1x1, 3x3, 5x5). Additionally, **max pooling** is also performed. The outputs are **concatenated** and sent to the next inception module.



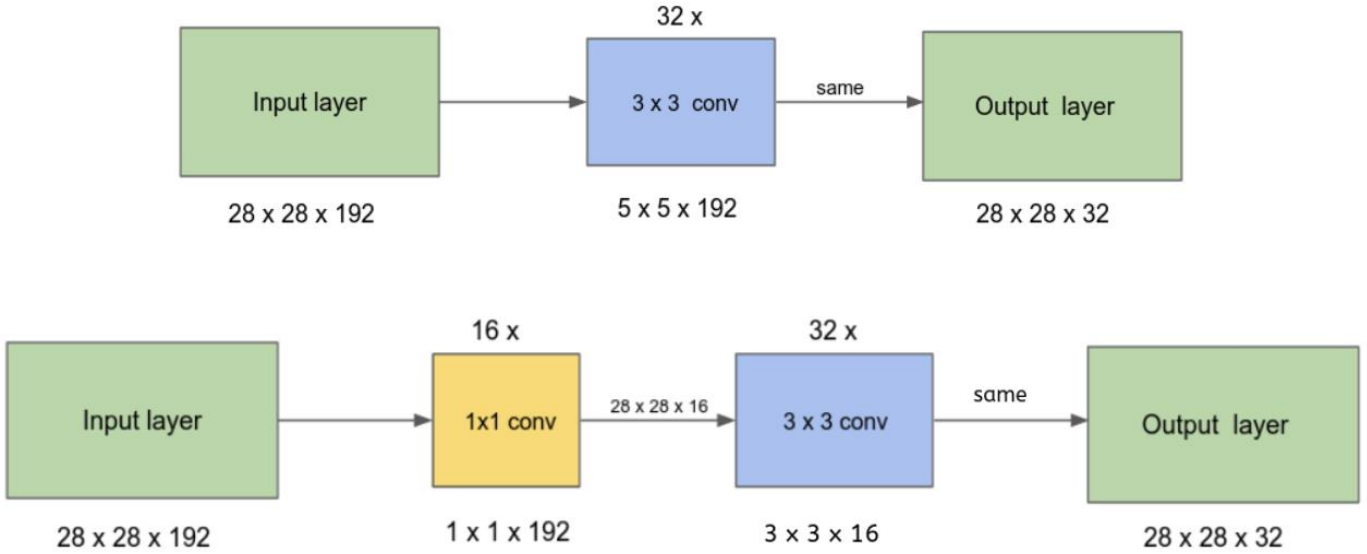
(a) Inception module, naïve version

- Deep neural networks are **computationally expensive**. To make it cheaper, the authors **limit** the number of **input channels** by adding an **extra 1x1 convolution** before the 3x3 and 5x5 convolutions. Though adding an extra operation may seem counterintuitive, 1x1 convolutions are far more cheaper than 5x5 convolutions, and the reduced number of input channels also help. Do note that however, the 1x1 convolution is introduced after the max pooling layer, rather than before.



(b) Inception module with dimension reductions

- Consider the following:

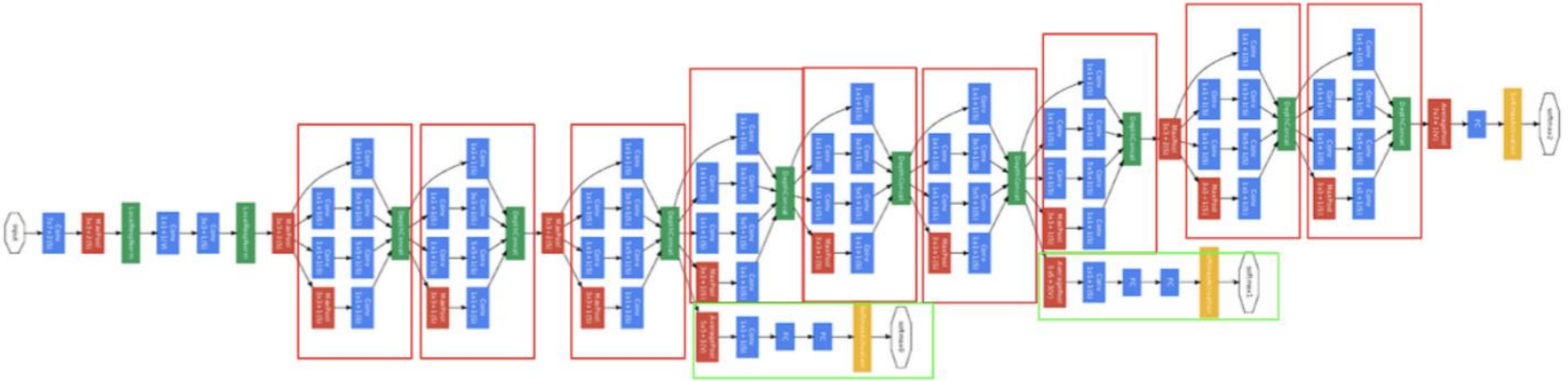


The Naive approach is computationally expensive:

- Computation cost = ((28 x 28 x 5 x 5) x 192) x 32 ≃ **120 Mil**
- We perform (28 x 28 x 5 x 5) operations along 192 channels for each of the 32 filters.

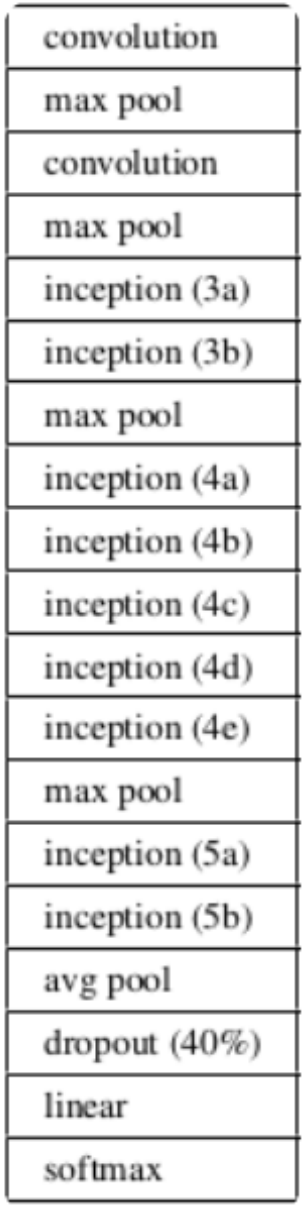
The dimension reduction approach is less computationally expensive:

- 1st layer computation cost = ((28 x 28 x 1 x 1) x 192) x 16 ≃ 2.4 Mil
- 2nd layer computation cost = ((28 x 28 x 5 x 5) x 16) x 32 ≃ 10 Mil
- Total computation cost ≃ **12.4 Mil**



- 9 Inception modules (red box)
- Max pooling were used instead of a Fully-connected layer.
- It enables adapting and fine-tuning on the network easily.
- 2 auxilaries softmax layer (green box)
- Their role is to push the network toward its goal and helps to ensure that the intermediate features are good enough for the network to learn.
- It turns out that softmax0 and sofmax1 gives regularization effect.
- During training, their loss gets added to the total loss with a discount weight (the losses of the auxiliary classifiers were weighted by 0.3).
- During inference, they are discarded.
- Structure:
- Average pooling layer with 5×5 filter size and stride 3 resulting in an output size:
- For 1st green box: 4x4x512.
- For 2nd green box: 4x4x528.
- 128 1x1 convolutions + ReLU.
- Fully-connected layer with 1024 units + ReLU.
- Dropout = 70%.
- Linear layer (1000 classes) + Softmax.

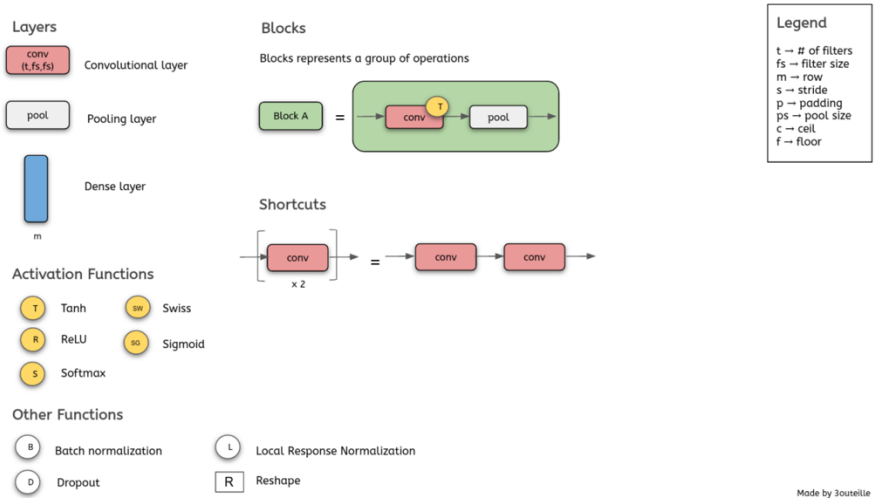
Summary of Architecture of Inception V1

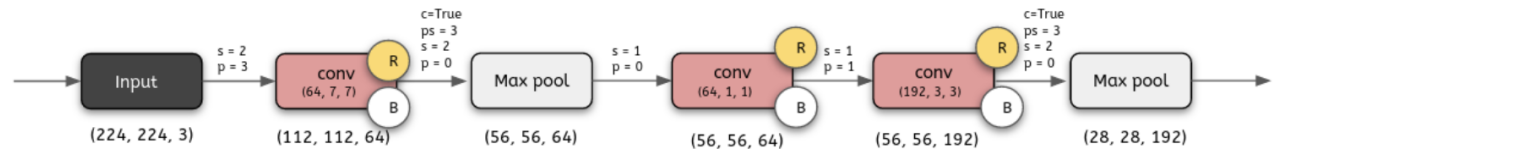


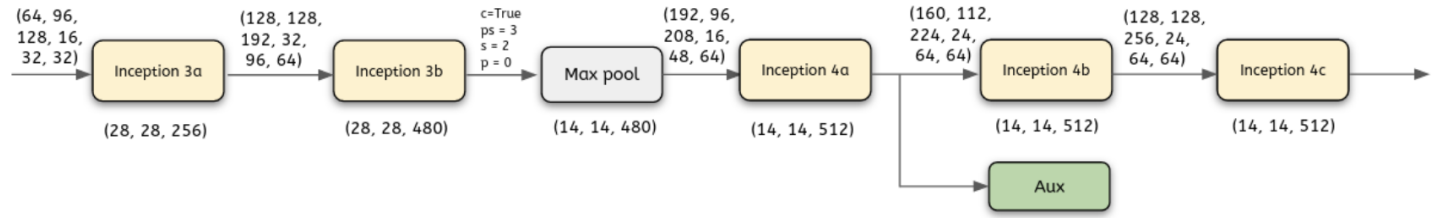
type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

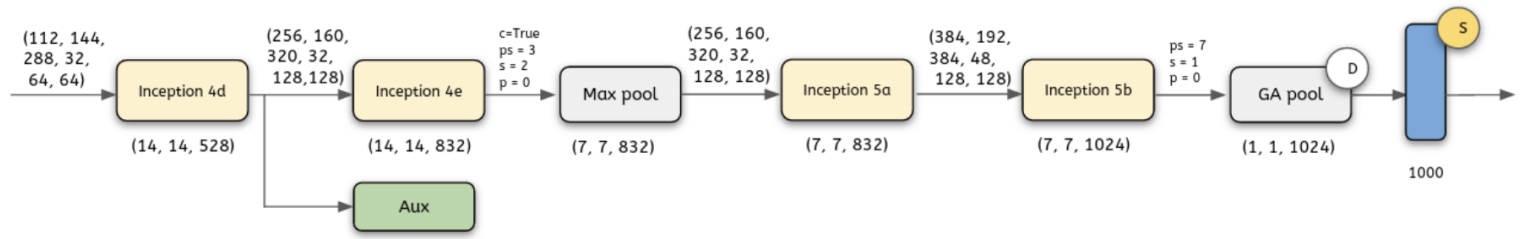
Characteristics and features of GoogLeNet configuration table

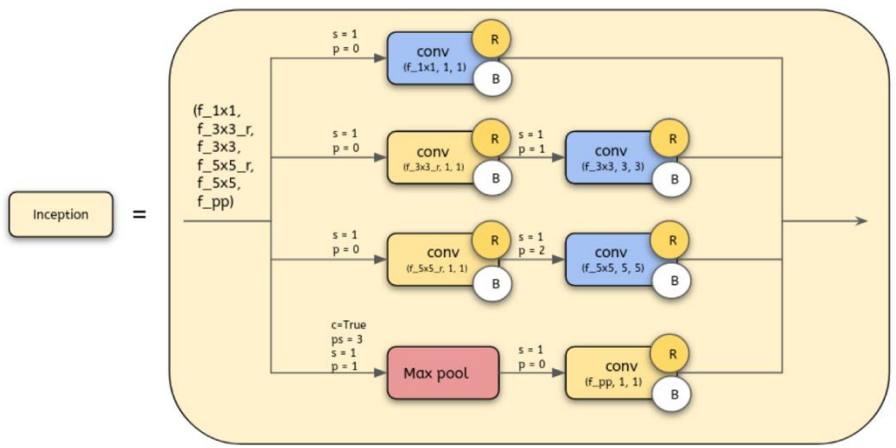
- The input layer of the GoogLeNet architecture takes in an image of the dimension 224 x 224.
- **Type:** This refers to the name of the current layer of the component within the architecture
- **Patch Size:** Refers to the size of the sweeping window utilised across conv and pooling layers. Sweeping windows have equal height and width.
- **Stride:** Defines the amount of shift the filter/sliding window takes over the input image.
- **Output Size:** The resulting output dimensions(height, width, number of feature maps) of the current architecture component after the input is passed through the layer.
- **Depth:** Refer to the number of levels/layers within an architecture component.
- **#1x1 #3x3 #5x5:** Refers to the various convolutions filters used within the inception module.
- **#3X3 reduce #5x5 reduce:** Refers to the numbers of 1x1 filters used before the convolutions.
- **Pool Proj:** This is the number of 1x1 filters used after pooling within an inception module.
- **Params:** Refers to the number of weights within the current architecture component.
- **Ops:** Refers to the number of mathematical operations carried out within the component.

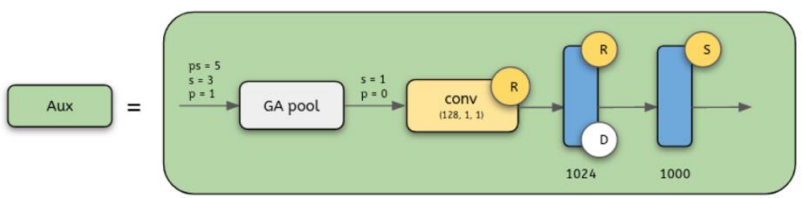












Legend

f_1x1 → # of filters for 1x1 conv.

f_3x3_r → # of filters for 1x1 conv before 3x3 conv.

f_3x3 → # of filters for 3x3 conv.

f_5x5_r → # of filters for 1x1 conv before 5x5 conv.

f_5x5 → # of filters for 5x5 conv.

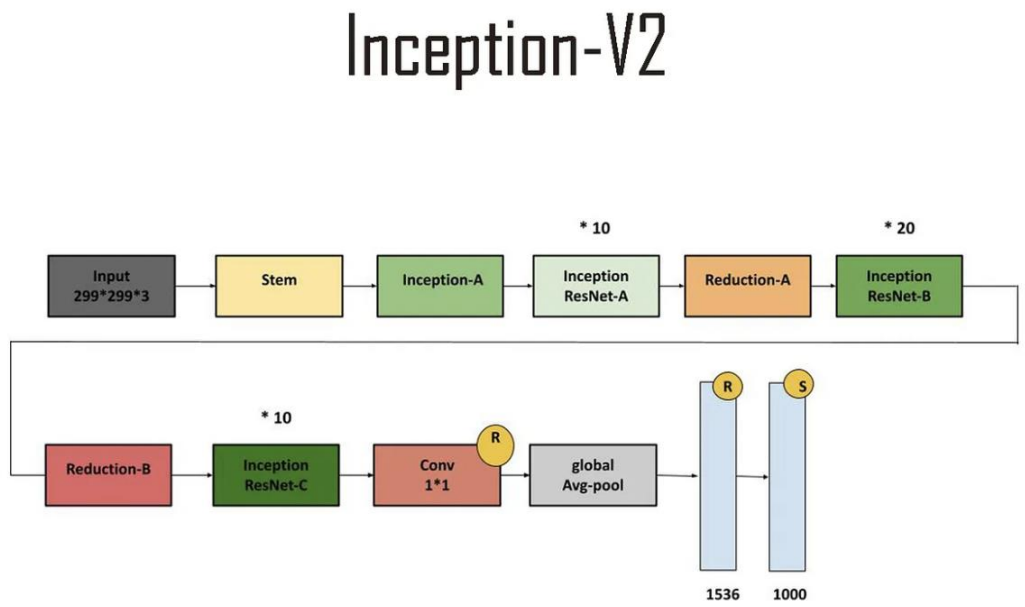
f_pp → # of filters for 1x1 conv after pooling.

Made by 3outeille

Please refer to Codes on Canvas

Inception V2

- The paper Rethinking the Inception Architecture for Computer Vision proposed a number of upgrades which increase the accuracy and reduce the computational complexity of a deep convolutional network. The authors studied it in the context of the Inception architecture.
- They also studied how factorizing convolutions and aggressive dimension reductions inside neural network can result in networks with relatively low computational cost while maintaining high quality.
- They benchmarked their results on the classification challenge validation with a single model that has:
- less than 25 million parameters
- 5 billion multiply-adds operations (for a single inference).
- top-1 and top-5 error rate of 21.2% and 5.6%.



Problems of Inception V1 architecture:

- lnception V1 have sometimes use convolutions such as *5*5* that causes the input dimensions to decrease by a large margin. This causes the neural network to uses some accuracy decrease. The reason behind that the neural network is susceptible to information loss if the input dimension decreases too drastically.

- Furthermore, there is also a complexity decrease when we use bigger convolutions like 5×5 as compared to 3×3. We can go further in terms of factorization i.e. that we can divide a 3×3 convolution into an asymmetric convolution of 1×3 then followed by a 3×1 convolution. This is equivalent to sliding a two-layer network with the same receptive field as in a 3×3 convolution but *33%* cheaper than 3×3.

- This factorization does not work well for early layers when input dimensions are big but only when the input size *mxm* (m is between 12 and 20). According to the Inception V1 architecture, the auxiliary classifier improves the convergence of the network. They argue that it can help reduce the effect of the vanishing gradient problem in the deep networks by pushing the useful gradient to earlier layers (to reduce the loss). But, the authors of this paper found that this classifier didn’t improve the convergence very much early in the training.

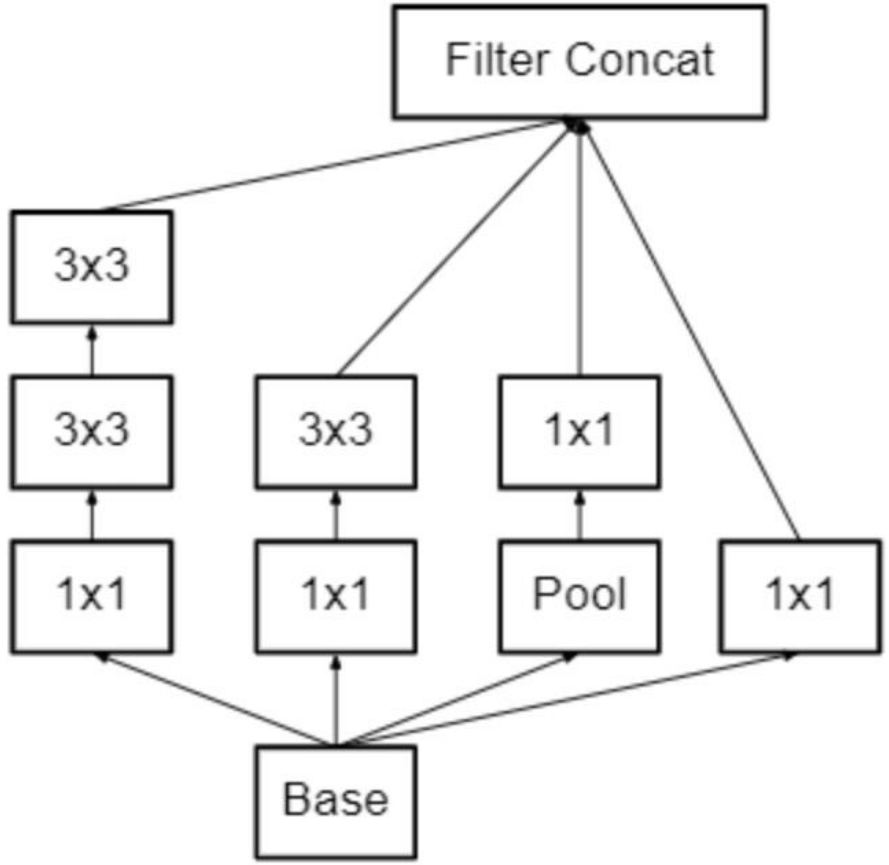
Note:

- Reduce representational bottleneck. The intuition was that, neural networks perform better when convolutions didn’t alter the dimensions of the input drastically. Reducing the dimensions too much may cause loss of information, known as a “representational bottleneck”
- Using smart factorization methods, convolutions can be made more efficient in terms of computational complexity.

Architectural Changes in Inception V2:

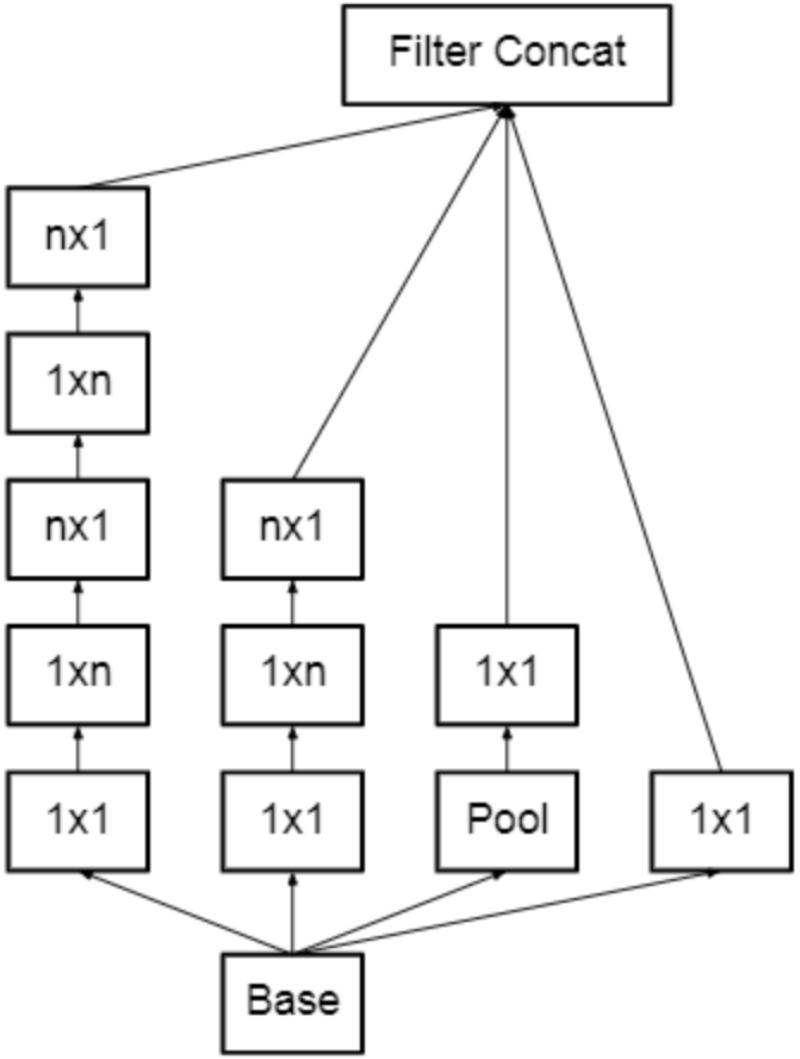
- In the Inception V2 architecture, the 5×5 convolution is replaced by the two 3×3 convolutions. This also decreases computational time and thus increases computational speed because a 5×5 convolution is 2.78 more expensive than a 3×3 convolution. So, Using two 3×3 layers instead of 5×5 increases the performance of architecture.

Note: The left-most 5x5 convolution of the old inception module, is now represented as two 3x3 convolutions



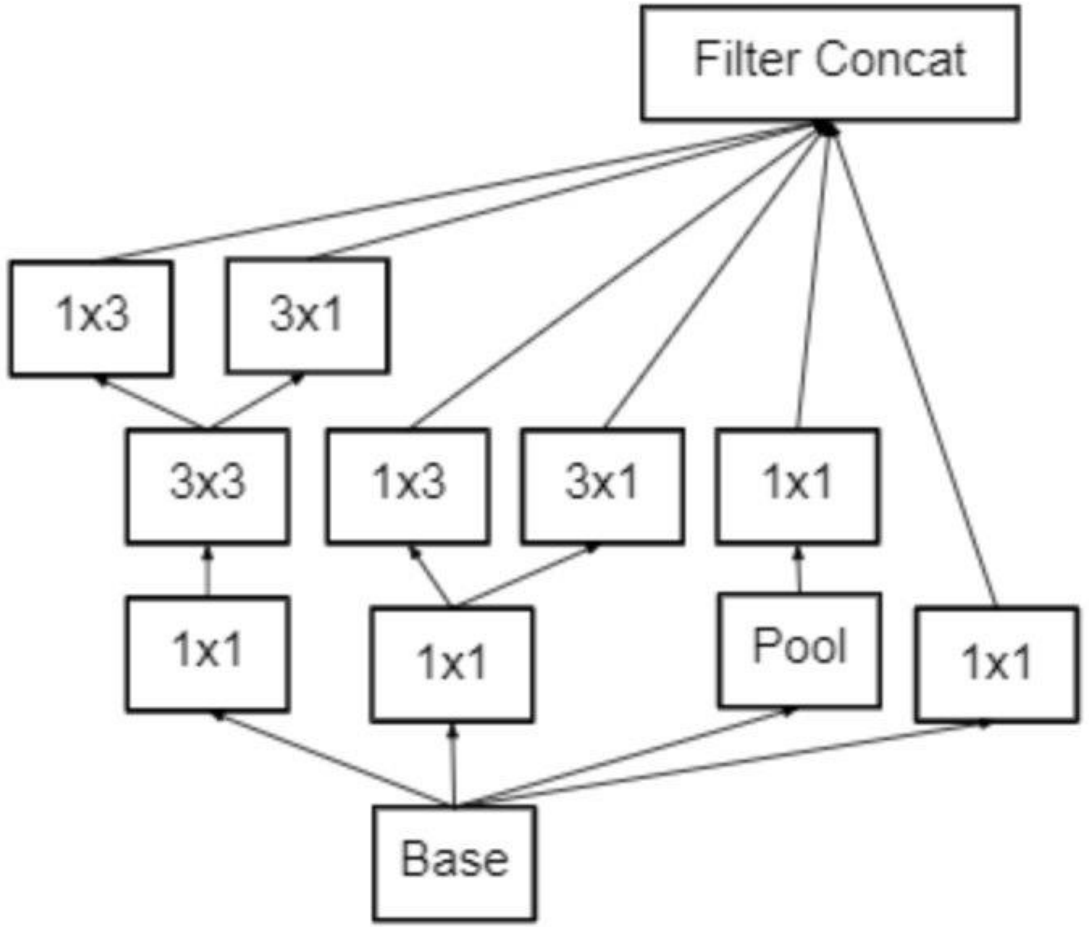
- This architecture also converts *nXn* factorization into 1xn and nx1 factorization. As we discussed above that a 3×3 convolution can be converted into *1×3* then followed by 3×1 convolution which is 33% cheaper in terms of computational complexity as compared to *3×3*.

Note: Here, put **n=3** to obtain the equivalent of the previous image. The left-most 5x5 convolution can be represented as two 3x3 convolutions, which in turn are represented as 1x3 and 3x1 in series.



- To deal with the problem of the representational bottleneck, the feature banks of the module were expanded instead of making it deeper. This would prevent the loss of information that causes when we make it deeper.

Note: Making the inception module wider.



The above three principles were used to build three different types of inception modules (Let's call them modules **A,B** and **C** in the order they were introduced. These names are introduced for clarity, and not the official names). The architecture is as follows:

Note: Figures 5, 6, 7 are the figures in the previous slides

type	patch size/stride or remarks	input size
conv	$3\times3/2$	$299\times299\times3$
conv	$3\times3/1$	$149\times149\times32$
conv padded	$3\times3/1$	$147\times147\times32$
pool	$3\times3/2$	$147\times147\times64$
conv	$3\times3/1$	$73\times73\times64$
conv	$3\times3/2$	$71\times71\times80$
conv	$3\times3/1$	$35\times35\times192$
$3\times$Inception	As in figure 5	$35\times35\times288$
$5\times$Inception	As in figure 6	$17\times17\times768$
$2\times$Inception	As in figure 7	$8\times8\times1280$
pool	8×8	$8\times8\times2048$
linear	logits	$1\times1\times2048$
softmax	classifier	$1\times1\times1000$

Inception V3

- The authors noted that the **auxiliary classifiers** didn’t contribute much until near the end of the training process, when accuracies were nearing saturation. They argued that they function as **regularizes**, especially if they have BatchNorm or Dropout operations.
- Possibilities to improve on the Inception v2 without drastically changing the modules were to be investigated.

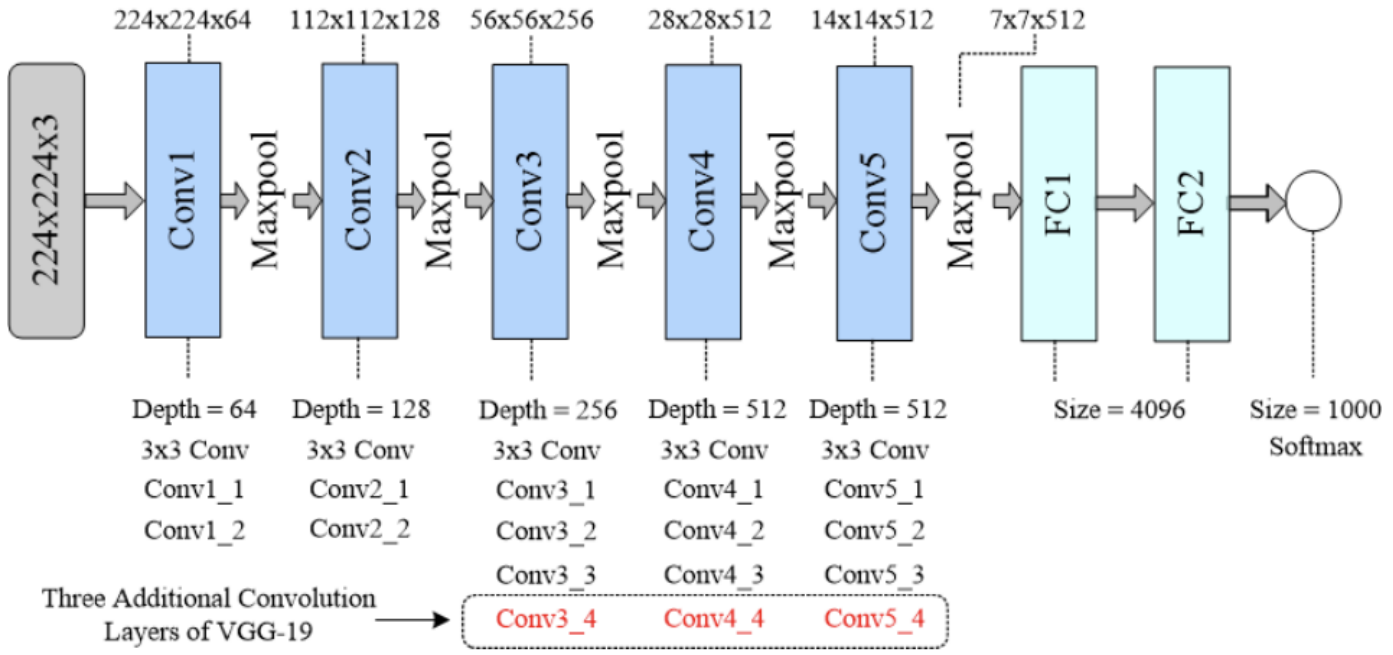
Please refer to Codes on Canvas

AlexNet
Input
Conv
Conv
Pool
Conv
Pool
FC
FC
Softmax
Layer1
Layer2
Layer3
Layer4
Layer5
Layer6
Layer7
VGGNet
Input
Conv
Conv
Pool
Conv
Conv
Pool
Conv
Conv
Pool
Conv
Conv
Pool
Conv
Conv
Pool
FC
FC
FC
Softmax
Layer1
Layer2
Layer3
Layer4
Layer5
Layer6
Layer7
Input : Image input
Conv : Convolutional layer
Pool : Max-pooling layer
FC : Fully-connected layer
Softmax : Softmax layer

Introduction to VGG Neural Network

- It is a typical deep Convolutional Neural Network (CNN) design with numerous layers, and the abbreviation VGG stands for Visual Geometry Group. The term “deep” describes the number of layers, with VGG-16 or VGG-19 having 16 or 19 convolutional layers, respectively.

- Innovative object identification models are built using the VGG architecture. The VGGNet, created as a deep neural network, outperforms benchmarks on a variety of tasks and datasets outside of ImageNet. It also remains one of the most often used image recognition architectures today.



Advantages of VGG

1.VGG brings with it a variety of structures based on the same idea. This provides us additional alternatives when it comes to determining which architecture would work best for our application.

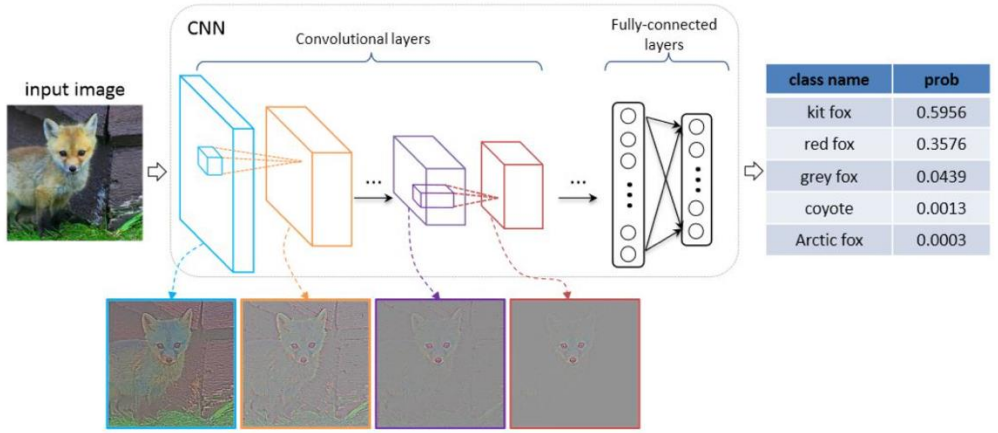
2.Non-linearity increased as the number of layers with smaller kernels increased, which is always a good thing in deep learning.

3.VGG resulted in a significant increase in accuracy as well as a significant increase in speed. This was mostly due to the model's increased depth and the addition of pretrained models.

Disadvantages of VGG

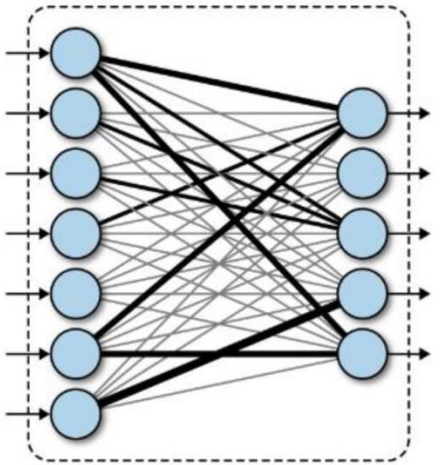
1.This model suffers from the vanishing gradient problem. When we look at my validation loss graph, we can see that it is steadily growing. None of the other models were in the same boat. The ResNet architecture was used to tackle the vanishing gradient problem.

2.The older VGG design is slower than the newer ResNet architecture, which introduced the idea of residual learning, another key accomplishment.

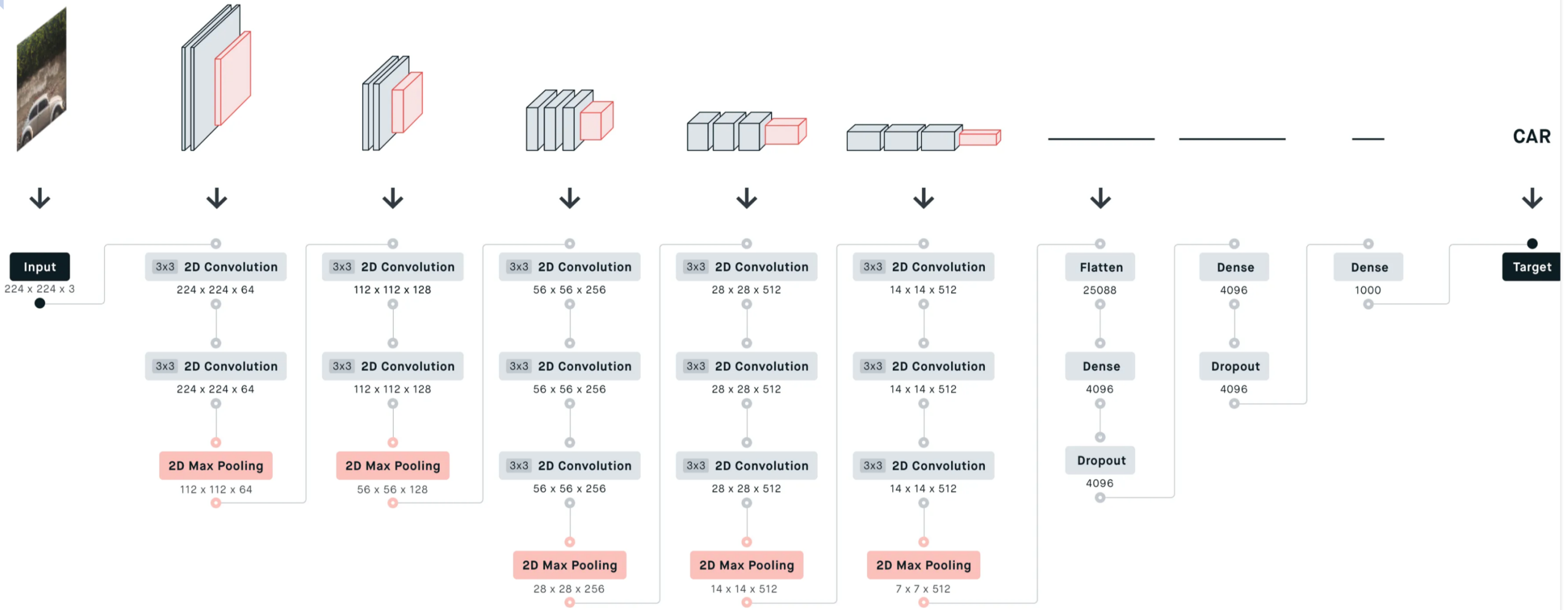


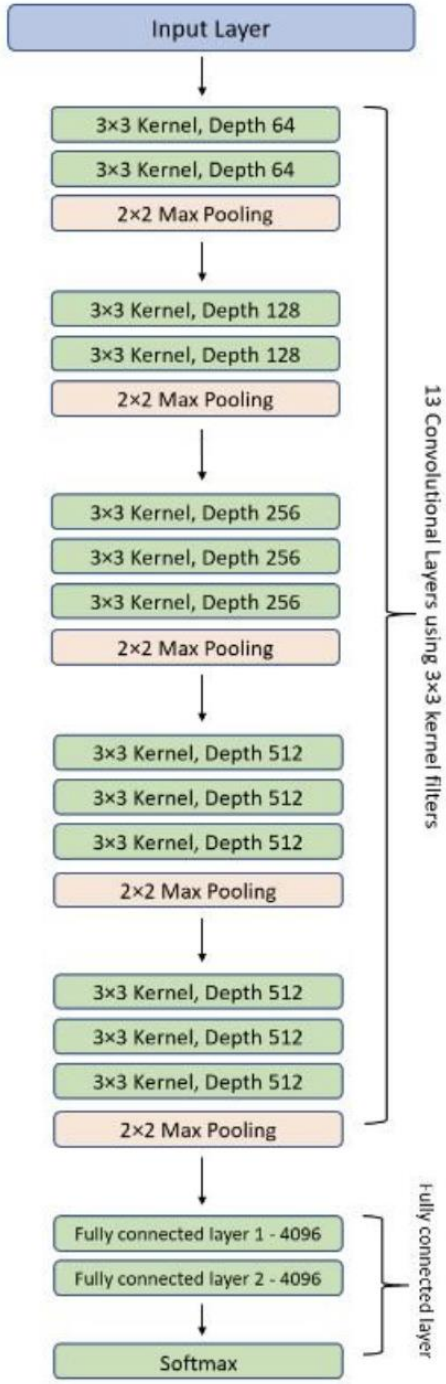
Architecture of VGG

- **Inputs**: The VGGNet accepts 224224-pixel images as input. To maintain a consistent input size for the ImageNet competition, the model's developers chopped out the central 224224 patches in each image.
- **Convolutional Layers**: VGG's convolutional layers use the smallest feasible receptive field, or 33, to record left-to-right and up-to-down movement. Additionally, 11 convolution filters are used to transform the input linearly. The next component is a ReLU unit, a significant advancement from AlexNet that shortens training time. Rectified linear unit activation function, or ReLU, is a piecewise linear function that, if the input is positive, outputs the input; otherwise, the output is zero. The convolution stride is fixed at 1 pixel to keep the spatial resolution preserved after convolution (stride is the number of pixel shifts over the input matrix).
- **Hidden Layers**: The VGG network's hidden layers all make use of ReLU. Local Response Normalization (LRN) is typically not used with VGG as it increases memory usage and training time. Furthermore, it doesn't increase overall accuracy.
- **Fully Connected Layers**: The VGGNet contains three layers with full connectivity. The first two levels each have 4096 channels, while the third layer has 1000 channels with one channel for each class.



can be used to describe how uniform it is.





VGG16 Architecture

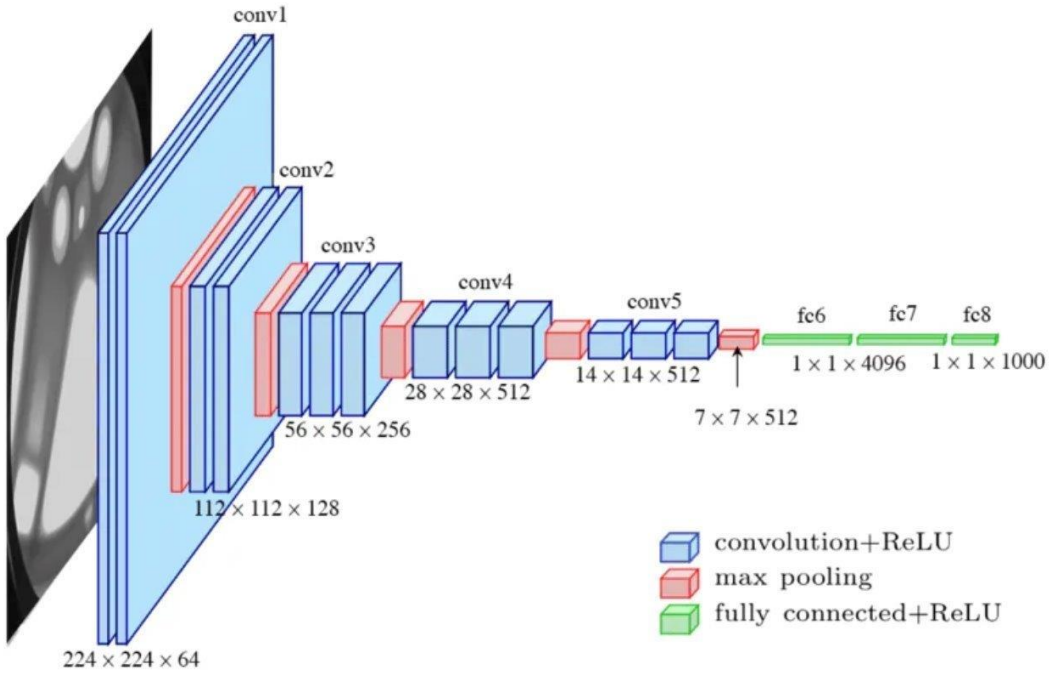
- The number 16 in the name VGG refers to the fact that it is 16 layers deep neural network (VGGnet). This means that VGG16 is a pretty extensive network and has a total of around 138 million parameters. Even according to modern standards, it is a huge network. However, VGGNet16 architecture's simplicity is what makes the network more appealing. Just by looking at its architecture, it can be said that it is quite uniform.

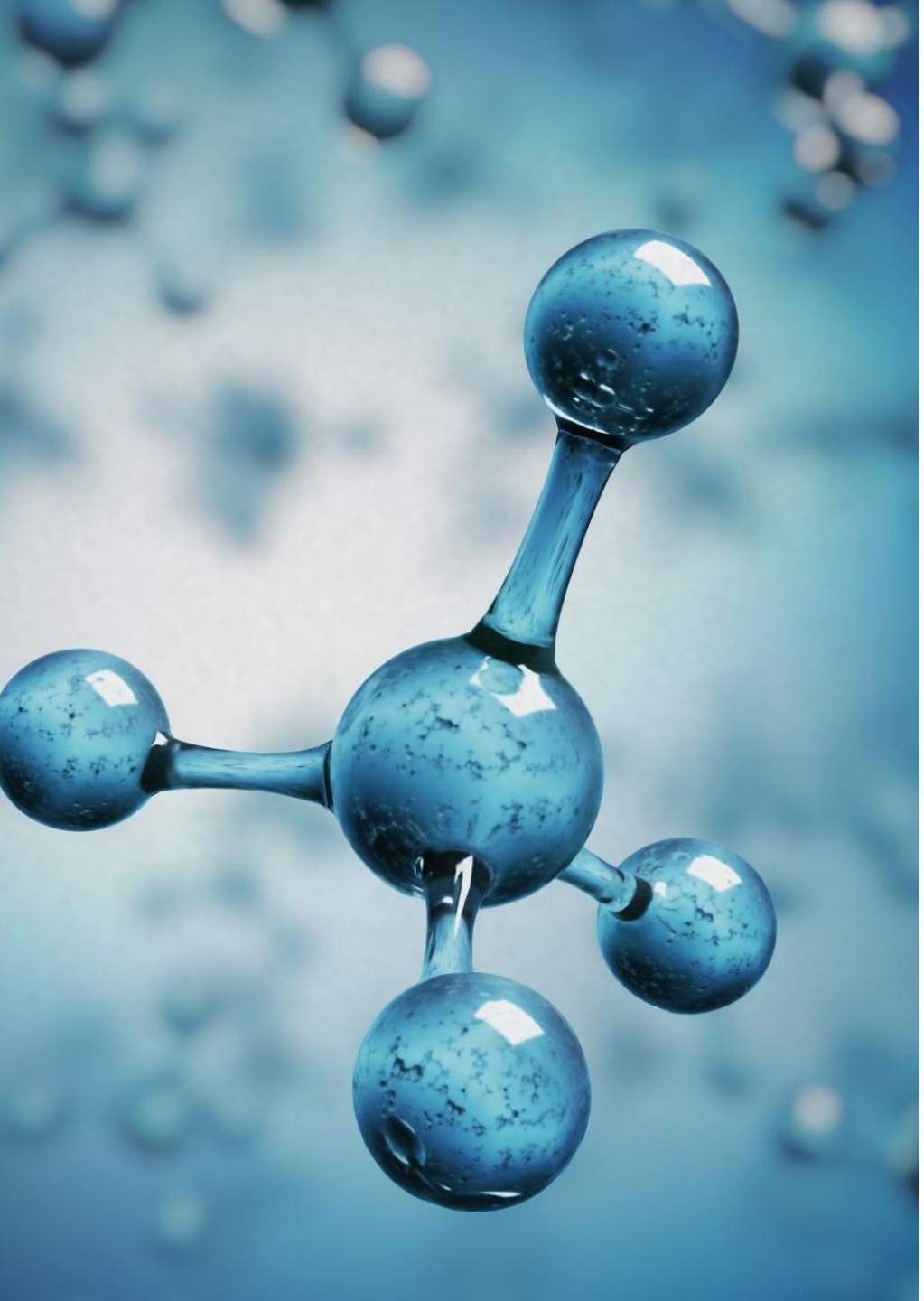
- There are a few convolution layers followed by a pooling layer that reduces the height and the width. If we look at the number of filters that we can use, around 64 filters are available that we can double to about 128 and then to 256 filters. In the last layers, we can use 512 filters.

Layer		Feature Map	Size	Kernel Size	Stride	Activation
Input	Image	1	224 x 224 x 3	-	-	-
1	2 X Convolution	64	224 x 224 x 64	3x3	1	relu
	Max Pooling	64	112 x 112 x 64	3x3	2	relu
3	2 X Convolution	128	112 x 112 x 128	3x3	1	relu
	Max Pooling	128	56 x 56 x 128	3x3	2	relu
5	2 X Convolution	256	56 x 56 x 256	3x3	1	relu
	Max Pooling	256	28 x 28 x 256	3x3	2	relu
7	3 X Convolution	512	28 x 28 x 512	3x3	1	relu
	Max Pooling	512	14 x 14 x 512	3x3	2	relu
10	3 X Convolution	512	14 x 14 x 512	3x3	1	relu
	Max Pooling	512	7 x 7 x 512	3x3	2	relu
13	FC	-	25088	-	-	relu
14	FC	-	4096	-	-	relu
15	FC	-	4096	-	-	relu
Output	FC	-	1000	-	-	Softmax

Limitations Of VGG 16:

- It is very slow to train (the original VGG model was trained on Nvidia Titan GPU for 2–3 weeks).
- The size of VGG-16 trained weights is *huge*. So, it takes quite a lot of disk space and bandwidth which makes it inefficient.
- 138 million parameters lead to exploding gradients problem





VGG-19

The VGG19 model (also known as VGGNet-19) has the same basic idea as the VGG16 model, with the exception that it supports 19 layers. The numbers “16” and “19” refer to the model’s weight layers (convolutional layers). In comparison to VGG16, VGG19 contains three extra convolutional layers.

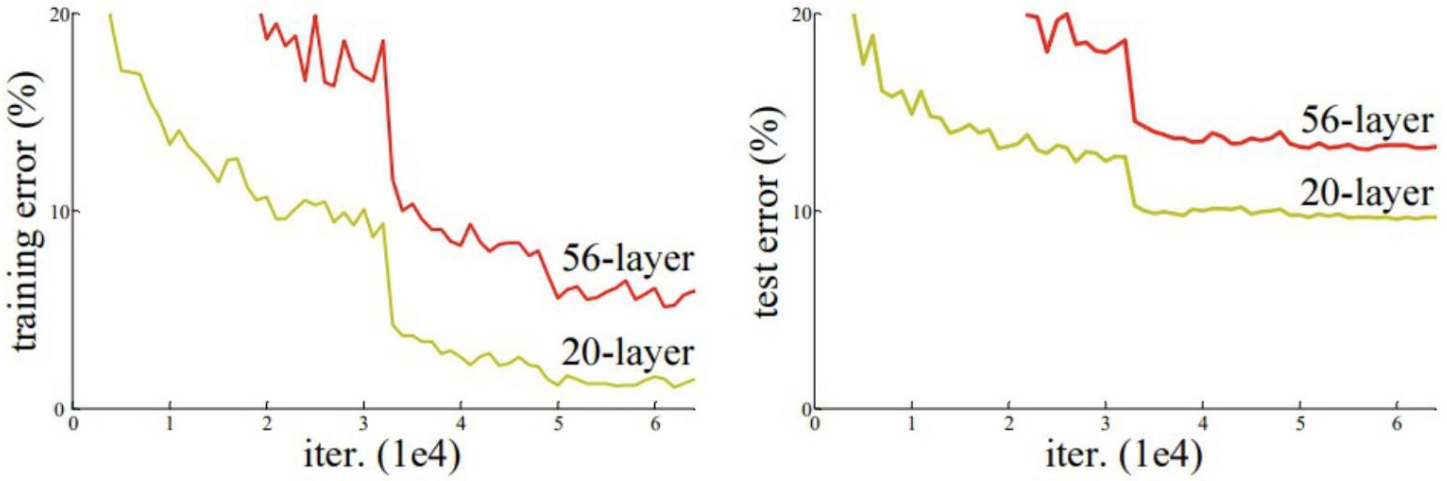
Configurations of VGG

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224×224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Please refer to Codes on Canvas

Residual Neural Network (ResNet)

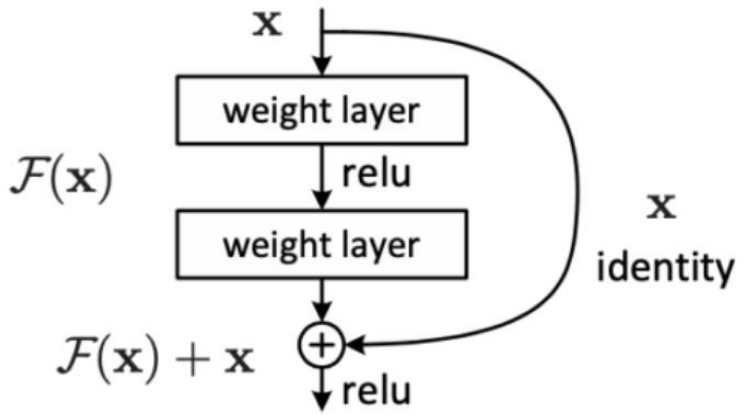
- After the first CNN-based architecture (AlexNet) that win the ImageNet 2012 competition, Every subsequent winning architecture uses more layers in a deep neural network to reduce the error rate. This works for a smaller number of layers, but when we increase the number of layers, there is a common problem in deep learning associated with that called the Vanishing/Exploding gradient. This causes the gradient to become 0 or too large. Thus, when we increases number of layers, the training and test error rate also increases.



- In the above plot, we can observe that a 56-layer CNN gives more error rate on both training and testing dataset than a 20-layer CNN architecture. After analyzing more on error rate, the authors were able to reach conclusion that it is caused by vanishing/exploding gradient.
- ResNet, which was proposed in 2015 by researchers at Microsoft Research introduced a new architecture called Residual Network.

- why Deeper Networks fail to perform better than their Shallow counterparts, it is sometimes better to look for empirical results for explanation and work backwards from there. The problem of training very deep networks has been alleviated with the introduction of a new neural network layer — The Residual Block. he image above shows a typical residual block. This can be expressed in Python code using the expression output = F(x) + x where x is an input to the residual block and output from the previous layer, and F(x) is part of a CNN consisting of several convolutional blocks.
- This technique smooths out the gradient flow during backpropagation, enabling the network to scale to 50, 100, or even 150 layers. Skipping a connection does not add additional computational load to the network.
- This technique of adding the input of the previous layer to the output of a subsequent layer is now very popular, and has been applied to many other neural network architectures including UNet and Recurrent Neural Networks (RNN).
- The image above shows a typical residual block. This can be expressed in Python code using the expression *output = F(x) + x* where x is an input to the residual block and output from the previous layer, and F(x) is part of a CNN consisting of several convolutional blocks.
- This technique smooths out the gradient flow during backpropagation, enabling the network to scale to 50, 100, or even 150 layers. Skipping a connection does not add additional computational load to the network.

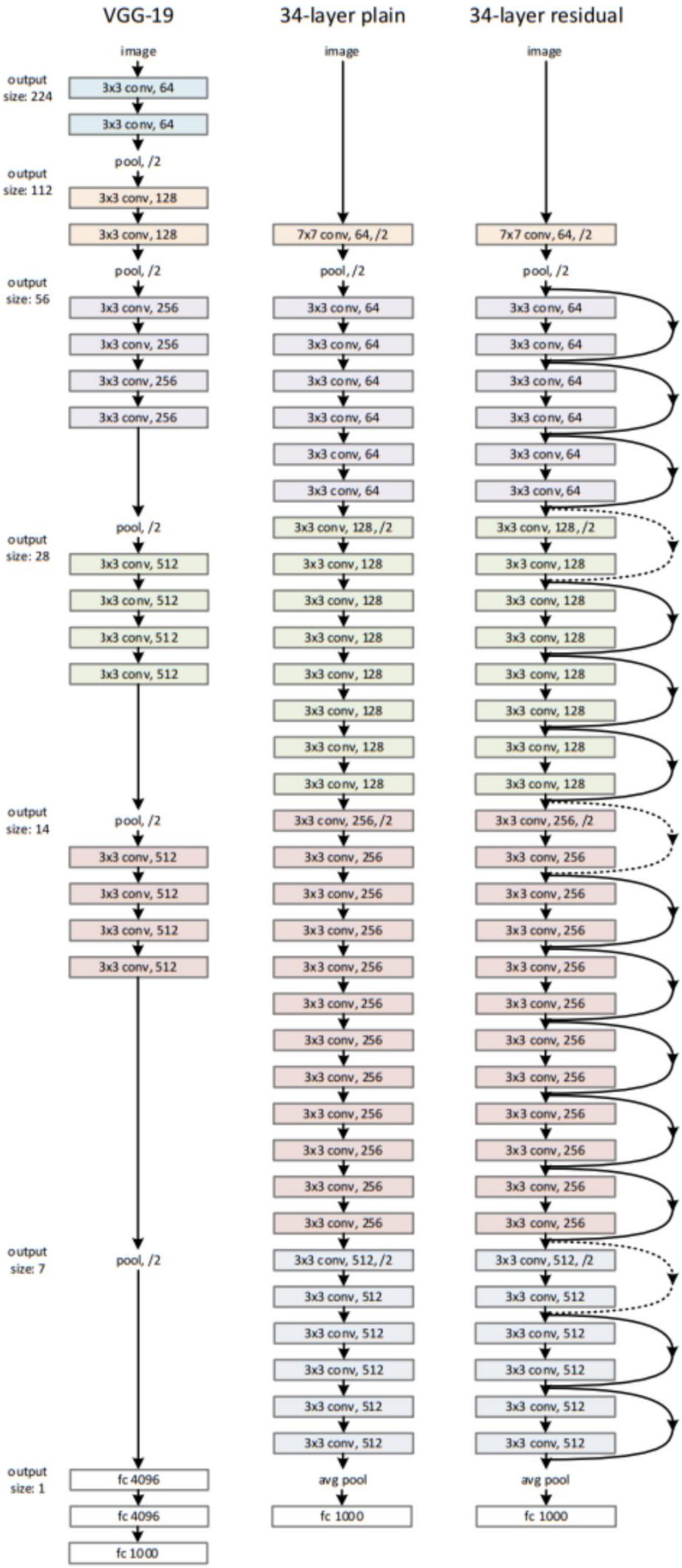
Skip Connections (or Shortcut Connections) as the name suggests skips some of the layers in the neural network and feeds the output of one layer as the input to the next layers.



How ResNet helps?

- The skip connections in ResNet solve the problem of vanishing gradient in deep neural networks by allowing this alternate shortcut path for the gradient to ow through. The other way that these connections help is by allowing the model to learn the identity functions which ensures that the higher layer will perform at least as good as the lower layer, and not worse. Let me explain this further. Say we have a shallow network and a deep network that maps an input ‘x’ to output ’y’ by using the function H(x). We want the deep network to perform at least as good as the shallow network and not degrade the performance as we saw in case of plain neural networks(without residual blocks). One way of achieving so is if the additional layers in a deep network learn the identity function and thus their output equals inputs which do not allow them to degrade the performance even with extra layers.

- To summarize, the Skip Connections between layers add the outputs from previous layers to the outputs of stacked layers. This results in the ability to train much deeper networks than what was previously possible.


VGG-19
34-layer plain
34-layer residual
image
output size: 224
3x3 conv, 64
3x3 conv, 64
pool, /2
output size: 112
3x3 conv, 128
3x3 conv, 128
pool, /2
output size: 56
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
pool, /2
output size: 28
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 14
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 7
fc 4096
fc 4096
fc 1000
output size: 1
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
avg pool
fc 1000

Architecture

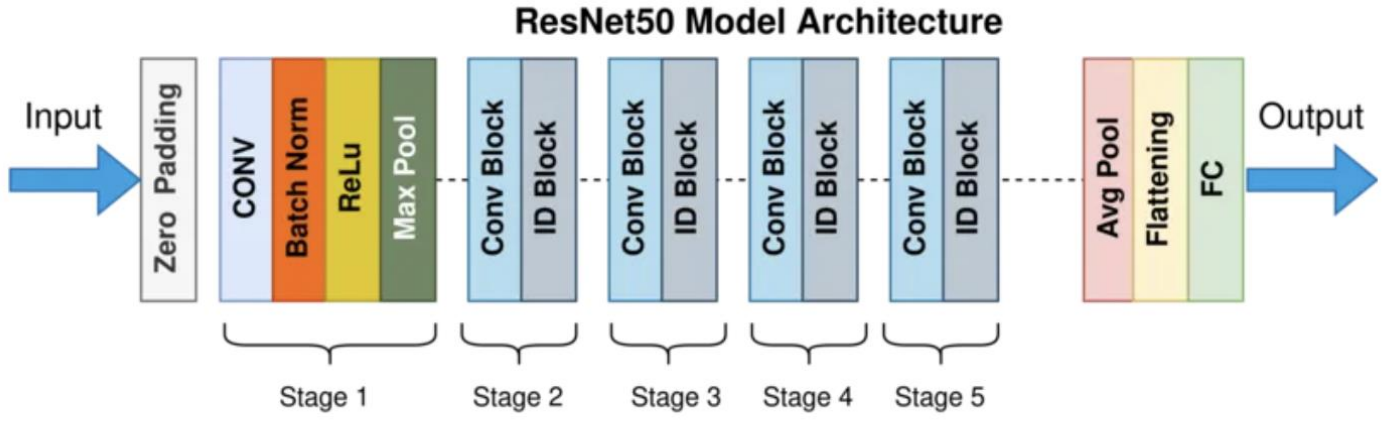
- In ResNet models, all convolutional layers apply the same convolutional window of size 3 × 3, the number of filters increases following the depth of networks, from 64 to 512 (for ResNet-18 and ResNet-34), from 64 to 2048 (for ResNet-50, ResNet-101, and ResNet-152).
- In all models, there is only one max-pooling layer with pooling size 3 × 3, and a stride of 2 is applied after the first layer. Therefore, reducing the resolution of the input is significantly limited during the training process.
- At the end of all models, the average pooling layer is applied to replace fully connected layers. This replacement has some advantages. Firstly, there is no parameter to optimize in this layer, hence it helps to reduce the model complexity. Secondly, this layer is more native to enforce the correspondences between feature maps and categories.
- The output layer has 1000 neurons which are corresponding to the number of categories in the ImageNet dataset. Besides, a SoftMax activation function is applied in this layer to give the probability that the input belonging to each class.

ResNet-50

- ResNet50 is a deep convolutional neural network (CNN) architecture that was developed by Microsoft Research in 2015. It is a variant of the popular ResNet architecture, which stands for “Residual Network.” The “50” in the name refers to the number of layers in the network, which is 50 layers deep.

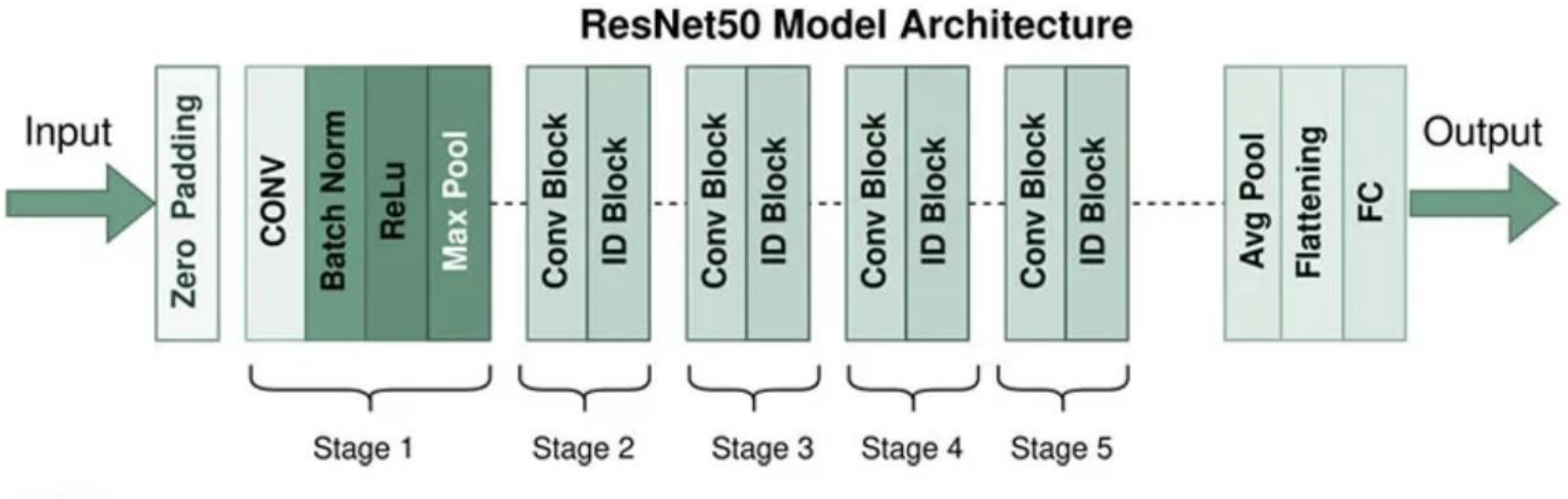
- The architecture of ResNet50 is divided into four main parts:
  - The convolutional layers,
  - The identity block,
  - The convolutional block, and
  - The fully connected layers

The convolutional layers are responsible for extracting features from the input image, while the identity block and convolutional block are responsible for processing and transforming these features. Finally, the fully connected layers are used to make the final classification.



ResNet50 Architecture

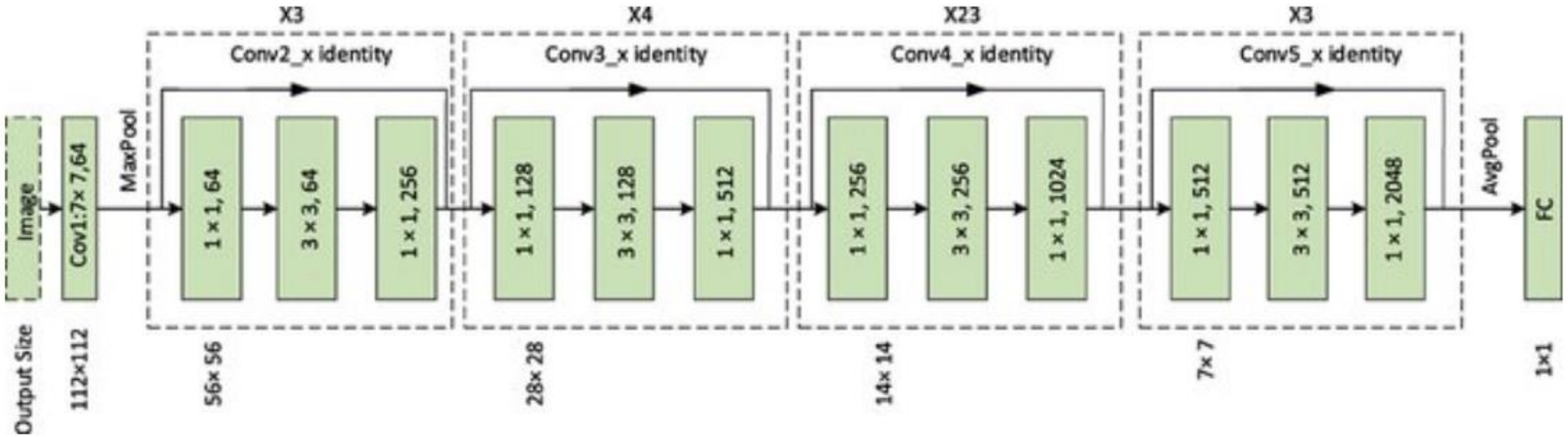
- The convolutional layers in ResNet50 consist of several convolutional layers followed by batch normalization and ReLU activation. These layers are responsible for extracting features from the input image, such as edges, textures, and shapes. The convolutional layers are followed by max pooling layers, which reduce the spatial dimensions of the feature maps while preserving the most important features.
- The identity block and convolutional block are the key building blocks of ResNet50. The identity block is a simple block that passes the input through a series of convolutional layers and adds the input back to the output. This allows the network to learn residual functions that map the input to the desired output. The convolutional block is similar to the identity block, but with the addition of a 1x1 convolutional layer that is used to reduce the number of filters before the 3x3 convolutional layer.
- The final part of ResNet50 is the fully connected layers. These layers are responsible for making the final classification. The output of the final fully connected layer is fed into a SoftMax activation function to produce the final class probabilities.



ResNet-101

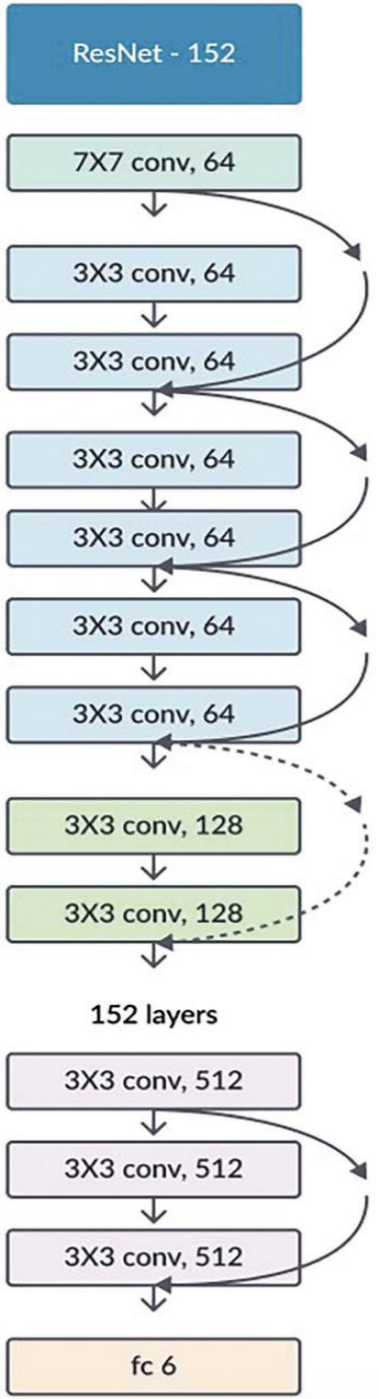
ResNet-101 is a convolutional neural network (CNN) architecture that belongs to the ResNet (Residual Network) family. The "101" in ResNet-101 refers to the depth of the network, specifically the number of layers it contains. ResNet-101 has 101 layers, which include convolutional layers, batch normalization layers, activation functions, and skip connections. These skip connections, also known as residual connections, help address the vanishing gradient problem by allowing gradients to flow more easily during training.

ResNet-101 has been pre-trained on large datasets like ImageNet and can be fine-tuned for specific tasks or used as a feature extractor in transfer learning scenarios. It has demonstrated state-of-the-art performance in various computer vision challenges and competitions.



ResNet-152

- ResNet-152 is another variant of the ResNet architecture, similar to ResNet-101 but with a deeper network consisting of 152 layers. Like other ResNet models, ResNet-152 utilizes residual connections to address the vanishing gradient problem, allowing for more effective training of very deep neural networks.
- The increased depth of ResNet-152 allows it to potentially capture more complex features and patterns in images, making it suitable for tasks that require a high level of detail and abstraction, such as fine-grained image classification, object detection, and semantic segmentation.
- Just like ResNet-101, ResNet-152 has been pretrained on large datasets like ImageNet and can be adapted for specific tasks through transfer learning or fine-tuning. It has demonstrated impressive performance in various computer vision benchmarks and competitions, showcasing its effectiveness in a wide range of visual recognition tasks.




THANK YOU
Any Questions?