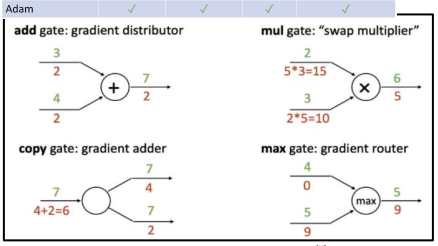
KNN - O(1) train, O(n) test
KNN Universal Approximation: with enough training samples, NN can represent any function.
Curse of Dimensionality: for uniform coverage of space, number of training points grows exponentially with dimension
Linear Classifier:
f(x, W) = Wx + b - cannot capture multiple modes of data
-use an average image template per category
-close form soln: $W^* = (X^T X)^{-1} X^T y$
Cross Validation: (find best hypoparam) split data into folds, try each as validation set and average, not usually used in DL

Cross-Entropy vs SVM Loss

$$L_i = -\log\left(\frac{\exp(s_{y_i})}{\sum_j \exp(s_j)}\right) \qquad L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Gradient Descent Shit:
W momentum: velocity: running mean of gradients, rho: friction
AdaGrad: element-wise scaling w historical sum of squares, damp steep direction, accelerate flat direction
RMSProp: weighted adagrad
Adam: RMSProp + Momentum + Bias correction

Algorithm	Tracks first moments (Momentum)	Tracks second moments (Adaptive learning rates)	Weighted second moments	Bias correction for moment estimates
SGD	x	x	x	x
SGD+Momentum	✓	x	x	x
AdaGrad	x	✓	x	x
RMSProp	x	✓	✓	x
Adam	✓	✓	✓	✓



Input volume: 3 x 32 x 32
10 5x5 filters with stride 1, pad 2

Output volume size: 10 x 32 x 32
Number of learnable parameters: **760**
Parameters per filter: 3*5*5 + 1 (for bias) = **76**
10 filters, so total is 10 * 76 = **760**

10*32*32 = 10,240 outputs; each output is the inner product of two 3x5x5 tensors (75 elems); total = 75*10240 = **768K**

AlexNet Architecture

Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv (C_{out}=20, K=5, P=2, S=1)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool(K=2, S=2)	20 x 14 x 14	
Conv (C_{out}=50, K=5, P=2, S=1)	50 x 14 x 14	50 x 20 x 5 x 5
ReLU	50 x 14 x 14	
MaxPool(K=2, S=2)	50 x 7 x 7	
Flatten	2450	
Linear (2450 -> 500)	500	2450 x 500
ReLU	500	
Linear (500 -> 10)	10	500 x 10

Batch Normalization for ConvNets

Batch Normalization for **fully-connected** networks

$x : N \times D$
Normalize
$\mu, \sigma : 1 \times D$
$\gamma, \beta : 1 \times D$
$y = \frac{(x-\mu)}{\sigma}\gamma + \beta$

Batch Normalization for **convolutional** networks (Spatial Batchnorm, BatchNorm2D)

$x : N \times C \times H \times W$
Normalize
$\mu, \sigma : 1 \times C \times 1 \times 1$
$\gamma, \beta : 1 \times C \times 1 \times 1$
$y = \frac{(x-\mu)}{\sigma}\gamma + \beta$

Benefit: model easier to train; faster convergence; robust to initialization; act like regularization; 0 overhead at test time

Layer	C	Input size H / W	filters	kernel	stride	pad	C	Output size H / W	memory (KB)
conv1	3	227	64	11	4	2	64	56	784

Number of output elements = C * H' * W'
= 64*56*56 = 200,704

Bytes per element = 4 (for 32-bit floating point)

KB = (number of elements) * (bytes per elem) / 1024
= 200704 * 4 / 1024
= **784**

Weight shape = C_{out} x C_{in} x K x K
= 64 x 3 x 11 x 11
Bias shape = C_{out} = 64
Number of weights = 64*3*11*11 + 64
= **23.296**

Number of floating point operations (multiply+add)
= (number of output elements) * (ops per output elem)
= (C_{out} x H' x W') * (C_{in} x K x K)
= (64 * 56 * 56) * (3 * 11 * 11)
= 200,704 * 363
= **72,855,552**

In general:
Input: W
Filter: K
Padding: P
Stride: S
Output: (W – K + 2P) / S + 1

$$(n_h - k_h + 2p_h + s_h)/s_h$$

Activation Functions

Sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$

tanh $\tanh(x)$

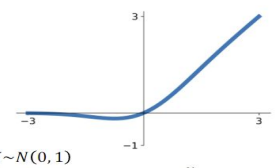
ReLU $\max(0, x)$

Leaky ReLU $\max(0.1x, x)$

Maxout $\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU $\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$

Activation Functions: Gaussian Error Linear Unit (GELU)



$X \sim N(0,1)$
$gelu(x) = xP(X \leq x) = \frac{x}{2}(1 + \text{erf}(x/\sqrt{2}))$
$\approx x\sigma(1.702x)$

- **Idea:** Multiply input by 0 or 1 at random; large values more likely to be multiplied by 1, small values more likely to be multiplied by 0 (data-dependent dropout)
- Take expectation over randomness
- Very common in Transformers (BERT, GPT, GPT-2, GPT-3)

Hendrycks and Gimpel, Gaussian Error Linear Units (GELUs), 2016

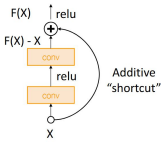
Matrix Mult BackProp

dL/dx = (dL/dy) W^T
[N x D] [N x M] [M x D]

dL/dw = x^T (dL/dy)
[D x M] [D x N] [N x M]

Models / Features	LeNet	AlexNet	VGG	G-Net	ResNet	ViT	swin-trans
	Classic CNN + avg pool + sigmoid	Deeper conv + max pool + relu	More and smaller kernels	Inception: branches convs + global avg. pool + extra classifiers	Residual/ skip connects + batch norm	No conv, patch image -> transformer encoder	Sequential ViTs, each block merges patches - window attention

Residual Block: learn identity functions, combat large model underfit



Kullback–Leibler divergence
$D_{KL}(P \,||\, Q) = \sum_y P(y) \log \frac{P(y)}{Q(y)}$

Choosing Hyperparameters:
Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer
Step 6: Look at loss curves
Step 7: GOTO step 5

Semantic Segm. Gives per pixel labels but merges instances of things and background items

Momentum - accelerate SGD

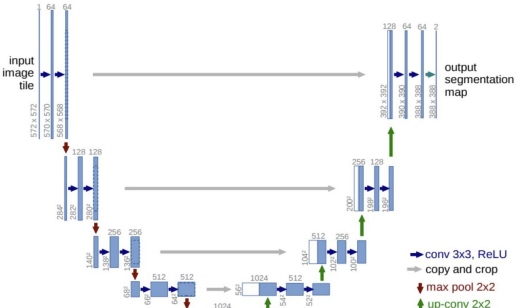
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

Kaiming Init. use Xavier and use ReLU sqrt(2/Din)

```
dims = [4096] * 7                    "Xavier" initialization:
hs = []                              std = 1/sqrt(Din)
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Regularization:
Norm - add L2 of params to loss
Dropout - randomly drop neurons/weights during training
Data Aug. - transform inputs
Cutout - Remove section of inputs
Mixup - blend inputs
Early Stopping - Halt training at train/val loss divergence
Ensemble - avg multiple models
Learning Rate Decay - linear, sin., inv. sqrt. decay to step size
Transfer Learning + Finetune

Latent Diffusion: DDPM/DDIM is still iterative in training, trying to generate large images is expensive so we use a VAE. use the encoder to encode input image to lower dim. latent space, THEN apply forward noise. Then use VAE decoder to upscale noisy image. Train backward.



Layer	C	Input size H / W	filters	kernel	stride	pad	C	Output size H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0	64	27	182	0	0
conv2	64	27	192	5	1	2	192	27	547	307	224
pool2	192	27		3	2	0	192	13	127	0	0
conv3	192	13	384	3	1	1	384	13	254	664	112
conv4	384	13	256	3	1	1	256	13	169	885	145
conv5	256	13	256	3	1	1	256	13	169	590	100
pool5	256	13		3	2	0	256	6	36	0	0
flatten	256	6					9216		36	0	0
fc6	9216		4096				4096		16	37,749	38

FC params = C_{in} * C_{out} + C_{out}
= 9216 * 4096 + 4096
= 37,725,832

FC flops = C_{in} * C_{out}
= 9216 * 4096
= 37,748,736

Diffusion (DDIM): DDPM improvement to remove iterative noise with reparametrization. Same loss.

$$q_\sigma(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0) := q_\sigma(\boldsymbol{x}_T|\boldsymbol{x}_0) \prod_{t=2}^{T} q_\sigma(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0)$$

$$q_\sigma(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) = \mathcal{N}\left(\sqrt{\alpha_{t-1}}\boldsymbol{x}_0 + \sqrt{1-\alpha_{t-1}-\sigma_t^2}\cdot\frac{\boldsymbol{x}_t - \sqrt{\alpha_t}\boldsymbol{x}_0}{\sqrt{1-\alpha_t}}, \sigma_t^2 \boldsymbol{I}\right)$$

Diffusion (DDPM): Iteratively add gaussian noise -> true gaussian distribution (forward) then iteratively denoise (backward). Noise is added w/ hyperparam alpha which can be found for any time step with reparametrization -> tends noise to 0 mean, unique variance. Train model to predict noise (supervised), VLB (KL-Div) loss, simplified - dist b/w real and pred noise per timestep:

Forward process

$$q_\sigma(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}, \boldsymbol{x}_0) = \frac{q_\sigma(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) q_\sigma(\boldsymbol{x}_t|\boldsymbol{x}_0)}{q_\sigma(\boldsymbol{x}_{t-1}|\boldsymbol{x}_0)}$$

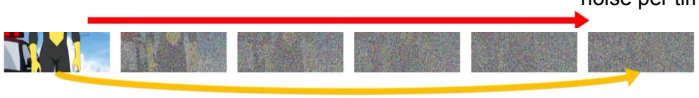
Backward process

$$p_\theta^{(t)}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) = \begin{cases} \mathcal{N}(f_\theta^{(1)}(\boldsymbol{x}_1), \sigma_1^2 \boldsymbol{I}) & \text{if } t = 1 \\ q_\sigma(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, f_\theta^{(t)}(\boldsymbol{x}_t)) & \text{otherwise,} \end{cases}$$

$$f_\theta^{(t)}(\boldsymbol{x}_t) := (\boldsymbol{x}_t - \sqrt{1-\alpha_t}\cdot\epsilon_\theta^{(t)}(\boldsymbol{x}_t))/\sqrt{\alpha_t}$$

$$\boldsymbol{x}_{t-1} = \sqrt{\alpha_{t-1}}\underbrace{\left(\frac{\boldsymbol{x}_t - \sqrt{1-\alpha_t}\epsilon_\theta^{(t)}(\boldsymbol{x}_t)}{\sqrt{\alpha_t}}\right)}_{\text{"predicted } \boldsymbol{x}_0\text{"}} + \underbrace{\sqrt{1-\alpha_{t-1}-\sigma_t^2}\cdot\epsilon_\theta^{(t)}(\boldsymbol{x}_t)}_{\text{"direction pointing to } \boldsymbol{x}_t\text{"}} + \underbrace{\sigma_t\epsilon_t}_{\text{random noise}}$$

$$L_{\text{simple}} = E_{t,x_0,\epsilon}\left[||\epsilon - \epsilon_\theta(x_t, t)||^2\right]$$



$$q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0) := \prod_{t=1}^{T} q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) \qquad q(\boldsymbol{x}_t|\boldsymbol{x}_0) := \mathcal{N}(\boldsymbol{x}_t; \sqrt{\alpha_t}\boldsymbol{x}_0, (1-\alpha_t)\boldsymbol{I})$$

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) := \mathcal{N}\left(\sqrt{\frac{\alpha_t}{\alpha_{t-1}}}\boldsymbol{x}_{t-1}, \left(1 - \frac{\alpha_t}{\alpha_{t-1}}\right)\boldsymbol{I}\right) \qquad \boldsymbol{x}_t = \sqrt{\alpha_t}\boldsymbol{x}_0 + \sqrt{1-\alpha_t}\epsilon, \quad \text{where} \quad \epsilon \sim \mathcal{N}(0, \boldsymbol{I})$$

$$p_\theta(\mathbf{x}_{0:T}) := p(\mathbf{x}_T)\prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \qquad p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) := \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t))$$

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \ldots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$\nabla_\theta \left\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)\right\|^2$
6: **until** converged

Algorithm 2 Sampling

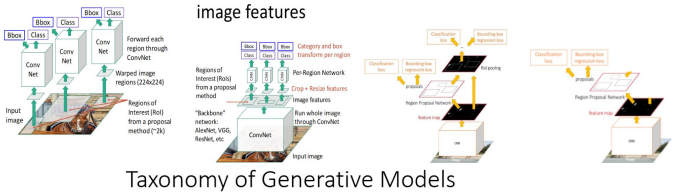
1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \ldots, 1$ **do**
3: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

"Slow" R-CNN: Run CNN independently for each region

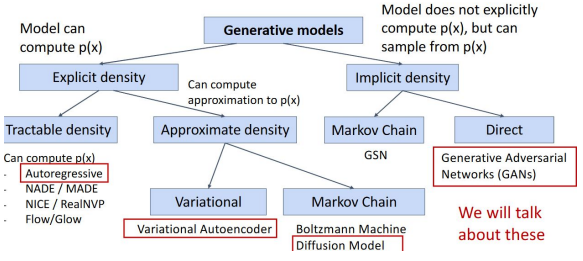
Fast R-CNN: Apply differentiable cropping to shared image features

Faster R-CNN: Compute proposals with CNN

Single-Stage: Fully convolutional detector

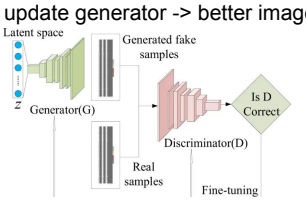


Taxonomy of Generative Models



AutoRegressive Models: model with LSTM. generate pixel based on hidden state and pixels to left top region. Pros: explicit p(x); movies good eval metric; good samples. Cons: sequential generation=slow

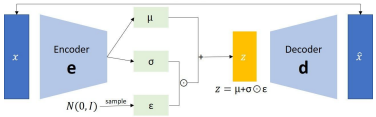
GAN: generator + discriminator, generates image, discriminator discerns if generated or not, use to update generator -> better images



$$\nabla_{\theta_d}\frac{1}{m}\sum_{i=1}^{m}\left[\log D\left(\boldsymbol{x}^{(i)}\right)+\log\left(1-D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right]$$

GAN loss: 1st term discrim. discerns real image, 2nd term discrim. discerns generated image (log-likelihood)

VAE: reconstructs input image by compressing, then sample from compressed (latent) space (normal random sample) but for grad to flow from random sample -> abstract randomness to eps. Can be used for segmentation, inpainting, generation



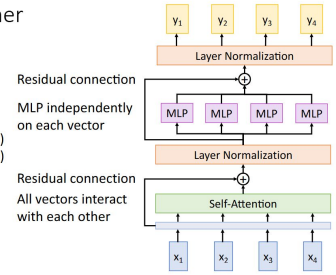
$$\sum_{i=1}^{N}\mathbb{E}_{q_\theta(Z)}\log p(x_i|Z)-D_{KL}(q_\theta(Z)\parallel p(Z))$$

VAE loss: 1st term loss for expected output image, 2nd term KL-divergence: regularizes latent toward normal distribution.
Modification: disentangle VAE by mult. coeff to 2nd term -> extracts latent features of output image, allows generating diff "styles"

The Transformer

Recall **Layer Normalization**:
Given $h_1, \dots, h_N$ (Shape: D)
scale: γ (Shape: D)
shift: β (Shape: D)
$\mu_i = (\sum_j h_{i,j})/D$ (scalar)
$\sigma_i = (\sum_j (h_{i,j} - \mu_i)^2/D)^{1/2}$ (scalar)
$z_i = (h_i - \mu_i) / \sigma_i$
$y_i = \gamma * z_i + \beta$

Ba et al, 2016

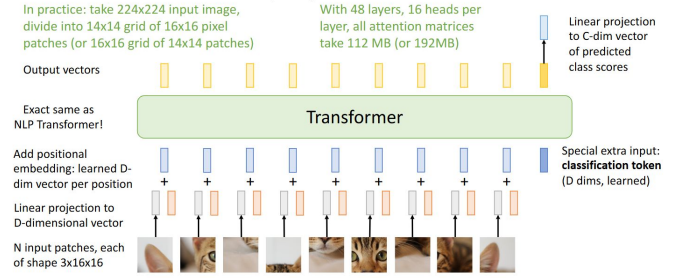


Vaswani et al, "Attention is all you need", NeurIPS 2017

Vision Transformer (ViT)

In practice: take 224x224 input image, divide into 14x14 grid of 16x16 pixel patches (or 16x16 grid of 14x14 patches)

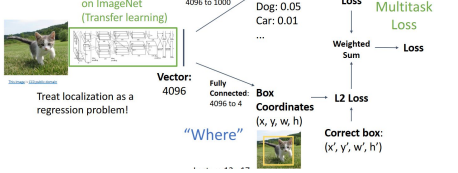
With 48 layers, 16 heads per layer, all attention matrices take 112 MB (or 192MB)



Dosovitskiy et al, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Cat image is free for commercial use under a Pixabay license

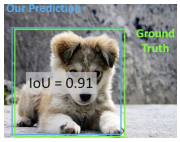
Detecting a single object



Total possible boxes:

$$\sum_{h=1}^{H}\sum_{w=1}^{W}(W-w+1)(H-h+1)$$

$$=\frac{H(H+1)}{2}\frac{W(W+1)}{2}$$



$$IoU=\frac{y\cap\hat{y}}{y\cup\hat{y}}$$

$$Dice=2\cdot IoU$$

IoU > 0.5 is "decent",
IoU > 0.7 is "pretty good",
IoU > 0.9 is "almost perfect"

Question: How many possible boxes are there in an image of size H x W?

Consider a box of size h x w:
Possible x positions: W – w + 1
Possible y positions: H – h + 1
Possible positions:
(W – w + 1) * (H – h + 1)

Total possible boxes:

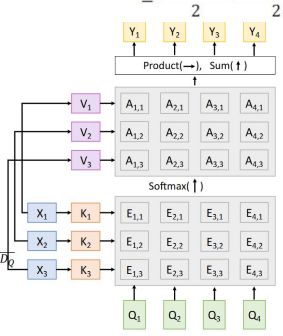
$$\sum_{h=1}^{H}\sum_{w=1}^{W}(W-w+1)(H-h+1)$$

$$=\frac{H(H+1)}{2}\frac{W(W+1)}{2}$$

Attention Layer

Inputs:
Query vectors: Q (Shape: N_Q x D_Q)
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)

Computation:
Key vectors: K = XW_K (Shape: N_X x D_Q)
Value Vectors: V = XW_V (Shape: N_X x D_V)
Similarities: E = $QK^T/\sqrt{D_Q}$ (Shape: N_Q x N_X) $E_{i,j} = (Q_i \cdot K_j)/\sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) (Shape: N_Q x N_X)
Output vectors: Y = AV (Shape: N_Q x D_V) $Y_i = \sum_j A_{i,j} V_j$



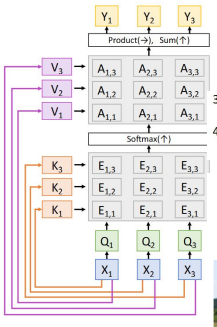
Self-Attention Layer

One **query** per **input vector**

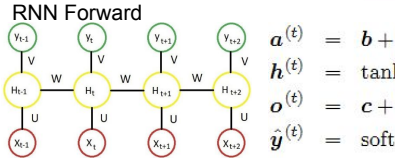
Inputs:
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)
Query matrix: W_Q (Shape: D_X x D_Q)

Computation:
Query vectors: Q = XW_Q
Key vectors: K = XW_K (Shape: N_X x D_Q)
Value Vectors: V = XW_V (Shape: N_X x D_V)
Similarities: E = $QK^T/\sqrt{D_Q}$ (Shape: N_X x N_X) $E_{i,j} = (Q_i \cdot K_j)/\sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) (Shape: N_X x N_X)
Output vectors: Y = AV (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$

$$\text{softmax}\left(\frac{1}{\sqrt{d_k}}QK^T\right)$$



RNN Forward



$$\boldsymbol{a}^{(t)} = \boldsymbol{b}+\boldsymbol{W}\boldsymbol{h}^{(t-1)}+\boldsymbol{U}\boldsymbol{x}^{(t)}$$
$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{a}^{(t)})$$
$$\boldsymbol{o}^{(t)} = \boldsymbol{c}+\boldsymbol{V}\boldsymbol{h}^{(t)}$$
$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}(\boldsymbol{o}^{(t)})$$

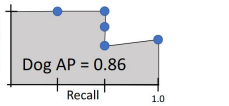
Recurrent Neural Network	1D Convolution	Self-Attention
Works on **Ordered Sequences**	Works on **Multidimensional Grids**	Works on **Sets of Vectors**
(+) Good at long sequences: After one RNN layer, h_t "sees" the whole sequence	(-) Bad at long sequences: Need to stack many conv layers for outputs to "see" the whole sequence	(-) Good at long sequences: after one self-attention layer, each output "sees" all inputs!
(-) Not parallelizable: need to compute hidden states sequentially	(+) Highly parallel: Each output can be computed in parallel	(+) Highly parallel: Each output can be computed in parallel
		(-) Very memory intensive

Transformers in CV:

ViT - encoder class. on image patches
Swin - Hierarchical ViT, merging patches
DETR - bounding box obj. det. using enc-dec arch.
DiT - Diffusion with Transformers (e.g., OpenAI Sora Video)

Evaluating Object Detectors: Mean Average Precision (mAP)

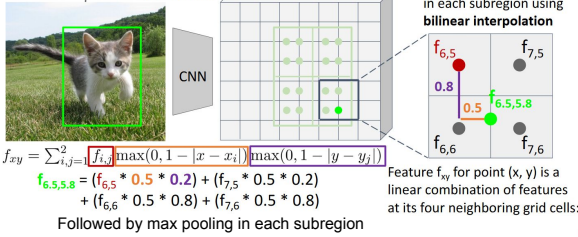
1. Run object detector on all test images (with NMS)
2. For each category, compute Average Precision (AP) = area under Precision vs Recall Curve
   1. For each detection (highest score to lowest score)
      1. If it matches some GT box with IoU > 0.5, mark it as positive and eliminate the GT
      2. Otherwise mark it as negative
      3. Plot a point on PR Curve
   2. Average Precision (AP) = area under PR curve
3. Mean Average Precision (mAP) = average of AP for each category
4. For "COCO mAP": Compute mAP@thresh for **each IoU threshold** (0.5, 0.55, 0.6, ..., 0.95) and take average



Cropping Features: RoI Align

RoI Pool=snapping + max pool 2x2

Divide into equal-sized subregions (may not be aligned to grid!)



$$f_{xy}=\sum_{i,j=1}^{2}f_{i,j}\max(0,1-|x-x_i|)\max(0,1-|y-y_j|)$$

$f_{6,5,5.8}$ = ($f_{6,5}$ * 0.5 * 0.2) + ($f_{7,5}$ * 0.5 * 0.2)
+ ($f_{6,6}$ * 0.5 * 0.8) + ($f_{7,6}$ * 0.5 * 0.8)

Feature f_{xy} for point (x, y) is a linear combination of features at its four neighboring grid cells:

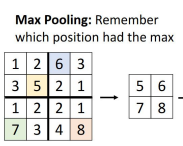
Followed by max pooling in each subregion

Fast R-CNN or Slow R-CNN: shared image features that are calculated by a ConvNet. Regions of Interest form a proposal method. Crop+resize features and then per region CNN. Slow had Regions of Interest from a proposal method, warp image, forward region through a ConvNet and then have a bounding box and class. **Faster**: Insert Region Proposal Network to predict proposals from features. **Region Proposal Network**: Run backbone CNN to get features, use anchor box. K boxes. Pass through ConvNet -> normal fastRCNN. Single-Stage: Fully convolutional detector. Featuremap -> RPN->proposals. **Faster:** stage 1(per image): bkbone net->reg pro net, stage2(per region): crop->predict object->bbox offset

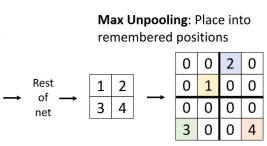
Style Transfer
GANs - Pix2Pix, CycleGAN, StyleGAN
Diffusion - Ctrl+X, ControlNet
Spatial Generation
NERFs, Gaussian Splatting - generate 3D space from image

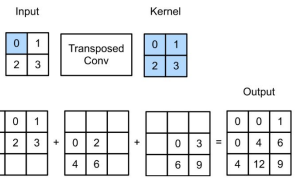
Semantic Seg
Downsampling: pooling, strided conv.
Upsampling:
Simple corner place
Nearest neighb.
Copy
Bilinear or Cubic interpol. (2 or 3 neighbor distances)

Max Pooling: Remember which position had the max



Max Unpooling: Place into remembered positions





Instance segmentation = Mask-RCNN

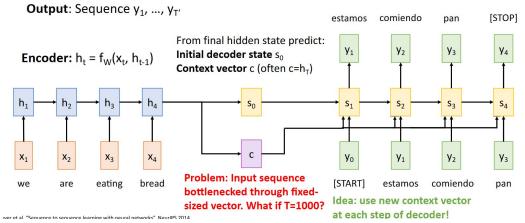
Transformer

Consider a transformer block (with one self-attention layer and one MLP layer) that processes N tokens of dimension D. How many parameters (excluding biases and layer norm) are in this block?

Self-attention layer has $3D^2$ parameters and MLP layer has D^2 parameters, so there are $4D^2$ total parameters.

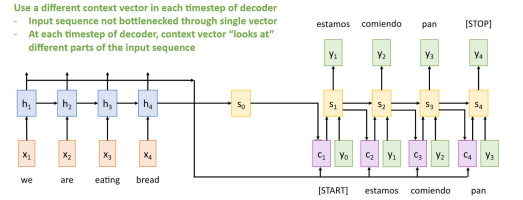
Sequence-to-Sequence with RNNs

Input: Sequence $x_1, \dots x_T$
Output: Sequence $y_1, \dots, y_{T'}$
Encoder: $h_t = f_W(x_t, h_{t-1})$
Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$



Sequence-to-Sequence with RNNs and Attention

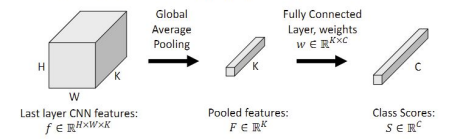
Use a different context vector in each timestep of decoder
- Input sequence not bottlenecked through single vector
- At each timestep of decoder, context vector "looks at" different parts of the input sequence



Visualization Methods

Saliency Maps - unsupervised segmentation
Class Activation Maps (CAM):

Class Activation Mapping (CAM)



$$F_k=\frac{1}{HW}\sum_{h,w}f_{h,w,k}\qquad S_c=\sum_k w_{k,c}F_k=\frac{1}{HW}\sum_k w_{k,c}\sum_{h,w}f_{h,w,k}=\frac{1}{HW}\sum_{h,w}\sum_k w_{k,c}f_{h,w,k}$$

Class Activation Maps: $M\in\mathbb{R}^{C,H,W}$
$M_{c,h,w}=\sum_k w_{k,c}f_{h,w,k}$

Grad-CAM - grad. adjusted CAM:

1. Pick any layer, with activations $A\in\mathbb{R}^{H\times W\times K}$
2. Compute gradient of class score S_c with respect to A:
$$\frac{\partial S_c}{\partial A}\in\mathbb{R}^{H\times W\times K}$$
3. Global Average Pool the gradients to get weights $\alpha\in\mathbb{R}^K$:
$$\alpha_k=\frac{1}{HW}\sum_{h,w}\frac{\partial S_c}{\partial A_{h,w,k}}$$
4. Compute activation map $M^c\in\mathbb{R}^{H,W}$:
$$M^c_{h,w}=ReLU\left(\sum_k\alpha_k A_{h,w,k}\right)$$

Grad. Ascent: view topological gradient of img
Feat. Inversion - use Adversarial Attack
Feature Pyramid Network - Segmentation

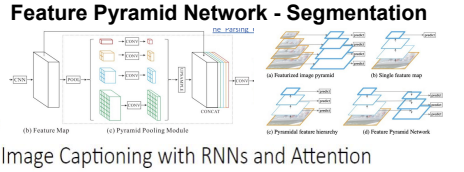


Image Captioning with RNNs and Attention

$e_{t,i,j} = f_{att}(s_{t-1}, h_{i,j})$
$a_{t,:,:} = \text{softmax}(e_{t,:,:})$
$c_t = \sum_{i,j} a_{t,i,j} h_{i,j}$

Each timestep of decoder uses a different context vector that looks at different parts of the input image