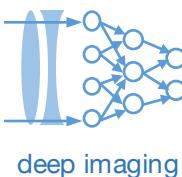


Lecture 9: Ingredients for a convolutional neural network

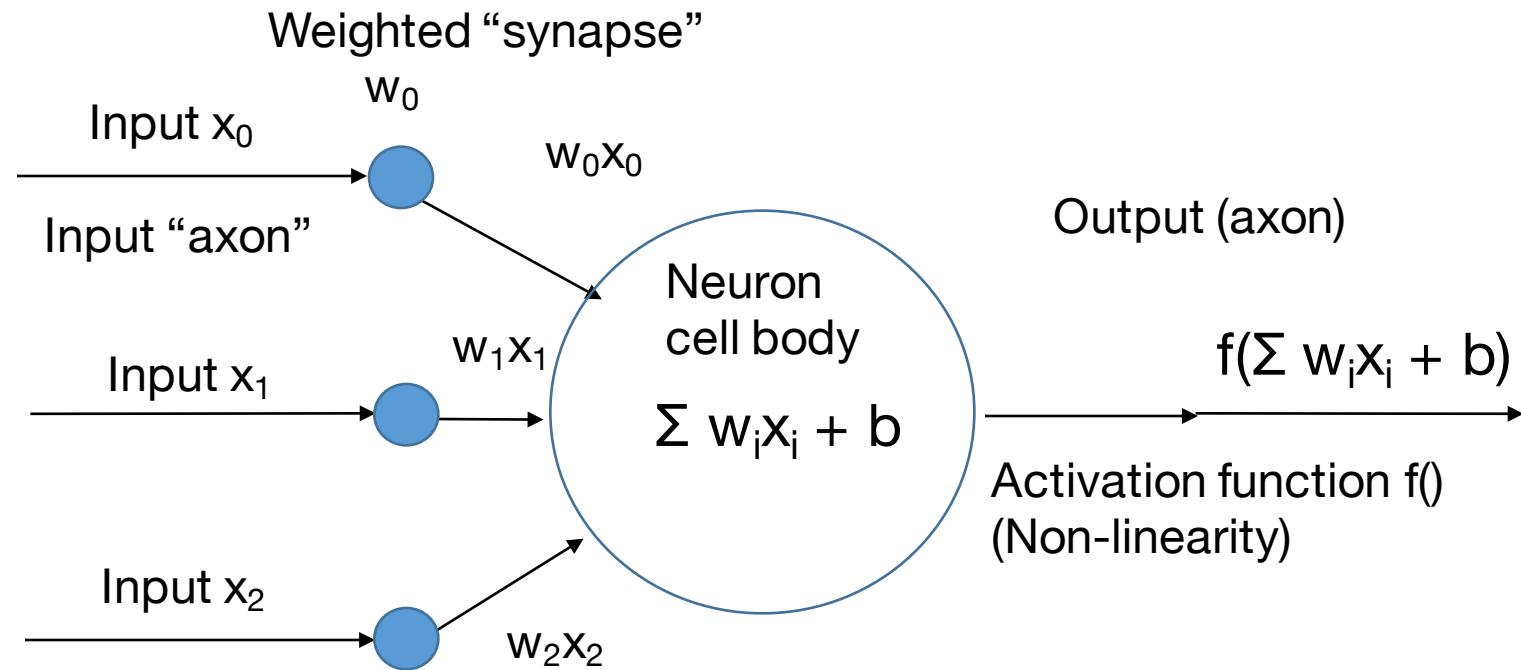
Machine Learning and Imaging

BME 590L
Roarke Horstmeyer

Note: Much material borrowed from Stanford CS231n, Lectures 4 - 10

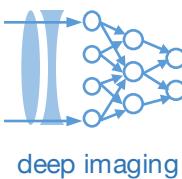


No more asides! Now lets get into neural networks...

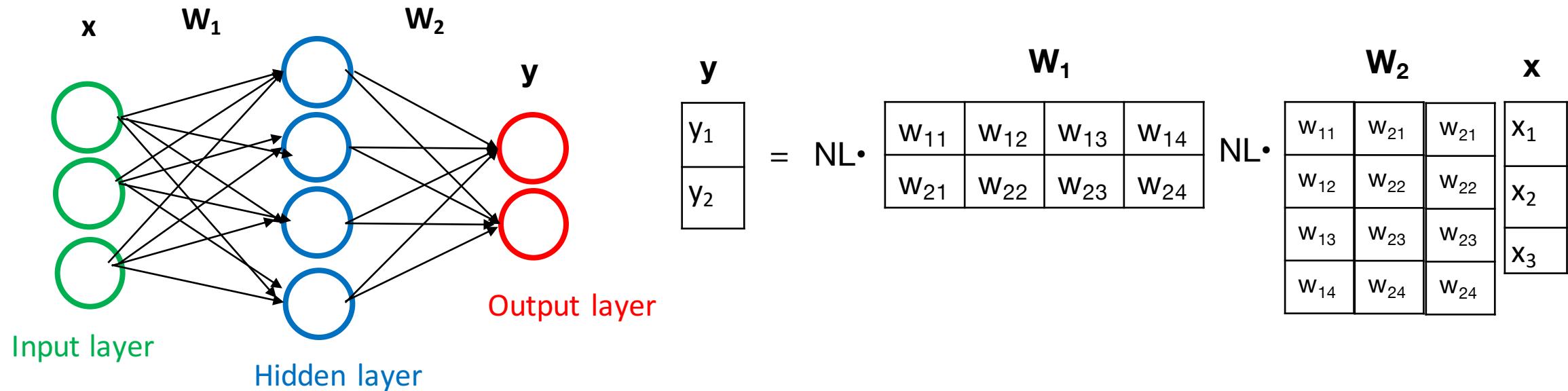


Single ”neuron”:
Inner product of
inputs \mathbf{x} with learned
weights \mathbf{w} & non-
linearity afterwards

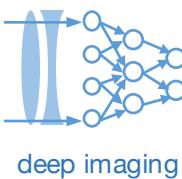
- Multiple weighted inputs: $\mathbf{x} \rightarrow \mathbf{y} = \mathbf{w}^T \mathbf{x}$ is “dendrites into cell body”
- Non-linearity $f()$ after sum = “neuron’s activation function” (loose interp.)



No more asides! Now lets get into neural networks...

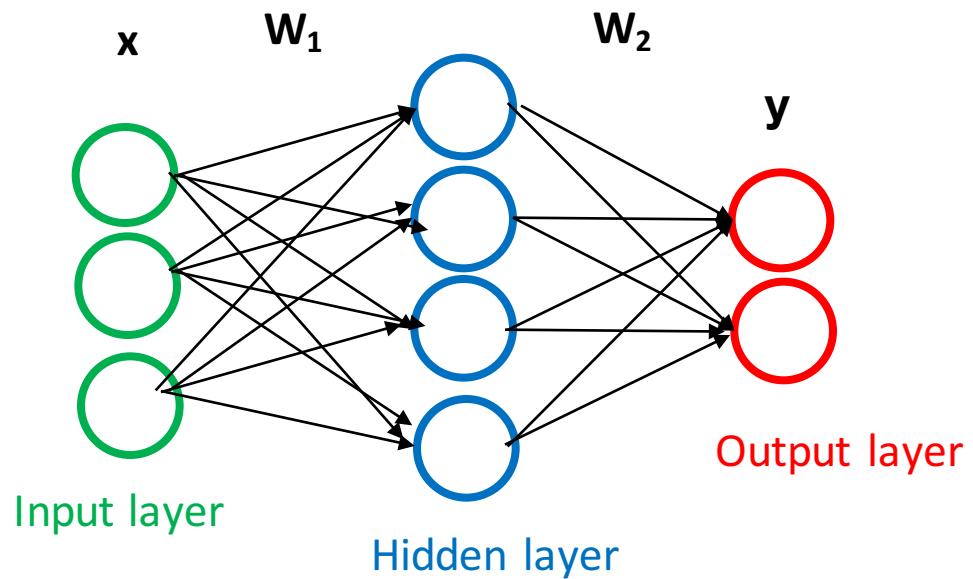


- For multiple cells (units), use matrix \mathbf{W} to connect inputs to outputs
- These cascade in layers

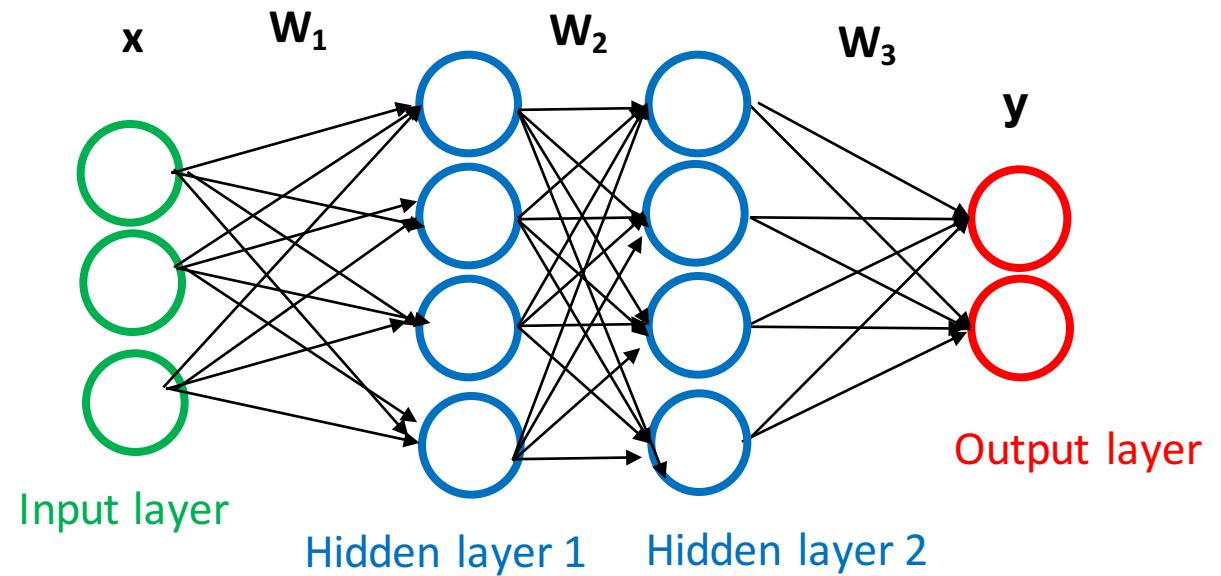


Neural networks = cascaded set of matrix multiplies and non-linearities

2-layer network:

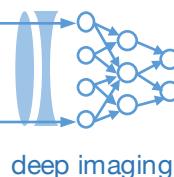


3-layer network:

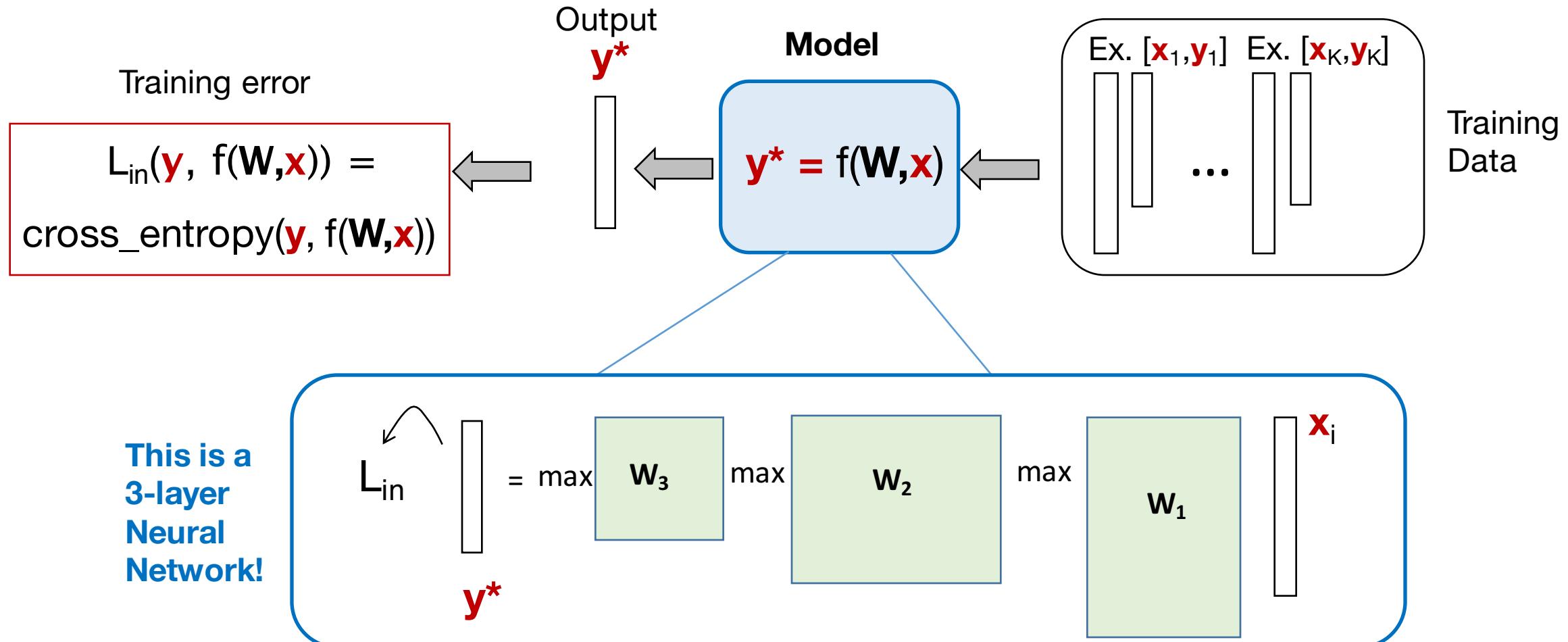


or 3-layer Neural Network

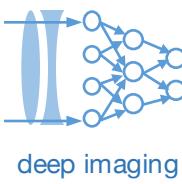
$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$



Our very basic convolutional neural network

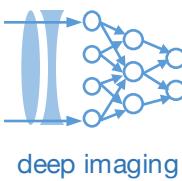


Forward pass: from x_i and current W 's, find L_{in}



Insight: Do we really need to mix every image pixel with every other image pixel to start?

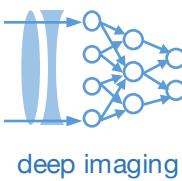




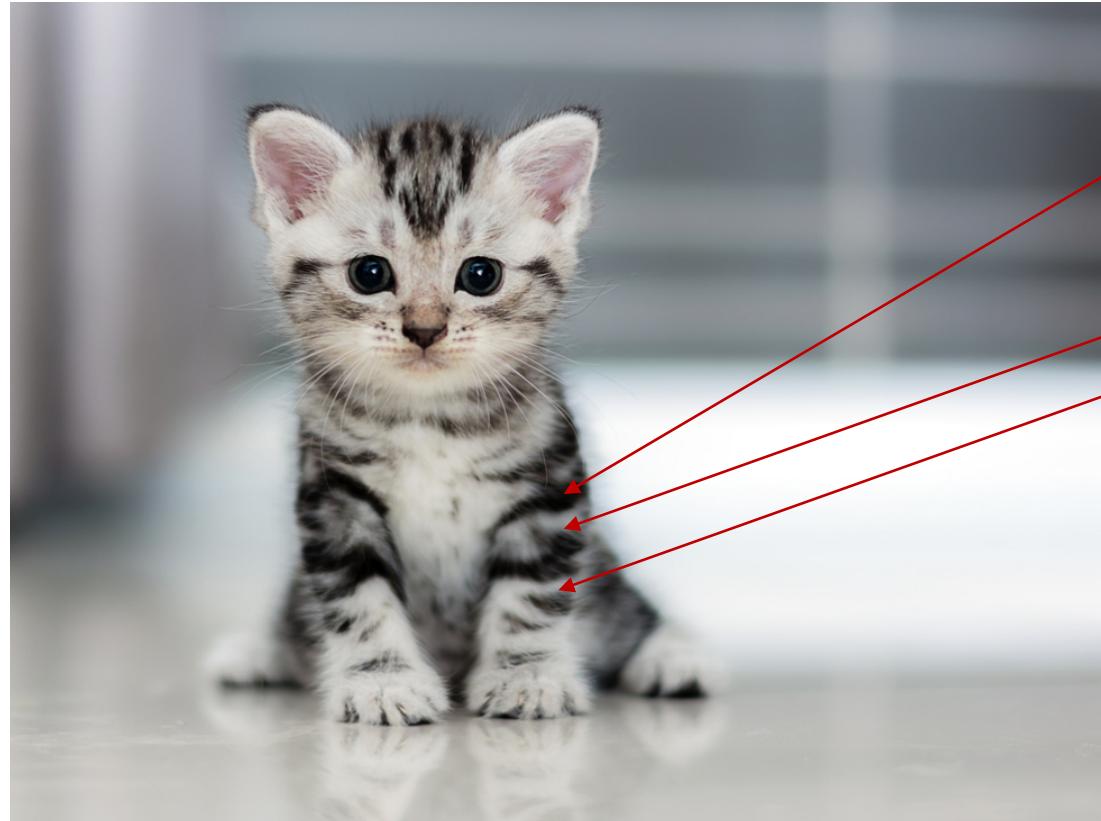
Insight: Do we really need to mix every image pixel with every other image pixel to start?

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

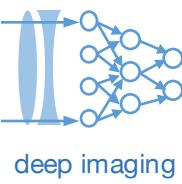




Insight: Do we really need to mix every image pixel with every other image pixel to start?



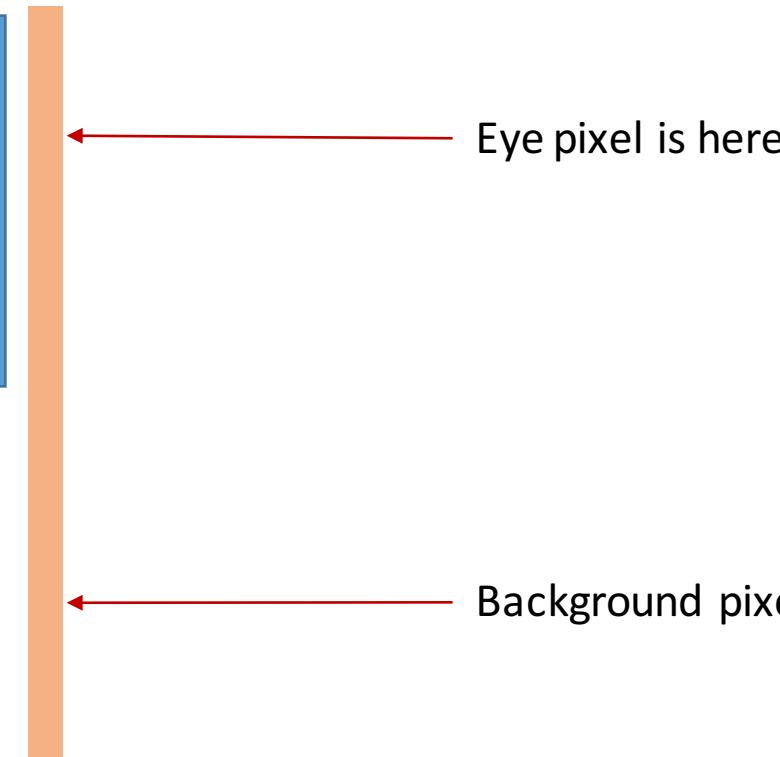
But understanding the stripes in these 3 pixels right near each other is going to be pretty helpful...

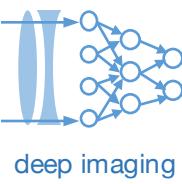


$x = \text{cat image}$

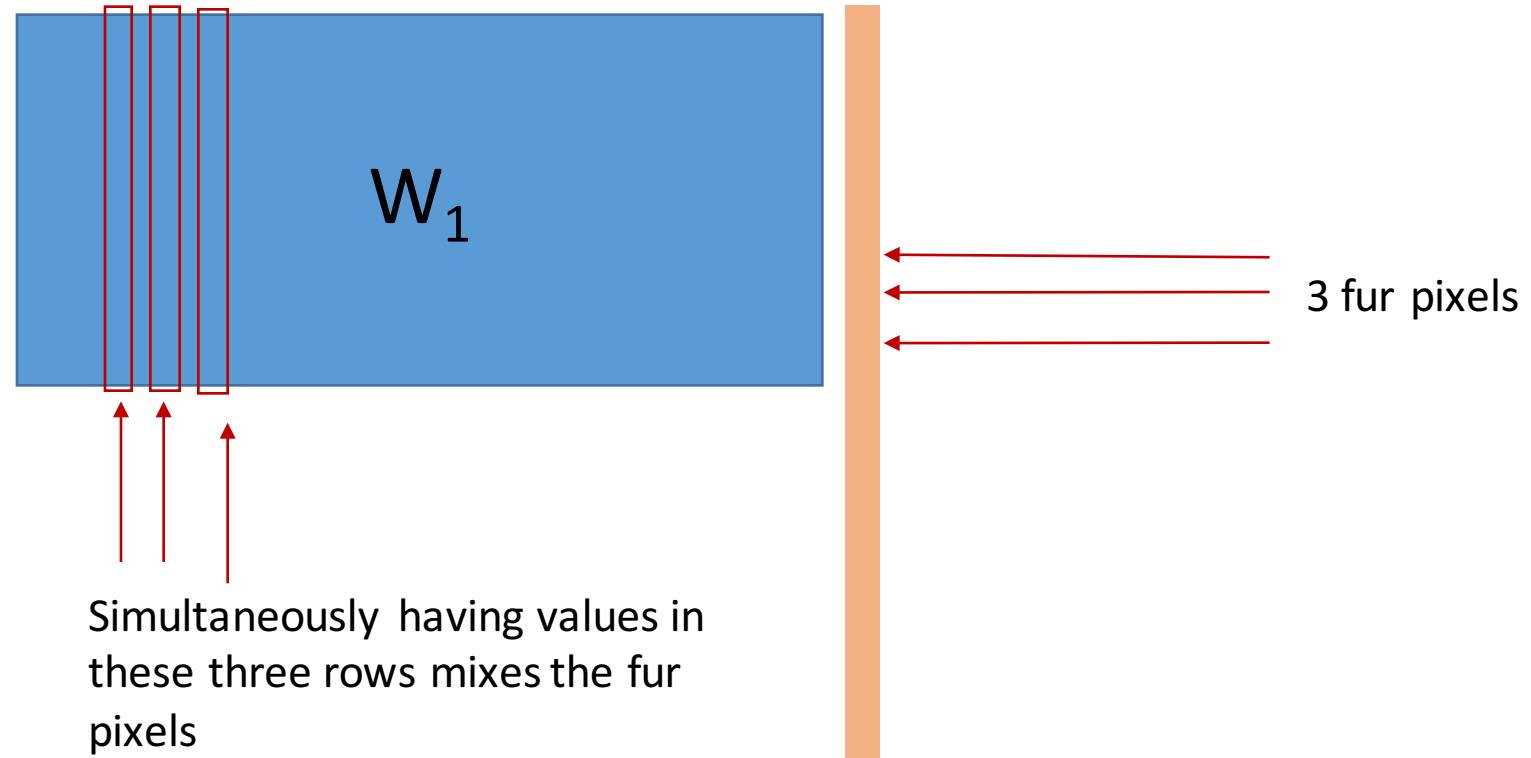


Simultaneously having values in
these two rows means the eye and
the background are “mixing”



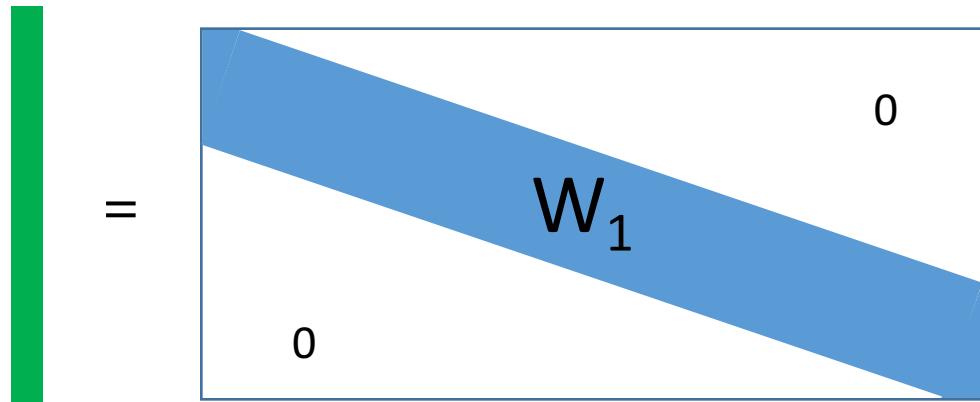


$x = \text{cat image}$



S

= Banded W



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: $k \cdot O(n)$

S

$$= \begin{matrix} & \text{Banded } W \\ \begin{matrix} & \\ & \text{---} \\ & k \\ & \text{---} \\ & \end{matrix} & \begin{matrix} & \\ & \text{---} \\ & W_1 \\ & \text{---} \\ & \end{matrix} \\ \begin{matrix} & \\ & 0 \\ & \\ & 0 \end{matrix} & \end{matrix}$$

$x = \text{cat image}$

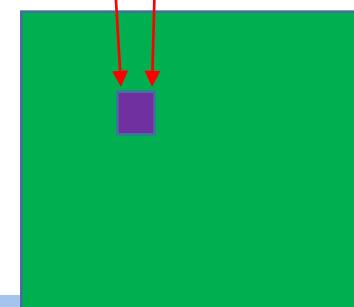
Image interpretation



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

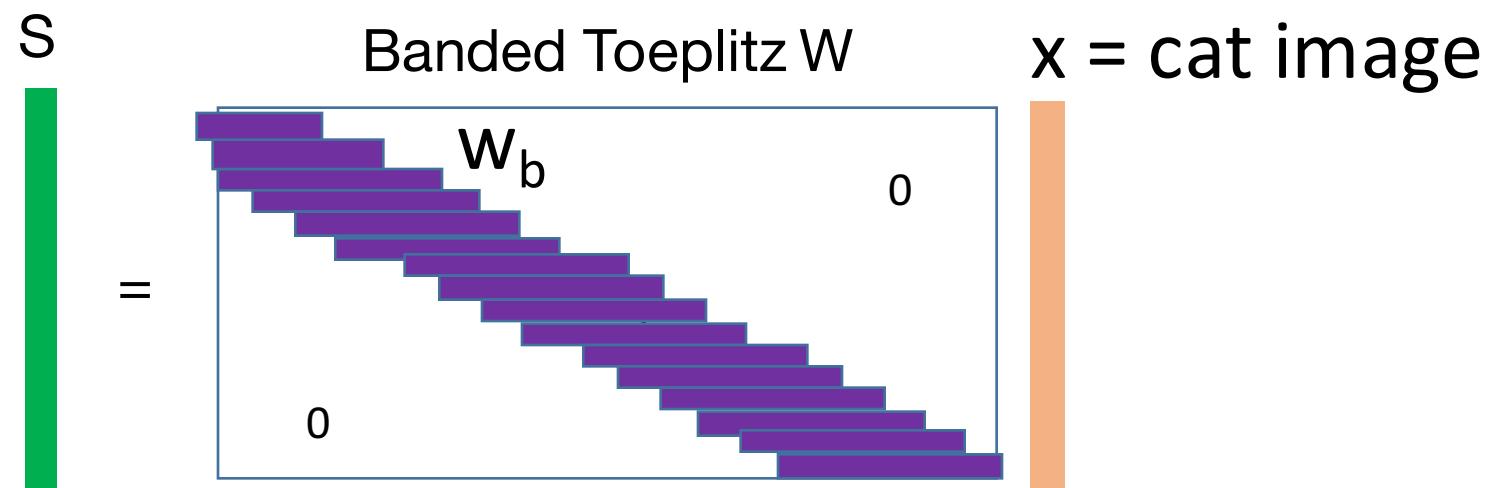
Full matrix: $O(n^2)$

Banded matrix: $k \cdot O(n)$



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*



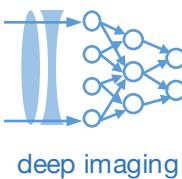
This type of matrix can dramatically reduce the number of weights that are used while still allowing *local* regions to mix:

Full matrix: $O(n^2)$

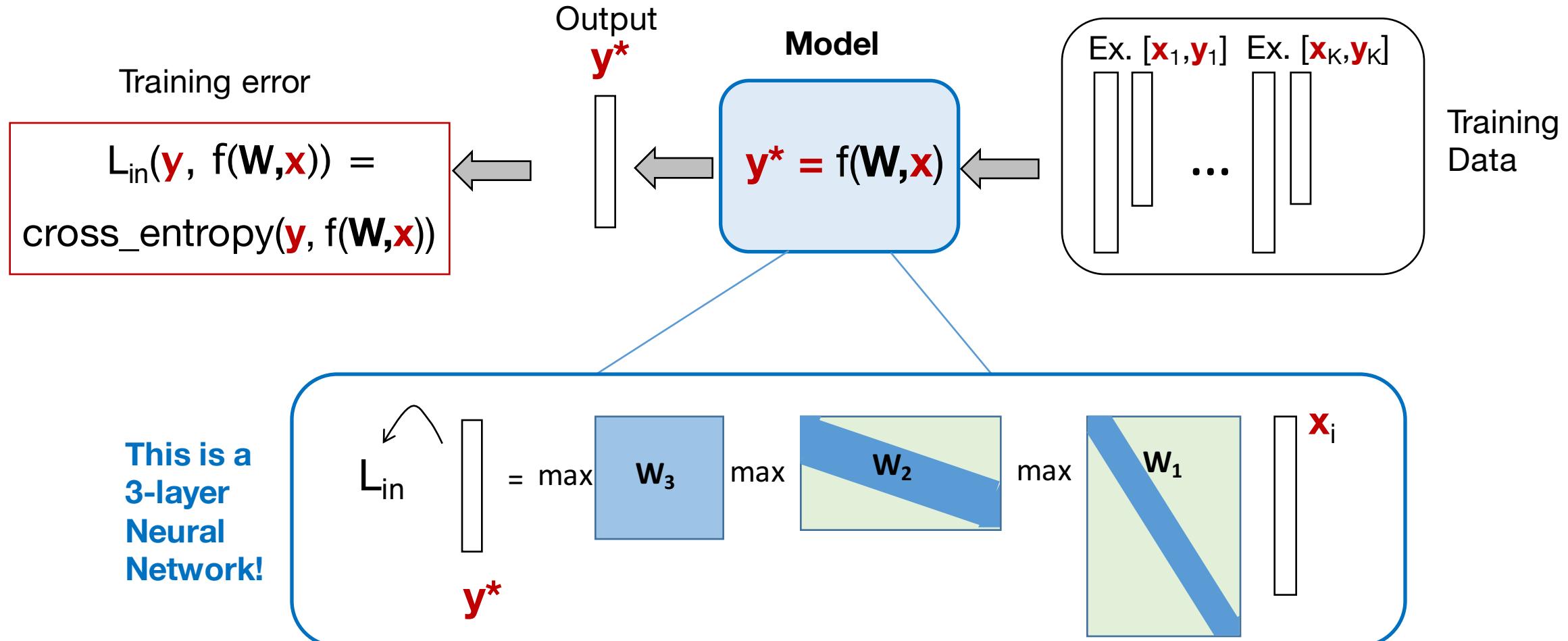
Banded matrix: $k \cdot O(n)$

Banded Toeplitz matrix: k

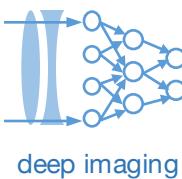
This is the definition of a convolution



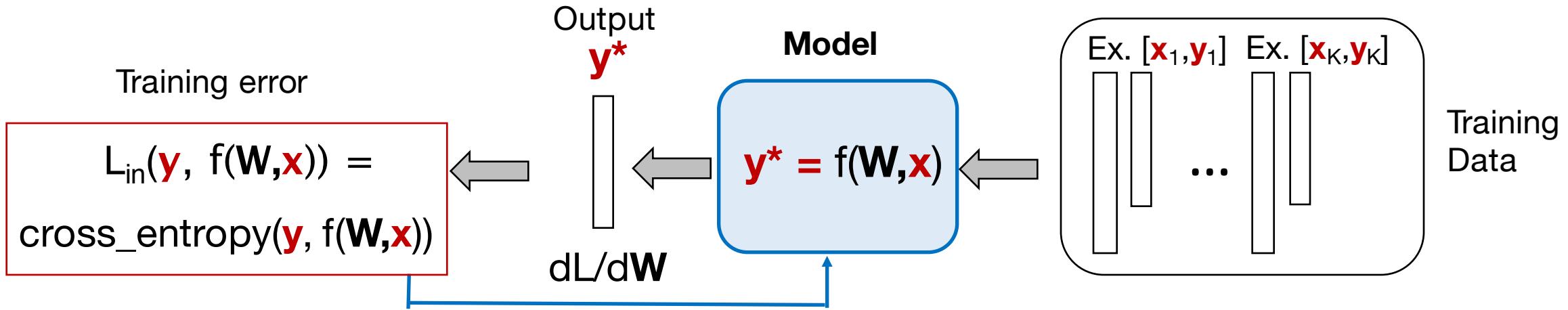
Our very basic convolutional neural network



Forward pass: from x_i and current W 's, find L_{in}



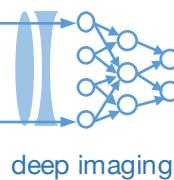
Our very basic convolutional neural network



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

\mathbf{W}_1 and \mathbf{W}_2 are banded Toeplitz matrices, \mathbf{W}_3 is a full matrix

3-layer network



Weights “savings” via convolution

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

- Having “fully connected” weight matrices can produce quite a lot of weights...

CIFAR10: 32x32 images = 1024 pixels

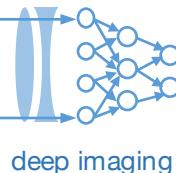
$W_1 = 1024 \times 512$

$W_2 = 512 \times 12$

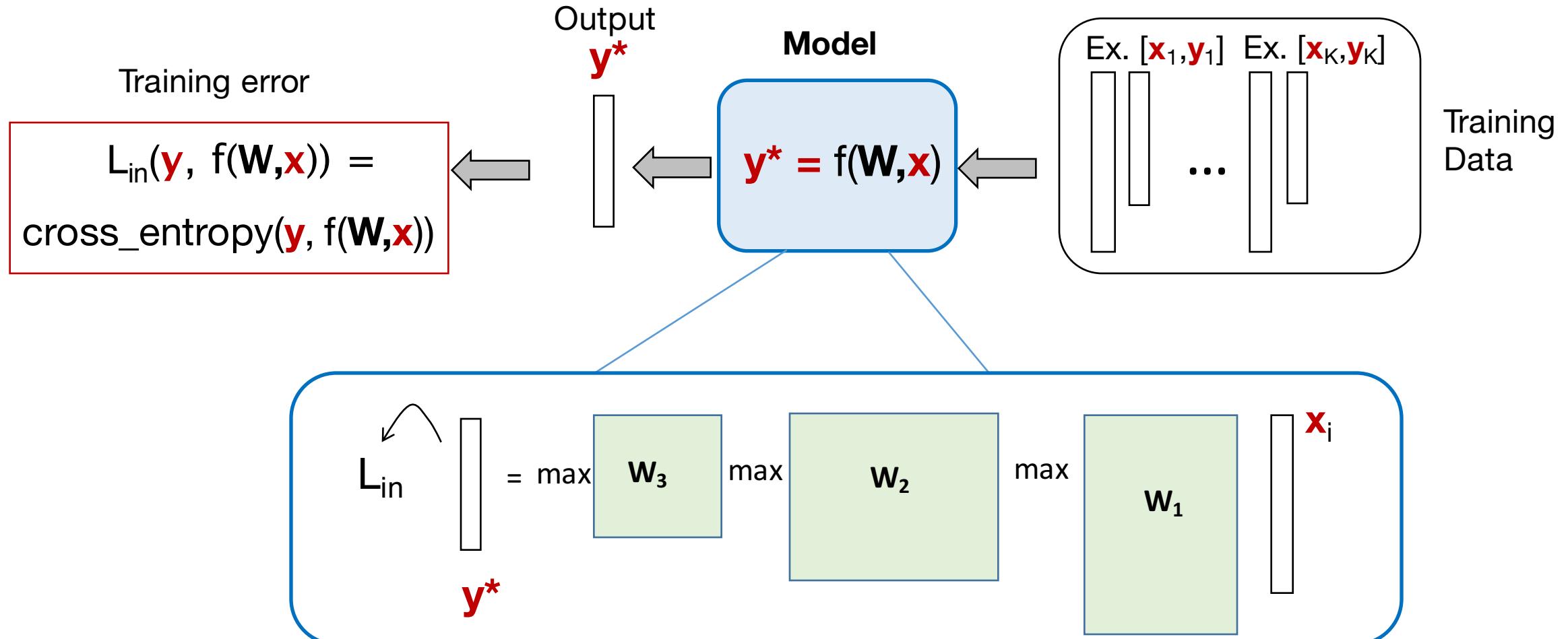
$W_3: 12 \times 12 = 144$

Total number of weights: 530,152

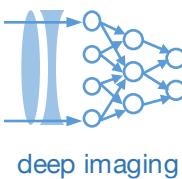
- Convolution (ballpark) = ?



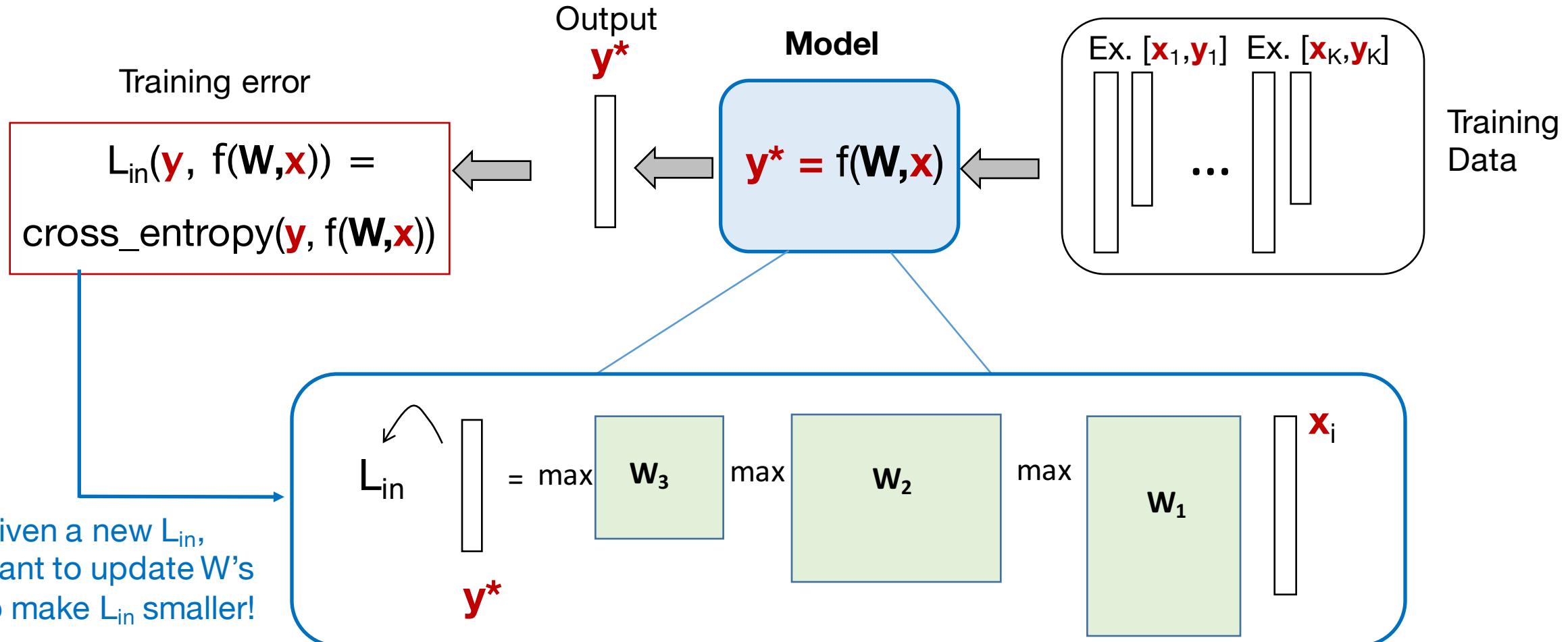
Our very basic convolutional neural network



Forward pass: from x_i and current W 's, find L_{in}



Our very basic convolutional neural network



Next class: Gradient descent via L_{in} to update many W 's



Break here to give brief introduction to CoLab

CO basic_tensorflow_eager_example.ipynb ☆

File Edit View Insert Runtime Tools Help

+ Code + Text

```
import numpy as np
import tensorflow as tf
tf.enable_eager_execution() # if we're using tf version 1.14, then we need to call this command; if using 2.0, then
```

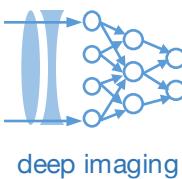
```
[ ] optimizer = tf.train.GradientDescentOptimizer(learning_rate=.2) # choose our optimizer and learning rate
x = tf.Variable(2.0) # define a variable to optimize, with an initial value of 2

for i in range(10): # iterative optimization loop
    with tf.GradientTape() as tape: # gradient tape keeps track of the gradients associated with all the operations
        # define our very simple minimization problem:
        loss = x ** 2 # we're going to minimize x^2, which occurs at x=0

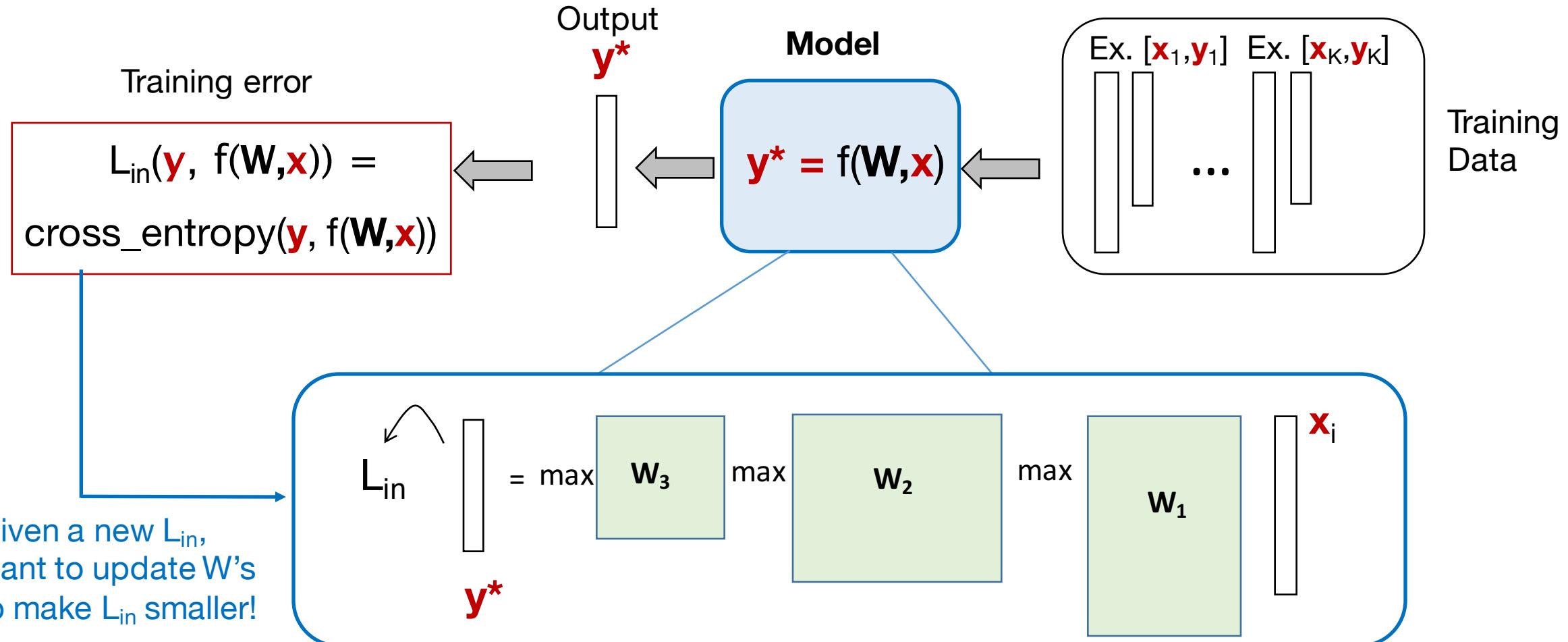
        # compute and apply gradients:
        gradient = tape.gradient(loss, x)
        optimizer.apply_gradients([(gradient, x)])

        # print out current iteration and loss value:
        print(i, 'loss = ' + str(loss.numpy()), 'x = ' + str(x.numpy()))
```

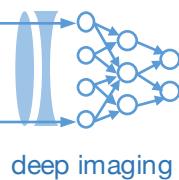
```
0 loss = 4.0 x = 1.2
1 loss = 1.44 x = 0.72
2 loss = 0.5184 x = 0.432
3 loss = 0.186624 x = 0.2592
4 loss = 0.06718464 x = 0.15552
5 loss = 0.024186473 x = 0.093312
6 loss = 0.008707129 x = 0.0559872
7 loss = 0.0031345668 x = 0.03359232
8 loss = 0.001128444 x = 0.020155393
9 loss = 0.00040623985 x = 0.012093236
```



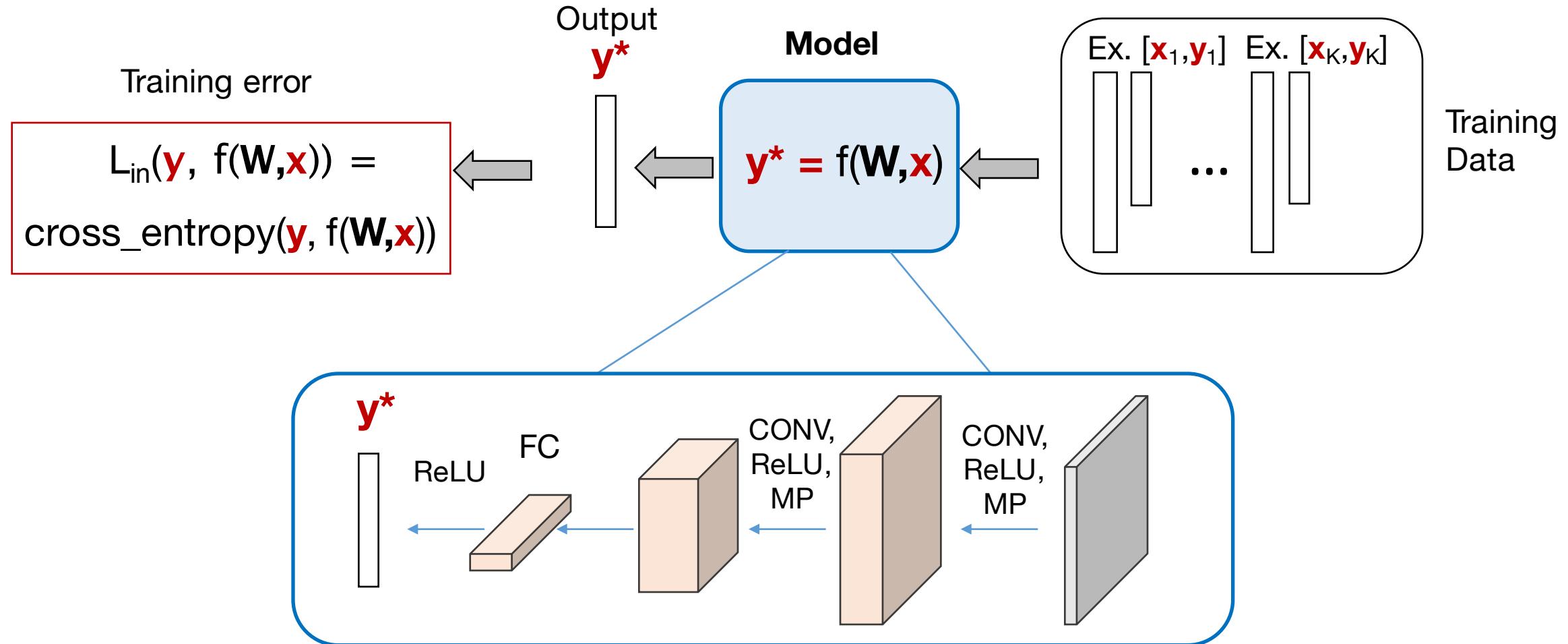
Our very basic convolutional neural network

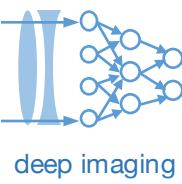


Next class: Gradient descent via L_{in} to update many W 's

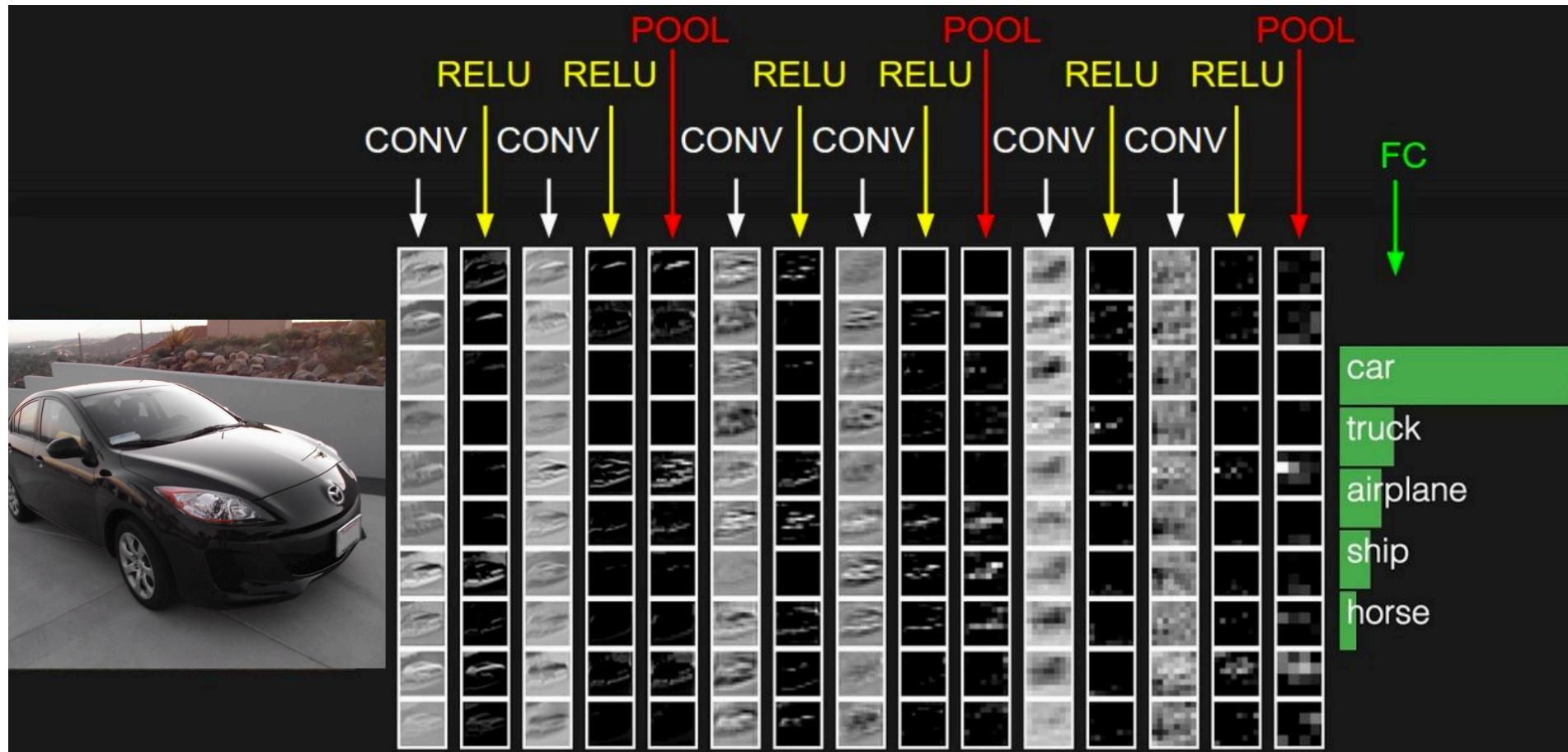


Our very basic convolutional neural network

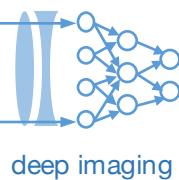




A standard CNN pipeline:

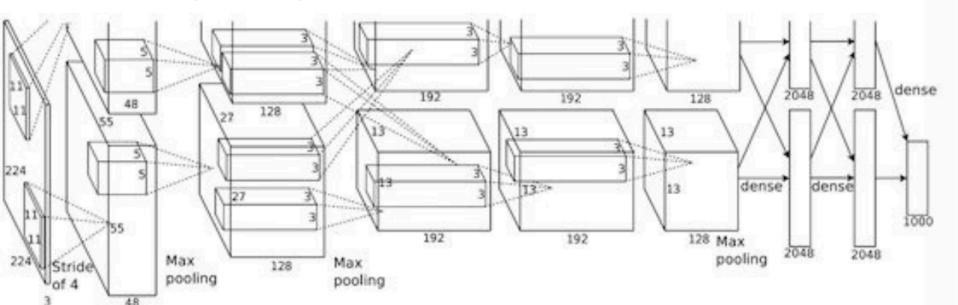


miniAlexNet, 2014



Complex networks are just an extension of this...

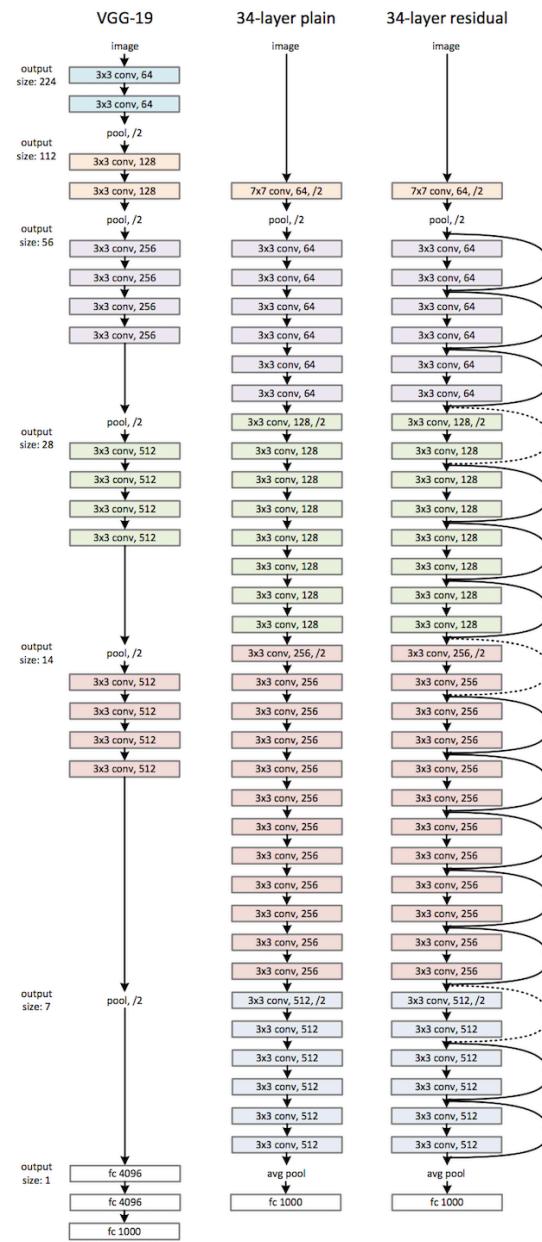
AlexNet (2012)

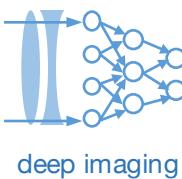


VGG (2014)

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
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	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	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	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

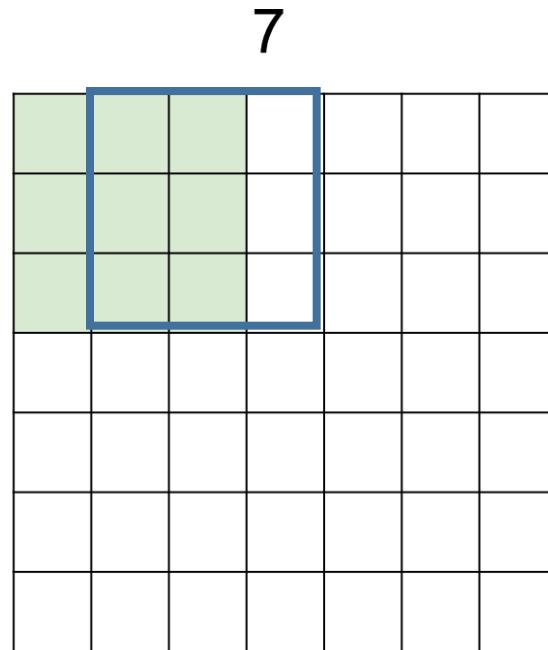
- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- Fully connected layers
- # of layers, dimensions per layer

Loss function & optimization

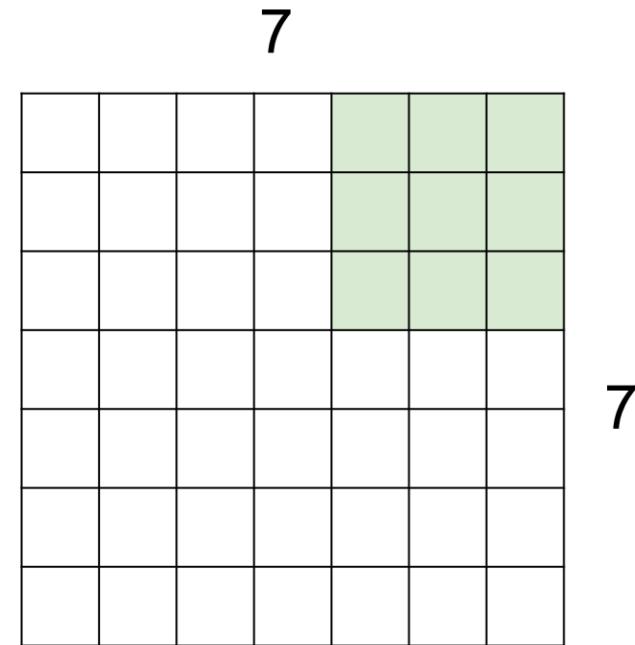
- Type of loss function
- Regularization
- Gradient descent method
- Gradient descent step size

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

Convolutions: size, stride and padding

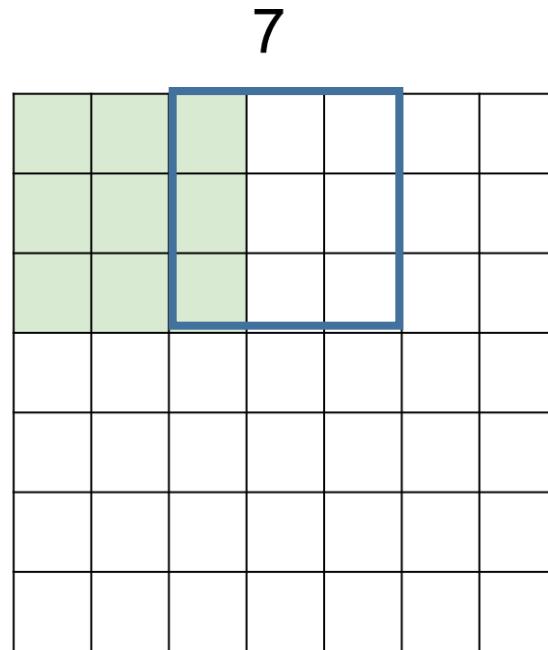


Slide one
pixel at a
time

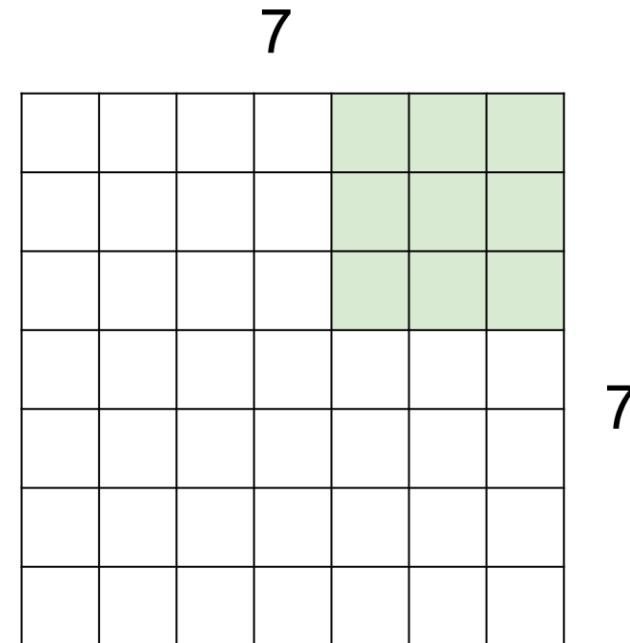


- 7x7 input image
 - 3x3 filter
- ↓
- 5x5 output

Convolutions: size, stride and padding



Slide two
pixels at a
time

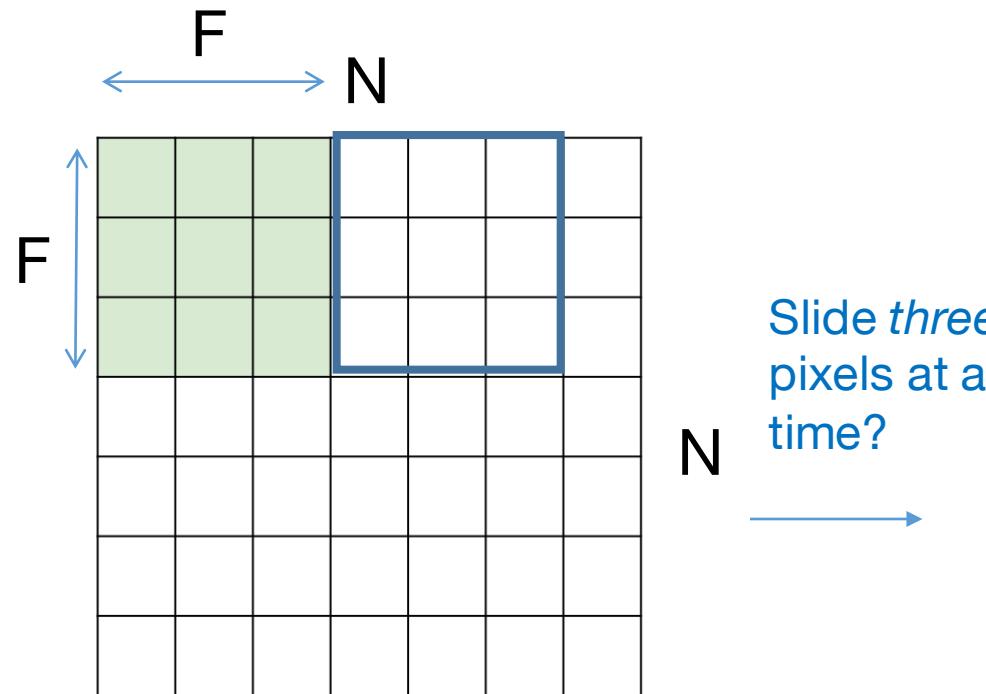


7

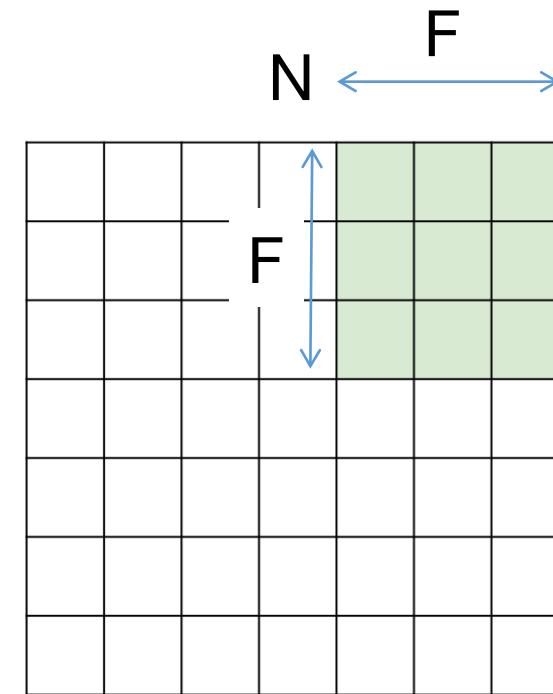
- 7x7 input image
 - 3x3 filter
- 3x3 output!

This is called a “stride 2” convolution

Convolutions: size, stride and padding



This is called a “stride 3” convolution



Output matrix width W :

$$W = (N-F)/\text{stride} + 1$$

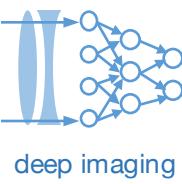
7 Example right: $N=7$, $F=3$

When stride = 1: $W = 5$

When stride = 2: $W = 3$

When stride = 3: $W = 2.33???$

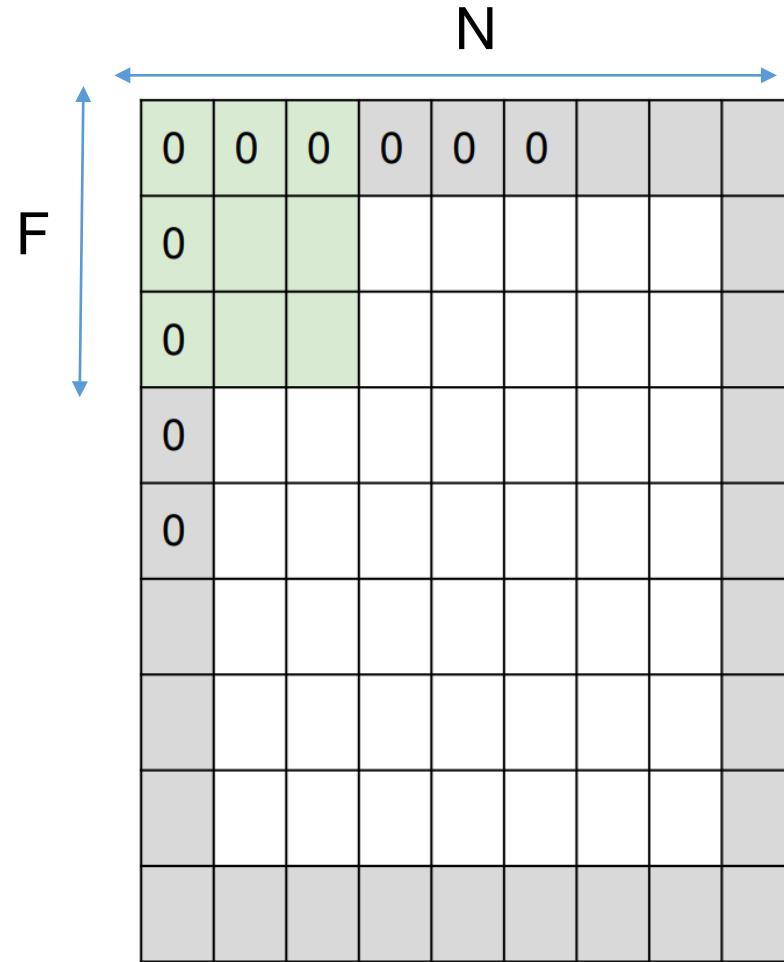
*Need to ensure integers work out!



Convolutions: size, stride and padding

Q: What if you really, really want to use a stride = 3 with N = 7 and F=3?

Convolutions: size, stride and padding



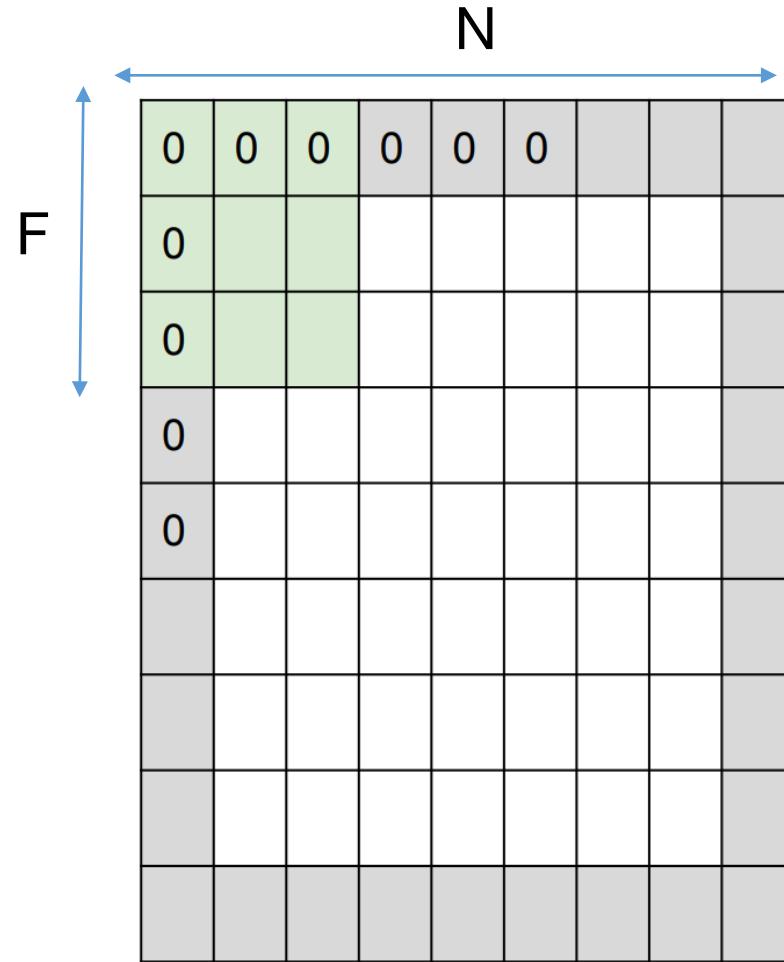
Q: What if you really, really want to use a stride = 3 with $N = 7$ and $F=3$?

A: Use *padding*

E.g., padding with 1 pixel around border makes $N=9$

Padding: add zeros around edge of image

Convolutions: size, stride and padding



Q: What if you really, really want to use a stride = 3 with $N = 7$ and $F=3$?

A: Use *padding*

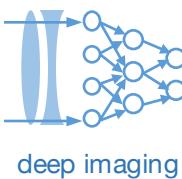
E.g., padding with 1 pixel around boarder makes $N=9$

$$W = (N-F)/\text{stride} + 1$$

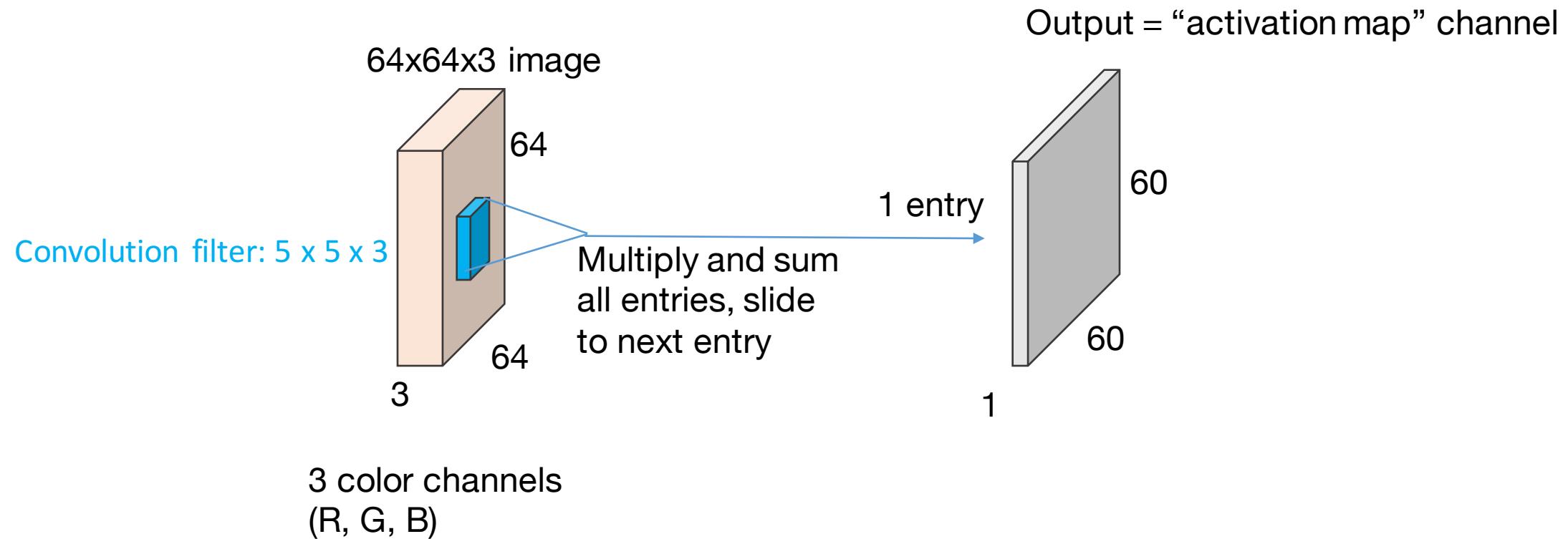
$$W = (9-3)/3 + 1 = 4$$

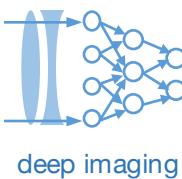
*Padding enables integer output!

Padding: add zeros around edge of image

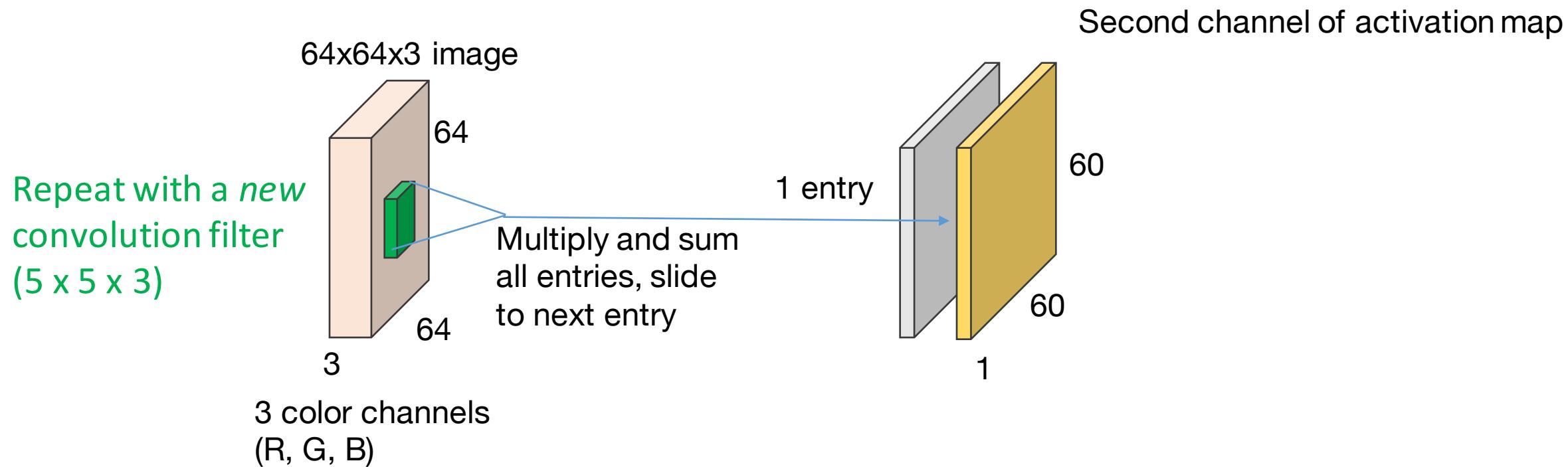


Convolution layer: learn multiple filters

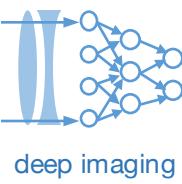




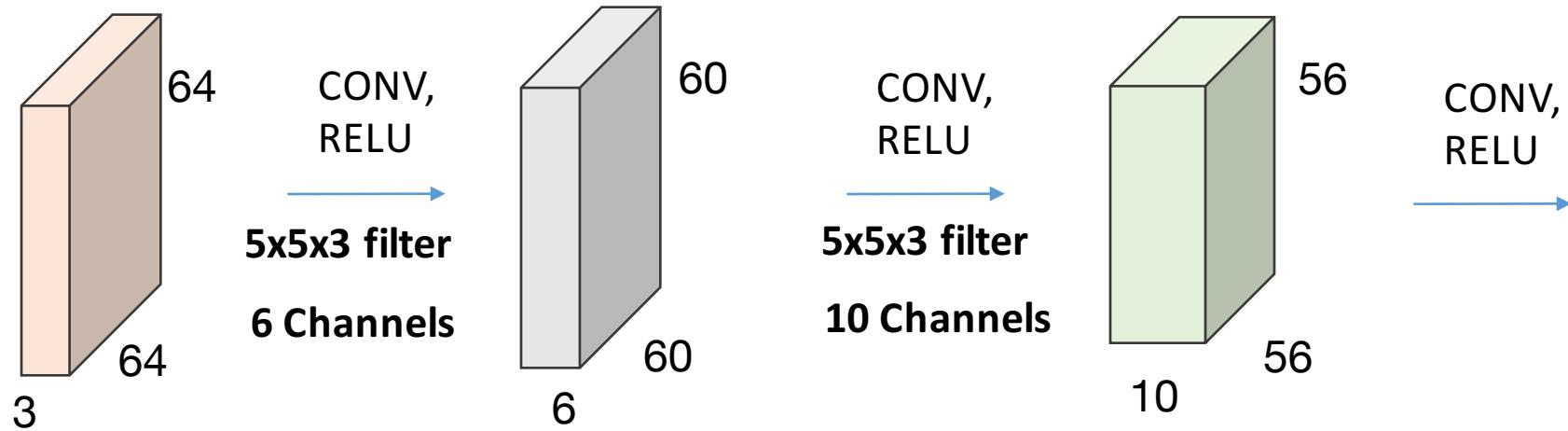
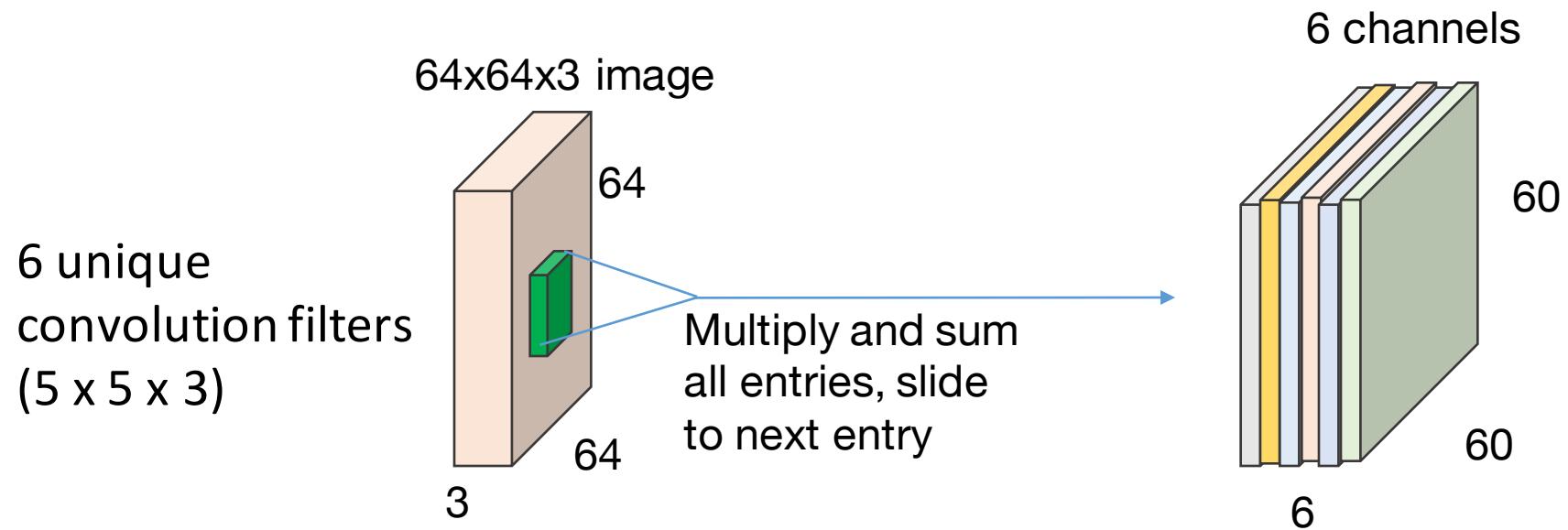
Convolution layer: learn multiple filters

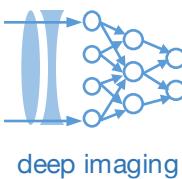


- Using more than one convolutional filter, with unknown weights that we will optimize for, creates more than one *channel*



Convolution layer: learn multiple filters



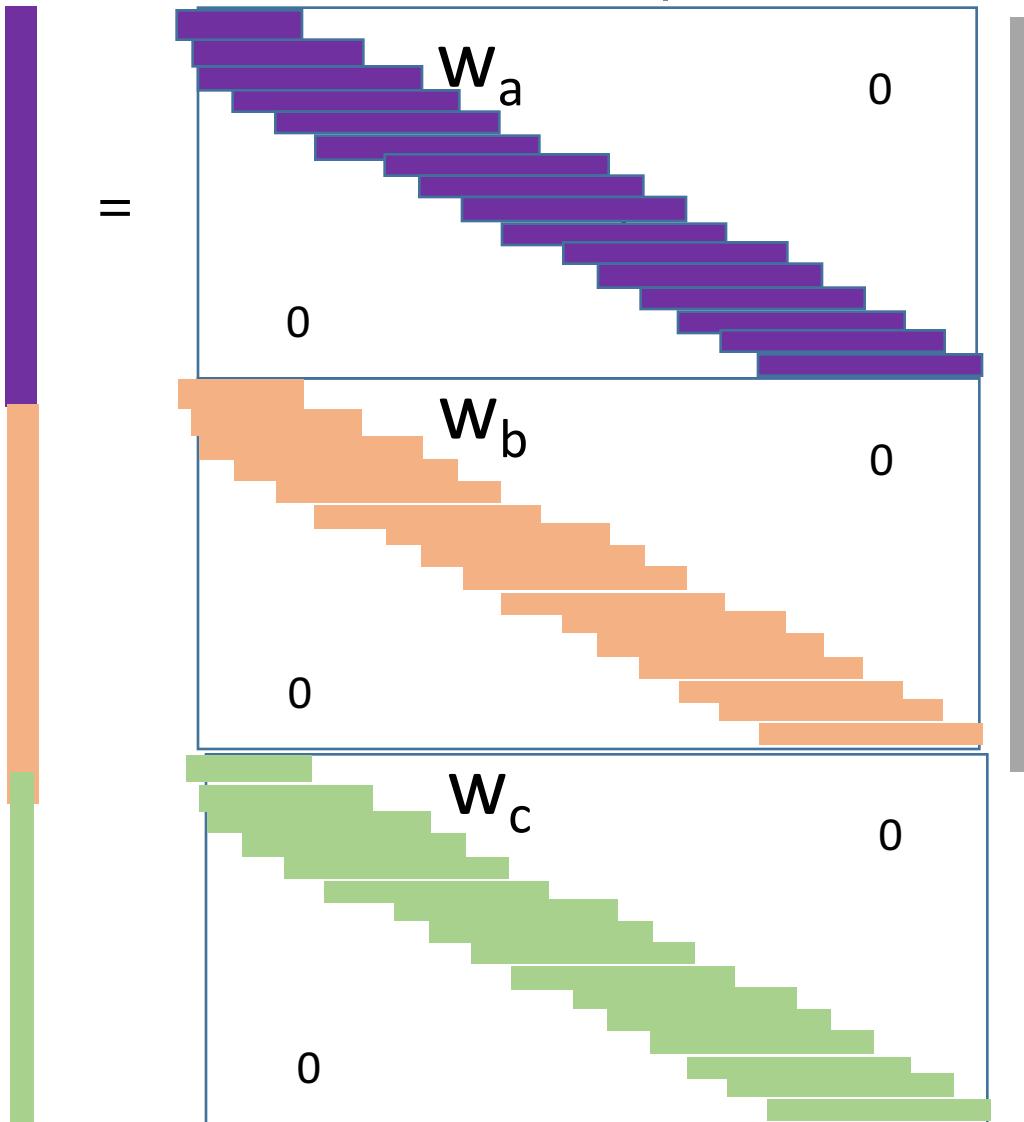


Summarize multiple filters with stacked matrices

x_o = output image

Banded Toeplitz W

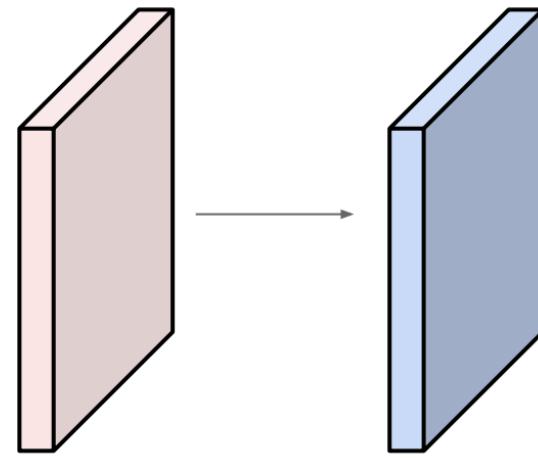
x_i = input image



Convolution layer example mapping

Examples time:

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

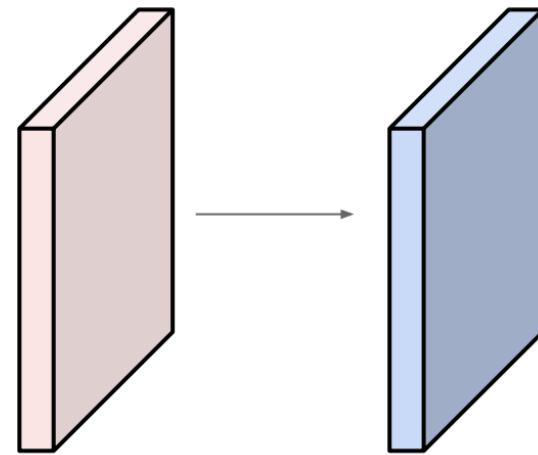


Output volume size: ?

Convolution layer example mapping

Examples time:

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



Output volume size: ?

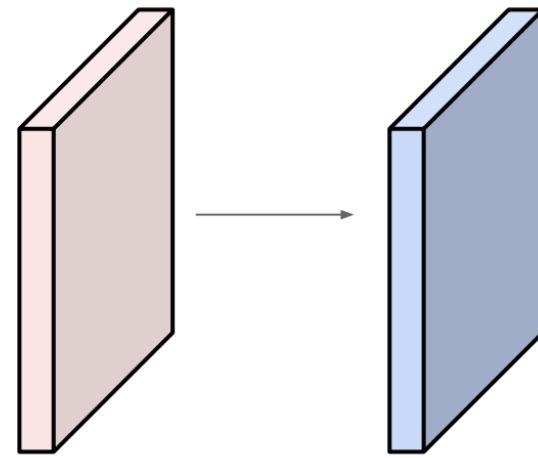
$$A: (N-F)/\text{stride} + 1 = (32+4-5)/1 + 1 = 32 \times 32 \text{ spatial extent}$$

So, output is **32x32x10**

Convolution layer example mapping

Examples time:

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

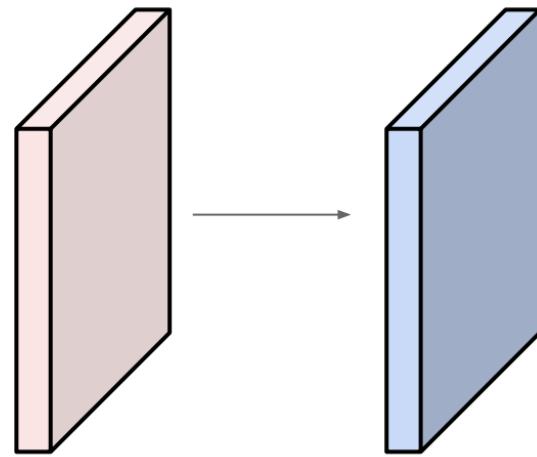


How many weights make up this transformation?

Convolution layer example mapping

Examples time:

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



How many weights make up this transformation?

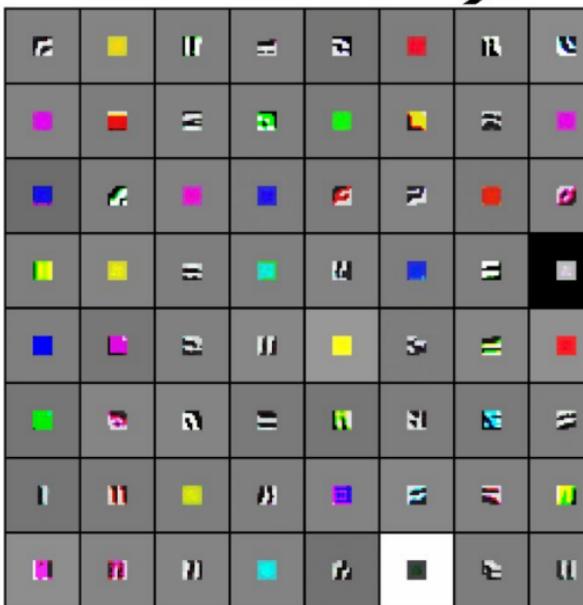
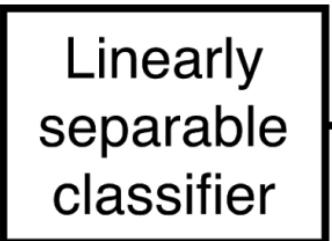
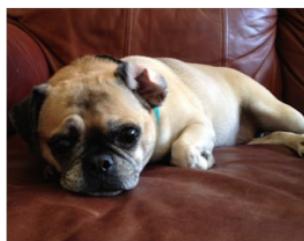
- A: Each convolution filter: $5 \times 5 \times 3$
 1 offset parameter **b** per filter (**untied** biases)
 Mapping to 10 output layers = 10 filters
 Total: $(5 \times 5 \times 3 + 1) \times 10 = 760$

What do these convolution filters look like after training?

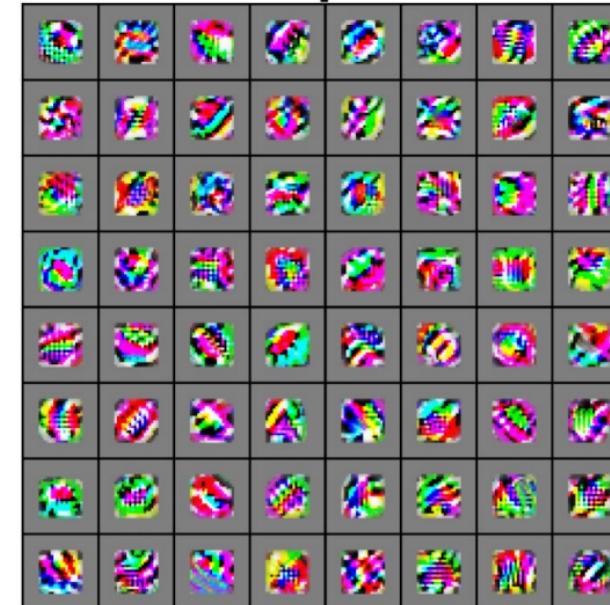
Preview

[Zeiler and Fergus 2013]

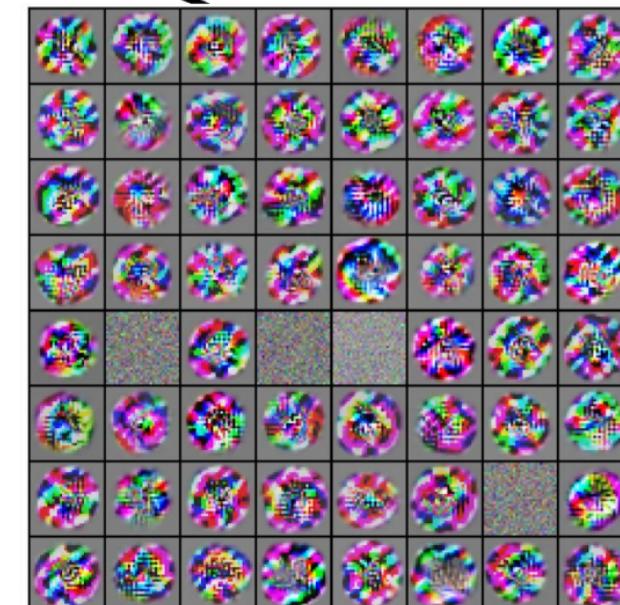
Visualization of VGG-16 by Lane McIntosh. VGG-16 architecture from [Simonyan and Zisserman 2014].



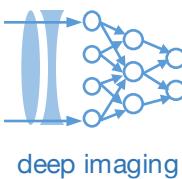
VGG-16 Conv1_1



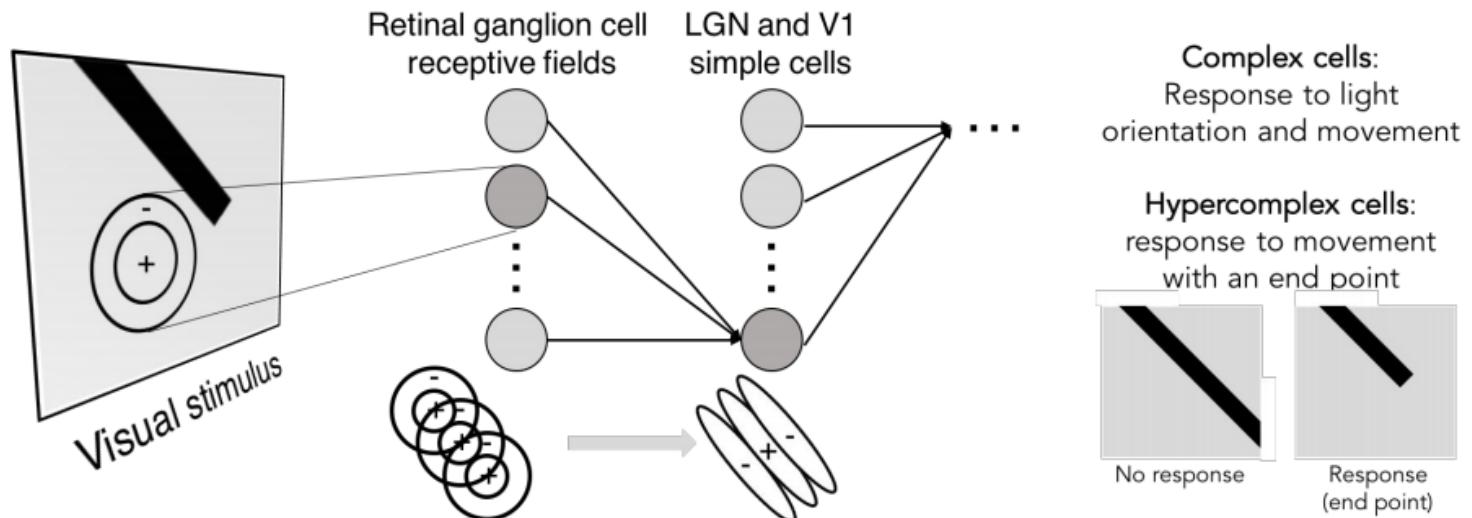
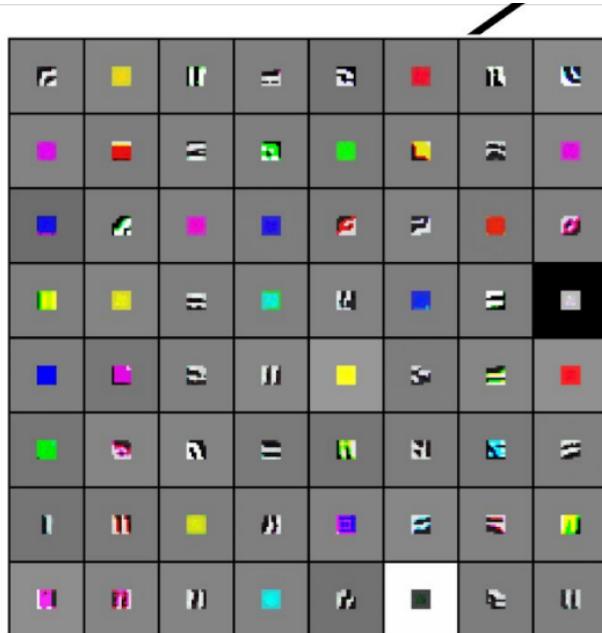
VGG-16 Conv3_2



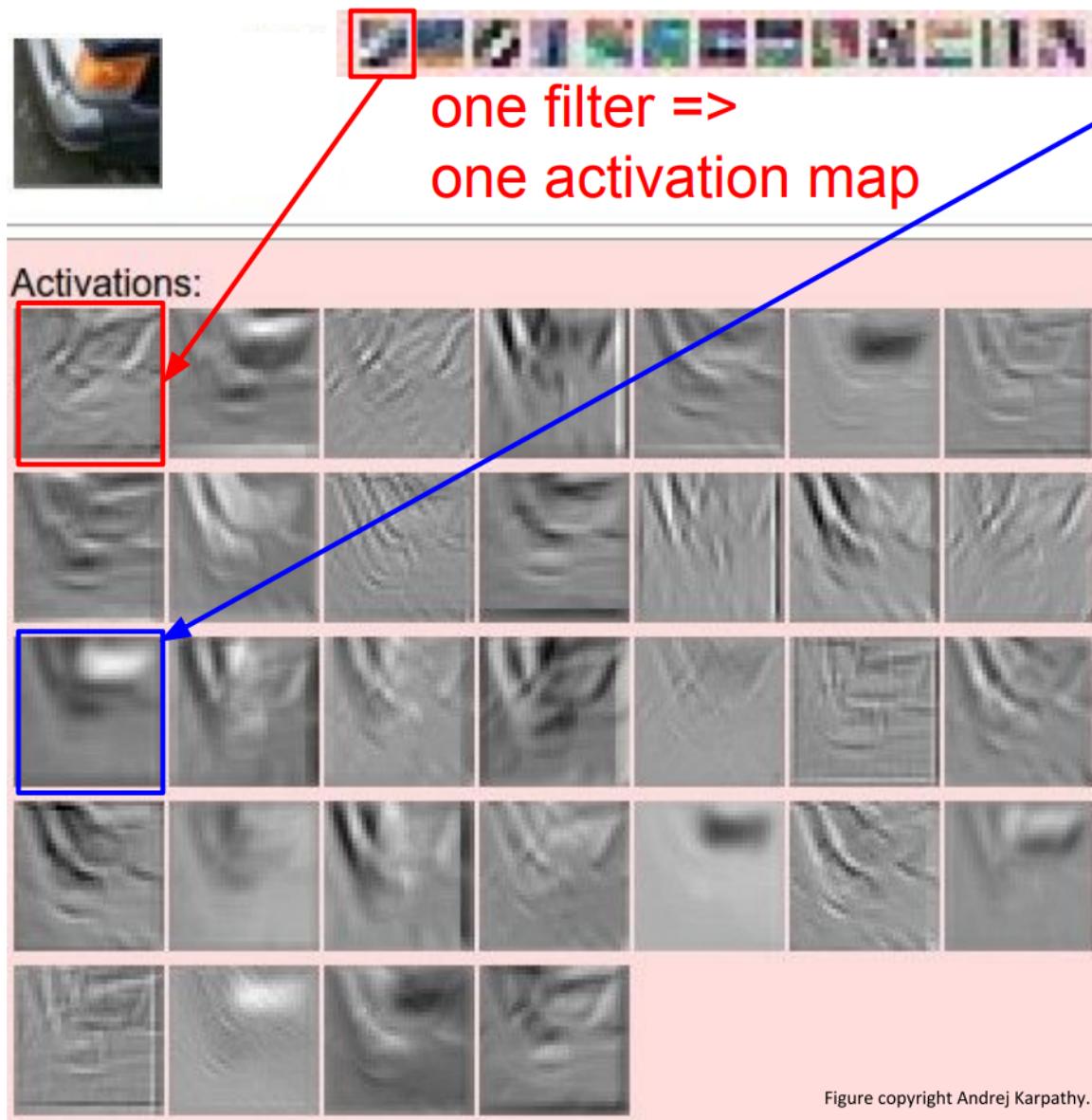
VGG-16 Conv5_3



What do these convolution filters look like after training?

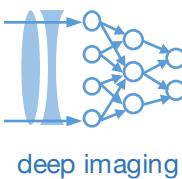


- "Wavey" or wavelet like features are common in first layer
- Match how neurons within our eye map image data to our brain in an effective manner



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

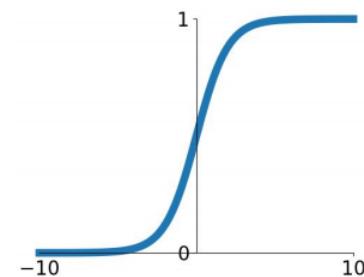
- Type of loss function
- Regularization
- Gradient descent method
- Gradient descent step size

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

Non-linear “activation” functions

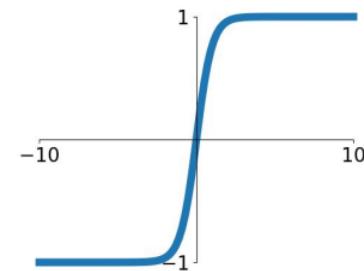
Sigmoid

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



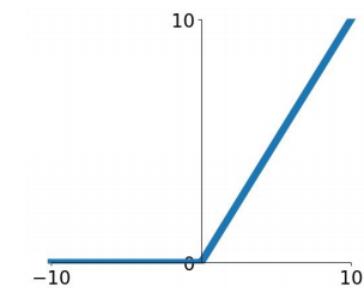
tanh

$$\tanh(x)$$



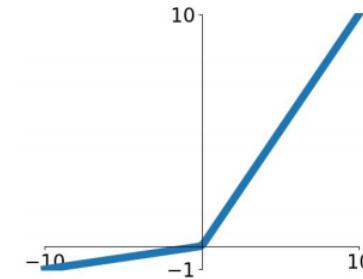
ReLU

$$\max(0, x)$$



Leaky ReLU

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

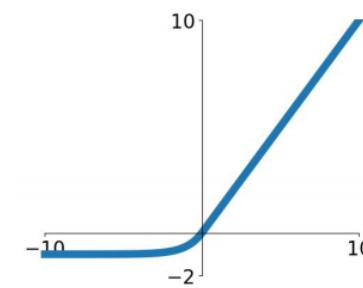


Maxout

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

ELU

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

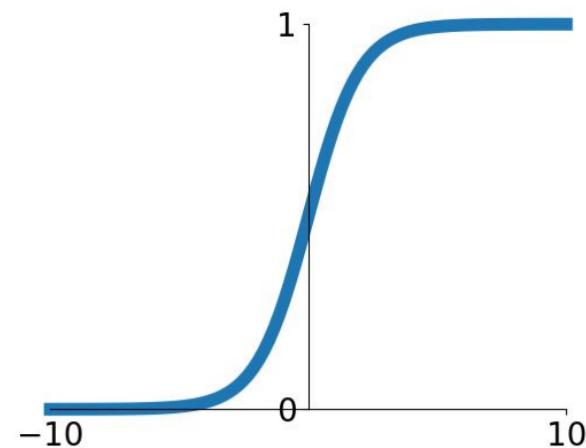


From Stanford CS231

Non-linear “activation” functions

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

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



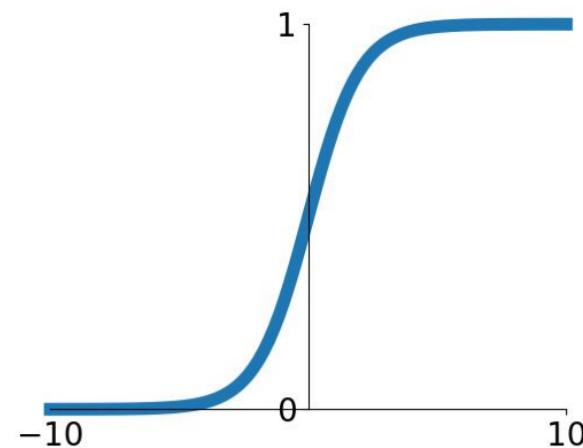
Sigmoid

From Stanford CS231

Non-linear “activation” functions

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

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



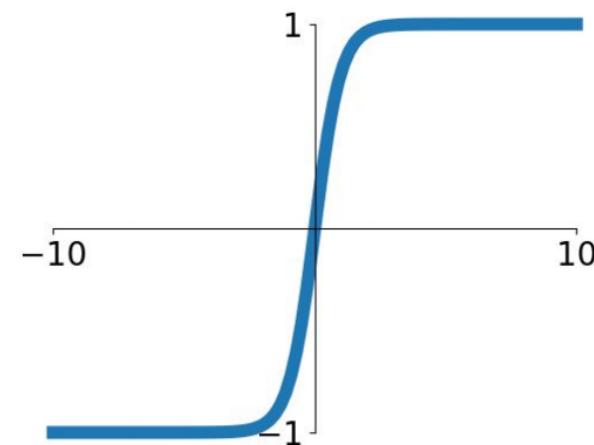
Sigmoid

3 problems:

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

From Stanford CS231

Non-linear “activation” functions



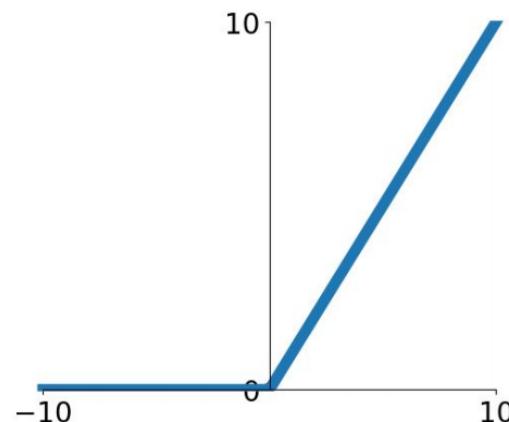
$\tanh(x)$

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

From Stanford CS231

Non-linear “activation” functions

Computes $f(x) = \max(0, x)$



ReLU
(Rectified Linear Unit)

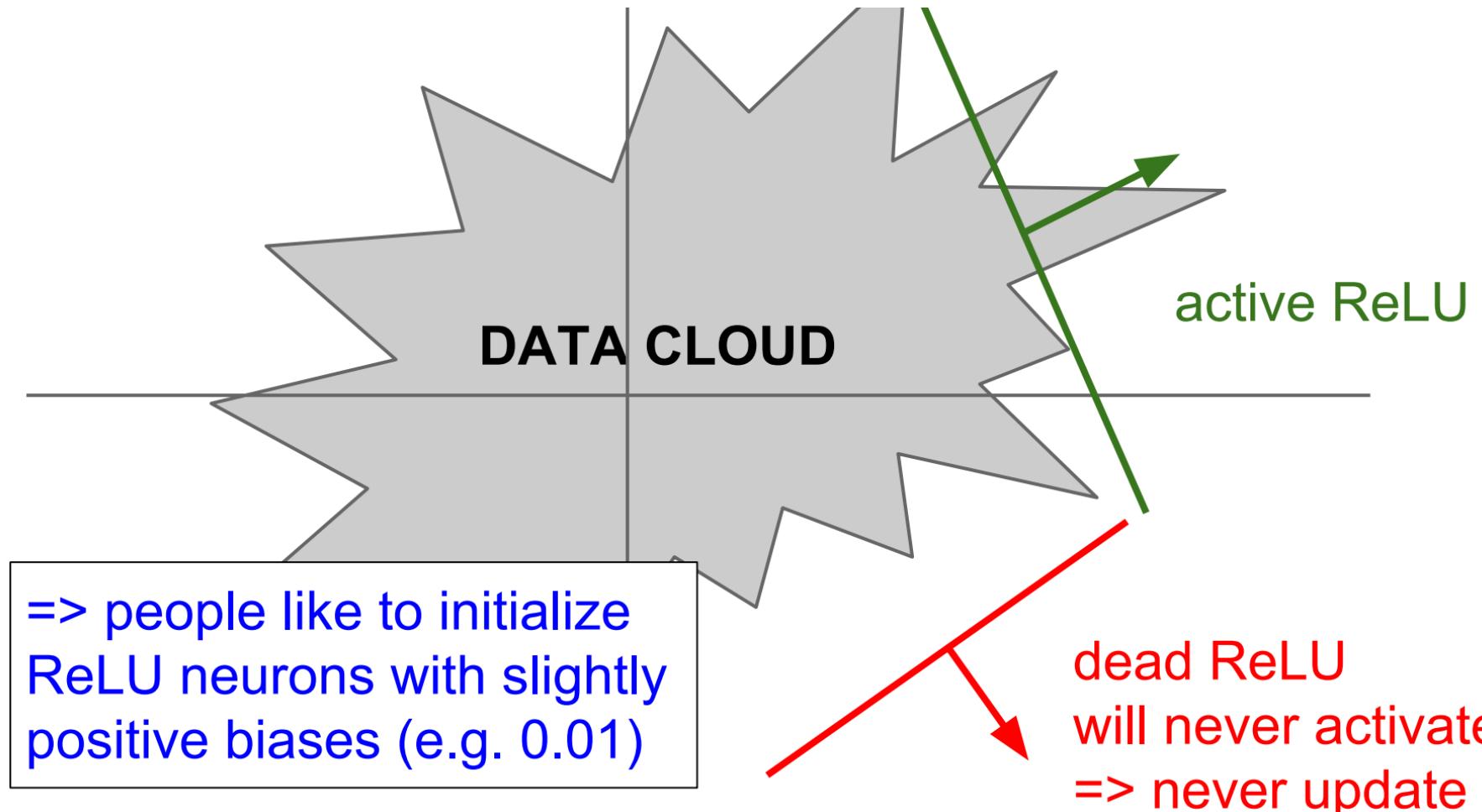
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

- Not zero-centered output
- An annoyance:

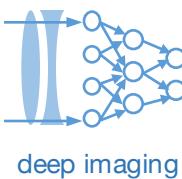
hint: what is the gradient when $x < 0$?

From Stanford CS231

Non-linear “activation” functions



From Stanford CS231



Important components of a CNN

CNN Architecture

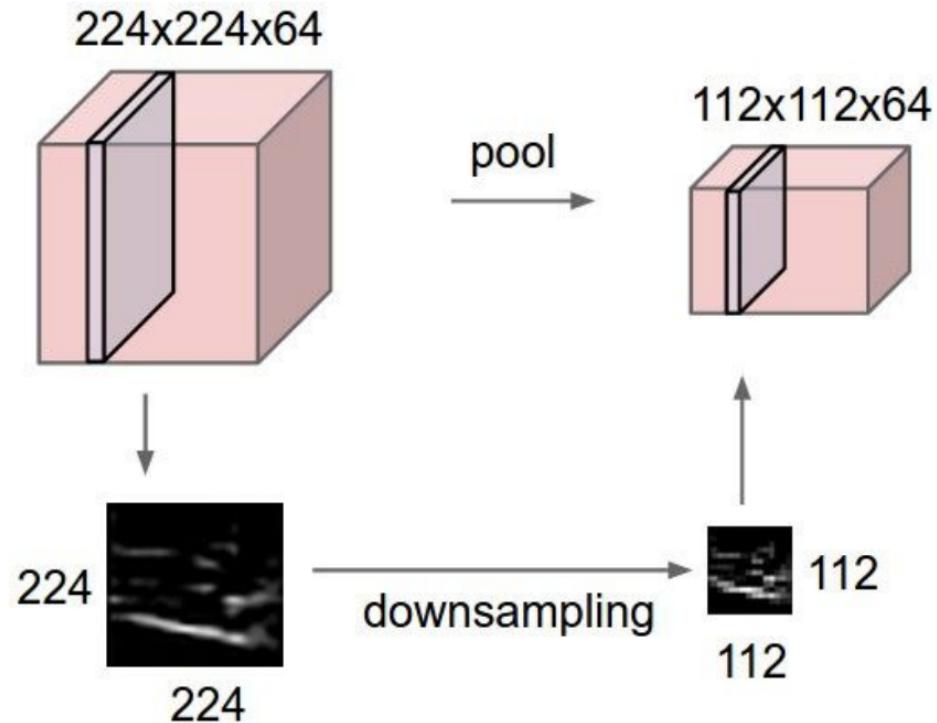
- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

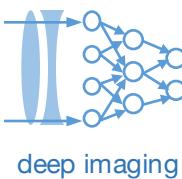
Loss function & optimization

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

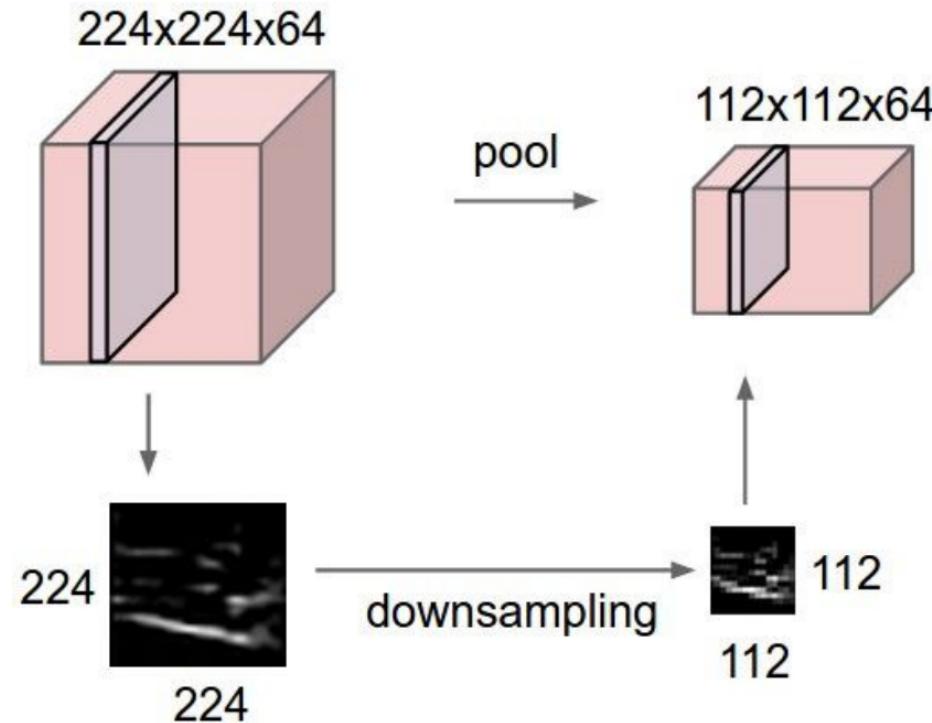
Other specifics: Pre-processing, initialization, dropout, batch normalization, augmentation

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

Single depth slice

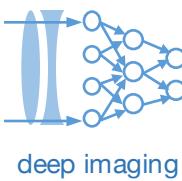
1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

x
y

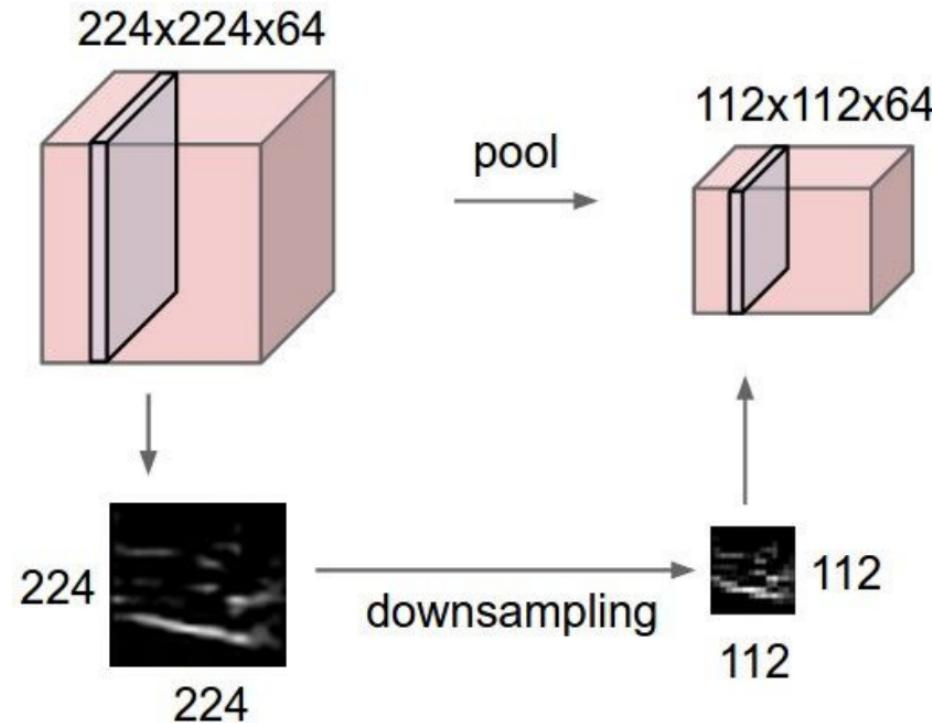
max pool with 2x2 filters
and stride 2

6	8
3	4

Related options: Sum pooling, mean pooling

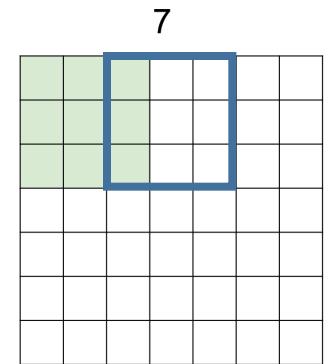


Pooling operation – reduce the size of data cubes along space



Common option #2: just use bigger strides

STRIDE = 2



7x7 input \rightarrow 3x3 output

$$\begin{array}{c|ccccc|cc} c_1 & & & & 0 & & c_1 \\ c_2 & c_1 & & & & & (skip) \\ & c_2 & c_1 & & & & c_1 \\ & & c_2 & c_1 & c_1 & & (skip) \\ 0 & & & c_2 & c_1 & c_2 & c_1 \\ & & & & c_2 & & c_2 \end{array}$$

Important components of a CNN

Let's
view
some
code!

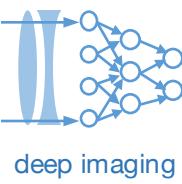
CNN Architecture

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

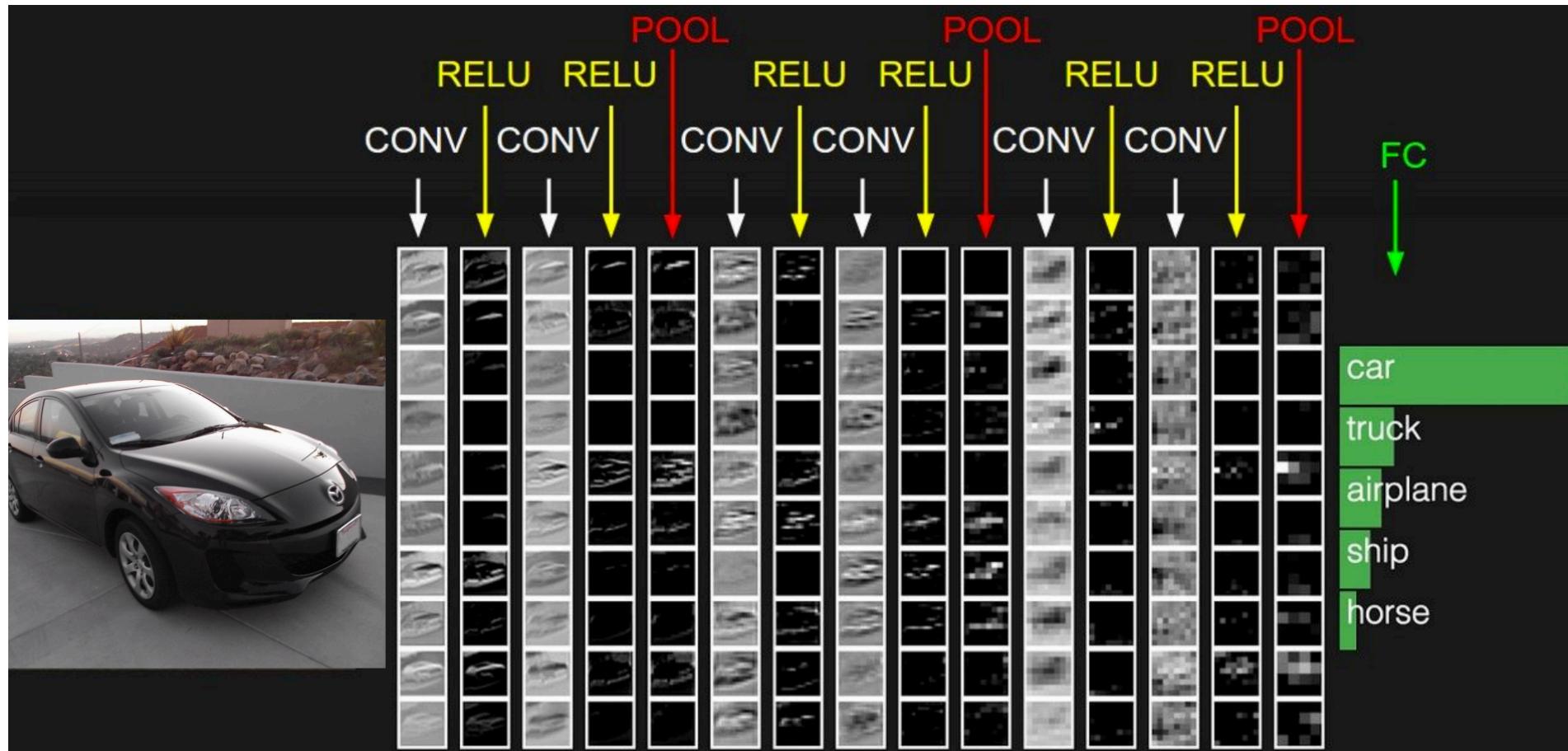
Loss function & optimization

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

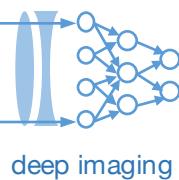
Other specifics: Pre-processing, initialization, dropout, batch normalization, augmentation



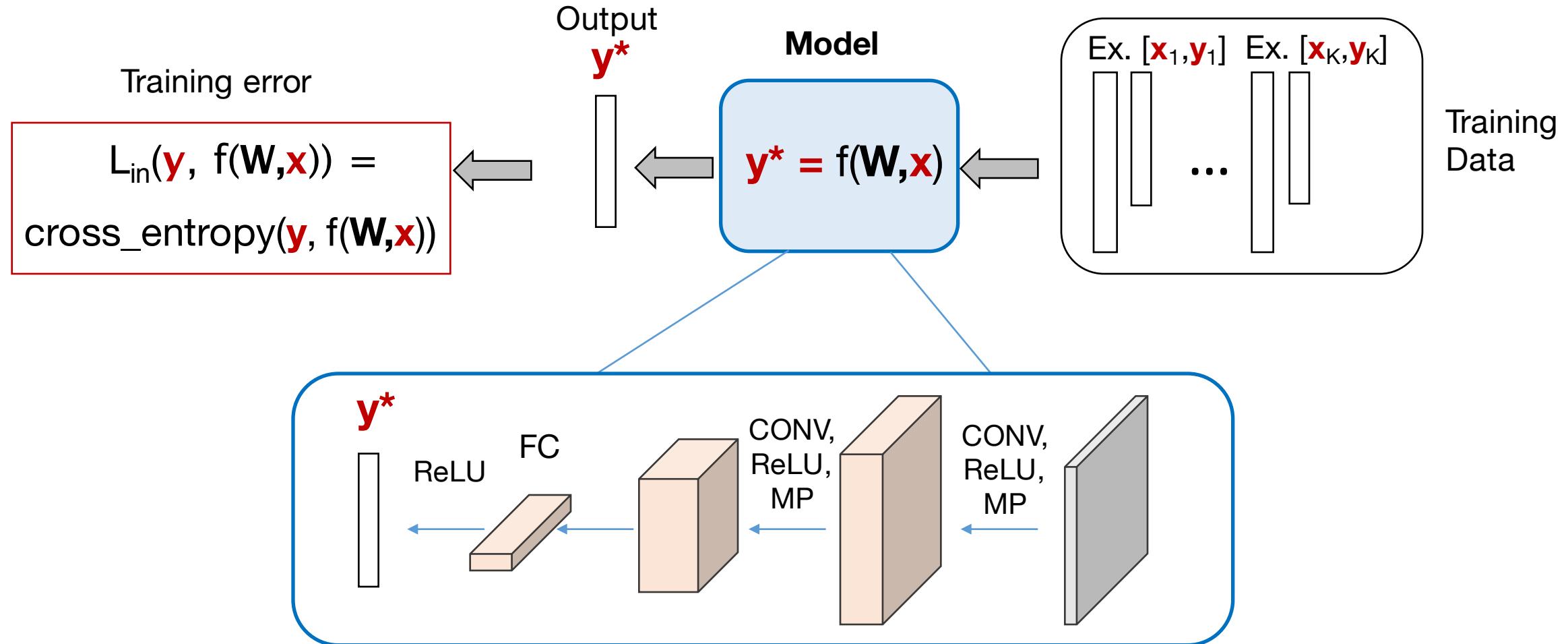
A standard CNN model: do we understand the architecture?

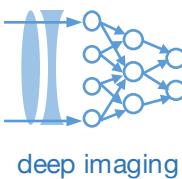


miniAlexNet, 2014



Our very basic convolutional neural network





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

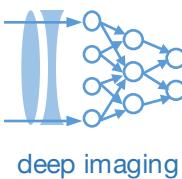
Loss function & optimization

- 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

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss: Model predictions should match training data}} + \lambda R(W)$$

λ = regularization strength (hyperparameter)

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

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

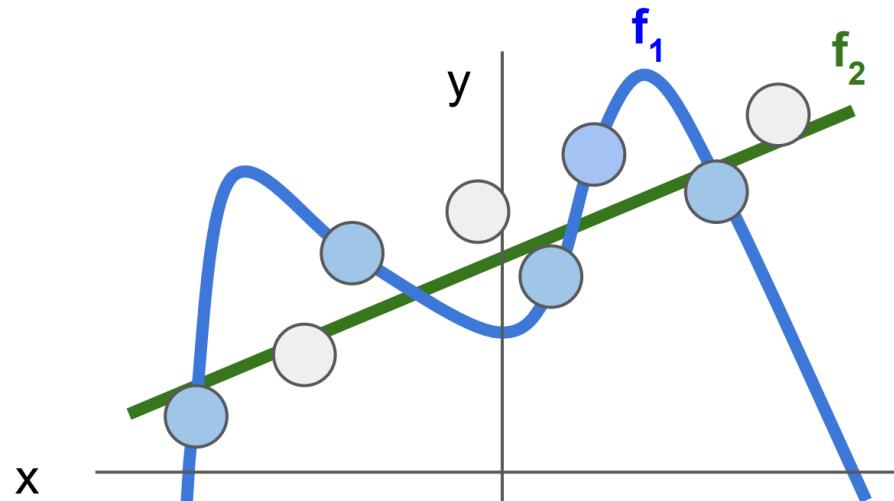
$$w_1 = [1, 0, 0, 0]$$

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

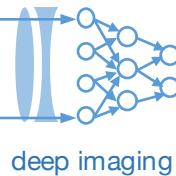
$$w_1^T x = w_2^T x = 1$$

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



A two-layer neural network with regularization:

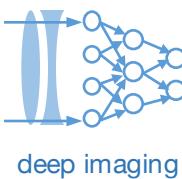
$$L(w) = \frac{1}{N} \sum_{i=1}^N \ln(1 + e^{-y_i W_2 \max(W_1 x_i, 0)}) + \lambda(||W_1||_2 + ||W_2||_2)$$

Q: How do we determine the best weights W_1 and W_2 to use from this model?

A: Gradient descent!

Q: How do we figure out the gradients for dL/dW_1 and dL/dW_2 ?

A: Chain rule! (next lectures)



Important components of a CNN

CNN Architecture

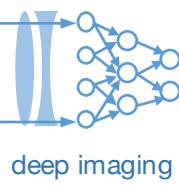
- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

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!

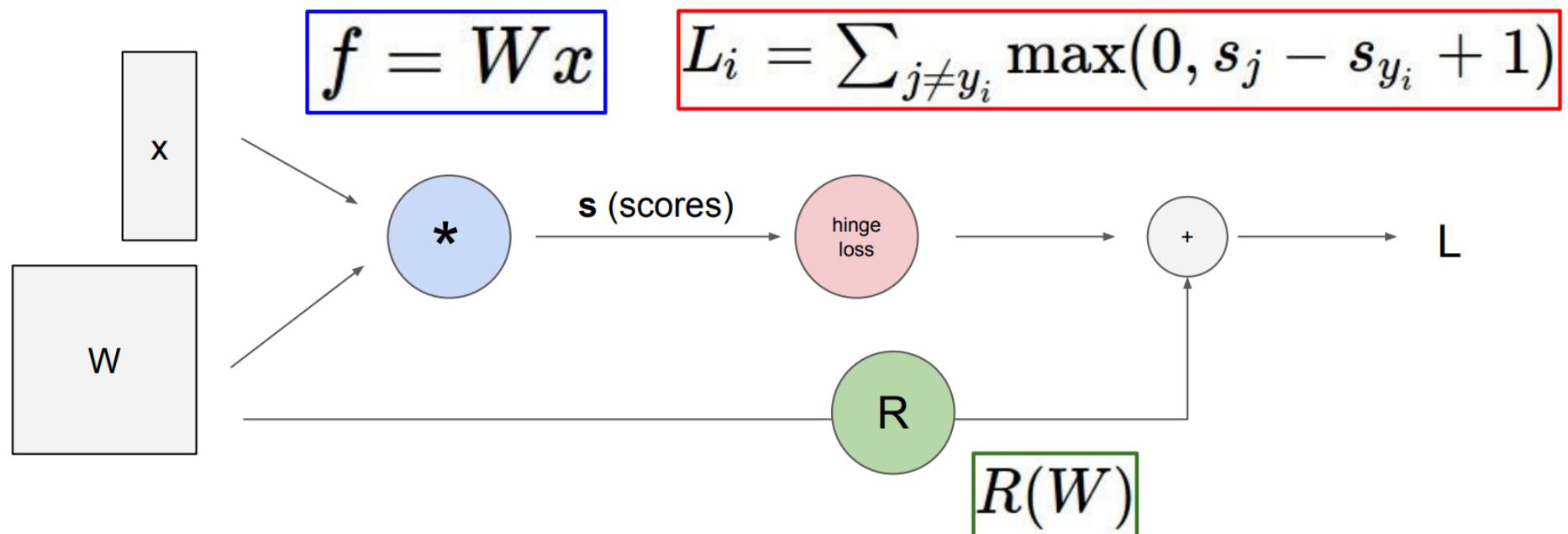
Approximate sum
using a **minibatch** of
examples
32 / 64 / 128 common

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)

Next lecture: how Tensorflow solves gradient descent for you

Computational Graphs and the Chain Rule!



$$R(W)$$