Illustration: Koma Zhang // Quanta Magazine

BBM444

FUNDAMENTALS OF COMPUTATIONAL PHOTOGRAPHY

Lecture #09 – Convolutional Neural Networks

Erkut Erdem // Hacettepe University // Spring 2023

HACETTEPE UNIVERSITY COMPUTER VISION LAB

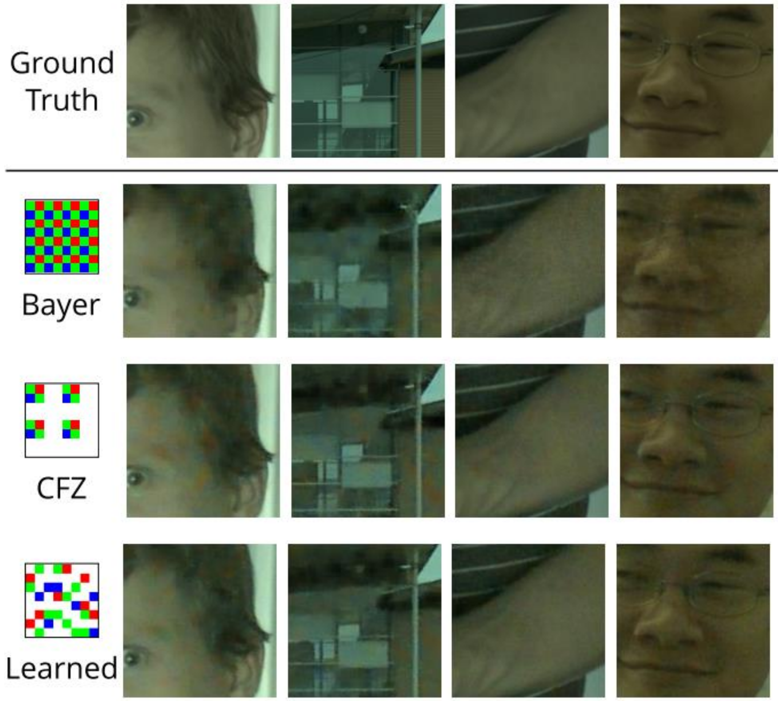
Today’s Lecture

- unreasonable effectiveness of data
- deep learning
- computation in a neural net
- optimization
- backpropagation
- convolutional neural networks
- applications in computational photography

Disclaimer: The material and slides for this lecture were borrowed from
— Costis Daskalakis and Aleksander Mądry’s MIT 6.883 class
— Bill Freeman, Antonio Torralba and Phillip Isola’s MIT 6.869 class

Neural Networks in Computational Photography

- Now: learned pipelines for computational imaging

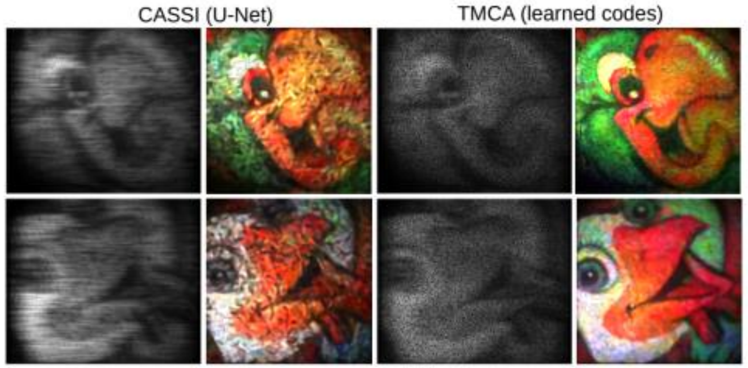


Learning CFAs

(b) Raw data via traditional pipeline

(c) Our result

Learning ISPs



Learning coded apertures

Neural Networks in Computational Photography

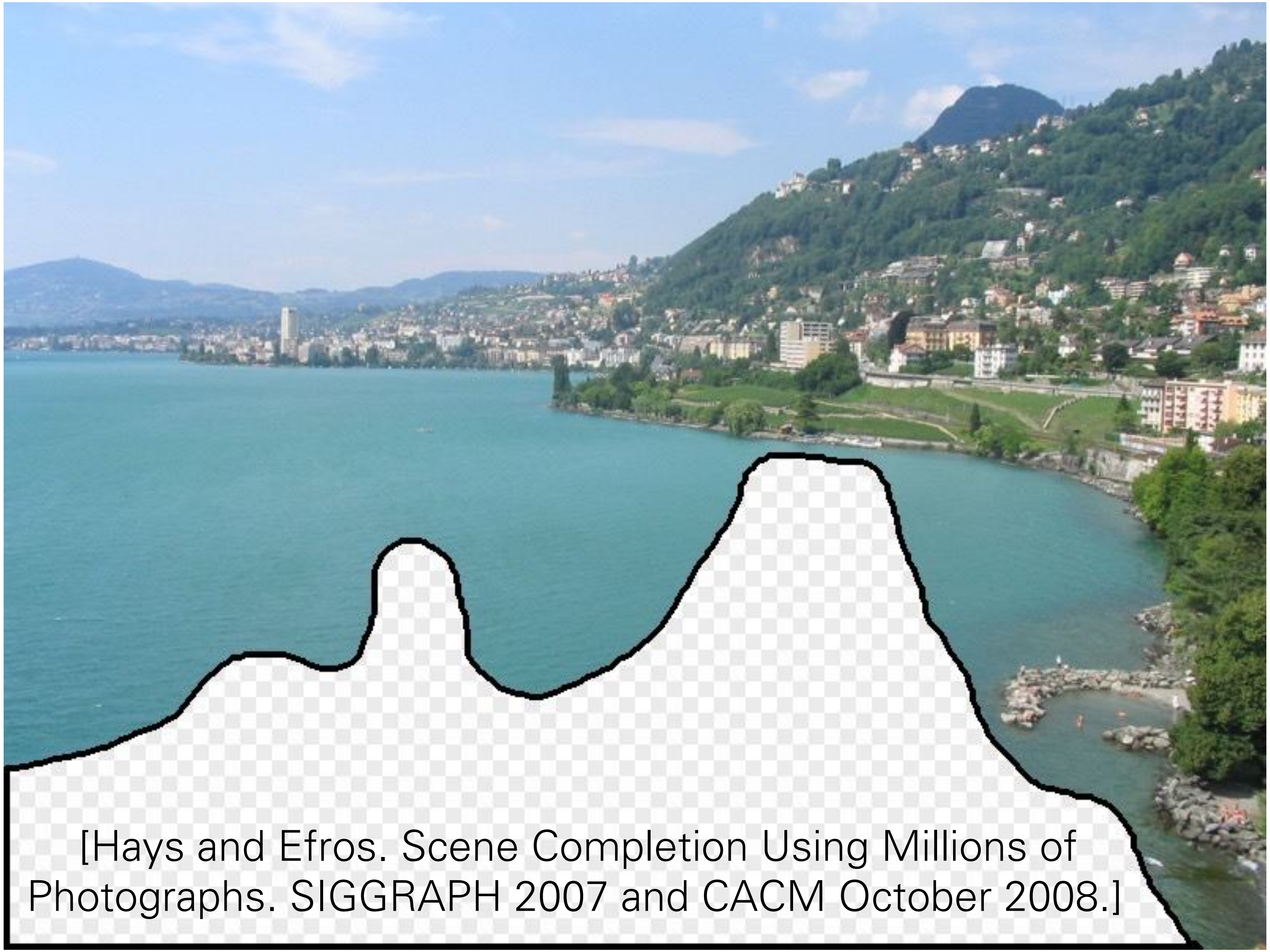
- Now: learned pipelines for computational imaging

Learning denoising

Learning deblurring

HDR Imaging

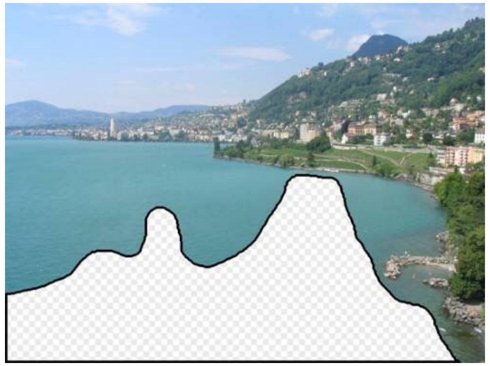
Unreasonable Effectiveness of Data

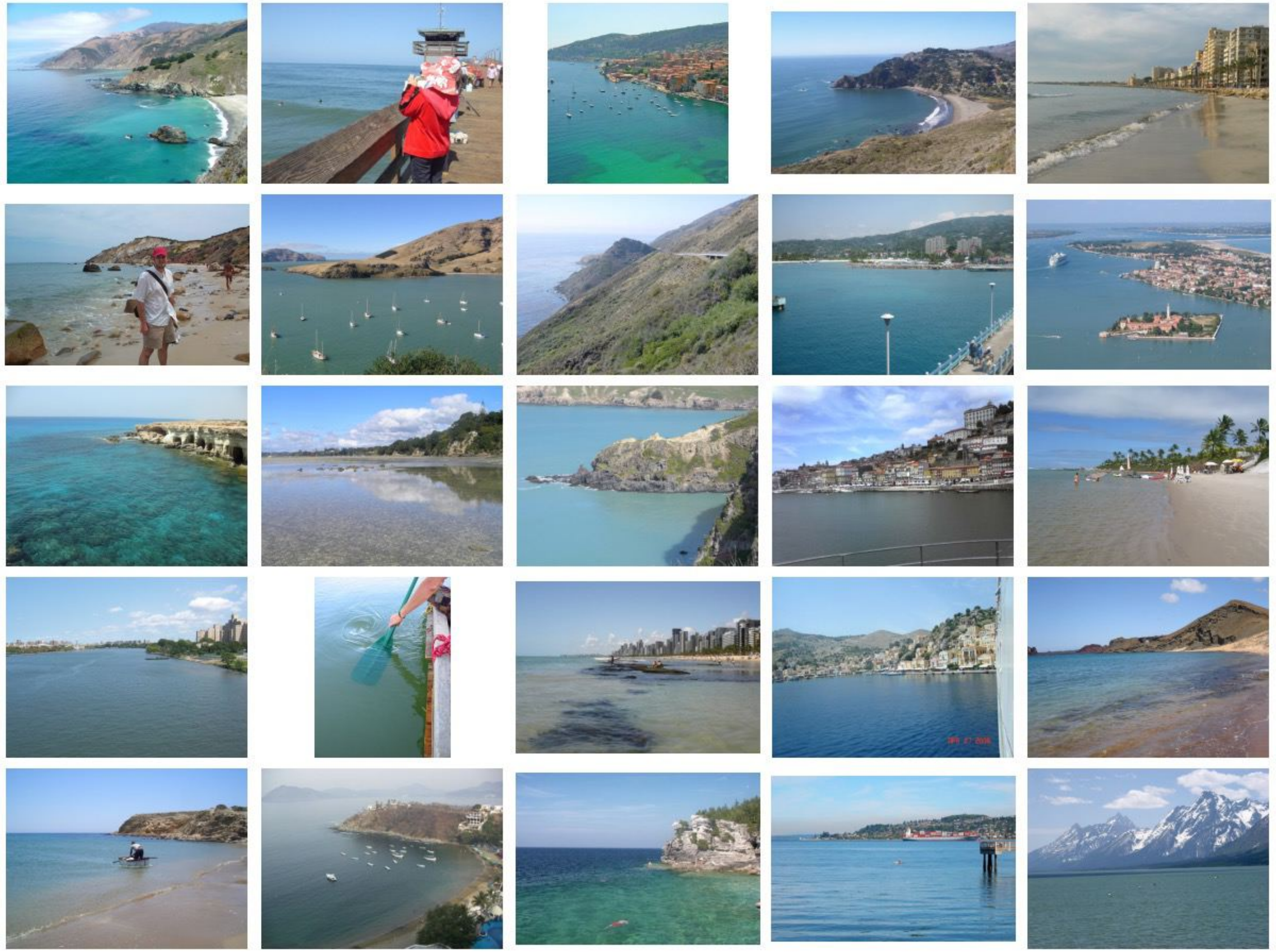


[Hays and Efros. Scene Completion Using Millions of Photographs. SIGGRAPH 2007 and CACM October 2008.]

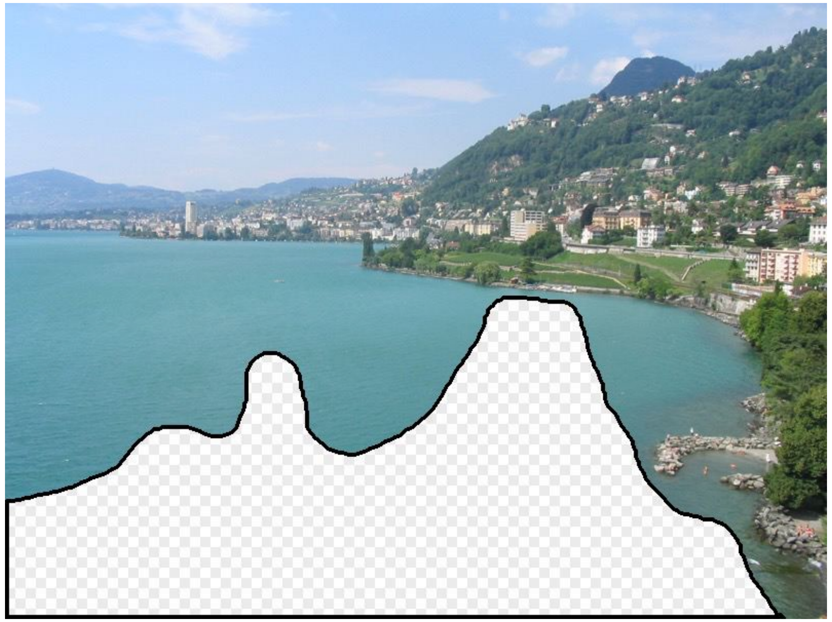
2 Million Flickr Images

Slide credit: Alexei Efros

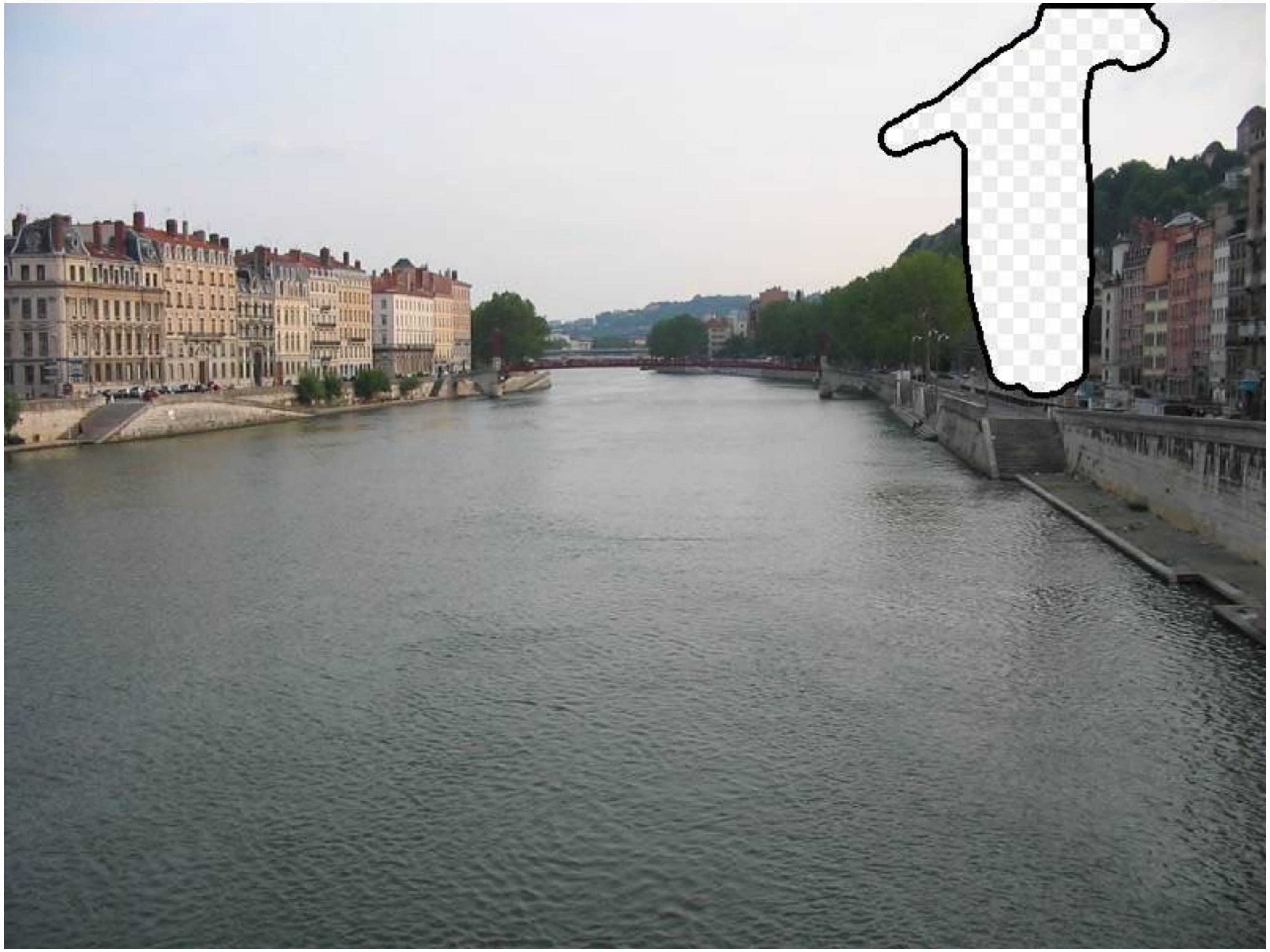


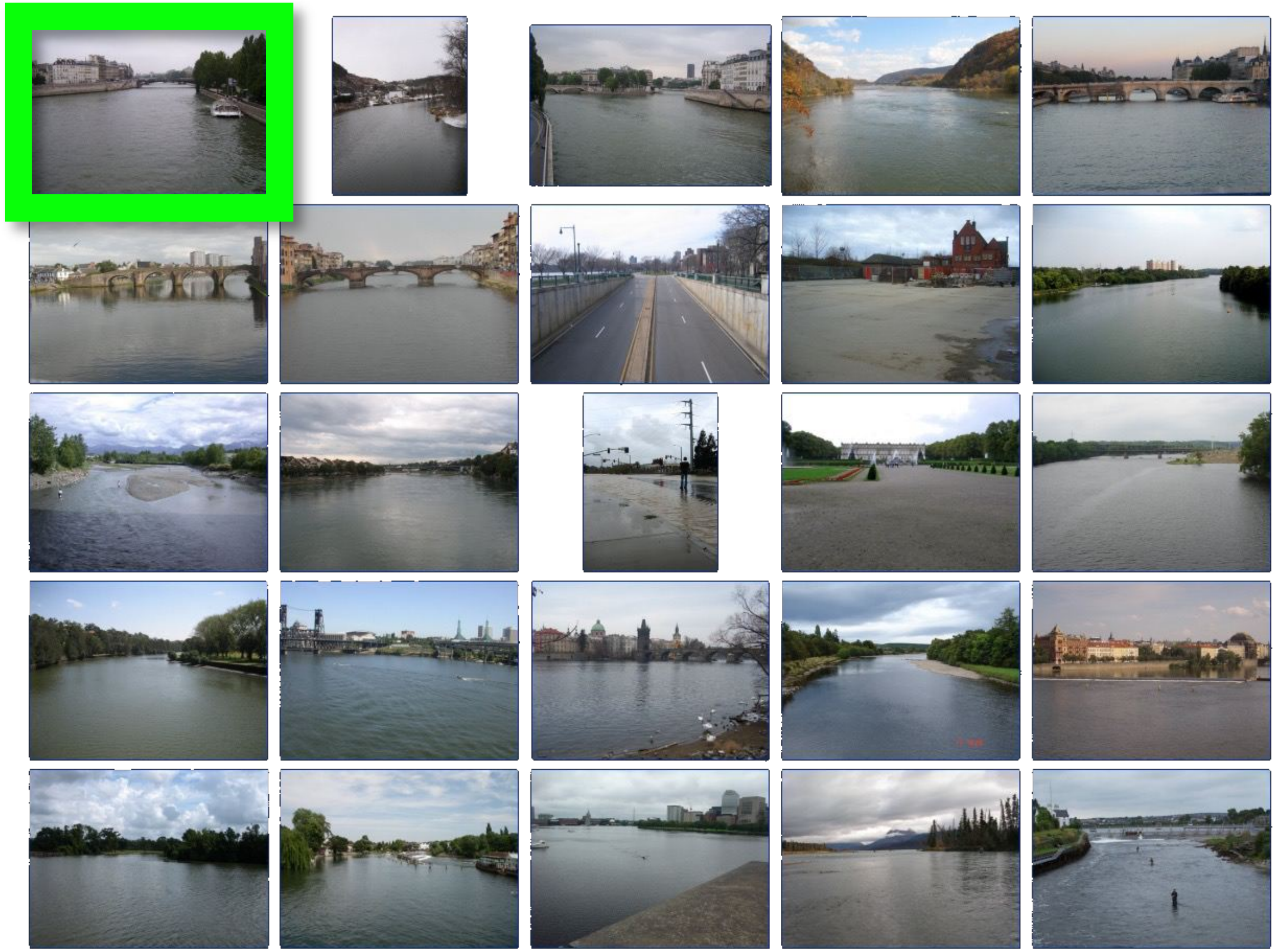


… 200 total

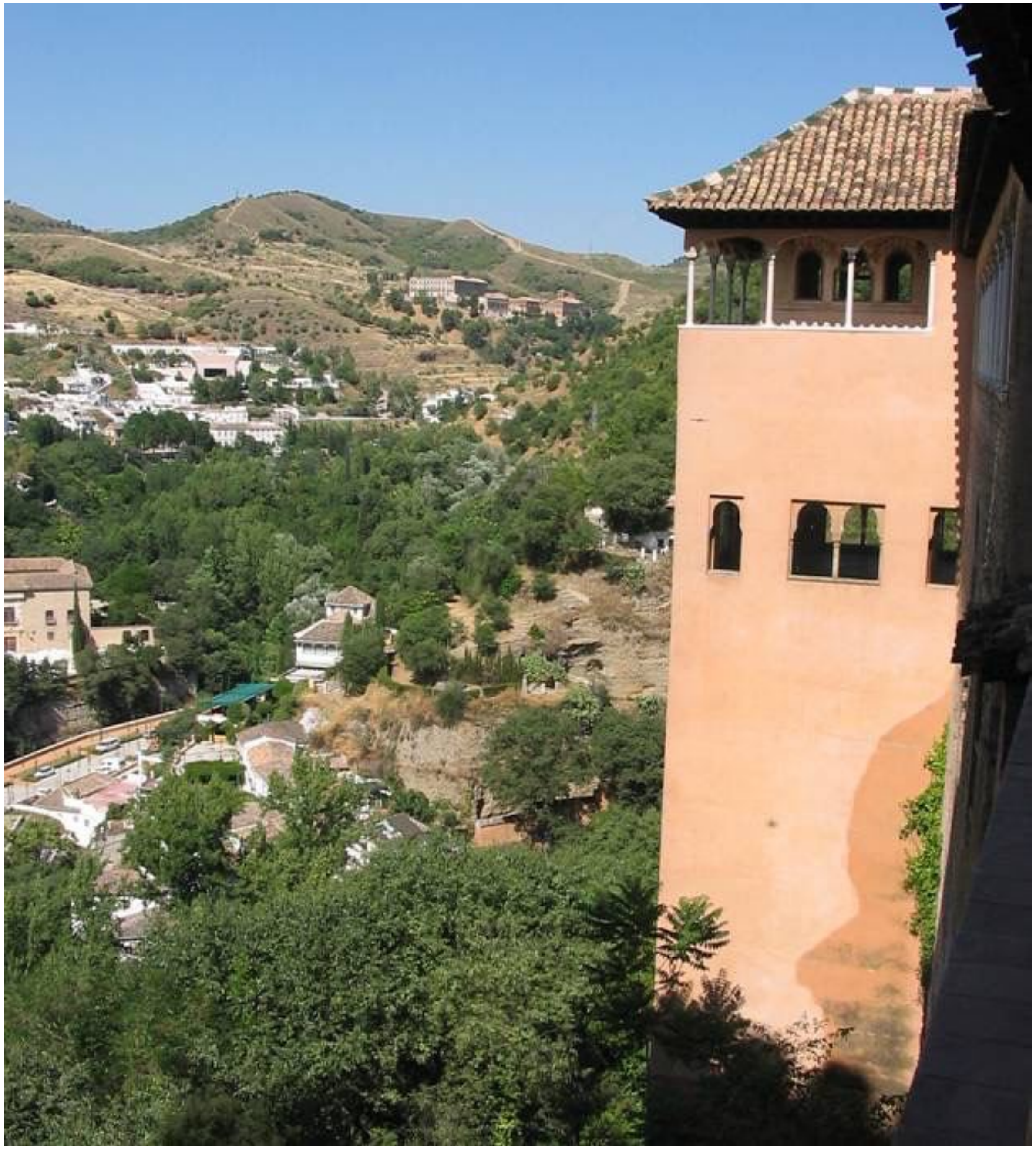


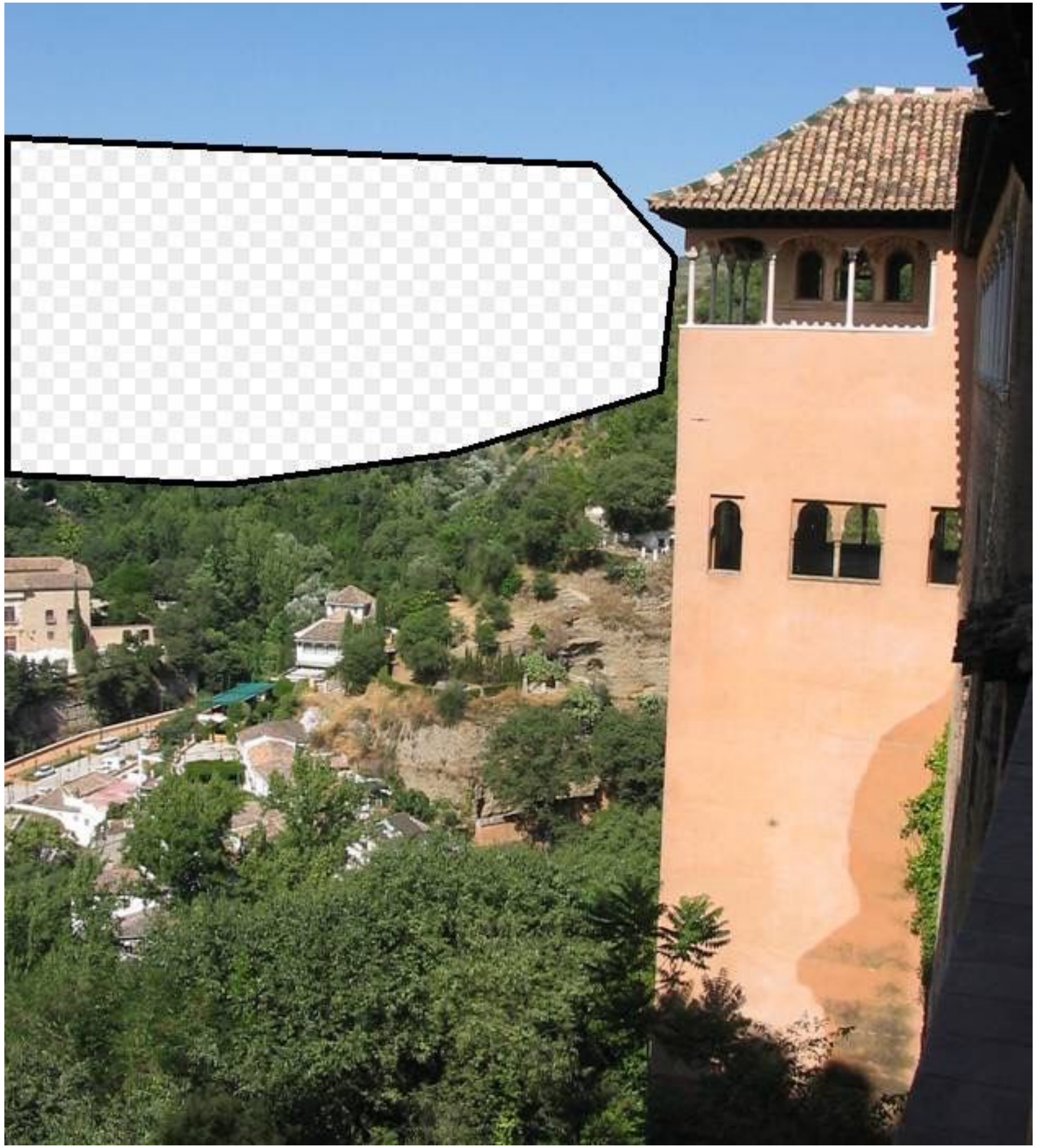
Slide credit: Alexei Efros



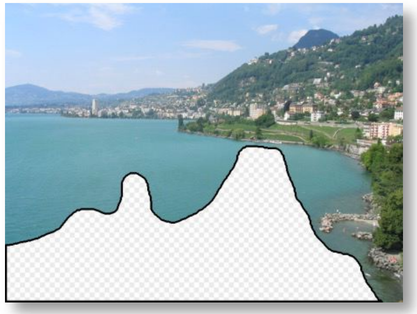


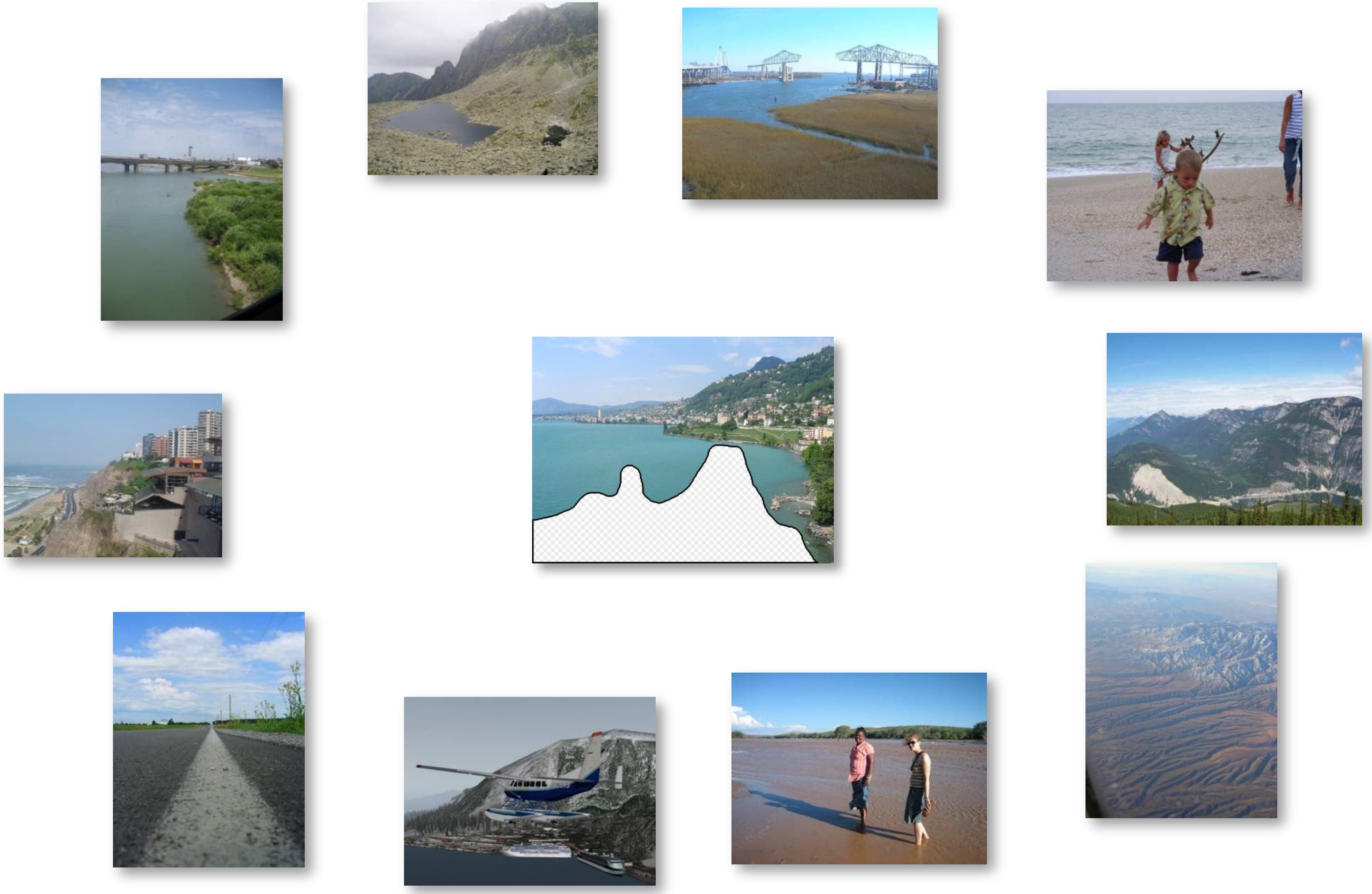
… 200 scene matches



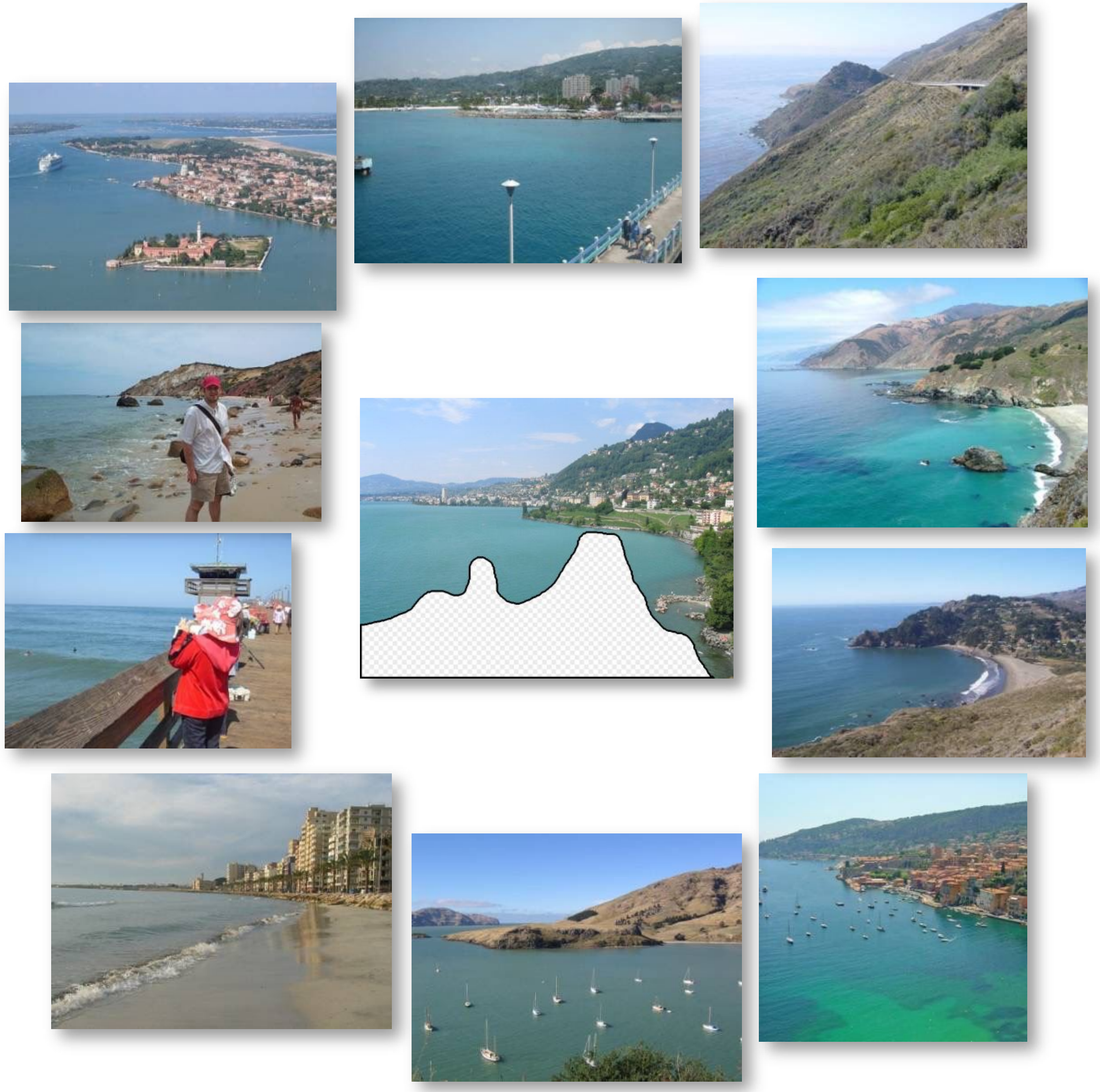


Why does it work?





Nearest neighbors from a collection of 20 thousand images



Nearest neighbors from a collection of 2 million images

“Unreasonable Effectiveness of Data”

[Halevy, Norvig, Pereira 2009]

- Parts of our world can be explained by elegant mathematics

  physics, chemistry, astronomy, etc.

- But much cannot

  psychology, economics, genetics, etc.

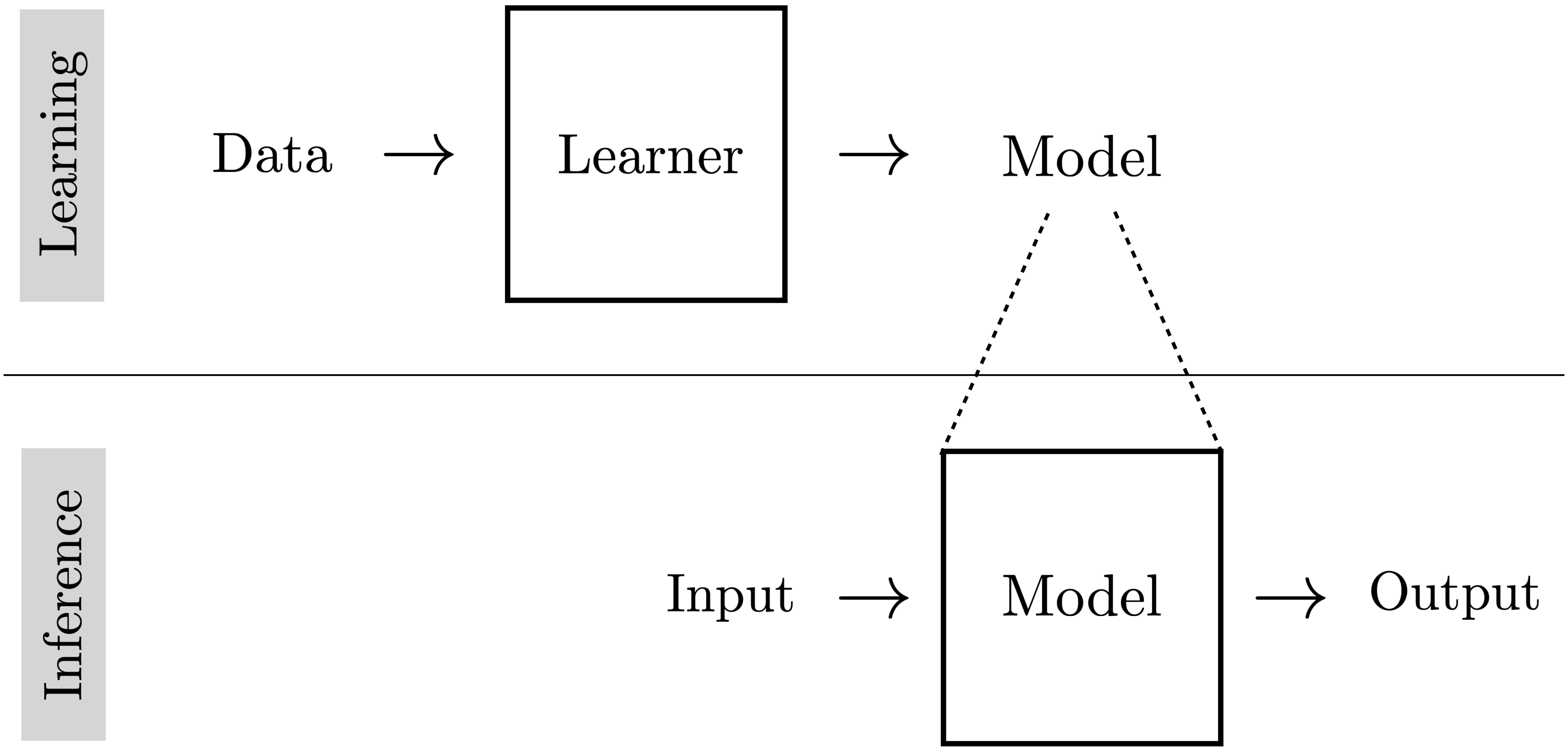
- Enter The Data!

  Great advances in several fields:

  e.g., speech recognition,
  machine translation

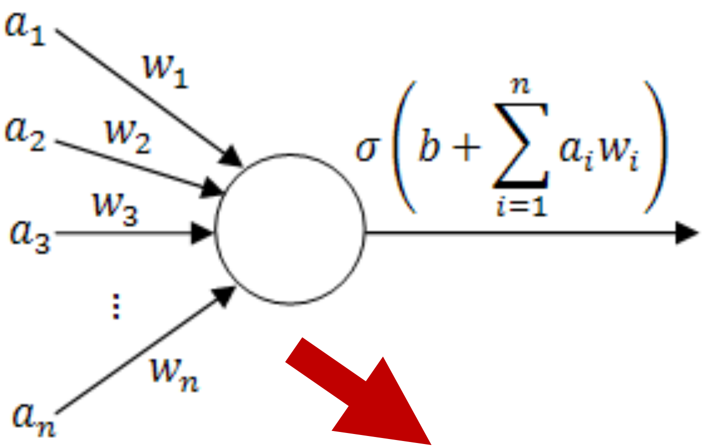
  Case study: Google

> “For many tasks, once we have a billion or so examples, we essentially have a closed set that represents (or at least approximates) what we need…”

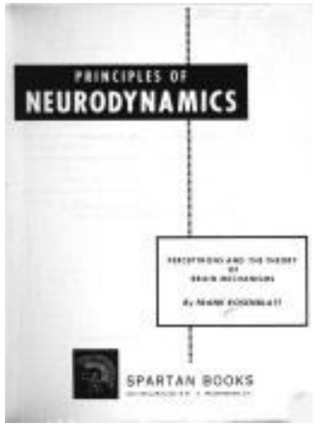

Learning
Data
Learner
Model
Inference
Input
Model
Output

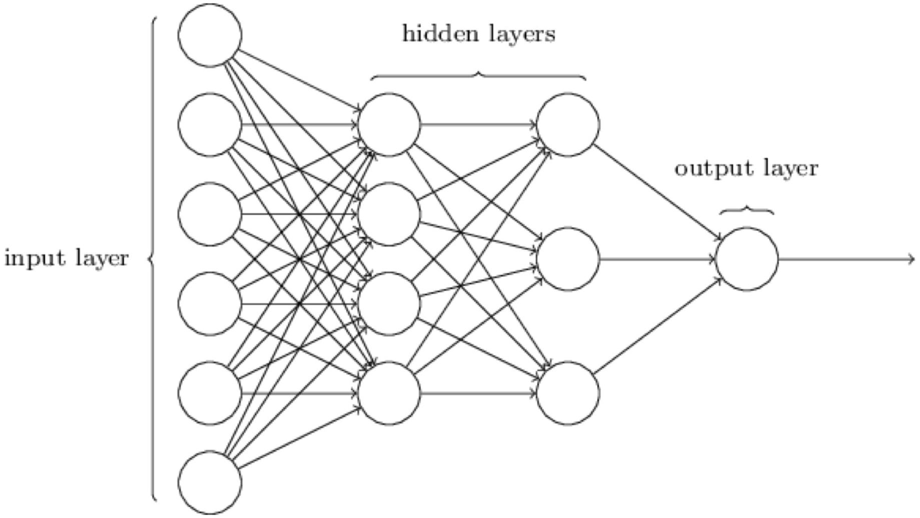
A Brief History of Deep Learning

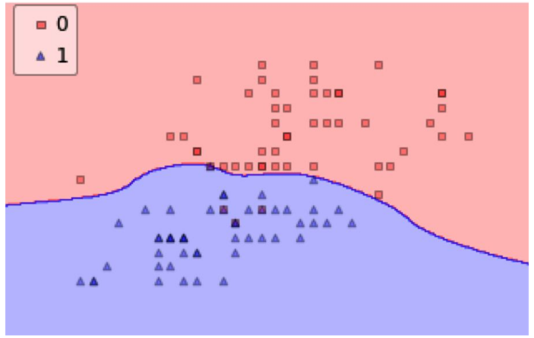
Humble beginnings

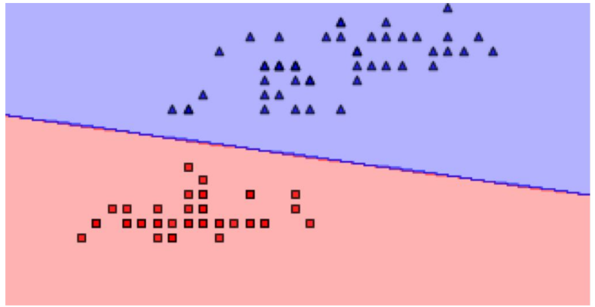


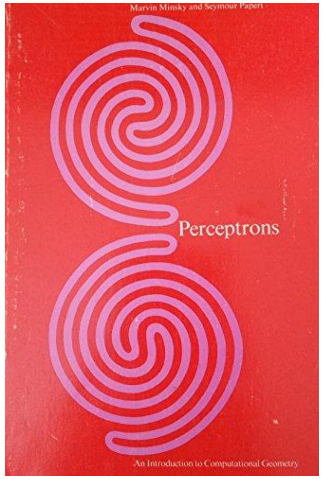
- Perceptron [Rosenblatt '58]





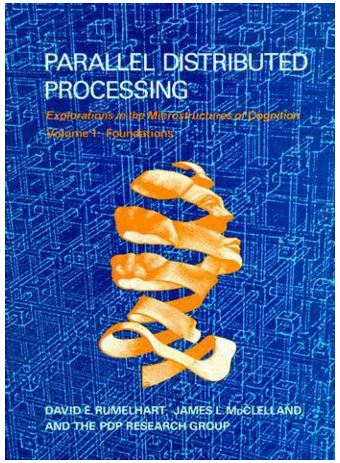




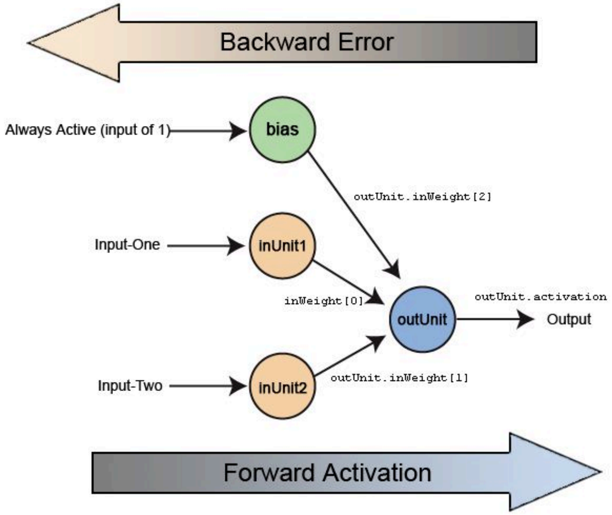


- Criticism of Perceptrons (XOR affair) [Minsky Papert '69]
  - Effectively causes a "deep learning winter"

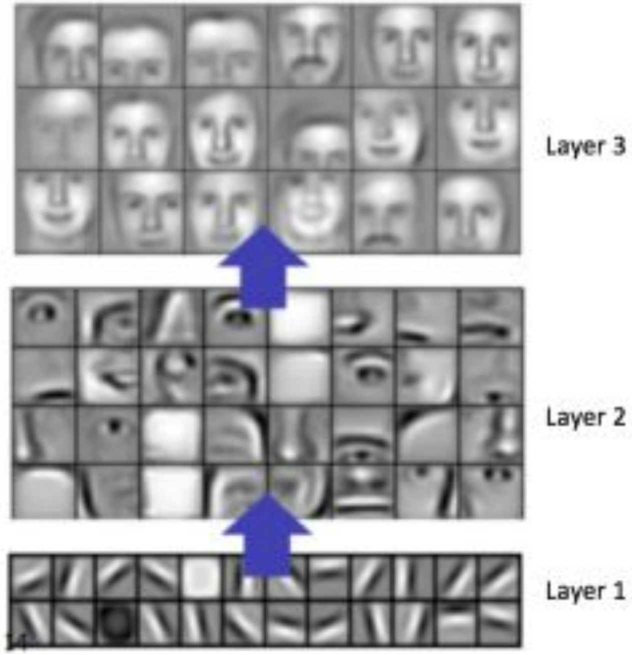
(Early) Spring



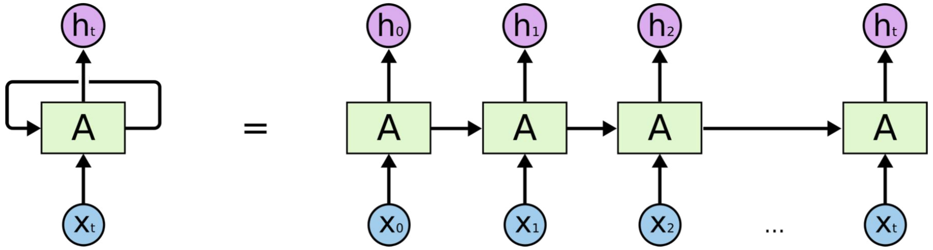
- Back-propagation [Rumelhart et al. '86, LeCun '85, Parker '85]



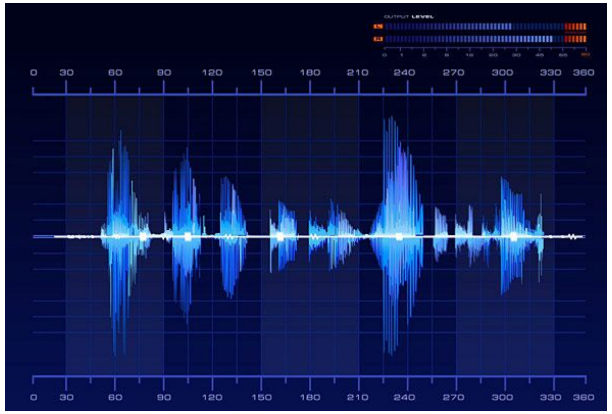
- Convolutional layers [LeCun et al. '90]



- Recurrent Neural Networks/Long Short-Term Memory (LSTM) [Hochreiter Schmidhuber '97]

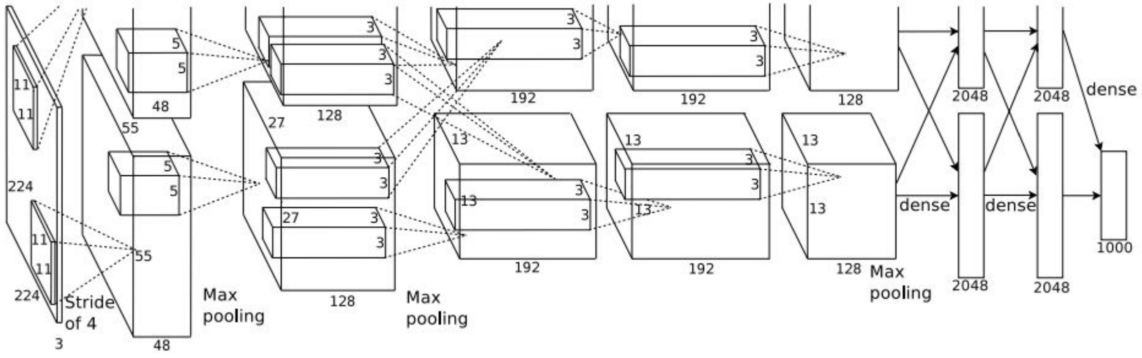


Summer

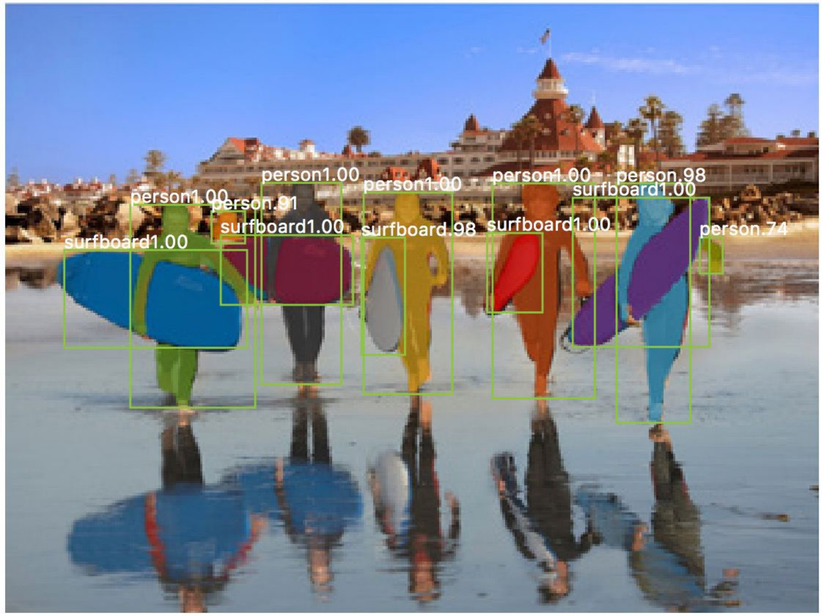


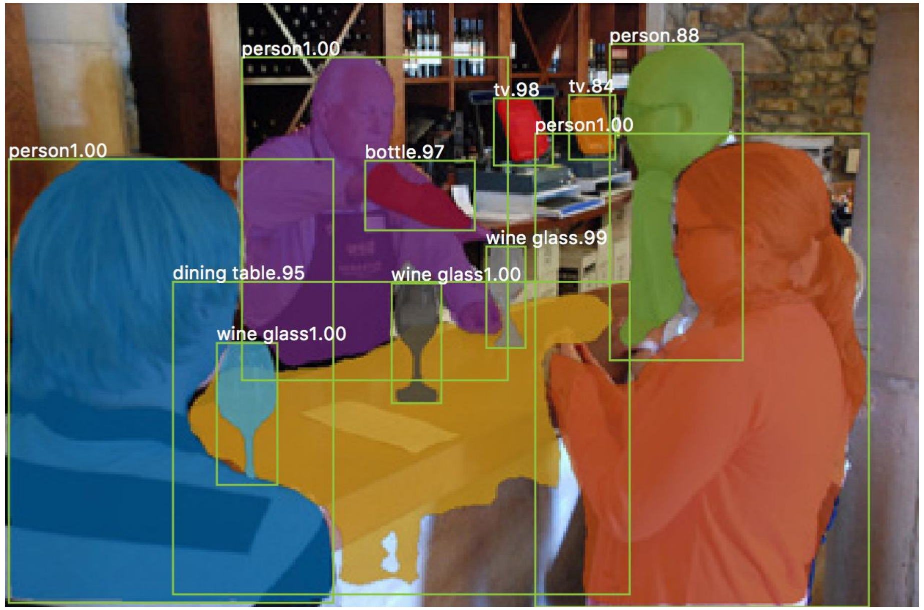
- **2006:** First big success: speech recognition

- **2012:** Breakthrough in computer vision: AlexNet [Krizhevsky et al. '12]

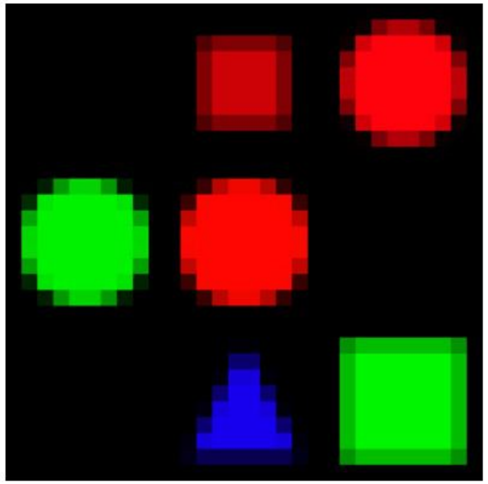


- **2015:** Deep learning-based vision models outperform humans





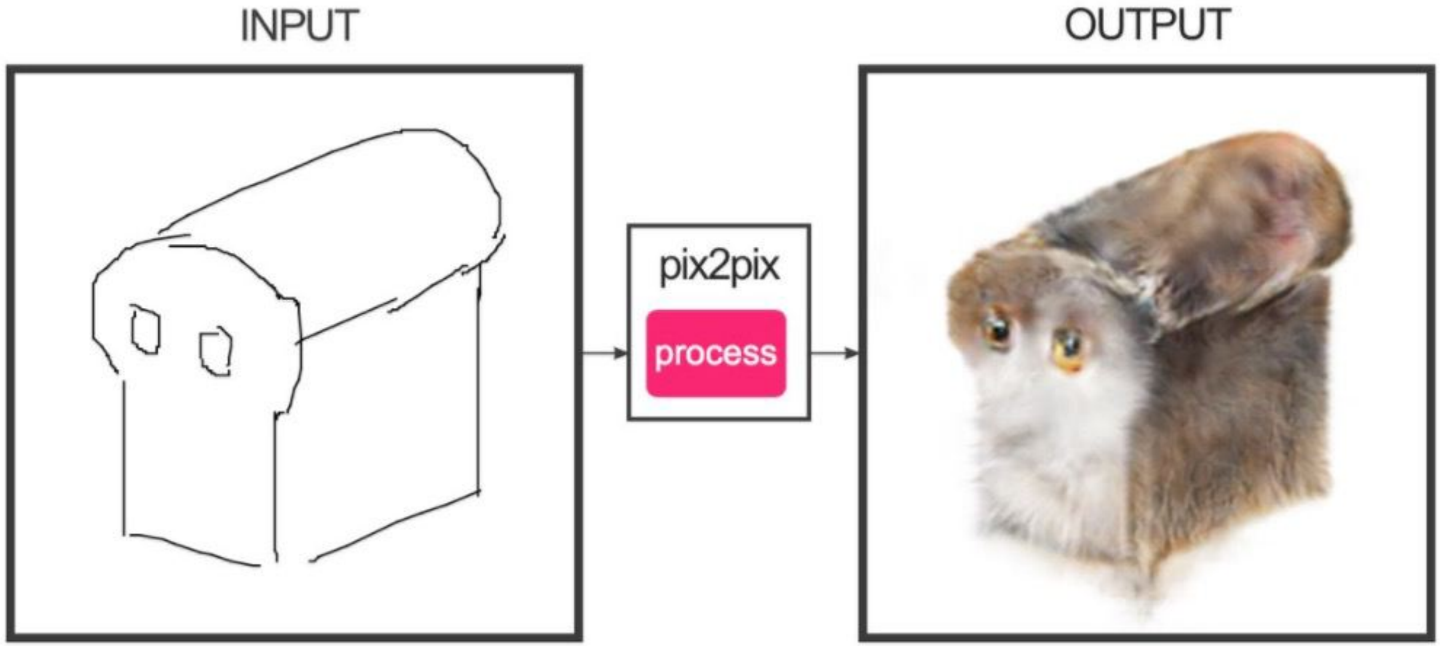
[“Mask RCNN”, He et al. 2017]

		
what color is the vase?	*is the bus full of passengers?*	*is there a red shape above a circle?*

`classify[color](`
`  attend[vase])`
	`measure[is](`
	`  combine[and](`
	`    attend[bus],`
	`    attend[full])`
		`measure[is](`
		`  combine[and](`
		`   attend[red],`
		`   re-attend[above](`
		`    attend[circle])))`

green (green)	yes (yes)	no (no)

["Neural module networks", Andreas et al. 2017]



Ivy Tasi @ivymyt

Vitaly Vidmirov @vvid

["pix2pix", Isola et al. 2017]

What enabled this success?

- Better architectures (e.g., ReLUs) and regularization techniques (e.g. Dropout)

IMAGENET

- Sufficiently large datasets

- Enough computational power

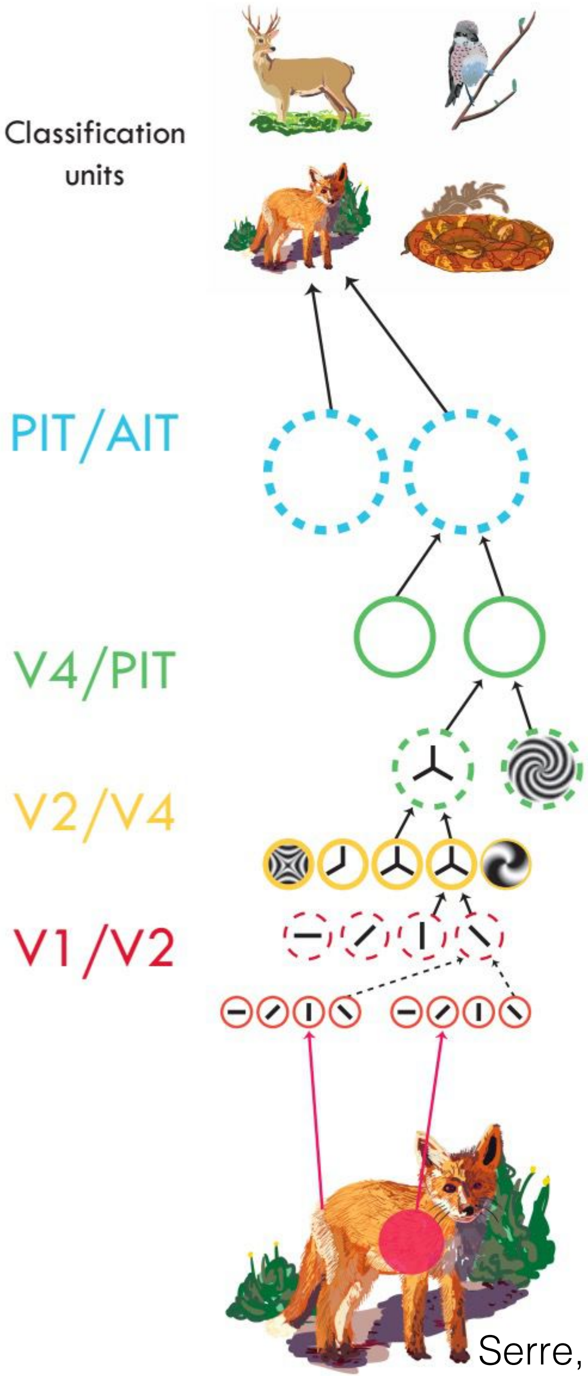
Deep learning

- Modeling the visual world is incredibly complicated. We need high capacity models.

- In the past, we didn't have enough data to fit these models. But now we do!

- We want a class of **high capacity models** that are **easy to optimize**.

Deep neural networks!


Classification units
PIT/AIT
V4/PIT
V2/V4
V1/V2

Serre, 2014

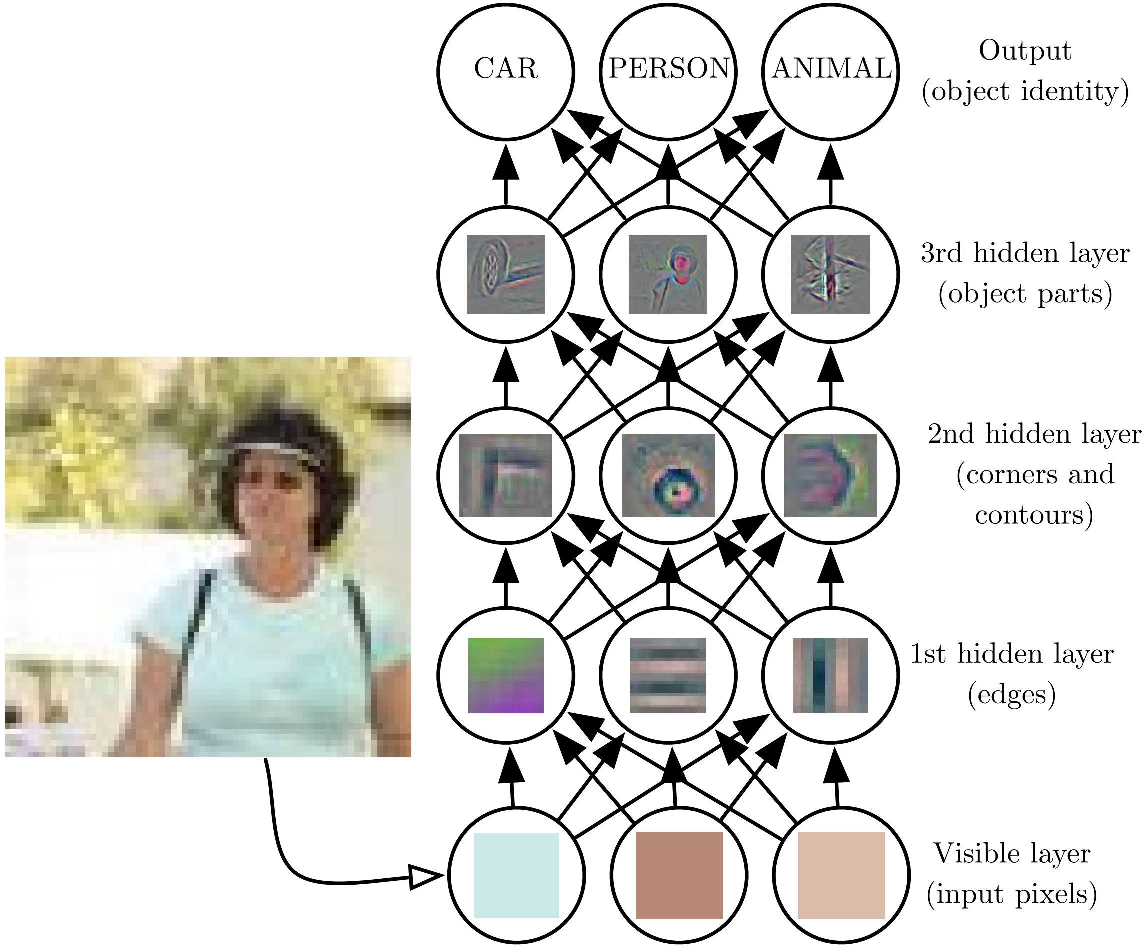
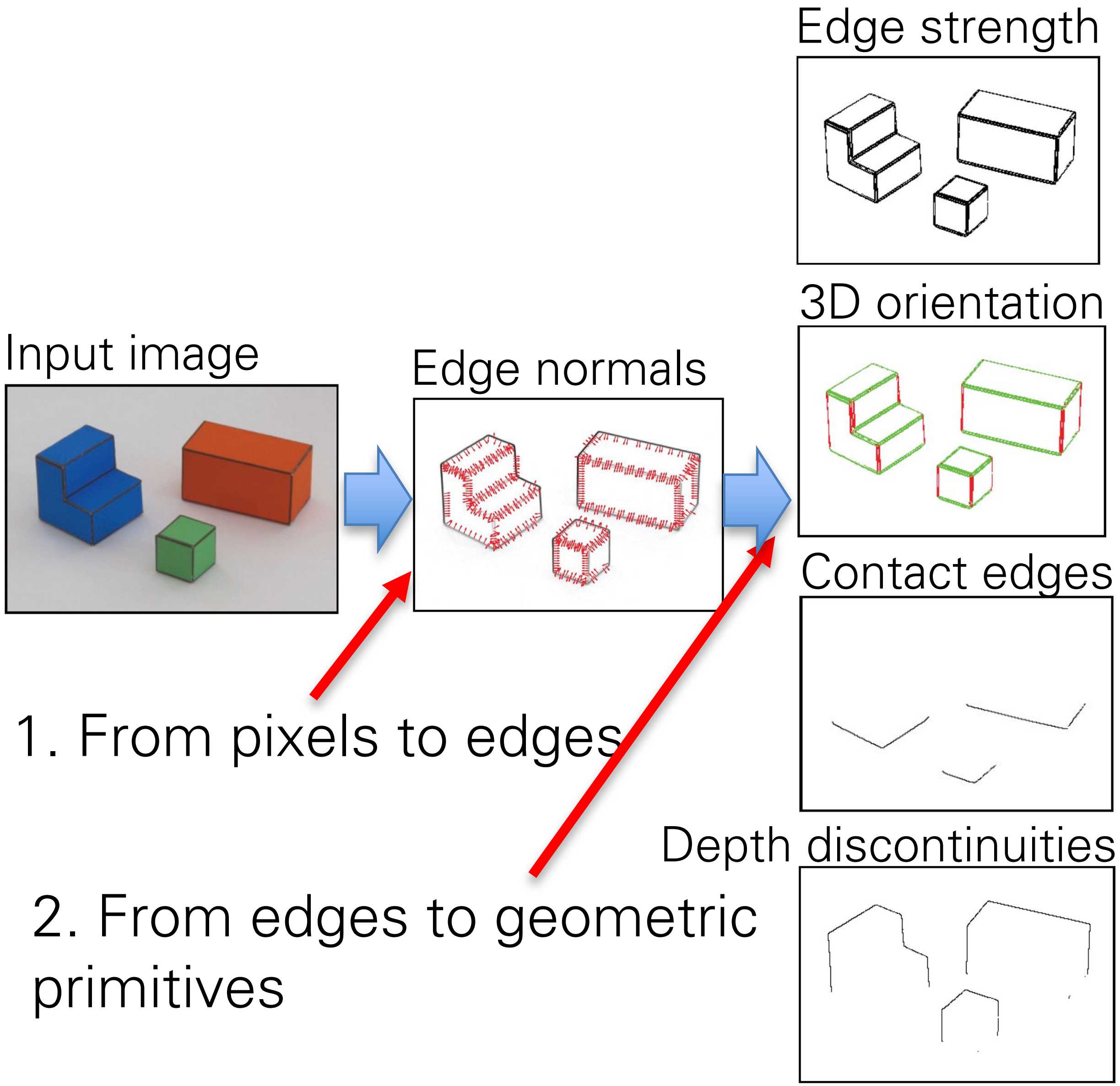
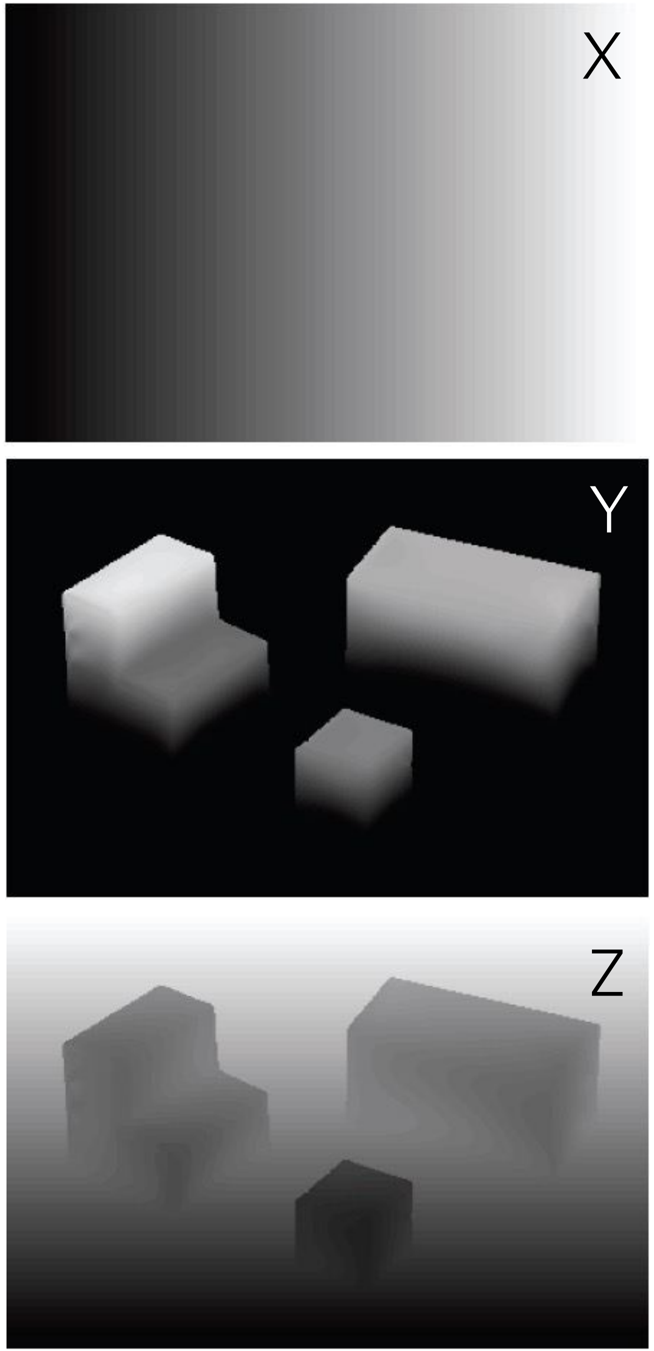
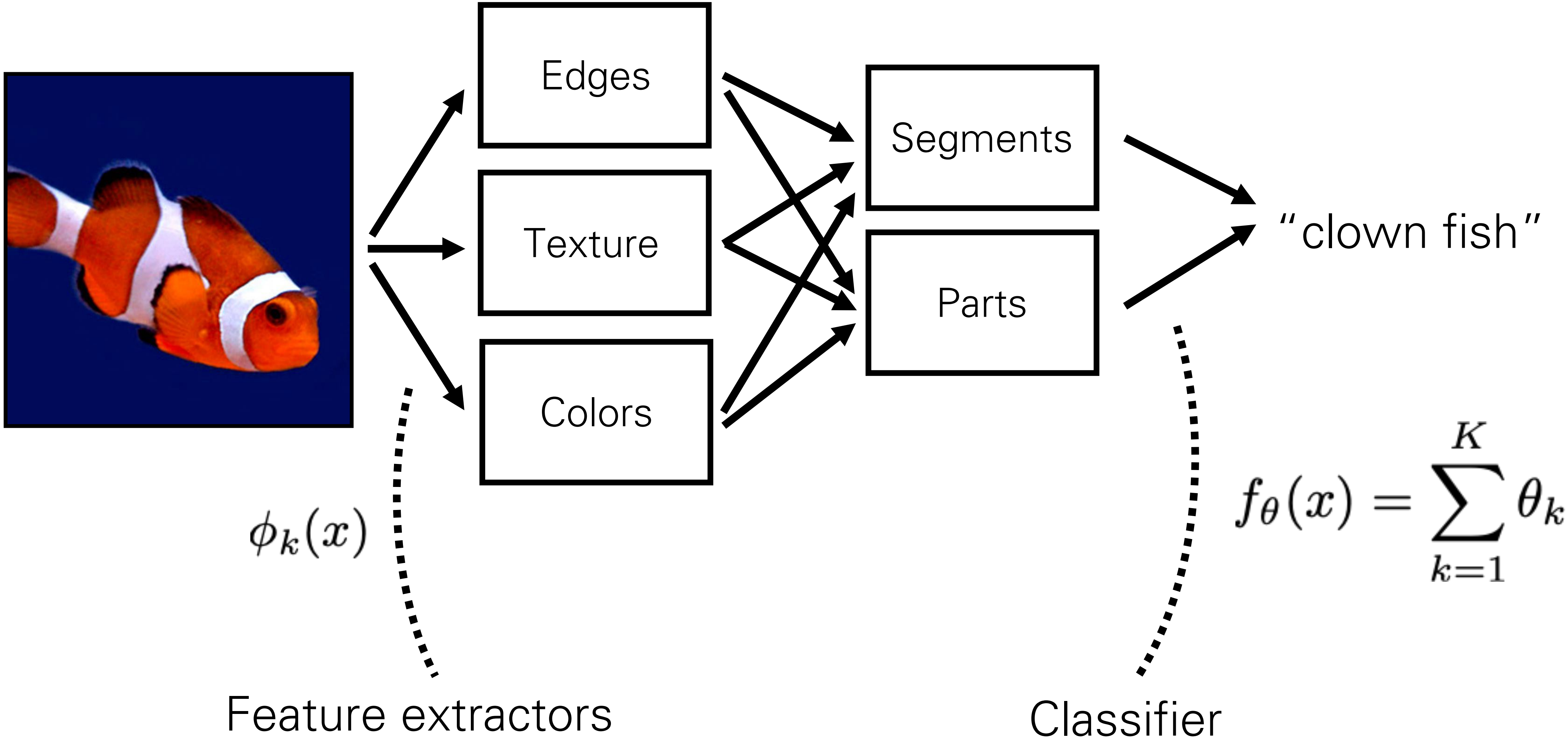

CAR
PERSON
ANIMAL
Output (object identity)
3rd hidden layer (object parts)
2nd hidden layer (corners and contours)
1st hidden layer (edges)
Visible layer (input pixels)

Image transformations



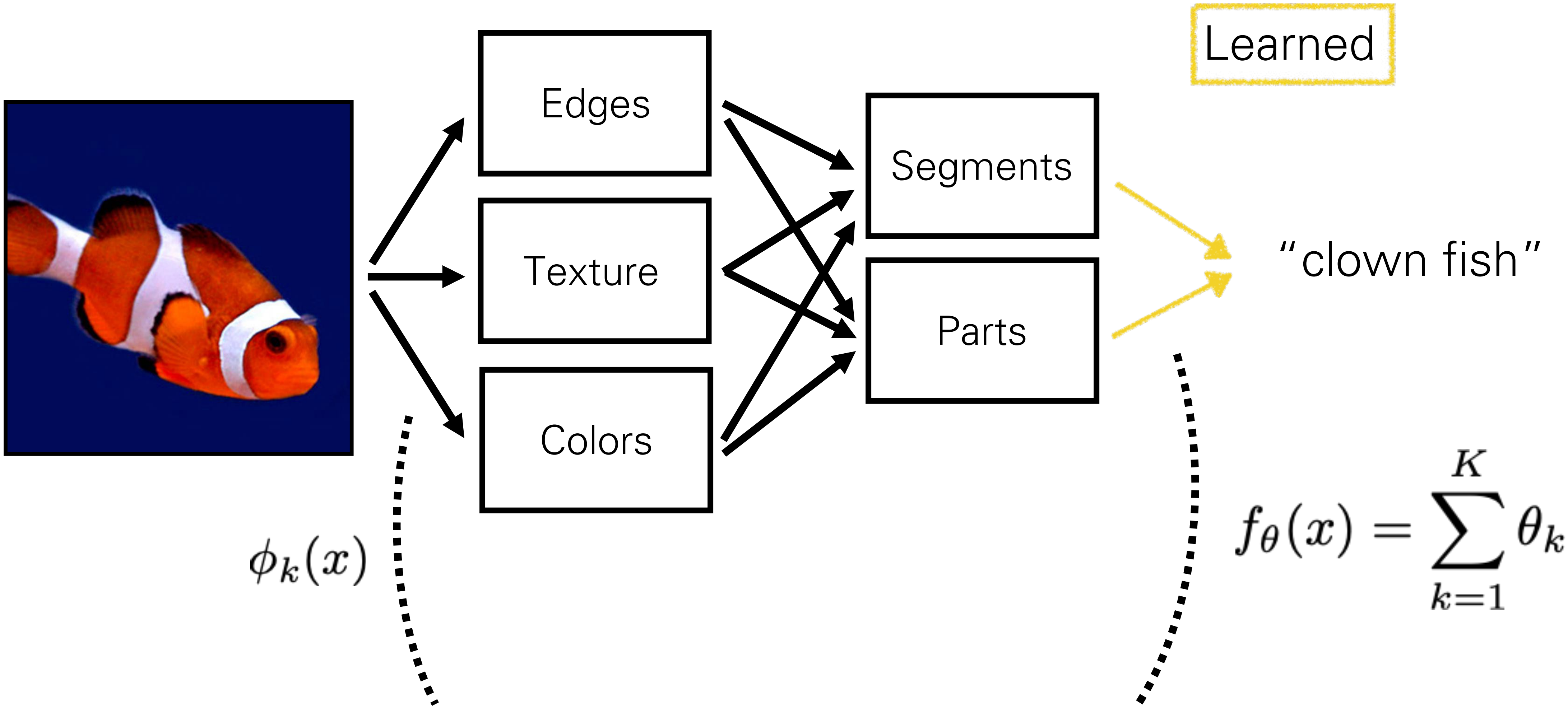


Object recognition



$$f_\theta(x) = \sum_{k=1}^{K} \theta_k \phi_k(x)$$

Object recognition



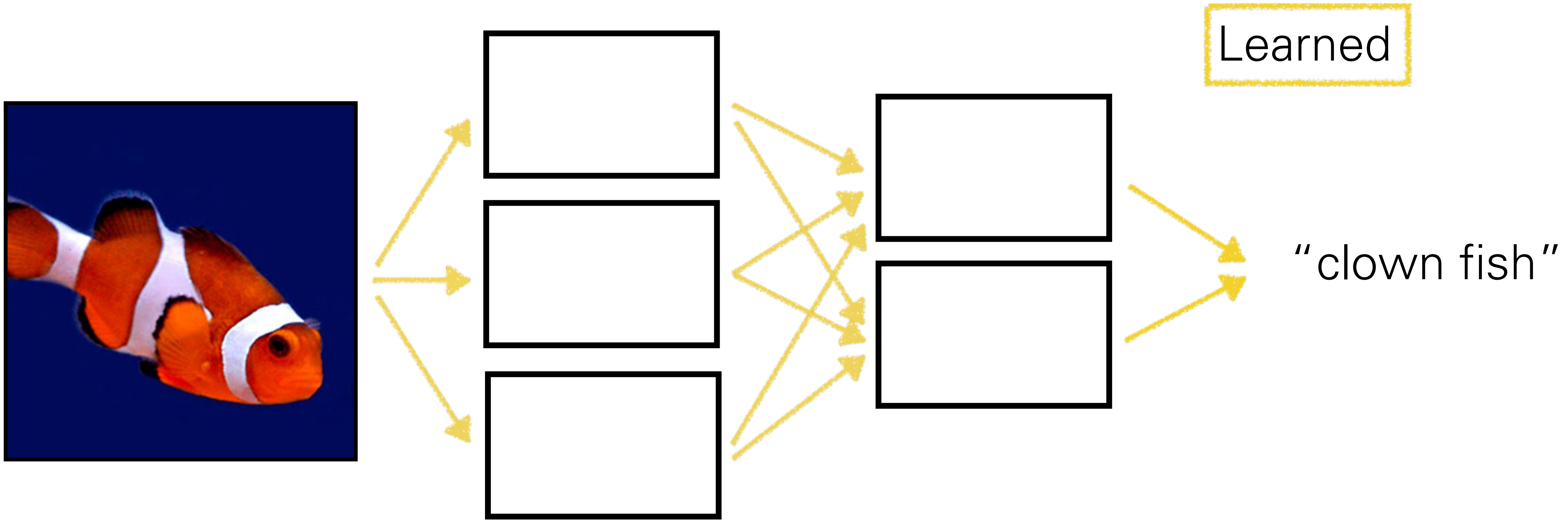
$\phi_k(x)$

$$f_\theta(x) = \sum_{k=1}^{K} \theta_k \phi_k(x)$$

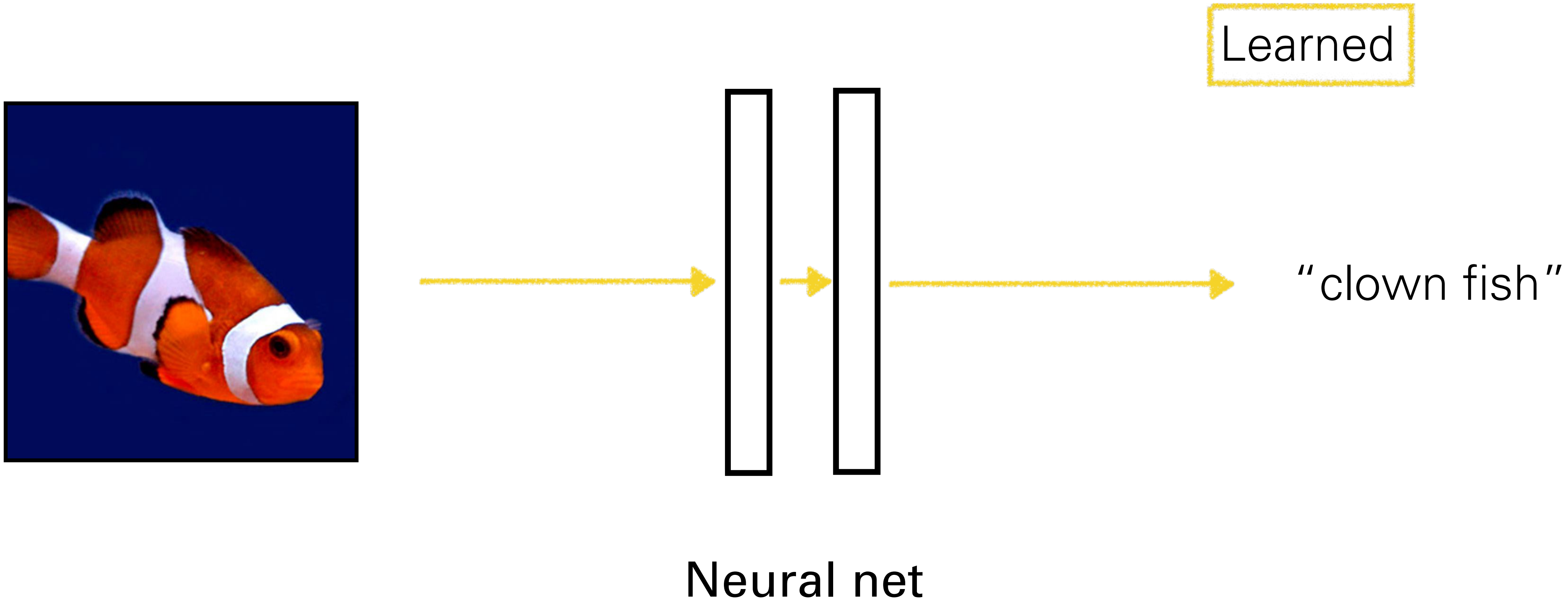
Feature extractors

Classifier

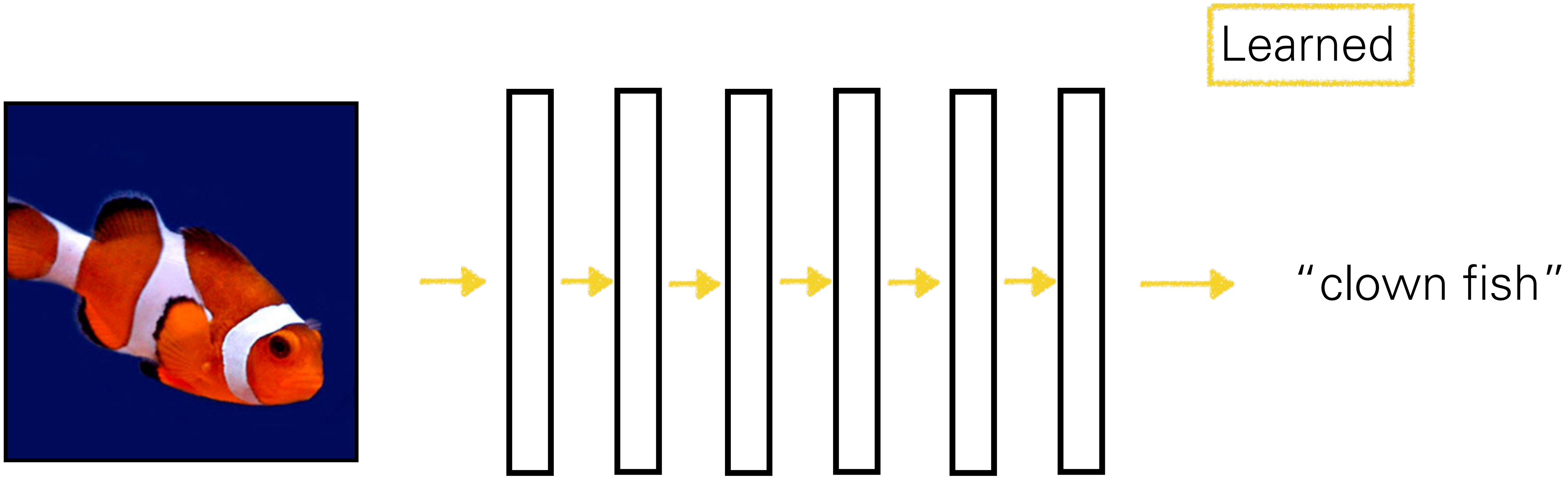
Object recognition



Object recognition

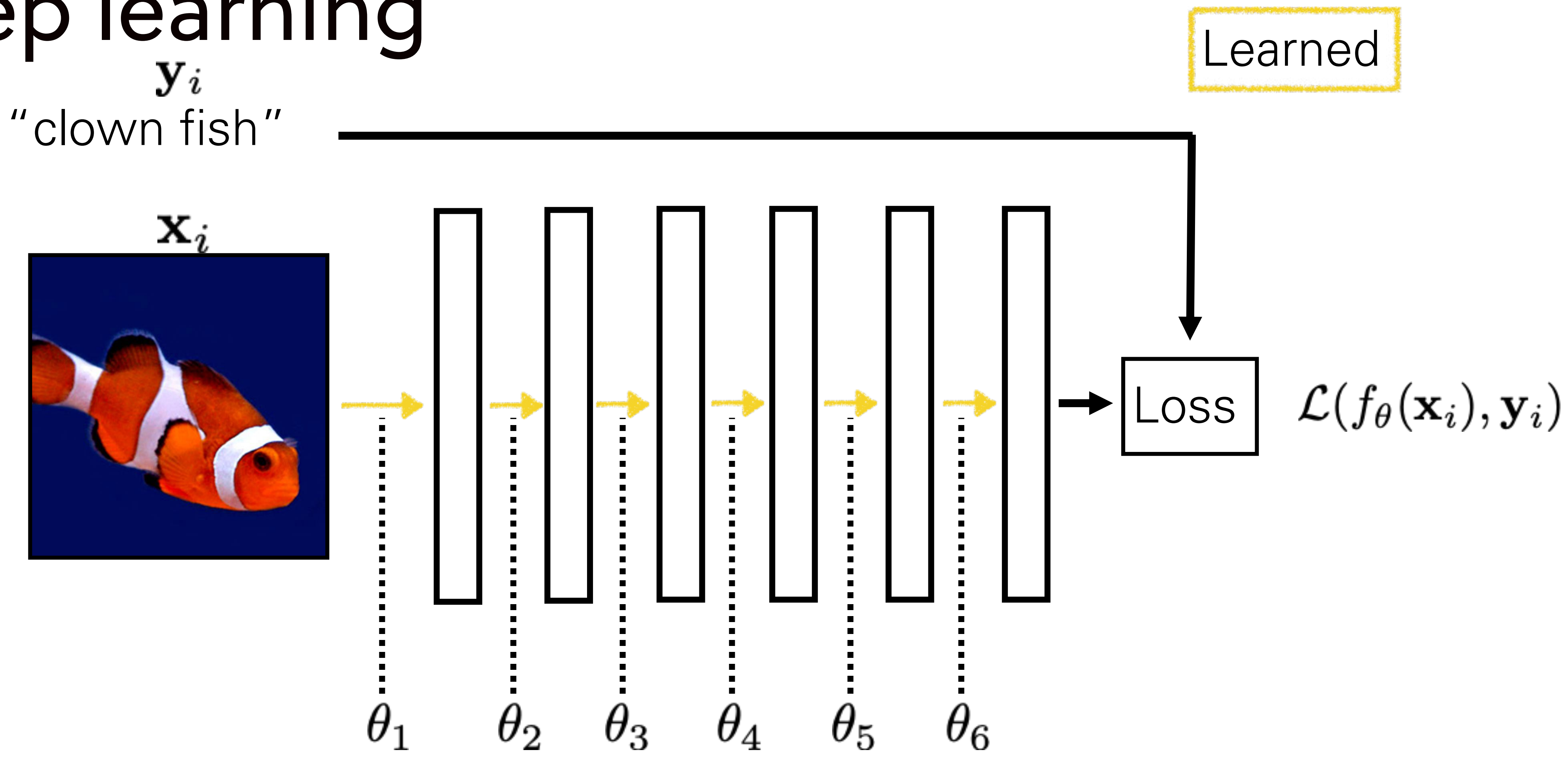


Object recognition



Deep neural net

Deep learning

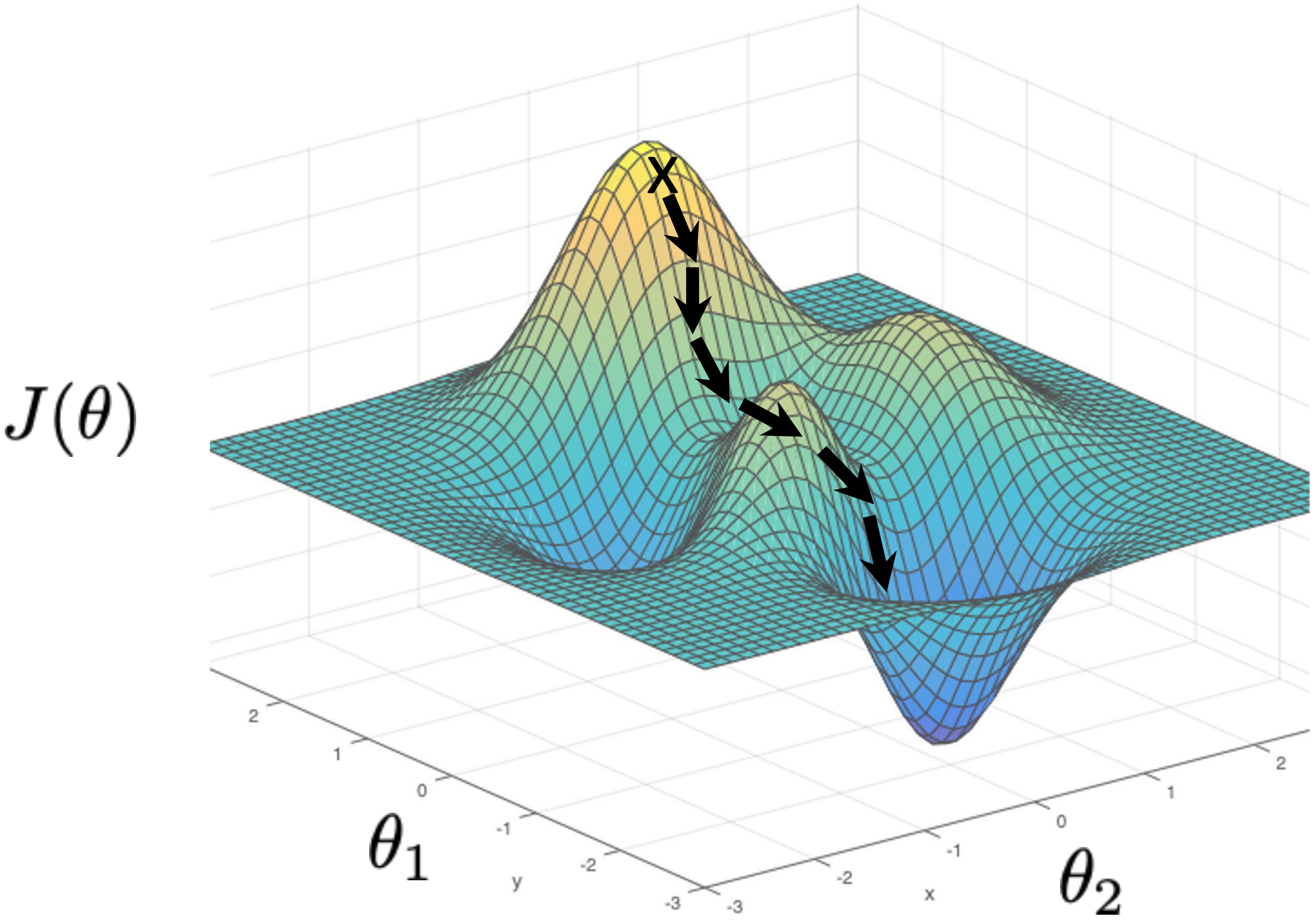


$$\theta^* = \arg\min_\theta \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)$$

Gradient descent

$$\theta^* = \arg\min_{\theta} \underbrace{\sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)}_{J(\theta)}$$

Gradient descent



$$\theta^* = \arg\min_{\theta} J(\theta)$$

Gradient descent

$$\theta^* = \arg\min_{\theta} \underbrace{\sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)}_{J(\theta)}$$

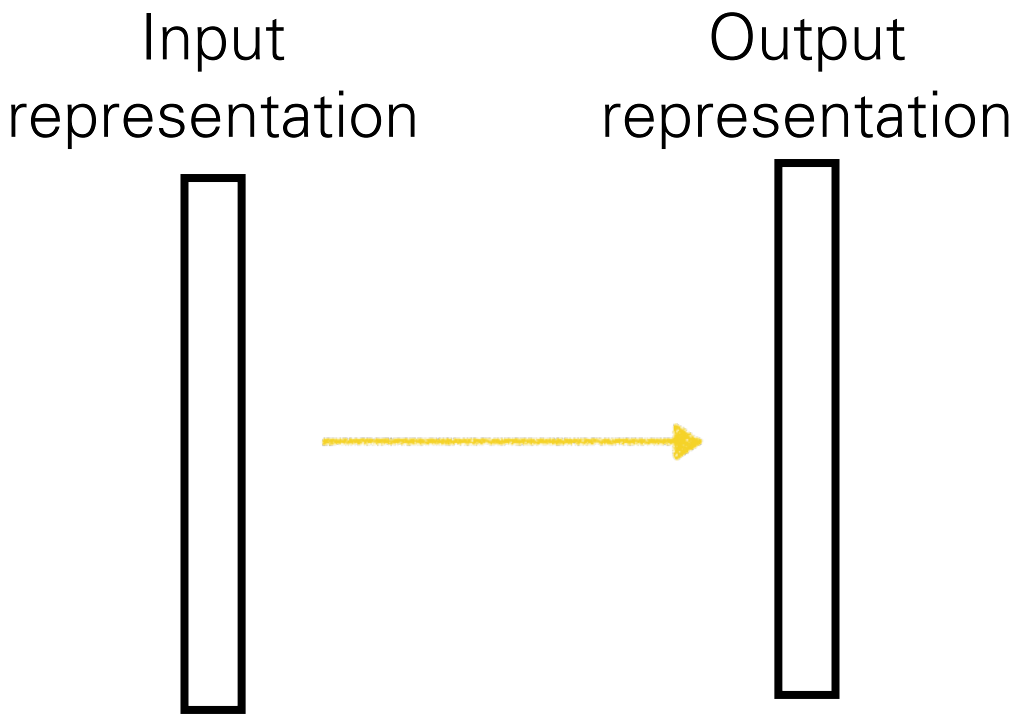
One iteration of gradient descent:

$$\theta^{t+1} = \theta^t - \eta_t \left.\frac{\partial J(\theta)}{\partial \theta}\right|_{\theta=\theta^t}$$

learning rate

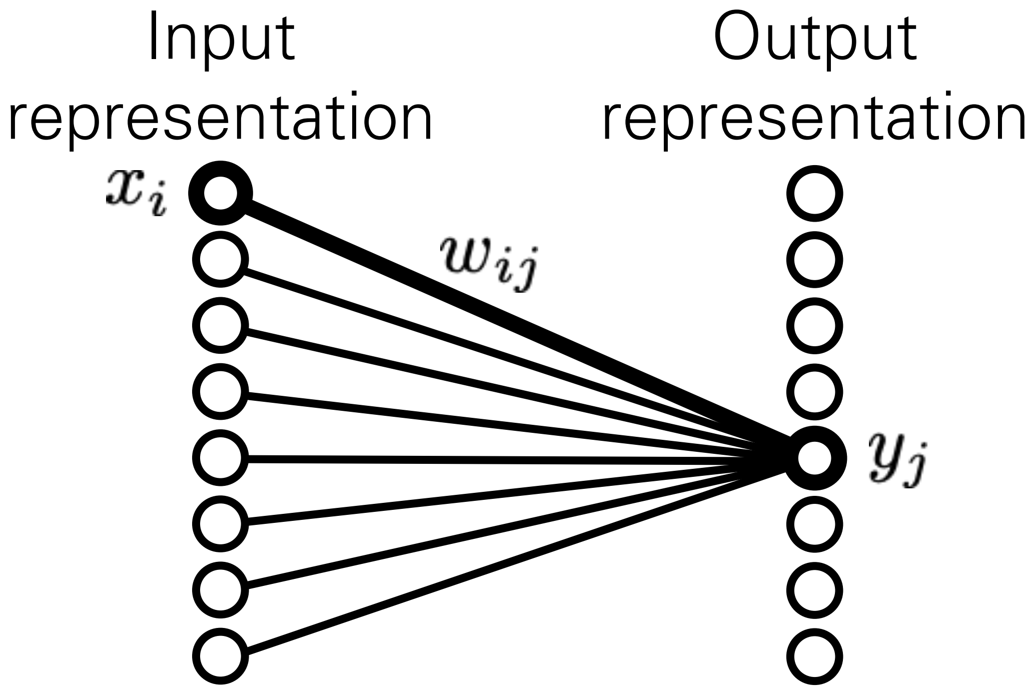
Computation in Neural Nets

Computation in a neural net



Computation in a neural net

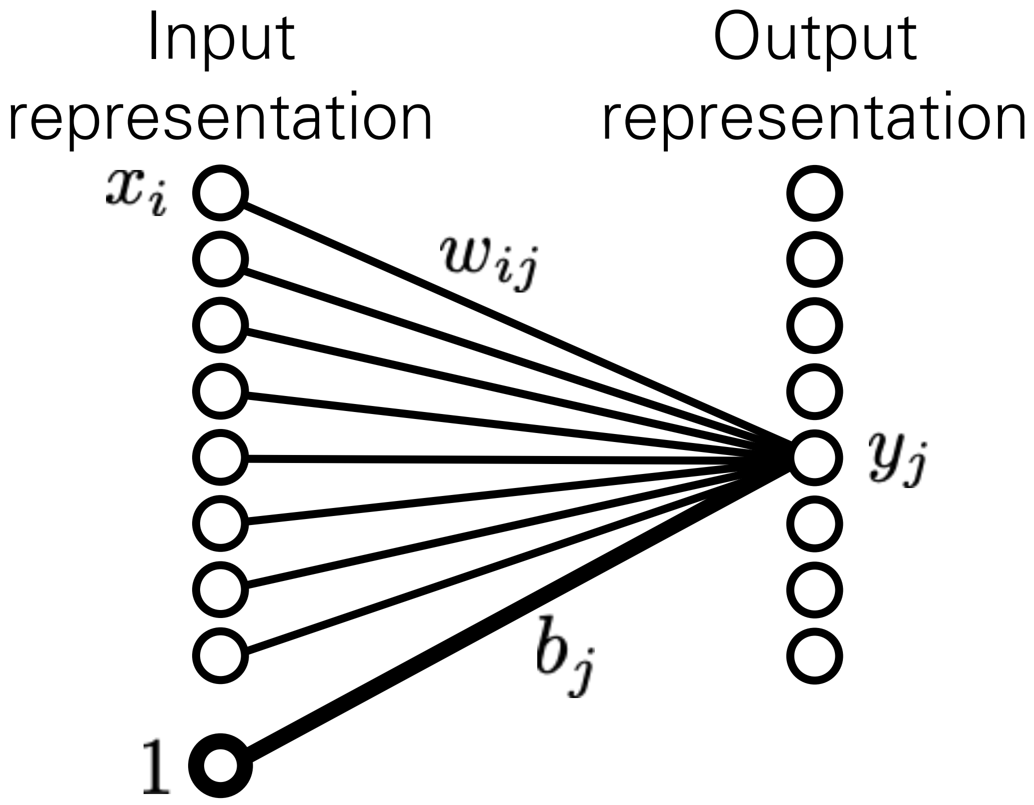
Linear layer



$$y_j = \sum_i w_{ij} x_i$$

Computation in a neural net

Linear layer



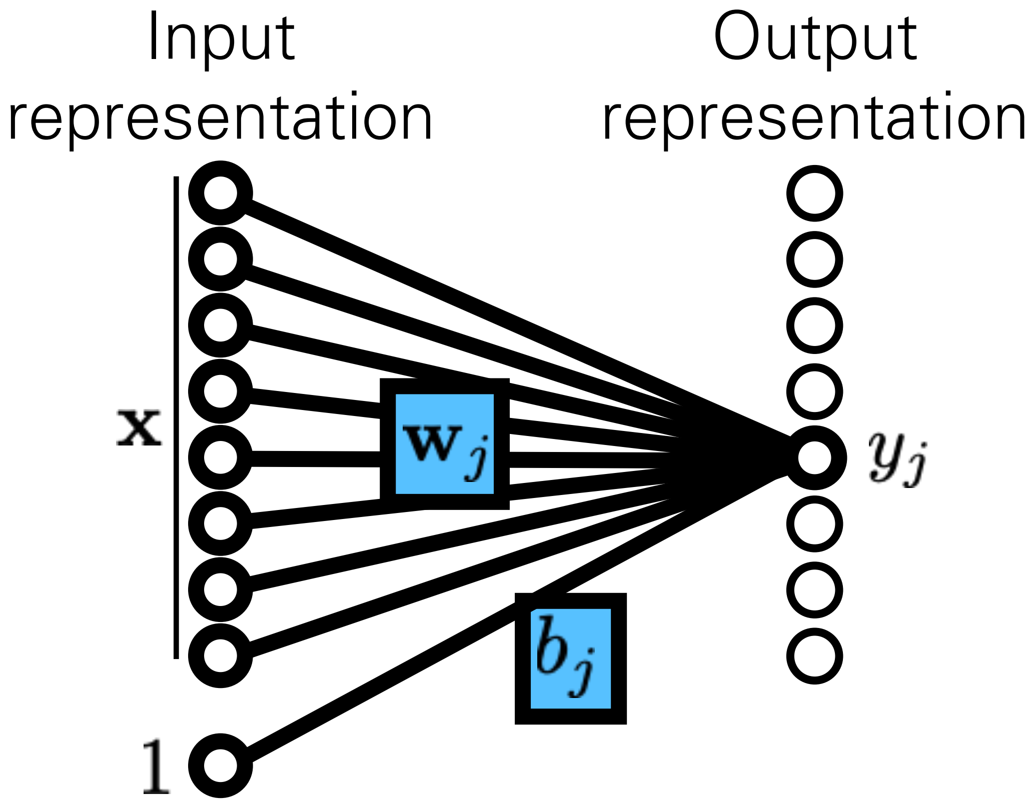
$$y_j = \sum_i w_{ij} x_i + b_j$$

weights

bias

Computation in a neural net

Linear layer



$$y_j = \mathbf{x}^T \mathbf{w}_j + b_j$$

weights

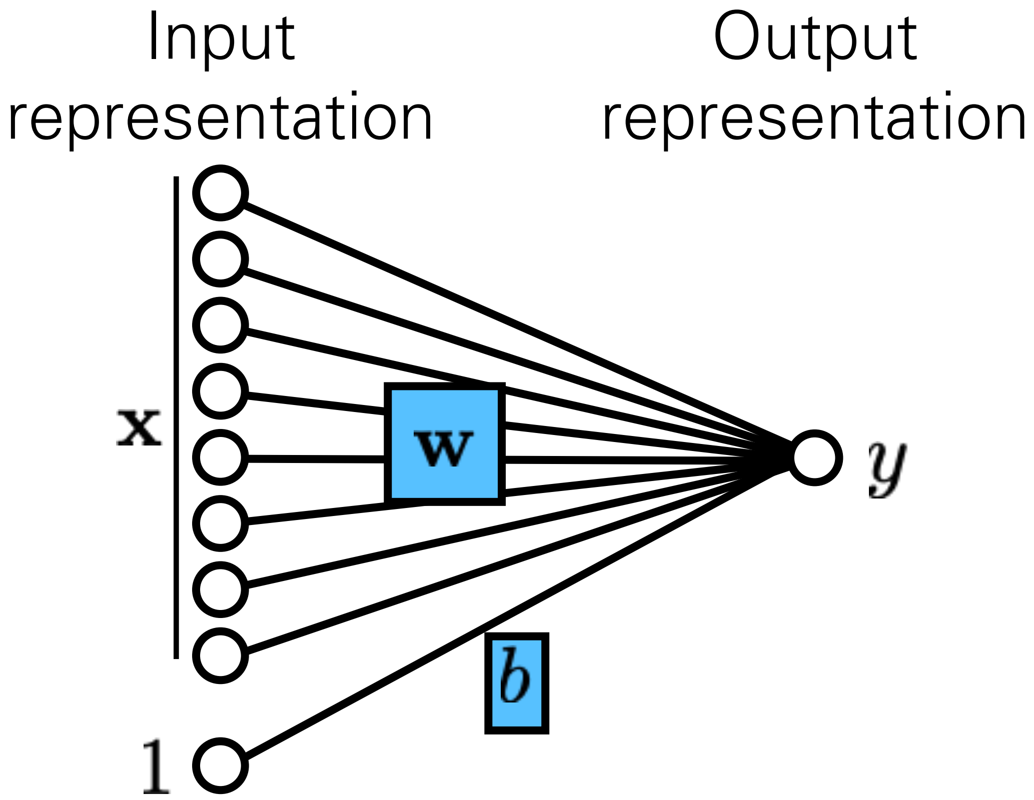
bias

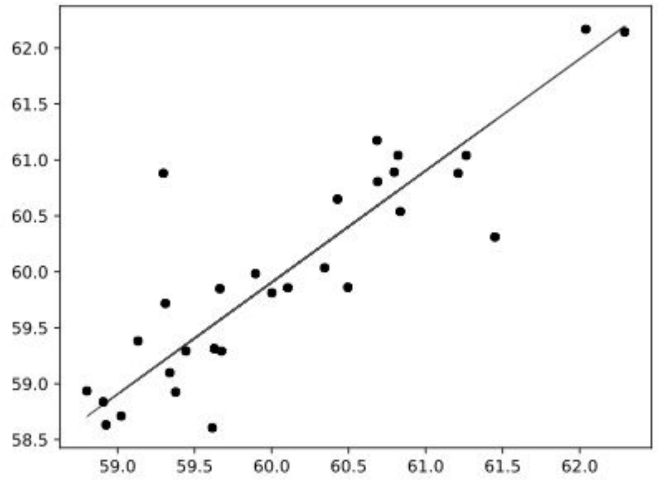
$$\theta = \{\mathbf{W}, \mathbf{b}\}$$

parameters of the model

Example: linear regression with a neural net

Linear layer

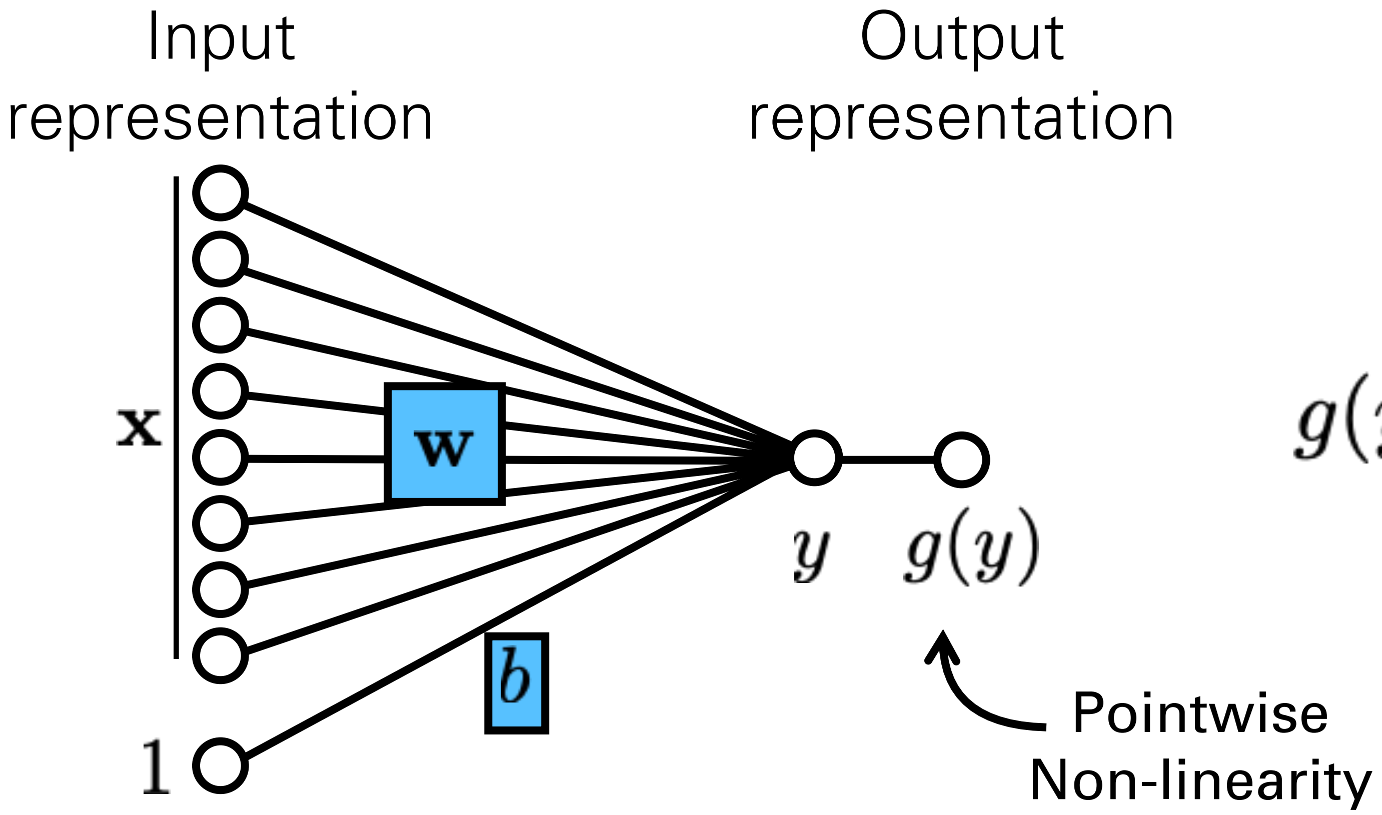




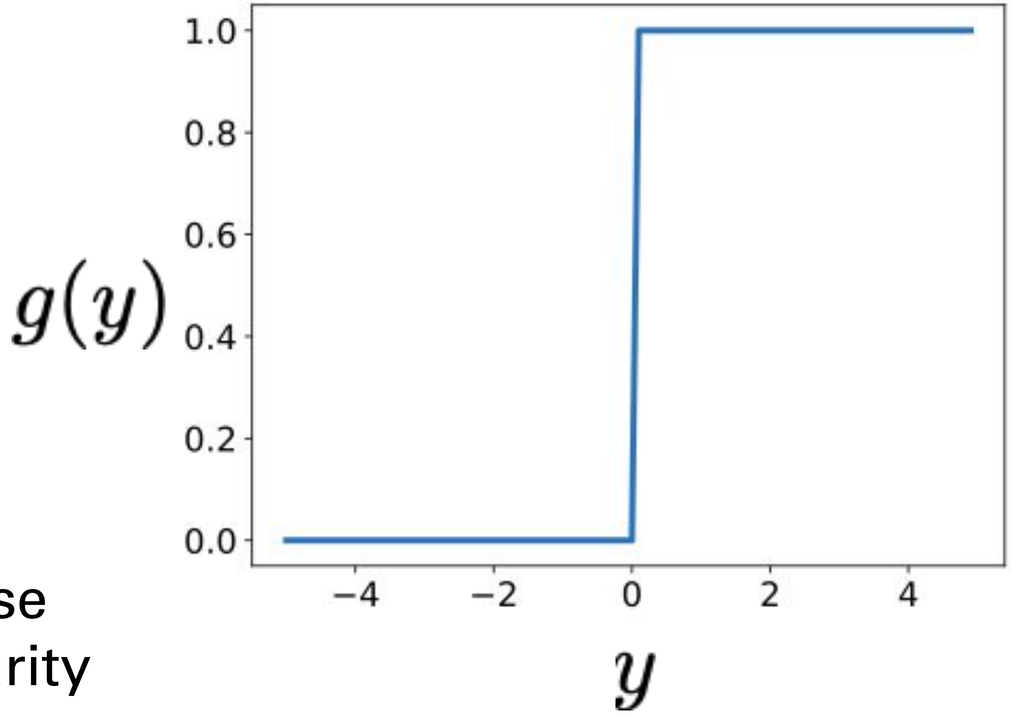
$$f_{\mathbf{w},b}(\mathbf{x}) = \mathbf{x}^T\mathbf{w} + b$$

Computation in a neural net

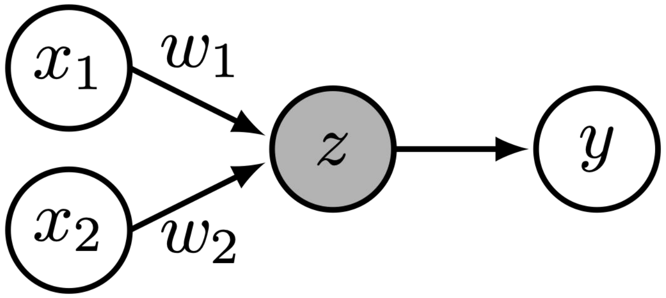
“Perceptron”



$$g(y) = \begin{cases} 1, & \text{if} \quad y > 0 \\ 0, & \text{otherwise} \end{cases}$$

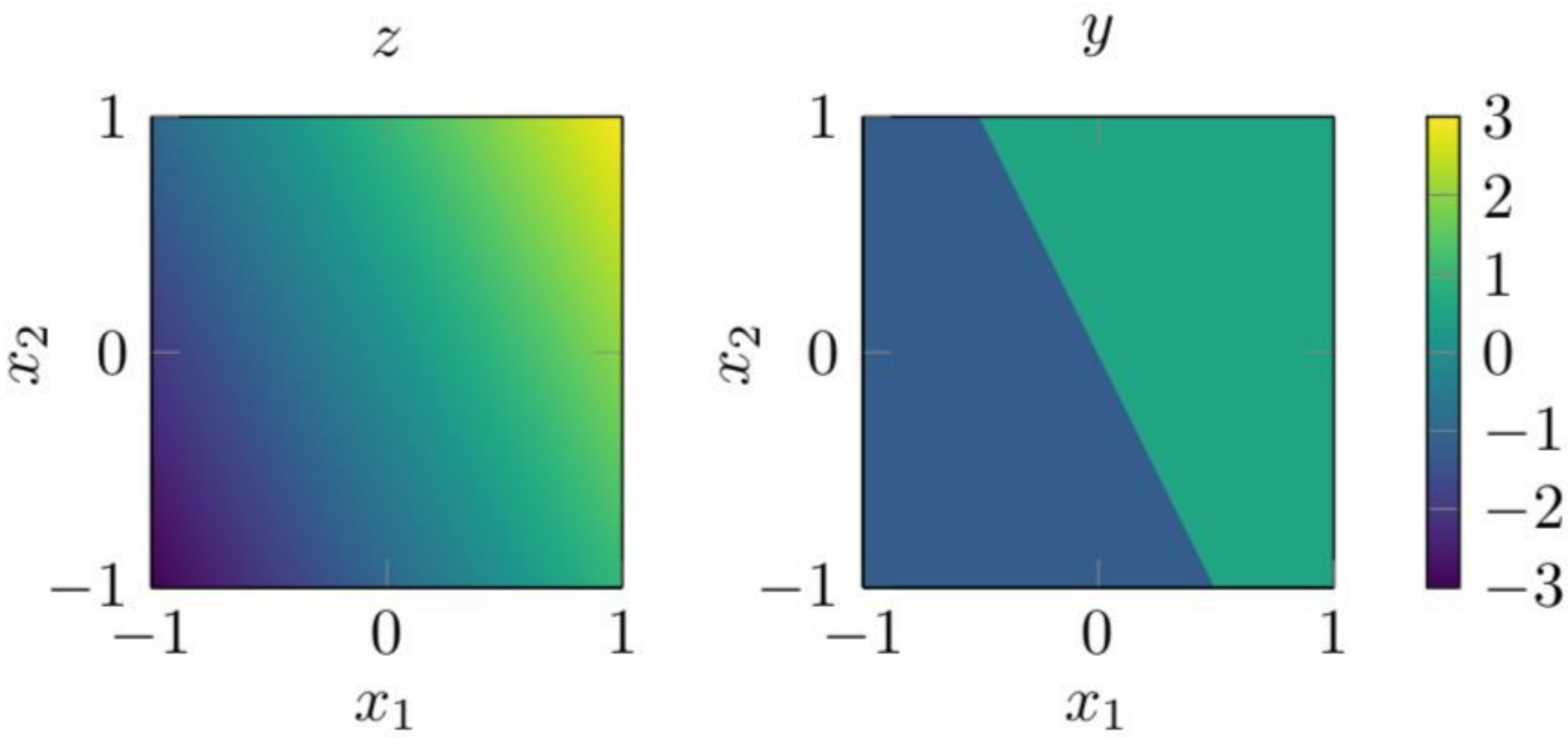


Example: linear classification with a perceptron



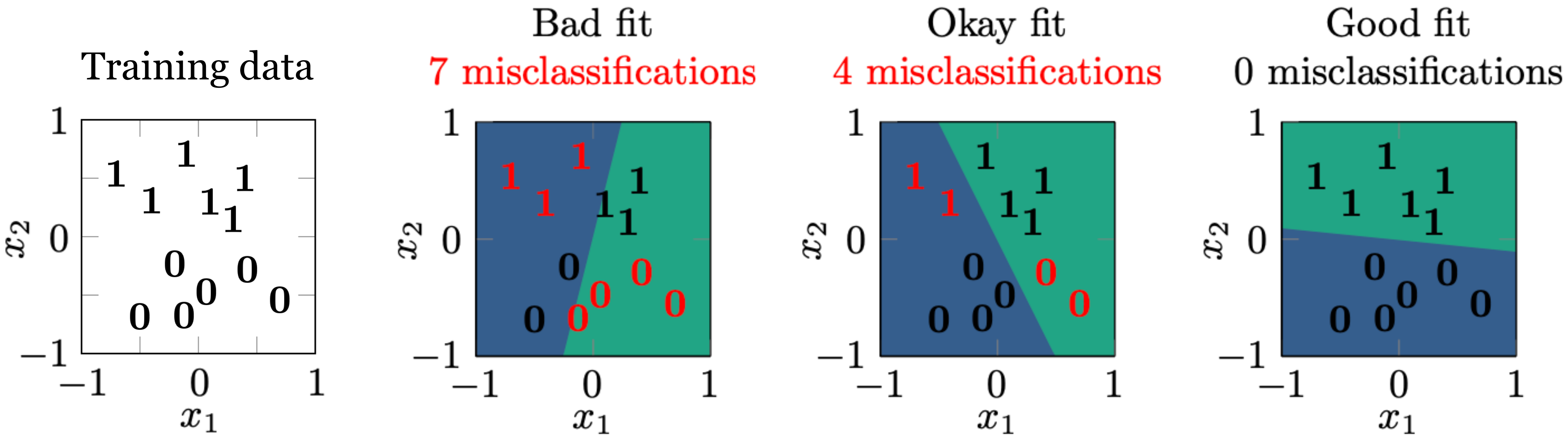
$$z = \mathbf{x}^T \mathbf{w} + b$$

$$y = g(z)$$



One layer neural net (perceptron) can perform linear classification!

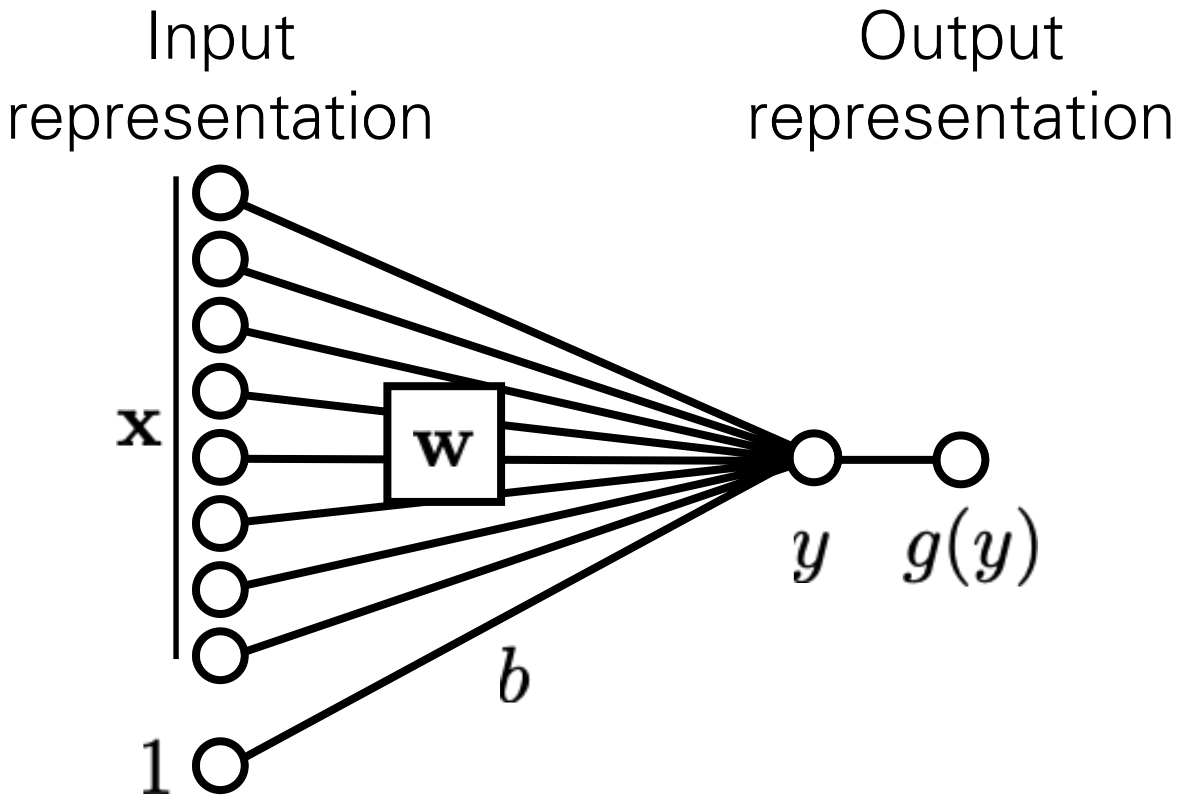
Example: linear classification with a perceptron

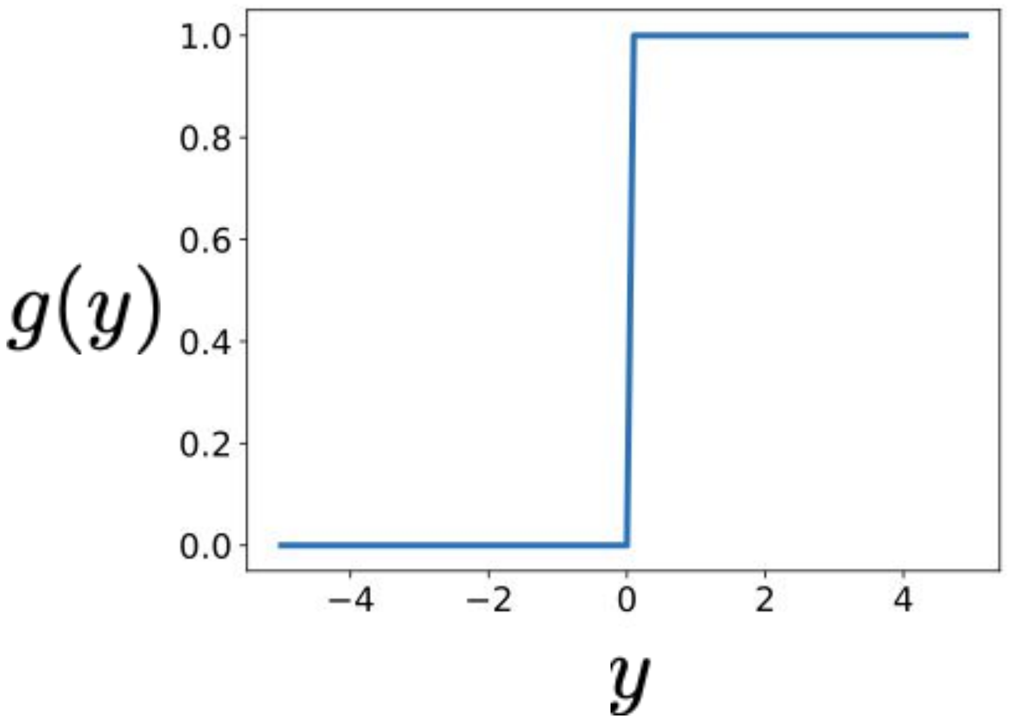


$$\mathbf{w}^*, b^* = \underset{\mathbf{w}, b}{\arg\min} \sum_{i=1}^{N} \mathcal{L}(g(z^{(i)}), y^{(i)})$$

Computation in a neural net

$$g(y) = \begin{cases} 1, & \text{if} \quad y > 0 \\ 0, & \text{otherwise} \end{cases}$$

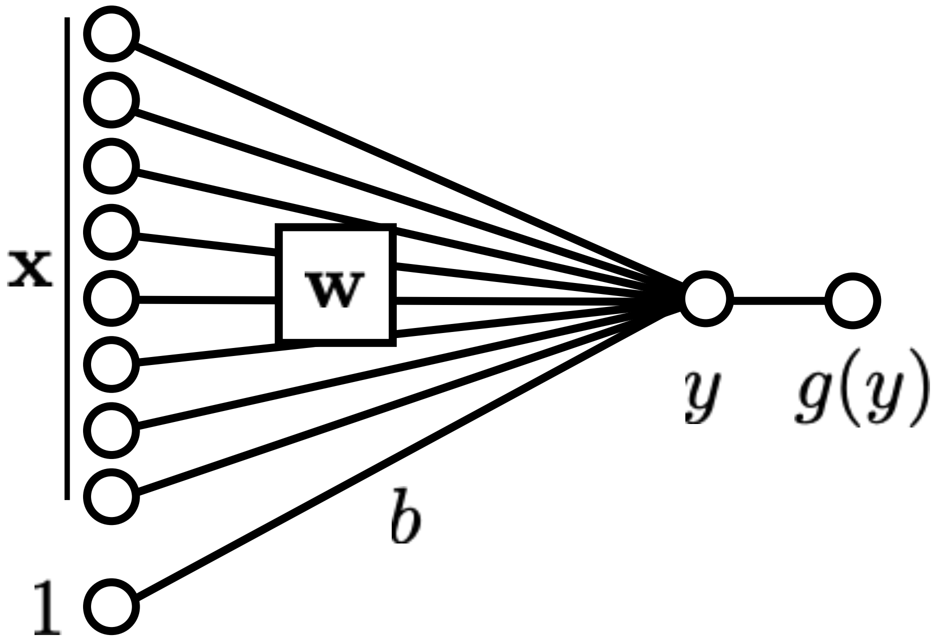




Computation in a neural net – nonlinearity

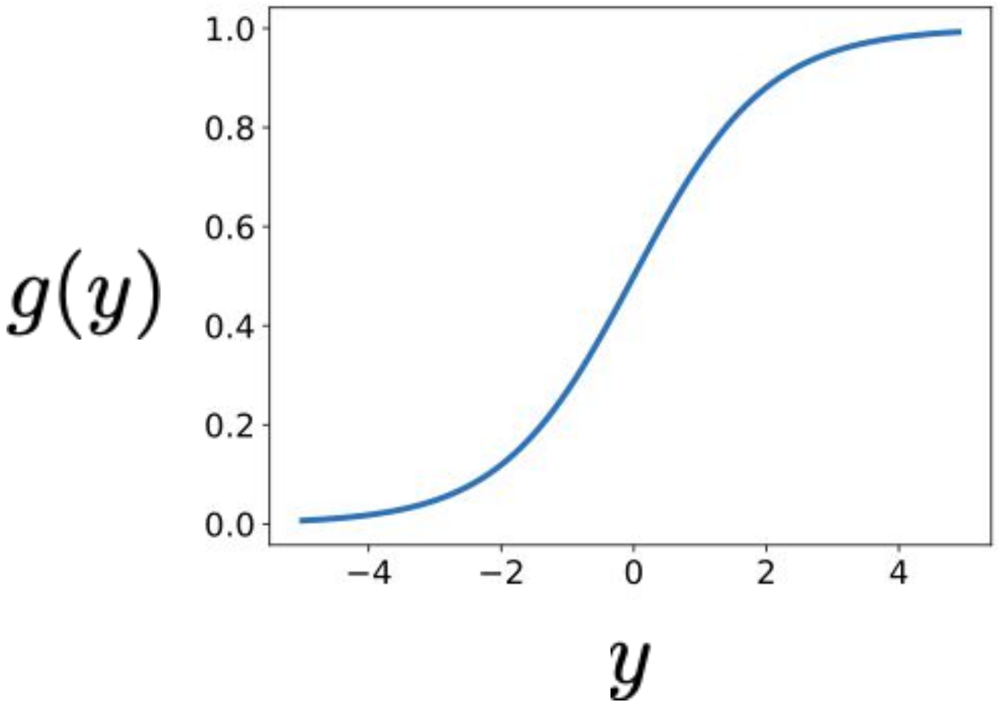
Input representation

Output representation



Sigmoid

$$g(y) = \frac{1}{1 + e^{-y}}$$

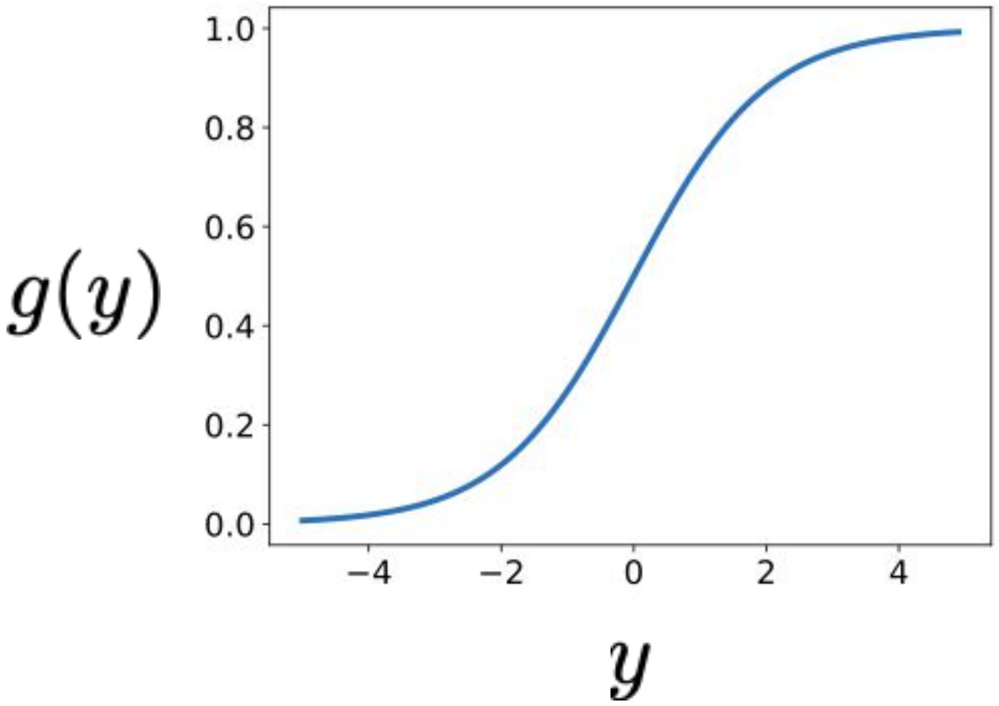


Computation in a neural net – nonlinearity

- Interpretation as firing rate of neuron
- Bounded between [0,1]
- Saturation for large +/- inputs
- Gradients go to zero
- Outputs centered at 0.5
  (poor conditioning)
- Not used in practice

Sigmoid

$$g(y) = \frac{1}{1 + e^{-y}}$$



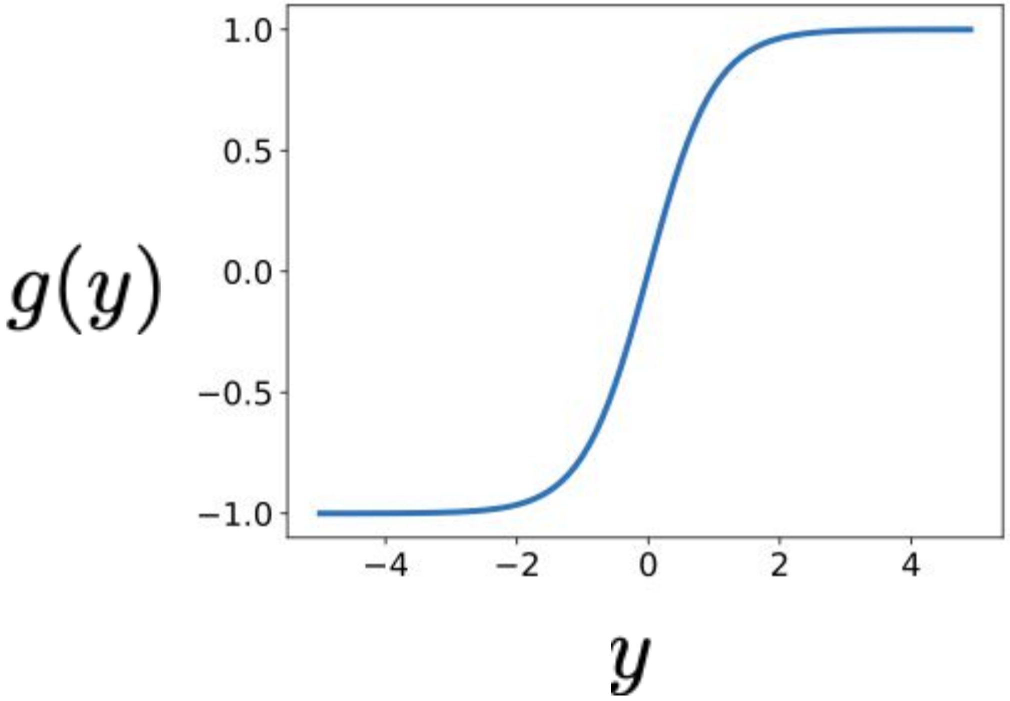
Computation in a neural net – nonlinearity

- Bounded between [-1,+1]
- Saturation for large +/- inputs
- Gradients go to zero
- Outputs centered at 0
- Preferable to sigmoid

$$\tanh(x) = 2\,\text{sigmoid}(2x) - 1$$

Tanh

$$g(y) = \frac{e^y - e^{-y}}{e^y + e^{-y}}$$



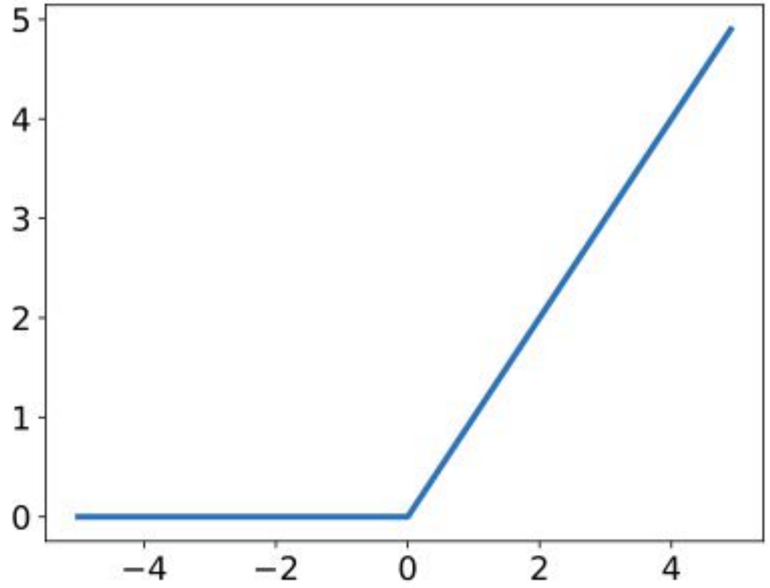
Computation in a neural net – nonlinearity

- Unbounded output (on positive side)
- Efficient to implement: $\frac{\partial g}{\partial y} = \begin{cases} 0, & \text{if} \quad y < 0 \\ 1, & \text{if} \quad y \geq 0 \end{cases}$
- Also seems to help convergence (see 6x speedup vs tanh in [Krizhevsky et al.])
- Drawback: if strongly in negative region, unit is dead forever (no gradient).
- Default choice: widely used in current models.

Rectified linear unit (ReLU)

$$g(y) = \max(0, y)$$

$g(y)$



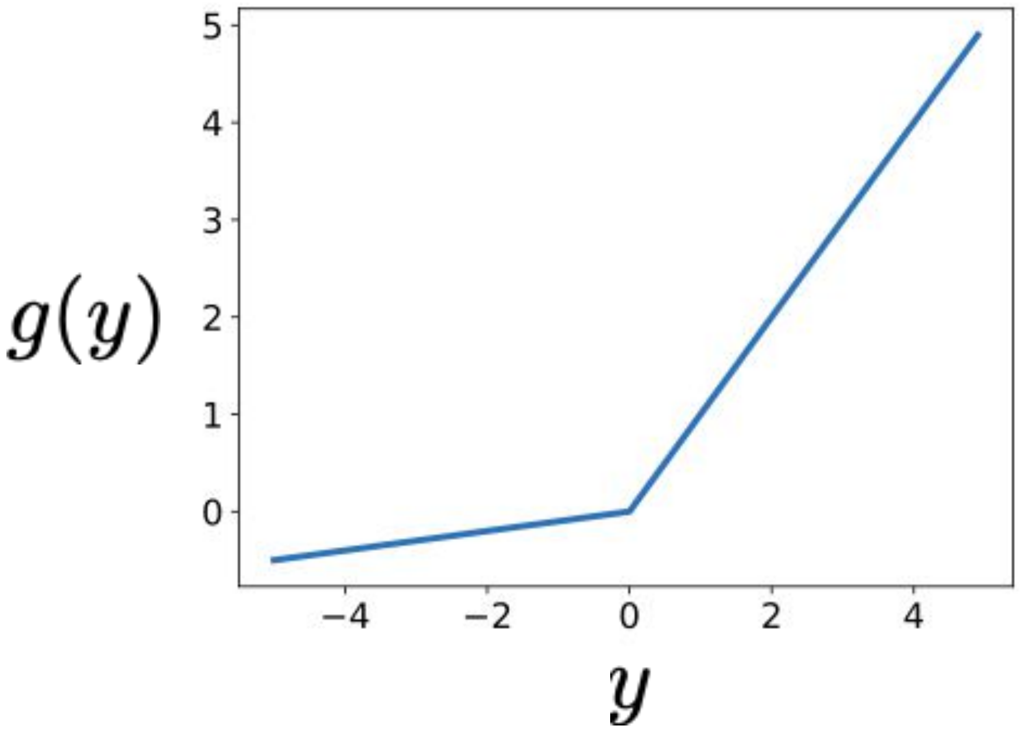
y

Computation in a neural net – nonlinearity

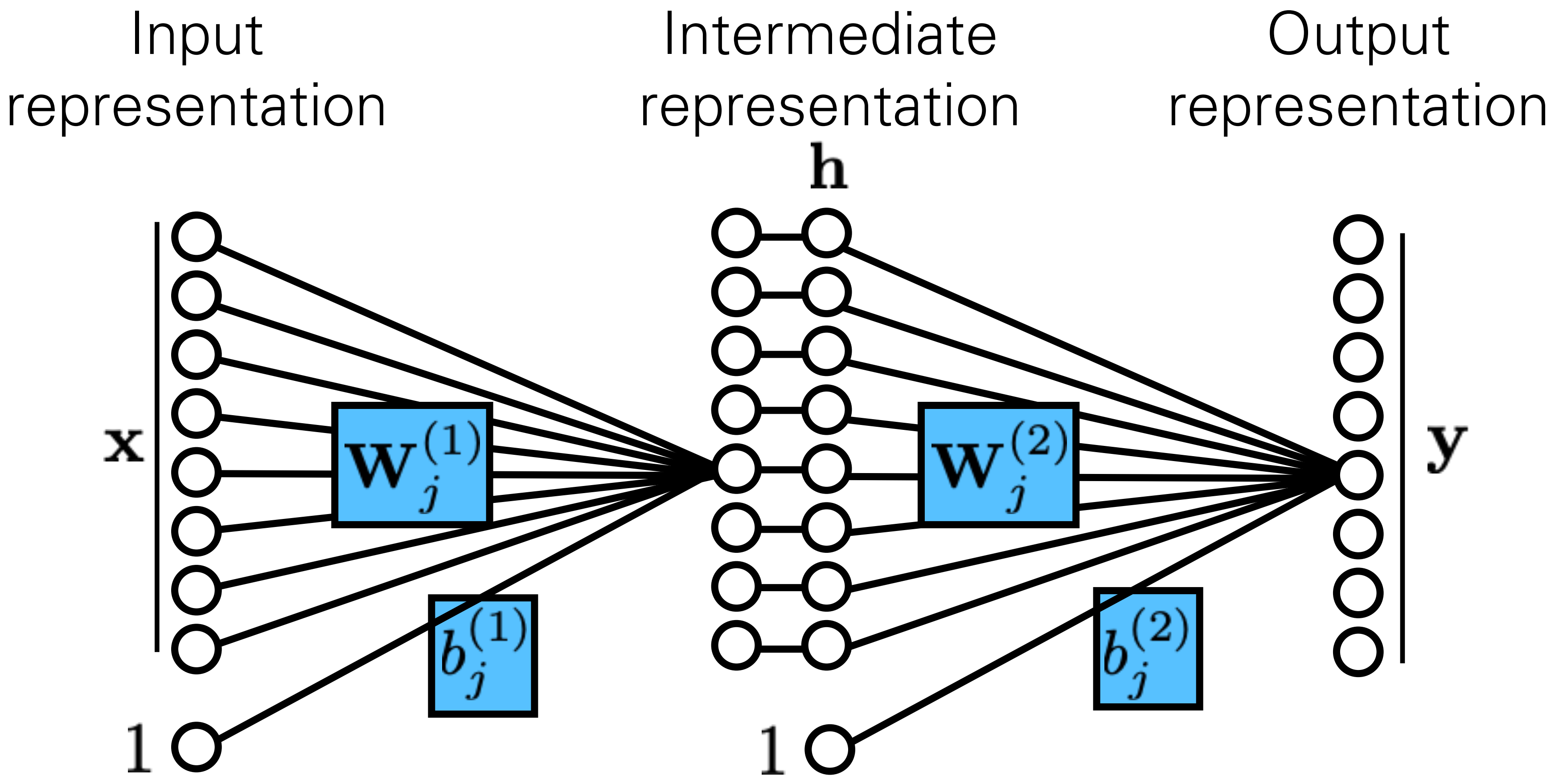
- where α is small (e.g. 0.02)
- Efficient to implement: $\frac{\partial g}{\partial y} = \begin{cases} -a, & \text{if} \quad y < 0 \\ 1, & \text{if} \quad y \geq 0 \end{cases}$
- Also known as probabilistic ReLU (PReLU)
- Has non-zero gradients everywhere (unlike ReLU)
- α can also be learned (see Kaiming He et al. 2015).

Leaky ReLU

$$g(y) = \begin{cases} \max(0, y), & \text{if} \quad y \geq 0 \\ a\min(0, y), & \text{if} \quad y < 0 \end{cases}$$

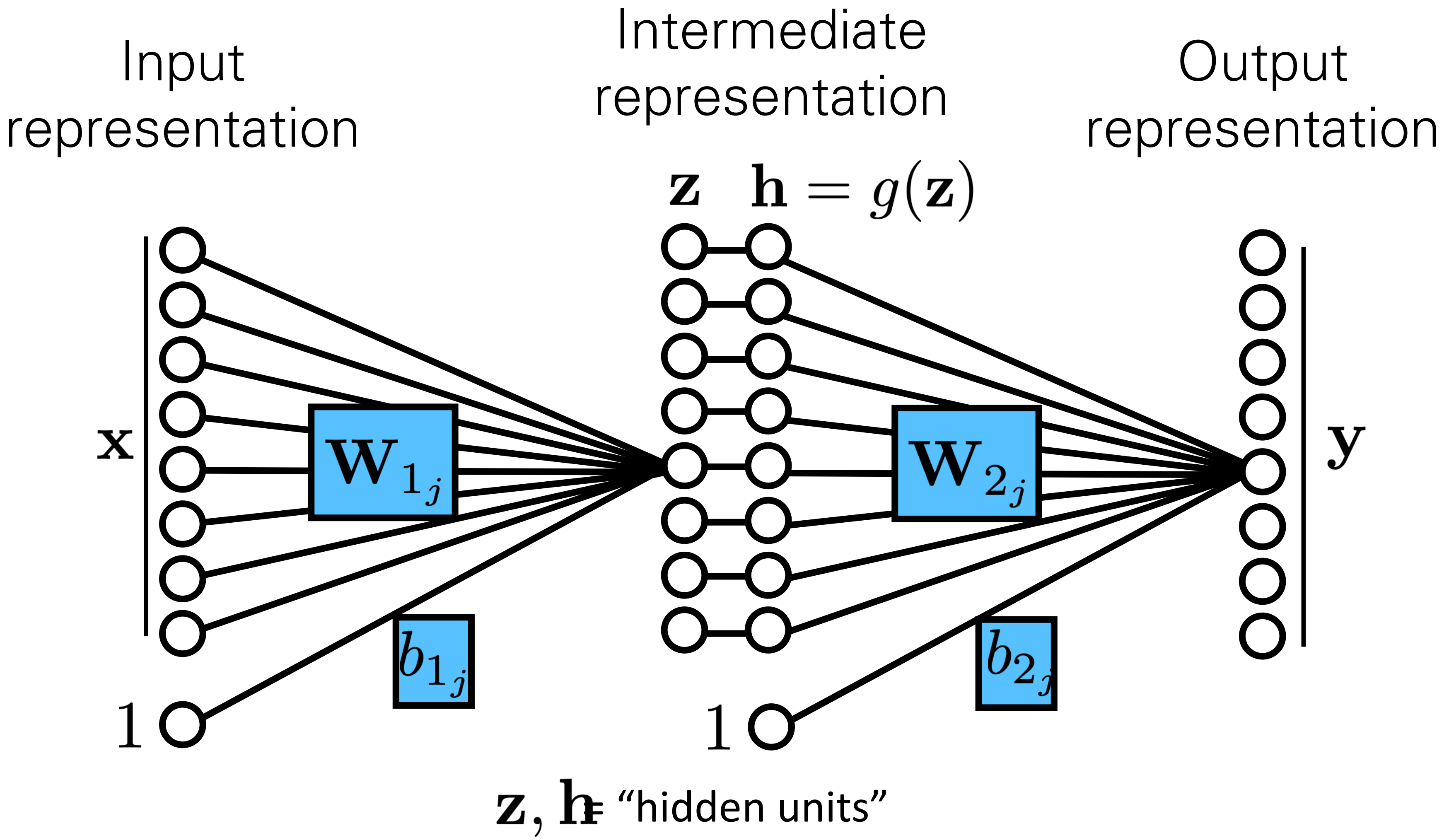


Stacking layers



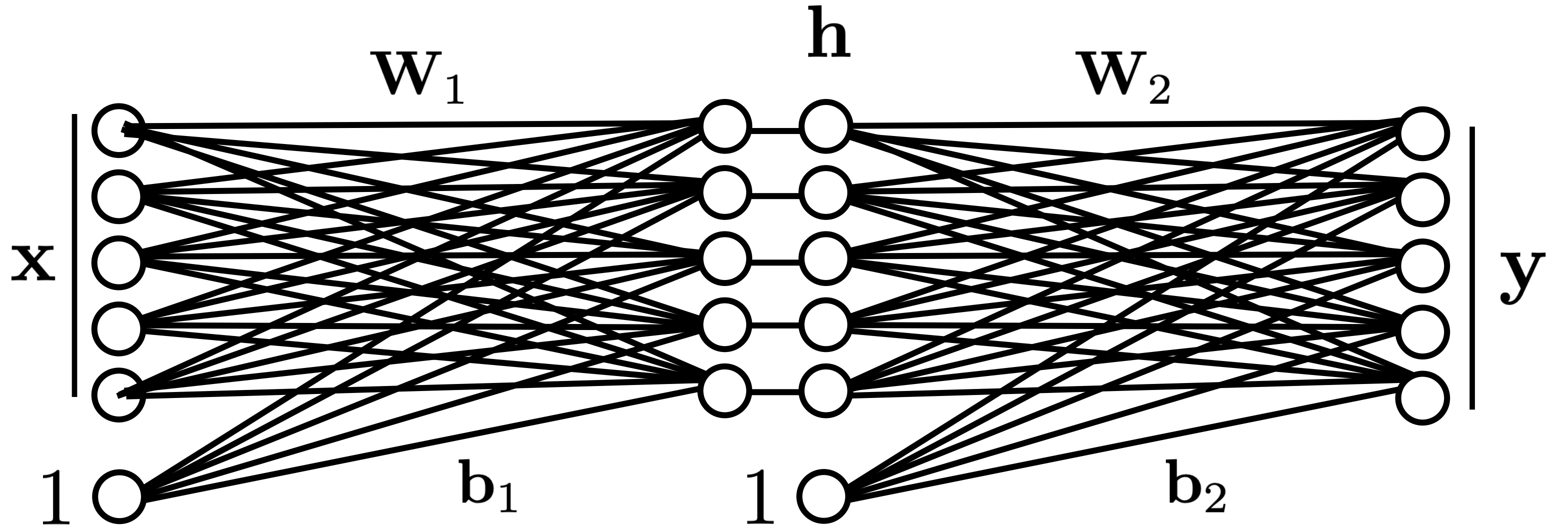
$\mathbf{h}$ = “hidden units”

Stacking layers



Stacking layers

Input representation | Intermediate representation | Output representation



$$\mathbf{h} = g(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \qquad \mathbf{y} = g(\mathbf{W}_2\mathbf{h} + \mathbf{b}_2)$$

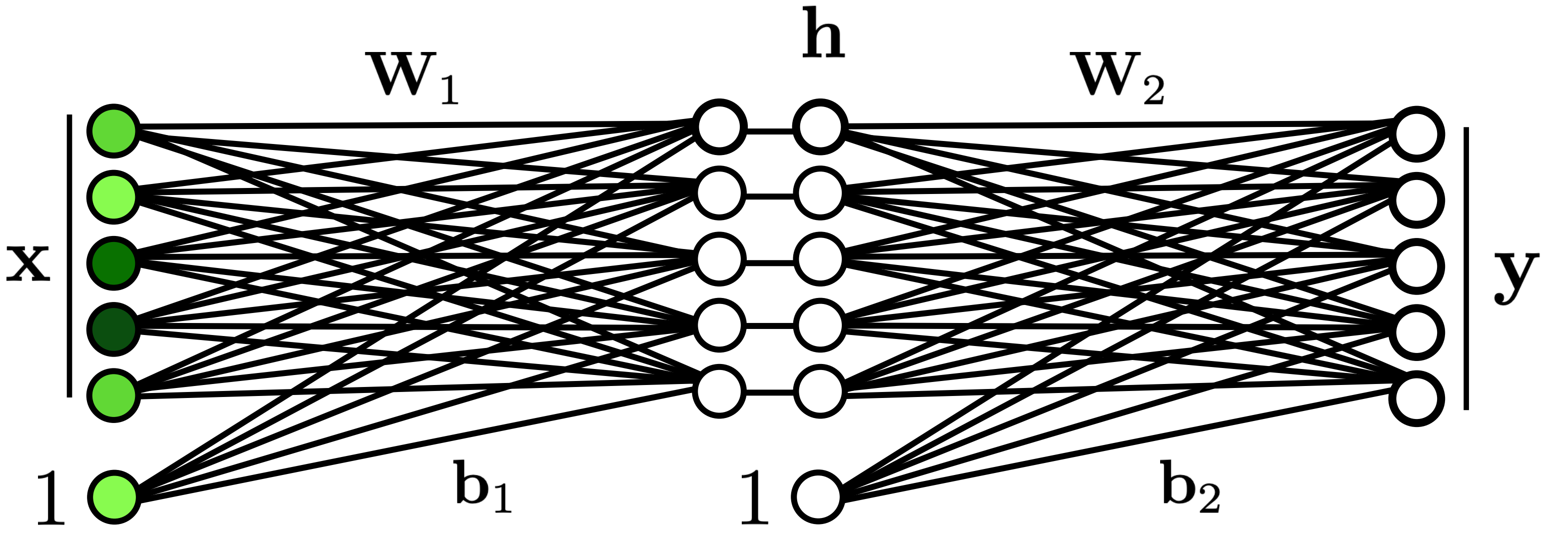
$$\theta = \{\mathbf{W}_1, \ldots, \mathbf{W}_L, \mathbf{b}_1, \ldots, \mathbf{b}_L\}$$

Stacking layers

Input representation

Intermediate representation

Output representation



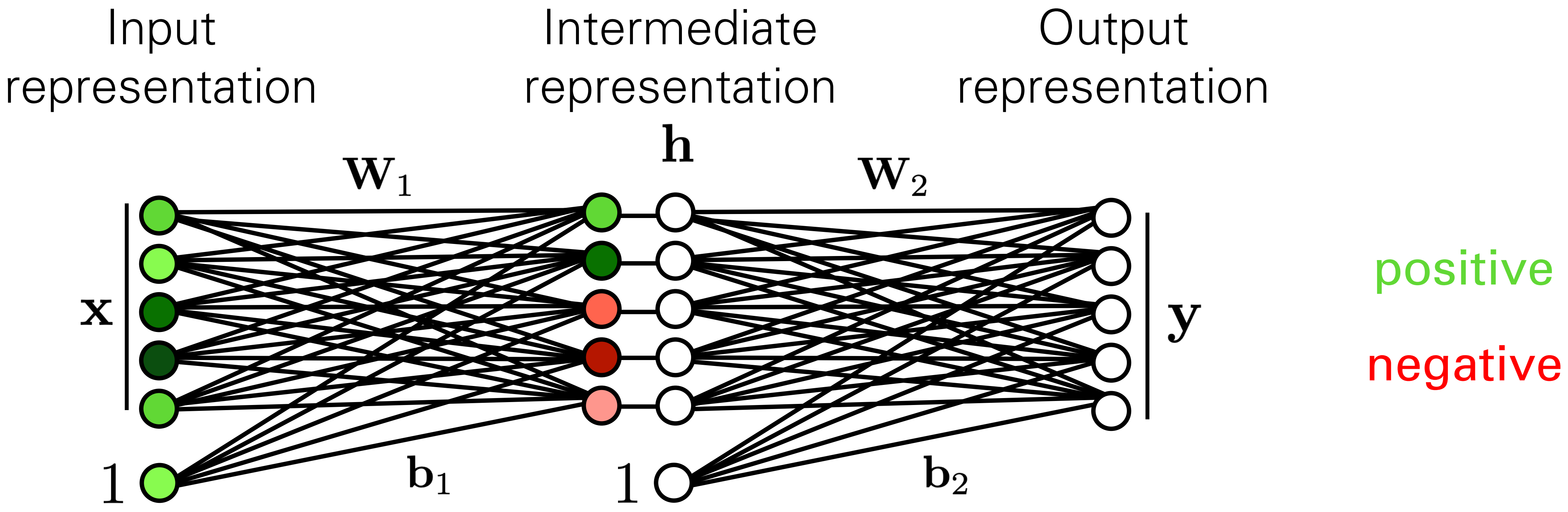
positive

negative

$$\mathbf{h} = g(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \qquad \mathbf{y} = g(\mathbf{W}_2\mathbf{h} + \mathbf{b}_2)$$

$$\theta = \{\mathbf{W}_1, \ldots, \mathbf{W}_L, \mathbf{b}_1, \ldots, \mathbf{b}_L\}$$

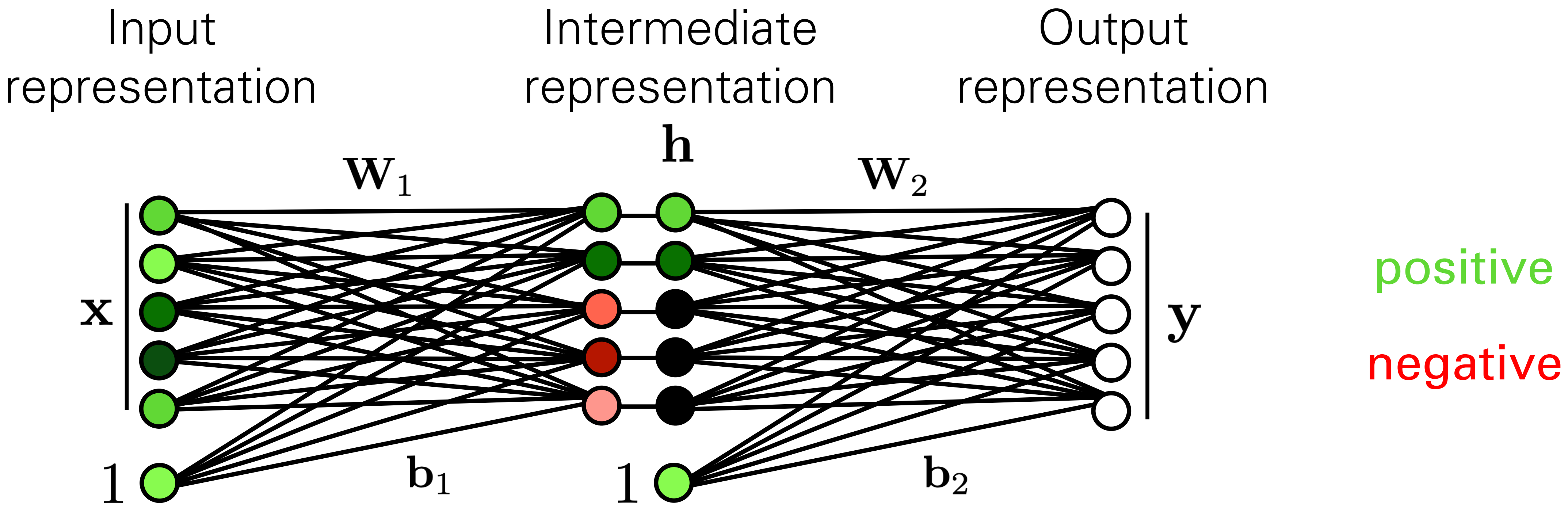
Stacking layers



$$\mathbf{h} = g(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \qquad \mathbf{y} = g(\mathbf{W}_2\mathbf{h} + \mathbf{b}_2)$$

$$\theta = \{\mathbf{W}_1, \ldots, \mathbf{W}_L, \mathbf{b}_1, \ldots, \mathbf{b}_L\}$$

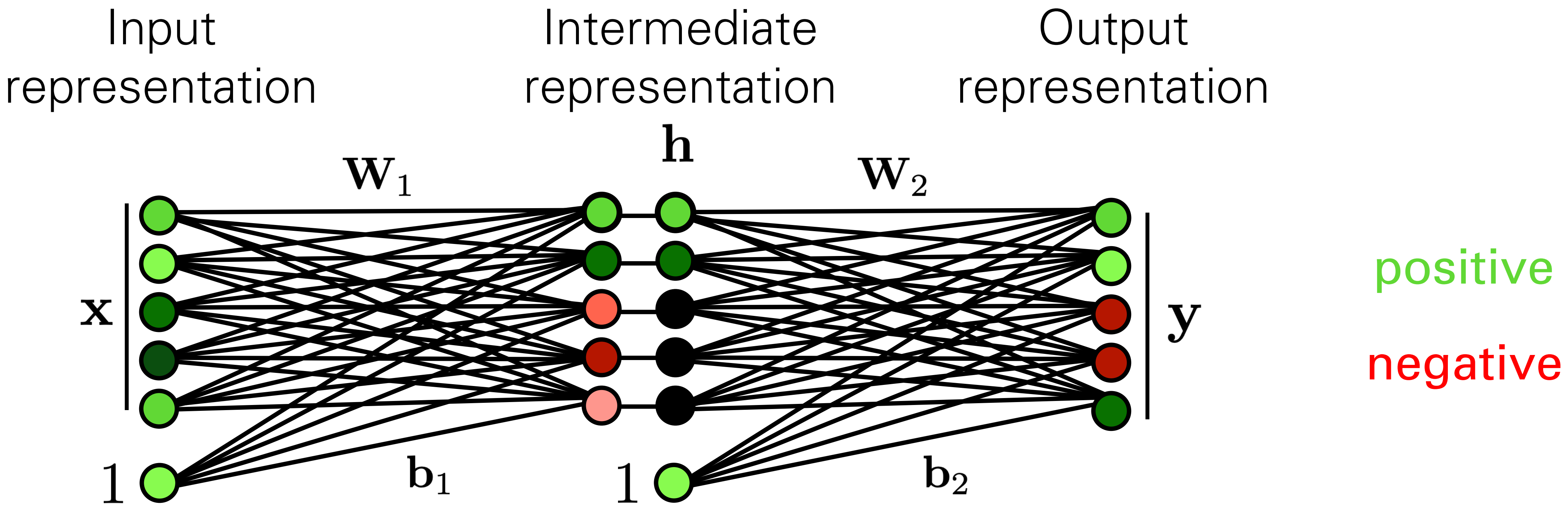
Stacking layers



$$\mathbf{h} = g(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \qquad \mathbf{y} = g(\mathbf{W}_2\mathbf{h} + \mathbf{b}_2)$$

$$\theta = \{\mathbf{W}_1, \ldots, \mathbf{W}_L, \mathbf{b}_1, \ldots, \mathbf{b}_L\}$$

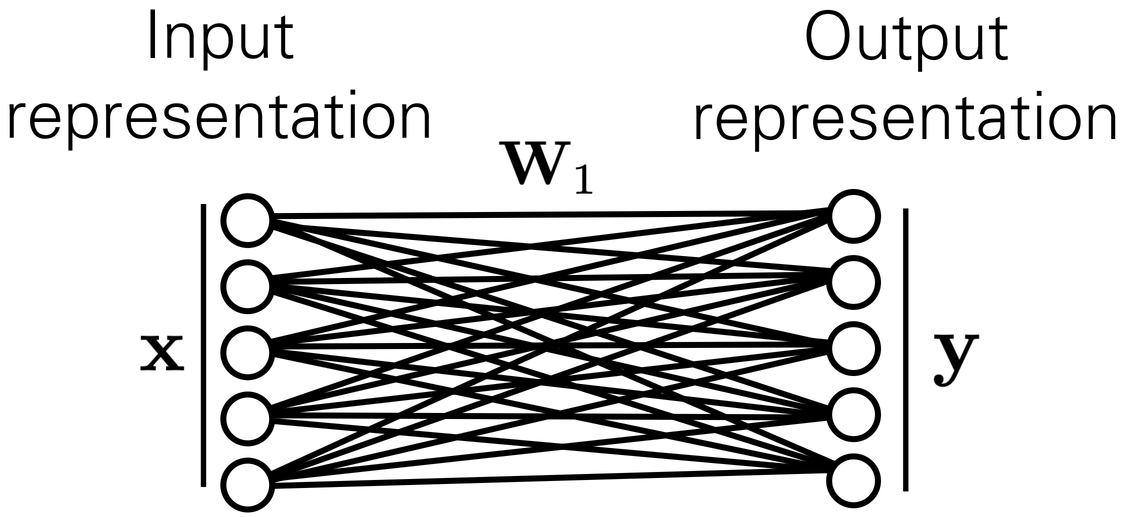
Stacking layers



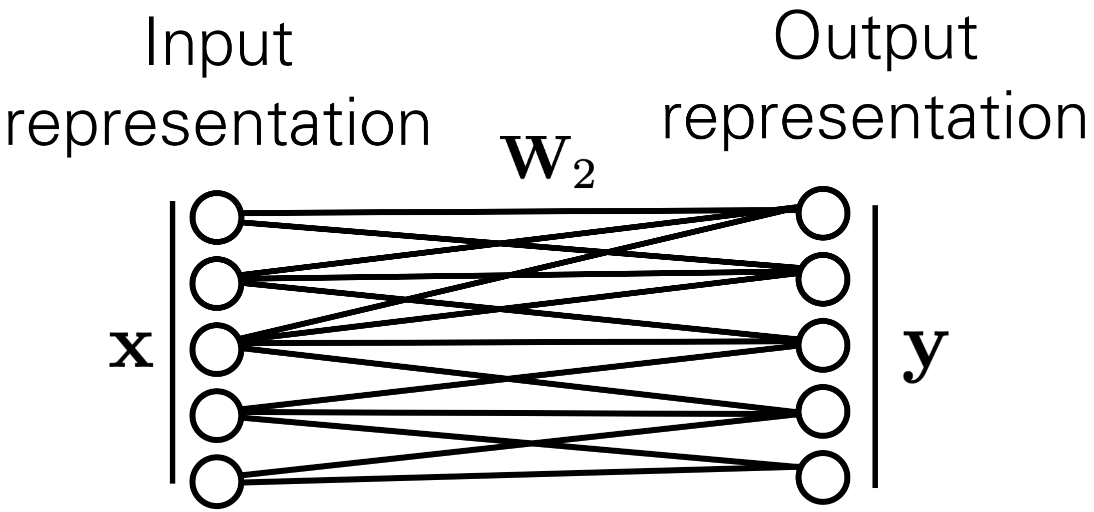
$$\mathbf{h} = g(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \qquad \mathbf{y} = g(\mathbf{W}_2\mathbf{h} + \mathbf{b}_2)$$

$$\theta = \{\mathbf{W}_1, \ldots, \mathbf{W}_L, \mathbf{b}_1, \ldots, \mathbf{b}_L\}$$

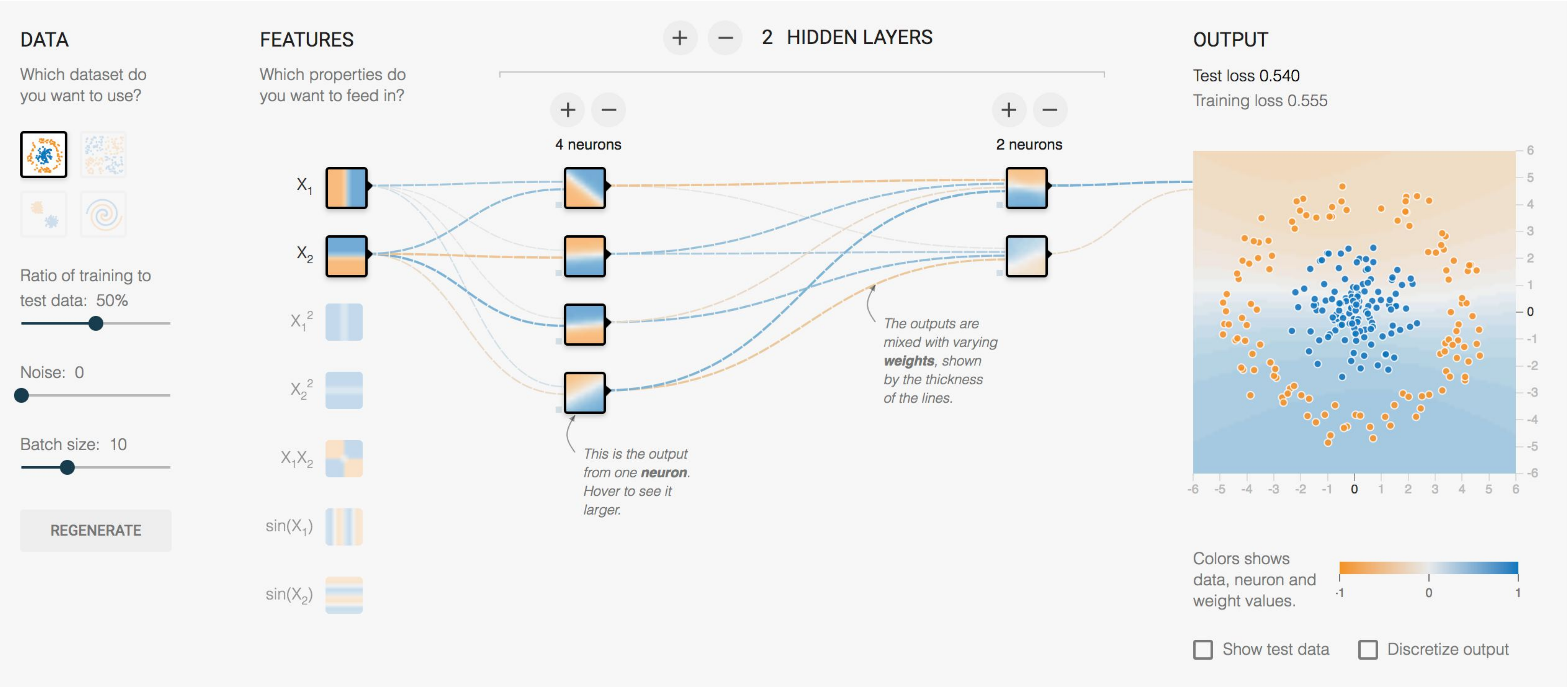
Connectivity Patterns



Fully connected layer

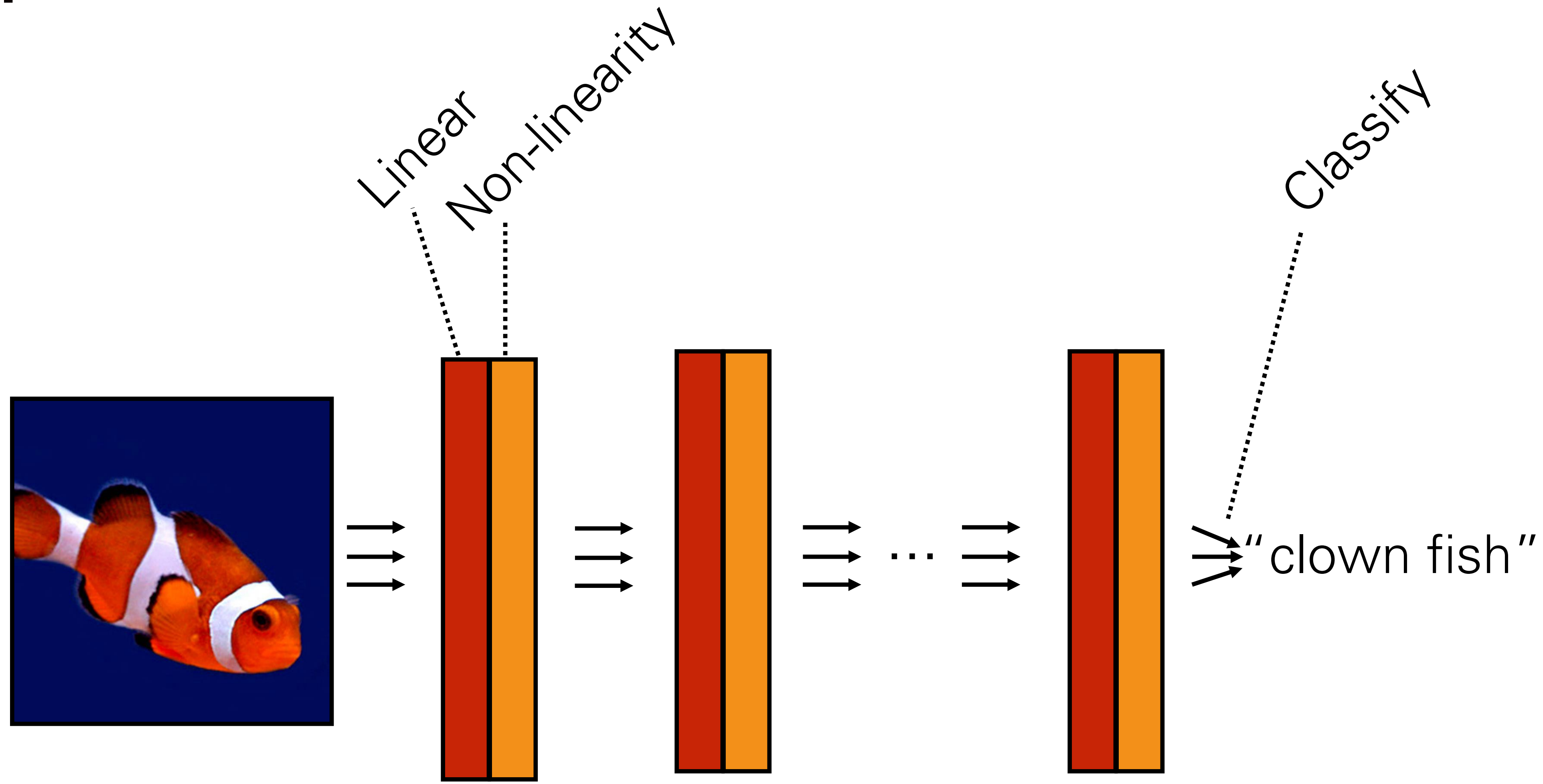


Locally connected layer
(Sparse W)



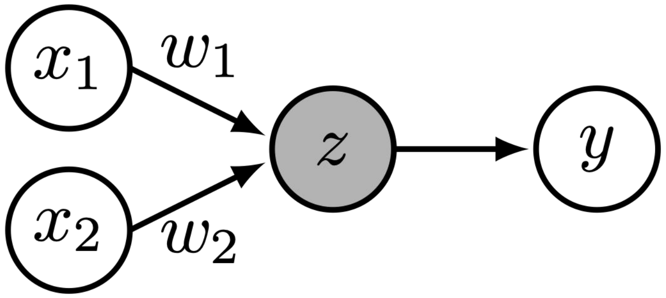
[http://playground.tensorflow.org]

Deep nets



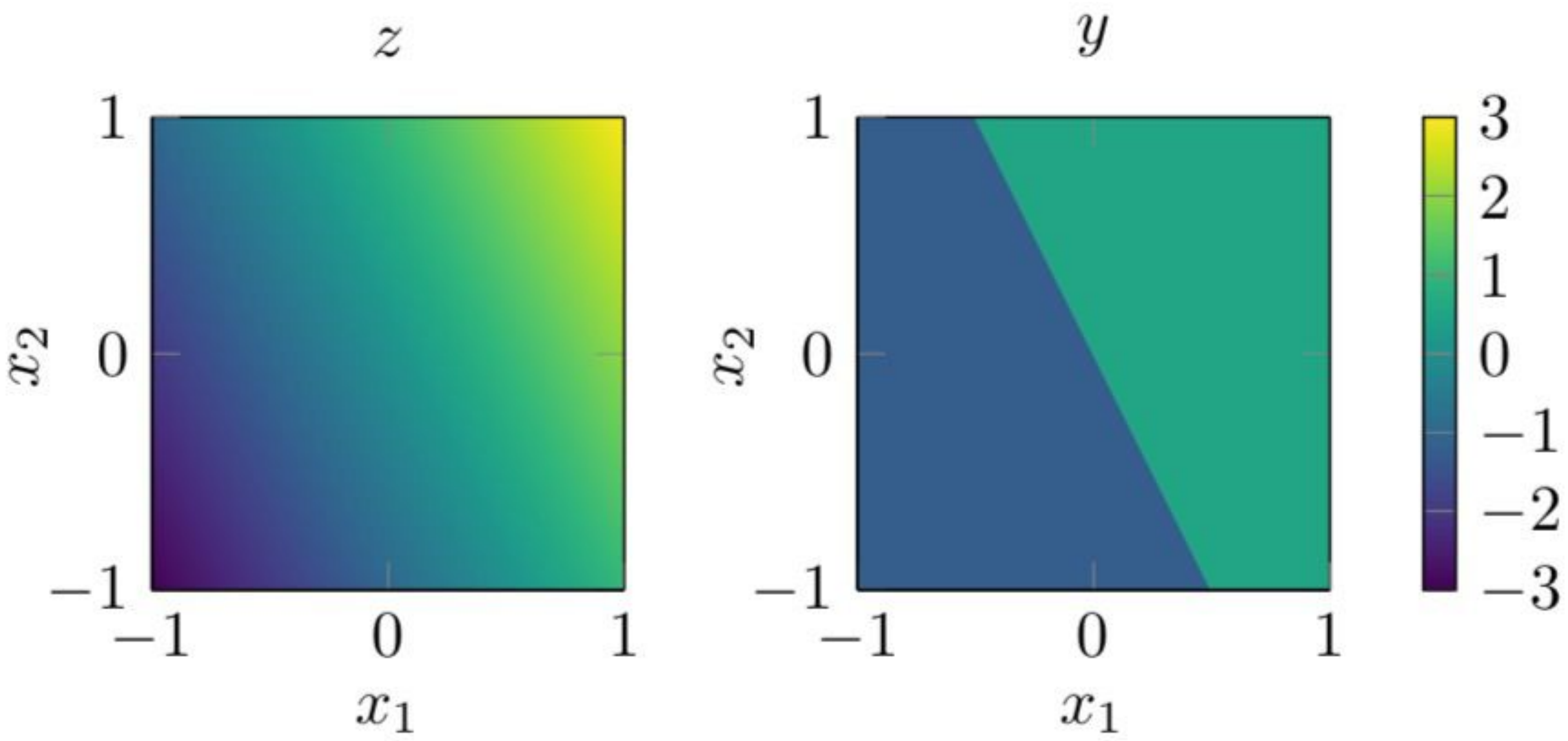
$$f(\mathbf{x}) = f_L(f_{L-1}(\ldots f_2(f_1(\mathbf{x}))))$$

Example: linear classification with a perceptron



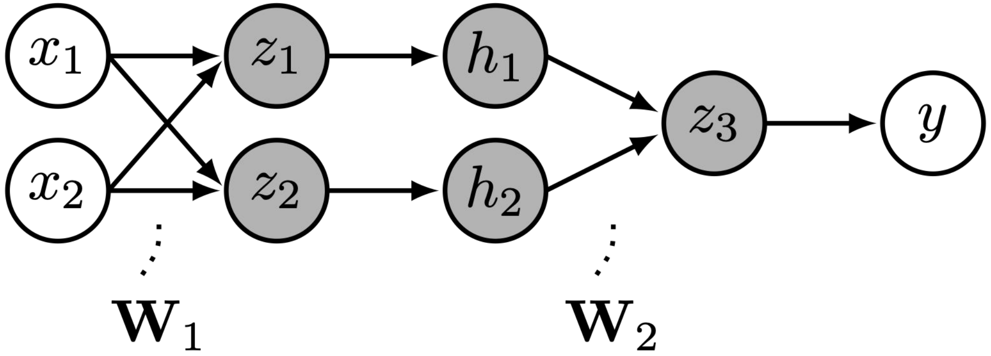
$$z = \mathbf{x}^T \mathbf{w} + b$$

$$y = g(z)$$

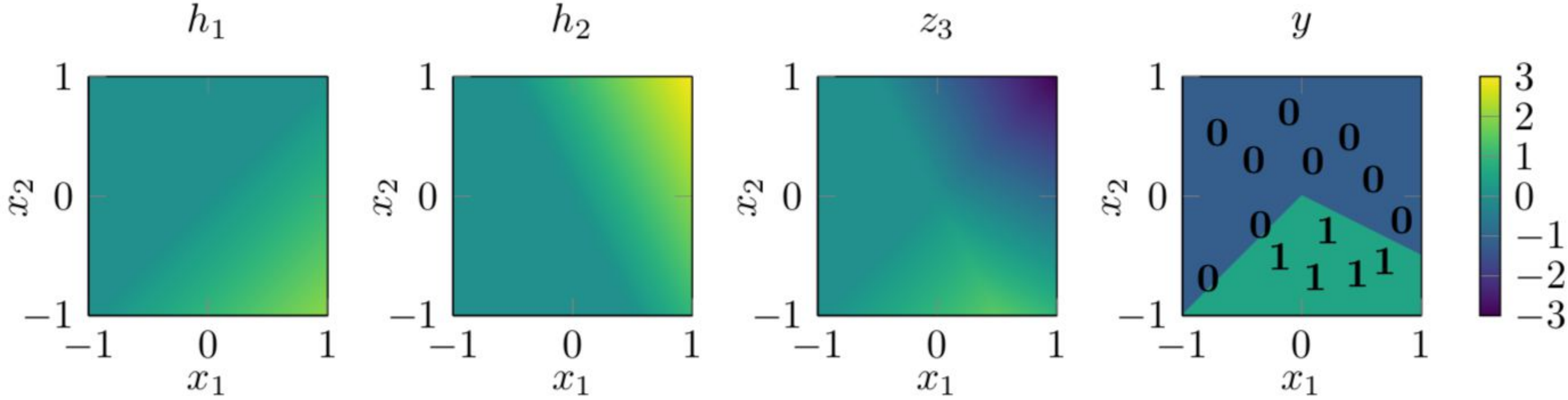


One layer neural net (perceptron) can perform linear classification!

Example: nonlinear classification with a deep net



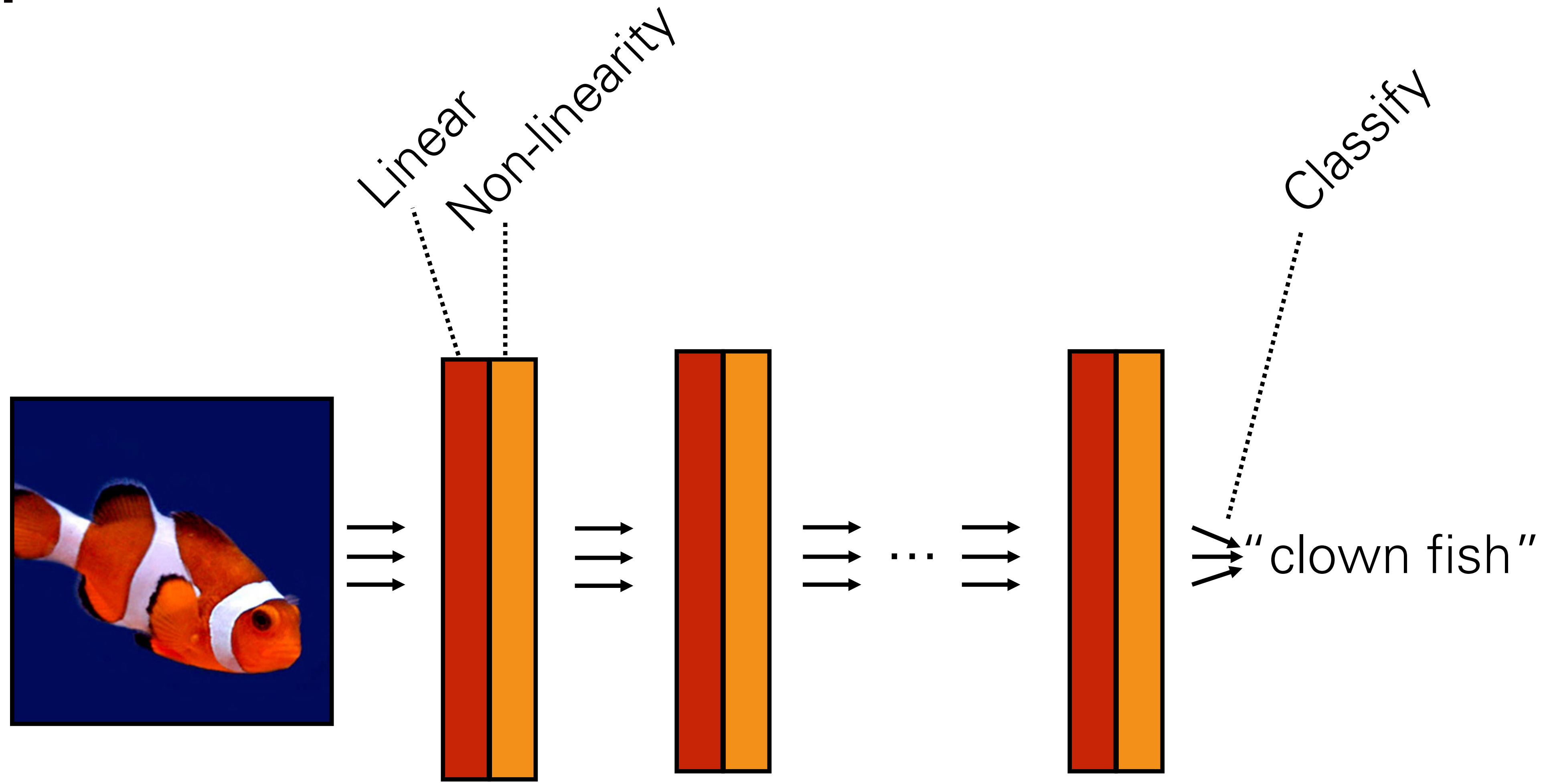
$$\mathbf{z} = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$$
$$\mathbf{h} = g(\mathbf{z})$$
$$z_3 = \mathbf{W}_2\mathbf{h} + b_2$$
$$y = 1(z_3 > 0)$$



Representational power

- 1 layer? Linear decision surface.
- 2+ layers? In theory, can represent any function. Assuming non-trivial non-linearity.
  - Bengio 2009, http://www.iro.umontreal.ca/~bengioy/papers/ftml.pdf
  - Bengio, Courville, Goodfellow book http://www.deeplearningbook.org/contents/mlp.html
  - Simple proof by M. Neilsen http://neuralnetworksanddeeplearning.com/chap4.html
  - D. Mackay book http://www.inference.phy.cam.ac.uk/mackay/itprnn/ps/482.491.pdf
- But issue is efficiency: very wide two layers vs narrow deep model? In practice, more layers helps.

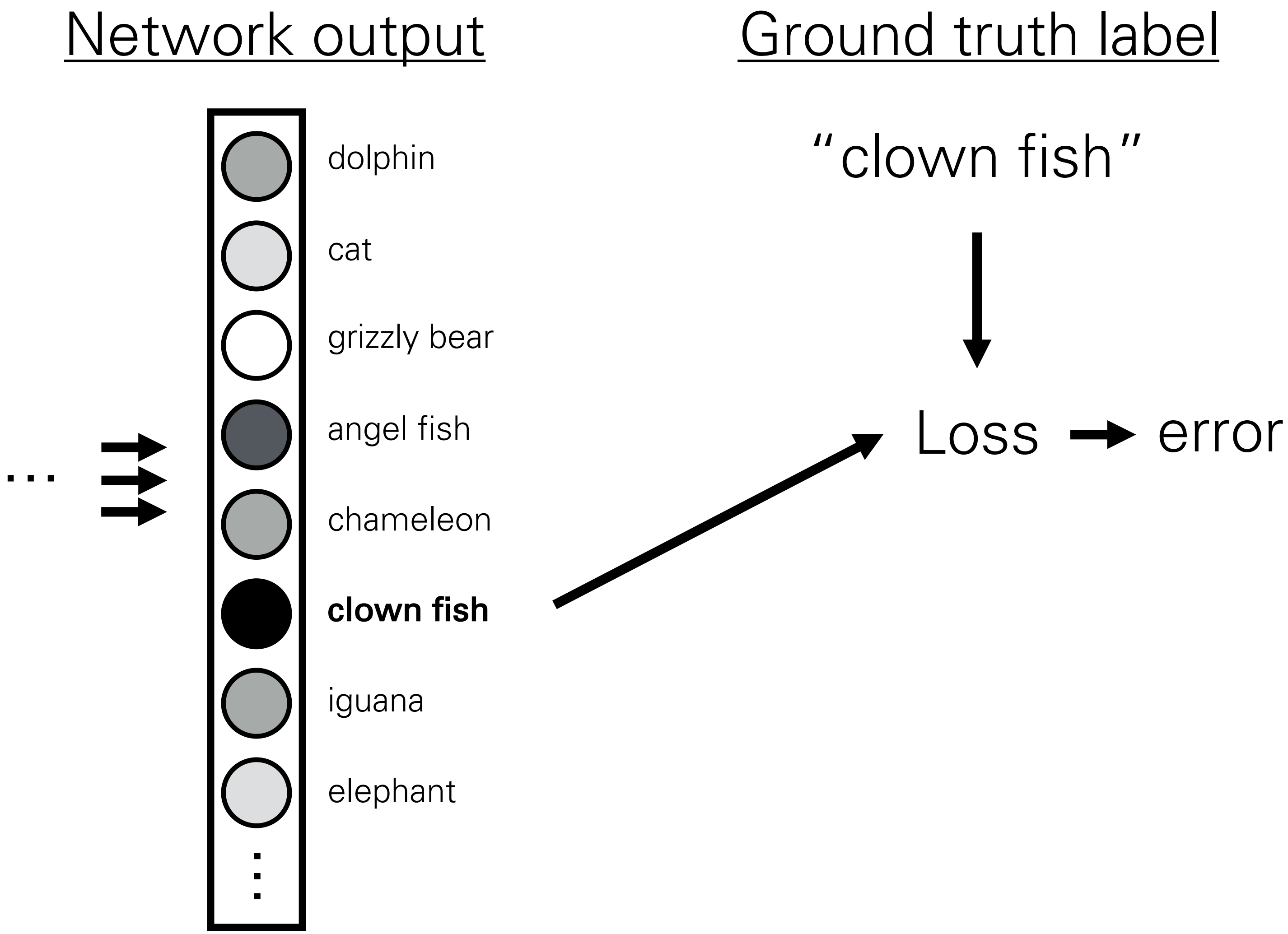
Deep nets



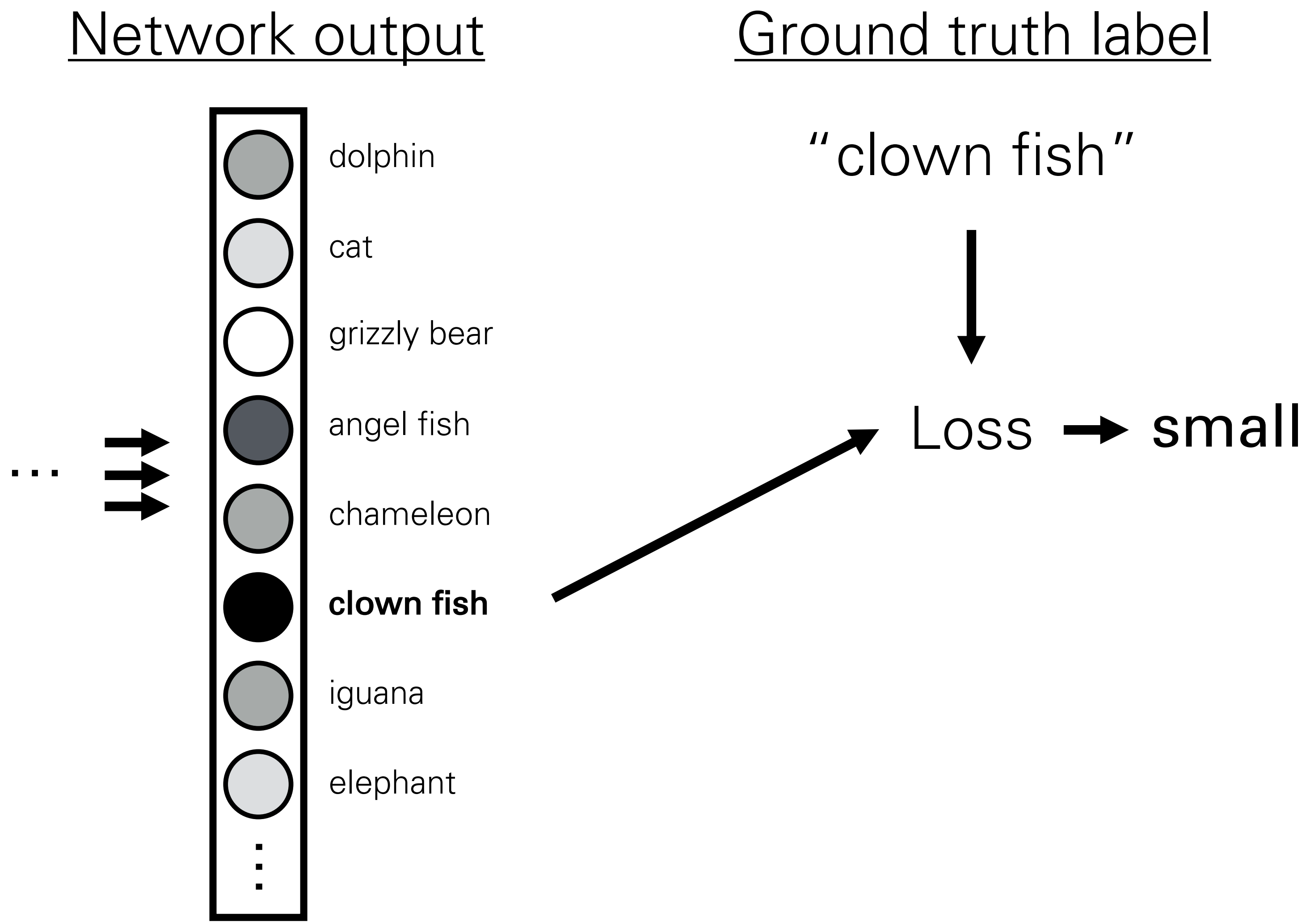
$$f(\mathbf{x}) = f_L(f_{L-1}(\dots f_2(f_1(\mathbf{x}))))$$

Classifier layer

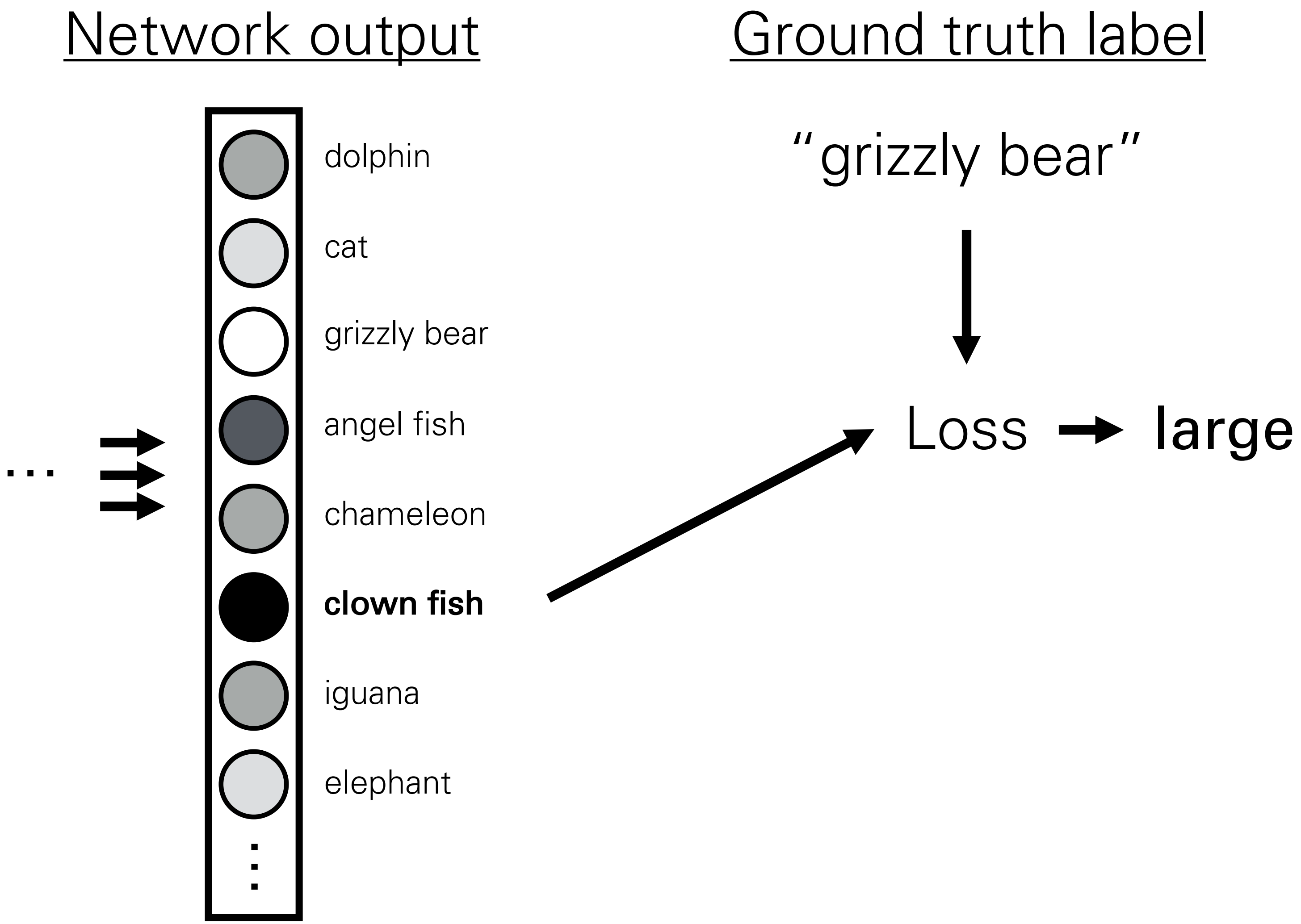
Loss function



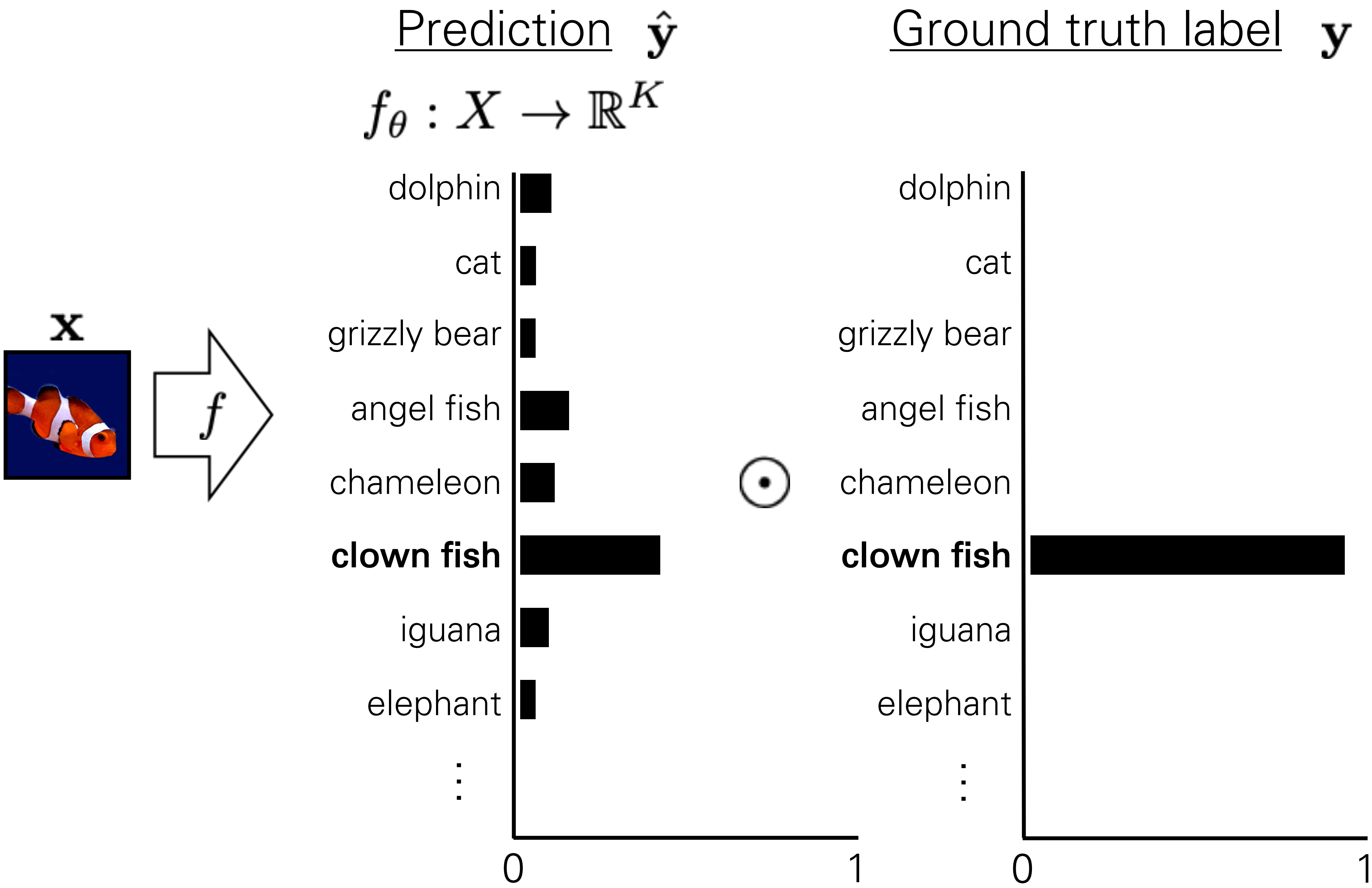
Loss function



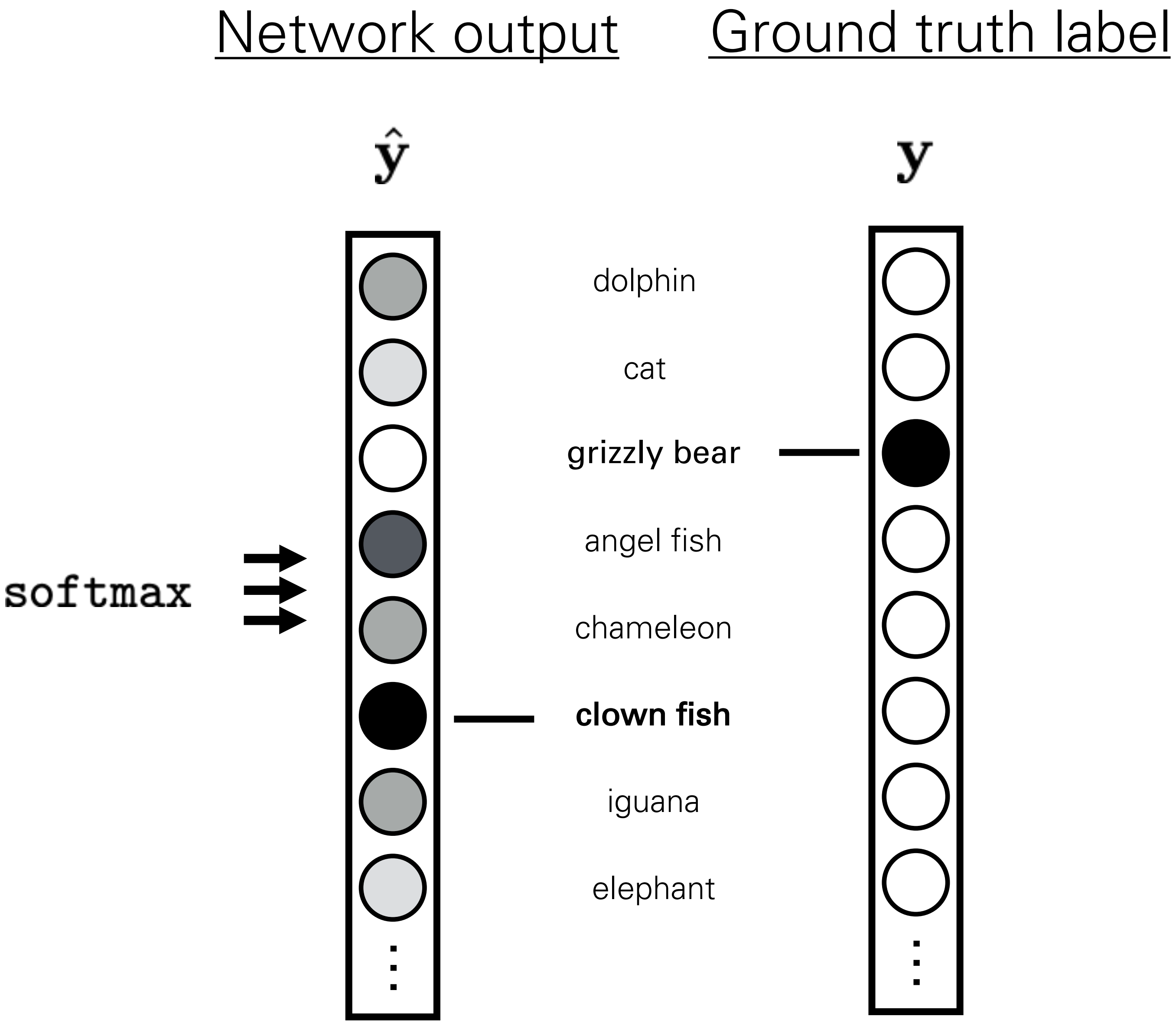
Loss function



Loss function



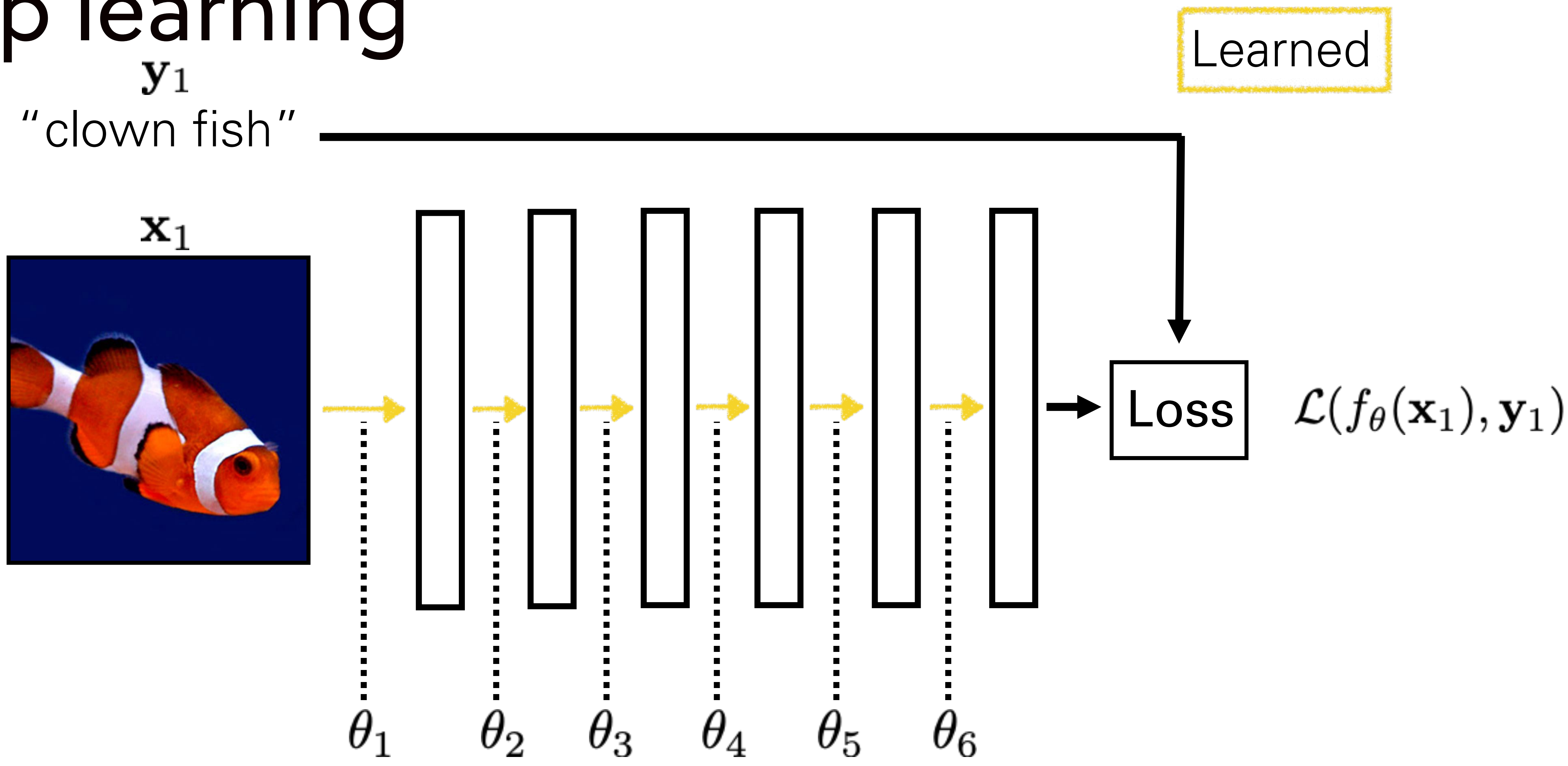
Loss function



Probability of the observed data under the model

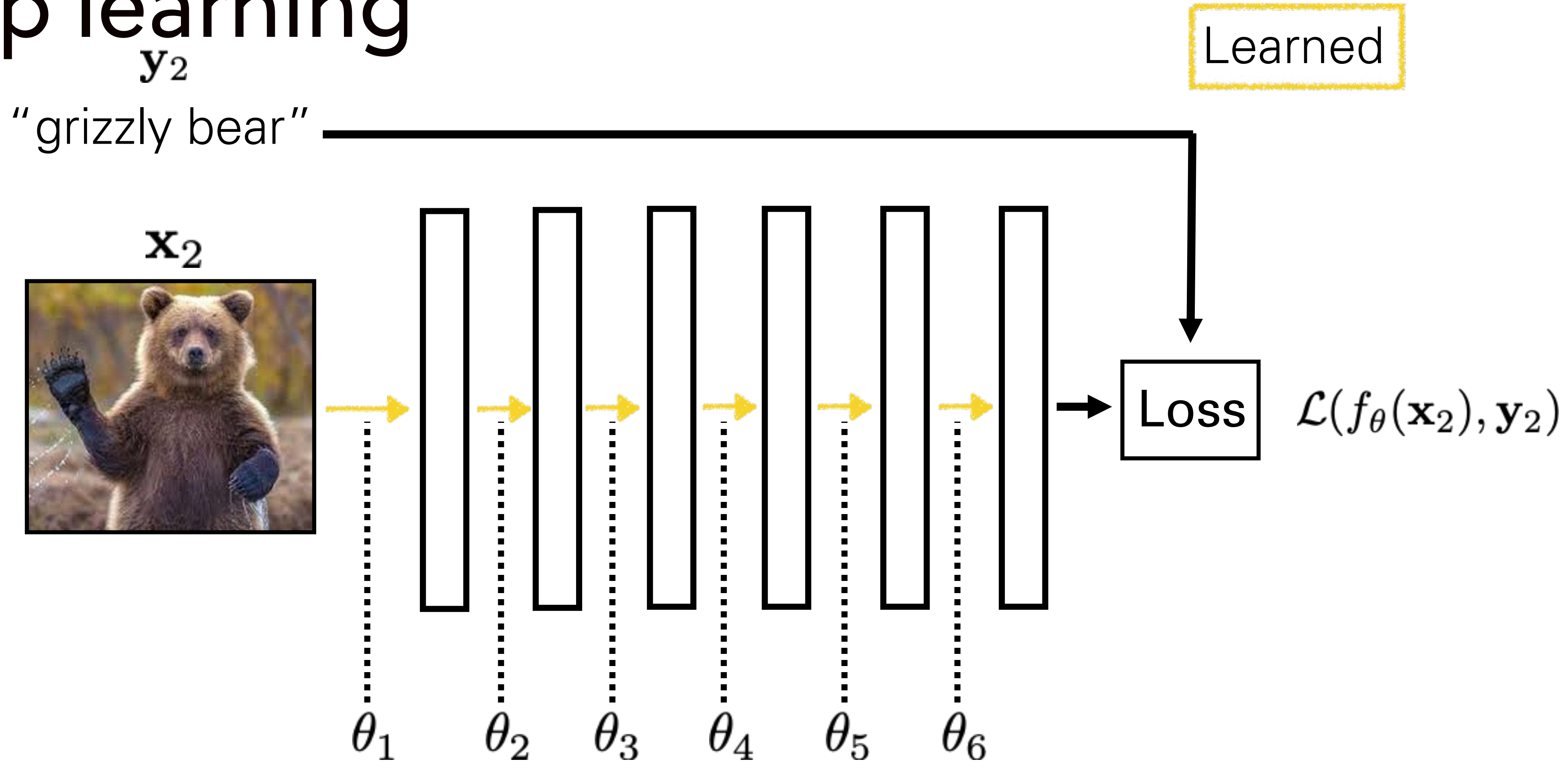
$$H(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

Deep learning



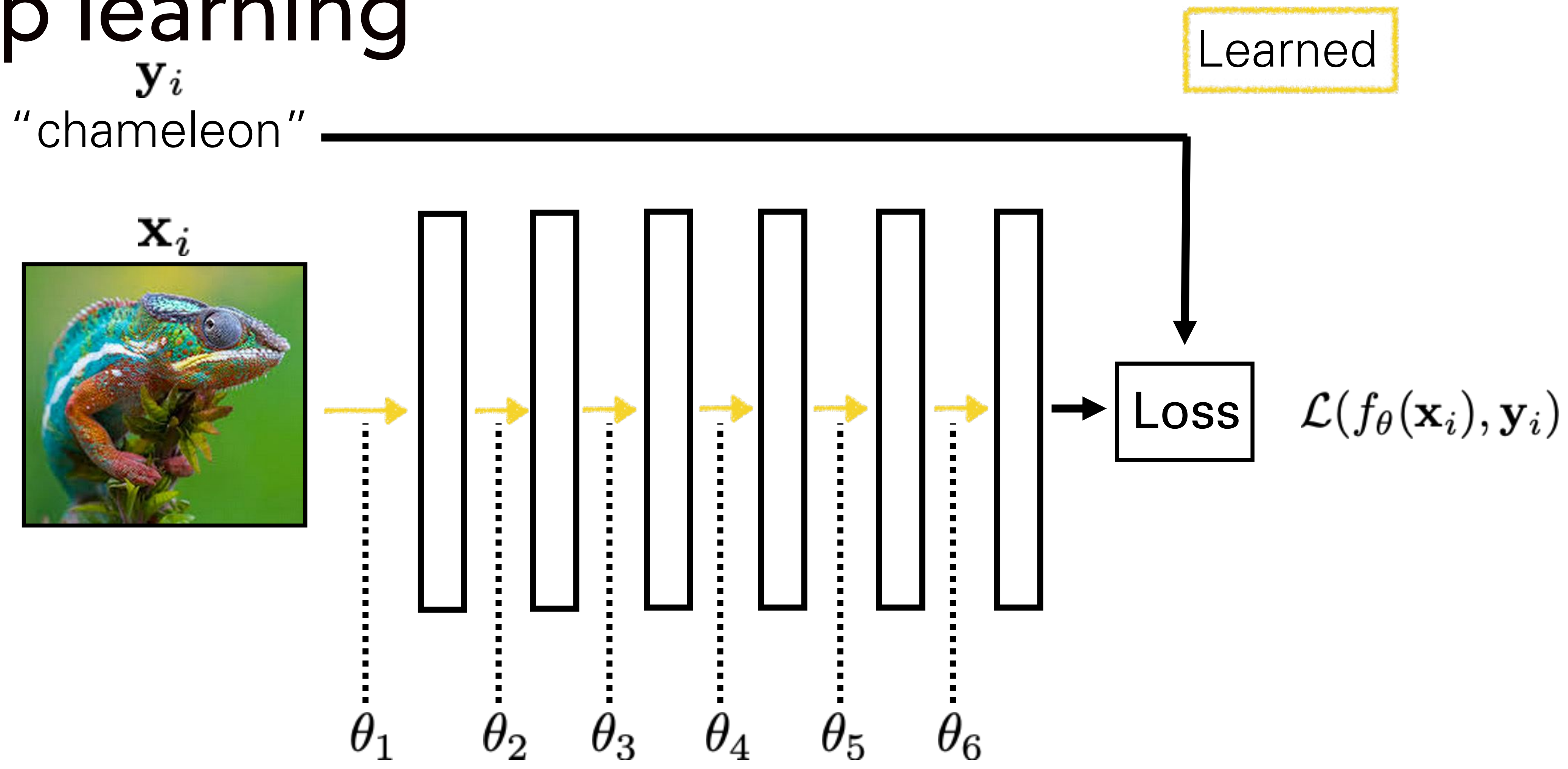
$$\theta^* = \arg\min_\theta \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)$$

Deep learning



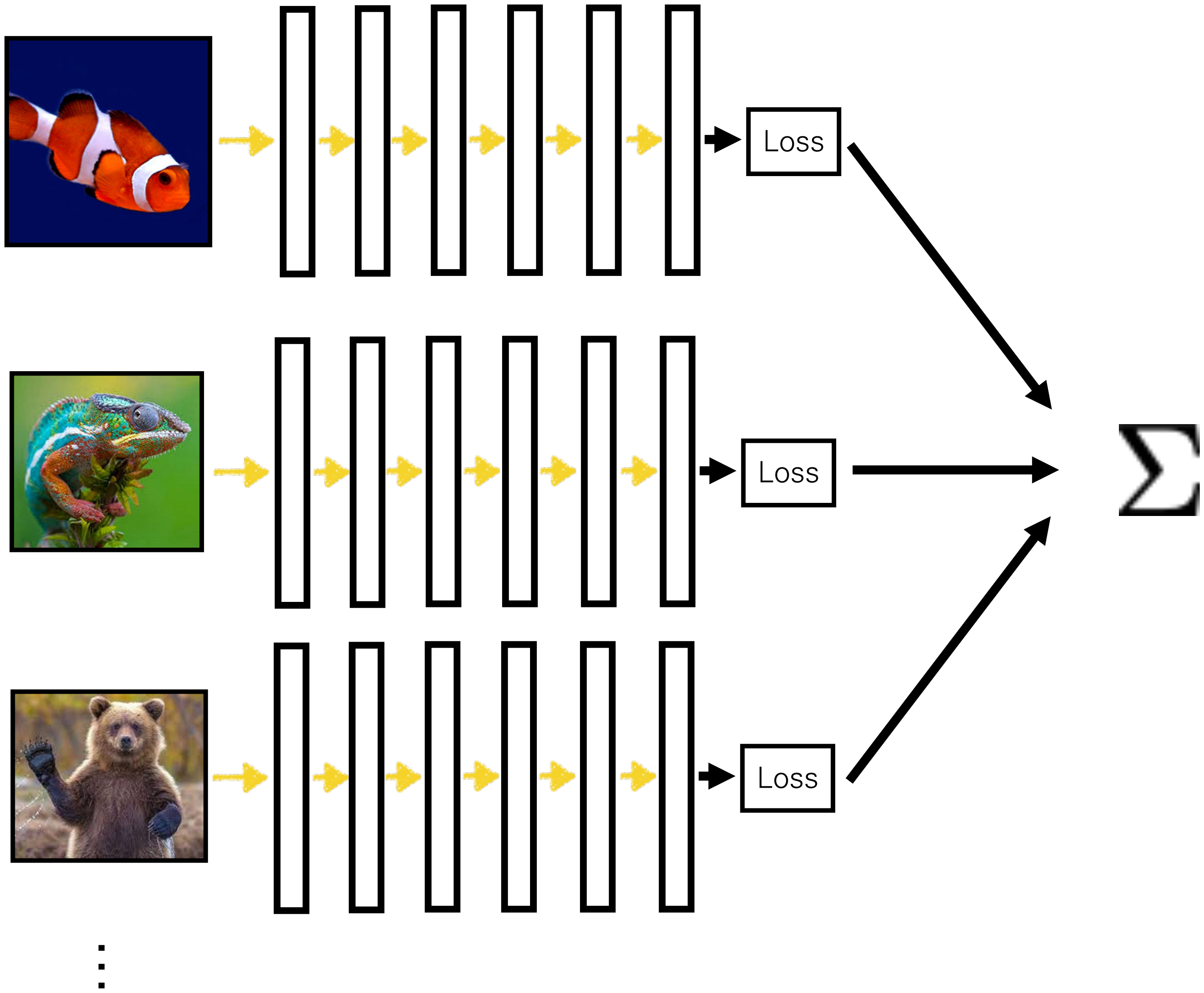
$$\theta^* = \arg\min_\theta \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)$$

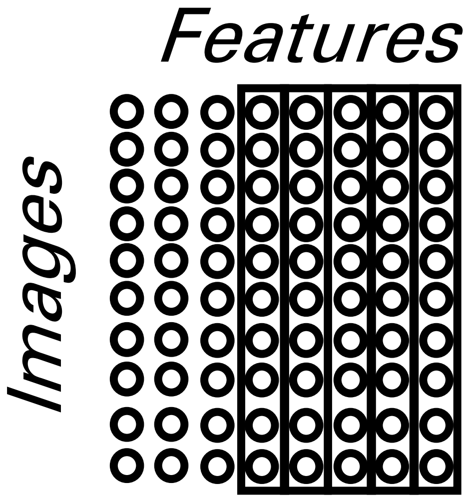
Deep learning



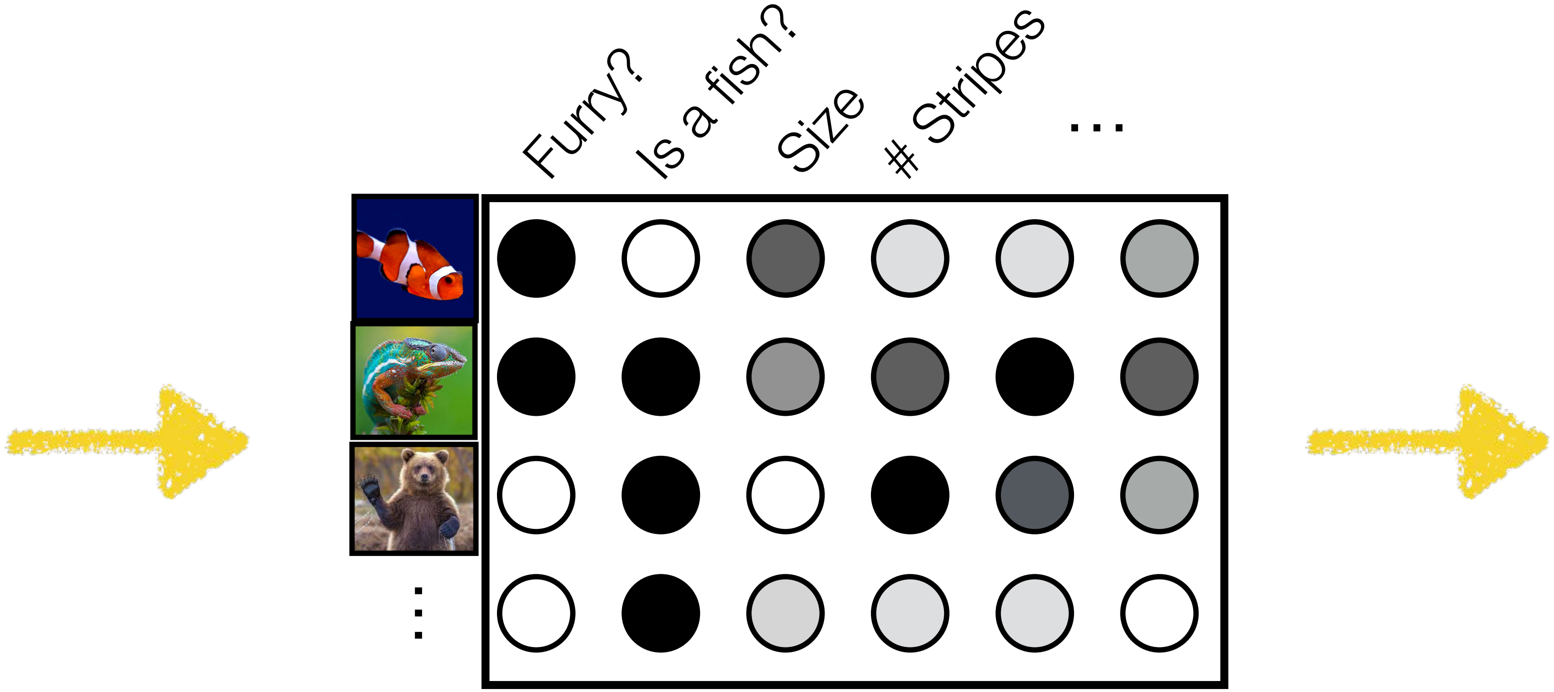
$$\theta^* = \arg\min_\theta \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)$$

Batch (parallel) processing



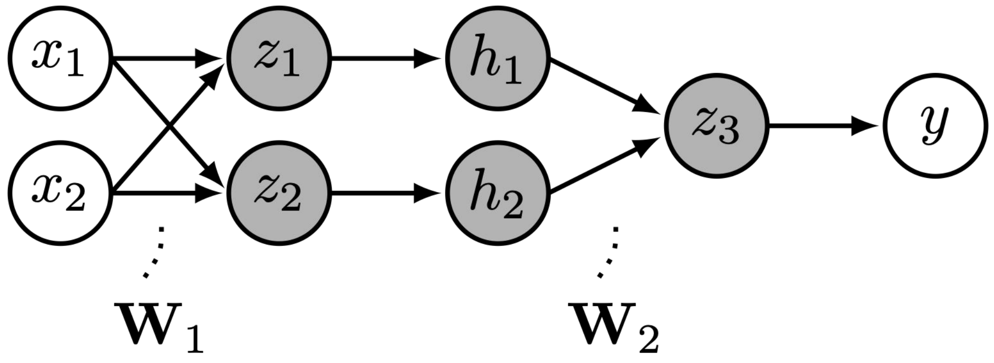


Tensors (multi-dimensional arrays)



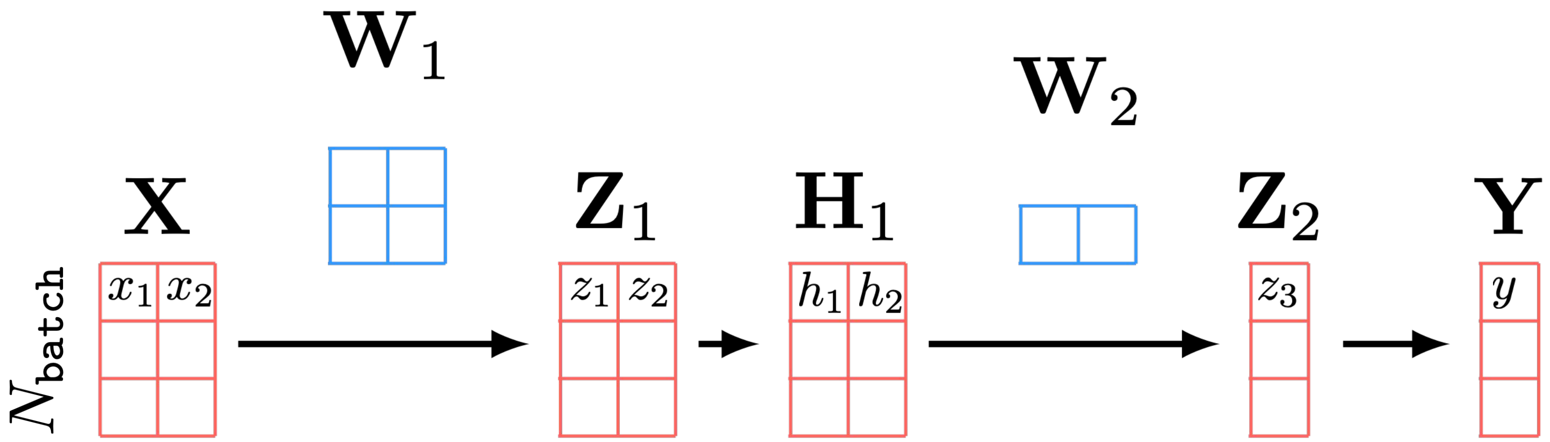
Each layer is a representation of the data

Everything is a tensor



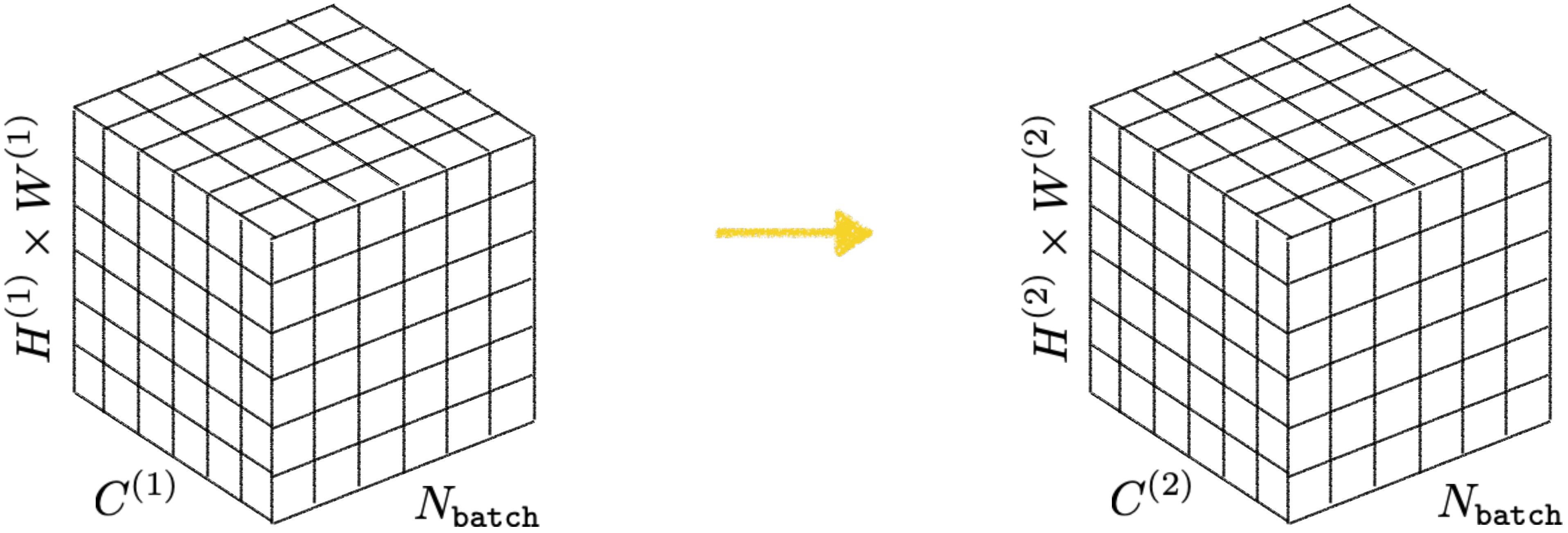
$$\mathbf{z} = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$$
$$\mathbf{h} = g(\mathbf{z})$$
$$z_3 = \mathbf{W}_2\mathbf{h} + b_2$$
$$y = 1(z_3 > 0)$$

Tensor processing with batch size = 3:



"Tensor flow"

$\mathbf{h}^{(1)} \in \mathbb{R}^{N_{\texttt{batch}} \times H^{(1)} \times W^{(1)} \times C^{(1)}}$ $\mathbf{h}^{(2)} \in \mathbb{R}^{N_{\texttt{batch}} \times H^{(2)} \times W^{(2)} \times C^{(2)}}$



Regularizing deep nets

Deep nets have millions of parameters!

On many datasets, it is easy to overfit — we may have more free parameters than data points to constrain them.

How can we regularize to prevent the network from overfitting?
1. Fewer neurons, fewer layers
2. Weight decay
3. Dropout
4. Normalization layers
5. …

Recall: regularized least squares

$$f_\theta(x) = \sum_{k=0}^{K} \theta_k x^k$$

$$R(\theta) = \lambda \|\theta\|_2^2$$

← Only use polynomial terms if you really need them! Most terms should be zero

ridge regression, a.k.a., **Tikhonov regularization**

Probabilistic interpretation: R is a Gaussian **prior** over values of the parameters.

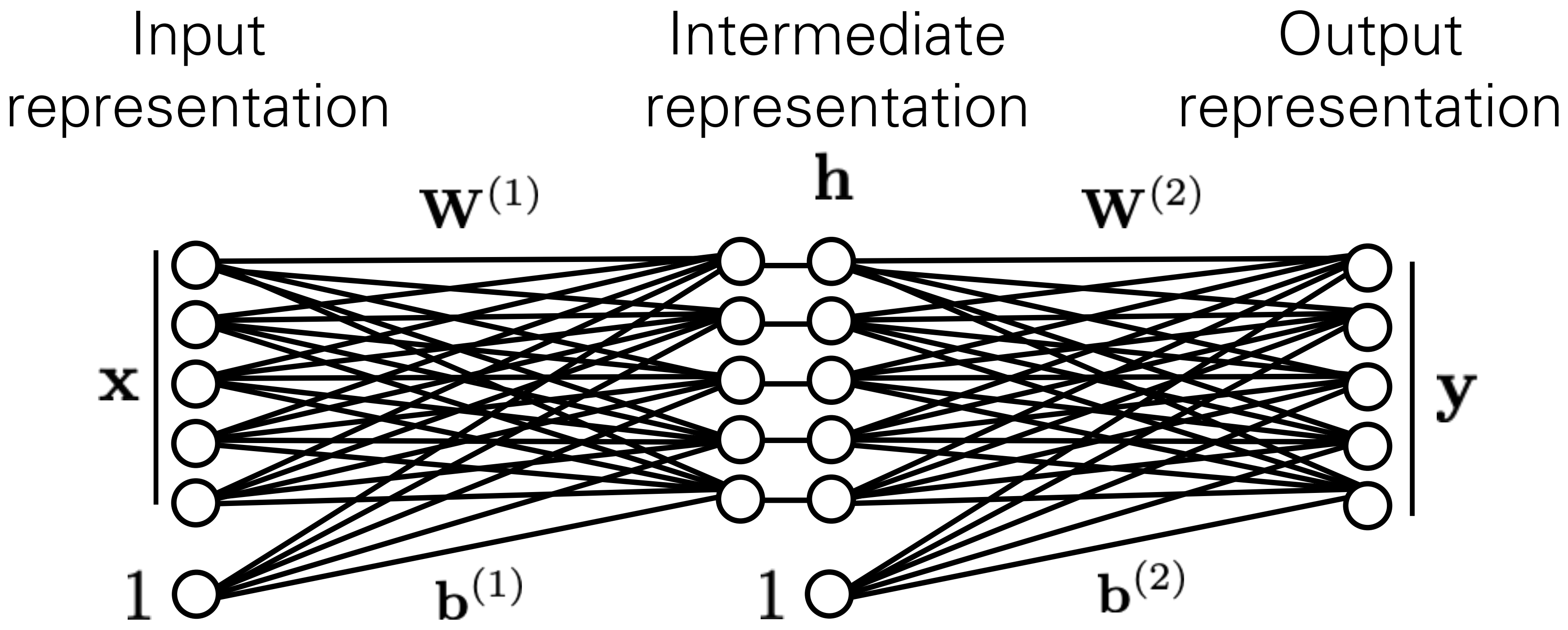
Recall: regularized least squares

$$\theta^* = \arg\min_{\theta} \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i) + R(\theta)$$

$$R(\mathbf{W}) = \lambda \|\mathbf{W}\|_2^2$$ ← weight decay

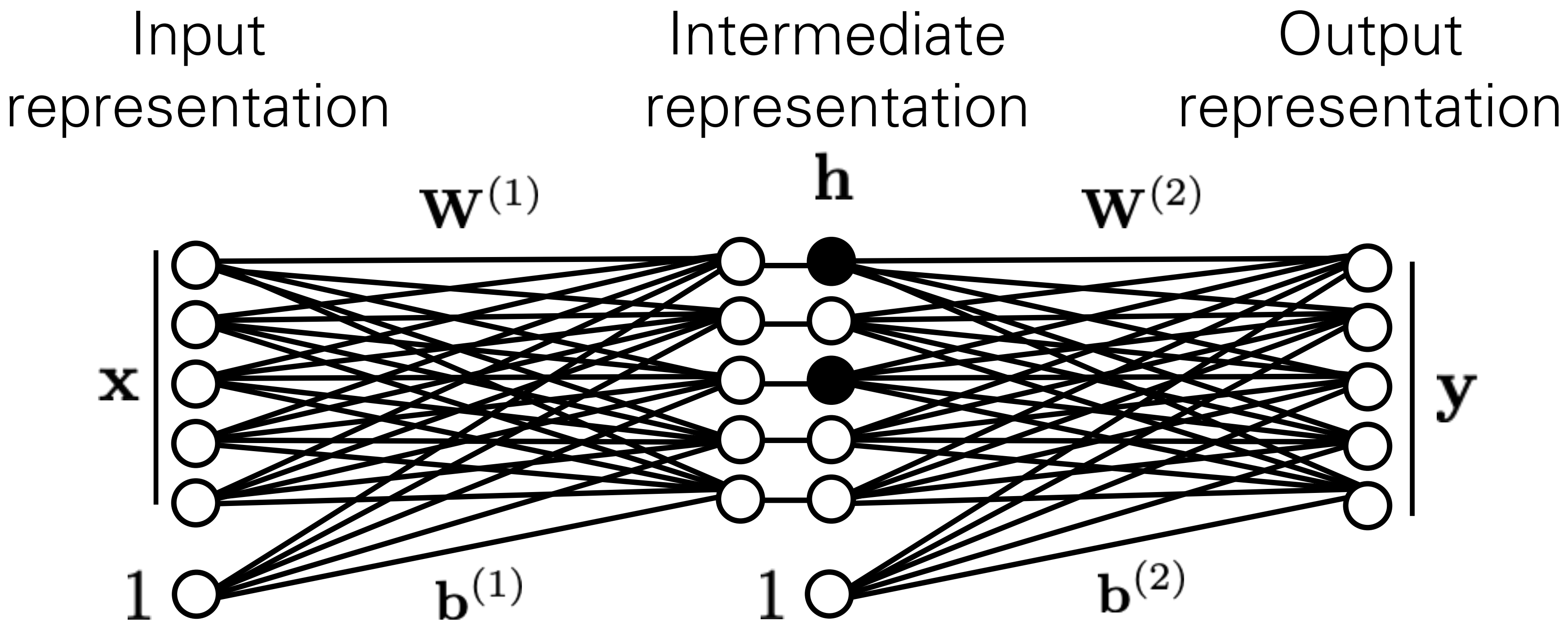
"We prefer to keep weights small."

Dropout



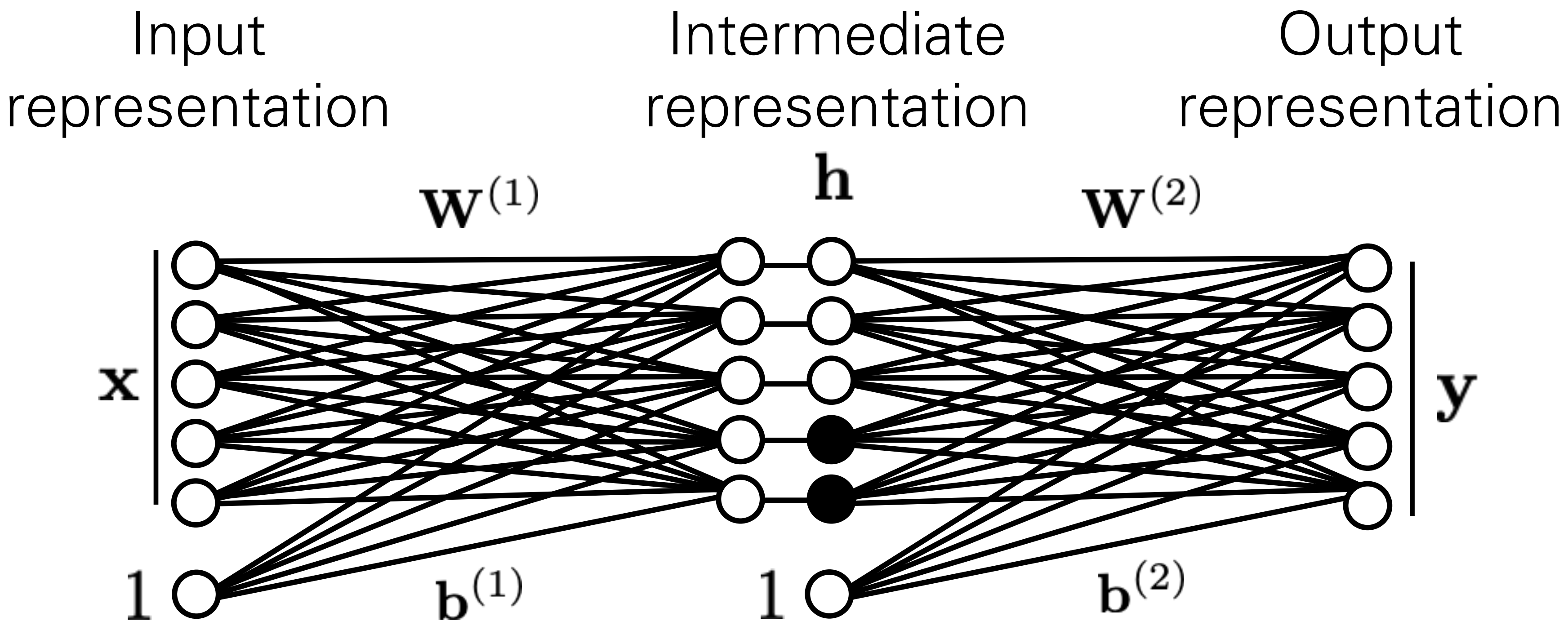
$$\theta = \{\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(L)}\}$$

Dropout



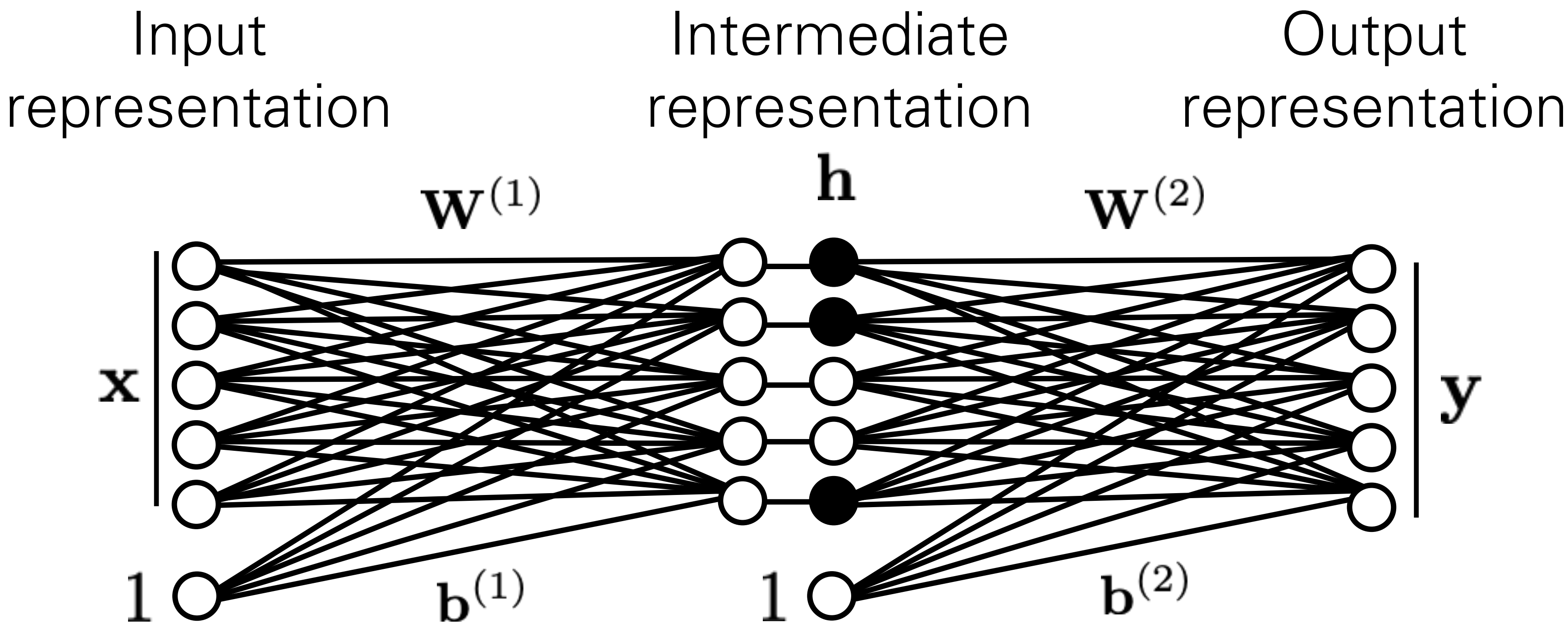
$$\theta = \{\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(L)}\}$$

Dropout



$$\theta = \{\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(L)}\}$$

Dropout



$$\theta = \{\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(L)}\}$$

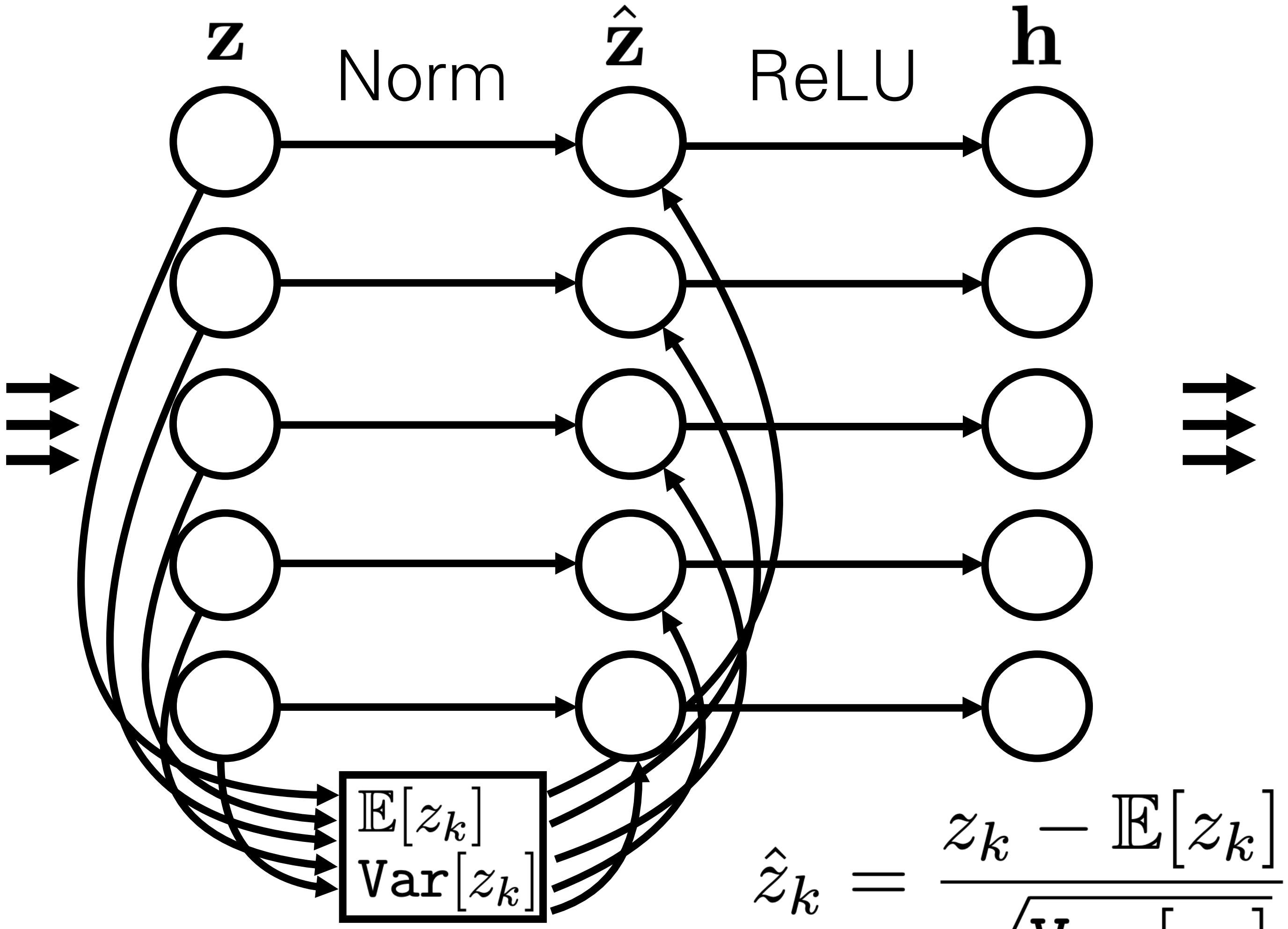
Dropout

Randomly zero out hidden units.

Prevents network from relying too much on spurious correlations between different hidden units.

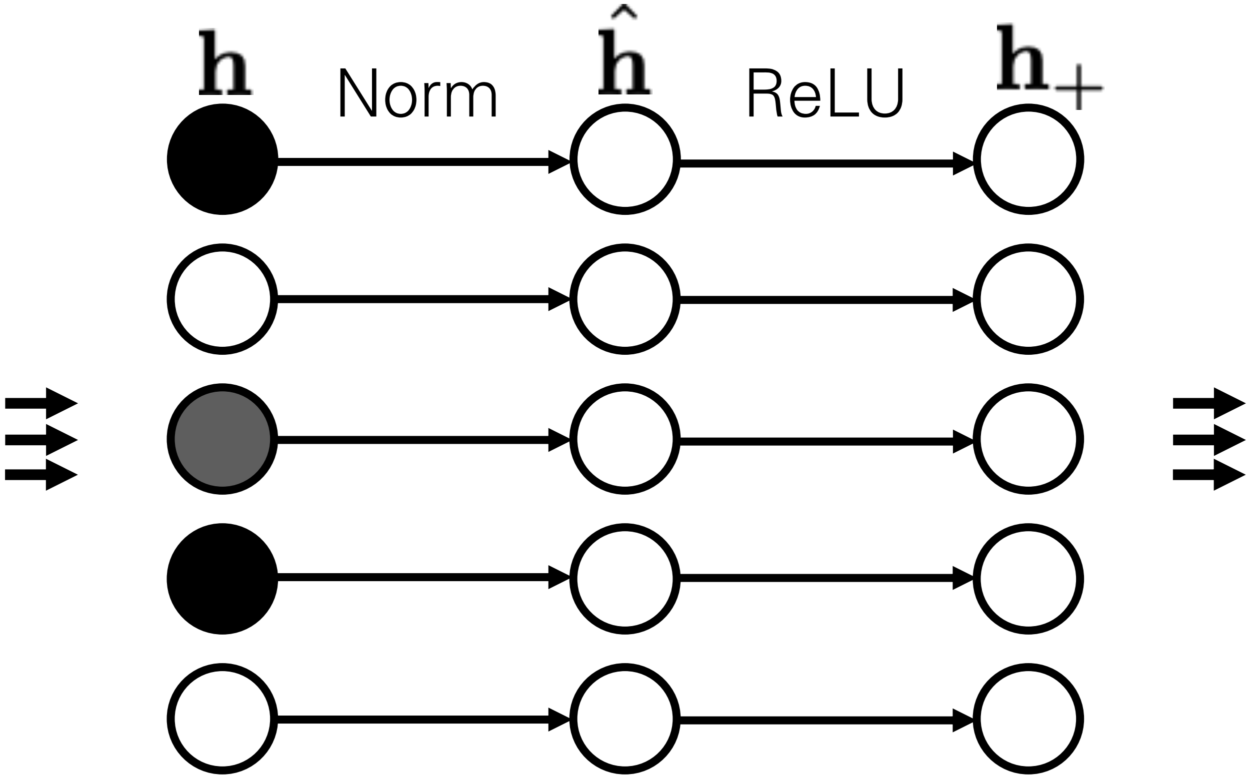
Can be understood as averaging over an exponential **ensemble** of subnetworks. This averaging smooths the function, thereby reducing the effective capacity of the network.

Normalization layers



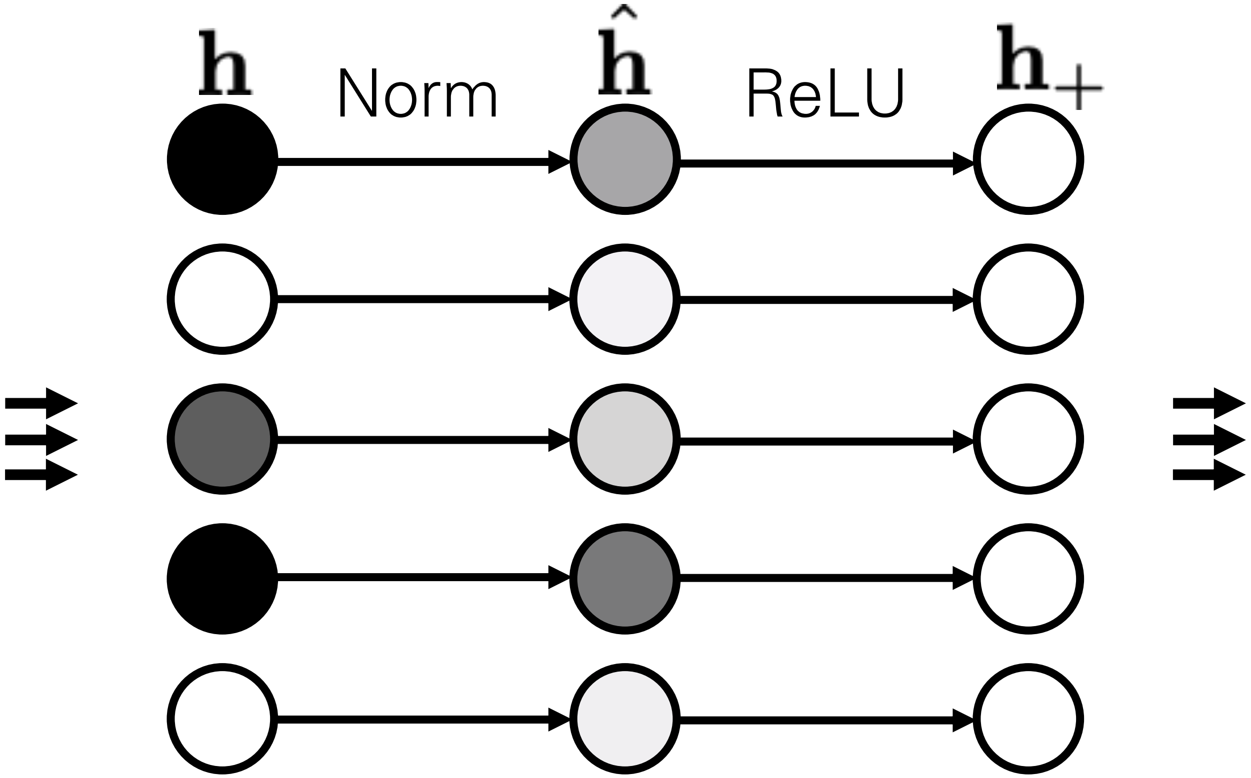
$$\hat{z}_k = \frac{z_k - \mathbb{E}[z_k]}{\sqrt{\texttt{Var}[z_k]}}$$

Normalization layers



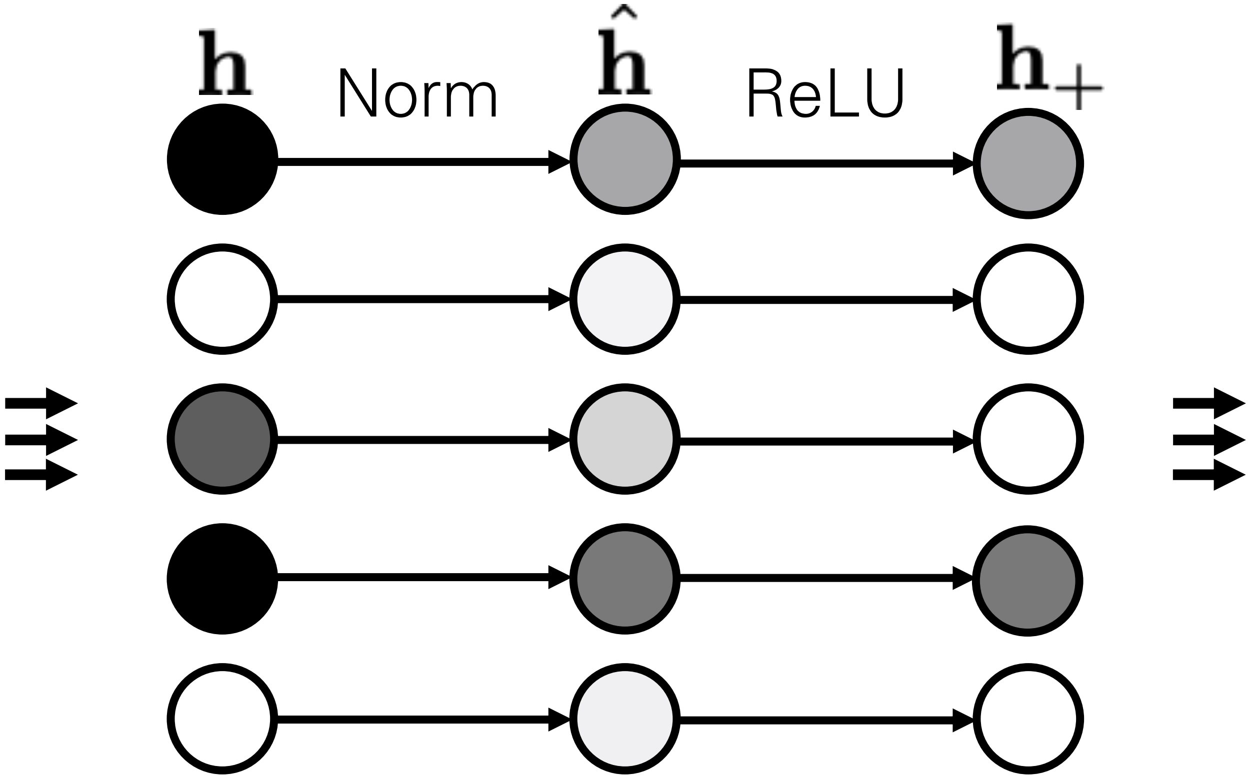
$$\hat{h}_k = \frac{h_k - \mathbb{E}[h_k]}{\sqrt{\mathtt{Var}[h_k]}}$$

Normalization layers



$$\hat{h}_k = \frac{h_k - \mathbb{E}[h_k]}{\sqrt{\mathtt{Var}[h_k]}}$$

Normalization layers



$$\hat{h}_k = \frac{h_k - \mathbb{E}[h_k]}{\sqrt{\texttt{Var}[h_k]}}$$

Normalization layers

Keep track of mean and variance of a unit (or a population of units) over time.

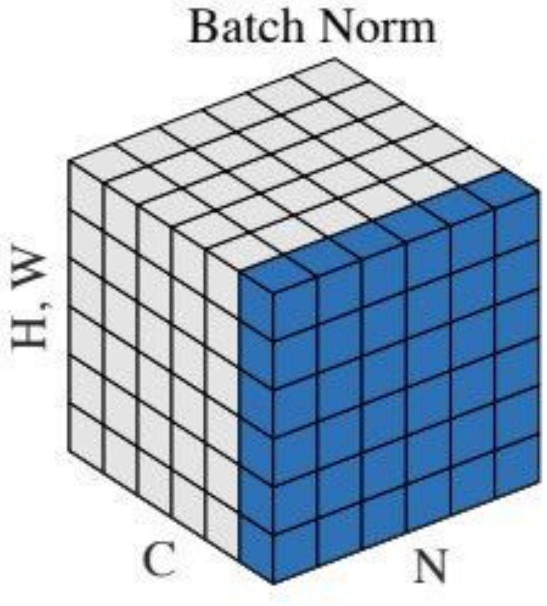
Standardize unit activations by subtracting mean and dividing by variance.

Squashes units into a **standard range**, avoiding overflow.

Also achieves **invariance** to mean and variance of the training signal.

Both these properties reduce the effective capacity of the model, i.e. regularize the model.

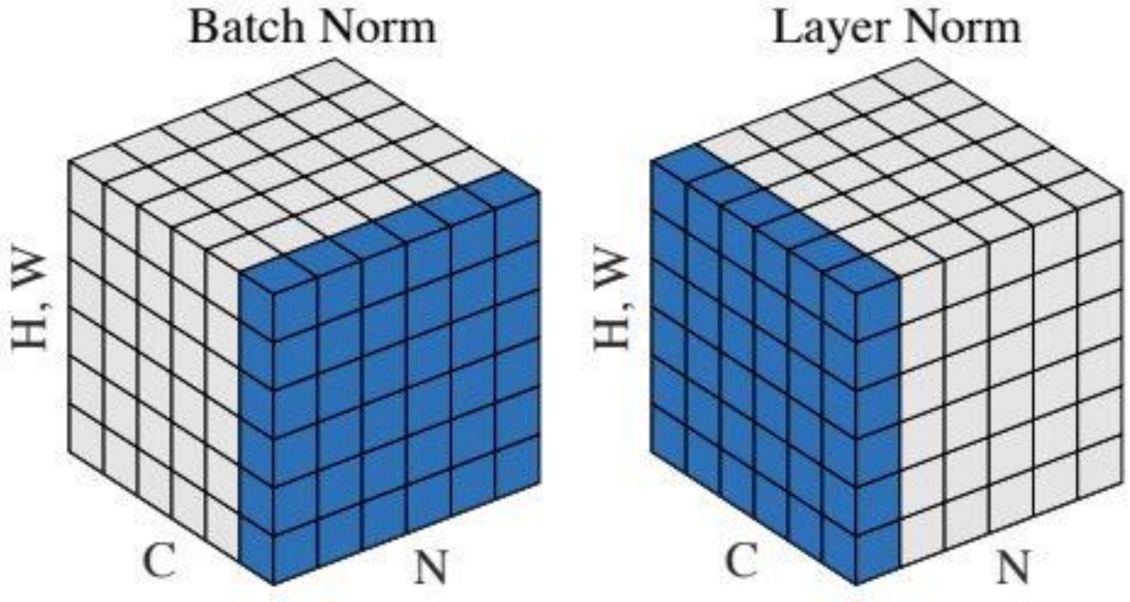
Normalization layers

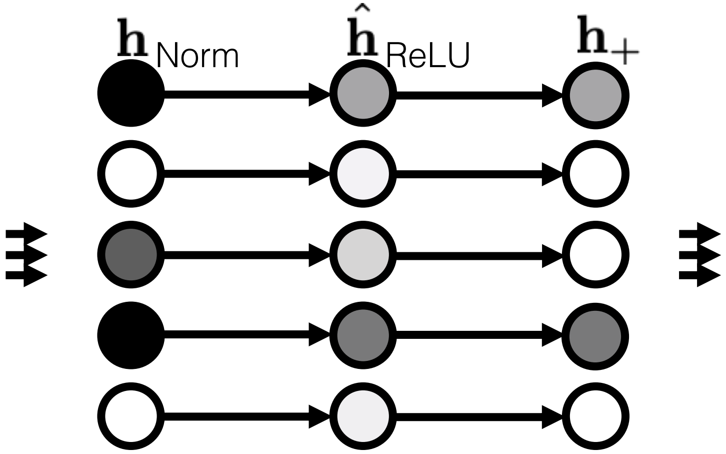


Normalize w.r.t. a single hidden unit's pattern of activation over training examples (a batch of examples).

[Figure from Wu & He, arXiv 2018]

Normalization layers

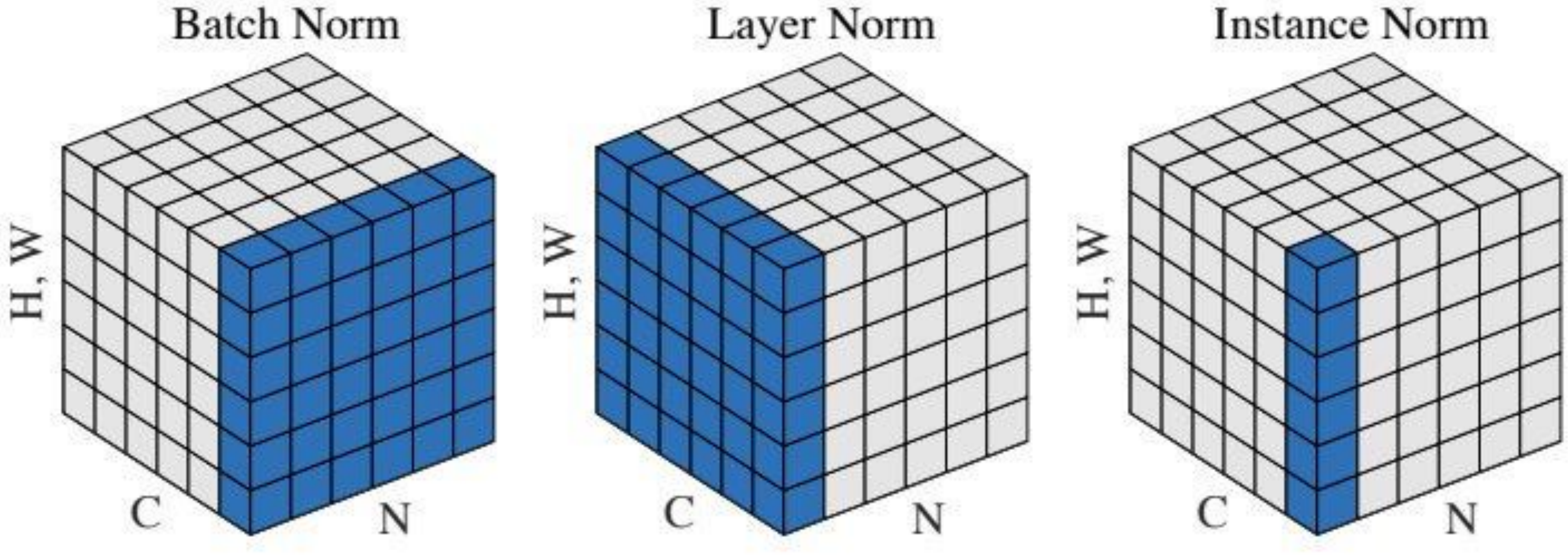




Normalize w.r.t. the mean and variance of the activations of all the hidden units (neurons) on this layer (c).

[Figure from Wu & He, arXiv 2018]

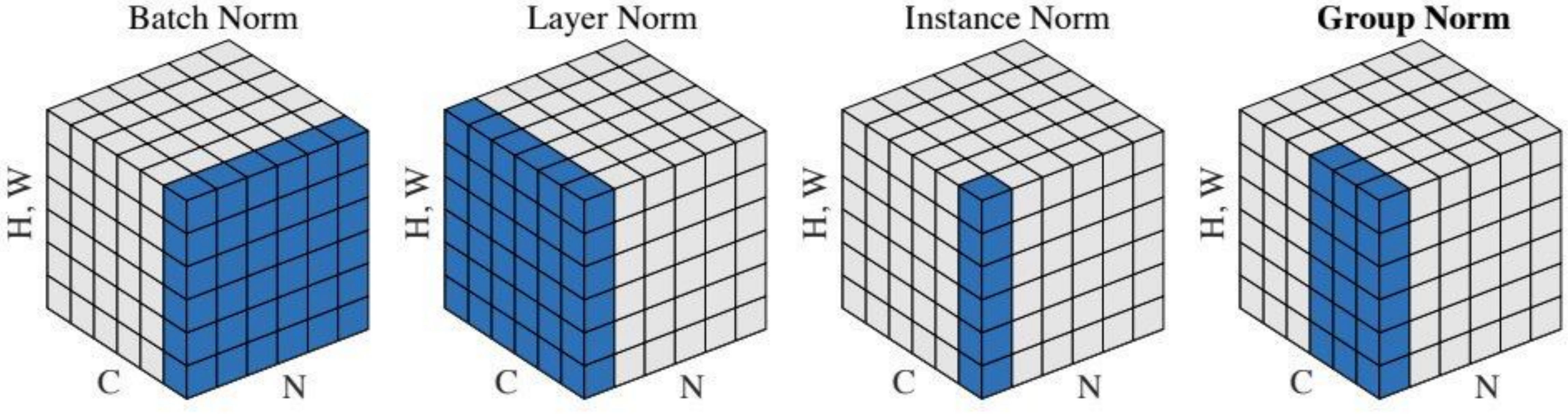
Normalization layers



Normalize w.r.t. the mean and variance of the activations of all the hidden units (neurons) on this layer (c) that process this particular location (h,w) in the image.

[Figure from Wu & He, arXiv 2018]

Normalization layers

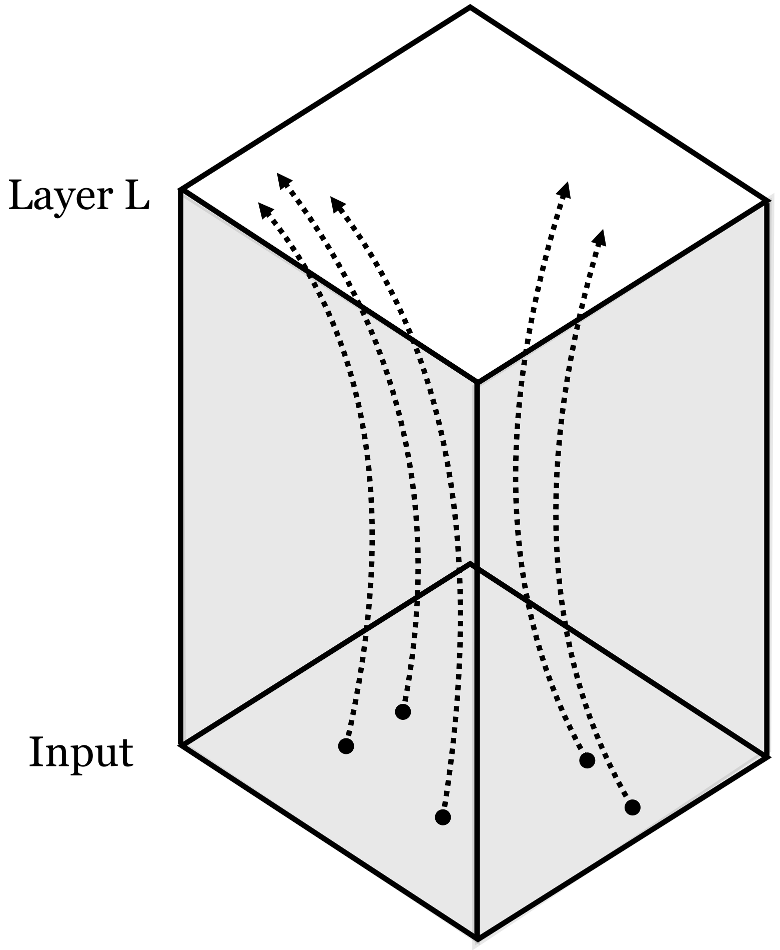


Might as well…

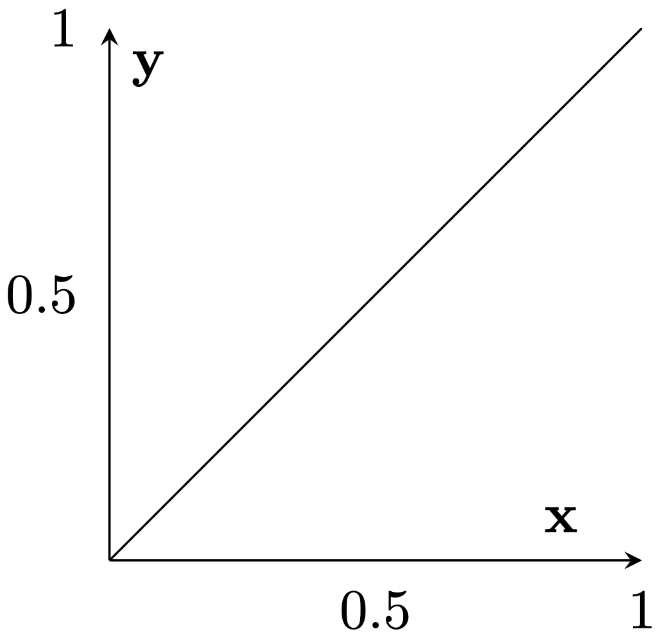
[Figure from Wu & He, arXiv 2018]

Deep nets are data transformers

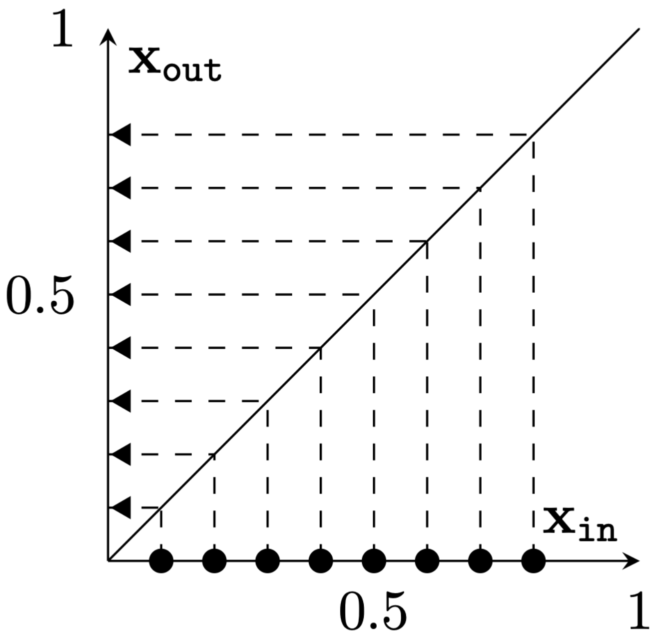
- Deep nets transform datapoints, layer by layer
- Each layer is a different representation of the data
- We call these representations **embeddings**



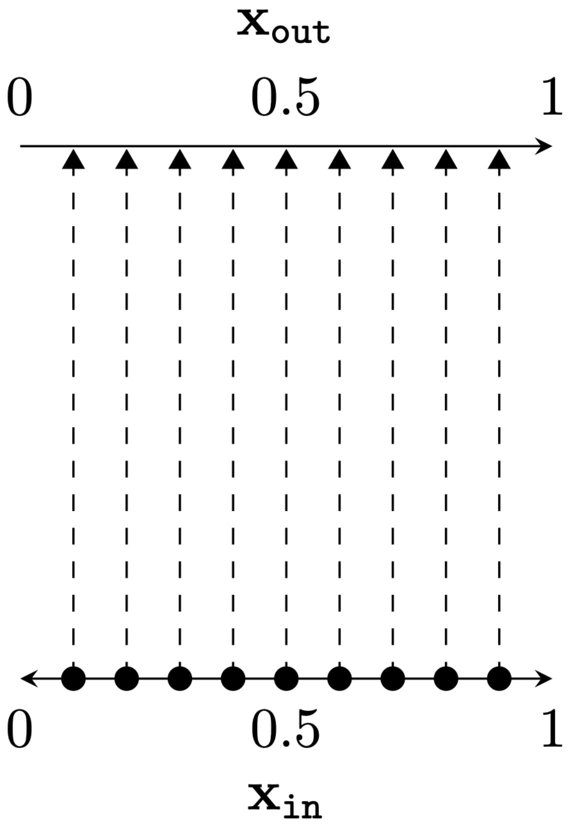
Two different ways to represent a function



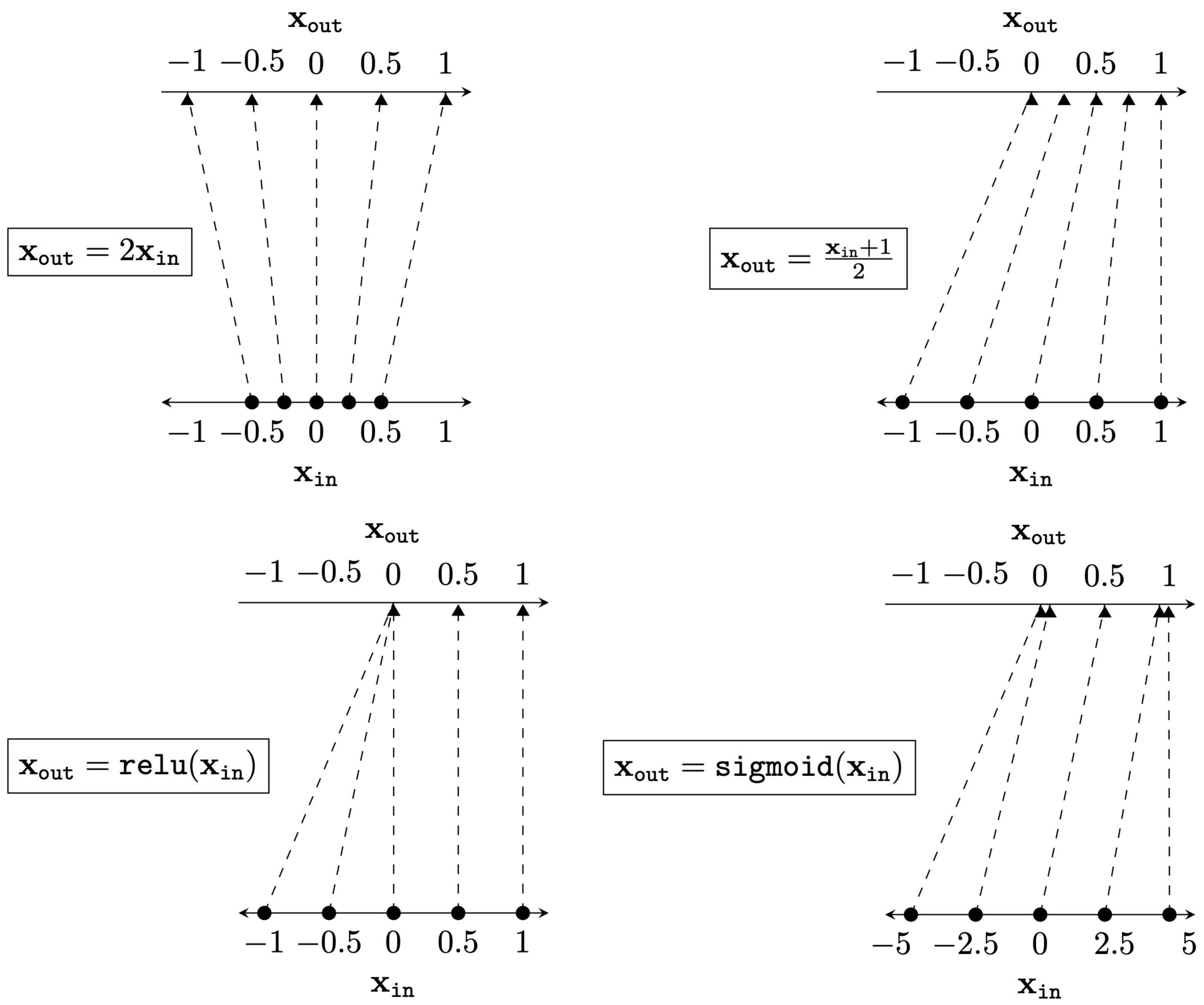
Two different ways to represent a function



$\Rightarrow$



Data transformations for a variety of neural net layers

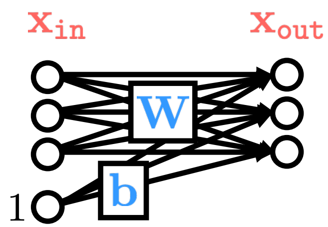


Activations

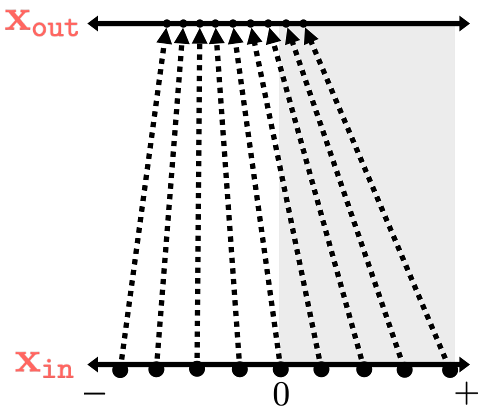
Parameters

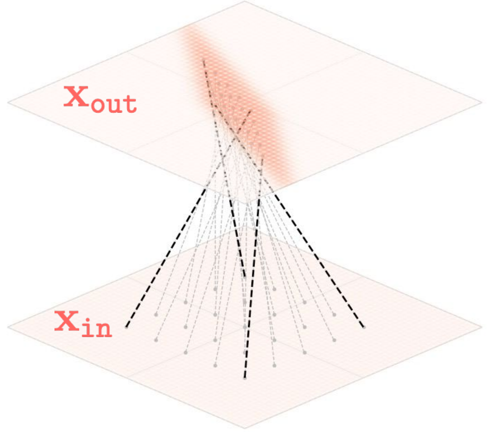
	Wiring graph	Equation	Mapping 1D	Mapping 2D

linear

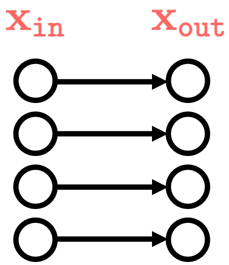


$$\mathbf{x}_{\mathtt{out}} = \mathbf{W}\mathbf{x}_{\mathtt{in}} + \mathbf{b}$$

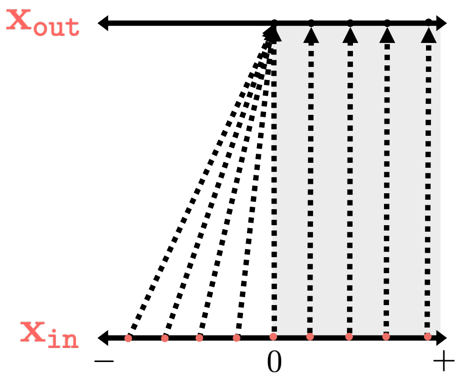


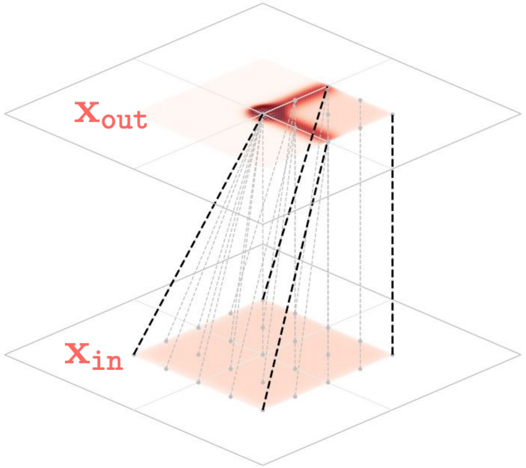


relu



$$x_{\mathtt{out}_i} = \max(x_{\mathtt{in}_i}, 0)$$

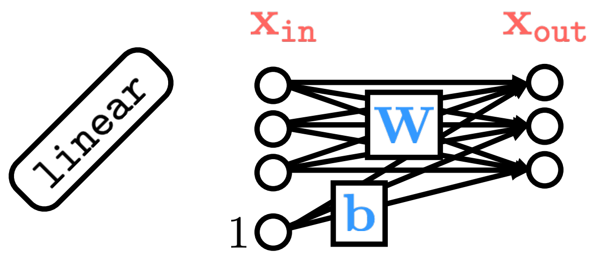




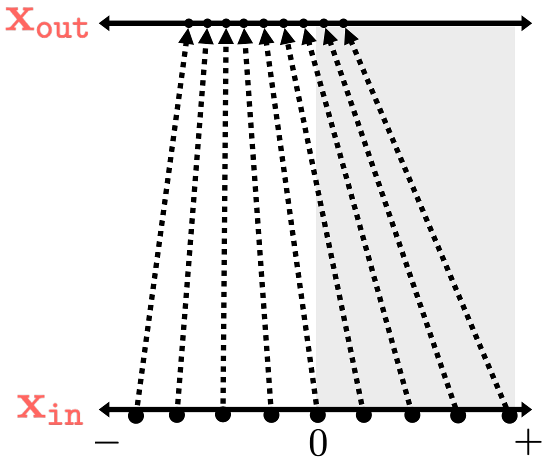
Activations

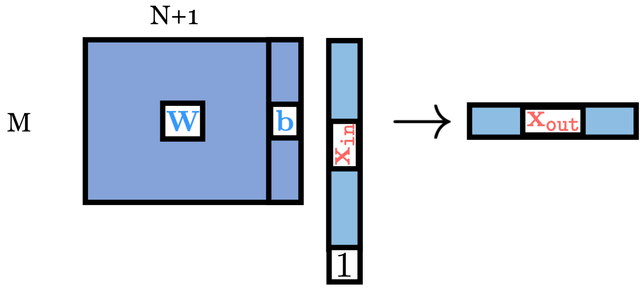
Parameters

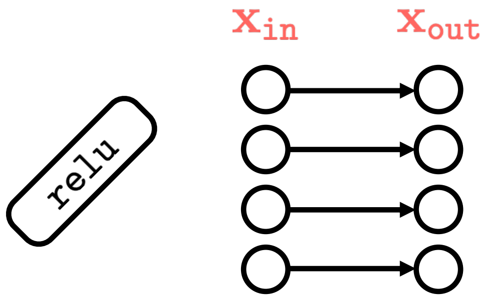
Wiring graph	Equation	Mapping	Matrix



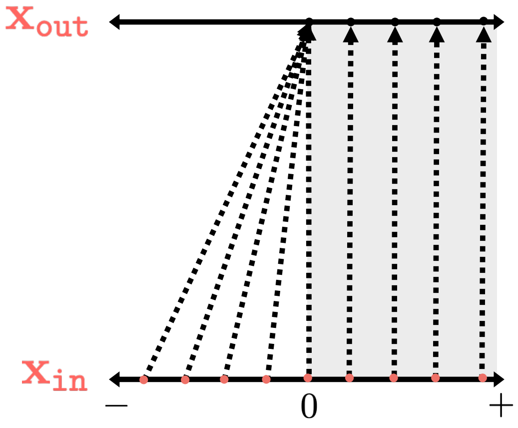
$$\mathbf{x}_{\mathtt{out}} = \mathbf{W}\mathbf{x}_{\mathtt{in}} + \mathbf{b}$$

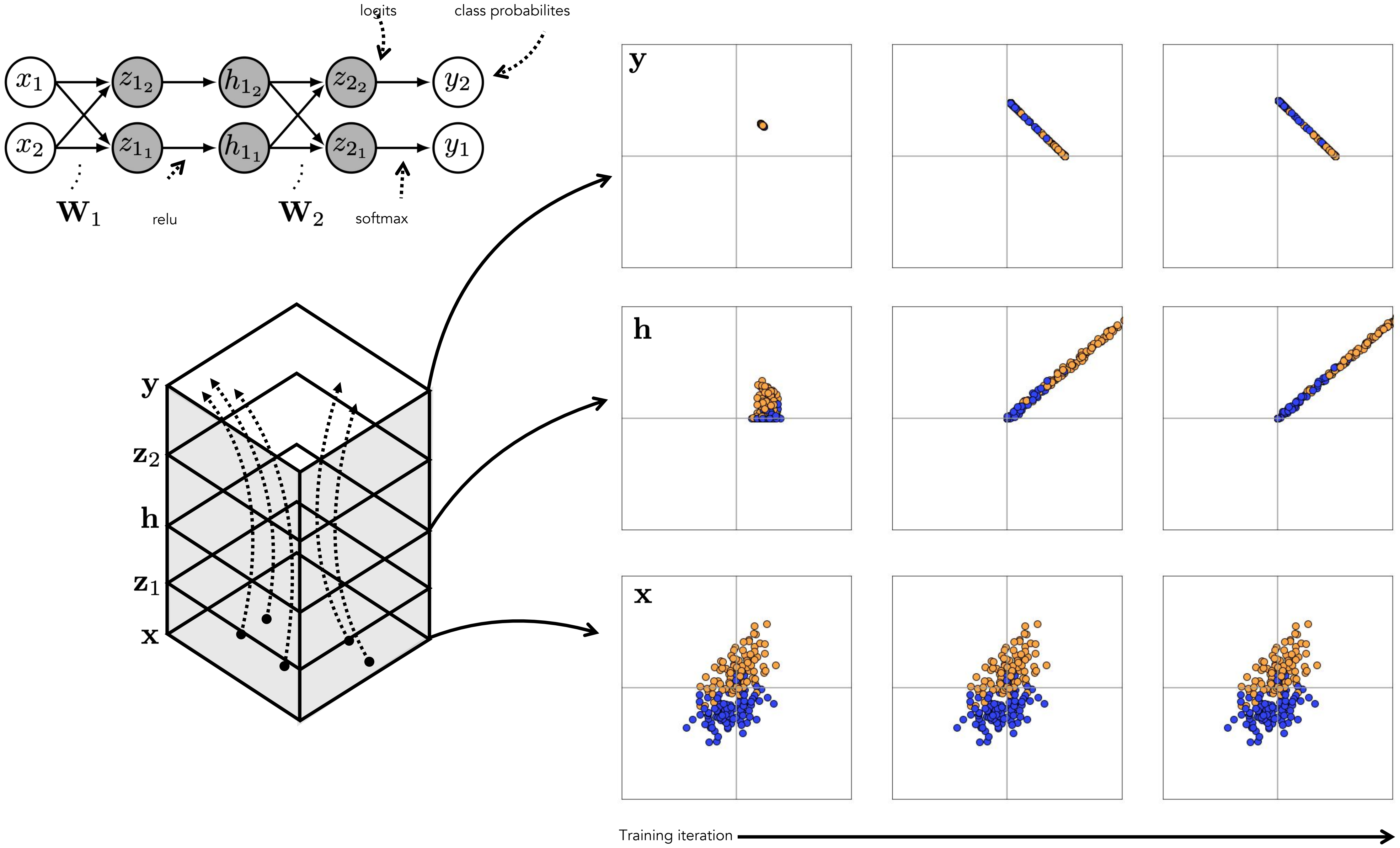




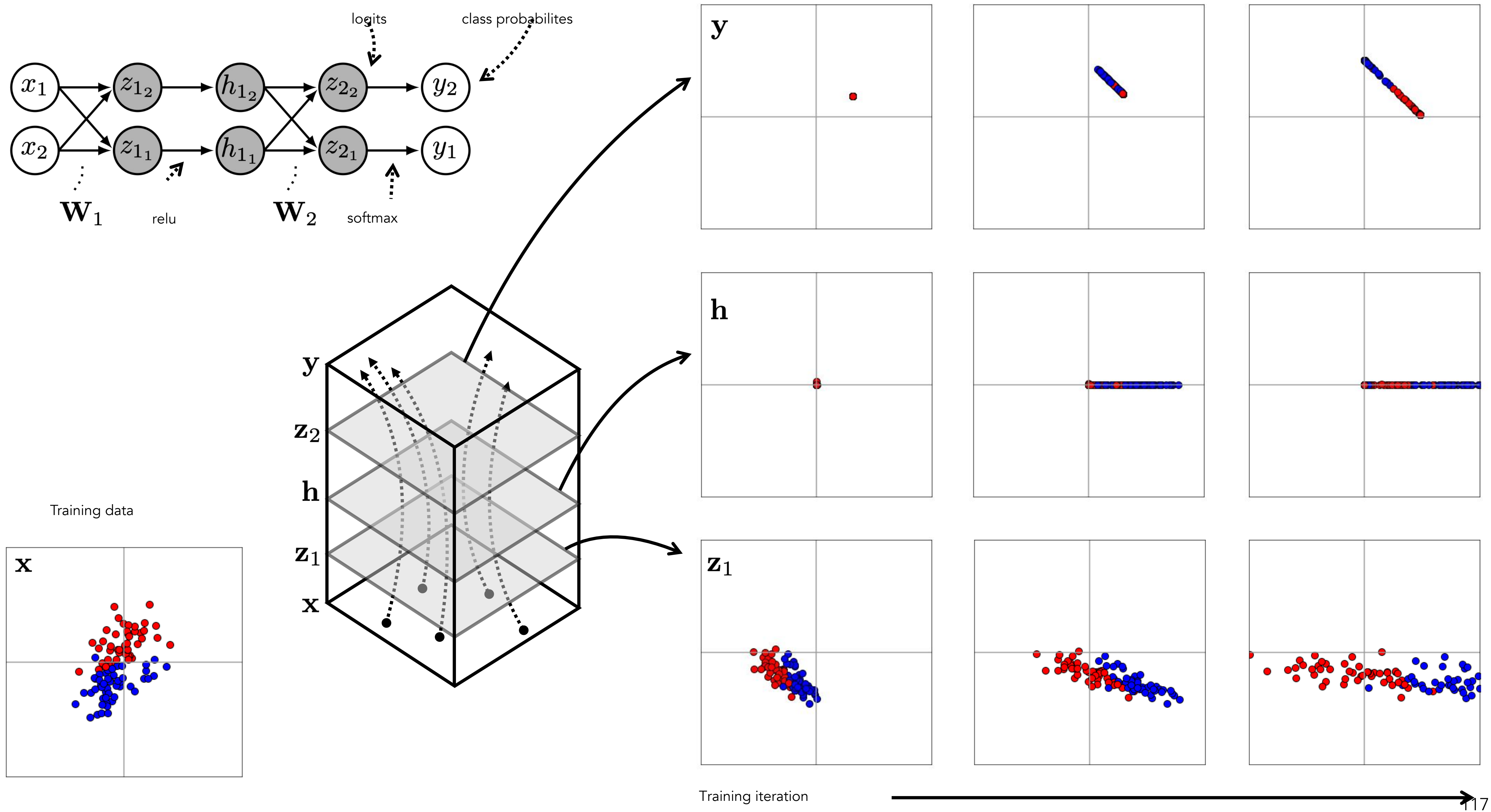


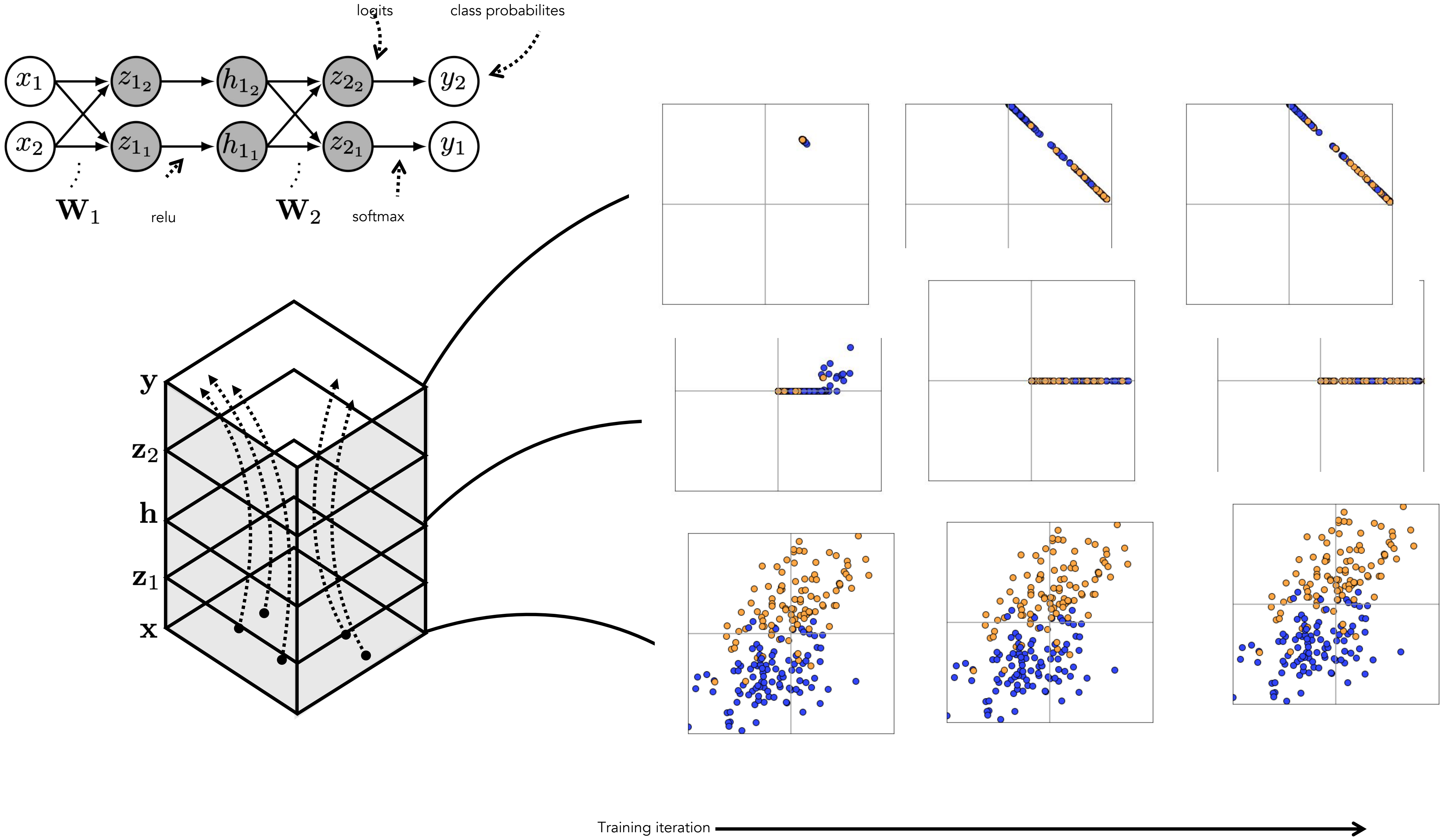
$$x_{\mathtt{out}_i} = \max(x_{\mathtt{in}_i}, 0)$$

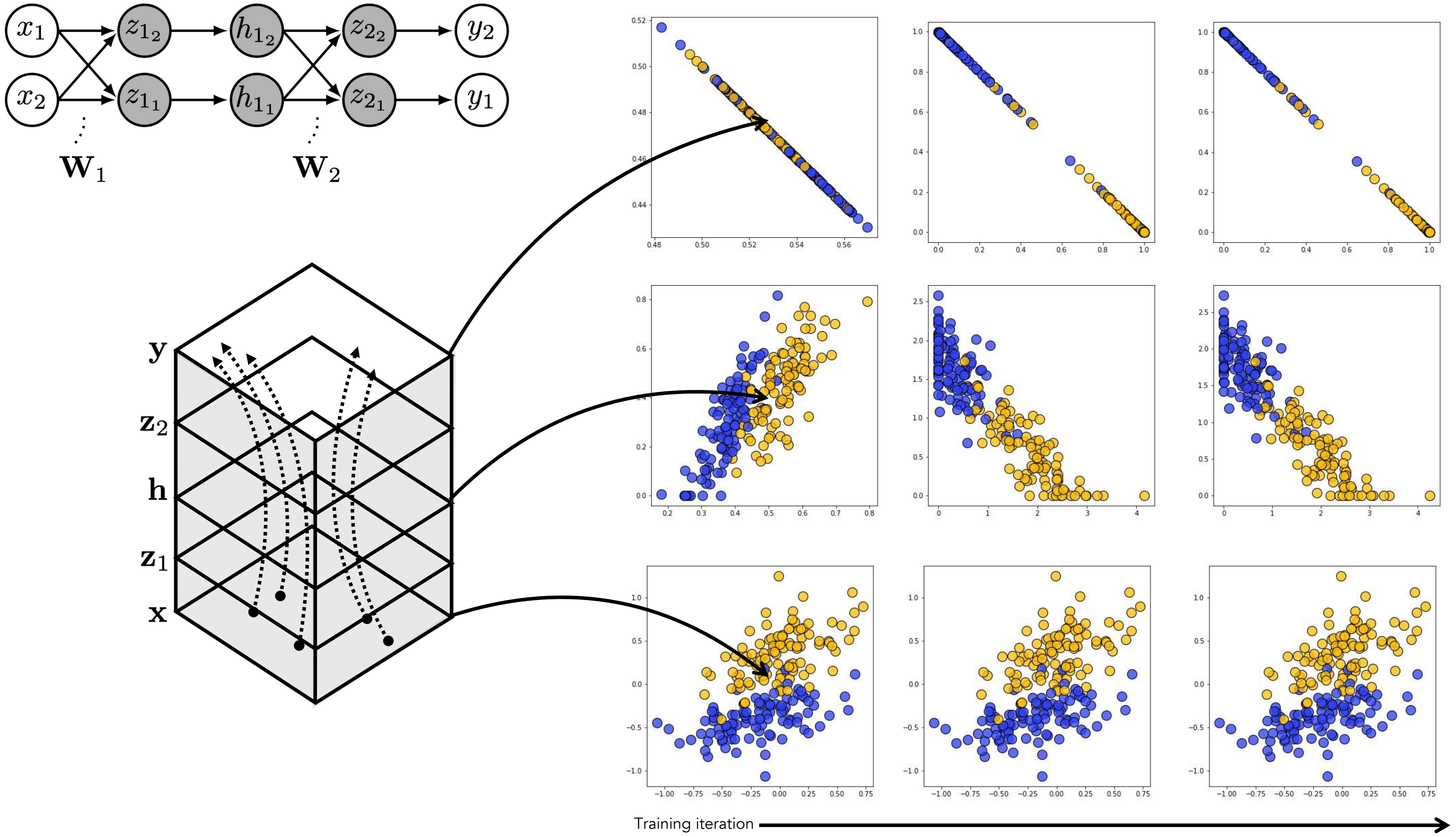


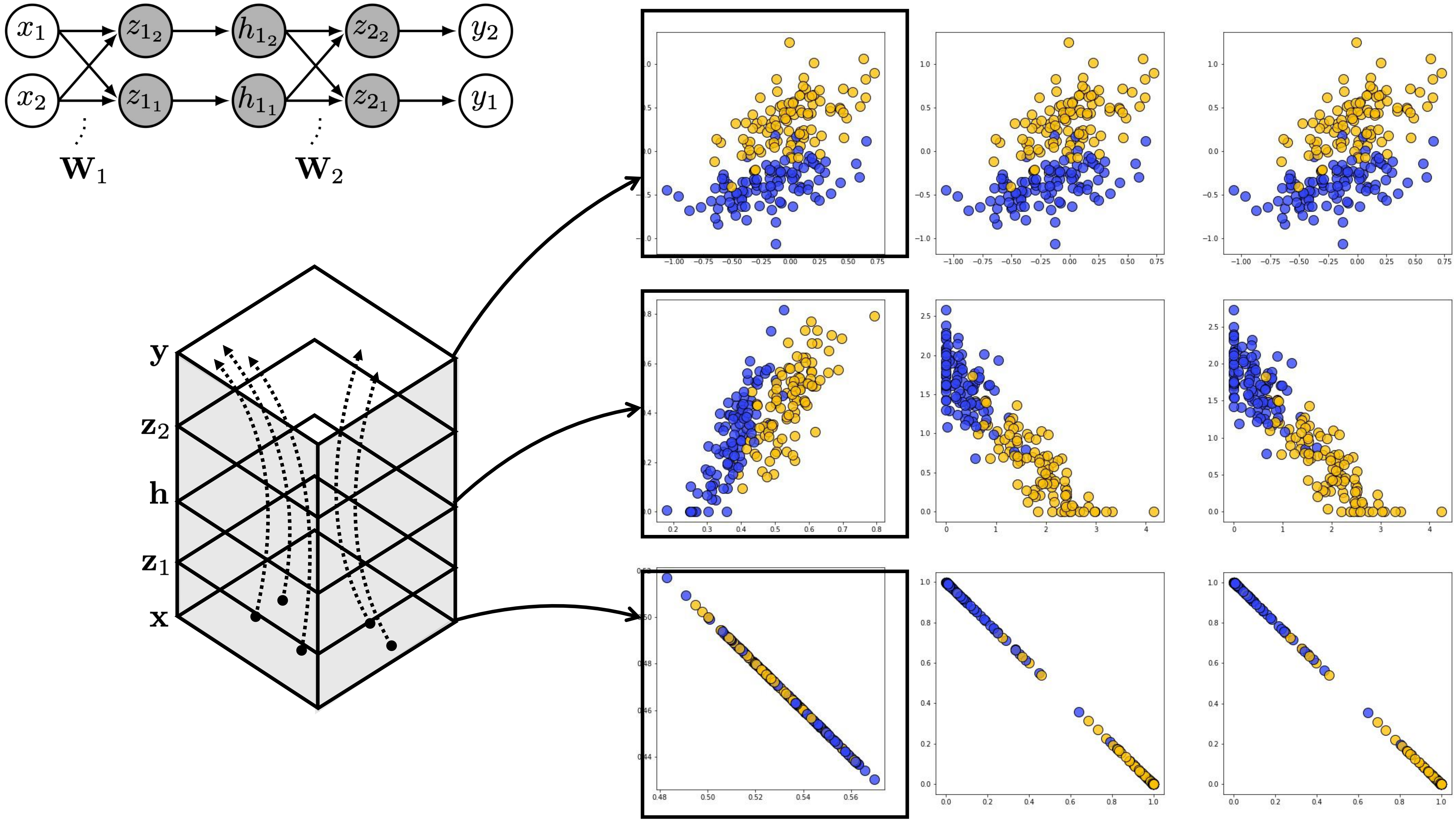


logits
class probabilites
x_1 x_2 z_{1_2} z_{1_1} h_{1_2} h_{1_1} z_{2_2} z_{2_1} y_2 y_1
$\mathbf{W}_1$
relu
$\mathbf{W}_2$
softmax
y
$\mathbf{z}_2$
h
$\mathbf{z}_1$
x
Training iteration

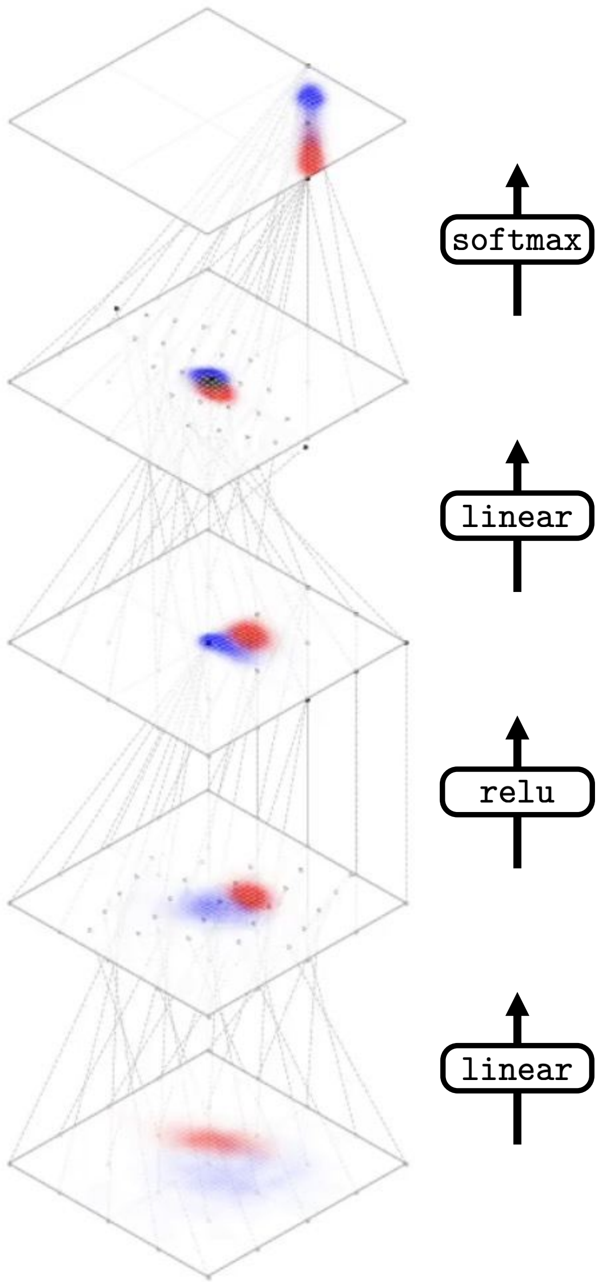

logits
class probabilites
x_1
x_2
z_{1_2}
z_{1_1}
h_{1_2}
h_{1_1}
z_{2_2}
z_{2_1}
y_2
y_1
$\mathbf{W}_1$
relu
$\mathbf{W}_2$
softmax
Training data
$\mathbf{x}$
$\mathbf{y}$
$\mathbf{z}_2$
$\mathbf{h}$
$\mathbf{z}_1$
$\mathbf{x}$
$\mathbf{y}$
$\mathbf{h}$
$\mathbf{z}_1$
Training iteration

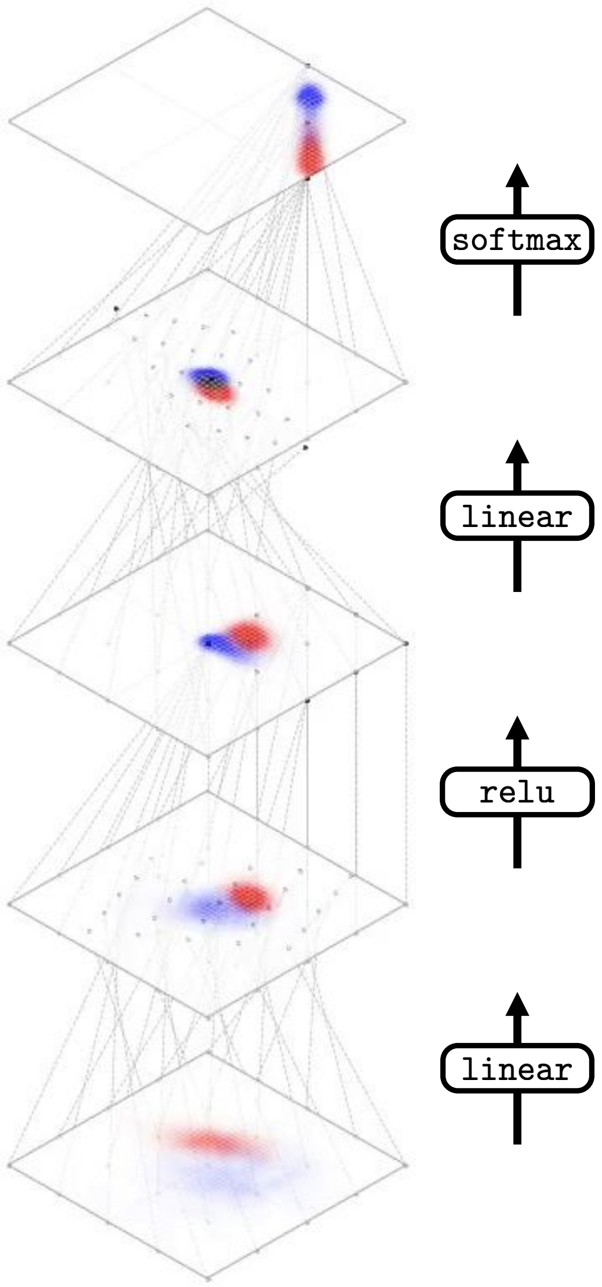

logits
class probabilites
W1
relu
W2
softmax
y
z2
h
z1
x
Training iteration

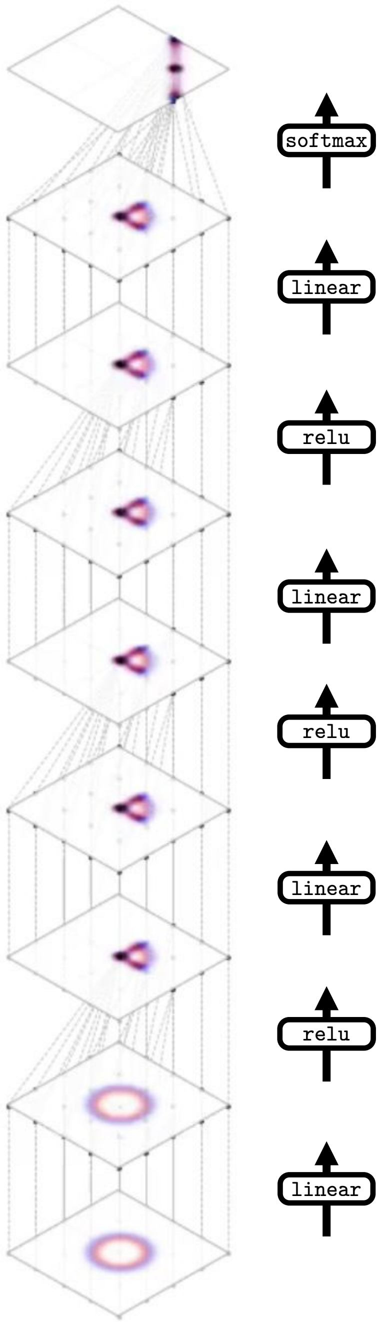

x_1
x_2
z_{1_2}
z_{1_1}
h_{1_2}
h_{1_1}
z_{2_2}
z_{2_1}
y_2
y_1
$\mathbf{W}_1$
$\mathbf{W}_2$
$\mathbf{y}$
$\mathbf{z}_2$
$\mathbf{h}$
$\mathbf{z}_1$
$\mathbf{x}$
Training iteration

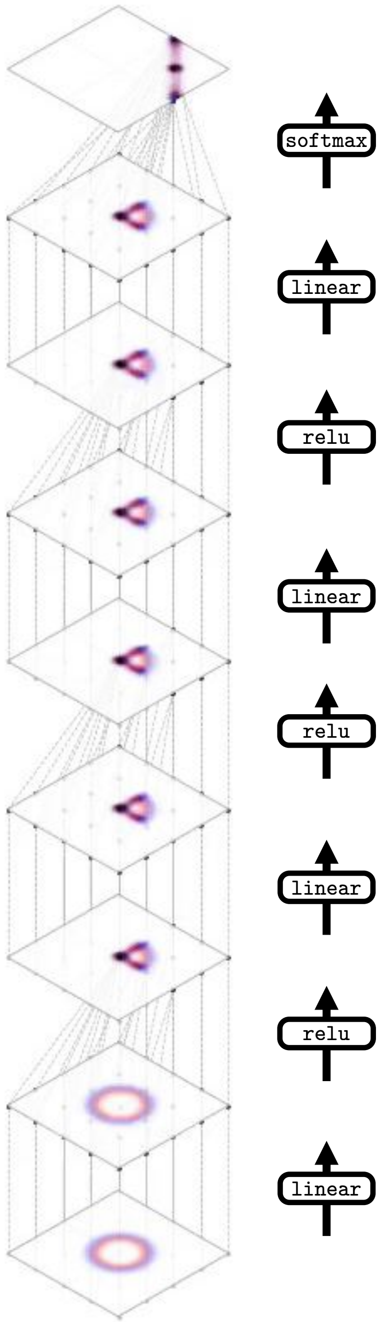


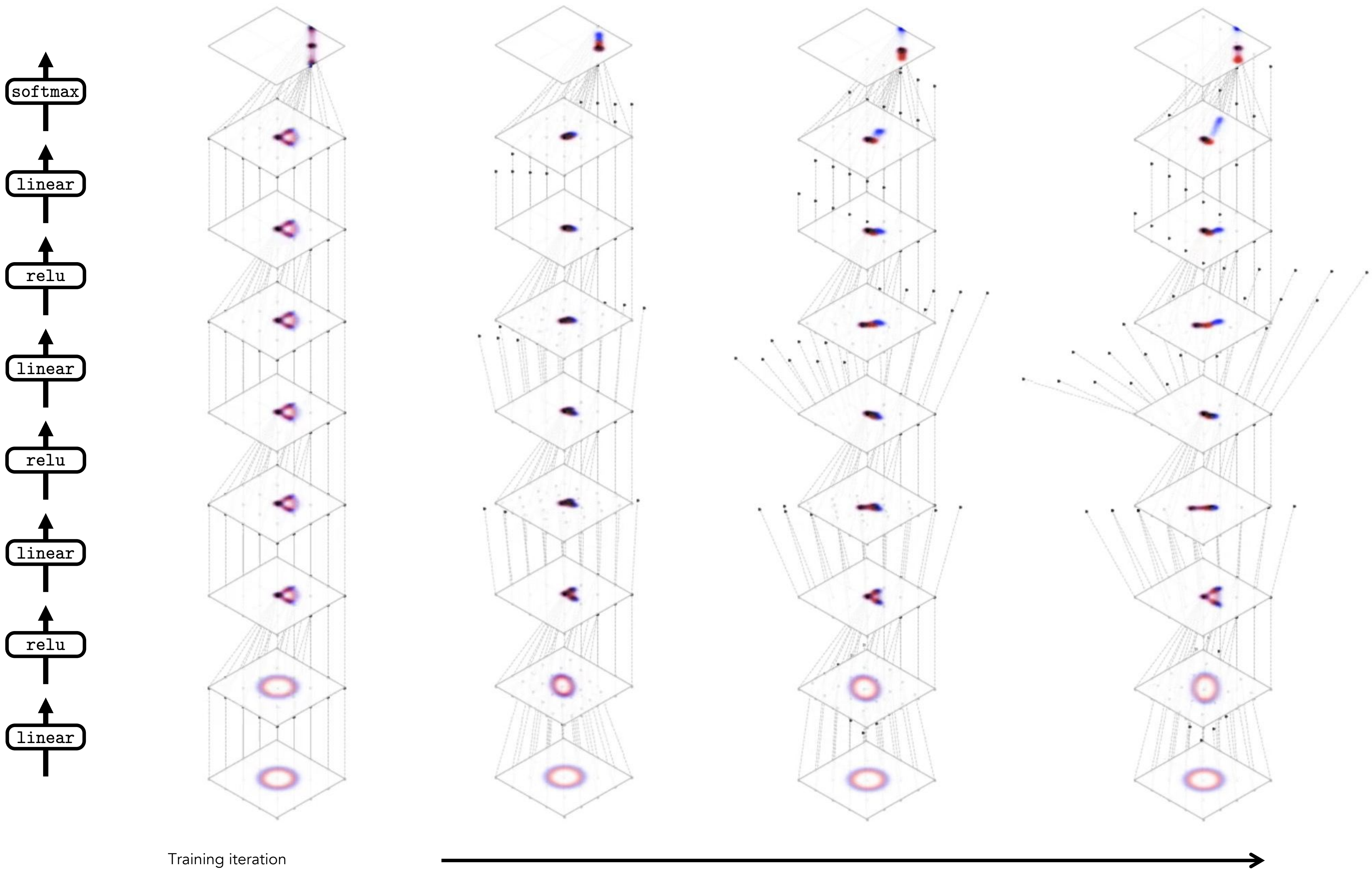
x_1
x_2
z_{1_2}
z_{1_1}
h_{1_2}
h_{1_1}
z_{2_2}
z_{2_1}
y_2
y_1
$\mathbf{W}_1$
$\mathbf{W}_2$
$\mathbf{y}$
$\mathbf{z}_2$
$\mathbf{h}$
$\mathbf{z}_1$
$\mathbf{x}$

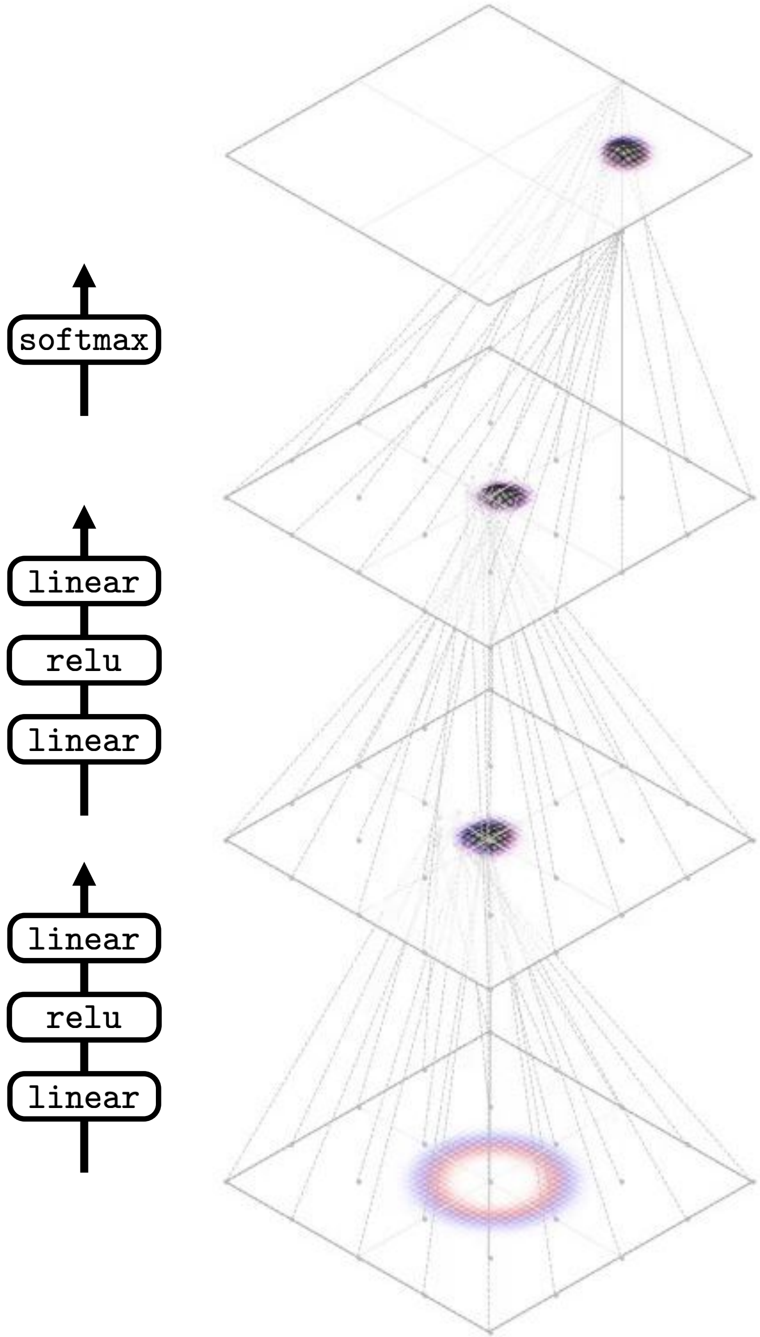

softmax
linear
relu
linear

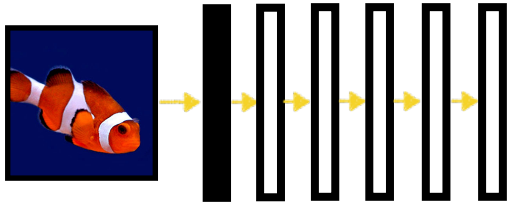

softmax
linear
relu
linear


softmax
linear
relu
linear
relu
linear
relu
linear

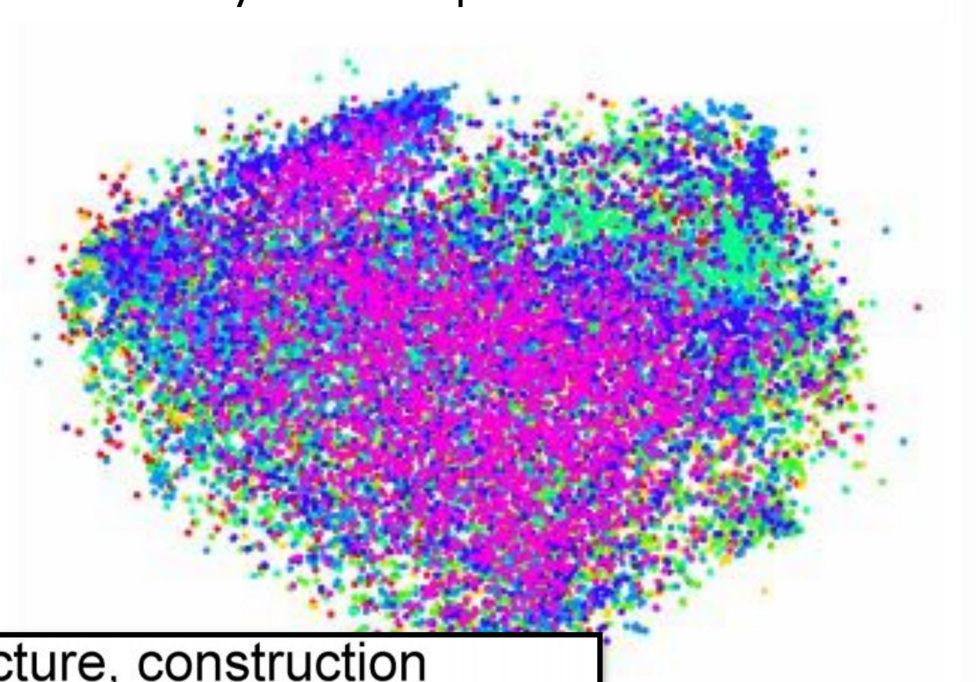

softmax
linear
relu
linear
relu
linear
relu
linear

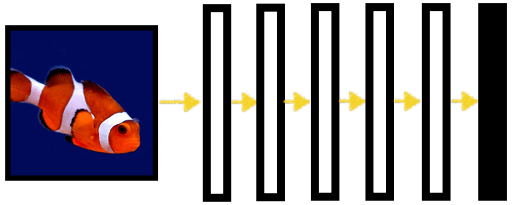

softmax
linear
relu
linear
relu
linear
relu
linear
Training iteration


softmax
linear
relu
linear
linear
relu
linear

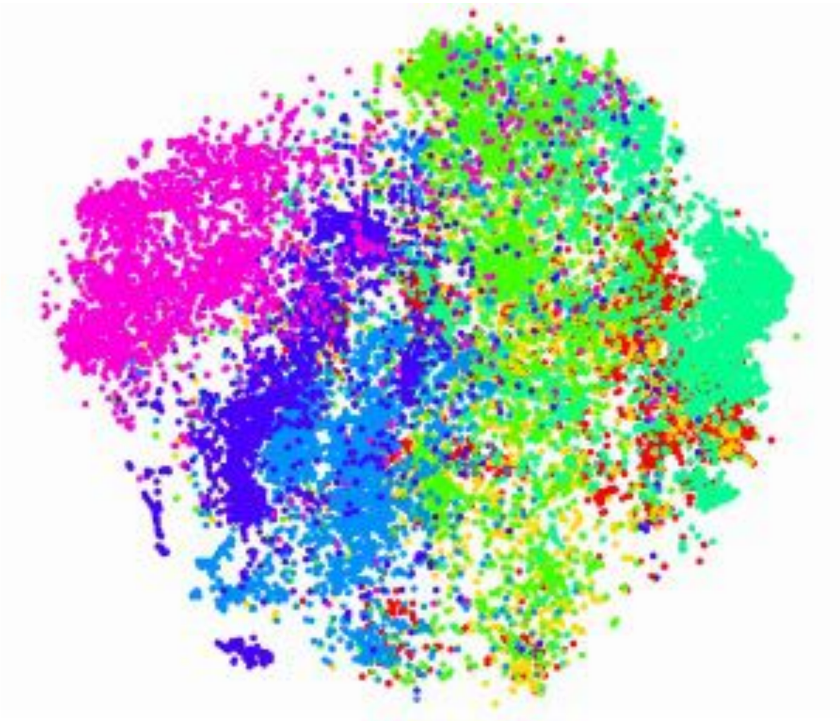


Layer 1 representation





Layer 6 representation



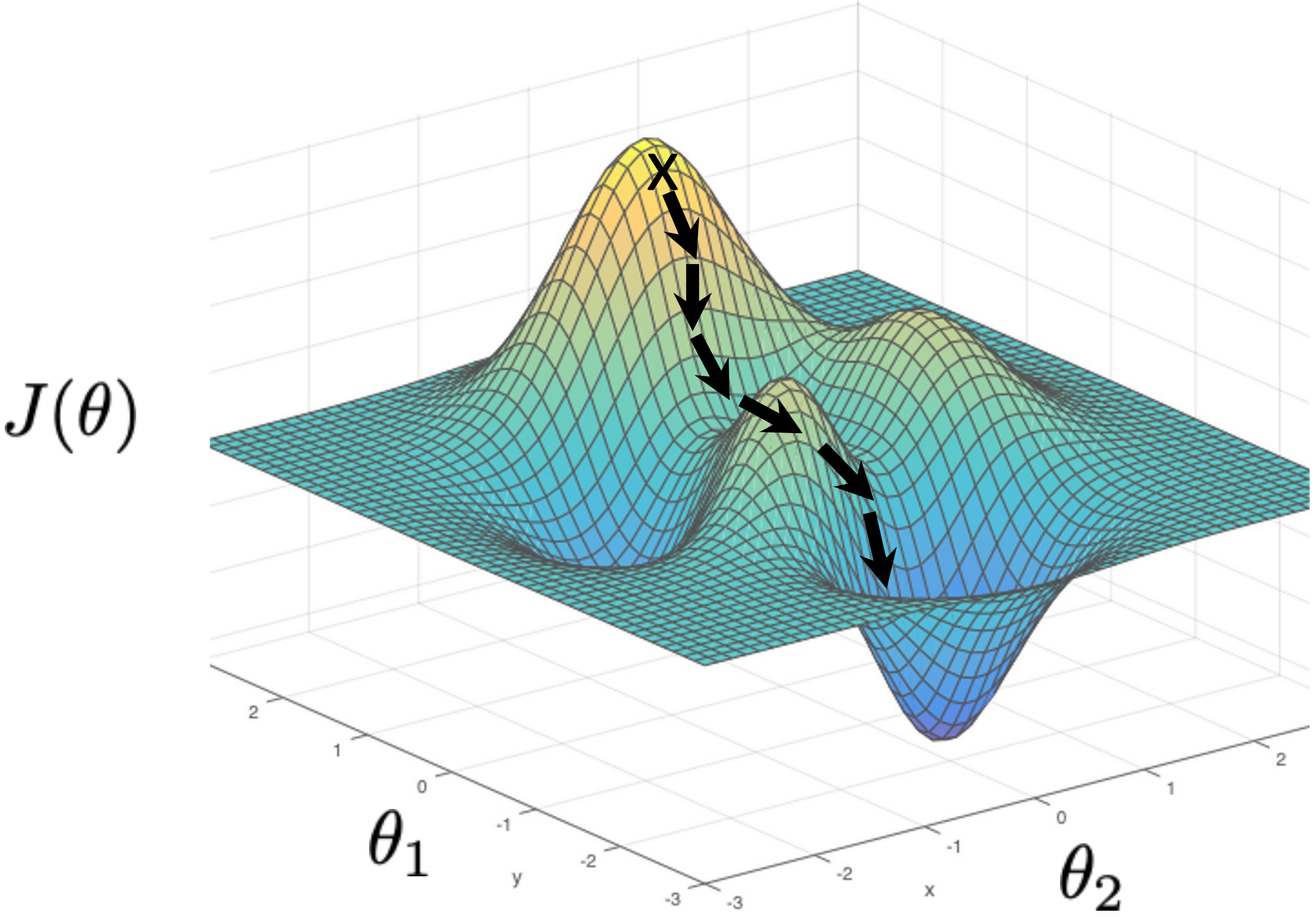
- structure, construction
- covering
- commodity, trade good, good
- conveyance, transport
- invertebrate
- bird
- hunting dog

[DeCAF, Donahue, Jia, et al. 2013]

[Visualization technique : t-sne, van der Maaten & Hinton, 2008]

Optimization

Gradient descent



$$\theta^* = \arg\min_{\theta} J(\theta)$$

Gradient descent

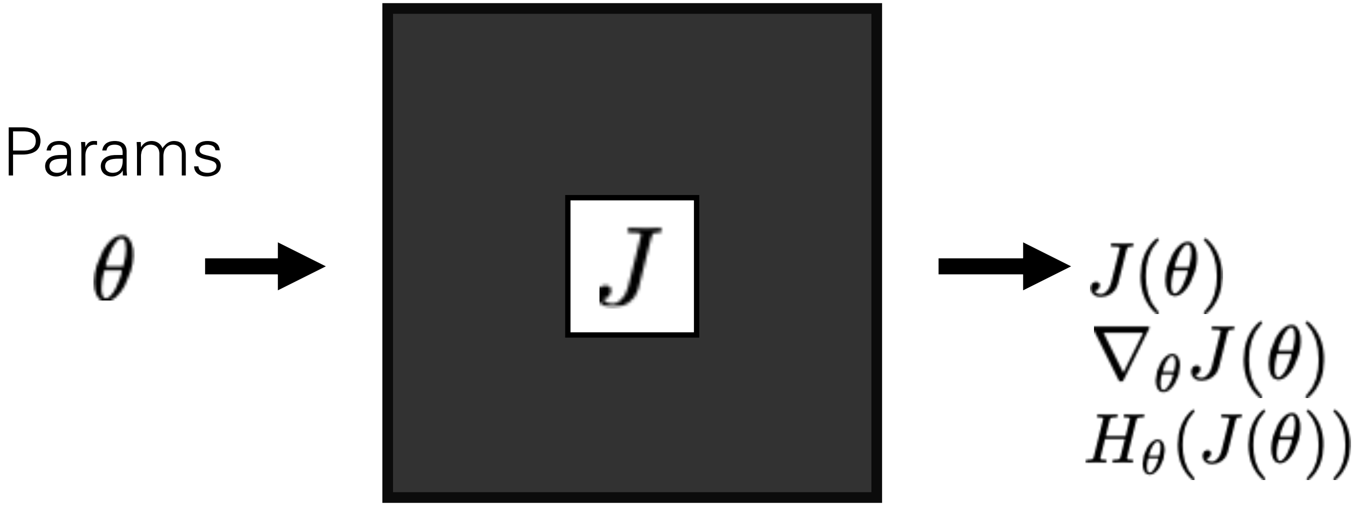
$$\theta^* = \arg\min_\theta \underbrace{\sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)}_{J(\theta)}$$

One iteration of gradient descent:

$$\theta^{t+1} = \theta^t - \eta_t \left.\frac{\partial J(\theta)}{\partial \theta}\right|_{\theta=\theta^t}$$

learning rate

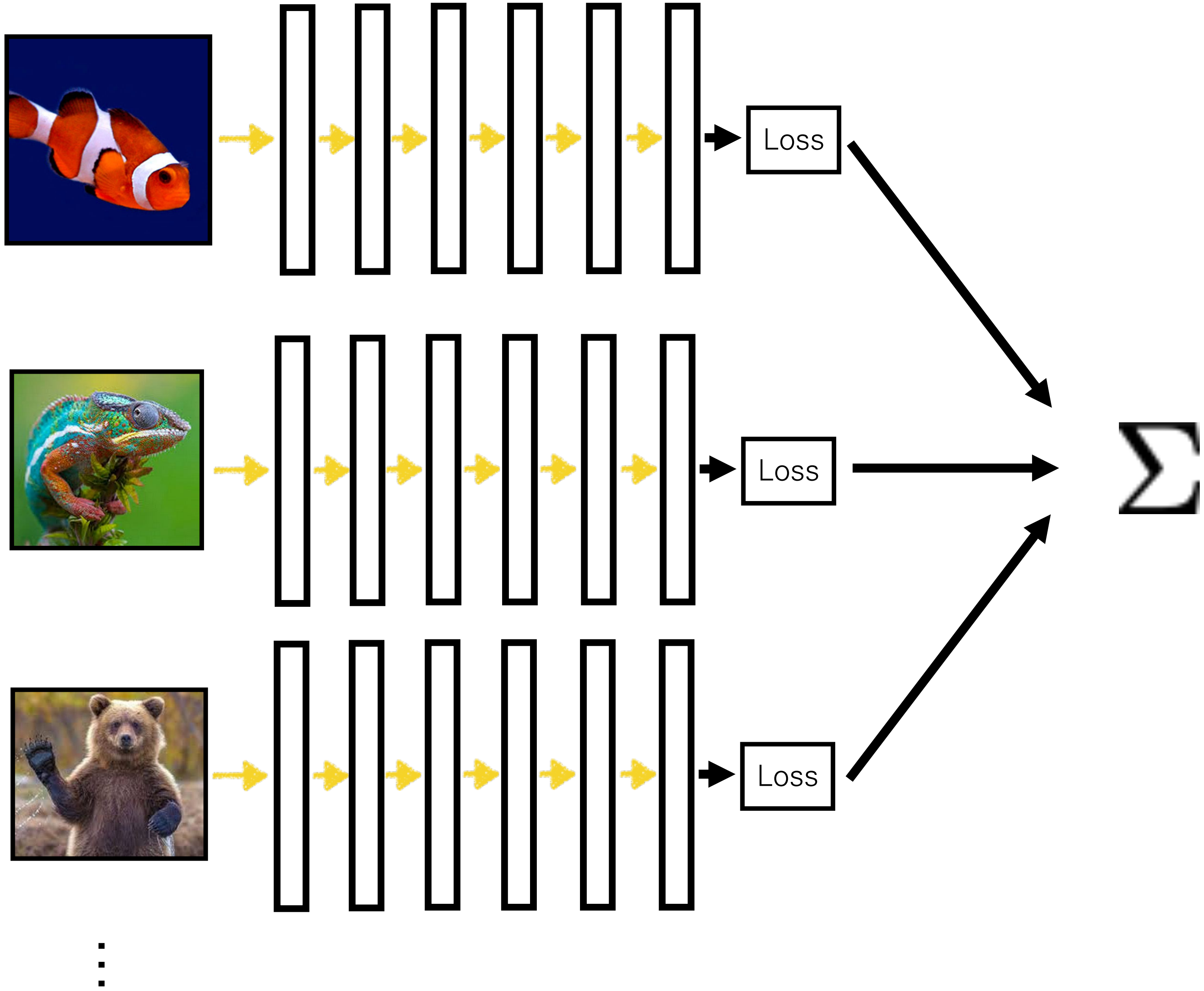
Optimization



$$\theta^* = \arg\min_\theta J(\theta)$$

- What's the knowledge we have about J?
  - We can evaluate $J(\theta)$ ← Black box optimization
  - We can evaluate $J(\theta)$ and $\nabla_\theta J(\theta)$ (Gradient) ← First order optimization
  - We can evaluate $J(\theta)$, $\nabla_\theta J(\theta)$, and $H_\theta(J(\theta))$ (Hessian) ← Second order optimization

Batch (parallel) processing



Stochastic gradient descent (SGD)

- Want to minimize overall loss function J, which is sum of individual losses over each example.
- In Stochastic gradient descent, compute gradient on sub-set (batch) of data.

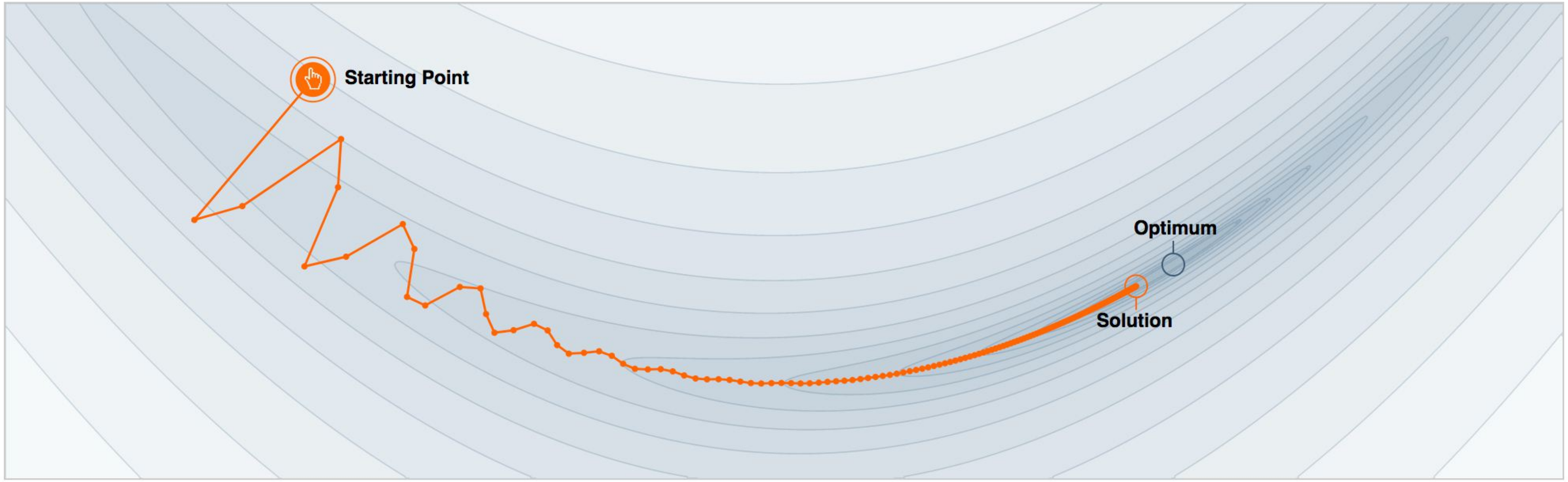
  If batchsize=1 then θ is updated after each example.

  If batchsize=N (full set) then this is standard gradient descent.
- Gradient direction is noisy, relative to average over all examples (standard gradient descent).
- Advantages
  - Faster: approximate total gradient with small sample
  - Implicit regularizer
- Disadvantages
  - High variance, unstable updates

Momentum

- Basic idea: like a ball rolling down a hill, we should build up speed so as to make faster progress when "on a roll"
- Can dampen oscillations in SGD updates
- Common in popular variants of SGD
    - Nesterov's method
    - RMSProp
    - Adam

Why Momentum Really Works



Step-size α = 0.02

0 0.003 0.006

Momentum β = 0.99

0.00 0.500 0.990

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

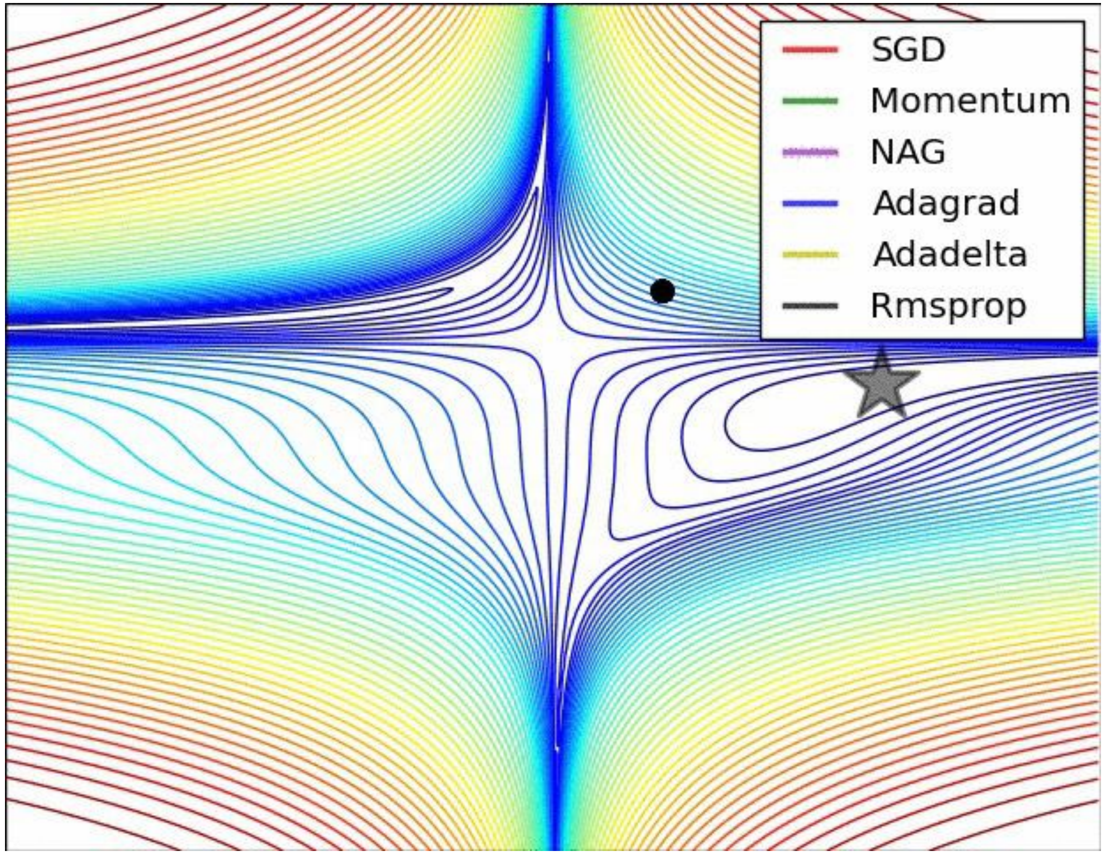
GABRIEL GOH
UC Davis

April. 4
2017

Citation:
Goh, 2017

[https://distill.pub/2017/momentum/]

Comparison of gradient descent variants



[http://ruder.io/optimizing-gradient-descent/]

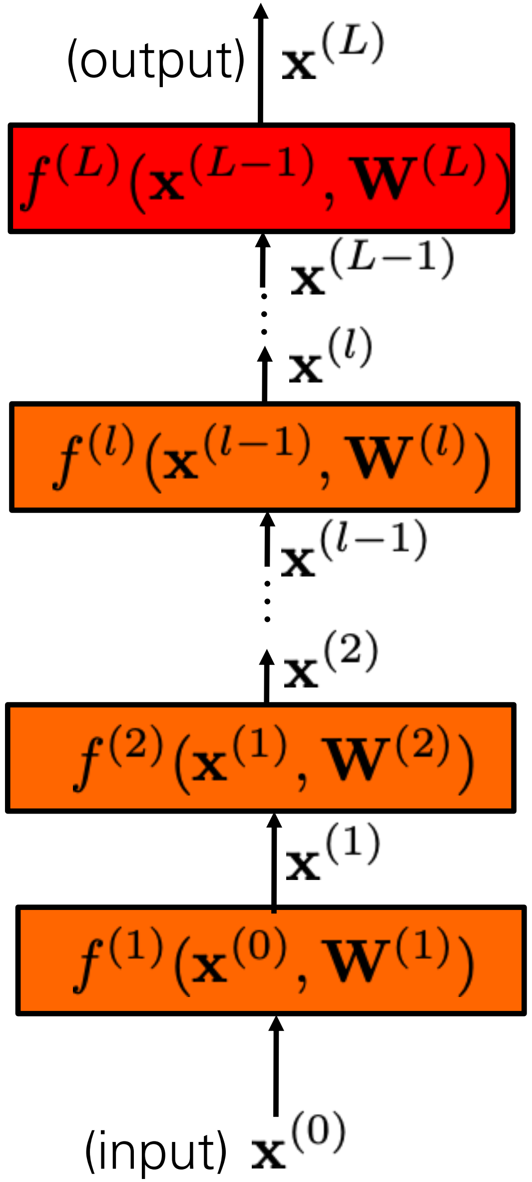
Backpropagation

Forward pass

- Consider model with L layers. Layer l has vector of weights $\mathbf{W}^{(l)}$
- **Forward pass:** takes input $\mathbf{x}^{(l-1)}$ and passes it through each layer $f^{(l)}$:

$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})$$

- Output of layer l is $\mathbf{x}^{(l)}$.
- Network output (top layer) is $\mathbf{x}^{(L)}$.



Forward pass

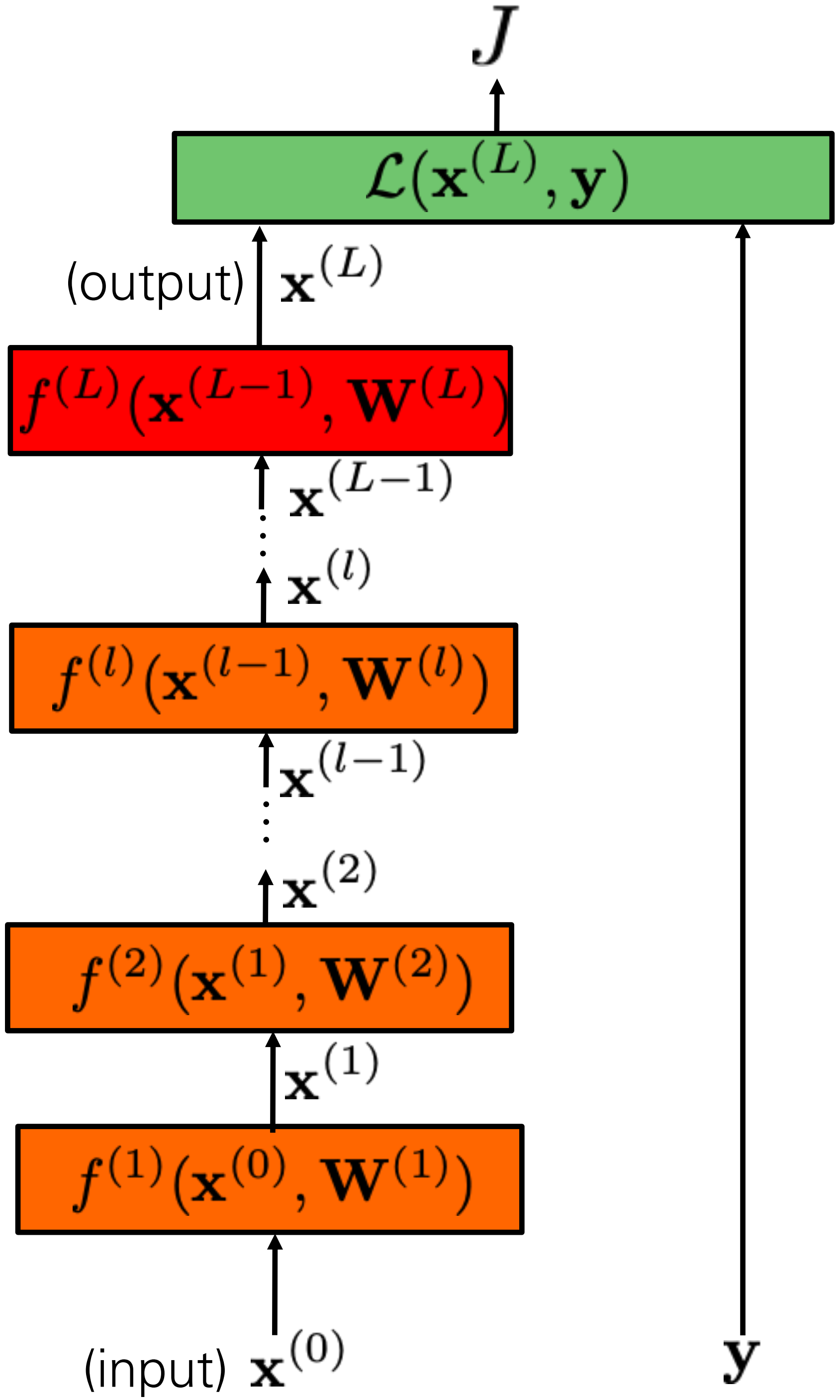
- Consider model with L layers. Layer l has vector of weights $\mathbf{W}^{(l)}$
- **Forward pass:** takes input $\mathbf{x}^{(l-1)}$ and passes it through each layer $f^{(l)}$:

$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})$$

- Output of layer l is $\mathbf{x}^{(l)}$.
- Network output (top layer) is $\mathbf{x}^{(L)}$.

- **Loss function** $\mathcal{L}$ compares $\mathbf{x}^{(L)}$ to $\mathbf{y}$.
- Overall energy is the sum of the cost over all training examples:

$$J = \sum_{i=1}^{N} \mathcal{L}(\mathbf{x}_i^{(L)}, \mathbf{y}_i)$$



Gradient descent

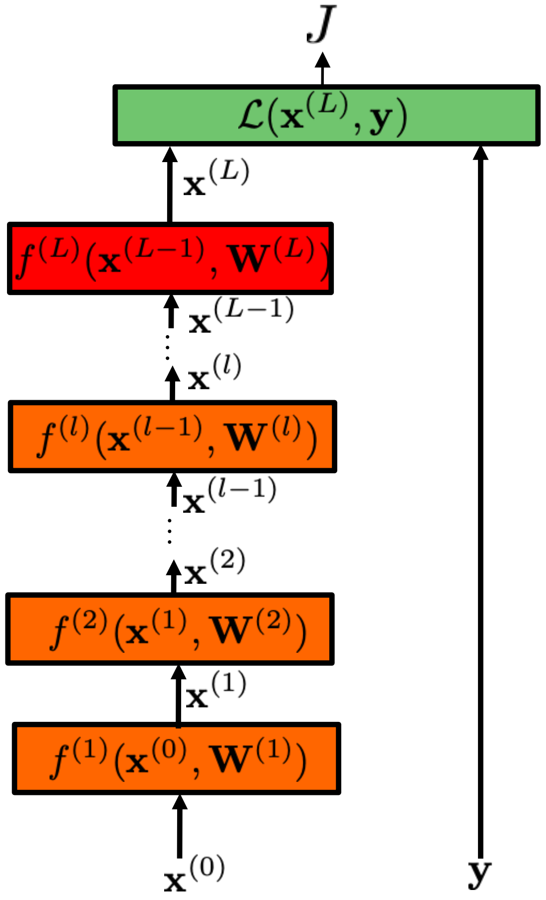
- We need to compute gradients of the cost with respect to model parameters $\mathbf{W}^{(l)}$.

- By design, each layer is differentiable with respect to its parameters and input.

Computing gradients

To compute the gradients, we could start by writing the full energy J as a function of the network parameters.

$$J(\mathbf{W}) = \sum_{i=1} \mathcal{L}(f^{(L)}(\ldots f^{(2)}(f^{(1)}(\mathbf{x}_i^{(0)}, \mathbf{W}^{(1)}), \mathbf{W}^{(2)}), \ldots \mathbf{W}^{(L)}), \mathbf{y}_i)$$



And then compute the partial derivatives… instead, we can use the chain rule to derive a compact algorithm: **backpropagation**

Computing gradients

The energy J is the sum of the losses associated to each training example $\{\mathbf{x}_i^{(0)}, \mathbf{y}_i\}$

$$J(\mathbf{W}) = \sum_{i=1}^{N} \mathcal{L}(\mathbf{x}_i^{(L)}, \mathbf{y}_i; \mathbf{W})$$

Its gradient with respect to each of the network's parameters w is:

$$\frac{\partial J(\mathbf{W})}{\partial w} = \sum_{i=1}^{N} \frac{\partial \mathcal{L}(\mathbf{x}_i^{(L)}, \mathbf{y}_i; \mathbf{W})}{\partial w}$$

is how much J varies when the parameter w is varied.

Computing gradients

We could write the loss function to get the gradients as:

$$\mathcal{L}(\mathbf{x}^{(L)}, \mathbf{y}; \mathbf{W}) = \mathcal{L}(f^{(L)}(\mathbf{x}^{(L-1)}, \mathbf{W}^{(L)}), \mathbf{y})$$

If we compute the gradient with respect to the parameters of the last layer (output layer) $W^{(L)}$, using the **chain rule**:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \cdot \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \cdot \frac{\partial f^{(L)}(\mathbf{x}^{(L-1)}, \mathbf{W}^{(L)})}{\partial \mathbf{W}^{(L)}}$$

(How much the cost changes when we change $W^{(L)}$ is the product between how much the loss changes when we change the output of the last layer and how much the output changes when we change the layer parameters.)

Computing gradients: loss layer

If we compute the gradient with respect to the parameters of the last layer (output layer) $W^{(L)}$, using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \cdot \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \cdot \frac{\partial f^{(L)}(\mathbf{x}^{(L-1)}, \mathbf{W}^{(L)})}{\partial \mathbf{W}^{(L)}}$$

For example, for an Euclidean loss:

$$\mathcal{L}(\mathbf{x}^{(L)}, \mathbf{y}) = \frac{1}{2} \left\| \mathbf{x}^{(L)} - \mathbf{y} \right\|_2^2$$

The gradient is:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} = \mathbf{x}^{(L)} - \mathbf{y}$$

Will depend on the layer structure and non-linearity.

Computing gradients: layer l

We could write the full loss function to get the gradients:

$$\mathcal{L}(\mathbf{x}^{(L)}, \mathbf{y}; \mathbf{W}) = \mathcal{L}(f^{(L)}(\ldots f^{(2)}(f^{(1)}(\mathbf{x}^{(0)}, \mathbf{W}^{(1)}), \mathbf{W}^{(2)}), \ldots \mathbf{W}^{(L)}), \mathbf{y})$$

If we compute the gradient with respect to w_i, using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \cdot \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{x}^{(L-1)}} \cdot \frac{\partial \mathbf{x}^{(L-1)}}{\partial \mathbf{x}^{(L-2)}} \cdots \frac{\partial \mathbf{x}^{(l+1)}}{\partial \mathbf{x}^{(l)}}}_{\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}}} \cdot \frac{\partial \mathbf{x}^{(l)}}{\partial \mathbf{W}^{(l)}}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}}$$

And this can be computed iteratively!

$$\frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{W}^{(l)}}$$

This is easy.

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \cdot \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{x}^{(L-1)}} \cdot \frac{\partial \mathbf{x}^{(L-1)}}{\partial \mathbf{x}^{(L-2)}} \cdots \frac{\partial \mathbf{x}^{(l+1)}}{\partial \mathbf{x}^{(l)}} \cdot \frac{\partial \mathbf{x}^{(l)}}{\partial \mathbf{W}^{(l)}}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}}$$

$$\frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{W}^{(l)}}$$

If we have the value of $\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}}$ we can compute the gradient at the layer below as:

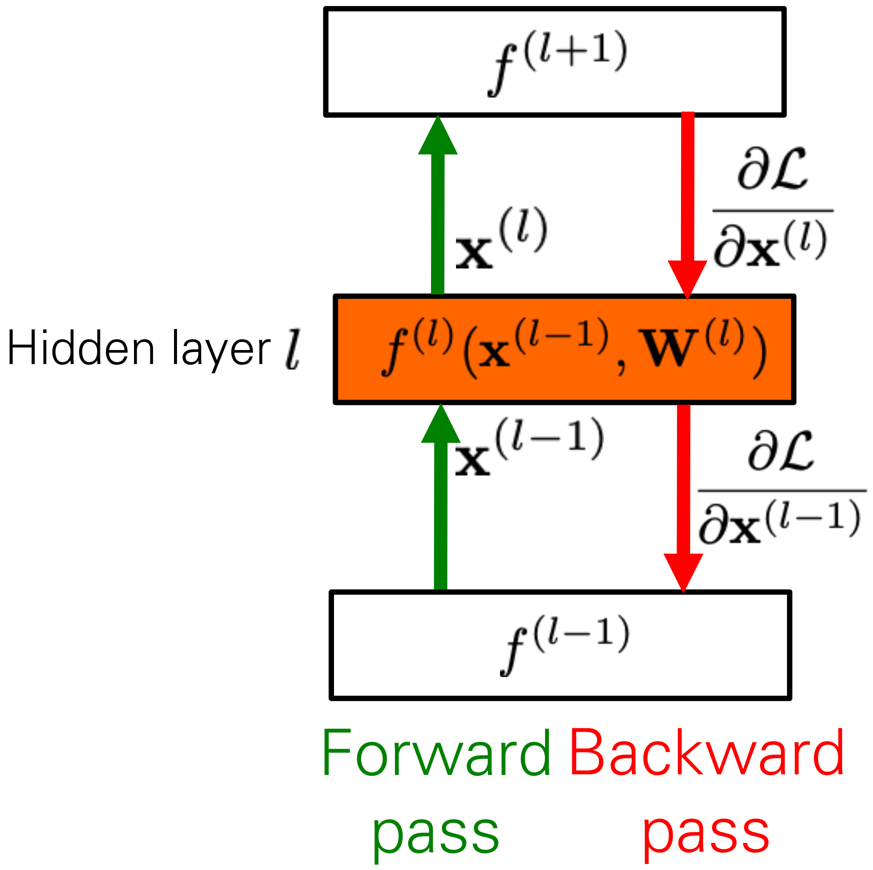
$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l-1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}} \cdot \frac{\partial \mathbf{x}^{(l)}}{\partial \mathbf{x}^{(l-1)}}$$

Gradient layer l-1

Gradient layer l

$$\frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{x}^{(l-1)}}$$

Backpropagation

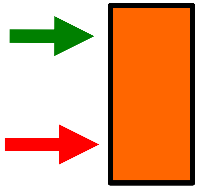


— Goal: to update parameters of layer l

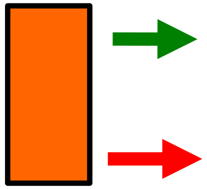
- Layer l has two inputs (during training)

$\mathbf{x}^{(l-1)}$

$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}}$



- We compute the outputs



$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l-1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}} \cdot \frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{x}^{(l-1)}}$$

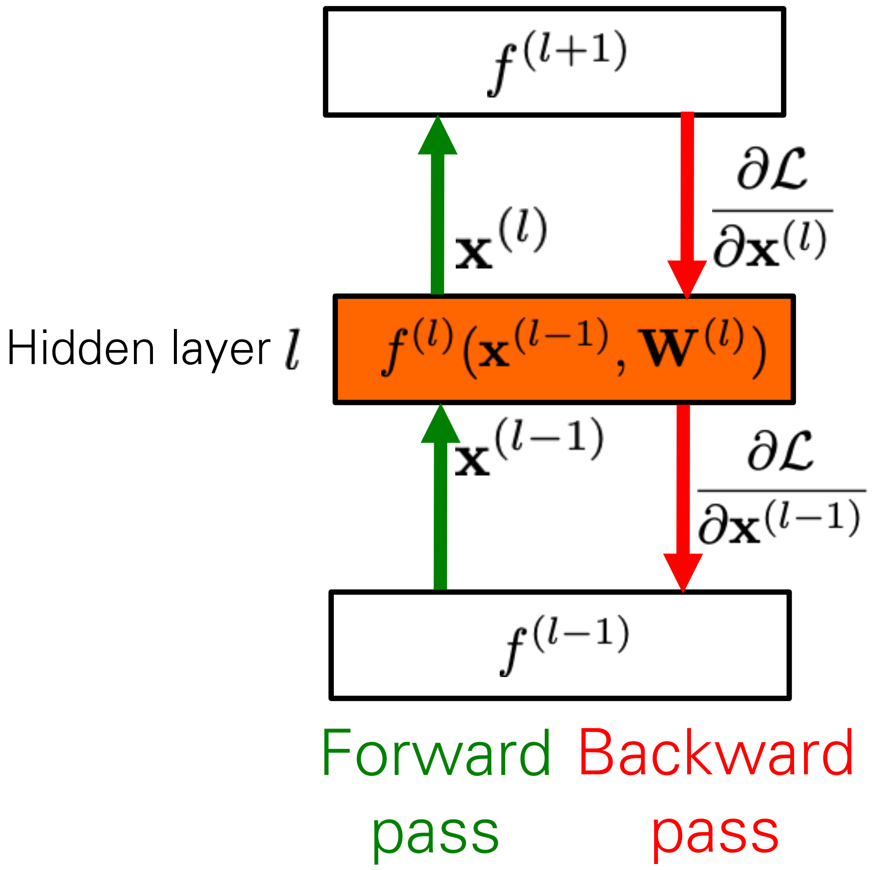
- To compute the output, we need:

$$\frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{x}^{(l-1)}}$$

- To compute the weight update, we need:

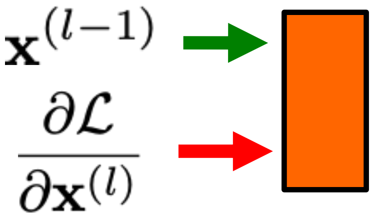
$$\frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{W}^{(l)}}$$

Backpropagation

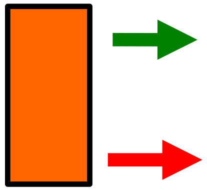


— Goal: to update parameters of layer l

- Layer l has two inputs (during training)



- We compute the outputs



$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l-1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}} \cdot \frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{x}^{(l-1)}}$$

- The weight update equation is:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}} \cdot \frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{W}^{(l)}}$$

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} + \eta \left(\frac{\partial J}{\partial \mathbf{W}^{(l)}} \right)^T$$

(sum over all training examples to get J)

Backpropagation Summary

- Forward pass: for each training example, compute the outputs for all layers:

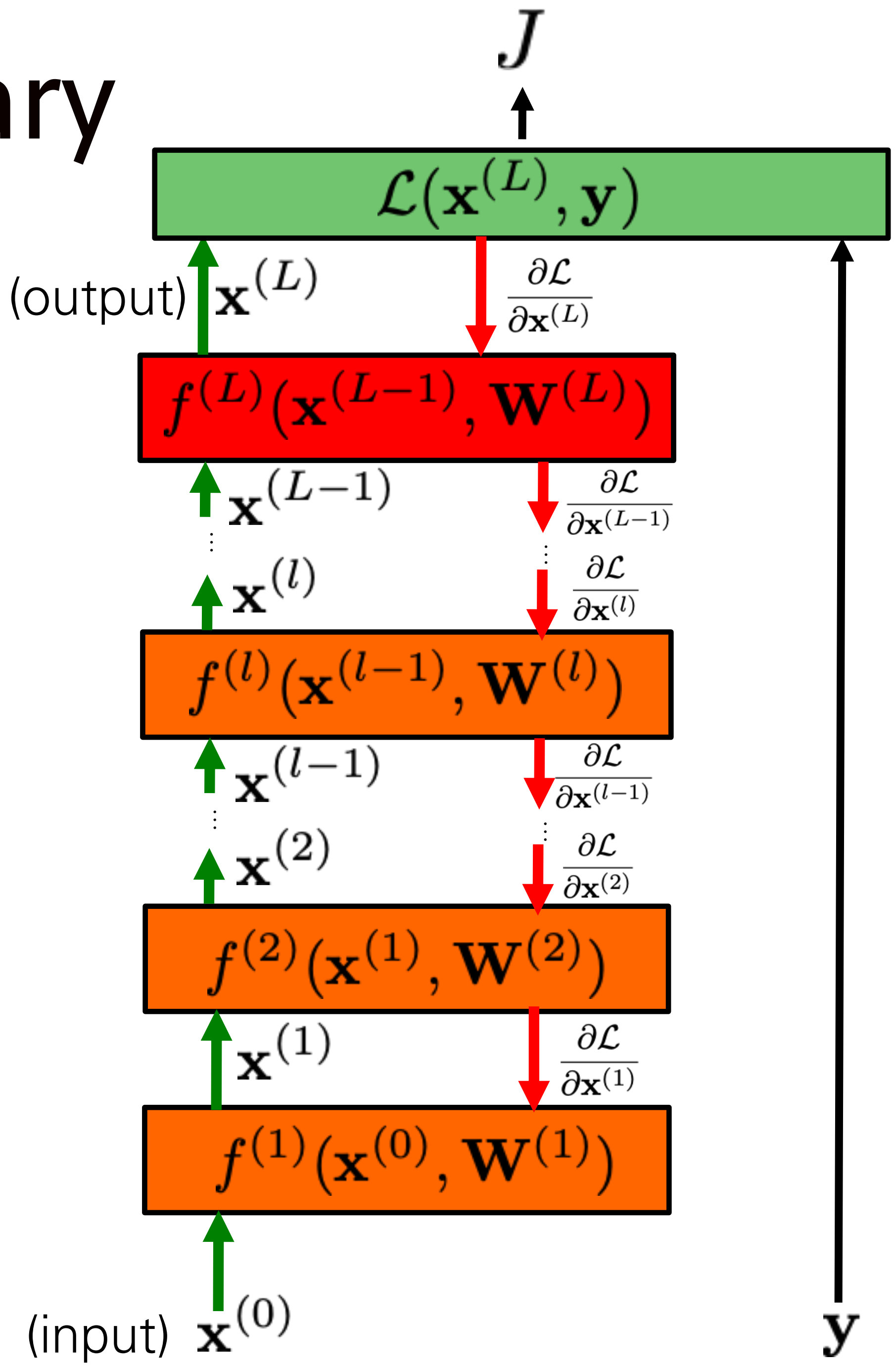
$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})$$

- Backwards pass: compute loss derivatives iteratively from top to bottom:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l-1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}} \cdot \frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{x}^{(l-1)}}$$

- Compute gradients w.r.t. weights, and update weights:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}} \cdot \frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{W}^{(l)}}$$

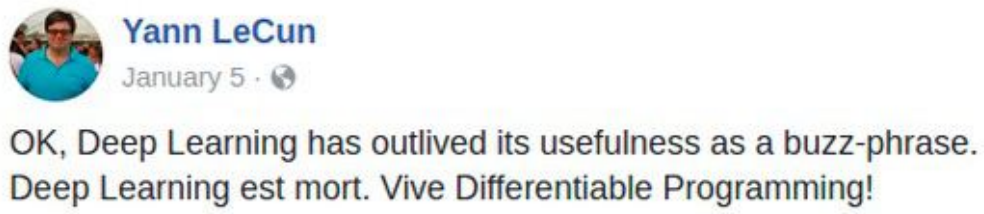


Differentiable programming

Deep nets are popular for a few reasons:

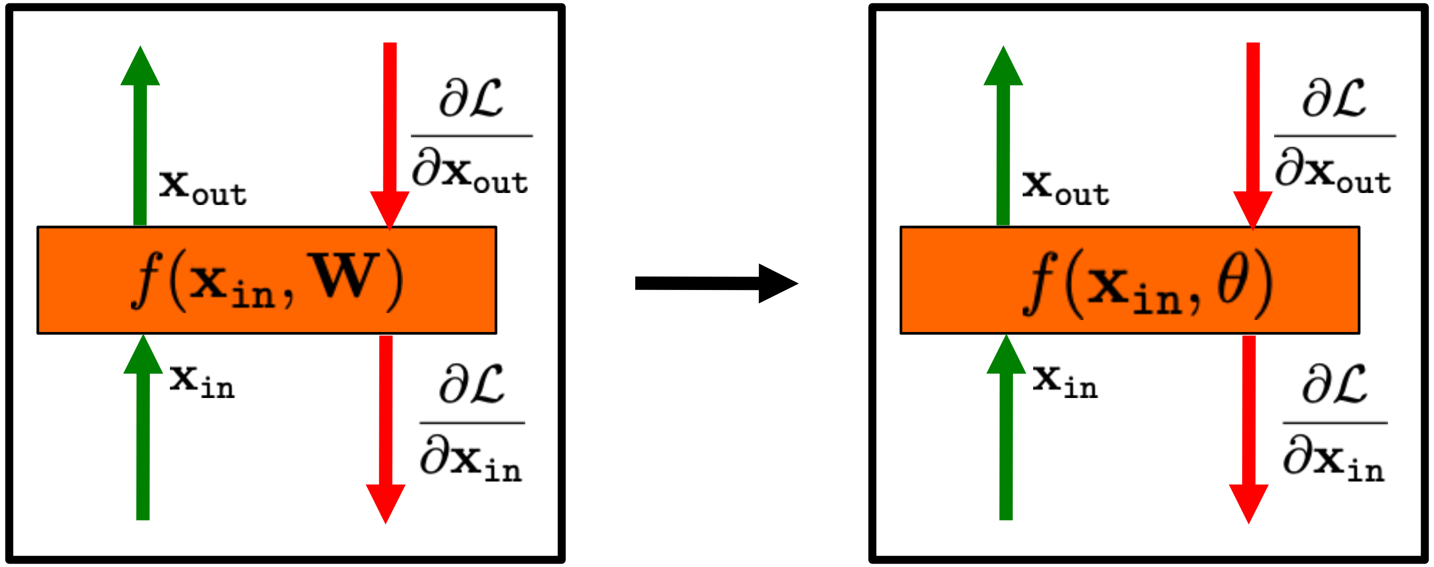
1. High capacity
2. Easy to optimize (differentiable)
3. Compositional “block based programming”

An emerging term for general models with these properties is **differentiable programming.**



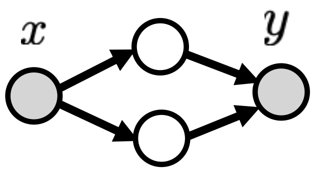
Differentiable programming

Deep learning → Differentiable programming



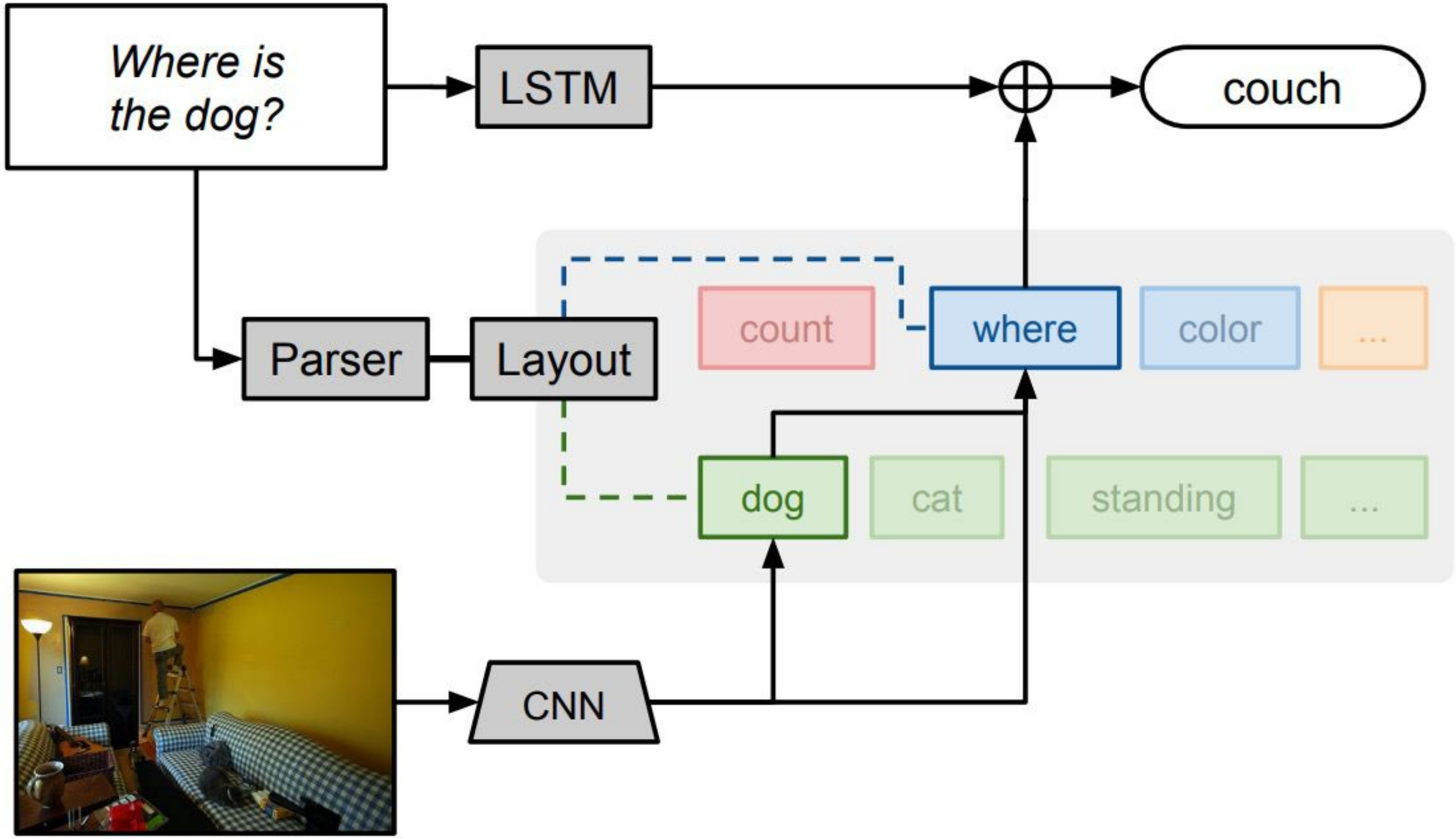
PyTorch

TensorFlow™




```
for i, data in enumerate(dataset):
  iter_start_time = time.time()
  if total_steps % opt.print_freq == 0:
      t_data = iter_start_time - iter_data_time
  visualizer.reset()
  total_steps += opt.batch_size
  epoch_iter += opt.batch_size
  model.set_input(data)
  model.optimize_parameters()
```

Differentiable programming



[Figure from "Neural Module Networks", Andreas et al. 2017]

Convolutional Neural Networks

Convolutional Neural Networks

LeCun et al. 1989

Neural network with specialized connectivity

Tailored to processing natural signals with a grid topology (e.g., images).

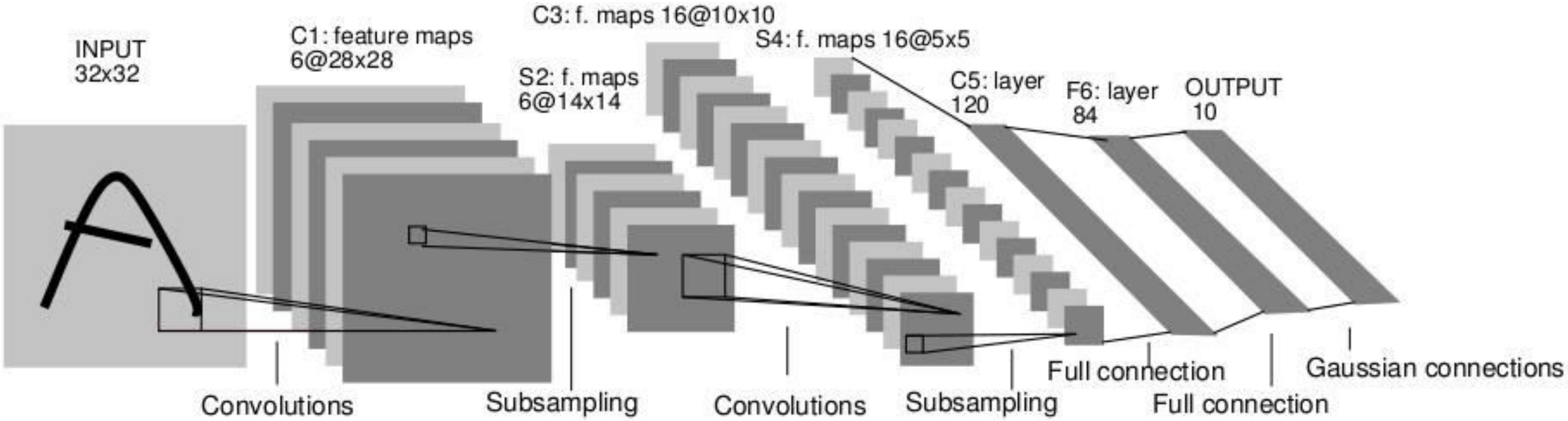


Image classification

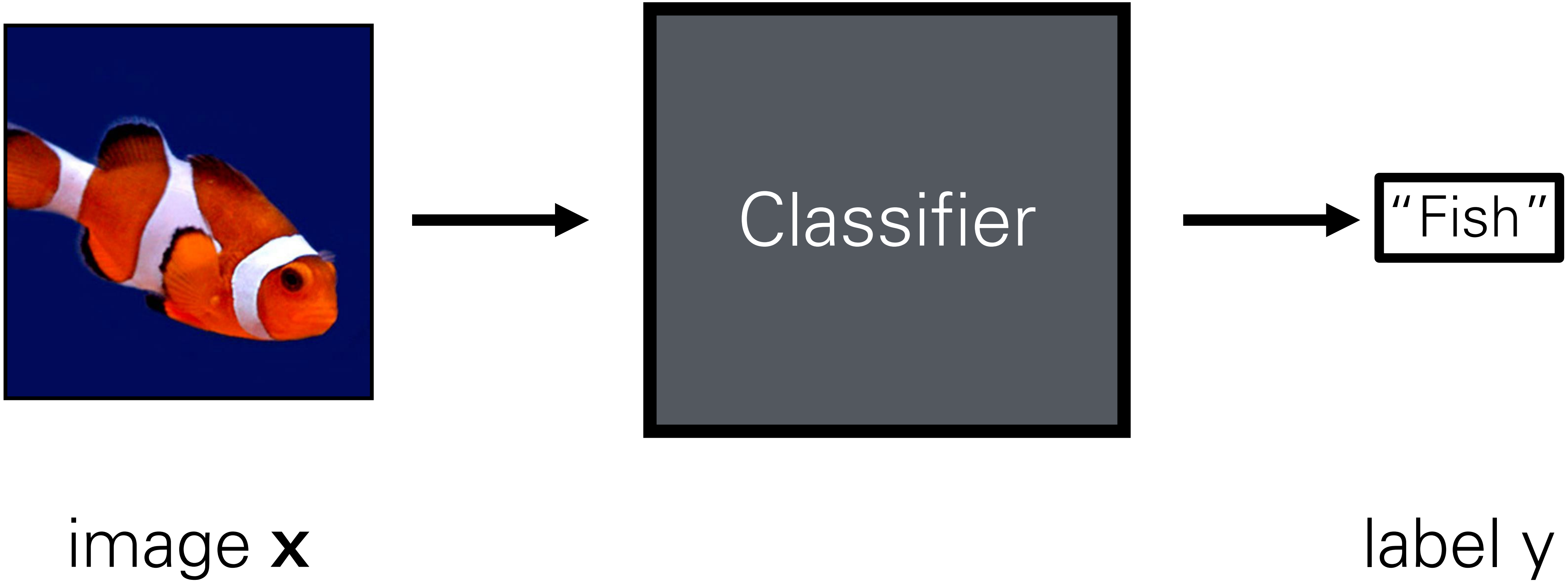
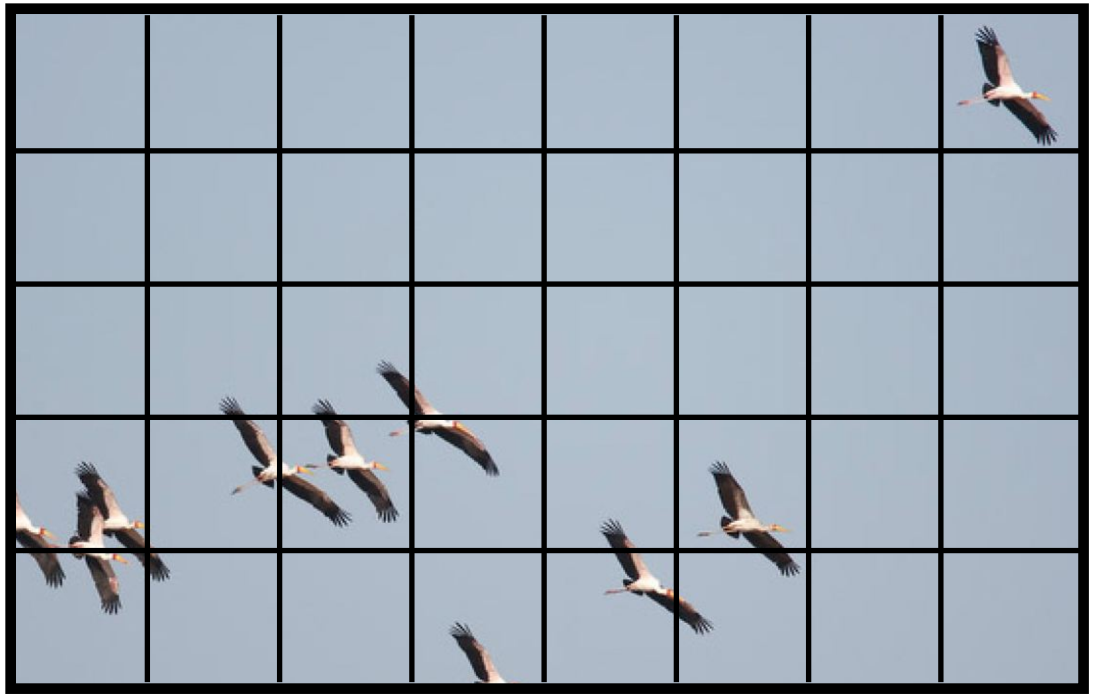
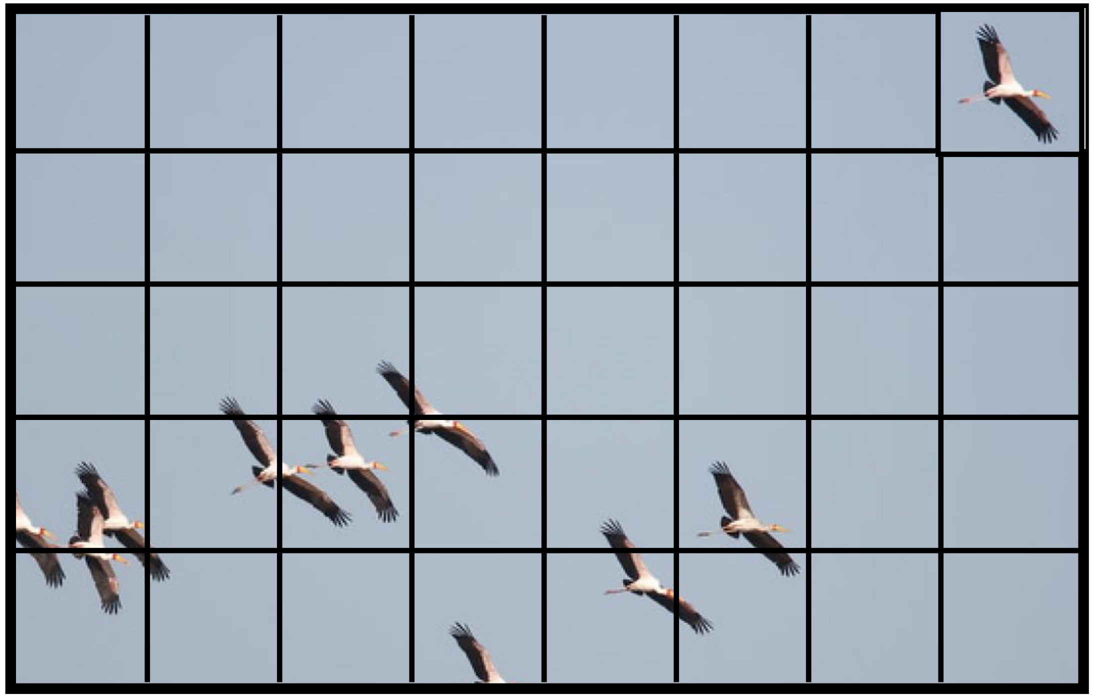
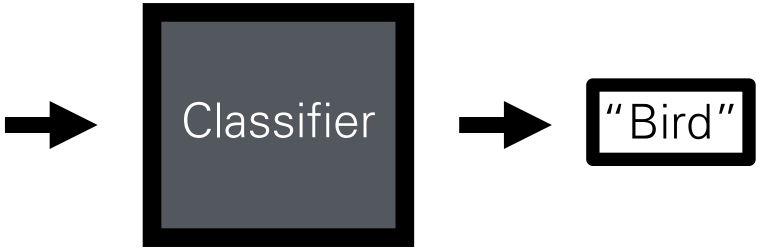
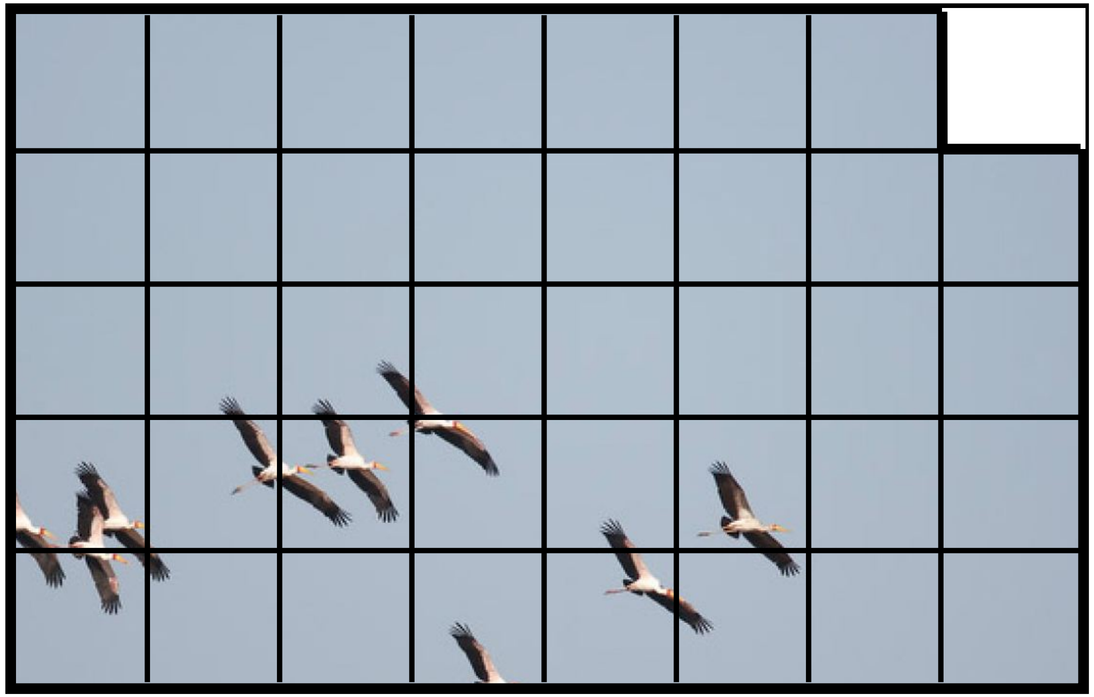


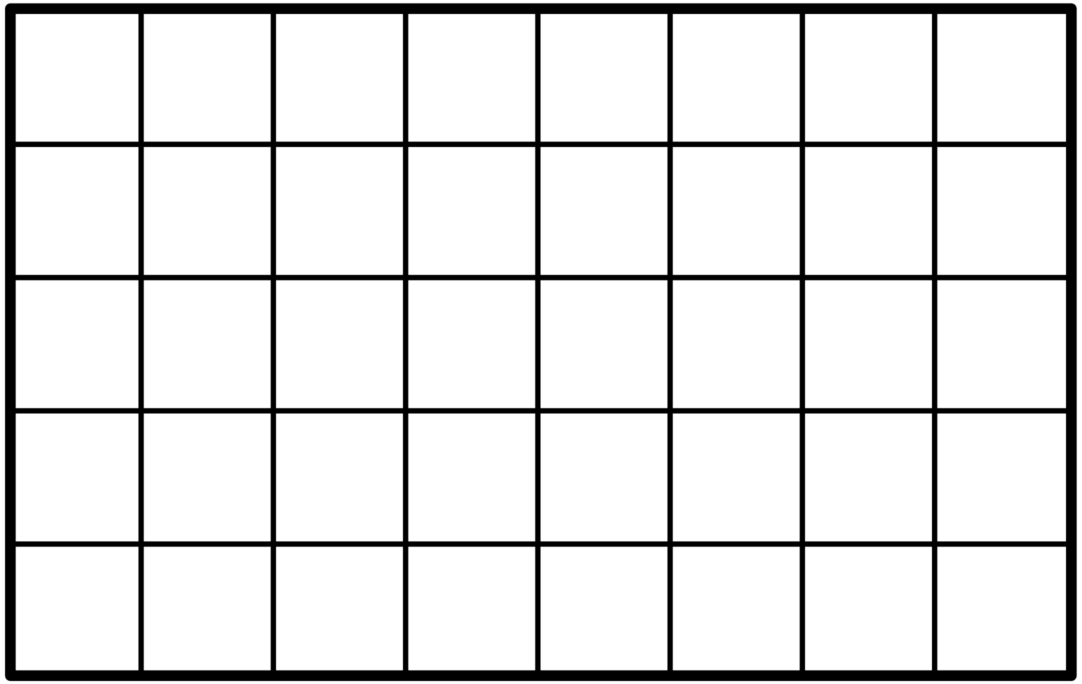
Photo credit: Fredo Durand



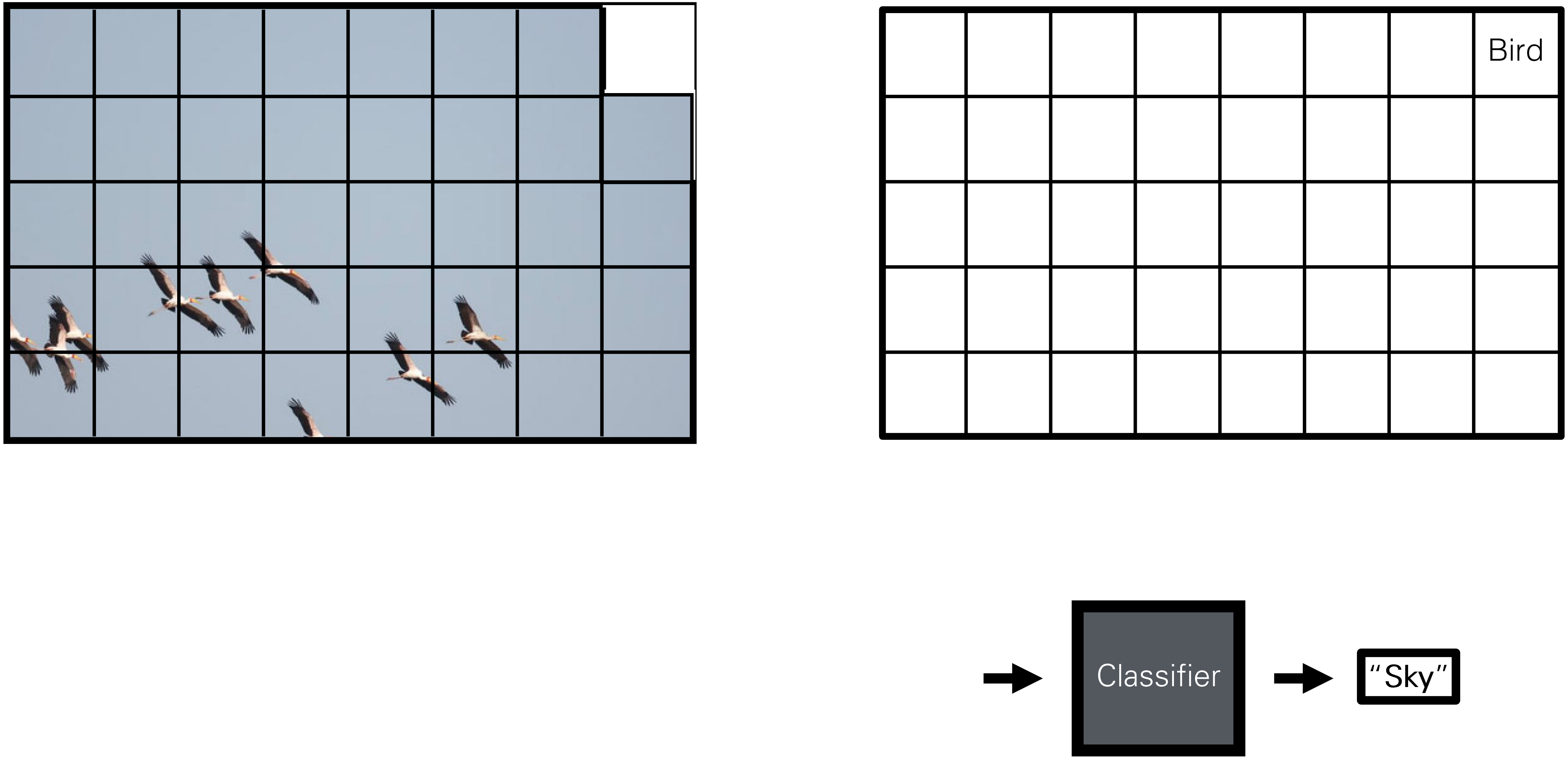


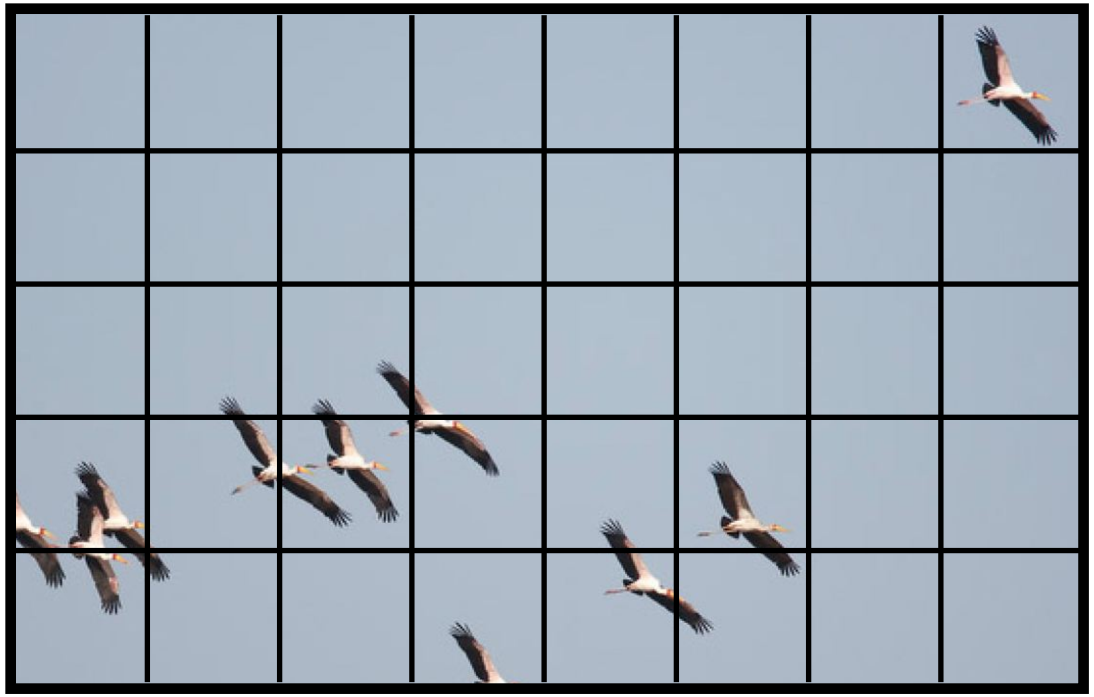

Classifier
"Bird"



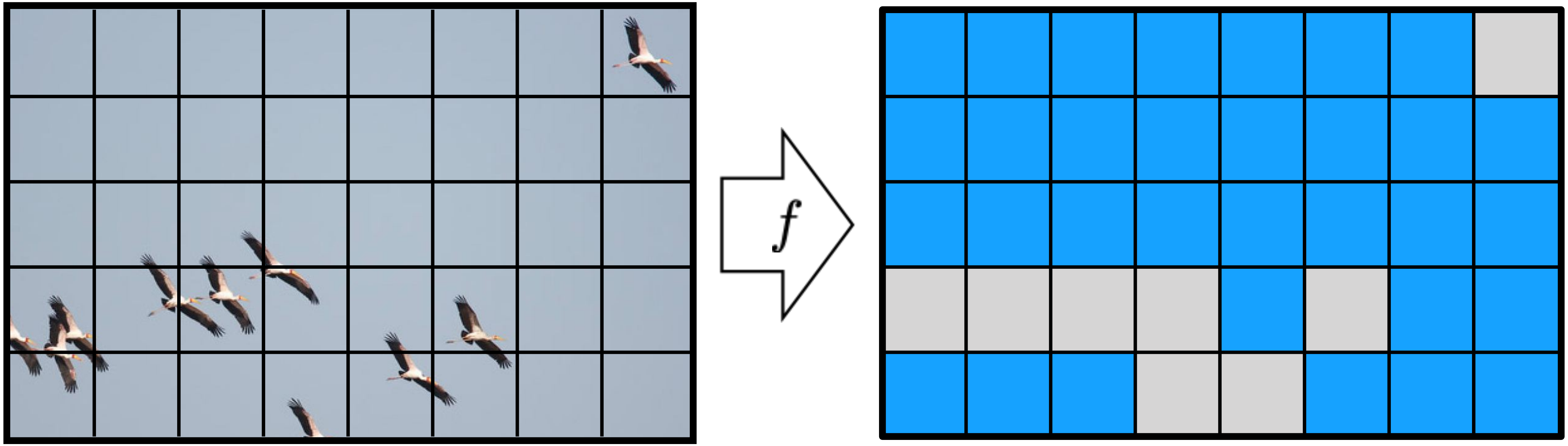


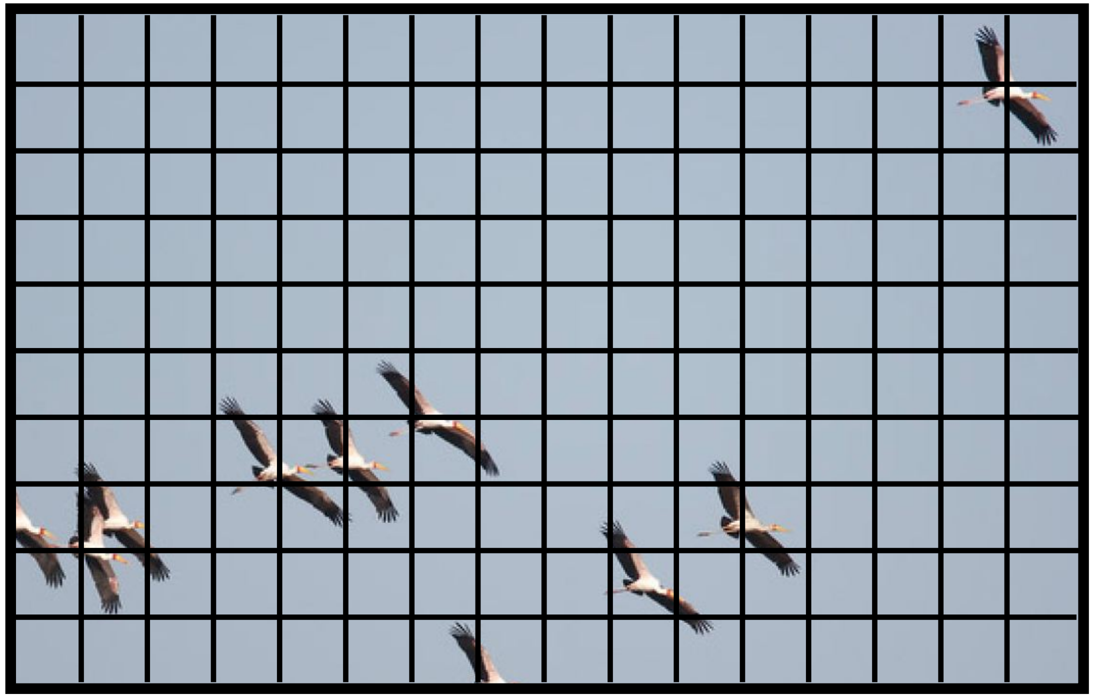

Classifier
“Bird”


Bird
Classifier
"Sky"

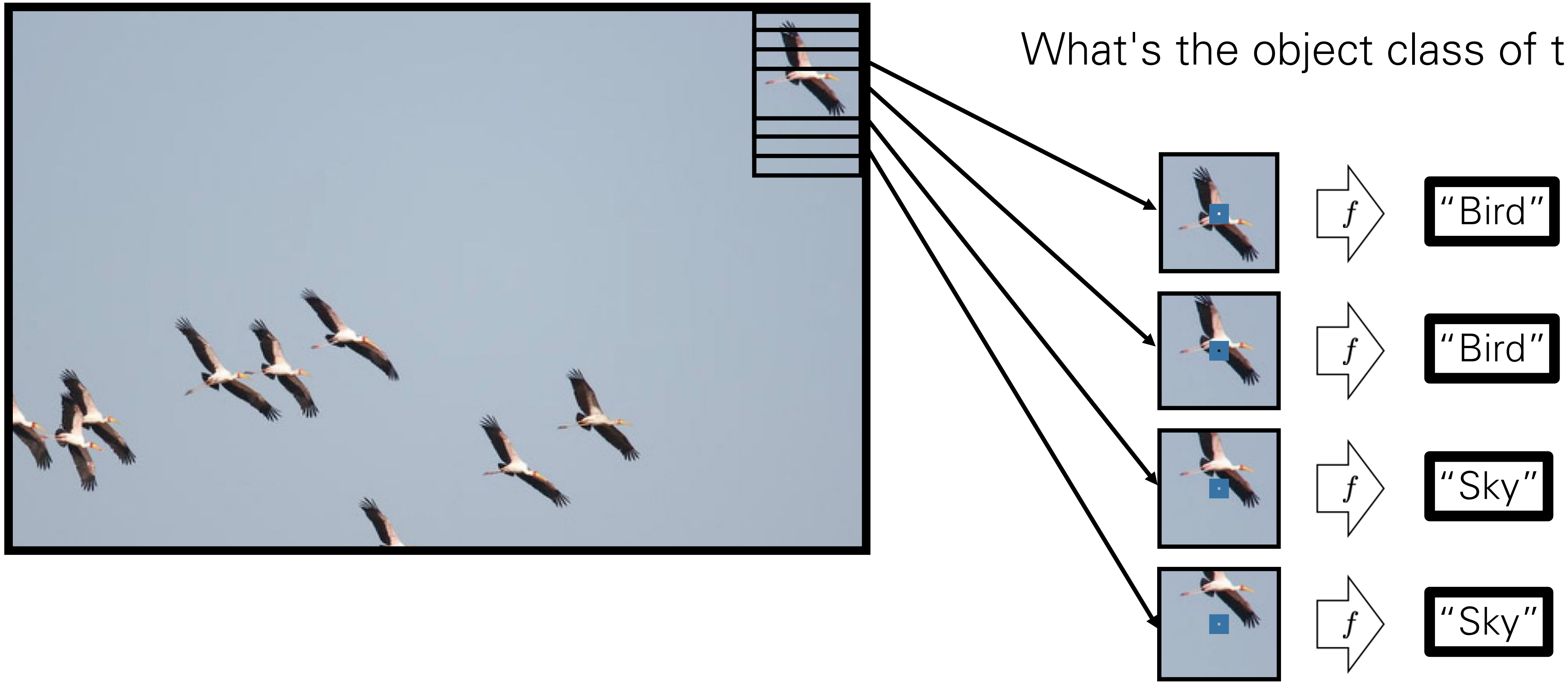


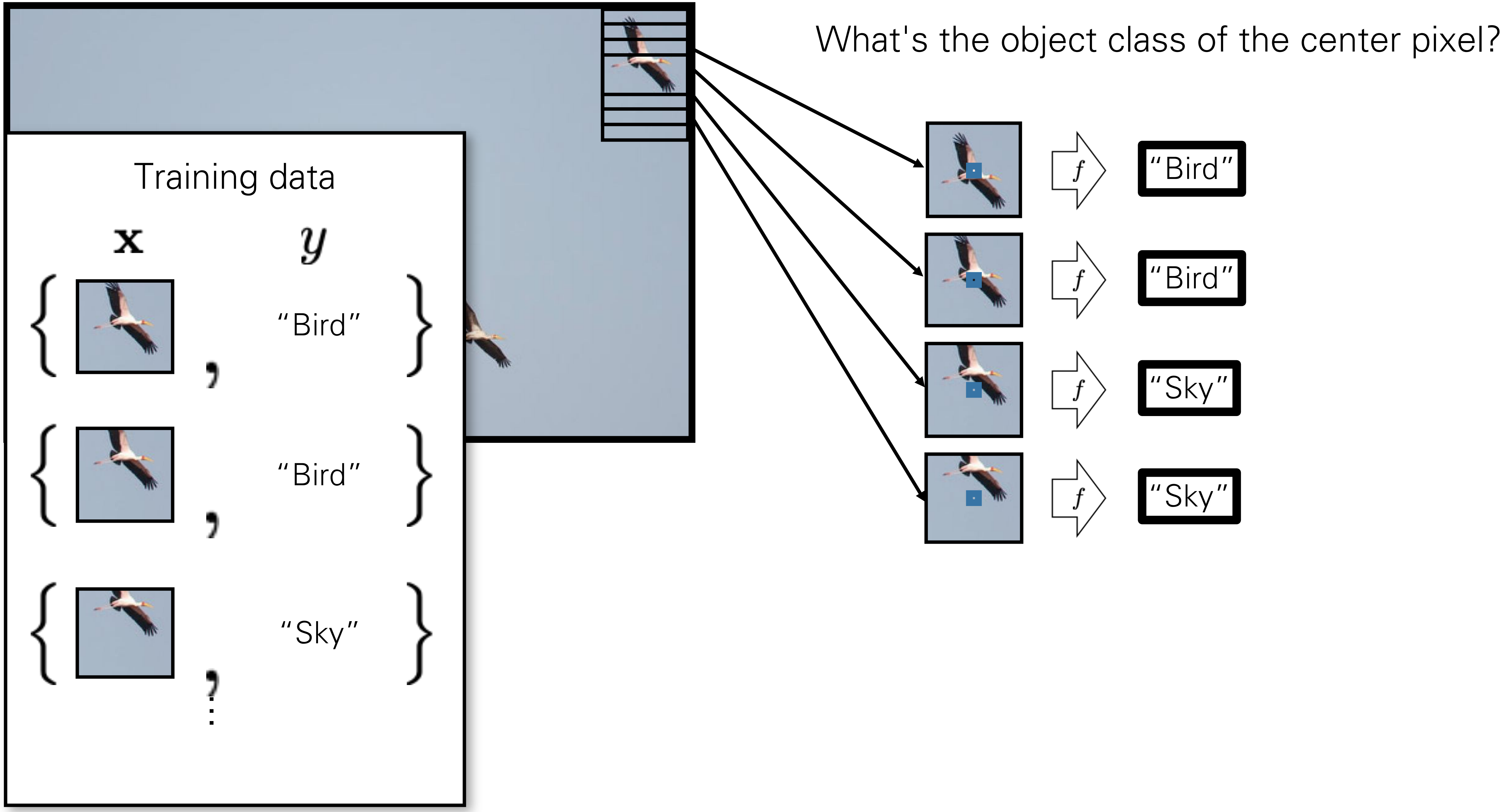
Sky	Sky	Sky	Sky	Sky	Sky	Sky	Bird
Sky	Sky	Sky	Sky	Sky	Sky	Sky	Sky
Sky	Sky	Sky	Sky	Sky	Sky	Sky	Sky
Bird	Bird	Bird	Sky	Bird	Sky	Sky	Sky
Sky	Sky	Sky	Bird	Sky	Sky	Sky	Sky

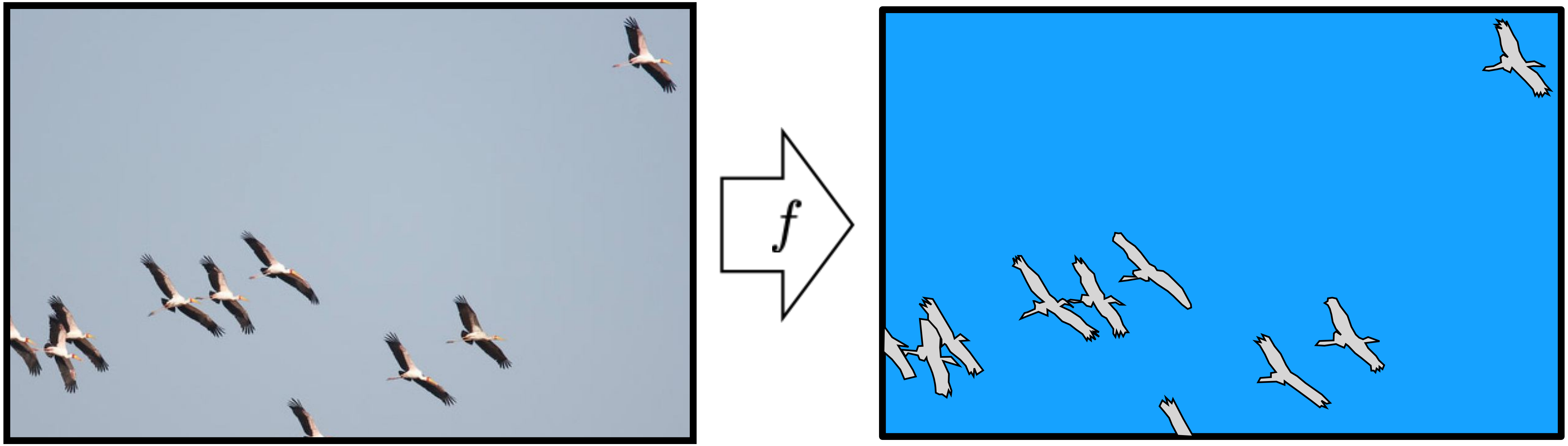

f



What's the object class of the center pixel?



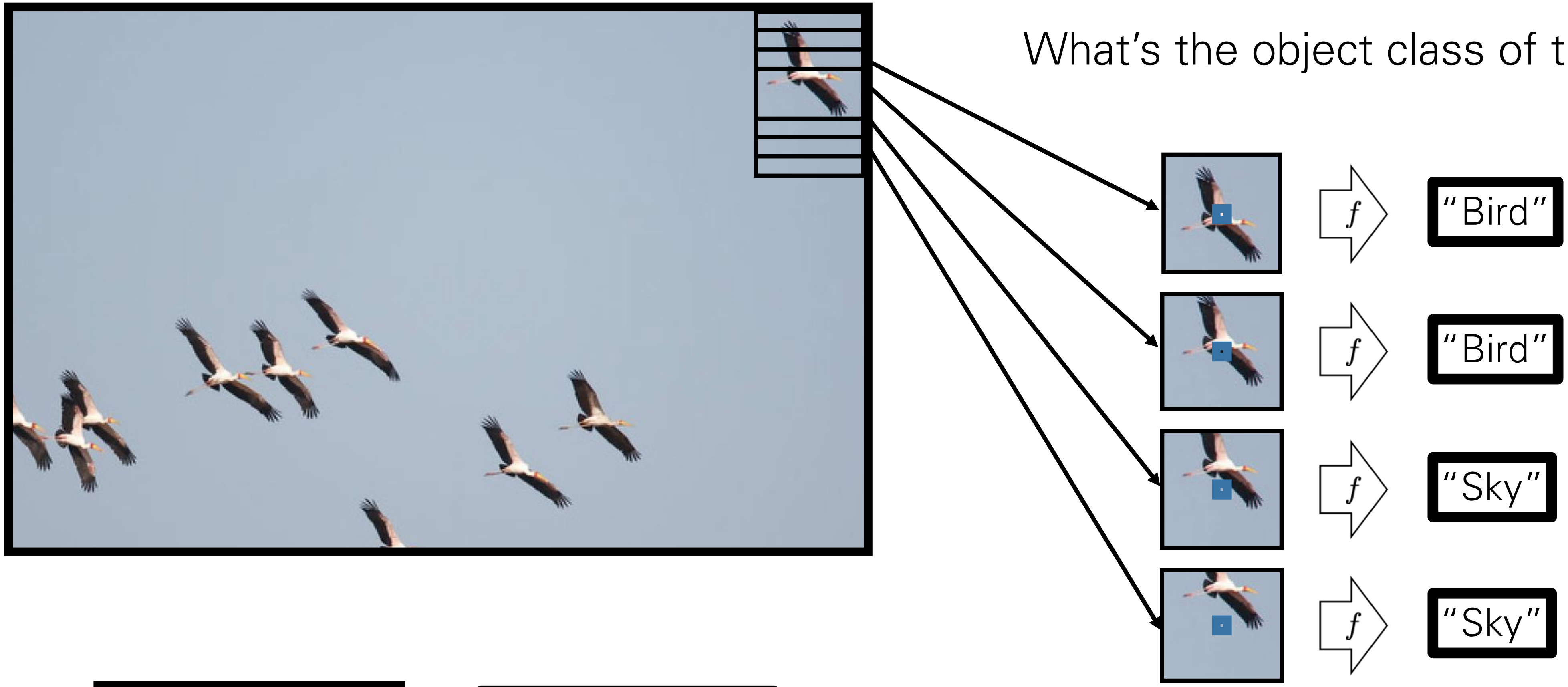

Training data
x
y
"Bird"
"Bird"
"Sky"
What's the object class of the center pixel?
f
"Bird"
"Bird"
"Sky"
"Sky"



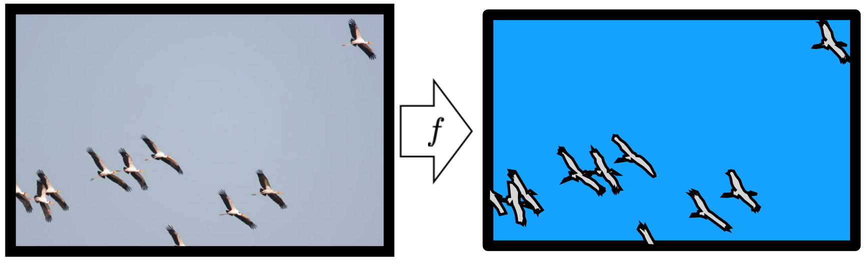
(Colors represent one-hot codes)

This problem is called **semantic segmentation**

What's the object class of the center pixel?



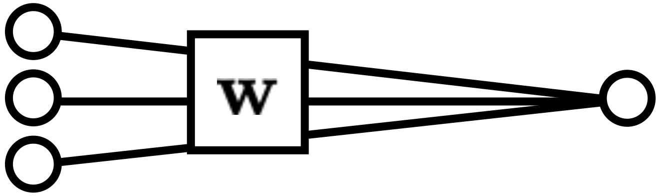
Translation invariance: process each patch in the same way.

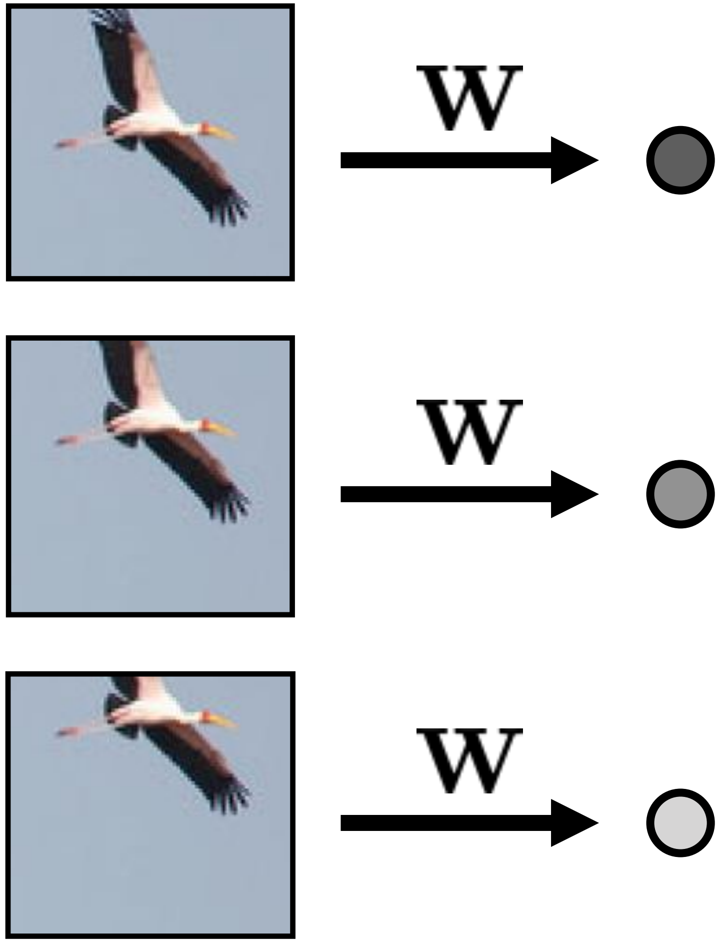


An equivariant mapping:

$$f(\texttt{translate}(x)) = \texttt{translate}(f(x))$$

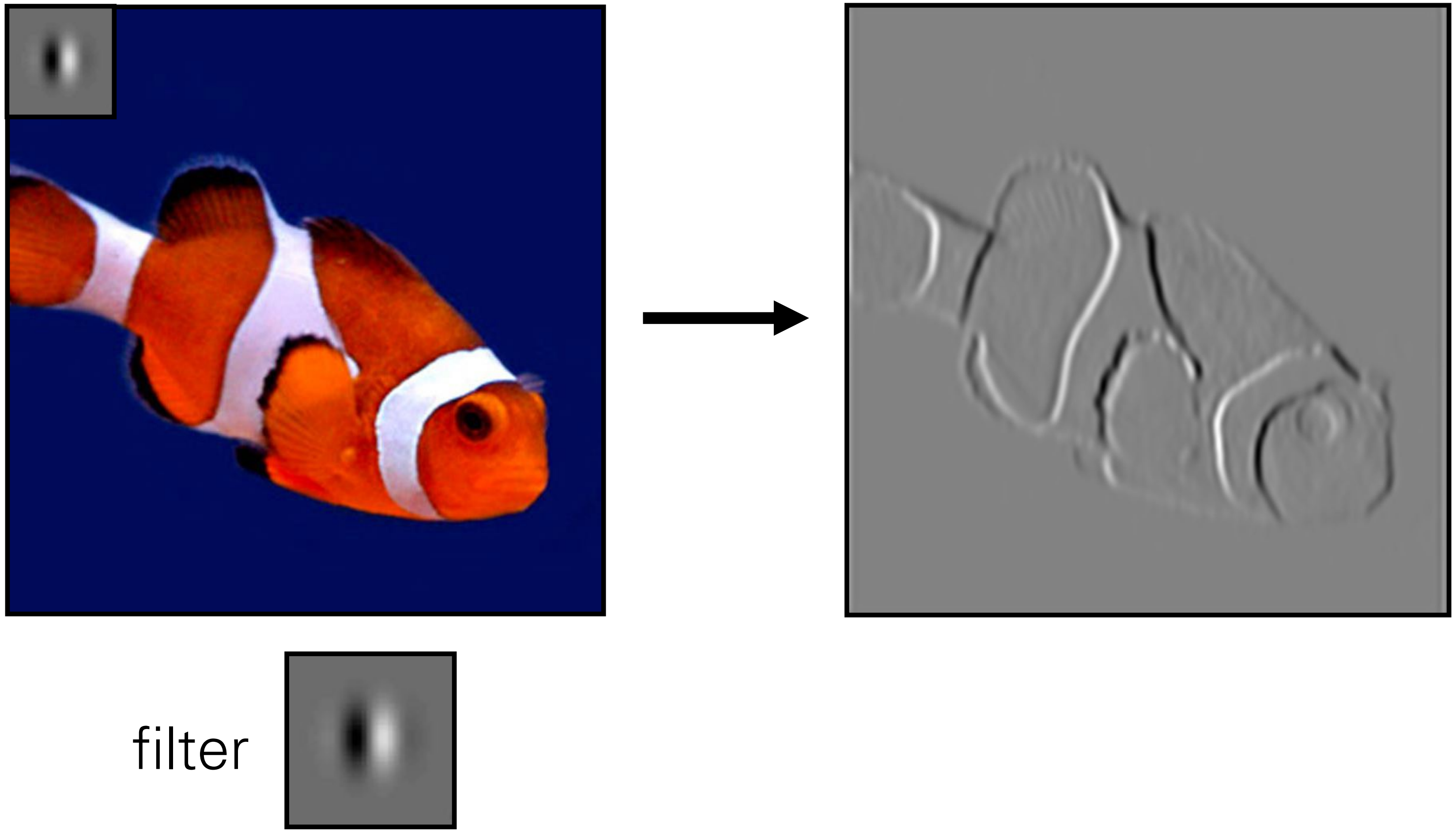
W computes a weighted sum of all pixels in the patch





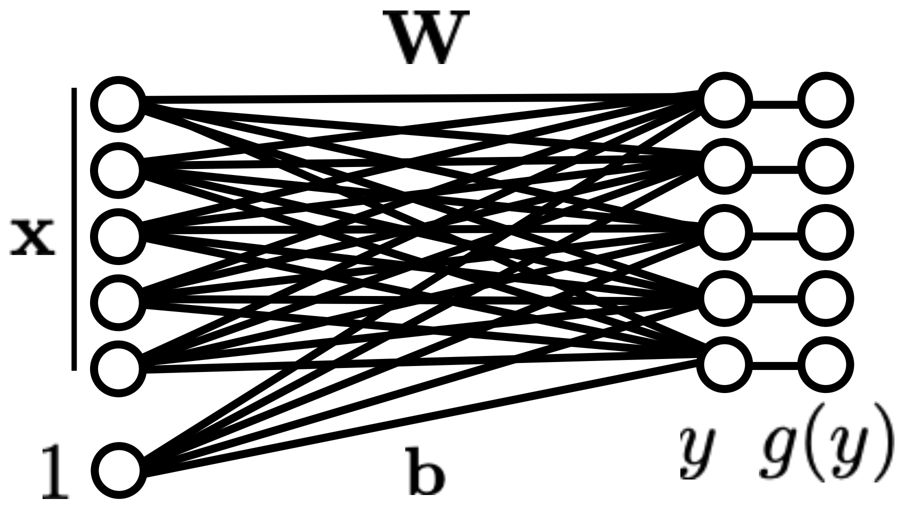
W is a convolutional kernel applied to the full image!

Convolution

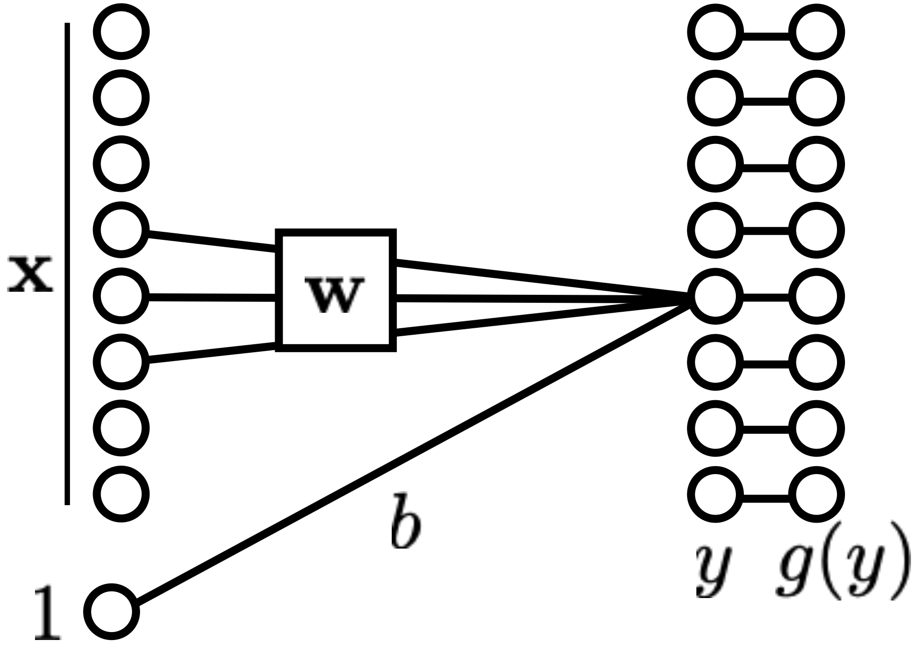


Fully-connected network

Fully-connected (fc) layer



Locally connected network

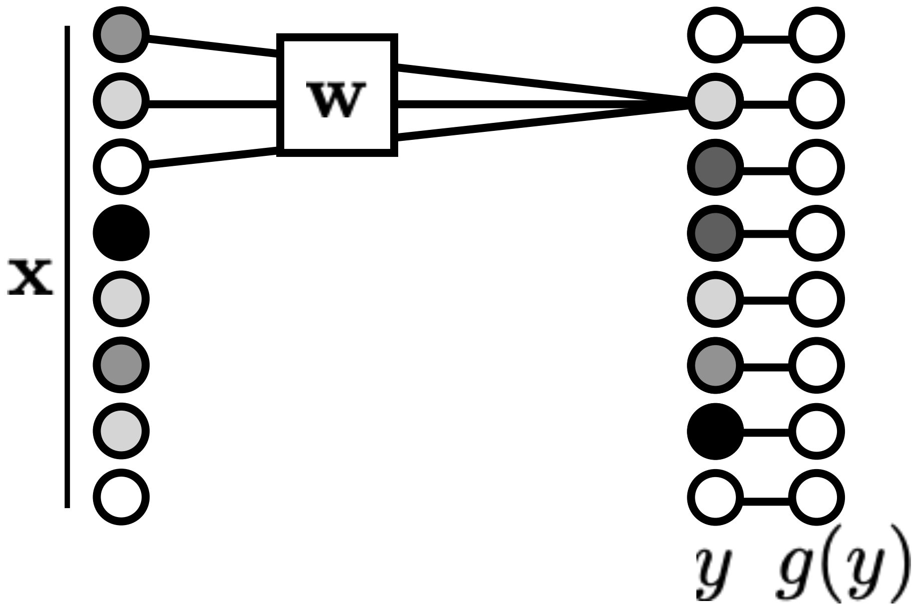


Often, we assume output is a **local** function of input.

If we use the same weights (**weight sharing**) to compute each local function, we get a convolutional neural network.

Convolutional neural network

Conv layer

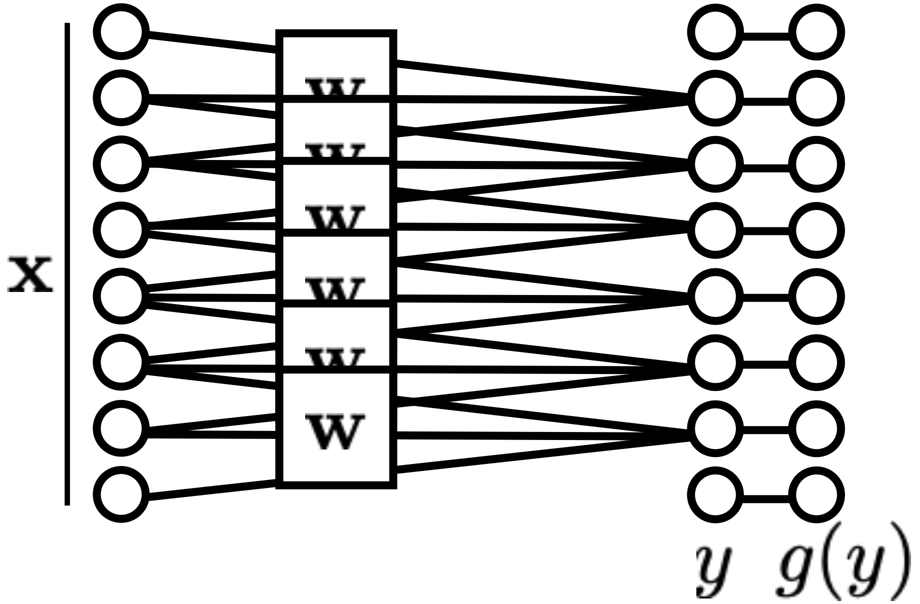


Often, we assume output is a **local** function of input.

If we use the same weights (**weight sharing**) to compute each local function, we get a convolutional neural network.

Weight sharing

Conv layer

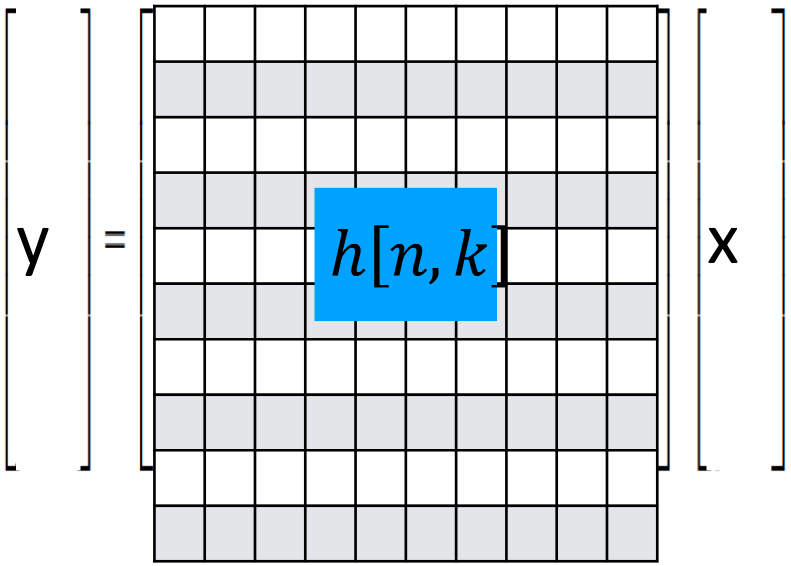


Often, we assume output is a **local** function of input.

If we use the same weights (**weight sharing**) to compute each local function, we get a convolutional neural network.

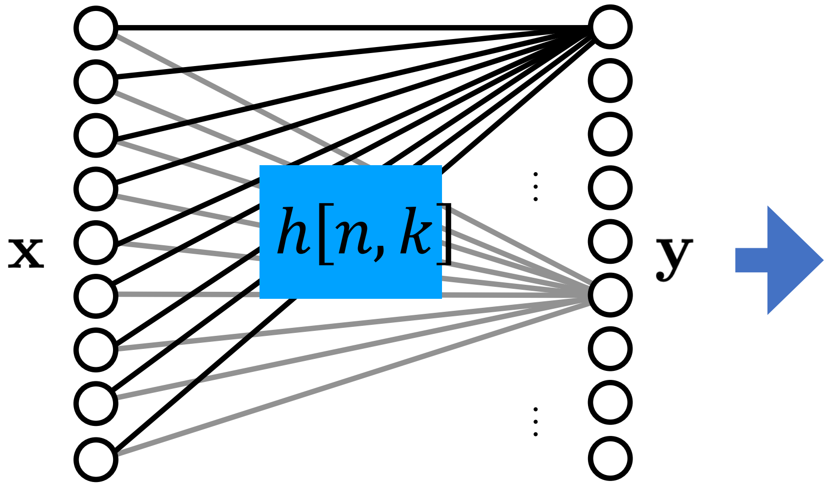
Linear system: $\mathbf{y} = f(\mathbf{x})$

A linear function f can be written as a matrix multiplication:



n indexes rows,
k indexes columns

It can also be represented as a fully connected linear neural network

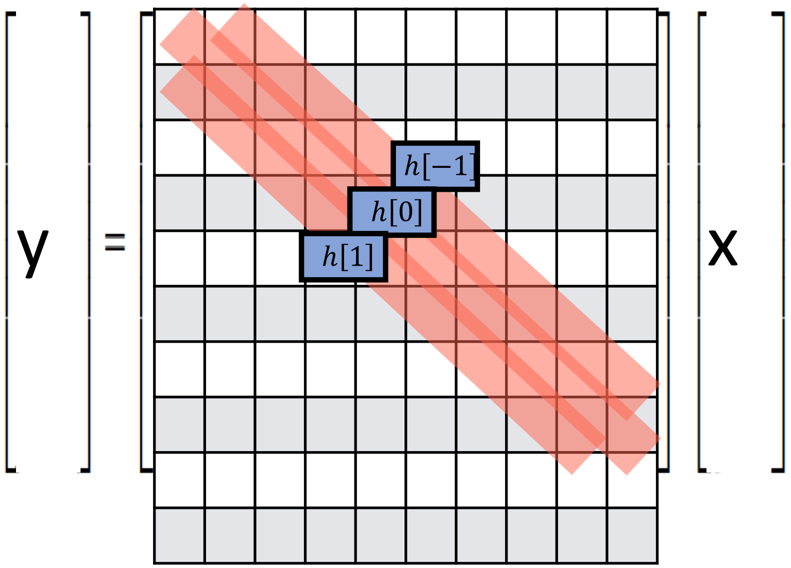


$$y[n] = \sum_{k=0}^{N-1} h[n,k]\,x[k]$$

$h[n,k]$ Is the strength of the connection between x[k] and y[n]

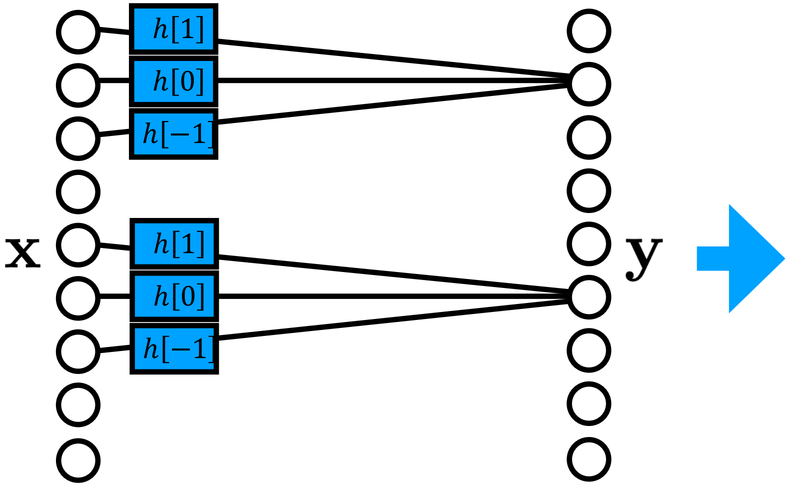
Convolution

A linear shift invariant (LSI) function f can be written as a matrix multiplication:



$h[n-k]$ n indexes rows, k indexes columns

It can also be represented as a convolutional layer of neural net:

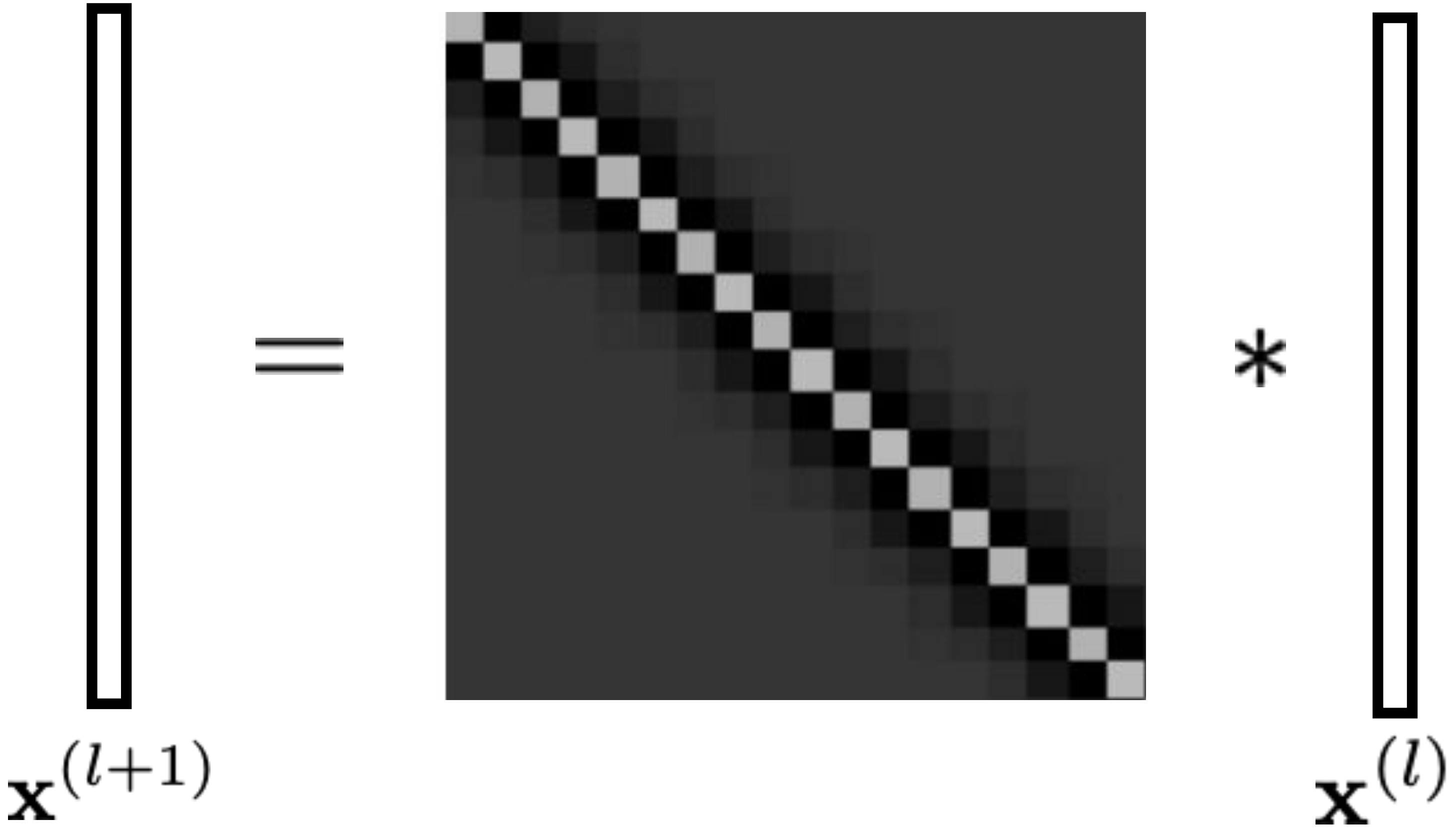


$$y[n] = \sum_{k=-1}^{1} h[k]x[n-k]$$

$h[n-k]$ Is the strength of the connection between x[k] and y[n]

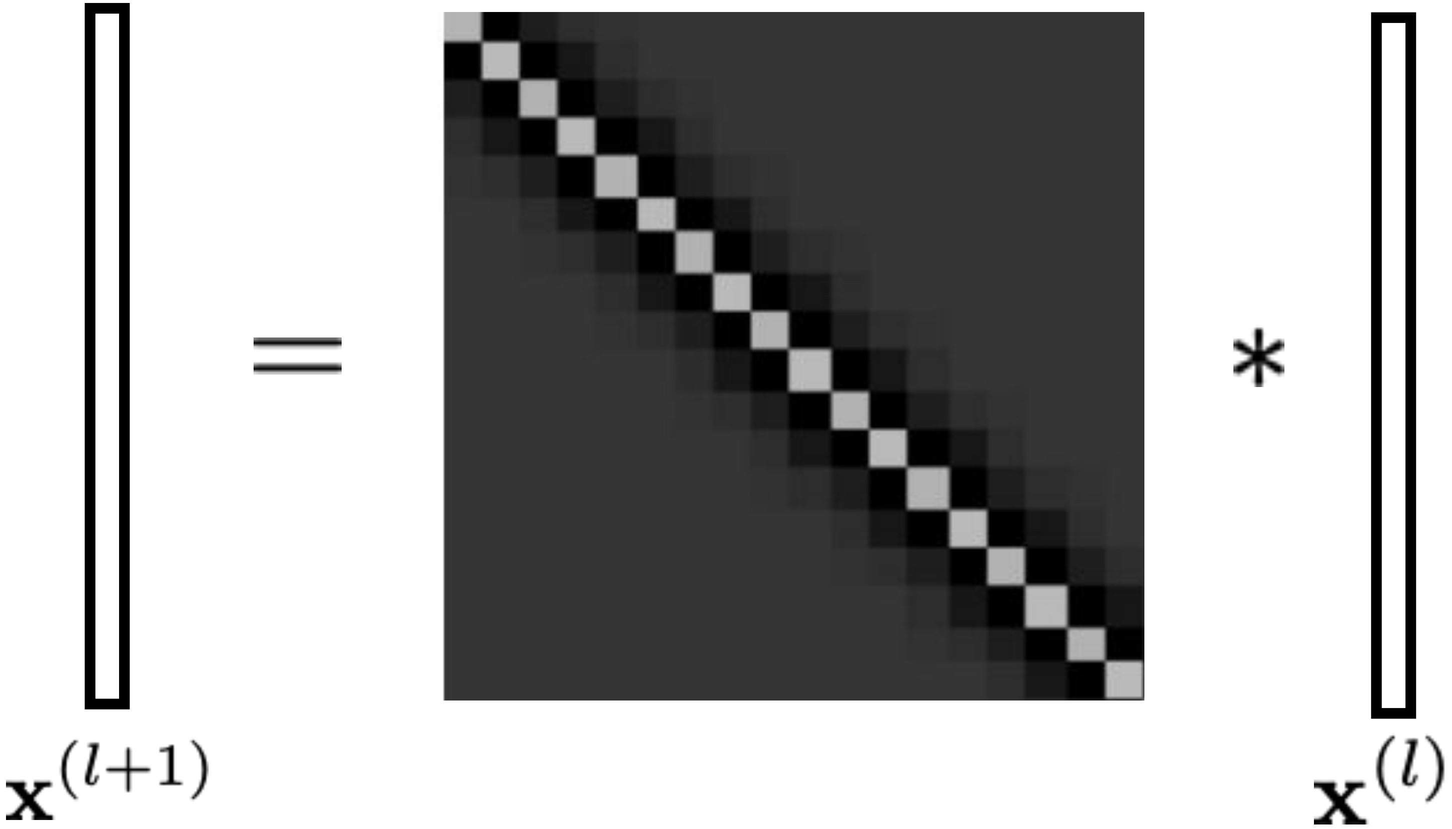
Toeplitz matrix

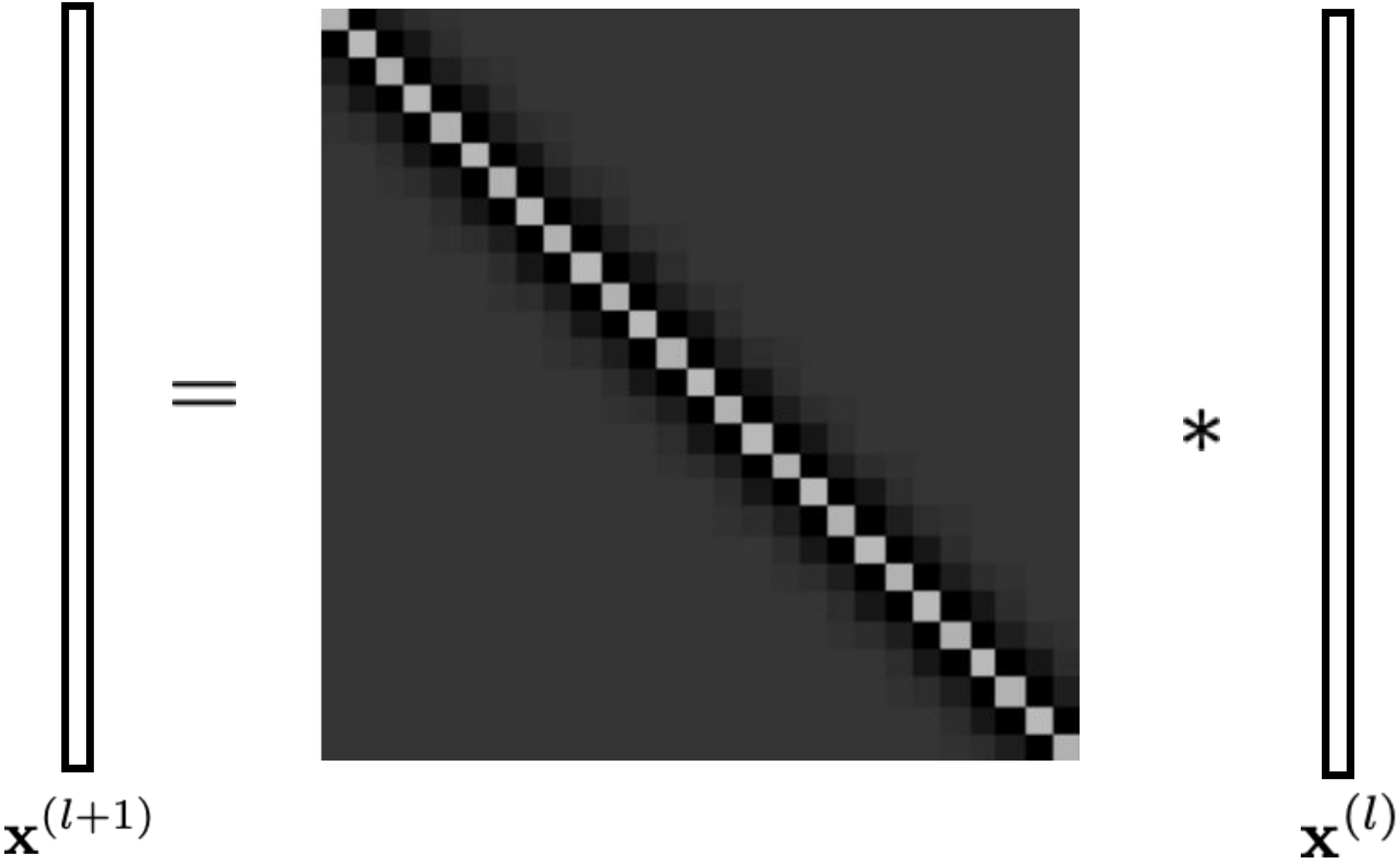
$$\begin{pmatrix} a & b & c & d & e \\ f & a & b & c & d \\ g & f & a & b & c \\ h & g & f & a & b \\ i & h & g & f & a \end{pmatrix}$$



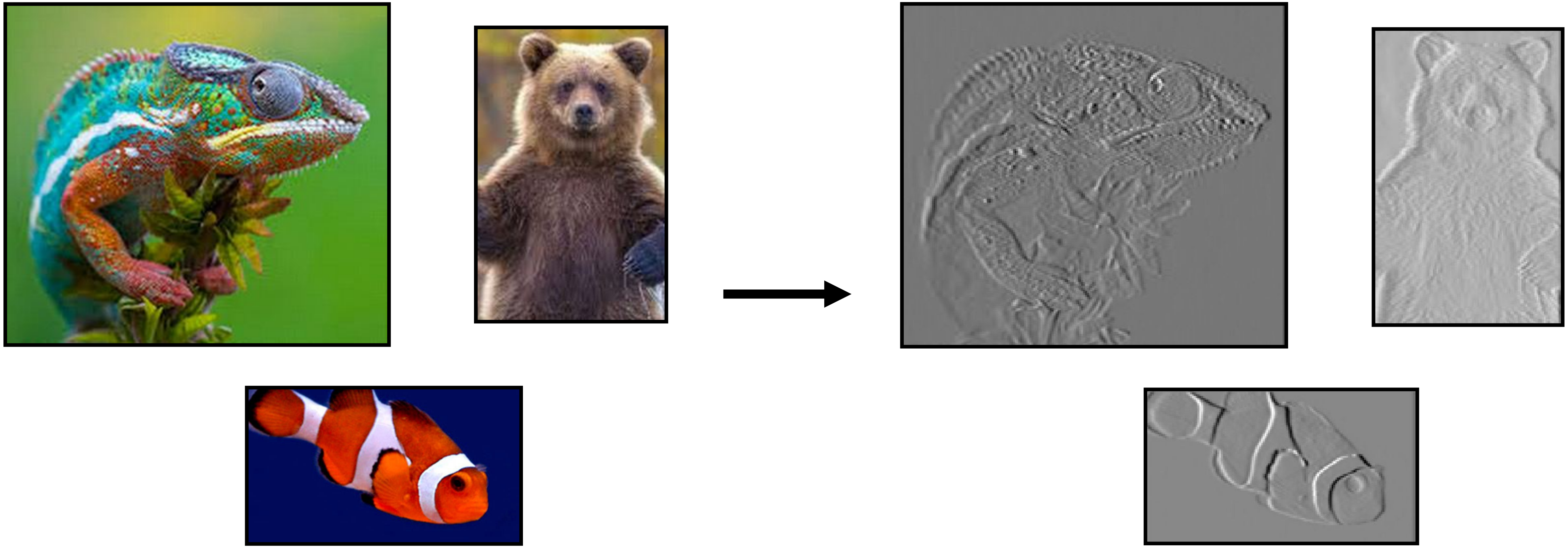
e.g., pixel image

- Constrained linear layer (infinitely strong regularization)
- Fewer parameters —> easier to learn, less overfitting


$\mathbf{x}^{(l+1)}$
=
*
$\mathbf{x}^{(l)}$



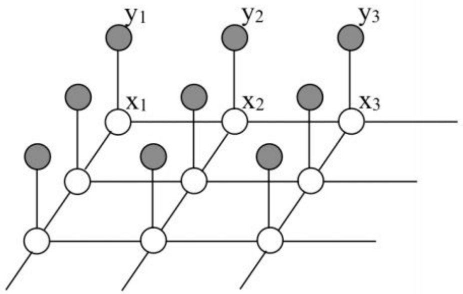
$\mathbf{x}^{(l+1)}$
=
*
$\mathbf{x}^{(l)}$



Conv layers can be applied to arbitrarily-sized inputs

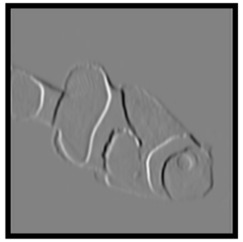
Five views on convolutional layers

1. Equivariant with translation (stationarity) $f(\texttt{translate}(x)) = \texttt{translate}(f(x))$

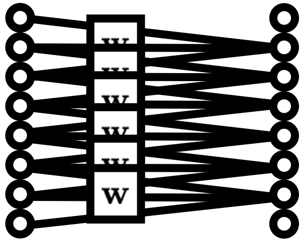


2. Patch processing (Markov assumption)

3. Image filter



4. Parameter sharing

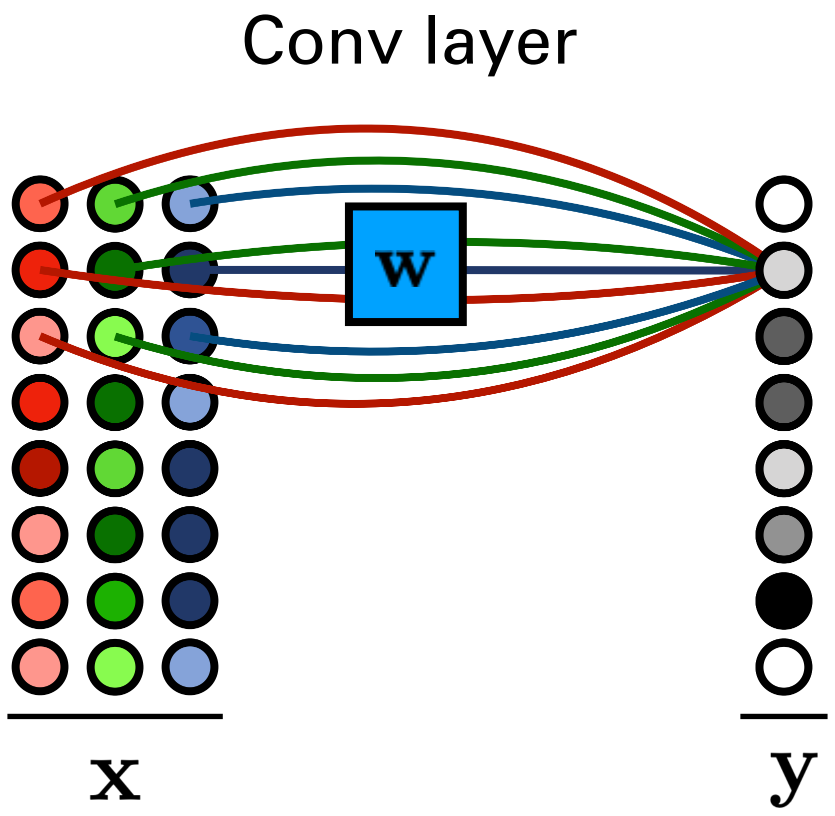


5. A way to process variable-sized tensors

What if we have color?

(aka multiple input channels?)

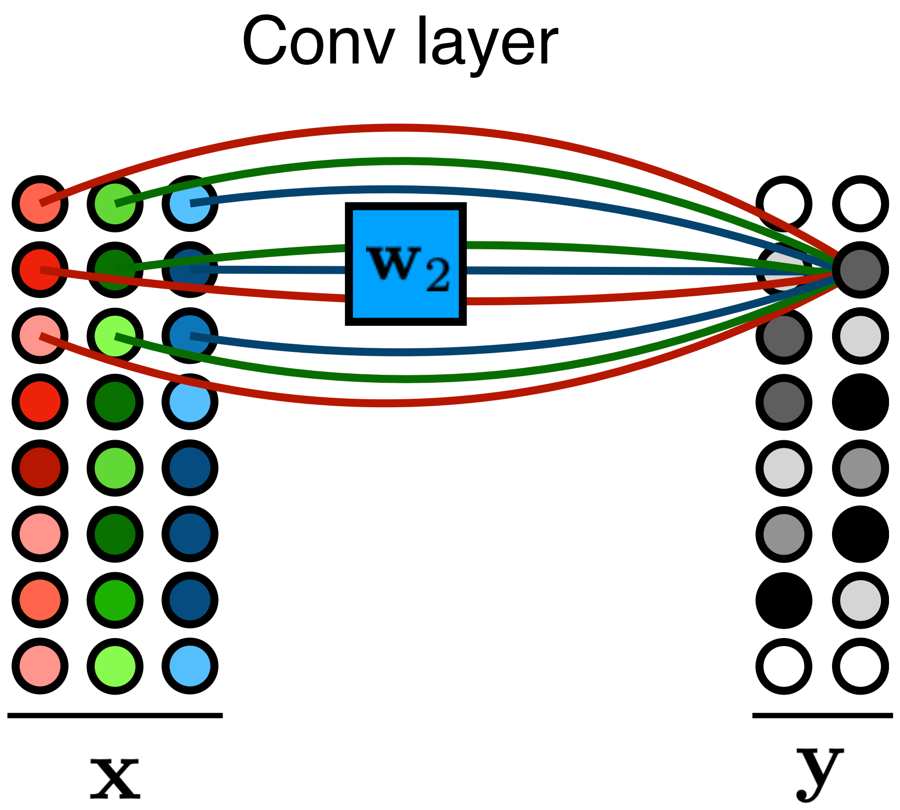
Multiple channels



$$\mathbf{y} = \sum_c \mathbf{w}_c \circ \mathbf{x}_c$$

$$\mathbb{R}^{N \times C} \rightarrow \mathbb{R}^{N \times 1}$$

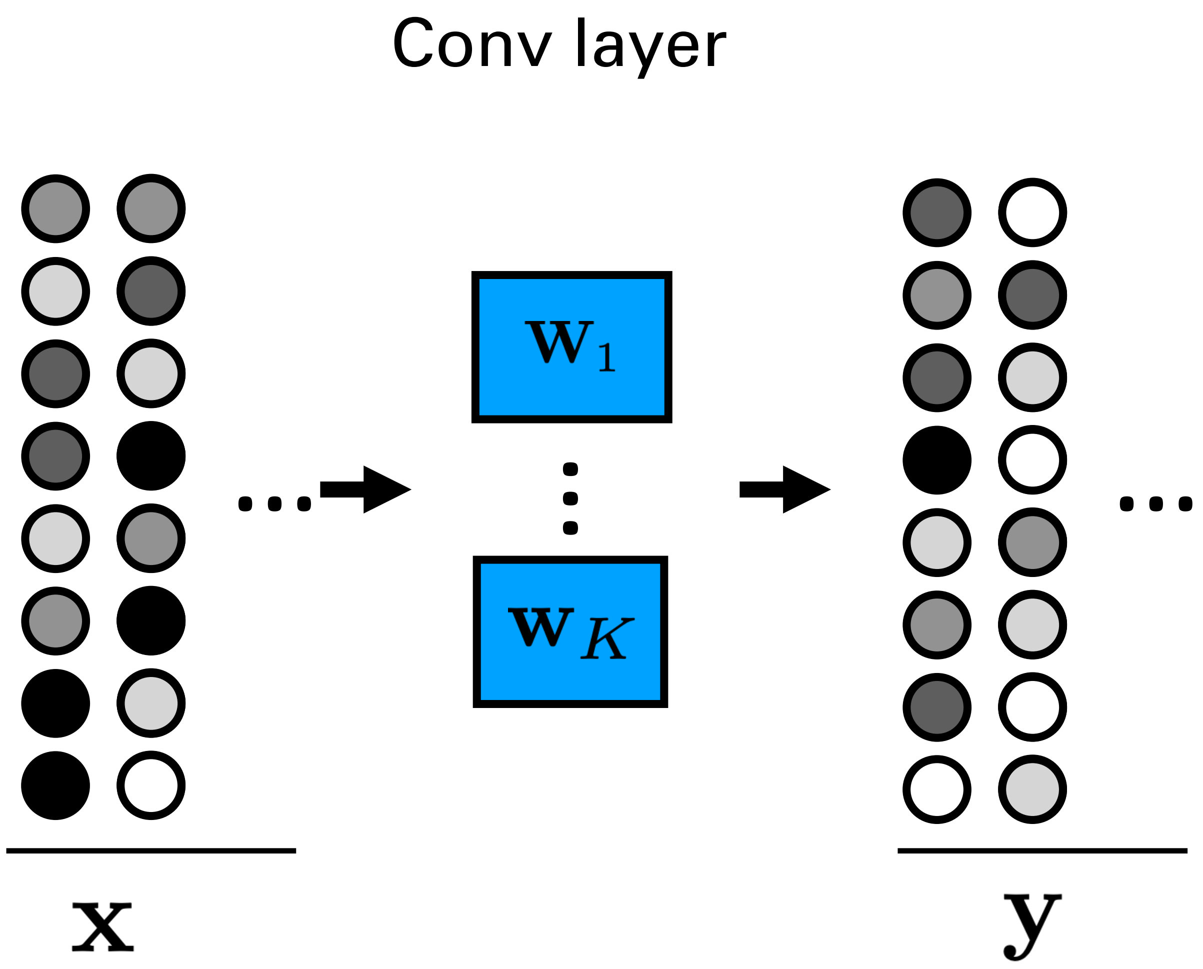
Multiple channels



$$\mathbf{y}_k = \sum_c \mathbf{w}_{k_c} \circ \mathbf{x}_c$$

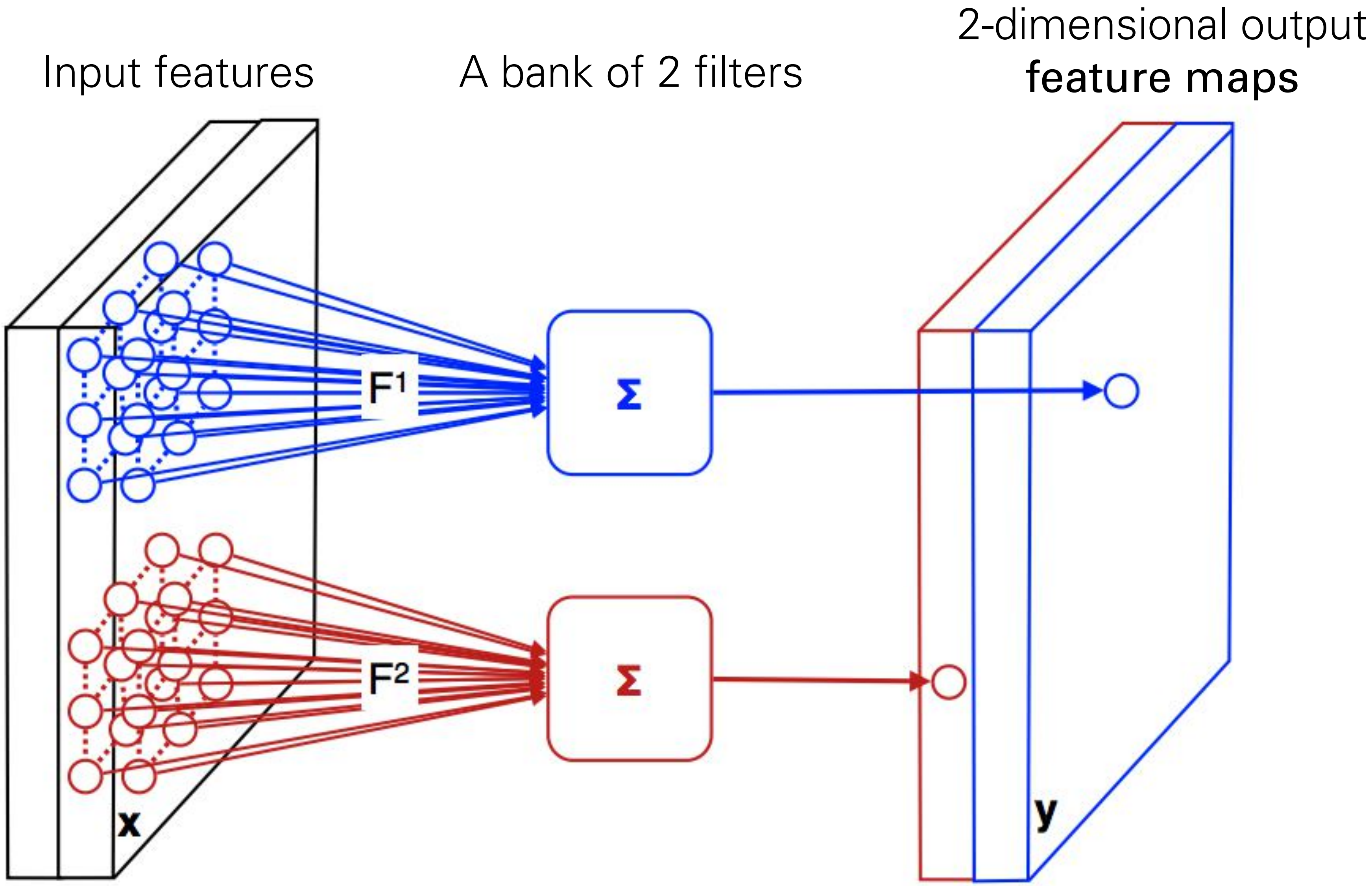
$$\mathbb{R}^{N \times C} \rightarrow \mathbb{R}^{N \times K}$$

Multiple channels



$$\mathbf{y}_k = \sum_c \mathbf{w}_{k_c} \circ \mathbf{x}_c$$

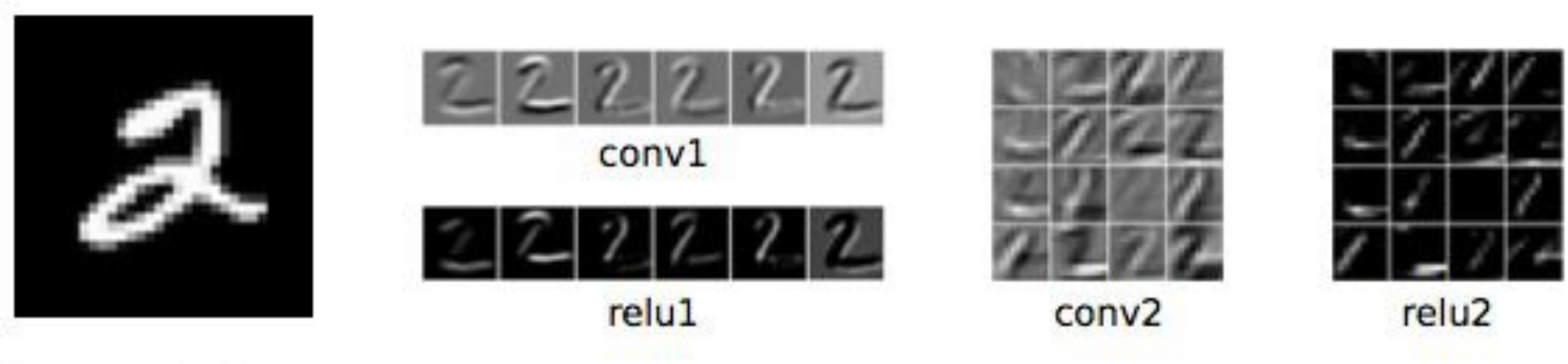
$$\mathbb{R}^{N \times C} \rightarrow \mathbb{R}^{N \times K}$$



$$\mathbb{R}^{H \times W \times C^{(l)}} \rightarrow \mathbb{R}^{H \times W \times C^{(l+1)}}$$

[Figure modified from Andrea Vedaldi]

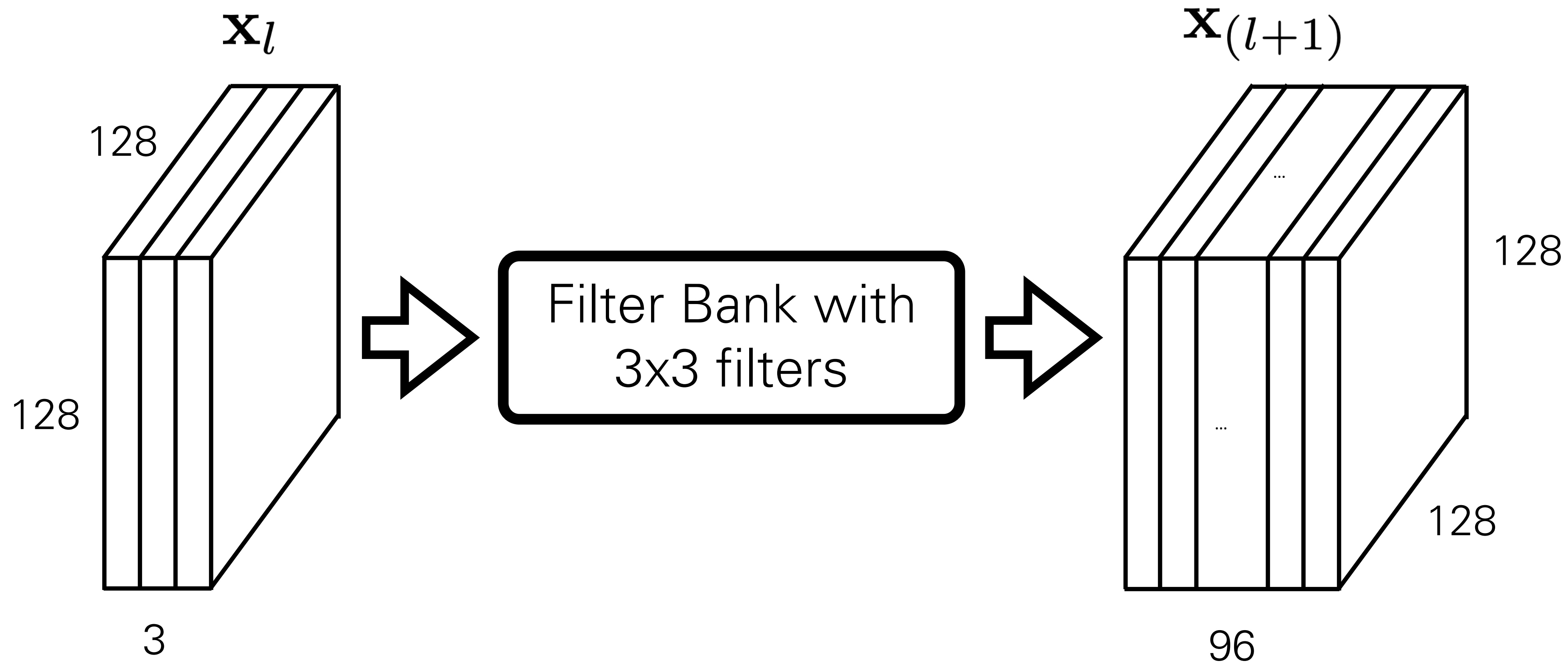
Feature maps



- Each layer can be thought of as a set of C **feature maps** aka **channels**
- Each feature map is an NxM image

Image source: https://stackoverflow.com/questions/45678473/convolution-neural-networks-all-feature-maps-are-blackpixel-value-is-0

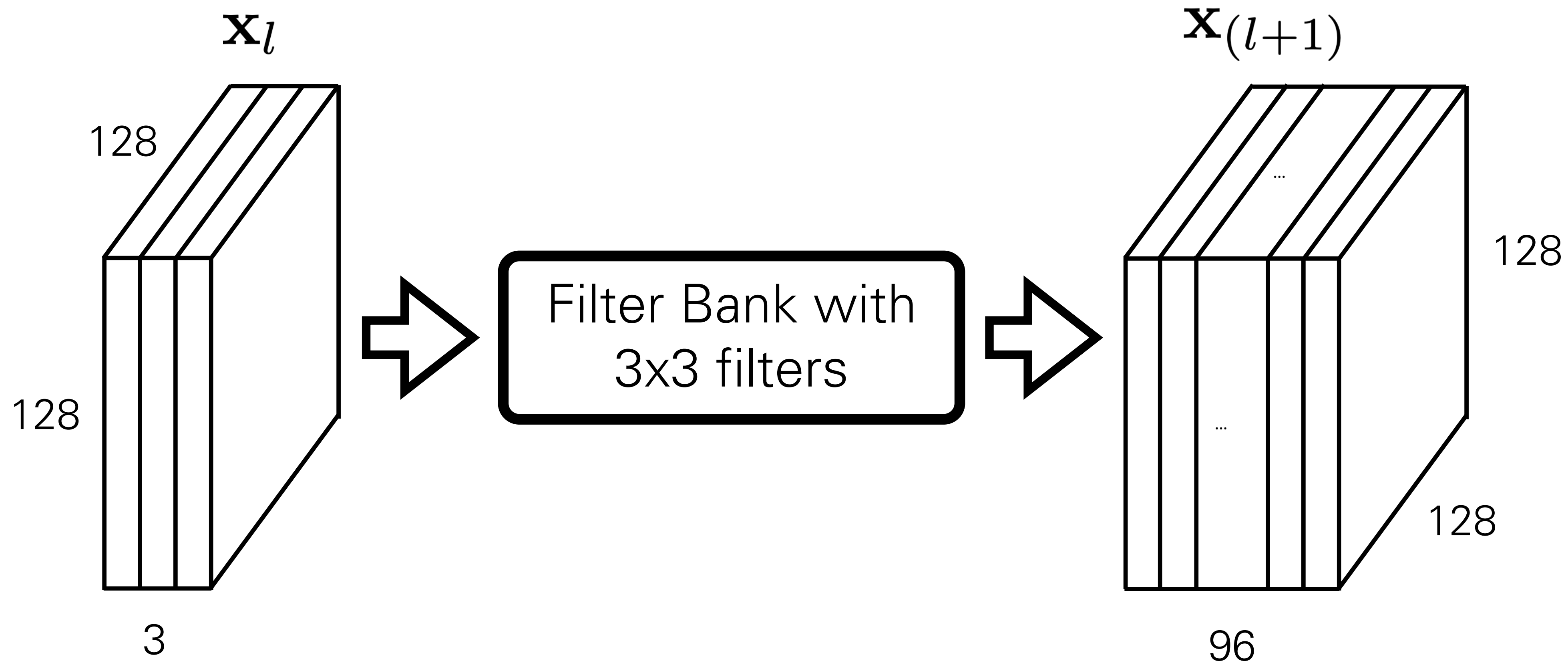
Multiple channels: Example



How many parameters does each filter have?

(a) 9 (b) 27 (c) 96 (d) 864

Multiple channels: Example



How many filters are in the bank?

(a) 3 (b) 27 (c) 96 (d) can't say

Filter sizes

When mapping from

$$\mathbf{x}_l \in \mathbb{R}^{H \times W \times C_l} \quad \rightarrow \quad \mathbf{x}_{(l+1)} \in \mathbb{R}^{H \times W \times C_{(l+1)}}$$

using an filter of spatial extent $M \times N$

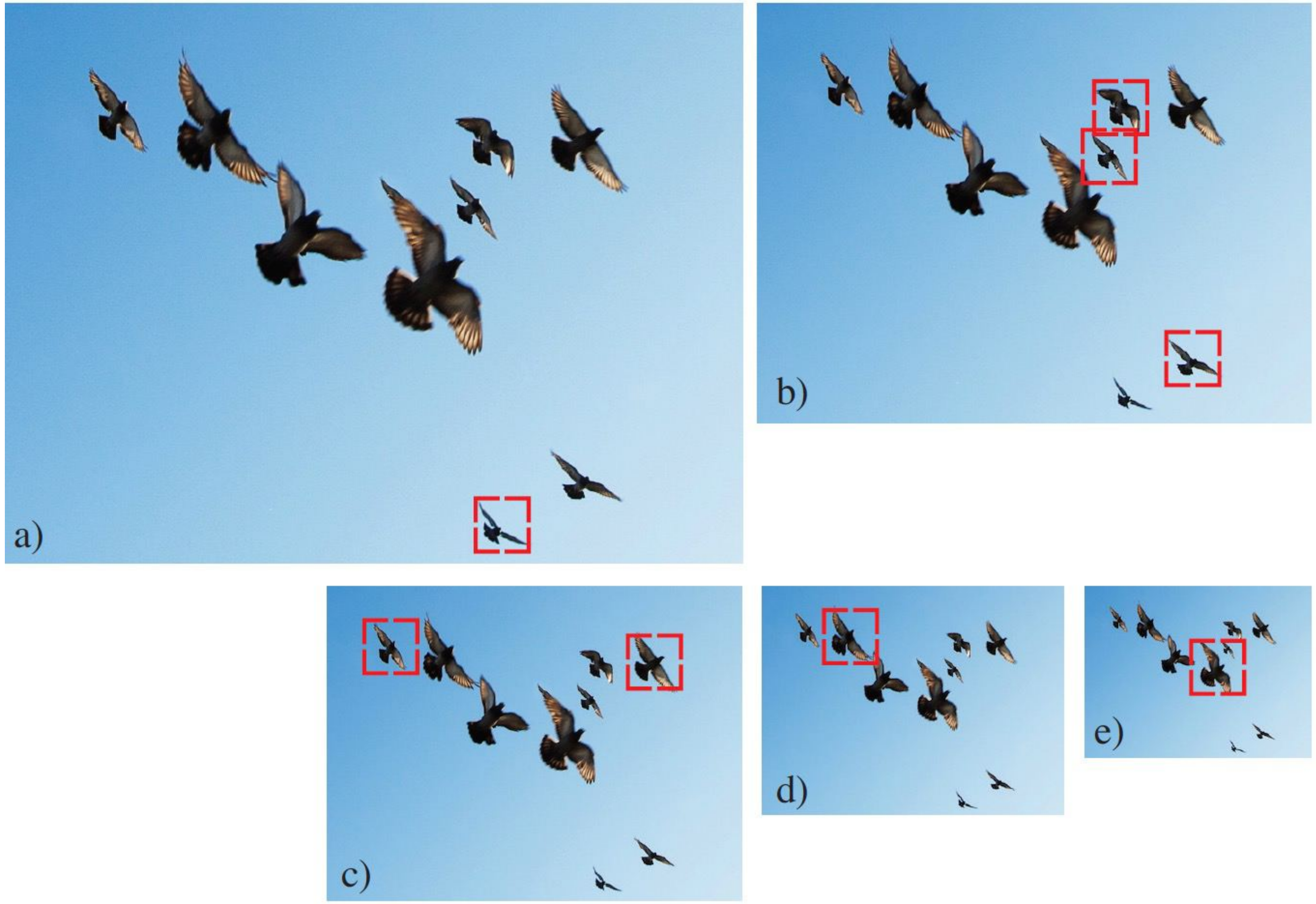
Number of parameters per filter: $M \times N \times C_l$

Number of filters: $C_{(l+1)}$

Pooling and downsampling

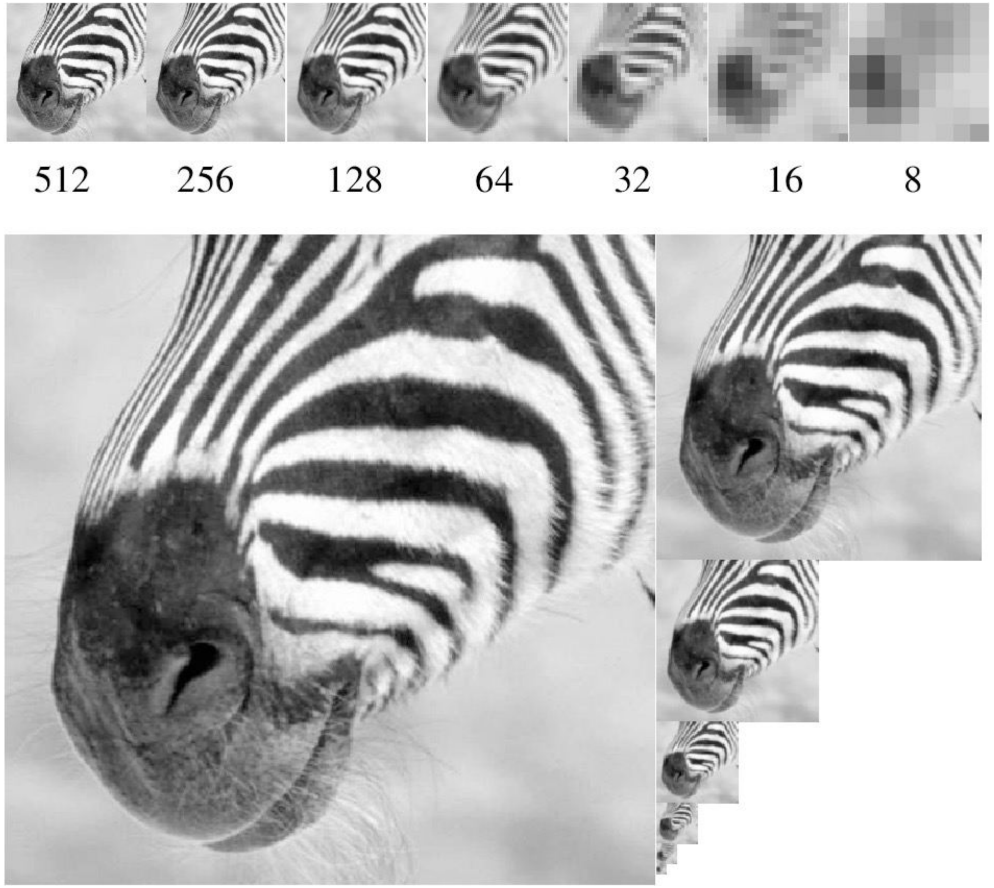
We need translation and **scale** invariance

Image pyramids

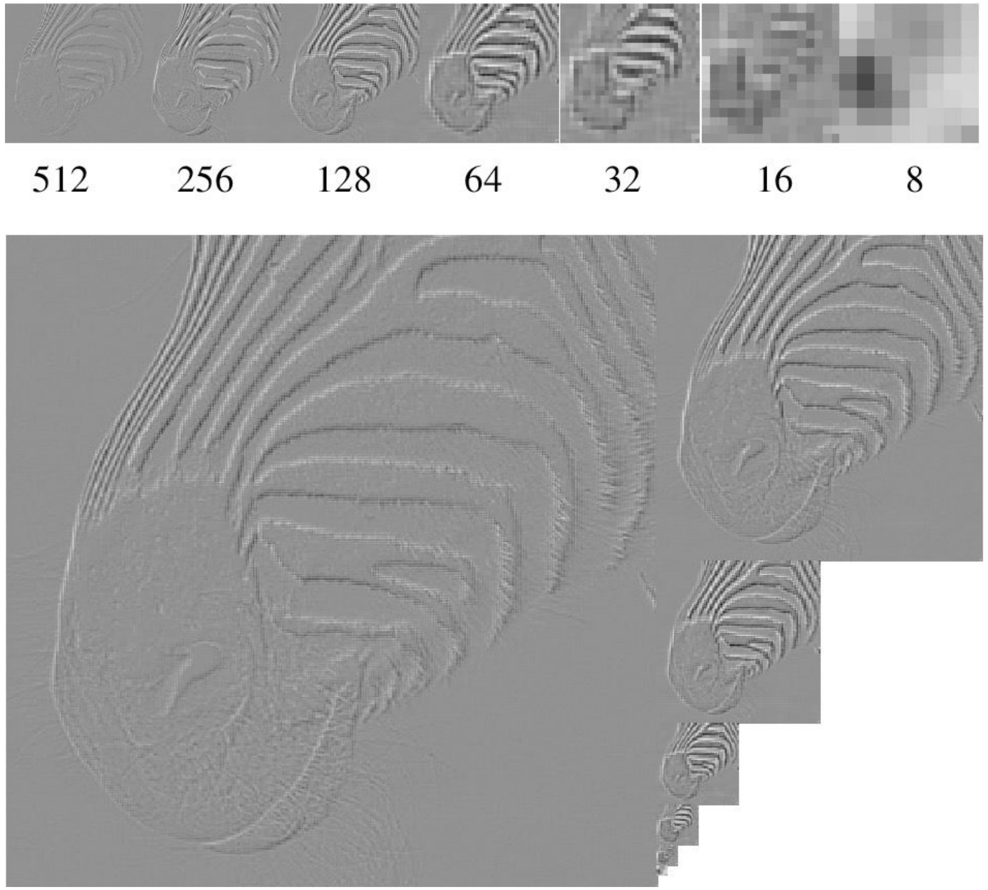


Gaussian Pyramid

Multiscale representations are great!



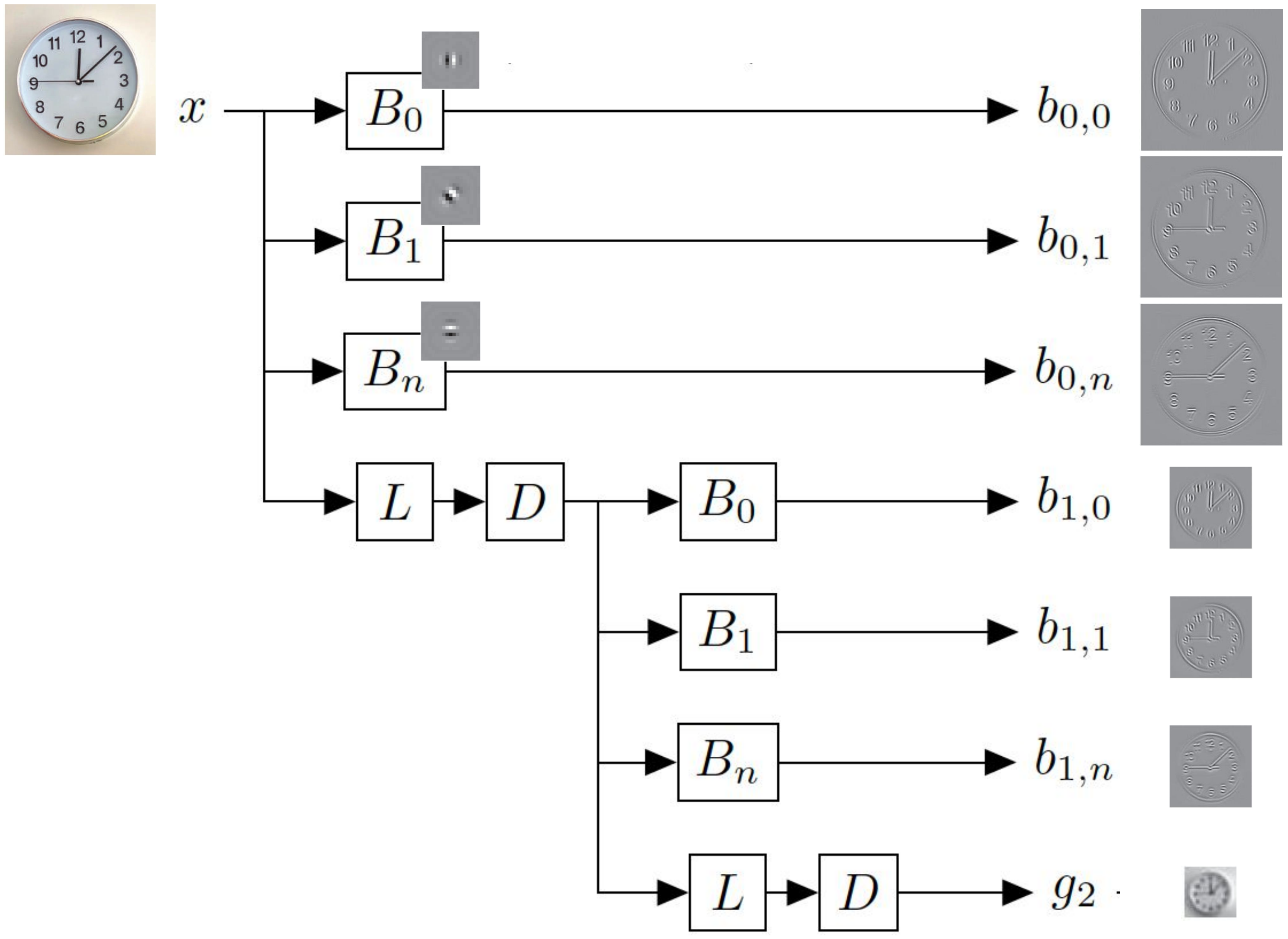
Gaussian Pyr

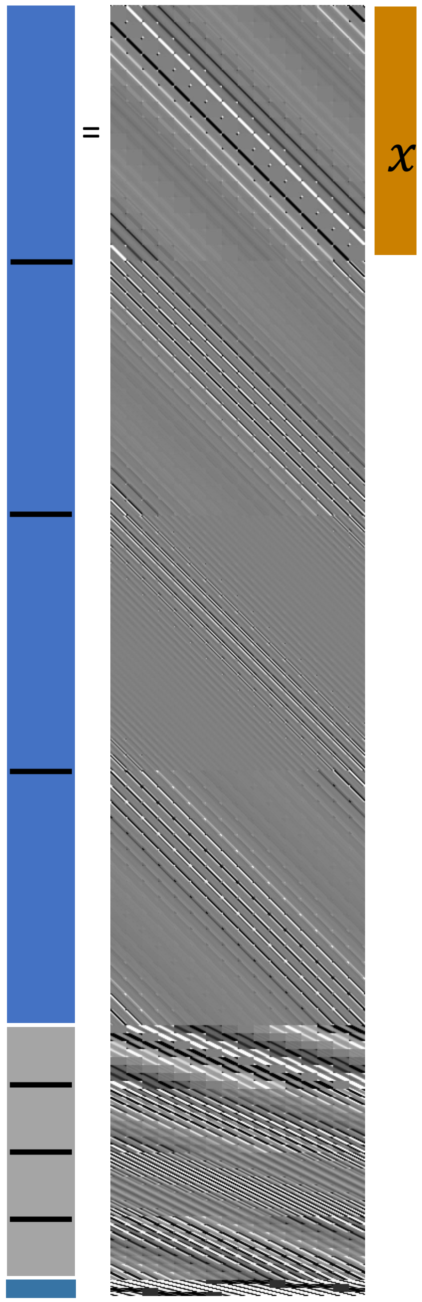


Laplacian Pyr

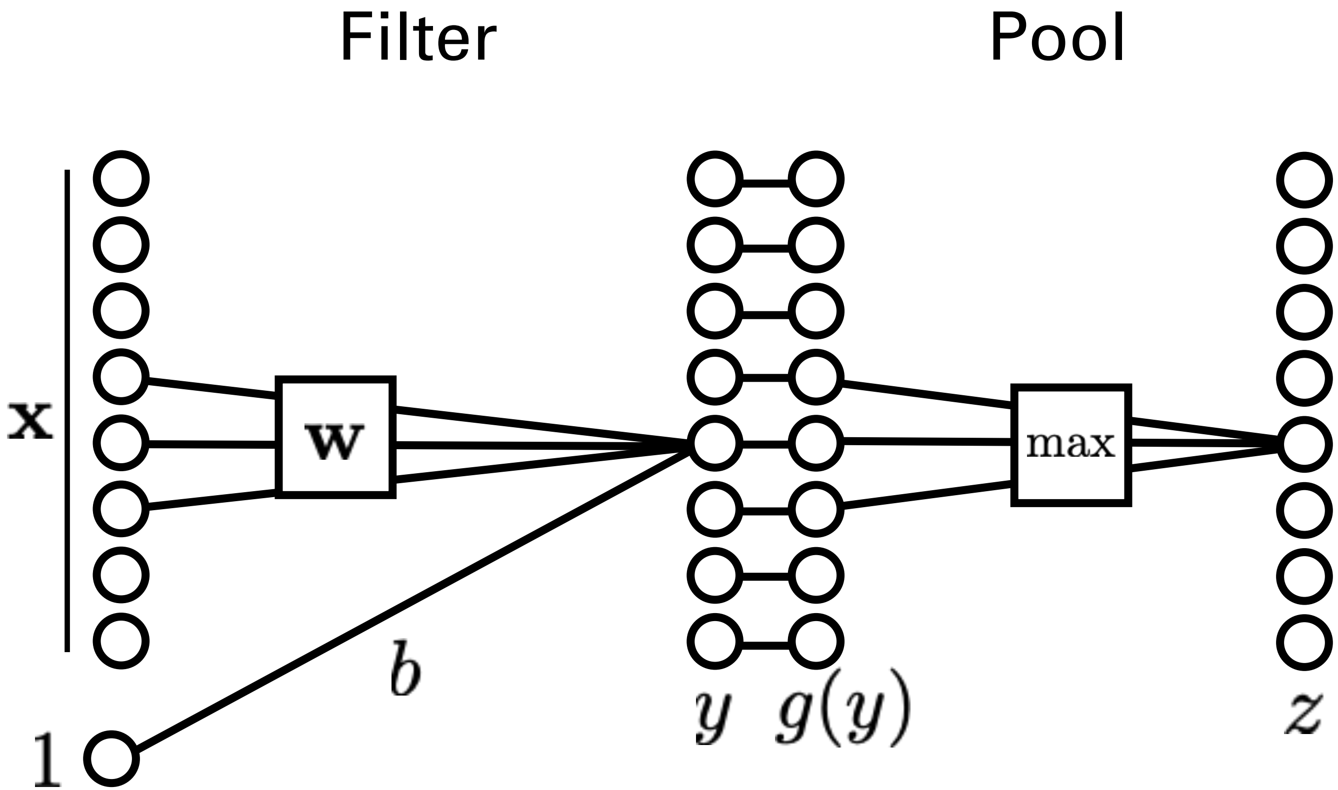
How can we use multi-scale modeling in Convnets?

Steerable Pyramid





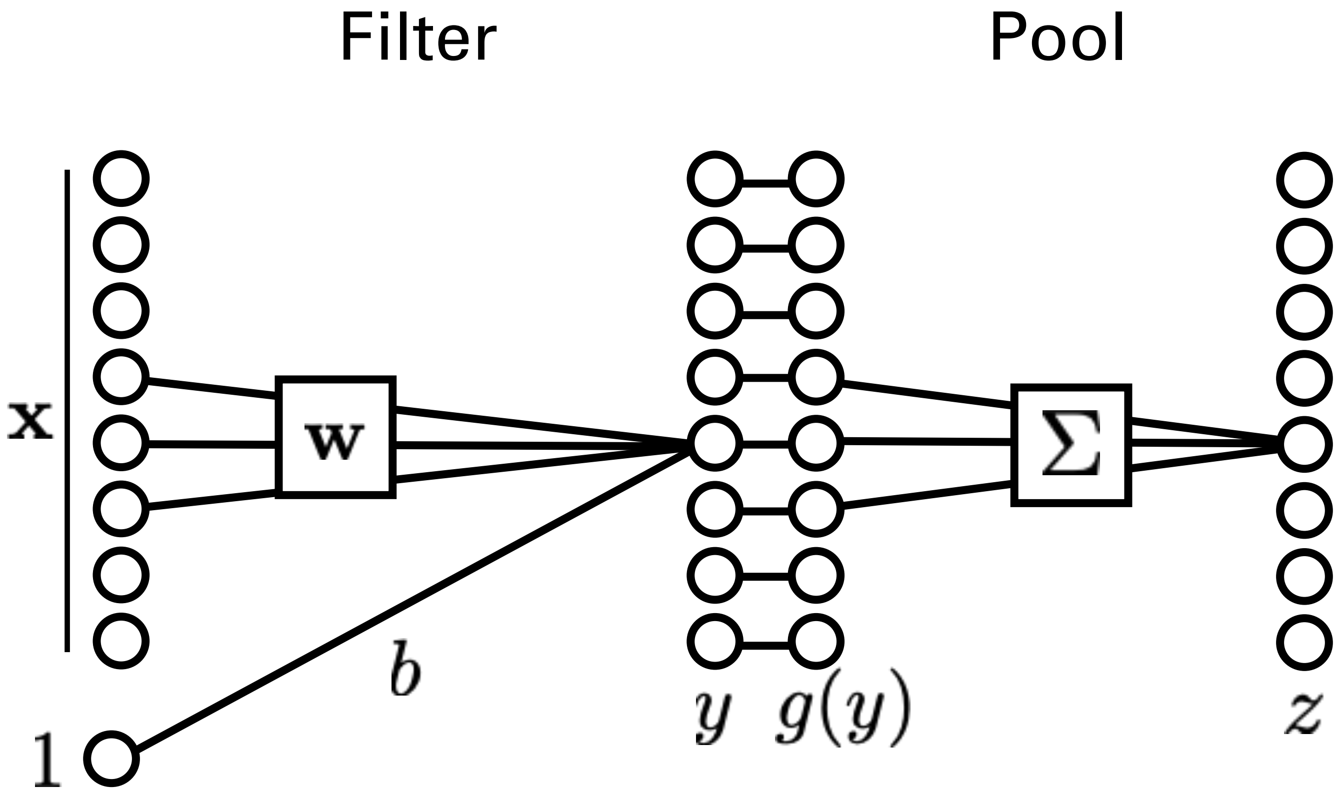
Pooling



Max pooling

$$z_k = \max_{j \in \mathcal{N}(j)} g(y_j)$$

Pooling



Max pooling

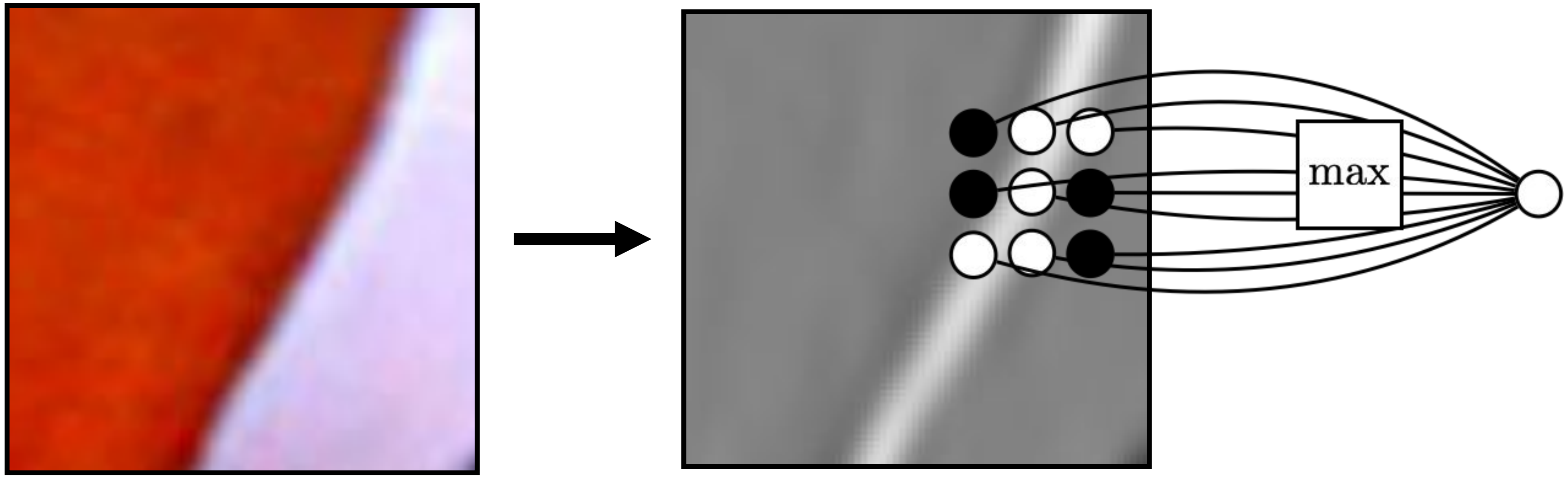
$$z_k = \max_{j \in \mathcal{N}(j)} g(y_j)$$

Mean pooling

$$z_k = \frac{1}{|\mathcal{N}|} \sum_{j \in \mathcal{N}(j)} g(y_j)$$

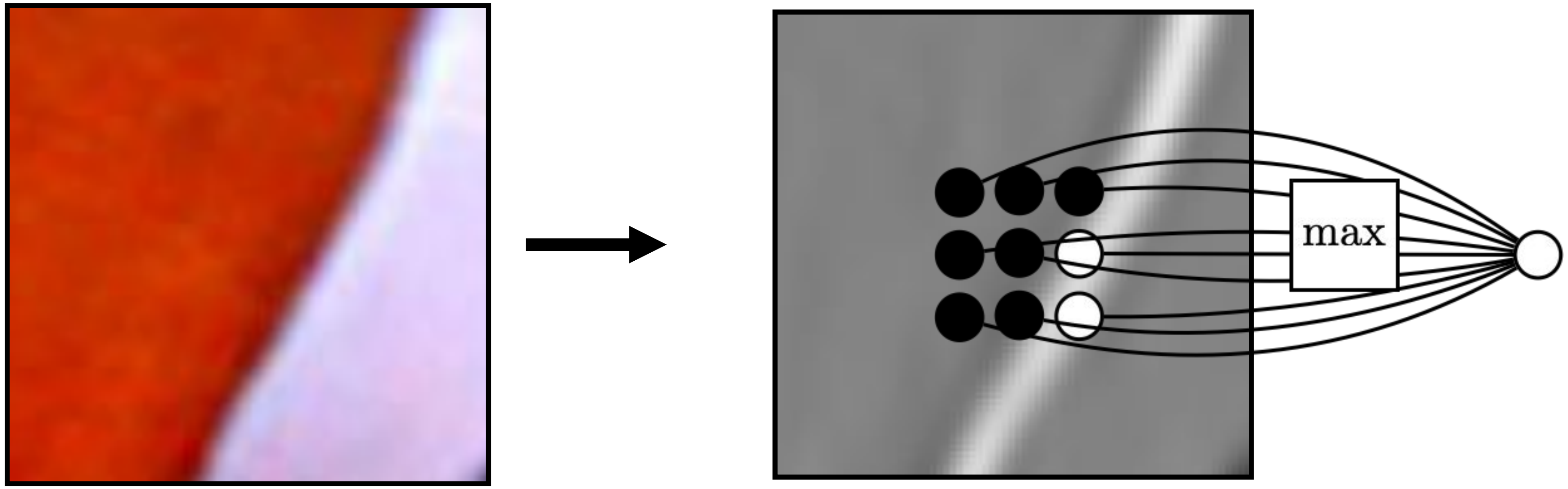
Pooling – Why?

Pooling across spatial locations achieves stability w.r.t. small translations:



Pooling – Why?

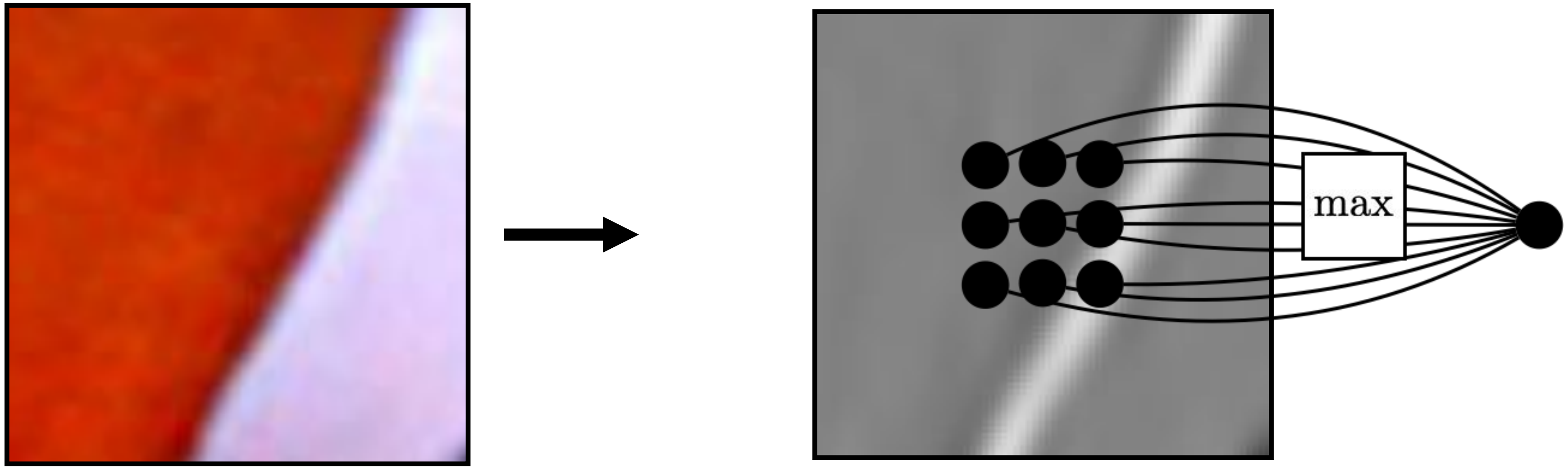
Pooling across spatial locations achieves stability w.r.t. small translations:



large response regardless of exact position of edge

Pooling – Why?

Pooling across spatial locations achieves stability w.r.t. small translations:



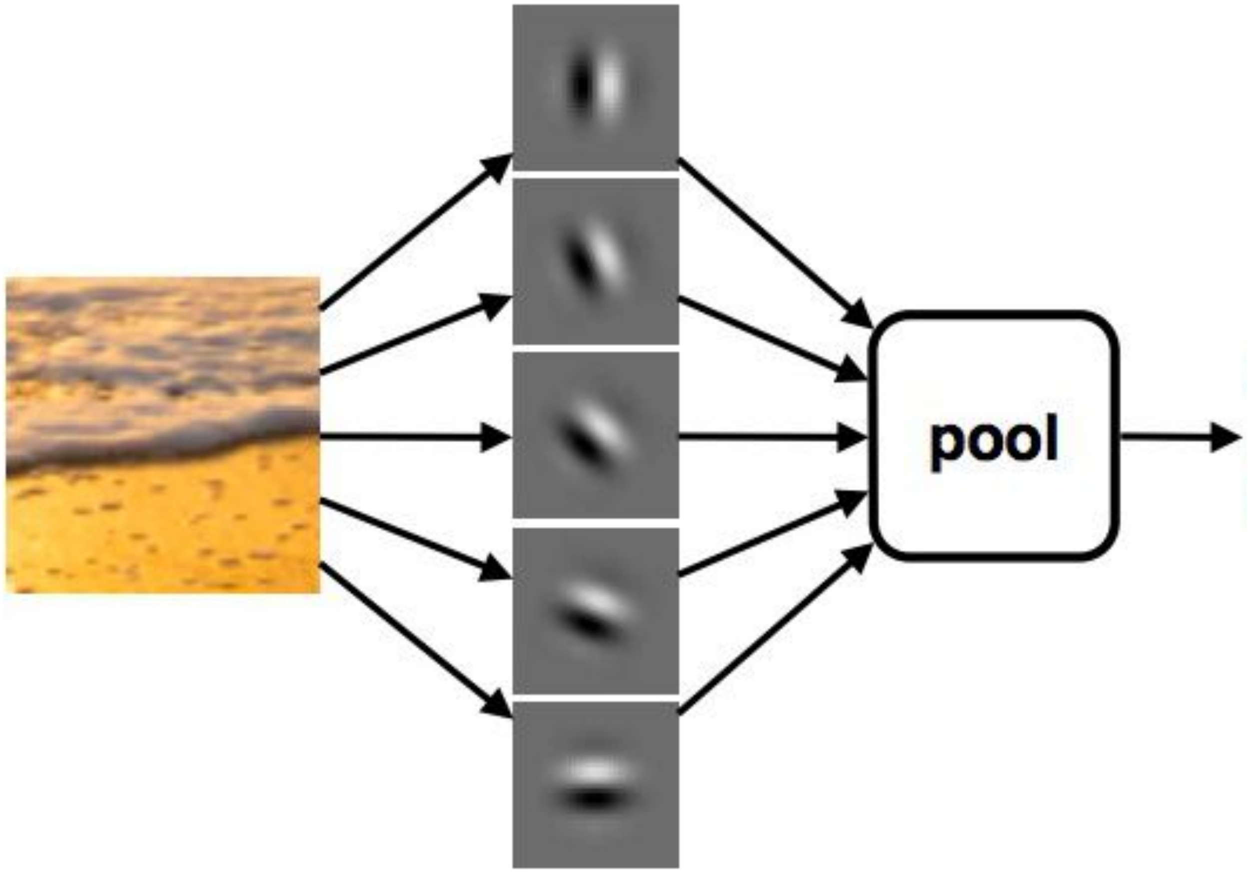
CNNs are stable w.r.t. diffeomorphisms

$\approx$

[“Unreasonable effectiveness of Deep Features as a Perceptual Metric”, Zhang et al. 201

Pooling – Why?

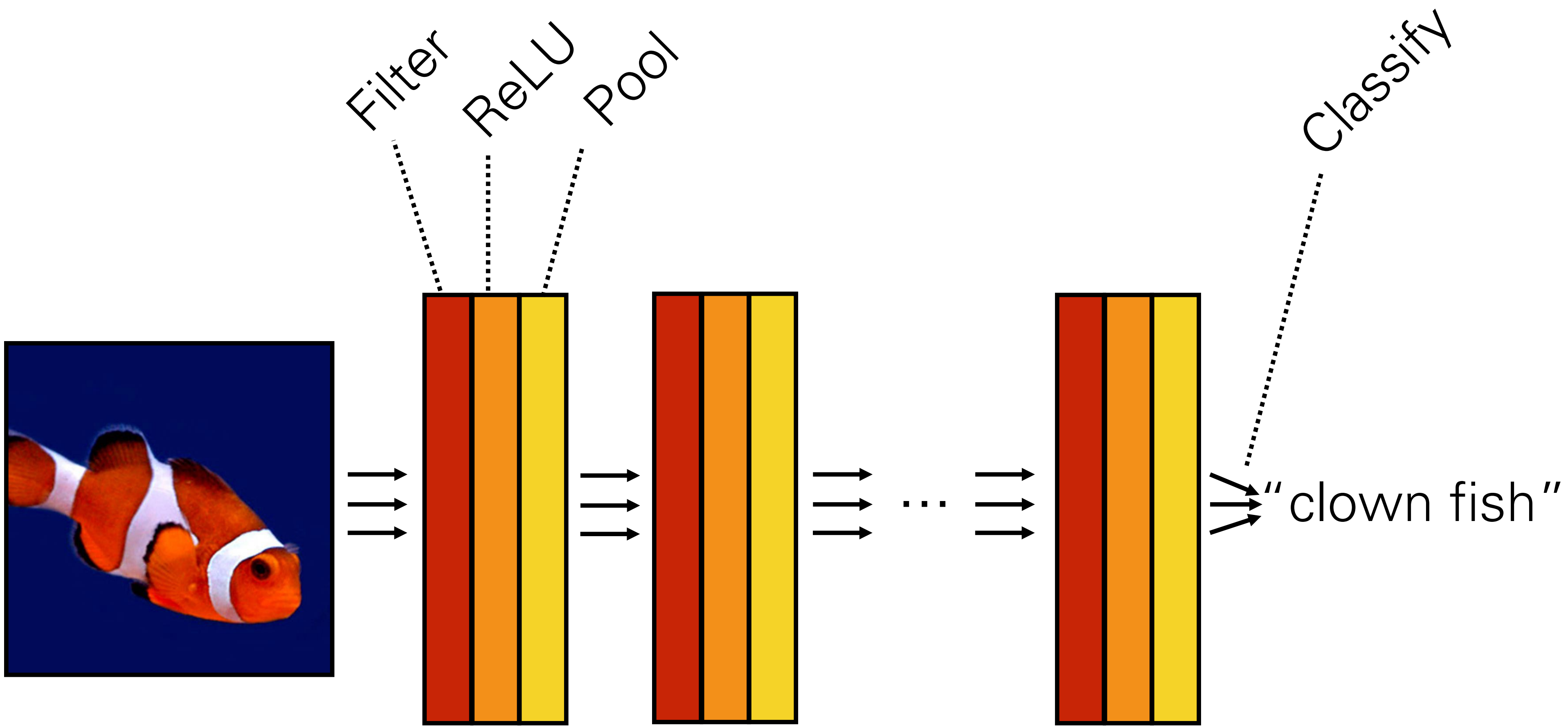
Pooling across feature channels (filter outputs) can achieve other kinds of invariances:



large response for any edge, regardless of its orientation

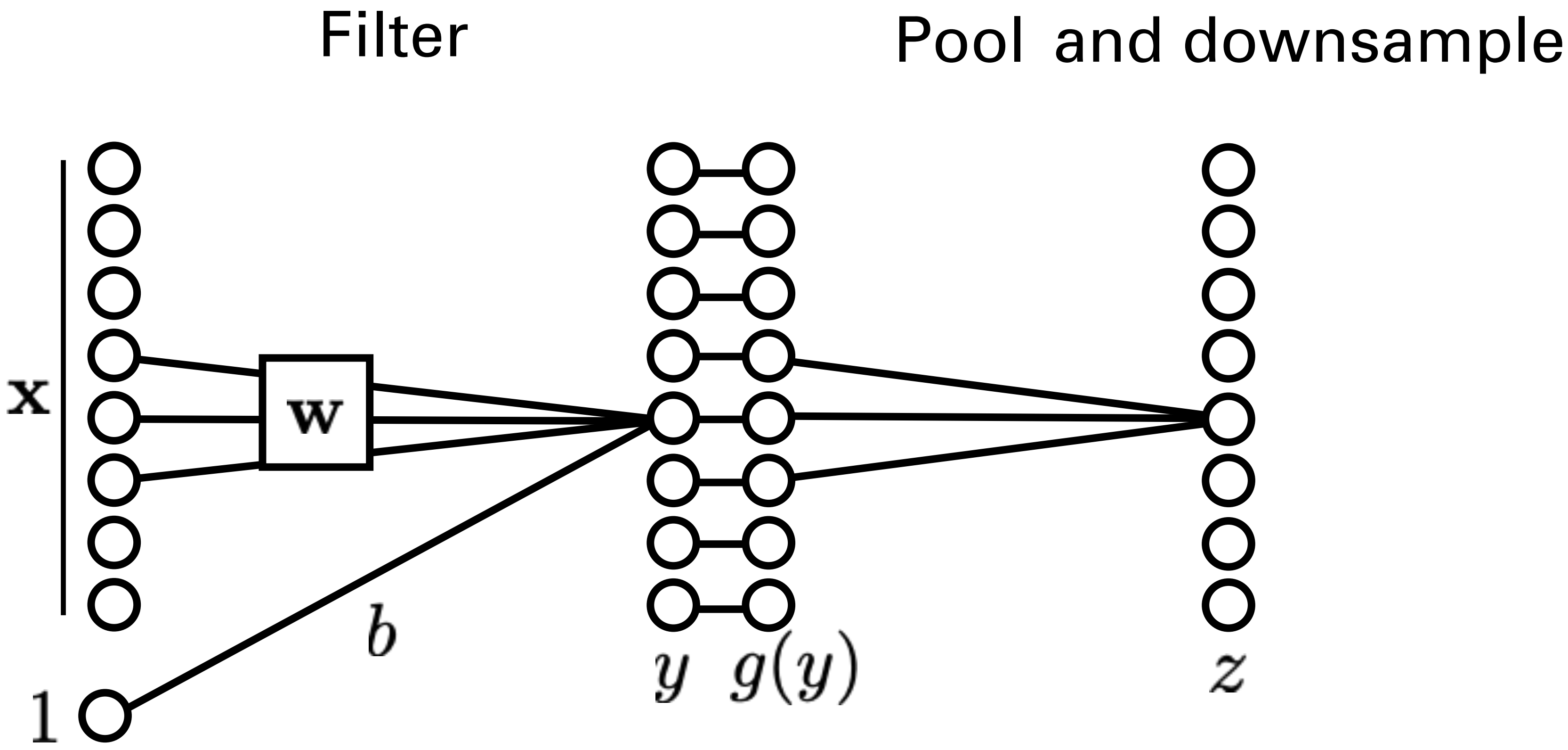
[Derived from slide by Andrea Vedaldi]

Computation in a neural net

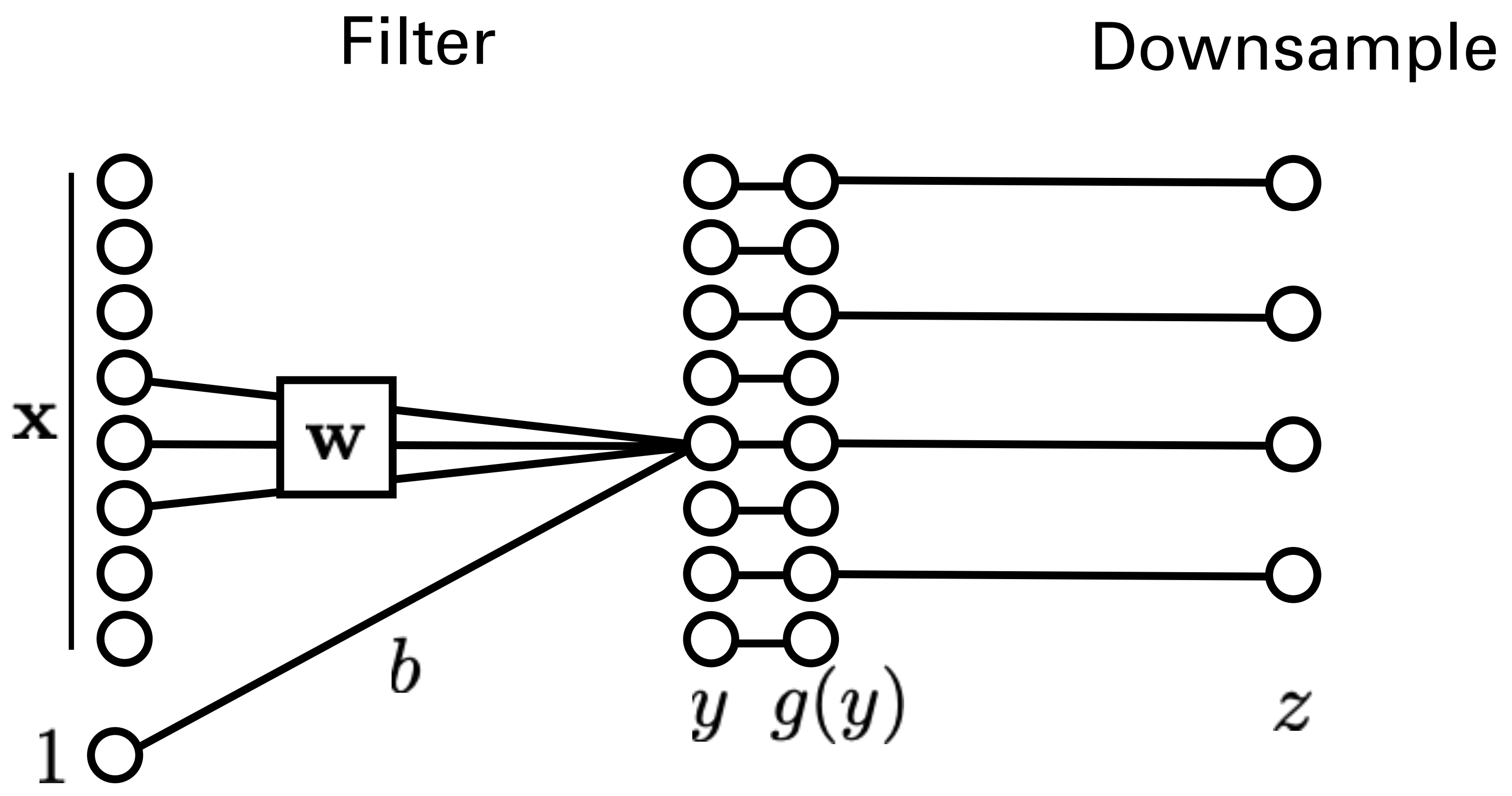


$$f(\mathbf{x}) = f_L(\dots f_2(f_1(\mathbf{x})))$$

Downsampling

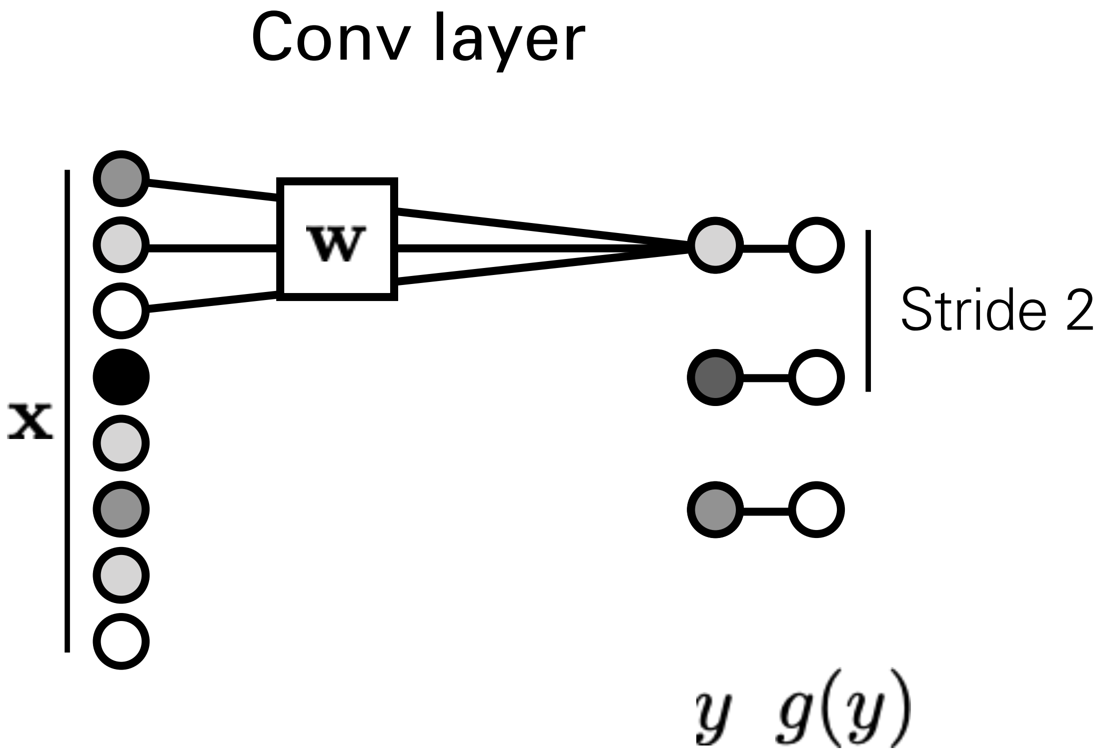


Downsampling



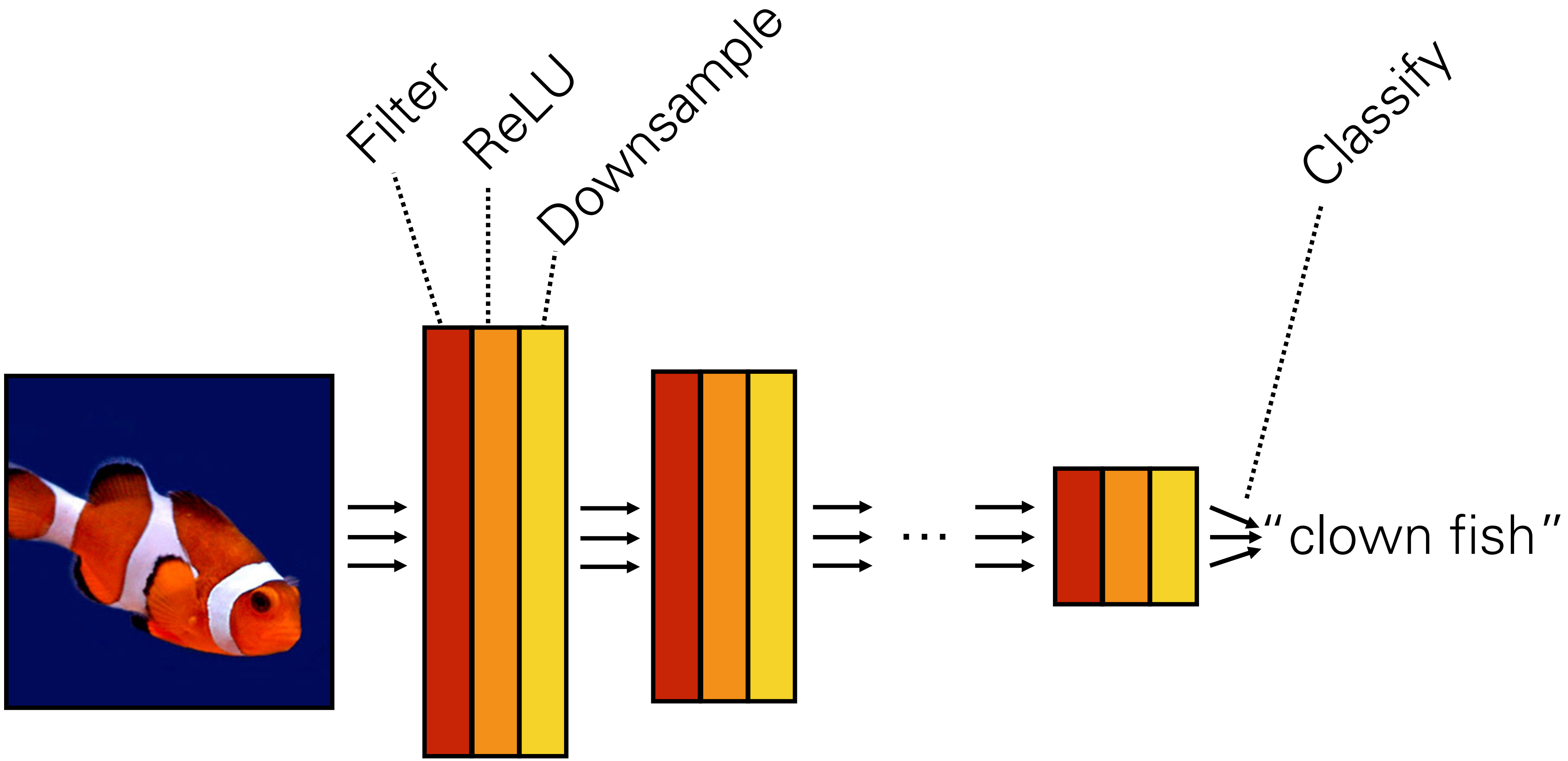
$$\mathbb{R}^{H^{(l)} \times W^{(l)} \times C^{(l)}} \to \mathbb{R}^{H^{(l+1)} \times W^{(l+1)} \times C^{(l+1)}}$$

Strided convolution



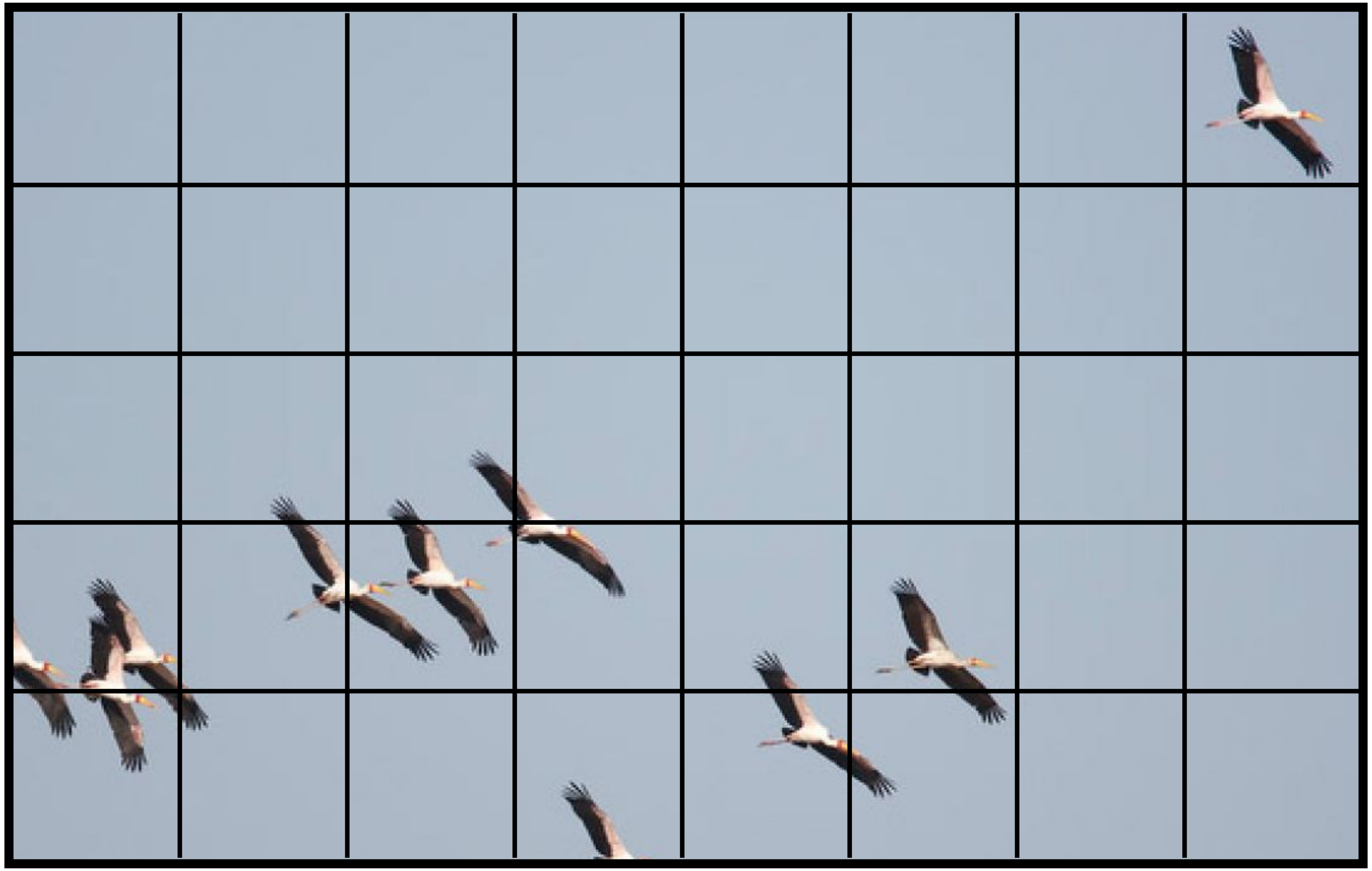
Strided convolutions combine convolution and downsampling into a single operation.

Computation in a neural net

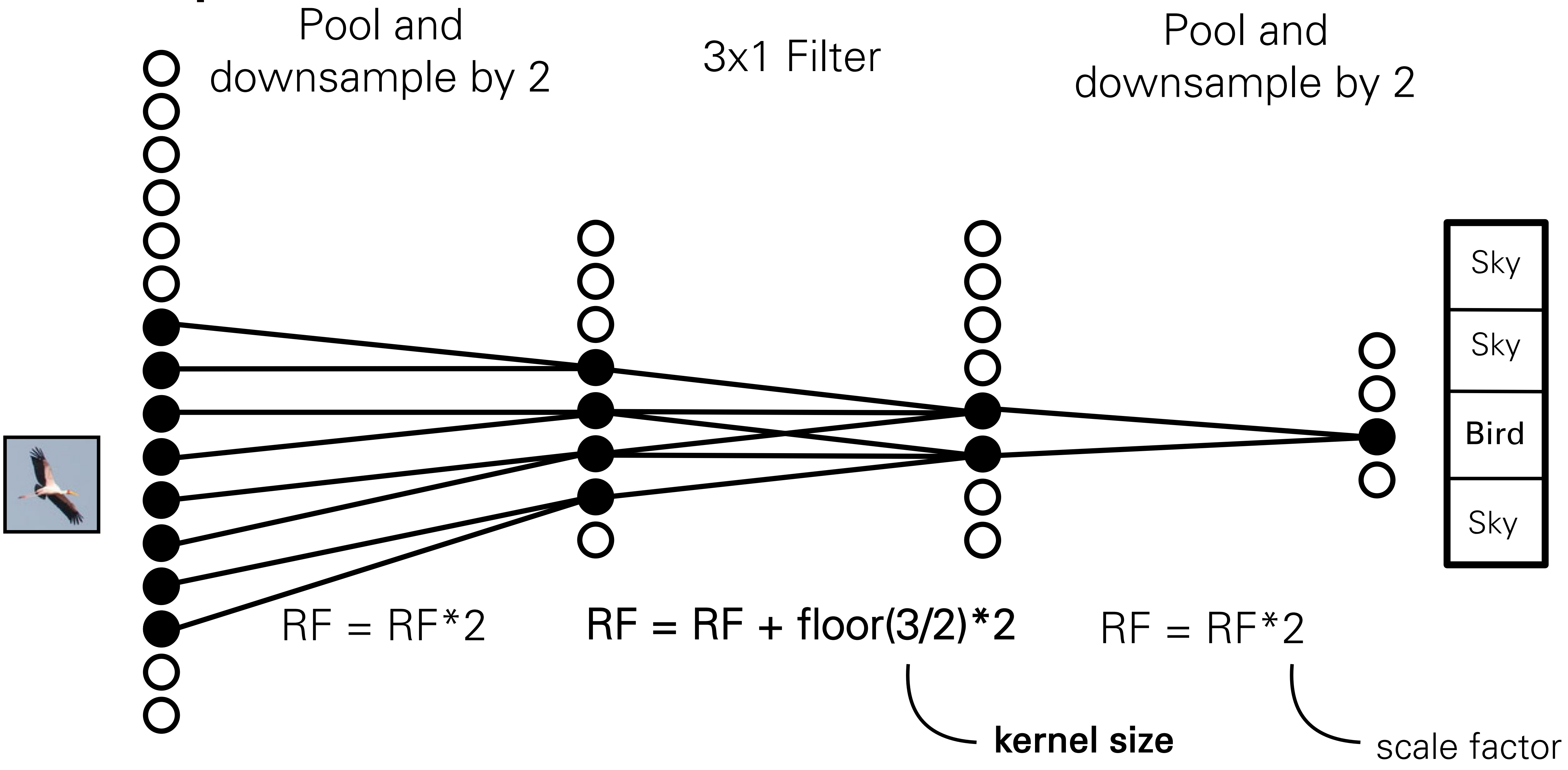


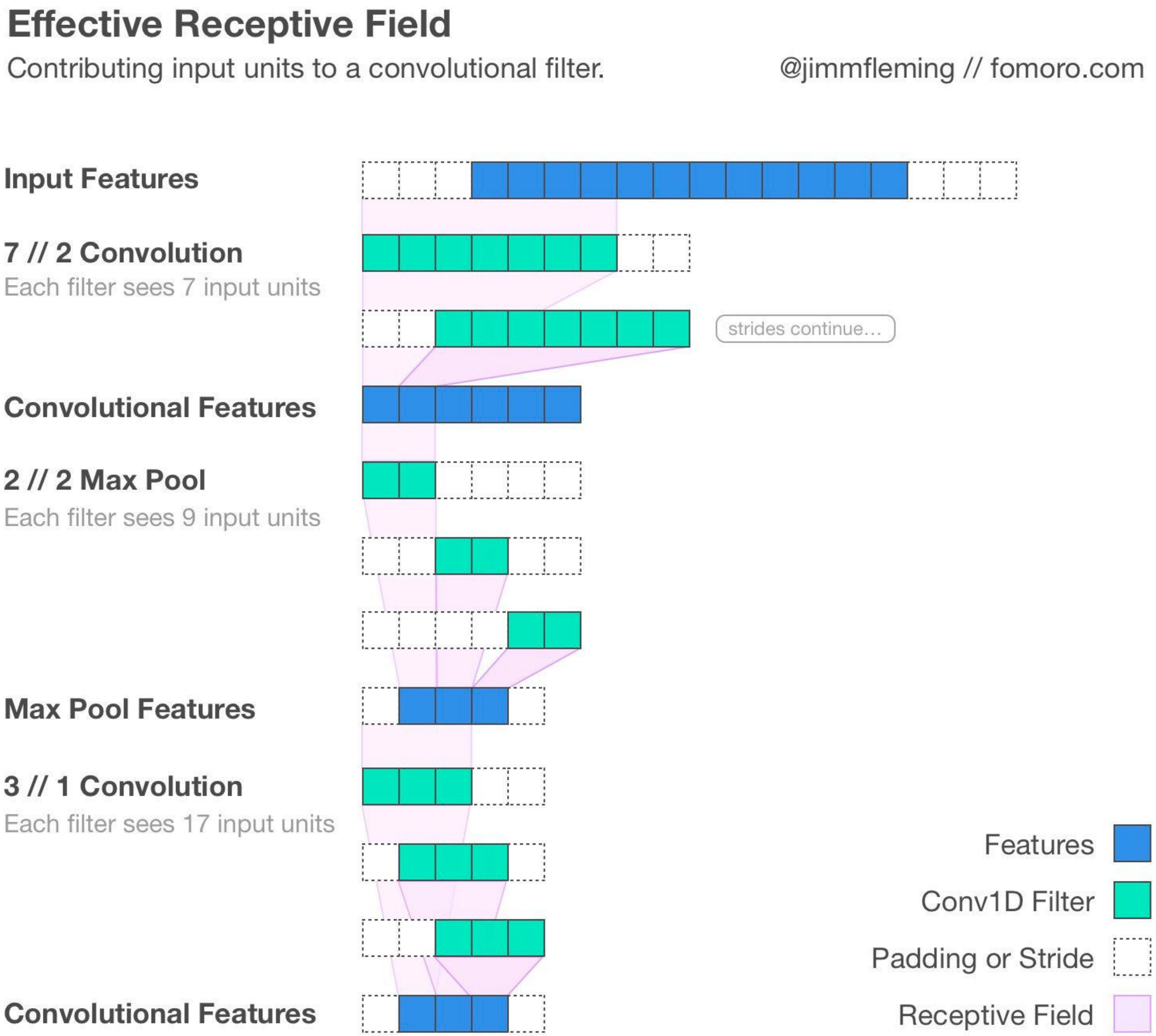
$$f(\mathbf{x}) = f_L(\ldots f_2(f_1(\mathbf{x})))$$

Receptive fields



Receptive fields





[http://fomoro.com/tools/receptive-fields/index.html]

CNNs – Why?

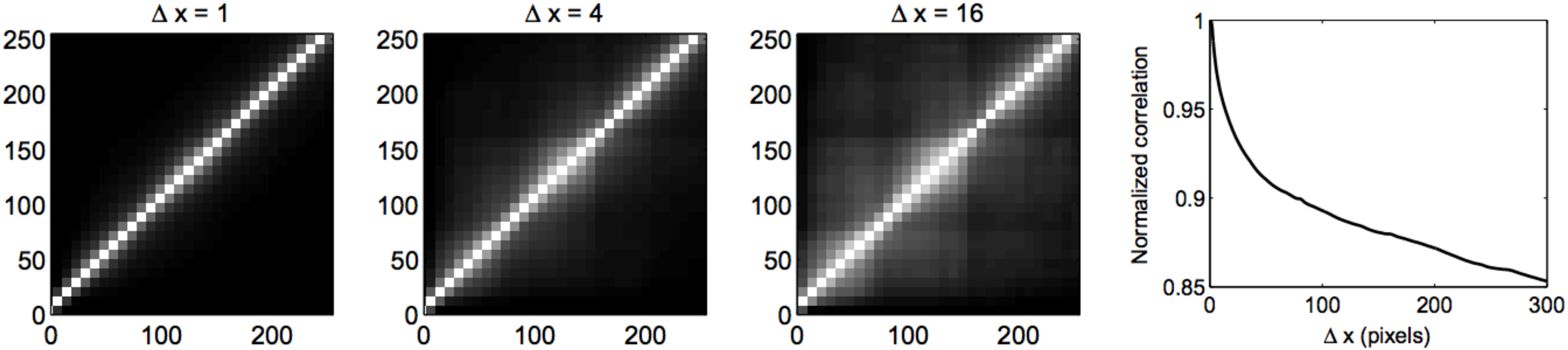


Fig. 1. (a) Scatterplots of pairs of pixels at three different spatial displacements, averaged over five examples images. (b) Autocorrelation function. Photographs are of New York City street scenes, taken with a Canon 10D digital camera, and processed in RAW linear sensor mode (producing pixel intensities are in roughly proportional to light intensity). Correlations were computed on the logs of these sensor intensity values [41].

[http://6.869.csail.mit.edu/fa18/notes/simoncelli2005.pdf]

CNNs – Why?

Statistical dependences between pixels decay as a power law of distance between the pixels.

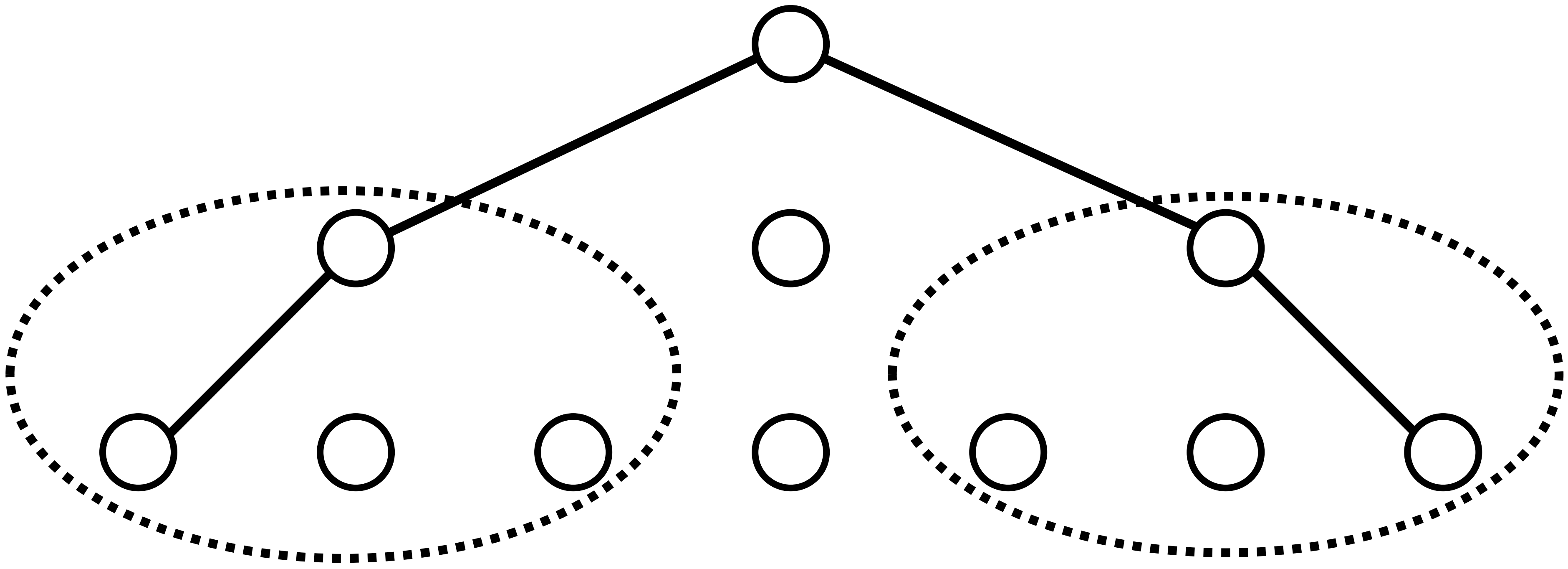
It is therefore often sufficient to model local dependences only. —> **Convolution**

More generally, we should allocate parameters that model dependences in proportion to the strength of those dependences. —> **Multiscale, hierarchical representations**

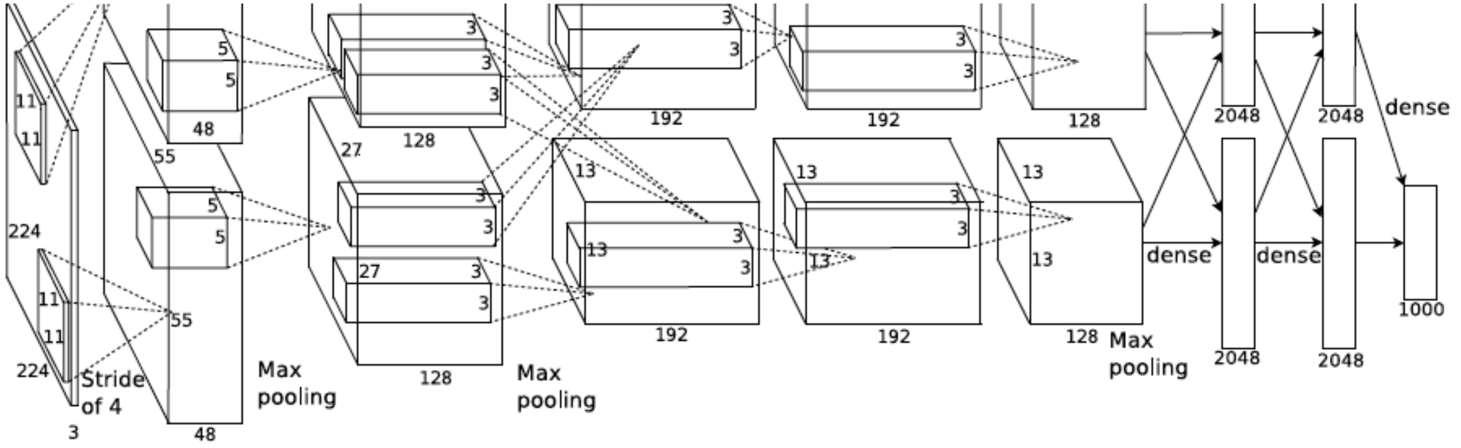
[For more discussion, see "Why does Deep and Cheap Learning Work So Well?", Lin et al. 2017]

CNNs – Why?

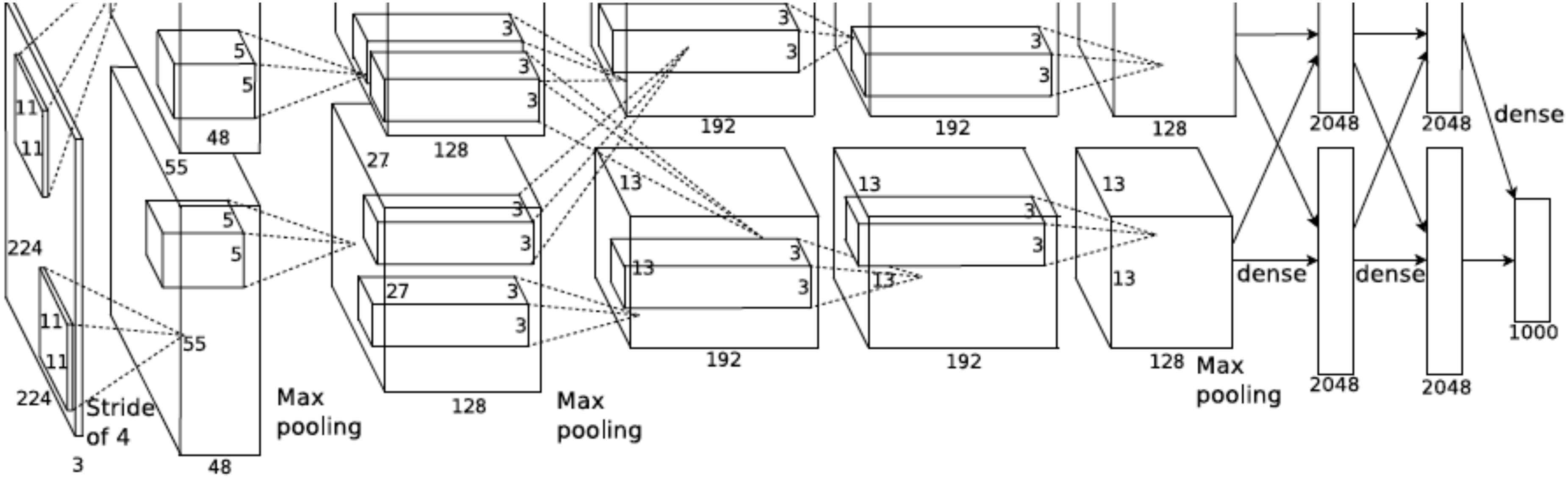
Capturing long-range dependences:



Alexnet — [Krizhevsky et al. NIPS 2012]



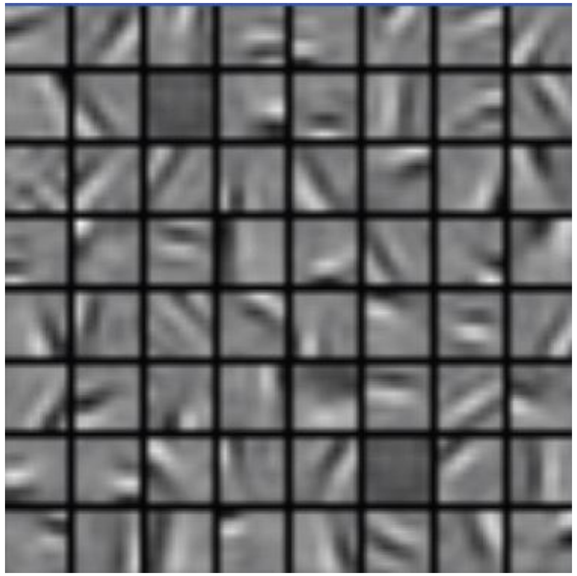
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2 [13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1 [13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1 [13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)



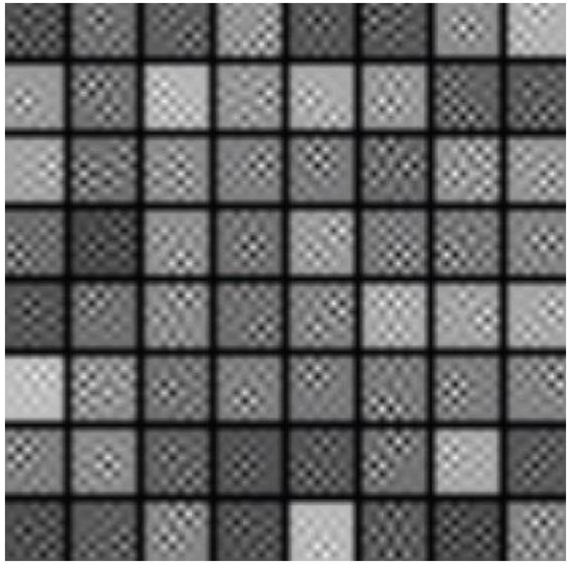
What filters are learned?

What filters are learned?

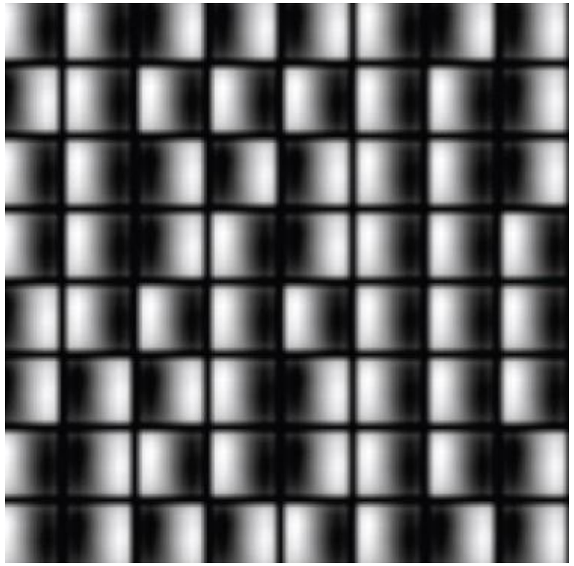
A



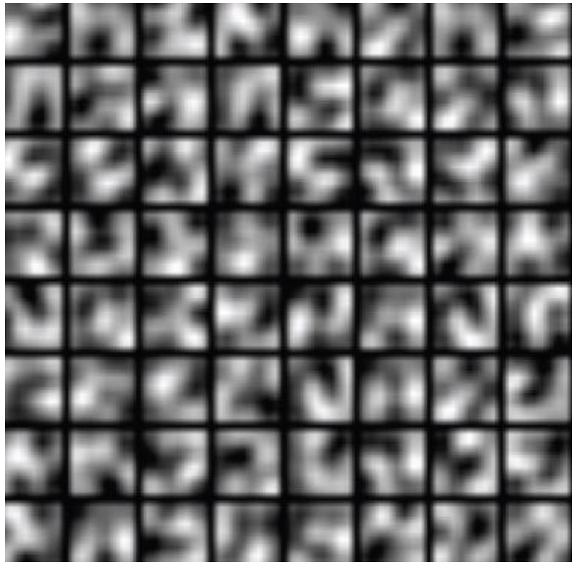
B



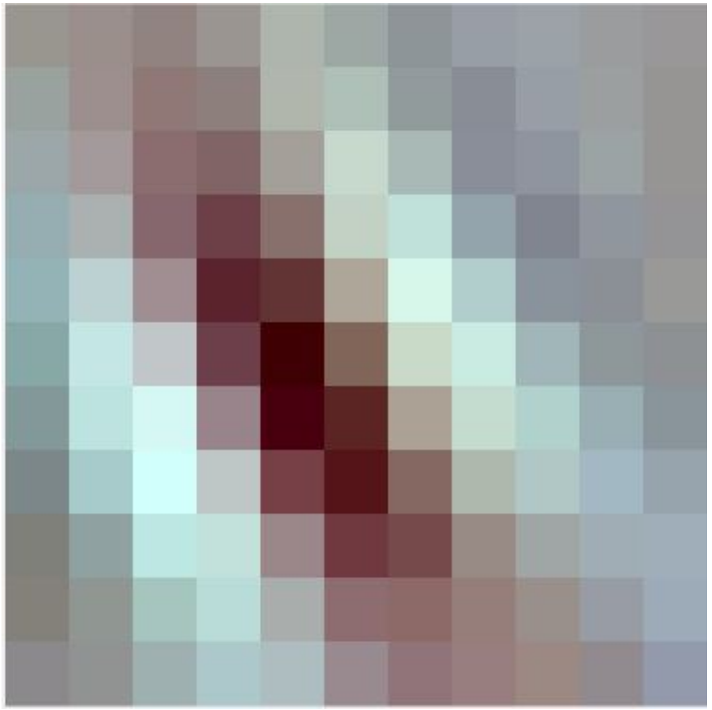
C

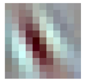


D

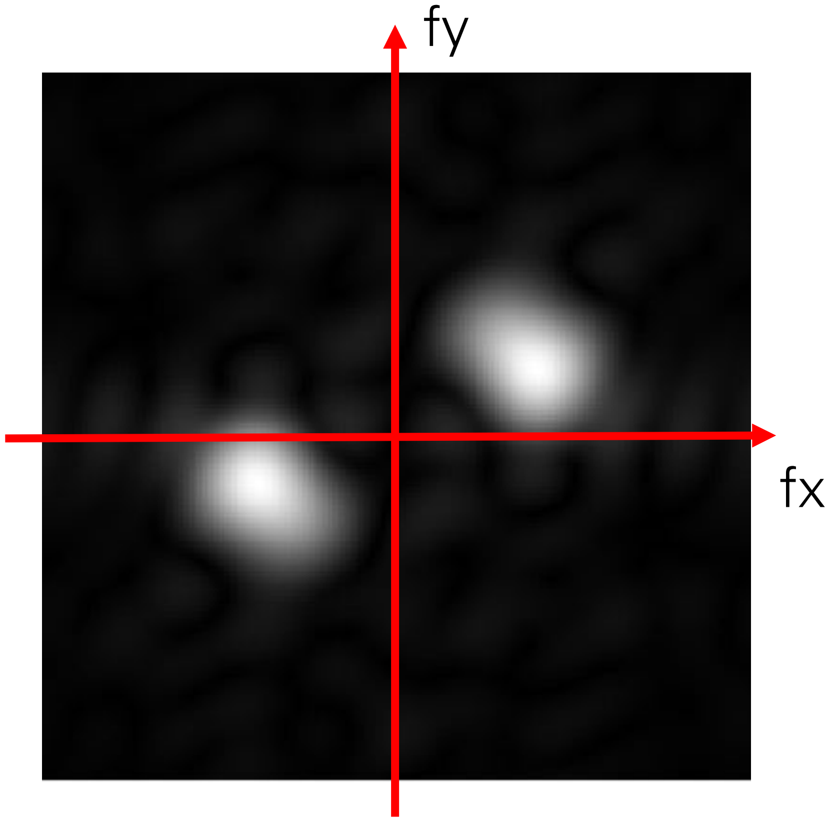


Get to know your units

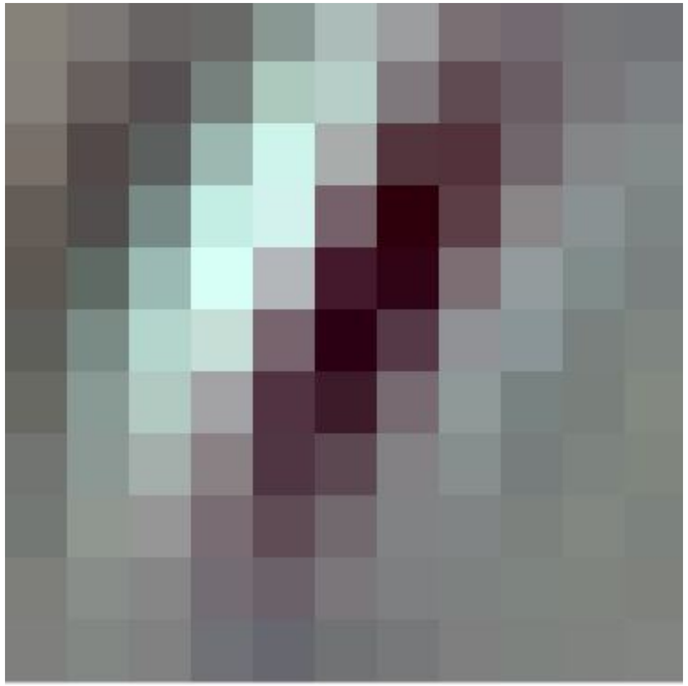


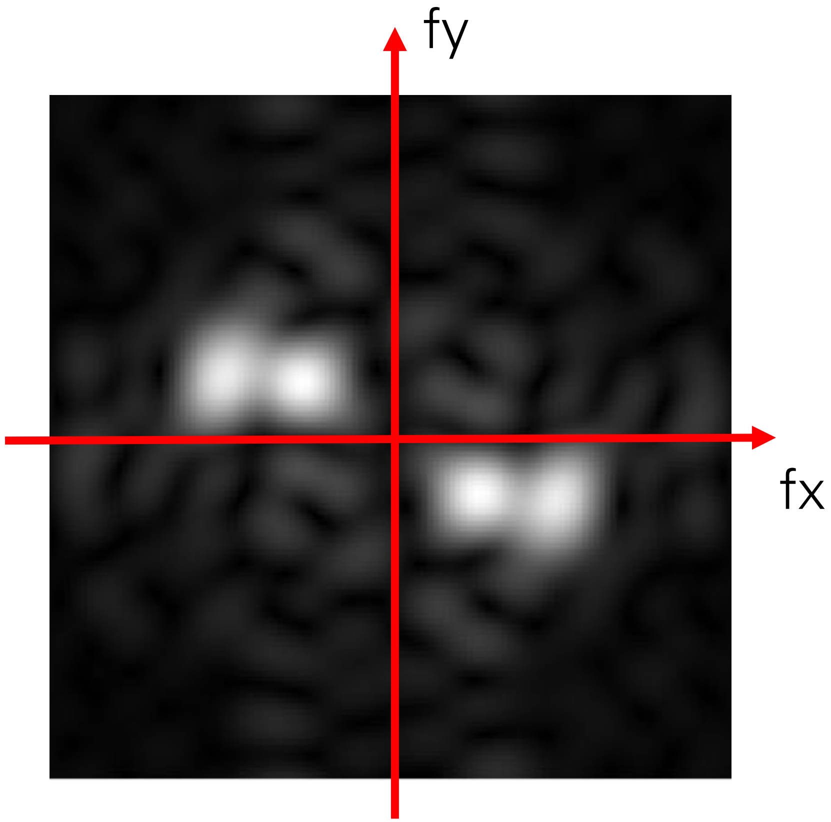


11x11 convolution kernel
(3 color channels)

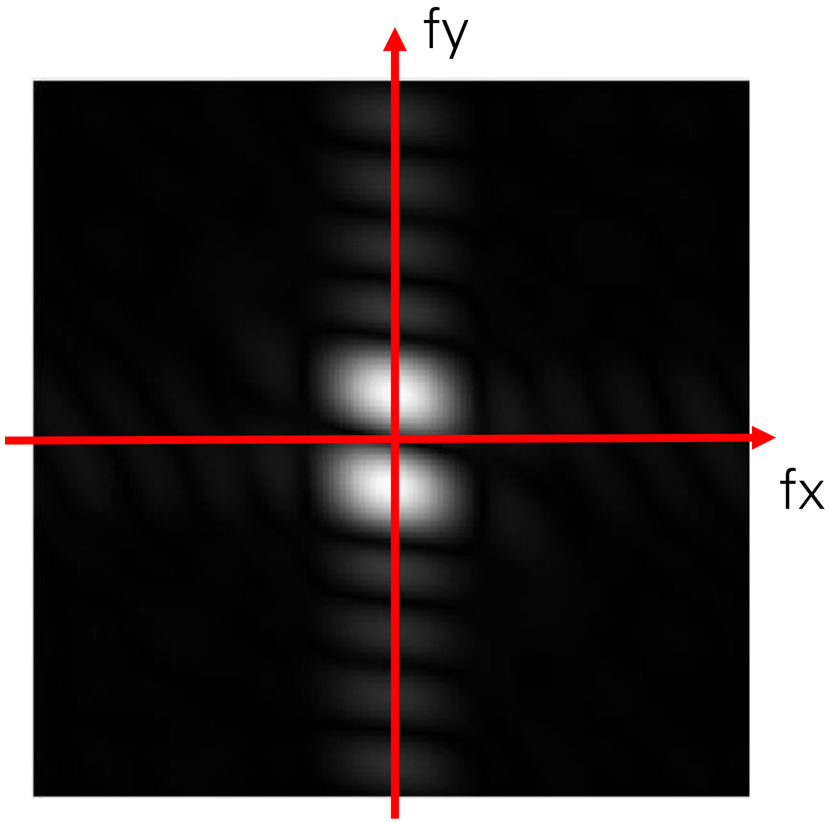


Get to know your units

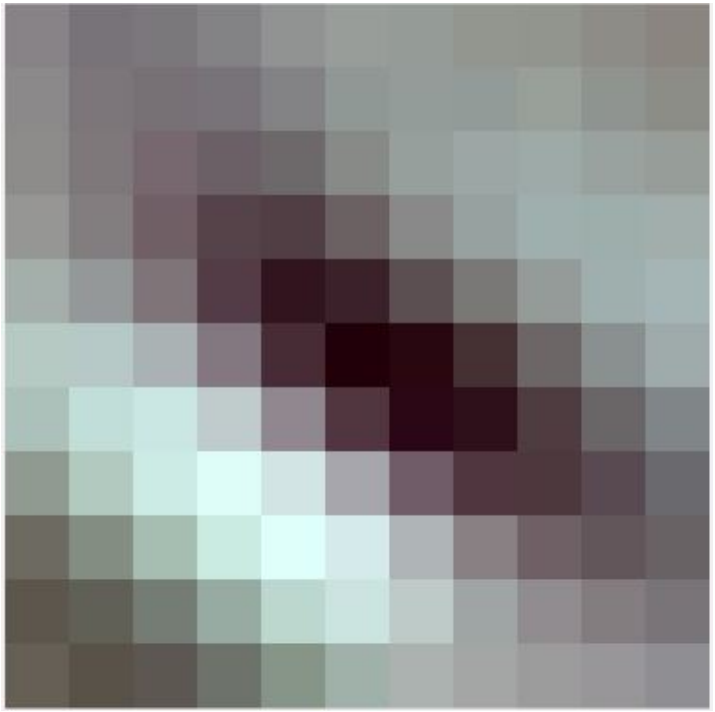


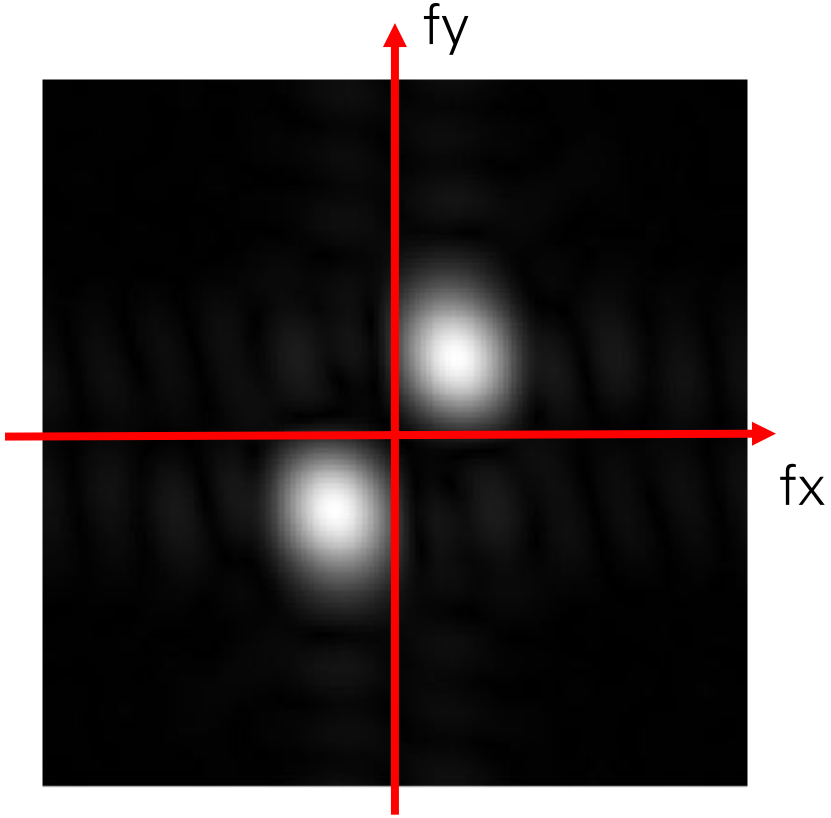


Get to know your units

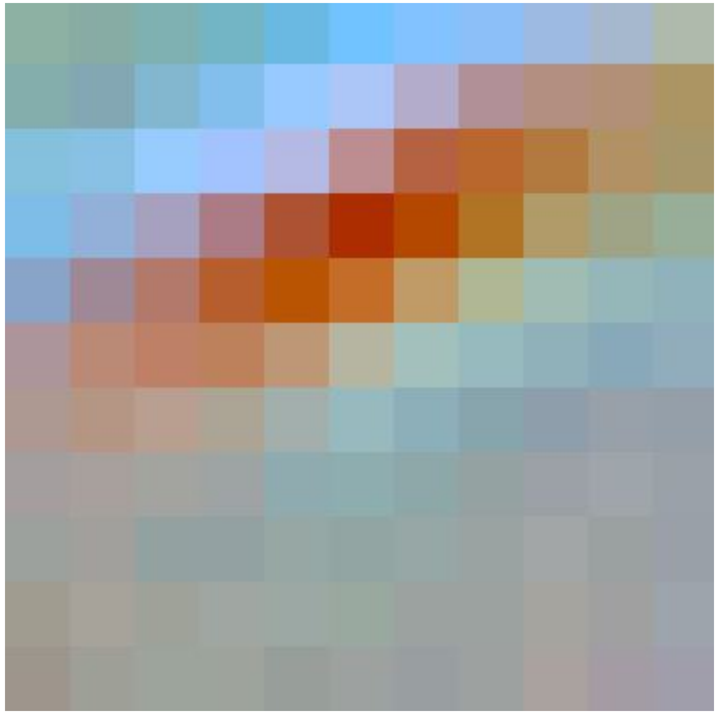


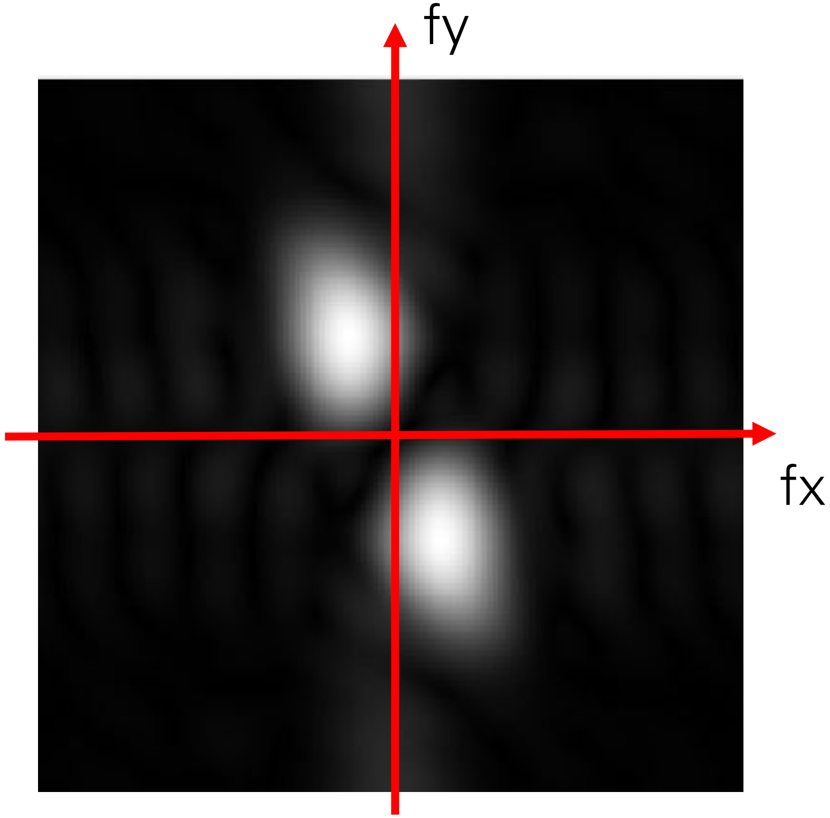
Get to know your units



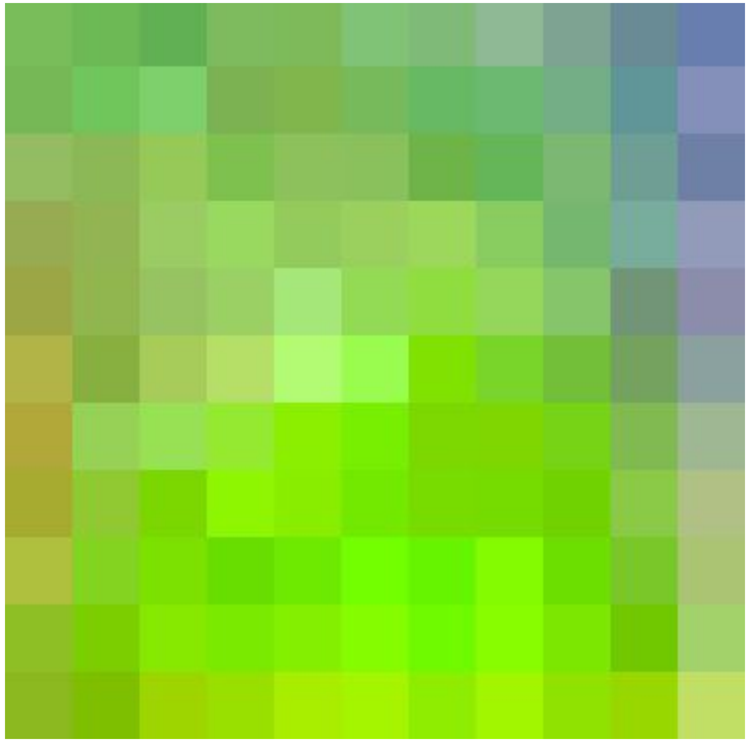


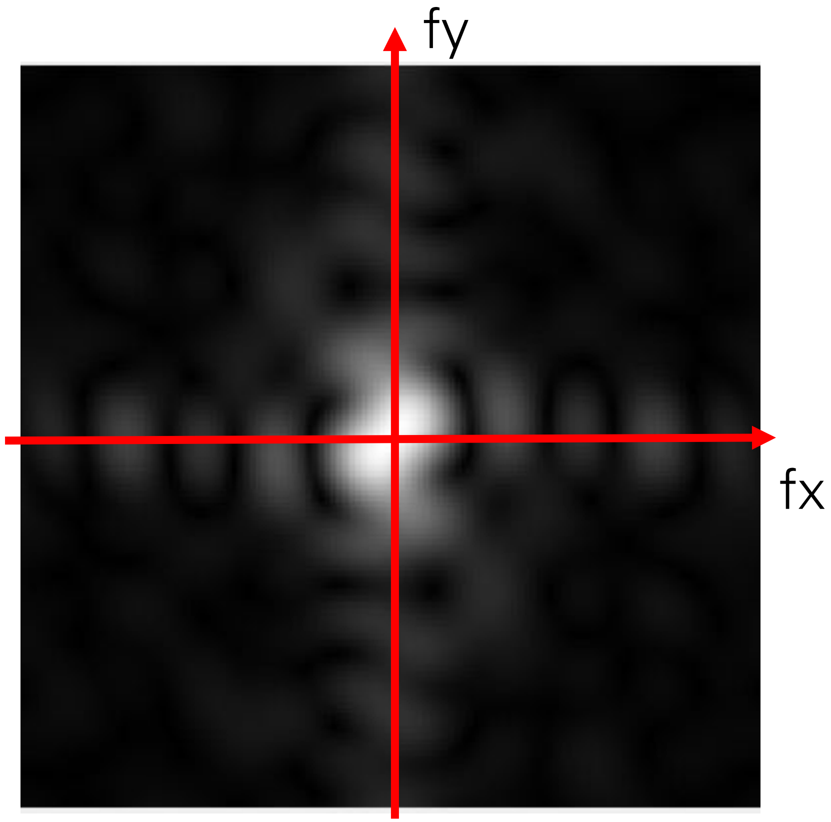
Get to know your units



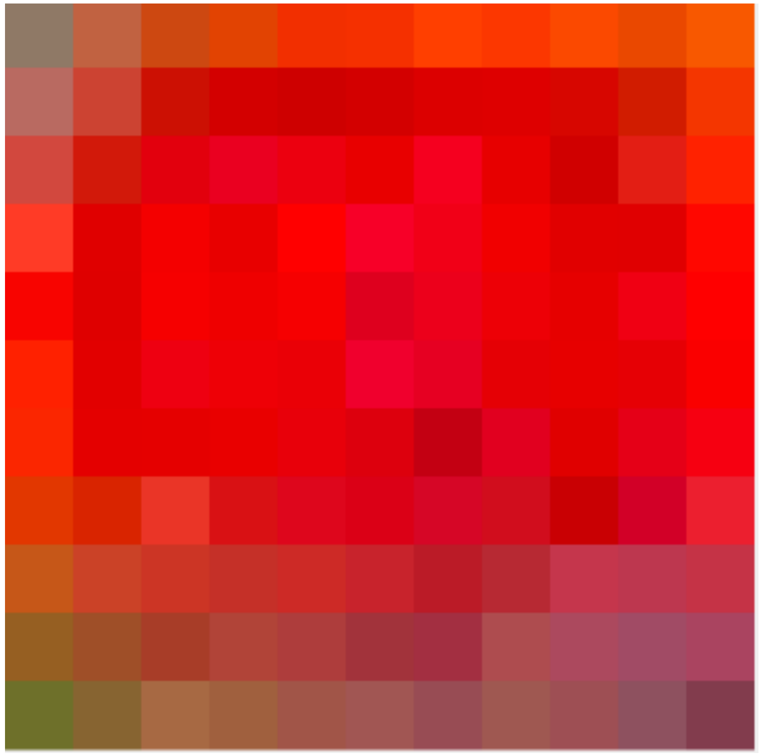


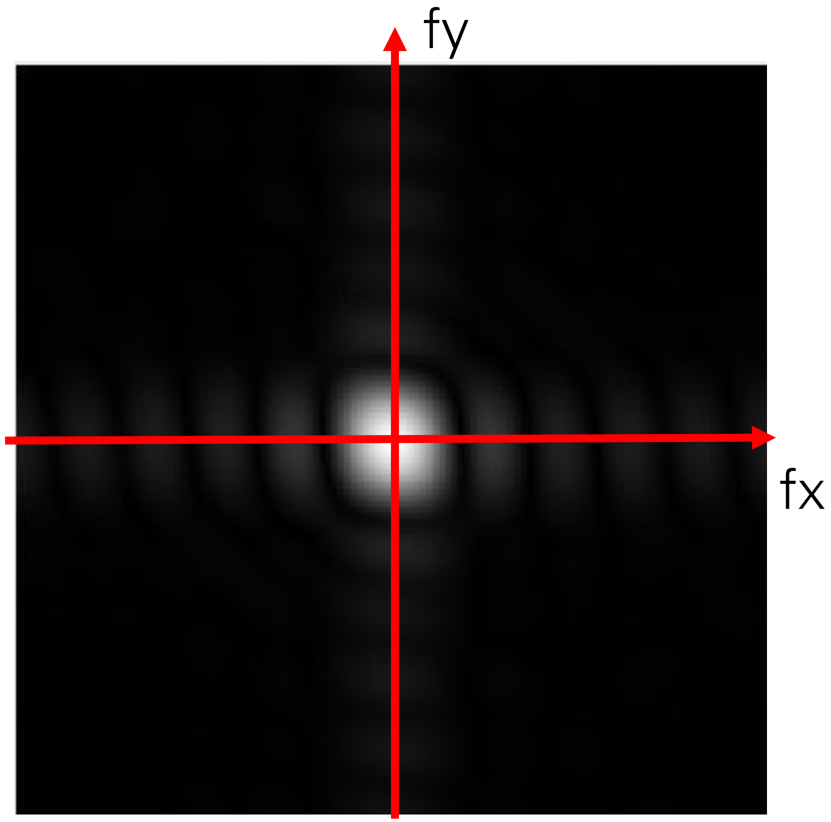
Get to know your units



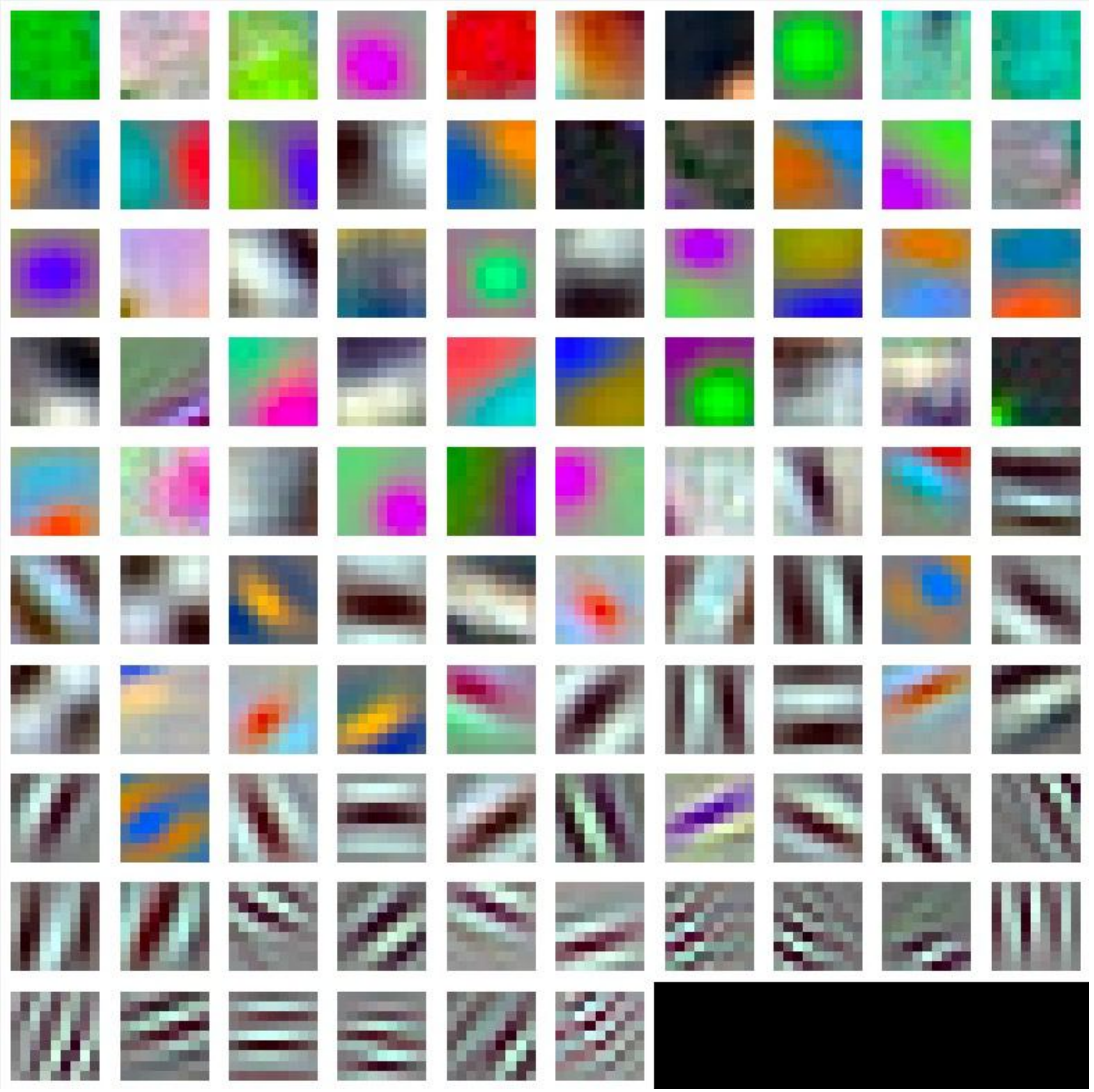


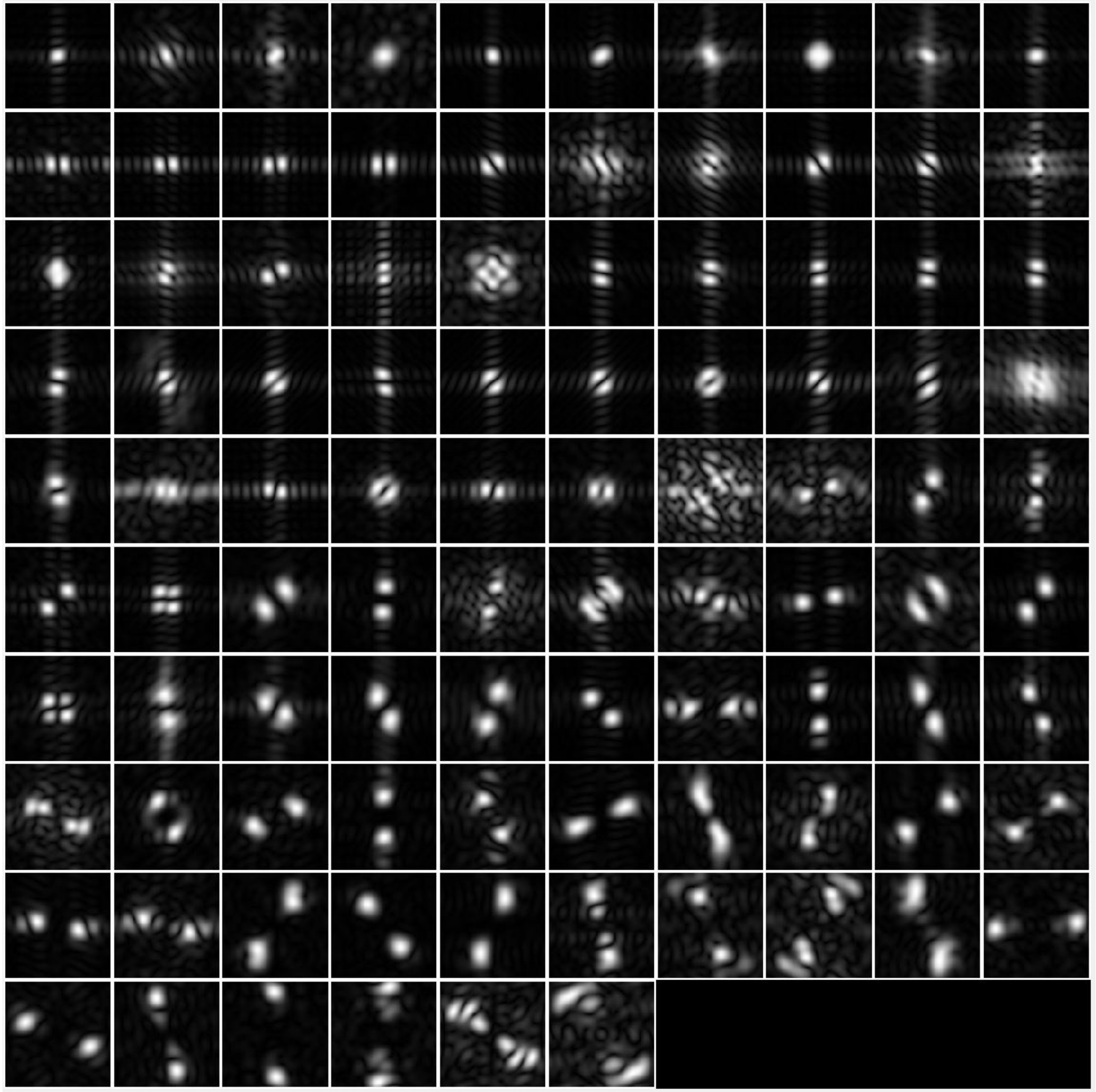
Get to know your units





Get to know your units

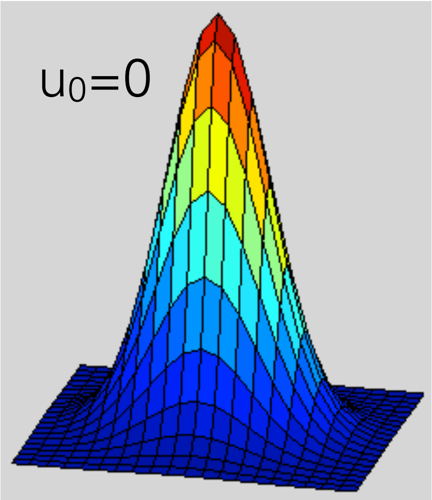


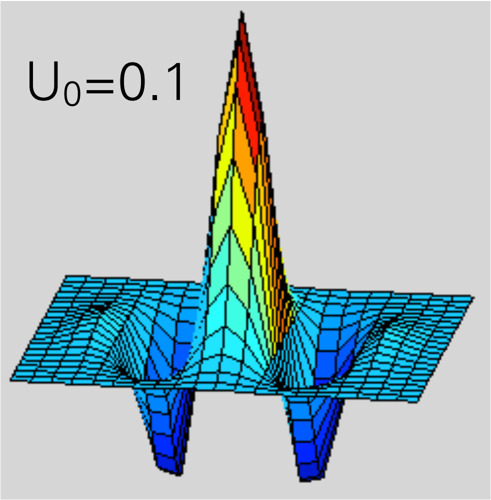


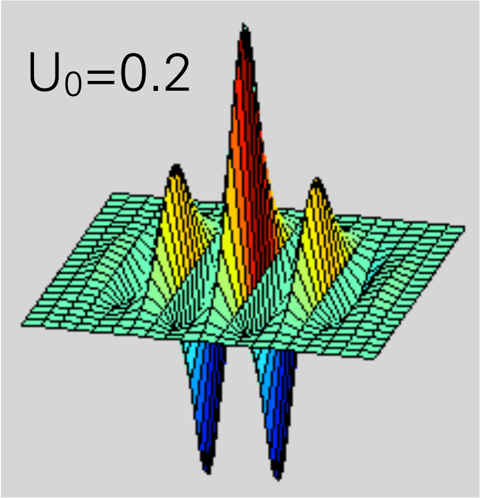
96 Units in conv1

Gabor wavelets

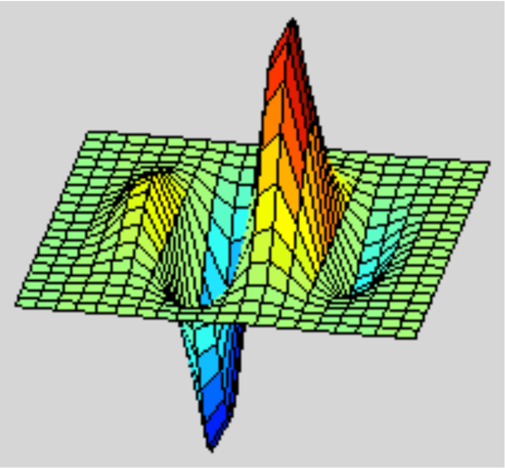
$$\psi_c(x,y) = e^{-\frac{x^2+y^2}{2\sigma^2}} \cos(2\pi u_0 x)$$

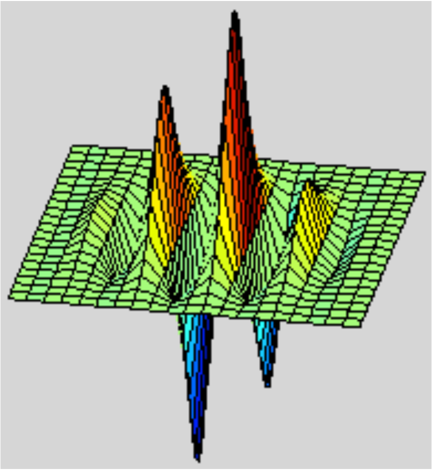


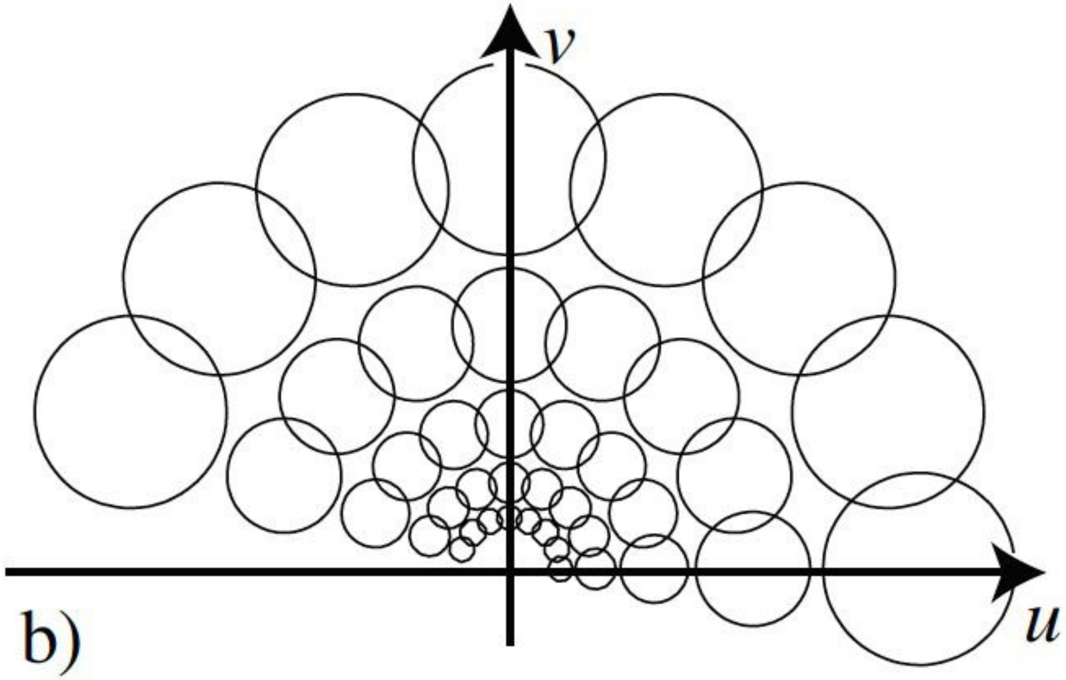


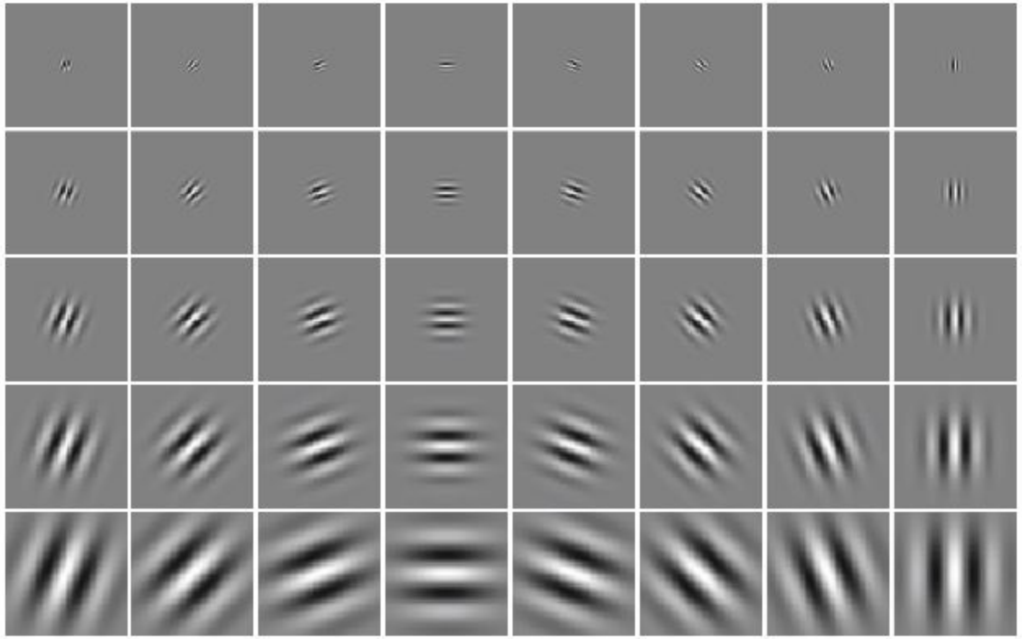


$$\psi_s(x,y) = e^{-\frac{x^2+y^2}{2\sigma^2}} \sin(2\pi u_0 x)$$






v
u
b)



Comparing Human and Machine Perception

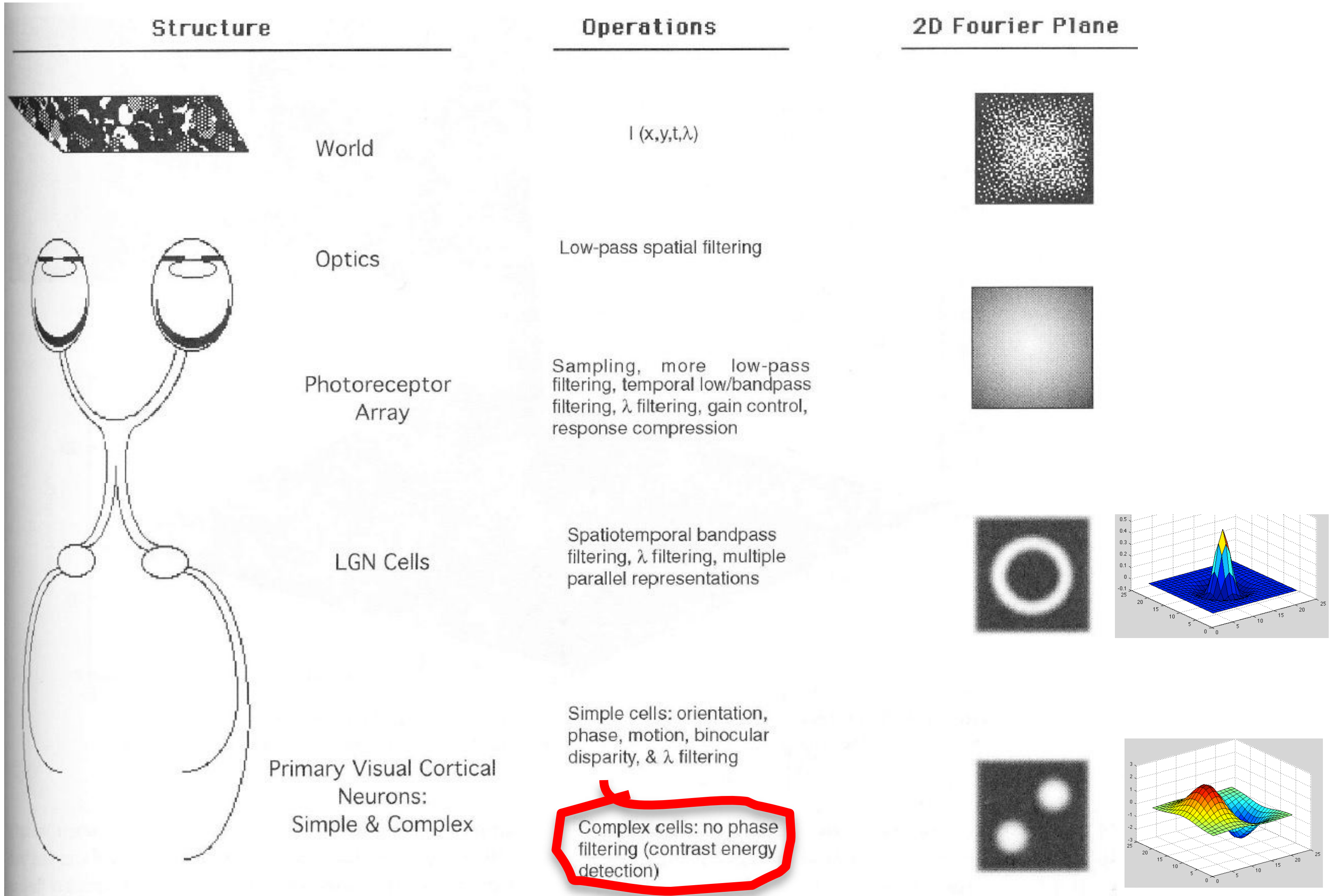


FIGURE 1 Schematic overview of the processing done by the early visual system. On the left, are some of the major structures to be discussed; in the middle, are some of the major operations done at the associated structure; in the right, are the 2-D Fourier representations of the world, retinal image, and sensitivities typical of a ganglion and cortical cell.

John Daugman, 1988

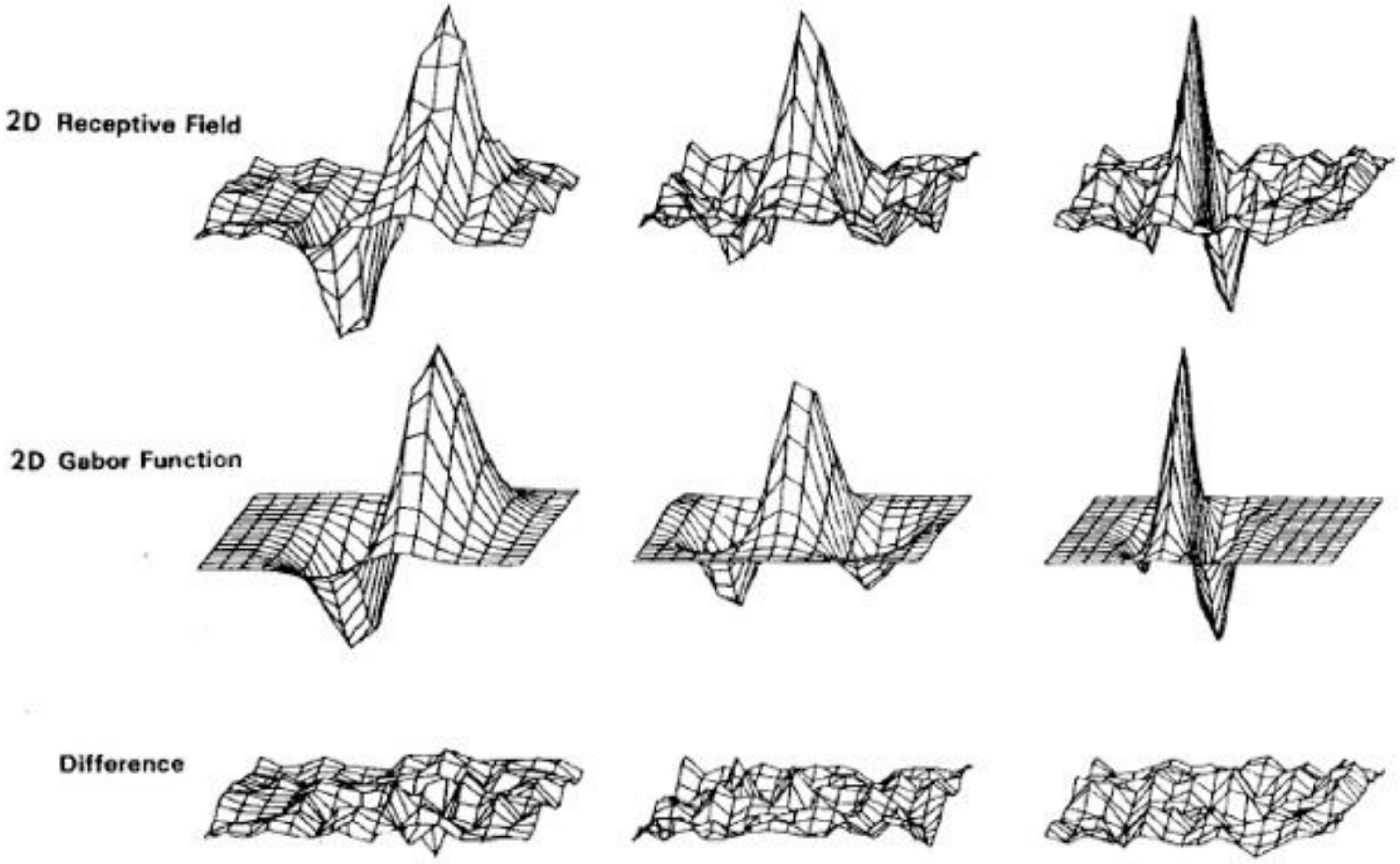
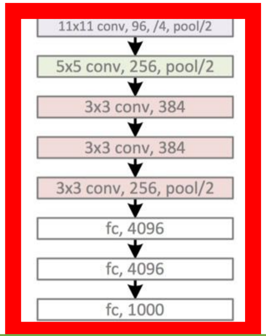


Fig. 5. Top row: illustrations of empirical 2-D receptive field profiles measured by J. P. Jones and L. A. Palmer (personal communication) in simple cells of the cat visual cortex. Middle row: best-fitting 2-D Gabor elementary function for each neuron, described by (10). Bottom row: residual error of the fit, indistinguishable from random error in the Chi-squared sense for 97 percent of the cells studied.

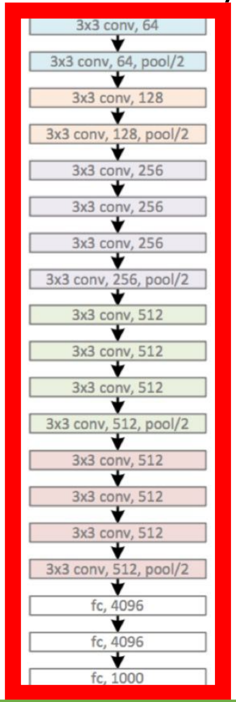
Deep Neural Networks for Visual Recognition

2012: AlexNet
5 conv. layers



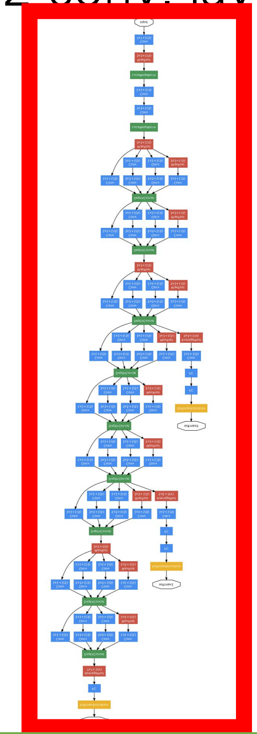
Error: 15.3%

2014: VGG
16 conv. layers

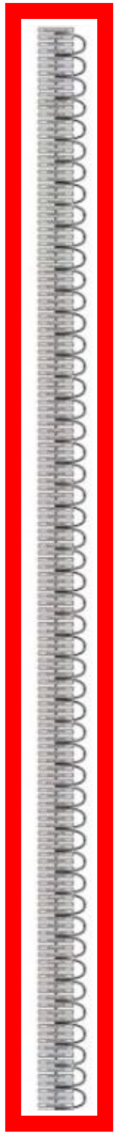


Error: 8.5%

2015: GoogLeNet
22 conv. layers



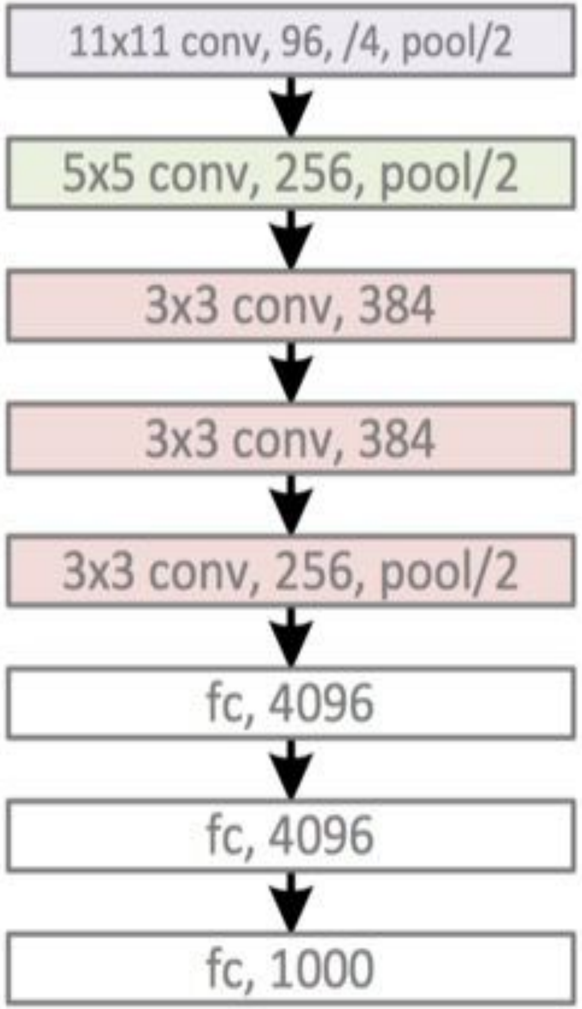
Error: 7.8%



2016: ResNet
>100 conv. layers

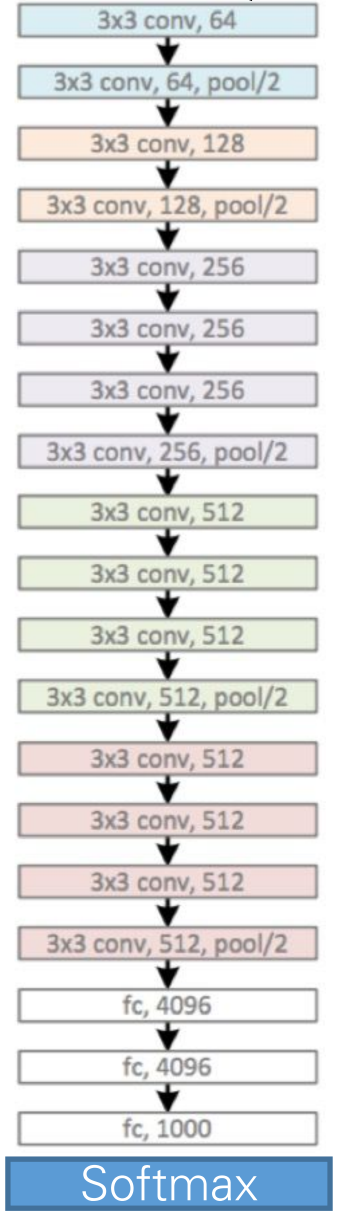
Error: 4.4%

2012: AlexNet
5 conv. layers



Error: 15.3%

2014: VGG
16 conv. layers

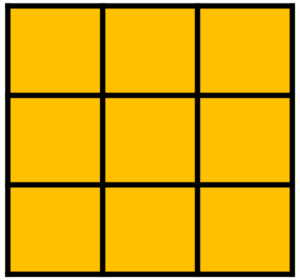


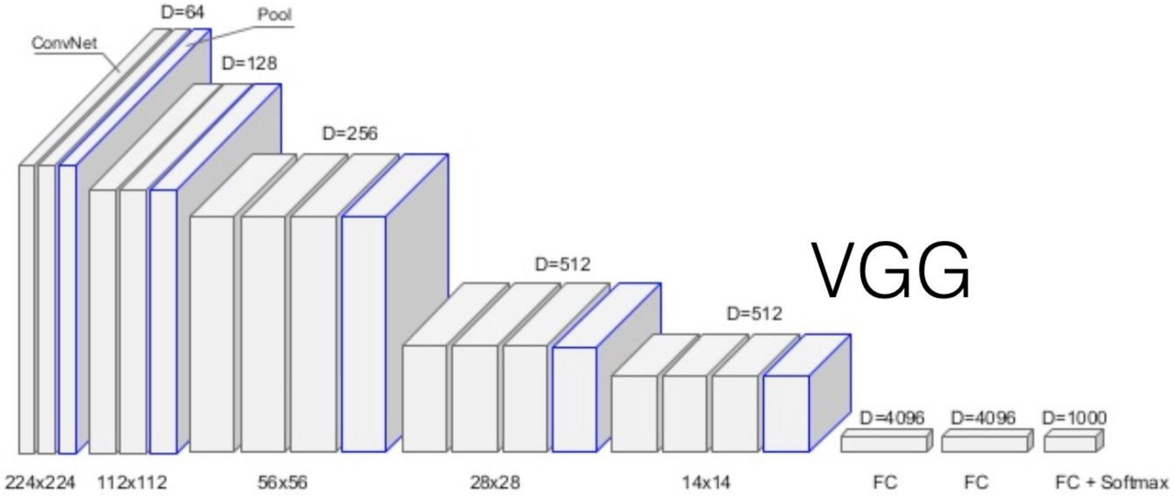
Error: 8.5%

VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION

https://arxiv.org/pdf/1409.1556.pdf

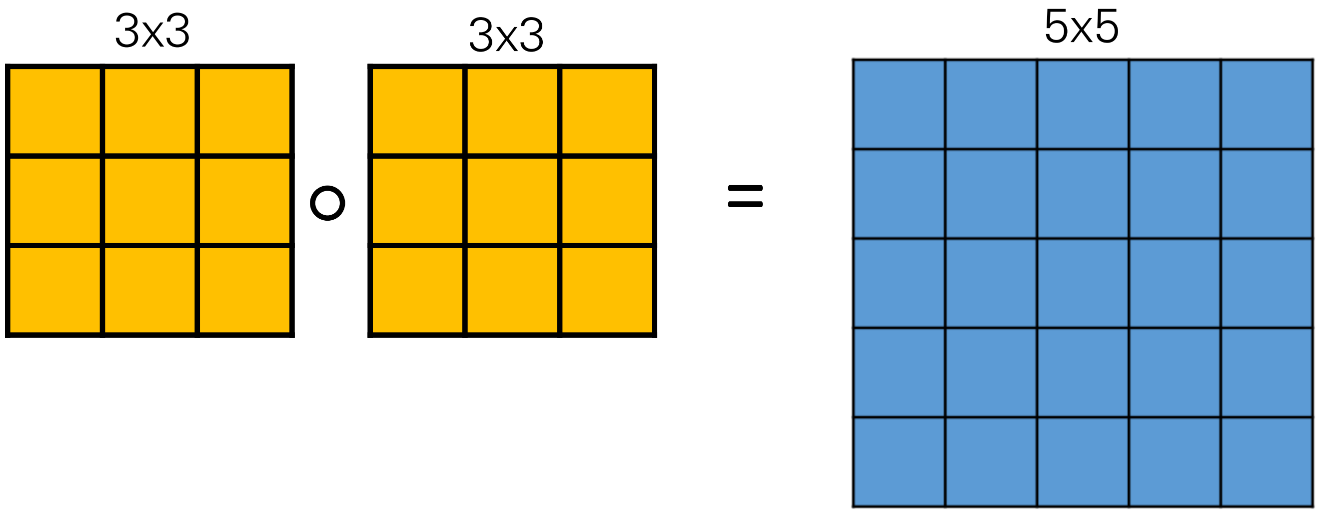
Small convolutional kernels: 3x3
ReLu non-linearities
>100 million parameters.



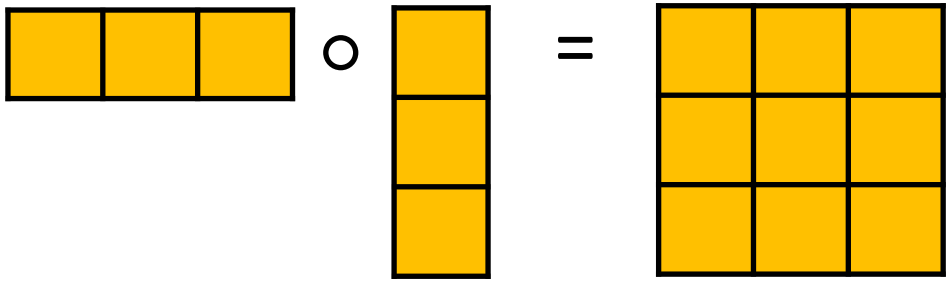


VGG

Chaining convolutions

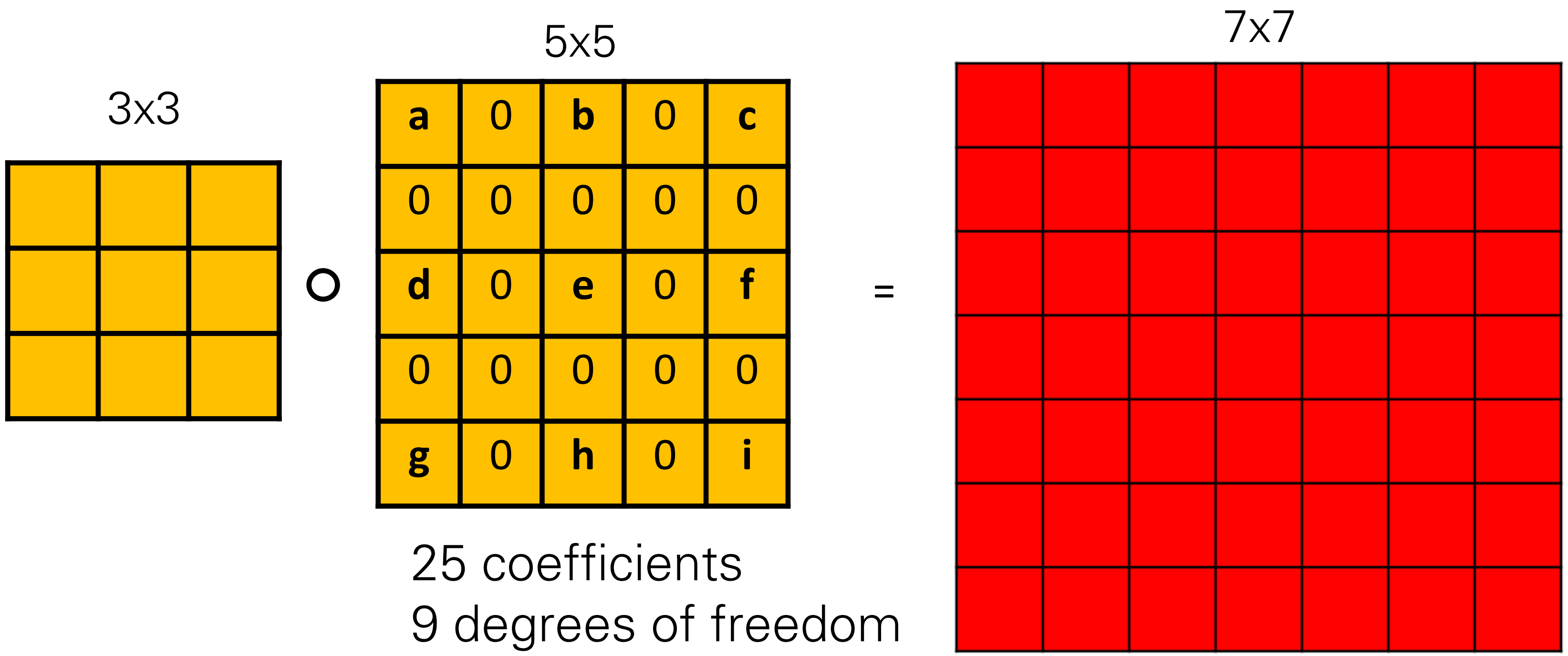


25 coefficients, but only
18 degrees of freedom



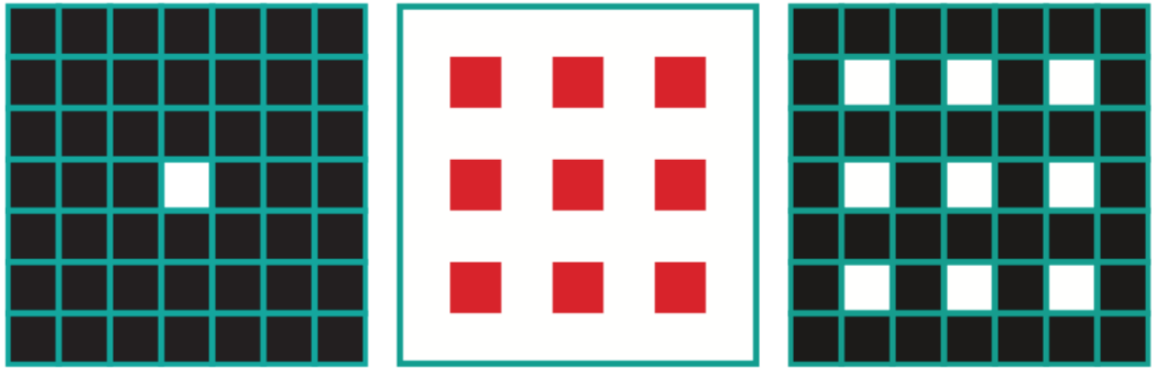
9 coefficients, but only
6 degrees of freedom.
Only separable filters… would this be enough?

Dilated convolutions



25 coefficients
9 degrees of freedom

49 coefficients
18 degrees of freedom



(a) Input (b) Dilation 2 (c) Output

What is lost?

[https://arxiv.org/pdf/1511.07122.pdf]

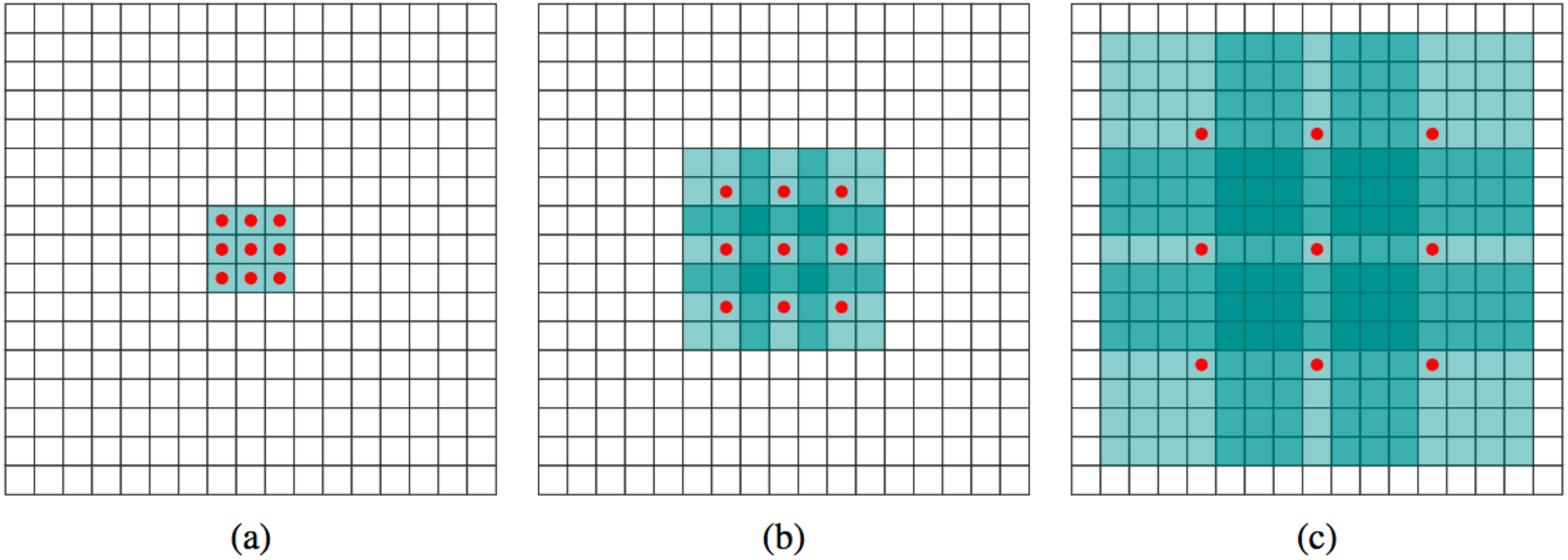


Figure 1: Systematic dilation supports exponential expansion of the receptive field without loss of resolution or coverage. (a) F_1 is produced from F_0 by a 1-dilated convolution; each element in F_1 has a receptive field of 3×3. (b) F_2 is produced from F_1 by a 2-dilated convolution; each element in F_2 has a receptive field of 7×7. (c) F_3 is produced from F_2 by a 4-dilated convolution; each element in F_3 has a receptive field of 15×15. The number of parameters associated with each layer is identical. The receptive field grows exponentially while the number of parameters grows linearly.

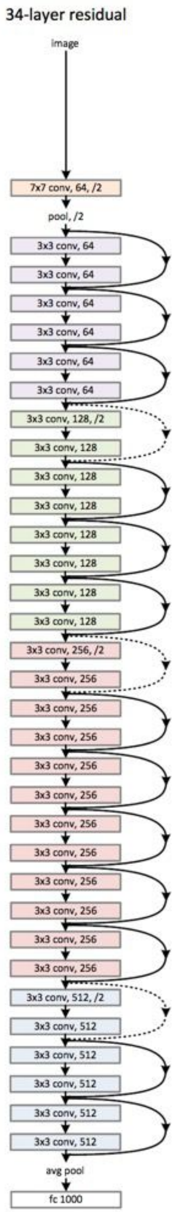
[https://arxiv.org/pdf/1511.07122.pdf]

2016: ResNet
>100 conv. layers

Error: 4.4%

Deep Residual Learning for Image Recognition

https://arxiv.org/pdf/1512.03385.pdf



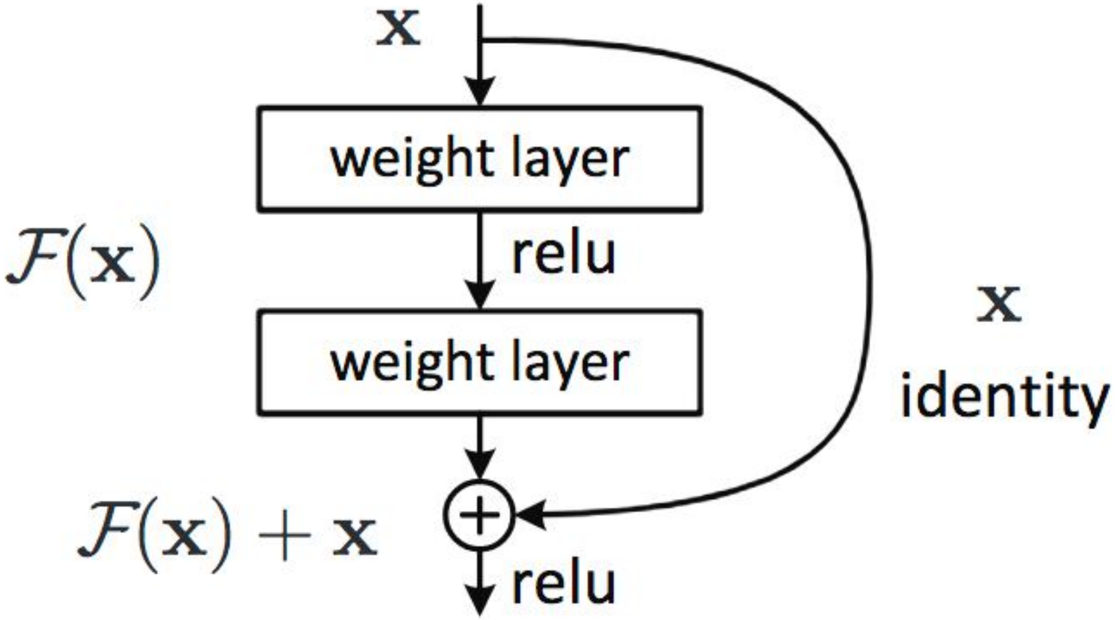
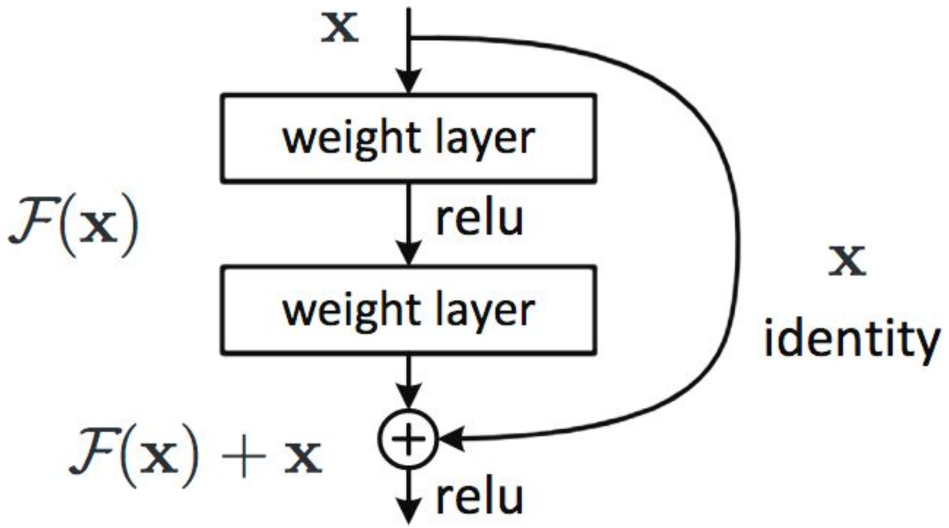
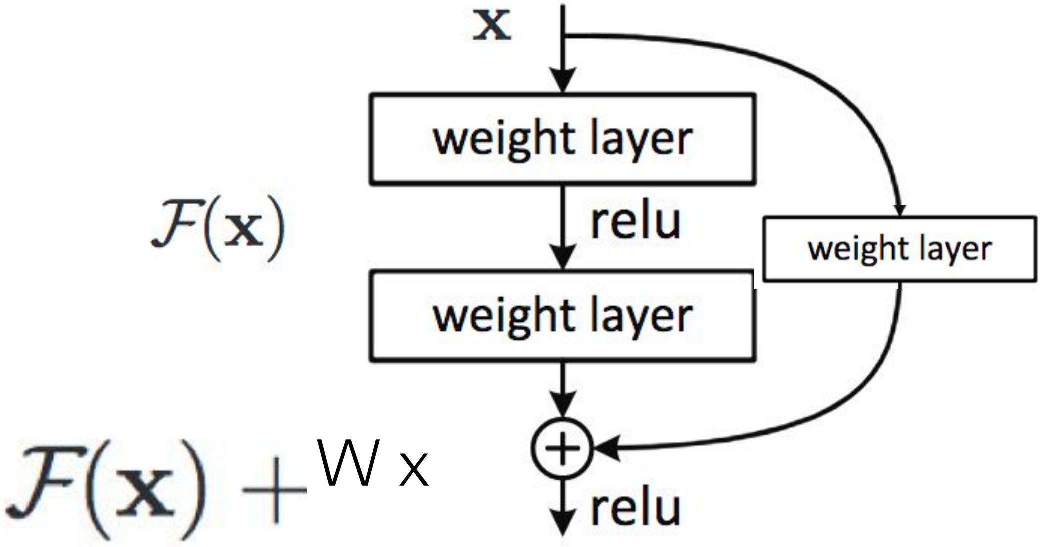


Figure 2. Residual learning: a building block.

If output has same size as input:

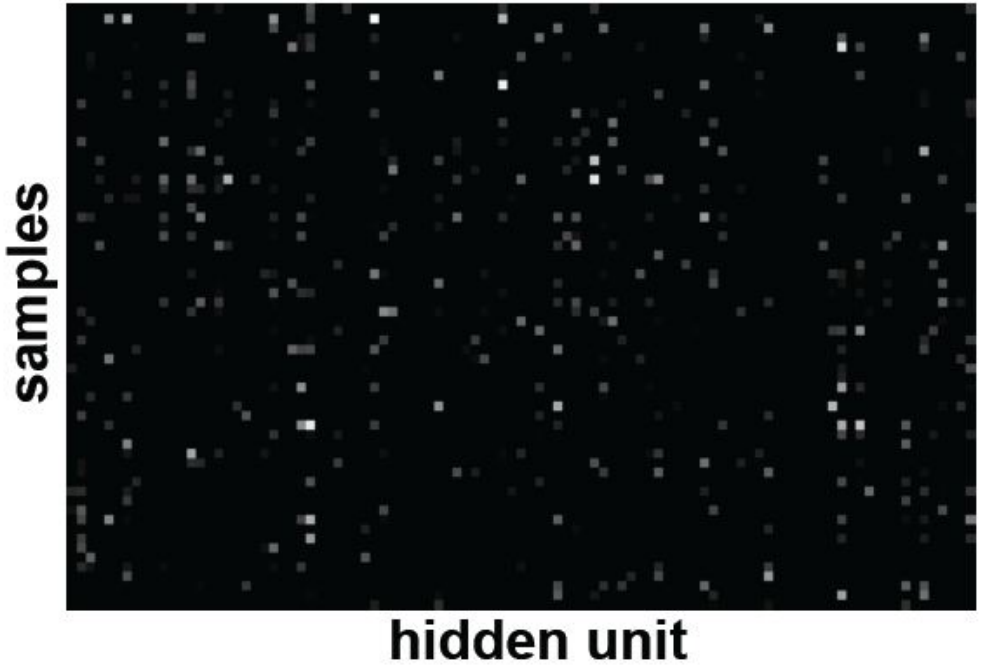


If output has a different size:



Other good things to know

- Check gradients numerically by finite differences
- Visualize hidden activations — should be uncorrelated and high variance



Good training: hidden units are sparse across samples and across features.

[Derived from slide by Marc'Aurelio Ranzato]

Other good things to know

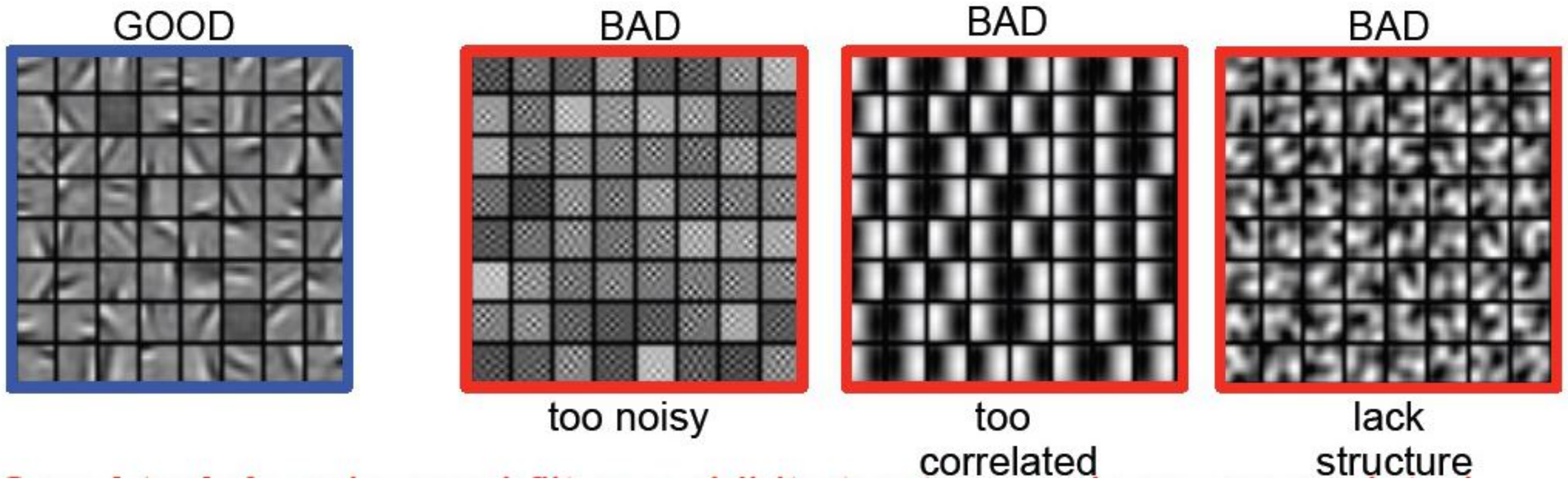
- Check gradients numerically by finite differences
- Visualize hidden activations — should be uncorrelated and high variance

Bad training: many hidden units ignore the input and/or exhibit strong correlations.

[Derived from slide by Marc'Aurelio Ranzato]

Other good things to know

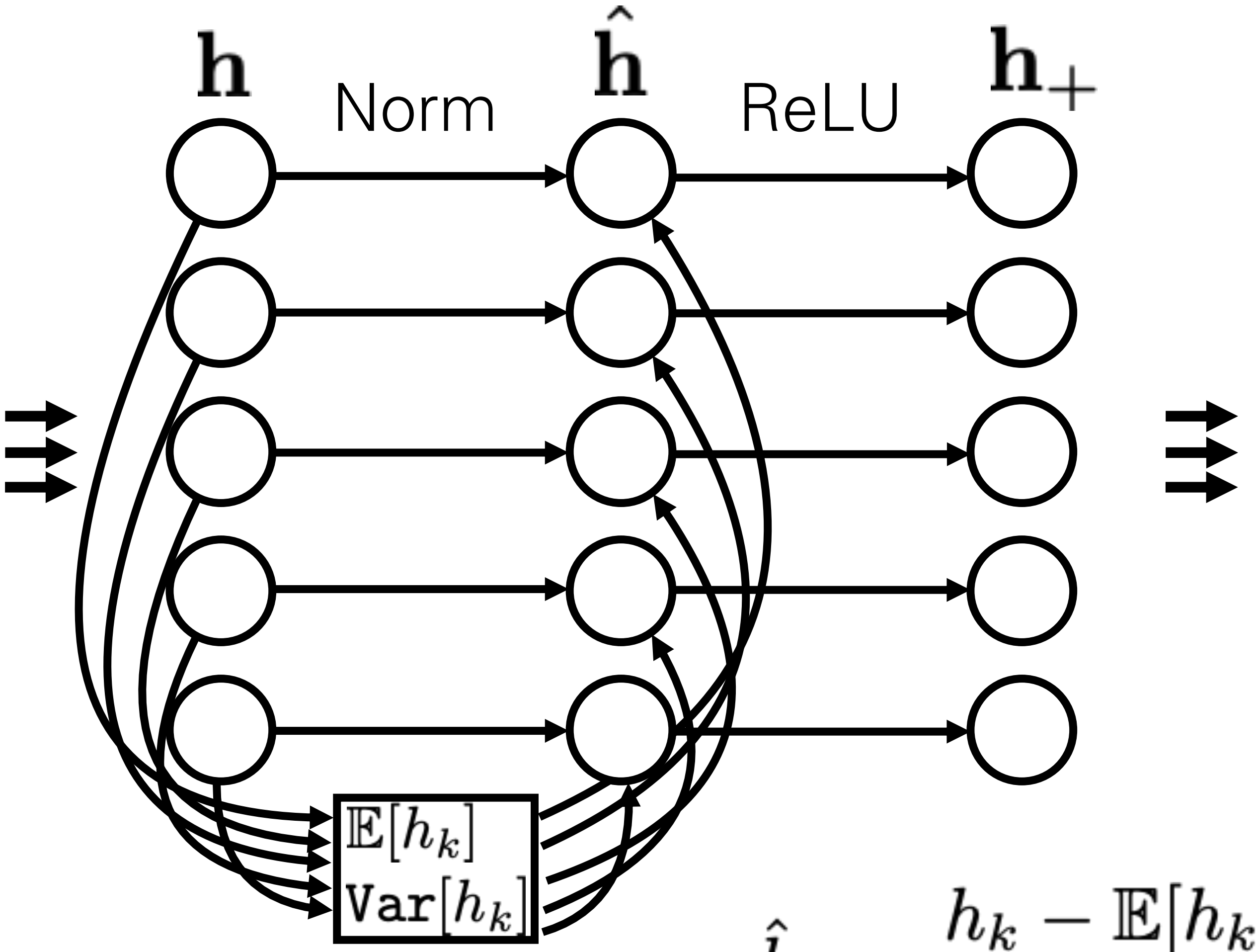
- Check gradients numerically by finite differences
- Visualize hidden activations — should be uncorrelated and high variance
- Visualize filters



Good training: learned filters exhibit structure and are uncorrelated.

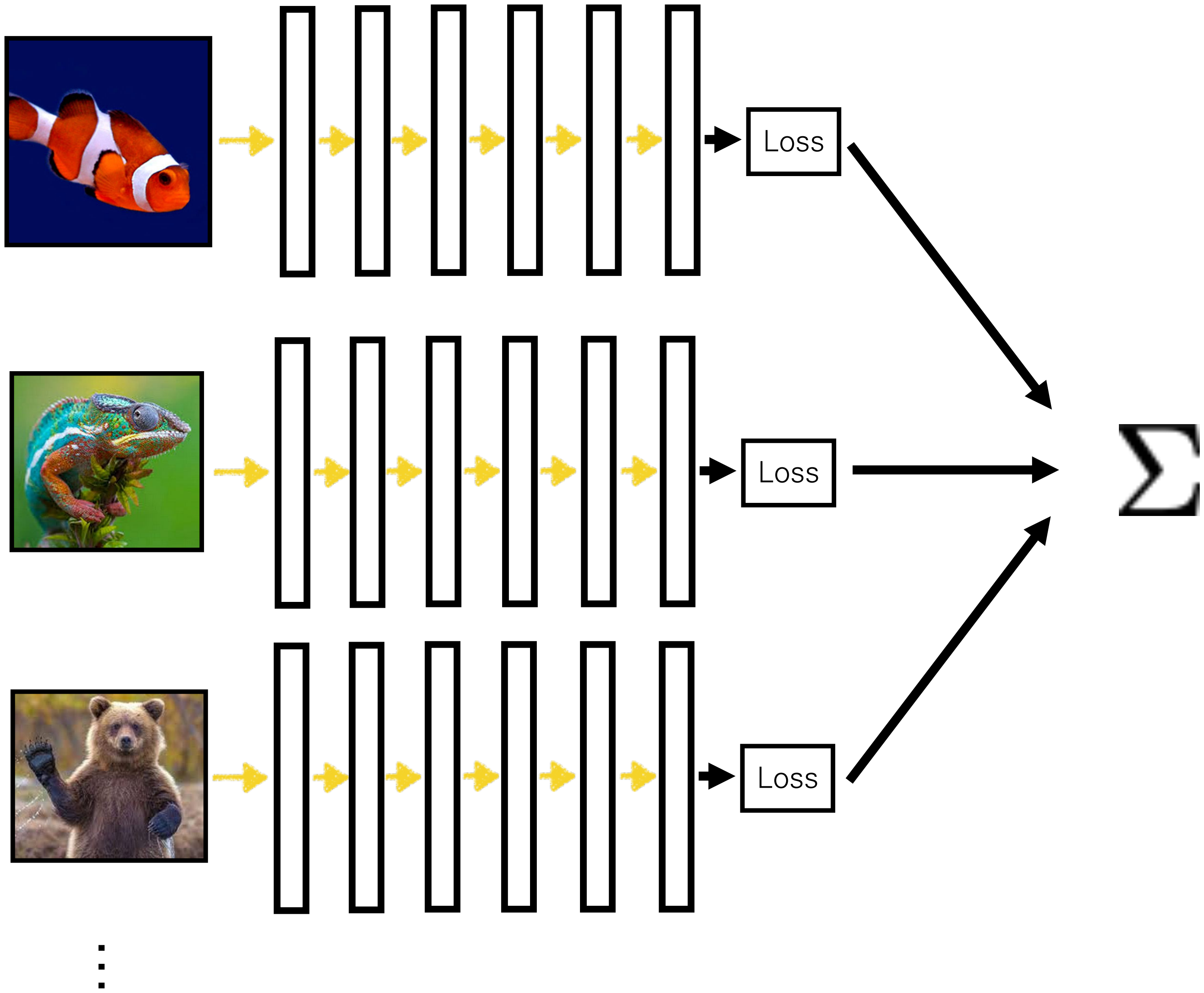
[Derived from slide by Marc'Aurelio Ranzato]

Normalization layers



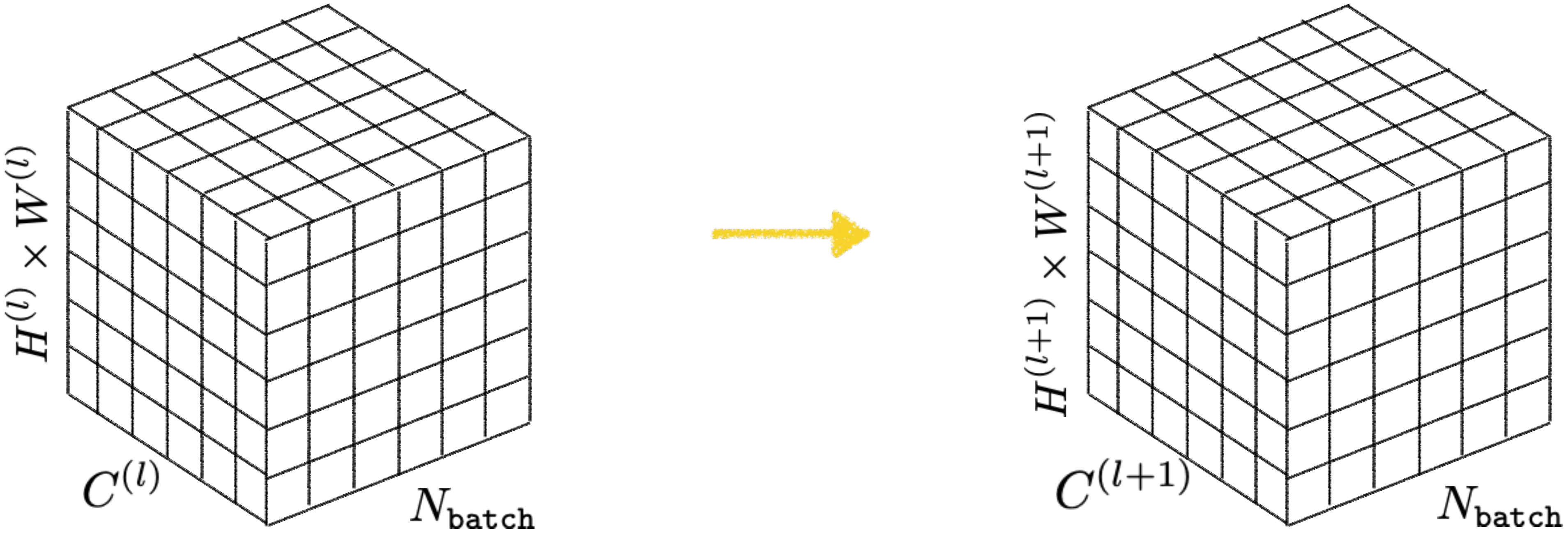
$$\hat{h}_k = \frac{h_k - \mathbb{E}[h_k]}{\sqrt{\mathtt{Var}[h_k]}}$$

Batch processing

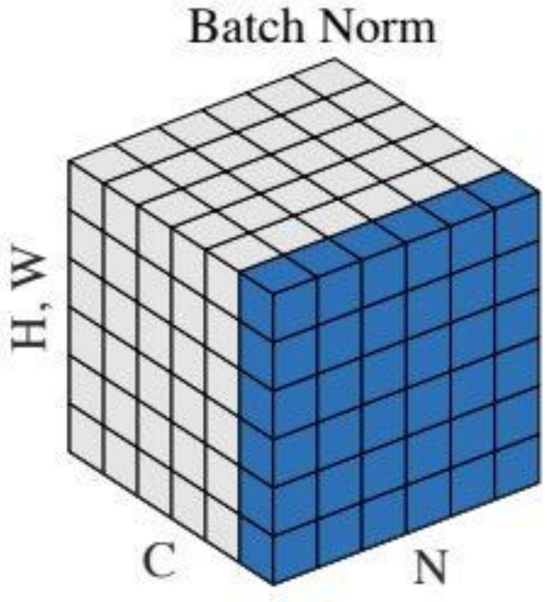


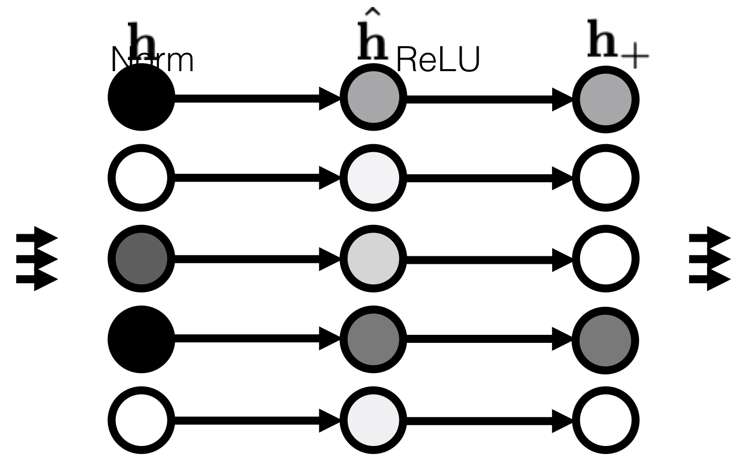
"Tensor flow"

$\mathbf{x}^{(l)} \in \mathbb{R}^{N_{\texttt{batch}} \times H^{(l)} \times W^{(l)} \times C^{(l)}}$ $\mathbf{x}^{(l+1)} \in \mathbb{R}^{N_{\texttt{batch}} \times H^{(l+1)} \times W^{(l+1)} \times C^{(l+1)}}$



Normalization layers

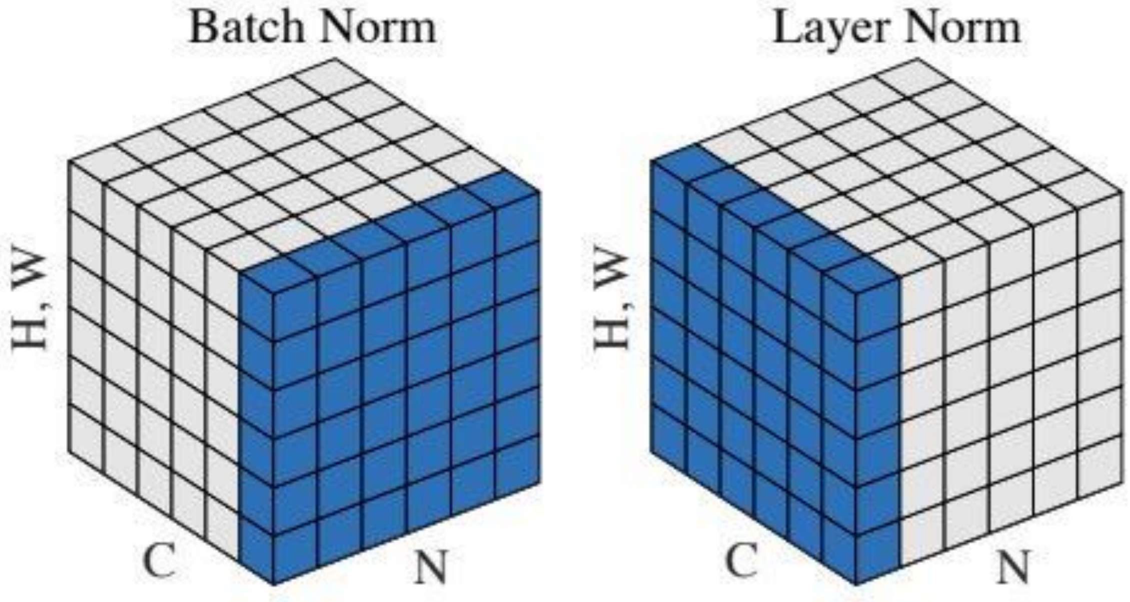




Normalize w.r.t. a single hidden unit's pattern of activation over training examples (a batch of examples).

[Figure from Wu & He, arXiv 2018]

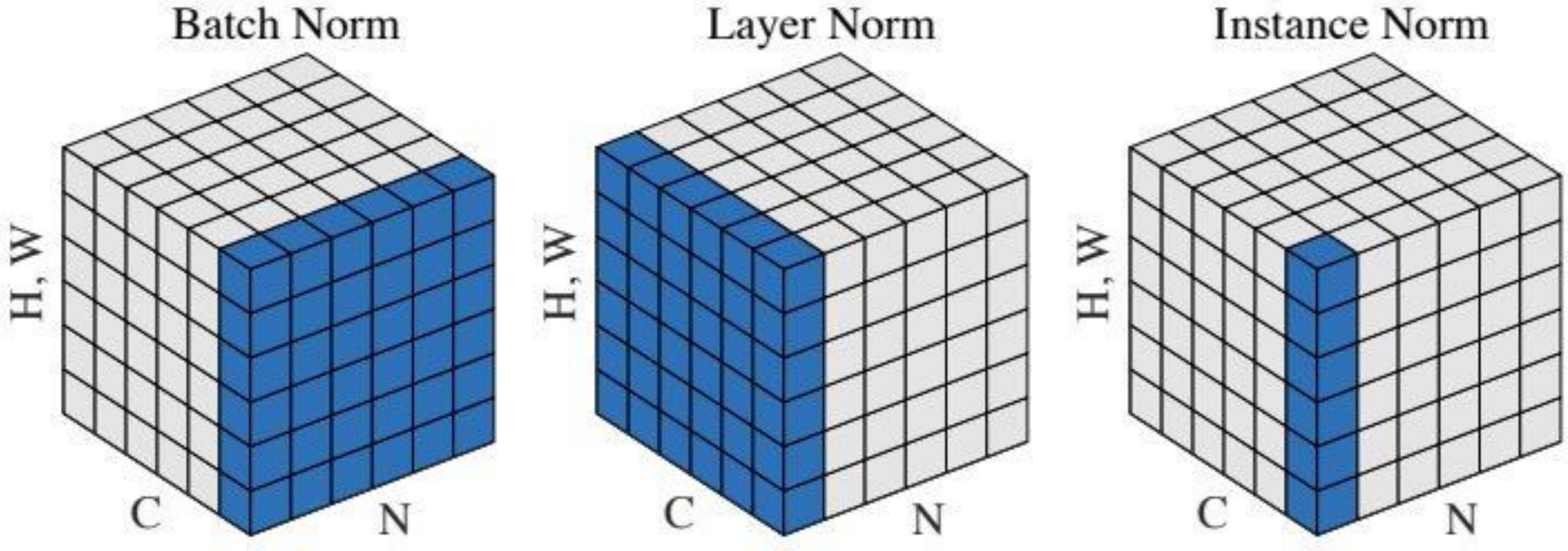
Normalization layers



Normalize w.r.t. the mean and variance of the activations of all the hidden units (neurons) on this layer (c).

[Figure from Wu & He, arXiv 2018]

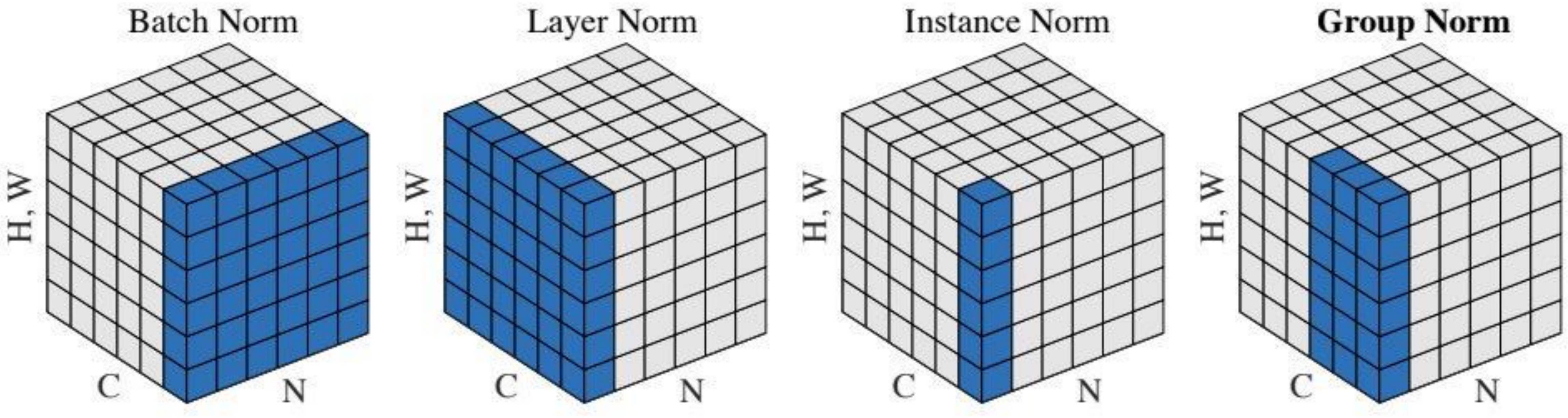
Normalization layers



Normalize w.r.t. the mean and variance of the activations of all the hidden units (neurons) on this layer (c) that process this particular location (h,w) in the image.

[Figure from Wu & He, arXiv 2018]

Normalization layers

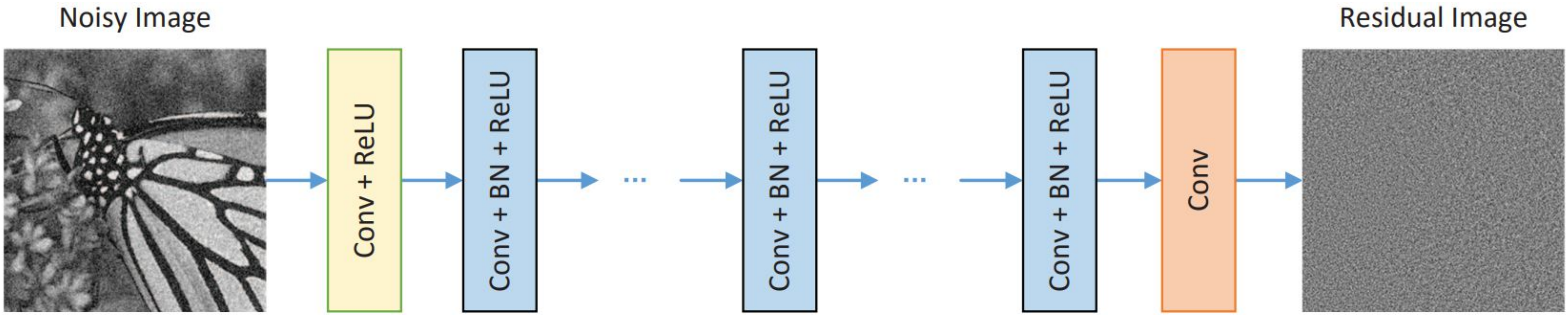


Might as well…

[Figure from Wu & He, arXiv 2018]

Applications in Computational Photography

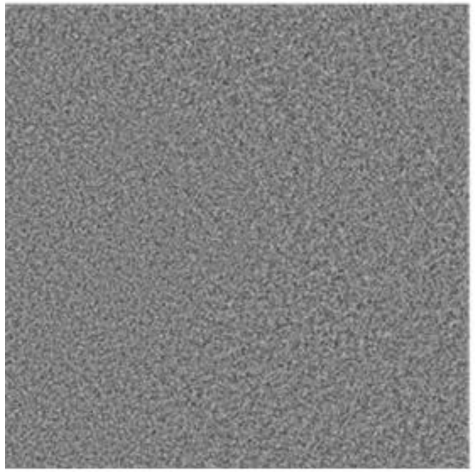
Image Denoising



Key idea: Residual learning

(Zhang et al., IEEE TIP 2017)

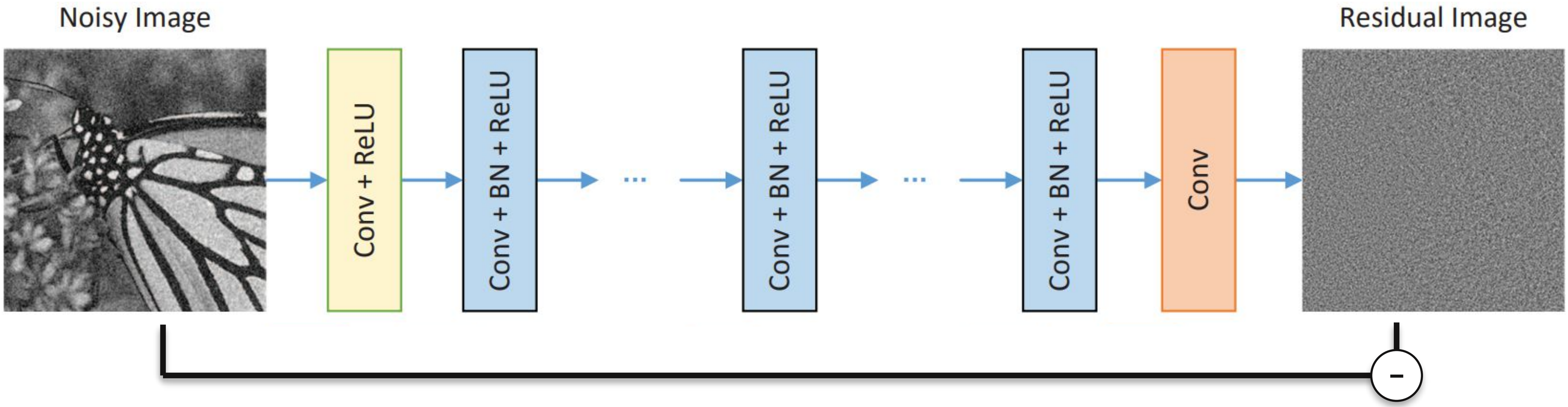
Image Denoising



Clean image = noisy image − estimated noise

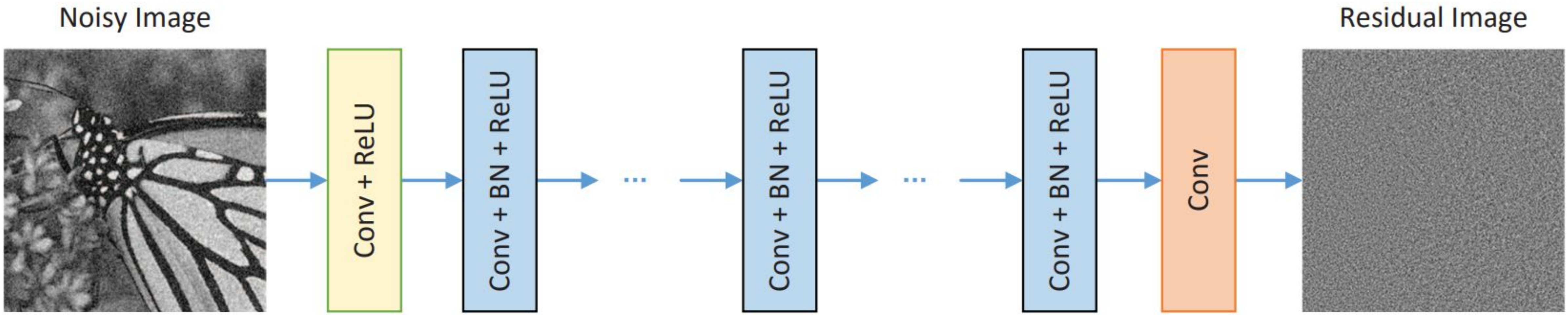
(Zhang et al., IEEE TIP 2017)

Image Denoising



(Zhang et al., IEEE TIP 2017)

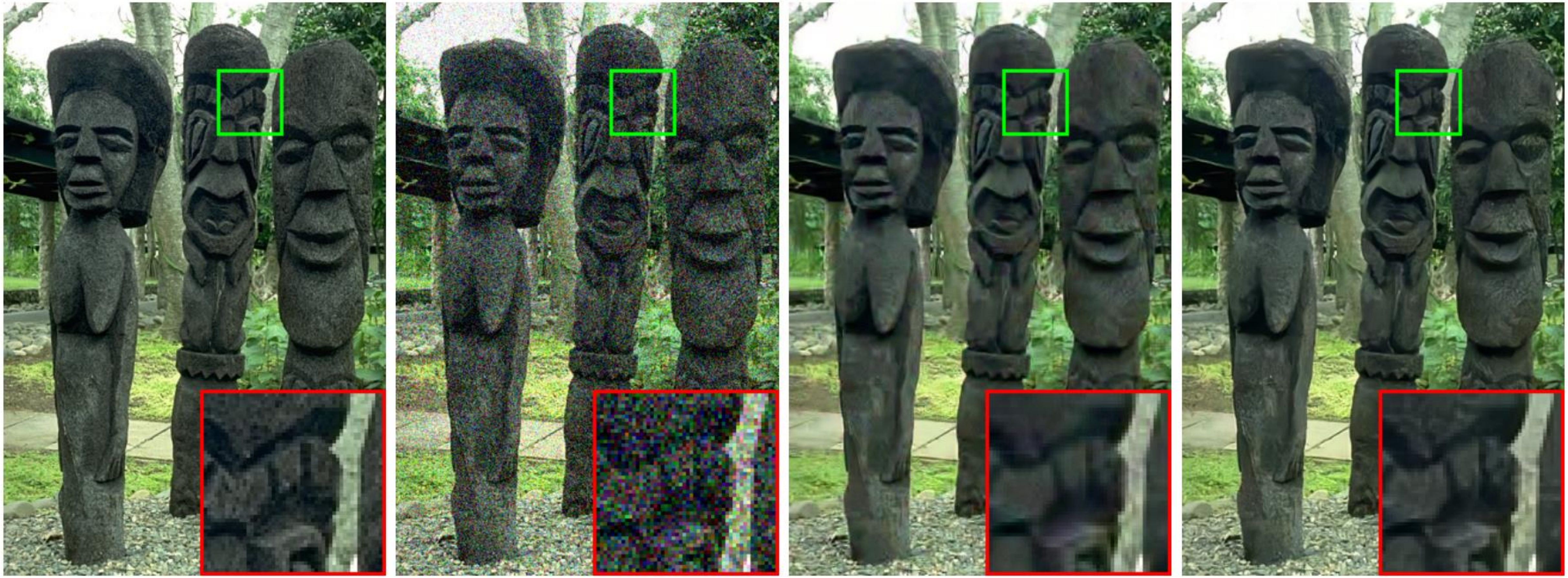
Image Denoising



No fully connected layers - can be applied to any input size

(Zhang et al., IEEE TIP 2017)

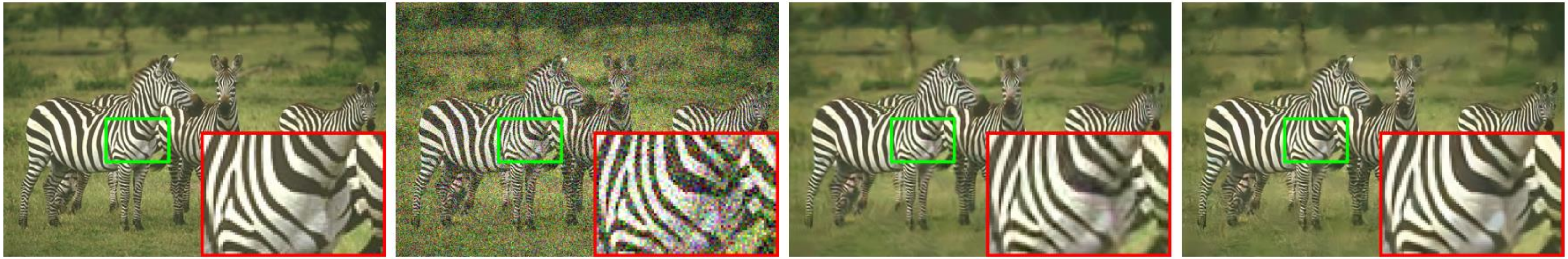
Image Denoising



(a) Ground-truth (b) Noisy / 17.25dB (c) CBM3D / 25.93dB (d) CDnCNN-B / 26.58dB

(Zhang et al., IEEE TIP 2017)

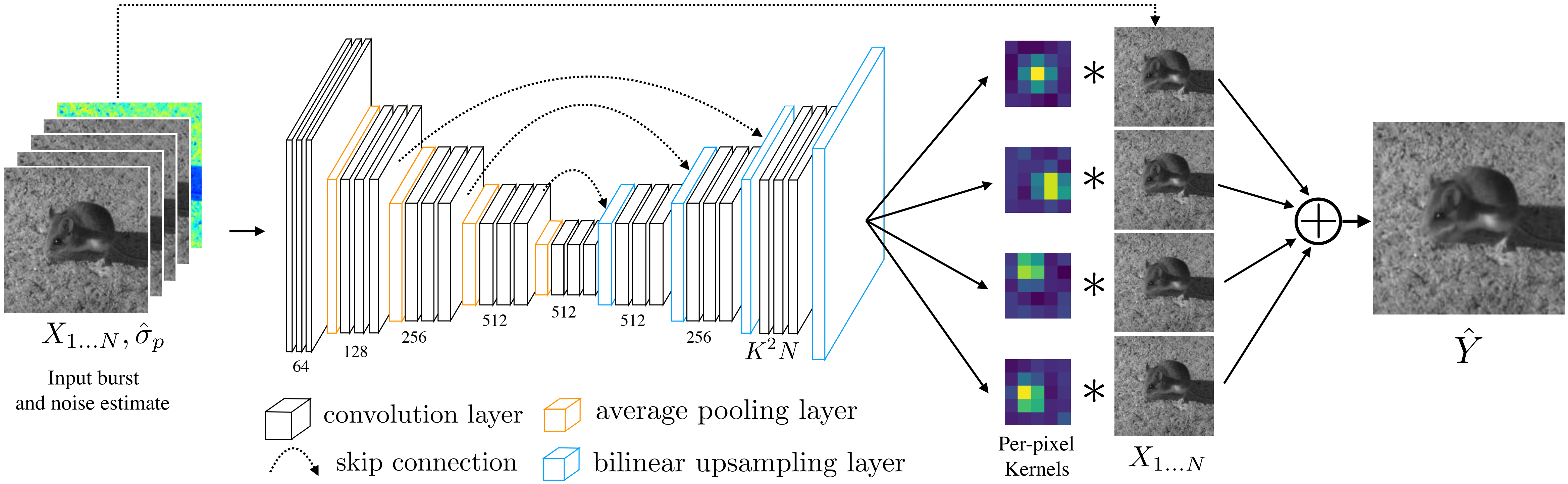
Image Denoising



(a) Ground-truth (b) Noisy / 15.07dB (c) CBM3D / 26.97dB (d) CDnCNN-B / 27.87dB

(Zhang et al., IEEE TIP 2017)

Image Denoising

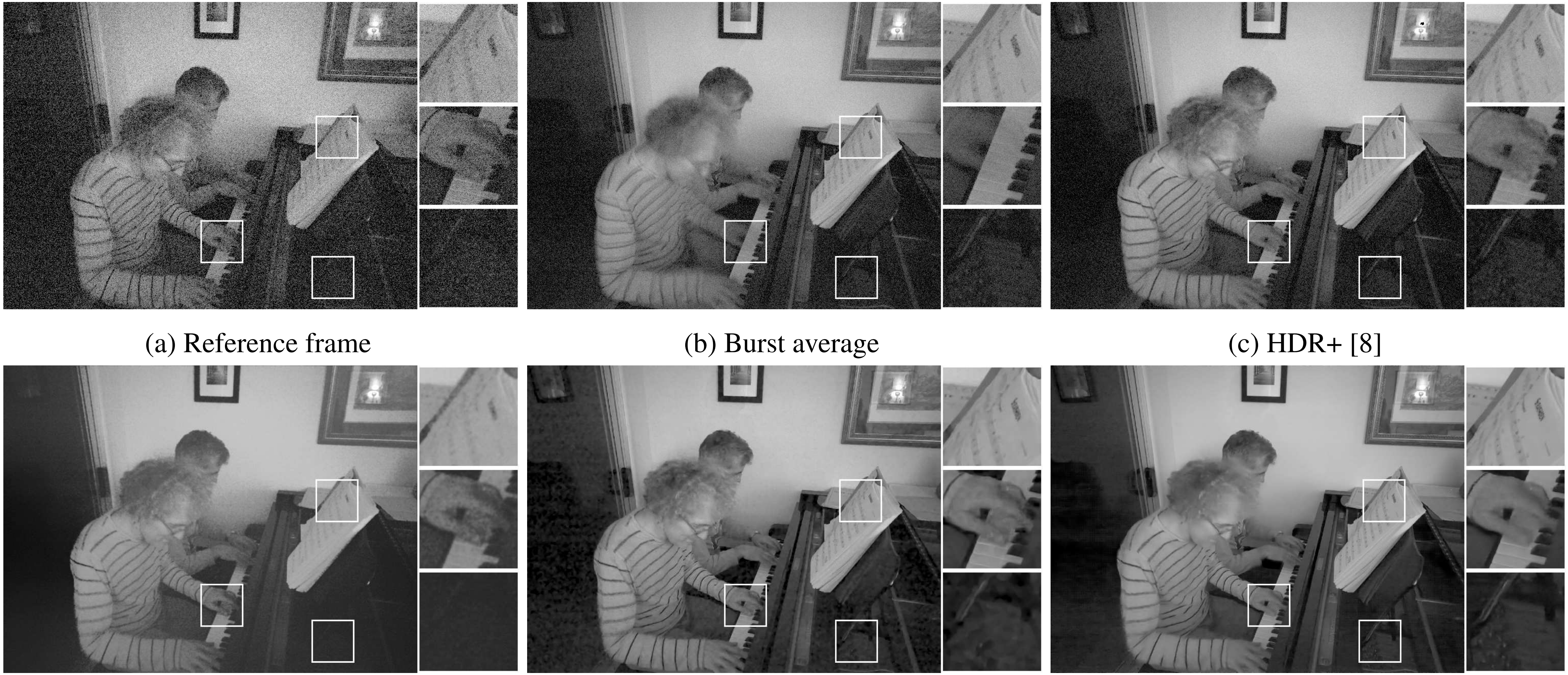


Key ideas:

- Perform denoising in RAW domain considering a burst of images
- Generates a stack of per-pixel filter kernels that jointly aligns, averages, and denoises a burst of images.

(Mildenhall et al., CVPR 2018

Image Denoising

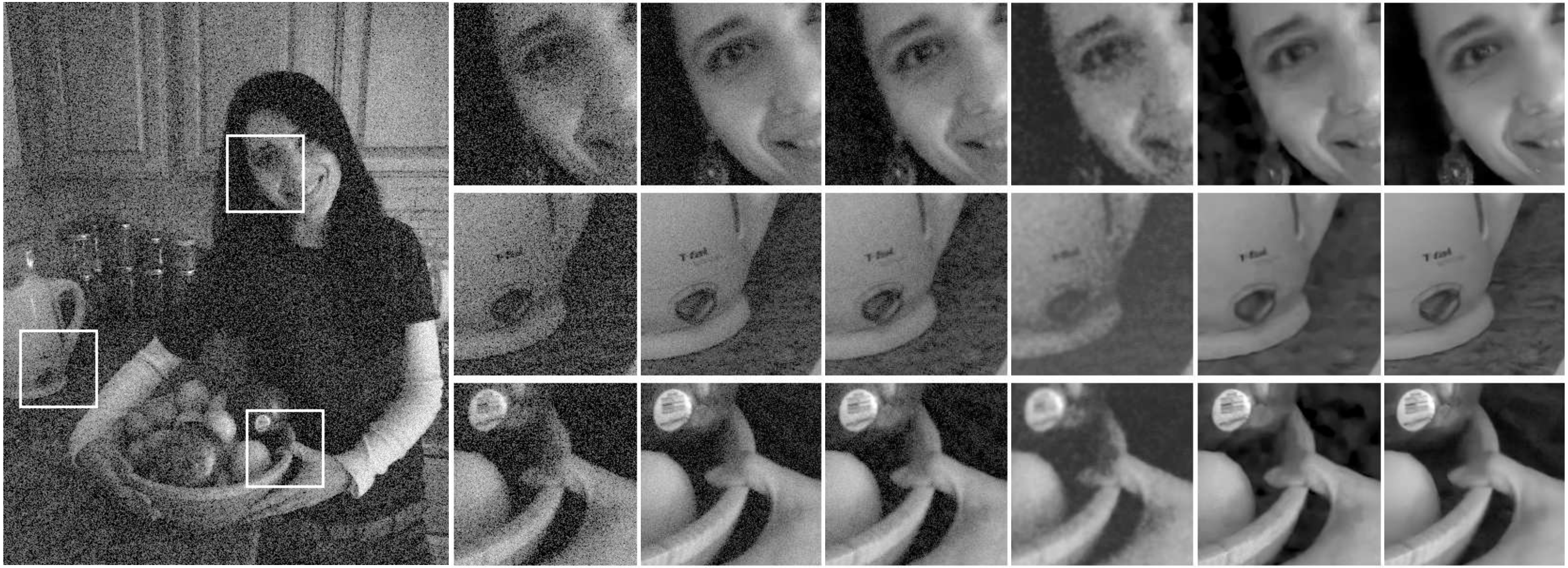


(a) Reference frame (b) Burst average (c) HDR+ [8]

(d) Non-local means [3] (e) VBM4D [17] (f) Our KPN model

(Mildenhall et al., CVPR 2018

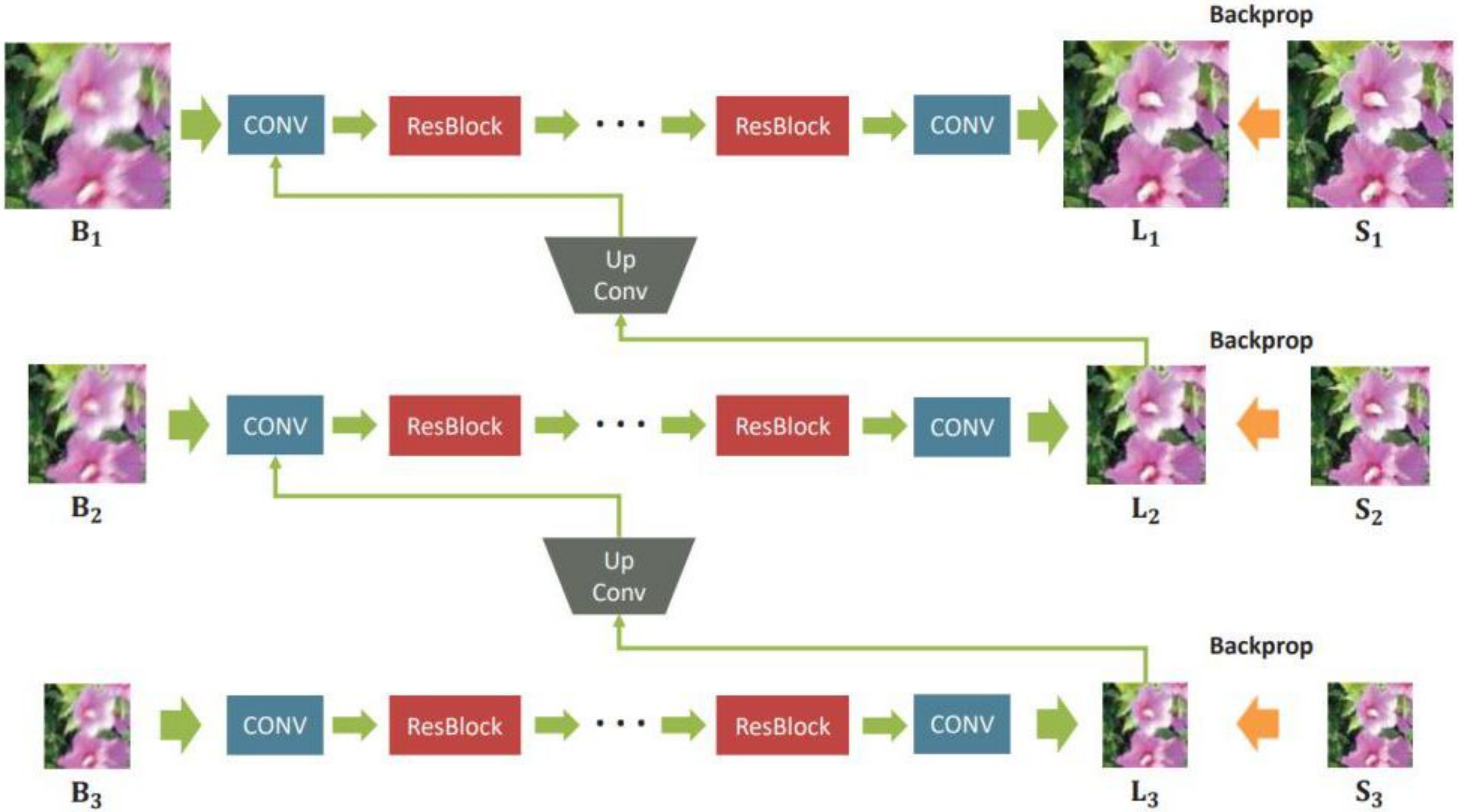
Image Denoising



Reference frame (a) Reference (b) Average (c) HDR+ (d) NLM (e) VBM4D (f) Ours (KPN)

(Mildenhall et al., CVPR 2018

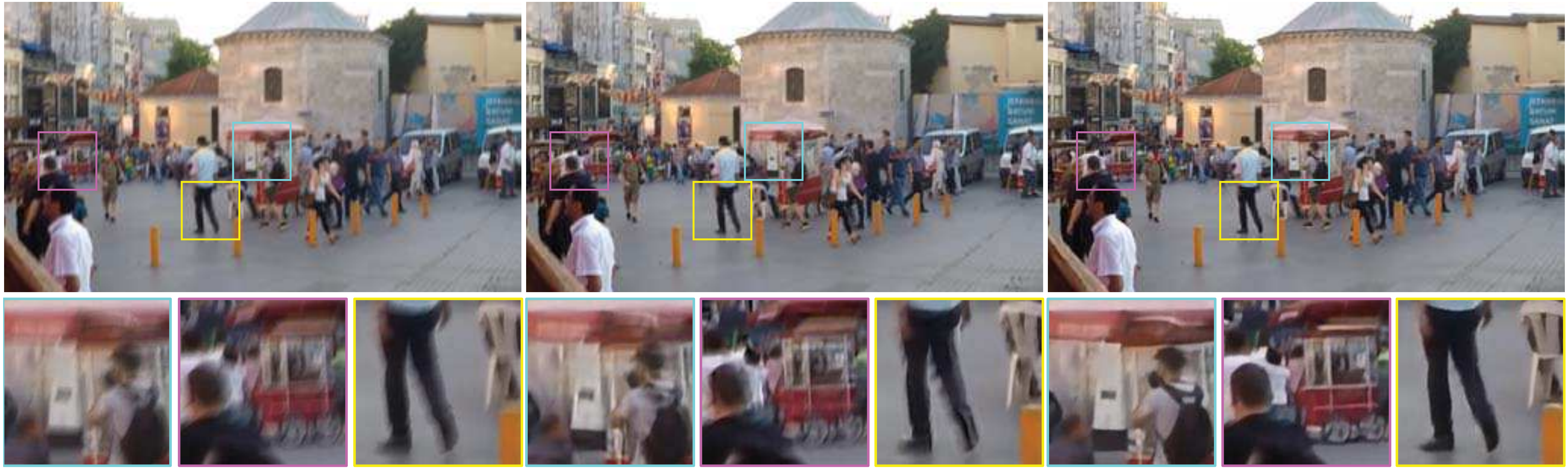
Image Deblurring



Key idea: Multi-scale processing, i.e., use image pyramid to process and deblur

(Nah et al., CVPR 2017)

Image Deblurring



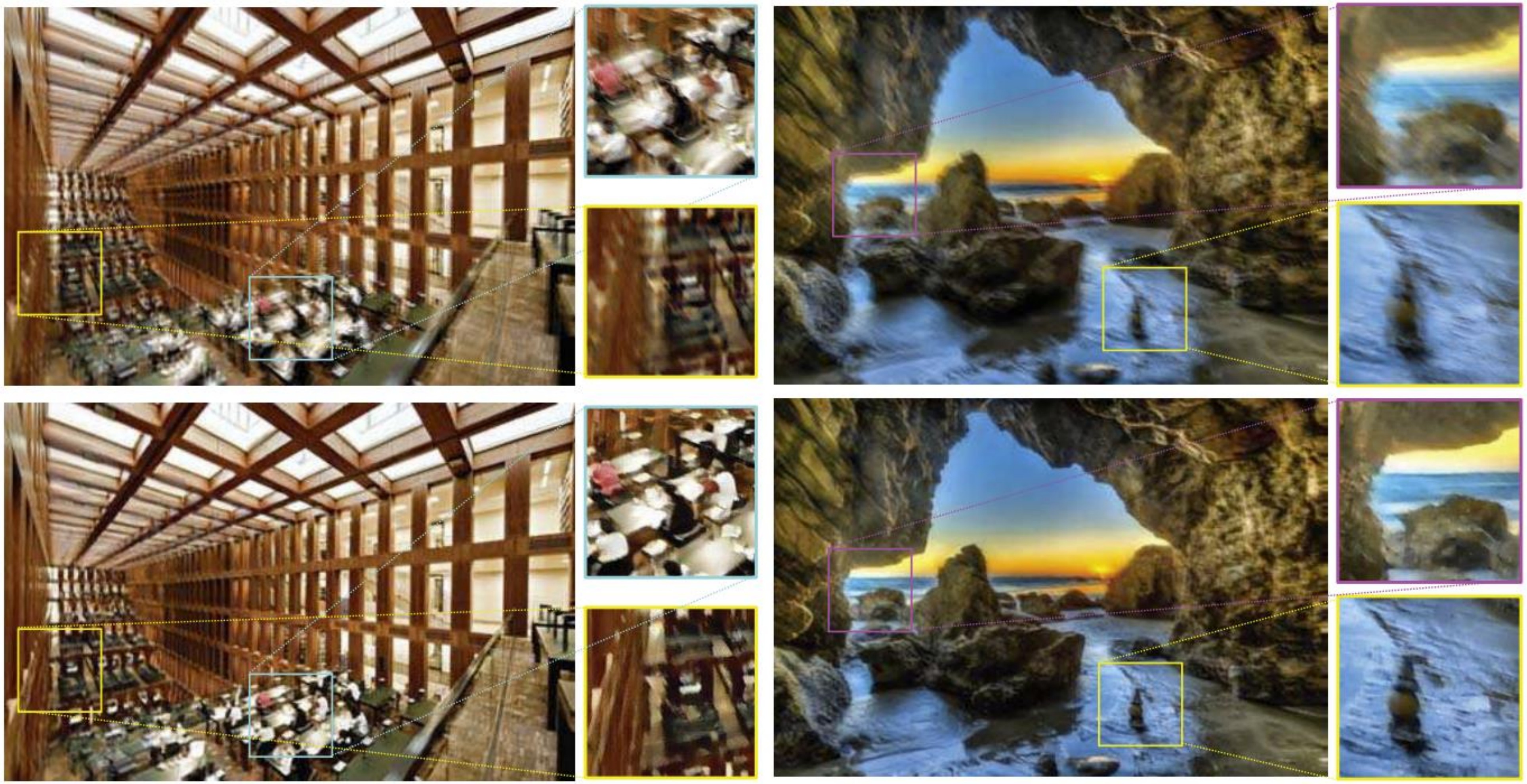
Input Single-scale Multi-scale

(Nah et al., CVPR 2017)

Image Deblurring

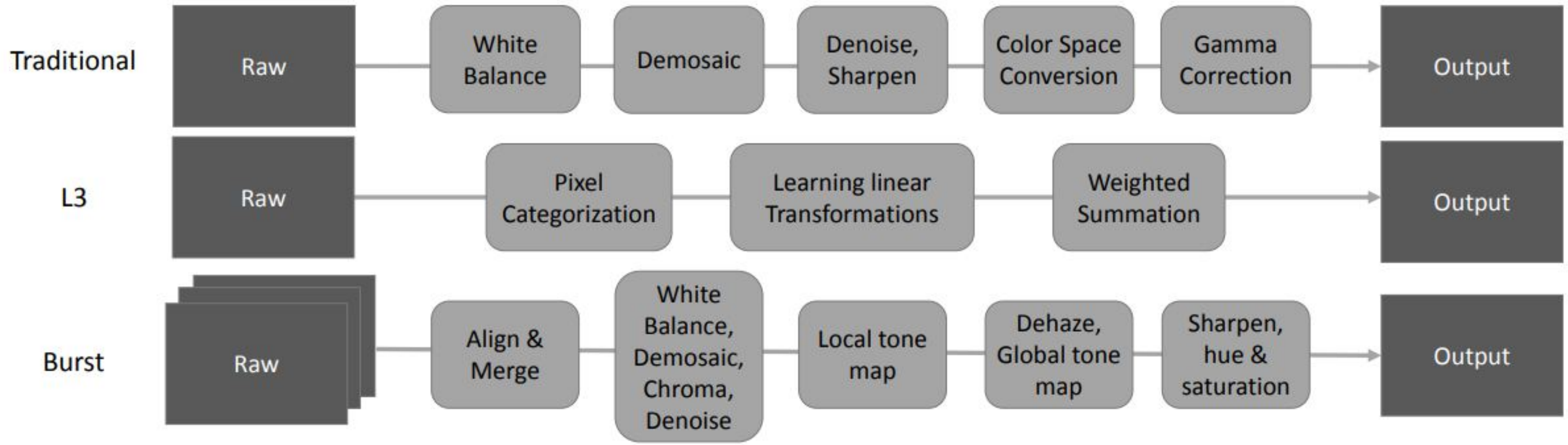
Single-scale

Multi-scale

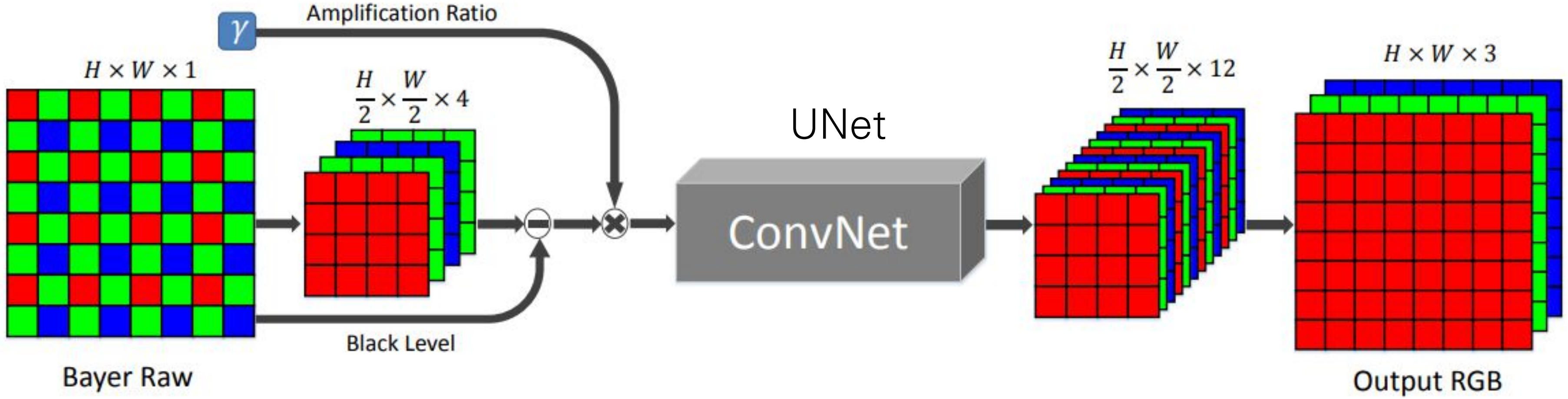


(Nah et al., CVPR 2017)

Learned ISP



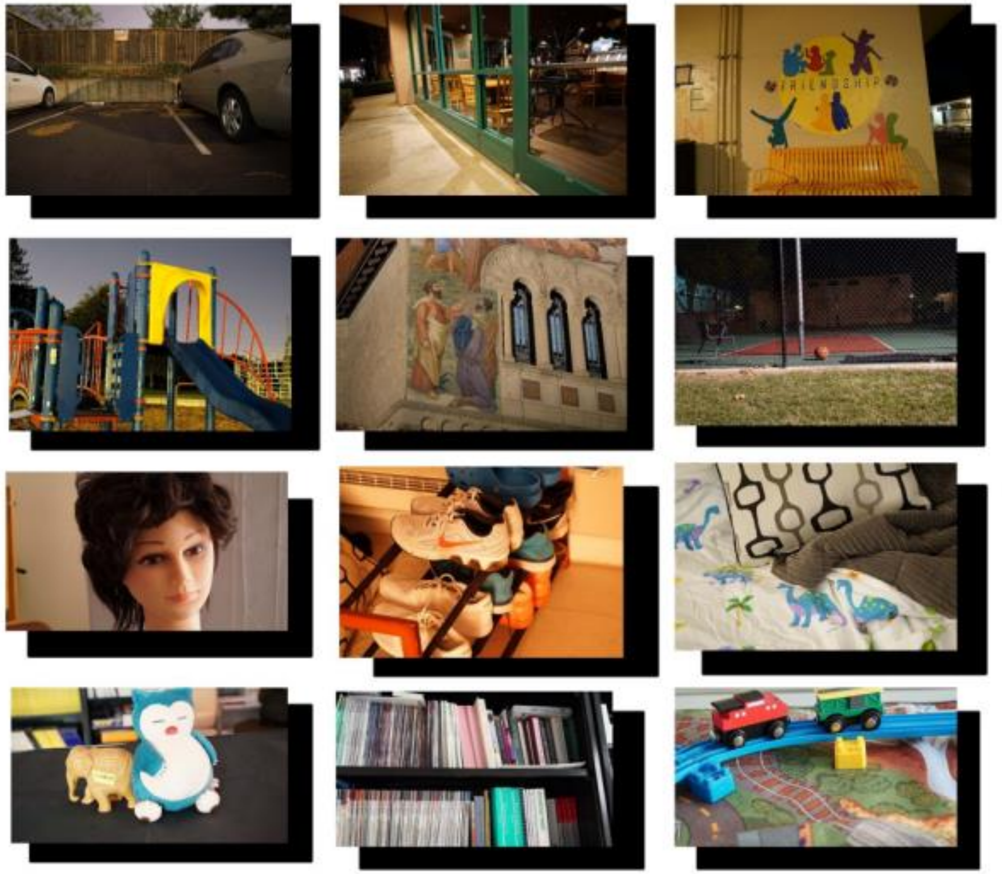
(a)

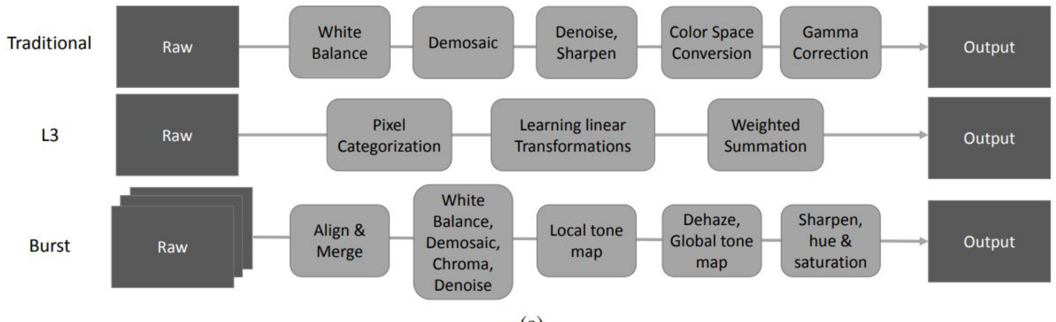


Key idea: Learn a convnet to enhance extremely low-light images

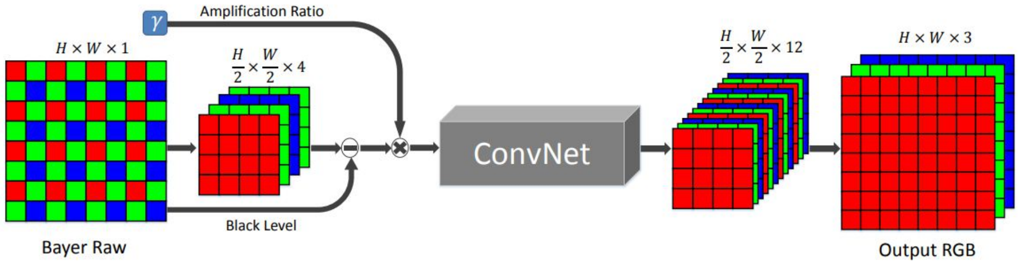
(Chen et al., CVPR 2018)

Learned ISP





(a)



Trained on short-exposure (noisy) / long-exposure image pairs

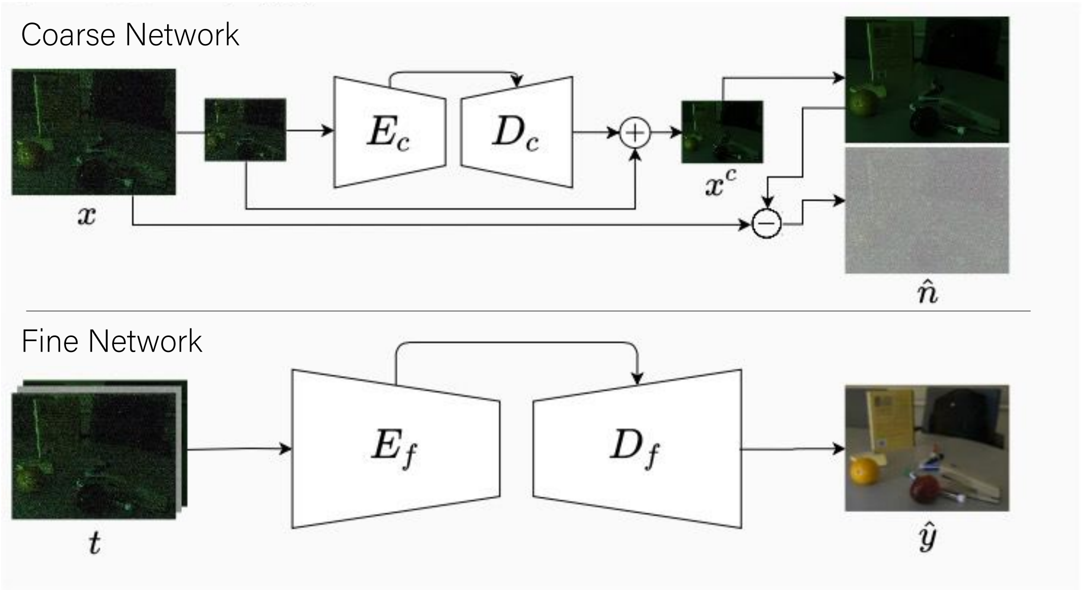
(Chen et al., CVPR 2018)

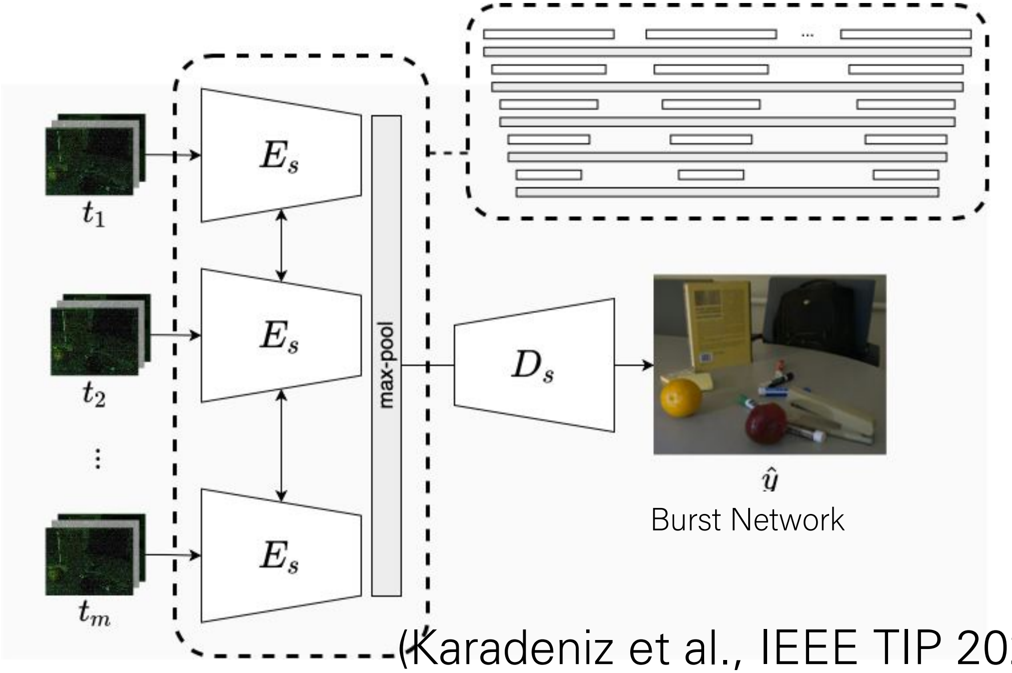
Learned ISP

- Key ideas:

1. A frame-level enhancement network that works in a coarse-to-fine manner

2. Extension of this model to a burst of dark images





(Karadeniz et al., IEEE TIP 2021)


Learned ISP
Noisy input

Learned ISP


Karadeniz et al. (single)
Chen et al., 2018
Guide
Wireless Communications and Networking
LINEAR DYNAMIC SYSTEMS AND SIGNALS

Learned ISP

Traditional pipeline

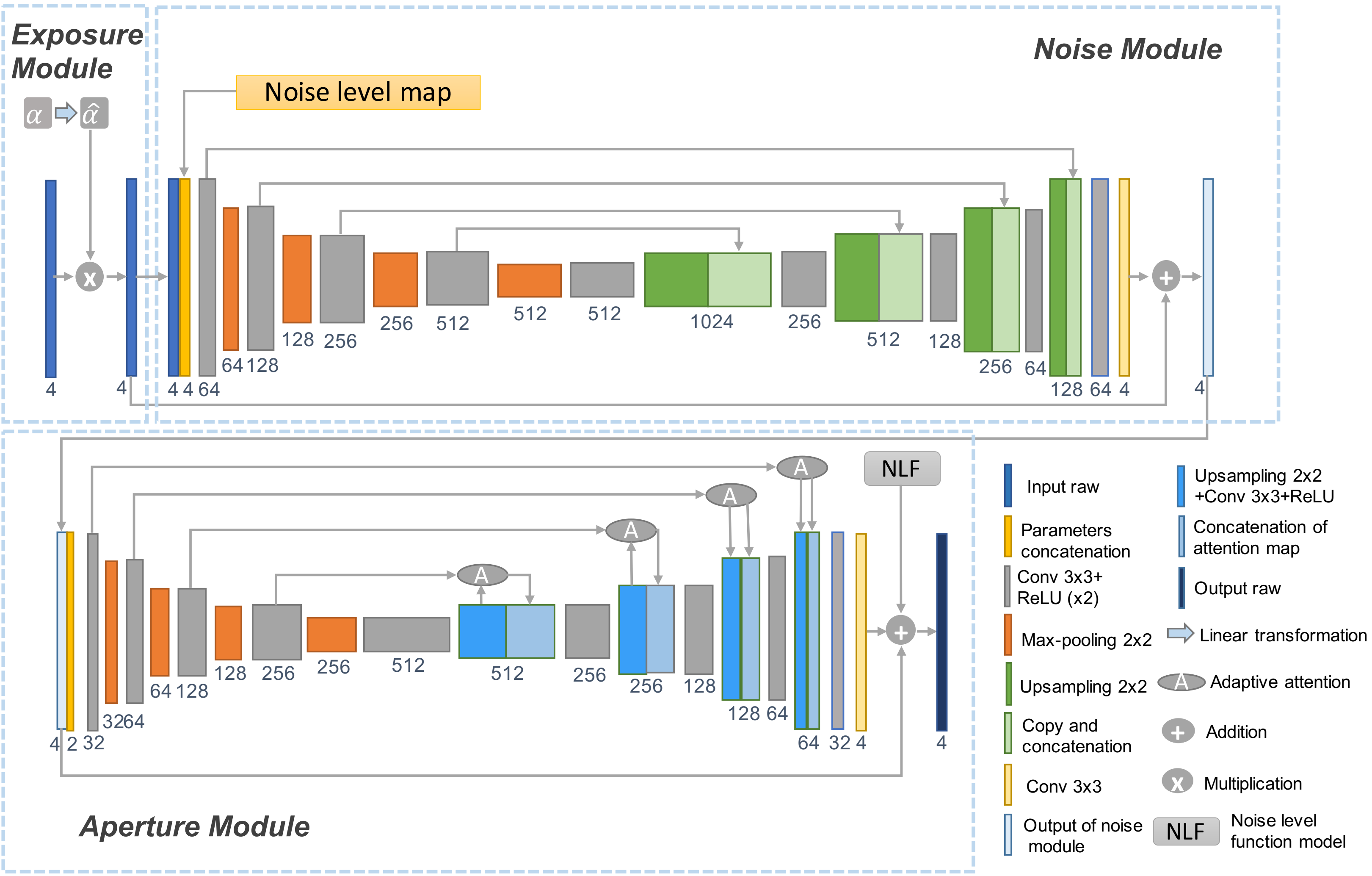

Learned ISP
Traditional pipeline + Scaling

Learned ISP

Learned ISP

Learned ISP

Learned ISP



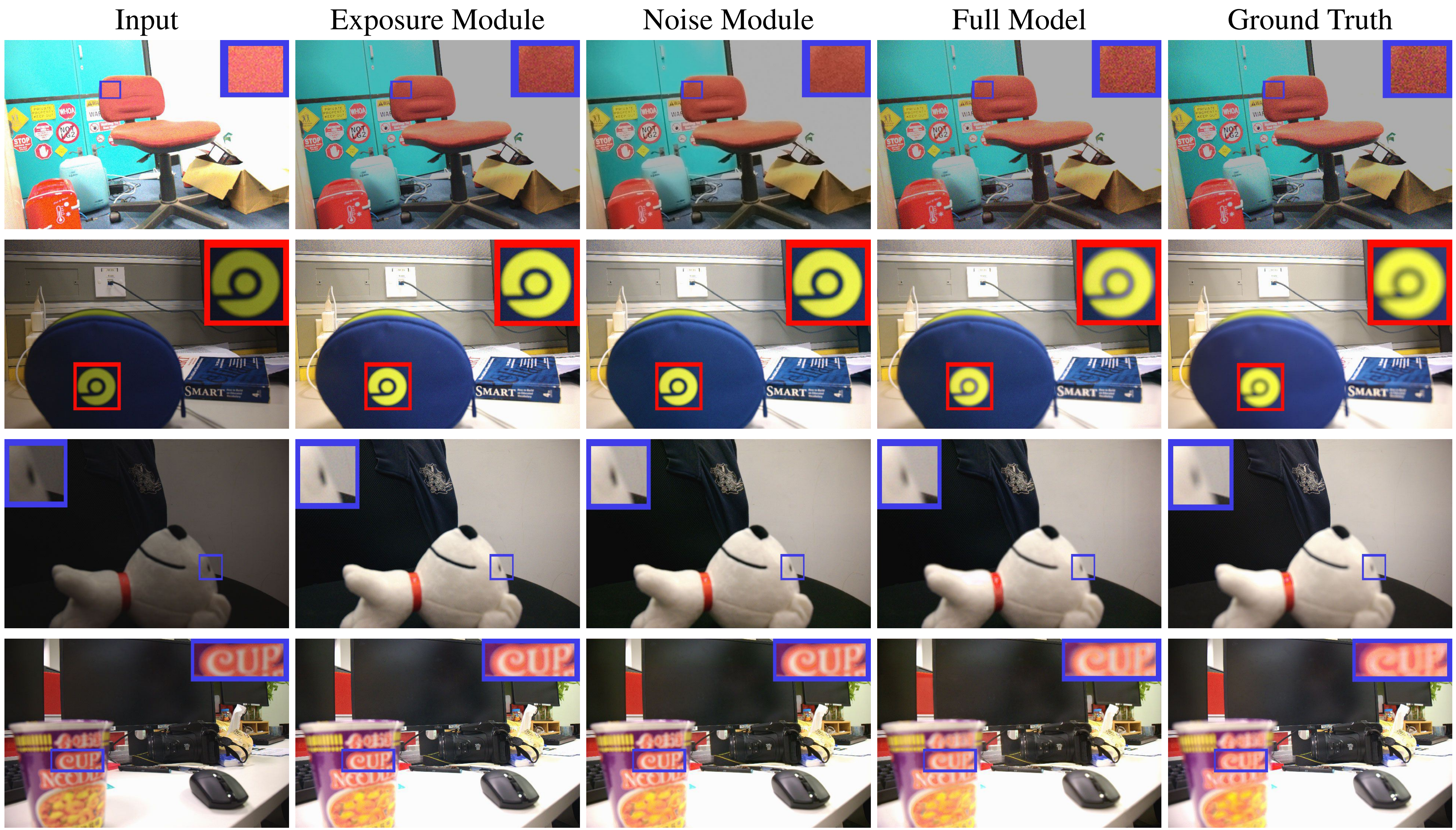
Key idea: deep neural networks as a controllable camera simulator to synthesize raw image data under different camera settings, including exposure time, ISO, and aperture.

(Ouyang et al., CVPR 2021)

Learned ISP

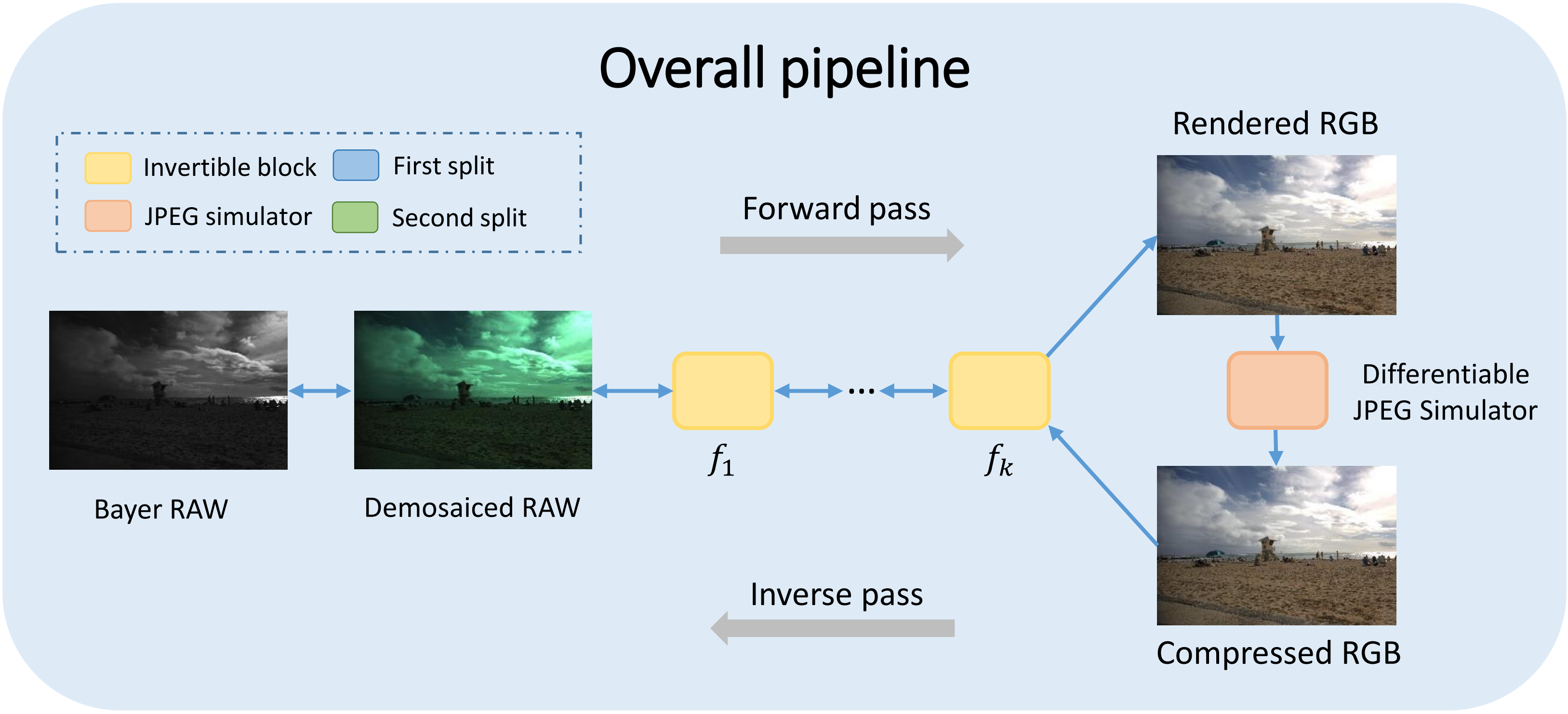
(Ouyang et al., CVPR 2021)

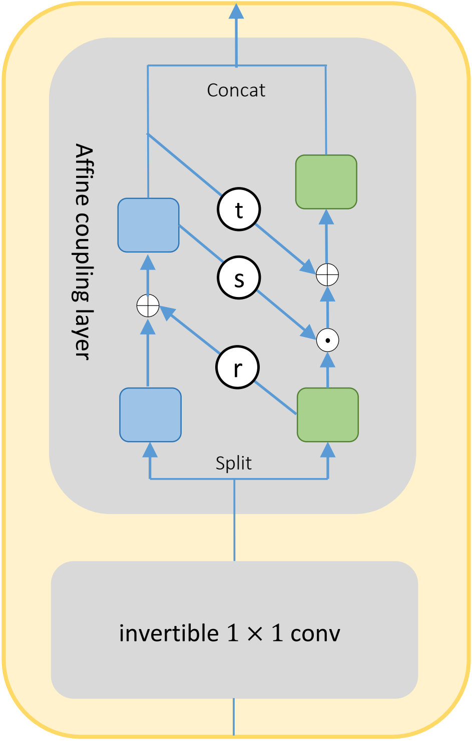
Learned ISP



(Ouyang et al., CVPR 2021)

Invertible ISP





Invertible block f_i

$$L = ||f(\mathbf{x}) - \mathbf{y}||_1 + \lambda||f^{-1}(\mathbf{y}) - \mathbf{x}||_1,$$

Key idea: Use invertible neural networks to design the invertible structure and integrate a differentiable JPEG simulator to enhance the network stability to JPEG compression.

(Xing et al., CVPR 2021)

Invertible ISP

Camera RAW

Our ISP f

Our rendered RGB

Inverse ISP f^{-1}

RAW error map

Our recovered RAW
(PSNR: 45.26)

Applications

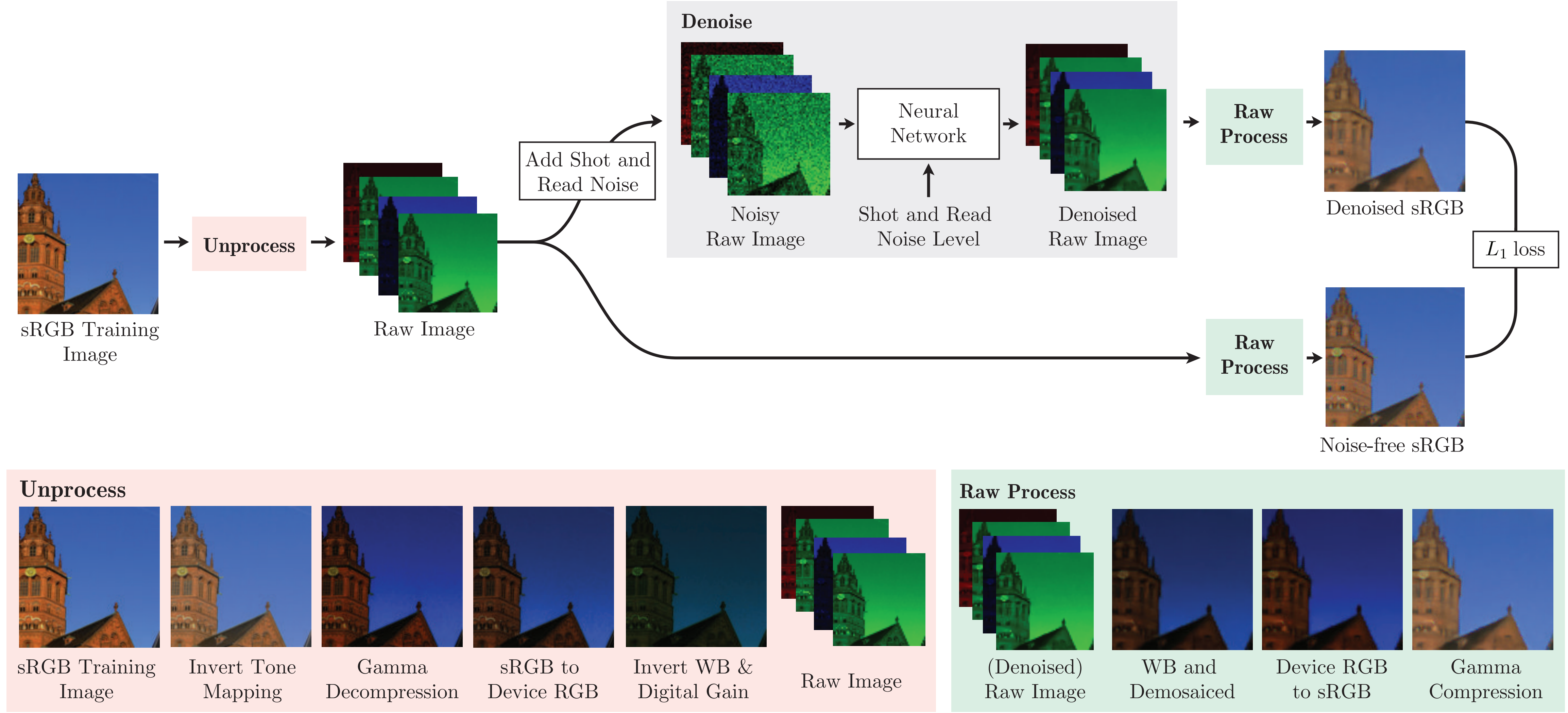
HDR reconstruction

Image retouching

Other application
(*i.e.*, RAW compression)

(Xing et al., CVPR 2021)

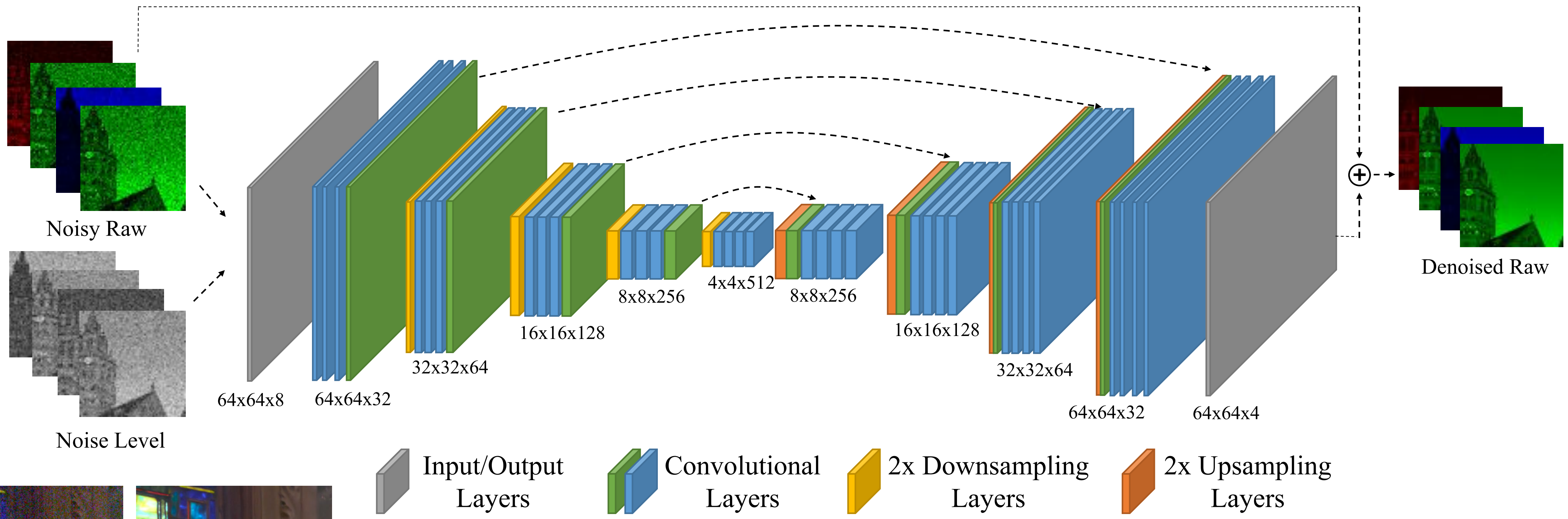
Image Denoising via Invertible ISP



Key idea: Transform sRGB images into RAW domain and perform denoising there

(Brooks et al., CVPR 2019)

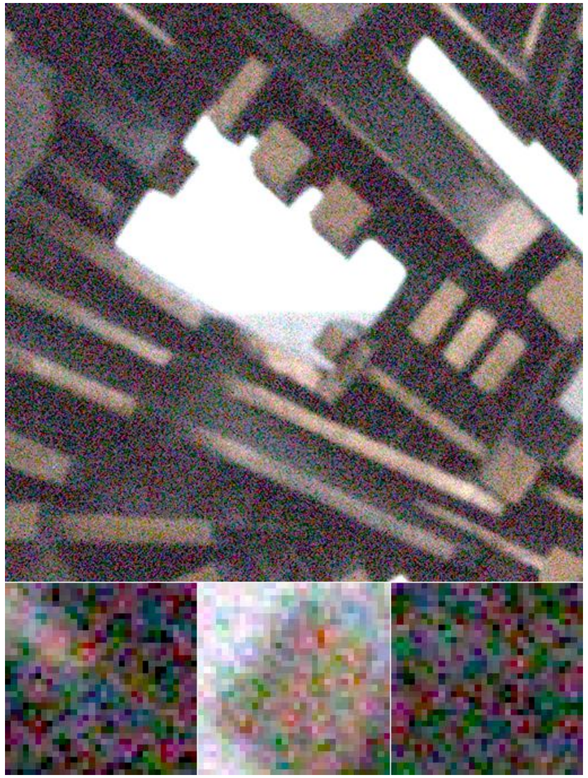
Image Denoising via Invertible ISP

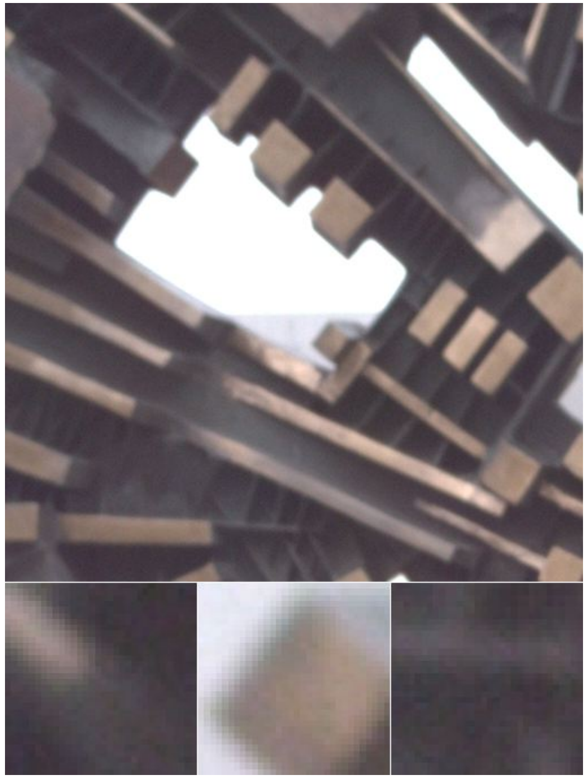


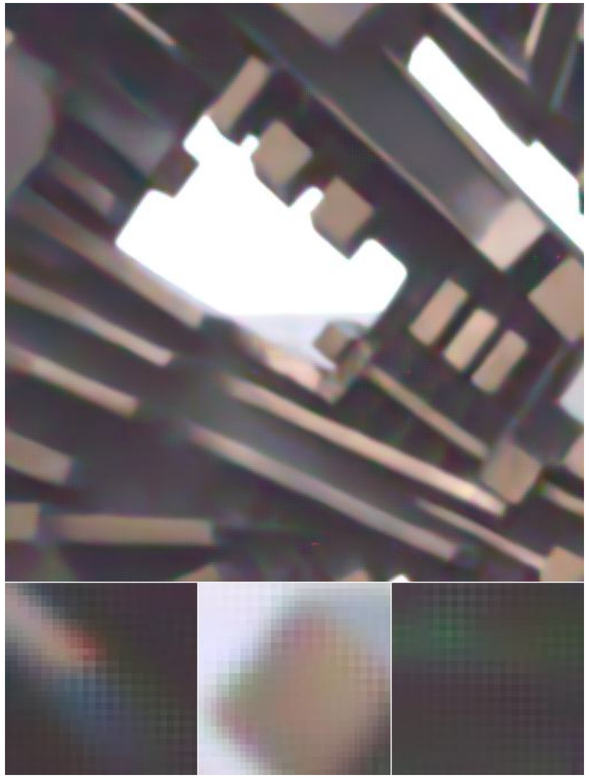
(a) Noisy Input (b) Our Model

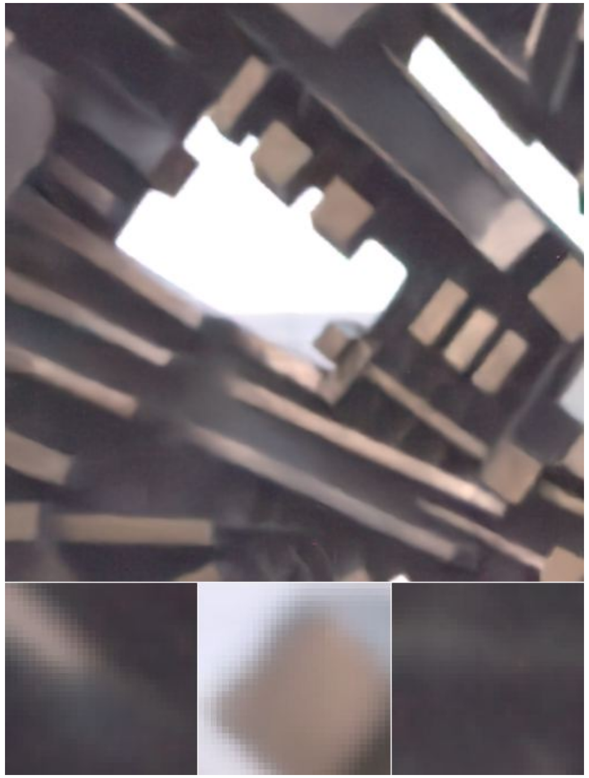
(Brooks et al., CVPR 2019)

Image Denoising via Invertible ISP


(a) Noisy Input, PSNR = 18.76

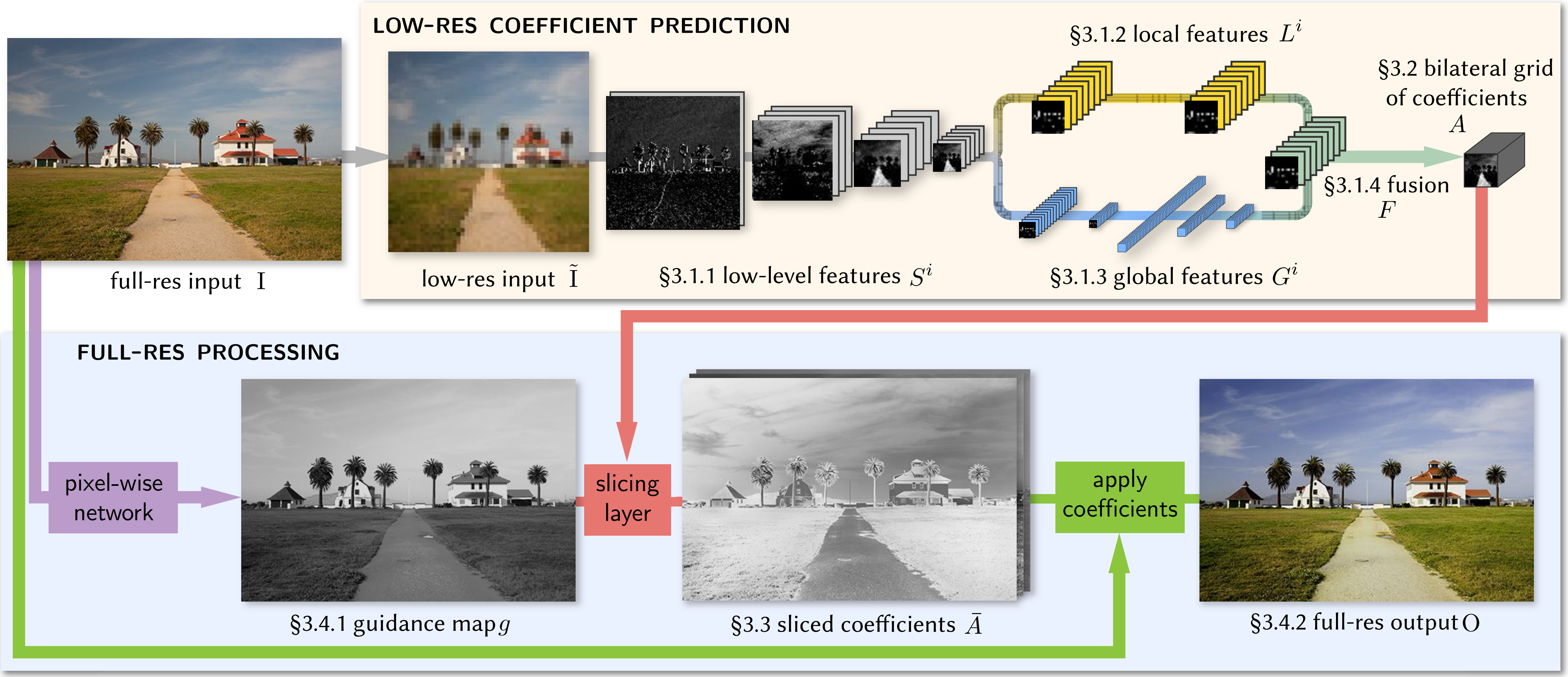

(b) Ground Truth


(c) N3Net [31], PSNR = 32.42


(d) Our Model, PSNR = 35.35

(Brooks et al., CVPR 2019)

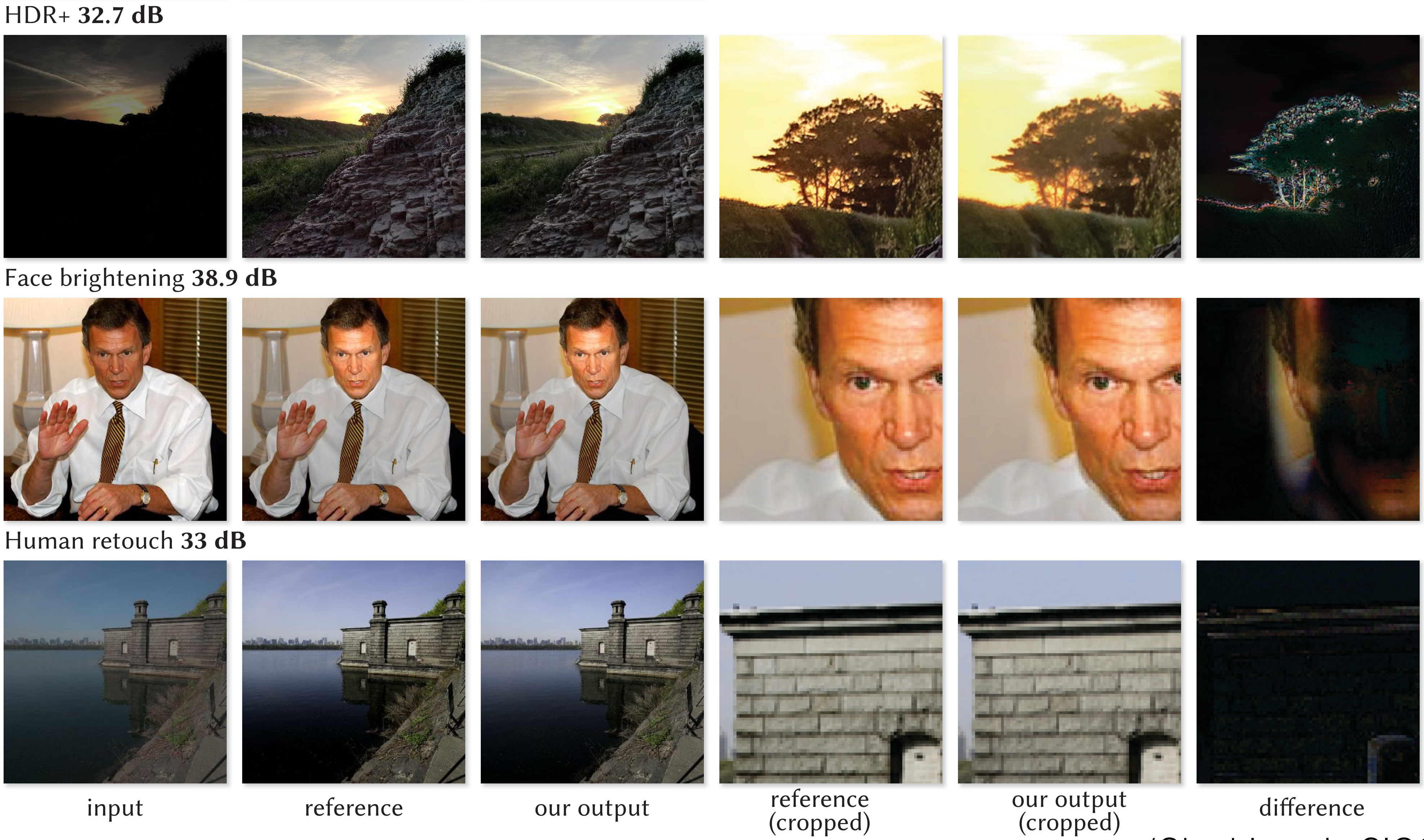
Image Enhancement



Key idea: Learn to imitate a reference operator, work much faster

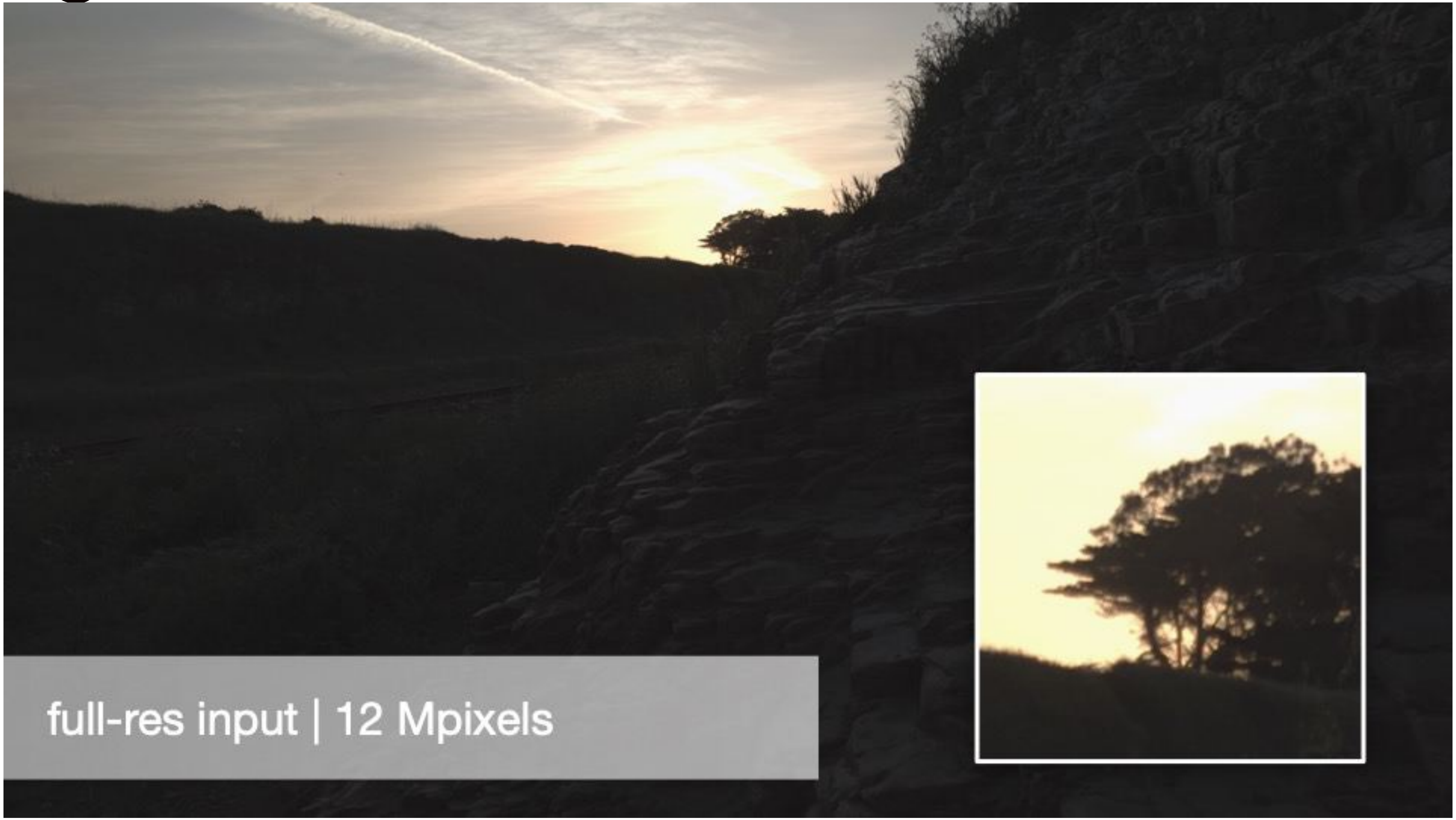
(Gharbi et al., SIGGRAPH 2017)

Image Enhancement



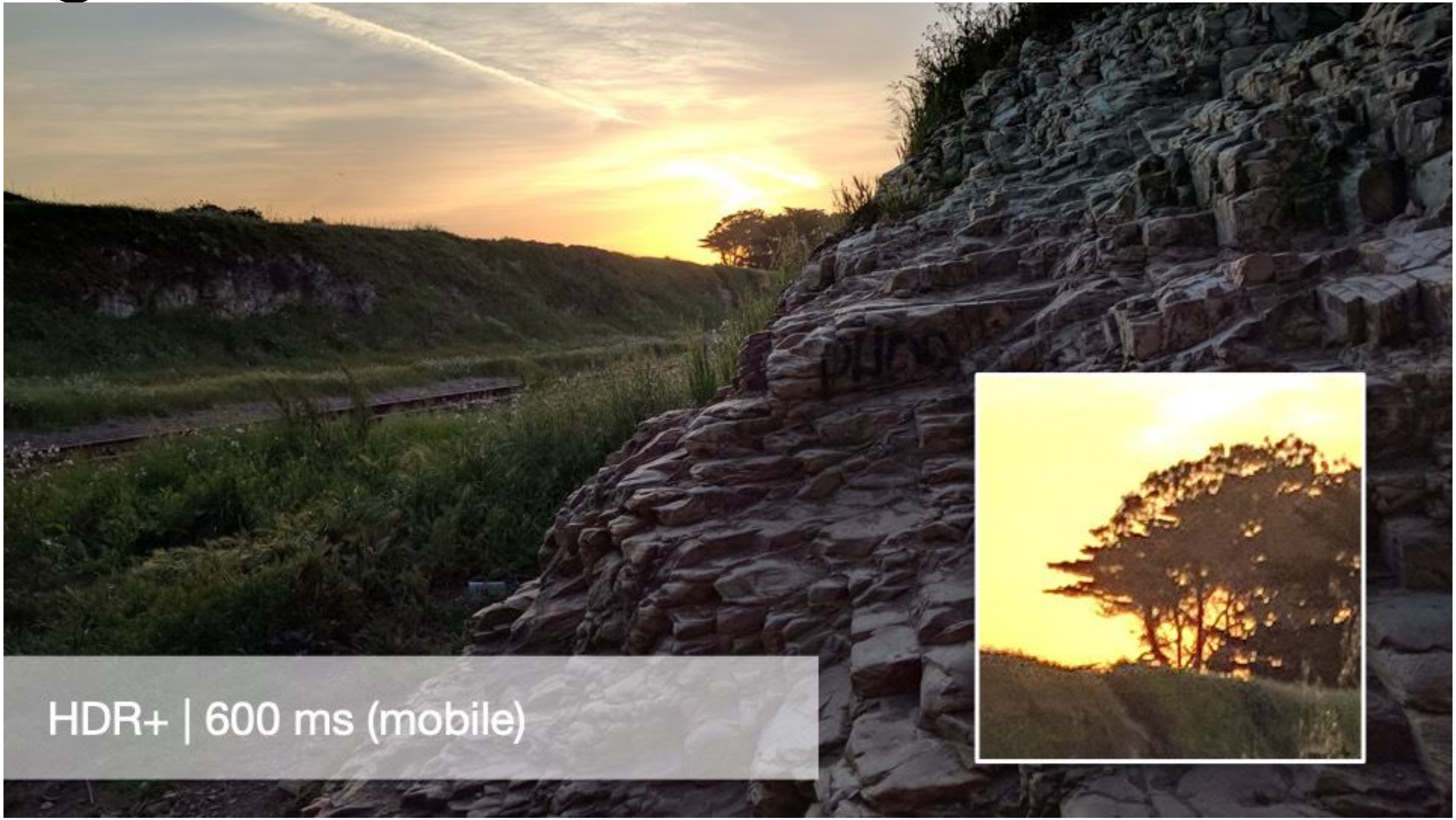
(Gharbi et al., SIGGRAPH 2017)

Image Enhancement



(Gharbi et al., SIGGRAPH 2017)

Image Enhancement

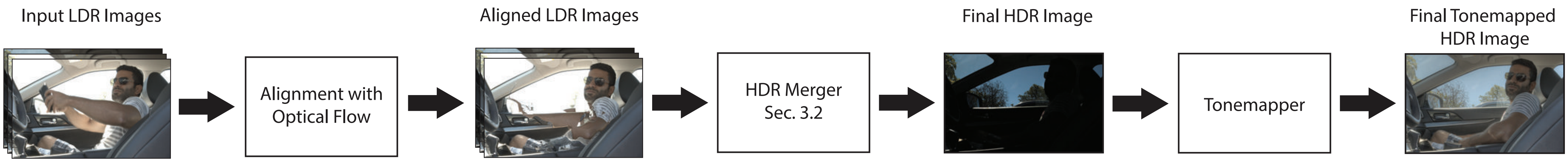


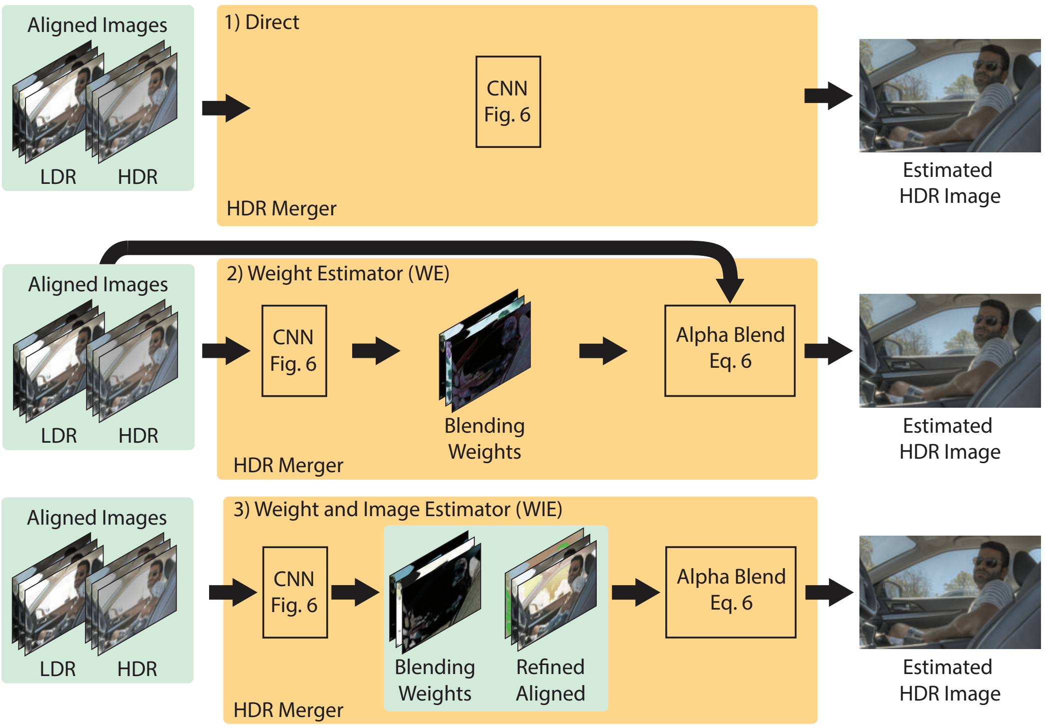
(Gharbi et al., SIGGRAPH 2017)

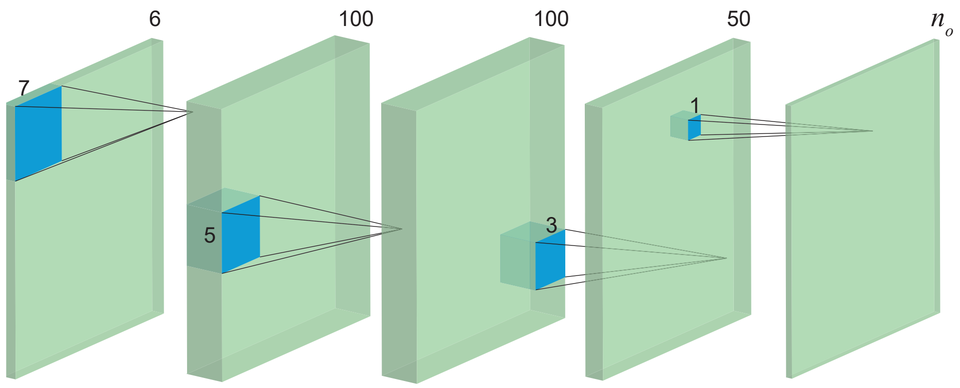
Image Enhancement

(Gharbi et al., SIGGRAPH 2017)

Deep HDR Reconstruction







Key idea: Cast HDR reconstruction problem as a learning problem

(Kalantari and Ramamoorthi, SIGGRAPH 2017)

Deep HDR Reconstruction

LDR Images | Our Tonemapped HDR Image | Kang (40.02 dB) | Sen (46.12 dB) | Ours (48.88 dB) | Ground Truth

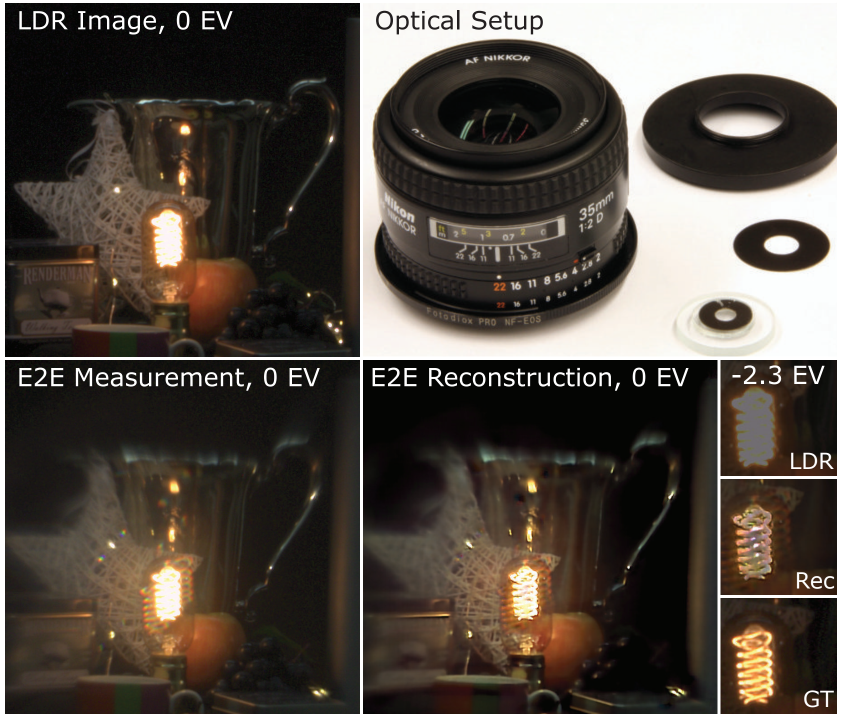
(Kalantari and Ramamoorthi, SIGGRAPH 2017)

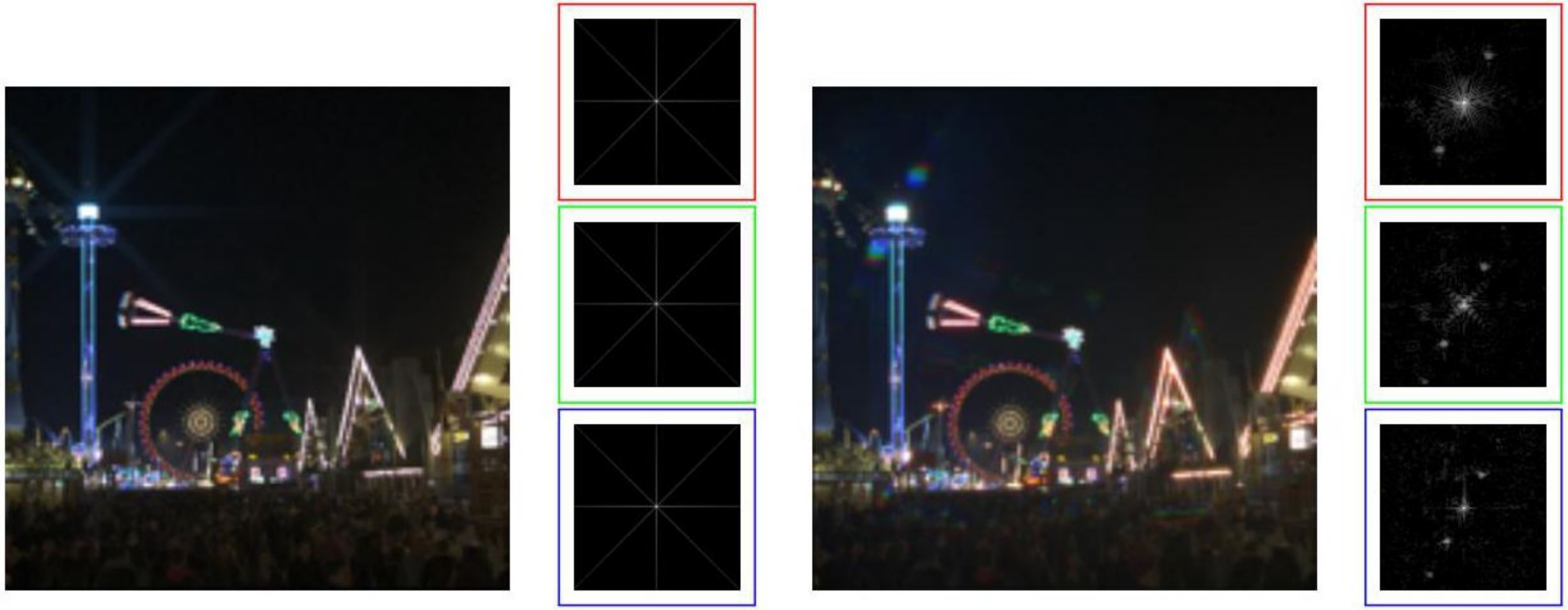
Deep HDR Reconstruction

Kang et al.	Oh et al.	Sen et al.	Hu et al.	Ours	Ground Truth
37.24	36.27	41.85	38.23	43.58	

(Kalantari and Ramamoorthi, SIGGRAPH 2017)

Deep Optics for HDR Imaging





(a) Star PSF

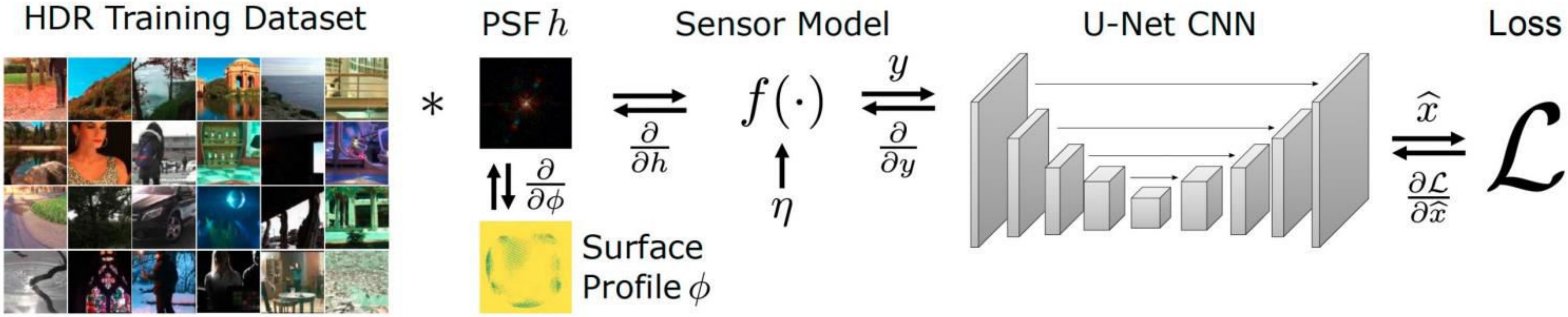
(b) E2E PSF

Key ideas:

- Minimize difference between reconstruction and tone-mapped GT images
- Jointly train an optical encoder and electronic decoder

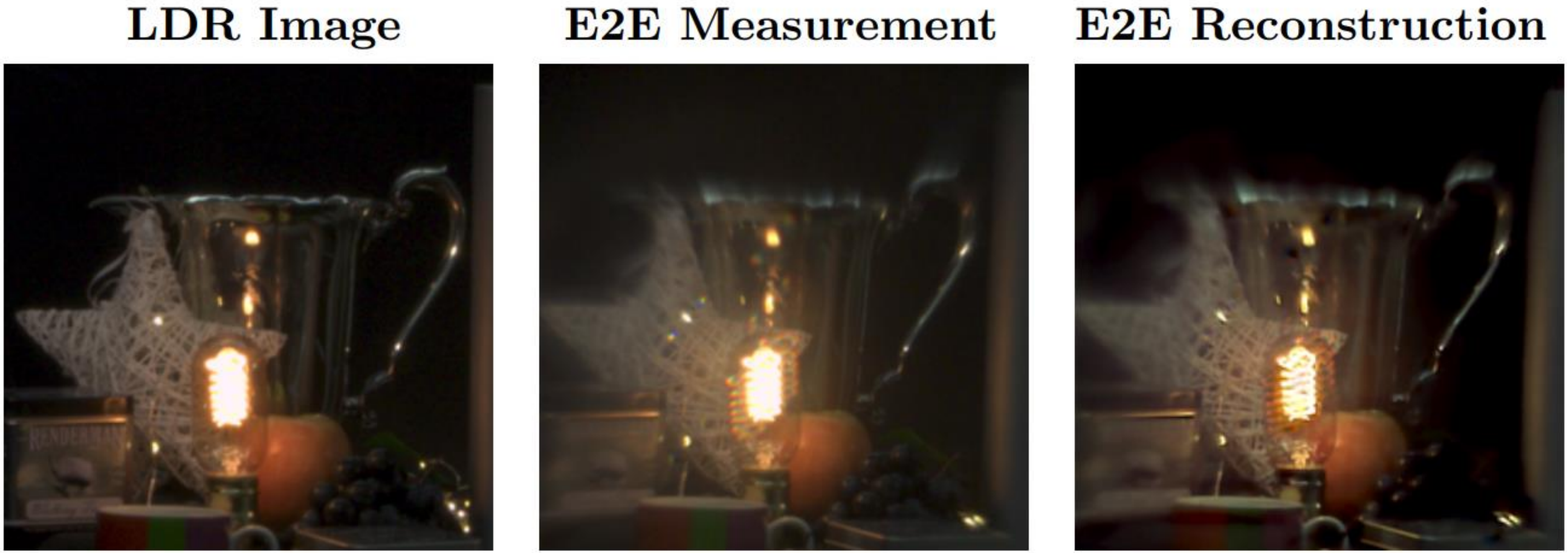
(Metzler et al., CVPR 2020)

Deep Optics for HDR Imaging



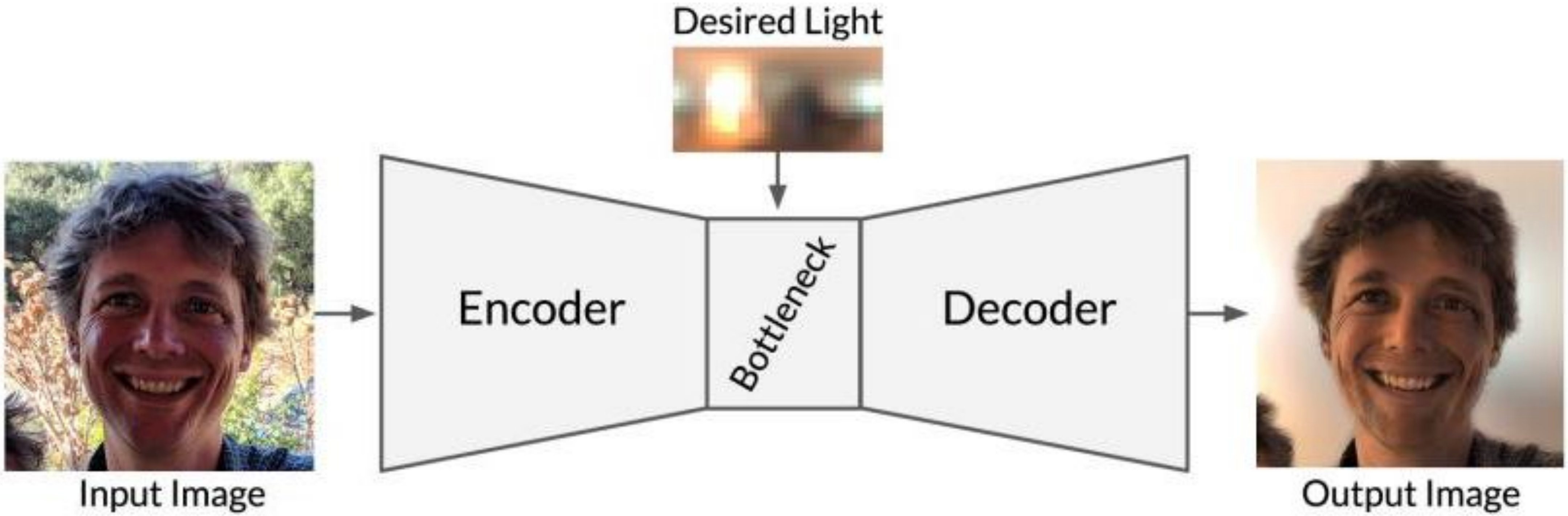
(Metzler et al., CVPR 2020)

Deep Optics for HDR Imaging



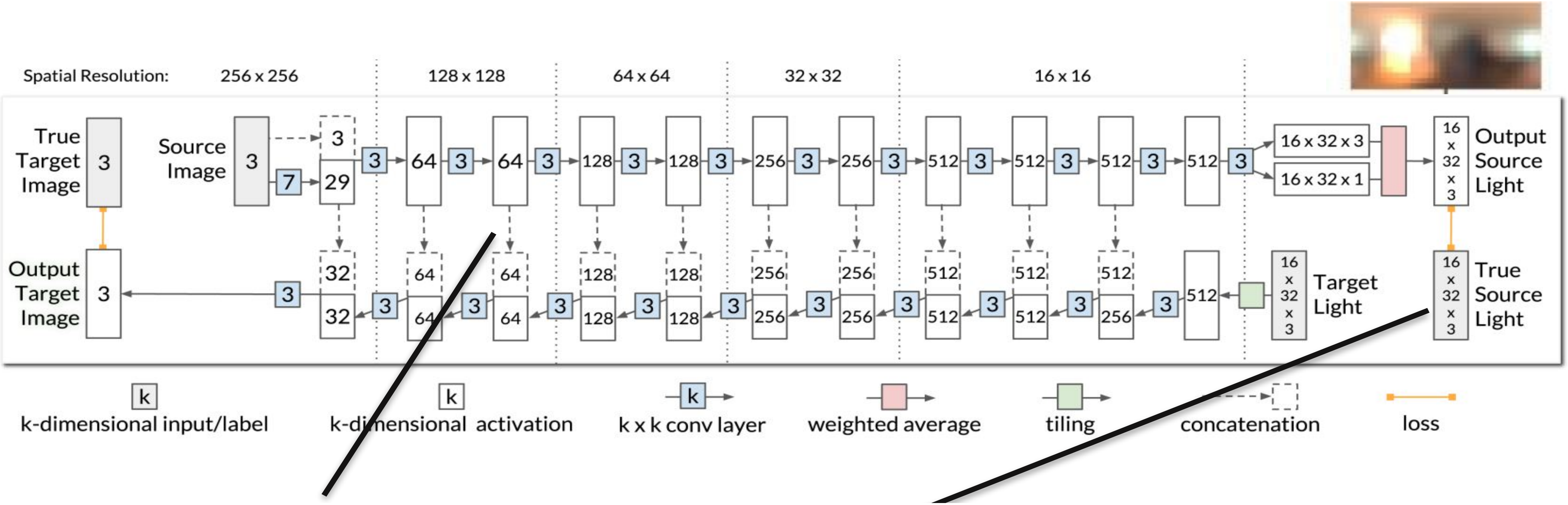
(Metzler et al., CVPR 2020)

Image Relighting



(Sun et al., ACM TOG 2019)

Image Relighting



Residual connections

bottleneck

(Sun et al., ACM TOG 2019)

Image Relighting

Light-stage dataset capture (Google)

(Sun et al., ACM TOG 2019)

Image Relighting

OLAT photos
(columns)

$$b = Ax$$

Re-rendered image

Environment map

(a) OLAT images (7 cameras). (b) Ground-truth renderings.

(Sun et al., ACM TOG 2019)

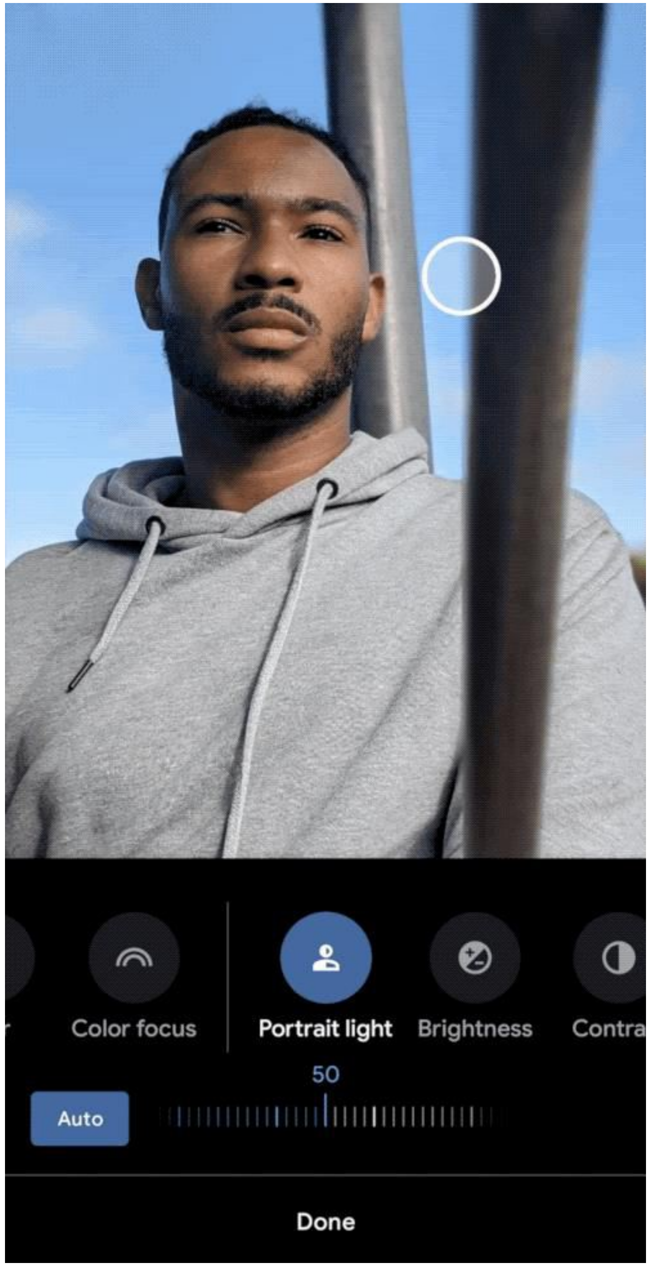
Image Relighting

(a) Input image and estimated lighting (b) Rendered images from our method under three novel illuminations

(Sun et al., ACM TOG 2019)

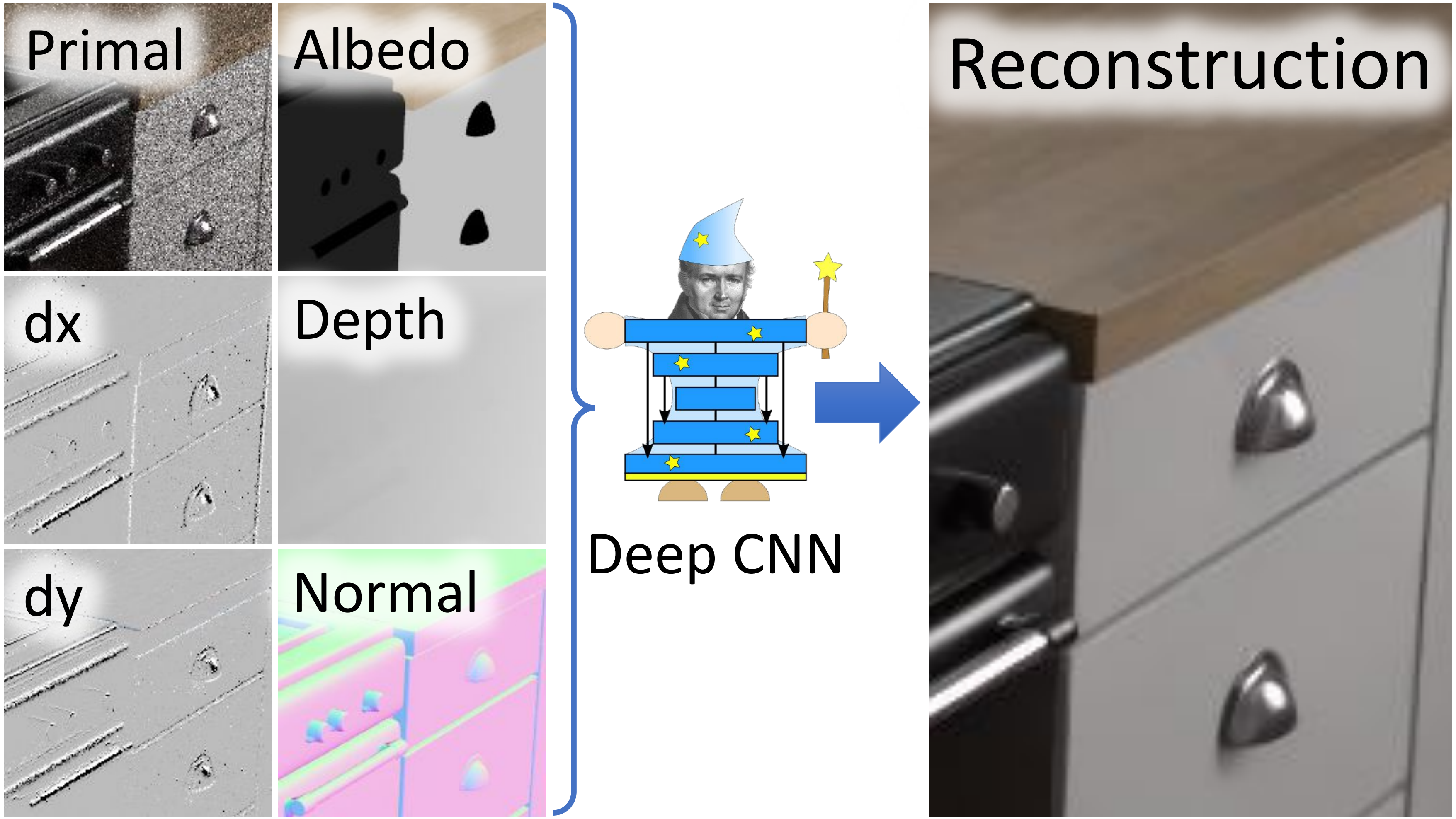
Image Relighting

Now a feature in Google Pixel phones



(Sun et al., ACM TOG 2019)

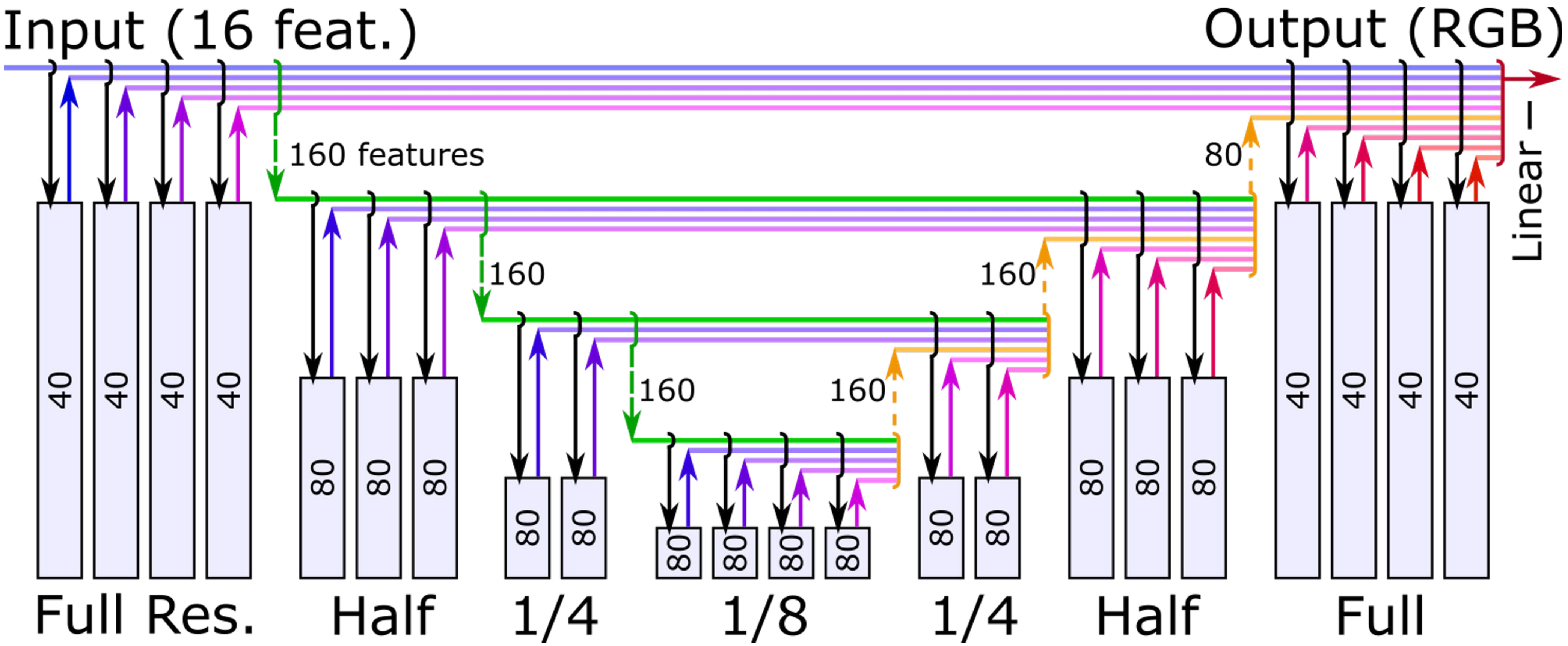
Gradient-Domain Rendering



Key idea: Cast gradient-domain rendering as a learning problem

(Kettunen et al., SIGGRAPH 2019)

Gradient-Domain Rendering



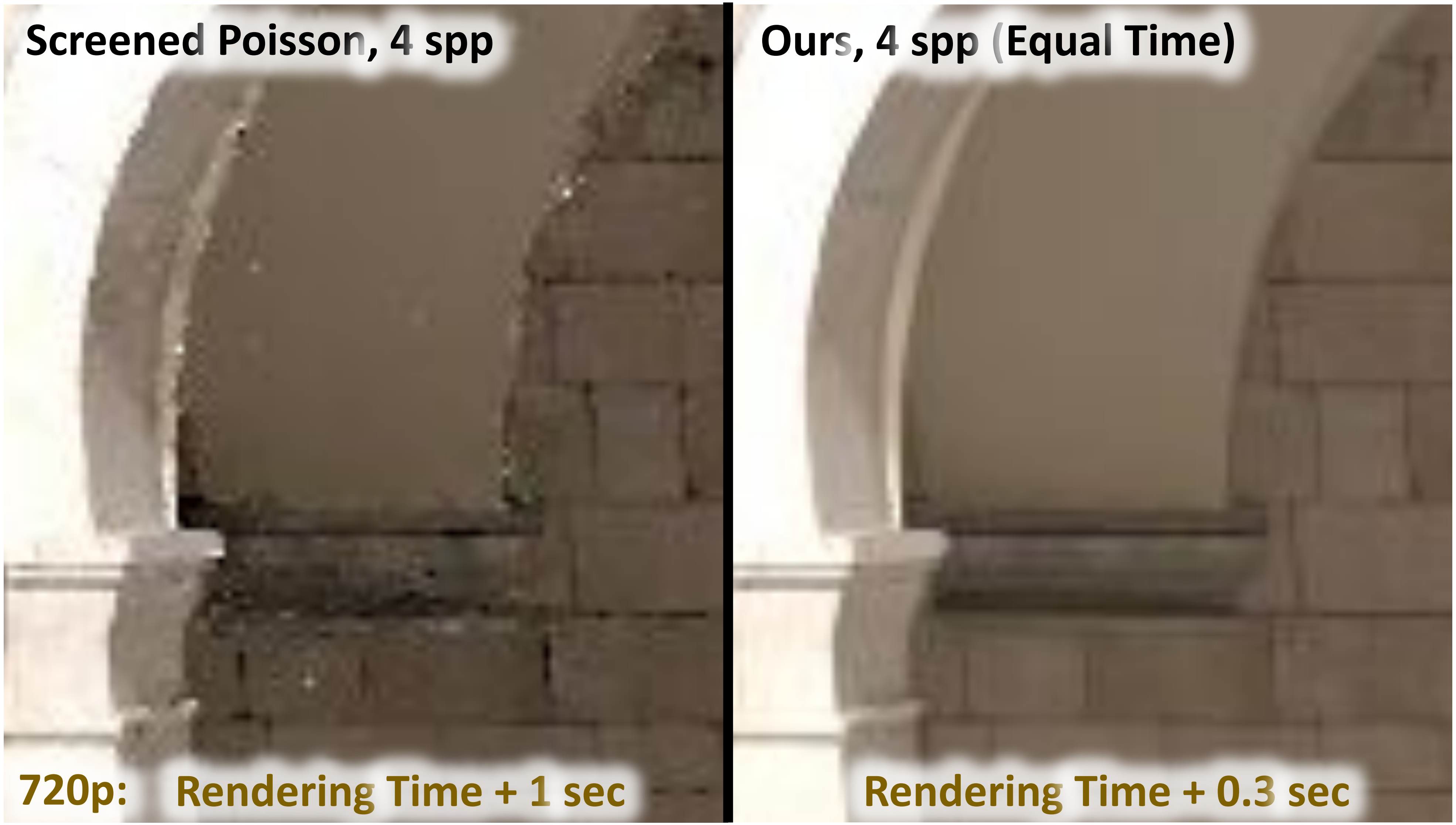
Based on DenseNet [Huang2016] and U-Net [Ronneberger2015]

(Kettunen et al., SIGGRAPH 2019)

Gradient-Domain Rendering

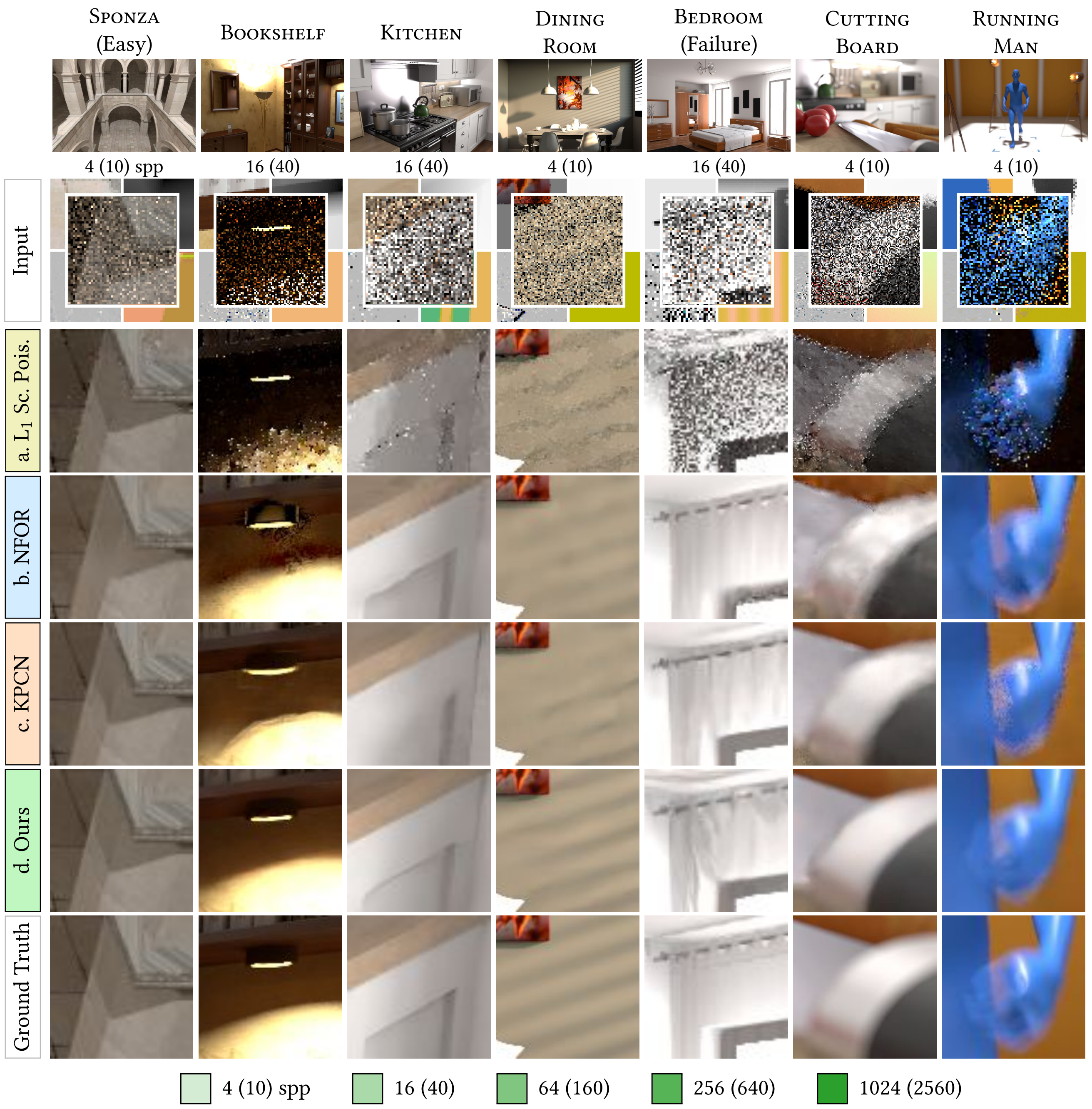
(Kettunen et al., SIGGRAPH 2019)

Gradient-Domain Rendering



(Kettunen et al., SIGGRAPH 2019)

Gradient-Domain Rendering



(Kettunen et al., SIGGRAPH 2019)

Next Lecture:
Deep Generative Models