Convolutional Neural Network

Introduction

Computer vision (CV) is making rapid advances due to Deep learning. Popular use include:

- Face recognition
- Self driving cars
- Object detection
- Generative art

Motivation

Rapid advances in computer vision are making new tools. In turn, other domains such as NLP are also influencing computer vision with their progress. The challenges in computer vision stems from the fact that input images could lead to 'heavy' networks. For example, if we consider $64 \times 64 \times 3$ RGB image we have 12288 features, which is manageable but a tiny amount of information. If we take a $1000 \times 1000 \times 3$ photograph i.e. 1 megapixel, we will have to deal with 3×10^6 features. A fully connected hidden layer will have dimensions of $N \times 3 \times 10^6$. If N=1000, we have 3 Billion parameters only in the first layer, not counting any other layer. This could become very computationally intensive. The solution lies in extracting information through 'convolutions'.

Convolutions Basics

Deep neural networks have compositional hierarchy. They detect basic shapes, like edges and corners in their early layers, which are generated by convolutions. Let us understand how that works.
Say, we have a matrix

$$\begin{bmatrix} 3 & 0 & 1 & 2 & 7 & 4 \\ 1 & 5 & 8 & 9 & 3 & 1 \\ 2 & 7 & 2 & 5 & 1 & 3 \\ 0 & 1 & 3 & 1 & 7 & 8 \\ 4 & 2 & 1 & 6 & 2 & 8 \\ 2 & 4 & 5 & 2 & 3 & 9 \end{bmatrix}_{6\times 6}$$

For a kernel i.e. a filter such as the one shown below:

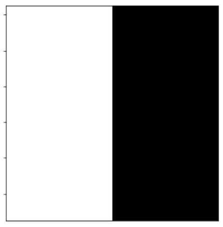
$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}_{3\times 3}$$

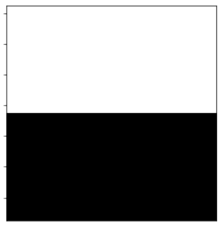
The output of "multiply-in-place & shift" would result in,

$$\begin{bmatrix} -5 & -4 & 0 & 8 \\ -10 & -2 & 2 & 3 \\ 0 & -2 & -4 & -7 \\ -3 & -2 & -3 & -16 \end{bmatrix}$$

"Multiply-in-place & shift" is the typical convolution operation. It takes a kernel and operates on the input at a position to produce a resultant output (which is recorded in a separate matrix). The kernels can be carefully selected to get desired effects such as detecting horizontal or vertical edges. Consider the `NumPy` array shown below with their image representations shown,

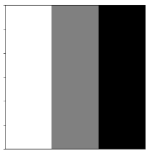
```
[[10 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0]      [[10 10 10 10 10 10 10 10 10 10 10 10 10 10]
 [10 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0]       [10 10 10 10 10 10 10 10 10 10 10 10 10 10]
 [10 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0]       [10 10 10 10 10 10 10 10 10 10 10 10 10 10]
 [10 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0]       [10 10 10 10 10 10 10 10 10 10 10 10 10 10]
 [10 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0]       [10 10 10 10 10 10 10 10 10 10 10 10 10 10]
 [10 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0]       [10 10 10 10 10 10 10 10 10 10 10 10 10 10]
 [10 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0]       [0  0  0  0  0  0  0  0  0  0  0  0  0  0 ]
 [10 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0]       [0  0  0  0  0  0  0  0  0  0  0  0  0  0 ]
 [10 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0]       [0  0  0  0  0  0  0  0  0  0  0  0  0  0 ]
 [10 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0]       [0  0  0  0  0  0  0  0  0  0  0  0  0  0 ]
 [10 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0]       [0  0  0  0  0  0  0  0  0  0  0  0  0  0 ]
 [10 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0]]      [0  0  0  0  0  0  0  0  0  0  0  0  0  0 ]]
```

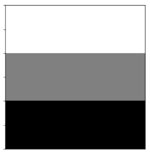




Let us consider two filters shown in the following `NumPy` pseudo-code - the first for vertical edges and the other for horizontal.

```
[[1  0  -1],         [[ 1   0,  1]
 [1  0  -1],          [ 0   0,  0]
 [1  0  -1]]          [-1  -1  -1]]
```

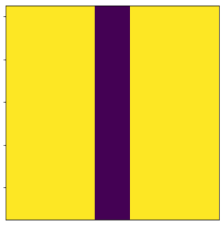


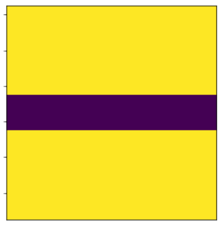


Applying these filter using the following operation from `SciPy`,

```
output = signal.convolve2d(img, filter)
```

we get the following outputs. (These images are somewhat directional. If we flip the filters, outputs would be produced 'inverted'.)





There are some variants to these edge detectors.

```
[[1  0 -1],           [[1  0  -1]           [[3  0  -3 ]
 [1  0 -1],            [2  0  -2]            [10 0  -10]
 [1  0 -1]]            [1  0  -1]]           [3  0  -3 ]]
```

Vertical Edge Det. Sobel Edge Det. Scharr Edge Det.

In ML, we can treat each number in this matrix as a parameter. This way, the backpropagation can tune the filter kernel. It has been one of the most astounding advances in CV field i.e. to be able to customize the convolution kernels.

Padding

When we perform convolution, the output gets smaller than input. The general rule is,

$$(N \times N) * (f \times f) \rightarrow (N - f + 1) \times (N - f + 1)$$

There are two downsides in the conventional definiton of convolution.

1. The edge pixels are used only once, as compared to the rest of the pixels, which get more participation via the sliding window. There is room for some improvement.
2. The image gets shrunk in the process

Before convolution, if we pad the image then it could mitigate both the deficiencies.

$$\begin{aligned}(\texttt{Padding}) &\rightarrow (N \times N) \rightarrow (N + 2p \times N + 2p)\\(\texttt{Convolution}) &\rightarrow (N + 2p \times N + 2p) * (f \times f) \rightarrow (N + 2p - f + 1) \times (N + 2p - f + 1)\end{aligned}$$

Valid & Same Padding

1. Valid $\rightarrow$ No padding. The output is expected to be smaller than input.

$$(N \times N) * (f \times f) \longrightarrow (N - f + 1) \times (N - f + 1)$$

2. Same $\rightarrow$ With padding. The output retains the same size as the input image.

$$(N \times N) \xrightarrow[\text{PAD}]{} [N + 2p \times N + 2p] \xrightarrow[\text{CONV}]{\text{f}\times\text{f}} (N \times N)$$

If padding is to result in `Same` convolution, the padding value has to be, $p = \left(\frac{f-1}{2}\right)$. The dimension/size of filter is chosen in odd values usually to facilitate easy padding, since even size could lead to asymmetric padding.

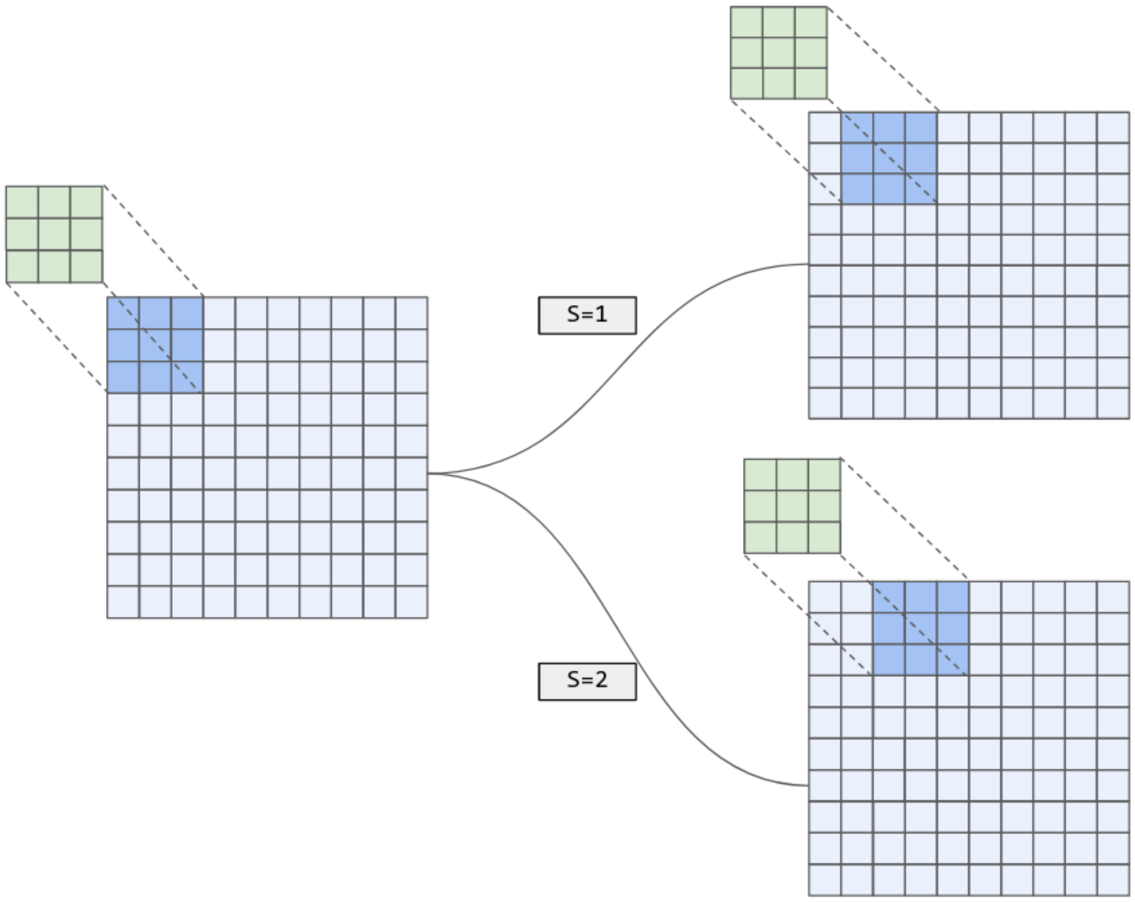
Strided Convolutions

Stride is the step length in the convolution operation. It determines how many positions the filter has to skip before the next computation. Without the information of stride, the stride length is assumed to be 1 and the filter shifts to the next laterally available position for operation.

Note:

In mathematics, the convolution in 2D is usually accompanied by horizontal and flipping of the kernel. But since the kernels are mostly symmetric, the effect is the same. Technically, mathematically the ML operation of convolution would correspond to cross-correlation.

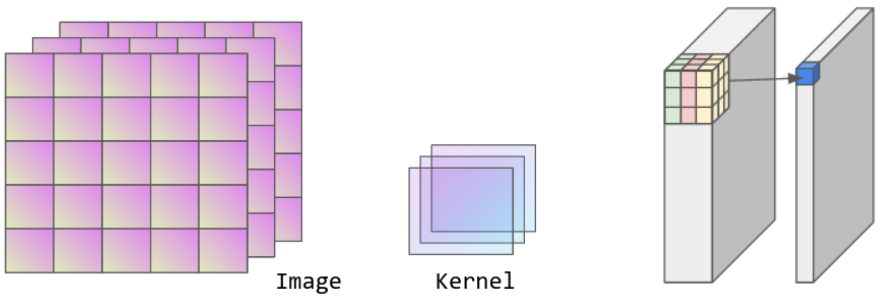
$$\begin{aligned}[N \times N] &\longrightarrow [(N + 2p) \times (N + 2p)]\\ [(N + 2p) \times (N + 2p)] &\xrightarrow[\text{f}\times\text{f}]{stride=s} \left[\left(\frac{N + 2p - f}{s} + 1\right), \left(\frac{N + 2p - f}{s} + 1\right)\right]\end{aligned}$$



Striding in convolution

Convolutions over volume

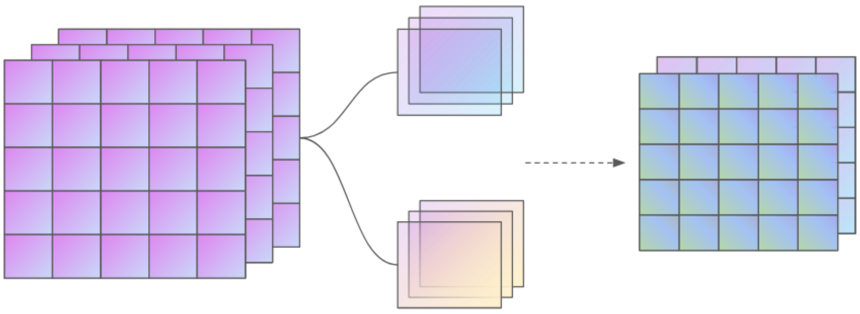
In CNNs, the filter has to have similar dimensions as the image it is being operated up. The result of the `multiply and replace` is a single pixel in the output. Hence, the output of a convolution from a 3D volume is a 2D matrix. The volume/depth variable is called `channel`. For example, in RGB image of $N \times N \times 3$, the $N \times N$ is the Height-Width, and the third attribute i.e. number of colors is the *channel*.



Multiple filters

If we want multiple filters at the same time, we can use them and stack the output as `channels` in the output. For N_c filters, in the convolution stage, we can expect:

$$[N \times N \times n_c] * N_c[f \times f \times f_c] \longrightarrow [(N-f+1) \times (N-f+1) \times N_c]$$


Multiple filters and corresponding output channels

One layer of CNN

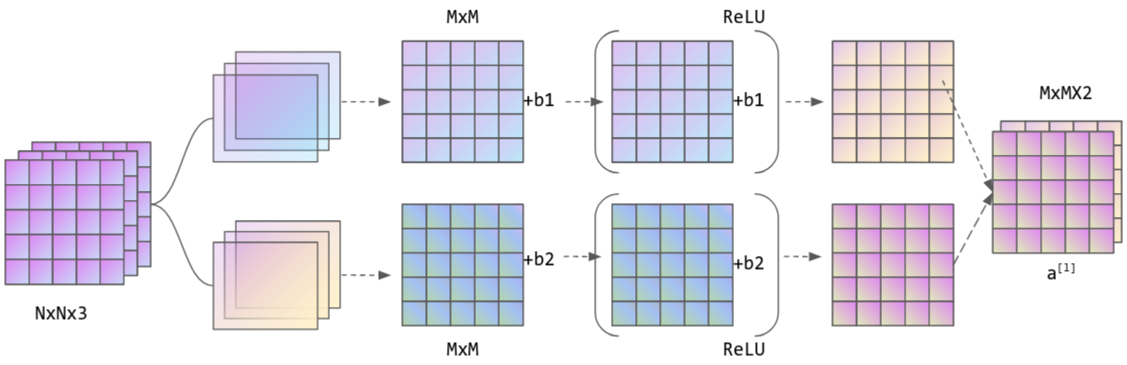
Filters take the role of weights in a Convolutional neural network.

Note

In a conventional network,

$$Z^{[l]} = W^{[l]} A^{[l-1]}$$
$$A^{[l]} = g(Z^{[l]})$$

In this case the input is the image. The weights convolution is similar to element wise operation followed by an addition of bias term. This output is passed through a non-linear activation.



Basics of CNN. The image is convolved by 2 filters. Their output is added with a bias term and passed through a non-linear function such as ReLU. The outputs are collated into a $M \times M \times 2$ output.

Convolution Summary

If layer '`l`' is a convolution layer, input is $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$, output is $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

- $f^{[l]}$ is the filter size
- $p^{[l]}$ is the padding
- $s^{[l]}$ is the stride
- $n_c^{[l]}$ is the number of filters

We could denote,

$$
\begin{aligned}
n_H^{[l]} &= \left\lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor \\
n_W^{[l]} &= \left\lfloor \frac{n_W^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor \\
n_c^{[l]} &= n_c^{[l]}
\end{aligned}
$$

- Each filter will be $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$
- Activations $a^{[l]} = m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$, where `m` is the batch size.
- Weights are $\underbrace{f^{[l]} \times f^{[l]} \times n_c^{[l-1]}}_{vol.of filter} \times \underbrace{n_c^{[l]}}_{\#filters}$

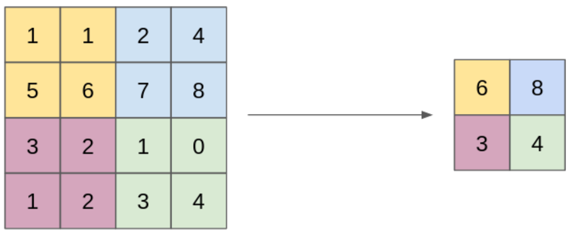
Types of Layers

Typically CNN networks involve,

1. Convolution (`CONV`)
2. Pooling (`POOL`)
3. Fully Connected (`FC`)

Pooling

Pooling reduces some of the computation going forward and makes the representation more robust. Role of noise is suppressed by the pooling operation after the convolution output. Pooling is commonly done via **Max Pooling** or **Average Pooling**. It does not learn from the backpropagation.



Max Pooling

Pooling Summary

For an image, input is $n_H \times n_W \times n_c$, output is $n'_H \times n'_W \times n_c$. The number of channels remain intact. Pooling is done layer by layer, unlike convolution which acts on the block. Padding is rare in `POOL`. There are no learnable parameters. Given,

1. Filter size = `f`
2. Stride = `s`
3. `Avg` or `Max` Pool

$$n_H \times n_W \times n_C \longrightarrow n'_H \times n'_W \times n_C$$
$$n'_H = \left[\frac{n_H - f}{s} + 1\right]$$
$$n'_W = \left[\frac{n_W - f}{s} + 1\right]$$
$$n_C \to n_C$$

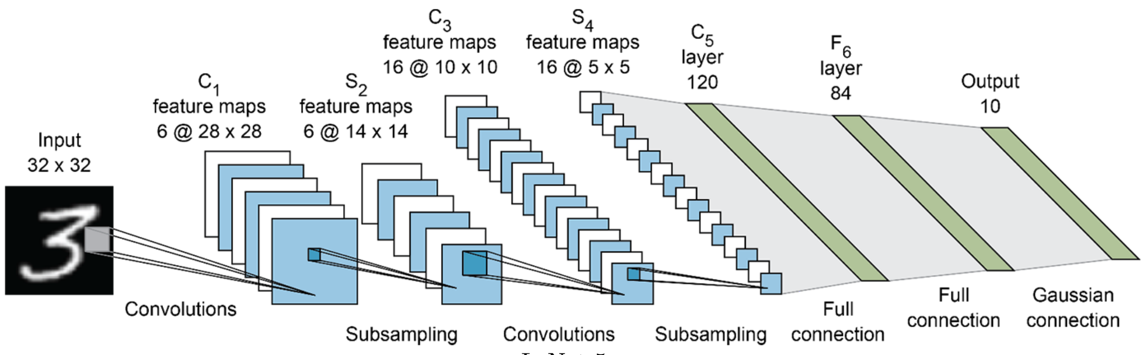
Need for CNN

- Parameter Sharing: A feature detector (such as a vertical edge detector) which is useful for one part of image can be useful in another part. One does not need full matrices. There is no requirement to have different detector as long as small kernels can compute by sliding windows.
- Sparsity of Connection: An output pixel is computed only by the window covering the pixels in concern. All other pixels do not have any contribution to output. Therefore, for each layer a value depends only on small number of connections from previous layer. This phenomenon has lot of significance when considering translation invariance.
- Reduced number of paramaters: For an MNIST image for example, the number of parameters are manageable at ˜250 as compared to about ˜15 million in FC networks.

Noteworthy architectures

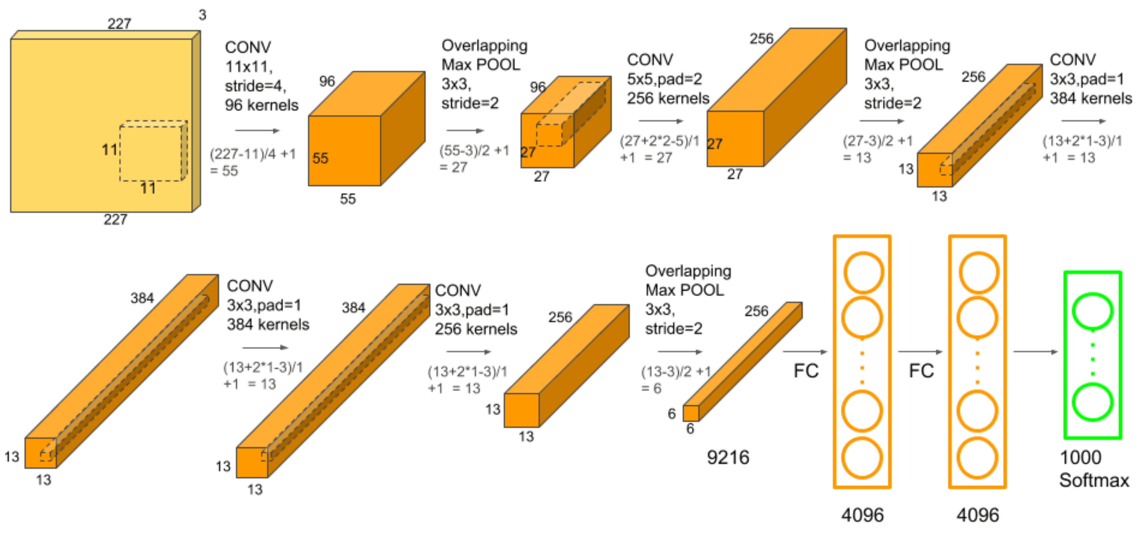
LeNet-5

Every time valid convolution was used. There were about 60,000 parameters. The height and width of image went down, while the number of channels went up for each successive layer. This was before ReLU was introduced and hence sigmoid or tanh activations were used. The non-linearity was applied after pooling (instead of after convolution).



LeNet-5

AlexNet



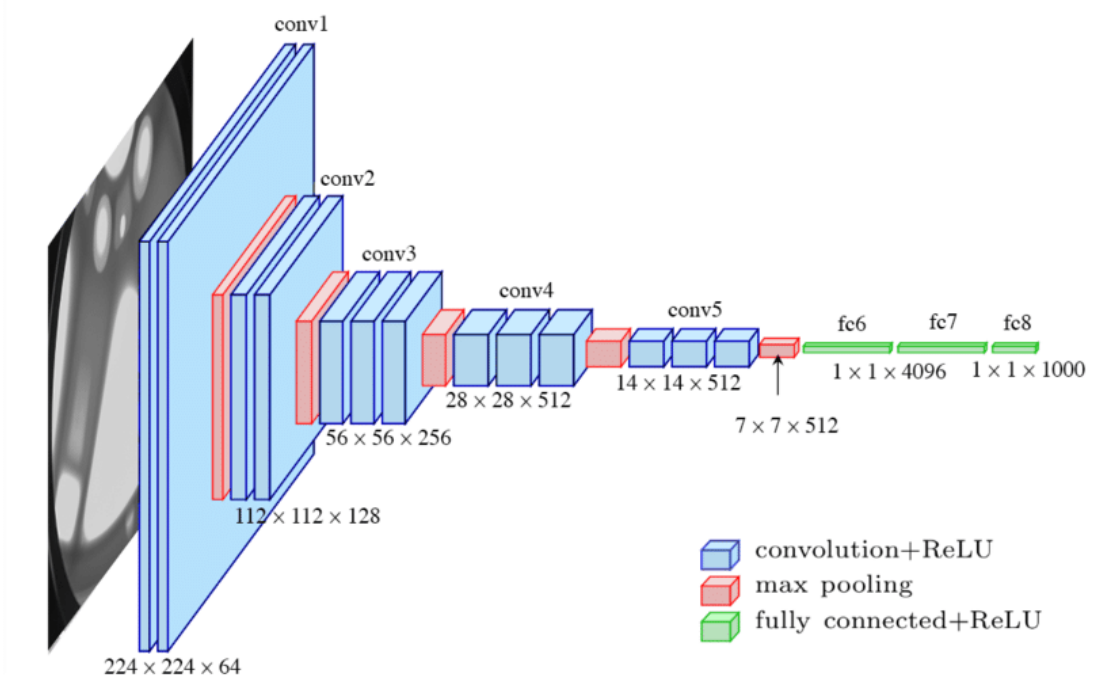
Alexnet

AlexNet had 60 million parameters. It employed ReLU activations. One new feature was `Local Response normalization`, which was normalizing a pixel across all the filters.

Gradient based learning applied to document recognition, Yann LeCun et al. IEEE Proceedings, 1998
ImageNet Classification with Deep Convolutional Neural Networks, Alex Krizhevsky, NIPS 2012

VGG-16

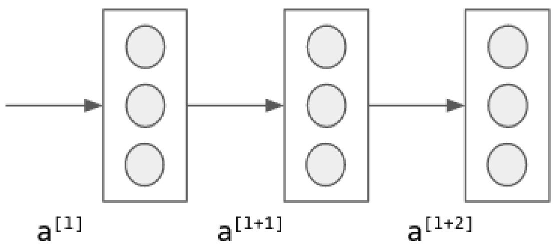
VGG-16 is a simple architecture. It has repetitive blocks of convolutions, followed by a pooling which shrinks the size of image by half for the next block of repetitive convolutions. The number of channels double in tandem. It has approximately 138 million parameters. This architecture is unique for its relative uniformity despite of large number of parameters encountered.



VGG 16

ResNet

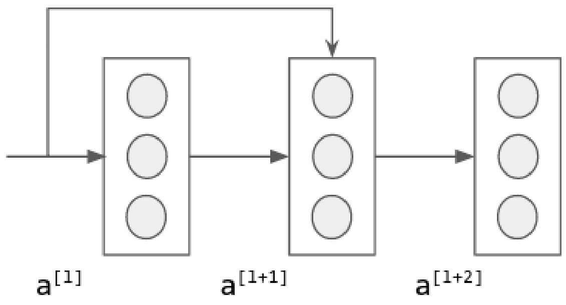
Very deep networks are hard to train in practice because of vanishing gradients. Conventionally, we have the activation fed into weights, followed by a non-linear activation to produce the next level of activations.



Normal block

In residual nets, we copy an activation much farther into the network as a short-circuit/skip connection. The skip activation gets added before the non-linear activation of a future activation (i.e. in the linear computation part).

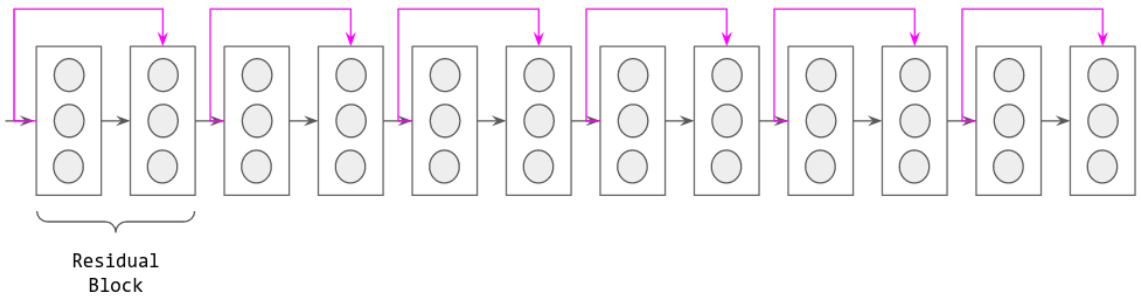
K. He et al., Deep Residual Learning for Image Recognition, CVPR 2016



Residual block

$$a^{[l+2]} = g\left(z^{[l+2]} + a^{[l]}\right)$$

Using residual blocks allows us to train very deep neural networks. In theory, if the depth of the network increases, error should be lesser. In practice, it does not happen as easily with conventional networks. ResNets perform better in this respect.



Deep network with Residual block

The reason that ResNet works so well is because the *identity function* is easy for a network to learn. Suppose we have one additional residual block after our main network backbone.

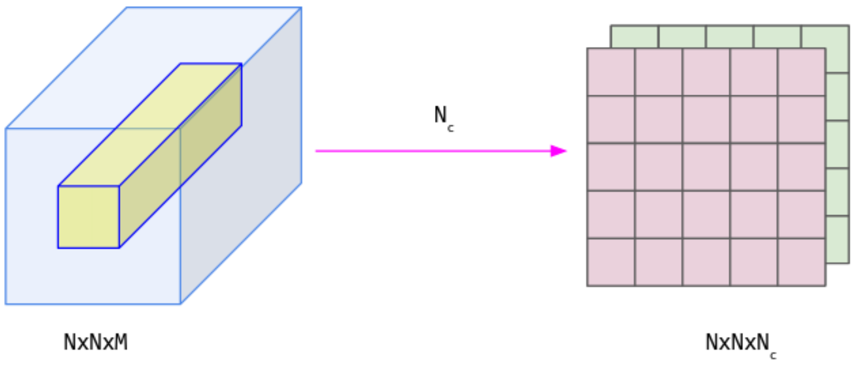
$$\begin{aligned} a^{[l+2]} &= g\left(z^{[l+2]} + a^{[l]}\right) \\ &= g\left(W^{[l+2]}a^{[l+1]} + b^{[l+2]} + a^{[l]}\right) \end{aligned}$$

If $W^{[l+2]} = b^{[l+2]} = 0$, then $a^{[l+2]} = a^{[l]}$, considering ReLU. Adding extra layers does not hurt the performance of the network. In very deep plain networks (without residual blocks), making the network deeper makes even the learning of an identity function quite difficult. ResNet employs same convolution. Hence dimension matching is less of a problem if we need to stack more residual blocks into the network. ResNet can be thought of as a special case of Highway Network which also employed a gated skip connection.

1×1 Convolutions

Proposed in a groundbreaking paper by Lin et al in 2013. 1×1 convolution may appear like multiplying with a number. But the significance is apparent when we consider different number of channels. In particular a 1×1 convolution will look at each of the $N \times N$ different grid positions and will take the element-wise product between M numbers on the matrix and the M numbers in the filter, followed by a ReLU non-linearity. It is similar to looking at each of the $N \times N$ positions, multiply through the volume at this position and produce a single output. In fact, one way to think

Network in Network, Lin et al. https://arxiv.org/abs/1312.4400 about the M numbers in this $1 \times 1 \times M$ filter is to consider a neuron i.e. taking as input M numbers, multiplying them by M weights and then applying non-linearity to it, leading to output like a `FC` network. Depending on how many filters we apply, we can get multiple `FC` networks as channels.

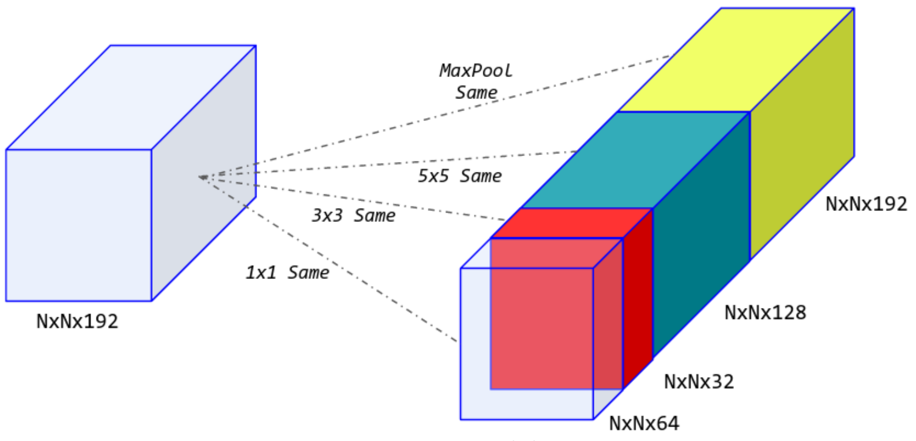


1×1 Convolutions with 2 filters

1×1 convolutions allow us to shrink the number of channels. If we are progressing through the network, we see the height and width dimensions go down but the number of channels go up. If the number of channels become too high to manage, then 1×1 convolutions can help us get more manageable numbers by specifying how many channels we need in the next stage ($\equiv$ number of channels in 1x1 filter).

Inception Network

Proposed by Szegedy et al in 2014. The Inception network or an inception layer proposed that instead of choosing the desired filter size in a `CONV` layer or whether we want a different layer, we can apply all of the choices. We just have to keep the Height/Width dimension same and the results can be stacked. (Any pooling will have to use *same padding*). We let the network understand the features, rather than designing the schema ourselves. `POOL` is sometimes used to match the height/width between the inception modules.

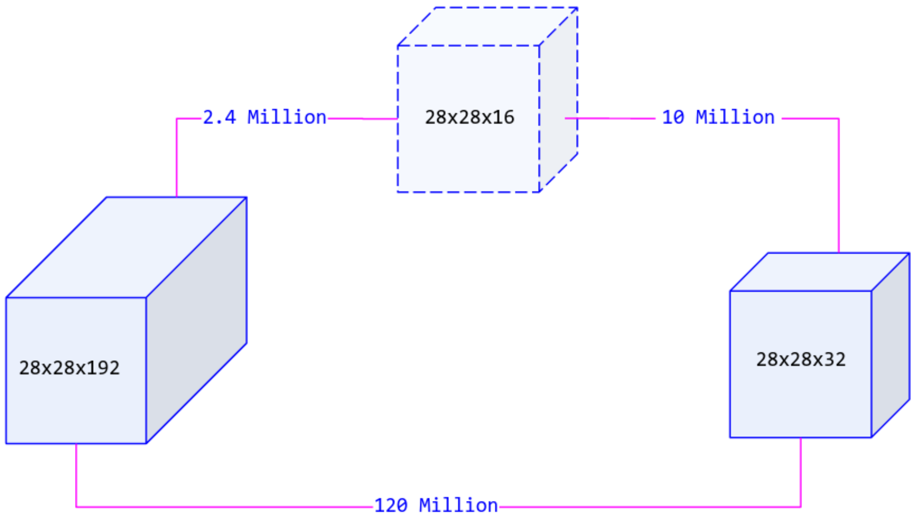


Inception module

Going deeper with convolutions, Szegedy et al., https://arxiv.org/abs/1409.4842

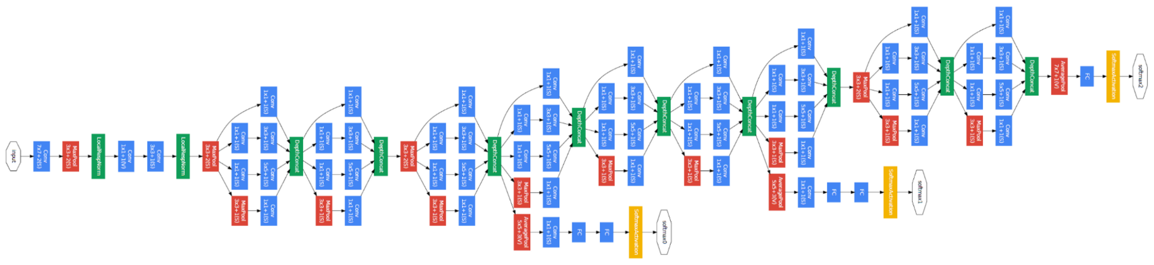
Computational cost

Unless some intervention is applied, Inception networks could become very costly in terms of computation. For example consider using a $28 \times 28 \times 192$ activation to be transformed to $28 \times 28 \times 32$ using 5×5 `SAME CONV`. The total number of computations involved is $\approx (28 \times 28 \times 32) \times (5 \times 5 \times 192) = 1.2 \times 10^8$. Doing the same via reducing to $28 \times 28 \times 16$ (via 1×1 `CONV`) and thereafter to $28 \times 28 \times 32$ would result in a tenth of the cost.



Bottleneck layer example

The overall inception network consists of a larger number of inception modules stacked together. Although the network seems complex, it is actually created of the same, though slightly modified blocks. There are side channels to tap into the network representations and evaluate performance (via softmax activation layers).



Inception Network or GoogleNet

Object Detection

The first step of object detection is `Object localisation` i.e. detect a bounding box around the object in question. This way, we can even address the issue of multiple objects being detected in an image. To localize, we need some extra outputs regarding the objects. We can specify b_x, b_y, b_h, b_w as parameters for the bounding box where (x, y) denotes the center of the box and (h, w) denote the height and width. By convention, $(0, 0)$ is the top-left corner, whereas $(1, 1)$ is the bottom right of the image.
We can use Bounding boxes to denote where the object has been detected. In such a case, our target label can be denoted as,

$$\mathbf{Y} = \begin{bmatrix} P_c \\ b_x \\ b_y \\ b_h \\ b_w \\ C_1 \\ \vdots \\ C_n \end{bmatrix} \begin{array}{l} \longrightarrow \text{Flag bit} \\ \\ \longrightarrow \text{Box Coordinates} \\ \\ \\ \\ \longrightarrow \text{One-hot enc. Classes} \\ \\ \end{array}$$

For 3 classes, the number of parameters all inclusive is 8. Training set in such cases will require bounding boxes for supervised learning.

If object **Present**

$$\mathbf{Y} = \begin{bmatrix} P_c \\ b_x \\ b_y \\ b_h \\ b_y \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{array}{l} \longrightarrow \text{Flag bit} = 1 \\ \\ \longrightarrow \text{Box Coordinates} \\ \\ \\ \\ \longrightarrow \text{Class-B} \\ \\ \end{array}$$

If object **Absent**

$$\mathbf{Y} = \begin{bmatrix} P_c \\ ? \\ \vdots \\ ? \\ ? \\ \vdots \\ ? \end{bmatrix} \begin{array}{l} \longrightarrow \text{Flag bit} = 0 \\ \\ \\ \\ \longrightarrow \text{Dont Care conditions} \\ \\ \\ \end{array}$$

Loss Function

$$L(\hat{y}, y) = \begin{cases} (\hat{y_1} - y_1)^2 + (\hat{y_1} - y_1)^2 \cdots (\hat{y_8} - y_8)^2 & P_c = 1 \\ (\hat{y_1} - y_1)^2 & P_c = 0, \text{(DC conditions, only flag bit remains)} \end{cases}$$

In practice, we can use log-likelihood for $C_1 \cdots C_n$, squared error for bounding box coordinates, and logistic regression for P_c flag bit. The example above is many a times good enough.

Landmark Detection

In more general cases, we want neural networks to recognize $(x.y)$ of some landmark positions. There are two ways to look at it:

- We output the center of a feature as a simple coordinate (x, y).
- We can output the (x, y) for all points encircling the feature.

By selecting a number of landmark points, we can mark out the feature which we want to identify for the neural network. For example if we are defining 64 landmark points for a particular feature,

$$y = \begin{bmatrix} P_c \\ l_{1,x} \\ l_{1,y} \\ \vdots \\ l_{64,x} \\ l_{64,y} \end{bmatrix}$$

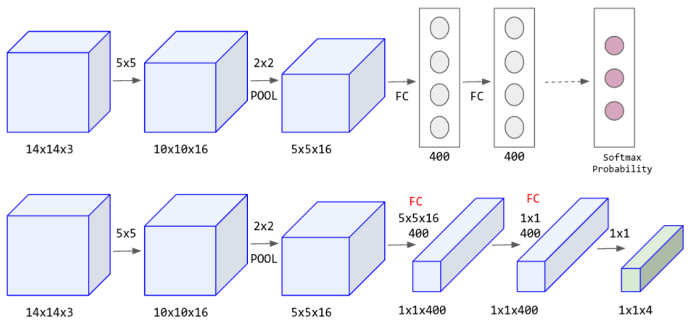
Detecting key landmarks is very necessary for computer vision to locate useful regions. Initially, annotators have to go through the `train` data and mark the regions laboriously. In pose detection, we also define landmarks such as extremities, torso, head etc. We can then predict the poses based on the manually annotated dataset. The identities of landmark should be consistent across the training samples. They should be consistent in terms of number of points, features delineated etc., across the data for meaningful output.

Naive Object detection

We can use a `Sliding window` detection algorithm in the most nascent way. We use closely cropped images (i.e. background minimized) to train the CNN. We use a sliding window to select a sub-region of the image, and compare its representation against the learned CNN. We use progressively larger windows to accommodate bigger objects and see if they exist in the image.
This however has a huge disadvantage in terms of computational cost. We are sliding many windows as tiles across each image, for the whole dataset. Unless we use a very small granularity (i.e. stride), we can never detect the exact bounding box. This is not only slow but inefficient. Before Deep learning, such techniques were in parlance, with hand engineered kernels to detect the features. Now this task can be improved massively by using CNN implementation of sliding windows.

Convolutional Object detection

Since sliding window approach is cumbersome, it is worthwhile to utilize the convolution. Before delving further, we observe that Fully-connected layers can be recreated by convolutional layers via proportionate number of filters and 1×1 convolution. Rather that consider the final layers as a set of nodes, i.e. 400×1, we are imagining the same in terms of $1 \times 1 \times 400$ volume.

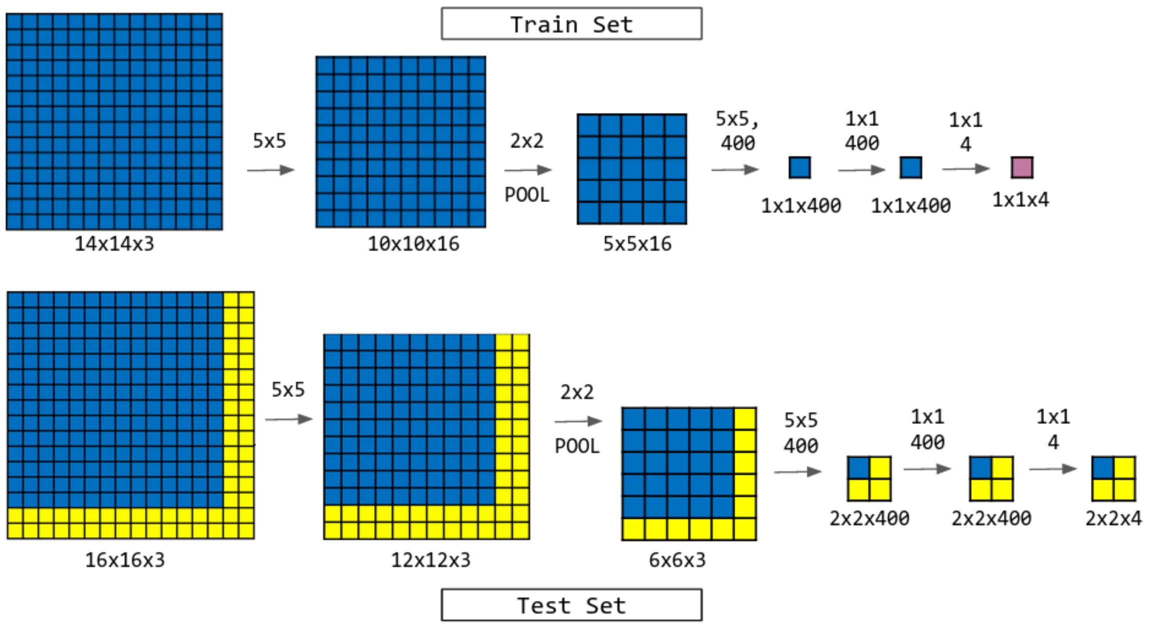


Convolutional equivalent of FC layer.

Suppose we train our object detector on a set of 14×14 images, with a final `softmax` layer having 4 outcomes. If our test images are 16×16, we can use the 14×14 window, and run it over the

OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks, Sermanet et al., https://arxiv.org/abs/1312.6229

16×16 image in few operations, using stride of 1. The information between the windows is highly duplicated. If we ran the first window, the output of the 14×14 will correspond to *Top-Left* pixel in the 2×2 output. Each time the stride moves the window, we can generate one more pixel of the output. The convolutional operation, instead of running subsets of matrix independently from start to finish, uses the common information to produce pixels of the output. A lot of information is shared between regions covered by the window. Instead of sequential input of patches, we can detect the object in one pass.



Convolutional sliding windows

Bounding Box Prediction

With sliding windows, we do not know the exact size of bounding box and the position of bounding box to cover the object. It becomes a grid search problem, which could turn out inefficient. `YOLO`: `You only look once`, attempts to solve this problem.
We divide the image into grid cells. We take the image classification and localization model and apply to each grid cell. For each grid cell, we specify a label y,

$$y = \begin{bmatrix} P_c \\ b_x \\ b_y \\ b_h \\ b_w \\ C_1 \\ \vdots \\ C_m \end{bmatrix}$$

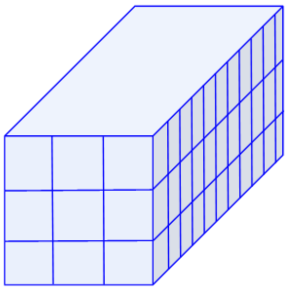
Whenever an object is detected, we take the midpoint and define the bounding box coordinates in whichever grid cell the midpoint is located. If the object is present in another box marginally, that

You Only Look Once: Unified, Real-Time Object Detection, Redmon et al., https://arxiv.org/abs/1506.02640

box is ignored.
Whenever an object is detected for a grid cell, we denote it as follows:

$$y = \begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

Therefore, we end up with a multidimensional matrix for each grid cell.



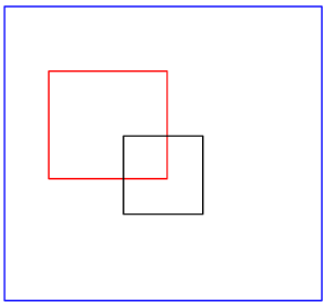
Each grid cell has multiple attributes for object detection

So during training, each image of a predefined size (e.g. $100 \times 100 \times 3$), is operated to produce a grid of $3 \times 3 \times 8$ (3-class detection). This information is used in backpropagation to map from any input image X to such output volume Y. [The image is the input. The bounding box attributes are the target labels]. The advantage of this algorithm is that bounding box is easy to obtain.
For a test image, we feed it to the model, and read off the output from the inference. We try to see which boxes have detected P_c flag set. As long as there is one object and only one box per grid cell, this method should work fine. In practice, YOLO uses bigger sized grid cells (˜19×19). With a pragmatic grid size, even if the object spans multiple boxes, it will be detected in one based on the classification algorithm and that cell becomes the detection zone for the object. The cell's center will be the midpoint of the image. Quality of detection is a function of pragmatic choices of grid size. If the size is too small, then there will be a lot of spurious results. If the size is too large, multiple objects may get contained in a single grid cell and only one of them will be detected. YOLO does image classification and then localization, based on training on gridded images. This works for any aspect ratio. It is a convolution based method and hence works very fast.
Note: In YOLO, the calculations are done from top-left, going towards bottom-right. $(0, 0)$ is Top-left and $(1, 1)$ is Bottom-right. b_x, b_y, b_h, b_w are all fractions relative to the bounding box. Naturally $b_x, b_y \in [0, 1]$ whereas b_h, b_w can be greater than 1 (i.e., the size could be greater than the detecting bounding box).

Intersection over Union (IOU)

To evaluate if object detection is working well, we have to consider the localization as well, which may leave some room for improvement. If object detection shows a bounding box different from the ground truth, we can evaluate the overlap via IOU. If IOU is greater than 0.5, it is usually a good detection (The overlap between box is more than 50% of the union size). If IOU = 1, the model is perfect in prediction. IOU is a measure of the overlap between the prediction and ground truth.

$$IOU = \frac{\text{Size of intersection}}{\text{Size of union of box}}$$



IOU Schema. The red box is the box predicted by the model. The black box is the ground truth.

Non-max suppression

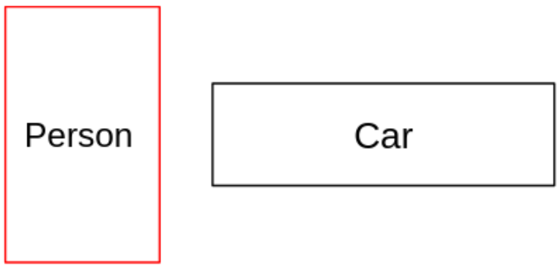
One of the problem in object detection is multiple boxes detecting the same object. Technically, only 1 grid cell should detect the object positively, and its center being the center of the object. In practice, several grid cells could return positive for classification of the same object, with a lot of overlapping bounding boxes seen. (because for many grid cells $P_c = 1$).
Let us consider P_c as a probability instead of a Boolean flag. Non-max suppression cleans up the detection process by evaluating the detection probability instead of a Boolean flag. It considers the box with the highest P_c value. Thereafter, it detects the bounding boxes with the highest IOU with the dominant bounding box, and eliminates them because these are the redundant choices.
In a simplified schema (i.e. say a single class detection), for a $N \times N$ grid:

- Find all the values of P_c for each bounding box
- Discard all P_c less than a threshold (~ 0.6)
- Pick box with highest P_c
- Discard boxes with IOU ≥ 0.5 with the selected box output i.e. ignore spurious detection.

If there are multiple object classes, it is advisable to do non-max suppression for bounding boxes relating to each of the classes.

Anchor Box

Each grid cell can so far detect only 1 object. Say if two objects are in the same grid cell (i.e. their centers are in the same grid cell), the technique so far can only detect on the classes and reject the other. Anchor boxes allow to detect more objects. With anchor boxes, we can pre-define shapes for different classes, and we can associate predictions to different anchor box shapes in the event the classes are in the same location in the image. To introduce anchor boxes, we just replicate the bounding box labels corresponding to the number of classes present. We can associate a set of attributes to a particular anchor box.



Anchor boxes for person and car

$$y = \begin{bmatrix} P_c \\ b_x \\ b_y \\ b_h \\ b_w \\ C_1 \\ C_2 \\ C_3 \\ P_c \\ b_x \\ b_y \\ b_h \\ b_w \\ C_1 \\ C_2 \\ C_3 \end{bmatrix}$$

Because shapes of classes are normally different, one bounding box can encode two different shapes/objects. The use of class labels $C_1 \ldots C_n$ remains unchanged to signify what the anchor box is reflecting. Previously, each object in training image was assigned to grid cell that contained the object's mid-point. The output vector was $3 \times 3 \times 8$. Now with anchor boxes, each object in training image is assigned to grid cell that contains object's mid-point and anchor box for grid cell with highest `IOU` with object shape. The detection is given to a pair i.e. [`cell`, `anchor-box`]. The output vector will be $3 \times 3 \times 2 \times 8$ (considering two anchor boxes per cell). This algorithms can have problems in two circumstances

- Two objects in the same grid cell, of the same type.
- Two different objects but similar anchor boxes.

YOLO Algorithm

Suppose we have 3 classes to detect, and we are using two anchor boxes, then for each image we can have a target label of size $3 \times 3 \times 2 \times 8$ i.e. each cell will have 16 values. To construct the training set, we go through each of the cells in images and populate the target vector Y. In practice, we use about $19 \times 19 \times 6 \times 8$. We train our convolutional networks with this target label. To make predictions, given the trained CNN it will output a $3 \times 3 \times 2 \times 8$ volume in inference. Wherever an object is detected, $P_c = 1$ and the bounding box attributes will be produced. Non-max suppression needs to be run to suppress spurious outputs. For each grid cell, produce at most 2 predicted boxed with different P_c. We can get rid of the low probabilities of bounding boxes. For each class, independently run non-max suppression to get more accurate object identification.

$$y = \begin{bmatrix} P_c \\ b_x \\ b_y \\ b_h \\ b_w \\ C_1 \\ C_2 \\ C_3 \\ P_c \\ b_x \\ b_y \\ b_h \\ b_w \\ C_1 \\ C_2 \\ C_3 \end{bmatrix}$$

Region Proposals

Region proposals are very influential body of work*. If we remember, the sliding window approach was greatly improved by convolutional method. In an image, a lot of grid cells have no values. There is no object present in them. R-CNN tries to pick regions which makes sense for object detection. It works by fast segmentation of the image and then find out which locales within the images have any value towards object identification.

$Image \longrightarrow$ Propose regions $\longrightarrow$ Classify region one-by-one $\longrightarrow$ Label + Bounding box
In a typical settings, 1000-2000 blobs are identified by segmentation and these are fed to CNN identifier. This process could get slow. There have been several modifications to the R-CNN approach mentioned in footnotes. FastR-CNN uses a convolutional approach instead of one-by-one identification. Faster R-CNN uses CNN to propose regions instead of conventional image processing pre-filtering.

R-CNN: Rich feature hierarchies for accurate object detection and semantic segmentation, Girshick et. al, 2013
Fast R-CNN, Ross Girshick, 2015
Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks, Ren et. al, 2016

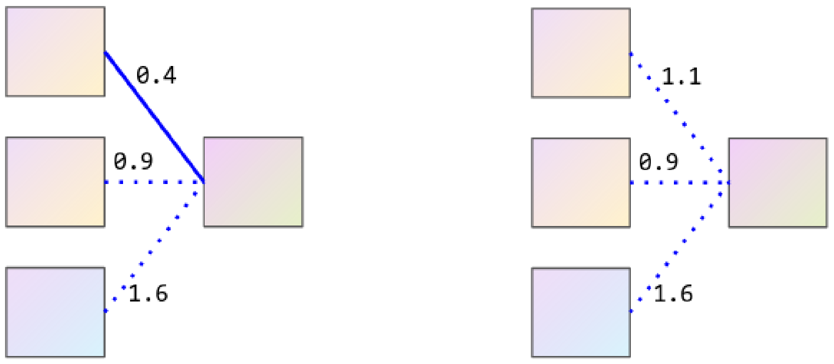
Face Recognition

Recognizing faces involves generally two kinds of tasks:

- Face verification, where input image(s) are matched to a person's features to see if it is the same person.
- Face Recognition, where input image(s) are matched to a database of known people, to see if it belongs to any of the registered entities.

One-shot learning

For most facial recognition tasks, we have to rely on one image of the person. Learning from a single sample is called 'One shot learning'. A trivial approach could be to assemble a few images from each user and create a multi-class classification system to match. However, this approach is wrought with issues if users are added or removed (which usually is the case with many organizations. Also it's effectiveness becomes questionable when the number of distinct classes (persons) becomes large and relatively one or few samples exist per person. Instead we try to learn a similarity function which evaluates a `degree of difference`. If the difference between two images is less than a threshold value (τ), then the samples are treated to have come from the same source. Else, we consider them from different sources i.e. different people.



$d \leq \tau$ for 1 label $d \geq \tau$ for all

If $d \leq \tau = 0.5$, we can reliably match sample to a known entity.

Siamese Networks

In a conventional CNN, the last layer value (usually depicted as a column vector) is the output to a translated image through the network. The N-dimensional vector is an `encoding` $f(x_i)$ of the input x_i. Such encoding can be used as good representations to compare two inputs.

$$d\left(x^{(1)}, x^{(2)}\right) = \left\| f\left(x^{(1)}\right) - f\left(x^{(2)}\right) \right\|_2^2$$

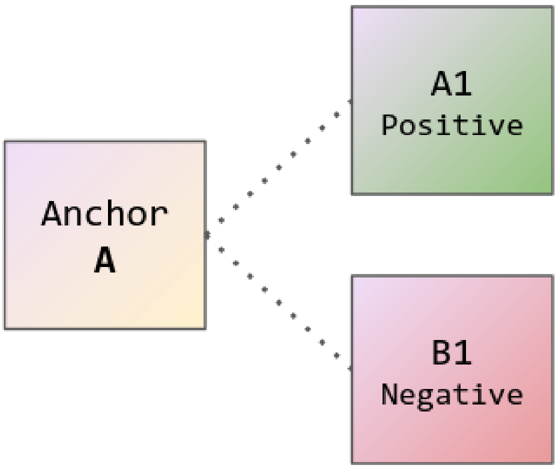
The equation represents the norm of the difference between two encoding of inputs. Parameters of neural network defines the encoding $f(x_i)$. Hence we have to train with output labels such that the network learns parameters to satisfy the following condition:

DeepFace: Closing the Gap to Human-Level Performance in Face Verification, Taigman et al, 2014

- If $x^{(i)}, x^{(j)}$ same, then the norm $\left\| f\left(x^{(i)}\right) - f\left(x^{(j)}\right)\right\|_2^2$ is small
- If $x^{(i)}, x^{(j)}$ different, then the norm $\left\| f\left(x^{(i)}\right) - f\left(x^{(j)}\right)\right\|_2^2$ is large

Triplet loss

One way to learn the parameters to satisfy the aforementioned condition is via `Triplet Loss`.



Triplet loss schema

For anchor image A, we want its encoding to be similar to positive sample A1, and very different from negative sample B1. We want $d(A, P) \leq d(A, N)$

$$\|f(A) - f(P)\|^2 \leq \|f(A) - f(N)\|^2$$
$$\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 \leq 0$$

In the case that encoding for cases are all zero, the solution is trivially true. In order that the neural network doesn't learn this case and is forced to find some other parameter weights, we include a parameter `Margin`.

$$\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 \leq -\alpha$$
$$\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha \leq 0$$

The margin forces the neural network to create a gap between positive and negative sample's norm with the anchor. Triplet Loss is defined as,

$$\mathbb{L}(A, P, N) = max(0, \|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha)$$

As long as this value is less than 0, the max (and the loss) is zero. If this is greater than zero, we have a positive loss. When we are trying to minimize the loss, the function tries to separate the value of $d(A, P)$ and $d(A, N)$ as best possible. It does not care how low the individual norms go down as long as the resultant value is less than zero.

The cost functions is an average of loss function

$$\mathbb{J} = \sum_{i=1}^{m} \left[A^{(i)}, P^{(i)}, N^{(i)} \right]$$

We need a lot of pairs of positive samples and negative examples, for a single label to train the triplet loss. Thereafter we can use it for one-shot learning. During training, if $\mathbb{A}, \mathbb{P}, \mathbb{N}$ are chosen carelessly, then $d(A, P) - d(A, N) + \alpha \leq 0$, is easily satisfied. Hence choose triplets which are hard to train such that $d(A, P) \approx d(A, N)$, so that learning algorithm learns to differentiate them better.

Face verification

To perform face verification, we compare the stored embedding with the on-the-fly embedding procured from the image by a logistic regression network.

- 1, if x_i, x_j same
- 0, if x_i, x_j different

The output $\hat{y}$ can be defined as,

$$\hat{y} = \sigma \left(\sum_{k=1}^{N} w_k \left| f(x_i)_k - f(x_j)_k \right| + b \right)$$

This corresponds to element-wise absolute value difference of embedding values. We train the logistic regression on some positive and negative samples to make good model generalization to new samples.

χ^2 Version

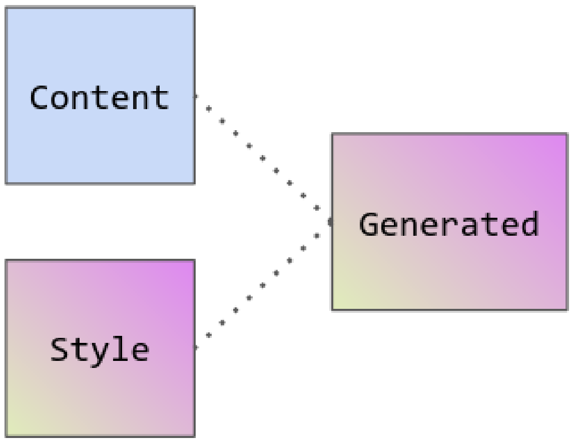
$$\hat{y} = \sigma \left(\sum_{k=1}^{N} w_k \frac{\left[f(x_i)_k - f(x_j)_k \right]^2}{f(x_i)_k + f(x_j)_k} + b \right)$$

We can pre-compute and save the embedding of registered faces in a database. During inference time, we only need to compute one embedding for verification.

FaceNet: A Unified Embedding for Face Recognition and Clustering, Schroff et al., 2015

Neural Style Transfer

In neural style transfer, the content image is transformed according to a style image image to create a generated image.



Neural Style transfer scheme

Overall style transfer cost function

In training a style transferring network, we can imagine the total cost as,

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

α, β are hyper-parameters which weight the style loss vs. the content loss.

- Initialize G randomly: G = $100 \times 100 \times 3$
- Use Gradient descent to minimize J(G): G = G - $\frac{\partial J(G)}{\partial G}$

Content cost function

Suppose we use a hidden layer, neither too shallow or deep, to compute content cost. Let $a^{[l](C)}, a^{[l](G)}$ be activations in the layer l. If $a^{[l](C)} \approx a^{[l](G)}$, then the images will look similar in the final layer.

$$J_{content}(G, C) = \left\| a^{[l](C)} - a^{[l](G)} \right\|^2$$

This is the squared L2 norm, i.e. element-wise sum of square of difference of activations between content and generated image. Gradient descent will give incentive for the images to be similar.

Style cost function

We define style in a photograph quantitatively as correlation between activations across channels. When we investigate the different channels pixel pixel-by-pixel (or unit-by-unit), the channels which are highly correlated will have similar amounts of activations in nodes/units of same layer. The degree of correlation measures if high-level textures occur/don't occur between different filters. If

A Neural Algorithm of Artistic Style, Gatys et al., 2015

one filter captures vertical texture and other captures a color tint, we can say the channels are correlated if, wherever the texture appears it is also accompanied by the color tint on another channel. Some high level features occur together always (e.g. a texture and a color), when two specific filters are well correlated. We can use this idea to create a Style matrix (also known as `Gram matrix` in linear algebra).
Let $a_{i,j,k}$ be activation at i, j, k (h, w, c) position for channel l. Let $G^{[l]}$ be $n_c^{[l]} \times n_c^{[l]}$, where n_c is number of channels. $G^{[l]}_{k,k'}$ denotes the comparison of activations between k, k' channels (channels in the range of 1 to n_c).

$$G^{[l]}_{k,k'} = \sum_{i=1}^{n_h} \sum_{i=1}^{n_w} a^{[l]}_{i,j,\mathbf{k}} \cdot a^{[l]}_{i,j,\mathbf{k}'}$$

All that it is doing is going position by position at the layer, and just multiplying the activations of the channels ℸ and ℸ′ for all pairs in the list of channels. If the channels have correlation, $G^{[l]}_{k,k'}$ will be a large value. It is non-normalized cross-covariance. We compute the style matrix for both style image and generated matrix.

$$\begin{aligned} J^{[l]}_{style} &= \left\| G^{[l](S)} - G^{[l](G)} \right\|^2_F \\ &= \frac{1}{2 \cdot n_h^{[l]} \cdot n_w^{[l]} \cdot n_c^{[l]}} \sum_k \sum_{k'} \left(G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'} \right) \end{aligned}$$

The denominator is a normalization constant abridged as `C` in several literature. We can get better results if we use combinations of layers to perform style transfer with arbitrary weighing factors (λ's)

$$J_{style}(S, G) = \sum_l \lambda^{[l]} \cdot J^{[l]}_{style}(S, G)$$