Session 5:
CNNs Overloaded

Varun Sundar, 1st October 2018

Outline

Review:

1. Building blocks of a CNN

Today's:

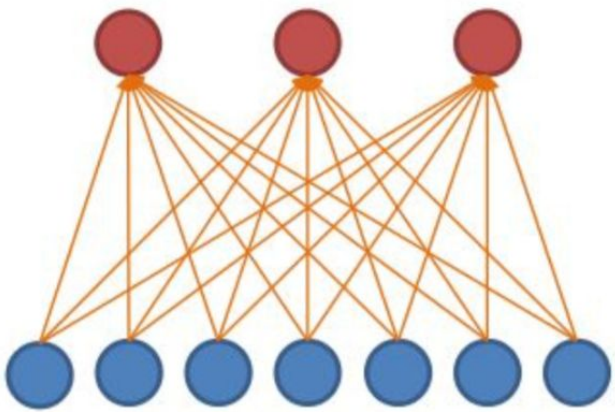
2. Backprop in CNNs
3. BatchNorm
4. CNN architectures
5. CNN in libraries

CNN Building Blocks

CNN vs MLP

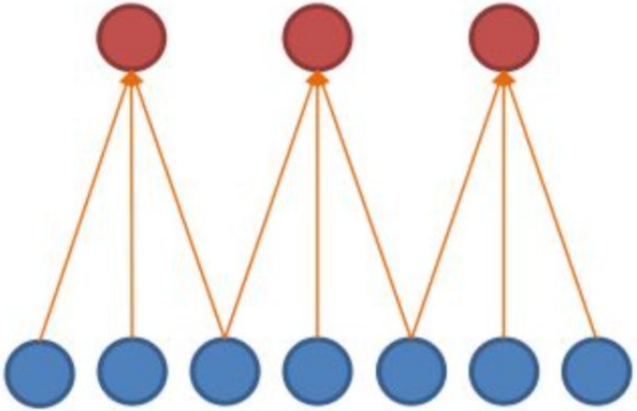
- CNNs are MLPs with two constraints:
  - Local Connectivity
  - Parameter Sharing

CNN: Local connectivity (LC)



Hidden layer (3 nodes)

Input layer (7 nodes)



MLNN (7 X 3 = 21 parameters)

MLNN-LC (3 X 3 = 9 parameters)

2.3X runtime and storage efficient.

In general for a level with *m* input and *n* output nodes and CNN-local connectivity of *k* nodes *(k<m)*:

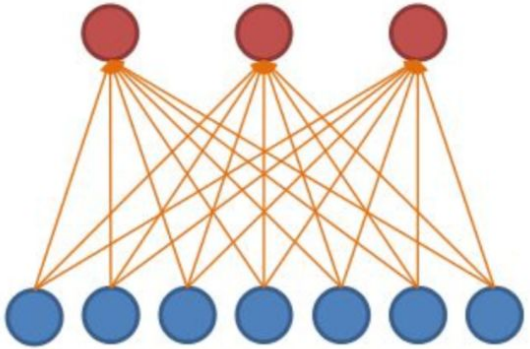
MLNN have

1. *m x n* parameters to store.
2. O(*m x n*) runtime

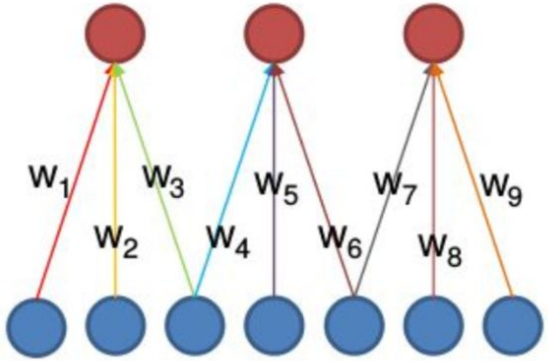
MLNN-LC have:

1. *k x n* parameters to store.
2. O(*k x n*) runtime

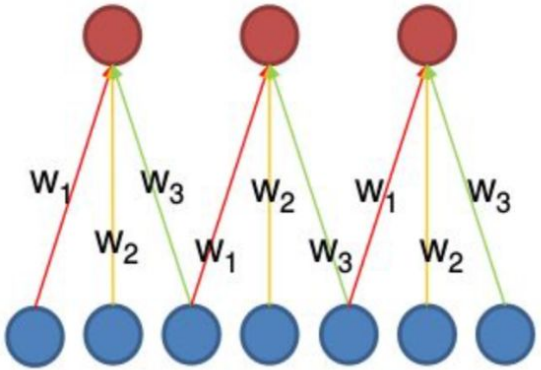
CNN: Parameter sharing (PS)



MLNN (21 parameters)



MLNN-LC (3 X 3 = 9 parameters)
2.3X runtime and storage efficient.



MLNN-LC-PS (3 parameters)
2.3X faster,
& 7X storage efficient.

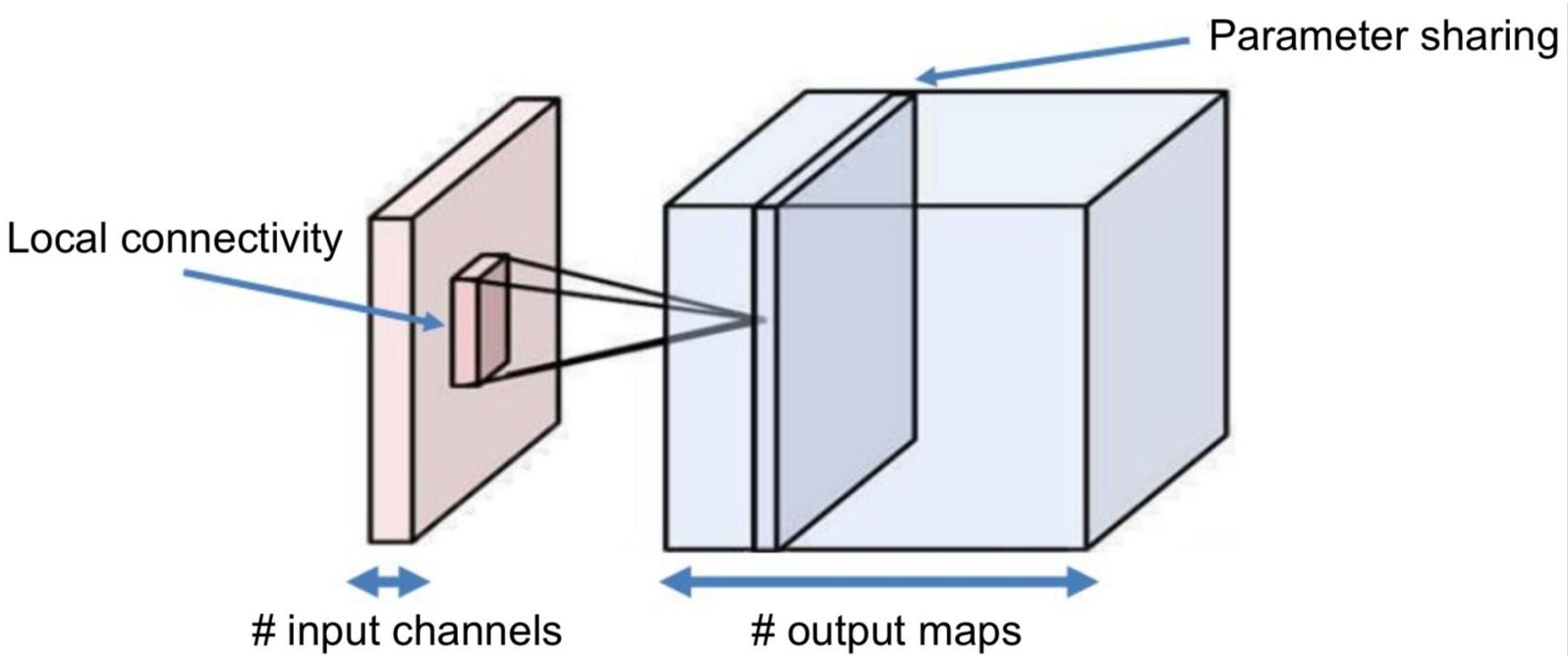
In general for a level with *m* input and *n* output nodes and CNN-local connectivity of *k* nodes *(k<m)*:

MLNN have
1. *m x n* parameters to store.
2. O(*m x n*) runtime

MLNN-LC have:
1. *k x n* parameters to store.
2. O(*k x n*) runtime

MLNN-LC-PS have:
1. *k* parameters to store.
2. O(*k x n*) runtime

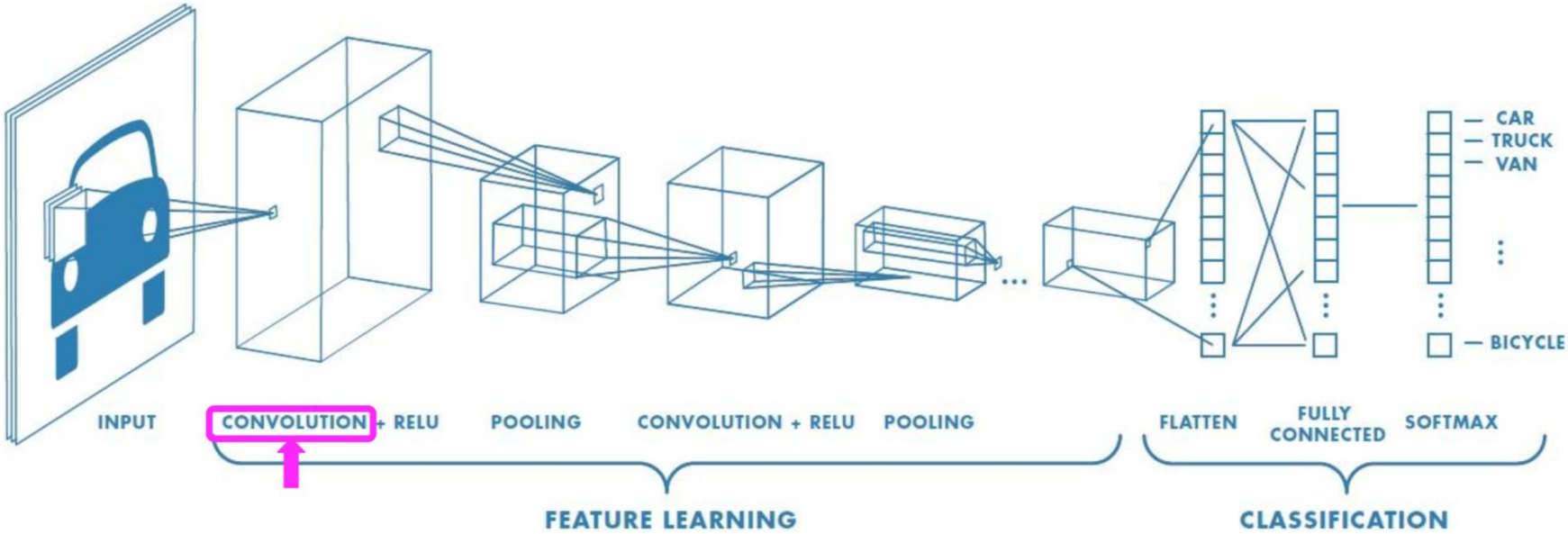
Generic Overview



CNN Blocks

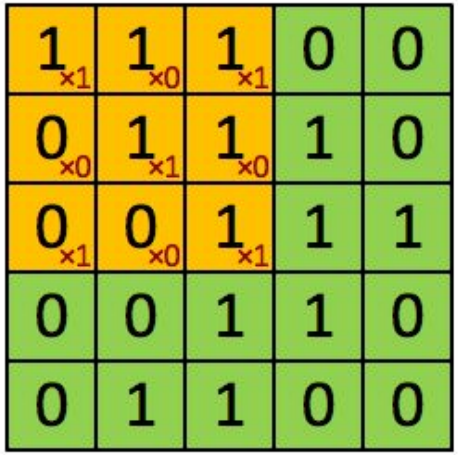
- Convolutional
- Activations
- Pooling
- Flattening
- Unpooling (recent)
- Deconvolution (more accurately transposed convolution)

CNN Blocks Overview

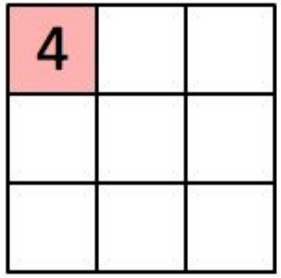


Convolution Layer

- Similar to signal convolution
- Inspiration from classical filtering, ISP.
- Actually uses correlation

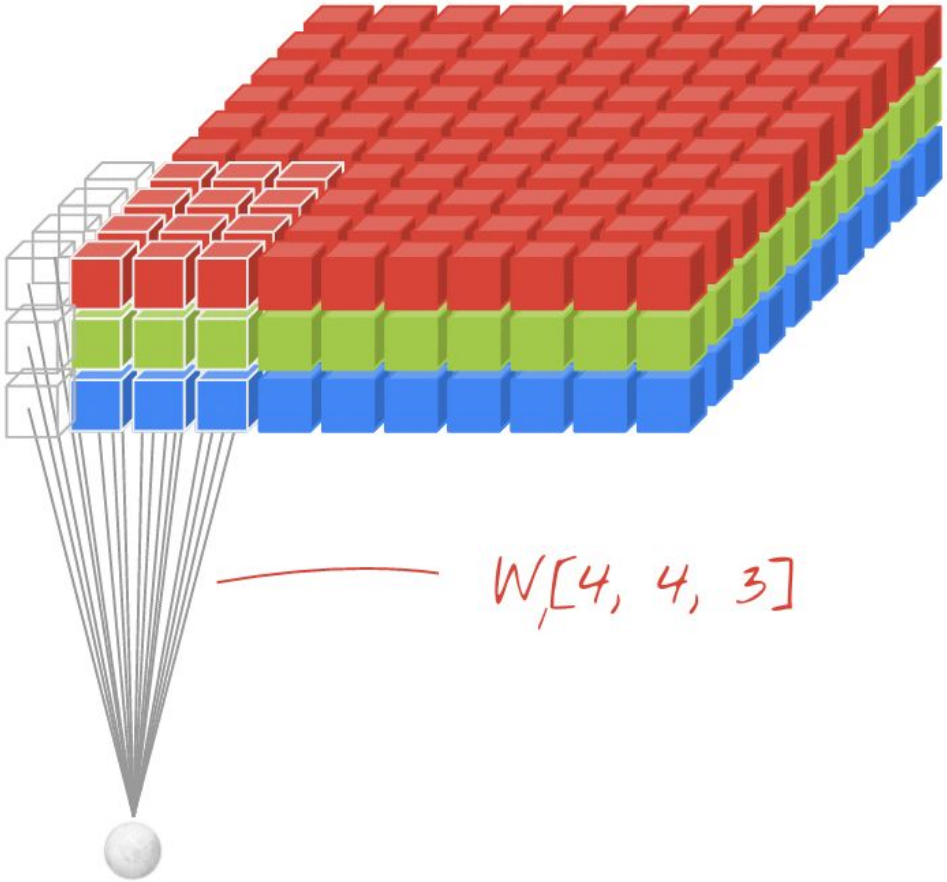


Image

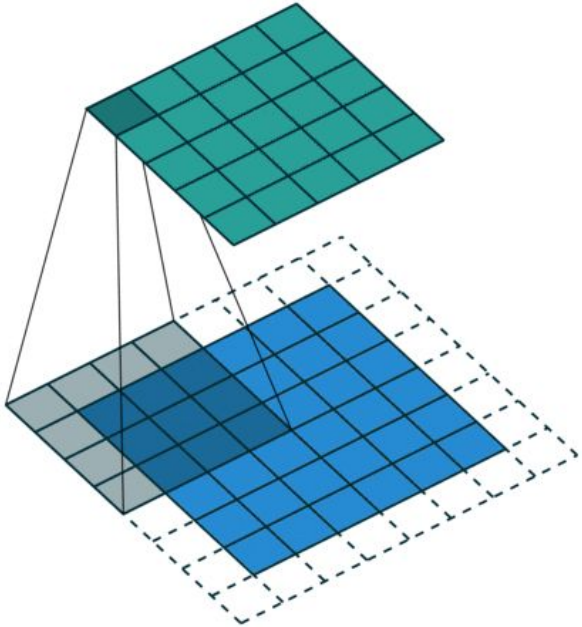


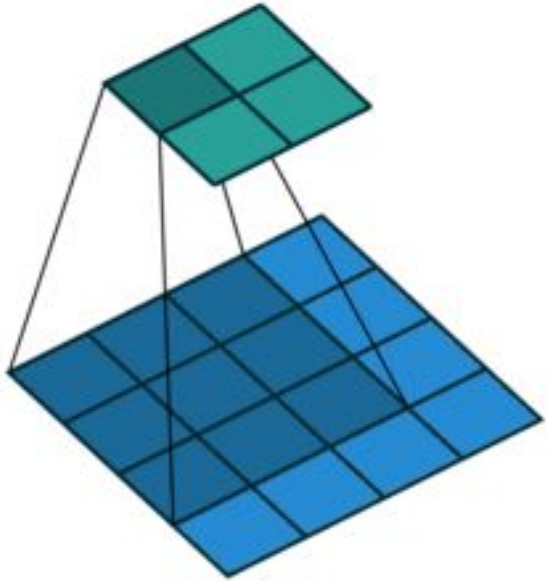
Convolved Feature

$$z(n_1, n_2) = \sum_{k_1=0}^{M_1-1} \sum_{k_2=0}^{M_2-1} x(k_1, k_2) y(n_1 - k_1, n_2 - k_2)$$


W[4, 4, 3]

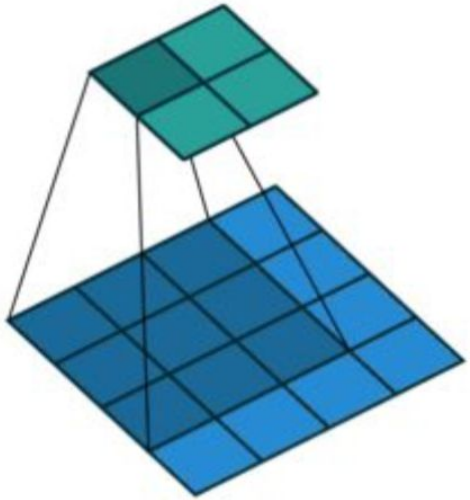
Conv Layer: Variations - Padding



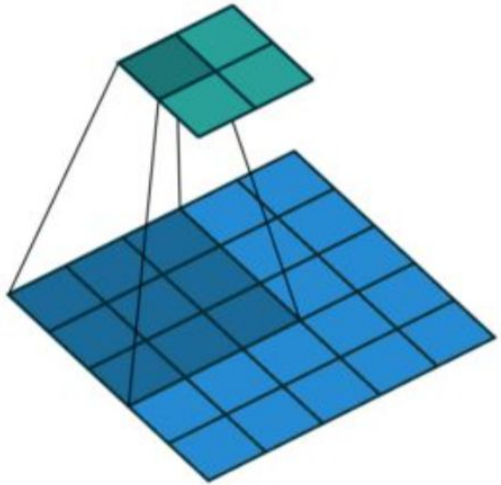


Conv Layer : Variations

2. Stride (to produce smaller output volumes spatially.)



Without stride (i.e., [1,1])

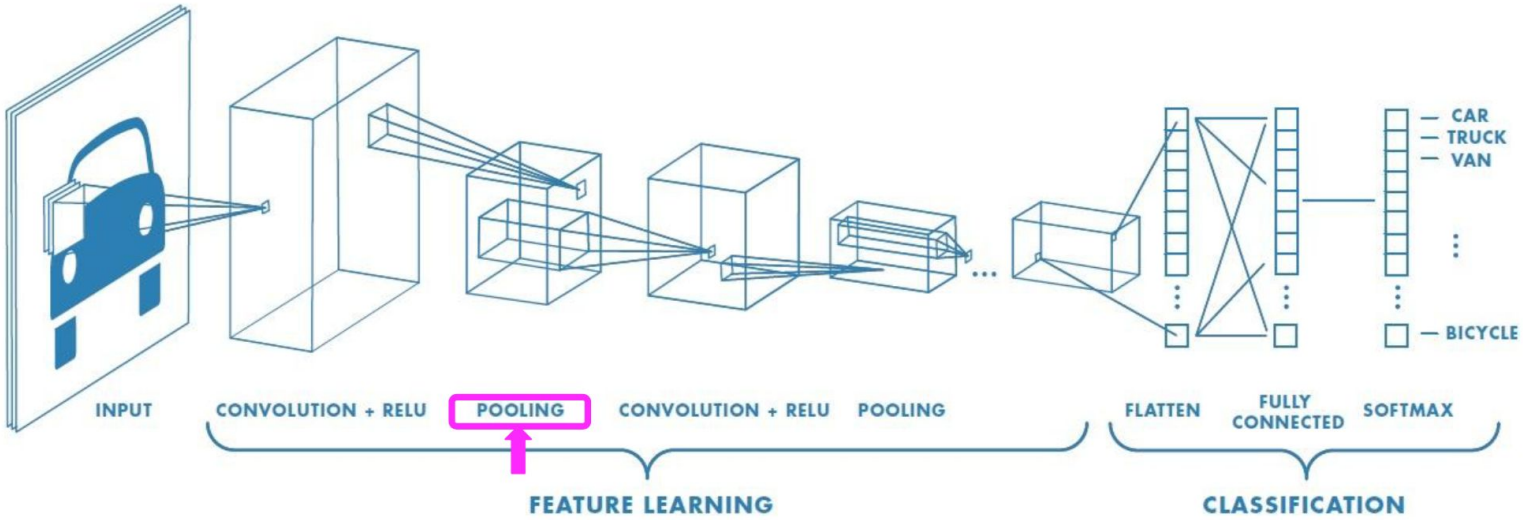


With stride [2,2]

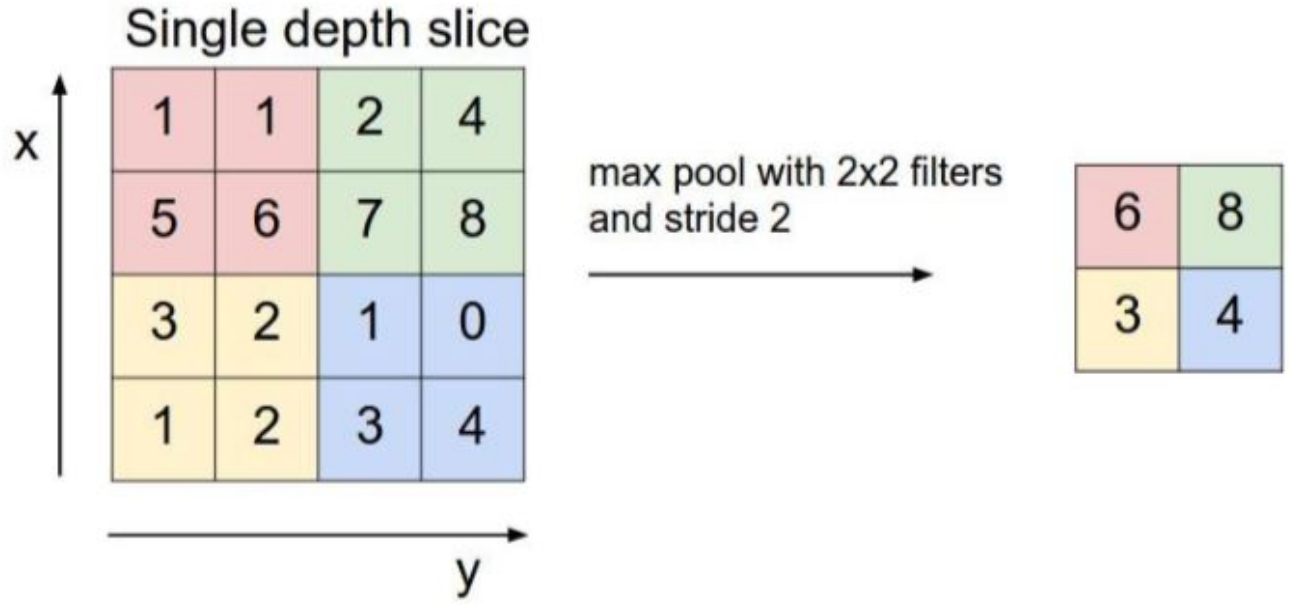
Multiple Channels

- Consider at layer *l*, **H*W*C**
- Kernel **D*D*C**
- Output is **(H - D + 1) * (W - D + 1)* 1**
- Stack K such filters, **(H - D + 1) * (W - D + 1)* K**
- Why?
  - Transforms spatial correspondence into channel
  - Reduce no of params, **K** is your choice.

Pooling


CAR
TRUCK
VAN
BICYCLE
INPUT
CONVOLUTION + RELU
POOLING
CONVOLUTION + RELU
POOLING
FLATTEN
FULLY CONNECTED
SOFTMAX
FEATURE LEARNING
CLASSIFICATION

CNN: Pooling layer

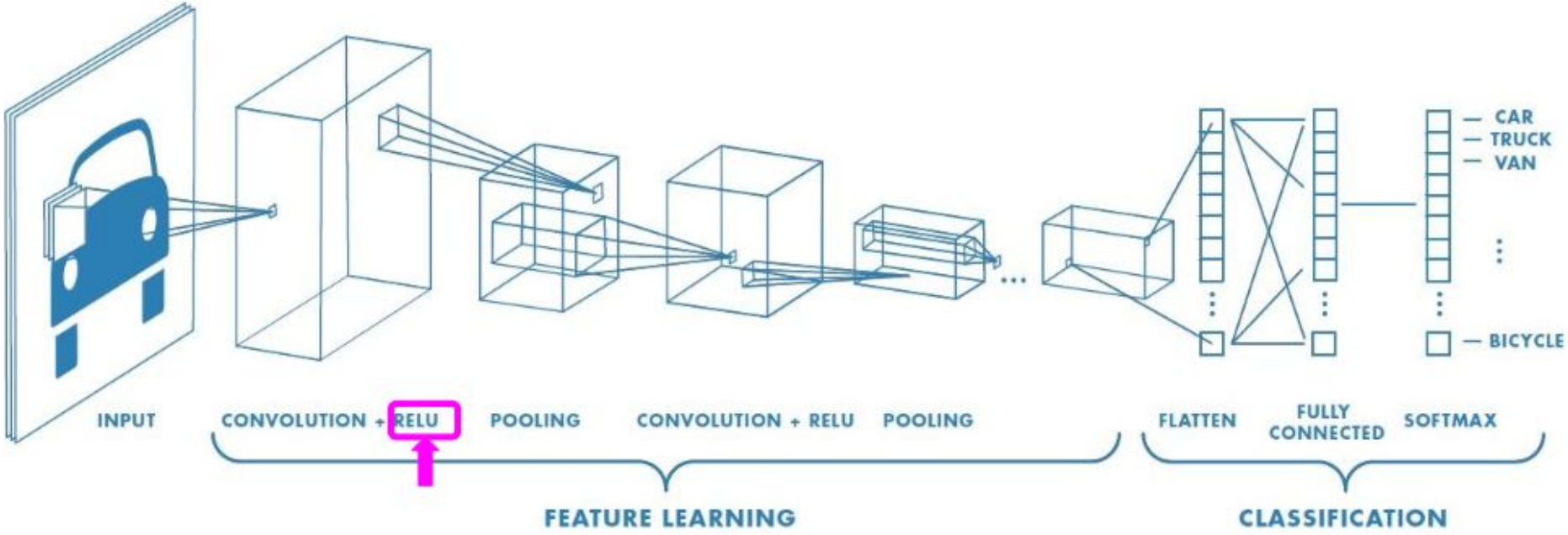


1. To reduce the spatial size of the representation to reduce the amount of parameters and computation in the network.
2. Average pooling or L2 pooling can also be used, but not popular like max pooling.

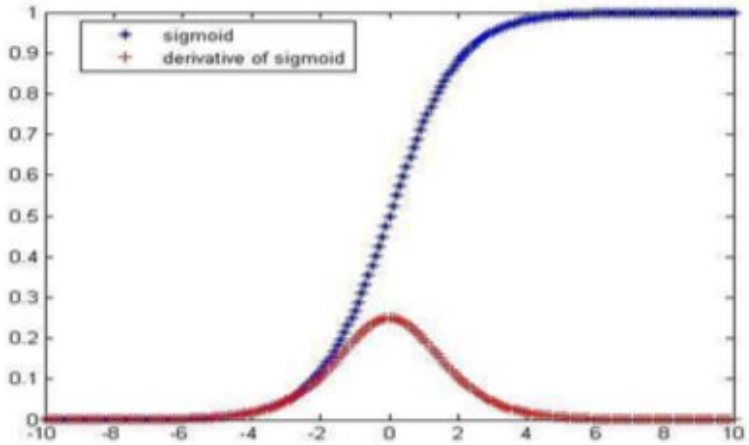
Pooling Layer

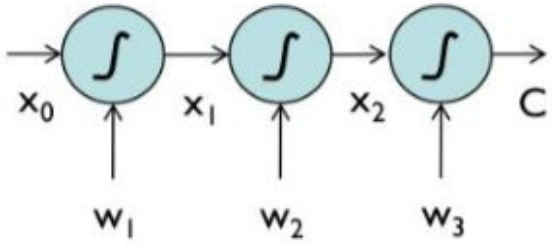
1. Consider:
   - **W1*H1*D1** as input
   - the spatial extent of filter ***F***
   - their stride ***S***
   - the amount of zero padding ***P*** (commonly ***P*** = 0).
2. Produces an output volume of size ***W2* X *H2* X *D2*** where:***W2=(W1−F+2P)/S+1, H2=(H1−F+2P)/S+1, D2=K***
3. Introduces **zero** parameters since it computes a fixed function of the input.

Different layers of CNN architecture



Activation functions: Sigmoidal function





$$x_i = \sigma'(w_i{}^T x_{i-1})$$

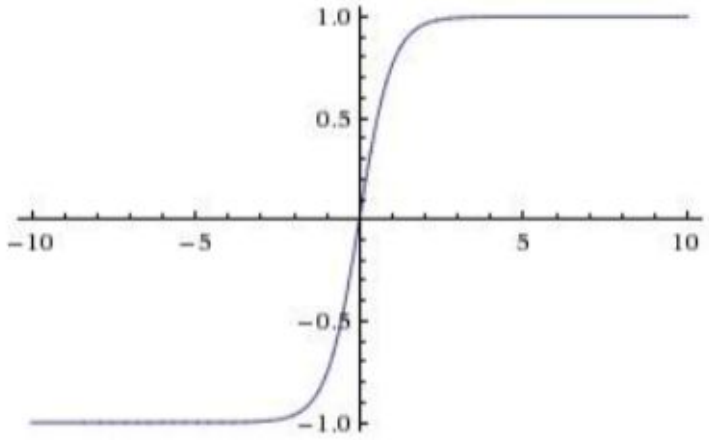
$$\frac{\partial C}{\partial w_1(j)} = \sigma'(w_3{}^T x_2) \cdot x_2(j) \cdot \sigma'(w_2{}^T x_1) \cdot x_1(j) \cdot \sigma'(w_1{}^T x_0) \cdot x_0(j)$$

Drawback 1: *Sigmoids saturate and kill gradients* (when the neuron's activation saturates at either tail of 0 or 1).

0 ⇒ fails to update weights while back-prop.

$$\frac{\partial C}{\partial w_1(i)} = \sigma'(w_3{}^T x_2) \cdot x_2(i) \cdot \sigma'(w_2{}^T x_1) \cdot x_1(i) \cdot \sigma'(w_1{}^T x_0) \cdot x_0(i)$$

Activation functions: tanh function



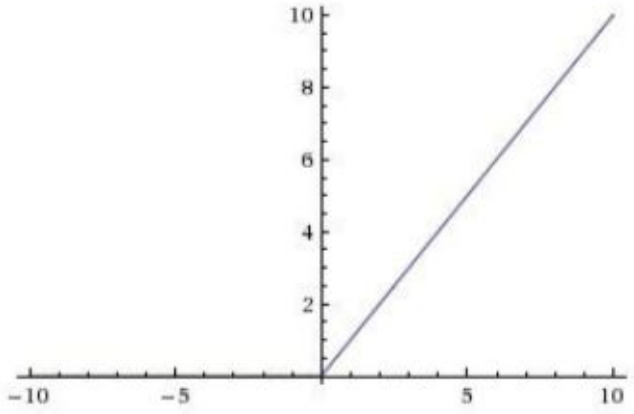
$$y = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

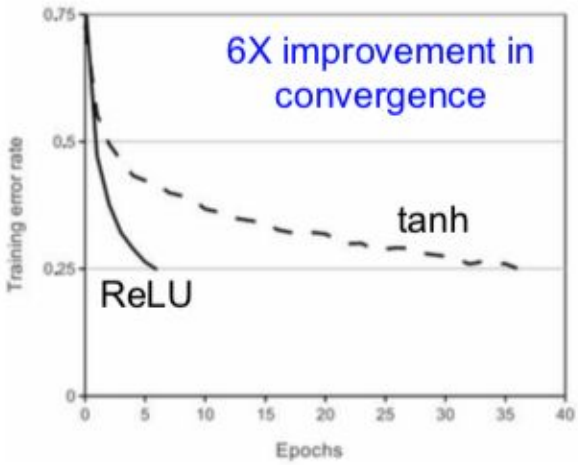
Advantage: **Eliminate** *Undesirable zig-zagging dynamics in the gradient updates for the weights* (because data coming into a neuron can be positive and negative.

$$\frac{\partial C}{\partial w_1(j)} = \sigma'({w_3}^T x_2) \cdot x_2(j) \cdot \sigma'({w_2}^T x_1) \cdot x_1(j) \cdot \sigma'({w_1}^T x_0) \cdot x_0(j)$$

Drawback 1: *But still saturate and kill gradients.*

Activation functions: Rectified Linear Unit (very popular).



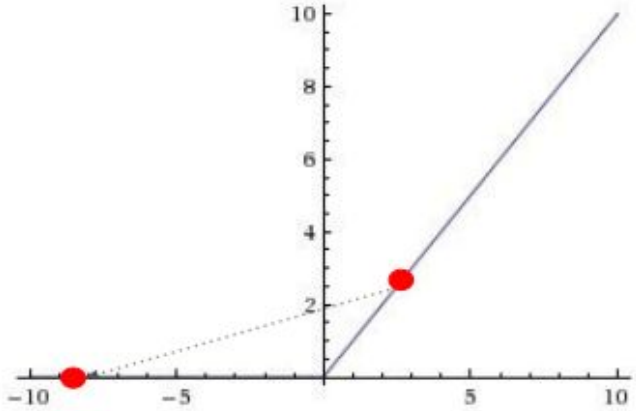


Advantage 1: **Eliminate** *saturation and killing of gradients for positive inputs*.

Advantage 2: Greatly **accelerate** convergence of SGD. (*Krizhevsky et al.* argued that this is due to its linear, non-saturating form.)

Advantage 3: tanh/sigmoid neurons involve expensive operations (exponentials, etc.), whereas ReLU can be implemented by **simply thresholding** activations at zero.

Activation functions: Rectified Linear Unit (very popular).



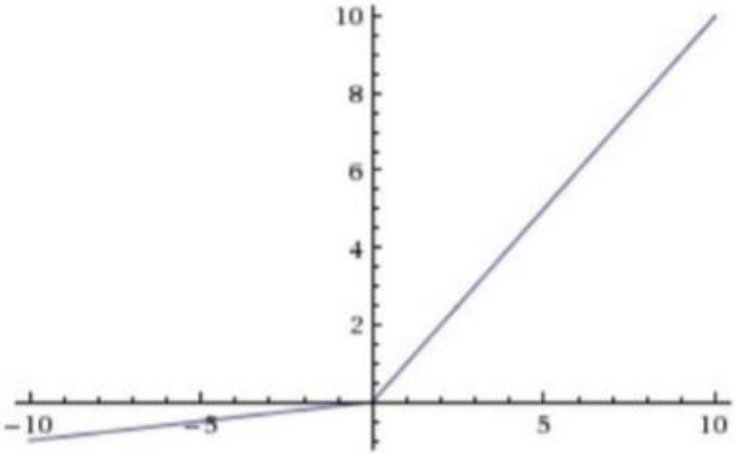
Drawback: ReLU units can **irreversibly die** during training. A large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again.

0 ⇒ fails to update weights while back-prop.

$$\frac{\partial C}{\partial w_1(i)} = \sigma'({w_3}^T x_2) \cdot x_2(i) \cdot \sigma'({w_2}^T x_1) \cdot x_1(i) \cdot \sigma'({w_1}^T x_0) \cdot x_0(i)$$

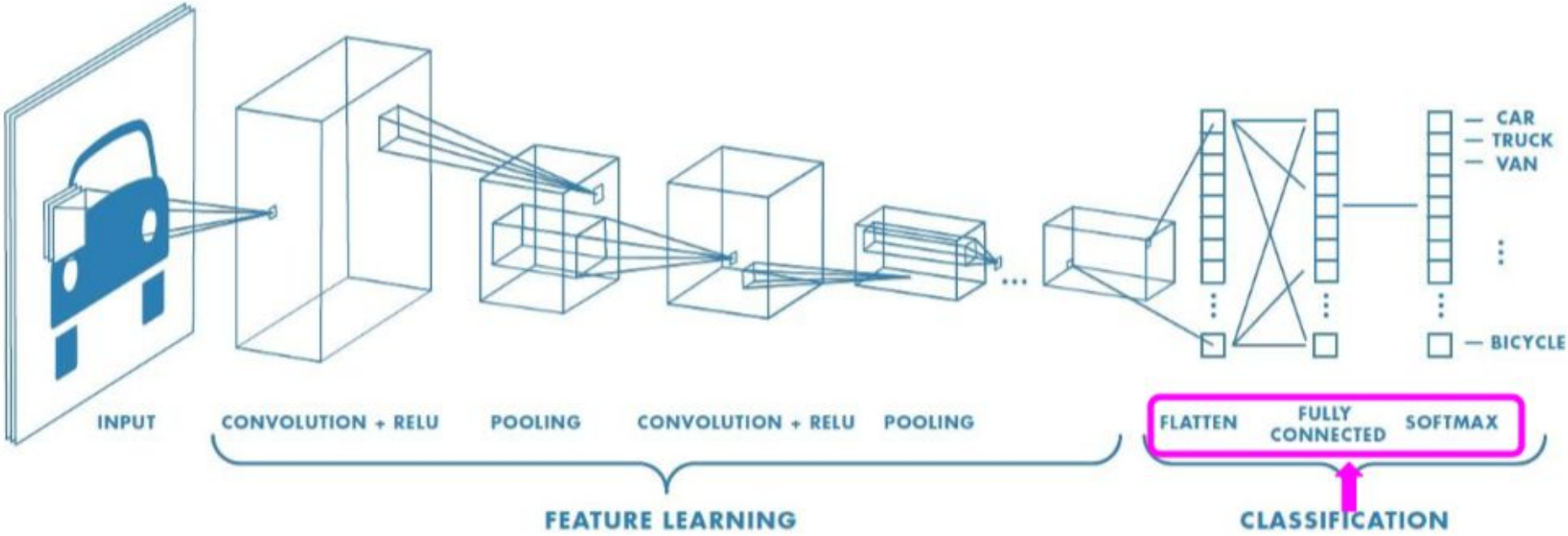
Activation functions: Leaky ReLU.

Instead of the function being zero when $x < 0$, a leaky ReLU will instead have a small negative slope (a hyper-parameter).

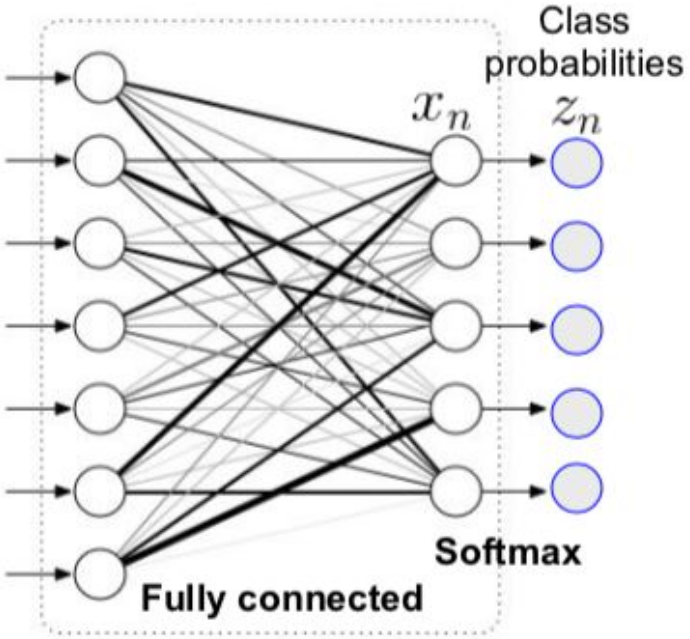


Advantages: **Eliminate** irreversible dying of neurons, as in ReLU.
Have all the advantages of ReLU.

Different layers of CNN architecture



Flattening, fully connected (FC) layer and softmax



Flattening

1. Vectorization (converting ***M X N X D*** tensor to a ***MND X 1*** vector).

FC layer

1. Multilayer perceptron.
2. Generally used in final layers to classify the object.
3. Role of a classifier.

Softmax layer

1. Normalize output as discrete class probabilities.

$$z_n = \frac{e^{x_n}}{\sum_{i=1}^{K} e^{x_i}}$$

Backprop in CNNs

Notations

- l is the lth layer where l = 1,2,...,L
- w l is the weights connecting layer to layer l+1i,j
- bl is the bias at layer l
- x l is defined asi,j
- where o l is the output vector at layer l after the non-linearityi,j
- f(.) is the non-linearity

Forward Propagation

- Mathematical representation

$$x_{i,j}^{l} = \mathrm{rot}_{180^\circ}\left\{\sum_{i'}\sum_{j'} w_{i',j'}^{l} o_{i+i',j+j'}^{l-1} + b_{i,j}^{l}\right\}$$

$$x_{i,j}^{l} = \mathrm{rot}_{180^\circ}\left\{w_{i,j}^{l}\right\} * o_{i,j}^{l-1} + b_{i,j}^{l}$$

$$o_{i,j}^{l} = f(x_{i,j}^{l})$$

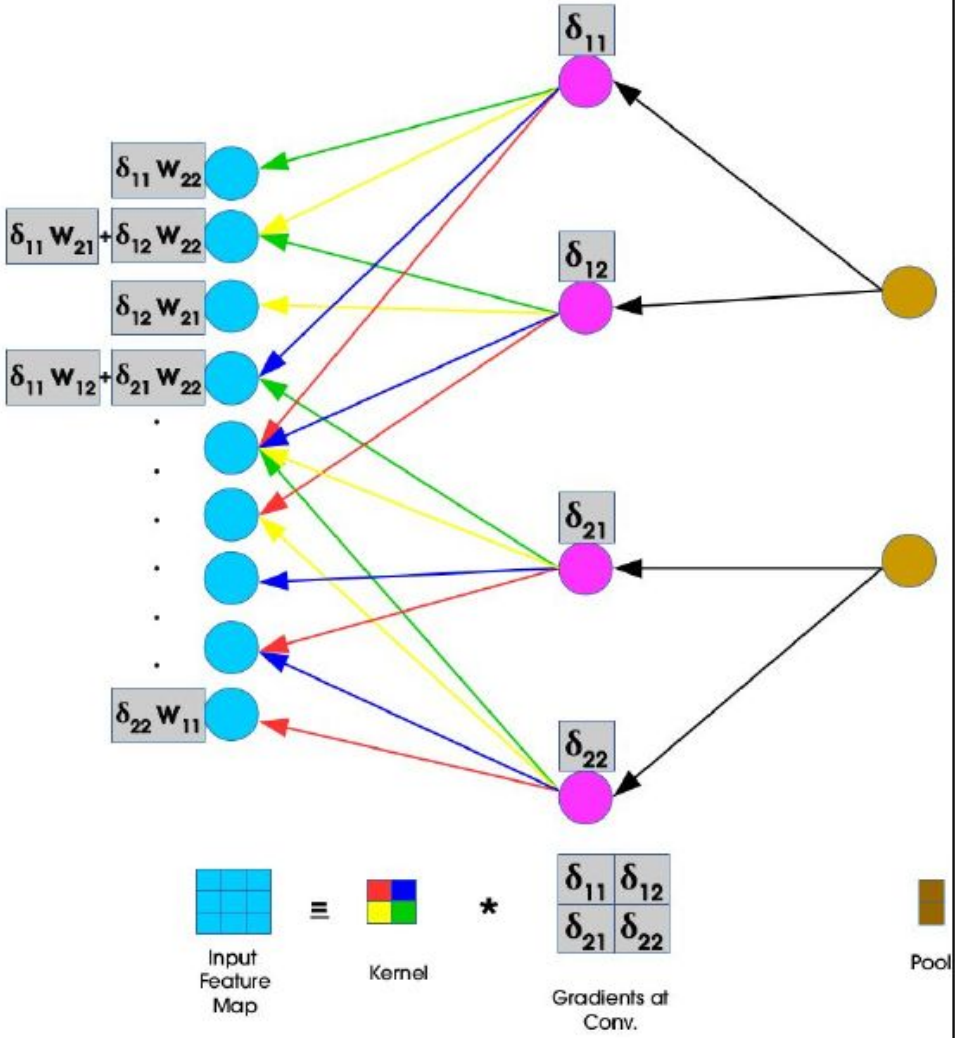
Error / Loss

- t_p - target labels
- a_p^L - predicted labels

$$E = \frac{1}{P}\sum_{p=1}^{P}(t_p - a_p^L)^2$$

Backpropagation

- Two updates are performed
  - For weights
  - For deltas



Backpropagation

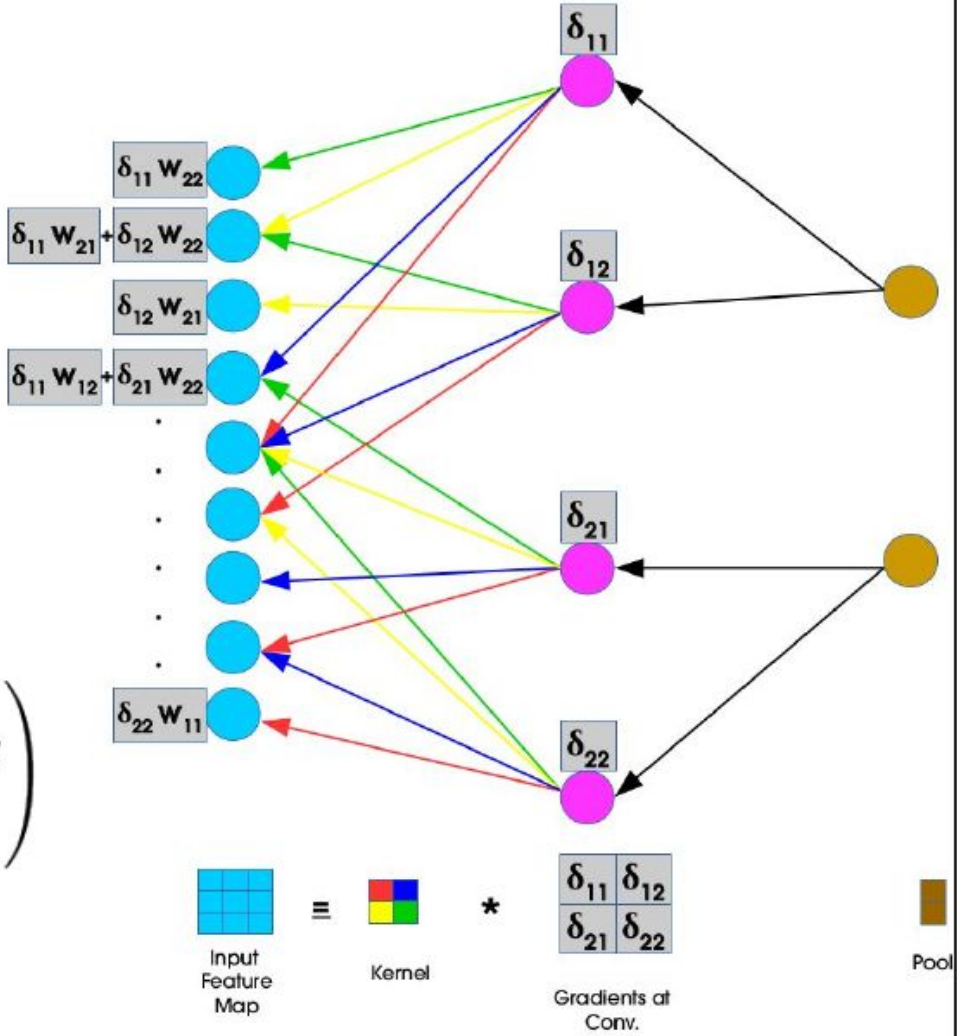
$$\frac{\partial E}{\partial w_{i,j}^l} = \sum_{i'} \sum_{j'} \frac{\partial E}{\partial x_{i',j'}^l} \frac{\partial x_{i',j'}^l}{\partial w_{i,j}^l}$$

$$= \sum_{i'} \sum_{j'} \delta_{i',j'}^l \frac{\partial x_{i',j'}^l}{\partial w_{i,j}^l}$$

Where,

$$\frac{\partial x_{i',j'}^l}{\partial w_{i,j}^l} = \frac{\partial}{\partial w_{i,j}^l} \left(\sum_{i''} \sum_{j''} w_{i'',j''}^l o_{i'-i'',j'-j''}^{l-1} + b^l \right)$$

$$= o_{i'-i,j'-j}^{l-1}$$



Backpropagation- Weight update

$$\frac{\partial E}{\partial w_{i,j}^{l}} = \sum_{i'} \sum_{j'} \delta_{i',j'}^{l} o_{i'-i,j'-j}^{l-1}$$

$$= \mathrm{rot}_{180^\circ} \left\{ \sum_{i'} \sum_{j'} \delta_{i',j'}^{l} o_{i'+i,j'+j}^{l-1} \right\}$$

$$= \mathrm{rot}_{180^\circ} \left\{ \delta_{i,j}^{l} \right\} * o_{i,j}^{l-1}$$

Delta update

$$\delta_{i,j}^{l} = \frac{\partial E}{\partial x_{i,j}^{l}}$$

$$\frac{\partial E}{\partial x_{i,j}^{l}} = \sum_{i'} \sum_{j'} \frac{\partial E}{\partial x_{i',j'}^{l+1}} \frac{\partial x_{i',j'}^{l+1}}{\partial x_{i,j}^{l}}$$

$$= \sum_{i'} \sum_{j'} \delta_{i',j'}^{l+1} \frac{\partial x_{i',j'}^{l+1}}{\partial x_{i,j}^{l}}$$

$$\frac{\partial x_{i',j'}^{l+1}}{\partial x_{i,j}^{l}} = \frac{\partial}{\partial x_{i,j}^{l}} \left(\sum_{i''} \sum_{j''} w_{i'',j''}^{l+1} o_{i'-i'',j'-j''}^{l} + b^{l+1} \right)$$

$$= \frac{\partial}{\partial x_{i,j}^{l}} \left(\sum_{i''} \sum_{j''} w_{i'',j''}^{l+1} f\left(x_{i'-i'',j'-j''}^{l} \right) + b^{l+1} \right)$$

Delta Update

$$
\begin{aligned}
\frac{\partial x_{i',j'}^{l+1}}{\partial x_{i,j}^{l}} &= \frac{\partial}{\partial x_{i,j}^{l}} \left(w_{0,0}^{l+1} f\left(x_{i'-0,j'-0}^{l}\right) + \cdots + w_{i'-i,j'-j}^{l+1} f\left(x_{i,j}^{l}\right) + \cdots + b^{l+1} \right) \\
&= \frac{\partial}{\partial x_{i,j}^{l}} \left(w_{i'-i,j'-j}^{l+1} f\left(x_{i,j}^{l}\right) \right) \\
&= w_{i'-i,j'-j}^{l+1} \frac{\partial}{\partial x_{i,j}^{l}} \left(f\left(x_{i,j}^{l}\right) \right) \\
&= w_{i'-i,j'-j}^{l+1} f'\left(x_{i,j}^{l}\right)
\end{aligned}
$$

Where $f'(x_{ij}^{l})$ is the derivative of the activation function $f(.)$

Delta update

$$\frac{\partial E}{\partial x_{i,j}^{l}} = \sum_{i'} \sum_{j'} \delta_{i',j'}^{l+1} w_{i'-i,j'-j}^{l+1} f'\left(x_{i,j}^{l}\right)$$

$$= \mathrm{rot}_{180^\circ} \left\{ \sum_{i'} \sum_{j'} \delta_{i',j'}^{l+1} w_{i'+i,j'+j}^{l+1} \right\} f'\left(x_{i,j}^{l}\right)$$

$$= \mathrm{rot}_{180^\circ} \left\{ \sum_{i'} \sum_{j'} \delta_{i'+i,j'+j}^{l+1} w_{i',j'}^{l+1} \right\} f'\left(x_{i,j}^{l}\right)$$

$$= \delta_{i,j}^{l+1} * \mathrm{rot}_{180^\circ} \left\{ w_{i,j}^{l+1} \right\} f'\left(x_{i,j}^{l}\right)$$

Where $f'(x_{ij}^{l})$ is the derivative of the activation function $f(.)$

Derivatives of some common activation functions

- Relu: $f'(x_{ij}^l) = \begin{cases} 1 & x_{ij}^l > 0 \\ 0 & x_{ij}^l \leq 0 \end{cases}$
- Sigmoid: $f'(x_{ij}^l) = (f(x_{ij}^l))(1 - f(x_{ij}^l))$
- Tanh: $f'(x_{ij}^l) = 1 - f(x_{ij}^l)^2$

Backprop in Pooling Layer

- Max Pooling
  - the error is just assigned to where it comes from
- Average Pooling
  - The error is multiplied by 1/(NxN) and assigned to the whole pooling block

BatchNorm

The need for normalisation

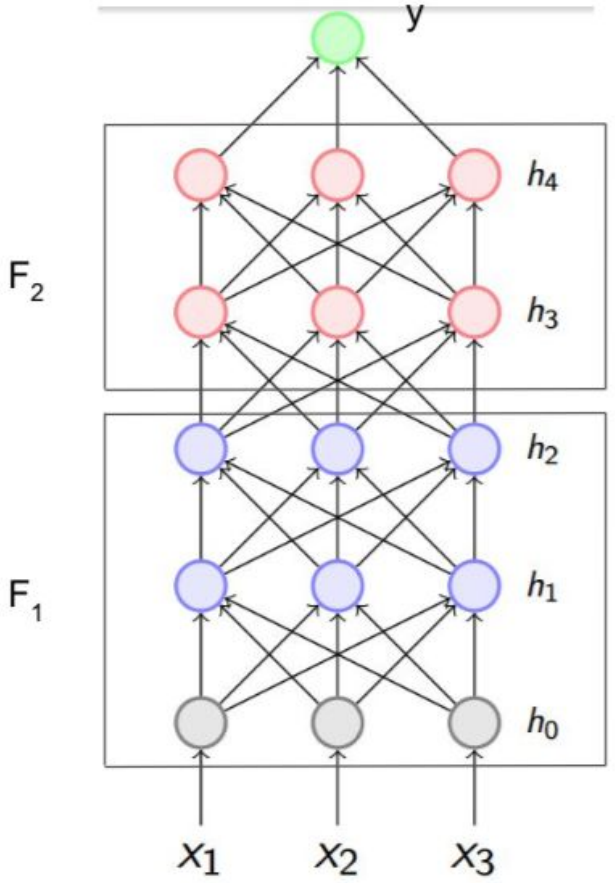
- Normalisation in general, even with correlated features speeds up training
- training complicated by fact that the inputs to each layer are affected by the parameters of all preceding layers
- small changes to the network parameters amplify as the network becomes deeper.
- Called **Internal Covariate Shift**

Motivation

- Consider a deep neural network

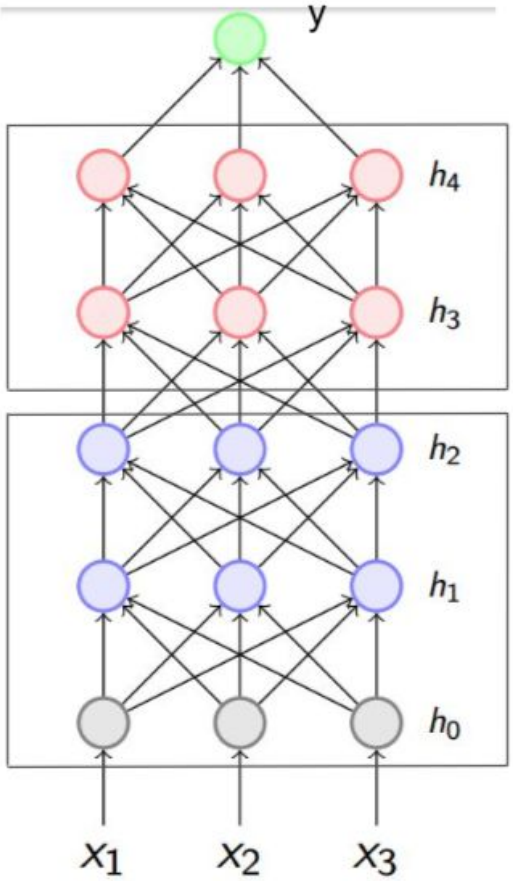
$$y = F_2(F_1(x; \theta_1); \theta_2)$$

$$y' = F_1(x; \theta_1)$$



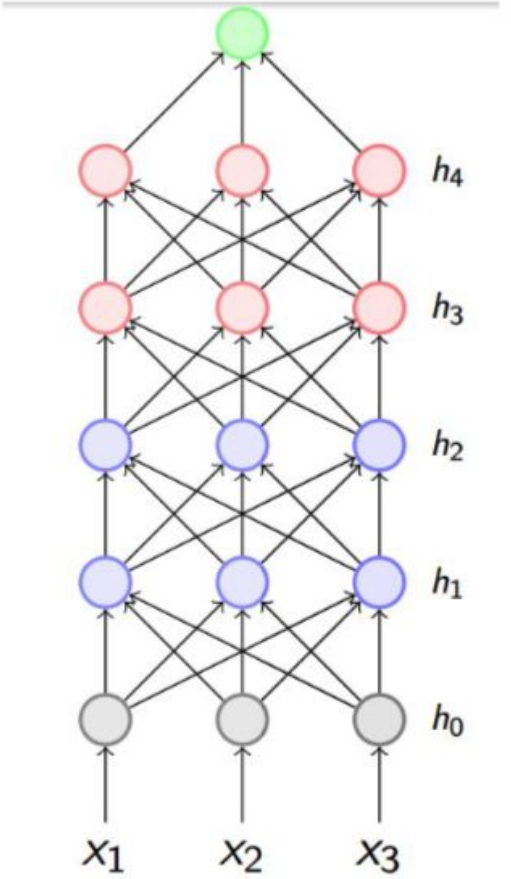
Motivation

- Consider a deep neural network
- P(x) remains constant
- P(y') keeps changing since parameters θ_1 are changing with each iteration
- Machine learning model assumes that data is sampled from the same distribution each time
- For F_2 this assumption is violated



Motivation

- It's desirable to have the output distribution of each layer as zero mean and unit variance Gaussian
- Why not explicitly transform the output distribution of each layer to a zero mean and unit variance Gaussian?



Solutions

- Whiten the inputs (LeCun, 1998):
  - Costly to do for each input (to each layer)
  - Need to compute Covariance matrix
- Also, if normalisation computed outside gradient step, model could blow up.
- Even with mini-batch, dont want to compute Cov matrix

Tractable solution - Batch normalization

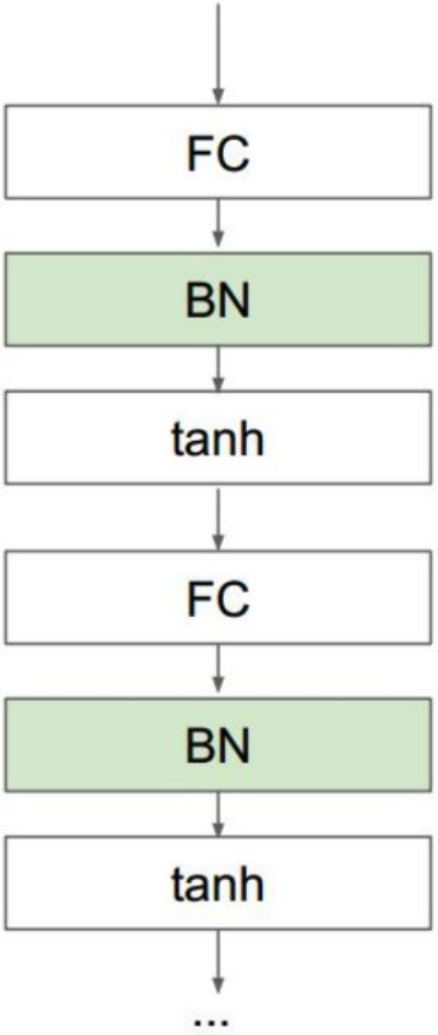
- Consider a batch of activations x^k at any layer and apply

$$\hat{x}^k = \frac{x^k - E[x^k]}{\sqrt{Var[x^k]}}$$

- Is this differentiable?

Batch normalization (BN) layer

- Where to insert the Batch Normalization (BN) layer



Batch Norm algorithm

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Credits: BN paper, Sergey, Szegedy.

Advantages of BN

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization
- Accelerates training

During Inference>>>

- Set beta and gamma from the last run (last batch).
- Caveat: Donot use BN on batch size of 1, with less data
- Can be stochastic, unstable.

Summary

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

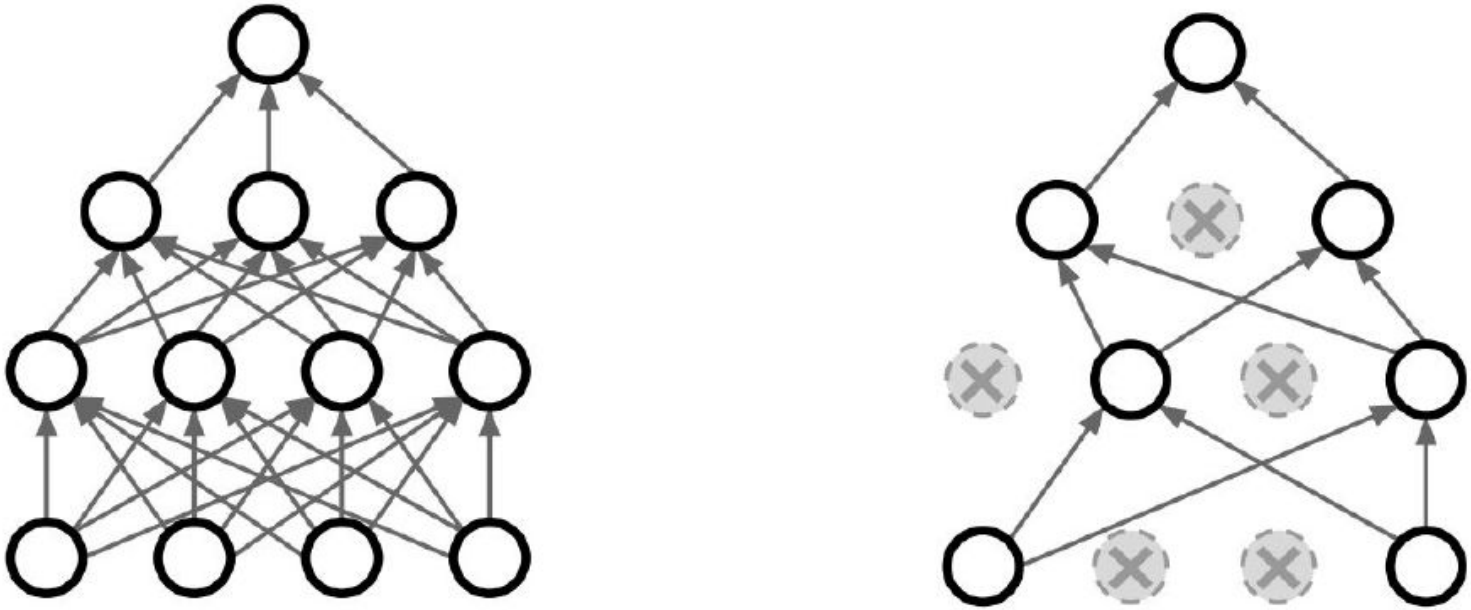
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Dropout

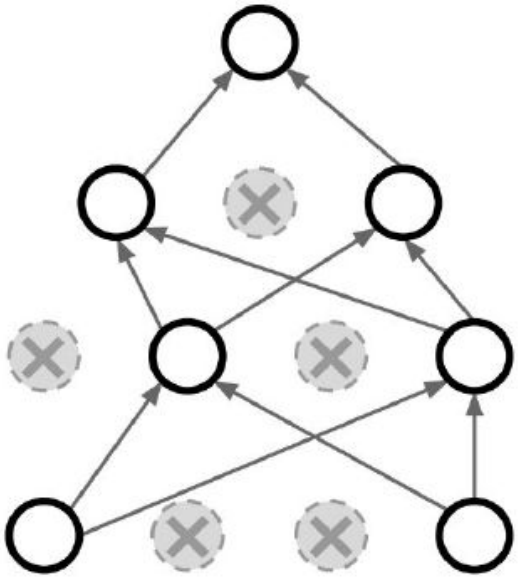
- A type of regularization
  - A method of ensemble learning applied to neural networks
- Impractical to train many neural networks
  - Dropout makes it practical

Dropout

- In each forward pass randomly set some units to zero.



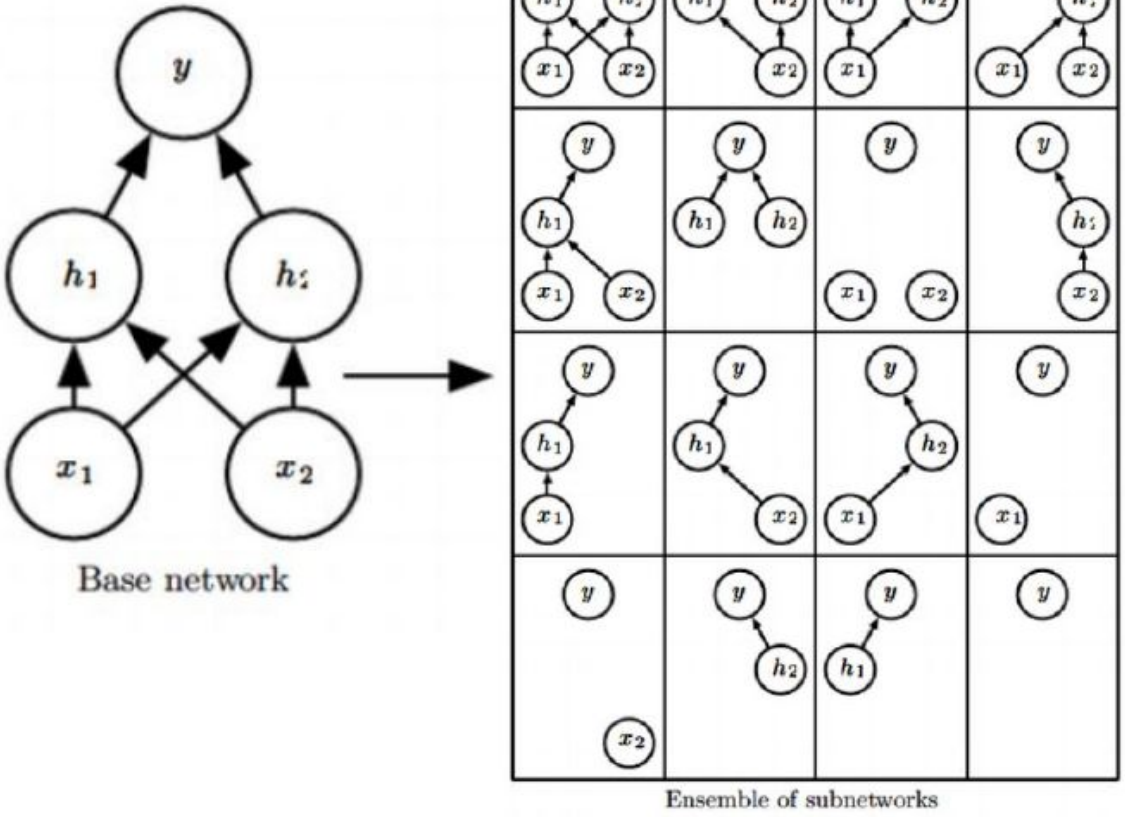
Why Dropout is a good idea?



Forces the network to have a redundant representation;
Prevents co-adaptation of features

Dropout - another interpretation

Ensemble of networks

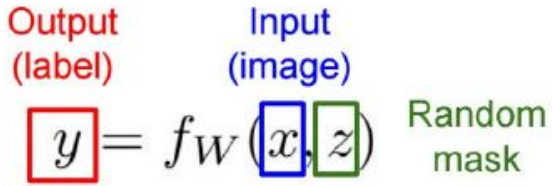


Dropout training

- At each step randomly sample a binary mask
  - Probability of including a unit is a hyperparameter - typically 0.5
- Multiply the units by the binary mask
  - Forward prop proceeds as usual

Dropout - test time

- Dropout makes the output random
- Average out randomness



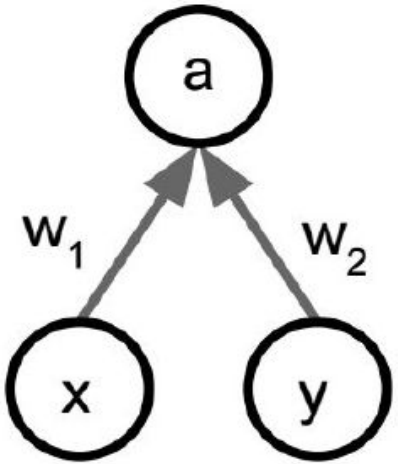
$$y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$$

Integral is hard to compute

Dropout - test time

- Approximate the integral $y = f(x) = E_z[f(x,z)] = \int p(z) f(x,z) dz$

Consider a single neuron



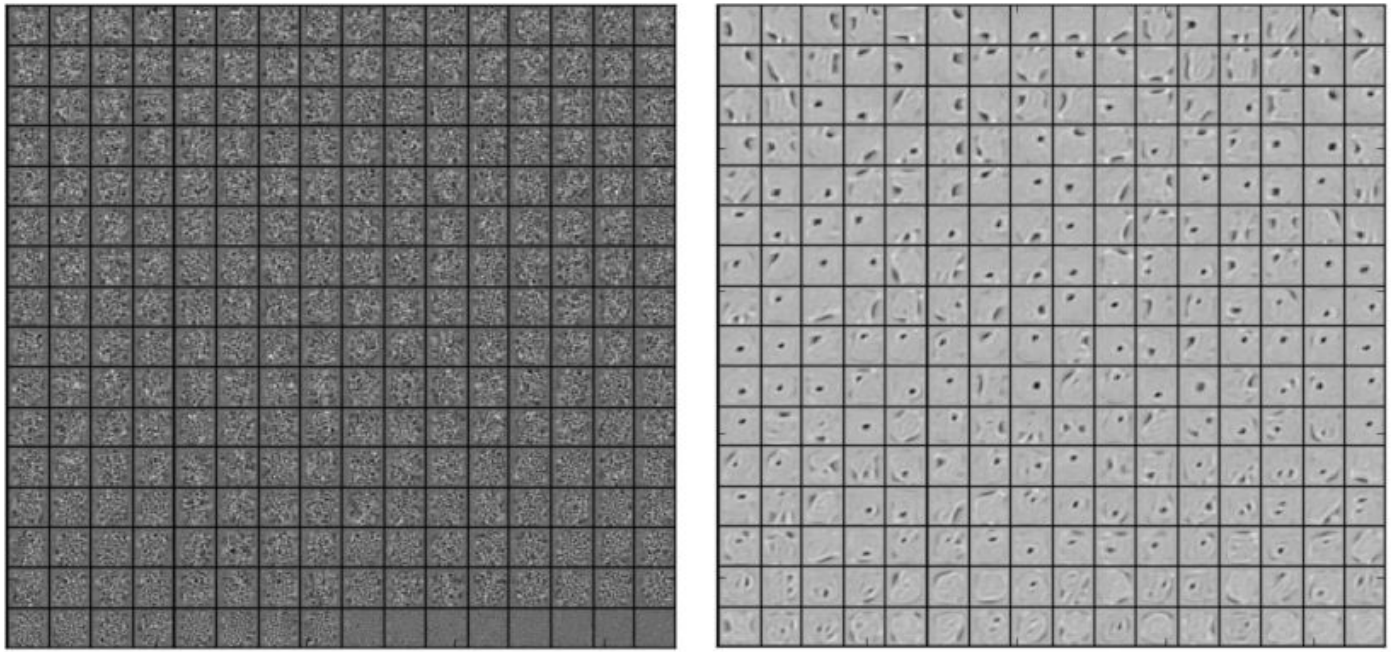
At test time, we have $E[a] = w_1 x + w_2 y$

During training, we have

$$E[a] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) = \frac{1}{2}(w_1 x + w_2 y)$$

At test time, multiply by dropout probability

Effects of using Dropout

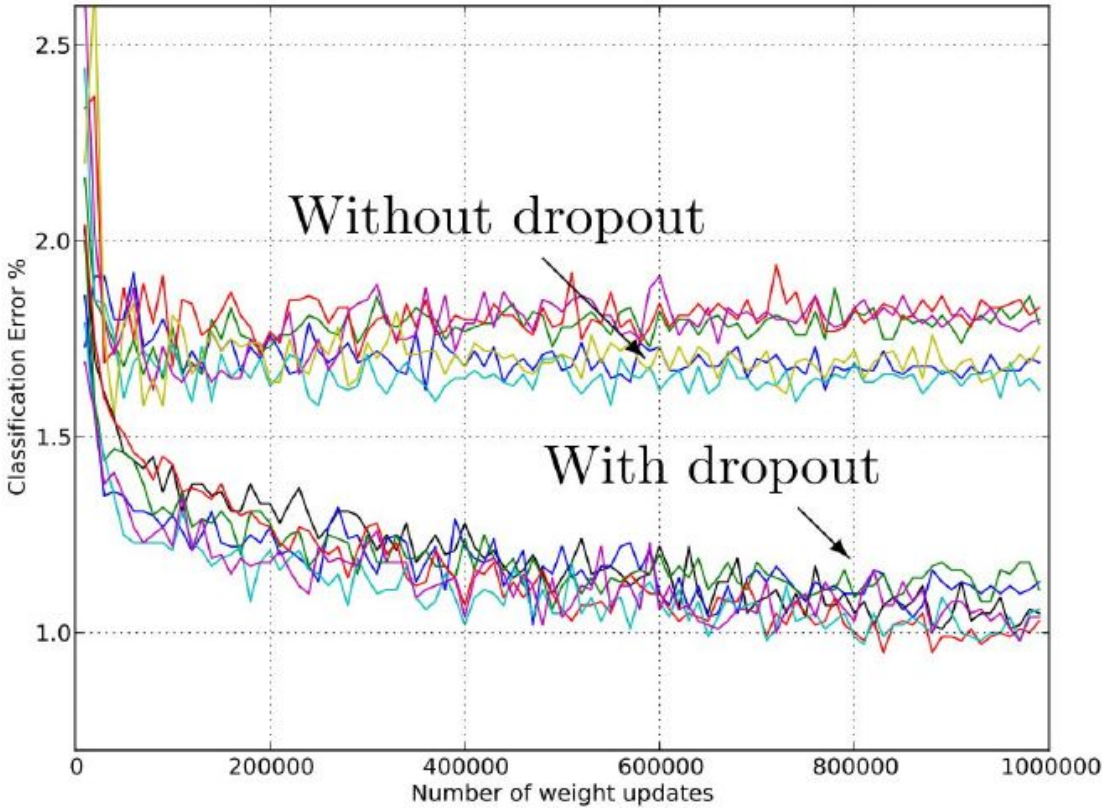


(a) Without dropout

(b) Dropout with $p = 0.5$.

Effects on a set of feature detectors taken from a convolutional neural network. While the features in (a) are mostly indistinguishable for humans, and seem to contain a big portion of white noise, the features in (b) are the product of a training with dropout. As visible the detectors are able to filter meaningful features such as spots, corners and strokes in an image. Image source: https://wiki.tum.de/display/lfdv/Dropout

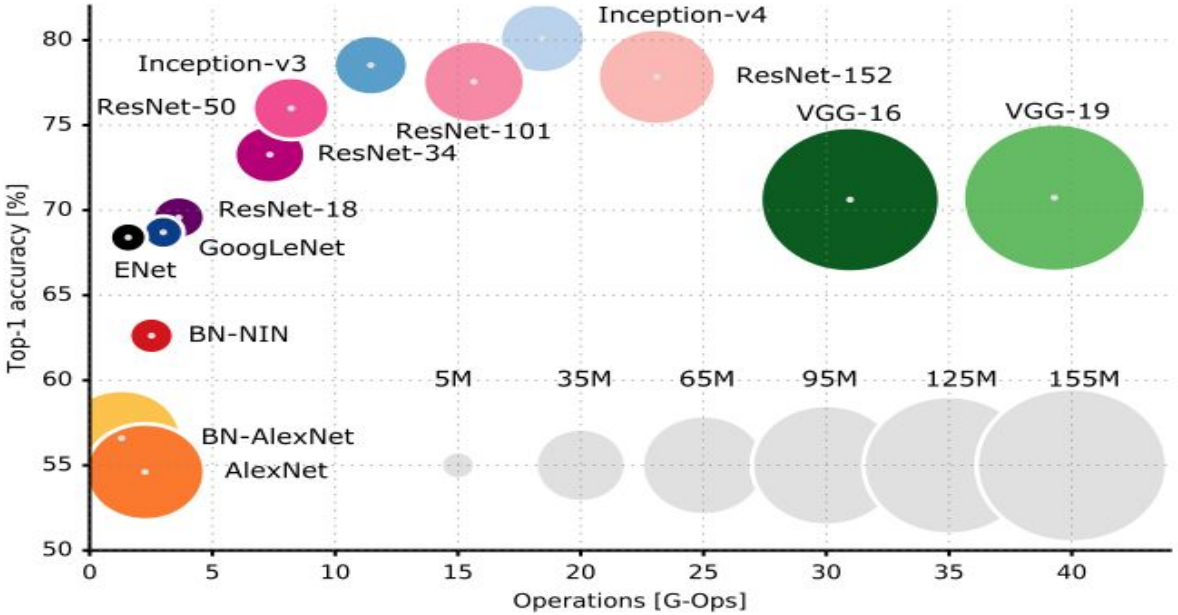
Effects of using Dropout



Comparison of test error with and without dropout. Each color represents a different network architecture, ranging from 2 to 4 hidden layers (each fully connected) and 1024 to 2048 units. Image source: https://wiki.tum.de/display/lfdv/Dropout

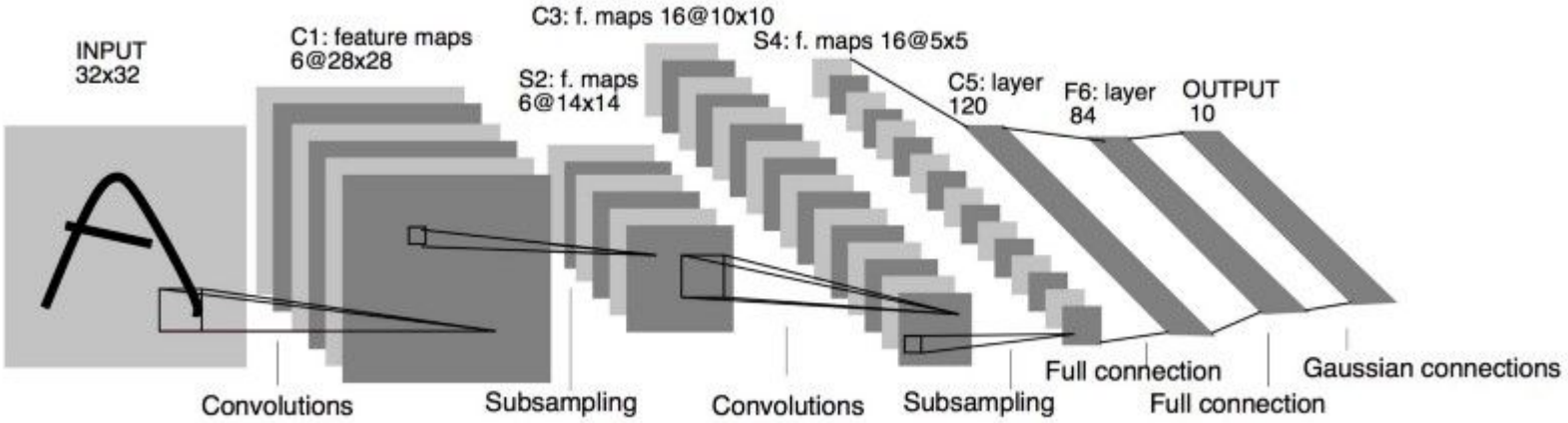
CNN Architectures

ConvNet architectures



LENET5

- Implemented in 1994 , one of the very first convolutional neural networks, and what propelled the field of Deep Learning. This pioneering work by Yann LeCun was named LeNet5 after many previous successful iterations since the year 1988.

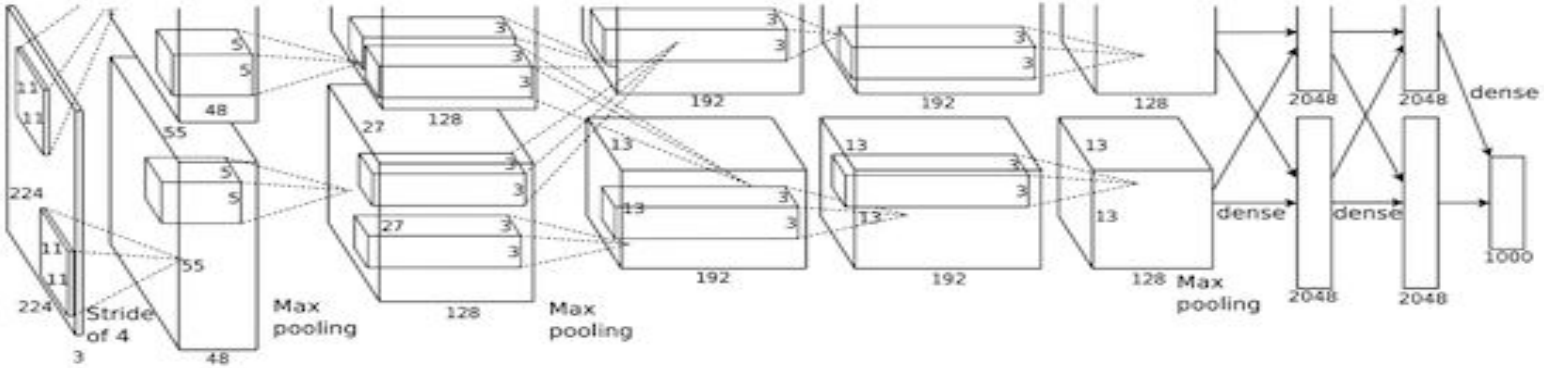


LENET5

- use sequence of 3 layers: convolution, pooling, non-linearity
- use convolution to extract spatial features
- non-linearity in the form of tanh or sigmoids (no ReLus back then)
- multi-layer neural network (MLP) as final classifier

AlexNET

- Brought DL back to mainstream in 2012, when Alex Krizhevsky released AlexNet which was a deeper and much wider version of the LeNet and won by a large margin the difficult ImageNet competition.



AlexNet

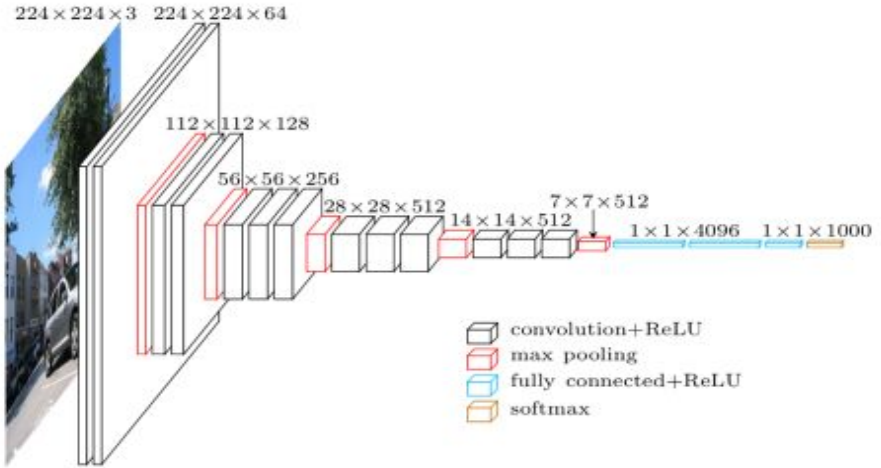
- use of rectified linear units (ReLU) as non-linearities
- use of dropout technique (Hinton *et al.*) to selectively ignore single neurons during training, a way to avoid overfitting of the model
- overlapping max pooling, avoiding the averaging effects of average pooling
- use of GPUs (NVIDIA GTX 580) to reduce training time

The success of AlexNet started a small revolution. Convolutional neural network were now the workhorse of Deep Learning, which became the new name for “large neural networks that can now solve useful tasks”.

VGG

- first to use much smaller 3×3 filters in each layer
- insight that multiple 3×3 convolution can replace 5x5 and 7x7 convolutions

- Fewer params than Alexnet, thrice as deep.
- VGG 16, 19.



Different VGG Architectures

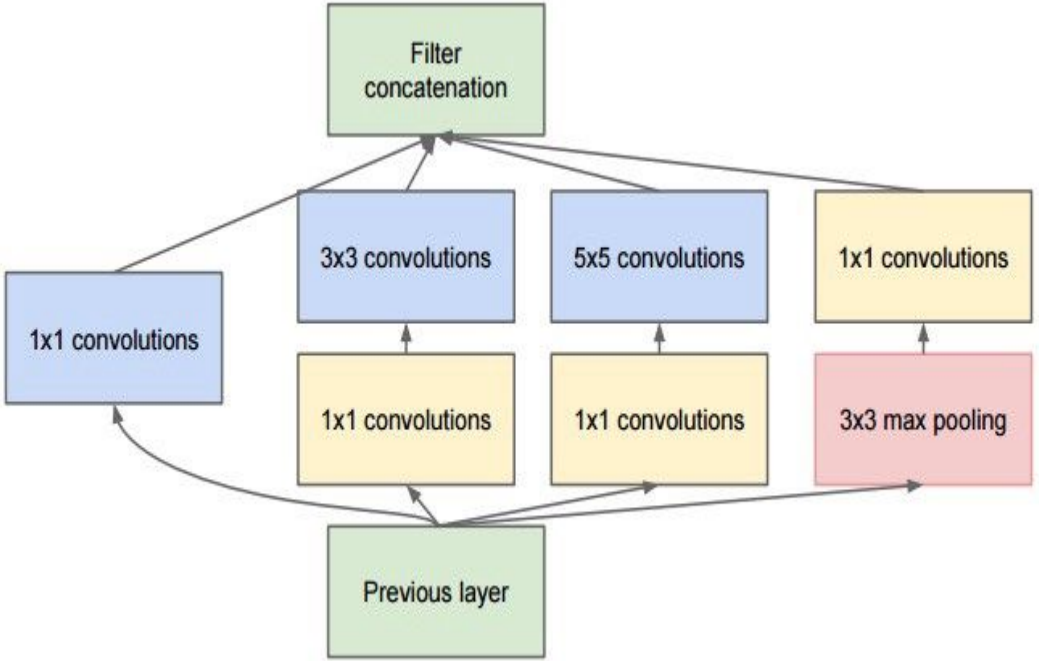
ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	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					

Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

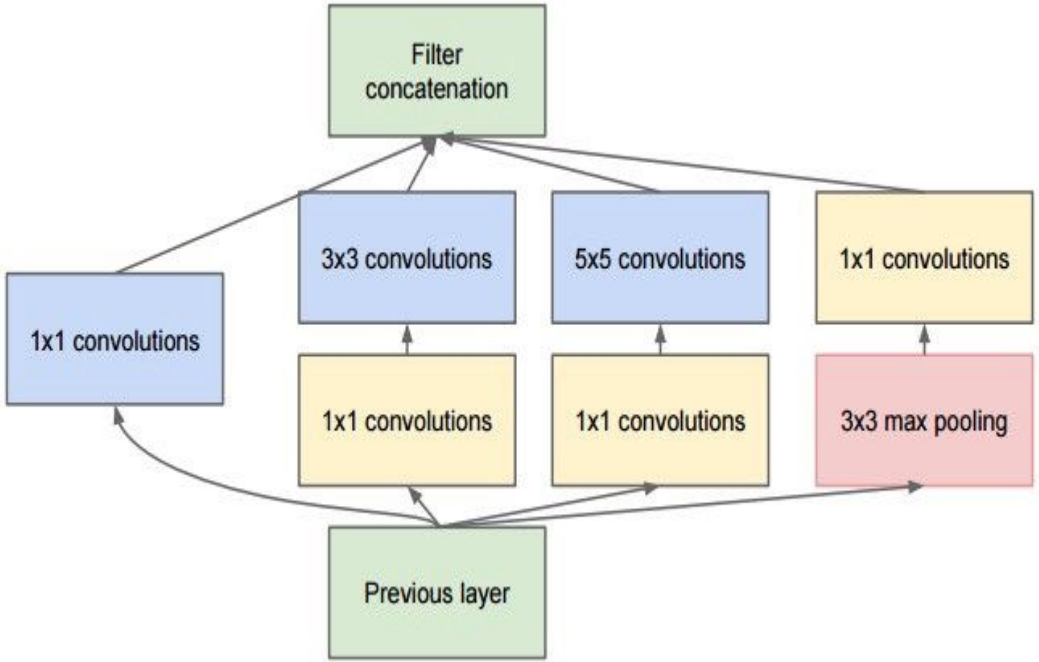
GoogLeNet and Inception

- Christian Szegedy and team from Google,
- aimed at reducing the computational burden of deep neural networks,
- devised GoogLeNet in 2014
- Won Imagenet that year.

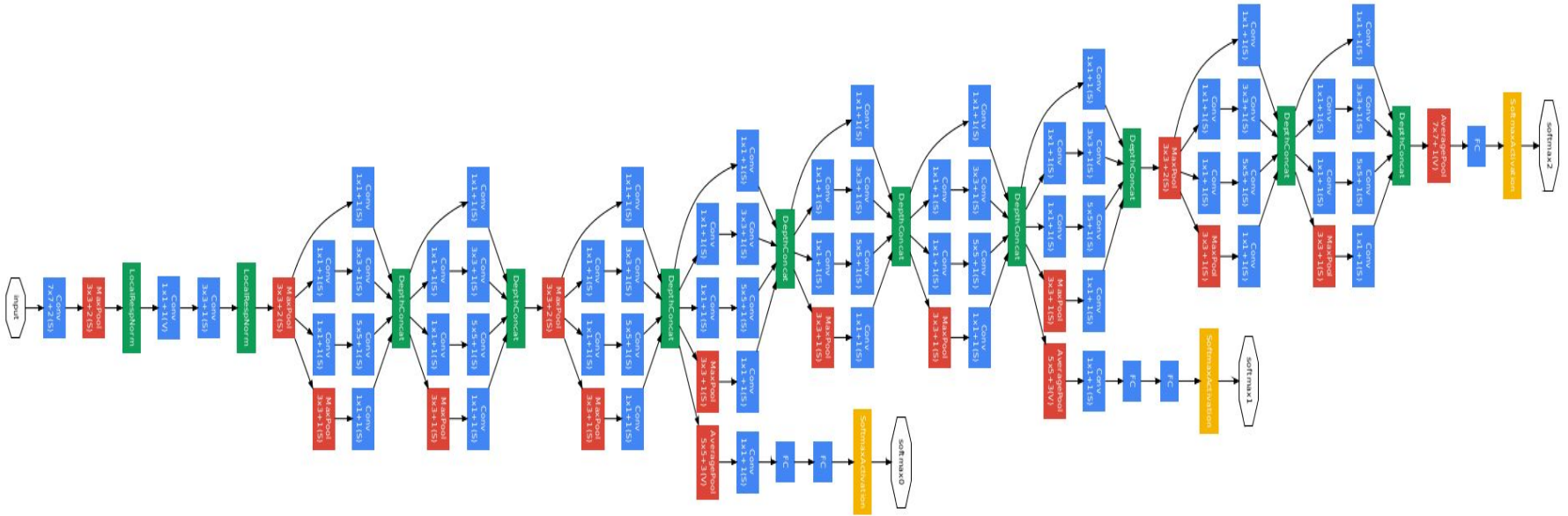


Inception Block

- Combination of 1×1, 3×3, and 5×5 convolutional filters
- Emulates Network in Network (NiN)
- 1x1 Convolutions save params
- Called Bottleneck

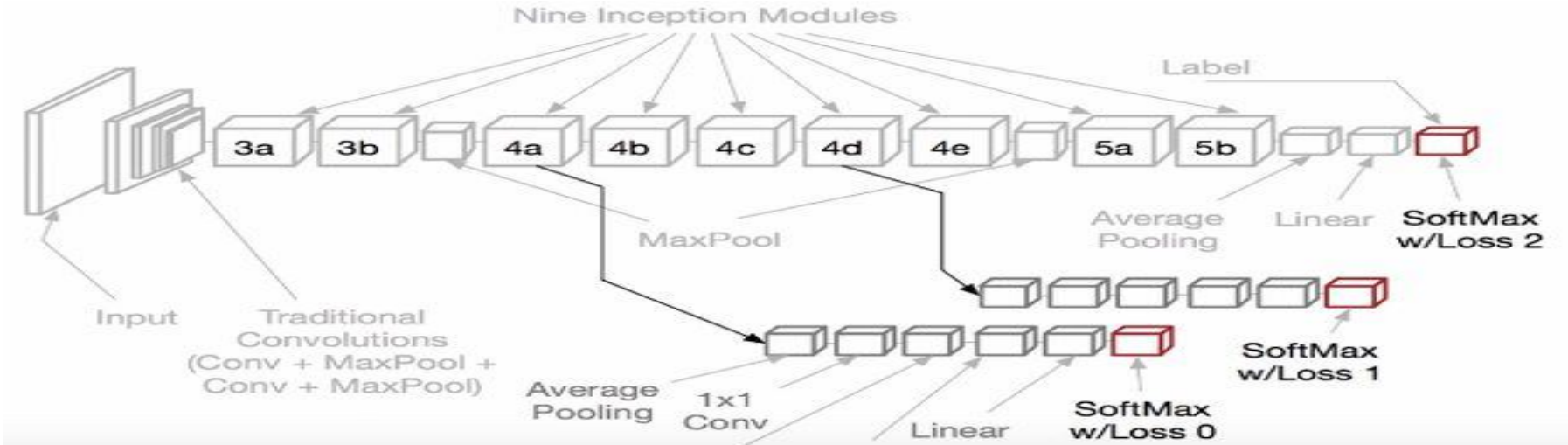


GoogLeNet and Inception



Why multiple softmaxes?

- 22 layers, danger of the vanishing gradients problem during training
- Added multiple softmaxes at inception 4a, 4d
- These blocks may learn meaningful representations
- Discarded at inference



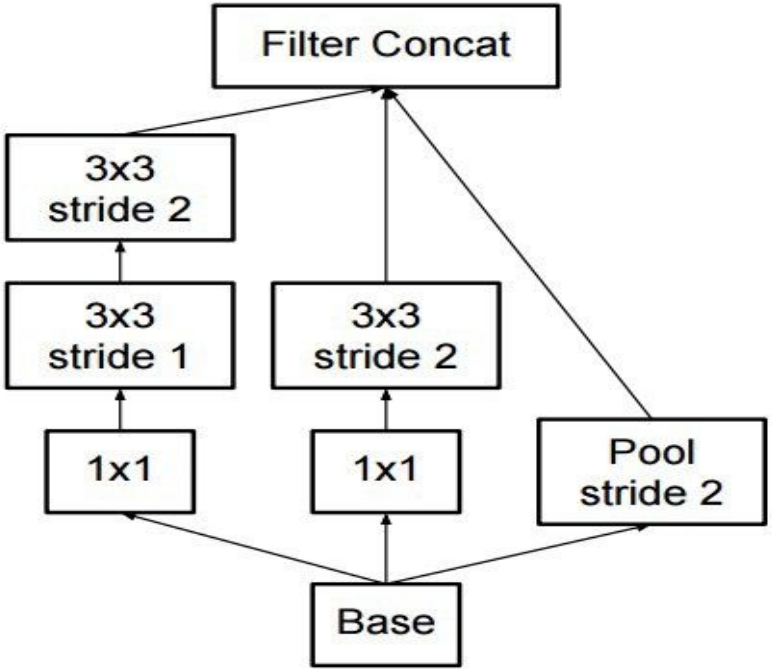
Inception V3 (and V2)

December 2015

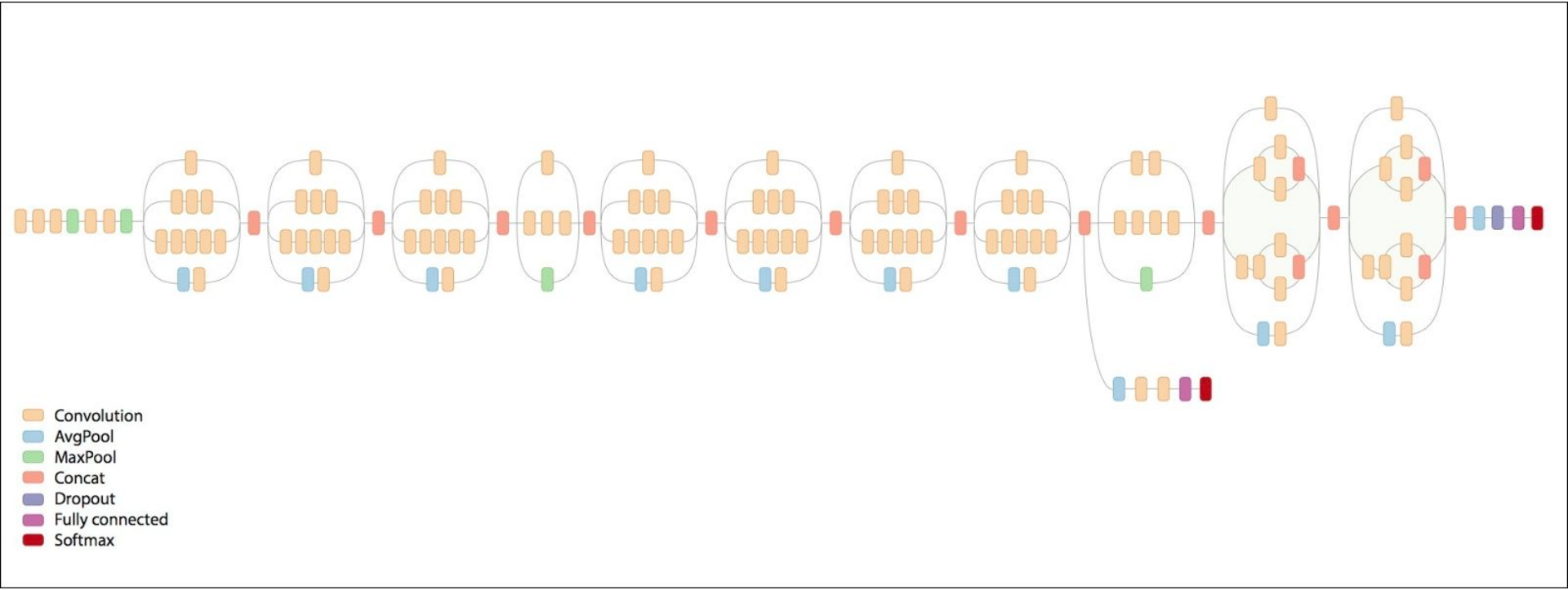
- Batchnorm added (incep v2)
- maximize information flow into the network, by carefully constructing networks that balance depth and width. Before each pooling, increase the feature maps.
- when depth is increased, the number of features, or width of the layer is also increased systematically
- use width increase at each layer to increase the combination of features.
- use only 3×3 convolution, when possible, given that filter of 5×5 and 7×7 can be decomposed with multiple 3×3

Inception V3

The Inception module shown uses convolutions with strides to decrease the size of the data

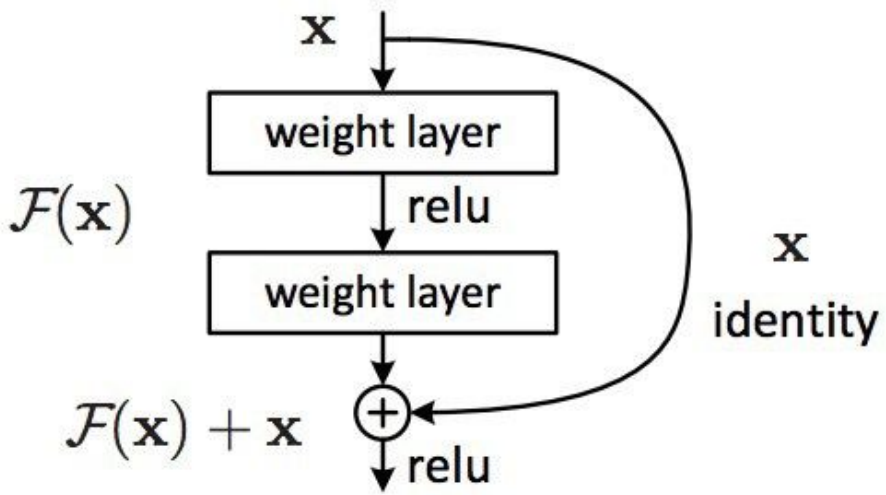


Complete Inception_v3 architecture

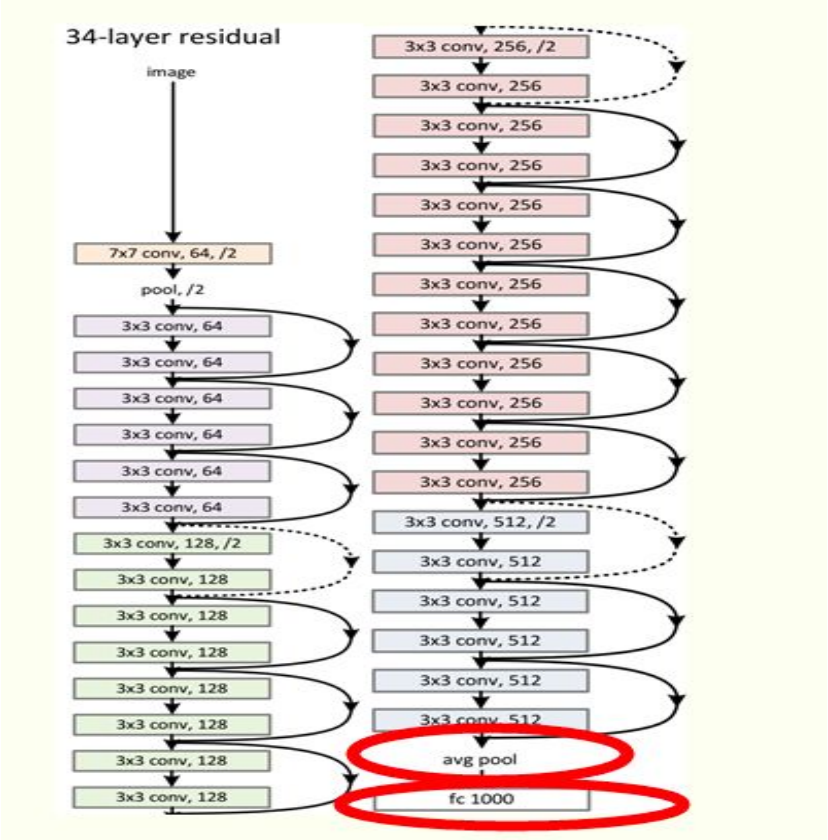


ResNet

- December 2015 (around Inception v3)
- Simple ideas:
  - Feed the output of two successive convolutional layers
  - Bypass the input to the next layers

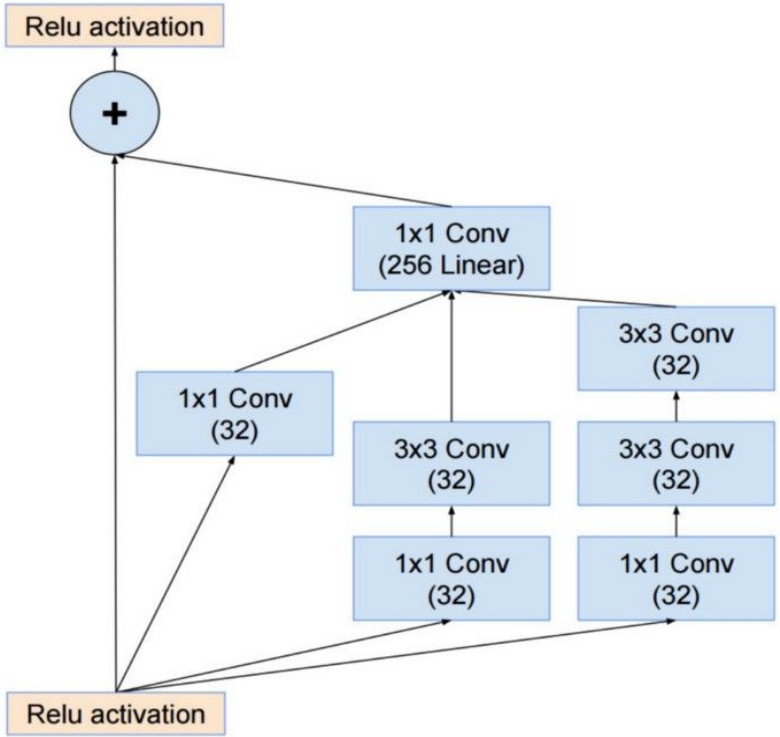


ResNet architecture



Inception v4 or Inception_Resnet_v2

- Added residual connections.



SqueezeNet

SqueezeNet can be 3 times faster and 500 times smaller than Alexnet with same accuracy.

- Using 1x1 filters to replace 3x3 filters.
- Using 1x1 filters as a bottleneck layer to reduce depth to reduce computation of the following 3x3 filters.
- Downsample late to keep a big feature map.

The building brick of SqueezeNet is called fire module, which contains two layers: a squeeze layer and an expand layer. A SqueezeNet stacks a bunch of fire modules and a few pooling layers.

Fire Modules

The squeeze layer and expand layer keep the same feature map size, while the former reduce the depth to a smaller number, the later increase it. The squeezing (bottoleneck layer) and expansion behavior is common in neural architectures. Another common pattern is increasing depth while reducing feature map size to get high level abstract features.

Fire Modules

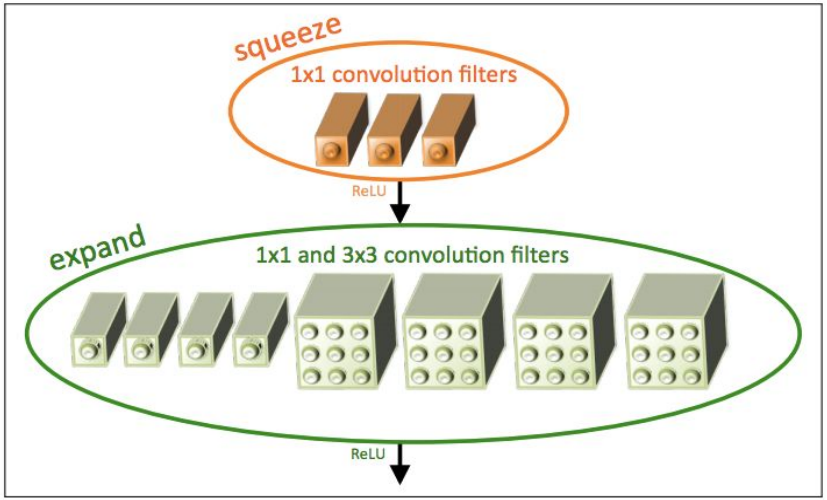


Figure 1: Microarchitectural view: Organization of convolution filters in the **Fire module**. In this example, $s_{1x1} = 3$, $e_{1x1} = 4$, and $e_{3x3} = 4$. We illustrate the convolution filters but not the activations.

Mobilenets

Core layers that MobileNet is built on which are depthwise separable filters (factorised filters).

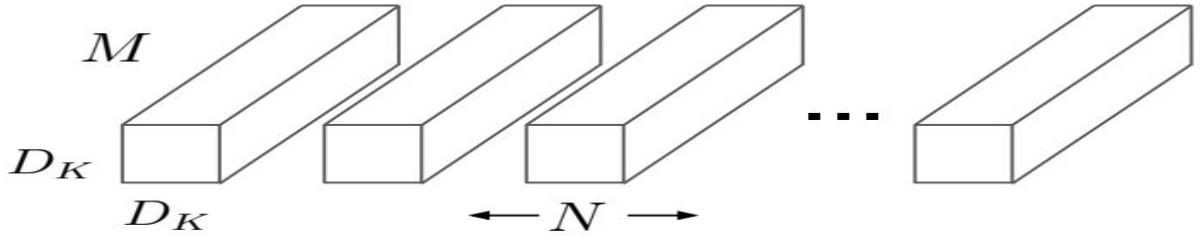
Depthwise separable convolution are made up of two layers: depthwise convolutions and pointwise convolutions.

Depthwise convolutions are used to apply a single filter per each input channel (input depth). Pointwise convolution, a simple 1x1
convolution, is then used to create a linear combination of the output of the depthwise layer. MobileNets use both batchnorm and ReLU nonlinearities for both layers.

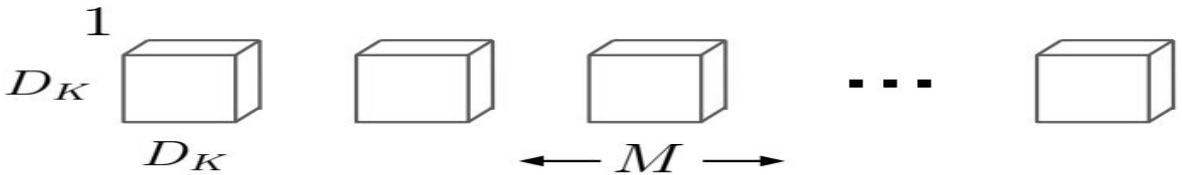
- Also uses width and resolution multipliers to save on computation
- Even more effective than Squeezenet

Depth wise convolutions

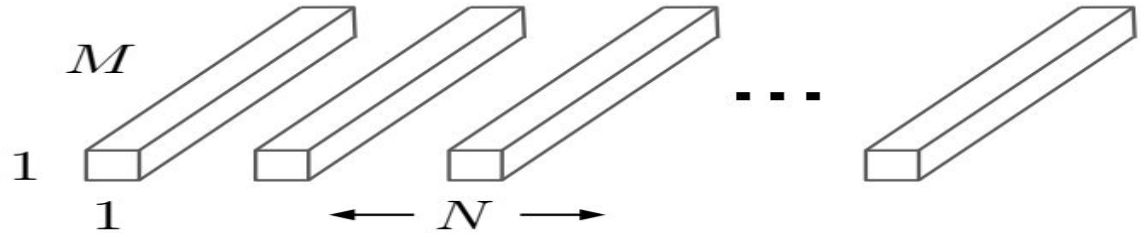
- form of factorized convolutions
- factorize a standard convolution into a depthwise convolution and a 1x1 convolution called a pointwise convolution



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

Table 4. Depthwise Separable vs Full Convolution MobileNet

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
Conv MobileNet	71.7%	4866	29.3
MobileNet	70.6%	569	4.2

Table 5. Narrow vs Shallow MobileNet

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
0.75 MobileNet	68.4%	325	2.6
Shallow MobileNet	65.3%	307	2.9

Table 6. MobileNet Width Multiplier

Width Multiplier	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
0.75 MobileNet-224	68.4%	325	2.6
0.5 MobileNet-224	63.7%	149	1.3
0.25 MobileNet-224	50.6%	41	0.5

Table 7. MobileNet Resolution

Resolution	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
1.0 MobileNet-192	69.1%	418	4.2
1.0 MobileNet-160	67.2%	290	4.2
1.0 MobileNet-128	64.4%	186	4.2