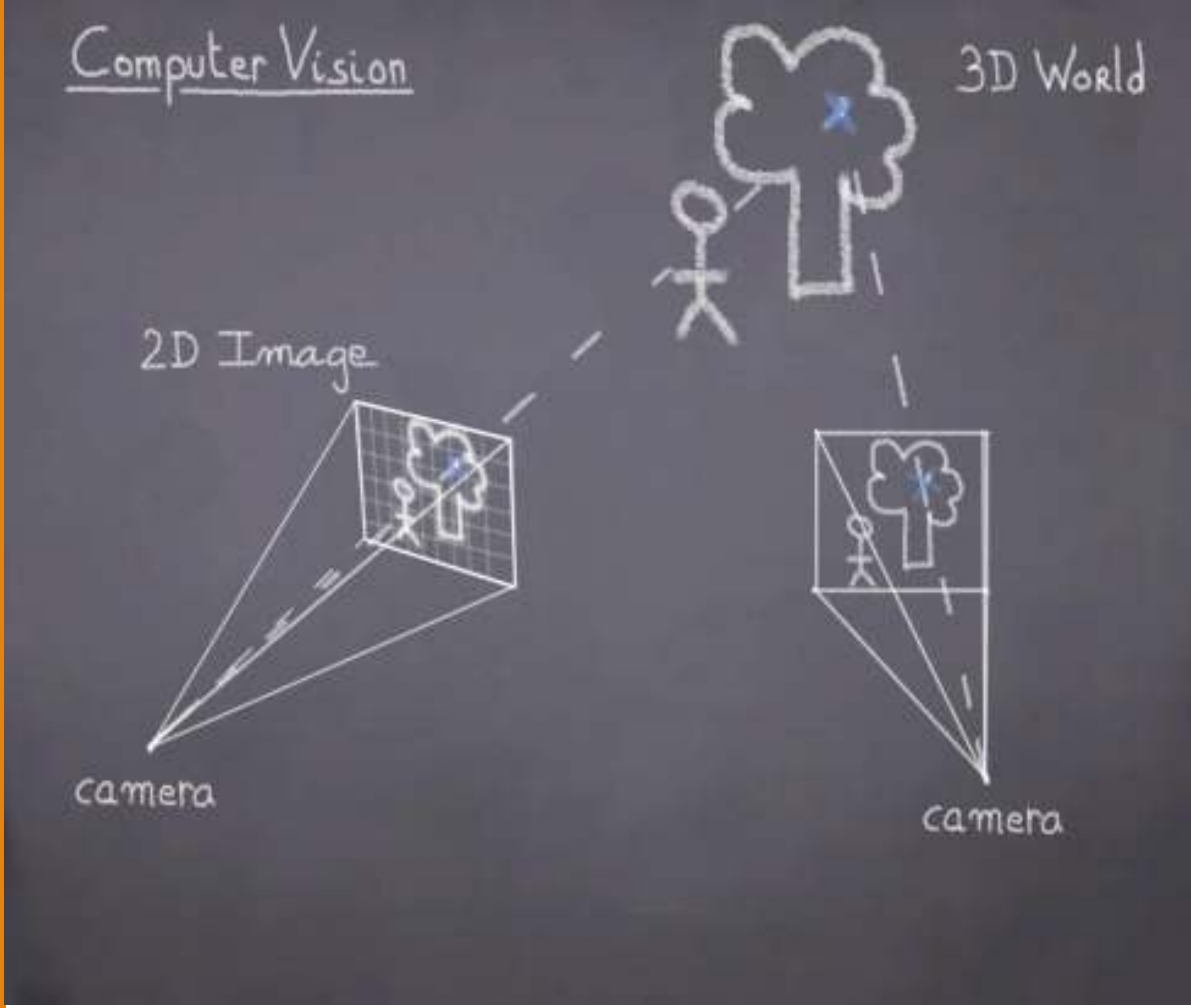
Deep Learning Intro

Max Welling

Special thanks to *Efstratios Gavves*
whose slides I am using and who has helped with the practicals

Also thanks to *Cees Snoek, Laurens van der Maaten & Arnold Smeulders* who all contributed to the slides

Deep Learning in Computer Vision



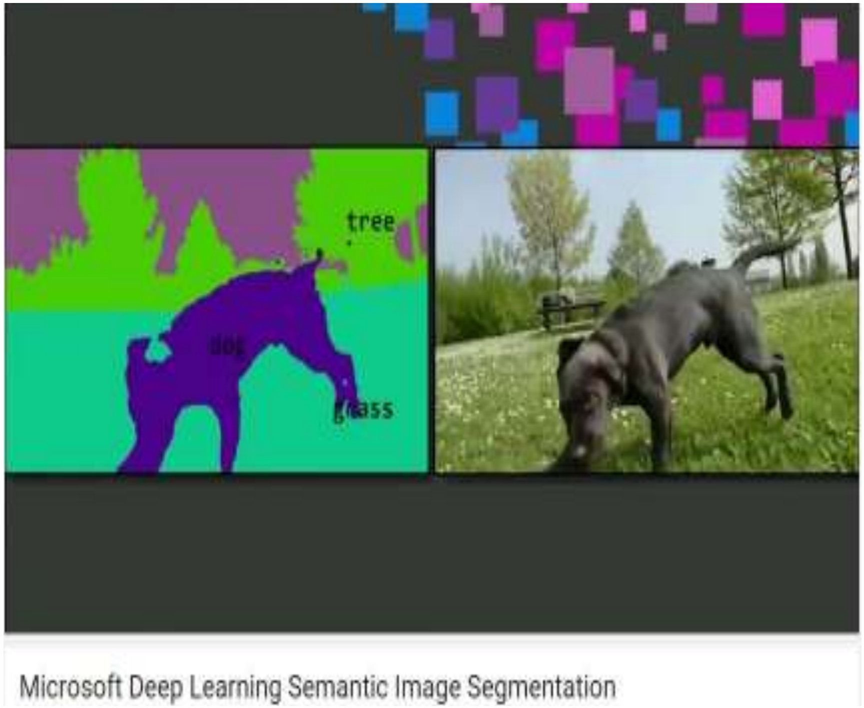
Object and activity recognition

[Click to go to the video in Youtube]

Large-scale Video Classification with Convolutional Neural Networks, CVPR 2014

Object detection, segmentation, pose estimation

Click to go to the video in Youtube



Deep Learning in Robotics

UVA DEEP LEARNING COURSE
EFSTRATIOS GAVVES
INTRODUCTION TO DEEP LEARNING AND NEURAL NETWORKS - 5

Self-driving cars

[Click to go to the video in Youtube]

Self Driving Cars HD

Drones and robots

[Click to go to the video in Youtube]

Deep Learning in NLP and Speech

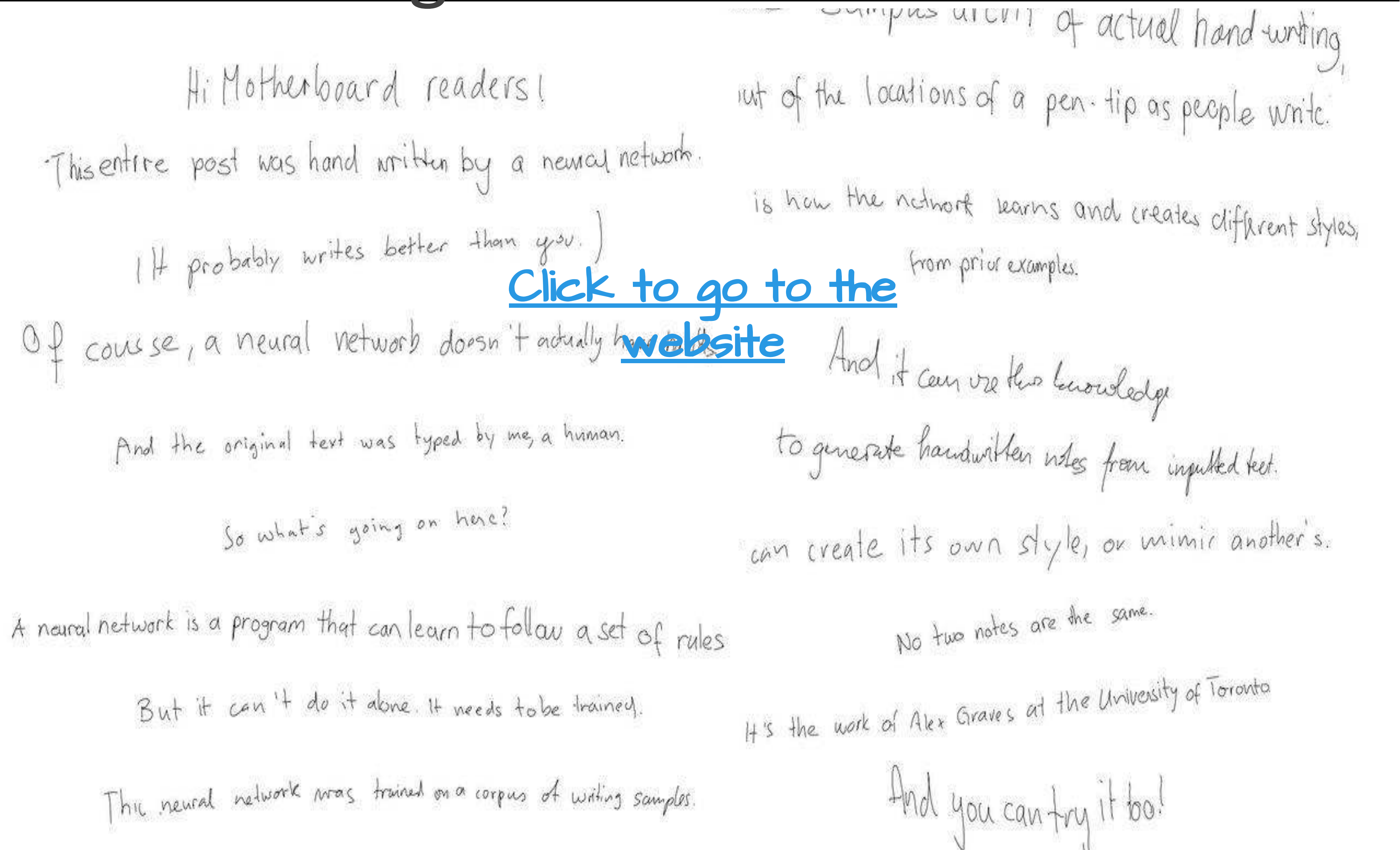
Speech recognition and Machine translation

Deep Learning in the arts

Imitating famous painters

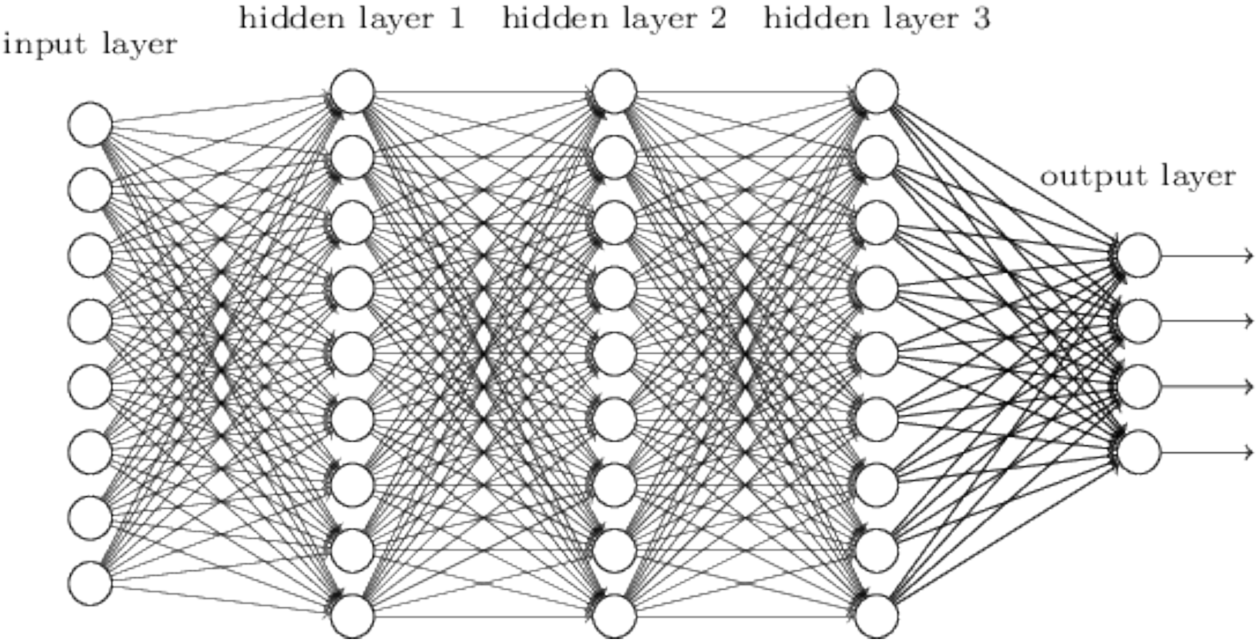
Gatys et al

Handwriting

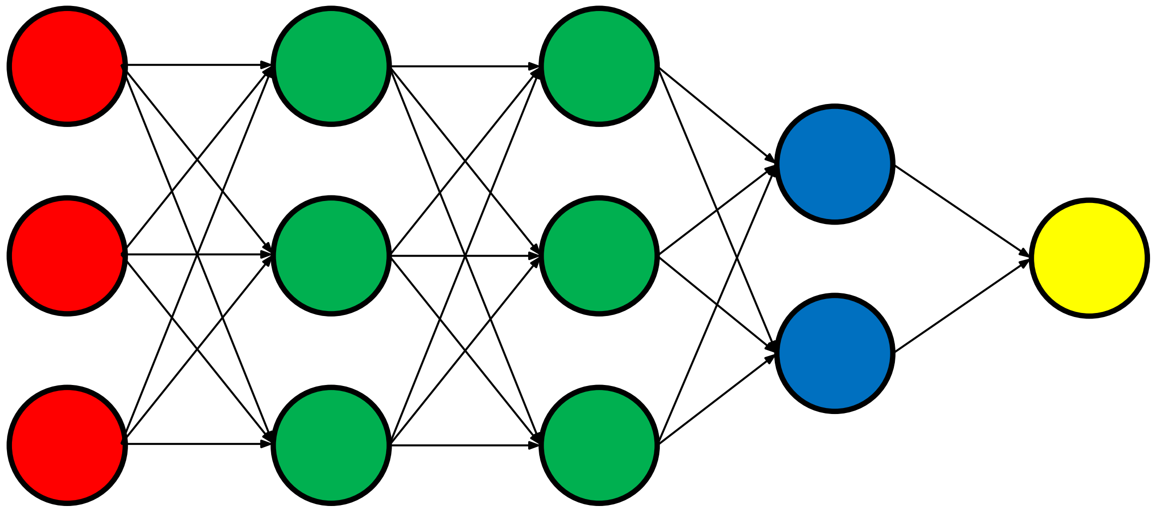


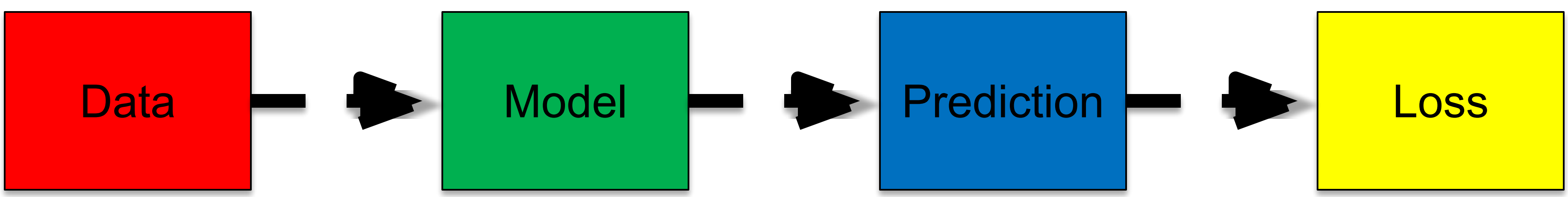
[Click to go to the website]

Deep Learning: The *What* and *Why*



A Neural Network perspective





5 convolutional layers
2 fully connected layers
----
100 convolutional layers
50 batch normalization layers

Softmax

Sigmoid

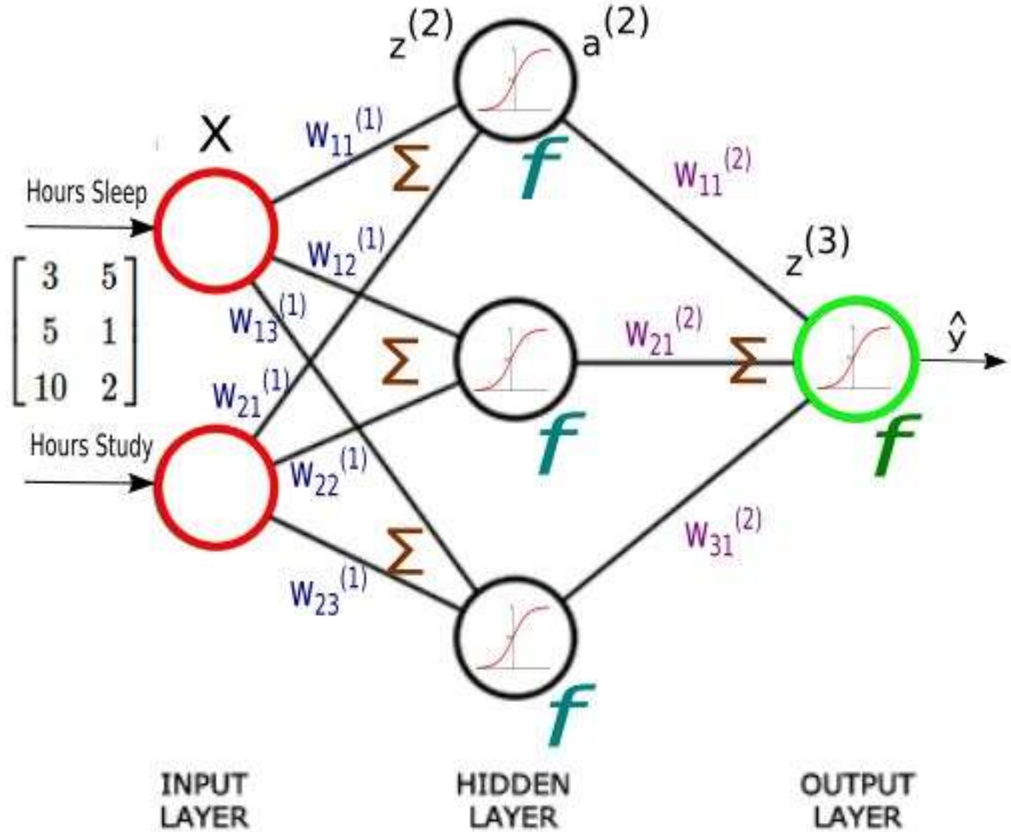
Linear etc.

Cross entropy

Euclidean

Contrastive

The Feedforward NN



$$a_{in}^{\ell+1} = h\left(\sum_j W_{ij}^{\ell} a_{jn}^{\ell} + b_i^{\ell}\right)$$

Module/Layer types?

Linear: $x_{i+1} = W \cdot x_i$ [parameteric → learnable]

Convolutional: $x_{i+1} = W * x_i$ [parameteric → learnable]

Nonlinearity: $x_{i+1} = h(x_0)$ [non-parameteric → defined]

Pooling: $x_1 = \text{downsample}\ (x_0)$ [non-parameteric → defined]

Normalization, *e.g.*: $x_1 = \ell_2(x_0)$ [non-parameteric → defined]

Regularization, *e.g.*: $x_1 = \text{dropout}(x_0)$ [non-parameteric → defined]

Practically, any 1st order [almost everywhere] differentiable function can be a module

Nonlinearities

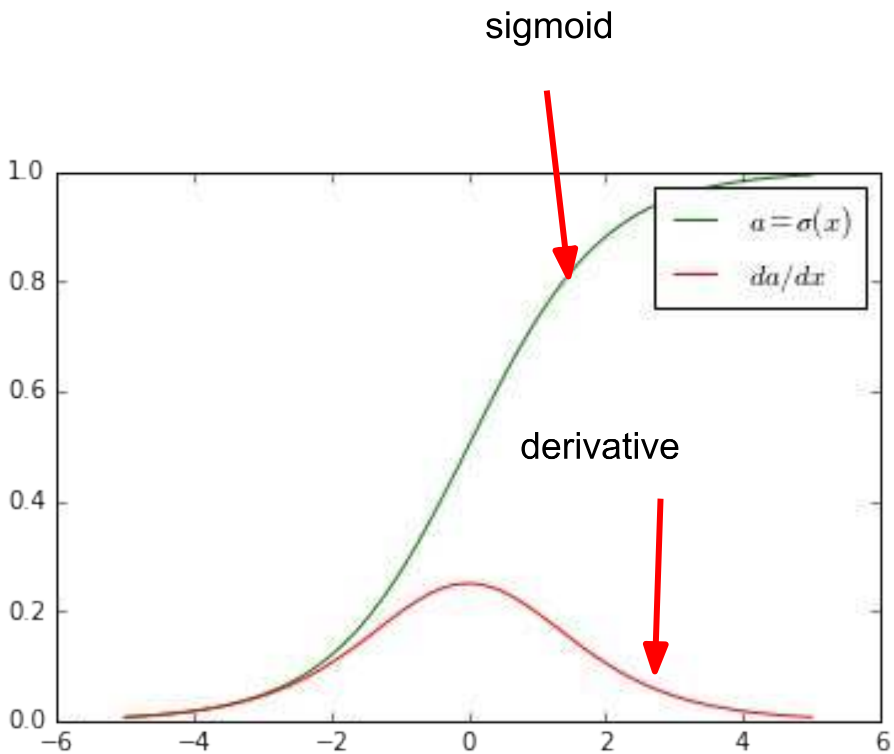
If we would only have N linear layers, we could replace them all with a single layer

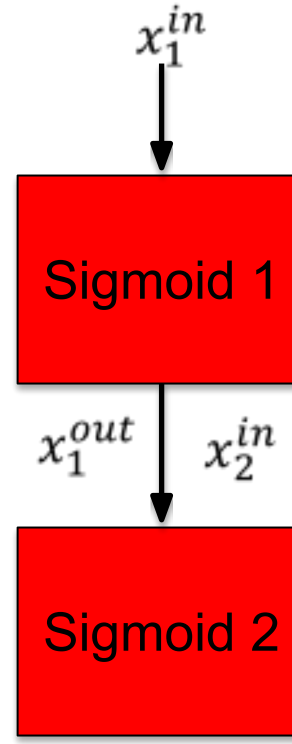
$$W_1 \cdot W_2 \cdot \ldots \cdot W_N = W$$

Nonlinearities allow for deeper networks

Any nonlinear function can work although some are more preferable than others

Sigmoid

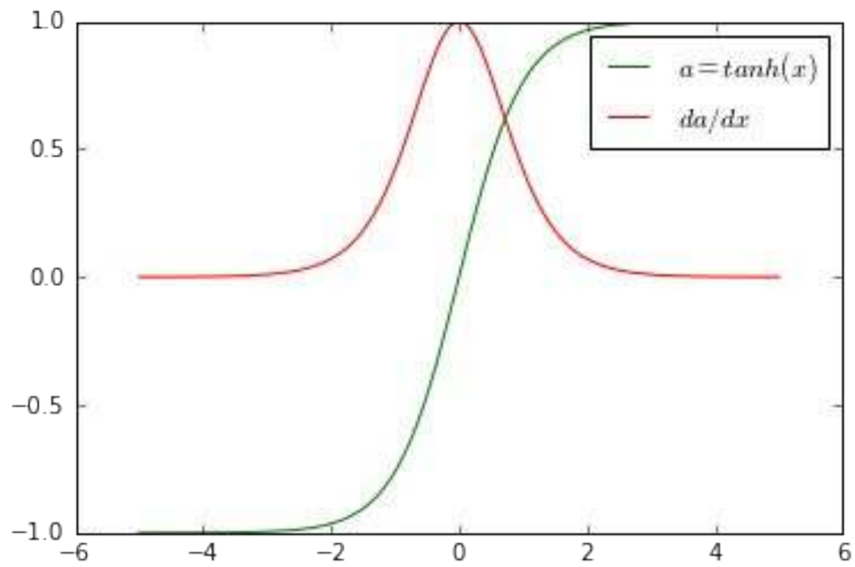




$$h(x) = \frac{1}{1+e^{-x}}$$

Tanh

$$h(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

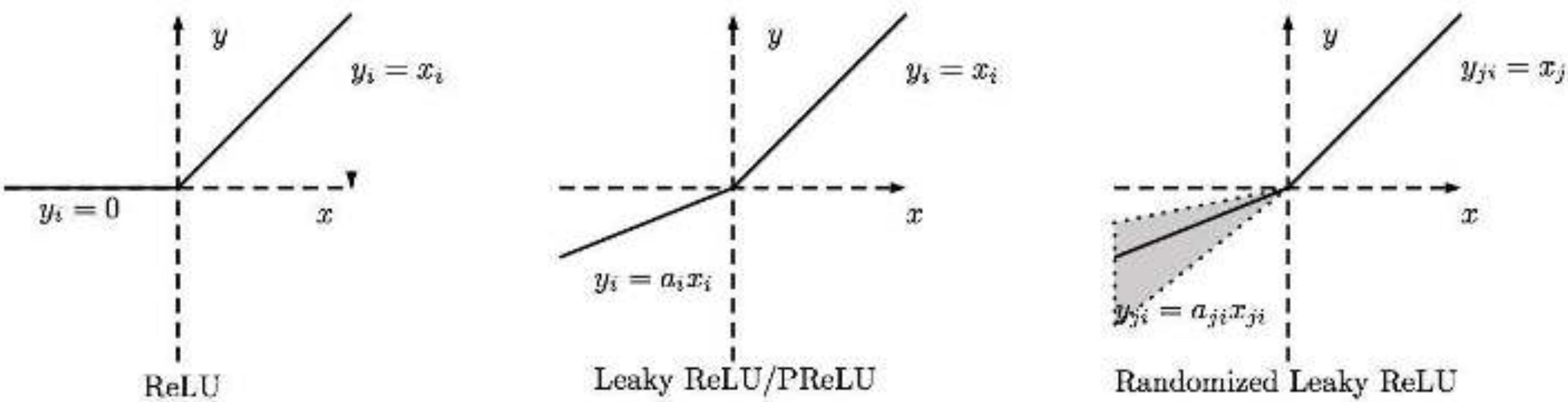


ReLU

Activation function $a = h(x) = \max(0, x)$

- Very popular in computer vision and speech recognition

Gradient wrt the input $\frac{\partial a}{\partial x} = \begin{cases} 0, if\ x \leq 0 \\ 1, if x > 0 \end{cases}$



ReLU

Much faster computations, gradients

- No vanishing/exploding gradients
- People claim biological plausibility :/

Sparse activations

No saturation

Non-symmetric

Non-differentiable at 0

A large gradient during training can cause a neuron to “die”. Higher learning rates mitigate the problem

Softmax

Activation function $a^{(k)} = softmax(x^{(k)}) = \frac{e^{x^{(k)}}}{\sum_j e^{x^{(j)}}}$

- Outputs probability distribution, $\sum_{k=1}^{K} a^{(k)} = 1$ for K classes
- Typically used as prediction layer

Because $e^{a+b} = e^a e^b$, we usually compute

$$a^{(k)} = \frac{e^{x^{(k)}-\mu}}{\sum_j e^{x^{(j)}-\mu}}, \mu = \max_k x^{(k)} \text{ because}$$

$$\frac{e^{x^{(k)}-\mu}}{\sum_j e^{x^{(j)}-\mu}} = \frac{e^{\mu} e^{x^{(k)}}}{e^{\mu} \sum_j e^{x^{(j)}}} = \frac{e^{x^{(k)}}}{\sum_j e^{x^{(j)}}}$$

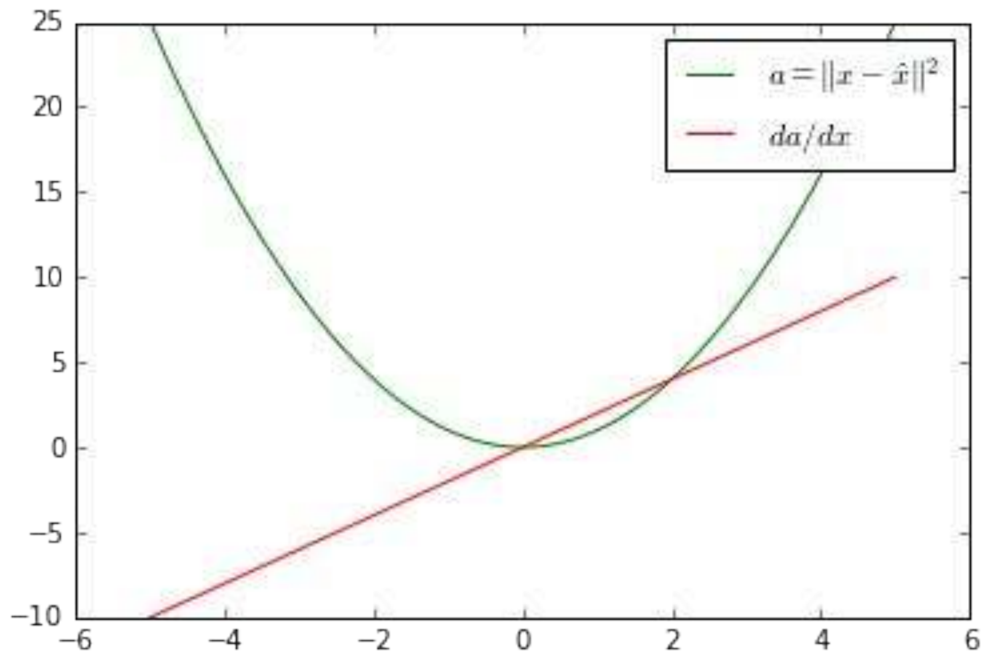
Avoid exponentianting large numbers → better stability

Euclidean Loss

Activation function $a(x) = 0.5\ \|y - x\|^2$

- ❑ Mostly used to measure the loss in **regression** tasks

Gradient with respect to the input $\frac{\partial a}{\partial x} = x - y$



Cross-entropy loss

Activation function $a(x) = -\sum_{k=1}^{K} y^{(k)} \log x^{(k)}, \quad y^{(k)} = \{0, 1\}$

Gradient with respect to the input $\frac{\partial a}{\partial x^{(k)}} = -\frac{1}{x^{(k)}}$

The cross-entropy is the most popular **classification loss** for classifiers that output probabilities (not SVM)

Cross-entropy loss couples well softmax/sigmoid module

- Often the modules are combined and joint gradients are computed

Generalization of logistic regression for more than 2 outputs

Training Neural Networks

1. The Neural Network

$$a_L(x;\theta_{1,\ldots,L}) = h_L\,(h_{L-1}(\ldots h_1(x,\theta_1),\theta_{L-1}),\theta_L)$$

2. Learning by minimizing empirical error

$$\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x;\theta_{1,\ldots,L}))$$

3. Optimizing with Gradient Descend based methods

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}$$

Intuitive Backpropagation

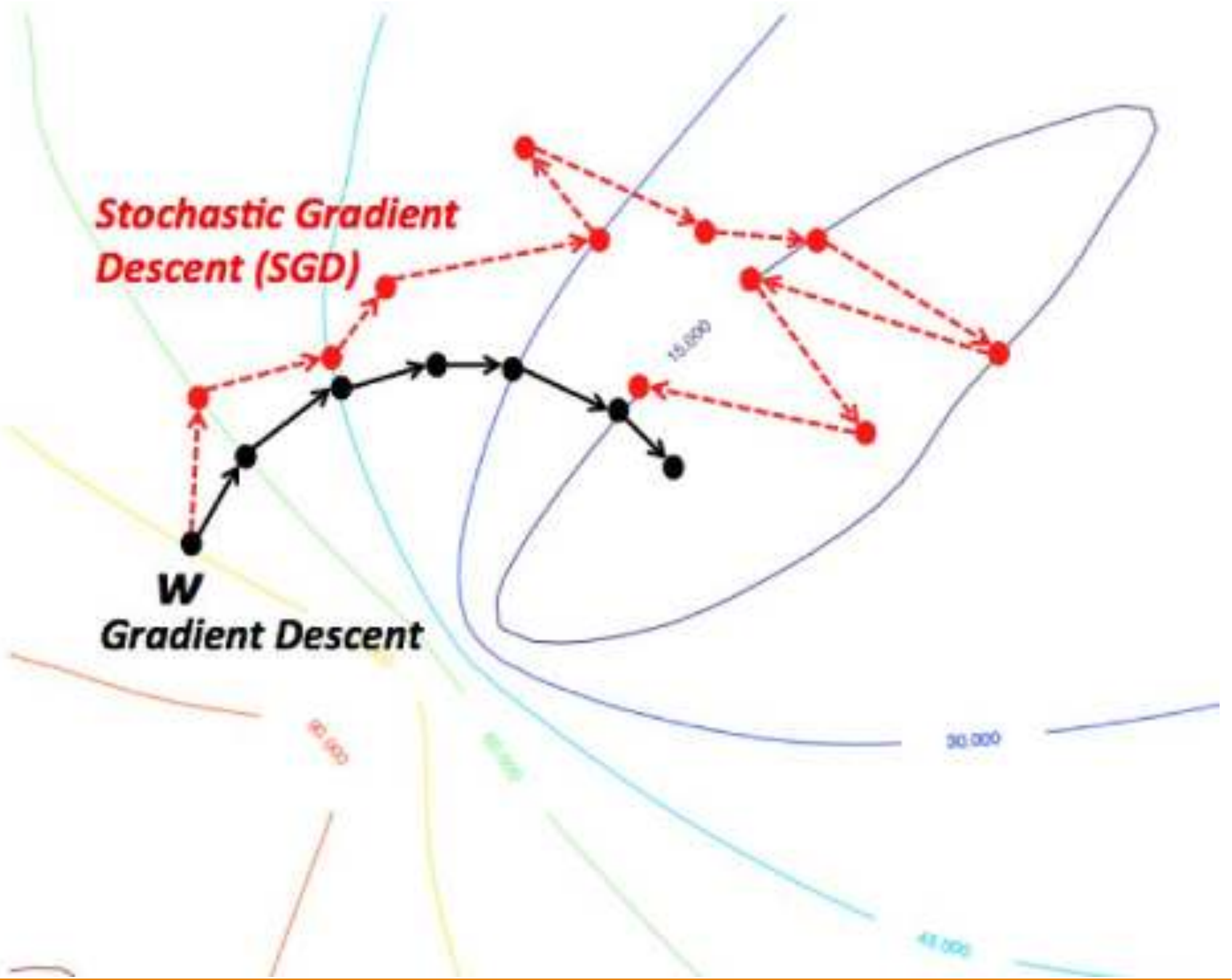
SGD

Sample small mini-batch of data-cases uniformly at random.
Compute average gradient based on this mini-batch.
Perform update based on gradient.

$$\text{Repeat until convergence } \{$$

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

$$\}$$



Backpropagation in practice

- Things are dead simple, just compute per module

$$\frac{\partial a(x;\theta)}{\partial x} \qquad \frac{\partial a(x;\theta)}{\partial \theta}$$

- Then follow iterative procedure

$$\boxed{\frac{\partial \mathcal{L}}{\partial a_l} = \left(\frac{\partial a_{l+1}}{\partial x_{l+1}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_{l+1}}} \qquad \boxed{\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left(\frac{\partial \mathcal{L}}{\partial a_l}\right)^T}$$

Backpropagation in practice

- Things are dead simple, just compute per module
  - $\frac{\partial a(x;\theta)}{\partial x}$
  - $\frac{\partial a(x;\theta)}{\partial \theta}$

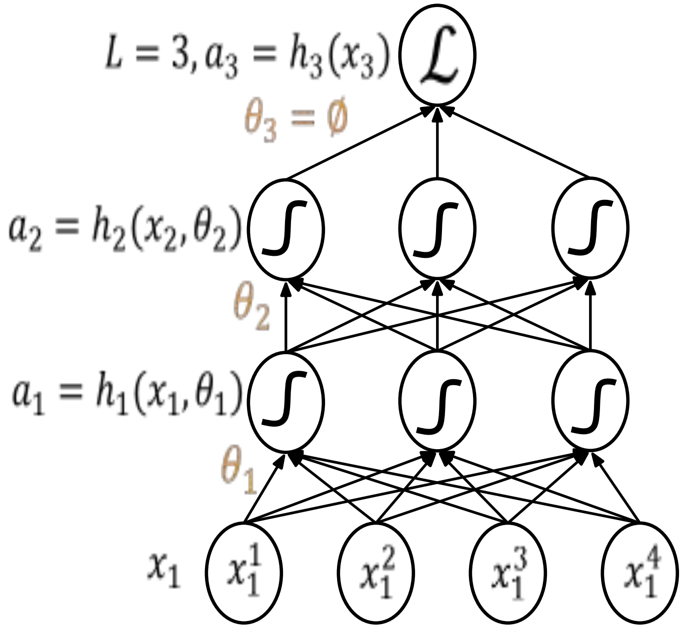
- Then follow iterative procedure [remember: $a_l = x_{l+1}$]

Derivatives from layer above

$$\boxed{\frac{\partial \mathcal{L}}{\partial a_l} = \left(\frac{\partial a_{l+1}}{\partial x_{l+1}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_{l+1}}} \qquad \boxed{\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left(\frac{\partial \mathcal{L}}{\partial a_l}\right)^T}$$

Module derivatives

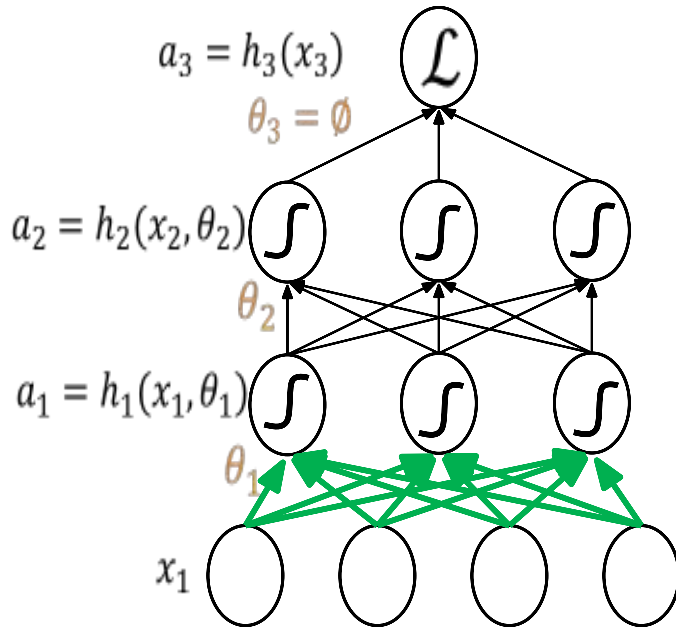
Backpropagation visualization



Backpropagation visualization at epoch (t)

Forward propagations

Compute and store $a_1 = h_1(x_1)$



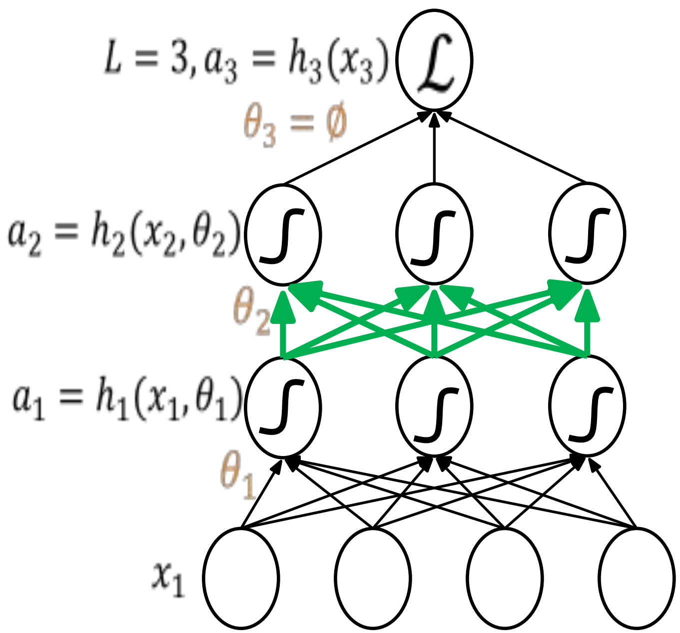
Exam ple $a_1 = \sigma(\theta_1 x_1)$

Store!!!

Backpropagation visualization at epoch (t)

Forward propagations

Compute and store $a_2 = h_2(x_2)$



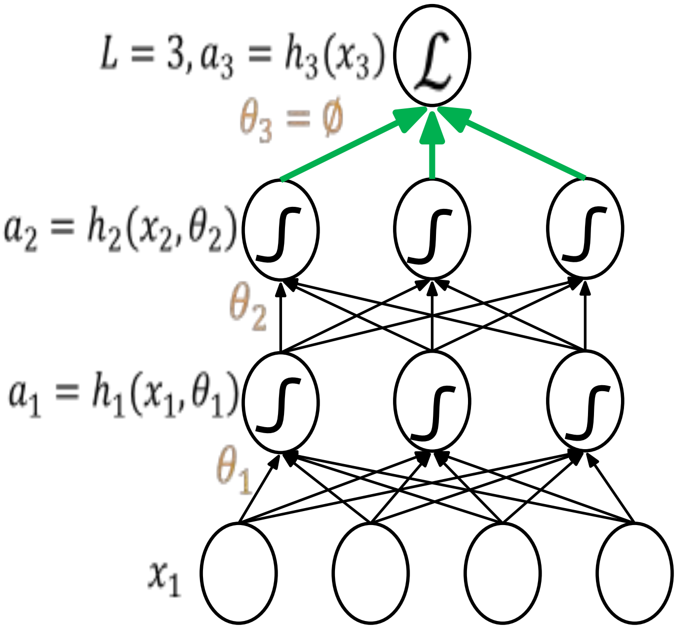
Exam
ple
$a_1 = \sigma(\theta_1 x_1)$
$a_2 = \sigma(\theta_2 x_2)$

Store!!!

Backpropagation visualization at epoch (t)

Forward propagations

Compute and store $a_3 = h_3(x_3)$



Example

$a_1 = \sigma(\theta_1 x_1)$

$a_2 = \sigma(\theta_2 x_2)$

$a_3 = \|y - x_3\|^2$

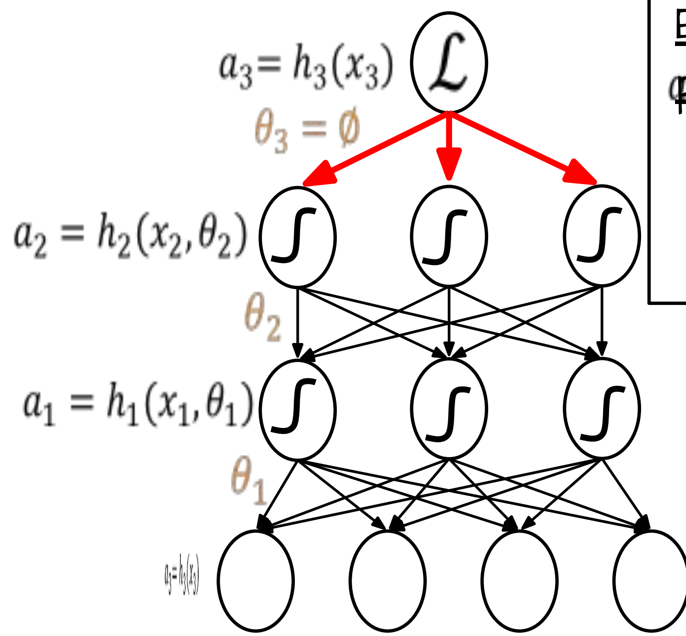
Store!!!

Backpropagation visualization at epoch (t)

Backpropagation

$\frac{\partial \mathcal{L}}{\partial a_3} = \ldots \leftarrow$ Direct computation

$\cancel{\frac{\partial \mathcal{L}}{\partial \theta_3}}$



Example $\mathcal{L}(y, x_3) = h_3(x_3) = 0.5\ \|y - x_3\|^2$

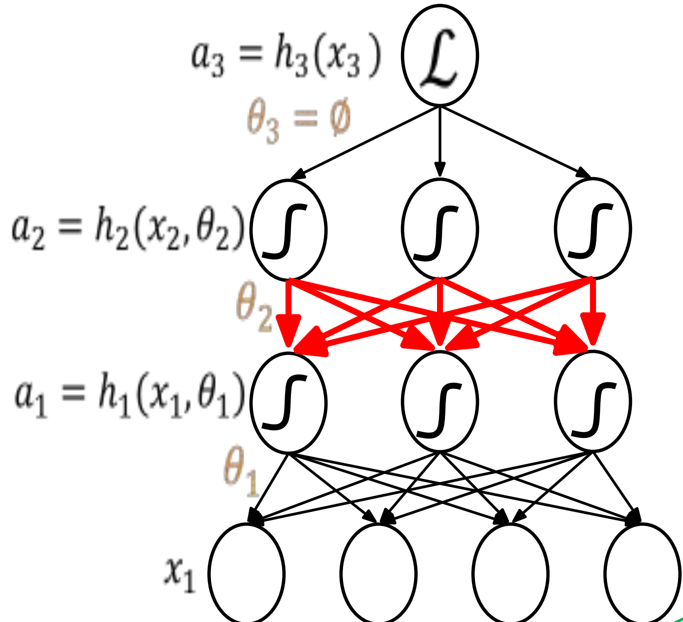
$$\frac{\partial \mathcal{L}}{\partial x_3} = -(y - x_3)$$

Backpropagation visualization at epoch (t)

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial a_2} = \frac{\partial \mathcal{L}}{\partial a_3} \cdot \frac{\partial a_3}{\partial a_2}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial \theta_2}$$



Example

$$\mathcal{L}(y, x_3) = 0.5\,\|y - x_3\|^2$$

$$x_3 = a_2$$

$$a_2 = \sigma(\theta_2 x_2)$$

$$\frac{\partial \mathcal{L}}{\partial a_2} = \frac{\partial \mathcal{L}}{\partial x_3} = -(y - x_3)$$

$$\partial \sigma(x) = \sigma(x)(1 - \sigma(x))$$

$$\frac{\partial a_2}{\partial \theta_2} = x_2 \sigma(\theta_2 x_2)(1 - \sigma(\theta_2 x_2))$$

$$= x_2 a_2 (1 - a_2)$$

$$\frac{\partial \mathcal{L}}{\partial a_2} = -(y - x_3)$$

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial \mathcal{L}}{\partial a_2} x_2 a_2 (1 - a_2)$$

Stored during forward computations

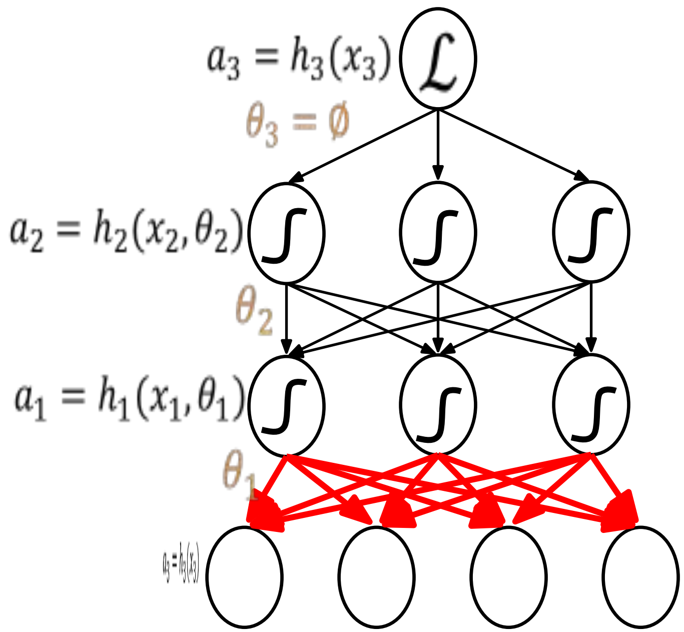
Store!!!

Backpropagation visualization at epoch (t)

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial a_1}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial \mathcal{L}}{\partial a_1} \cdot \frac{\partial a_1}{\partial \theta_1}$$



Example

$$\mathcal{L}(y, a_3) = 0.5 \, \|y - a_3\|^2$$

$$a_2 = \sigma(\theta_2 x_2)$$

$$x_2 = a_1$$

$$a_1 = \sigma(\theta_1 x_1)$$

$$\frac{\partial a_2}{\partial a_1} = \frac{\partial a_2}{\partial x_2} = \theta_2 a_2 (1 - a_2)$$

$$\frac{\partial a_1}{\partial \theta_1} = x_1 a_1 (1 - a_1)$$

$$\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial a_2} \theta_2 a_2 (1 - a_2)$$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial \mathcal{L}}{\partial a_1} x_1 a_1 (1 - a_1)$$

Computed from the exact previous backpropagation step (Remember, recursive rule)

Adam [Ba & Kingma 2014]

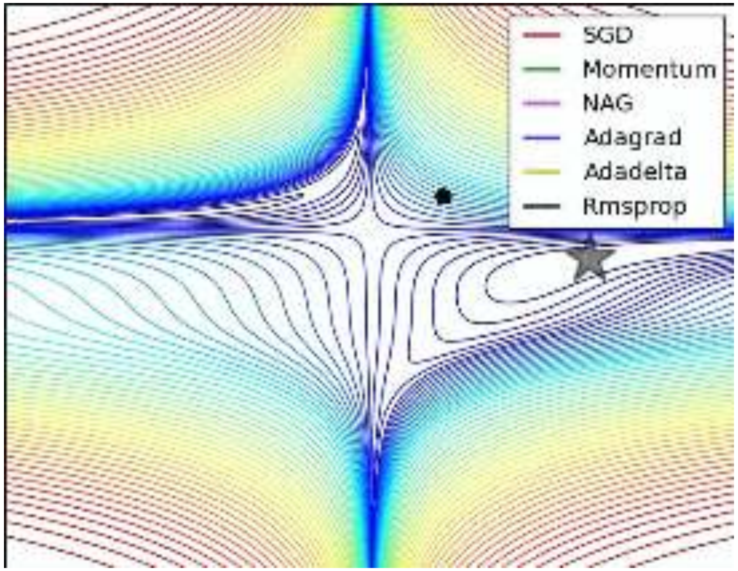
- One of the most popular learning algorithms

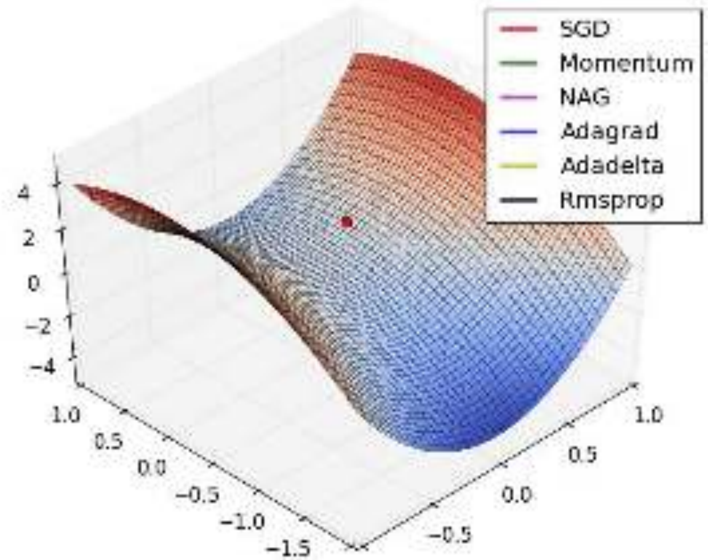
$$g_t = \nabla_\theta \mathcal{L}$$
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$
$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \frac{\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}} + \varepsilon}$$

  - Recommended values: $\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$

- Similar to RMSprop, but with momentum & correction bias

Visual overview





Picture credit:
[Alec Radford]

Convolutional Neural Networks

Or just Convnets/CNNs

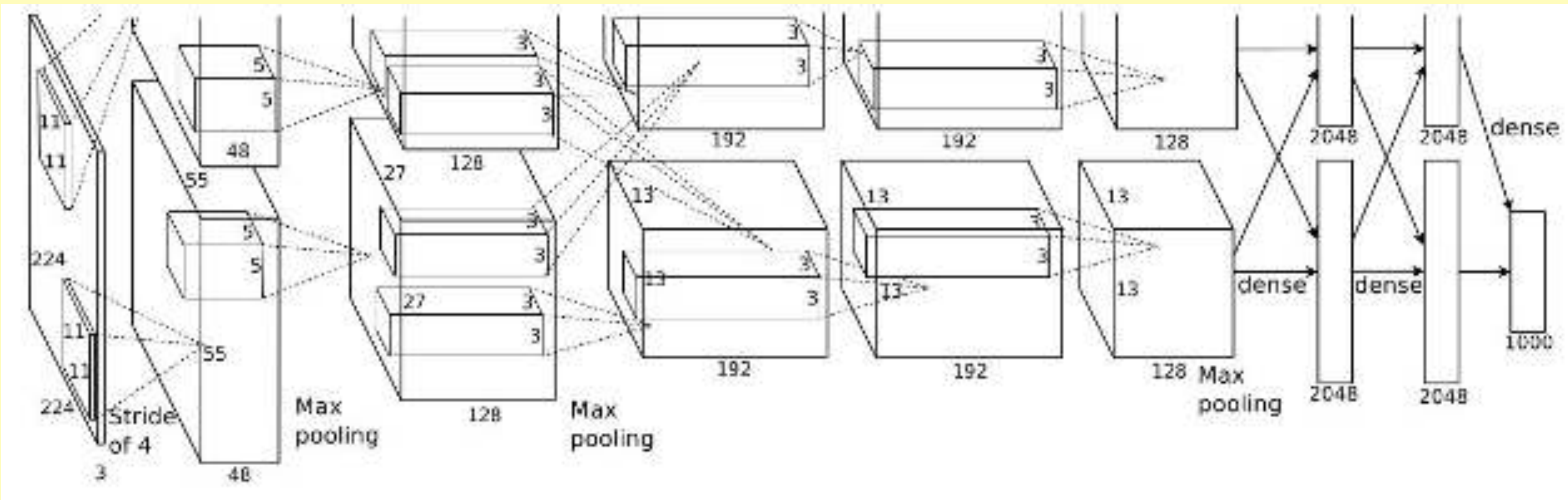


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

Convnets vs NNs

Question: Spatial structure?

- NNs: not modelled
- Convnets: Convolutional filters

Question: Huge input dimensionalities?

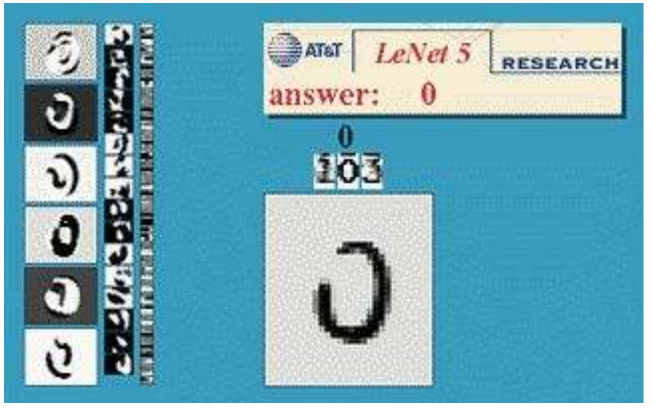
- NNs: scale badly in nr parameters and compute efficiency
- Convnets: Parameter sharing

Question: Local invariances?

- NNs: not hardwired into model
- Convnets: Pooling

Question: Translation equivariance?

- NN: not hardwired
- Convnets: translation equivariant



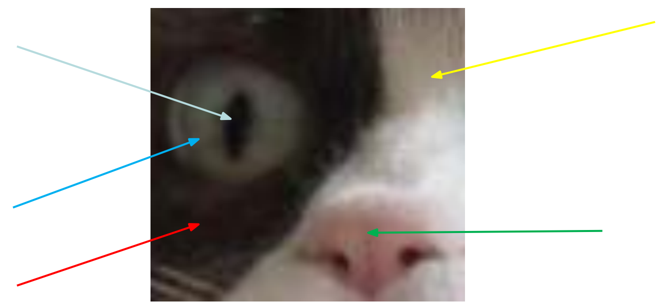
Spatial Information?

One pixel alone does not carry much information

Many pixels in the right order though → tons of information

I.e., Neighboring variables are correlated

And the variable correlations is the visual structure we want to learn



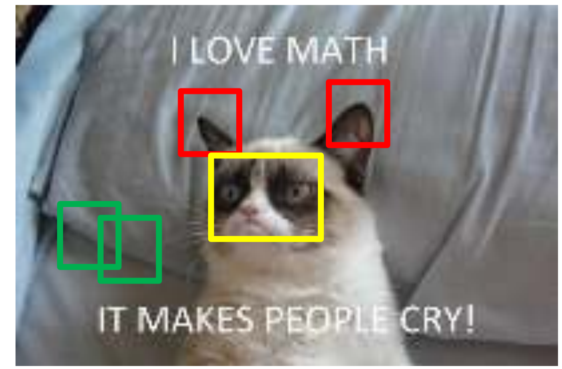
Parameter Sharing

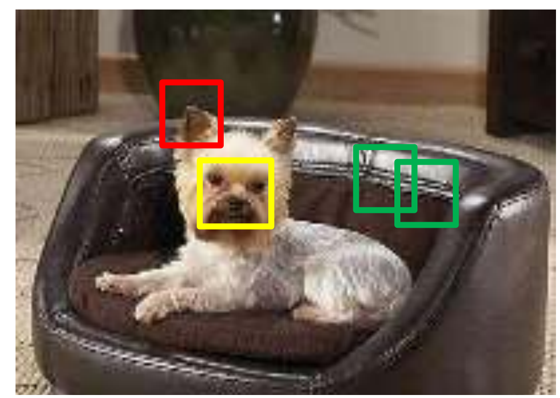
Natural images are stationary

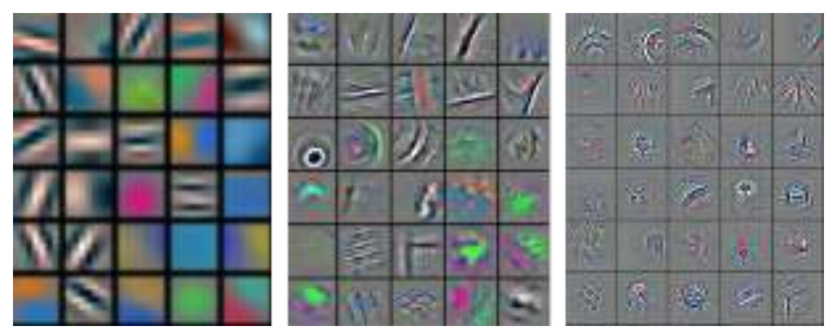
Visual features are common for
different parts of one or multiple image

If features are **local** *and* **similar** across
locations, why not **reuse** filters?

Local parameter sharing → Convolutions





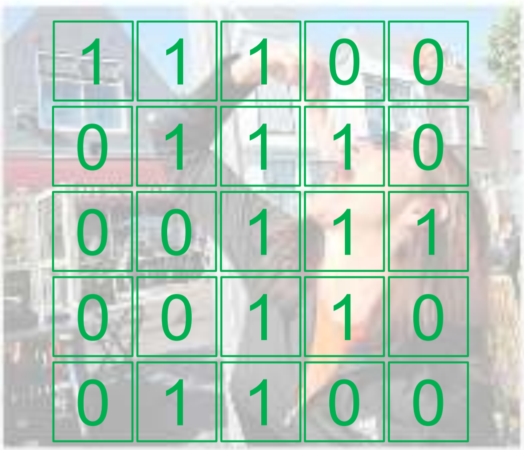


Convolutional filters

Original image

Convolutional filters

Original image



1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Convolutional filter 1

1	0	1
0	1	0
1	0	1

Convolutional filters

Original image

Convolutional filter 1

1	0	1
0	1	0
1	0	1

Convolving the image

1 x1	1 x0	1 x1	0	0
0 x0	1 x1	1 x0	1	0
0 x1	0 x0	1 x1	1	1
0	0	1	1	0
0	1	1	0	0

Result

4		

$$I(x,y) * h = \sum_{i=-a}^{a} \sum_{j=-b}^{b} \overbrace{I(x-i, y-j) \cdot h(i,j)}^{\text{Inner product}}$$

Convolutional filters

Original image

Convolutional filter 1

1	0	1
0	1	0
1	0	1

Convolving the image

1	1 x1	1 x0	0 x1	0
0	1 x0	1 x1	1 x0	0
0	0 x1	1 x0	1 x1	1
0	0	1	1	0
0	1	1	0	0

Result

4	3	

$$I(x,y) * h = \sum_{i=-a}^{a} \sum_{j=-b}^{b} \overbrace{I(x-i, y-j) \cdot h(i,j)}^{\text{Inner product}}$$

Convolutional filters

Original image

Convolutional filter 1

1	0	1
0	1	0
1	0	1

Convolving the image

1	1	1	0	0
0	1	1	1	0
0	0	1 x1	1 x0	1 x1
0	0	1 x0	1 x1	0 x0
0	1	1 x1	0 x0	0 x1

Result

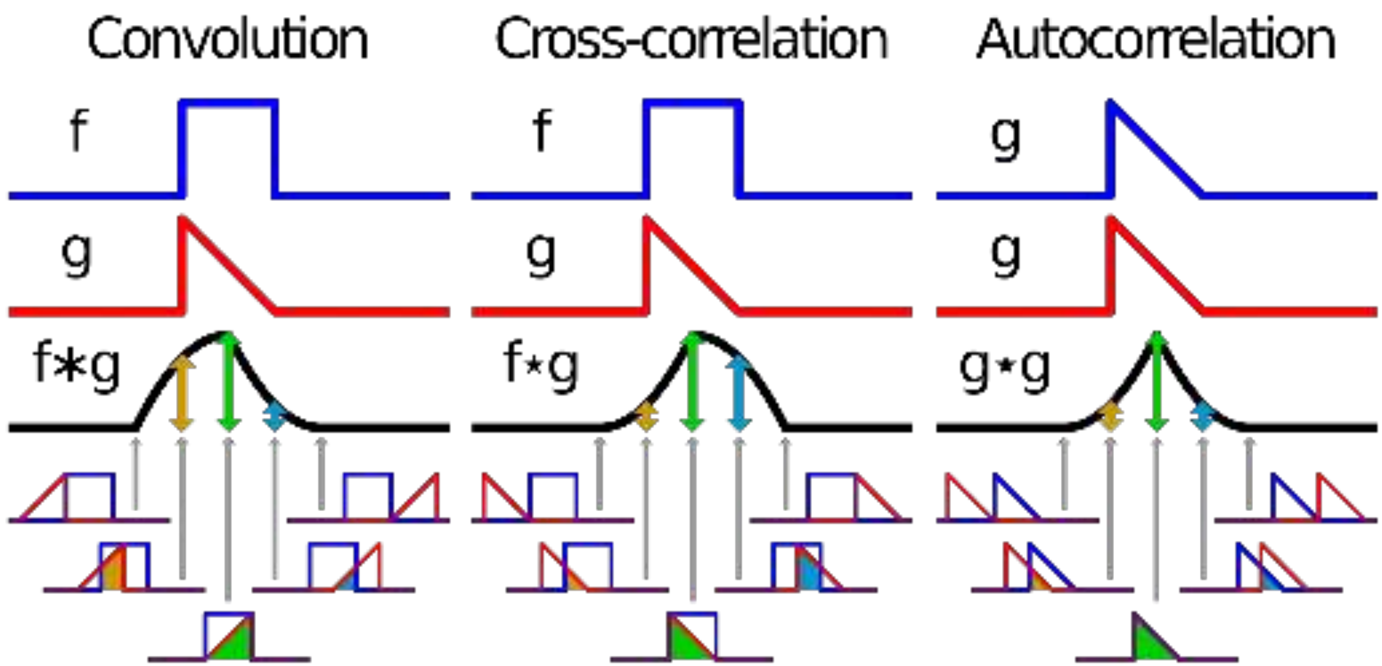
4	3	4
2	4	3
2	3	4

$$I(x,y) * h = \sum_{i=-a}^{a} \sum_{j=-b}^{b} \overbrace{I(x-i, y-j) \cdot h(i,j)}^{\text{Inner product}}$$

Why call them convolutions?

Definition *The convolution of two functions f and g is denoted by $*$ as the integral of the product of the two functions after one is reversed and shifted*

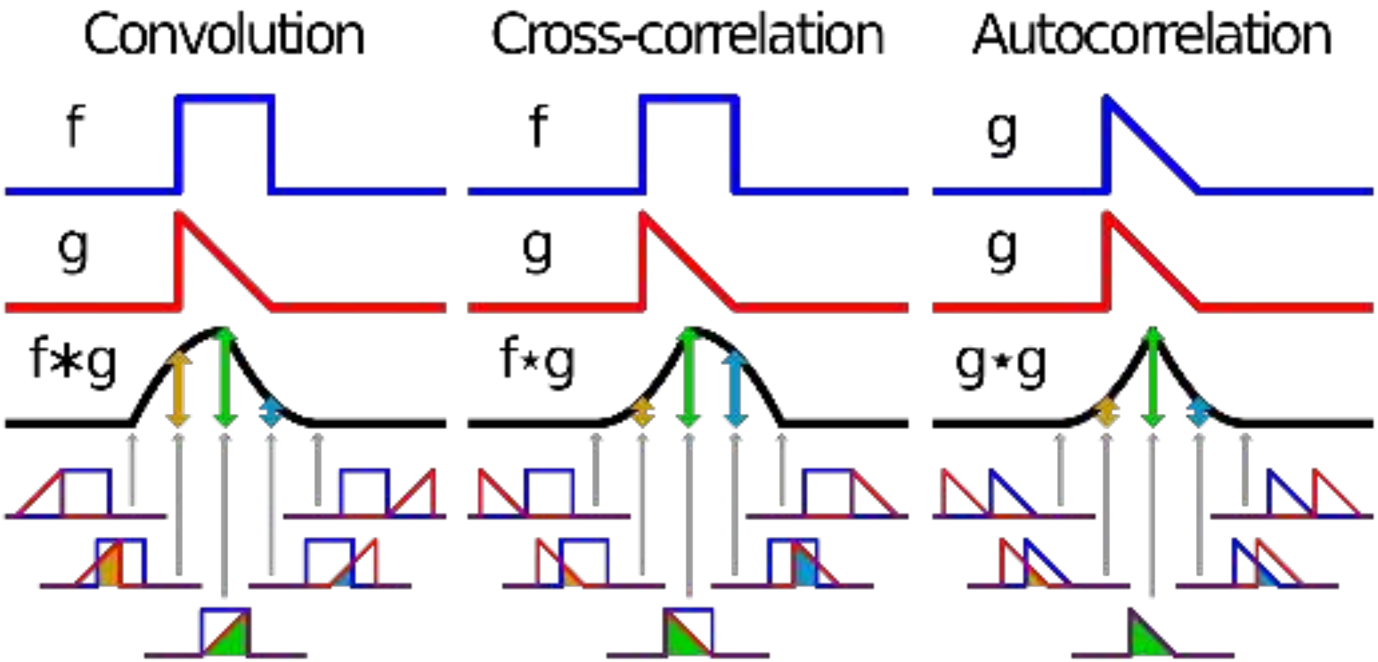
$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)\,d\tau$$



Quiz: Notice anything weird?

Definition *The convolution of two functions f and g is denoted by $*$ as the integral of the product of the two functions after one is reversed and shifted*

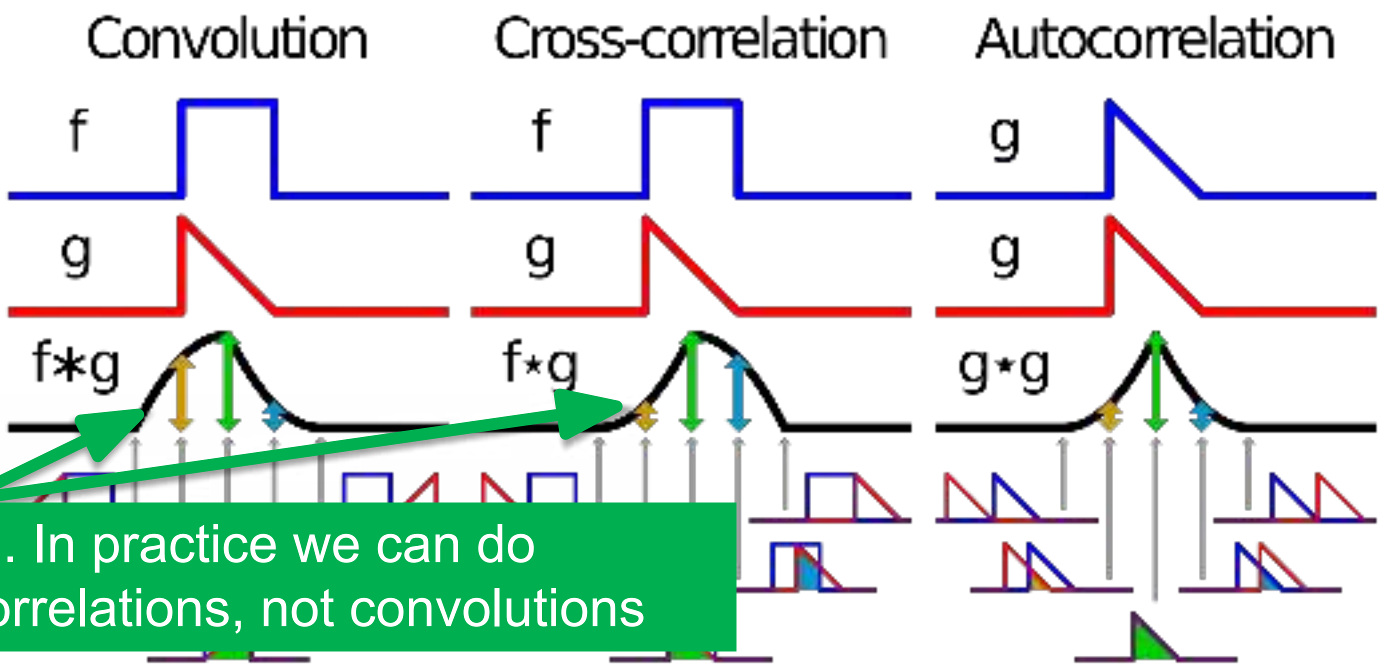
$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)\,d\tau$$



Quiz: Notice anything weird?

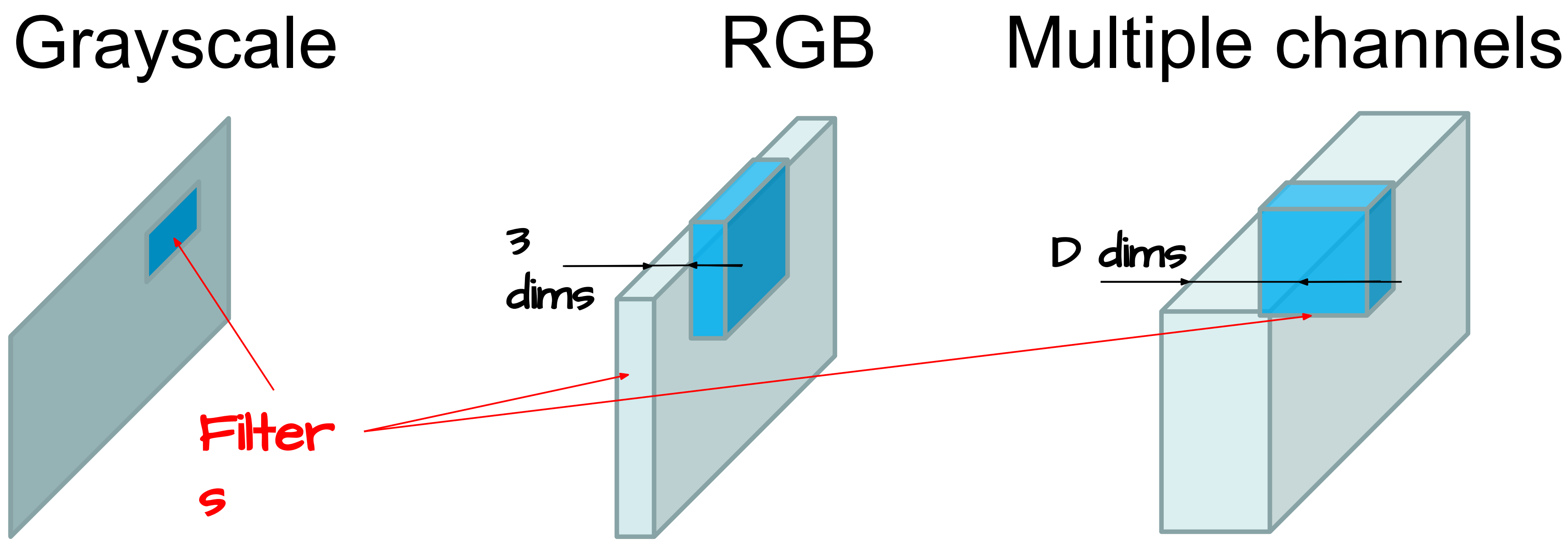
Definition *The convolution of two functions f and g is denoted by $*$ as the integral of the product of the two functions after one is reversed and shifted*

$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)\,d\tau$$



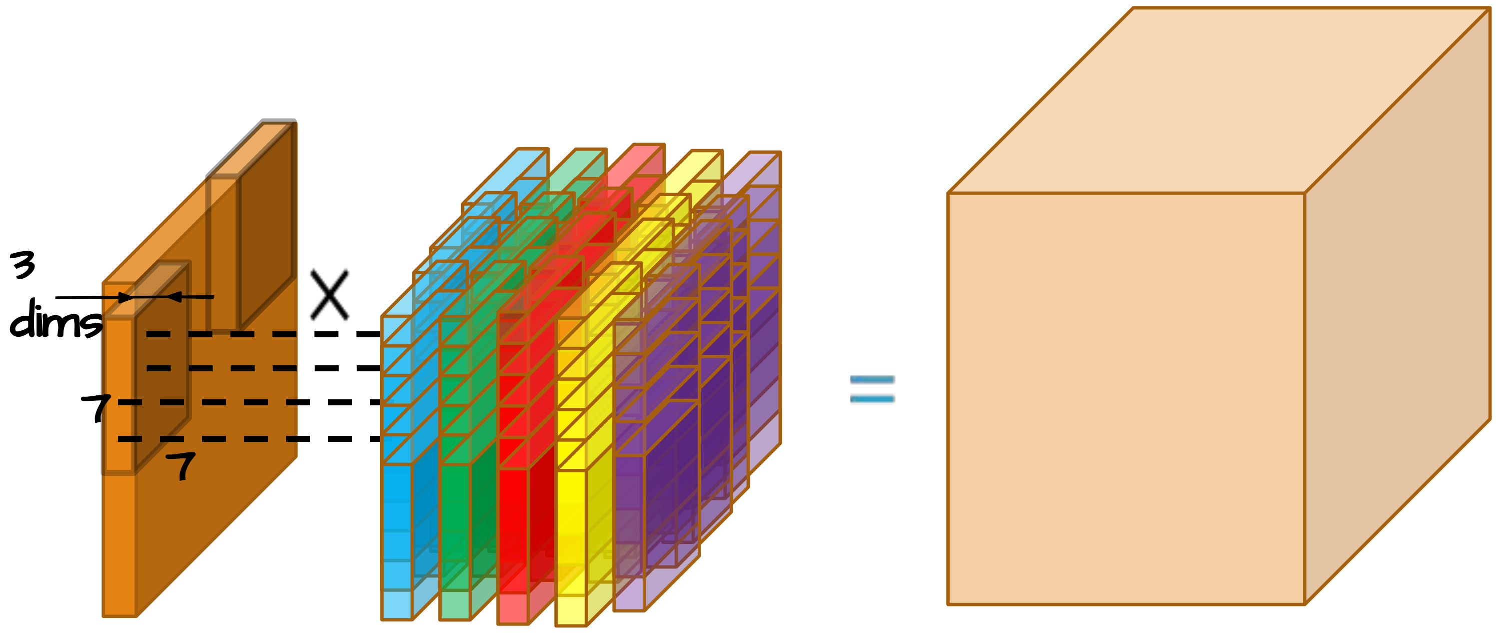
Inverted. In practice we can do cross-correlations, not convolutions

Filters have width/height/depth



How many parameters **per** filter? $\#params = H \times W \times D$

Parameter Sharing



$$A'_{i_x i_y, c} = \sum_{j_x, j_y, k} W_{i_x - j_x, i_y - j_y, ck} A_{j_x, j_y, k}\,.$$

Assume the image is 30x30x3.
1 column of filters common across the image.
How many parameters in total?

Depth of 5
$\times$ 7 * 7 * 3 parameters per filter

735 parameters in total

Local connectivity

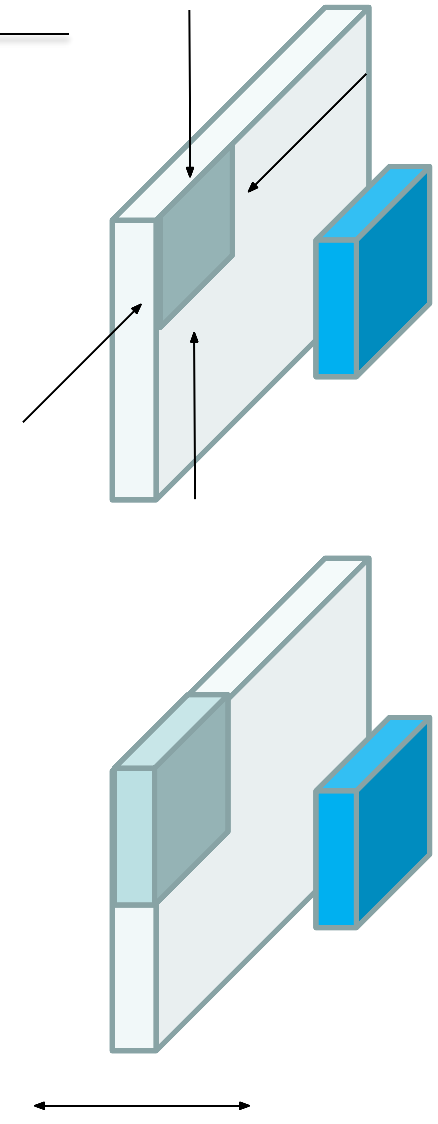
The weight connections are surface-wise local!

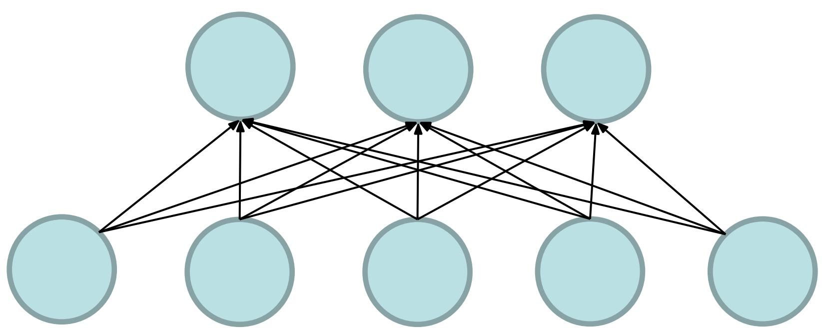
- Local connectivity

The weights connections are depth-wise global

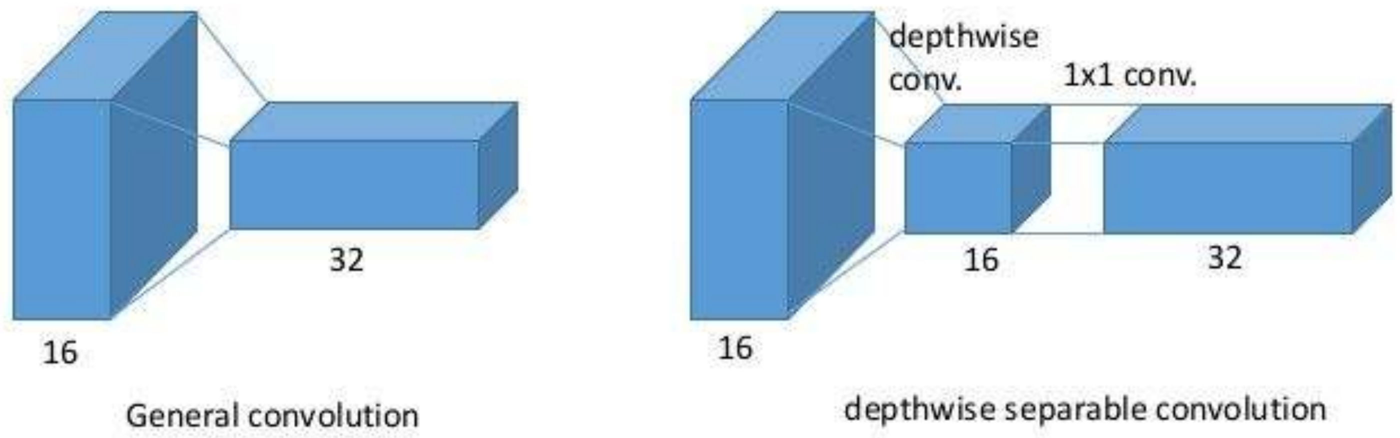
For standard neurons no local connectivity

- Everything is connected to everything





Depthwise Convolution



$$A'_{i_x i_y, c} = \sum_{j_x, j_y, k} W_{i_x - j_x, i_y - j_y, ck} \, A_{j_x, j_y, k} \rightarrow$$

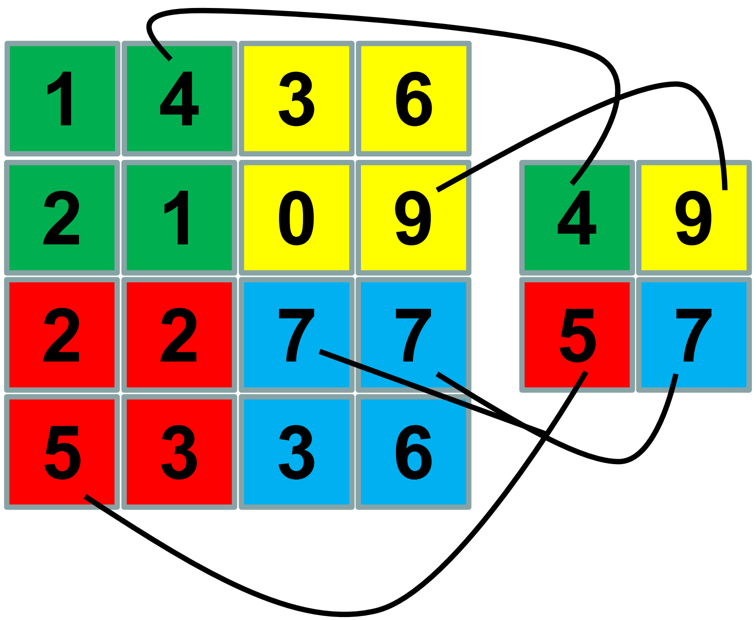
$$\sum_k K_{ck} \sum_{j_x, j_y} W_{i_x - j_x, i_y - j_y, k} \, A_{j_x, j_y, k}$$

Pooling

Often we want to summarize the local information into a single code vector

Feature aggregation ≡ Pooling

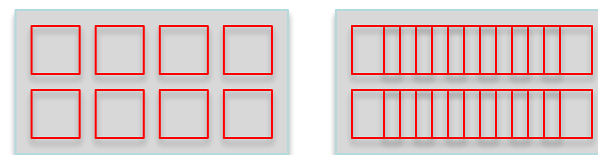
- ❑ Pooled feature invariant to small local transformations. Only the strongest activation is retained
- ❑ Output dimensions → Faster computations
- ❑ Keeps most salient information
- ❑ Different dimensionality inputs can now be compared



Implementation details

Stride

- every how many pixels do you compute a convolution
- equivalent to sampling coefficient, influences output size

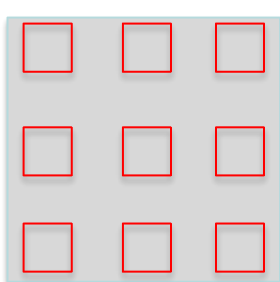


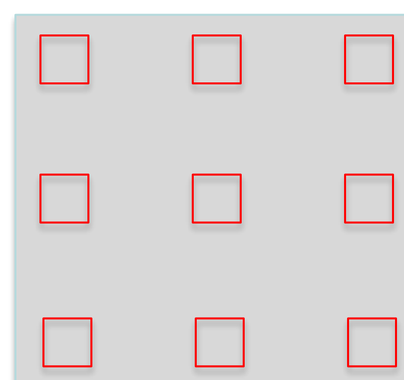
Padding

- Add 0s (or another value) around the layer input
- Prevent output from getting smaller and smaller

Dilation

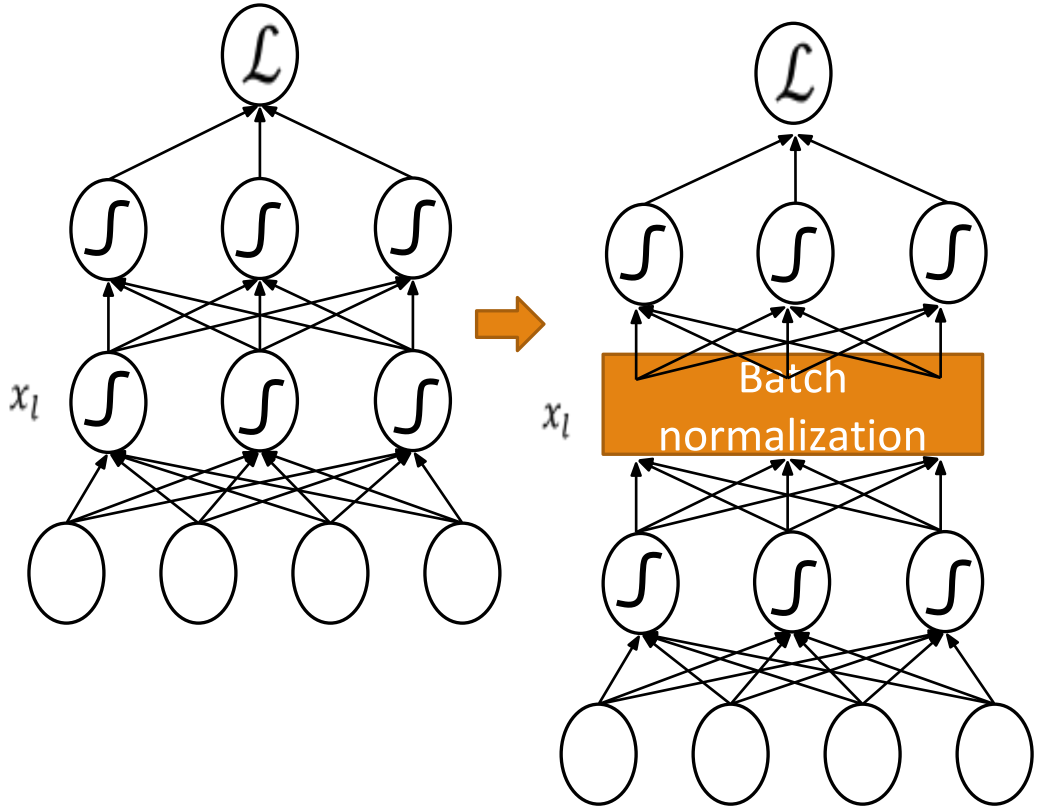
- Atrous convolutions





Batch normalization [Ioffe2015]

- Weights change → the distribution of the layer inputs changes per round
  - Covariance shift
- Normalize the layer inputs with batch normalization
  - Roughly speaking, normalize x_l to $N(0,1)$ and rescale



Batch normalization – The algorithm

- $\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ [compute mini-batch mean]
- $\sigma_B \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$ [compute mini-batch variance]
- $\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$ [normalize input]
- $\hat{y}_i \leftarrow \gamma x_i + \beta$ [scale and shift input]

Trainable parameters

Regularization

- Neural networks typically have thousands, if not millions of parameters
  - Usually, the dataset size smaller than the number of parameters
- Overfitting is a grave danger
- Proper weight regularization is crucial to avoid overfitting

$$\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \ell(y, a_L(x;\theta_{1,\dots,L})) + \lambda\Omega(\theta)$$

- Possible regularization methods
  - ℓ_2-regularization
  - ℓ_1-regularization
  - Dropout

ℓ_2-regularization

- Most important (or most popular) regularization

$$\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x;\theta_{1,\dots,L})) + \frac{\lambda}{2}\sum_l \|\theta_l\|^2$$

- The ℓ_2-regularization can pass inside the gradient descent update rule

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t(\nabla_\theta \mathcal{L} + \lambda\theta_l) \Rightarrow$$
$$\theta^{(t+1)} = (1 - \lambda\eta_t)\theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}$$

- λ is usually about $10^{-1}, 10^{-2}$

"Weight decay", because weights get smaller

ℓ_1-regularization

- ℓ_1-regularization is one of the most important regularization techniques

$$\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x; \theta_{1,\ldots,L})) + \frac{\lambda}{2}\sum_l \|\theta_l\|$$

- Also ℓ_1-regularization passes inside the gradient descent update rule

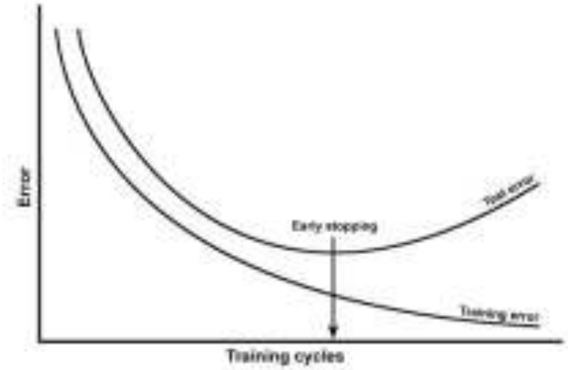
$$\theta^{(t+1)} = \theta^{(t)} - \lambda\eta_t \frac{\theta^{(t)}}{|\theta^{(t)}|} - \eta_t \nabla_\theta \mathcal{L}$$

Sign function

- ℓ_1-regularization → sparse weights
  - $\lambda \nearrow$ → more weights become 0

Early stopping

- To tackle overfitting another popular technique is early stopping
- Monitor performance on a separate validation set
- Training the network will decrease training error, as well validation error (although with a slower rate usually)
- Stop when validation error starts increasing
  - This quite likely means the network starts to overfit

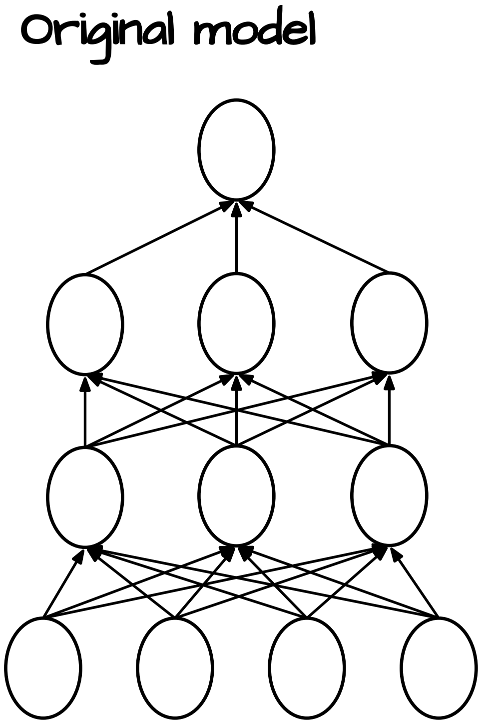


Dropout [Srivastava2014]

- During training setting activations randomly to 0
  - Neurons sampled at random from a Bernoulli distribution with $p = 0.5$
- At test time all neurons are used
  - Neuron activations reweighted by p
- Benefits
  - Reduces complex co-adaptations or co-dependencies between neurons
  - No "free-rider" neurons that rely on others
  - Every neuron becomes more robust
  - Decreases significantly overfitting
  - Improves significantly training speed

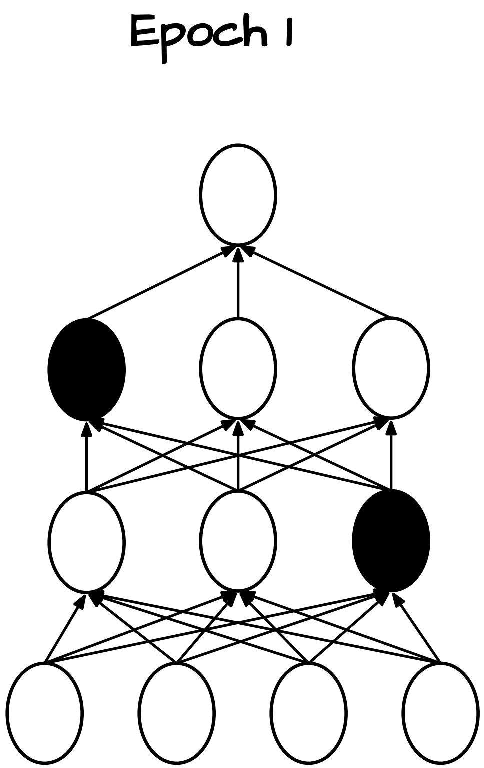
Dropout

- Effectively, a different architecture at every training epoch
  - Similar to model ensembles



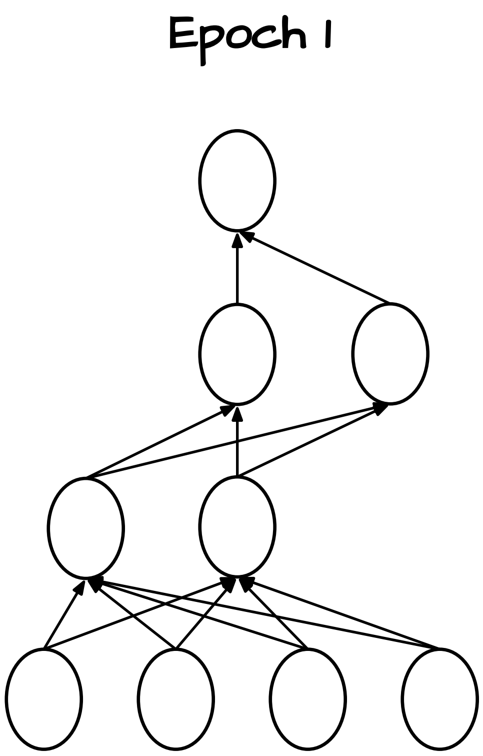
Dropout

- Effectively, a different architecture at every training epoch
  - Similar to model ensembles



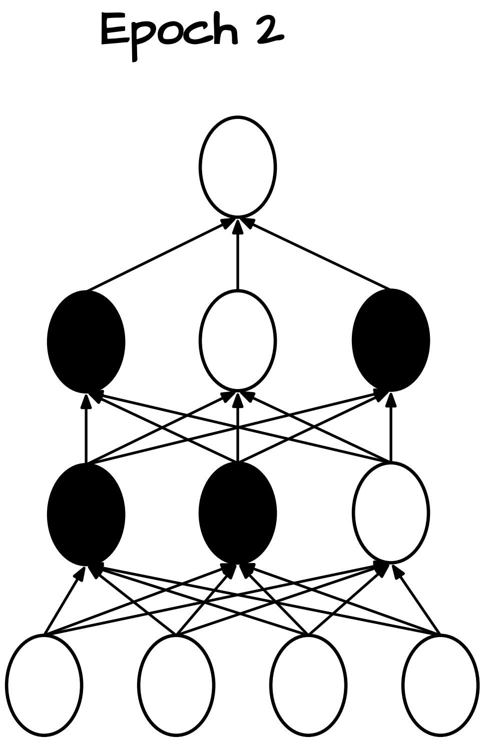
Dropout

- Effectively, a different architecture at every training epoch
  - Similar to model ensembles



Dropout

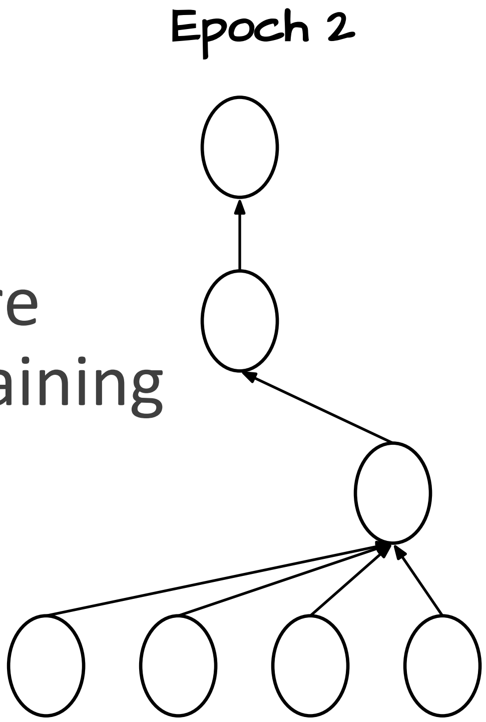
- Effectively, a different architecture at every training epoch
  - Similar to model ensembles



Dropout

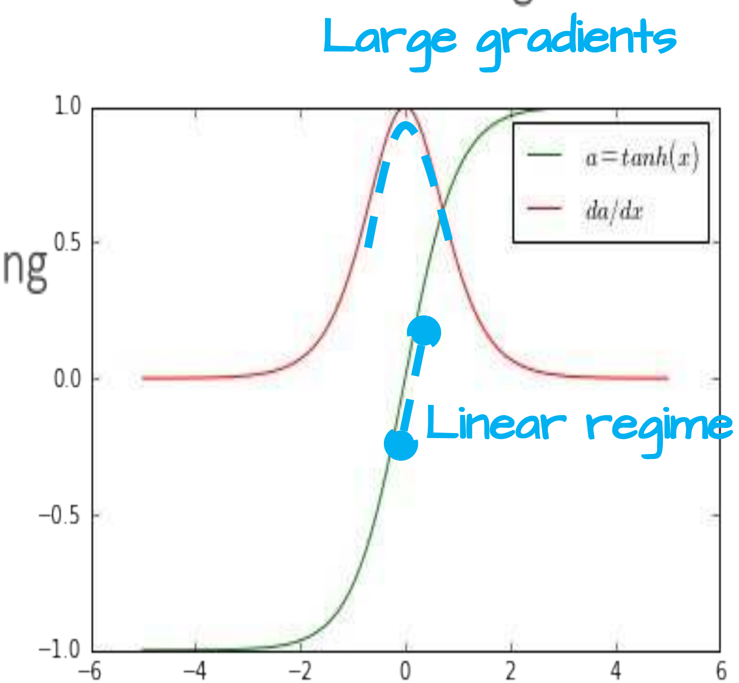
- Effectively, a different architecture at every training epoch
  - Similar to model ensembles

- At test time keep all neurons but multiply output by p (e.g. 0.5) to compensate for the fact that more of them are active than during training



Weight initialization

- There are few contradictory requirements
- Weights need to be small enough
  - around origin ($\vec{\mathbf{0}}$) for symmetric functions (tanh, sigmoid)
  - When training starts better stimulate activation functions near their linear regime
  - larger gradients → faster training
- Weights need to be large enough
  - Otherwise signal is too weak for any serious learning



Xavier initialization [Glorot2010]

- For $a = \theta x$ the variance is

$$var(a) = E[x]^2 var(\theta) + \mathrm{E}[\theta]^2 var(x) + var(x) var(\theta)$$

- Since $E[x] = E[\theta] = 0$

$$var(a) = var(x) var(\theta) \approx d \cdot var\left(x^i\right) var\left(\theta^i\right)$$

- For $var(a) = var(x) \Rightarrow var\left(\theta^i\right) = \frac{1}{d}$
- Draw random weights from

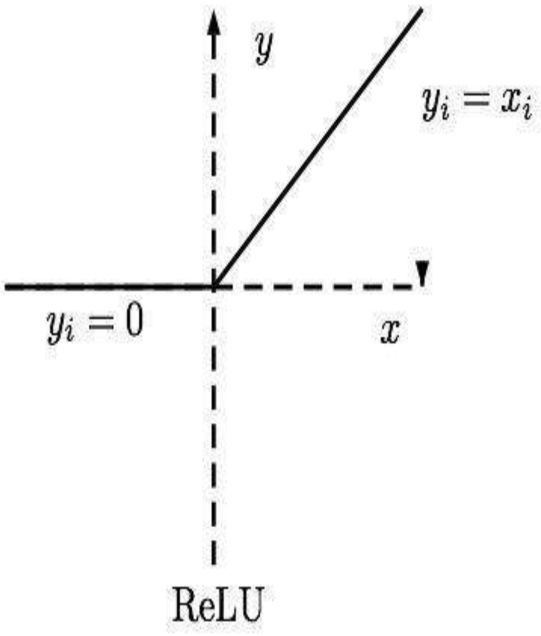
$$\theta \sim N\left(0, \sqrt{1/d}\right)$$

where d is the number of neurons in the input

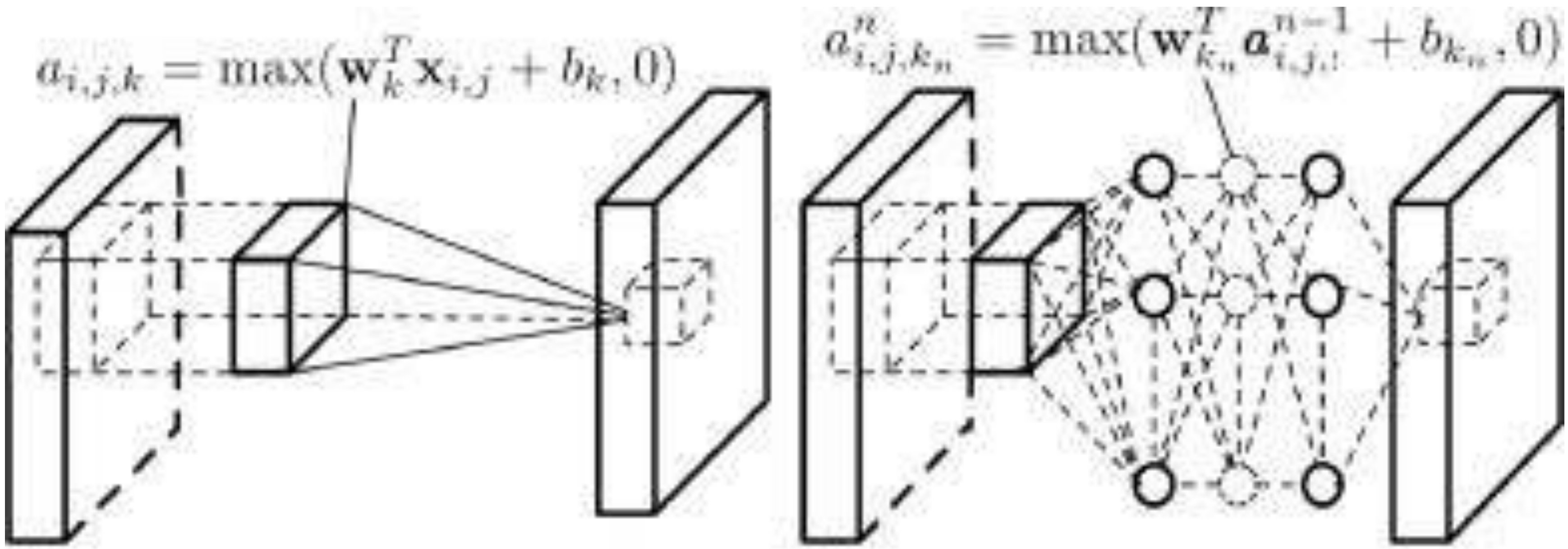
[He2015] initialization for ReLUs

- Unlike sigmoids, ReLUs ground to 0 the linear activations half the time
- Double weight variance
  - Compensate for the zero flat-area →
  - Input and output maintain same variance
  - Very similar to Xavier initialization
- Draw random weights from $w \sim N\left(0, \sqrt{2/d}\right)$

  where d is the number of neurons in the input



Network-in-network [Lin et al., arXiv 2013]



(a) Linear convolution layer (b) Mlpconv layer

ResNet [He et al., CVPR 2016]

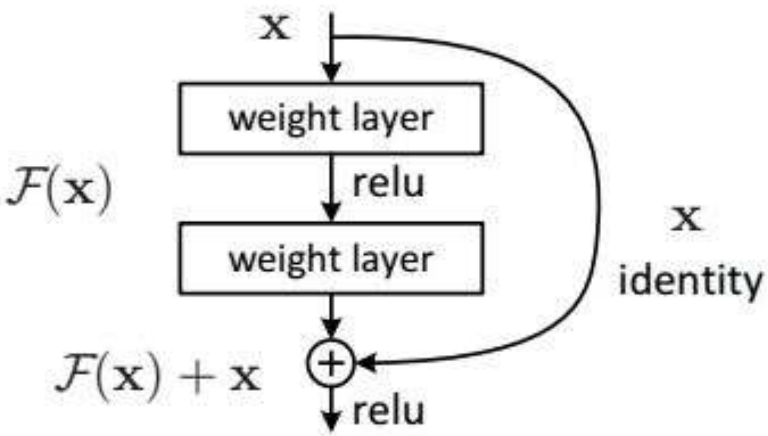
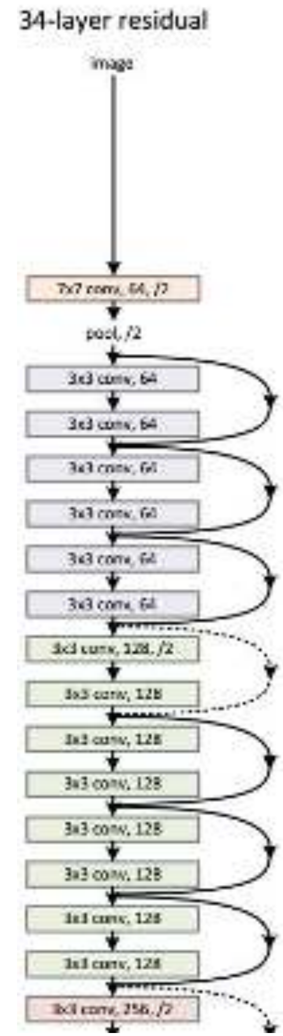


Figure 2. Residual learning: a building block.



No degradation anymore

Without residual connections deeper networks are untrainable

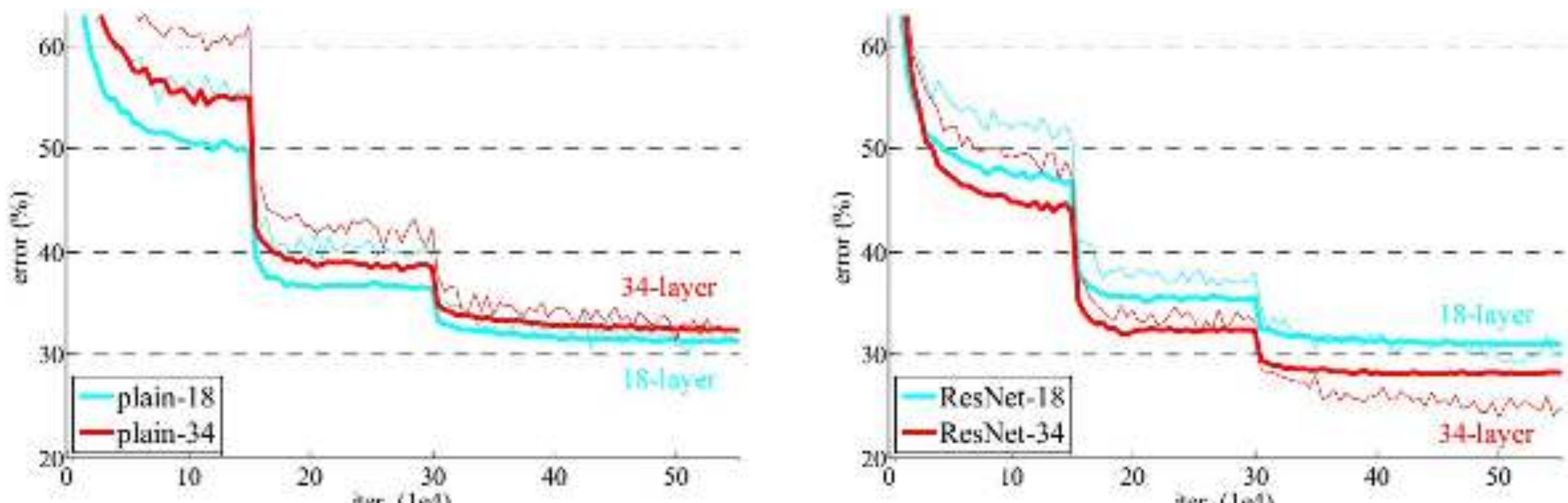


Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

ResNet breaks records

Ridiculously low error in ImageNet

Up to 1000 layers ResNets trained

- Previous deepest network ~30-40 layers on simple datasets

method	top-5 err. (**test**)
VGG [41] (ILSVRC'14)	7.32
GoogLeNet [44] (ILSVRC'14)	6.66
VGG [41] (v5)	6.8
PReLU-net [13]	4.94
BN-inception [16]	4.82
ResNet (ILSVRC'15)	**3.57**

Table 5. Error rates (%) of **ensembles**. The top-5 error is on the test set of ImageNet and reported by the test server.

Data augmentation [Krizhevsky2012]

Origin
al

Fli

Random
crop

Cont
rast

Ti
nt

Some practical tricks of the trade

- For classification use entropy loss
- Use variant of ReLU as nonlinearity
- Use Adam SGD
- Use random minibatch at each iteration
- Normalize input to zero mean, unit variance
- Use batch-normalization
- Use dropout on fully connected layers
- Use ResNet architecture
- Think about weight inititalization
- Do extensive hyperparameter search
- Use data augmentation

Case studies

Alexnet

- Or the modern version of it, VGGnet

ResNet

- From 14 to 1000 layers

Google Inception

- Networks as Direct Acyclic Graphs (DAG)

Alexnet

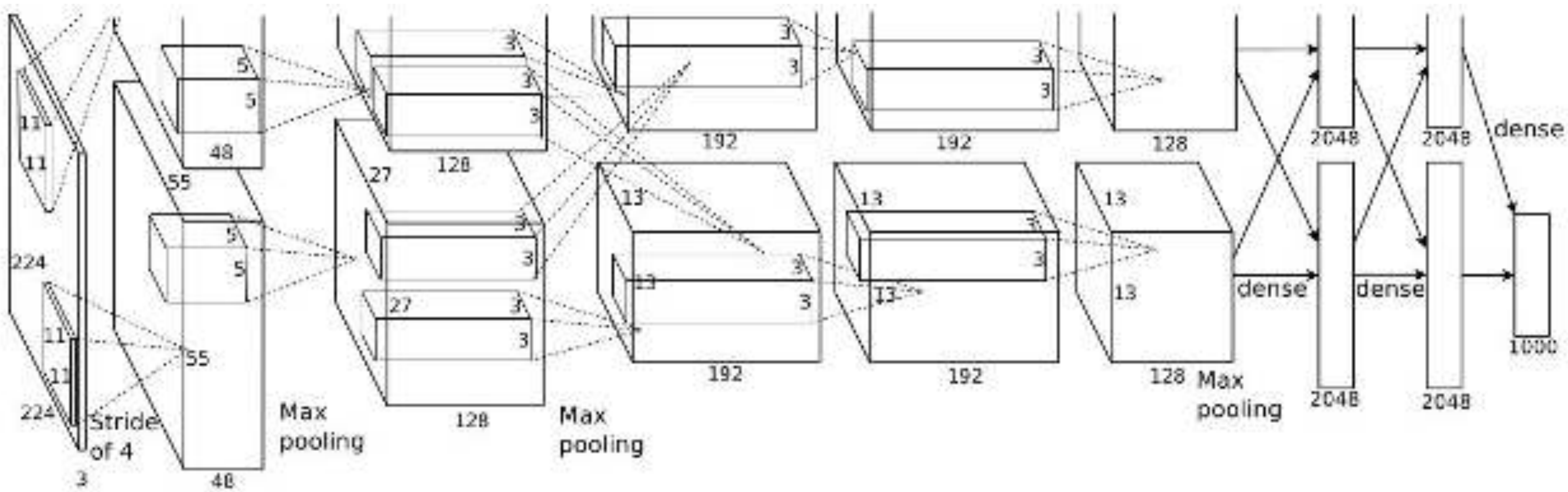
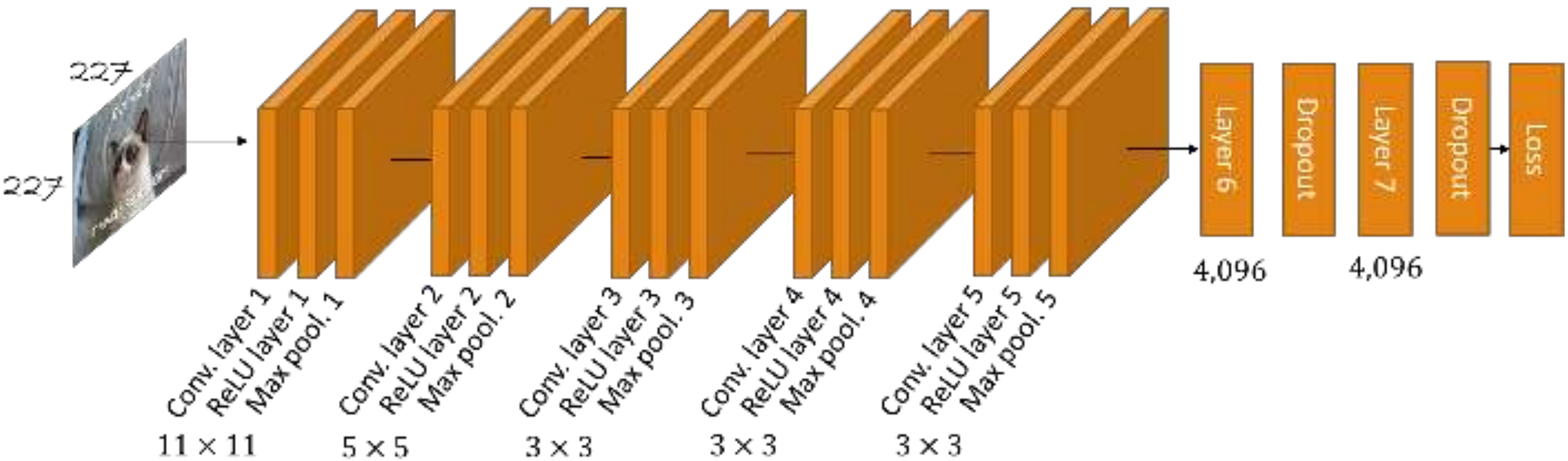


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

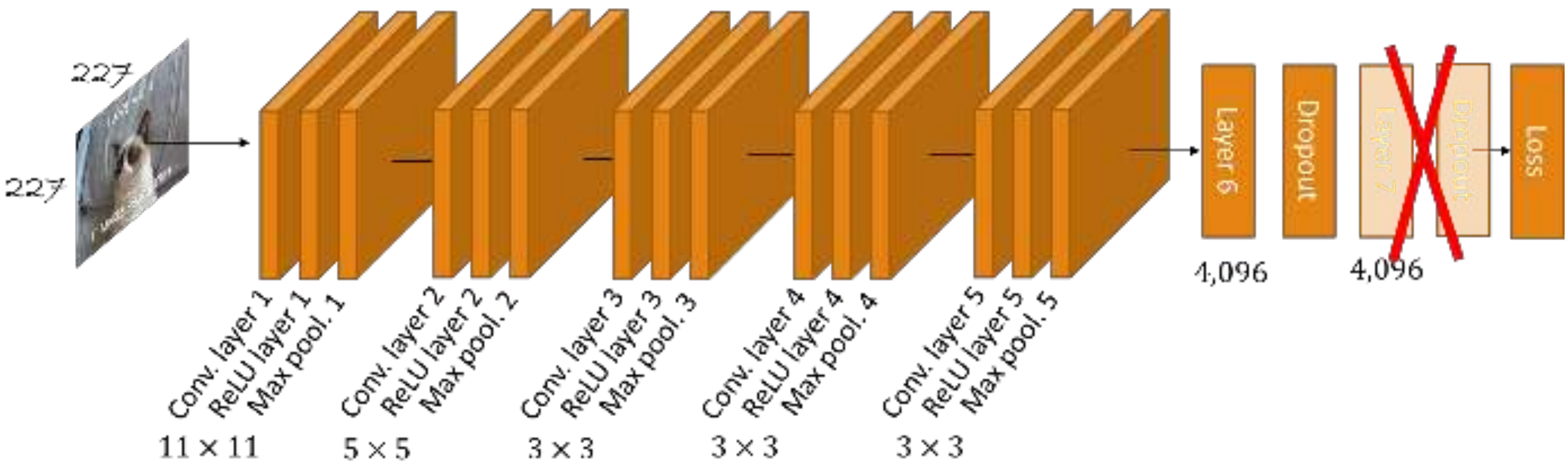
Architectural details

18.2% error in Imagenet



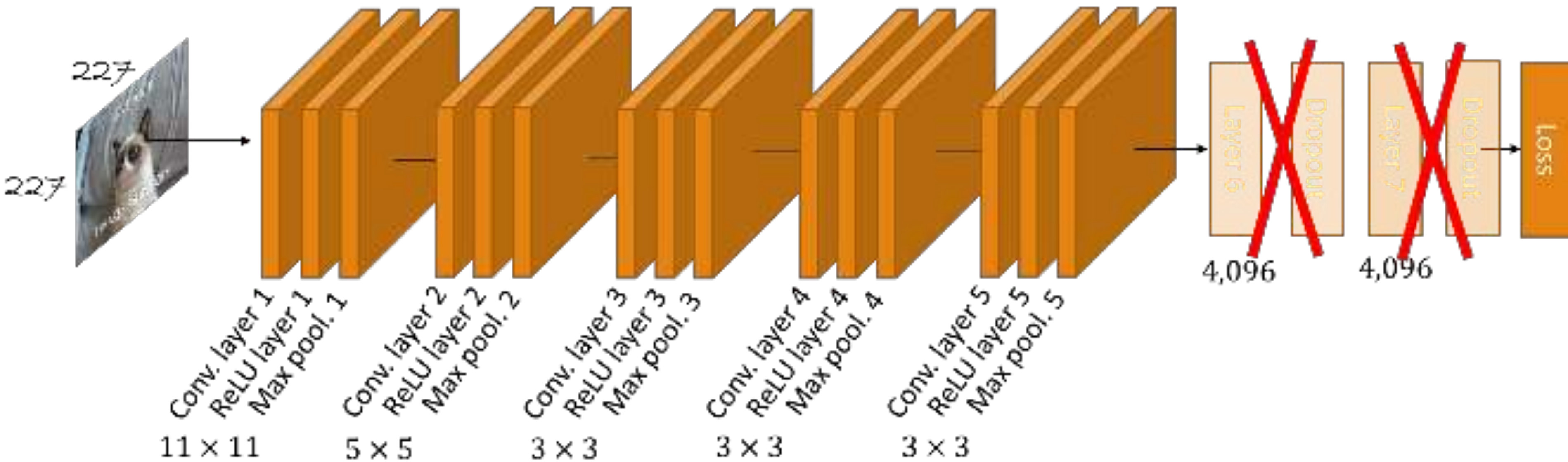
Removing layer 7

1.1% drop in performance, 16 million less parameters



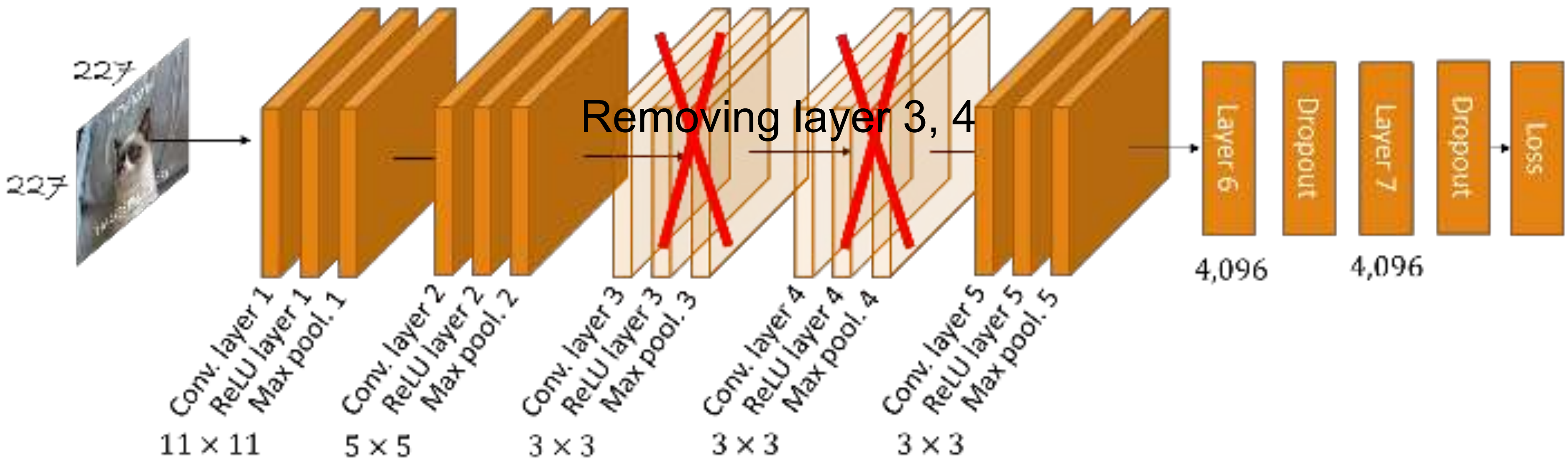
Removing layer 6, 7

5.7% drop in performance, 50 million less parameters



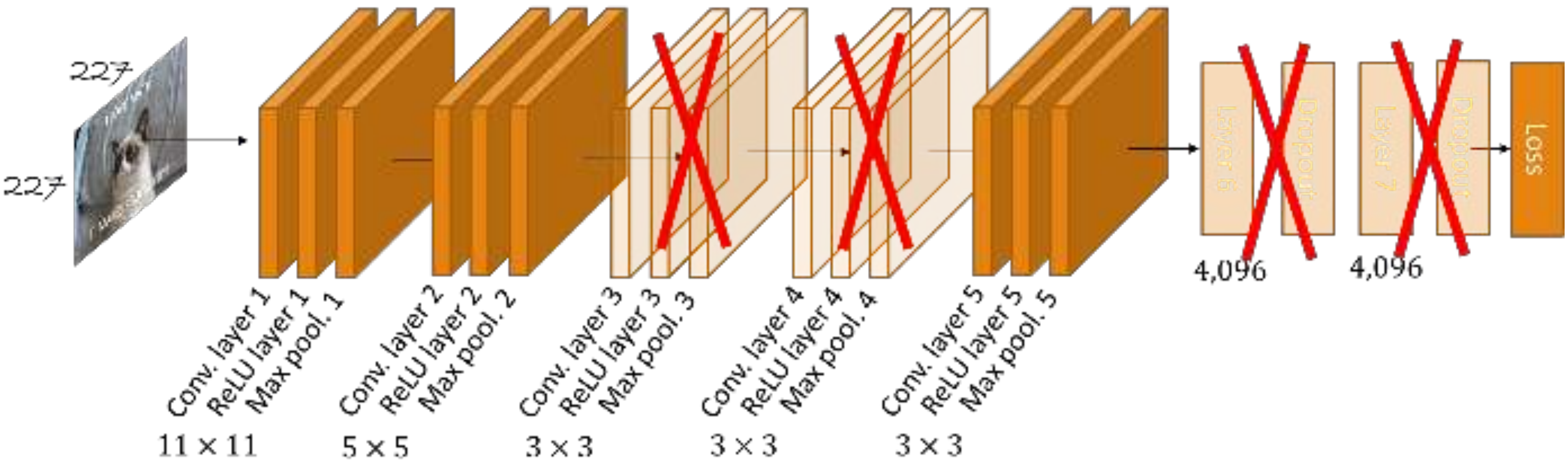
Removing layer 3, 4

3.0% drop in performance, 1 million less parameters. Why?



Removing layer 3, 4, 6, 7

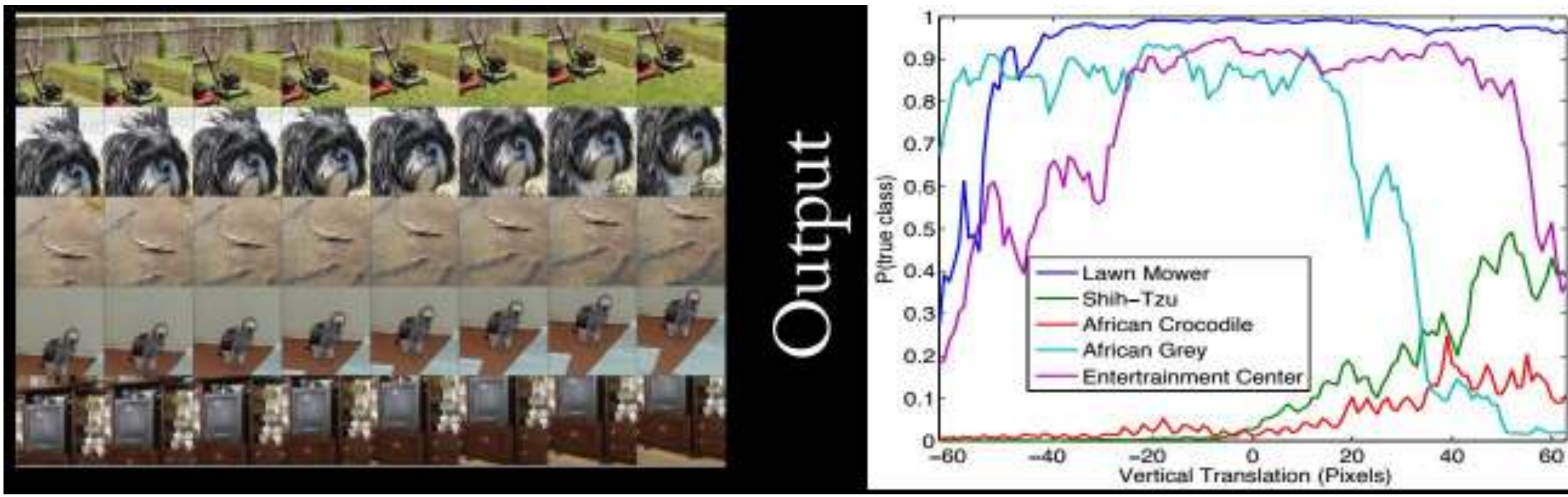
33.5% drop in performance. Conclusion? Depth!



Quiz: Translation invariance?

Credit: R. Fergus slides in Deep Learning Summer School 2016

Translation invariance

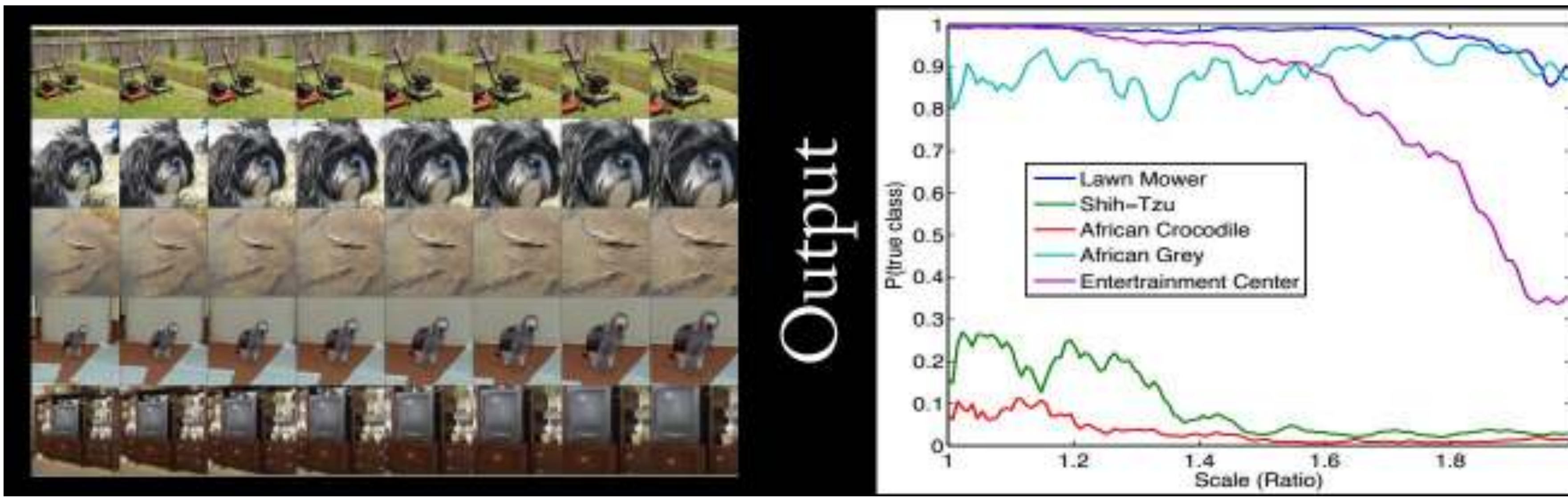


Credit: R. Fergus slides in Deep Learning Summer School 2016

Quiz: Scale invariance?

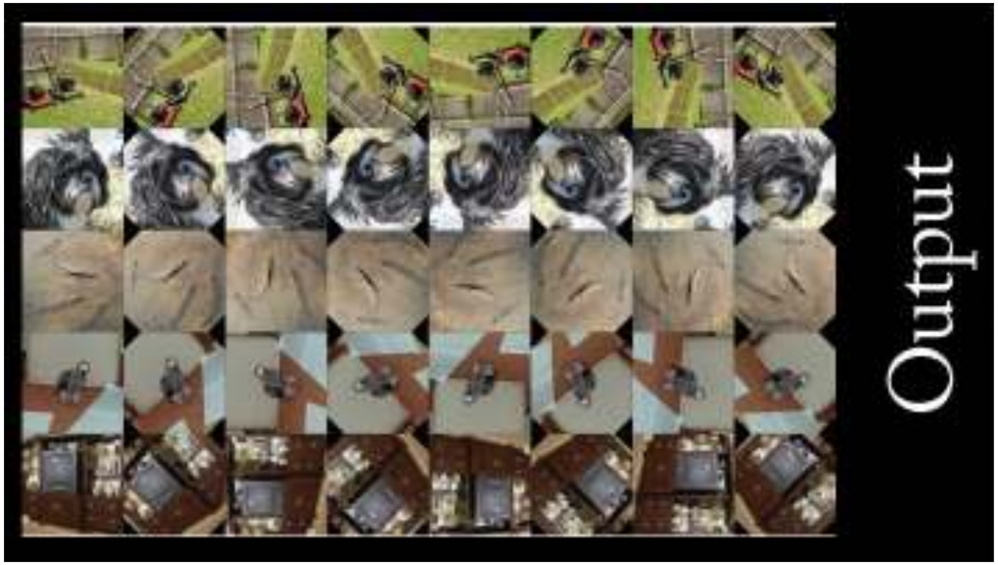
Credit: R. Fergus slides in Deep Learning Summer School 2016

Scale invariance



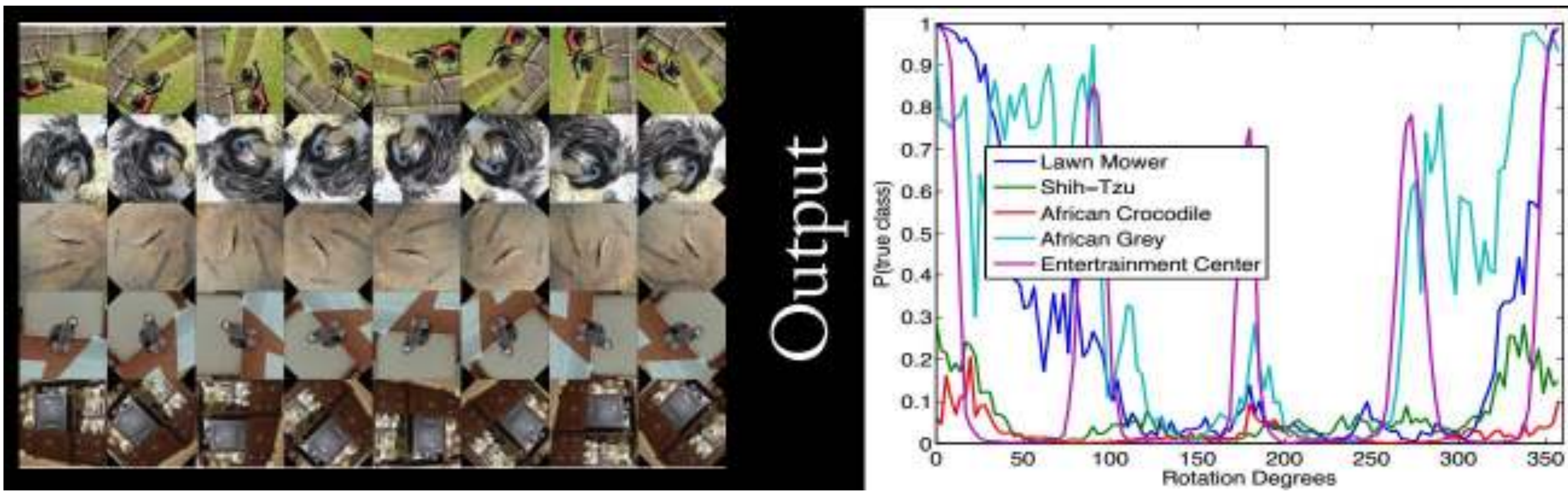
Credit: R. Fergus slides in Deep Learning Summer School 2016

Quiz: Rotation invariance?



Credit: R. Fergus slides in Deep Learning Summer School 2016

Rotation invariance

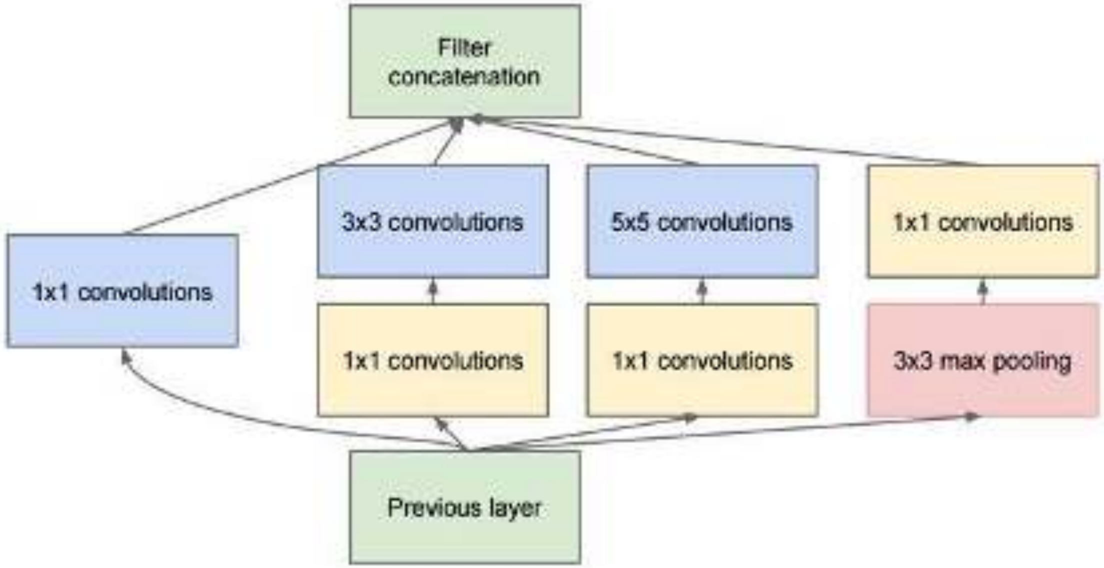


Credit: R. Fergus slides in Deep Learning Summer School 2016

Google Inception V1

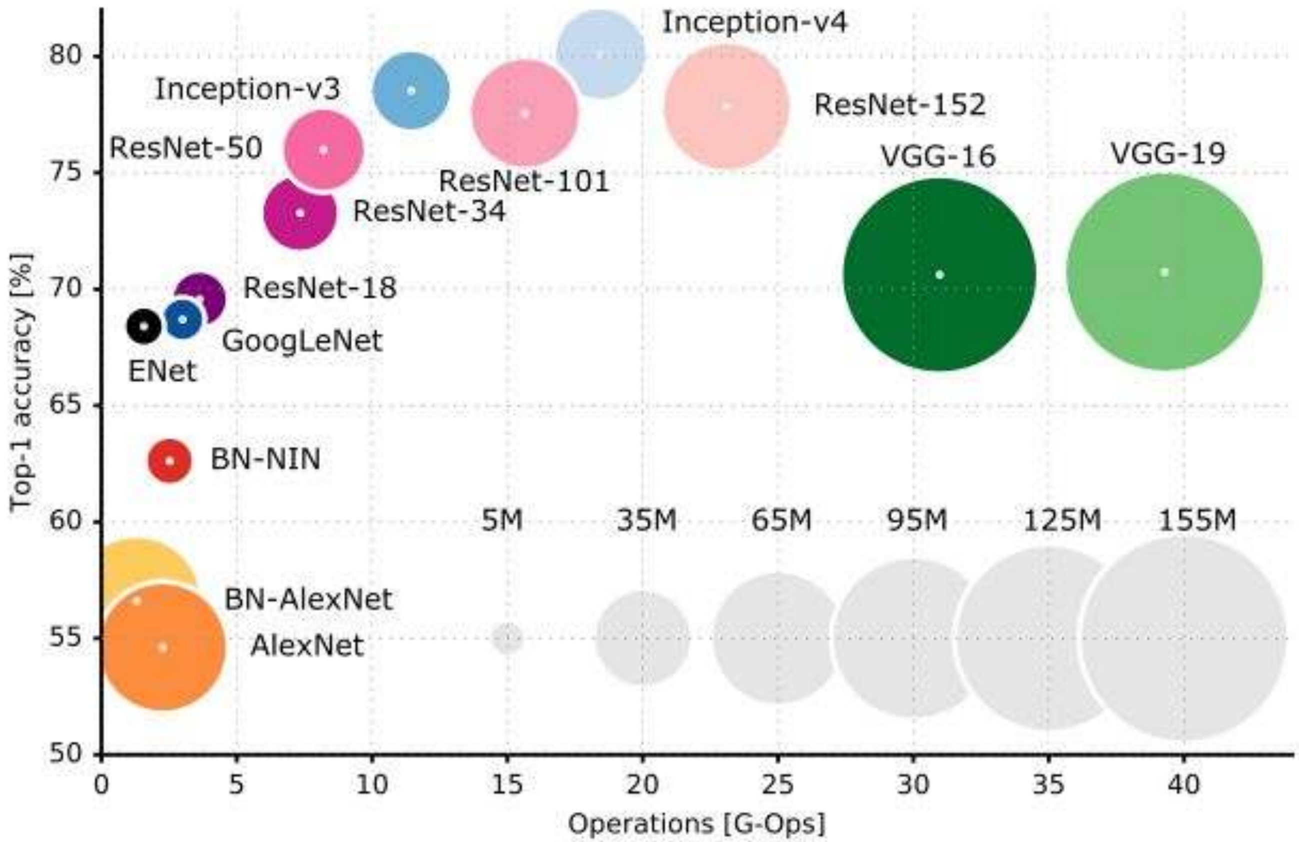
Instead of having convolutions (e.g. 3×3) directly, first reduce features by 1×1 convolutions

- E.g., assume we have 256 features in the previous layer
- Convolve with $256 \times 64 \times 1 \times 1$
- Then convolve with $64 \times 64 \times 3 \times 3$
- Then convolve with $64 \times 256 \times 1 \times 1$



Credit: https://culurciello.github.io/tech/2016/06/04/nets.html

State-of-the-art



Credit: https://culurciello.github.io/tech/2016/06/04/nets.html

Recurrent Networks

So far, all tasks assumed ***stationary*** data

Neither all data, nor all tasks are stationary though

Sequential data

Or …

What about text that is naturally sequential?

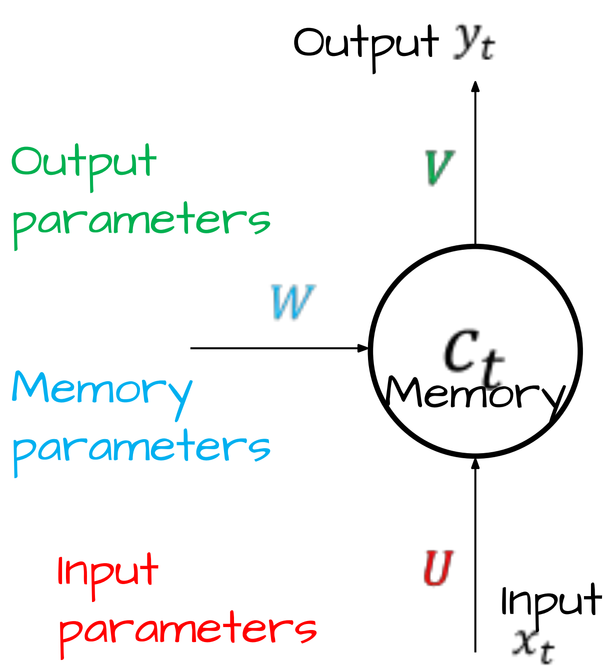
We need memory to handle long range correlations.

$$\boxed{\Pr(x) = \prod_i \Pr(x_i \mid x_1, \ldots, x_{i-1})}$$

Recurrent Networks

Simplest model

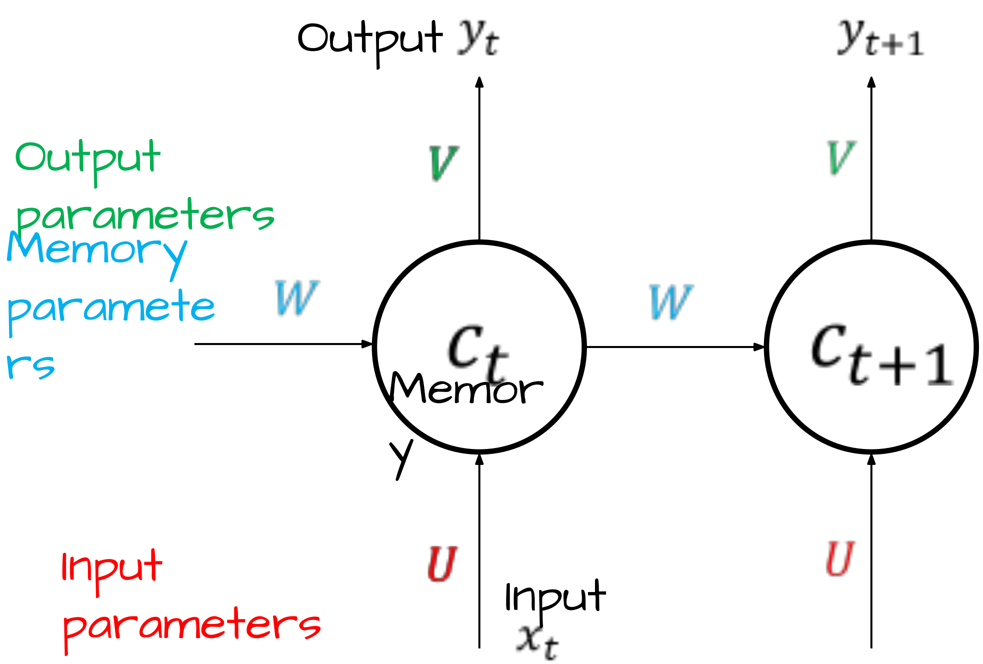
- Input with parameters U
- Memory embedding with parameters W
- Output with parameters V



Recurrent Networks

Simplest model

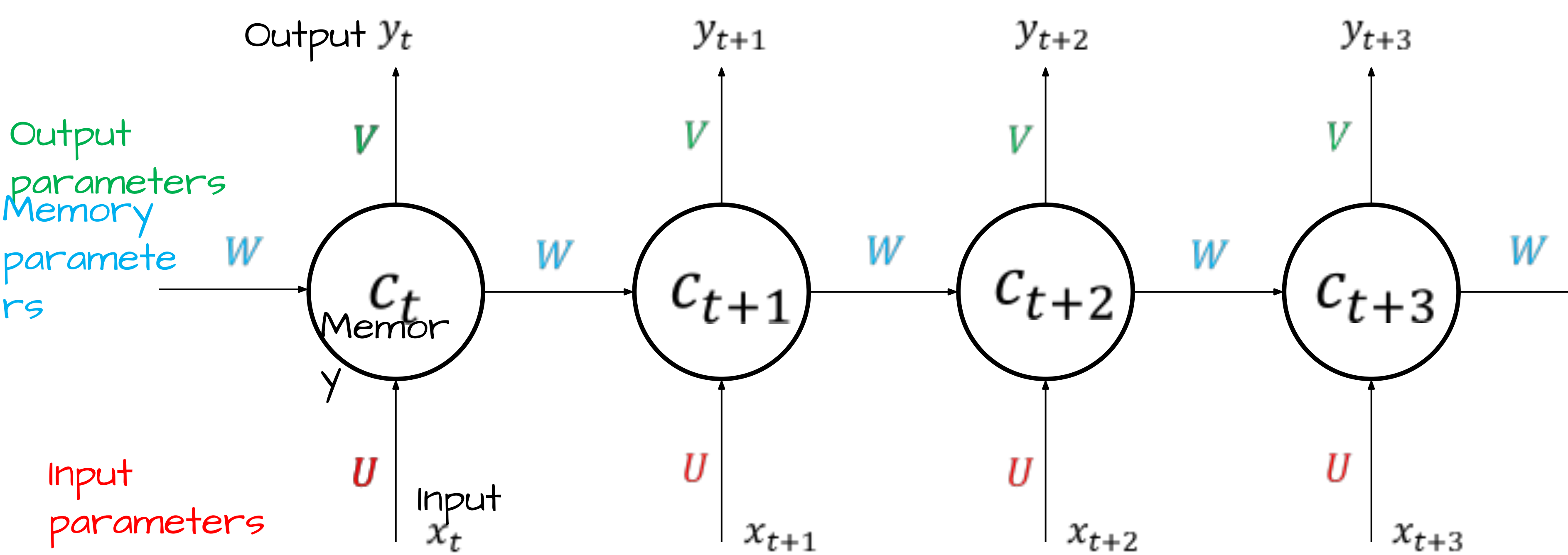
- Input with parameters U
- Memory embedding with parameters W
- Output with parameters V



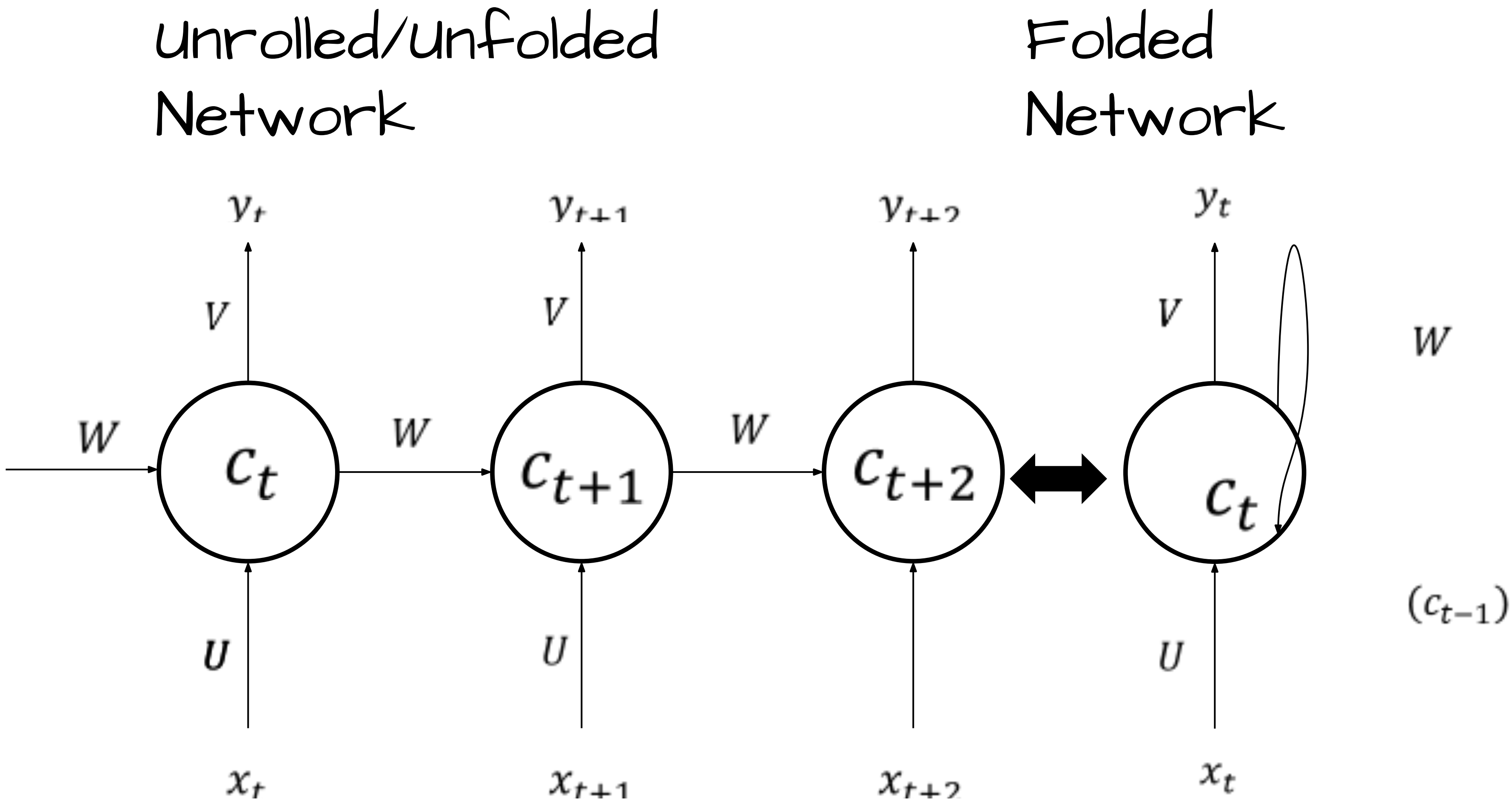
Recurrent Networks

Simplest RNN

- ❑ Input with parameters U
- ❑ Memory embedding with parameters W
- ❑ Output with parameters V



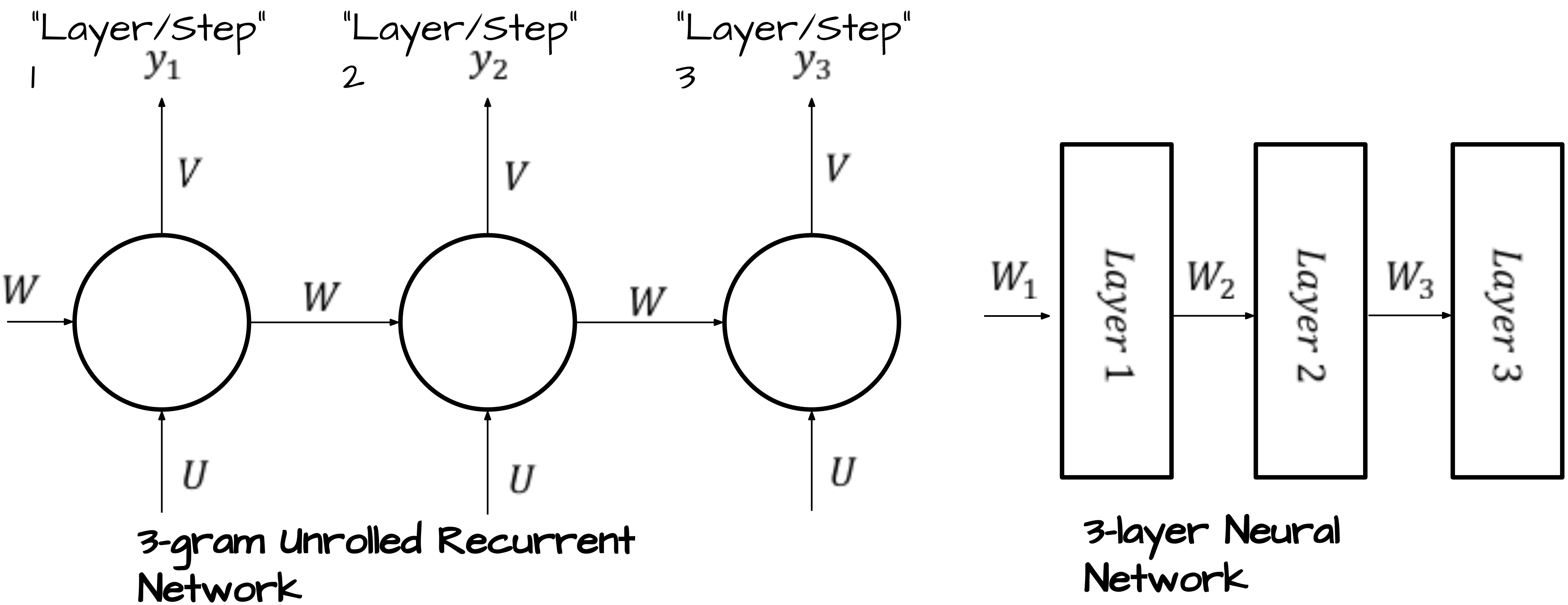
Folding the memory



RNN vs NN

What is really different?

- ❑ Steps instead of layers
- ❑ Step parameters shared whereas in a Multi-Layer Network they are different
- ❑ Input at every layer instead of only at first layer.



Training an RNN

Cross-entropy loss

$$P = \prod_{t,k} y_{tk}^{l_{tk}} \quad \Rightarrow \quad \mathcal{L} = -\log P = \sum_t \mathcal{L}_t = -\frac{1}{T} \sum_t l_t \log y_t$$

Backpropagation Through Time (BPTT)

Be careful of the recursion. The non-linearity is influencing itself. The gradients at one time step depends on gradients on previous time steps

- Like in NN → Chain Rule
- Only difference: Gradients survive over time steps

RNN Gradients

$$\mathcal{L} = L(c_T(c_{T-1}(\ldots(c_1(x_1, c_0; W); W); W); W)$$

$$\frac{\partial \mathcal{L}_t}{\partial W} = \sum_{\tau=1}^{\mathrm{t}} \frac{\partial \mathcal{L}_t}{\partial c_t} \frac{\partial c_t}{\partial c_\tau} \frac{\partial c_\tau}{\partial W}$$

$$\frac{\partial \mathcal{L}}{\partial c_t} \frac{\partial c_t}{\partial c_\tau} = \frac{\partial \mathcal{L}}{\partial \mathrm{c_t}} \cdot \frac{\partial c_t}{\partial \mathrm{c_{t-1}}} \cdot \frac{\partial c_{t-1}}{\partial \mathrm{c_{t-2}}} \cdot \ldots \cdot \frac{\partial c_{\tau+1}}{\partial \mathrm{c_\tau}} \leq \eta^{t-\tau} \frac{\partial \mathcal{L}_t}{\partial c_t}$$

The RNN gradient is a recursive product of $\frac{\partial c_t}{\partial \mathrm{c_{t-1}}}$

Vanishing/Exploding gradients

$$\frac{\partial \mathcal{L}}{\partial c_t} = \underset{<1}{\frac{\partial \mathcal{L}}{\partial \mathrm{c_T}}} \cdot \underset{<1}{\frac{\partial c_T}{\partial \mathrm{c_{T-1}}}} \cdot \frac{\partial c_{T-1}}{\partial \mathrm{c_{T-2}}} \cdot \ldots \cdot \underset{<1}{\frac{\partial c_{t+1}}{\partial \mathrm{c}_{c_t}}}$$

$\frac{\partial \mathcal{L}}{\partial W} \ll 1 \Rightarrow$ Vanishing gradient

$$\frac{\partial \mathcal{L}}{\partial c_t} = \underset{>1}{\frac{\partial \mathcal{L}}{\partial \mathrm{c_T}}} \cdot \underset{>1}{\frac{\partial c_T}{\partial \mathrm{c_{T-1}}}} \cdot \frac{\partial c_{T-1}}{\partial \mathrm{c_{T-2}}} \cdot \ldots \cdot \underset{>1}{\frac{\partial c_1}{\partial \mathrm{c}_{c_t}}}$$

$\frac{\partial \mathcal{L}}{\partial W} \gg 1 \Rightarrow$ Exploding gradient

Advanced RNN: LSTM

$\sigma \in (0,1)$: control gate – something like a switch

$\tanh \in (-1,1)$: recurrent nonlinearity

$$i = \sigma\left(x_t U^{(i)} + m_{t-1} W^{(i)}\right)$$

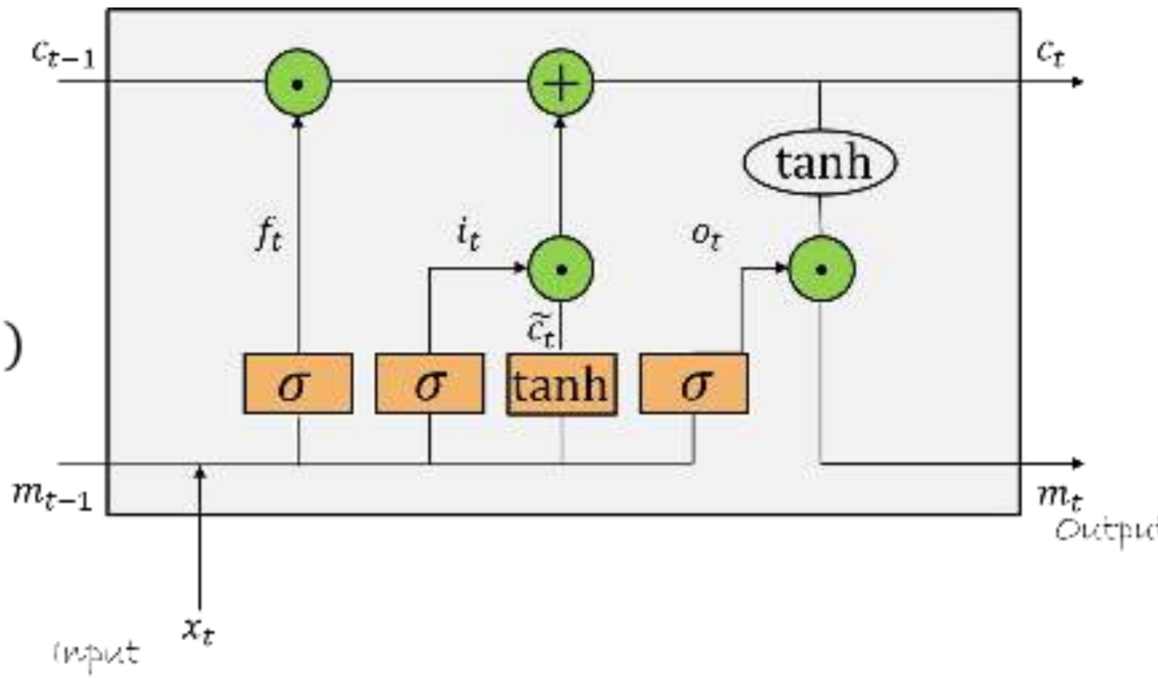
$$f = \sigma\left(x_t U^{(f)} + m_{t-1} W^{(f)}\right)$$

$$o = \sigma\left(x_t U^{(o)} + m_{t-1} W^{(o)}\right)$$

$$\widetilde{c}_t = \tanh(x_t U^{(g)} + m_{t-1} W^{(g)})$$

$$c_t = c_{t-1} \odot f + \widetilde{c}_t \odot i$$

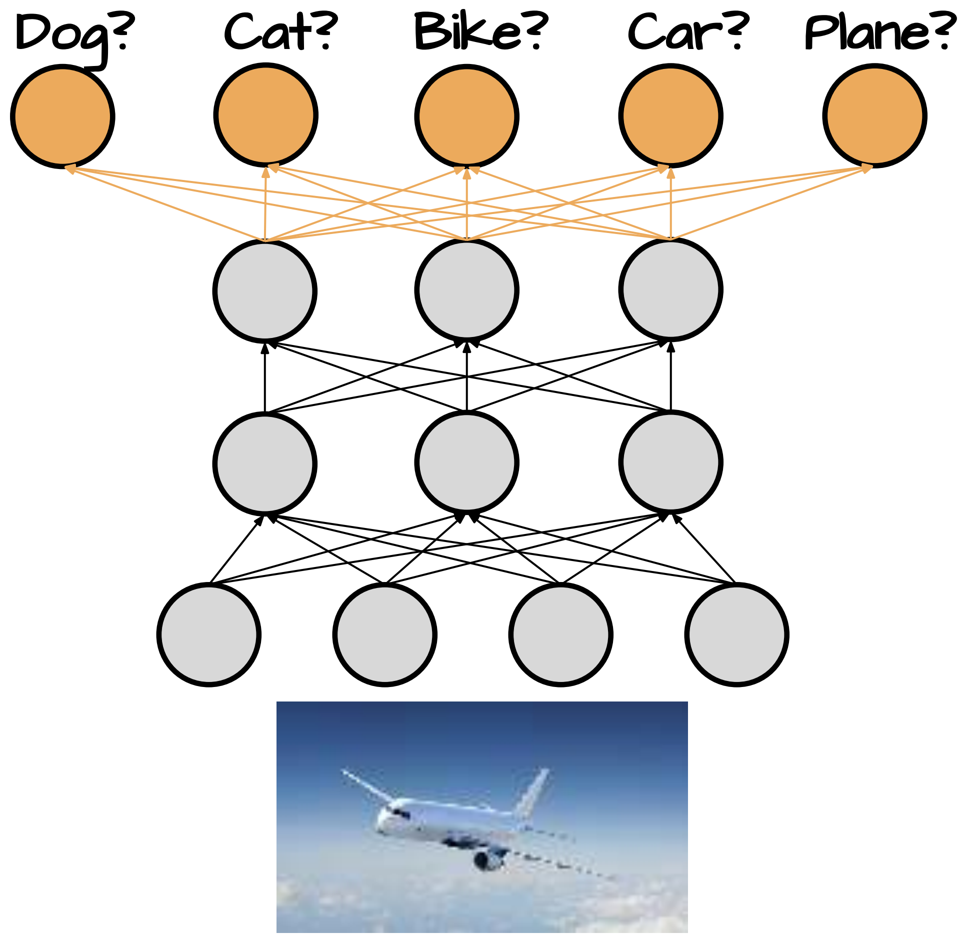
$$m_t = \tanh(c_t) \odot o$$



Bringing Structure to Visual Deep Learning

Standard inference

N-way classification

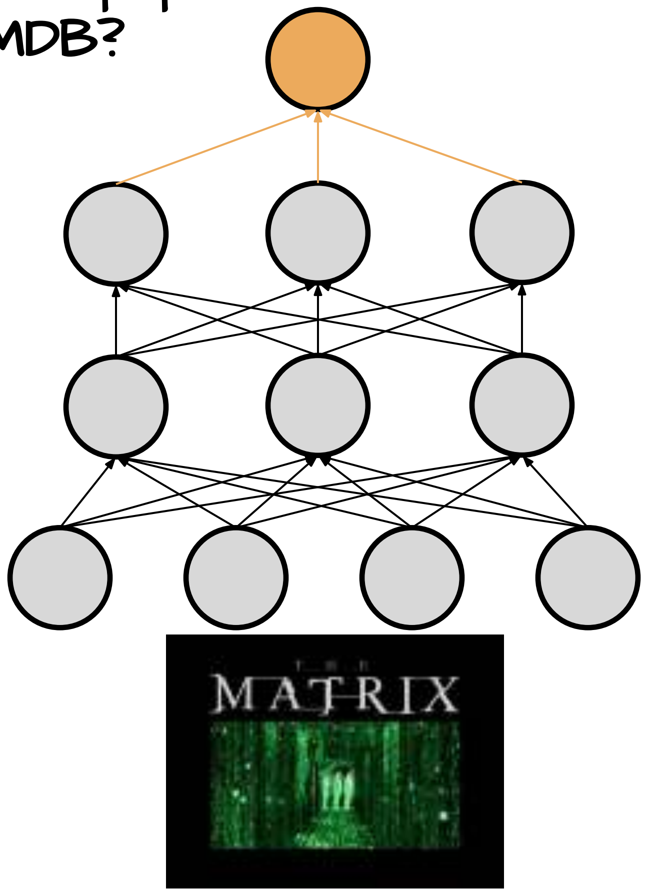


Standard inference

N-way classification

Regression

How popular will this movie be in IMDB?



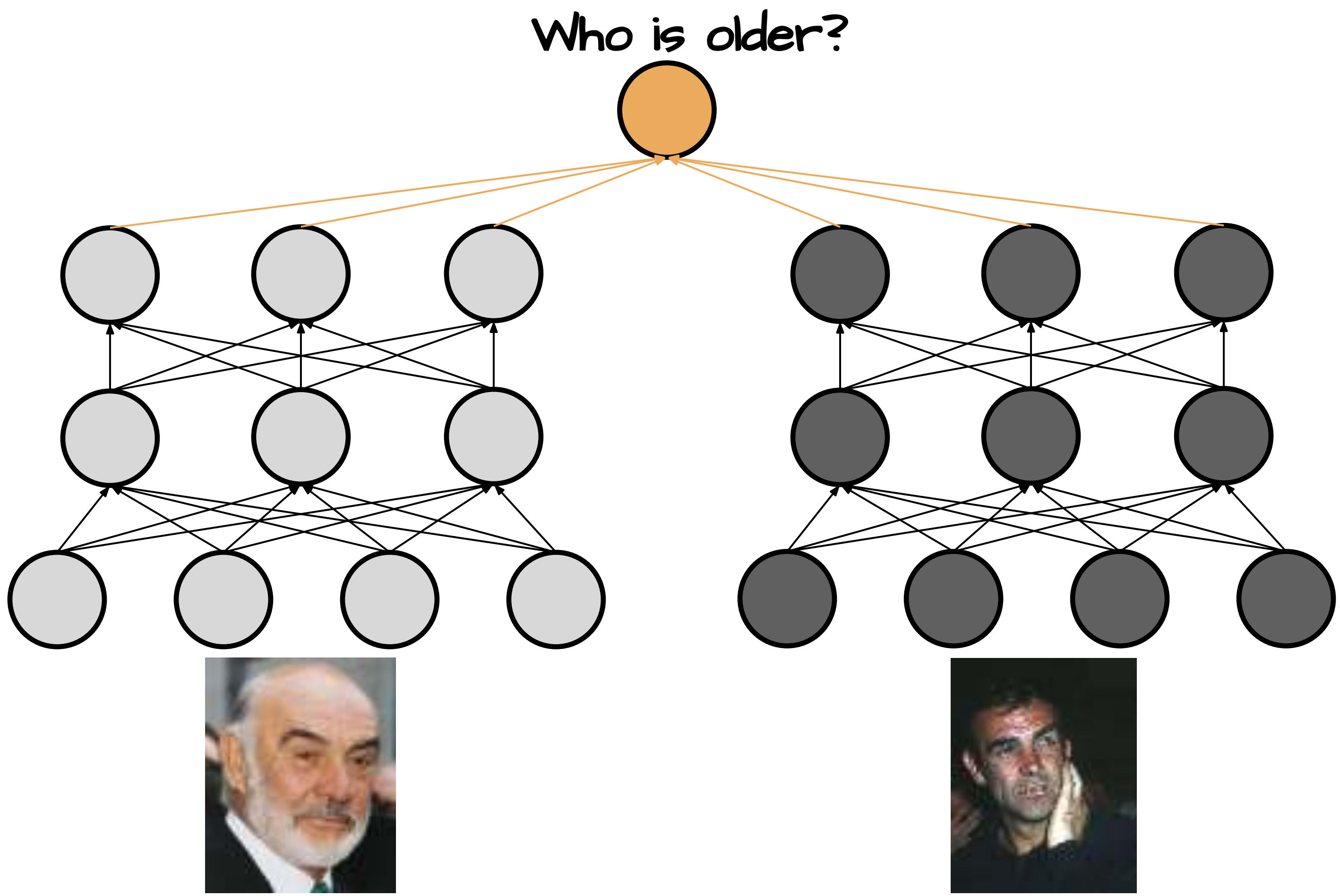
Standard inference

N-way classification

Regression

Ranking

…



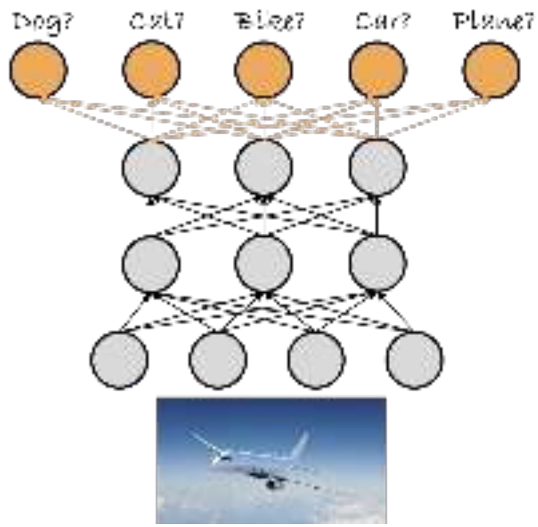
Quiz: What is common?

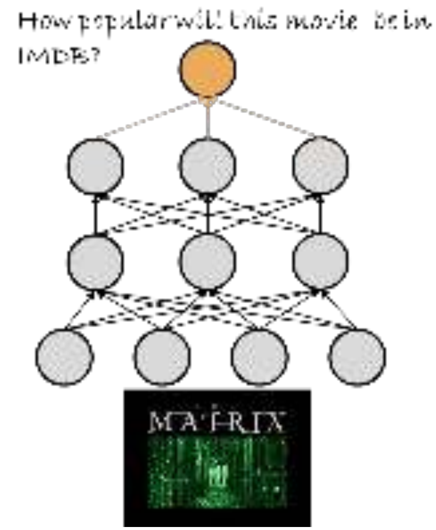
N-way classification

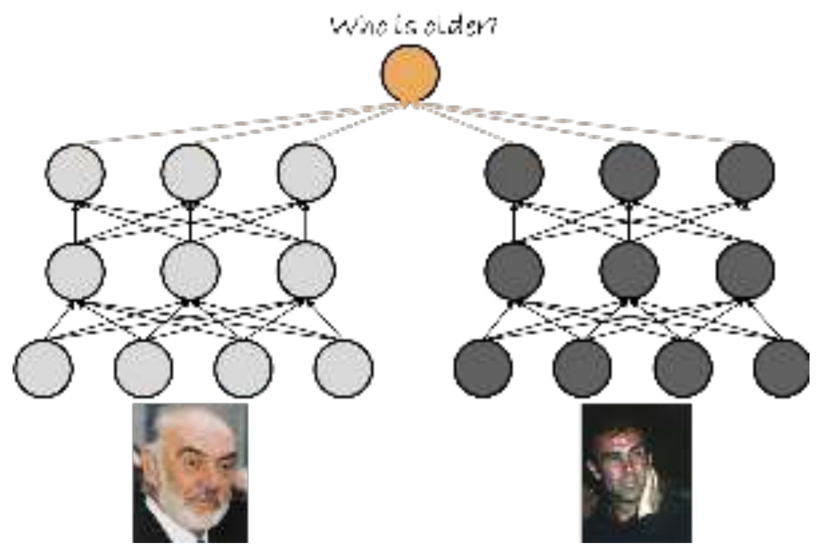
Regression

Ranking

…



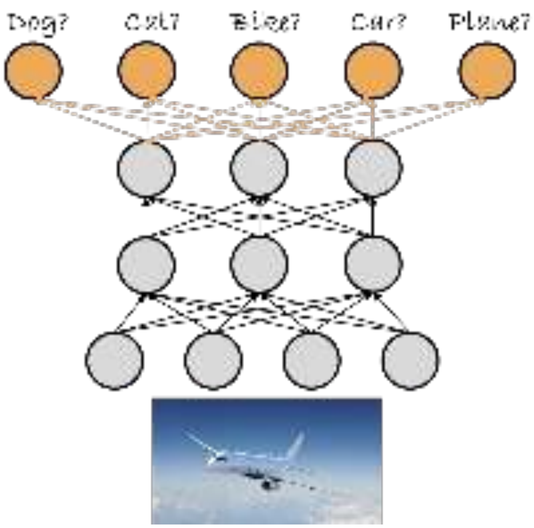


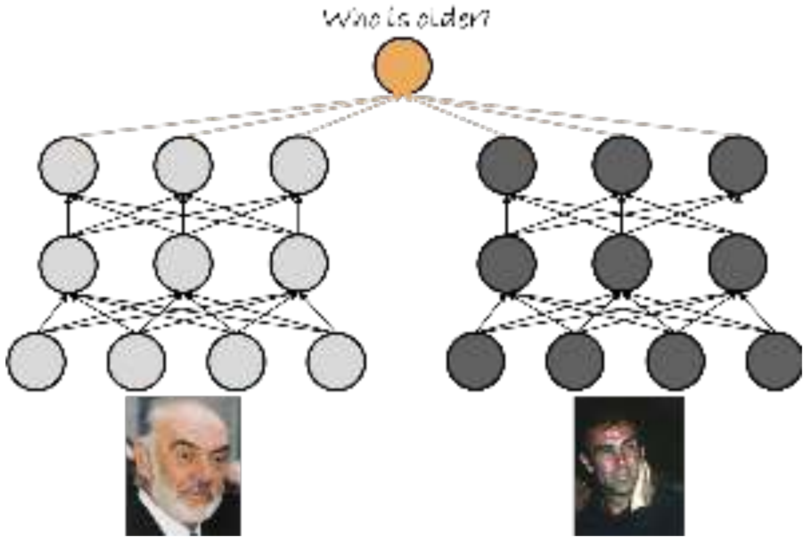


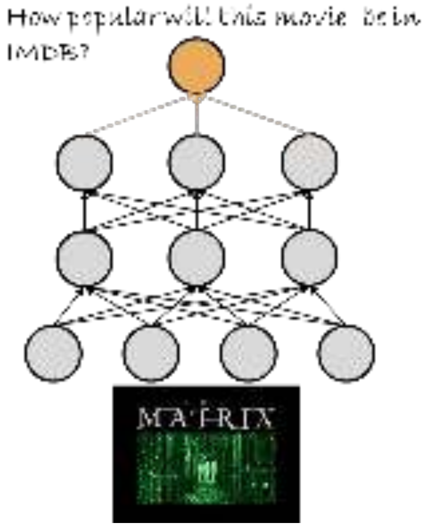
Quiz: What is common?

They all make “single value” predictions

Do all our machine learning tasks
 boil down to “single value” predictions?

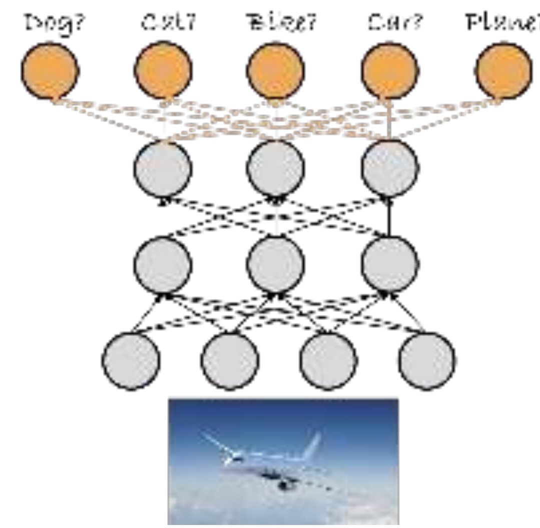






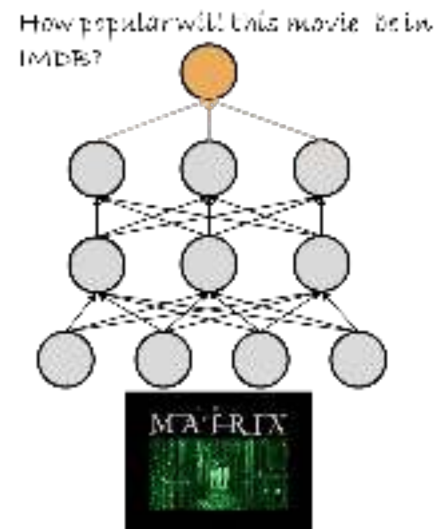
Beyond “single value” predictions?

Do all our machine learning tasks boil to “single value” predictions?



Are there tasks where outputs are somehow correlated?

Is there some structure in these output correlations?



How can we predict such structures?

- Structured prediction

Object detection

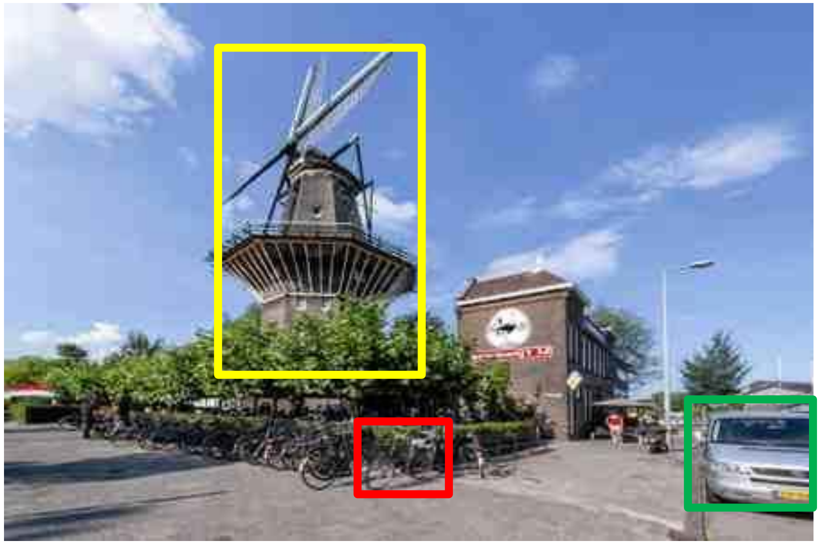
Predict a box around an object

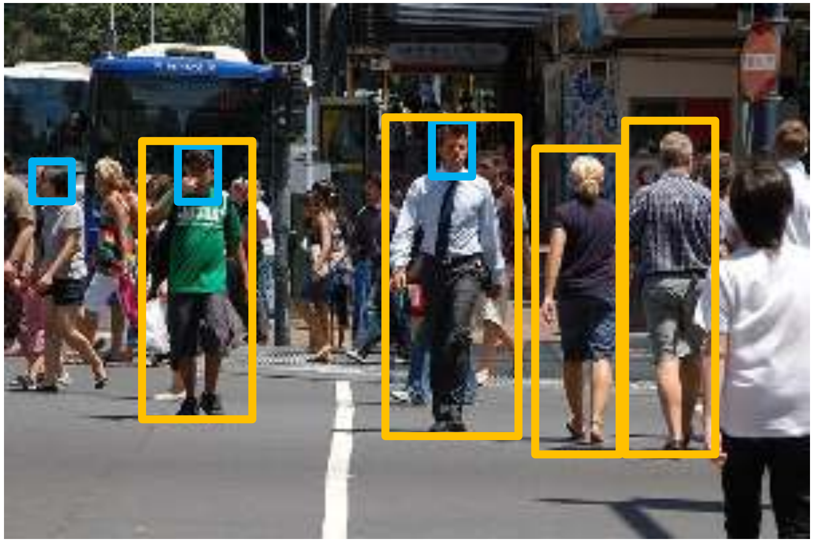
Images

- ❑ Spatial location
- ❑ bounding box (bbox)

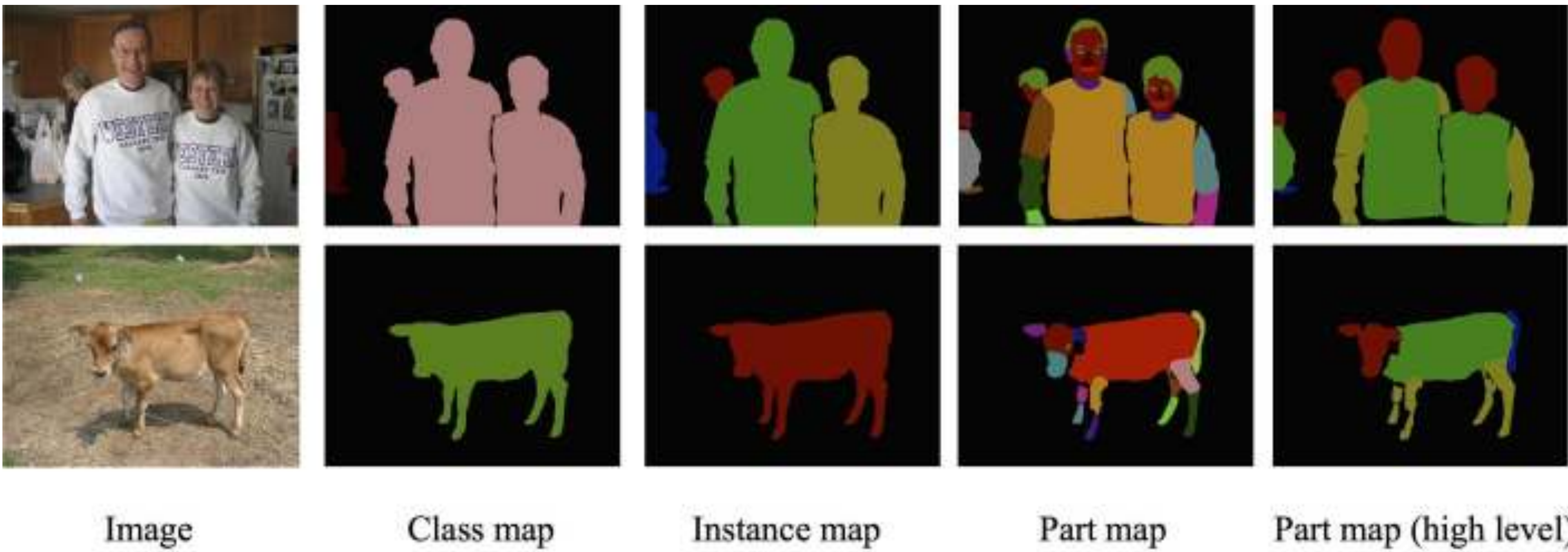
Videos

- ❑ Spatio-temporal location
- ❑ bbox@t, bbox@t+1, …

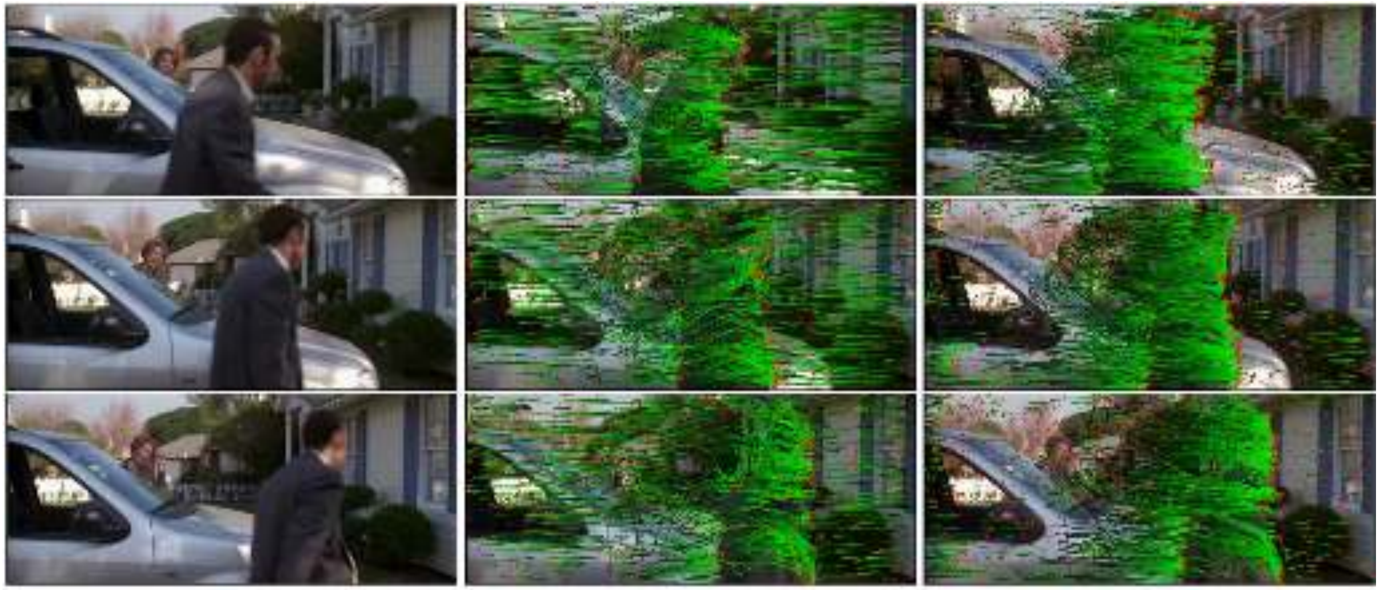




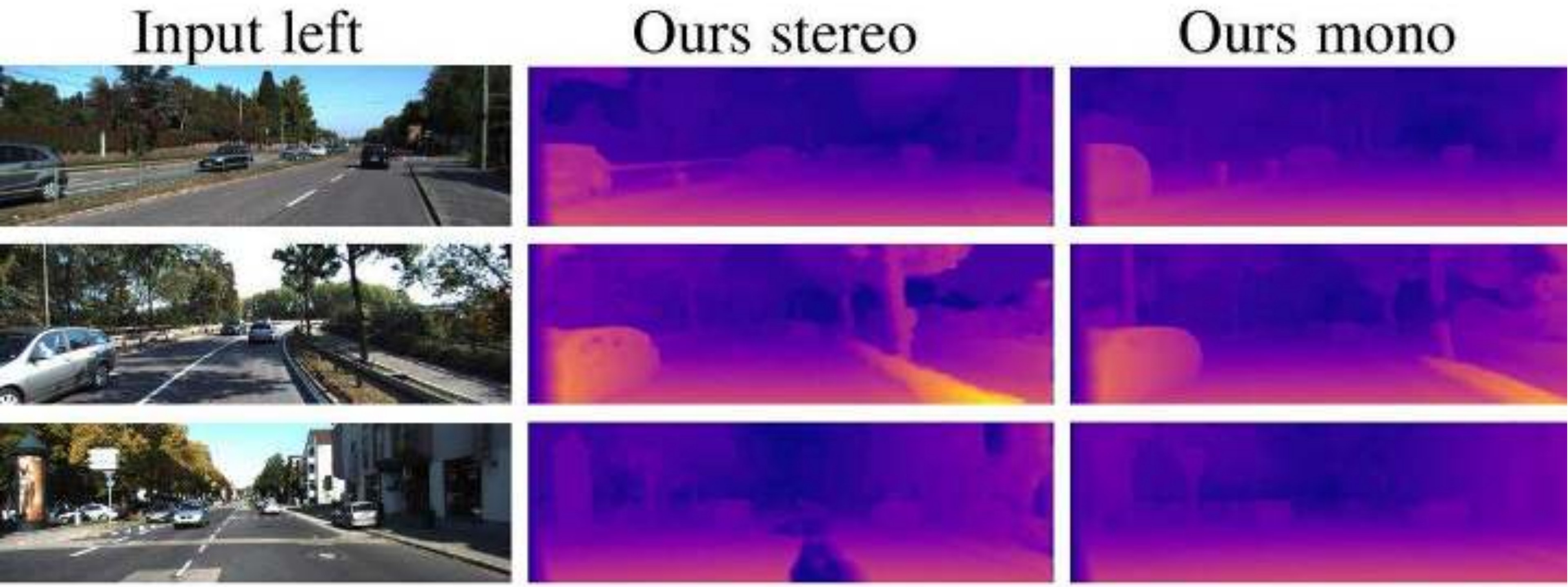
Object segmentation



Optical flow & motion estimation



Depth estimation



Godard et al., Unsupervised Monocular Depth Estimation with Left-Right Consistency, 2016

Structured prediction

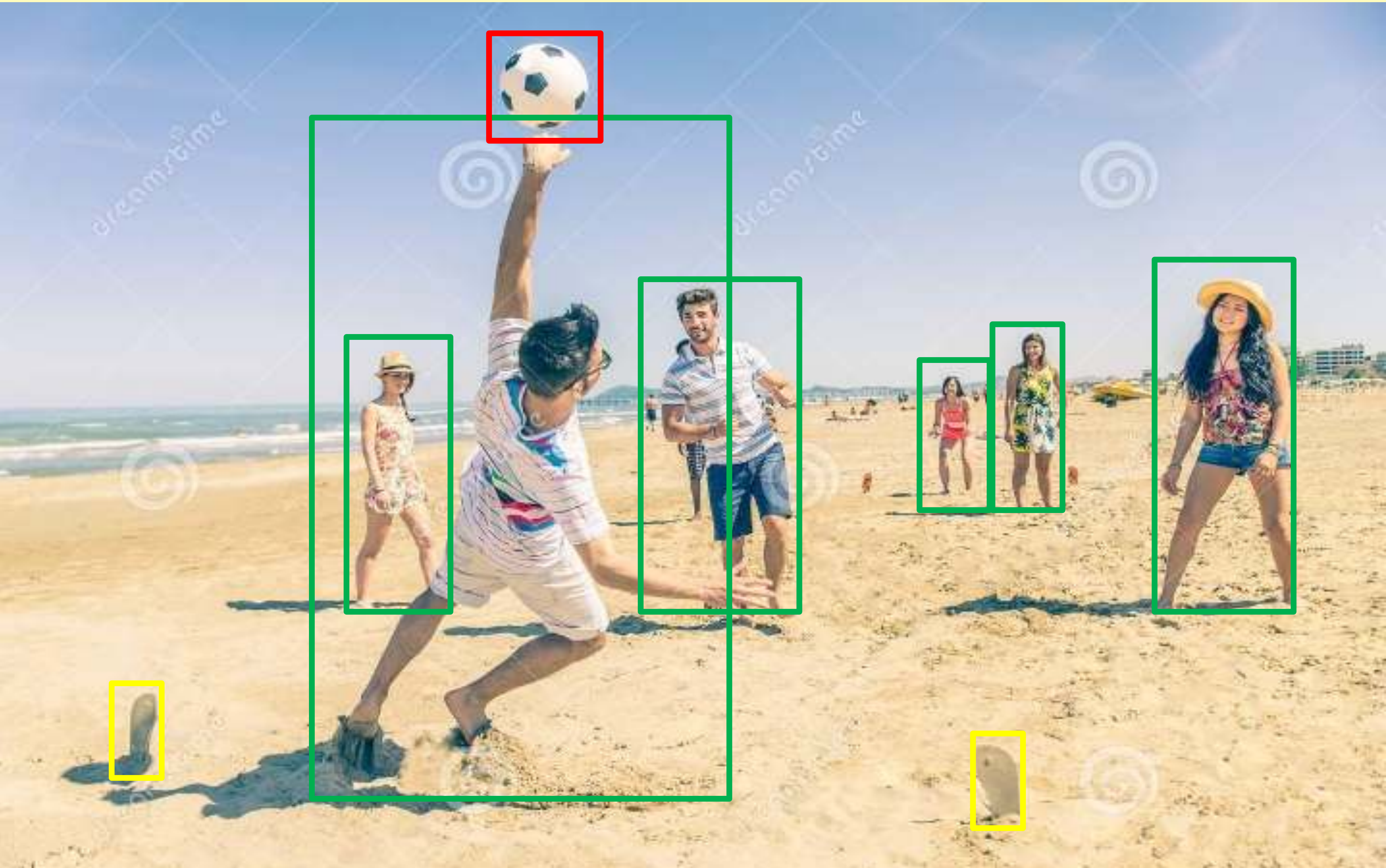
Prediction goes beyond asking for “single values”

Outputs are complex and output dimensions correlated

Output dimensions have latent structure

Can we make deep networks to return <u>**structured predictions?**</u>

Convnets for structured prediction

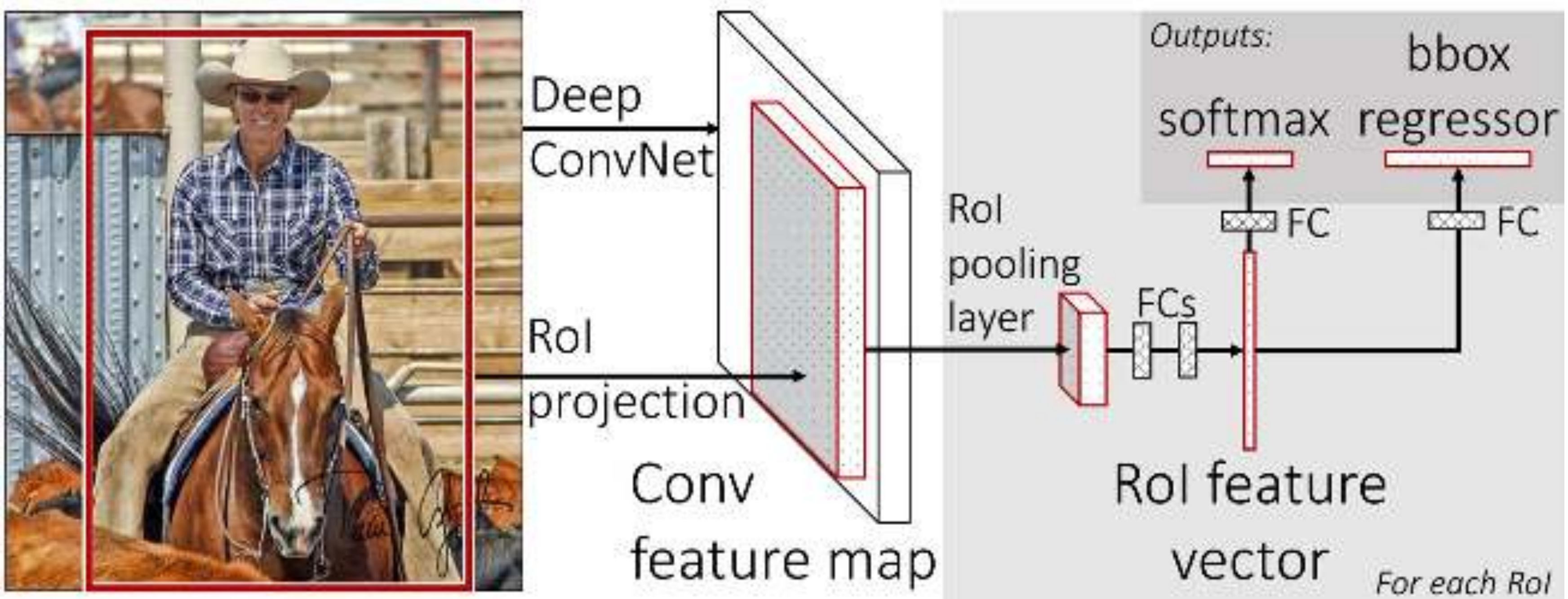


Sliding window on feature maps

Selective Search Object Proposals [Uijlings2013]

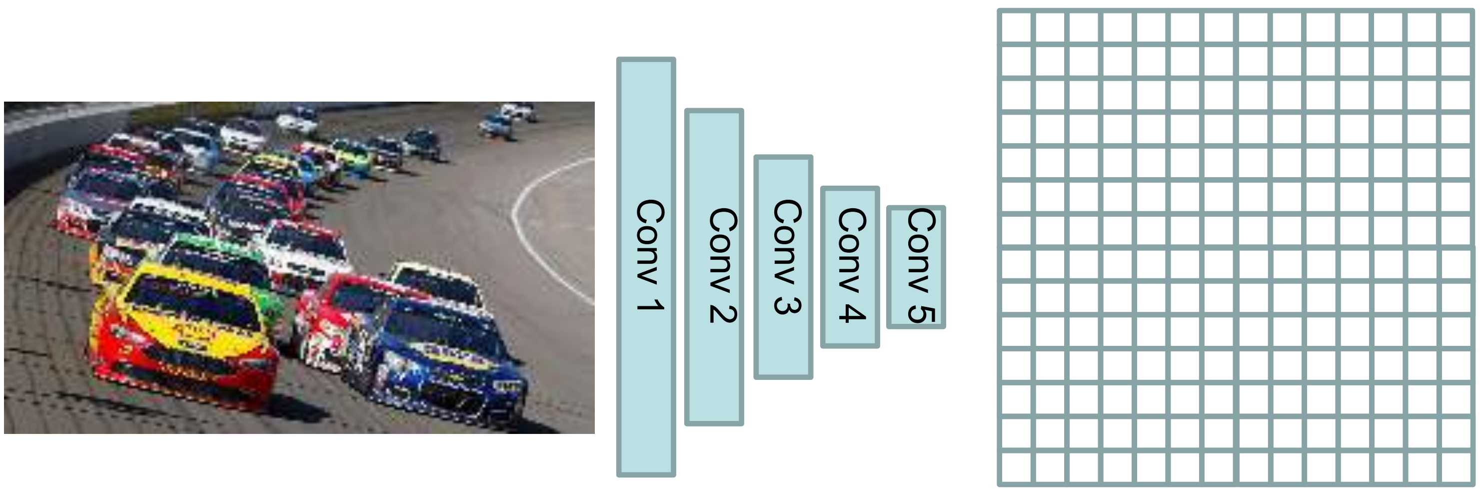
SPPnet [He2014]

Fast R-CNN [Girshick2015]



Fast R-CNN: Steps

Fast R-CNN [Girshick2015]

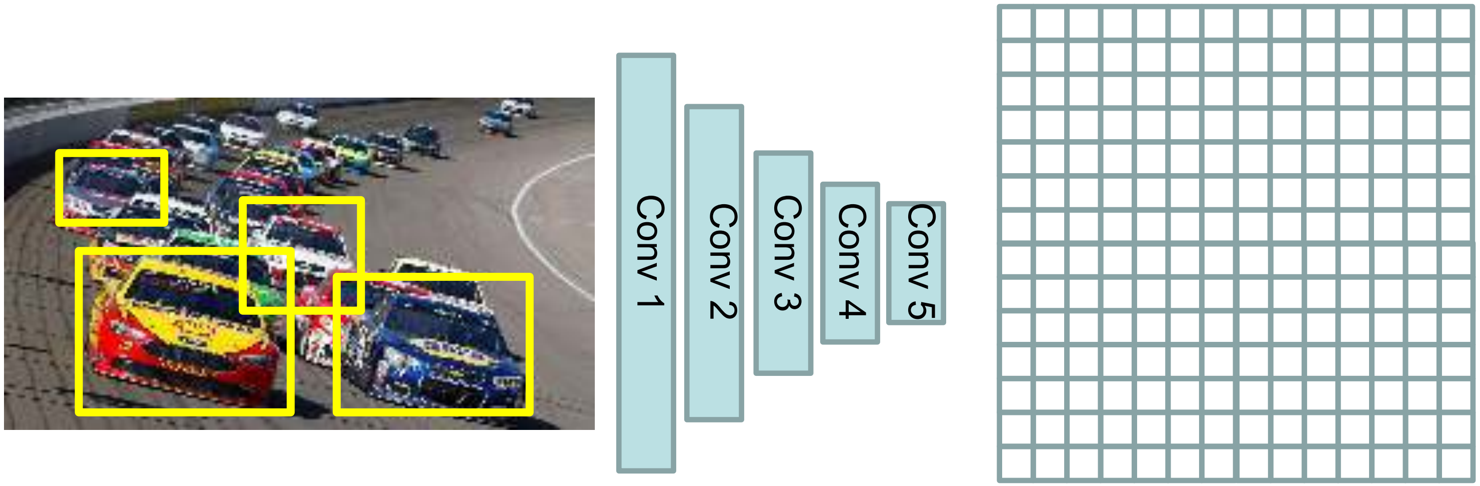


Conv 5 feature map

Fast R-CNN: Steps

Process the whole image up to conv5

Compute possible locations for objects



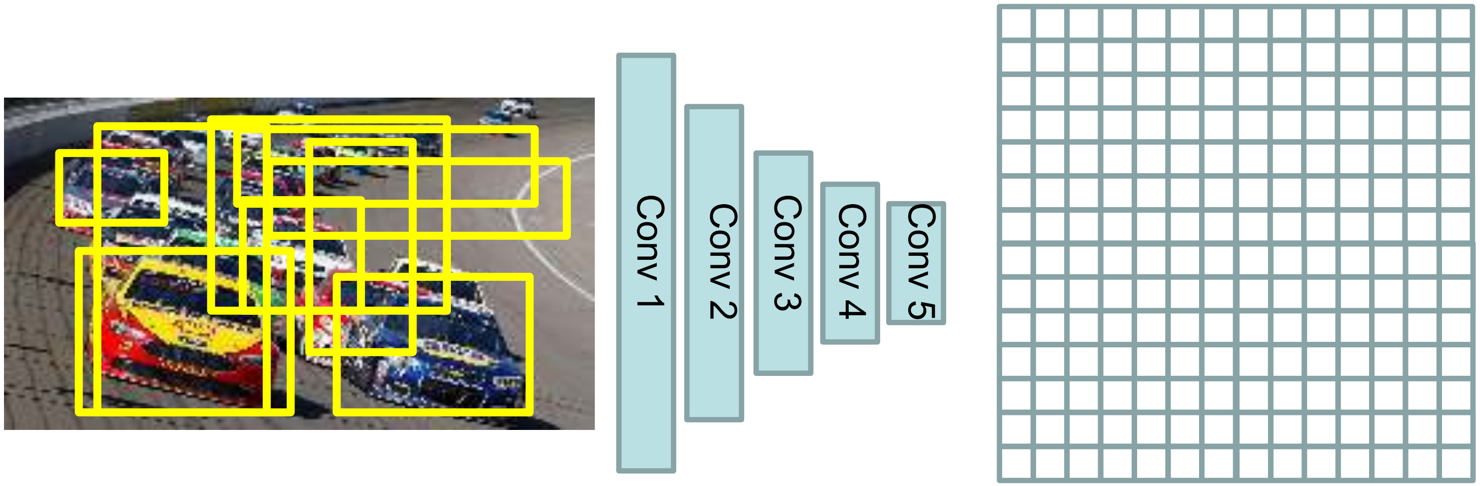
Conv 5 feature map

Fast R-CNN: Steps

Process the whole image up to conv5

Compute possible locations for objects

- some correct, most wrong



Conv 5 feature map

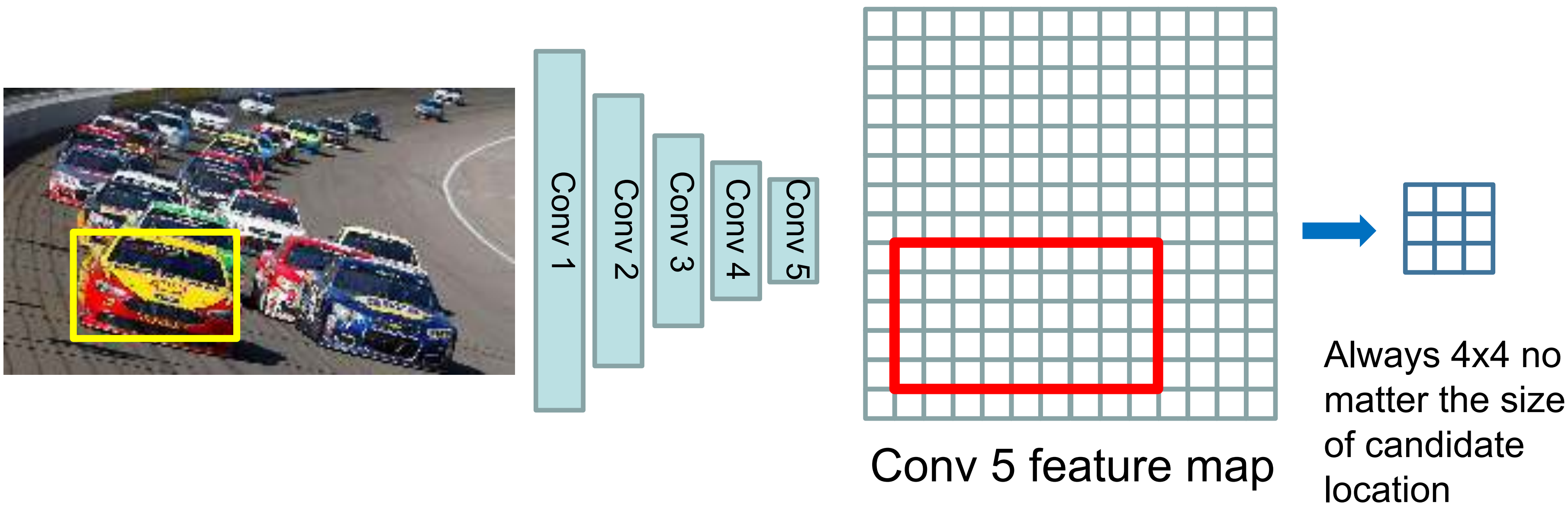
Fast R-CNN: Steps

Process the whole image up to conv5

Compute possible locations for objects

- some correct, most wrong

Given single location → ROI pooling module extracts fixed length feature



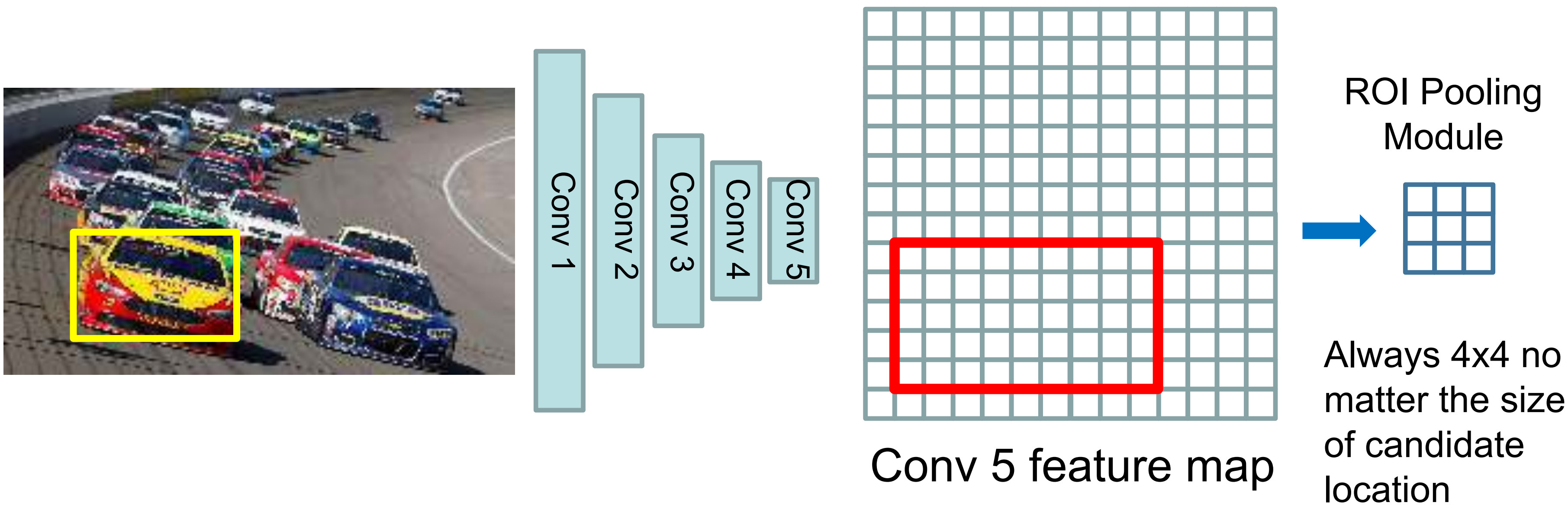
Fast R-CNN: Steps

Process the whole image up to conv5

Compute possible locations for objects

- some correct, most wrong

Given single location → ROI pooling module extracts fixed length feature



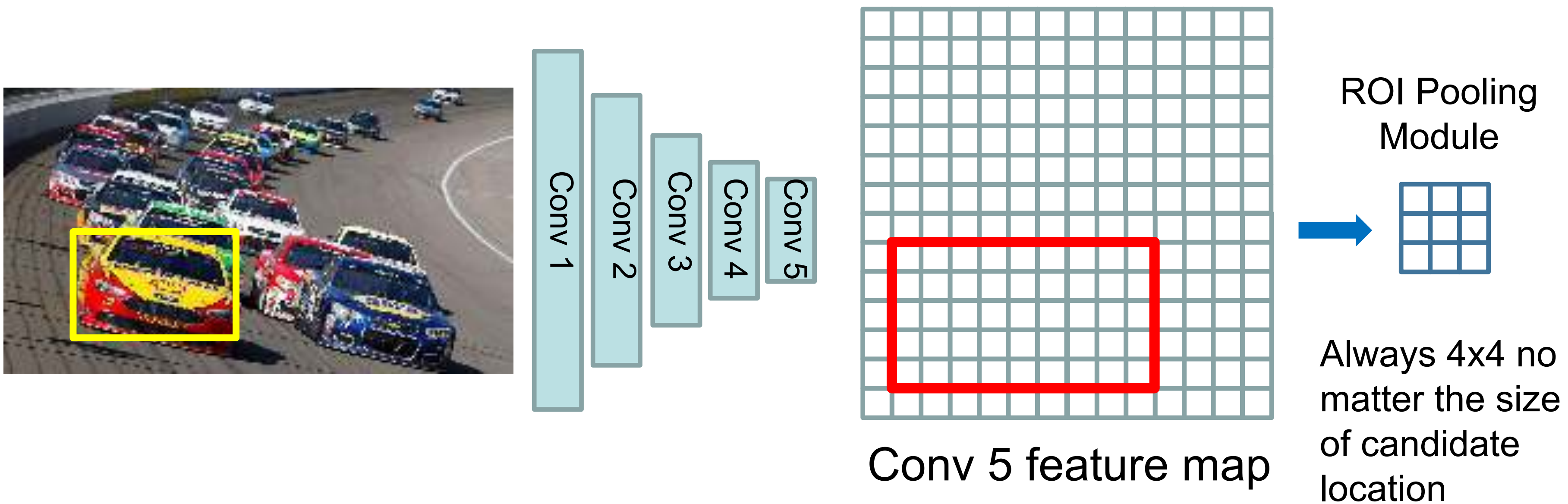
Fast R-CNN: Steps

Process the whole image up to conv5

Compute possible locations for objects

- some correct, most wrong

Given single location → ROI pooling module extracts fixed length feature



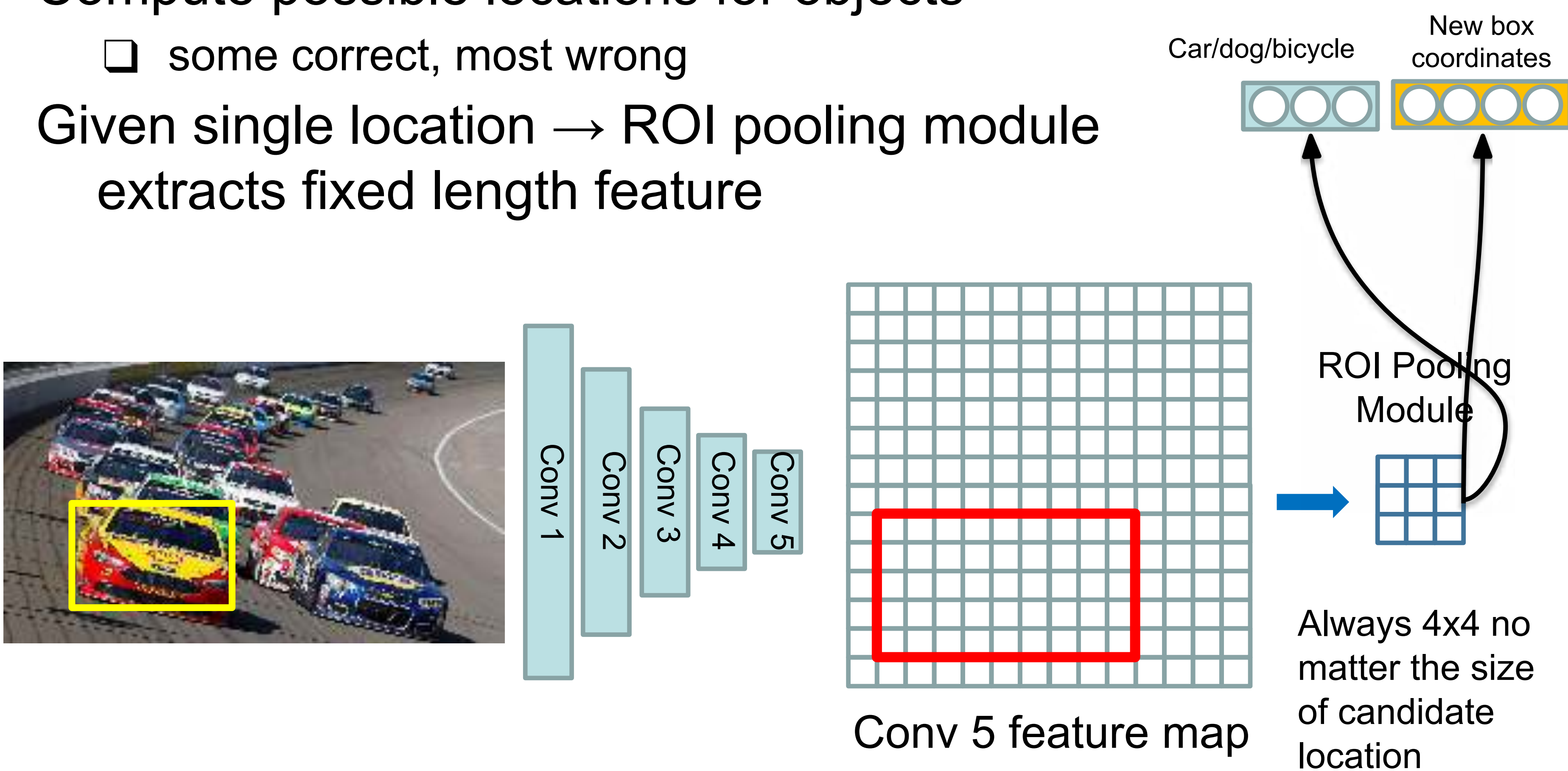
Fast R-CNN: Steps

Process the whole image up to conv5

Compute possible locations for objects

- some correct, most wrong

Given single location → ROI pooling module extracts fixed length feature



Some results

Fast R-CNN

Reuse convolutions for different candidate boxes

- ❑ Compute feature maps only once

Region-of-Interest pooling

- ❑ Define stride relatively $\rightarrow$ box width divided by predefined number of “poolings” T
- ❑ Fixed length vector

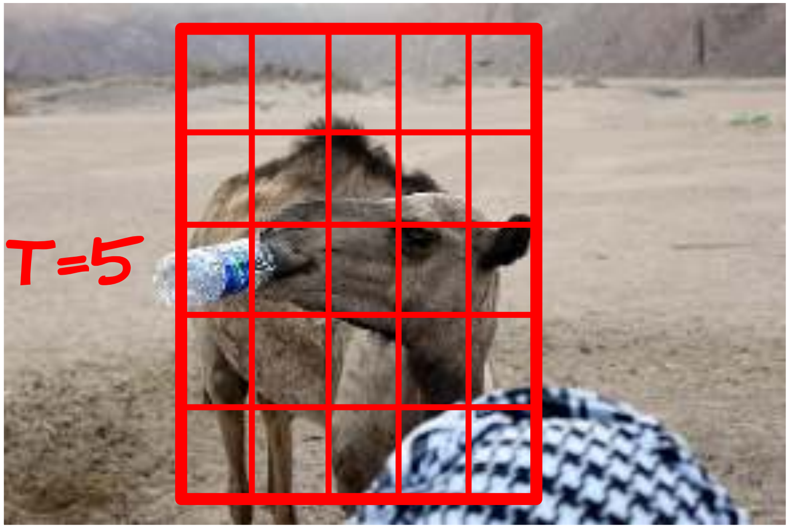
End-to-end training!

(Very) Accurate object detection

(Very) Faster

- ❑ Less than a second per image

External box proposals needed

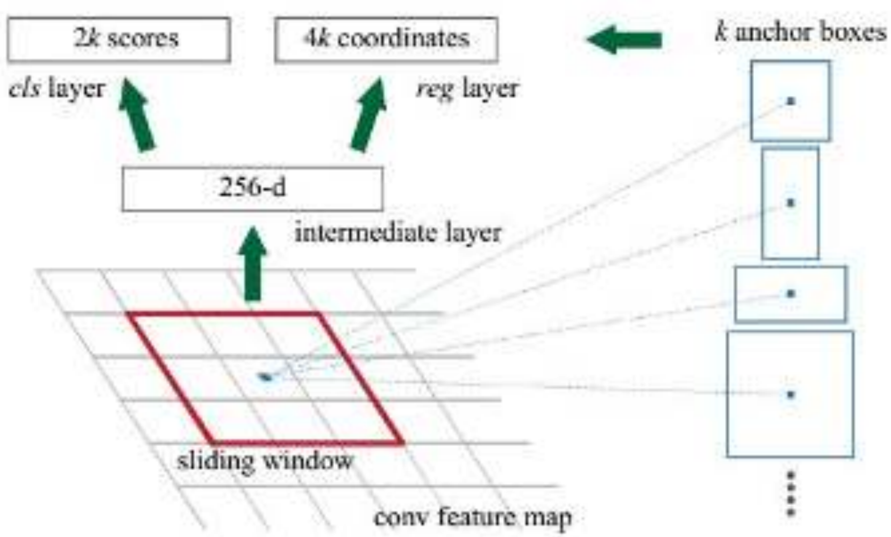


Faster R-CNN [Girshick2016]

Fast R-CNN
- ❑ external candidate locations

Faster R-CNN
- ❑ deep network proposes candidate

Slide the feature map
- ❑ k anchor boxes per slide



Region Proposal Network

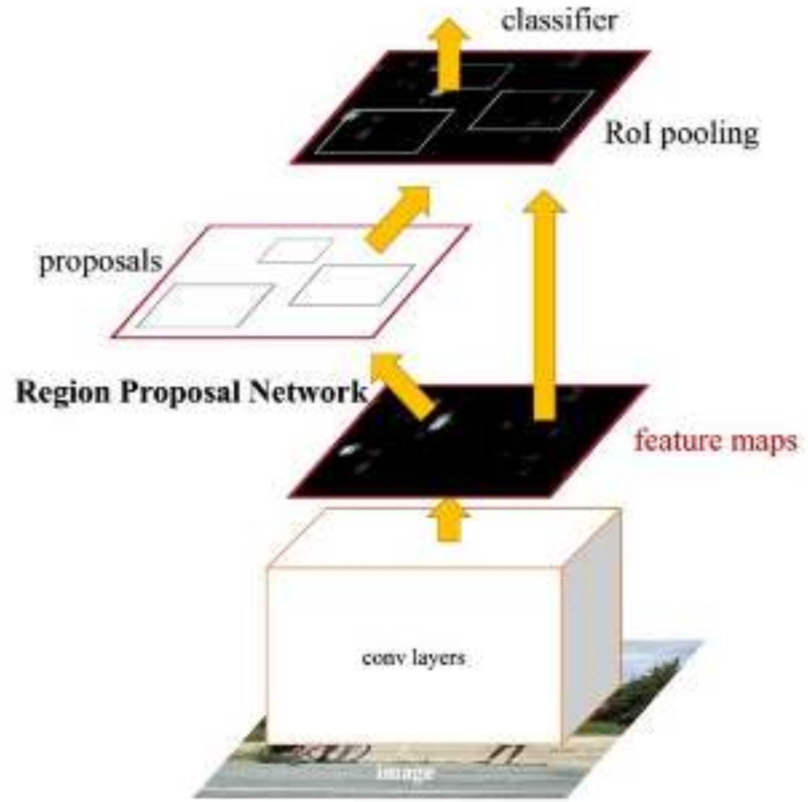


Figure 2: Faster R-CNN is a single, unified network for object detection. The RPN module serves as the 'attention' of this unified network.

Image Segmentation: Fully Convolutional

[LongCVPR2014]

Image larger than network input

- slide the network

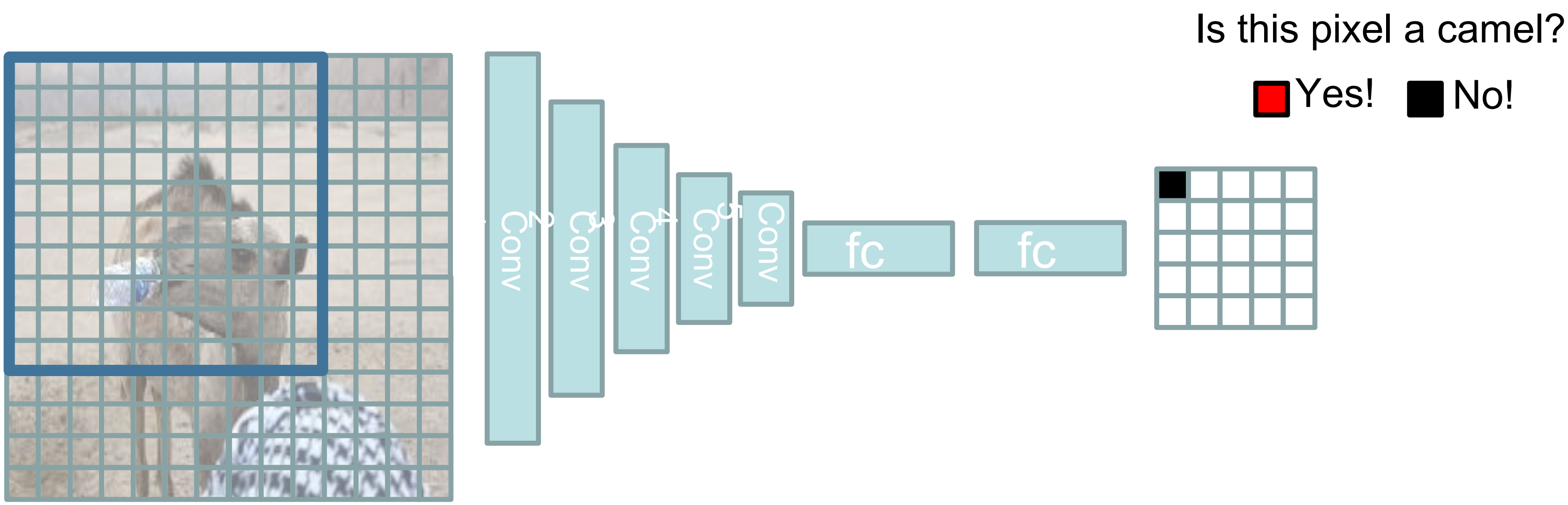


Image Segmentation: Fully Convolutional

[LongCVPR2014]

Image larger than network input

- slide the network

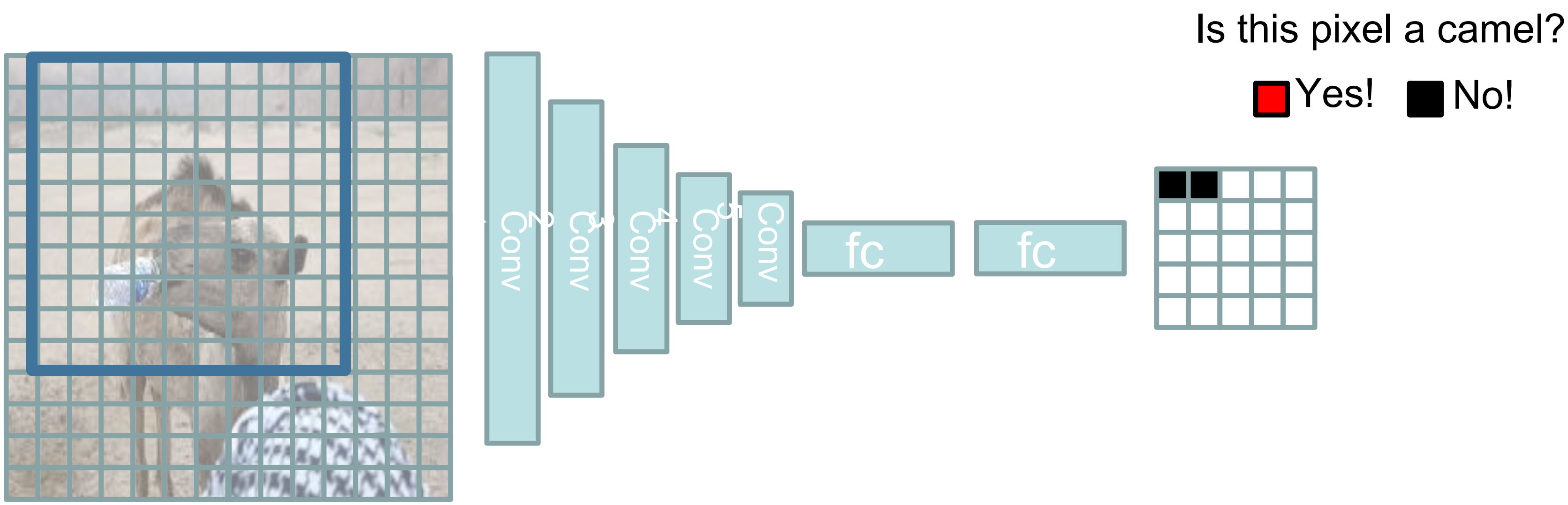


Image Segmentation: Fully Convolutional

[LongCVPR2014]

Image larger than network input

- slide the network

Is this pixel a camel?

Yes! No!

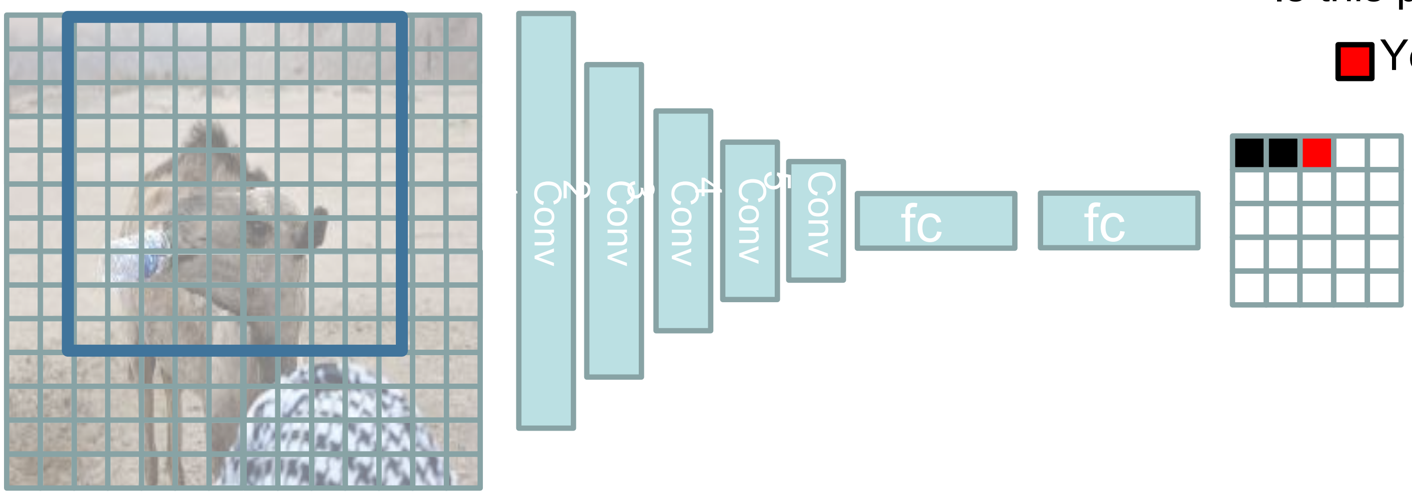


Image Segmentation: Fully Convolutional

[LongCVPR2014]

Image larger than network input

- slide the network

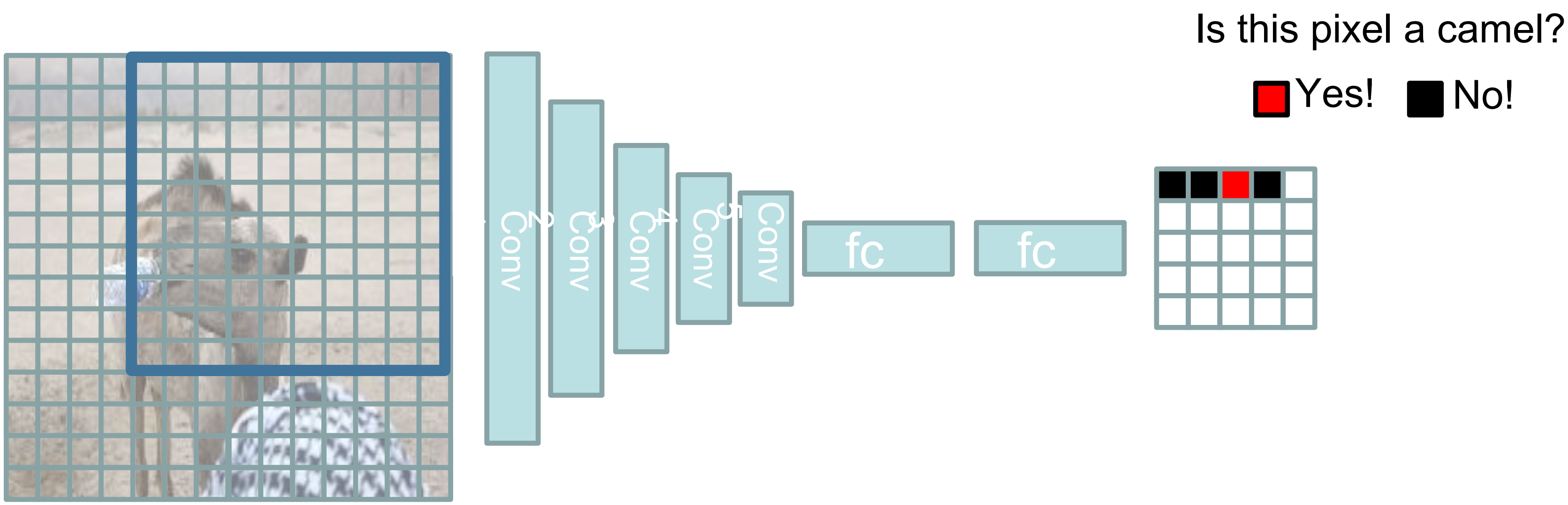
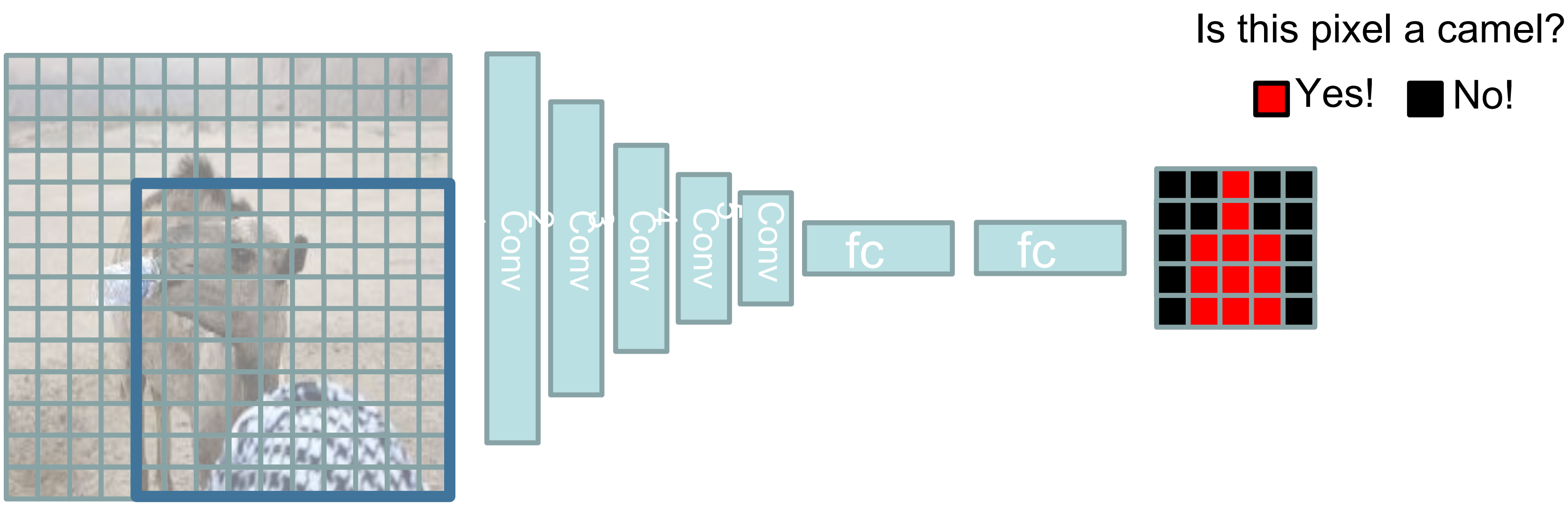


Image Segmentation: Fully Convolutional

[LongCVPR2014]

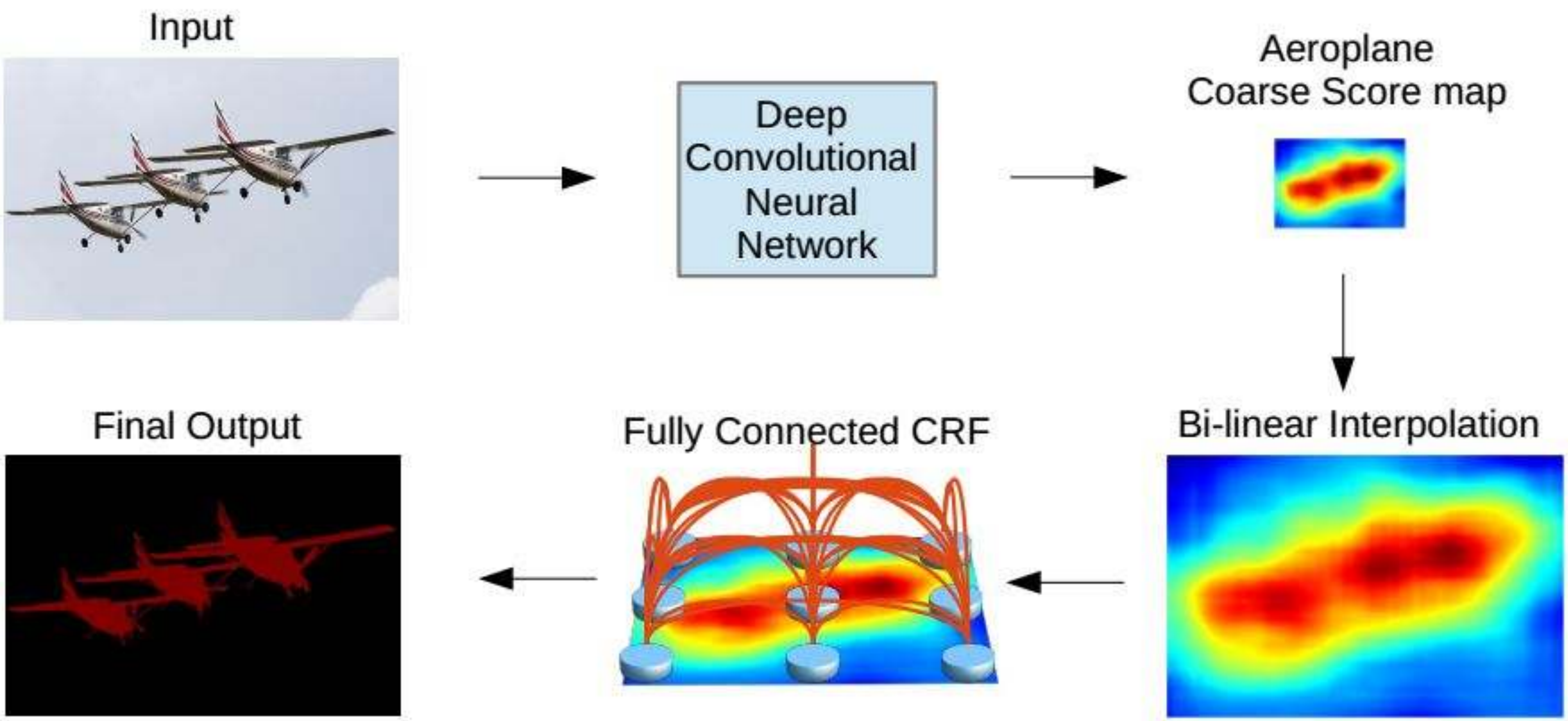
Image larger than network input

- slide the network



Deep ConvNets with CRF loss

[Chen, Papandreou 2016]



Deep ConvNets with CRF loss

[Chen, Papandreou 2016]

Segmentation map is good but not pixel-precise

- Details around boundaries are lost

Cast fully convolutional outputs as unary potentials

Consider pairwise potentials between output dimensions

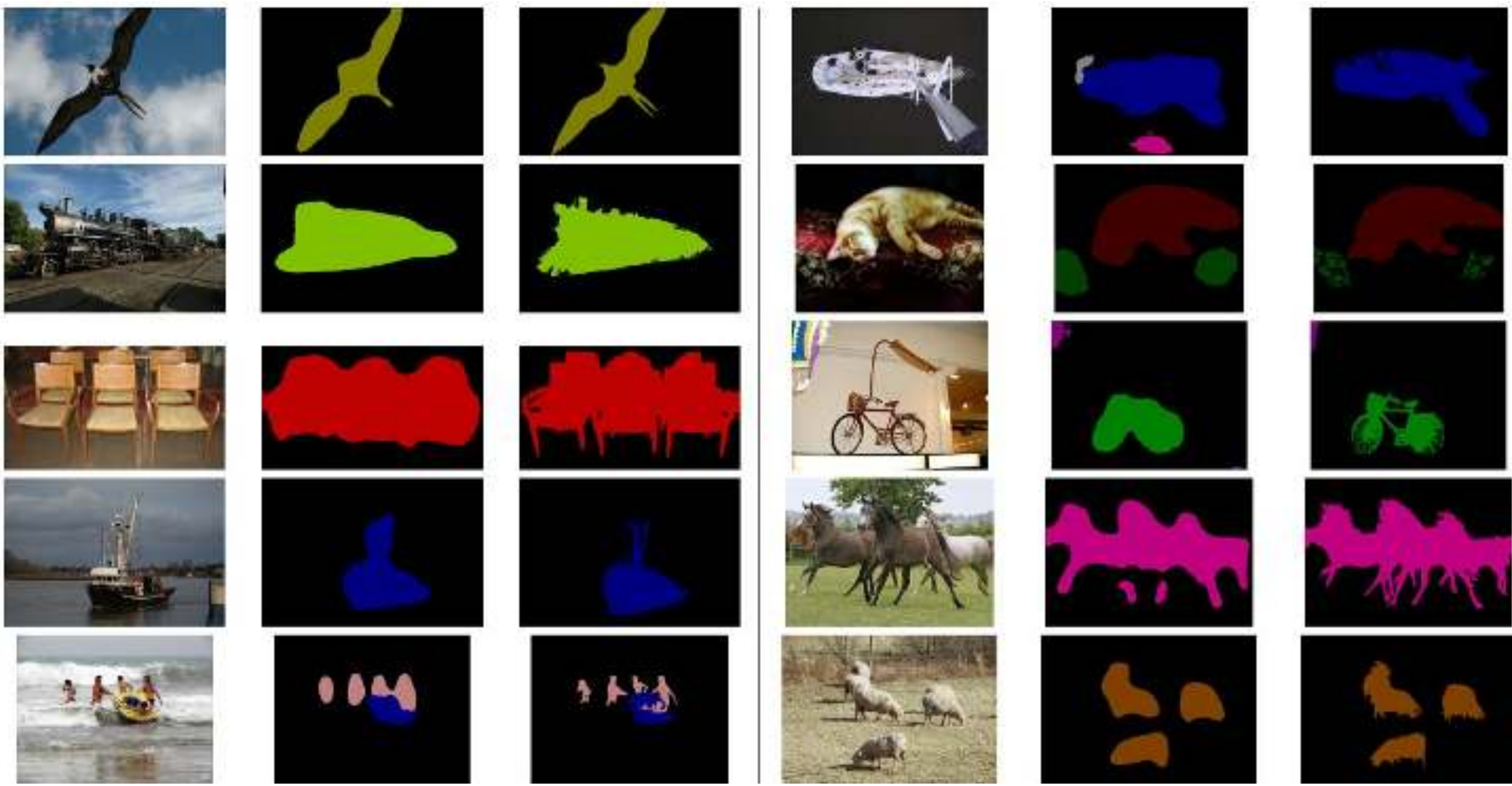
Include Fully Connected CRF loss to refine segmentation

$$E(x) = \sum \theta_i(x_i) + \sum \theta_{ij}(x_i, x_j)$$

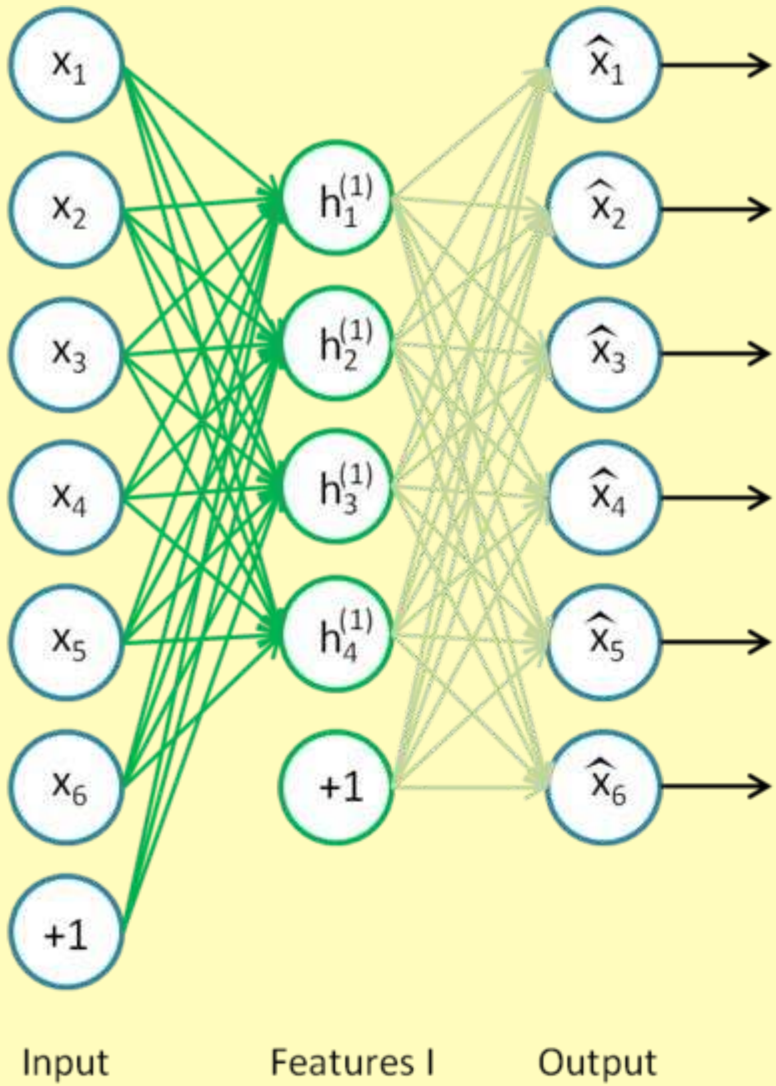
Total loss　Unary loss　Pairwise loss

$$\theta_{ij}(x_i, x_j) \sim w_1 \exp\left(-\alpha |p_i - p_j|^2 - \beta |I_i - I_j|^2\right) + w_2 \exp(-\gamma |p_i - p_j|^2)$$

Examples



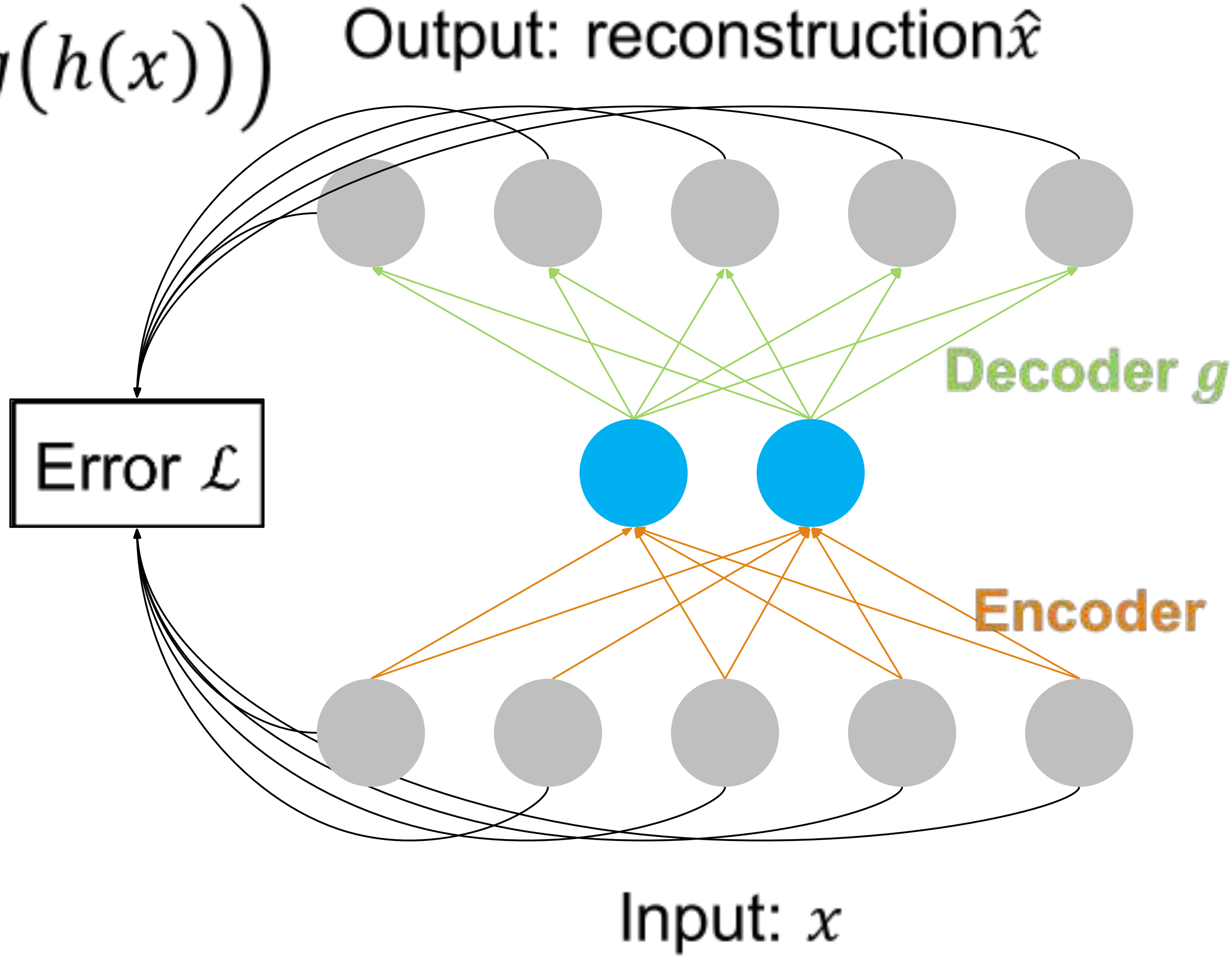
Discovering structure



Standard Autoencoder

$$\mathcal{L} = \sum_{d} \ell\left(x, g(h(x))\right)$$

$$\ell = |x - \hat{x}|^2$$



Standard Autoencoder

The latent space should have fewer dimensions than input

- **Undercomplete** representation
- Bottleneck architecture

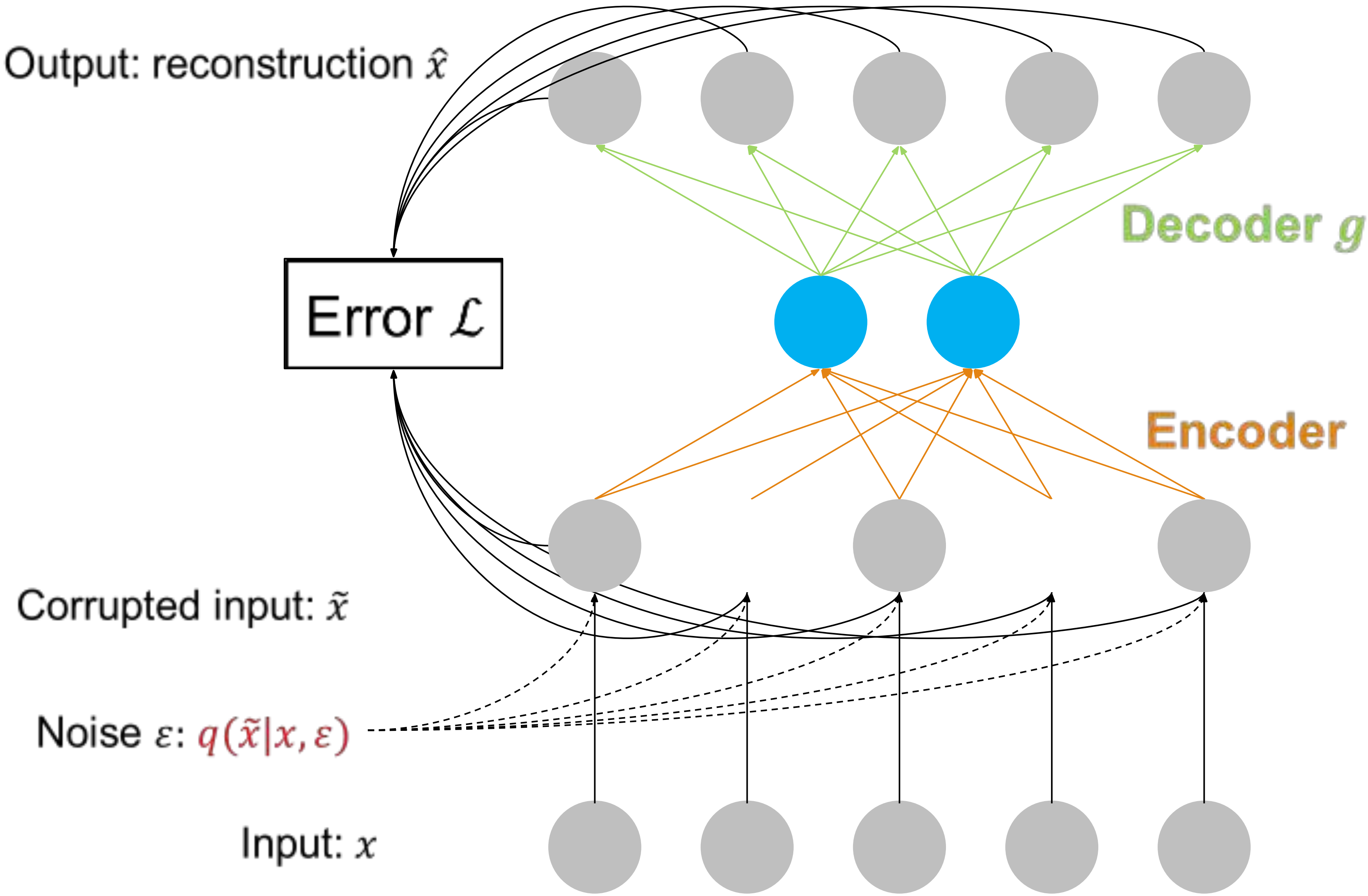
Otherwise (overcomplete) autoencoder might learn the identity function

$$W \propto I \implies \tilde{x} = x \implies \mathcal{L} = 0$$

- Assuming no regularization
- Often in practice still works though

Also, if $z = Wx + b$ (linear) autoencoder learns same subspace as PCA

Denoising Autoencoder



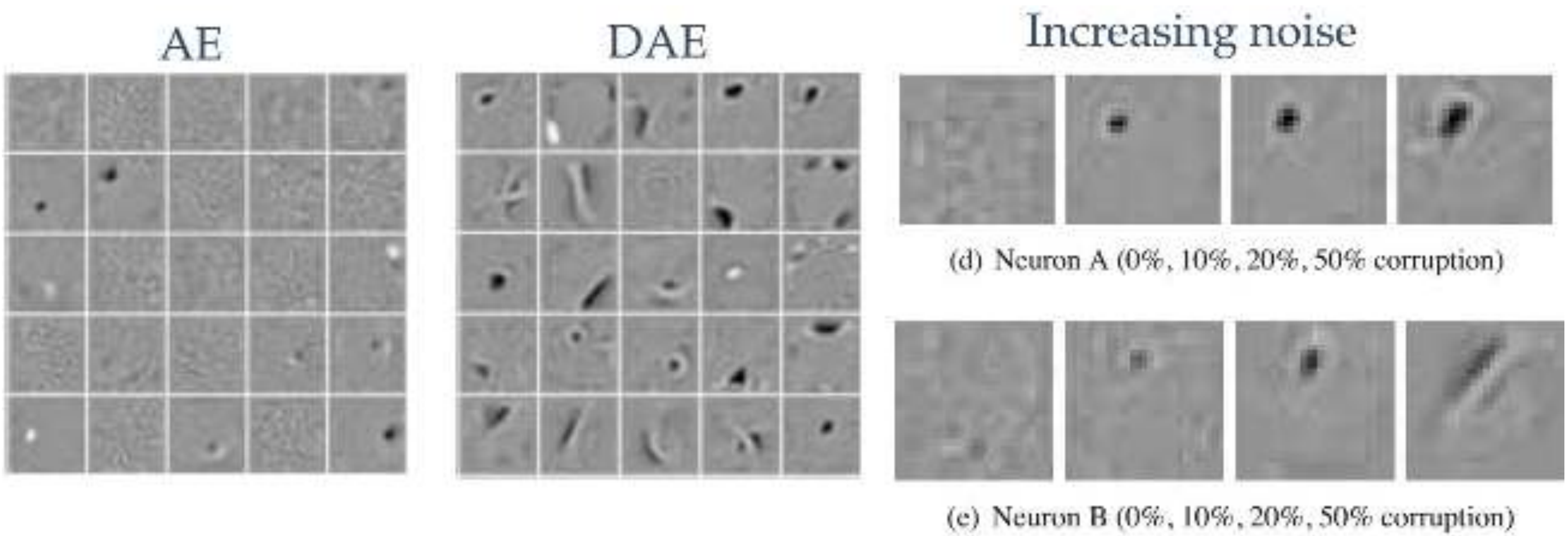
Denoising Autoencoder

The network does not overlearn the data

- Can even use overcomplete latent spaces

Model forced to learn more intelligent, robust representations

- Learn to ignore noise or trivial solutions(identity)
- Focus on “underlying” data generation process



Variational Autoencoder

We want to model the data distribution

$$p(x) = \int p_\theta(z) p_\theta(x|z) dz$$

Posterior $p_\theta(z|x)$ is intractable for complicated likelihood functions $p_\theta(x|z)$, e.g. a neural network → $p(x)$ is also intractable

Introduce an inference machine $q_\varphi(z|x)$ (e.g. another neural network) that **learns to approximate** the posterior $p_\theta(z|x)$

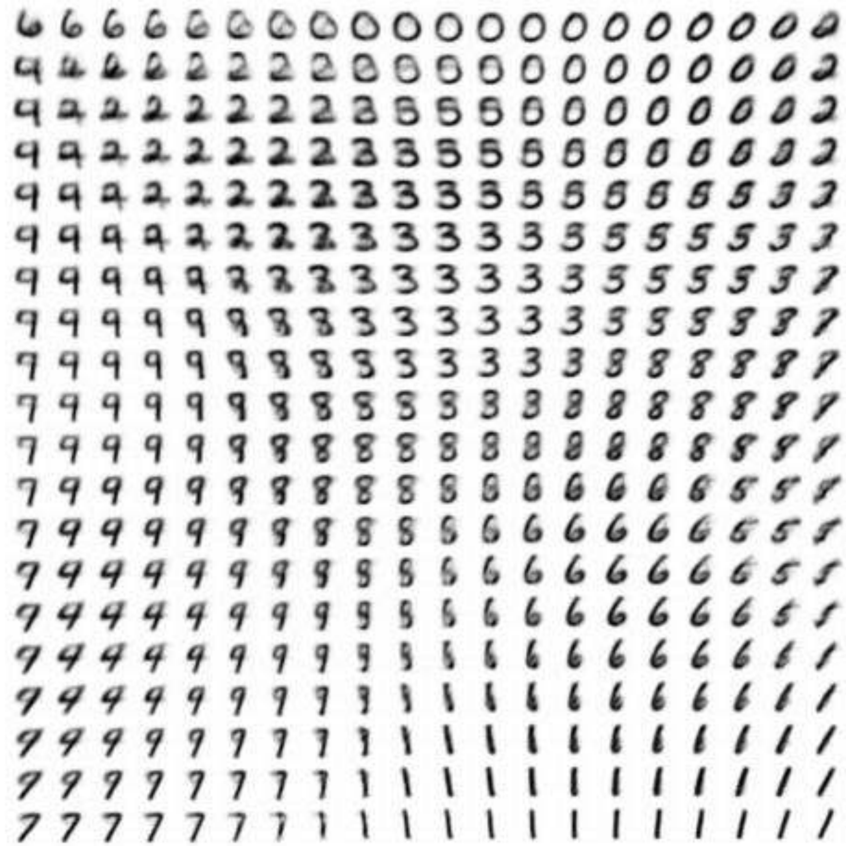
- Since we cannot know $p_\theta(z|x)$ define a variational lower bound to optimize instead

$$\mathcal{L}(\theta, \varphi, x) = -D_{KL}\left(q_\varphi(z|x) \| p_\theta(z)\right) + E_{q_\varphi(z|x)}(\log p_\theta(x|z))$$

Regularization term — Reconstruction term

Examples

(a) Learned Frey Face manifold



(b) Learned MNIST manifold

Figure 4: Visualisations of learned data manifold for generative models with two-dimensional latent space, learned with AEVB. Since the prior of the latent space is Gaussian, linearly spaced coordinates on the unit square were transformed through the inverse CDF of the Gaussian to produce values of the latent variables $\mathbf{z}$. For each of these values $\mathbf{z}$, we plotted the corresponding generative $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ with the learned parameters $\boldsymbol{\theta}$.

Generative Adversarial Networks

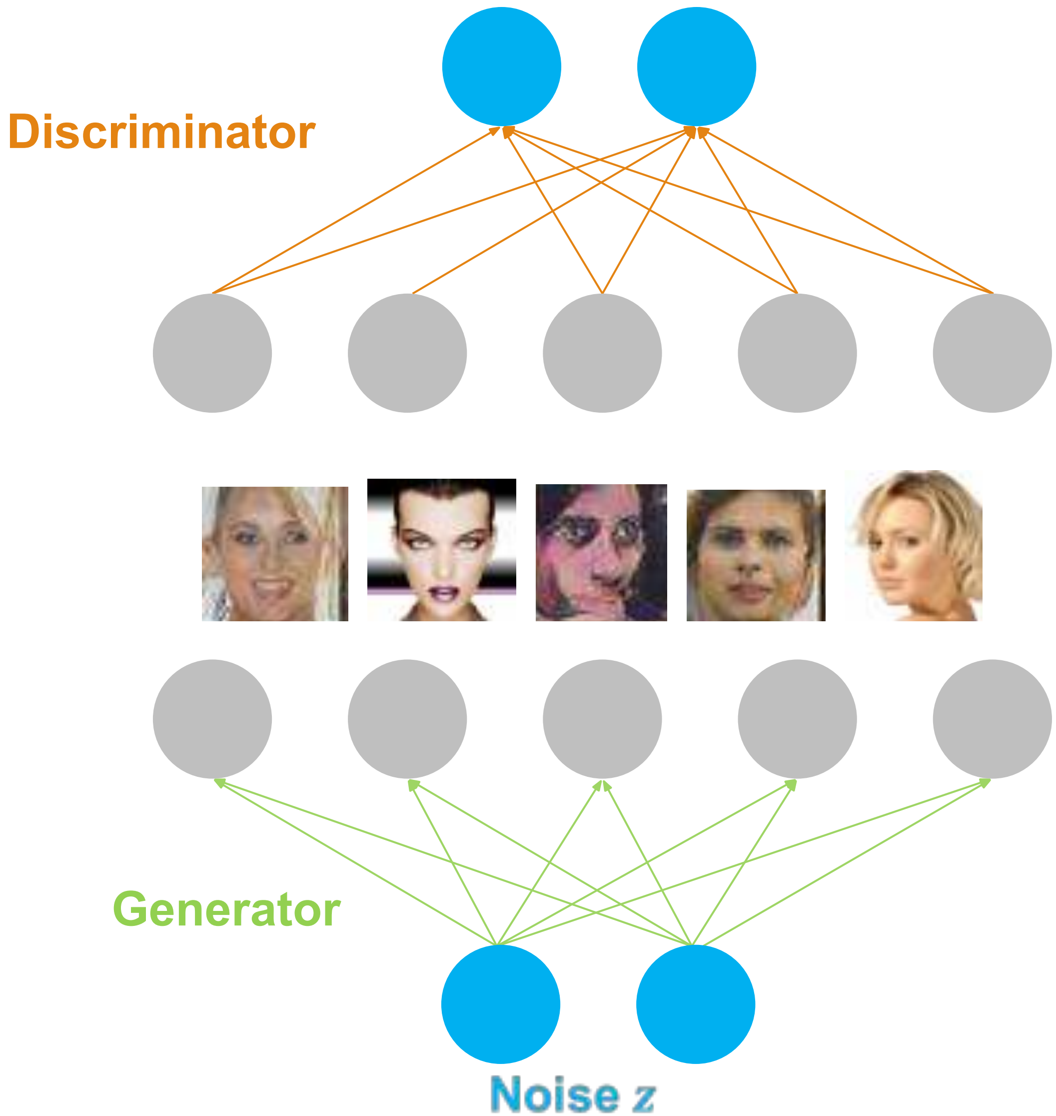
Composed of two successive networks

- Generator network (like upper half of autoencoders)
- Discriminator network (like a convent)

Learning

- Sample “noise” vectors z
- Per z the generator produces a sample x
- Make a batch where half samples are real, half are the generated ones
- The discriminator needs to predict what is real and what is fake

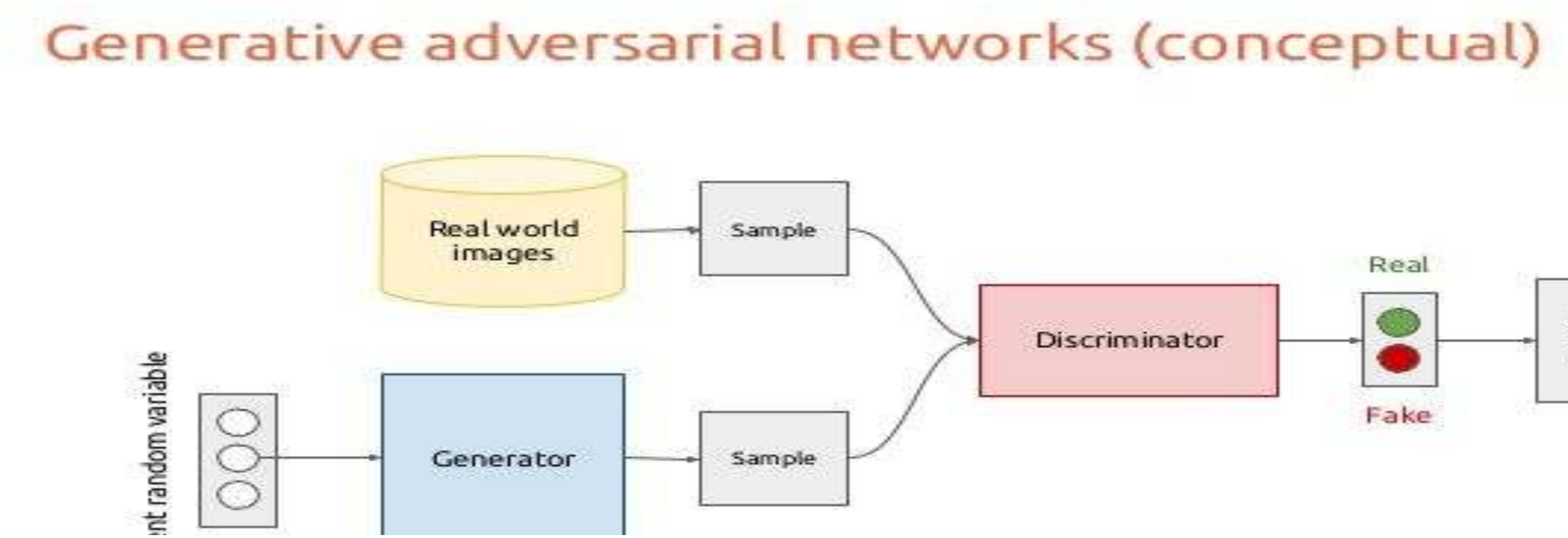
Generative Adversarial Networks



“Police vs Thief”

Generator and discriminator networks optimized together

- ❑ The generator (thief) tries to fool the discriminator
- ❑ The discriminator (police) tries to not get fooled by the generator



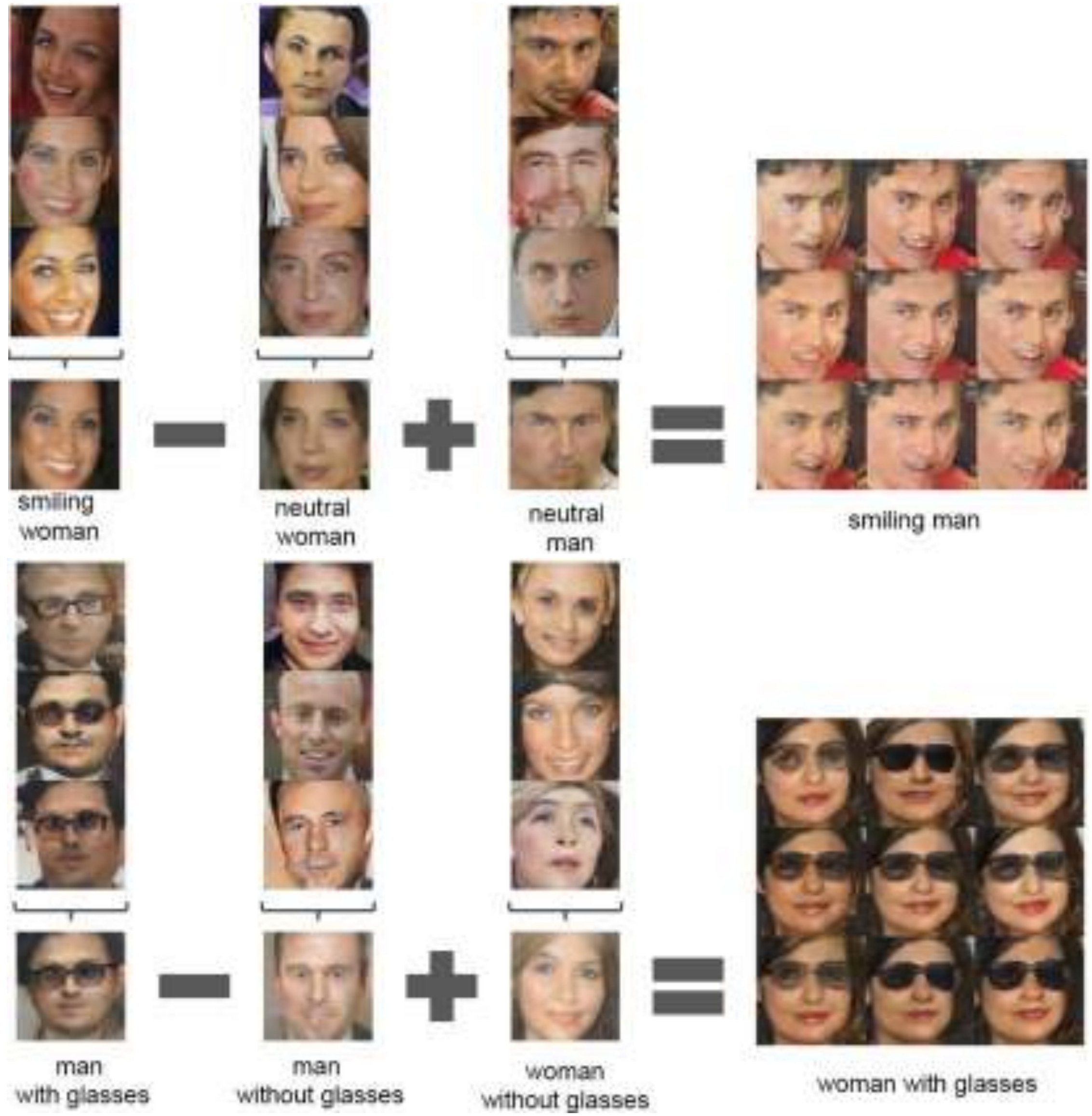
$$J^{(D)} = -\frac{1}{2}\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}} \log D(\boldsymbol{x}) - \frac{1}{2}\mathbb{E}_{\boldsymbol{z}} \log\left(1 - D\left(G(\boldsymbol{z})\right)\right)$$

$$J^{(G)} = -J^{(D)}$$

Examples

Bedrooms

Image “arithmetics”



Thank you!