CONVOLUTIONAL NEURAL NETWORK
FALL SEMESTER 2018

INSTRUCTOR: PROF. ANKITA PRAMANIK, SILADITTYA MANNA

Tips: *As you read, try to visualize the information/data flow also. Also, try to recreate the algorithms if possible. Related Codes and tutorials links will be made available in the Github repository.*

Computer Vision Problems:

1. Image Classification
2. Object Detection and Localization
3. Neural Style Transfer

Why CNN?

Suppose, you have a 28X28 RGB image. So, the total number of inputs in a neural network will be 28X28X3 = 1872.

Now, let you have a 1000X1000 RGB image. In this case the total number of inputs in a neural network will be 3 million, which is pretty large.

Since, the number of inputs have increased, the number of weight parameters, will also increase. If there are 1000 nodes in the first layer, the number of elements in the weight matrix of the first layer will be, 3 billion.

Thus, we can see that with the increase in the dimension of the image, there is a huge increase in the number of parameters, in a feedforward neural network.

Thus, it is pretty difficult to train a neural network with such a large number of parameters.

This discussion will continue at the end of this lecture note.

Computer Vision Problem

So, how is a computer vision problem framed using NN?

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

Suppose this is 6X6 grayscale image

And we wish to detect vertical edges in it.

So, the filter or kernel we will use is this

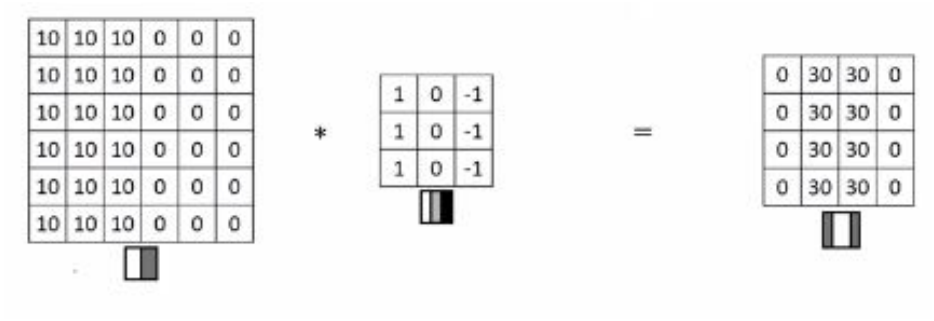
1	0	-1
1	0	-1
1	0	-1

After convolution, the resultant matrix, we get is

-5	-4	0	8
-10	-2	2	3
0	-2	-4	-7
-3	-2	-3	-16

Now, have a look at another example in the next page

Below Image taken from Convolutional Neural Network by Andrew Ng



This is an example of vertical edge detection. Likewise, using different filters we can detect horizontal edges and also edges at different angles.

Now, these filters can also be learnt using neural networks, which will determine the 9 values values of the filter.

What we can do is, treat each element of the filter as parameters and learn these parameters using back-propagation, similar to the ordinary neural network.

A short summary of convolutional operations

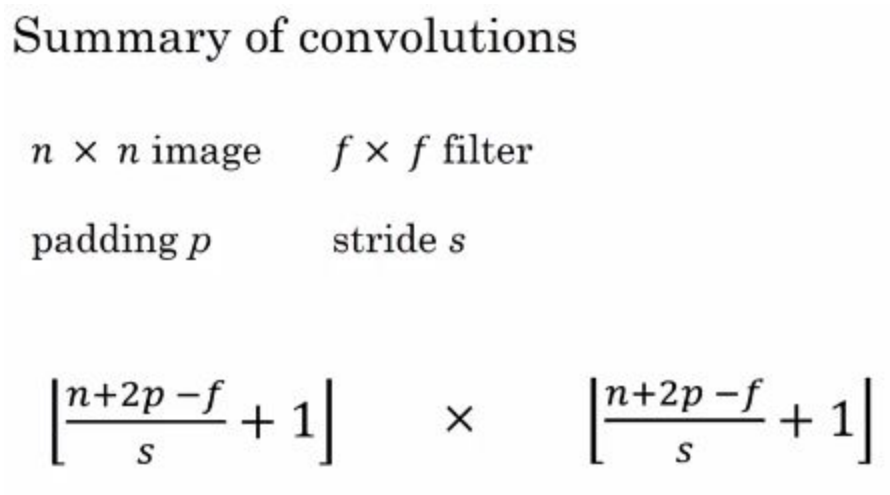
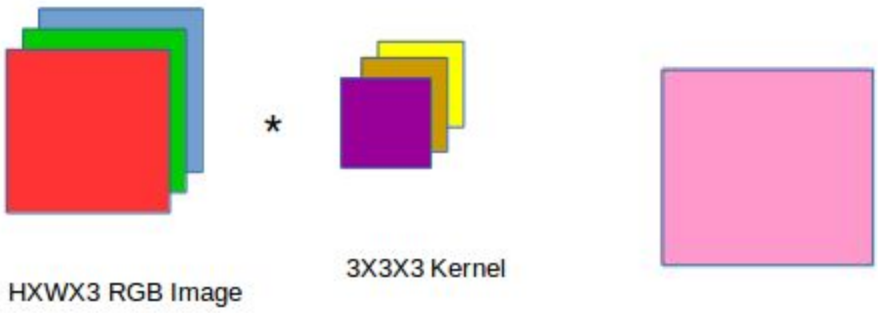


Image taken from Convolutional Neural Network by Andrew Ng

How to do convolutions on RGB Images?

Now, since an RGB image consists of 3 channels, we will need to have 3 filters for each channel.

So, for an image 6X6X3, we will need a filter of shape 3X3X3



So, how is this convolution computed?

As in 2D convolution, the first filter is convoluted with the Red channel, the second filter with the Green channel, and the third filter with the Blue channel. Now the values at each convolutional step are added over the channels to give the final result, which will output a single channel, or a 2D matrix.

Now, suppose, the above 3X3X3 filter used was for detecting vertical edges. Now, suppose that we also want to detect Horizontal edges. So, we will need another 3X3X3 filter for that purpose, which will again output a 2D matrix.

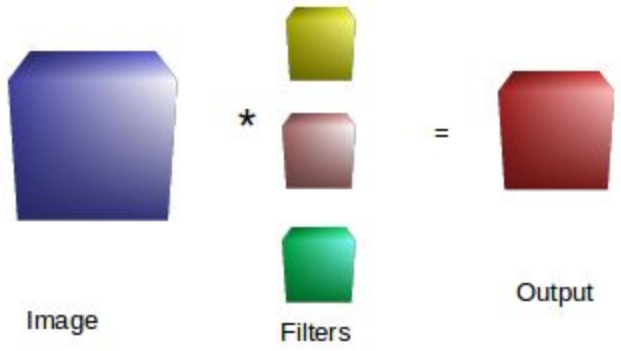
By stacking the output of these two filters, we get a [(H-f+1) X (W-f+1) X 2] output (considering no padding).

So, the number of channels in the output is equal to the number of filters we are using.

And, the number of channels in each filter = number of filters in the input

However, before stacking up the outputs, bias is added to the output and passed through the activation function, which is then used as input to the next layer.

Convolutional Layer



Now, there are various types of layers in a CNN:

1. Convolutional
2. Pooling
3. Fully Connected

Pooling Layers:

1	3	2	1
2	9	1	1
1	3	2	3
5	6	1	2

Suppose, we have this 2D matrix

Max-Pooling:

1	3	2	1
2	9	1	1
1	3	2	3

5	6	1	2

Now, pooling is effectively, taking the max of the elements in a **f** X **f** region.

Suppose, we take a 2X2 filter, with strides 2, the output will be a 2X2 2D matrix

Now, the elements in the output will be max of the elements in the 2X2 region, the filter is passed over.

Going by this way, the output will be

9	2
6	3

Now, if we have a 3D input, the max-pooling output will have the same number of channels as in input. If the number of channels in the input is n_c, then the number of channels in the output of max-pooling will also be n_c.

Average Pooling

Instead of taking the max of the elements, we take the average in this technique.

3.75	1.25
3.75	2

One important point to note about Pooling layers is that, there are no trainable parameters in Pooling layers.

Now, coming back to the first topic, Why convolutions?

As is evident, that the number of trainable parameters is greatly reduced in CNN.

The two important features of CNNs are:

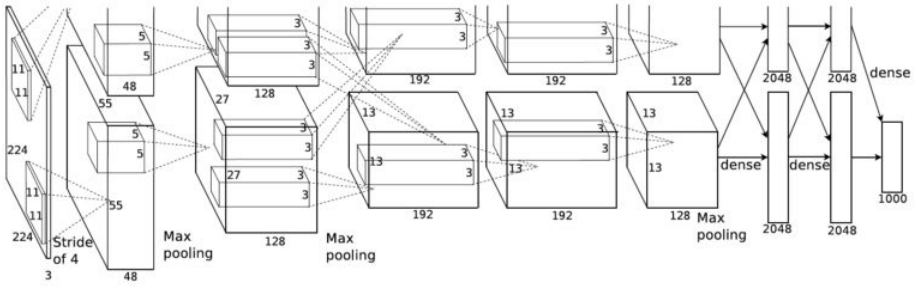
1. **Parameter Sharing**: A filter learnt can be used to detect a feature over all of the input image. For example, a vertical edge detector, can be used to detect edges in the top left part of an image, and also in the bottom right part of an image, we don't need separate filters for different parts of an image to detect the same feature.

2. **Sparsity of Connections**: In each layer, each output value is dependent only on a small number of inputs.

These two features allows the CNNs to have a small number of parameters. CNNs also have translation invariance, that is even if you shift the image by a few pixels, the same features will be captured.

CASE STUDY:

AlexNet:



Layers	Size	Number of filters	Stride	Padding	Output Size
Input	227X227X3				
Conv_1+ReLU	11X11	96	4		55X55X96
MaxPool_1	3X3		2		27X27X96
LRN					
Conv_2+ReLU	5X5	256	1	2	27X27X256
MaxPool_2	3X3		2		13X13X256
LRN					
Conv_3+ReLU	3X3	384	1	1	13X13X384
Conv_4+ReLU	3X3	384	1	1	13X13X384
Conv_5+ReLU	3X3	256	1	1	13X13X256
MaxPool_3	3X3		2		6X6X256

Dropout(0.5)					
FC6+rReLU					4096
Dropout(0.5)					
FC7+ReLU					4096
FC8+Softmax					1000

Total Number of Parameters = 62.3 million

Total number of Forward Computation = 1.1 billion

Local Response Normalization(LRN)

Normalized response $b^i_{x,y}$ is given by

$$b^i_{x,y} = a^i_{x,y}/(k + \alpha \sum_{j=max(0,i-n/2)}^{min(N,i+n/2)} (a^i_{x,y})^2$$

Where, the sum runs over n "adjacent" kernel maps at the same spatial position, and N is the total number of kernels in the layer.

$a^i_{x,y}$ is the activity of a neuron obtained by applying kernel i at position (x,y) and then applying ReLU non-linearity.

This sort of response normalization implements a form of lateral inhibition inspired by the type found in real neurons, creating competition for big activities amongst neuron outputs computed using different kernels. The constants k,n,α and β are hyper-parameters whose values are determined using a validation set.

Highlights of the Paper:

1. Used ReLU instead of Tanh. Speed accelerated by 6 times
2. Used dropout regularization to reduce over-fitting. Training time doubles with 0.5 dropout rate
3. Overlap pooling used. Increased top-1 and top-5 accuracy by 0.4% and 0.3% respectively

VGG

VGG-net was introduced by Simonyan and Zisserman in their 2014 paper, 'Very Deep Convolutional Networks for Large Scale Image Recognition'

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					

Preprocessing step:

Subtracting the mean RGB value, computed on the training set, from each pixel

The VGG-net uses a small receptive field of 3X3 and also has used 1x1 filters which acts as a linear transformation of the input channels.

Convolutional stride is fixed to 1 pixel, and padding is such that after each convolution the spatial dimension of the inputs is preserved, i.e. padding is 1 pixel for 3X3 filters.

Max-Pooling is performed over a 2X2 window and a stride of 2, which downsizes the output of the convolutional layers, to half.

Lastly, a stack of convolutional layers is followed by 3 fully-connected layers.

First two FC layers have 4096 channels each and the third one has 1000, which performs the classification for 100 classes in ILSVRC Dataset.

The final layer is the softmax layer.

Non-linearity used is ReLU.

It is easy to see that a stack of two 3×3 conv. layers (without spatial pooling in between) has an effective receptive field of 5×5; three such layers have a 7 × 7 effective receptive field.

The reason for using 3 consecutive 3X3 filters instead of a single 7X7 filter

First, by introducing 3 consecutive non-linear rectification layers instead of a single one, which makes the decision function more discriminative

Second, decrease in the number of parameters. : assuming that both the input and the output of a three-layer 3X3 convolution stack has C channels, the stack is parameterized by $3(3^2C^2) = 27C^2$ weights; at the same time, a single 7X7 conv. layer would require $7^2C^2 = 49C^2$ parameters, i.e. 81% more. This can be seen as imposing a regularisation on the 7X7 conv. filters, forcing them to have a decomposition through the 3X3 filters(with non-linearity injected in between them).

The incorporation of 1X1 conv. layers (Table) is a way to increase the nonlinearity of the decision function without affecting the receptive fields of the conv. layers. Even though in our case the 1X1 convolution is essentially a linear projection onto the space of the same

dimensionality (the number of input and output channels is the same), an additional non-linearity is introduced by the rectification function

Training

Training is carried out by optimizing the multinomial logistic regression objective using mini-batch gradient descent with momentum.

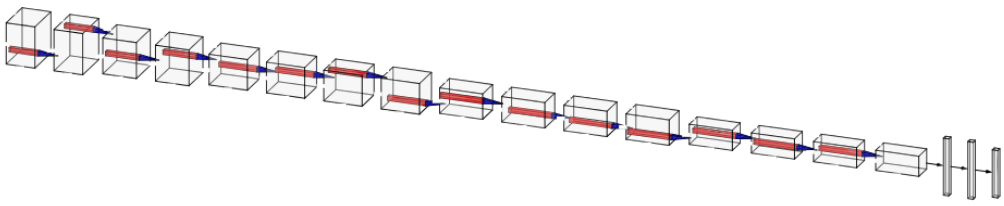
During training, the regularization methods used are L2 (weight decay multiplier = 5.10^{-4}) and Dropout(0.5)

Learning rate was set to 0.01 and decreased by a factor of 10 when the validation accuracy stopped increasing.

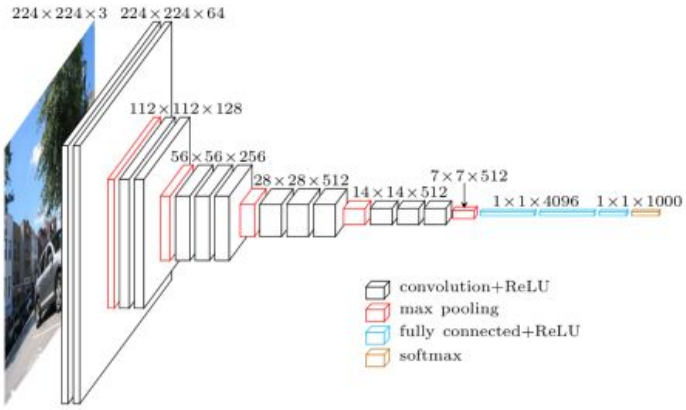
Also, they used Xavier Initialization for initializing the weights in the network.

Another approach they used, is to use single scaled training at two fixed scales: S = 256 and S = 384 and second approach is to randomly sample S (smallest side of an isotropically-rescaled training image) from a certain range [S_{min},S_{max}] (S_{min} = 256 and S_{max} = 512). After that the images are cropped to 224X224, which will correspond to a small part of the image, containing a small object or an object part.

VGG-19



VGG-16



source

Inception Net (GoogLeNet) and its variants

This meme was referenced in the first Inception paper.

<u>Motivations behind Inception architecture</u>:

1. Deeper networks have large number of parameters, which makes them prone to overfitting.
2. Increased network size causes increase in requirement of computational resources.
3. The main idea of the Inception Architecture is based on finding out how an optimal local sparse structure in a Convolutional Vision Network can be approximated and covered by readily available dense components.

The fundamental way of solving both issues, as mentioned in their paper, would be by moving from fully connected to sparsely connected architectures even inside the convolutions. The result of Arora et al. states that if the probability distribution of the data-set is represented by a large, very sparse deep neural network, then the optimal network topology can be constructed layer by layer by analyzing the correlation statistics of the activations of the last layer and clustering neurons with the highly correlated outputs.

The vast literature on sparse matrix computations suggests that clustering sparse matrices into relatively dense submatrices tends to give state of the art practical performance for sparse matrix multiplication. The clusters formed by clustering neurons with highly correlated outputs form the units of the ext layer and are connected to the units in the previous layer. It is assumed that each unit from the earlier layer corresponds to some region of the input image and these units are grouped into filter banks.

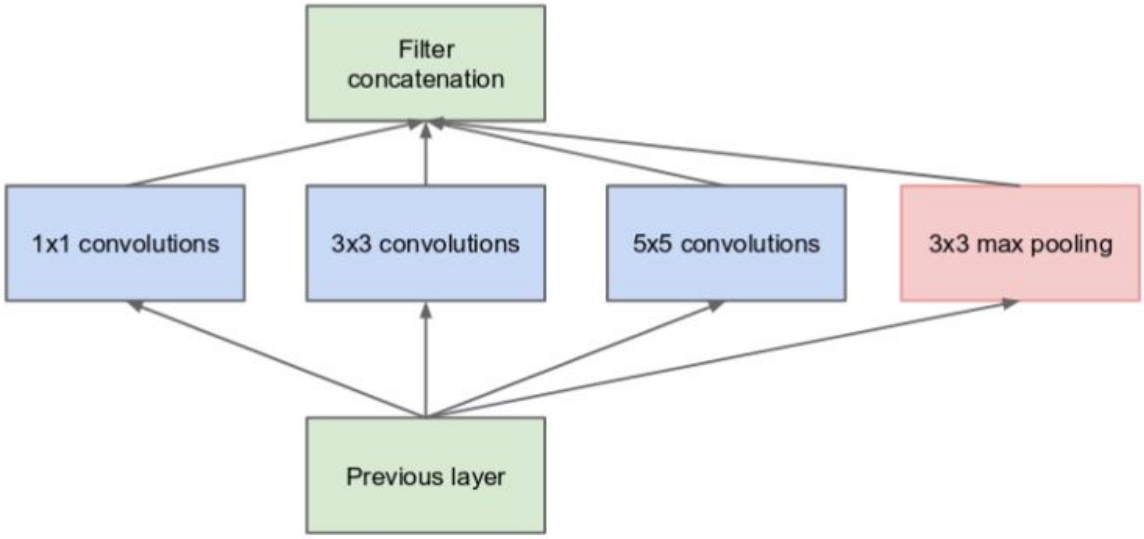
In lower layers, correlated units would concentrate in local regions. This means, a lot of clusters will be concentrated in a single region and they can be covered by 1X1 convolutions in the next layer. However, in addition to these local features, there will be some spatially spread out clusters that can be covered by convolutions over large patches, and there will be decreasing number of patches over larger and larger regions.

The output of these filter banks will be concatenated to form a single vector which will be the input of the next layer.

Additionally, since pooling operations have been essential for the success in current state of the art convolutional networks, it suggests that adding an alternative parallel pooling path in each such stage should have additional beneficial effect, too.

As these “Inception modules” are stacked on top of each other, their output correlation statistics are bound to vary: as features of higher abstraction are captured by higher layers, their spatial concentration is expected to decrease suggesting that the ratio of 3×3 and 5×5 convolutions should increase as we move to higher layers.

The naive Inception module:

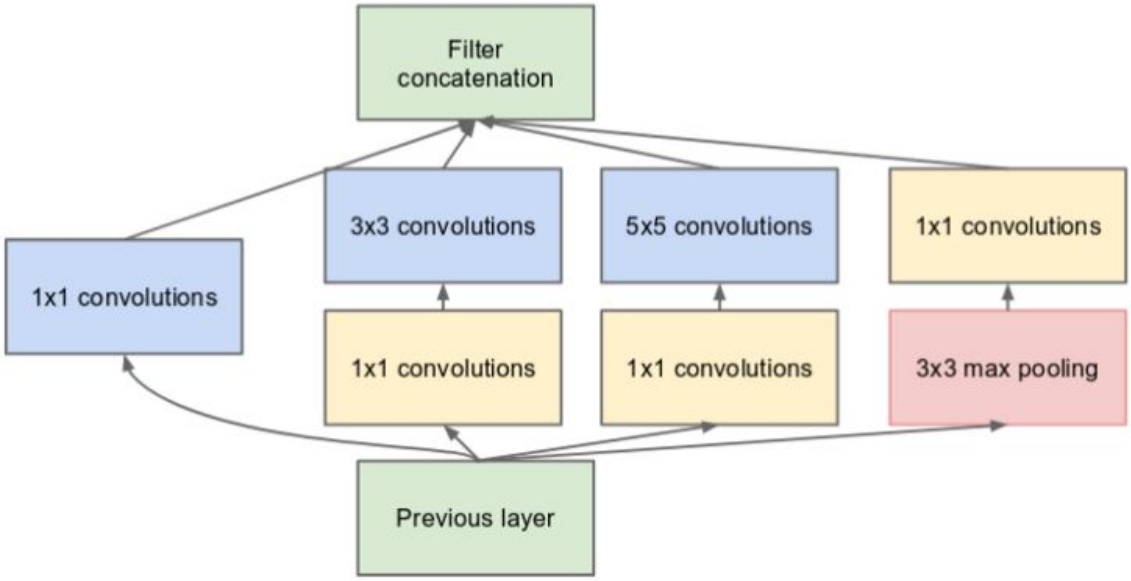


(a) Inception module, naïve version

But even a modest number of 5X5 convolutions causes a large increase in the number of computations. This problem becomes more pronounced, if the pooling layers are merged with the output of the convolutional layers. Thus, even though this architecture represents optimal sparse structure, it would be very inefficient. Hence, the authors proposed a modification.

Low dimensional embeddings can contain a lot of information about a relatively large image patch. So, they used 1X1 convolutions before 3X3 and 5X5 convolutional layers, which reduces the dimensions, thereby compressing the information. Also the ReLU used as activation, makes them dual-purpose.

Inception Module with dimension reductions:



(b) Inception module with dimension reductions

In their paper, they also used auxiliary classifiers. The main idea behind it was that, even shallower networks, are capable of performing well in the task, suggests that the features produced by the middle of the network, should be very discriminative. Thus by adding auxiliary classifiers to the intermediate layers, encouraged discrimination in the lower layers, and increase the gradient signal propagating back and also provide additional regularization. During training the loss of these auxiliary classifiers are added to the total loss of the network, multiplied by a factor of 0.3. However, during testing phase, these auxiliary classifiers are discarded.

The structure for these auxiliary classifiers are as follows:

1. An average pooling layer with 5×5 filter size and stride 3, resulting in an 4×4×512 output for the (4a), and 4×4×528 for the (4d) stage.
2. A 1×1 convolution with 128 filters for dimension reduction and rectified linear activation.
3. A fully connected layer with 1024 units and rectified linear activation.
4. A dropout layer with 70% ratio of dropped outputs.

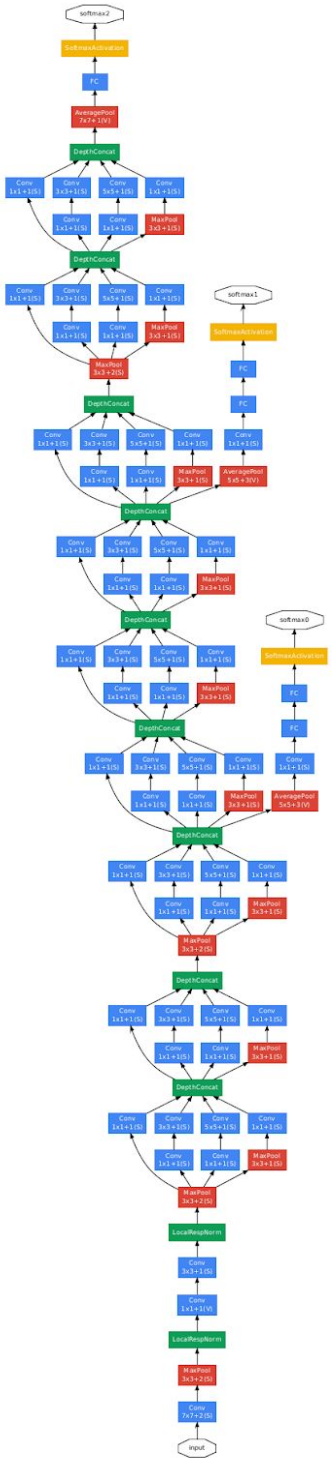
5. A linear layer with softmax loss as the classifier (predicting the same 1000 classes as the main classifier, but removed at inference time).

The use of average pooling before the classifier is based on *Network in Networks*, although here the implementation differs in that they have used an extra linear layer. This enables adapting and fine-tuning our networks for other label sets easily.

Training Methodology:

1. Used asynchronous stochastic gradient descent with momentum (0.9)
2. Decreasing learning rate by 4% every 8 epochs
3. Polyak averaging was used to create final model at inference time
4. Sampling various sized patches of the image, whose size is evenly distributed between 8% to 100% of the image area and whose aspect ratio is chosen randomly between ¾ and 4/3
5. Photometric distortions
6. Random interpolation methods for resizing
7. Dropout

GoogLeNet architecture:



Inception v2:

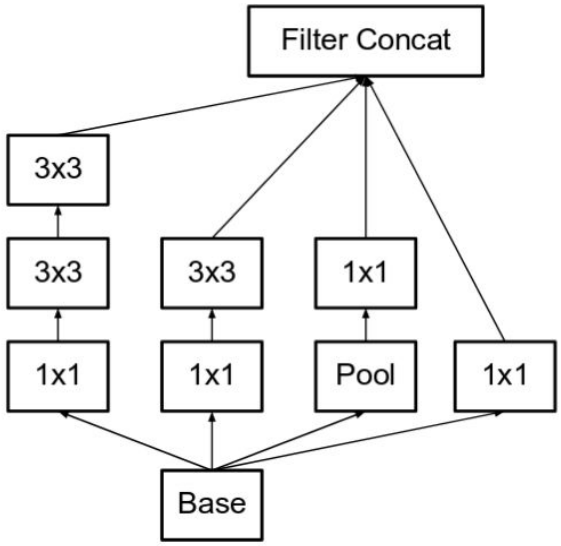
General Design Principles:

1. Avoid representational bottlenecks using extreme compression.
2. Higher dimensional representation are easier to process locally within a network. Increasing the activations per tile in a CNN allows for more disentangled features. As a result the network trains faster.
3. Spatial aggregation can be done on lower dimensional embeddings without much or any loss of information. Because of the strong correlation between adjacent unit results in much less loss of information during dimensional reduction(spatial aggregation context). Hence easily compressible. Also this promotes faster learning.
4. Balancing the depth and width of the network. Increasing both can lead to higher quality networks.

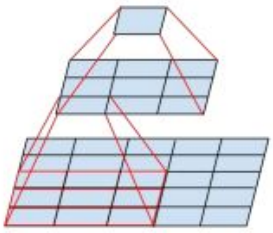
Factorizing Convolutions with Large Filter Size

Suitable factorization (for example, a 5X5 layer into two 3X3 layers stacked upon each other) results in more disentangled(change in one feature causes less or no change in other features) features and therefore faster training.

Factorization into smaller convolutions



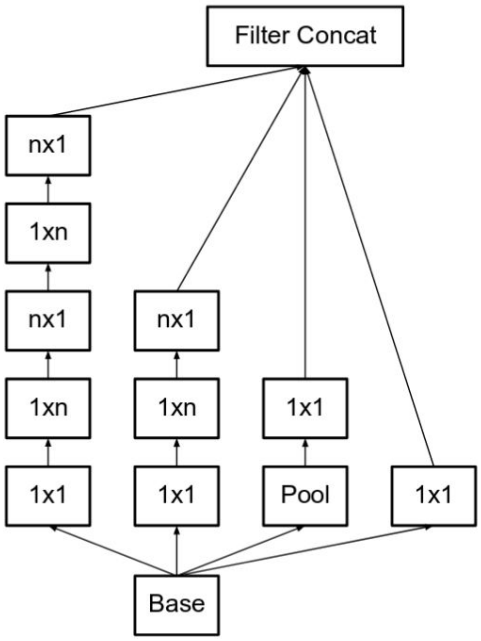
A 5X5 layer can be replaced by two 3X3 convolutional layers stacked on top of each other.



This setup reduces the number of parameters by sharing the weights between two 3X3 layers.

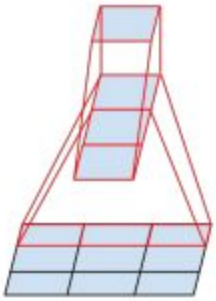
[Figure 5 -> Module A]

Spatial Factorization into Asymmetric Convolutions



[Figure 6 -> Module B]

In theory, we could go even further and argue that one can replace any n × n convolution by a 1 × n convolution followed by a n × 1 convolution and the computational cost saving increases dramatically as n grows.



For example using a 3 × 1 convolution followed by a 1 × 3 convolution is equivalent to sliding a two layer network with the same receptive field as in a 3 × 3 convolution. Still the two-layer solution is 33% cheaper for the same number of output filters, if the number of input and output filters is equal.

Utility of Auxiliary Classifiers

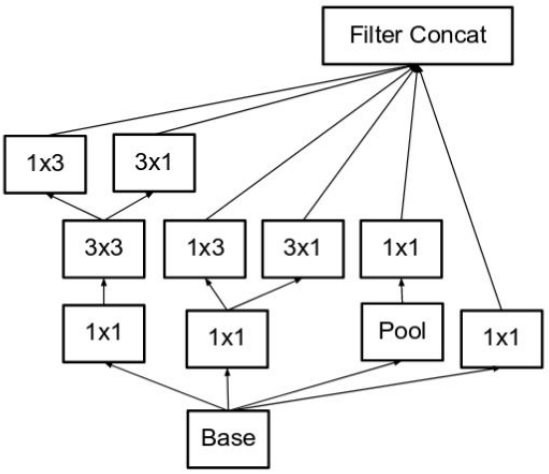
The original motivation was to push useful gradients to the lower layers to make them immediately useful and improve the convergence during training by combating the vanishing gradient problem in very deep networks.

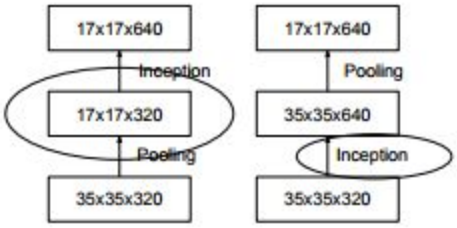
Interestingly, they found that auxiliary classifiers did not result in improved convergence early in the training: the training progression of network with and without side head looks virtually identical before both models reach high accuracy.

Instead, we argue that the auxiliary classifiers act as regularizer. This is supported by the fact that the main classifier of the network performs better if the side branch is batch-normalized or has a dropout layer.

Representation Bottleneck

Traditionally, convolutional networks used some pooling operation to decrease the grid size of the feature maps. In order to avoid a representational bottleneck, before applying maximum or average pooling the activation dimension of the network filters is expanded. For example, starting a d×d grid with k filters, to arrive at a d/2 × d/2 grid with 2k filters, we first need to compute a stride-1 convolution with 2k filters and then apply an additional pooling step. This means that the overall computational cost is dominated by the expensive convolution on the larger grid using $2d^2k^2$ operations. One possibility would be to switch to pooling with convolution and therefore resulting in $2(d/2)^2k^2$, reducing the computational cost by a quarter. However this creates a **representation bottleneck** as the overall dimensionality of the representation drops to $(d/2)^2k$ resulting i**n less expressive networks**.

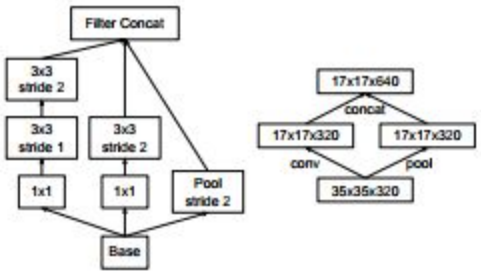




[Figure 7 -> Module C]

The **filter banks** in the module were **expanded** (made wider instead of deeper) to remove the representational bottleneck. If the module was made deeper instead, there would be excessive reduction in dimensions, and hence loss of information. This solution is used on only on the coarsest (8X8) grid, since that is the place where producing high dimensional sparse representation is the most critical as the ratio of local processing (by 1 × 1 convolutions) is increased compared to the spatial aggregation

Efficient Grid Size Reduction



Used two parallel stride 2 blocks: P and C. P is a pooling layer (either average or maximum pooling) the activation, both of them are stride 2 the filter banks of which are concatenated

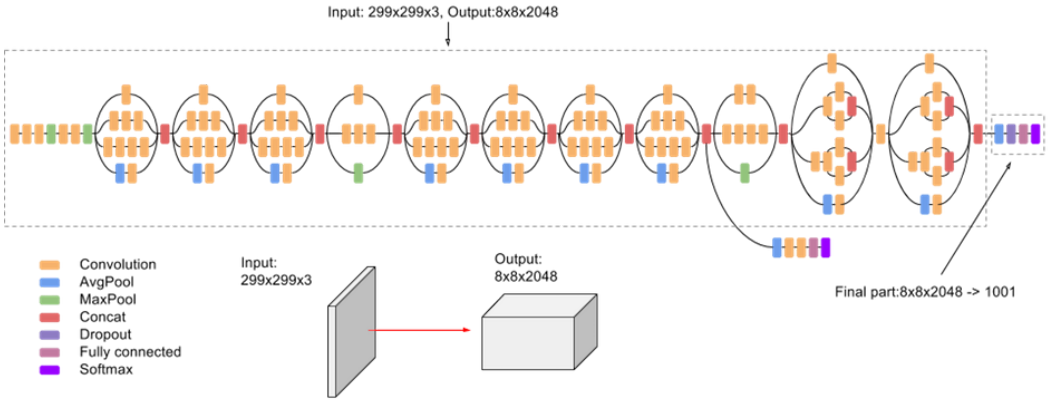
Inception-v2

type	patch size/stride or remarks	input size
conv	3×3/2	299×299×3
conv	3×3/1	149×149×32
conv padded	3×3/1	147×147×32
pool	3×3/2	147×147×64
conv	3×3/1	73×73×64
conv	3×3/2	71×71×80
conv	3×3/1	35×35×192
3×Inception	As in figure 5	35×35×288
5×Inception	As in figure 6	17×17×768
2×Inception	As in figure 7	8×8×1280
pool	8 × 8	8 × 8 × 2048
linear	logits	1 × 1 × 2048
softmax	classifier	1 × 1 × 1000

Inception-v3

Upgrades from Inception-v2:

1. RMSProp Optimizer.
2. Factorized 7x7 convolutions.
3. BatchNorm in the Auxiliary Classifiers.
4. Label Smoothing



ResNet

Is learning better networks is as easy as stacking layers?

As mentioned in the ResNet paper , an obstacle to answering this question was the notorious problem of vanishing /exploding gradients, which hamper convergence from the beginning. However, there has been several attempts to remedy this problem by using input normalization and batch normalization, which enables networks of depth of tens or more to start converging for SGD with Back Propagation.

When deeper networks start converging, a ***degradation*** problem has been exposed: with the network depth increasing, accuracy gets saturated and then degrades rapidly. But unexpectedly, such degradation is not caused by overfitting, and adding more layers to a suitable deep model leads to *higher training error*.

There is a solution by construction to the deeper model: the added layers are identity mapping while the other layers are copied from a shallower model. The existence of this

constructed solution indicates that a deeper model should produce no higher training error than its shallower counterpart.

The authors of the ResNet paper, address the degradation problem by introducing a *deep residual learning* framework. Instead of making the stacked layers fit an underlying mapping, they fitted the layers to a residual mapping.

If the underlying mapping of the stacked layers is H(x) then the stacked layers were fitted to another mapping, which is F(x) := H(x) - x. The original underlying mapping, therefore becomes H(x) = F(x) + x.

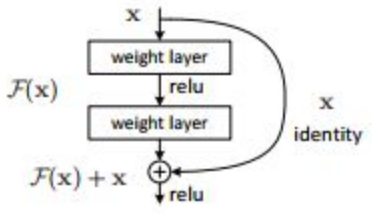
The authors of the ResNet paper hypothesized that it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping. To the extreme, if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers.

The formulation of F(x) + x, has been realized by shortcut connections, and in this case, the shortcut connections simply perform identity mapping, and their outputs are added to the outputs of the stacked layers. Also, these identity connections do not add extra parameters to the network and also doesn't increase the computational complexity.

The degradation problem suggests that the solvers might have difficulties in approximating identity mappings by multiple nonlinear layers. With the residual learning reformulation, if identity mappings are optimal, the solvers may simply drive the weights of the multiple nonlinear layers toward zero to approach identity mappings. However, it it unlikely that identity mappings are optimal.

Let, the building block be defined as:

$y = F(x, \{W_i\}) + x$, where x and y are the input and output vectors of the layers considered. The function F(x,{W_i}) represents the residual learning to be learned.



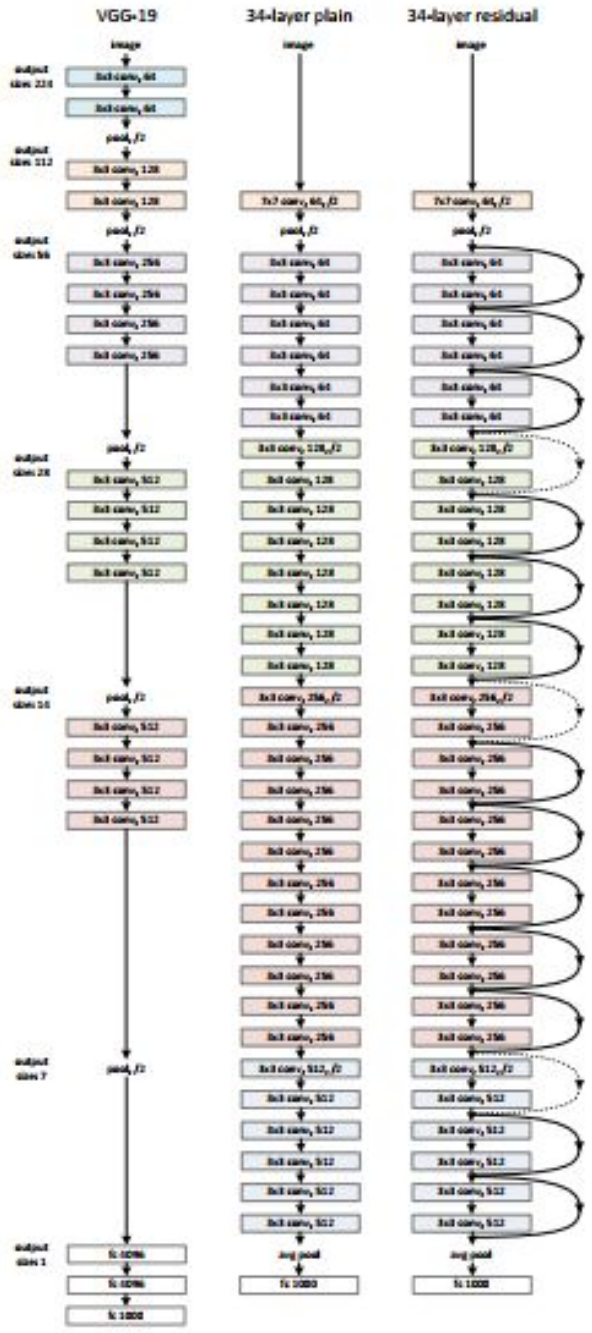
In this figure, there are two layers

$F = W_2\sigma(W_1x)$ in which σ denotes ReLU and the biases are omitted for simplification.

The operation $F + x$ is performed by a shortcut connection and element-wise addition. The second non-linearity is added after the addition, as can be seen in the figure.

However the dimensions of x and F must be equal.If this is not the case, then we have to perform a linear projection on x to match the dimension by multiplication, with W_s

$$y = F(x, \{W_i\}) + W_s.x$$



The authors of the ReNet paper used the 34 layered plain network along with another network which has shortcut connections inserted between every two layers.

They also adopted Batch Normalization after each convolution and before applying ReLU.

Important point to note: Any shallower network is a subspace of a deeper network.

In not-so-deep networks, ResNet eases the optimization by providing faster convergence at the early stage.

In case of shortcut connections between two different dimension layer, the shortcuts can either be zero-padded or the shortcuts can be projection shortcuts.

The authors used three options for increasing dimensions: (A) zero-padding shortcuts, all shortcuts are parameter free (B) projection shortcuts and other shortcuts are identity; and (C) all shortcuts are projections.

B is slightly better than A. This is because the zero-padded dimensions in A indeed have no residual learning. C is marginally better than B, and this is due to the extra parameters introduced by many (thirteen) projection shortcuts. But the small differences among A/B/C indicate that projection shortcuts are not essential for addressing the degradation problem.

Deeper Bottleneck Architectures:

The authors modified the the building block as a bottleneck design. For each residual function F, we use a stack of 3 layers instead of 2. The three layers are 1×1, 3×3, and 1×1 convolutions, where the 1×1 layers are responsible for reducing and then increasing (restoring) dimensions, leaving the 3×3 layer a bottleneck with smaller input/output dimensions.

The parameter-free identity shortcuts are particularly important for the bottleneck architectures. If the identity shortcut in the above figure (right) is replaced with projection, one can show that the time complexity and model size are doubled, as the shortcut is connected to the two high-dimensional ends. So identity shortcuts lead to more efficient models for the bottleneck designs.

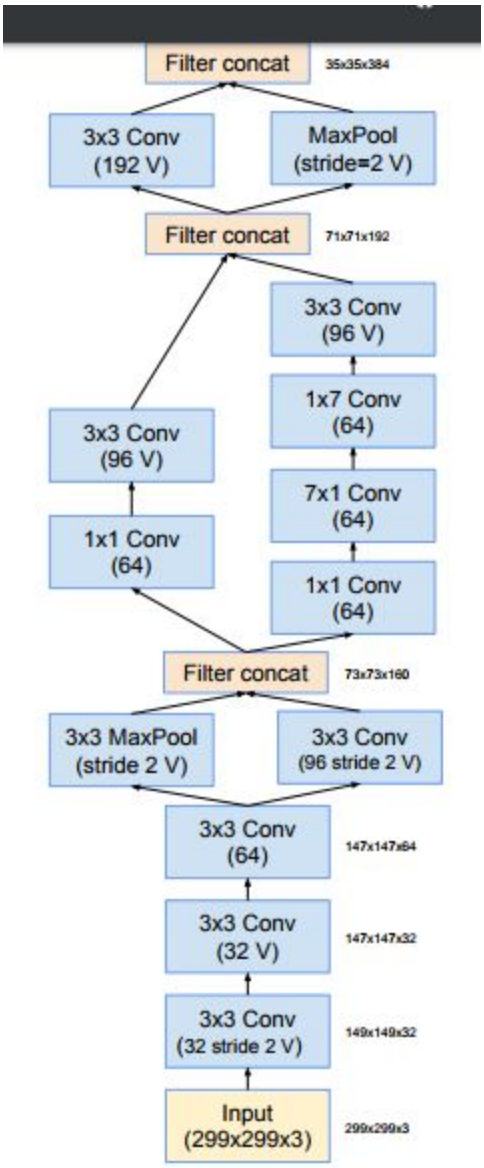
They replaced each 2-layer block in the 34-layer net with this 3-layer bottleneck block, resulting in a 50-layer ResNet. We use option B for increasing dimensions. This model has 3.8 billion FLOPs

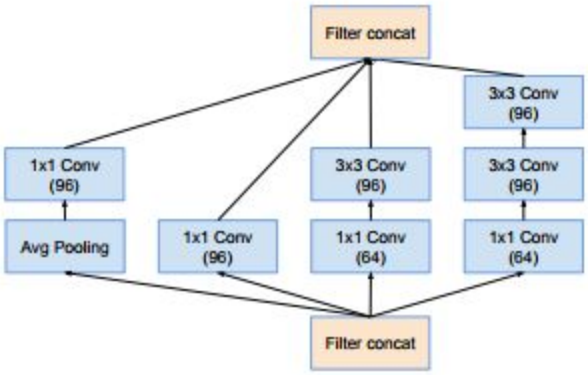
layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Inception-v4:

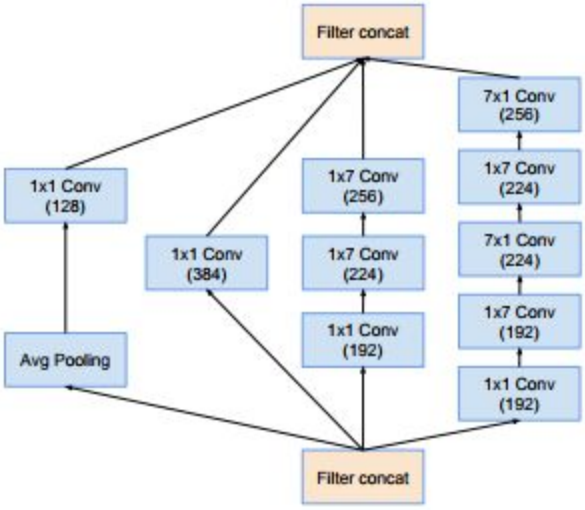
All the figures in this section has been taken from “Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning”. Click here.

The schema for stem of the pure Inception-v4 and Inception-ResNet-v2 networks. This is the input part of those networks

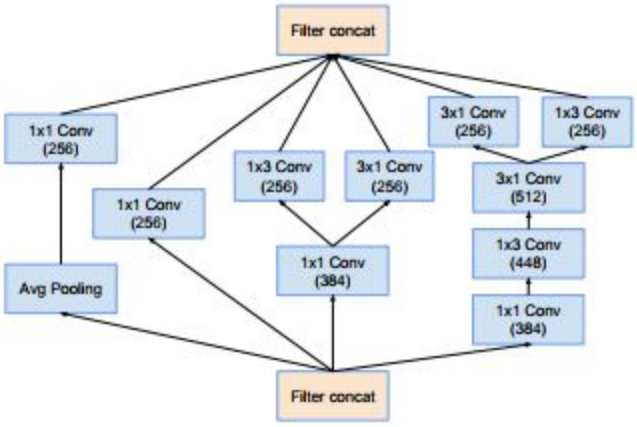




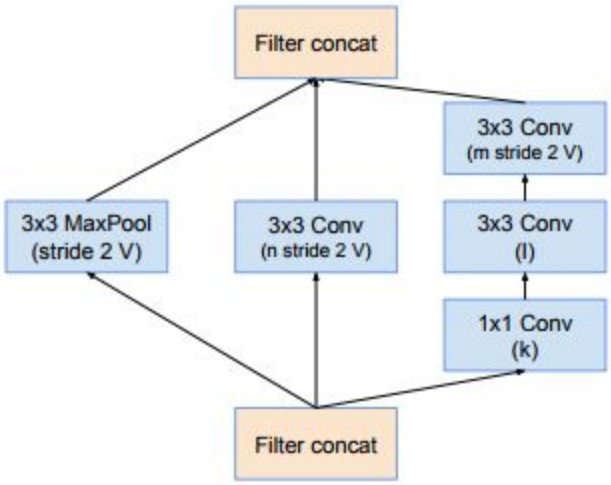
The schema for 35 × 35 grid modules of the pure Inception-v4 network. This is the Inception-A block



The schema for 17 × 17 grid modules of the pure Inception-v4 network. This is the Inception-B block.



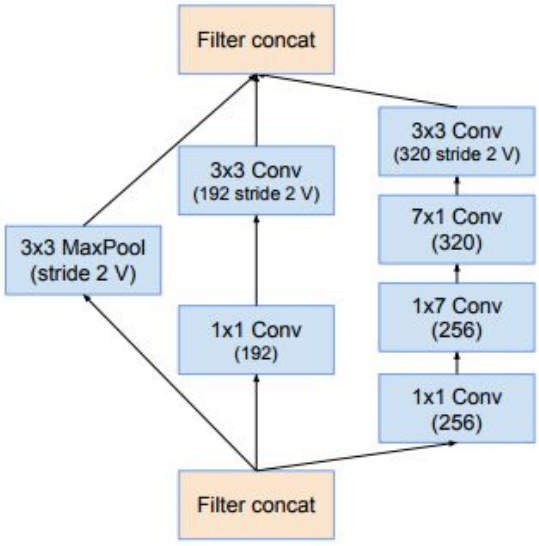
The schema for 8×8 grid modules of the pure Inceptionv4 network. This is the Inception-C block



The schema for 35 × 35 to 17 × 17 reduction module.

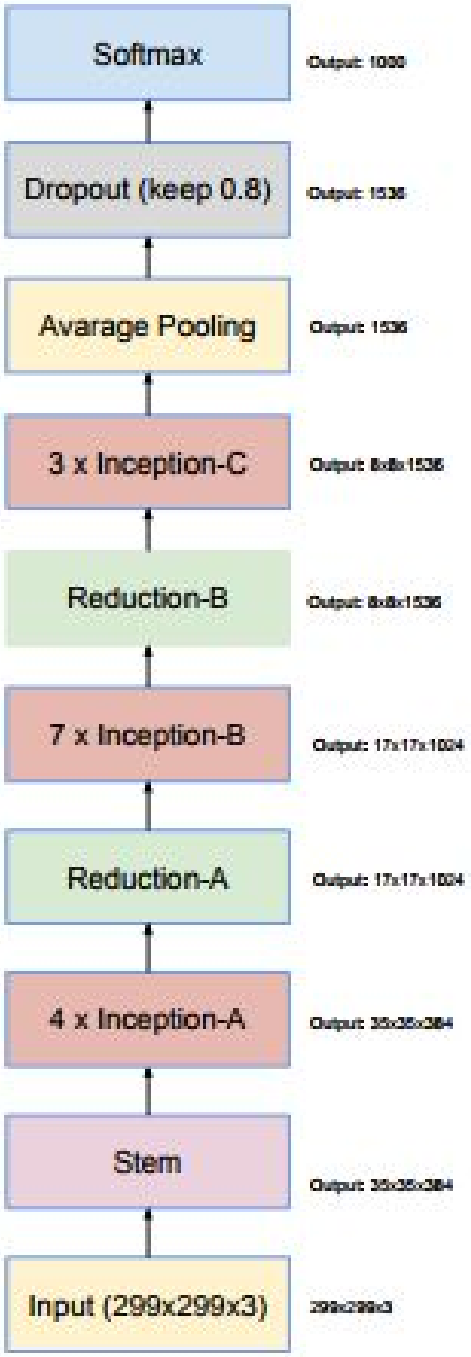
Different variants of this blocks (with various number of filters) are used in each of the new Inception(-v4, - ResNet-v1, -ResNet-v2) variants

The k, l, m, n numbers represent filter bank sizes which can be looked up in Table 1

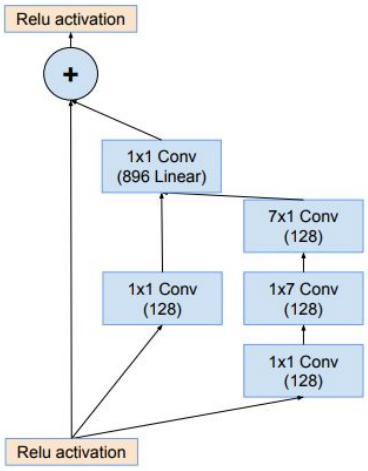


The schema for 17 × 17 to 8 × 8 grid-reduction module. This is the reduction module used by the pure Inception-v4 network

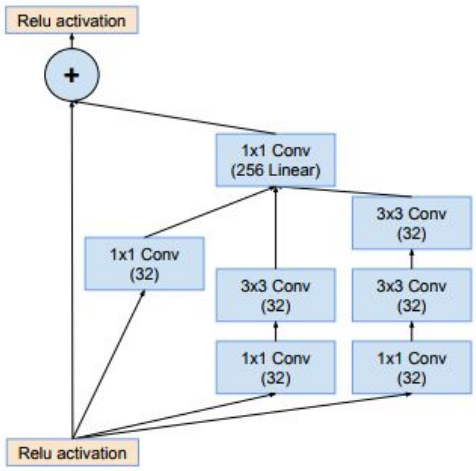
Inception-v4 architecture:



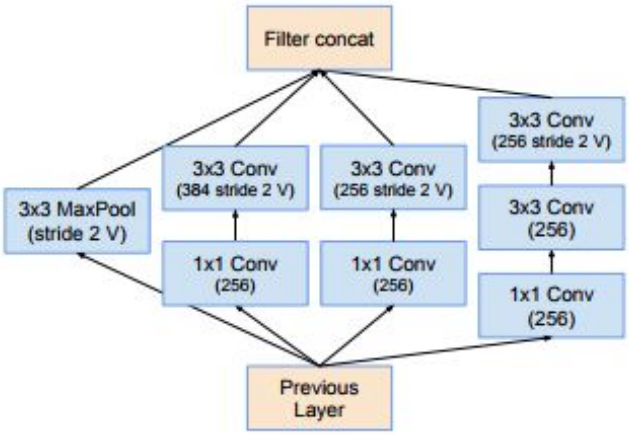
Inception-Resnet-v1:



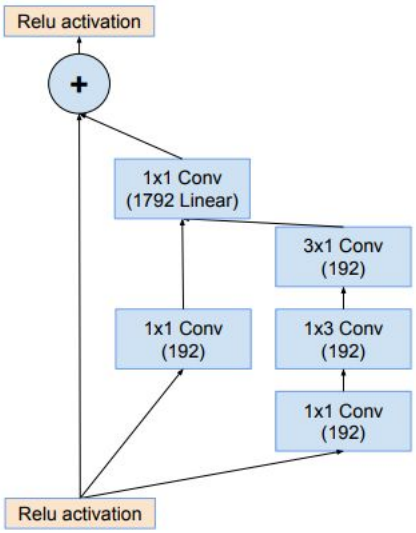
The schema for 17 × 17 grid (Inception-ResNet-B) module of Inception-ResNet-v1 network.



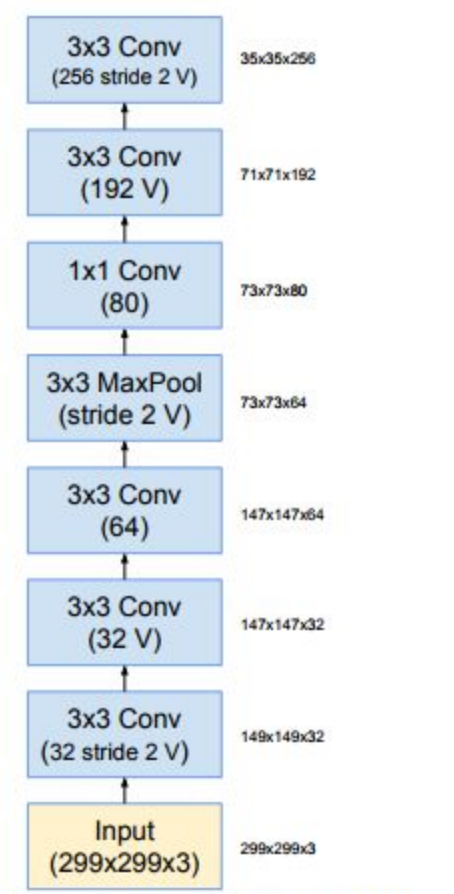
The schema for 35 × 35 grid (Inception-ResNet-A) module of Inception-ResNet-v1 network.



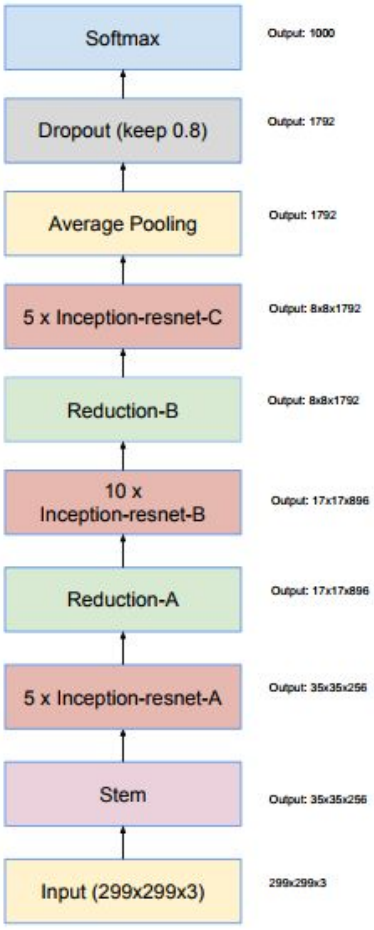
"Reduction-B" 17×17 to 8×8 grid-reduction module. This module used by the smaller Inception-ResNet-v1 network.



The schema for 8×8 grid (Inception-ResNet-C) module of Inception-ResNet-v1 network.

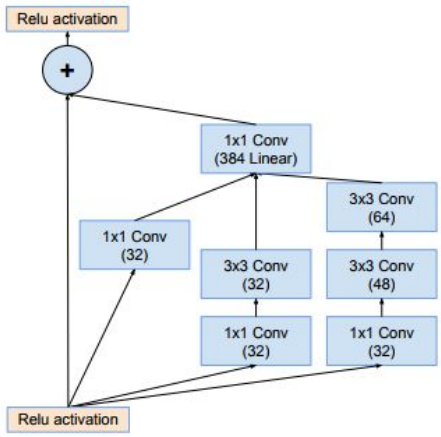


The stem of the Inception-ResNet-v1 network.

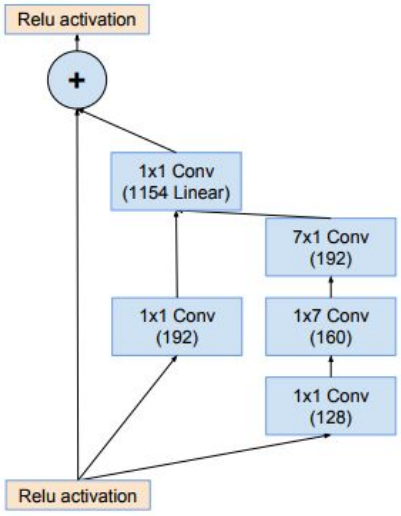


Schema for Inception-ResNet-v1 and Inception-ResNet-v2 networks. This schema applies to both networks but the underlying components differ.

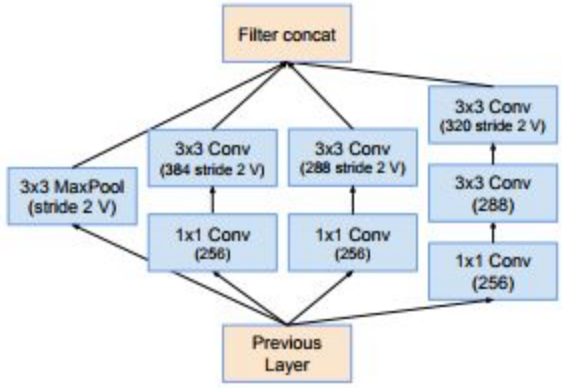
Inception-Resnet-v2:



The schema for 35 × 35 grid (Inception-ResNet-A) module of the Inception-ResNet-v2 network

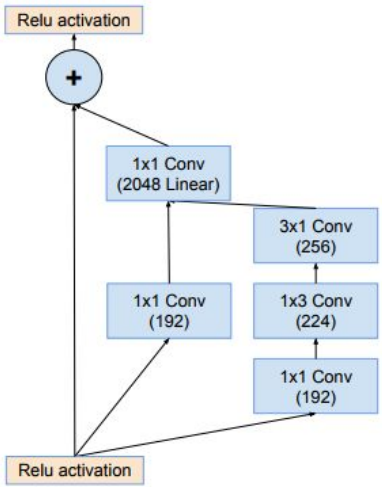


The schema for 17 × 17 grid (Inception-ResNet-B) module of the Inception-ResNet-v2 network.



The schema for 17 × 17 to 8 × 8 grid-reduction module.

The schema for 8×8 grid (Inception-ResNet-C) module of the Inception-ResNet-v2 network



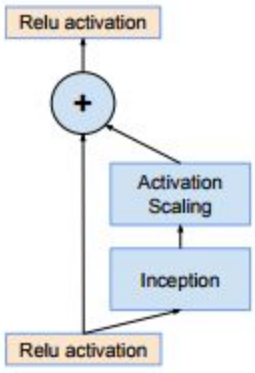
Reduction-B module used by the wider Inception-ResNet-v1 network.

Table-1

Network	k	l	m	n
Inception-v4	192	224	256	384
Inception-Resnet-v1	192	192	256	384
Inception-Resnet-v2	256	256	384	384

Scaling of the Residuals

Also the authors found that if the number of filters exceeded 1000, the residual variants started to exhibit instabilities and the network has just “died” early in the training, meaning that the last layer before the average pooling started to produce only zeros after a few tens of thousands of iterations. This could not be prevented, neither by lowering the learning rate, nor by adding an extra batch-normalization to this layer.



They found that scaling down the residuals before adding them to the previous layer activation seemed to stabilize the training. In general they picked some scaling factors between 0.1 and 0.3 to scale the residuals before their being added to the accumulated layer activations.

Resnet-v2:

Each unit in ResNet can be expressed as

$$y_l = h(x_l) + F(x_l, W_l)$$

$$x_{l+1} = f(y_l)$$

where x_l and x_{l+1} are input and output of the l-th unit, and F is a residual function. In the ResNet paper, $h(x_l) = x_l$ is an identity mapping and f is a ReLU function.

The derivations by the authors reveal that if both $h(x_l)$ and $f(y_l)$ are identity mappings, the signal could be directly propagated from one unit to any other units, in both forward and backward passes. Their experiments empirically show that training in general becomes easier when the architecture is closer to the above two conditions

If ***f*** is also an identity mapping,

$$x_{l+1} = x_l + F(x_l, W_l)$$

Recursively, $x_L = x_l + \sum_{i=l}^{L-1} F(x_i, W_i)$,for *any deeper unit L and any shallower unit l*.

The above equation exhibits some nice properties. (i) The feature x_L of any deeper unit L can be represented as the feature x_l of any shallower unit l plus a residual function in a form of $\sum_{i=1}^{L-1} F$, indicating that the model is in a residual fashion between any units L and l.

The feature $x_L = x_0 + \sum_{i=0}^{L-1} F(x_i, W_i)$ of any deep unit L, is the summation of the outputs of all preceding residual functions (plus x_0). This is in contrast to a "plain network" where a feature x_L is a series of matrix-vector products, say, $\prod_{i=0}^{L-1} W_i x_0$ (ignoring BN and ReLU).

Let L be the loss, then

$$\frac{\partial L}{\partial x_l} = \frac{\partial L}{\partial x_L}\frac{\partial x_L}{\partial x_l} = \frac{\partial L}{\partial x_L}(1 + \frac{\partial}{\partial x_l}\sum_{i=l}^{L-1} F(x_i, W_i))$$

Thus the loss $\frac{\partial L}{\partial x_l}$ can be decomposed into two terms: a term of $\frac{\partial L}{\partial x_L}$ that propagates information directly without concerning any weight layers and another term of $\frac{\partial L}{\partial x_L}(\frac{\partial}{\partial x_l}\sum_{i=l}^{L-1} F(x_i, W_i))$ that propagates through the weight layers. The additive term of $\partial L/\partial x_L$ ensures that information is directly propagated back to any shallower unit l.

The above equation also suggests that it is unlikely for the gradient $\partial L/\partial x_l$ to be canceled out for a mini-batch, because in general the term $\frac{\partial}{\partial x_l}\sum_{i=l}^{L-1} F$ cannot be always -1 for all samples in a mini-batch. This implies that the gradient of a layer does not vanish even when the weights are arbitrarily small.

Importance of Identity Skip connections

If $h(x_l) = \lambda_l x_l$ then from the first two equations we get,

$$x_{l+1} = \lambda_l x_l + F(x_l, W_l)$$

Recursively applying this formulation we get

$$x_L = (\prod_{i=l}^{L-1}\lambda_i)x_l + \sum_{i=l}^{L-1}(\prod_{j=i+1}^{L-1}\lambda_j)F(x_i, W_i) \text{ or, } x_L = (\prod_{i=l}^{L-1}\lambda_i)x_l + \sum_{i=l}^{L-1}\widehat{F}(x_i, W_i)$$

Backpropagation is of the form:

$$\frac{\partial L}{\partial x_l} = \frac{\partial L}{\partial x_L}\frac{\partial x_L}{\partial x_l} = \frac{\partial L}{\partial x_L}((\prod_{j=i+1}^{L-1}\lambda_j) + \frac{\partial}{\partial x_l}\sum_{i=l}^{L-1}\widehat{F}(x_i, W_i))$$

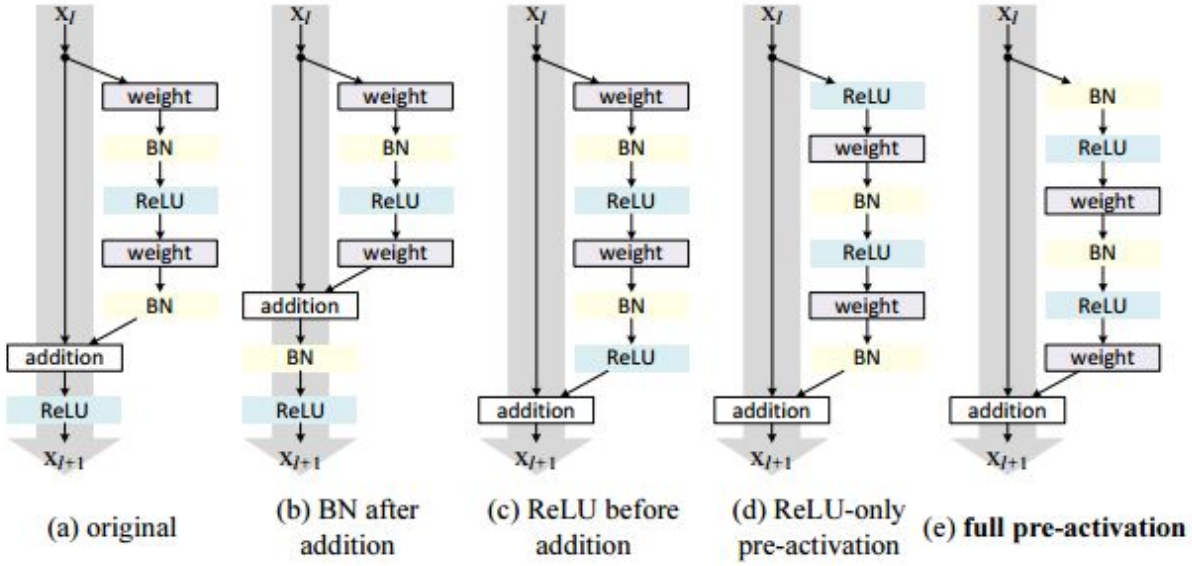
If $\lambda_i > 1$ then the factor is exponentially large

If $\lambda_i < 1$ then the factor is exponentially small and vanish, which blocks the backpropagated signal from the shortcut and forces it to flow through the weight layers. This results in optimization difficulties.

Usage of Activation Functions

The above derivations were carried out under the assumptions that the activation function is also an identity mapping.

To make the ***f*** an identity mapping, the activation functions BN and ReLU are re-arranged.



Post-activation or pre-activation?

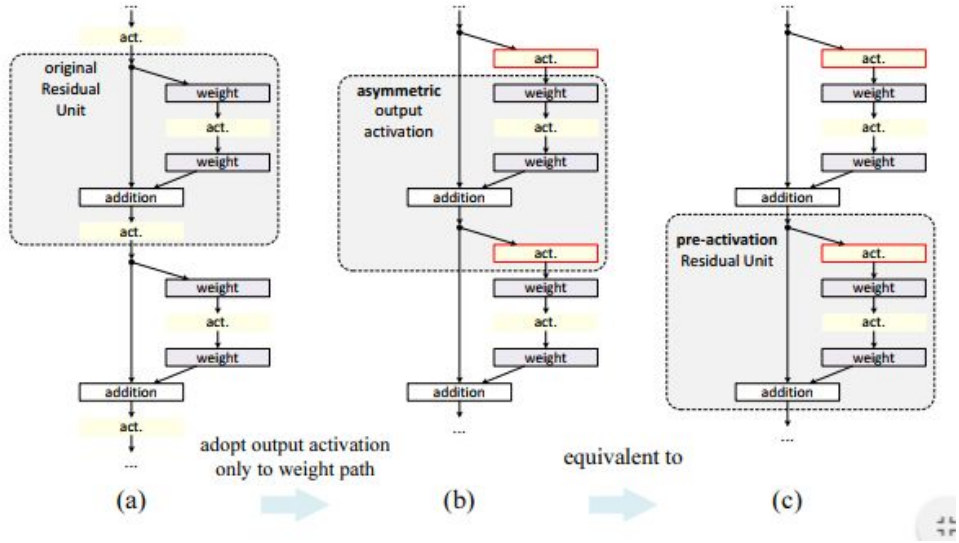
In the original design , the activation $x_{l+1} = f(y_l)$ affects both paths in the next Residual Unit: $y_{l+1} = f(y_l) + F(f(y_l), W_l)$.

Next they developed an asymmetric form where an activation $\widehat{f}$ only affects the F path: $y_{l+1} = y_l + F(\widehat{f}(y_l), W_l)$, for any l.

By renaming the notations, we have the following form: $x_{l+1} = x_l + F(\widehat{f}(x_l), W_l)$

It is easy to see that the above equation is similar to 3rd equation, and can enable a backward formulation.

For this new Residual Unit, the new after-addition activation becomes an identity mapping. This design means that if a new after-addition activation $\widehat{f}$ is asymmetrically adopted, it is equivalent to recasting $\widehat{f}$ as the pre-activation of the next Residual Unit.

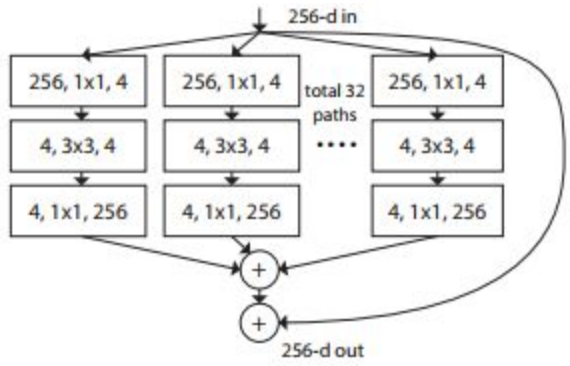


Using asymmetric after-addition activation is equivalent to constructing a pre-activation Residual Unit

ResNeXt

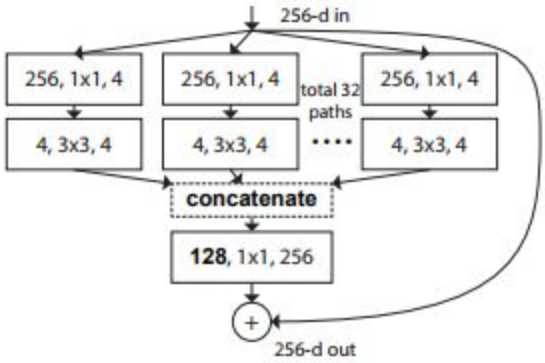
ResNeXt adopts he VGG/ResNet strategy of repeating layers exploiting the split-transform-merge strategy in an easy and extensible way.

A module in ResNext network performs a set of transformations, each on a low-dimensional embedding, whose outputs are aggregated by summation. The transformations to be aggregated are all of the same topology.



Aggregated residual transformations

Equivalent formulations similar to Inception-Resnet



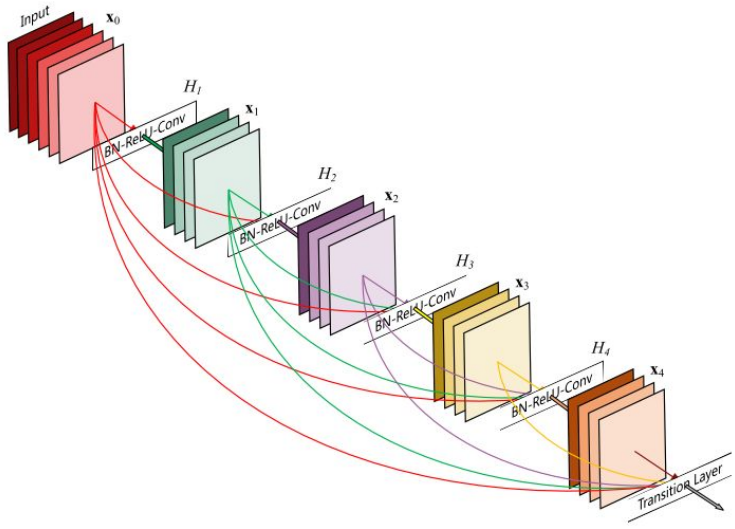
The authors introduces another hyper-parameter called cardinality, which is the size of the set of transformations, or the number of paths. Experiments demonstrate that increasing cardinality is a more effective way of gaining accuracy than going deeper or wider, especially when depth and width starts to give diminishing returns for existing models.

DenseNet

Dense Convolutional Network (DenseNet), connects each layer to every other layer in a feed-forward fashion. Whereas traditional convolutional networks with L layers have L connections—one between each layer and its subsequent layer—our network has L(L+1)/2 direct connections. For each layer, the feature-maps of all preceding layers are used as inputs, and its own feature-maps are used as inputs into all subsequent layers. DenseNets have several compelling advantages: they alleviate the vanishing-gradient problem, strengthen feature propagation, encourage feature reuse, and substantially reduce the number of parameters.

Crucially, in contrast to ResNets, the features are not combined through summation before they are passed into a layer; instead, the features are combined by concatenating them. Hence, the l^{th} layer has l inputs, consisting of the feature-maps of all preceding convolutional blocks. Its own feature-maps are passed on to all (L− l) subsequent layers. This introduces L(L+1)/2 connections in an L-layer network, instead of just L, as in traditional architecture.

The network comprises L layers, each of which implements a non-linear transformation $H_l(\cdot)$, where l indexes the layer. $H_l(\cdot)$ can be a composite function of operations such as Batch Normalization (BN), rectified linear units (ReLU), Pooling, or Convolution (Conv). We denote the output of the l^{th} layer as x_l.



The adjacent figure illustrates the layout of the resulting DenseNet schematically.

Consequently the l^{th} layer receives the feature-maps of all preceding layers, $x_0,...,x_{l-1}$, as input:

$X_l = H_l([x_0,x_1,....,x_{l-1}])$, where $[x_0,x_1,....,x_{l-1}]$ refers to the concatenation of the feature-maps produced in the layers 0,....l-1.

If each function $H_l(.)$ produces k feature maps, it follows that the l^{th} layer has $k_0 + k \times (l-1)$ input feature-maps, where k_0 is the number of channels in the input layer. An important difference between DenseNet and existing network architectures is that DenseNet can have very narrow layers, e.g., k = 12. The authors refer to the hyperparameter **k** as the growth rate of the network.

Below is shown the DenseNet architectures for ImageNet. The growth rate for all the networks is k=32. Each "conv" layer shown corresponds the sequence BN-ReLU-Conv

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv, 3 × 3 conv] × 24	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48	[1 × 1 conv, 3 × 3 conv] × 64
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv, 3 × 3 conv] × 16	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

Applications of ConvNet:

Object Detection

RCNN and its variants

RCNN

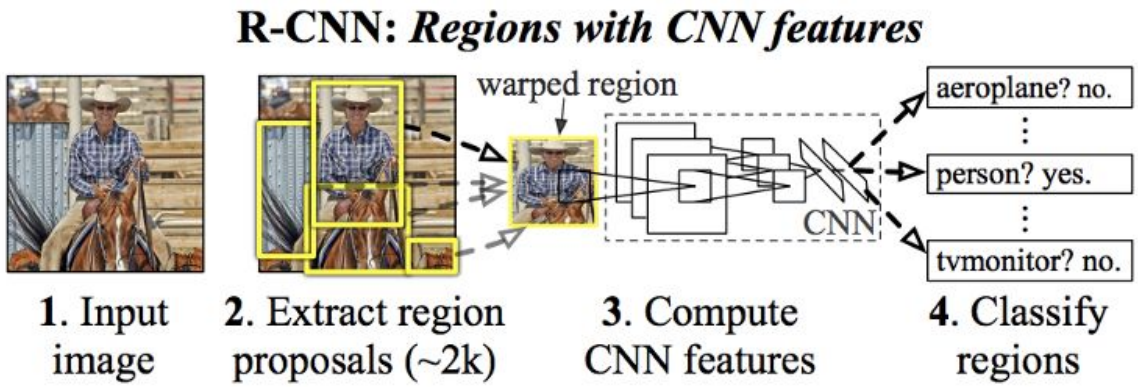
Module Design

Region Proposals: Selective Search method was used to obtain region proposals in R-CNN.
Selective Search:

1. Generate initial sub-segmentation, we generate many candidate regions

2. Use greedy algorithm to recursively combine similar regions into larger ones

3. Use the generated regions to produce the final candidate region proposals



Object Proposal Transformations

The first method (“tightest square with context”) encloses each object proposal inside the tightest square and then scales (isotropically) the image contained in that square to the CNN input size. A variant on this method (“tightest square without context”) excludes the image content that surrounds the original object proposal.
The second method (“warp”) anisotropically scales each object proposal to the CNN input size.

Feature Extraction

Extract a 4096-dimensional feature vector from each region proposal using AlexNet. Features are computed by forward propagating a mean-subtracted 227 × 227 RGB image through five convolutional layers and two fully connected layers.

Test-Time detection

At test time, run selective search on the test image to extract around 2000 region proposals (we use selective search's "fast mode" in all experiments). We warp each proposal and forward propagate it through the CNN in order to compute features. Then, for each class, we score each extracted feature vector using the SVM trained for that class. Given all scored regions in an image, we apply a greedy non-maximum suppression (for each class independently) that rejects a region if it has an intersection-over-Union (IoU) overlap with a higher threshold scoring selected region larger than a learned threshold.

Training

1. Supervised pre-training of CNN on Imagenet dataset.
2. Domain specific fine-tuning to adapt the CNN to the new task(detection) and the new domain(warped proposal windows), using SGD. Aside from replacing the CNN's ImageNetspecific 1000-way classification layer with a randomly initialized (N + 1)-way classification layer (where N is the number of object classes, plus 1 for background), the CNN architecture is unchanged. Treat all region proposals with ≥ 0.5 IoU overlap with a ground-truth box as positives for that box's class and the rest as negatives.
3. Object Category Classifiers: Once features are extracted and training labels are applied, optimize one linear SVM per class. Since the training data is too large to fit in memory, the standard hard negative mining method was adopted.
4. Bounding Box Regression: We use a simple bounding-box regression stage to improve localization performance. After scoring each selective search proposal with a class-specific detection SVM, we predict a new bounding box for the detection using a class-specific bounding-box regressor.
   The input to our training algorithm is a set of N training pairs $\{(P_i , G_i)\}$ $i=1,\ldots,N$, where $P_i = (P_i^x , P_i^y , P_i^w, P_i^h)$ specifies the pixel coordinates of the center of proposal

P_i 's bounding box together with P_i 's width and height in pixels. Hence forth, we drop the superscript i unless it is needed.

Each ground-truth bounding box G is specified in the same way: $G = (G_x, G_y, G_w, G_h)$. The goal is to learn a transformation that maps a proposed box P to a ground-truth box G.

We parameterize the transformation in terms of four functions $d_x(P)$, $d_y(P)$, $d_w(P)$, and $d_h(P)$. The first two specify a scale-invariant translation of the center of P's bounding box, while the second two specify log-space translations of the width and height of P's bounding box.

After learning these functions, we can transform an input proposal P into a predicted ground-truth box $\hat{G}$ by applying the transformation

$\hat{G}_x = P_w d_x(P) + P_x$

$\hat{G}_y = P_h d_y(P) + P_y$

$\hat{G}_w = P_w \exp(d_w(P))$

$\hat{G}_h = P_h \exp(d_h(P))$

Each function $d_*(P)$ (where * is one of x, y, h, w) is modeled as a linear function of the pool5 features of proposal P, denoted by $\varphi_5(P)$. (The dependence of $\varphi_5(P)$ on the image data is implicitly assumed.)

Thus we have $d_*(P) = w^T_* \varphi_5(P)$, where w_* is a vector of learnable model parameters.

We learn w_* by optimizing the regularized least squares objective (ridge regression):

$$w_* = argmin_{\hat{w}_*} \sum_i^N (t^i_* - \hat{w}^T_* \phi_5(P^i))^2 + \lambda ||\hat{w}_*||^2$$

The regression targets t_* for the training pair (P, G) are defined as

$t_x = (G_x - P_x)/P_w$

$t_y = (G_y - P_y)/P_h$

$t_w = \log(G_w/P_w)$

$t_h = \log(G_h/P_h)$

As a standard regularized least squares problem, this can be solved efficiently in closed form.

5. Positive vs. negative examples and softmax

Two design choices warrant further discussion. The first is: Why are positive and negative examples defined differently for fine-tuning the CNN versus training the object detection SVMs? To review the definitions briefly, for finetuning the authors

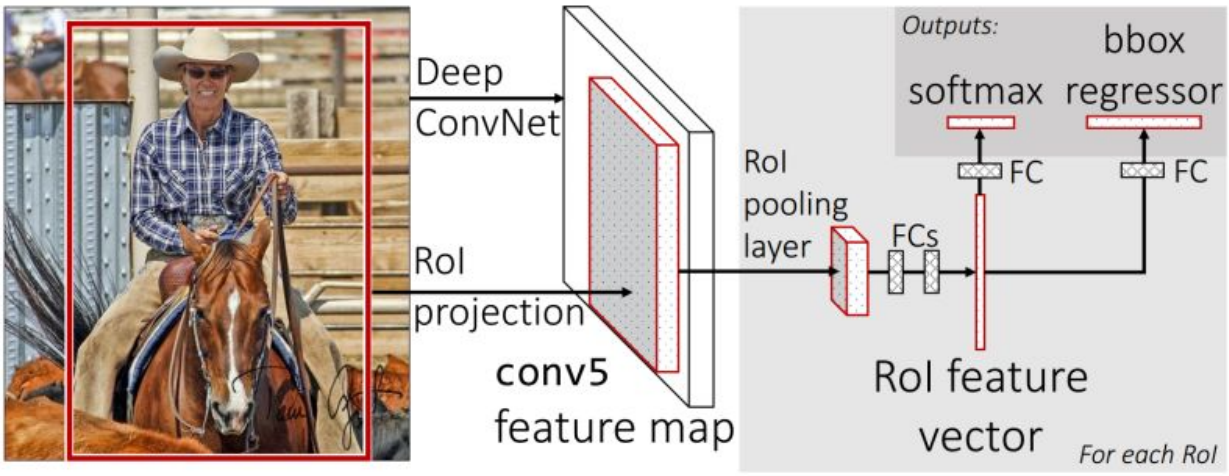
mapped each object proposal to the ground-truth instance with which it has maximum IoU overlap (if any) and label it as a positive for the matched ground-truth class if the IoU is at least 0.5. All other proposals are labeled “background” (i.e., negative examples for all classes). For training SVMs, in contrast, the authors took only the ground-truth boxes as positive examples for their respective classes and label proposals with less than 0.3 IoU overlap with all instances of a class as a negative for that class. Proposals that fall into the grey zone (more than 0.3 IoU overlap, but are not ground truth) are ignored.

When the authors started using fine-tuning, we initially used the same positive and negative example definition as we were using for SVM training. However, we found that results were much worse than those obtained using our current definition of positives and negatives. The authors also noted that using these jittered (those proposals with overlap between 0.5 and 1, but not ground truth) examples is likely suboptimal because the network is not being fine-tuned for precise localization.

This leads to the second issue: Why, after fine-tuning, train SVMs at all? It would be cleaner to simply apply the last layer of the fine-tuned network, which is a 21-way softmax regression classifier, as the object detector. The authors tried this and found that performance on VOC 2007 dropped from 54.2% to 50.9% mAP. This performance drop likely arises from a combination of several factors including that the definition of positive examples used in fine-tuning does not emphasize precise localization and the softmax classifier was trained on randomly sampled negative examples rather than on the subset of “hard negatives” used for SVM training.

Fast-RCNN

Architecture



Training

A Fast R-CNN network takes as input an entire image and a set of object proposals. The network first processes the whole image with several convolutional (conv) and max pooling layers to produce a conv feature map. Then, for each object proposal(using selective search) a region of interest (RoI) pooling layer extracts a fixed-length feature vector from the feature map. Each feature vector is fed into a sequence of fully connected (fc) layers that finally branch into two sibling output layers: one that produces softmax probability estimates over K object classes plus a catch-all "background" class and another layer that outputs four real-valued numbers for each of the K object classes. Each set of 4 values encodes refined bounding-box positions for one of the K classes.

1. **The RoI pooling Layer**: The RoI pooling layer uses max pooling to convert the features inside any valid region of interest into a small feature map with a fixed spatial extent of HXW, where H and W are also hyper-parameters, independent of any particular RoI. Each RoI is defined by a four-tuple (r,c,h,w), where (r,c) is the top-left corner of the region and (h,w) is its height and width. RoI max pooling works by dividing the h × w RoI window into an H × W grid of sub-windows of approximate size h/H × w/W and then max-pooling the values in each sub-window into the

corresponding output grid cell. Pooling is applied independently to each feature map channel, as in standard max pooling.

2. **Initializing from pre-trained networks**: The experimented with three pre-trained networks, each with five max-pooling layers and between five and thirteen convolutional layers. The three transformations that Fast R-CNN undergoes according to the author is:

   A. The last max-pooling layer is replaced by RoI pooling layer that is configured by setting H and W to be compatible with the net's fully connected layer(e.g H=W=7 for VGG16).

   B. The network's last fully connected layer and softmax are replaced with two sibling layers described earlier (a fully connected layer and softmax over K + 1 categories and category-specific bounding-box regressors).

   C. Third, the network is modified to take two data inputs: a list of images and a list of RoIs in those images.

3. **Fine-tuning for detection**: In Fast RCNN training, stochastic gradient descent (SGD) mini batches are sampled hierarchically, first by sampling N images and then by sampling R/N RoIs from each image. Critically, RoIs from the same image share computation and memory in the forward and backward passes. Making N small decreases mini-batch computation.

Multi-task Loss:

A Fast R-CNN network has two sibling output layers. The first outputs a discrete probability distribution (per RoI), $p = (p_0,....,p_K)$, over K+1 categories. As usual, p is computed by a softmax over the K+1 outputs of a fully connected layer. The second sibling layer outputs bounding-box regression offsets, $t^k = (t^k_x,t^k_y,t^k_w,t^k_h)$, for each of the K object classes, indexed by k. We use the parameterization for t^k given in the literature for R-CNN., in which t^k specifies a scale-invariant translation and log-space height/width shift relative to an object proposal.

Each training RoI is labeled with a ground-truth class u and a ground-truth bounding-box regression target v. We use a multi-task loss L on each labeled RoI to jointly train for classification and bounding-box regression:

$$L(p, u, t^u, v) = L_{cls}(p, u) + \lambda[u \geq 1] L_{loc}(t^u, v)$$

In which $L_{cls}(p,u) = -\log p_u$ is the log loss for true class u.

The second task loss, L_{loc}, is defined over a tuple of true bounding-box regression targets for class u, $v = (v_x, v_y, v_w, v_h)$, and a predicted tuple $t^u = (t^u_x, t^u_y, t^u_w, t^u_h)$, again for class u. The Iverson bracket indicator function [u ≥ 1] evaluates to 1 when u ≥ 1 and 0 otherwise. By convention the catch-all background class is labeled u = 0. For background RoIs there is no notion of a ground-truth bounding box and hence L_{loc} is ignored. For bounding-box regression, we use the loss

$$L_{loc}(t^u, v) = \sum_{i \in \{x,y,w,h\}} smooth_{L_1}(t_i^u - v_i) \text{ in which}$$

$$smooth_{L_1}(x) = 0.5x^2 \text{ if } |x| < 1 \text{ and } |x| - 0.5 \text{ otherwise}$$

The hyper-parameter λ in Eq. 1 controls the balance between the two task losses. The author normalized the ground-truth regression targets v_i to have zero mean and unit variance. The author used λ = 1 in all experiments.

Fast R-CNN Detection Procedure

The network takes an input image and a list of R object proposals to score at test-time

R is typically around 2000

When using an image pyramid, each RoI is assigned to the scale such that the scaled RoI is closest to 224^2 pixels in area.

For each test RoI r, the forward pass outputs a class posterior probability distribution p and a set of predicted bounding-box offsets relative to r

Each of the K classes gets its own refined bounding-box prediction

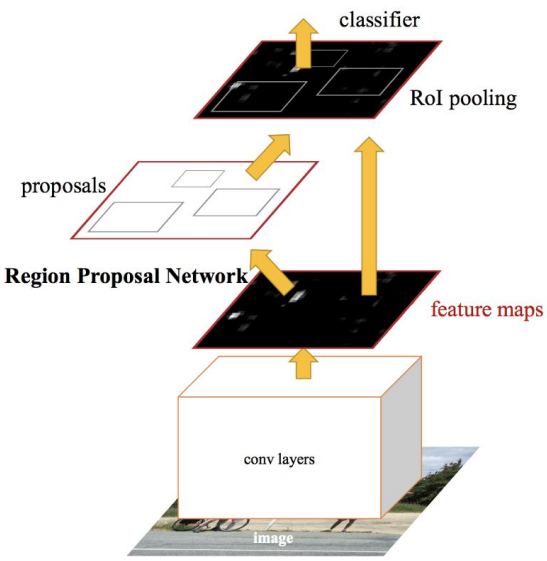
A detection confidence is assigned to r for each object class k using the estimated probability $Pr(class = k \mid r) = p_k$

Then perform non-maximum suppression independently for each class using the algorithm and settings from R-CNN

Faster-RCNN

Instead of using selective search algorithm on the feature map to identify the region proposals, a separate network is used to predict the region proposals.

The authors observed that the convolutional feature maps used by region-based detectors, like Fast RCNN, can also be used for generating region proposals. On top of these convolutional features, they constructed an RPN by adding a few additional convolutional layers that simultaneously regress region bounds and objectness scores at each location on a regular grid. The RPN is thus a kind of fully convolutional network (FCN) and can be trained end-to-end specifically for the task for generating detection proposals.



The faster R-CNN is composed of two modules. The first module is a deep fully convolutional network, that proposes regions. The second module is the Fast R-CNN Detector that uses the proposed regions.

Region Proposal Networks

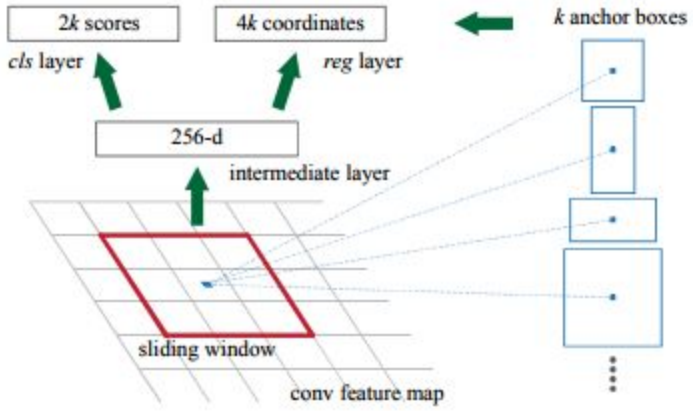
To generate region proposals:

1. Slide a small network over the convolutional feature map output by the last shared convolutional layer. This small network takes as input an n × n spatial window of the input convolutional feature map.
2. Each sliding window is mapped to a lower-dimensional feature.
3. This feature is fed into two sibling fully-connected layers—a box-regression layer (reg) and a box-classification layer.
4. This architecture is naturally implemented with an n×n convolutional layer followed by two sibling 1 × 1 convolutional layers (for reg and cls, respectively).

Anchors

1. At each sliding-window location, we simultaneously predict multiple region proposals, where the number of maximum possible proposals for each location is denoted as k.
2. So the reg layer has 4k outputs encoding the coordinates of k boxes
3. And the cls layer outputs 2k scores that estimate probability of object or not object for each proposal.
4. The k proposals are parameterized relative to k reference boxes, which we call *anchors.*

An anchor is centered at the sliding window in question, and is associated with a scale and aspect ratio.



The anchors are also translation invariant. If one translates an object in an image, the proposal should translate and the same function should be able to predict the proposal in either location.

This anchor-based method is built on a pyramid of anchors, which is more cost-efficient. This method classifies and regresses bounding boxes with reference to anchor boxes of multiple scales and aspect ratios. It only relies on images and feature maps of a single scale, and uses filters (sliding windows on the feature map) of a single size.

Loss Function

For training RPNs, the authors assigned a binary class label (of being an object or not) to each anchor. We assign a positive label to two kinds of anchors: (i) the anchor/anchors with the highest Intersection-over-Union (IoU) overlap with a ground-truth box, or (ii) an anchor

that has an IoU overlap higher than 0.7 with 5 any ground-truth box. Note that a single ground-truth box may assign positive labels to multiple anchors. Usually the second condition is sufficient to determine the positive samples; but the authors still adopt the first condition for the reason that in some rare cases the second condition may find no positive sample. The authors assigned a negative label to a non-positive anchor if its IoU ratio is lower than 0.3 for all ground-truth boxes. Anchors that are neither positive nor negative do not contribute to the training objective.

The loss function for an image is defined as

$$L(\{p_i\}, \{t_i\}) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i^*) + \lambda \frac{1}{N_{reg}} \sum_i p_i^* L_{reg}(t_i, t_i^*)$$

Here, i is the index of an anchor in a mini-batch and pi is the predicted probability of anchor i being an object. The ground-truth label p^*_i is 1 if the anchor is positive, and is 0 if the anchor is negative. t_i is a vector representing the 4 parameterized coordinates of the predicted bounding box, and t^*_i is that of the ground-truth box associated with a positive anchor. The classification loss Lcls is log loss over two classes (object vs. not object). For the regression loss, we use $L_{reg}(t_i , t^*_i) = R(t_i - t^*_i)$ where R is the robust loss function (smooth L1) defined in Fast R-CNN.

The term $p^*_i L_{reg}$ means the regression loss is activated only for positive anchors ($p^*_i = 1$) and is disabled otherwise ($p^*_i = 0$). The outputs of the cls and reg layers consist of $\{p_i\}$ and $\{t_i\}$ respectively.

The two terms are normalized by N_{cls} and N_{reg} and weighted by a balancing parameter λ.

For bounding box regression, adopt the parameterizations of the 4 coordinates following R-CNN.

Training RPNs

The RPN can be trained end-to-end by backpropagation and stochastic gradient descent. Each mini-batch arises from a single image that contains many positive and negative example anchors. Randomly sample 256 anchors in an image to compute the loss function of a mini-batch, where the sampled positive and negative anchors have a ratio of up to 1:1.

In this paper, the authors adopted a pragmatic 4-step training algorithm to learn shared features via alternating optimization. In the first step, they trained the RPN as described in the last paragraph.(Section 3.1.3 of the Faster R-CNN paper). This network is initialized with an ImageNet-pre-trained model and fine-tuned end-to-end for the region proposal task. In the second step, they trained a separate detection network by Fast R-CNN using the proposals generated by the step-1 RPN. This detection network is also initialized by the ImageNet-pre-trained model. At this point the two networks do not share convolutional layers. In the third step, they used the detector network to initialize RPN training, but they fix the shared convolutional layers and only fine-tune the layers unique to RPN. Now the two networks share convolutional layers. Finally, keeping the shared convolutional layers fixed, they fine-tuned the unique layers of Fast R-CNN. As such, both networks share the same convolutional layers and form a unified network.

Some RPN proposals highly overlap with each other. To reduce redundancy, the authors adopted non-maximum suppression (NMS) on the proposal regions based on their cls scores. They fixed the IoU threshold for NMS at 0.7, which gave about 2000 proposal regions per image. After NMS, they used the top-N ranked proposal regions for detection.

YOLO (You Only Look Once)

Important Features of YOLO:

A single convolutional network simultaneously predicts multiple bounding boxes and class probabilities for those boxes. YOLO trains on full images and directly optimizes detection performance.

First, YOLO is extremely fast. Since we frame detection as a regression problem we don't need a complex pipeline. We simply run our neural network on a new image at test time to predict detections. Furthermore, YOLO achieves more than twice the mean average precision of other real-time systems.

Second, YOLO reasons globally about the image when 1 arXiv:1506.02640v5 [cs.CV] 9 May 2016 making predictions. Unlike sliding window and region proposal-based techniques, YOLO sees the entire image during training and test time so it implicitly encodes contextual information about classes as well as their appearance. YOLO makes less than half the number of background errors compared to Fast R-CNN.

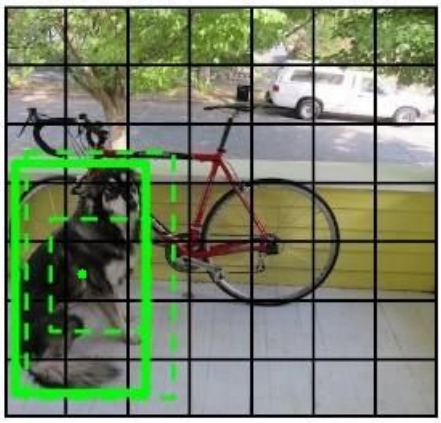
Third, YOLO learns generalizable representations of objects. When trained on natural images and tested on artwork, YOLO outperforms top detection methods like DPM and R-CNN by a wide margin. Since YOLO is highly generalizable it is less likely to break down when applied to new domains or unexpected inputs.

YOLO still lags behind state-of-the-art detection systems in accuracy. While it can quickly identify objects in images it struggles to precisely localize some objects, especially small ones.

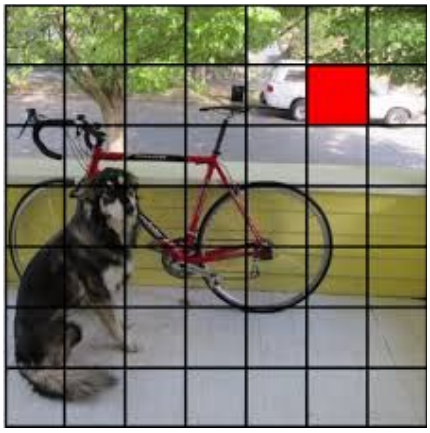
Unified Detection

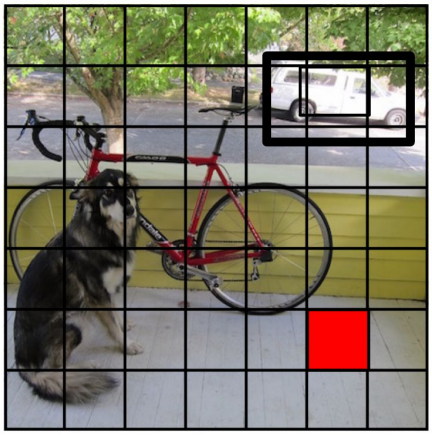
1. Divide the Image into SXS grid.

2. For each grid, if the centre of an object falls into a grid cell, that cell is responsible for detecting that object.



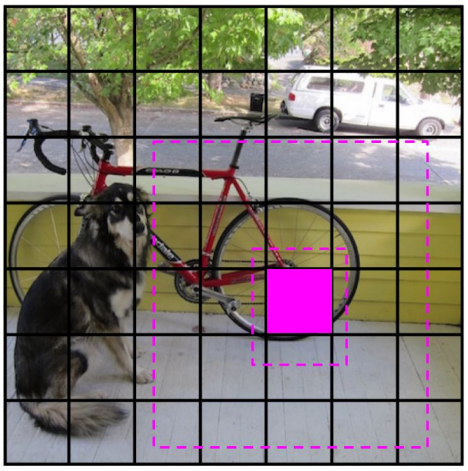
3. Each grid cell predicts B bounding boxes and confidence scores for those boxes. These confidence scores reflect how confident the model is that the box contains an object and also how accurate it thinks the box is that it predicts.





Confidence is defined as $Pr(Object) * IOU^{truth}_{pred}$.

4. If no object exists in that cell, the confidence scores should be zero. Otherwise we want the confidence score to equal the intersection over union (IOU) between the predicted box and the ground truth.

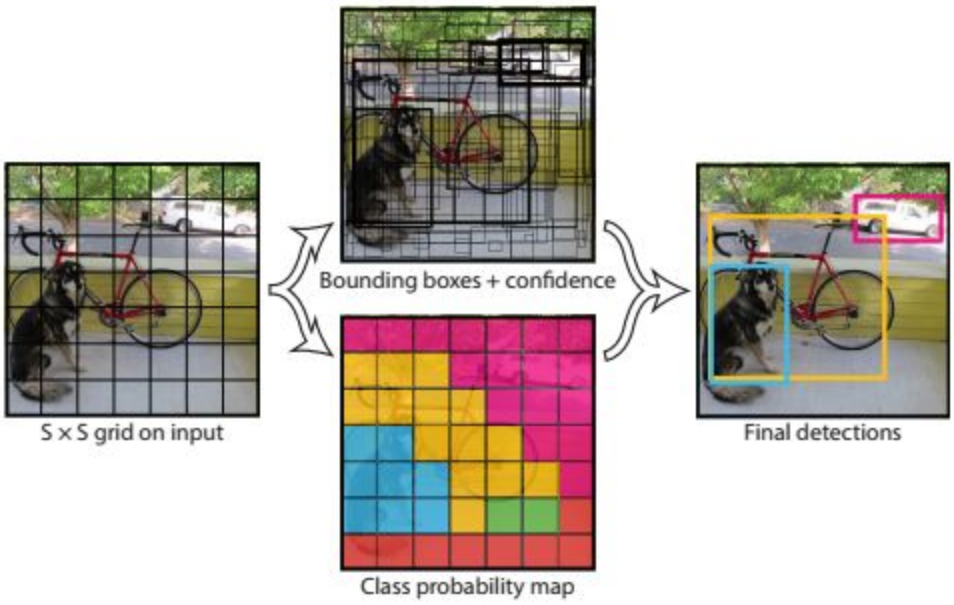


5. Each bounding box consists of 5 predictions: x, y, w, h, and confidence. The (x, y) coordinates represent the center of the box relative to the bounds of the grid cell. The width and height are predicted relative to the whole image. Finally the confidence prediction represents the IOU between the predicted box and any ground truth box.

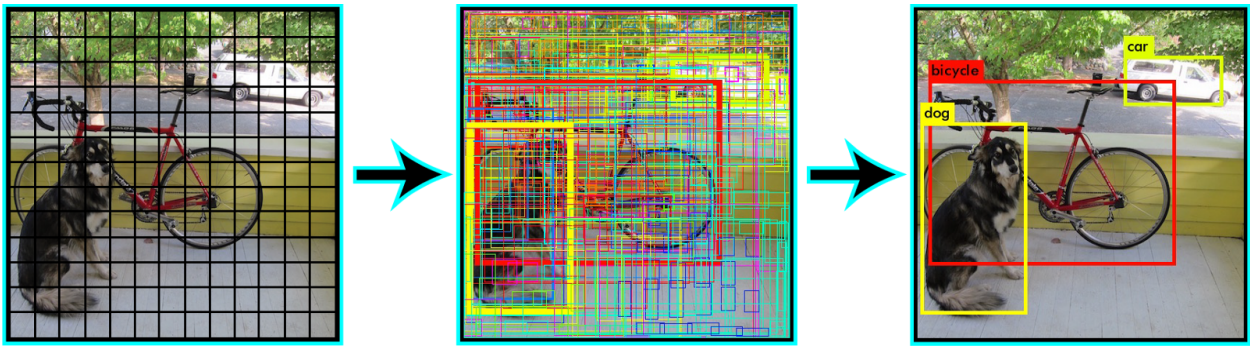
6. Each grid cell also predicts C conditional class probabilities, $Pr(Class_i | Object)$. These probabilities are conditioned on the grid cell containing an object. We only predict one set of class probabilities per grid cell, regardless of the number of boxes B.

At test time we multiply the conditional class probabilities and the individual box confidence predictions, $Pr(Class_i | Object)*Pr(Object)*IOU^{truth}_{pred} = Pr(Class_i)*IOU^{truth}_{pred}$

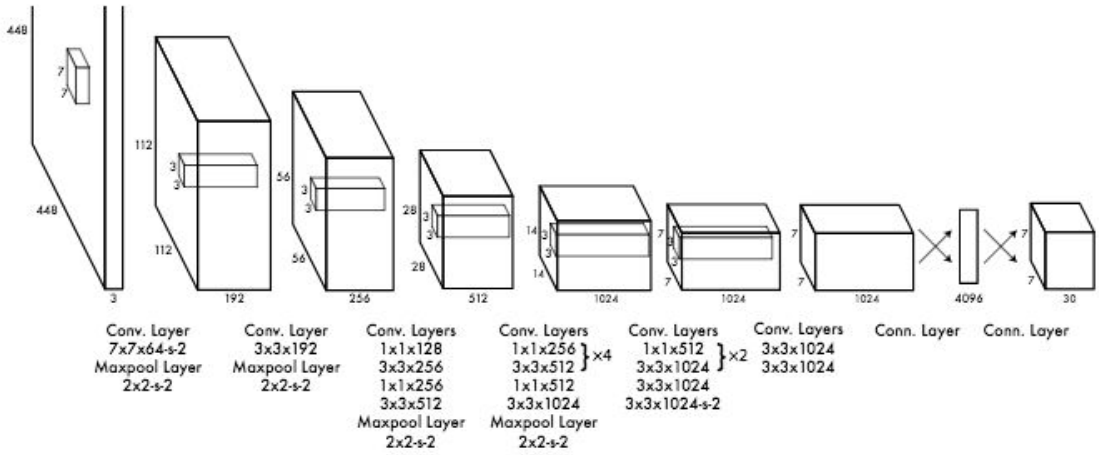
Which gives us class-specific confidence scores for each box. These scores encode both the probability of that class appearing in the box and how well the predicted box fits the object.



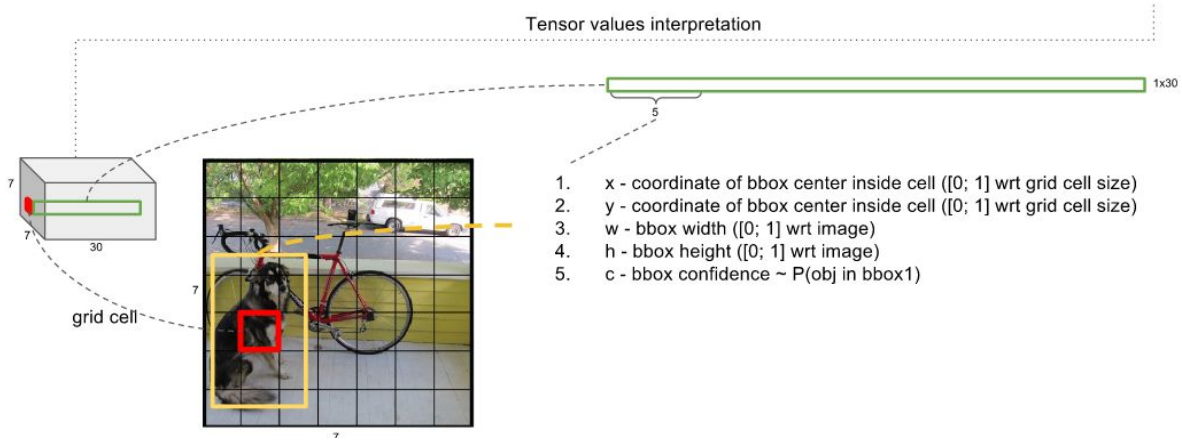
The predictions are encoded as an S X S X (BX5 + C) tensor, where B is the number of bounding boxes for each grid and C is the class probabilities



Network Design



The initial convolutional layers of the network extract features from the image while the fully connected layers predict the output probabilities and coordinates. Network architecture is inspired by the GoogLeNet model for image classification. Network has 24 convolutional layers followed by 2 fully connected layers. Instead of the inception modules used by GoogLeNet, the authors simply used 1 × 1 reduction layers followed by 3 × 3 convolutional layer. The final output of our network is the 7 × 7 × 30 tensor of predictions.



Training

1. Pretrain convolutional networks on ImageNet dataset. For pretraining we use the first 20 convolutional layers followed by a average-pooling layer and a fully connected layer.
2. Add four convolutional layers and two fully connected layers with randomly initialized weights. Detection often requires fine-grained visual information so increase the input resolution of the network from 224 × 224 to 448 × 448.
3. The final layer predicts both class probabilities and bounding box coordinates. Normalize the bounding box width and height by the image width and height so that they fall between 0 and 1. Parametrize the bounding box x and y coordinates to be offsets of a particular grid cell location so they are also bounded between 0 and 1.
4. Use a linear activation function for the final layer and all other layers use the following leaky rectified linear activation.
5. Optimize for sum-squared error in the output of the model. The authors used sum-squared error because it is easy to optimize, however it does not perfectly align with our goal of maximizing average precision.

Also, in every image many grid cells do not contain any object. This pushes the “confidence” scores of those cells towards zero, often overpowering the gradient from cells that do contain objects. This can lead to model instability, causing training to diverge early on.

To remedy this, the authors have devised to increase the loss from bounding box coordinate predictions and decrease the loss from confidence predictions for boxes that don’t contain objects. We use two parameters, λ_{coord} and λ_{noobj} to accomplish this. We set $\lambda_{coord} = 5$ and $\lambda_{noobj} = 0.5$

Sum-squared error also equally weights errors in large boxes and small boxes. The error metric should reflect that small deviations in large boxes matter less than in small boxes. To partially address this we predict the square root of the bounding box width and height instead of the width and height directly.

The loss function which is optimized during training is

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(x_i - \hat{x}_i\right)^2 + \left(y_i - \hat{y}_i\right)^2$$

$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2$$

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i\right)^2$$

$$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i\right)^2$$

$$+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} \left(p_i(c) - \hat{p}_i(c)\right)^2 \qquad (3)$$

Where 1^{obj}_{i} denotes if the object appears in cell *i* and 1^{obj}_{ij} denotes that the *j*th bounding box predictor in cell *i* is "responsible" for that prediction.

Note that the loss function only penalizes classification error if an object is present in that grid cell (hence the conditional class probability discussed earlier). It also only penalizes bounding box coordinate error if that predictor is "responsible" for the ground truth box (i.e. has the highest IOU of any predictor in that grid cell).

Limitations

YOLO imposes strong spatial constraints on bounding box predictions since each grid cell only predicts two boxes and can only have one class. This spatial constraint limits the number of nearby objects that our model can predict. The model struggles with small objects that appear in groups, such as flocks of birds.

Since our model learns to predict bounding boxes from data, it struggles to generalize to objects in new or unusual aspect ratios or configurations.

Finally, while the authors have trained on a loss function that approximates detection performance, our loss function treats errors the same in small bounding boxes versus large bounding boxes. A small error in a large box is generally benign but a small error in a small box has a much greater effect on IOU.

YOLO9000

In this paper, the authors propose a new method to harness the large amount of classification data we already have and use it to expand the scope of current detection systems. This method uses a hierarchical view of object classification that allows us to combine distinct datasets together.

The authors also proposed a joint training algorithm that allows us to train object detectors on both detection and classification data. This method leverages labeled detection images to learn to precisely localize objects while it uses classification images to increase its vocabulary and robustness.

Using this method the authors trained YOLO9000, a real-time object detector that can detect over 9000 different object categories. First they improved upon the base YOLO detection system to produce YOLOv2, a state-of-the-art, real-time detector. Then they used our dataset combination method and joint training algorithm to train a model on more than 9000 classes from ImageNet as well as detection data from COCO.

Better

Accuracy Improvements:

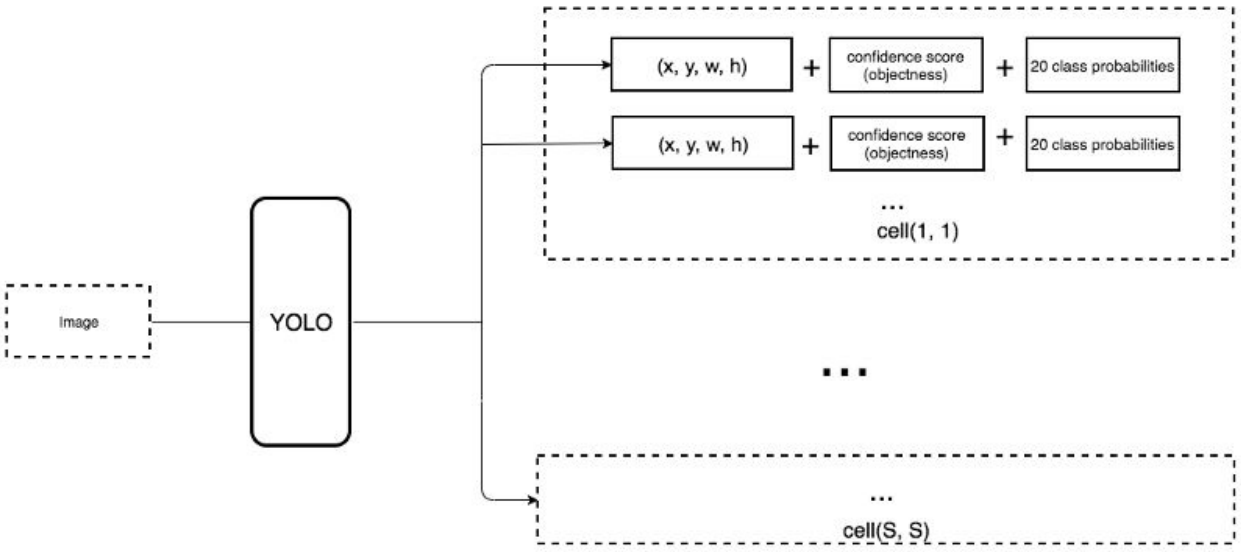
1. **Batch Normalization:** 2% improvement in mAP by using BN on all Convolutional Layers in YOLO.
2. **High Resolution Classifier:** The original YOLO trains the classifier network at 224 × 224 and increases the resolution to 448 for detection. This means the network has to simultaneously switch to learning object detection and adjust to the new input resolution. For YOLOv2 the authors first fine tune the classification network at the full 448 × 448 resolution for 10 epochs on ImageNet. This gives the network time to adjust its filters to work better on higher resolution input. The authors then fine

tune the resulting network on detection. This high resolution classification network gives us an increase of almost 4% mAP.

3. **Convolutional With Anchor Boxes**: YOLO predicts the coordinates of bounding boxes directly using fully connected layers on top of the convolutional feature extractor.

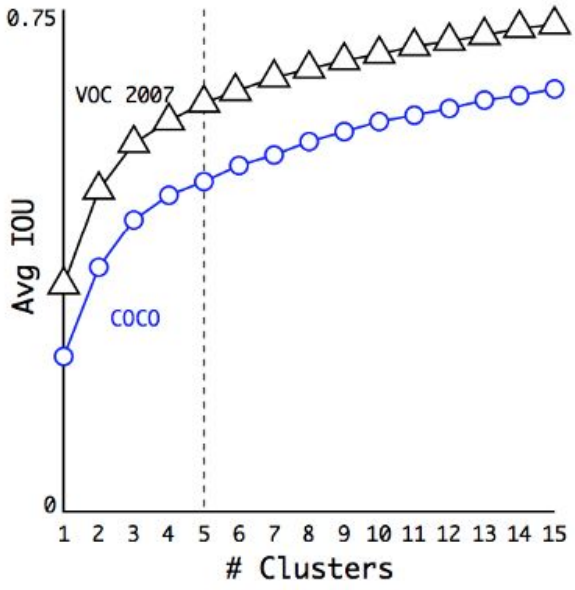
   The authors removed the fully connected layers from YOLO and used anchor boxes to predict bounding boxes. First they eliminated one pooling layer to make the output of the network's convolutional layers higher resolution. They also shrunk the network to operate on 416 input images instead of 448×448. They did this because they wanted an odd number of locations in our feature map so there is a single center cell. Objects, especially large objects, tend to occupy the center of the image so it's good to have a single location right at the center to predict these objects instead of four locations that are all nearby. *YOLO's convolutional layers downsample the image by a factor of 32 so by using an input image of 416 we get an output feature map of 13 × 13.*

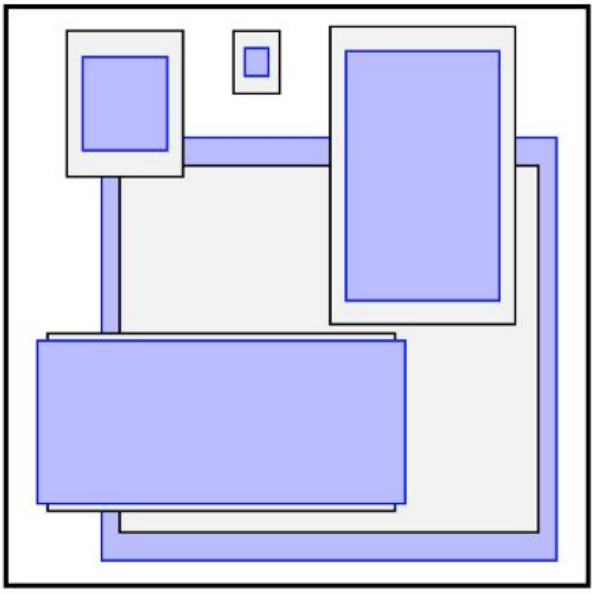
   When they moved to anchor boxes they also decoupled the class prediction mechanism from the spatial location and instead predict class and objectness for every anchor box. Image taken from source



   The objectness prediction still predicts the IOU of the ground truth and the proposed box and the class predictions predict the conditional probability of that class given that there is an object.

4. **Dimension CLusters**: Two issues were encountered with anchor boxes when using with YOLO. First, the box dimensions were hand-picked. The network can learn to adjust the boxes appropriately but if we pick better priors for the network to start with we can make it easier for the network to learn to predict good detections. Instead of choosing the authors ran k-means clustering on the training set bounding boxes to automatically find good priors. They chose k=5, as a good tradeoff between model complexity and high recall.





At only 5 priors the centroids perform similarly to 9 anchor boxes with an average IOU of 61.0 compared to 60.9. If we use 9 centroids we see a much higher average IOU of 67.2.

5. **Direct Location Prediction:** When using Anchor Boxes with YOLO, the second issue that was encountered was **model instability** during early iterations. Most of the instability comes from predicting the (x, y) locations for the box. In region proposal networks the network predicts values t_x and t_y and the (x, y) center coordinates are calculated as:

   $x = (t_x * w_a) - x_a$

   $y = (t_y * h_a) - y_a$

   This formulation is unconstrained so any anchor box can end up at any point in the image, regardless of what location predicted the box.

   Instead of predicting offsets the authors followed the approach of YOLO and

predicted location coordinates relative to the location of the grid cell. This bounds the ground truth to fall between 0 and 1. They used a logistic activation to constrain the network's predictions to fall in this range.

The network predicts 5 bounding boxes at each cell in the output feature map. The network predicts 5 coordinates for each bounding box, t_x, t_y, t_w, t_h, and t_o. If the cell is offset from the top left corner of the image by (c_x, c_y) and the bounding box prior has width and height p_w, p_h, then the predictions correspond to:

$b_x = \sigma(t_x) + c_x$

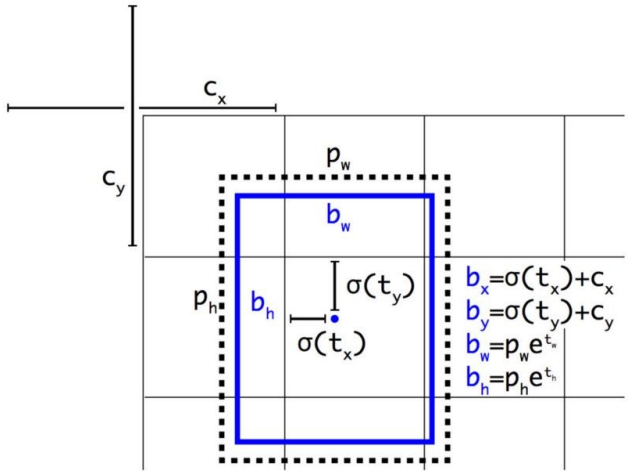
$b_y = \sigma(t_y) + c_y$

$b_w = p_w e^{tw}$

$b_h = p_h e^{th}$

$Pr(object) * IOU(b,object) = \sigma(t_o)$

Since we constrain the location prediction the parametrization is easier to learn, making the network more stable.

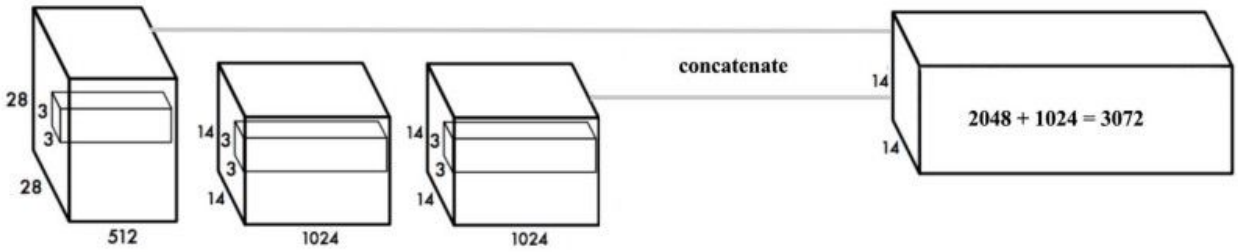


Bounding boxes with dimension priors and location prediction. We predict the width and height of the box as offsets from cluster centroids. We predict the center coordinates of the box relative to the location of filter application using a sigmoid function.

6. **Fine Grained Features:** This modified YOLO predicts detections on a 13 × 13 feature map. While this is sufficient for large objects, it may benefit from finer grained features for localizing smaller objects. Faster R-CNN and SSD both run their proposal networks at various feature maps in the network to get a range of resolutions. The authors took a different approach, simply adding a passthrough layer that brings features from an earlier layer at 26 × 26 resolution.

   The passthrough layer concatenates the higher resolution features with the low resolution features by stacking adjacent features into different channels instead of spatial locations, similar to the identity mappings in ResNet. This turns the 26 × 26 ×

512 feature map into a 13 × 13 × 2048 feature map, which can be concatenated with the original features. The detector runs on top of this expanded feature map so that it has access to fine grained features. Image_source



7. **Multiscale Training:** The original YOLO uses an input resolution of 448 × 448. With the addition of anchor boxes we changed the resolution to 416×416. However, since our model only uses convolutional and pooling layers it can be resized on the fly. Instead of fixing the input image size, change the network every few iterations. Every 10 batches the network randomly chooses a new image dimension size. Since the model downsampled by a factor of 32, pull from the following multiples of 32: {320, 352, ..., 608}. Thus the smallest option is 320 × 320 and the largest is 608 × 608. Resize the network to that dimension and continue training. This regime forces the network to learn to predict well across a variety of input dimensions. This means the same network can predict detections at different resolutions. The network runs faster at smaller sizes so YOLOv2 offers an easy tradeoff between speed and accuracy.

FASTER

VGG-16 is a powerful, accurate classification network but it is needlessly complex. The convolutional layers of VGG-16 require 30.69 billion floating point operations for a single pass over a single image at 224 × 224 resolution.

The YOLO framework uses a custom network based on the Googlenet architecture. This network is faster than VGG-16, only using 8.52 billion operations for a forward pass. However, it's accuracy is slightly worse than VGG16. For single-crop, top-5 accuracy at 224 × 224, YOLO's custom model gets 88.0% ImageNet compared to 90.0% for VGG-16.

Darknet-19 We propose a new classification model to be used as the base of YOLOv2. Our model builds off of prior work on network design as well as common knowledge in the

field. Similar to the VGG models we use mostly 3 × 3 filters and double the number of channels after every pooling step. The final model, called Darknet-19, has 19 convolutional layers and 5 maxpooling layers.

Training For Detection

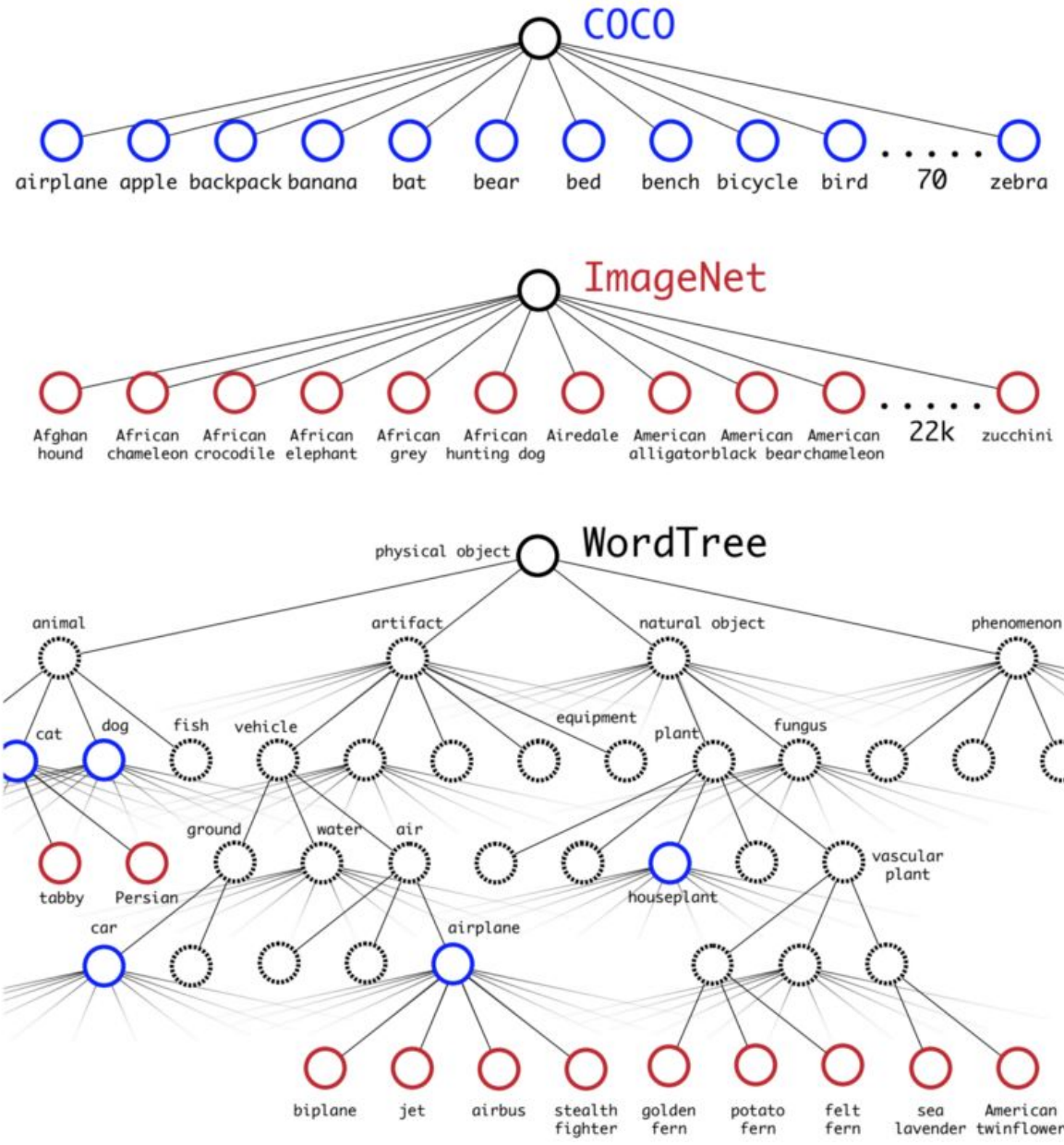
We modify this network for detection by removing the last convolutional layer and instead adding on three 3 × 3 convolutional layers with 1024 filters each followed by a final 1 × 1 convolutional layer with the number of outputs we need for detection. For VOC we predict 5 boxes with 5 coordinates each and 20 classes per box so 125 filters. We also add a passthrough layer from the final 3 × 3 × 512 layer to the second to last convolutional layer so that our model can use fine grain features.

STRONGER

We propose a mechanism for jointly training on classification and detection data. Our method uses images labelled for detection to learn detection-specific information like bounding box coordinate prediction and objectness as well as how to classify common objects. It uses images with only class labels to expand the number of categories it can detect. During training we mix images from both detection and classification datasets. When our network sees an image labelled for detection we can backpropagate based on the full YOLOv2 loss function. When it sees a classification image we only backpropagate loss from the classification specific parts of the architecture.

Hierarchical Classification

ImageNet labels are pulled from WordNet, a language database that structures concepts and how they relate. In WordNet, “Norfolk terrier” and “Yorkshire terrier” are both hyponyms of “terrier” which is a type of “hunting dog”, which is a type of “dog”, which is a “canine”, etc.
By choosing the shortest path from a node to its parent, a tree is built, called the **WordTree**, a hierarchical model of visual concepts. To perform classification with WordTree we predict conditional probabilities at every node for the probability of each hyponym of that synset given that synset.

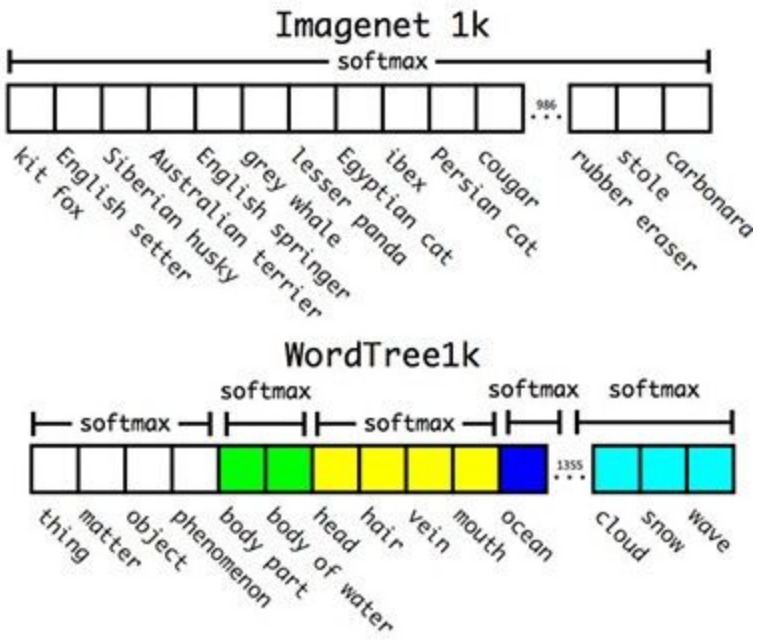


If we want to compute the absolute probability for a particular node we simply follow the path through the tree to the root node and multiply to conditional probabilities. So if we want to know if a picture is of a Norfolk terrier we compute:

Pr(Norfolk terrier) = Pr(Norfolk terrier|terrier) ∗Pr(terrier|hunting dog) ∗ . . .

∗Pr(mammal|Pr(animal) ∗Pr(animal|physical object)

For classification purposes we assume that the the image contains an object: P r(physical object) = 1.



To validate this approach the authors trained the Darknet-19 model on WordTree built using the 1000 class ImageNet. To build WordTree1k they added in all of the intermediate nodes which expands the label space from 1000 to 1369. During training they propagated ground truth labels up the tree so that if an image is labelled as a "Norfolk terrier" it also gets labelled as a "dog" and a "mammal", etc. To compute the conditional probabilities our model predicts a vector of 1369 values and we compute the softmax over all sysnsets that are hyponyms of the same concept.

This formulation also works for detection. Now, instead of assuming every image has an object, we use YOLOv2's objectness predictor to give us the value of P r(physical object). The detector predicts a bounding box and the tree of probabilities. We traverse the tree down, taking the highest confidence path at every split until we reach some threshold and we predict that object class.

Joint Classification and Detection

Now that we can combine datasets using WordTree we can train our joint model on classification and detection. We want to train an extremely large scale detector so we create our combined dataset using the COCO detection dataset and the top 9000 classes from the full ImageNet release

Using this dataset train YOLO9000 using the base YOLOv2 architecture but only 3 priors instead of 5 to limit the output size. When the network sees a detection image we backpropagate loss as normal. For classification loss, we only backpropagate loss at or above the corresponding level of the label. For example, if the label is "dog" we do not assign any error to predictions further down in the tree, "German Shepherd" versus

“Golden Retriever”, because we do not have that information. When it sees a classification image we only backpropagate classification loss. To do this we simply find the bounding box that predicts the highest probability for that class and we compute the loss on just its predicted tree. We also assume that the predicted box overlaps what would be the ground truth label by at least .3 IOU and we backpropagate objectness loss based on this assumption.

YOLOv3

1. Bounding Box Prediction:
   Same as YOLO9000.
   YOLOv3 predicts an objectness score for each bounding box using logistic regression. This should be 1 if the bounding box prior overlaps a ground truth object by more than any other bounding box prior. If the bounding box prior is not the best but does overlap a ground truth object by more than some threshold we ignore the prediction, following Faster R-CNN. The authors of YOLOv3 used the threshold of .5. Unlike Faster R-CNN this system only assigns one bounding box prior for each ground truth object. If a bounding box prior is not assigned to a ground truth object it incurs no loss for coordinate or class predictions, only objectness.
2. Class Prediction:
   Each box predicts the classes the bounding box may contain using multilabel classification. The authors did not use a softmax as they have found it is unnecessary for good performance, instead they simply used independent logistic classifiers. During training they used binary cross-entropy loss for the class predictions. This formulation helps when we move to more complex domains like the Open Images Dataset. Using a softmax imposes the assumption that each box has exactly one class which is often not the case. A multilabel approach better models the data.
3. Predictions Across Scales:
   YOLOv3 predicts boxes at 3 different scales. Our system extracts features from those scales using a similar concept to feature pyramid networks.
   From our base feature extractor we add several convolutional layers. The last of these predicts a 3-d tensor encoding bounding box, objectness, and class

predictions.

In the experiments with COCO, the authors predicted 3 boxes at each scale so the tensor is N × N × [3 ∗ (4 + 1 + 80)] for the 4 bounding box offsets, 1 objectness prediction, and 80 class predictions.

Next, take the feature map from 2 layers previous and upsample it by 2×. Also take a feature map from earlier in the network and merge it with our upsampled features using concatenation. This method allows us to get more meaningful semantic information from the upsampled features and finer-grained information from the earlier feature map. Then add a few more convolutional layers to process this combined feature map, and eventually predict a similar tensor, although now twice the size.

Perform the same design one more time to predict boxes for the final scale. Thus our predictions for the 3rd scale benefit from all the prior computation as well as fine-grained features from early on in the network.

The authors used k-means clustering to determine the bounding box priors. They just sort of chose 9 clusters and 3 scales arbitrarily and then divide up the clusters evenly across scales. On the COCO dataset the 9 clusters were: (10×13),(16×30),(33×23),(30×61),(62×45),(59× 119),(116 × 90),(156 × 198),(373 × 326).

4. Feature Extractor:

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

The new network is a hybrid approach between the network used in YOLOv2, Darknet-19, and residual network. The network uses successive 3 × 3 and 1 × 1 convolutional layers but now has some shortcut connections as well and is significantly larger. It has 53 convolutional layers, hence called Darknet-53!

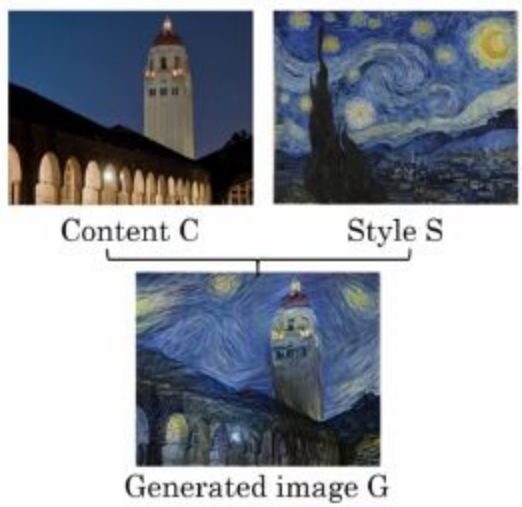
Training

Train on full images with no hard negative mining or any of that stuff. Use multi-scale training, lots of data augmentation, batch normalization, all the standard stuff.

Neural Style Transfer [Gatys et al., 2015. A neural algorithm of artistic style]

Now we know, that Convolutional Neural Networks learn a hierarchical representation of features. This property is the basis for NST.

Now, while doing Style Transfer, we are not training a neural network.



We have a Content image which we want to style.

We also have a Style Image, whose style we want to transfer to the Content image

Cost Function

The Cost function consists of two parts

$J(G) = \alpha J_{content}(C,G) + \beta J_{style}(S,G)$

We have to minimize this cost function using GD.

How to generate the Generated(Style Transferred) Image G:

1. Initiate G randomly
   Let the size be 100X100X3
2. Use GD to minimize J(G)
   $G = G - \alpha \frac{\partial J}{\partial G}$

Content Cost Function ($J_{content}$)

Take a hidden layer to compute the content cost. Say the ***l***th layer.

Take a pre-trained Network. Say VGG-19

Let $a^{[l](C)}$ and $a^{[l](G)}$ be the values of the activation of layer ***l*** after passing the images through the pretrained network(Forward propagation).

If the values of $a^{[l](C)}$ and $a^{[l](G)}$ are similar then, the images have similar content.

So, we can define $J_{content}$ as simply as

$J_{content}(C, G) = \frac{1}{2}||a^{[l](C)} - a^{[l](G)}||^2$, which is the element wise sum of squared differences between the activations of the ***l***th layer for the two images.

Style Cost Function (J_{style})

What does style means?

Style is defined is the correlation between activations across channels.

Style Matrix

Measures Correlation

Let $a^{[l]}_{i,j,k}$ = activation at (i,j,k).

Style Matrix G[l] is $n_c^{[l]}$ X $n_c^{[l]}$

$G^{[l]}_{k,k'}$ measure how correlated are the activations in Channel k with the Channel k'.

$$G^{[l]}_{k,k'} = \sum_{i}^{n^{[l]}_H} \sum_{j}^{n^{[l]}_W} a^{[l]}_{ijk} a^{[l]}_{ijk'}$$

This is actually unnormalized cross-covariance.

This is done for both Style and Generated Image.

For Style Image, we calculate $G^{[l](S)}$

For the Generated image, we calculate $G^{[l](G)}$

These matrices are called Gram Matrix in llnear Algebra

Now, coming back to Style Cost Function

$$J_{style}^{[l]}(S, G) = ||G^{[l](S)} - G^{[l](G)}||^2_F$$

However, the authors used a normalization constant, adding which the Style Cost Function becomes

$$J_{style}^{[l]}(S,G) = \frac{1}{(2n_H^{[l]}n_W^{[l]}n_C^{[l]})^2}\sum_k\sum_{k'}(G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'})^2$$

You get more visually pleasing results if style cost function from multiple results are used.

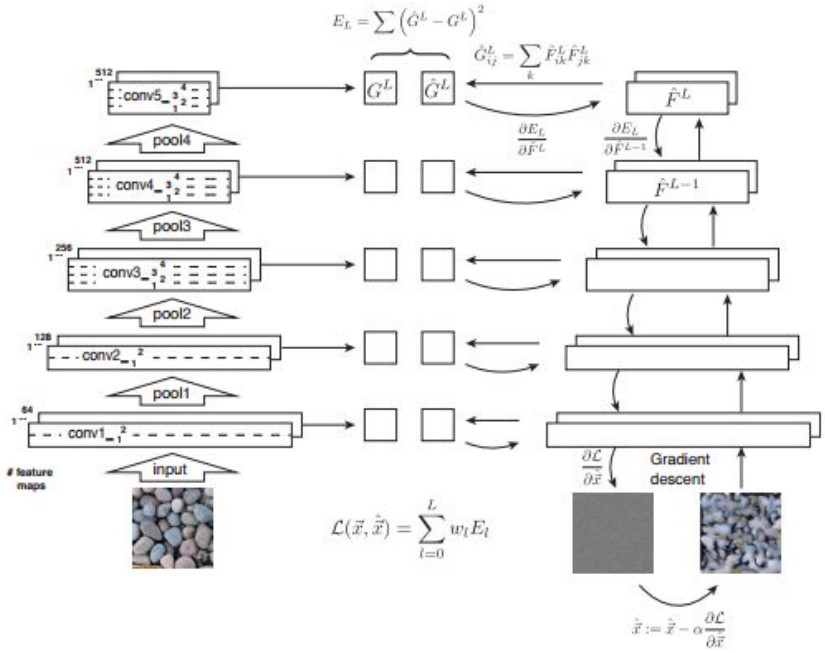
$J_{style}(S,G) = \sum_l \lambda^{[l]} J_{style}^{[l]}(S,G)$, where $\lambda^{[l]}$ is also a hyper-parameter

This allows to use both high level and low level features.

Thus,

$$J(G) = \alpha J_{content}(C,G) + \beta J_{style}(S,G)$$

Optimize this Cost Function using GD.



Source

FACE DETECTION and RECOGNITION

Using ONE-SHOT Learning

ONE SHOT LEARNING

In deep learning we normally need large amount of data to get good acceptable results.

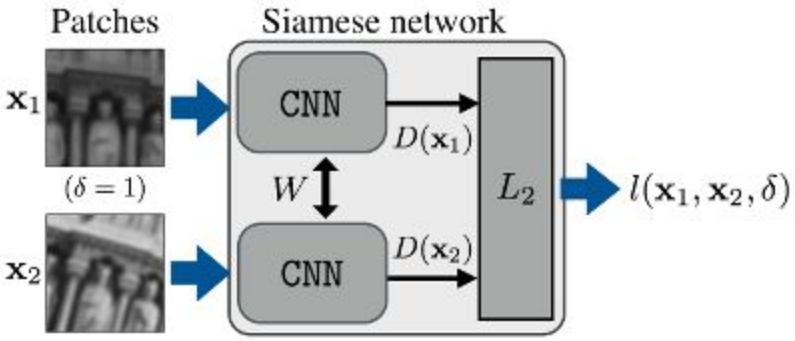
However, since good data is hard to find, it will be very convenient if we can train the neural network to learn only from a few data.

SIAMESE NETWORK

Siamese network is a type of neural network, which has two sister networks. Sister networks means they have the same architectural configurations, same weights and parameters. Parameter update is same for both networks.

Siamese networks are also used for face-recognition, where generally we have less number of samples of faces of employees.
Siamese architectures are good in these tasks because

1. Sharing weights across the two sister networks means **fewer parameters**.
2. Each subnetwork essentially produces a representation of its input. If your inputs are of the same kind, then the distance between the features can be thresholded to produce predictions.



Distance measure
$d(x_1,x_2) = ||D(x_1)-D(x_2)||^2_2$

If $d(x_1,x_2)$ is small, means the two inputs are similar. And vice-versa.

Triplet Loss

Here we have an Anchor image, a Negative Image and a Positive image

source

Now, we want the encodings of anchor and positive to be similar and the encodings of anchor and negative images to be dissimilar.

That is, $||D(A)-D(P)||^2 < ||D(A)-D(N)||^2$

To prevent trivial solution of all zero, we use a variable α (~0.2) as the margin between the two distances.

Loss Function

Given 3 Images A,P,N, the Loss Function is

$$L(A,P,N) = max(\ ||D(A)-D(P)||^2 - ||D(A)-D(N)||^2 + \alpha, 0)$$

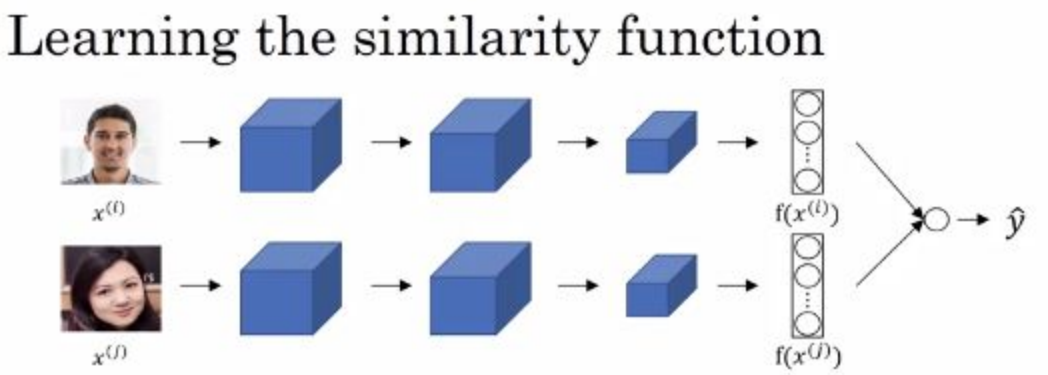
Cost Function

$$J = \sum_{i=1}^{M} L(A^{(i)}, P^{(i)}, N^{(i)})$$

For training, you need to make sure there are multiple images of the same class/person

Choose, those triplets which are hard to train on, that is the distance values are close.

FACE VERIFICATION



source

$$\hat{y} = \sigma(\sum_{k=1}^{128} w_i |f(x^{(i)})_k - f(x^{(j)})_k| + b)$$

Back-propagate the error as in a neural network

For inference, pass the new image through a branch and the distribution images through the other branch. Also, the features of the stored images can be pre-computed. By comparing the distance, we can infer whether the person is an employee or not, or verify the identity of the person.