

CS163: Deep Learning for Computer Vision

Course Summary

Deep Learning for Computer Vision

Building artificial systems
that process, perceive, and
reason about **visual data**

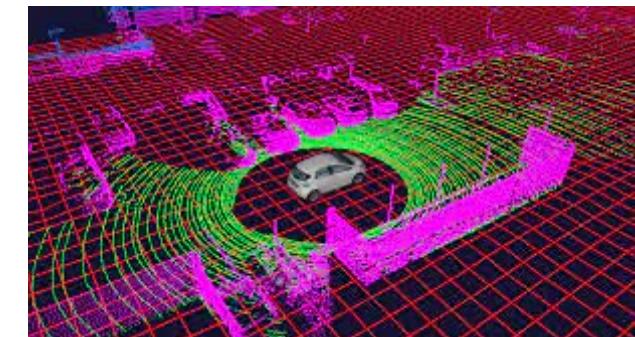
image



video



3D point clouds



Deep Learning for Computer Vision

Hierarchical learning algorithms
with many “layers”, (very) loosely
inspired by the brain

Artificial Intelligence

Computer
Vision

Machine Learning

Deep
Learning

This class

Lecture 1: a bit history on deep learning

IMAGENET Large Scale Visual Recognition Challenge

The Image Classification Challenge:
1,000 object classes
1,431,167 images



Output:
Scale
T-shirt
Steel drum
Drumstick
Mud turtle

Deng et al, 2009
Russakovsky et al. IJCV 2015

1959
Hubel & Wiesel

1963
Roberts

1970s
David Marr

1979
Gen. Cylinders

1986
Canny

1997
Norm. Cuts

1999
SIFT

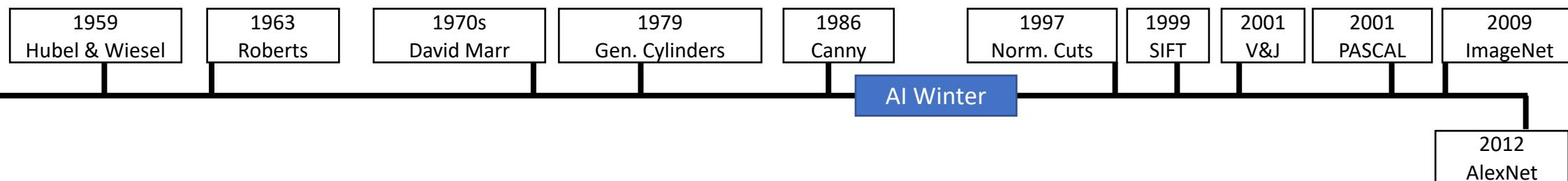
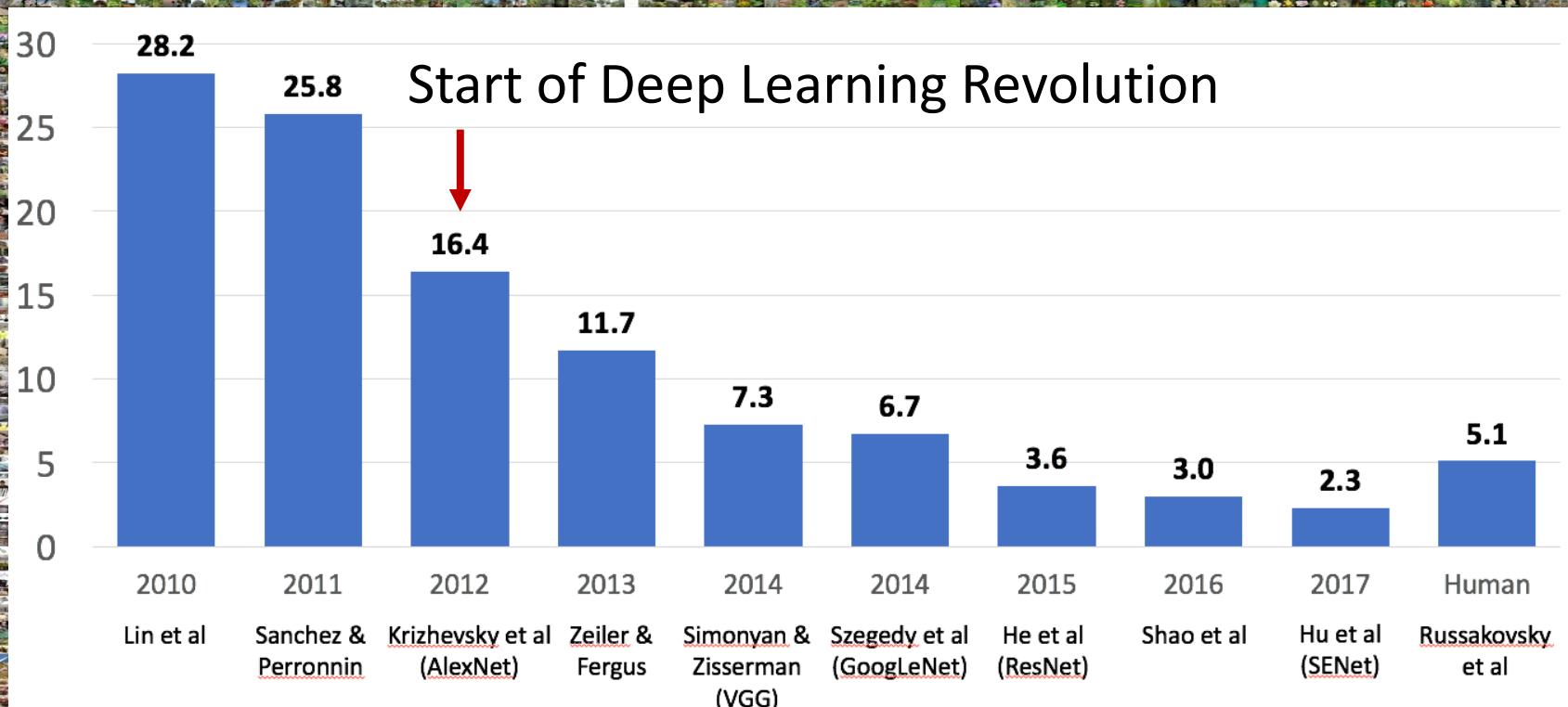
2001
V&J

2001
PASCAL

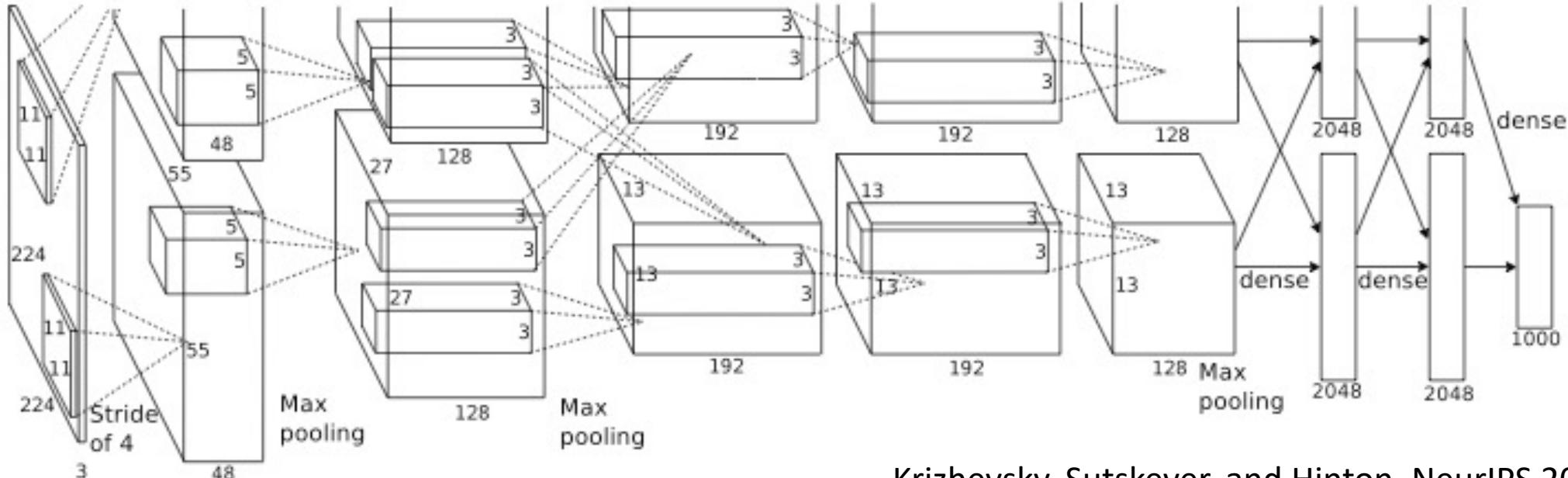
2009
ImageNet

AI Winter

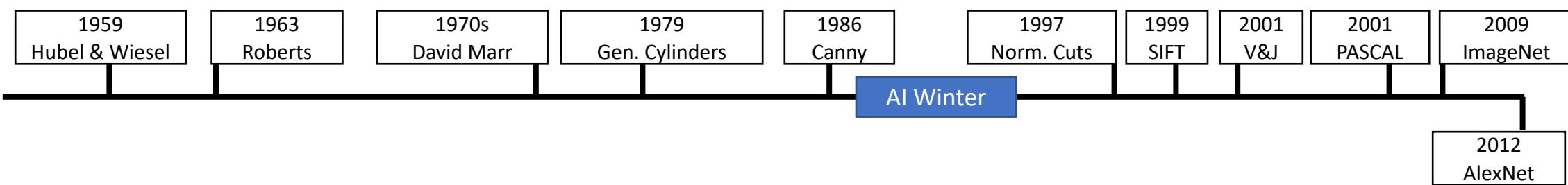
IMAGENET Large Scale Visual Recognition Challenge



AlexNet: Deep Learning Approach for CV



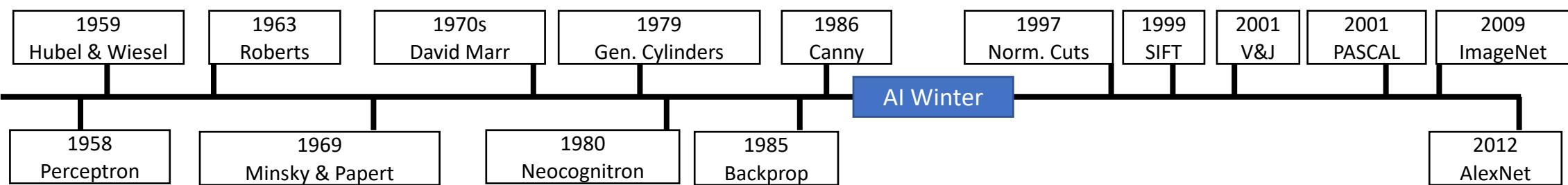
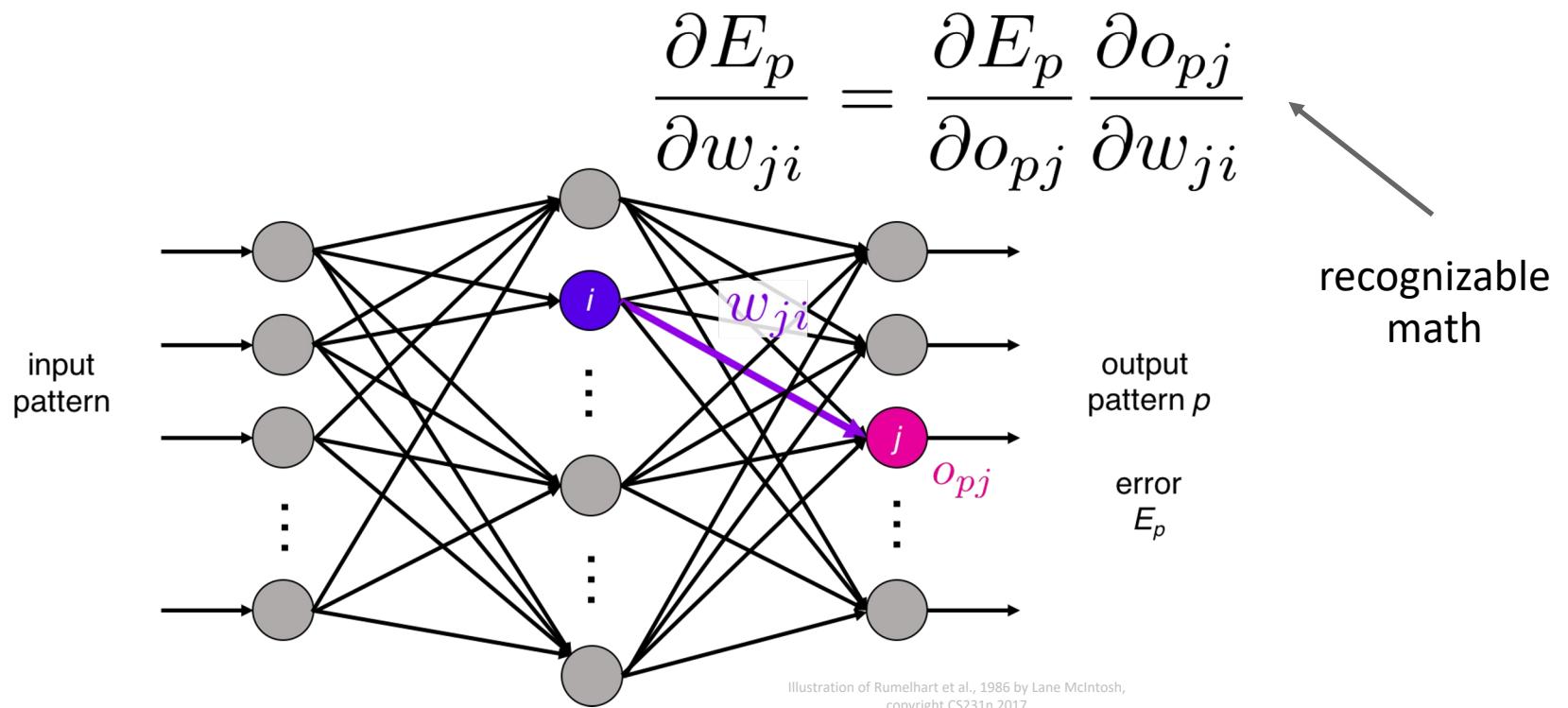
Krizhevsky, Sutskever, and Hinton, NeurIPS 2012



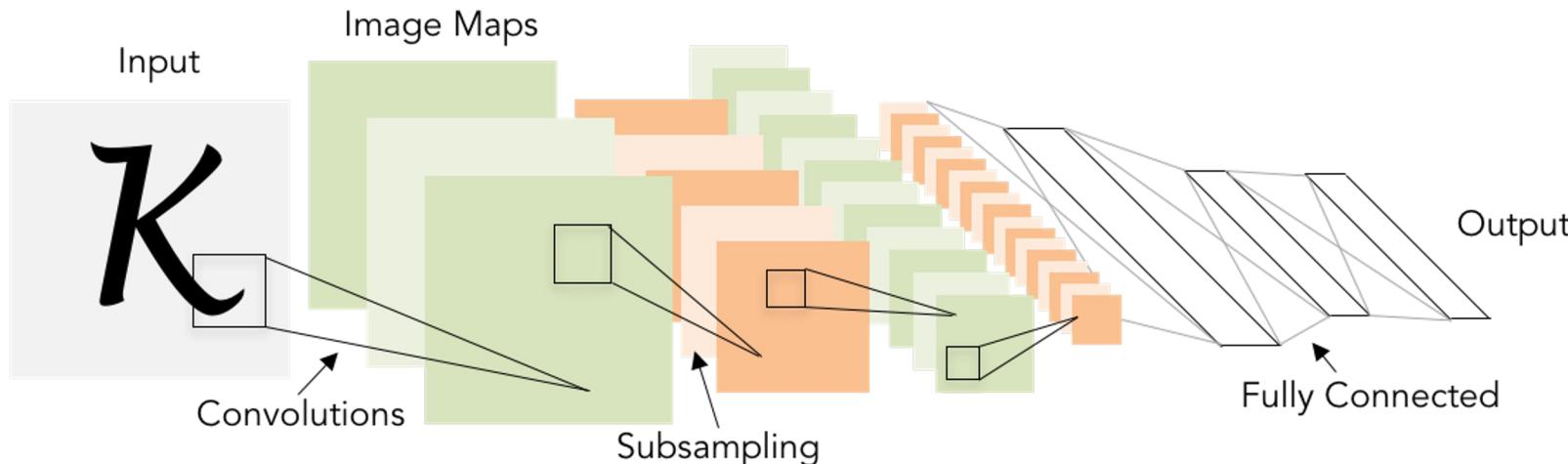
Backprop: Rumelhart, Hinton, and Williams, 1986

Introduced backpropagation
for computing gradients in
neural networks

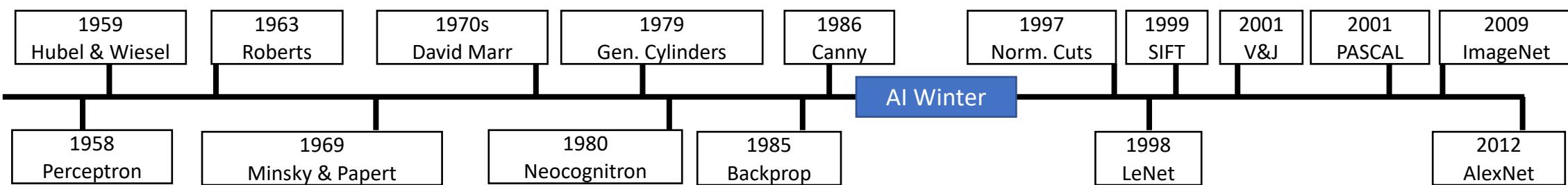
Successfully trained
perceptrons with multiple
layers



Convolutional Networks: LeCun et al, 1998



Applied backprop algorithm to a Neocognitron-like architecture
Learned to recognize handwritten digits
Was deployed in a commercial system by NEC, processed handwritten checks
Very similar to our modern convolutional networks!

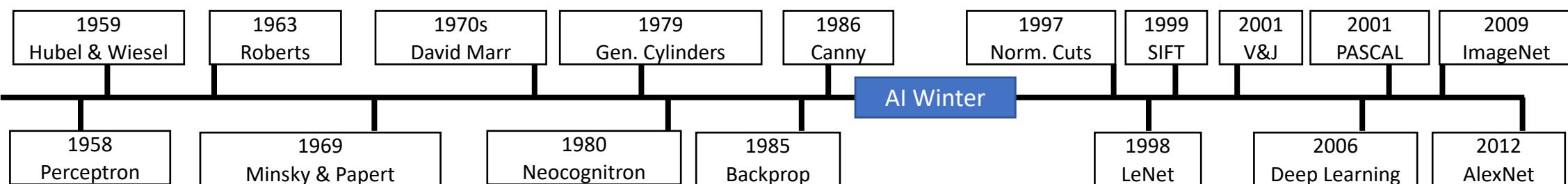
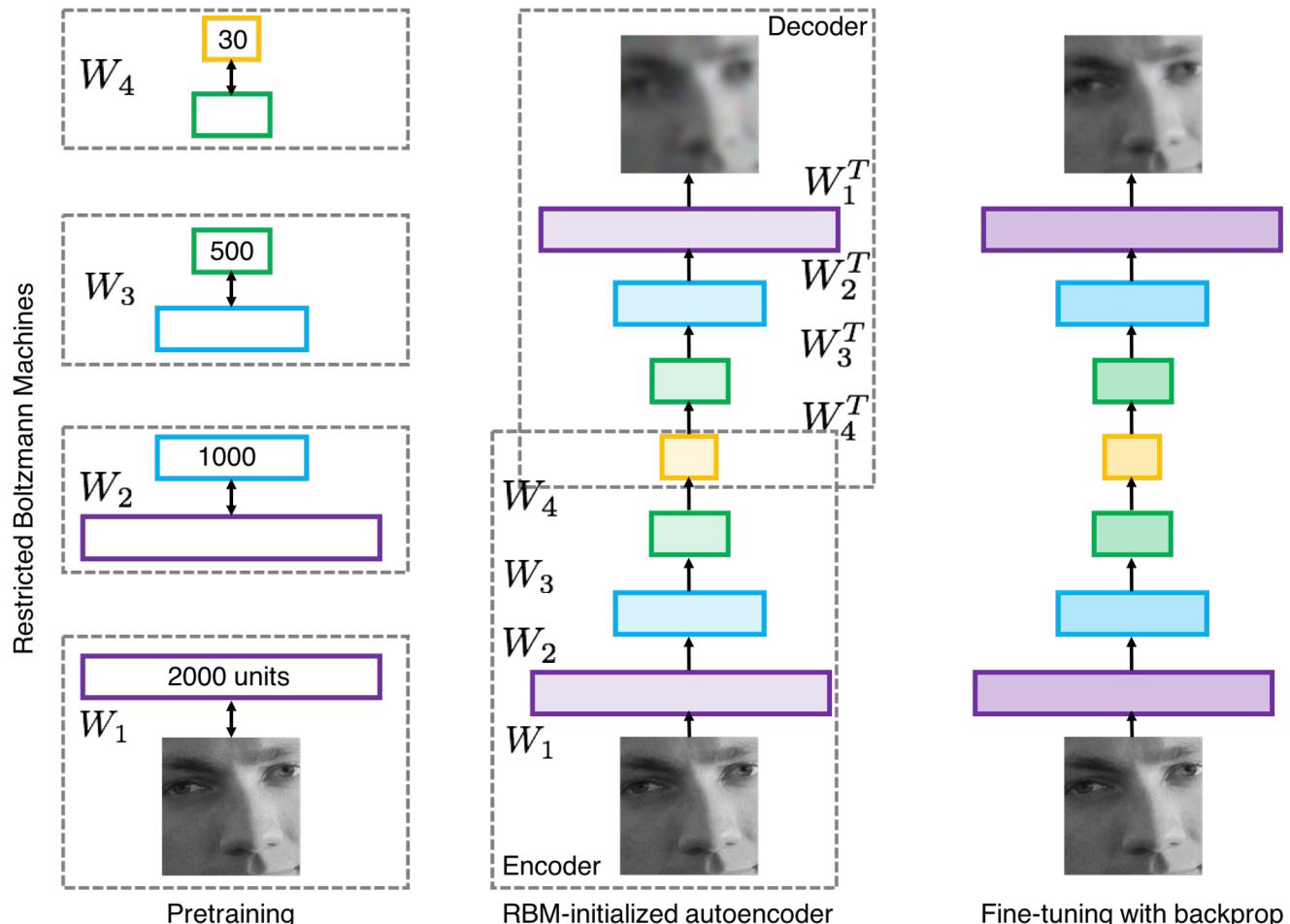


2000s: “Deep Learning”

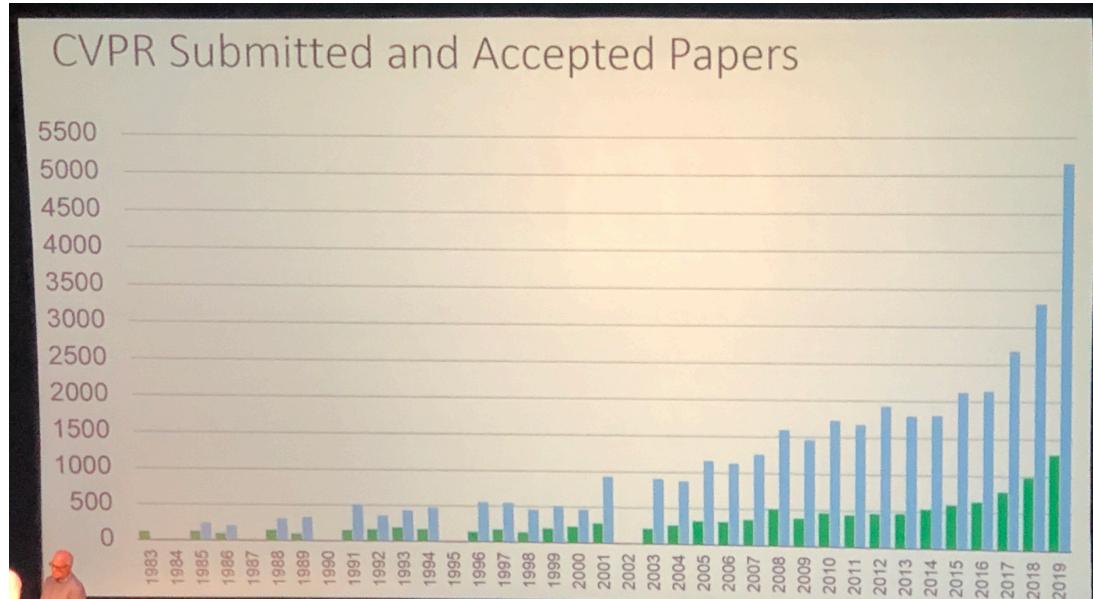
People tried to train neural networks that were deeper and deeper

Not a mainstream research topic at this time

Hinton and Salakhutdinov, 2006
Bengio et al, 2007
Lee et al, 2009
Glorot and Bengio, 2010



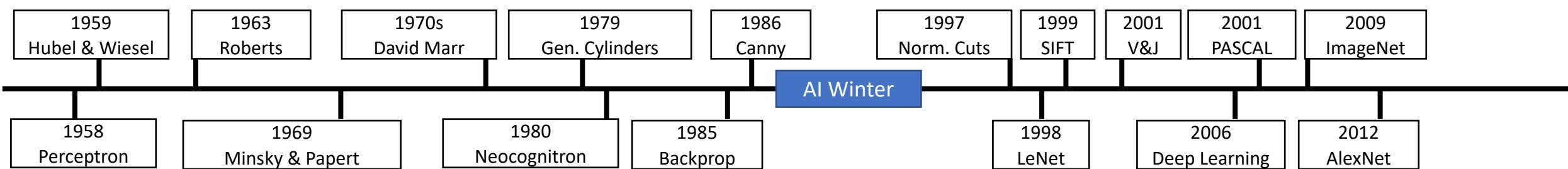
2012 to Present: Deep Learning Explosion



No. CVPR'20 Submissions: ~7500 (+50% increase)
No. CVPR'21 Submissions: 8161 (9% increase)

I had my first paper submission for CVPR'11
~1600 submissions

Publications at top Computer Vision conference CVPR



2012 to Present: Neural Nets are everywhere

Image Classification

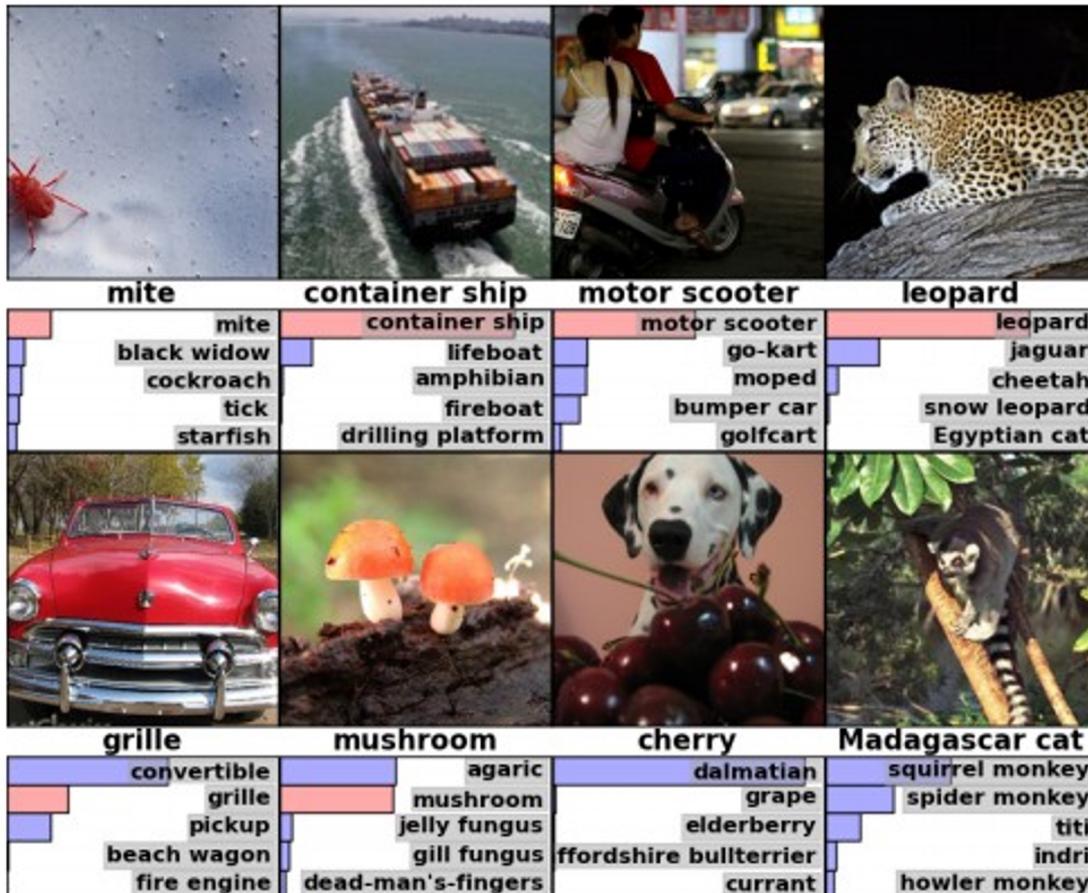
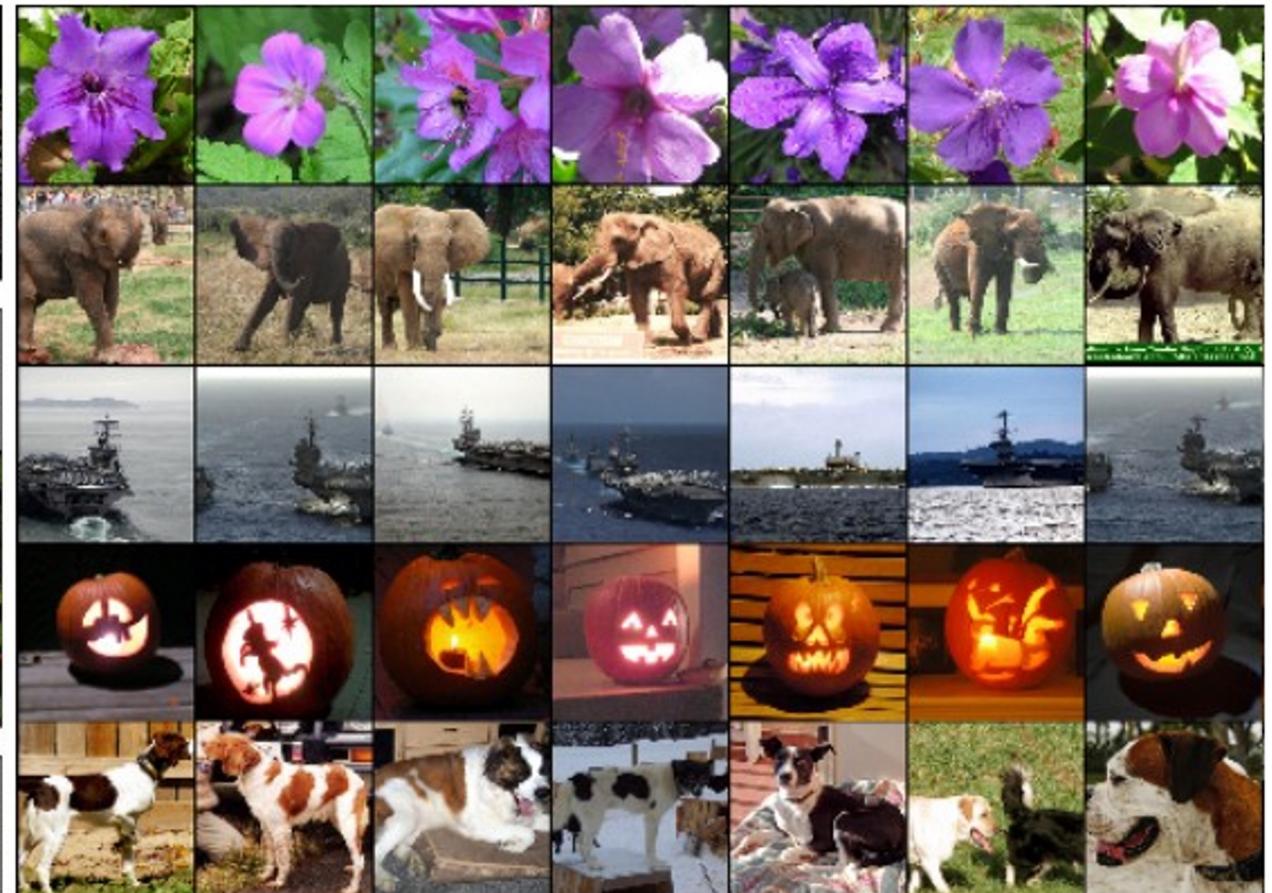


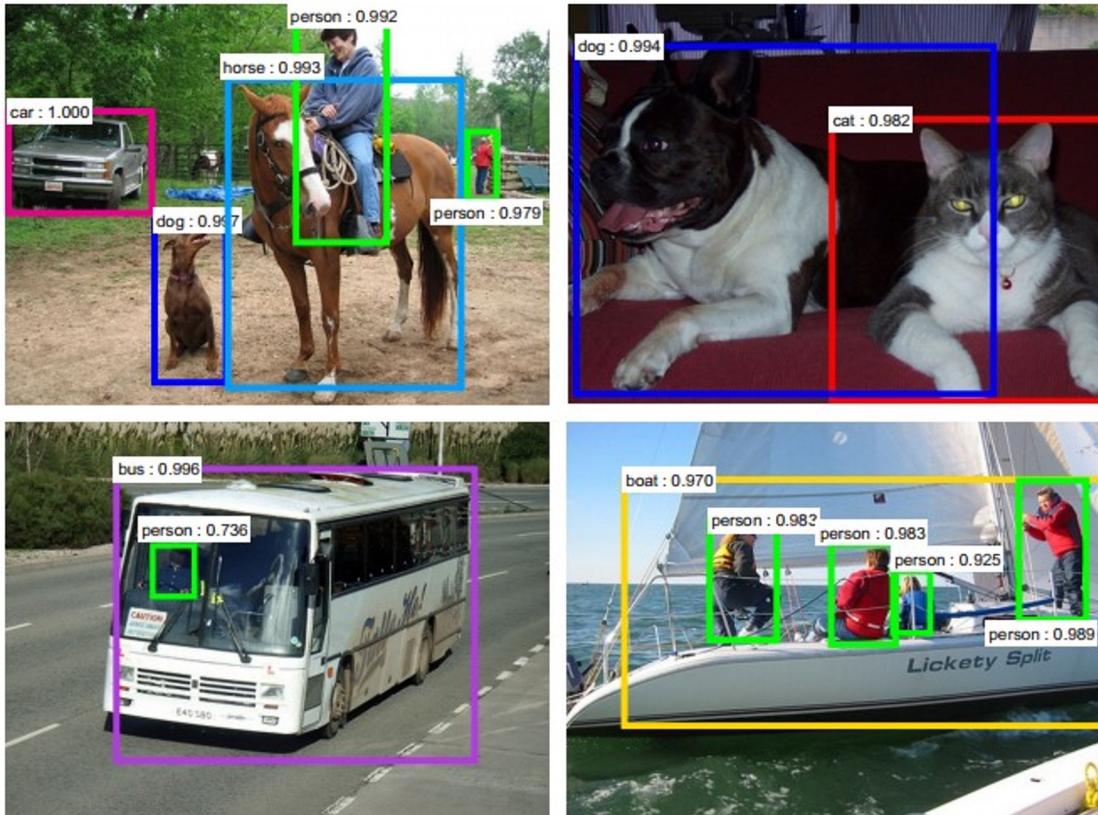
Image Retrieval



Figures copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

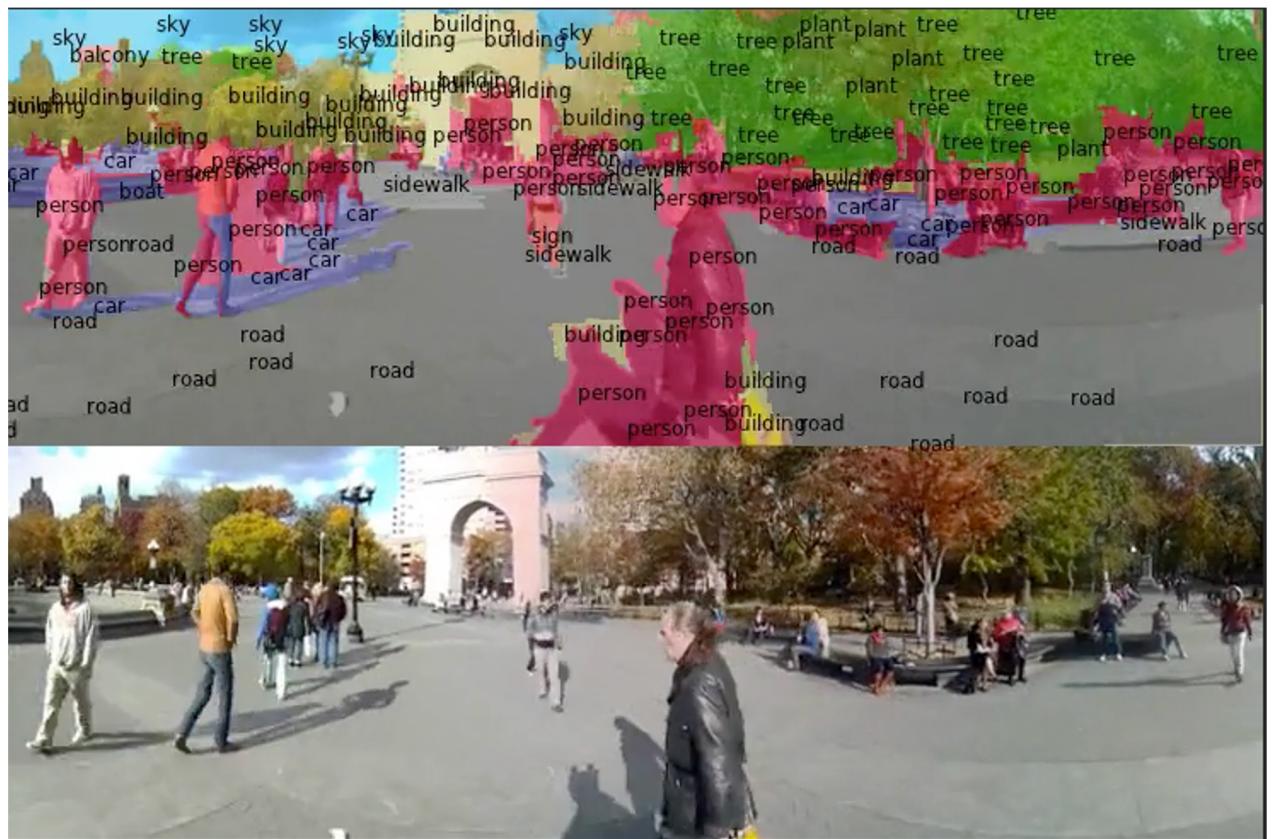
2012 to Present: Neural Nets are everywhere

Object Detection



Ren, He, Girshick, and Sun, 2015

Image Segmentation



Fabaret et al, 2012

Lecture 2: Image Classification

Image Classification: A core computer vision task

Input: image



Output: Assign image to one of a fixed set of categories

chair

bed

sofa

table

cabinet

Many challenges for image classification

Challenges: Viewpoint Variation



Challenges: Fine-Grained Categories

Office chairs



Wooden chairs



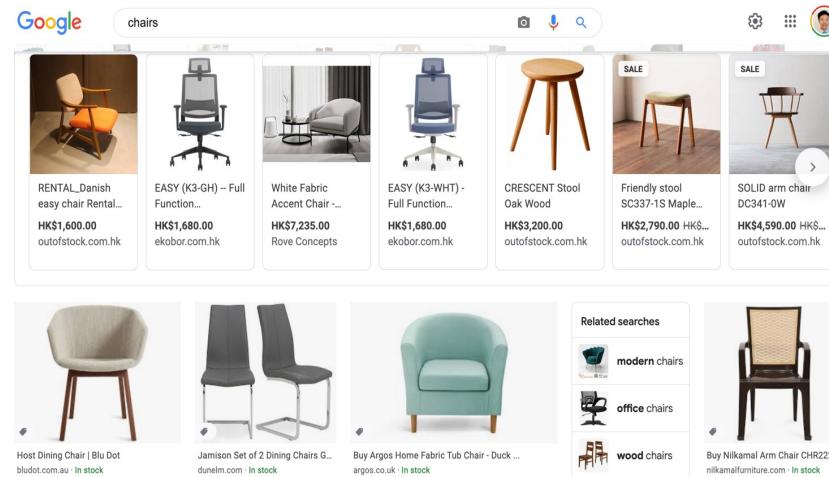
Avocado chairs?



LÄNGFJÄLL Office chair wit...

Ergonomic Chair - HOMELE...

Challenges: Intraclass Variation



Challenges: Objects in Scene Context

Occlusion, non-canonical view, clutter



Many challenges for image classification

Challenges: Domain Changes



Challenges: Functionality

Definition of a chair: anything people can sit?

Chairless chair



Chair as a weapon?



Chair as a transportation?



[Handy Geng](#)

Challenges: Cross-class similarity



Challenges: Variation in Materials and Textures



Machine Learning: Data-Driven Approach

1. Collect a dataset of images and labels
2. Use Machine Learning to train a classifier
3. Evaluate the classifier on new images

Example training set

```
def train(images, labels):  
    # Machine learning!  
    return model
```

```
def predict(model, test_images):  
    # Use model to predict labels  
    return test_labels
```

Highway



Playground



Mountain



Forest



Distance Metric to compare images

L1 distance:

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

test image				training image				pixel-wise absolute value differences			
56	32	10	18	10	20	24	17	46	12	14	1
90	23	128	133	8	10	89	100	82	13	39	33
24	26	178	200	12	16	178	170	12	10	0	30
2	0	255	220	4	32	233	112	2	32	22	108

-

add → 456

```
import numpy as np

class NearestNeighbor:
    def __init__(self):
        pass

    def train(self, X, y):
        """ X is N x D where each row is an example. Y is 1-dimension of size N """
        # the nearest neighbor classifier simply remembers all the training data
        self.Xtr = X
        self.ytr = y

    def predict(self, X):
        """ X is N x D where each row is an example we wish to predict label for """
        num_test = X.shape[0]
        # lets make sure that the output type matches the input type
        Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

        # loop over all test rows
        for i in xrange(num_test):
            # find the nearest training image to the i'th test image
            # using the L1 distance (sum of absolute value differences)
            distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
            min_index = np.argmin(distances) # get the index with smallest distance
            Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

        return Ypred
```

Nearest Neighbor Classifier

Memorize training data

```

import numpy as np

class NearestNeighbor:
    def __init__(self):
        pass

    def train(self, X, y):
        """ X is N x D where each row is an example. Y is 1-dimension of size N """
        # the nearest neighbor classifier simply remembers all the training data
        self.Xtr = X
        self.ytr = y

    def predict(self, X):
        """ X is N x D where each row is an example we wish to predict label for """
        num_test = X.shape[0]
        # lets make sure that the output type matches the input type
        Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

        # loop over all test rows
        for i in xrange(num_test):
            # find the nearest training image to the i'th test image
            # using the L1 distance (sum of absolute value differences)
            distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
            min_index = np.argmin(distances) # get the index with smallest distance
            Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

        return Ypred

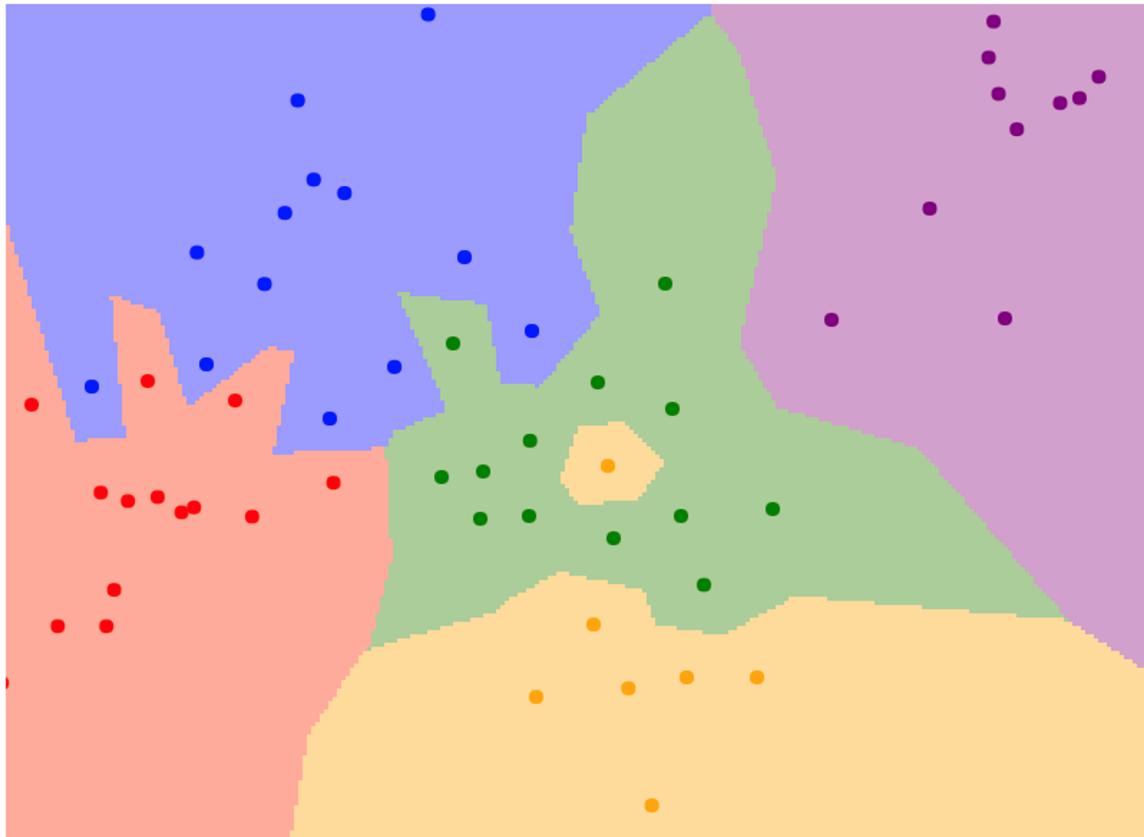
```

Nearest Neighbor Classifier

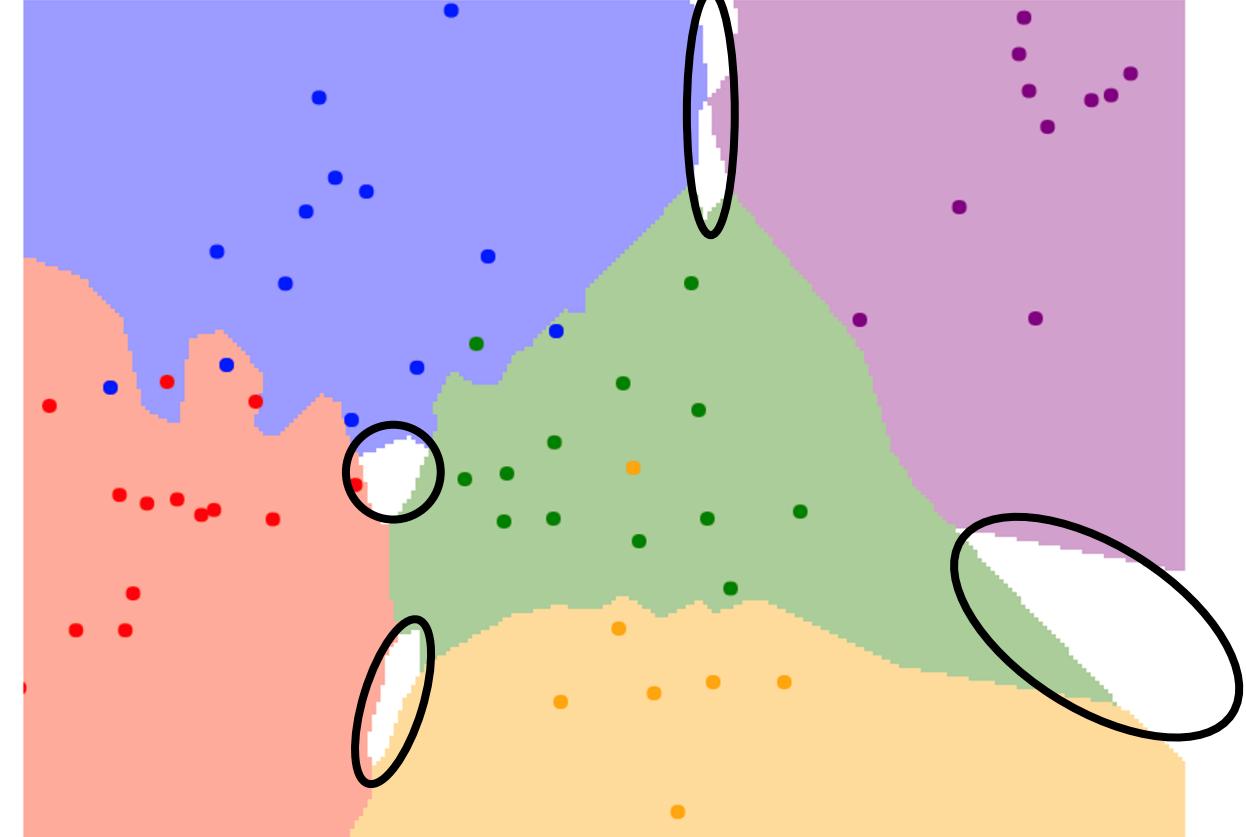
For each test image:
 Find nearest training image
 Return label of nearest image

K-Nearest Neighbors: hyper-parameter K

$K = 1$



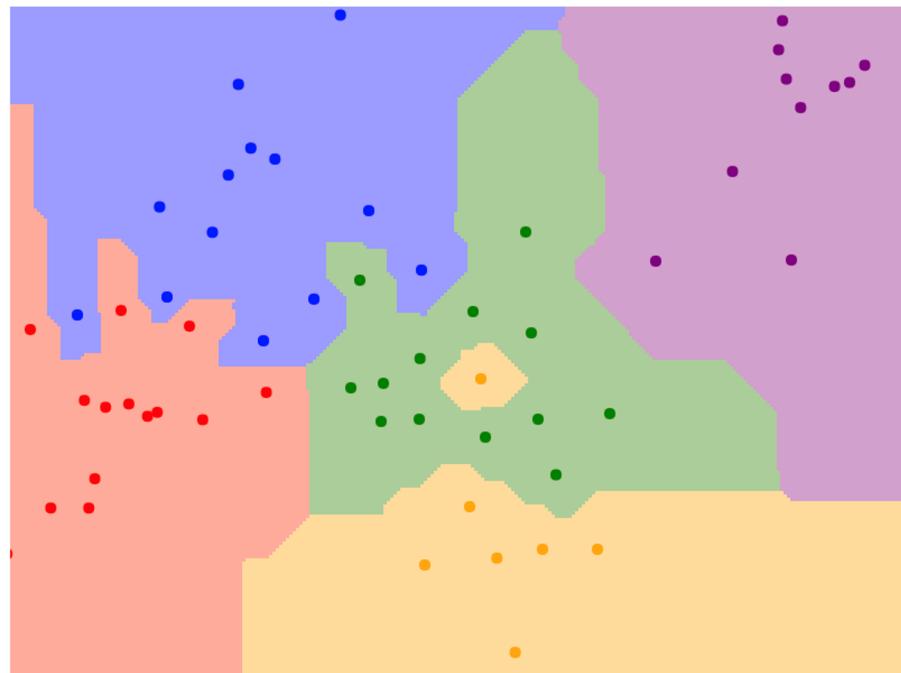
$K = 3$



K-Nearest Neighbors: Distance Metric

L1 (Manhattan) distance

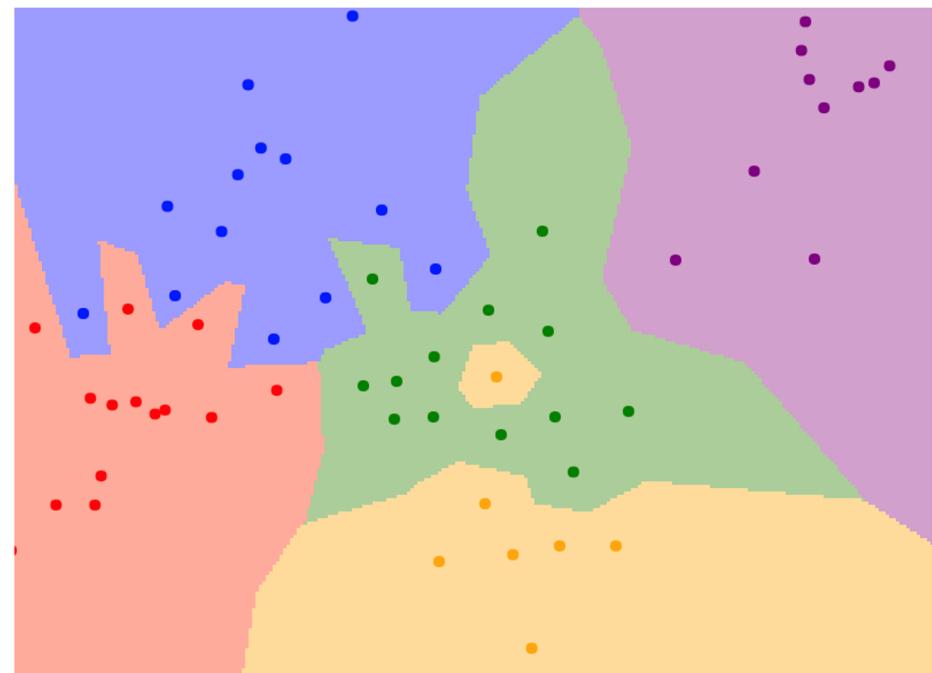
$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$



L2 (Euclidean) distance

$$d_1(I_1, I_2) = \left(\sum_p (I_1^p - I_2^p)^2 \right)^{\frac{1}{2}}$$

$K = 1$



Setting Hyperparameters

Idea #1: Choose hyperparameters that work best on the data

BAD: K = 1 always works perfectly on training data

Your Dataset

Idea #2: Split data into **train** and **test**, choose hyperparameters that work best on test data

BAD: No idea how algorithm will perform on new data

train

test

Idea #3: Split data into **train**, **val**, and **test**; choose hyperparameters on val and evaluate on test

Better!

train

validation

test

Setting Hyperparameters

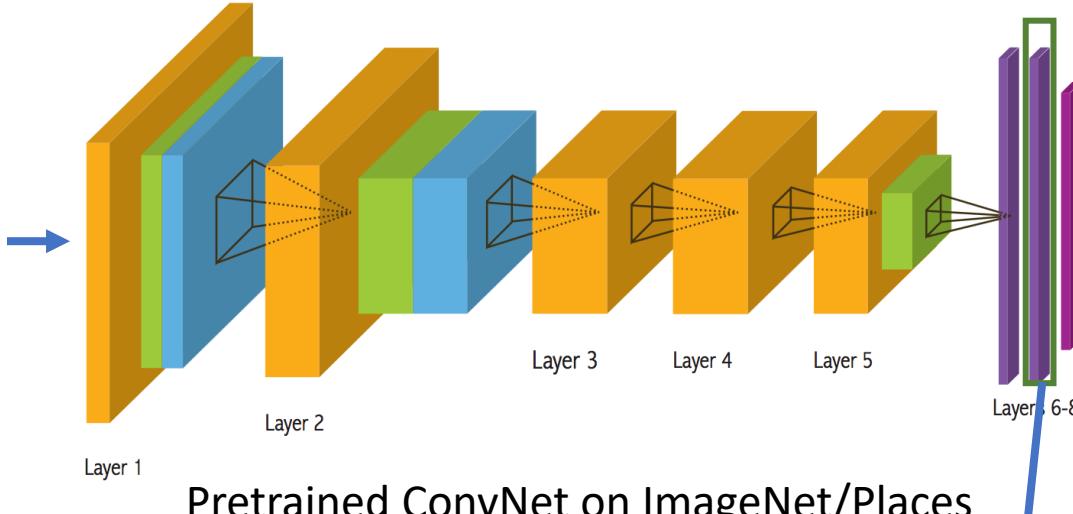
Your Dataset

Idea #4: Cross-Validation: Split data into **folds**, try each fold as validation and average the results

fold 1	fold 2	fold 3	fold 4	fold 5	test
fold 1	fold 2	fold 3	fold 4	fold 5	test
fold 1	fold 2	fold 3	fold 4	fold 5	test

Useful for small datasets, but (unfortunately) not used too frequently in deep learning

Using pre-trained ConvNet features



Pretrained ConvNet on ImageNet/Places

Pull out the second last layer's activation
as feature
* Last layer is the probability output

Nearest Neighbor with ConvNet features works well!



Devlin et al, "Exploring Nearest Neighbor Approaches for Image Captioning", 2015

Lecture 3: Linear Classifier

Parametric Approach: Linear Classifier

Image



$$f(\mathbf{x}, \mathbf{W}) = \mathbf{W}\mathbf{x}$$

$$f(\mathbf{x}, \mathbf{W})$$

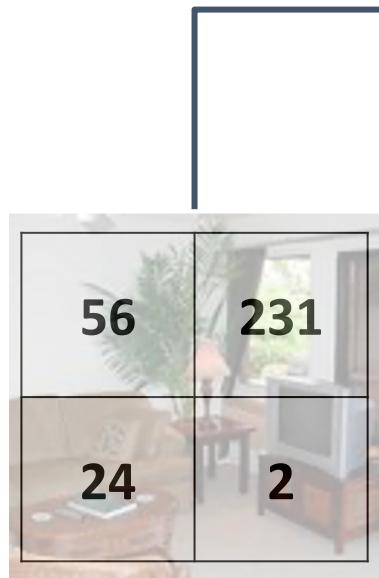
20 numbers giving
class scores

Array of **32x32x3** numbers
(3072 numbers total)

W
parameters
or weights

Example for 2x2 image, 3 classes (livingroom/highway/mountain)

Stretch 4 pixels into a vector



Input image
(2, 2)

0.2	-0.5	0.1	2.0
1.5	1.3	2.1	0.0
0	0.25	0.2	-0.3

W (3, 4)

56
231
24
2

(4,)

$$f(x, W) = Wx + b$$

1.1
3.2
-1.2
0.25

+

b
(3,)

-96.8
437.9
61.95

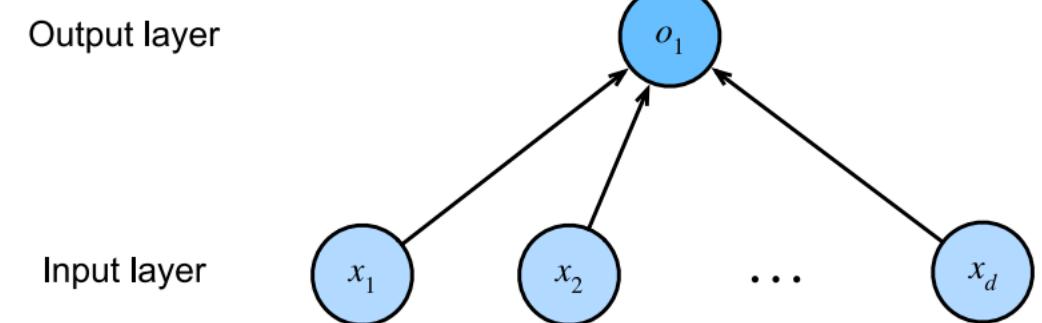
(3,)

Linear Classifier is a single-layer neural network

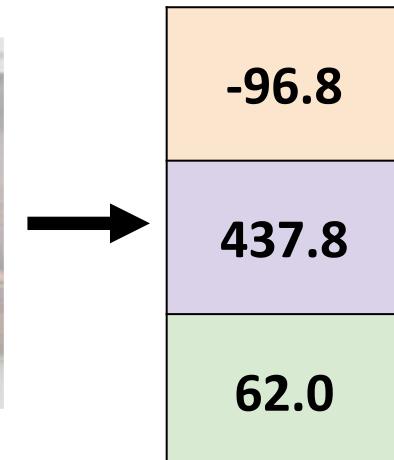


Outdoor-ness

→ 0.15

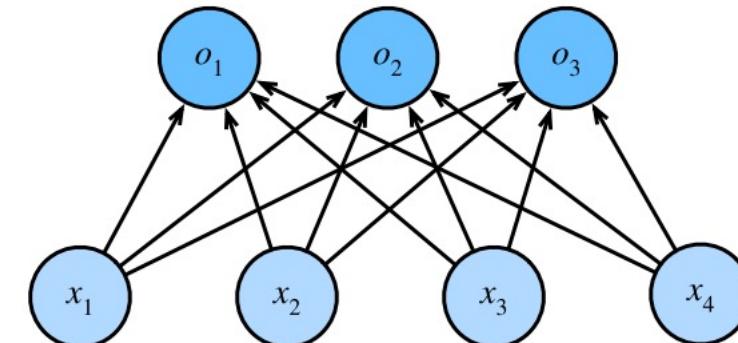


3 Scene Classification



Output layer

Input layer



Interpreting a Linear Classifier

$$f(x, W) = Wx + b$$

Stretch pixels into column

56	231
24	2

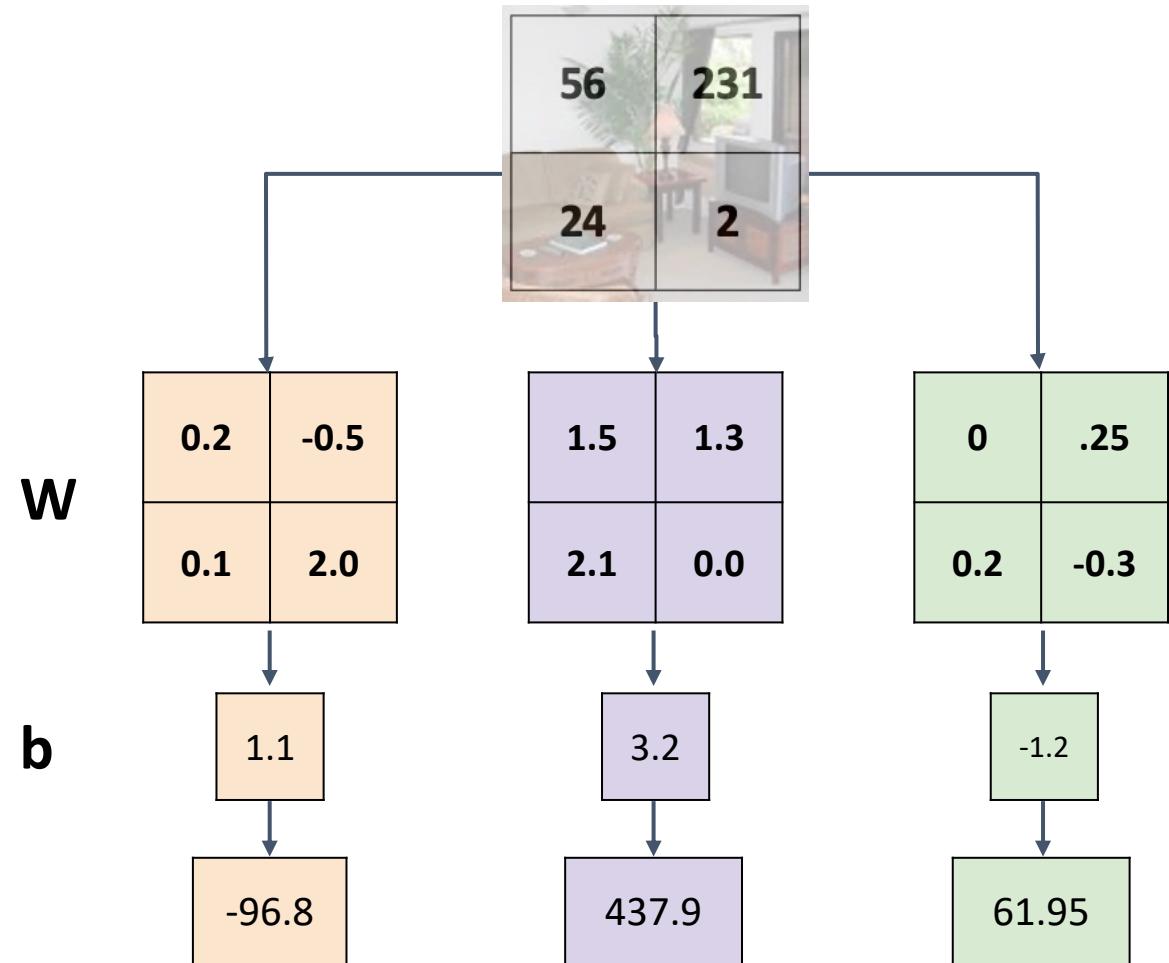
Input image (2, 2)

$W \quad (3, 4)$

$$\begin{matrix} 0.2 & -0.5 & 0.1 & 2.0 \\ 1.5 & 1.3 & 2.1 & 0.0 \\ 0 & 0.25 & 0.2 & -0.3 \end{matrix} + \begin{matrix} 56 \\ 231 \\ 24 \\ 2 \end{matrix} = \begin{matrix} 1.1 \\ 3.2 \\ -1.2 \end{matrix} \quad \begin{matrix} -96.8 \\ 437.9 \\ 61.95 \end{matrix}$$

$b \quad (3,)$

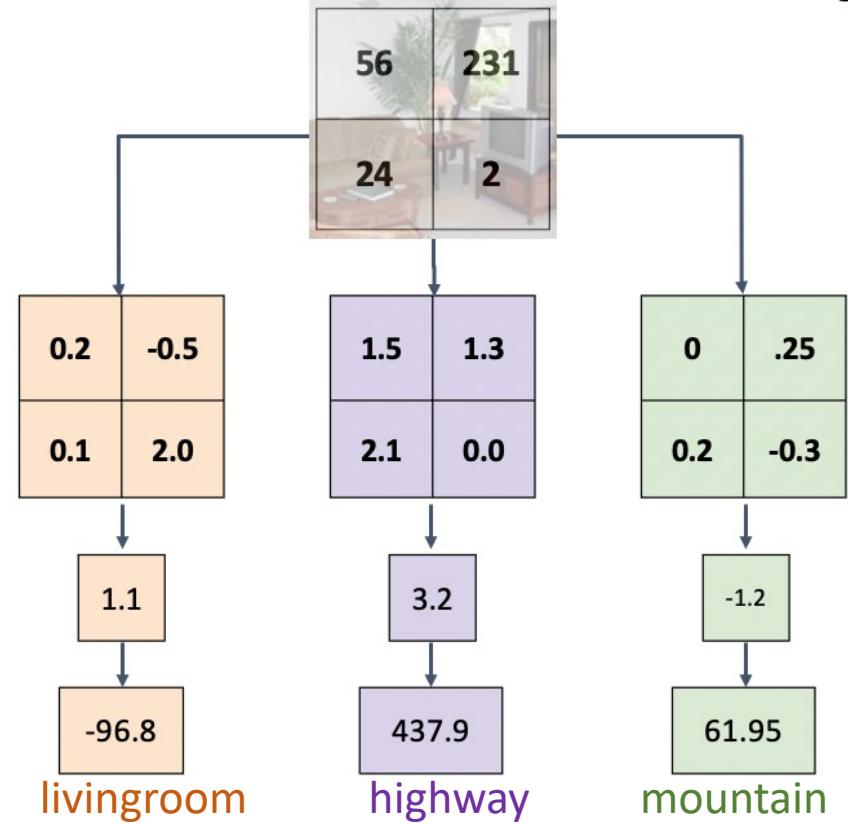
(4,)



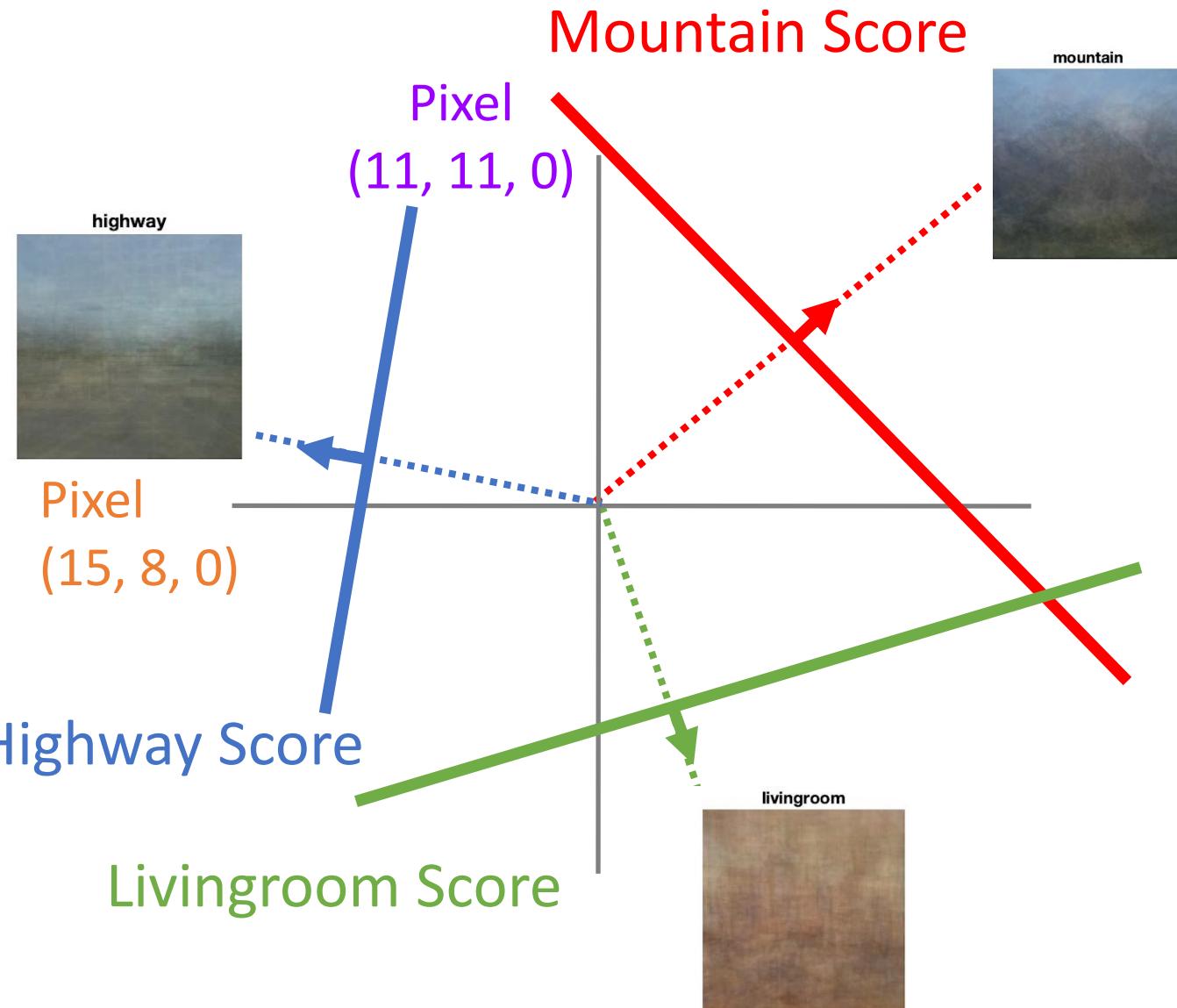
Interpreting a Linear Classifier

Linear classifier has one “template” per category, then compute the correlation.

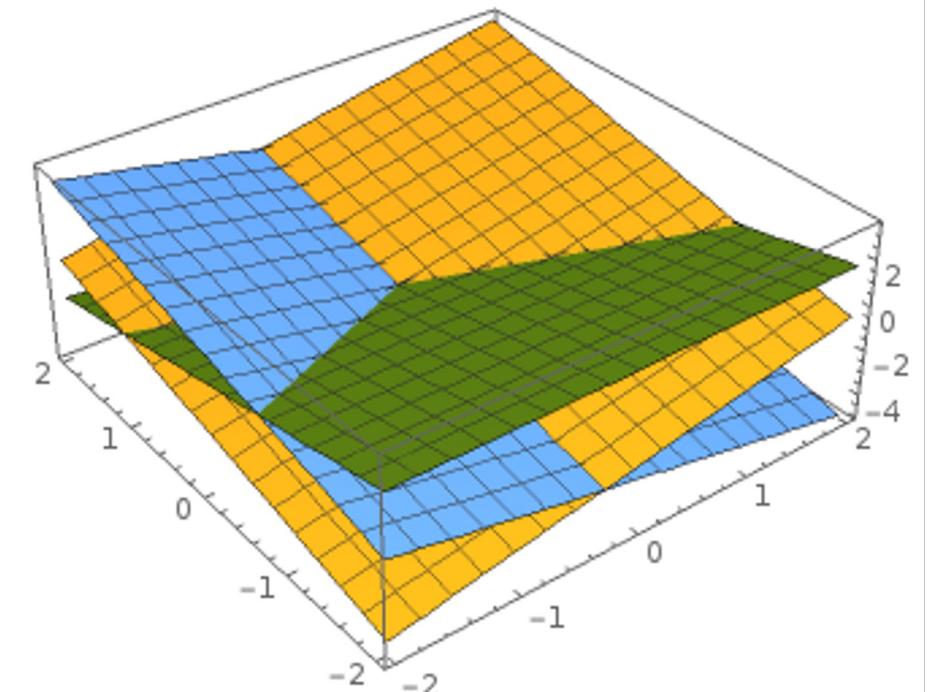
Drawback: A single template cannot capture multiple modes of the data



Interpreting a Linear Classifier: Geometric Viewpoint



Hyperplanes carving up a high-dimensional space



Plot created using [Wolfram Cloud](#)

Loss Function

A **loss function** tells how good our current classifier is

Low loss = good classifier
High loss = bad classifier

(Also called: **objective function**; **cost function**)

Negative loss function sometimes called **reward function**, **profit function**, **utility function**, **fitness function**, etc

Given a dataset of examples

$$\{(x_i, y_i)\}_{i=1}^N$$

Where x_i is image and
 y_i is (integer) label

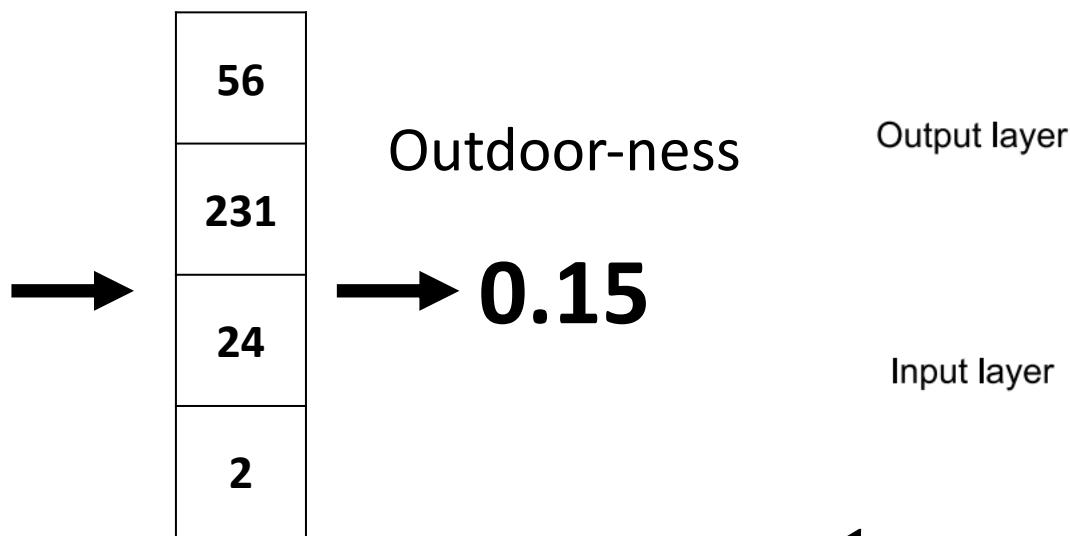
Loss for a single example is

$$L_i(f(x_i, W), y_i)$$

Loss for the dataset is average of per-example losses:

$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$

Linear Regression Loss



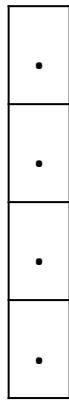
Squared loss for one sample: $L_i(W) = \frac{1}{2}(f(x_i, W) - y_i)^2$

Squared loss for all the training samples:

$$L(W) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2}(f(x_i, W) - y_i)^2$$

Learning objective: $W^* = \operatorname{argmin}_W L(W)$

Linear Regression Loss



Model output

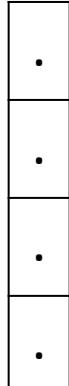
→ **0.15**

Loss

$$\frac{1}{2}(0.15 - 1)^2$$

Ground truth

1



→ **0.05**

$$\frac{1}{2}(0.05 - 0)^2$$

0

Cross-Entropy Loss (Multinomial Logistic Regression)



Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

Probabilities
must be ≥ 0

$$P(Y = k | X = x_i) = \frac{\exp(s_k)}{\sum_j \exp(s_j)}$$

Softmax
function

Probabilities
must sum to 1

$$L_i = -\log P(Y = y_i | X = x_i)$$

Bridge
Mountain
Coast

3.2
5.1
-1.7

Unnormalized log-probabilities / logits

\exp

24.5
164.0
0.18

unnormalized probabilities

normalize

0.13
0.87
0.00

probabilities

$$L_i = -\log(0.13) \\ = 2.04$$

Maximum Likelihood Estimation
Choose weights to maximize the likelihood of the observed data

Cross-Entropy Loss (Multinomial Logistic Regression)

One-hot encoding for the categorical label of each sample



Bridge	1	1	0	0
--------	---	---	---	---

Mountain	0	0	1	0
----------	---	---	---	---

Coast	0	0	0	1
-------	---	---	---	---

Cross-Entropy Loss (Multinomial Logistic Regression)



Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

Probabilities
must be ≥ 0

$$P(Y = k | X = x_i) = \frac{\exp(s_k)}{\sum_j \exp(s_j)}$$

Softmax
function

Probabilities
must sum to 1

$$L_i = -\log P(Y = y_i | X = x_i)$$

Bridge

3.2

\rightarrow
 \exp

5.1

24.5

normalize
 \rightarrow

164.0

0.13

Compare \leftarrow

-1.7

0.18

Unnormalized log-
probabilities / logits

unnormalized
probabilities

0.87

0.00

probabilities

$$D_{KL}(P || Q) = \sum_y P(y) \log \frac{P(y)}{Q(y)}$$

1.00

0.00

0.00

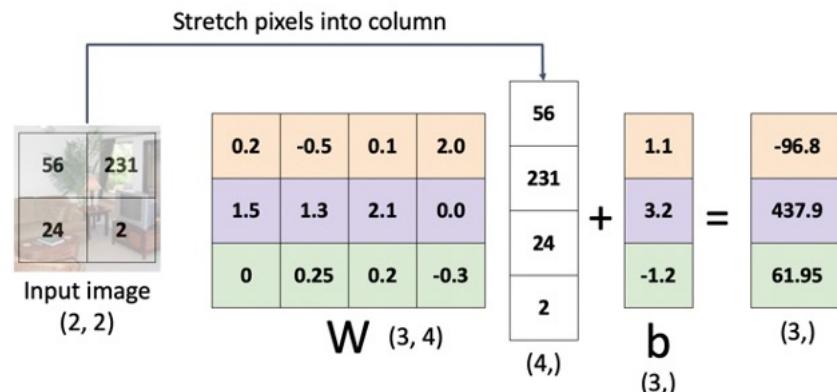
Correct
probs

Linear NN/Classifier: Three Viewpoints

Equation Viewpoint

$$f(x, W) = Wx$$

$$f(x, W) = Wx + b$$



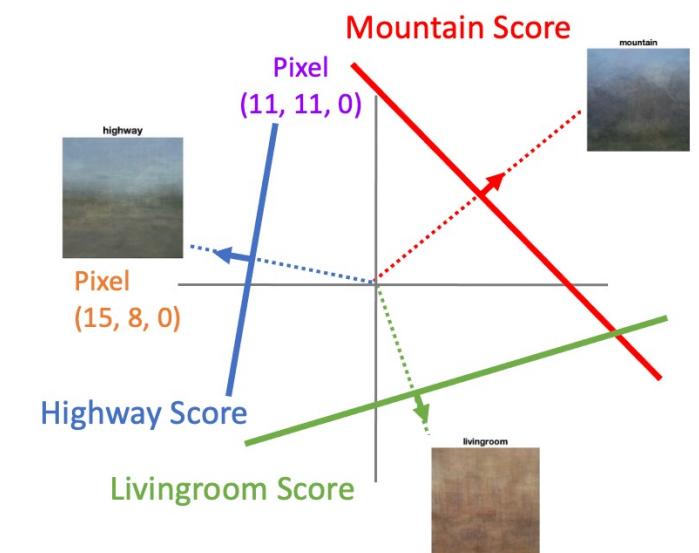
Visualization Viewpoint

One template per class



Geometric Viewpoint

Hyperplanes cutting up space



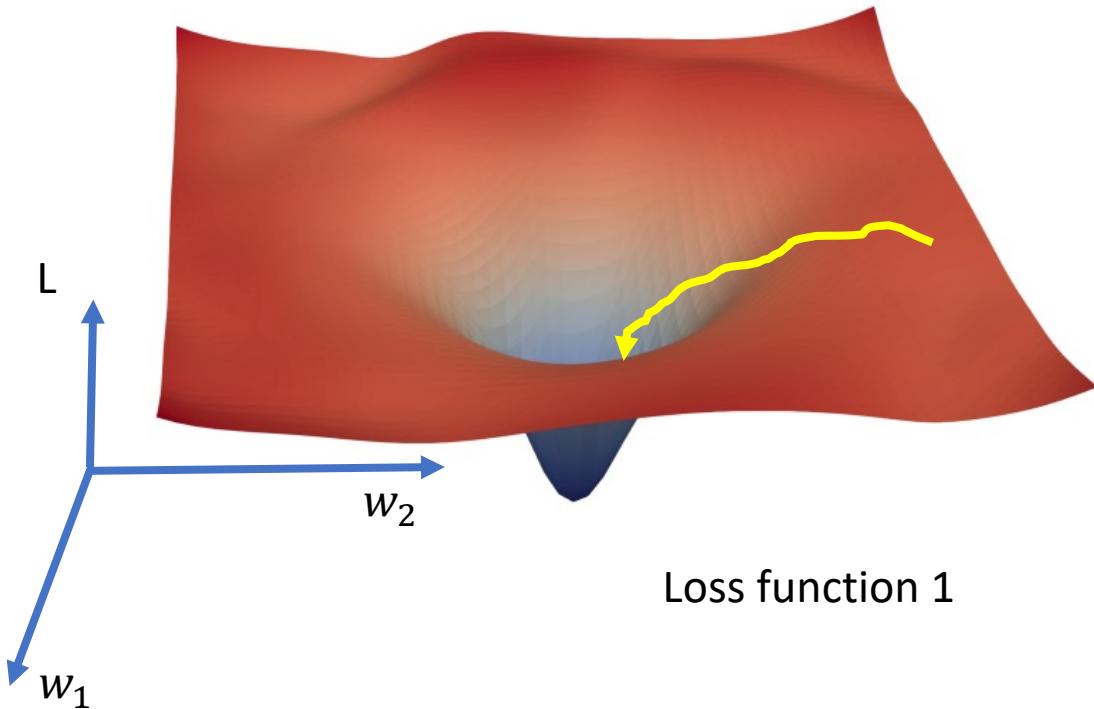
Lecture 4: optimization algorithms

Optimization

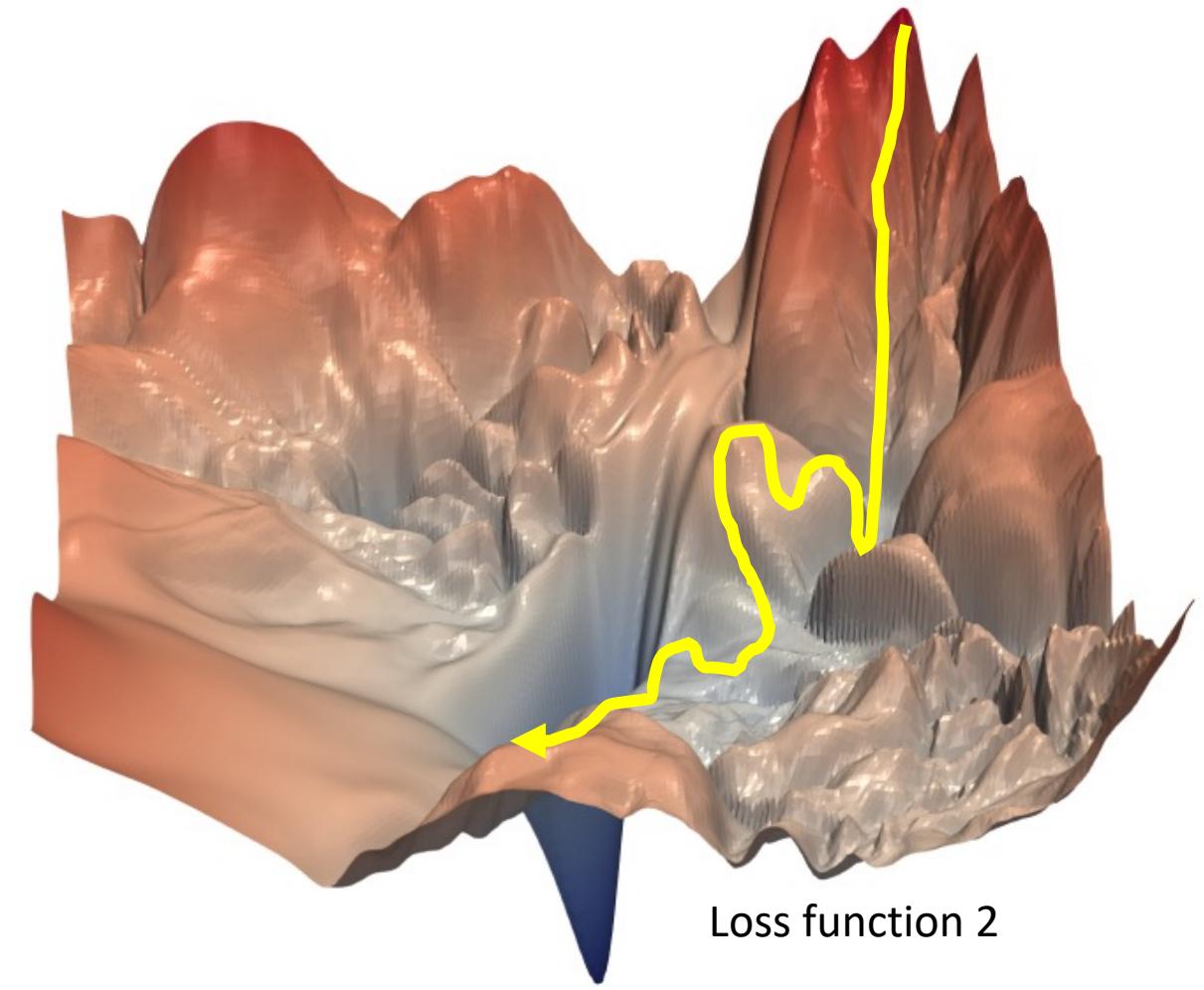
$$w^* = \arg \min_w L(w)$$

Loss Landscape $L = f(w_1, w_2)$

$$w^* = \arg \min_w L(w)$$



Loss function 1

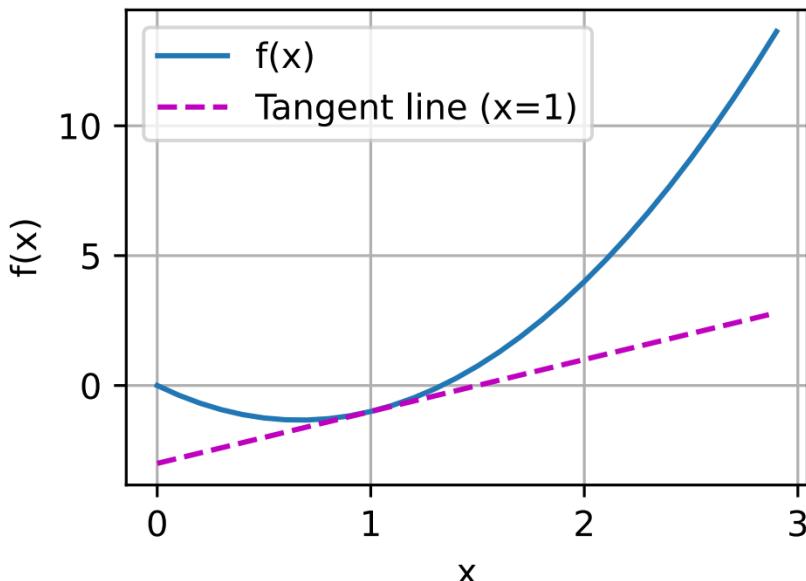


Loss function 2

Follow the slope

In 1-dimension, the **derivative** of a function gives the slope:

$$\frac{df}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$



Follow the slope

In 1-dimension, the **derivative** of a function gives the slope:

$$\frac{df}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_n} \right]^\top,$$

Gradient Descent

Iteratively step in the direction of
the negative gradient
(direction of local steepest descent)

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
    dw = compute_gradient(loss_fn, data, w)
    w -= learning_rate * dw
```

Hyperparameters:

- Weight initialization method
- Number of steps
- Learning rate

One-Dimensional Gradient Descend: Why the gradient descend algorithm may reduce the value of the objective function

Using a Taylor expansion, we have: $f(x + \epsilon) = f(x) + \epsilon f'(x) + O(\epsilon^2)$

Let $\epsilon = -\eta f'(x)$ we can have: $f(x - \eta f'(x)) = f(x) - \eta f'^2(x) + O(\eta^2 f'^2(x))$

Thus $f(x - \eta f'(x)) \leq f(x)$

So we can iterate over $x \leftarrow x - \eta f'(x)$, the value of function $f(x)$ might decline

One-Dimensional Gradient Descend

Gradient Descend

```
def gd(eta, f_grad):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * f_grad(x)
        results.append(float(x))
    print(f'epoch 10, x: {x:f}')
    return results

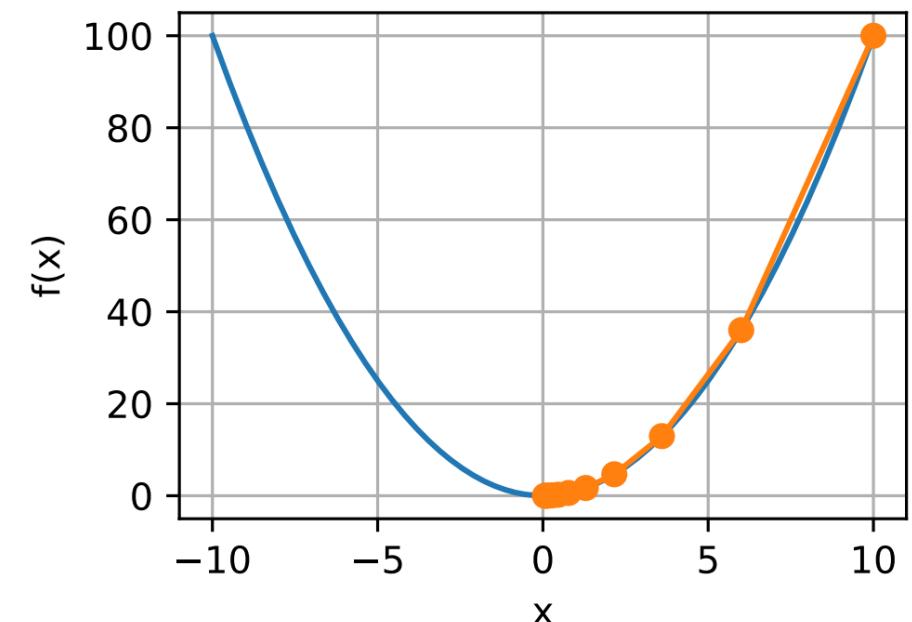
results = gd(0.2, f_grad)
```

```
def f(x): # Objective function
    return x ** 2

def f_grad(x): # Gradient (derivative) of the objective function
    return 2 * x
```

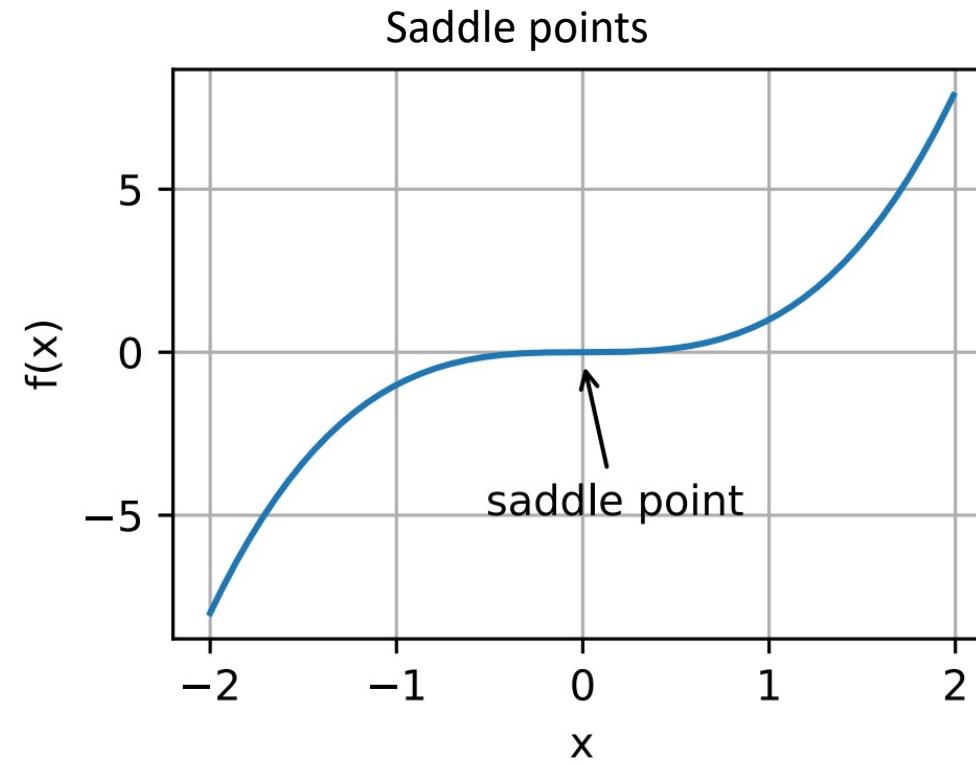
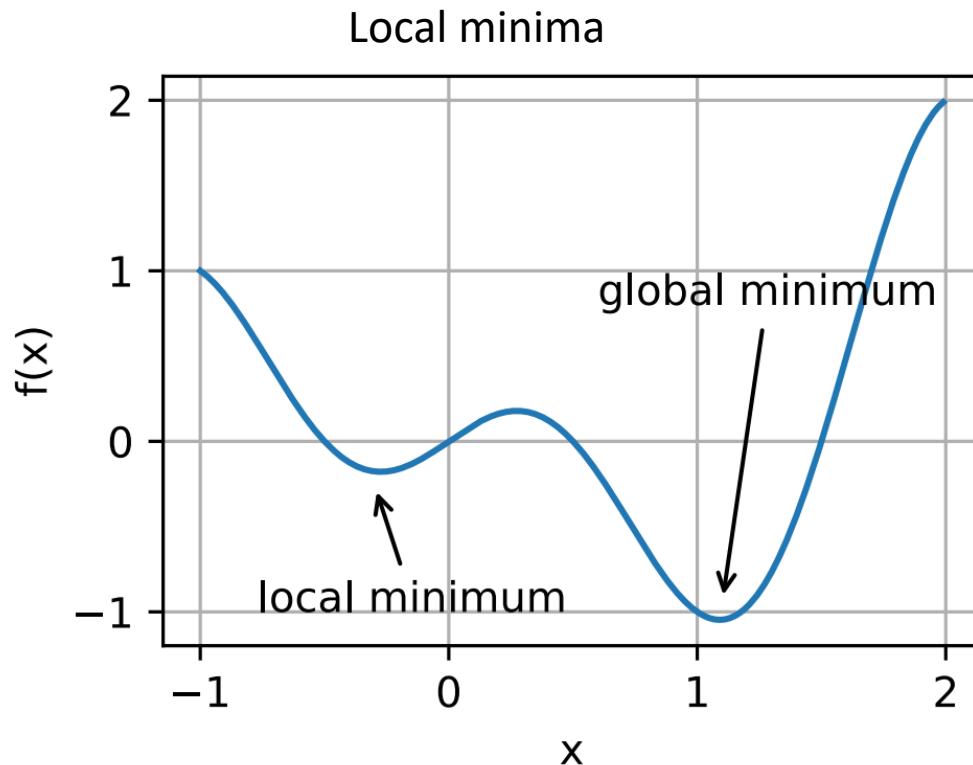
```
def show_trace(results, f):
    n = max(abs(min(results)), abs(max(results)))
    f_line = torch.arange(-n, n, 0.01)
    d2l.set_figsize()
    d2l.plot([f_line, results], [[f(x) for x in f_line],
                                  [f(x) for x in results]], 'x', 'f(x)', fmts=['-', '-o'])

show_trace(results, f)
```



Challenges with Gradient Descend

Gradient becomes zeros or vanishes



Zero gradient, gradient descent gets stuck

Stochastic Gradient Descent (SGD)

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

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

Full sum expensive
when N is large!

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

```
# Stochastic gradient descent
w = initialize_weights()
for t in range(num_steps):
    minibatch = sample_data(data, batch_size)
    dw = compute_gradient(loss_fn, minibatch, w)
    w == learning_rate * dw
```

Hyperparameters:

- Weight initialization
- Number of steps
- Learning rate
- Batch size
- Data sampling

Stochastic Gradient Descent (SGD)

$$\begin{aligned} L(W) &= \mathbb{E}_{(x,y) \sim p_{data}} [L(x, y, W)] \\ &\approx \frac{1}{N} \sum_{i=1}^N L(x_i, y_i, W) \end{aligned}$$

Think of loss as an expectation over the full **data distribution** p_{data}

Approximate expectation via sampling

$$\begin{aligned} \nabla_W L(W) &= \nabla_W \mathbb{E}_{(x,y) \sim p_{data}} [L(x, y, W)] \\ &\approx \sum_{i=1}^N \nabla_w L_W(x_i, y_i, W) \end{aligned}$$

SGD + Momentum

SGD

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

```
for t in range(num_steps):
    dw = compute_gradient(w)
    w -= learning_rate * dw
```

SGD+Momentum

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

```
v = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    v = rho * v + dw
    w -= learning_rate * v
```

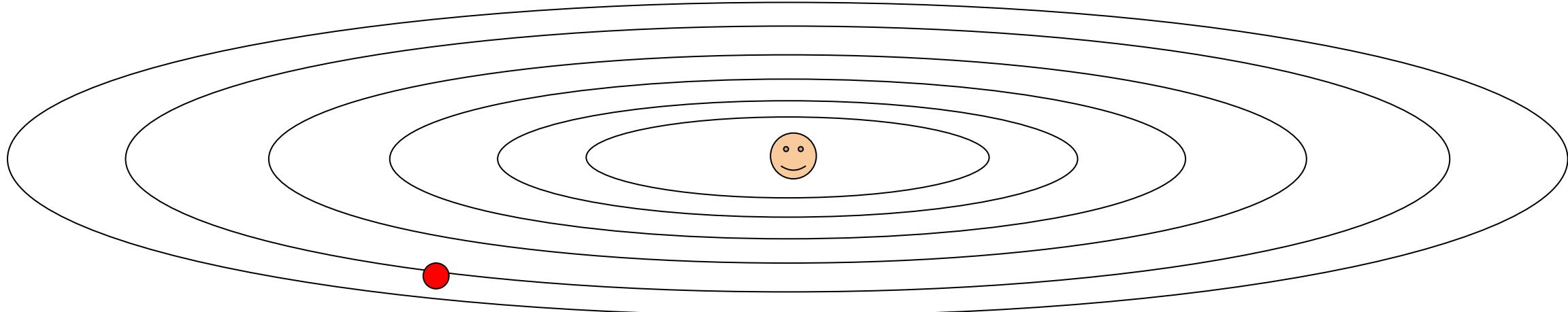
- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

AdaGrad

```
grad_squared = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    grad_squared += dw * dw
    w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

$$\mathbf{g}_t = \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})),$$
$$\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2,$$

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t.$$



RMSProp: “Leaky Adagrad”

```
grad_squared = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    grad_squared += dw * dw
    w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

$$\mathbf{g}_t = \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})),$$
$$\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2,$$
$$\mathbf{w}_t = \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t.$$

AdaGrad

```
grad_squared = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dw * dw
    w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

RMSProp

$$\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1 - \gamma) \mathbf{g}_t^2,$$
$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t.$$



Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    moment1 = beta1 * moment1 + (1 - beta1) * dw
    moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
    w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

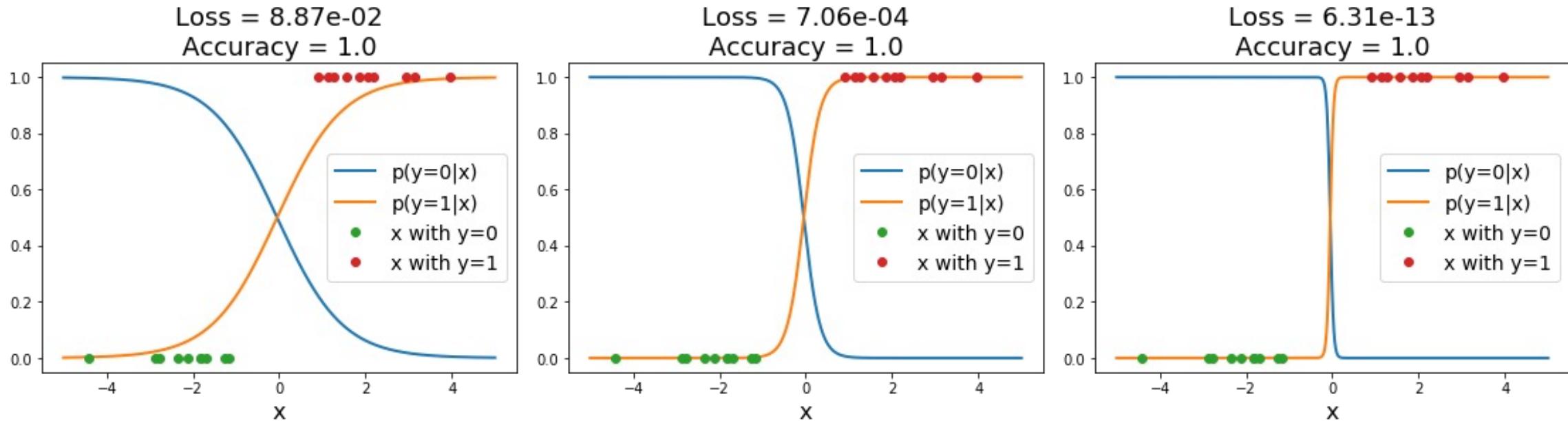
Common choice: $\text{beta1} = 0.9$, $\text{beta2} = 0.999$

Optimization Algorithm Comparison

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

Overfitting

A model is **overfit** when it performs too well on the training data, and has poor performance for unseen data



Both models have perfect accuracy on train data!

Example: Linear classifier with 1D inputs, 2 classes, softmax loss

$$s_i = w_i x + b_i$$
$$p_i = \frac{\exp(s_i)}{\exp(s_1) + \exp(s_2)}$$
$$L = -\log(p_y)$$

Low loss, but unnatural “cliff” between training points

Regularization: Beyond Training Error

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

Data loss: Model predictions should match training data

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

Simple examples

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

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

λ is a hyperparameter giving regularization strength

Regularization: Expressing Preferences

L2 Regularization

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

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

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

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

L2 regularization prefers weights to be “spread out”

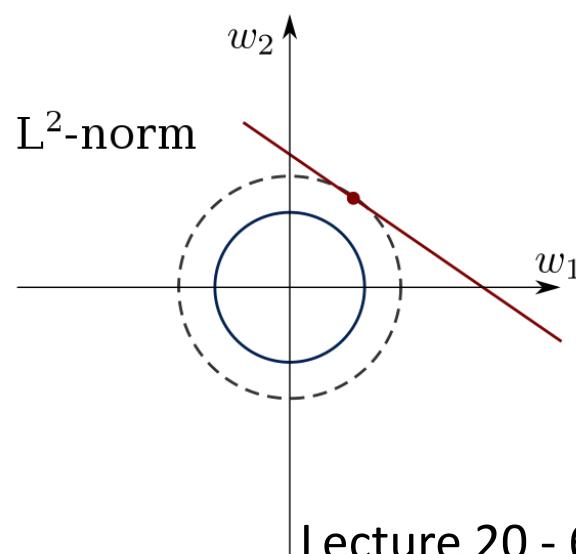
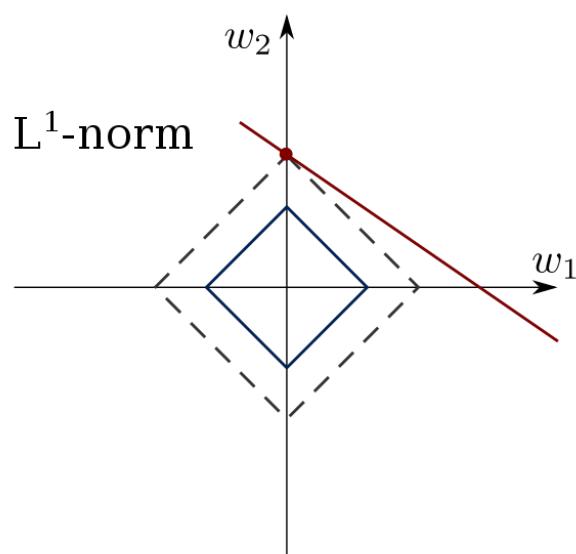
$$w_1^T x = w_2^T x = 1$$

Same predictions, so data loss will always be the same

L1 Regularization versus L2 Regularization

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i) + \sum_{k,l} W_{k.l}^2$$

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i) + \sum_{k,l} |W_{k.l}|$$



L1 regularization leads to more sparse weights

L2 Regularization vs Weight Decay

Optimization Algorithm

$$L(w) = L_{data}(w) + L_{reg}(w)$$

$$g_t = \nabla L(w_t)$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \alpha s_t$$

L2 Regularization and Weight Decay are equivalent for SGD, SGD+Momentum so people often use the terms interchangeably!

L2 Regularization

$$L(w) = L_{data}(w) + \lambda |w|^2$$

$$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \alpha s_t$$

Weight Decay

$$L(w) = L_{data}(w)$$

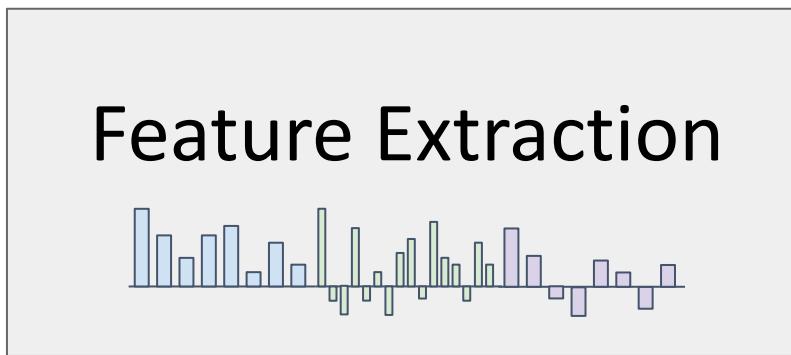
$$g_t = \nabla L_{data}(w_t)$$

$$s_t = \text{optimizer}(g_t) + 2\lambda w_t$$

$$w_{t+1} = w_t - \alpha s_t$$

Lecture 5: Neural Networks

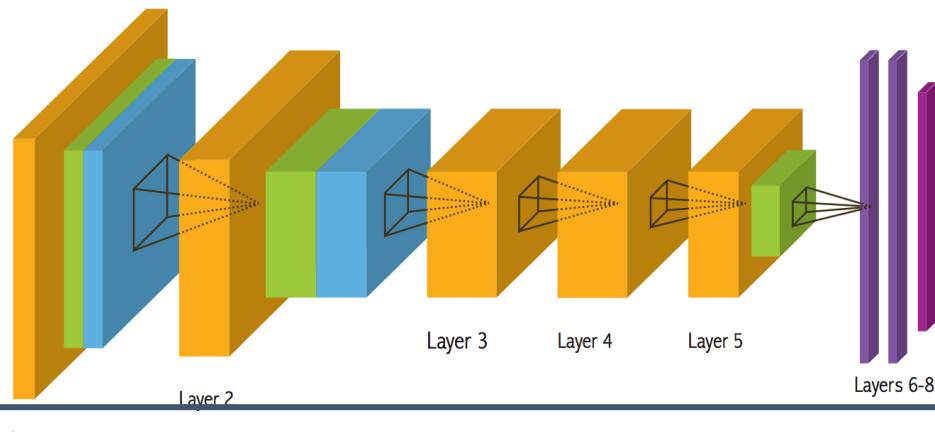
Image Features vs Neural Networks



f

**20 numbers giving
scores for classes**

training



Krizhevsky, Sutskever, and Hinton, "Imagenet classification with deep convolutional neural networks", NIPS 2012.

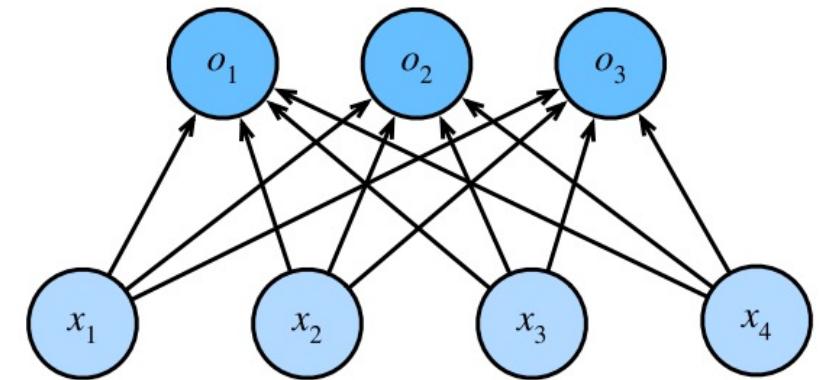
training

**20 numbers giving
scores for classes**

Neural Networks

Input: $x \in \mathbb{R}^D$ **Output:** $f(x) \in \mathbb{R}^C$

Before: Linear Classifier: $f(x) = Wx + b$
Learnable parameters: $W \in \mathbb{R}^{D \times C}, b \in \mathbb{R}^C$

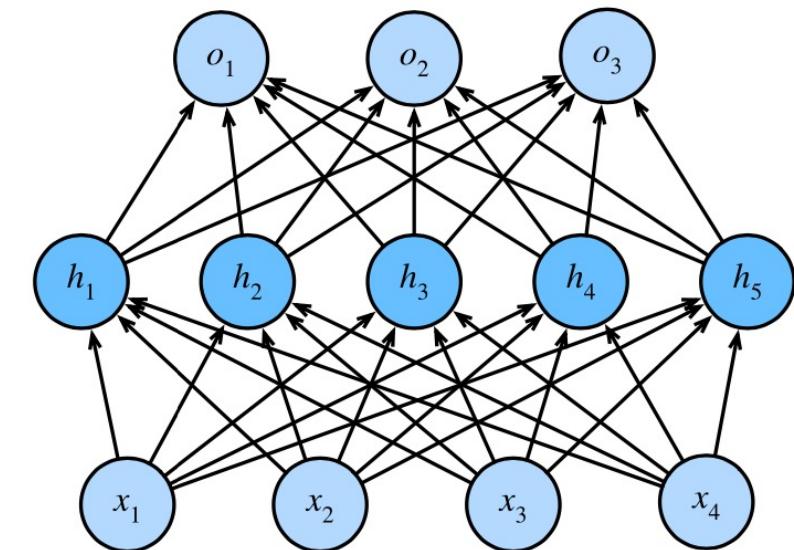


Neural Networks

Input: $x \in \mathbb{R}^D$ **Output:** $f(x) \in \mathbb{R}^C$

Before: Linear Classifier: $f(x) = Wx + b$

Learnable parameters: $W \in \mathbb{R}^{D \times C}, b \in \mathbb{R}^C$



Now: Two-Layer Neural Network: $f(x) = W_2 \max(0, W_1 x + b_1) + b_2$

Neural Networks

Before: Linear classifier

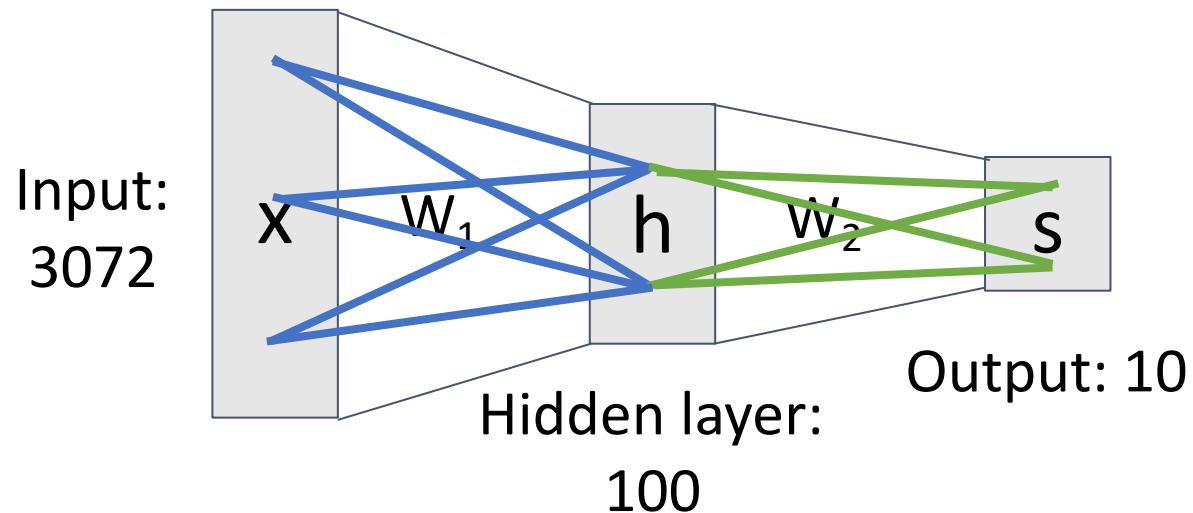
$$f(x) = Wx + b$$

Now: 2-layer Neural Network

$$f(x) = W_2 \max(0, W_1 x + b_1) + b_2$$

Element (i, j) of W_1
gives the effect on
 h_i from x_j

All elements
of x affect all
elements of h



Element (i, j) of W_2
gives the effect on
 s_i from h_j

All elements
of h affect all
elements of s

Fully-connected neural network
Also “Multi-Layer Perceptron” (MLP)

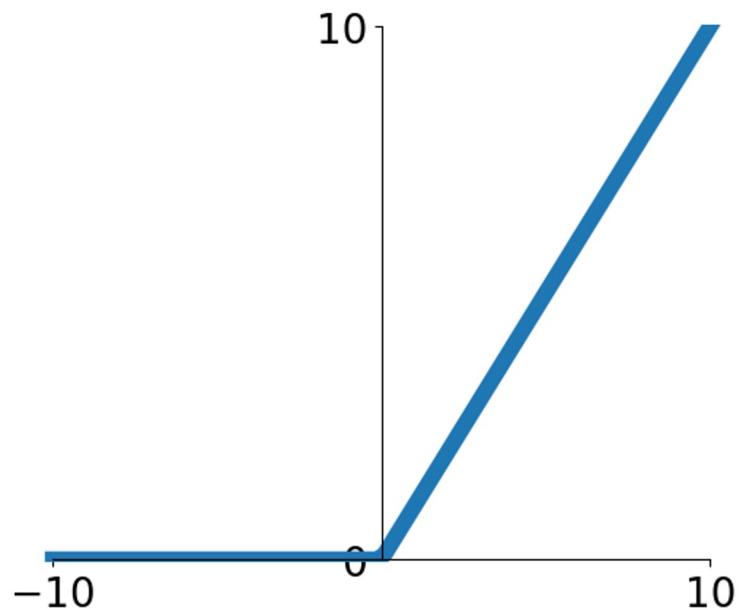
Activation Functions

2-layer Neural Network

$$f(x) = W_2 \max(0, W_1 x + b_1) + b_2$$

The function $ReLU(z) = \max(0, z)$
is called “Rectified Linear Unit”

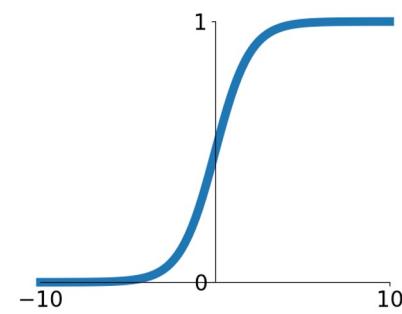
This is called the **activation function** of
the neural network



Activation Functions

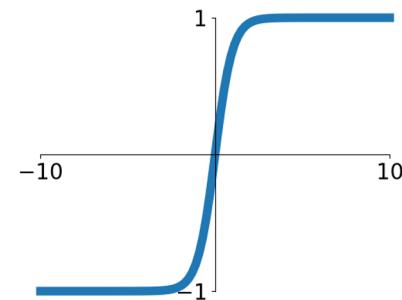
Sigmoid

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



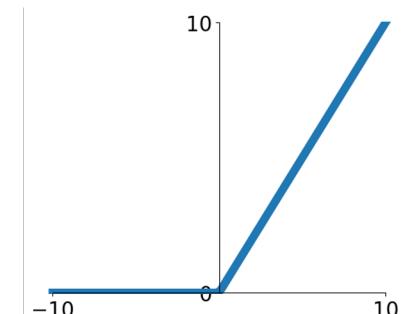
tanh

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$



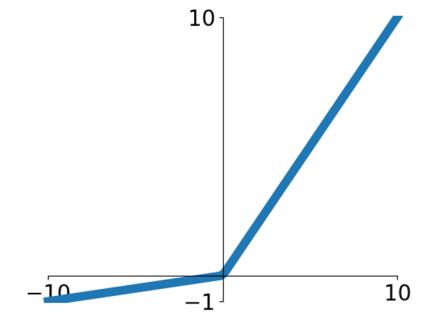
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.2x, x)$$

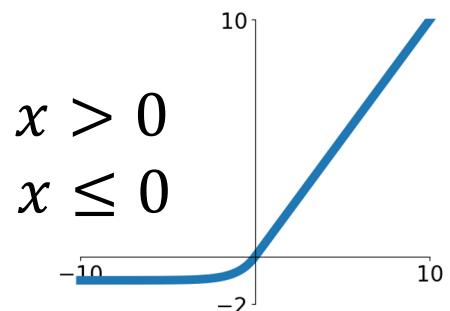


Softplus

$$\log(1 + \exp(x))$$

ELU

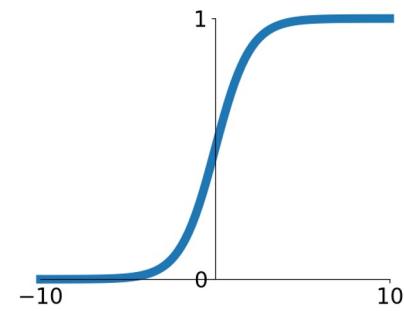
$$f(x) = \begin{cases} x, & x > 0 \\ \alpha(\exp(x) - 1), & x \leq 0 \end{cases}$$



Activation Functions

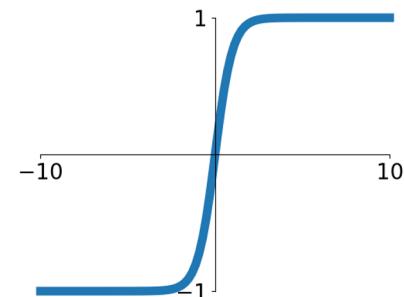
Sigmoid

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



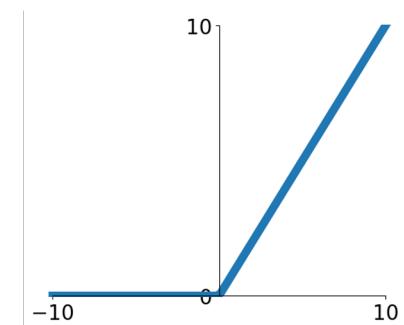
tanh

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$



ReLU

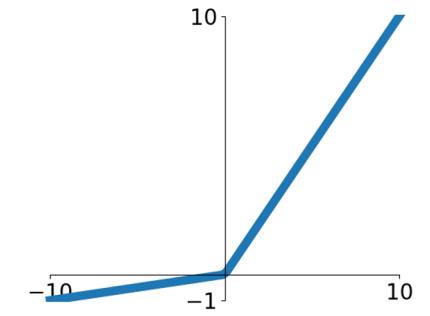
$$\max(0, x)$$



ReLU is a good default choice
for most problems

Leaky ReLU

$$\max(0.2x, x)$$

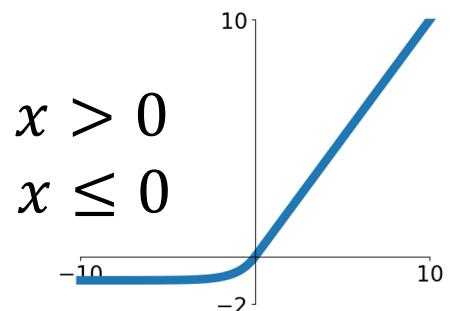


Softplus

$$\log(1 + \exp(x))$$

ELU

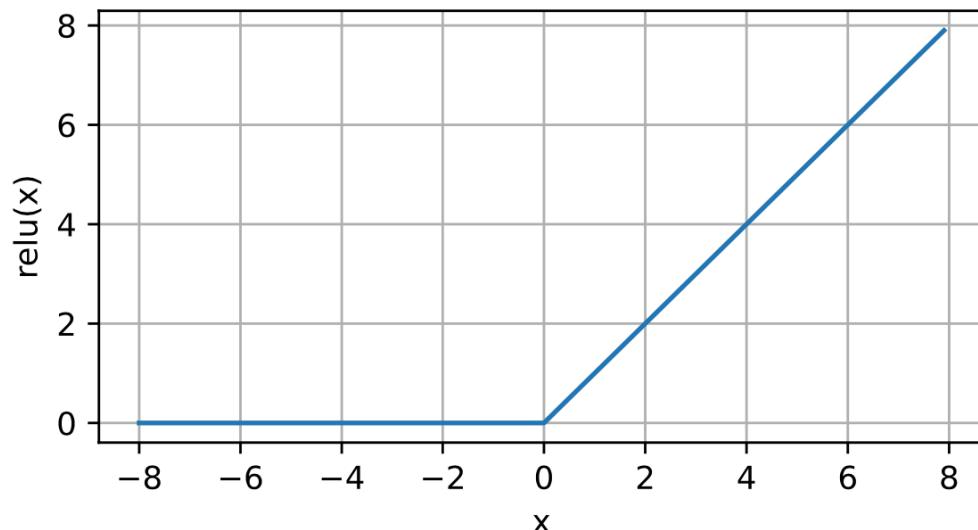
$$f(x) = \begin{cases} x, & x > 0 \\ \alpha(\exp(x) - 1), & x \leq 0 \end{cases}$$



Why ReLU

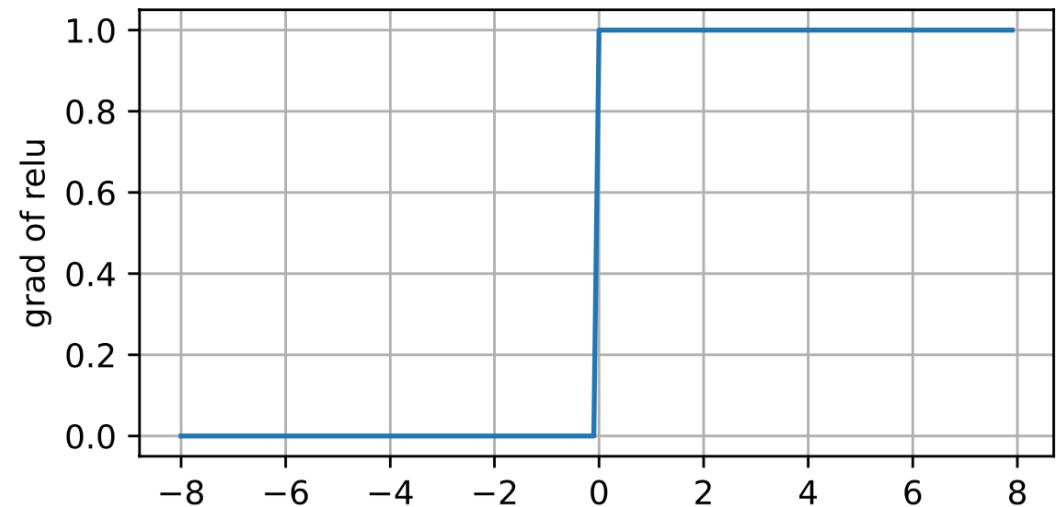
Forward pass

```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



Backward pass

```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



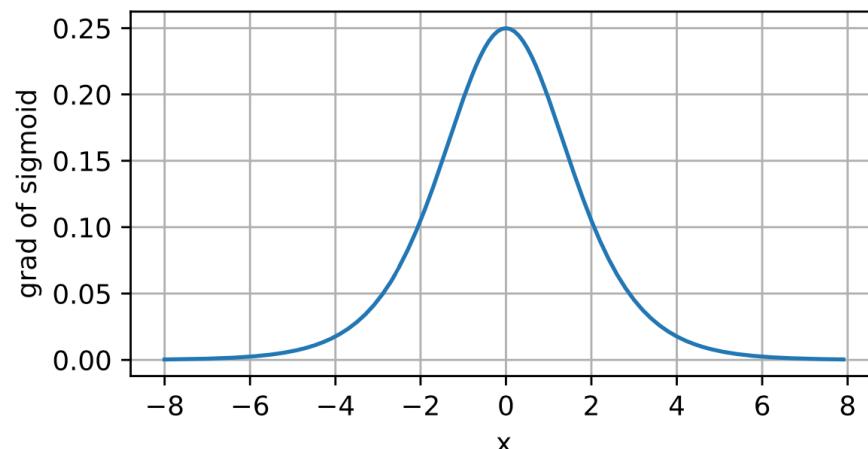
its derivatives are particularly well behaved: either they vanish or they just let the argument through. This makes optimization better behaved and it mitigated the well-documented problem of vanishing gradients

Issues with sigmoid and tanh

sigmoid is also called squashing function, turn $(-\infty, \infty)$ into $(0,1)$

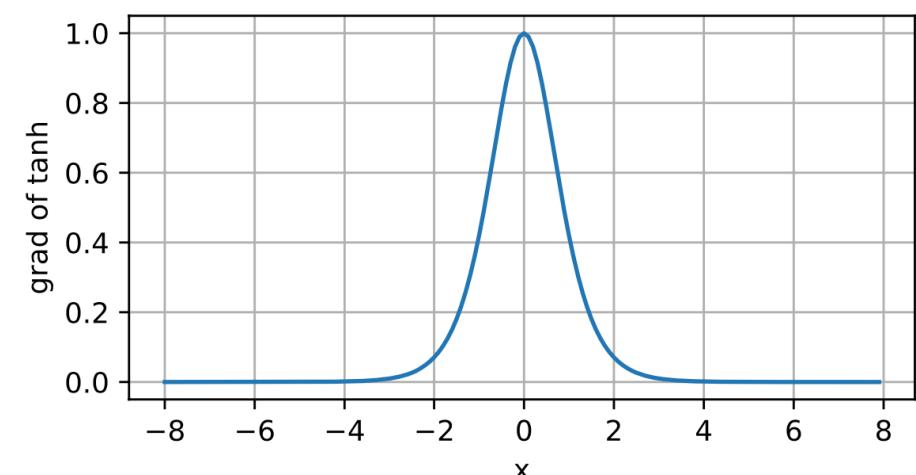
$$\frac{d}{dx} \text{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \text{sigmoid}(x)(1 - \text{sigmoid}(x)).$$

```
y = torch.sigmoid(x)
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



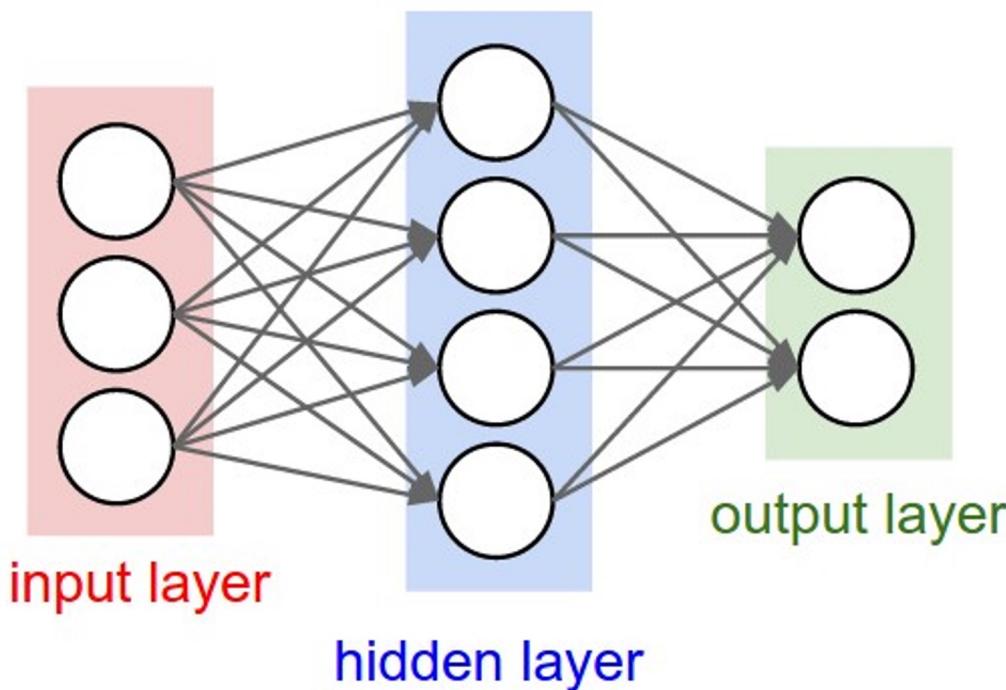
$$\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x).$$

```
y = torch.tanh(x)
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```



As the input diverges from 0 in either direction, the derivative approaches 0, vanishing gradient happens.

Neural Net in <20 lines!



```
num_inputs, num_outputs, num_hiddens = 784, 10, 256
```

```
W1 = nn.Parameter(torch.randn(  
    num_inputs, num_hiddens, requires_grad=True) * 0.01)  
b1 = nn.Parameter(torch.zeros(num_hiddens, requires_grad=True))  
W2 = nn.Parameter(torch.randn(  
    num_hiddens, num_outputs, requires_grad=True) * 0.01)  
b2 = nn.Parameter(torch.zeros(num_outputs, requires_grad=True))
```

```
params = [W1, b1, W2, b2]
```

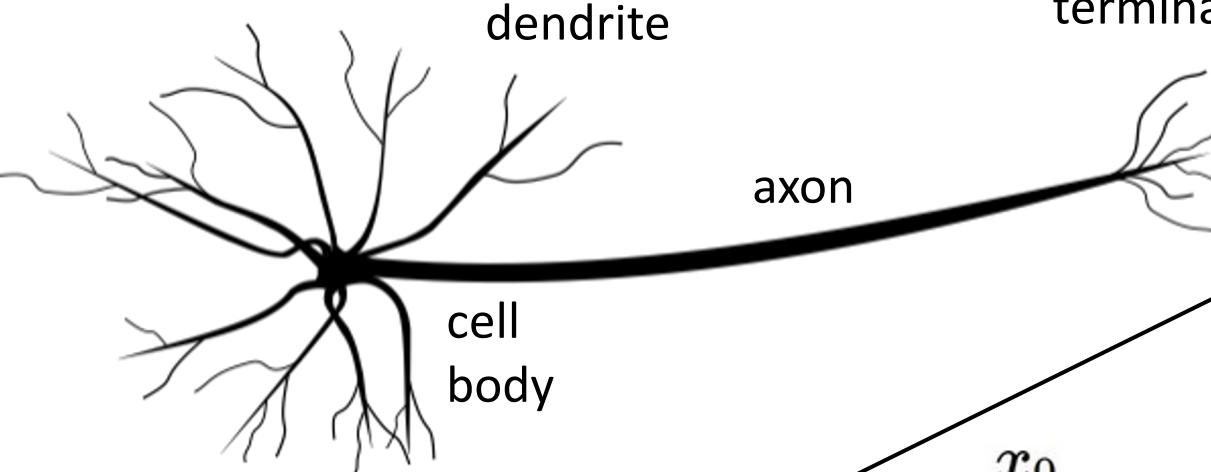
```
def relu(X):  
    a = torch.zeros_like(X)  
    return torch.max(X, a)
```

```
def net(X):  
    X = X.reshape((-1, num_inputs))  
    H = relu(X@W1 + b1) # Here '@' stands for matrix multiplication  
    return (H@W2 + b2)
```

```
loss = nn.CrossEntropyLoss()
```

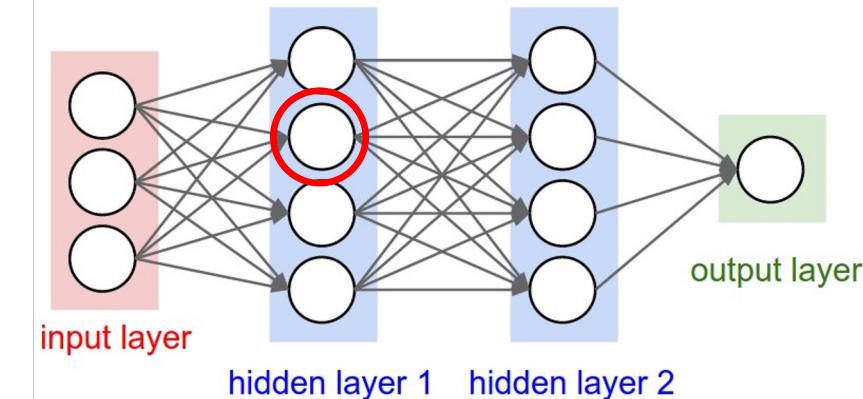
```
num_epochs, lr = 10, 0.1  
updater = torch.optim.SGD(params, lr=lr)  
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, updater)
```

Biological Neuron



presynaptic
terminal

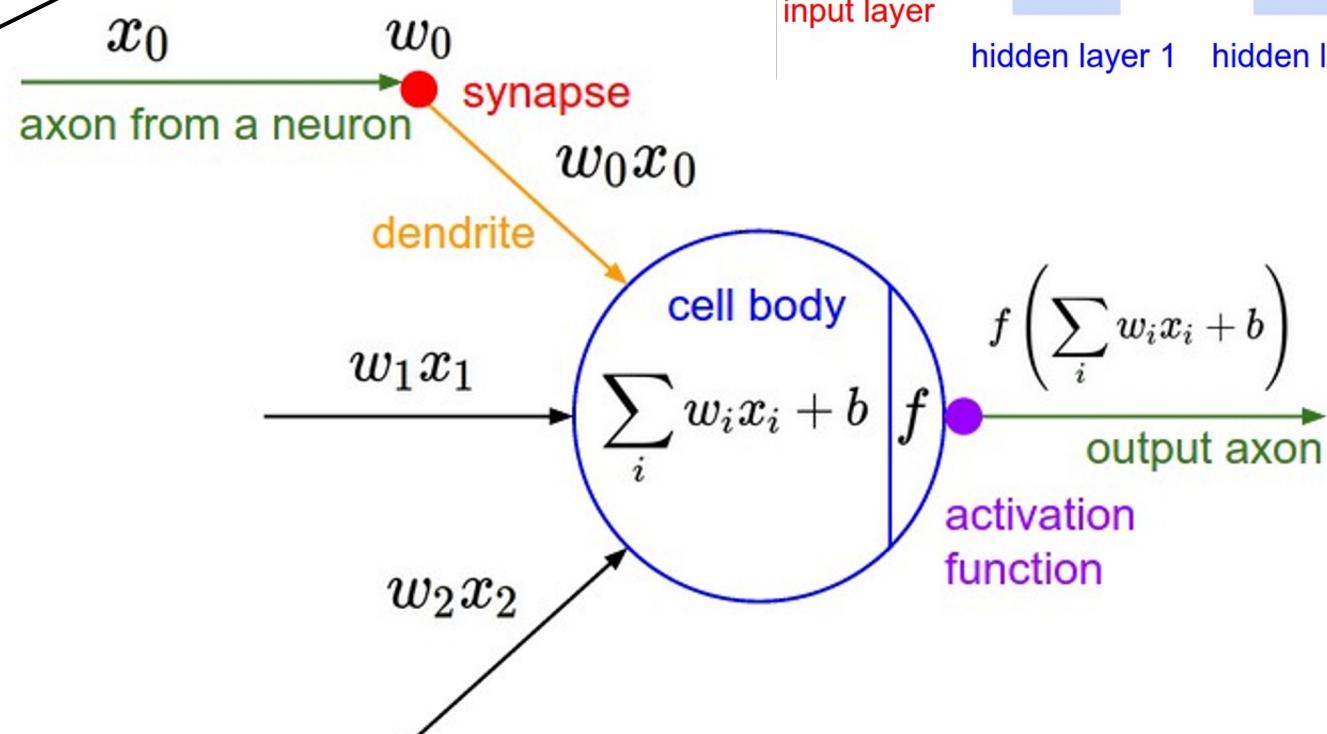
Artificial Neuron



input layer

hidden layer 1 hidden layer 2

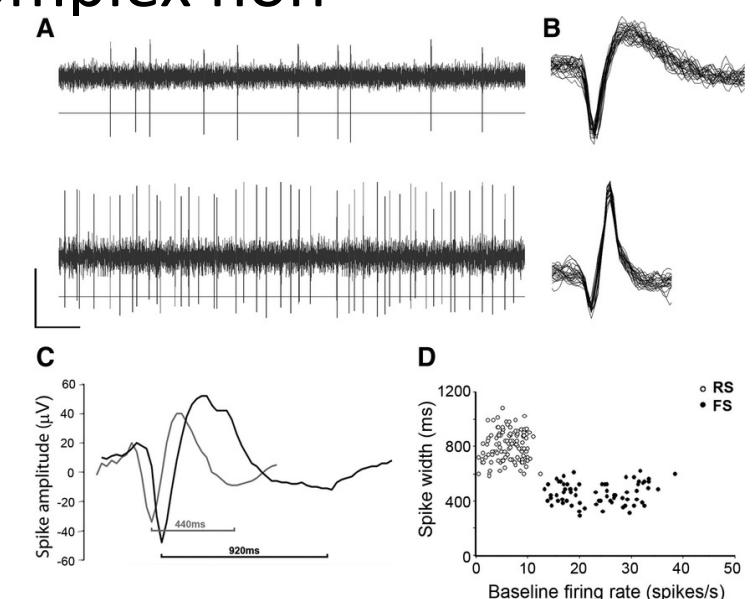
output layer



Are neural networks biologically inspired? Weakly

Biological Neurons:

- Many different types
- Dendrites can perform complex non-linear computations
- Synapses are not a single weight but a complex non-linear dynamical system
- Rate code may not be adequate

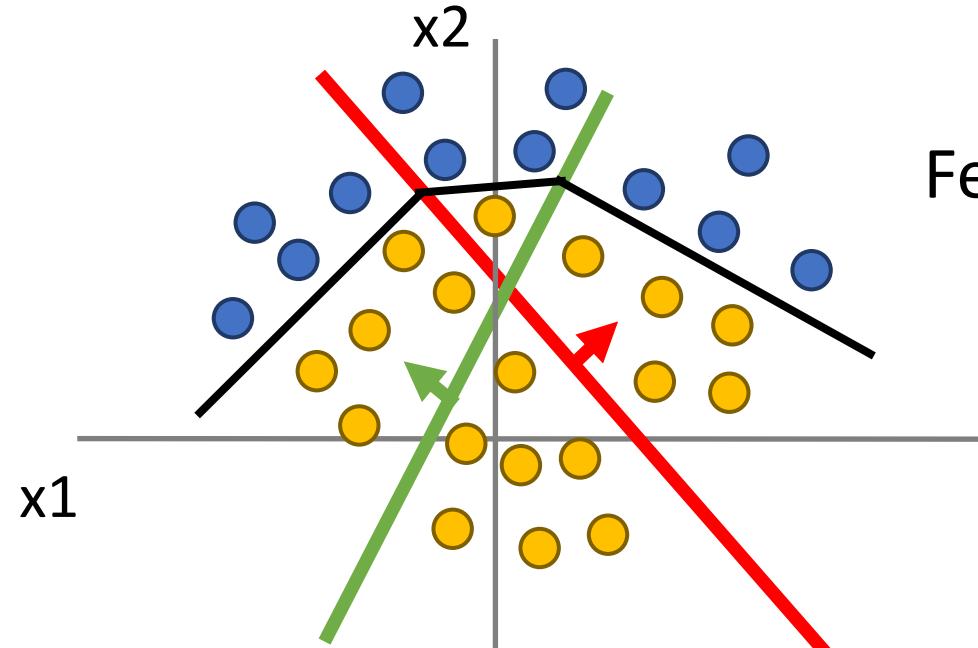


[Dendritic Computation. London and Häusser]

Differential Involvement of Excitatory and Inhibitory Neurons of Cat Motor Cortex in Coincident Spike Activity Related to Behavioral Context

Why ReLU? Space Warping

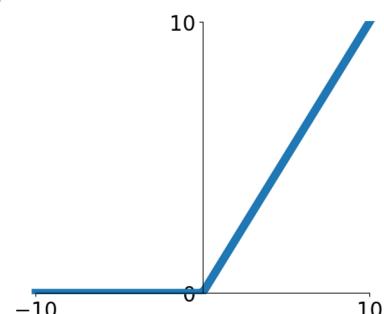
Points not linearly
separable in original space



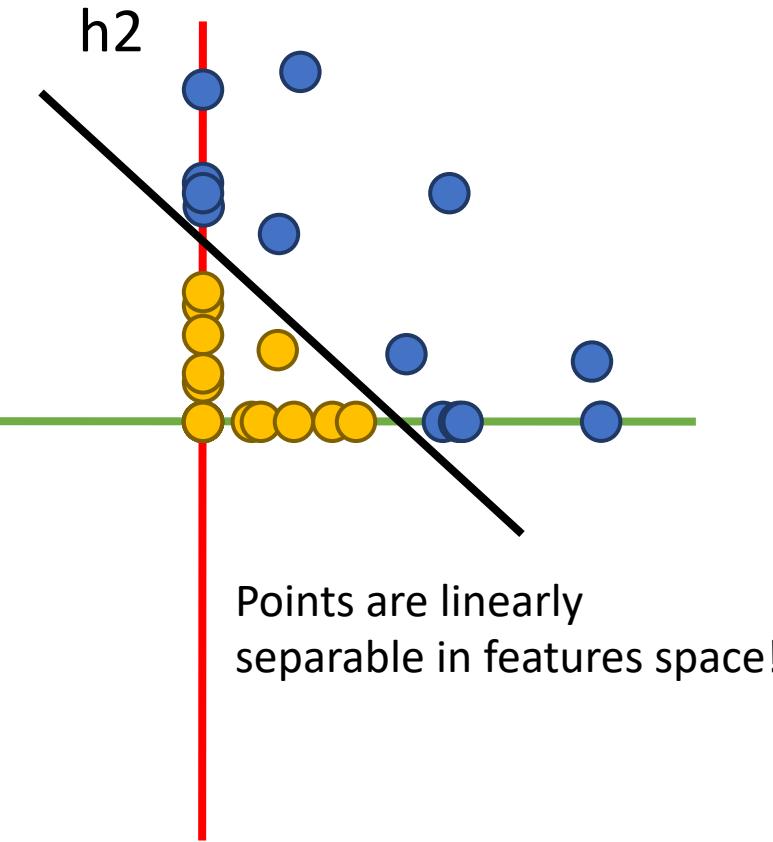
Linear classifier in feature
space gives nonlinear
classifier in original space

Feature transform:

$$h = \text{ReLU}(Wx)$$



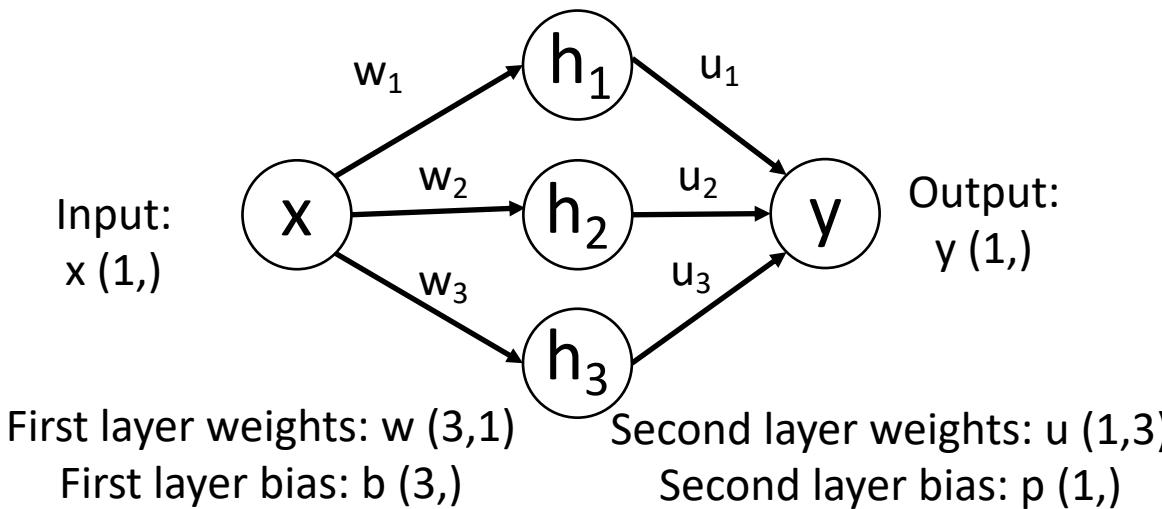
Consider a neural net hidden layer:
 $h = \text{ReLU}(Wx) = \max(0, Wx)$
Where x, h are both 2-dimensional



Points are linearly
separable in features space!

Universal Approximation

Example: Approximating a function $f: \mathbb{R} \rightarrow \mathbb{R}$ with a two-layer ReLU network



$$h_1 = \max(0, w_1 * x + b_1)$$

$$h_2 = \max(0, w_2 * x + b_2)$$

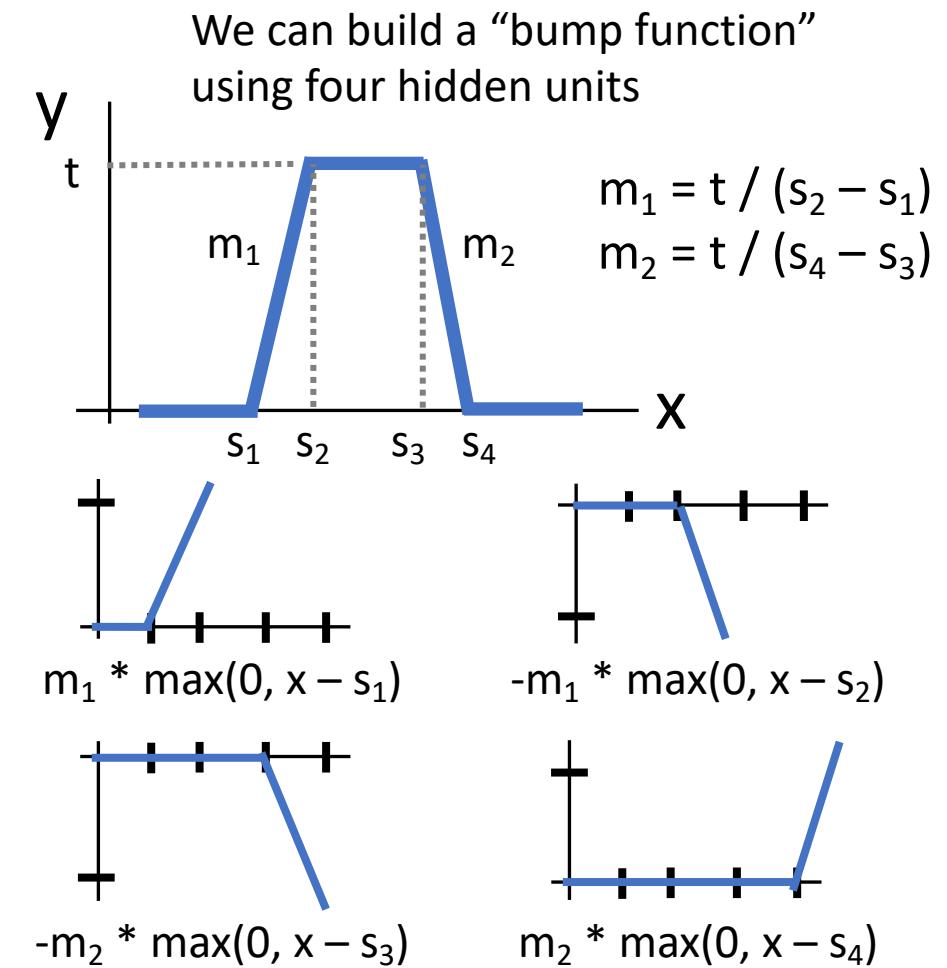
$$h_3 = \max(0, w_3 * x + b_3)$$

$$y = u_1 h_1 + u_2 * h_2 + u_3 * h_3 + p$$

$$y = u_1 * \max(0, w_1 * x + b_1)$$

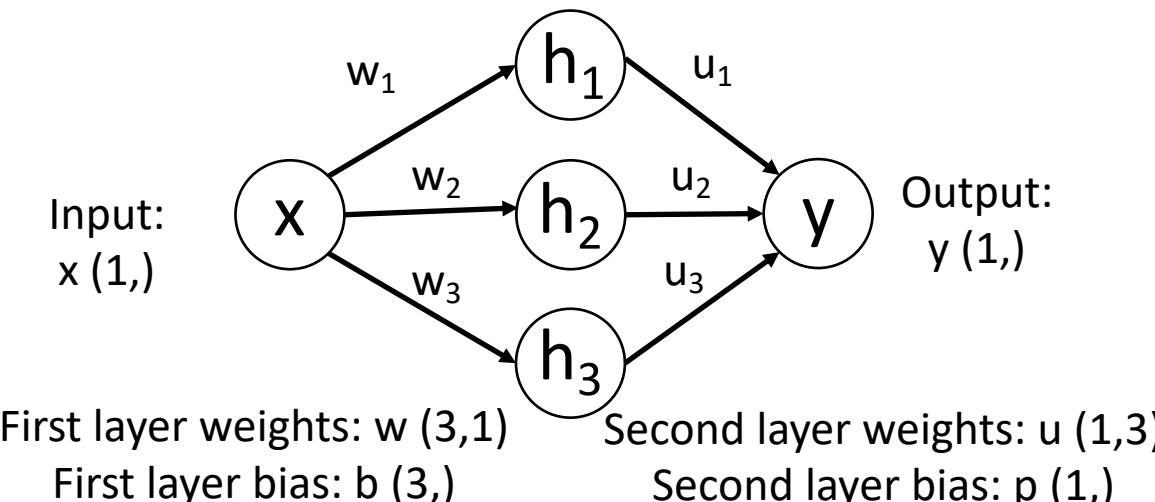
$$+ u_2 * \max(0, w_2 * x + b_2)$$

$$+ u_3 * \max(0, w_3 * x + b_3) + p$$



Universal Approximation

Example: Approximating a function $f: \mathbb{R} \rightarrow \mathbb{R}$ with a two-layer ReLU network



$$h_1 = \max(0, w_1 * x + b_1)$$

$$h_2 = \max(0, w_2 * x + b_2)$$

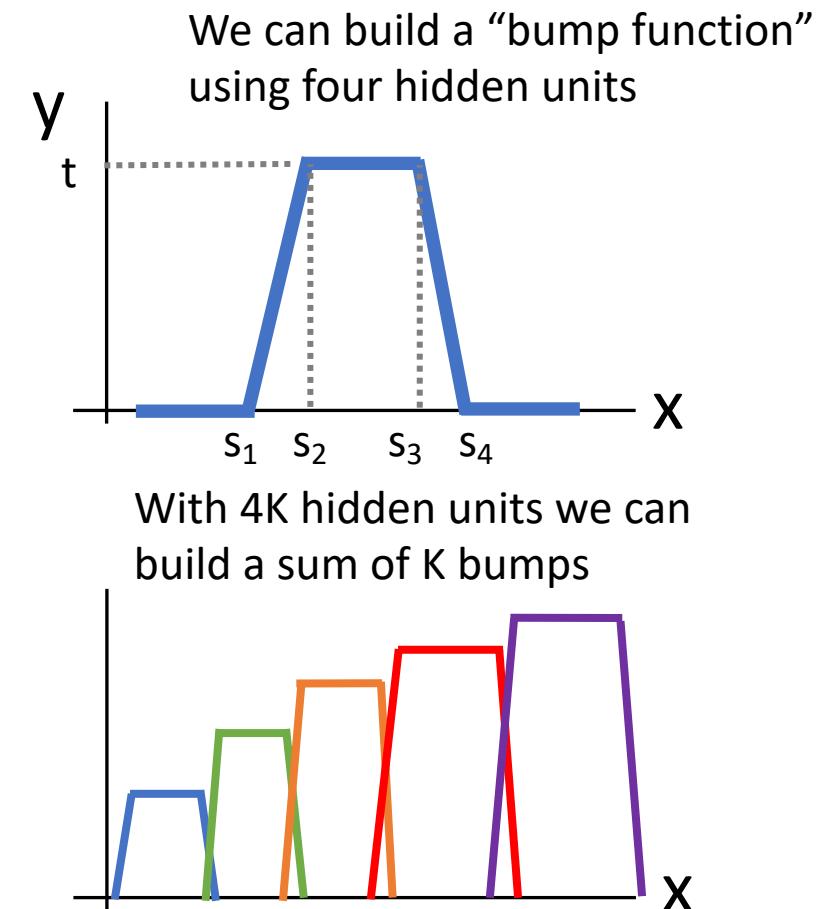
$$h_3 = \max(0, w_3 * x + b_3)$$

$$y = u_1 h_1 + u_2 * h_2 + u_3 * h_3 + p$$

$$y = u_1 * \max(0, w_1 * x + b_1)$$

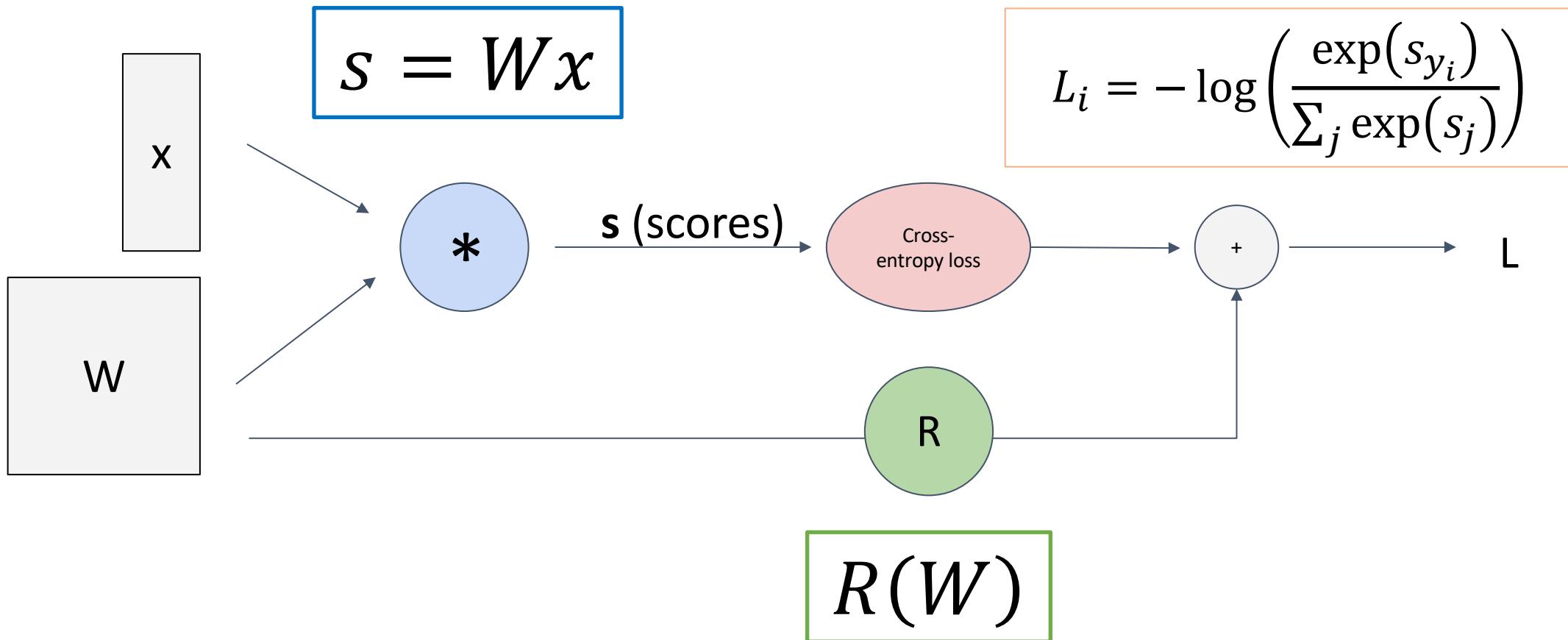
$$+ u_2 * \max(0, w_2 * x + b_2)$$

$$+ u_3 * \max(0, w_3 * x + b_3) + p$$



Lecture 6: Backpropagation

Computational Graphs



Simplified computational graph to denote neural networks

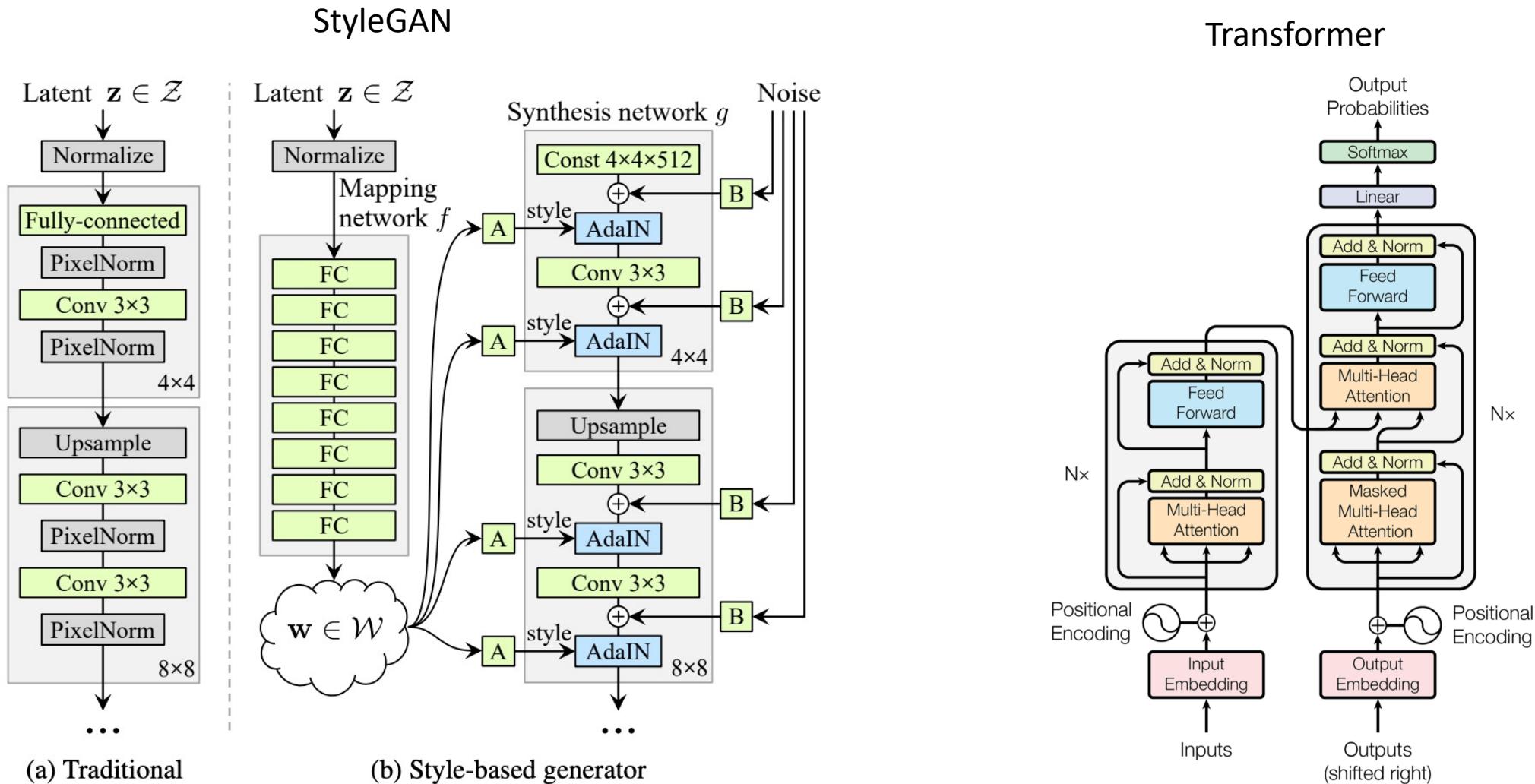


Figure 1: The Transformer - model architecture.

Chain Rule

Suppose that functions $y = f(u)$ and $u = g(x)$ are both differentiable, then the chain rule states that

$$\frac{dy}{dx} = \frac{dy}{du} \frac{du}{dx}$$

In more general scenario, suppose that y has variables u_1, u_2, \dots, u_m , where each differentiable function u_i has variables x_1, x_2, \dots, x_n . Then the chain rule gives

$$\frac{dy}{dx_i} = \frac{dy}{du_1} \frac{du_1}{dx_i} + \frac{dy}{du_2} \frac{du_2}{dx_i} + \dots + \frac{dy}{du_m} \frac{du_m}{dx_i}$$

for any $i = 1, 2, \dots, n$

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

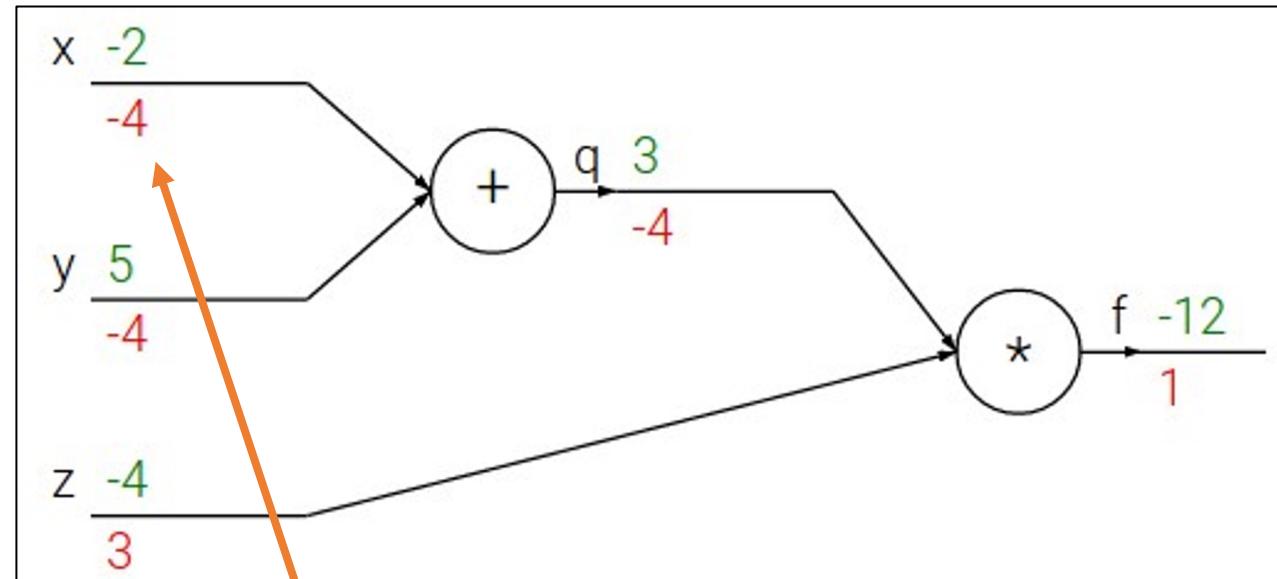
e.g. $x = -2$, $y = 5$, $z = -4$

1. Forward pass: Compute outputs

$$q = x + y \quad f = q \cdot z$$

2. Backward pass: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Chain Rule

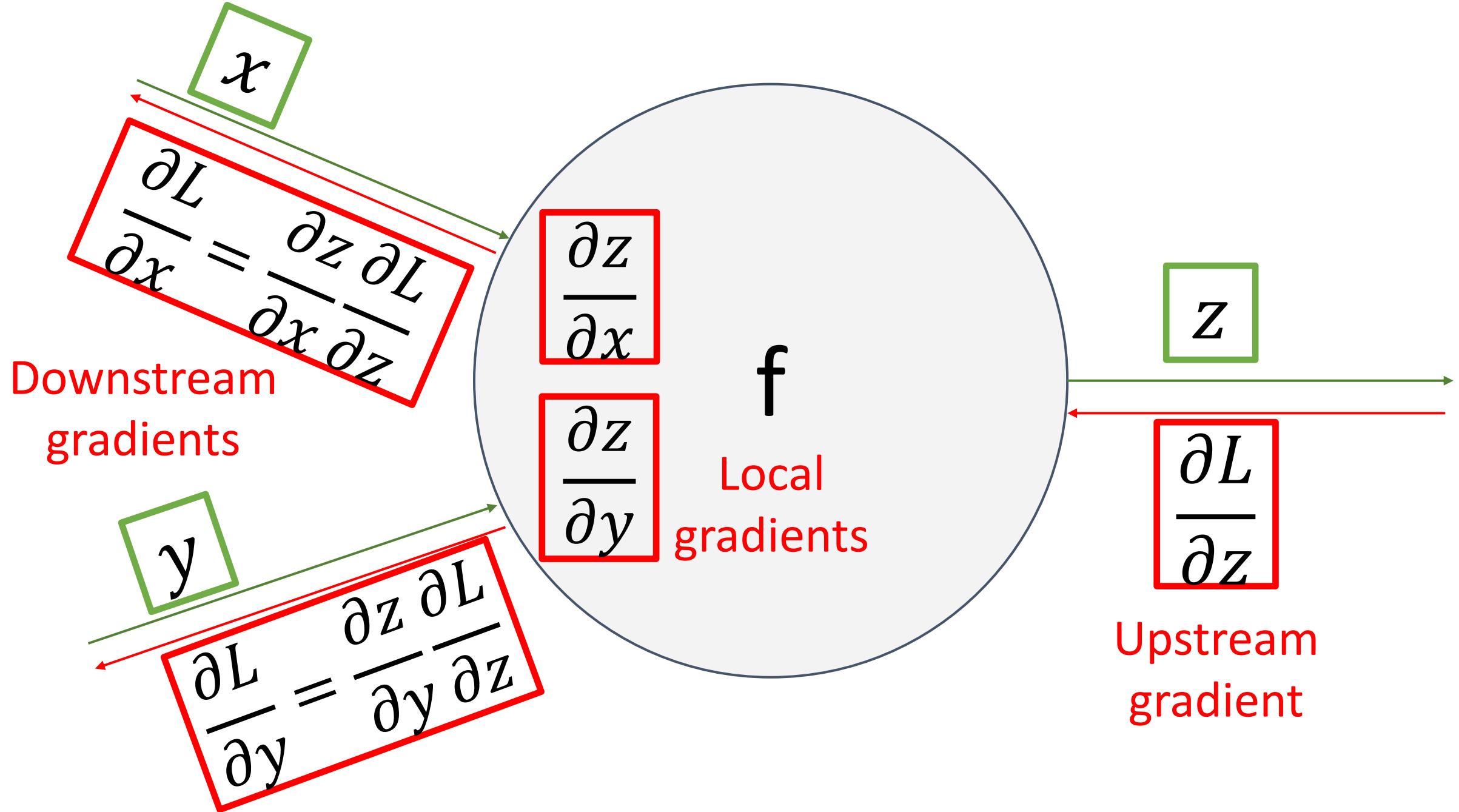
$$\frac{\partial f}{\partial x} = \frac{\partial q}{\partial x} \frac{\partial f}{\partial q}$$

Downstream
Gradient

Local
Gradient

Upstream
Gradient

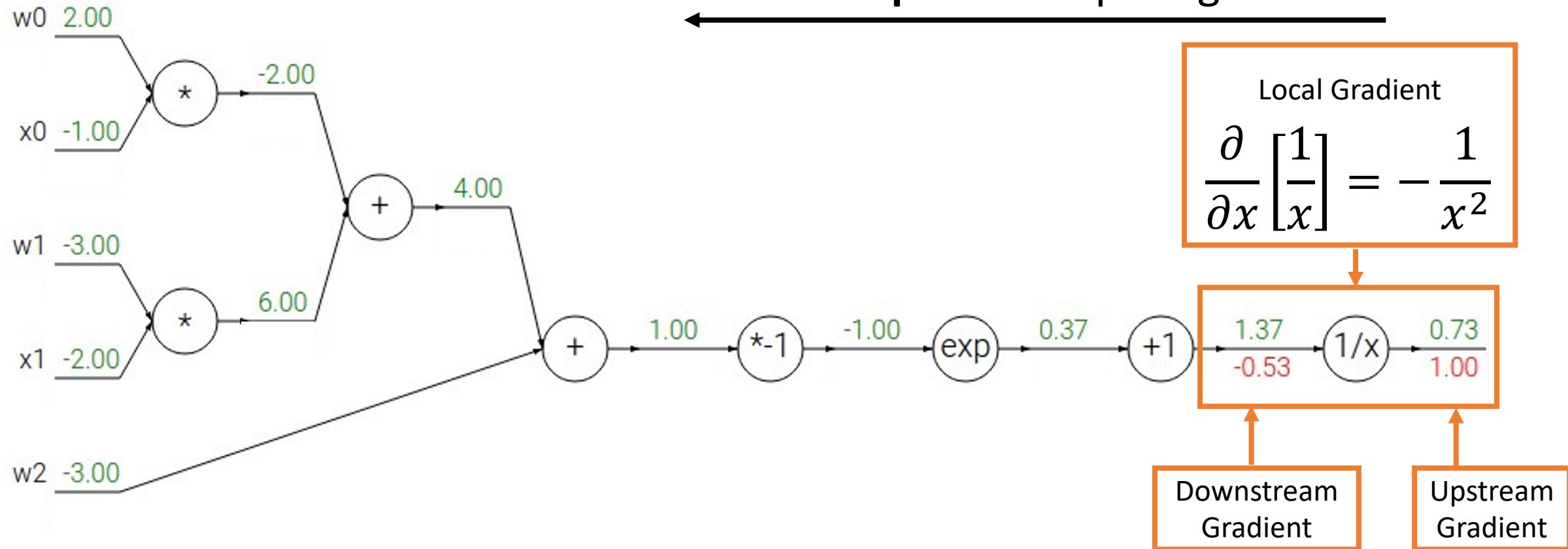
$$\frac{\partial q}{\partial x} = 1$$



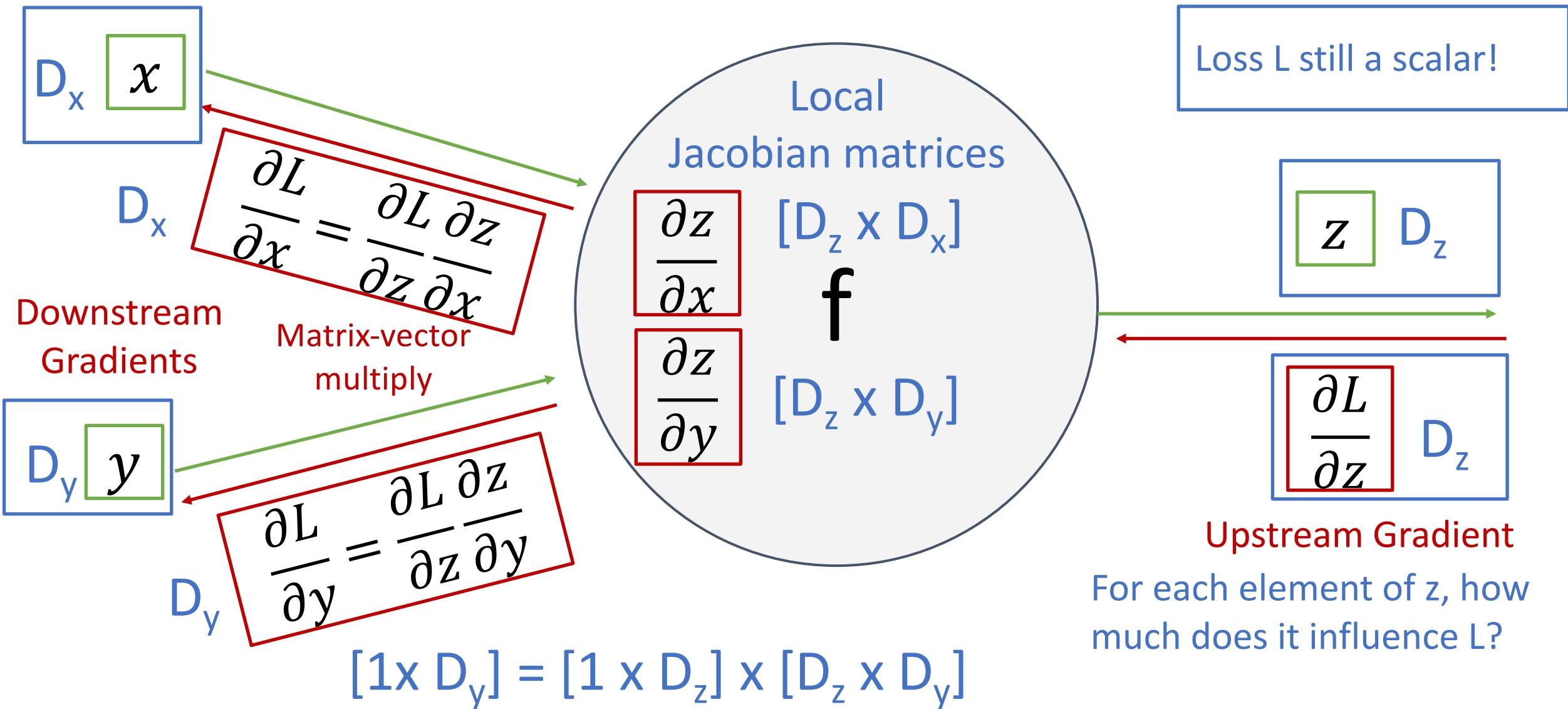
Another Example

$$f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

Backward pass: Compute gradients



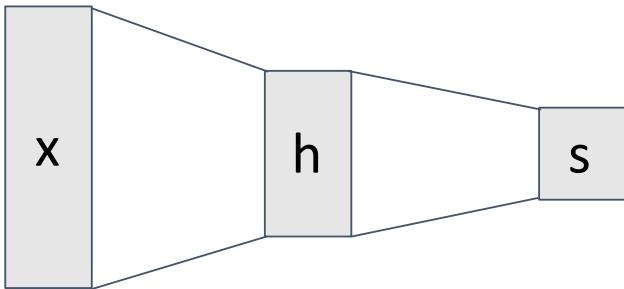
Backprop with Vectors



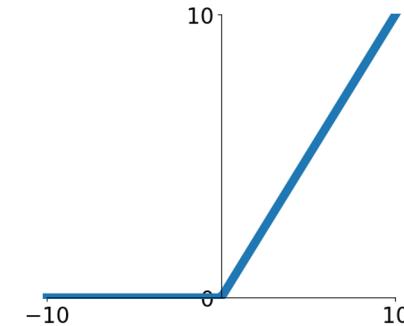
Lecture 7: Convolutional Neural Network (ConvNet)

Components of a Convolutional Network

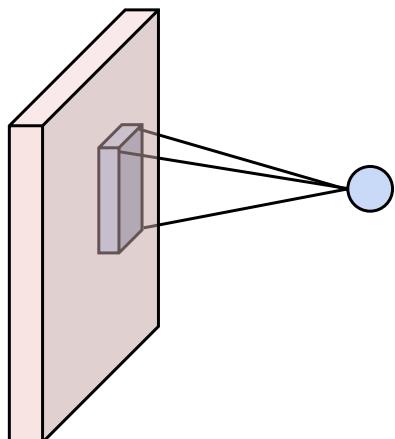
Fully-Connected Layers



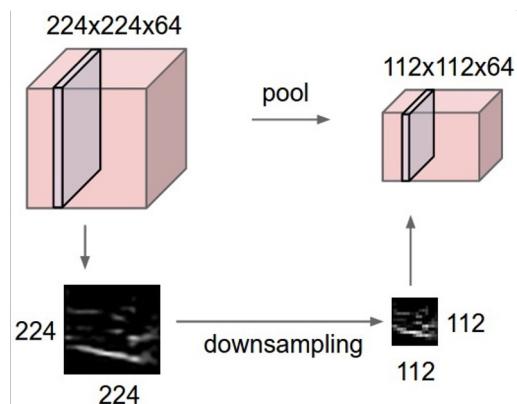
Activation Function



Convolution Layers



Pooling Layers

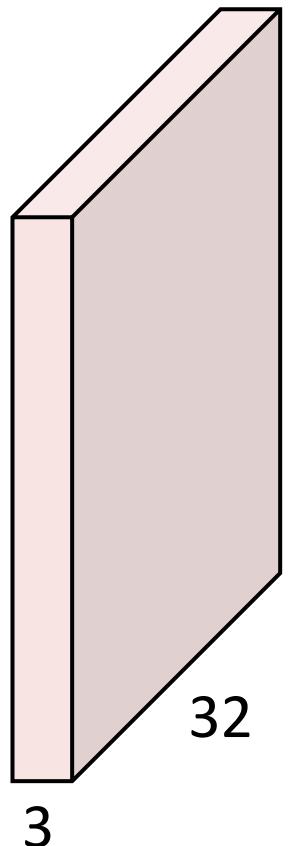


Normalization

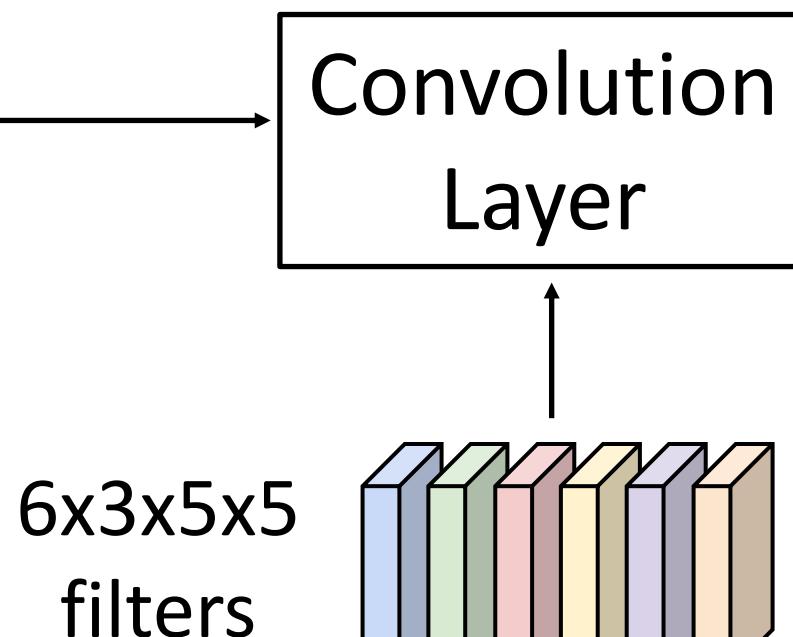
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Convolution Layer

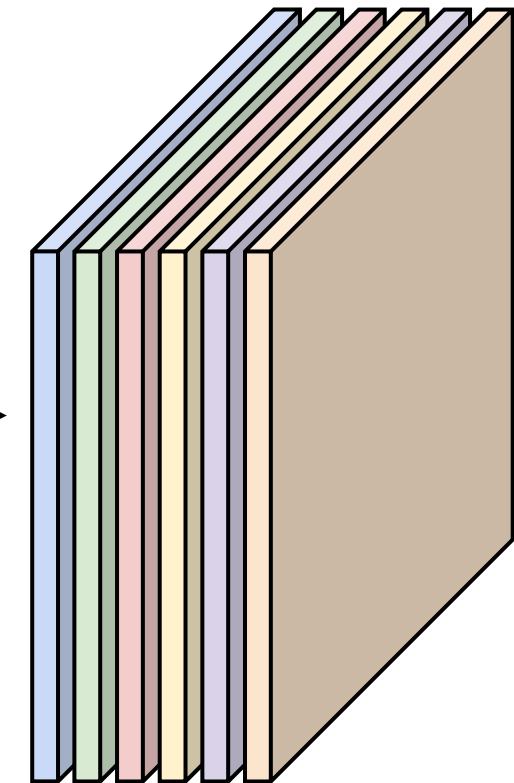
3x32x32 image



Consider 6 filters,
each 3x5x5

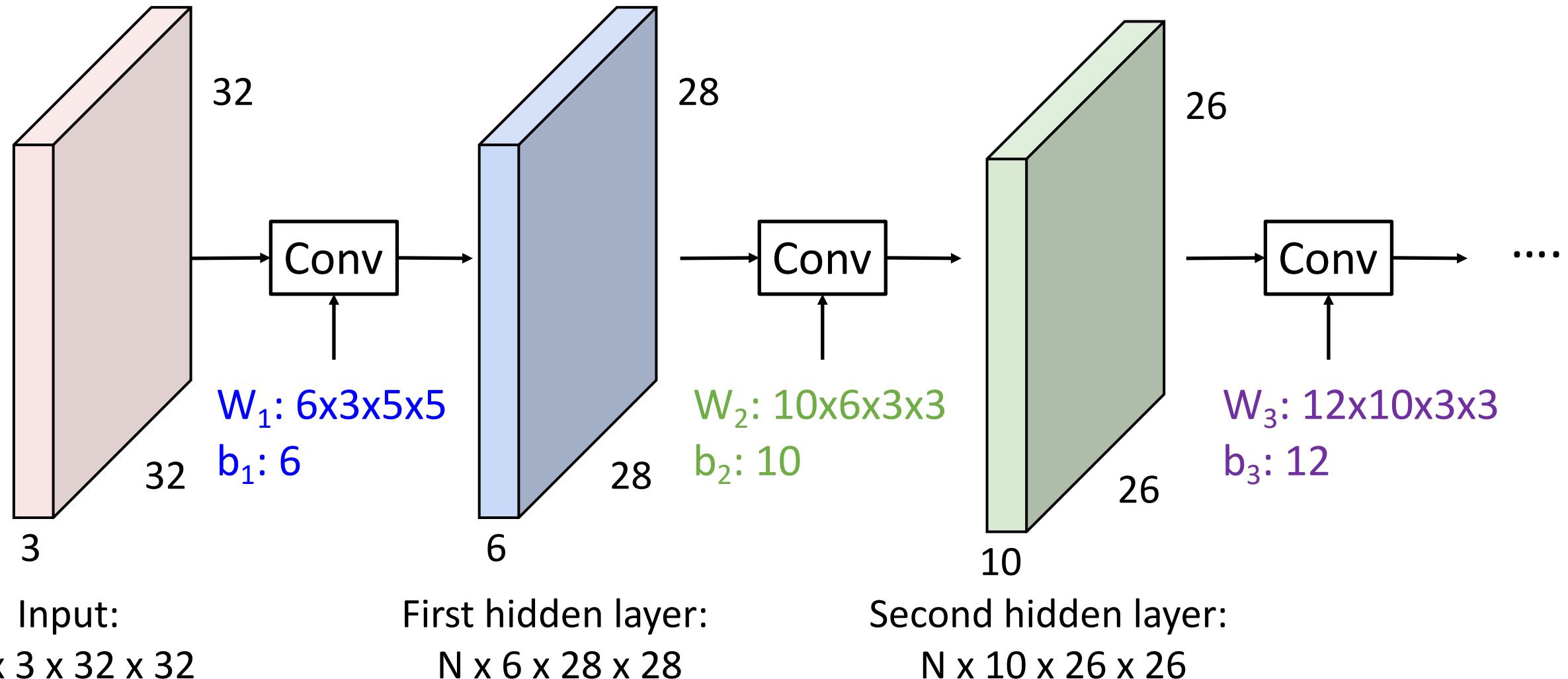


6 activation maps,
each 1x28x28



Stack activations to get a
6x28x28 output image!

Stacking Convolutions



A closer look at spatial dimensions

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Input: 7x7

Filter: 3x3

Output: 5x5

In general:

Input: W

Filter: K

Padding: P

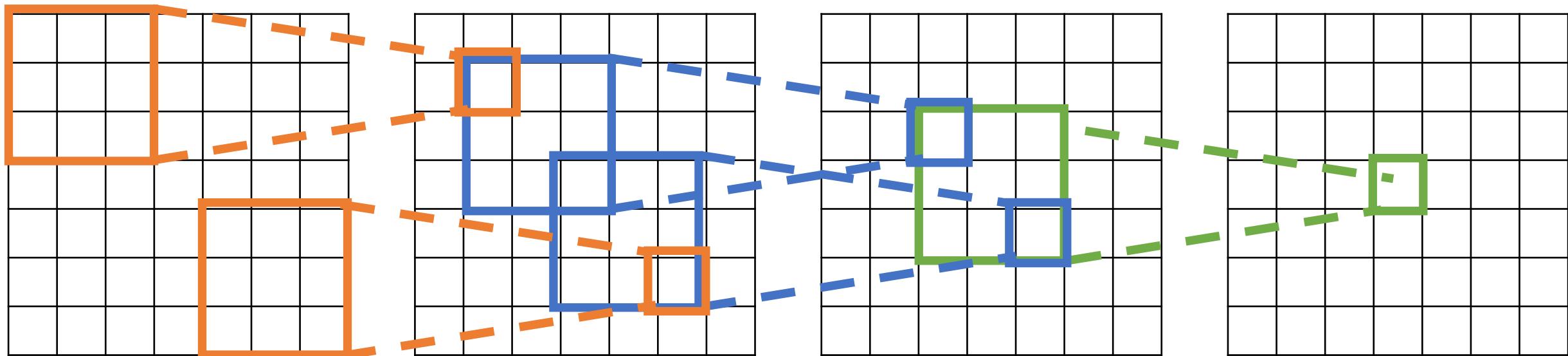
Output: $W - K + 1 + 2P$

Very common:

Set $P = (K - 1) / 2$ to
make output have
same size as input!

Receptive Fields

Each successive convolution adds $K - 1$ to the receptive field size
With L layers the receptive field size is $1 + L * (K - 1)$



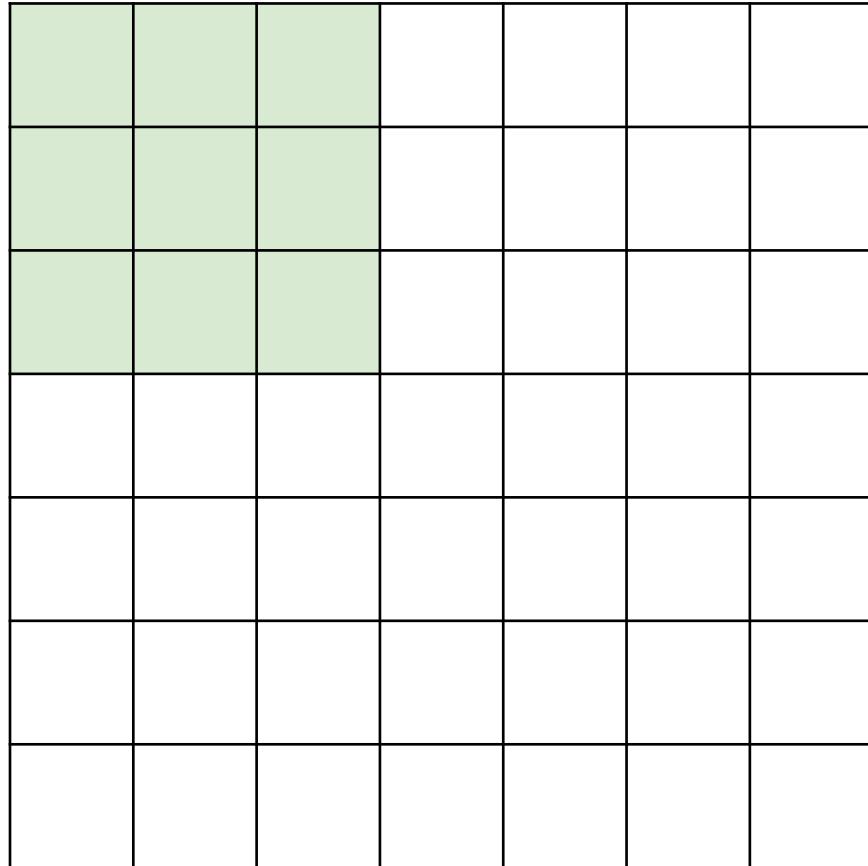
Input

Problem: For large images we need many layers
for each output to “see” the whole image image

Solution: Downsample inside the network

Output

Strided Convolution

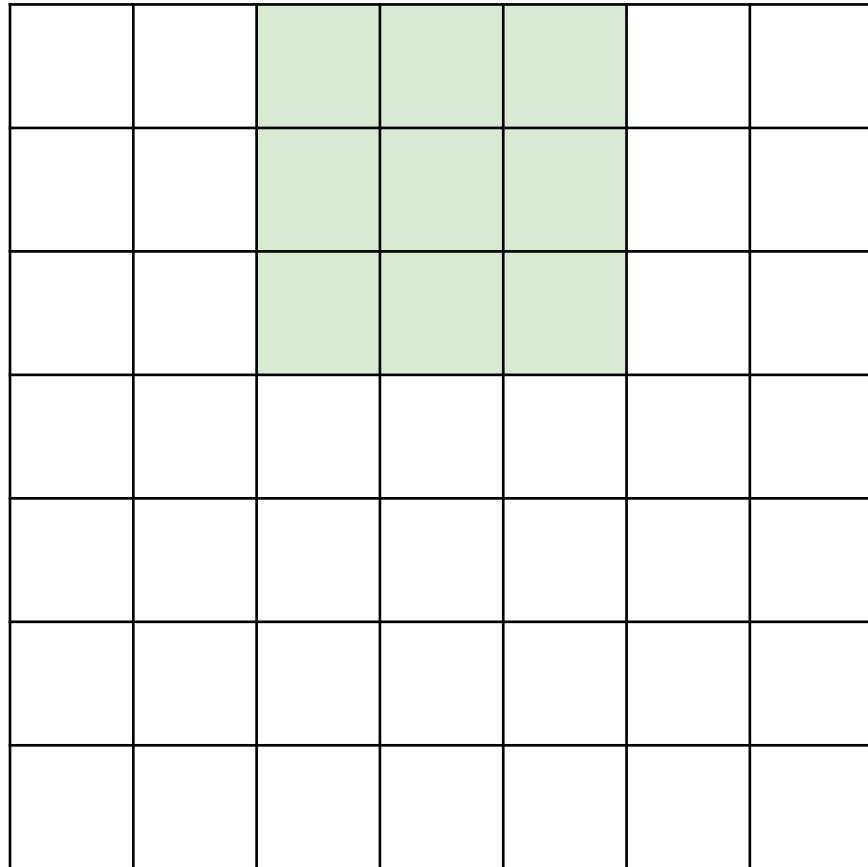


Input: 7x7

Filter: 3x3

Stride: 2

Strided Convolution

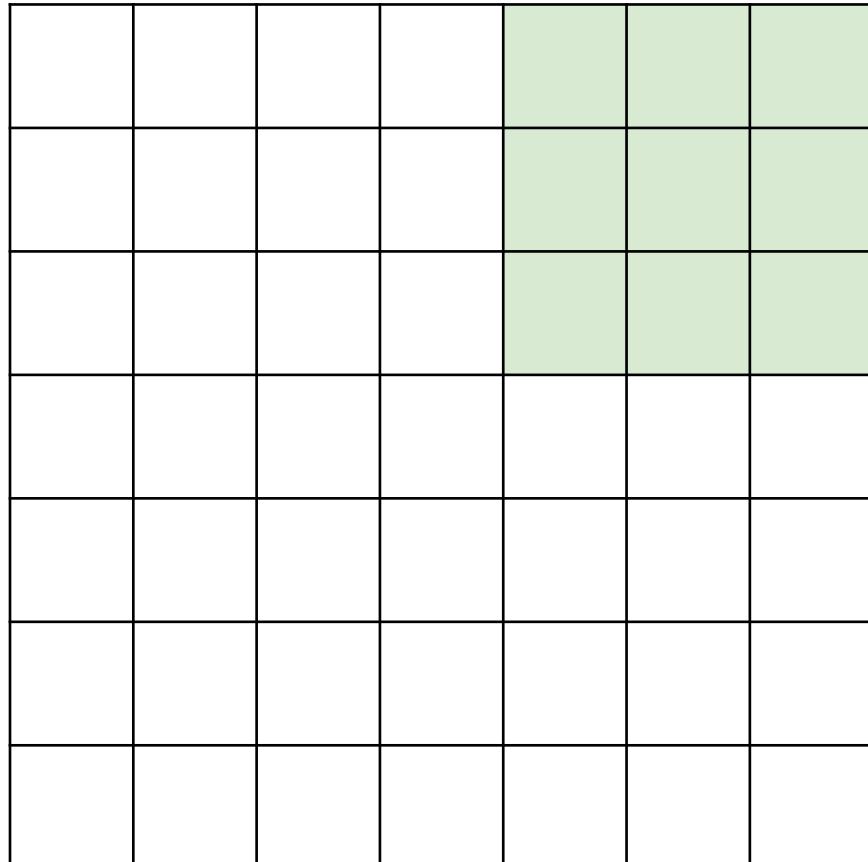


Input: 7x7

Filter: 3x3

Stride: 2

Strided Convolution



Input: 7x7

Filter: 3x3

Stride: 2

Output: 3x3

In general:

Input: W

Filter: K

Padding: P

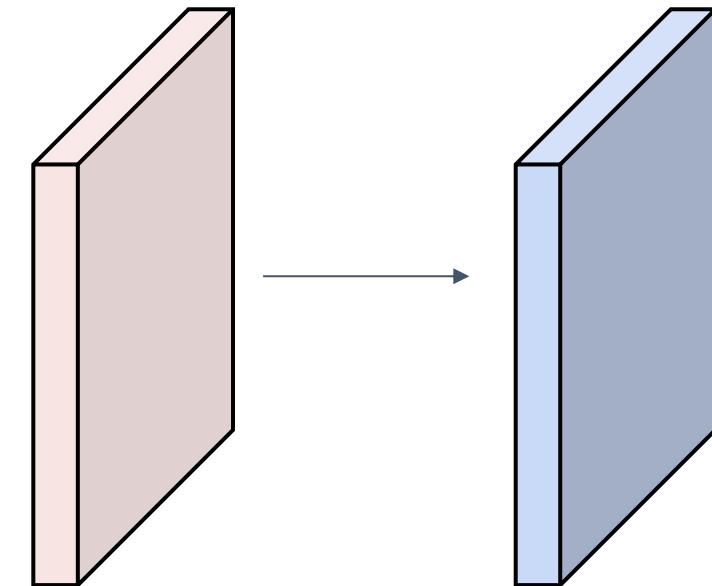
Stride: S

Output: $(W - K + 2P) / S + 1$

Convolution Example

Input volume: **3 x 32 x 32**

10 **5x5** filters with stride 1, pad 2



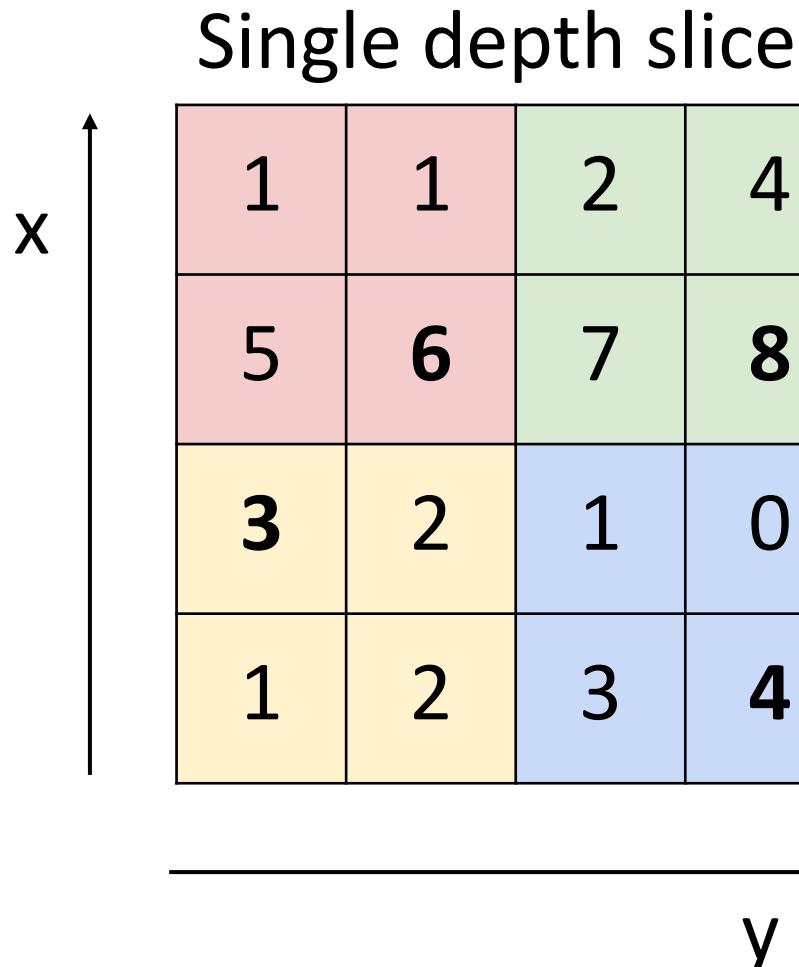
Output volume size: **10 x 32 x 32**

Number of learnable parameters: 760

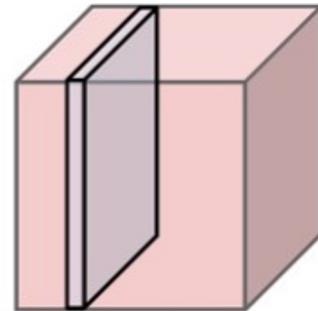
Number of multiply-add operations: **768,000**

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

Max Pooling



64 x 224 x 224



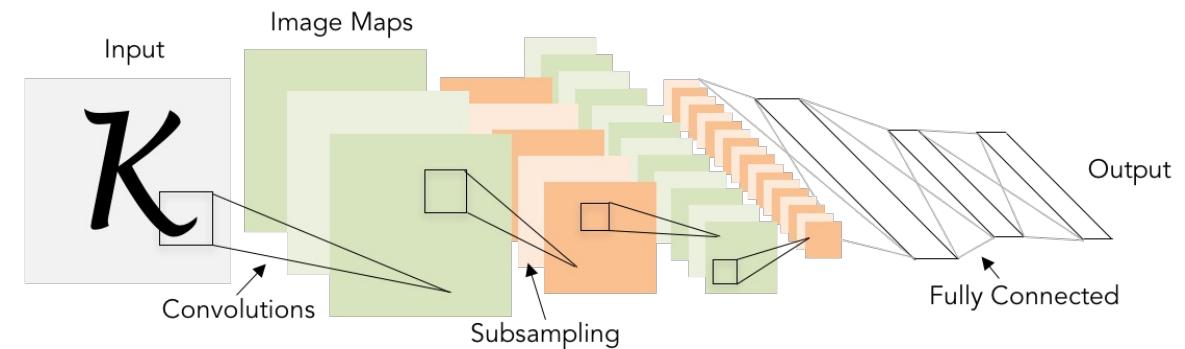
Max pooling with 2x2
kernel size and stride 2

6	8
3	4

Introduces **invariance** to
small spatial shifts
No learnable parameters!

Example: LeNet-5

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



As we go through the network:

Spatial size **decreases**
(using pooling or strided conv)

Number of channels **increases**
(total “volume” is preserved!)

Batch Normalization

Idea: “Normalize” the outputs of a layer so they have zero mean and unit variance

Why? Helps reduce “internal covariate shift”, improves optimization

We can normalize a batch of activations like this:

$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x]}}$$

This is a **differentiable function**, so we can use it as an operator in our networks and backprop through it!

Example of Batchnorm in NN

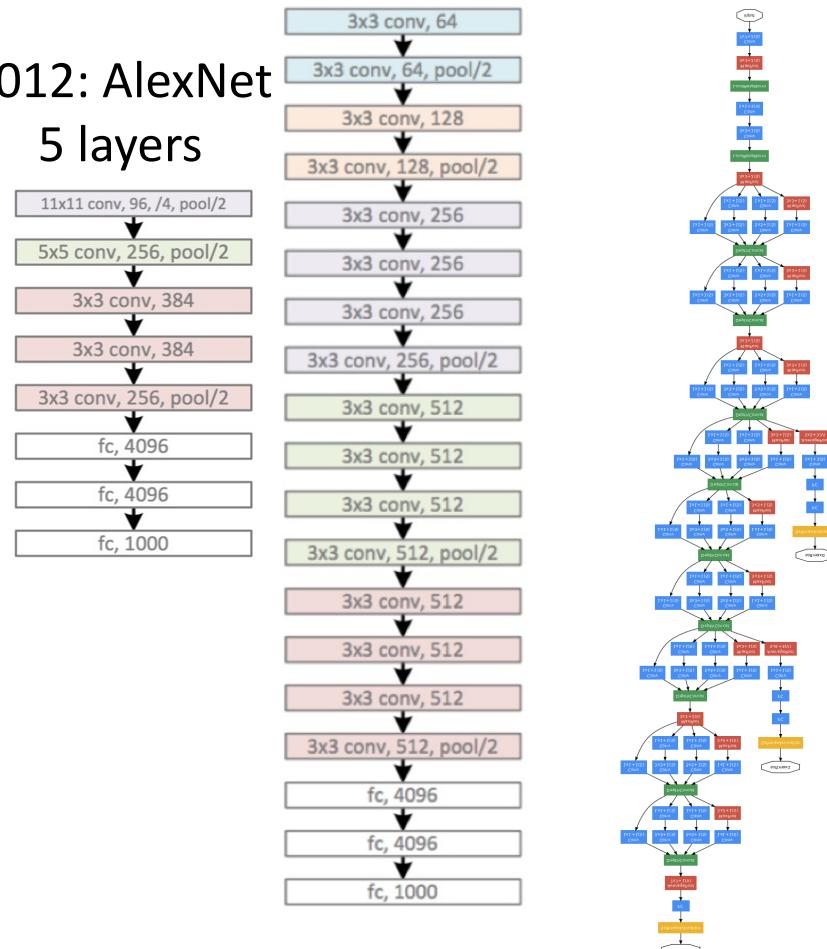
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1=nn.Conv2d(1,32,3,1)
        self.conv1_bn=nn.BatchNorm2d(32)
        self.conv2=nn.Conv2d(32,64,3,1)
        self.conv2_bn=nn.BatchNorm2d(64)
        self.dropout1=nn.Dropout(0.25)
        self.fc1=nn.Linear(9216,128)
        self.fc1_bn=nn.BatchNorm1d(128)
        self.fc2=nn.Linear(128,10)
    def forward(self,x):
        x=self.conv1(x)
        x=F.relu(self.conv1_bn(x))
        x=self.conv2(x)
        x=F.relu(self.conv2_bn(x))
        x=F.max_pool2d(x,2)
        x=self.dropout1(x)
        x=torch.flatten(x,1)
        x=self.fc1(x)
        x=F.relu(self.fc1_bn(x))
        x=self.fc2(x)
        output=F.log_softmax(x,dim=1)
        return output
```

Lecture 8: Modern CNN Architectures

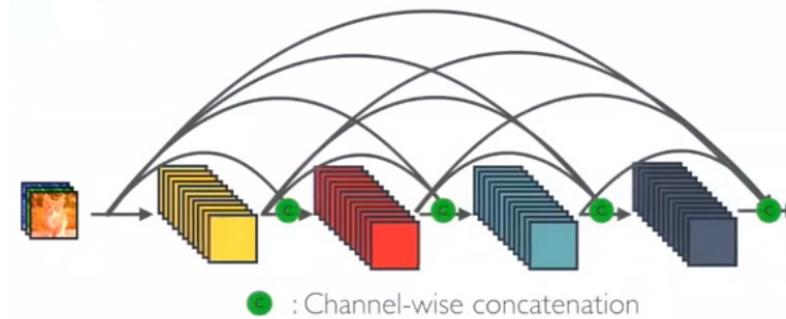
2016: ResNet
>100 layers

2014: VGG 2015: GoogLeNet
16 layers 22 layers

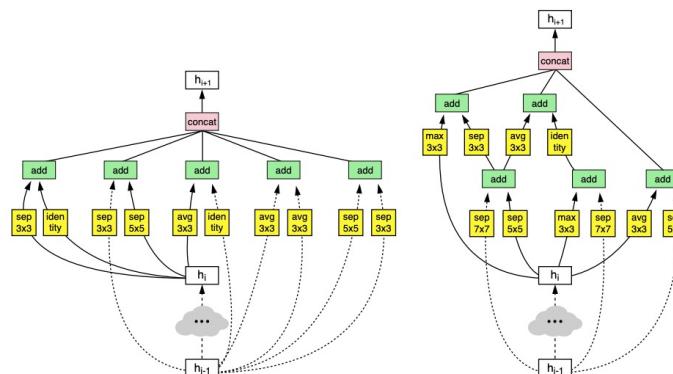
2012: AlexNet
5 layers



This Lecture: A Zoo of CNN Architectures



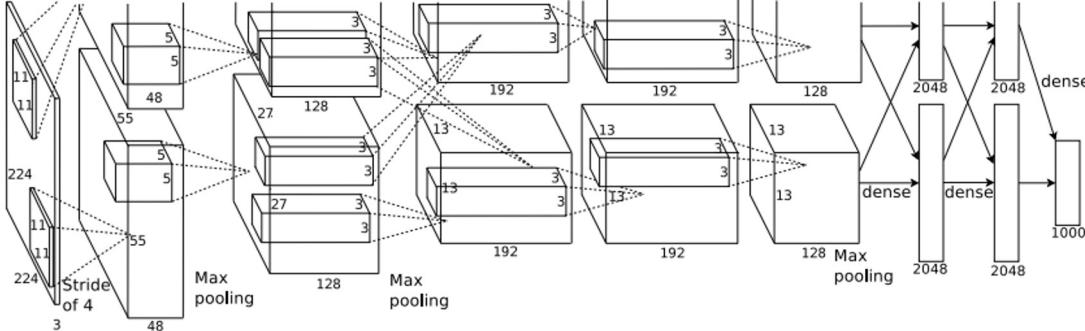
2017: DenseNet
>100 layers



2017: NASNet

AlexNet

227 x 227 inputs
5 Convolutional layers
Max pooling
3 fully-connected layers
ReLU nonlinearities

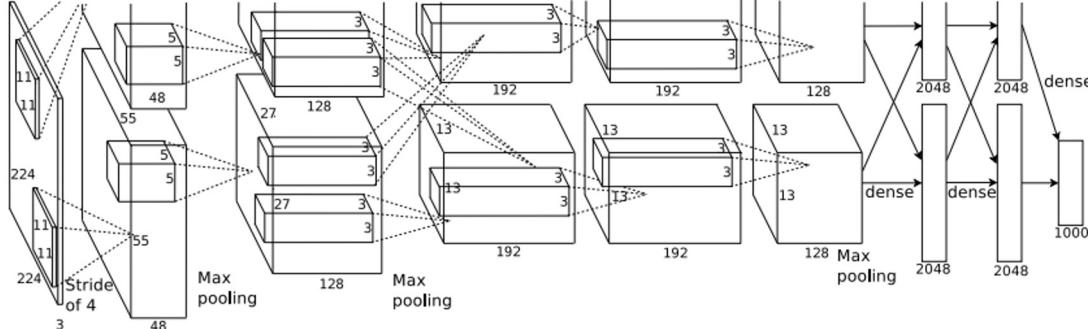


Used “Local response normalization”;
Not used anymore

Trained on two GTX 580 GPUs – only
3GB of memory each! Model split
over two GPUs

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

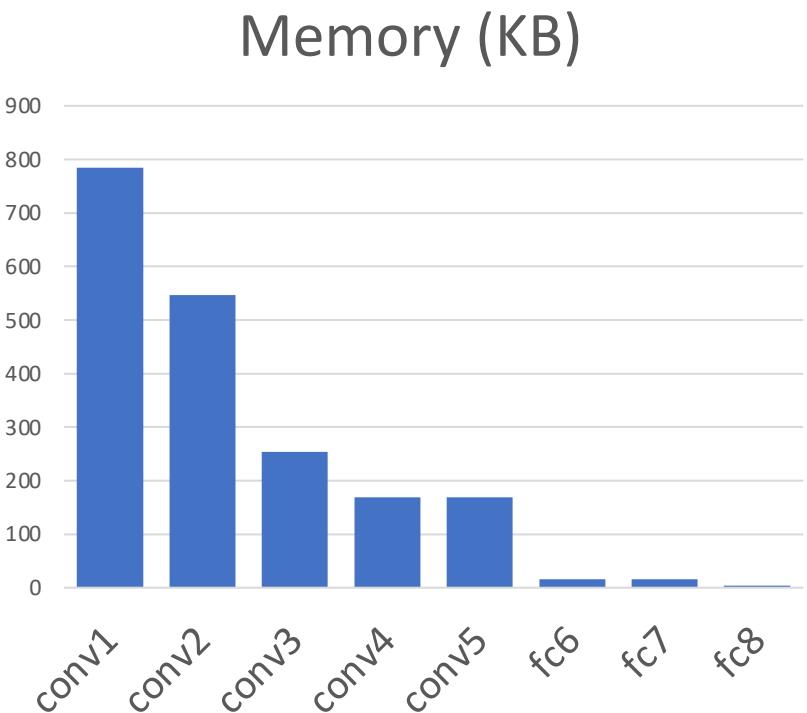
AlexNet



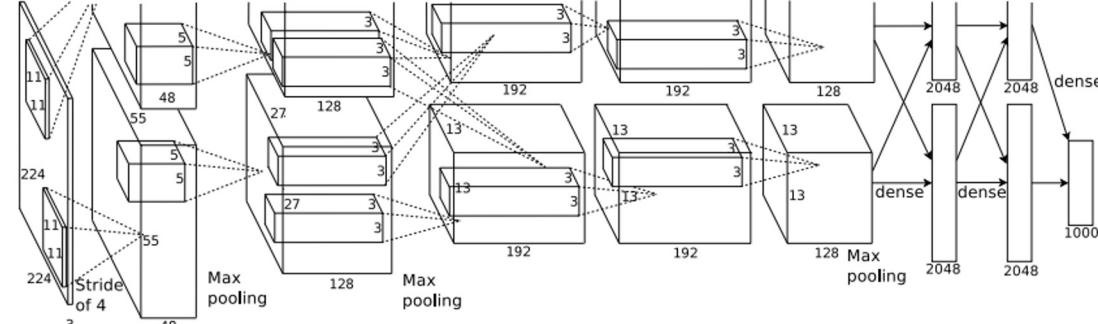
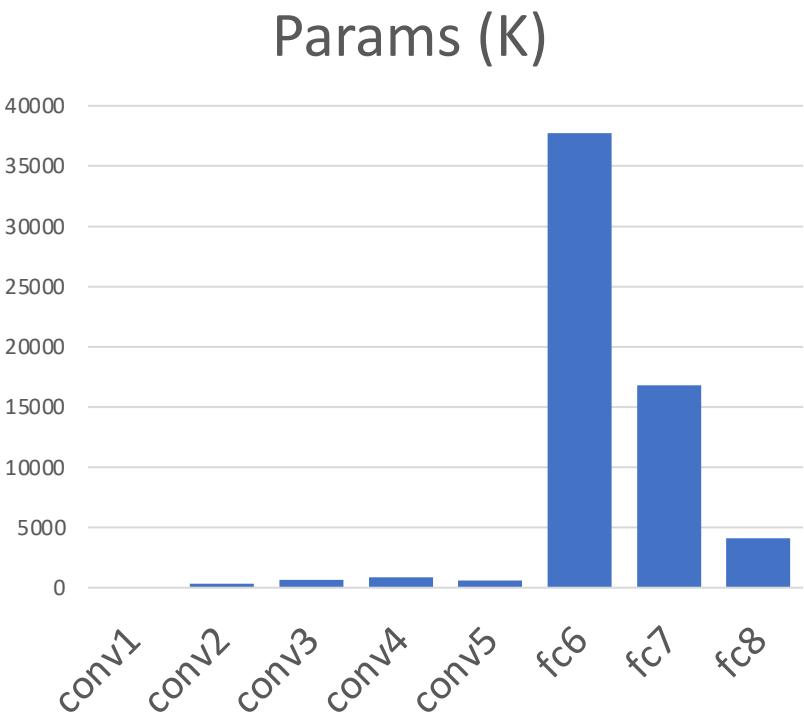
	Input size		Layer					Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)	
conv1	3	227	64	11	4	2	64	56	784	23	73	
pool1	64	56		3	2	0	64	27	182	0	0	
conv2	64	27	192	5	1	2	192	27	547	307	224	
pool2	192	27		3	2	0	192	13	127	0	0	
conv3	192	13	384	3	1	1	384	13	254	664	112	
conv4	384	13	256	3	1	1	256	13	169	885	145	
conv5	256	13	256	3	1	1	256	13	169	590	100	
pool5	256	13		3	2	0	256	6	36	0	0	
flatten	256	6					9216		36	0	0	
fc6	9216		4096				4096		16	37,749	38	
fc7	4096		4096				4096		16	16,777	17	
fc8	4096		1000				1000		4	4,096	4	

AlexNet

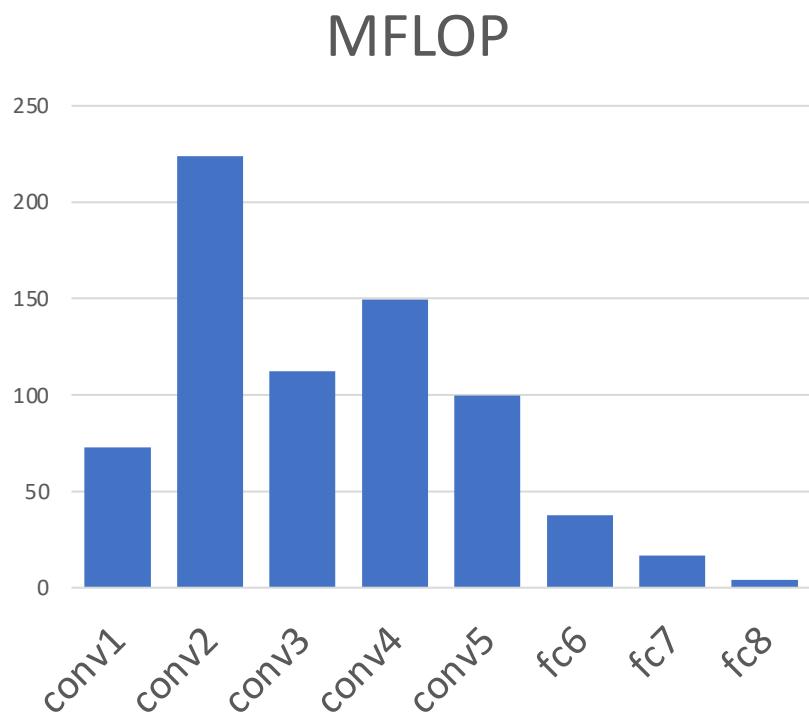
Most of the **memory usage** is in the early convolution layers



Nearly all **parameters** are in the fully-connected layers



Most **floating-point ops** occur in the convolution layers



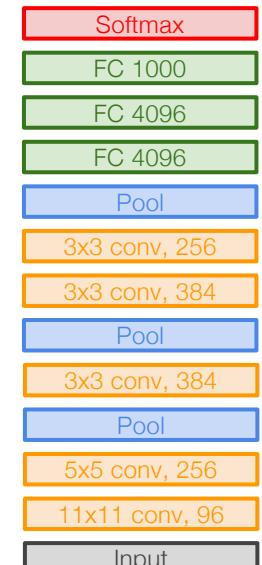
VGG: Deeper Networks, Regular Design

VGG Design rules:

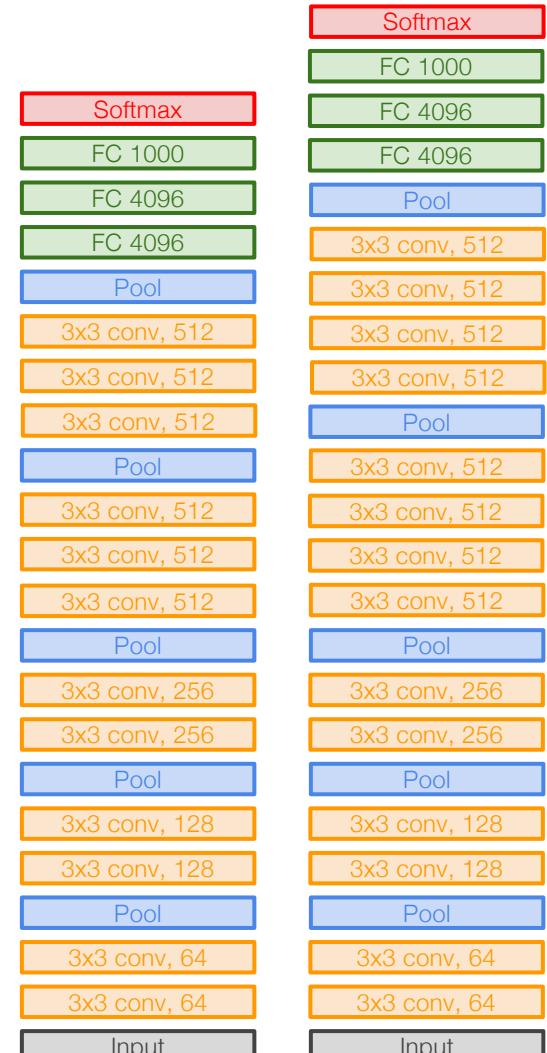
All conv are 3x3 stride 1 pad 1

All max pool are 2x2 stride 2

After pool, double #channels



AlexNet



VGG16

VGG19

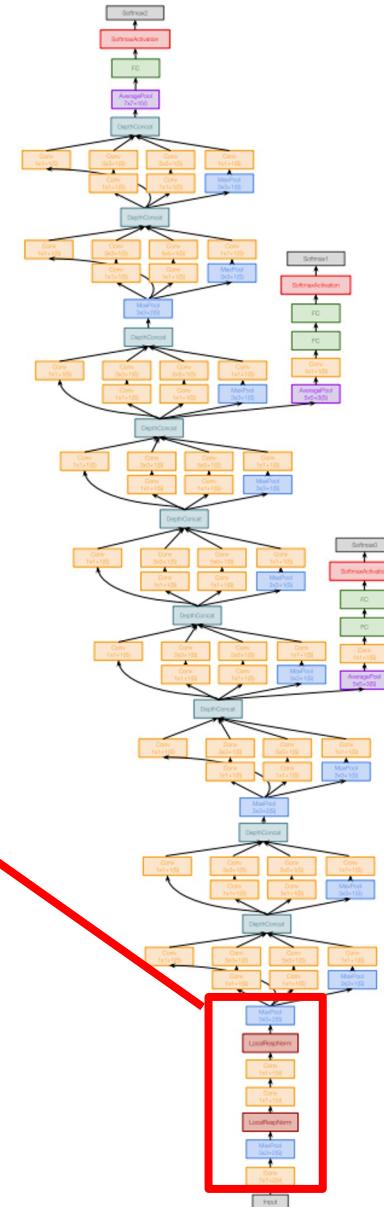
GoogLeNet: Aggressive Stem

Stem network at the start aggressively downsamples input
(Recall in VGG-16: Most of the compute was at the start)

Layer	Input size			Layer			Output size			memory (KB)	params (K)	flop (M)
	C	H / W	filters	kernel	stride	pad	C	H/W				
conv	3	224	64	7	2	3	64	112	3136	9	118	
max-pool	64	112		3	2	1	64	56	784	0	2	
conv	64	56	64	1	1	0	64	56	784	4	13	
conv	64	56	192	3	1	1	192	56	2352	111	347	
max-pool	192	56		3	2	1	192	28	588	0	1	

Total from 224 to 28 spatial resolution:
Memory: 7.5 MB
Params: 124K
MFLOP: 418

Compare VGG-16:
Memory: 42.9 MB (5.7x)
Params: 1.1M (8.9x)
MFLOP: 7485 (17.8x)

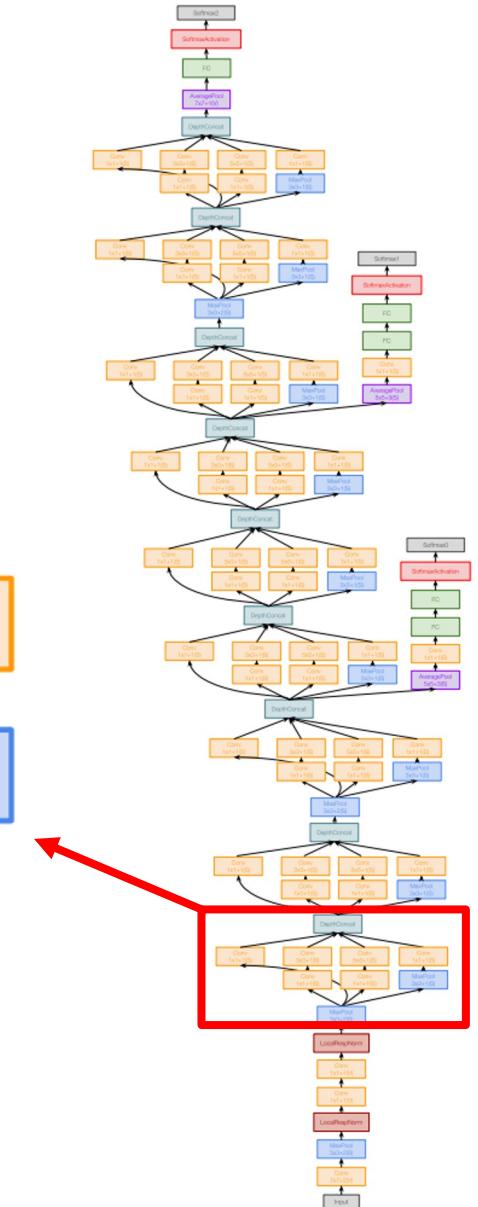
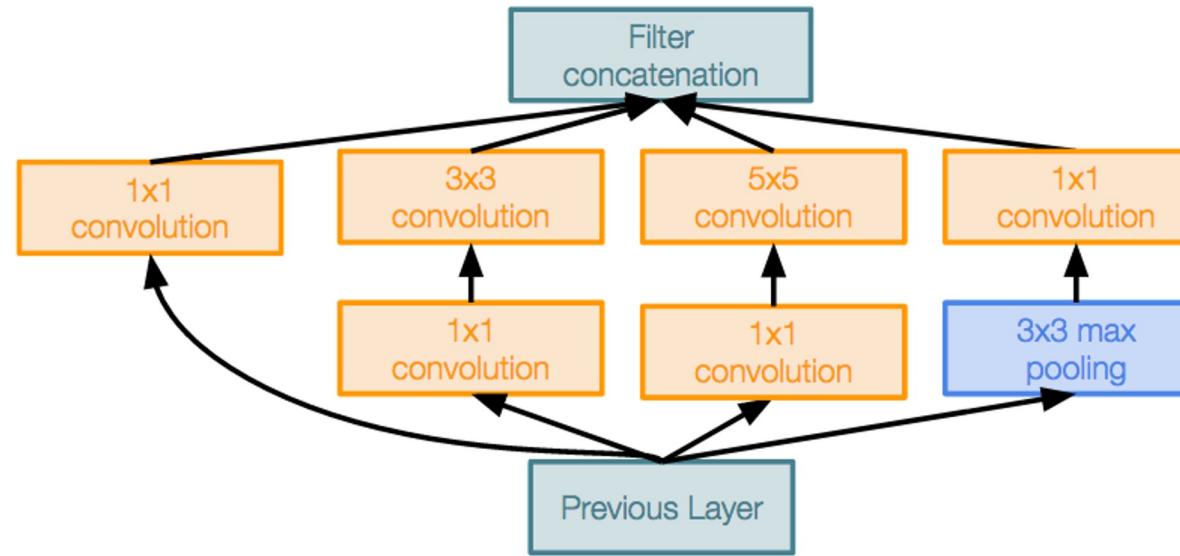


GoogLeNet: Inception Module

Inception module

Local unit with parallel branches

Local structure repeated many times throughout the network



GoogLeNet: Global Average Pooling

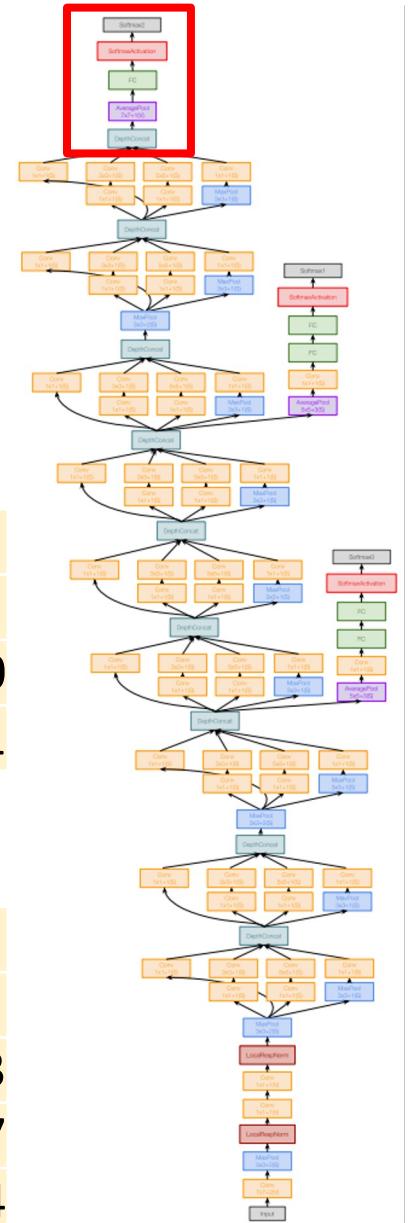
No large FC layers at the end! Instead uses **global average pooling** to collapse spatial dimensions, and one linear layer to produce class scores
 (Recall VGG-16: Most parameters were in the FC layers!)

e.g. GAP: $1024 \times 7 \times 7 \rightarrow 1024 \times 1$

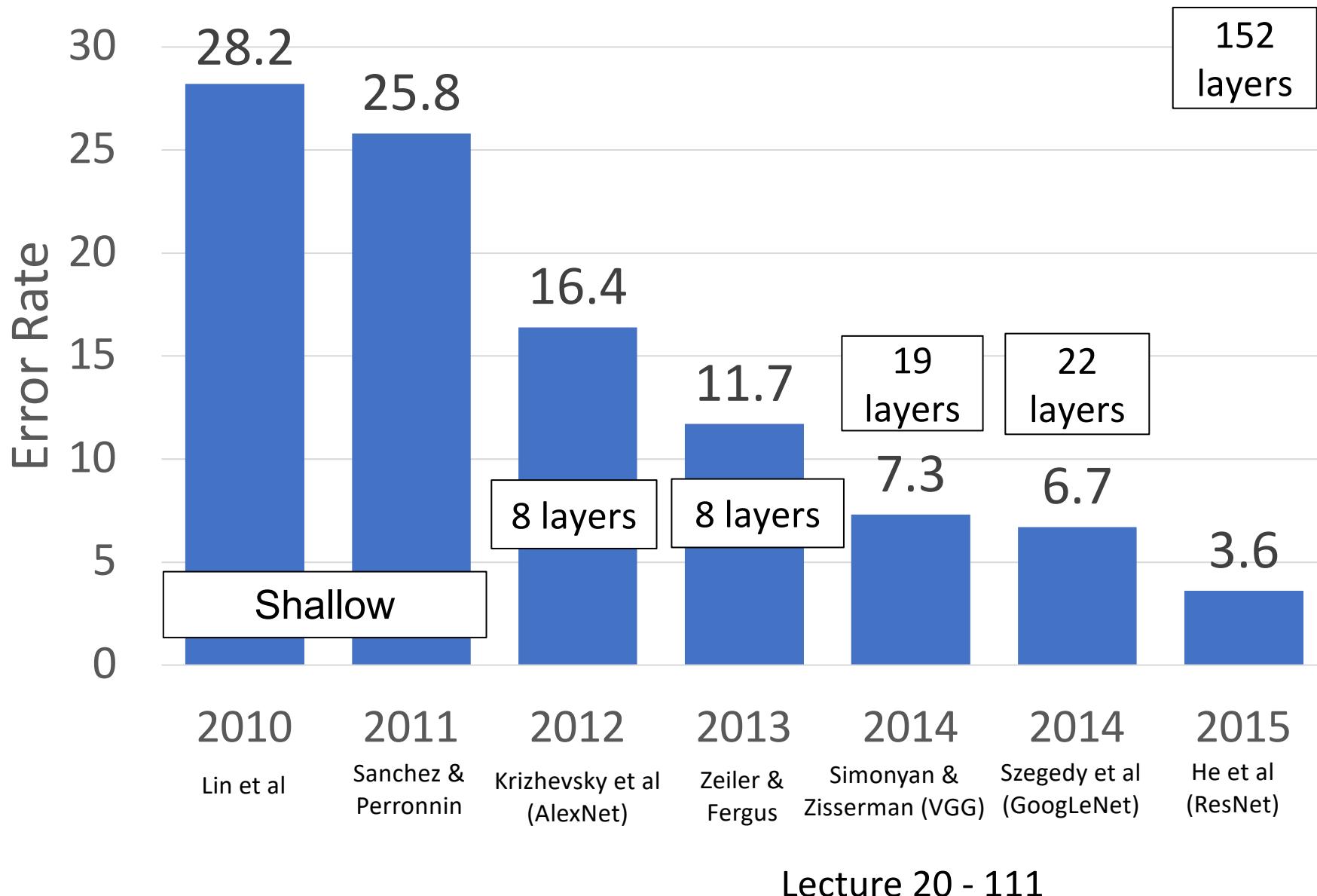
	Input size		Layer				Output size				
Layer	C	H/W	filters	kernel	stride	pad	C	H/W	memory (KB)	params (k)	flop (M)
avg-pool	1024	7		7	1	0	1024	1	4	0	0
fc	1024		1000				1000		0	1025	1

Compare with VGG-16:

Layer	C	H/W	filters	kernel	stride	pad	C	H/W	memory (KB)	params (K)	flop (M)
flatten	512	7					25088		98		
fc6	25088		4096				4096		16	102760	103
fc7	4096		4096				4096		16	16777	17
fc8	4096		1000				1000		4	4096	4



ImageNet Classification Challenge

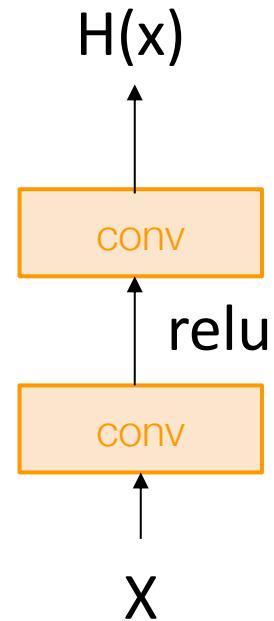


Kaiming He



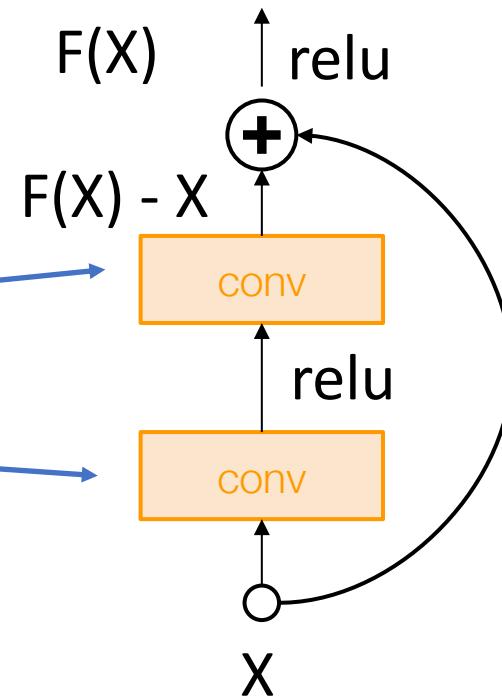
Residual Networks

Solution: Change the network so learning identity functions with extra layers is easy!



“Plain” block

If you set these to
0, the whole block
will compute the
identity function!



Residual Block

Additive
“shortcut”

Residual Block Implementation

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l

class Residual(nn.Module): #@save
    """The Residual block of ResNet."""
    def __init__(self, input_channels, num_channels, use_1x1conv=False,
                 strides=1):
        super().__init__()
        self.conv1 = nn.Conv2d(input_channels, num_channels, kernel_size=3,
                             padding=1, stride=strides)
        self.conv2 = nn.Conv2d(num_channels, num_channels, kernel_size=3,
                             padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2d(input_channels, num_channels,
                                 kernel_size=1, stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm2d(num_channels)
        self.bn2 = nn.BatchNorm2d(num_channels)

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)
```

Residual Networks

ResNet-18:

Stem: 1 conv layer

Stage 1 (C=64): 2 res. block = 4 conv

Stage 2 (C=128): 2 res. block = 4 conv

Stage 3 (C=256): 2 res. block = 4 conv

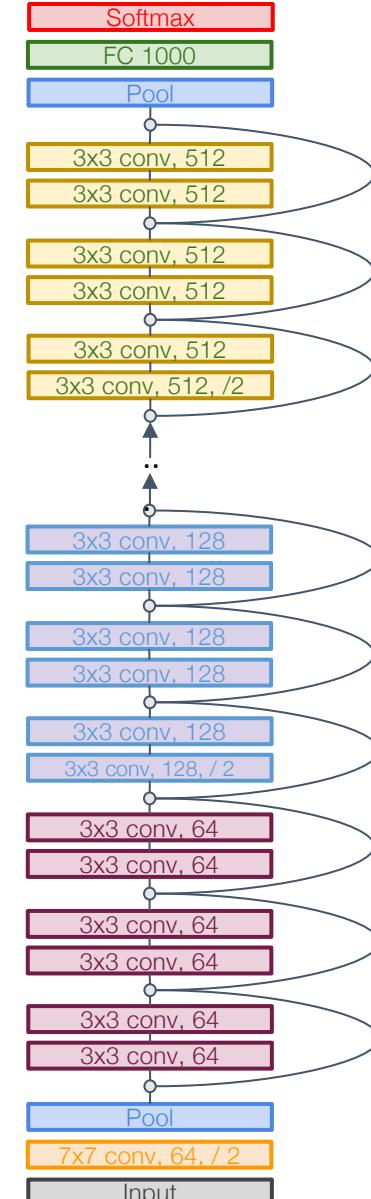
Stage 4 (C=512): 2 res. block = 4 conv

Linear

ImageNet top-5 error: 10.92

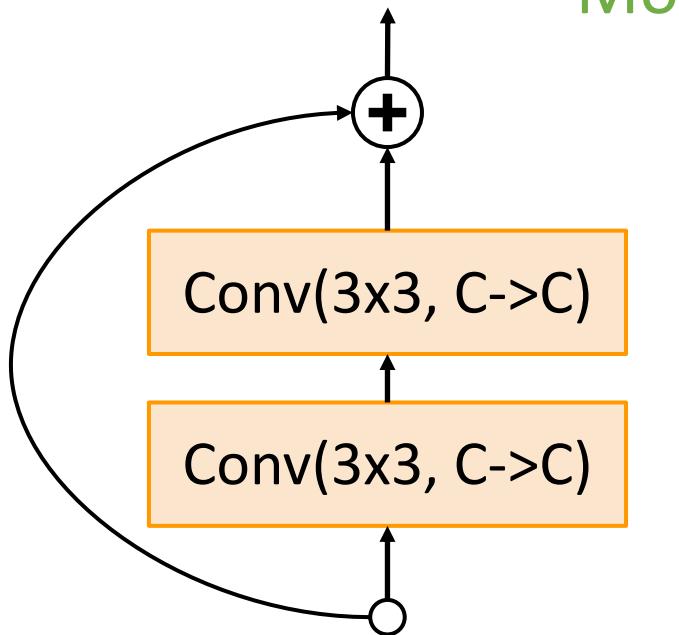
GFLOP: 1.8

He et al, "Deep Residual Learning for Image Recognition", CVPR 2016
Error rates are 224x224 single-crop testing, reported by [torchvision](#)



Residual Networks: Bottleneck Block

More layers, less computational cost!



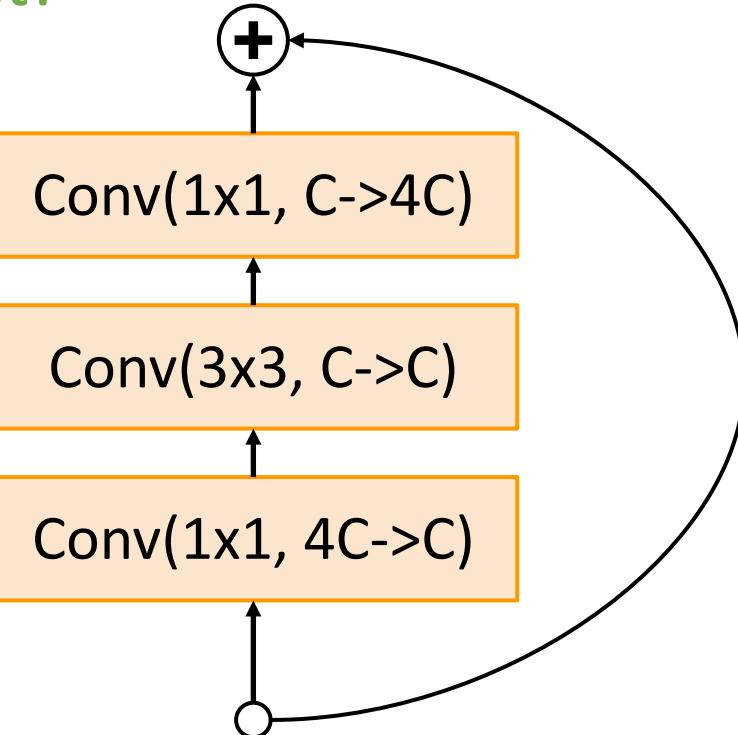
"Basic"
Residual block

Total FLOPs:
 $18HWC^2$

FLOPs: $4HWC^2$

FLOPs: $9HWC^2$

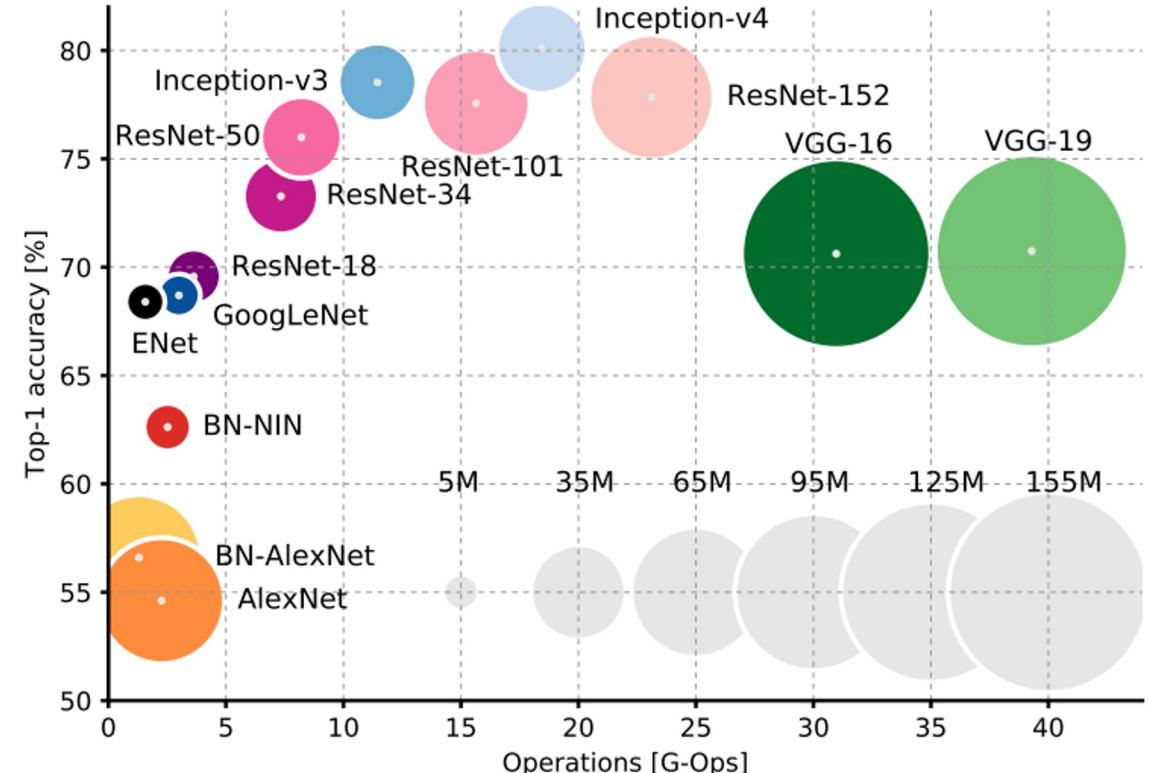
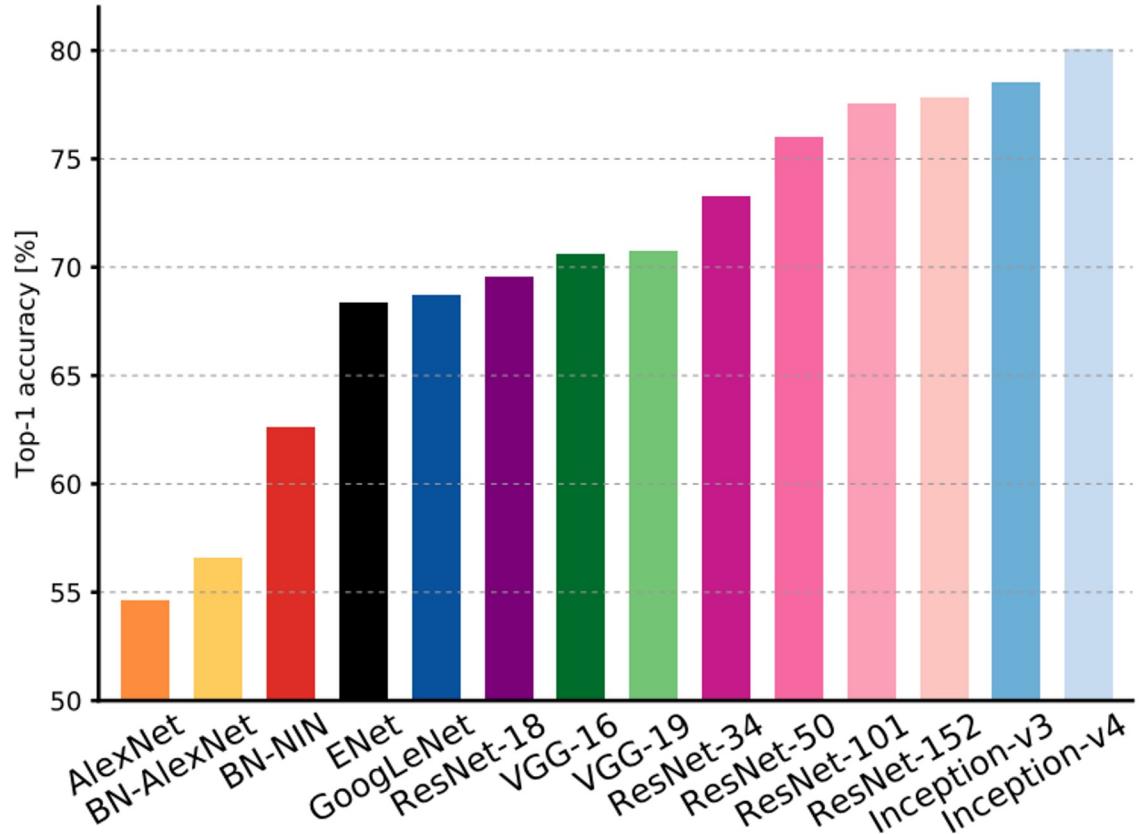
FLOPs: $4HWC^2$



"Bottleneck"
Residual block

Total FLOPs:
 $17HWC^2$

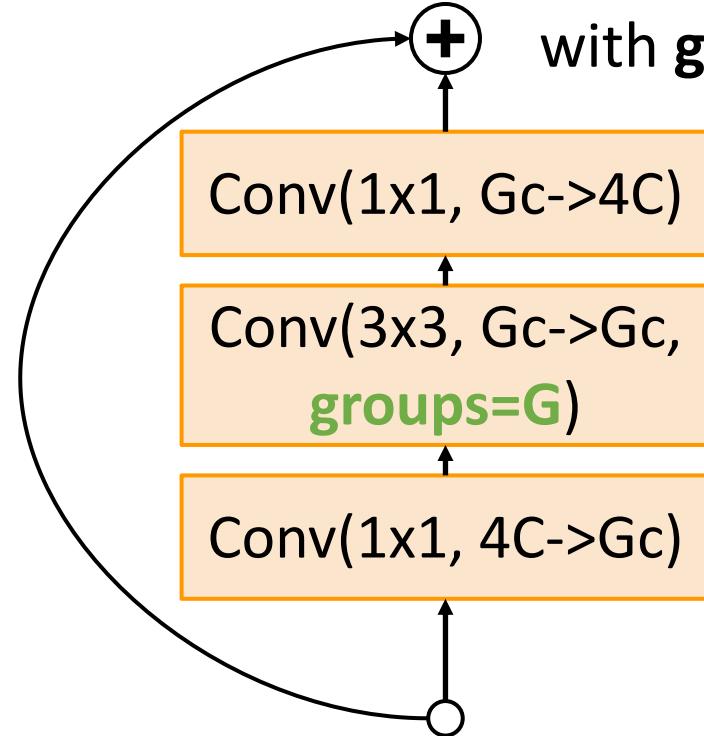
Comparing Complexity



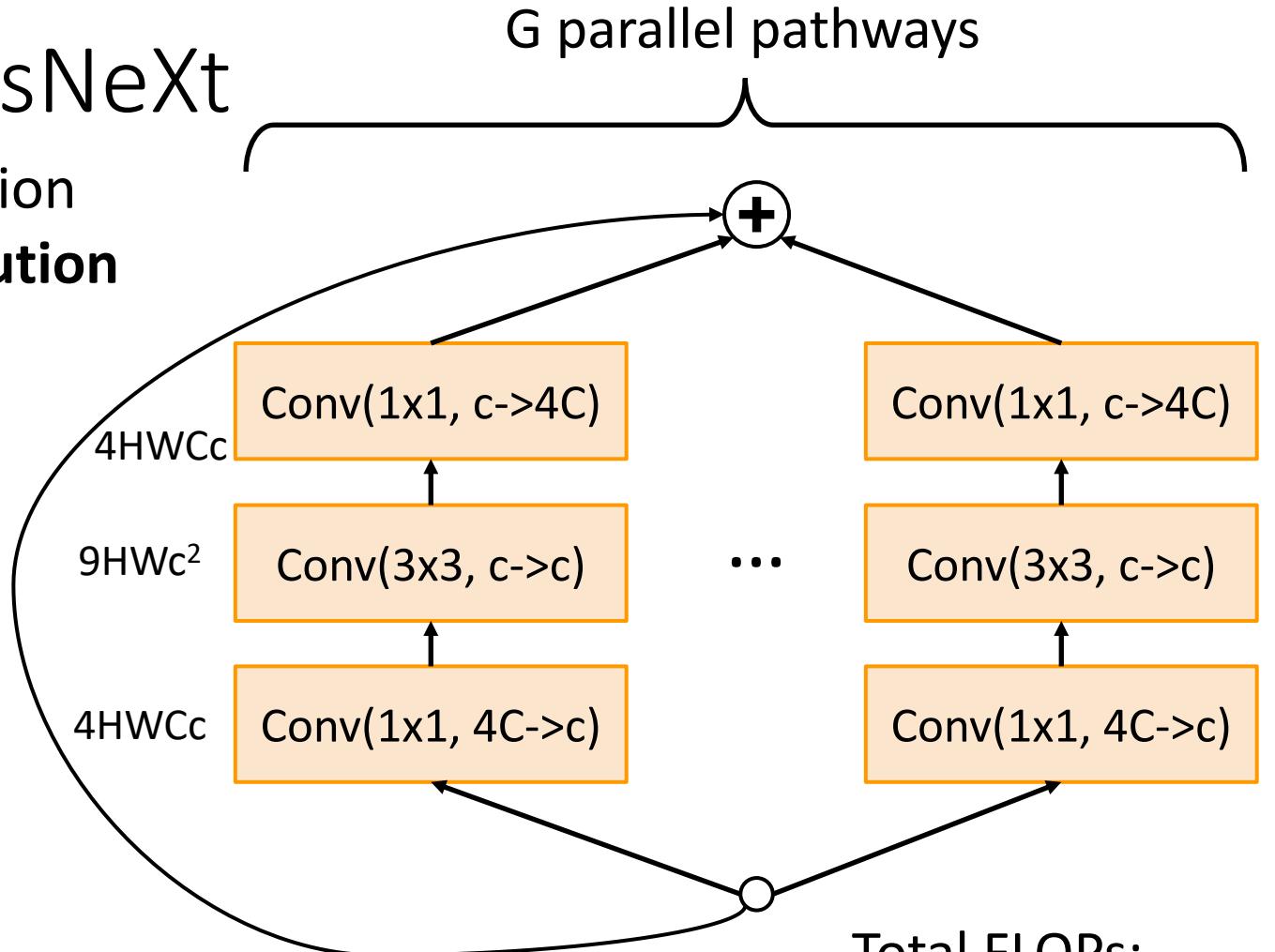
Canziani et al, "An analysis of deep neural network models for practical applications", 2017

Improving ResNets: ResNeXt

Equivalent formulation
with **grouped convolution**



ResNeXt block:
Grouped convolution



Equal cost when
 $9Gc^2 + 8GCc - 17C^2 = 0$

Example: $C=64, G=4, c=24; C=64, G=32, c=4$

Total FLOPs:
 $(8Cc + 9c^2)*HWG$

Grouped Convolution

Convolution with groups=1:

Normal convolution

Input: $C_{in} \times H \times W$

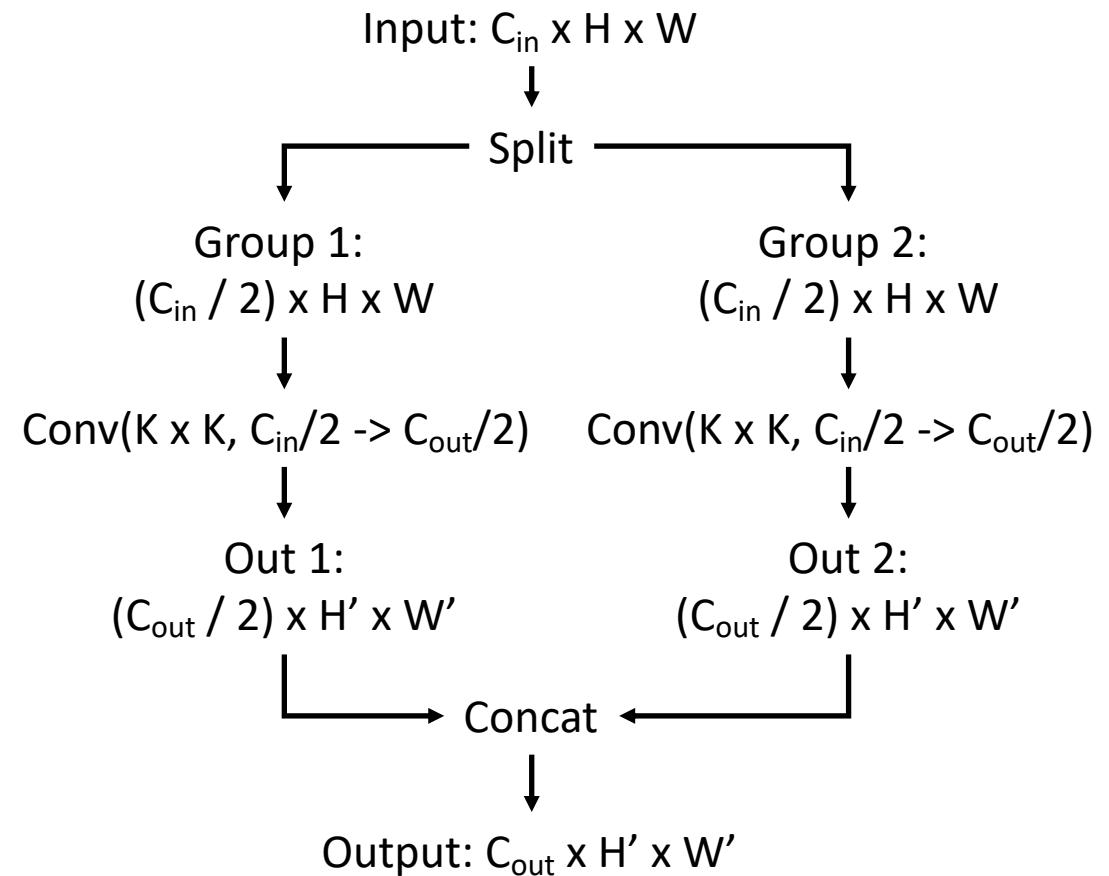
Weight: $C_{out} \times C_{in} \times K \times K$

Output: $C_{out} \times H' \times W'$

FLOPs: $C_{out} C_{in} K^2 HW$

All convolutional kernels touch
all C_{in} channels of the input

Convolution with groups=2:
Two parallel convolution layers that
work on half the channels



Lecture 9: Training Neural Networks

Overview

1. One time setup

Activation functions, data preprocessing, weight initialization, regularization

2. Training dynamics

Learning rate schedules; hyperparameter optimization

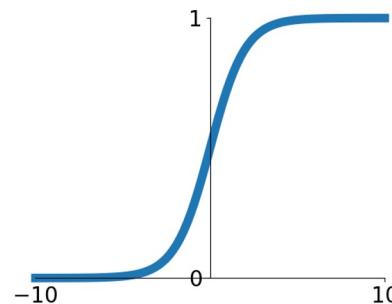
3. After training

Model ensembles, transfer learning

Activation Functions

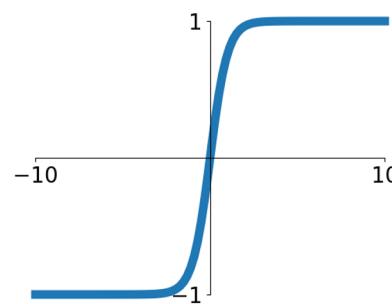
Sigmoid

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



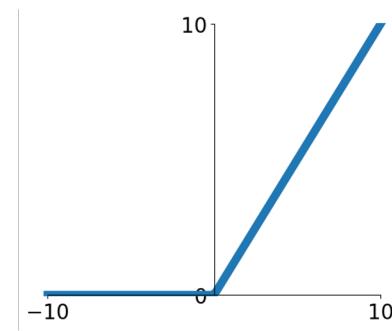
tanh

$$\tanh(x)$$



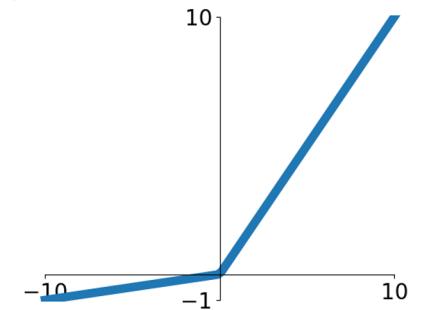
ReLU

$$\max(0, x)$$



Leaky ReLU

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

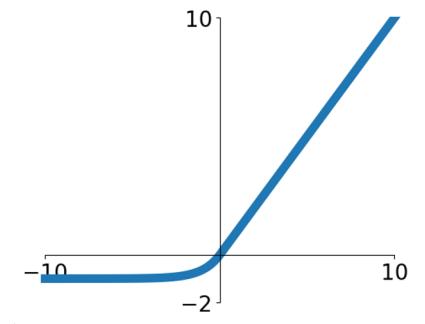


Maxout

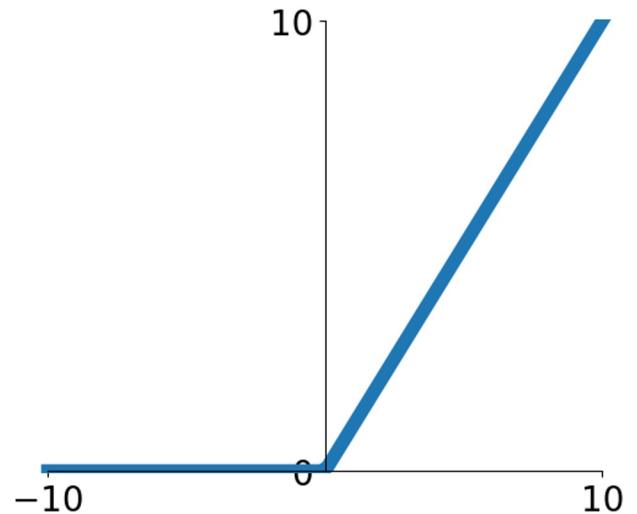
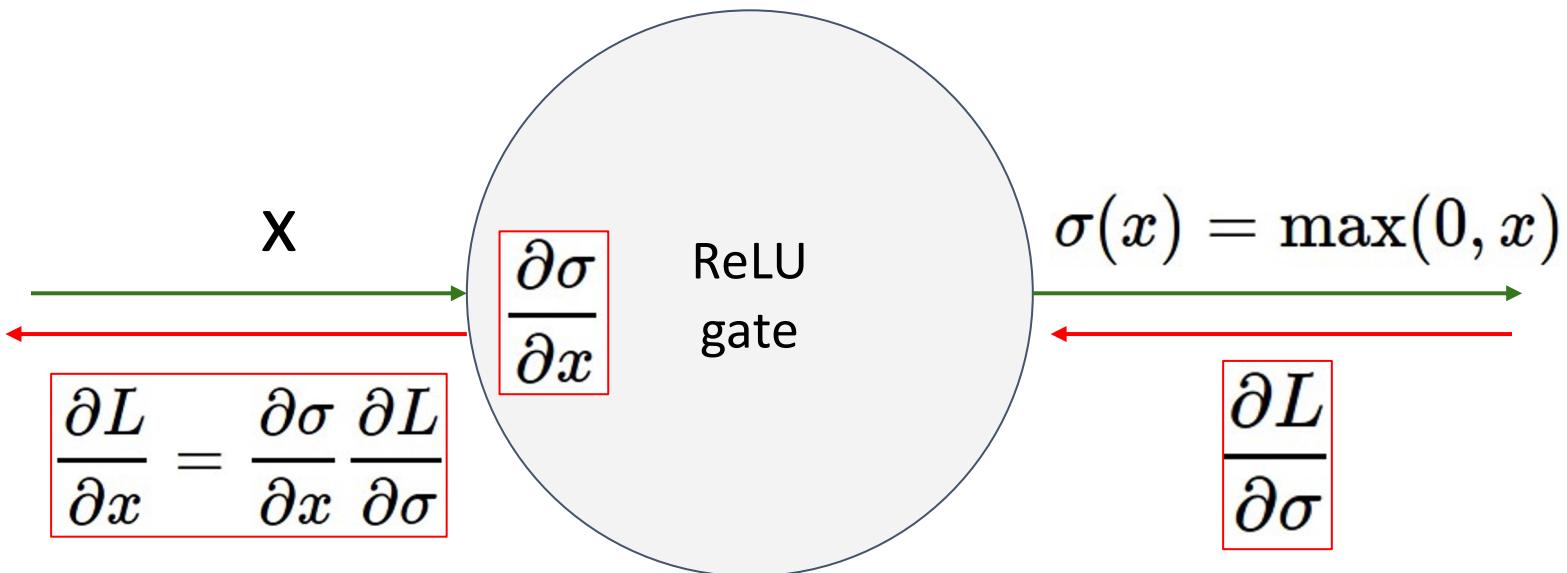
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

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



Activation Functions: ReLU

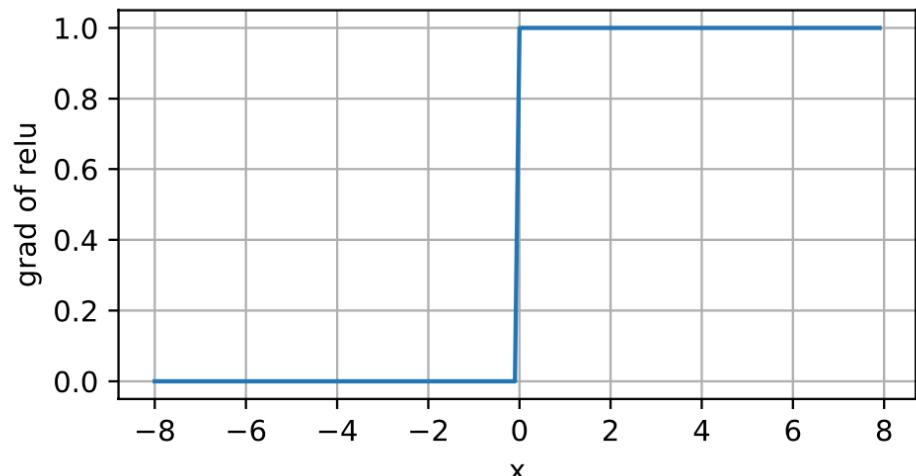


What happens when $x = -8$?

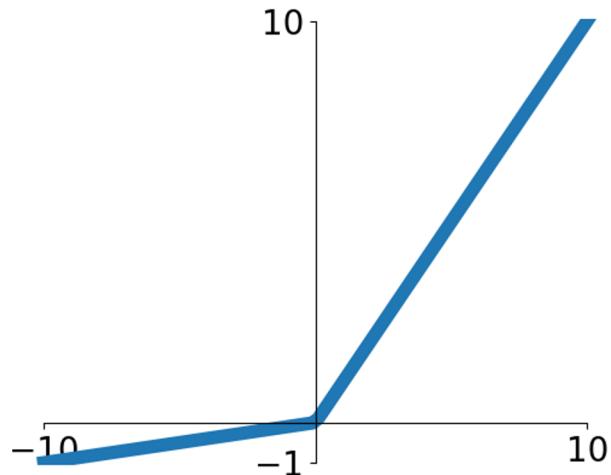
What happens when $x = 0$?

What happens when $x = 8$?

dead ReLU will never activate
=> never update



Activation Functions: Leaky ReLU



Leaky ReLU

$$f(x) = \max(\alpha x, x)$$

α is a hyperparameter,
often $\alpha = 0.1$

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not “die”.

Parametric ReLU (PReLU)

$$f(x) = \max(\alpha x, x)$$

α is learned via backprop

He et al, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, ICCV 2015

Data Preprocessing for Images

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
(mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
(mean along each channel = 3 numbers)
- Subtract per-channel mean and
Divide by per-channel std (e.g. ResNet)
(mean along each channel = 3 numbers)

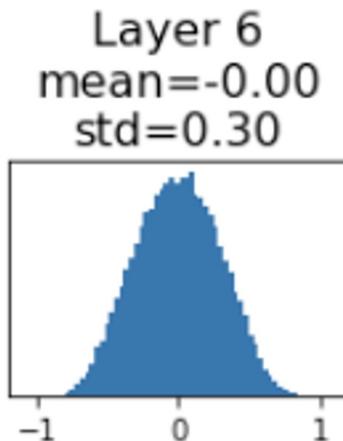
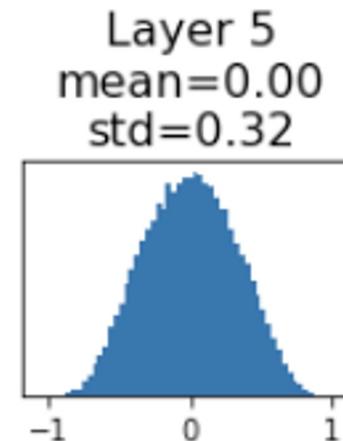
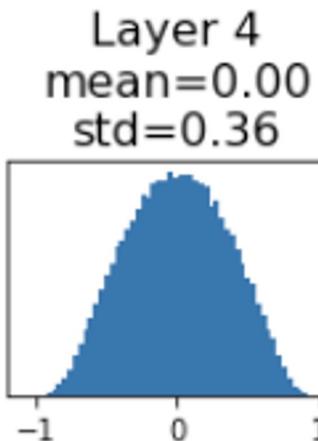
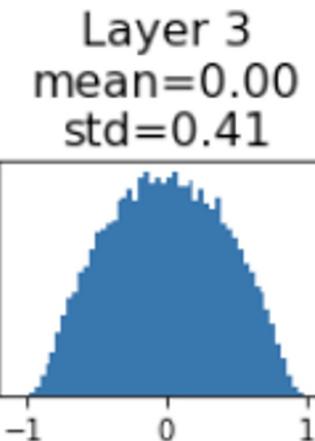
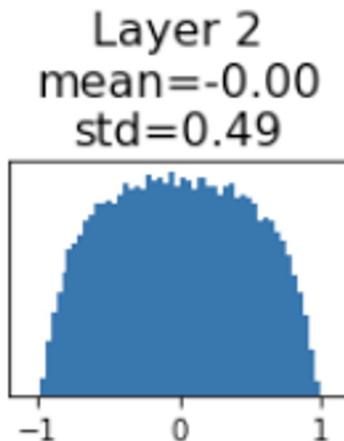
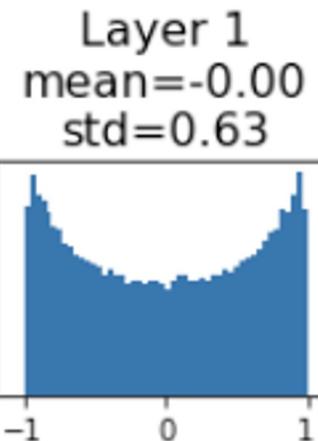
Not common to do PCA or whitening, as correlated features contain information!

Weight Initialization: Xavier Initialization

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

“Just right”: Activations are nicely scaled for all layers!

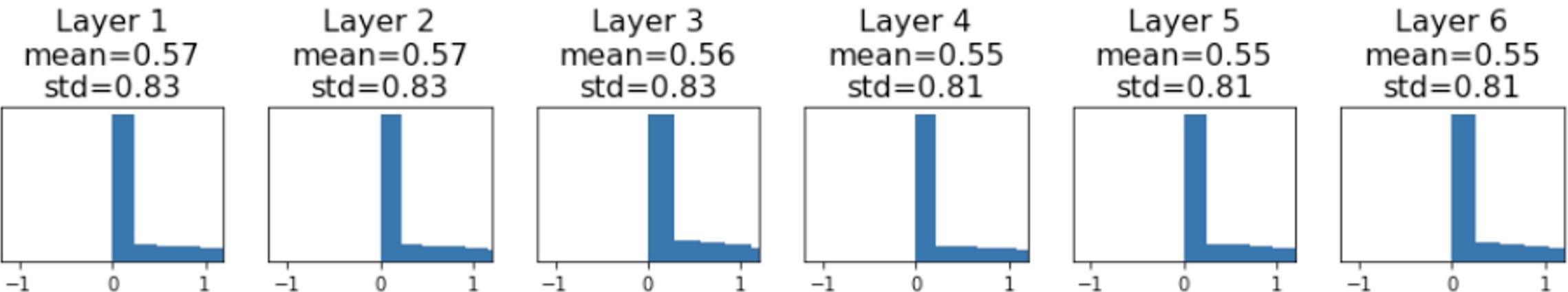
For conv layers, Din is $\text{kernel_size}^2 * \text{input_channels}$



Weight Initialization: Kaiming Initialization

```
dims = [4096] * 7 # ReLU correction: std = sqrt(2 / Din)
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

"Just right" – activations nicely scaled for all layers

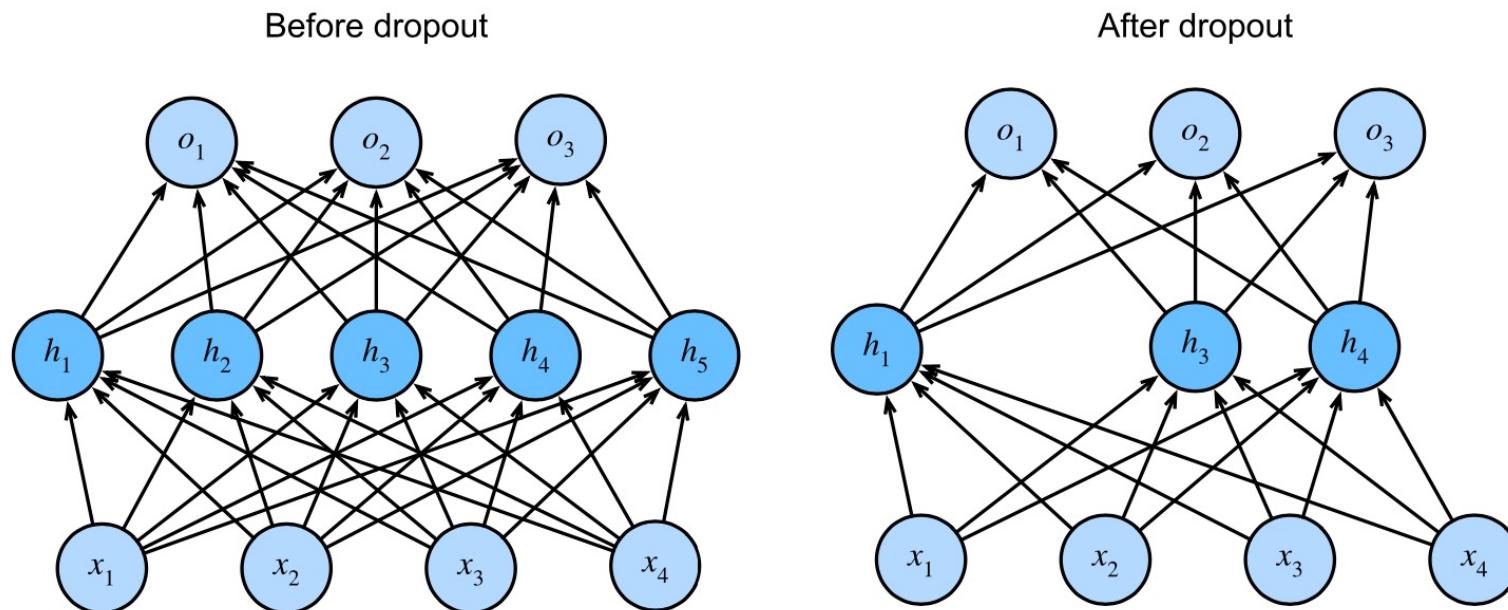


See this paper for detailed derivation!

He et al, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", ICCV 2015

Regularization: Dropout

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common

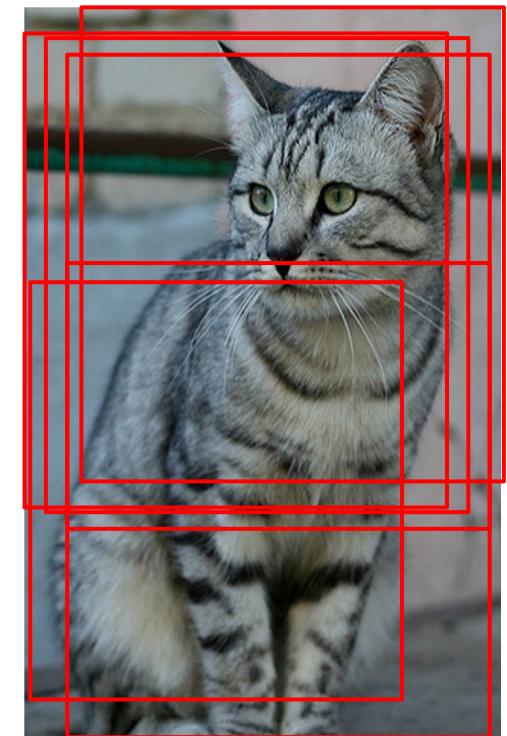


Data Augmentation: Random Crops and Scales

Training: sample random crops / scales

ResNet:

1. Pick random L in range $[256, 480]$
2. Resize training image, short side = L
3. Sample random 224×224 patch



Data Augmentation: more image distortions

Random mix/combinations of :

- translation
- rotation
- stretching
- shearing,
- lens distortions,
- ... (go crazy)



Regularization: Mixup

Training: Train on random blends of images

Testing: Use original images

Examples:

Dropout

Batch Normalization

Data Augmentation

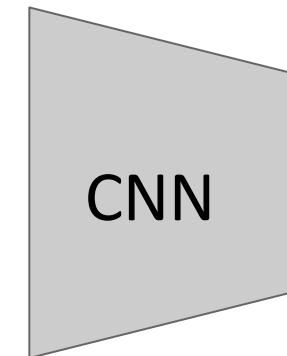
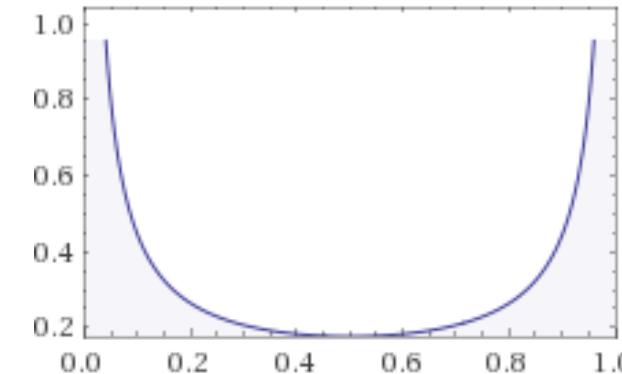
DropConnect

Fractional Max Pooling

Stochastic Depth

Cutout

Mixup



Target label:
cat: 0.4
dog: 0.6

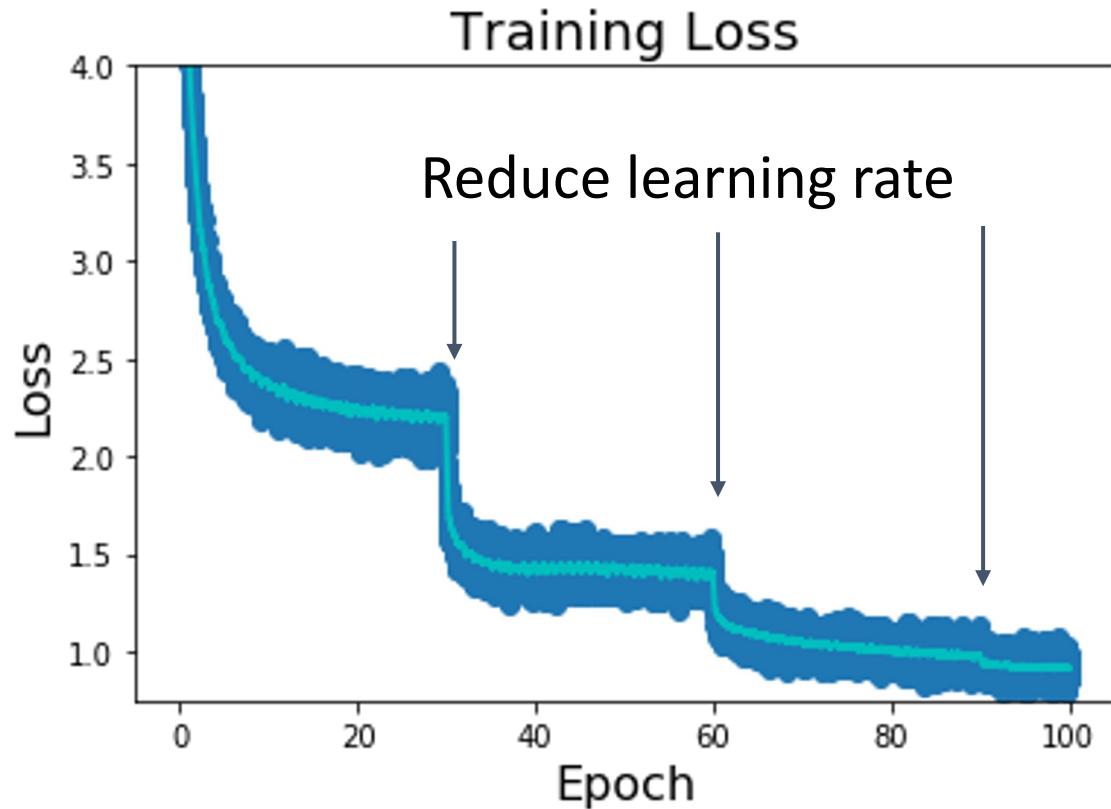
Randomly blend the pixels of pairs of training images, e.g.
40% cat, 60% dog

Zhang et al, "mixup: Beyond Empirical Risk Minimization", ICLR 2018

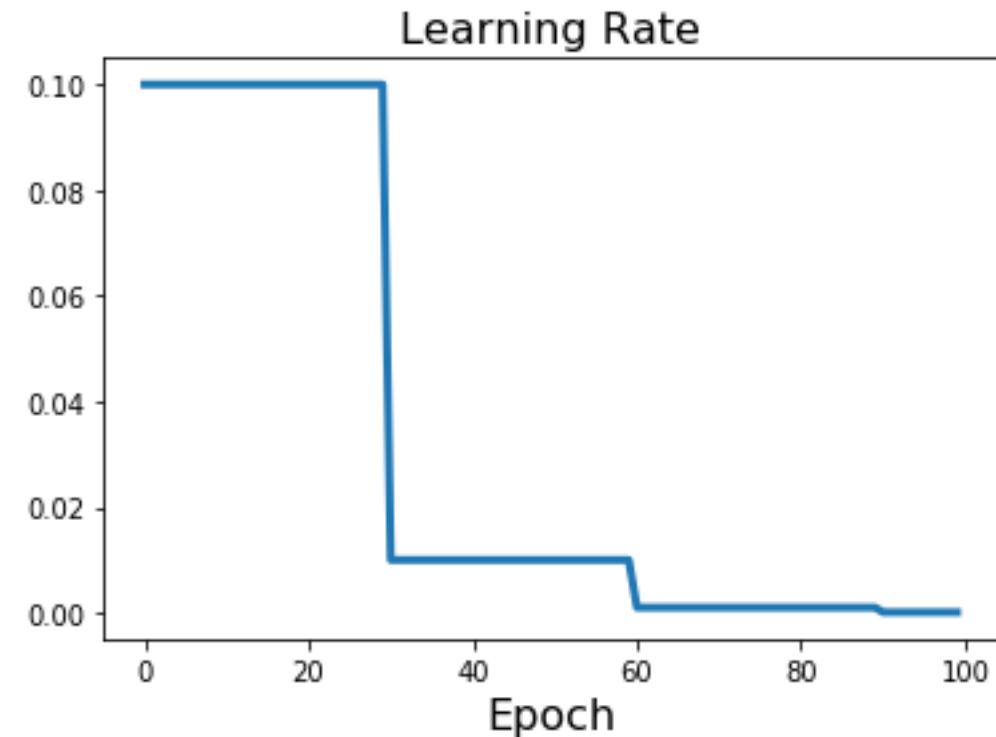
Why it works? some interesting analysis in the paper:

<https://arxiv.org/pdf/1710.09412.pdf>

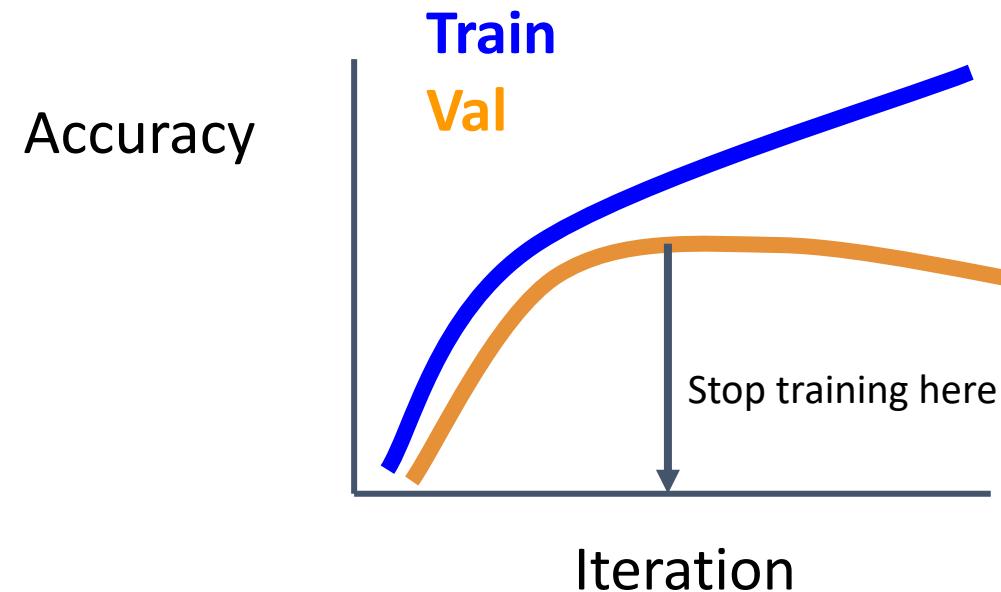
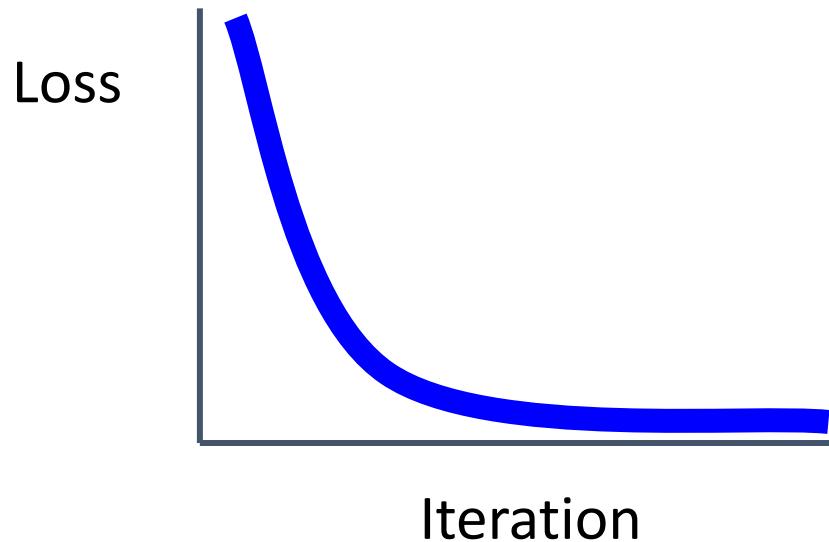
Learning Rate Decay: Step (most common)



Step: Reduce learning rate at a few fixed points.
E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.



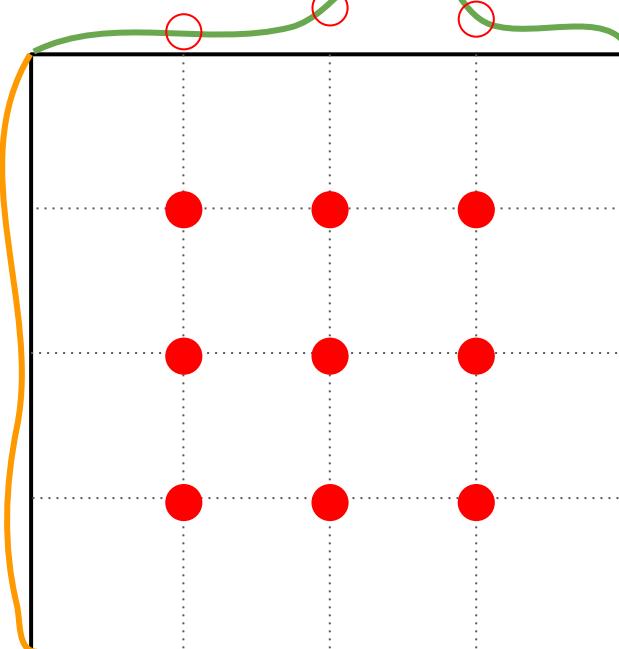
How long to train? Early Stopping



Stop training the model when accuracy on the validation set decreases
Or train for a long time, but always keep track of the model snapshot that worked best on val. **Always a good idea to do this!**

Hyperparameters: Random vs Grid Search

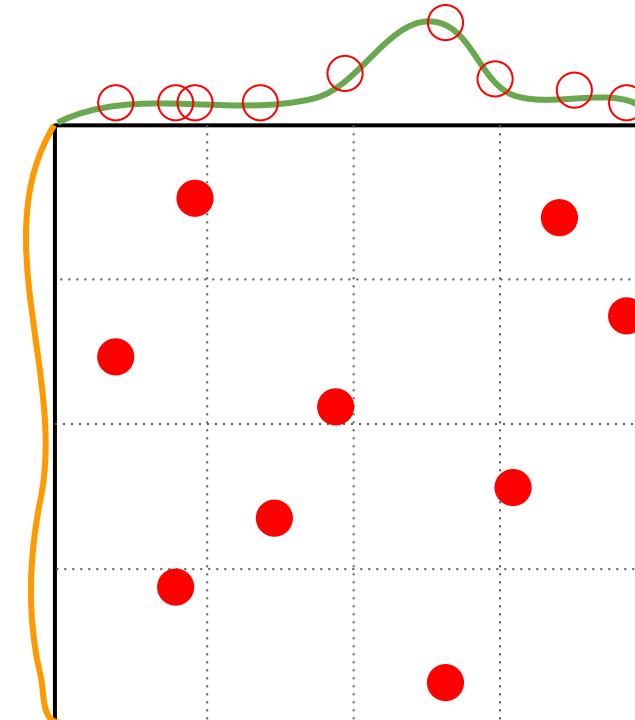
Grid Layout



Important
Parameter

Unimportant
Parameter

Random Layout



Important
Parameter

Unimportant
Parameter

Model Ensembles

1. Train multiple independent models
2. At test time average their results
(Take average of predicted probability distributions, then choose argmax)

Enjoy 2% extra performance

Transfer Learning with CNNs



More specific

More generic

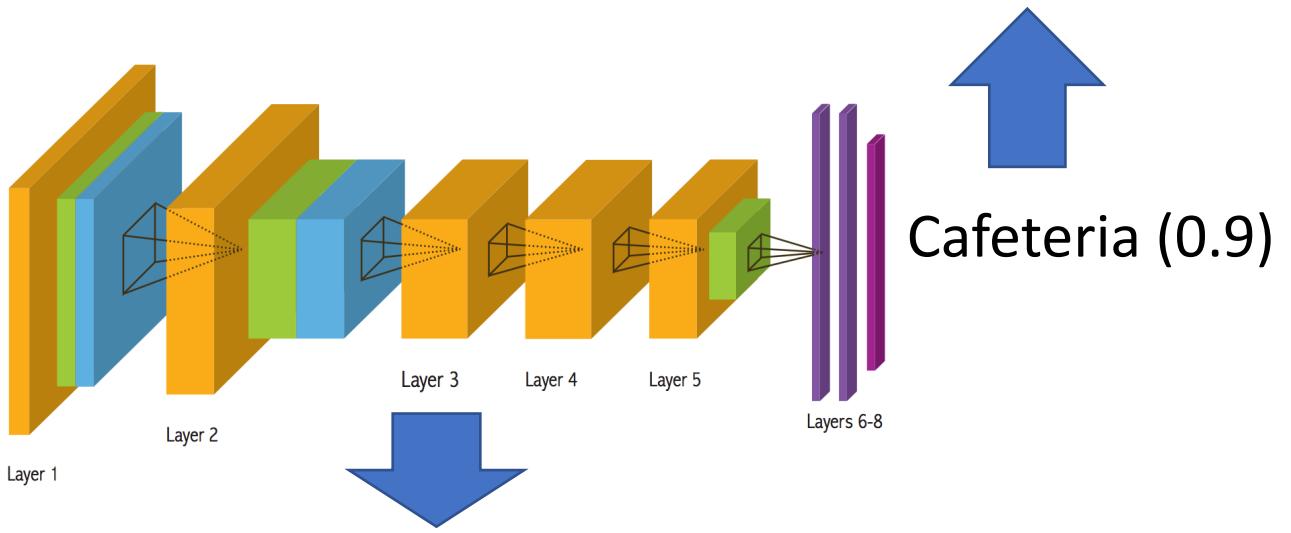
	Dataset similar to ImageNet	Dataset very different from ImageNet
very little data (10s to 100s)	Use Linear Classifier on top layer	You're in trouble... Try linear classifier from different layers
quite a lot of data (100s to 1000s)	Finetune a few layers	Finetune a larger number of layers

Lecture 10: Visualizing and Understanding Neural Network

What's going on inside ConvNet?



2. Why is this output?



1. What have been learned inside?

Unit2 at Layer4: Lamp



Unit5 at Layer3 : Trademark

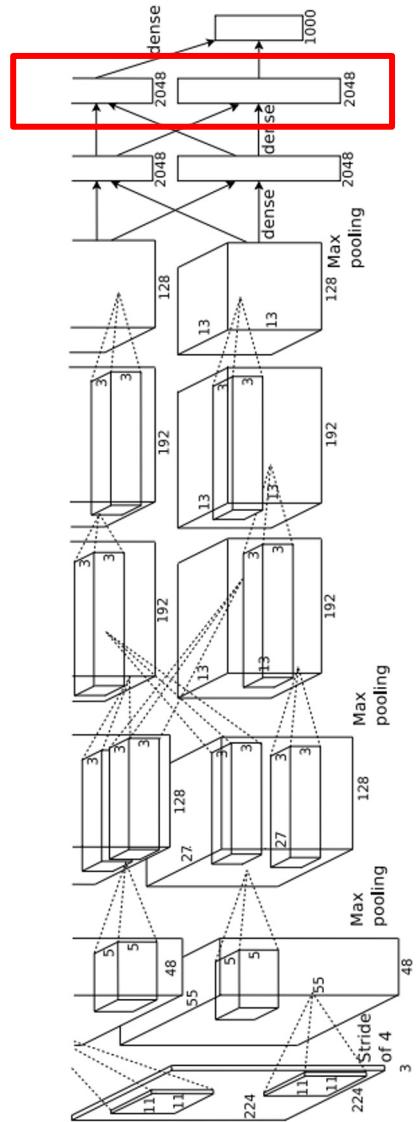
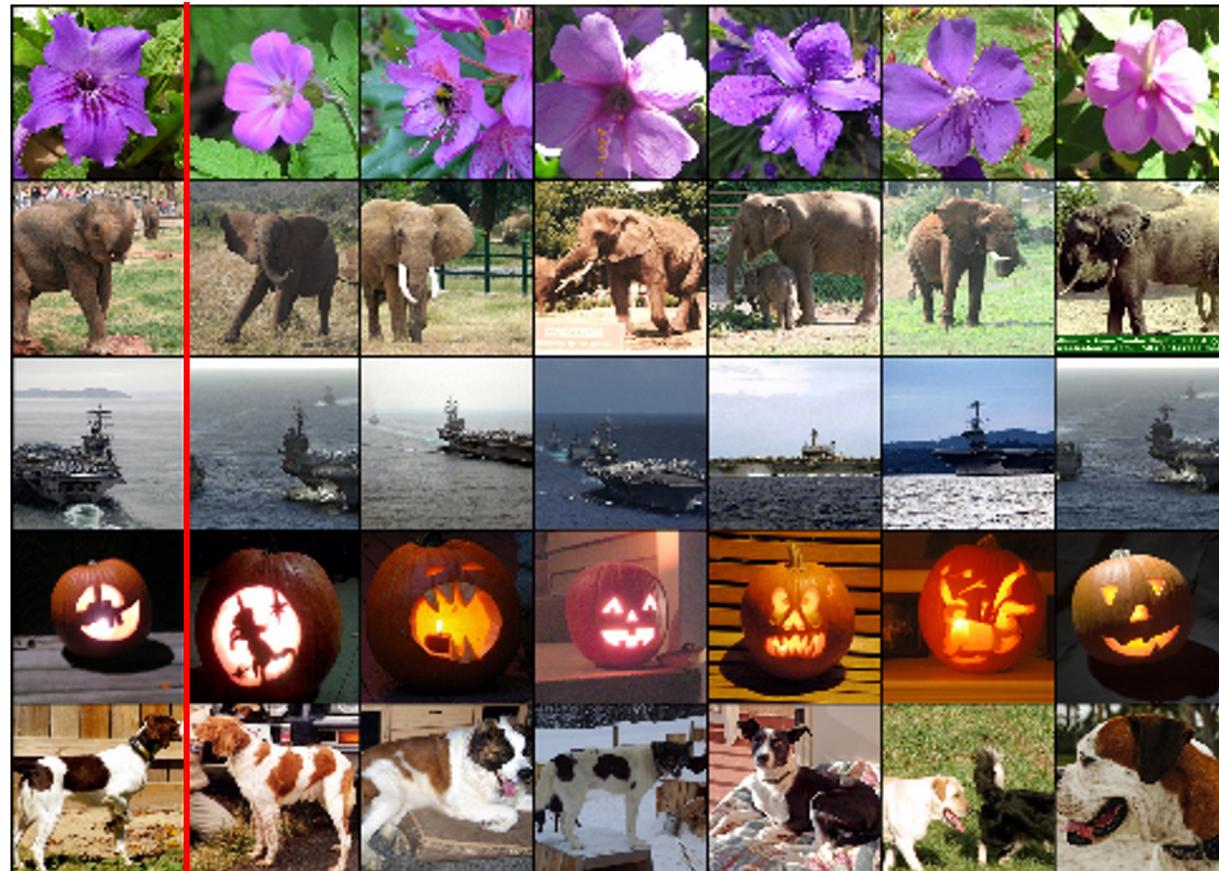


Last Layer: Nearest Neighbors

Recall: Nearest neighbors in pixel space



Test
image L2 Nearest neighbors in feature space



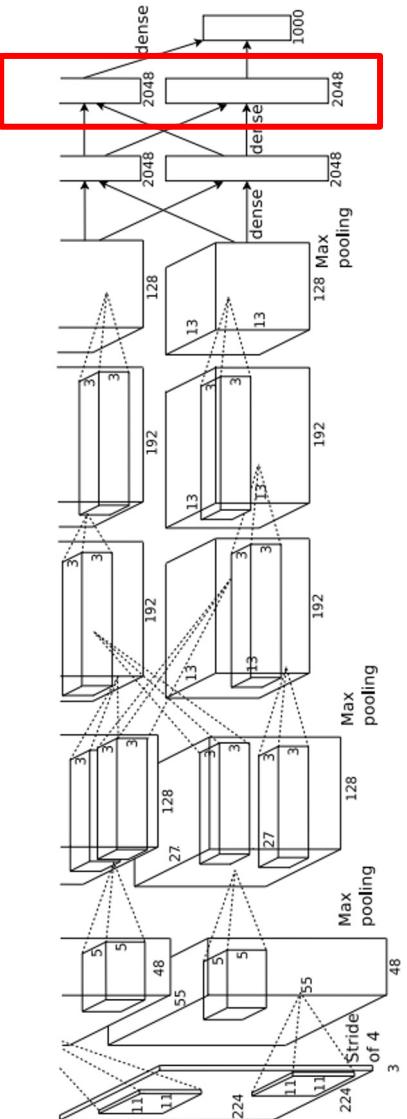
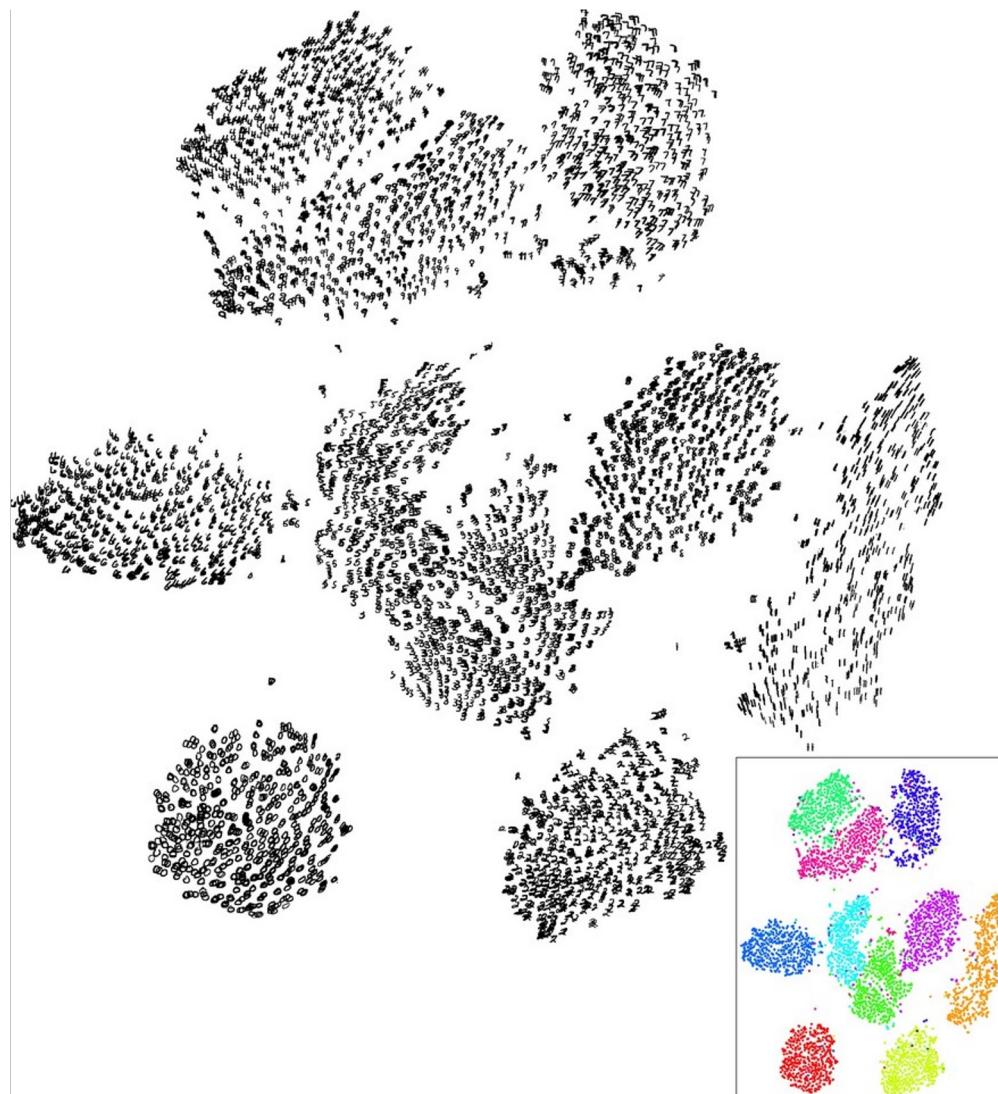
Krizhevsky et al, "ImageNet Classification with Deep Convolutional Neural Networks", NeurIPS 2012.
Figures reproduced with permission.

Last Layer: Dimensionality Reduction

Visualize the “space” of FC7
feature vectors by reducing
dimensionality of vectors from
4096 to 2 dimensions

Simple algorithm: Principal
Component Analysis (PCA)

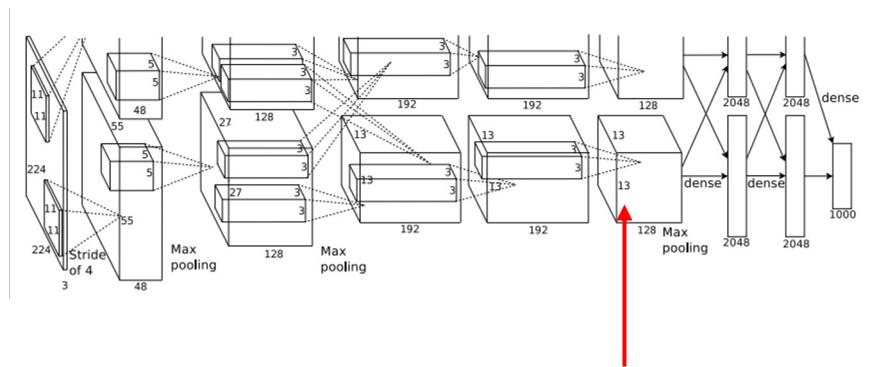
More complex: t-SNE



Van der Maaten and Hinton, “Visualizing Data using t-SNE”, JMLR 2008

Figure copyright Laurens van der Maaten and Geoff Hinton, 2008. Reproduced with permission.

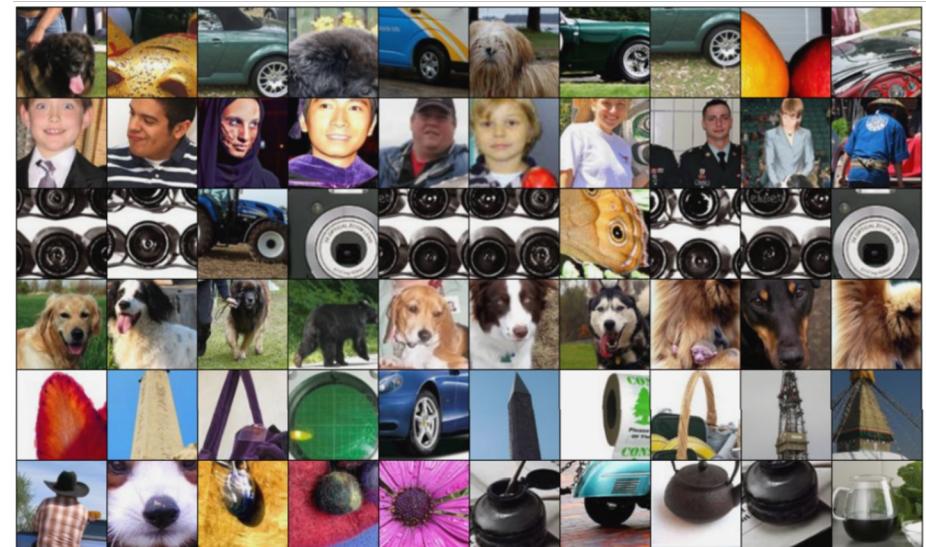
Units in Intermediate Layers: Maximally Activating Patches



Pick a layer and a channel; e.g. conv5 is $128 \times 13 \times 13$, pick channel 17/128

Run many images through the network,
record values of chosen channel

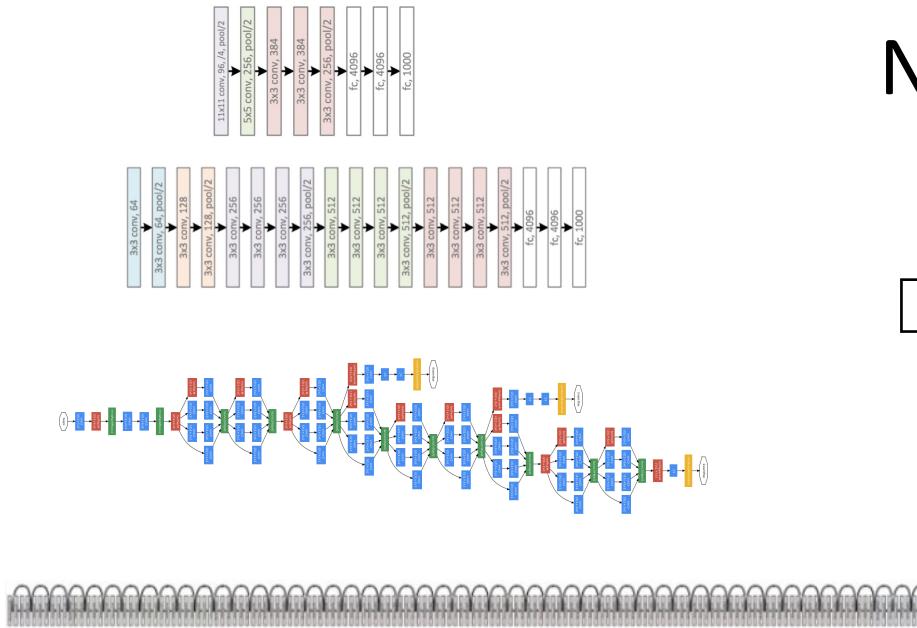
Visualize image patches that correspond to
maximal activations



Springenberg et al, "Striving for Simplicity: The All Convolutional Net", ICLR Workshop 2015
Figure copyright Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller, 2015; reproduced with permission.

Quantify the Interpretability of Networks

Network Dissection



Layer5 unit 79

car (object)

IoU=0.13



Layer5 unit 107

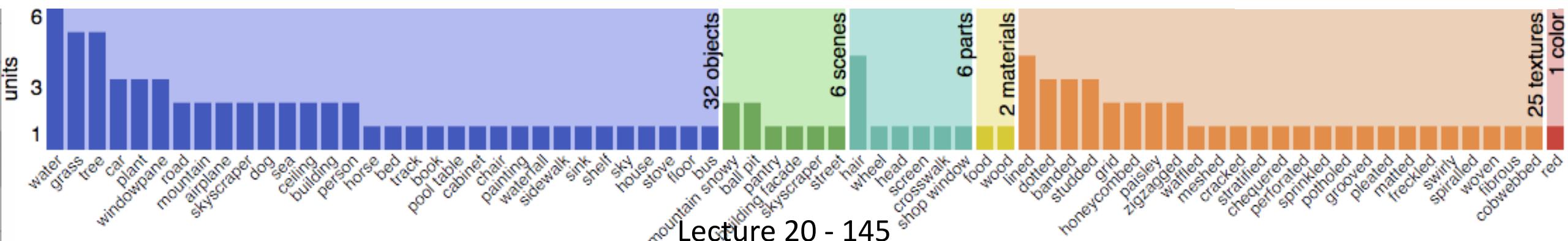
road (object)

IoU=0.15



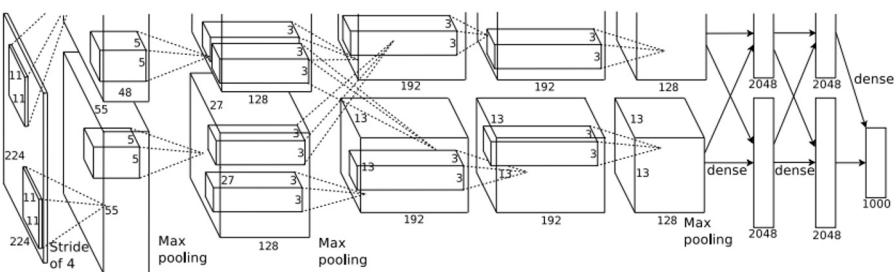
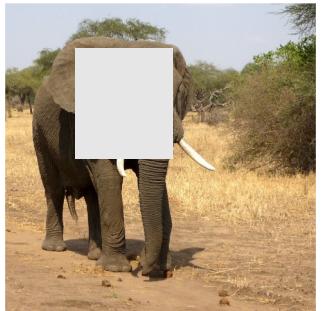
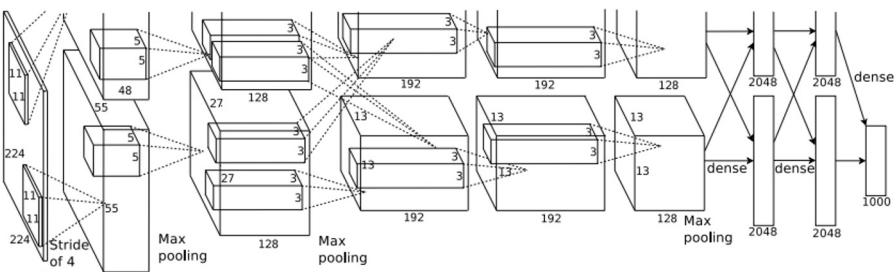
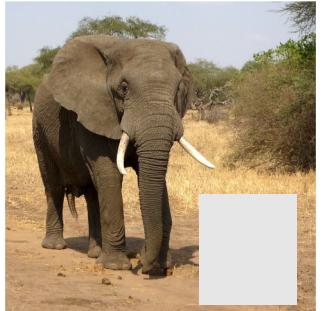
118/256 units covering 72 unique concepts

places
THE SCENE RECOGNITION DATABASE

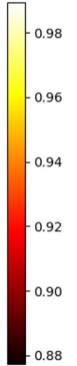
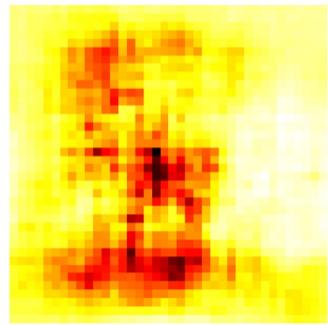


Which Pixels Matter? Saliency via Occlusion

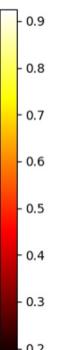
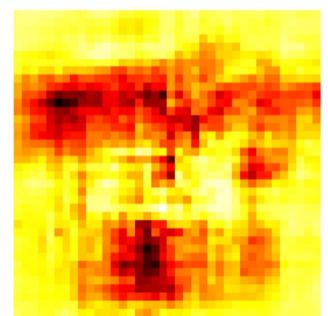
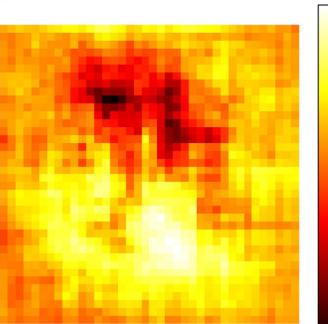
Mask part of the image before feeding to CNN,
check how much predicted probabilities change



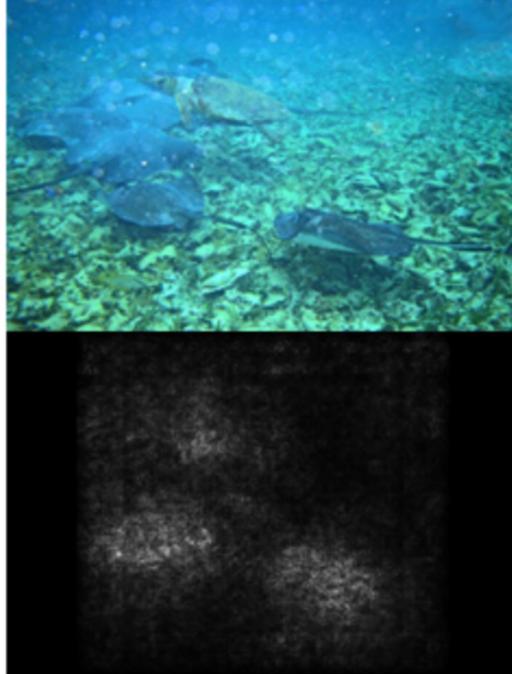
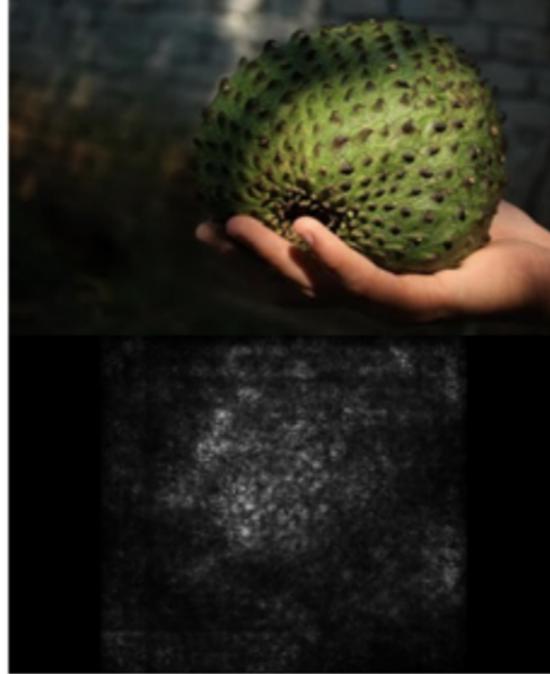
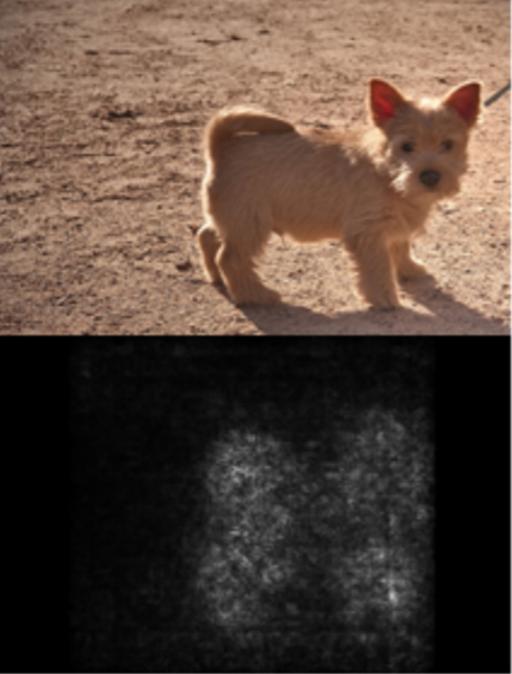
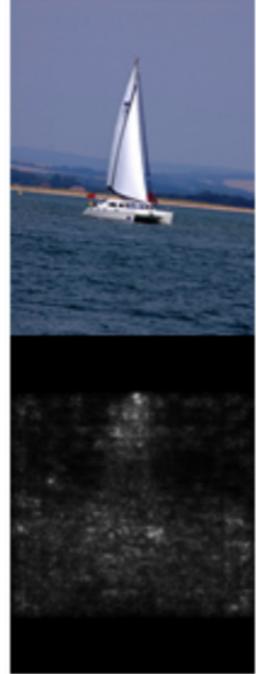
[Boat image](#) is CCO public domain
[Elephant image](#) is CCO public domain
[Go-Karts image](#) is CCO public domain



African elephant, *Loxodonta africana*

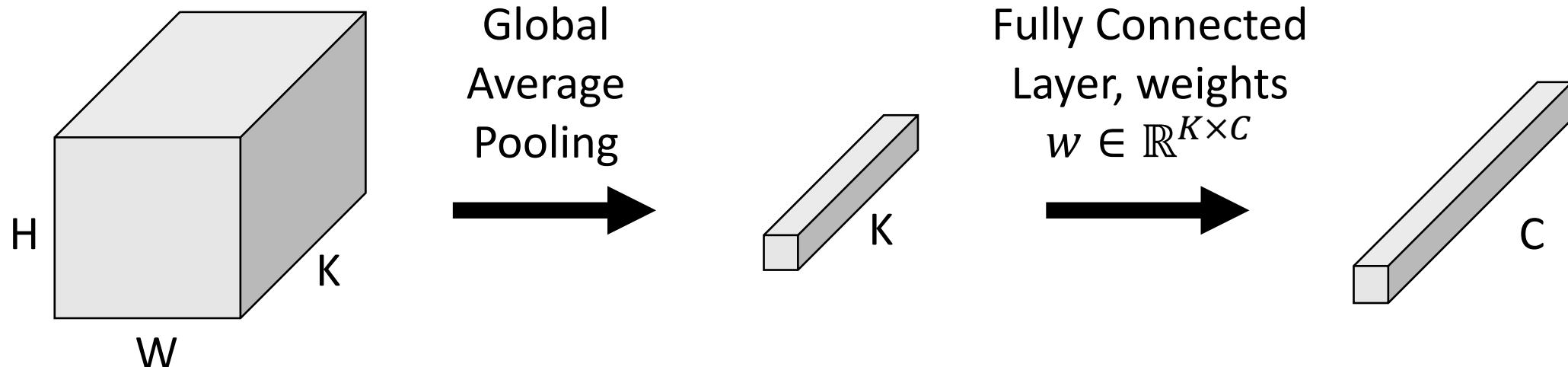


Which pixels matter? Saliency via Backprop



Simonyan, Vedaldi, and Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.
Figures copyright Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman, 2014; reproduced with permission.

Class Activation Mapping (CAM)



Last layer CNN features:

$$f \in \mathbb{R}^{H \times W \times K}$$

Pooled features:

$$F \in \mathbb{R}^K$$

Class Scores:

$$S \in \mathbb{R}^C$$

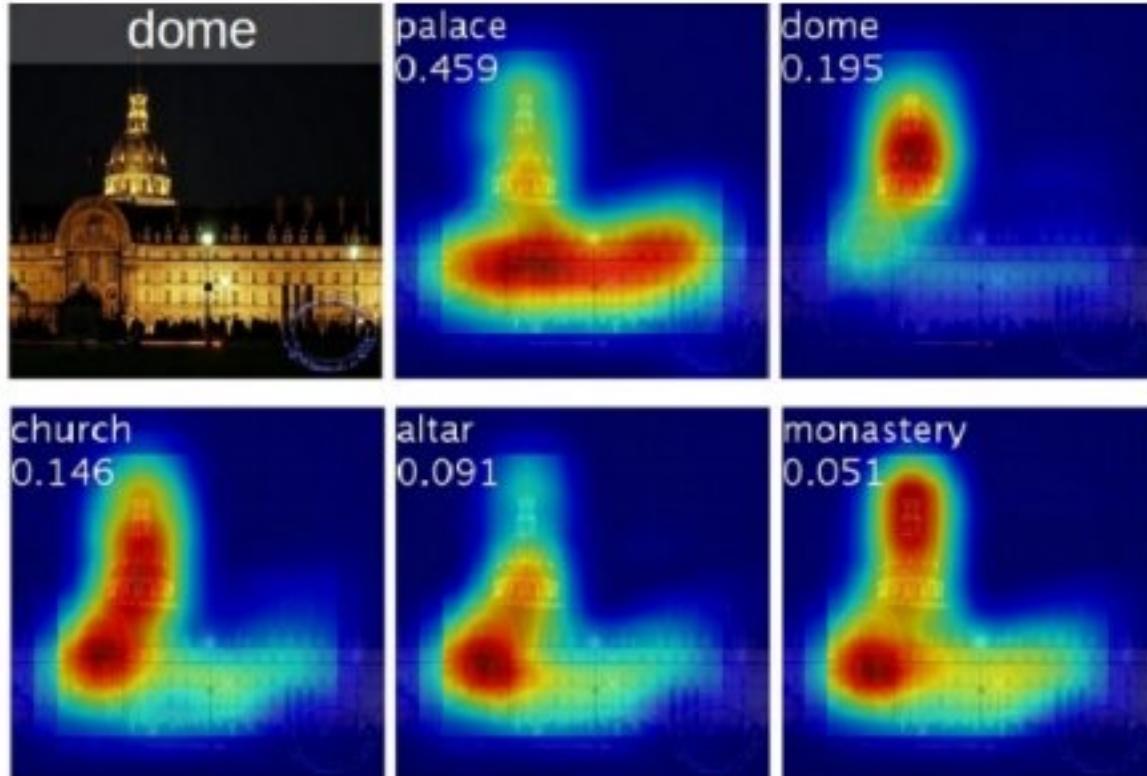
$$\begin{aligned} F_k &= \frac{1}{HW} \sum_{h,w} f_{h,w,k} & S_c &= \sum_k w_{k,c} F_k = \frac{1}{HW} \sum_k w_{k,c} \sum_{h,w} f_{h,w,k} \\ &&&= \frac{1}{HW} \sum_{h,w} \sum_k w_{k,c} f_{h,w,k} \end{aligned}$$

Class Activation Maps:

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

Zhou et al, "Learning Deep Features for Discriminative Localization", CVPR 2016

Class Activation Mapping (CAM)



Class activation maps of top 5 predictions

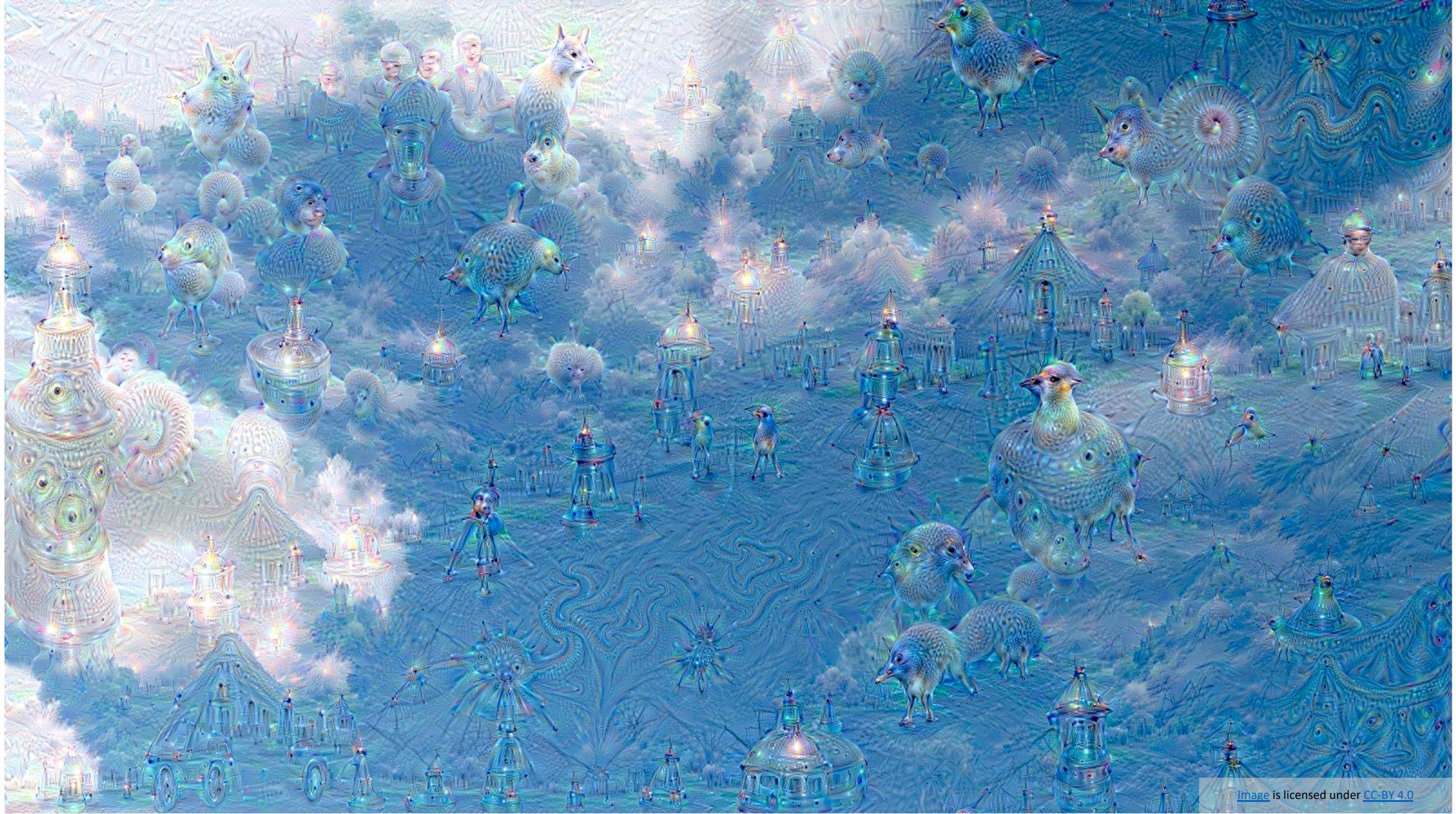


Class activation maps for one object class

Visualizing CNN Features: Gradient Ascent



Nguyen et al, "Multifaceted Feature Visualization: Uncovering the Different Types of Features Learned By Each Neuron in Deep Neural Networks", ICML Visualization for Deep Learning Workshop 2016.



[Image](#) is licensed under [CC-BY 4.0](#)

Adversarial Examples

African elephant



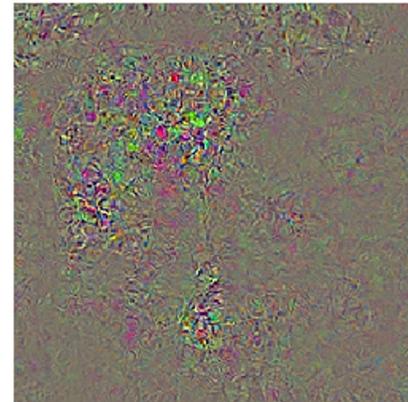
koala



Difference



10x Difference



schooner



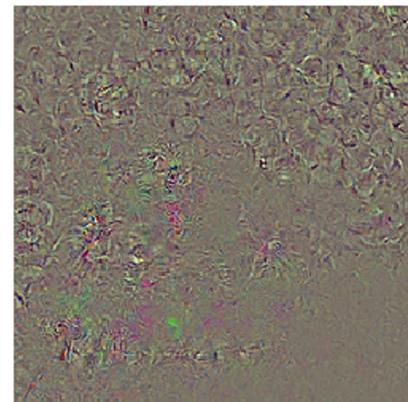
iPod



Difference



10x Difference

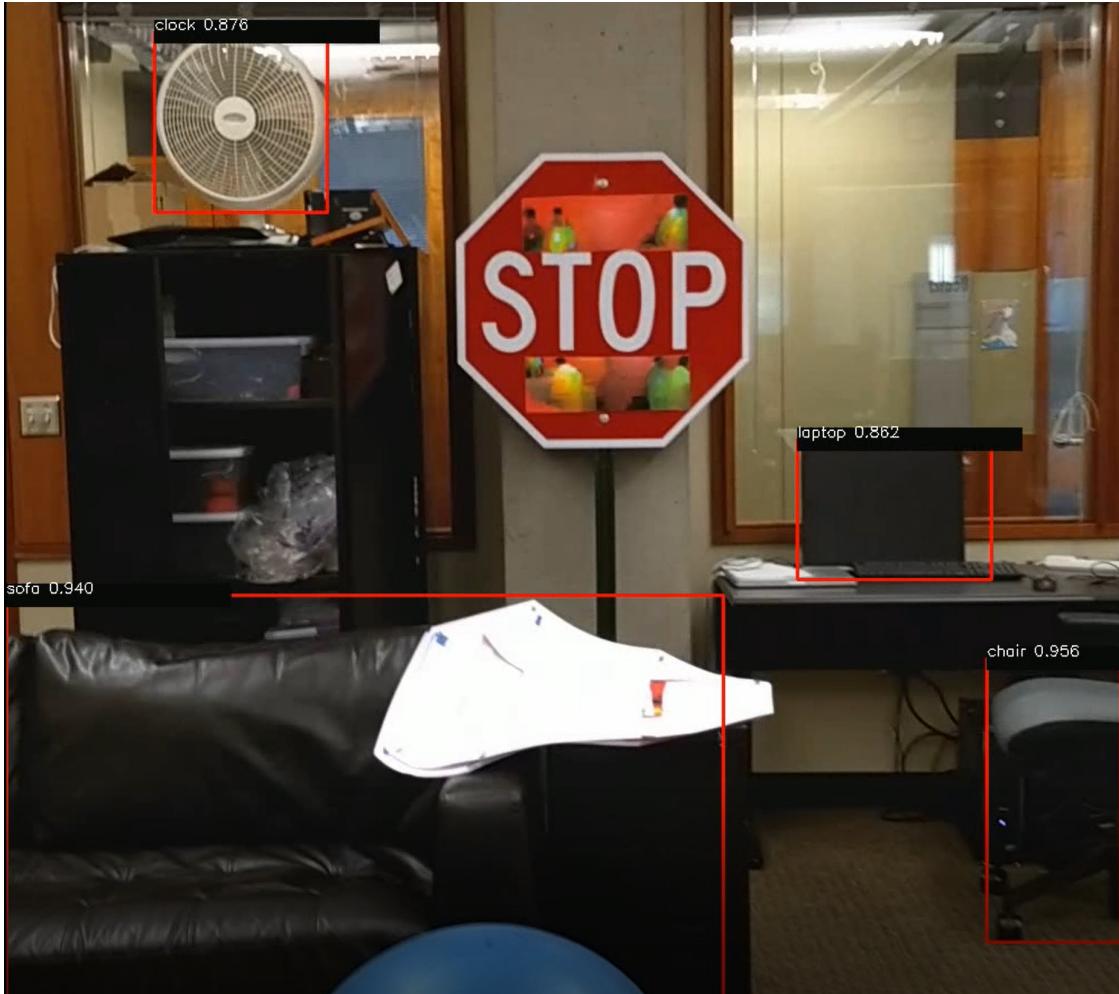


Boat image is [CC0 public domain](#)

Elephant image is [CC0 public domain](#)

Adversarial Machine Learning becomes a big thing

Adversarial examples for object detectors



Lecture 11, 12: RNN, Attention, Transformers, ViT

Sequence-to-Sequence with RNNs

Input: Sequence x_1, \dots, x_T

Output: Sequence $y_1, \dots, y_{T'}$

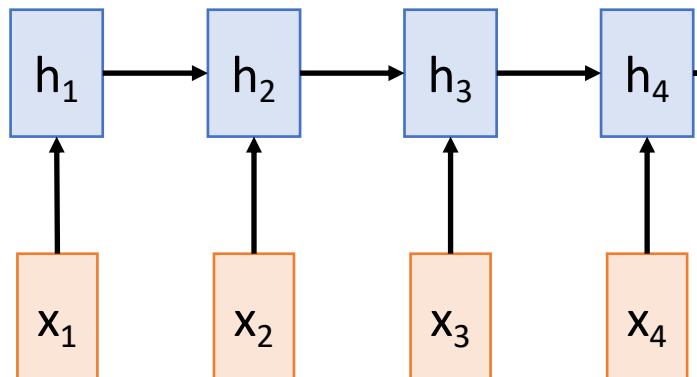
Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$

Encoder: $h_t = f_W(x_t, h_{t-1})$

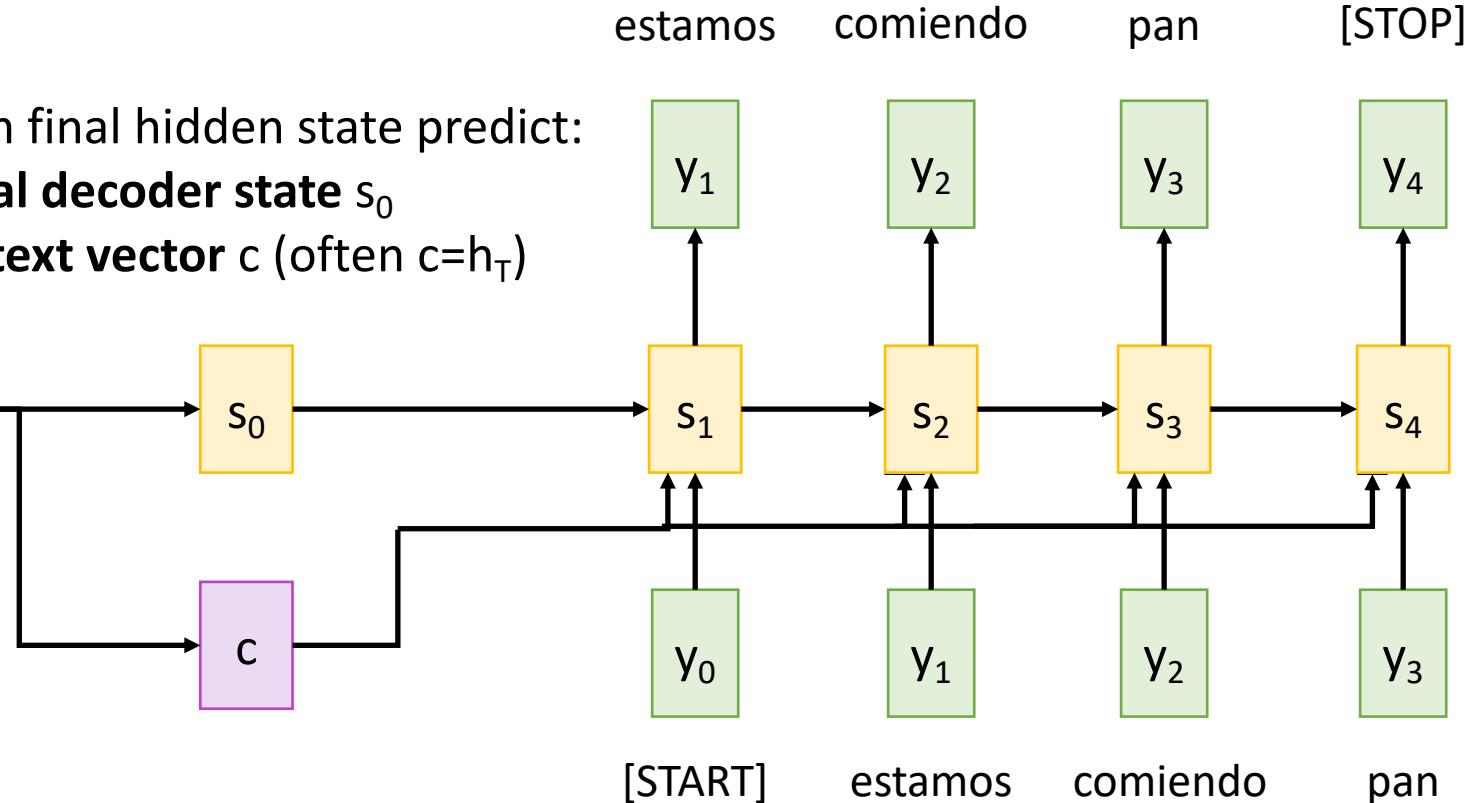
From final hidden state predict:

Initial decoder state s_0

Context vector c (often $c=h_T$)

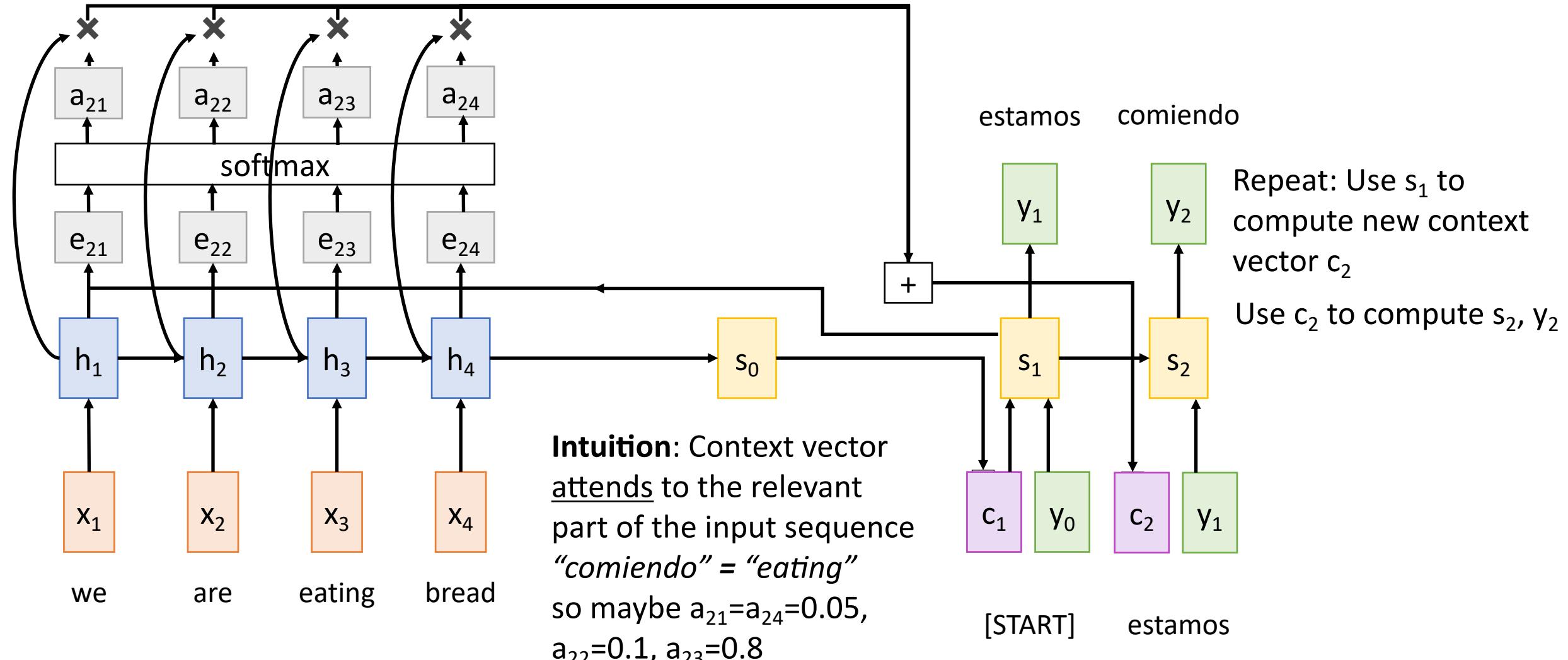


we are eating bread



[START] estamos comiendo pan

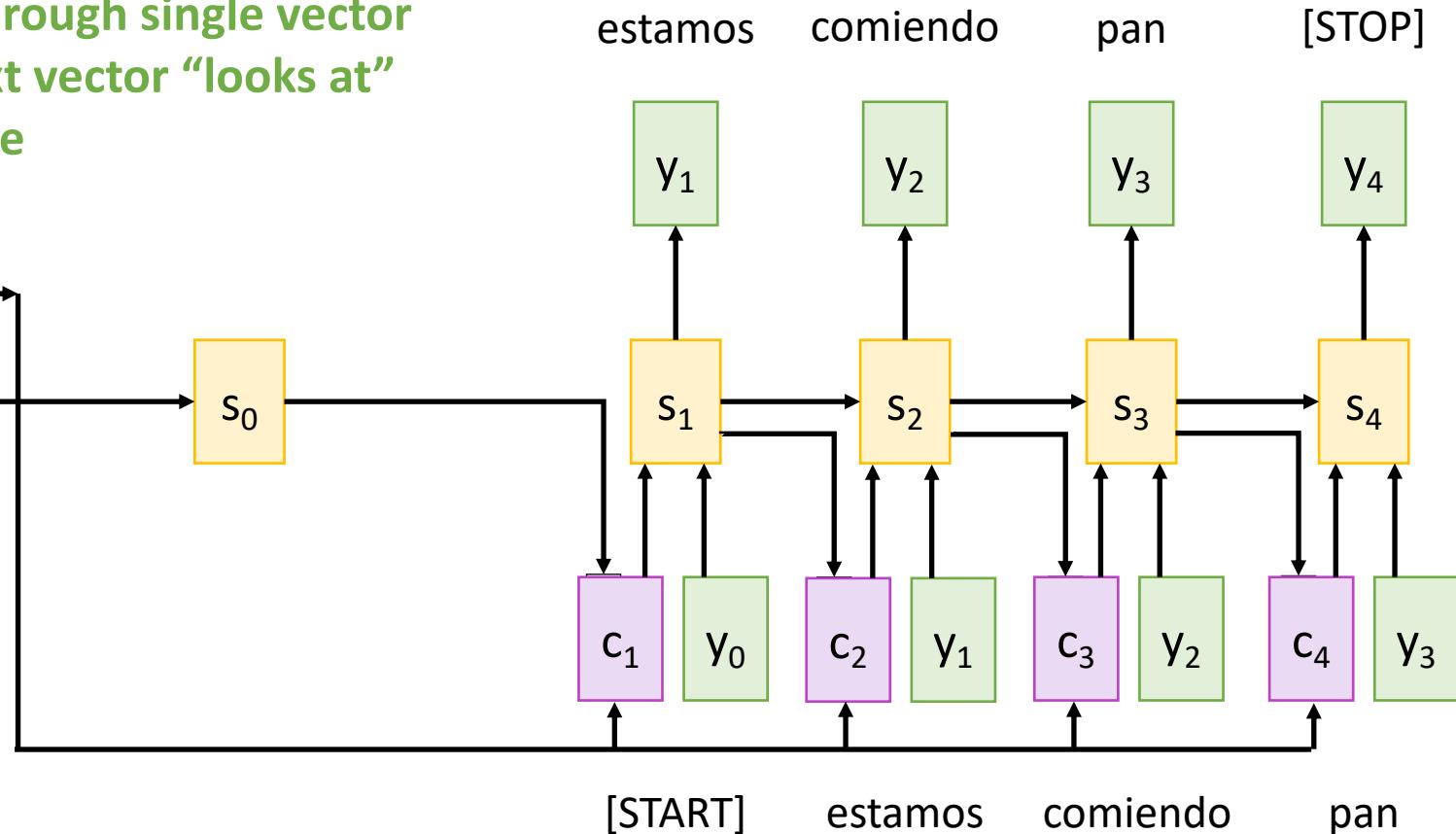
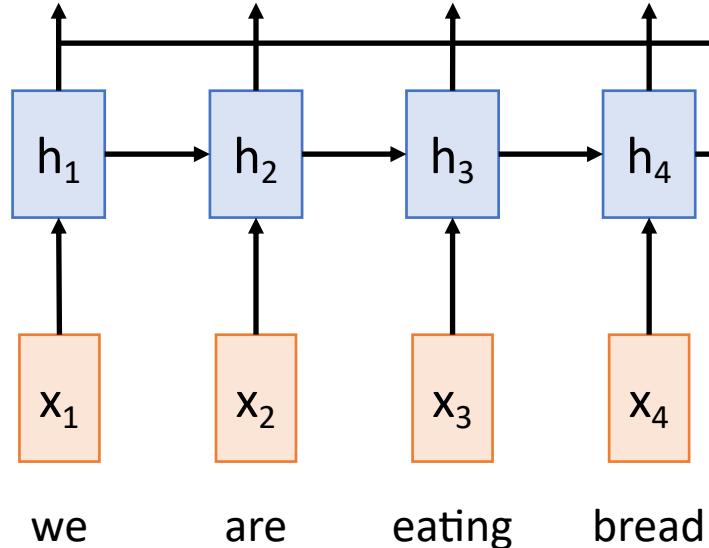
Sequence-to-Sequence with RNNs and Attention



Sequence-to-Sequence with RNNs and Attention

Use a different context vector in each timestep of decoder

- Input sequence not bottlenecked through single vector
- At each timestep of decoder, context vector “looks at” different parts of the input sequence



Attention Layer

Inputs:

Query vectors: \mathbf{Q} (Shape: $N_Q \times D_Q$)

Input vectors: \mathbf{X} (Shape: $N_X \times D_X$)

Key matrix: \mathbf{W}_K (Shape: $D_X \times D_Q$)

Value matrix: \mathbf{W}_V (Shape: $D_X \times D_V$)

Computation:

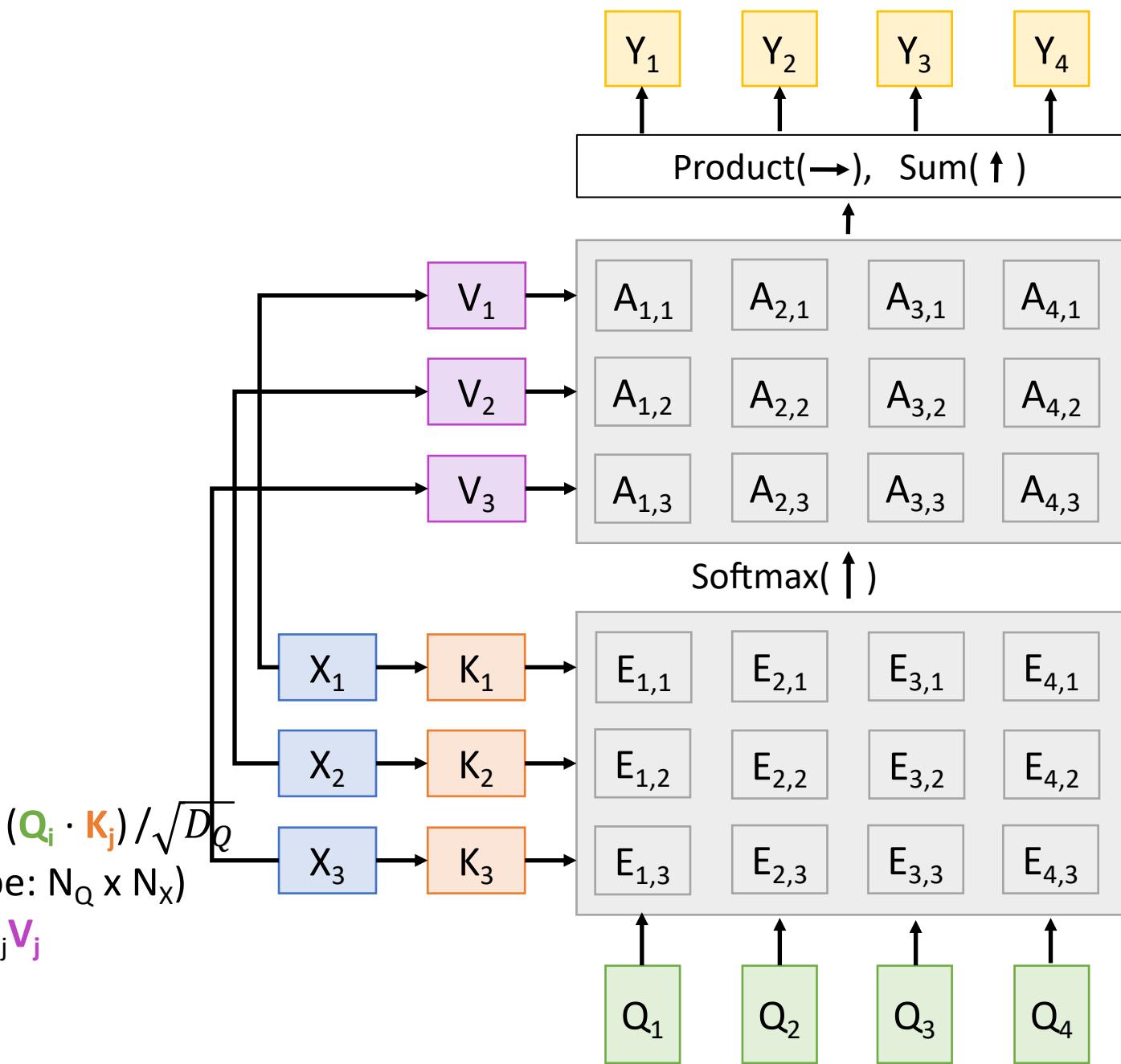
Key vectors: $\mathbf{K} = \mathbf{X}\mathbf{W}_K$ (Shape: $N_X \times D_Q$)

Value Vectors: $\mathbf{V} = \mathbf{X}\mathbf{W}_V$ (Shape: $N_X \times D_V$)

Similarities: $E = \mathbf{Q}\mathbf{K}^T / \sqrt{D_Q}$ (Shape: $N_Q \times N_X$) $E_{i,j} = (\mathbf{Q}_i \cdot \mathbf{K}_j) / \sqrt{D_Q}$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_Q \times N_X$)

Output vectors: $\mathbf{Y} = A\mathbf{V}$ (Shape: $N_Q \times D_V$) $Y_i = \sum_j A_{i,j} \mathbf{V}_j$



Self-Attention Layer

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

Inputs:

Input vectors: X (Shape: $N_x \times D_x$)

Key matrix: W_K (Shape: $D_x \times D_Q$)

Value matrix: W_V (Shape: $D_x \times D_V$)

Query matrix: W_Q (Shape: $D_x \times D_Q$)

Computation:

Query vectors: $Q = XW_Q$

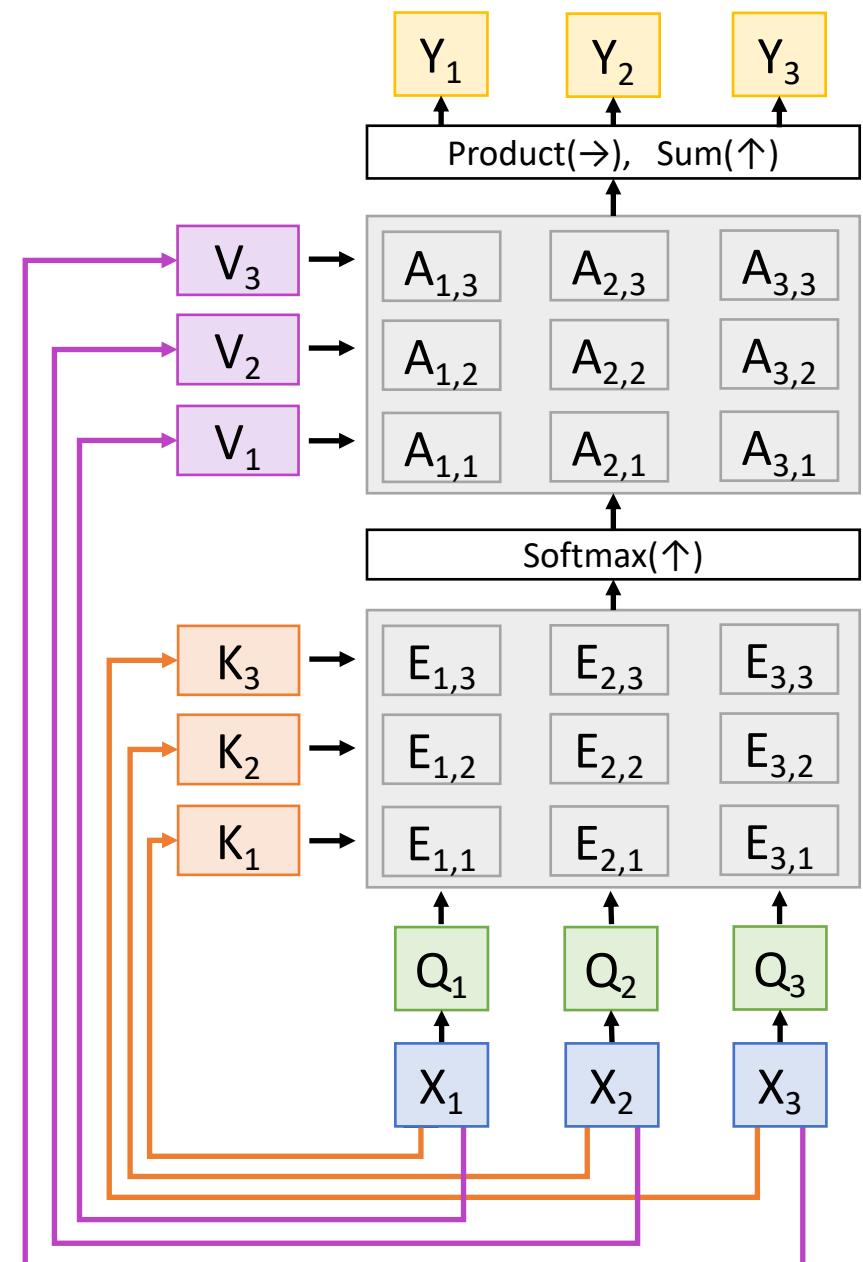
Key vectors: $K = XW_K$ (Shape: $N_x \times D_Q$)

Value Vectors: $V = XW_V$ (Shape: $N_x \times D_V$)

Similarities: $E = QK^T / \sqrt{D_Q}$ (Shape: $N_x \times N_x$) $E_{i,j} = (Q_i \cdot K_j) / \sqrt{D_Q}$

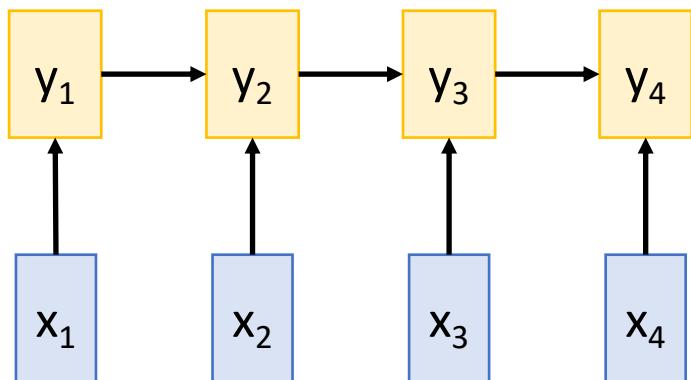
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_x \times N_x$)

Output vectors: $Y = A V$ (Shape: $N_x \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

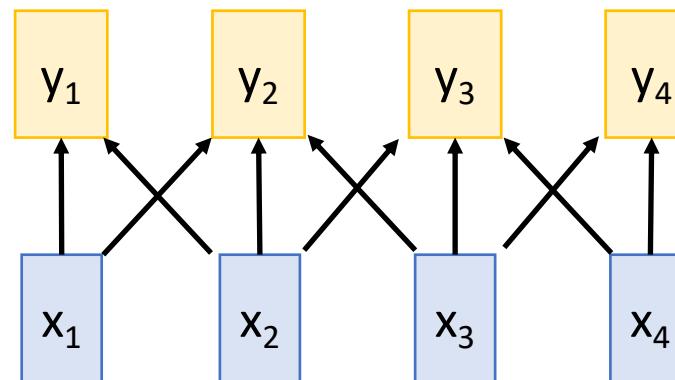


Three Ways of Processing Sequences

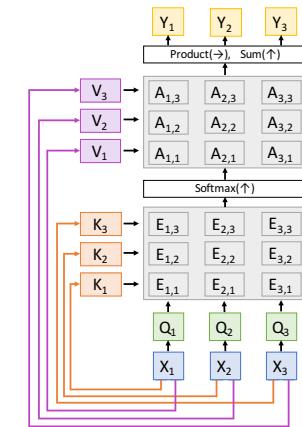
Recurrent Neural Network



1D Convolution



Self-Attention



Works on **Ordered Sequences**

- (+) Good at **long sequences**: After one RNN layer, h_T "sees" the whole sequence
- (-) Not parallelizable: need to compute hidden states sequentially

Works on **Multidimensional Grids**

- (-) Bad at **long sequences**: Need to stack many conv layers for outputs to "see" the whole sequence
- (+) Highly parallel: Each output can be computed in parallel

Works on **Sets of Vectors**

- (-) Good at **long sequences**: after one self-attention layer, each output "sees" all inputs!
- (+) Highly parallel: Each output can be computed in parallel
- (-) Very memory intensive

The Transformer

Transformer Block:

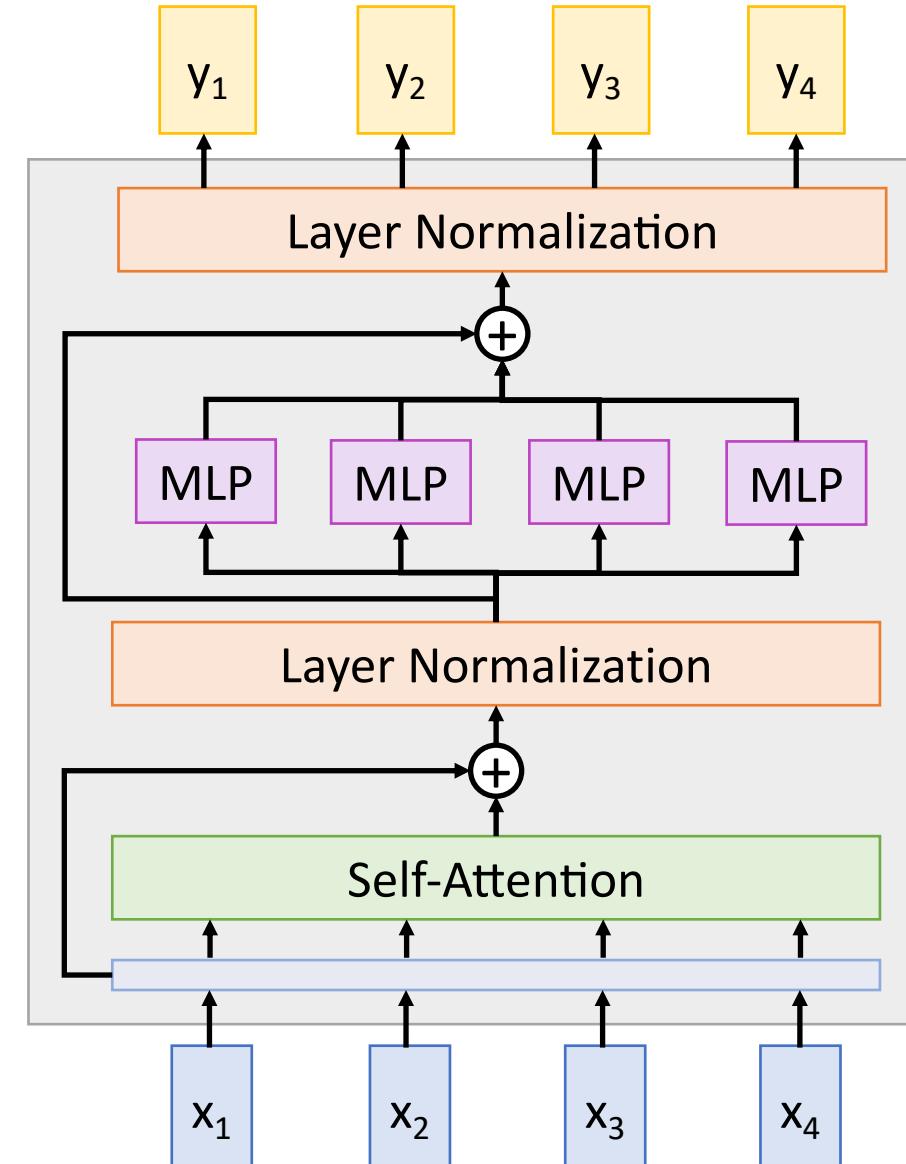
Input: Set of vectors x

Output: Set of vectors y

Self-attention is the only
interaction between vectors!

Layer norm and MLP work
independently per vector

Highly scalable, highly
parallelizable



The Transformer

Transformer Block:

Input: Set of vectors x

Output: Set of vectors y

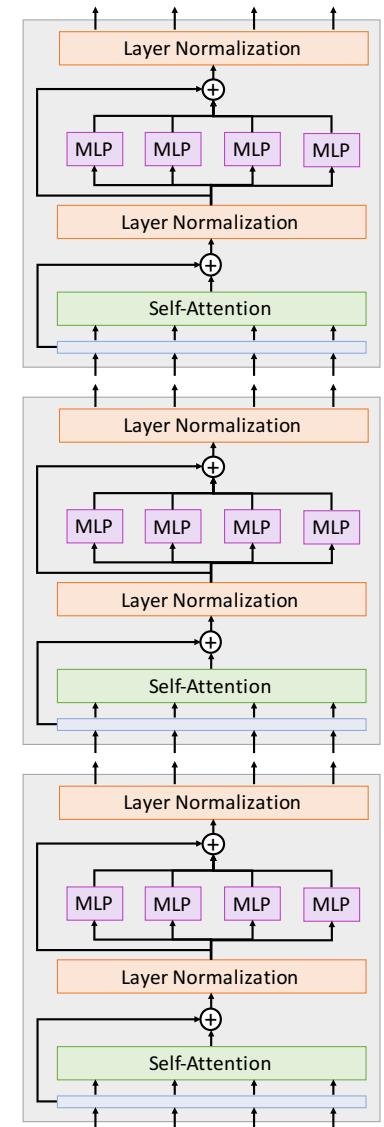
Self-attention is the only interaction between vectors!

Layer norm and MLP work independently per vector

Highly scalable, highly parallelizable

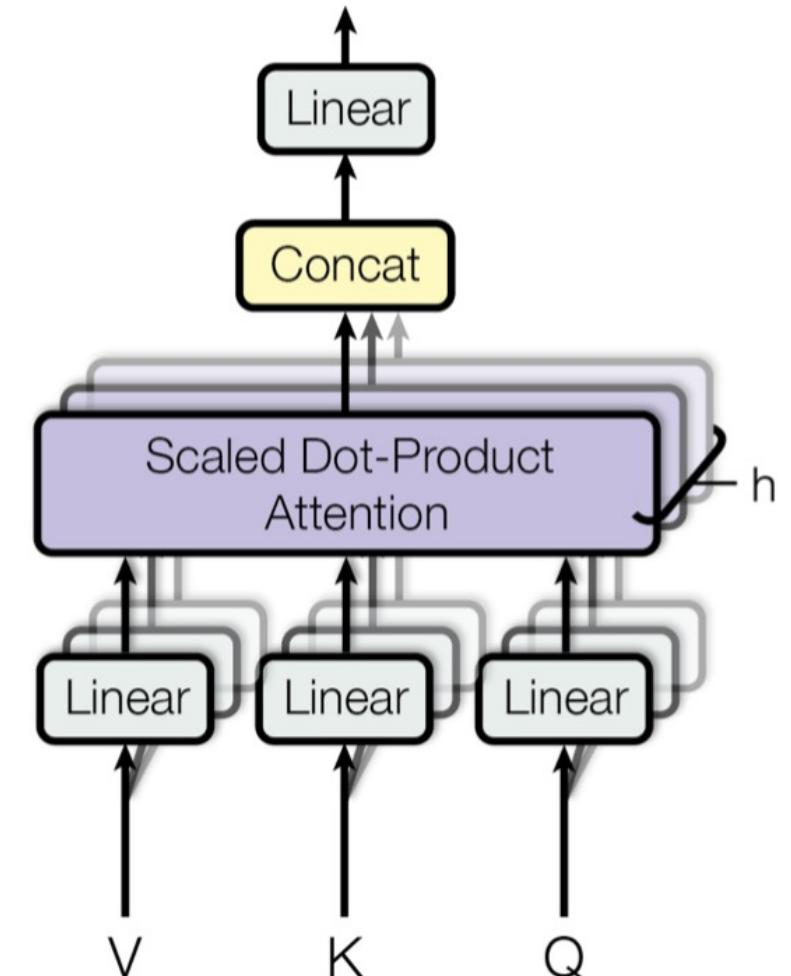
A **Transformer** is a sequence of transformer blocks

Vaswani et al:
12 blocks, $D_Q=512$, 6 heads



Multi-Head Attention

Multi-head Attention is a module for attention mechanisms which runs through an attention mechanism several times in parallel. The independent attention outputs are then concatenated and linearly transformed into the expected dimension.



Typically, if the dimension of the inputs X is D and there are H heads, the values, queries, and keys will all be of size D/H , as this allows for an efficient implementation.

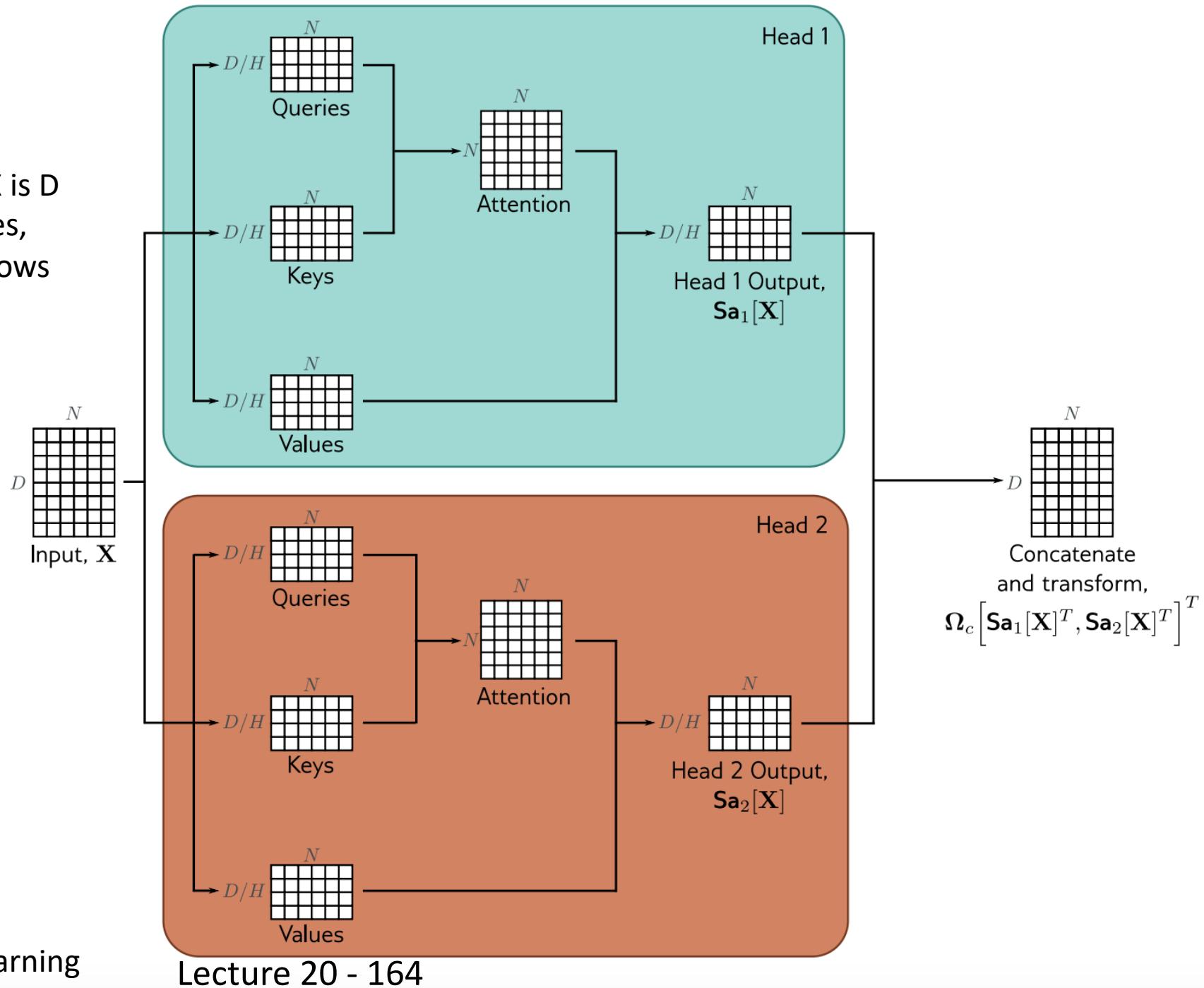
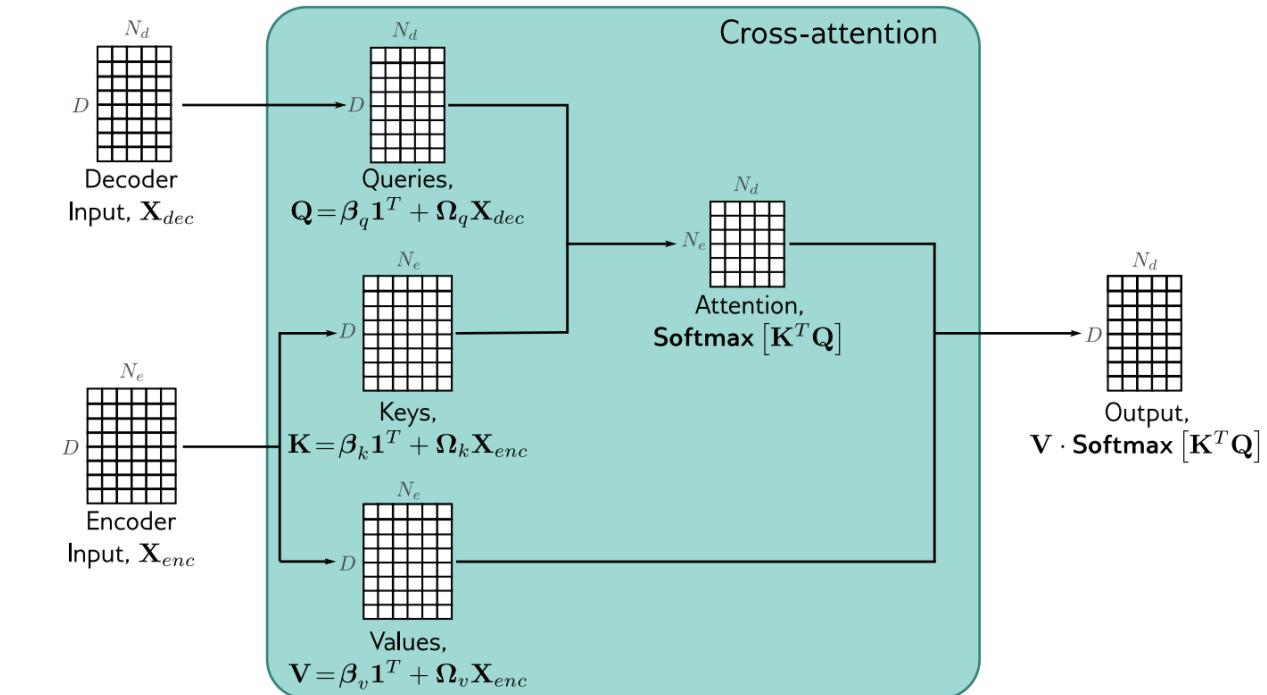
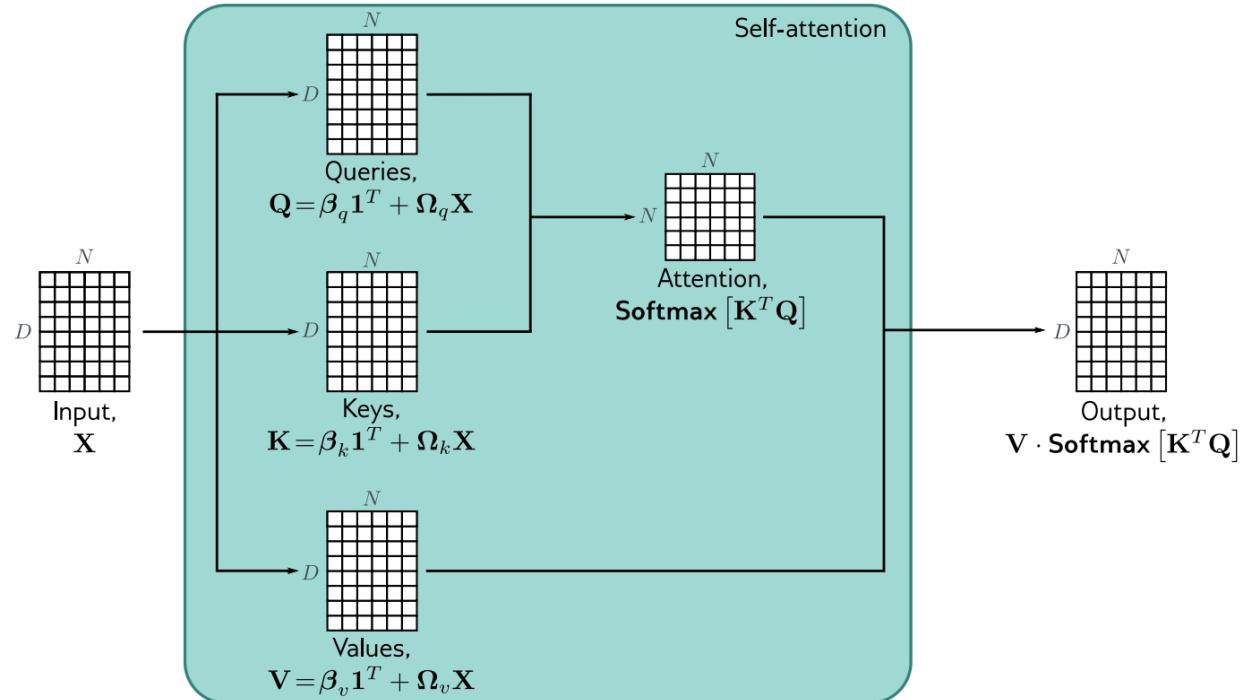


Figure 12.6 from Understanding Deep Learning

Self-Attention vs. Cross-Attention



the queries are calculated from the decoder embeddings X_{dec} , and the keys and values from the encoder embeddings X_{enc} . In the context of translation, the encoder contains information about the source language, and the decoder contains information about the target language statistics.

Vision Transformer (ViT)

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

Output vectors



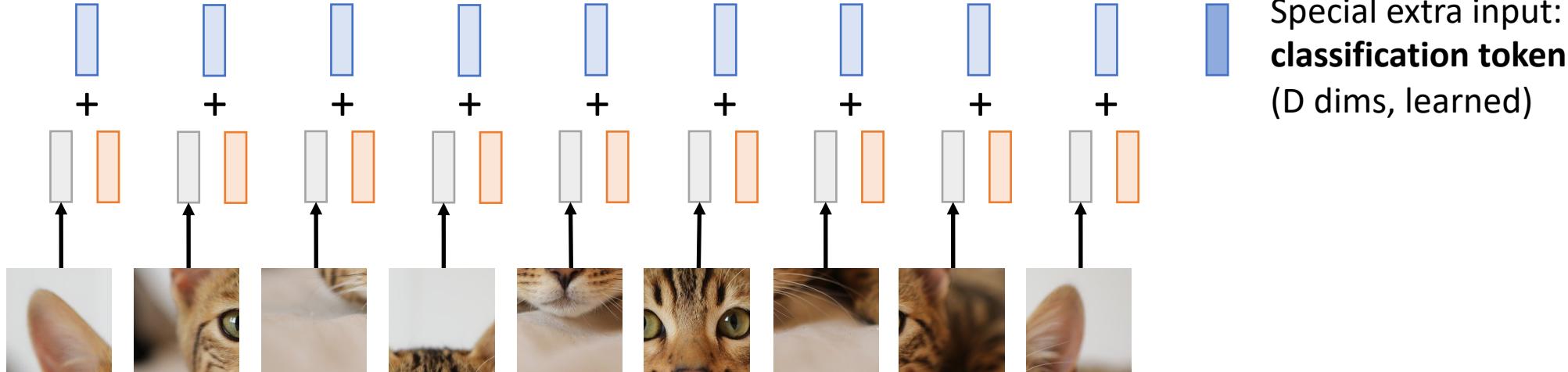
Each attention matrix has $14^4 = 38,416$ entries, takes 150 KB (or 65,536 entries, takes 256 KB)

Linear projection to C-dim vector of predicted class scores

Exact same as NLP Transformer!

Transformer

Add positional embedding: learned D-dim vector per position



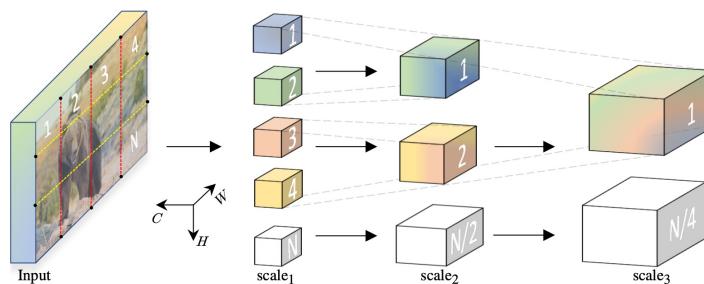
N input patches, each of shape 3x16x16

Cat image is free for commercial use under a [Pixabay license](#)

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

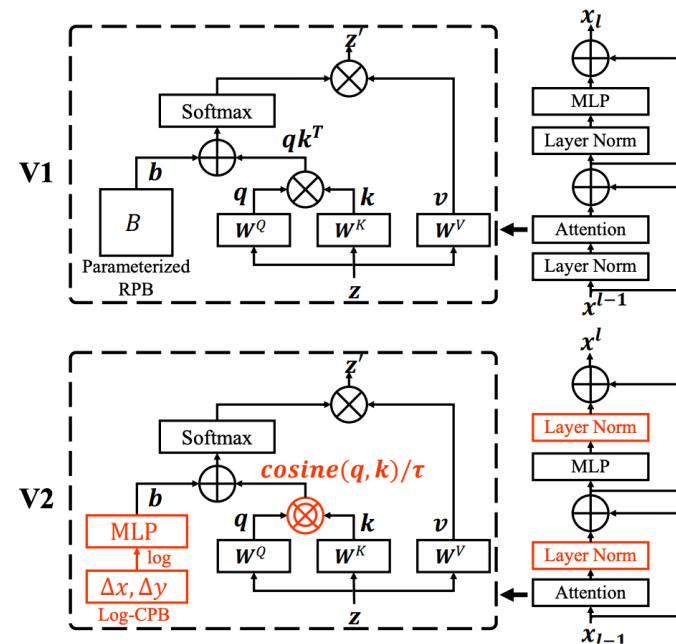
Hierarchical Vision Transformers

MViT



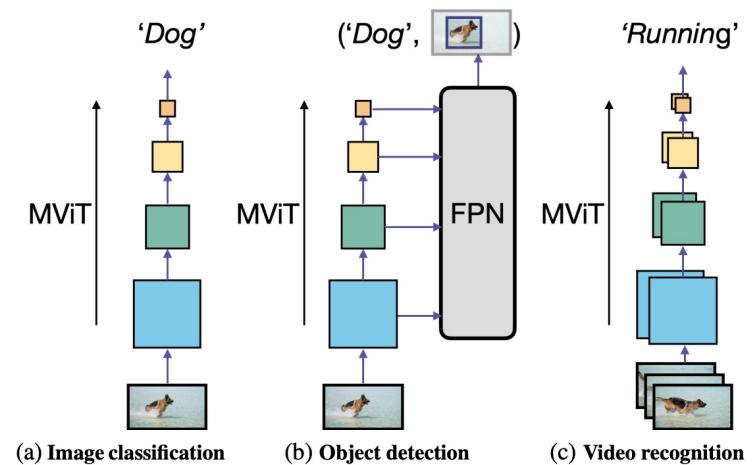
Fan et al, "Multiscale Vision Transformers", ICCV 2021

Swin, Swin-V2



Liu et al, "Swin Transformer V2: Scaling up Capacity and Resolution", CVPR 2022

Improved MViT



Li et al, "Improved Multiscale Vision Transformers for Classification and Detection", arXiv 2021

Lecture 13 and 14: Object Detection and Dense Prediction

Computer Vision Tasks

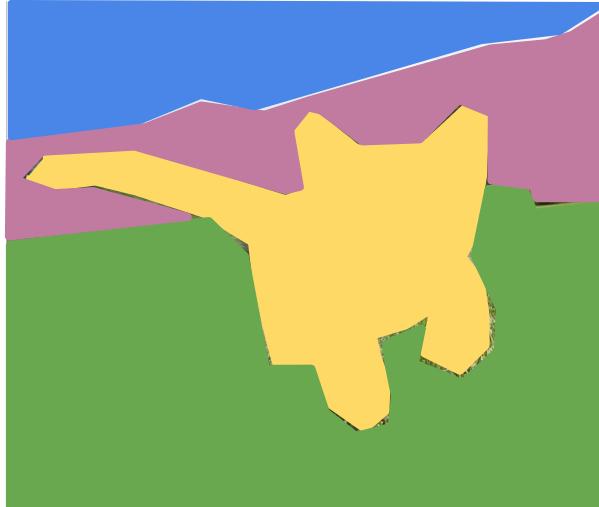
Classification



CAT

No spatial extent

Semantic Segmentation



GRASS, CAT, TREE,
SKY

No objects, just pixels

Object Detection



DOG, DOG, CAT

Multiple Objects

Instance Segmentation



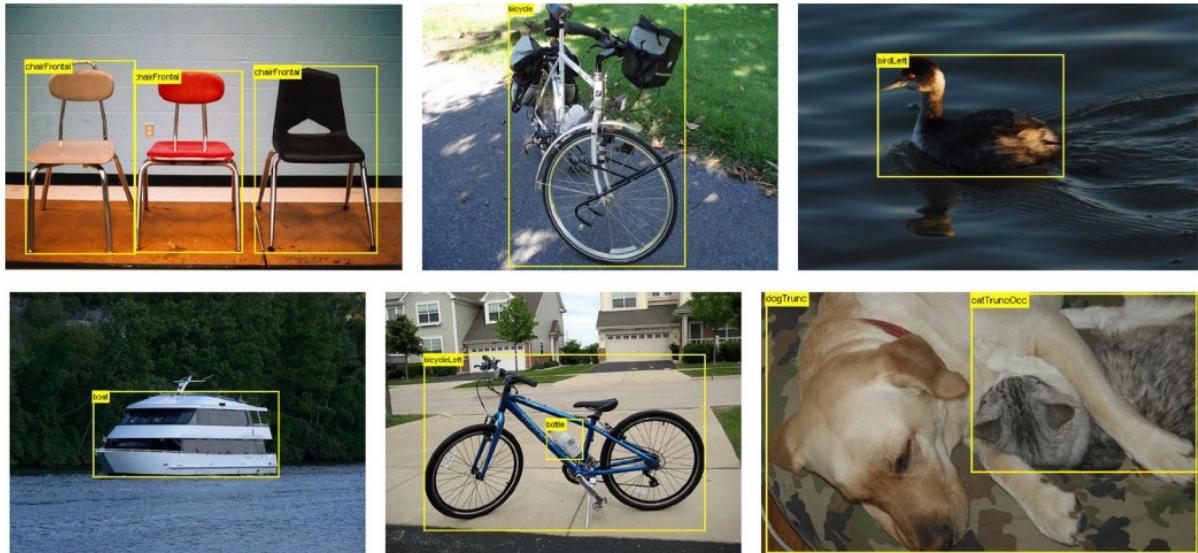
DOG, DOG, CAT

[This image is CC0 public domain](#)

Object Detection Datasets

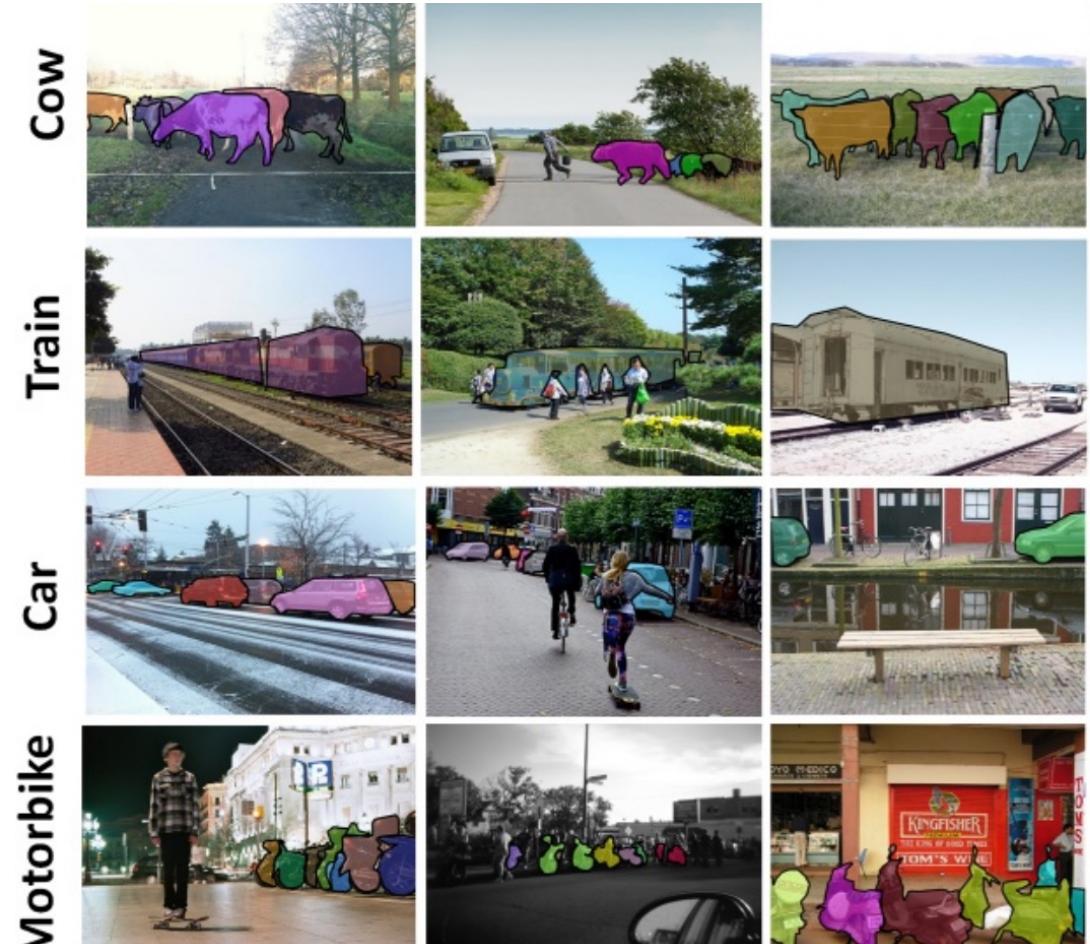
Pascal VOC object detection (20 classes, ~ 10 K images)
2005-2012

Vehicles	Household	Animals	Other
Aeroplane	Bottle	Bird	Person
Bicycle	Chair	Cat	
Boat	Dining table	Cow	
Bus	Potted plant	Dog	
Car	Sofa	Horse	
Motorbike	TV/Monitor	Sheep	
Train			



<http://host.robots.ox.ac.uk/pascal/VOC/index.html>

COCO Dataset (80 classes, 120K training images)
Annotations come with polygon masks



<https://cocodataset.org/>

Detecting Multiple Objects: Sliding Window



Apply a CNN to many different crops of the image, CNN classifies each crop as object or background

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

Consider a box of size $h \times w$:

Possible x positions: $W - w + 1$

Possible y positions: $H - h + 1$

Possible positions:

$(W - w + 1) * (H - h + 1)$

800 x 600 image
has ~58M boxes!
No way we can
evaluate them all

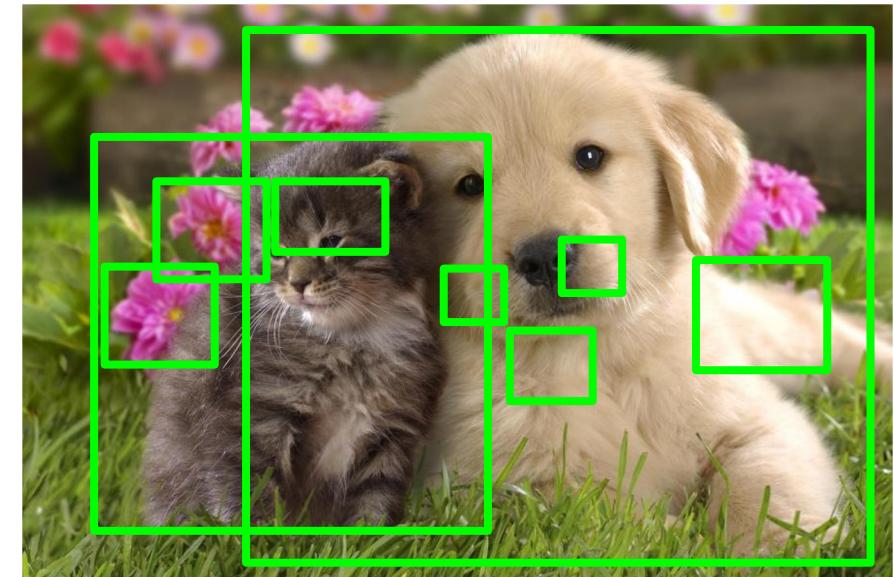
Total possible boxes:

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

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

Region Proposals

- Find a small set of boxes that are likely to cover all objects
- Often based on heuristics: e.g. look for “blob-like” image regions
- Relatively fast to run; e.g. Selective Search gives 2000 region proposals in a few seconds on CPU



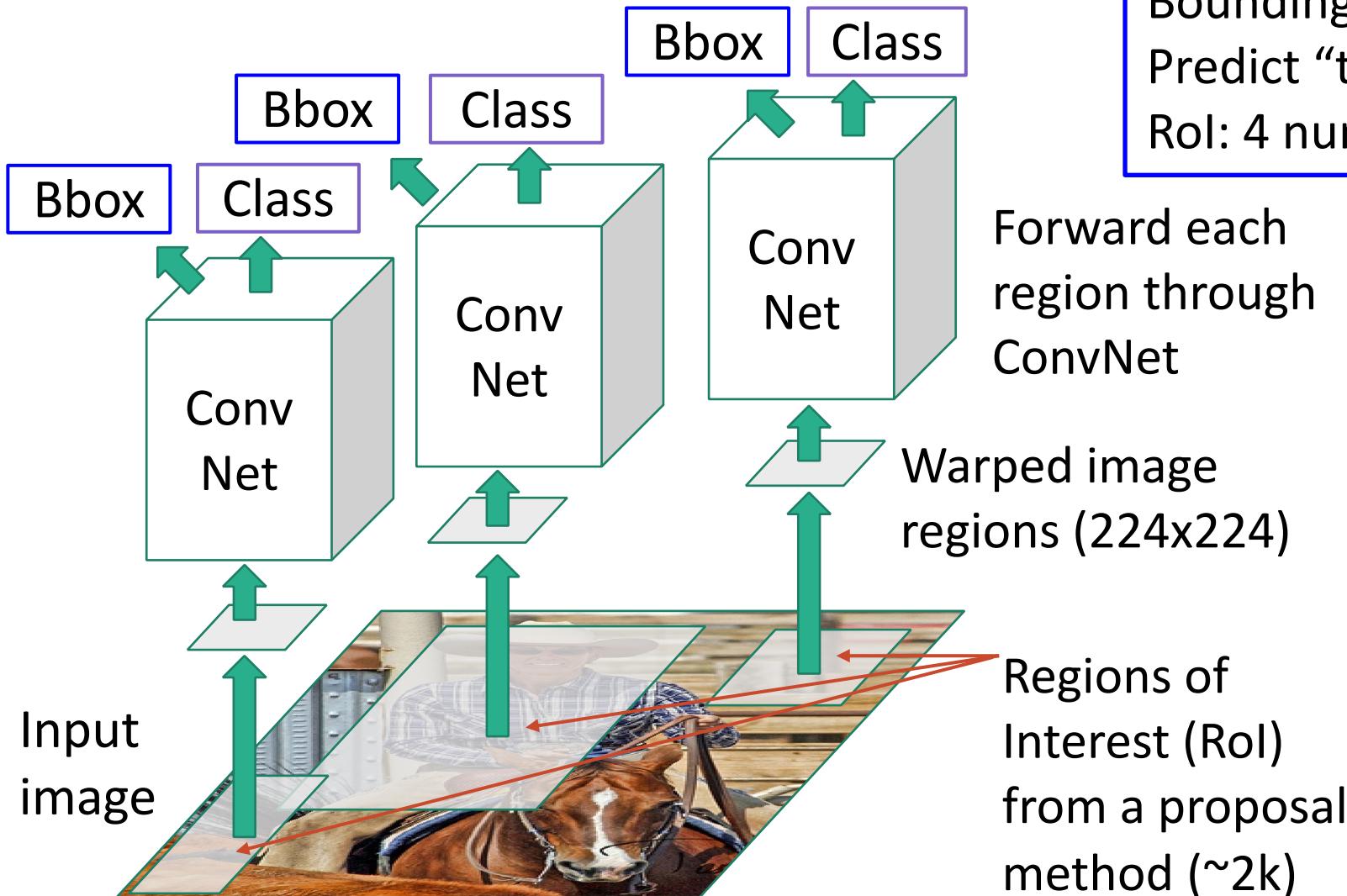
Alexe et al, “Measuring the objectness of image windows”, TPAMI 2012

Uijlings et al, “Selective Search for Object Recognition”, IJCV 2013

Cheng et al, “BING: Binarized normed gradients for objectness estimation at 300fps”, CVPR 2014

Zitnick and Dollar, “Edge boxes: Locating object proposals from edges”, ECCV 2014

R-CNN: Region-Based CNN



Classify each region

Bounding box regression:
Predict “transform” to correct the
RoI: 4 numbers (t_x, t_y, t_h, t_w)

Region proposal: (p_x, p_y, p_h, p_w)
Transform: (t_x, t_y, t_h, t_w)
Output box: (b_x, b_y, b_h, b_w)

Translate relative to box size:
 $b_x = p_x + p_w t_x \quad b_y = p_y + p_h t_y$

Log-space scale transform:
 $b_w = p_w \exp(t_w) \quad b_h = p_h \exp(t_h)$

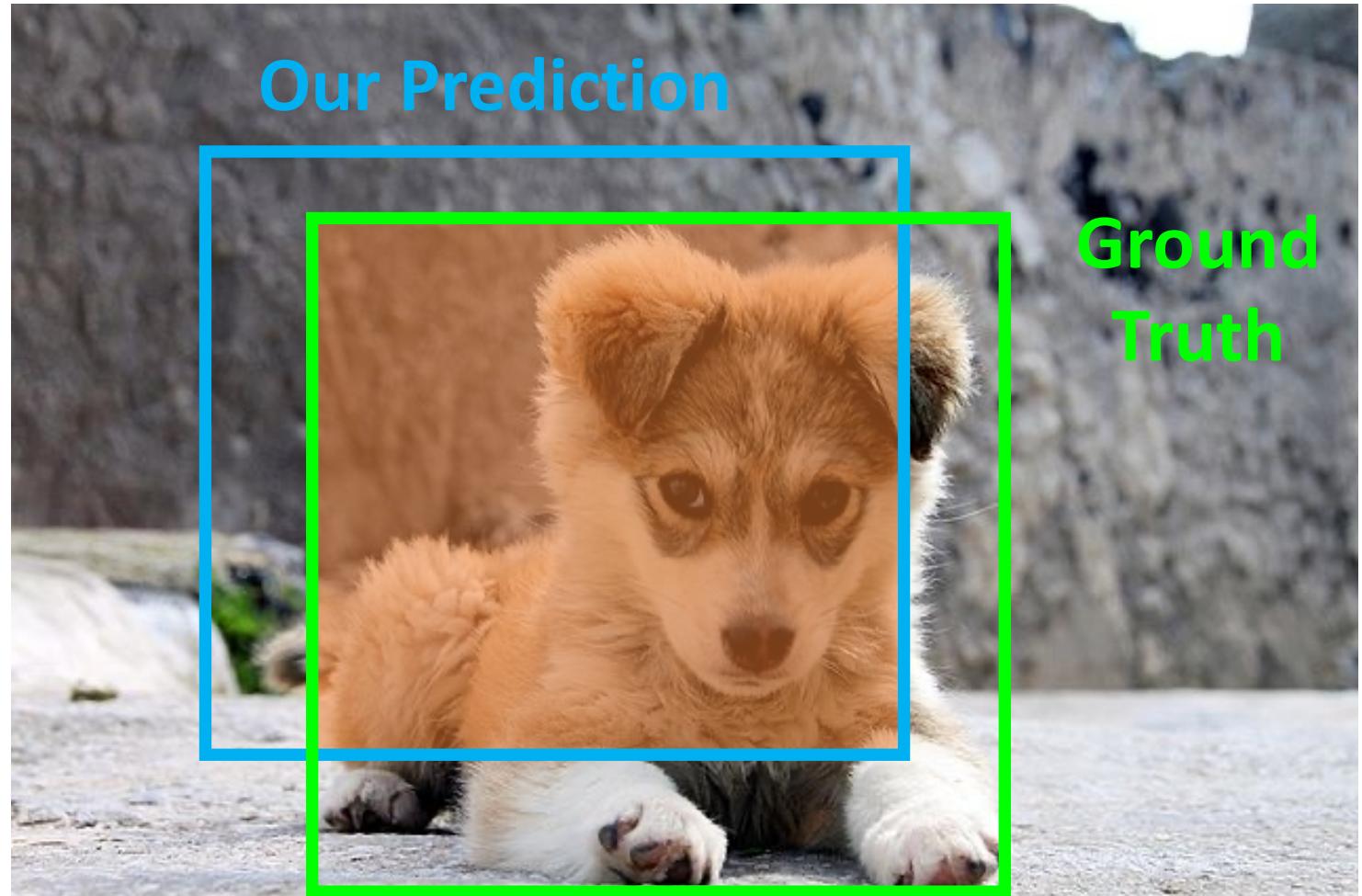
Girshick et al, “Rich feature hierarchies for accurate object detection and semantic segmentation”, CVPR 2014.
Figure copyright Ross Girshick, 2015; [source](#). Reproduced with permission.

Comparing Boxes: Intersection over Union (IoU)

How can we compare our prediction to the ground-truth box?

Intersection over Union (IoU)
(Also called “Jaccard similarity” or
“Jaccard index”):

$$\frac{\text{Area of Intersection}}{\text{Area of Union}}$$



[Puppy image](#) is licensed under [CC-A 2.0 Generic license](#). Bounding boxes and text added by Justin Johnson.

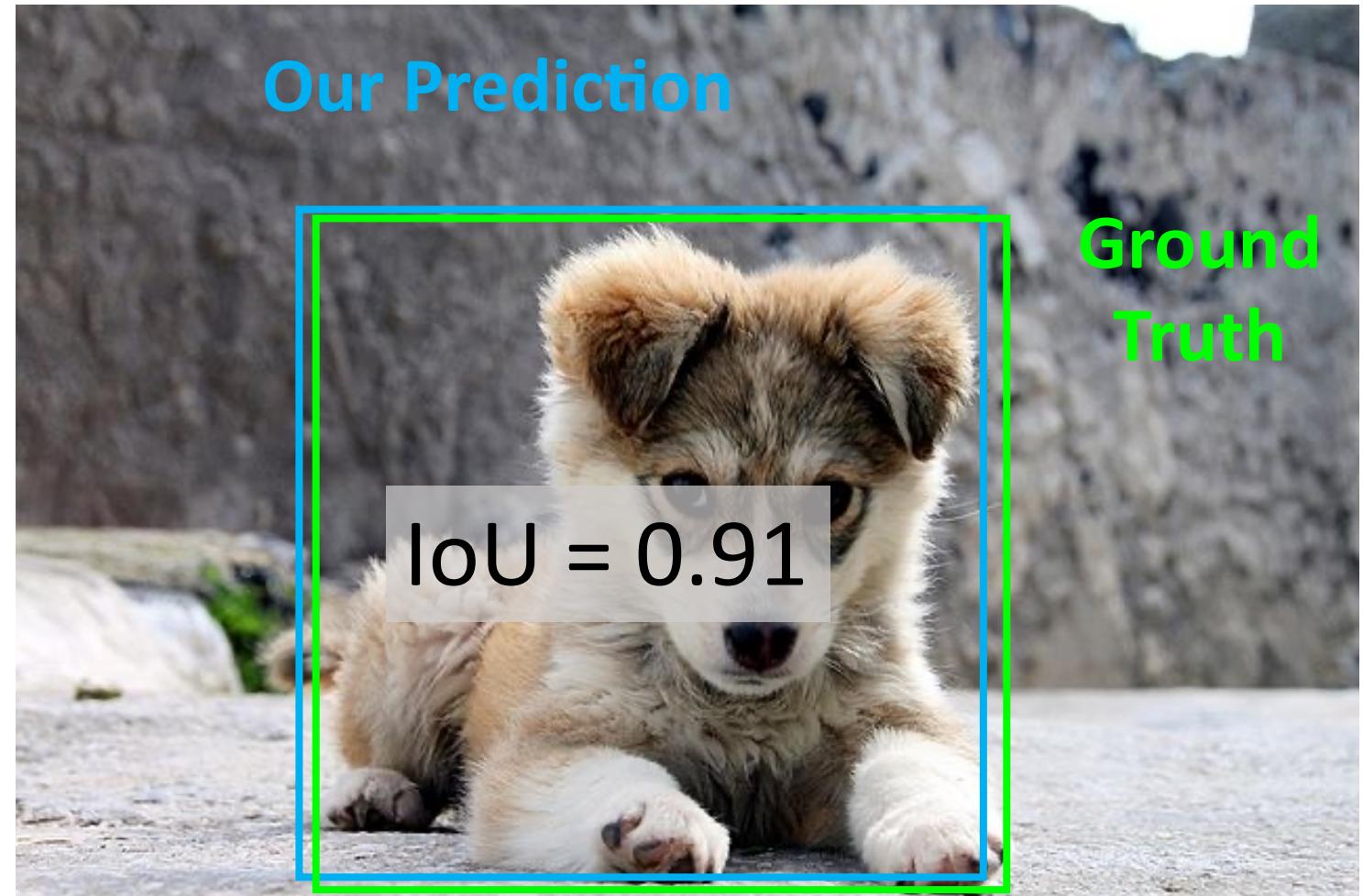
Comparing Boxes: Intersection over Union (IoU)

How can we compare our prediction to the ground-truth box?

Intersection over Union (IoU)
(Also called “Jaccard similarity” or
“Jaccard index”):

$$\frac{\text{Area of Intersection}}{\text{Area of Union}}$$

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



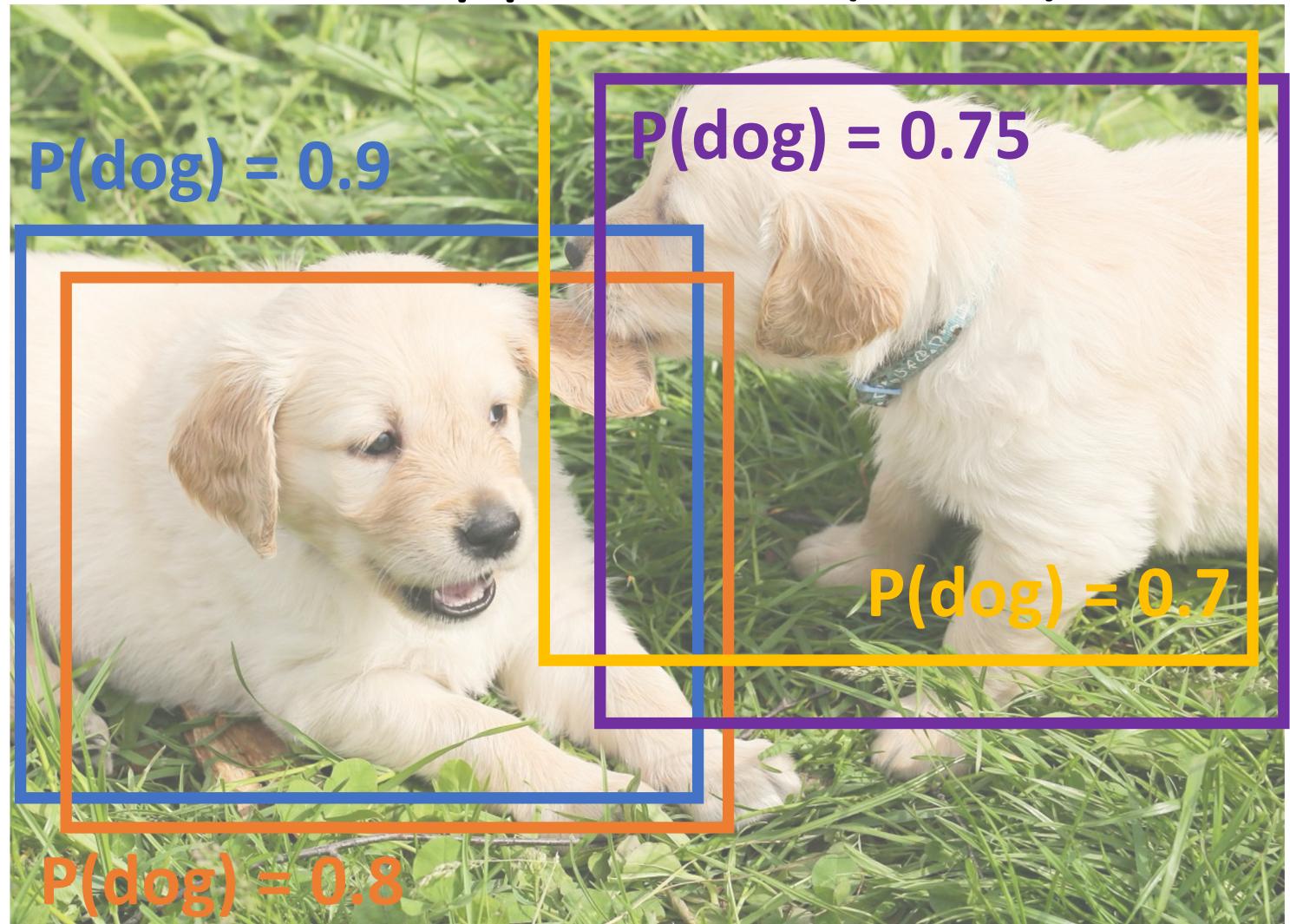
[Puppy image](#) is licensed under [CC-A 2.0 Generic license](#). Bounding boxes and text added by Justin Johnson.

Overlapping Boxes: Non-Max Suppression (NMS)

Problem: Object detectors often output many overlapping detections:

Solution: Post-process raw detections using **Non-Max Suppression (NMS)**

1. Select next highest-scoring box
2. Eliminate lower-scoring boxes with $\text{IoU} > \text{threshold}$ (e.g. 0.7)
3. If any boxes remain, GOTO 1



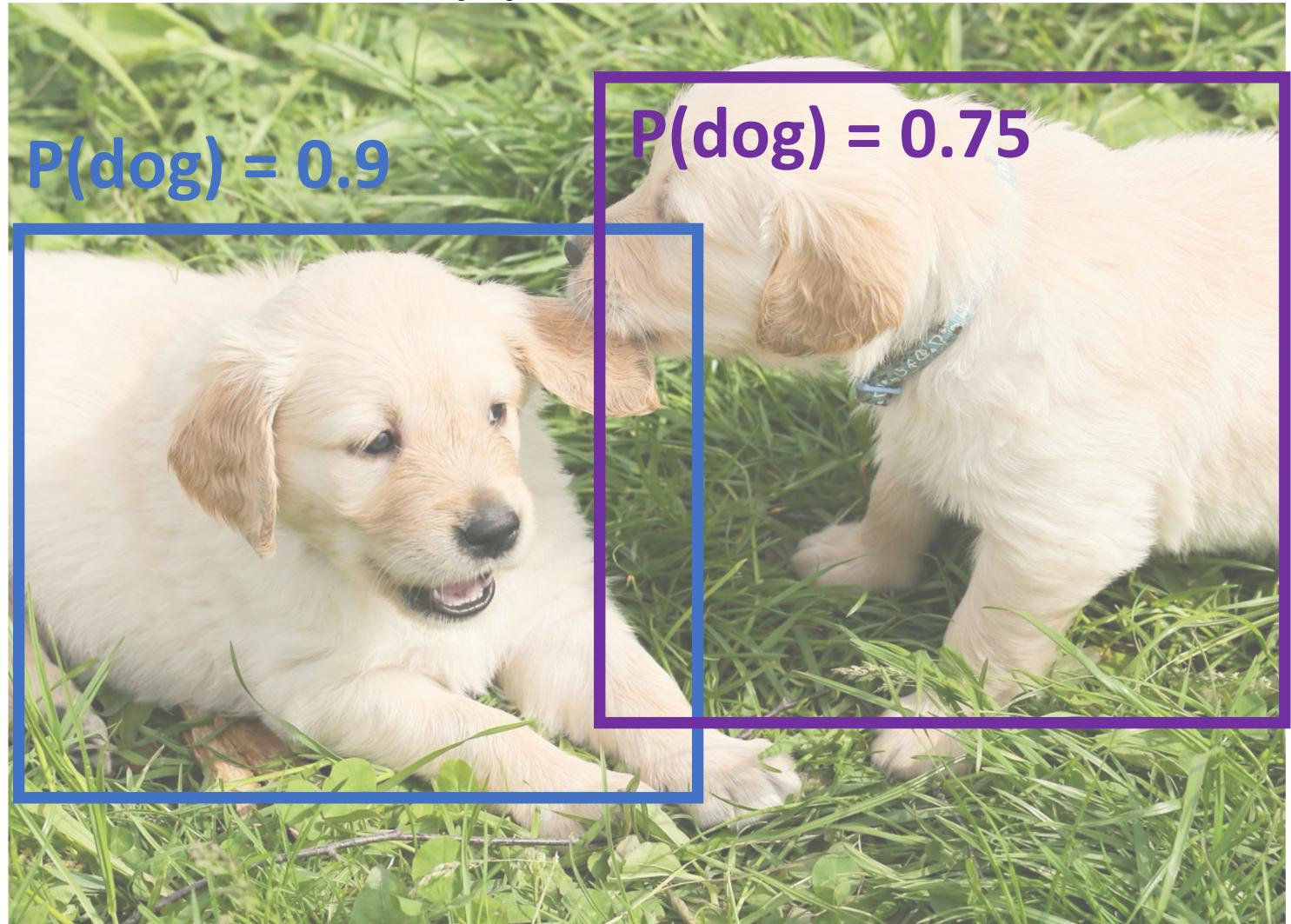
[Puppy image is CC0 Public Domain](#)

Overlapping Boxes: Non-Max Suppression (NMS)

Problem: Object detectors often output many overlapping detections:

Solution: Post-process raw detections using **Non-Max Suppression (NMS)**

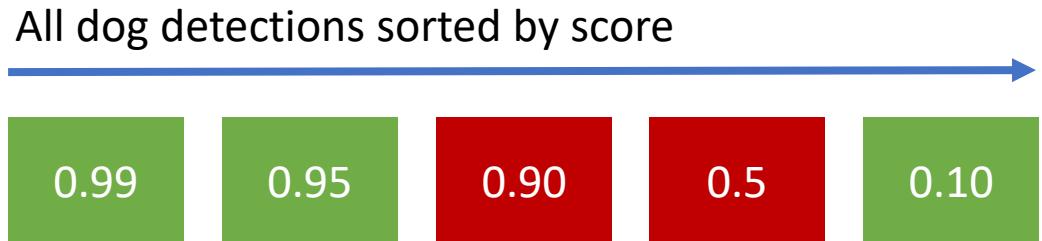
1. Select next highest-scoring box
2. Eliminate lower-scoring boxes with $\text{IoU} > \text{threshold}$ (e.g. 0.7)
3. If any boxes remain, GOTO 1



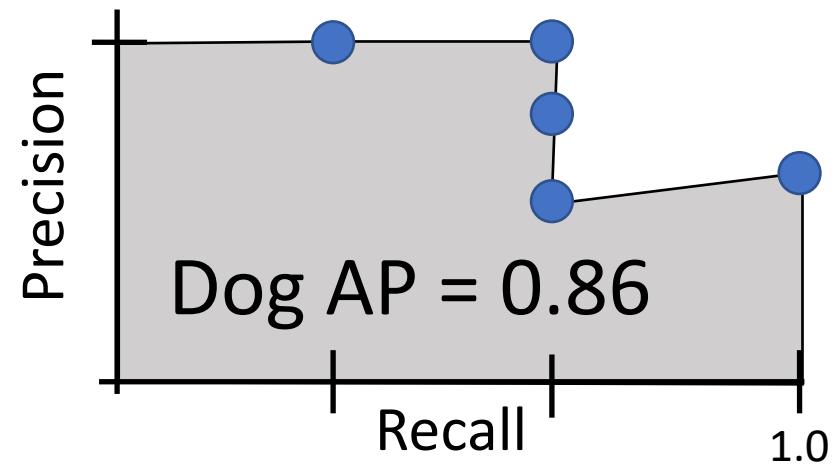
[Puppy image is CC0 Public Domain](#)

Evaluating Object Detectors: Mean Average Precision (mAP)

1. Run object detector on all test images (with NMS)
2. For each category, compute Average Precision (AP) = area under Precision vs Recall Curve
 1. For each detection (highest score to lowest score)
 1. If it matches some GT box with $\text{IoU} > 0.5$, mark it as positive and eliminate the GT
 2. Otherwise mark it as negative
 3. Plot a point on PR Curve
 2. Average Precision (AP) = area under PR curve



All ground-truth dog boxes



Evaluating Object Detectors: Mean Average Precision (mAP)

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

$\text{mAP}@0.5 = 0.77$

$\text{mAP}@0.55 = 0.71$

$\text{mAP}@0.60 = 0.65$

...

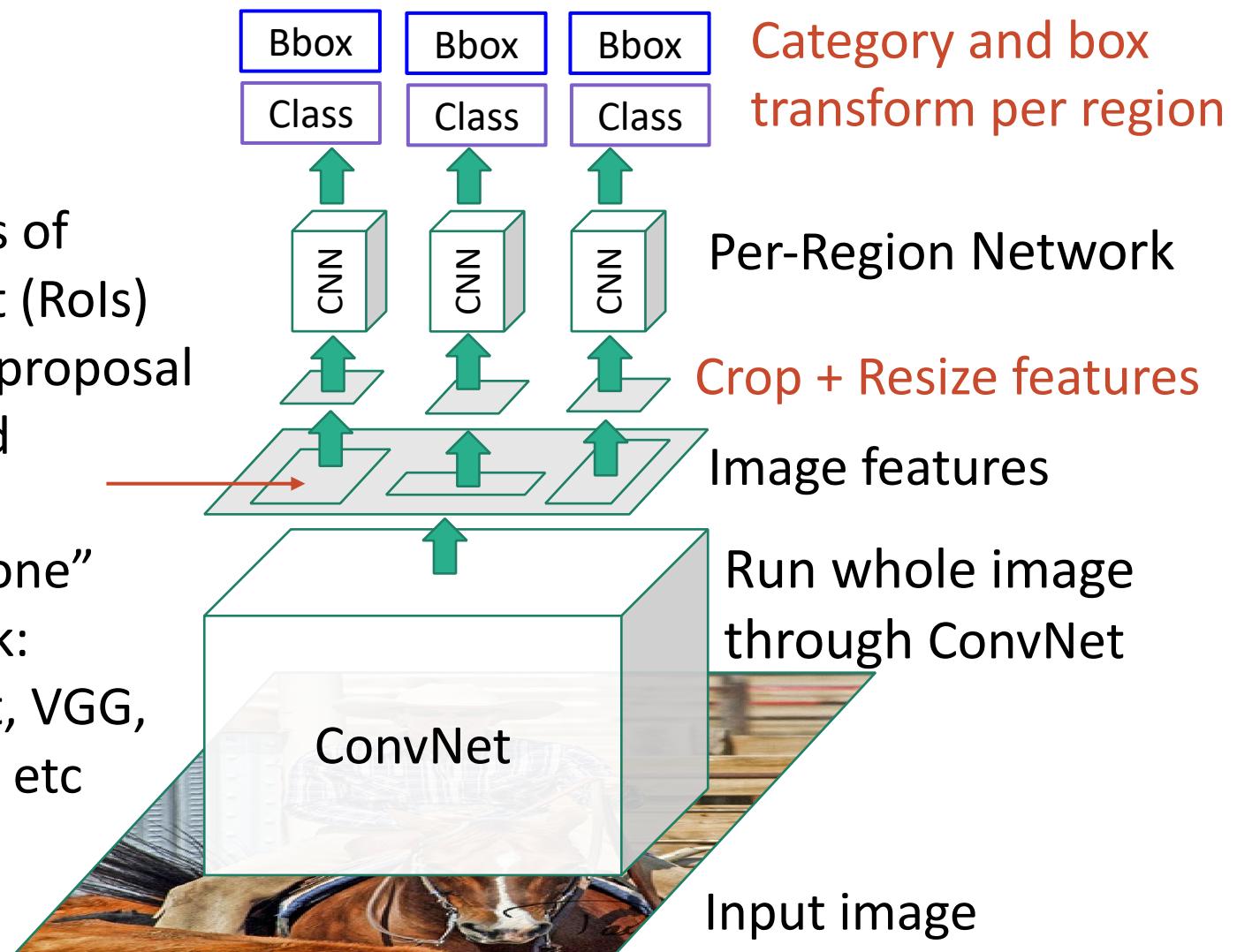
$\text{mAP}@0.95 = 0.2$

COCO mAP = 0.4

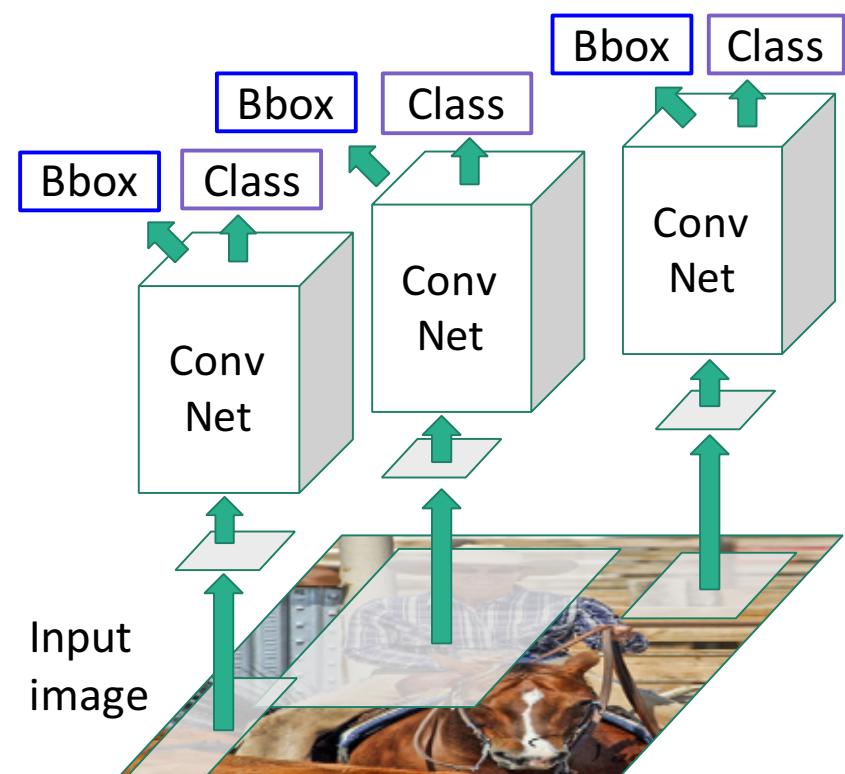
Fast R-CNN

Regions of Interest (Rois)
from a proposal
method

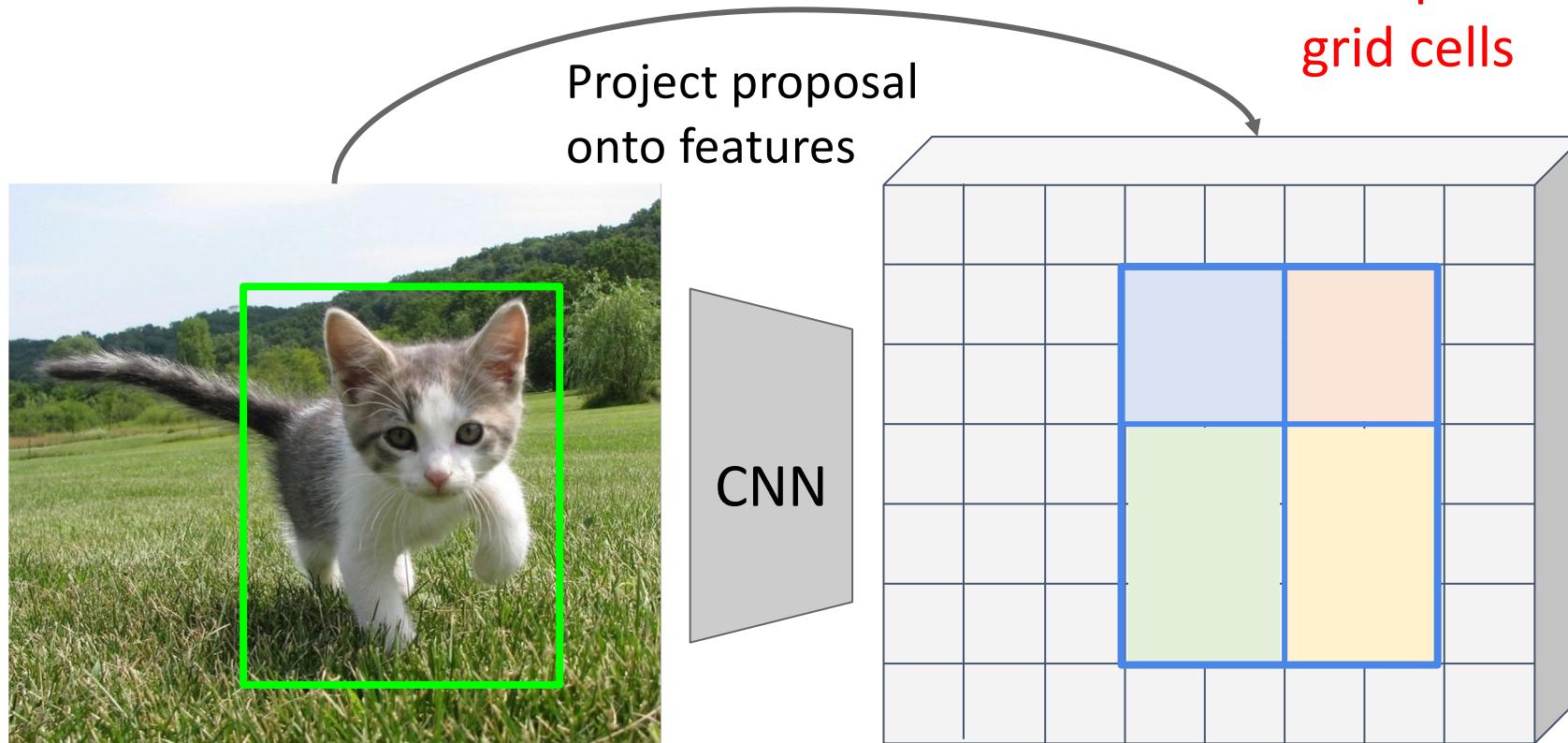
“Backbone”
network:
AlexNet, VGG,
ResNet, etc



“Slow” R-CNN
Process each region independently



Cropping Features: RoI Pool



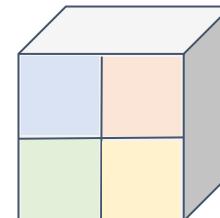
Input Image
(e.g. $3 \times 640 \times 480$)

Image features
(e.g. $512 \times 20 \times 15$)

Problem: Slight misalignment due to snapping; different-sized subregions is weird

Divide into 2×2 grid of (roughly) equal subregions

Max-pool within each subregion

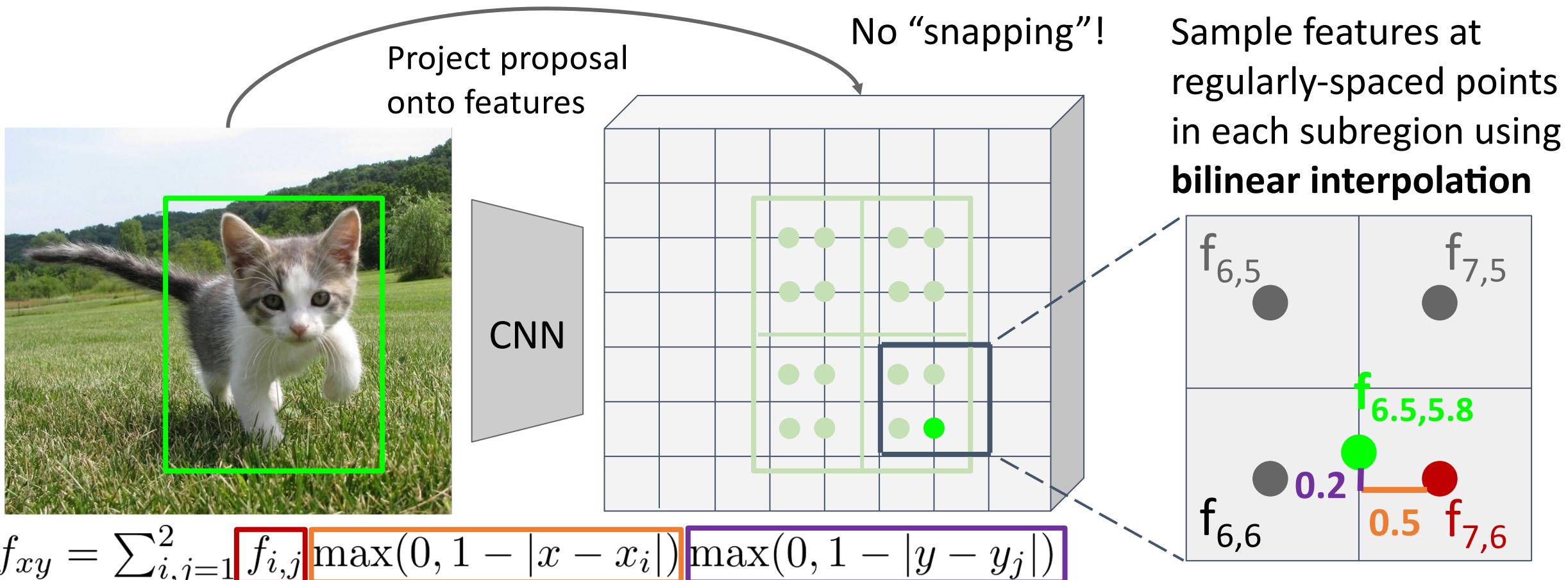


Region features
(here $512 \times 2 \times 2$;
In practice e.g. $512 \times 7 \times 7$)

Region features always the same size even if input regions have different sizes!

Cropping Features: RoI Align

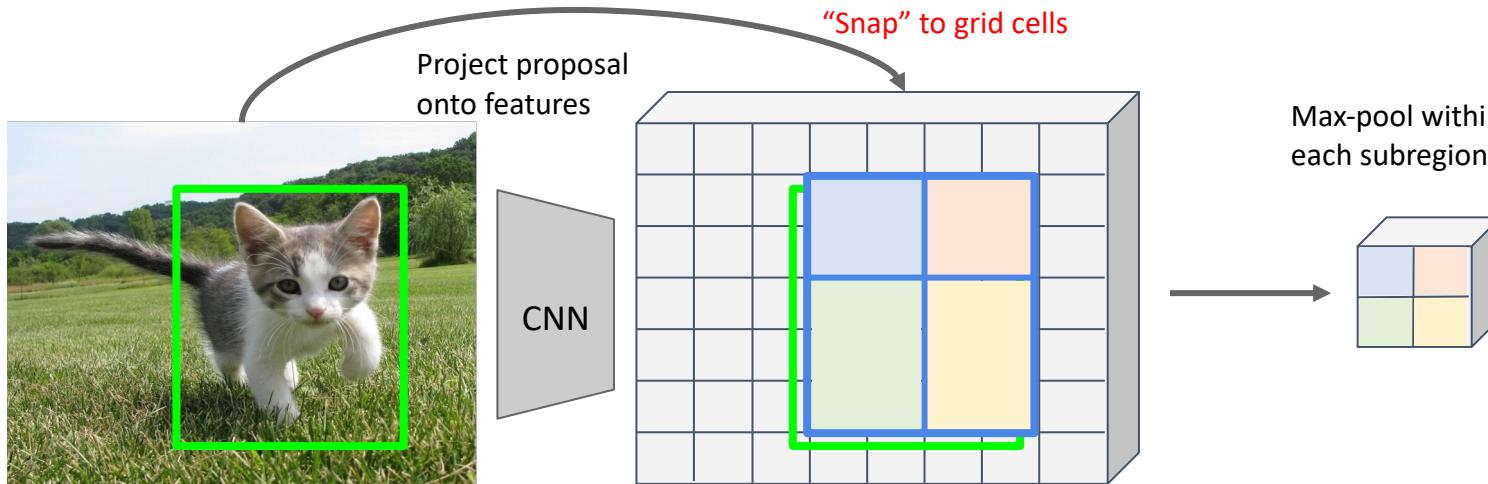
Divide into equal-sized subregions
(may not be aligned to grid!)



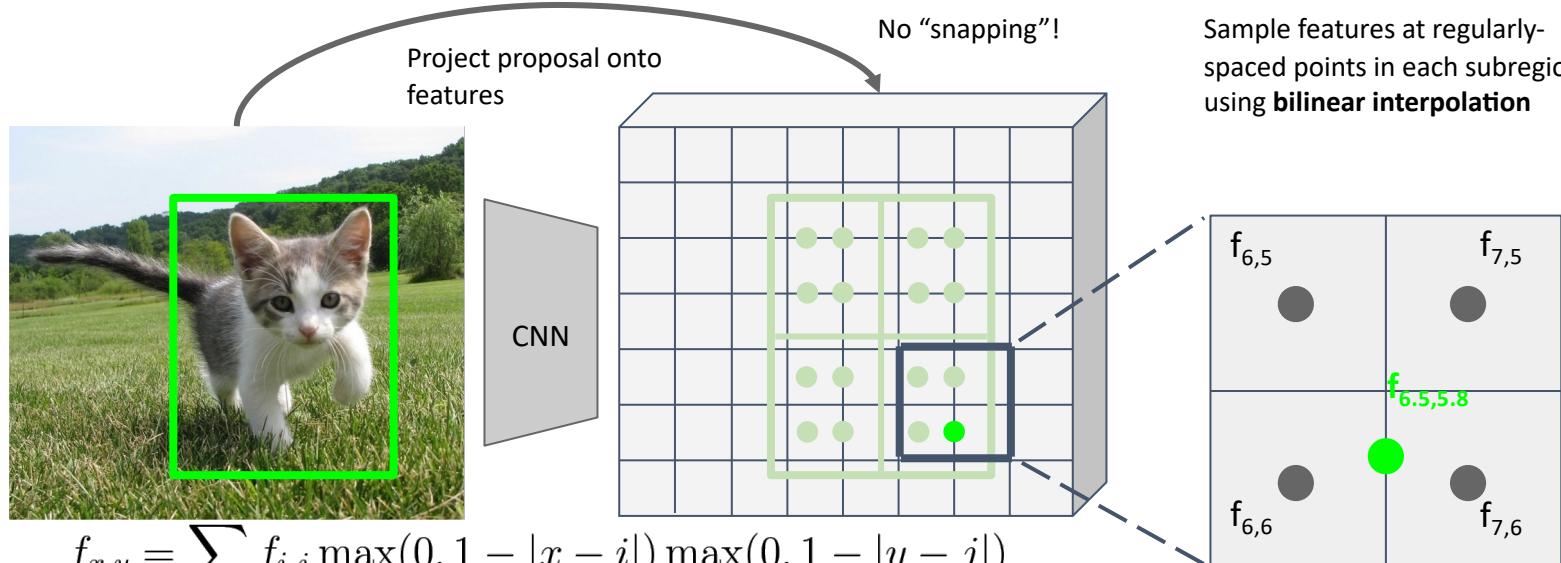
Feature f_{xy} for point (x, y) is a linear combination of features at its four neighboring grid cells:

Cropping Features

RoI Pool



RoI Align



$$f_{x,y} = \sum f_{i,j} \max(0, 1 - |x - i|) \max(0, 1 - |y - j|)$$

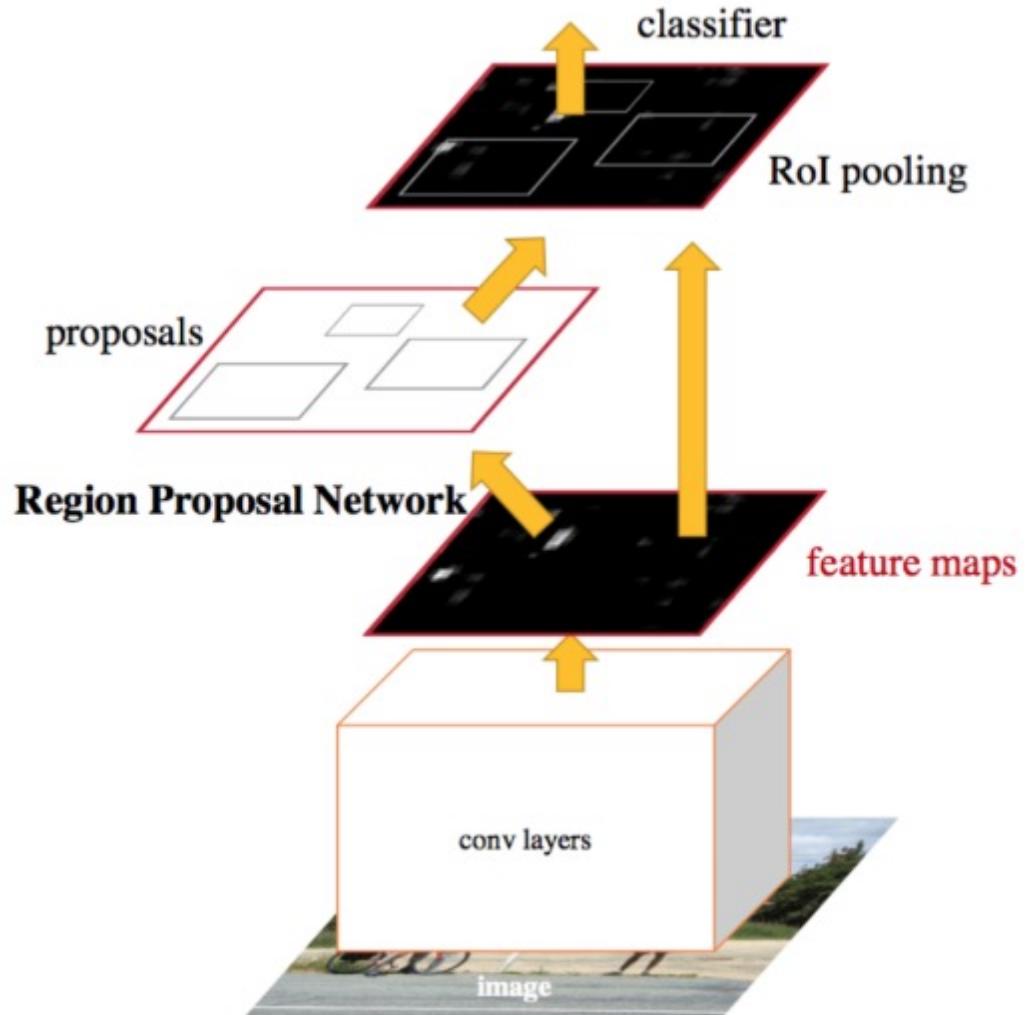
$$\begin{aligned} f_{6,5,8} &= (f_{6,5} * 0.5 * 0.2) + (f_{7,5} * 0.5 * 0.2) \\ &\quad + (f_{6,6} * 0.5 * 0.8) + (f_{7,6} * 0.5 * 0.8) \end{aligned}$$

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

Faster R-CNN: Learnable Region Proposals

Insert **Region Proposal Network (RPN)** to predict proposals from features

Otherwise same as Fast R-CNN:
Crop features for each proposal, classify each one



Ren et al, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", NIPS 2015
Figure copyright 2015, Ross Girshick; reproduced with permission

Region Proposal Network (RPN)

Run backbone CNN to get features aligned to input image



Input Image
(e.g. $3 \times 640 \times 480$)

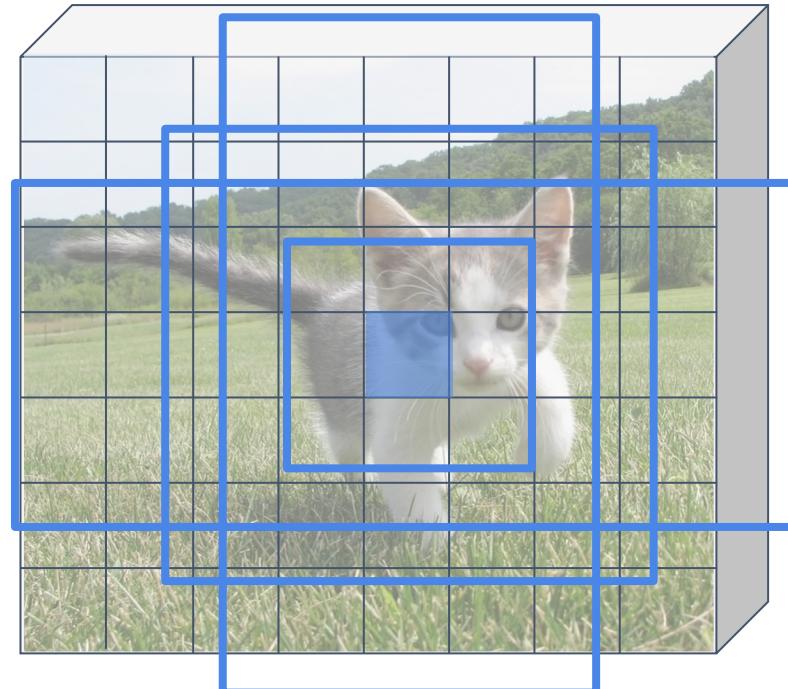
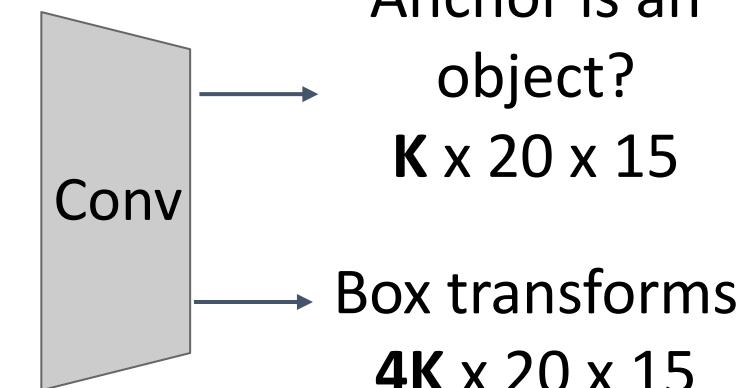


Image features
(e.g. $512 \times 20 \times 15$)

Problem: Anchor box may have the wrong size / shape
Solution: Use K different anchor boxes at each point!

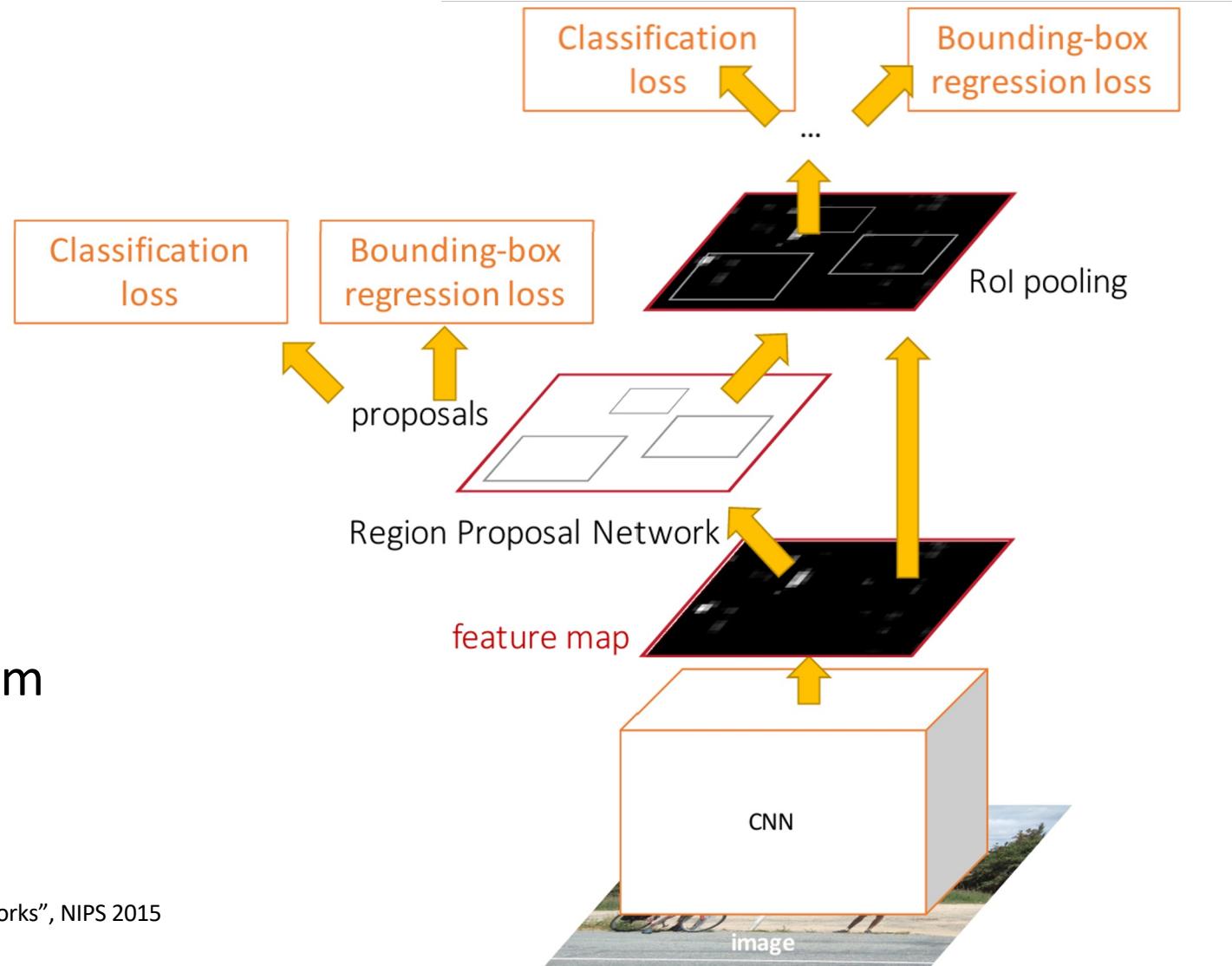


At test time: sort all $K \times 20 \times 15$ boxes by their score, and take the top ~ 300 as our region proposals

Faster R-CNN: Learnable Region Proposals

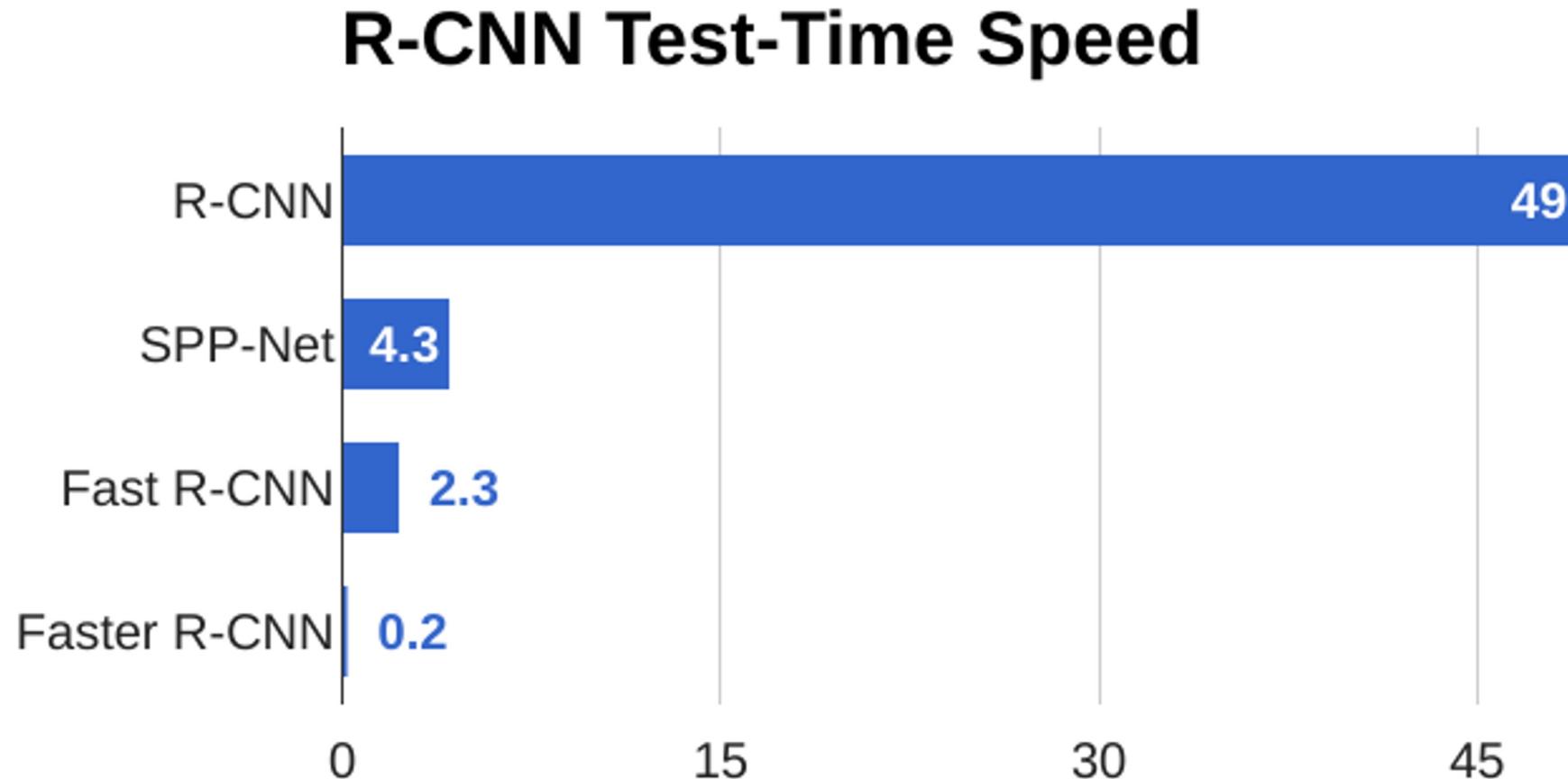
Jointly train with 4 losses:

1. **RPN classification**: anchor box is object / not an object
2. **RPN regression**: predict transform from anchor box to proposal box
3. **Object classification**: classify proposals as background / object class
4. **Object regression**: predict transform from proposal box to object box



Ren et al, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", NIPS 2015
Figure copyright 2015, Ross Girshick; reproduced with permission

Faster R-CNN: Learnable Region Proposals



Single-Stage Object Detection: YOLO and SSD

Run backbone CNN to get
features aligned to input image



Input Image
(e.g. $3 \times 640 \times 480$)

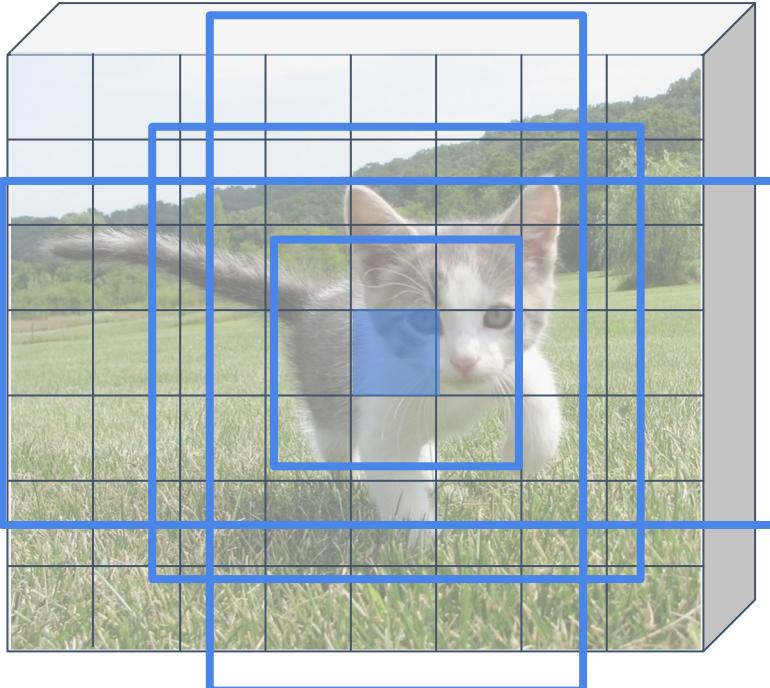
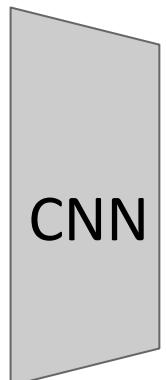


Image features
(e.g. $512 \times 20 \times 15$)

RPN: Classify each anchor as
object / not object
Single-Stage Detector: Classify
each object as one of C
categories (or background)

Anchor category
 $\rightarrow (C+1) \times K \times 20 \times 15$



Box transforms
 $C \times 4K \times 20 \times 15$

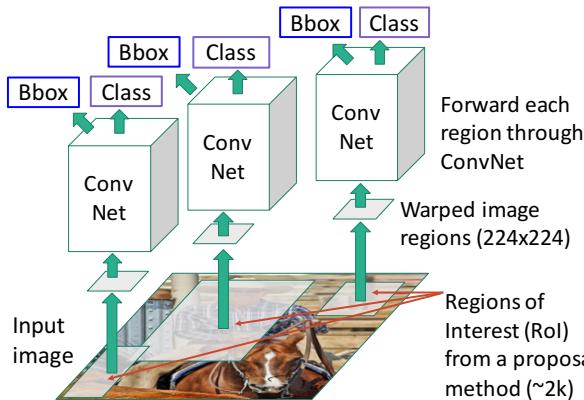
Sometimes use **category-specific regression**: Predict
different box transforms for
each category

Redmon et al, "You Only Look Once: Unified, Real-Time Object Detection", CVPR 2016
Liu et al, "SSD: Single-Shot MultiBox Detector", ECCV 2016
Lin et al, "Focal Loss for Dense Object Detection", ICCV 2017

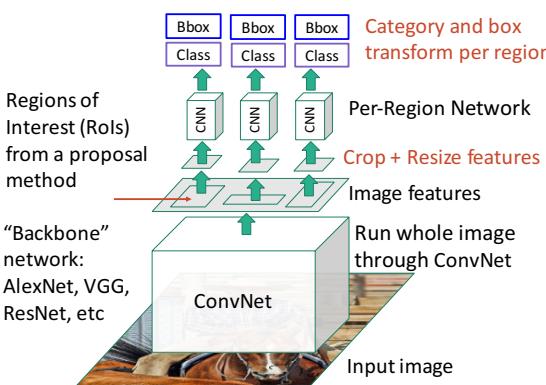
YOLO series

Summary of object detection

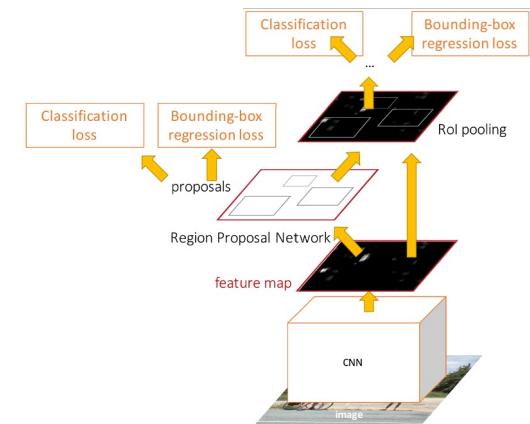
“Slow” R-CNN: Run CNN independently for each region



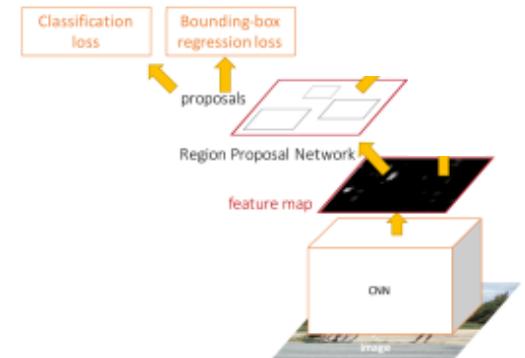
Fast R-CNN: Apply differentiable cropping to shared image features



Faster R-CNN: Compute proposals with CNN



Single-Stage: Fully convolutional detector



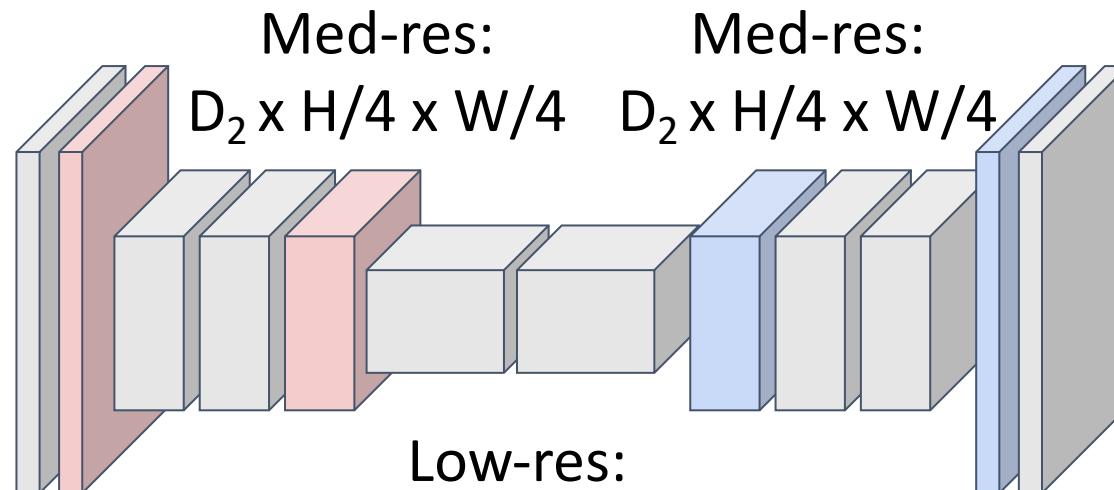
Semantic Segmentation: Fully Convolutional Network

Downsampling:
Pooling, strided
convolution



Input:
 $3 \times H \times W$

High-res:
 $D_1 \times H/2 \times W/2$



Design network as a bunch of convolutional layers, with
downsampling and **upsampling** inside the network!

Upsampling:
Interpolation,
transposed conv



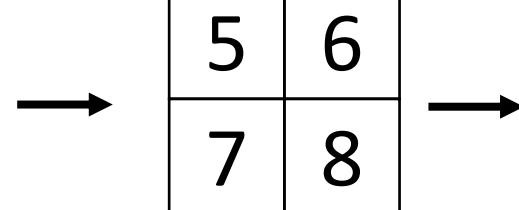
Predictions:
 $H \times W$

Loss function: Per-Pixel cross-entropy

In-Network Upsampling: “Max Unpooling”

Max Pooling: Remember which position had the max

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

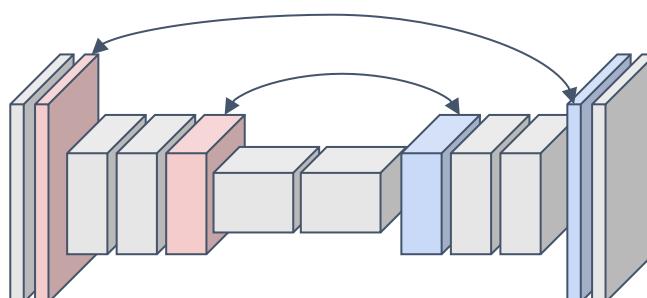


Rest
of
net

1	2
3	4



0	0	2	0
0	1	0	0
0	0	0	0
3	0	0	4



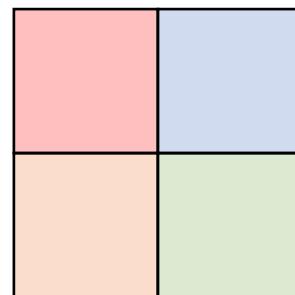
Pair each downsampling layer with an upsampling layer

Noh et al, “Learning Deconvolution Network for Semantic Segmentation”, ICCV 2015

Learnable Upsampling: Transposed Convolution

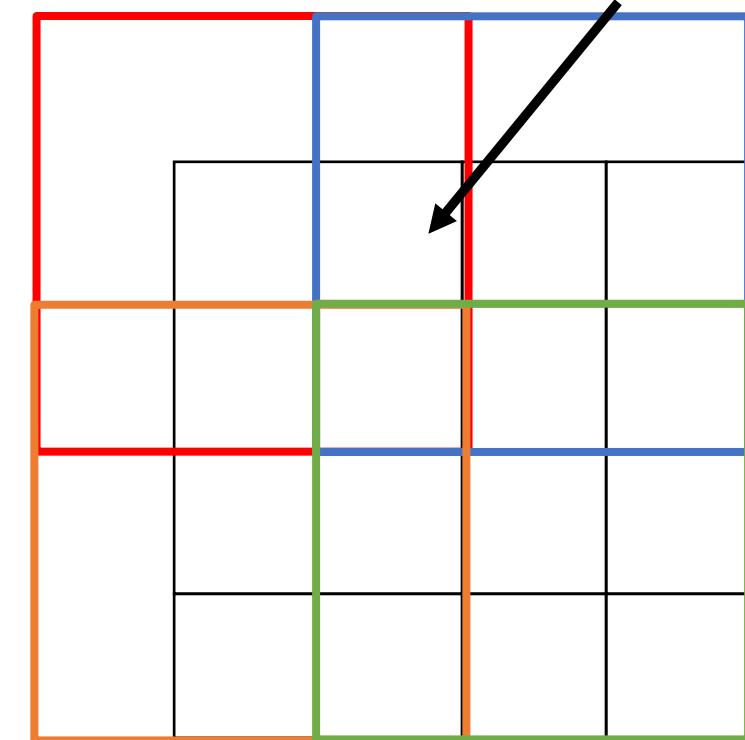
3 x 3 convolution transpose, stride 2

This gives 5x5 output – need to trim one pixel from top and left to give 4x4 output



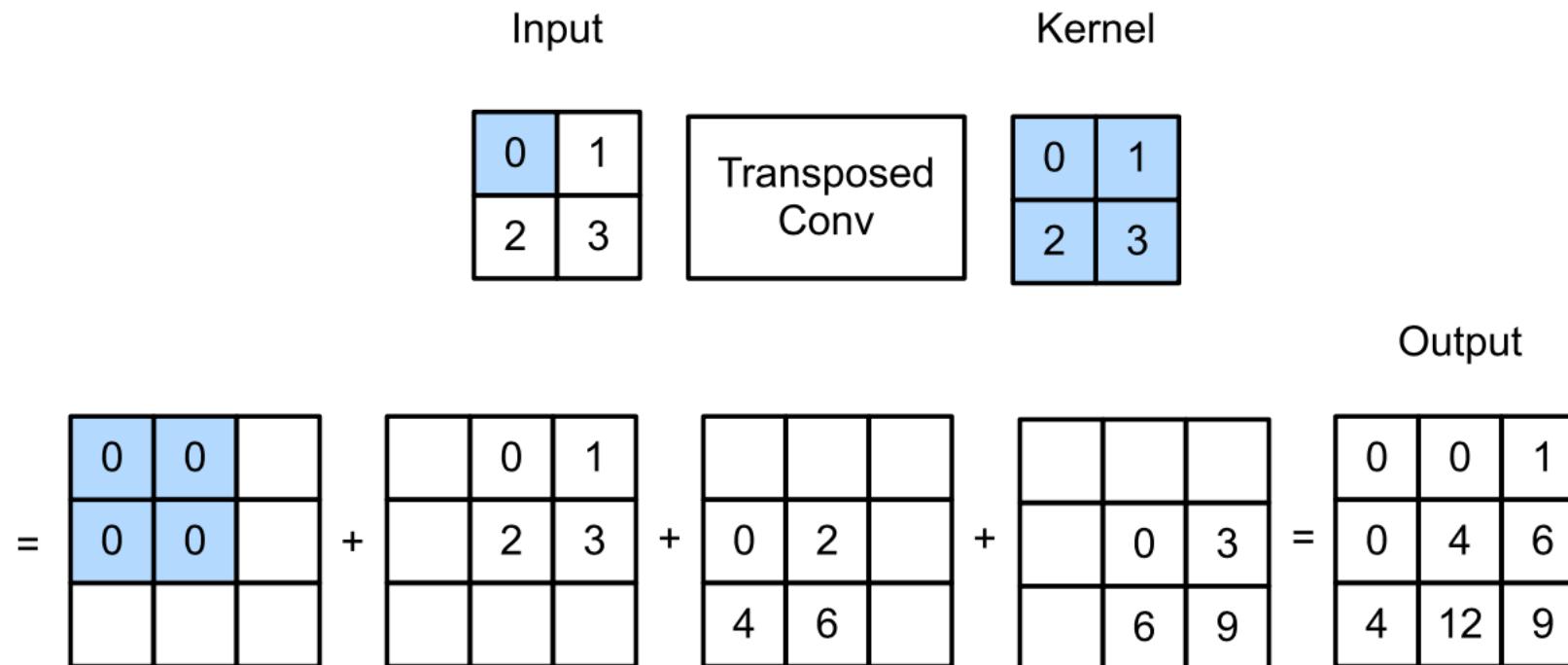
Input: 2 x 2

Weight filter by
input value and
copy to output



Output: 4 x 4

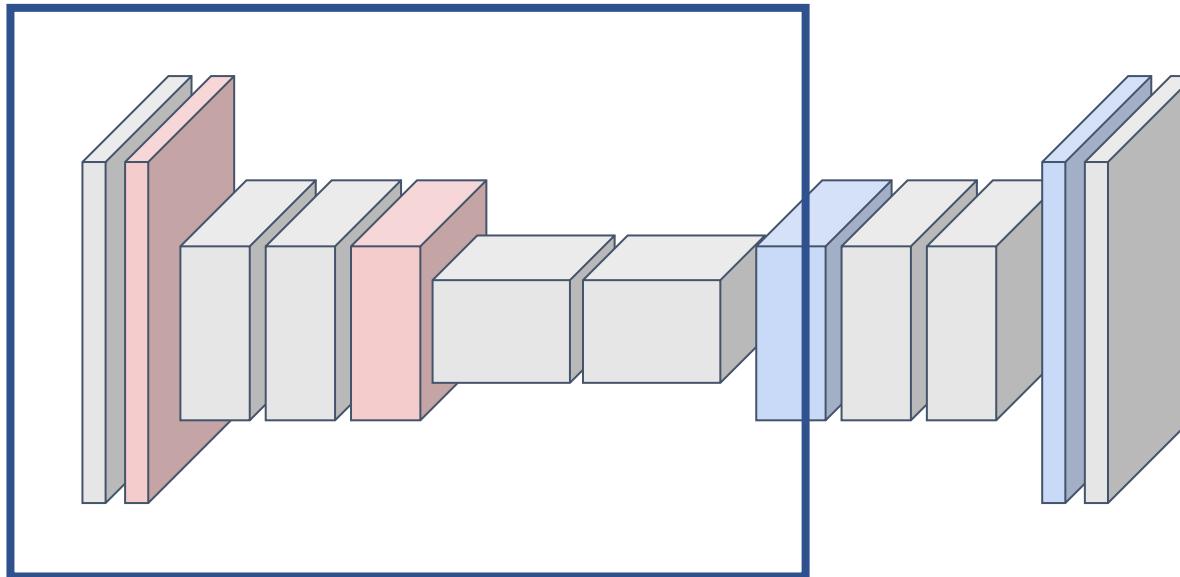
Transposed Convolution: 2D example



Innovation on encoder: How to better encode the scene context



Input:
 $3 \times H \times W$



Dilated convolution, feature pyramid structures, U-Net structure



Predictions:
 $H \times W$

Instance Segmentation: Mask R-CNN

Object
Detection



DOG, DOG, CAT



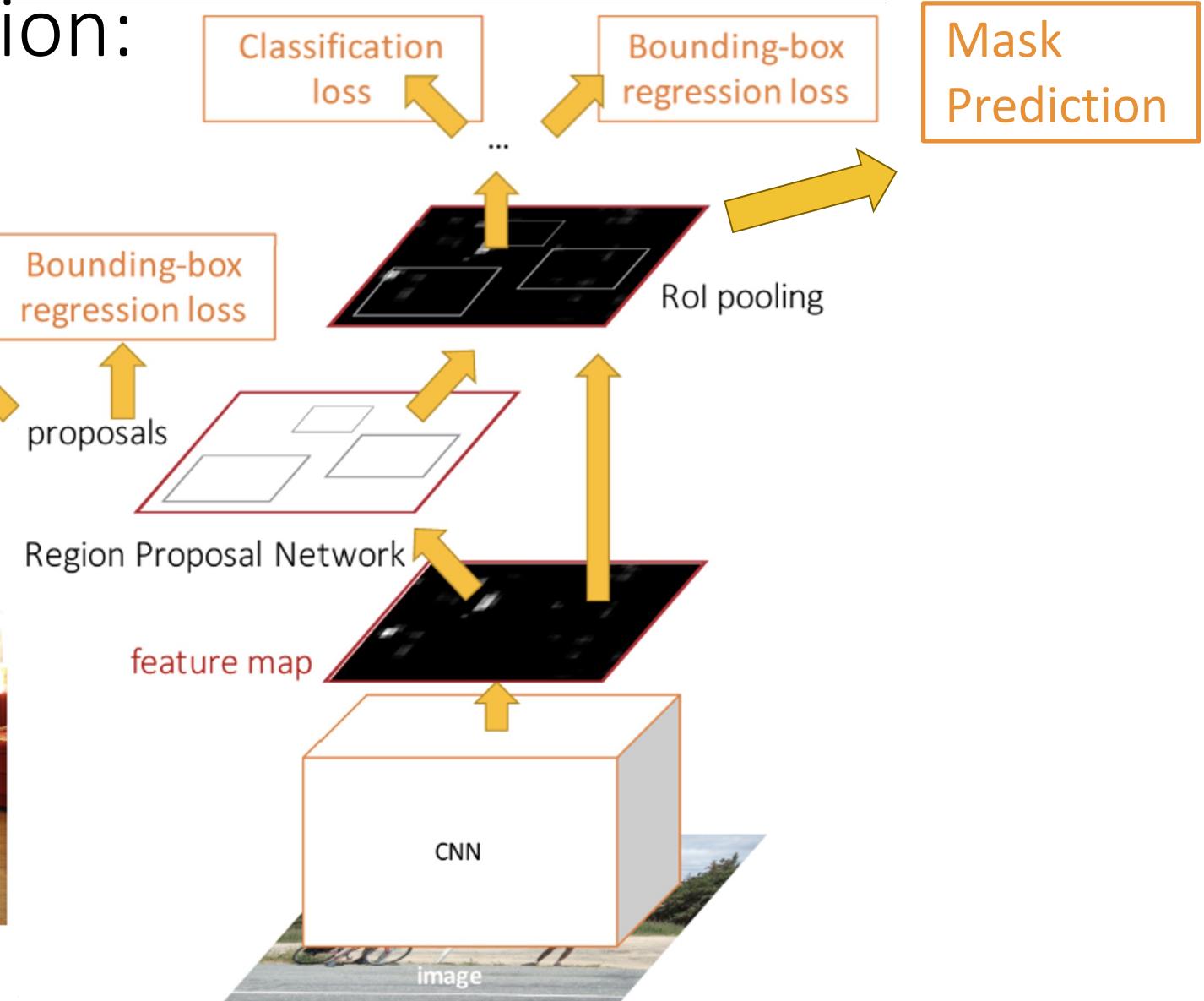
DOG, DOG, CAT

Instance Segmentation

Classification
loss

Bounding-box
regression loss

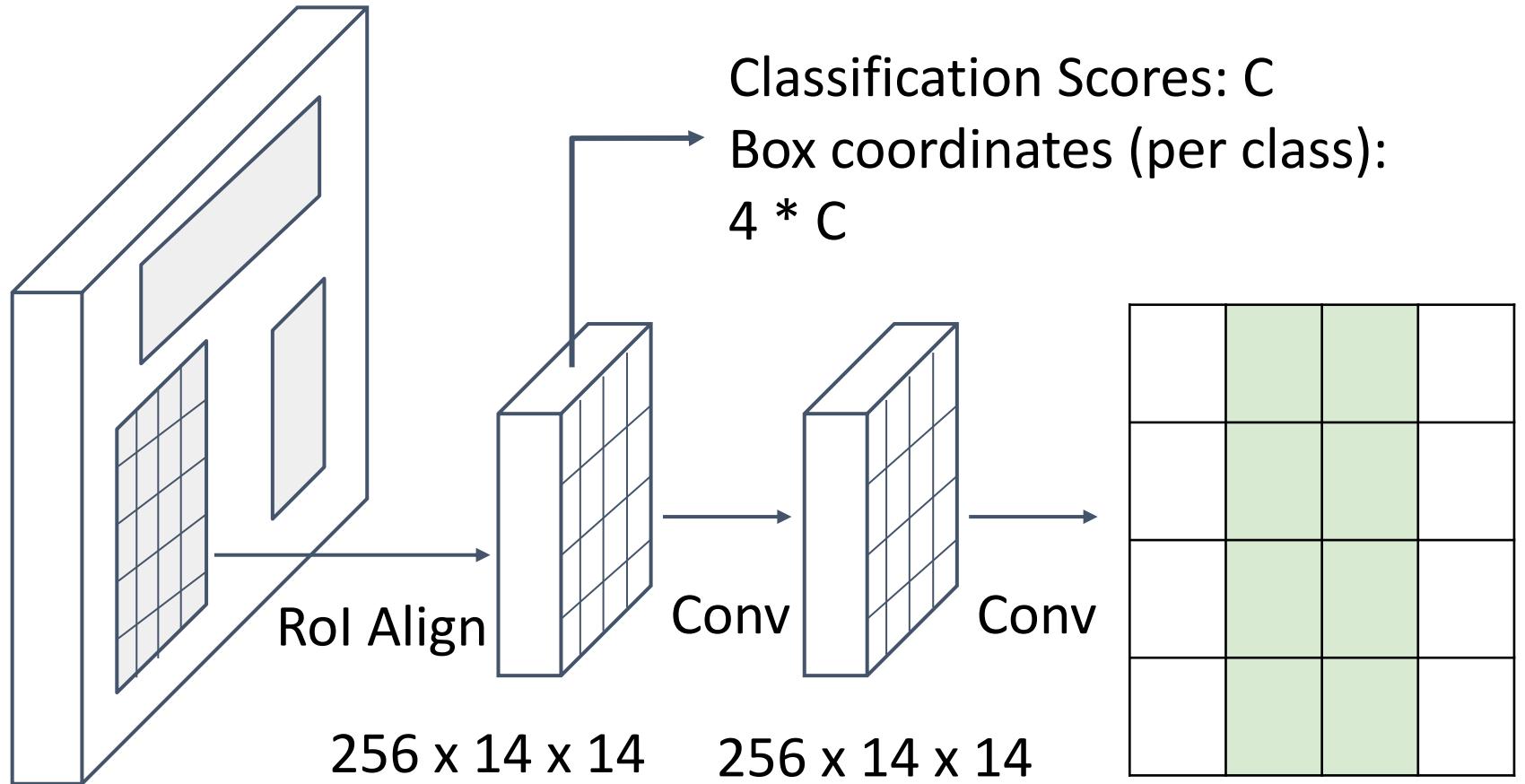
Mask
Prediction



Mask R-CNN

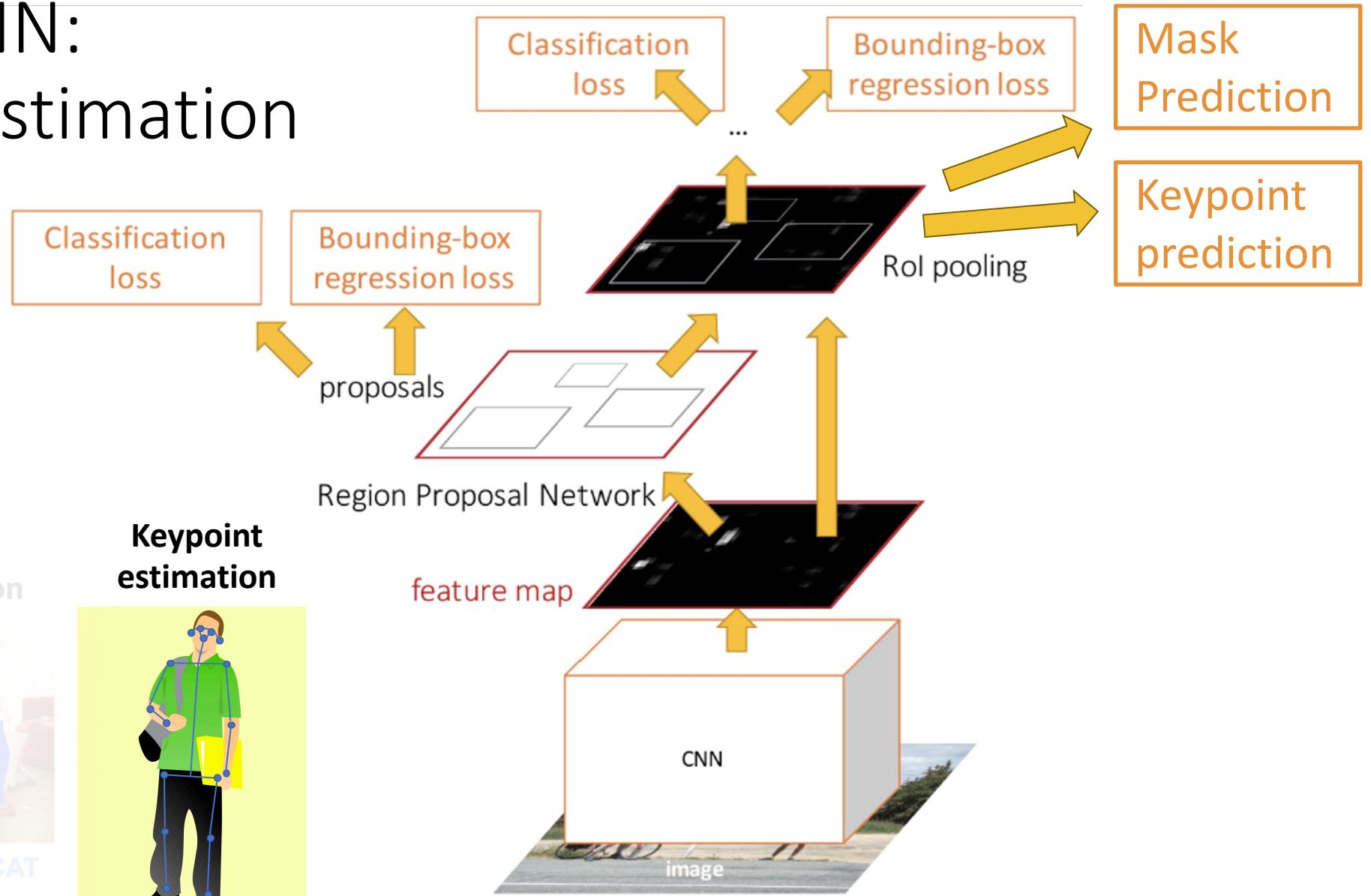
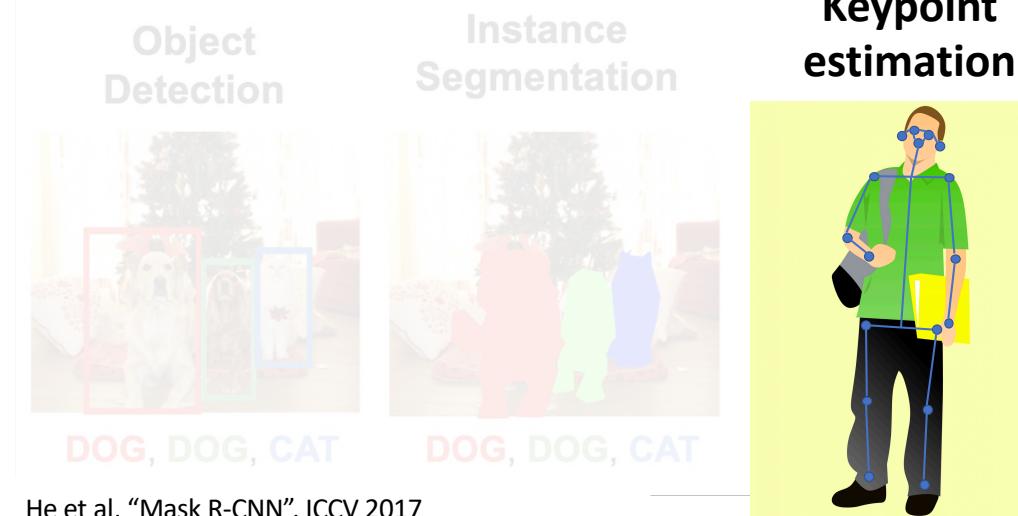


CNN
+RPN



Predict a mask for
each of C classes:
 $C \times 28 \times 28$ (mask size)

Mask R-CNN: Keypoint Estimation



Lecture 15, 16: Generative Models in Computer Vision

Supervised vs Unsupervised Learning

Supervised Learning

Data: (x, y)

x is data, y is label

Goal: Learn a *function* to map $x \rightarrow y$

Examples: Classification, regression,
object detection, semantic
segmentation, image captioning, etc.

Unsupervised Learning

Data: x

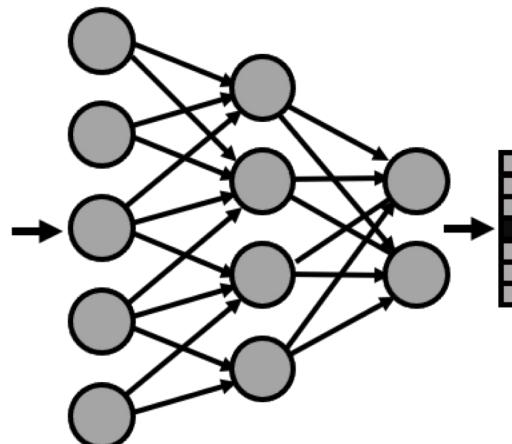
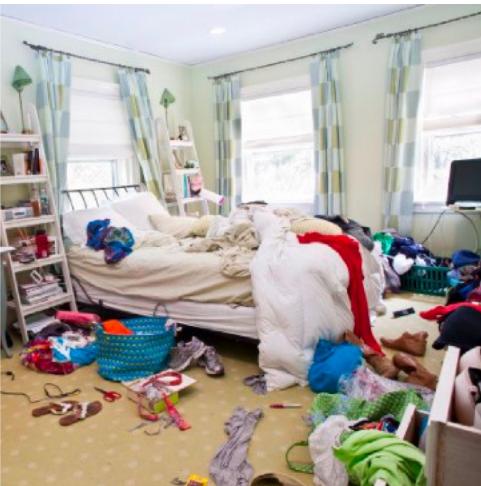
Just data, no labels!

Goal: Learn some underlying
hidden *structure* of the data

Examples: Clustering,
dimensionality reduction, feature
learning, density estimation, etc.

Discriminative vs Generative Models in CV

Discriminative models are everywhere for visual recognition



Scene categorization
Bedroom

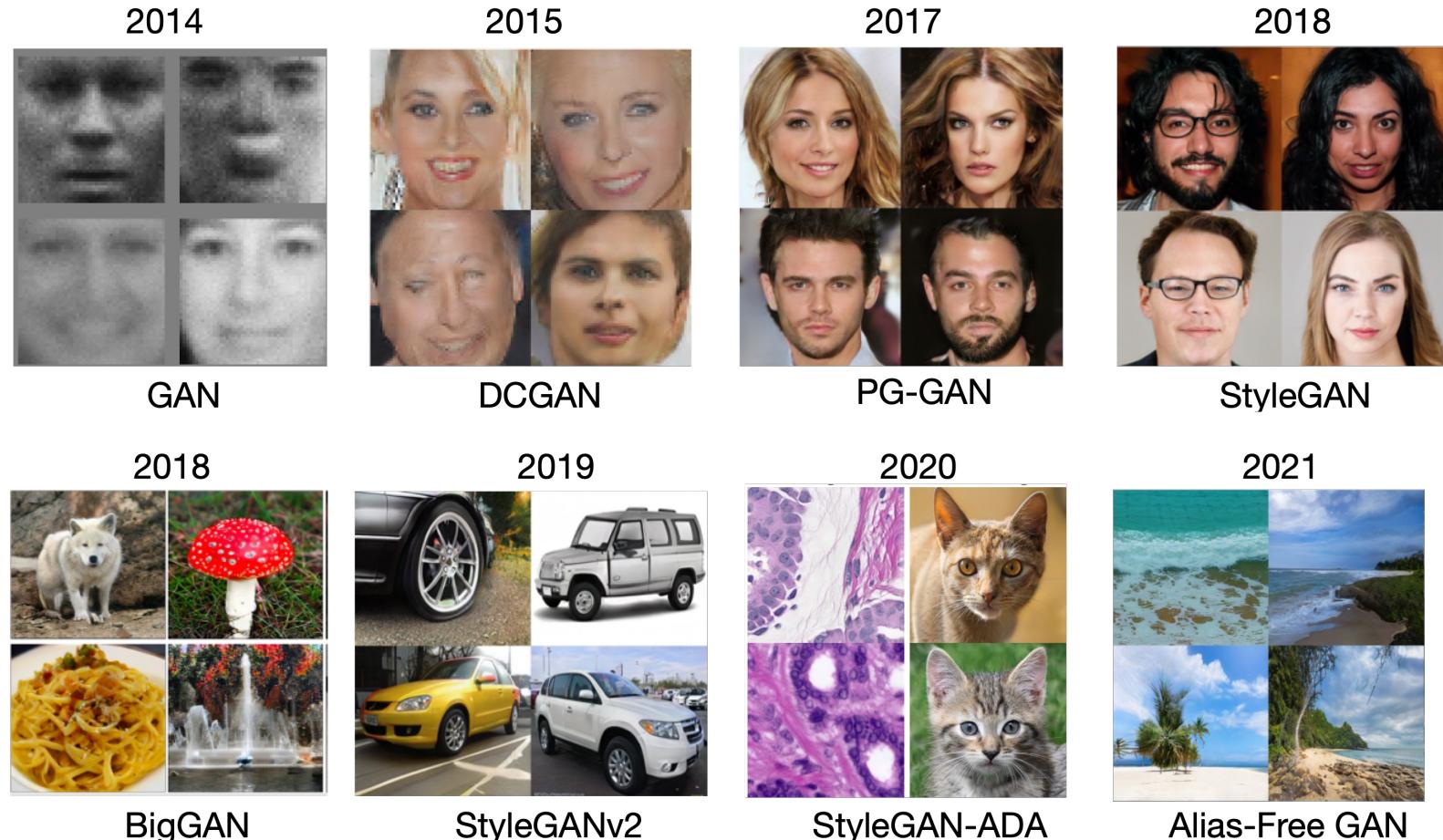
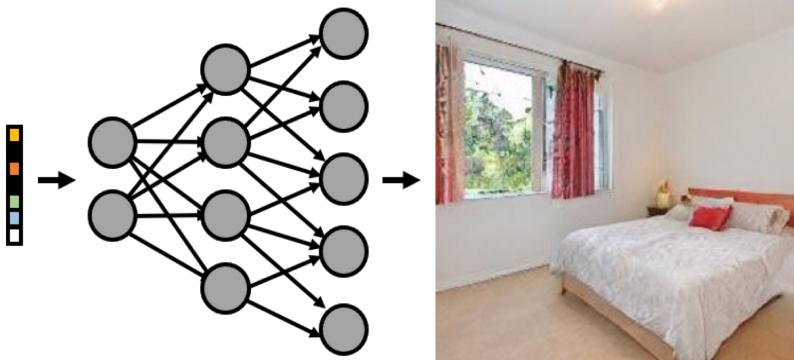
Attribute prediction
Messy, natural-lighting

Semantic segmentation



Recent Advances in Generative Models

Generative Model



Taxonomy of Generative Models

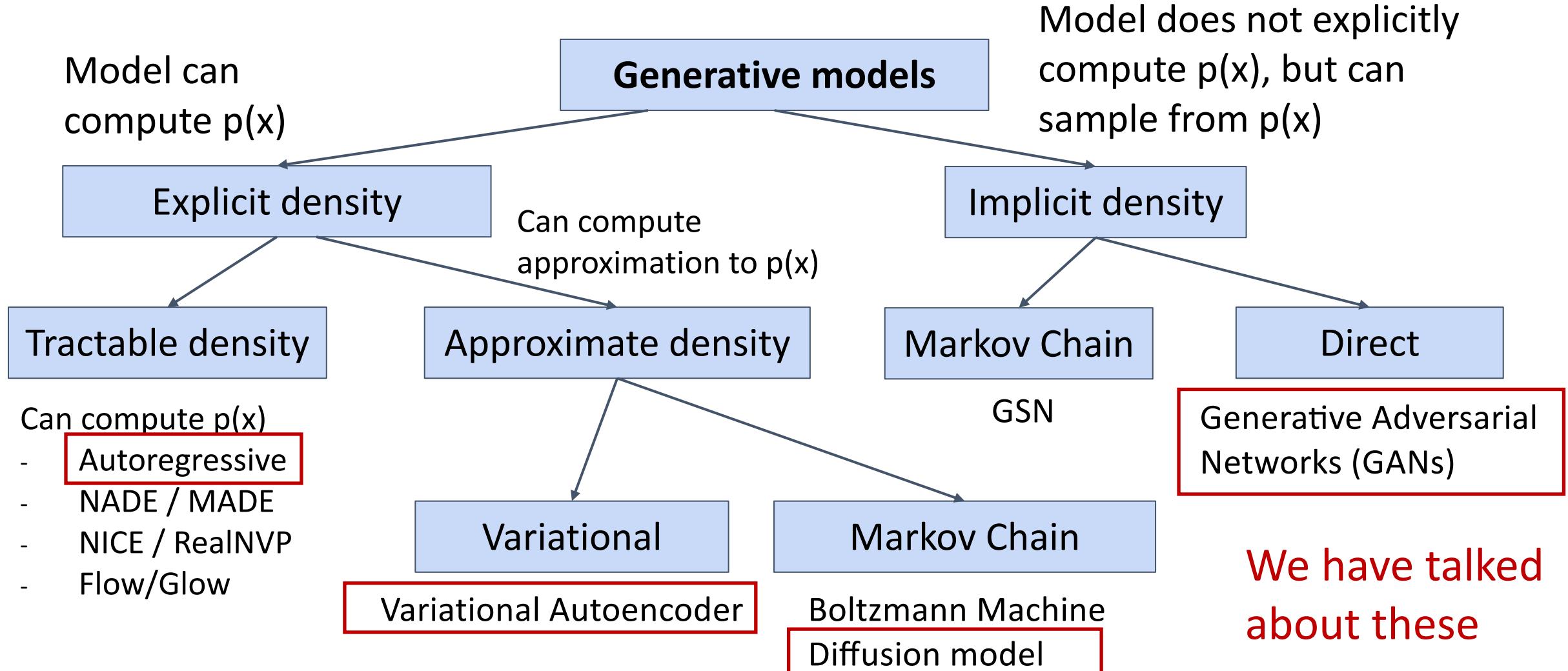


Figure adapted from Ian Goodfellow, Tutorial on Generative Adversarial Networks, 2017.

Autoregressive Models: PixelRNN and PixelCNN

Pros:

- Can explicitly compute likelihood $p(x)$
- Explicit likelihood of training data gives good evaluation metric
- Good samples

Con:

- Sequential generation => slow

Improving PixelCNN performance

- Gated convolutional layers
- Short-cut connections
- Discretized logistic loss
- Multi-scale
- Training tricks
- Etc...

See

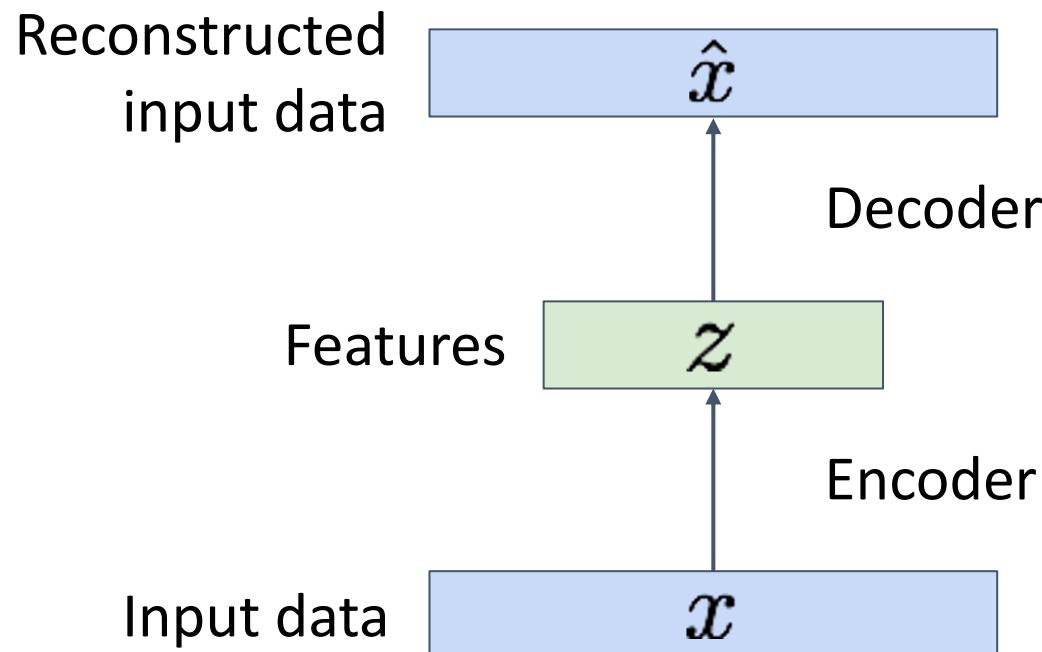
- Van der Oord et al. NIPS 2016
- Salimans et al. 2017 (PixelCNN++)

(Regular, non-variational) Autoencoders

Autoencoders learn **latent features** for data without any labels!

Can use features to initialize a **supervised** model

Not probabilistic: No way to sample new data from learned model



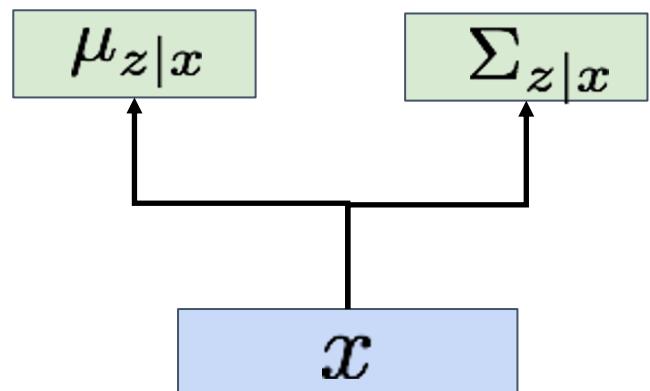
Variational Autoencoders

Jointly train **encoder** q and **decoder** p to maximize the **variational lower bound** on the data likelihood

$$\log p_\theta(x) \geq E_{z \sim q_\phi(z|x)} [\log p_\theta(x|z)] - D_{KL} (q_\phi(z|x), p(z))$$

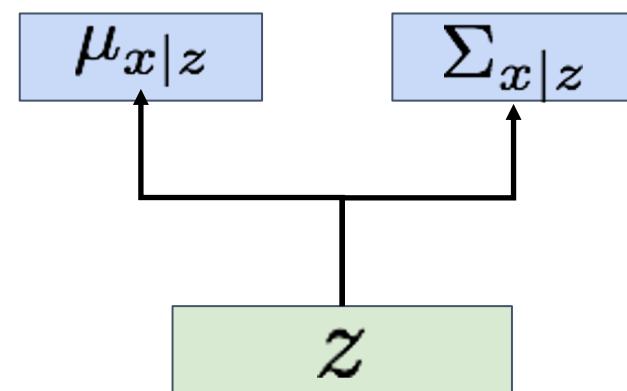
Encoder Network

$$q_\phi(z | x) = N(\mu_{z|x}, \Sigma_{z|x})$$



Decoder Network

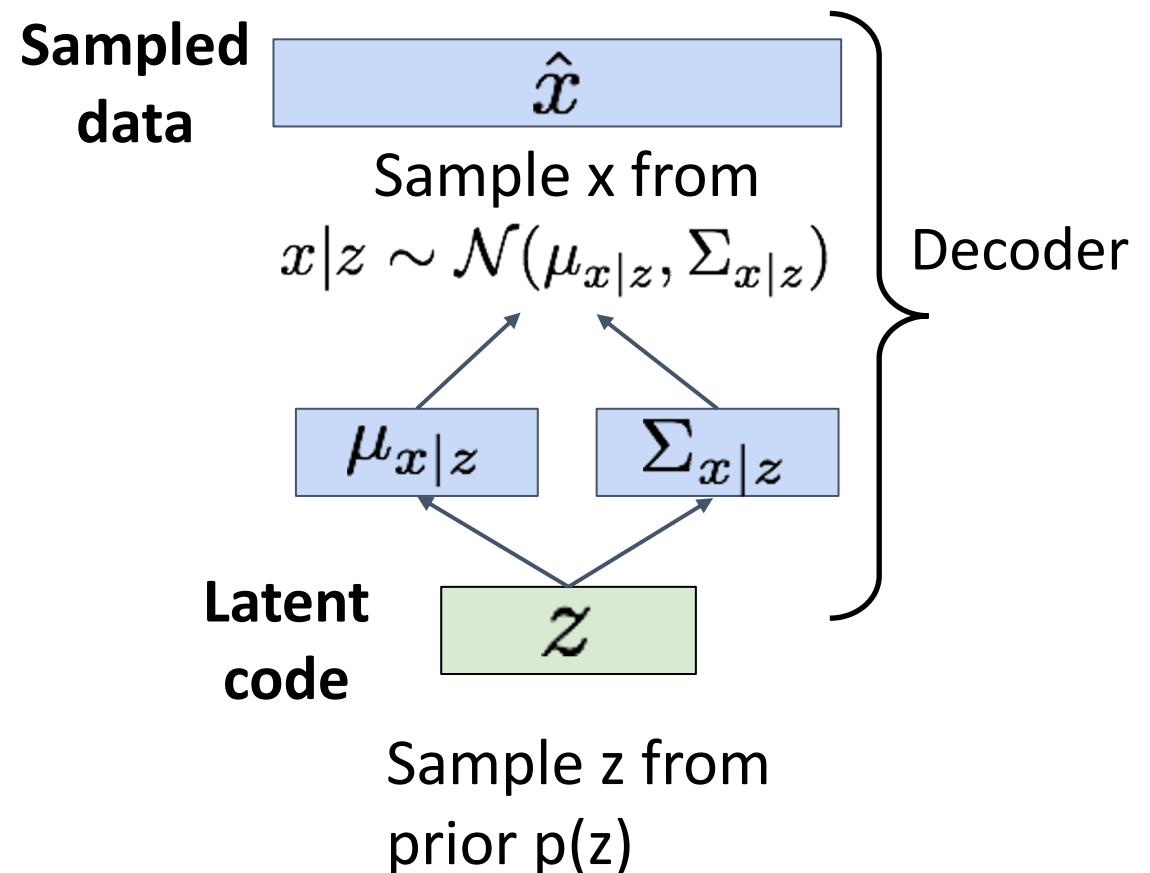
$$p_\theta(x | z) = N(\mu_{x|z}, \Sigma_{x|z})$$



Variational Autoencoders: Generating Data

After training we can generate new data!

1. Sample z from prior $p(z)$
2. Run sampled z through decoder to get distribution over data x
3. Sample from distribution in (2) to generate data



So far: Two types of generative models

Autoregressive models

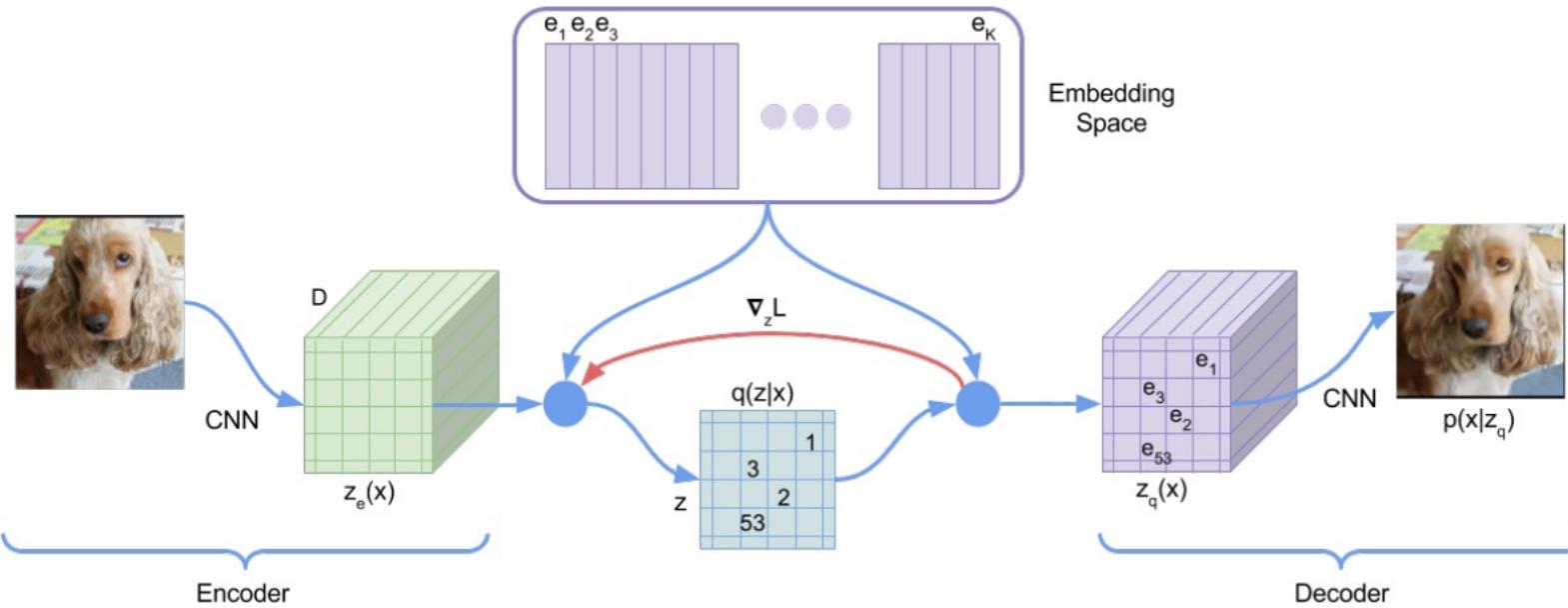
- Directly maximize $p(\text{data})$
- High-quality generated images
- Slow to generate images
- No explicit latent codes

Variational models

- Maximize lower-bound on $p(\text{data})$
- Generated images often blurry
- Very fast to generate images
- Learn rich latent codes

Can we combine them and get the best of both worlds?

Combining VAE + Autoregressive: Vector-Quantized Variational Autoencoder



run Vector Quantization in the latent space (continuous value->discrete distribution), then learn a PixelCNN on discrete latents as a prior

Generative Adversarial Networks

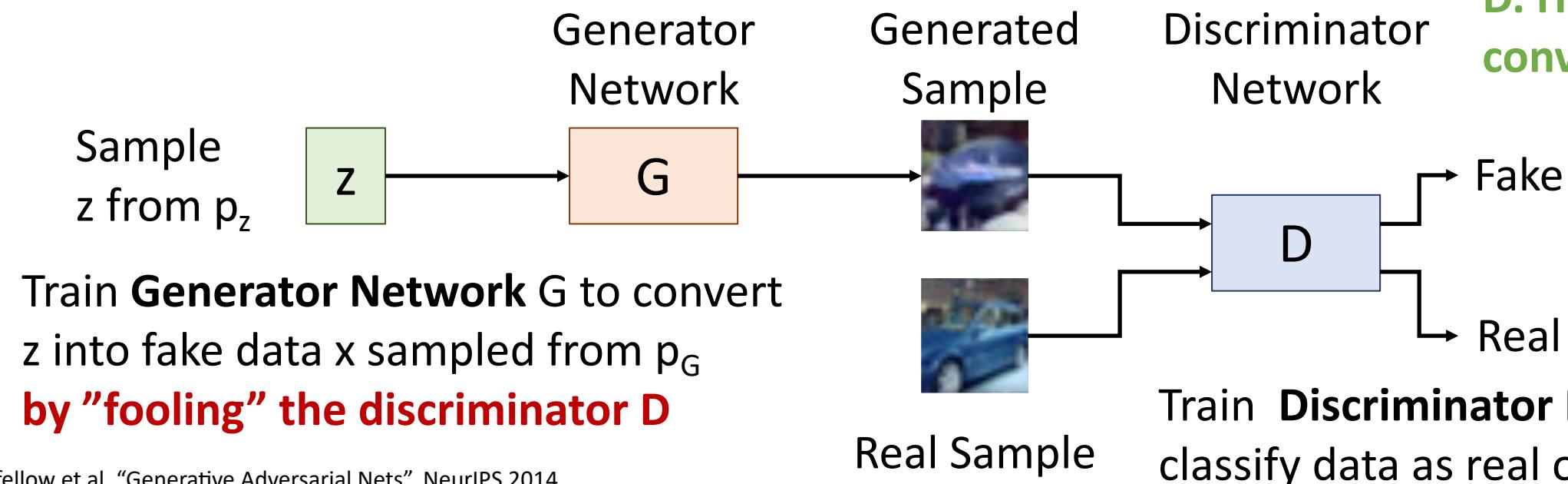
Setup: Assume we have data x_i drawn from distribution $p_{\text{data}}(x)$. Want to sample from p_{data} .

Idea: Introduce a latent variable z with simple prior $p(z)$.

Sample $z \sim p(z)$ and pass to a **Generator Network** $x = G(z)$

Then x is a sample from the **Generator distribution** p_G . Want $p_G = p_{\text{data}}$!

Jointly train **G** and **D**. Hopefully p_G converges to p_{data} !



Generative Adversarial Networks: Training Objective

Jointly train generator G and discriminator D with a **minimax game**

Train G and D using alternating gradient updates

$$\begin{aligned} & \min_{\mathbf{G}} \max_{\mathbf{D}} \left(E_{x \sim p_{data}} [\log \mathbf{D}(x)] + E_{\mathbf{z} \sim p(\mathbf{z})} [\log (1 - \mathbf{D}(\mathbf{G}(\mathbf{z})))] \right) \\ &= \min_{\mathbf{G}} \max_{\mathbf{D}} V(\mathbf{G}, \mathbf{D}) \end{aligned}$$

We are not minimizing any overall loss! No training curves to look at!

For t in 1, ... T:

1. (Update D) $\mathbf{D} = \mathbf{D} + \alpha_{\mathbf{D}} \frac{\partial V}{\partial \mathbf{D}}$
2. (Update G) $\mathbf{G} = \mathbf{G} - \alpha_{\mathbf{G}} \frac{\partial V}{\partial \mathbf{G}}$

Example code of training GAN

```
for epoch in range(opt.niter):
    for i, data in enumerate(dataloader, 0):
        #####
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        #####
        # train with real
        netD.zero_grad()
        real_cpu = data[0].to(device)
        batch_size = real_cpu.size(0)
        label = torch.full((batch_size,), real_label,
                           dtype=real_cpu.dtype, device=device)

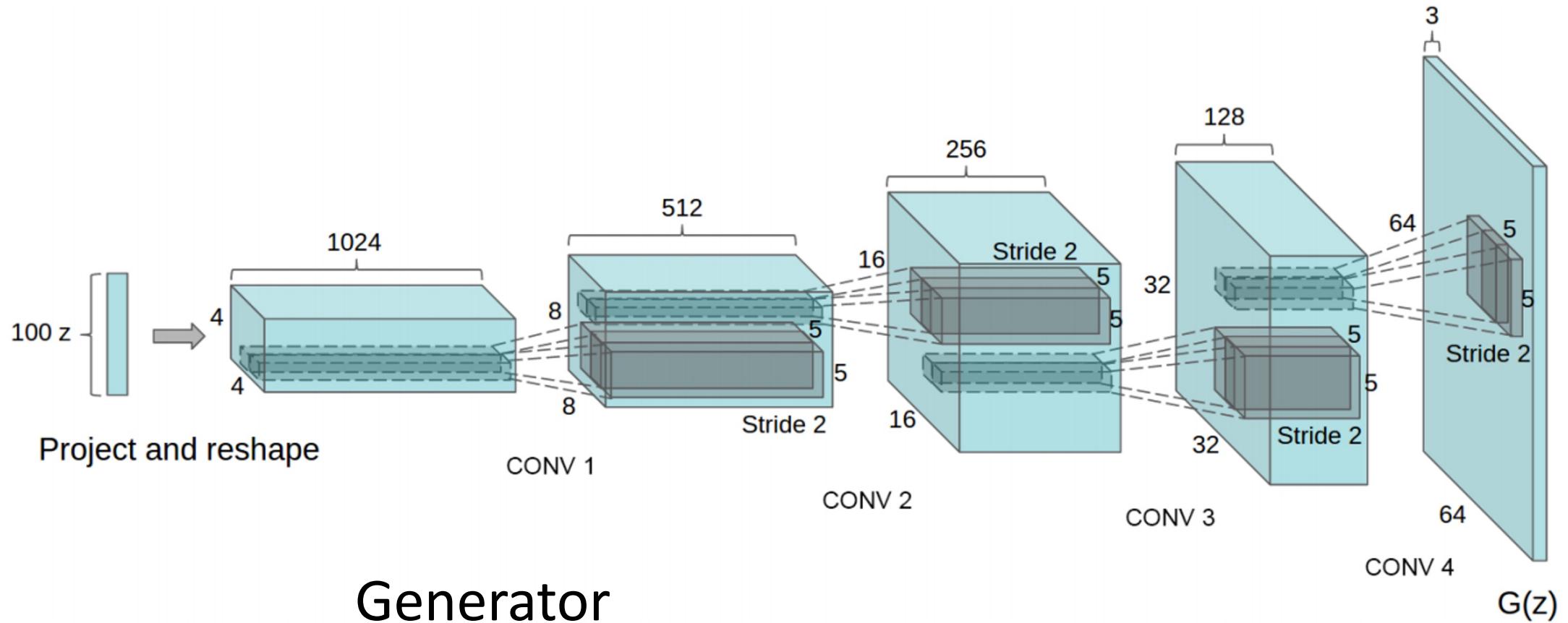
        output = netD(real_cpu)
        errD_real = criterion(output, label)
        errD_real.backward()
        D_x = output.mean().item()

        # train with fake
        noise = torch.randn(batch_size, nz, 1, 1, device=device)
        fake = netG(noise)
        label.fill_(fake_label)
        output = netD(fake.detach())
        errD_fake = criterion(output, label)
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        errD = errD_real + errD_fake
        optimizerD.step()
```

```
#####
# (2) Update G network: maximize log(D(G(z)))
#####
netG.zero_grad()
label.fill_(real_label) # fake labels are real for generator cost
output = netD(fake)
errG = criterion(output, label)
errG.backward()
D_G_z2 = output.mean().item()
optimizerG.step()
```

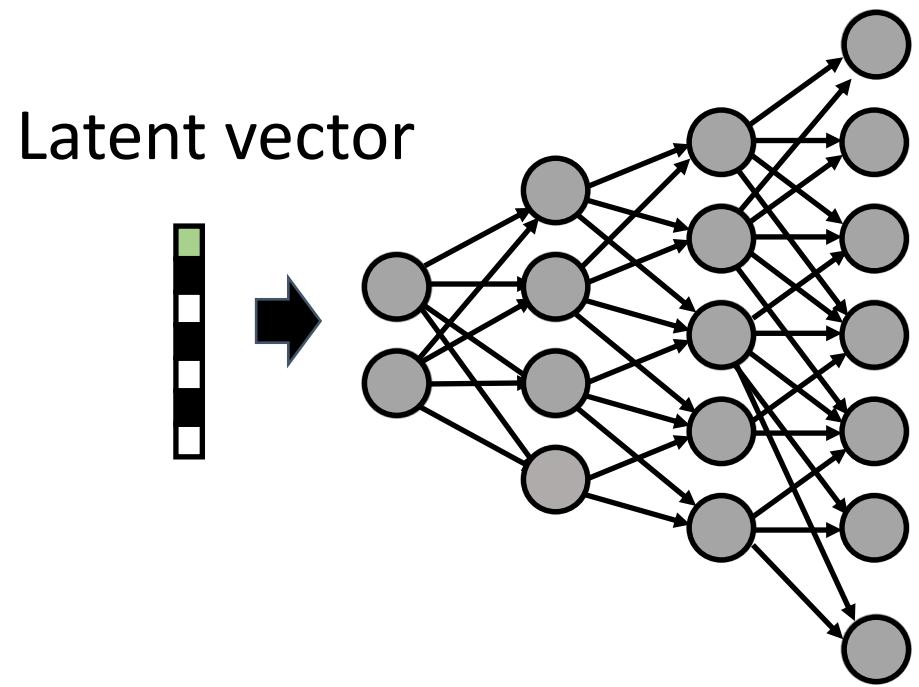
```
205     criterion = nn.BCELoss()
```

Generative Adversarial Networks: DC-GAN

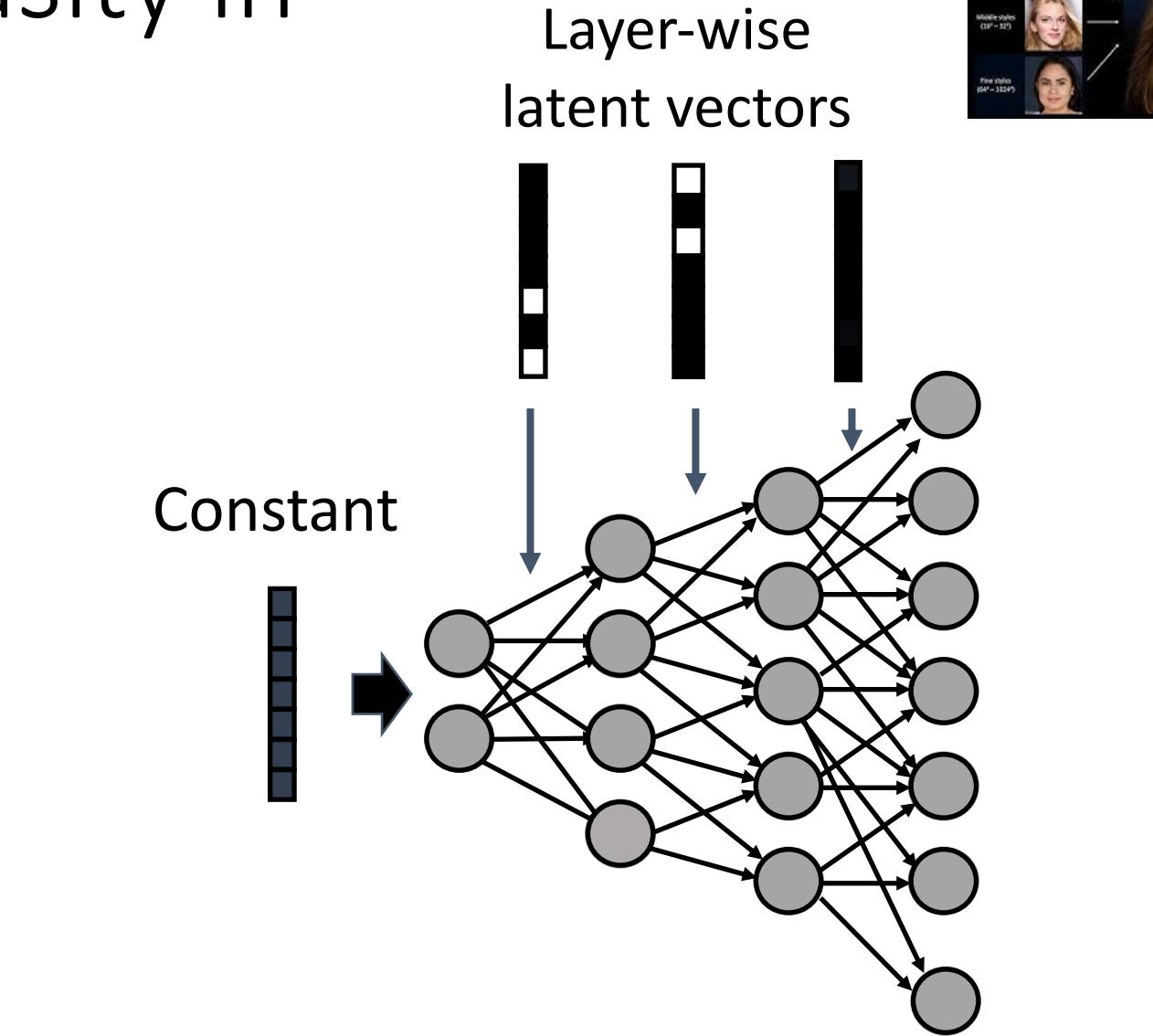


Radford et al, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks", ICLR 2016

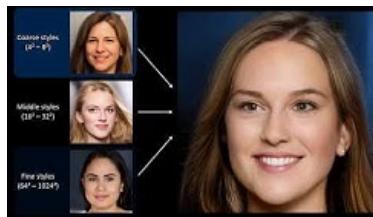
Layer-wise Stochasticity in StyleGAN



DC-GAN, PG-GAN

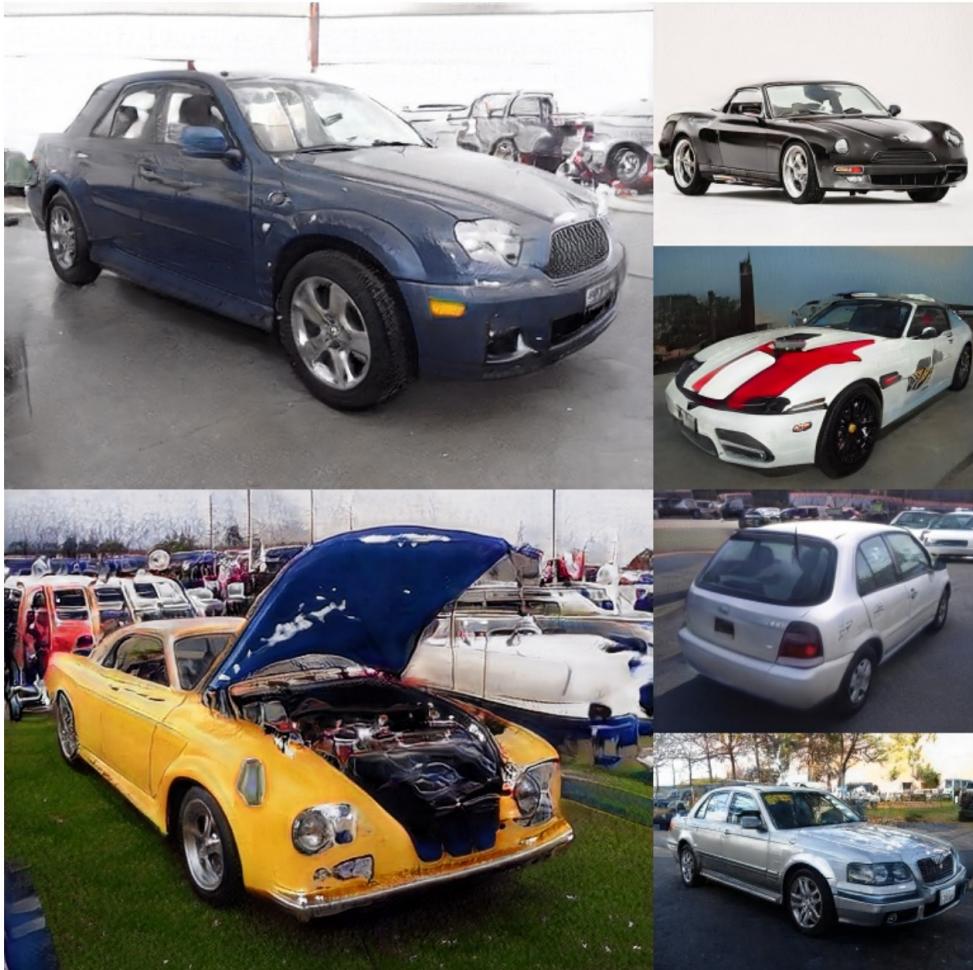


StyleGAN, StyleGANv2 [Karras et al]



GAN Improvements: StyleGAN for Higher Resolution

512 x 384 cars



1024 x 1024 faces

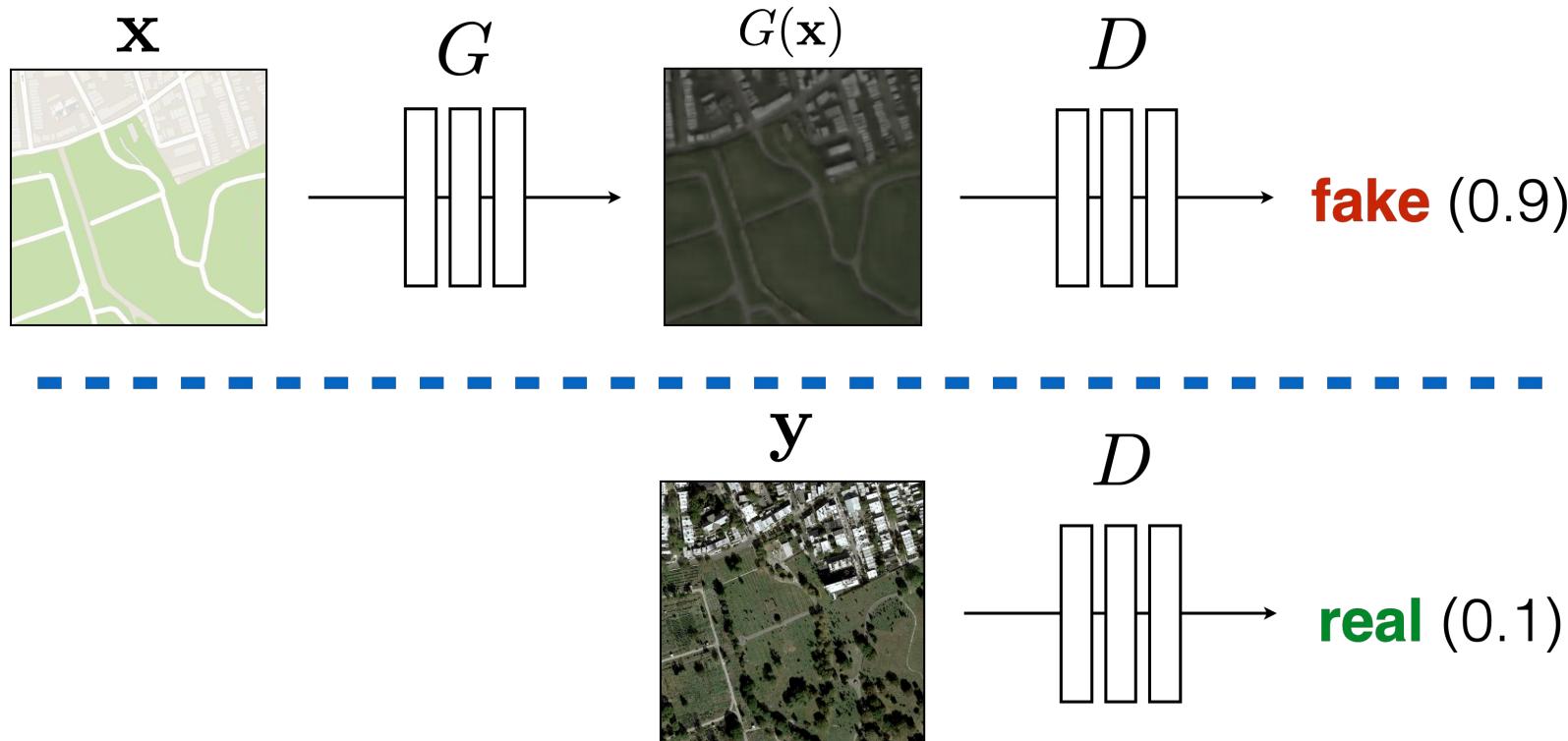


Karras et al, "A Style-Based Generator Architecture for Generative Adversarial Networks", CVPR 2019

[Images](#) are licensed under [CC BY-NC 4.0](#)

Image-to-Image Translation: Pix2Pix

Instead of input a random noise, we can input an image



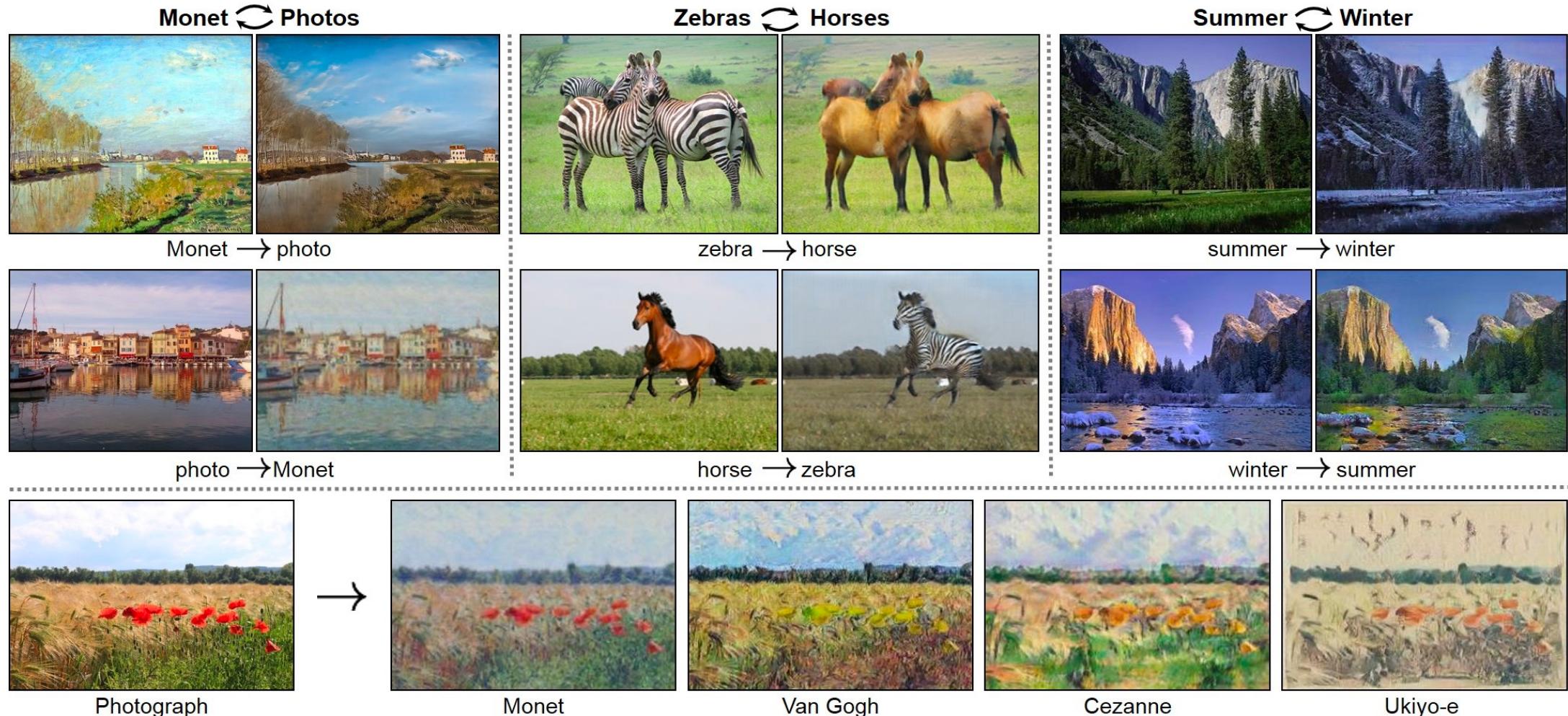
Philip Isola at MIT



$$\arg \max_D \mathbb{E}_{\mathbf{x}, \mathbf{y}} [\log D(G(\mathbf{x})) + \log(1 - D(\mathbf{y}))]$$

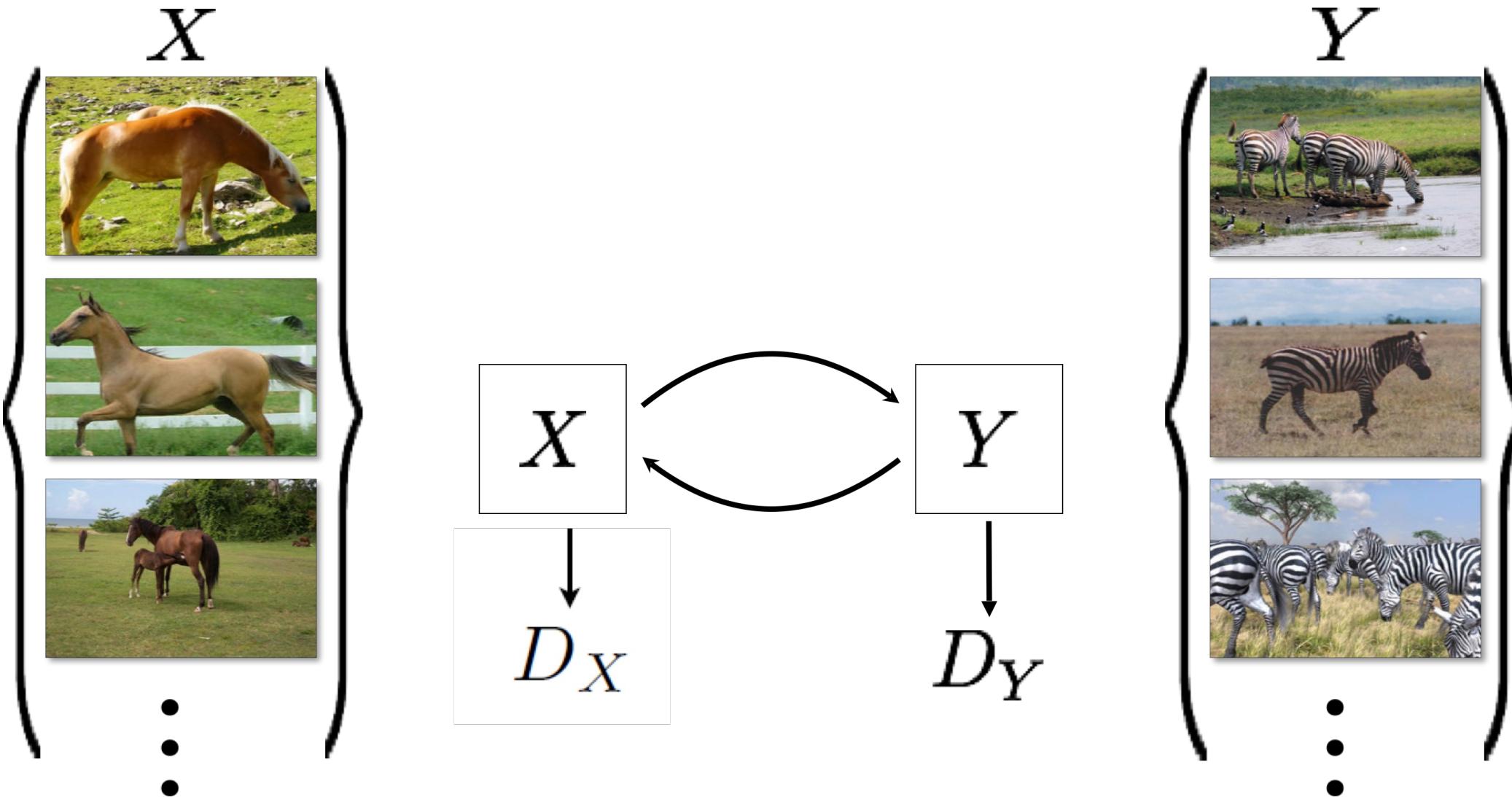
Isola et al, "Image-to-Image Translation with Conditional Adversarial Nets", CVPR 2017

Unpaired Image-to-Image Translation: CycleGAN

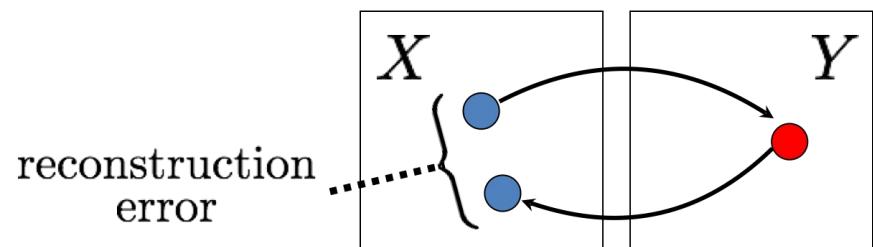
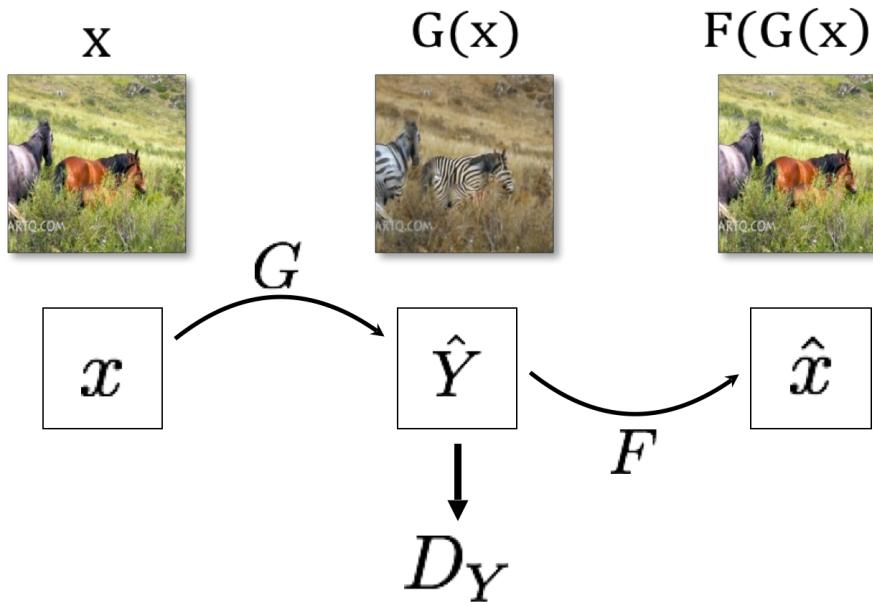


Zhu et al, "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks", ICCV 2017

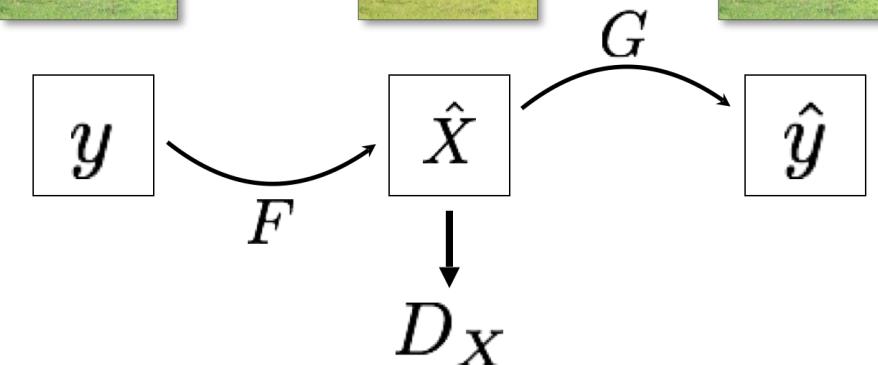
Unpaired Image-to-Image Translation: CycleGAN



Cycle Consistency Loss

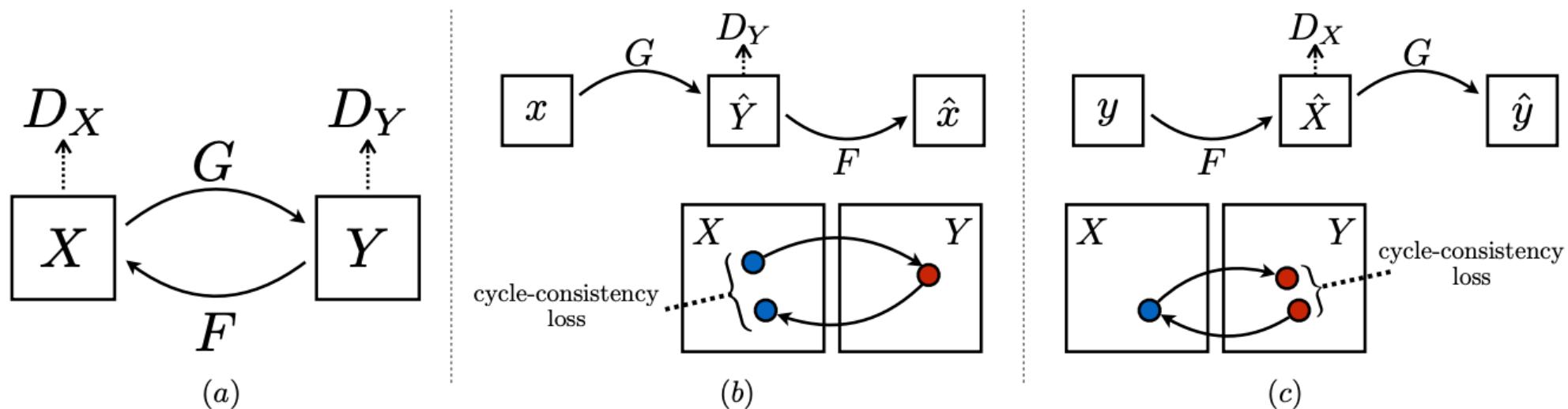


$$\|F(G(x)) - x\|_1$$



$$\|G(F(y)) - y\|_1$$

Cycle Consistency Loss: full objective

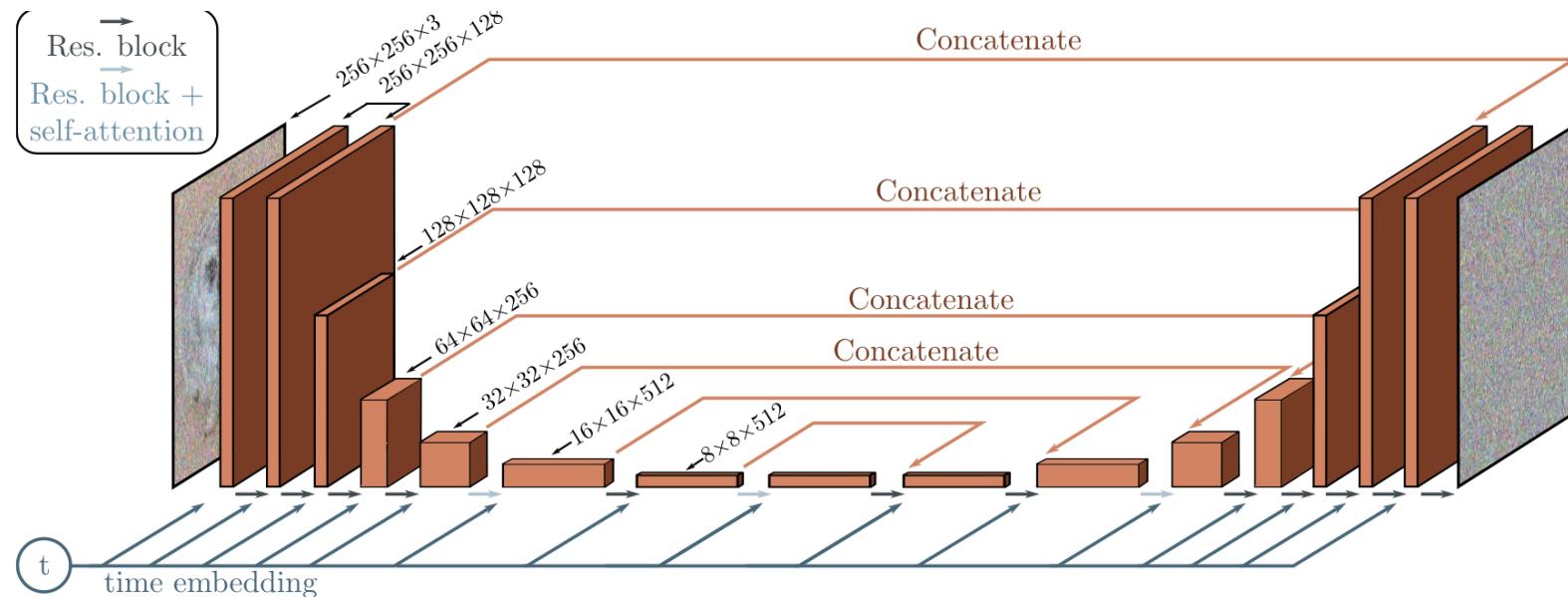
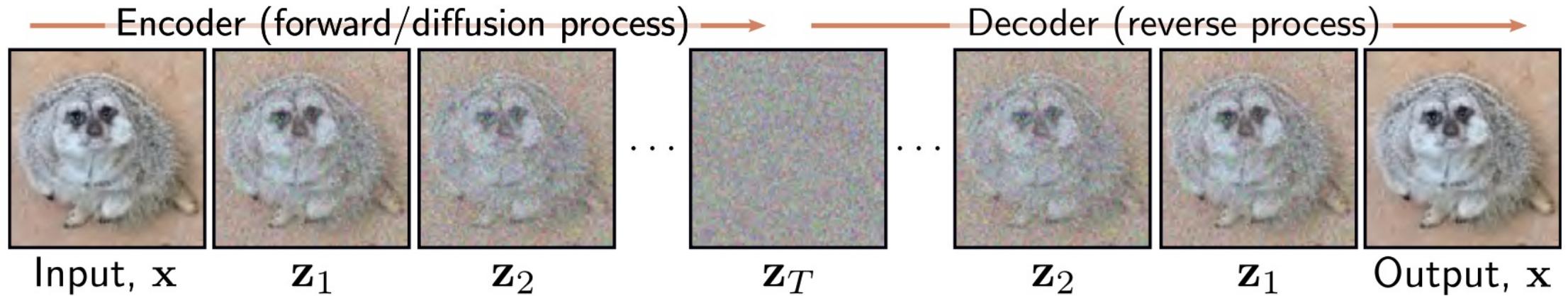


$$\begin{aligned}\mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) &= \mathbb{E}_{y \sim p_{\text{data}}(y)} [\log D_Y(y)] \\ &\quad + \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log(1 - D_Y(G(x)))]\end{aligned}$$

$$\begin{aligned}\mathcal{L}_{\text{cyc}}(G, F) &= \mathbb{E}_{x \sim p_{\text{data}}(x)} [\|F(G(x)) - x\|_1] \\ &\quad + \mathbb{E}_{y \sim p_{\text{data}}(y)} [\|G(F(y)) - y\|_1]\end{aligned}$$

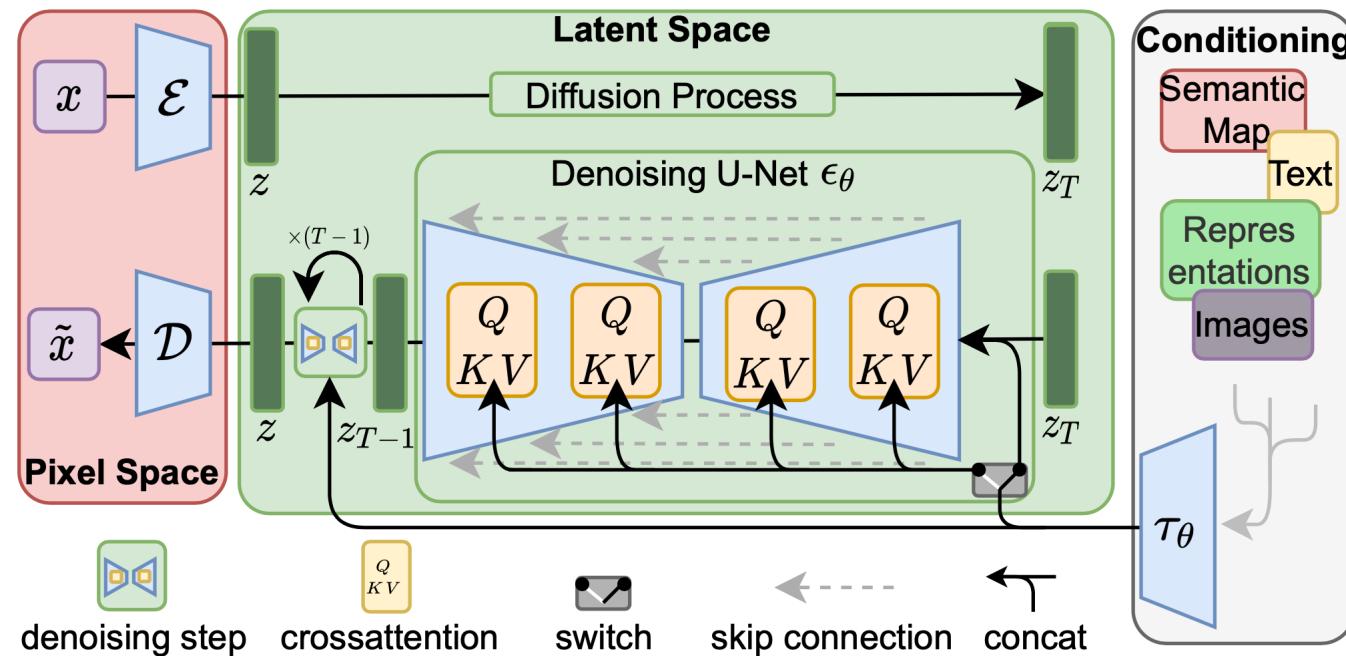
$$\begin{aligned}\mathcal{L}(G, F, D_X, D_Y) &= \mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) \\ &\quad + \mathcal{L}_{\text{GAN}}(F, D_X, Y, X) \\ &\quad + \lambda \mathcal{L}_{\text{cyc}}(G, F)\end{aligned}$$

Diffusion Models: U-Net used in image generation



Latent Diffusion Model

Run diffusion process in the latent space instead of pixel space, making training cost lower and inference speed faster

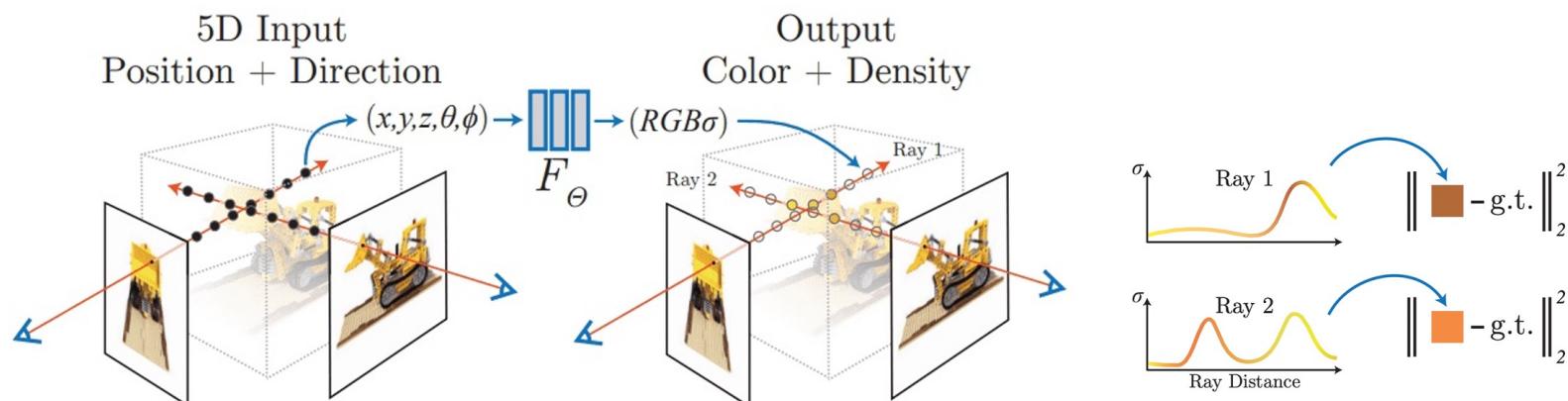


augmenting their underlying U-Net backbone with the cross-attention mechanism

Neural Rendering

What you need to know:

- The working principles of NeRF
- How to implement NeRF from scratch (see assignment 4)



For each pixel in each image, we know the ray that generated it, so

