Convolutional Neural Networks

Atharva Parulekar

December 7, 2018

Differences?

Predicting heart disease using ECG

Categorizing News

Captioning Images

Predicting life expectancy using dna sequence.

Classifying images

Predicting medicine dosage using biological attributes

Predicting trajectory of an aircraft using sensor data.

Using economic factors to predict stock price.

Using wine characteristics to determine how good it is.

A first approach

The tools we have had so far : Expensive to learn!! Will not generalize well!! Does not exploit the order and local relations in the data!!

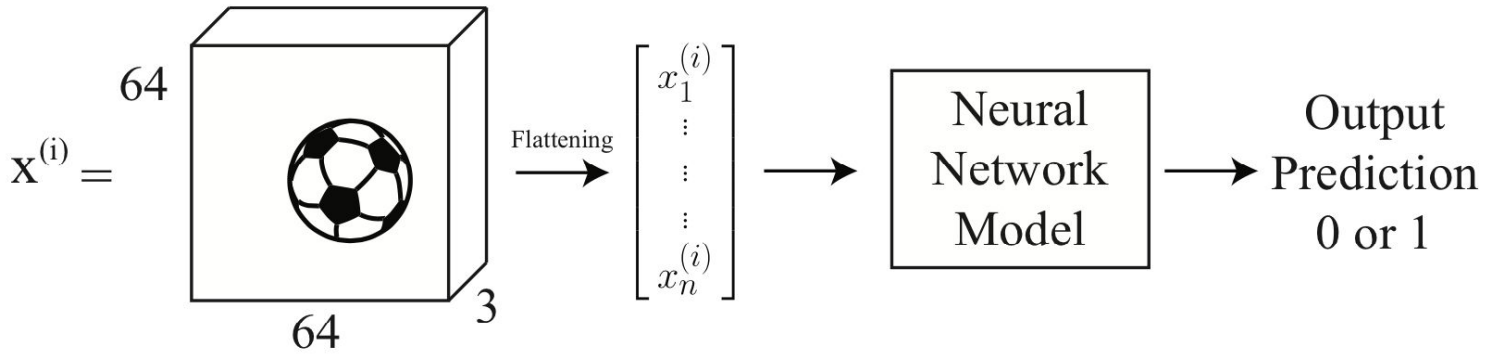
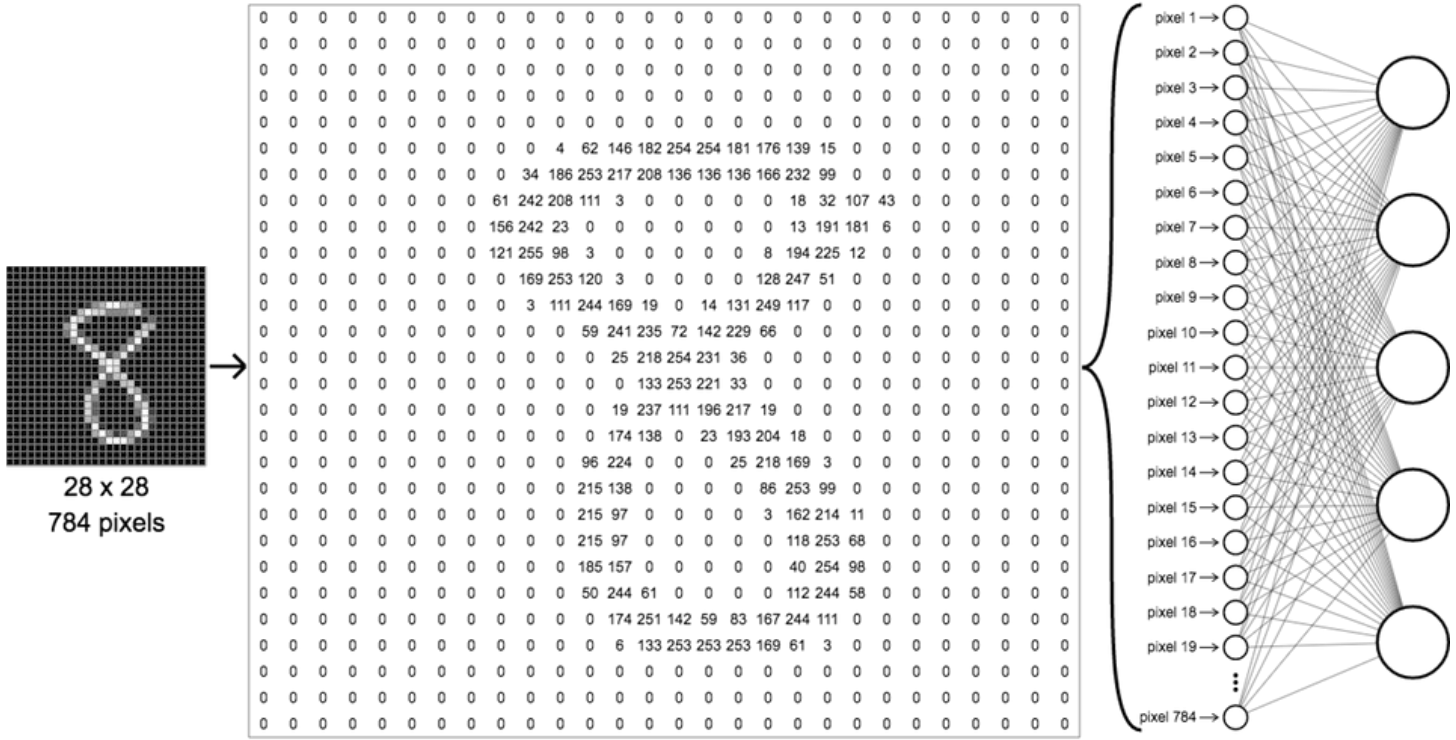


Image credit: Deep learning lecture notes

Let us look at images in detail

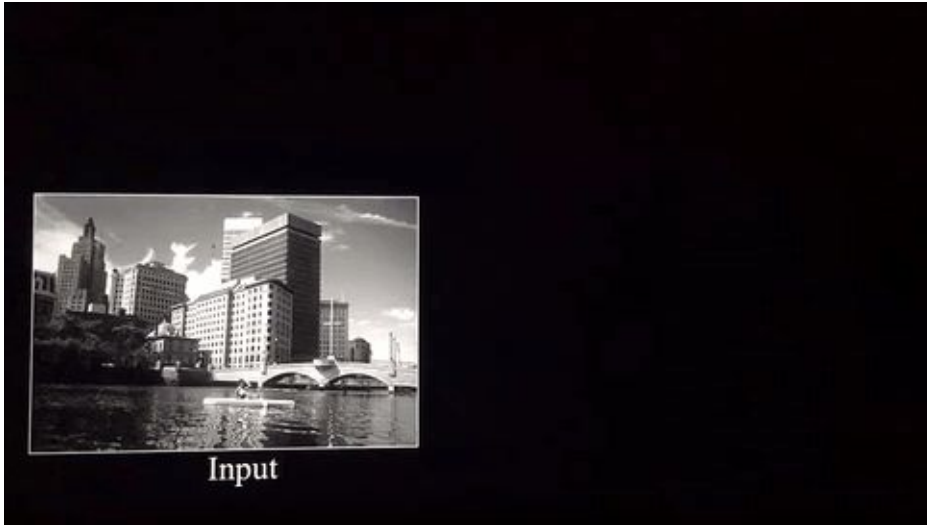


Filters

Why not extract features using filters?

Better, why not let the data dictate what filters to use?

Learnable filters!!



1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved Feature

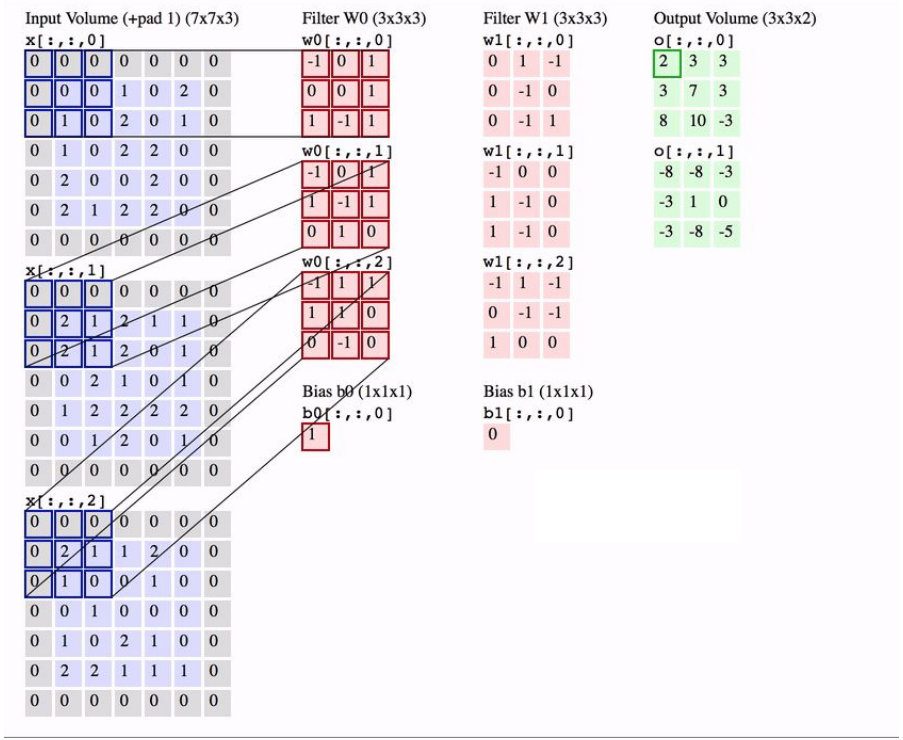
Filters on multiple channels

Images are generally RGB !!

How would a filter work on a image with RGB channels?

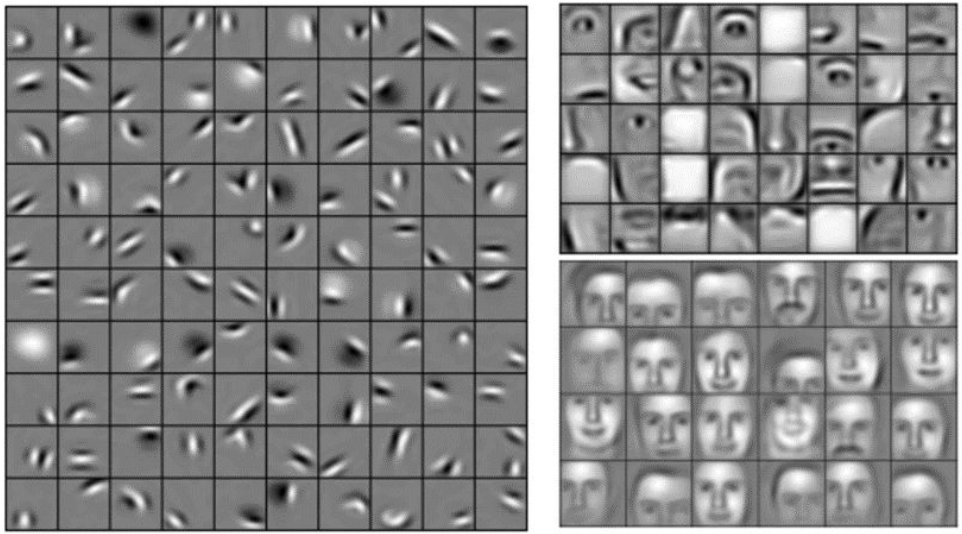
The filter should also have 3 channels.

Now the output has a channel for every filter we have used.

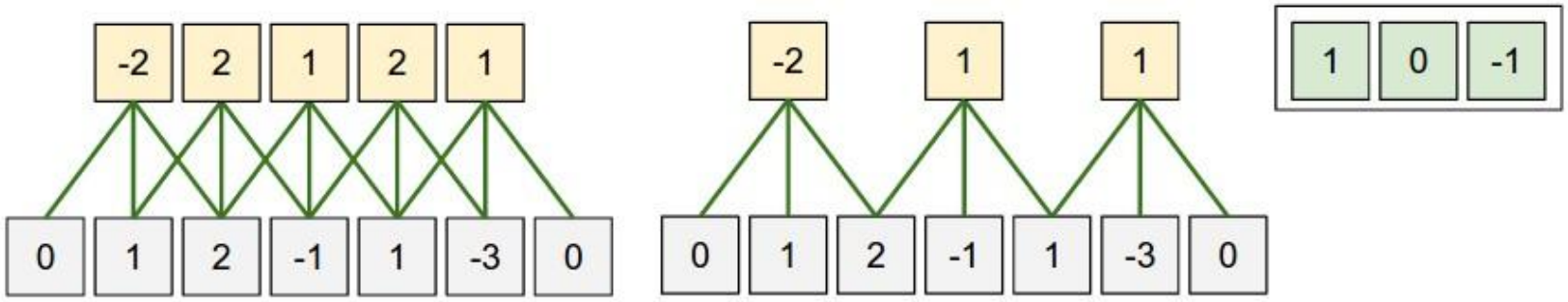


Hierarchical features

How deeper layers can learn deeper embeddings. How an eye is made up of multiple curves and a face is made up of two eyes.



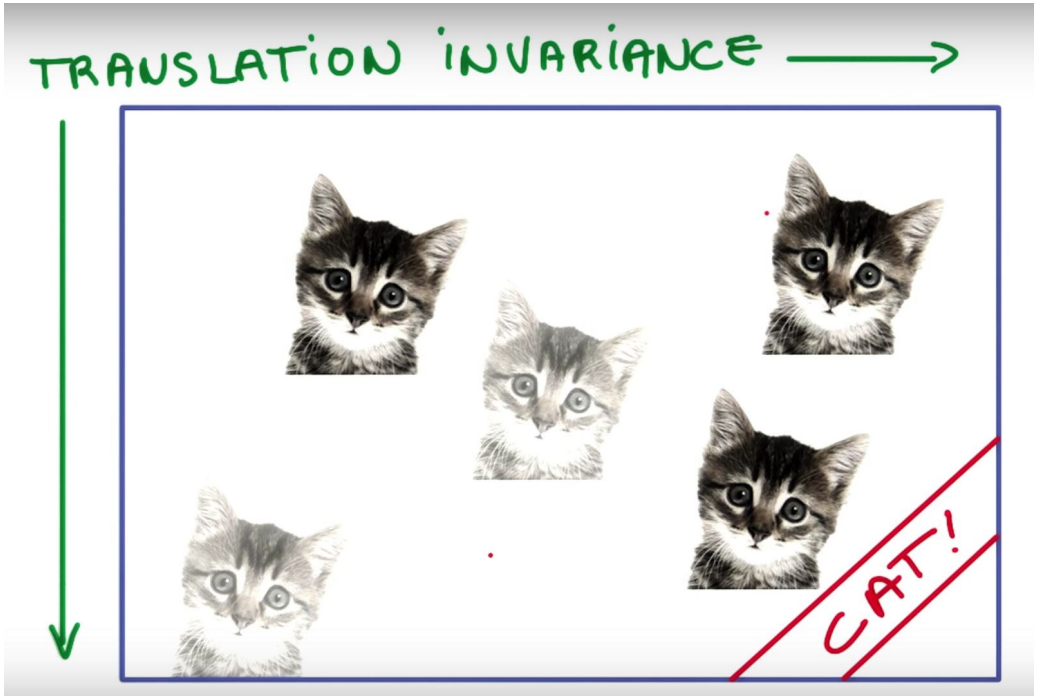
Parameter Sharing



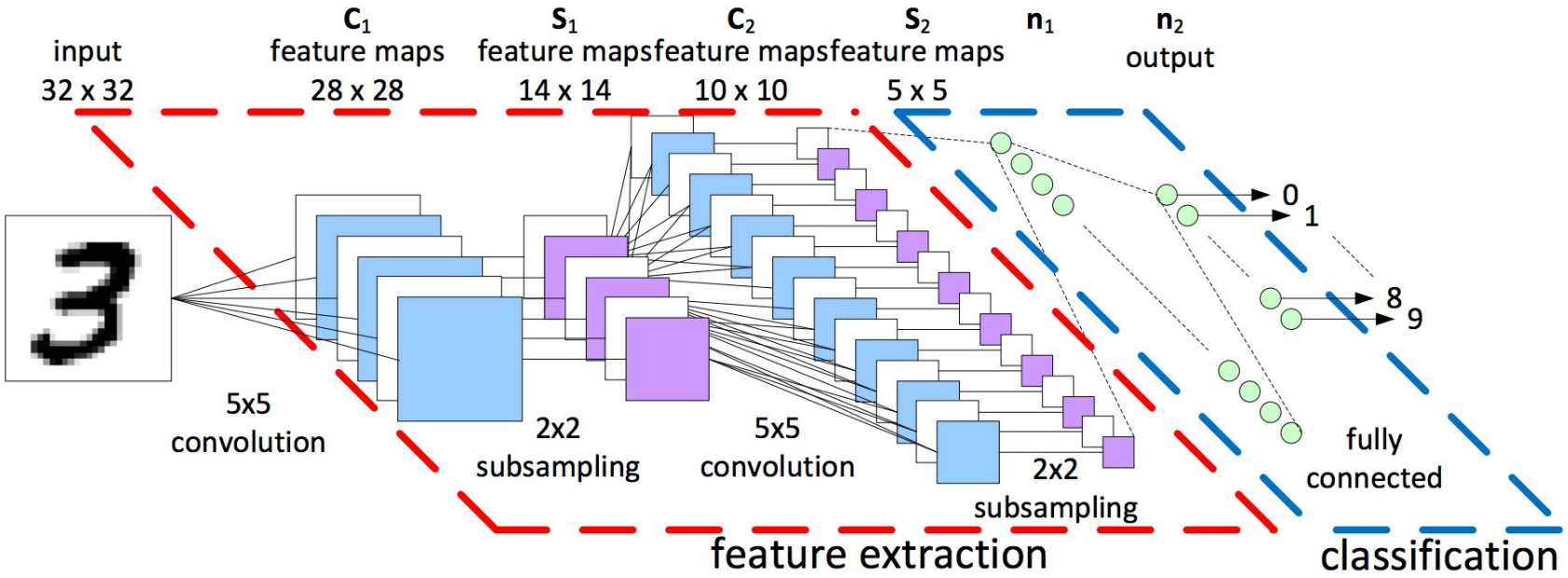
Lesser the parameters less computationally intensive the training. This is a win win as we are reusing parameters.

Translational invariance

Since we are training filters to detect cats and the moving these filters over the data, a differently positioned cat will also get detected by the same set of filters.



How do we use convolutions?



Let convolutions extract features and let normal cnn's decide on them.

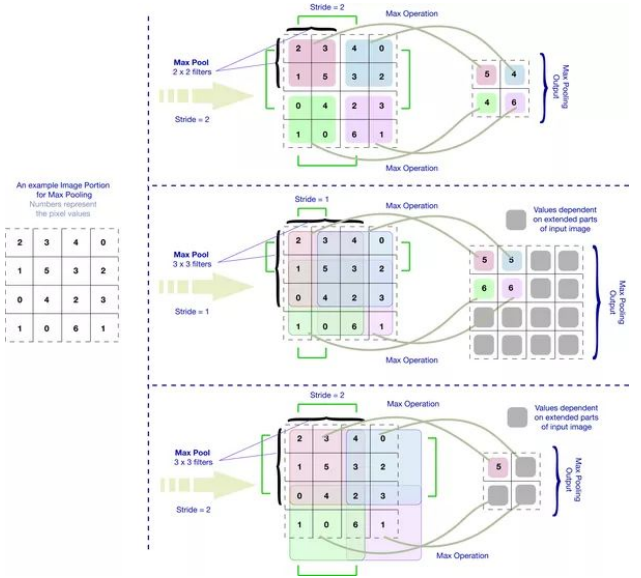
Image credit: LeCun et al. (1998)

Pooling

Problem: Convolutions with one stride keeps the image size approximately the same.

If the image is 100 x 100 then during the final fully connected part it will cause 100 x 100 x h x f weights where h is the number of hidden neurons in the fully connected layer, and f is the number of filters in the final conv layer.

Solution: The solution to this is something called pooling layers



The conv-pool duo

Convolution Layer:

- Input: W x H x D
- Filters: K, Size filter: F
- Stride: S
- Padding P

Output size:

- (W-F+2P)/S+1
- (H-F+2P)/S+1
- K

Pooling Layer:

- Input: W x H x D
- Size filter: F
- Stride: S

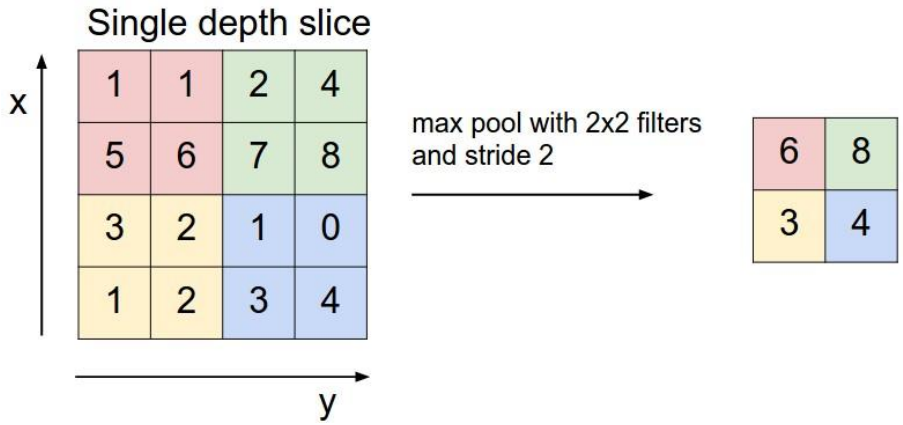
Output size:

- (W-F)/S+1
- (H-F)/S+1
- D

Why is pooling a problem?

Max/Min pooling causes loss of information.

Average pooling does not work in practice and is expensive to pass gradients through.



Convolution as a linear operation

In fact convolution is a giant matrix multiplication.

We can expand the 2 dimensional image into a vector and the conv operation into a matrix.

$$\begin{pmatrix} x1 & x2 & x3 \\ x4 & x5 & x6 \\ x7 & x8 & x9 \end{pmatrix} * \begin{pmatrix} k1 & k2 \\ k3 & k4 \end{pmatrix}$$

$$\begin{pmatrix} k1 & k2 & 0 & k3 & k4 & 0 & 0 & 0 & 0 \\ 0 & k1 & k2 & 0 & k3 & k4 & 0 & 0 & 0 \\ 0 & 0 & 0 & k1 & k2 & 0 & k3 & k4 & 0 \\ 0 & 0 & 0 & 0 & k1 & k2 & 0 & k3 & k4 \end{pmatrix} \cdot \begin{pmatrix} x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \\ x7 \\ x8 \\ x9 \end{pmatrix}$$

$$\begin{pmatrix} k1\,x1 + k2\,x2 + k3\,x4 + k4\,x5 \\ k1\,x2 + k2\,x3 + k3\,x5 + k4\,x6 \\ k1\,x4 + k2\,x5 + k3\,x7 + k4\,x8 \\ k1\,x5 + k2\,x6 + k3\,x8 + k4\,x9 \end{pmatrix}$$

Activation

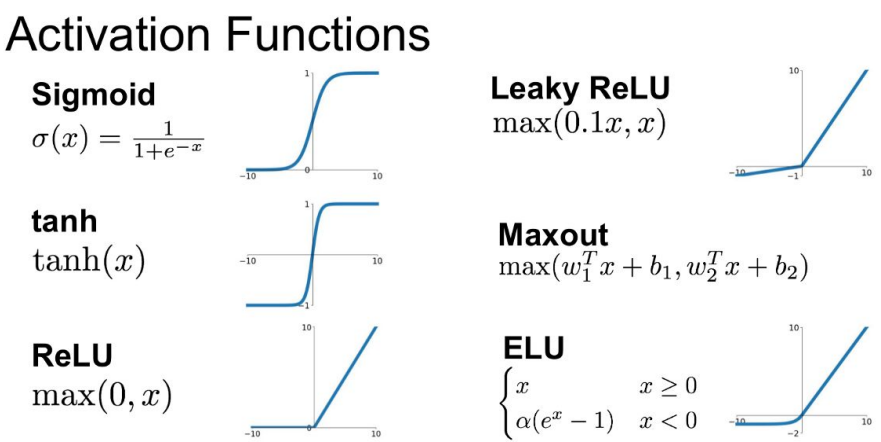
Since convolution is just a linear layer and so is pooling

We need to use an activation function to add non linearity

What kind of activation do we need?

Why RELu?

Vanishing gradients, derivative always 1 or 0.



Batch Normalization Layer

We insert this layer before fully connected layer

Mitigates a covariate shift

Makes the network less sensitive to hyperparameters like learning rate etc. Now you don't have do do a lot of searching.

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

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

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

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

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

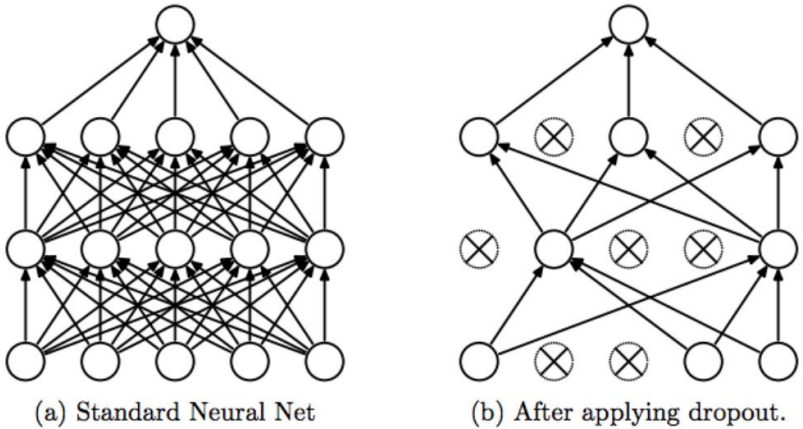
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Dropout

What does cutting off some network connections do?

Trains multiple smaller networks in an ensemble.

Acts like a really good regularizer



(a) Standard Neural Net

(b) After applying dropout.

Initialization

Can we initialize all neurons to zero?

If all the weights are same we will not be able to break symmetry of the network and all filters will end up learning the same thing.

Large numbers, might knock relu units out.

Relu units once knocked out and their output is zero, their gradient flow also becomes zero.

Thus, we need small random numbers as initialization.

Variance : 1/sqrt(n)

Mean: 0

How do we learn?

Backpropogation:

Convolution layer: easy since its a linear operation

Pooling layers: Only activated neurons allow gradient to flow

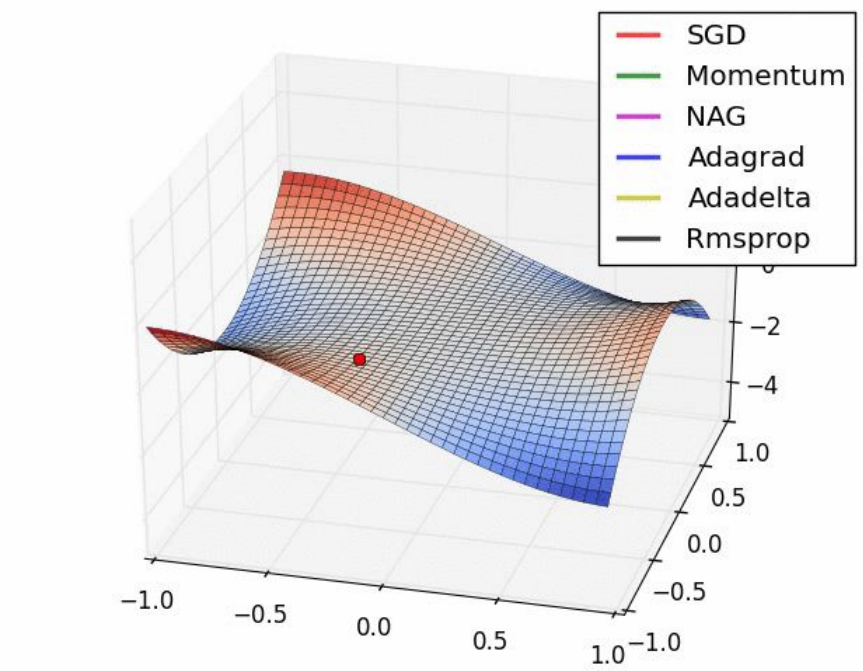
Normalization Layer: (Extremely tricky)

Dropout: only activated neurons send back gradients

How do we learn?

Optimizers:

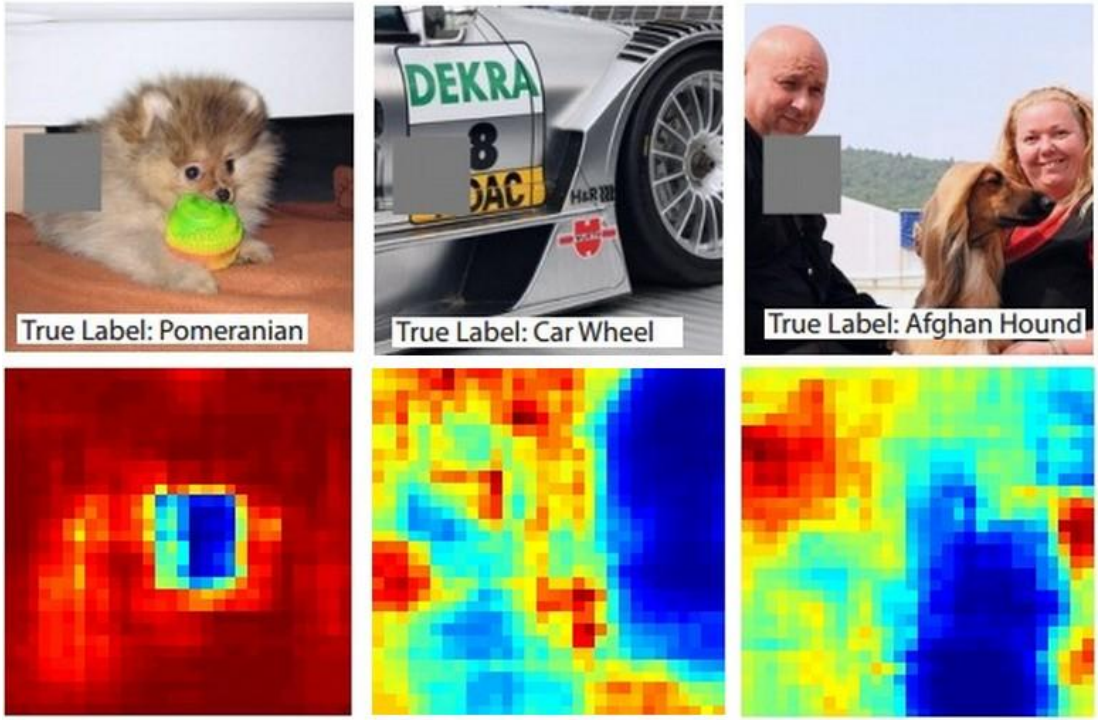
- Momentum: Gradient + Momentum
- Nestrov: Momentum + Gradients
- Adagrad: Normalize with sum of sq
- RMSprop: Normalize with moving avg of sum of squares
- ADAM: RMsprop + momentum



Are we actually learning?

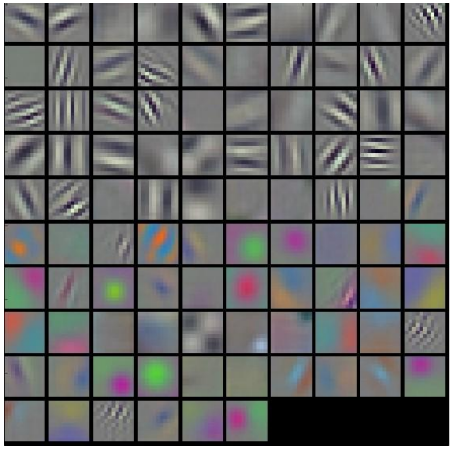
Lets play a game where we black out a square from a part of the image serially and then try to feed the image into a CNN.

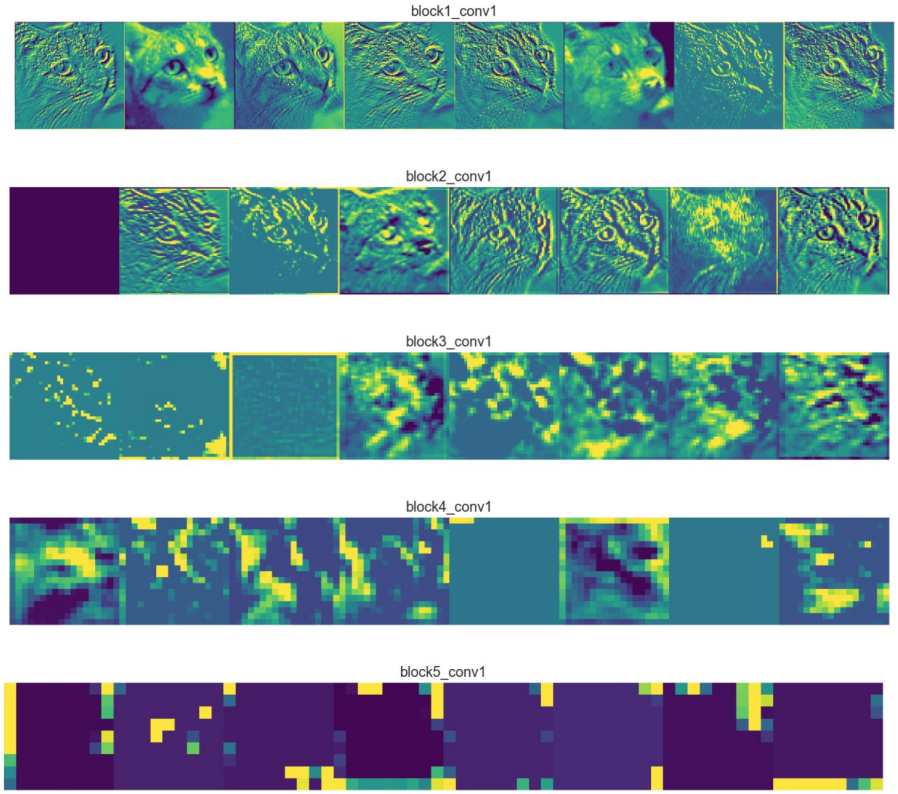
Then plot the probability of getting the correct label in a heatmap.



How does learning look like?

The first filters of an AlexNet CNN and the the conv layers visualized. Note the information loss happening as we go ahead



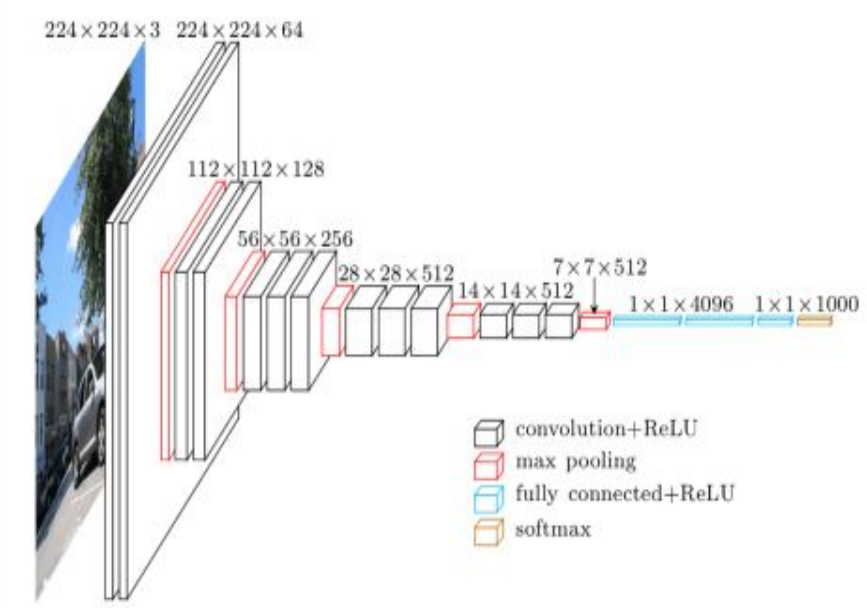


Is Deeper better?

Deeper networks seem to be more powerful but harder to train.

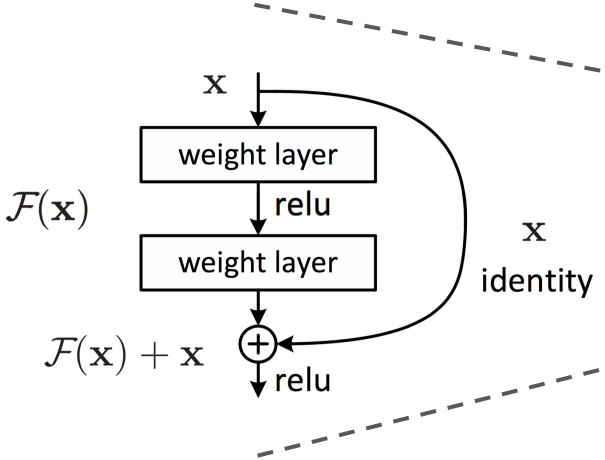
- Loss of information during forward prop
- Loss of gradient info during back prop

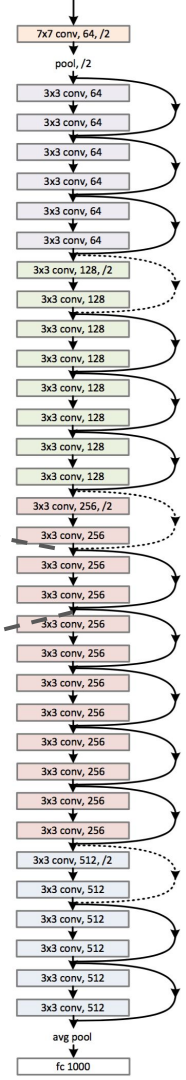
We now come up with a new trick to deal with it



Solution

Connect the layers, create a gradient highway or information highway.





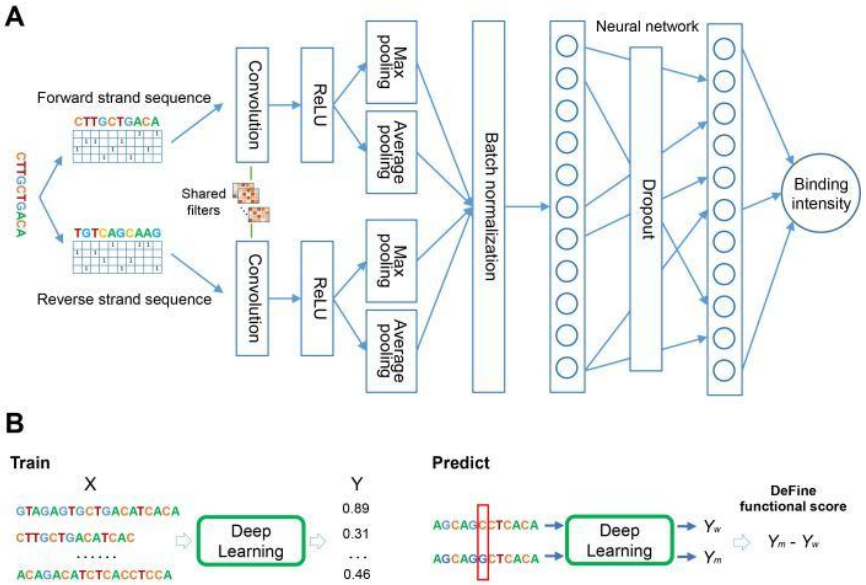
ResNet (2015)

Image credit: He et al. (2015)

1D architectures

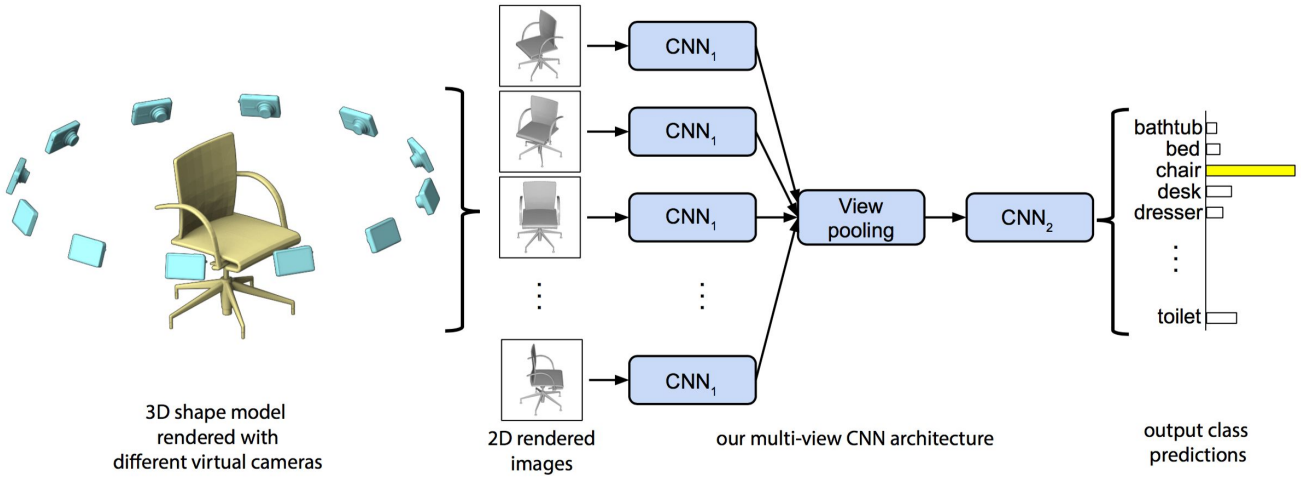
CNN’s work on any data where there is a local pattern.

We use 1 d convolutions on DNA sequences, text sequences and music notes.



2.5D architectures

Use multiple cameras to find out what's going on

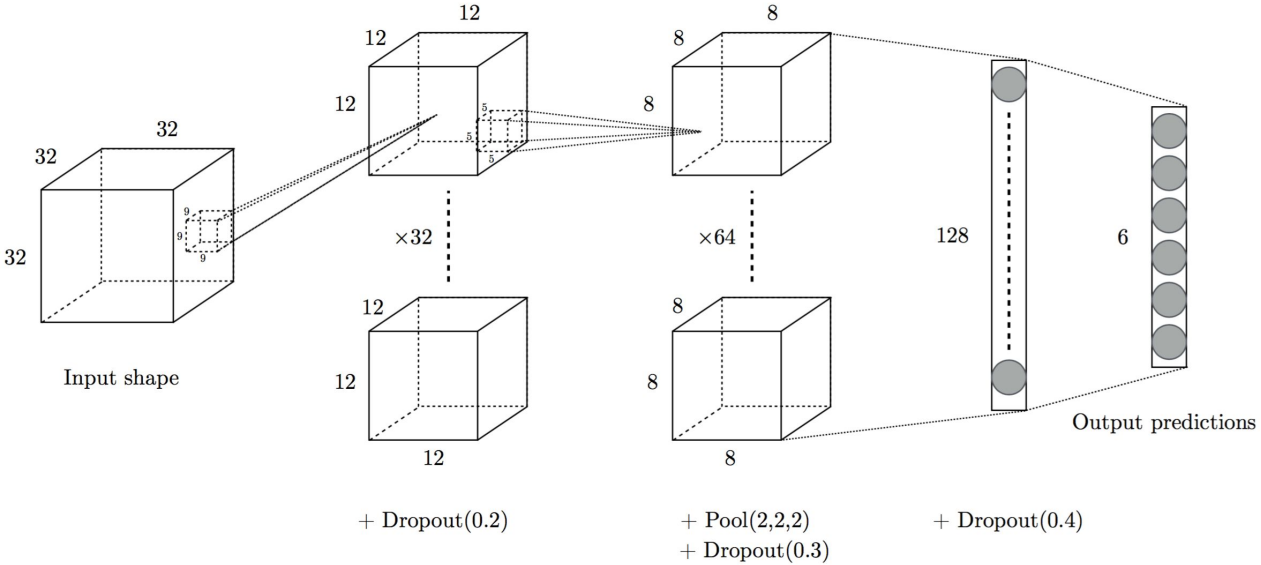


Multi-view CNN (2015)

Image credit: Su et al. (2015)

3D architectures

Use sensors like lidar to map what's going on. Use 3d convolutions.



Voxnet (2015)

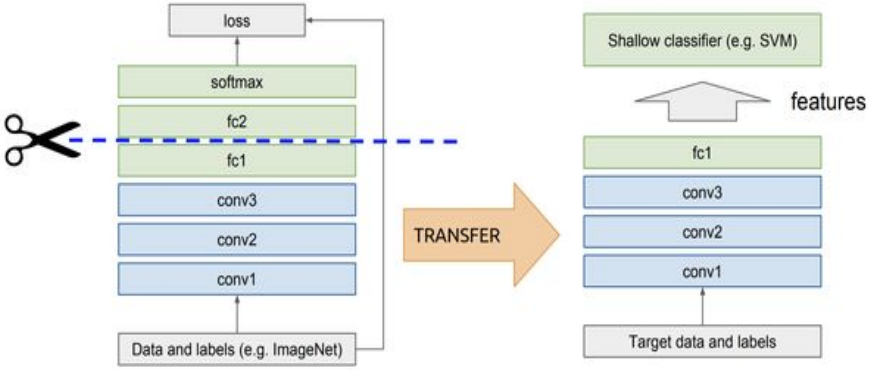
Image credit: Maturana et al. (left, 2015), Amidi et al. (right, 2017)

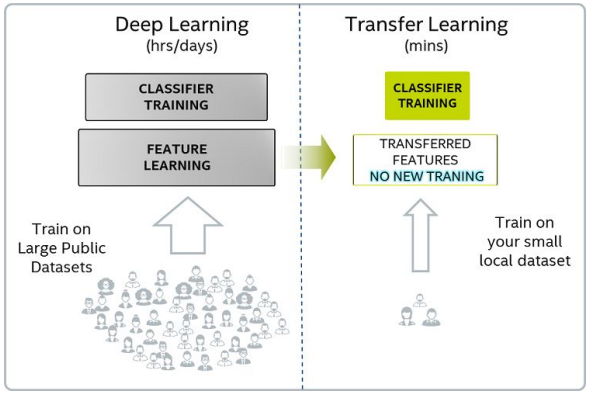
Learning is expensive

Training CNNs requires GPUs and access to special hardware.

In some cases, we can just use pre trained CNNs and modify or give them some additional training for our specific task.

CNNs can be used as fixed feature extractors by freezing the first few layers.

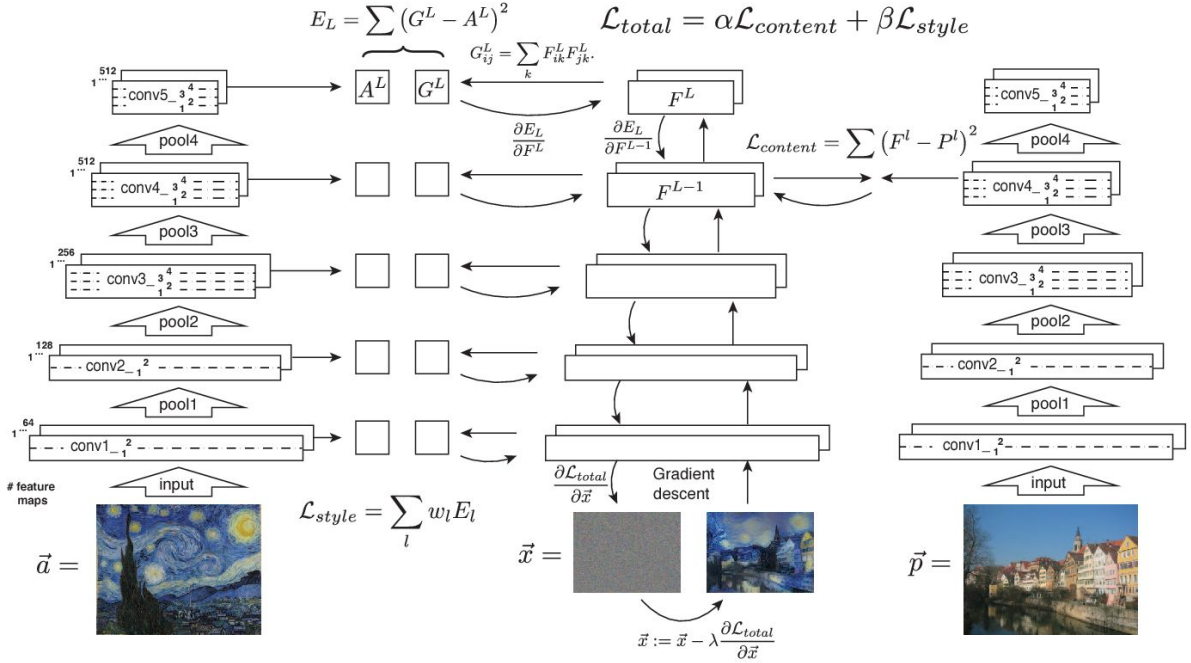




CNNs can be used for some cool stuff

Style transfer can be done by performing gradient descent on an empty image using a special loss function.

Neural Style Transfer



We have two losses here:

- Content loss: This is the L2 difference between both images
- Style loss: This is the difference between gram matrices of the two images.