CS163: Deep Learning for Computer Vision

Course Summary

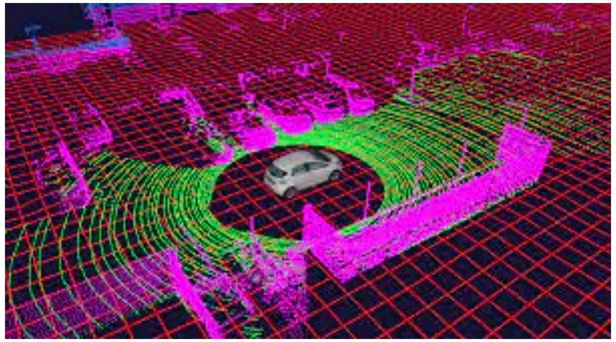
Deep Learning for Computer Vision

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

image

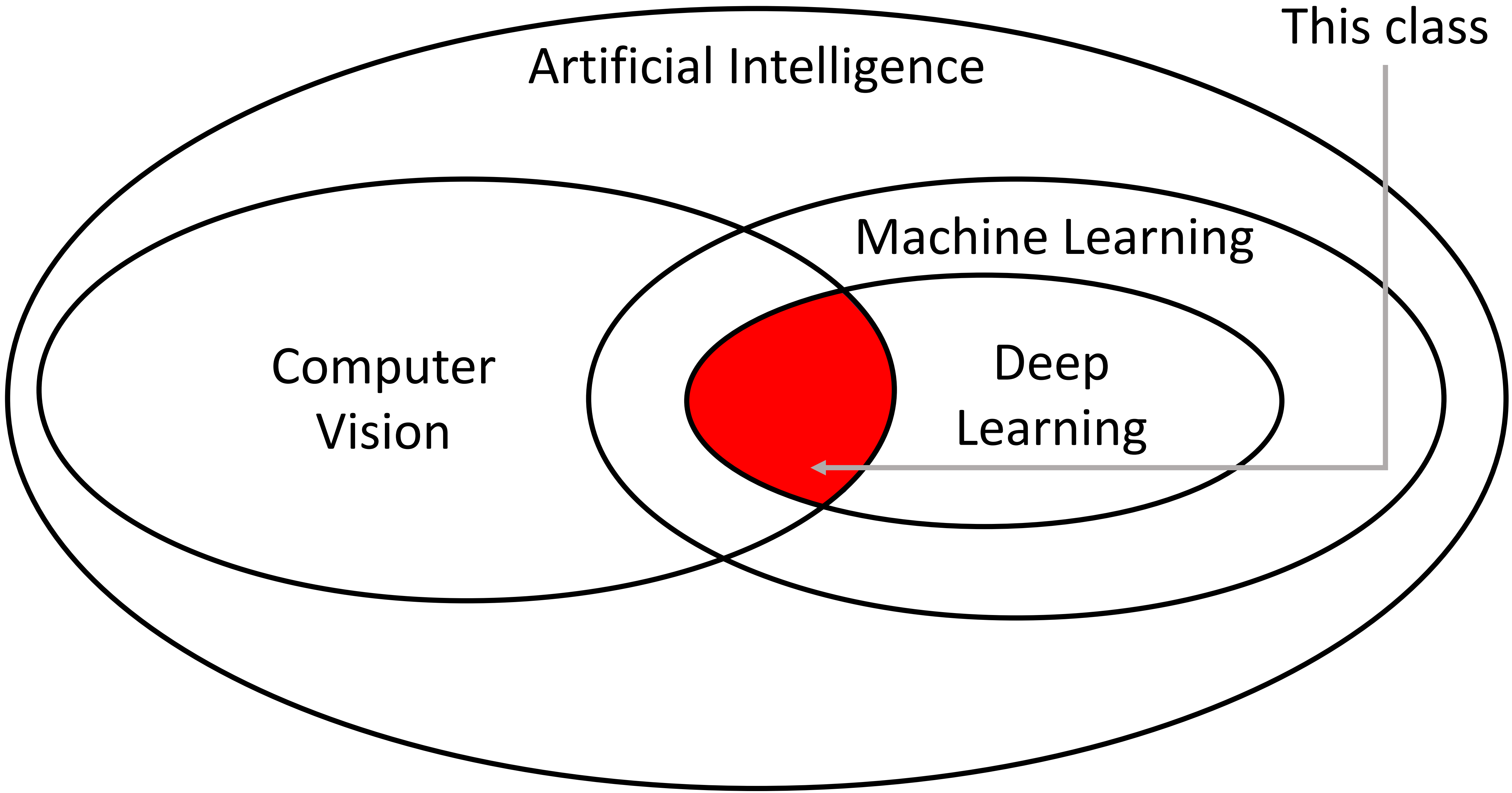
video

3D point clouds



Deep Learning for Computer Vision

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


This class
Artificial Intelligence
Machine Learning
Computer Vision
Deep Learning

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

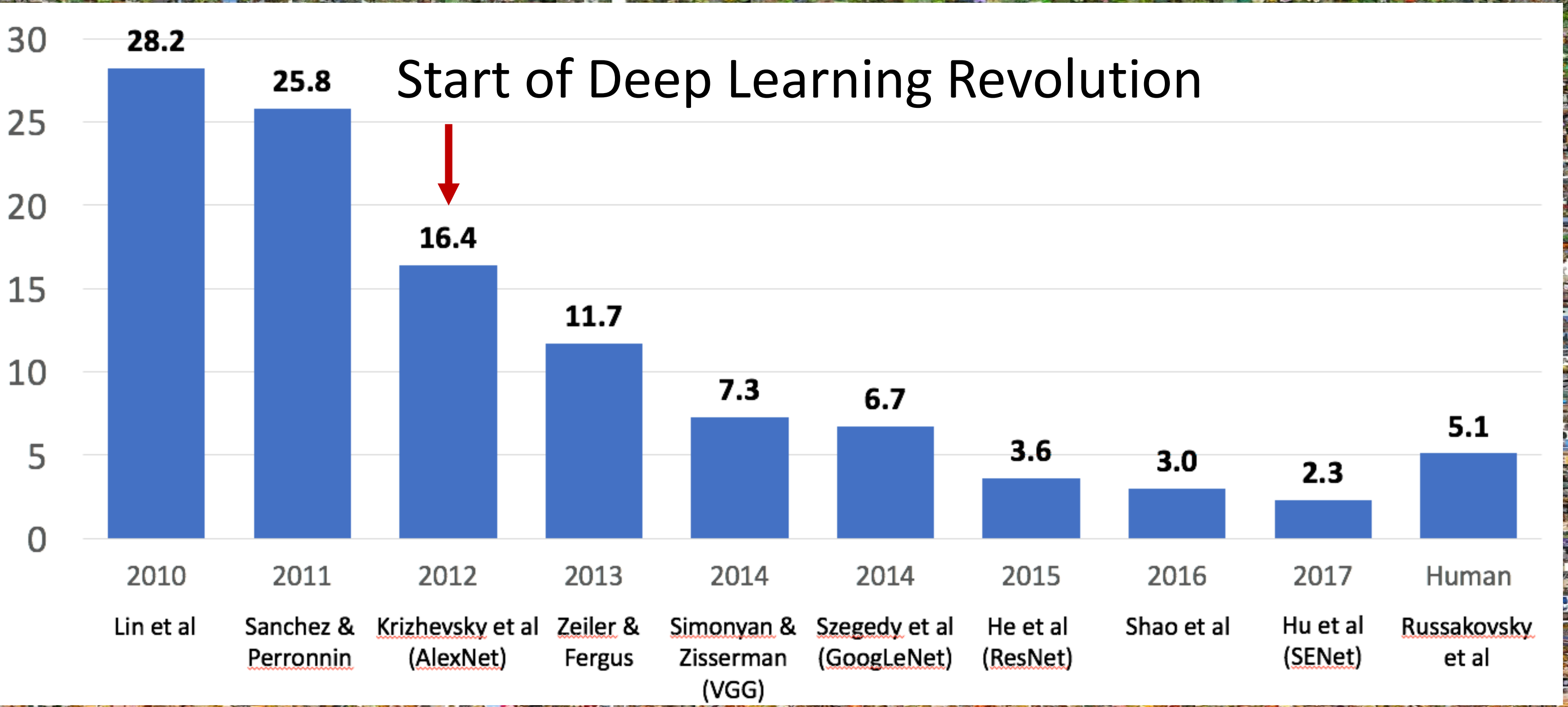
Output:
Scale
T-shirt
<u>Steel drum</u>
Drumstick
Mud turtle

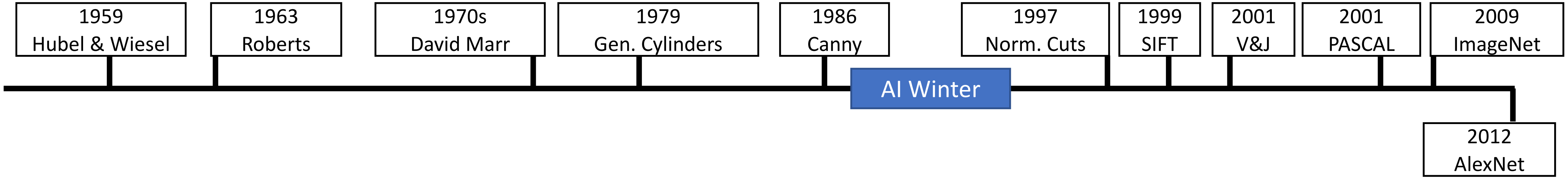
Deng et al, 2009
Russakovsky et al. IJCV 2015

1959	1963	1970s	1979	1986	1997	1999	2001	2001	2009
Hubel & Wiesel	Roberts	David Marr	Gen. Cylinders	Canny	Norm. Cuts	SIFT	V&J	PASCAL	ImageNet

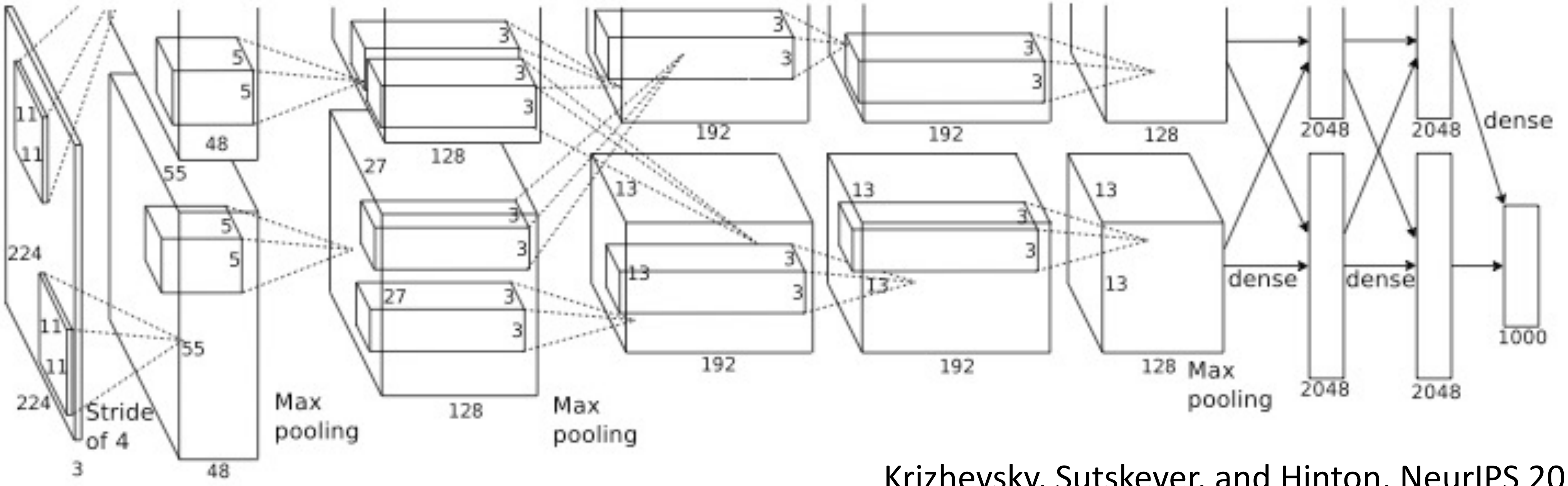
AI Winter

IMAGENET Large Scale Visual Recognition Challenge

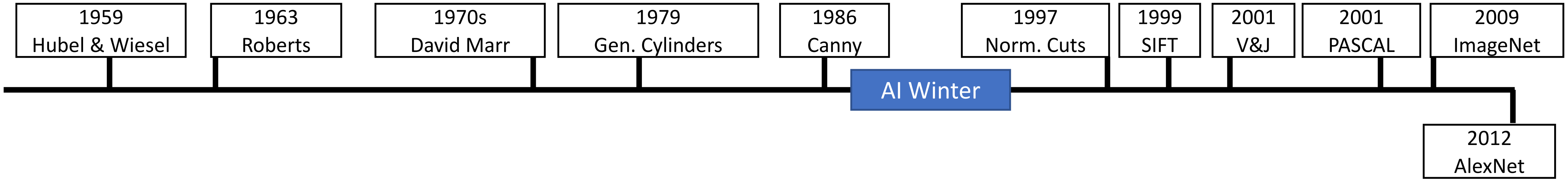




AlexNet: Deep Learning Approach for CV



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

$$\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial w_{ji}}$$

recognizable math

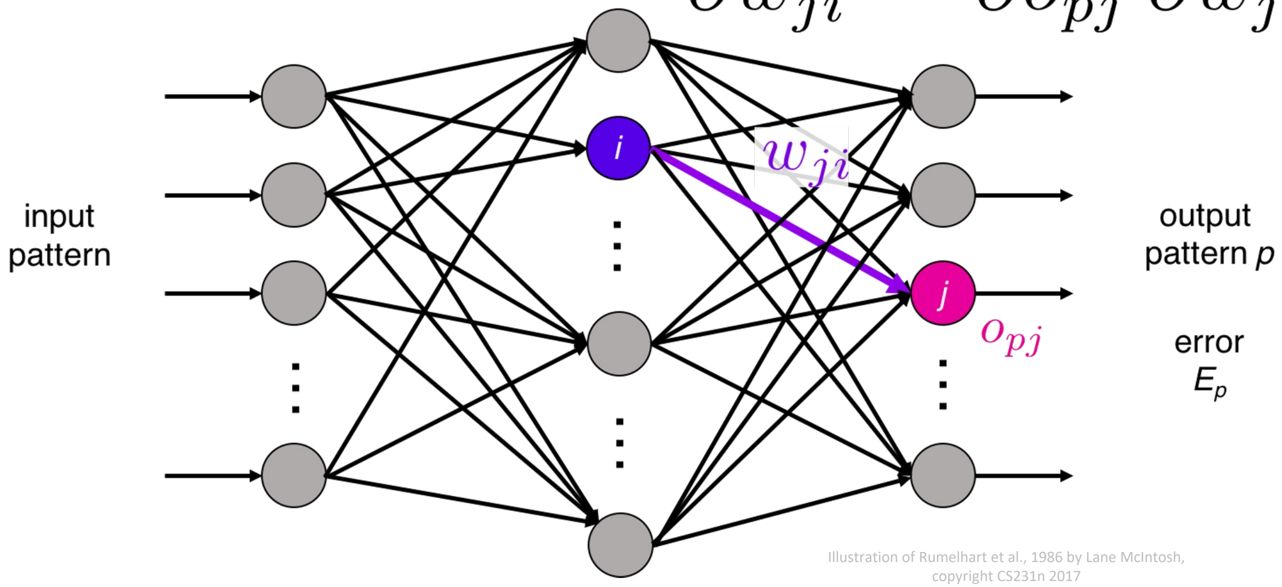
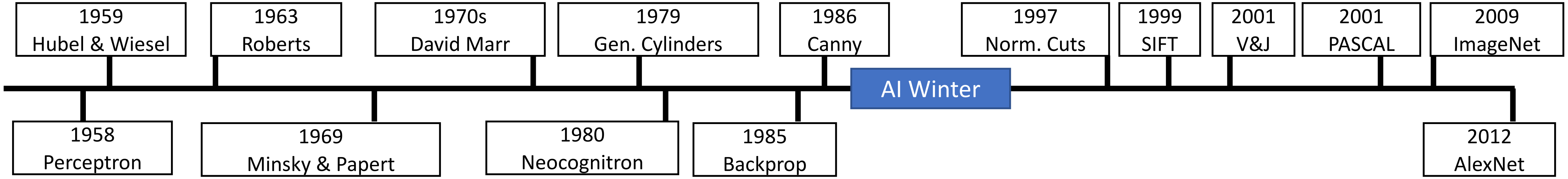
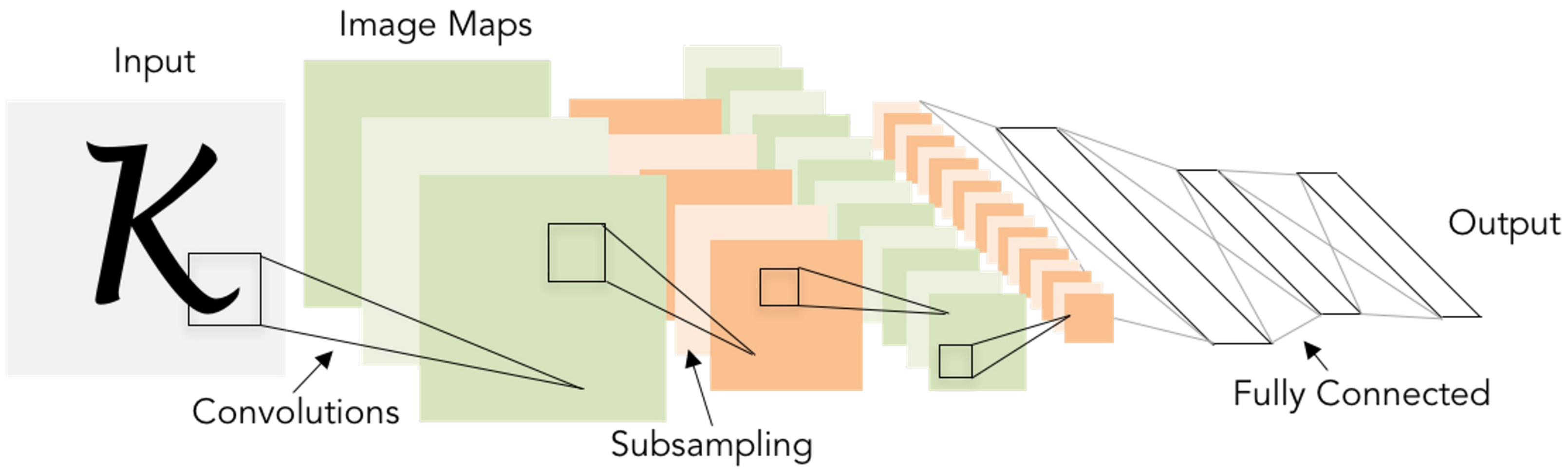


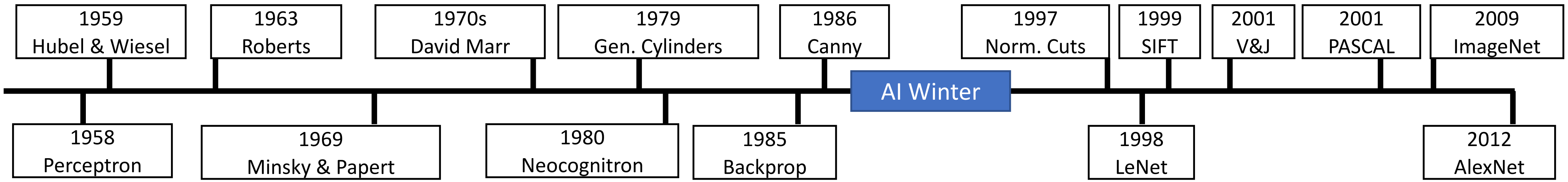
Illustration of Rumelhart et al., 1986 by Lane McIntosh, copyright CS231n 2017



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!

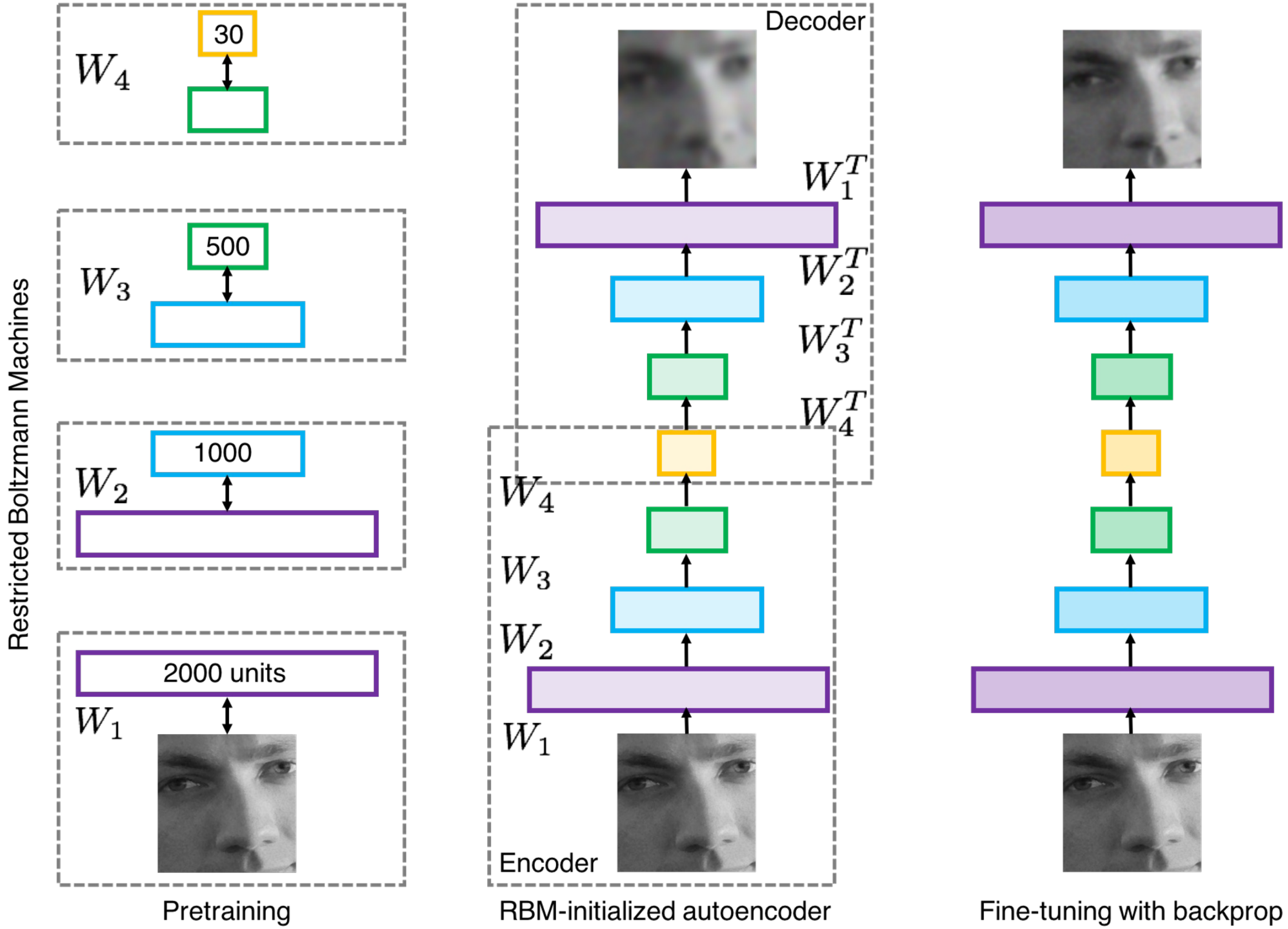


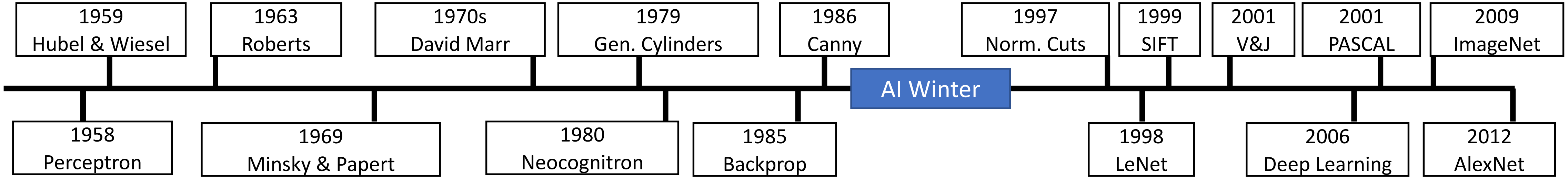
2000s: “Deep Learning”

People tried to train neural networks that were deeper and deeper

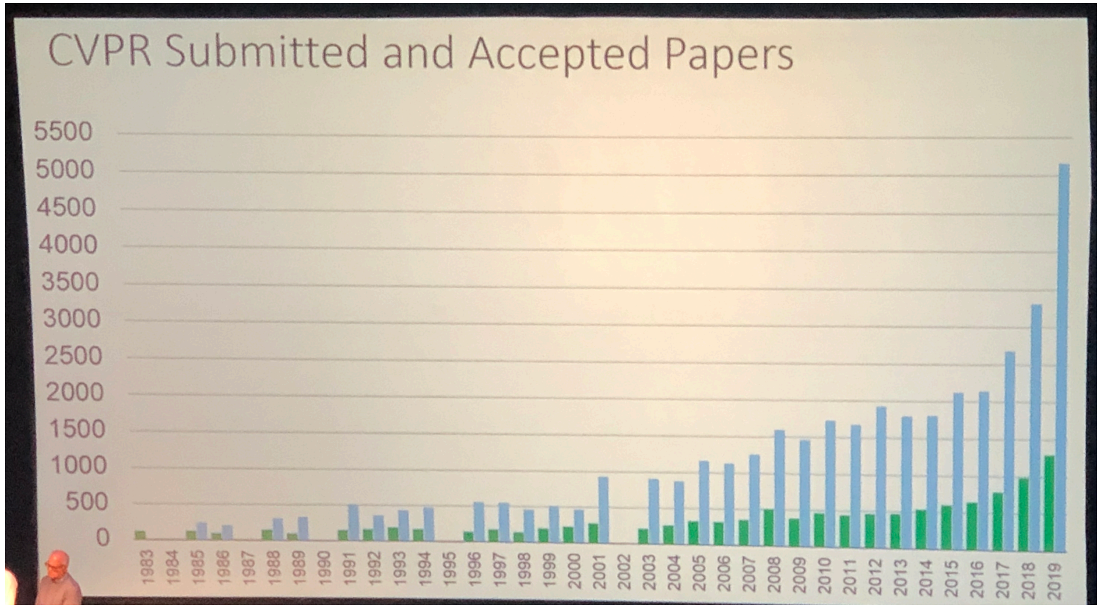
Not a mainstream research topic at this time

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

1958 Perceptron
1959 Hubel & Wiesel
1963 Roberts
1969 Minsky & Papert
1970s David Marr
1979 Gen. Cylinders
1980 Neocognitron
1985 Backprop
1986 Canny
AI Winter
1997 Norm. Cuts
1998 LeNet
1999 SIFT
2001 V&J
2001 PASCAL
2006 Deep Learning
2009 ImageNet
2012 AlexNet

2012 to Present: Neural Nets are everywhere

Image Classification

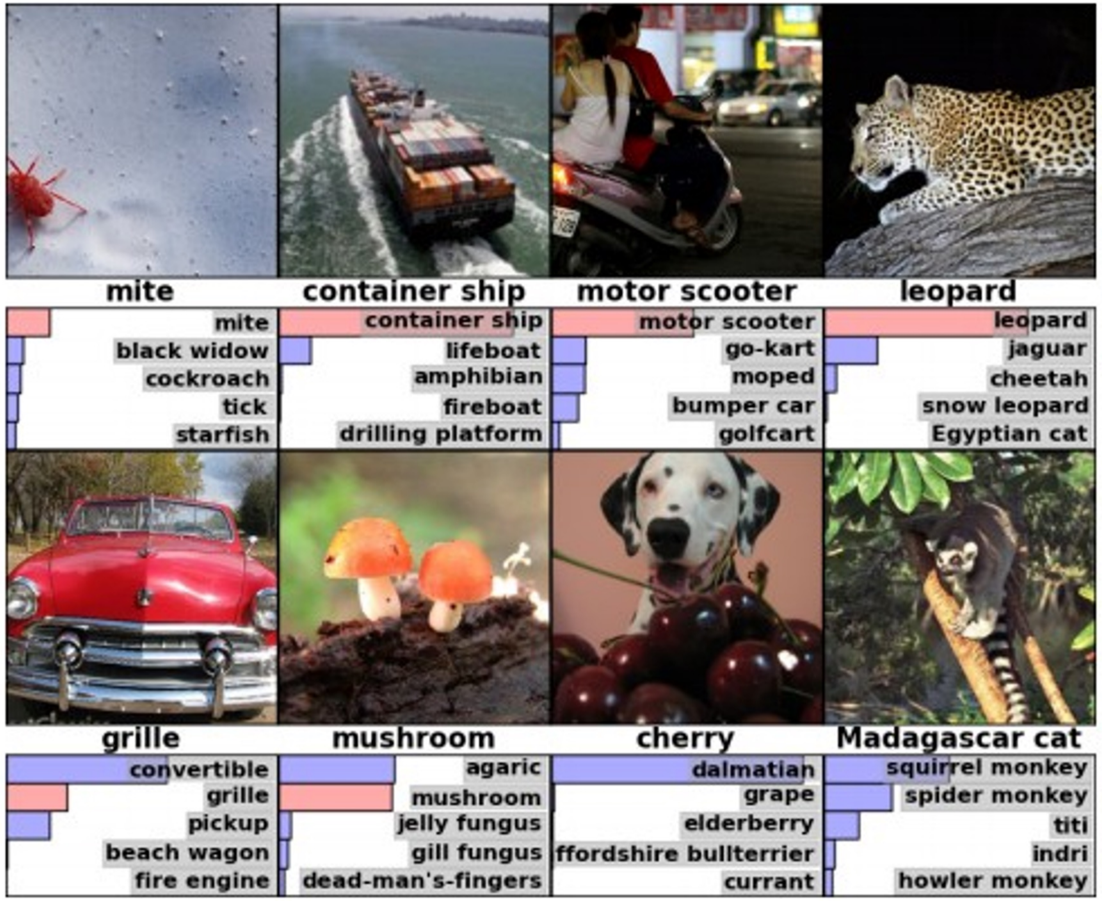
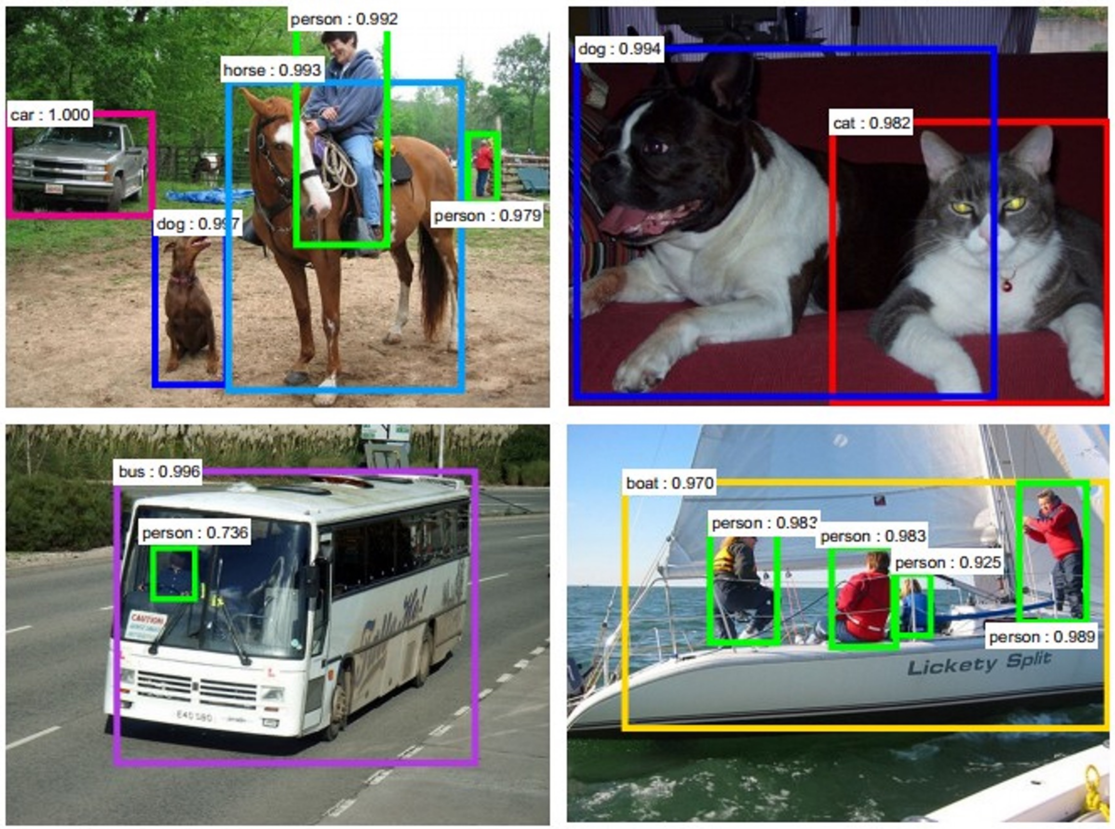


Image Retrieval

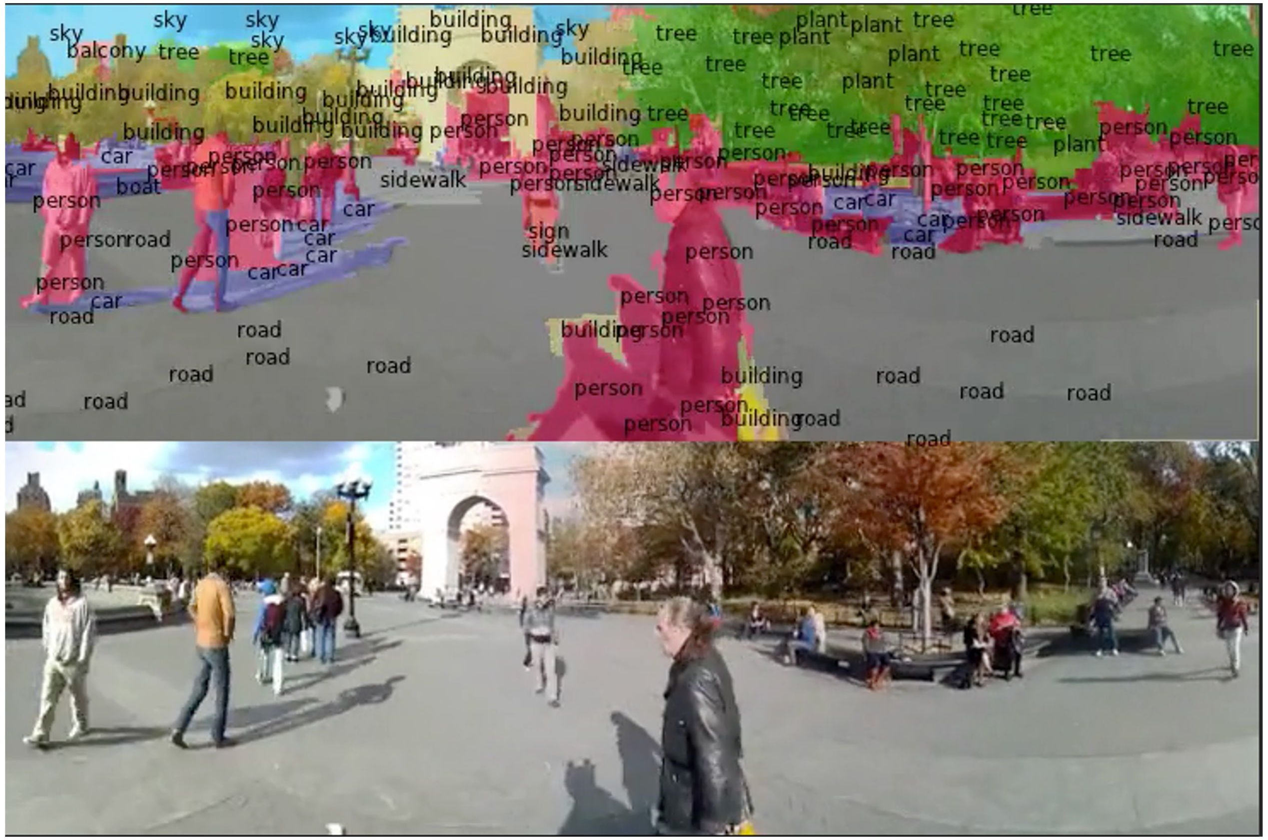
2012 to Present: Neural Nets are everywhere

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

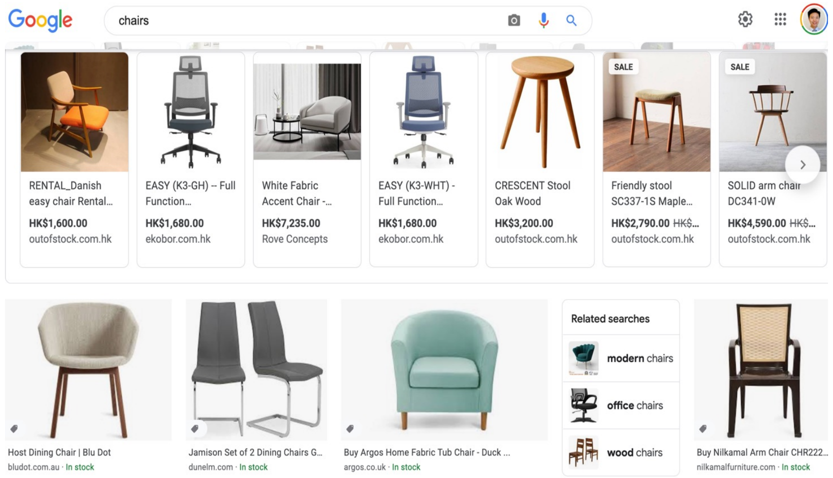
chair
bed
sofa
table
cabinet

Source image is from https://www.ikea.com/en/products/coffee-and-side-table/armchairs/ekenaset-art-30453028

Many challenges for image classification

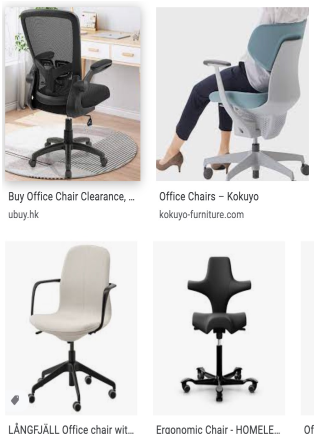
Challenges: Viewpoint Variation

Challenges: Intraclass Variation

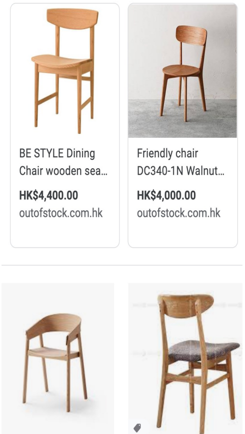


Challenges: Fine-Grained Categories

Office chairs



Wooden chairs



Avocado chairs?

Challenges: Objects in Scene Context

Occlusion, non-canonical view, clutters

Many challenges for image classification

Challenges: Domain Changes

Art clip

iStock

Drawing

shutterstock.com · 264375335

Photo

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

Chair

Sofa

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

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

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

Example training set

Highway

Playground

Mountain

Forest

Distance Metric to compare images

L1 distance:

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

$$\left| \underset{\text{test image}}{\begin{array}{|c|c|c|c|}\hline 56 & 32 & 10 & 18 \\ \hline 90 & 23 & 128 & 133 \\ \hline 24 & 26 & 178 & 200 \\ \hline 2 & 0 & 255 & 220 \\ \hline \end{array}} - \underset{\text{training image}}{\begin{array}{|c|c|c|c|}\hline 10 & 20 & 24 & 17 \\ \hline 8 & 10 & 89 & 100 \\ \hline 12 & 16 & 178 & 170 \\ \hline 4 & 32 & 233 & 112 \\ \hline \end{array}} \right| = \underset{\text{pixel-wise absolute value differences}}{\begin{array}{|c|c|c|c|}\hline 46 & 12 & 14 & 1 \\ \hline 82 & 13 & 39 & 33 \\ \hline 12 & 10 & 0 & 30 \\ \hline 2 & 32 & 22 & 108 \\ \hline \end{array}} \xrightarrow{\text{add}} 456$$

```python
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

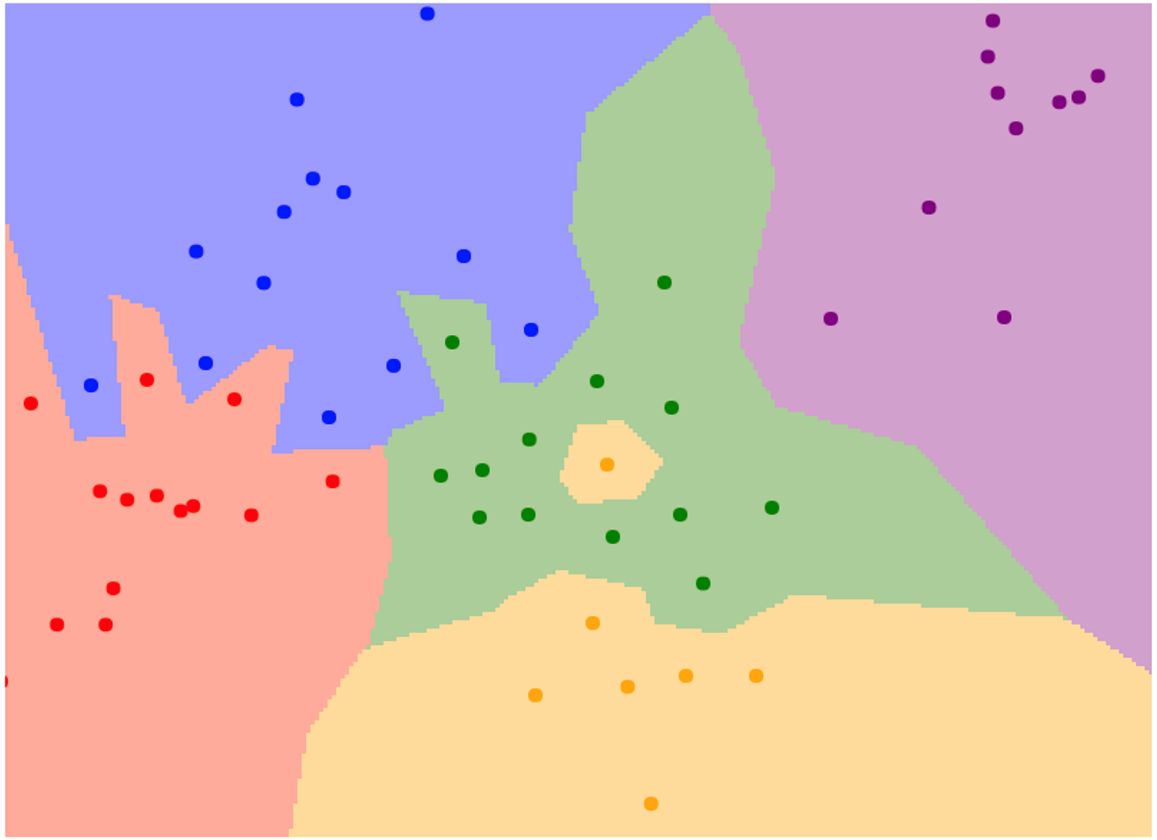
Memorize training data

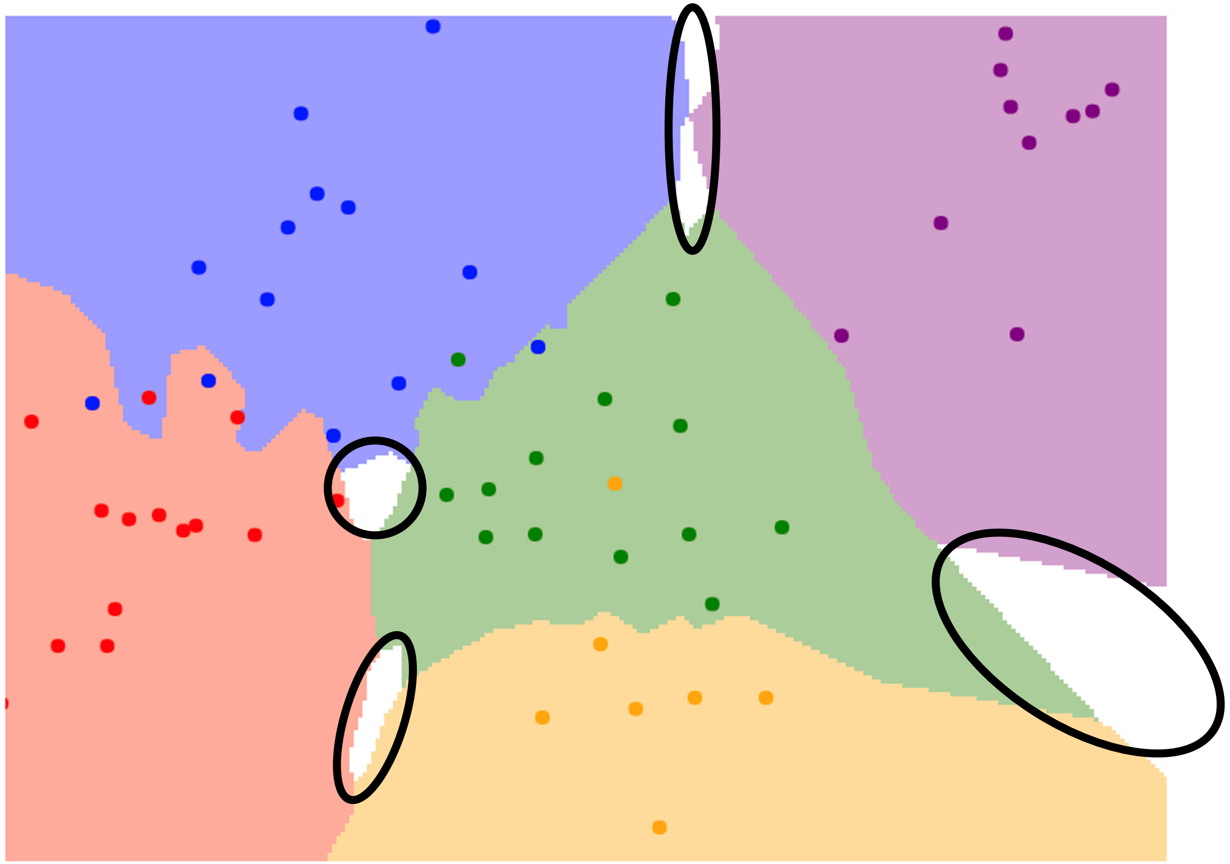
Nearest Neighbor Classifier

```python
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
```

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

K-Nearest Neighbors: hyper-parameter K

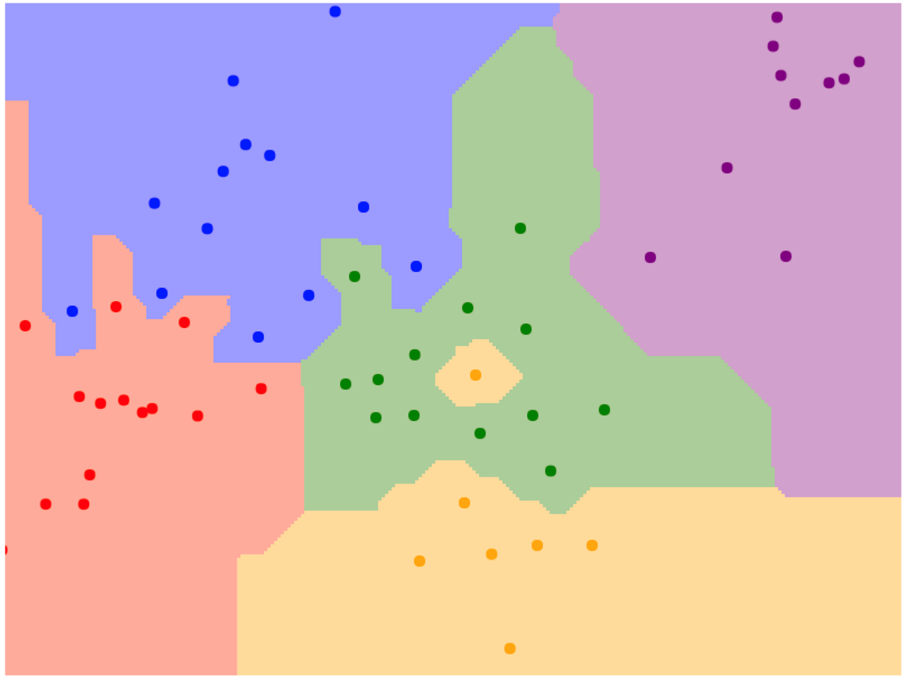




K-Nearest Neighbors: **Distance Metric**

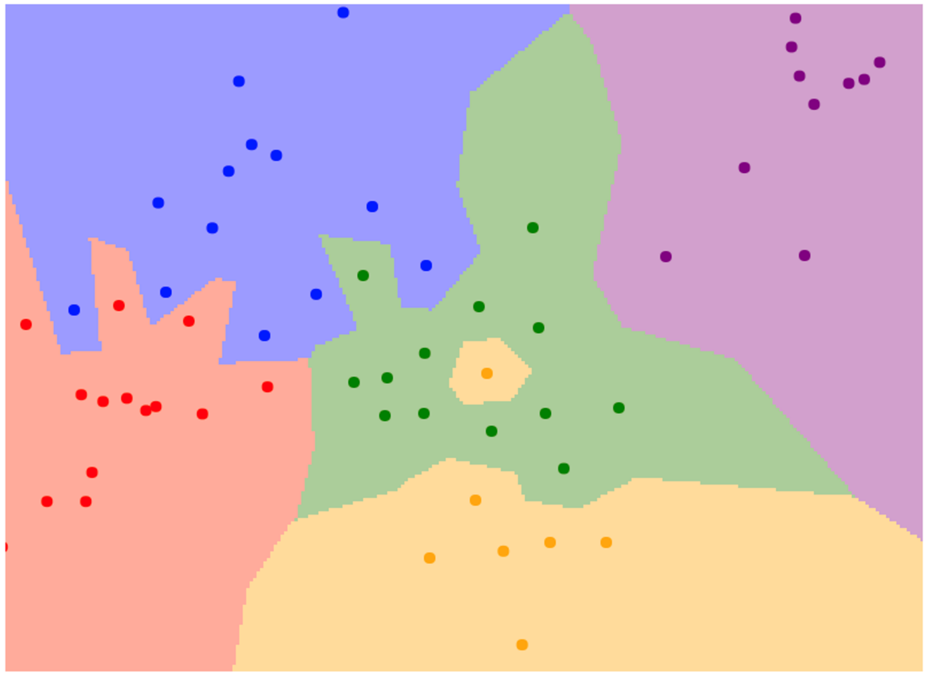
L1 (Manhattan) distance

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



L2 (Euclidean) distance

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



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

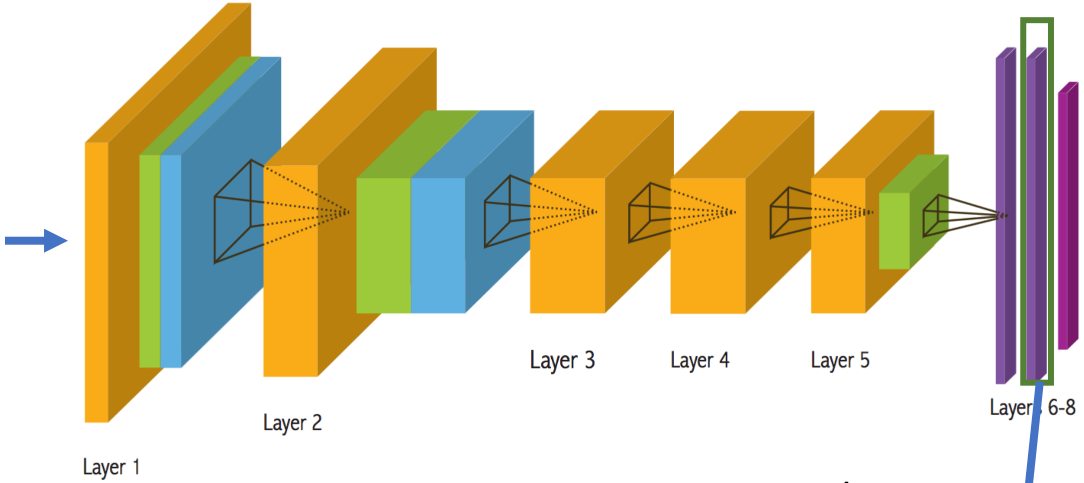
Your Dataset

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



Cafeteria (0.9)

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

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

Image

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

$\longrightarrow$ f(**x**,**W**) $\longrightarrow$ **20** numbers giving class scores

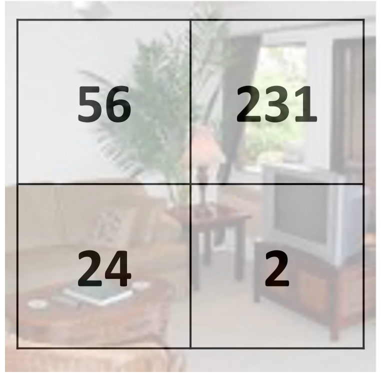
W

parameters
or weights

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

Stretch 4 pixels into a vector

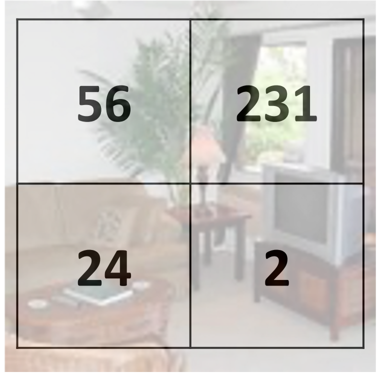
$f(x,W) = Wx + b$



Input image
(2, 2)

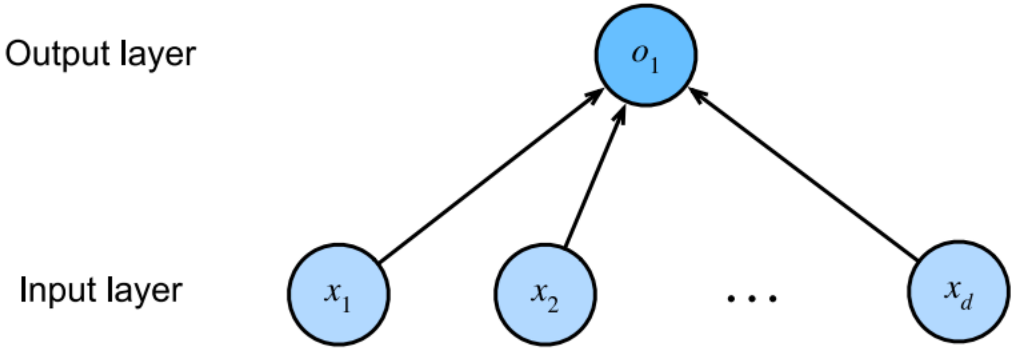
$$\underbrace{\begin{bmatrix} 0.2 & -0.5 & 0.1 & 2.0 \\ 1.5 & 1.3 & 2.1 & 0.0 \\ 0 & 0.25 & 0.2 & -0.3 \end{bmatrix}}_{W\ (3,\,4)} \underbrace{\begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix}}_{(4,)} + \underbrace{\begin{bmatrix} 1.1 \\ 3.2 \\ -1.2 \end{bmatrix}}_{b\ (3,)} = \underbrace{\begin{bmatrix} -96.8 \\ 437.9 \\ 61.95 \end{bmatrix}}_{(3,)}$$

Linear Classifier is a single-layer neural network

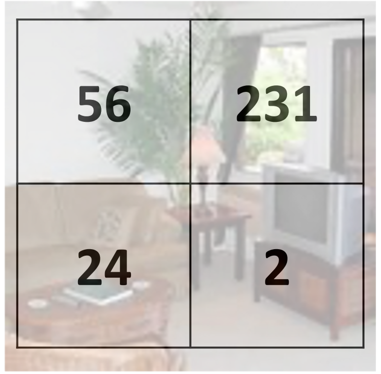


Outdoor-ness

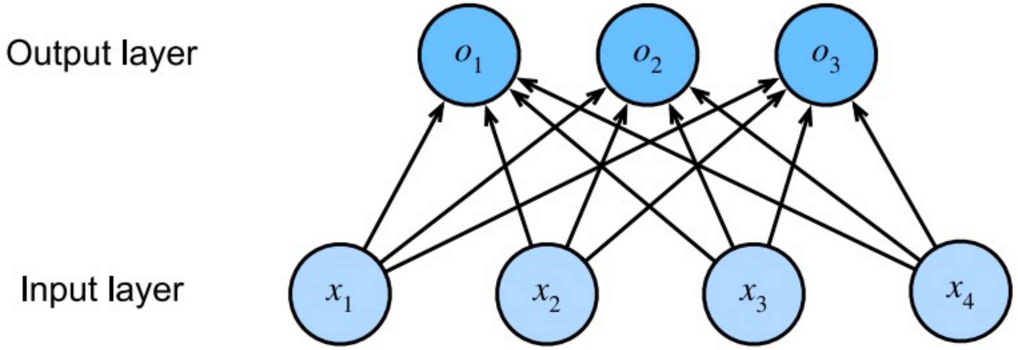
0.15



3 Scene Classification

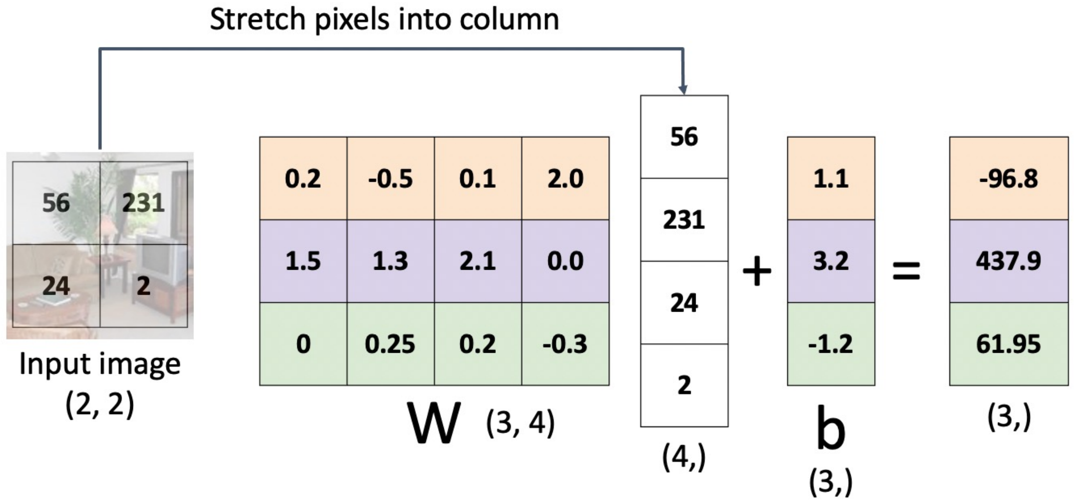


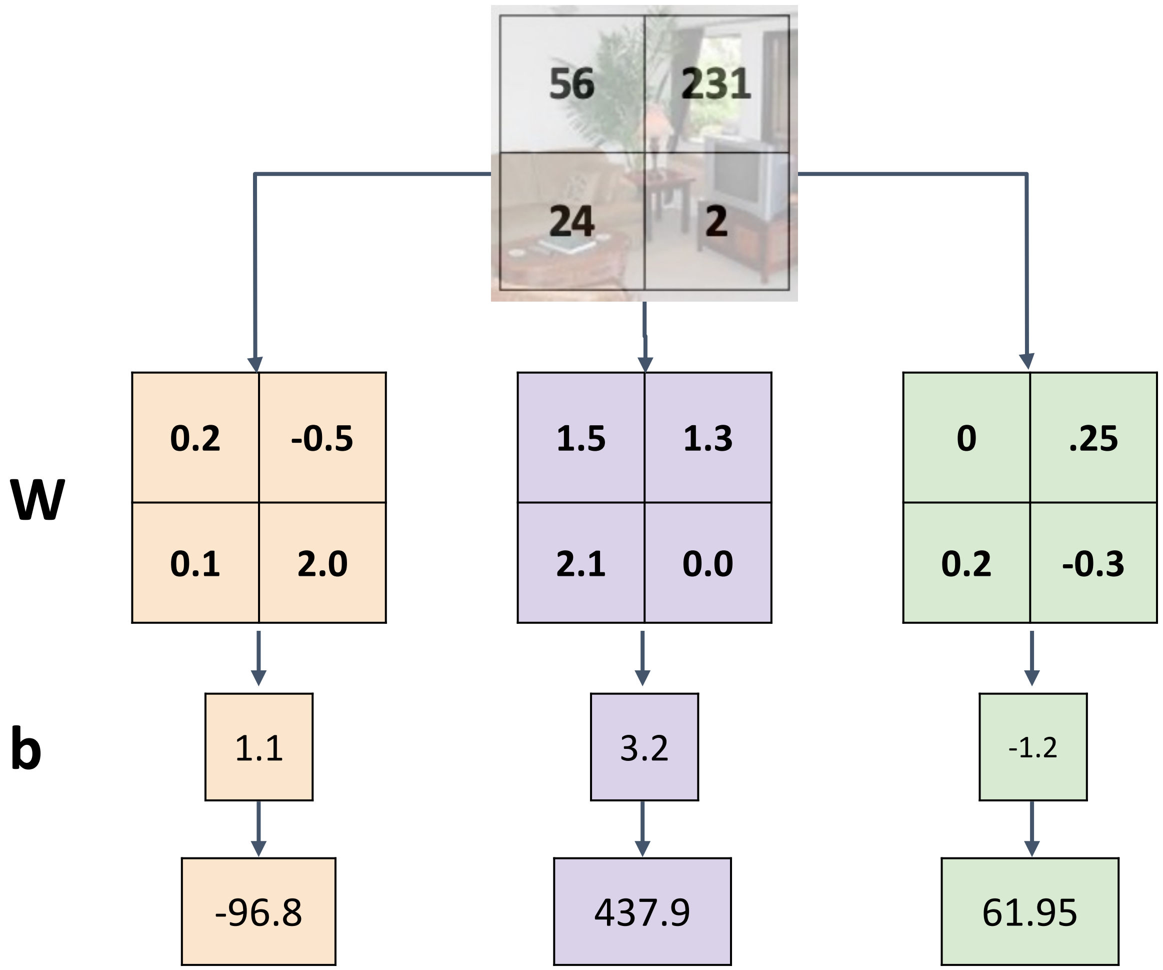
-96.8
437.8
62.0



Interpreting a Linear Classifier

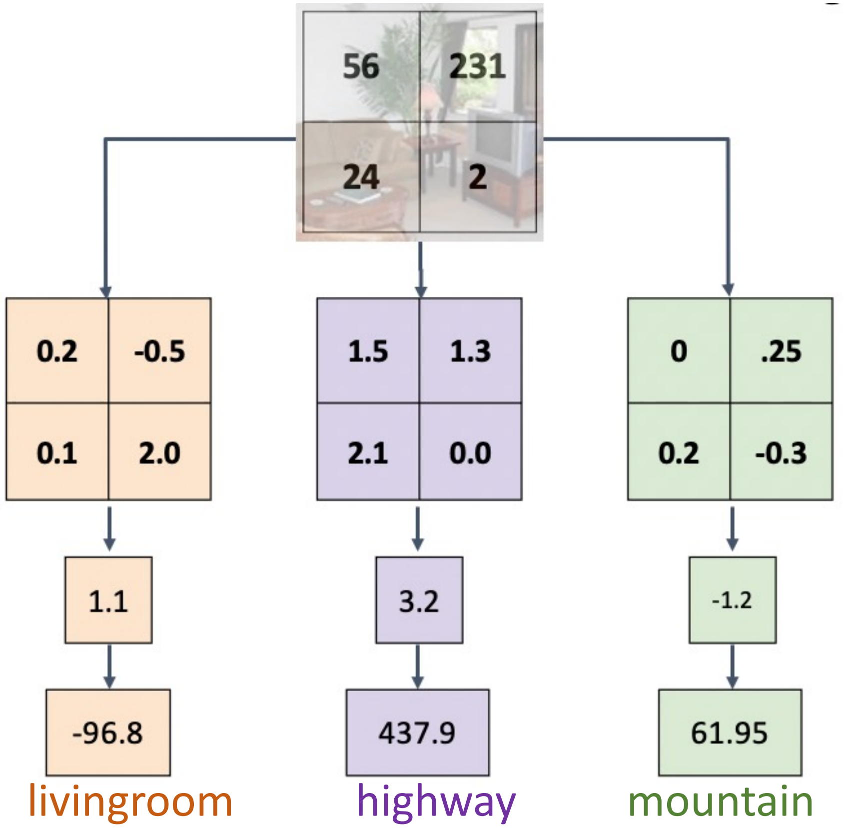
$f(x,W) = Wx + b$



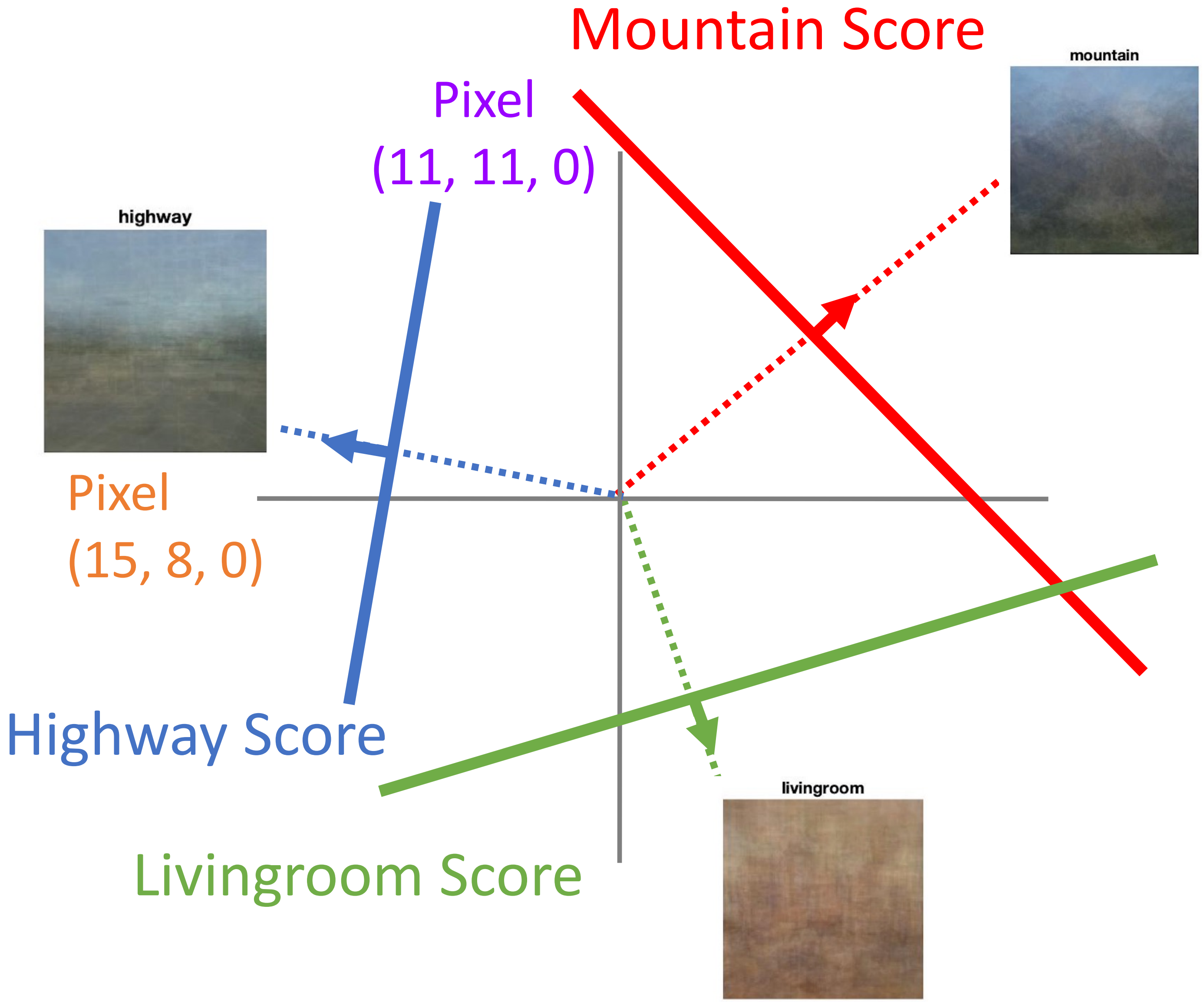


Interpreting a Linear Classifier

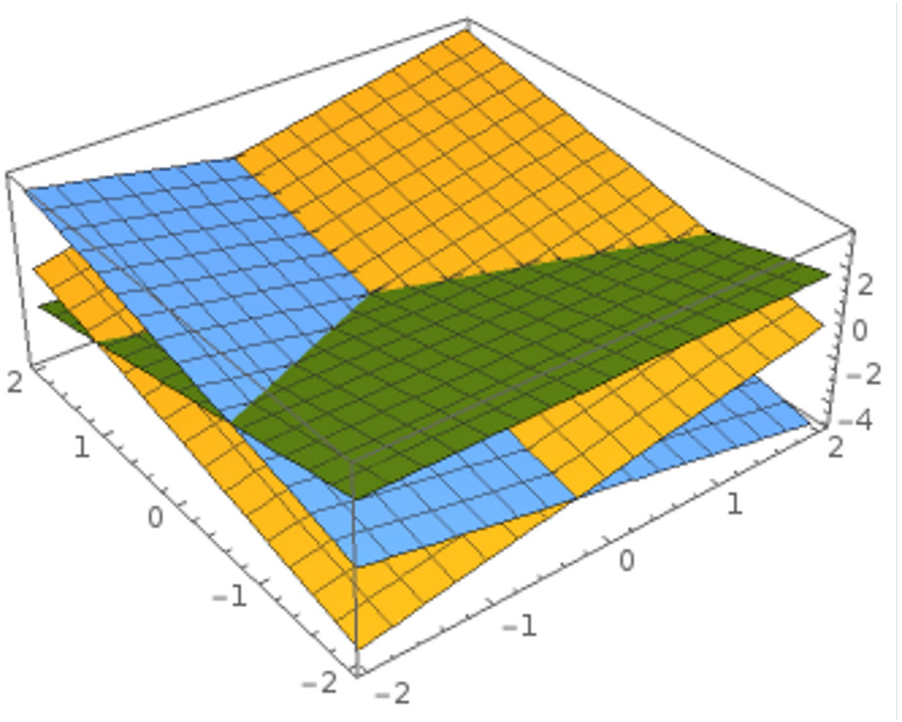
Linear classifier has one “template” per category, then compute the correlation.
Drawback: A single template cannot capture multiple modes of the data



Interpreting a Linear Classifier: <u>Geometric Viewpoint</u>



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

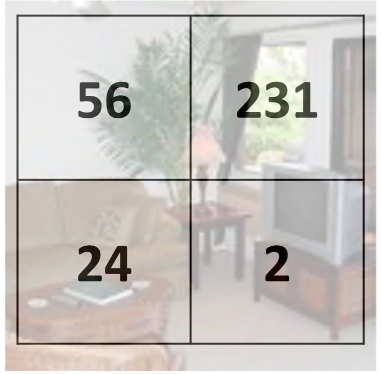
$$L_i(f(x_i, W), y_i)$$

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

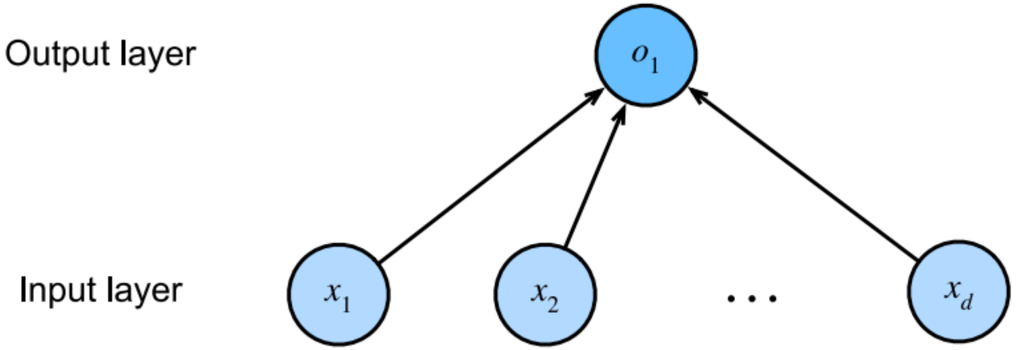
$$s = f(x_i; W)$$



56
231
24
2

Outdoor-ness

0.15



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: $W^* = \underset{W}{\operatorname{argmin}}\, L(W)$

Linear Regression Loss

		Model output	**Loss**	**Ground truth**
	→	→ **0.15**	$\frac{1}{2}(0.15-1)^2$	**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**

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

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

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

Probabilities must be >= 0

Probabilities must sum to 1

	Unnormalized log-probabilities / logits	$\xrightarrow{\text{exp}}$	unnormalized probabilities	$\xrightarrow{\text{normalize}}$	probabilities
Bridge	**3.2**		**24.5**		**0.13**
Mountain	5.1		164.0		0.87
Coast	-1.7		0.18		0.00

$L_i = -\log(0.13)$
$= \mathbf{2.04}$

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**

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

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

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

Probabilities must be >= 0

Probabilities must sum to 1

	Unnormalized log-probabilities / logits		unnormalized probabilities		probabilities		Correct probs
Bridge	**3.2**	exp →	**24.5**	normalize →	**0.13**	→ Compare ←	**1.00**
Mountain	5.1		164.0		0.87		0.00
Coast	-1.7		0.18		0.00		0.00

Kullback–Leibler divergence

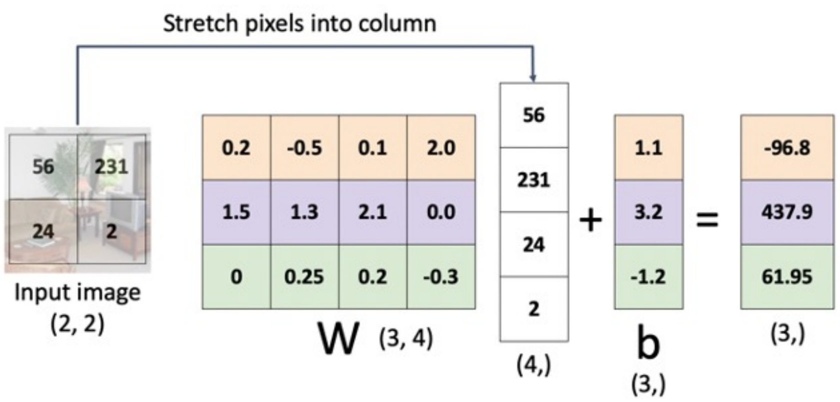
$$D_{KL}(\boldsymbol{P} \,||\, \boldsymbol{Q}) = \sum_y \boldsymbol{P}(y) \log \frac{\boldsymbol{P}(y)}{\boldsymbol{Q}(y)}$$

Linear NN/Classifier: Three Viewpoints

Equation Viewpoint

$f(x,W) = Wx$

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

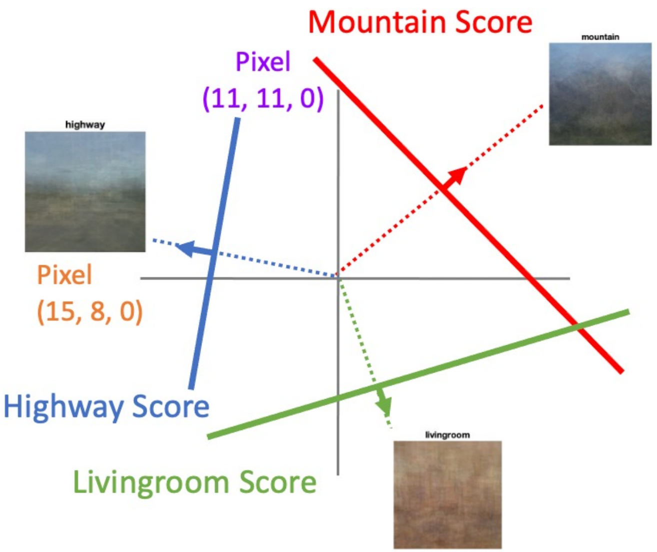


Visualization Viewpoint

One template per class

Geometric Viewpoint

Hyperplanes
cutting up space



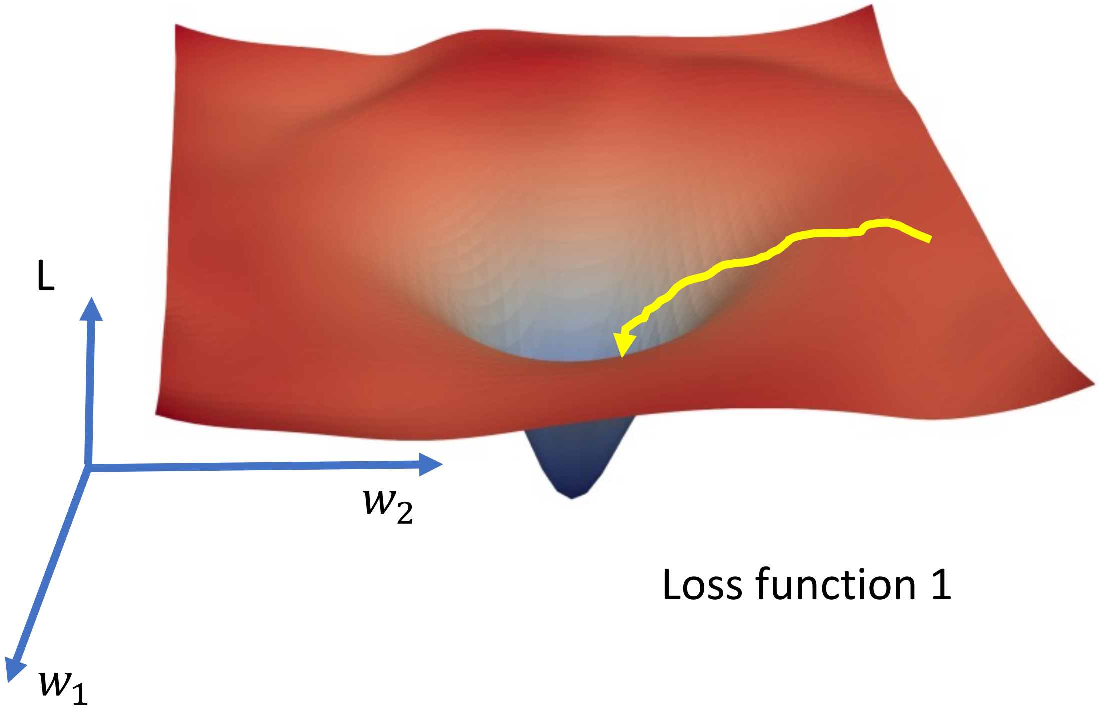
Lecture 4: optimization algorithms

Optimization

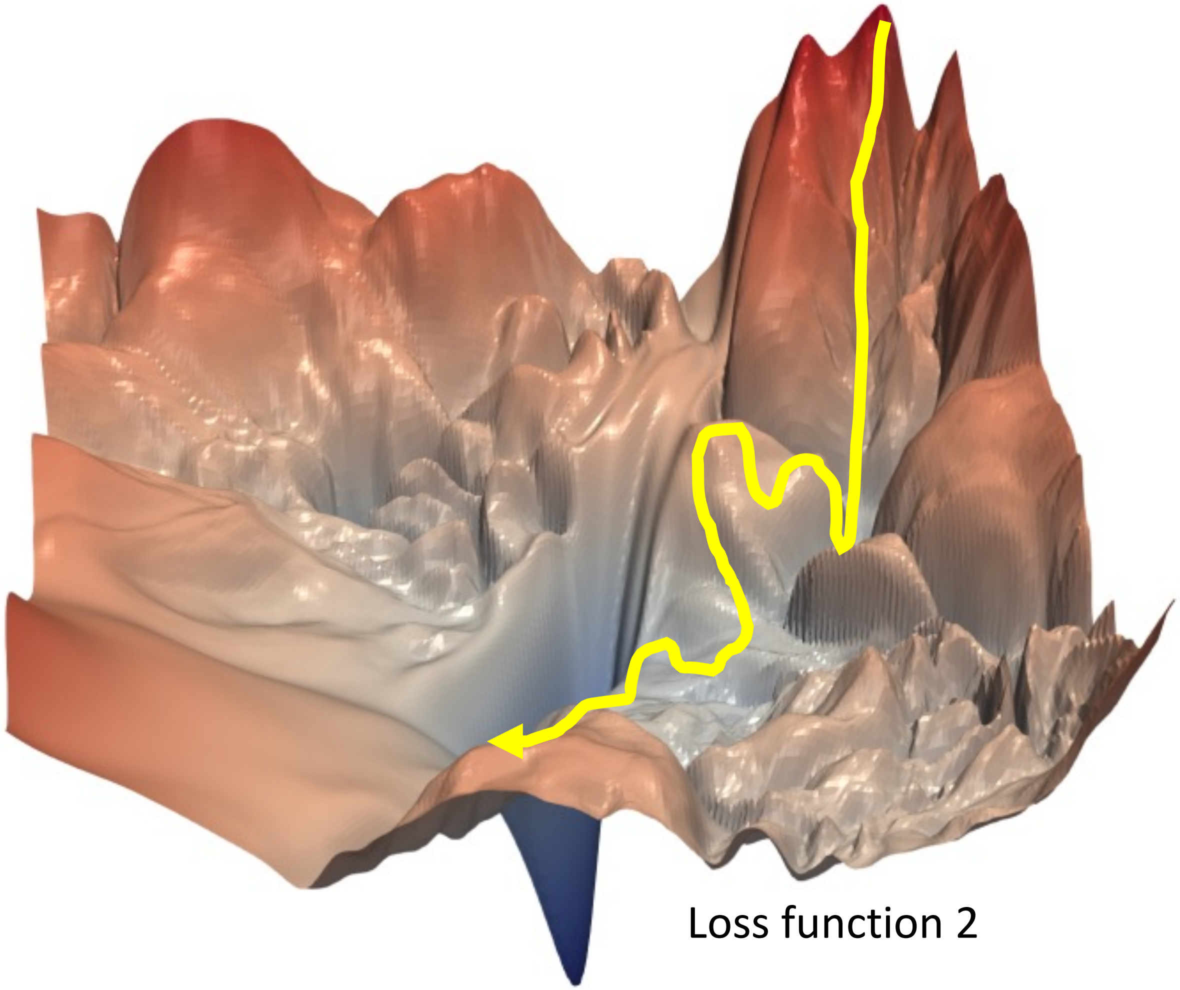
$$w^* = \arg\min_{w} L(w)$$

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

$$w^* = \arg\min_{w} L(w)$$



Loss function 1

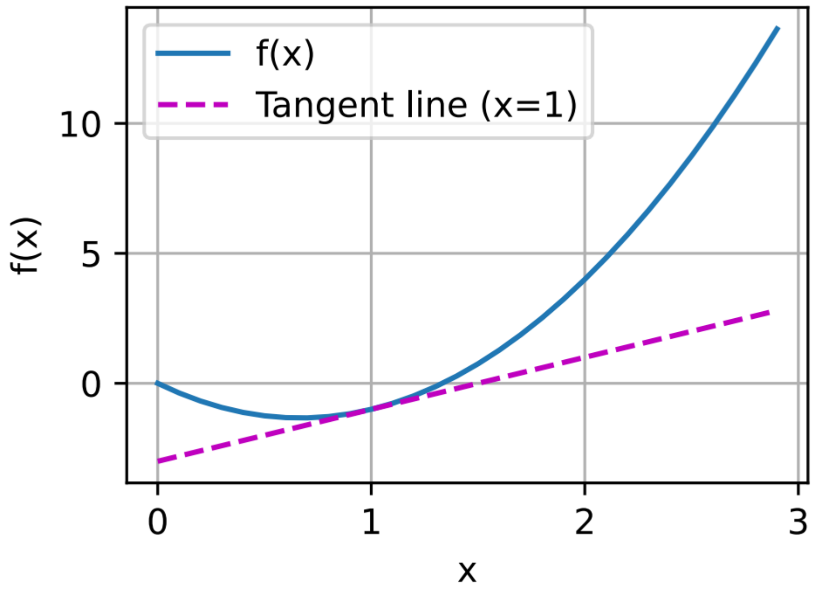


Loss function 2

Follow the slope

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

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



Follow the slope

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

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

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

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

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

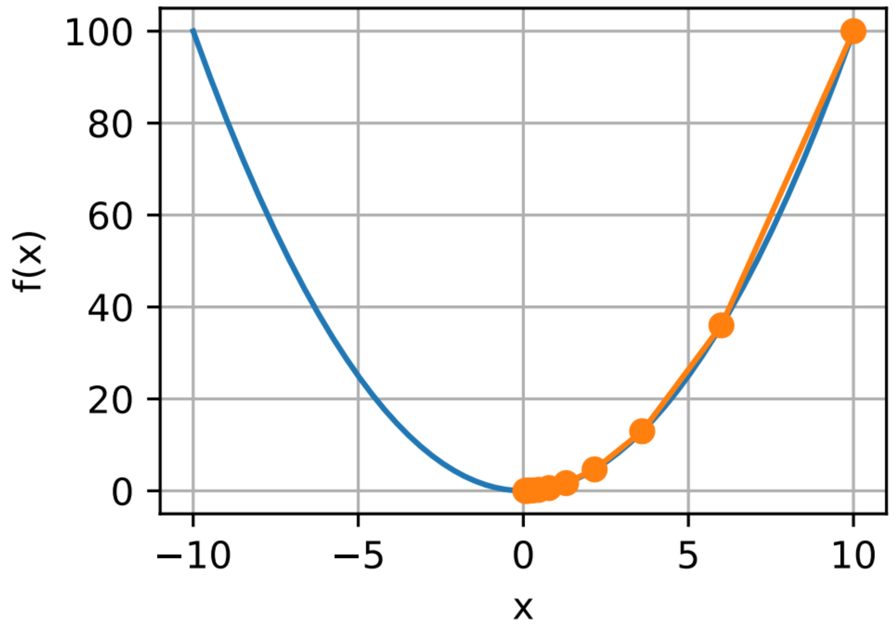
One-Dimensional Gradient Descend

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)
```

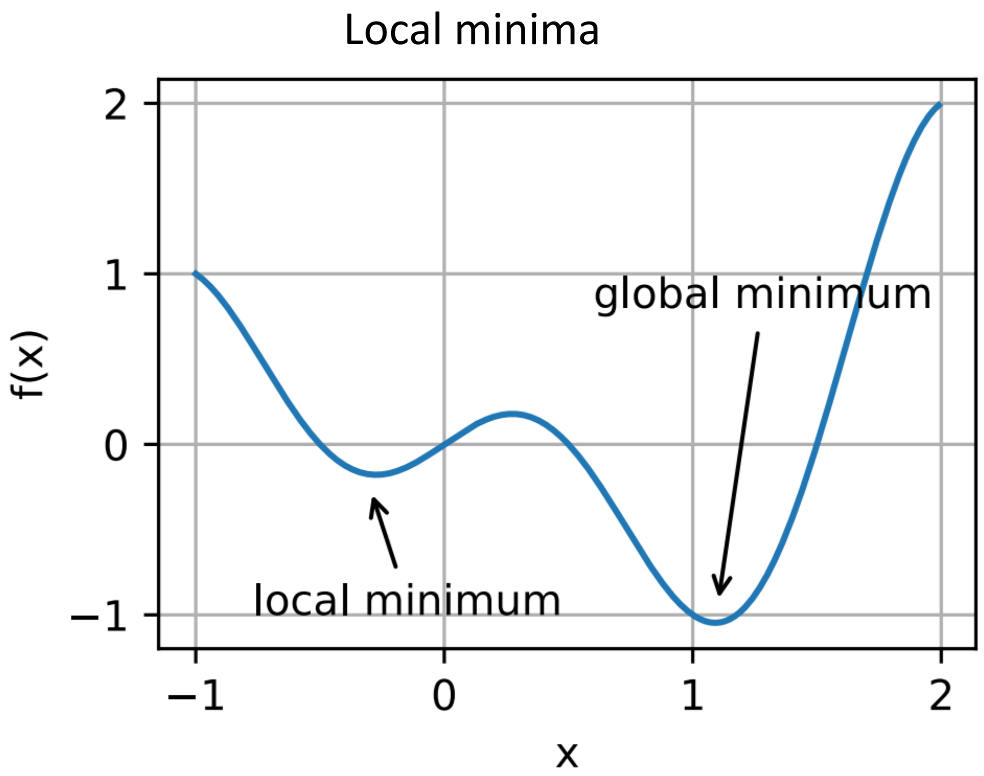
```
def f(x):  # Objective function
    return x ** 2

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

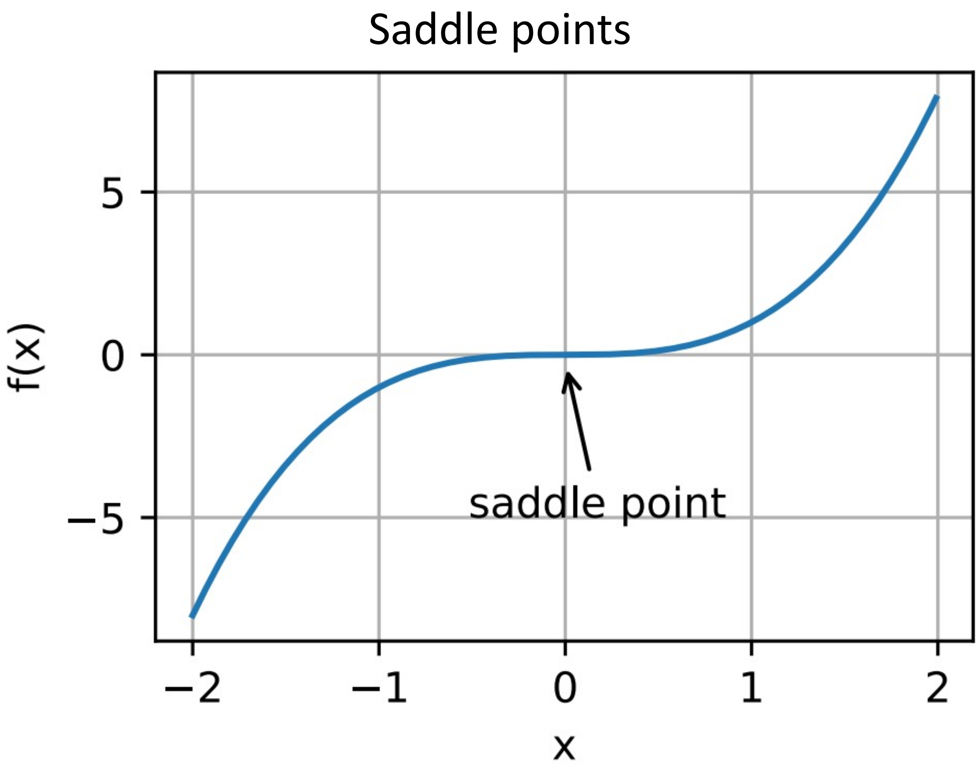
```
def show_trace(results, f):
    n = max(abs(min(results)), abs(max(results)))
    f_line = torch.arange(-n, n, 0.01)
    d2l.set_figsize()
    d2l.plot([f_line, results], [[f(x) for x in f_line], [
        f(x) for x in results]], 'x', 'f(x)', fmts=['-', '-o'])

show_trace(results, f)
```



Challenges with Gradient Descend

Gradient becomes zeros or vanishes





Zero gradient, gradient descent gets stuck

Figures from d2l.ai

Stochastic Gradient Descent (SGD)

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

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

```python
# 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
```

Full sum expensive when N is large!

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

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

Stochastic Gradient Descent (SGD)

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

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

Approximate expectation via sampling

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

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

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

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

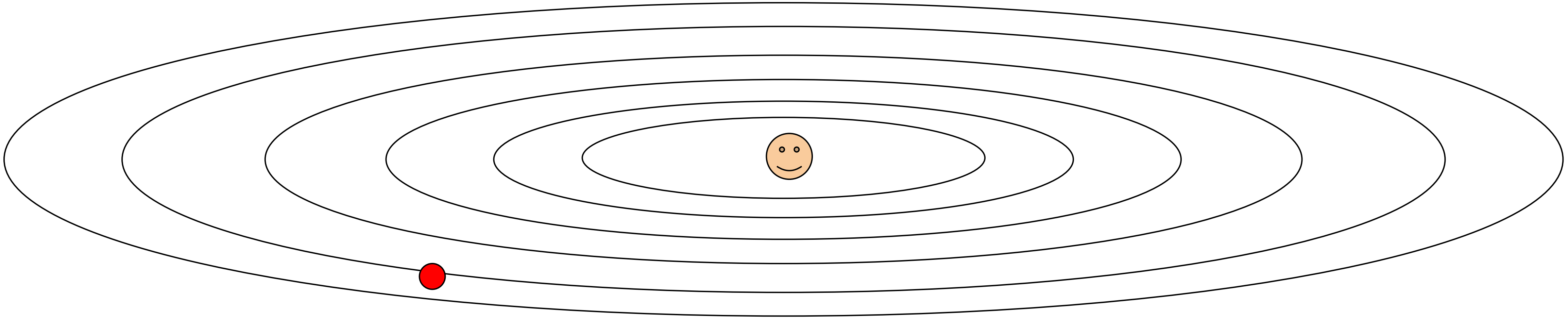
- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

AdaGrad

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

```
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)
```



Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR 2011

RMSProp: "Leaky Adagrad"

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

```
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)
```

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

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

Tieleman and Hinton, 2012

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: beta1 = 0.9, beta2 = 0.999

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

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

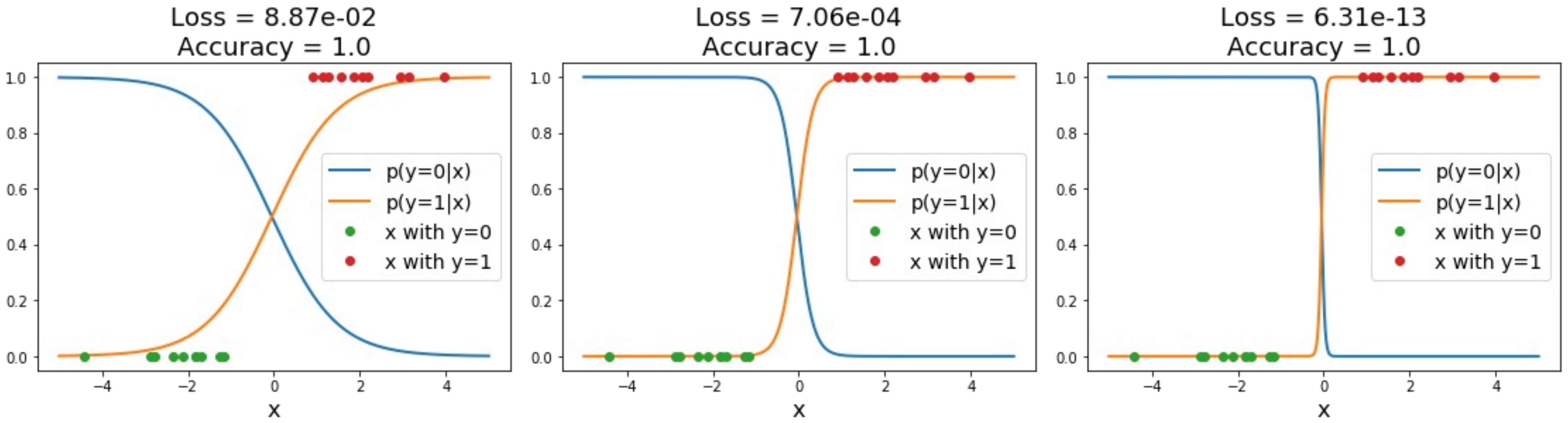
Example: Linear classifier with 1D inputs, 2 classes, softmax loss

$$s_i = w_i x + b_i$$

$$p_i = \frac{\exp(s_i)}{\exp(s_1) + \exp(s_2)}$$

$$L = -\log(p_y)$$

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!

Low loss, but unnatural "cliff" between training points

Regularization: Beyond Training Error

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

λ is a hyperparameter giving regularization strength

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

Regularization: Expressing Preferences

L2 Regularization

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

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

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

$$\boxed{w_2 = [0.25, 0.25, 0.25, 0.25]}$$

L2 regularization prefers weights to be "spread out"

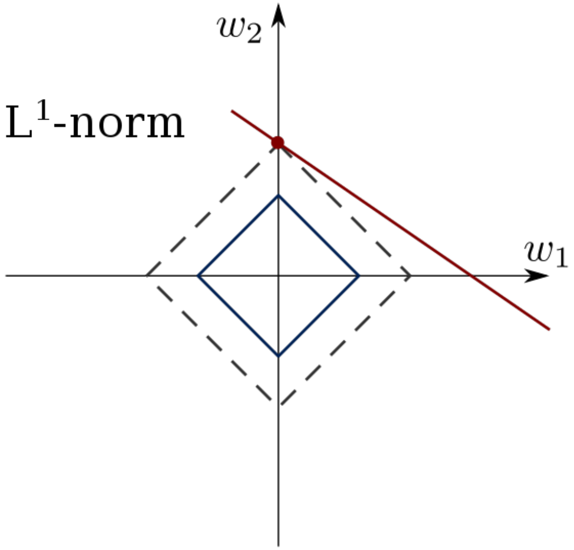
$$w_1^T x = w_2^T x = 1$$

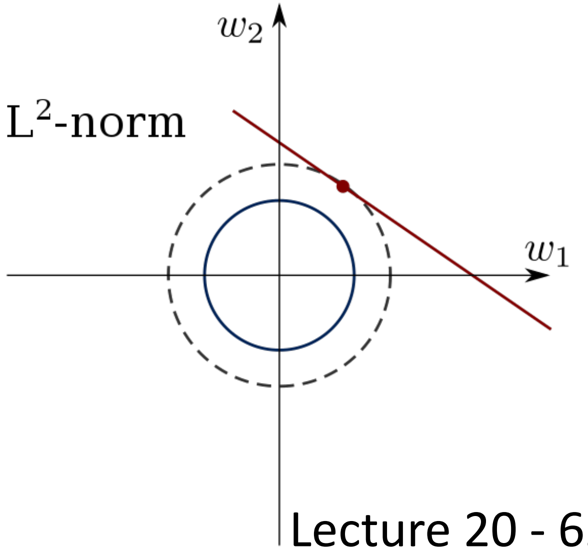
Same predictions, so data loss will always be the same

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

https://en.wikipedia.org/wiki/Lasso_(statistics)

L2 Regularization vs Weight Decay

Optimization Algorithm

$$L(w) = L_{data}(w) + L_{reg}(w)$$
$$g_t = \nabla L(w_t)$$
$$s_t = optimizer(g_t)$$
$$w_{t+1} = w_t - \alpha s_t$$

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

L2 Regularization

$$L(w) = L_{data}(w) + \boldsymbol{\lambda}|\boldsymbol{w}|^2$$
$$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + \boldsymbol{2\lambda w_t}$$
$$s_t = optimizer(g_t)$$
$$w_{t+1} = w_t - \alpha s_t$$

Weight Decay

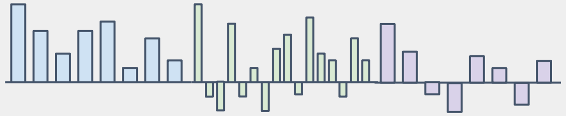
$$L(w) = L_{data}(w)$$
$$g_t = \nabla L_{data}(w_t)$$
$$s_t = optimizer(g_t) + \boldsymbol{2\lambda w_t}$$
$$w_{t+1} = w_t - \alpha s_t$$

Loshchilov and Hutter, “Decoupled Weight Decay Regularization”, ICLR 2019

Lecture 5: Neural Networks

Image Features vs Neural Networks

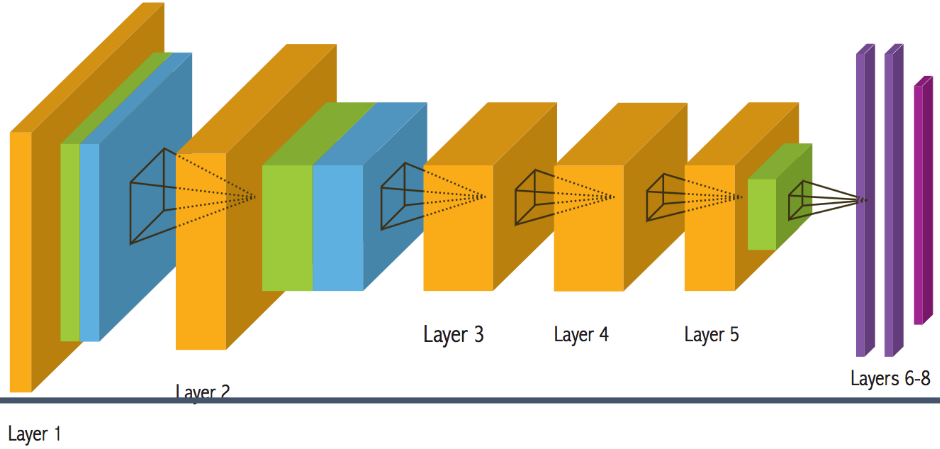
Feature Extraction



f

20 numbers giving scores for classes

training



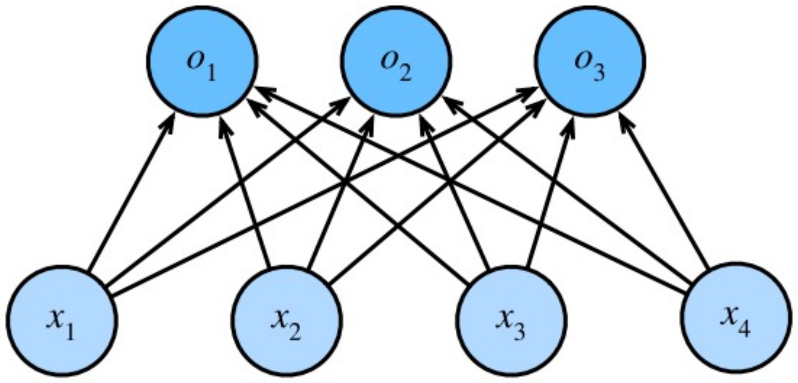
Krizhevsky, Sutskever, and Hinton, "Imagenet classification with deep convolutional neural networks", NIPS 2012.

20 numbers giving scores for classes

training

Neural Networks

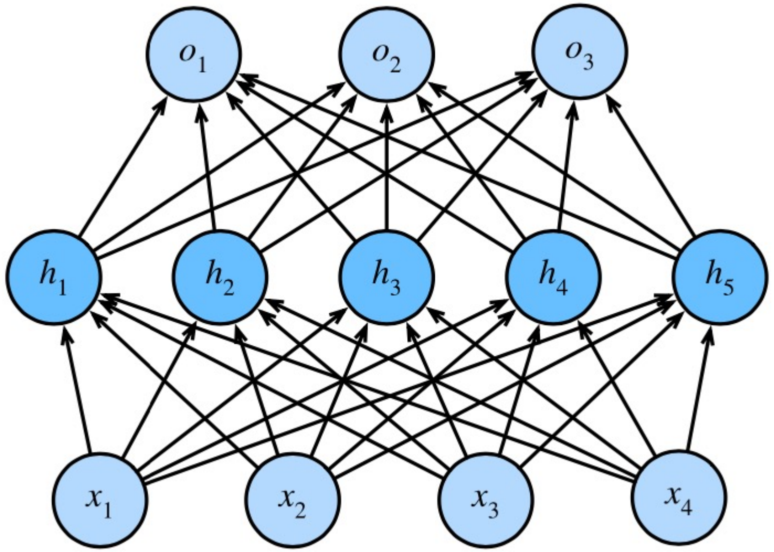
Input: $x \in \mathbb{R}^D$ **Output**: $f(x) \in \mathbb{R}$^C



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**: $f(x) \in \mathbb{R}\hat{}C$

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: $f(x) = W_2 \max(0, W_1 x + b_1) + b2$

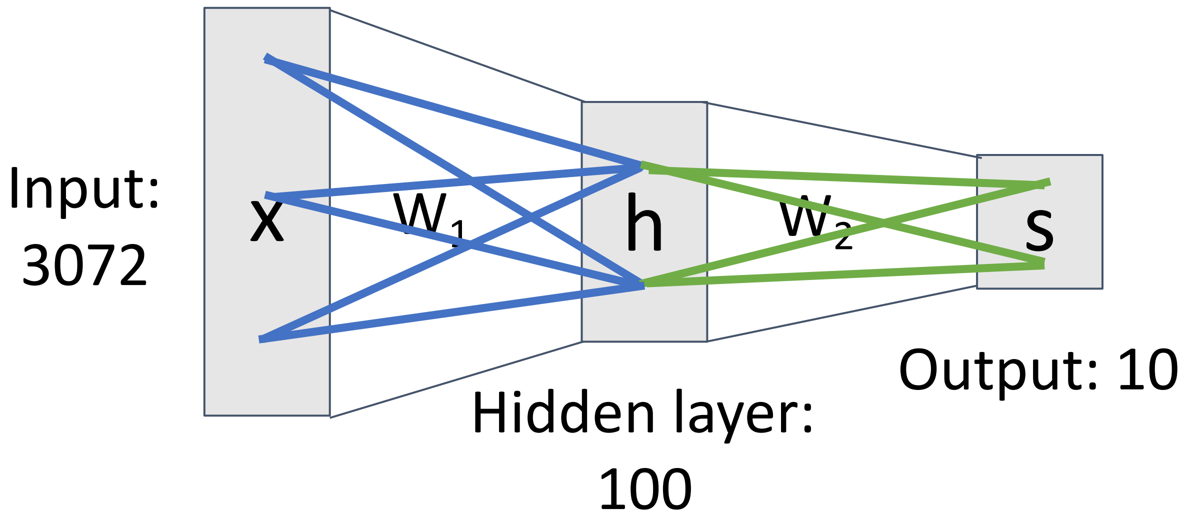
Neural Networks

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

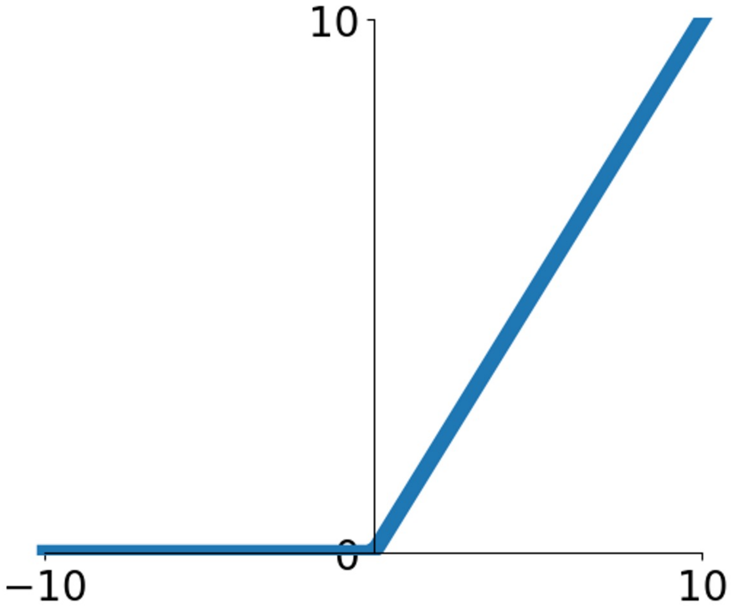
All elements of h affect all elements of s

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

Activation Functions

2-layer Neural Network $f(x) = W_2 \boxed{\max(0,} W_1 x + b_1) + b_2$

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

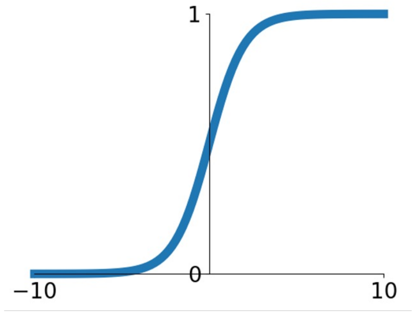


This is called the **activation function** of the neural network

Activation Functions

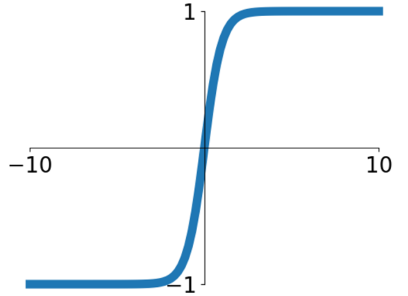
Sigmoid

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



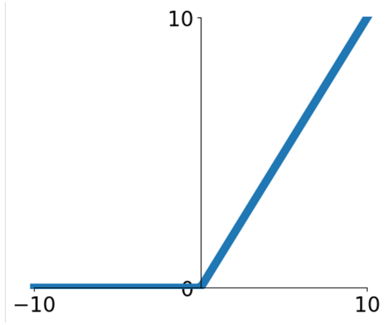
tanh

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



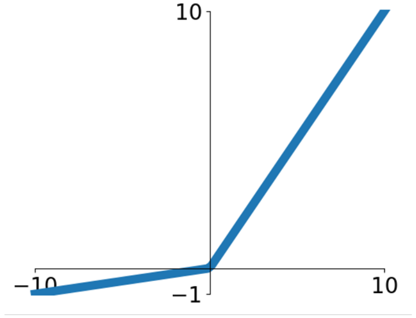
ReLU

$$\max(0, x)$$



Leaky ReLU

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

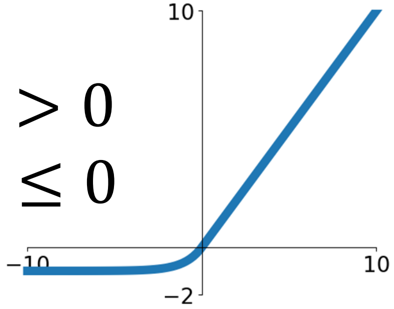


Softplus

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

ELU

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

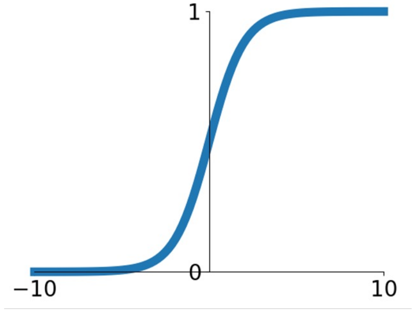


Activation Functions

ReLU is a good default choice for most problems

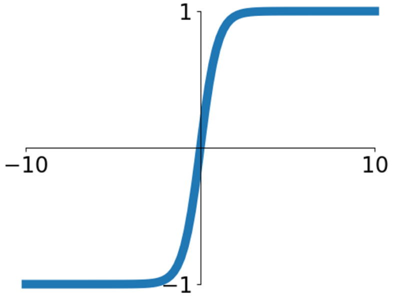
Sigmoid

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



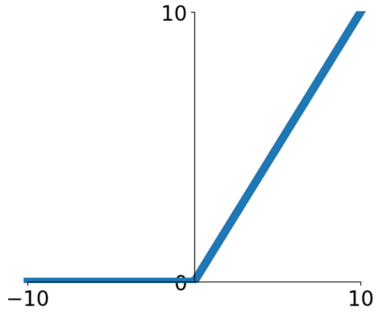
tanh

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



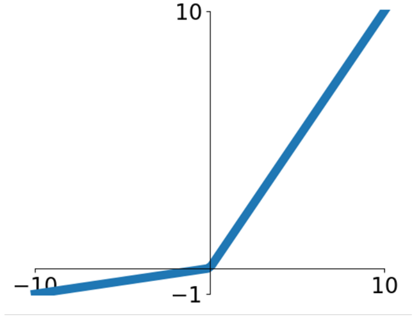
ReLU

$$\max(0, x)$$



Leaky ReLU

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

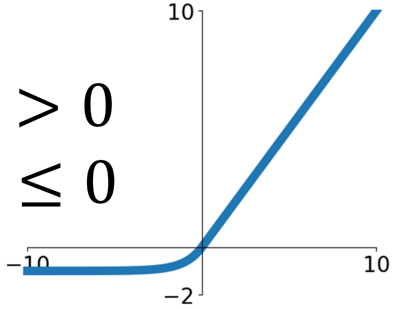


Softplus

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

ELU

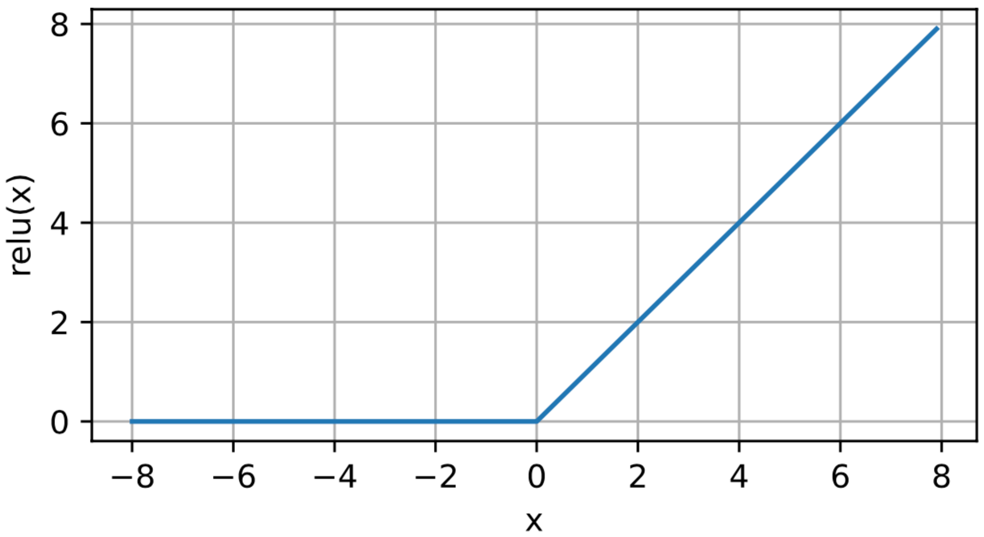
$$f(x) = \begin{cases} x, & x > 0 \\ \alpha(\exp(x) - 1), & x \le 0 \end{cases}$$



Why ReLU

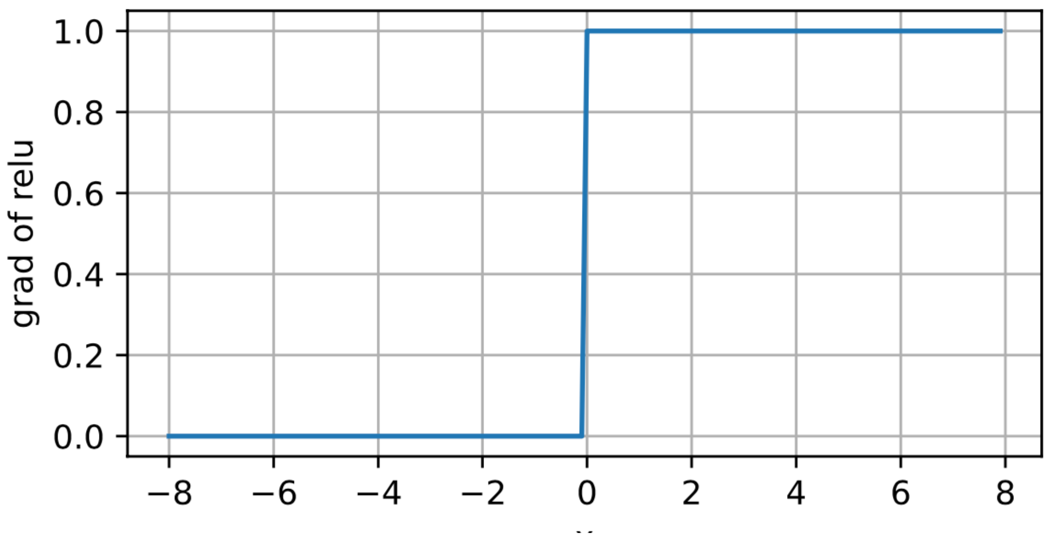
Forward pass

```
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
```



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

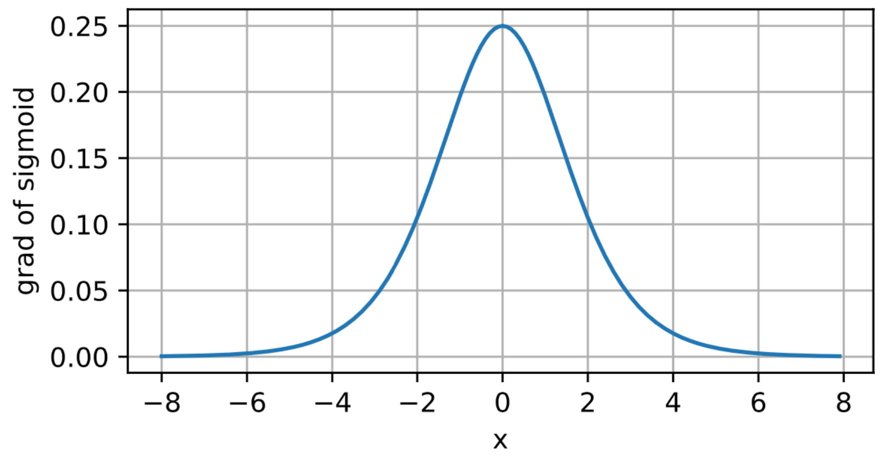
sigmoid is also called squashing function, turn (-inf, inf) into (0,1)

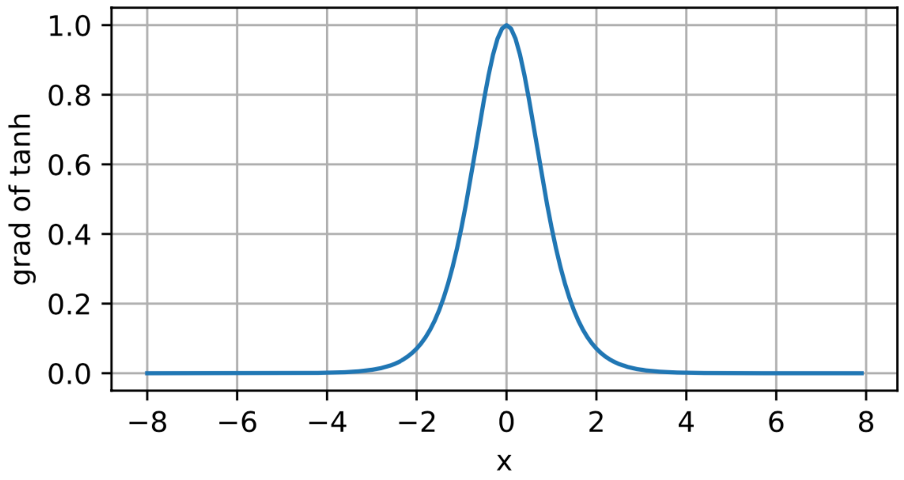
$$\frac{d}{dx}\operatorname{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \operatorname{sigmoid}(x)\left(1-\operatorname{sigmoid}(x)\right).$$

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

```python
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))
```

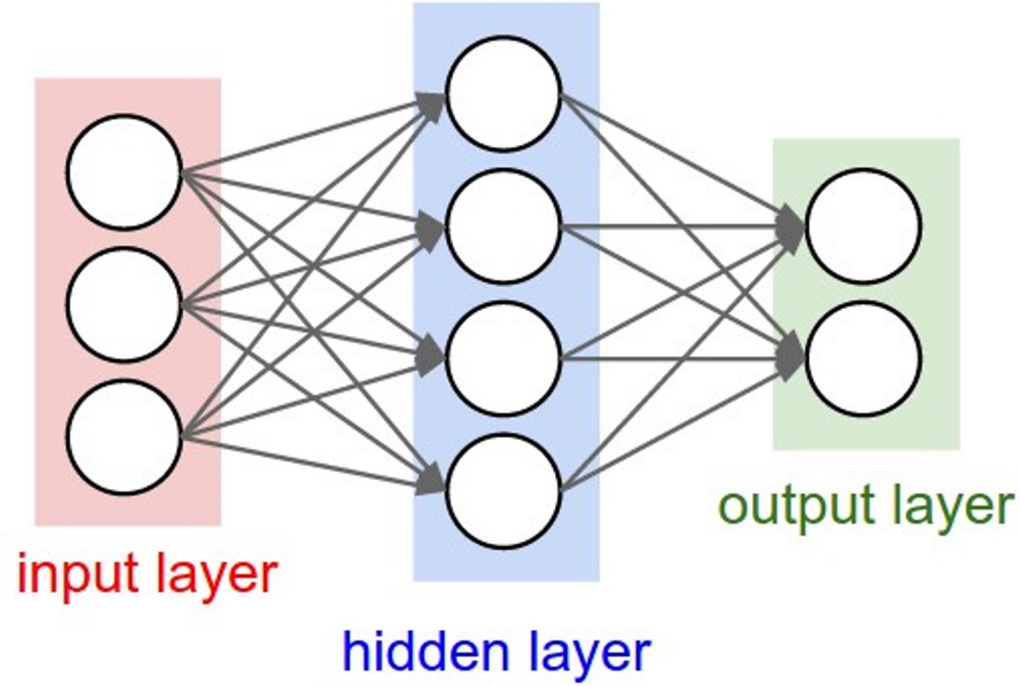
```python
y = torch.tanh(x)
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```





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

Neural Net in <20 lines!




```python
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]
```

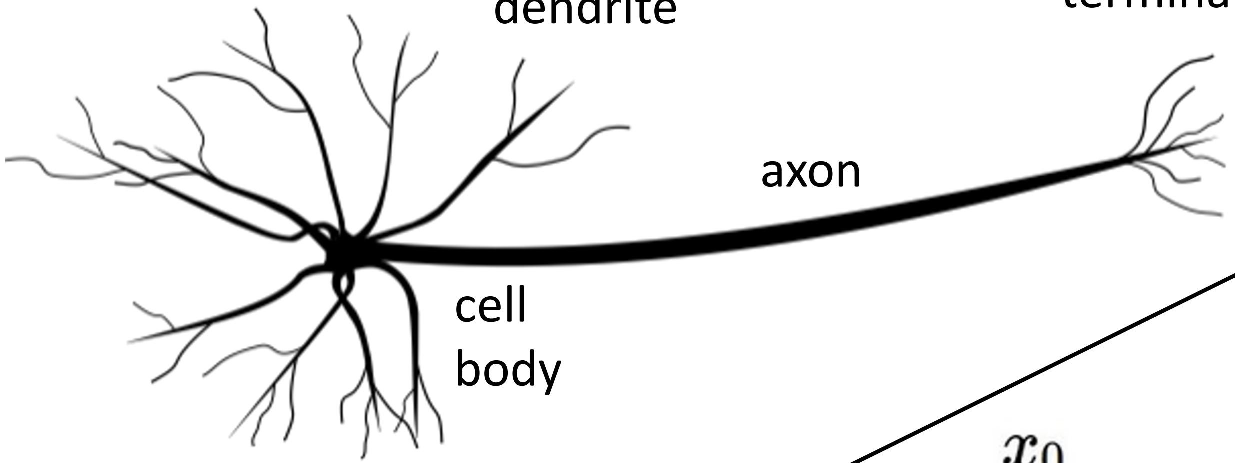
```python
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

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

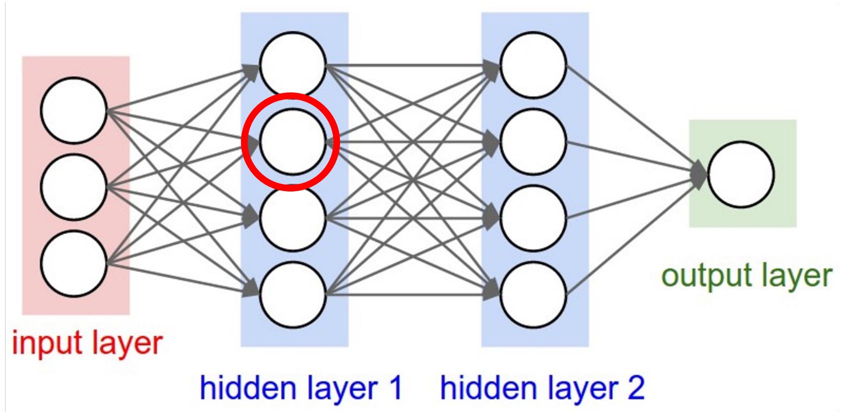
```python
loss = nn.CrossEntropyLoss()
```

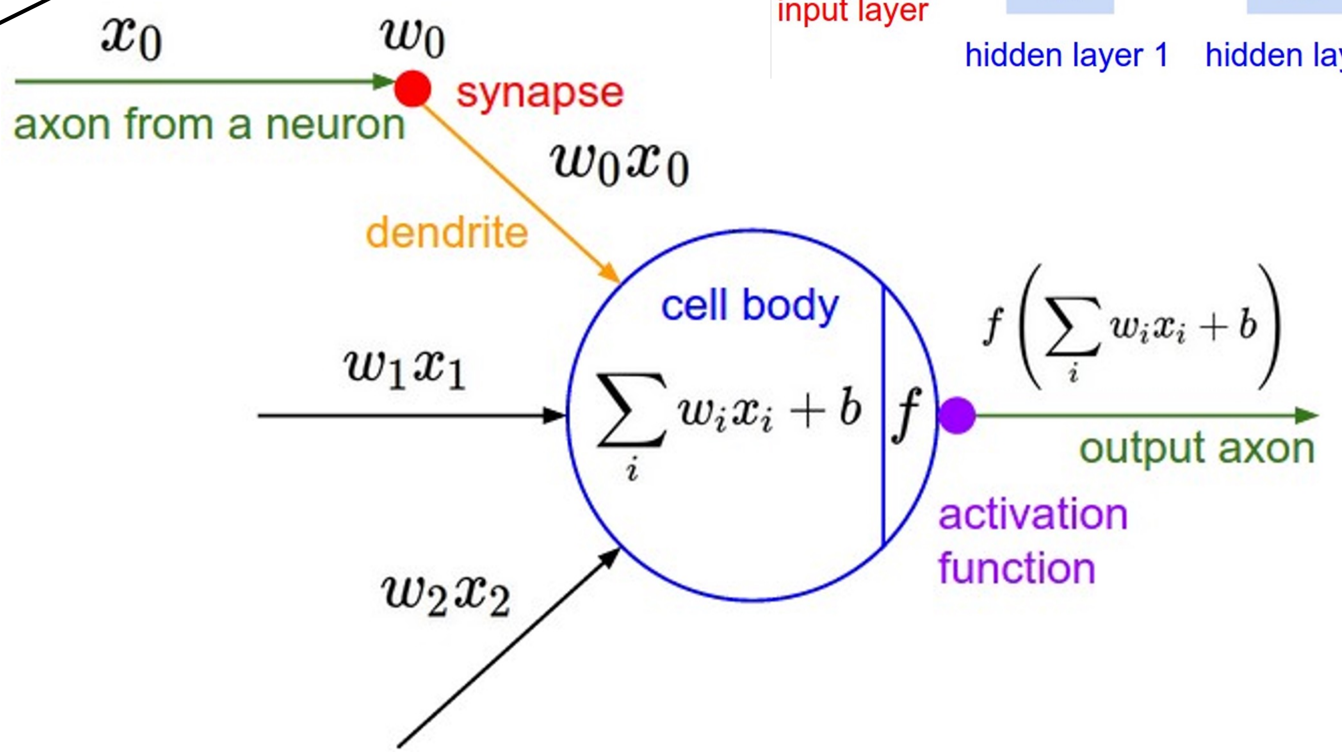
```python
num_epochs, lr = 10, 0.1
updater = torch.optim.SGD(params, lr=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, updater)
```

Biological Neuron



Artificial Neuron



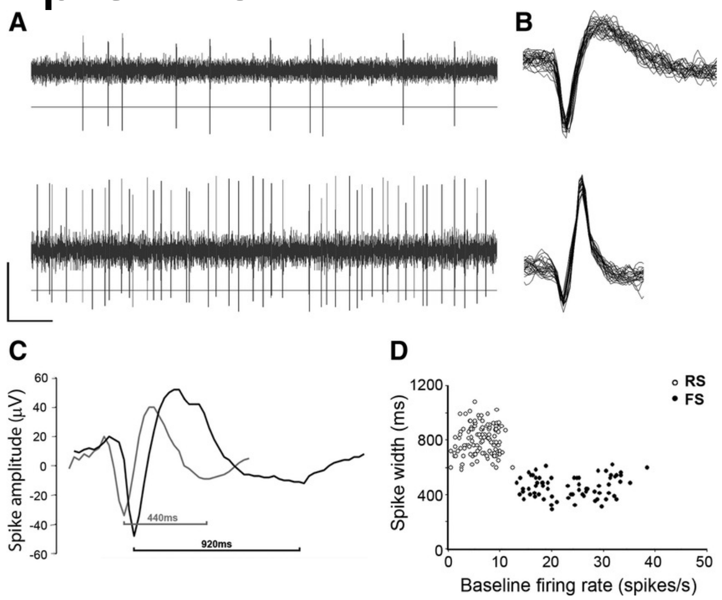


Are neural networks biologically inspired? Weakly

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

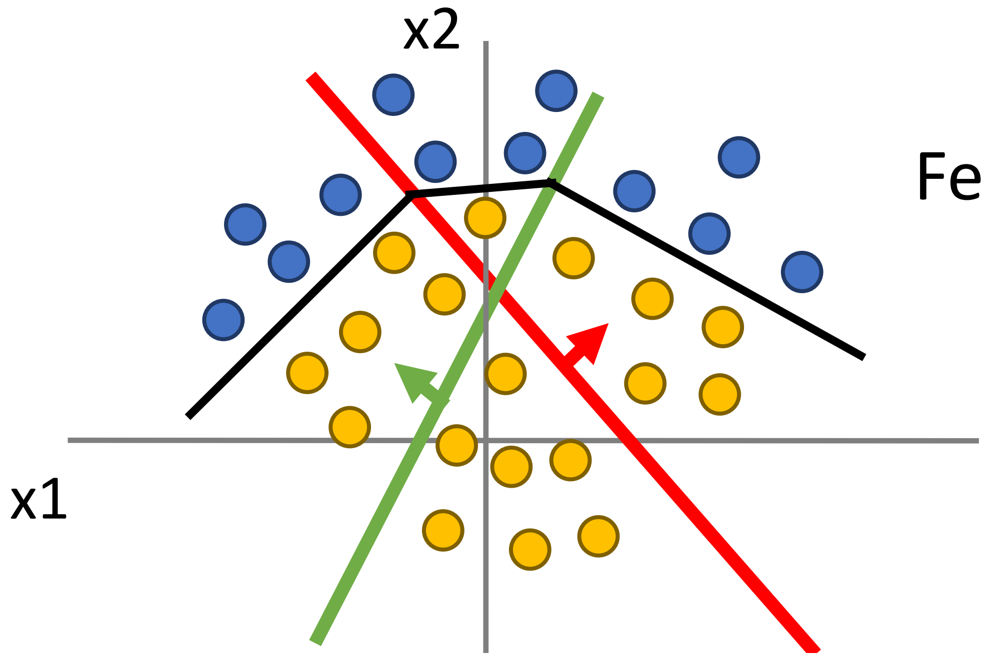
[Dendritic Computation. London and Hausser]



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

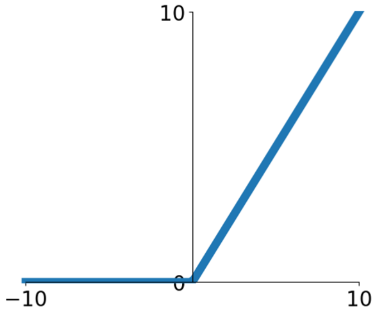
Why ReLU? Space Warping

Points not linearly separable in original space

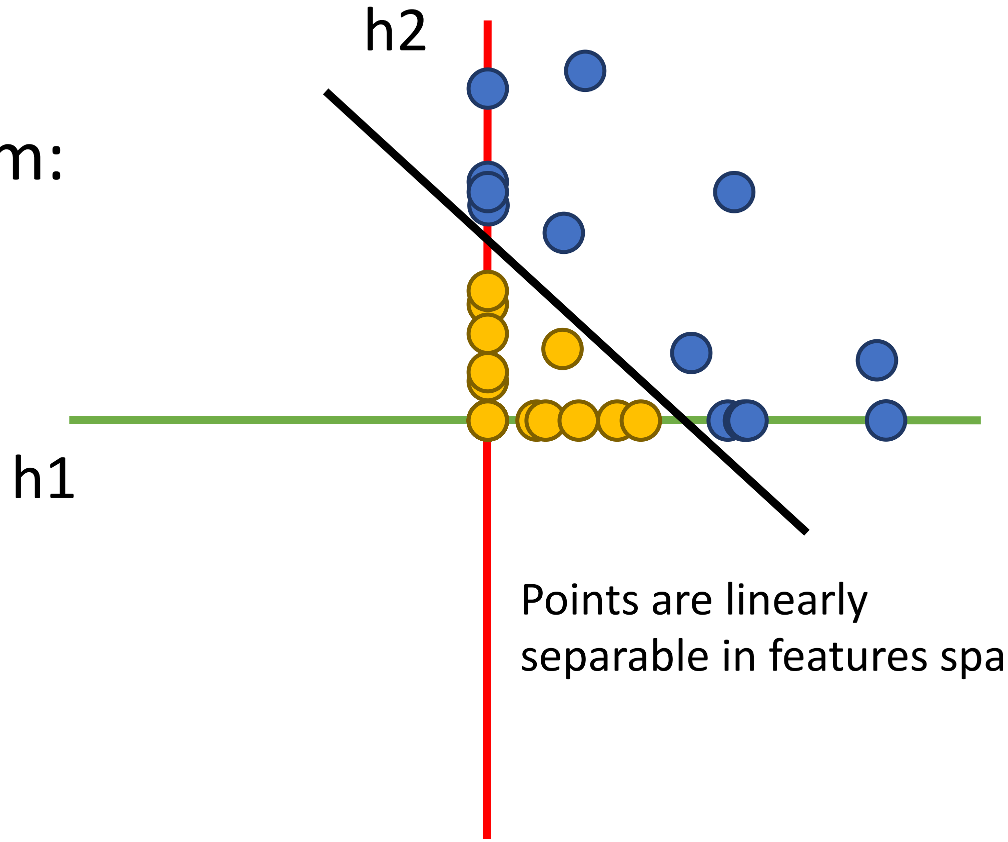


Linear classifier in feature space gives nonlinear classifier in original space

Feature transform:
h = ReLU(Wx)



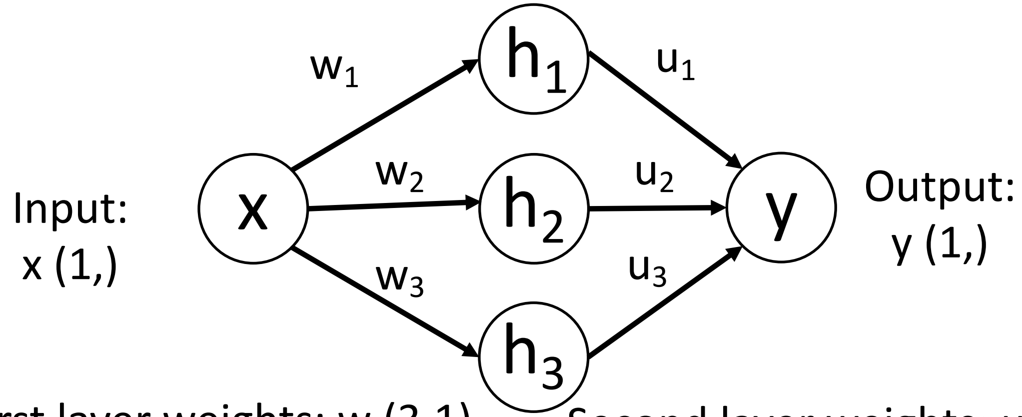
Consider a neural net hidden layer:
h = ReLU(Wx) = max(0, Wx)
Where x, h are both 2-dimensional



Points are linearly separable in features space!

Universal Approximation

Example: Approximating a function f: R -> R with a two-layer ReLU network



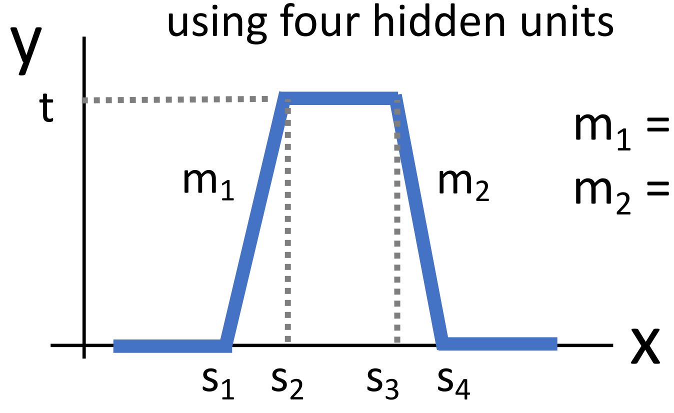
First layer weights: w (3,1)
First layer bias: b (3,)

Second layer weights: u (1,3)
Second layer bias: p (1,)

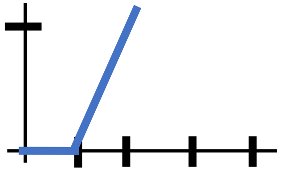
$$h_1 = \max(0, w_1 * x + b_1)$$
$$h_2 = \max(0, w_2 * x + b_2)$$
$$h_3 = \max(0, w_3 \ * x + b_3)$$
$$y = u_1 h_1 + u_2 * h_2 + u_3 * h_3 + p$$

$$y = u_1 * \max(0, w_1 * x + b_1)$$
$$+ u_2 * \max(0, w_2 * x + b_2)$$
$$+ u_3 * \max(0, w_3 * x + b_3) + p$$

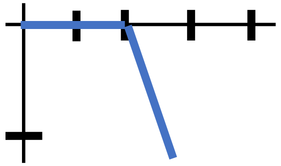
We can build a "bump function" using four hidden units



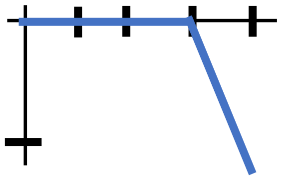
$m_1 = t / (s_2 - s_1)$
$m_2 = t / (s_4 - s_3)$



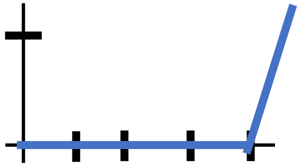
m_1 * max(0, $x - s_1$)



-m_1 * max(0, $x - s_2$)



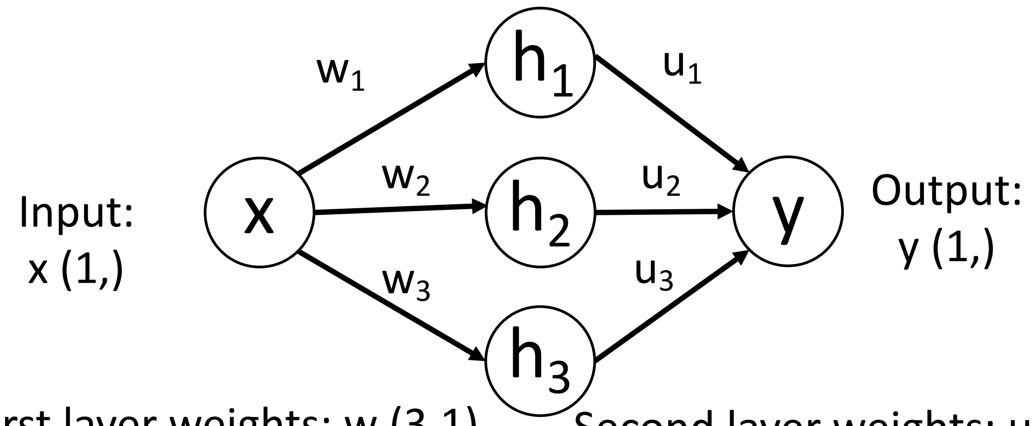
-m_2 * max(0, $x - s_3$)



m_2 * max(0, $x - s_4$)

Universal Approximation

Example: Approximating a function f: R -> R with a two-layer ReLU network



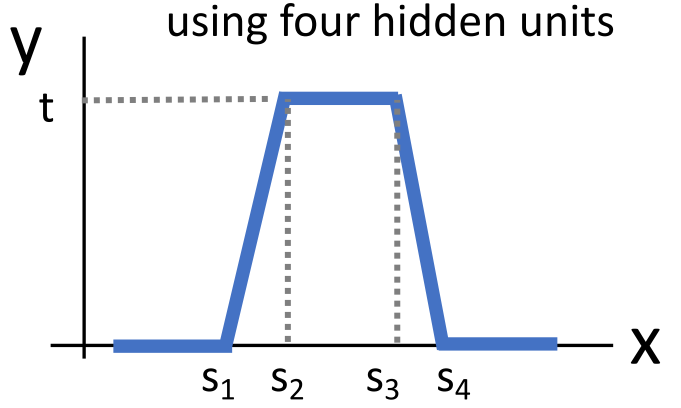
First layer weights: w (3,1)
First layer bias: b (3,)

Second layer weights: u (1,3)
Second layer bias: p (1,)

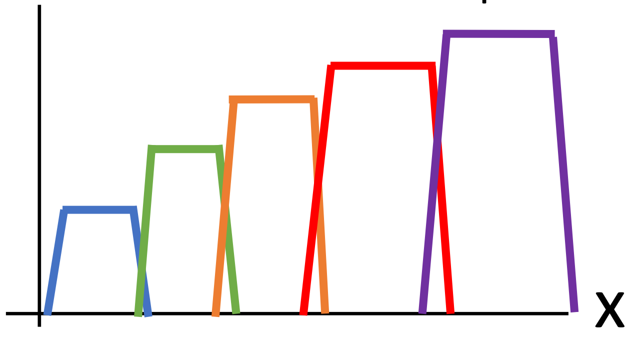
$$h_1 = \max(0, w_1 * x + b_1)$$
$$h_2 = \max(0, w_2 * x + b_2)$$
$$h_3 = \max(0, w_3 * x + b_3)$$
$$y = u_1 h_1 + u_2 * h_2 + u_3 * h_3 + p$$

$$\begin{aligned} y = u_1 * \max(0, w_1 * x + b_1) \\ + u_2 * \max(0, w_2 * x + b_2) \\ + u_3 * \max(0, w_3 * x + b_3) + p \end{aligned}$$

We can build a "bump function" using four hidden units

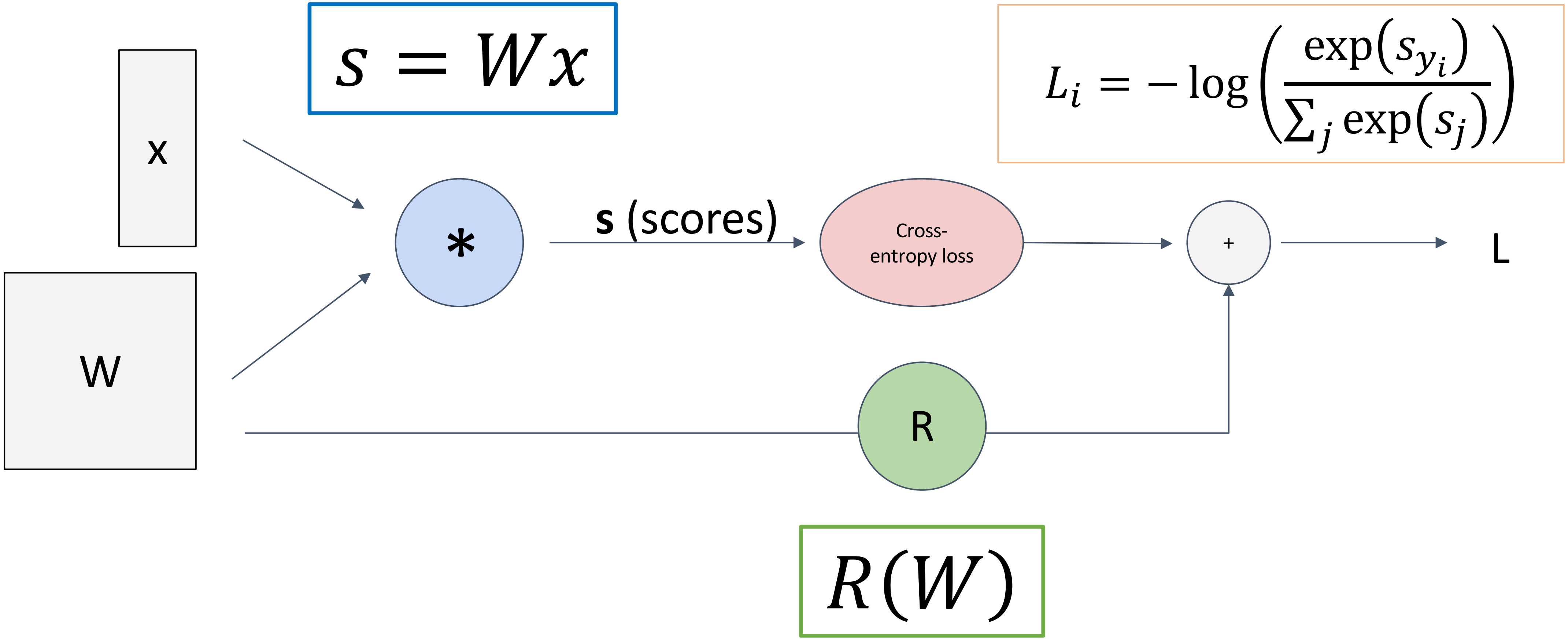


With 4K hidden units we can build a sum of K bumps



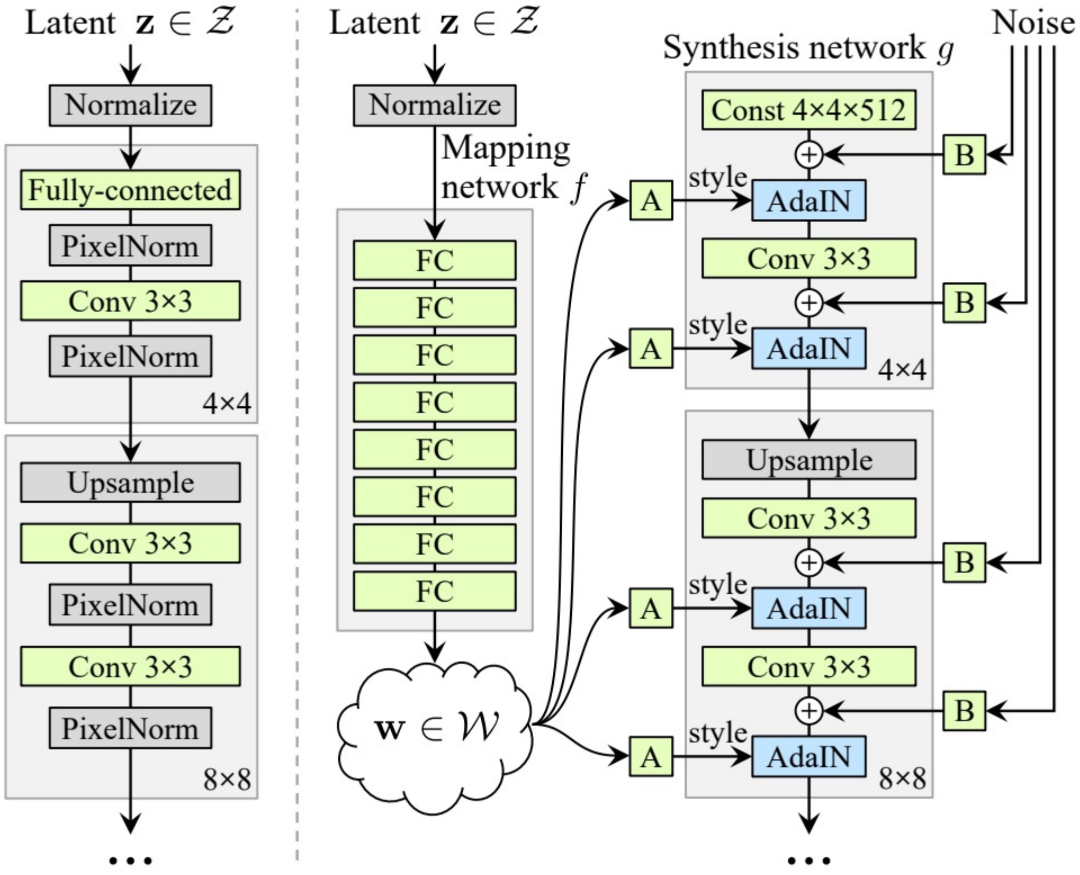
Lecture 6: Backpropagation

Computational Graphs



Simplified computational graph to denote neural networks

StyleGAN



(a) Traditional

(b) Style-based generator

Transformer

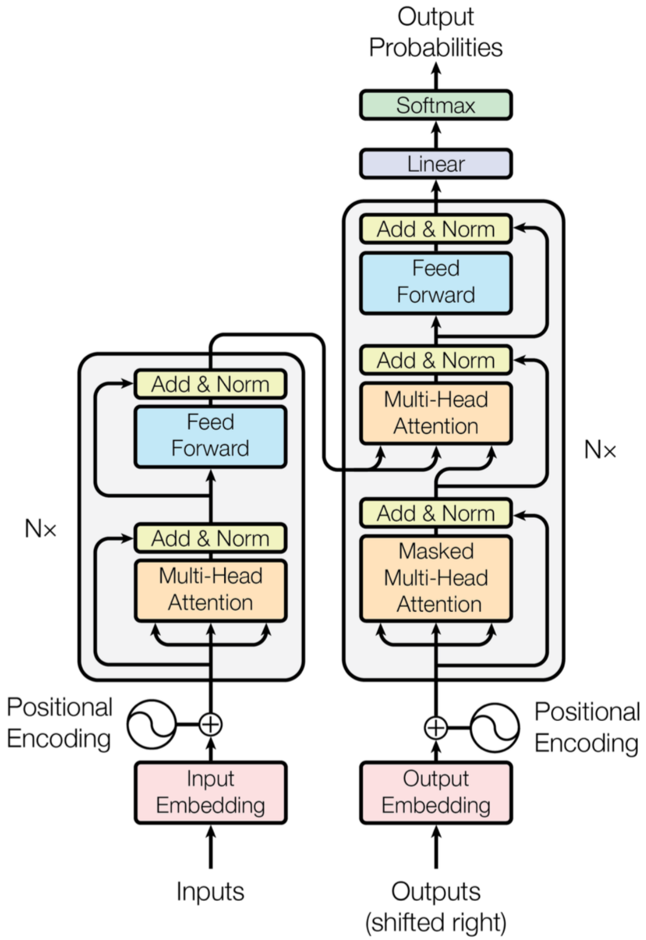


Figure 1: The Transformer - model architecture.

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 $u_1, u_2, \ldots, u_m$, where each differentiable function u_i has variables $x_1, x_2, \ldots, x_n$. Then the chain rule gives

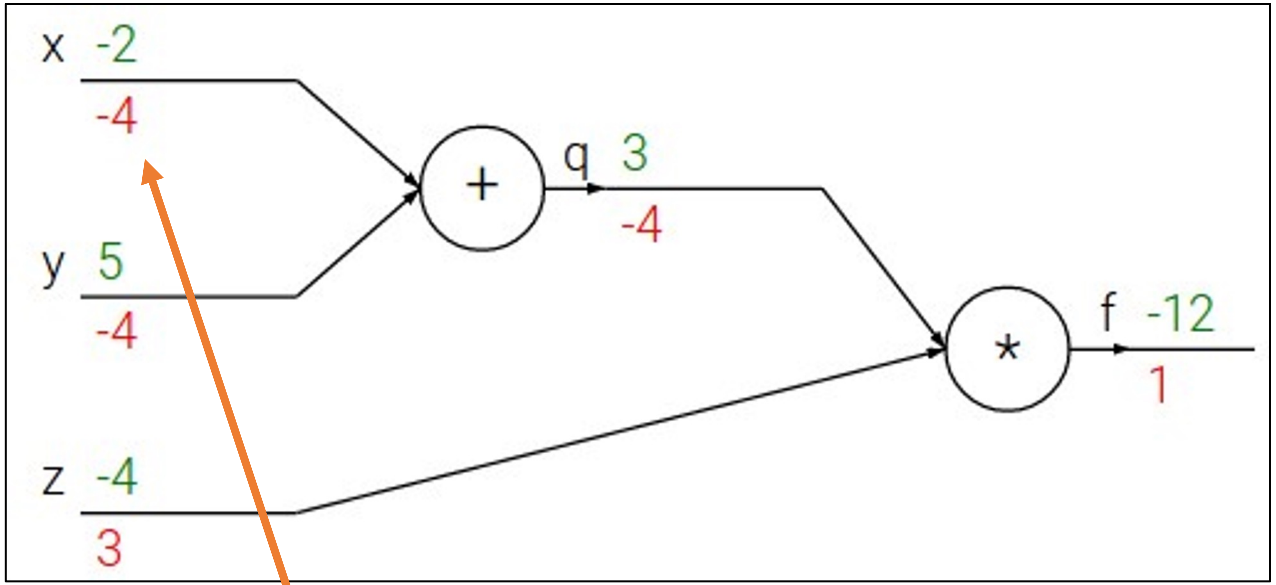
$$\frac{dy}{dx_i} = \frac{dy}{du_1}\frac{du_1}{dx_i} + \frac{dy}{du_2}\frac{du_2}{dx_i} + \ldots + \frac{dy}{du_m}\frac{du_m}{dx_i}$$

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

Backpropagation: Simple Example

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

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



1. Forward pass: Compute outputs

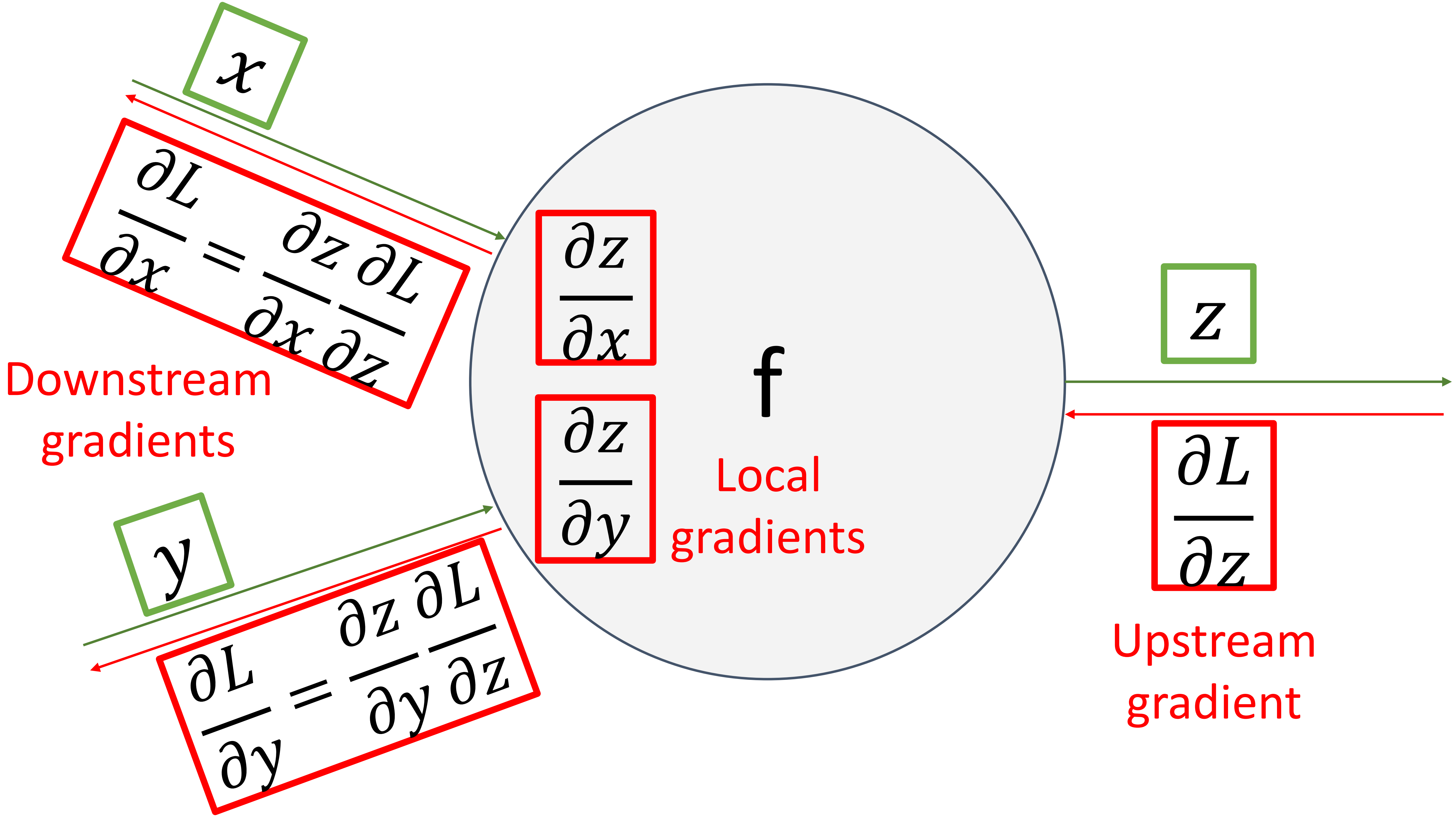
$$\boxed{q = x + y} \qquad f = q \cdot z$$

2. Backward pass: Compute derivatives

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

Chain Rule

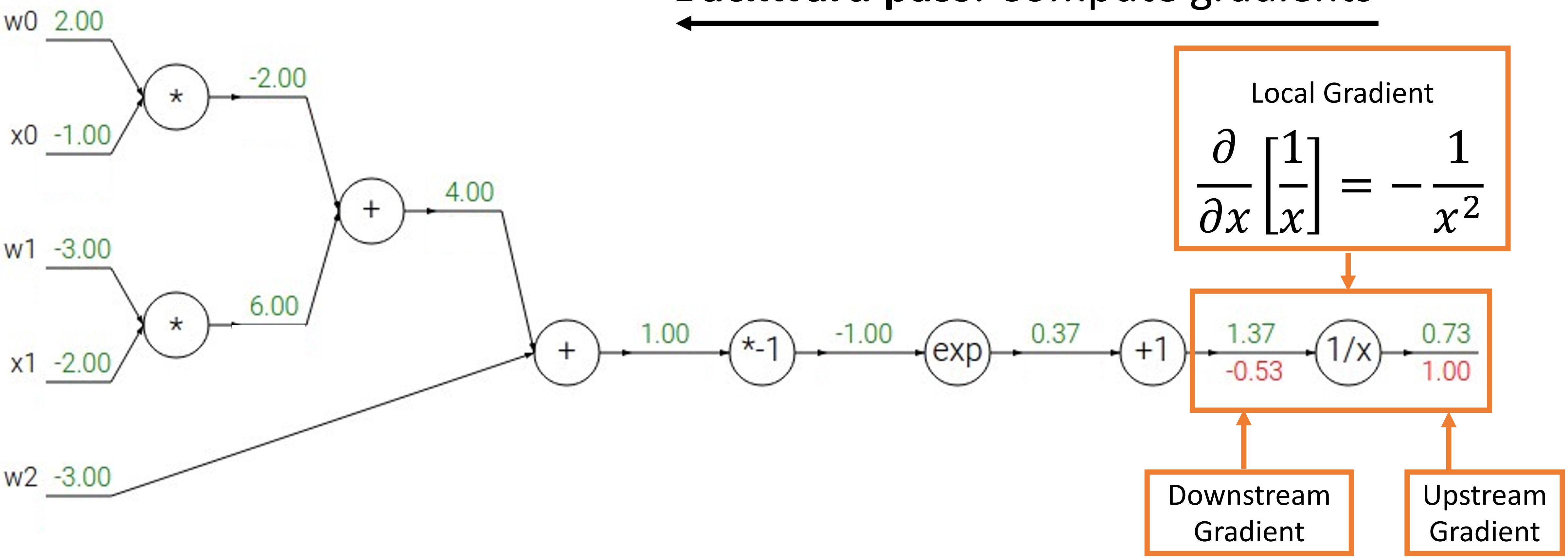
$$\boxed{\underbrace{\frac{\partial f}{\partial x}}_{\text{Downstream Gradient}} = \underbrace{\frac{\partial q}{\partial x}}_{\text{Local Gradient}} \underbrace{\frac{\partial f}{\partial q}}_{\text{Upstream Gradient}}} \qquad \boxed{\frac{\partial q}{\partial x} = 1}$$



x
$\frac{\partial L}{\partial x} = \frac{\partial z}{\partial x}\frac{\partial L}{\partial z}$
Downstream gradients
y
$\frac{\partial L}{\partial y} = \frac{\partial z}{\partial y}\frac{\partial L}{\partial z}$
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
f
Local gradients
z
$\frac{\partial L}{\partial z}$
Upstream gradient

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

Backward pass: Compute gradients



Backprop with Vectors

Loss L still a scalar!

D_x x

D_y y

z D_z

Local

Jacobian matrices

$\frac{\partial z}{\partial x}$ $[D_z \times D_x]$

f

$\frac{\partial z}{\partial y}$ $[D_z \times D_y]$

$\frac{\partial L}{\partial z}$ D_z

Upstream Gradient

For each element of z, how much does it influence L?

Downstream Gradients

Matrix-vector multiply

D_x $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$

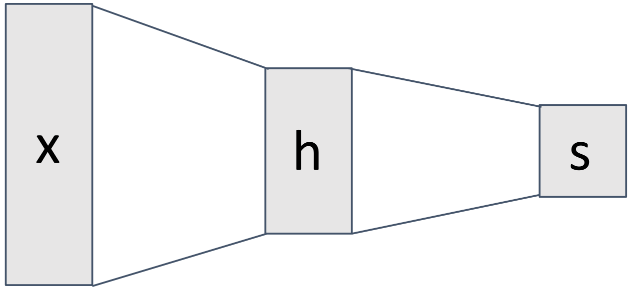
D_y $\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$

$[1 \times D_y] = [1 \times D_z] \times [D_z \times D_y]$

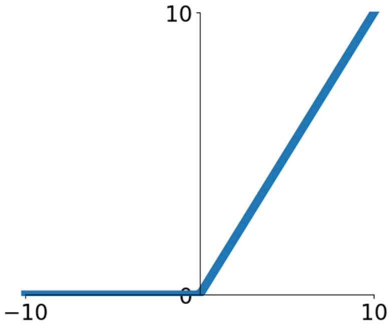
Lecture 7: Convolutional Neural Network (ConvNet)

Components of a Convolutional Network

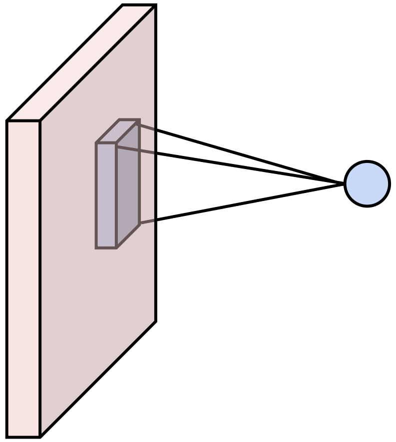
Fully-Connected Layers



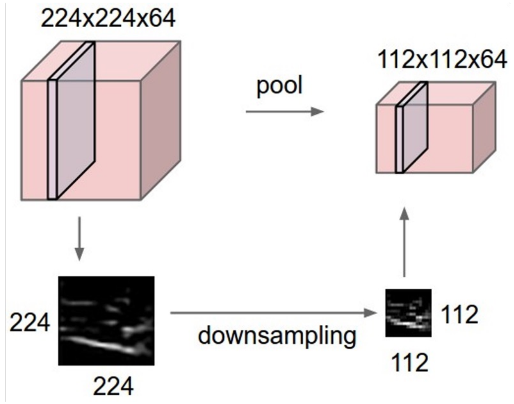
Activation Function



Convolution Layers



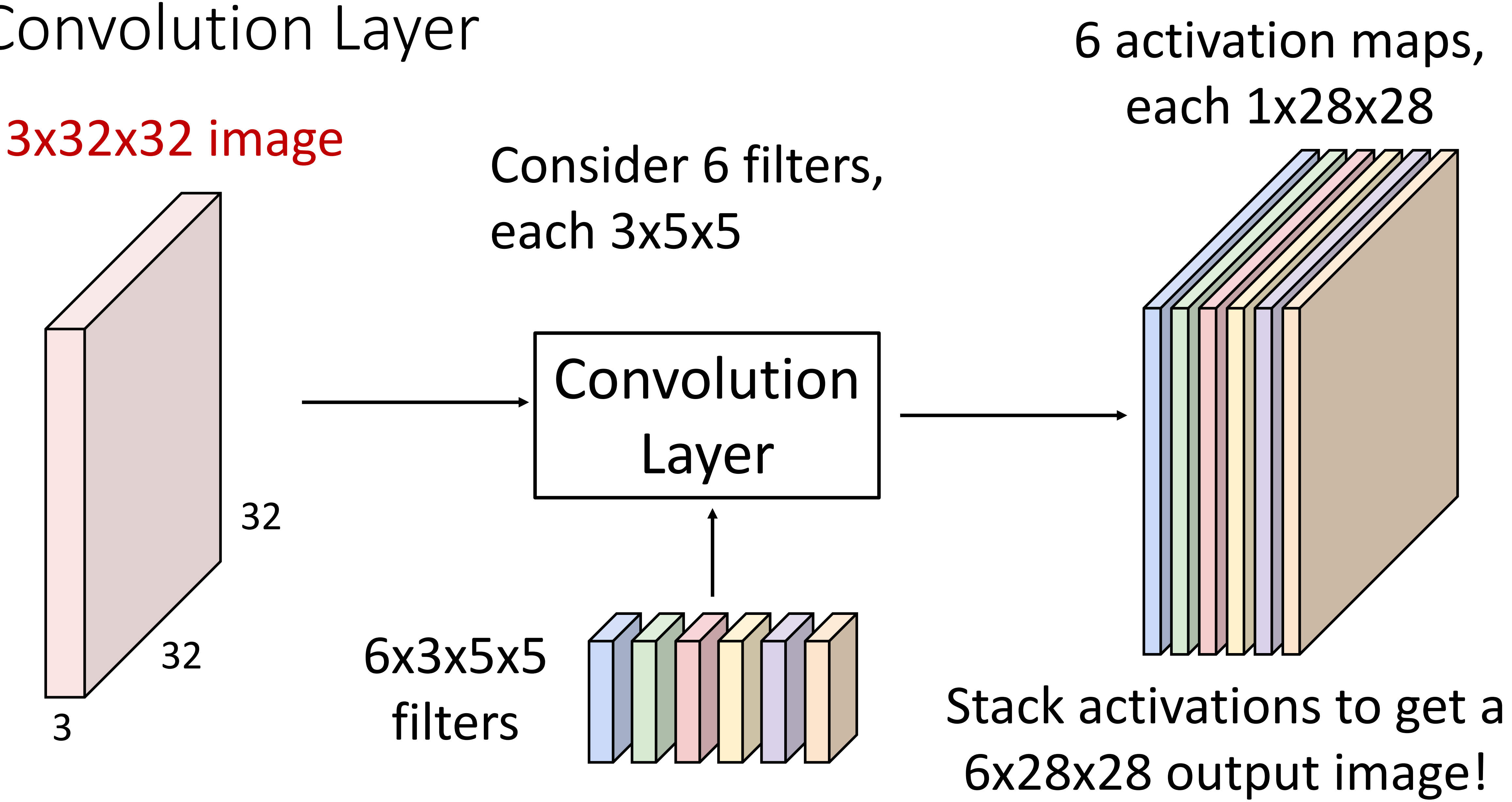
Pooling Layers



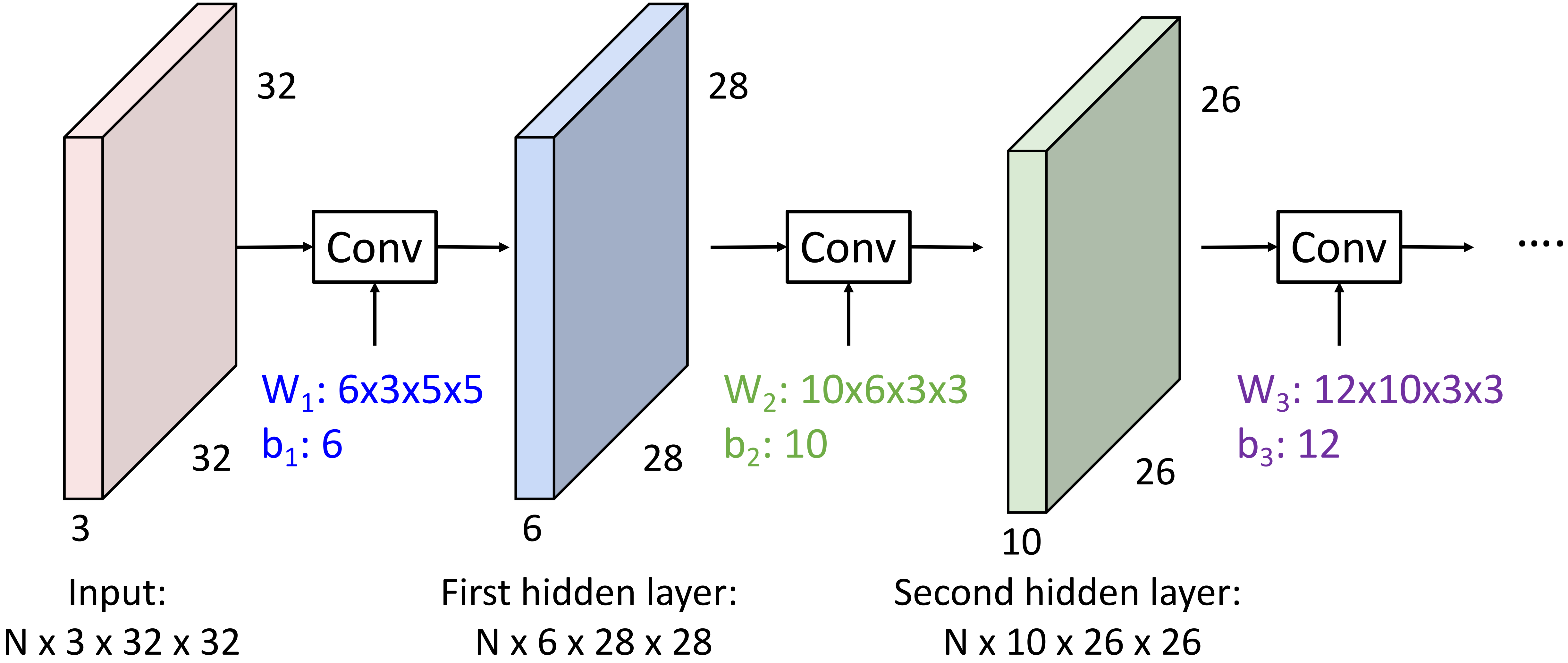
Normalization

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

Convolution Layer



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

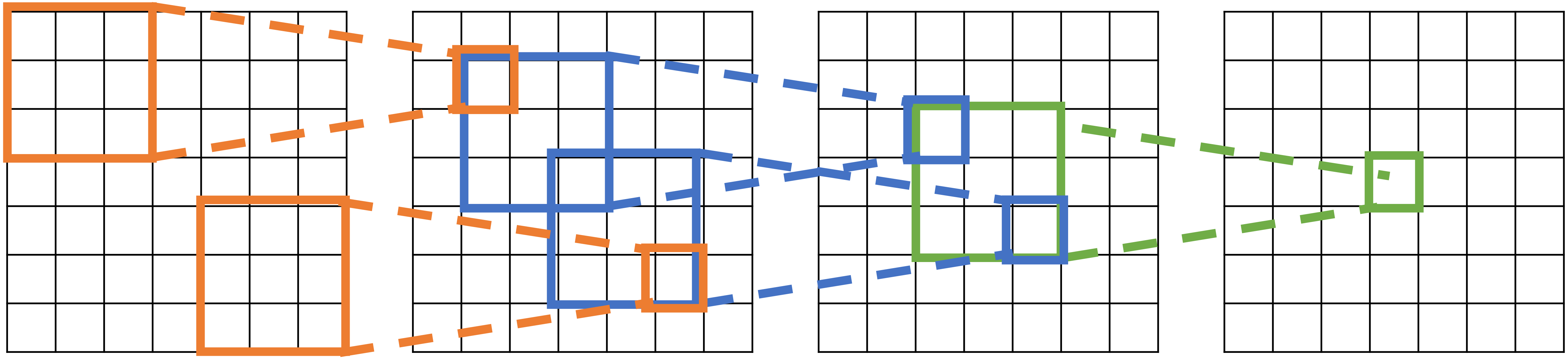
Input: 7x7
Filter: 3x3
Output: 5x5

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)



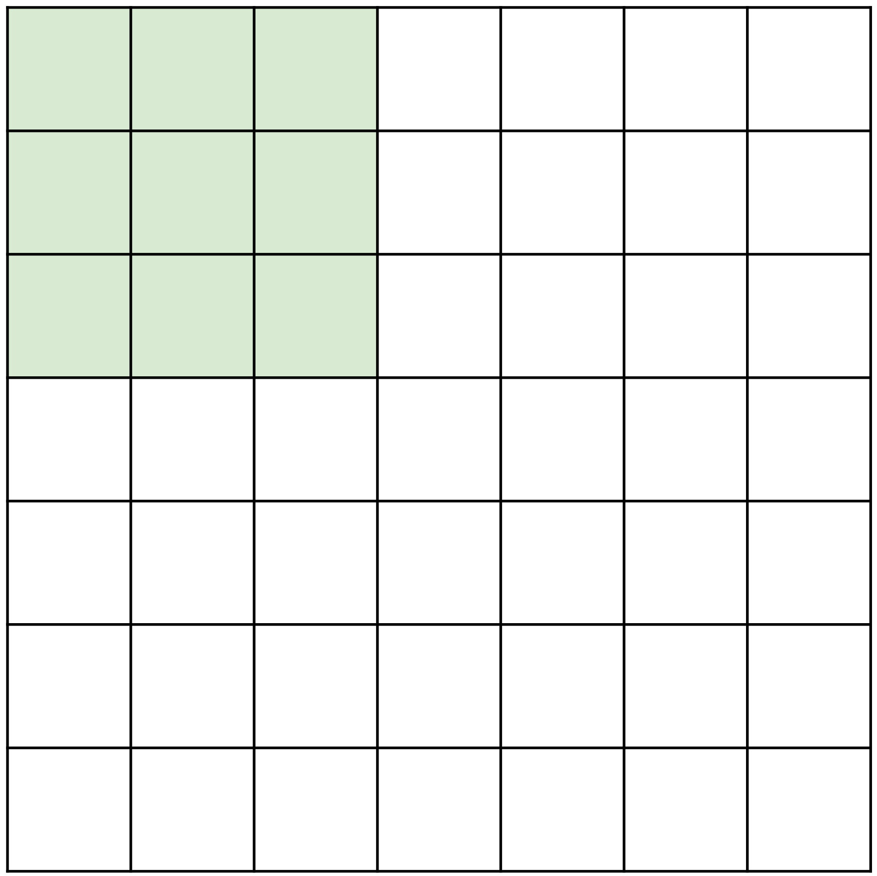
Input

Output

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

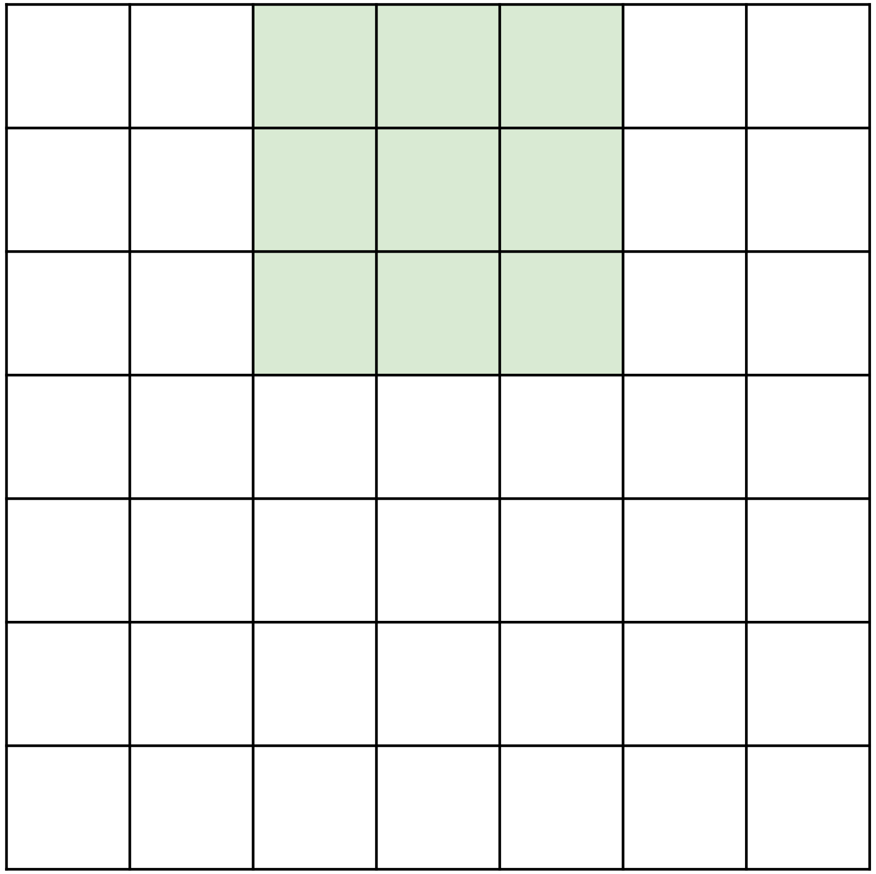
Solution: Downsample inside the network

Strided Convolution



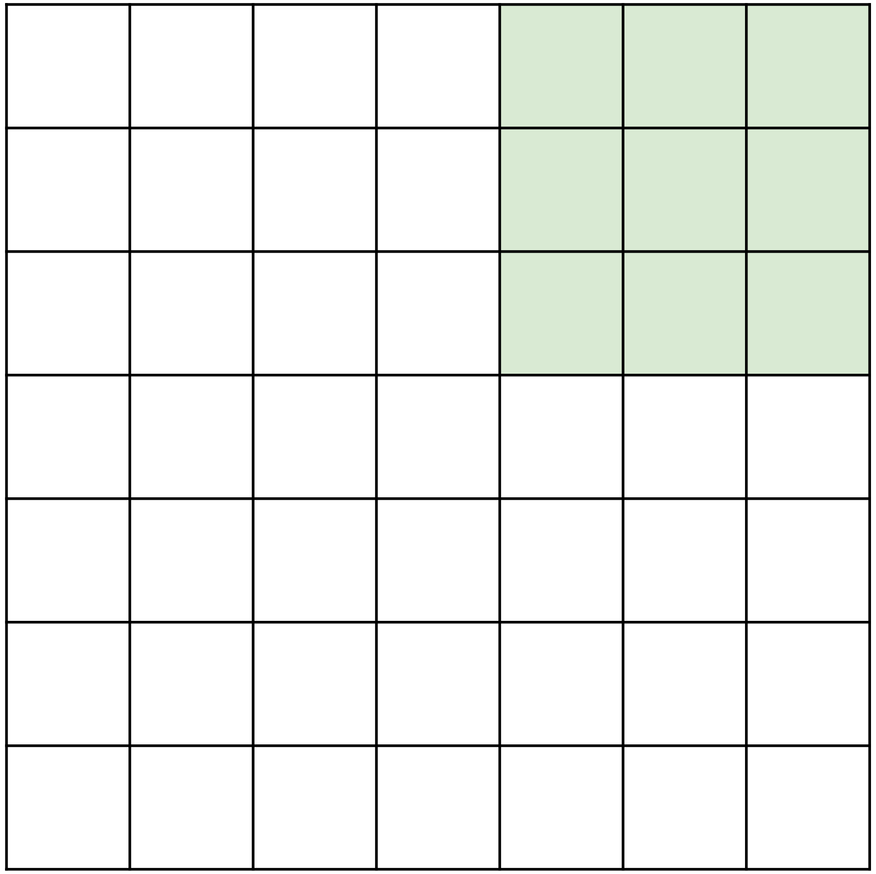
Input: 7x7
Filter: 3x3
Stride: 2

Strided Convolution



Input: 7x7
Filter: 3x3
Stride: 2

Strided Convolution



Input: 7x7

Filter: 3x3

Stride: 2

Output: 3x3

In general:

Input: W

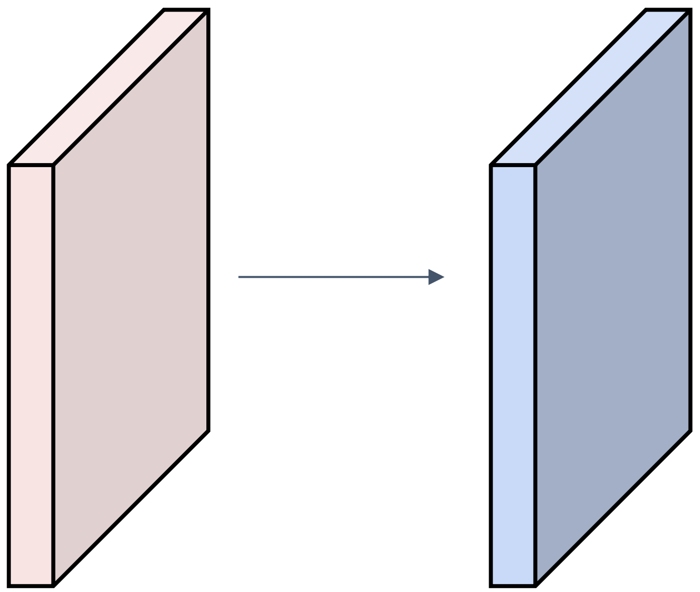
Filter: K

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 **3**x**5x5** tensors (75 elems); total = 75*10240 = **768K**

Max Pooling

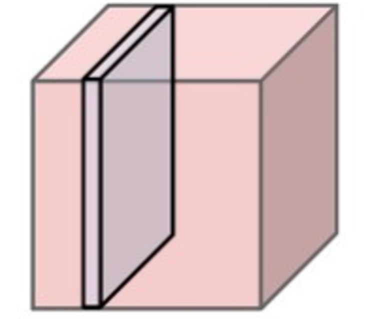
Single depth slice

x

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

y

64 x 224 x 224



Max pooling with 2x2 kernel size and stride 2

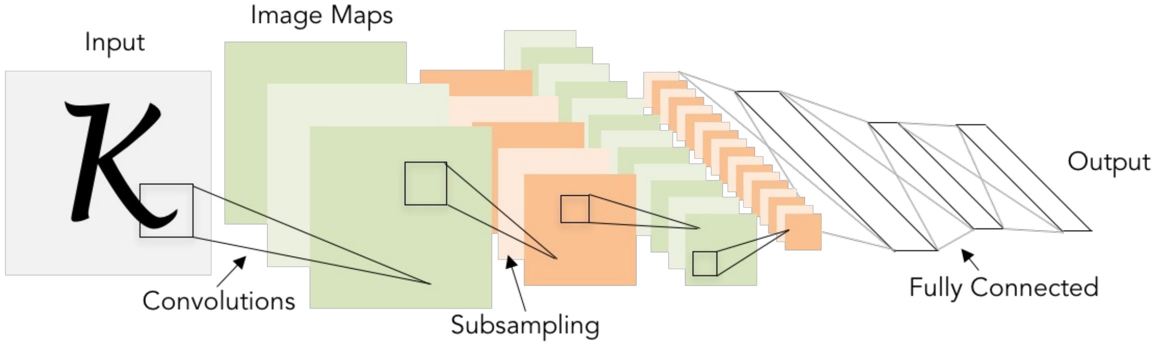
6	8
3	4

Introduces **invariance** to small spatial shifts

No learnable parameters!

Example: LeNet-5

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



As we go through the network:

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

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

Lecun et al, "Gradient-based learning applied to document recognition", 1998

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!

Ioffe and Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift”, ICML 2015

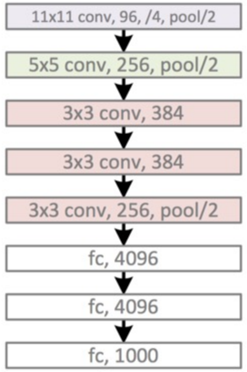
Example of Batchnorm in NN

```
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
```

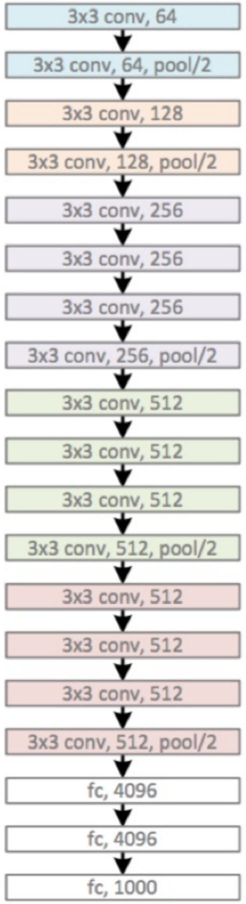
Lecture 8: Modern CNN Architectures

This Lecture: A Zoo of CNN Architectures

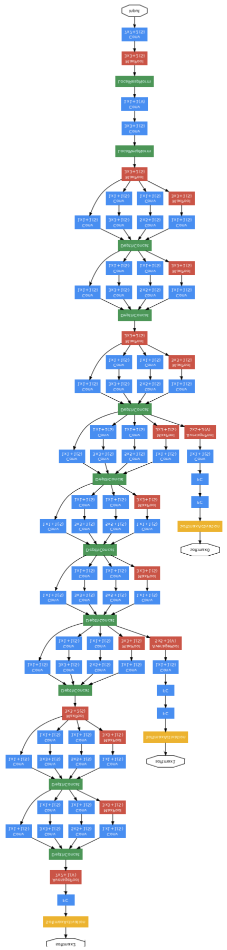
2012: AlexNet
5 layers



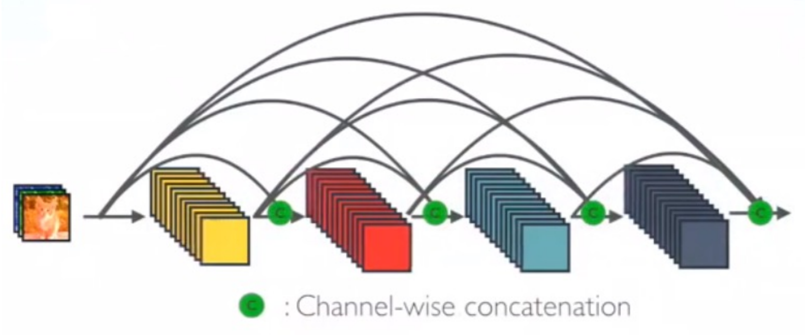
2014: VGG
16 layers



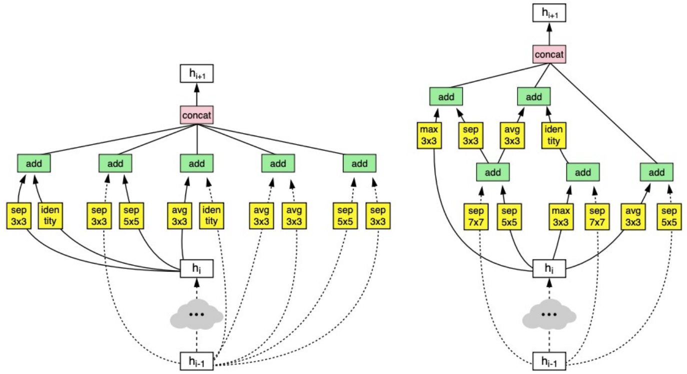
2015: GoogLeNet
22 layers



2016: ResNet
>100 layers



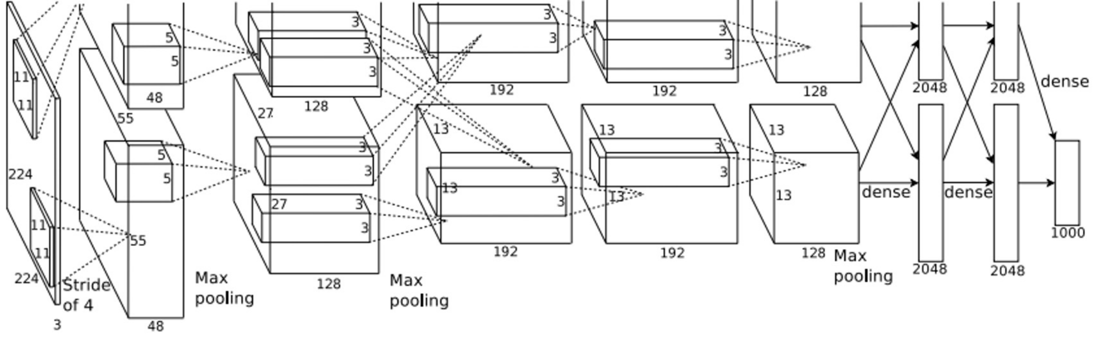
2017: DenseNet
>100 layers



2017: NASNet

https://github.com/osmr/imgclsmob

AlexNet

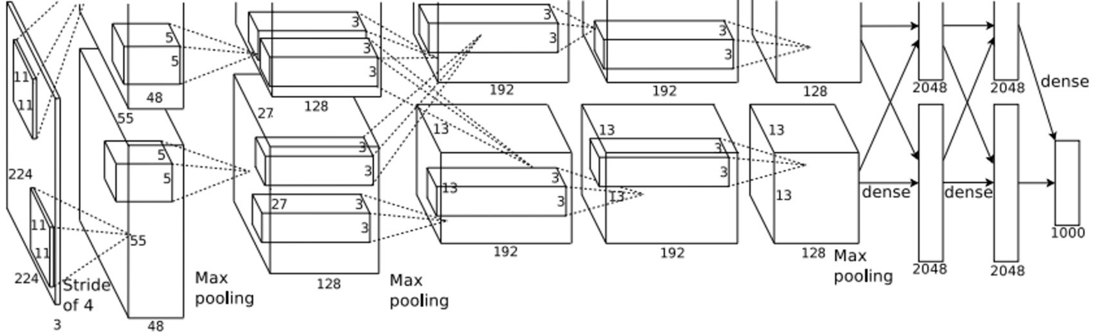


227 x 227 inputs
5 Convolutional layers
Max pooling
3 fully-connected layers
ReLU nonlinearities

Used “Local response normalization”; Not used anymore

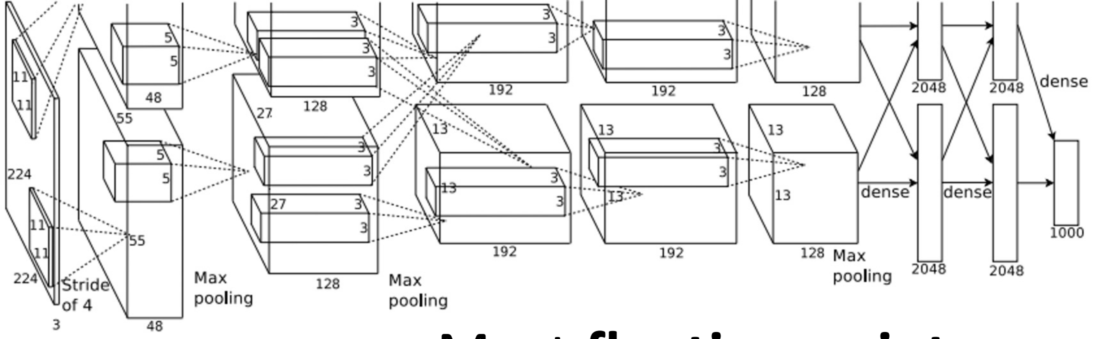
Trained on two GTX 580 GPUs – only 3GB of memory each! Model split over two GPUs

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

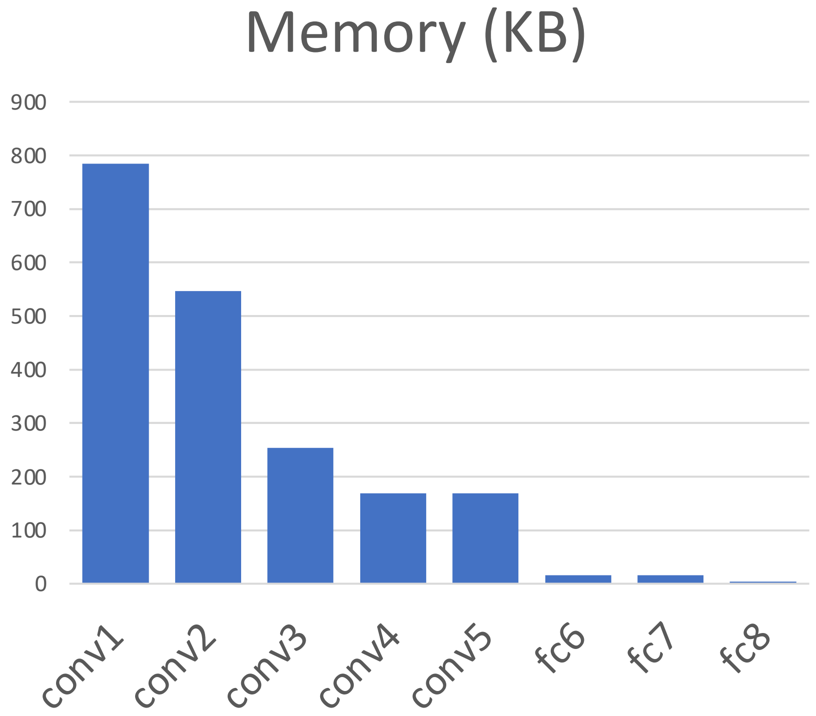
AlexNet

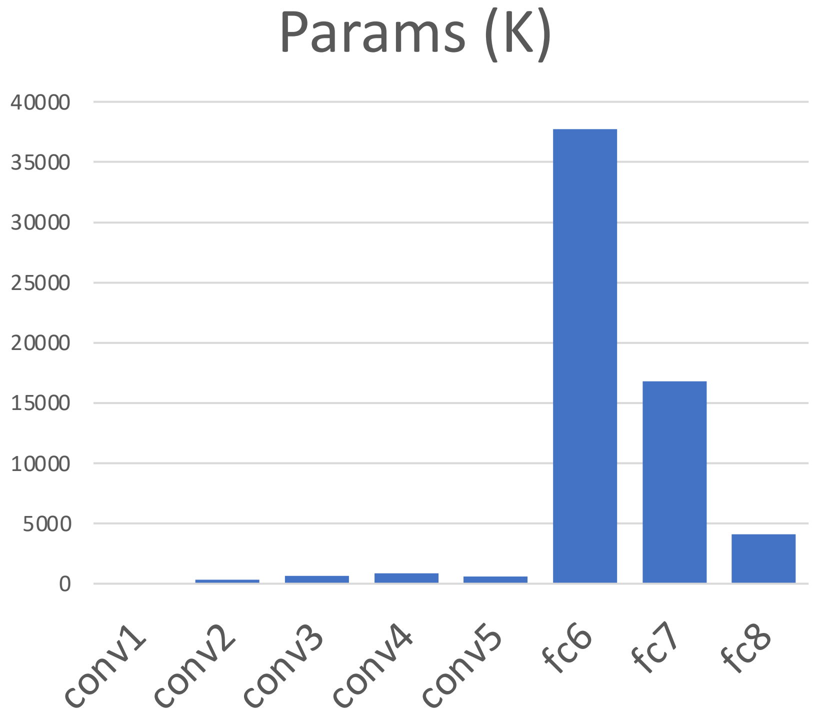


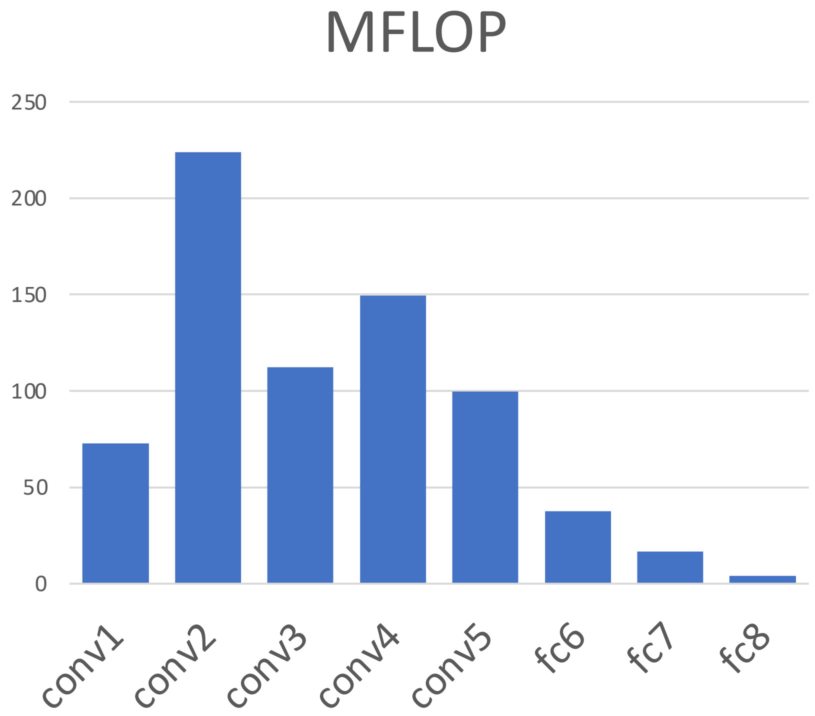
Most of the **memory usage** is in the early convolution layers

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

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







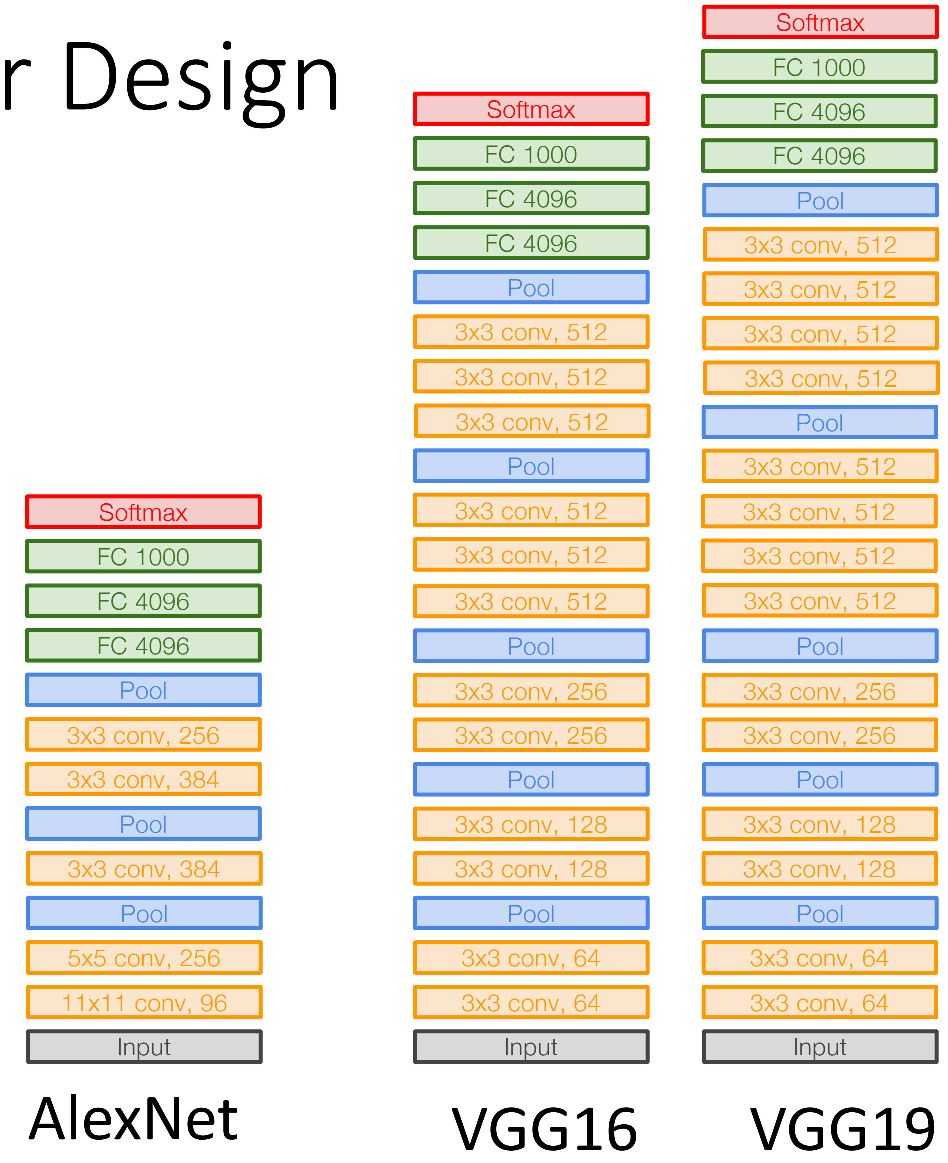
VGG: Deeper Networks, Regular Design

<u>VGG Design rules:</u>

All conv are 3x3 stride 1 pad 1

All max pool are 2x2 stride 2

After pool, double #channels

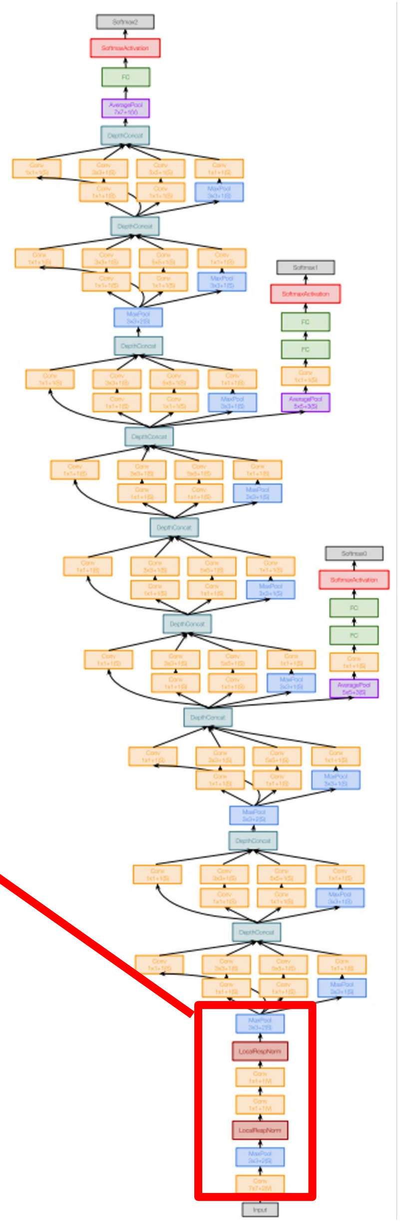


Simonyan and Zissermann, “Very Deep Convolutional Networks for Large-Scale Image Recognition”, ICLR 2015

GoogLeNet: Aggressive Stem

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

	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H/W	memory (KB)	params (K)	flop (M)
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

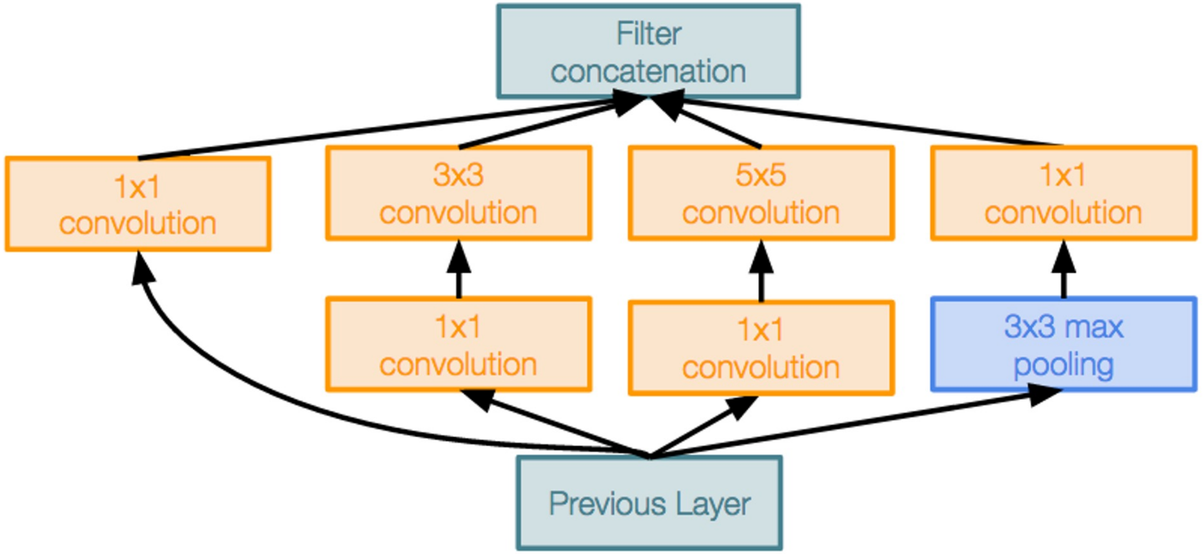
Compare VGG-16:
Memory: 42.9 MB (5.7x)
Params: 1.1M (8.9x)
MFLOP: 7485 (17.8x)

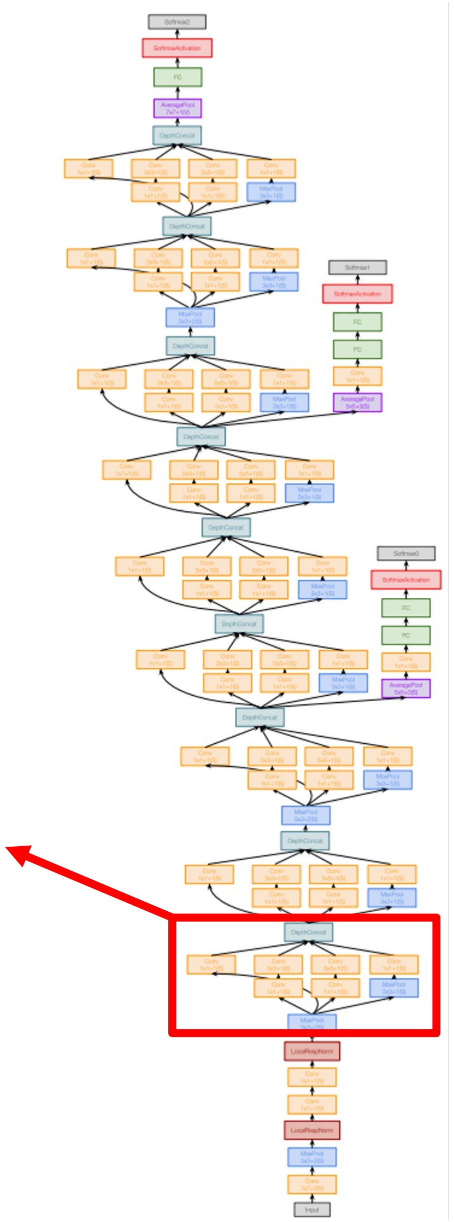
Szegedy et al, "Going deeper with convolutions", CVPR 2015

GoogLeNet: Inception Module

Inception module
Local unit with parallel branches

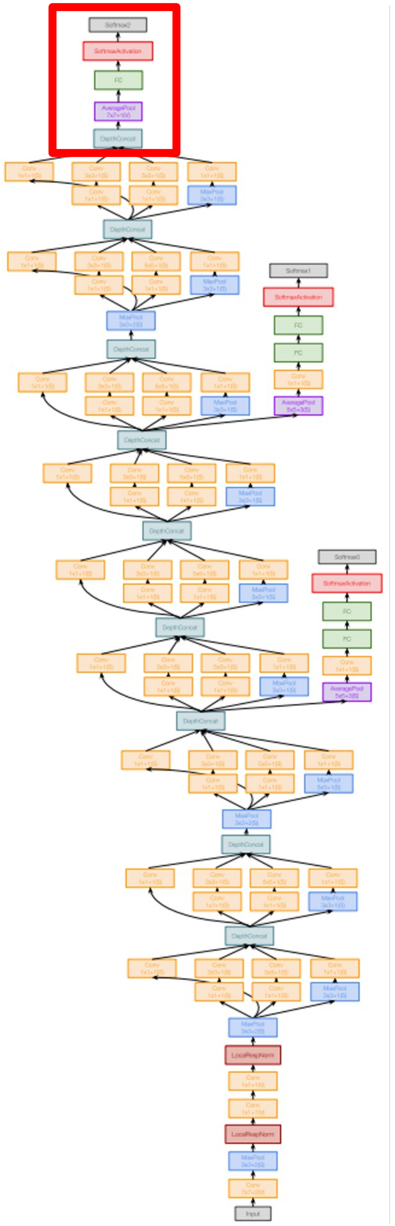
Local structure repeated many times throughout the network





Szegedy et al, “Going deeper with convolutions”, CVPR 2015

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!)

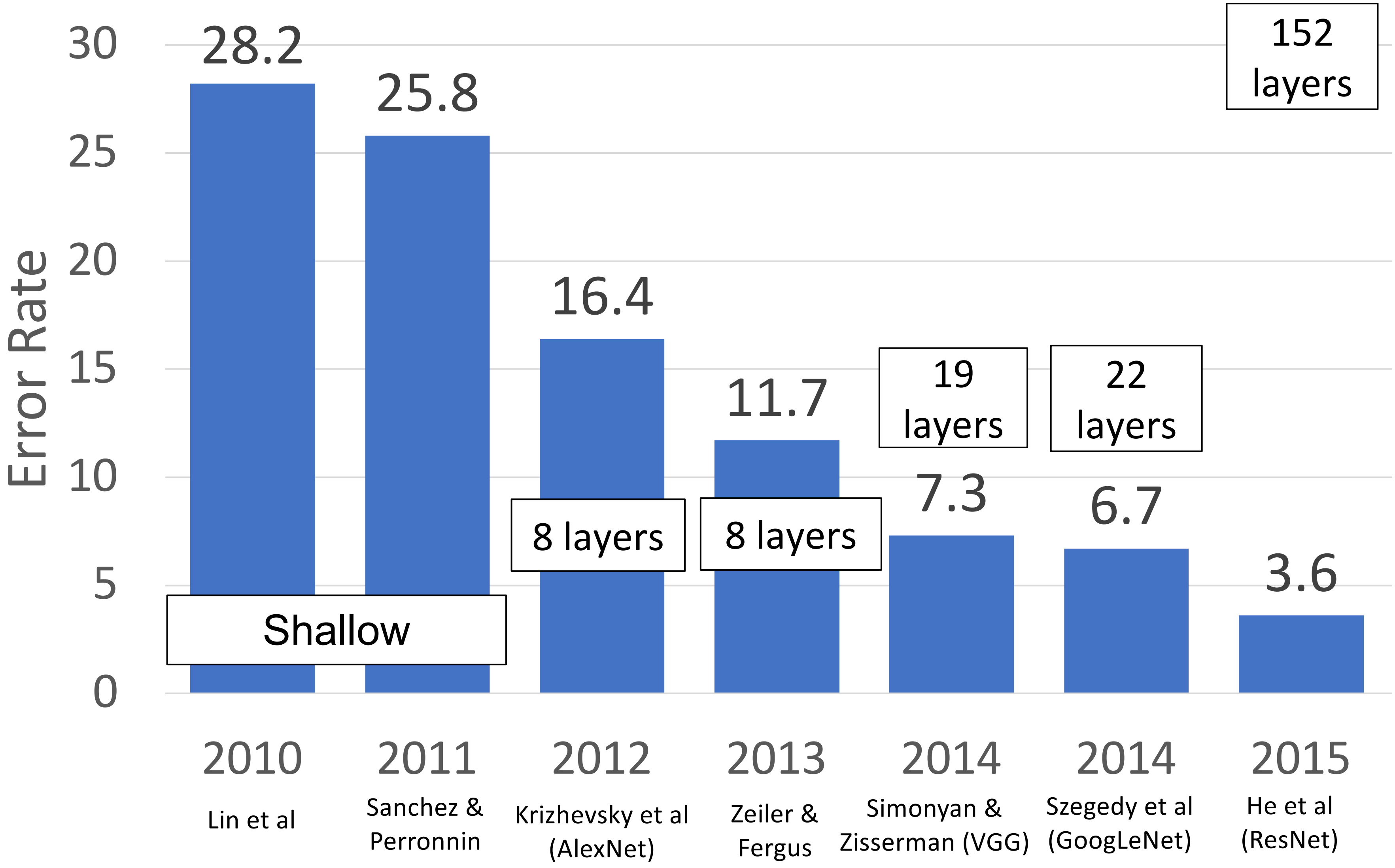
e.g. GAP: 1024 x 7 x 7 -> 1024 x 1

	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

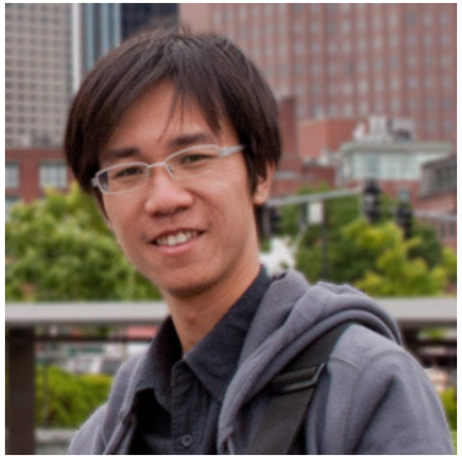
Compare with VGG-16:

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

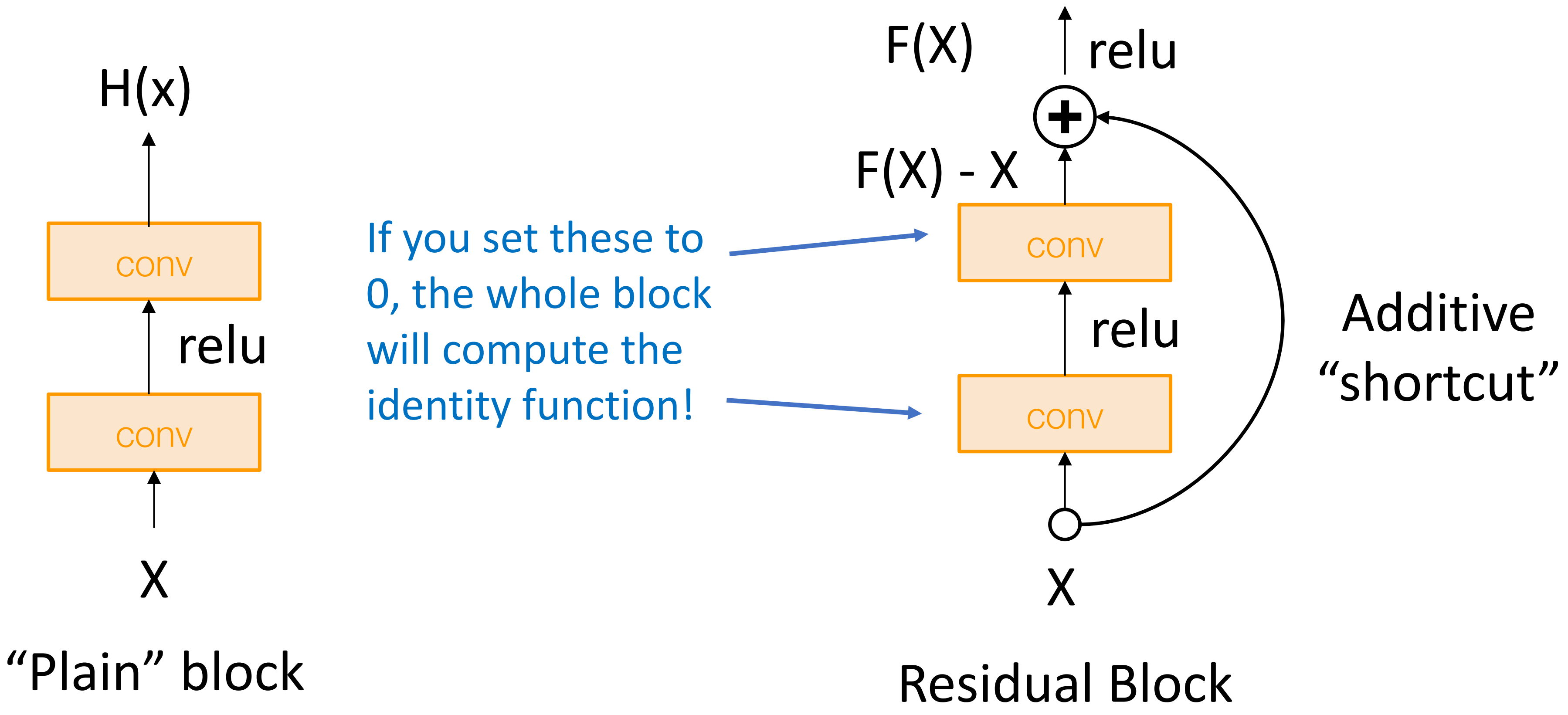


Kaiming He



Residual Networks

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



He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

Residual Block Implementation

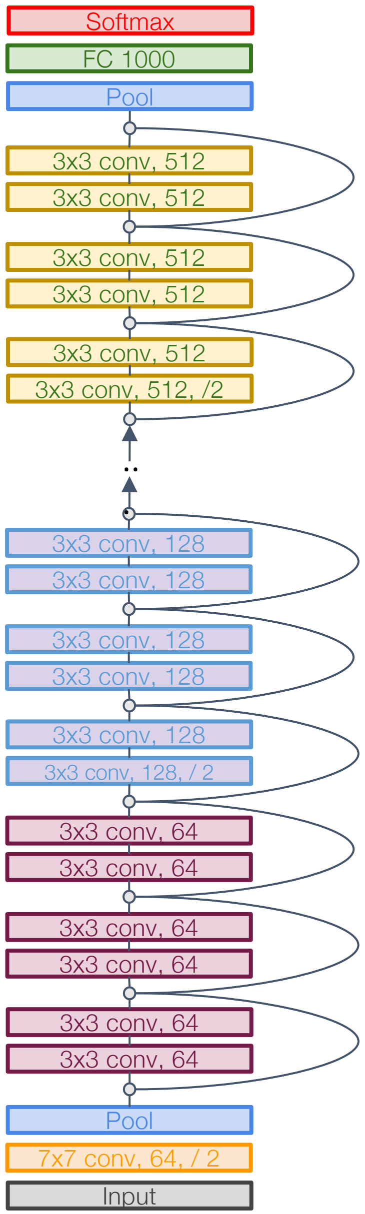
```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l

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)
```

D2L Chapter 7.6

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

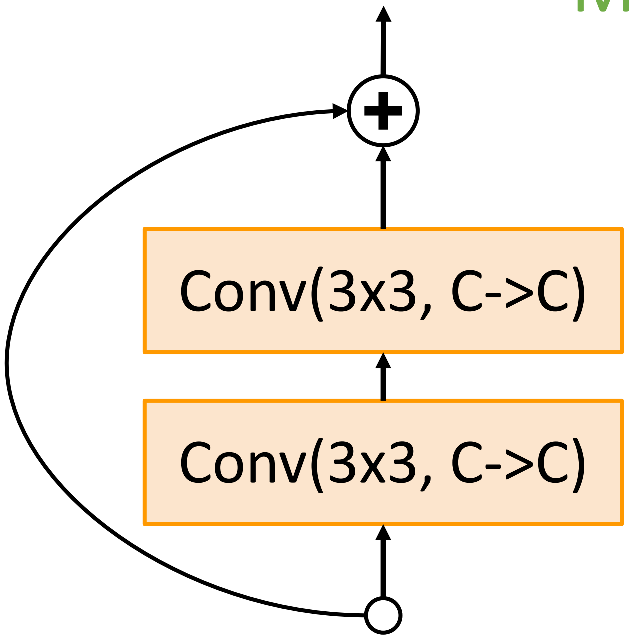
ImageNet top-5 error: 10.92
GFLOP: 1.8



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!

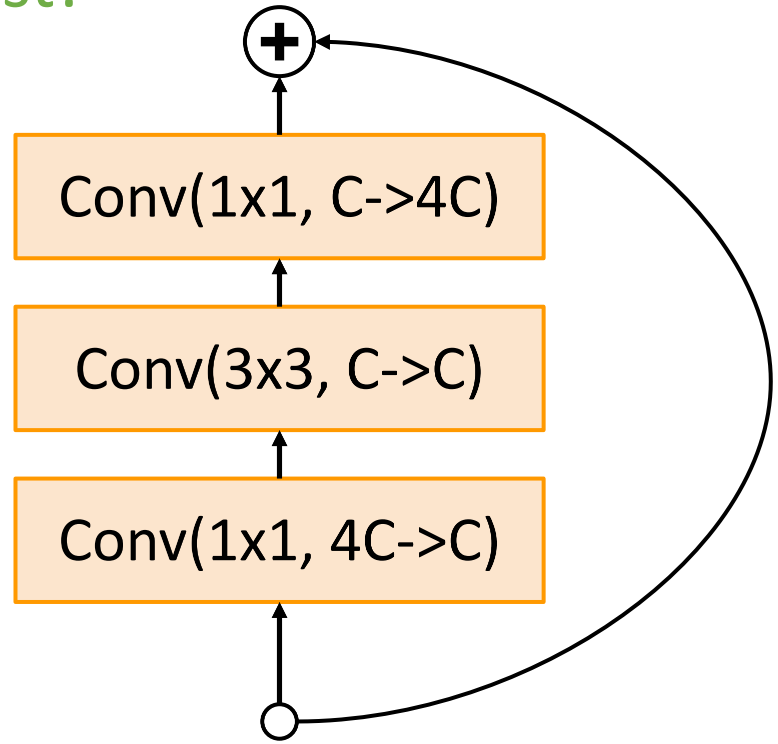


FLOPs: $9HWC^2$

FLOPs: $9HWC^2$

“Basic”
Residual block

Total FLOPs:
$18HWC^2$



FLOPs: $4HWC^2$

FLOPs: $9HWC^2$

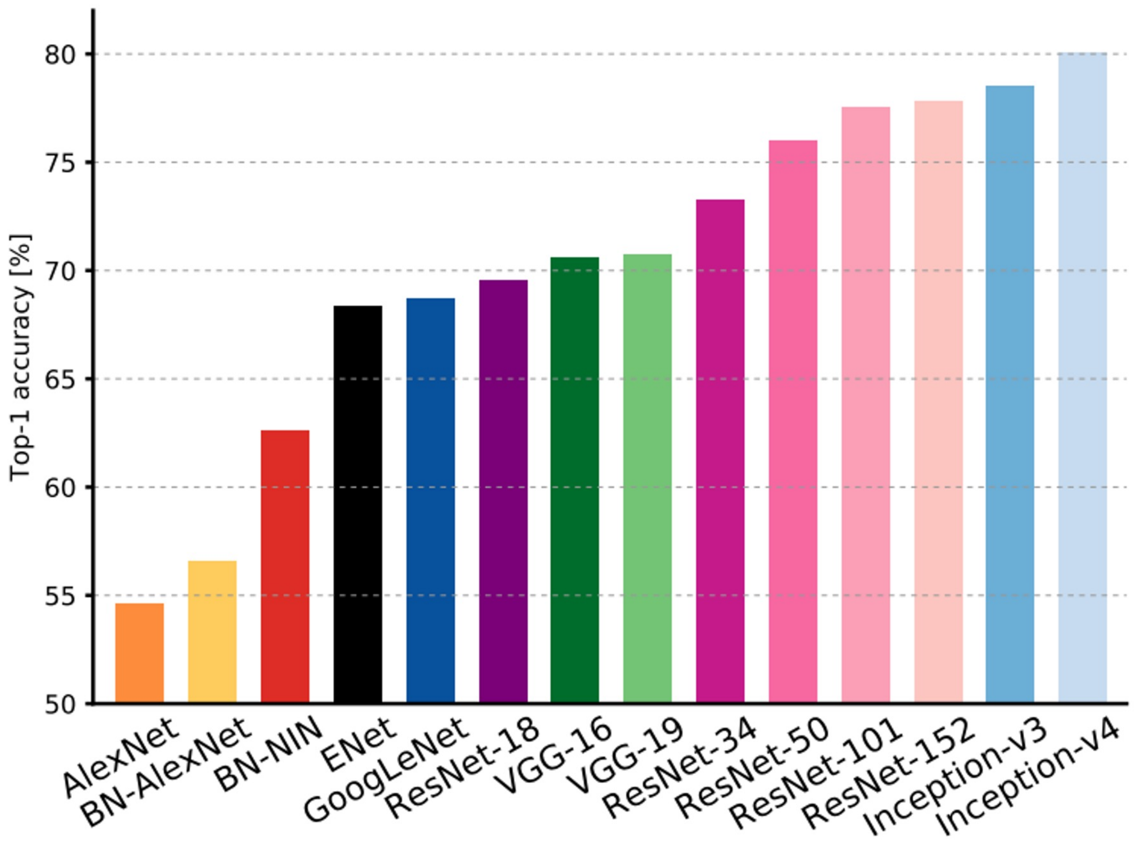
FLOPs: $4HWC^2$

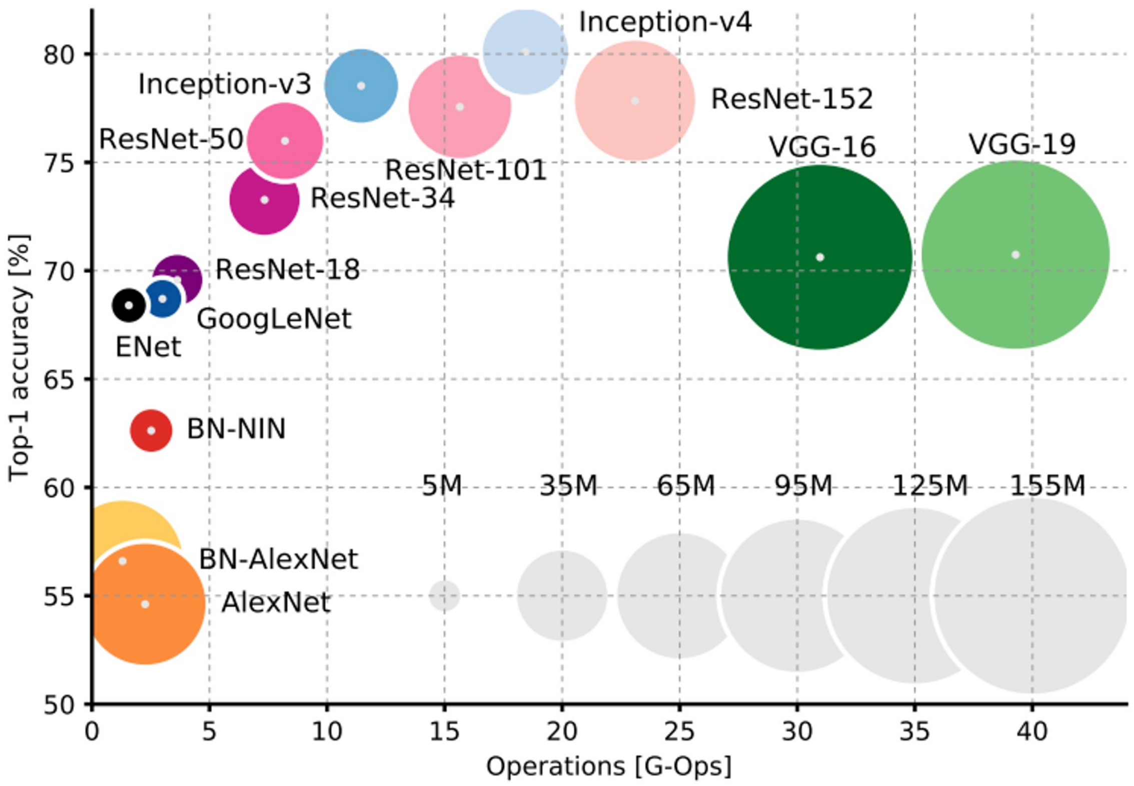
Total FLOPs:
$17HWC^2$

“Bottleneck”
Residual block

He et al, “Deep Residual Learning for Image Recognition”, CVPR 2016

Comparing Complexity

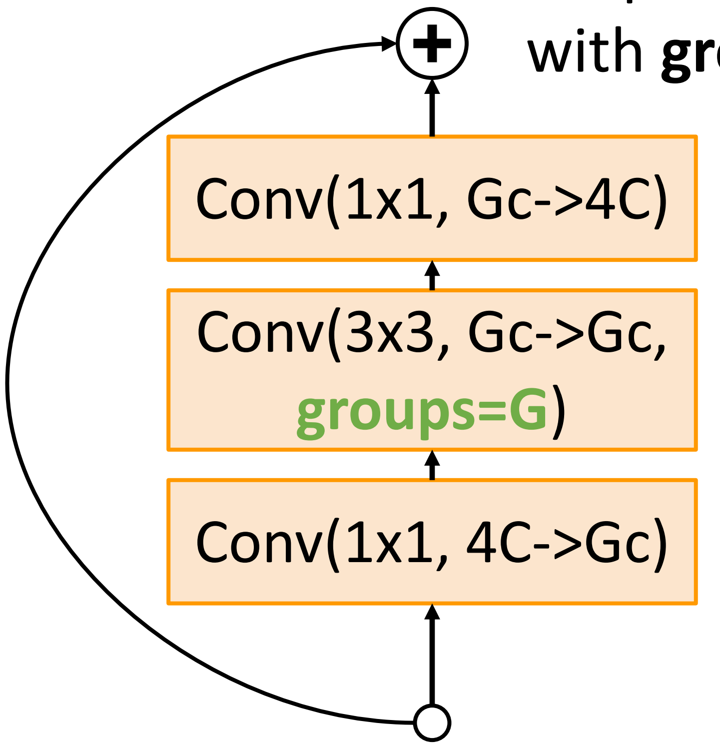




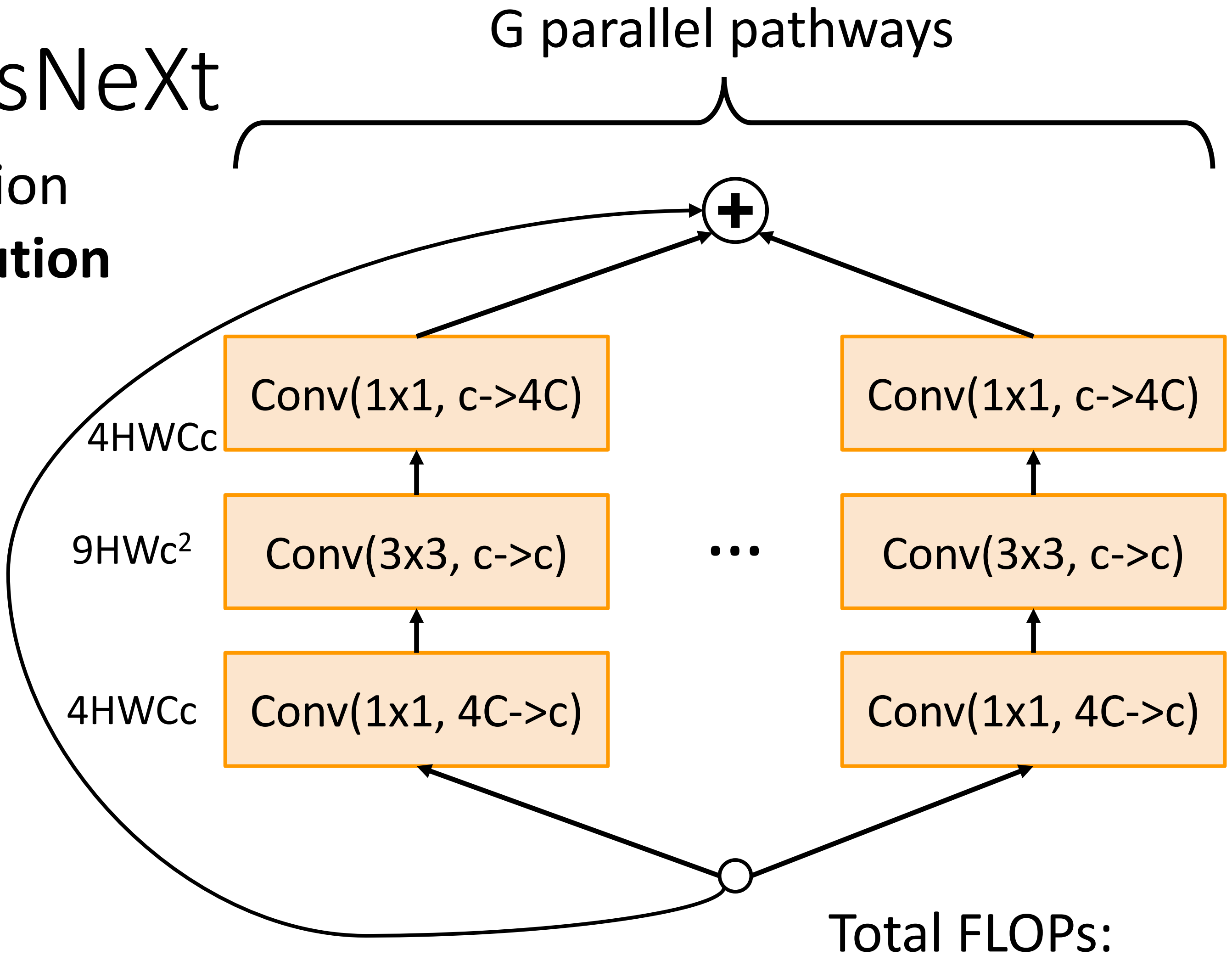
Canziani et al, "An analysis of deep neural network models for practical applications", 2017

Improving ResNets: ResNeXt

Equivalent formulation with **grouped convolution**



ResNeXt block:
Grouped convolution



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

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

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

Xie et al, "Aggregated residual transformations for deep neural networks", CVPR 2017

Grouped Convolution

Convolution with groups=1:
Normal convolution

Input: C_{in} x H x W
Weight: C_{out} x C_{in} x K x K
Output: C_{out} x H’ x W’
FLOPs: $C_{out}C_{in}K^2HW$

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

Input: C_{in} x H x W

Split

Group 1:
(C_{in} / 2) x H x W

Conv(K x K, C_{in}/2 -> C_{out}/2)

Out 1:
(C_{out} / 2) x H’ x W’

Group 2:
(C_{in} / 2) x H x W

Conv(K x K, C_{in}/2 -> C_{out}/2)

Out 2:
(C_{out} / 2) x H’ x W’

Concat

Output: C_{out} x H’ x W’

Research opportunity @ VAIL

Vision and Autonomy Intelligence Lab

https://vail-ucla.github.io

- Accepting two or three highly motivated students
- Research Topics
  - Simulation
  - Digital Human
  - Robot Learning
  - Gen AI
  - 3D Vision

What we are looking for

- Highly motivated young guns
- Skilled and capable: hacker, coder, quick at making things happen
- Committed:
  - Willing to spend 10 hours per week on the research project
  - Working on only one project at a time
- Responsible
- You will:
  - Work with a graduate student(s) on a research project that can lead to a publication
  - Have access to reasonable amount of good GPU cards
  - Get mentorship and guidance

- If you are interested in any topics above, please sign up by the end of this week
- Sign up link: https://forms.gle/Zcc9ABcoYp4oPE659
- Along with my graduate students, we will evaluate the applications, and we will invite the shortlisted students for interviews

Lecture 9: Training Neural Networks

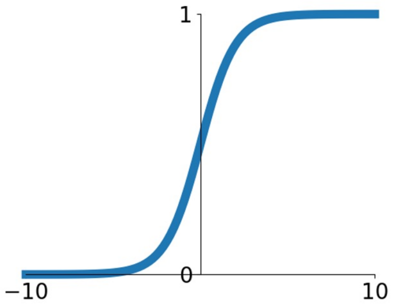
Overview

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

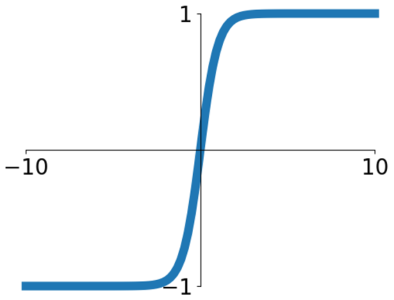
Sigmoid

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



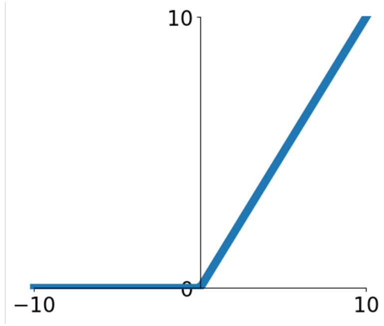
tanh

$\tanh(x)$



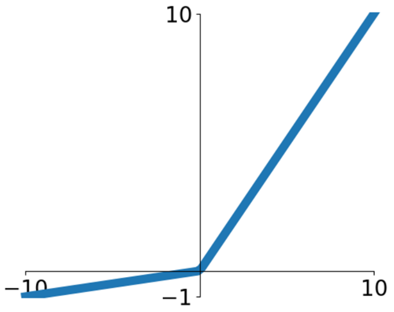
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

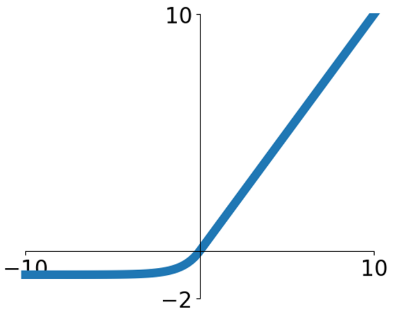


Maxout

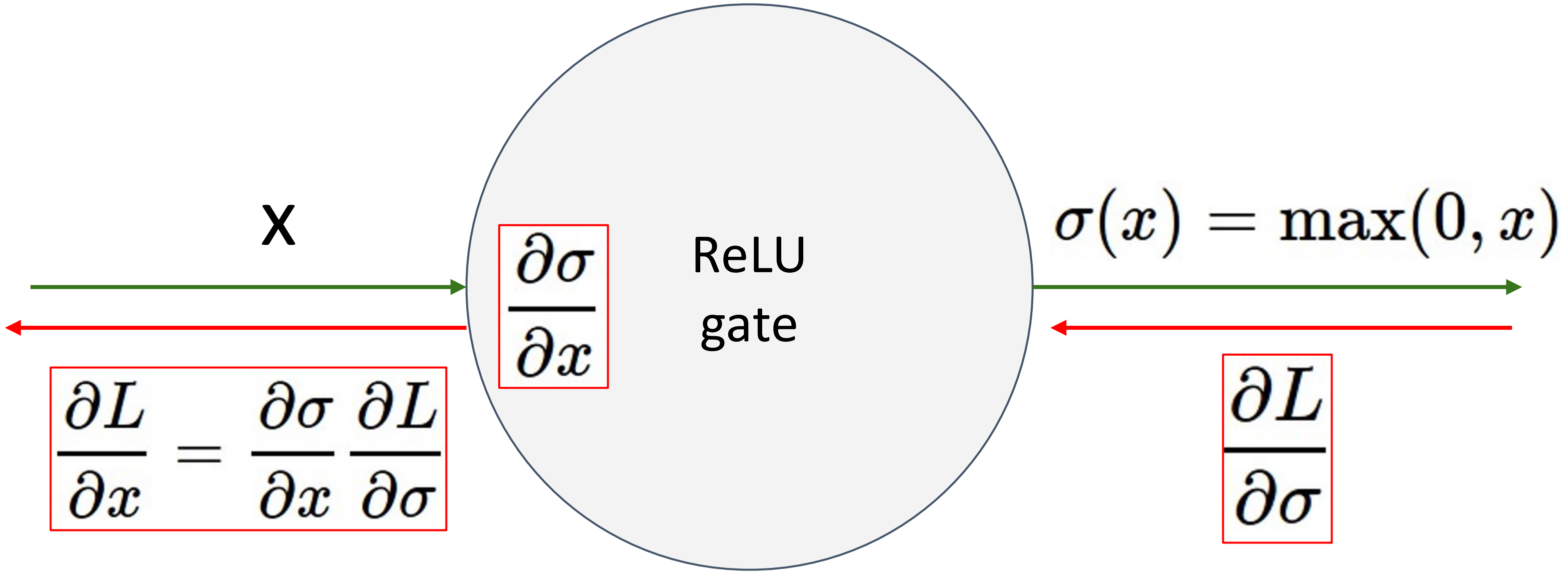
$\max(w_1^T x + b_1, w_2^T x + b_2)$

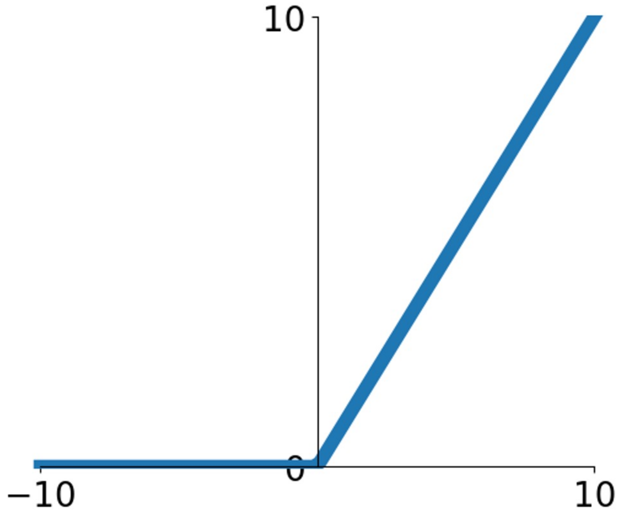
ELU

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



Activation Functions: ReLU





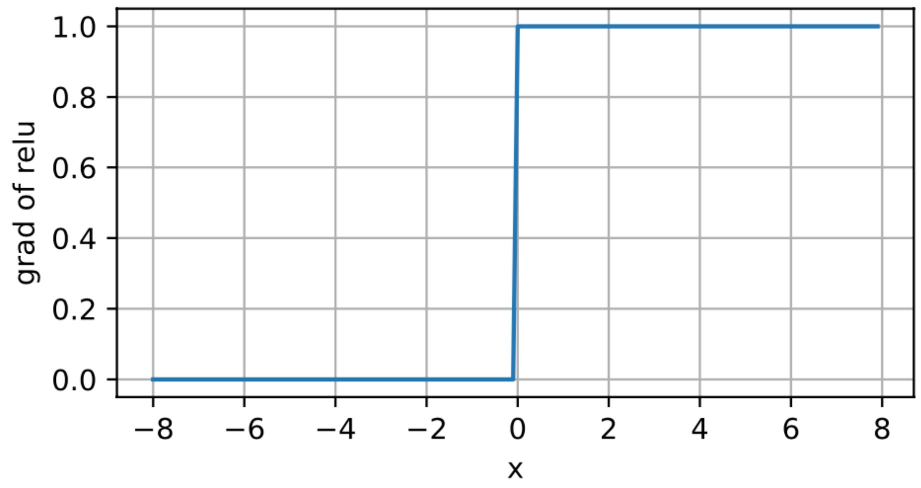
What happens when x = -8?

What happens when x = 0?

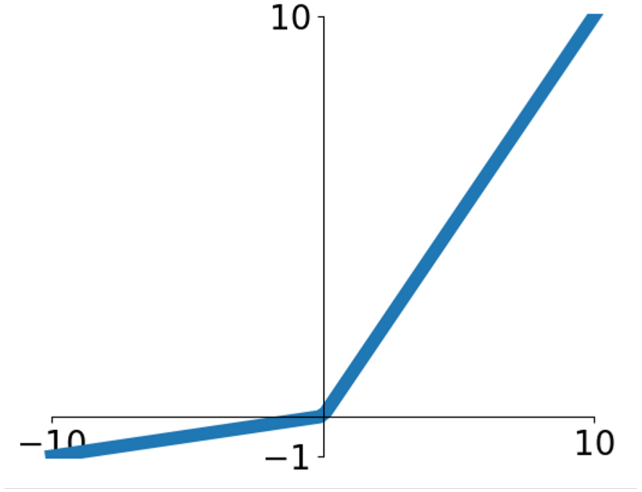
What happens when x = 8?

dead ReLU will never activate

=> never update



Activation Functions: Leaky ReLU



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

Leaky ReLU

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

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

Maas et al, "Rectifier Nonlinearities Improve Neural Network Acoustic Models", ICML 2013

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!

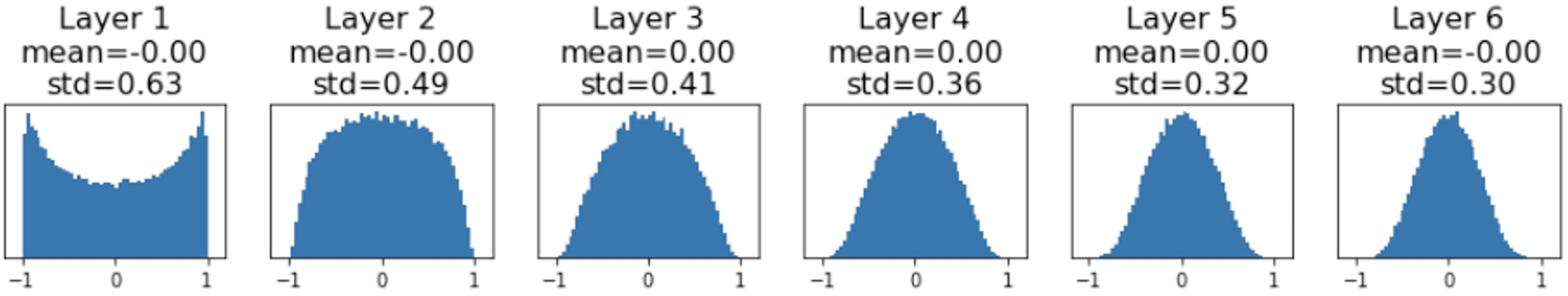
Weight Initialization: Xavier Initialization

```
dims = [4096] * 7
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.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

"Just right": Activations are nicely scaled for all layers!

For conv layers, Din is kernel_size^2 * input_channels



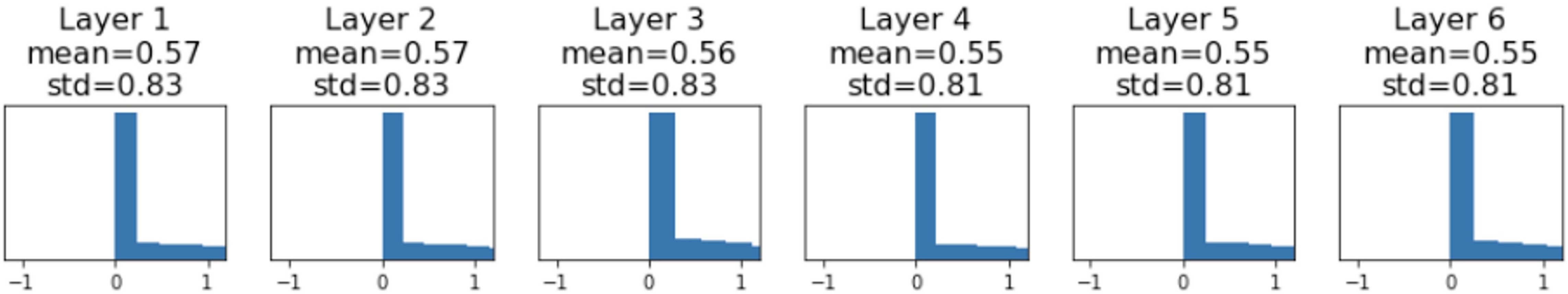
Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

Weight Initialization: Kaiming Initialization

```
dims = [4096] * 7
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)
```

ReLU correction: std = sqrt(2 / Din)

"Just right" – activations nicely scaled for all layers

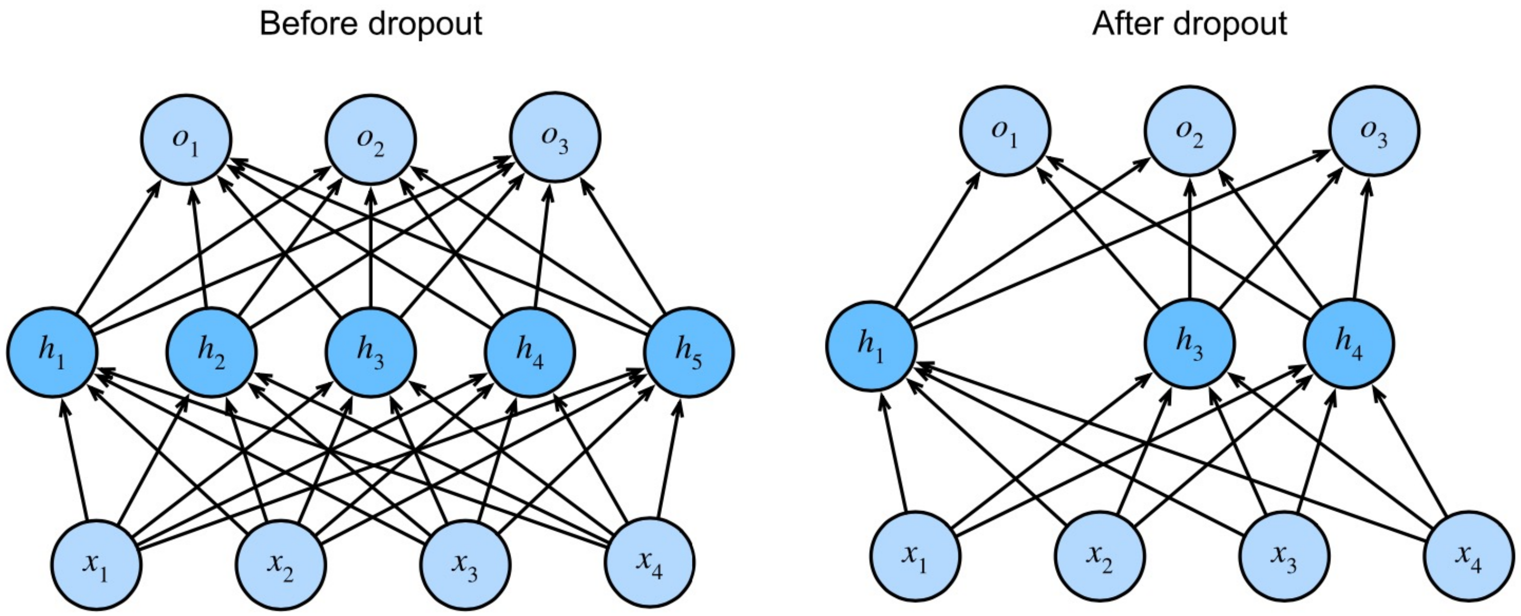


See this paper for detailed derivation!

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



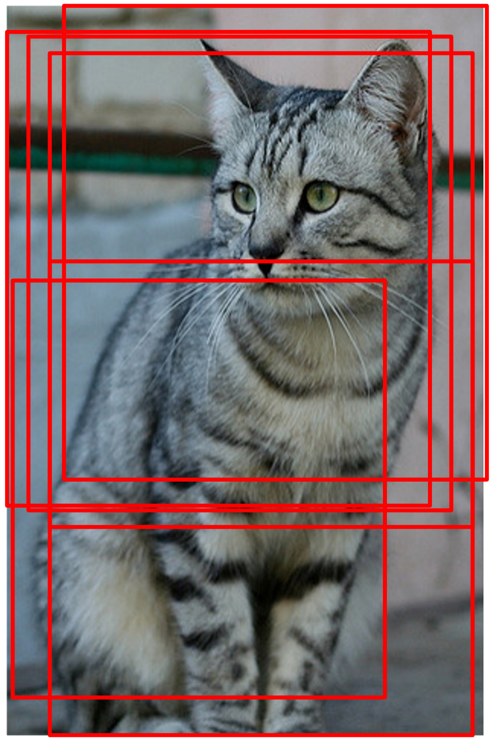
Srivastava et al, "Dropout: A simple way to prevent neural networks from overfitting", JMLR 2014

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 x 224 patch



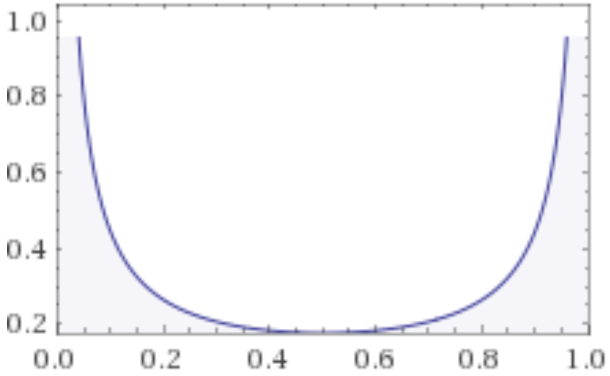
Data Augmentation: more image distortions

Random mix/combinations of :

- translation
- rotation
- stretching
- shearing,
- lens distortions,

… (go crazy)

Regularization: Mixup



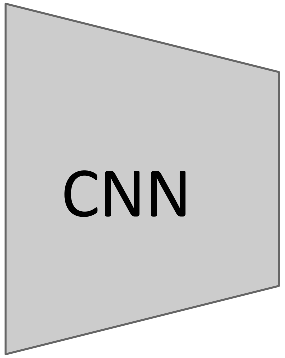
Sample blend probability from a beta distribution Beta(a, b) with a=b≈0 so blend weights are close to 0/1

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

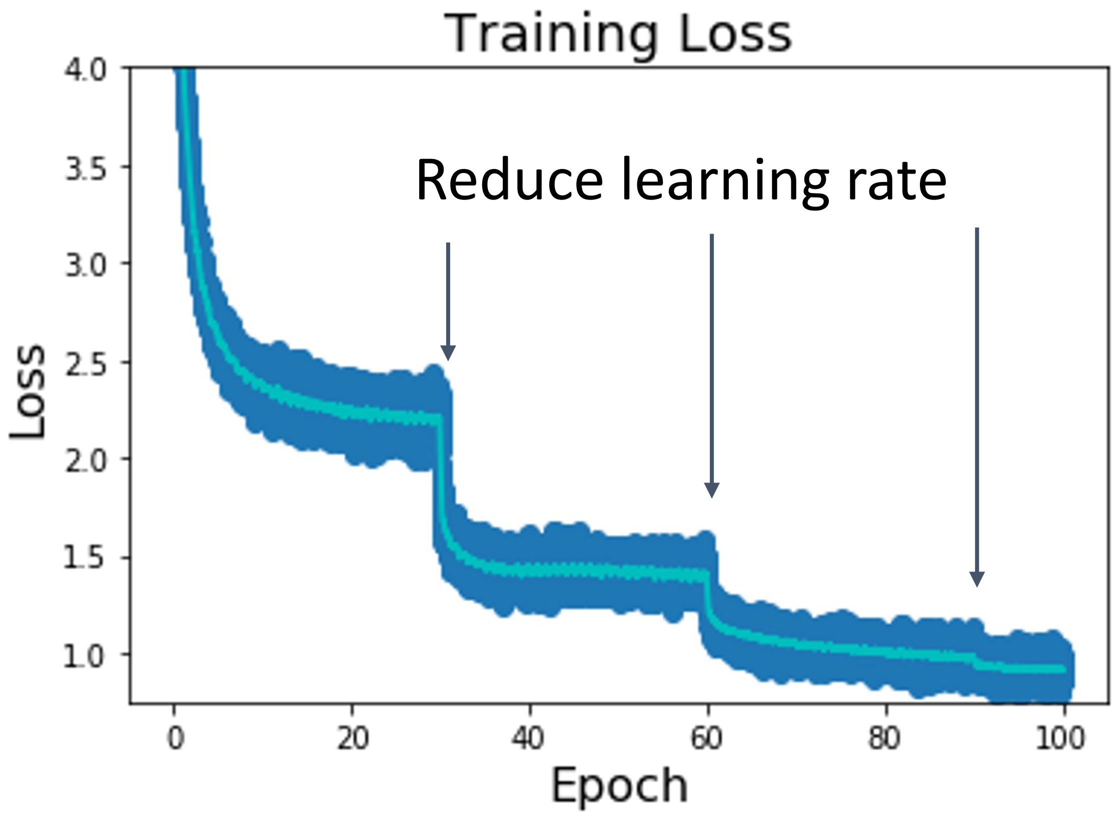
Randomly blend the pixels of pairs of training images, e.g. 40% cat, 60% dog

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

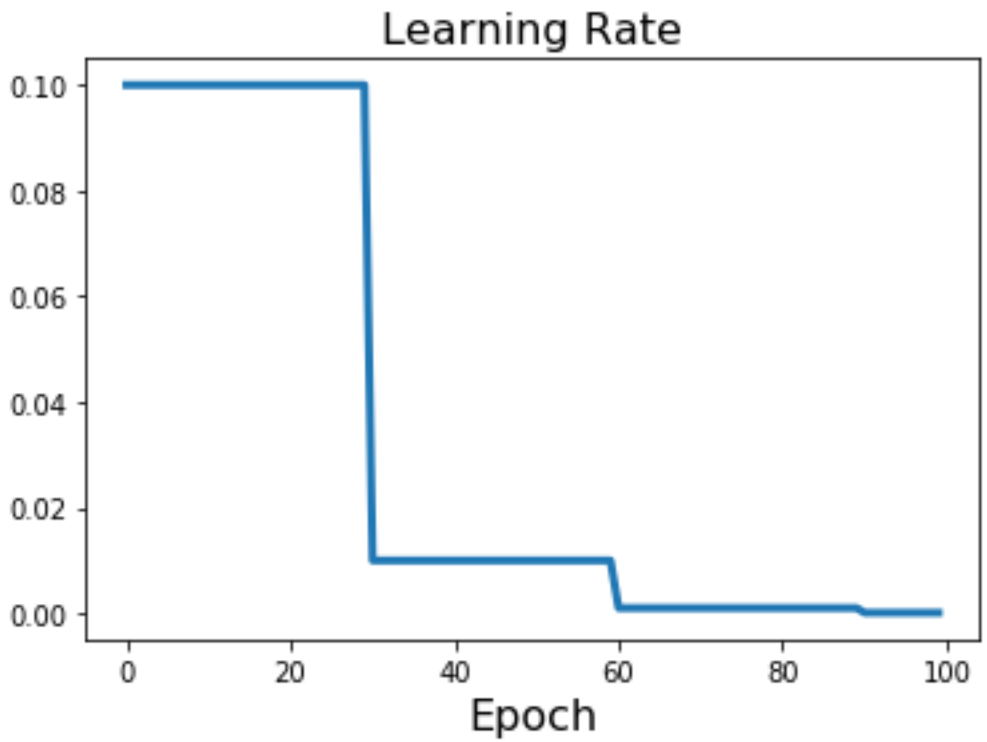
Why it works? some interesting analysis in the paper:

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

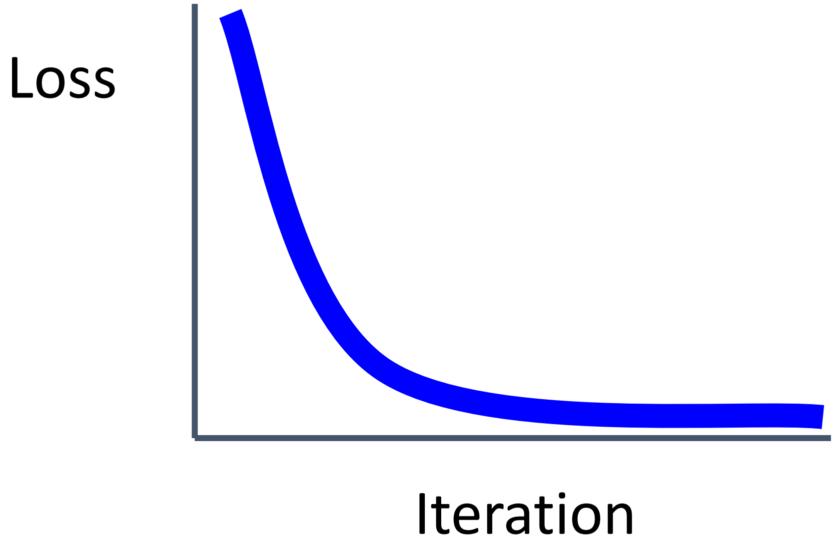
Learning Rate Decay: Step (most common)

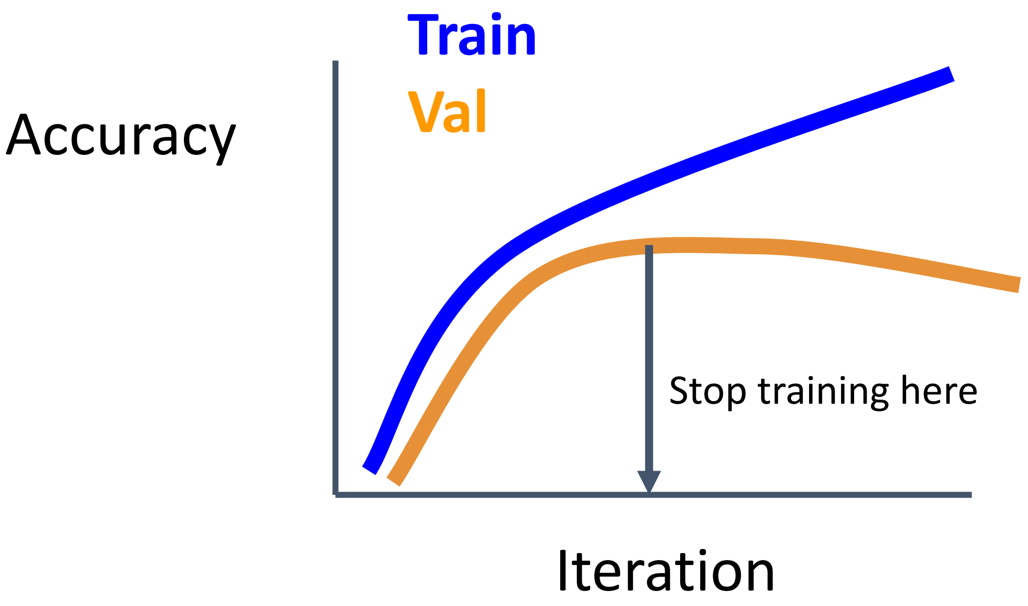


Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.



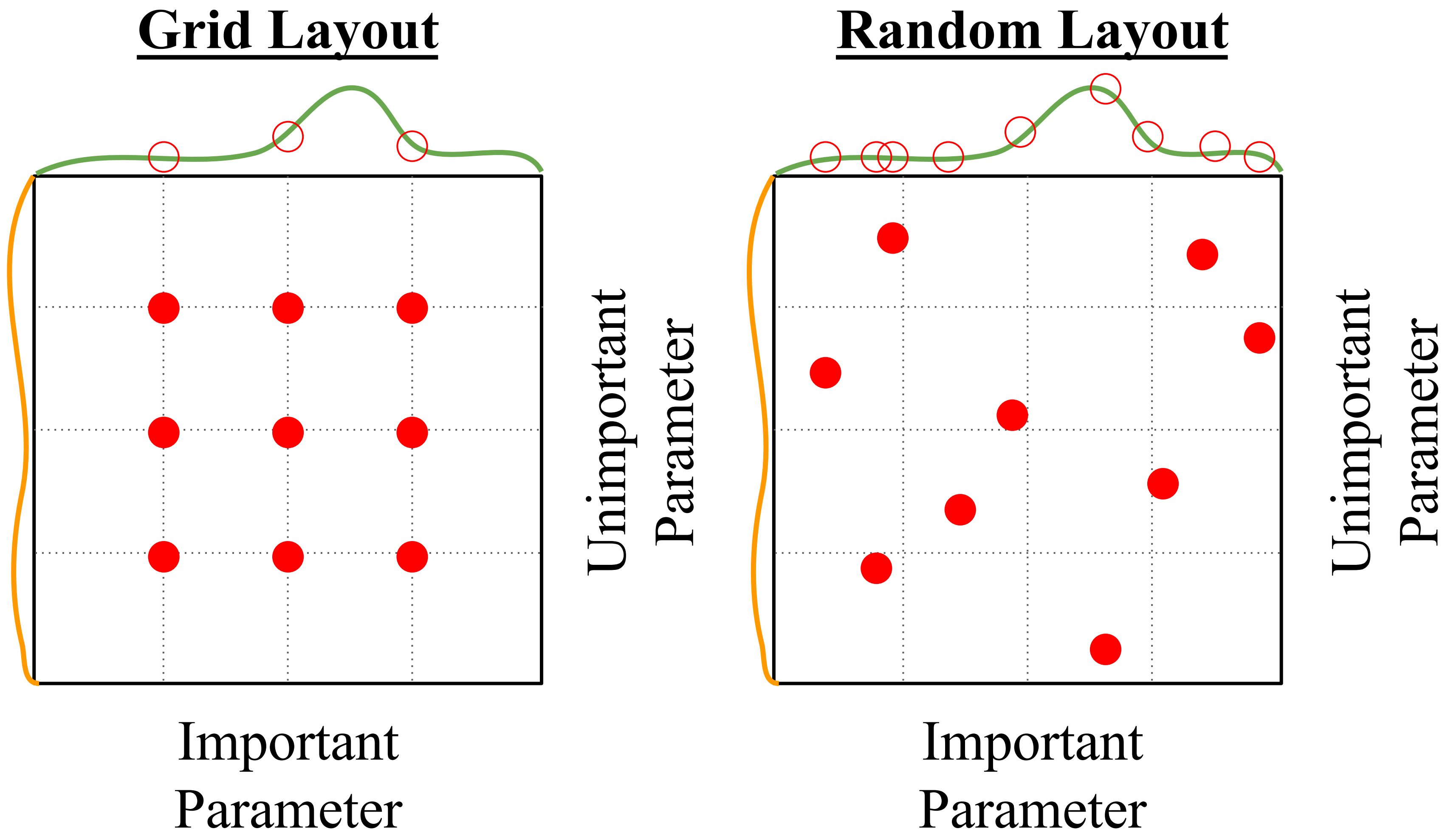
How long to train? **Early Stopping**





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



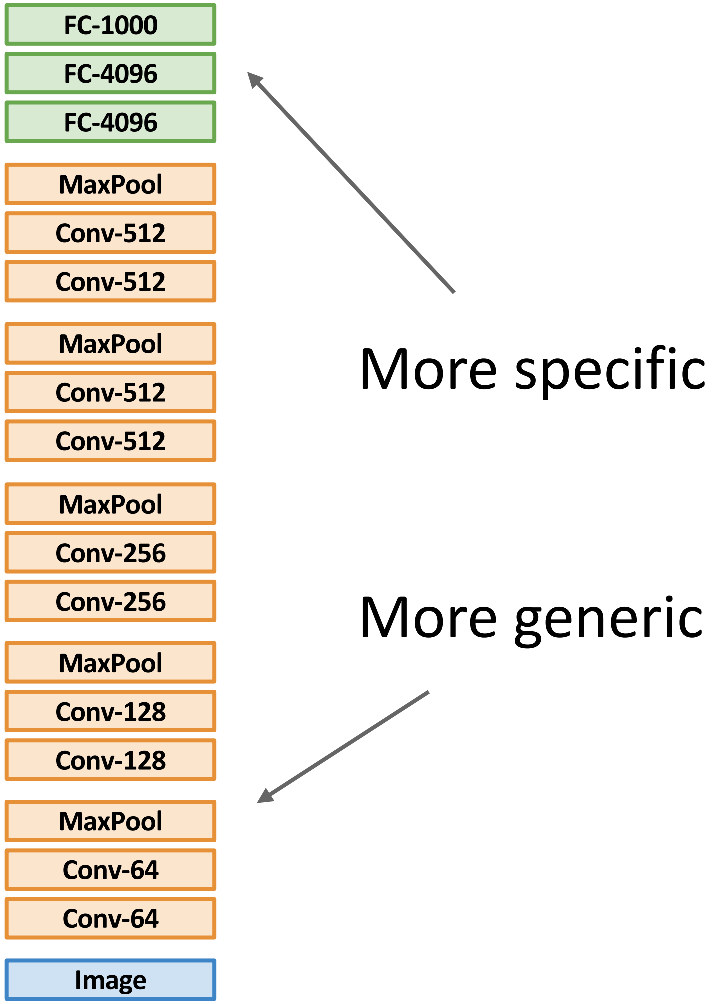
Bergstra and Bengio, “Random Search for Hyper-Parameter Optimization”, JMLR 2012

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

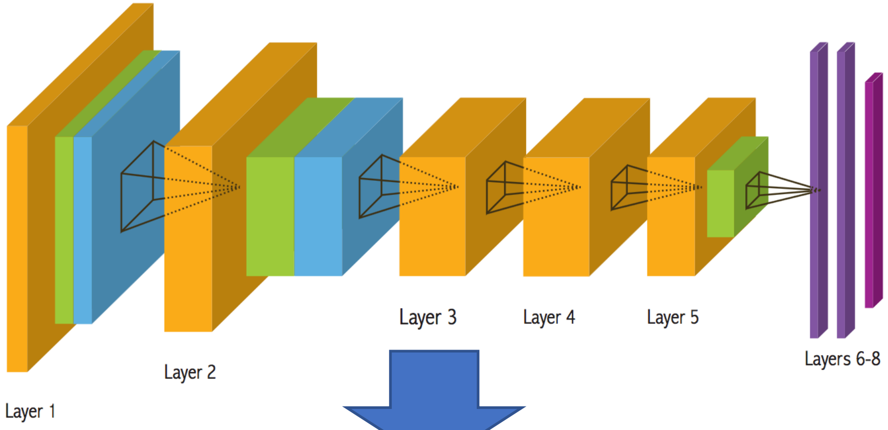


	Dataset similar to ImageNet	Dataset very different from ImageNet
very little data (10s to 100s)	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different layers
quite a lot of data (100s to 1000s)	Finetune a few layers	Finetune a larger number of layers

Lecture 10: Visualizing and Understanding Neural Network

What’s going on inside ConvNet?

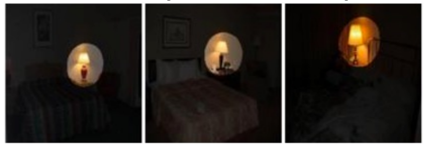
2. Why is this output?



Cafeteria (0.9)

1. What have been learned inside?

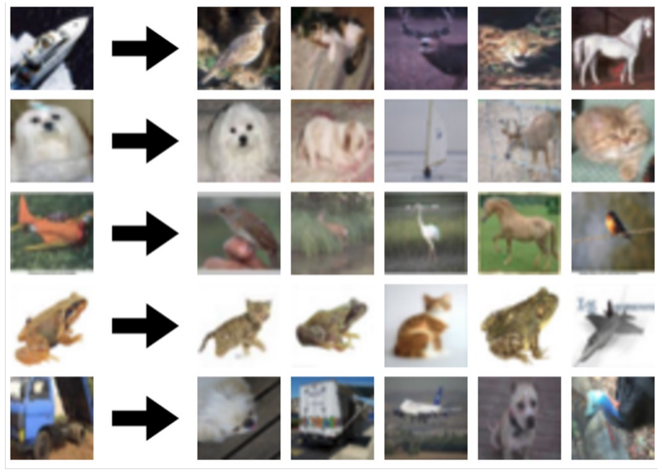
Unit2 at Layer4: Lamp



Unit5 at Layer3 : Trademark

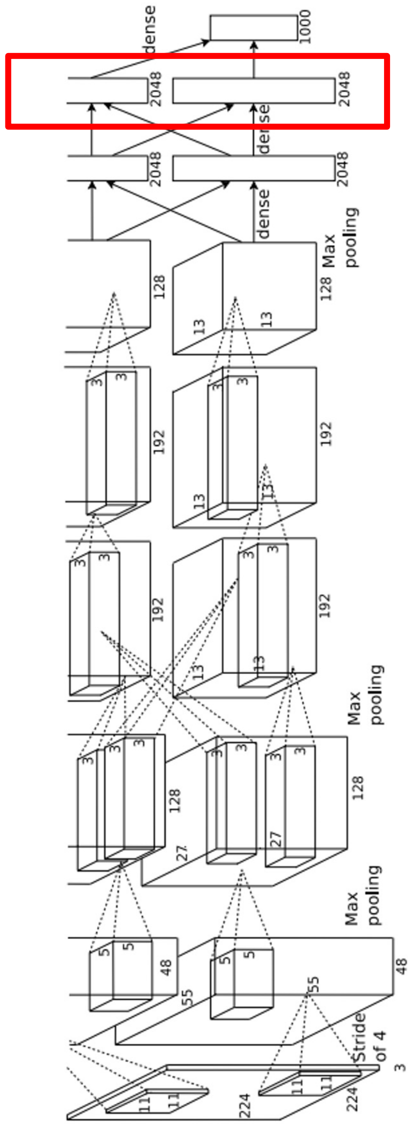
Last Layer: Nearest Neighbors

Recall: Nearest neighbors in pixel space



Test image

L2 Nearest neighbors in feature space



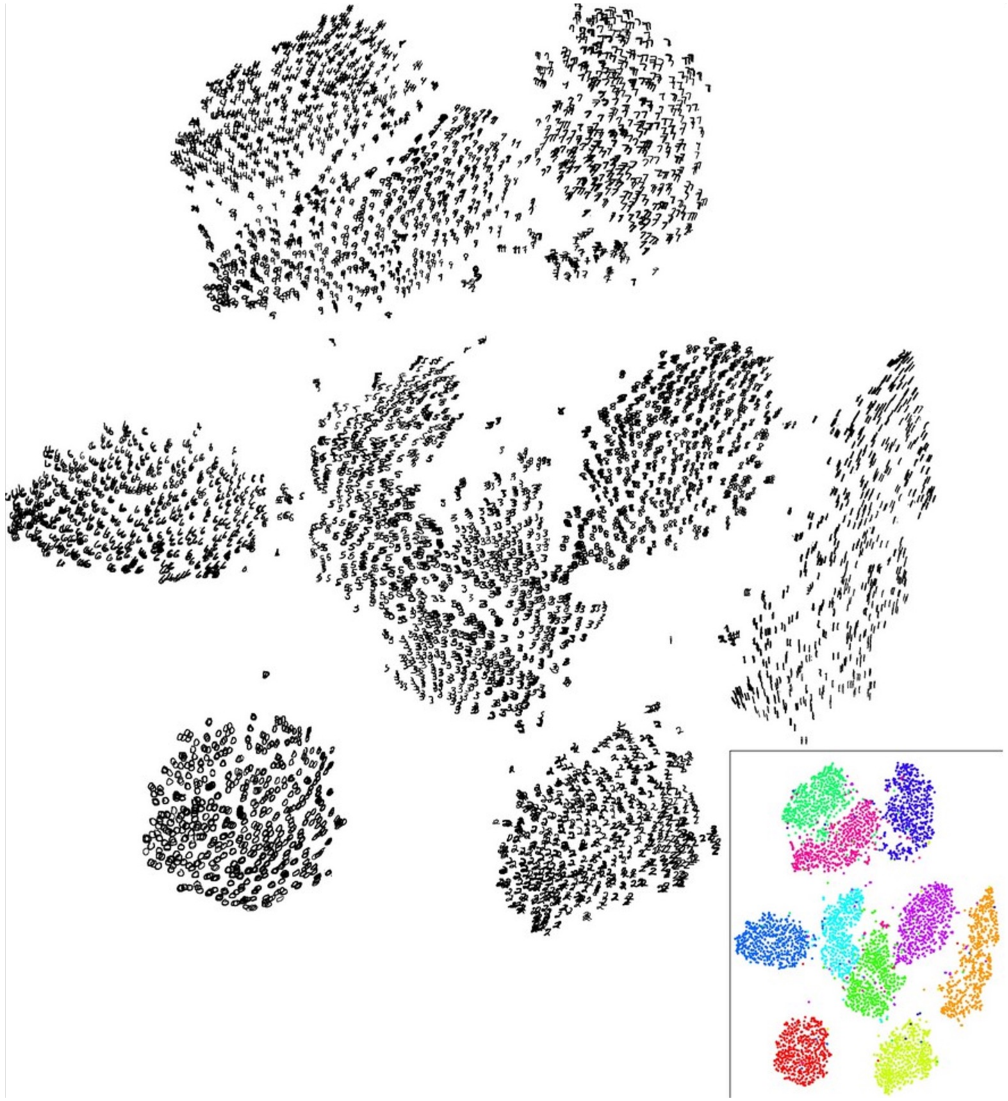
Krizhevsky et al, "ImageNet Classification with Deep Convolutional Neural Networks", NeurIPS 2012.
Figures reproduced with permission.

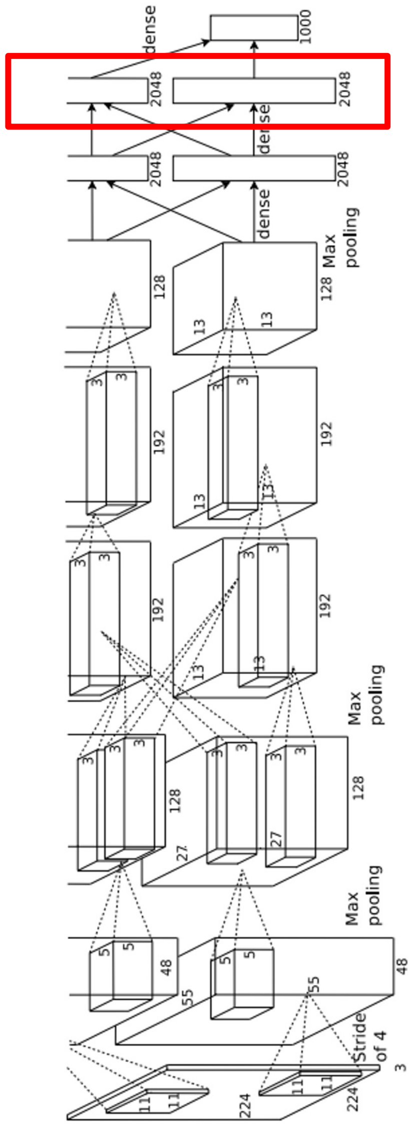
Last Layer: Dimensionality Reduction

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

Simple algorithm: Principal Component Analysis (PCA)

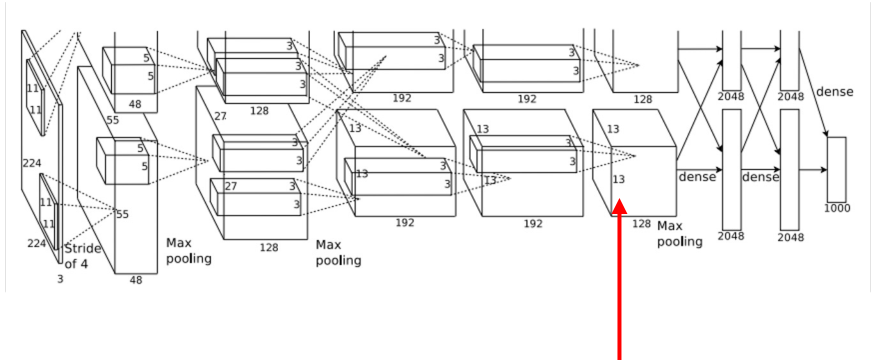
More complex: **t-SNE**





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

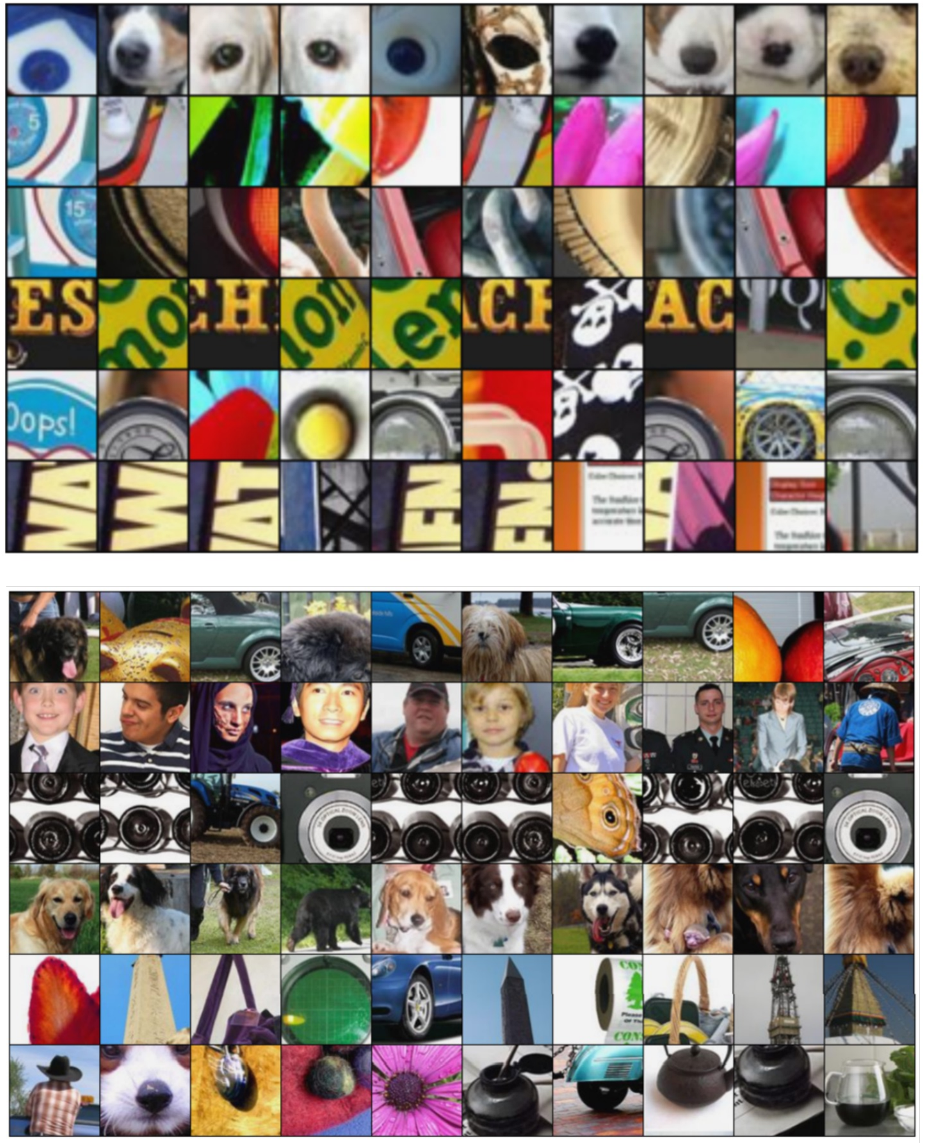
Units in Intermediate Layers: Maximally Activating Patches



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

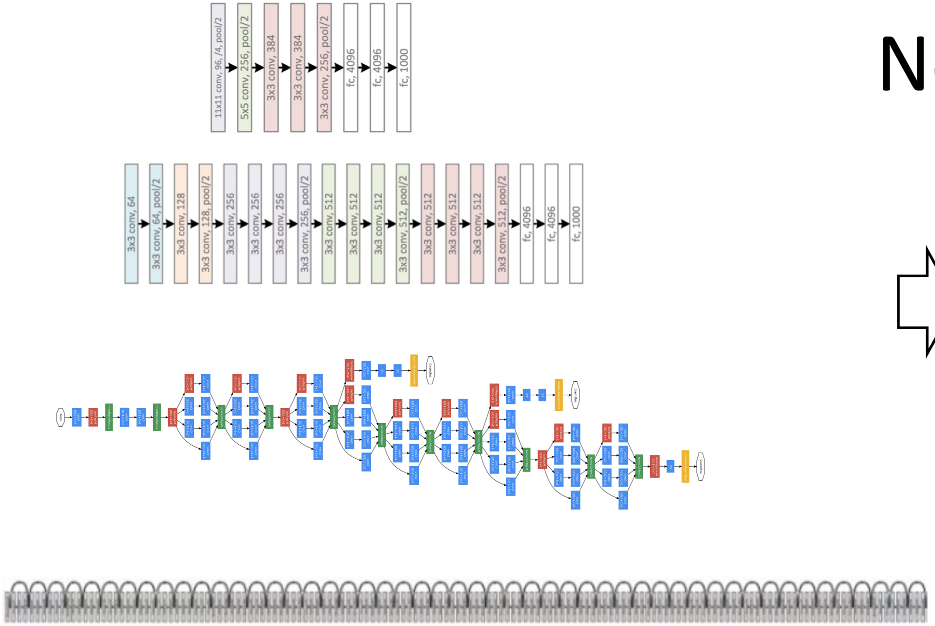
Run many images through the network, record values of chosen channel

Visualize image patches that correspond to maximal activations



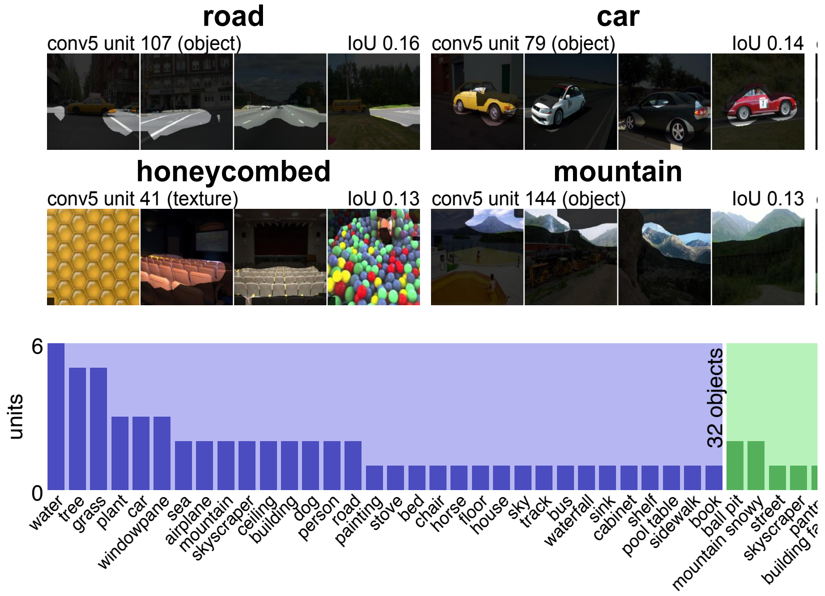
Springenberg et al, "Striving for Simplicity: The All Convolutional Net", ICLR Workshop 2015

Quantify the Interpretability of Networks



Network Dissection

Interpretable Units



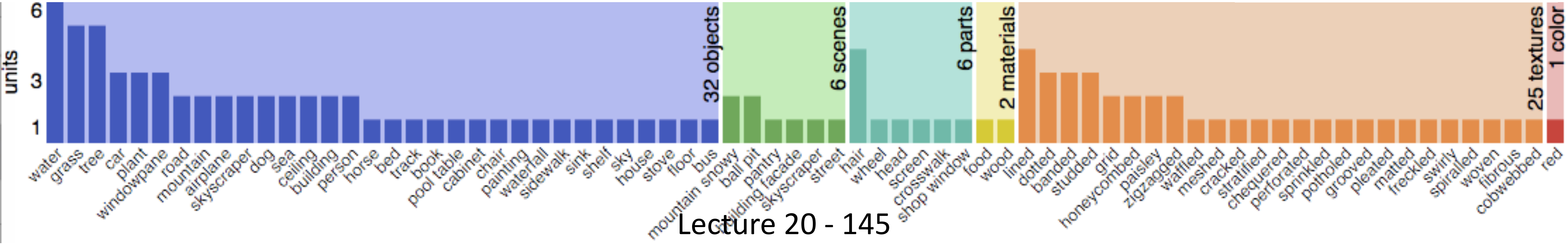
Bau*, Zhou*, Khosla, Oliva, Torralba. CVPR 2017

Layer5 unit 79 car (object) IoU=0.13

Layer5 unit 107 road (object) IoU=0.15

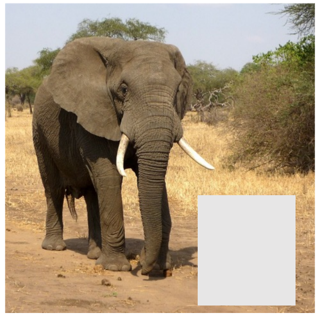
places
THE SCENE RECOGNITION DATABASE

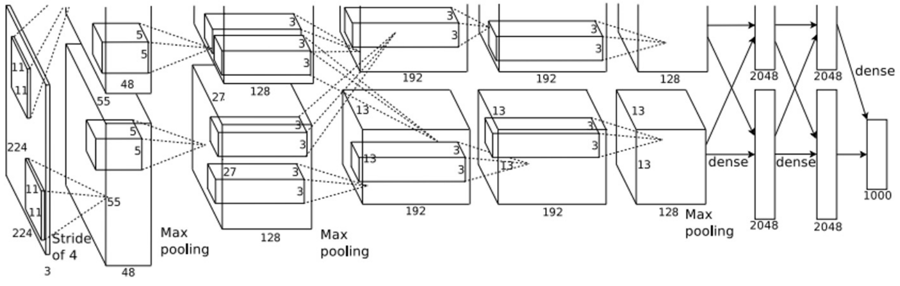
118/256 units covering 72 unique concepts

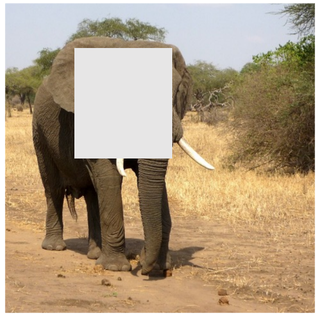


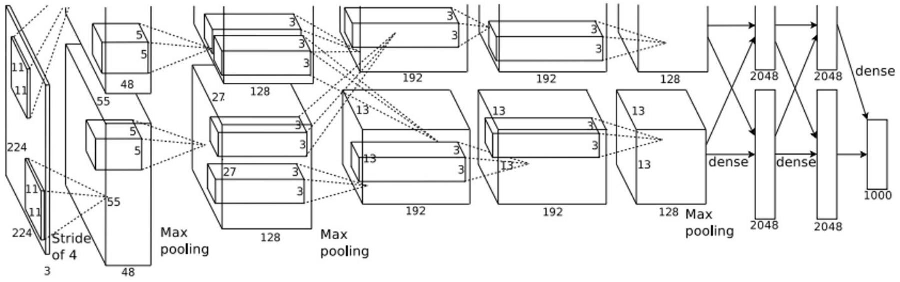
Which Pixels Matter? Saliency via Occlusion

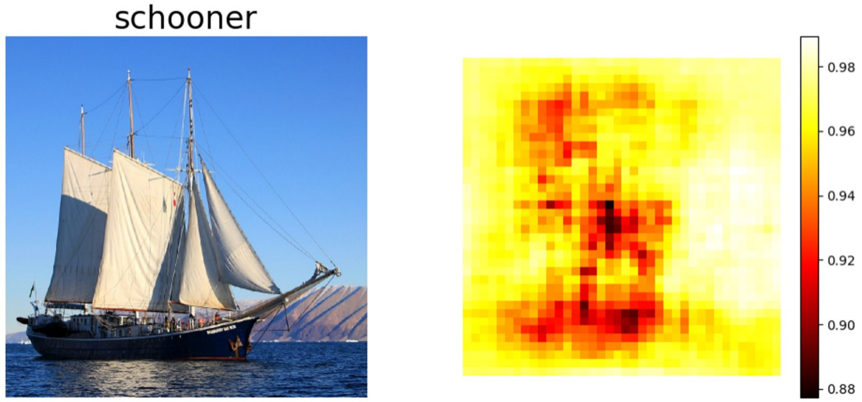
Mask part of the image before feeding to CNN, check how much predicted probabilities change

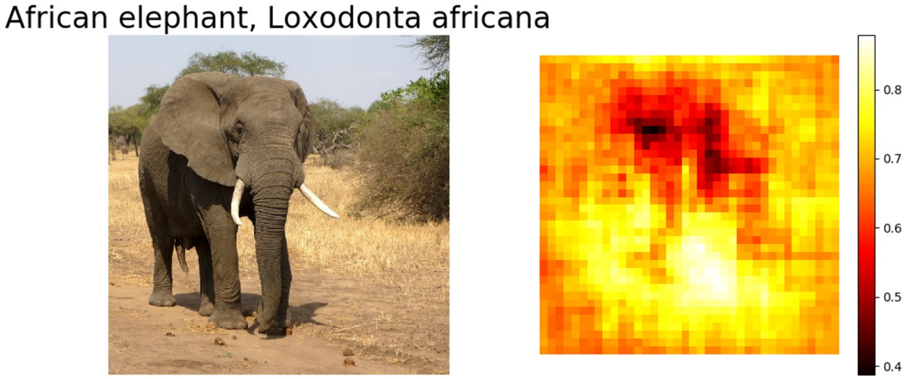


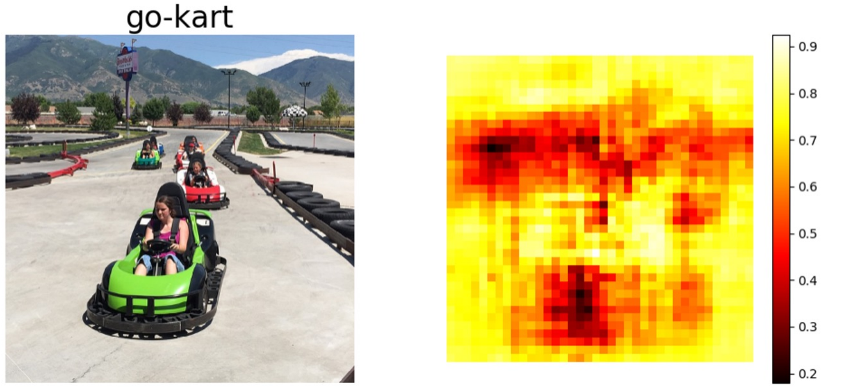






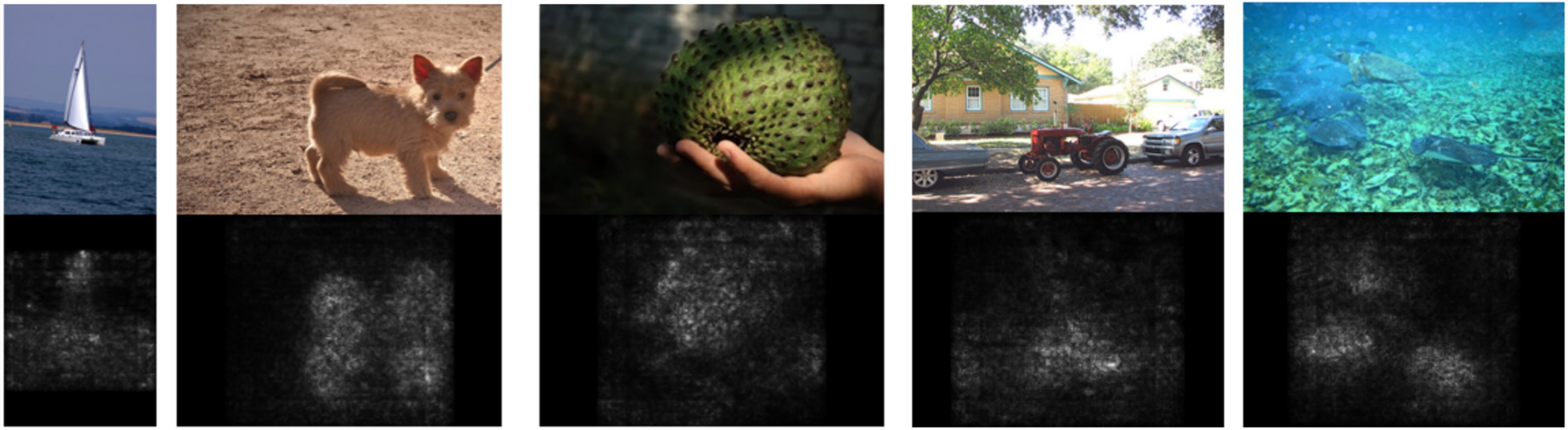






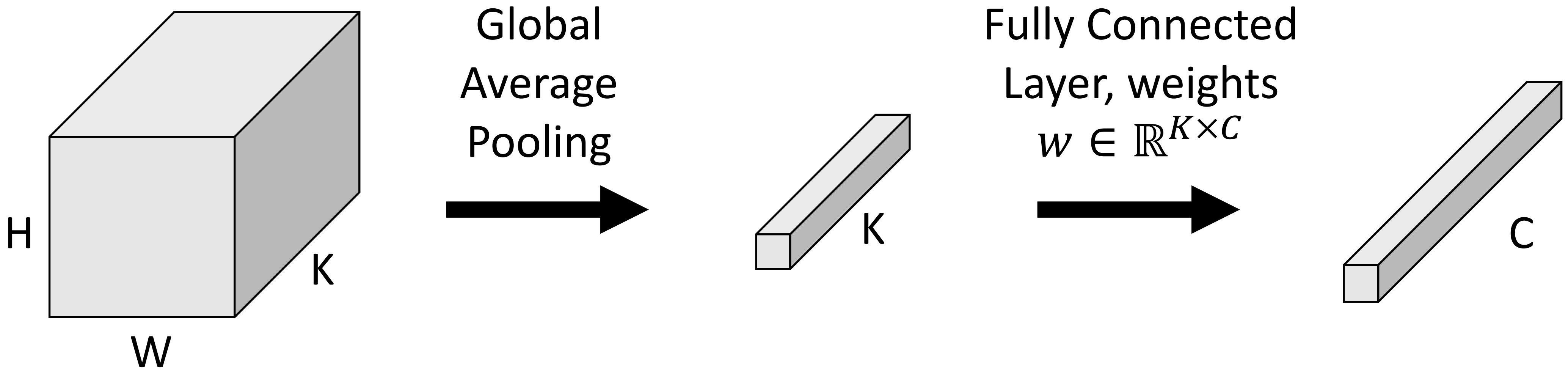
Zeiler and Fergus, "Visualizing and Understanding Convolutional Networks", ECCV 2014

Which pixels matter? Saliency via Backprop



Simonyan, Vedaldi, and Zisserman, “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps”, ICLR Workshop 2014.

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

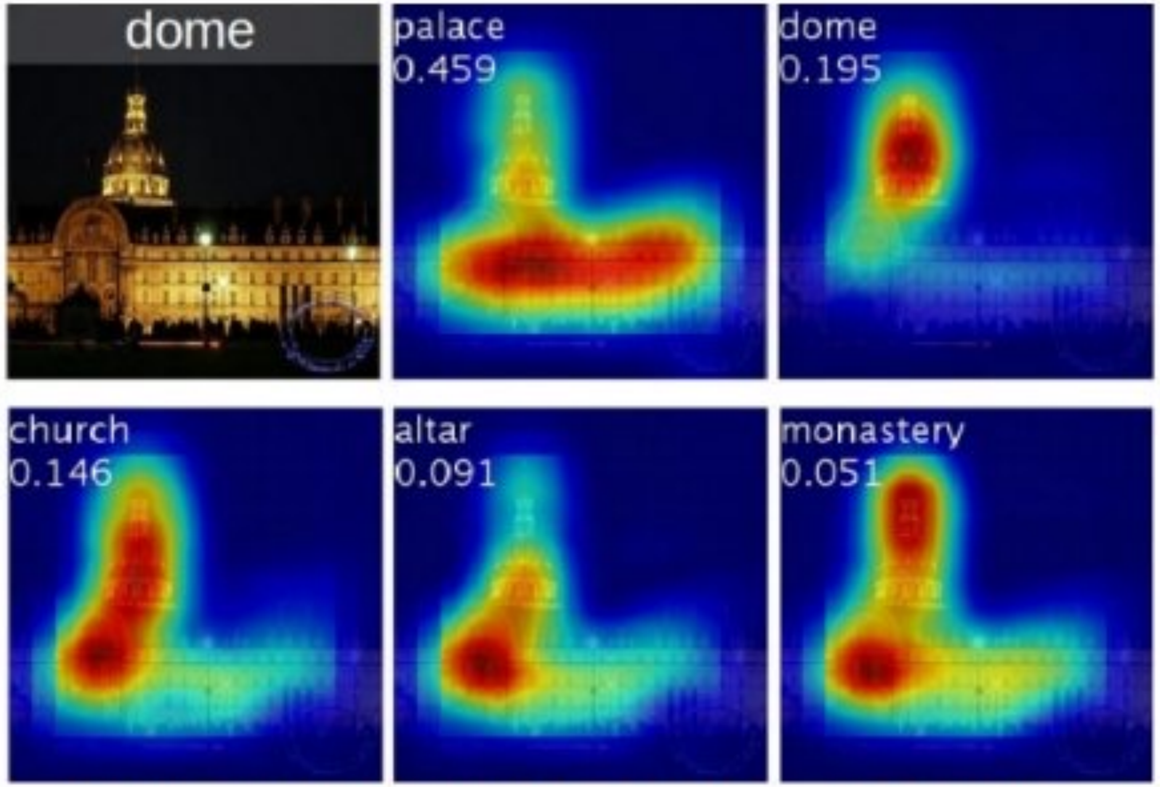
$$F_k = \frac{1}{HW} \sum_{h,w} f_{h,w,k} \qquad S_c = \sum_k w_{k,c} F_k = \frac{1}{HW} \sum_k w_{k,c} \sum_{h,w} f_{h,w,k}$$
$$= \frac{1}{HW} \sum_{h,w} \sum_k w_{k,c} f_{h,w,k}$$

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)



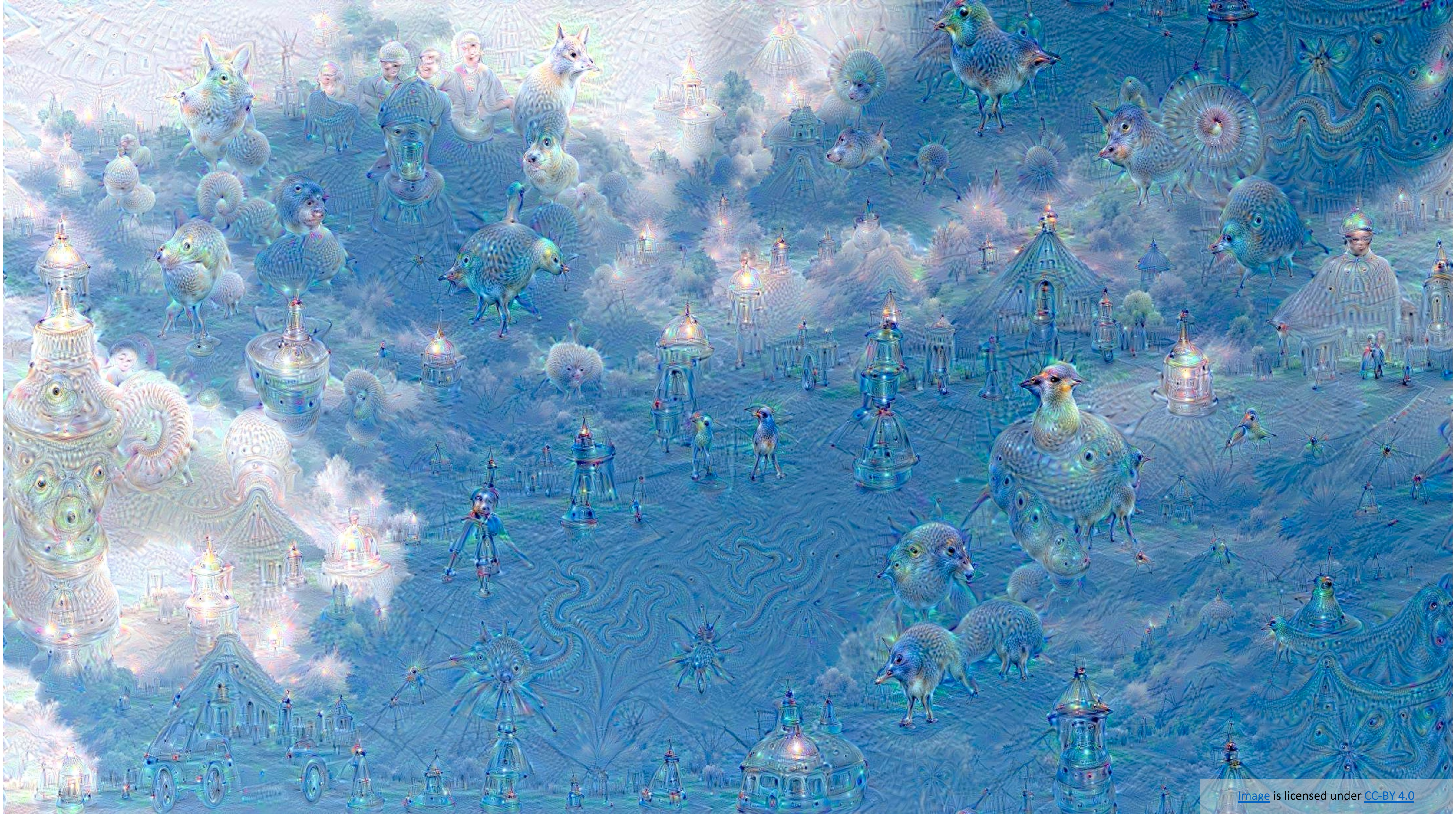
Class activation maps of top 5 predictions

Class activation maps for one object class

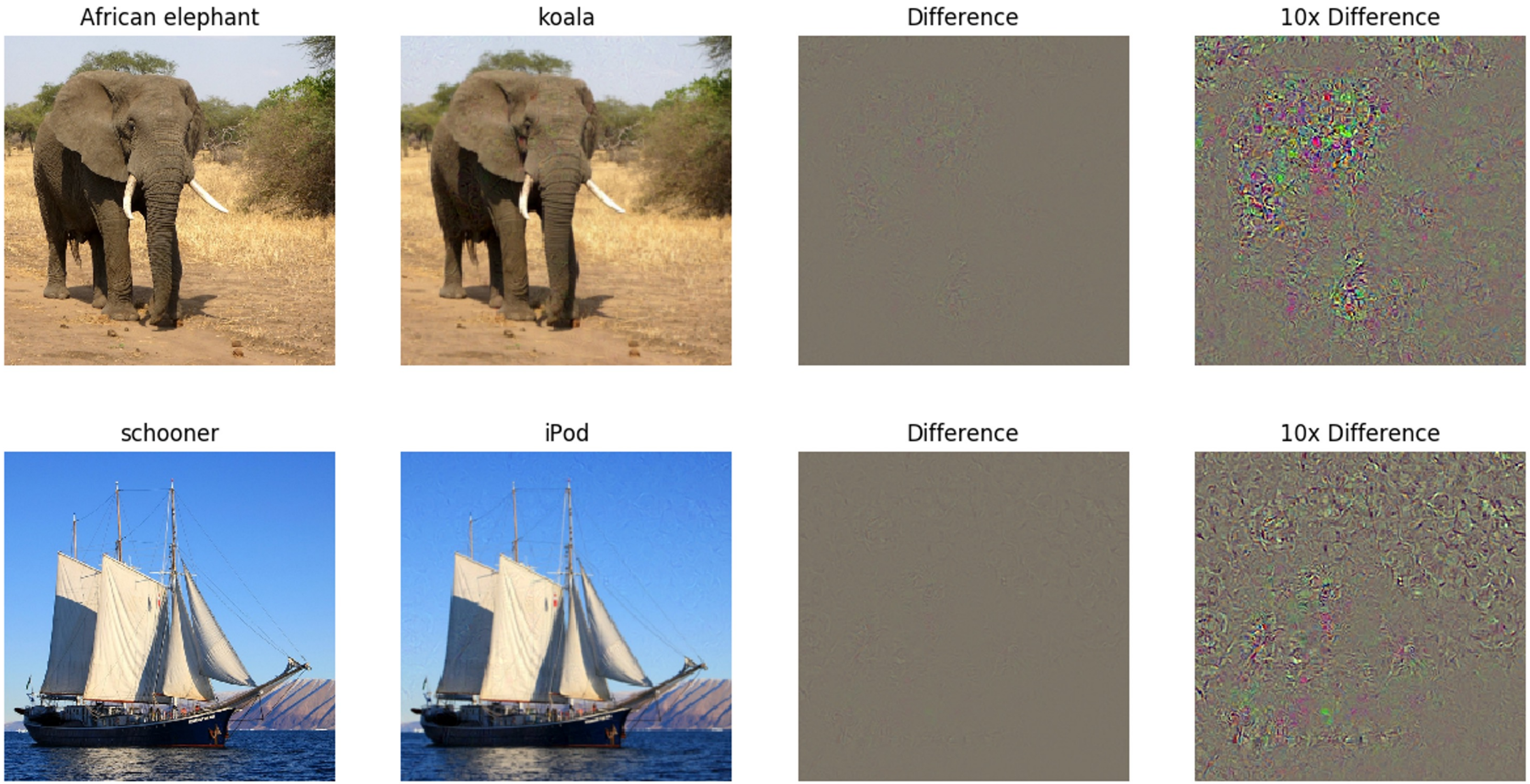
Zhou et al, “Learning Deep Features for Discriminative Localization”, CVPR 2016

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.

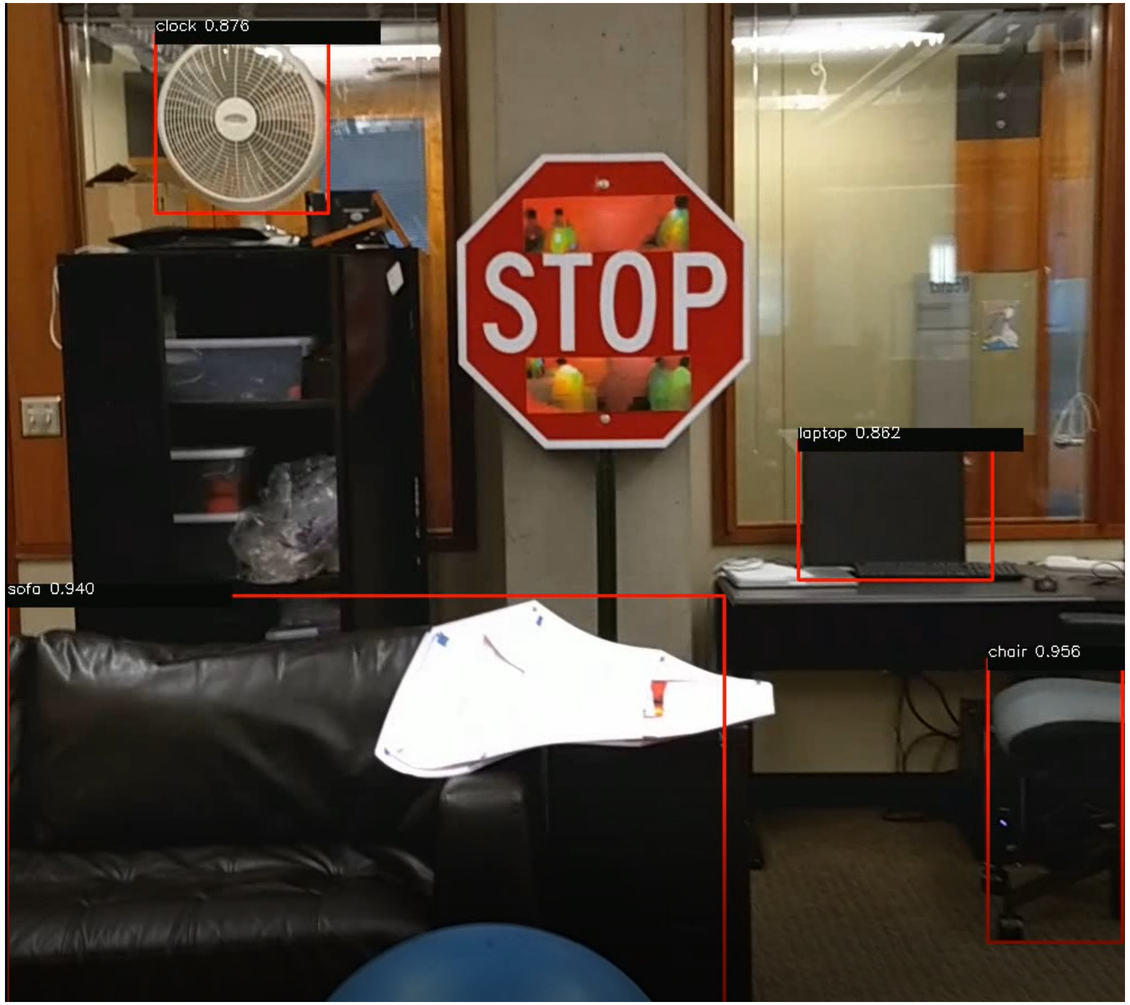


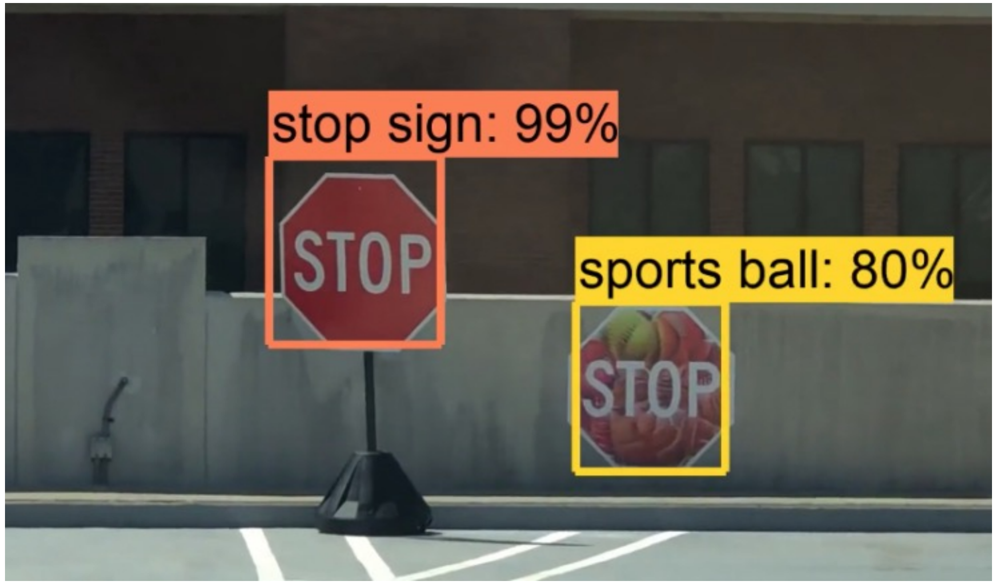
Adversarial Examples



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 $x_1, \ldots x_T$

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

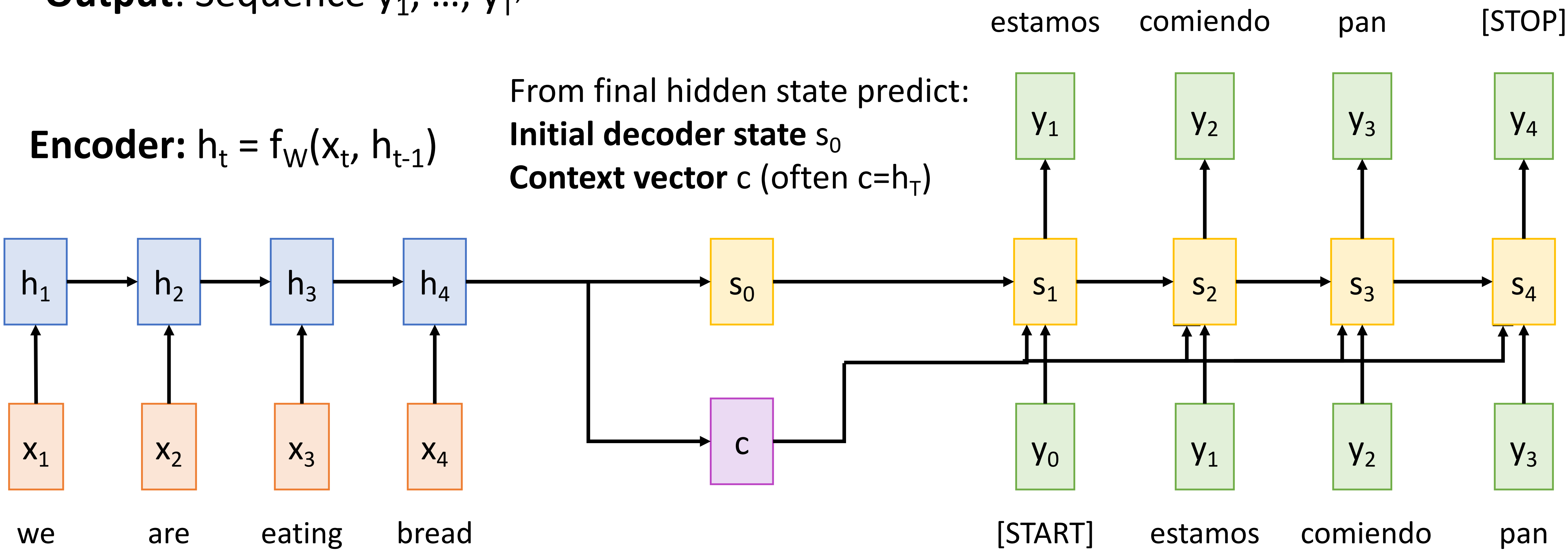
Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$

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

From final hidden state predict:

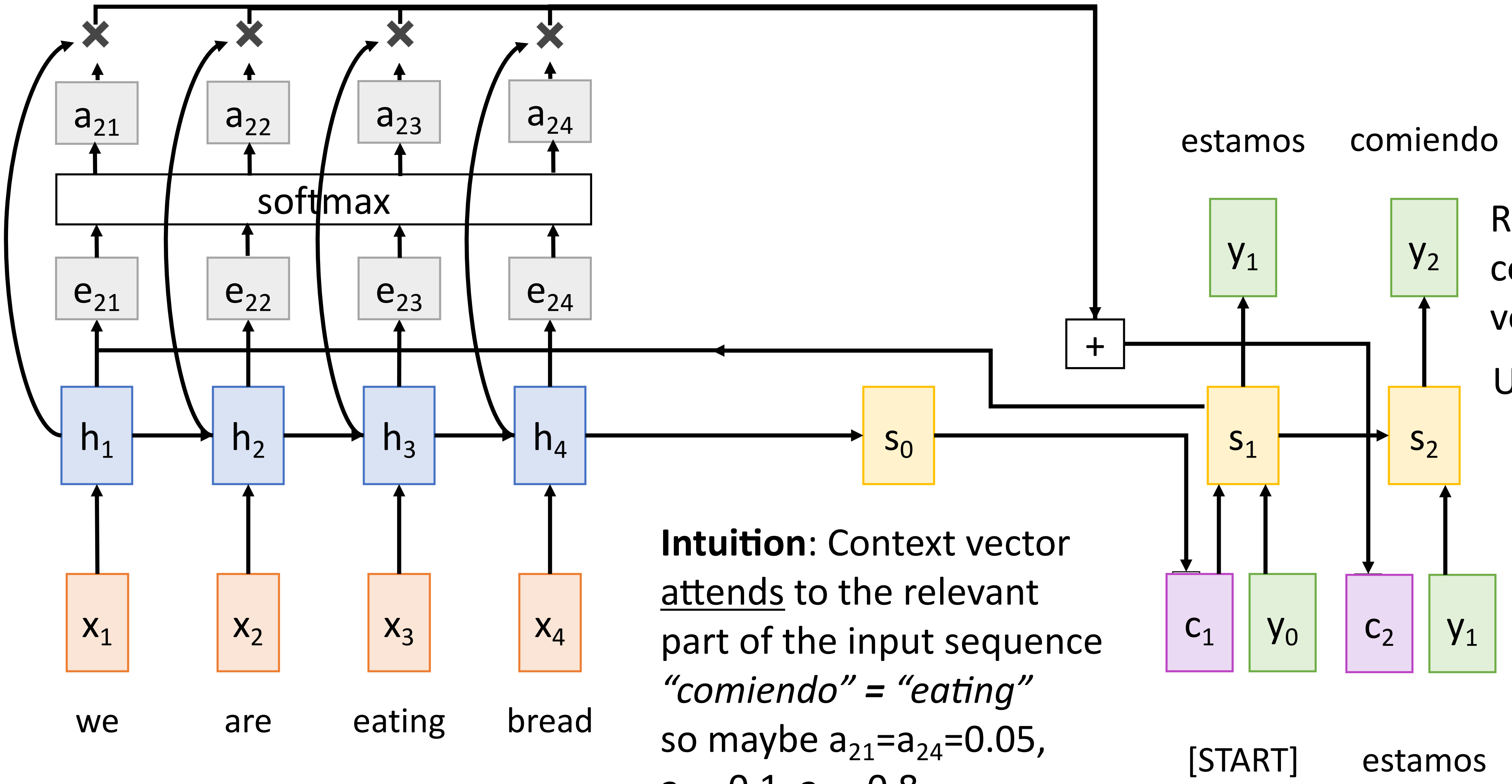
Initial decoder state s_0

Context vector c (often $c=h_T$)



Sutskever et al, "Sequence to sequence learning with neural networks", NeurIPS 2014

Sequence-to-Sequence with RNNs **and Attention**



Repeat: Use s_1 to compute new context vector c_2

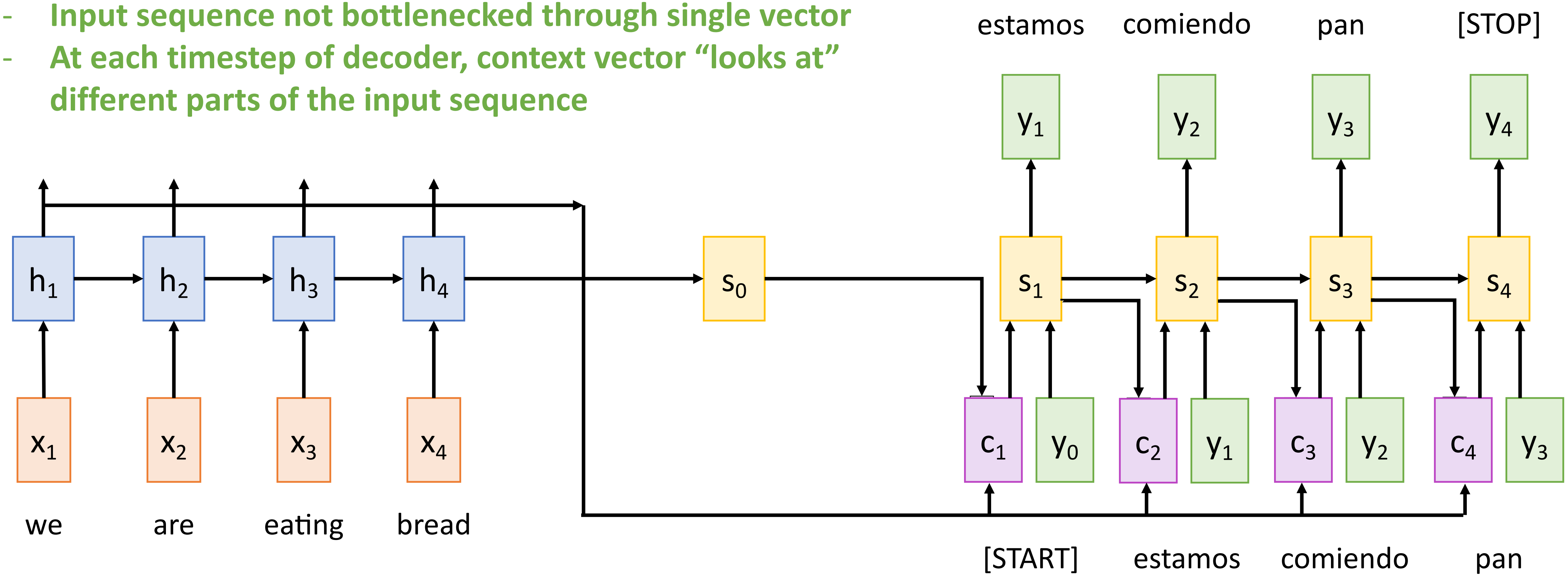
Use c_2 to compute s_2, y_2

Intuition: Context vector attends to the relevant part of the input sequence
"comiendo" = "eating"
so maybe a_{21}=a_{24}=0.05, a_{22}=0.1, a_{23}=0.8

Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

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**

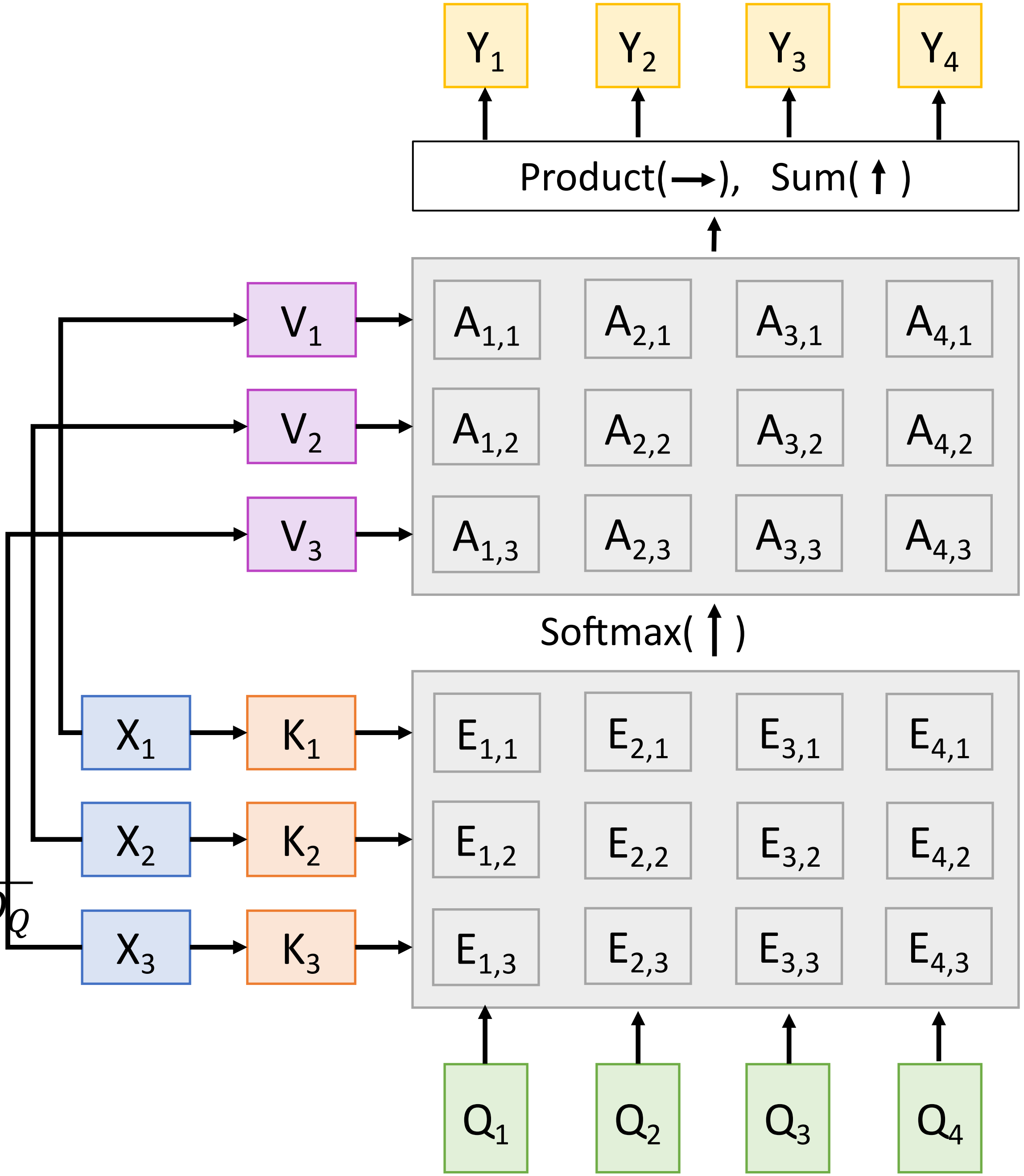


Bahdanau et al, “Neural machine translation by jointly learning to align and translate”, ICLR 2015

Attention Layer

Inputs:
Query vectors: **Q** (Shape: N_Q x D_Q)
Input vectors: **X** (Shape: N_X x D_X)
Key matrix: $\mathbf{W_K}$ (Shape: D_X x D_Q)
Value matrix: $\mathbf{W_V}$ (Shape: D_X x D_V)

Computation:
Key vectors: $\mathbf{K} = \mathbf{XW_K}$ (Shape: N_X x D_Q)
Value Vectors: $\mathbf{V} = \mathbf{XW_V}$ (Shape: N_X x D_V)
Similarities: $E = \mathbf{QK}^T / \sqrt{D_Q}$ (Shape: N_Q x N_X) $E_{i,j} = (\mathbf{Q_i} \cdot \mathbf{K_j}) / \sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) (Shape: N_Q x N_X)
Output vectors: $Y = A\mathbf{V}$ (Shape: N_Q x D_V) $Y_i = \sum_j A_{i,j}\mathbf{V_j}$

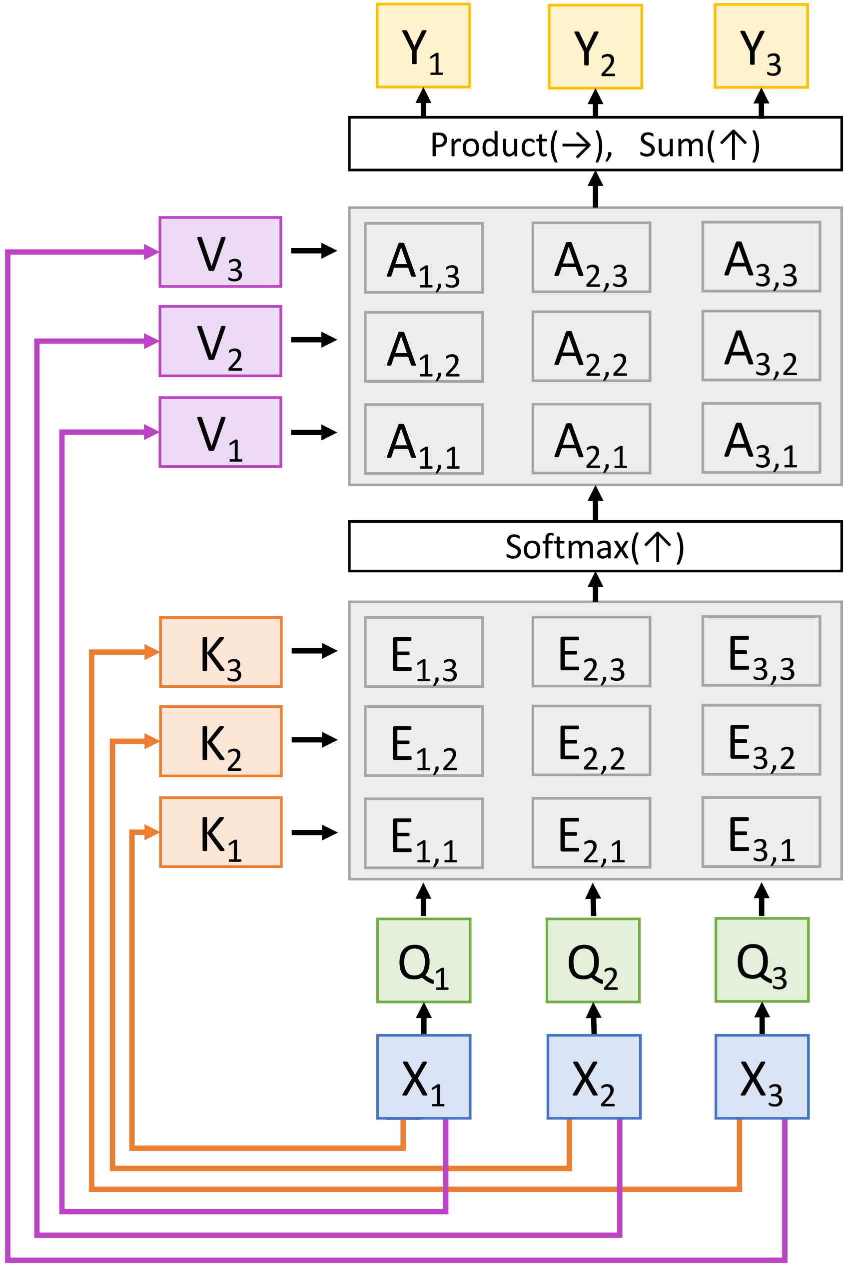


Self-Attention Layer

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

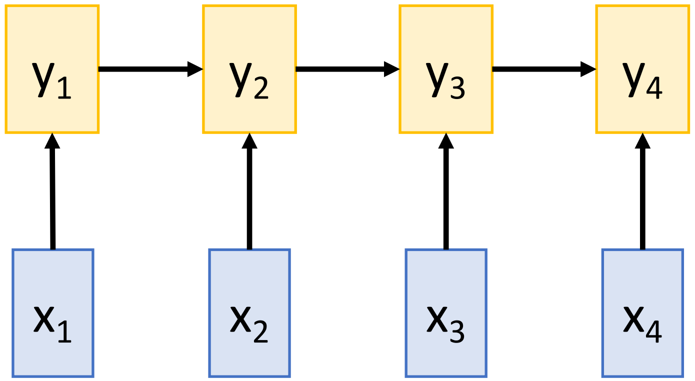
<u>Inputs</u>:
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)
Query matrix: W_Q (Shape: D_X x D_Q)

<u>Computation</u>:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value Vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T / \sqrt{D_Q}$ (Shape: N_X x 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 x N_X)
Output vectors: $Y = AV$ (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$



Three Ways of Processing Sequences

Recurrent Neural Network

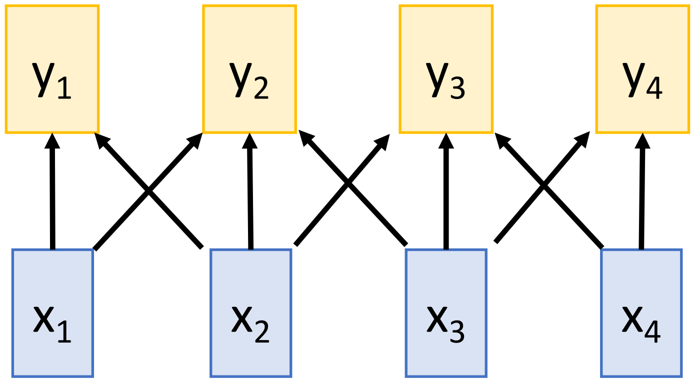


Works on **Ordered Sequences**

(+) Good at long sequences: After one RNN layer, h_T ”sees” the whole sequence

(-) Not parallelizable: need to compute hidden states sequentially

1D Convolution

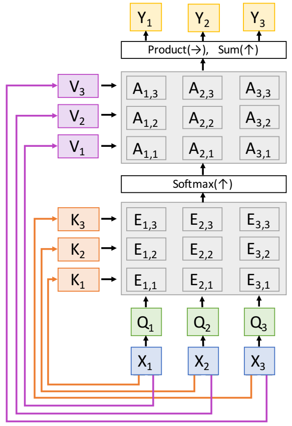


Works on **Multidimensional Grids**

(-) 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

Self-Attention



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

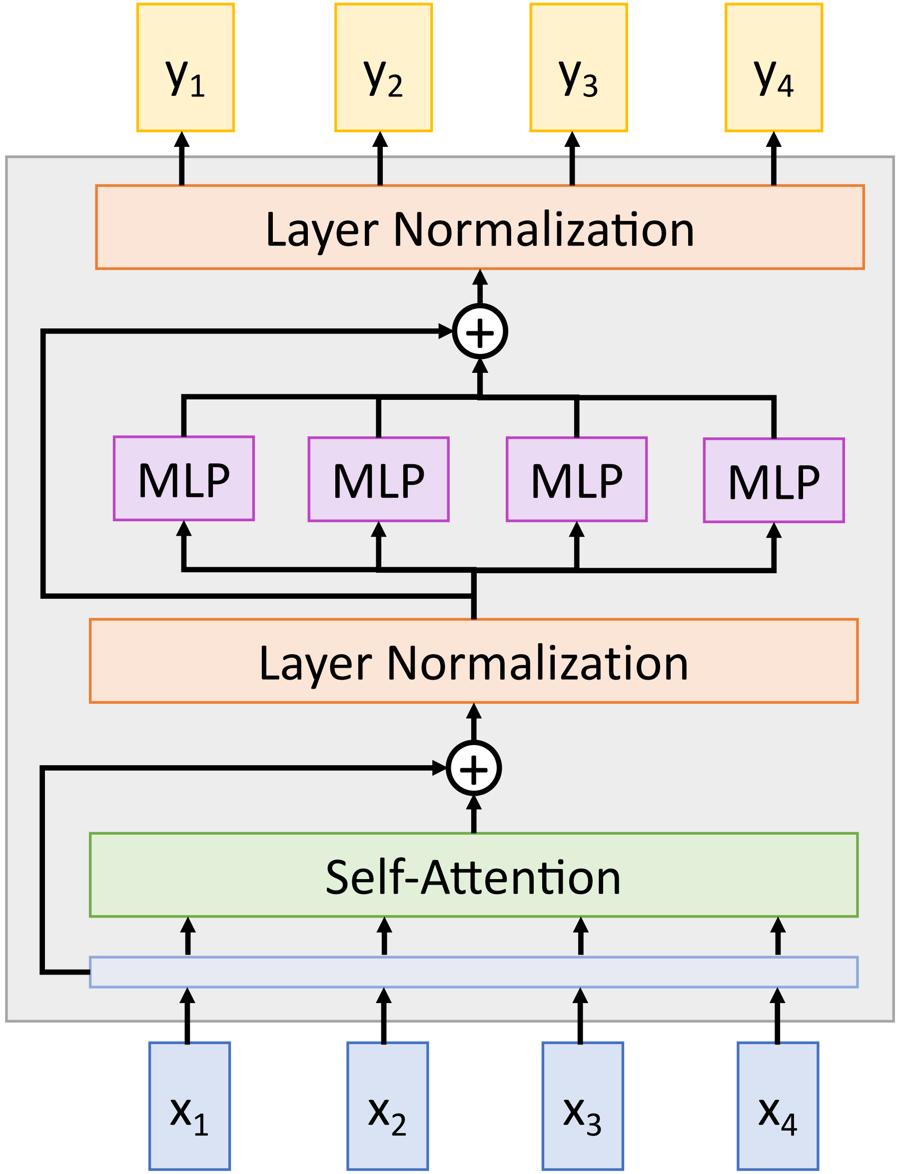
The Transformer

<u>**Transformer Block:**</u>
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



Vaswani et al, "Attention is all you need", NeurIPS 2017

The Transformer

<u>Transformer Block</u>:
Input: Set of vectors x
Output: Set of vectors y

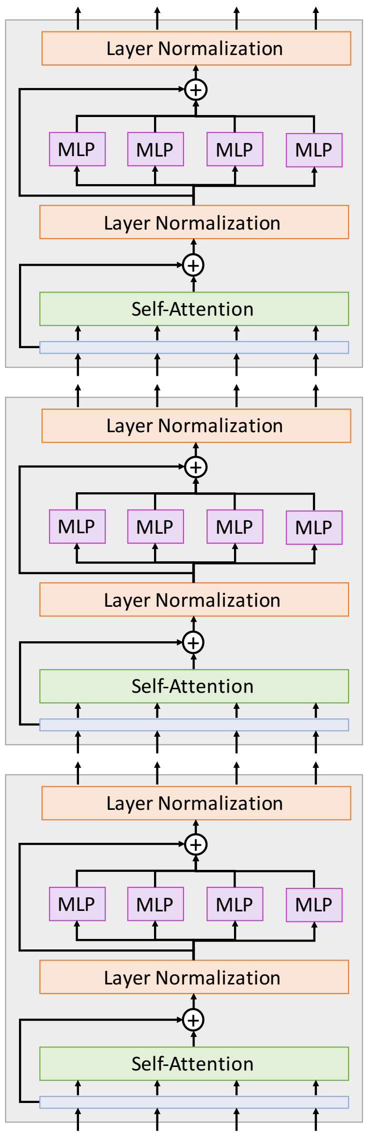
Self-attention is the only interaction between vectors!

Layer norm and MLP work independently per vector

Highly scalable, highly parallelizable

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

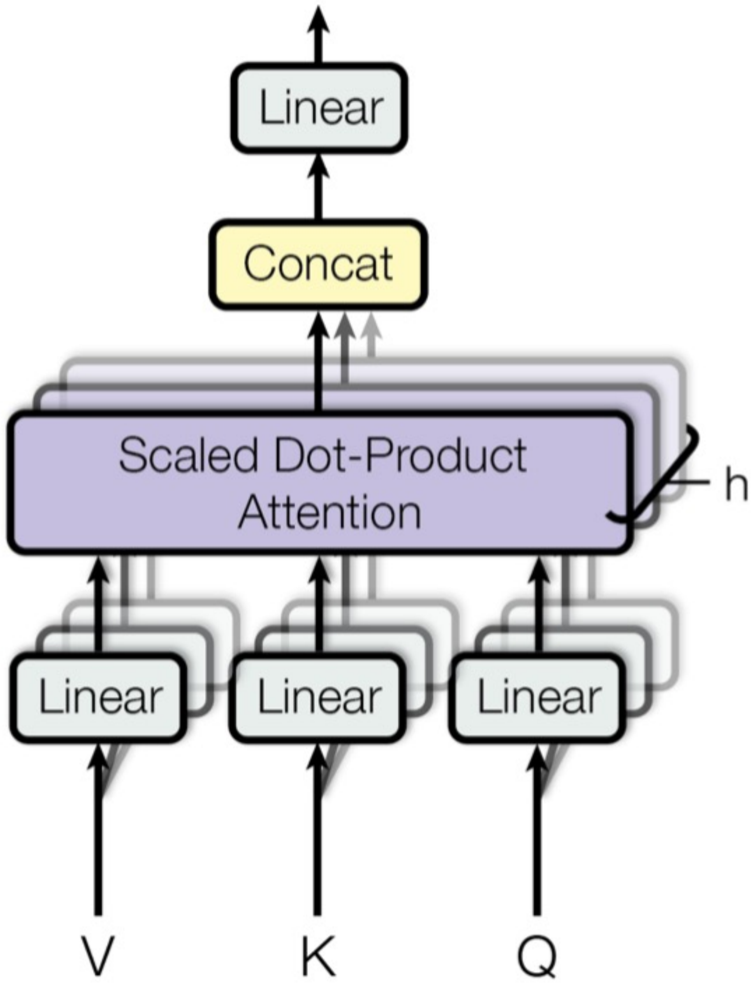
Vaswani et al:
12 blocks, D_Q=512, 6 heads



Vaswani et al, "Attention is all you need", NeurIPS 2017

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.

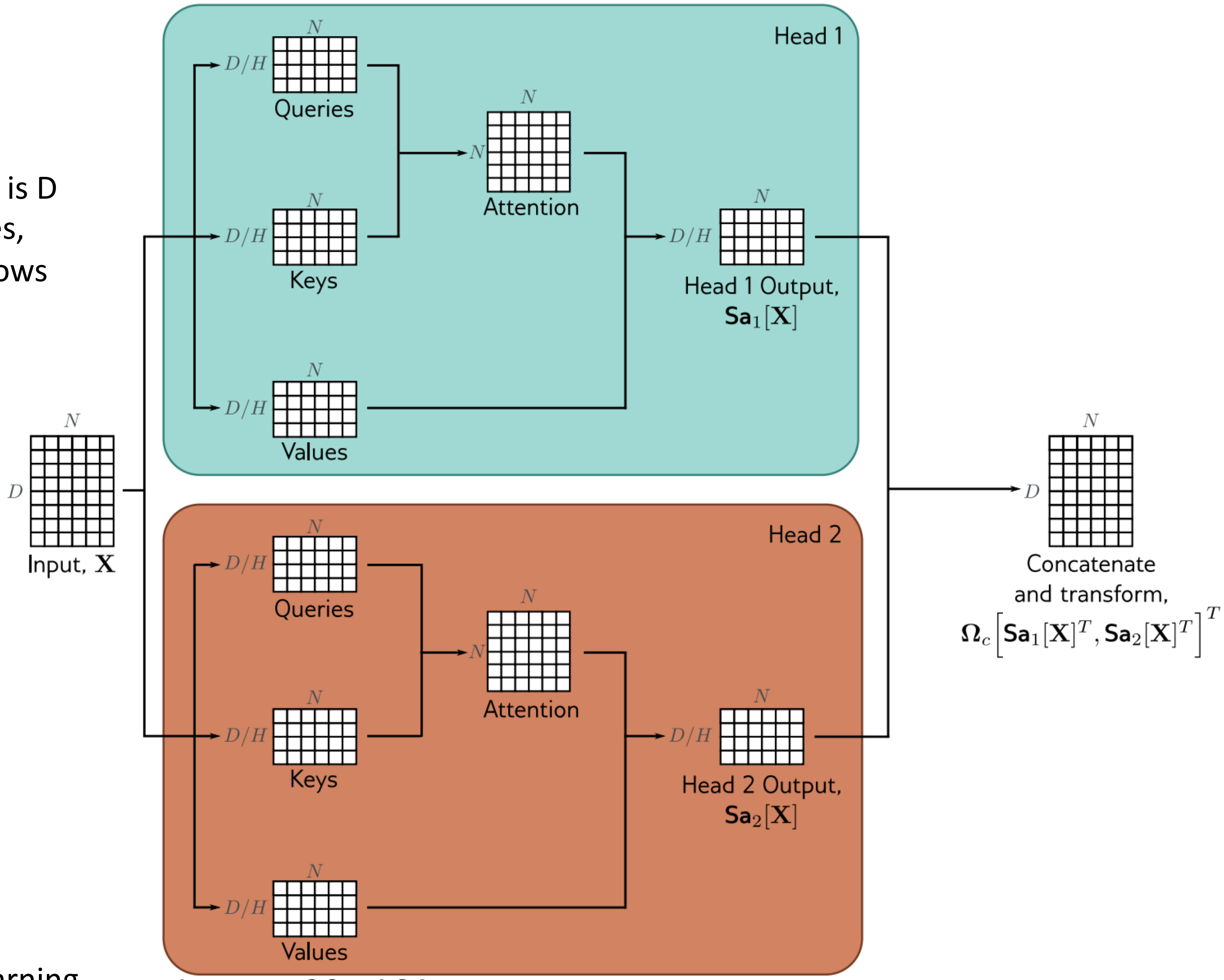
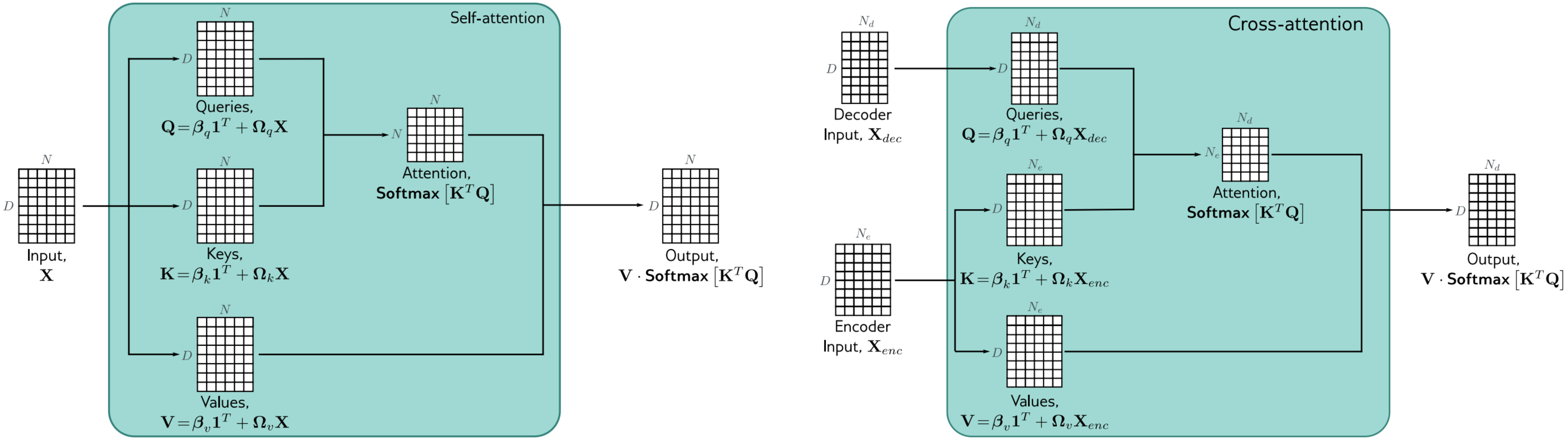


Figure 12.6 from Understanding Deep Learning

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)

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

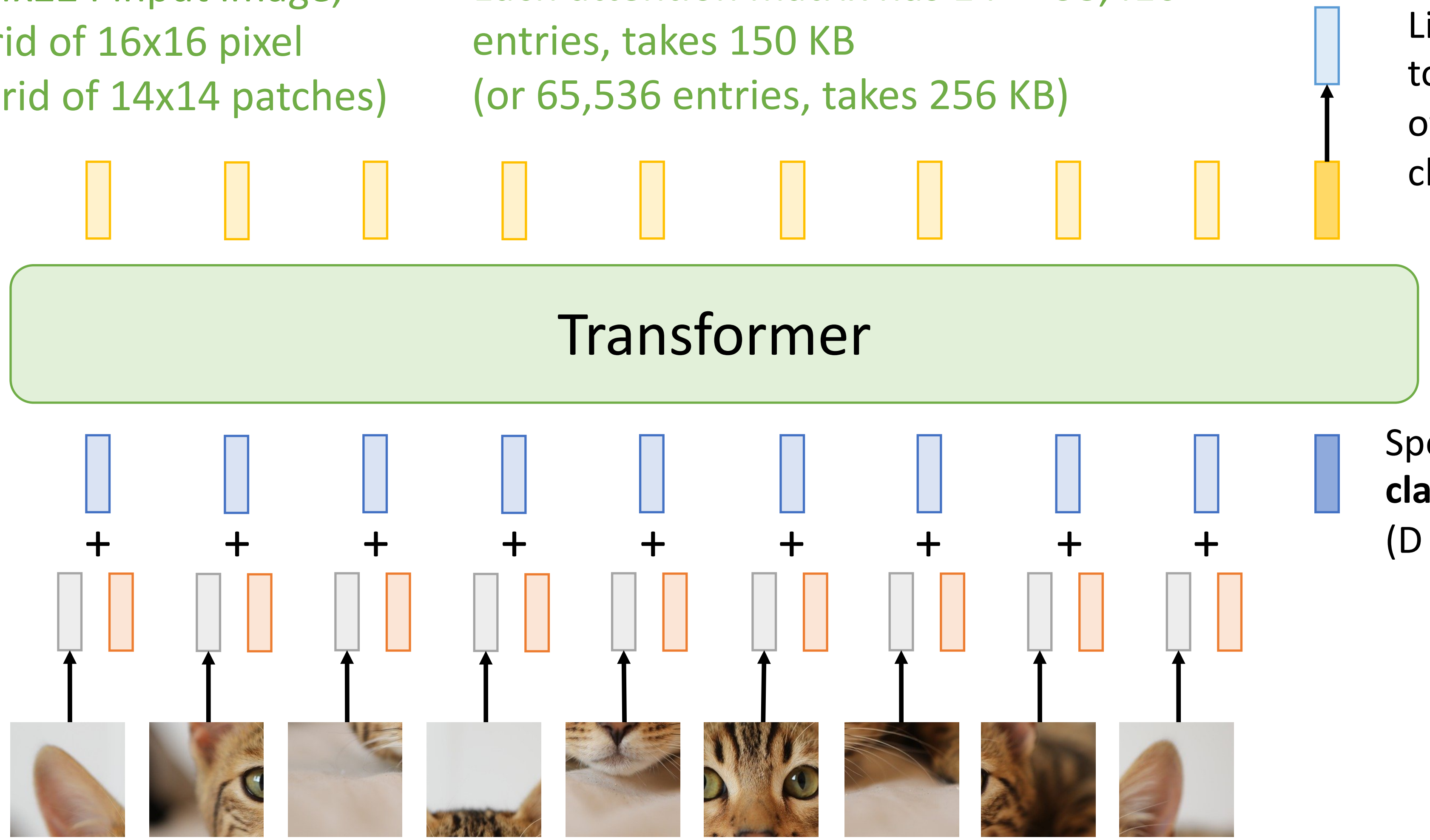
Output vectors

Exact same as NLP Transformer!

Add positional embedding: learned D-dim vector per position

Linear projection to D-dimensional vector

N input patches, each of shape 3x16x16



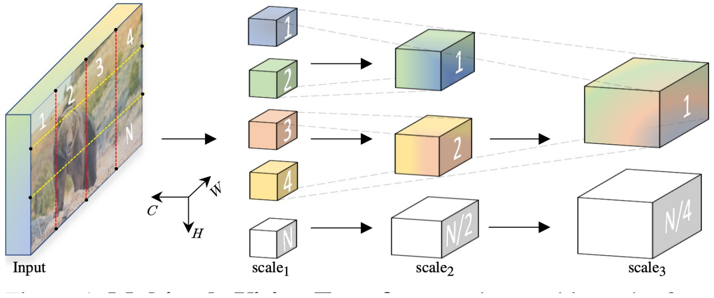
Linear projection to C-dim vector of predicted class scores

Special extra input: **classification token** (D dims, learned)

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

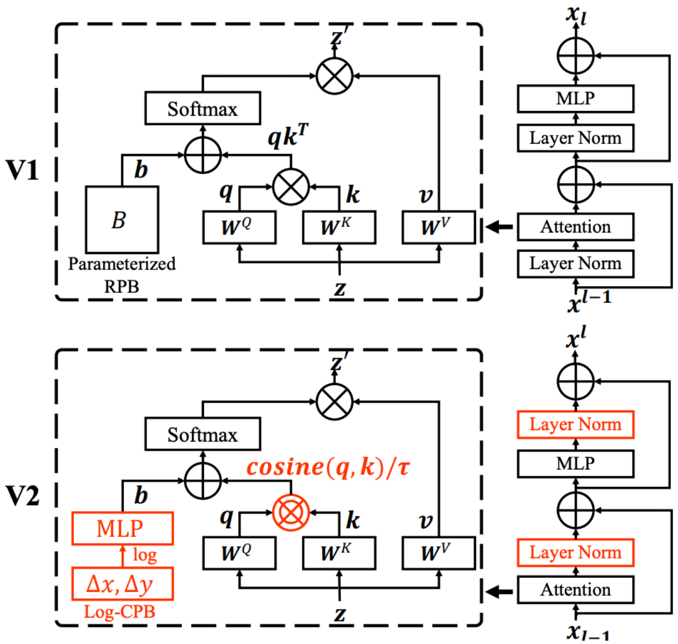
Hierarchical Vision Transformers

MViT



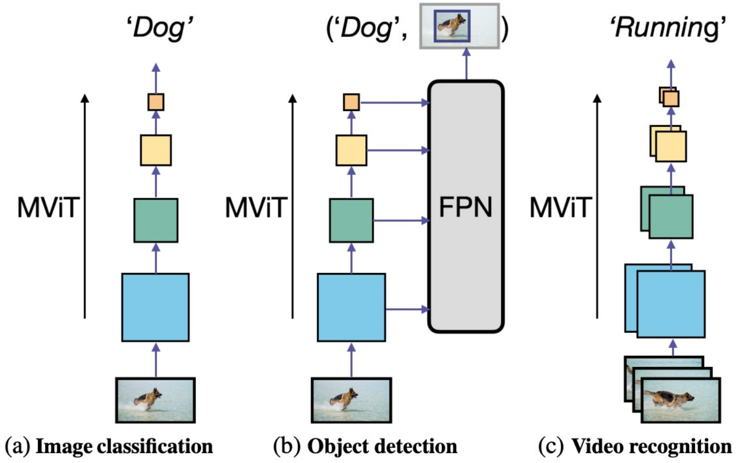
Fan et al, "Multiscale Vision Transformers", ICCV 2021

Swin, Swin-V2



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

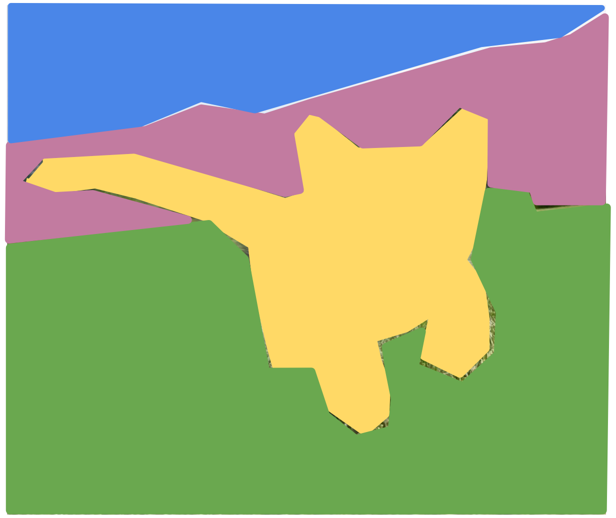
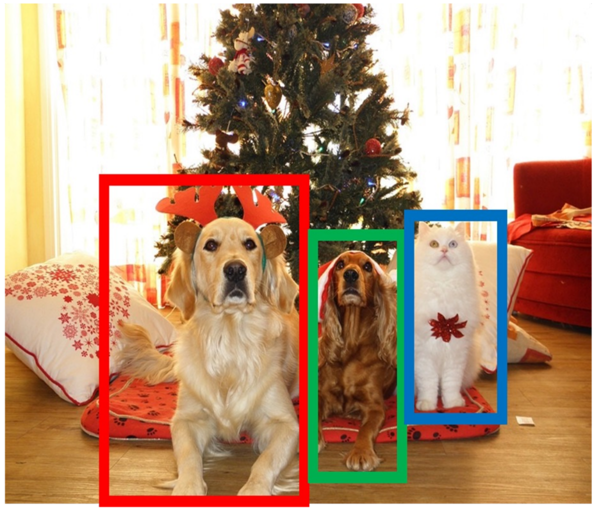
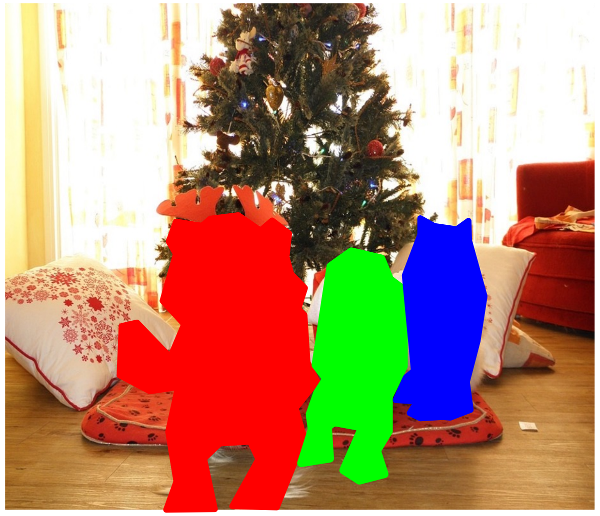
Improved MViT



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	Semantic Segmentation	Object Detection	Instance Segmentation
			
CAT	GRASS, CAT, TREE, SKY	DOG, DOG, CAT	DOG, DOG, CAT

No spatial extent

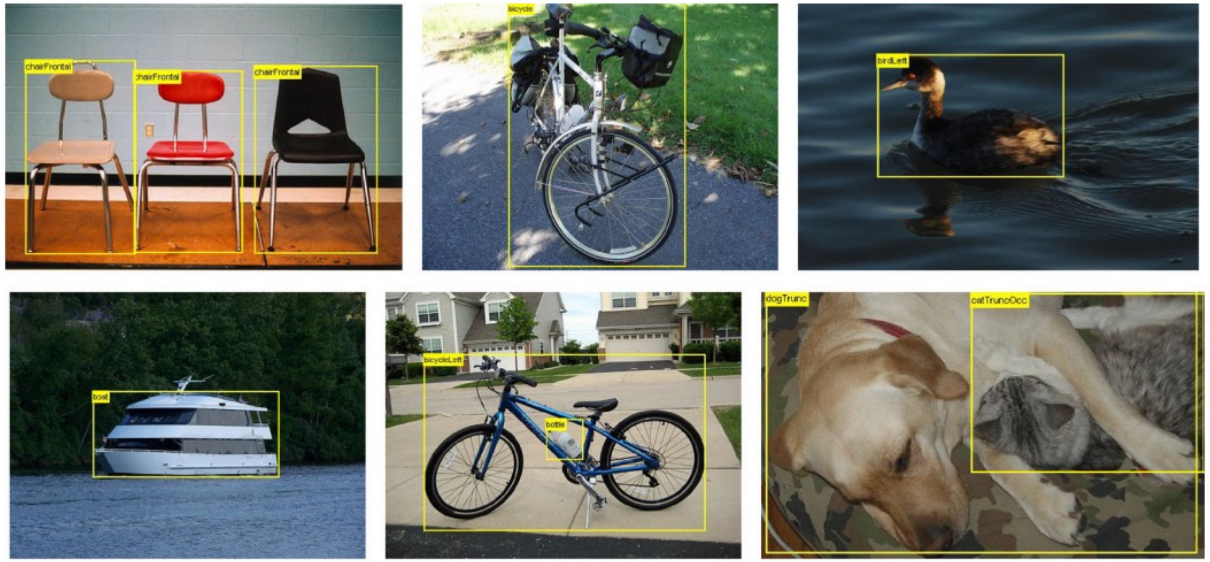
No objects, just pixels

Multiple Objects

Object Detection Datasets

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

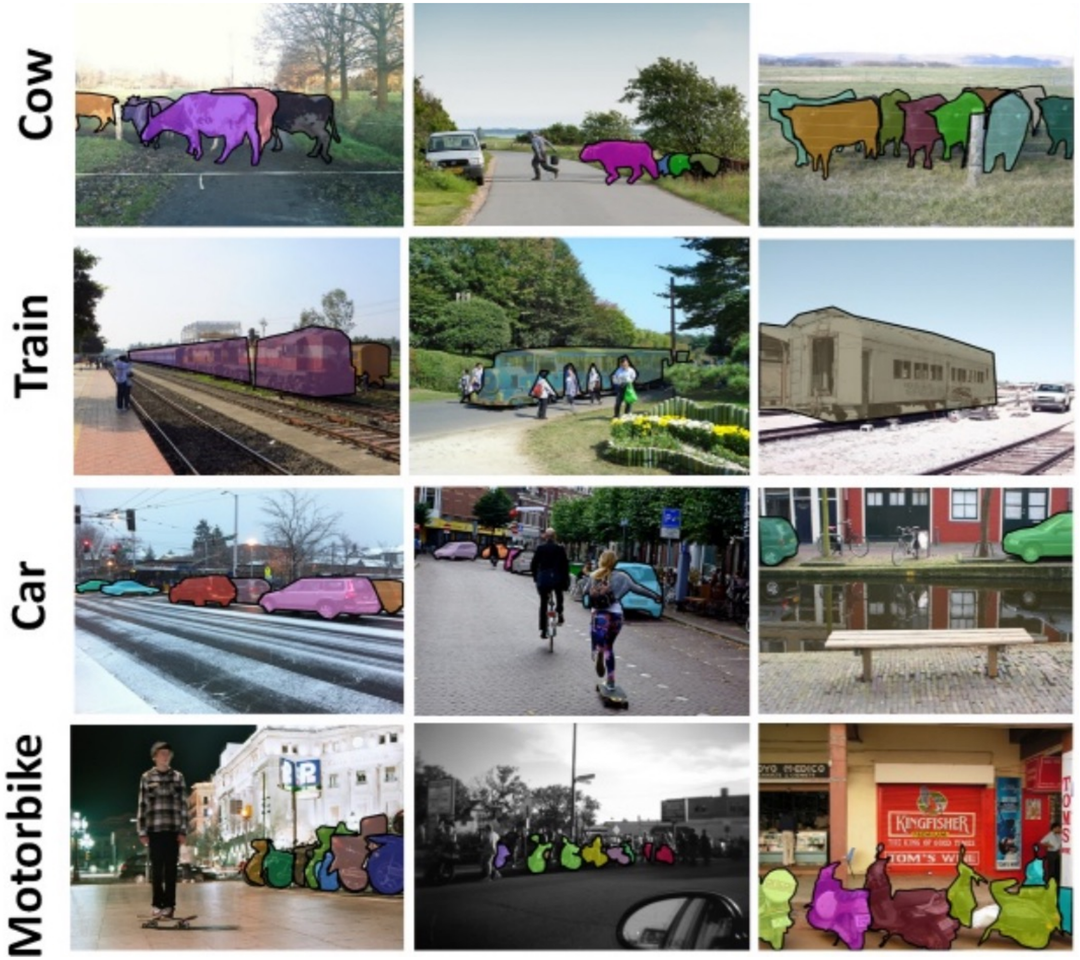
Vehicles	Household	Animals	Other
Aeroplane	Bottle	Bird	Person
Bicycle	Chair	Cat	
Boat	Dining table	Cow	
Bus	Potted plant	Dog	
Car	Sofa	Horse	
Motorbike	TV/Monitor	Sheep	
Train			



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

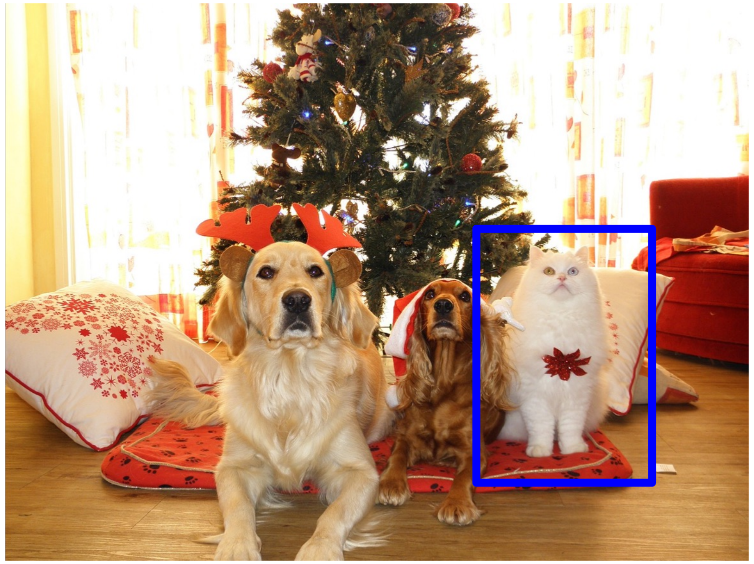
COCO Dataset (80 classes, 120K training images)

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

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

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

Consider a box of size h x w:
Possible x positions: W – w + 1
Possible y positions: H – h + 1
Possible positions:
(W – w + 1) * (H – h + 1)

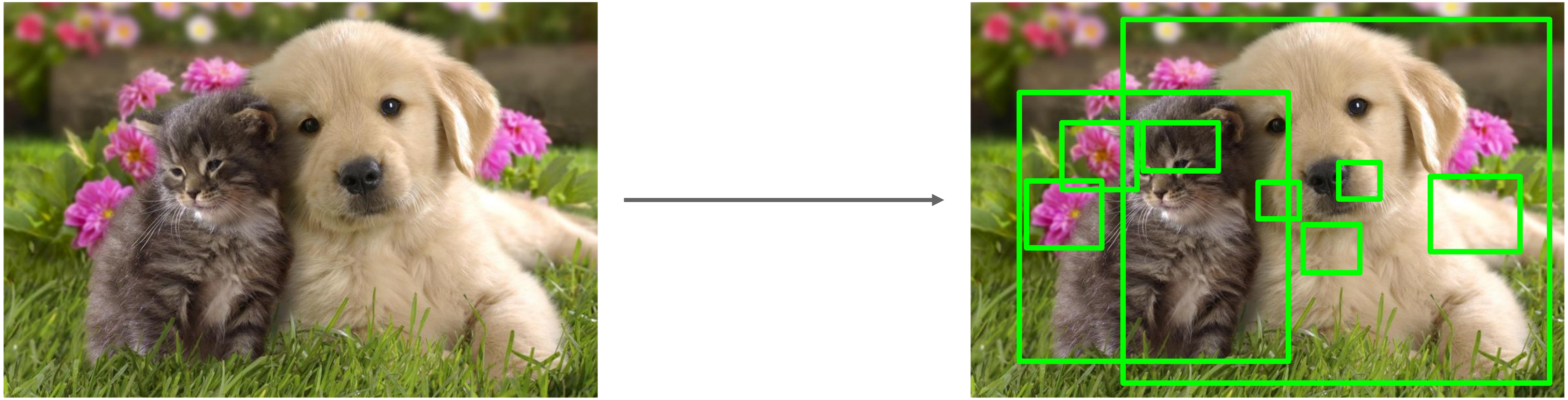
Total possible boxes:

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

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

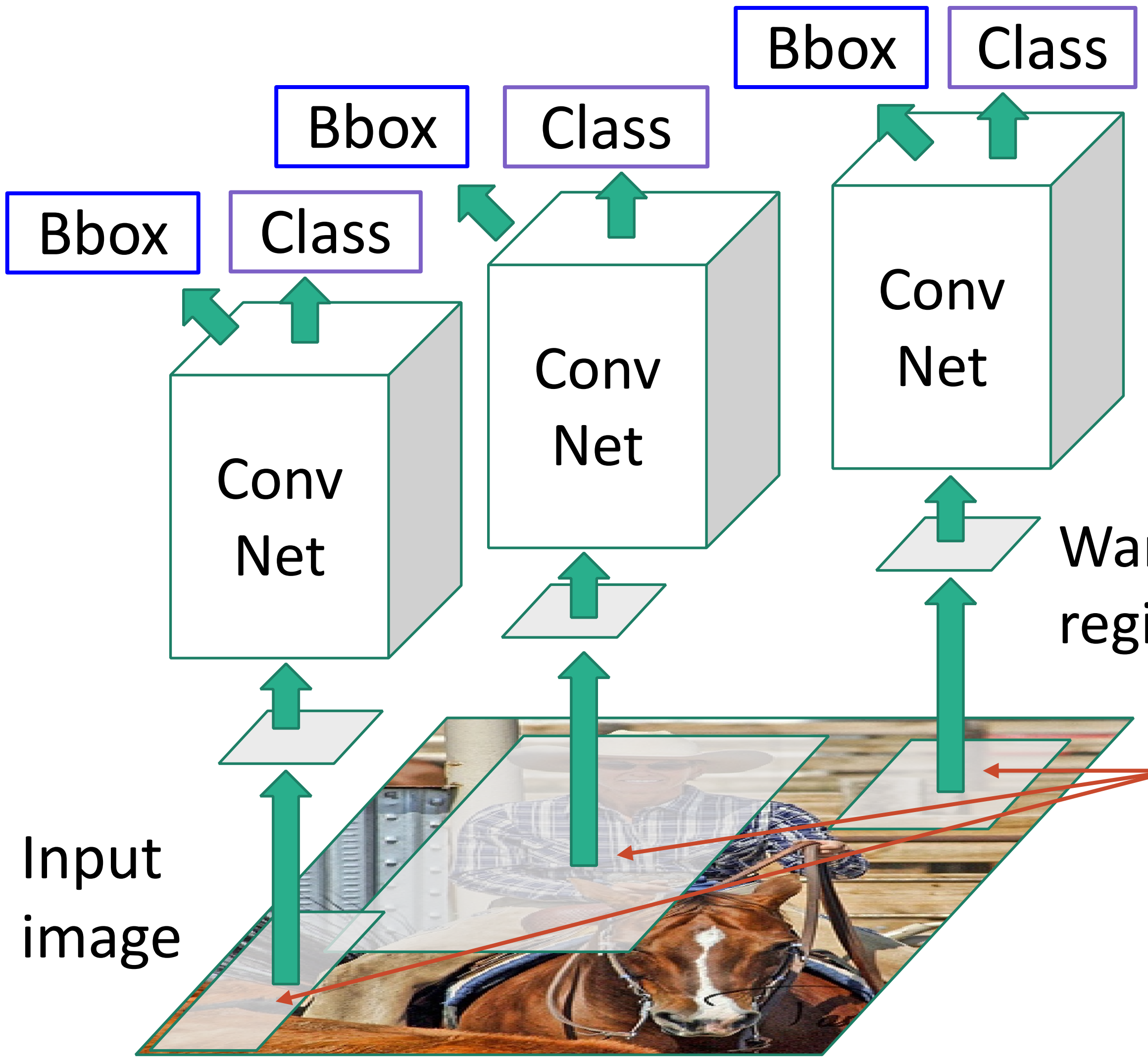
Region Proposals

- 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)

Forward each region through ConvNet

Warped image regions (224x224)

Regions of Interest (RoI) from a proposal method (~2k)

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:
$b_w = p_w \exp(t_w)$ $\quad$ $b_h = p_h \exp(t_h)$

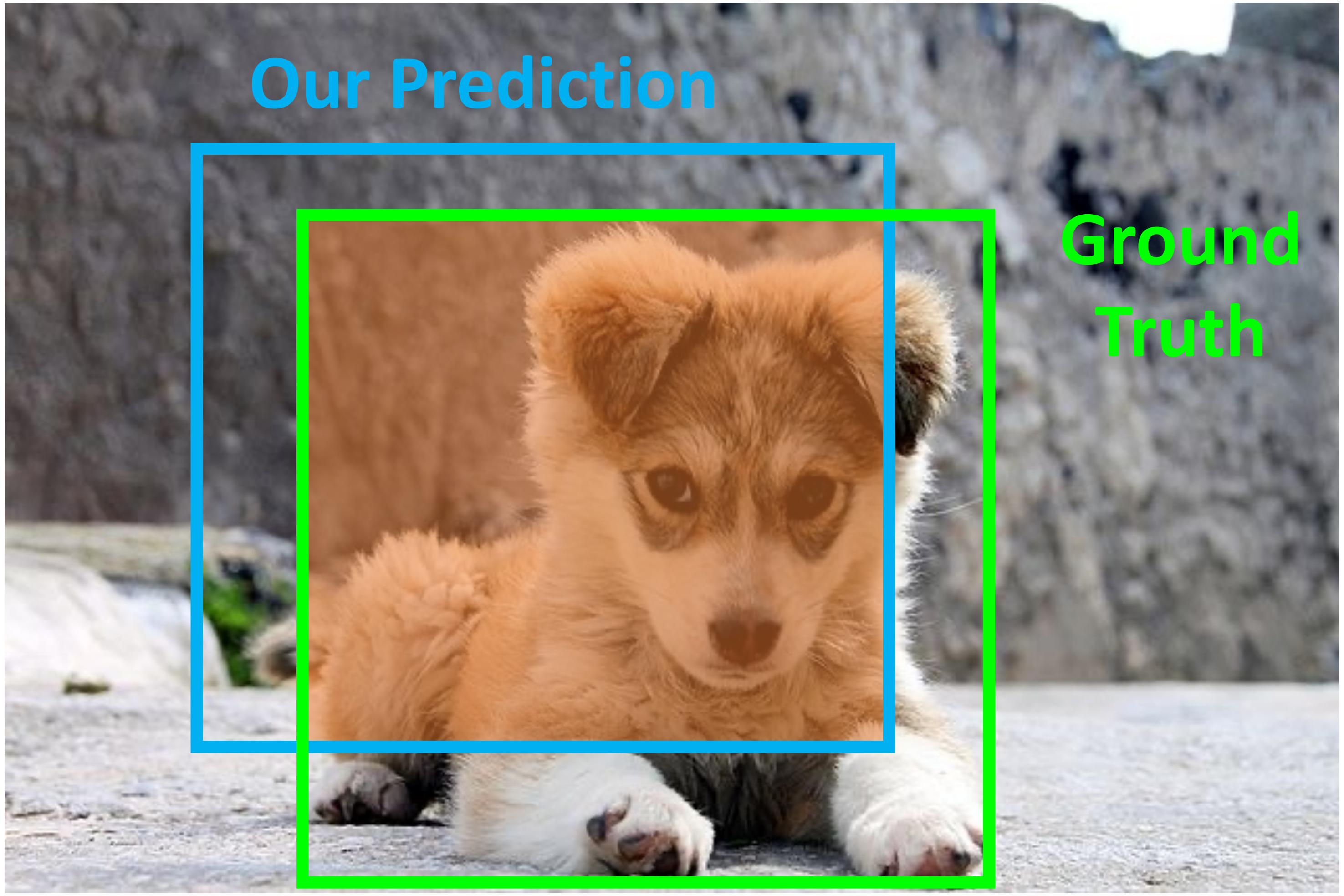
Girshick et al, "Rich feature hierarchies for accurate object detection and semantic segmentation", CVPR 2014.

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"):

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



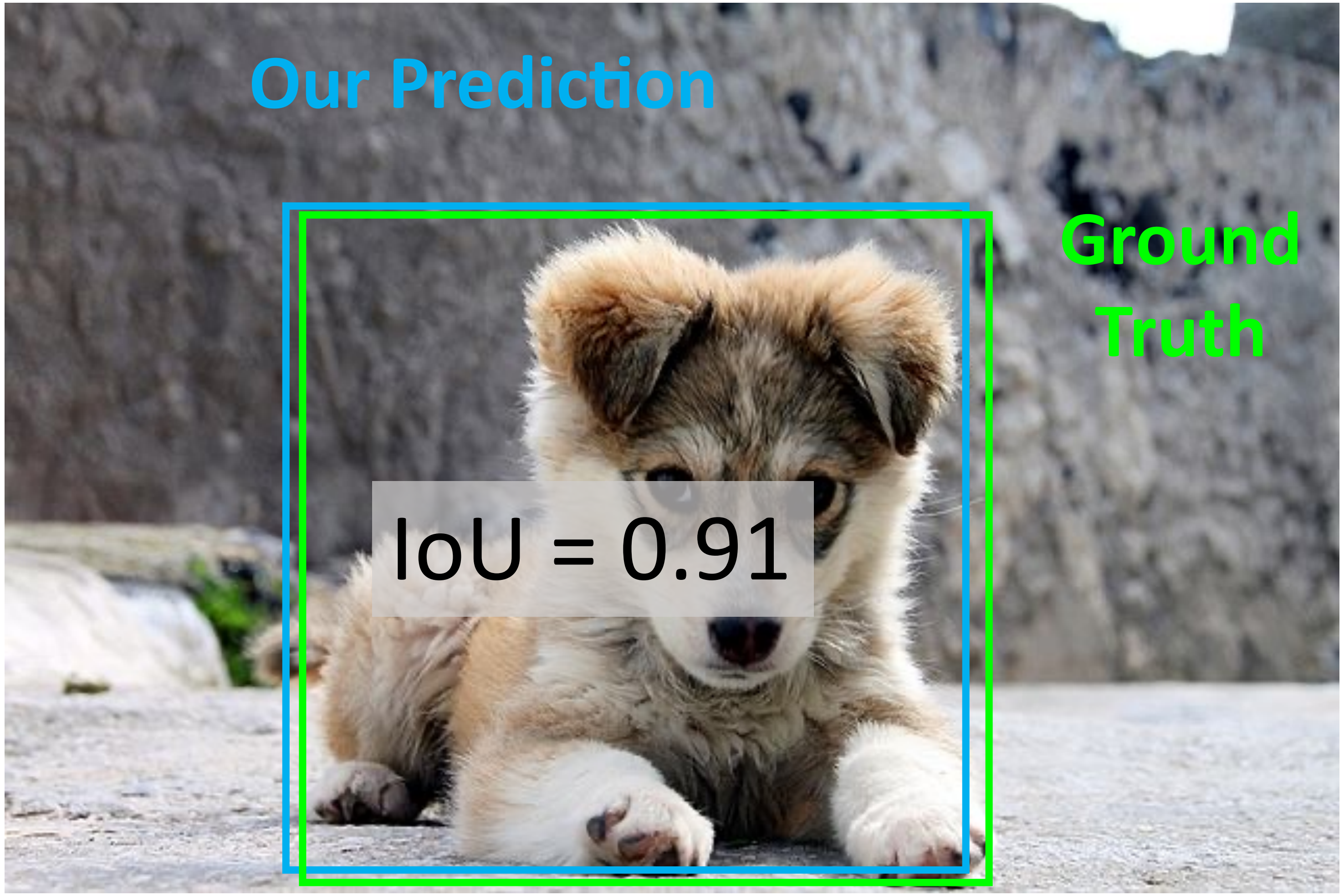
Comparing Boxes: Intersection over Union (IoU)

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

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

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

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

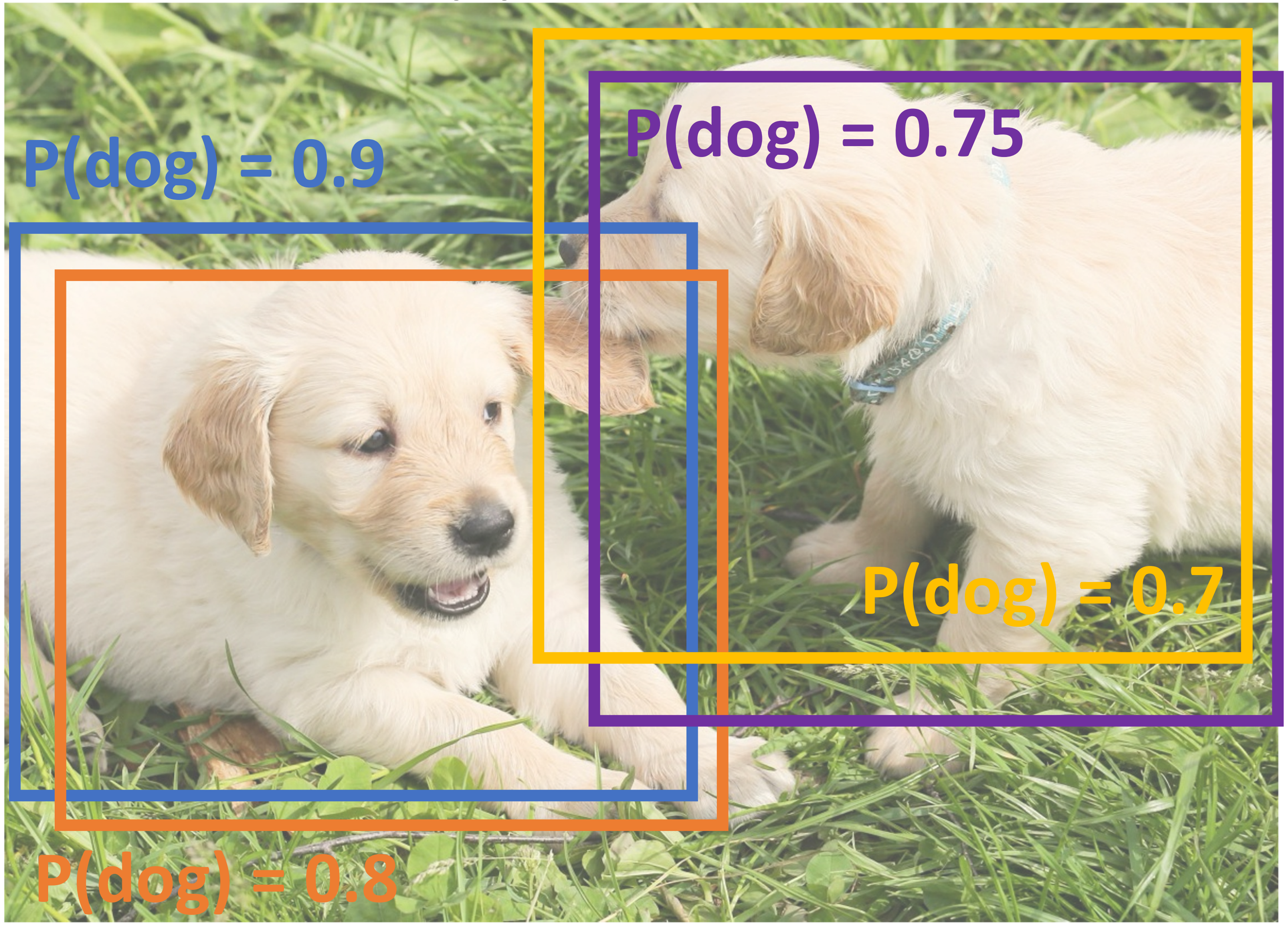


Overlapping Boxes: **Non-Max Suppression (NMS)**

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 IoU > threshold (e.g. 0.7)
3. If any boxes remain, GOTO 1

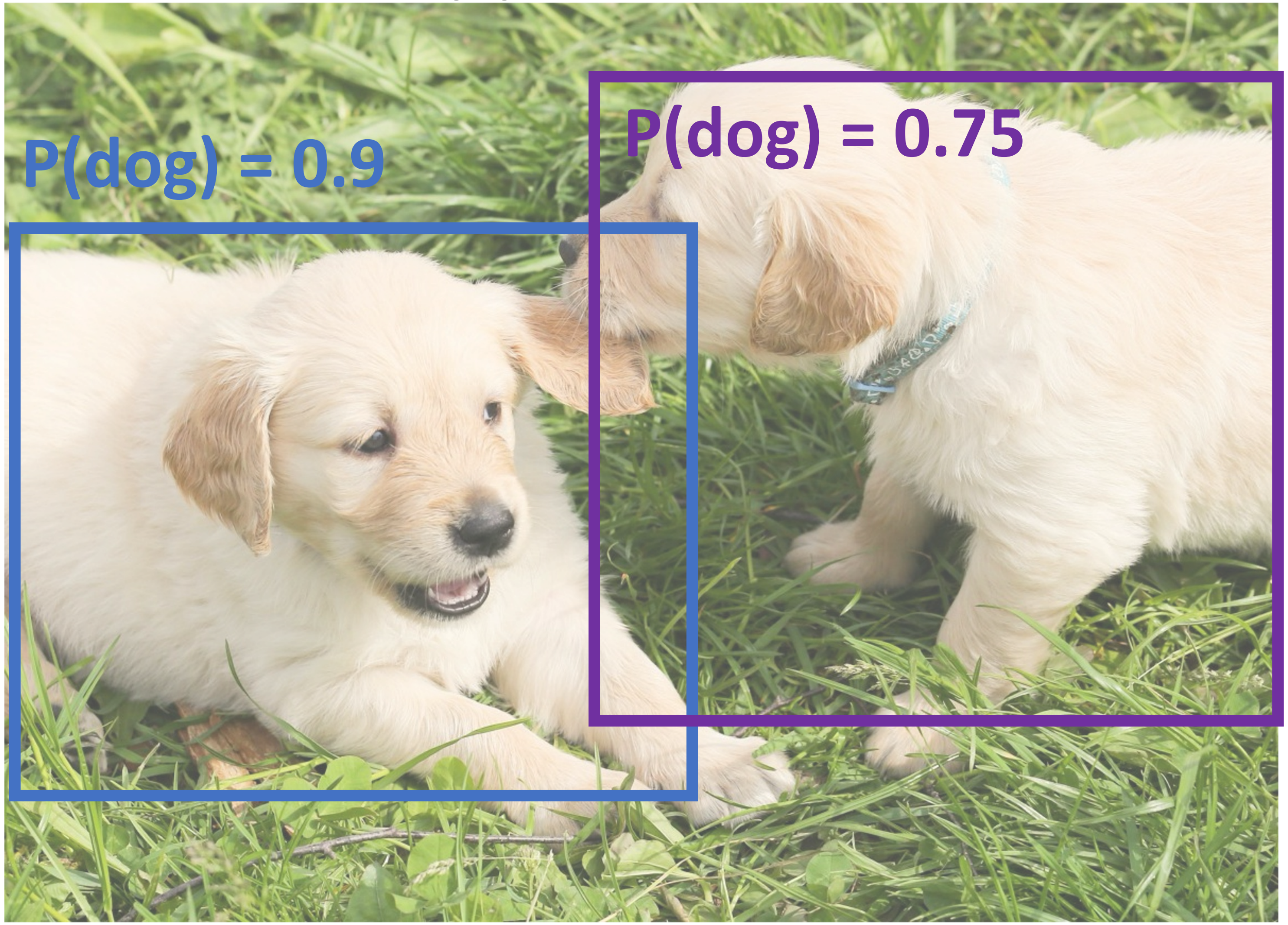


Overlapping Boxes: **Non-Max Suppression (NMS)**

Problem: Object detectors often output many overlapping detections:

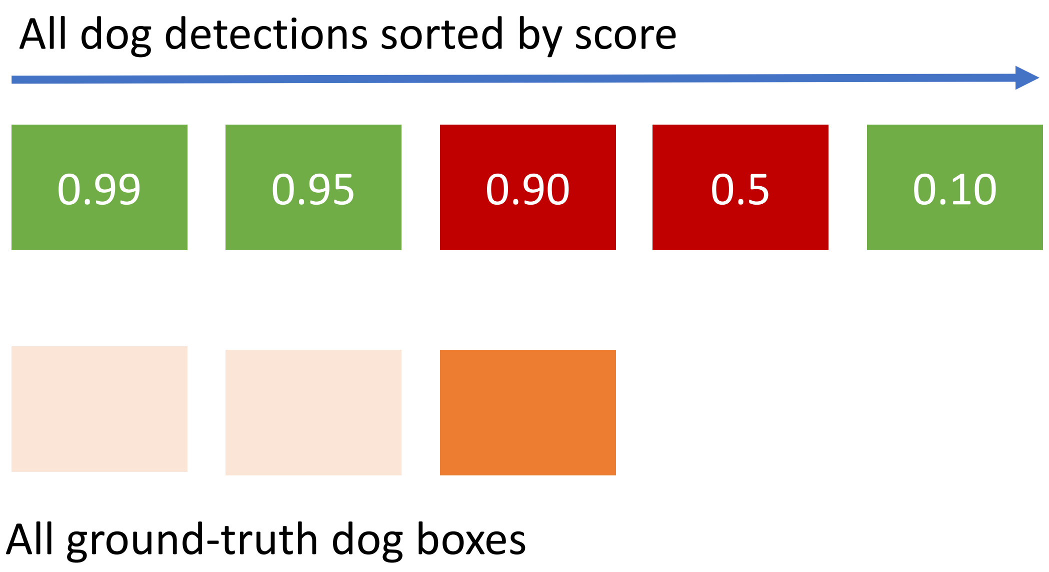
Solution: Post-process raw detections using **Non-Max Suppression (NMS)**

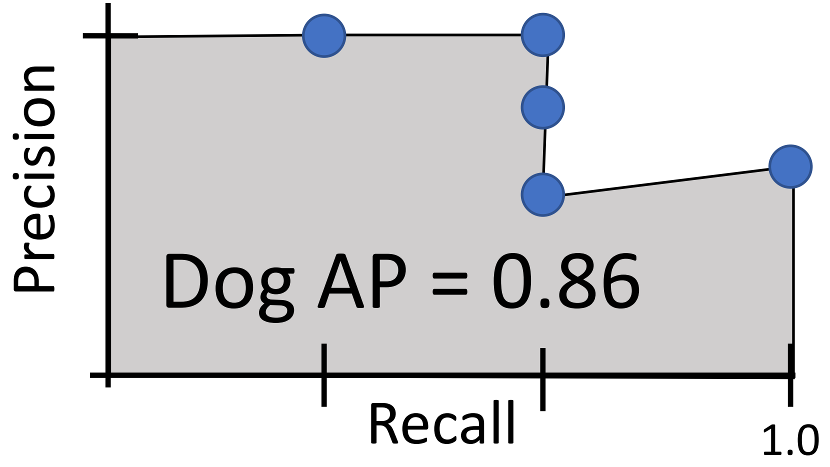
1. Select next highest-scoring box
2. Eliminate lower-scoring boxes with IoU > threshold (e.g. 0.7)
3. If any boxes remain, GOTO 1



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 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





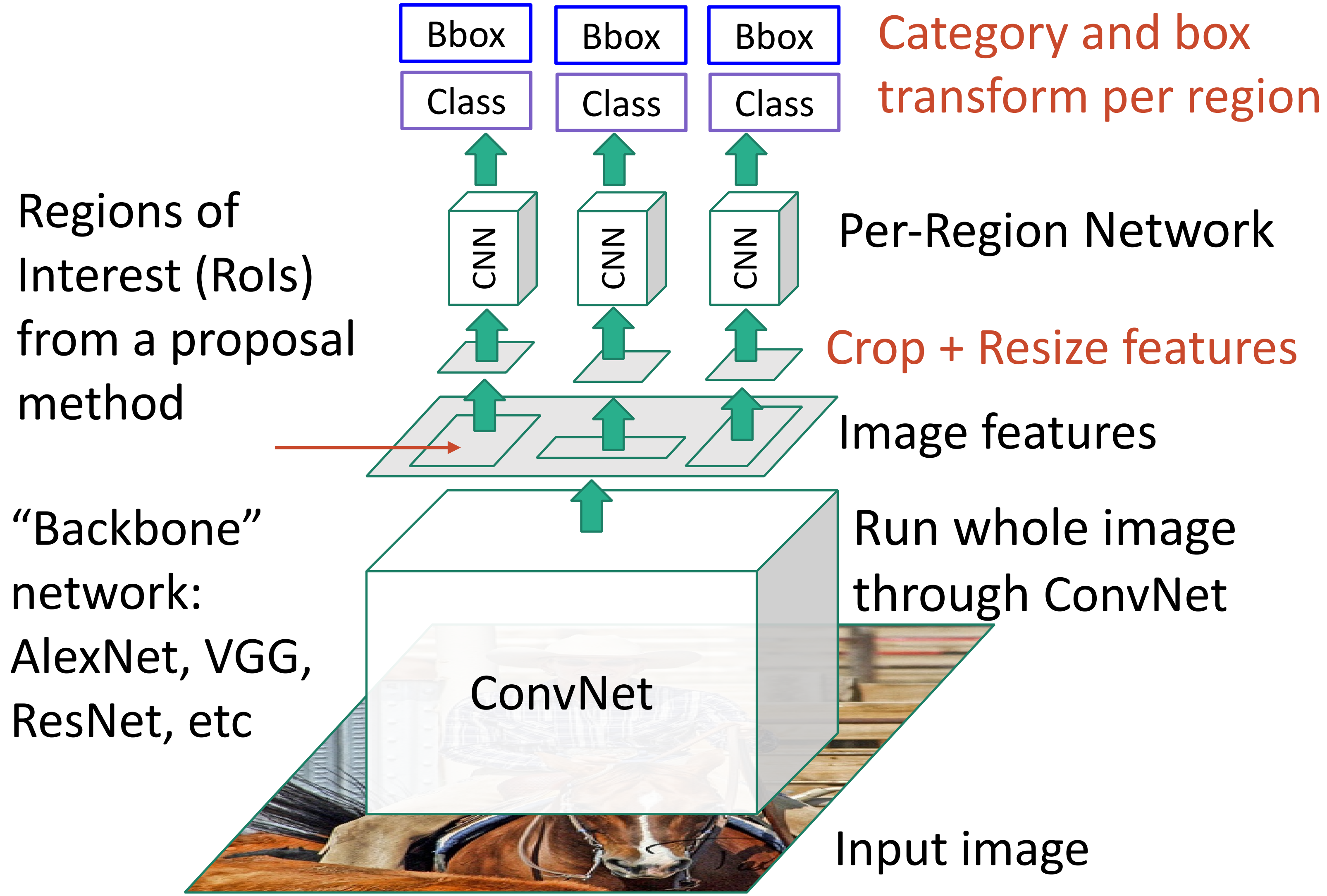
Evaluating Object Detectors: Mean Average Precision (mAP)

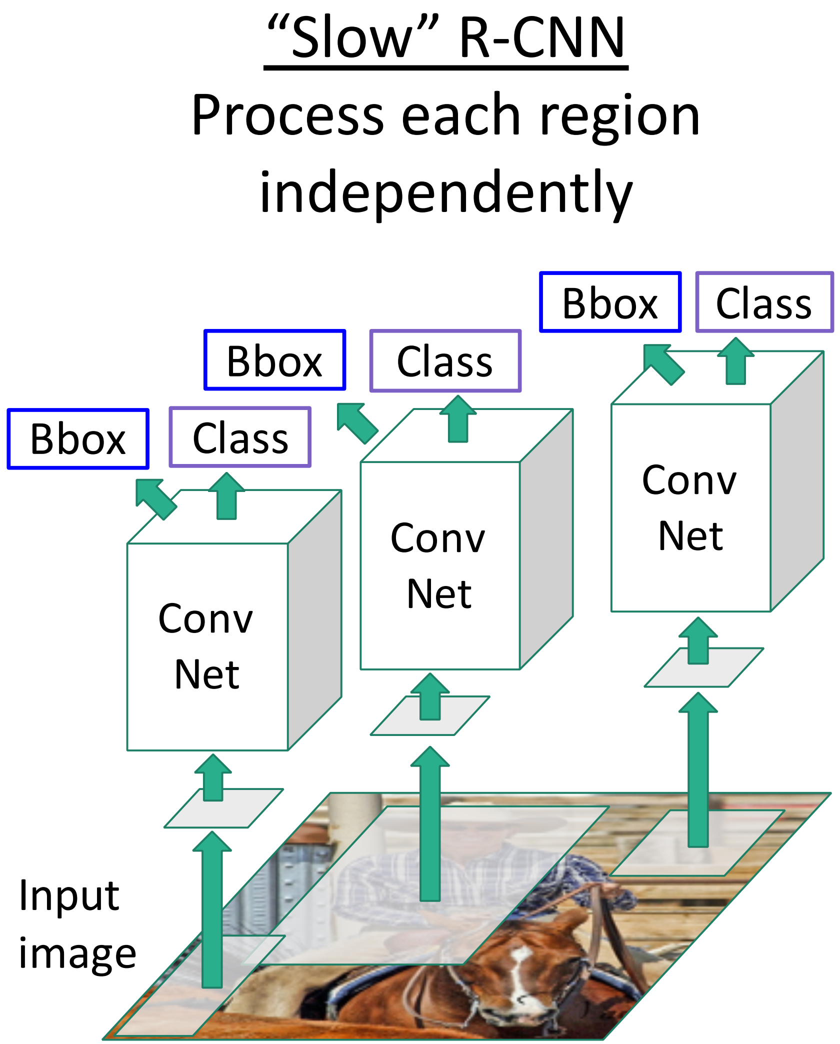
1. Run object detector on all test images (with NMS)
2. For each category, compute Average Precision (AP) = area under Precision vs Recall Curve
    1. For each detection (highest score to lowest score)
        1. If it matches some GT box with IoU > 0.5, mark it as positive and eliminate the GT
        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

mAP@0.5 = 0.77
mAP@0.55 = 0.71
mAP@0.60 = 0.65
…
mAP@0.95 = 0.2

COCO mAP = 0.4

Fast R-CNN





Girshick, "Fast R-CNN", ICCV 2015. Figure copyright Ross Girshick, 2015; source. Reproduced with permission.

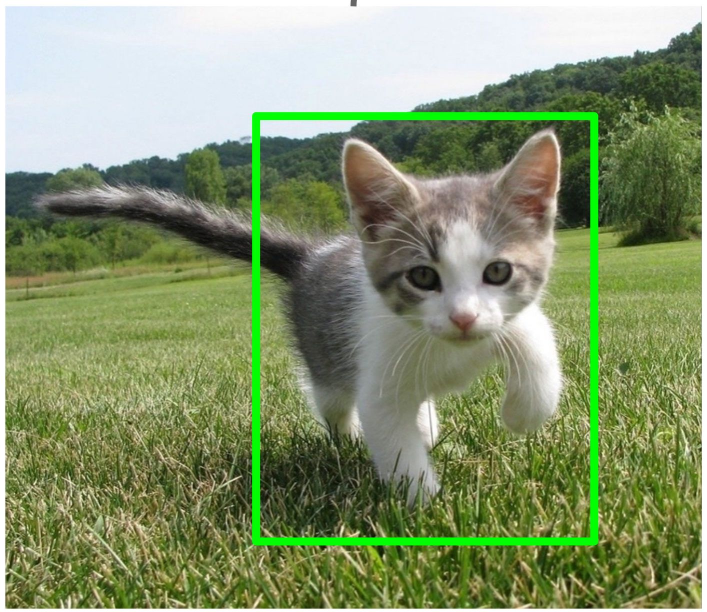
Cropping Features: RoI Pool

Project proposal onto features

"Snap" to grid cells

Divide into 2x2 grid of (roughly) equal subregions

Max-pool within each subregion



Input Image
(e.g. 3 x 640 x 480)

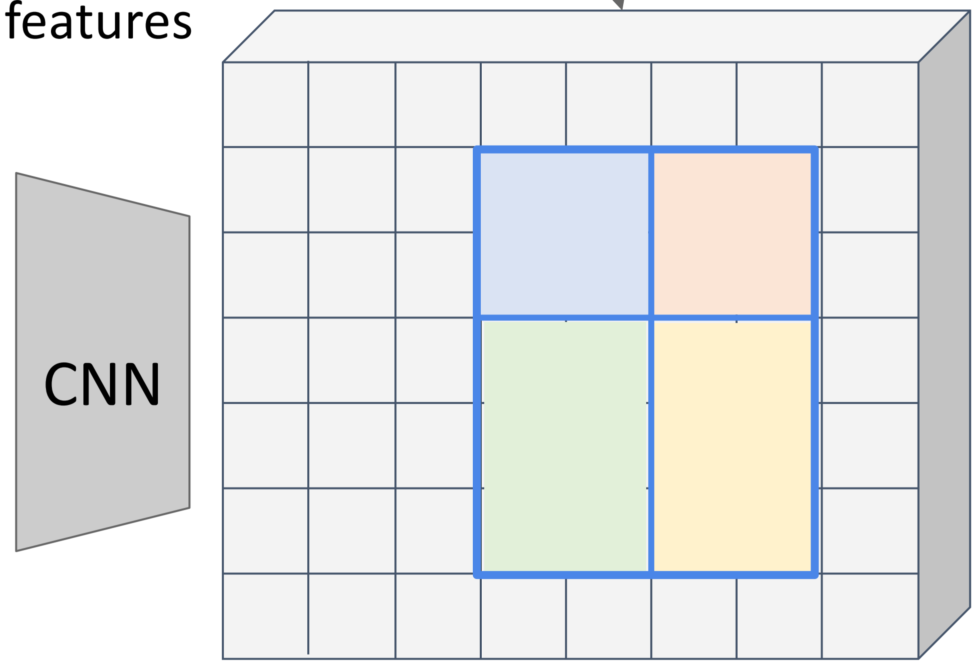
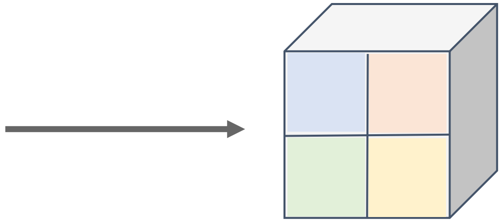


Image features
(e.g. 512 x 20 x 15)



Region features
(here 512 x 2 x 2;
In practice e.g 512 x 7 x 7)

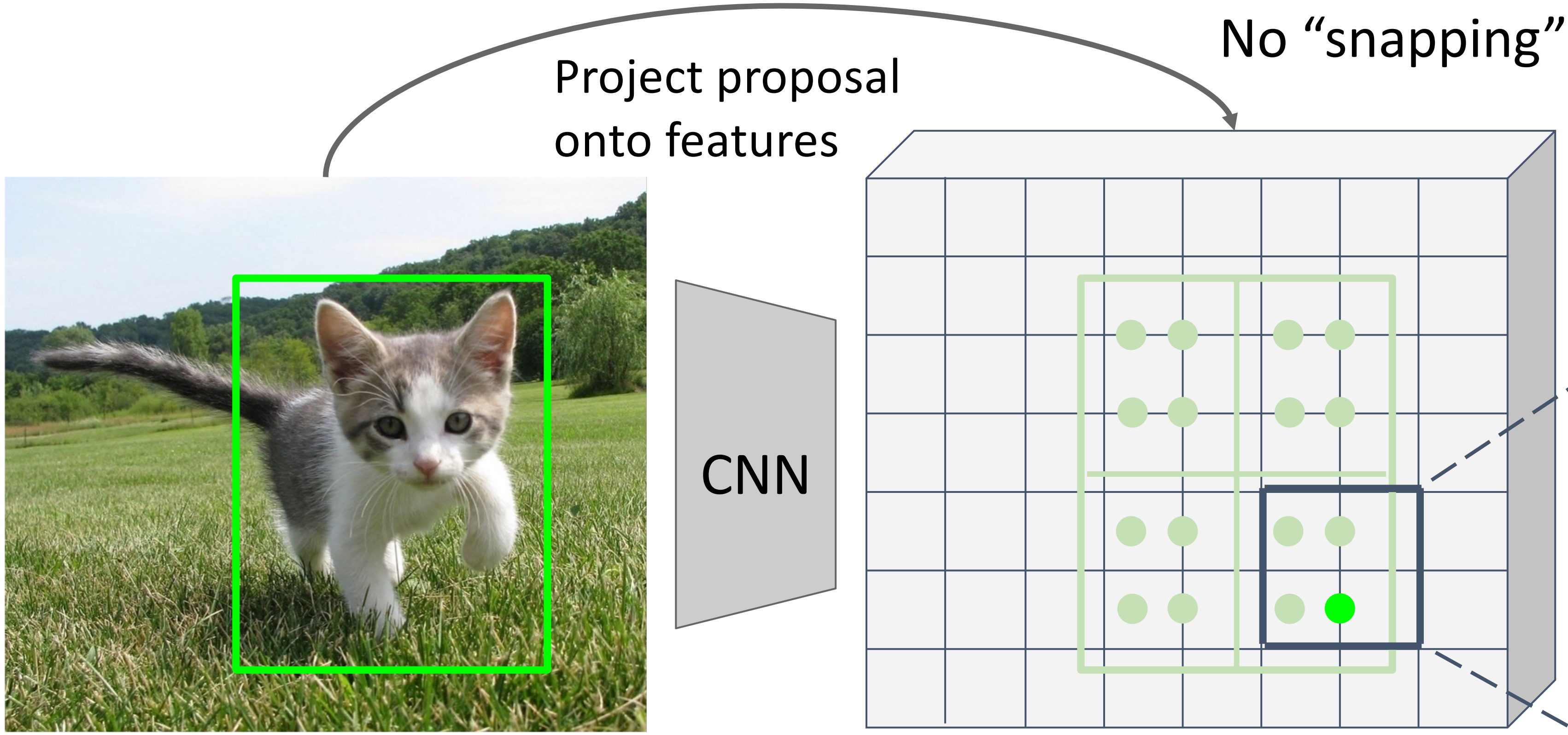
Problem: Slight misalignment due to snapping; different-sized subregions is weird

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

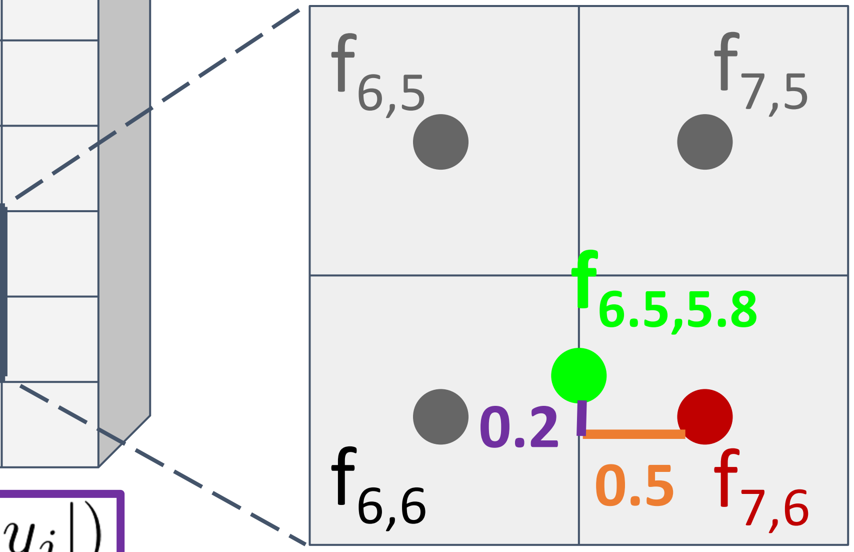
Girshick, "Fast R-CNN", ICCV 2015.

Cropping Features: RoI Align

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



Sample features at regularly-spaced points in each subregion using **bilinear interpolation**



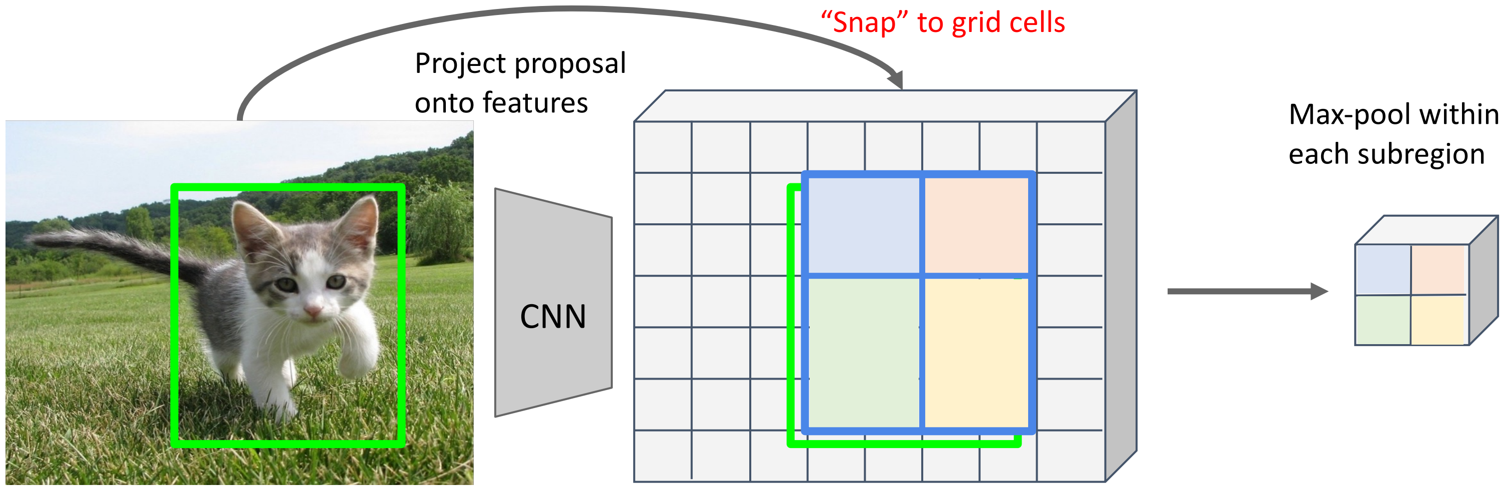
$$f_{xy} = \sum_{i,j=1}^{2} \boxed{f_{i,j}} \boxed{\max(0, 1 - |x - x_i|)} \boxed{\max(0, 1 - |y - y_j|)}$$

$$f_{6.5,5.8} = (f_{6,5} * 0.5 * 0.2) + (f_{7,5} * 0.5 * 0.2) + (f_{6,6} * 0.5 * 0.8) + (f_{7,6} * 0.5 * 0.8)$$

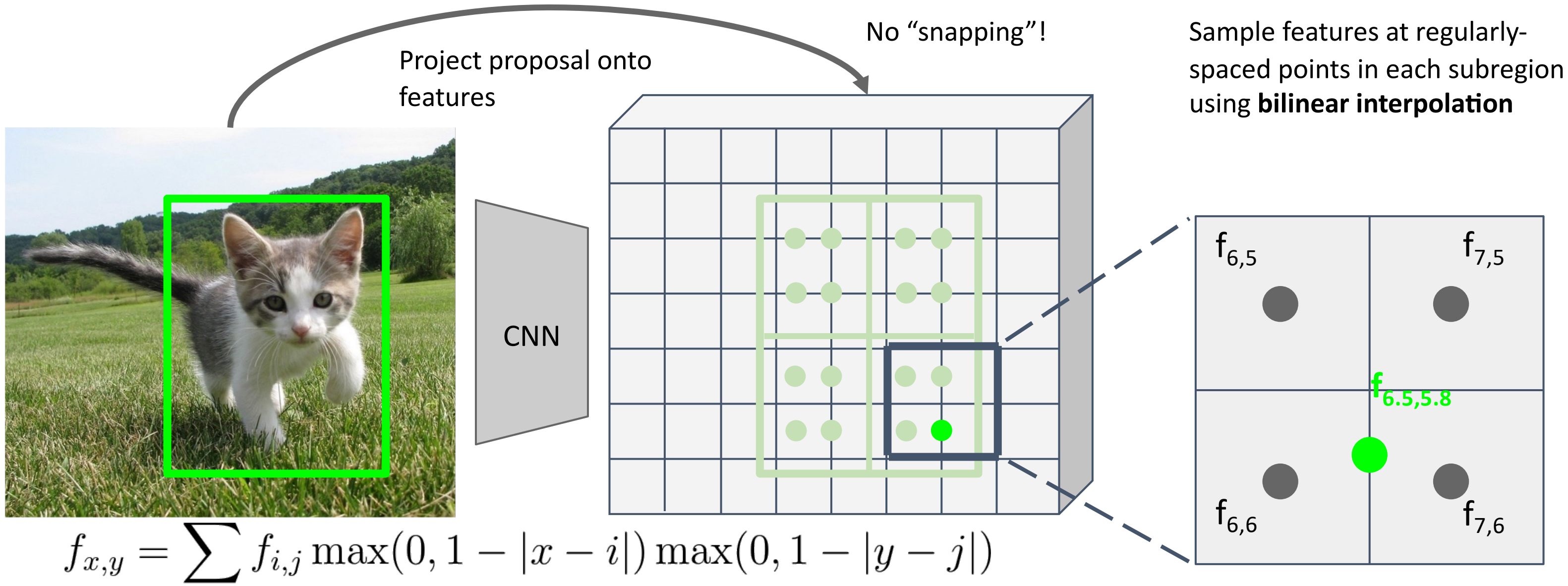
Feature f_{xy} for point (x, y) is a linear combination of features at its four neighboring grid cells:

Cropping Features

RoI Pool



RoI Align



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

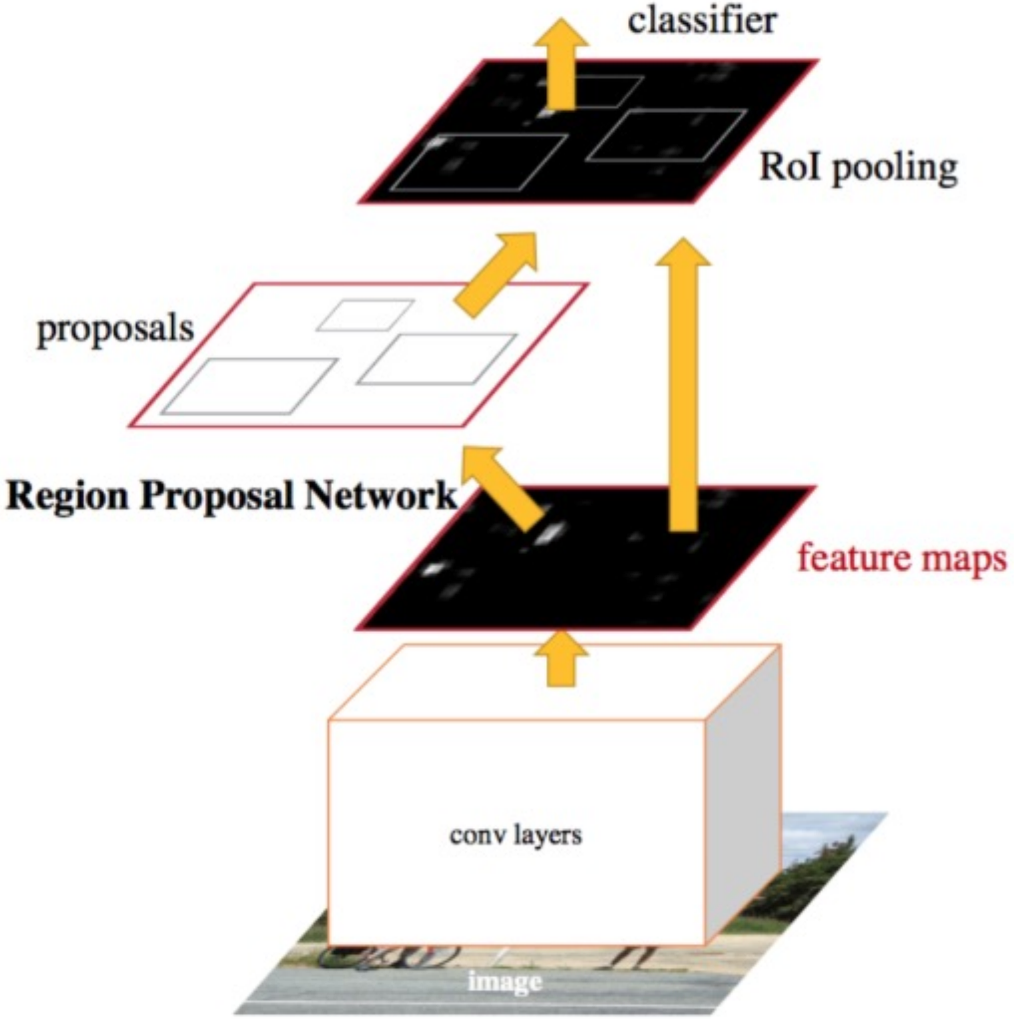
$f_{6.5,5.8} = (f_{6,5} * 0.5 * 0.2) + (f_{7,5} * 0.5 * 0.2)$
$+ (f_{6,6} * 0.5 * 0.8) + (f_{7,6} * 0.5 * 0.8)$

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

Fast<u>er</u> 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

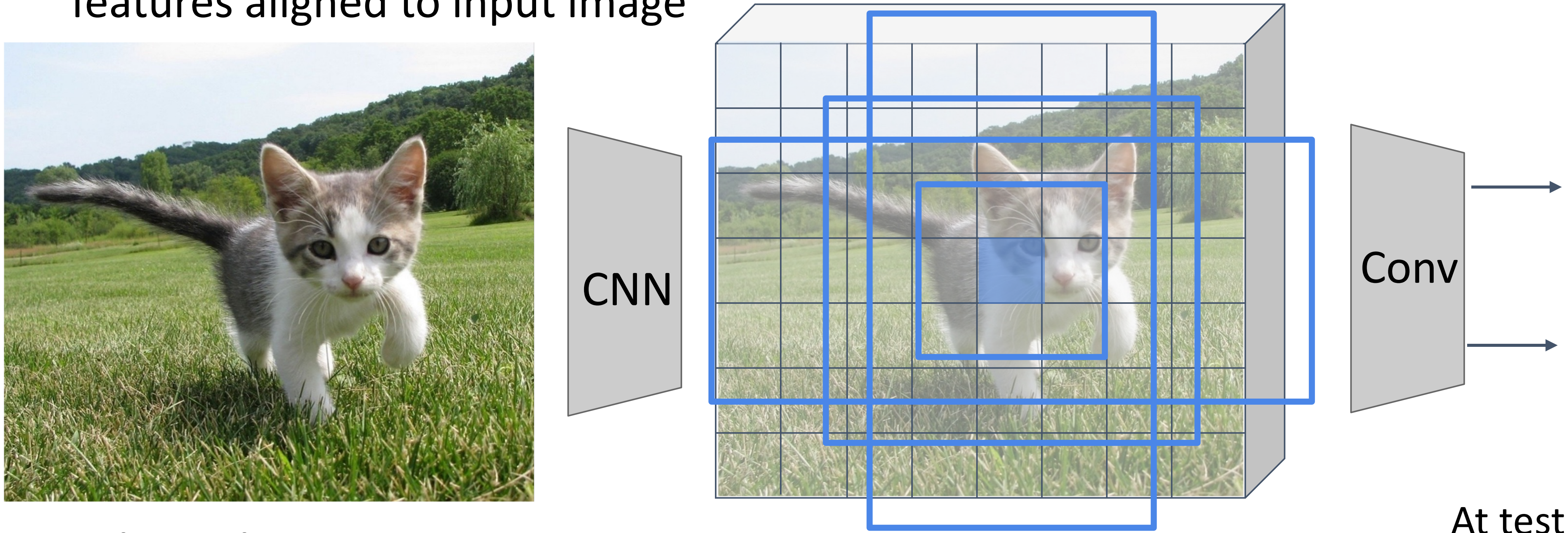


Ren et al, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”, NIPS 2015

Region Proposal Network (RPN)

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

Run backbone CNN to get features aligned to input image



Anchor is an object?
K x 20 x 15

Box transforms
4K x 20 x 15

Input Image
(e.g. 3 x 640 x 480)

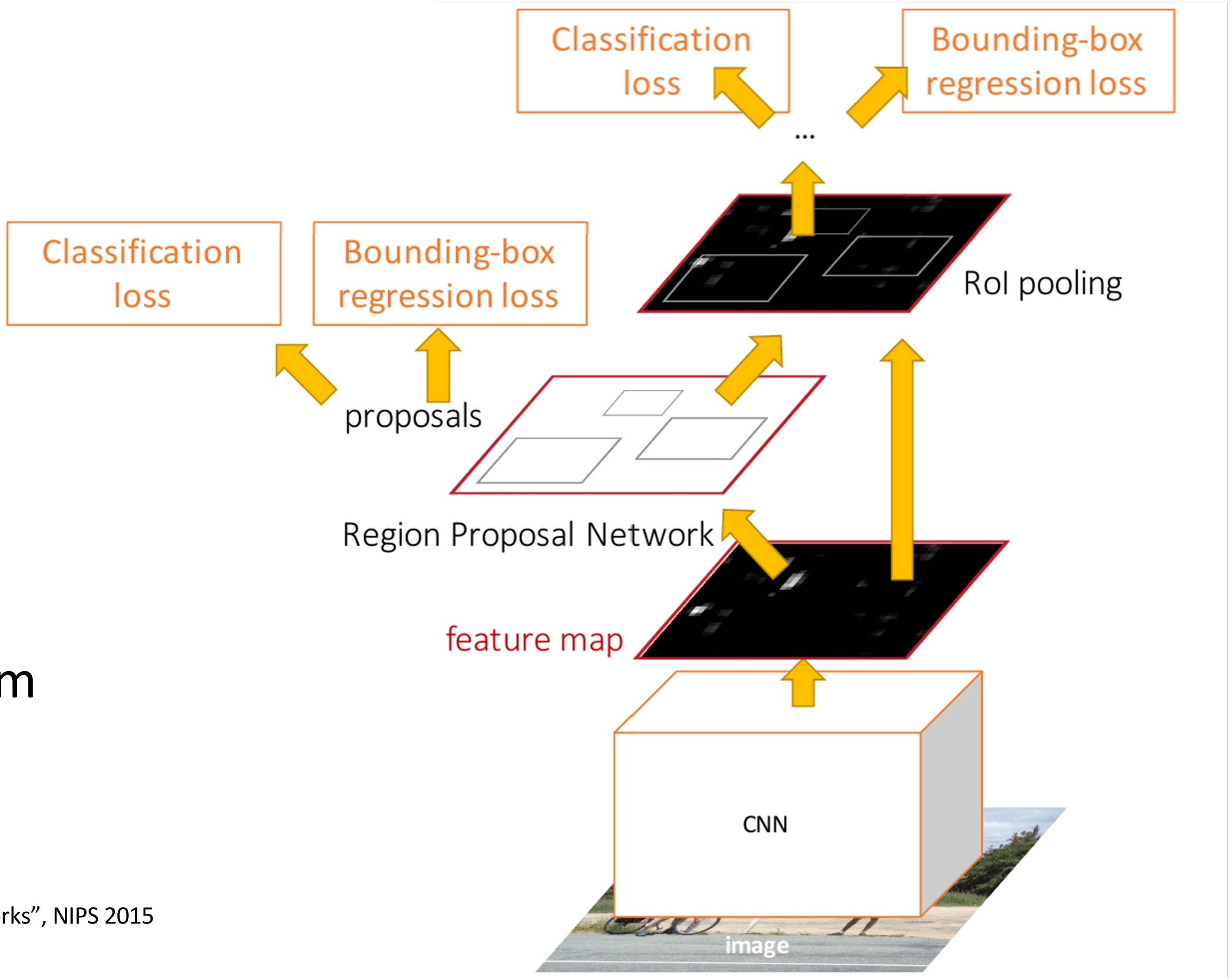
Image features
(e.g. 512 x 20 x 15)

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

Faster R-CNN: Learnable Region Proposals

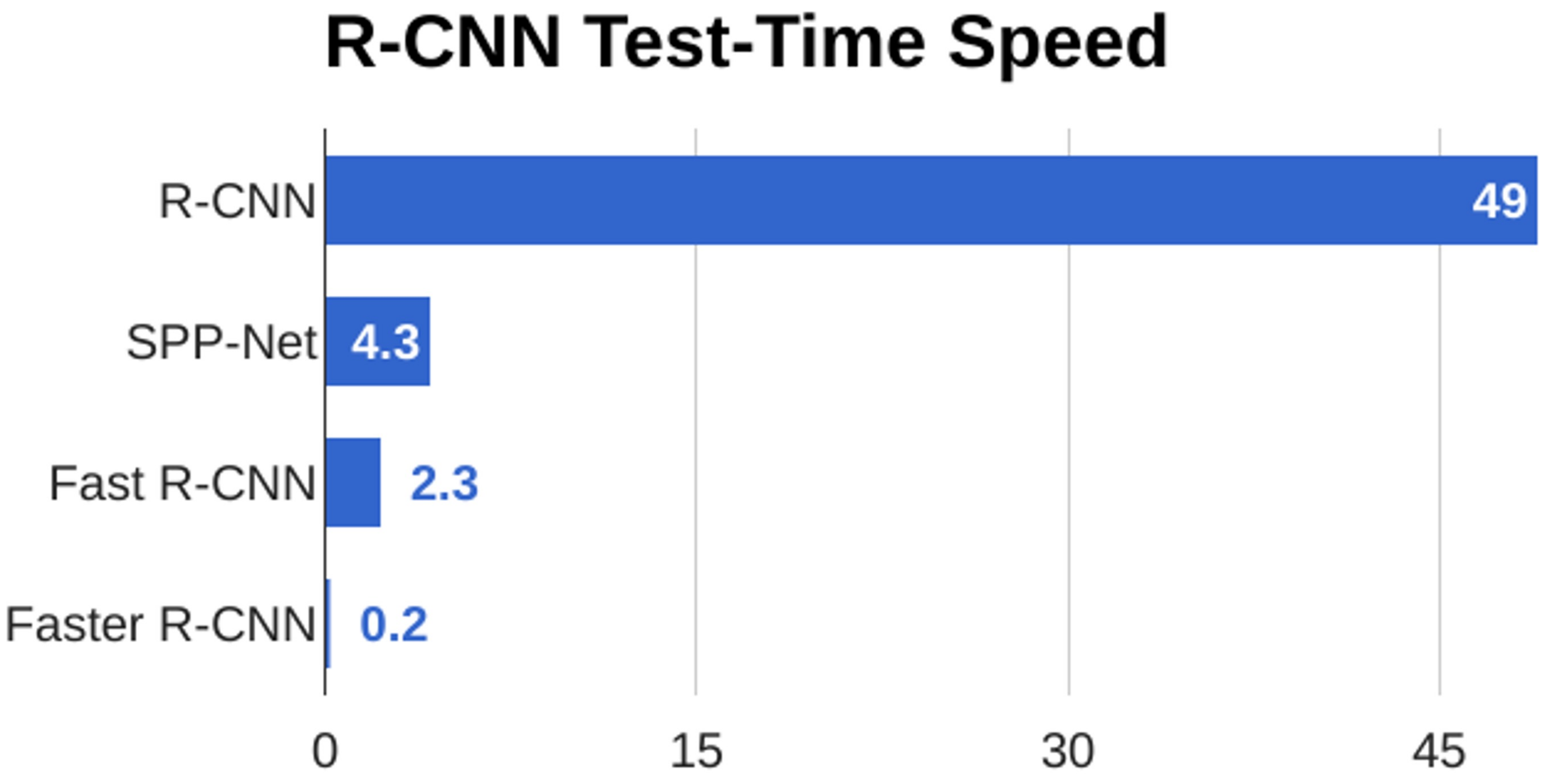
Jointly train with 4 losses:

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

Faster R-CNN: Learnable Region Proposals



Single-Stage Object Detection: YOLO and SSD

RPN: Classify each anchor as object / not object

Single-Stage Detector: Classify each object as one of C categories (or background)

Run backbone CNN to get features aligned to input image

Input Image
(e.g. 3 x 640 x 480)

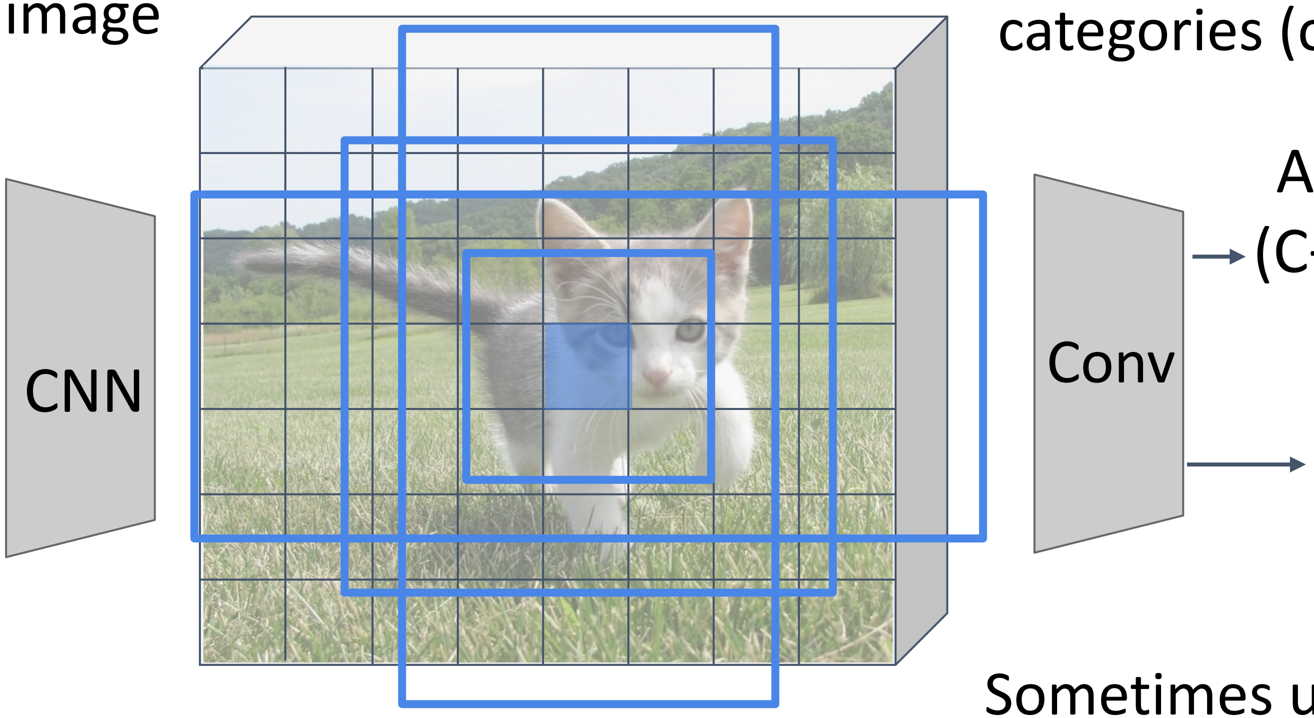


Image features
(e.g. 512 x 20 x 15)

Anchor category
→ (C+1) x K x 20 x 15

Box transforms
C x 4K x 20 x 15

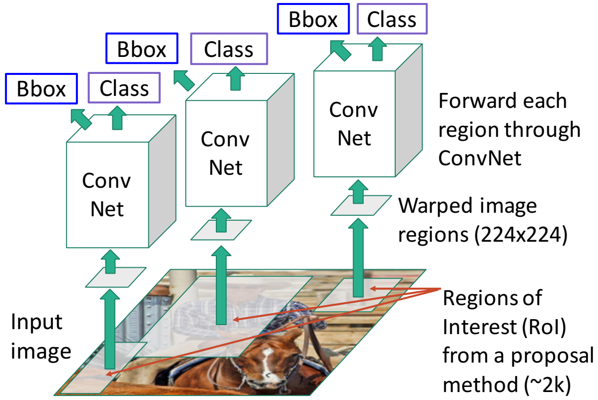
Sometimes use **category-specific regression**: Predict different box transforms for each category

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

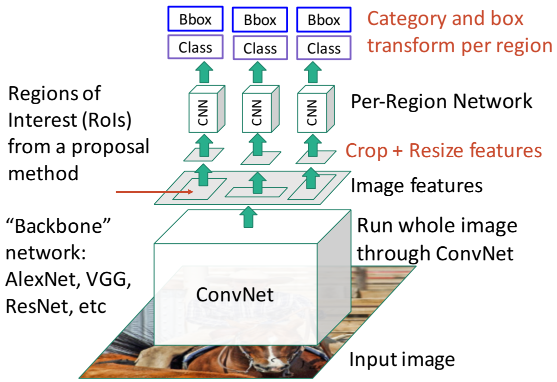
YOLO series

Summary of object detection

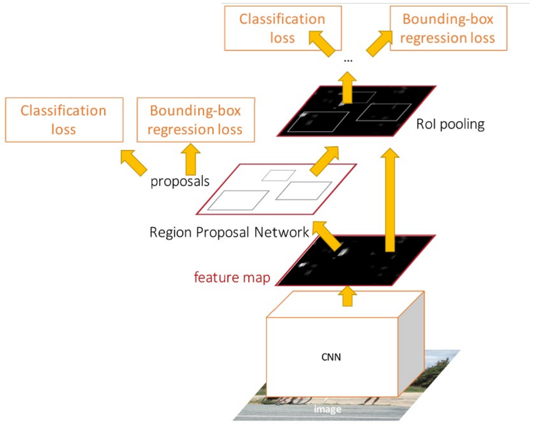
"Slow" R-CNN: Run CNN independently for each region



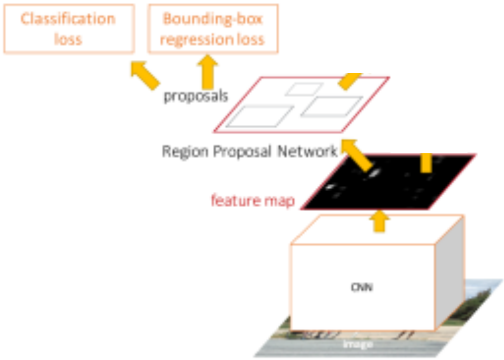
Fast R-CNN: Apply differentiable cropping to shared image features



Faster R-CNN: Compute proposals with CNN



Single-Stage: Fully convolutional detector



Semantic Segmentation: Fully Convolutional Network

Downsampling: Pooling, strided convolution

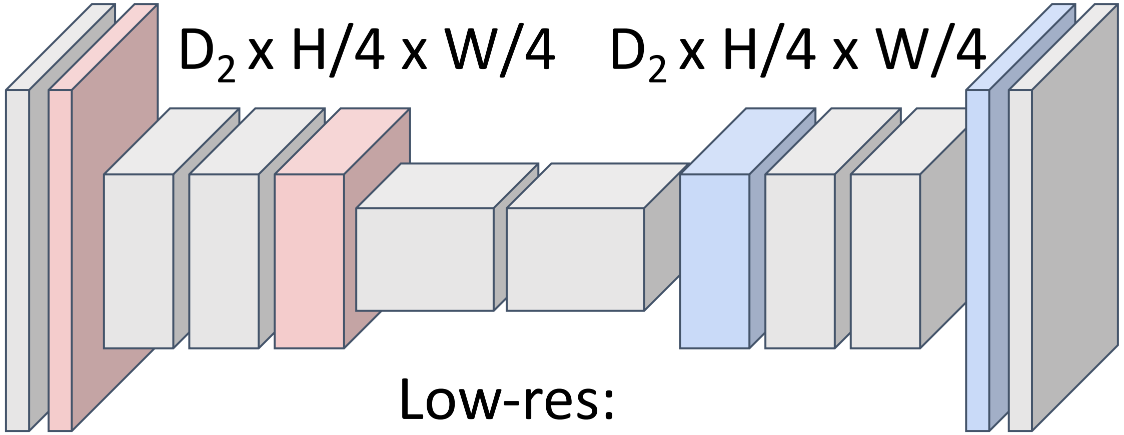
Design network as a bunch of convolutional layers, with **downsampling** and **upsampling** inside the network!

Upsampling: Interpolation, transposed conv

Input:
3 x H x W

Med-res:
D_2 x H/4 x W/4

Med-res:
D_2 x H/4 x W/4



High-res:
D_1 x H/2 x W/2

Low-res:
D_3 x H/4 x W/4

High-res:
D_1 x H/2 x W/2

Predictions:
H x W

Loss function: Per-Pixel cross-entropy

Long, Shelhamer, and Darrell, "Fully Convolutional Networks for Semantic Segmentation", CVPR 2015
Noh et al, "Learning Deconvolution Network for Semantic Segmentation", ICCV 2015

In-Network Upsampling: "Max Unpooling"

Max Pooling: Remember which position had the max

Max Unpooling: Place into remembered positions

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

→

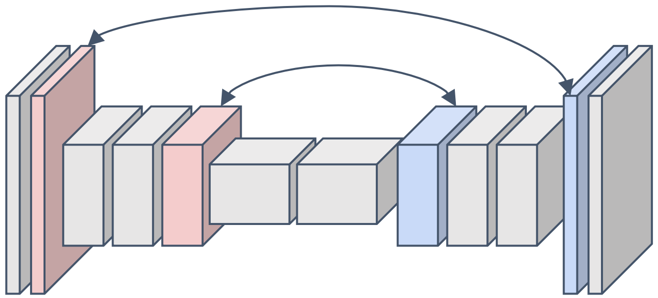
5	6
7	8

→ 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

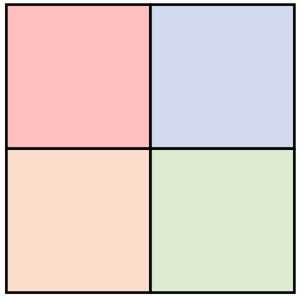
Noh et al, "Learning Deconvolution Network for Semantic Segmentation", ICCV 2015

Learnable Upsampling: Transposed Convolution

3 x 3 **convolution transpose**, stride 2

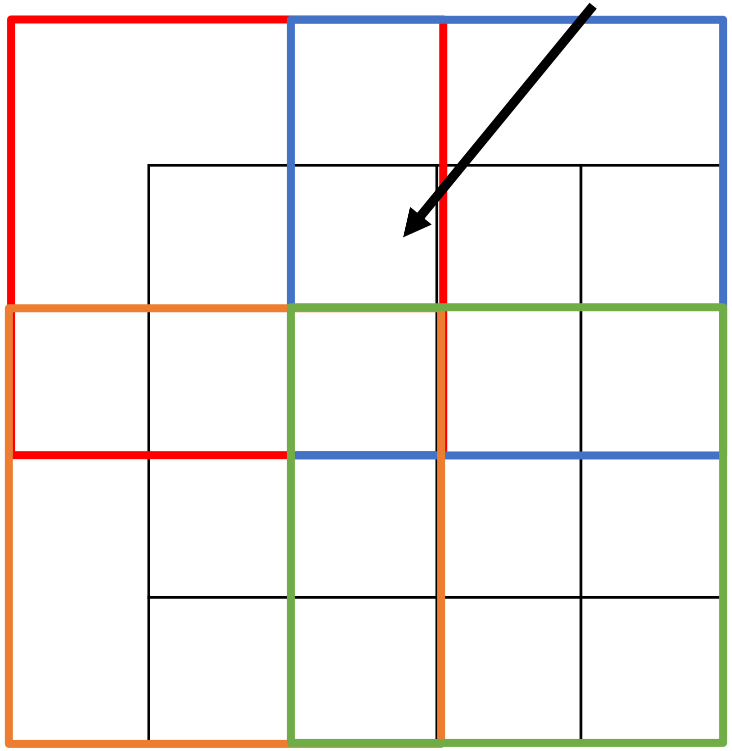
Sum where output overlaps

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



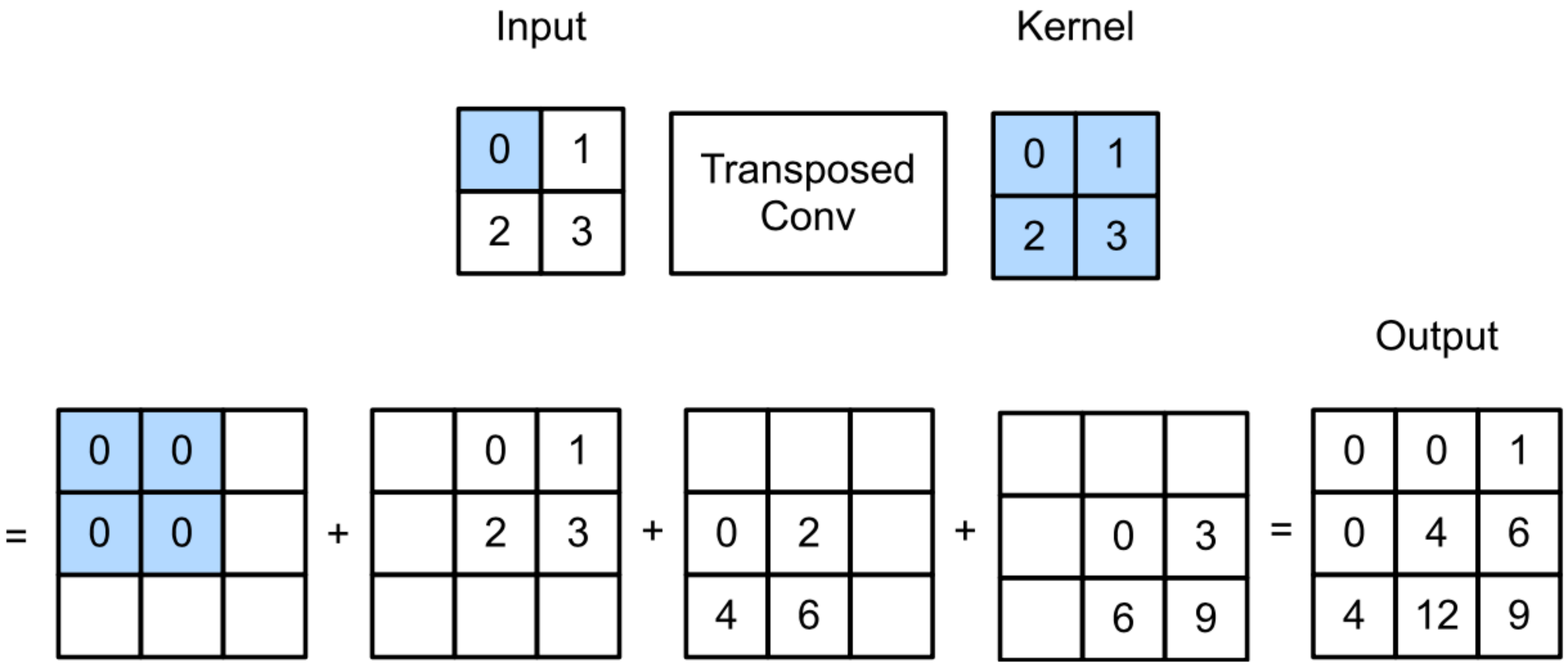
Input: 2 x 2

Weight filter by input value and copy to output



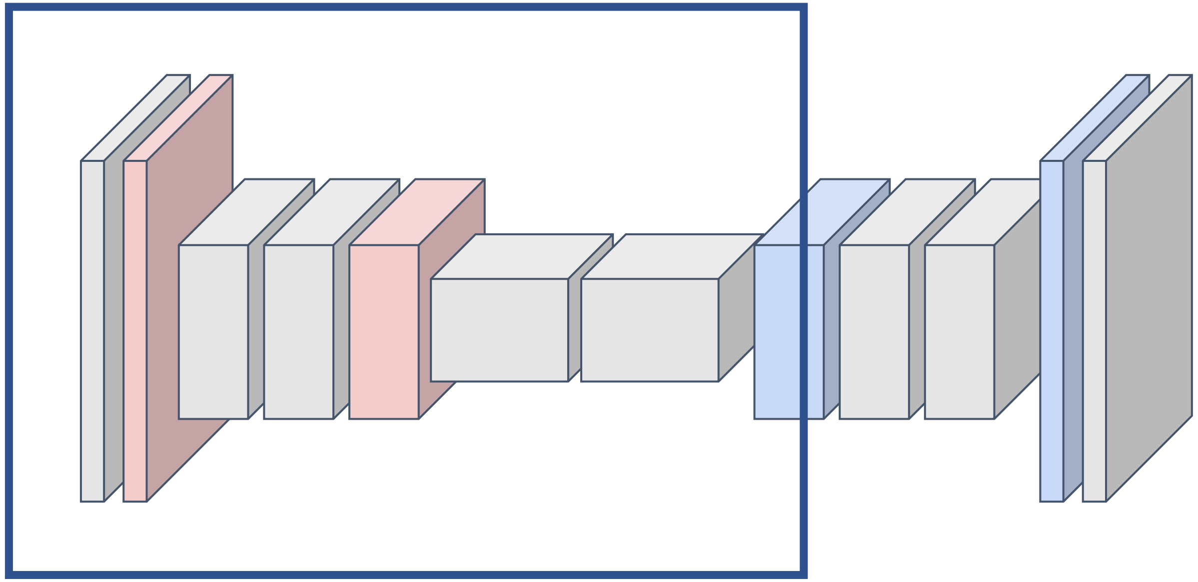
Output: 4 x 4

Transposed Convolution: 2D example



Innovation on encoder: How to better encode the scene context

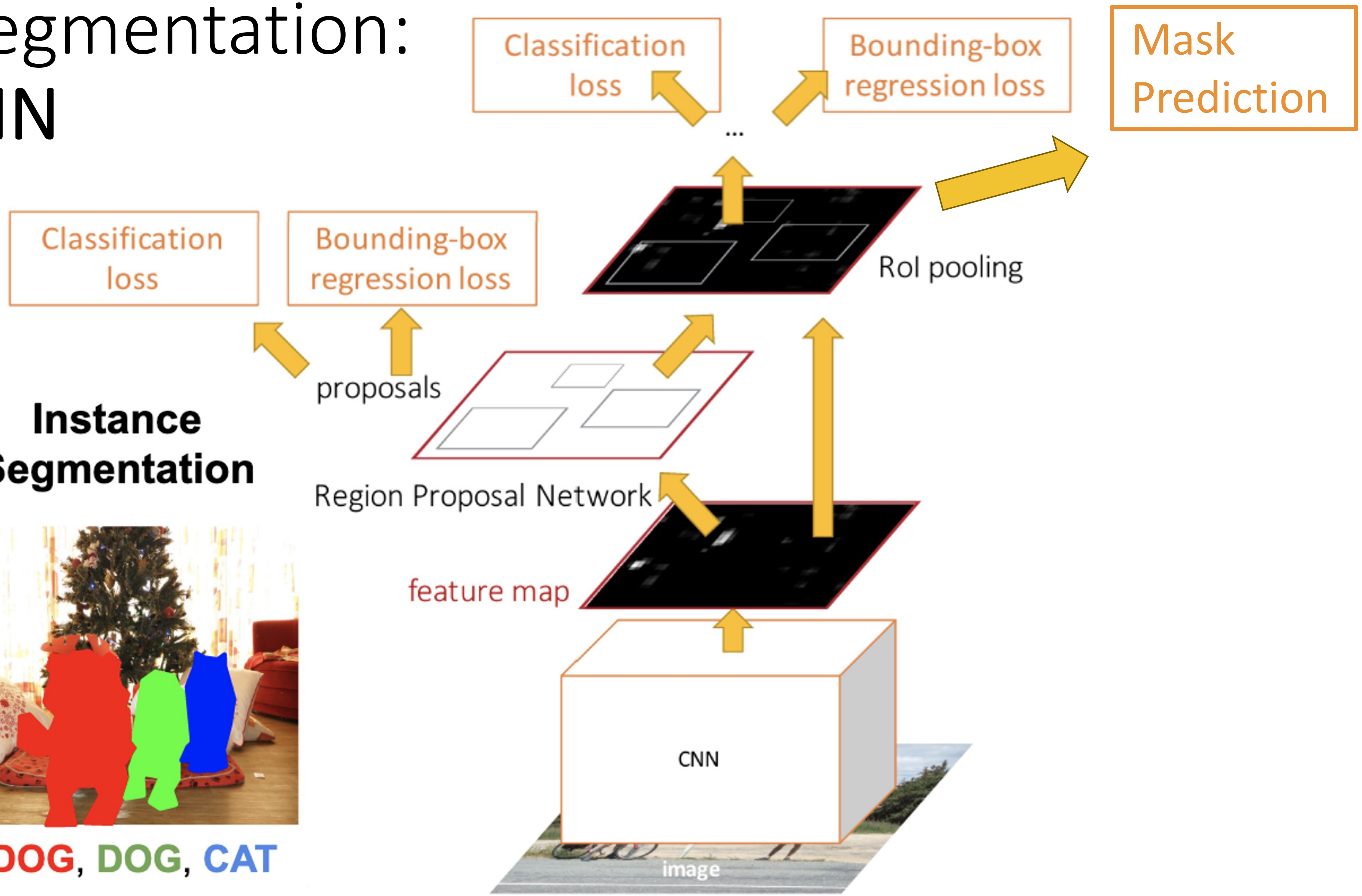
Input:
3 x H x W



Predictions:
H x W

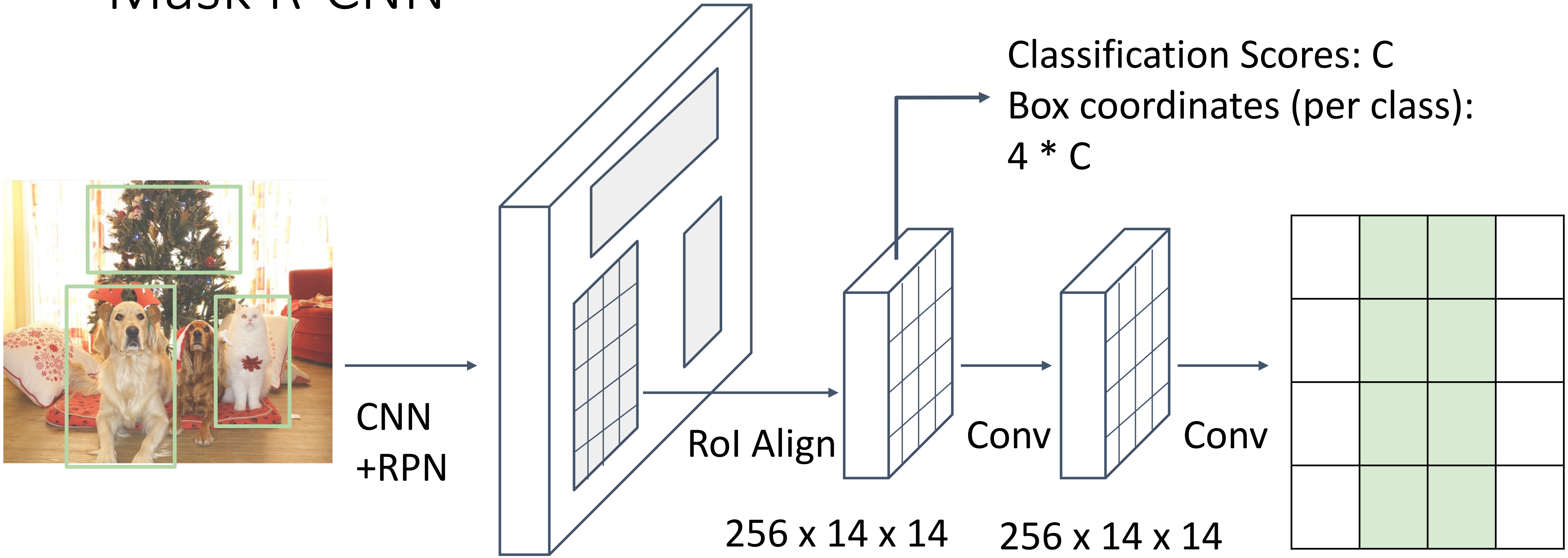
Dilated convolution, feature pyramid structures, U-Net structure

Instance Segmentation: Mask R-CNN



He et al, "Mask R-CNN", ICCV 2017

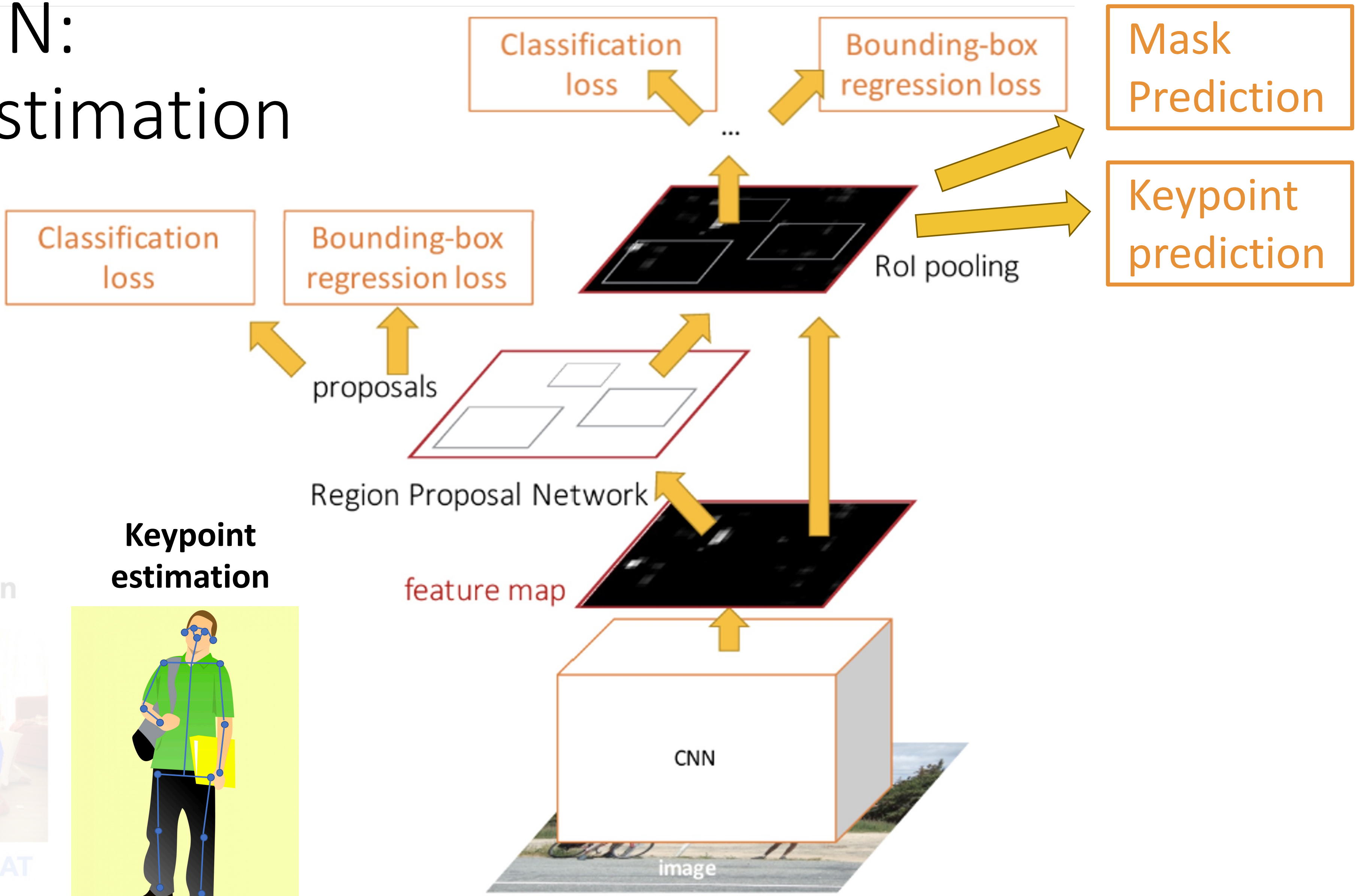
Mask R-CNN



Predict a mask for each of C classes:
C x 28 x 28 (mask size)

He et al, “Mask R-CNN”, ICCV 2017

Mask R-CNN: Keypoint Estimation



Object Detection

DOG, DOG, CAT

Instance Segmentation

DOG, DOG, CAT

Keypoint estimation

He et al, “Mask R-CNN”, ICCV 2017

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 -> y

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

Unsupervised Learning

Data: x

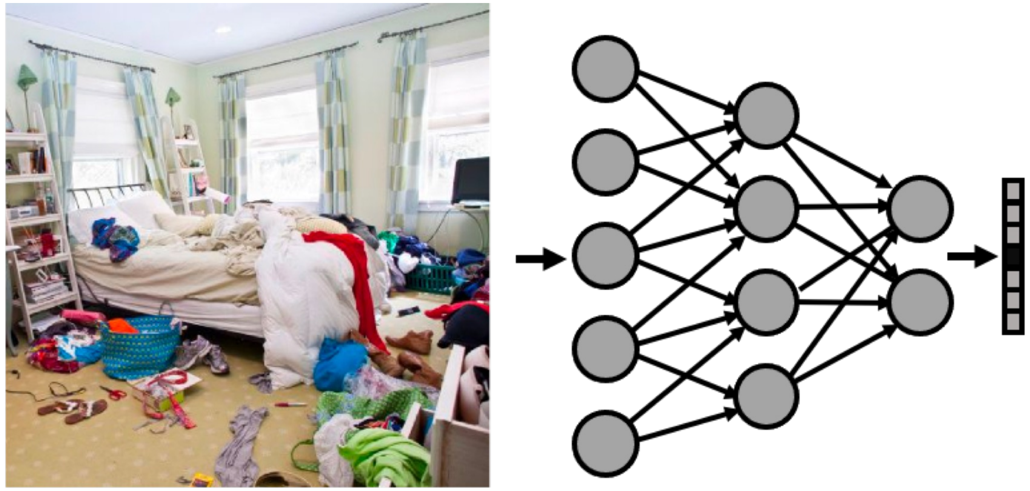
Just data, no labels!

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



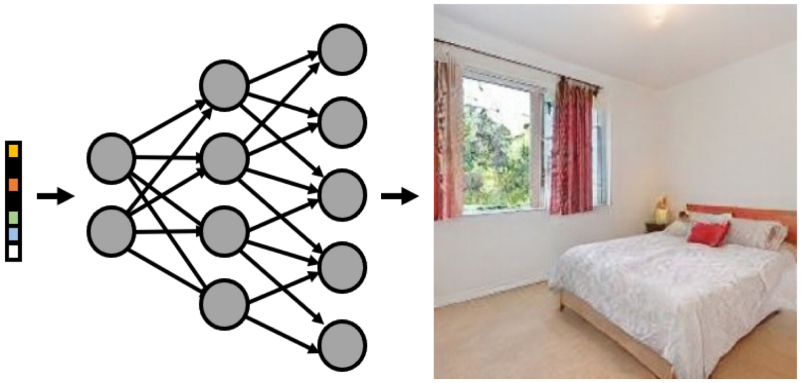
Scene categorization
Bedroom

Attribute prediction
Messy, natural-lighting

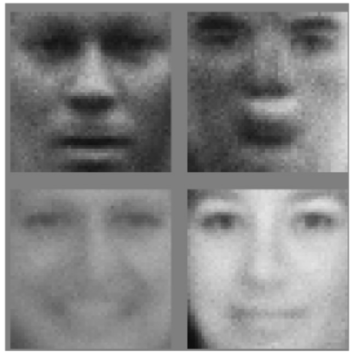
Semantic segmentation

Recent Advances in Generative Models

Generative Model



2014



GAN

2015

DCGAN

2017

PG-GAN

2018

StyleGAN

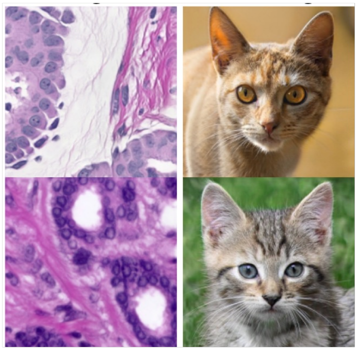
2018

BigGAN

2019

StyleGANv2

2020



StyleGAN-ADA

2021

Alias-Free GAN

Taxonomy of Generative Models

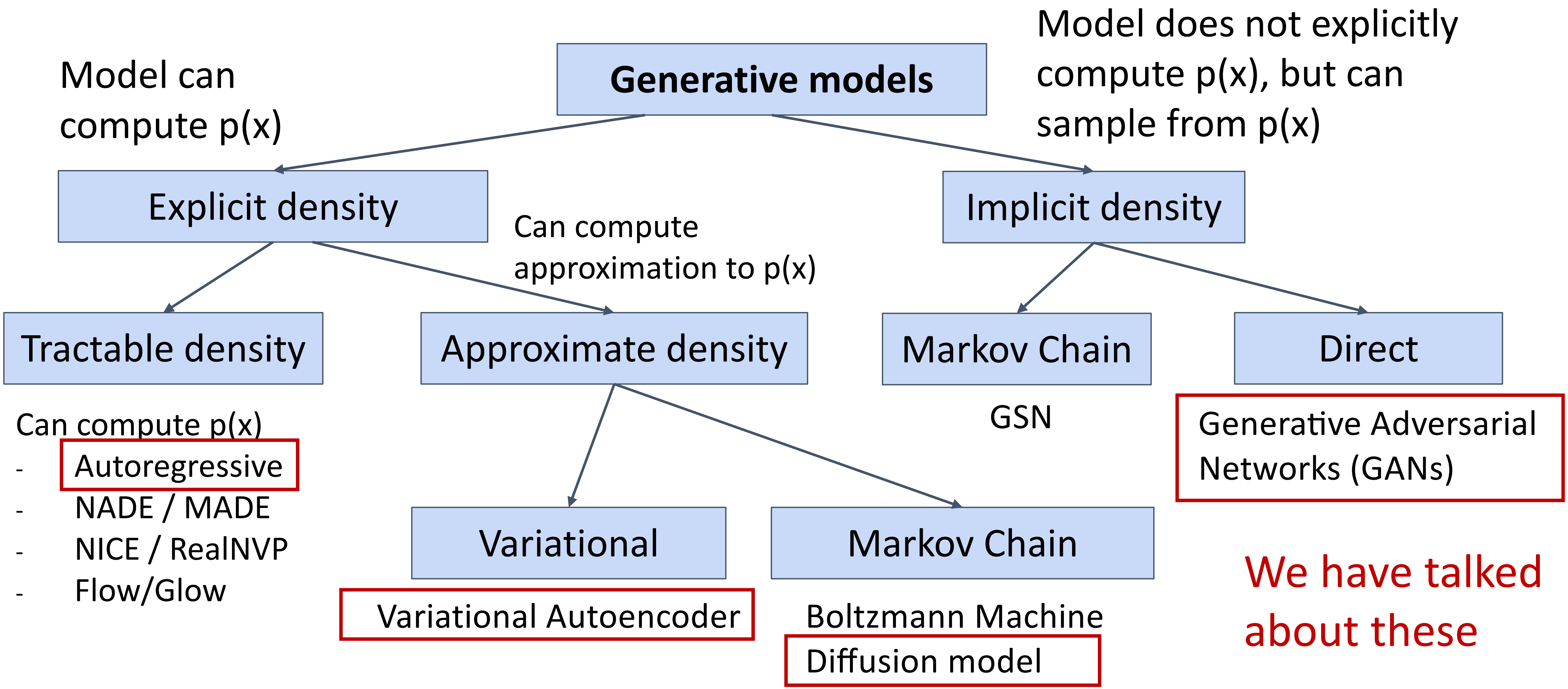


Figure adapted from Ian Goodfellow, Tutorial on Generative Adversarial Networks, 2017.

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

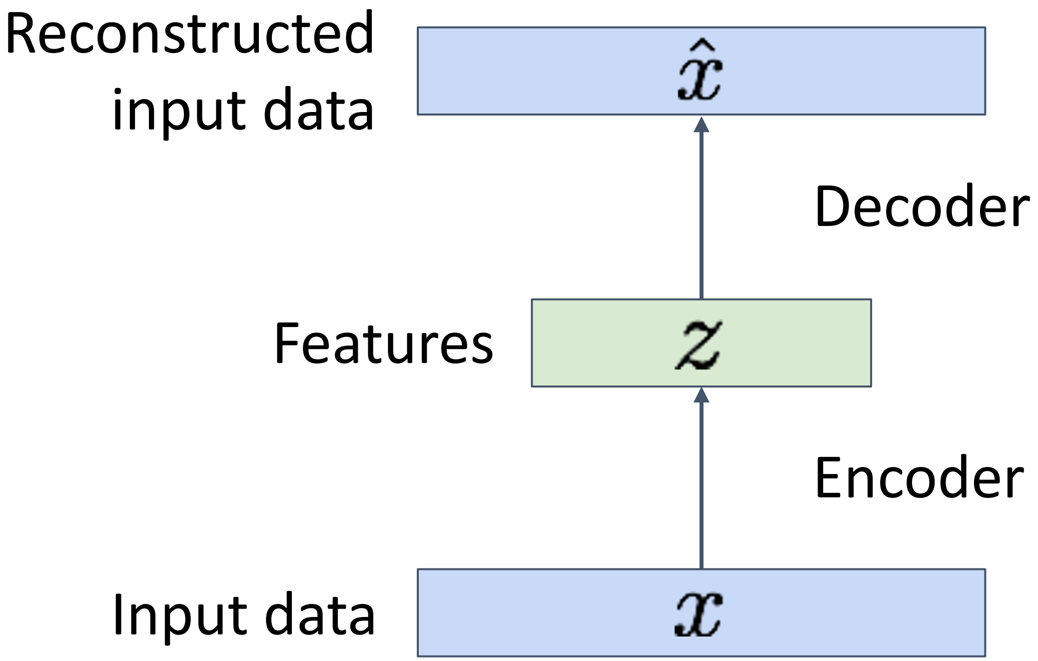
- Van der Oord et al. NIPS 2016
- Salimans et al. 2017 (PixelCNN++)

(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



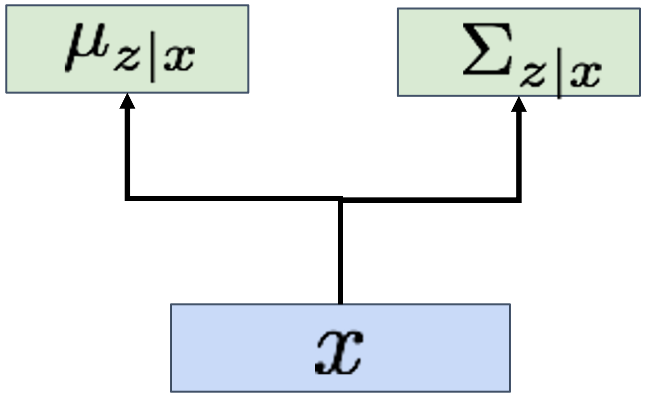
Variational Autoencoders

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

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

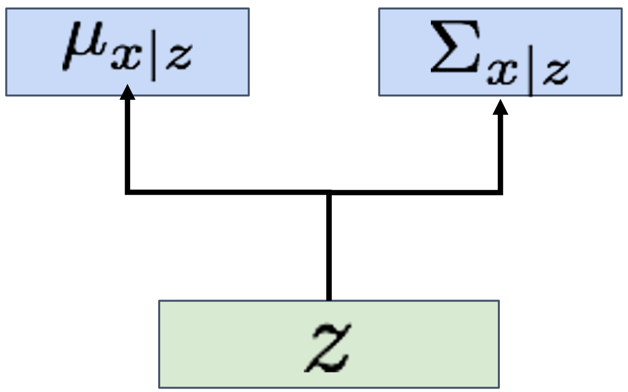
Encoder Network

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



Decoder Network

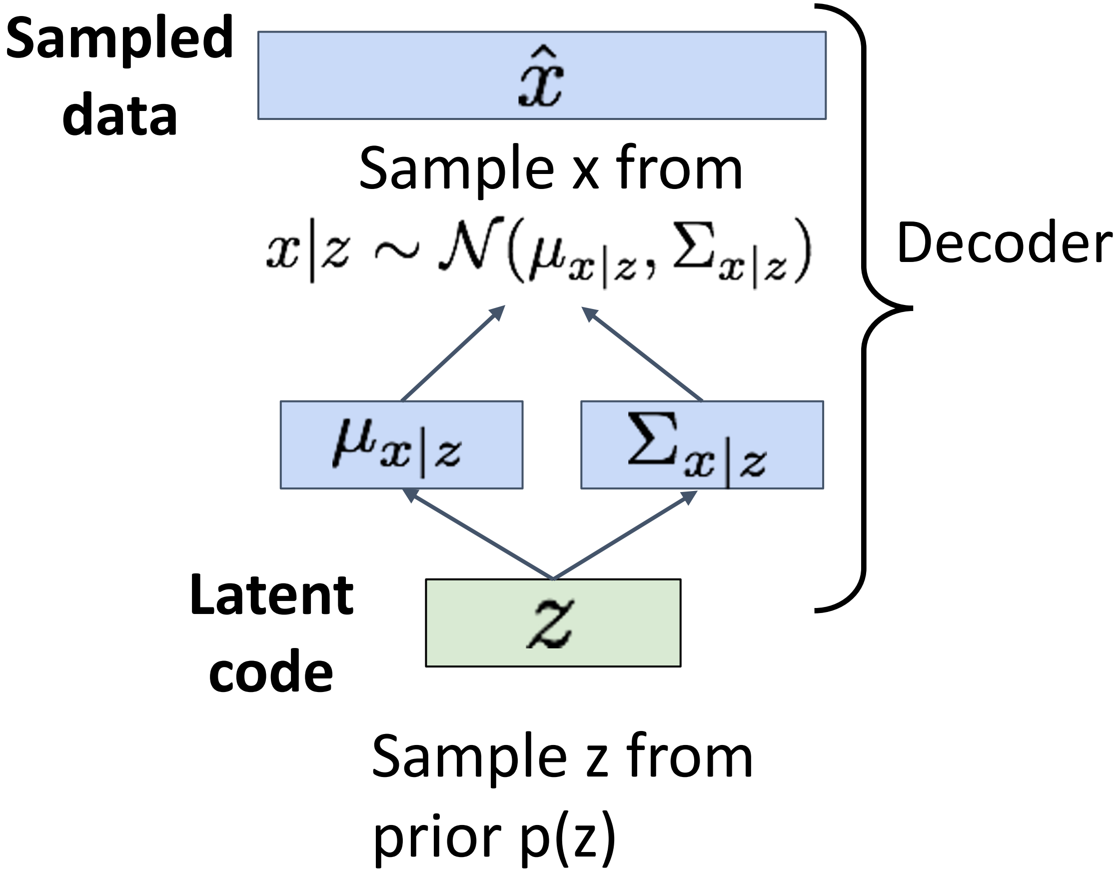
$$p_\theta(x \mid z) = N(\mu_{x|z}, \Sigma_{x|z})$$



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

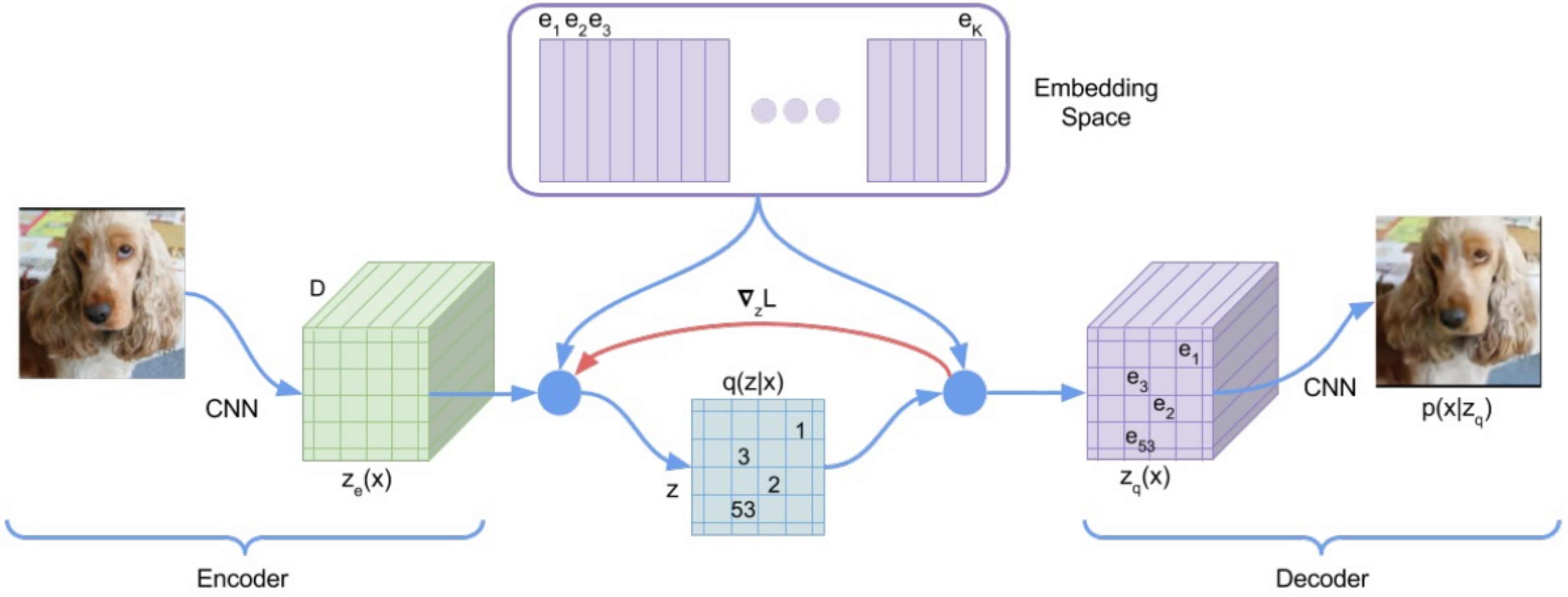
- Directly maximize p(data)
- High-quality generated images
- Slow to generate images
- No explicit latent codes

Variational models

- Maximize lower-bound on p(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

https://arxiv.org/pdf/1711.00937.pdf NeurIPS’17

Generative Adversarial Networks

Setup: Assume we have data x_i drawn from distribution $p_{data}(x)$. 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 $p_G = p_{data}$!

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

Generator Network

Generated Sample

Discriminator Network

Sample z from p_z

z

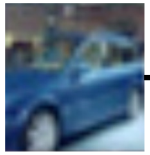
G

D

Fake

Real

Train **Generator Network** G to convert z into fake data x sampled from p_G **by "fooling" the discriminator D**



Real Sample

Train **Discriminator Network** D to classify data as real or fake (1/0)

Goodfellow et al, "Generative Adversarial Nets", NeurIPS 2014

Generative Adversarial Networks: Training Objective

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

Train G and D using alternating gradient updates

$$\min_{G} \max_{D} \left(E_{x \sim p_{data}} [\log D(x)] + E_{z \sim p(z)} \left[\log \left(1 - D(G(z)) \right) \right] \right)$$

$$= \min_{G} \max_{D} V(G, D)$$

For t in 1, … T:

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

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

Goodfellow et al, “Generative Adversarial Nets”, NeurIPS 2014

Example code of training GAN

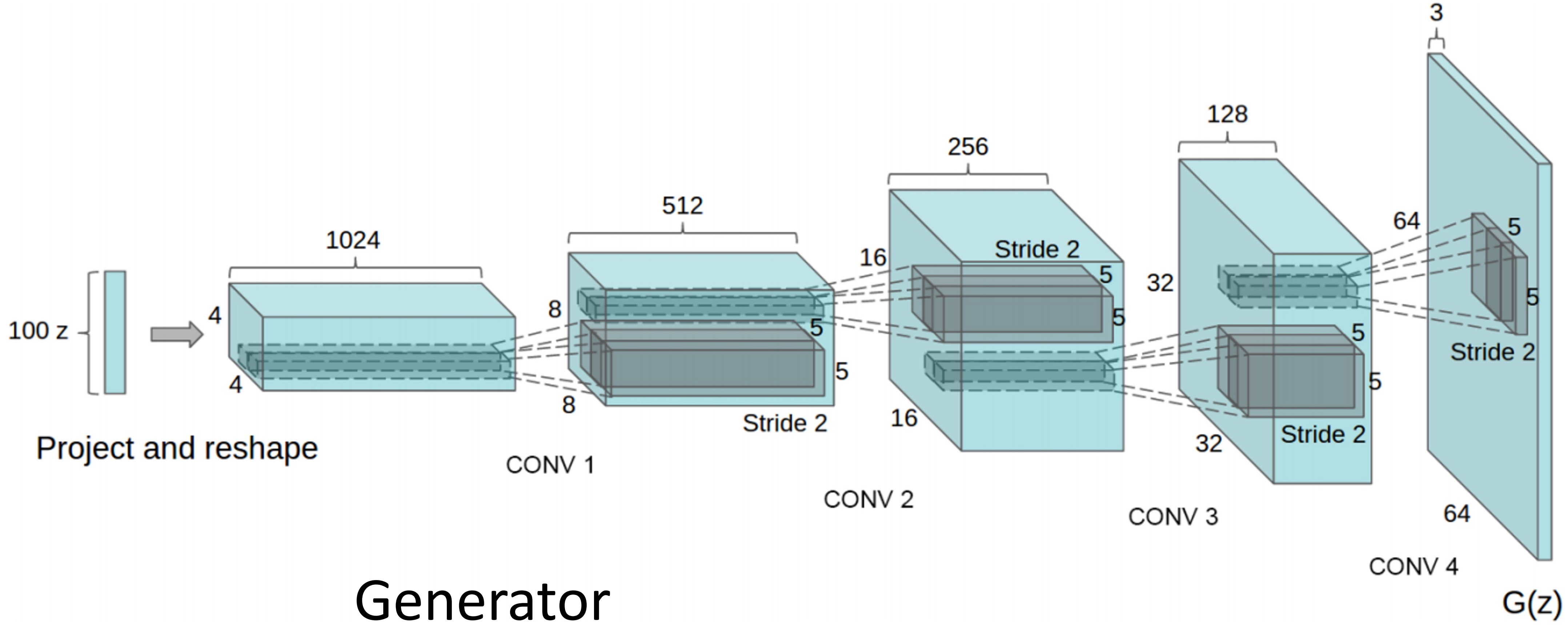
```
for epoch in range(opt.niter):
    for i, data in enumerate(dataloader, 0):
        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
        # train with real
        netD.zero_grad()
        real_cpu = data[0].to(device)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

https://github.com/pytorch/examples/blob/master/dcgan/main.py

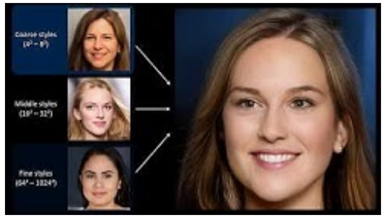
Generative Adversarial Networks: DC-GAN

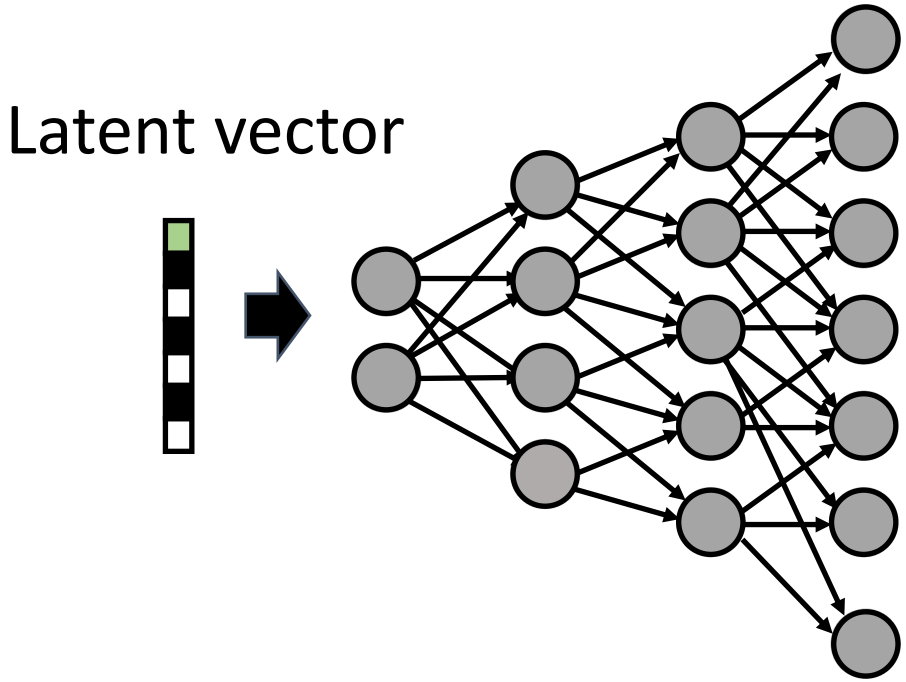


Generator

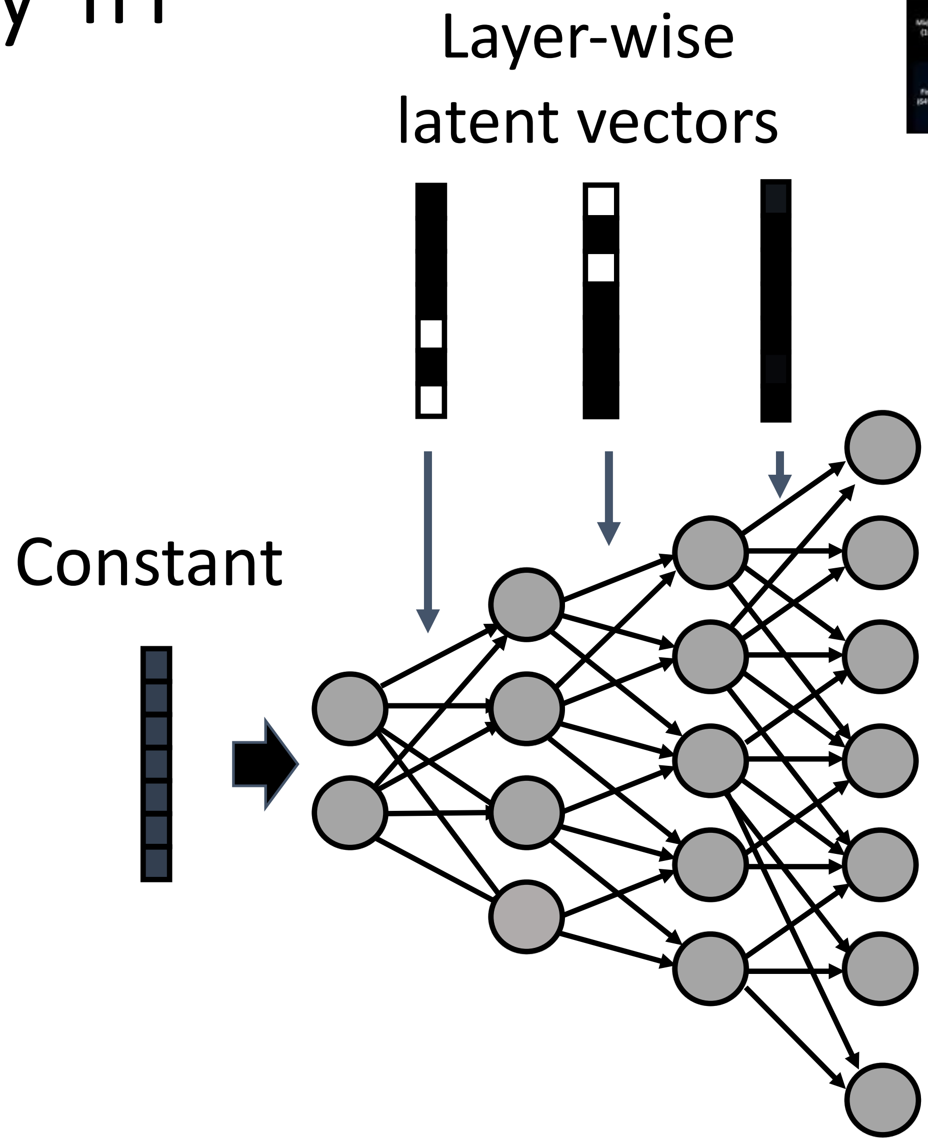
Radford et al, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks", ICLR 2016

Layer-wise Stochasity in StyleGAN





DC-GAN, PG-GAN



StyleGAN, StyleGANv2 [Karras et al]

GAN Improvements: StyleGAN for Higher Resolution

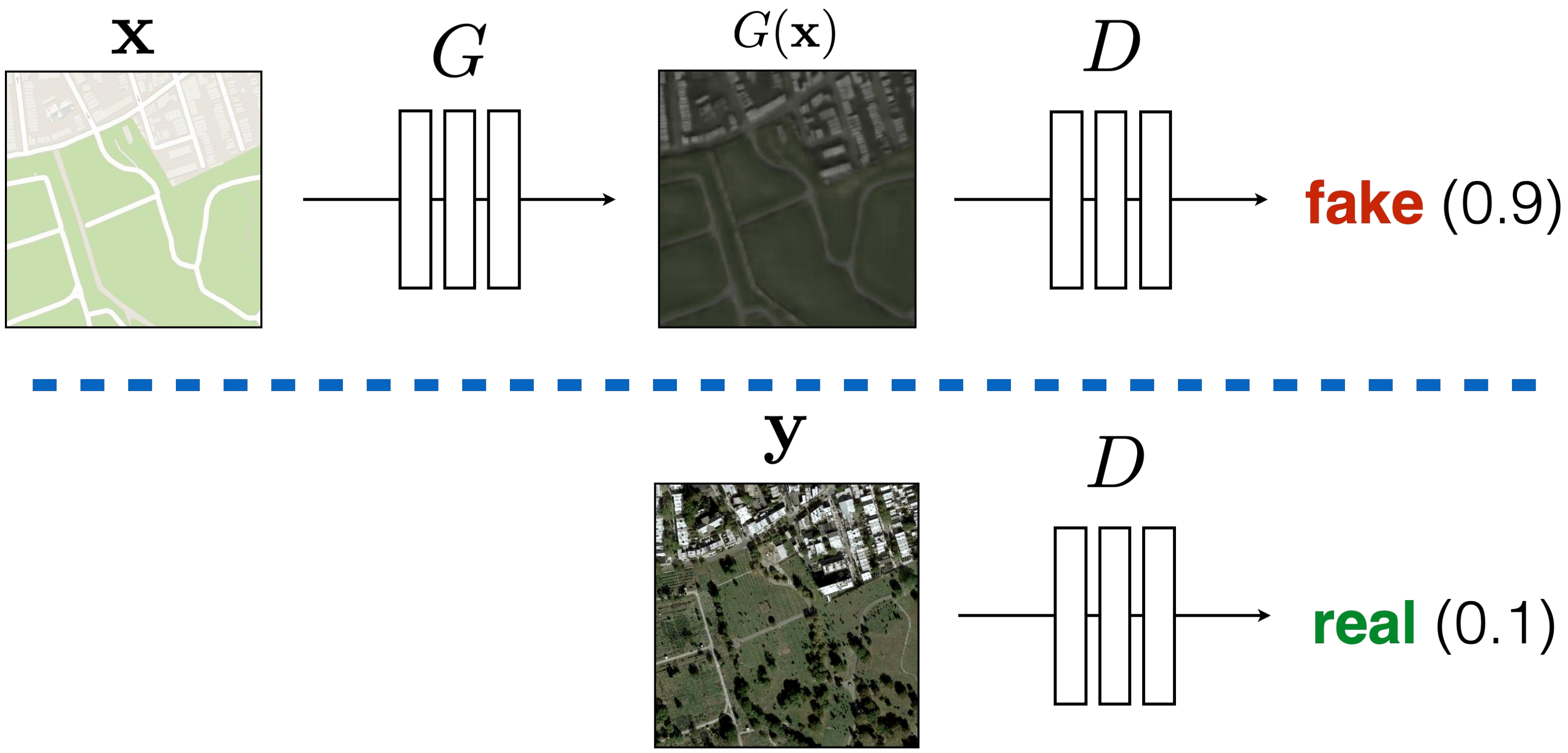
512 x 384 cars

1024 x 1024 faces

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

Image-to-Image Translation: Pix2Pix

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

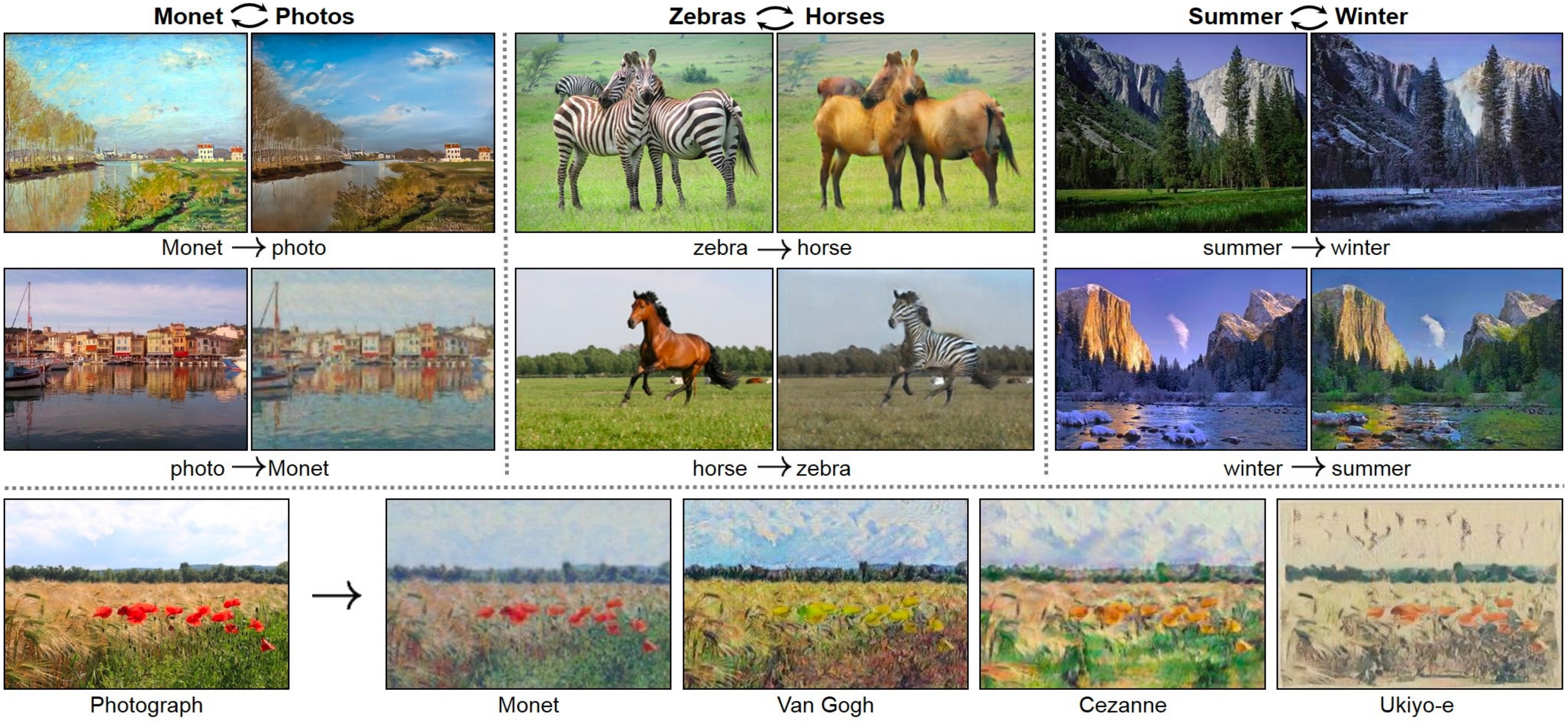


Philip Isola at MIT

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

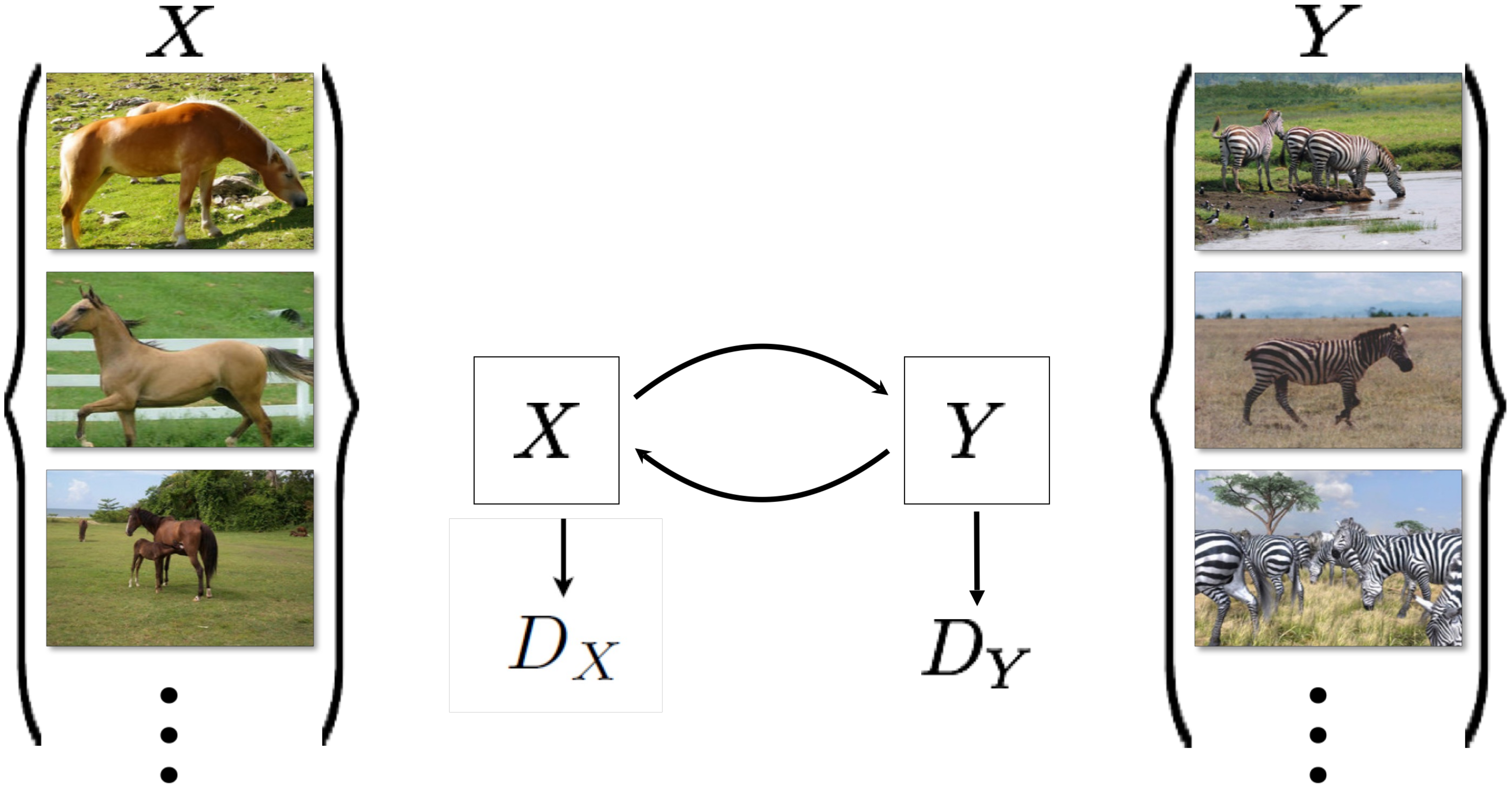
Isola et al, “Image-to-Image Translation with Conditional Adversarial Nets”, CVPR 2017

Unpaired Image-to-Image Translation: CycleGAN

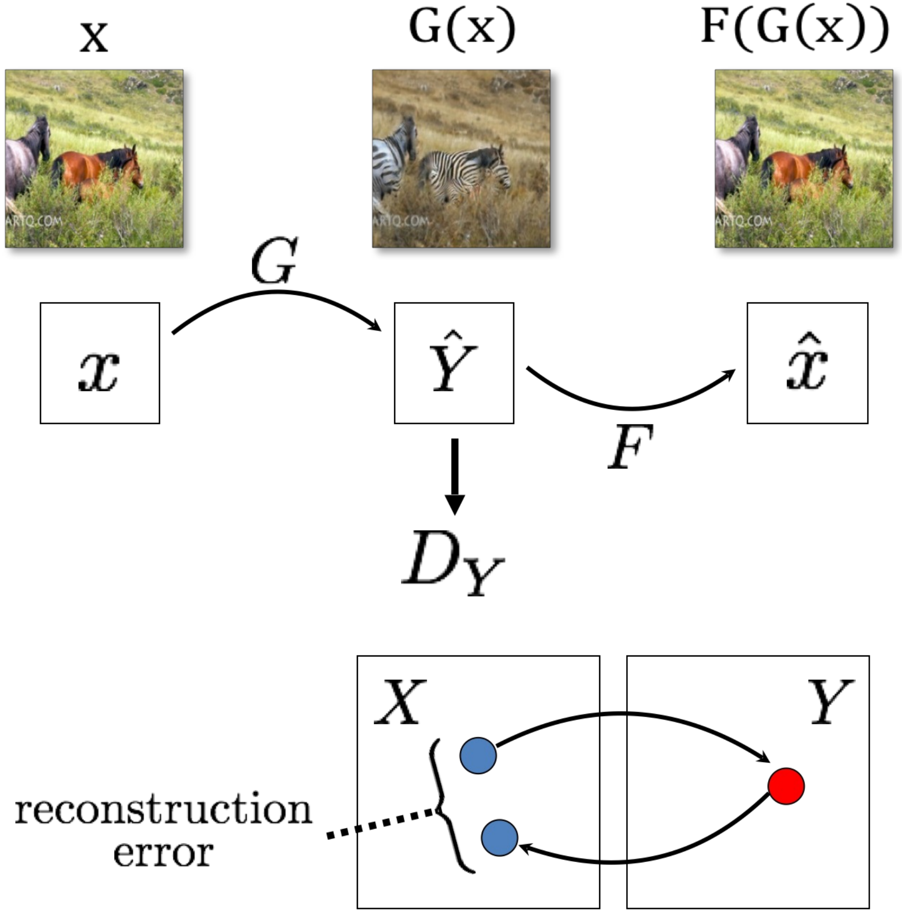


Zhu et al, “Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks”, ICCV 2017

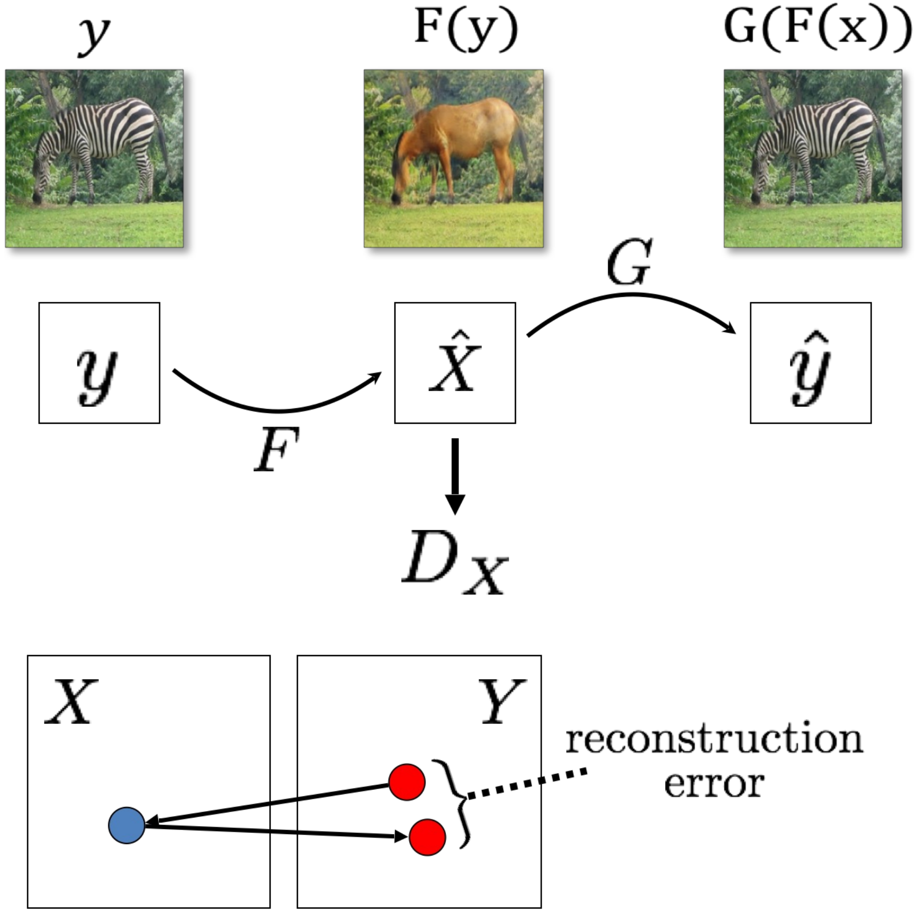
Unpaired Image-to-Image Translation: CycleGAN



Cycle Consistency Loss

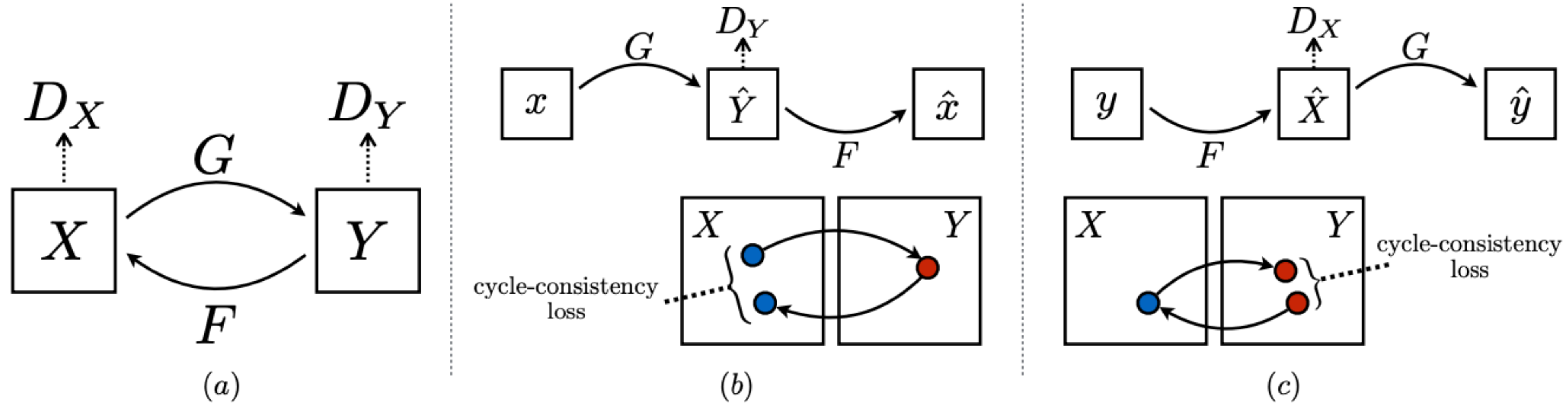


$$\left\|F(G(x)) - x\right\|_1$$



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

Cycle Consistency Loss: full objective

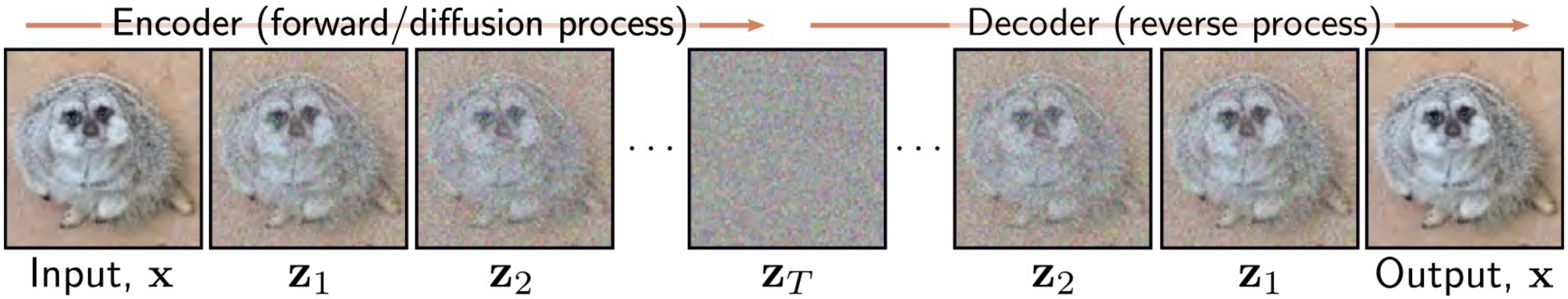


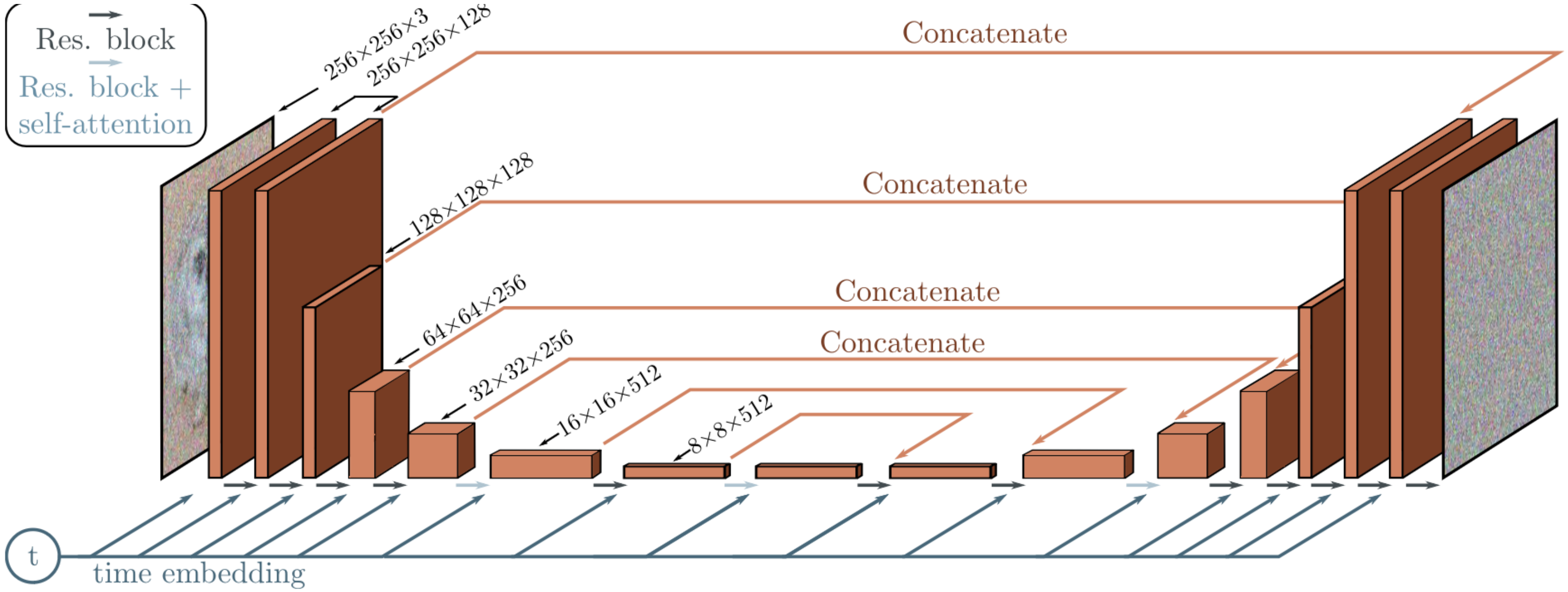
$$\mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) = \mathbb{E}_{y \sim p_{\text{data}}(y)}[\log D_Y(y)] + \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log(1 - D_Y(G(x))],$$

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

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

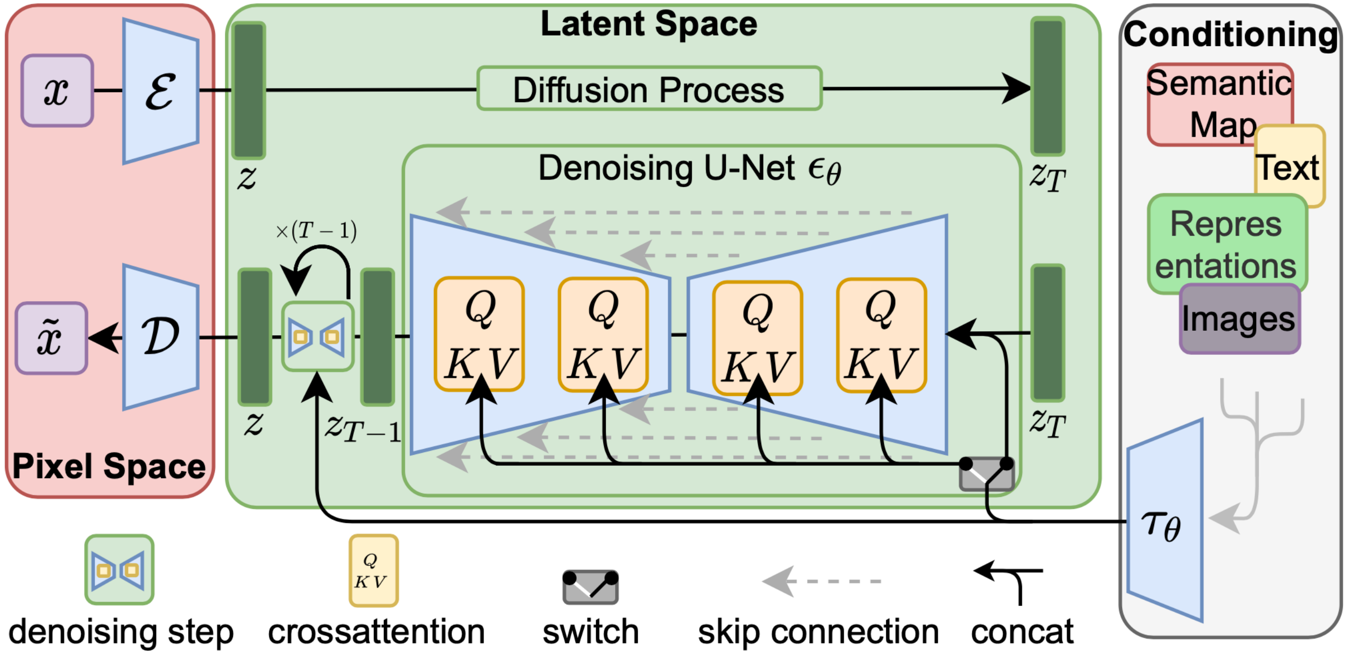
Diffusion Models: U-Net used in image generation





Latent Diffusion Model

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



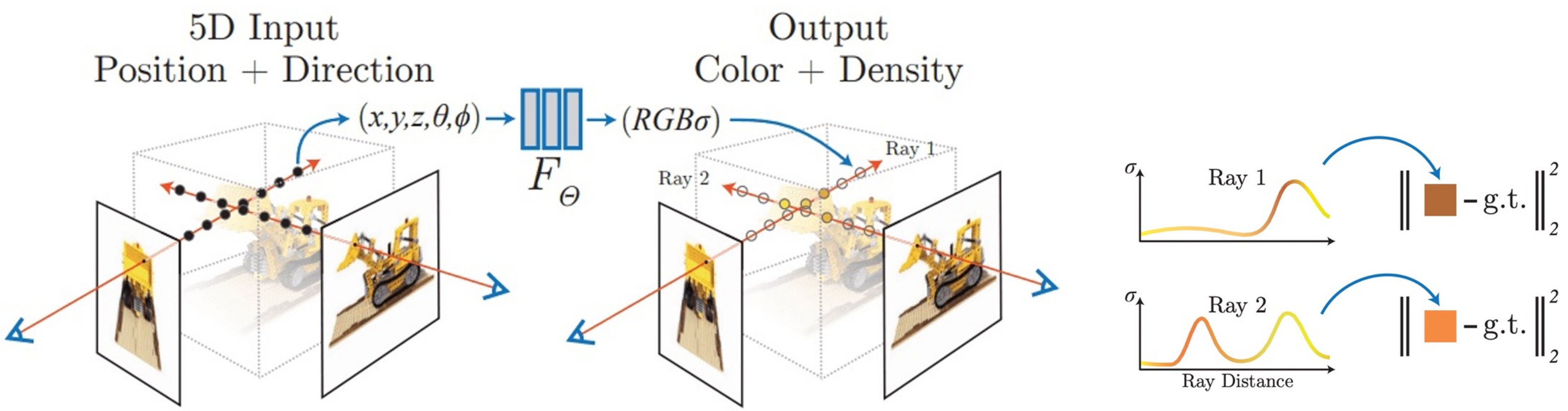
augmenting their underlying U-Net backbone with the cross-attention mechanism

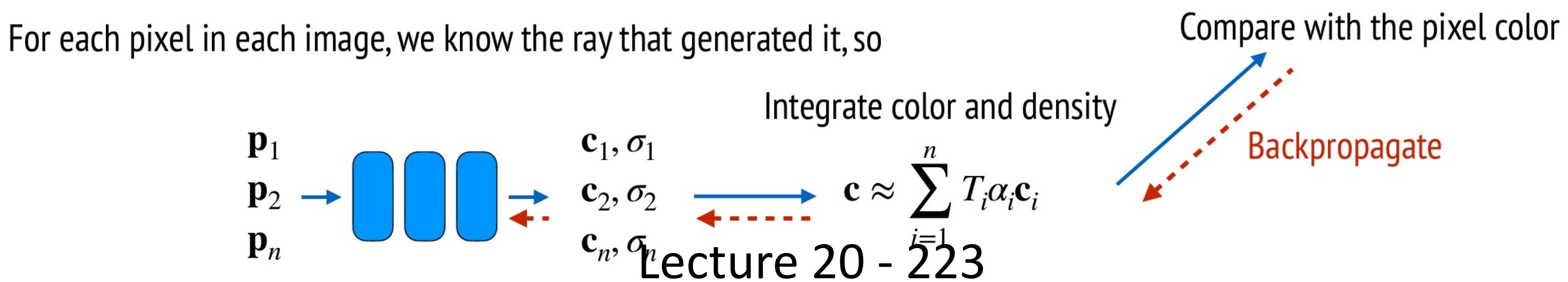
https://arxiv.org/pdf/2112.10752.pdf

Neural Rendering

What you need to know:

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





- Thank you for your participation and best luck to all of you!
- Consider to give us a good thumb-up through the course evaluation.

- Join the **UCLA Vision Seminar/Info Mail list** to receive announcement of future CV seminars or research opportunities: https://groups.google.com/a/lists.ucla.edu/g/vision-seminar or send an email to vision-seminar+subscribe@lists.ucla.edu

UCLA VAIL Vision and Autonomy Intelligence Lab

https://vail-ucla.github.io/

- Sign up link of applying to be undergraduate student researcher: https://forms.gle/Zcc9ABcoYp4oPE659
- Along with my graduate students, we will evaluate the applications, and we will invite the shortlisted students for interviews