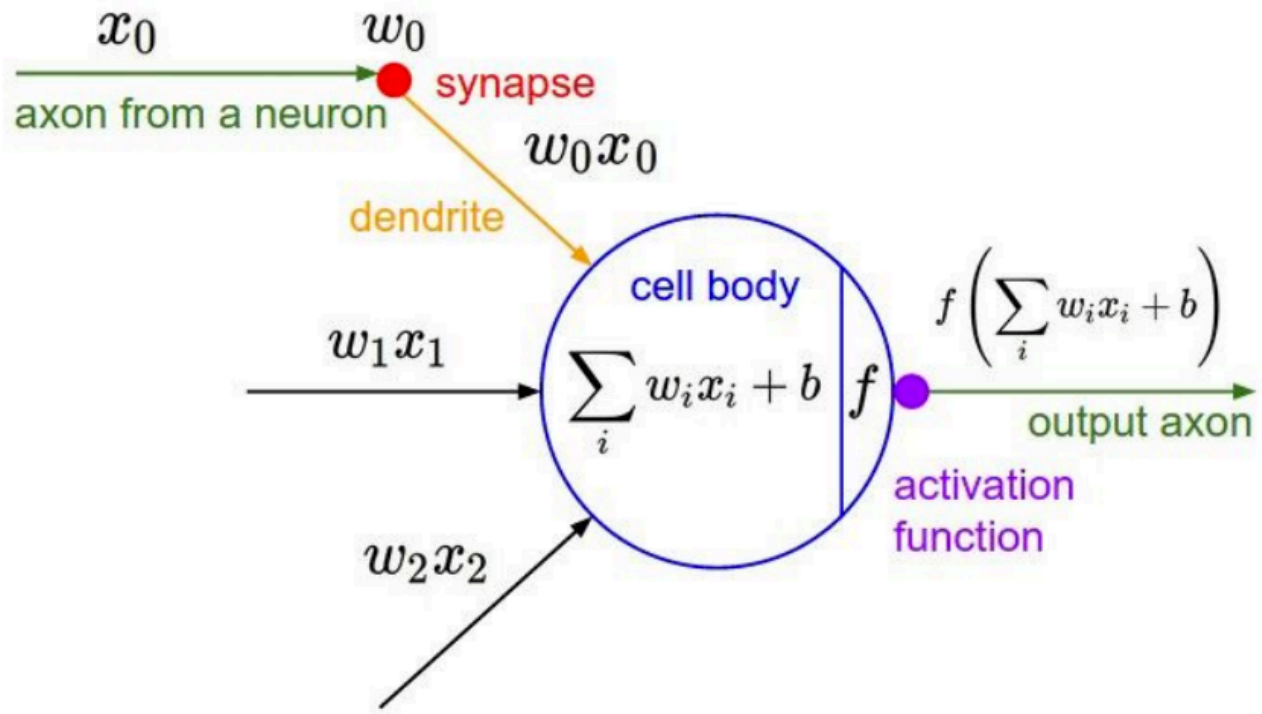
Machine Learning in Imaging

BME 590L
Roarke Horstmeyer

Lecture 8: Ingredients for a convolutional neural network

Note: Borrowed much material from Stanford CS231n Lectures 4 - 10

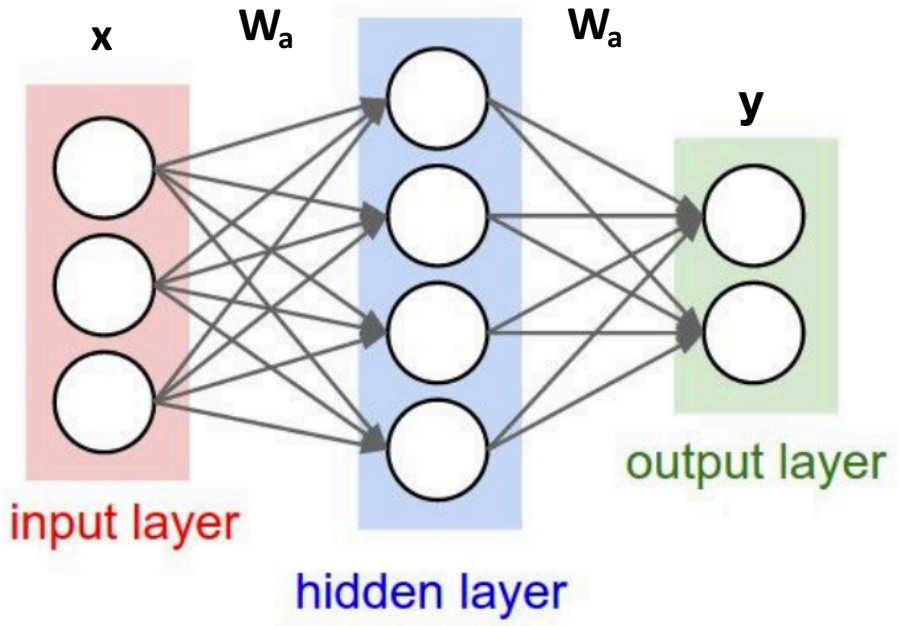
No more asides! Now lets get into neural networks…



Single ”neuron”:
Inner product of inputs **x** with learned weights **w** & non-linearity afterwards

- Multiple weighted inputs: **x** -> y = $\mathbf{w}^T\mathbf{x}$ is “dendrites into cell body”
- Non-linearity f () after sum = “neuron’s activation function” (loose interp.)

No more asides! Now lets get into neural networks…



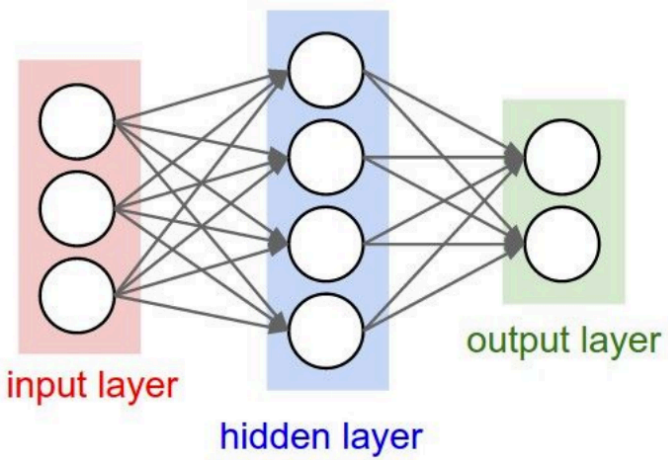
$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = NL\bullet \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix} NL\bullet \begin{bmatrix} w_{11} & w_{21} & w_{21} \\ w_{12} & w_{22} & w_{22} \\ w_{13} & w_{23} & w_{23} \\ w_{14} & w_{24} & w_{24} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$\mathbf{y}$ $\mathbf{W_b}$ $\mathbf{W_a}$ $\mathbf{x}$

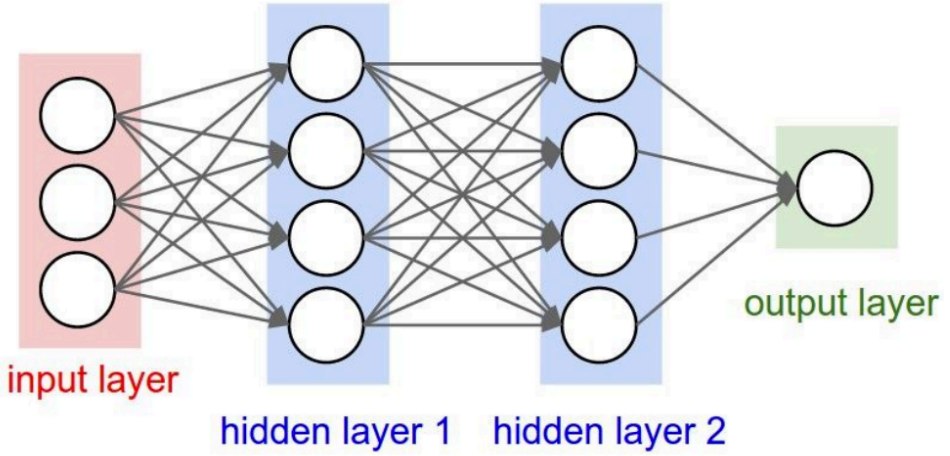
- For multiple cells (units), use matrix **W** to connect inputs to outputs
- These cascade in layers

Next step: Neural networks! Pretty much the same thing…

2-layer network:



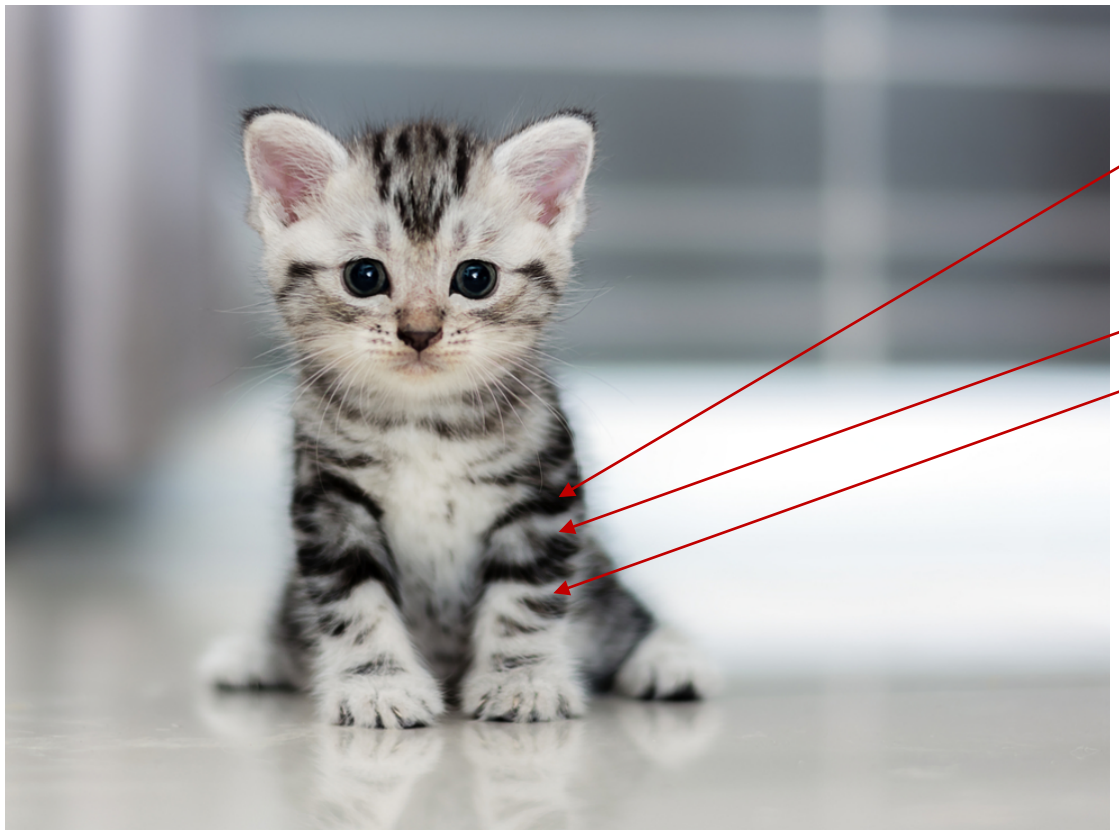
3-layer network:



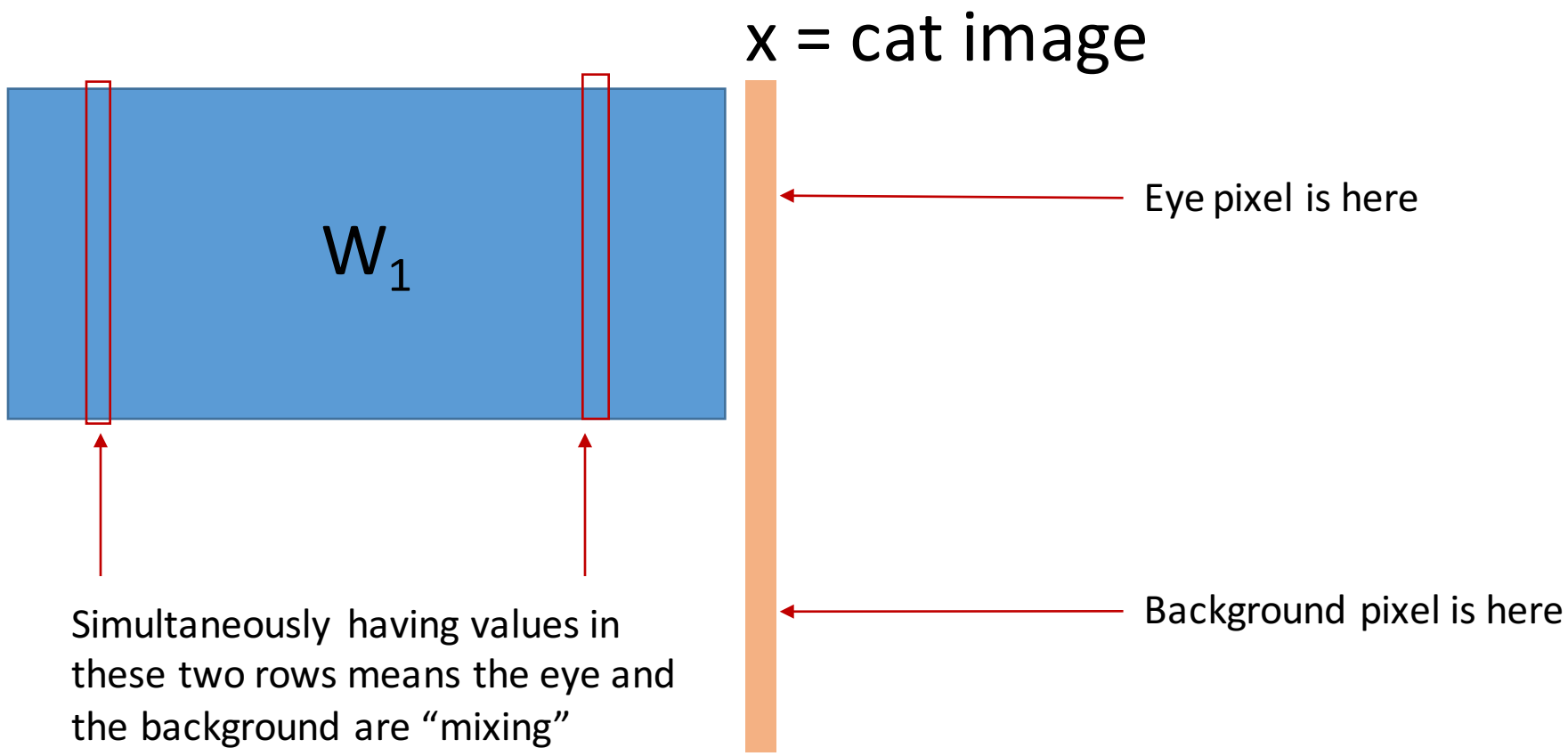
or 3-layer Neural Network

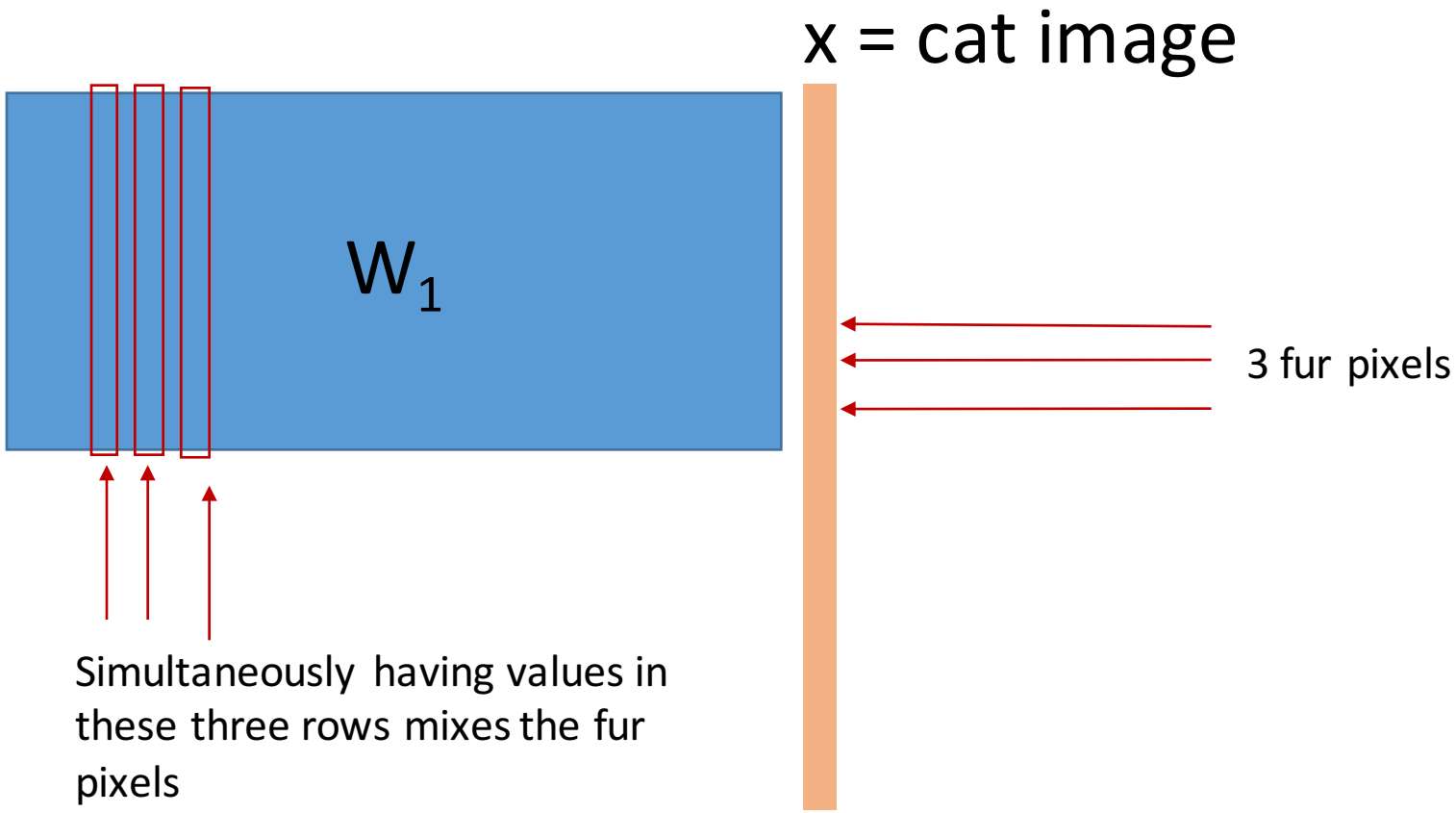
$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$

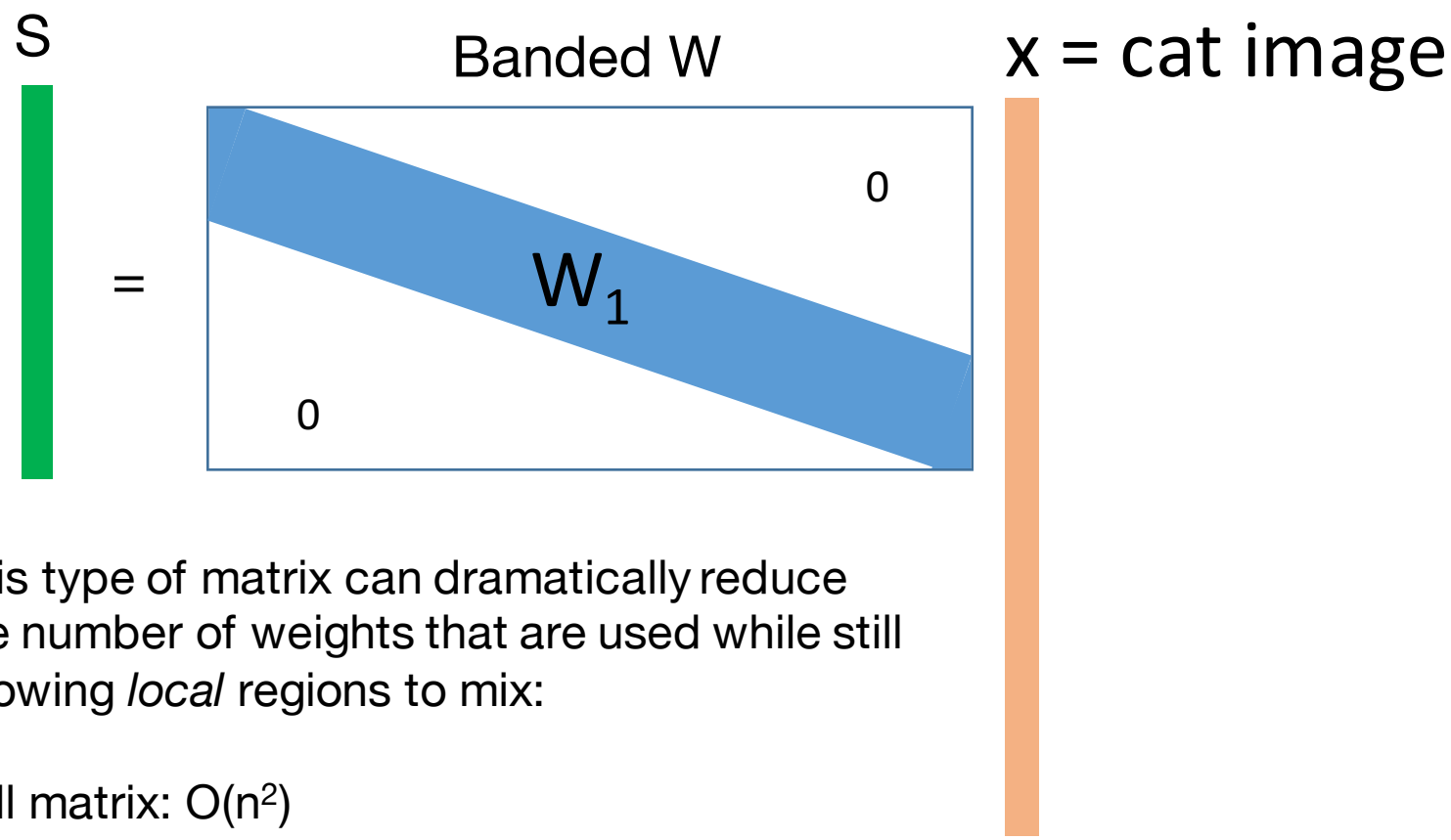
We probably don’t need to mix these two pixels to figure out that this is a cat



But understanding the stripes in these 3 pixels right near each other is going to be pretty helpful…


x = cat image
W_1
Eye pixel is here
Background pixel is here
Simultaneously having values in these two rows means the eye and the background are “mixing”

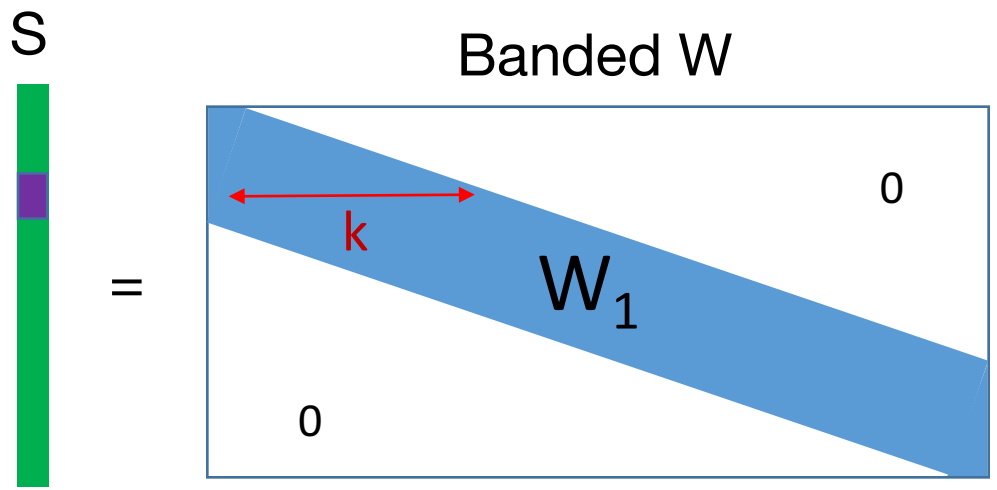

x = cat image
W_1
3 fur pixels
Simultaneously having values in these three rows mixes the fur pixels



This type of matrix can dramatically reduce the number of weights that are used while still allowing *local* regions to mix:

Full matrix: $O(n^2)$

Banded matrix: k·O(n)



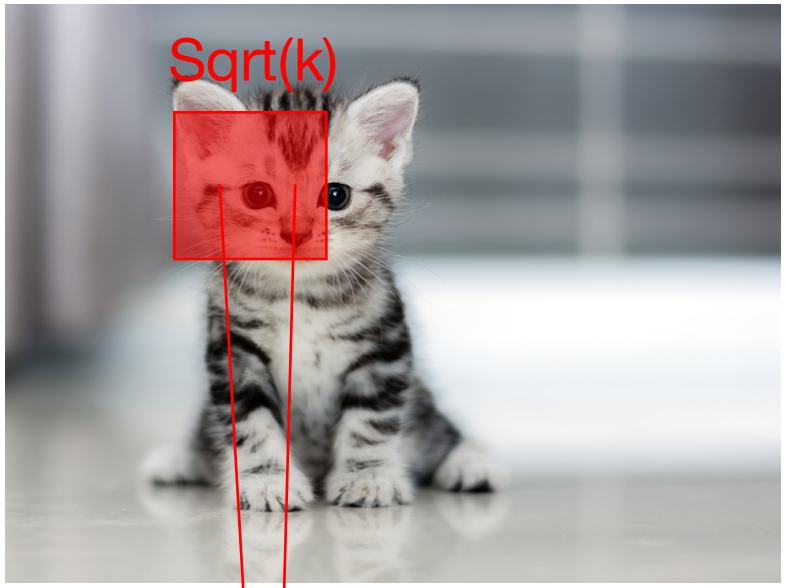
x = cat image

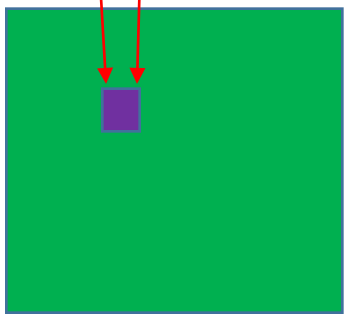
This type of matrix can dramatically reduce the number of weights that are used while still allowing *local* regions to mix:

Full matrix: $O(n^2)$

Banded matrix: k·O(n)

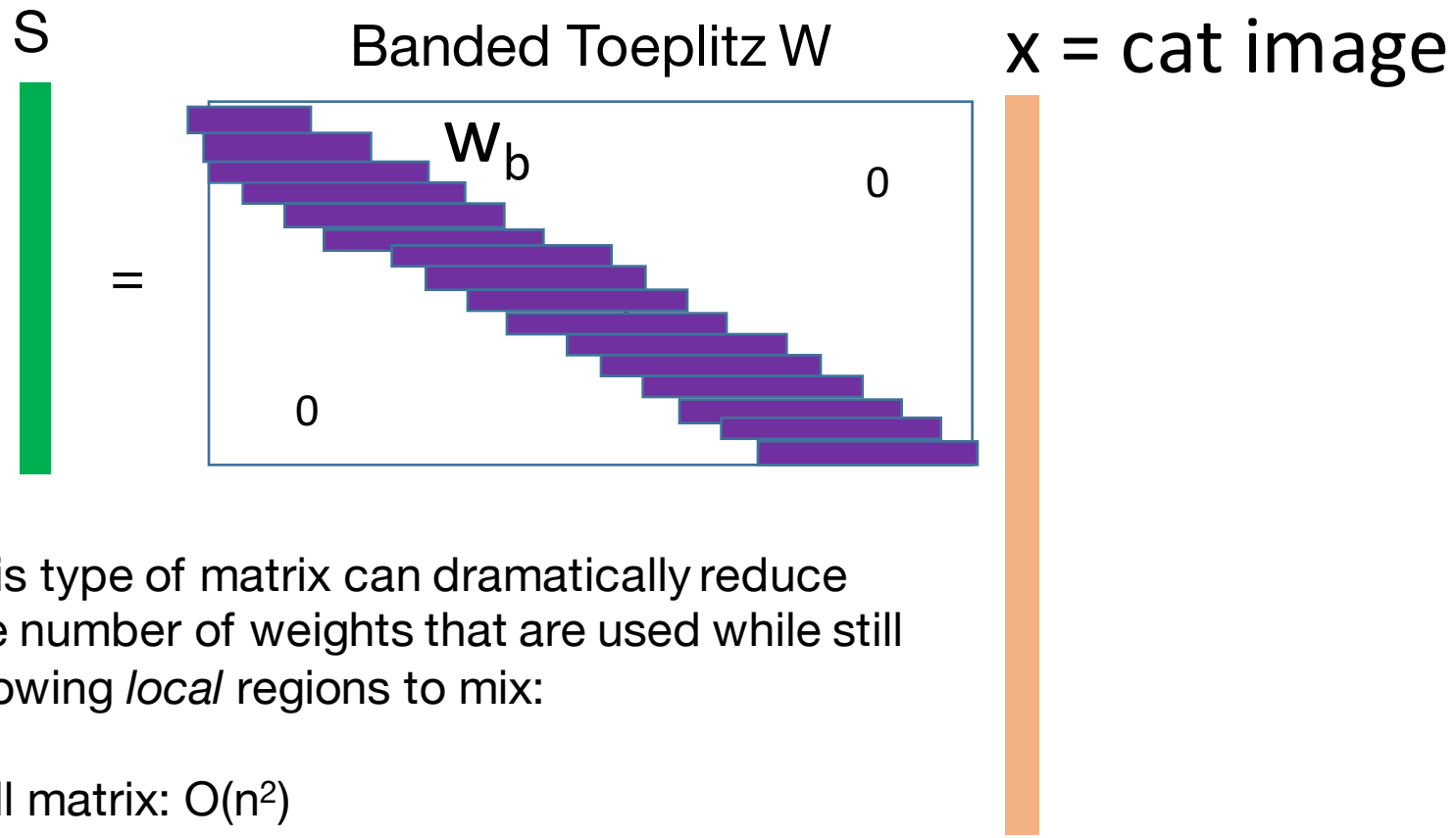
Image interpretation





Mix all the pixels in the red box, with associated weights, to form this entry of S

Simplification #2: Have each band be *the same weights*

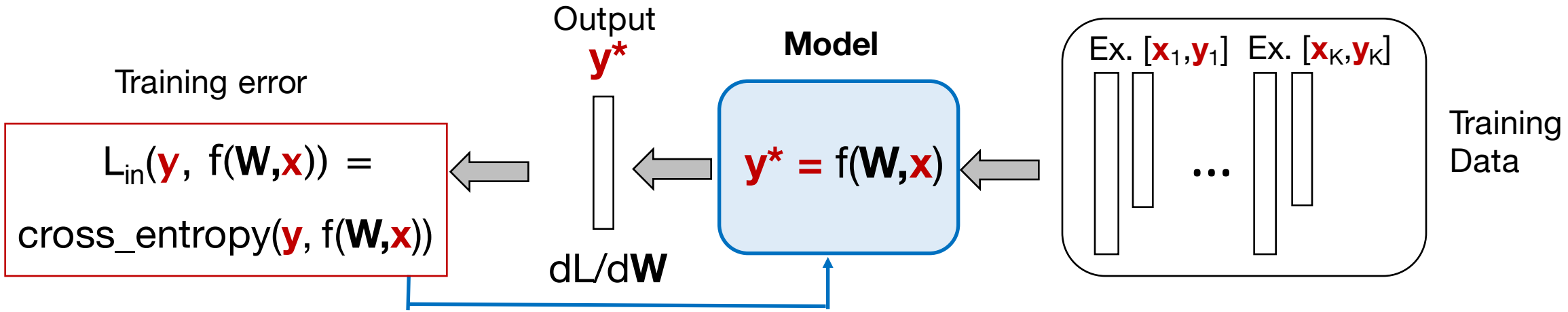


This type of matrix can dramatically reduce the number of weights that are used while still allowing *local* regions to mix:

Full matrix: $O(n^2)$
Banded matrix: $k \cdot O(n)$
Banded Toeplitz matrix: k

This is the definition of a convolution

Our very basic convolutional neural network

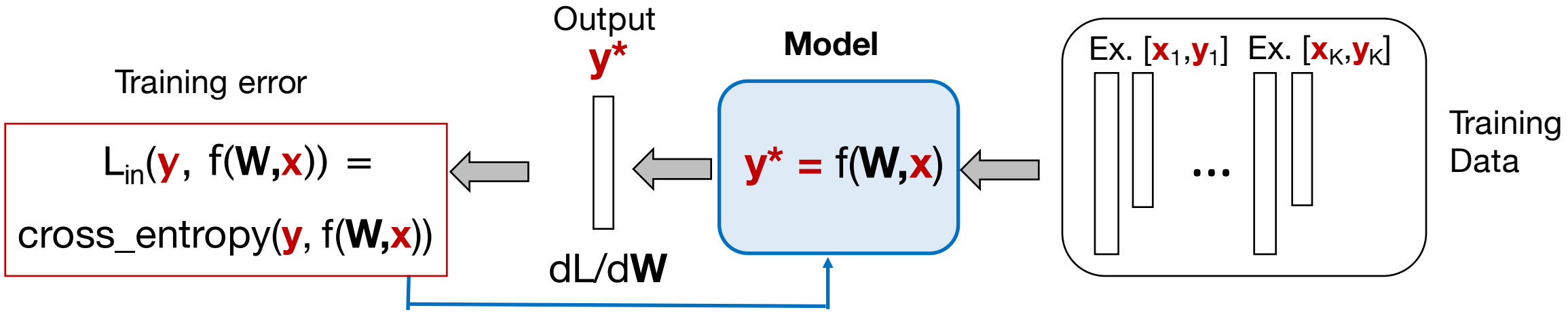


$$L(w) = \frac{1}{N}\sum_{i=1}^{N} \ln(1 + e^{-y_i \max(0, \mathbf{W}_2 \max(0, \mathbf{W}_1 x_i))})$$

W_1 is banded Toeplitz matrix, W_2 is a full matrix

2-layer network

Our very basic convolutional neural network

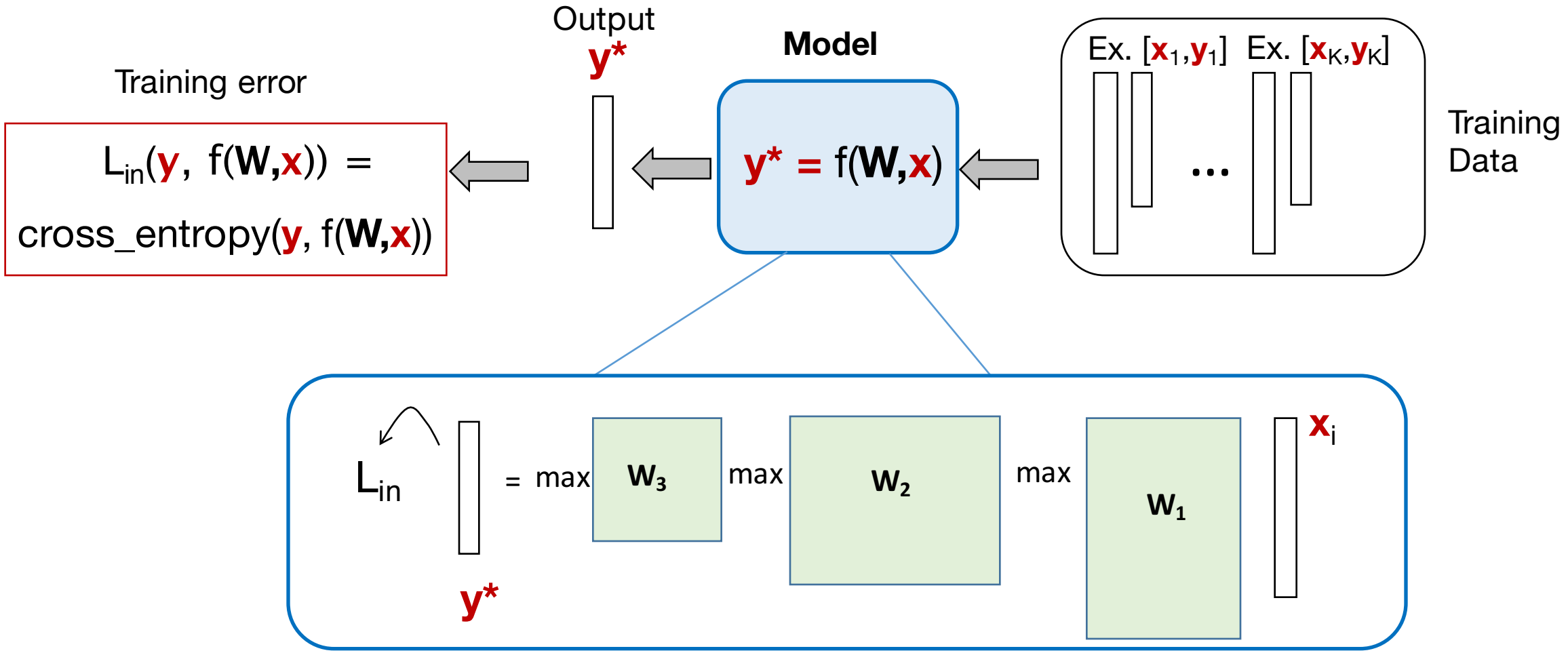


$$L(w) = \frac{1}{N}\sum_{i=1}^{N} \ln(1 + e^{-y_i \max(0, \mathbf{W}_3 \max(0, \mathbf{W}_2 \max(0, \mathbf{W}_1 x_i)))})$$

W_1 and W_2 are banded Toeplitz matrices, W_3 is a full matrix

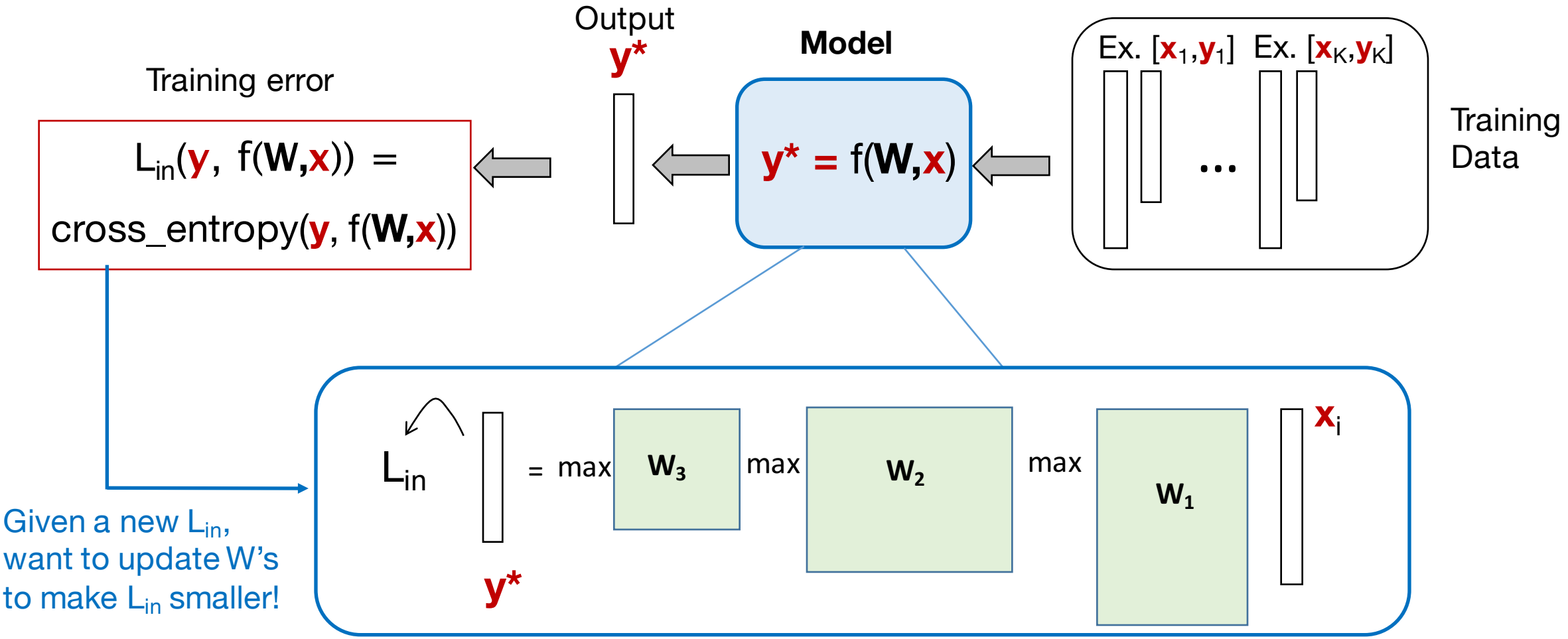
3-layer network

Our very basic convolutional neural network



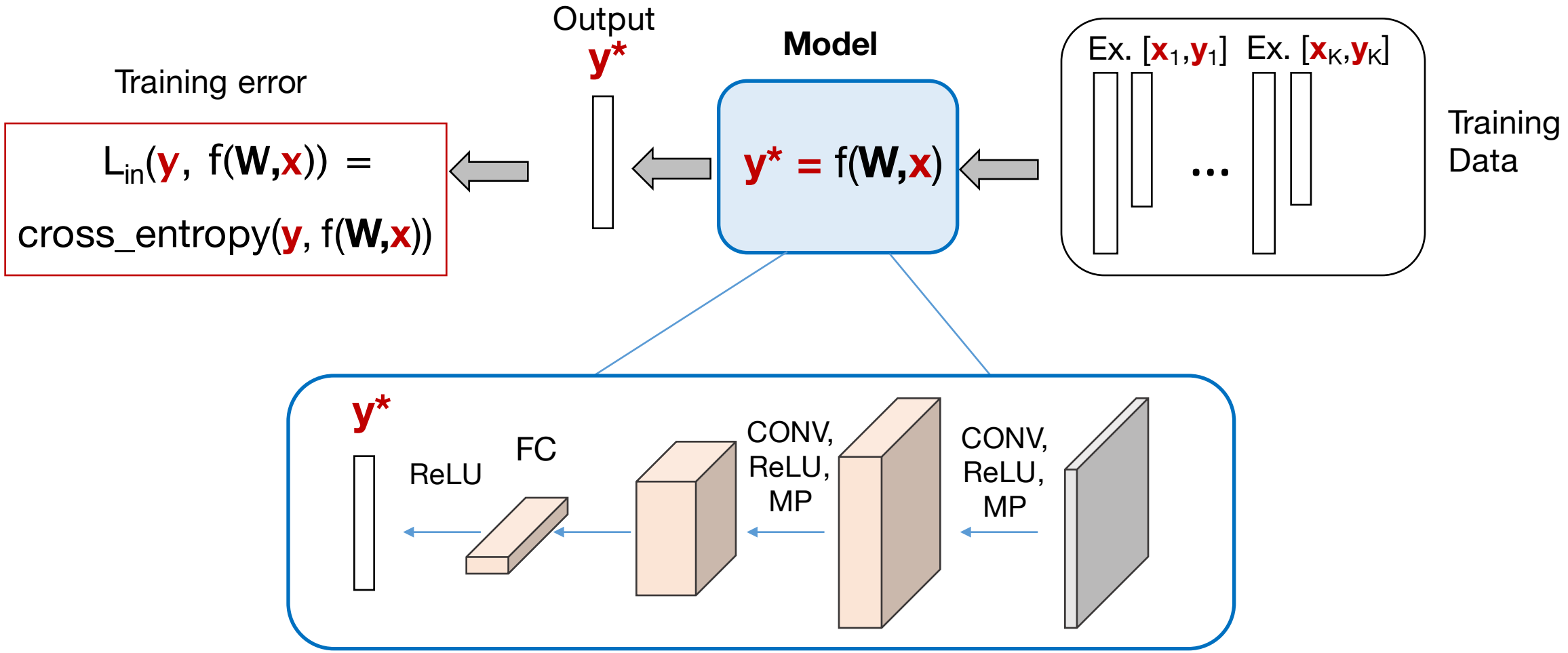
Forward pass: from $\mathbf{x}_i$ and current $\mathbf{W}$'s, find L_{in}

Our very basic convolutional neural network



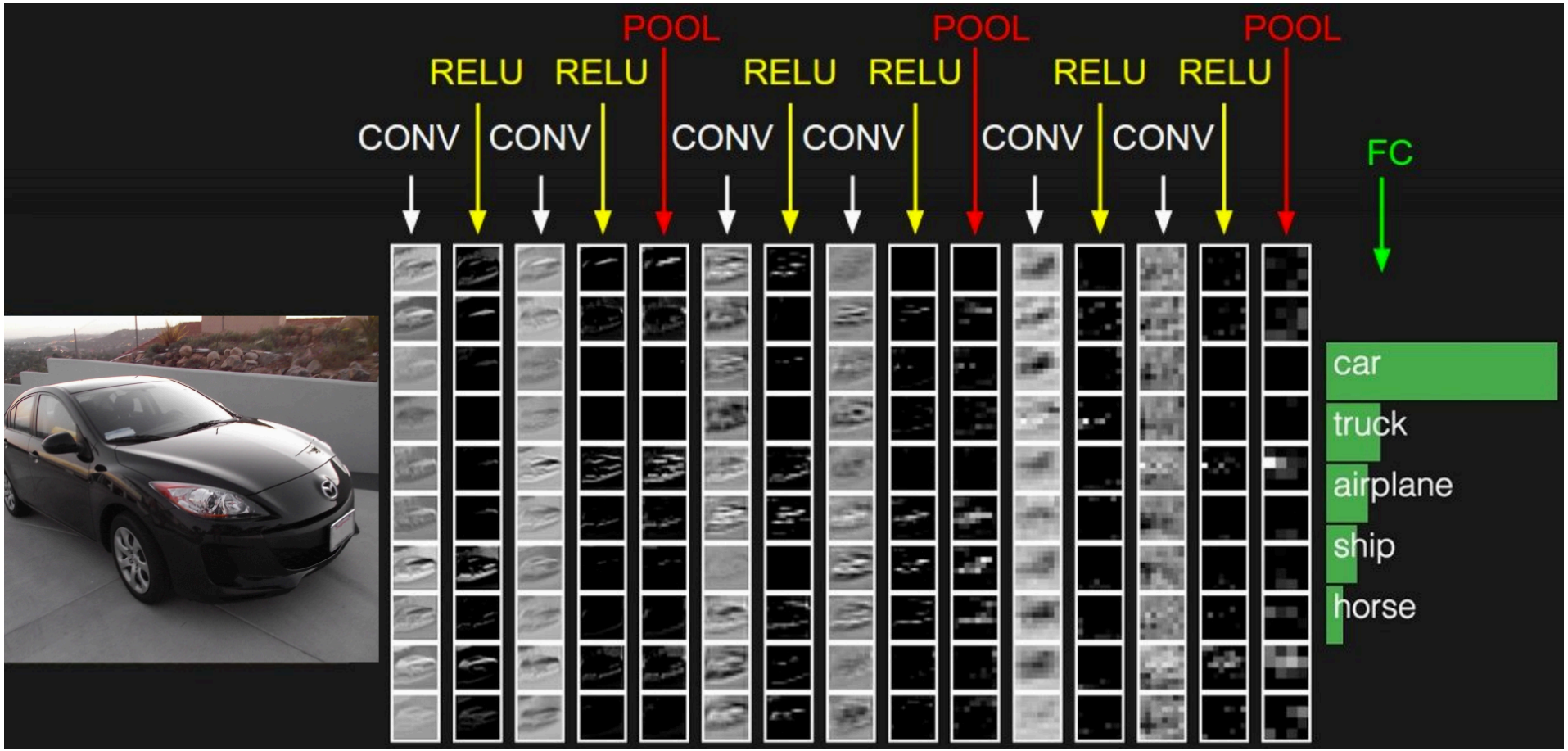
Next class: Backwards pass uses new L_{in} to update **W**'s

Our very basic convolutional neural network



3-layer network for 2D images

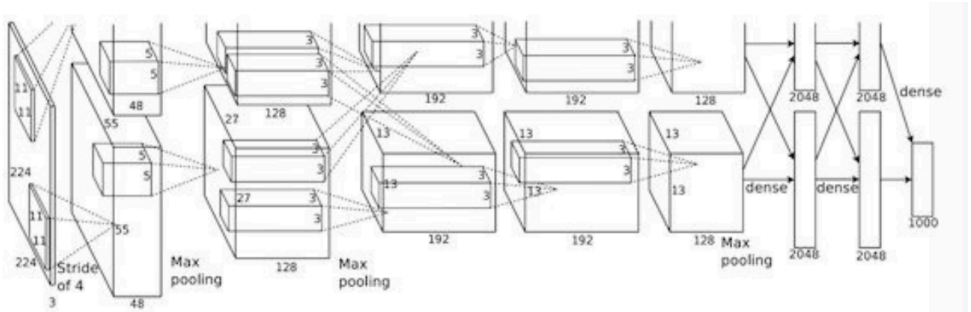
A standard CNN pipeline:



miniAlexNet, 2014

Complex networks are just an extension of this…

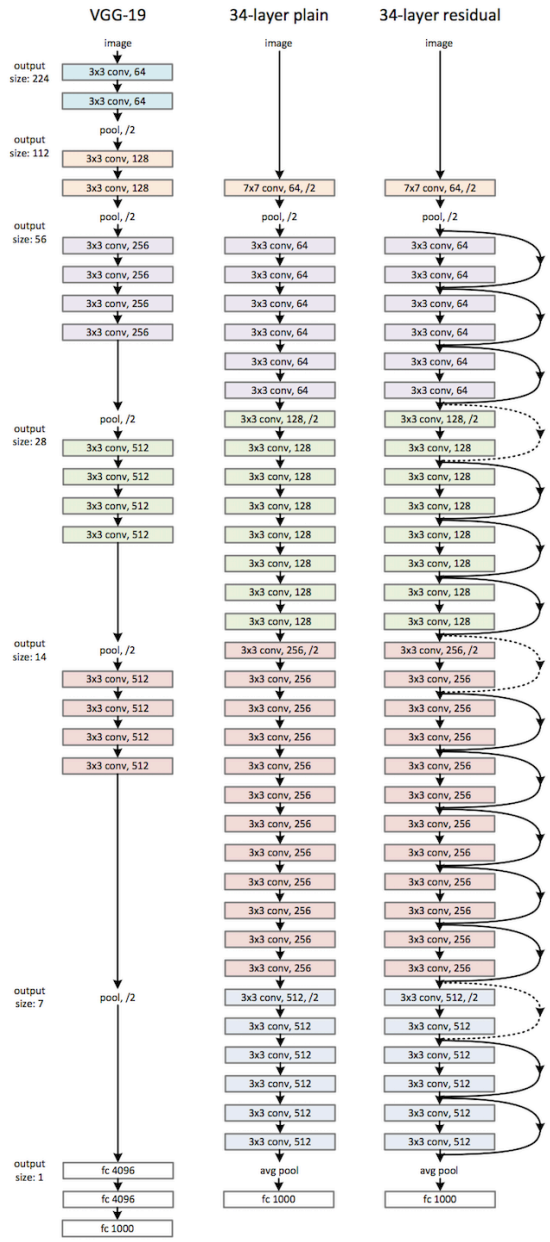
AlexNet (2012)



VGG (2014)

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

ResNet (2015)



Important components of a CNN

CNN Architecture

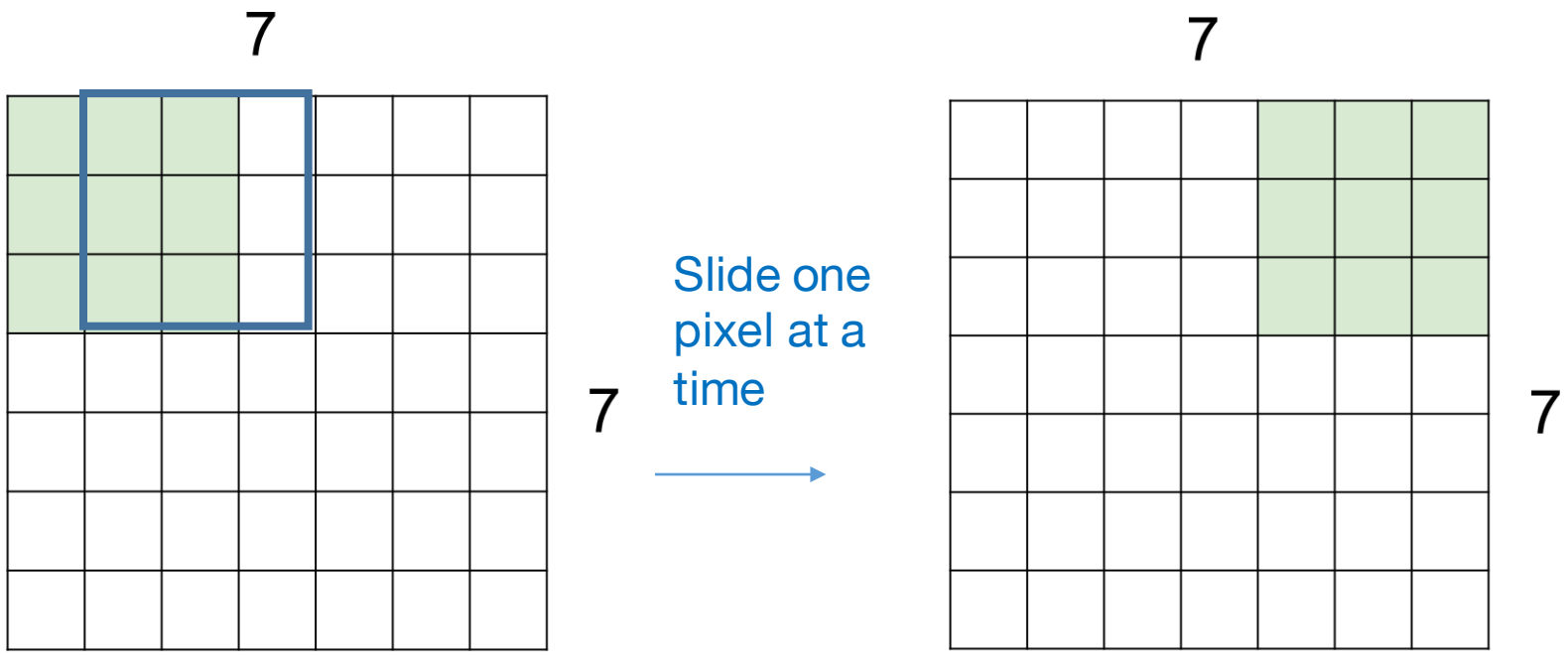
- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- Fully connected layers
- # of layers, dimensions per layer

Loss function & optimization

- Type of loss function
- Regularization
- Gradient descent method
- Gradient descent step size

Other specifics: Pre-processing**,** initialization**,** dropout, batch normalization, batch size

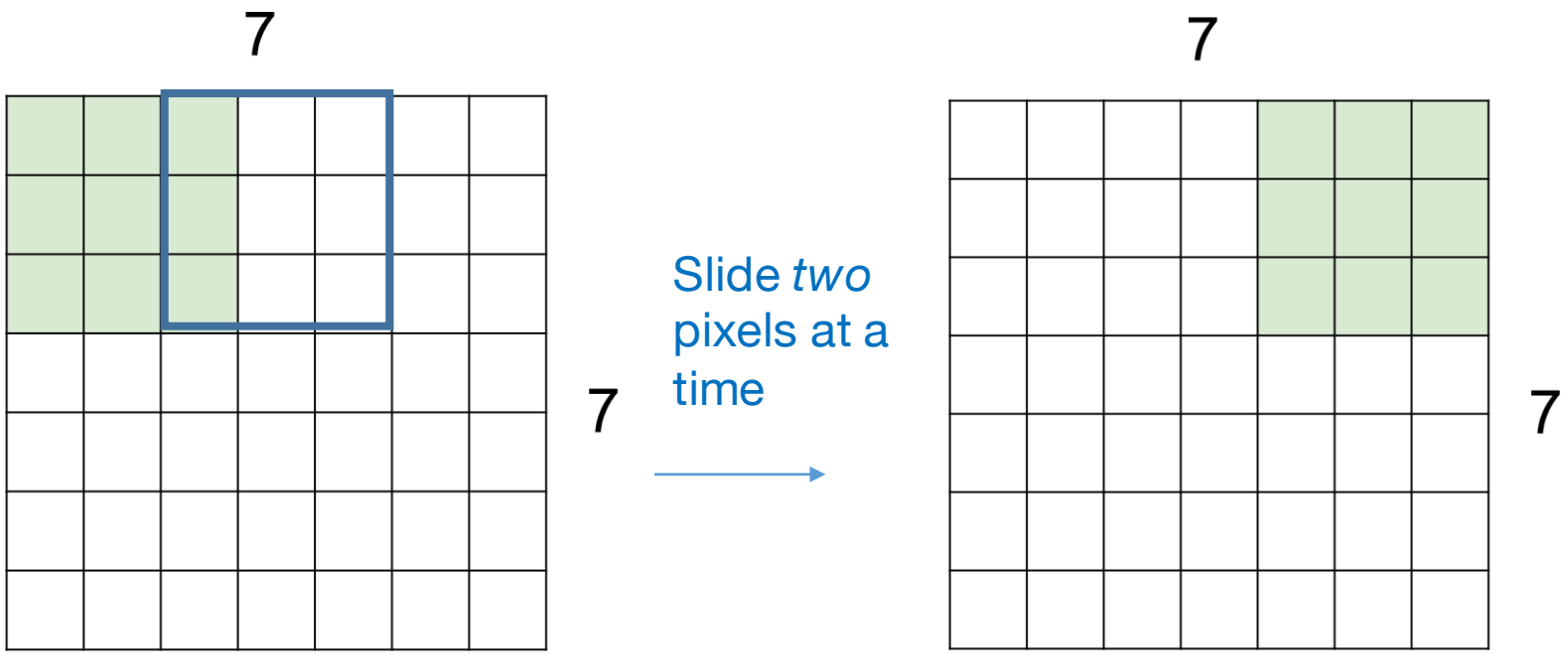
Convolutions: size, stride and padding



- 7x7 input image
- 3x3 filter

- 5x5 output

Convolutions: size, stride and padding

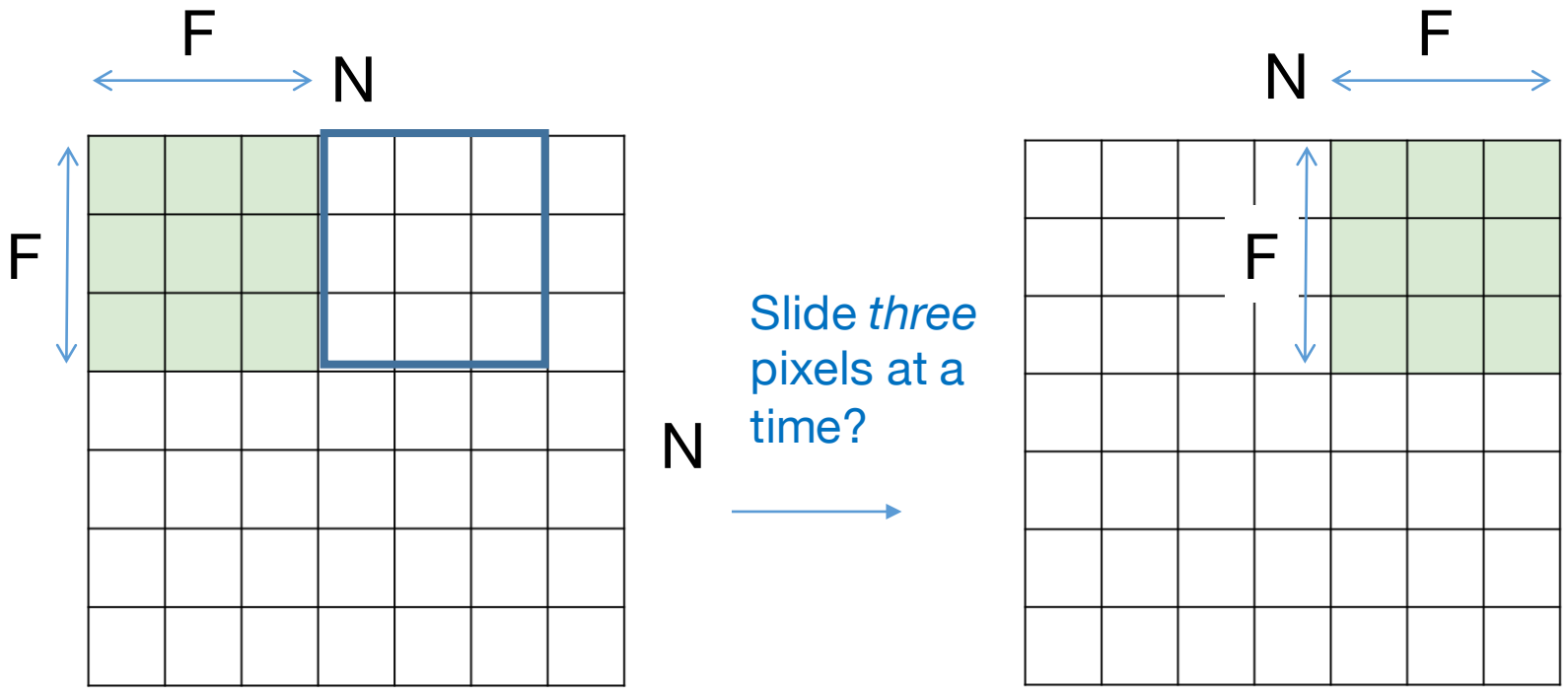


- 7x7 input image
- 3x3 filter

- 3x3 output!

This is called a “stride 2” convolution

Convolutions: size, stride and padding

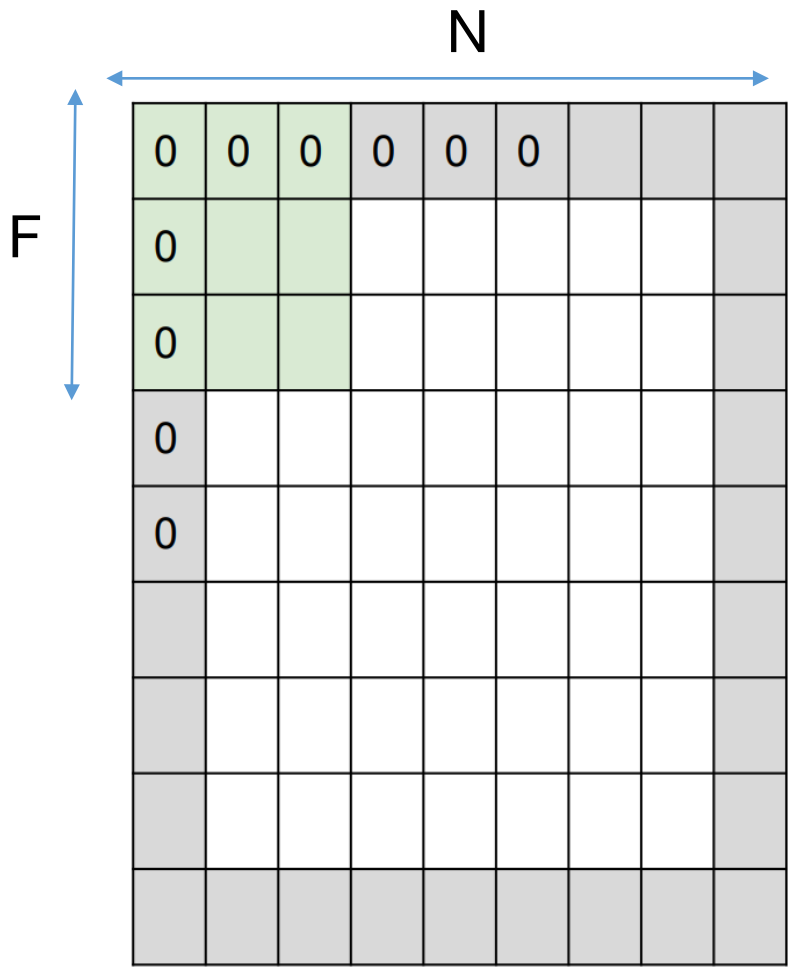


Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

This is called a “stride 3” convolution

Convolutions: size, stride and padding

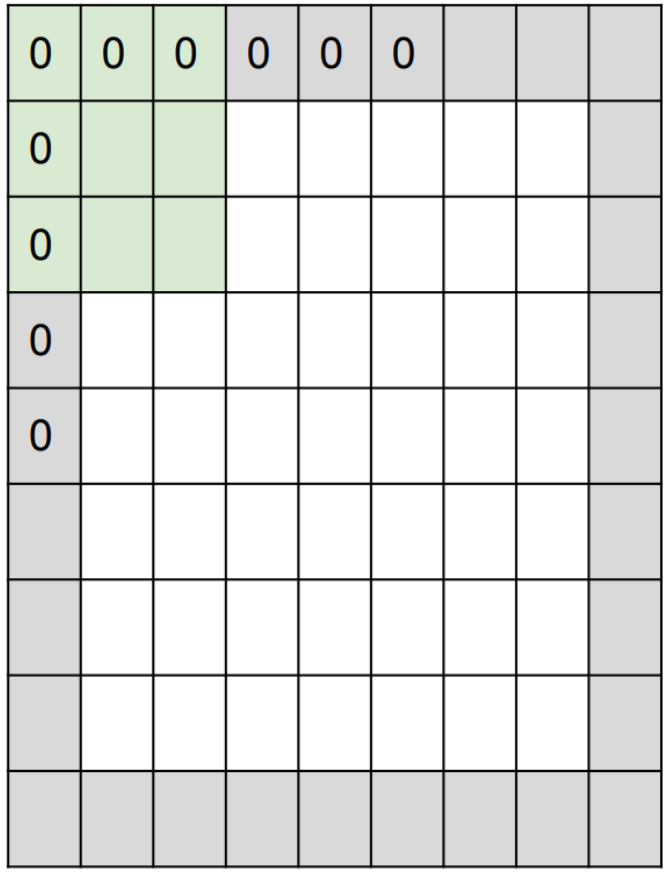


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

(recall:)
(N - F) / stride + 1

Padding: add zeros around edge of image

Convolutions: size, stride and padding



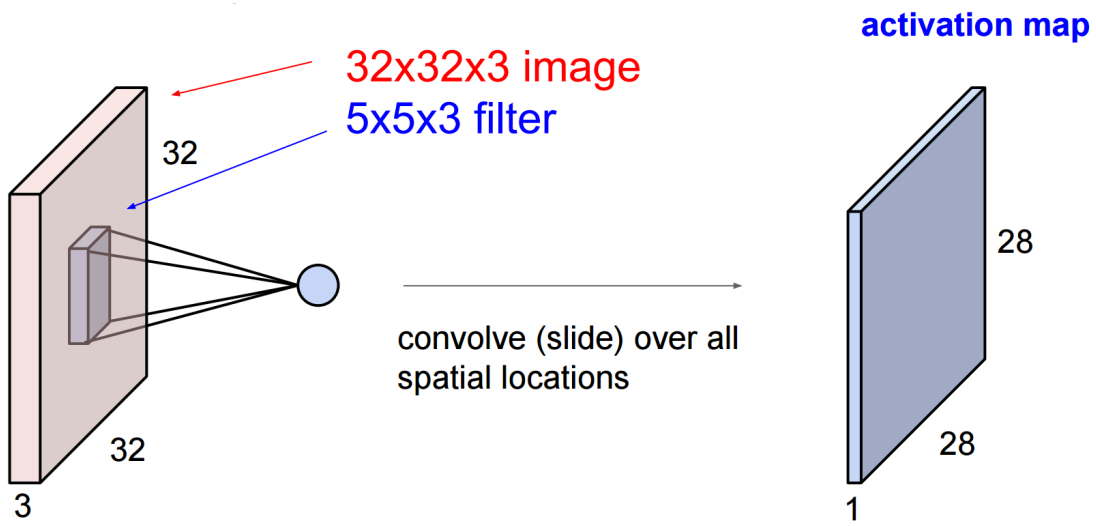
e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

Output = 7x7

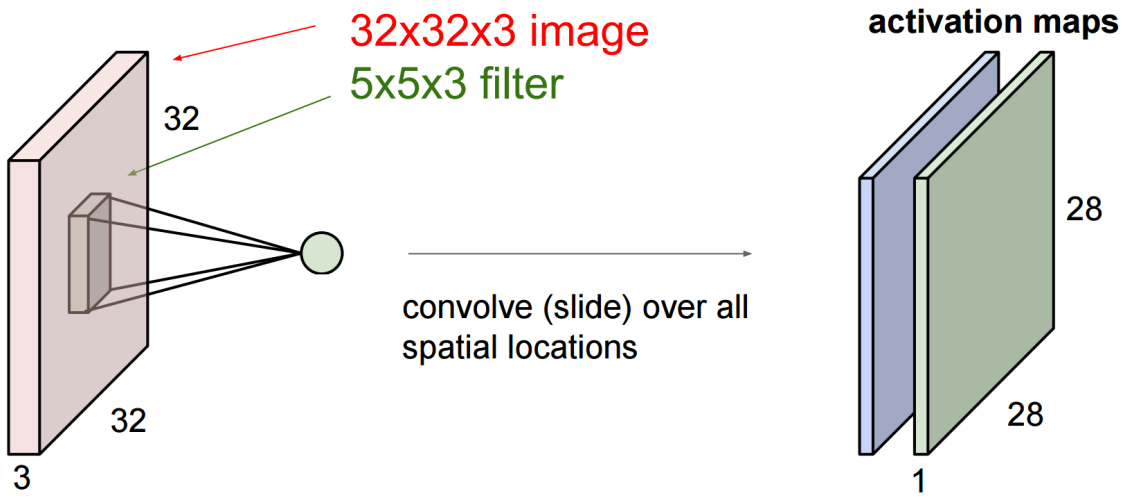
(recall:)
(N - F) / stride + 1

Padding: add zeros around edge of image

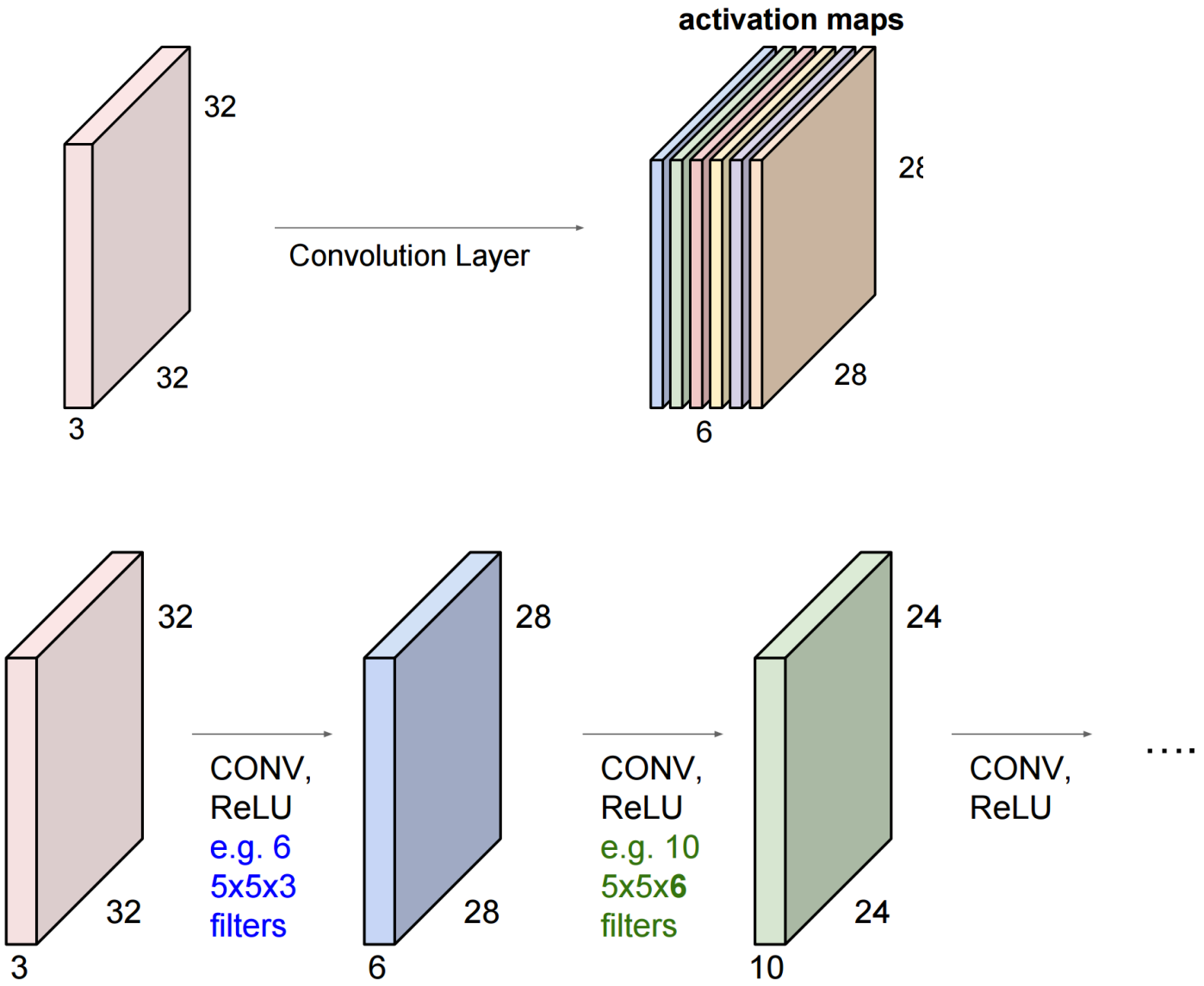
Convolution layer: learn multiple filters



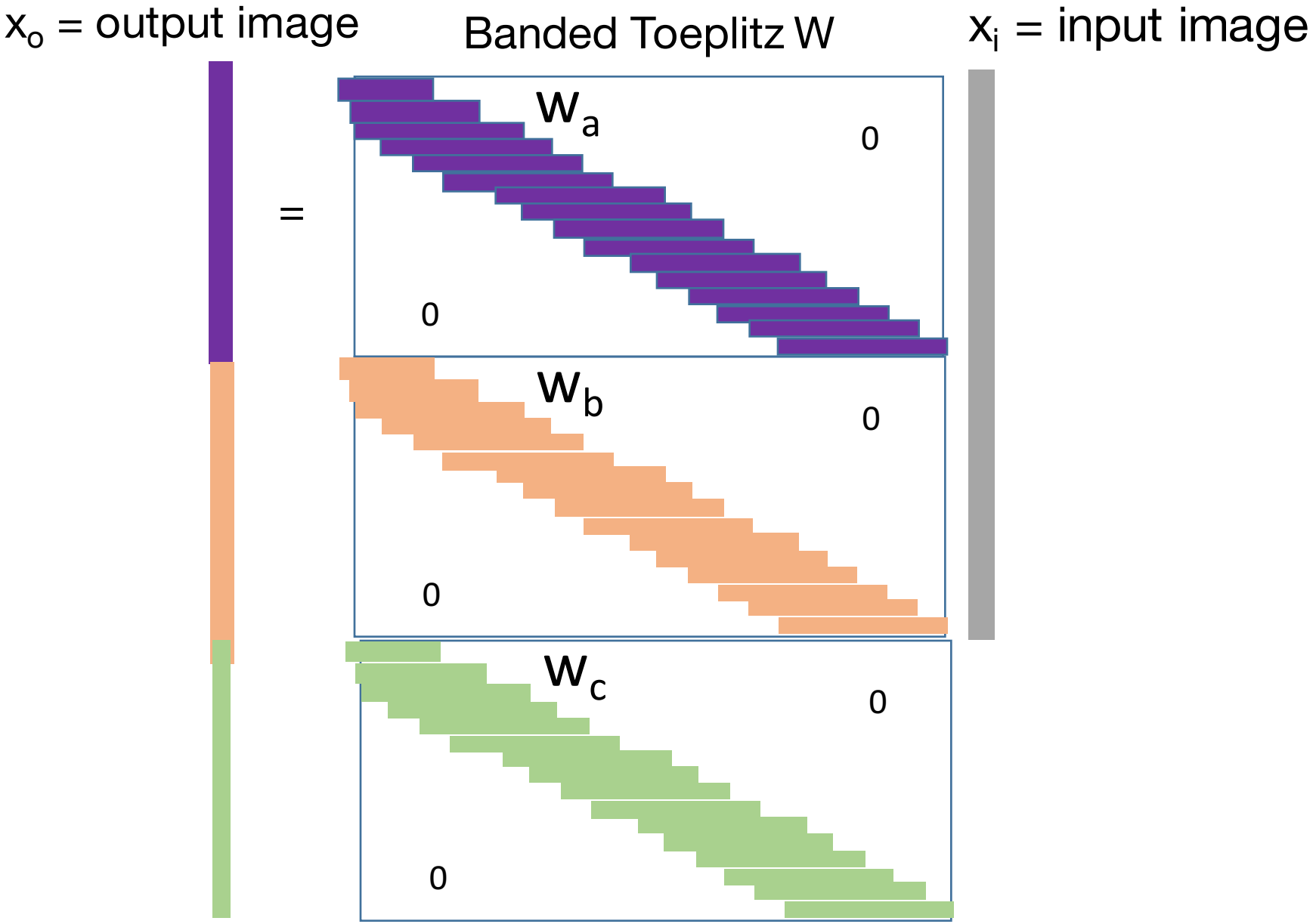
Convolution layer: learn multiple filters



Convolution layer: learn multiple filters



Summarize multiple filters with stacked matrices

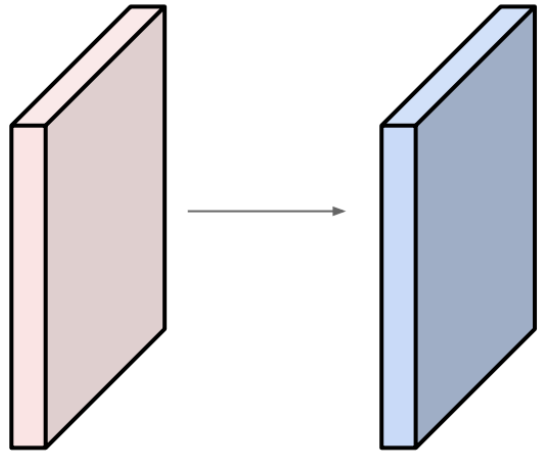


Convolution layer example mapping

Examples time:

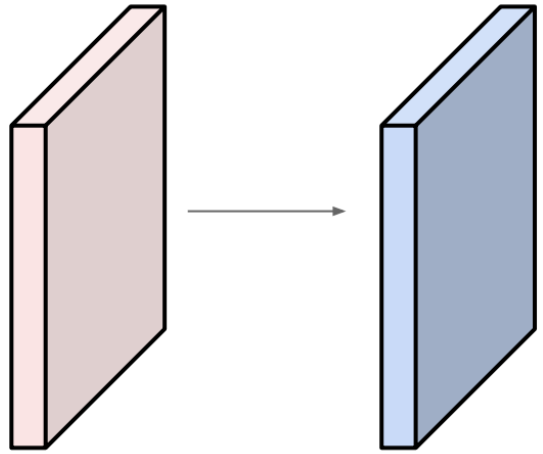
Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size: ?



Convolution layer example mapping

Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size: ?

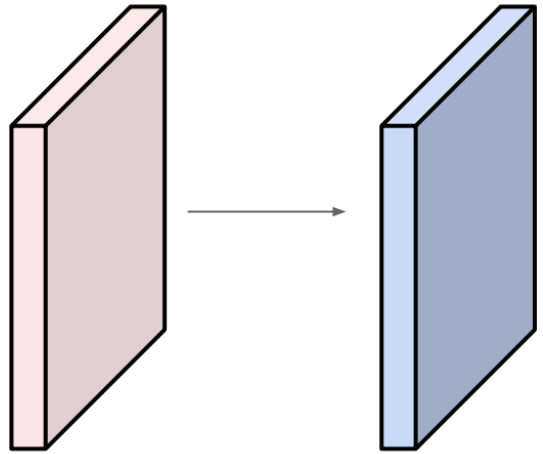
A: (N-F)/stride + 1 = (32+4-5)/1 + 1 = 32x32 spatial extent

So, output is **32x32x10**

Convolution layer example mapping

Examples time:

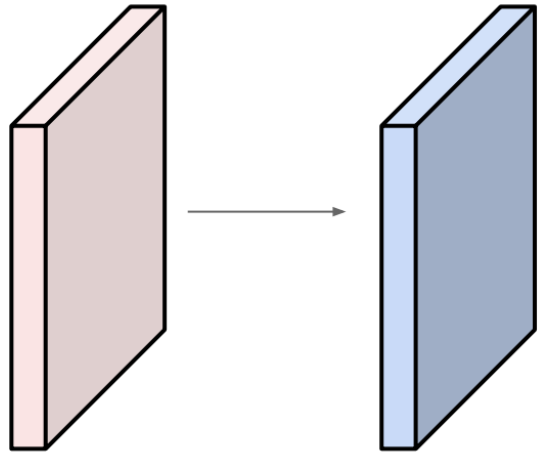
Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2



How many weights make up this transformation?

Convolution layer example mapping

Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

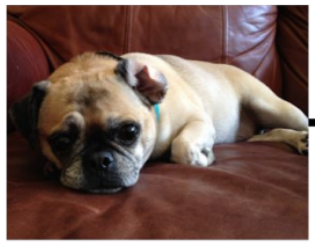
How many weights make up this transformation?

A: Each convolution filter: 5x5
1 offset parameter **b** per filter (**untied** biases)
Mapping 3 input layers to 10 output layers = 30 filters
Total: (5x5+1)*30 = **780**

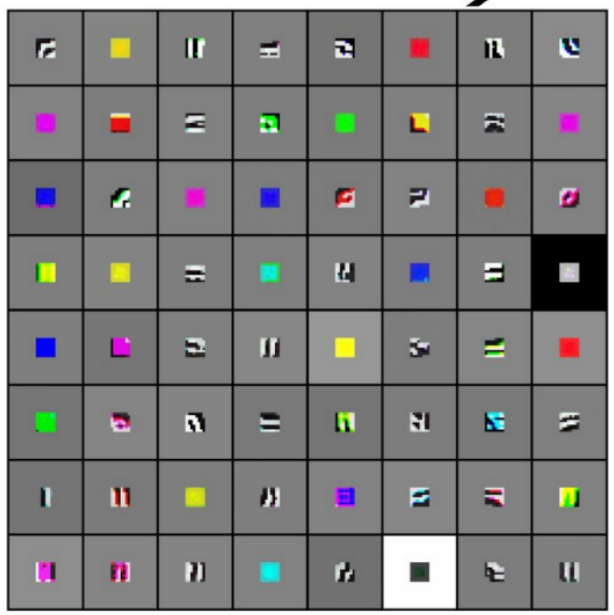
Preview

[Zeiler and Fergus 2013]

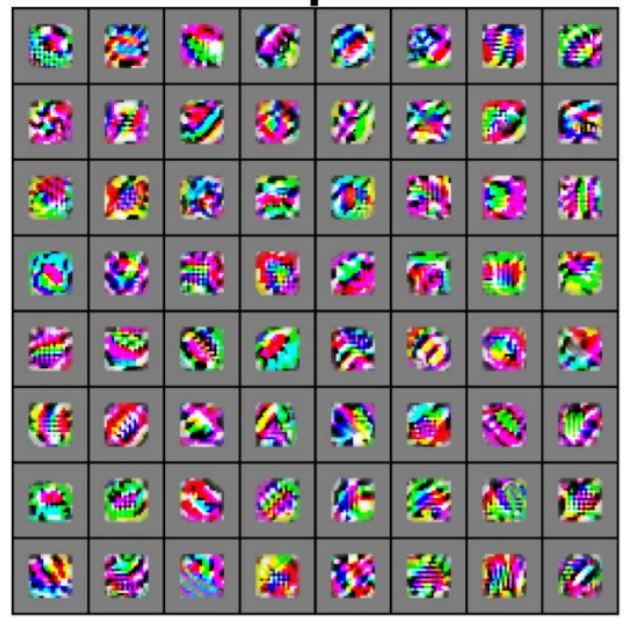
Visualization of VGG-16 by Lane McIntosh. VGG-16 architecture from [Simonyan and Zisserman 2014].



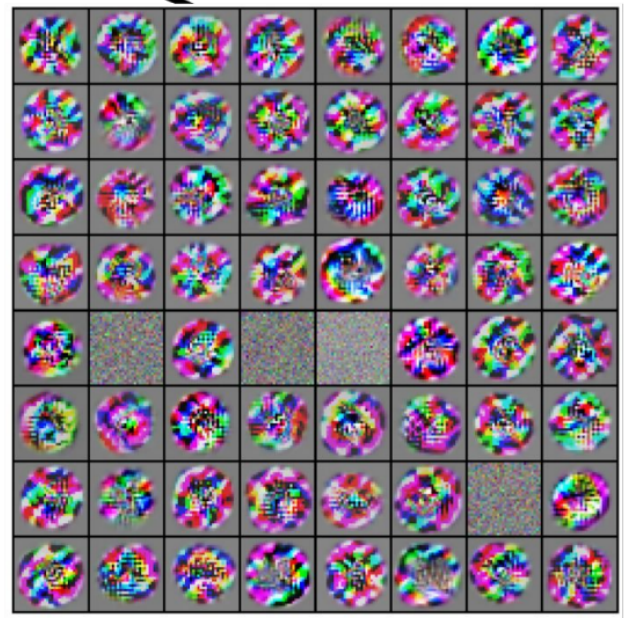
Low-level features → Mid-level features → High-level features → Linearly separable classifier



VGG-16 Conv1_1



VGG-16 Conv3_2



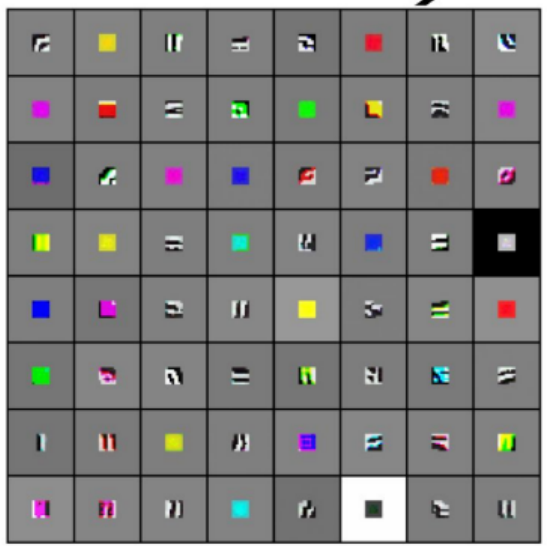
VGG-16 Conv5_3

Low-level features

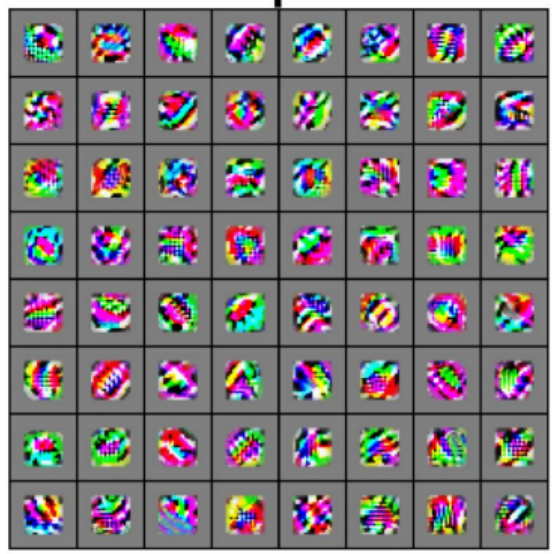
Mid-level features

High-level features

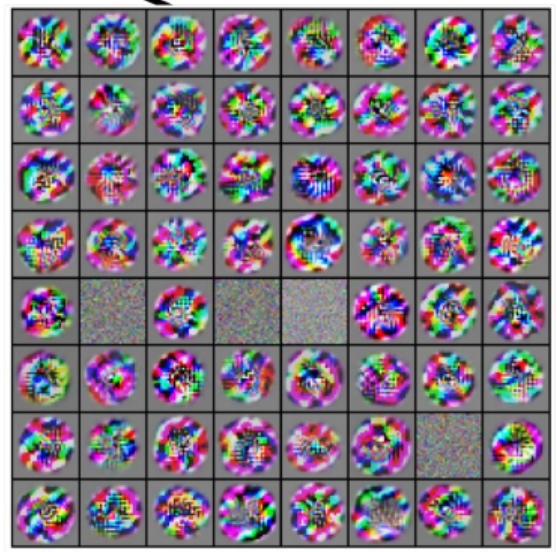
Linearly separable classifier



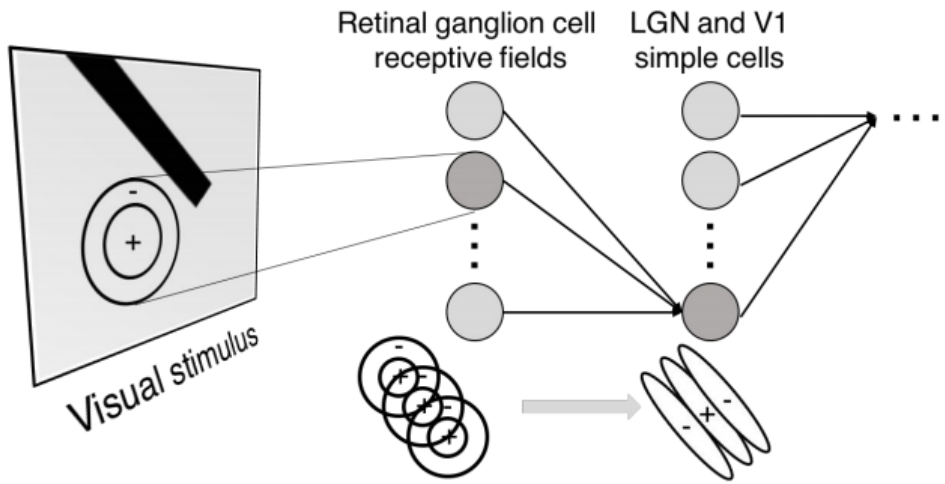
VGG-16 Conv1_1



VGG-16 Conv3_2

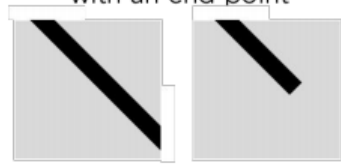


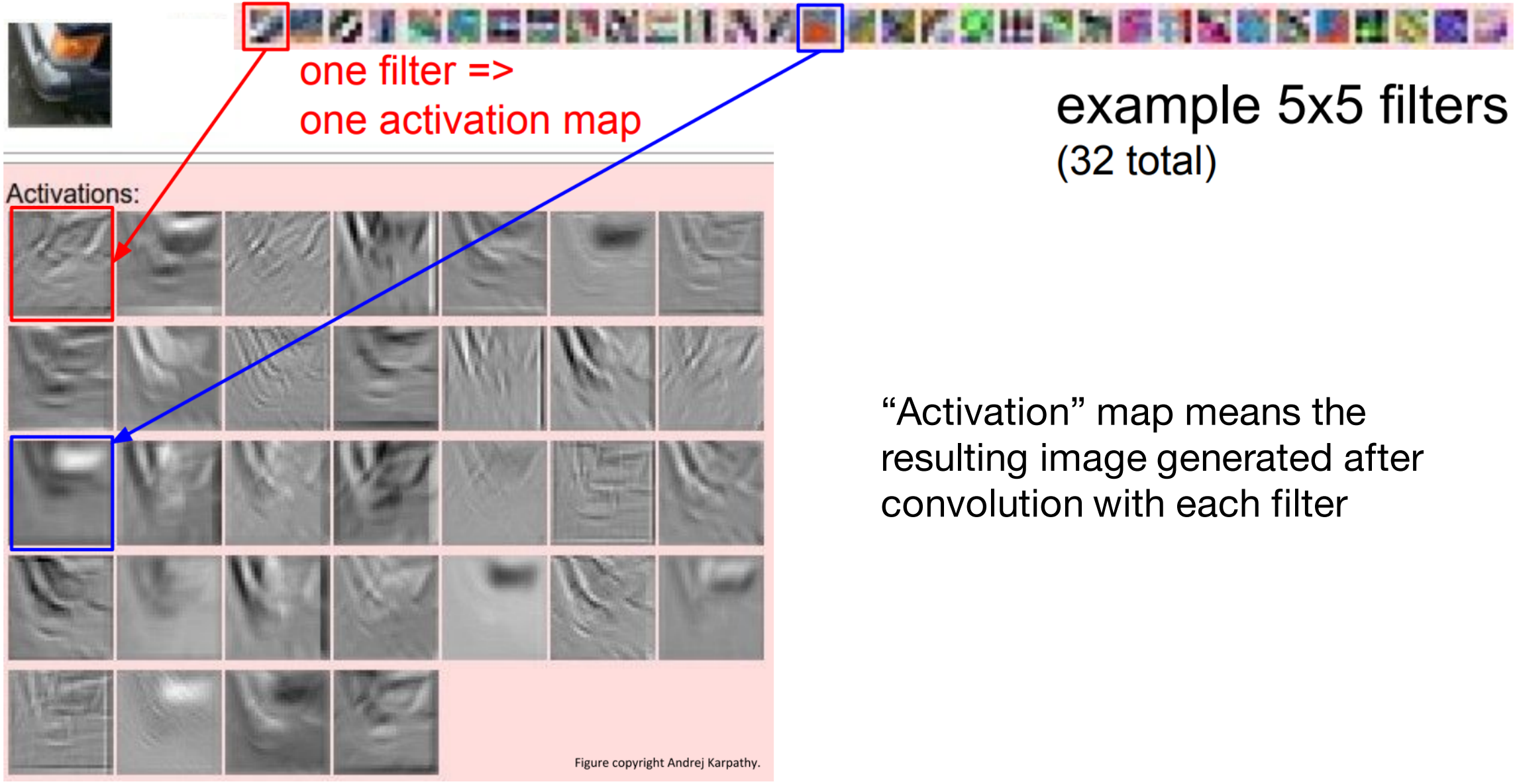
VGG-16 Conv5_3



Complex cells:
Response to light orientation and movement

Hypercomplex cells:
response to movement with an end point





example 5x5 filters
(32 total)

“Activation” map means the resulting image generated after convolution with each filter

Important components of a CNN

CNN Architecture

- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

Loss function & optimization

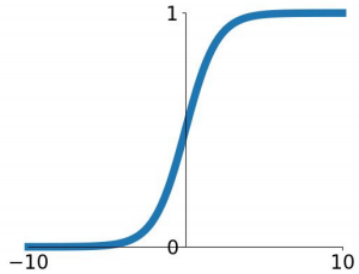
- Type of loss function
- Regularization
- Gradient descent method
- Gradient descent step size

Other specifics: Pre-processing**,** initialization**,** dropout, batch normalization, batch size

Non-linear “activation” functions

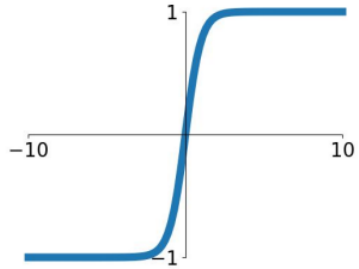
Sigmoid

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



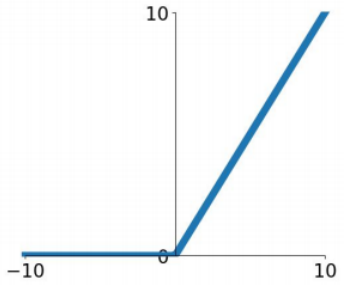
tanh

$$\tanh(x)$$



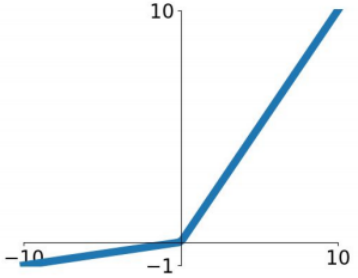
ReLU

$$\max(0, x)$$



Leaky ReLU

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

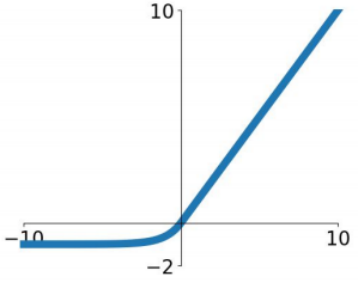


Maxout

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

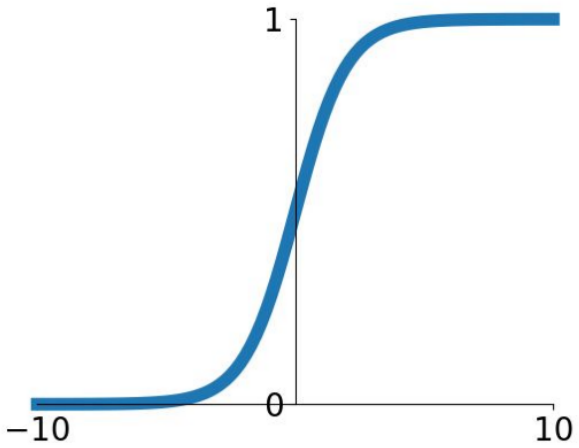
ELU

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



From Stanford CS231

Non-linear “activation” functions

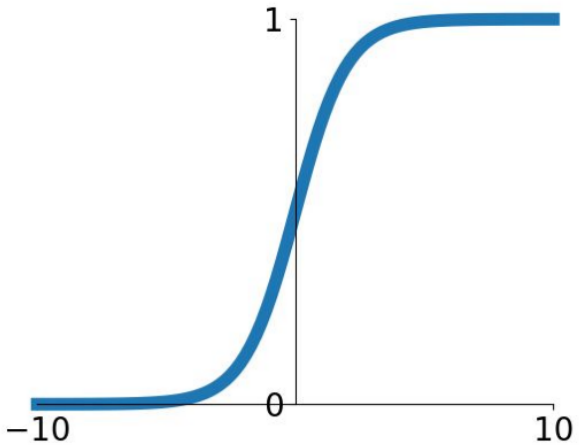


Sigmoid

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

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

Non-linear “activation” functions



Sigmoid

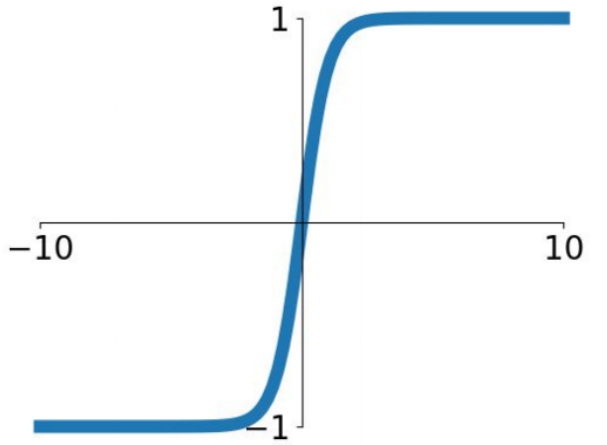
$$\sigma(x) = 1/(1 + e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

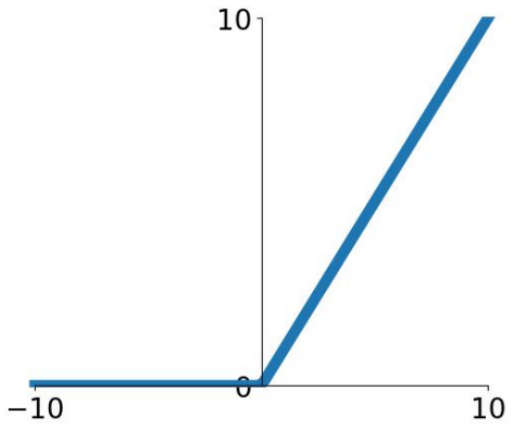
Non-linear “activation” functions



tanh(x)

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

Non-linear “activation” functions



ReLU
(Rectified Linear Unit)

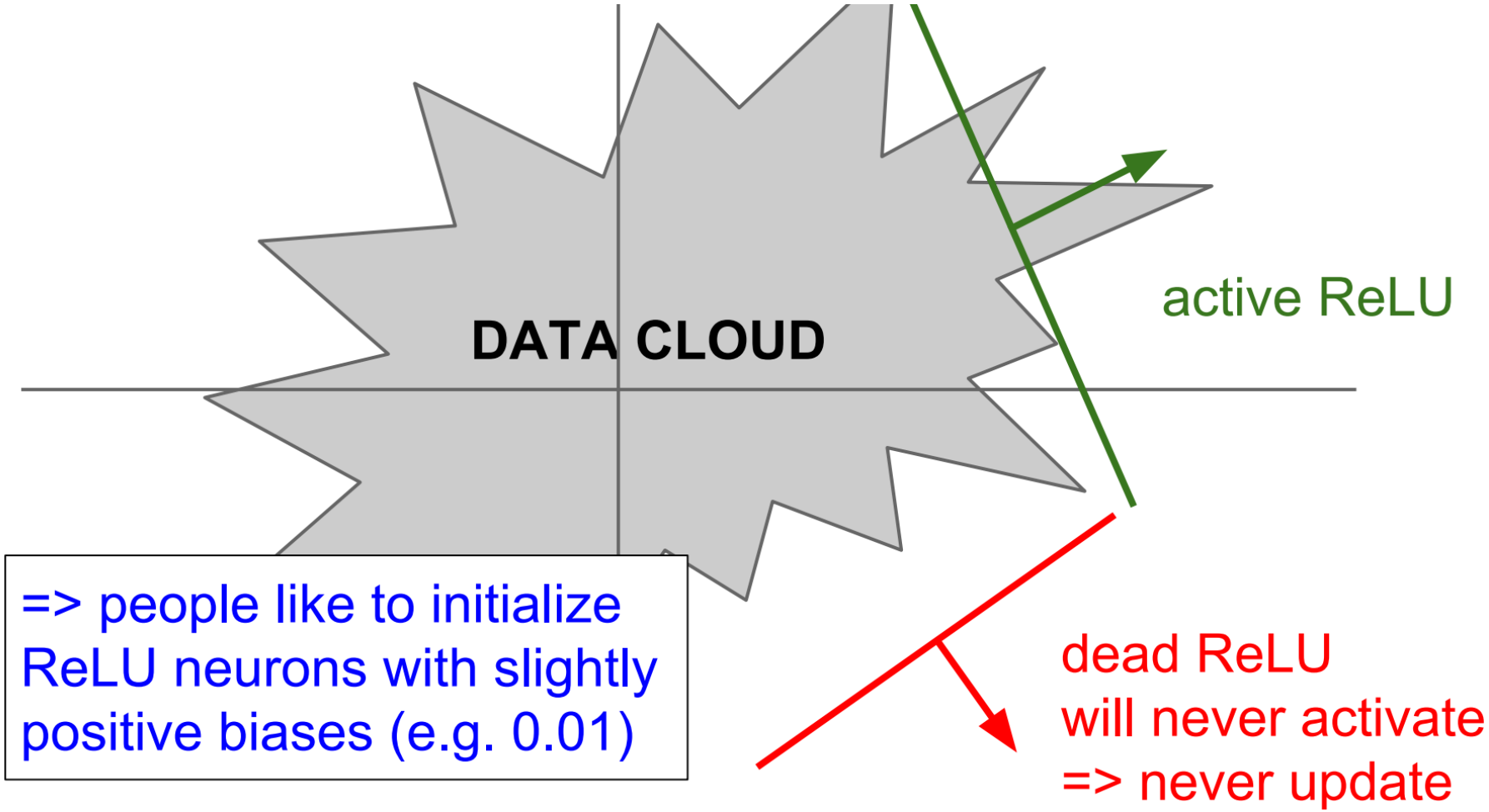
Computes **f(x) = max(0,x)**

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

- Not zero-centered output
- An annoyance:

hint: what is the gradient when x < 0?

Non-linear “activation” functions



Important components of a CNN

CNN Architecture

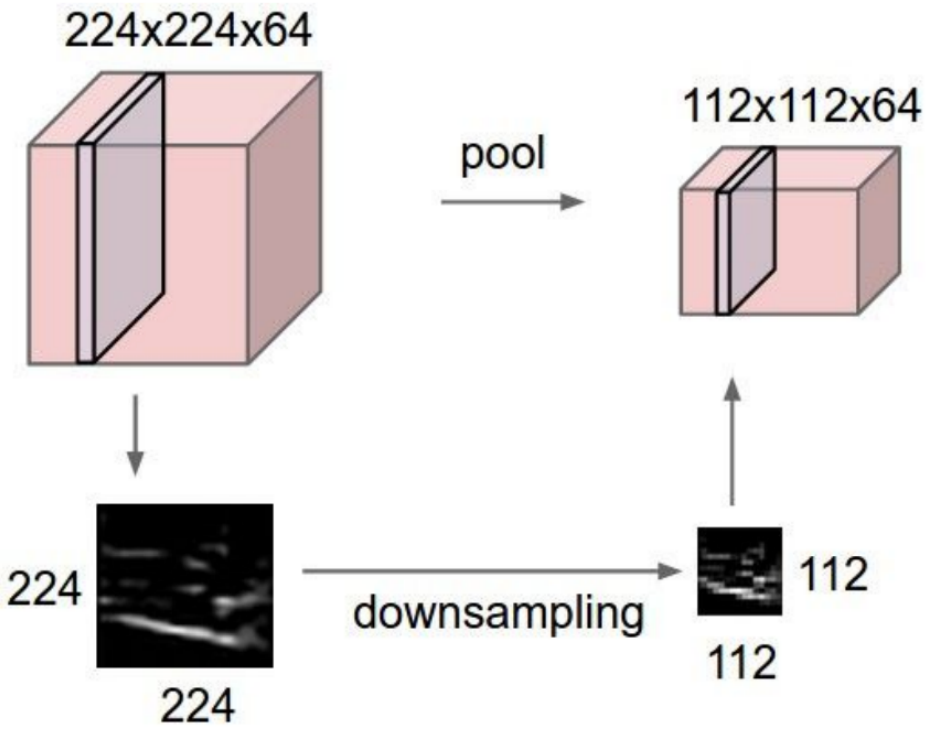
- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

Loss function & optimization

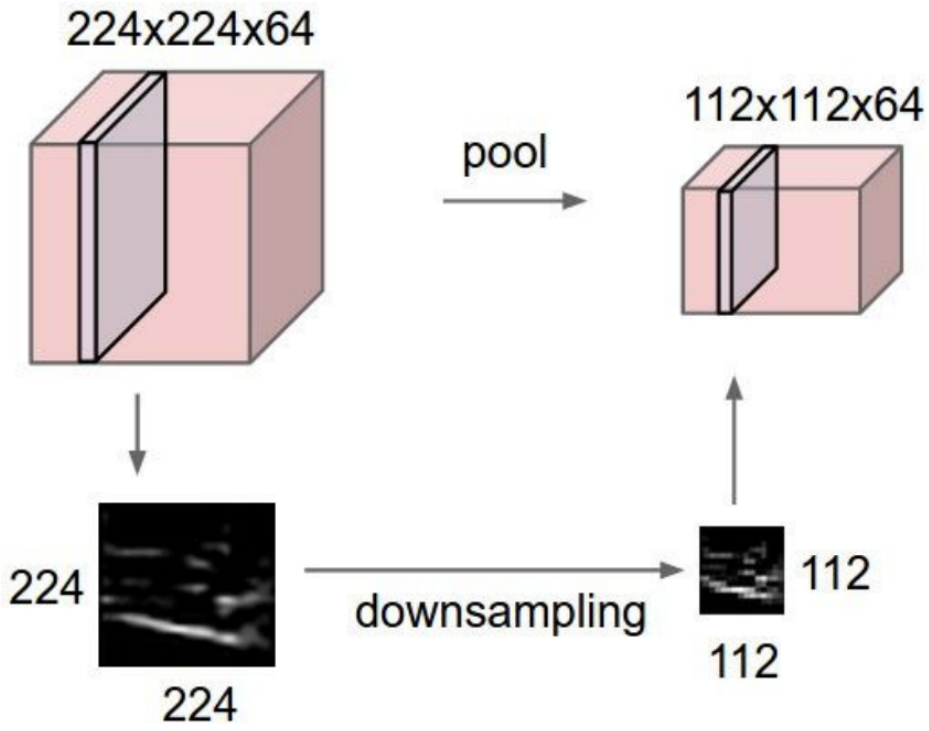
- Type of loss function
- Regularization
- Gradient descent method
- SGD batch and step size

Other specifics: Pre-processing, initialization, dropout, batch normalization, augmentation

Pooling operation – reduce the size of data cubes along space

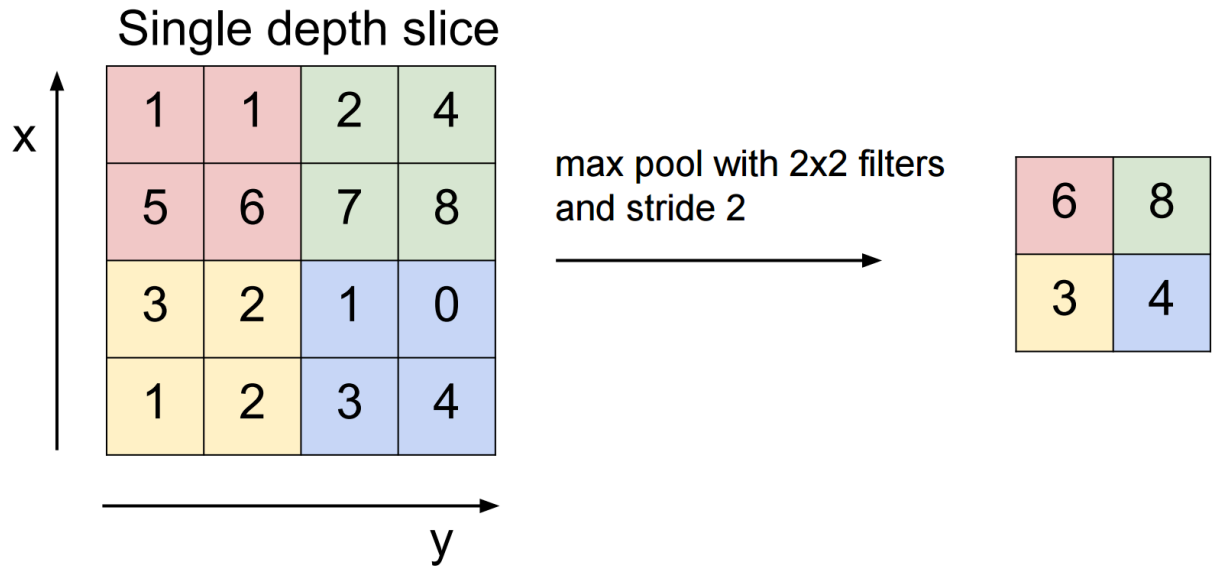


Pooling operation – reduce the size of data cubes along space



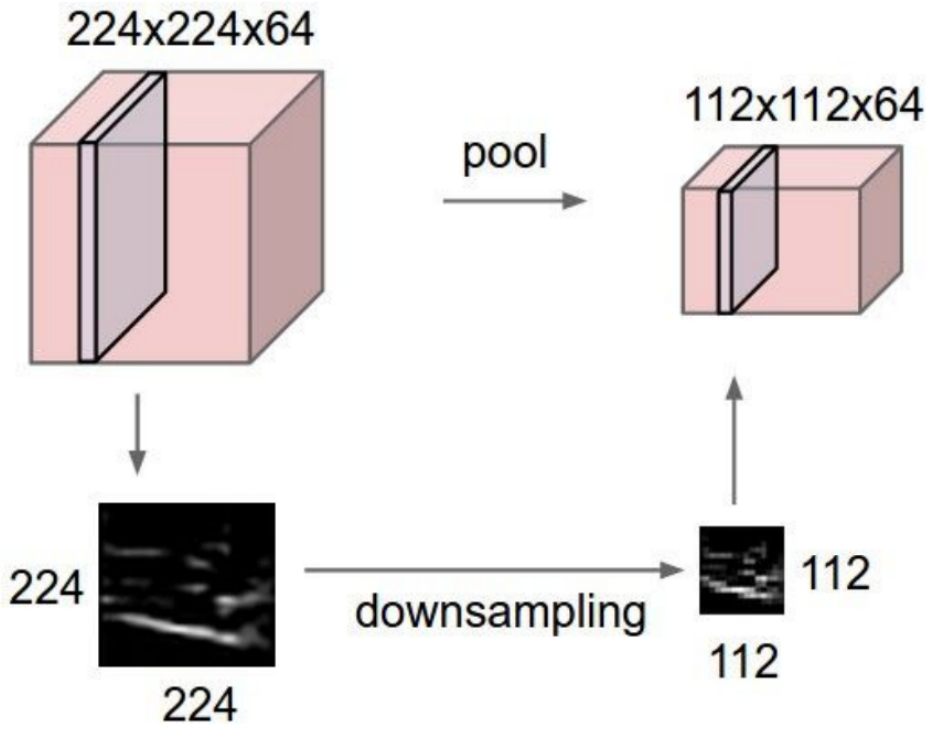
Common option #1:

MAX POOLING



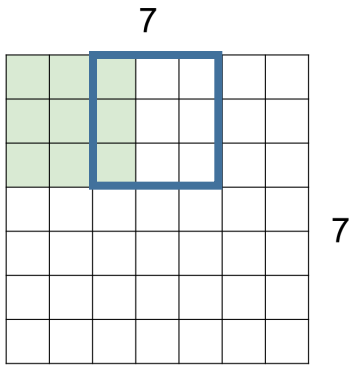
Related options: Sum pooling, mean pooling

Pooling operation – reduce the size of data cubes along space

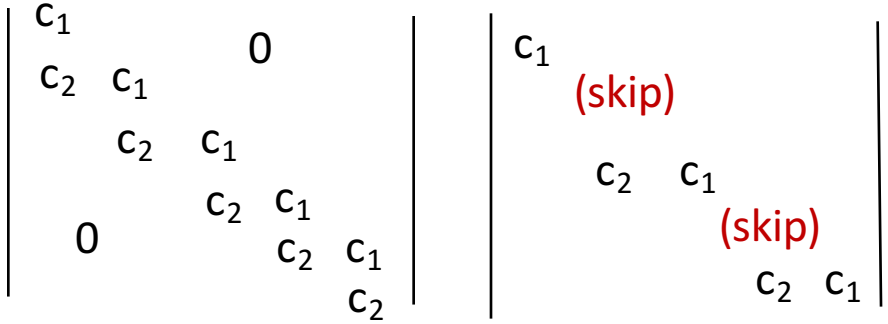


Common option #2: just use bigger strides

STRIDE = 2



7x7 input -> 3x3 output



Important components of a CNN

CNN Architecture

- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

Let's view some code!

Loss function & optimization

- Type of loss function
- Regularization
- Gradient descent method
- SGD batch and step size

Other specifics: Pre-processing, initialization, dropout, batch normalization, augmentation

Important components of a CNN

CNN Architecture

- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

Loss function & optimization

- Type of loss function
- Regularization
- Gradient descent method
- SGD batch and step size

Other specifics: Pre-processing, initialization, dropout, batch normalization, augmentation

Common loss functions used for CNN optimization

- Cross-entropy loss function
  - Softmax cross-entropy – use with single-entry labels
  - Weighted cross-entropy – use to bias towards true pos./false neg.
  - Sigmoid cross-entropy
  - KL Divergence
- Pseudo-Huber loss function
- L1 loss loss function
- MSE (Euclidean error, L2 loss function)
- Mixtures of the above functions

Important components of a CNN

CNN Architecture

- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

Loss function & optimization

- Type of loss function
- Regularization
- Gradient descent method
- SGD batch and step size

Other specifics: Pre-processing, initialization, dropout, batch normalization, augmentation

Regularization – the basics

λ = regularization strength (hyperparameter)

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Simple examples

L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Regularization prefers less complex models & help avoids overfitting

$$x = [1, 1, 1, 1]$$

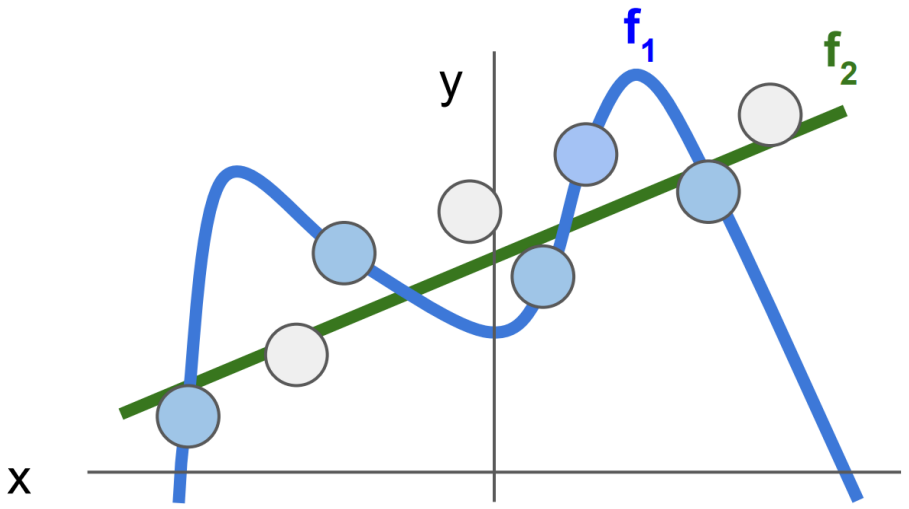
$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$w_1^T x = w_2^T x = 1$$

L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$



Regularization pushes against fitting the data *too* well so we don't fit noise in the data

Important components of a CNN

CNN Architecture

- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

Loss function & optimization

- Type of loss function
- Regularization
- Gradient descent method
- SGD batch and step size

Very quick outline – details next class!

Other specifics: Pre-processing, initialization, dropout, batch normalization, augmentation

A variety of gradient descent solvers available in Tensorflow

- Stochastic Gradient Descent

Implementation detail #1 – method for gradient descent

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Stochastic Gradient Descent (SGD)

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Full sum expensive when N is large!

Approximate sum using a **minibatch** of examples
32 / 64 / 128 common

Implementation detail #1 – method for gradient descent

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Stochastic Gradient Descent (SGD)

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Full sum expensive when N is large!

Approximate sum using a **minibatch** of examples
32 / 64 / 128 common

A variety of gradient descent solvers available in Tensorflow

- Stochastic Gradient Descent (bread-and-butter, when in doubt…)
- Adam Optimizer (update learning rates with mean and variance)
- Nesterov / Momentum (add a velocity term)
- AdaGrad (Adaptive Subgradients, change learning rates)
- Proximal AdaGrad (Proximal = solve second problem to stay close)
- Ftrl Proximal (Follow-the-regularized-leader)
- AdaDelta (Adaptive learning rate)