Dall-E

CNNs and Computer Vision

Applied Machine Learning
Derek Hoiem

Week	Date	Topic	Link	Reading/Notes
1	Jan 17 (Tues)	Introduction	ppt ; pdf	Jupyter notebook tutorial vid ipynb cc Numpy tutorial vid cc Linear algebra tutorial vid cc
		Supervised Learning Fundamentals		
1	Jan 19 (Thurs)	KNN, key concepts in ML	ppt ; pdf	AML Ch 1
2	Jan 24 (Tues)	Probability and Naïve Bayes	ppt ; pdf	AML Ch 1
2	Jan 26 (Thurs)	Linear Least Squares and Logistic Regression	ppt ; pdf	AML 10.1-10.2, 11
3	Jan 31 (Tues)	Decision Trees	ppt ; pdf	AML Ch 2
3	Feb 2 (Thurs)	Consolidation and Review	ppt ; pdf	
	Feb 6 (Mon)	*HW 1 (Classification & Regression) due*		
4	Feb 7 (Tues)	Ensembles and Random Forests	ppt ; pdf	AML Ch 2, Ch 12
4	Feb 9 (Thurs)	SVMs and SGD	ppt ; pdf	AML Ch 2
5	Feb 14 (Tues)	MLPs and Backprop	ppt ; pdf	AML Ch 16
5	Feb 16 (Thurs)	Deep Learning	ppt ; pdf	AML Ch 16; ResNet (He et al. 2016)
6	Feb 21 (Tues)	Consolidation and Review	ppt ; pdf	
		Vision, Language, and Applications		
6	Feb 23 (Thurs)	CNNs in Computer Vision		AML Ch 17-18
7	*Feb 27 (Mon)*	*HW 2 (Trees & MLPs) due*		
7	Feb 28 (Tues)	Language Models		
7	Mar 2 (Thurs)	Transformers in Language and Vision		
8	Mar 7 (Tues)	Foundation Models: CLIP and GPT-3		
8	*Mar 9 (Thurs)*	*Exam 1 (on PrairieLearn)*		
9	*Mar 11-19*	*Spring Break (no classes)*		
10	Mar 21 (Tues)	Task Adaptation		
10	Mar 23 (Thurs)	Fairness and impact on society		
11	*Mar 27 (Mon)*	*HW 3 (Application Domains) due*		
11	Mar 28 (Tues)	Big Data and Dataset Bias		
		Pattern Discovery		
11	Mar 30 (Thurs)	K-Means, KD-tree, LSH		AML Ch 8
12	Apr 4 (Tues)	Missing Data and EM		AML Ch 9
12	Apr 6 (Thurs)	Density estimation: MoG, Kernels, Hists		AML Ch 9
13	Apr 11 (Tues)	Data visualization: PCA and t-SNE		AML Ch 11
13	Apr 13 (Thurs)	Topic Modeling		
14	*Apr 17 (Mon)*	*HW 4 (Pattern Discovery) due*		
14	Apr 18 (Tues)	CCA		AML Ch6, Ch 19
		More Applications and Topics		
14	Apr 20 (Thurs)	Audio		Audio Deep Learning
15	Apr 25 (Tues)	TBD		
16	Apr 27 (Thurs)	Machine Learning: Practice vs. Theory		
16	May 2 (Tues)	Looking Forward & Requested Topics		

You are here!

Today's Lecture

- ImageNet Challenge Overview
- ResNet model in more detail
- Adapting a pre-trained network to new tasks
- Mask-RCNN line of detection/segmentation
- U-Net Architecture

IMAGENET

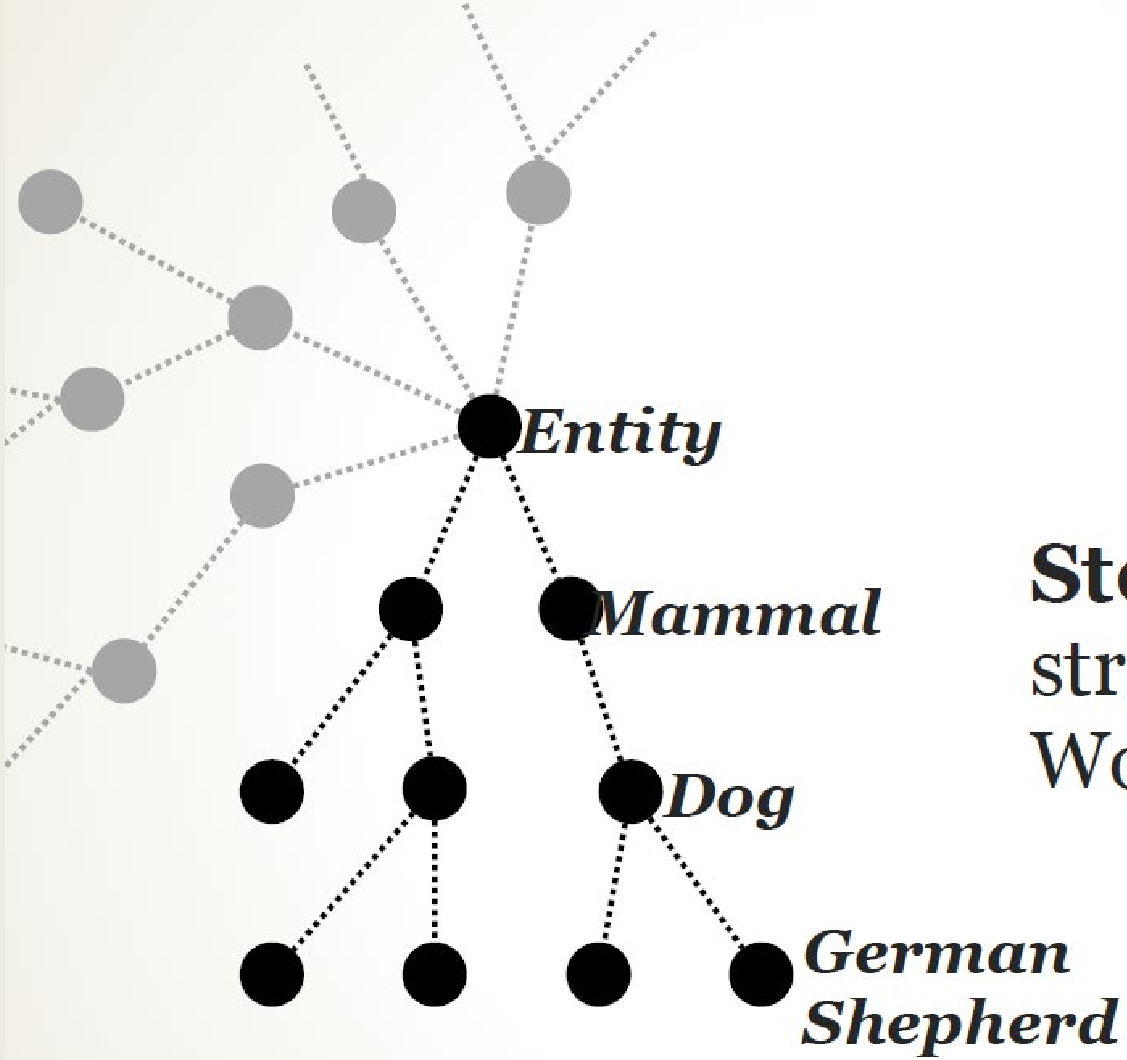
22K categories and **15M** images

- Animals
  - Bird
  - Fish
  - Mammal
  - Invertebrate
- Plants
  - Tree
  - Flower
- Food
- Materials
- Structures
- Artifact
  - Tools
  - Appliances
  - Structures
- Person
- Scenes
  - Indoor
  - Geological Formations
- Sport Activity

www.image-net.org

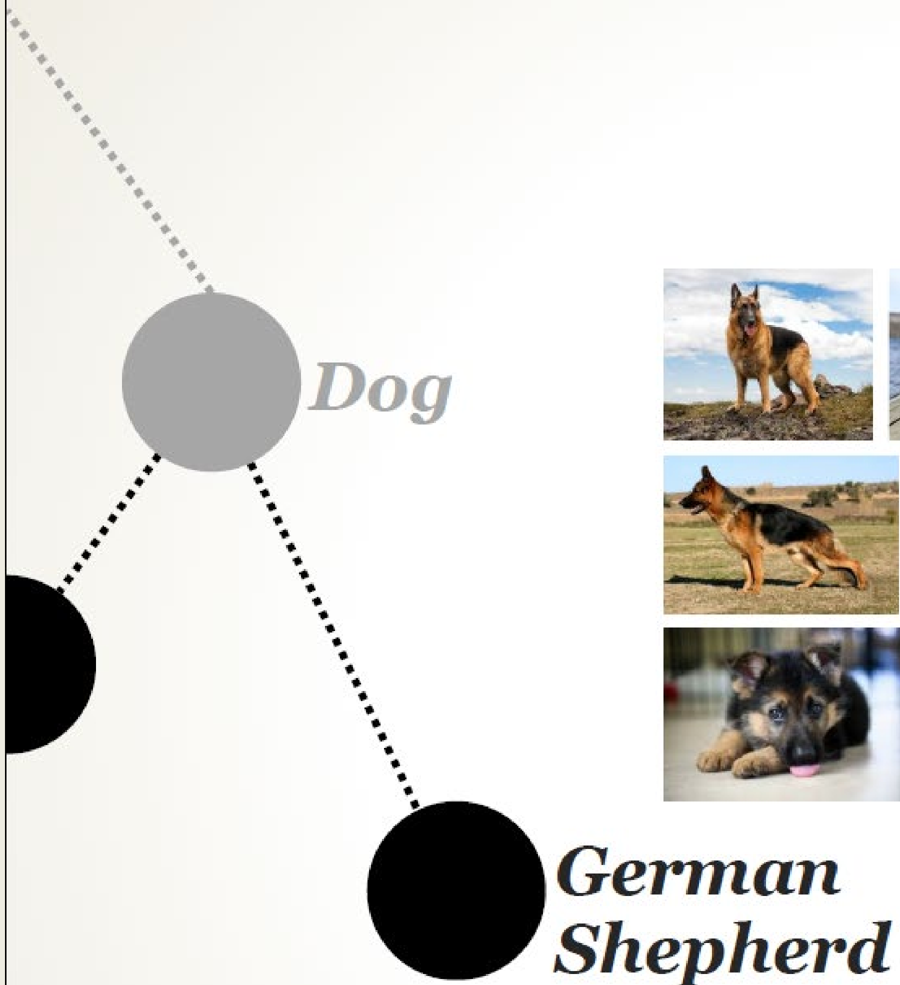
Deng et al. 2009,
Russakovsky et al. 2015

Slide credit: Fei-Fei Li



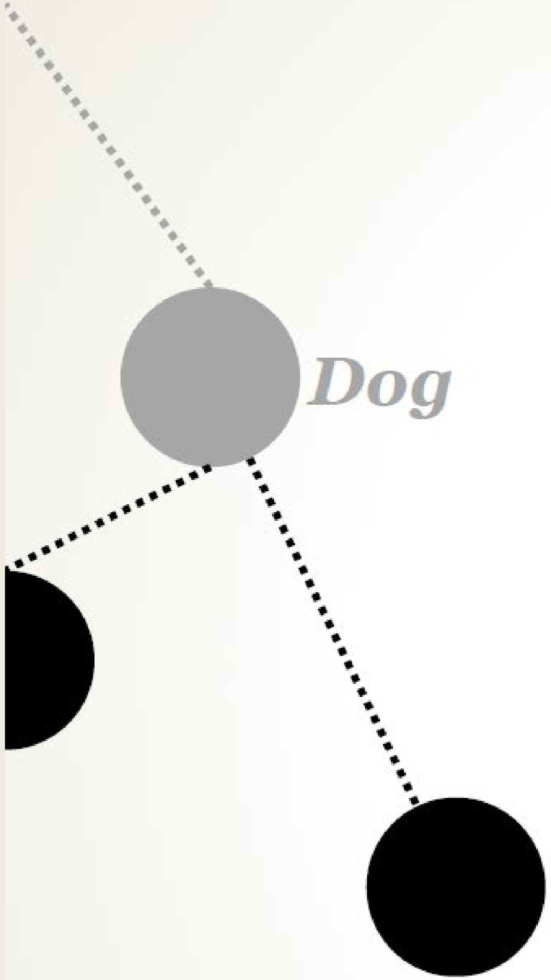
Step 1: Ontological structure based on WordNet

Slide credit: Fei-Fei Li and Jia Deng



Step 2: Populate categories with thousands of images from the Internet

Slide credit: Fei-Fei Li and Jia Deng



Dog

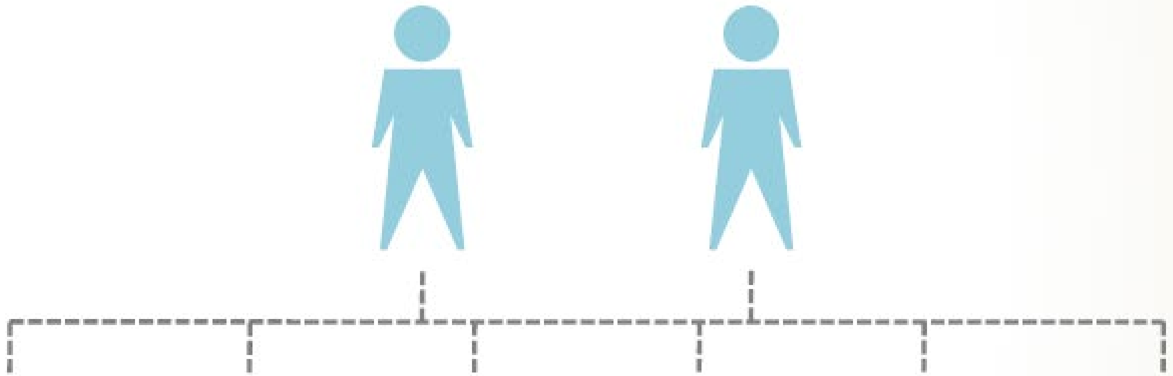
German Shepherd

Step 3: Clean results by hand

Slide credit: Fei-Fei Li and Jia Deng

Crowdsourcing

ImageNet PhD Students



Crowdsourced Labor

amazon mechanical turk™

Artificial Artificial Intelligence

49k Workers *from* 167 Countries 2007-2010

Slide credit: Fei-Fei Li and Jia Deng

ILSVRC image classification task

Steel drum

Output:
Scale
T-shirt
<u>Steel drum</u>
Drumstick
Mud turtle

✔

Output:
Scale
T-shirt
Giant panda
Drumstick
Mud turtle

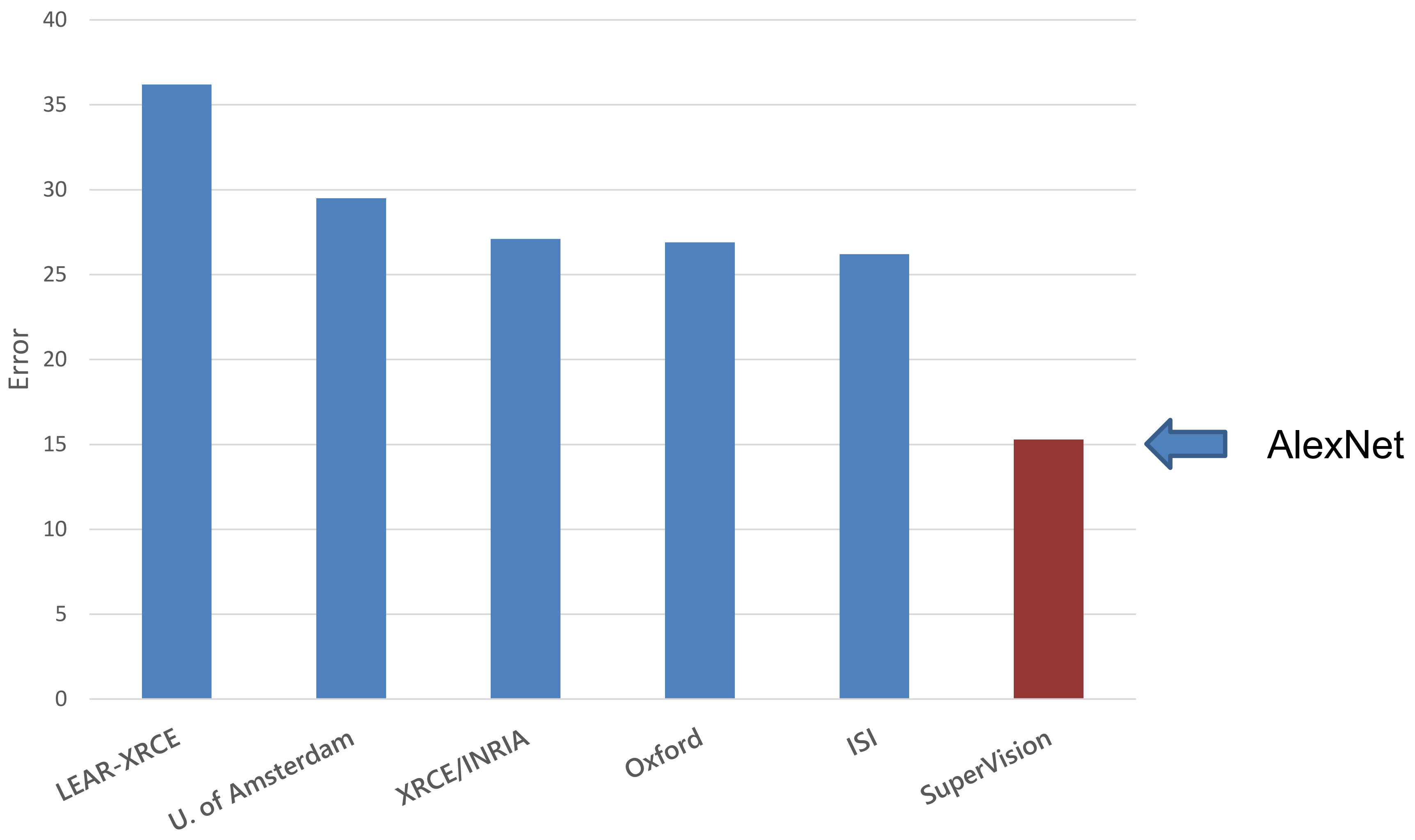
✗

$$\text{Error} = \frac{1}{100{,}000} \sum_{\substack{100{,}000 \\ \text{images}}} 1[\text{incorrect on image i}]$$

Slide source

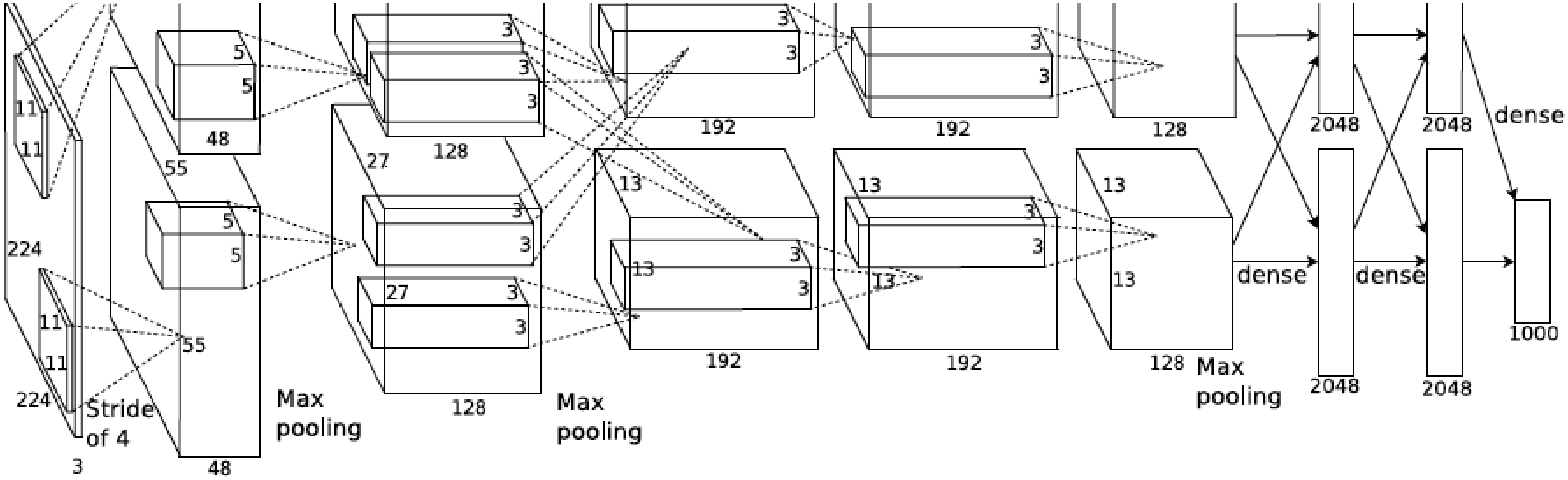
2012 ImageNet 1K

(Fall 2012)



Slide: Jia-bin Huang

AlexNet: ILSVRC 2012 winner

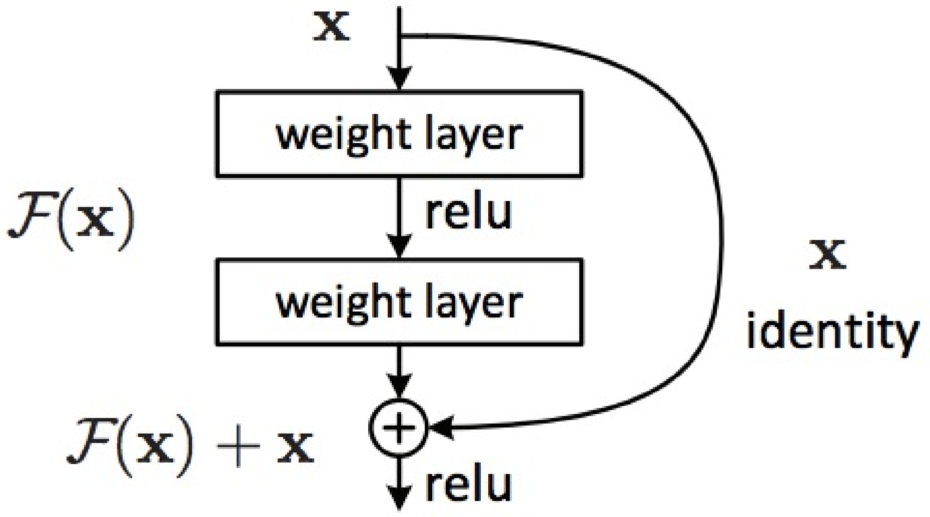


- Similar framework to LeNet but:
  - Max pooling, **ReLU nonlinearity**
  - **More data** and **bigger model** (7 hidden layers, 650K units, 60M params)
  - GPU implementation (**50x speedup** over CPU)
    - Trained on two GPUs for a week
  - Dropout regularization

A. Krizhevsky, I. Sutskever, and G. Hinton, [ImageNet Classification with Deep Convolutional Neural Networks], NIPS 2012

ResNet: the residual module

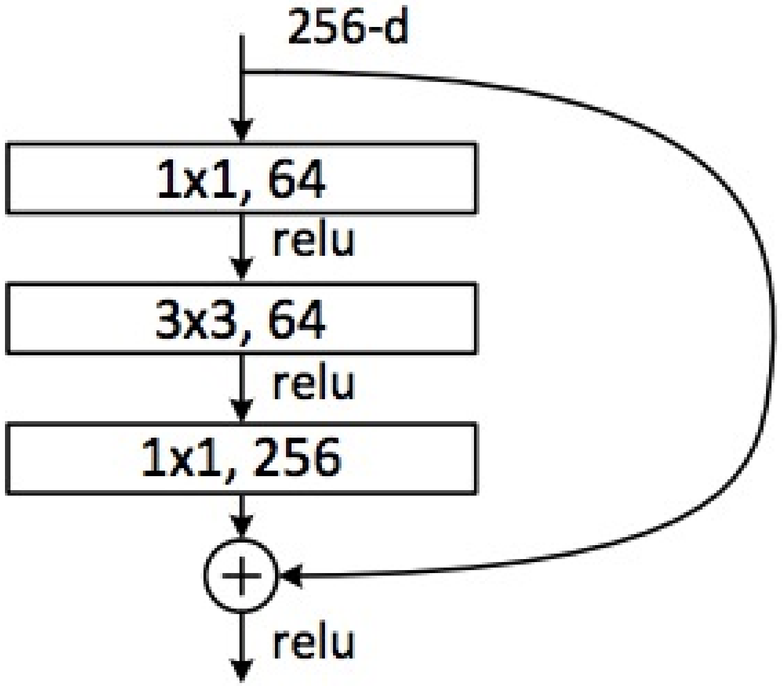
- Use *skip* or *shortcut* connections around 2-3 layer MLPs
- Gradients can flow quickly back through skip connections
- Each module needs only add information to the previous layers



Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, [Deep Residual Learning for Image Recognition], CVPR 2016 (Best Paper), 150K+ citations

ResNet: Residual Bottleneck Module

Used in 50+ layer networks

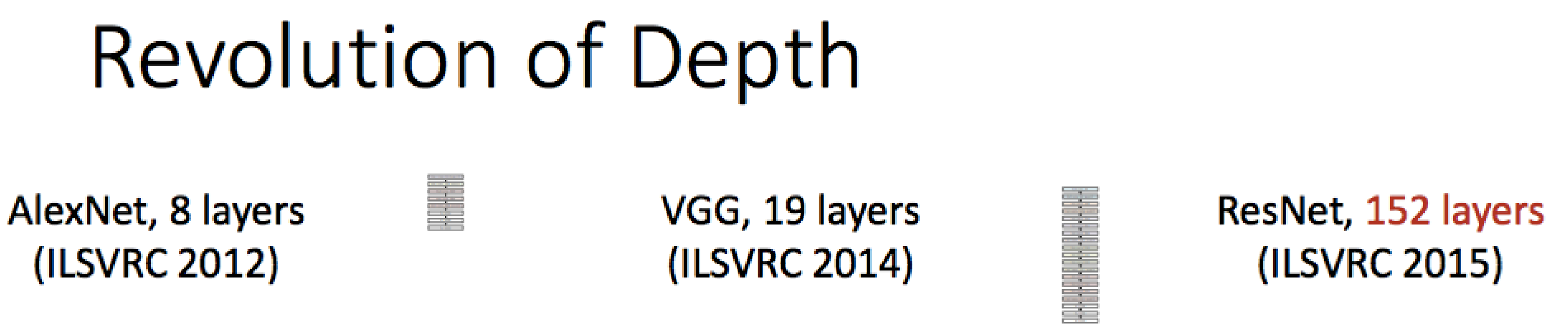


- Directly performing 3x3 convolutions with 256 feature maps at input and output: 256 x 256 x 3 x 3 ~ 600K operations
- Using 1x1 convolutions to reduce 256 to 64 feature maps, followed by 3x3 convolutions, followed by 1x1 convolutions to expand back to 256 maps:
  256 x 64 x 1 x 1 ~ 16K
  64 x 64 x 3 x 3 ~ 36K
  64 x 256 x 1 x 1 ~ 16K
  Total: ~70K

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, [Deep Residual Learning for Image Recognition](), CVPR 2016 (Best Paper)

Slide: Lazebnik

ResNet: going real deep



Despite depth, the residual connections enable error gradients to “skip” all the way back to the beginning

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, Deep Residual Learning for Image Recognition, CVPR 2016

ResNet CNN Components

- Conv2d: learned 2D convolutional filters (same linear weights applied to a patch surrounding each pixel)
- BatchNorm2D: Convolutional batch normalization (see next slide)
- ReLU: non-linearity with gradient={0,1}
- Linear layer: feature projection / final linear classifier

Batch Normalization

- During training, the feature distribution at intermediate layers keep changing as the network learns
- This destabilizes training
- BatchNorm normalizes features of each mini-batch according to its mean and variance and learned parameters γ, β
- Using BatchNorm often improves speed and effectiveness of training

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

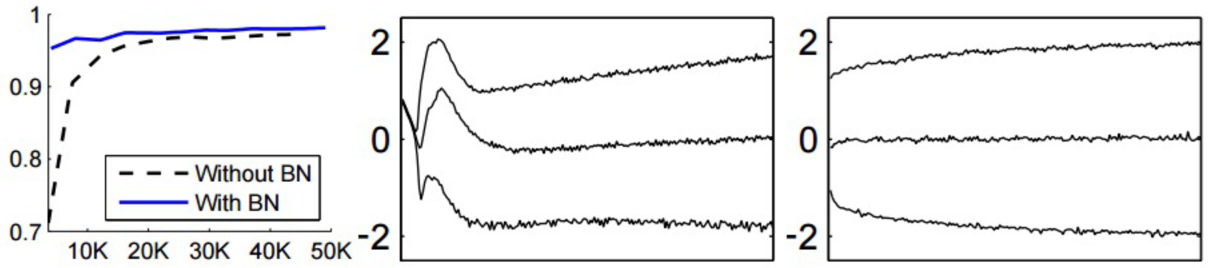
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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



(a) (b) Without BN (c) With BN

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift [Ioffe and Szegedy 2015]

Example code: ResBlock

"channels" = # feature maps
kernel_size = filter size, e.g. 3x3
stride = # pixels to skip when evaluating convolution
padding: to calculate filter values near edge of image/map

```
class ResBlock(nn.Module):
    def __init__(self, in_channels, out_channels, downsample):
        super().__init__()
        if downsample:
            self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=2, padding=1)
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=2),
                nn.BatchNorm2d(out_channels)
            )
        else:
            self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1)
            self.shortcut = nn.Sequential()

        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)

    def forward(self, input):
        shortcut = self.shortcut(input)
        input = nn.ReLU()(self.bn1(self.conv1(input)))
        input = nn.ReLU()(self.bn2(self.conv2(input)))
        input = input + shortcut
        return nn.ReLU()(input)
```

If downsampling, do it here too so dimensions match

This '+' is the skip connection!

Example code: ResNet-18 architecture for ImageNet

```
class Network(nn.Module):
    def __init__(self, num_classes=1000):
        super().__init__()
        resblock = ResBlock
        self.layer0 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU()
        )
        self.layer1 = nn.Sequential(
            resblock(64, 64, downsample=False),
            resblock(64, 64, downsample=False)
        )
        self.layer2 = nn.Sequential(
            resblock(64, 128, downsample=True),
            resblock(128, 128, downsample=False)
        )
        self.layer3 = nn.Sequential(
            resblock(128, 256, downsample=True),
            resblock(256, 256, downsample=False)
        )
        self.layer4 = nn.Sequential(
            resblock(256, 512, downsample=True),
            resblock(512, 512, downsample=False)
        )
        self.gap = torch.nn.AdaptiveAvgPool2d(1)
        self.fc = torch.nn.Linear(512, num_classes)
```

```
def forward(self, input):
    input = self.layer0(input)
    input = self.layer1(input)
    input = self.layer2(input)
    input = self.layer3(input)
    input = self.layer4(input)
    input = self.gap(input)
    input = torch.flatten(input, 1)
    input = self.fc(input)

    return input
```

[Pretrained Torch models]

ResNet Architectures and Results

<table>
<tr><th>layer name</th><th>output size</th><th>18-layer</th><th>34-layer</th><th>50-layer</th><th>101-layer</th><th>152-layer</th></tr>
<tr><td>conv1</td><td>112×112</td><td colspan="5">7×7, 64, stride 2</td></tr>
<tr><td rowspan="2">conv2_x</td><td rowspan="2">56×56</td><td colspan="5">3×3 max pool, stride 2</td></tr>
<tr><td>[3×3, 64
3×3, 64] ×2</td><td>[3×3, 64
3×3, 64] ×3</td><td>[1×1, 64
3×3, 64
1×1, 256] ×3</td><td>[1×1, 64
3×3, 64
1×1, 256] ×3</td><td>[1×1, 64
3×3, 64
1×1, 256] ×3</td></tr>
<tr><td>conv3_x</td><td>28×28</td><td>[3×3, 128
3×3, 128] ×2</td><td>[3×3, 128
3×3, 128] ×4</td><td>[1×1, 128
3×3, 128
1×1, 512] ×4</td><td>[1×1, 128
3×3, 128
1×1, 512] ×4</td><td>[1×1, 128
3×3, 128
1×1, 512] ×8</td></tr>
<tr><td>conv4_x</td><td>14×14</td><td>[3×3, 256
3×3, 256] ×2</td><td>[3×3, 256
3×3, 256] ×6</td><td>[1×1, 256
3×3, 256
1×1, 1024] ×6</td><td>[1×1, 256
3×3, 256
1×1, 1024] ×23</td><td>[1×1, 256
3×3, 256
1×1, 1024] ×36</td></tr>
<tr><td>conv5_x</td><td>7×7</td><td>[3×3, 512
3×3, 512] ×2</td><td>[3×3, 512
3×3, 512] ×3</td><td>[1×1, 512
3×3, 512
1×1, 2048] ×3</td><td>[1×1, 512
3×3, 512
1×1, 2048] ×3</td><td>[1×1, 512
3×3, 512
1×1, 2048] ×3</td></tr>
<tr><td></td><td>1×1</td><td colspan="5">average pool, 1000-d fc, softmax</td></tr>
<tr><td colspan="2">FLOPs</td><td>1.8×10^9</td><td>3.6×10^9</td><td>3.8×10^9</td><td>7.6×10^9</td><td>11.3×10^9</td></tr>
</table>

method	top-1 err.	top-5 err.
VGG [41] (ILSVRC'14)	-	8.43†
GoogLeNet [44] (ILSVRC'14)	-	7.89
VGG [41] (v5)	24.4	7.1
PReLU-net [13]	21.59	5.71
BN-inception [16]	21.99	5.81
ResNet-34 B	21.84	5.71
ResNet-34 C	21.53	5.60
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60
ResNet-152	**19.38**	**4.49**

Table 4. Error rates (%) of **single-model** results on the ImageNet validation set (except † reported on the test set).

(vs. 15% top-5 err for AlexNet)

Another common trick: “data augmentation”

- Randomly translate, crop, rotate, mirror, shift colors, or overlay images to create more variations
  - Apply random transformation to each sample in each batch as it is processed in training
  - Simulates a larger training set, and makes it so that the network will learn from variations of the original example in each epoch

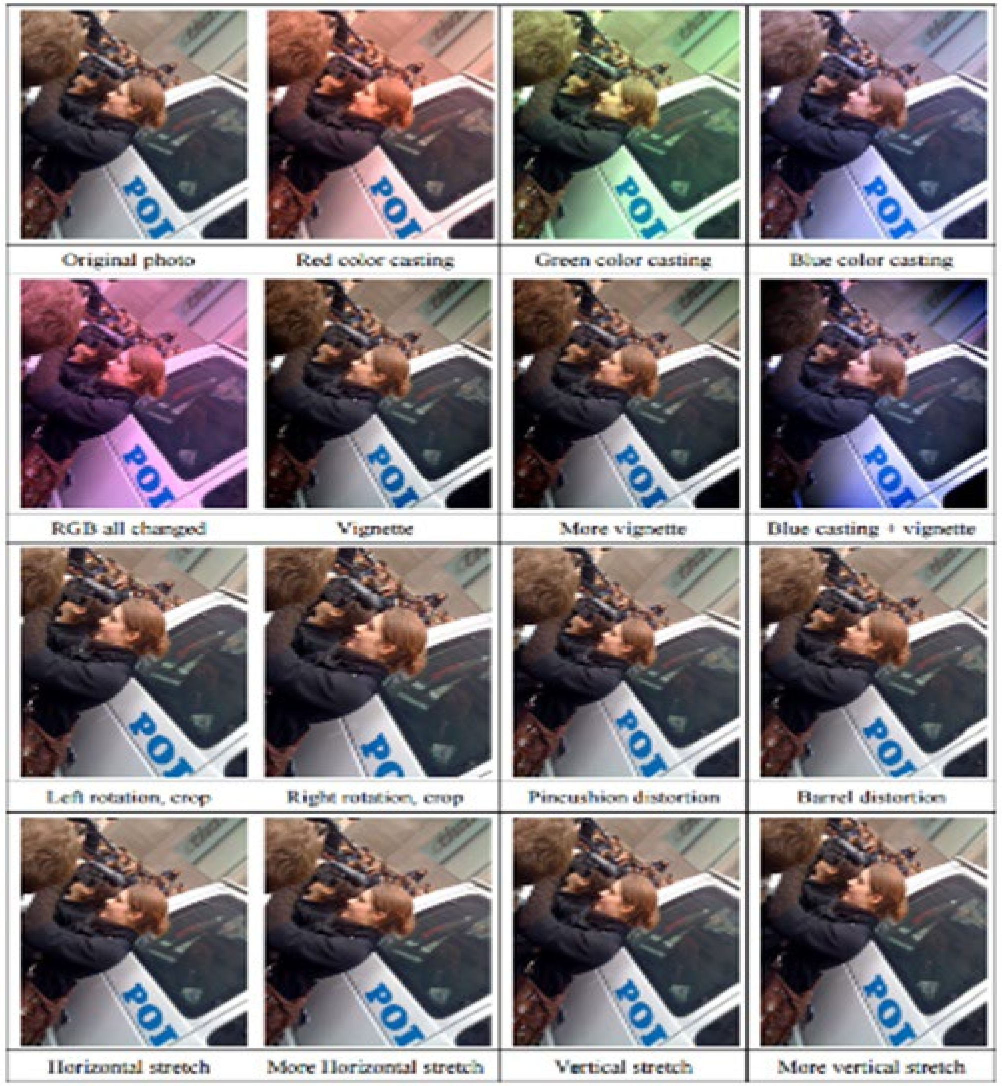
- Can improve performance, even with fairly large datasets

Slide: Jiabin Huang

Data Augmentation (Jittering)

- Create *virtual* training samples
  - Horizontal flip
  - Random crop
  - Color casting
  - Geometric distortion

- Idea goes back to Pomerleau 1995 at least (neural net for car driving)



Deep Image [Wu et al. 2015]

Applying Data Augmentation

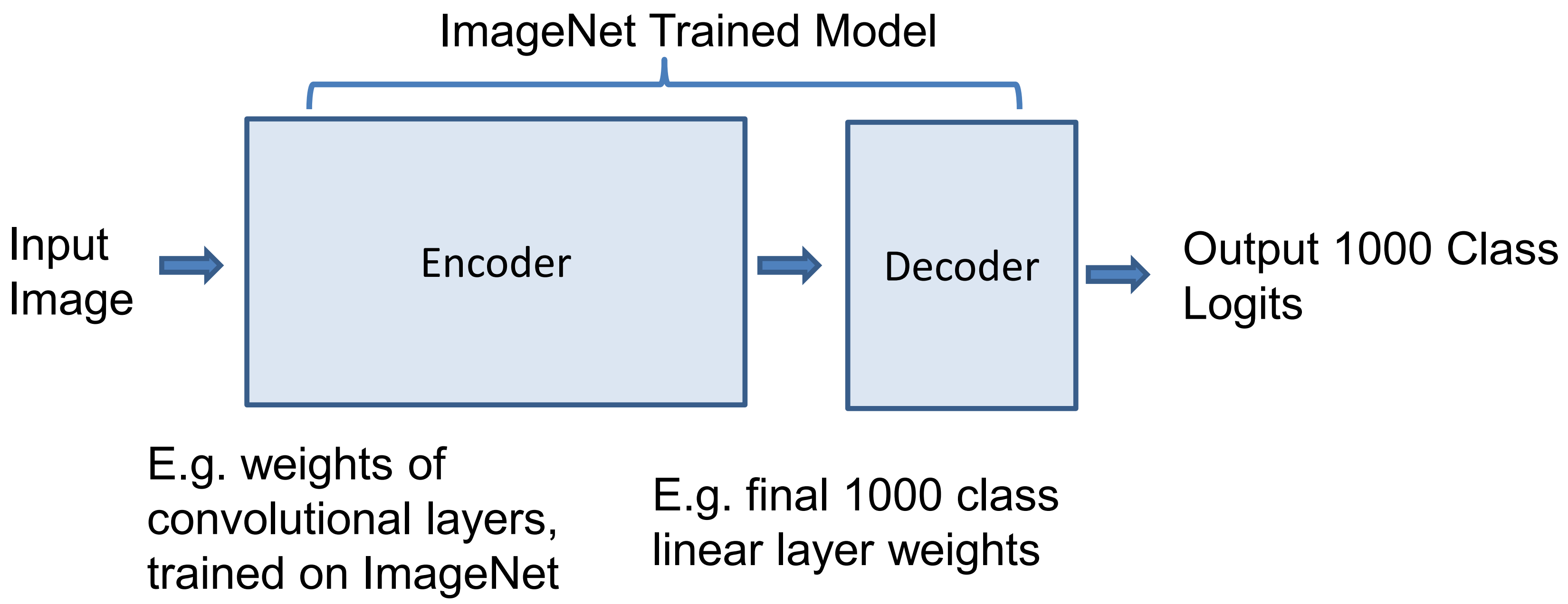
1. Define transformation sequence
2. Input transform specification to data loader

```
import torch
from torchvision import datasets, transforms

batch_size=200

train_loader = torch.utils.data.Dataloader(
    dataset.MNIST('../data', train=True, download=True,
                  transform=transforms.Compose([
                      transforms.RandomHorizontalFlip(),
                      transforms.RandomVerticalFlip(),
                      transforms.RandomRotation(15),
                      transforms.RandomRotation([90, 180, 270]),
                      transforms.Resize([32, 32]),
                      transforms.RandomCrop([28, 28]),
                      transforms.ToTensor()
                  ])),
    batch_size=batch_size, shuffle=True)
```

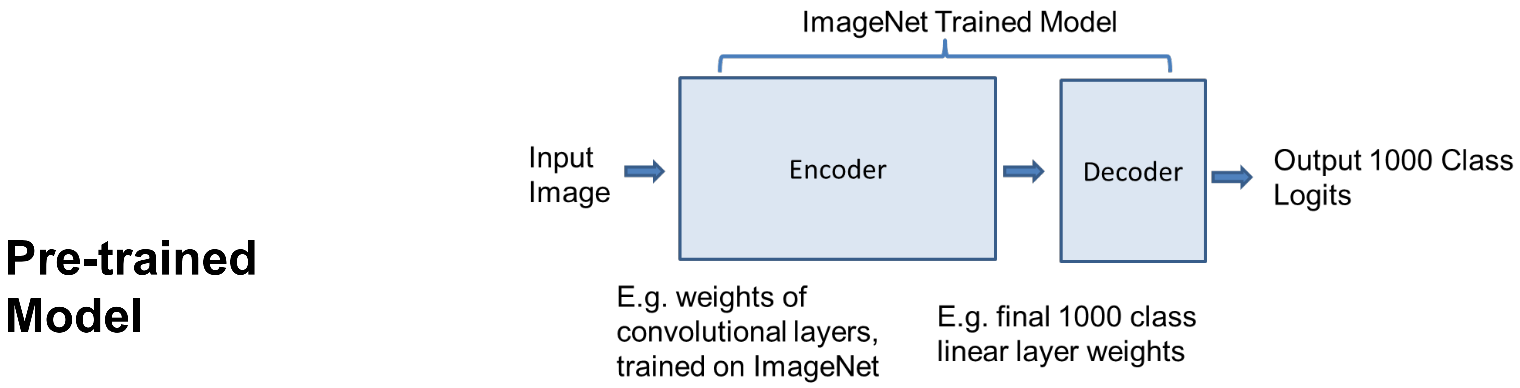
References:
https://medium.com/dejunhuang/learning-day-23-data-augmentation-in-pytorch-e375e19100c3
https://pytorch.org/vision/main/transforms.html

What if we want to do some new task?

- Suppose we've trained ImageNet model
- But we want to do something else, e.g. classify flowers or dog breeds
- We don't have a huge dataset for that task

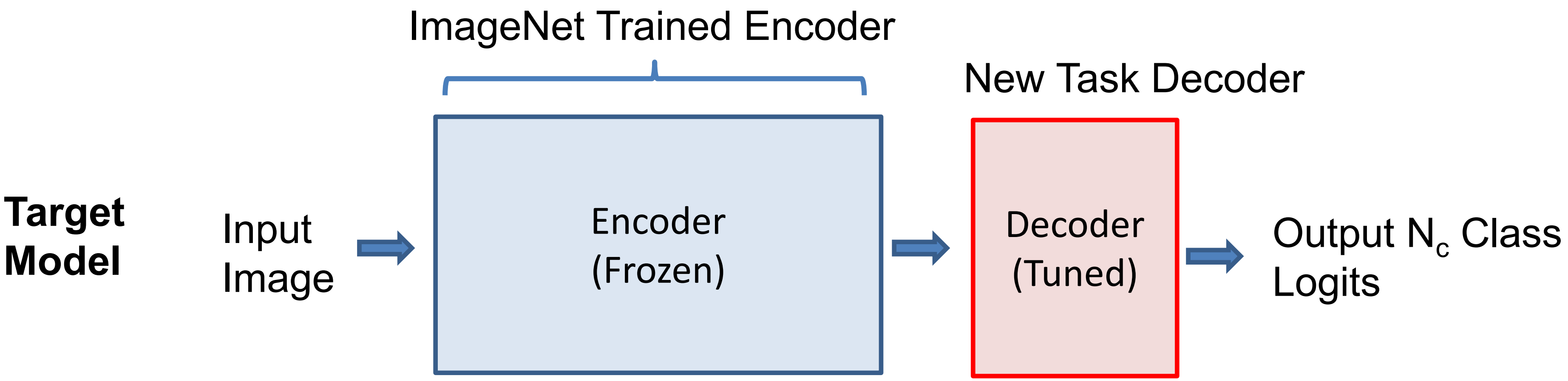


New Task Solution 1: “Linear probe” / “Feature extraction”



Keep original encoder weights. Replace decoder linear layer and train its weights on new task without changing encoder.

Equivalently, extract features from encoder and train linear model on those features



How to apply linear probe

Pre-compute features method

1. Load pretrained model (many available)
   https://pytorch.org/vision/stable/models.html
2. Remove prediction final layer
3. Apply model to each image to get features; save them
4. Train new linear model (e.g. logistic regression or SVM) on the features

```
import torch
import torch.nn as nn
from torchvision import models

model = models.alexnet(pretrained=True)

# remove last fully-connected layer
new_classifier = nn.Sequential(*list(model.classifier.children())[:-1])
model.classifier = new_classifier
```

Freeze encoder method

1. Load pretrained model (many available)
   https://pytorch.org/vision/stable/models.html
2. Set network to not update weights
3. Replace last layer
4. Retrain network with new dataset

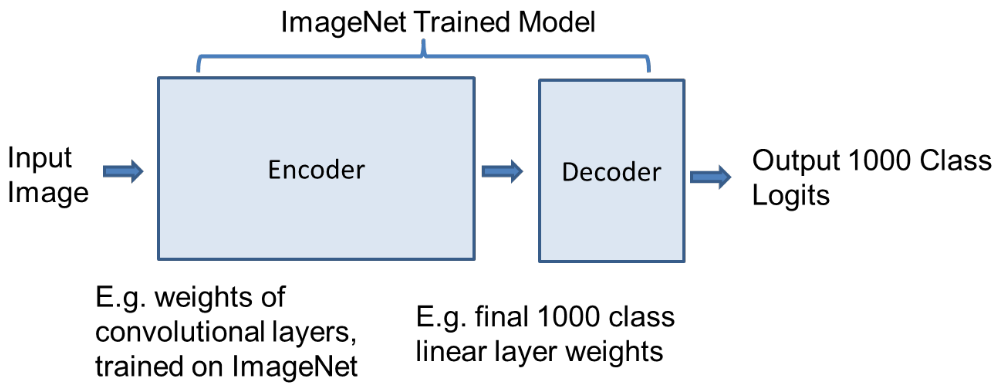
- Slower than method on left but does not require storing features, and can apply data augmentation

```
model = torchvision.models.vgg19(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
    # Replace the last fully-connected layer
    # Parameters of newly constructed modules have requires_grad=True by default
model.fc = nn.Linear(512, 8) # assuming that the fc7 layer has 512 neurons, other
model.cuda()
```

Source

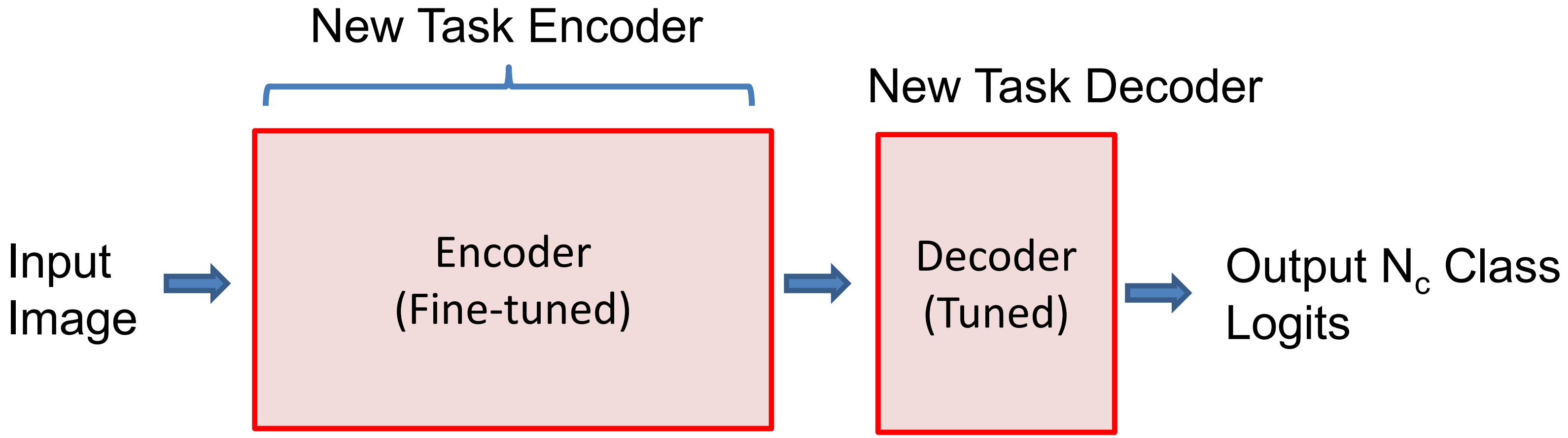
New Task Solution 2: “Fine-tuning”

Pre-trained Model



1. Initialize with original encoder weights.
2. Replace decoder linear layer.
3. Use 10x smaller learning rate than normal and train

Target Model



How to apply fine-tuning

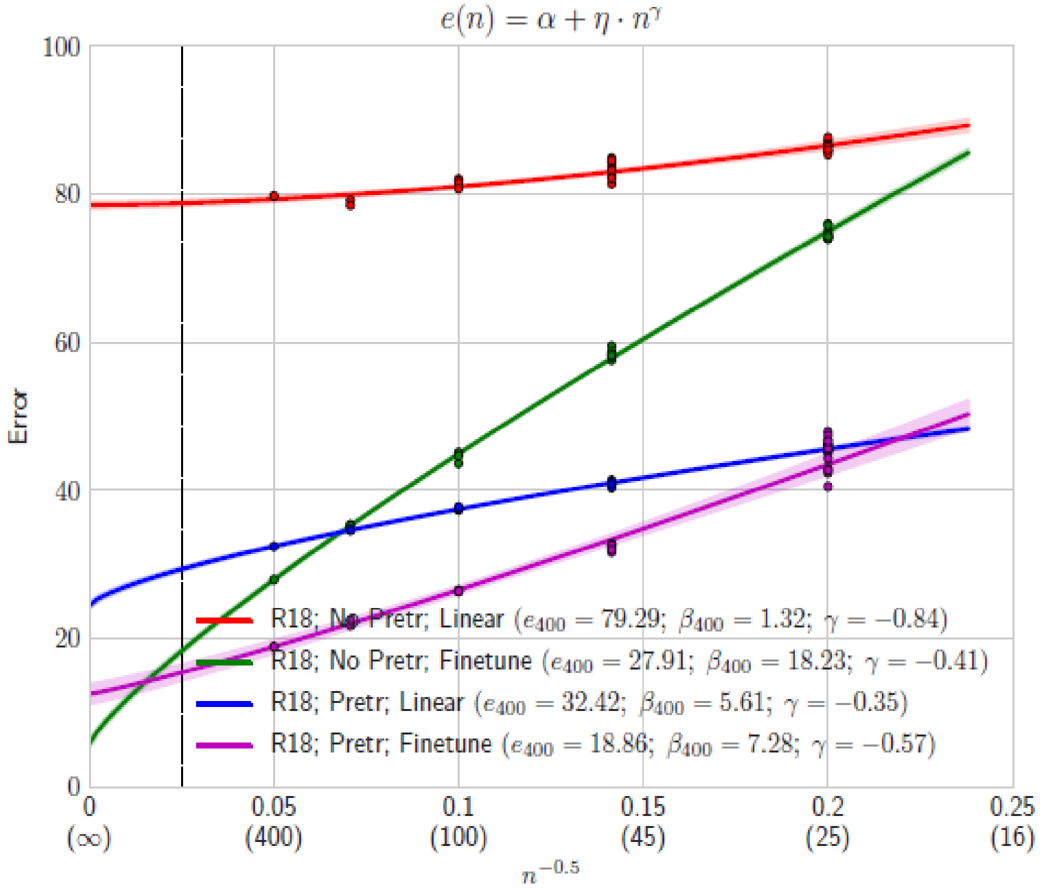
1. Load pre-trained model
2. Replace last layer
3. Set a low learning rate (e.g. lr=e-4)
   - Learning rate is often at least 10x lower than for “scratch” training
   - Can “warm start” by freezing earlier layers initially and then unfreezing after a few epochs when the linear layer is mostly trained (avoids messing up encoder while classifier is adjusting)
   - Can set lower learning rate for earlier layers

```
target_class = 37
model = torch.hub.load('pytorch/vision:v0.10.0', 'resnet34', pretrained=True)
model.fc = nn.Linear(512, target_class)
```

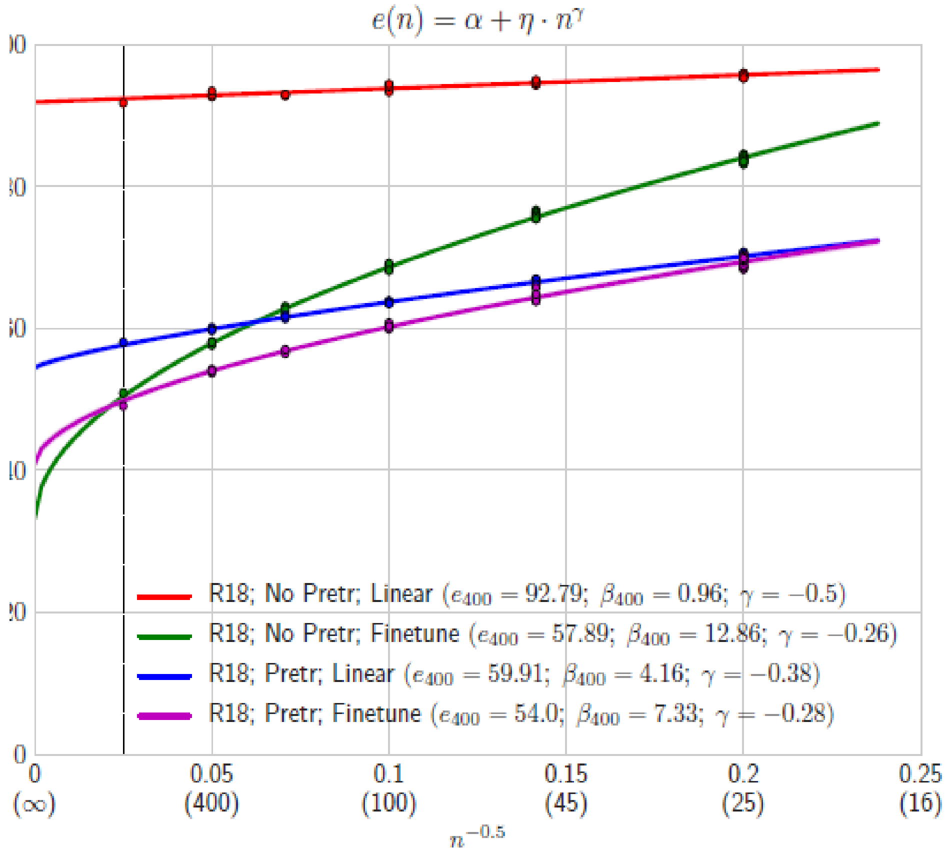
Assumes last layer has 512 features and is called “fc”

[Other examples of layer customization](#)(from Weijie)

Task transfer vs. # target task examples



(a) Transfer: ImageNet to Cifar100



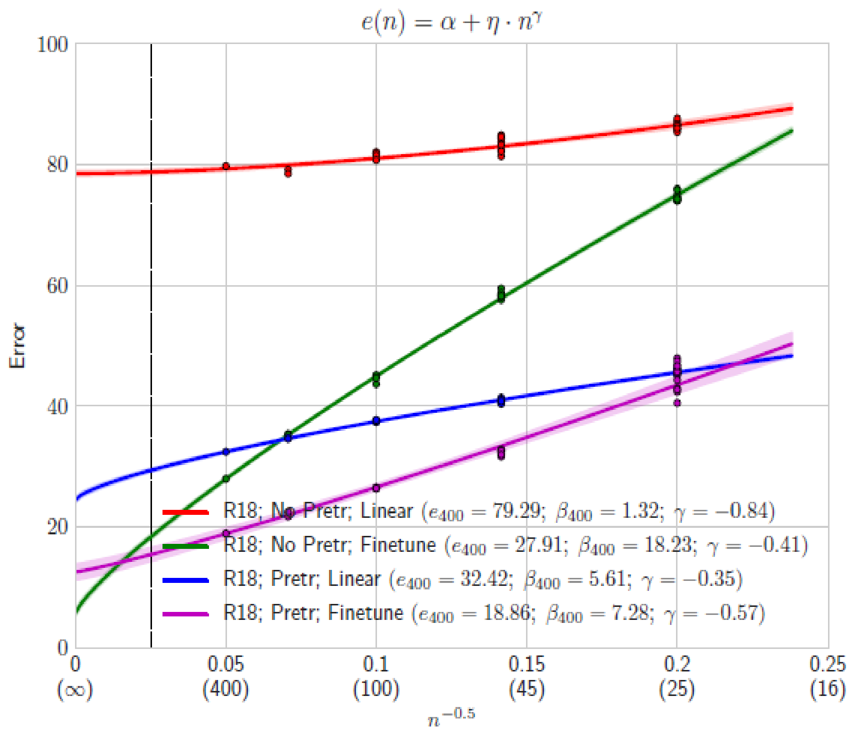
(b) Transfer: ImageNet to Places365

Green: Train from scratch
Blue: Linear Probe from ImageNet
Purple: Fine-tune from ImageNet
ResNet18, Err vs # examples / class (in paren)

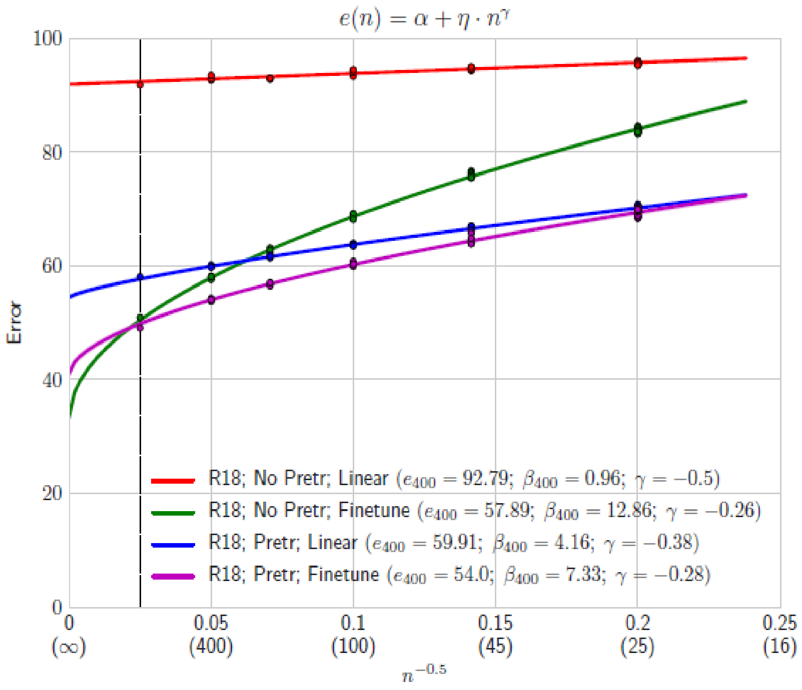
“Learning Curves” (2021) pdf

2 Minute Break

- Comparing linear probe, fine-tuning, and training from scratch, when does each have an advantage and why?



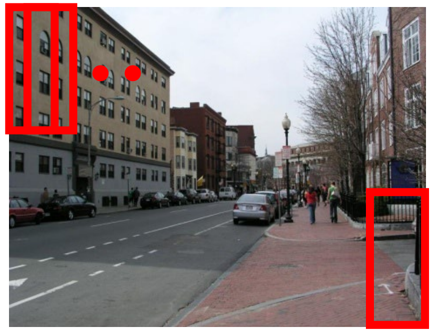
(a) Transfer: ImageNet to Cifar100



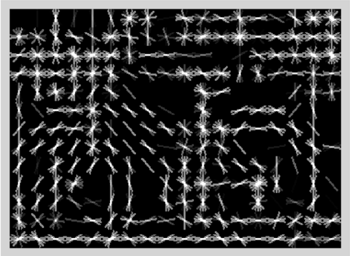
(b) Transfer: ImageNet to Places365

Statistical template approach to object detection

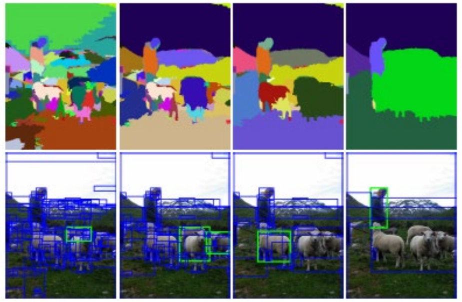
Propose Window → Extract Features → Classify → Post-process



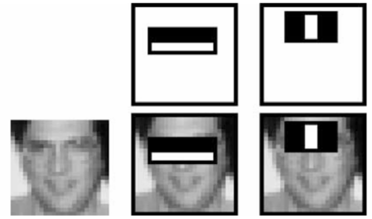
Sliding window: scan image pyramid



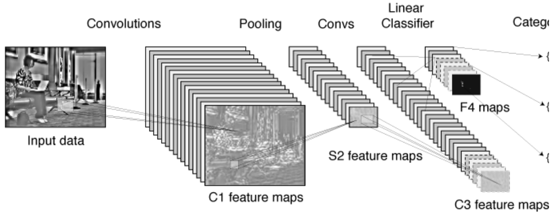
Region proposals: edge/region-based, resize to fixed window



HOG



Fast randomized features



CNN features

SVM

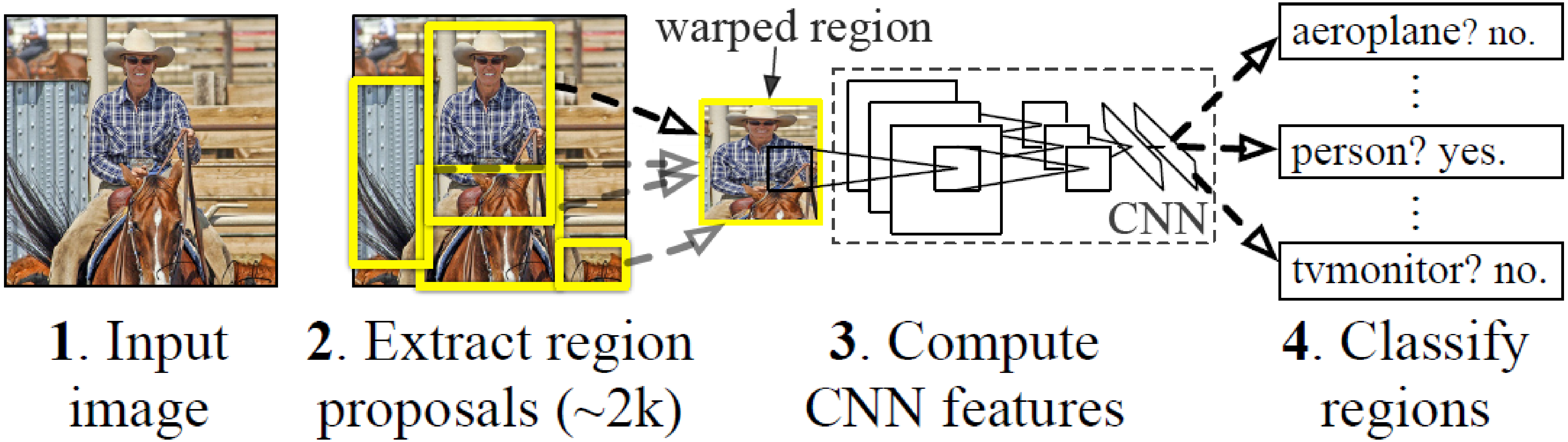
Boosted stubs

Neural network

Non-max suppression

Segment or refine localization

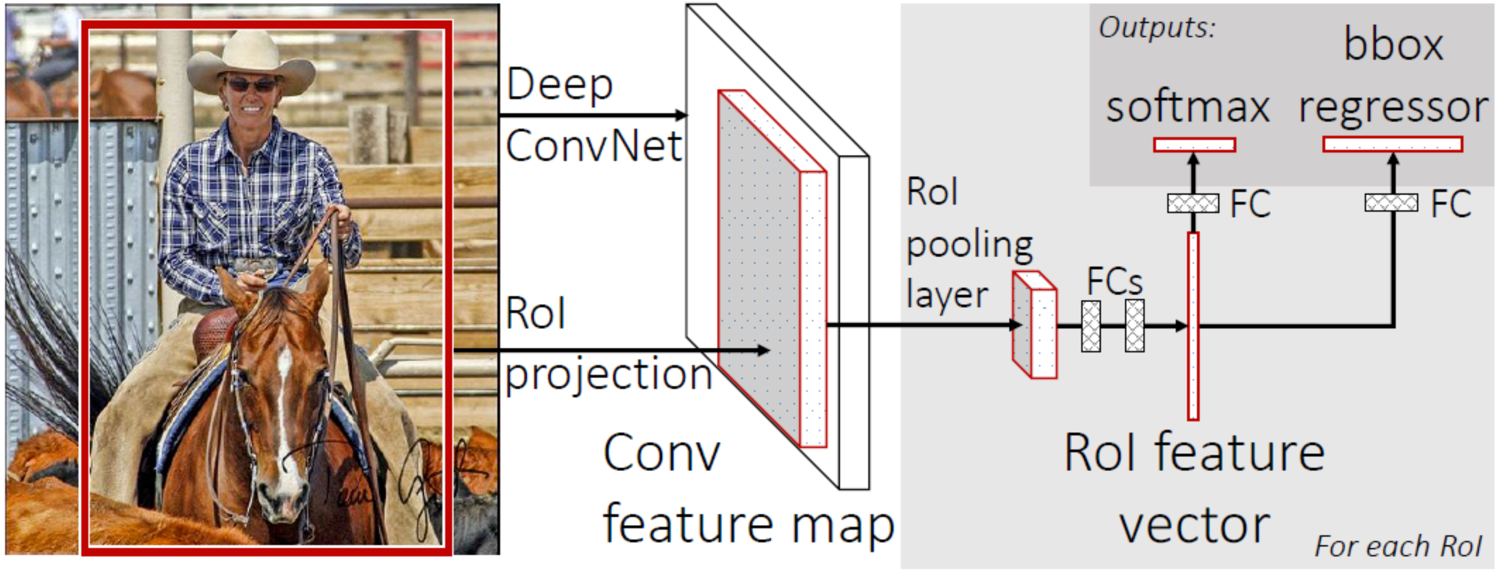
R-CNN (Girshick et al. CVPR 2014)



- Extract regions using Selective Search method (Uijilings et al. IJCV 2013)
- Extract rectangles around regions and resize to 227x227
- Extract features with fine-tuned CNN (that was initialized with network trained on ImageNet before training)
- Classify last layer of network features with SVM

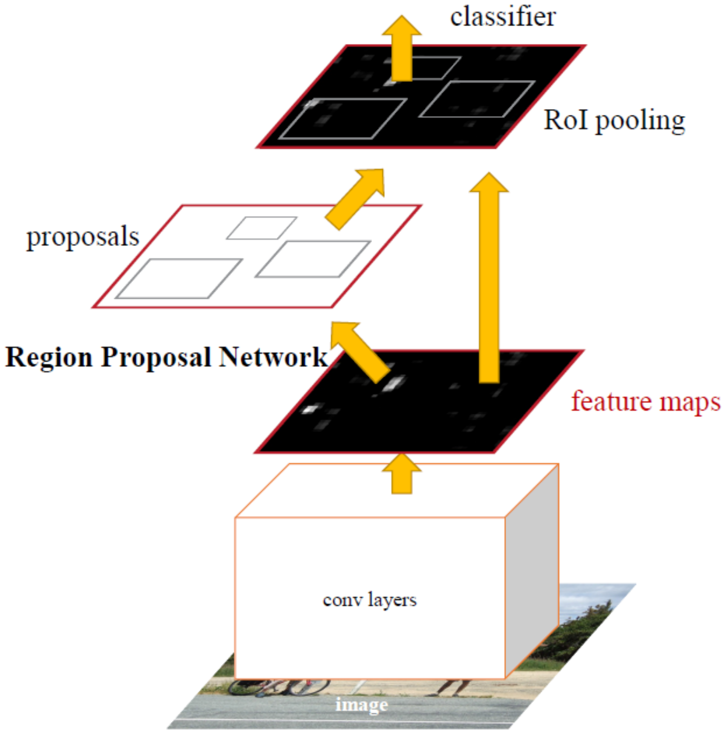
http://arxiv.org/pdf/1311.2524.pdf

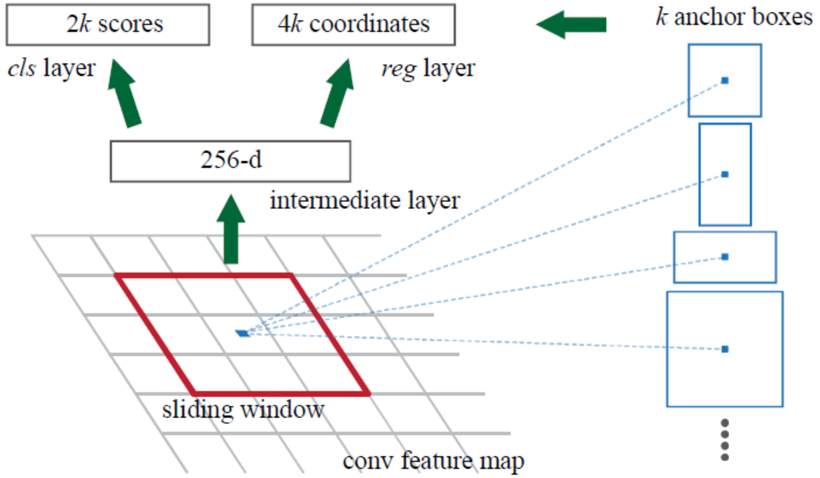
Fast R-CNN – Girshick 2015



- Compute CNN features for image once
- ROI Pooling: Pool into 7x7 spatial bins for each region proposal, output class scores and regressed bboxes
- Other refinements: compress classification layer, use network for final classification, end-to-end training
- 100x speed up of R-CNN (0.02 – 0.1 FPS → 0.5-20 FPS) with similar accuracy

Faster R-CNN – Ren et al. 2016

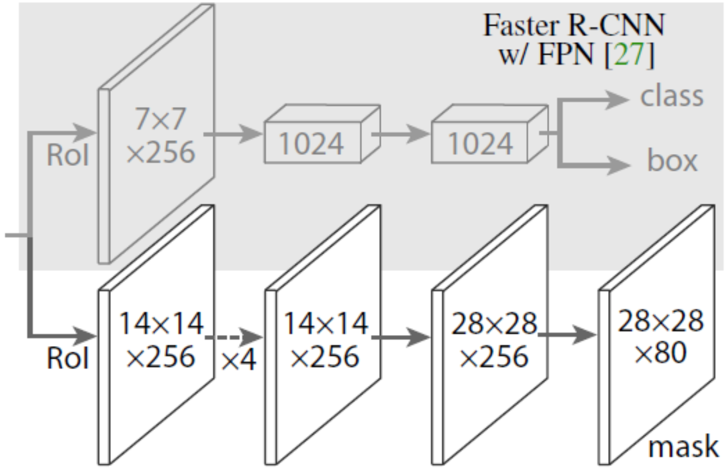




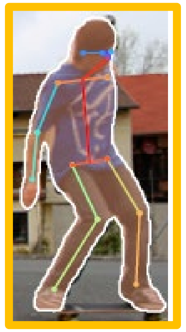
- Convolutional features used for generating proposals and scoring
  - Generate proposals with "objectness" scores and refined bboxes for each of k "anchors"
  - Score proposals in same way as Fast R-CNN
- Similar accuracy to Fast R-CNN with 10x speedup

Mask R-CNN – He Gxioxari Dollar Girshick (2017)

- Same network as Faster R-CNN, except
  - Bilinearly interpolate when extracting 7x7 cells of ROI features for better alignment of features to image
  - Instance segmentation: produce a 28x28 mask for each object category
  - Keypoint prediction: produce a 56x56 mask for each keypoint (aim is to label single pixel as correct keypoint)



Example ROI and predicted mask



Example ROI and predicted mask and keypoints

Top performing object detector, keypoint segmenter, instance segmenter (at time of release and for a bit after)

	backbone	AP^{bb}	AP^{bb}_{50}	AP^{bb}_{75}	AP^{bb}_{S}	AP^{bb}_{M}	AP^{bb}_{L}
Faster R-CNN+++ [19]	ResNet-101-C4	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN [27]	ResNet-101-FPN	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI [21]	Inception-ResNet-v2 [37]	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM [36]	Inception-ResNet-v2-TDM	36.8	57.7	39.2	16.2	39.8	**52.1**
Faster R-CNN, RoIAlign	ResNet-101-FPN	37.3	59.6	40.3	19.8	40.2	48.8
Mask R-CNN	ResNet-101-FPN	38.2	60.3	41.7	20.1	41.1	50.2
Mask R-CNN	ResNeXt-101-FPN	**39.8**	**62.3**	**43.4**	**22.1**	**43.2**	51.2

Table 3. **Object detection** *single-model* results (bounding box AP), *vs.* state-of-the-art on test-dev. Mask R-CNN usir

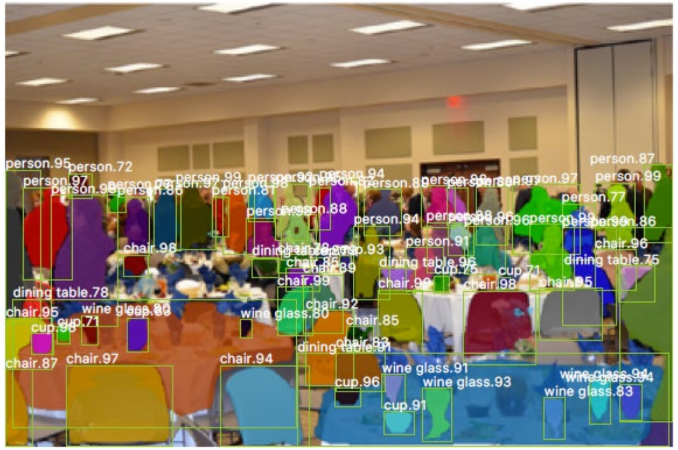
	backbone	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
MNC [10]	ResNet-101-C4	24.6	44.3	24.8	4.7	25.9	43.6
FCIS [26] +OHEM	ResNet-101-C5-dilated	29.2	49.5	-	7.1	31.3	50.0
FCIS+++ [26] +OHEM	ResNet-101-C5-dilated	33.6	54.5	-	-	-	-
Mask R-CNN	ResNet-101-C4	33.1	54.9	34.8	12.1	35.6	51.1
Mask R-CNN	ResNet-101-FPN	35.7	58.0	37.8	15.5	38.1	52.4
Mask R-CNN	ResNeXt-101-FPN	**37.1**	**60.0**	**39.4**	**16.9**	**39.9**	**53.5**

Table 1. **Instance segmentation** *mask* AP on COCO test-dev. MNC [10] and FCIS [26] are the winners of the COCO 2015 and 2016

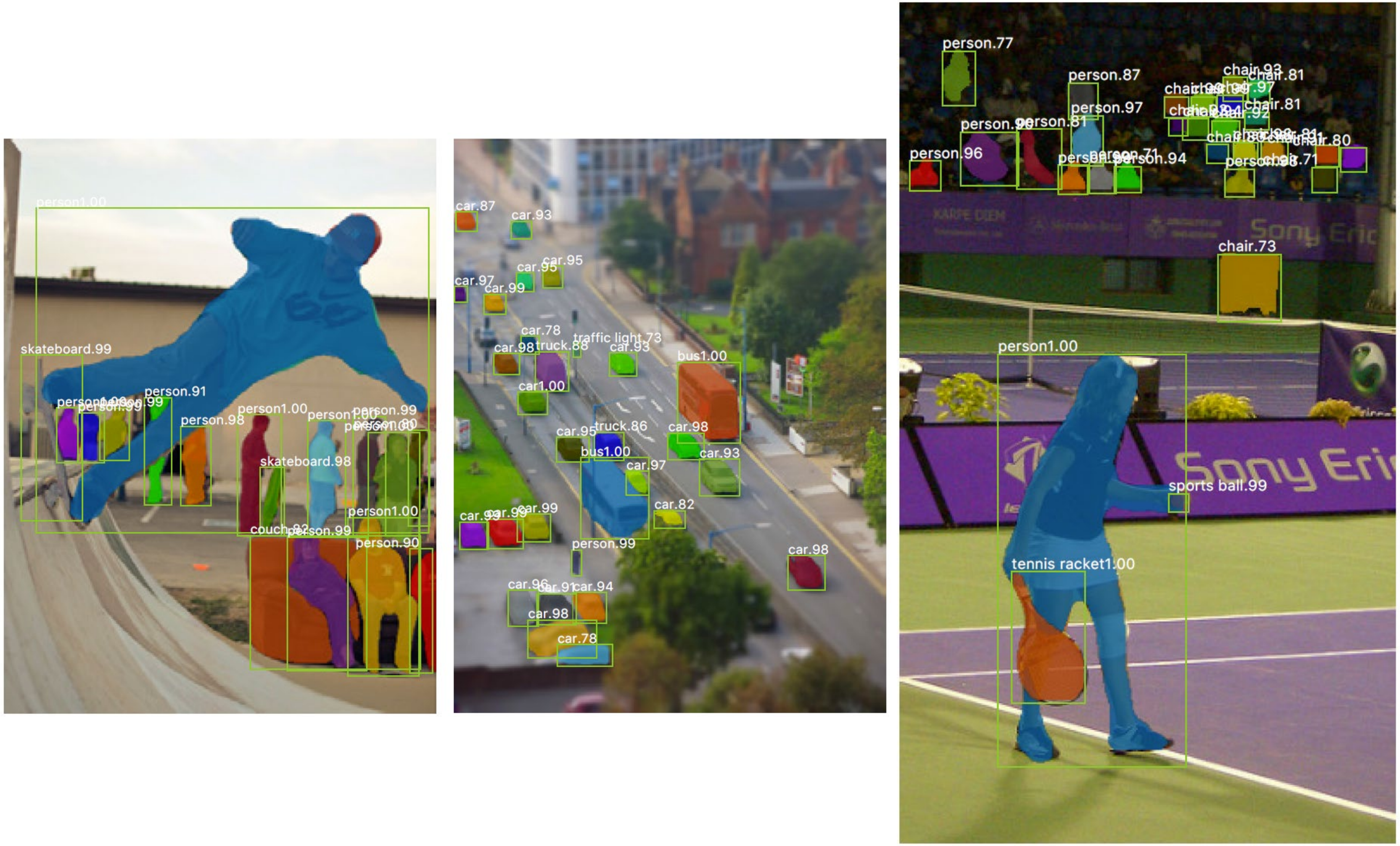
	AP^{kp}	AP^{kp}_{50}	AP^{kp}_{75}	AP^{kp}_{M}	AP^{kp}_{L}
CMU-Pose+++ [6]	61.8	84.9	67.5	57.1	68.2
G-RMI [31]†	62.4	84.0	68.5	**59.1**	68.1
Mask R-CNN, keypoint-only	62.7	87.0	68.4	57.4	71.1
Mask R-CNN, keypoint & mask	**63.1**	**87.3**	**68.7**	57.8	**71.4**

Table 4. **Keypoint detection** AP on COCO test-dev. Ours

Example detections and instance segmentations



Example detections and instance segmentations

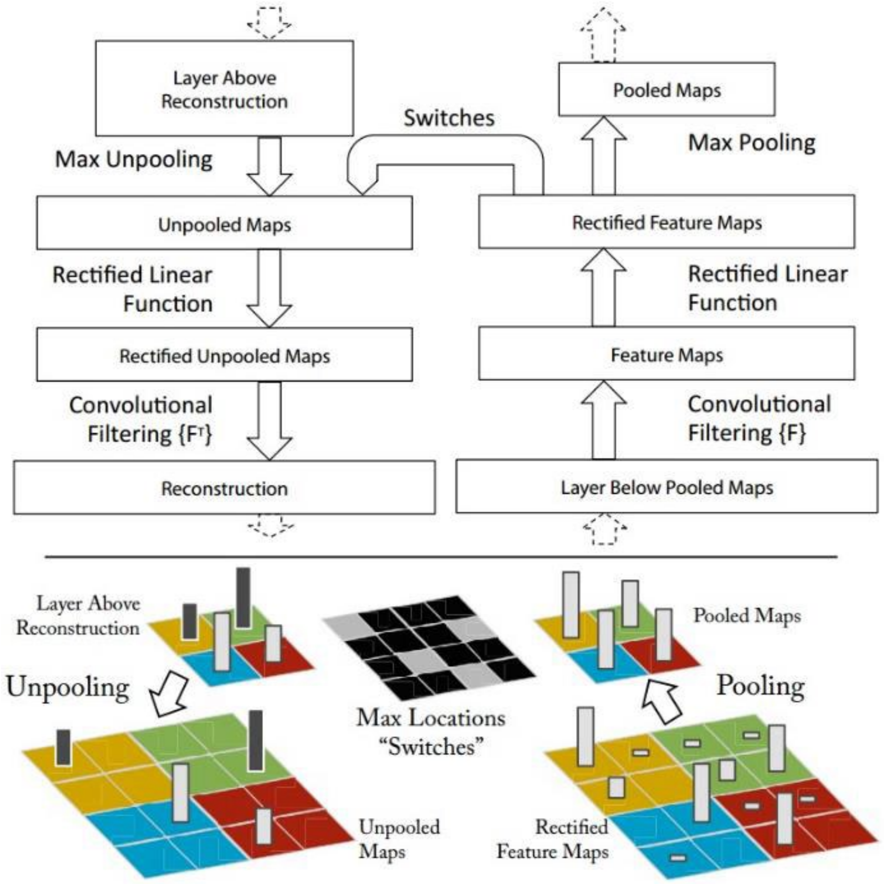


Example keypoint detections

What does the CNN learn?

Map activation back to the input pixel space

- What input pattern originally caused a given activation in the feature maps?



Visualizing and Understanding Convolutional Networks [Zeiler and Fergus, ECCV 2014]

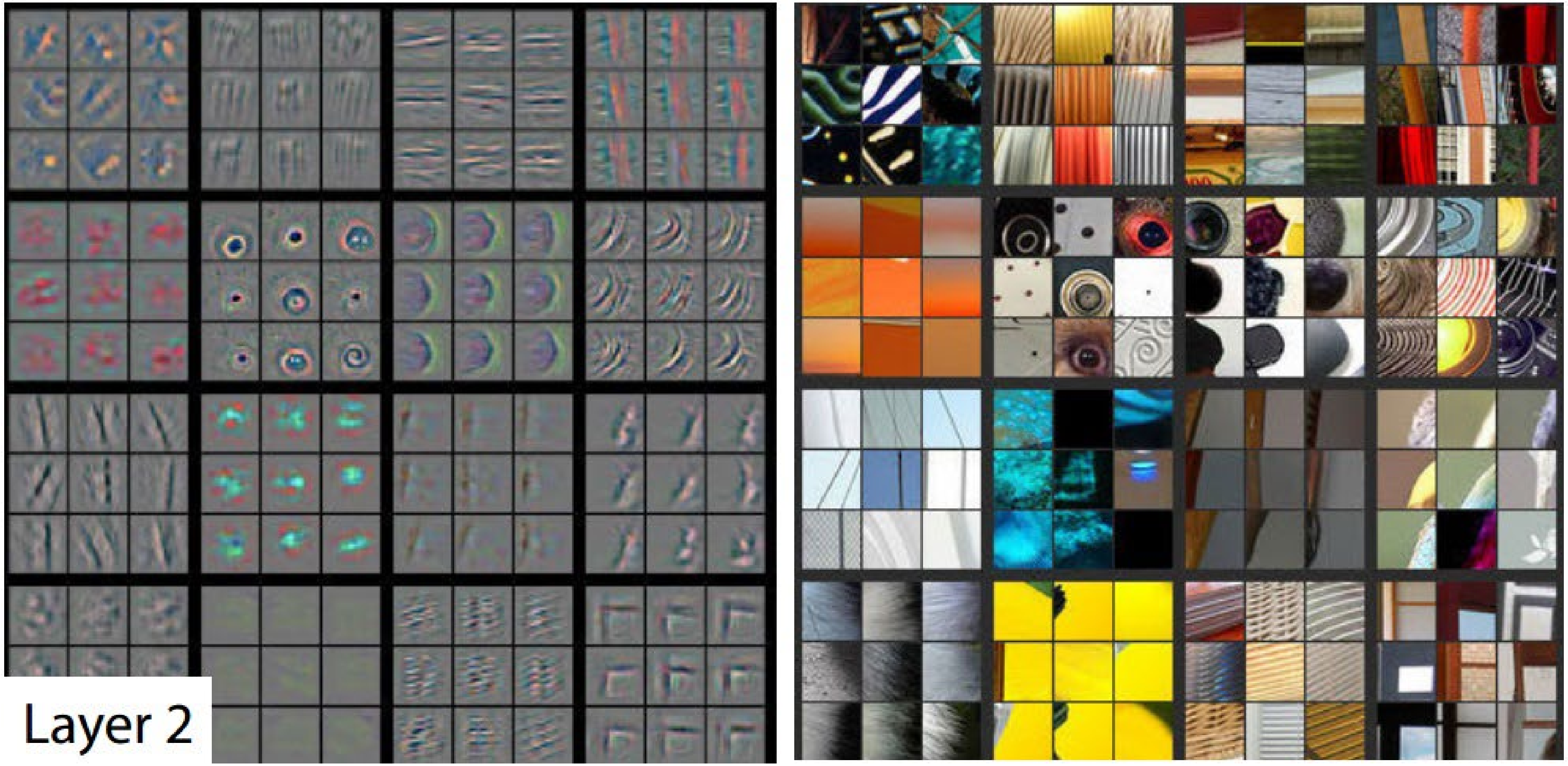
Layer 1 (visualization of randomly sampled features)

Activations (which pixels caused the feature to have a high magnitude)

Image patches that had high activations

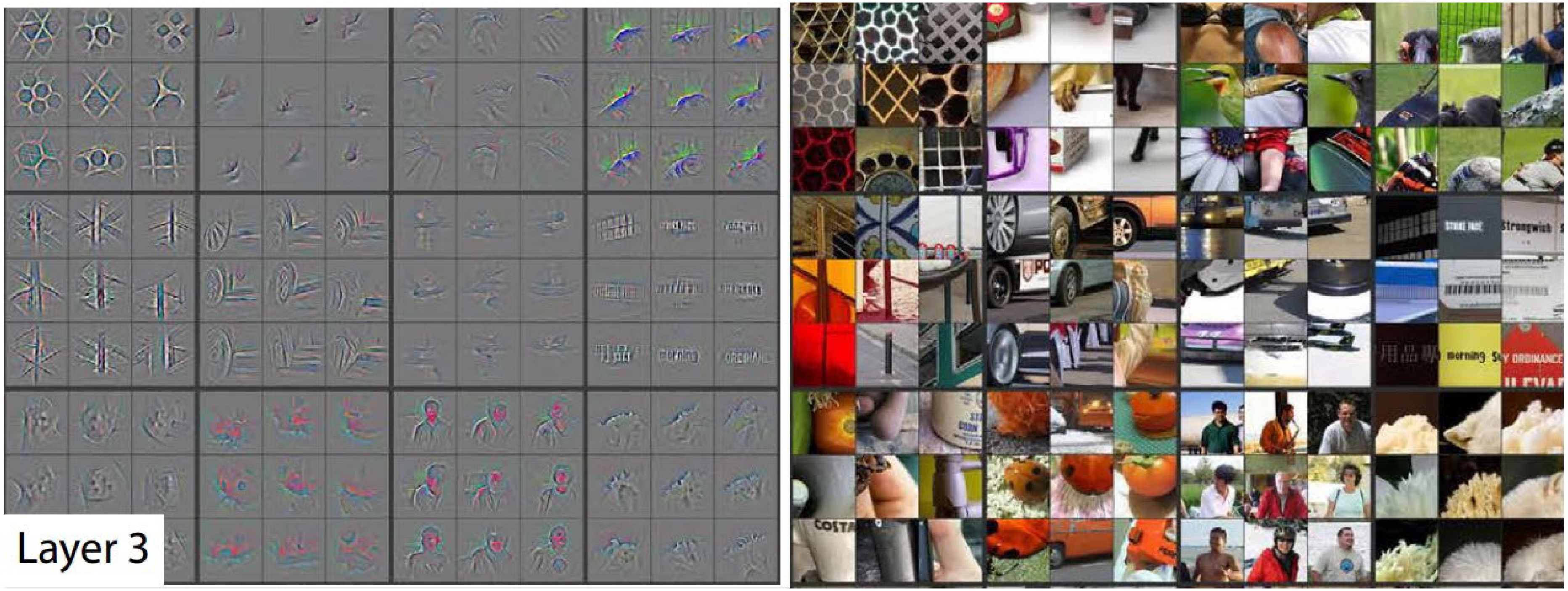
Visualizing and Understanding Convolutional Networks [Zeiler and Fergus, ECCV 2014]

Layer 2



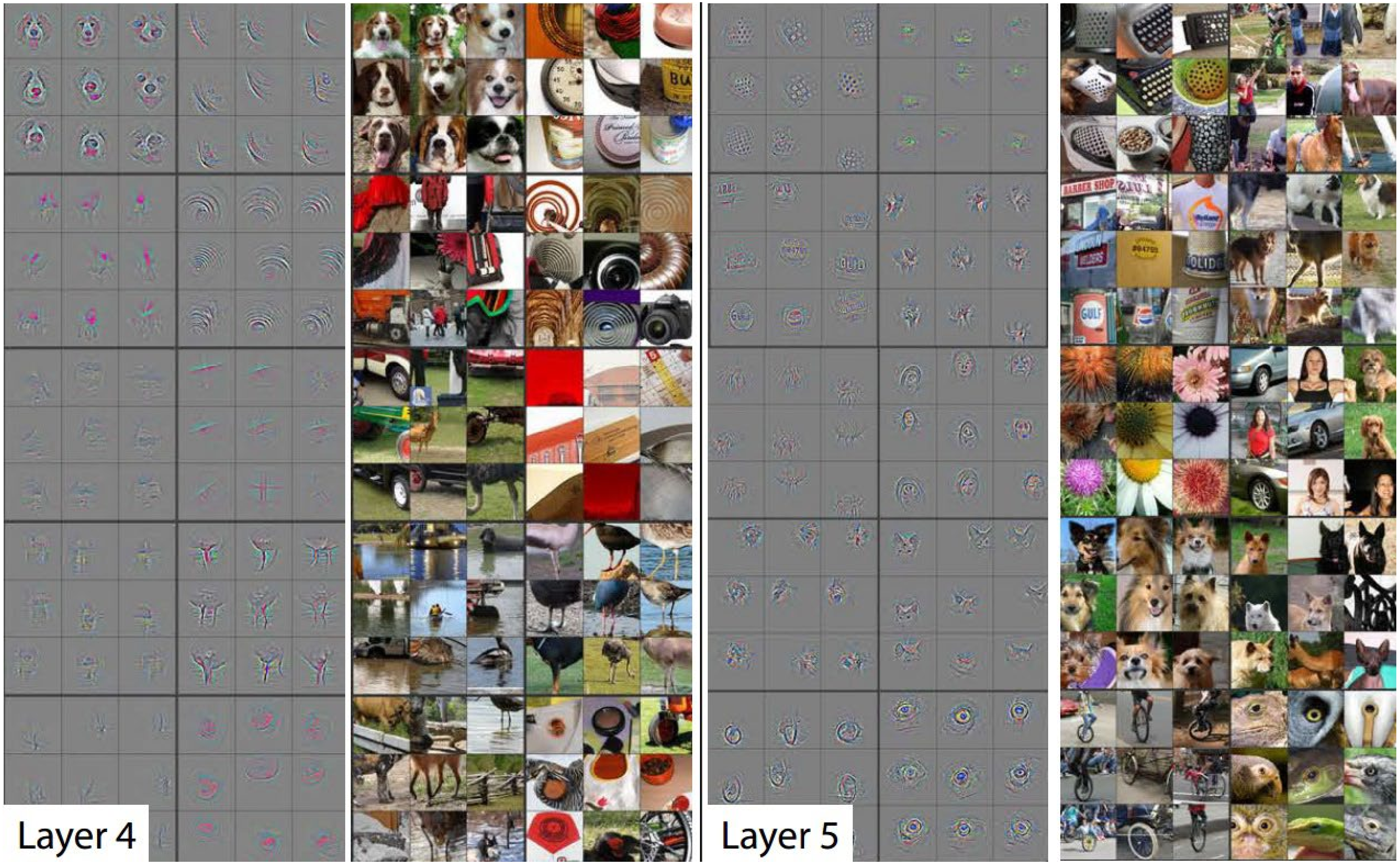
Visualizing and Understanding Convolutional Networks [Zeiler and Fergus, ECCV 2014]

Layer 3



Visualizing and Understanding Convolutional Networks [Zeiler and Fergus, ECCV 2014]

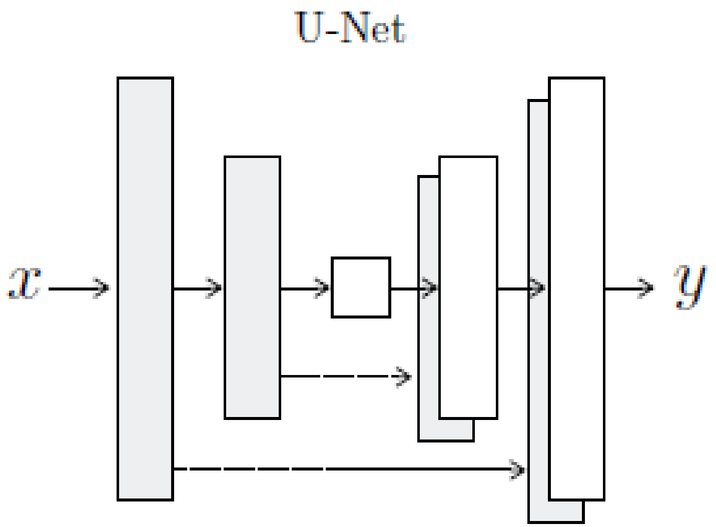
Layer 4 and 5

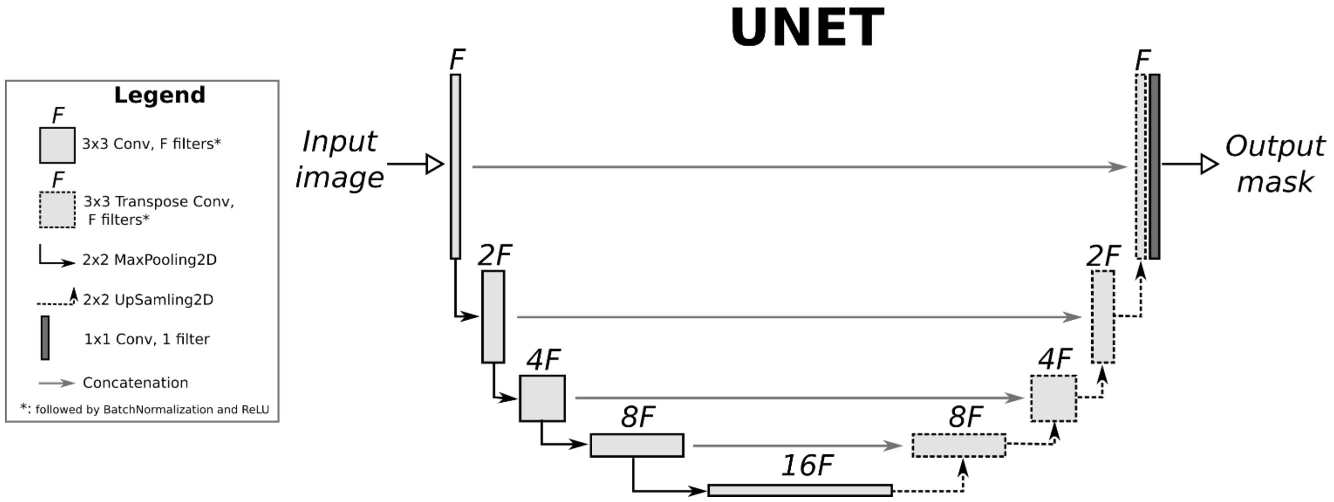


Visualizing and Understanding Convolutional Networks [Zeiler and Fergus, ECCV 2014]

U-Net Architecture

O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In MICCAI, 2015.





The "U-Net" is an encoder-decoder with skip connections between mirrored layers in the encoder and decoder stacks.

U-Net style architectures are used to generate pixel maps (e.g., RGB images or per-pixel labels)

Fig from Isola et al. 2017

Fig src

Things to remember

- Massive ImageNet dataset was an ingredient to deep learning breakthrough
- Skip connections, data augmentation, and batch normalization are commonly used techniques
- Models trained on ImageNet are used as pretrained “backbones” for other vision tasks
- Mask-RCNN samples patches in feature maps and predicts boxes, object region, and keypoints
- Many image generation and segmentation methods are based on U-Net downsamples while deepening features, then upsamples with skip connections

IMAGENET
22K categories and 15M images