Convolutional Neural Networks

Changyou Chen

Department of Computer Science and Engineering
University at Buffalo, SUNY
changyou@buffalo.edu

March 12, 2019

Basic Gradients

The gradient of the loss function with respect to the weight $W_{cij}^{(1)}$ is given by

Basic Gradients

The gradient of the loss function with respect to the weight $W_{cij}^{(1)}$ is given by

$$\frac{\partial \mathcal{L}}{\partial W_{cij}^{(1)}} = \sum_{k,m,n} \frac{\partial \mathcal{L}}{\partial H_{kmn}^{(1)}} \frac{\partial H_{kmn}^{(1)}}{\partial W_{cij}^{(1)}}$$

Basic Gradients

The gradient of the loss function with respect to the weight $W_{cij}^{(1)}$ is given by

$$\frac{\partial \mathcal{L}}{\partial W_{cij}^{(1)}} = \sum_{k,m,n} \frac{\partial \mathcal{L}}{\partial H_{kmn}^{(1)}} \frac{\partial H_{kmn}^{(1)}}{\partial W_{cij}^{(1)}}$$

$$= \sum_{k,m,n} \frac{\partial \mathcal{L}}{\partial H_{kmn}^{(1)}} \frac{\partial \sum_{p,q} W_{pqm}^{(1)} V_{k+p,q,n}}{\partial W_{cij}^{(1)}}$$

Basic Gradients

The gradient of the loss function with respect to the weight $W_{cij}^{(1)}$ is given by

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial W_{cij}^{(1)}} &= \sum_{k,m,n} \frac{\partial \mathcal{L}}{\partial H_{kmn}^{(1)}} \frac{\partial H_{kmn}^{(1)}}{\partial W_{cij}^{(1)}} \\
&= \sum_{k,m,n} \frac{\partial \mathcal{L}}{\partial H_{kmn}^{(1)}} \frac{\partial \sum_{p,q} W_{pqm}^{(1)} V_{k+p,q,n}}{\partial W_{cij}^{(1)}} \\
&= \sum_{k,n} \frac{\partial \mathcal{L}}{\partial H_{kjn}^{(1)}} \frac{\partial \sum_{p,q} W_{pqj}^{(1)} V_{k+p,q,n}}{\partial W_{cij}^{(1)}}
\end{aligned}$$

Basic Gradients

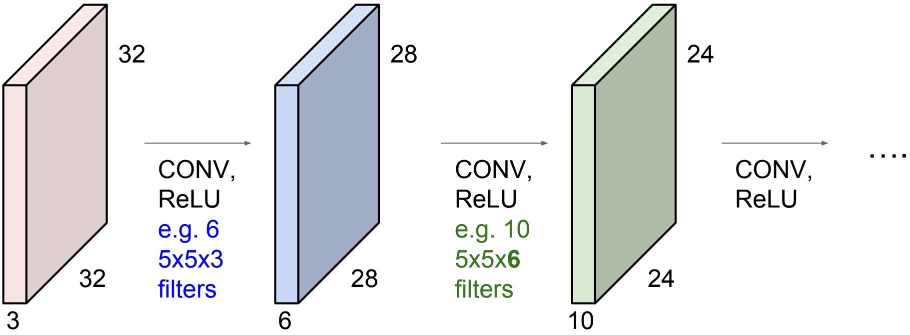
The gradient of the loss function with respect to the weight $W_{cij}^{(1)}$ is given by

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial W_{cij}^{(1)}} &= \sum_{k,m,n} \frac{\partial \mathcal{L}}{\partial H_{kmn}^{(1)}} \frac{\partial H_{kmn}^{(1)}}{\partial W_{cij}^{(1)}} \\
&= \sum_{k,m,n} \frac{\partial \mathcal{L}}{\partial H_{kmn}^{(1)}} \frac{\partial \sum_{p,q} W_{pqm}^{(1)} V_{k+p,q,n}}{\partial W_{cij}^{(1)}} \\
&= \sum_{k,n} \frac{\partial \mathcal{L}}{\partial H_{kjn}^{(1)}} \frac{\partial \sum_{p,q} W_{pqj}^{(1)} V_{k+p,q,n}}{\partial W_{cij}^{(1)}} \\
&= \sum_{k,n} \frac{\partial \mathcal{L}}{\partial H_{kjn}^{(1)}} V_{k+c,i,n}
\end{aligned}
$$

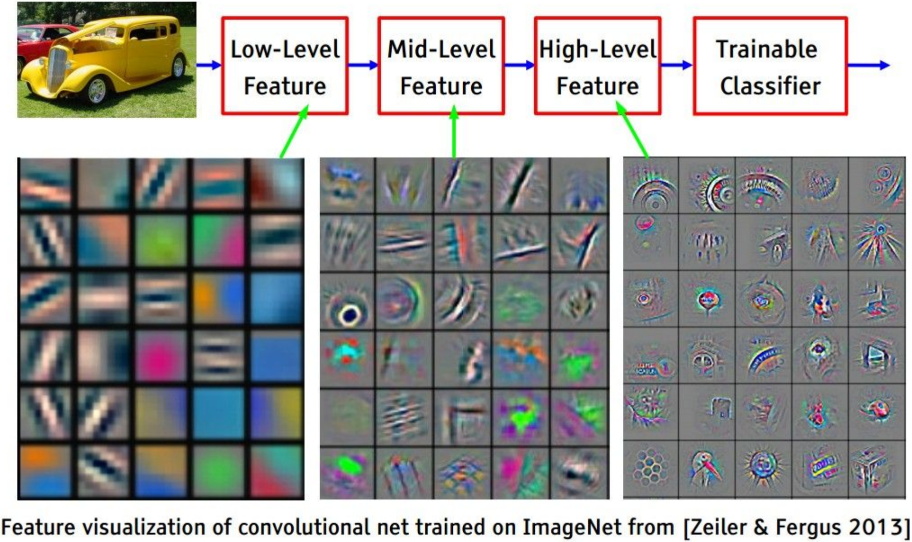
Convolutional Neural Networks (ConvNet)

Convolutional Neural Networks

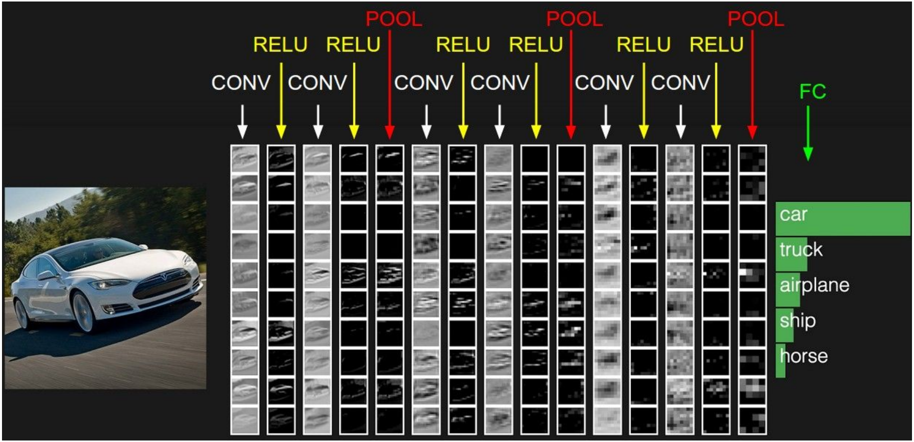
ConvNet is a sequence of Convolutional Layers, interspersed with activation functions



Features from ConvNet

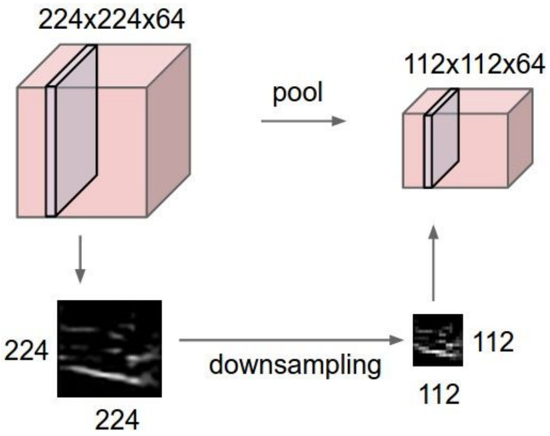


Two More Layers to Go: POOL/FC

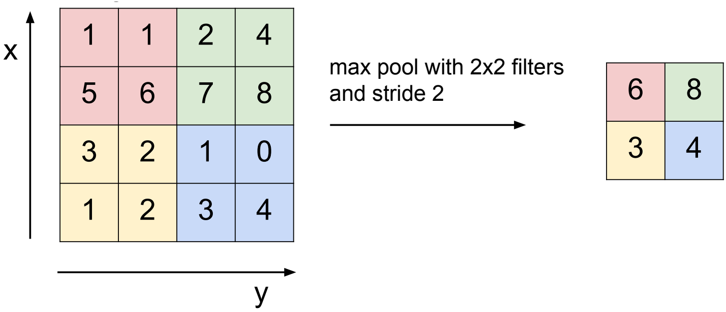


Pooling Layer

1. Makes the representations smaller and more manageable.
2. Operates over each activation map independently.



Max Pooling



Common settings:

- Filter size: 2×2 or 3×3
- Stride: 2

Max pooling is a non-linear operator.

Max Pooling

Q: What is the gradient of a max pooling operation?

Case Study: CNN for MNIST

Padding in Tensorflow

- Input width = 13; Filter width = 6; Stride = 5.
- "VALID" only ever drops the right-most columns (or bottom-most rows).
- "SAME" tries to pad evenly left and right, but if the amount of columns to be added is odd, it will add the extra column to the right.

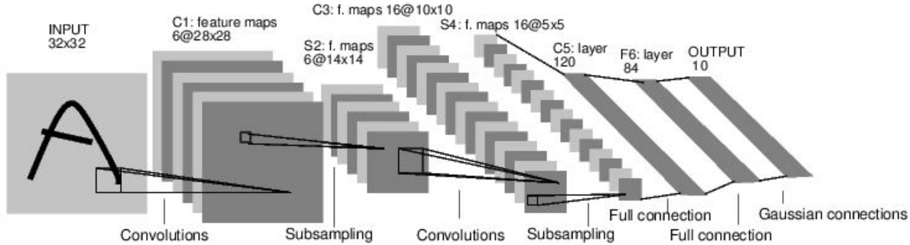
`"VALID"` = without padding:

```
   inputs:         1  2  3  4  5  6  7  8  9  10 11 (12 13)
                  |________________|                dropped
                                 |_________________|
```

`"SAME"` = with zero padding:

```
               pad|                                      |pad
   inputs:      0 |1  2  3  4  5  6  7  8  9  10 11 12 13|0  0
               |________________|
                              |_________________|
                                             |________________|
```

Case Study: LeNet-5 [LeCun *et al.*, 1998]



Conv filters were 5x5, applied at stride 1
Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-CONV-FC]

Implementation

https://github.com/sujaybabruwad/LeNet-in-Tensorflow/blob/master/LeNet-Lab.ipynb

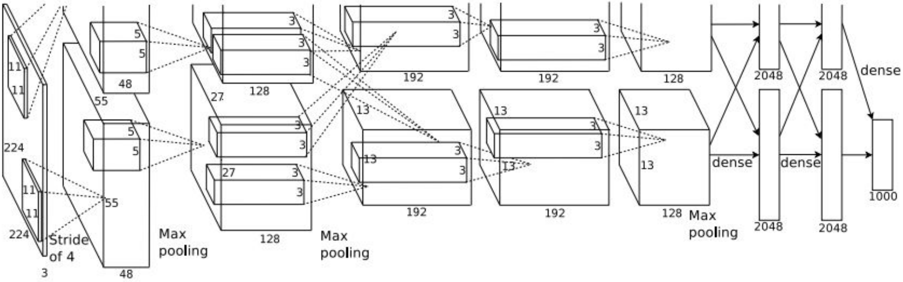
ImageNet

- About 14M images, 1000 classes.

http://www.image-net.org/

Case Study: AlexNet [Krizhevsky *et al.*, 2012]

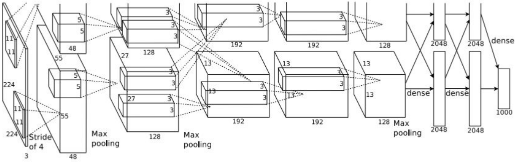
- Input: $227 \times 227 \times 3$ images.
- First layer (CONV1): 96 11×11 filters applied at stride 4.



- Q: What are the output volume and number of parameters in each layer?

Case Study: AlexNet

[Krizhevsky et al. 2012]



Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

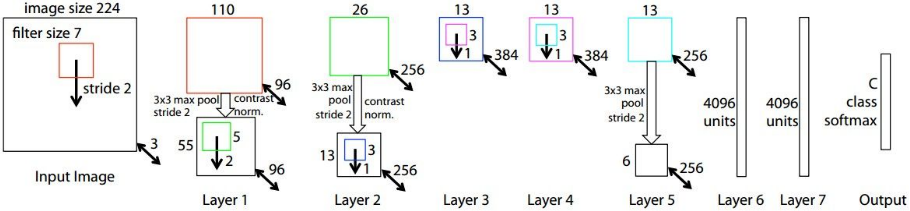
Details/Retrospectives:
- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

Implementation

https://github.com/vigneshthakkar/Deep-Nets/blob/master/AlexNet.py

Case Study: ZFNet [Zeiler and Fergus, 2013]

- AlexNet but:
  - CONV1: change from (11 $\times$ 11 stride 4) to (7 $\times$ 7 stride 2)
  - CONV3, 4, 5: instead of 384, 384, 256 filters use 512, 1024, 512
- ImageNet top 5 error: 15.4% -> 14.8%.



Implementation

https://github.com/vigneshthakkar/Deep-Nets/blob/master/ZFNet.py

Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Only 3x3 CONV stride 1, pad 1
and 2x2 MAX POOL stride 2

best model

11.2% top 5 error in ILSVRC 2013
->
7.3% top 5 error

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					

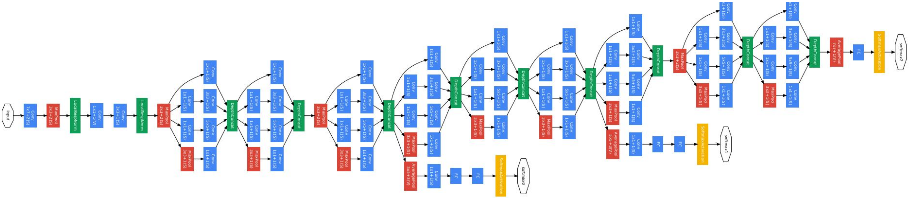
Table 2: **Number of parameters** (in millions).

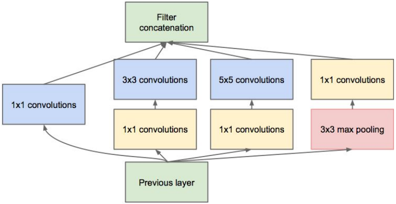
Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

Implementation

https://github.com/vigneshthakkar/Deep-Nets/blob/master/VGGNet.py

Case Study: GoogLeNet [Szegedy *et al.*, 2014]





Inception module

ILSVRC 2014 winner (6.7% top 5 error)

Case Study: GoogLeNet [Szegedy *et al.*, 2014]

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

Fun features:

- Only 5 million params! (Removes FC layers completely)

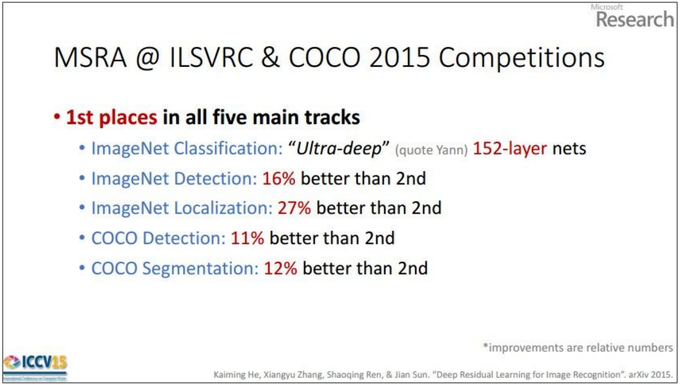
Compared to AlexNet:
- 12X less params
- 2x more compute
- 6.67% (vs. 16.4%)

Implementation

https://github.com/vigneshthakkar/Deep-Nets/blob/master/GoogLeNet.py

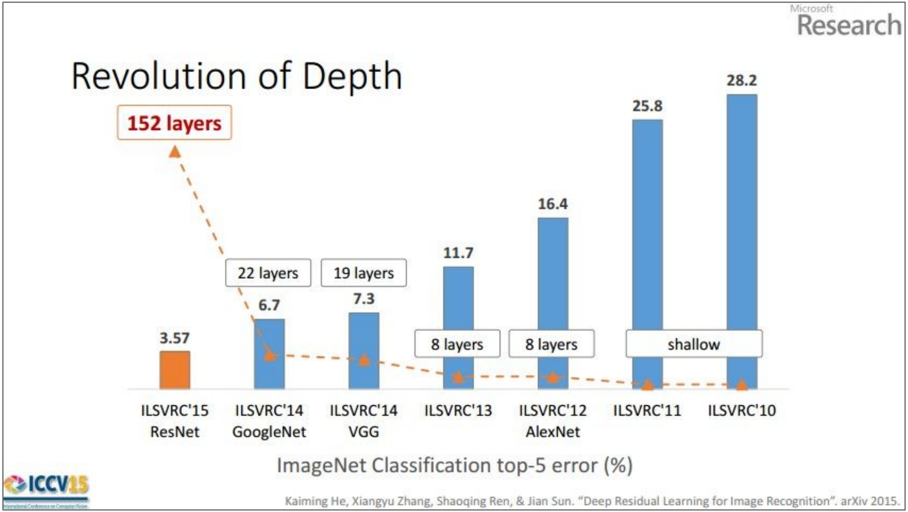
Case Study: ResNet [He *et al.*, 2015]

ILSVRC 2015 winner (3.6% top 5 error)

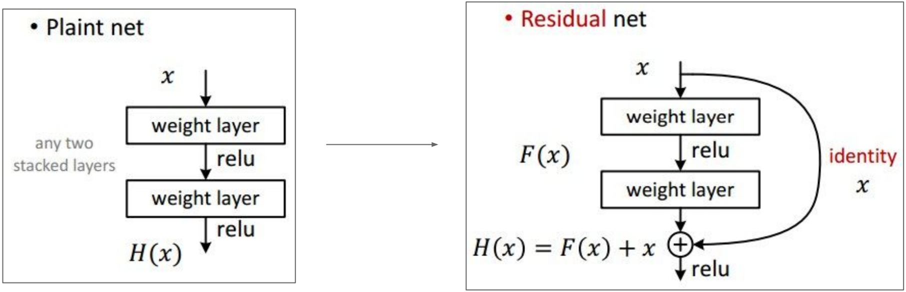


Slide from Kaiming He's recent presentation https://www.youtube.com/watch?v=1PGLj-uKT1w

Case Study: ResNet [He *et al.*, 2015]



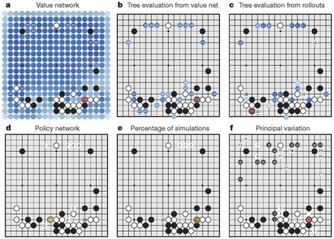
Case Study: ResNet [He *et al.*, 2015]



Implementation

https://github.com/vigneshthakkar/Deep-Nets/blob/master/ResNet.py

Case Study Bonus: DeepMind's AlphaGo



Case Study Bonus: DeepMind's AlphaGo

> The input to the policy network is a 19 × 19 × 48 image stack consisting of 48 feature planes. The first hidden layer zero pads the input into a 23 × 23 image, then convolves *k* filters of kernel size 5 × 5 with stride 1 with the input image and applies a rectifier nonlinearity. Each of the subsequent hidden layers 2 to 12 zero pads the respective previous hidden layer into a 21 × 21 image, then convolves *k* filters of kernel size 3 × 3 with stride 1, again followed by a rectifier nonlinearity. The final layer convolves 1 filter of kernel size 1 × 1 with stride 1, with a different bias for each position, and applies a softmax function. The match version of AlphaGo used *k* = 192 filters; Fig. 2b and Extended Data Table 3 additionally show the results of training with *k* = 128, 256 and 384 filters.

policy network:
[19x19x48] Input
CONV1: 192 5x5 filters , stride 1, pad 2 => [19x19x192]
CONV2..12: 192 3x3 filters, stride 1, pad 1 => [19x19x192]
CONV: 1 1x1 filter, stride 1, pad 0 => [19x19] *(probability map of promising moves)*

Summary

1. ConvNets stack CONV,POOL,FC layers.
2. Trend towards smaller filters and deeper architectures.
3. Trend towards getting rid of POOL/FC layers (just CONV and action layers).
4. Typical architectures look like:

   [(CONV-ReLU) $*N$ **-Pool?]** $*M$ **-(FC-ReLU)** $*K$ **, SOFTMAX**

   - N is usually up to ~ 5, M is large, $0 \leq K \leq 2$.
   - recent advances such as ResNet/GoogLeNet challenge this paradigm.