Dall-E

Deep Learning Optimization and Computer Vision

Applied Machine Learning
Derek Hoiem

Today's Lecture

- Other architecture and training tricks
  - Batch normalization
  - Data augmentation
- Defining and training a deep network w/ PyTorch
- Adopting the network to new tasks
  - Fine-tuning
  - Linear probe
- Mask RCNN recognition system

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

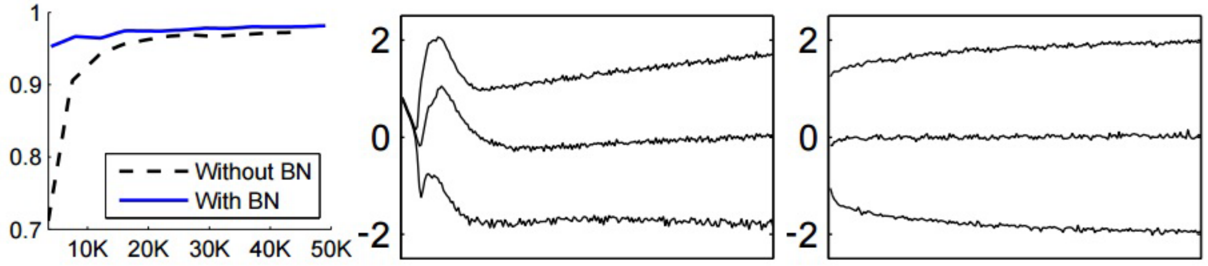
Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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



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

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

Example code: 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
```

Forward applies prediction, going through each layer

Backward applies backpropagation to compute the loss gradient with respect to parameters in each layer

[Pretrained Torch models]

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!

Training a deep network

1. Define the network model

   (see ResNet example in previous slides)

Convolutional network for Digits Classification

```
class Network(nn.Module):
    def __init__(self, num_classes=10, dropout = 0.5):
        super(Network, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 256, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )

        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(p=dropout),
            nn.Linear(256 * 6 * 6, 512),
            nn.ReLU(inplace=True),
            nn.Dropout(p=dropout),
            nn.Linear(512, 512),
            nn.ReLU(inplace=True),
            nn.Linear(512, num_classes),
        )


    def forward(self, x):
        N, c, H, W = x.shape
        features = self.features(x)
        pooled_features = self.avgpool(features)
        output = self.classifier(torch.flatten(pooled_features, 1))
        return output
```

Training a deep network

1. Define the network model
2. Set the key training parameters: # epochs, initial learning rate and schedule, optimizer, loss function, data loaders

```
# Set up the training
num_epochs = 20
test_interval = 1

# set initial learning rate and optimizer
learn_rate = 3E-4
optimizer = torch.optim.AdamW(model.parameters(), lr=learn_rate)

# define your learning rate scheduler, e.g. StepLR
lr_scheduler =  torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.5)

# set the loss
criterion = torch.nn.CrossEntropyLoss()
```

```
train_loader = DataLoader(dataset=train_set,
                          batch_size=64,
                          shuffle=True,
                          num_workers=2)

test_loader = DataLoader(dataset=test_set,
                         batch_size=64,
                         shuffle=False,
                         num_workers=2)
```

Training a deep network

1. Define the network model
2. Set the key training parameters
3. Train and track performance

Top-level of training

```
# Iterate over the DataLoader for training data
for epoch in tqdm(range(num_epochs), total=num_epochs, desc="Training ...", position=1):
    train_loss = train(train_loader, model, criterion, optimizer) # Train the Network for one epoch

    # TO DO: uncomment the line below. It should be called each epoch to apply the lr_scheduler
    lr_scheduler.step()

    train_losses.append(train_loss)
    print(f'Loss for Training on epoch {str(epoch)} is {str(train_loss)} \n')

    # Also compute validation loss/error every few epochs
    # Tools like TensorFlow and Weights&Biases make it easier to track and visualize experiments
```

Training a deep network

1. Define the network model
2. Set the key training parameters
3. Train and track performance

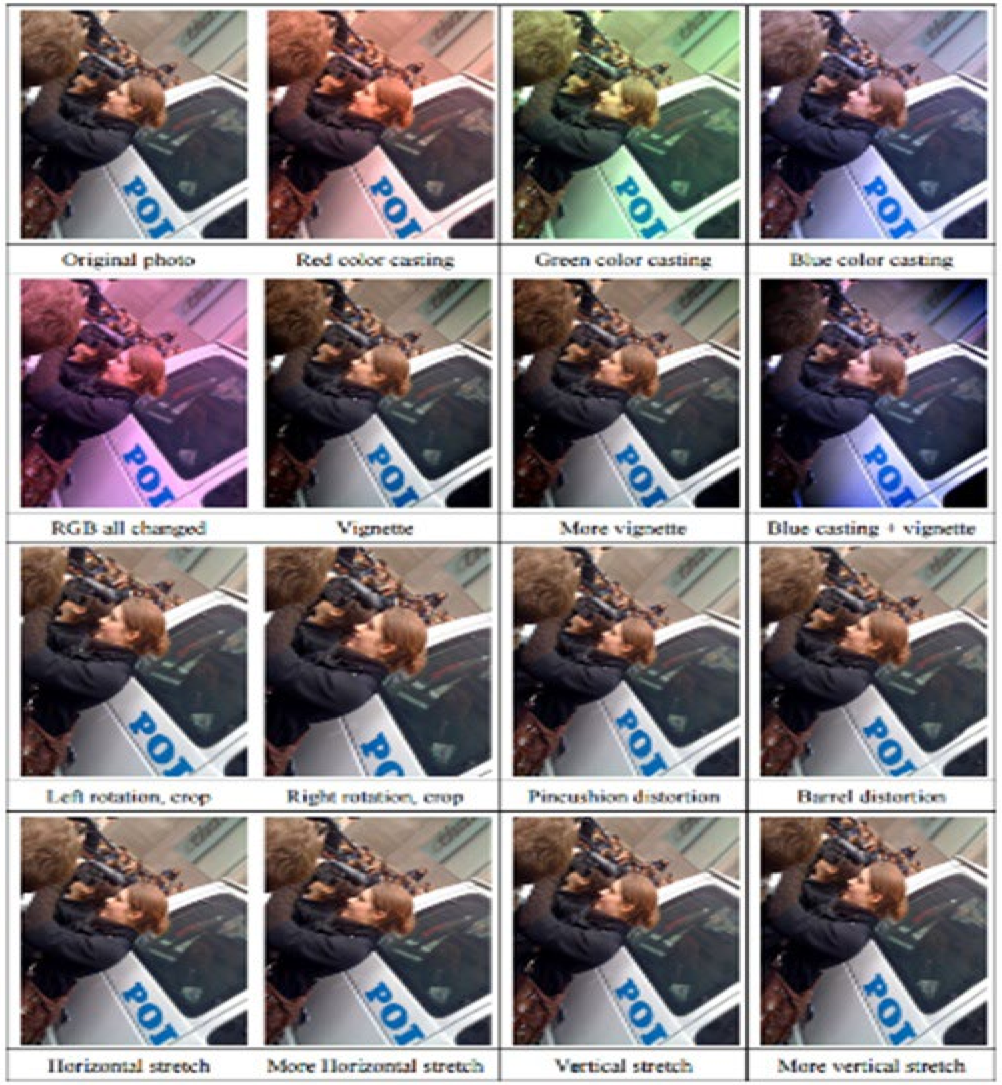
```
def train(train_loader, model, criterion, optimizer):
    """
    Train network
    :param train_loader: training dataloader
    :param model: model to be trained
    :param criterion: criterion used to calculate loss (should be CrossEntropyLoss
    :param optimizer: optimizer for model's params (Adams or SGD)
    :return: mean training loss
    """
    model.train()
    loss_ = 0.0
    losses = []

    # train for one epoch
    it_train = tqdm(enumerate(train_loader), total=len(train_loader), desc="Trainin
    for i, (images, labels) in it_train:

        # get images, labels for this batch
        images, labels = images.to(device), labels.to(device)

        # clear the gradients
        optimizer.zero_grad()

        # generate output for each image in the batch
        prediction = model(images)

        # compute the loss for each example
        loss = criterion(prediction, labels)

        it_train.set_description(f'loss: {loss:.3f}') # update displayed statement

        # compute the gradients
        loss.backward()

        # update the weights
        optimizer.step()

        # keep track of the loss to monitor the process
        losses.append(loss)

    return torch.stack(losses).mean().item()
```

Training Trick: Data Augmentation

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

- Simulates a larger training set, often improves improve performance

- 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

Training deep networks is a craft

- https://karpathy.github.io/2019/04/25/recipe/
- https://myrtle.ai/learn/how-to-train-your-resnet/

Questions to check knowledge

https://tinyurl.com/441deep24

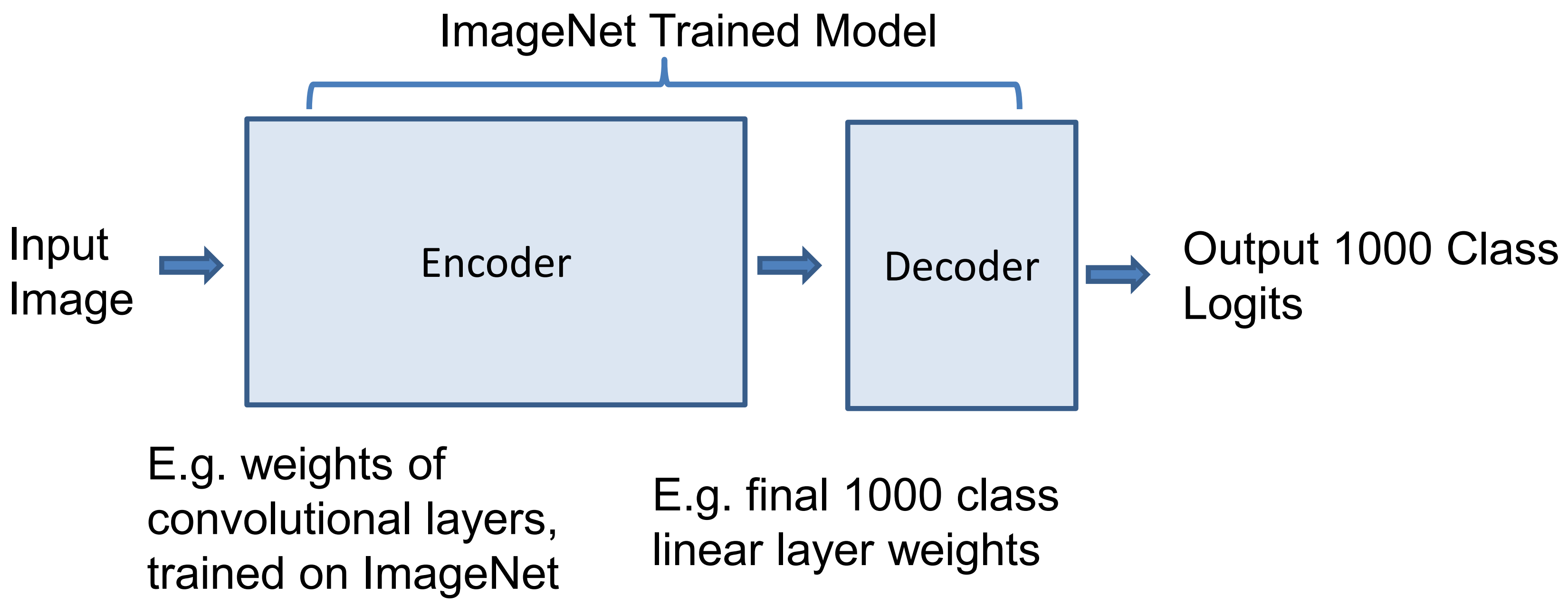
Adapting Networks to New Tasks

- Training a deep network from scratch requires a lot of data and a lot of compute

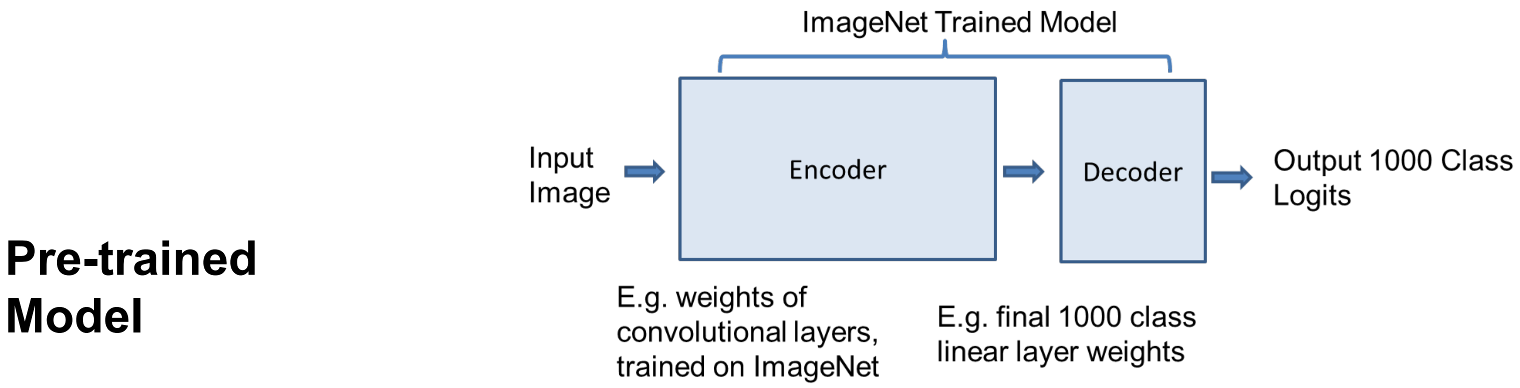
- **Critical concept**: We can start with a "pre-trained" network and adapt it to a new task
  - Linear probe
  - Fine-tuning

Adapting Networks to New Tasks

- 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

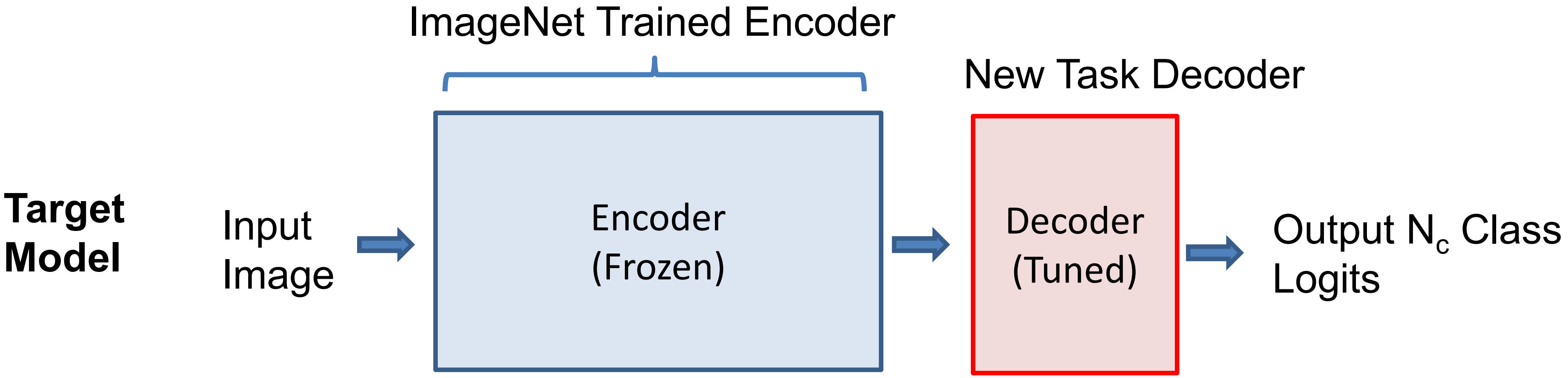


Linear probe, a.k.a. Feature extraction



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 with labels
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

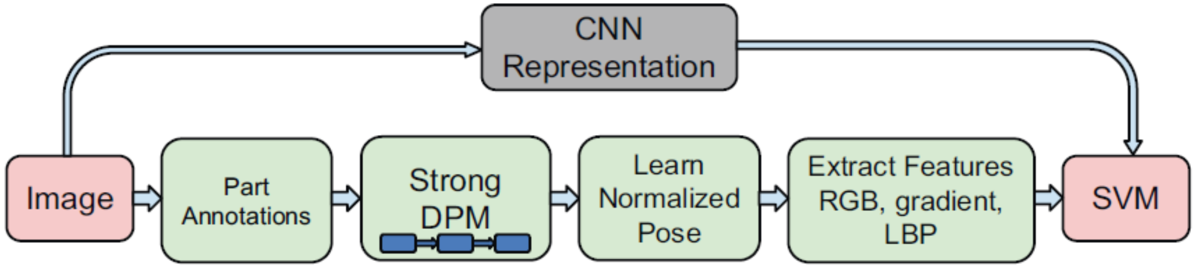
1. Load pretrained model (many available)
   https://pytorch.org/vision/stable/models.html
2. Set network to not update weights
3. Replace last layer
4. Retrain network with new dataset

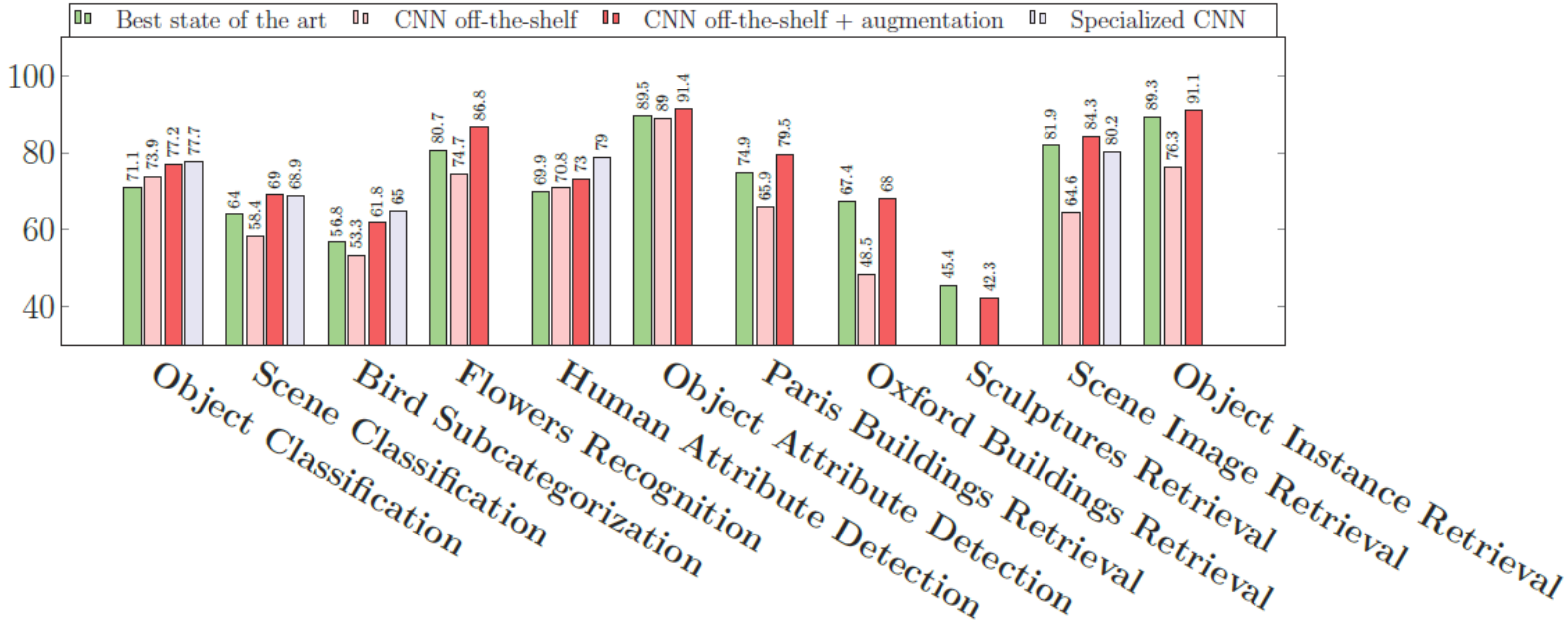
- Slower than method on left but does not require storing features, and can apply data augmentation

```
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

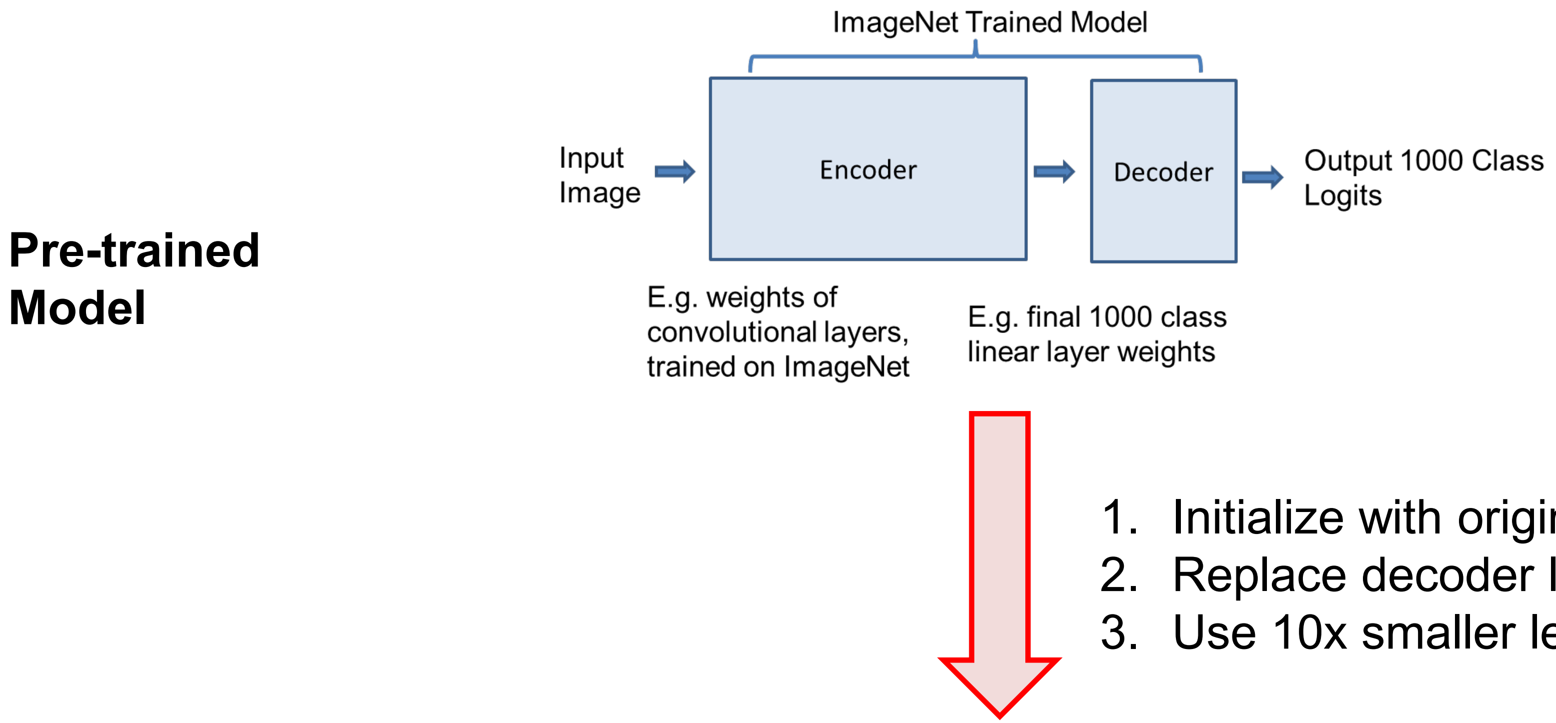
Pre-trained networks can provide very good features, as shown in "CNN Features off-the-shelf: an Astounding Baseline for Recognition"



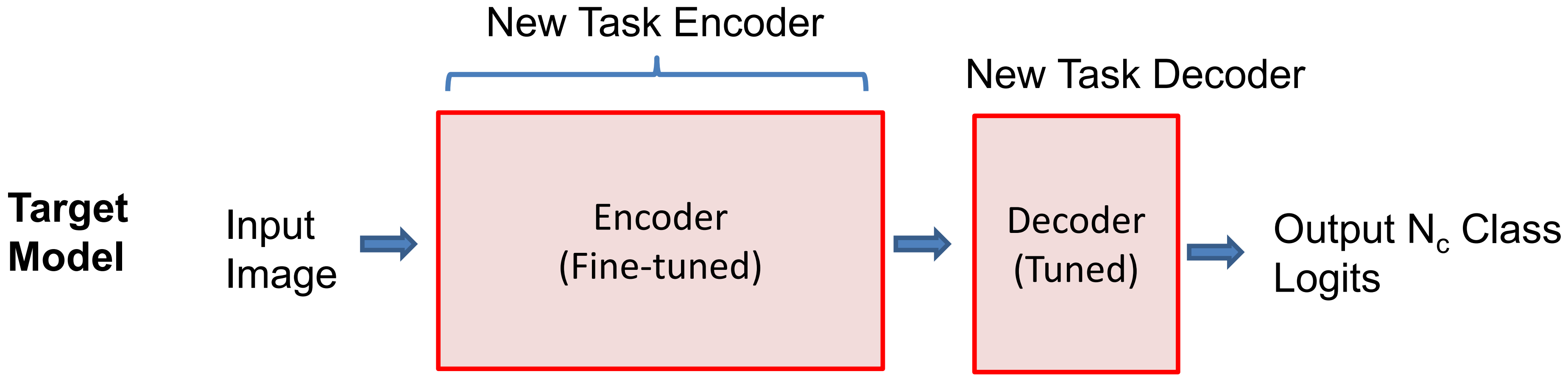


Razavian et al. CVPR 2014

Fine-tuning



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



How to apply fine-tuning

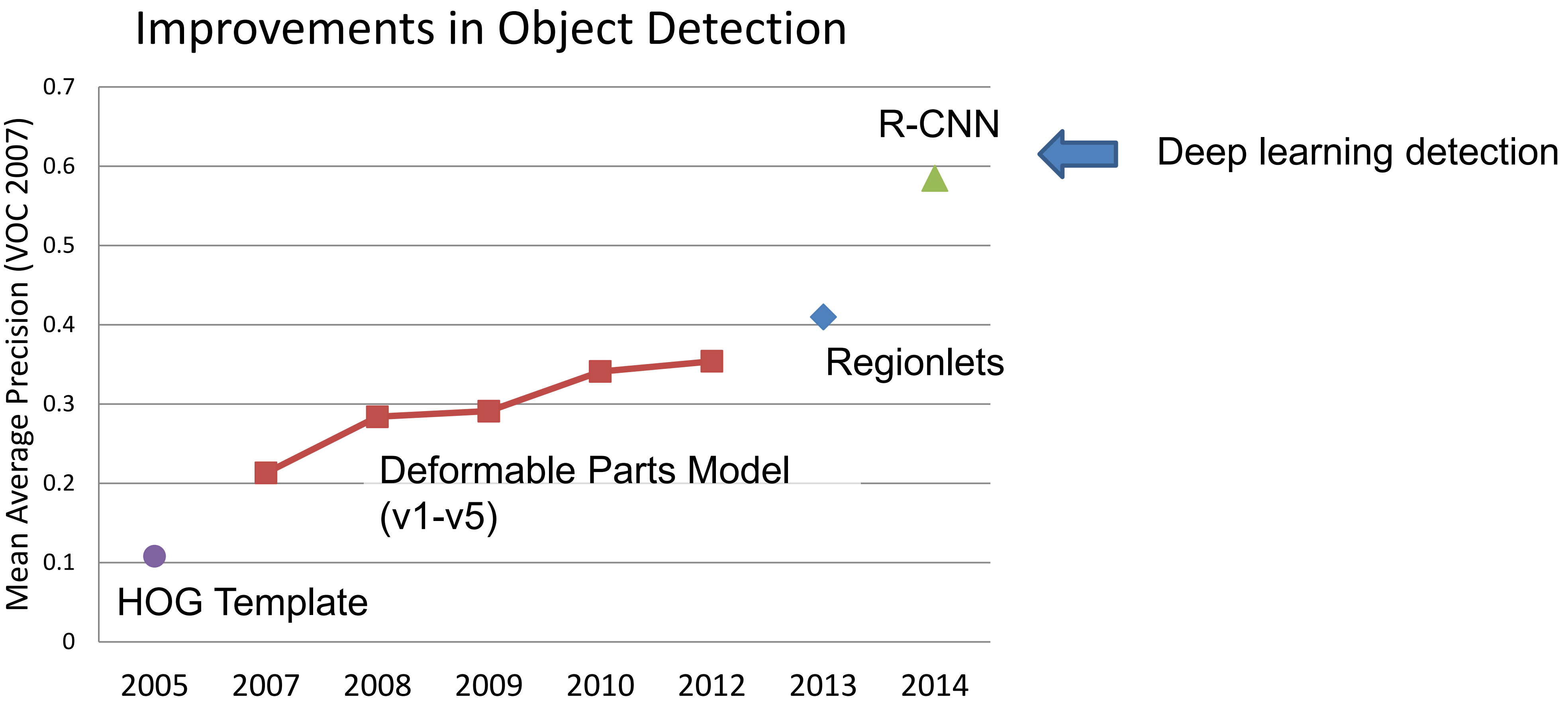
1. Load pre-trained model
2. Replace last layer
3. Set a low learning rate (e.g. lr=e-4)
   - Very sensitive to learning rate because you want to improve but not drift too far from the initial model
   - Learning rate is often at least 10x lower than from “scratch” training
   - Can “warm start” by freezing 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)
```

In this example, last layer has 512 input features and is called “fc”

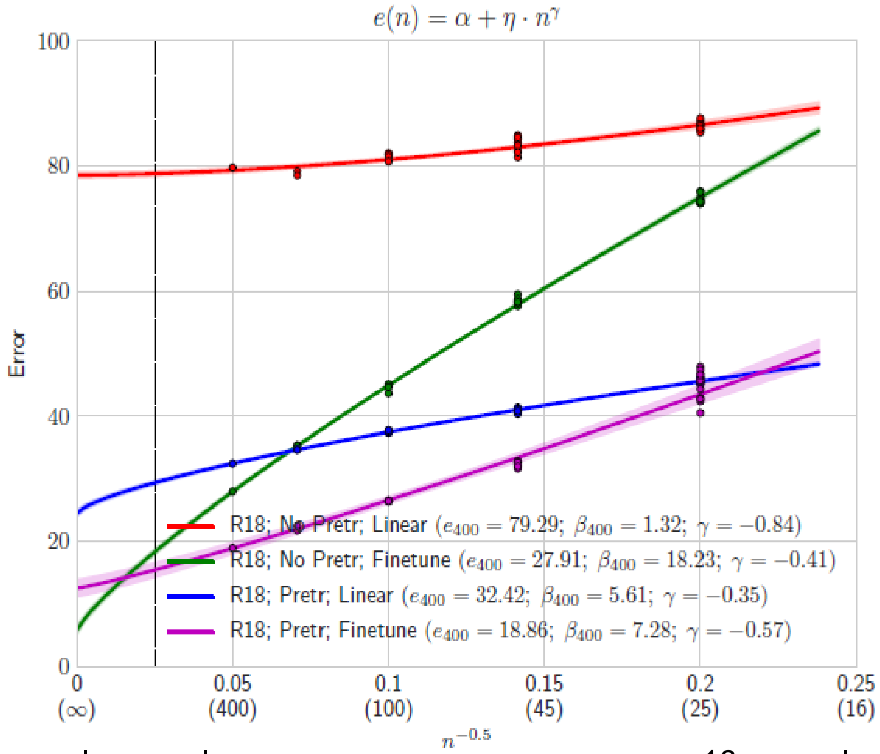
Other examples of layer customization (from ‘23 TA Weijie)

R-CNN first demonstrated major detection improvement by pre-training on ImageNet and fine-tuning on PASCAL VOC



HOG: Dalal-Triggs 2005   DPM: Felzenszwalb et al. 2008-2012   Regionlets: Wang et al. 2013   R-CNN: Girshick et al. 2014

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



400+ examples per class

16 examples per class

(a) Transfer: ImageNet to Cifar100

ResNet18, Err vs # examples / class (in paren)

Green: Train from scratch
Blue: Linear Probe from ImageNet
Purple: Fine-tune from ImageNet

Very little data
- Use linear probe on pre-trained model

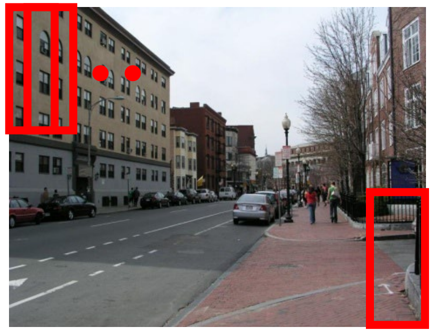
Moderate data
- Fine-tune pre-trained model

Very large dataset
- Either fine-tune or train from scratch

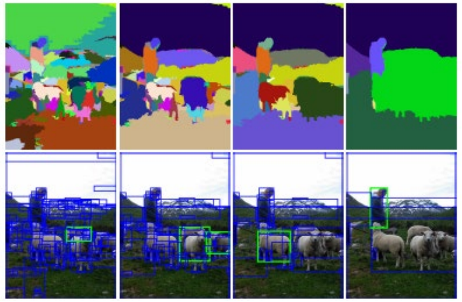
"Learning Curves" (2021) [pdf]

Statistical template approach to object detection

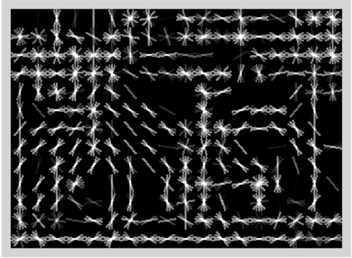
Propose Window → Extract Features → Classify → Post-process



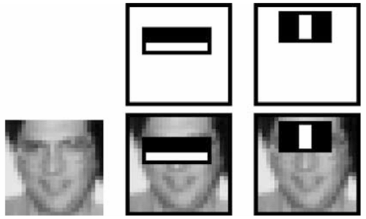
Sliding window: scan image pyramid



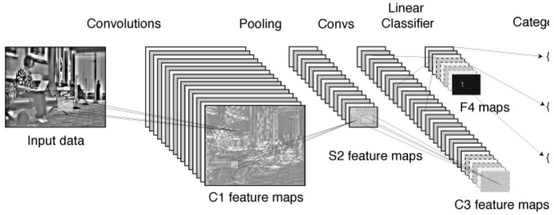
Region proposals: edge/region-based, resize to fixed window



HOG



Fast randomized features



CNN features

SVM

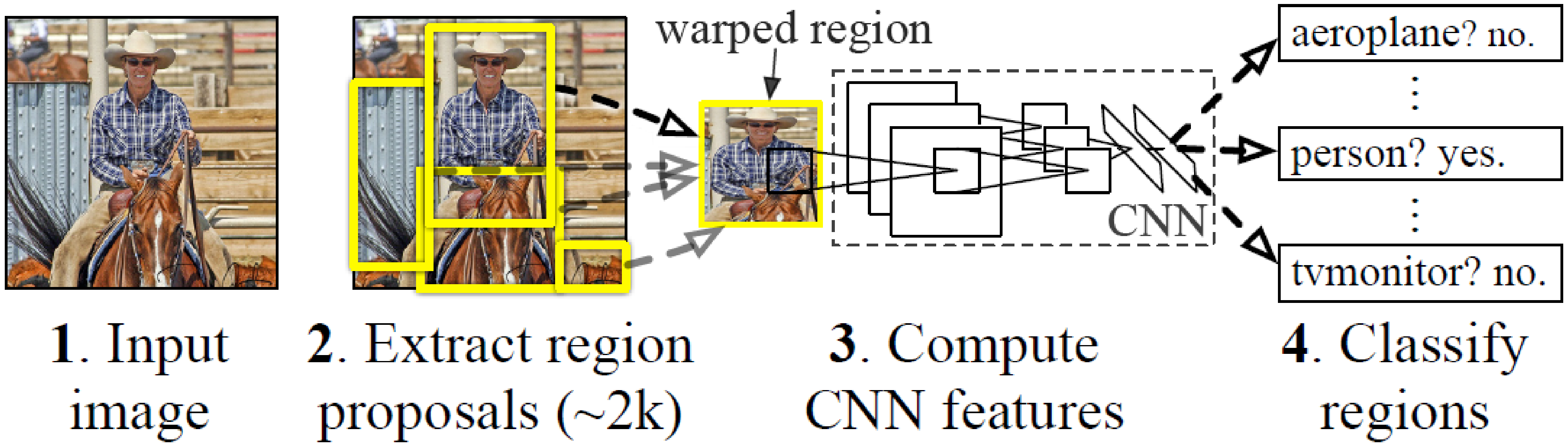
Boosted stubs

Neural network

Non-max suppression

Segment or refine localization

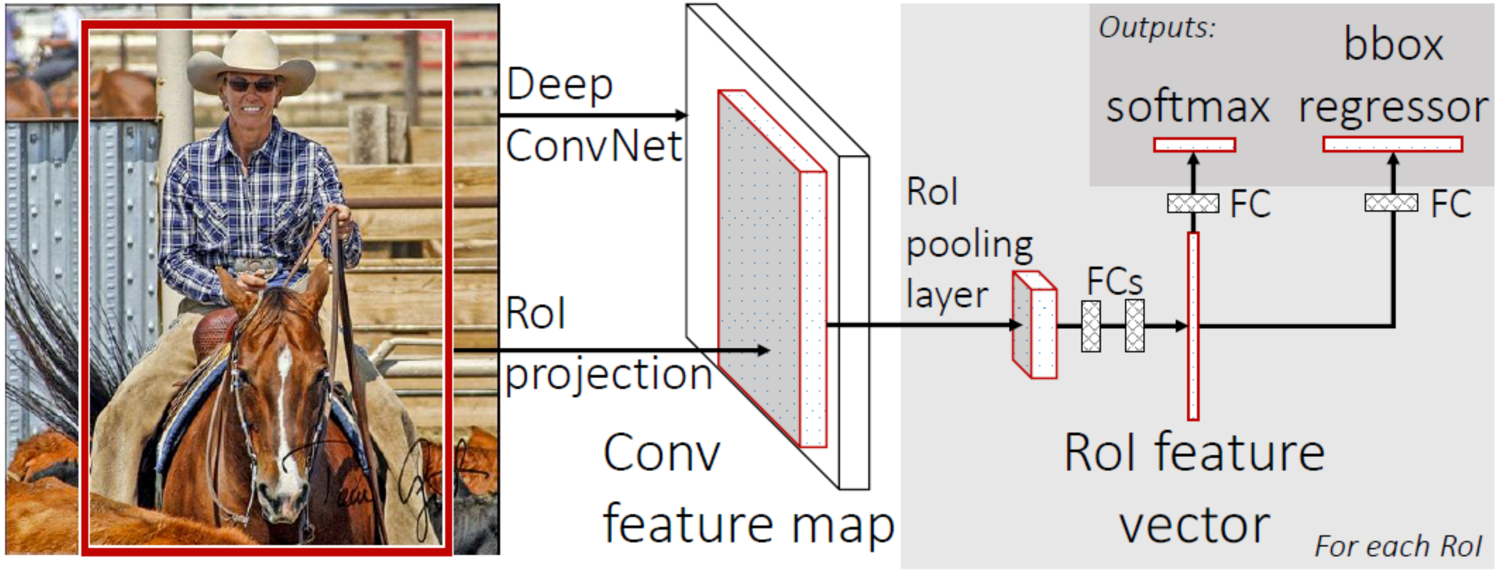
R-CNN (Girshick et al. CVPR 2014)



- 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

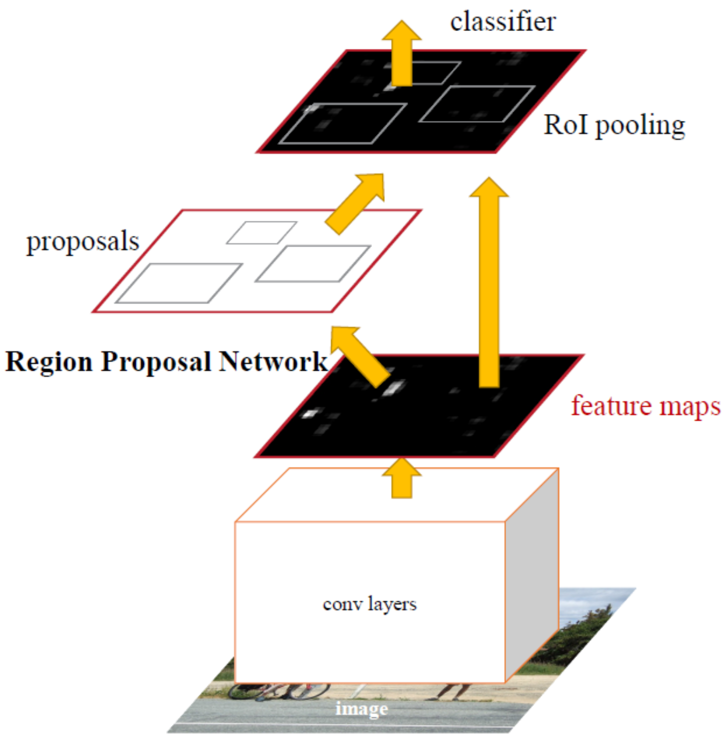
http://arxiv.org/pdf/1311.2524.pdf

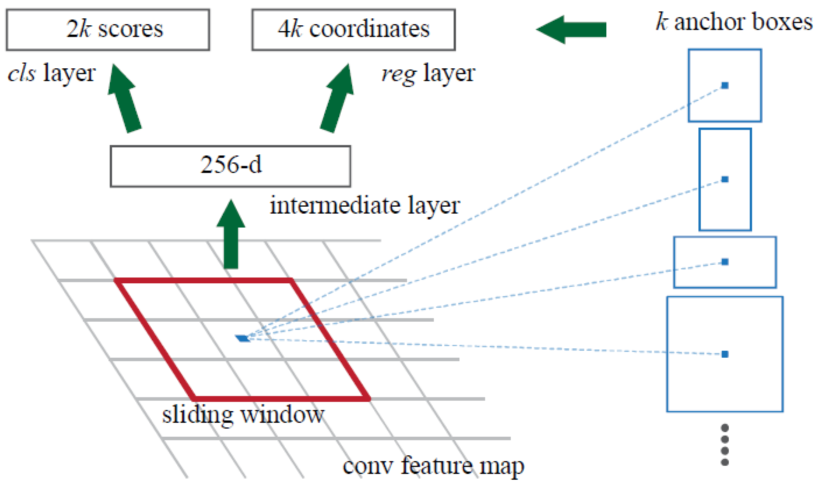
Fast R-CNN – Girshick 2015



- 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

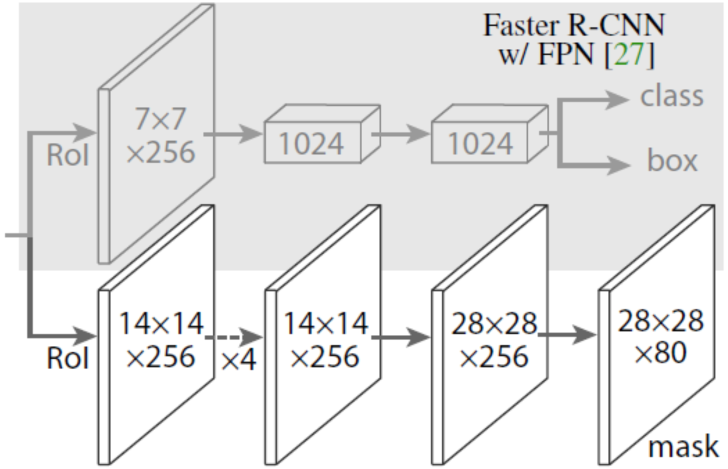




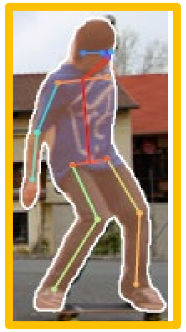
- Convolutional features used for generating proposals and scoring
  - Generate proposals with “objectness” scores and refined bboxes for each of k “anchors”
  - Score proposals in same way as Fast R-CNN
- Similar accuracy to Fast R-CNN with 10x speedup

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

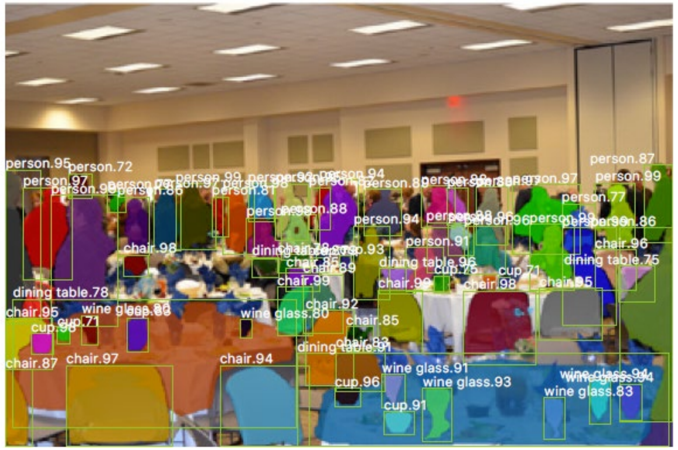
	backbone	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
MNC [10]	ResNet-101-C4	24.6	44.3	24.8	4.7	25.9	43.6
FCIS [26] +OHEM	ResNet-101-C5-dilated	29.2	49.5	-	7.1	31.3	50.0
FCIS+++ [26] +OHEM	ResNet-101-C5-dilated	33.6	54.5	-	-	-	-
Mask R-CNN	ResNet-101-C4	33.1	54.9	34.8	12.1	35.6	51.1
Mask R-CNN	ResNet-101-FPN	35.7	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

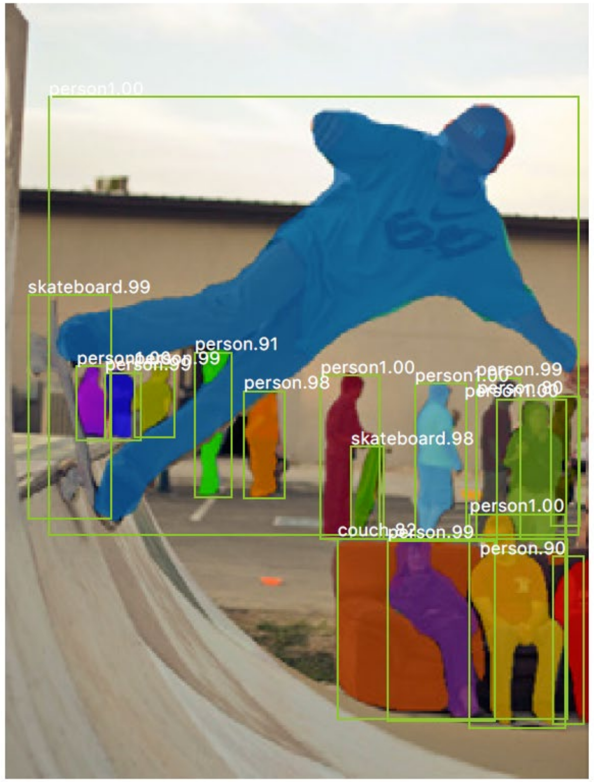
	AP^{kp}	AP^{kp}_{50}	AP^{kp}_{75}	AP^{kp}_{M}	AP^{kp}_{L}
CMU-Pose+++ [6]	61.8	84.9	67.5	57.1	68.2
G-RMI [31]†	62.4	84.0	68.5	**59.1**	68.1
Mask R-CNN, keypoint-only	62.7	87.0	68.4	57.4	71.1
Mask R-CNN, keypoint & mask	**63.1**	**87.3**	**68.7**	57.8	**71.4**

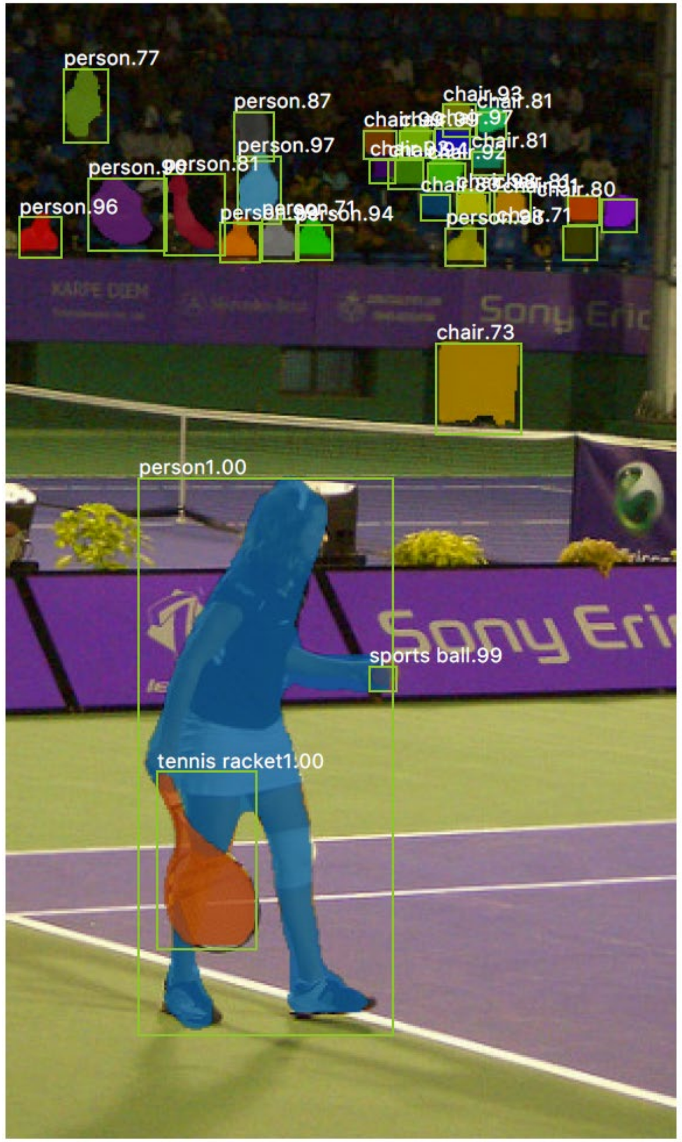
Table 4. **Keypoint detection** AP on COCO test-dev. Ours

Example detections and instance segmentations



Example detections and instance segmentations

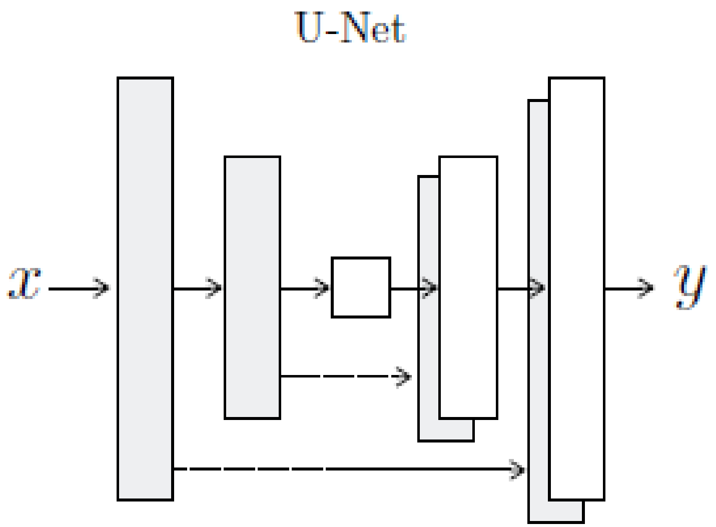


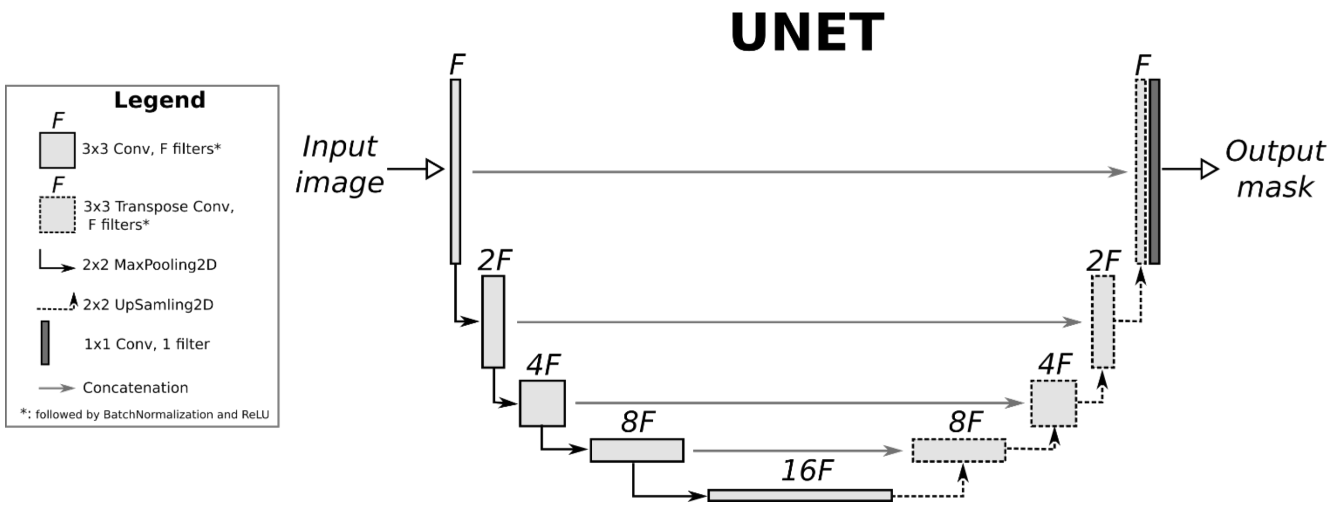


Example keypoint detections

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)

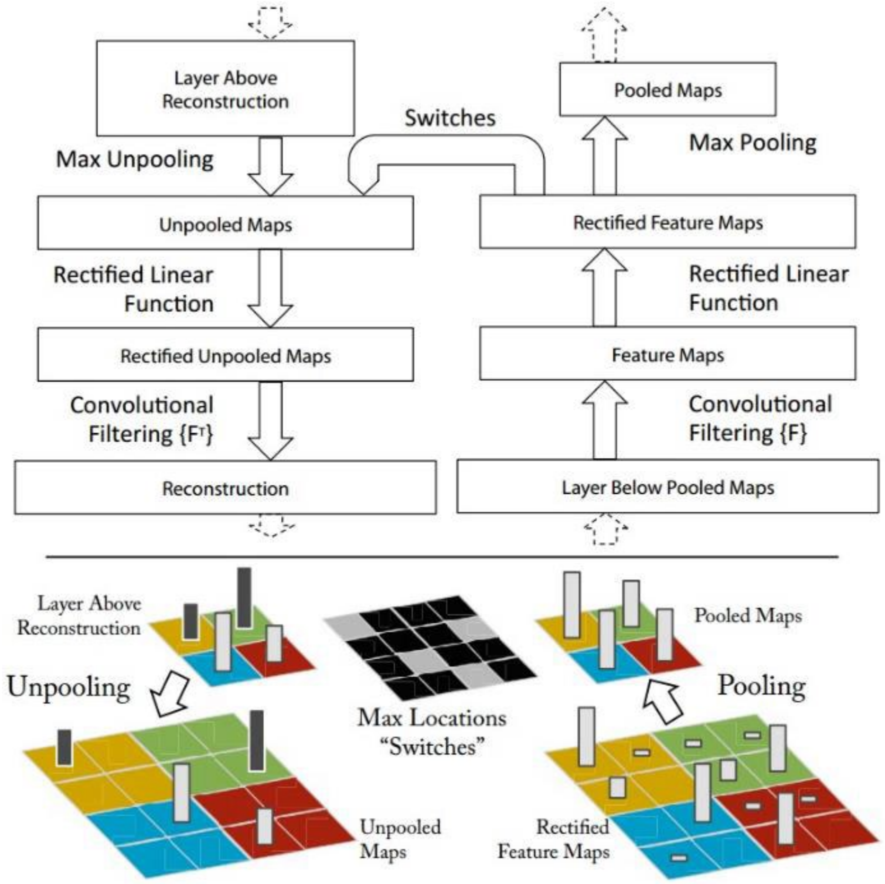
Fig from Isola et al. 2017

Fig src

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]

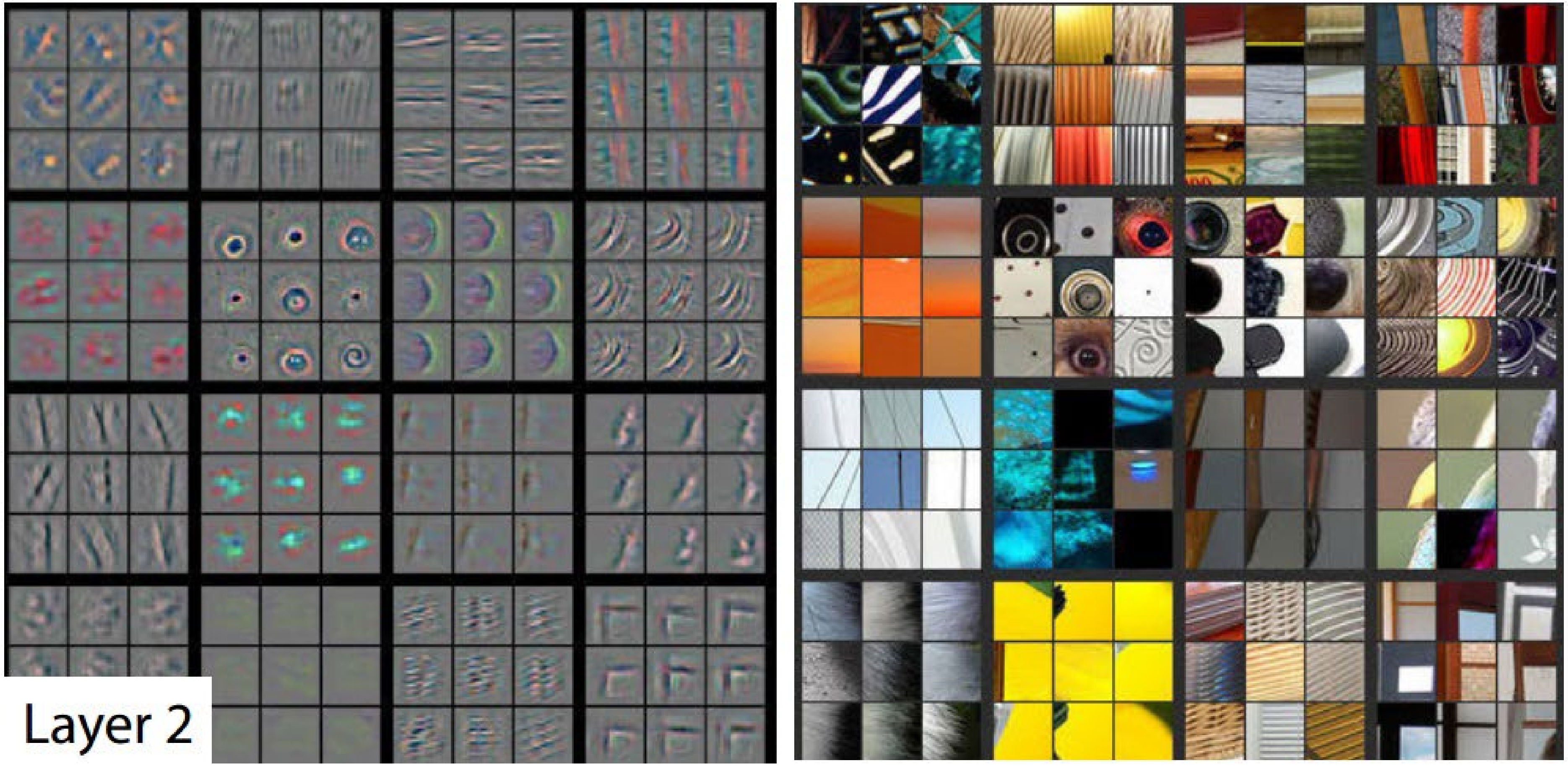
Layer 1 (visualization of randomly sampled features)

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

Image patches that had high activations

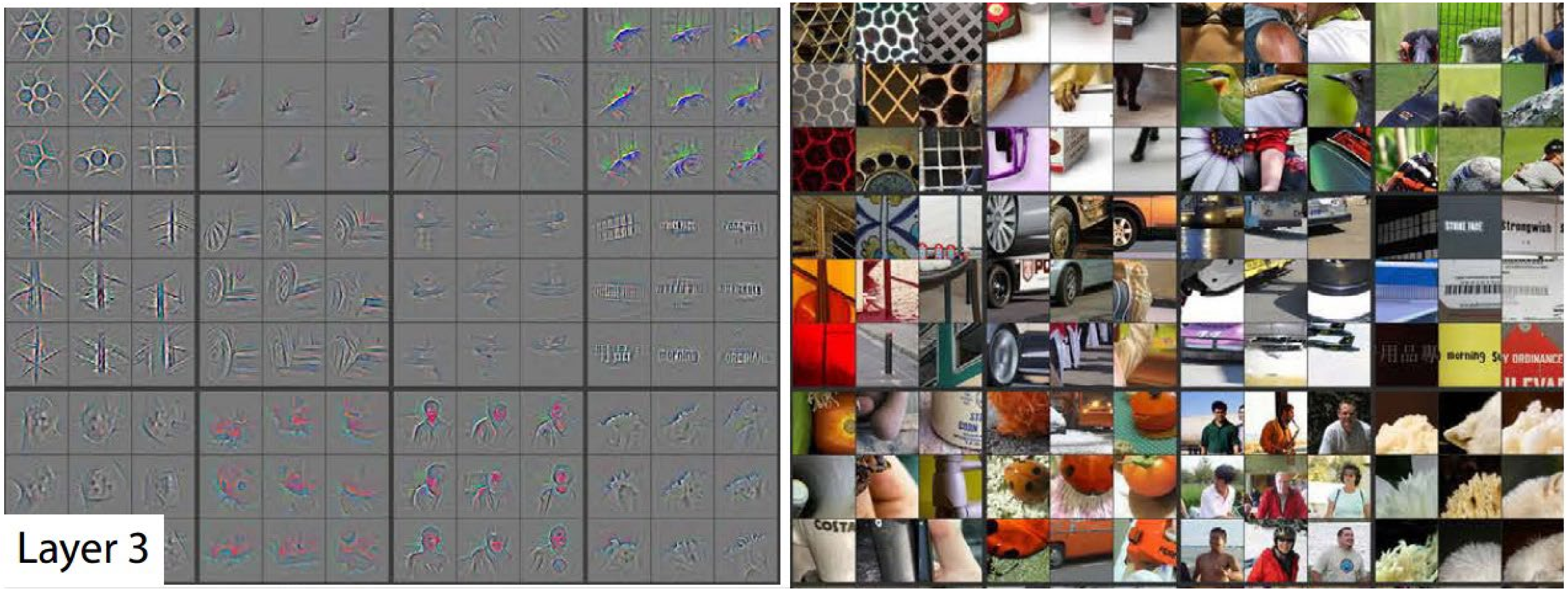
Visualizing and Understanding Convolutional Networks [Zeiler and Fergus, ECCV 2014]

Layer 2



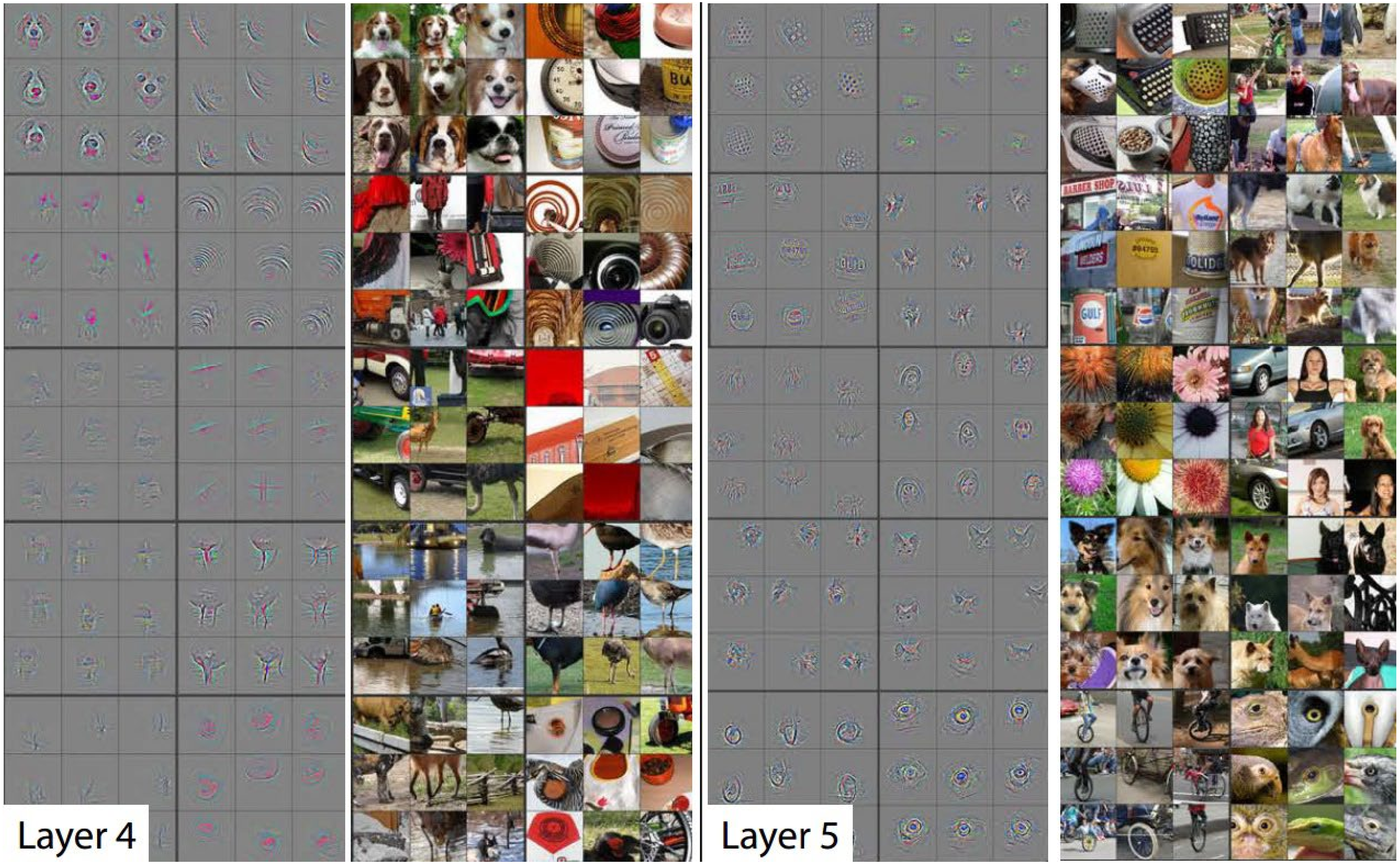
Visualizing and Understanding Convolutional Networks [Zeiler and Fergus, ECCV 2014]

Layer 3



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

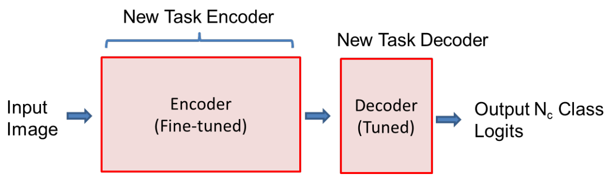
Layer 4 and 5



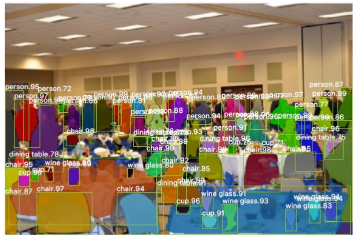
Visualizing and Understanding Convolutional Networks [Zeiler and Fergus, ECCV 2014]

Things to remember

- 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