$$f \otimes h = \sum_k \sum_l f(k,l)h(k,l)$$

f = Image

h = Kernel

Filtering, Convolution
Hue represents color in angle of HSV cone
0-60: red 60-120: yellow 120-180: green
80-240: cyan 240-300: blue 300-360: magenta
Filtering: Noise removal, feature extraction
Linear filtering: output is a linear combination of pixel values in some neighborhood

f ⊗ h

f_1	f_2	f_3
f_4	f_5	f_6
f_7	f_8	f_9

⊗

h_1	h_2	h_3
h_4	h_5	h_6
h_7	h_8	h_9

→ $f \otimes h = f_1h_1 + f_2h_2 + f_3h_3 + f_4h_4 + f_5h_5 + f_6h_6 + f_7h_7 + f_8h_8 + f_9h_9$

Fourier Transform

Definition

$$F(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-j2\pi(ux+vy)}\,dx\,dy,$$

$$f(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(u,v)e^{j2\pi(ux+vy)}\,du\,dv$$

Image Frequency: obtained quantitatively using 2D Fourier Transform,
Fourier analysis -> decompose complex waves into simple constituent simple sin/cos waves

Low Pass Filtering: retains low spatial freq, remove high spatial freq components (noise, texture)

$\frac{1}{9}$

1	1	1
1	1	1
1	1	1

> generally image becomes blurry. bigger kernel -> more blurry
> denoising: low-pass filtering to remove signal comps with high spatial freq
> i.e. gaussian blur fx = ae^((x-b^2) / (2c^2))
> i.e. average blur (takes average of pixel values in kernel)
> i.e. non-linear filtering/median filtering -> takes median of kernel as output more robust against outliers (salt n pepper), computationally intense

High Pass Filtering: retains high spatial freq components

−1	−1	−1
−1	8	−1
−1	−1	−1

> Additive Gaussian Noise: I(distorted) = I(original) + n (where n is norm dist) generally more contrast but also more grainy
> Edge detection: detect sigificant change in intensity values. provide shape info/sharpening
> i.e. prewitt operator to detect horizontal or vertical edges

(prewitt computes diff of neighbouring pixels to indicate likelihood of edges)

[[-1,0,1], [-1,0,1], [-1,0,1]] Detect vertical edge

[[-1,-1,-1], [0, 0, 0], [1, 1, 1]] Detect horizontal edge

> i.e. Sobel operator: similar to prewitt but has larger weights for edges near centre output edges are sharper than prewitt

Image Classification
Top-n accuracy: outputs k confidence for each of the k classes, generates top 1/2 etc. classes
eval: no. correct /no. images
Object localization: produce class label and bounding box for each input image
eval: label best matching GT, bounding box overlapping GT.
Error if label is incorrect or box has <50% overlap
Object Detection: model predicts sets of labels, bounding boxes and confidence scores.
Penalizes objects in image that are not annotated by model or >1 label for the same object
Challenges in image classification: Primitive data (arrays of values), Diff variation for every class (i.e. perspective, scale, clutter, occlusion, intra-class variations, deformation, lighting)
Data driven approach to solve: provide Big Data. Use learning algorithm.

Training/Learning (usually offline)

Training set: images with known class information → Learn a model using some algorithm → A model for this specific classification problem and classifier

Testing/Evaluation (usually online)

Testing set (with label during evaluation, without label in an application) → Classifier algorithm → Predicted class information for this new image (compare with ground-truth during evaluation)

Linear Classifier: L = f(x; W, b) = Wx + b OR s = [W b] [x 1].T
Given a test image x, produce the confidence score for each class using linear transformation (total: Kclasses) Higher confidence score for a class -> more likely to be the ground-truth class
Test image x: flatten to a Dx1 column vector, D is the image resolution times number of channel:

$$w_k^T x_i = \|w_k\|\|x_i\| \cos\theta$$

Measures similarity btw wk and xi

Input x: Dx1 Weight W: KxD
Bias b: Kx1 Score s:Kx1
Each row of W extracts features of a specific class
Loss Function: meausres consistency of GT labels and score output for some W

Softmax Classifier
Regard output of the score function f(x; W) as the unnormalized log probabilityof each class

softmax function (exp, then normalize): exp helps avoid small numbers -> rounding/floating point errors

$$\text{softmax}(f) = \frac{e^{f_m}}{\sum_{j=1}^{K} e^{f_j}}$$

For the m-th class

Cross-entropy loss (apply –log(.) to only the ground-truth class):

$$L_i = -\log \frac{e^{f_{y_i}}}{\sum_{j=1}^{K} e^{f_j}}$$

For the i-th training sample

Softmax: Loss for N training samples (xi, yi): L = (1/N) * sum(Li)
Output is normalized -> confidence across classes sum to 1
Cross-entropy Loss: 1. softmax (exp, normalize) 2. -log(GT class)
> Q: estimated prob (softmax) P: true prob (GT label)
Kullback-Liebler (KL) divergence: DKL(P||Q) = -sum(P(i) * log(Q(i) / P(i))) goal: Q is close to P
Multiclass SVM Loss: correct class has higher score than the incorrect by a fixed margin d
Li = sum(max(0, sj-syi+d))
sigmoid(f) = 1 / (1 + e^-f)
tanh = (e^z - e^-z)/ (e^z + e^-z)
- saturation, vanishing gradient
- slow and difficult to train with grad desc
ReLU = max(0, z)
- reduce issue of vanishing grad
- sparse representation

$$D_{KL}(P||Q) = -\sum_i P(i)\log\frac{Q(i)}{P(i)} = \sum_i P(i)(\log\frac{1}{Q(i)} - \log\frac{1}{P(i)})$$

$$D_{KL}(P||Q) = -\sum_x p(x)\log q(x) + \sum_x p(x)\log p(x)$$

can be log or ln

$$= H(P,Q) - H(P)$$

Cross entropy of P and Q

Entropy of P H(P) = 0 in this case

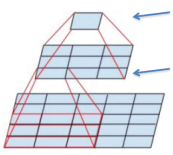
CNN
> Image filtering is used as feature detection. Produces large outputs when patterns similar to the filter is detected in the image. Different filters used to detect different features
> Training images are used to find out what filter kernels are useful
> Feature detection used in hierarchical manner to recognize object/high level concept
Convolutional Layer: provides local connectivity (receptive field, depth of input), shares params, uses filtering/convolution instead of matrix multiplication to detect features (edge, corner, faces)
> Assumption of image (locality of pixel dependencies, stationary of image statistics, translation invariance, using the same set of filters for the whole image)
Pooling Layer: progressively reduce spatial size of feature map, reduce model params
> operates independently on each feature map
> can be overlap or non-overlap, avg or max pool, is translation invariant
Receptive Field:Part of the image visible to a neuron. Neurons in higher layers can see large portions of the image

Stride = 1

Conv 2

Conv 1

Input layer (image)



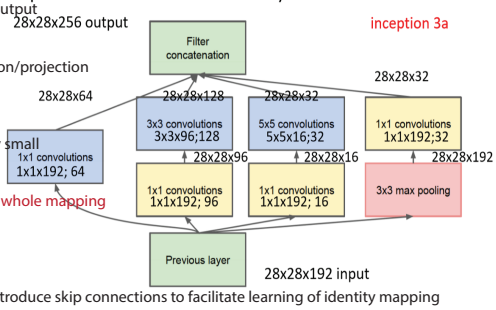
- 3x3 receptive field size w.r.t. conv1
- 5x5 receptive field size w.r.t. input image

3x3 receptive field size (filter size)

Output size of conv = ((N-F)/stride) - 1

Inception Module
> use filters of diff size in the same layer, concat results as output
> 1x1 conv filter to reduce complexity (no. channels)
> increase width and depth of network
> all convolutions use ReLU, including 1x1 conv for reduction/projection

- Inception module with dimensionality reduction



Challenges of going deeper:
Vanishing gradient: gradient is backpropagated to earlier layers, repeated multiplication may make the gradient very small
Difficult to learn an identity mapping in deep architecture

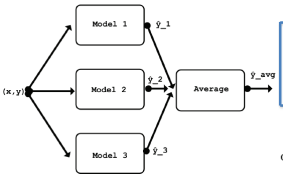
Deep Residual Learning: learn residual mapping instead of whole mapping
> optimal mapping is close to identity mapping, residual learning is a construction to ease the learning
Performs elem-wise addition channel by channel
Optimization in complex large scale system:
> to ensure going deeper doesn't degrade performance, introduce skip connections to facilitate learning of identity mapping
Ensemble Learning:
> Residue networks r like a collection of paths with many lengths. They sdhow ensemble-like behaviour (don't strongly depend on each other) -> most gradients come from short paths (10-34 layers deep) to avoid vanishin gradient
> Residual networks behave like ensembles of relatively shallow networks

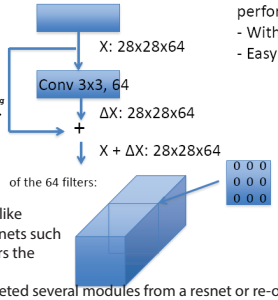
Deep residual learning

– Learn the residual mapping instead of the entire mapping

Want ΔX to be small, so that the output feature maps would not cause dramatic change (degrade) in performance
- With skip connection, need W ≈ 0
- Easy to learn with regularization on W

Ensemble Method Learning:
> multiple learning algos to improve prediction
> each model trained independently
> inputs are eval mutiple times
> smooth performance wrt no. models





"We hypothesize that it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping. To the extreme, if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers."

deleting a layer in resnet at test time is like zeroing half the paths. in feed-forward nets such as AlexNet or VGG, deleting a layer alters the only path from input to output.
Error also increases smoothly when deleted several modules from a resnet or re-ordering the net by shuffling the modules. Resnet distribution of all path lengths follows a binom distribution. Gradient introduced appears to decay exponentially with no. modules

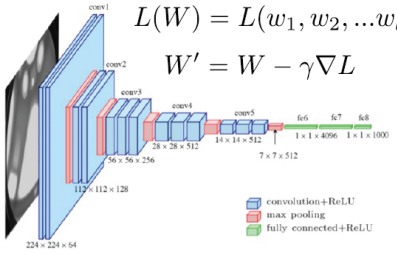
Gradient Descent: Method of learning W
Using the loss function L(W): L = 1/N * sum(Li + lr * R(W)) = L(W; (x1, y1), (x2, y2), (x3, y3)...)
> Grad Desc: Start from random W, iteratively improve W (reduce L(W))

- Update W by W+ΔW, using the gradient

$$L(W) = L(w_1, w_2, ...w_l...)$$

$$W' = W - \gamma\nabla L$$

Gradient descent



$$\nabla L = [\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, ... \frac{\partial L}{\partial w_l}, ...]^T$$

$$w_l' = w_l - \gamma\frac{\partial L}{\partial w_l}$$

Find $\frac{\partial L}{\partial w_l}$ for every w_l

γ: step size (learning rate), a hyperparameter

Gradient is perpendicular (orthogonal) to the level curve. Steepest direction (steepest descent)

- Update W by W+ΔW, using the gradient

$$w_l' = w_l - \gamma\frac{\partial L}{\partial w_l}$$

$$L = \frac{1}{N}\sum_i L_i + \lambda R(W)$$

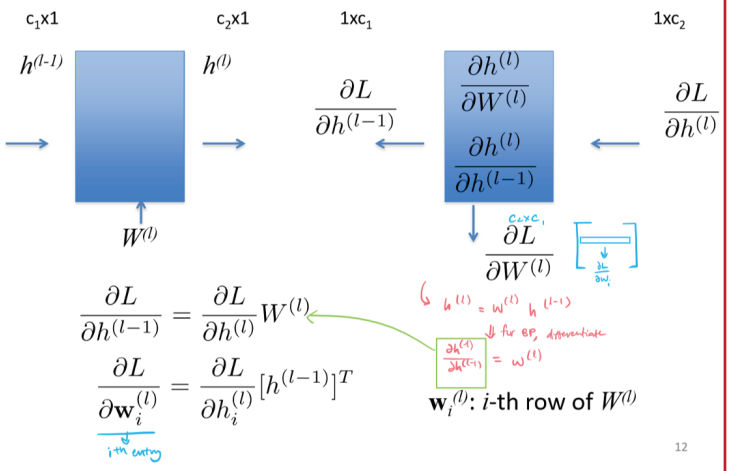
$$\frac{\partial L}{\partial w_l} = \frac{1}{N}\sum_i \frac{\partial L_i}{\partial w_l} + \lambda\frac{\partial R(W)}{\partial w_l}$$

-Sum gradients for all (partial) training samples for one w_l
-Make one update of W once we have **all the gradients**
Mini-batch gradient descent / stochastic gradient descent: use small batch (64, 128, 256) for one update of W

$$\frac{\partial L}{\partial w_l} = \frac{1}{N_{batch}}\sum_i \frac{\partial L_i}{\partial w_l} + \lambda\frac{\partial R(W)}{\partial w_l}$$

Computing Gradient: 2 approaches
1. Compute L(Wl) then differentiate > tedius and inflexible (if layers change, need to re-compute all)
2. Backprop > assume we have dL/dh
1. Grad descent (forward): dL/dW(l) = dL/dh(l) x dh(l)/dW(l)
2. Backprop: dL/dh(l-1) = dL/dh(l) x dh(l)/dh(l-1)
Modular design for backprop :
> Each module knows how to compute local gradients. Multiply by the global gradient from upstream
> Generalize to other modules, e.g. ReLU
> Obtain exactly same gradient as Approach 1

Back prop: fully connected



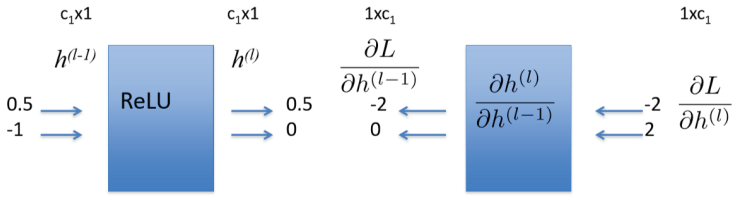
$$\frac{\partial L}{\partial h^{(l-1)}} = \frac{\partial L}{\partial h^{(l)}} W^{(l)}$$

$$\frac{\partial L}{\partial \mathbf{w}_i^{(l)}} = \frac{\partial L}{\partial h_i^{(l)}} [h^{(l-1)}]^T$$

$\mathbf{w}_i^{(l)}$: i-th row of $W^{(l)}$

12

Back prop: ReLU



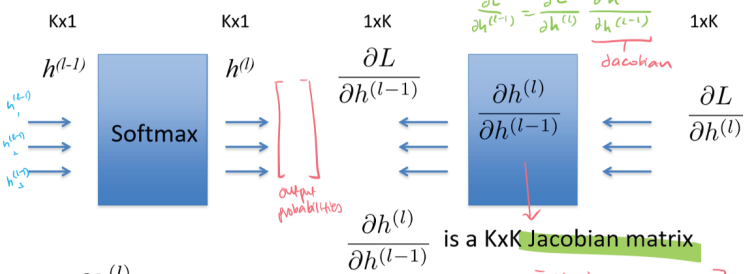
$$h_i^{(l)} = \max(0, h_i^{(l-1)}) \qquad \frac{\partial h_i^{(l)}}{\partial h_i^{(l-1)}} = \mathbf{I}(h_i^{(l-1)} > 0)$$

$\mathbf{I}$(c) = 1 if c is true, 0 otherwise

Gate: gradient can or cannot pass through

16

Back prop: Softmax

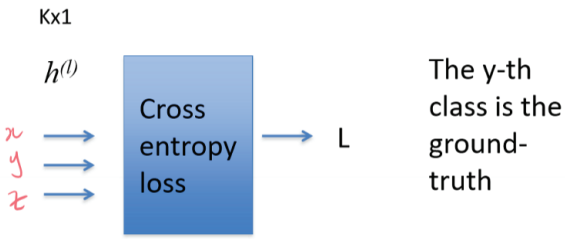


$\frac{\partial h^{(l)}}{\partial h^{(l-1)}}$ is a KxK Jacobian matrix

$$\frac{\partial h_i^{(l)}}{\partial h_j^{(l-1)}} = h_i^{(l)}(1 - h_j^{(l)}) \qquad i = j$$

$$\frac{\partial h_i^{(l)}}{\partial h_j^{(l-1)}} = -h_i^{(l)} h_j^{(l)} \qquad i \neq j$$

Back prop: Cross entropy loss



The y-th class is the ground-truth

$$L = -\log(h_y^{(l)})$$

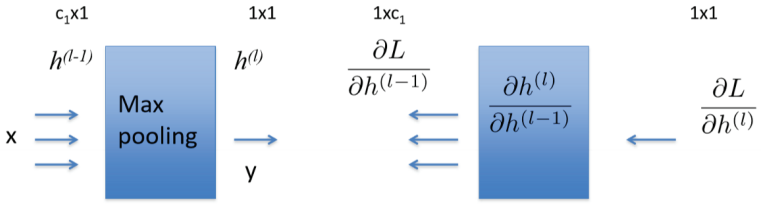
GT class

$$\frac{\partial L}{\partial h^{(l)}} = [0, 0, ..., \frac{-1}{h_y^{(l)}}, ..., 0]$$

Back prop starts from the loss function

Back prop: max pooling

Takes the largest number in window (reduce param space)



If x is max, y=x, dy/dx = 1

If x is not max, y and x are independent, dy/dx=0

If not max, will not affect loss function

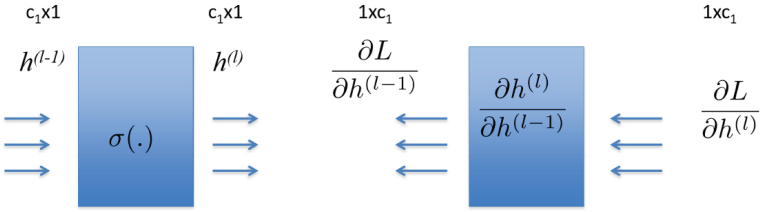
$$\frac{\partial h_i^{(l)}}{\partial h_i^{(l-1)}} = \mathbf{I}(h_i^{(l-1)} \text{ is max}) \qquad \frac{\partial L}{\partial h_i^{(l-1)}} = \frac{\partial L}{\partial h^{(l)}} \mathbf{I}(h_i^{(l-1)} \text{ is max})$$

$\mathbf{I}$(c) = 1 if c is true, 0 otherwise

Only 1 branch has gradient, and gradient goes to the max input branch

13

Back prop: Sigmoid



$$h_i^{(l)} = \sigma(h_i^{(l-1)}) \qquad \frac{\partial h_i^{(l)}}{\partial h_i^{(l-1)}} = \sigma(h_i^{(l-1)})(1 - \sigma(h_i^{(l-1)}))$$

Softmax, $h^{(l)} = \left[\frac{e^2}{e^2+e^{-0.6}}, \frac{e^{-0.6}}{e^2+e^{-0.6}}\right] = [0.931, 0.069]$

$L = -\log(0.931) = 0.07$

Backwards:

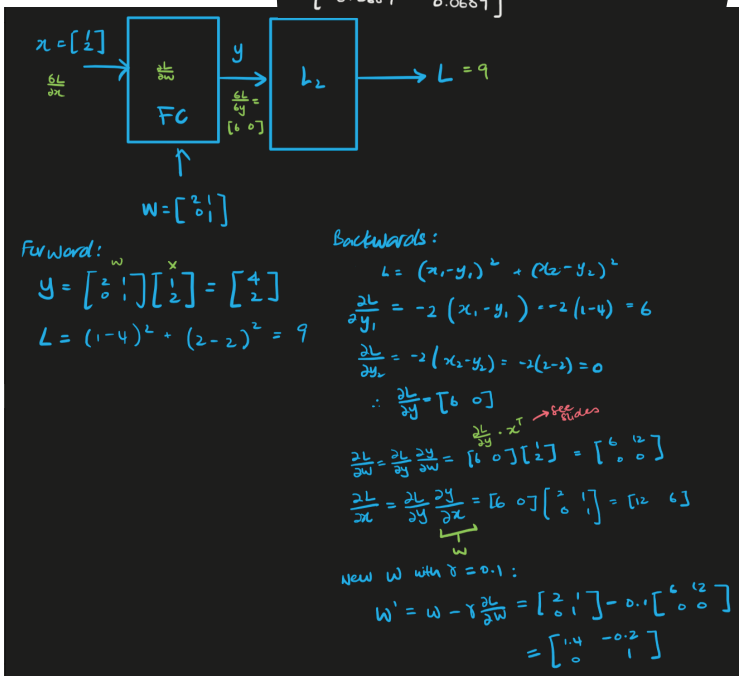
$\frac{\partial L}{\partial h^{(l)}} = \left[-\frac{1}{0.931}, 0\right] = [-1.0743, 0]$

Softmax Backwards $\frac{\partial L}{\partial h^{(l-1)}} = \frac{\partial L}{\partial h^{(l)}} \frac{\partial h^{(l)}}{\partial h^{(l-1)}} = [-1.0743, 0] \begin{bmatrix} 0.931(1-0.931) & -0.069(0.931) \\ -0.069(0.931) & 0.069(1-0.069) \end{bmatrix}$

Follow algorithm for Softmax BP in the prev slides!

$= [-1.0743, 0] \begin{bmatrix} 0.0642 & -0.0642 \\ -0.0642 & -0.0642 \end{bmatrix}$

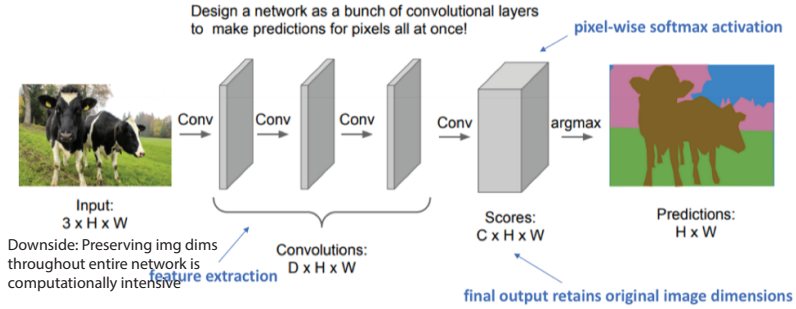
$= [-0.0689 \quad 0.0689]$



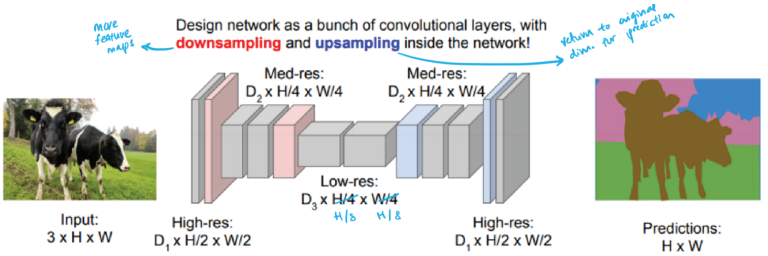
Semantic Segmentation

With Conv Nets: pass a block of the image to conv net, for each block, compute class label of centre pixel (pixel wise classification) -> computationally intensive, not easy to share information, difficult to determine ideal patch size (to balance context vs localization accuracy)

Pixel-wise Softmax Loss: L = spatial sum of loss at each pixel (probability vector of C classes for each pixel, dim: HxWxC)



Downside: Preserving img dims throughout entire network is computationally intensive



Soln: make network deep and work at lower spatial resolution for many of the layers (downsample and upsample)

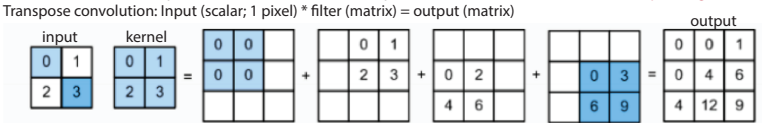
Upsampling Methods:

1. Unpooling: inverse of max/avg pooling. e.g. Nearest Neighbour > duplicate values to create larger feature map
2. Bed of Nails: fill with 0 (like 0 padding for each pixel)
3. Transpose Convolution (Learnable upsampling):

Standard convolution: Input (matrix) * filter (matrix) = output (scalar)

Transpose convolution: Input (scalar; 1 pixel) * filter (matrix) = output (matrix)

Outputsize of transpose conv = (N - 1)*stride - 2*padding + (F-1) +1



Fully Convolutional Network:

> Uses well pre-trained network as encoder, transpose convolution upsample coarse feature map to full seg map, trained end-end, pixel-pixel using backprop. Made up of both FC and Fully convolutional layers.

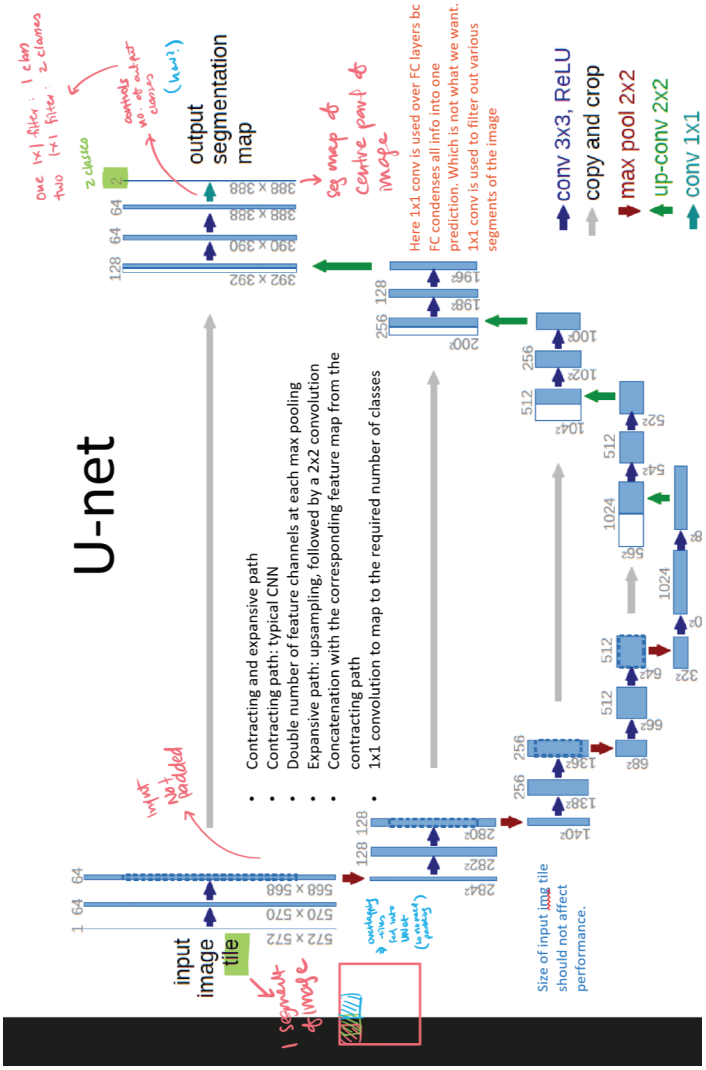
> Involves FC layers that discard spatial info (like convolutions where F=N and 4096 masks) --> no spatial info, bad for SS

Output: 1x1x4096 feature map

> Fully convolutional: uses filter to some spatial extent

> global info tells you what the object is (shallow, appearance), local info tells you where (deep, semantic) > combined using pooling and upsampling to produce output SS image

Challenges in SS: Balancing and combining fine and coarse layers (spatial vs semantic info)



Loss = sum of weighted loss at each pixel $w(\mathbf{x}) = w_c(\mathbf{x}) + w_0 \cdot \exp\left(-\frac{(d_1(\mathbf{x}) + d_2(\mathbf{x}))^2}{2\sigma^2}\right)$

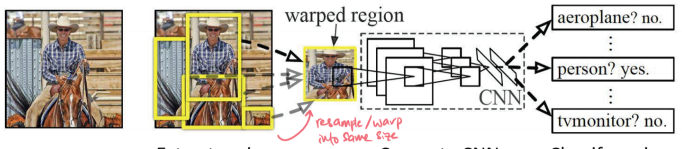
Eval: Intersection over Union (IOU) = overlap/union

Object Detection: Finding diff objects in an image and classify them

Challenges: illumination, perspective, clutter, occlusion, pose, intra-class variations (i.e. diff species of cat)

1. Template matching: output correlation map of image against template (poor against images with many obj)

Regional CNN (R-CNN): Uses a set of region proposals to predict class label for the section of image given as inp



Extract region proposal (~2K) | Compute CNN feature | Classify regions

Process:

1. Region Proposal Extraction (e.g. SS) -> get regions which have high prob of having an object

> M1: Slide window over image, do exhaustive search & classify each patch (all locations/scale/ratios) -> computationally intensive

> M2: Selective Search: output BB corresponding to all patches w high prob of containing obj. Can be noisy, overlapped, may not contain full obj. Some will be close to the correct obj. High recall: many TP and FP.

> M3: Selective Search by SS and hierarchical grouping (grp similar size/texture/color/shape)

1. SS over-segmentation based on pixel intensity
2. Add BB based on SS
3. Merge adj segments based on similarity (bottom up approach) -> extract histogram bins, merge based on intersection (i.e. similarity(ri, rj) = sum(min(cij, cik)) -> sum smaller val of comparison btw every 2 bins
4. repeat 2-3 until properly segmented

2. Feature Extraction: Crop & scale section of image based on region proposal for each proposal bounding box

Challenge: labelled data in detection is insufficient to train large CNN

--> discriminative pre-training on large aux dataset (e.g. imagenet) then do domain specific fine-tuning (warped) Domain Specific Fine Tuning: Replace specific 1000 way classification layer with random init (N+1) way classification layer (N classes + 1 bg class). Smaller lr is used in SGD so W doesn't deviate too far from init values

Summary: For each RP, extract feature vector (robust, discriminative, low-dim feature representations) -> softmax

3. Classification

> One linear binary SVM per class

4. Bounding Box Regression

> Bounding box regression: Original bb from selective search may not be accurate

> Predict new bb with new params: Given P(x, y, w, h) (bb from selective search), try predict G(x, y, w, h) (GT label)

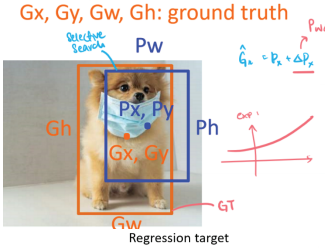
> Assume each function is a linear function of the pool5 features of the region proposal P: sig5(P)

> Goal: dx(P) ~ (Gx - Px)/Pw and dw(P) ~ log(Gw/Pw)

Scale invariant translation

Px, Py, Pw, Ph: selective search result

Gx, Gy, Gw, Gh: ground truth



$$\hat{G}_x = P_w d_x(P) + P_x$$
$$\hat{G}_y = P_h d_y(P) + P_y$$
$$\hat{G}_w = P_w \exp(d_w(P))$$
$$\hat{G}_h = P_h \exp(d_h(P)).$$

scaling

$$\mathbf{w}_\star = \underset{\hat{\mathbf{w}}_\star}{\operatorname{argmin}} \sum_i^N (t_\star^i - \hat{\mathbf{w}}_\star^{\mathrm{T}} \phi_5(P^i))^2 + \lambda \|\hat{\mathbf{w}}_\star\|^2$$

Prediction model

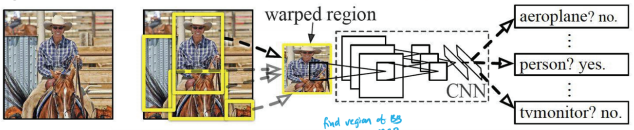
$$d_\star(P) = \mathbf{w}_\star^{\mathrm{T}} \phi_5(P)$$

Issues with Fast R-CNN:

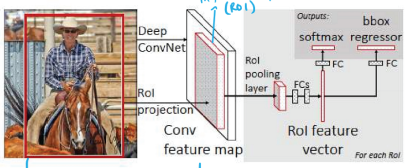
1. Computationally intensive (individual CNN forward pass for each RPs in each image, can be ~2k per img etc)

> Soln: Train 3 systems separately: feature extraction (domain specific fine-tuning), classifier (SVM), BB regression

Fast R-CNN



Input: image and region proposal (bounding boxes from selective search)



[Ross Girshick; 2015]

- Key ideas:

ROI pooling: Single pass of CNN to extract feature

> need to convert arbitrary projected region into fixed spatial size

> divide into hxw grid, max pool in each grid, size H/h x W/w. Pooling applied individually for each fmap channel

> for each ROI, 2 outputs: prob over K+1 classes, bounding box params

> Supervised training: For each ROI with GT label u and GT bb v, (multitask) Loss = classification loss wrt u and bb loss wrt v

$$L(p, u, t^u, v) = L_{\text{cls}}(p, u) + \lambda[u \geq 1] L_{\text{loc}}(t^u, v)$$

classification loss (K+1 classes): cross entropy

$$L_{\text{cls}}(p, u) = -\log p_u$$

$$L_{\text{loc}}(t^u, v) = \sum_{i \in \{\text{x,y,w,h}\}} \text{smooth}_{L_1}(t_i^u - v_i),$$

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise,} \end{cases}$$

OpenCV and Numpy [15 points]
(a) Given the following python code,

```
>>> a = [1,3,4,5,2,10]
>>> b = []
>>> for i in a:
....        temp = (i*5) %2
....        b.append(temp)
>>> print(b)
[1, 1, 0, 1, 0, 0]
```

Refactor the above python code using numpy and NO loops to achieve the same output (3 pts)

```
>>> import numpy as np
>>> a = [1,3,4,5,2,10]
### YOUR SOLUTION HERE
>>> a = np.array(a)
>>> b = (a*5) % 2
>>> b = list(b)
>>> print(b) # NOTE: b is a python list
[1, 1, 0, 1, 0, 0]
```

(b) As seen in Problem Set 1, we use the formula below to convert images to grayscale
0.299×R+0.587×G+0.114×B
Briefly explain why the implementation below of converting color images to grayscale is wrong. (2 pts)

```
>>> import cv2
>>> #args - image_path: String path of color image
>>> def rgb2gray(image_path):
        img = cv2.imread(image_path, cv2.IMREAD_COLOR)
        out = np.around( \
        img[:,:,0]* 0.299 \
         + img[:,:,1]*0.587 \
         + img[:,:,2]*0.114).astype(np.uint8)
        return out
```

OpenCV splices images in order of B,G,R instead of R,G,B hence the above indices for R and B are flipped.
(c) Given a 3-channel colour square image (10 pts):

```
>>> img = cv2.imread('sample.jpg', cv2.IMREAD_COLOR)
>>> img = cv2.resize(img, (50,50))
>>> H,W,_ = img.shape
>>> x =np.array([-1, 0, 1])
>>> M1 = cv2.getRotationMatrix2D((W/2,H/2),90,1)
>>> M2 = cv2.getRotationMatrix2D((W,H),90,1)
```

Choose the best matches between OpenCV code and numpy code that perform the same functions. Briefly discuss the description/objective of the corresponding codes. If there is no match, write NIL

Numpy:

(i) [img[:,:,2], img[:,:,1], img[:,:,0]]
(ii) scale = np.sum(np.exp(-np.power(x,2)/2))
np.exp(-np.power(x,2)/2)/scale
(iii) np.transpose(img, [1, 0, 2])[::-1]
(iv) np.pad(img, ((1,1), (1,1), (0, 0)),mode='constant', constant_values=0)

OpenCV:

(1) cv2.warpAffine(img, M1, (H,W))
(2) cv2.filter2D (img, -1, np.ones((5,5), np.float32)/25.0)
(3) cv2.merge(cv2.split(img))
(4) cv2.copyMakeBorder(img,1,1,1,1,cv2.BORDER_CONSTANT,value=(0,0,0))
(5) cv2.getGaussianKernel(5, 1)
(6) cv2.split(img)
(7) cv2.warpAffine(img, M2, (W,H))

Answer:

Numpy OpenCV Description
(i) NIL Get individual colour splits
(Since img is read using cv2.imread() the img is already in BGR ie img[:,:,0] is blue hence (i) should be [img[:,:,0], img[:,:,1], img[:,:,2]]
(ii) NIL Create gaussian kernel of size (3,1)
(iii) 1 Rotate image 90 degrees anticlockwise
(iv) 4 Add one row and column of zero pad around width and height image of shape

Answer:

Numpy	OpenCV	Description
(i)	NIL	Get individual colour splits *(Since img is read using cv2.imread() the img is already in BGR ie img[:,:,0] is blue hence (i) should be [img[:,:,0], img[:,:,1], img[:,:,2]]*
(ii)	NIL	Create gaussian kernel of size (3,1)
(iii)	1	Rotate image 90 degrees anticlockwise
(iv)	4	Add one row and column of zero padding around width and height image of shape

Calculating no. params in a CNN

$$W_c = K^2 \times C \times N$$
$$B_c = N$$
$$P_c = W_c + B_c$$

W_c = Number of weights of the Conv Layer.
B_c = Number of biases of the Conv Layer.
P_c = Number of parameters of the Conv Layer.
K = Size (width) of kernels used in the Conv Layer.
N = Number of kernels.
C = Number of channels of the input image.

There are no params in pooling layers
Calculating Params in FC layers
1. If FC is connected to Conv layer

$$W_{cf} = O^2 \times N \times F$$
$$B_{cf} = F$$
$$P_{cf} = W_{cf} + B_{cf}$$

W_{cf} = Number of weights of a FC Layer which is connected to a Conv Layer.
B_{cf} = Number of biases of a FC Layer which is connected to a Conv Layer.
O = Size (width) of the output image of the previous Conv Layer.
N = Number of kernels in the previous Conv Layer.
F = Number of neurons in the FC Layer.

2. If FC is connected to FC layer

$$W_{ff} = F_{-1} \times F$$
$$B_{ff} = F$$
$$P_{ff} = W_{ff} + B_{ff}$$

W_{ff} = Number of weights of a FC Layer which is connected to an FC Layer.
B_{ff} = Number of biases of a FC Layer which is connected to an FC Layer.
P_{ff} = Number of parameters of a FC Layer which is connected to an FC Layer.
F = Number of neurons in the FC Layer.
F_{-1} = Number of neurons in the previous FC Layer.