Object Detection, CNN based Networks

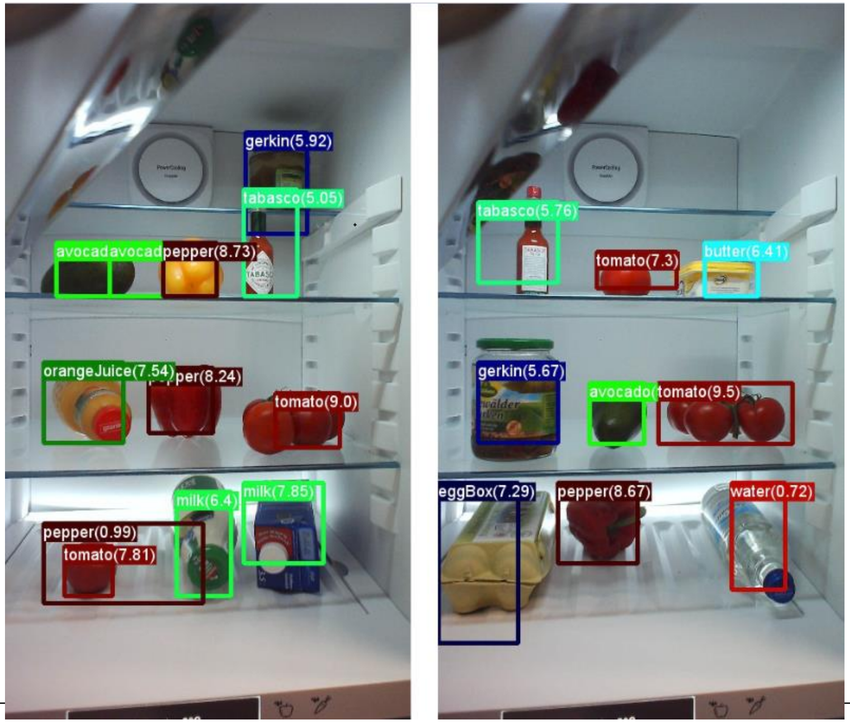
Ganesh Ramakrishnan, Dept of CSE, IITB

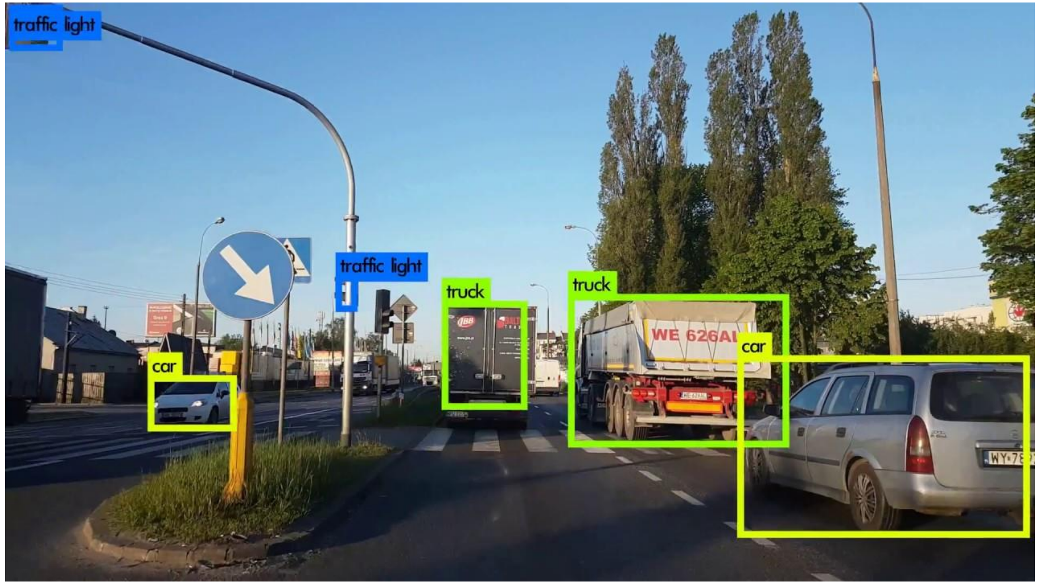
Acknowledgement: Anurag Mundhada, Arjun Jain

Object Detection

Practically:

The task of assigning a label and a bounding box to all objects in an image





Data: The oil driving object detection research

Object Detection datasets

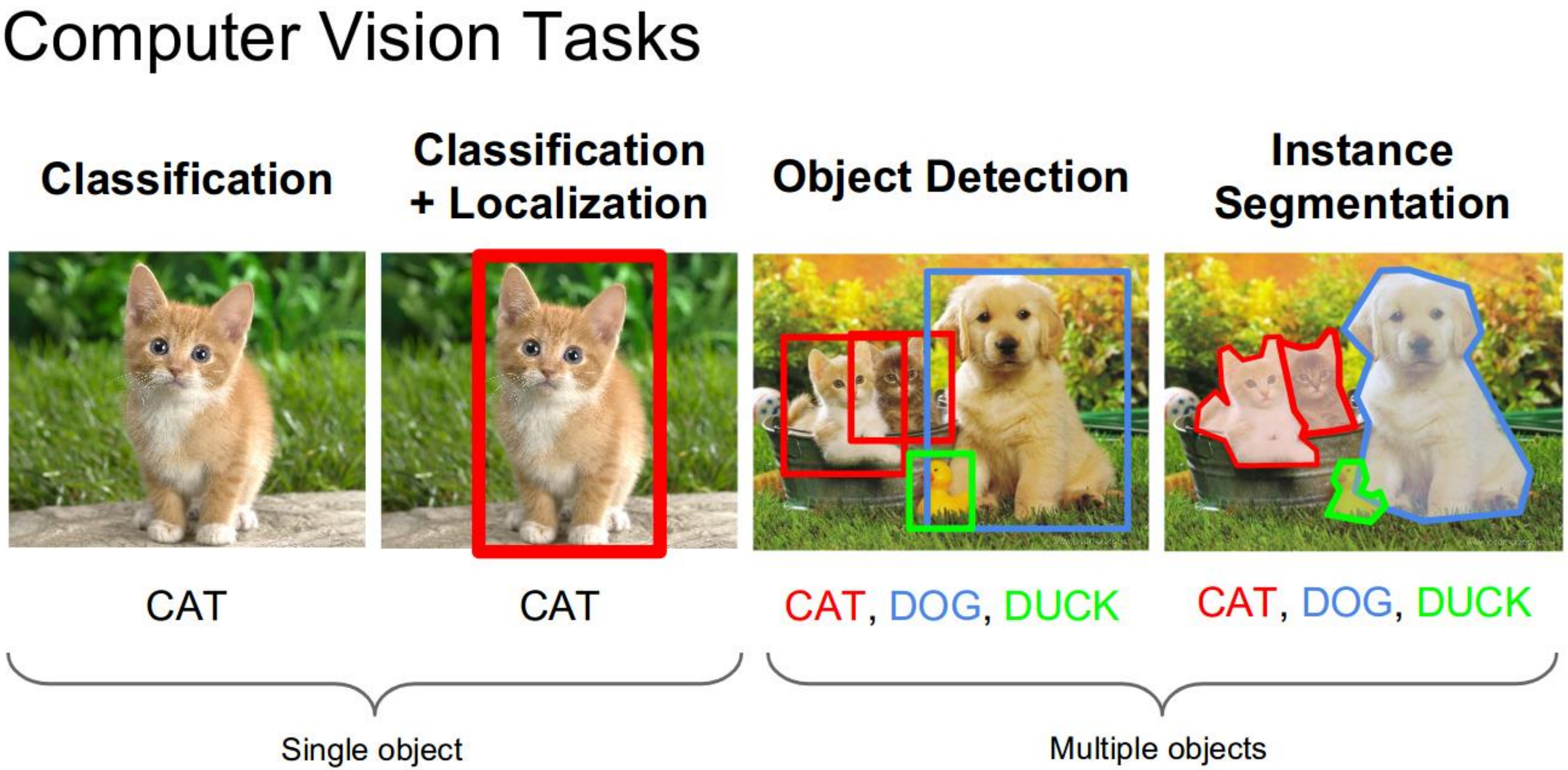
200,000 images and 80 object categories

Detection: 500,000 images and 200 object categories

Google Open Images v4

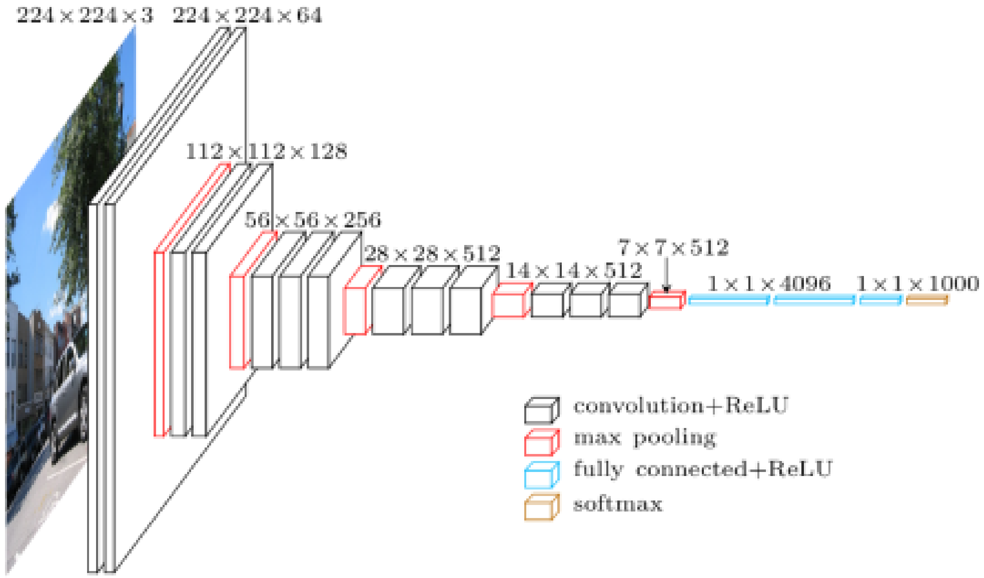
15,440,132 boxes on 600 categories

Object Detection



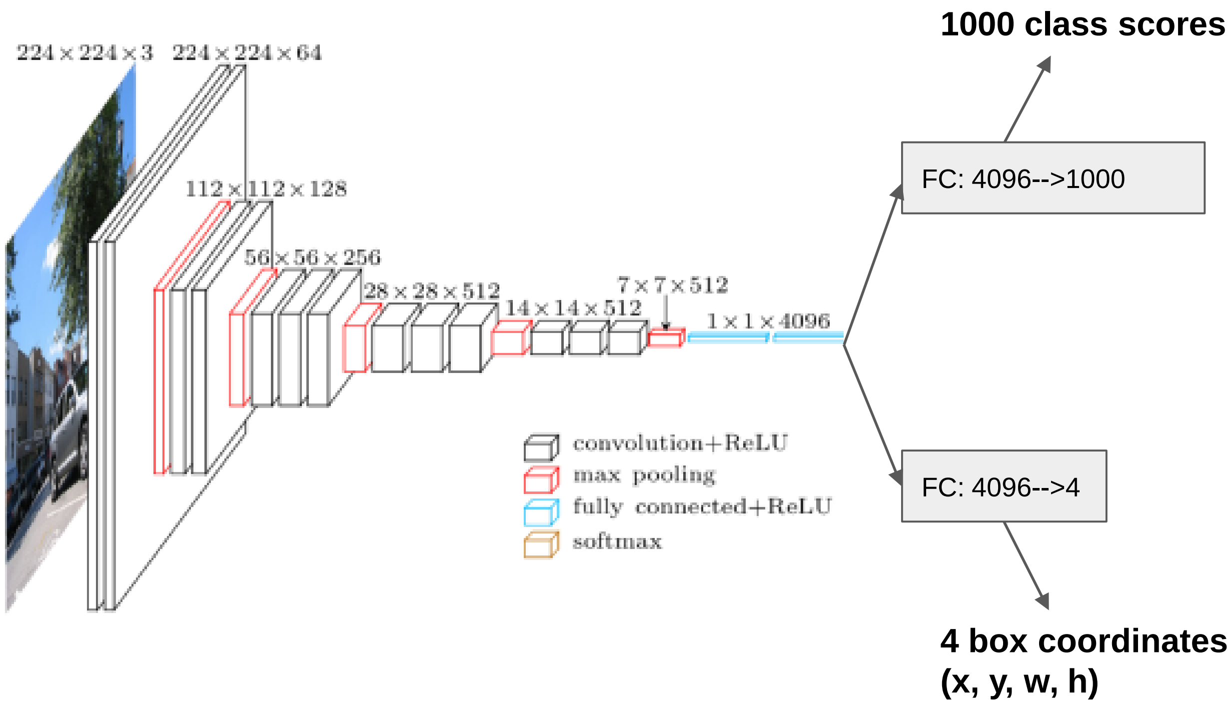
CS231 Lecture Notes, Stanford University

Classification



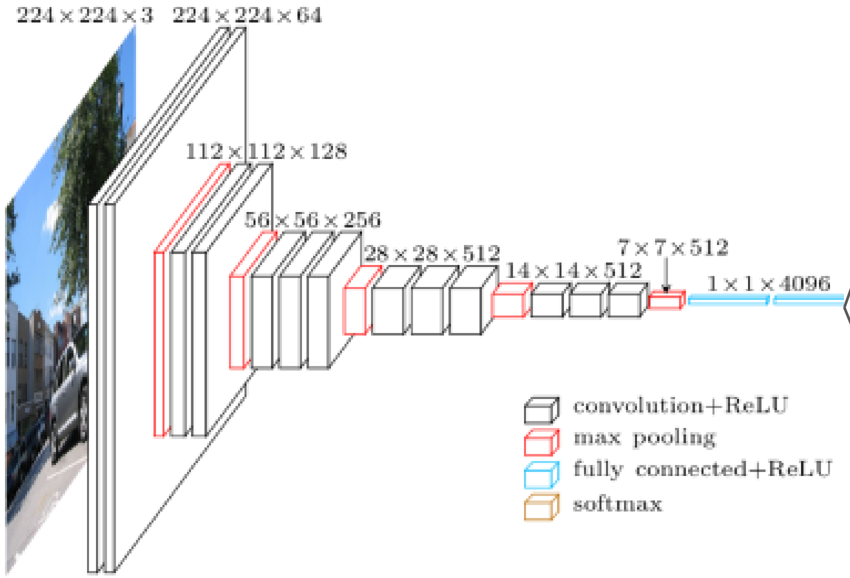
1000 class scores

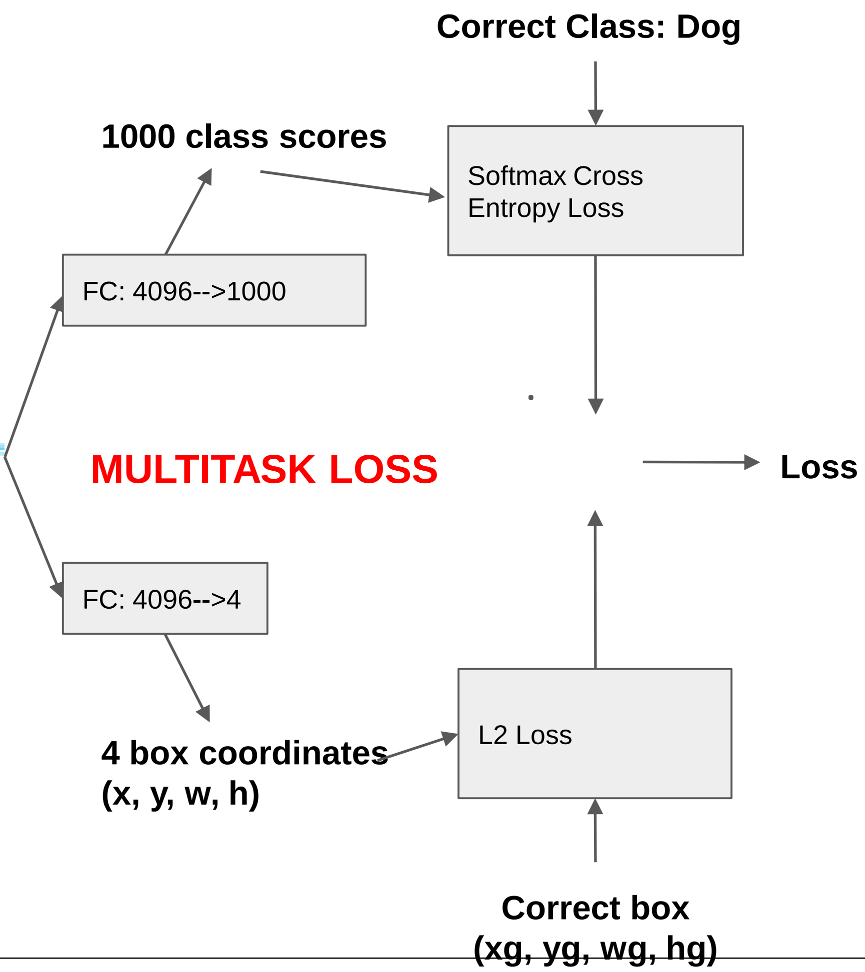
Classification + Localization



Treat localization as a regression problem

Classification + Localization

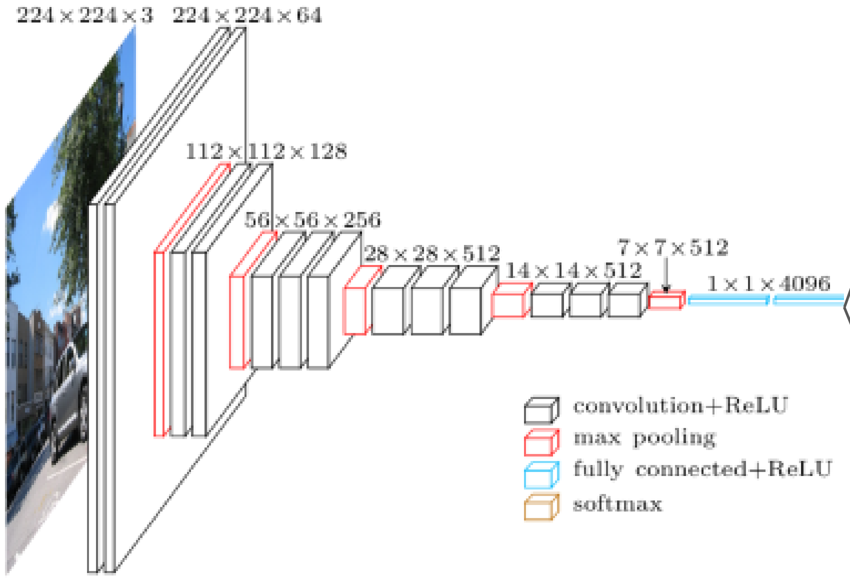


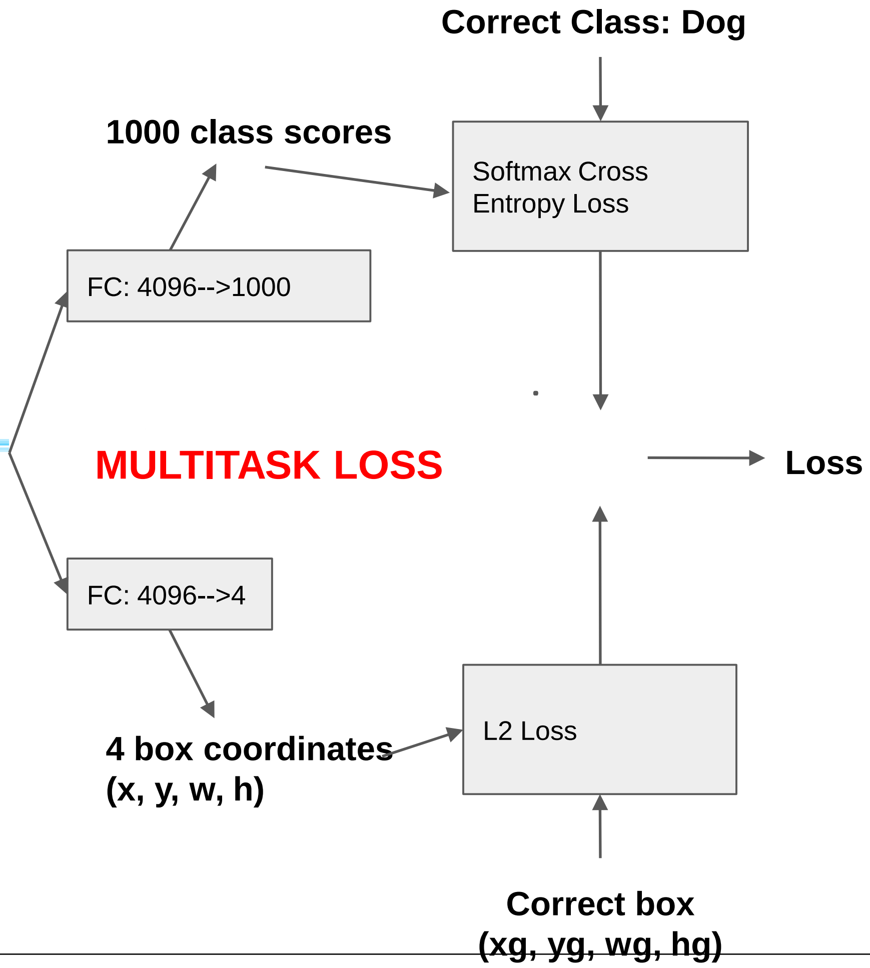


Treat localization as a regression problem

Classification + Localization

How would you do it
for multiple objects?

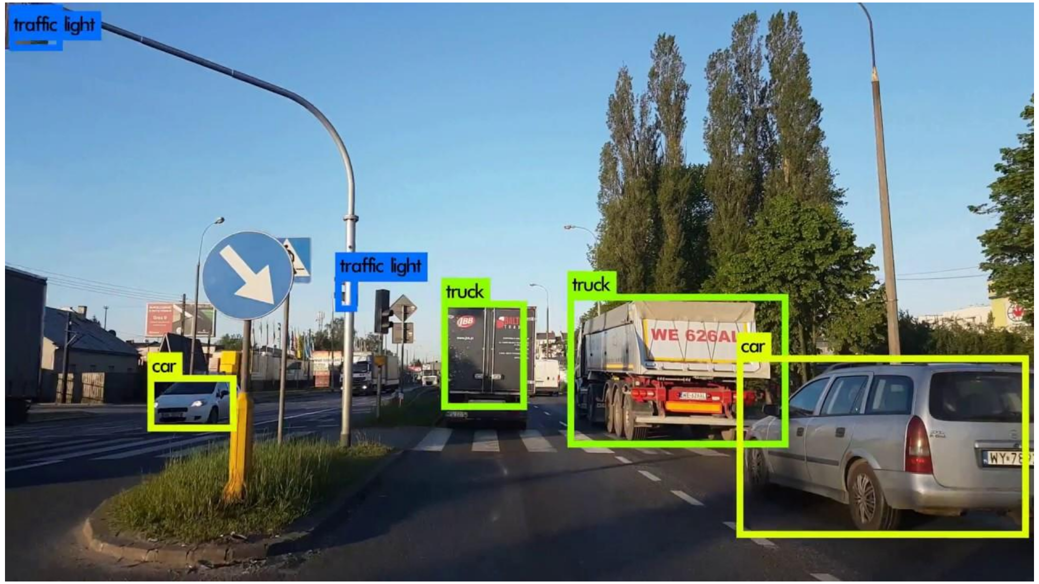




Treat localization as a
regression problem

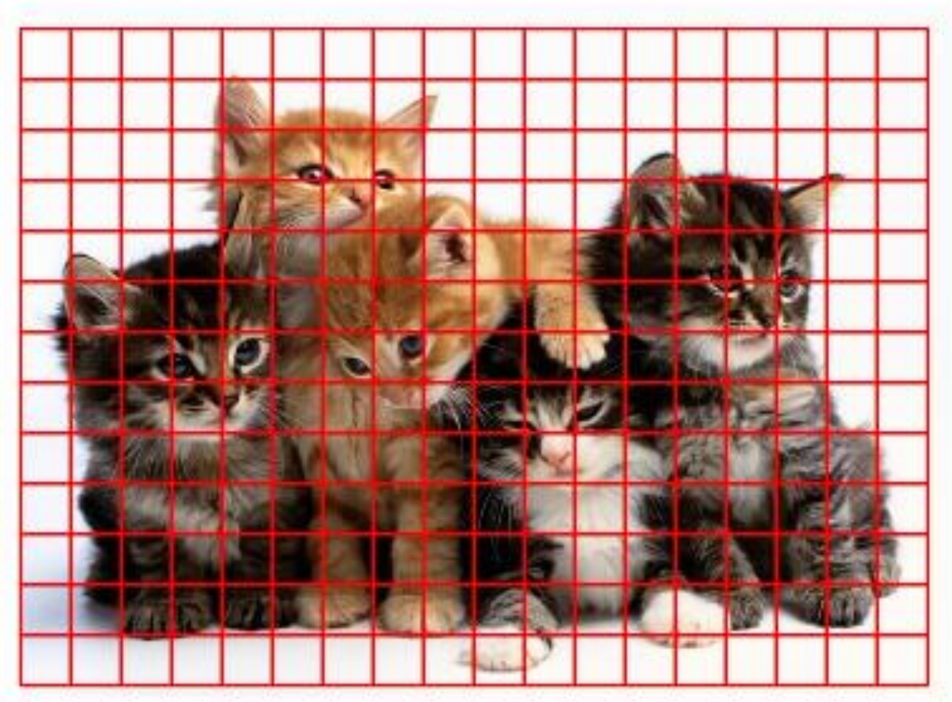
Object Detection

Classification + bounding box regression for **multiple boxes**



Object Detection

Brute force approach - run a classifier for **every possible box**



Object Detection

Brute force approach - run a classifier for **every possible box** with a **sliding window**

Histograms of Oriented Gradients for Human Detection Dalal and Triggs 2005

Object Detection

Brute force approach - run a classifier for **every possible box** with a **sliding window at every scale on an image pyramid**

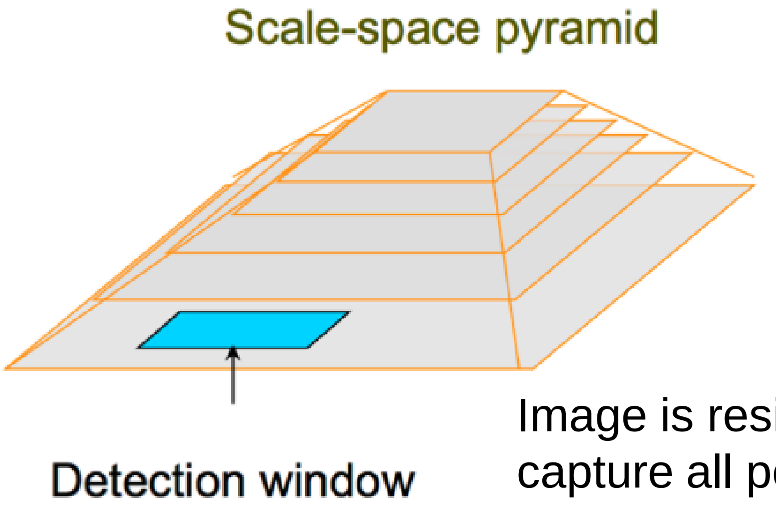


Image is resized several times to capture all possible scales

Histograms of Oriented Gradients for Human Detection Dalal and Triggs 2005

Object Detection

Brute force approach - run a classifier for **every possible box** with a **sliding window at every scale on an image pyramid**

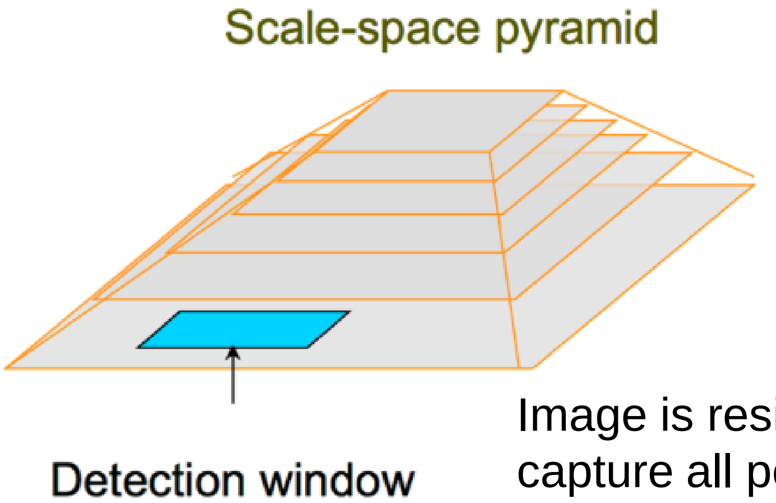


Image is resized several times to capture all possible scales

Histograms of Oriented Gradients for Human Detection Dalal and Triggs 2005

Very slow, unfeasible if your classifier is heavy

Object Detection

Smarter approach?

Object Detection

Smarter approach - run classifier for **region proposals or boxes likely to contain objects**

Object Detection

Smarter approach - run classifier for **region proposals or boxes likely to contain objects**

- How to reduce number of boxes?

Object Detection

Smarter approach - run classifier for **region proposals or boxes likely to contain objects**

- How to reduce number of boxes?
- Find ‘blobby’ image regions which are likely to contain objects

Object Detection

Smarter approach - run classifier for **region proposals or boxes likely to contain objects**

- How to reduce number of boxes?
- Find ‘blobby’ image regions which are likely to contain objects
- Class-agnostic object detector - “**Region Proposals**”

Region Proposals

Greedily combine sub-segmentation to produce larger candidate object locations

Selective Search for Object Recognition, Uijlings et al (2013)

Region Proposals

Greedily combine sub-segmentation to produce larger candidate object locations

Selective Search for Object Recognition, Uijlings et al (2013)

Candidate object locations

↓

Extract HOG features over windows

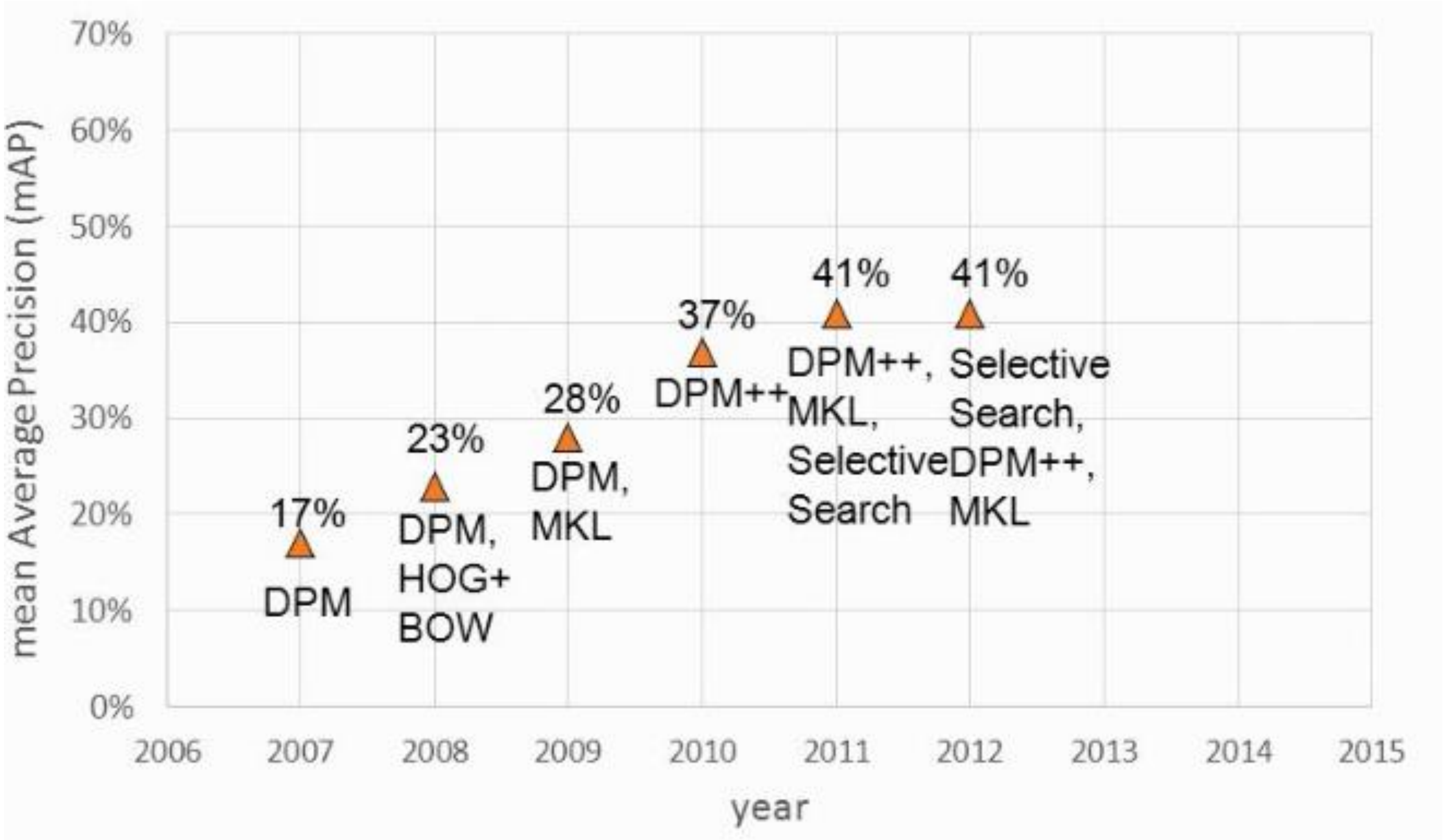
↓

Run Linear SVM classifier

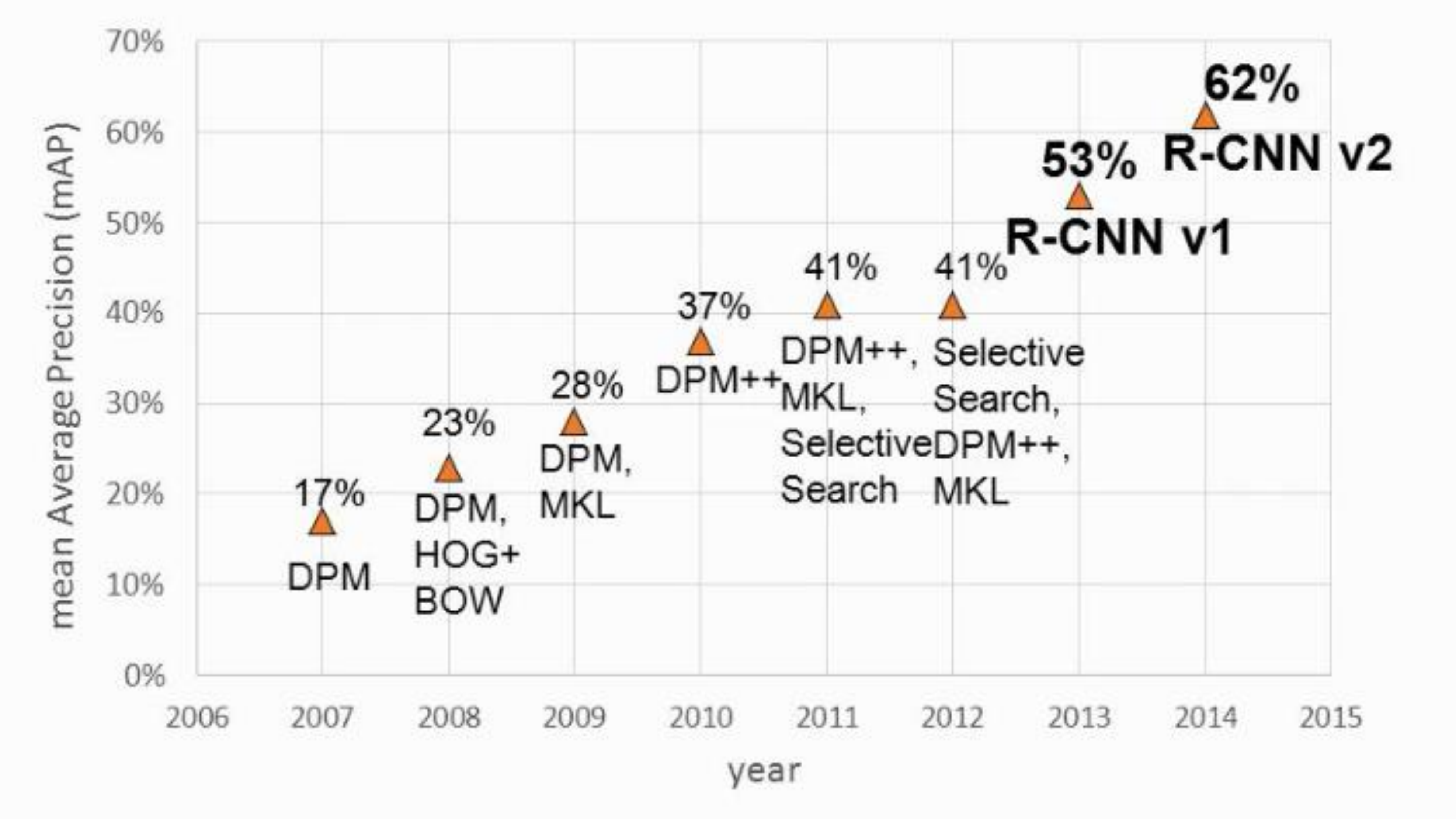
↓

Run post-processing (Non-Maximum Suppression)

Pre deep learning



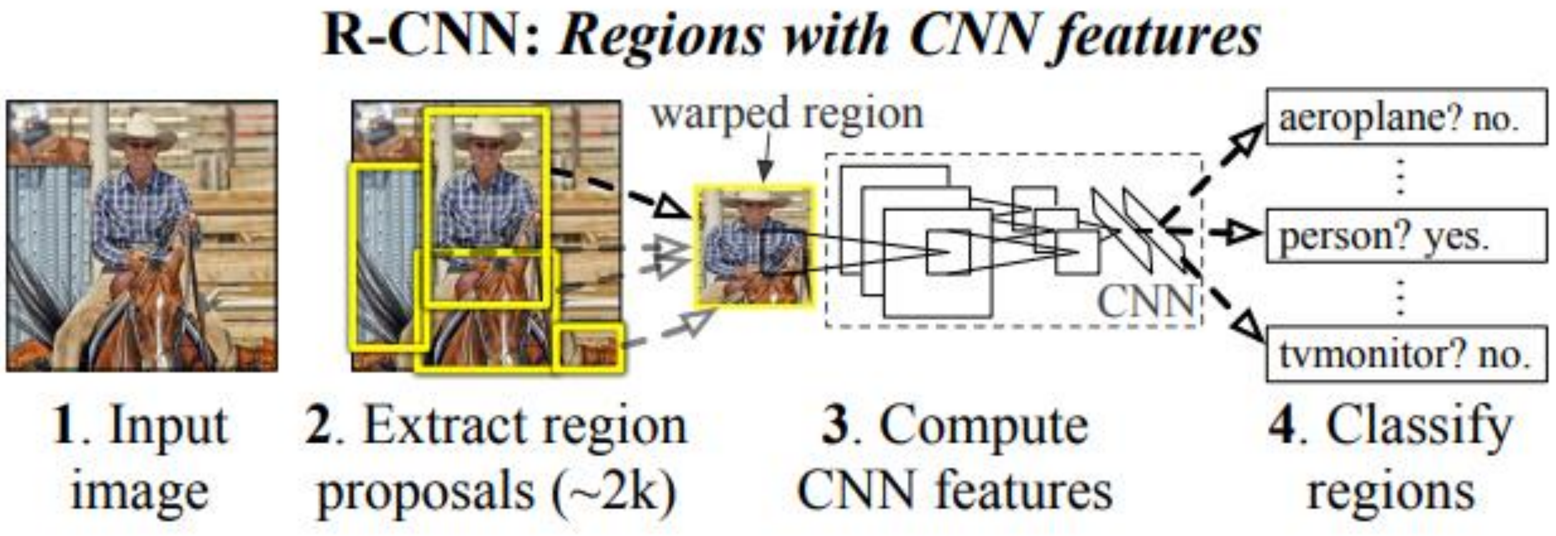
Post deep learning



81.3%

R-CNN

Rich feature hierarchies for accurate object detection and semantic segmentation, Girshick et al (2013)



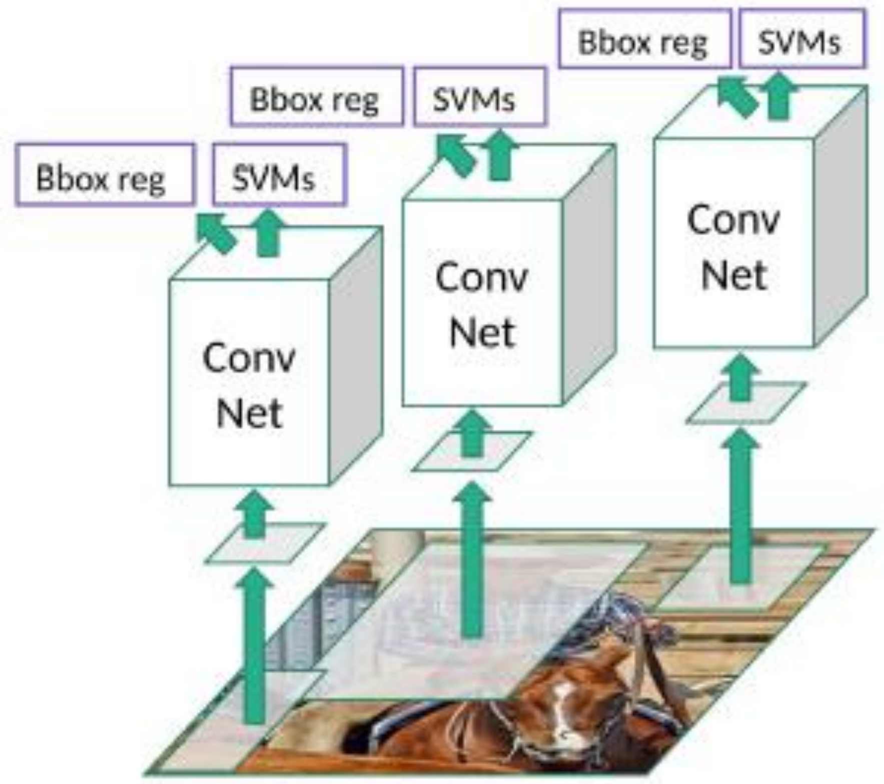
SVM classifier

Exactly the same as Selective Search, except for CNN features instead of HOG

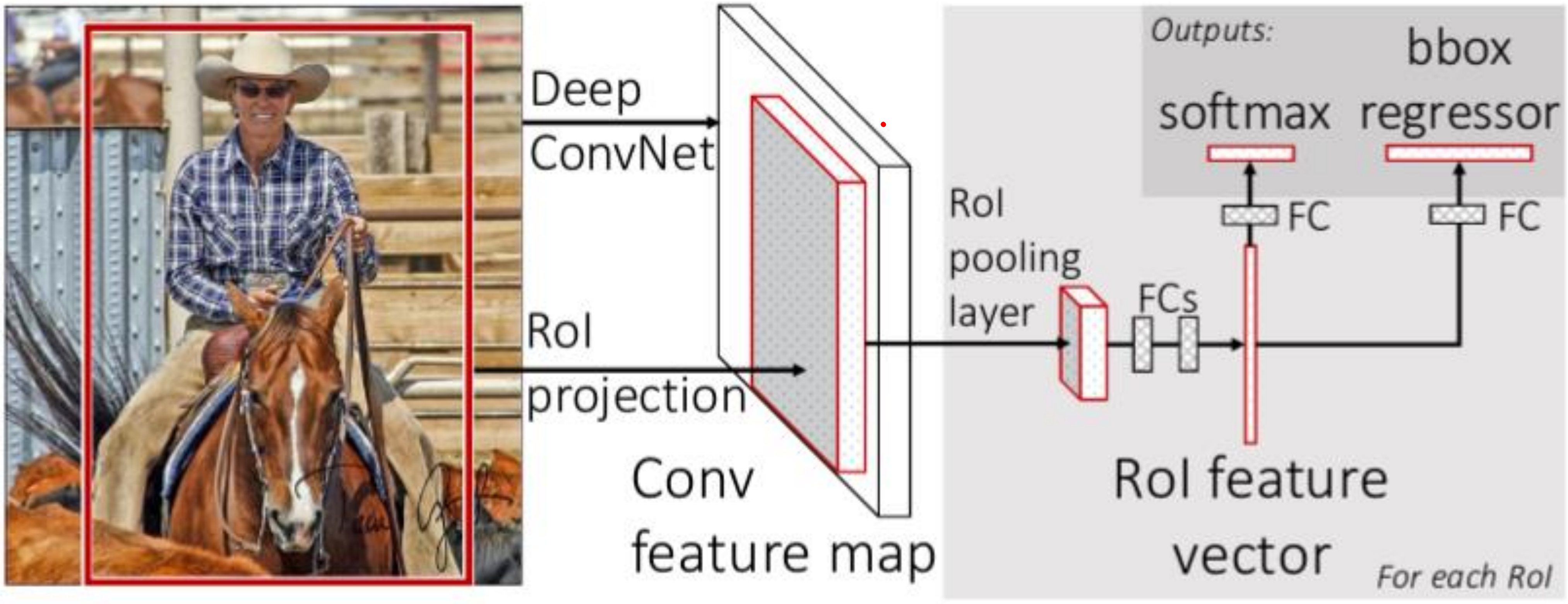
https://arxiv.org/abs/1311.2524

R-CNN: Problems

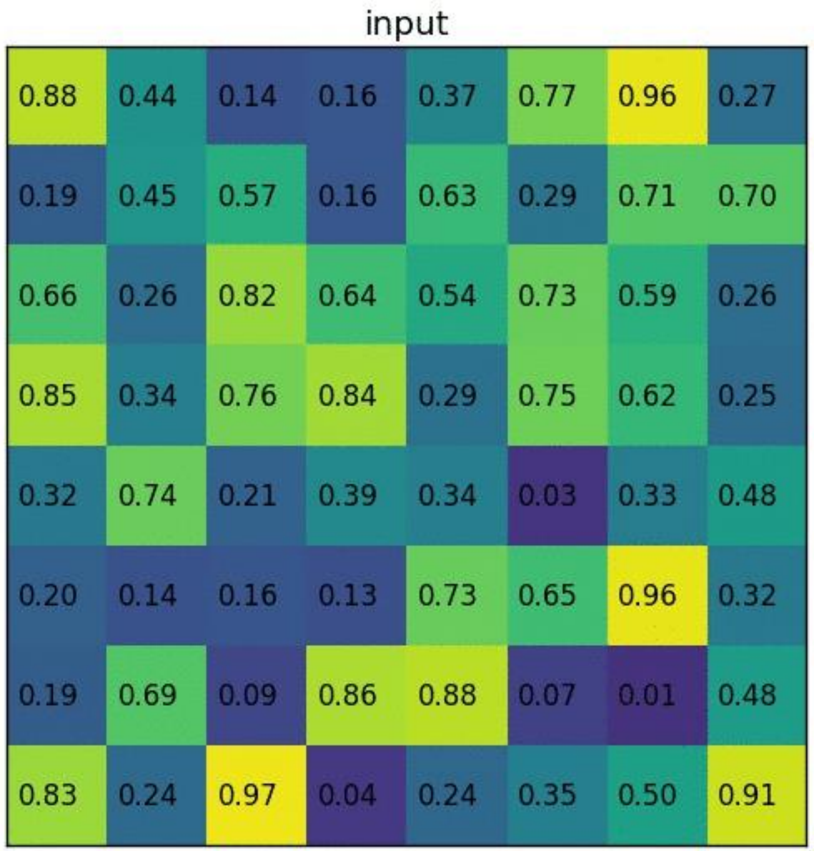
- Ad hoc training objectives
    - Fine-tune network with softmax classifier (log loss
    - Train post-hoc linear SVMs (hinge loss)
    - Train post-hoc bounding-box regressions (least squares)
- Training is slow (84h), takes a lot of disk space
- Inference (detection) is slow
    - 47s / image with VGG16 [Simonyan & Zisserman. ICLR15]
    - Fixed by SPP-net [He et al. ECCV14]



Fast R-CNN: Region Proposals in Feature Space

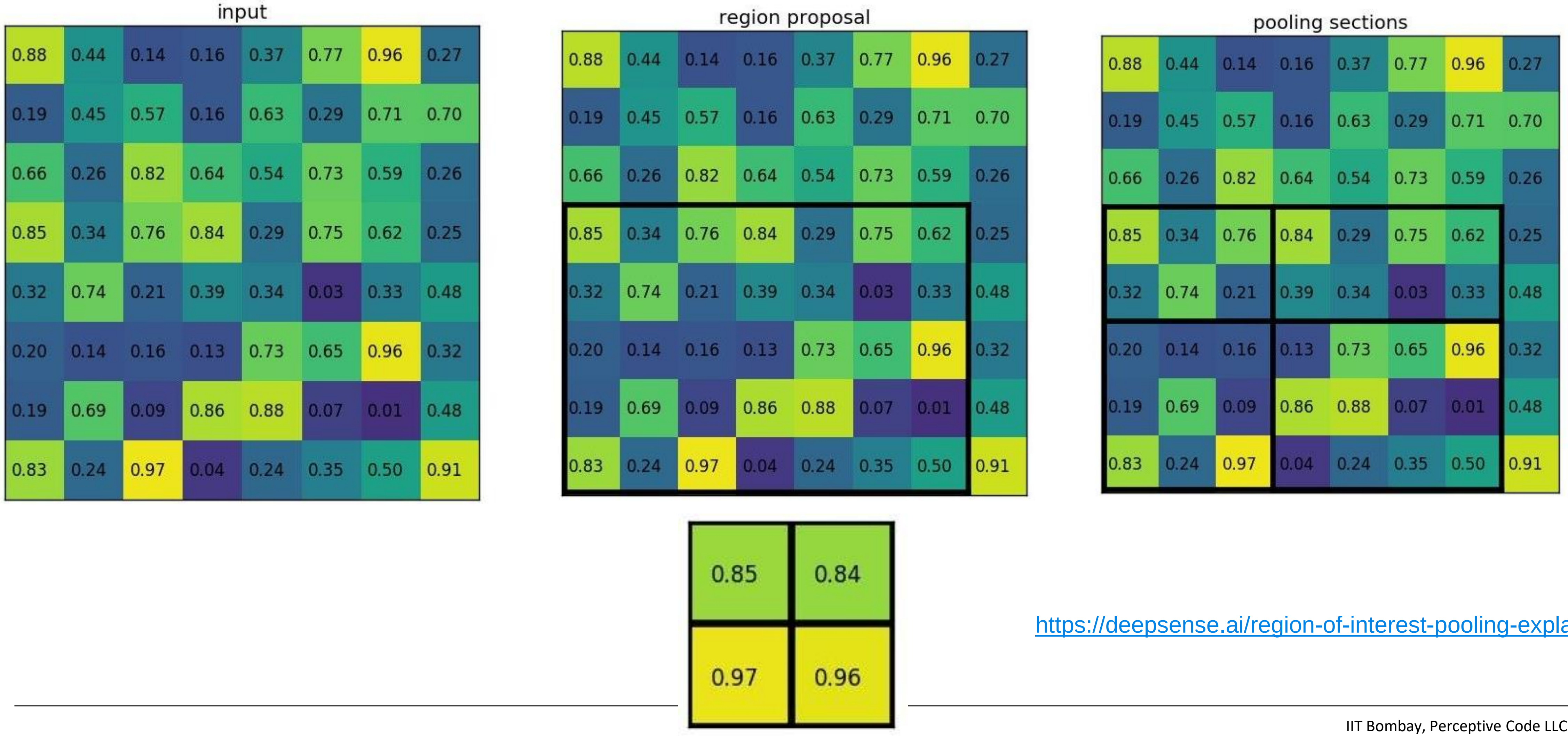


Fast R-CNN: ROI Pooling



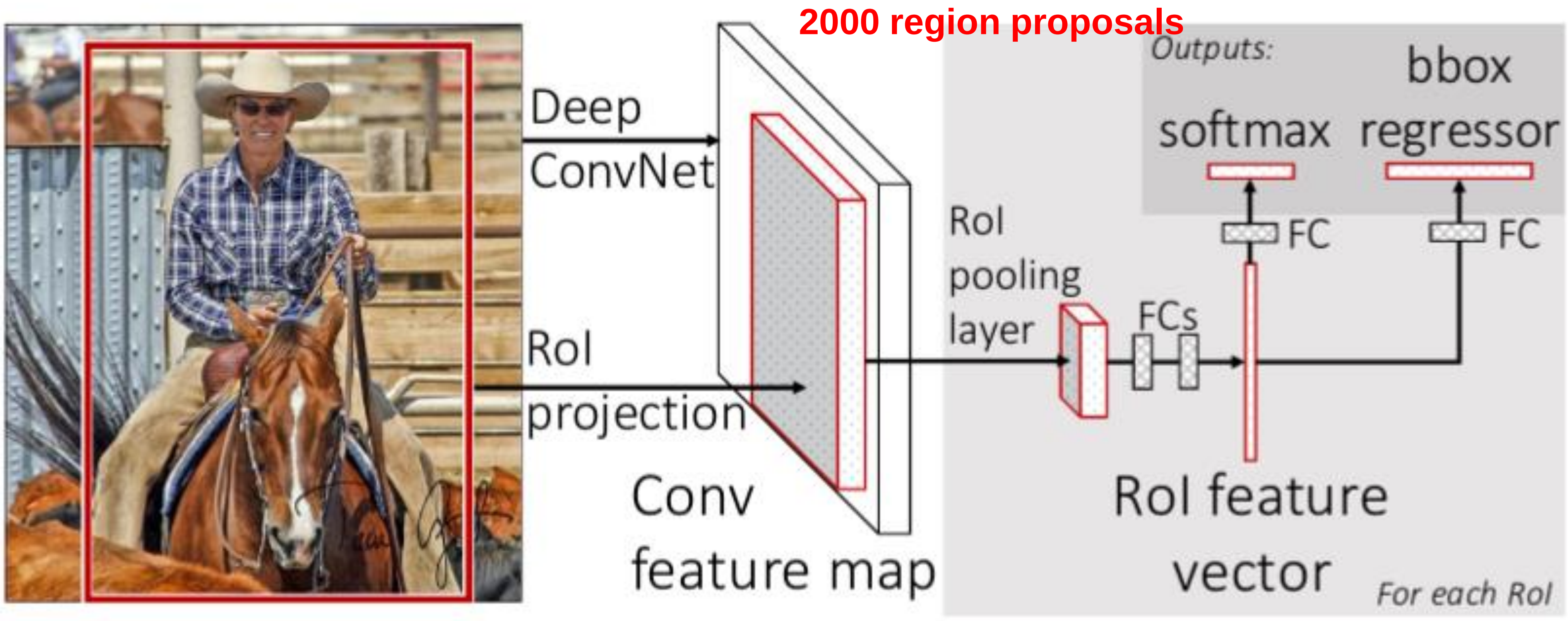
https://deepsense.ai/region-of-interest-pooling-explained/

Fast R-CNN: ROI Pooling



https://deepsense.ai/region-of-interest-pooling-explained/

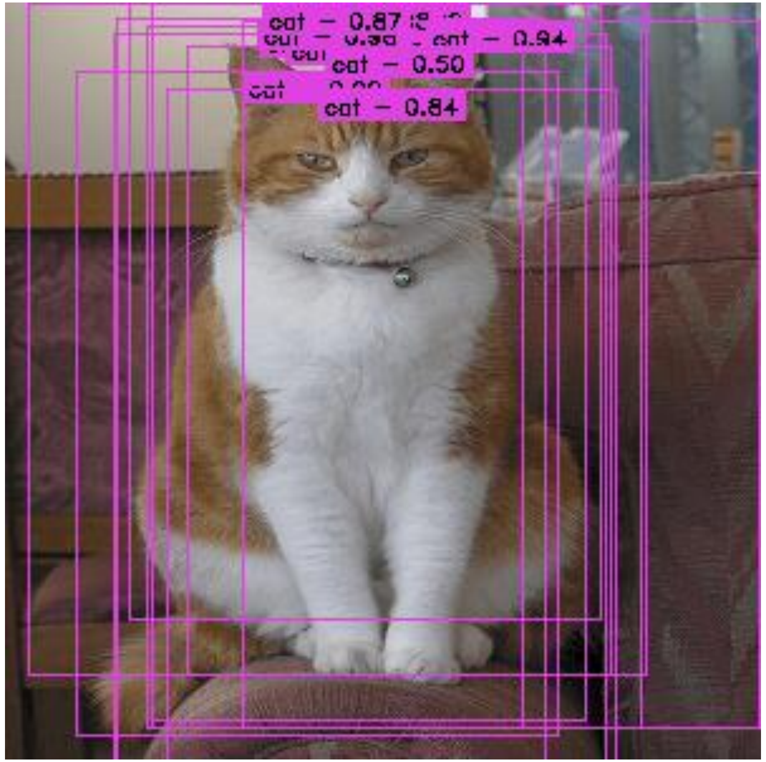
Fast R-CNN



2000 region proposals
Deep
ConvNet
RoI
projection
Conv
feature map
RoI
pooling
layer
FCs
RoI feature
vector
Outputs:
softmax
bbox
regressor
FC
FC
For each RoI

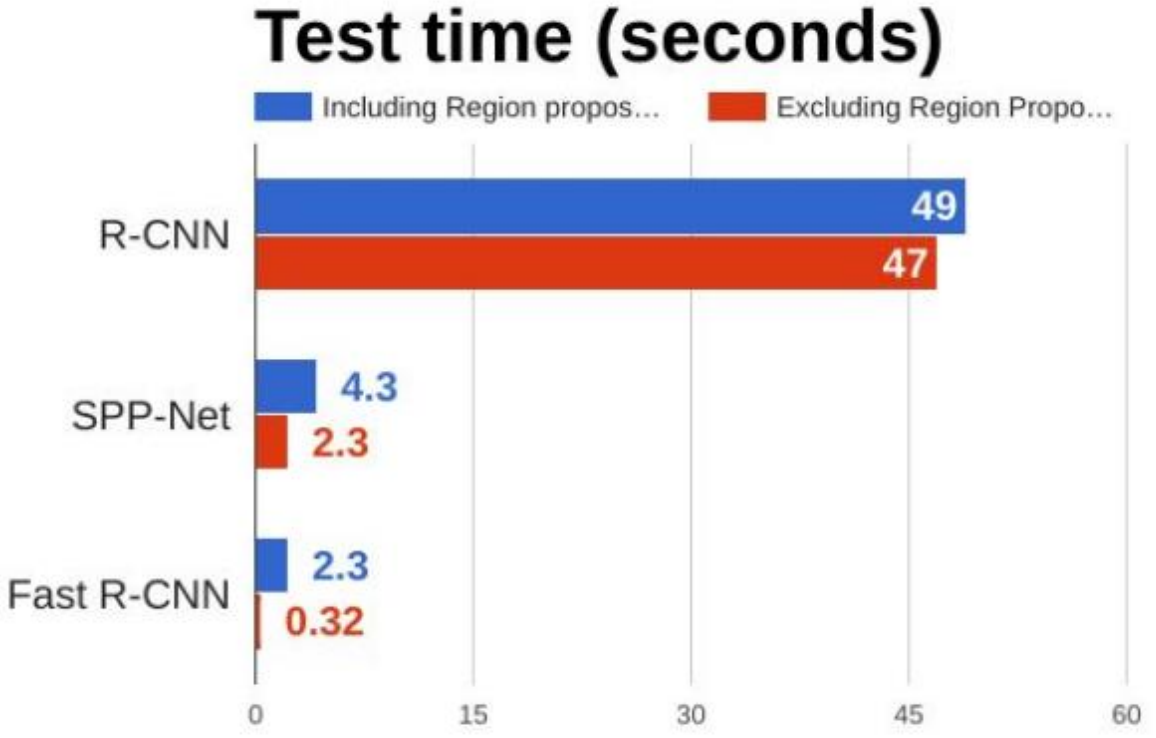
Non Maximum Suppression

Remove boxes with high overlap but lower score



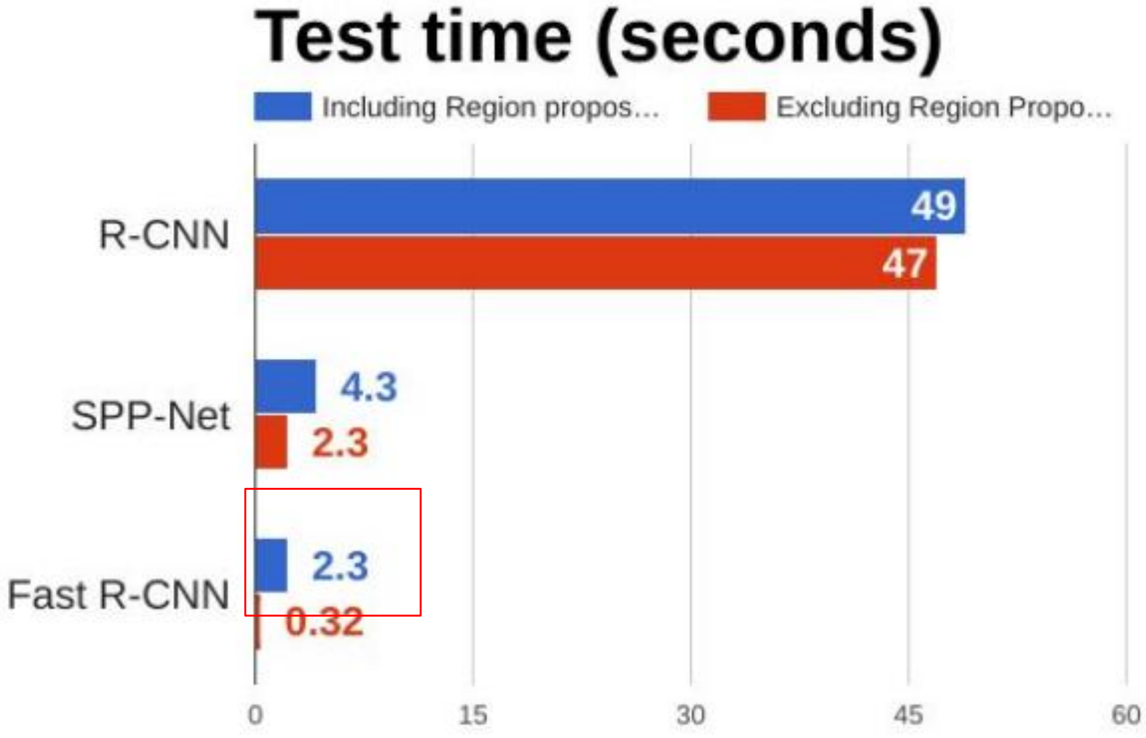
- Discard boxes with softmax_output<confidence_threshold
- Sort boxes in descending order of confidence
- For box in sorted(box_list)
  - Discard all boxes with IoU>0.5 with box

Region Proposals



Region Proposals

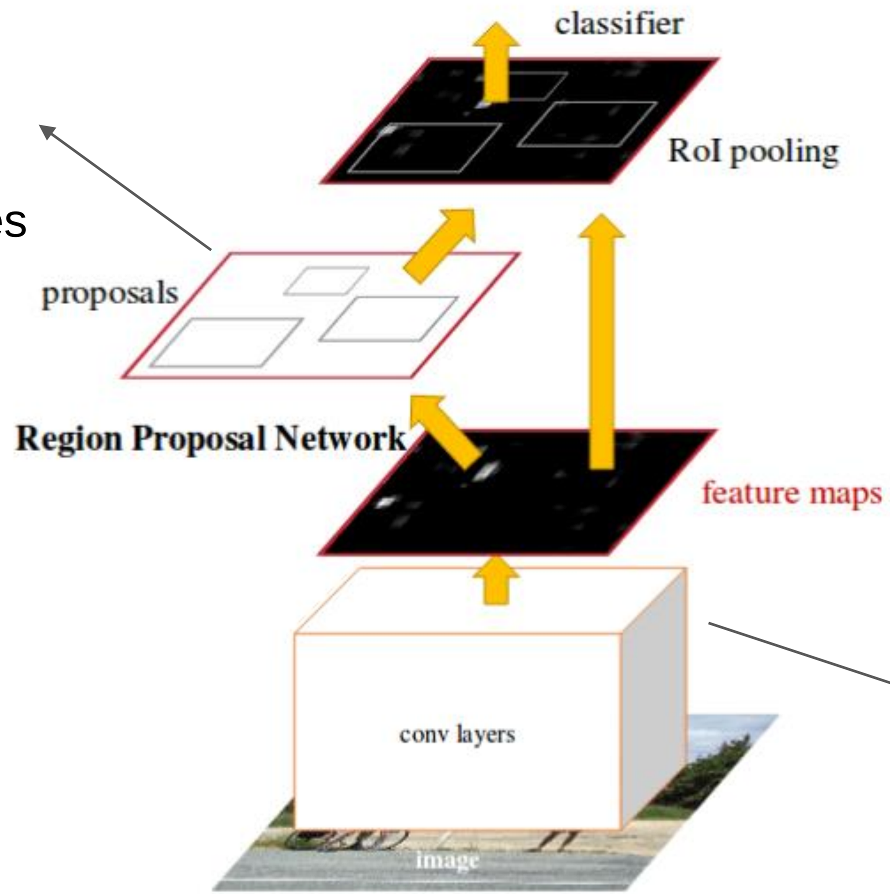
Maximum time still taken by Region proposal generation



Faster R-CNN

Faster R-CNN: Use a CNN for region proposals

Trained to regress
2400x
4 Box coordinates
2 objectness scores



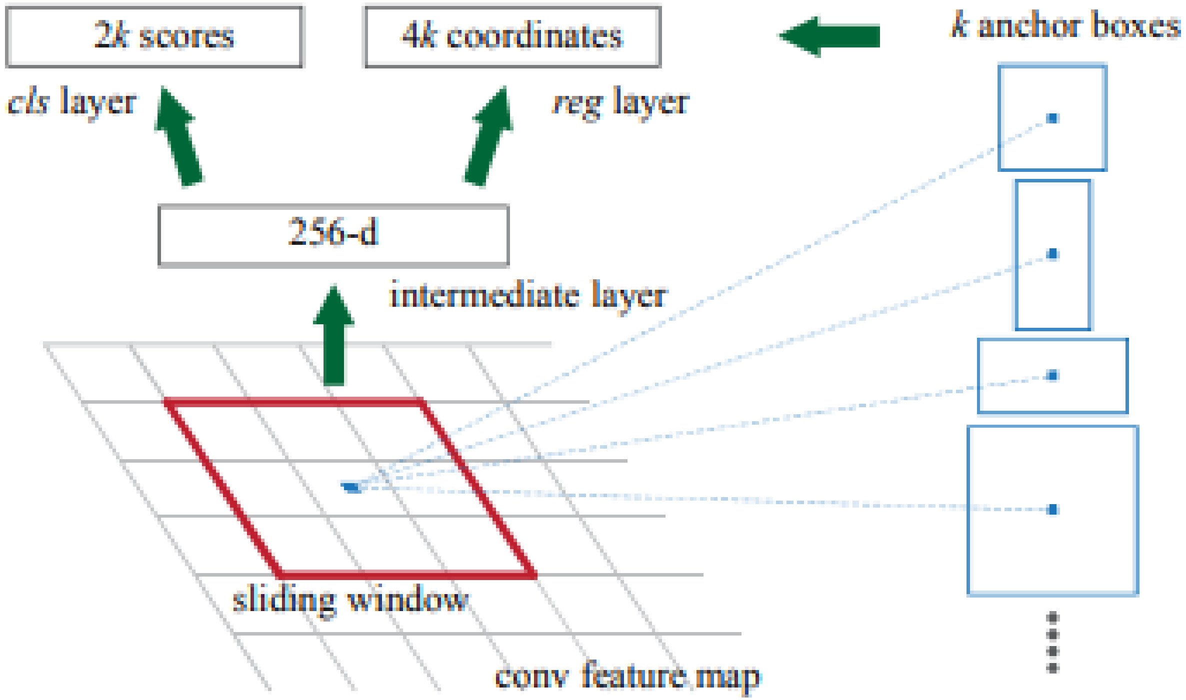
Core network computation shared by RPN and classifier head

https://arxiv.org/abs/1506.01497

Region Proposals using CNN

At each uniformly sampled grid cell, set of anchor boxes with different aspect ratios and scales

For anchors, we use 3 scales with box areas of 1282 , 2562 , and 5122 pixels, and 3 aspect ratios of 1:1, 1:2, and 2:1.



For each anchor box

- Regress to a final box with 4 numbers: (*dx, dy, dh, dw)*
- Predict 2 objectness scores (including background as a separate class)

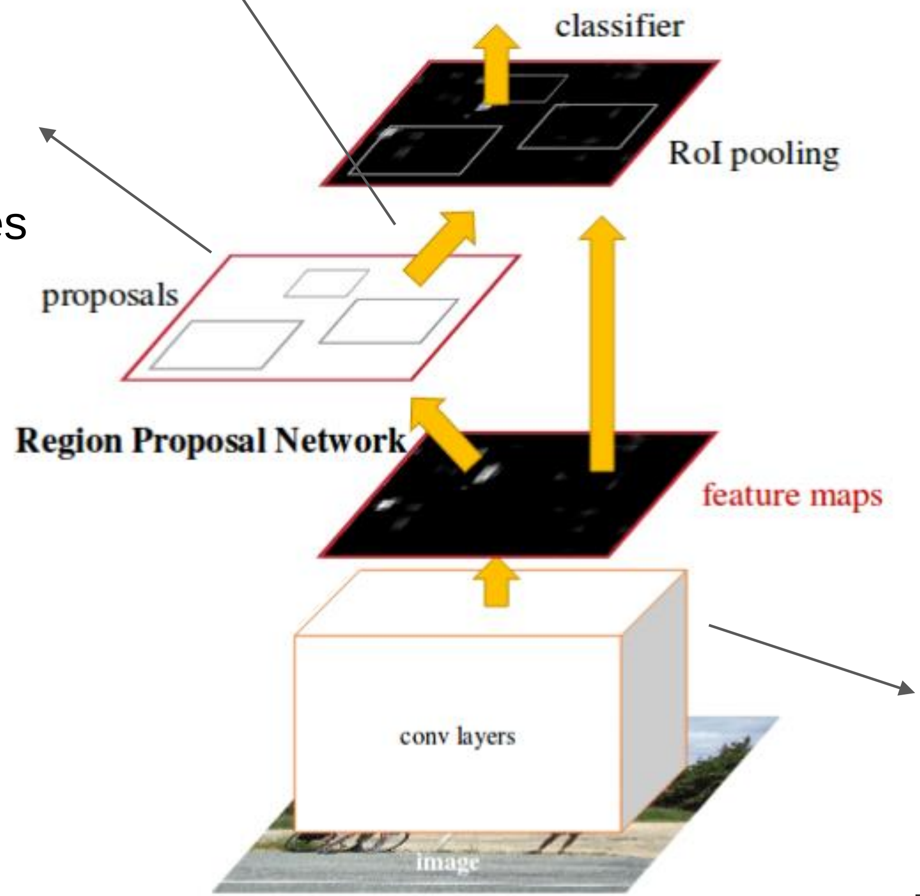
https://arxiv.org/abs/1506.01497

Faster R-CNN

3:1 negative:positive boxes selected
for training classifier

Runs at 200 ms per image

Trained to regress
2400x 20000x
4 Box coordinates
2 objectness scores

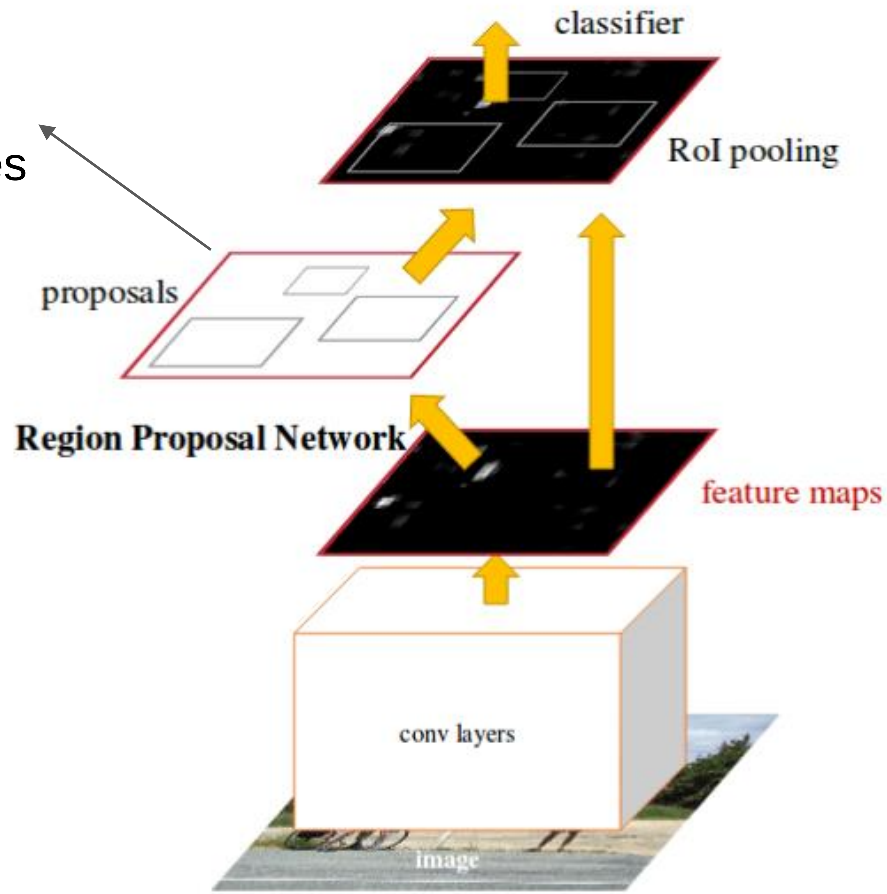


Core network computation shared
by RPN and classifier head

https://arxiv.org/abs/1506.01497

Faster R-CNN

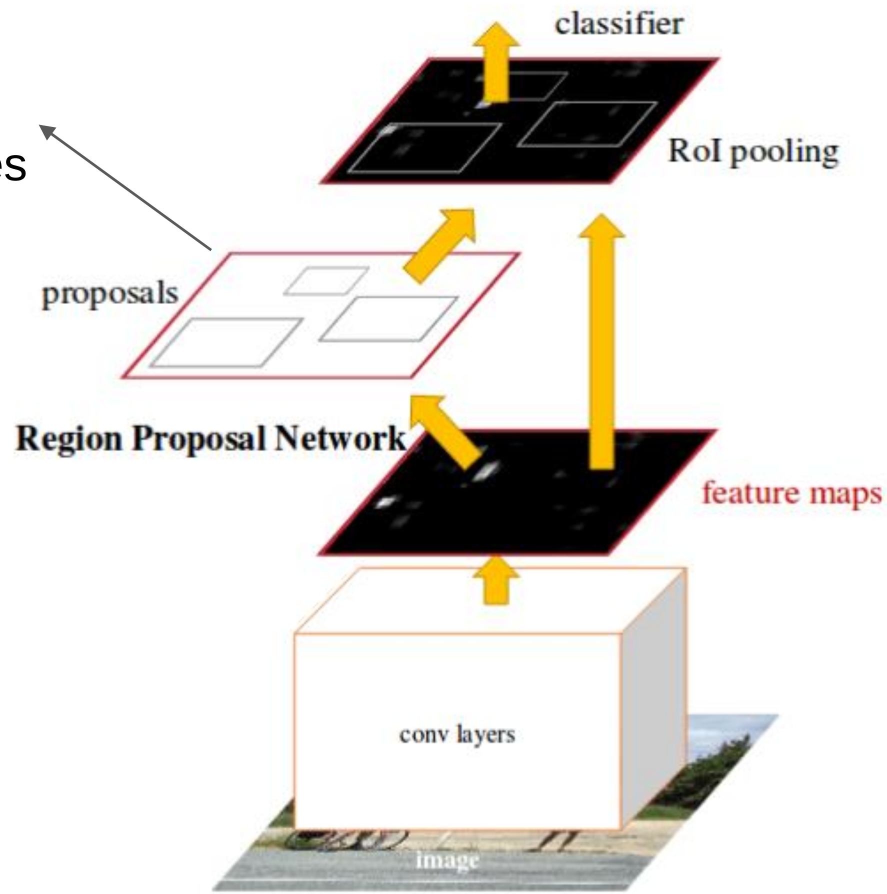
Trained to regress
4 Box coordinates
2 objectness scores



Can this be further simplified?

Faster R-CNN

Trained to regress
4 Box coordinates
2 objectness scores
1 class score

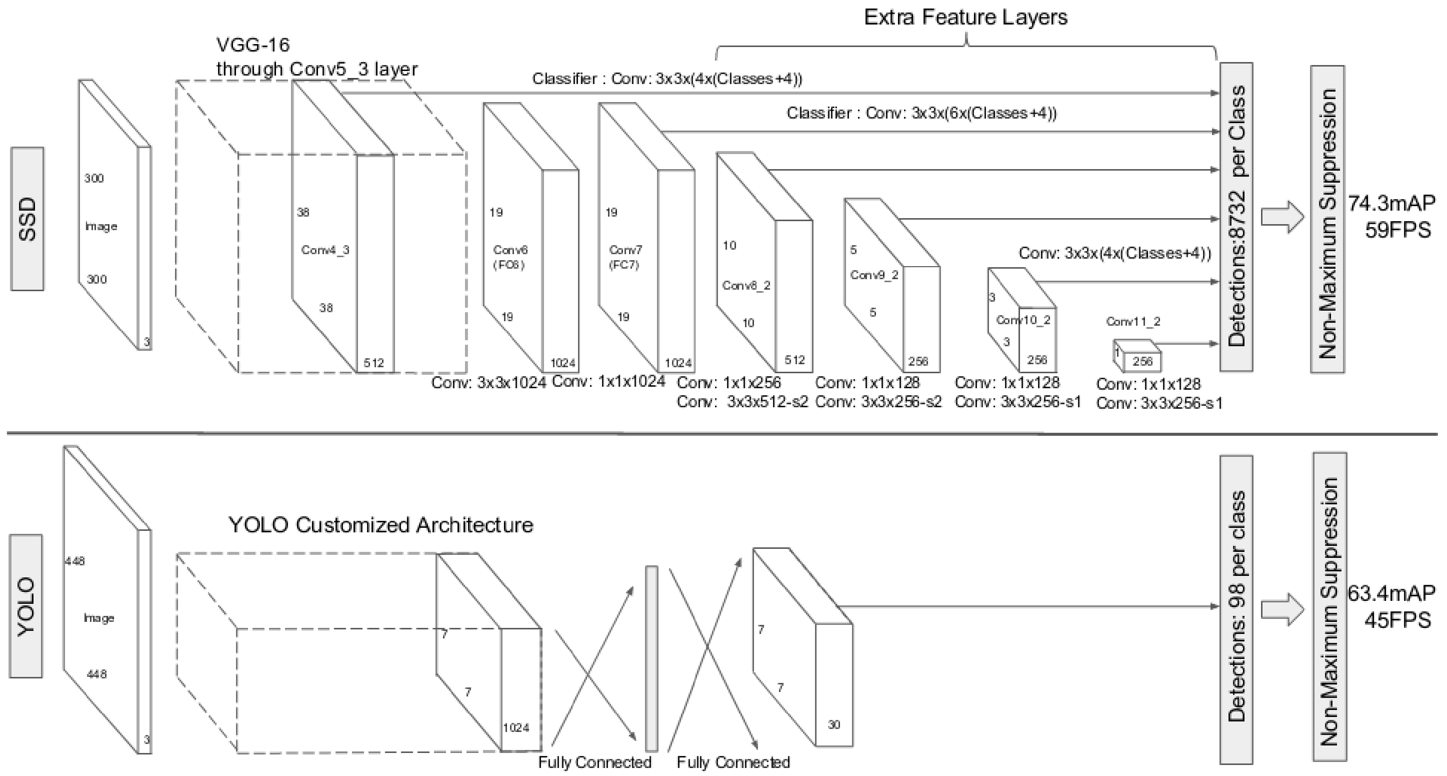


Compress into single stage -
Remove region-based
computation

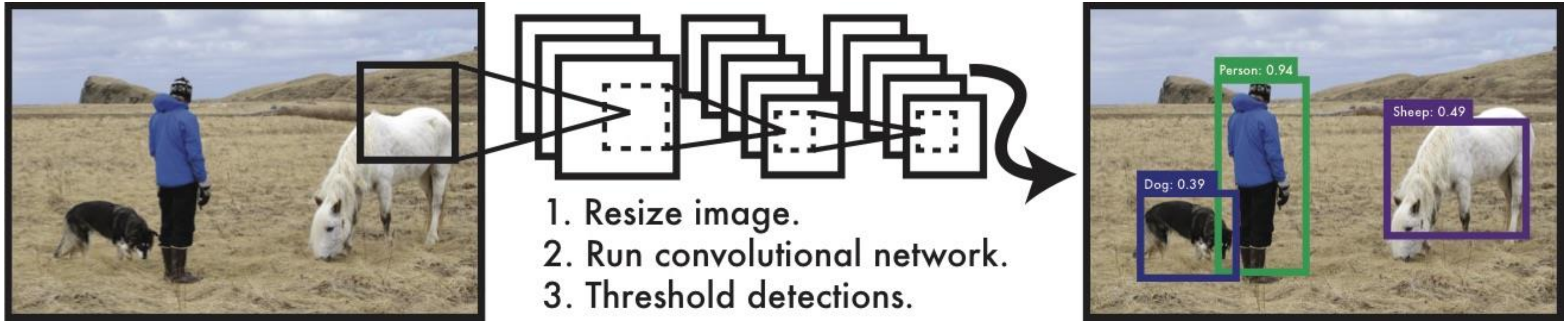
Single Stage Object Detection

YOLO: You Only Look Once & SSD: Single Shot Multibox Detector

Directly estimate box coordinates and class scores



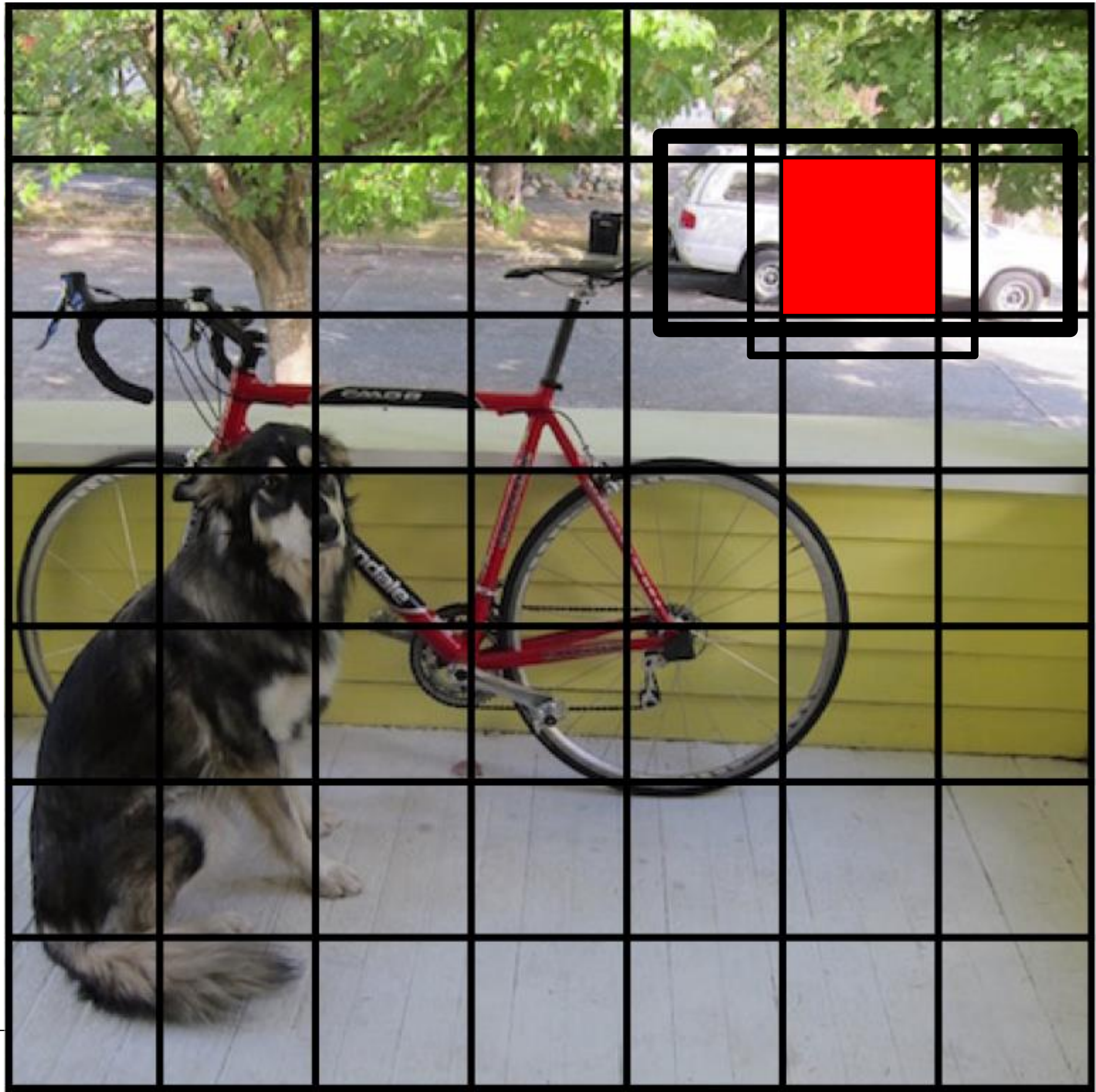
YOLO: You Only Look Once



https://arxiv.org/abs/1506.02640

YOLO: You Only Look Once

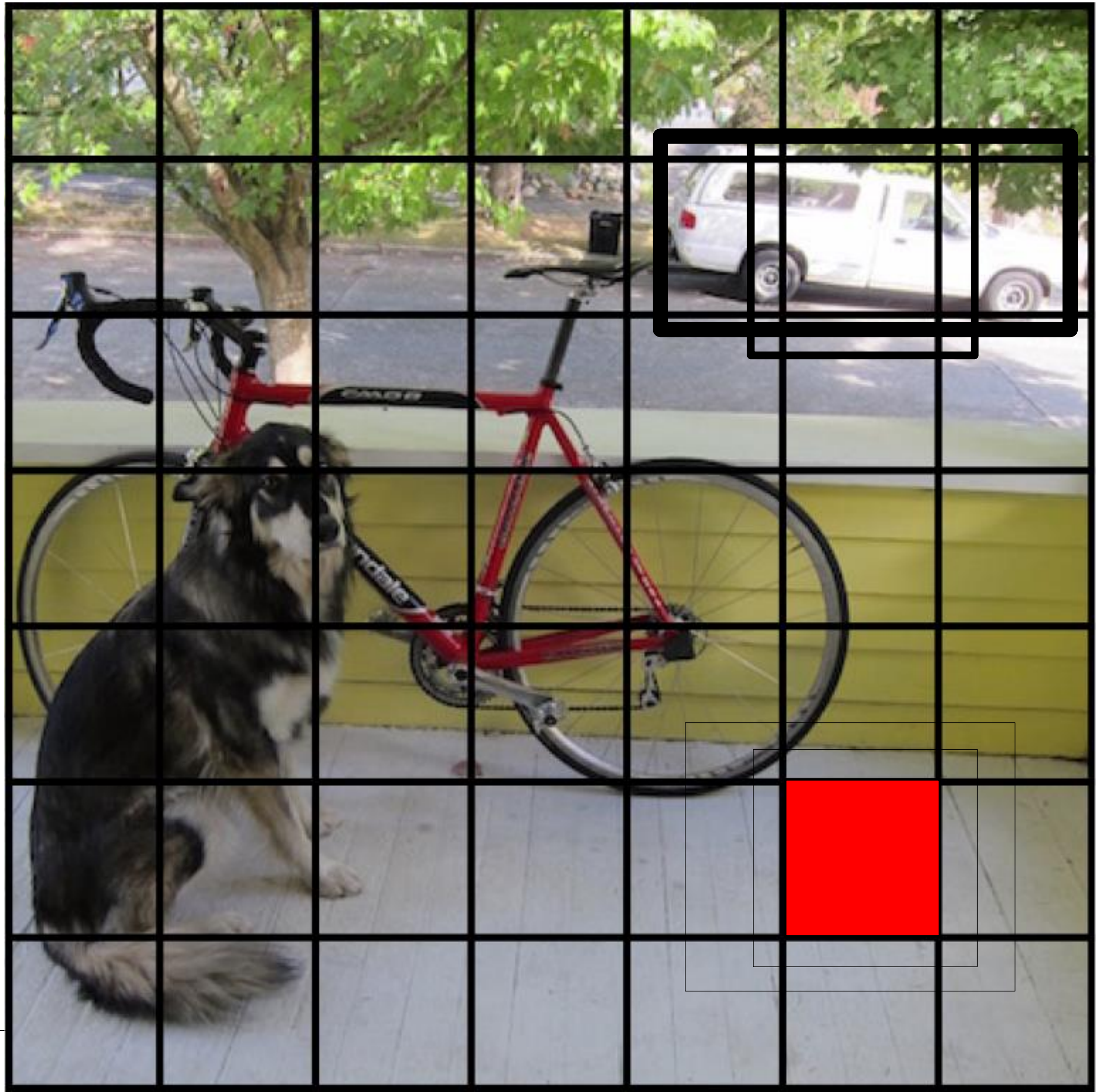
- Split the image into a grid
- Each cell predicts boxes and confidences P(Object)



https://arxiv.org/abs/1506.02640

YOLO: You Only Look Once

- Split the image into a grid
- Each cell predicts boxes and confidences P(Object)



https://arxiv.org/abs/1506.02640

YOLO: You Only Look Once

- Split the image into a grid
- Each cell predicts boxes and confidences P(Object)

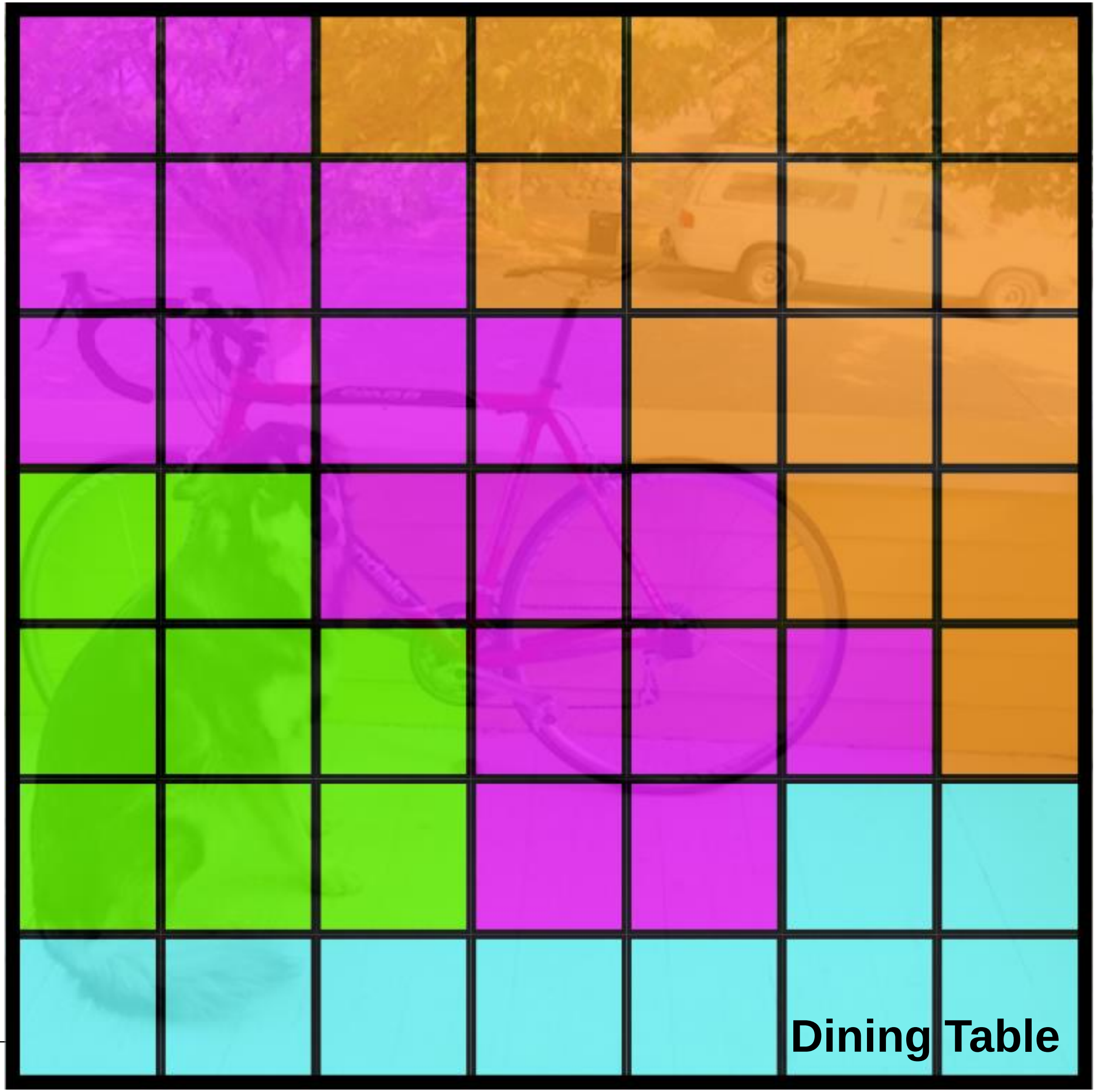
https://arxiv.org/abs/1506.02640

YOLO: You Only Look Once

- Split the image into a grid
- Each cell predicts boxes and confidences P(Object)
- Each cell also predicts a class probability

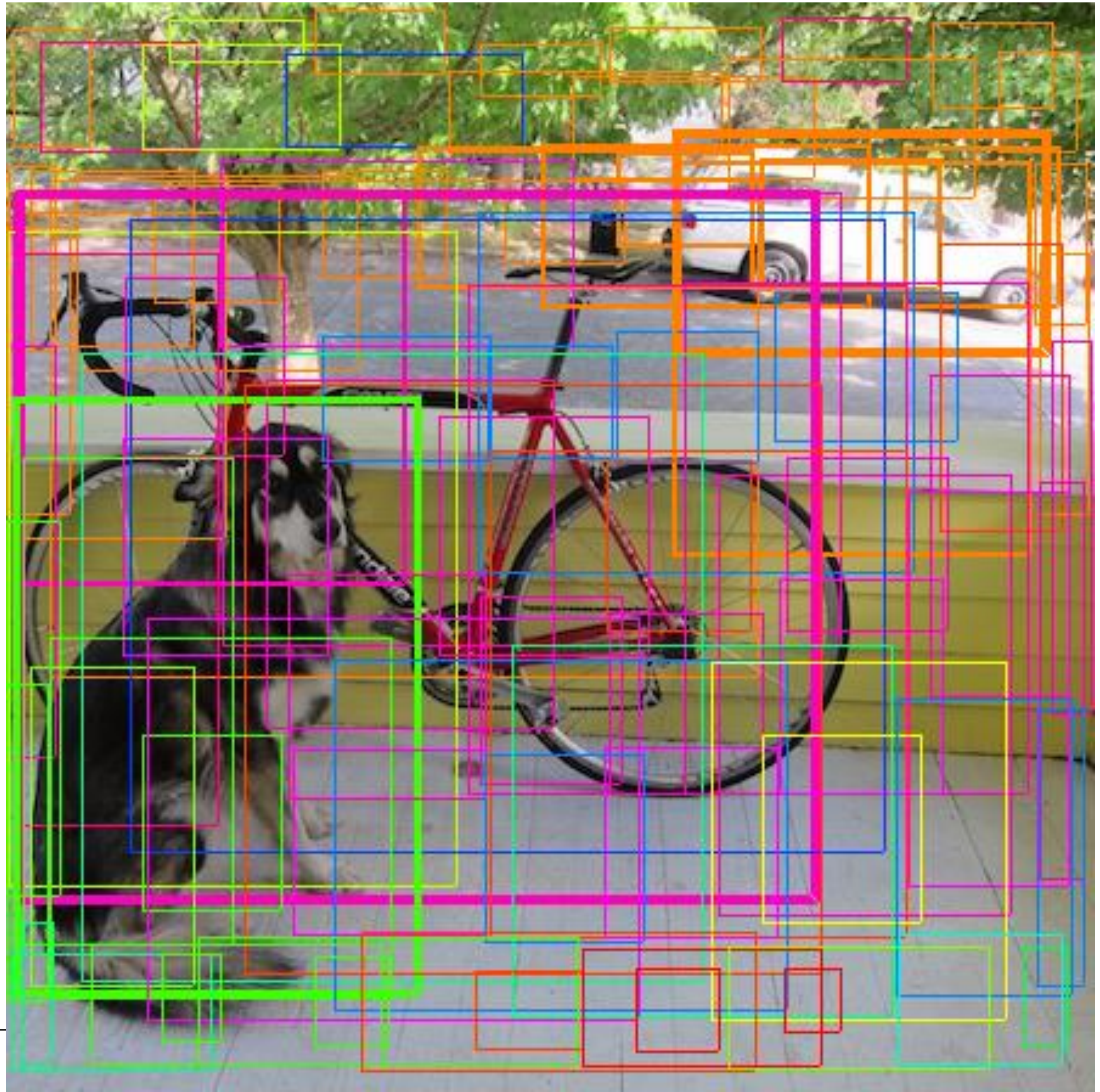
Conditioned on object: P(Car | Object)

https://arxiv.org/abs/1506.02640



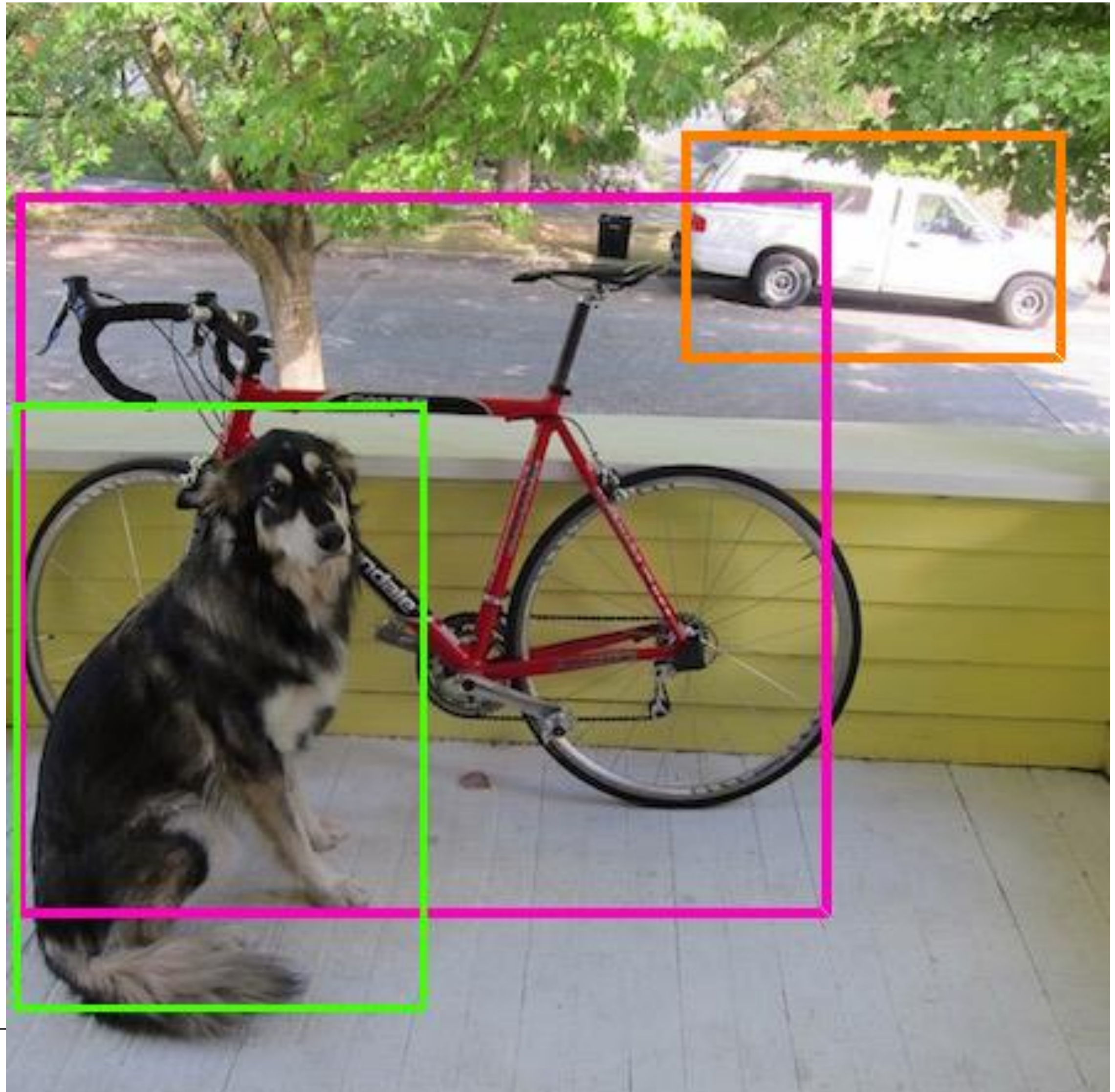
YOLO: You Only Look Once

- Split the image into a grid
- Each cell predicts boxes and confidences P(Object)
- Each cell also predicts a class probability
- Then box and class predictions are combined



YOLO: You Only Look Once

- Split the image into a grid
- Each cell predicts boxes and confidences P(Object)
- Each cell also predicts a class probability
- Then box and class predictions are combined
- Followed by NMS



YOLO: You Only Look Once output space

Each cell predicts:

- For each bounding box:

- 4 coordinates (x, y, w, h)
- 1 confidence value

- Some number of class probabilities

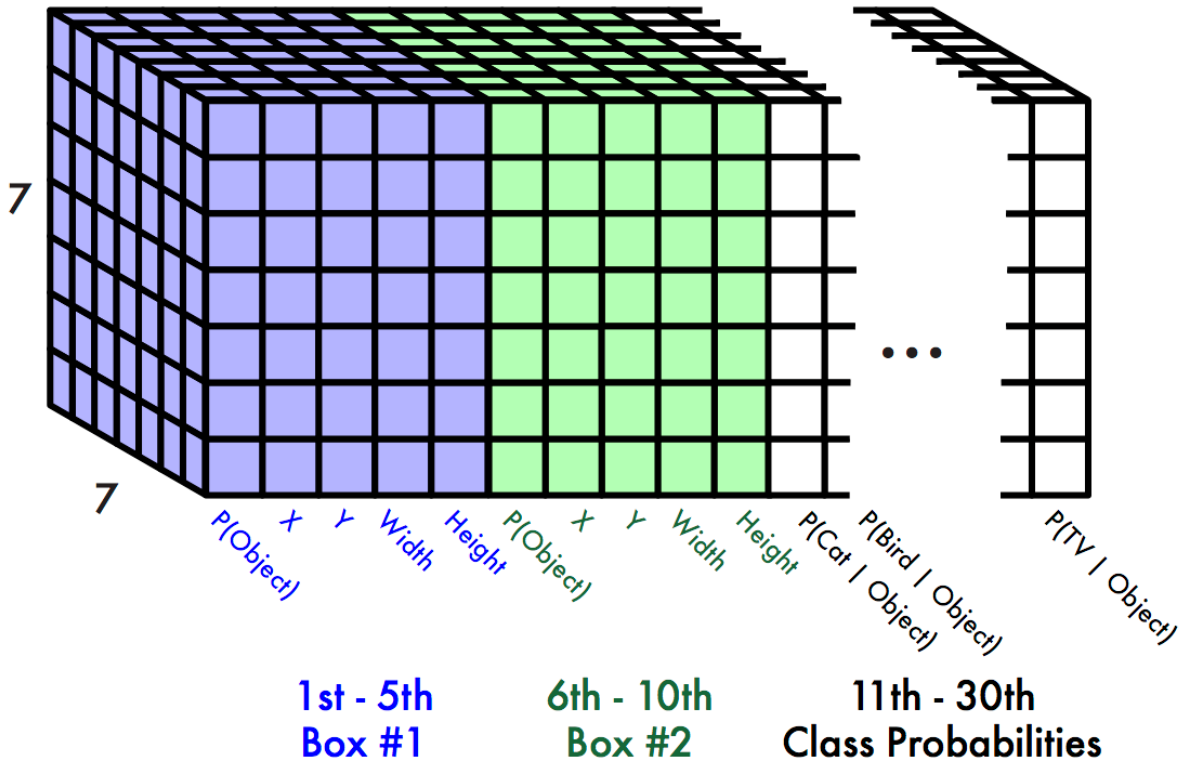
For Pascal VOC:

- 7x7 grid

- 2 bounding boxes / cell

- 20 classes

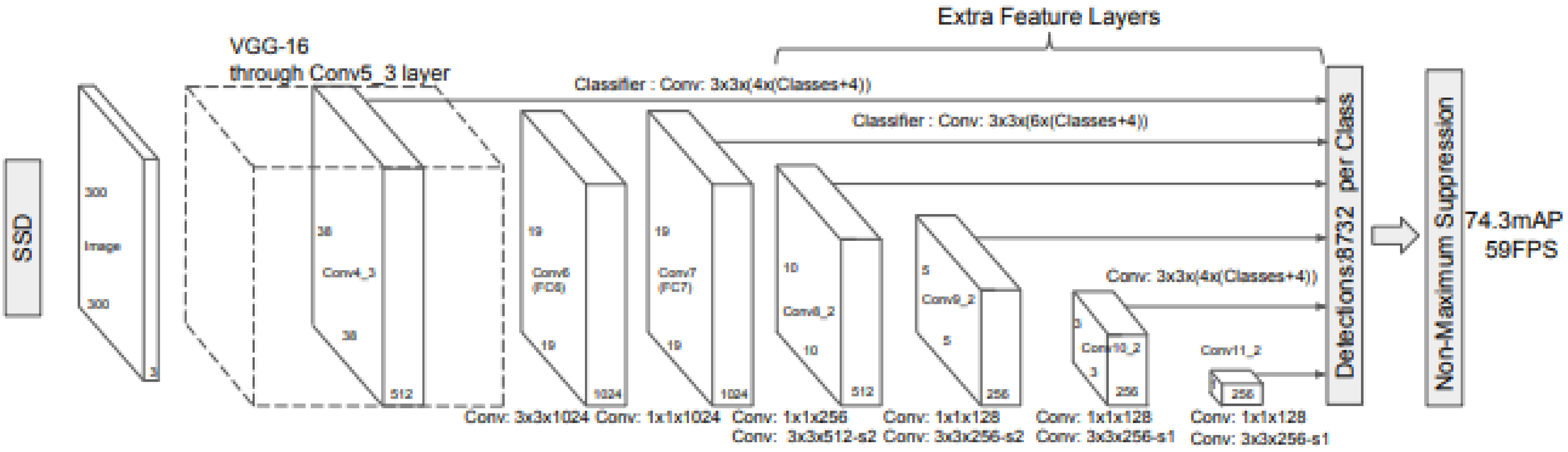
7 x 7 x (2 x 5 + 20) = 7 x 7 x 30 tensor = **1470 outputs**



https://arxiv.org/abs/1506.02640

SSD: Single Shot Multibox Detector

Like YOLO, but predicts using a multi-scale pyramidal feature hierarchy



https://arxiv.org/abs/1512.02325

Performance metrics for Object Detection

Performance metrics for Object Detection

- IoU
- Precision and Recall
- Mean Average Precision (mAP)

Performance metrics for Object Detection

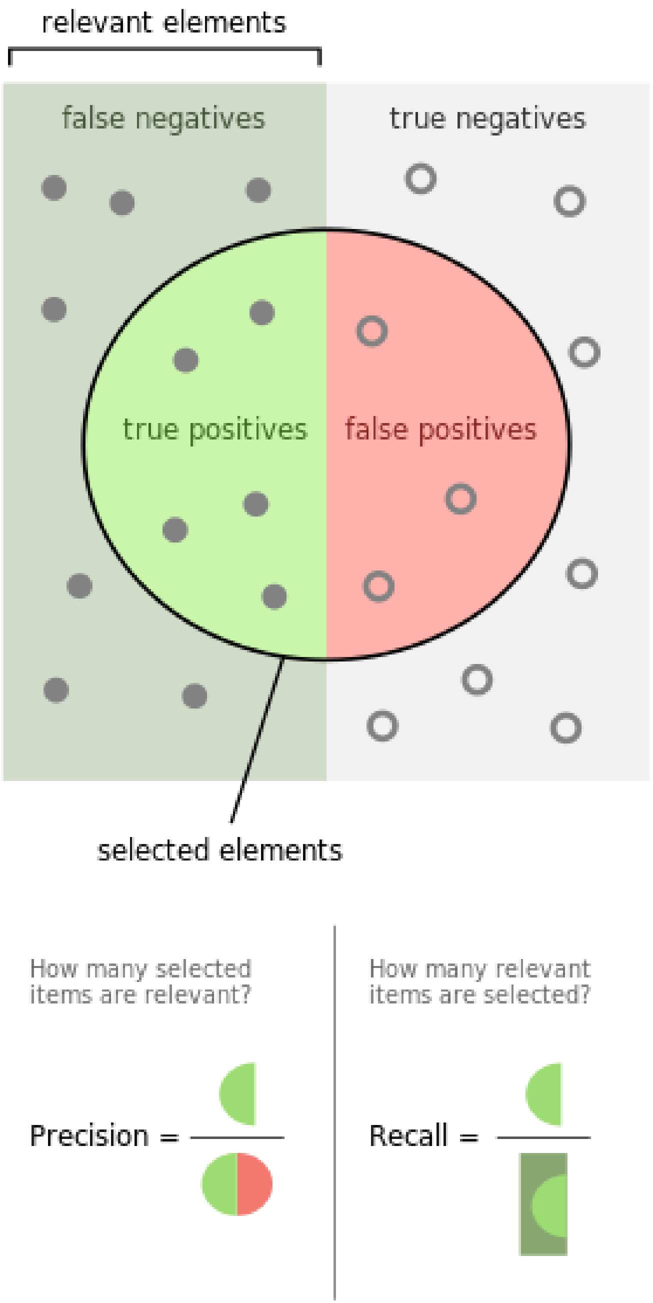
IoU (intersection over union): Measure of box similarity

Area of overlap / area of union

Performance metrics for Object Detection

Precision: what percentage of your positive predictions are correct

Recall: what percentage of ground truth objects were found



Performance metrics for Object Detection

Mean Average Precision (mAP) for object detection

Step 1: Sort predictions according to confidence (usually classifier's output after softmax)

Step 2: Calculate IoU of every predicted box with every ground truth box

Step 3: Match predictions to ground truth using IoU, correct predictions are those with IoU > threshold (typically 0.5)

Confidence	Rank	Correct
0.91	1	TRUE
0.87	2	TRUE
0.83	3	FALSE
0.81	4	TRUE
0.77	5	FALSE
0.65	6	TRUE
0.56	7	TRUE
0.40	8	FALSE
0.32	9	FALSE
0.31	10	TRUE

Performance metrics for Object Detection

Mean Average Precision (mAP) for object detection

Total number of ground truth boxes = 6

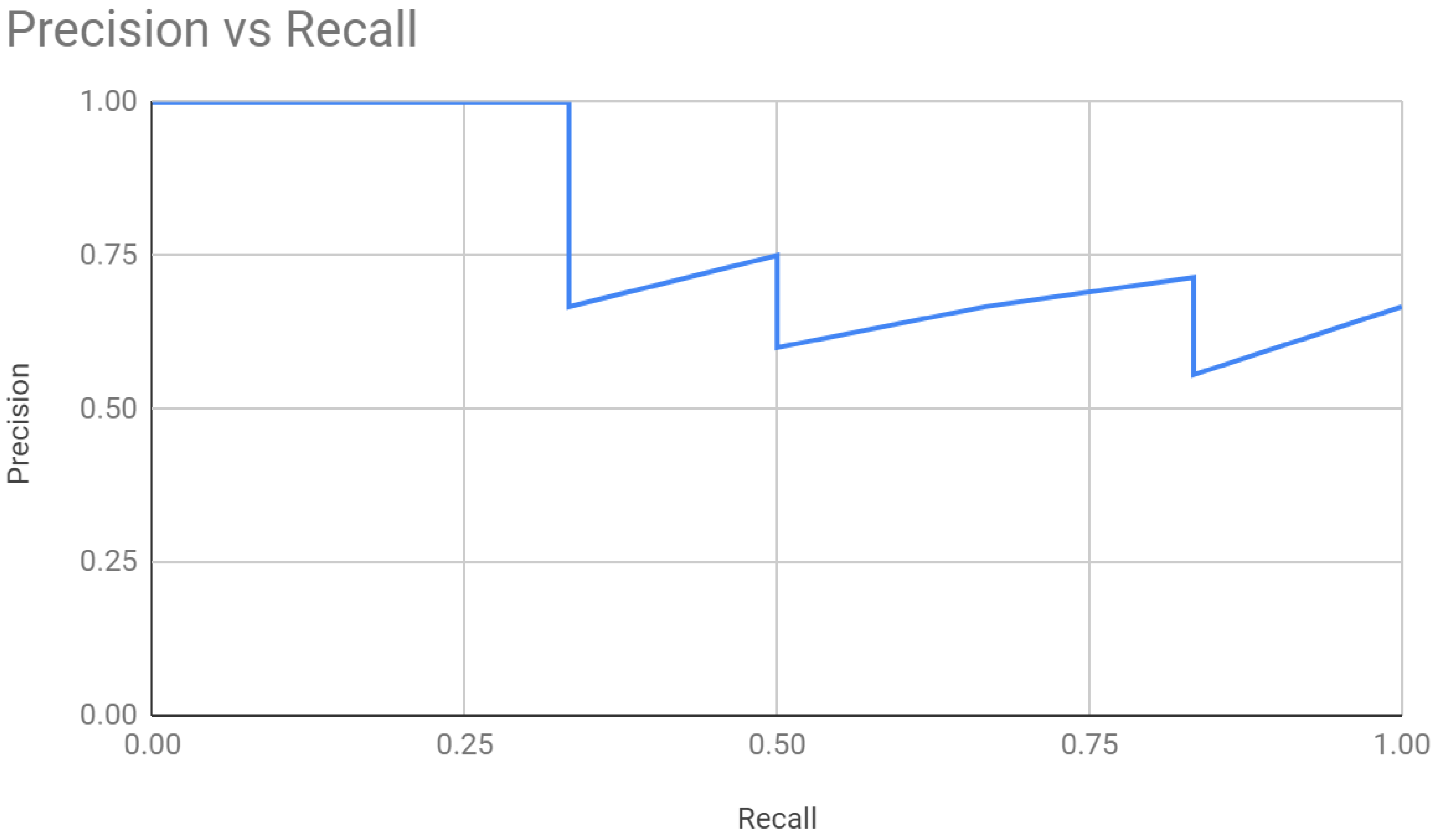
Step 4: Calculate precision and recall at every row

Step 5: Take the mean of **maximum precision at 11 recall values** {0.0, 0.1, … 1.0) to get AP

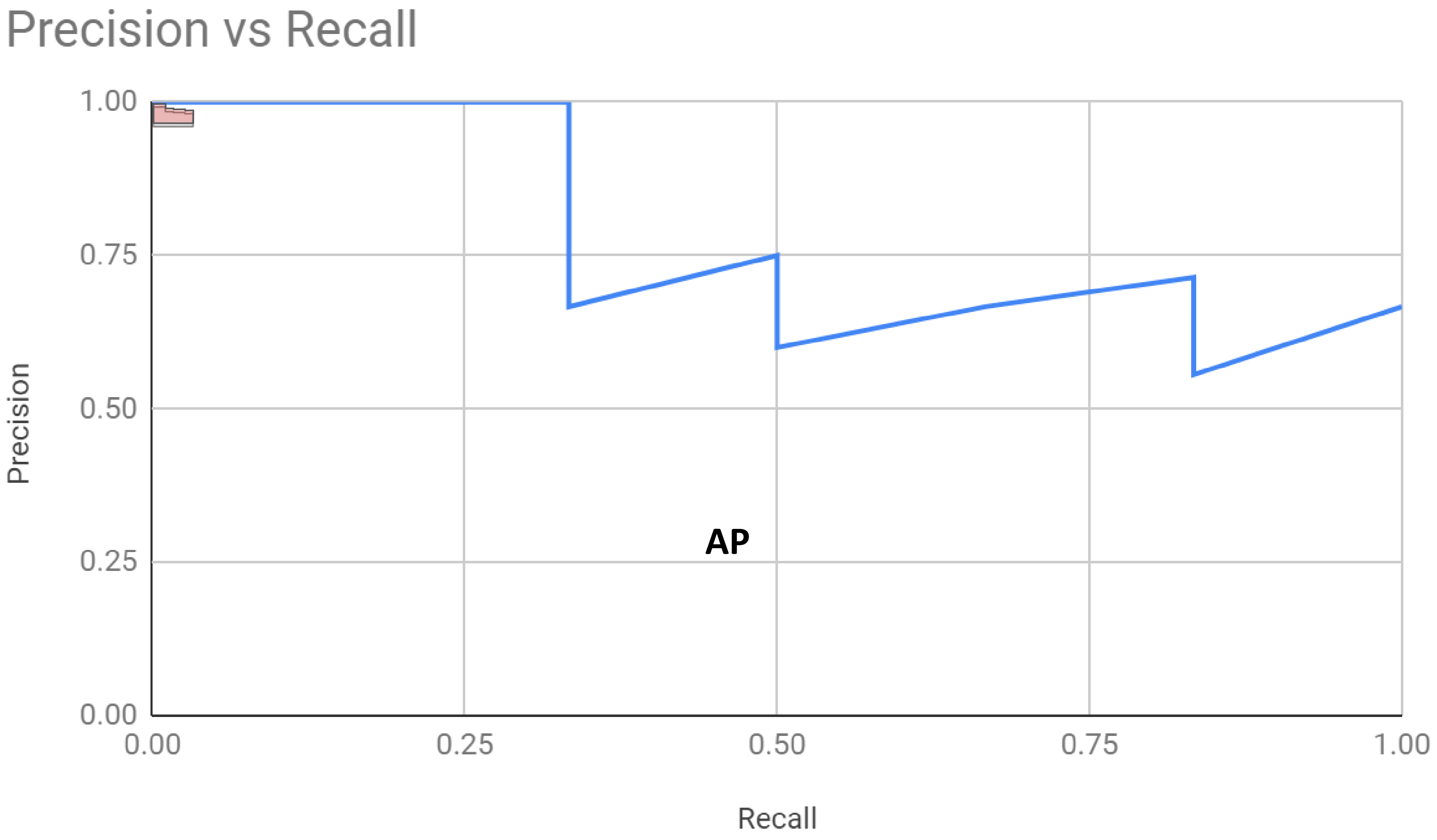
Step 6: Average across all classes to get the mAP score

Confidence	Rank	Correct	Precision	Recall
0.91	1	TRUE	1.00	0.17
0.87	2	TRUE	1.00	0.33
0.83	3	FALSE	0.67	0.33
0.81	4	TRUE	0.75	0.50
0.77	5	FALSE	0.60	0.50
0.65	6	TRUE	0.67	0.67
0.56	7	TRUE	0.71	0.83
0.40	8	FALSE	0.63	0.83
0.32	9	FALSE	0.56	0.83
0.31	10	TRUE	0.67	1.00

Precision and Recall curve



Precision and Recall curve

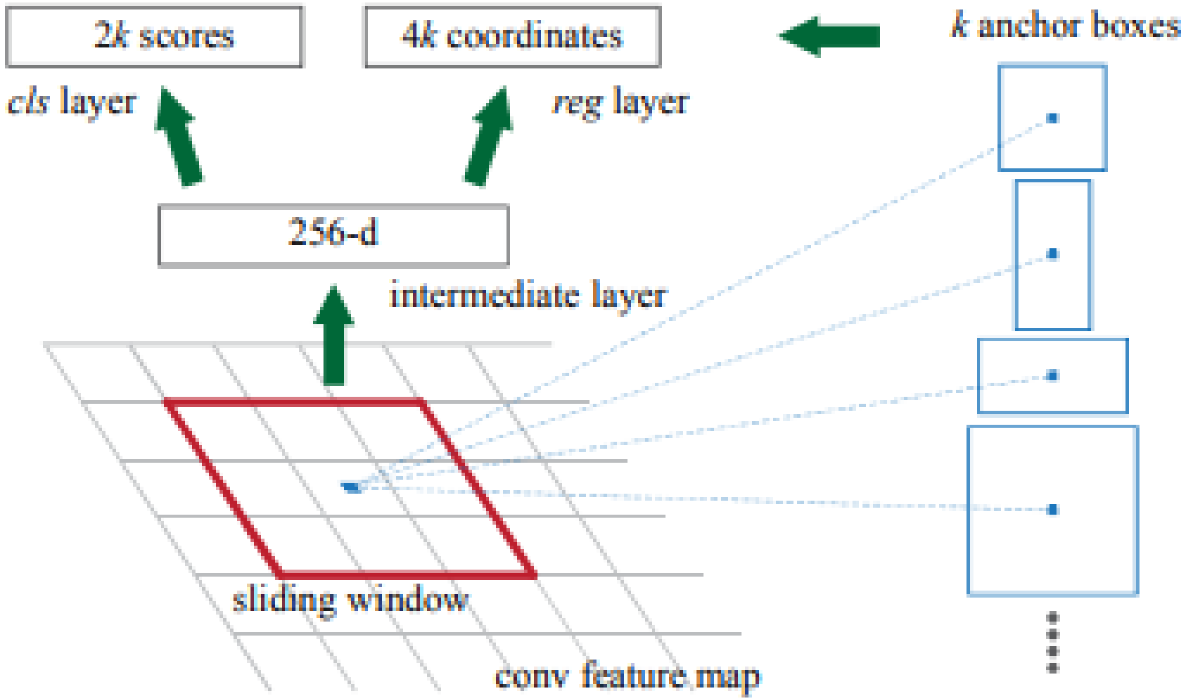


Anchor free box prediction

- Directly predict **(dx, dy, w, h)** at each grid cell
  - → Like YOLO (which does this 2x)
- Center of grid cell is center of bbox
- Harder problem, as output space has higher variance

Anchored box prediction

At each uniformly sampled grid cell, set of anchor boxes with different aspect ratios and scales



Faster R-CNN: For anchors, we use 3 scales with box areas of 128 , 256, and 512 pixels, and 3 aspect ratios of 1:1, 1:2, and 2:1.

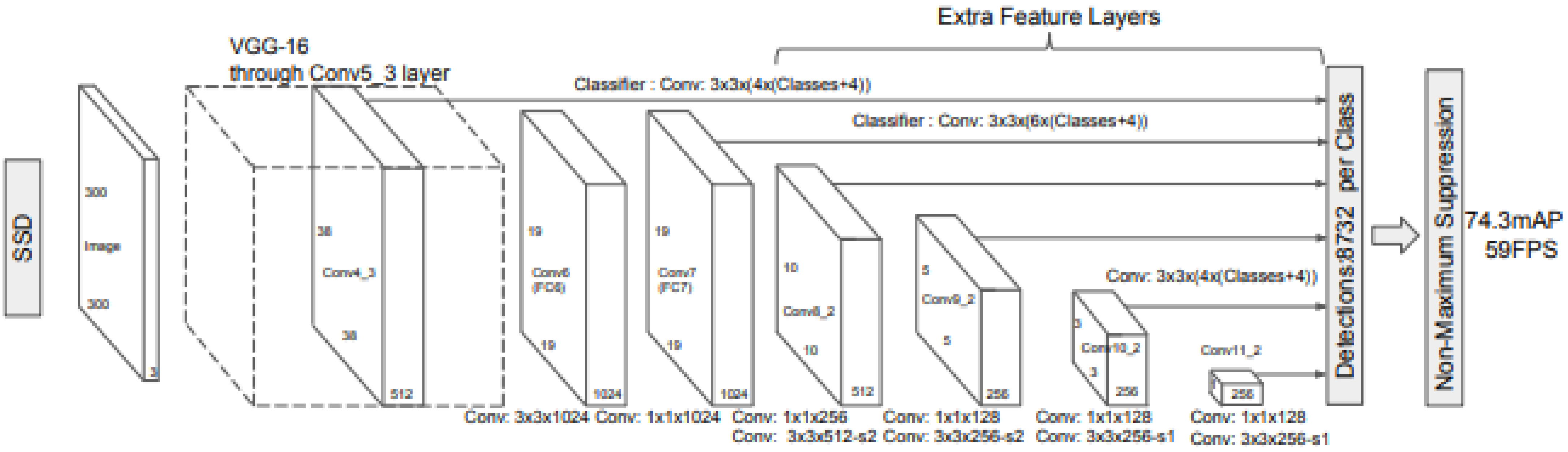
For each anchor box

- Regress to a final box with 4 numbers: ***(dx, dy, dw, dh)***
- Predict 2 objectness scores (including background as a separate class)

https://arxiv.org/abs/1506.01497

SSD: Single Shot Multibox Detector

Like YOLO, but predicts using a multi-scale pyramidal feature hierarchy

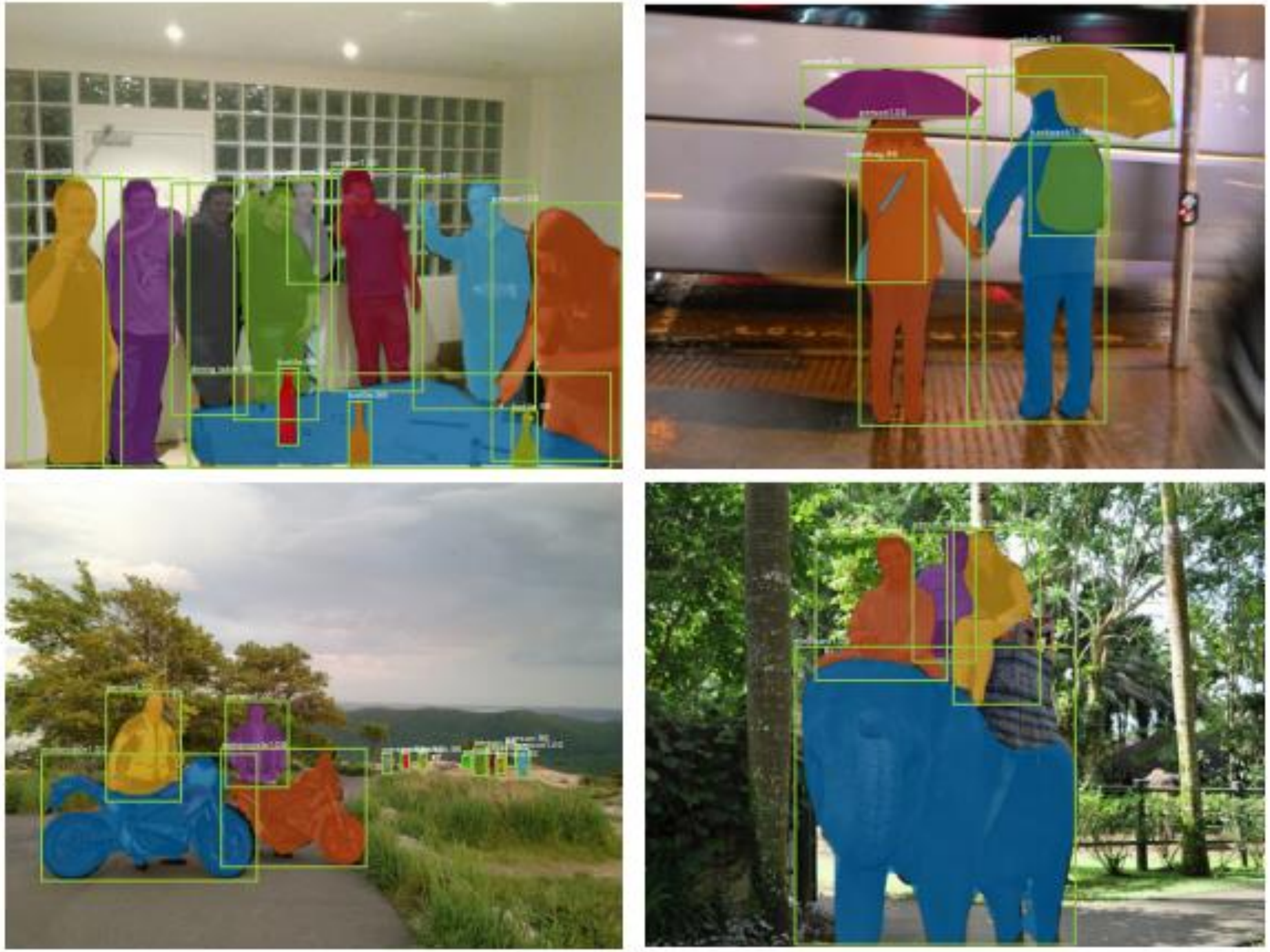


https://arxiv.org/abs/1512.02325

Why still use a two-stage object detector?

- Better recall of RPN as compared to SSD/YOLO
  - Trained with all object instances
  - Generic first stage, usable for multitask
- Finer control over training classifier
  - Custom minibatch (sampling 3:1 negative samples)
- Instance-level multitask (Mask-RCNN)

Mask R-CNN – towards instance-level understanding



https://arxiv.org/abs/1703.06870

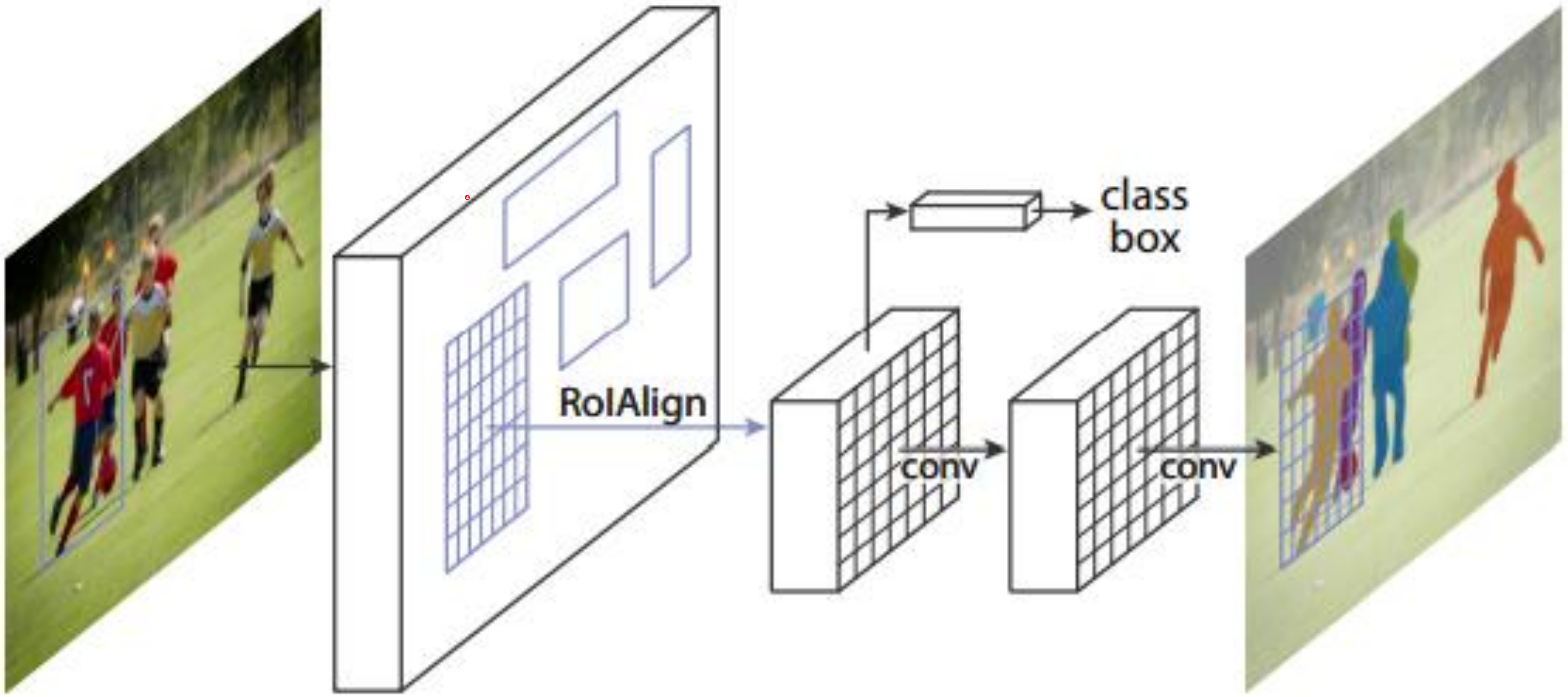
Mask R-CNN – towards instance-level understanding

Zoom in on instances

https://arxiv.org/abs/1703.06870

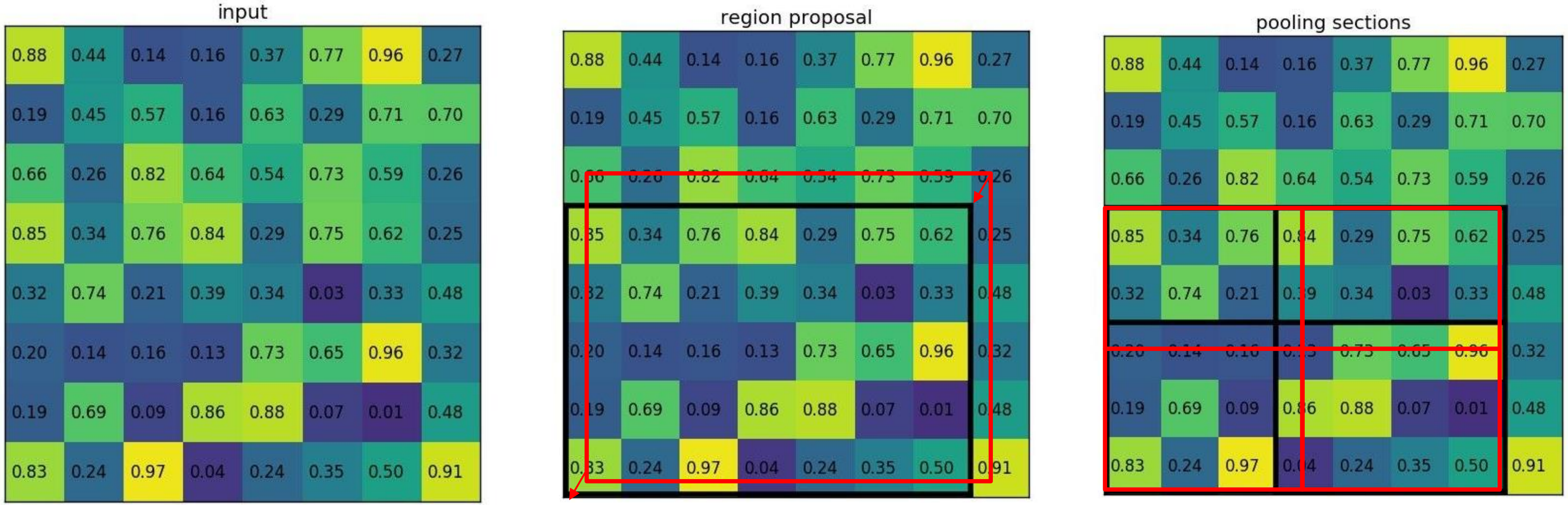
Mask R-CNN

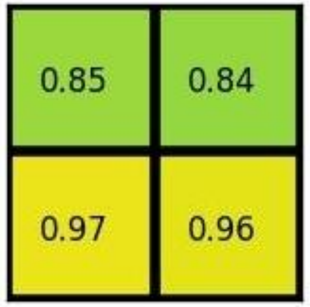
Preserves pixel-to-pixel



https://arxiv.org/abs/1703.06870

Fast R-CNN: ROI Pooling



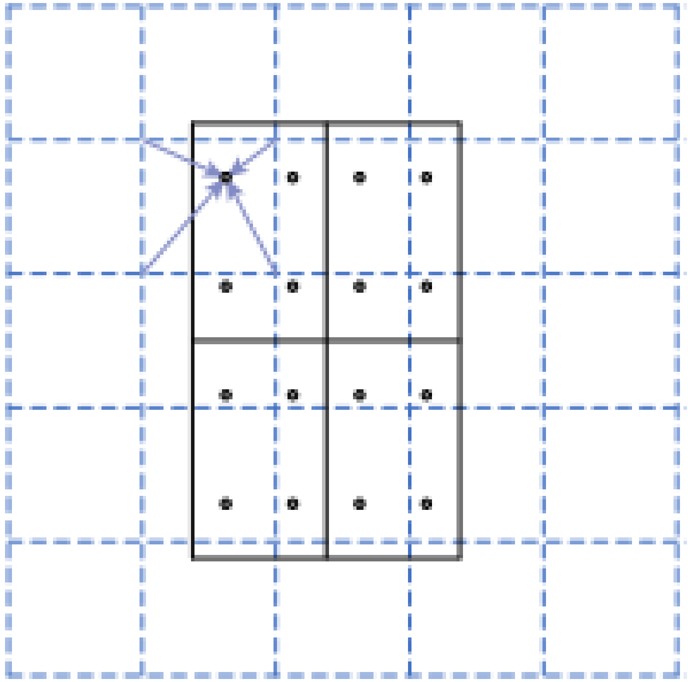


Double Quantization – loss of pixel-to-pixel alignment

https://deepsense.ai/region-of-interest-pooling-explained/

ROI Align

Improvement on ROI Pooling



- Input: Feature map (5x5 here) and region proposal (normalized float coordinates)
- Output: 2x2 ‘pooled’ bins
- Sample 4 points in every bin uniformly
- Compute value at each bin using bilinear interpolation
- Max or average the 4 bins

https://arxiv.org/abs/1703.06870

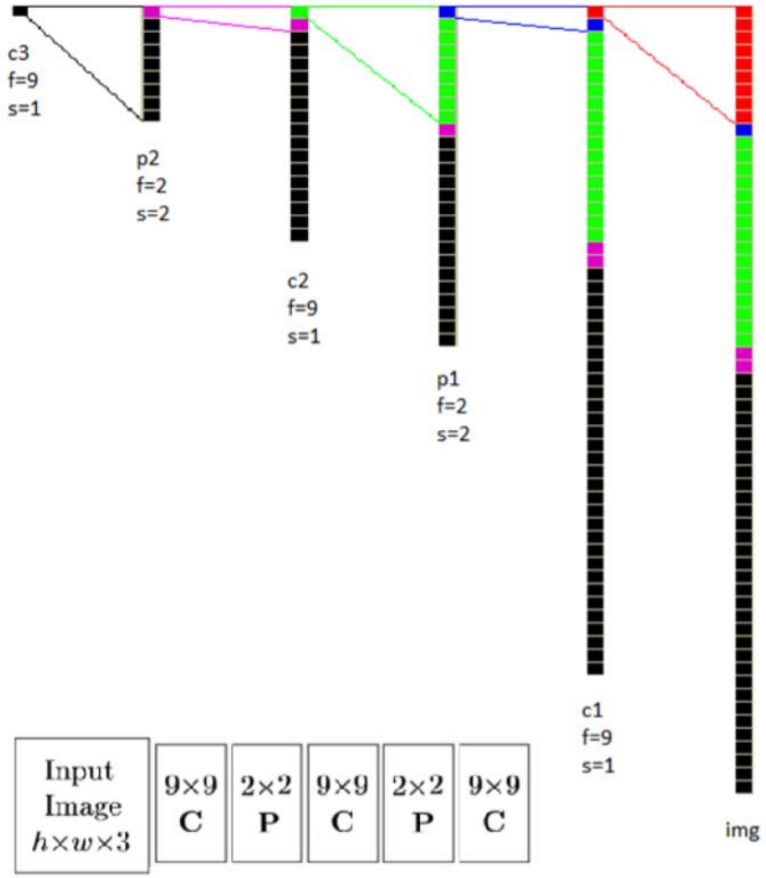
Effective Receptive Field Calculation

- Let $R_{k,j}$ be the effective receptive field of a neuron in layer k projected on layer j
- f_j, s_j respectively be filter size and stride of layer j

Writing $R_{k,j}$ in terms of $R_{k,j+1}$:

$$R_{k,j} = (R_{k,j+1} - 1)\, s_{j+1} + f_{j+1}$$

Finally, $R_{k,k} = 1$

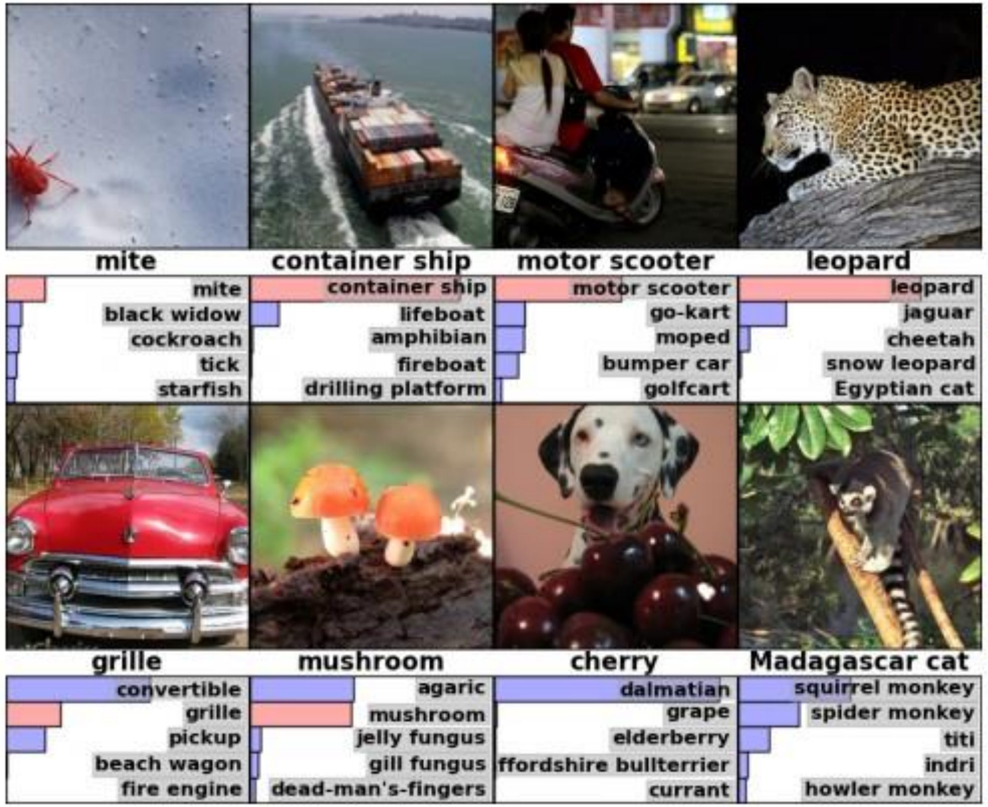


https://arxiv.org/pdf/1705.07049.pdf

ConvNets are Everywhere

ConvNets are everywhere

Classification

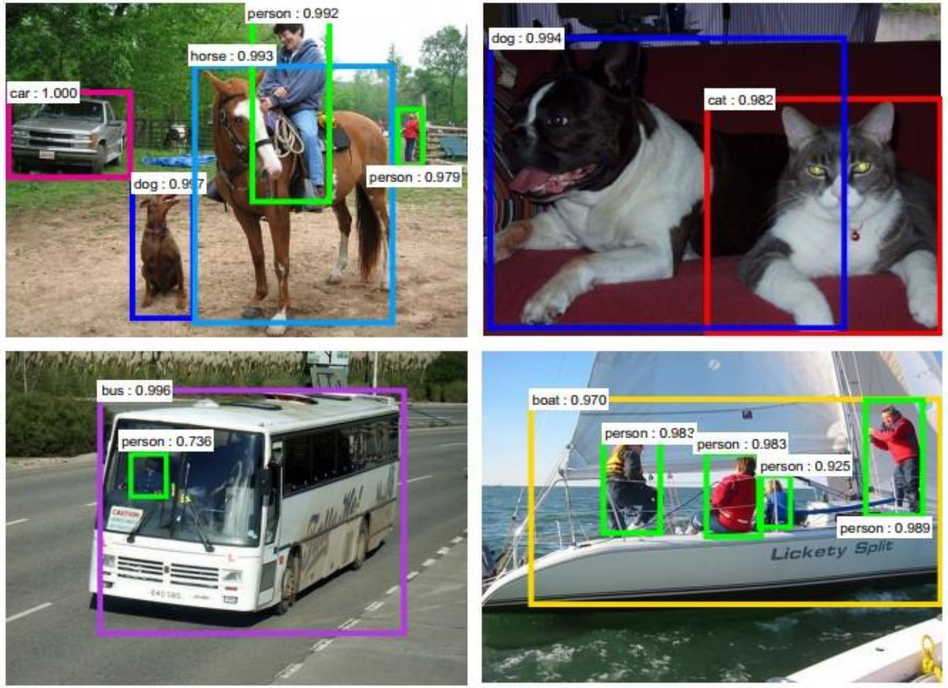


Retrieval

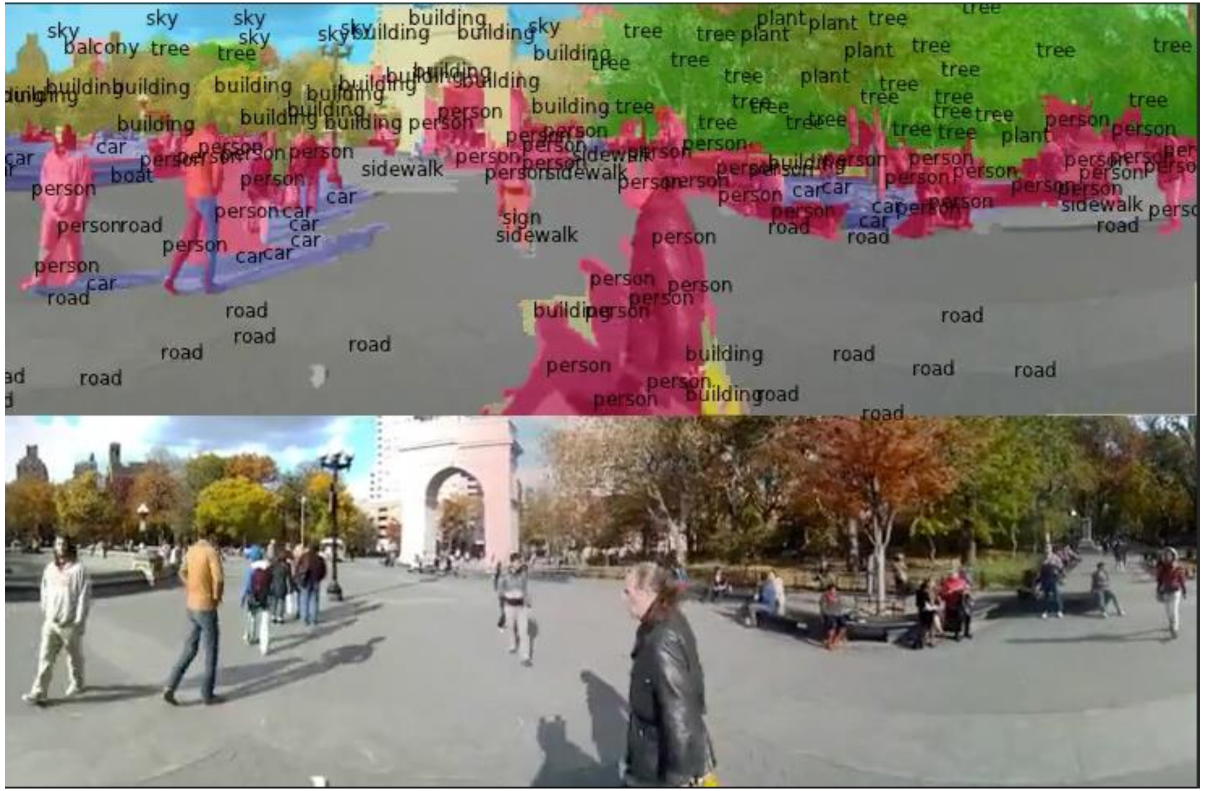
[Krizhevsky 2012]

ConvNets are everywhere

Detection

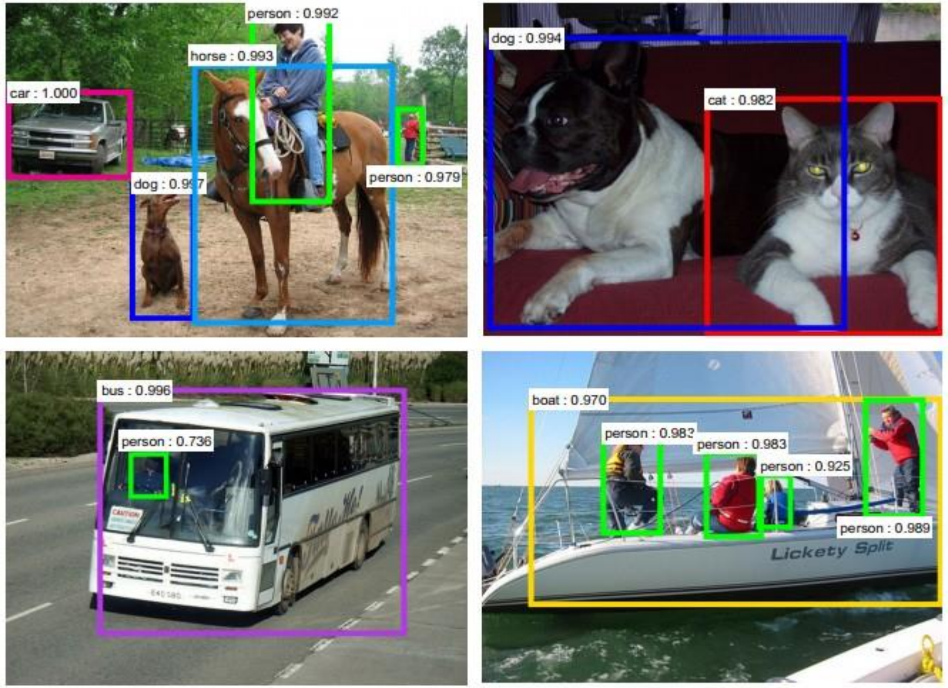


Segmentation



ConvNets are everywhere

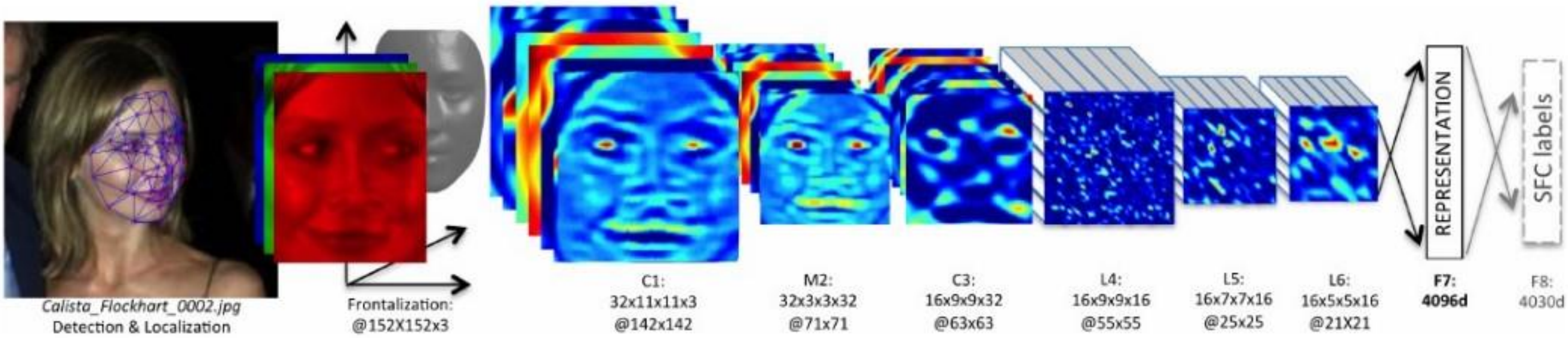
Detection

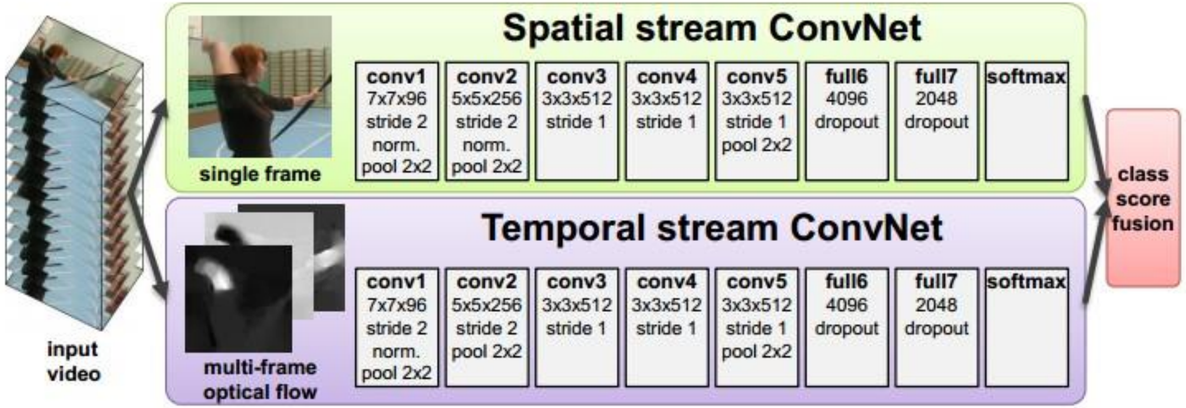


Instance Segmentation

ConvNets are everywhere

[Taigman et al. 2014]

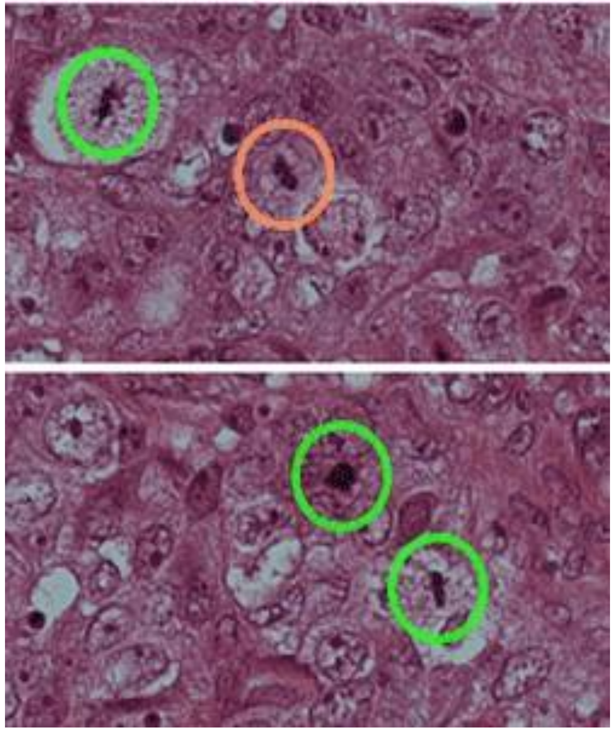




[Simonyan et al. 2014]

[Goodfellow 2014]

ConvNets are everywhere



[Ciresan et al. 2013]

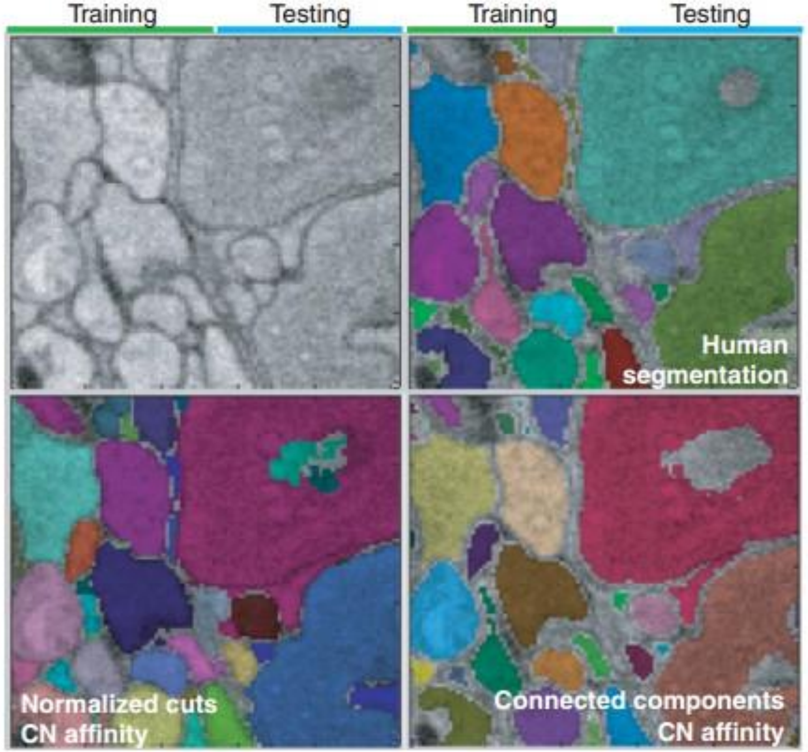
[Sermanet et al. 2011]
[Ciresan et al.]

ConvNets are everywhere

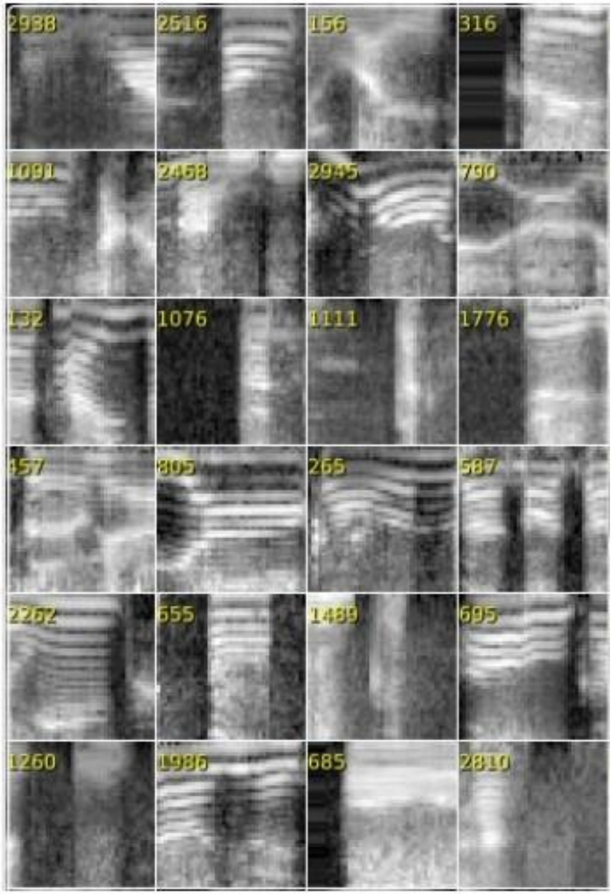
self-driving cars

NVIDIA Tegra X1

ConvNets are everywhere



[Turaga et al., 2010]



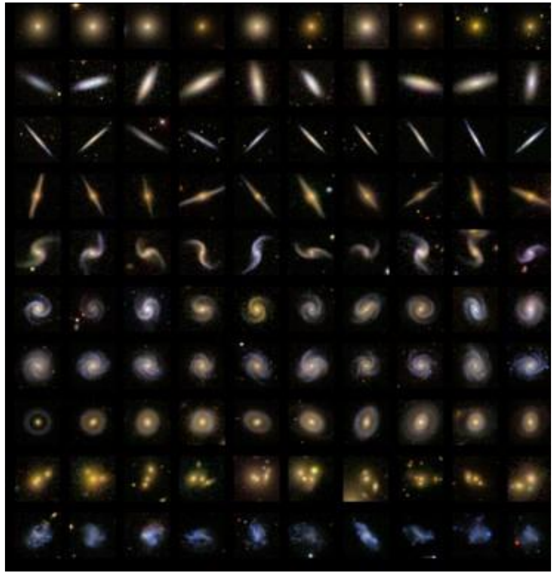
I caught this movie on the Sci-Fi channel recently. It actually turned out to be pretty decent as far as B-list horror/suspense films go. **Two guys (one naive and one loud mouthed a **) take a road trip to stop a wedding but have the worst possible luck when a maniac in a freaky, make-shift tank/truck hybrid decides to play cat-and-mouse with them.** Things are further complicated when they pick up a ridiculously whorish hitchhiker. What makes this film unique is that the combination of comedy and terror actually work in this movie, unlike so many others. The two guys are likable enough and there are some good chase/suspense scenes. Nice pacing and comic timing make this movie more than passable for the horror/slasher buff. **Definitely worth checking out.**

I just saw this on a local independent station in the New York City area. **The cast showed promise but when I saw the director, George Cosmotos, I became suspicious. And sure enough, it was every bit as bad, every bit as pointless and stupid as every George Cosmotos movie I ever saw.** He's like a stupid man's Michael Bey – with all the awfulness that accolade promises. There's no point to the conspiracy, no burning issues that urge the conspirators on. We are left to ourselves to connect the dots from one bit of graffiti on various walls in the film to the next. Thus, the current budget crisis, the war in Iraq, Islamic extremism, the fate of social security, 47 million Americans without health care, stagnating wages, and the death of the middle class are all subsumed by the sheer terror of graffiti. A truly, stunningly idiotic film.

Graphics is far from the best part of the game. **This is the number one best TH game in the series.** Next to Underground. **It deserves strong love. It is an insane game.** There are massive levels, massive unlockable characters... it's just a massive game. **Waste your money on this game. This is the kind of money that is wasted properly.** And even though graphics suck, thats doesn't make a game good. Actually, the graphics were good at the time. Today the graphics are crap. WHO CARES? As they say in Canada, This is the fun game, aye. (You get to go to Canada in THPS3) Well, I don't know if they say that, but they might. who knows. Well, Canadian people do. Wait a minute, I'm getting off topic. This game rocks. Buy it, play it, enjoy it, love it. It's PURE BRILLIANCE.

The first was good and original. I was a not bad horror/comedy movie. So I heard a second one was made and I had to watch it . What really makes this movie work is Judd Nelson's character and the sometimes clever script. **A pretty good script for a person who wrote the Final Destination films and the direction was okay.** Sometimes there's scenes where it looks like it was filmed using a home video camera with a grainy - look. Great made - for - TV movie. **It was worth the rental and probably worth buying just to get that nice eerie feeling and watch Judd Nelson's Stanley doing what he does best.** I suggest newcomers to watch the first one before watching the sequel, just so you'll have an idea what Stanley is like and get a little history background.

[Denil et al. 2014]

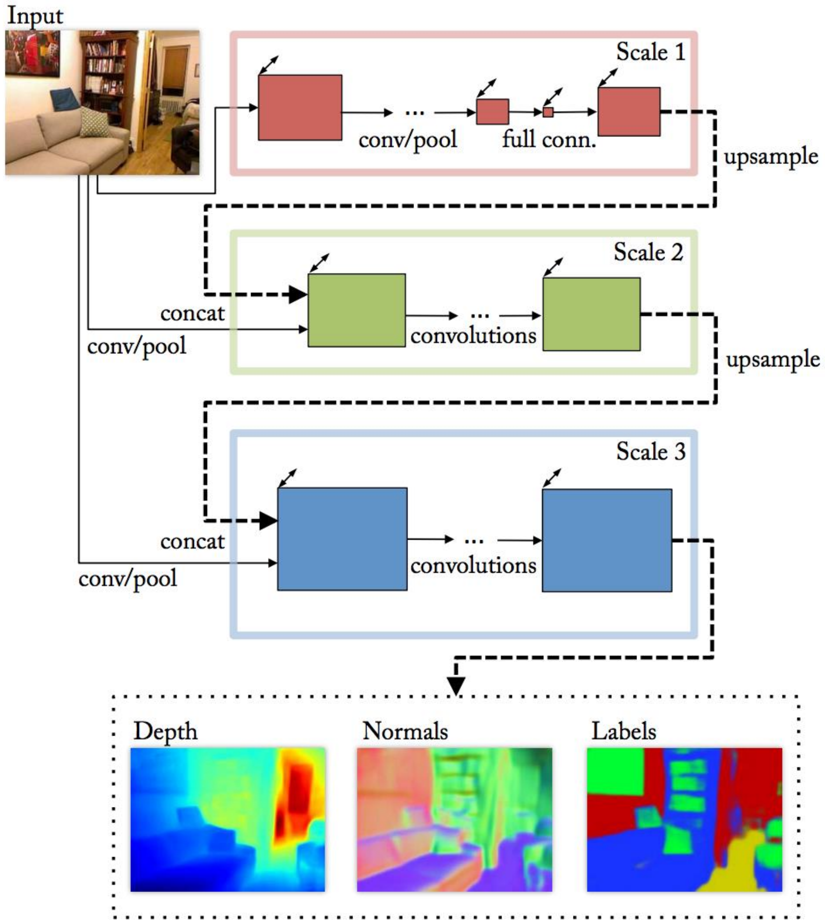


ConvNets are everywhere

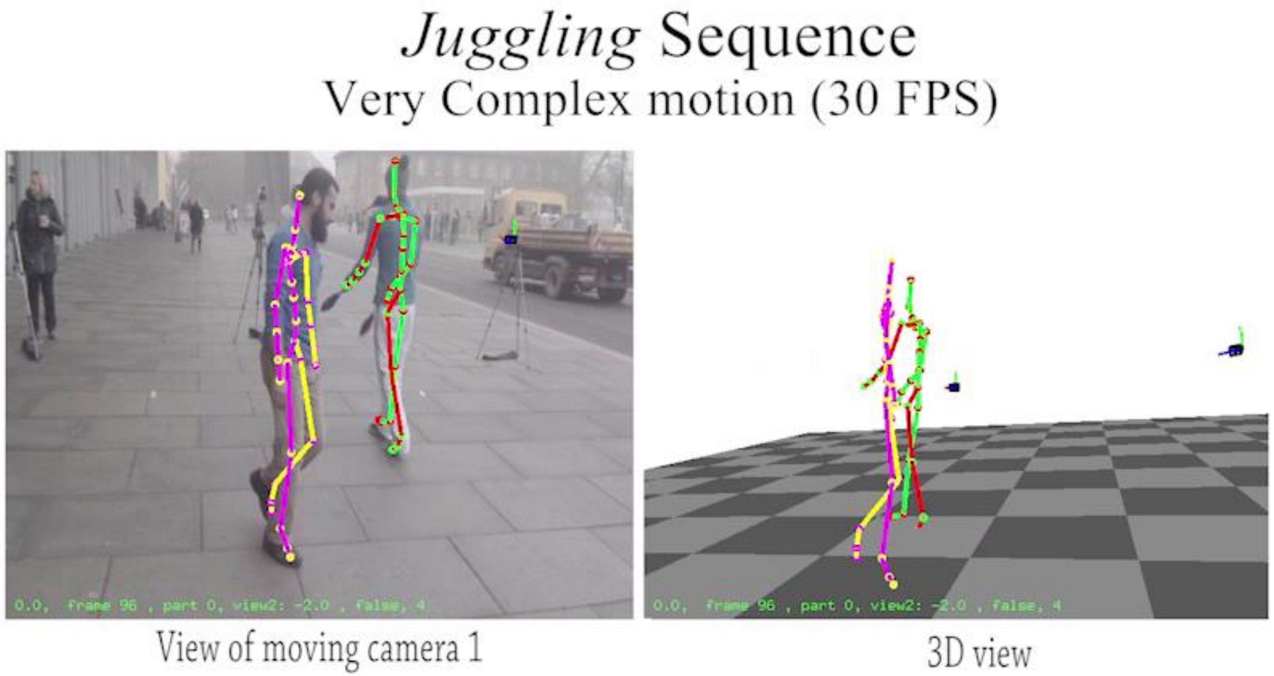
Whale recognition, Kaggle Challenge

Mnih and Hinton, 2010

ConvNets are everywhere

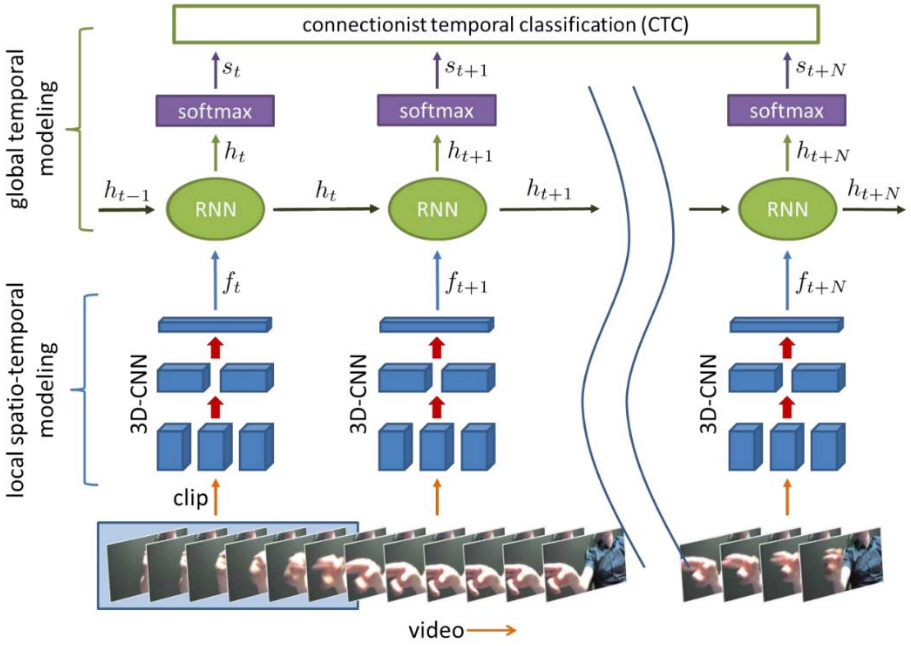


David Eigen and Rob Fergus, 2015

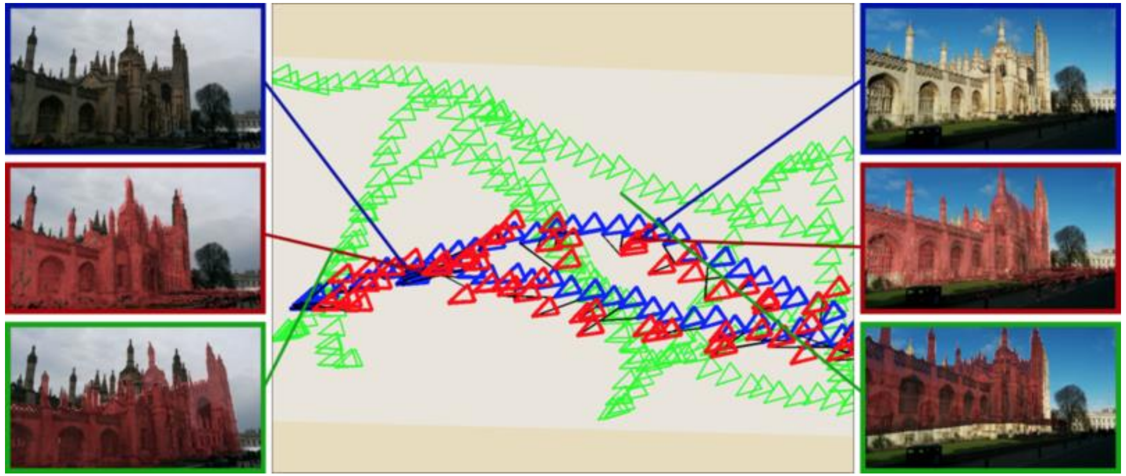


Elhayek et al., 2016

ConvNets are everywhere



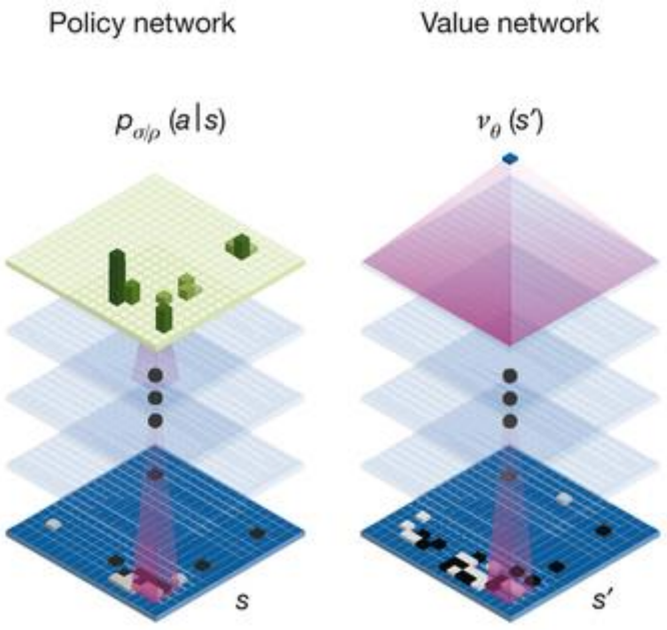
Molchanov et al., 2016



Kendall et al., 2016

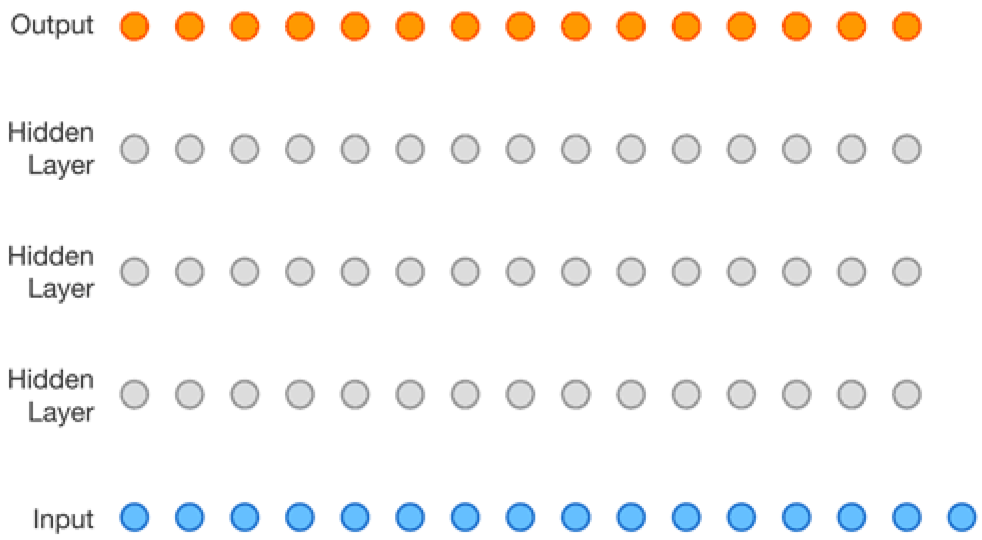
ConvNets are everywhere

Reinforcement Learning



AlphaGo, 2016
(Go Board treated as an image)

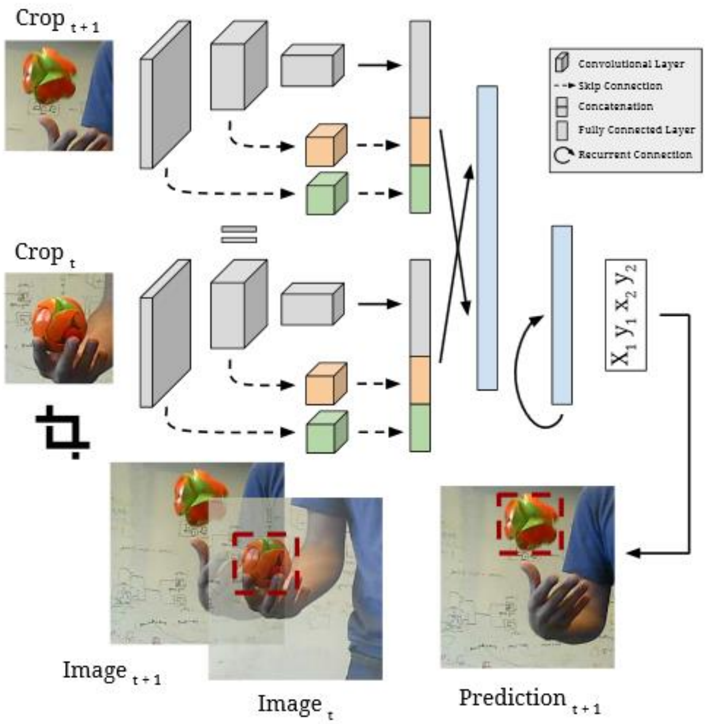
Automatic Speech Recognition



Wavenet, 2016
(Dilated 1D Convolutions)

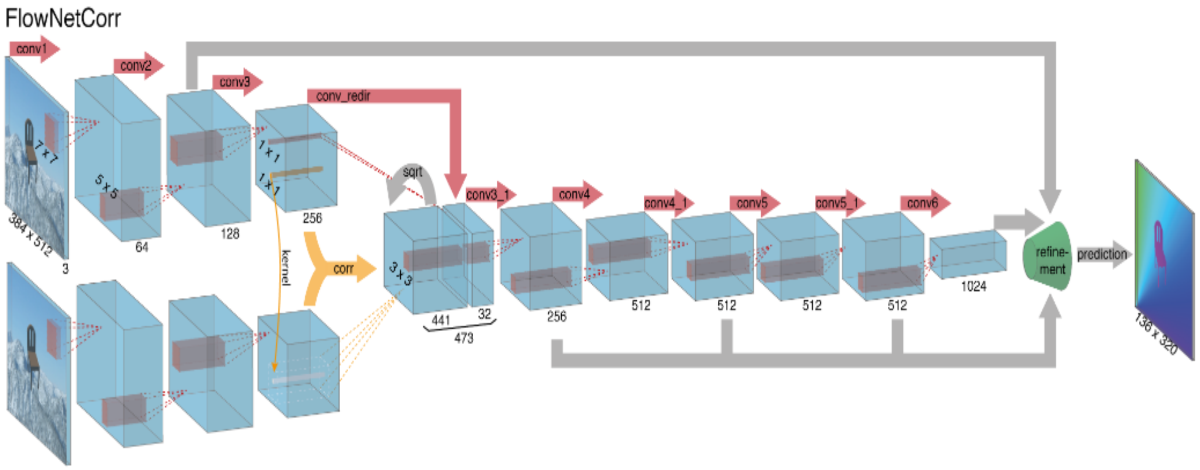
ConvNets are everywhere

Object Tracking



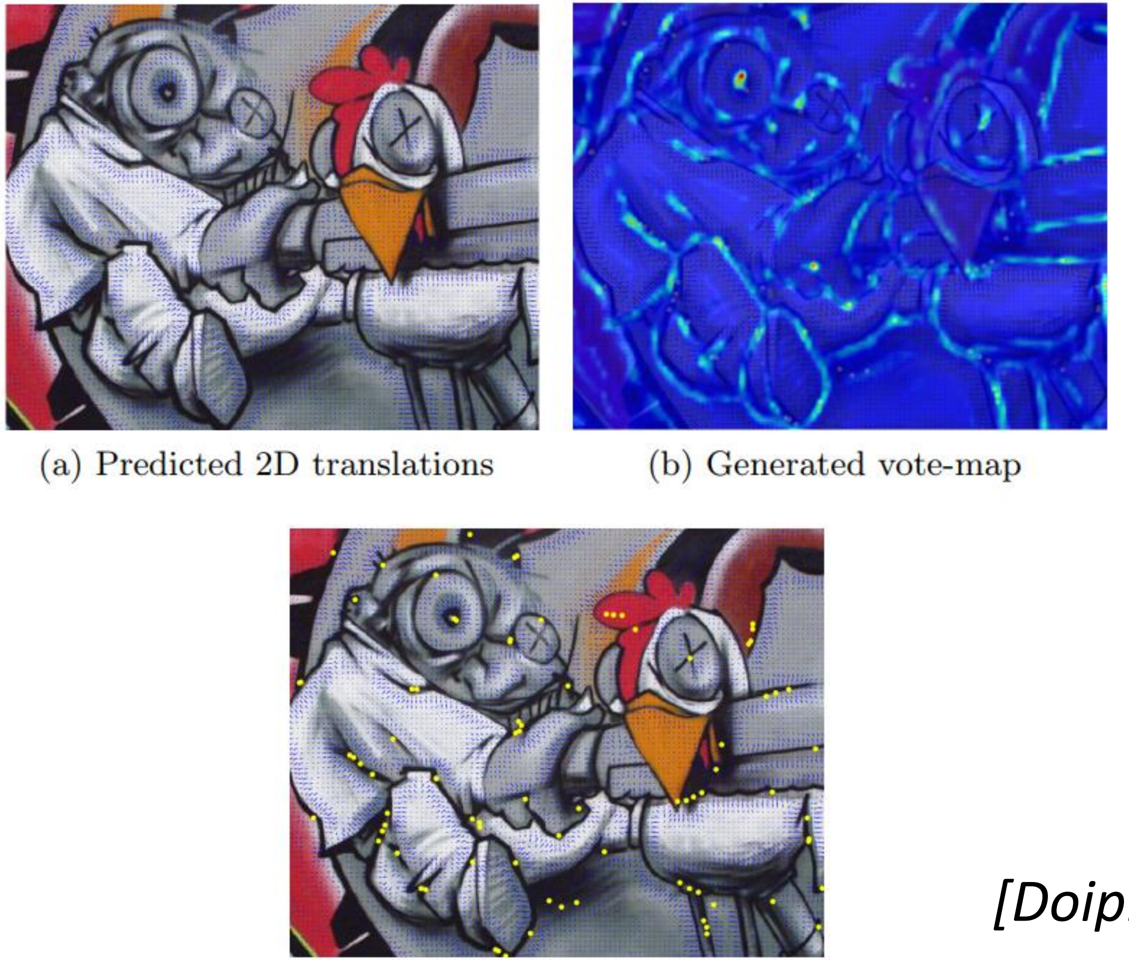
RE3, 2017

Optical Flow



FlowNet, 2015

ConvNets are everywhere



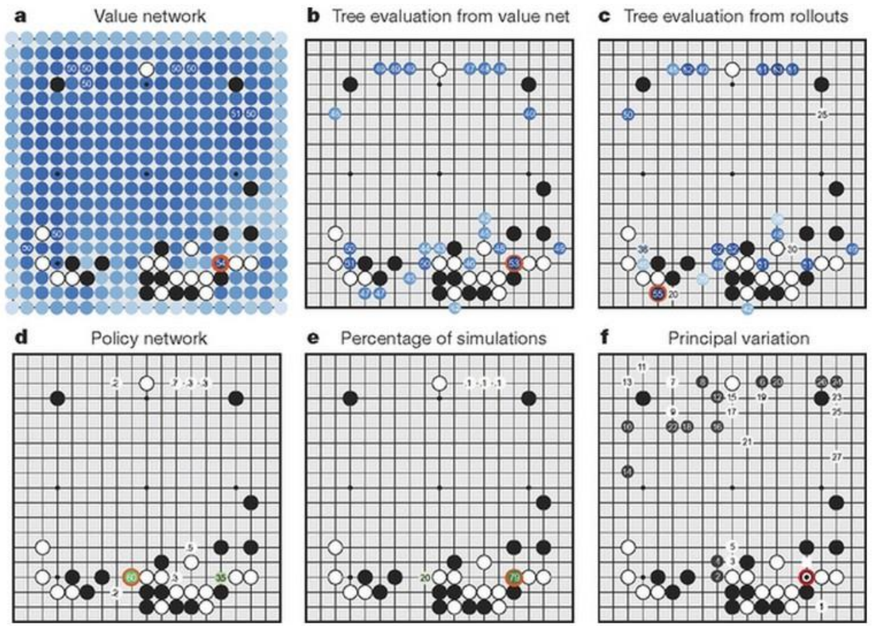
(a) Predicted 2D translations

(b) Generated vote-map

(c) Final extracted points

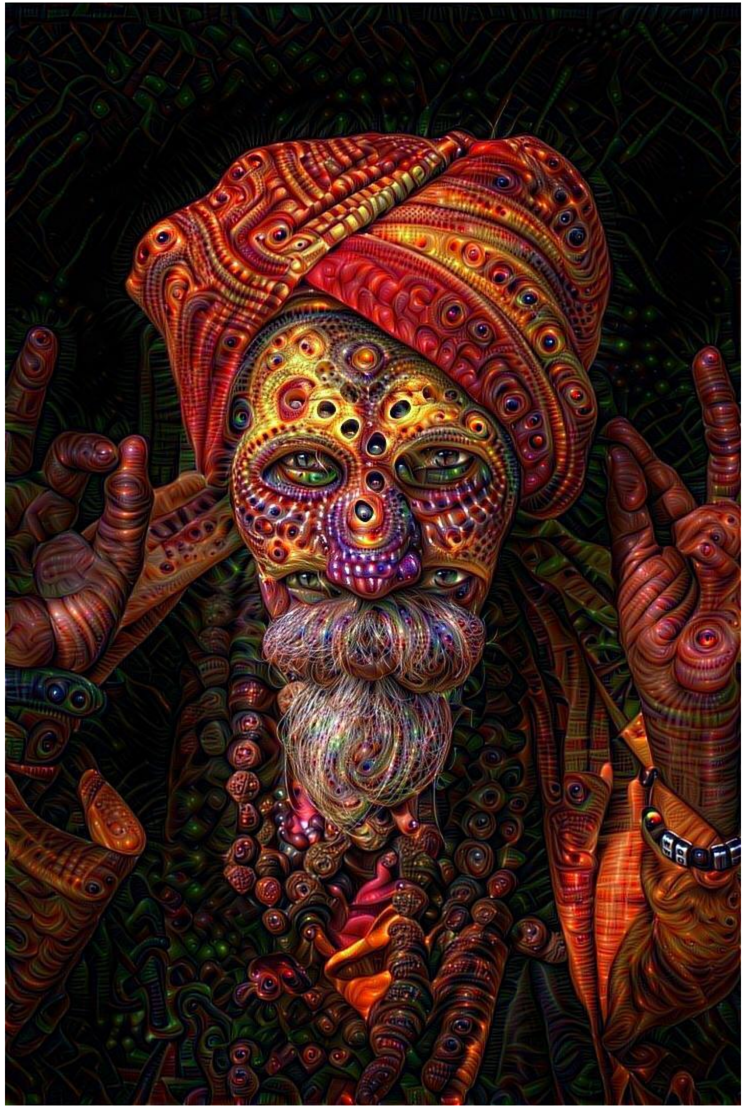
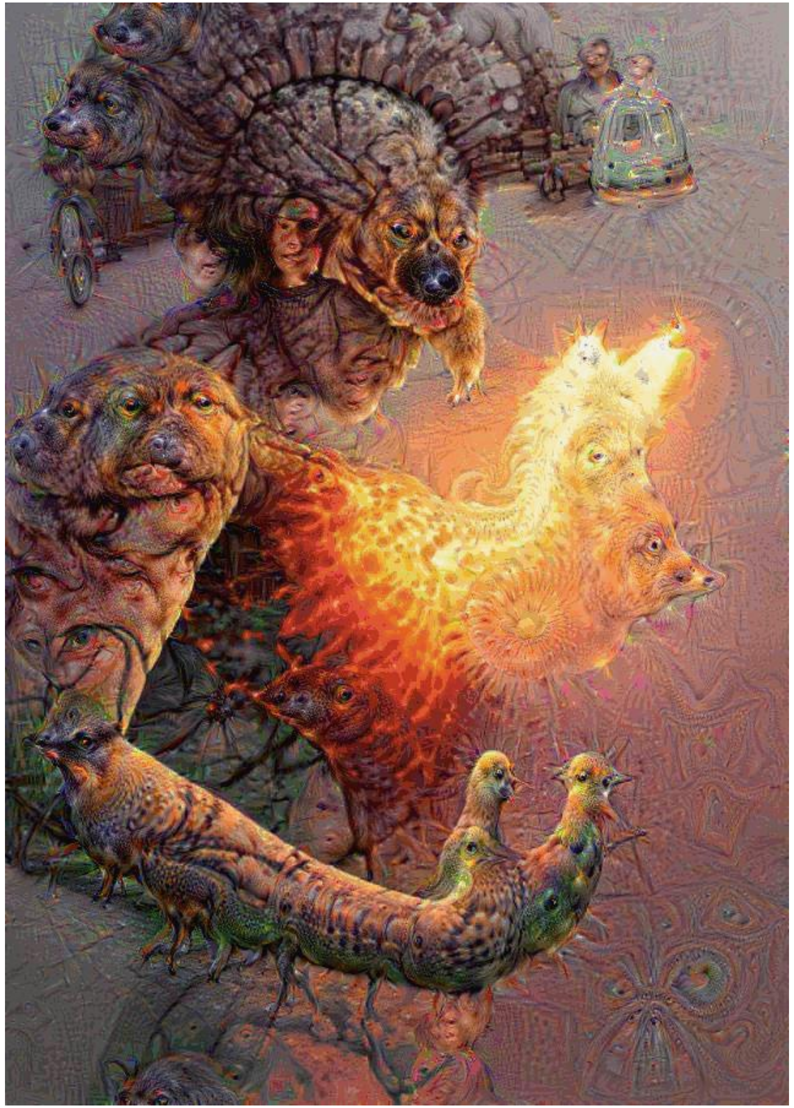
[Doiphode et al.]

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)*

reddit.com/r/deepdream

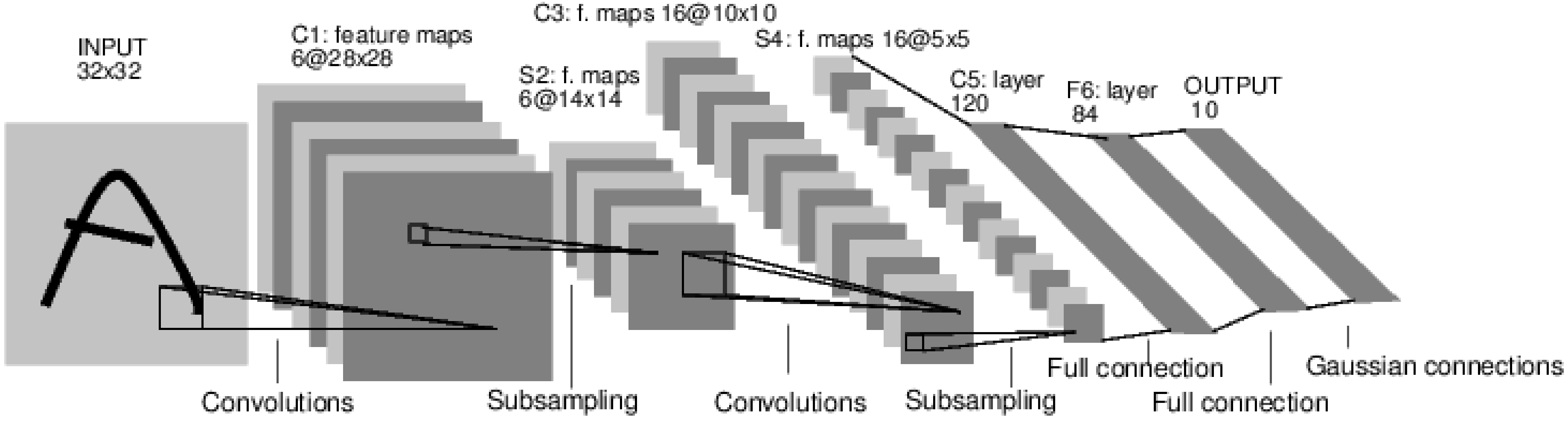
ConvNets are everywhere

TLDR

A Few CNN Case Studies

Case Study: LeNet-5

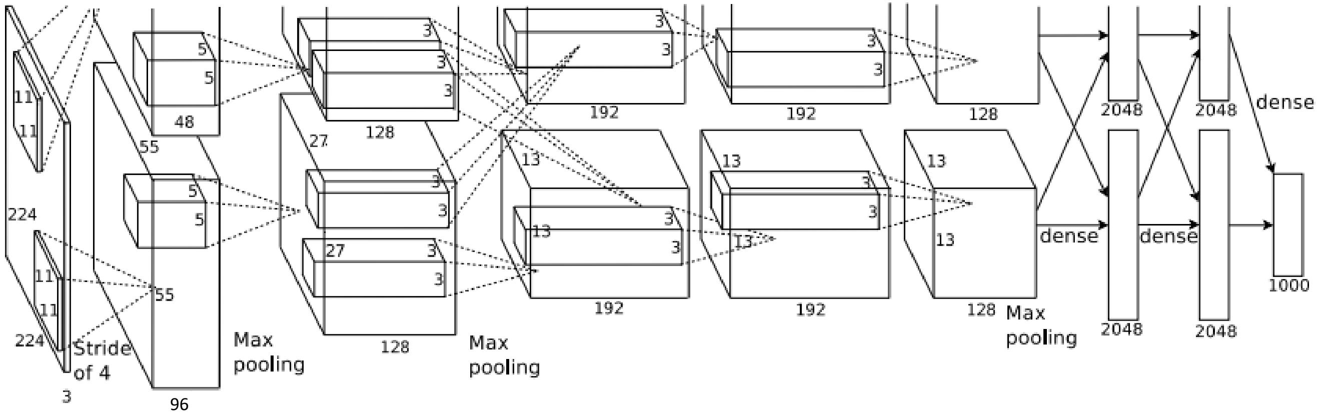
[LeCun et al., 1998]



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]

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4

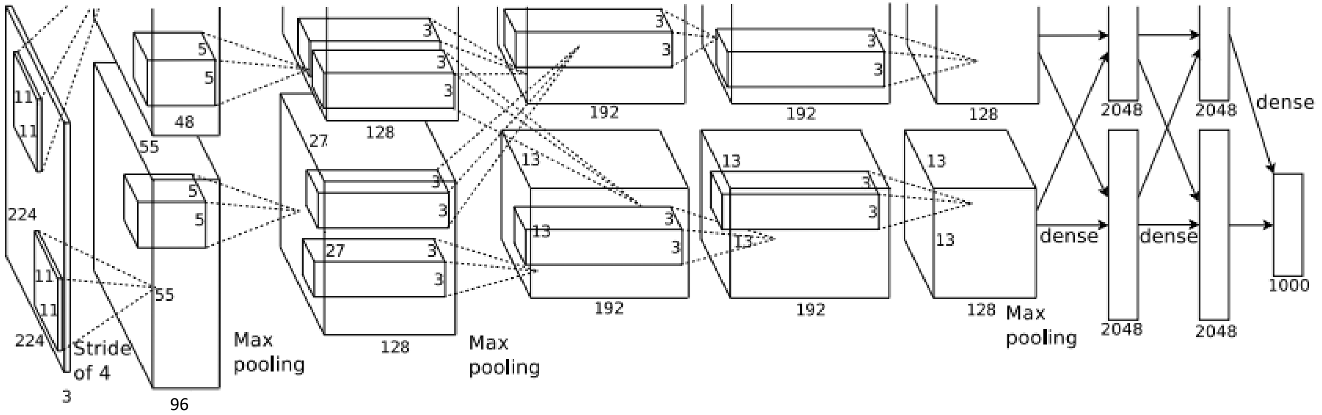
=>

Q: what is the output volume size?

ILVRC 2012

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

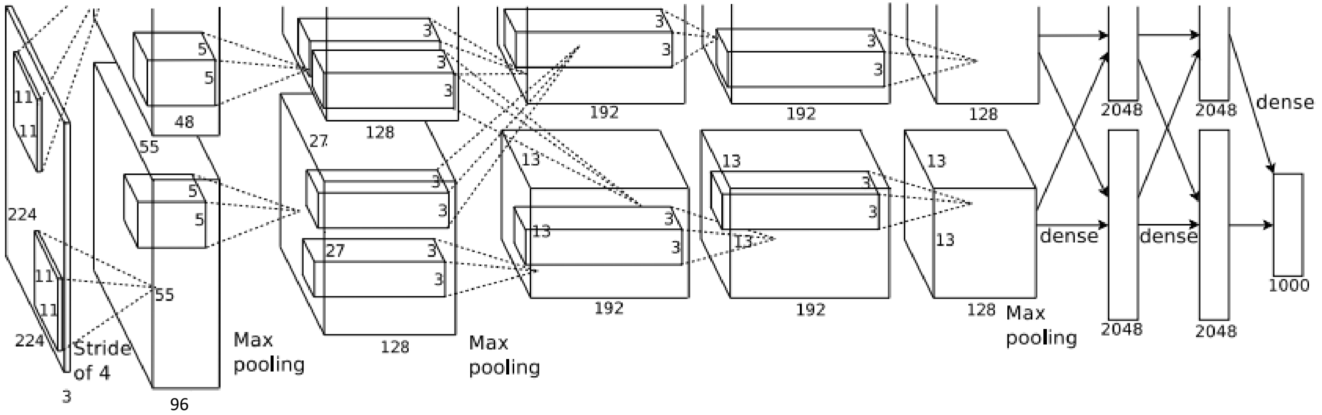
First layer (CONV1): 96 11x11 filters applied at stride 4

=>

Q: what is the output volume size? (227-11)/4+1 = 55

Case Study: AlexNet

[Krizhevsky et al. 2012]



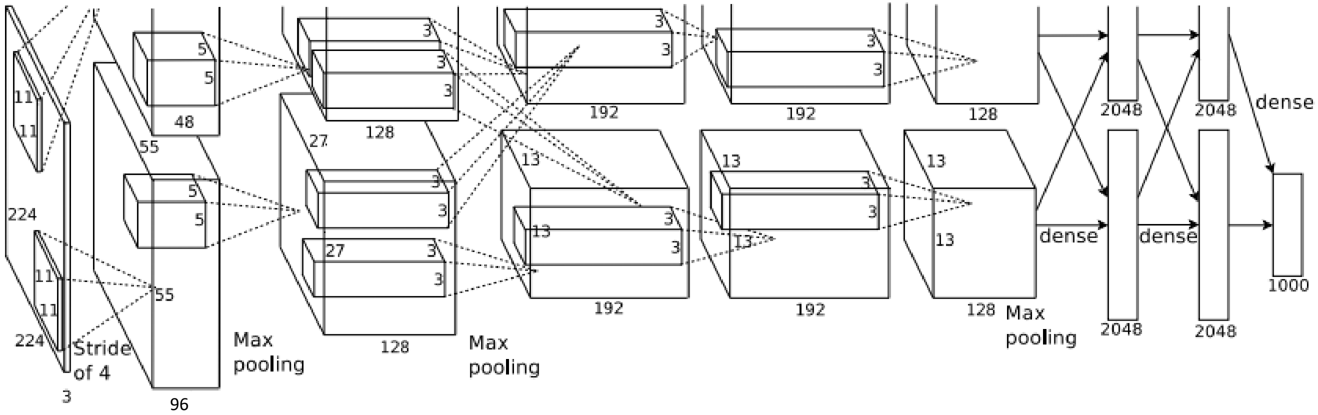
Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4
=>
Output volume **[55x55x96]**

Q: What is the total number of parameters in this layer?

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4

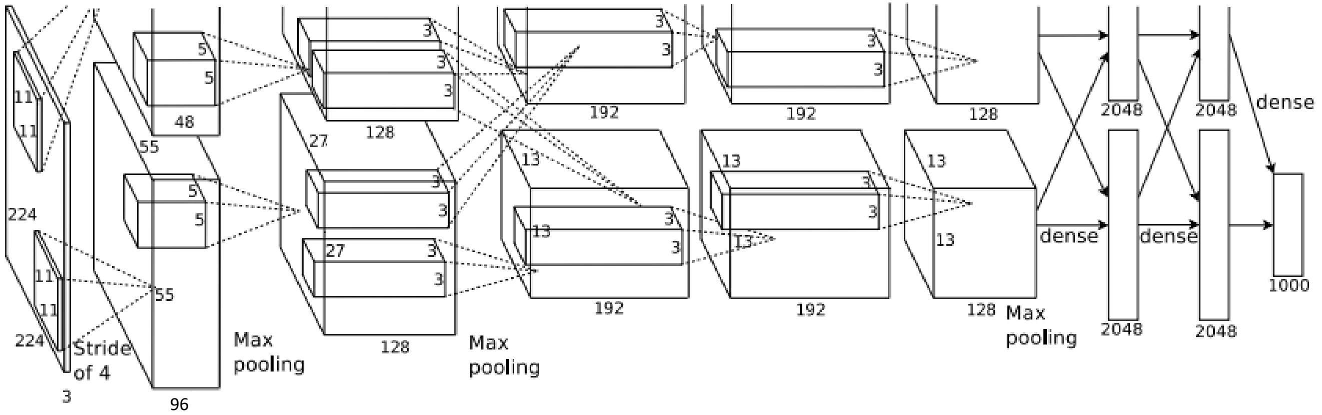
=>

Output volume **[55x55x96]**

Parameters: (11*11*3)*96 = **35K**

Case Study: AlexNet

[Krizhevsky et al. 2012]



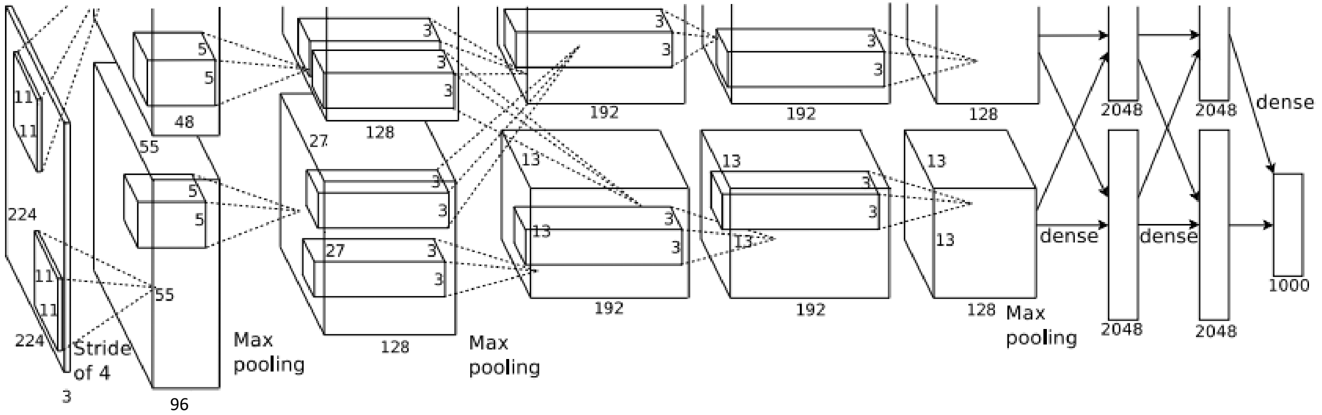
Input: 227x227x3 images
After CONV1: 55x55x96

Second layer (POOL1): 3x3 filters applied at stride 2

Q: what is the output volume size? Hint: (55-3)/2+1 = 27

Case Study: AlexNet

[Krizhevsky et al. 2012]



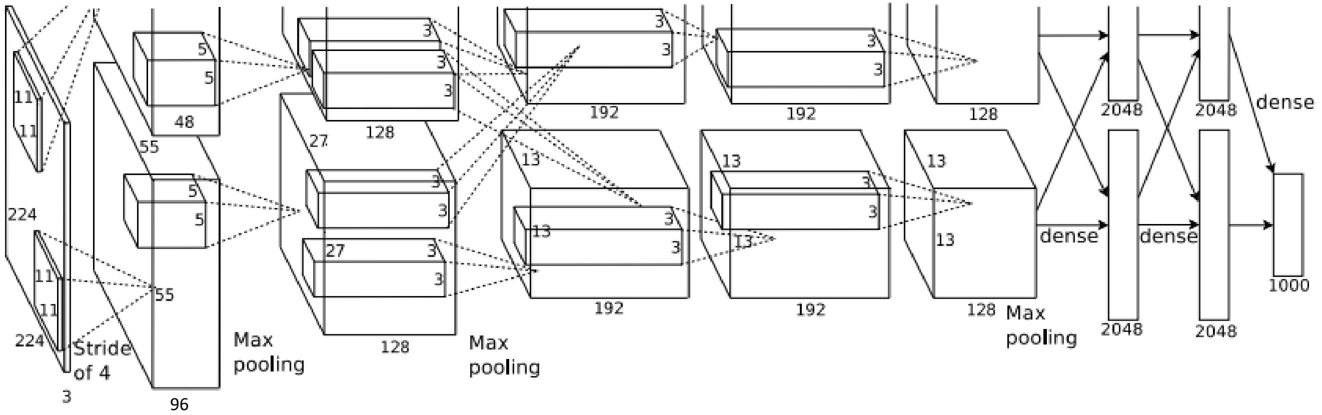
Input: 227x227x3 images
After CONV1: 55x55x96

Second layer (POOL1): 3x3 filters applied at stride 2
Output volume: 27x27x96

Q: what is the number of parameters in this layer?

Case Study: AlexNet

[Krizhevsky et al. 2012]

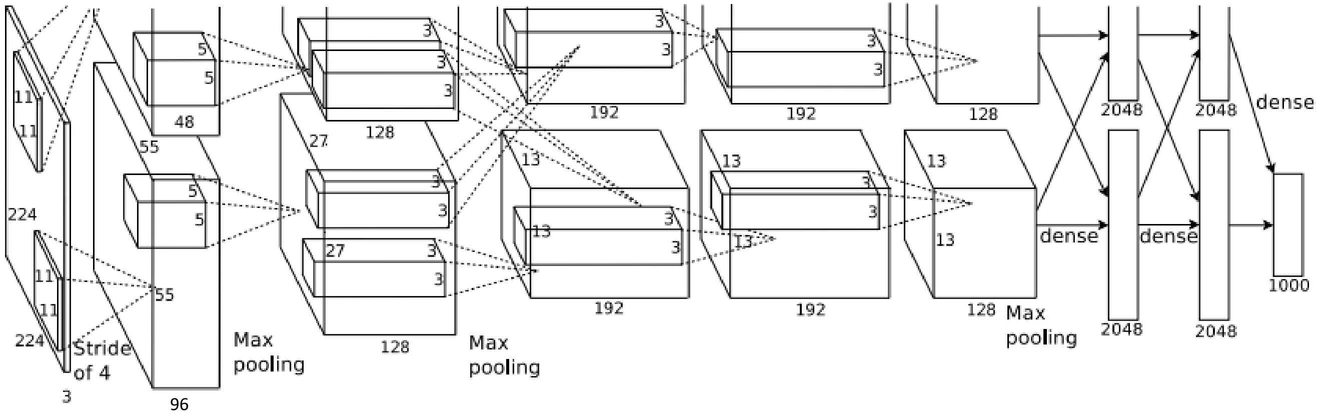


Input: 227x227x3 images
After CONV1: 55x55x96

Second layer (POOL1): 3x3 filters applied at stride 2
Output volume: 27x27x96
Parameters: 0!

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

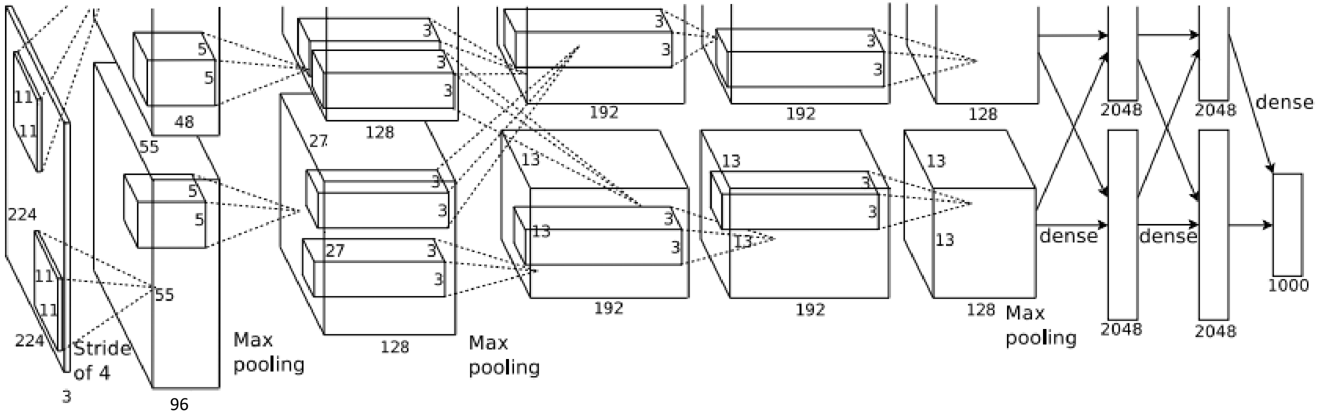
After CONV1: 55x55x96

After POOL1: 27x27x96

...

Case Study: AlexNet

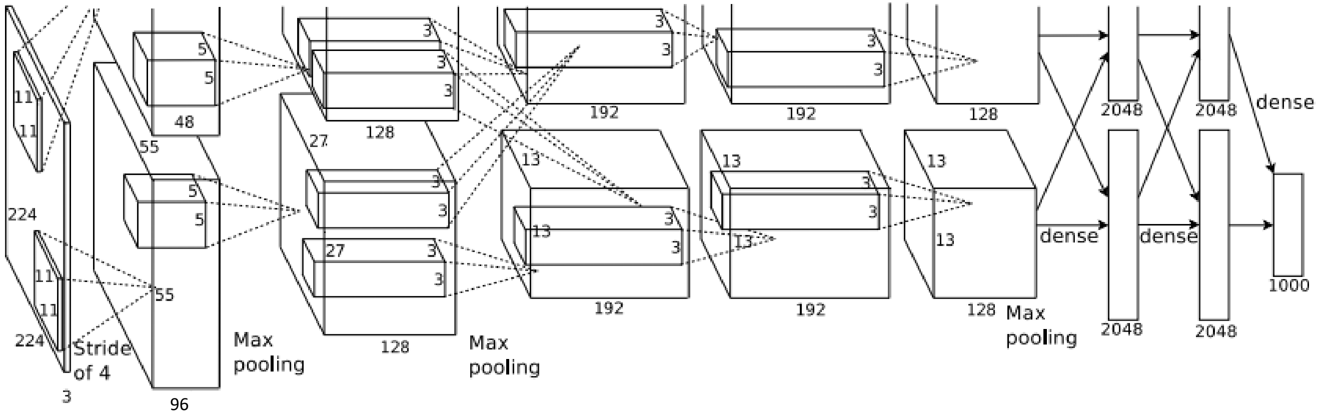
[Krizhevsky et al. 2012]



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)

Case Study: AlexNet

[Krizhevsky et al. 2012]

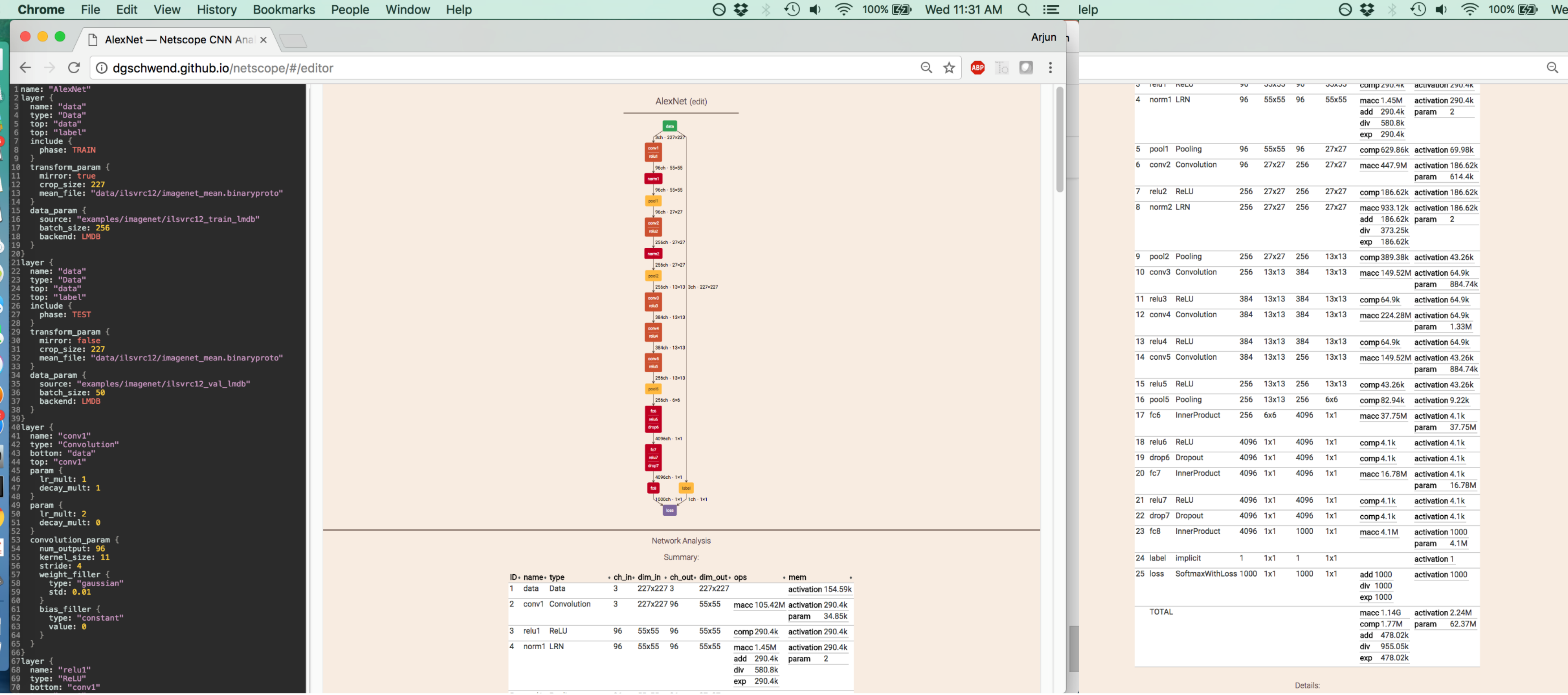


Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] 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:
- Popularized use of ReLU in Vision
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5 in only last few fully-connected
- 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%

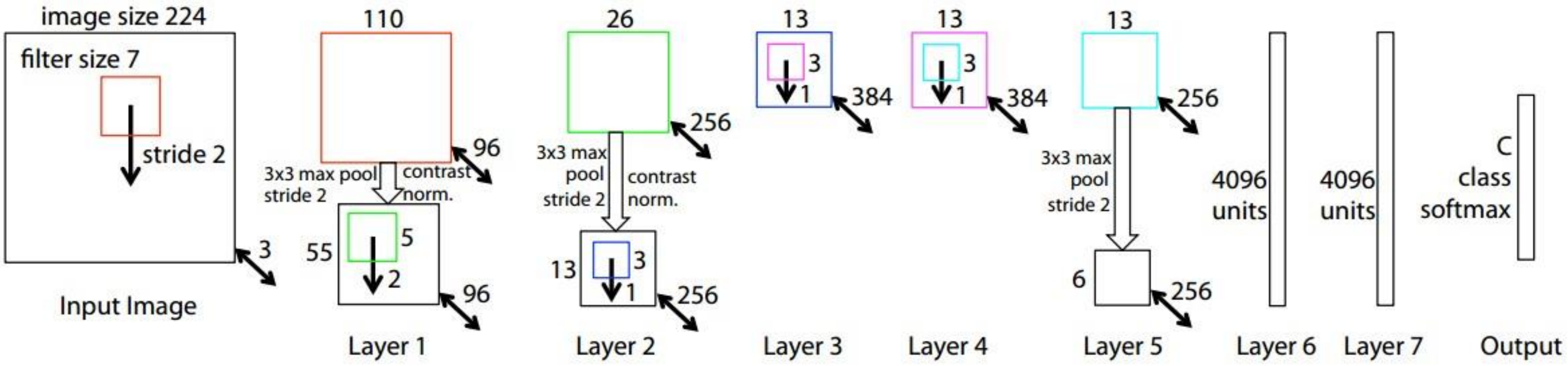
Aside: Useful Tool

http://dgschwend.github.io/netscope/#/editor



Case Study: ZFNet

[Zeiler and Fergus, 2013]



ILVRC 2013

AlexNet but:

CONV1: change from (11x11 stride 4) to (7x7 stride 2)

CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512

ImageNet top 5 error: 15.4% -> 14.8 (and later 11.2)%

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 (did not win!)

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

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

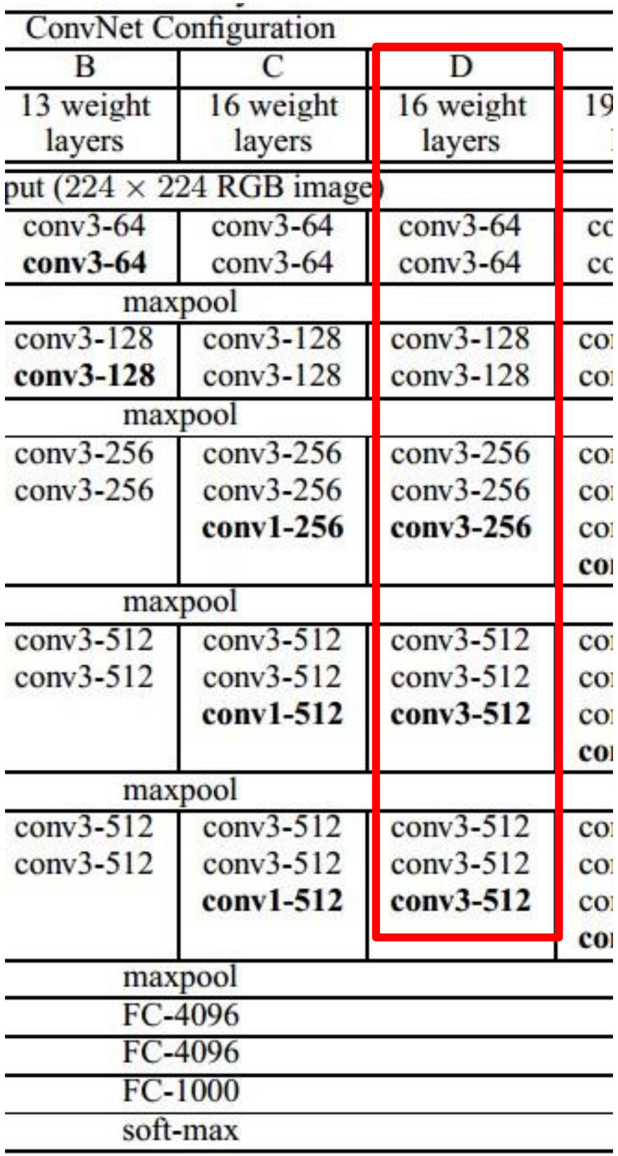
(params not counting biases)

ConvNet Configuration		
B	C	D
13 weight layers	16 weight layers	16 weight layers
put (224 × 224 RGB image)		
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
maxpool		
conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**
maxpool		
conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**
maxpool		
conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**
maxpool		
FC-4096		
FC-4096		
FC-1000		
soft-max		

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

(params not counting biases)

TOTAL memory: 24M * 4 bytes ~= 93MB / image
(only forward! Backward?)
TOTAL params: 138M parameters



ConvNet Configuration		
B	C	D
13 weight layers	16 weight layers	16 weight layers
put (224 × 224 RGB image)		
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
maxpool		
conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**
maxpool		
conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**
maxpool		
conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**
maxpool		
FC-4096		
FC-4096		
FC-1000		
soft-max		

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: **224*224*64=3.2M** params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: **224*224*64=3.2M** params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = **102,760,448**
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

(params not counting biases)

Note:
Most memory is in early CONV

Most params are in late FC
(Avg. Pool)

TOTAL memory: 24M * 4 bytes ~= 93MB / image
(only forward! ~*2 for bwd)
TOTAL params: 138M parameters

Case Study: GoogLeNet

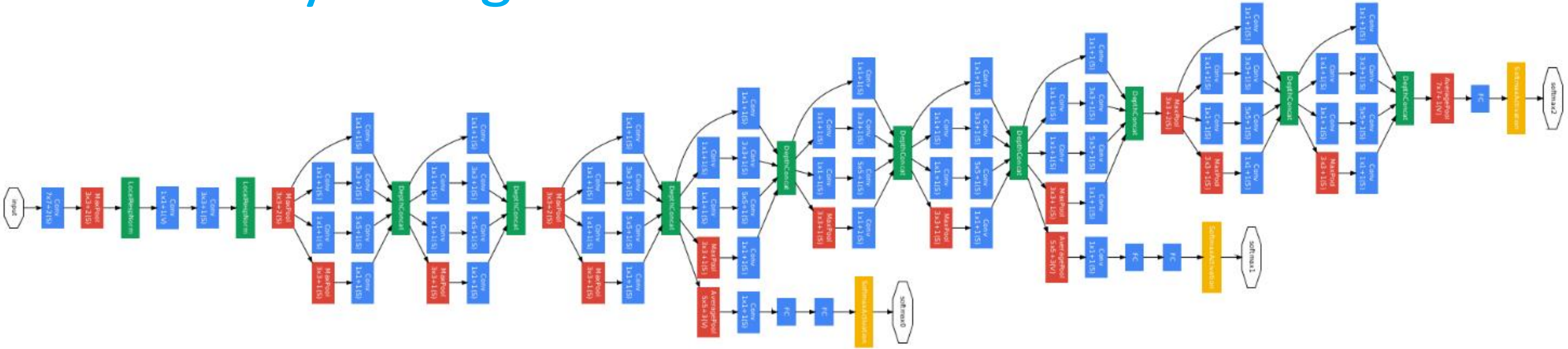
[Szegedy et al., 2014]

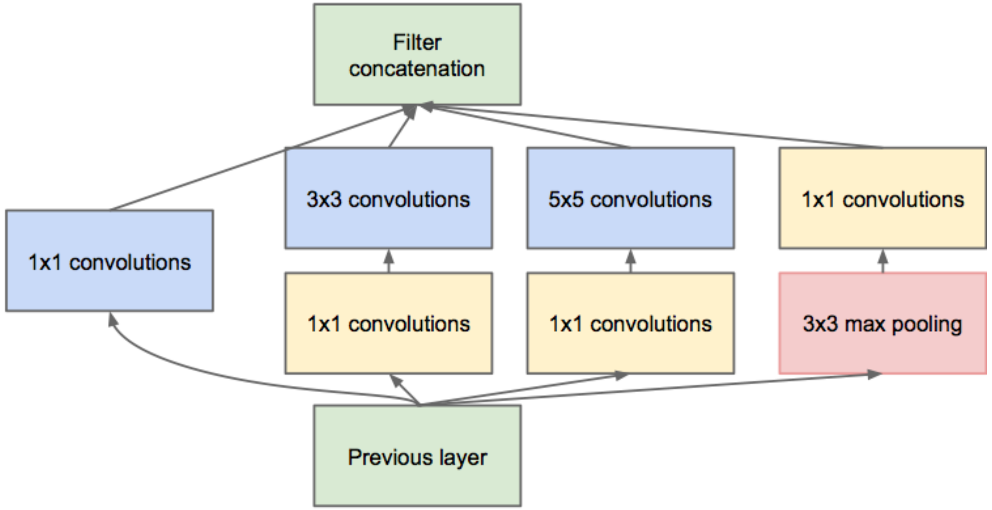
In this paper, we will focus on an efficient deep neural network architecture for computer vision, codenamed Inception, which derives its name from the Network in network paper by Lin et al [12] in conjunction with the famous "we need to go deeper" internet meme [1]. In our case, the word

[1] Know your meme: We need to go deeper. http://knowyourmeme.com/memes/we-need-to-go-deeper. Accessed: 2014-09-15.

Case Study: GoogLeNet

[Szegedy et al., 2014]

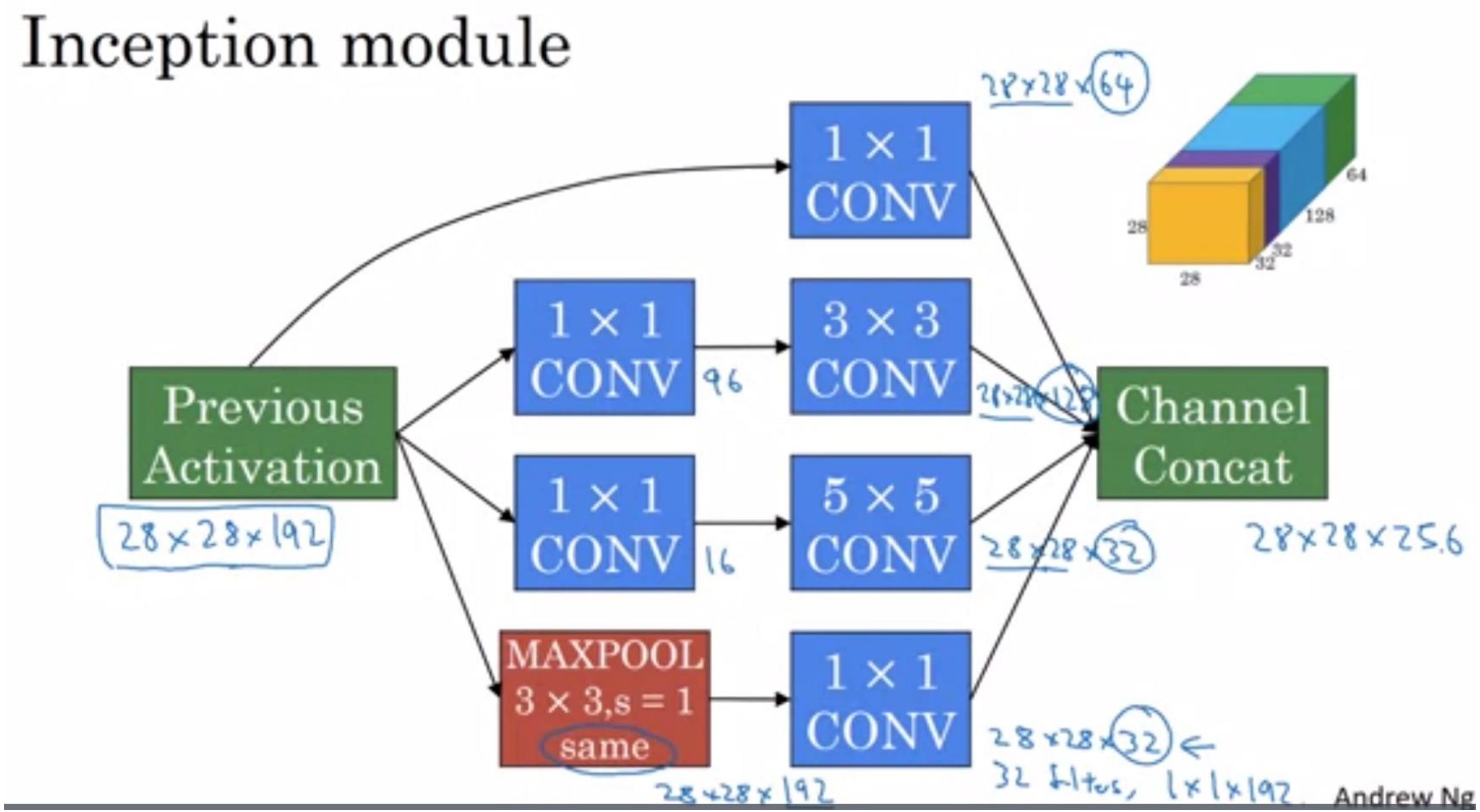




Inception module

ILSVRC 2014 winner (6.7% top 5 error)

Case Study: GoogLeNet



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%)

Case Study: ResNet

[He et al., 2015]

ILSVRC 2015 winner (3.6% top 5 error)

Microsoft Research

MSRA @ ILSVRC & COCO 2015 Competitions

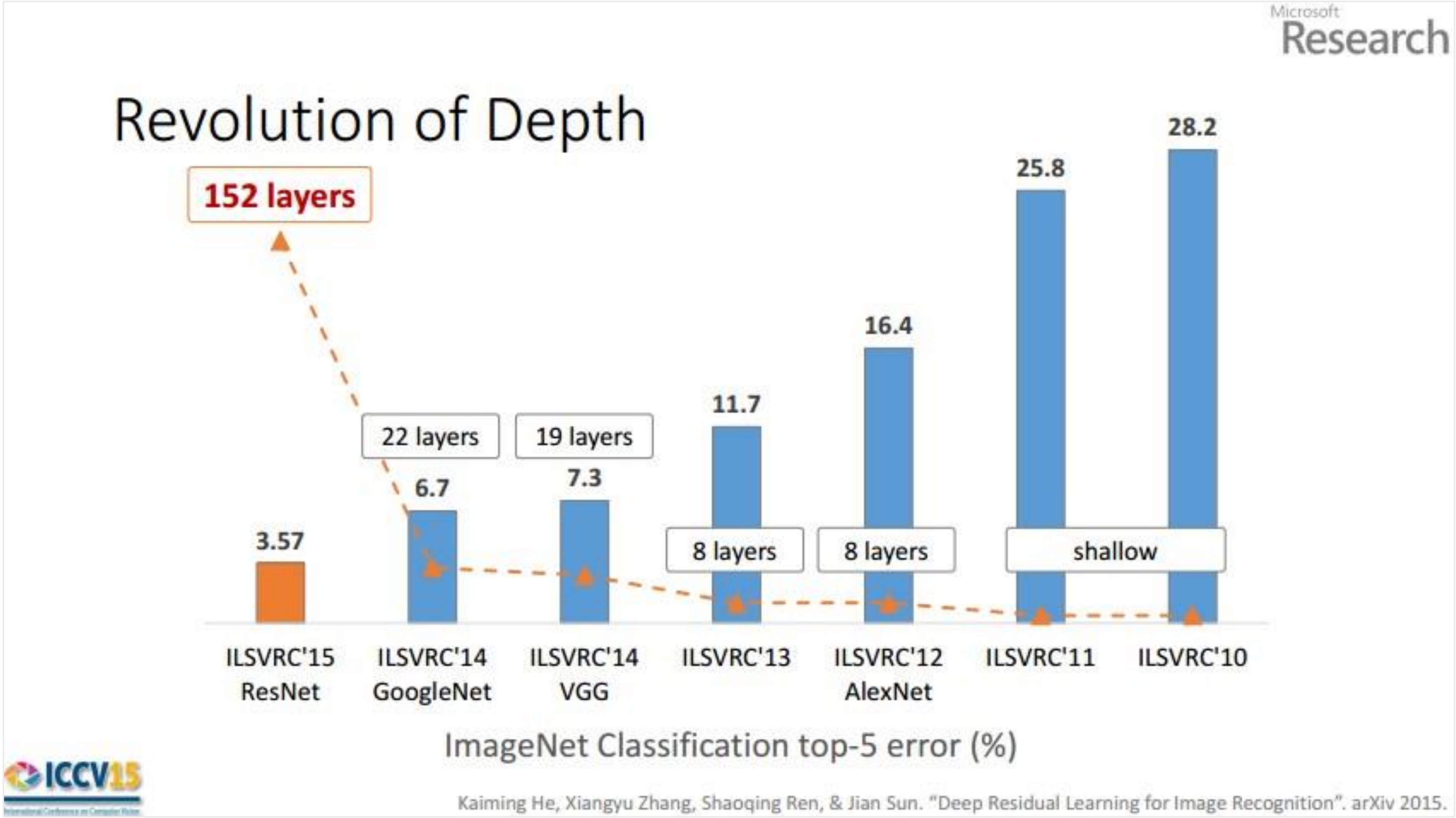
- **1st places in all five main tracks**
  - ImageNet Classification: *"Ultra-deep"* (quote Yann) 152-layer nets
  - ImageNet Detection: 16% better than 2nd
  - ImageNet Localization: 27% better than 2nd
  - COCO Detection: 11% better than 2nd
  - COCO Segmentation: 12% better than 2nd

*improvements are relative numbers

ICCV15

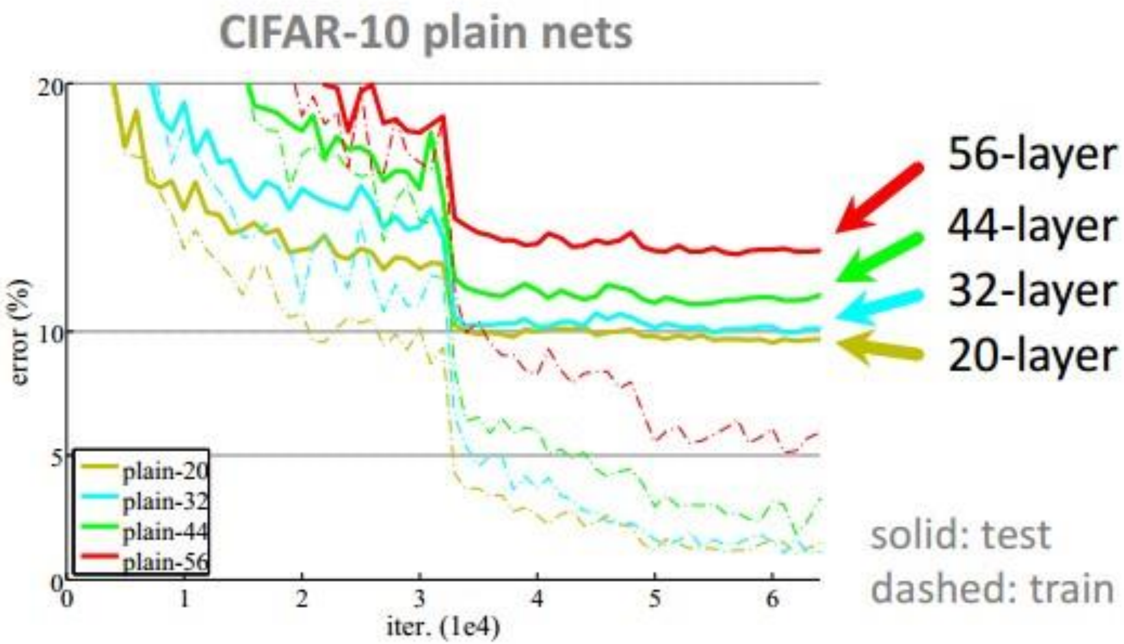
Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". arXiv 2015.

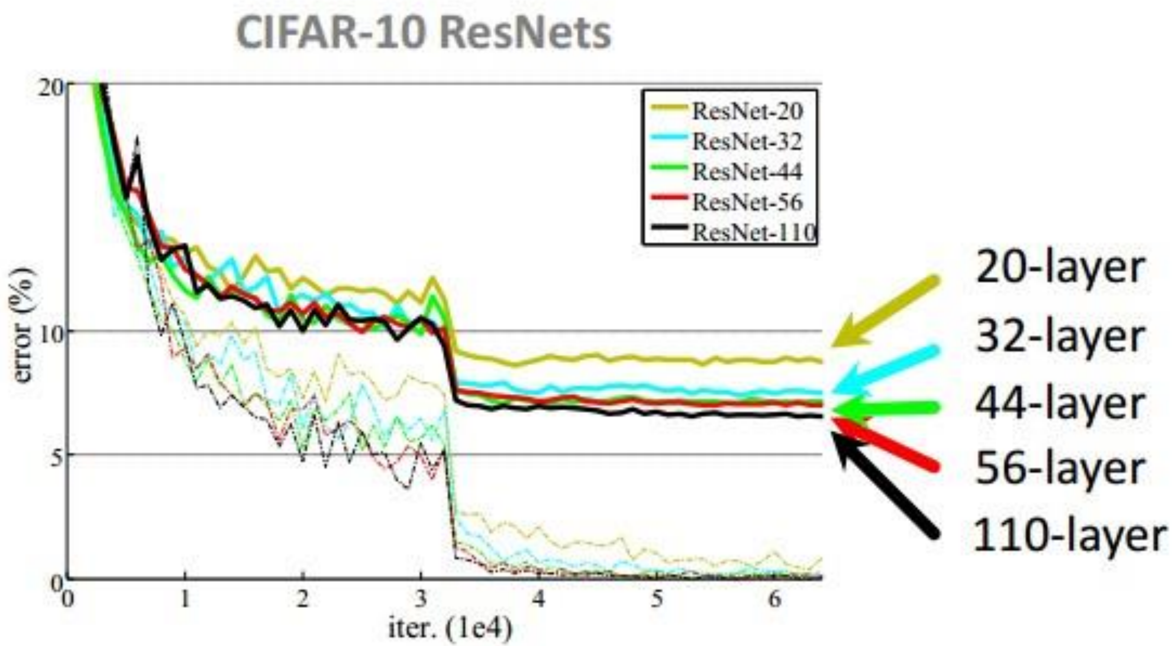
Slide from Kaiming He's recent presentation
https://www.youtube.com/watch?v=1PGLj-uKT1w



(slide from Kaiming He's recent presentation)

CIFAR-10 experiments

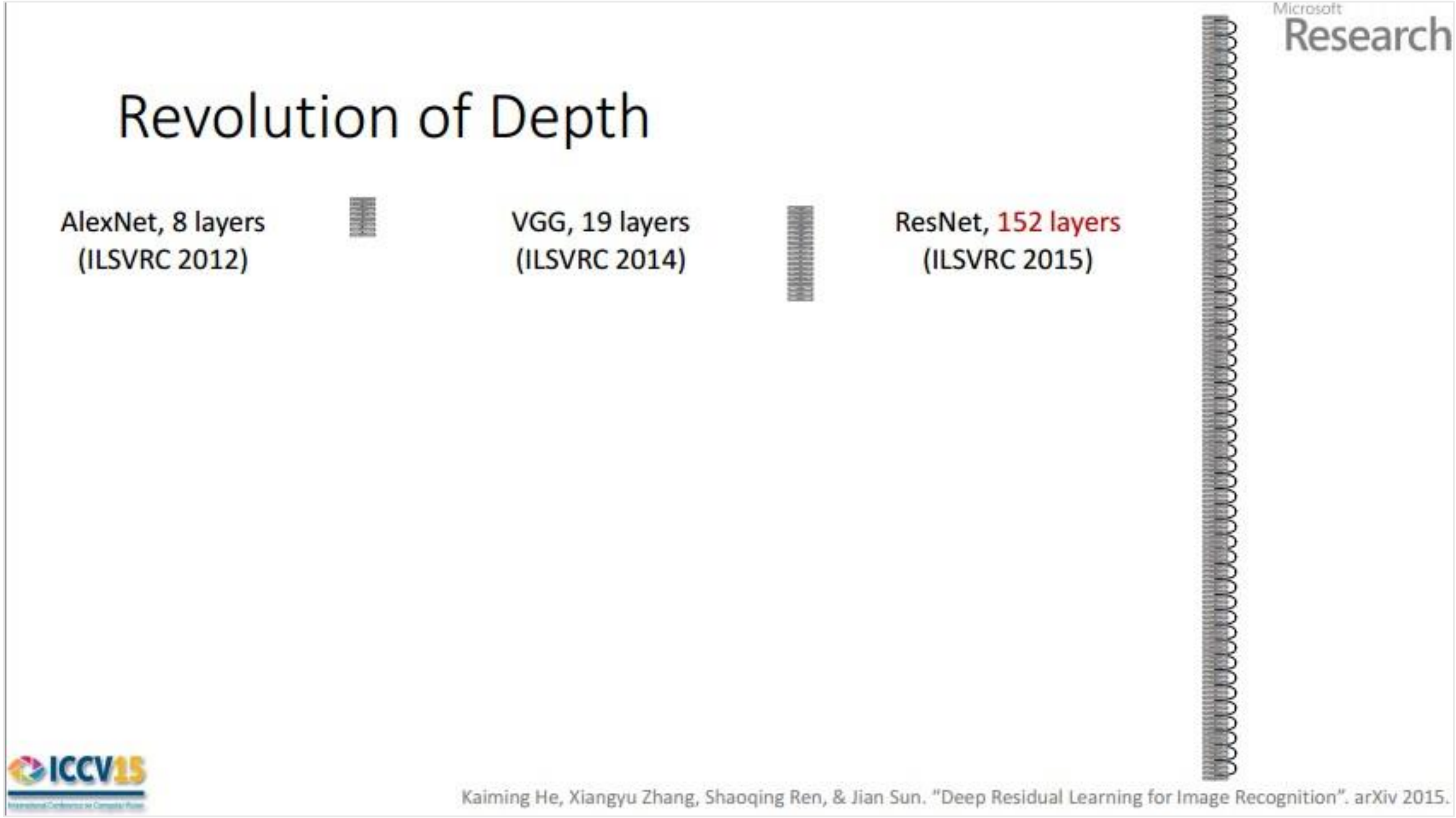




Case Study: ResNet

[He et al., 2015]

ILSVRC 2015 winner (3.6% top 5 error)



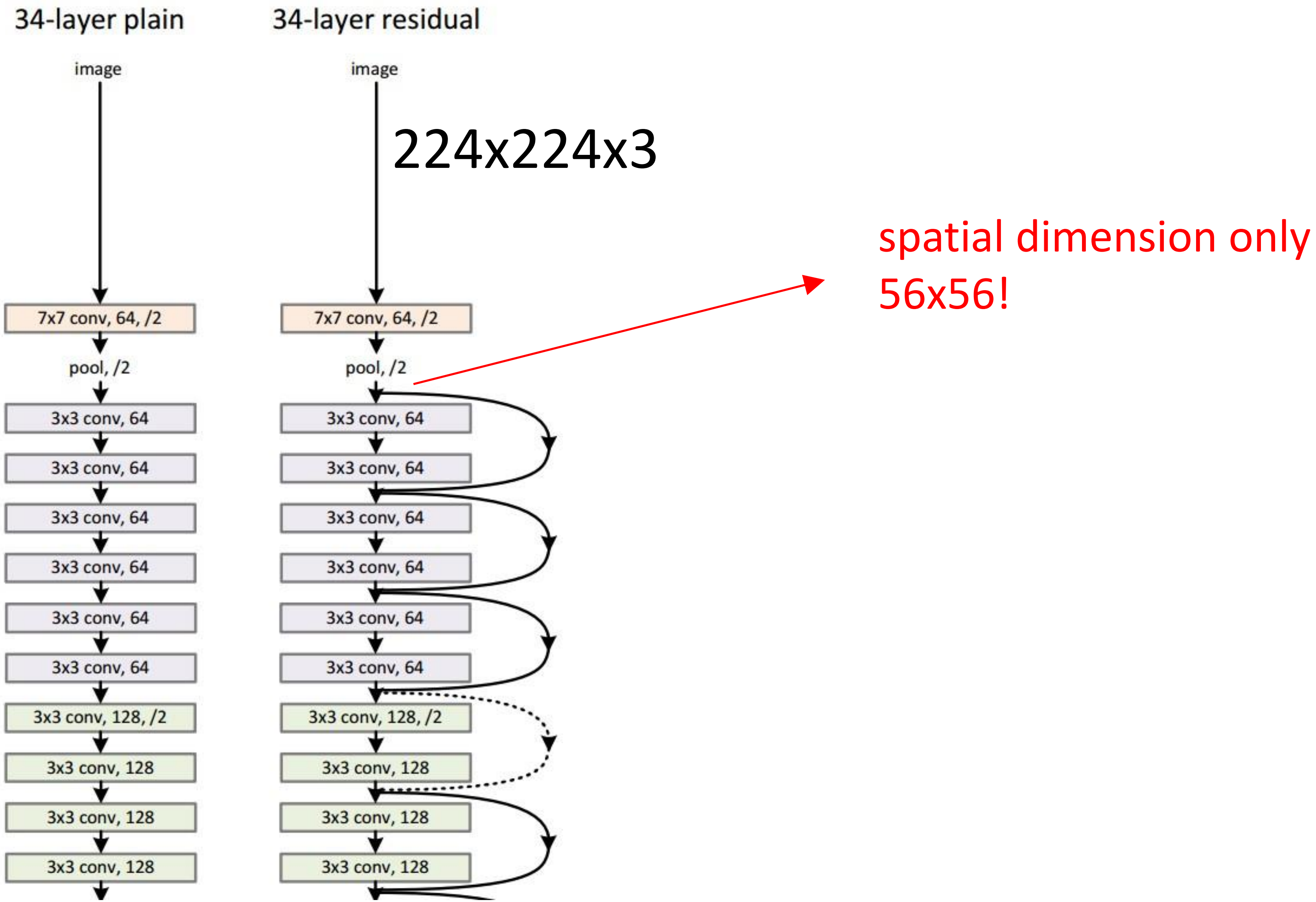
2-3 weeks of training on 8 GPU machine

at runtime: faster than a VGGNet! (even though it has 8x more layers)

(slide from Kaiming He's recent presentation)

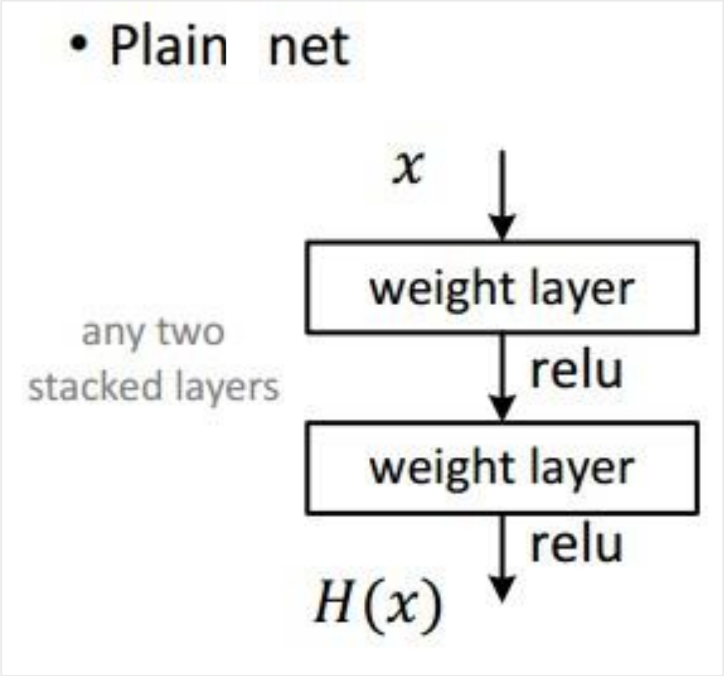
Case Study: ResNet

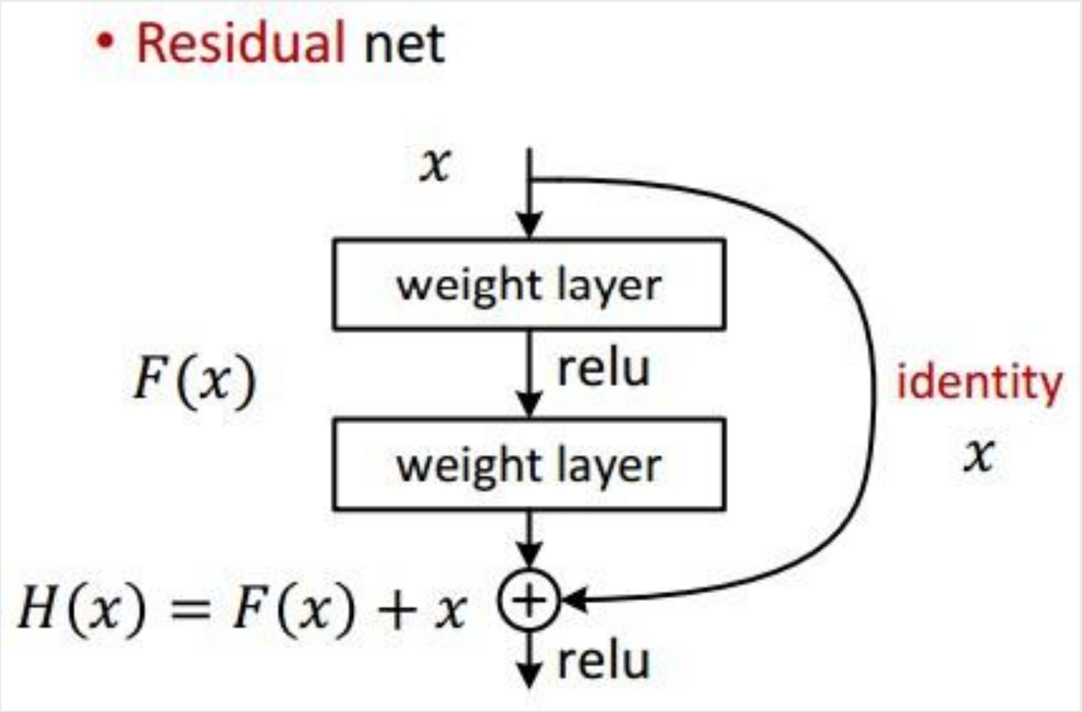
[He et al., 2015]



Case Study: ResNet

[He et al., 2015]





Case Study: ResNet

[He et al., 2015]

- Batch Normalization after every CONV layer
- Xavier/2 initialization from He et al.
- SGD + Momentum (0.9)
- Learning rate: 0.1, divided by 10 when validation error plateaus
- Mini-batch size 256
- Weight decay of 1e-5
- No dropout used

ILVRC 2016

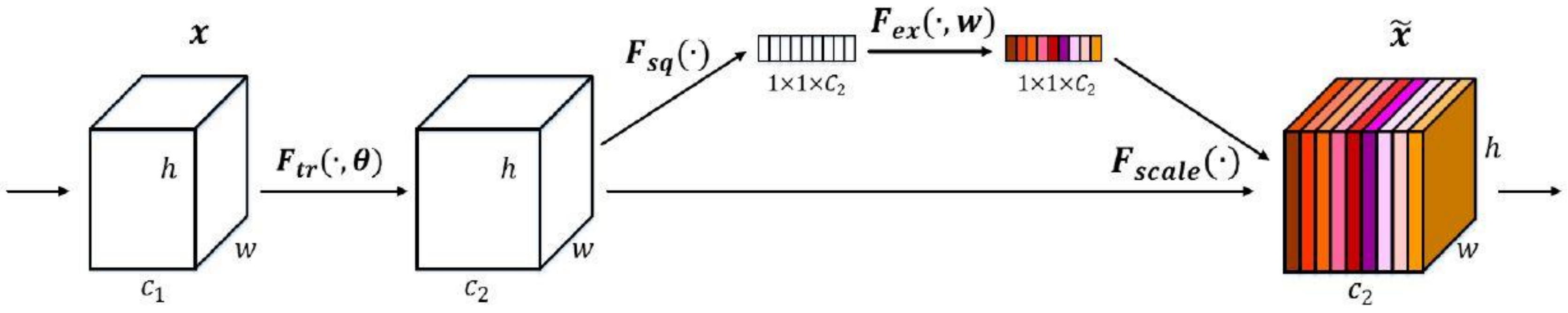
- Trimps-Soushen was the winner with an ensemble approach.
  - Jie Shao, Xiaoteng Zhang, Zhengyan Ding, Yixin Zhao, Yanjun Chen, Jianying Zhou, Wenfei Wang, Lin Mei, Chuanping Hu, The Third Research Institute of the Ministry of Public Security, P.R. China.
- Classification error is down to 3.0% from 3.6% last year.
- A bit boring:
  - https://www.reddit.com/r/MachineLearning/comments/54jiyy/large_scale_visual_recognition_challenge_2016/
  - http://image-net.org/challenges/LSVRC/2016/results#loc

ILVRC 2017, Squeeze & Excitation Network

- Squeeze and Excitation block that can be added to a ConvLayer
- Distributed deep learning training suing “ROCS”. Our system trains SE-ResNet152 with a minibatch size of 2048 on 64 Nvidia Pascal Titan X GPUs in 20 hours using synchronous SGD
- Winning entry - ensemble of SENets that obtain a 2.251% top-5 error on the test set. This result represents a ∼25% relative improvement on the winning entry of 2016 (2.99% top-5 error).

ILVRC 2017, Squeeze & Excitation Network

- Squeeze and Excitation block that can be added to a ConvLayer
- Let's add parameters to each channel of a convolutional block so that the network can adaptively adjust the weighting of each feature map.



Squeeze

- Shrinking feature maps $\in \mathbb{R}^{w\times h\times c_2}$ through spatial dimensions $(w \times h)$
- Global distribution of channel-wise responses

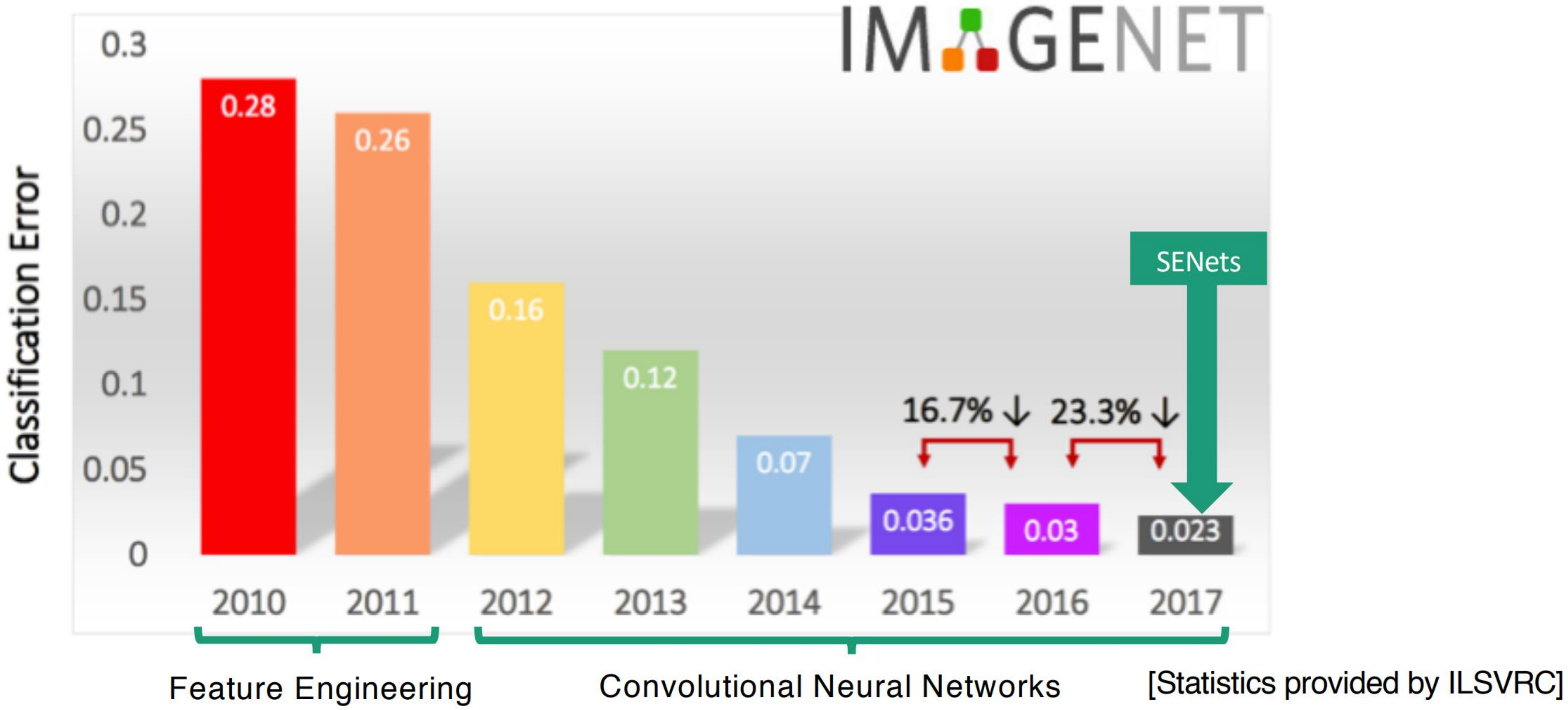
Excitation

- Learning $W \in \mathbb{R}^{c_2\times c_2}$ to explicitly model channel-association
- Gating mechanism to produce channel-wise weights

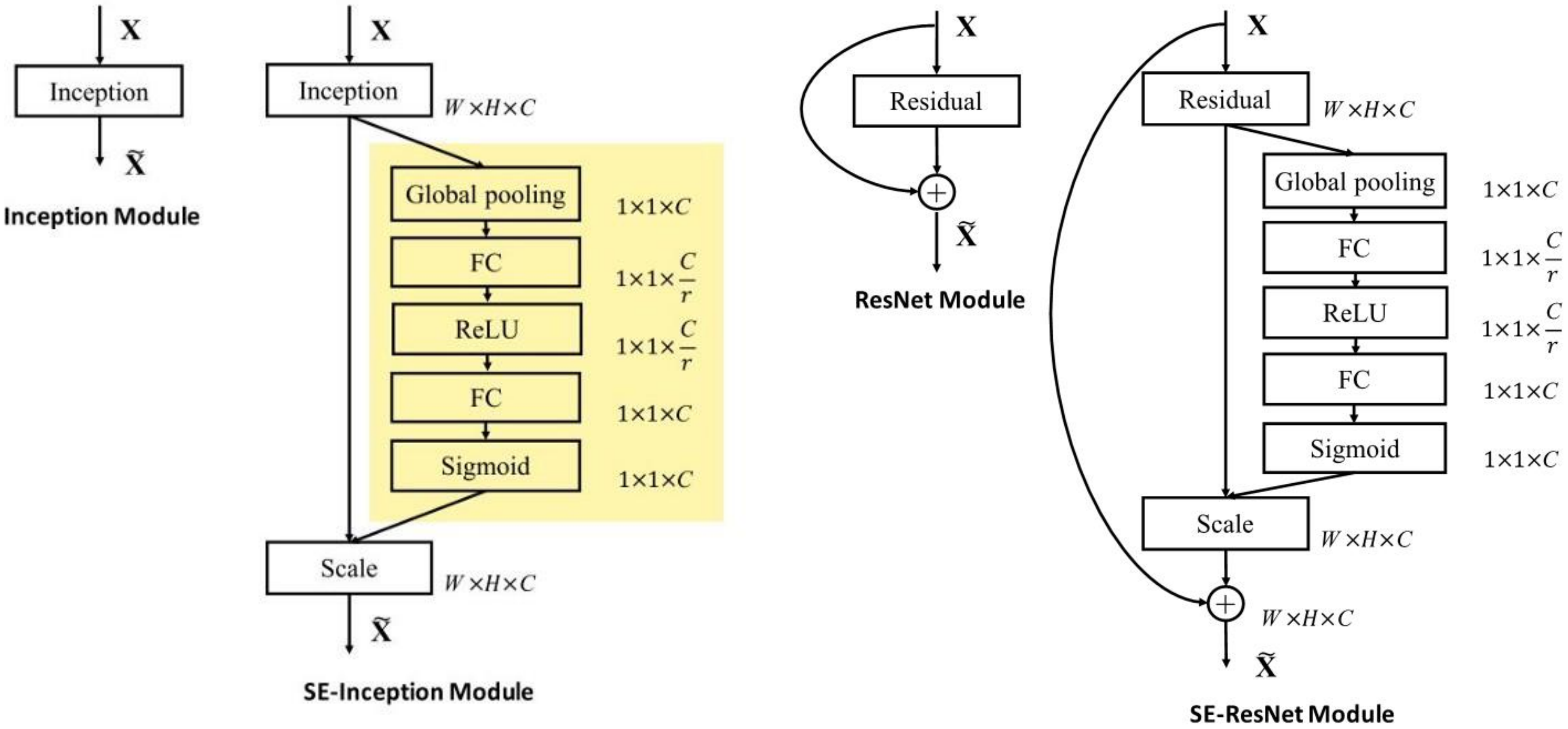
Scale

- Reweighting the feature maps $\in \mathbb{R}^{w\times h\times c_2}$

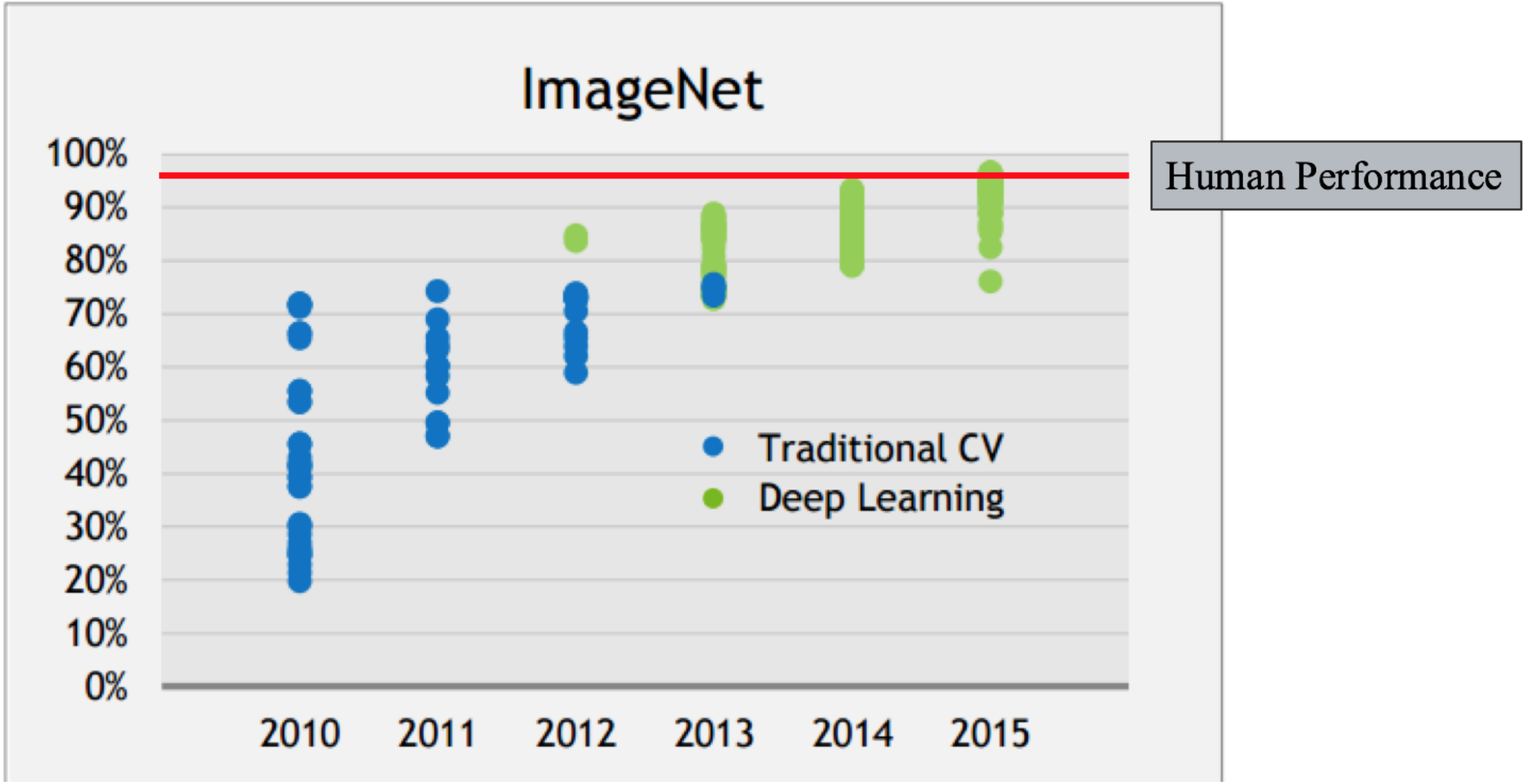
ILVRC 2012 - 2017



ILVRC 2017, Squeeze & Excitation Network



Why ConvNets?



Summary

- ConvNets stack CONV, POOL, FC layers
- Trend towards smaller filters and deeper architectures
- Trend towards getting rid of POOL/FC layers (just CONV)
- Typical architectures look like **[(CONV-RELU)*N-POOL?]*M-(FC-RELU)*K-SOFTMAX**
  where N is usually up to ~5, M is large, 0 <= K <= 2.
- but recent advances such as ResNet/GoogLeNet challenge this paradigm

Things you should know:

"ConvNets need a lot of data to train"

Things you should know:

"ConvNets need a lot of data to train"

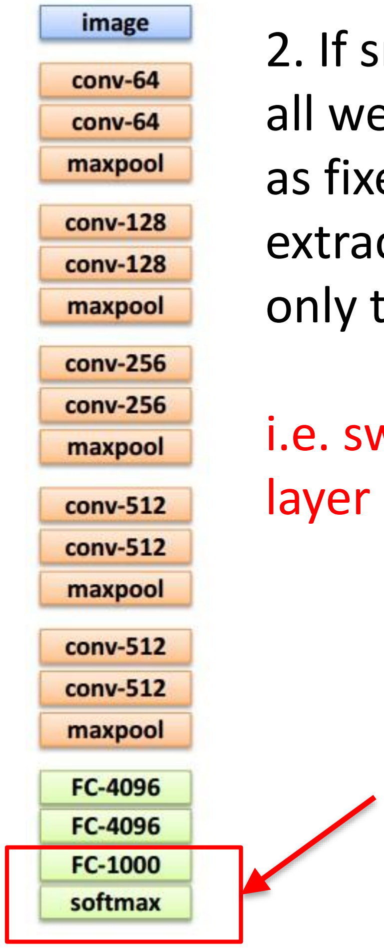
finetuning! we don't always need to train ConvNets from scratch.

1. Train on ImageNet

2. Finetune network on your own data

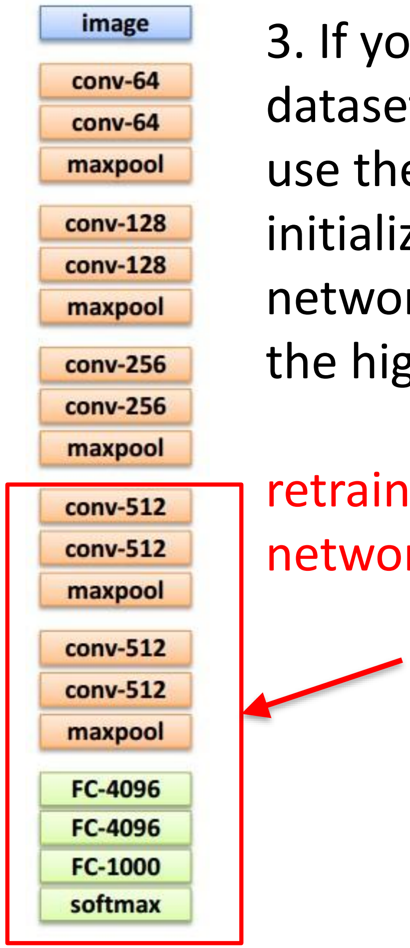
Transfer Learning with CNNs

1. Train on ImageNet



2. If small dataset: fix all weights (treat CNN as fixed feature extractor), retrain only the classifier

i.e. swap the Softmax layer at the end



3. If you have medium sized dataset, **"finetune"** instead: use the old weights as initialization, train the full network or only some of the higher layers

retrain bigger portion of the network, or even all of it.

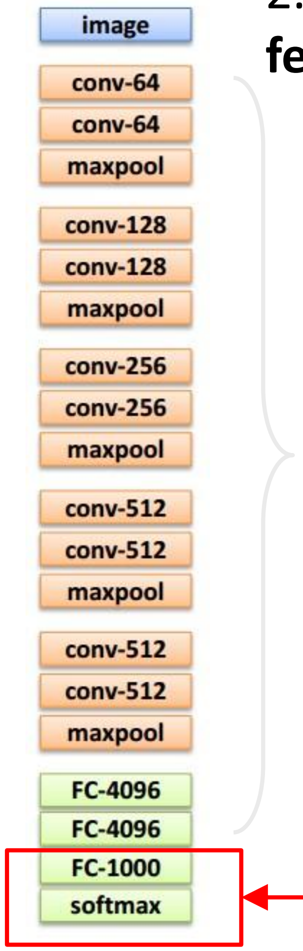
Transfer Learning with CNNs

1. Train on Imagenet

Transfer Learning with CNNs

Transfer Learning with CNNs

1. Train on Imagenet



2. Small dataset: **feature extractor**

Freeze these

Train this

3. Medium dataset: **finetuning**

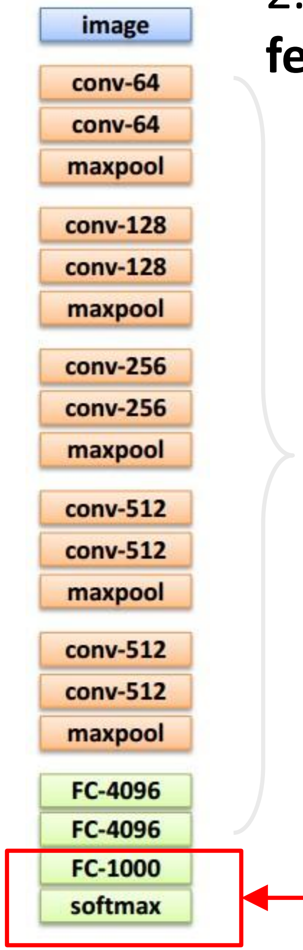
more data = retrain more of the network (or all of it)

Freeze these

Train this

Transfer Learning with CNNs

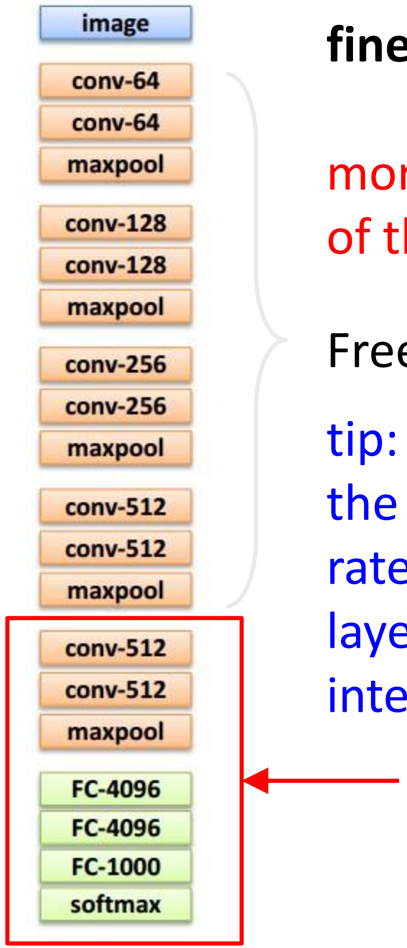
1. Train on Imagenet



2. Small dataset: **feature extractor**

Freeze these

Train this



3. Medium dataset: **finetuning**

more data = retrain more of the network (or all of it)

Freeze these

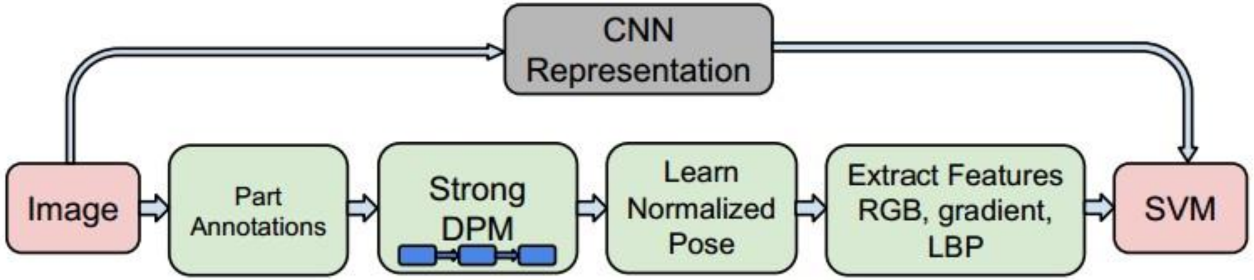
tip: use only ~1/10th of the original learning rate in finetuning top layer, and ~1/100th on intermediate layers

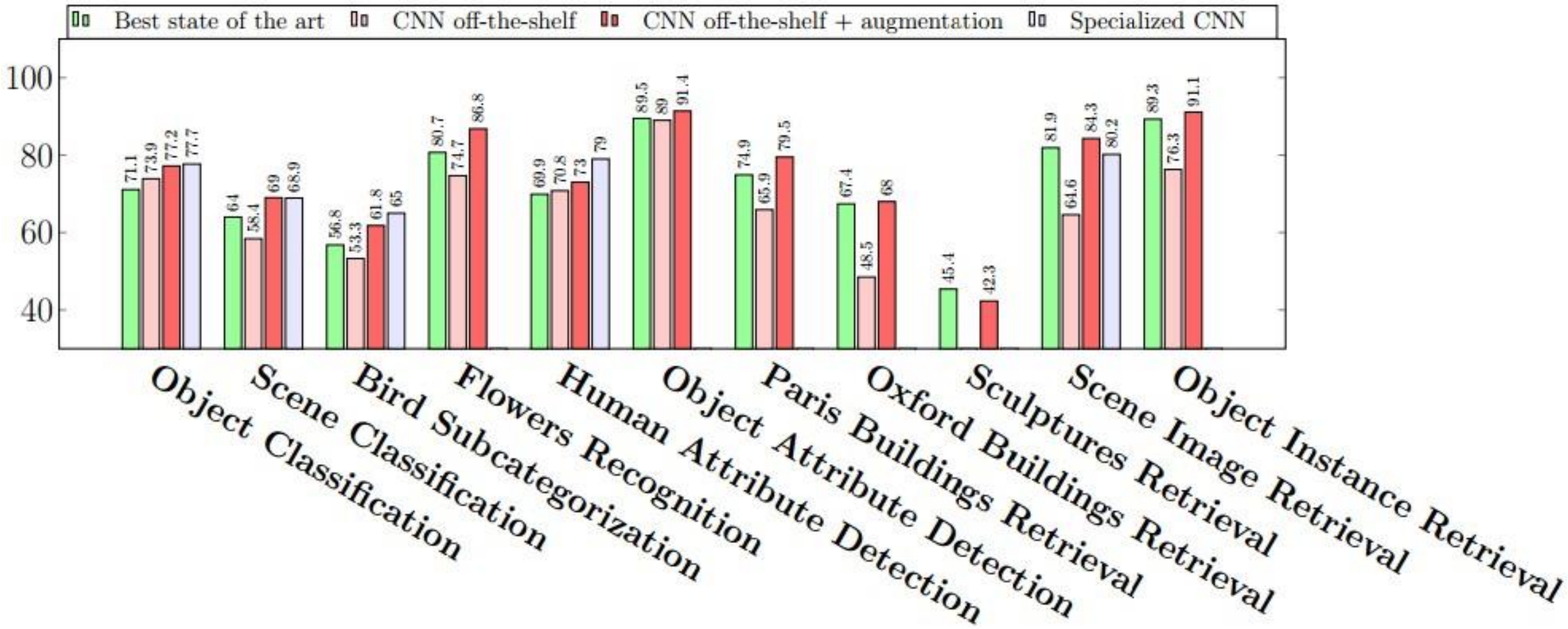
Train this

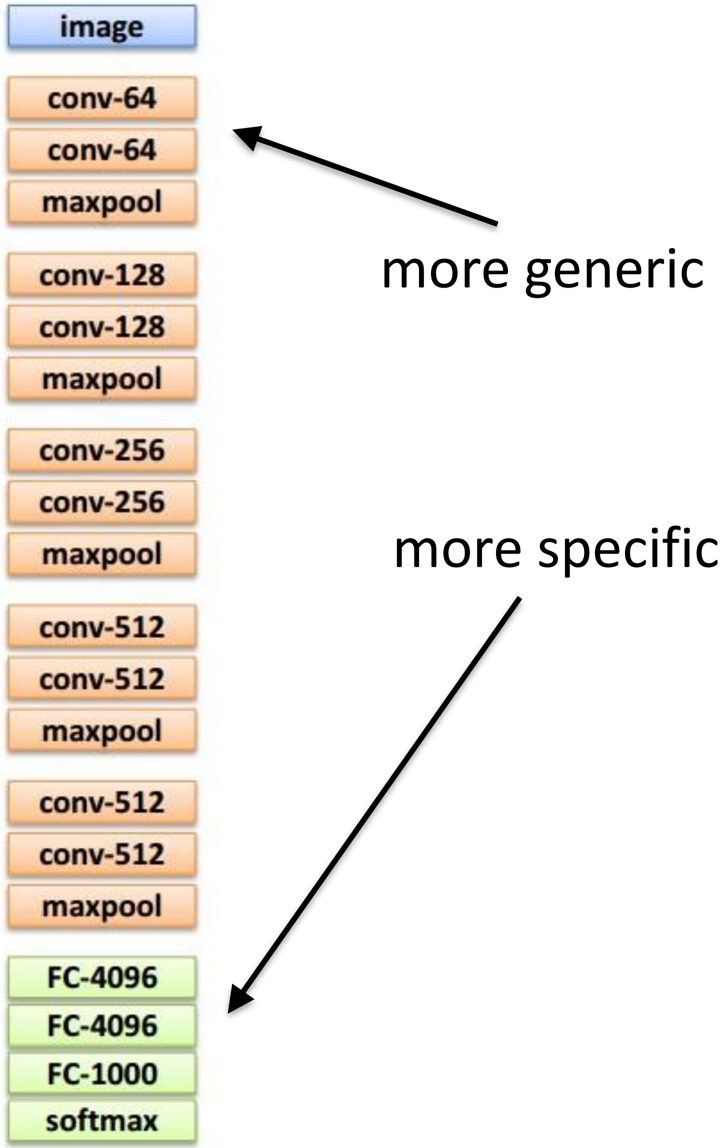
CNN Features off-the-shelf: an Astounding Baseline for Recognition [Razavian et al, 2014]

DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition [Donahue, Jia*, et al., 2013]*

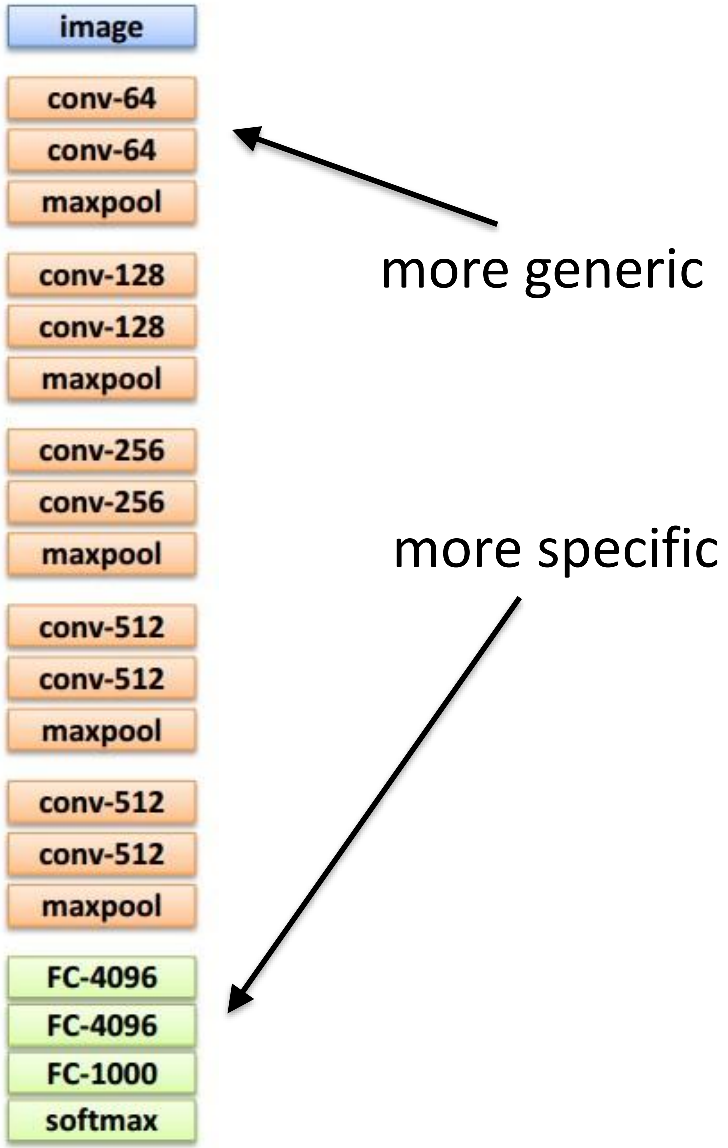
	$DeCAF_6$	$DeCAF_7$
LogReg	**40.94 ± 0.3**	40.84 ± 0.3
SVM	39.36 ± 0.3	40.66 ± 0.3
Xiao et al. (2010)	38.0	



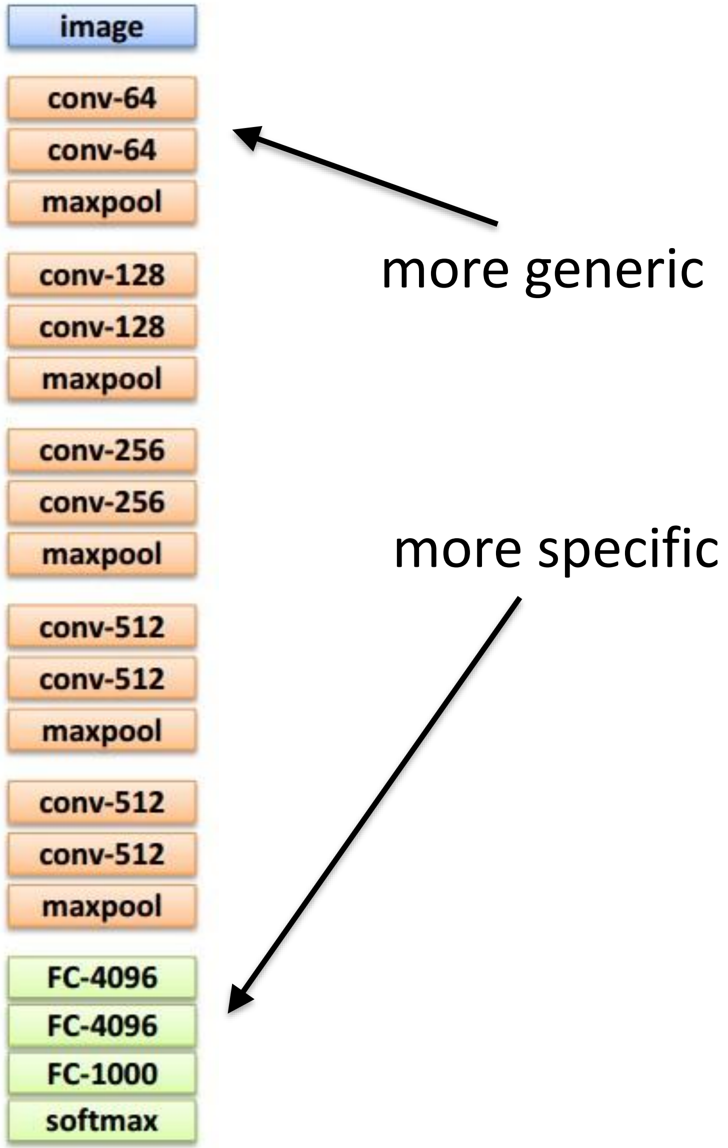




	very similar dataset	**very different dataset**
very little data	?	?
quite a lot of data	?	?

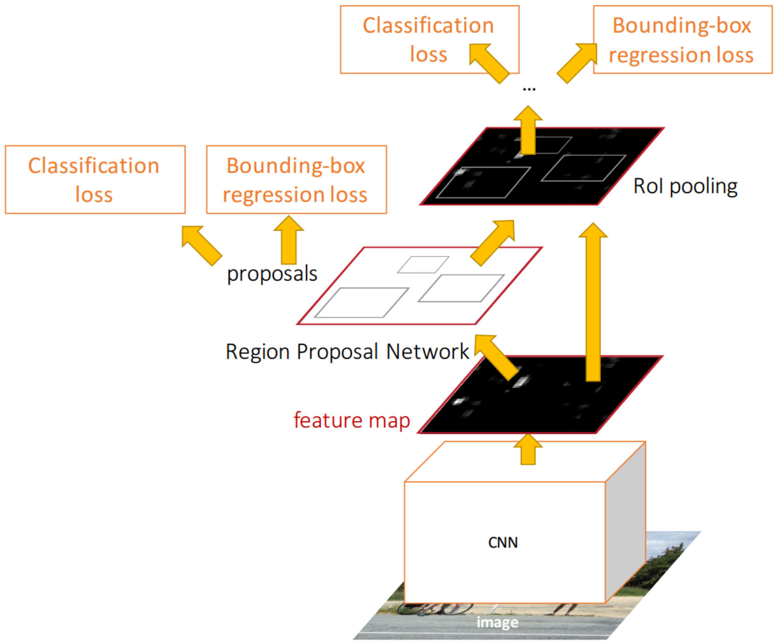


	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	?
quite a lot of data	Finetune a few layers	?



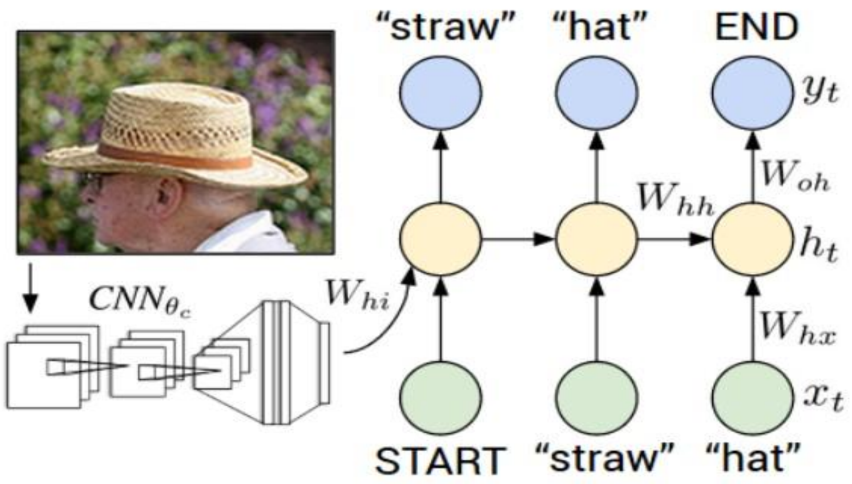
	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

Transfer learning with CNNs is pervasive…
(it's the norm, not an exception)

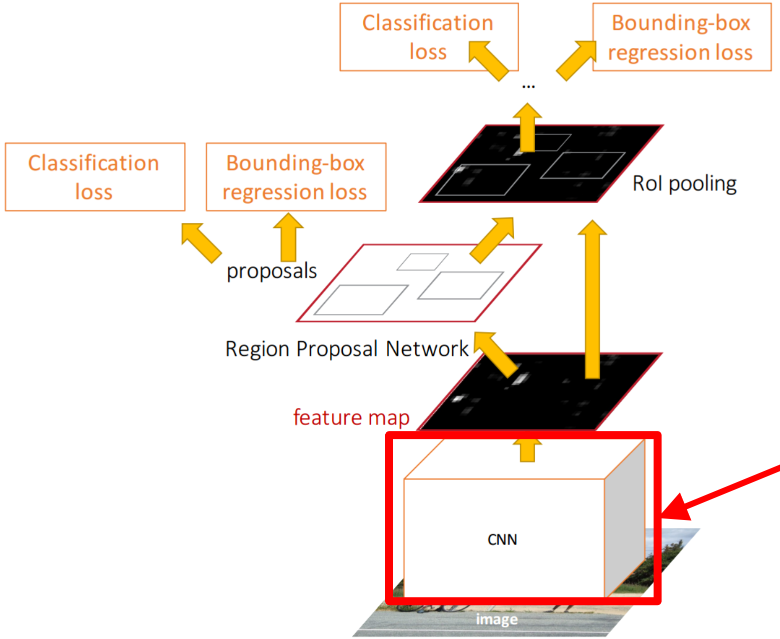


Object Detection
(Faster R-CNN)

Image Captioning: CNN + RNN

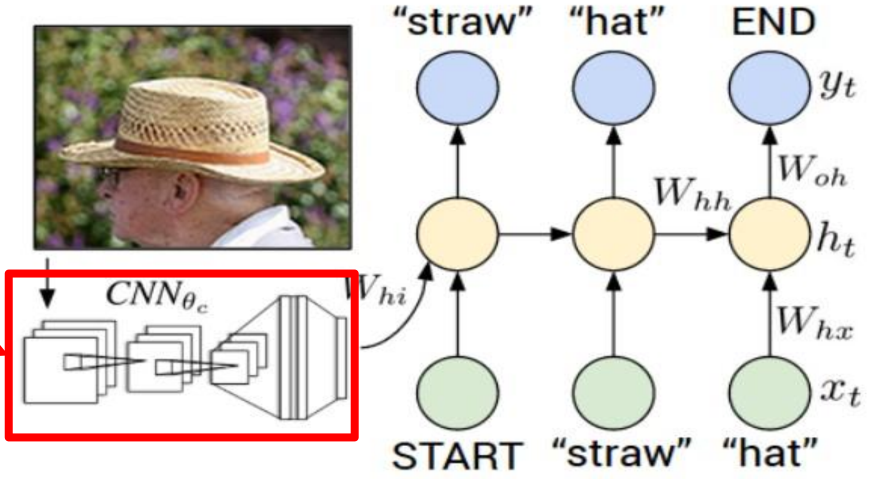


Transfer learning with CNNs is pervasive…
(it's the norm, not an exception)



CNN pretrained on ImageNet

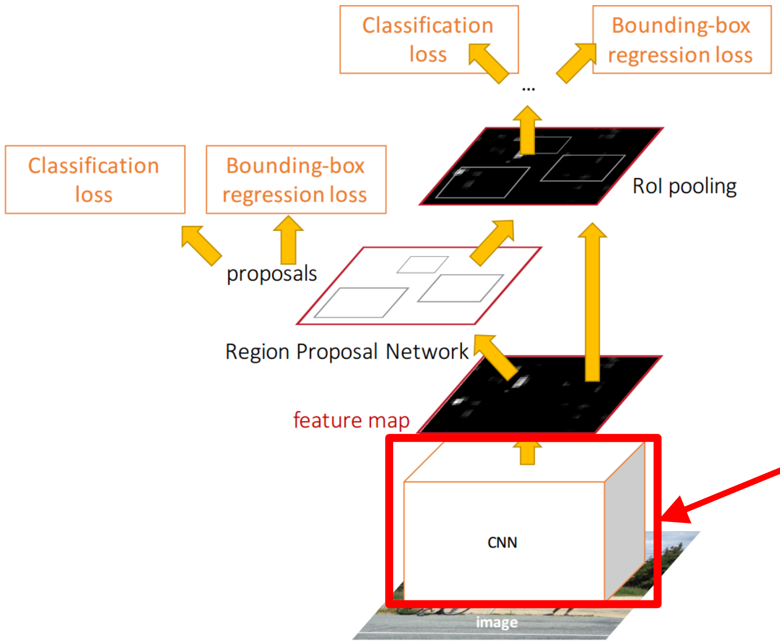
Image Captioning: CNN + RNN



Object Detection
(Faster R-CNN)

Transfer learning with CNNs is pervasive…

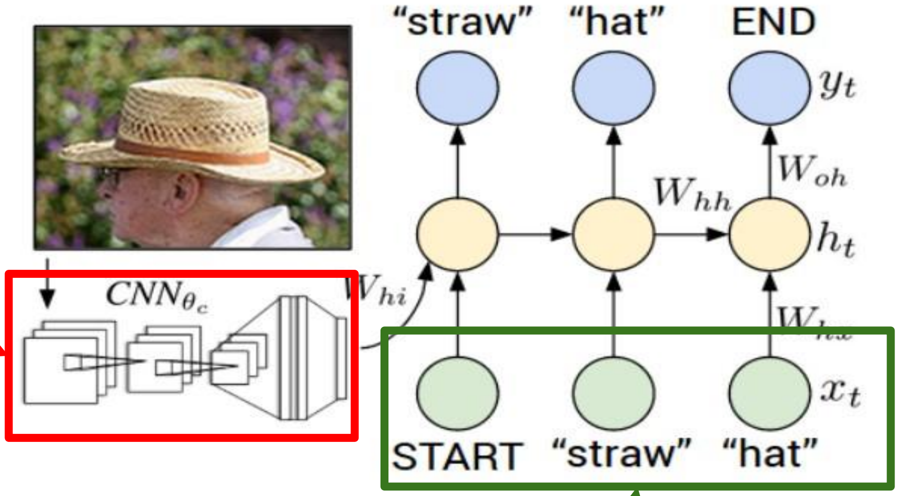
(it’s the norm, not an exception)



Object Detection
(Faster R-CNN)

CNN pretrained on ImageNet

Image Captioning: CNN + RNN



Word vectors pretrained from word2vec

E.g. Caffe Model Zoo: Lots of pretrained ConvNets

https://github.com/BVLC/caffe/wiki/Model-Zoo
https://github.com/szagoruyko/loadcaffe

Model Zoo

ELM edited this page 21 days ago · 56 revisions

Check out the model zoo documentation for details.

To acquire a model:

1. download the model gist by `./scripts/download_model_from_gist.sh <gist_id> <dirname>` to load the model metadata, architecture, solver configuration, and so on. (`<dirname>` is optional and defaults to caffe/models).
2. download the model weights by `./scripts/download_model_binary.py <model_dir>` where `<model_dir>` is the gist directory from the first step.

or visit the model zoo documentation for complete instructions.

Berkeley-trained models

- Finetuning on Flickr Style: same as provided in `models/`, but listed here as a Gist for an example.
- BVLC GoogleNet: `models/bvlc_googlenet`

Network in Network model

The Network in Network model is described in the following ICLR-2014 paper:

```
Network In Network
M. Lin, Q. Chen, S. Yan
International Conference on Learning Representations, 2014 (arXiv:1409.1556)
```

please cite the paper if you use the models.

Models:

- NIN-imagenet: a small(29MB) model for imagenet, yet performs slightly better than AlexNet, and fast to train. (Note: a more caffe-compatible version with correct convolutional weights shape: https://drive.google.com/folderview?id=0B0IedYUunOQINEFtUi1QNWVhVVU&usp=drive_web)
- NIN-CIFAR10: NIN model on CIFAR10, originally published in the paper Network in Network. The error rate of this model is 10.4% on CIFAR10.

Models from the BMVC-2014 paper "Return of the Devil in the Details: Delving Deep into Convolutional Nets"

The models are trained on the ILSVRC-2012 dataset. The details can be found on the project page or in the following BMVC-2014 paper:

```
Return of the Devil in the Details: Delving Deep into Convolutional Nets
K. Chatfield, K. Simonyan, A. Vedaldi, A. Zisserman
British Machine Vision Conference, 2014 (arXiv ref. cs1405.3531)
```

Please cite the paper if you use the models.

Models:

- VGG_CNN_S: 13.1% top-5 error on ILSVRC-2012-val
- VGG_CNN_M: 13.7% top-5 error on ILSVRC-2012-val
- VGG_CNN_M_2048: 13.5% top-5 error on ILSVRC-2012-val
- VGG_CNN_M_1024: 13.7% top-5 error on ILSVRC-2012-val
- VGG_CNN_M_128: 15.6% top-5 error on ILSVRC-2012-val
- VGG_CNN_F: 16.7% top-5 error on ILSVRC-2012-val

Models used by the VGG team in ILSVRC-2014

Places-CNN model from MIT.

Places CNN is described in the following NIPS 2014 paper:

```
B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva
Learning Deep Features for Scene Recognition using Places Database.
Advances in Neural Information Processing Systems 27 (NIPS) spotlight, 2014.
```

The project page is here

Models:

- Places205-AlexNet: CNN trained on 205 scene categories of Places Database (used in NIPS'14) with ~2.5 million images. The architecture is the same as Caffe reference network.
- Hybrid-CNN: CNN trained on 1183 categories (205 scene categories from Places Database and 978 object categories from the train data of ILSVRC2012 (ImageNet) with ~3.6 million images. The architecture is the same as Caffe reference network.
- Places205-GoogLeNet: GoogLeNet CNN trained on 205 scene categories of Places Database. It is used by Google in the deep dream visualization

GoogLeNet GPU implementation from Princeton.

We implemented GoogLeNet using a single GPU. Our main contribution is an effective way to initialize the network and a trick to overcome the GPU memory constraint by accumulating gradients over two training iterations.

- Please check http://vision.princeton.edu/pvt/GoogLeNet/ for more information. Pre-trained models on ImageNet and Places, and the training code are available for download.
- Make sure cls2_fc2 and cls3_fc have num_output = 1000 in the prototxt. Otherwise, the trained model would crash on test.

Fully Convolutional Semantic Segmentation Models (FCN-Xs)

These models are described in the paper:

```
Fully Convolutional Models for Semantic Segmentation
Jonathan Long, Evan Shelhamer, Trevor Darrell
CVPR 2015
arXiv:1411.4038
```

They are available under the same license as the Caffe-bundled models (i.e., for unrestricted use; see http://caffe.berkeleyvision.org/model_zoo.html#bvlc-model-license).

These are pre-release models. They do not run in any current version of BVLC/caffe, as they require unmerged PRs. They should run in the preview branch provided at https://github.com/longjon/caffe/tree/future. The FCN-32s PASCAL-Context model is the most complete example including network definitions, solver configuration, and Python scripts for solving and inference.

Models trained on PASCAL (using extra data from Hariharan et al. and finetuned from the ILSVRC-trained VGG-16 model above):

- FCN-32s PASCAL: single stream, 32 pixel prediction stride version
- FCN-16s PASCAL: two stream, 16 pixel prediction stride version
- FCN-8s PASCAL: three stream, 8 pixel prediction stride version
- FCN-AlexNet PASCAL: AlexNet (CaffeNet) single stream, 32 pixel prediction stride version

To reproduce the validation scores, use the seg11valid split defined by the paper in footnote 7. Since SBD train and PASCAL VOC 11 segval intersect, we only evaluate on the non-intersecting set for validation purposes.

Models trained on SIFT Flow (also finetuned from VGG-16):

- FCN-16s SIFT Flow: two stream, 16 pixel prediction stride version

Models trained on NYUDv2 (also finetuned from VGG-16, and using HHA features from Gupta et al. https://github.com/s-gupta/rcnn-depth):

- FCN-32s NYUDv2: single stream, 32 pixel prediction stride version
- FCN-16s NYUDv2: two stream, 16 pixel prediction stride version

Models trained on PASCAL-Context including training model definition, solver configuration, and barebones solving script (finetuned from the ILSVRC-trained VGG-16 model):

- FCN-32s PASCAL-Context: single stream, 32 pixel prediction stride version
- FCN-16s PASCAL-Context: two stream, 16 pixel prediction stride version
- FCN-8s PASCAL-Context: three stream, 8 pixel prediction stride version

CaffeNet fine-tuned for Oxford flowers dataset

https://gist.github.com/jimgoo/0179e52305ca768a601f

The is the reference CaffeNet (modified AlexNet) fine-tuned for the Oxford 102 category flower dataset. The number of outputs in the inner product layer has been set to 102 to reflect the number of flower categories. Hyperparameter choices reflect those in Fine-tuning CaffeNet for Style Recognition on "Flickr Style" Data. The global learning rate is reduced while the learning rate for the final fully connected is increased relative to the other layers.

After 50,000 iterations, the top-1 error is 7% on the test set of 1,020 images.

CNN Models for Salient Object Subitizing.

CNN models described in the following CVPR'15 papger "Salient Object Subitizing":

```
Salient Object Subitizing
J. Zhang, S. Ma, M. Sameki, S. Sclaroff, M. Betke, Z. Lin, X. Shen, B. Price and R.
CVPR, 2015.
```

Models:

- AlexNet: CNN model finetuned on the Salient Object Subitizing dataset (~5500 images). The architecture is the same as the Caffe reference network.
- VGG16: CNN model finetuned on the Salient Object Subitizing dataset (~5500 images). The architecture is the same as the VGG16 network. This model gives better performance than the AlexNet model, but is slower for training and testing.

Deep Learning of Binary Hash Codes for Fast Image Retrieval

We present an effective deep learning framework to create the hash-like binary codes for fast image retrieval. The details can be found in the following "CVPRW'15 paper":

```
Deep Learning of Binary Hash Codes for Fast Image Retrieval
K. Lin, H.-F. Yang, J.-H. Hsiao, C.-S. Chen
CVPR 2015, DeepVision workshop
```

please cite the paper if you use the model:

- caffe-cvprw15: See our code release on Github, which allows you to train your own deep hashing model and create binary hash codes.
- CIFAR10-48bit: Proposed 48-bits CNN model trained on CIFAR10.

Places_CNDS_models on Scene Recognition

- Places-CNDS-8 is a "8conv3fc layer" deep Convolutional neural Networks model trained on MIT Places Dataset with Deep Supervision.

The details of training this model are described in the following report. Please cite this work if the model is useful for you.

```
Training Deeper Convolutional Networks with Deep Supervision
L.Wang, C.Lee, Z.Tu, S. Lazebnik, arXiv:1505.02496, 2015
```

Models for Age and Gender Classification.

- Age/Gender.net are models for age and gender classification trained on the Adience-OUI dataset. See the Project page.

The models are described in the following paper:

```
Age and Gender Classification using Convolutional Neural Networks
Gil Levi and Tal Hassner
IEEE Workshop on Analysis and Modeling of Faces and Gestures (AMFG),
at the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR), Boston, June 20
```

If you find our models useful, please add suitable reference to our paper in your work.

GoogLeNet_cars on car model classification

GoogLeNet_cars is the GoogLeNet model pre-trained on ImageNet classification task and fine-tuned on 431 car models in CompCars dataset. It is described in the technical report. Please cite the following work if the model is useful for you.

```
A Large-Scale Car Dataset for Fine-Grained Categorization and Verification
L. Yang, P. Luo, C. C. Loy, X. Tang, arXiv:1506.08959, 2015
```

Holistically-Nested Edge Detection

The model and code provided are described in the ICCV 2015 paper:

```
Holistically-Nested Edge Detection
Saining Xie and Zhuowen Tu
ICCV 2015
```

For details about training/evaluating HED, please take a look at http://github.com/s9xie/hed.

Model trained on BSDS-500 Dataset (finetuned from the VGGNet):

- HED BSDS-500

Translating Videos to Natural Language

These models are described in this NAACL-HLT 2015 paper.

```
Translating Videos to Natural Language Using Deep Recurrent Neural Networks
S. Venugopalan, H. Xu, J. Donahue, M. Rohrbach, R. Mooney, K. Saenko
NAACL-HLT 2015
```

More details can be found on this project page.

Model:
Video2Text_VGG_mean_pool: This model is an improved version of the mean pooled model described in the NAACL-HLT 2015 paper. It uses video frame features from the VGG-16 layer model. This is trained only on the Youtube video dataset.

Compatibility: These are pre-release models. They do not run in any current version of BVLC/caffe, as they require unmerged PRs. The models are currently supported by the `recurrent` branch of the Caffe fork provided at https://github.com/jeffdonahue/caffe/tree/recurrent and https://github.com/vsubhashini/caffe/tree/recurrent.

VGG Face CNN descriptor

These models are described in this BMVC 2015 paper.

```
Deep Face Recognition
Omkar M. Parkhi, Andrea Vedaldi, Andrew Zisserman
BMVC 2015
```

More details can be found on this project page.

Model: VGG Face: This is the very deep architecture based model trained from scratch using 2.6 Million images of celebrities collected from the web. The model has been imported to work with Caffe from the original model trained using MatConvNet library.

If you find our models useful, please add suitable reference to our paper in your work.

Yearbook Photo Dating

Model from the ICCV 2015 Extreme Imaging Workshop paper:

```
A Century of Portraits: Exploring the Visual Historical Record of American High Scho
Shiry Ginosar, Kate Rakelly, Brian Yin, Sarah Sachs, Alyosha Efros
ICCV Workshop 2015
```

Model and prototxt files: Yearbook

CCNN: Constrained Convolutional Neural Networks for Weakly Supervised Segmentation

These models are described in the ICCV 2015 paper.

```
Constrained Convolutional Neural Networks for Weakly Supervised Segmentation
Deepak Pathak, Philipp Krähenbühl, Trevor Darrell
ICCV 2015
arXiv:1506.03648
```

These are pre-release models. They do not run in any current version of BVLC/caffe, as they require unmerged PRs. Full details, source code, models, prototxts are available here: CCNN.

...

Thank you!

Data Preprocessing

Data Preprocessing

TLDR: In practice for Images: center only

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

Not common to normalize variance, to do PCA or whitening

Data Augmentation

Data Augmentation

4. Get creative!

Random mix/combinations of :

- translation (what about a pure ConvNet?)
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)

Data Augmentation: Takeaway

Simple to implement, use it

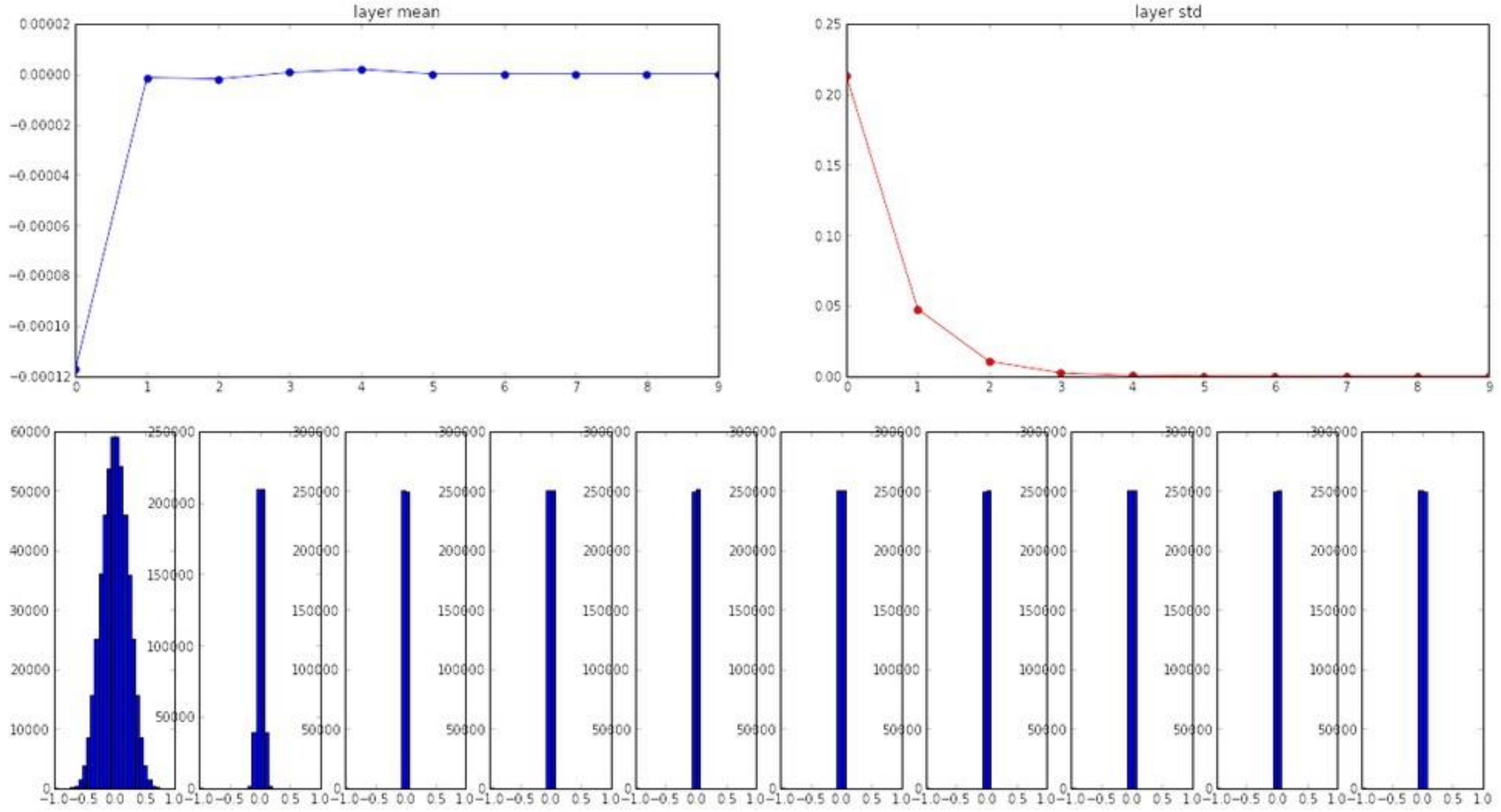
Especially useful for small datasets, helps a lot with fixing overfitting

Fits into framework of noise / marginalization

Weight Initialization

Lets look at some activation statistics

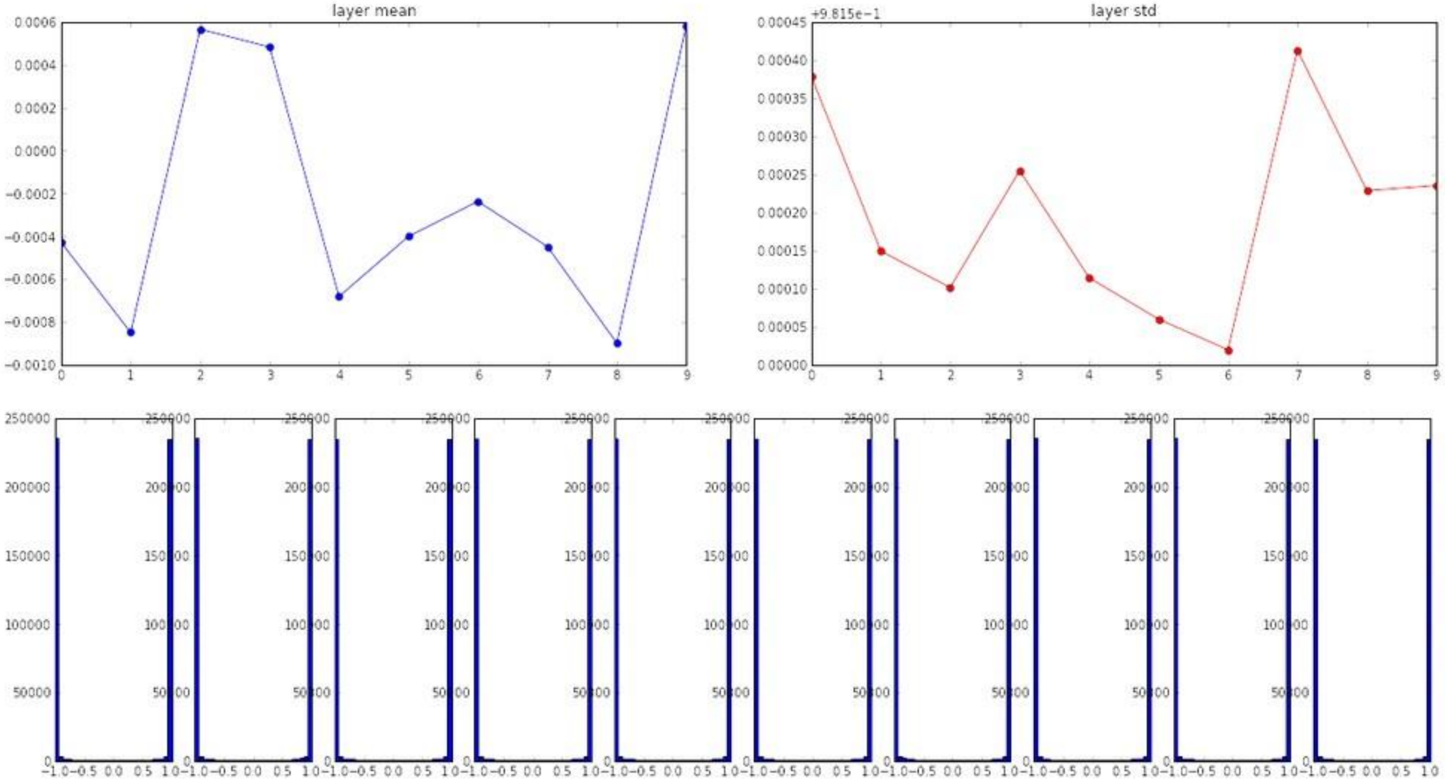
```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```



```
self.W = torch.randn(fan_out, fan_in) * 1.0
self.b = torch.randn(fan_out)* 1.0
```

*1.0 instead of *0.01

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```

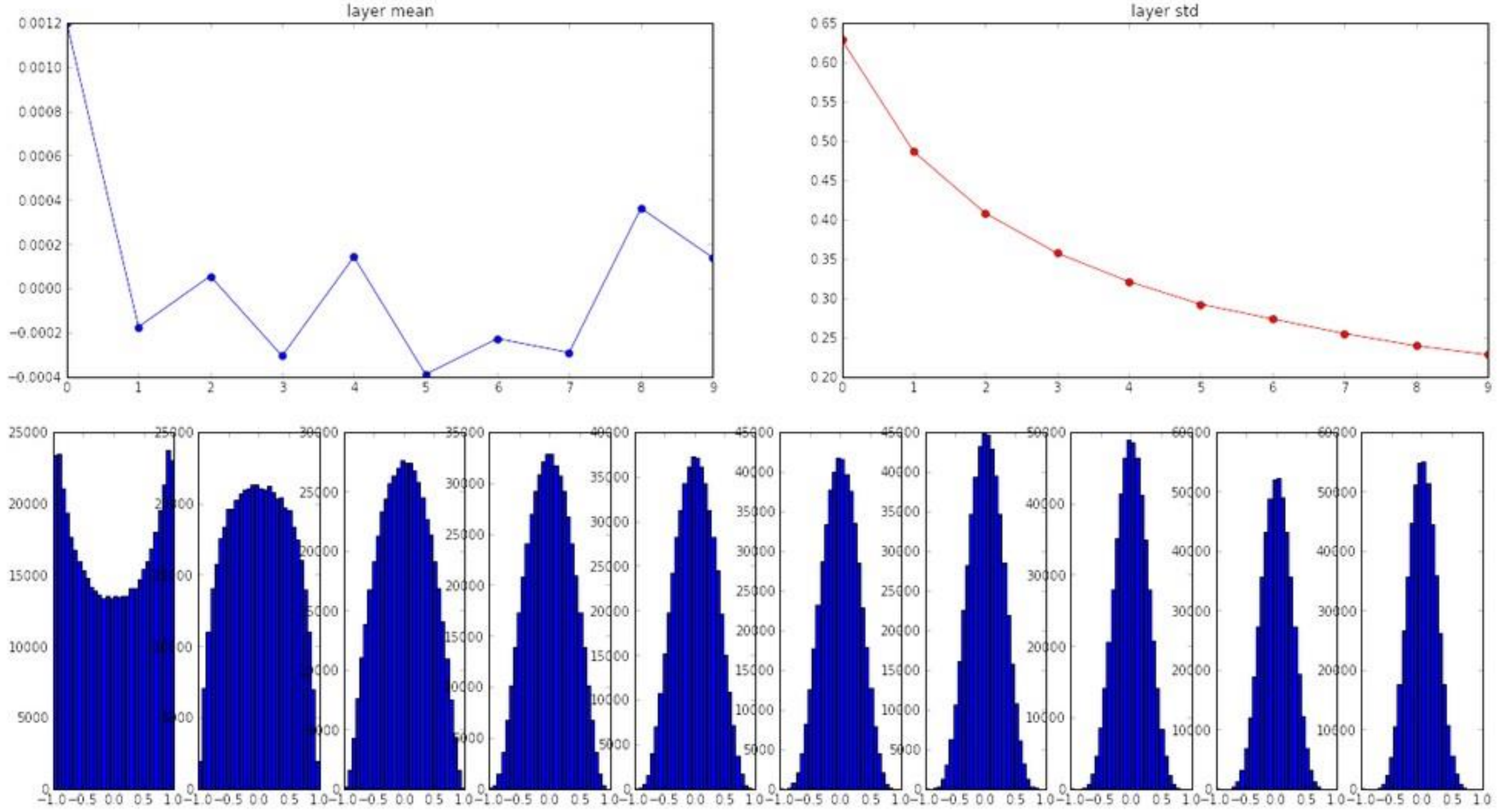


Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

```
self.W = torch.randn(fan_out, fan_in) / math.sqrt(fan_in)
self.b = torch.randn(fan_out) / math.sqrt(fan_in)
```

"Xavier initialization"
[Glorot et al., 2010]



Reasonable initialization.
(Mathematical derivation assumes linear activations)

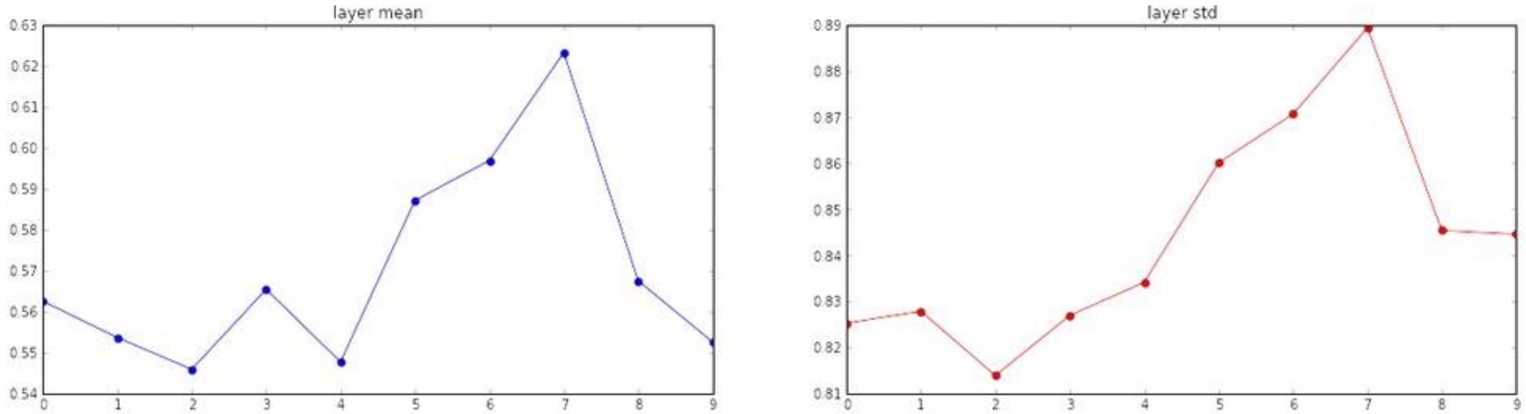
$$\mathcal{W}_l \sim \mathcal{N}(0, \sqrt{\frac{1}{n_l}})$$

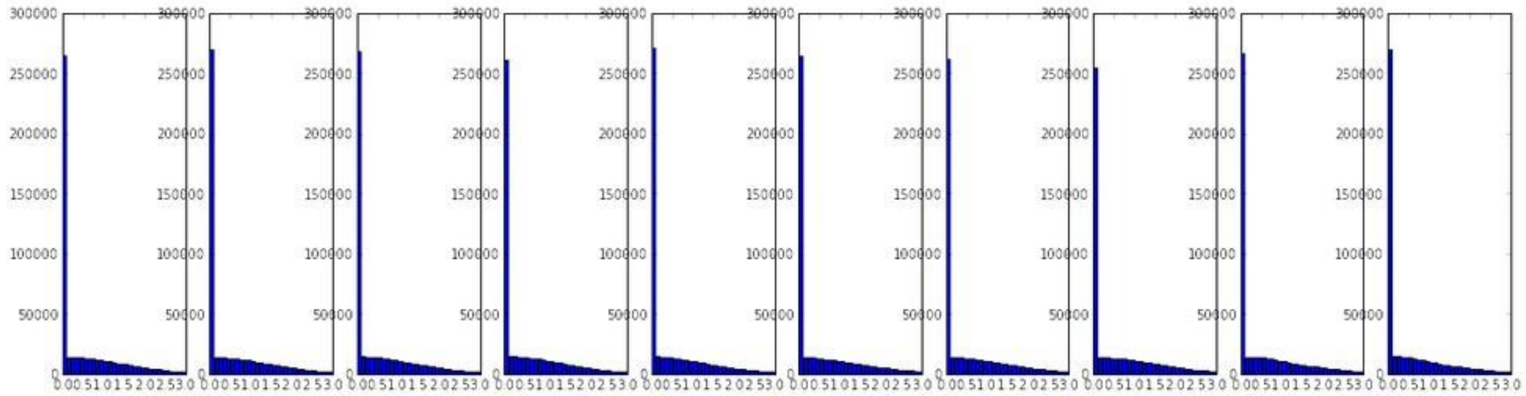
$$b_l \sim \mathcal{N}(0, \sqrt{\frac{1}{n_l}})$$

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```

```python
self.W = torch.randn(fan_out, fan_in) / math.sqrt(fan_in/2)
self.b = torch.randn(fan_out) / math.sqrt(fan_in/2)
```

He et al., 2015
(note additional /2)





$$\mathcal{W}_l \sim \mathcal{N}(0, \sqrt{\frac{2}{n_l}})$$

$$b_l \sim \mathcal{N}(0, \sqrt{\frac{2}{n_l}})$$

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want unit gaussian activations? just make them so."

consider a batch of activations at some layer. To make each feature dimension unit gaussian, apply:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

Batch Normalization

[Ioffe and Szegedy, 2015]

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

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

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

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

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

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

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Babysitting the Learning Process

Notebook

https://github.com/stencilman/CS763_Spring2017/blob/master/Lec3%2C4/CrossEntropy-Linear.ipynb

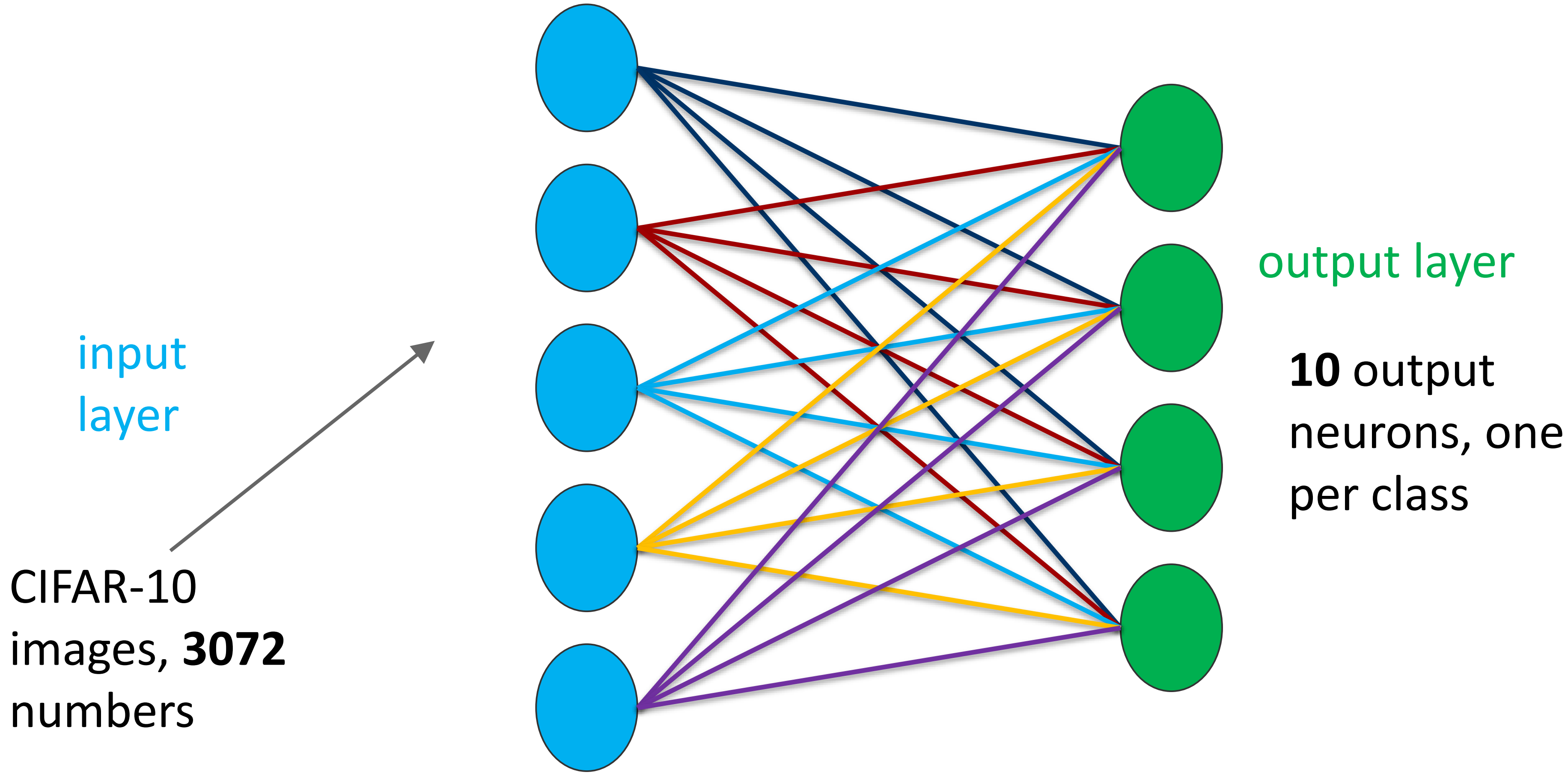
2. Data Preprocessing: We compute the mean and standard deviation 'images' and then subtract and divide by the same respectively (like AlexNet). We also visualize them.

In [3]:
```
x_mean = torch.mean(tr_x:float(), 1)
x_std = torch.std(tr_x:float(), 1)
itorch.image(x_mean)
itorch.image(x_std)
```

In [7]:
```
function get_xi(data_x, idx)
    xi = (data_x[idx]:float() - x_mean)
    xi = xi:cdiv(x_std)
    xi = xi:reshape(3*32*32)
    return xi
end
```

Step 2: Choose the architecture:

Say we start with single layer network:



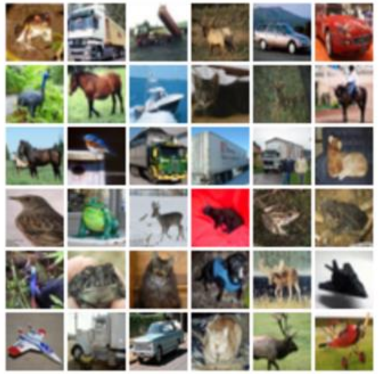
1. Data Loading: Let us load the training and the test data and check the size of the tensors. Let us also display the first few images from the training set.

In [1]:
```
-- load trainin images
tr_x = torch.load('cifar10/tr_data.bin')
-- load trainin labels
tr_y = torch.load('cifar10/tr_labels.bin'):double() + 1
-- load test images
te_x = torch.load('cifar10/te_data.bin')
-- load test labels
te_y = torch.load('cifar10/te_labels.bin'):double() + 1
print(tr_x:size())
print(tr_y:size())
```

Out[1]:
```
 50000
     3
    32
    32
[torch.LongStorage of size 4]

 50000
[torch.LongStorage of size 1]
```

In [2]:
```
-- display the first 36 training set images
require 'image';
itorch.image(tr_x[{{1,36},{},{},{}}])
```



Double check that the loss is reasonable:

```
op = model:forward(xi)
loss_tr = criterion:forward(op, ti)
print(loss_tr)
```

```
-- run it
lr = 0.00001
lambda = 0.0
train_and_test_loop(1, lr, lambda)
```

disable regularization

Print Loss

```
Out[11]: 2.2656910718829
```

loss ~2.3.
"correct " for
10 classes

Double check that the loss is reasonable:

```
op = model:forward(xi)
loss_tr = criterion:forward(op, ti,
print(loss_tr)
```

```
-- run it
lr = 0.00001
lambda = 1e3
train_and_test_loop(1, lr, lambda)
```

Crank it way up regularization

```
Out[12]: 12.582525307612
```

Print Loss

loss went up, good. (sanity check)

Lets try to train now…

Tip: Make sure that you can overfit very small portion of the training data

```
-- run it
lr = 0.0001
lambda = 0
train_and_test_loop(100000, lr, lambda)
```

The above code:

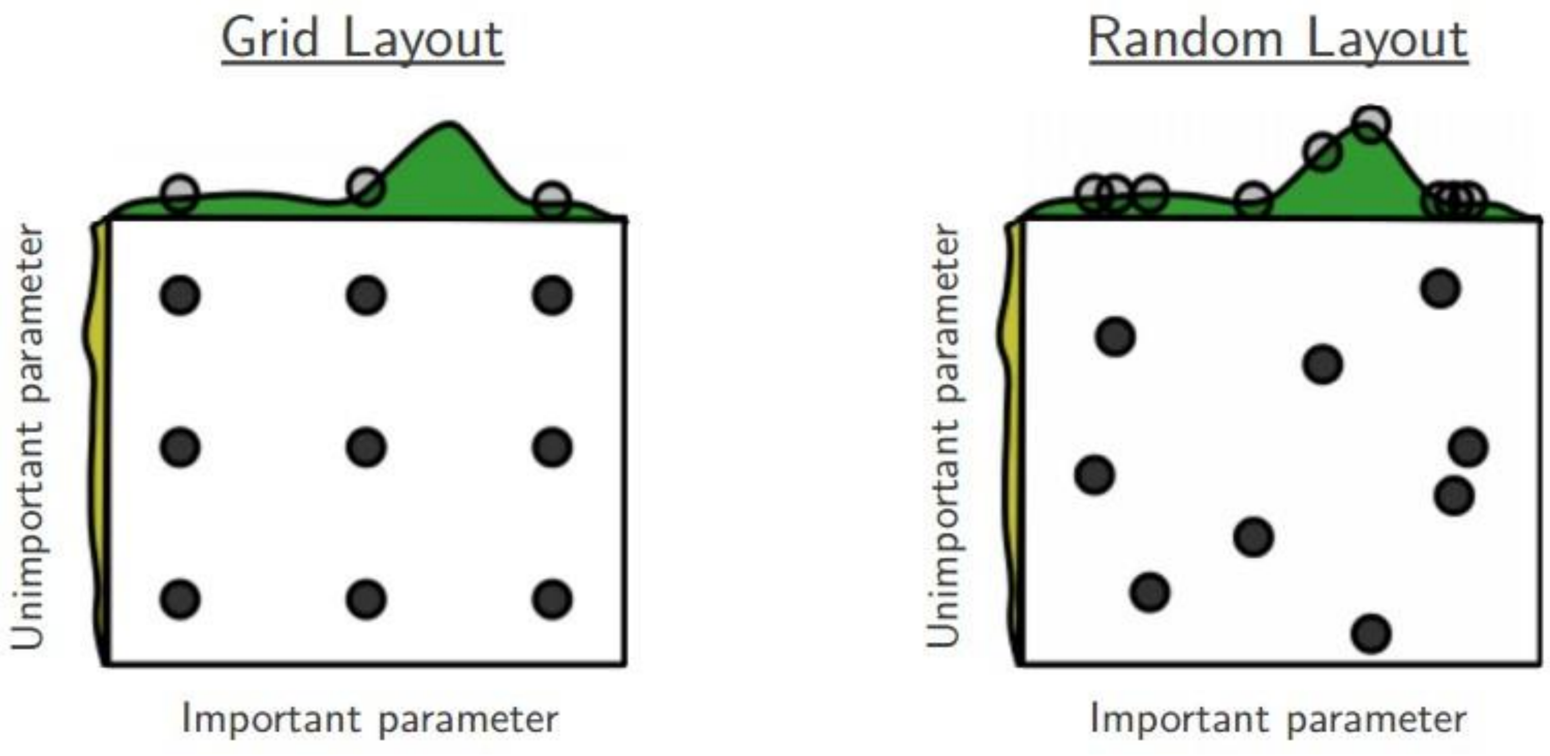
- take the first 20 examples from CIFAR-10
- turn off regularization (reg = 0.0)
- use simple vanilla ‘sgd’

```
tr_x = tr_x[{{1,20},{},{},{}}]
te_x = tr_x[{{1,20},{},{},{}}]
tr_y = tr_y[{{1,20}}]
print(tr_x:size())
print(tr_y:size())
```

```
Out[14]:  20
           3
          32
          32
         [torch.LongStorage of size 4]

          20
         [torch.LongStorage of size 1]
```

Random Search vs. Grid Search



Random Search for Hyper-Parameter Optimization

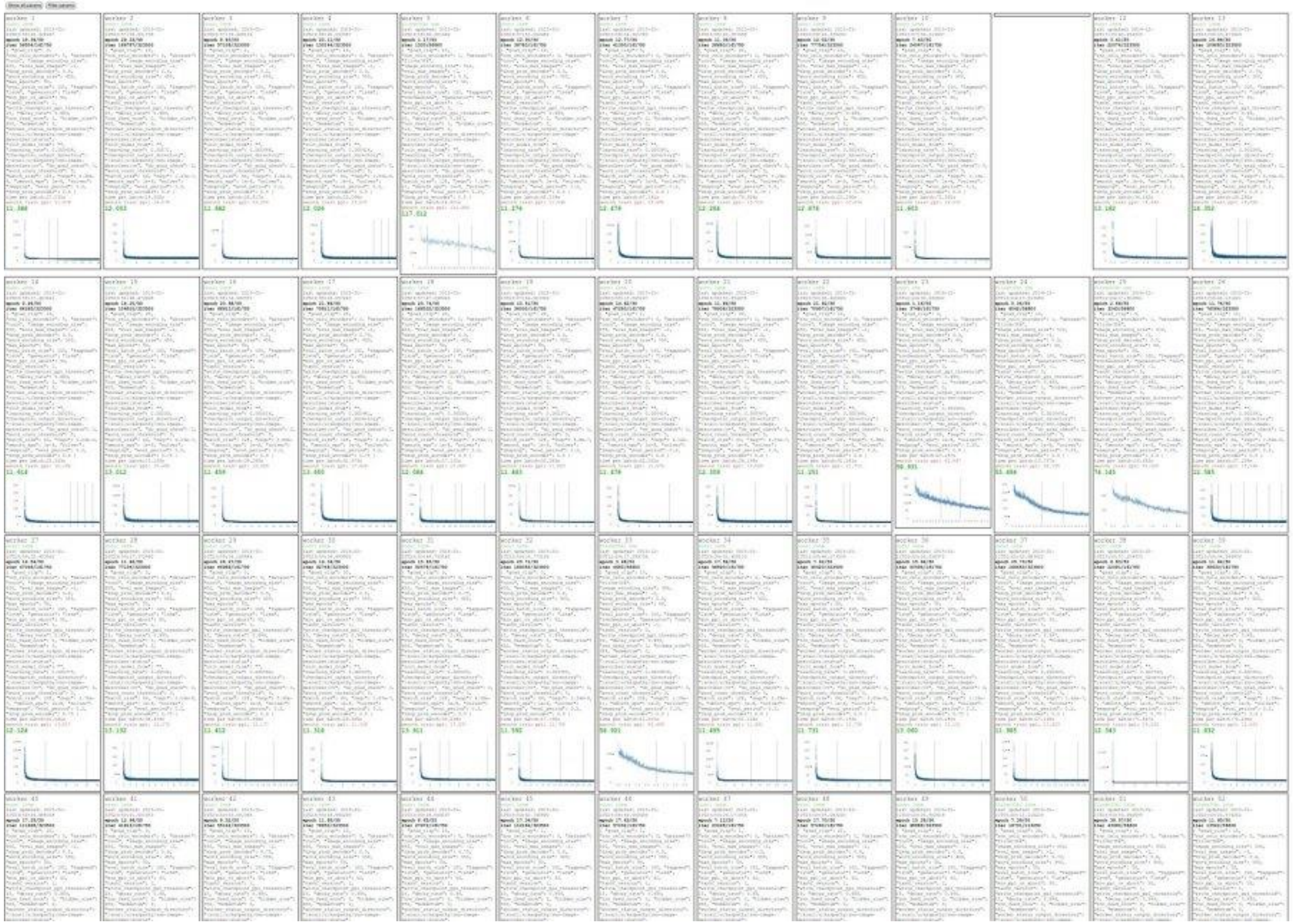
Bergstra and Bengio, 2012

Hyperparameters to play with:

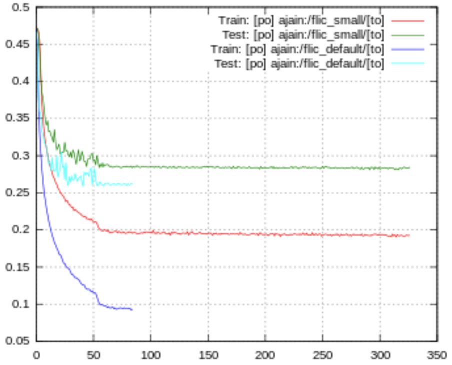
- network architecture
- learning rate, its multiplier schedule
- regularization (L2/Dropout strength)

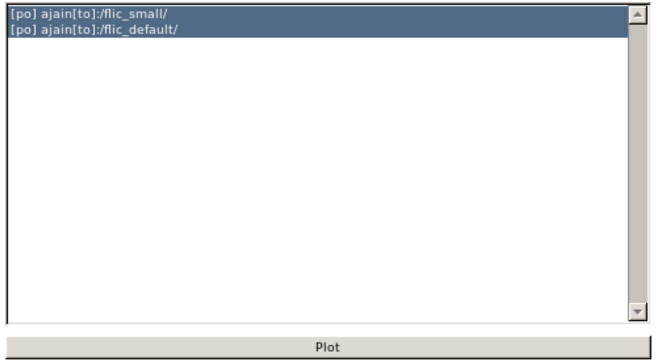
neural networks practitioner
music = loss function

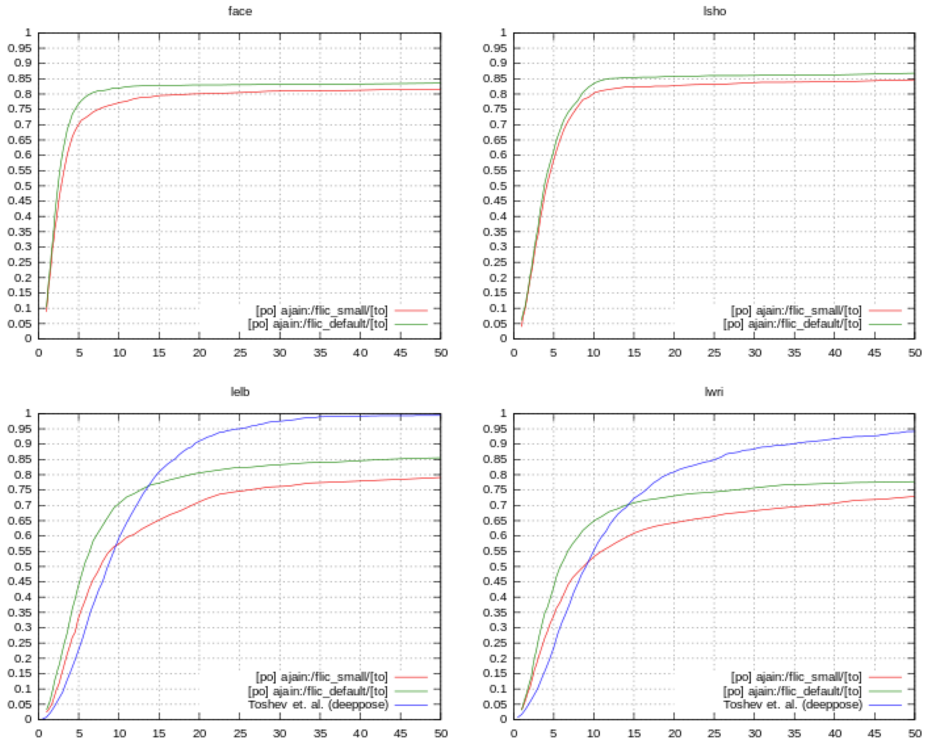
Karpathy’s cross-validation “command center”



My cross-validation "command center"







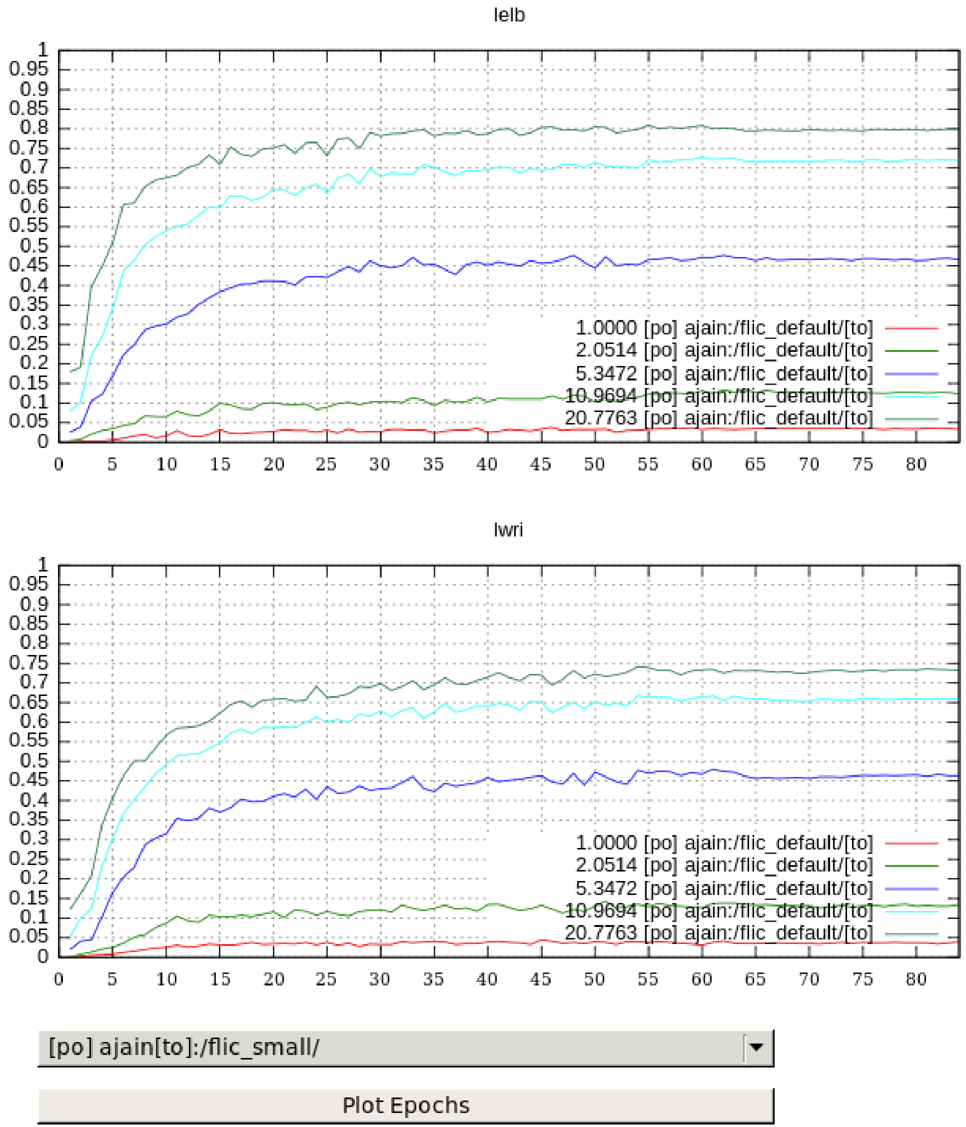
[po] ajain:/flic_small/[to]

```
2  batch_normalization = false
3  batch_size = 16
4  batch_size_per_gpu = 16
5  big_model = false
6  body_scale_range = {}
7  compress_src_model =
8  conv_1x1_size = 1
9  conv_nfeats = { 1 = { 1 = 16, 2 = 16, 3 = 16, }, 2 = { 1 = 16, 2 = 16, 3 = 16, }, 3 = { 1 = 16, 2 = 16, 3 = 16, },}
10 conv_pool = { 1 = { 1 = 2, 2 = 2, 3 = 1, }, 2 = { 1 = 2, 2 = 2, 3 = 1, }, 3 = { 1 = 2, 2 = 2, 3 = 1, },}
11 conv_size = { 1 = { 1 = 5, 2 = 5, 3 = 5, }, 2 = { 1 = 5, 2 = 5, 3 = 5, }, 3 = { 1 = 5, 2 = 5, 3 = 5, },}
12 crop_gradOutput = false
```

[po] ajain:/flic_default/[to]

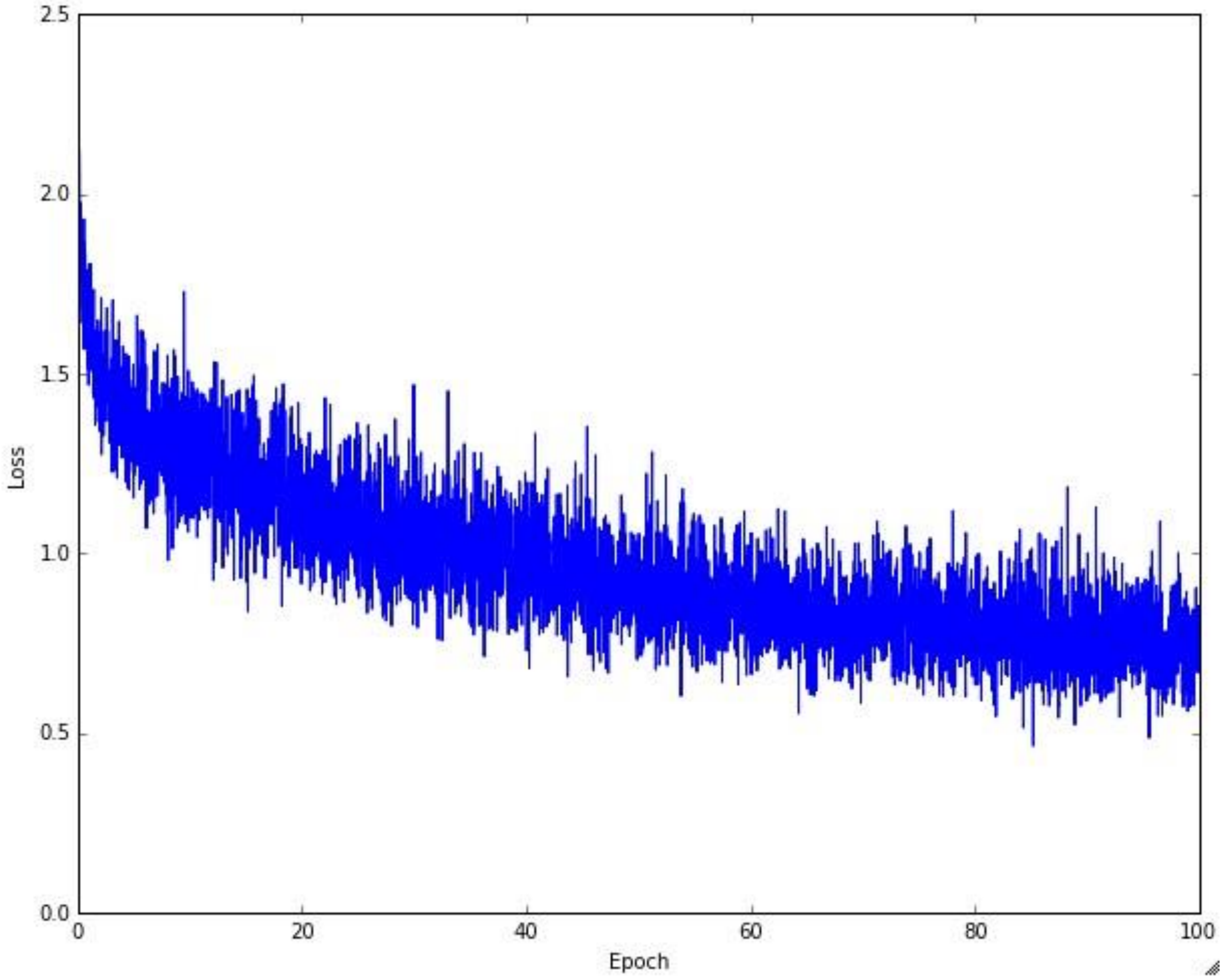
```
2  batch_normalization = false
3  batch_size = 16
4  batch_size_per_gpu = 16
5  big_model = true
6  body_scale_range = {}
7  compress_src_model =
8  conv_1x1_size = 1
9  conv_nfeats = { 1 = { 1 = 128, 2 = 128, 3 = 128, }, 2 = { 1 = 128, 2 = 128, 3 = 128, }, 3 = { 1 = 128, 2 = 128, 3 = 128, },}
10 conv_pool = { 1 = { 1 = 2, 2 = 2, 3 = 1, }, 2 = { 1 = 2, 2 = 2, 3 = 1, }, 3 = { 1 = 2, 2 = 2, 3 = 1, },}
11 conv_size = { 1 = { 1 = 5, 2 = 5, 3 = 5, }, 2 = { 1 = 5, 2 = 5, 3 = 5, }, 3 = { 1 = 5, 2 = 5, 3 = 5, },}
12 crop_gradOutput = false
```

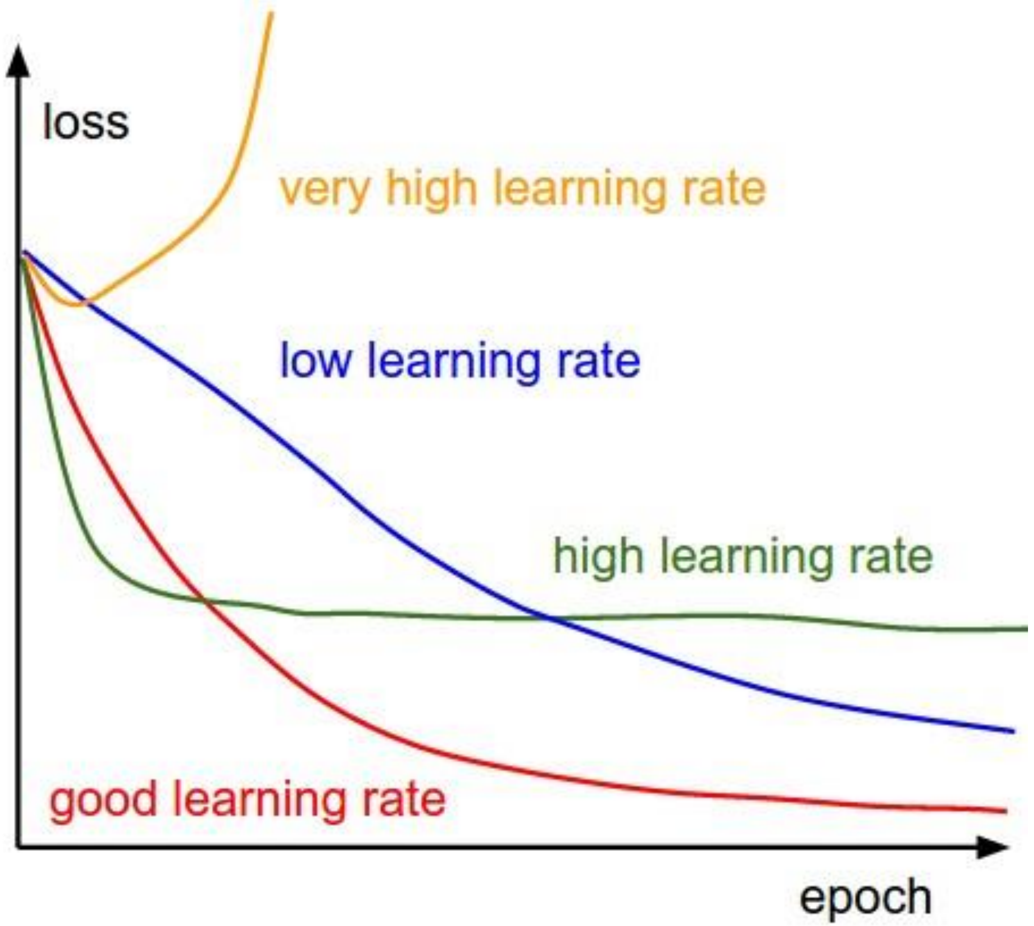
My cross-validation “command center”

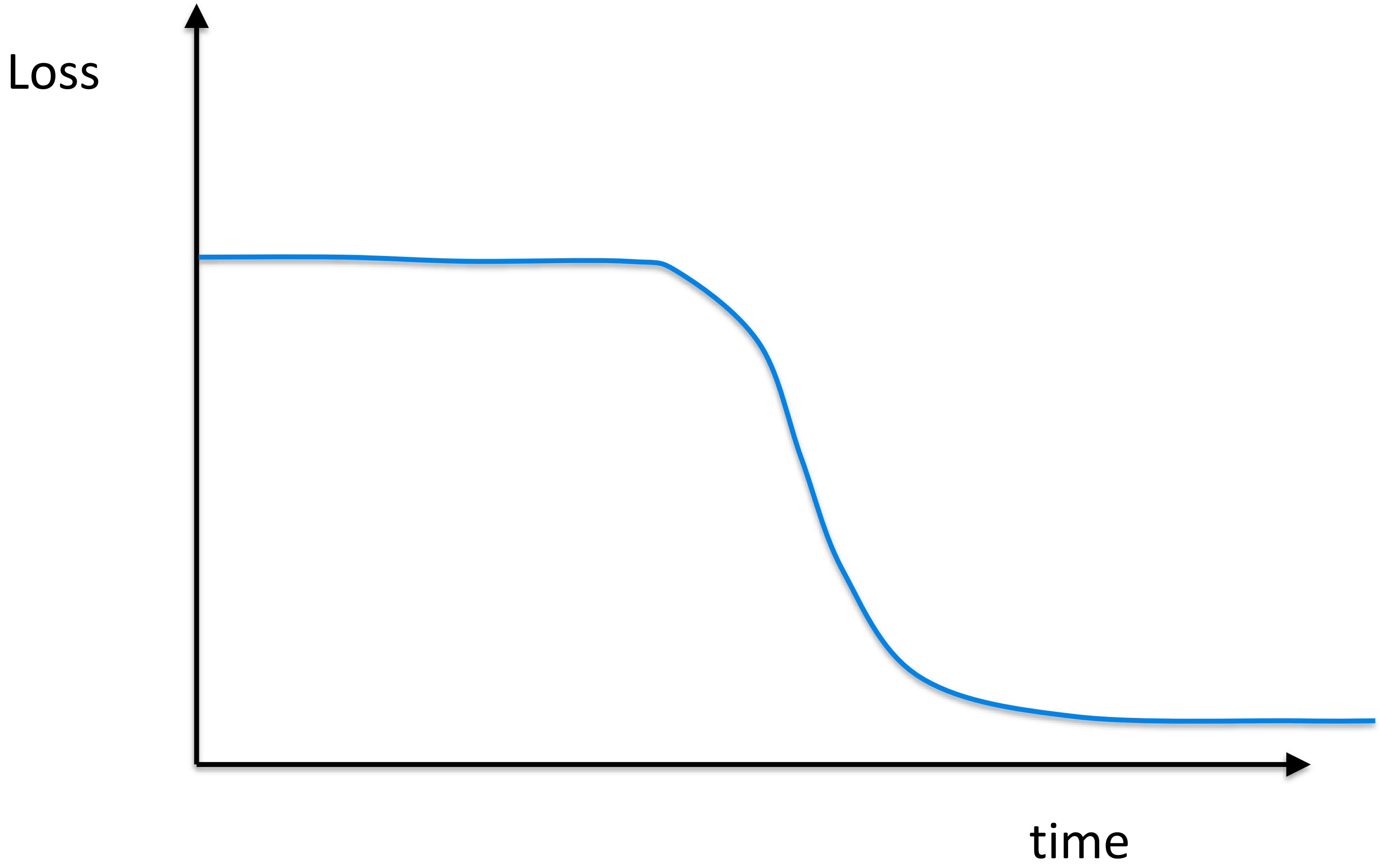


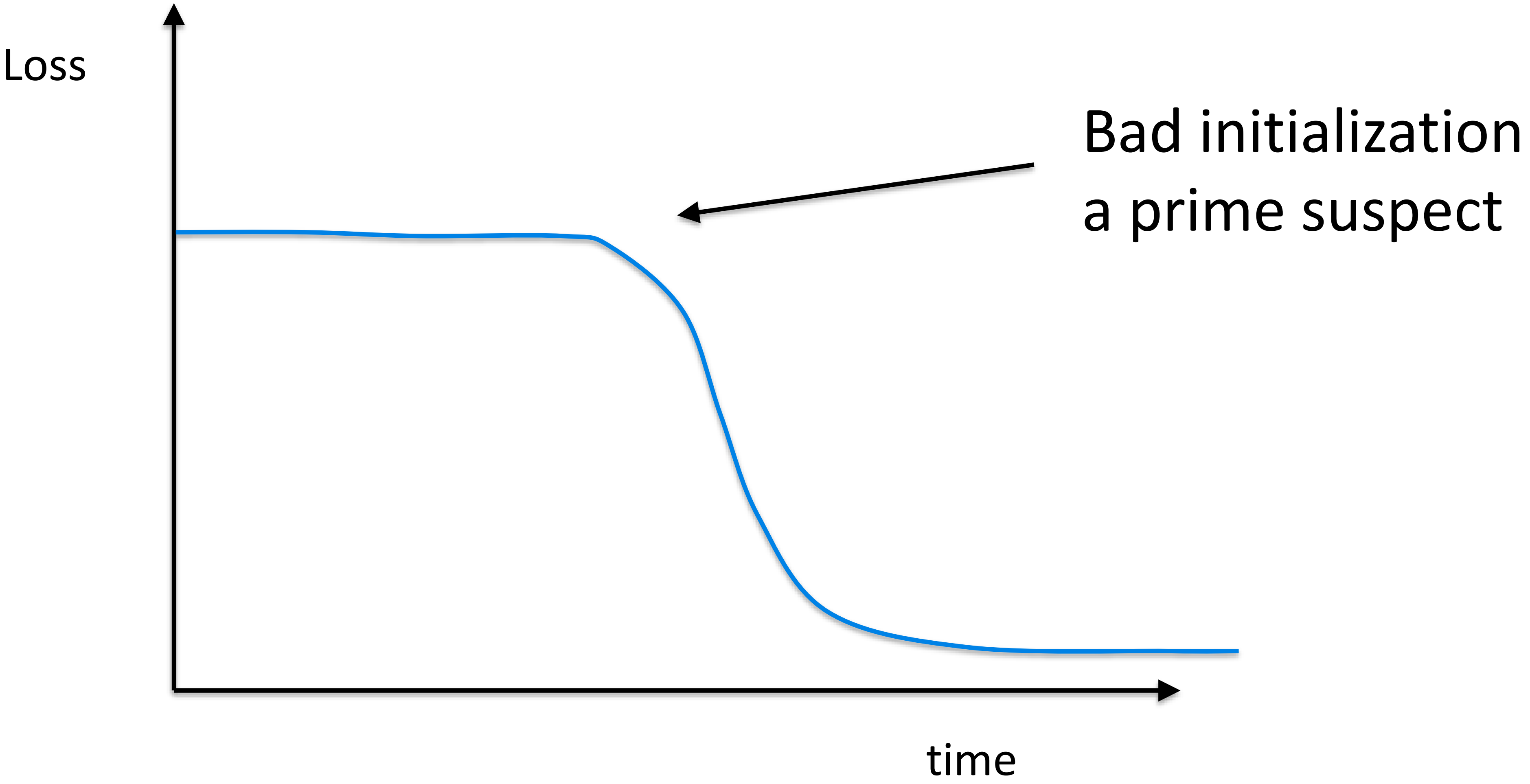
My cross-validation "command center"

Monitor and visualize the loss curve

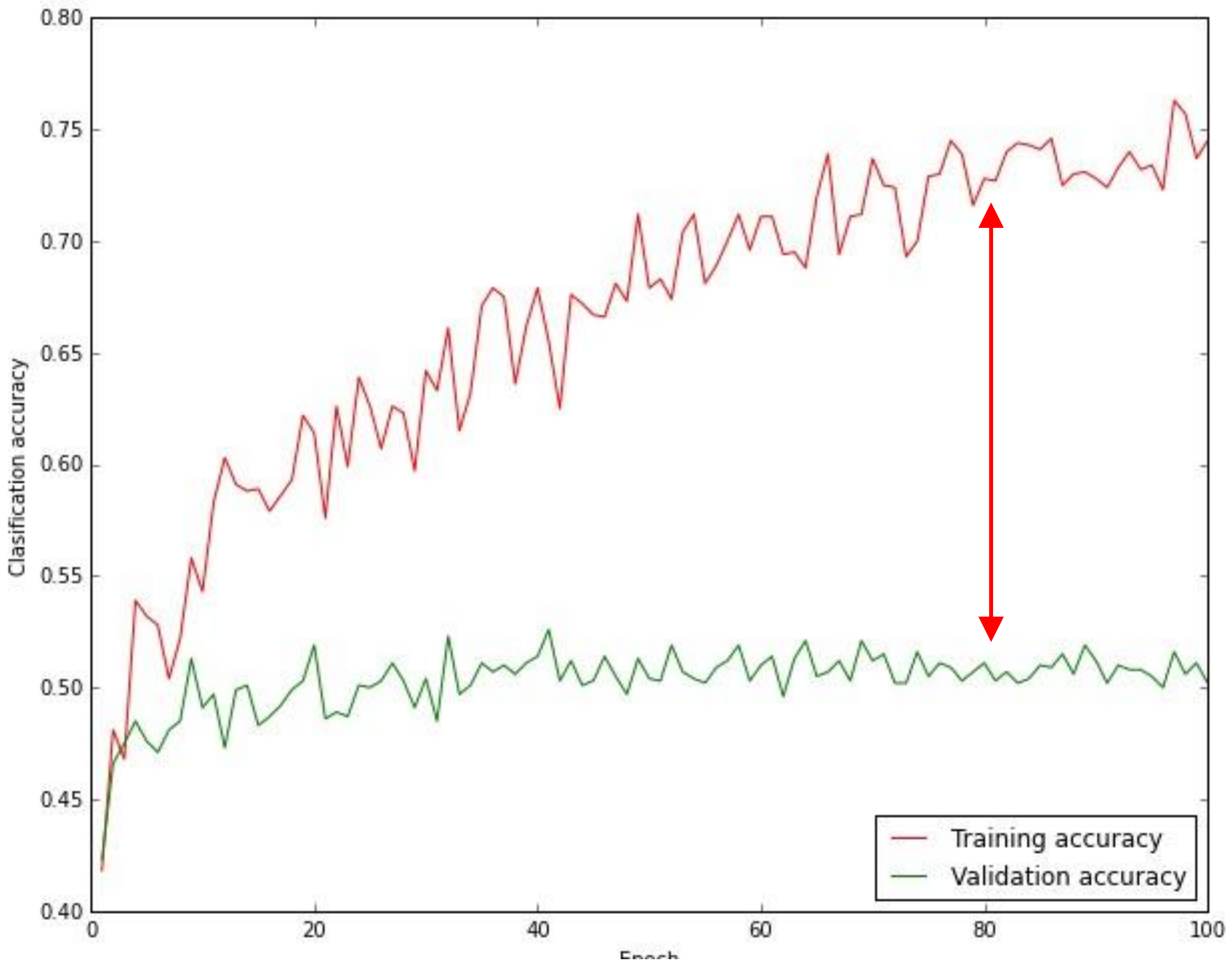





Loss
time


Loss
Bad initialization
a prime suspect
time

Monitor and visualize the accuracy:



big gap = overfitting

=> increase regularization strength?

no gap

=> increase model capacity?

Track the ratio of weight updates / weight magnitudes:

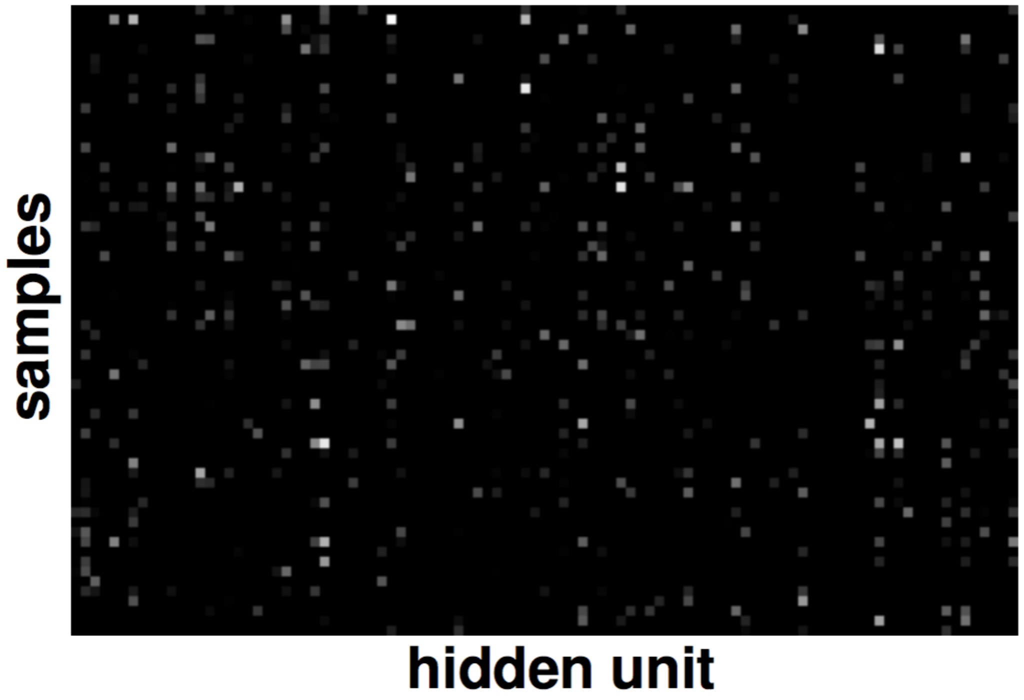
```
function gradient_descent(model, lr)
    w_scale = torch.norm(model.W:view(model.W:nElement()), 2, 1)
    update_scale = torch.norm(lr * model.gradW:view(model.gradW:nElement()), 2, 1)
    model.W = model.W + lr * model.gradW
    model.b = model.b + lr * model.gradb
    print(update_scale/w_scale) -- Want ~1e-3
end
```

ratio between the values and updates: ~ 0.0002 / 0.02 = 0.01 (about okay)

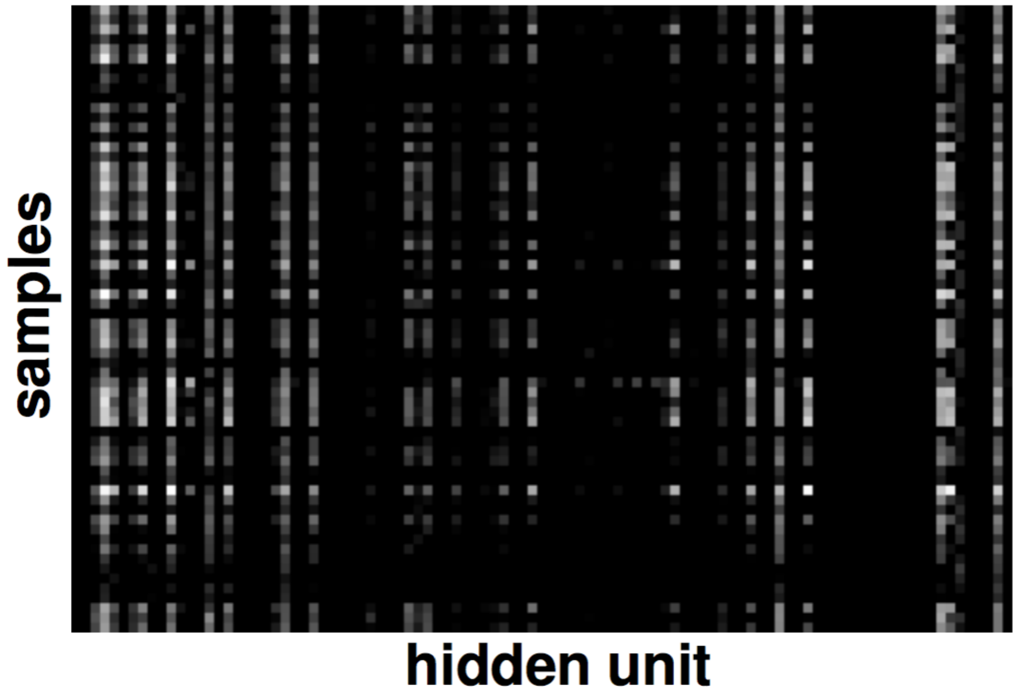
want this to be somewhere around 0.001 or so

Visualize Activations

- Visualize features (feature maps need to be uncorrelated) and have high variance.



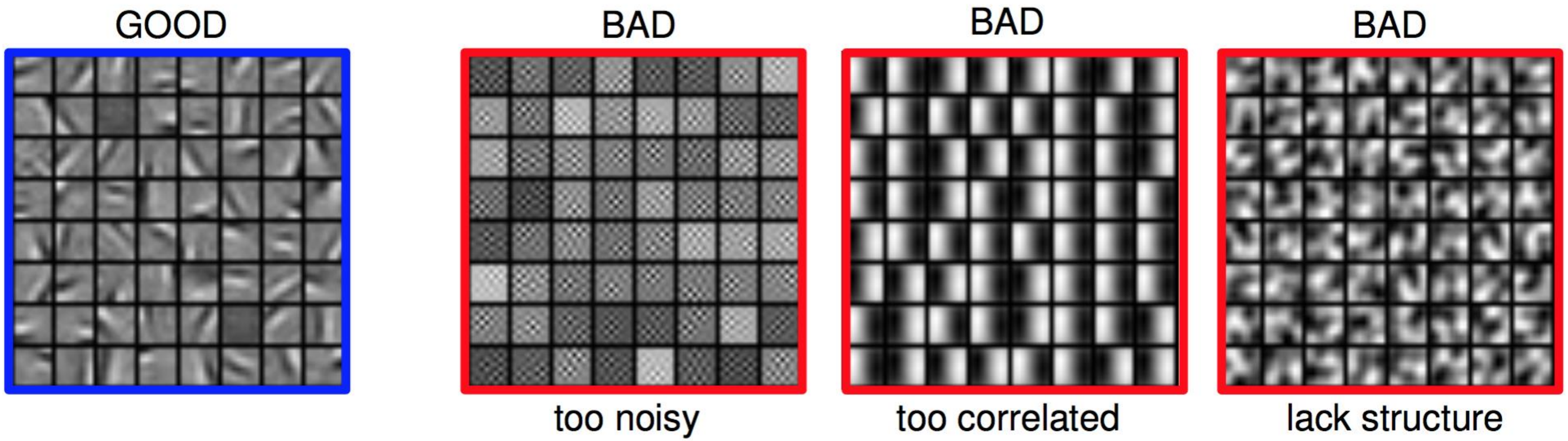
Good training: hidden units are sparse across samples and across features.



Bad training: many hidden units ignore the input and/or exhibit strong correlations.

Visualize (initial) Convolution Layer Weights

- Visualize features (feature maps need to be uncorrelated) and have high variance.



Good training: learned filters exhibit structure and are uncorrelated.

Visualize Linear Layer (Fully-Connected) Weights

- Visualization of Linear layer weights for some networks
- It has a banded structure repeated 28 times (Why?!) Hint: Images are 28x28
- Thus, looking at the weights we get some intuition