

Lecture 2: Image Classification pipeline

Administrative: Piazza

For questions about midterm, poster session, projects, etc, use Piazza!

SCPD students: Use your @stanford.edu address to register for Piazza; contact scpd-customerservice@stanford.edu for help.

Administrative: Assignment 1

Out yesterday, due 4/18 11:59pm

- K-Nearest Neighbor
- Linear classifiers: SVM, Softmax
- Two-layer neural network
- Image features

Administrative: Friday Discussion Sections

Fridays 12:30pm - 1:20pm in Skilling Auditorium

Hands-on tutorials, with more practical detail than main lecture

Check course website for schedule:

<http://cs231n.stanford.edu/syllabus.html>

This Friday: Python / numpy / Google Cloud setup

Administrative: Python + Numpy

CS231n Convolutional Neural Networks for Visual Recognition

Python Numpy Tutorial

This tutorial was contributed by [Justin Johnson](#).

We will use the Python programming language for all assignments in this course. Python is a great general-purpose programming language on its own, but with the help of a few popular libraries (numpy, scipy, matplotlib) it becomes a powerful environment for scientific computing.

We expect that many of you will have some experience with Python and numpy; for the rest of you, this section will serve as a quick crash course both on the Python programming language and on the use of Python for scientific computing.

<http://cs231n.github.io/python-numpy-tutorial/>

Administrative: Google Cloud

We will be using Google Cloud in this class

We will be distributing coupons to all enrolled students

See our tutorial here for walking through Google Cloud setup:

<http://cs231n.github.io/gce-tutorial/>

Image Classification: A core task in Computer Vision



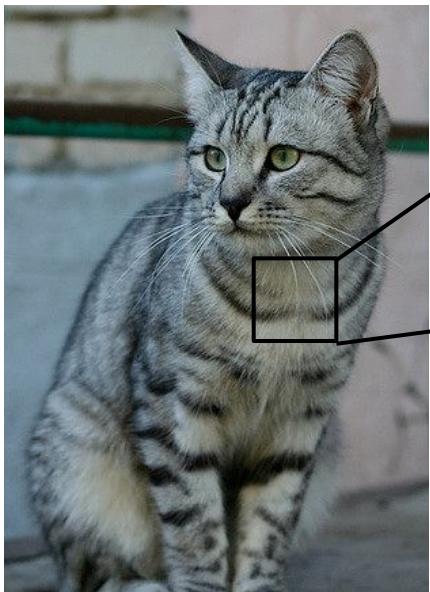
This image by [Nikita](#) is
licensed under [CC-BY 2.0](#).

(assume given set of discrete labels)
{dog, cat, truck, plane, ...}



cat

The Problem: Semantic Gap



[105 112 108 111 104 99 106 99 96 103 112 119 104 97 93 87]
[91 98 102 106 104 79 98 103 99 105 123 136 110 105 94 85]
[76 85 90 105 128 105 87 96 95 99 115 112 106 103 99 85]
[99 81 93 120 131 127 100 95 98 102 99 96 93 101 94]
[106 91 61 64 69 91 88 85 101 107 109 98 75 84 96 95]
[114 108 85 55 55 69 64 54 64 87 112 129 98 74 84 91]
[133 137 147 103 65 81 80 65 52 54 74 84 102 93 85 82]
[128 137 144 140 109 95 86 70 62 65 63 63 60 73 86 101]
[125 133 148 137 119 121 117 94 65 79 80 65 54 64 72 98]
[127 125 131 147 133 127 126 131 111 96 89 75 61 64 72 84]
[115 114 109 123 150 148 131 118 113 109 100 92 74 65 65 72 78]
[89 93 90 97 108 147 131 118 113 114 113 109 106 95 77 80]
[63 77 86 81 77 79 102 123 117 115 117 125 125 130 115 87]
[62 65 82 89 78 71 80 101 124 126 119 101 107 114 131 119]
[63 65 75 88 89 71 62 81 128 138 135 105 81 98 110 118]
[87 65 71 87 106 95 69 45 76 130 126 107 92 94 105 112]
[118 97 82 86 117 123 116 66 41 51 95 93 89 95 102 107]
[164 146 112 88 82 120 124 104 76 48 45 66 88 101 102 109]
[157 170 157 128 93 86 114 132 112 97 69 55 70 62 99 94]
[130 128 134 161 139 100 109 118 121 134 114 87 65 53 69 86]
[128 112 96 117 150 144 128 115 104 107 102 93 87 81 72 79]
[123 107 96 86 83 112 153 149 122 109 104 75 80 107 112 99]
[122 121 102 88 82 86 94 117 145 148 153 102 58 78 92 107]
[122 164 148 103 71 56 78 83 93 103 119 139 102 61 69 84]]

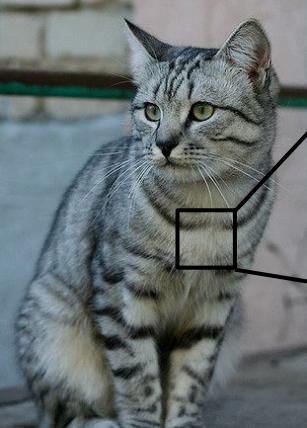
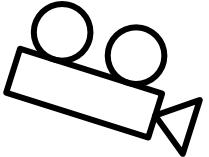
What the computer sees

An image is just a big grid of numbers between [0, 255]:

e.g. 800 x 600 x 3
(3 channels RGB)

This image by [Nikita](#) is
licensed under [CC-BY 2.0](#)

Challenges: Viewpoint variation



```
[1185 112 188 111 184 99 186 99 96 183 112 119 184 97 93 87]  
[ 91 98 182 106 184 79 98 183 99 185 123 136 118 185 94 85]  
[ 76 85 98 185 128 105 87 96 95 99 115 112 106 183 99 85]  
[ 99 81 98 185 128 105 127 96 98 101 115 112 106 183 99 85]  
[104 91 86 84 69 91 68 85 101 115 112 106 183 99 85]  
[114 108 85 55 55 69 64 54 64 87 112 129 98 74 84 96 95]  
[133 137 147 103 65 81 80 65 52 54 74 84 102 93 85 82]  
[128 137 144 148 105 95 86 78 62 65 63 63 68 73 86 101]  
[102 125 131 147 133 127 116 131 111 98 89 75 61 64 72 84]  
[127 125 131 147 133 127 116 131 111 98 89 75 61 64 72 84]  
[115 115 189 123 150 148 131 118 113 109 108 92 74 65 72 78]  
[ 89 93 98 97 108 147 131 118 113 114 113 108 106 95 77 80]  
[ 63 77 86 81 77 79 182 123 137 115 111 125 125 130 115 87]  
[ 62 85 88 89 73 62 81 128 138 135 105 81 98 118 118]  
[ 63 65 75 88 89 73 62 81 128 138 135 105 81 98 118 118]  
[ 87 65 71 87 100 95 69 45 76 138 126 107 92 94 105 112]  
[118 97 82 86 117 123 116 66 41 51 95 93 89 89 95 102 107]  
[164 149 112 88 100 108 128 184 78 48 66 66 66 101 102 108]  
[157 98 100 118 93 86 104 128 184 78 48 66 66 66 101 102 108]  
[138 128 134 161 139 180 109 118 121 134 114 87 65 53 69 86]  
[128 112 96 117 150 144 120 115 104 107 102 93 87 81 72 79]  
[123 107 96 86 83 112 153 149 122 189 104 75 88 107 112 99]  
[122 121 102 88 82 86 94 117 145 148 153 105 58 78 92 107]  
[122 164 148 103 71 56 78 83 93 103 119 139 102 61 69 84]
```

All pixels change when
the camera moves!

[This image](#) by [Nikita](#) is
licensed under [CC-BY 2.0](#)

Challenges: Illumination



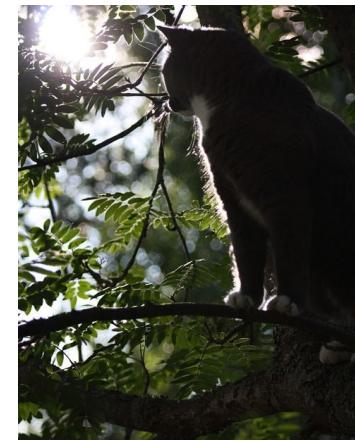
This image is [CC0 1.0 public domain](#)



This image is [CC0 1.0 public domain](#)

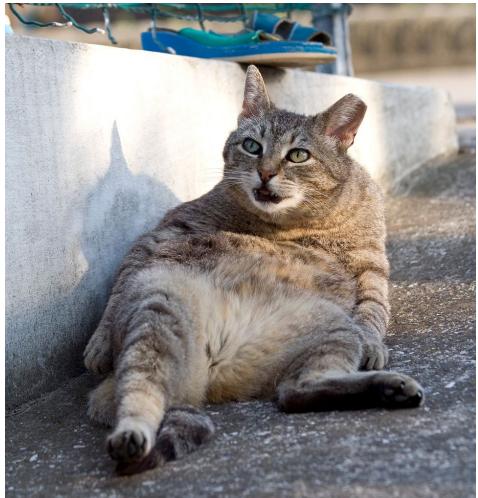


This image is [CC0 1.0 public domain](#)



This image is [CC0 1.0 public domain](#)

Challenges: Deformation



[This image by Umberto Salvagnin](#)
is licensed under [CC-BY 2.0](#)



[This image by Umberto Salvagnin](#)
is licensed under [CC-BY 2.0](#)



[This image by sare bear is](#)
licensed under [CC-BY 2.0](#)



[This image by Tom Thai is](#)
licensed under [CC-BY 2.0](#)

Challenges: Occlusion



[This image is CC0 1.0 public domain](#)



[This image is CC0 1.0 public domain](#)



[This image by jonsson is licensed under CC-BY 2.0](#)

Challenges: Background Clutter



[This image is CC0 1.0 public domain](#)



[This image is CC0 1.0 public domain](#)

Challenges: Intraclass variation



[This image is CC0 1.0 public domain](#)

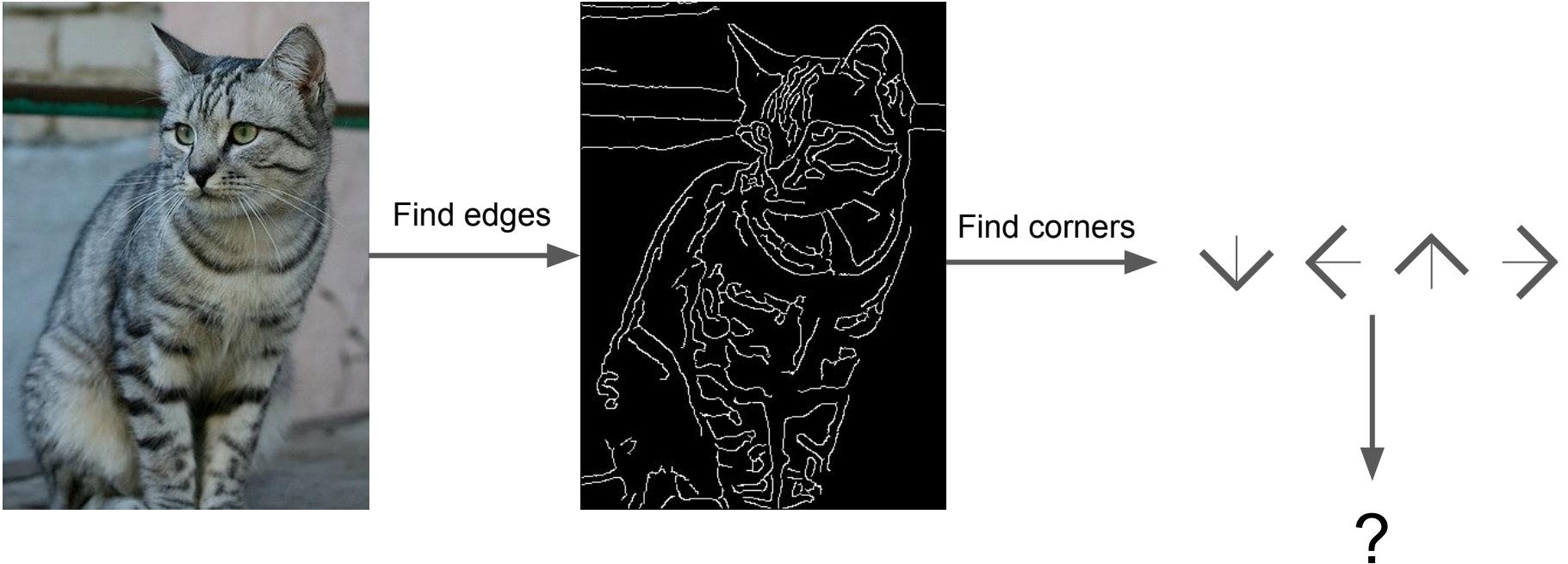
An image classifier

```
def classify_image(image):  
    # Some magic here?  
    return class_label
```

Unlike e.g. sorting a list of numbers,

no obvious way to hard-code the algorithm for
recognizing a cat, or other classes.

Attempts have been made



John Canny, "A Computational Approach to Edge Detection", IEEE TPAMI 1986

Machine Learning: Data-Driven Approach

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

Example training set

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

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

airplane



automobile



bird



cat



deer



First classifier: Nearest Neighbor

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



Memorize all
data and labels

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



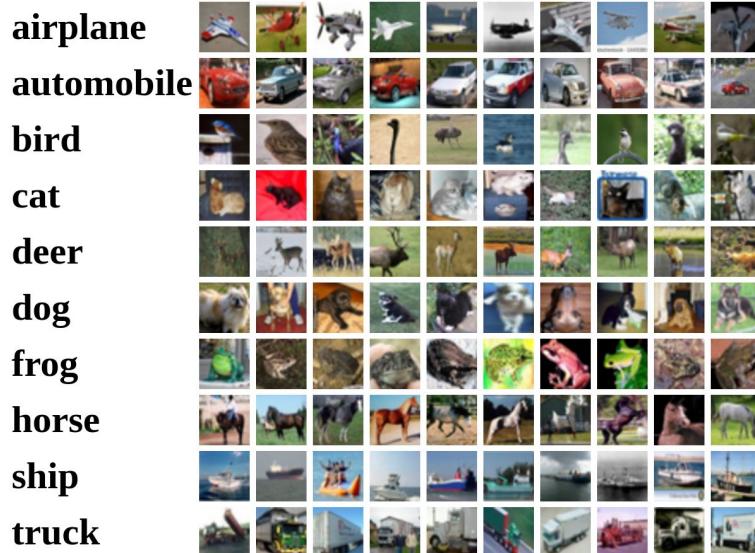
Predict the label
of the most similar
training image

Example Dataset: CIFAR10

10 classes

50,000 training images

10,000 testing images



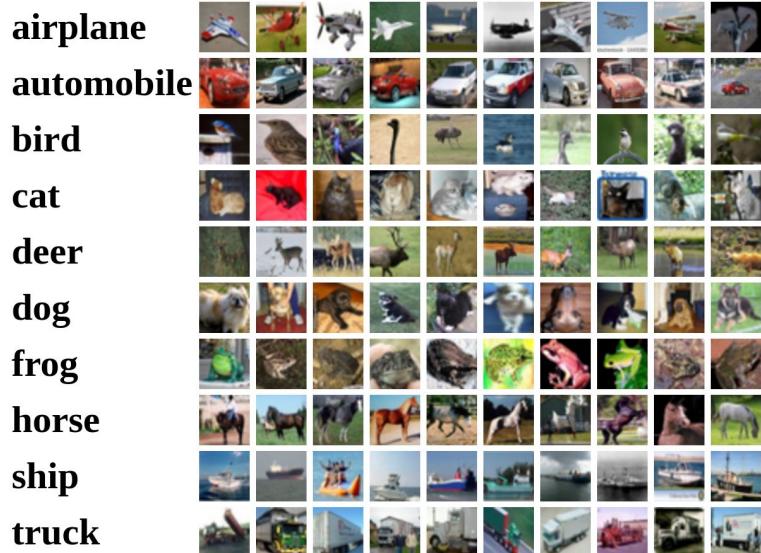
Alex Krizhevsky, "Learning Multiple Layers of Features from Tiny Images", Technical Report, 2009.

Example Dataset: CIFAR10

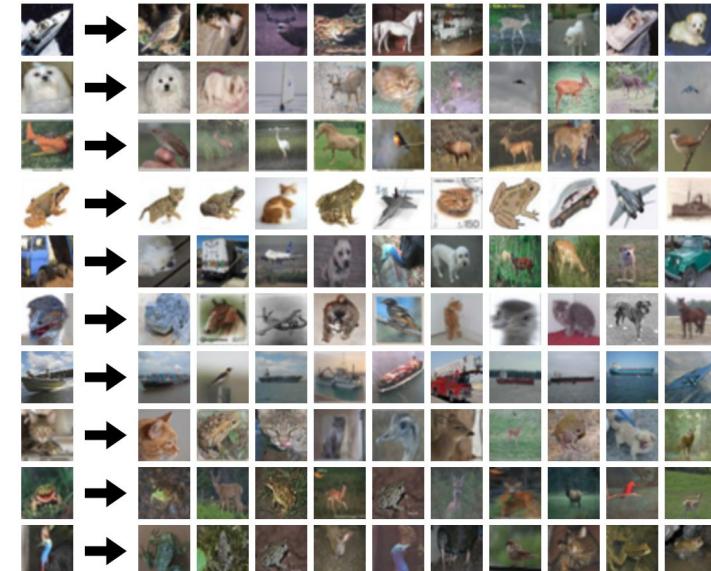
10 classes

50,000 training images

10,000 testing images



Test images and nearest neighbors



Alex Krizhevsky, "Learning Multiple Layers of Features from Tiny Images", Technical Report, 2009.

Distance Metric to compare images

L1 distance:

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

test image			
56	32	10	18
90	23	128	133
24	26	178	200
2	0	255	220

training image			
10	20	24	17
8	10	89	100
12	16	178	170
4	32	233	112

pixel-wise absolute value differences

46	12	14	1
82	13	39	33
12	10	0	30
2	32	22	108

add
→ 456

```

import numpy as np

class NearestNeighbor:
    def __init__(self):
        pass

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

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

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

        return Ypred

```

Nearest Neighbor classifier

```

import numpy as np

class NearestNeighbor:
    def __init__(self):
        pass

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

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

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

        return Ypred

```

Nearest Neighbor classifier

Memorize training data

```

import numpy as np

class NearestNeighbor:
    def __init__(self):
        pass

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

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

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

        return Ypred

```

Nearest Neighbor classifier

For each test image:
 Find closest train image
 Predict label of nearest image

```

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

Q: With N examples,
how fast are training
and prediction?

```

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

Q: With N examples,
how fast are training
and prediction?

A: Train O(1),
predict O(N)

```

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

```

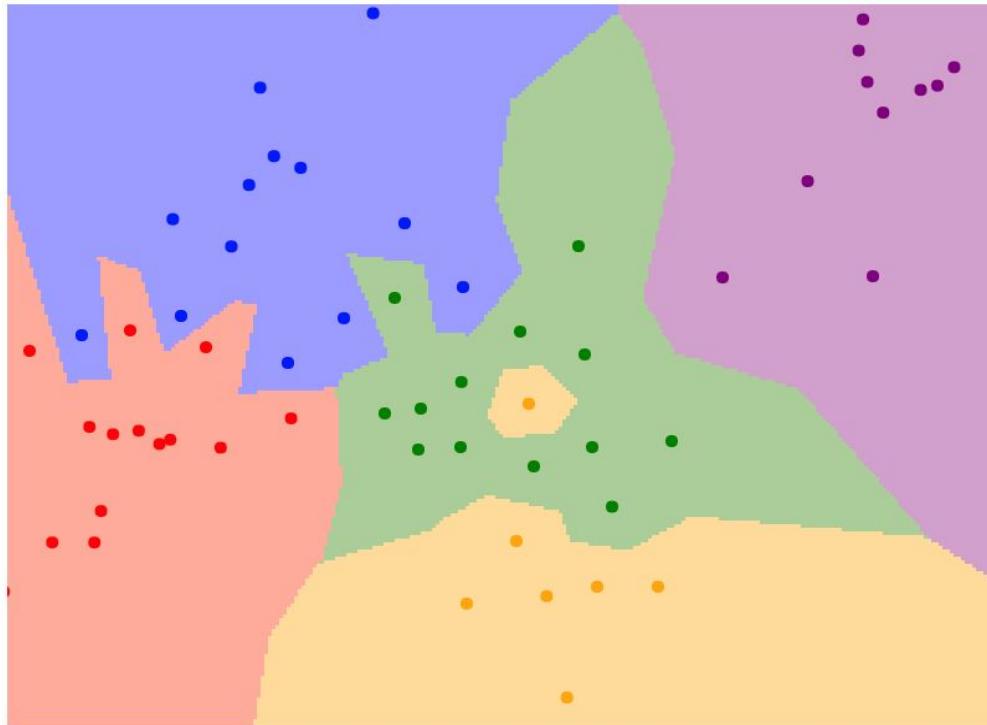
Nearest Neighbor classifier

Q: With N examples, how fast are training and prediction?

A: Train O(1), predict O(N)

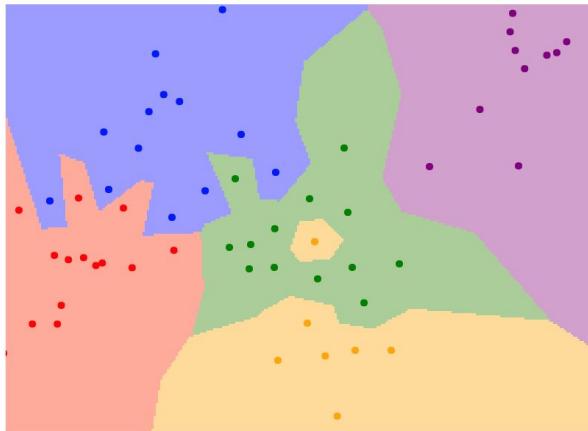
This is bad: we want classifiers that are **fast** at prediction; **slow** for training is ok

What does this look like?

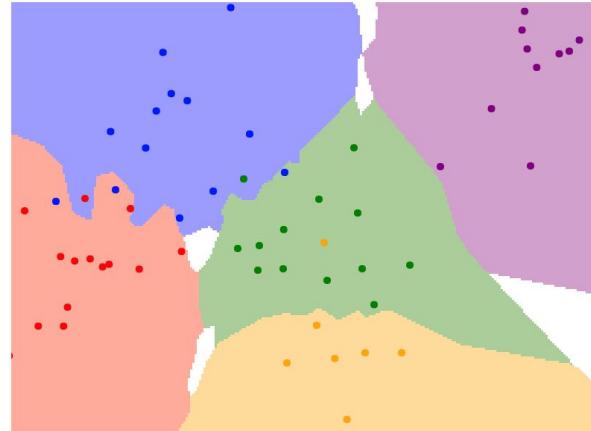


K-Nearest Neighbors

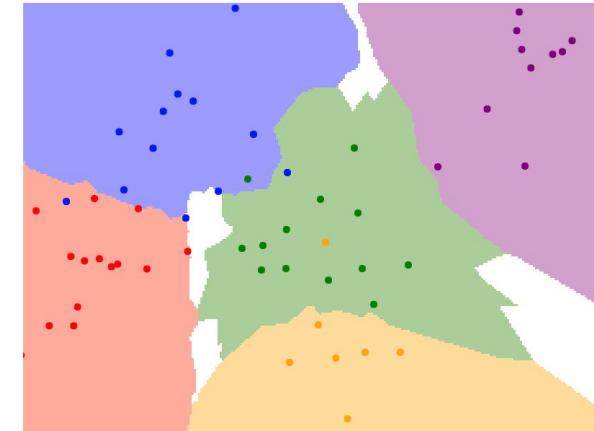
Instead of copying label from nearest neighbor,
take **majority vote** from K closest points



$K = 1$



$K = 3$



$K = 5$

What does this look like?



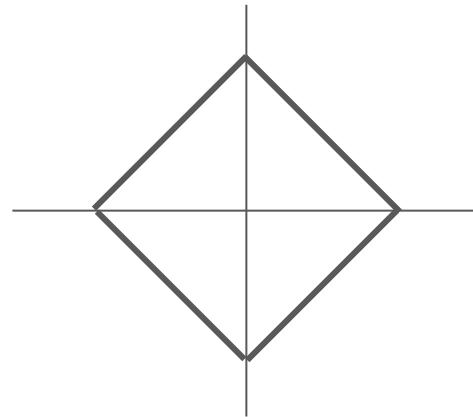
What does this look like?



K-Nearest Neighbors: Distance Metric

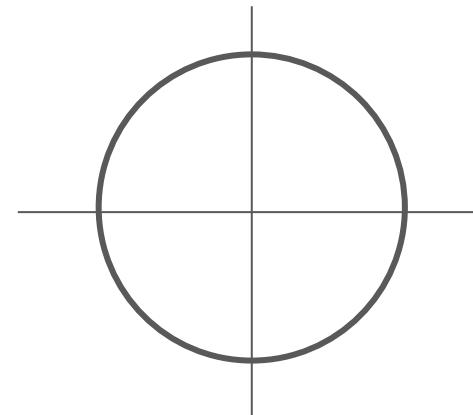
L1 (Manhattan) distance

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



L2 (Euclidean) distance

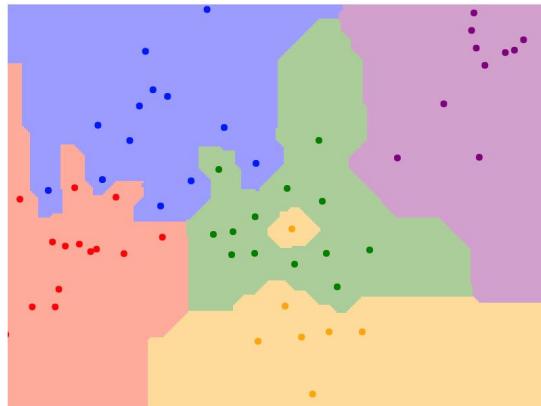
$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$



K-Nearest Neighbors: Distance Metric

L1 (Manhattan) distance

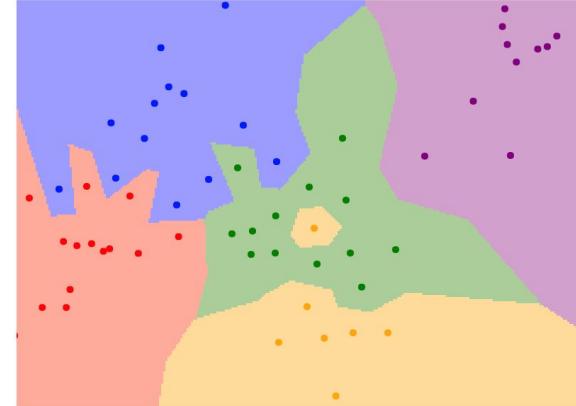
$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$



$K = 1$

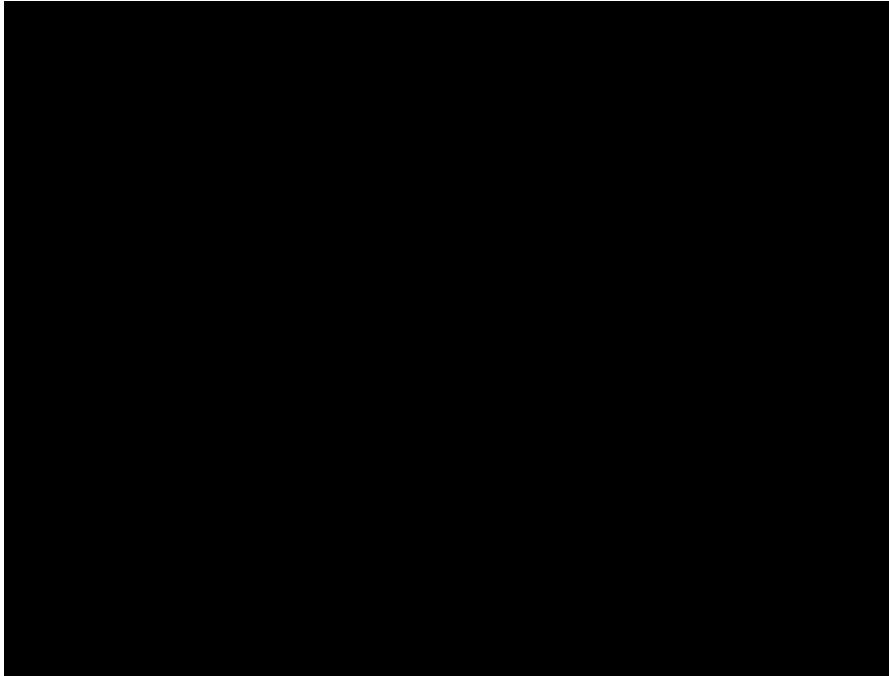
L2 (Euclidean) distance

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$



$K = 1$

K-Nearest Neighbors: Demo Time



<http://vision.stanford.edu/teaching/cs231n-demos/knn/>

Hyperparameters

What is the best value of k to use?

What is the best **distance** to use?

These are **hyperparameters**: choices about the algorithm that we set rather than learn

Hyperparameters

What is the best value of k to use?

What is the best **distance** to use?

These are **hyperparameters**: choices about the algorithm that we set rather than learn

Very problem-dependent.

Must try them all out and see what works best.

Setting Hyperparameters

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

Your Dataset

Setting Hyperparameters

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

BAD: $K = 1$ always works
perfectly on training data

Your Dataset

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

train

test

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

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

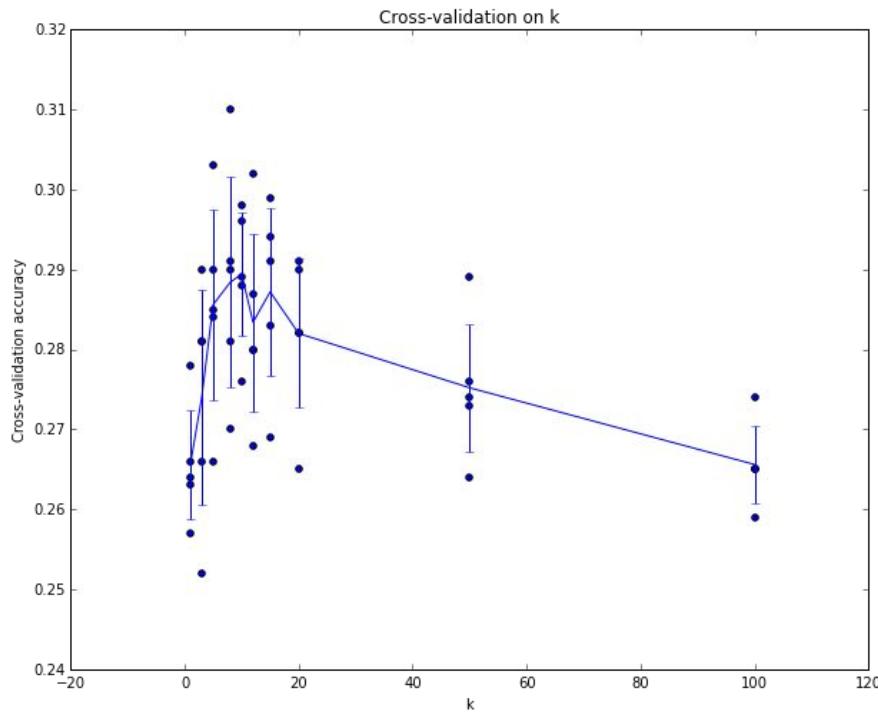
fold 1	fold 2	fold 3	fold 4	fold 5	test
--------	--------	--------	--------	--------	------

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 not used too frequently in deep learning

Setting Hyperparameters



Example of
5-fold cross-validation
for the value of **k**.

Each point: single
outcome.

The line goes
through the mean, bars
indicated standard
deviation

(Seems that $k \sim 7$ works best
for this data)

k-Nearest Neighbor on images **never used**.

- Very slow at test time
- Distance metrics on pixels are not informative

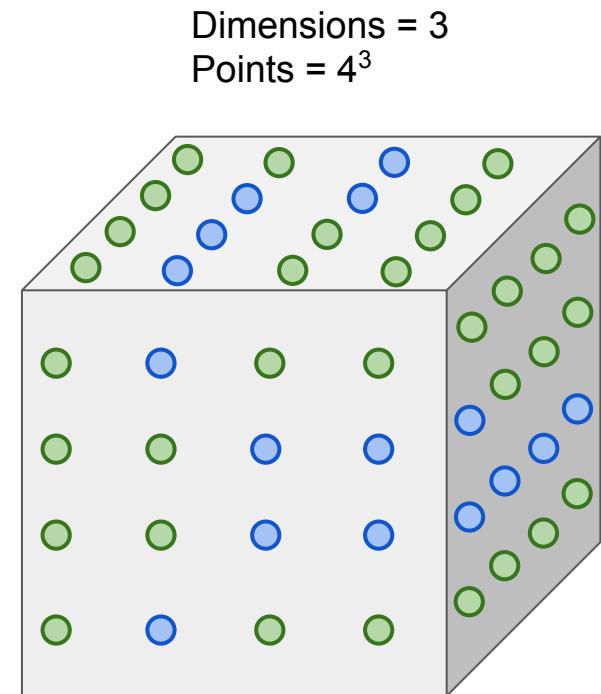
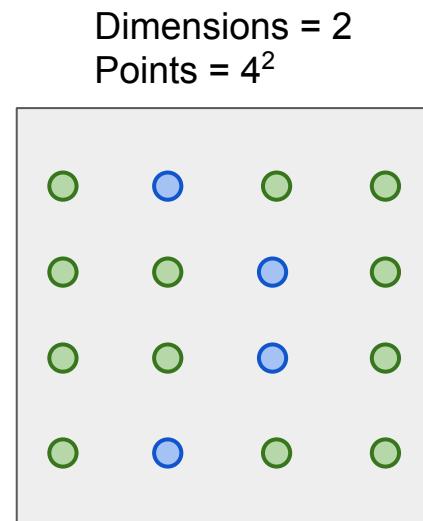
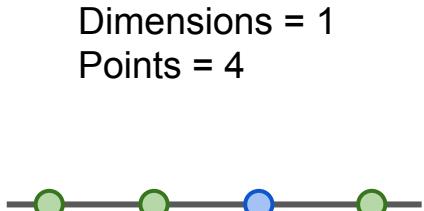


(all 3 images have same L2 distance to the one on the left)

Original image is
CC0 public domain

k-Nearest Neighbor on images **never** used.

- Curse of dimensionality



K-Nearest Neighbors: Summary

In **Image classification** we start with a **training set** of images and labels, and must predict labels on the **test set**

The **K-Nearest Neighbors** classifier predicts labels based on nearest training examples

Distance metric and K are **hyperparameters**

Choose hyperparameters using the **validation set**; only run on the test set once at the very end!

Linear Classification

Neural Network

Linear
classifiers



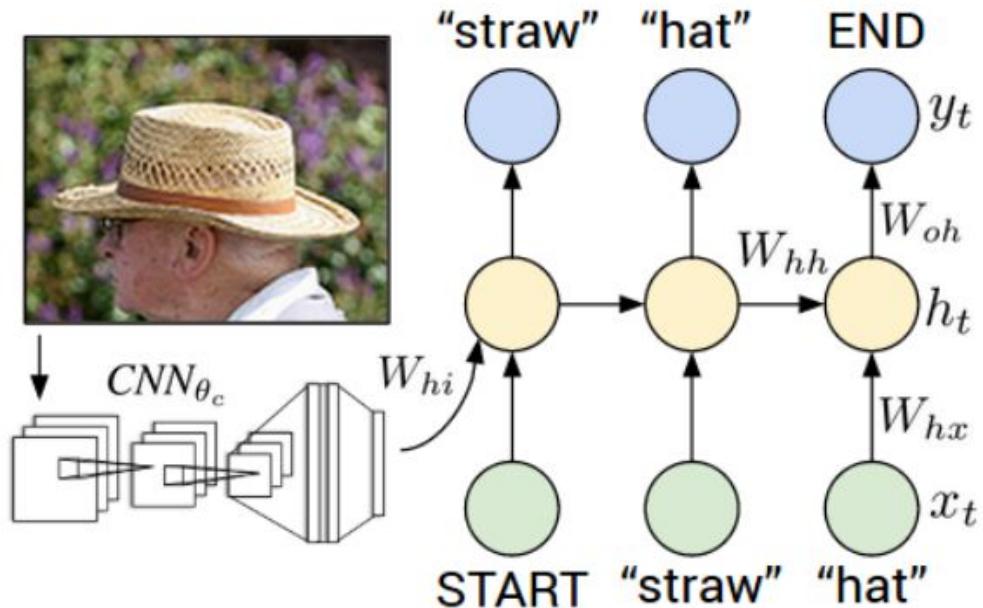
This image is CC0 1.0 public domain

Two young girls are playing with lego toy. *Boy is doing backflip on wakeboard*



Man in black shirt is playing guitar.

Construction worker in orange safety vest is working on road.



Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figures copyright IEEE, 2015. Reproduced for educational purposes.

Recall CIFAR10

airplane



automobile



bird



cat



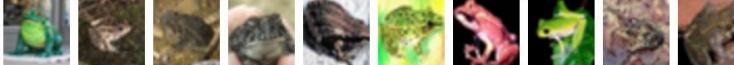
deer



dog



frog



horse



ship



truck



50,000 training images
each image is **32x32x3**

10,000 test images.

Parametric Approach

Image



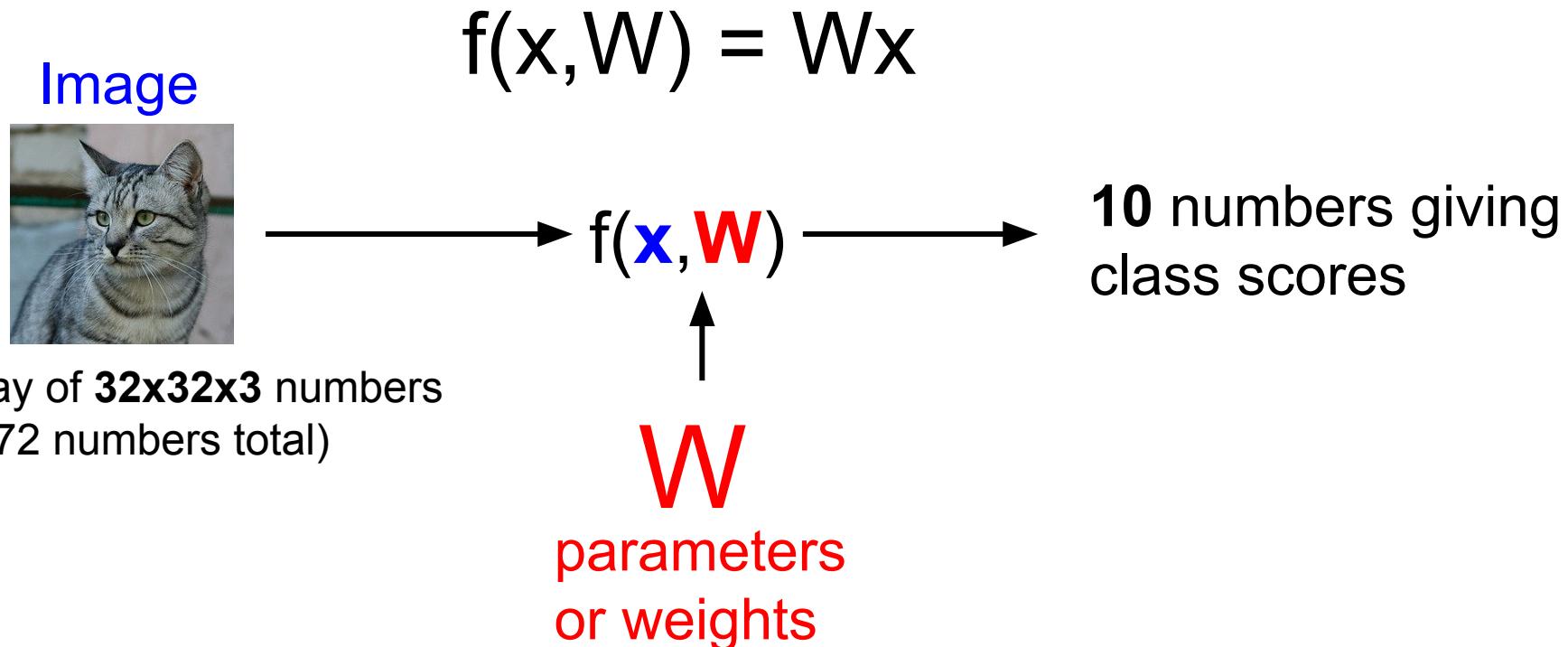
Array of **32x32x3** numbers
(3072 numbers total)

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

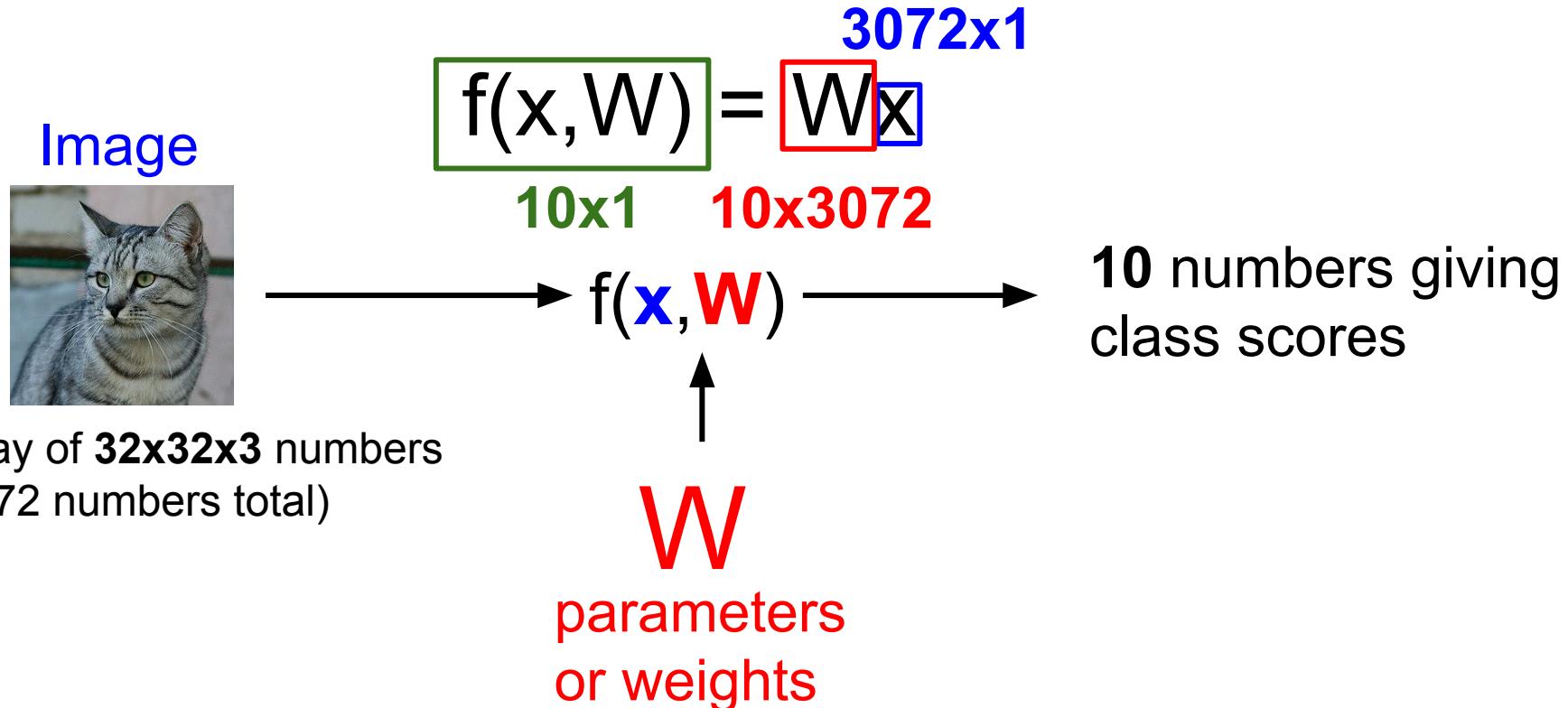
W
parameters
or weights

10 numbers giving
class scores

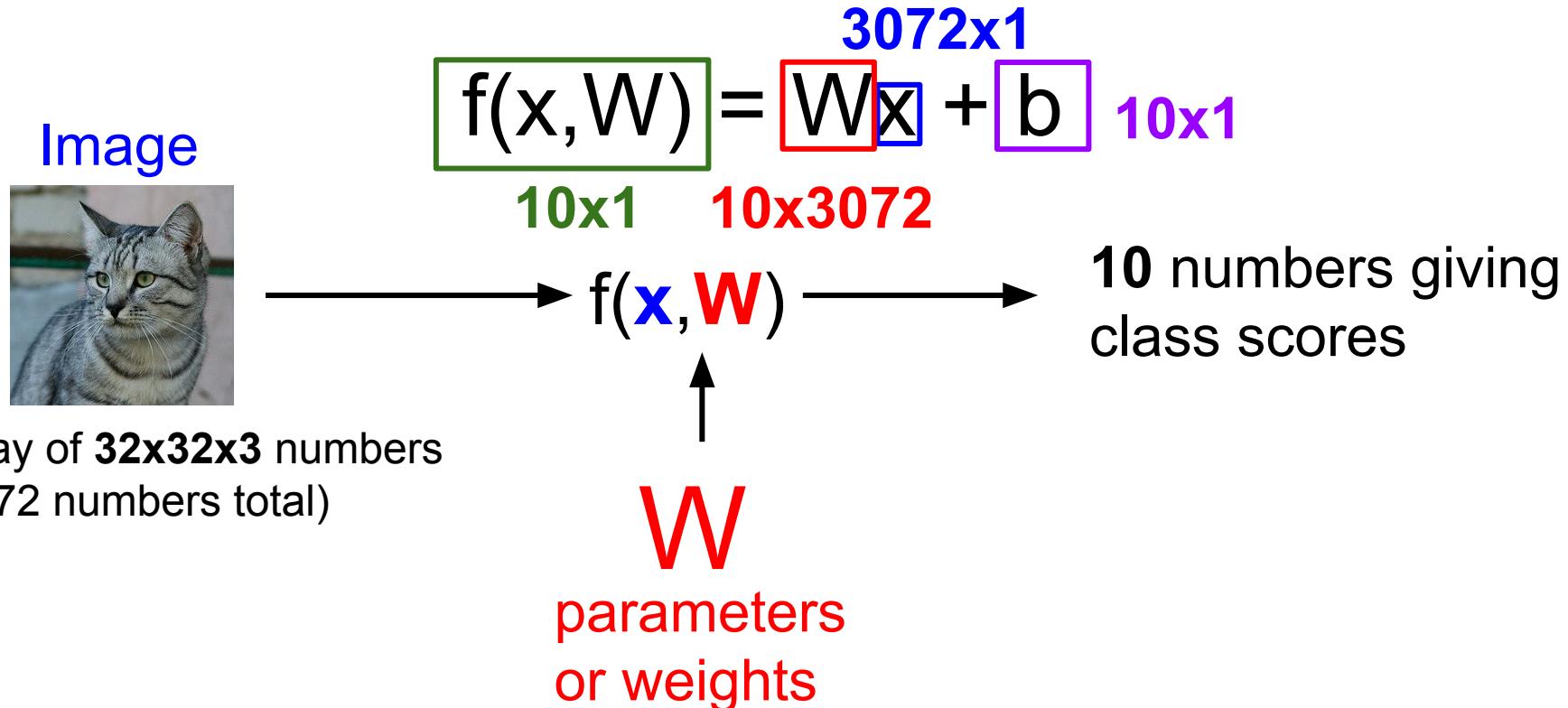
Parametric Approach: Linear Classifier



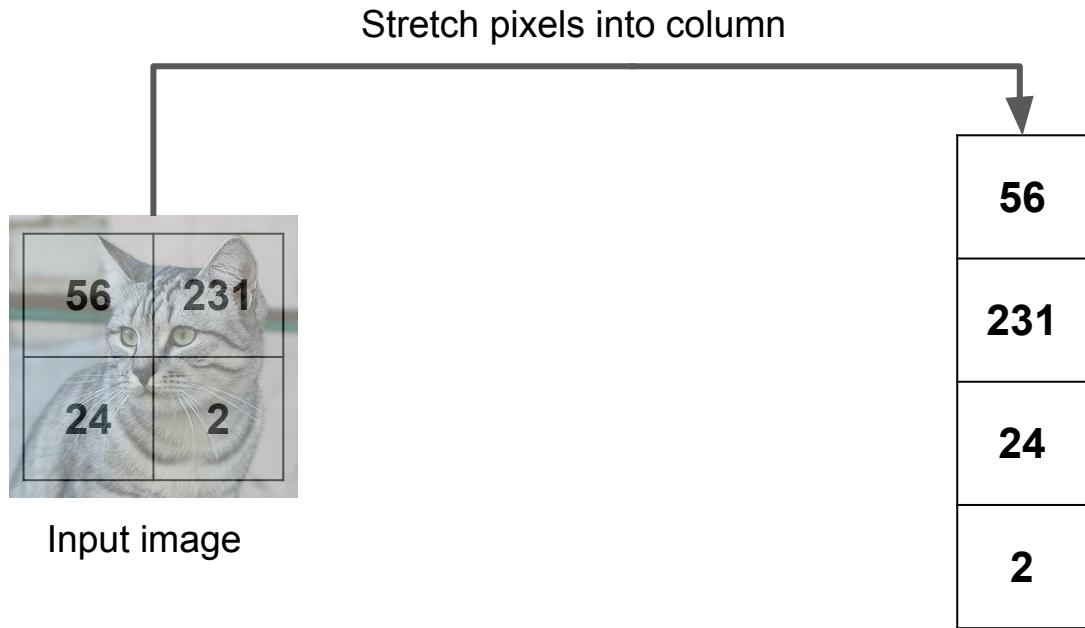
Parametric Approach: Linear Classifier



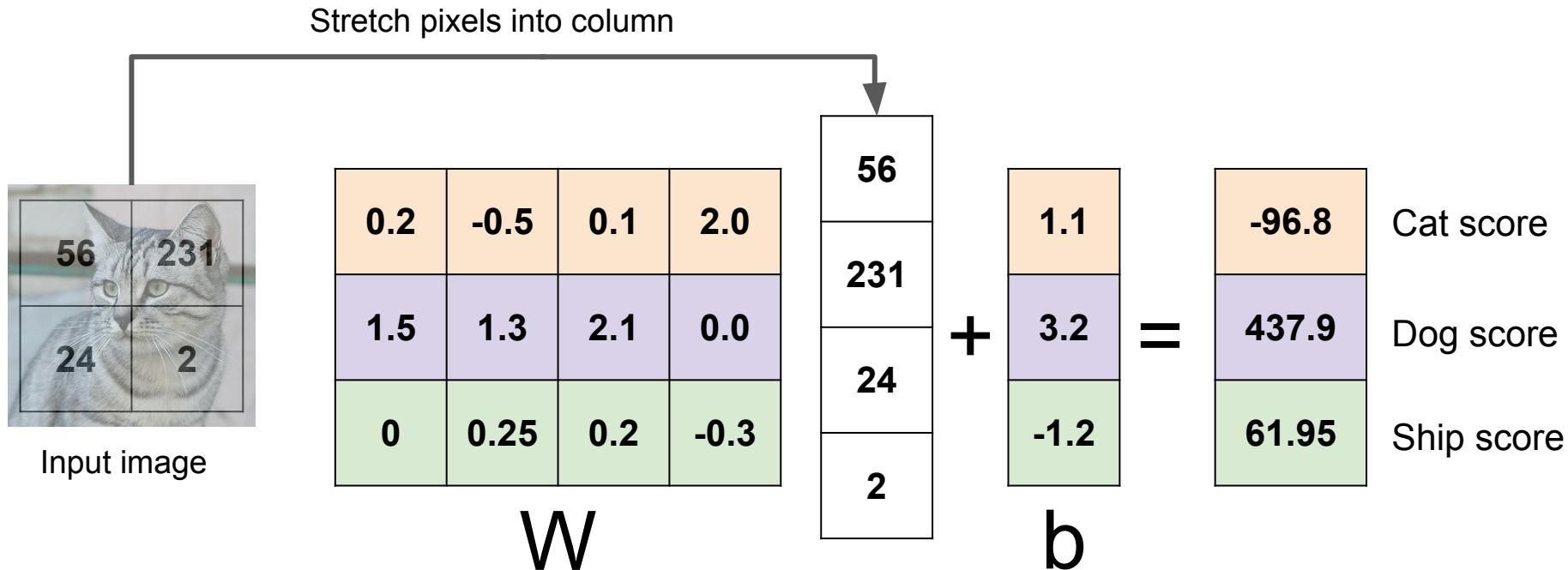
Parametric Approach: Linear Classifier



Example with an image with 4 pixels, and 3 classes (**cat/dog/ship**)



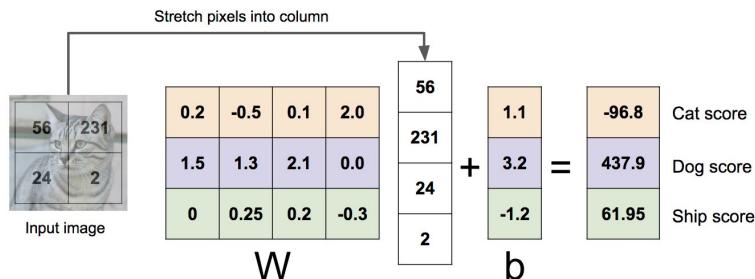
Example with an image with 4 pixels, and 3 classes (cat/dog/ship)



Example with an image with 4 pixels, and 3 classes (**cat/dog/ship**)

Algebraic Viewpoint

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



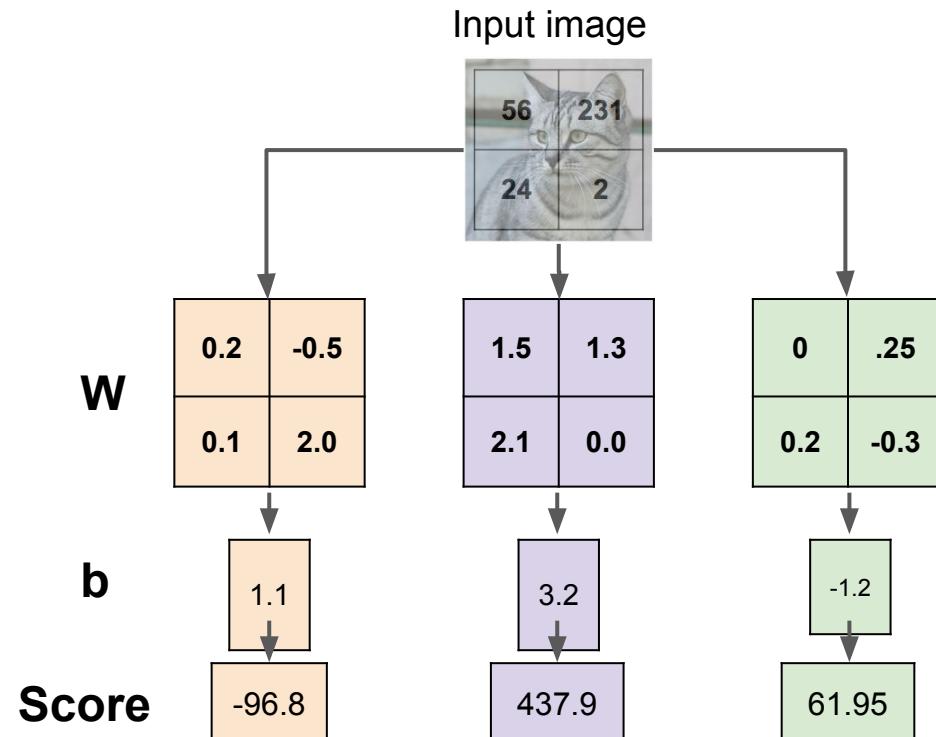
Example with an image with 4 pixels, and 3 classes (cat/dog/ship)

Algebraic Viewpoint

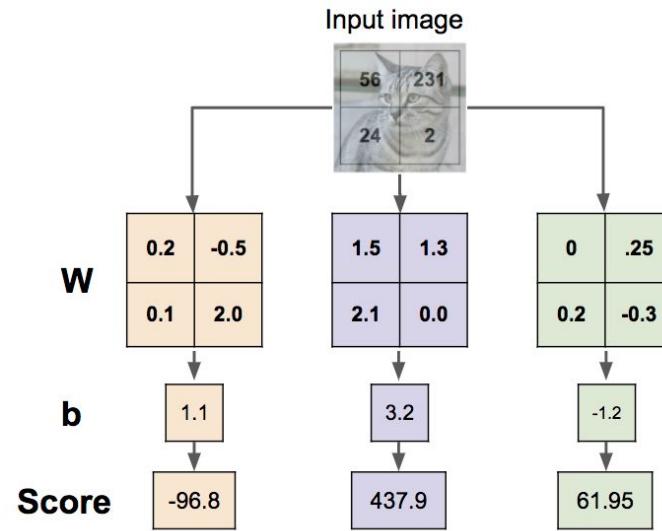
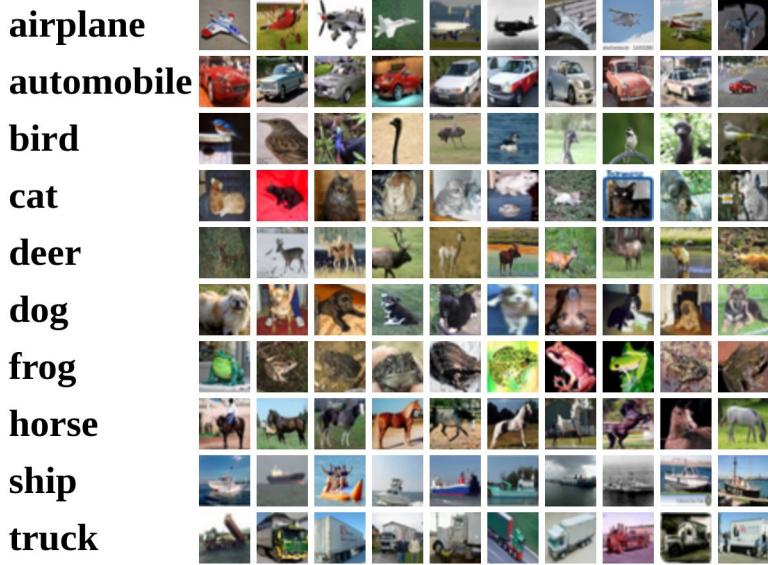
$$f(x, W) = Wx$$

Stretch pixels into column

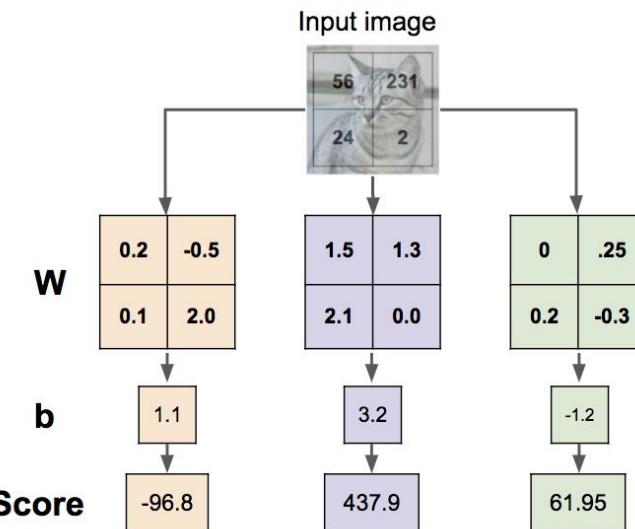
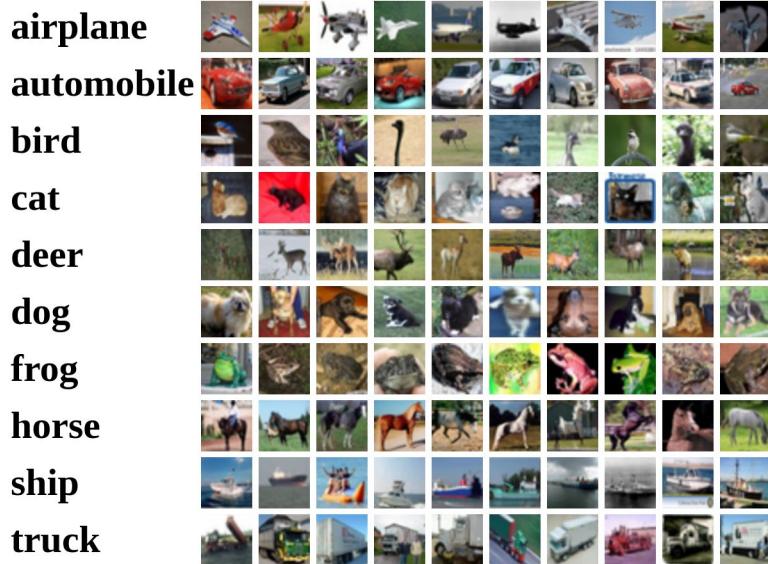
$$\begin{matrix} \text{Input image} \\ \begin{bmatrix} 56 & 231 \\ 24 & 2 \end{bmatrix} \end{matrix} \xrightarrow{\text{Stretch pixels into column}} \begin{matrix} W \\ \begin{bmatrix} 0.2 & -0.5 & 0.1 & 2.0 \\ 1.5 & 1.3 & 2.1 & 0.0 \\ 0 & 0.25 & 0.2 & -0.3 \end{bmatrix} \end{matrix}$$
$$\begin{matrix} b \\ \begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix} \end{matrix} + \begin{matrix} b \\ \begin{bmatrix} 1.1 \\ 3.2 \\ -1.2 \end{bmatrix} \end{matrix} = \begin{matrix} \text{Cat score} \\ -96.8 \end{matrix} \quad \begin{matrix} \text{Dog score} \\ 437.9 \end{matrix} \quad \begin{matrix} \text{Ship score} \\ 61.95 \end{matrix}$$



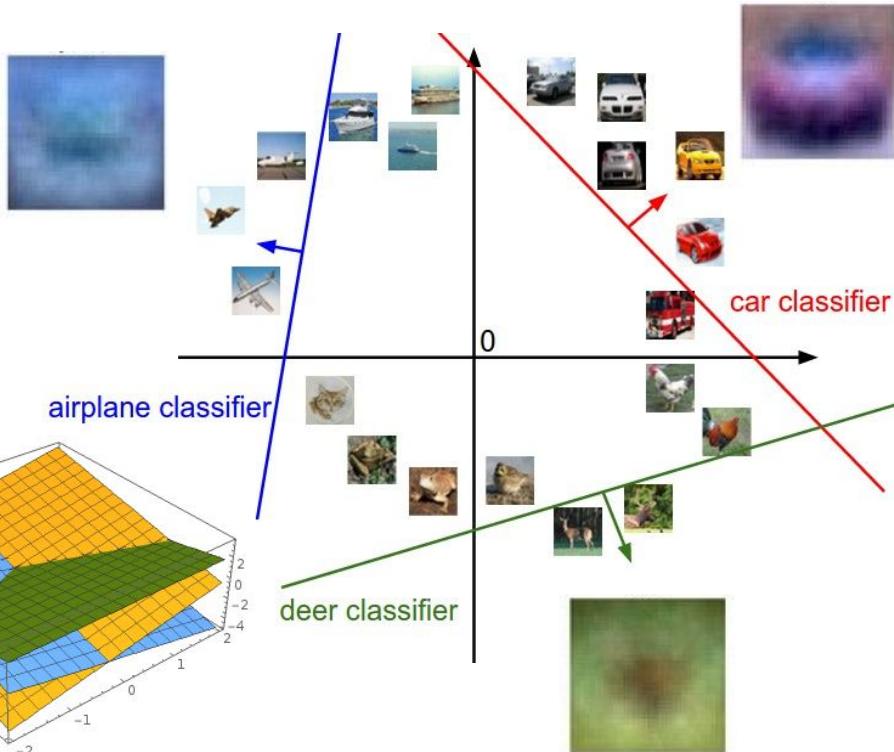
Interpreting a Linear Classifier



Interpreting a Linear Classifier: Visual Viewpoint



Interpreting a Linear Classifier: Geometric Viewpoint



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



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

Plot created using [Wolfram Cloud](#)

[Cat image](#) by [Nikita](#) is licensed under [CC-BY 2.0](#)

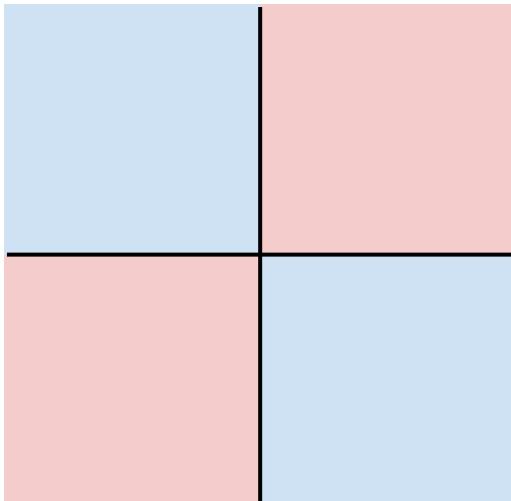
Hard cases for a linear classifier

Class 1:

First and third quadrants

Class 2:

Second and fourth quadrants

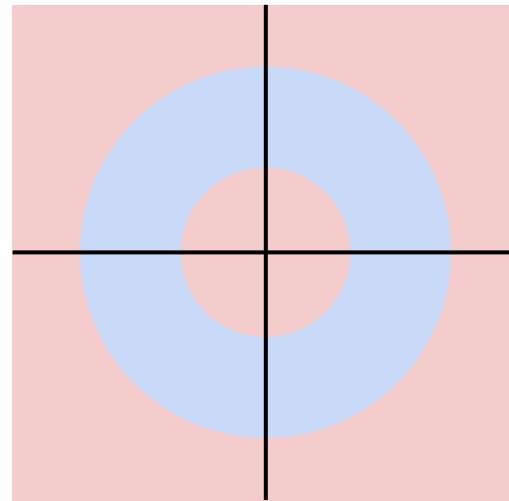


Class 1:

$1 \leq L_2 \text{ norm} \leq 2$

Class 2:

Everything else

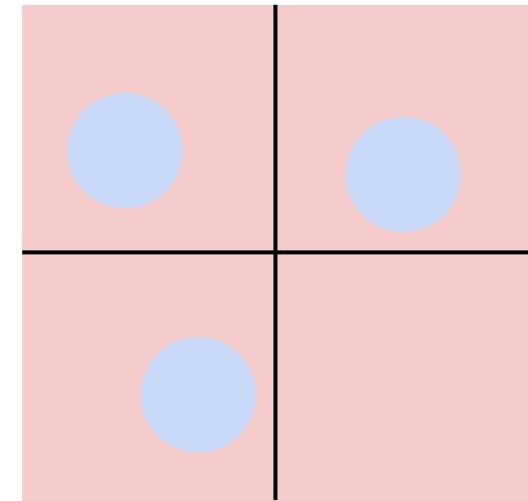


Class 1:

Three modes

Class 2:

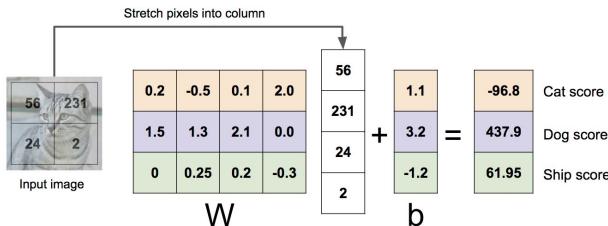
Everything else



Linear Classifier: Three Viewpoints

Algebraic Viewpoint

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



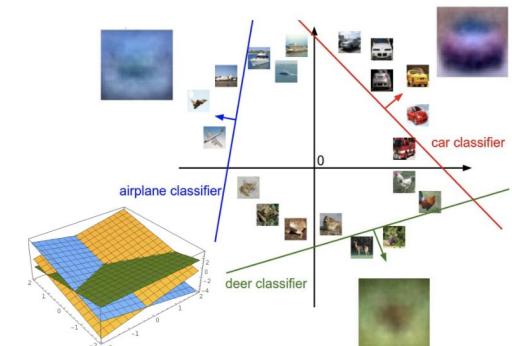
Visual Viewpoint

One template per class



Geometric Viewpoint

Hyperplanes cutting up space



So far: Defined a (linear) score function $f(x, W) = Wx + b$

Example class scores for 3 images for some W :

How can we tell whether this W is good or bad?



airplane	-3.45	-0.51	3.42
automobile	-8.87	6.04	4.64
bird	0.09	5.31	2.65
cat	2.9	-4.22	5.1
deer	4.48	-4.19	2.64
dog	8.02	3.58	5.55
frog	3.78	4.49	-4.34
horse	1.06	-4.37	-1.5
ship	-0.36	-2.09	-4.79
truck	-0.72	-2.93	6.14

[Cat image](#) by [Nikita](#) is licensed under [CC-BY 2.0](#)

[Car image](#) is [CC0 1.0](#) public domain

[Frog image](#) is in the public domain

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

Coming up:

- Loss function
- Optimization
- ConvNets!

(quantifying what it means to have a “good” W)

(start with random W and find a W that minimizes the loss)

(tweak the functional form of f)

Lecture 3: Loss Functions and Optimization

Administrative: Live Questions

We'll use Zoom to take questions from remote students live-streaming the lecture

Check Piazza for instructions and meeting ID:

<https://piazza.com/class/jdmurnqexkt47x?cid=108>

Administrative: Office Hours

Office hours started this week, schedule is on the course website:

http://cs231n.stanford.edu/office_hours.html

Areas of expertise for all TAs are posted on Piazza:

<https://piazza.com/class/jdmurnqexkt47x?cid=155>

Administrative: Assignment 1

Assignment 1 is released:

<http://cs231n.github.io/assignments2018/assignment1/>

Due **Wednesday April 18, 11:59pm**

Administrative: Google Cloud

You should have received an email yesterday about claiming a coupon for Google Cloud; make a private post on Piazza if you didn't get it

There was a problem with @cs.stanford.edu emails; this is resolved

If you have problems with coupons: **Post on Piazza**

DO NOT email me, DO NOT email Prof. Phil Levis

Administrative: SCPD Tutors

This year the SCPD office has hired tutors specifically for SCPD students taking CS231N; you should have received an email about this yesterday (4/9/2018)

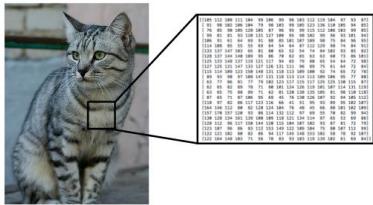
Administrative: Poster Session

Poster session will be Tuesday June 12 (our final exam slot)

Attendance is mandatory for non-SCPD students; if you don't have a legitimate reason for skipping it then you forfeit the points for the poster presentation

Recall from last time: Challenges of recognition

Viewpoint



Illumination



[This image](#) is CC0 1.0 public domain

Deformation



[This image](#) by Umberto Salvagnin
is licensed under CC-BY 2.0

Occlusion



[This image](#) by jonsson is licensed
under CC-BY 2.0

Clutter



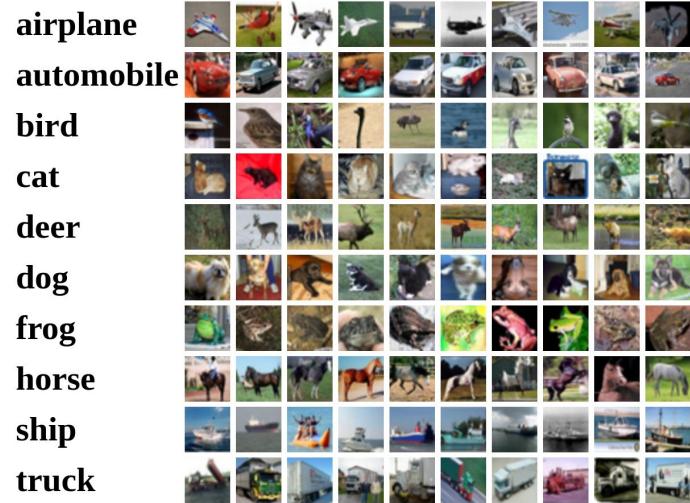
[This image](#) is CC0 1.0 public domain

Intraclass Variation

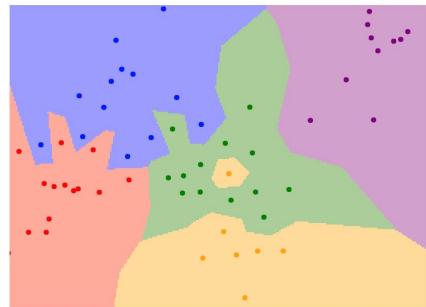


[This image](#) is CC0 1.0 public domain

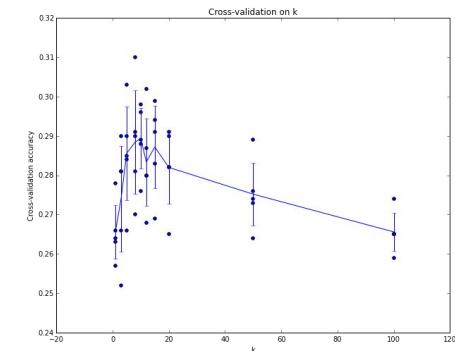
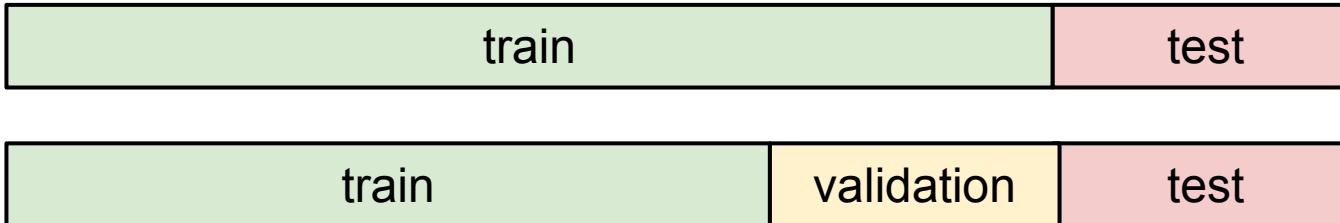
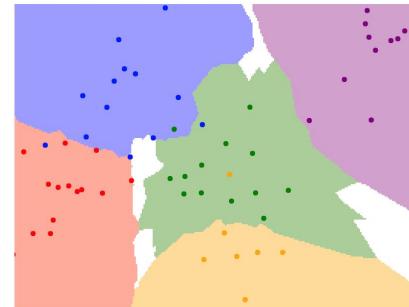
Recall from last time: data-driven approach, kNN



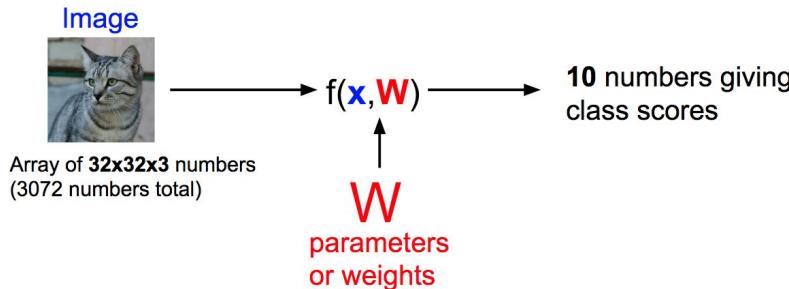
1-NN classifier



5-NN classifier



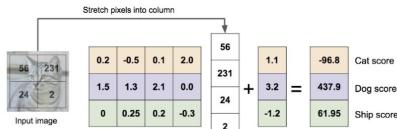
Recall from last time: Linear Classifier



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

Algebraic Viewpoint

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



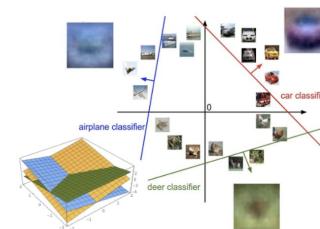
Visual Viewpoint

One template per class



Geometric Viewpoint

Hyperplanes cutting up space

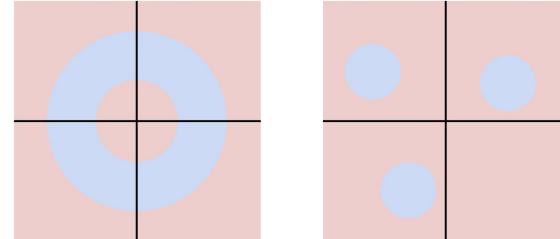


Class 1:
 $1 \leq L_2 \text{ norm} \leq 2$

Class 2:
Everything else

Class 1:
Three modes

Class 2:
Everything else



Recall from last time: Linear Classifier



airplane	-3.45	-0.51	3.42
automobile	-8.87	6.04	4.64
bird	0.09	5.31	2.65
cat	2.9	-4.22	5.1
deer	4.48	-4.19	2.64
dog	8.02	3.58	5.55
frog	3.78	4.49	-4.34
horse	1.06	-4.37	-1.5
ship	-0.36	-2.09	-4.79
truck	-0.72	-2.93	6.14

TODO:

1. Define a **loss function** that quantifies our unhappiness with the scores across the training data.
2. Come up with a way of efficiently finding the parameters that minimize the loss function.
(optimization)

Cat image by Nikita is licensed under CC-BY 2.0; Car image is CC0 1.0 public domain; Frog image is in the public domain

Suppose: 3 training examples, 3 classes.

With some W the scores $f(x, W) = Wx$ are:



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1

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

Given a dataset of examples

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

Where x_i is image and
 y_i is (integer) label

Loss over the dataset is a sum of loss over examples:

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

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1

Multiclass SVM loss:

Given an example (x_i, y_i) where x_i is the image and where y_i is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

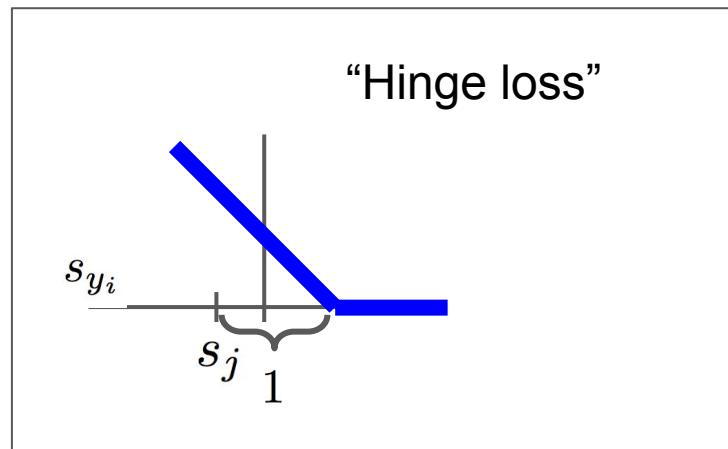
$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases}$$
$$= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1

Multiclass SVM loss:



$$\begin{aligned}
 L_i &= \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases} \\
 &= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)
 \end{aligned}$$

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1

Multiclass SVM loss:

Given an example (x_i, y_i) where x_i is the image and where y_i is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1
Losses:	2.9		

Multiclass SVM loss:

Given an example (x_i, y_i) where x_i is the image and where y_i is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

$$\begin{aligned}
 L_i &= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \\
 &= \max(0, 5.1 - 3.2 + 1) \\
 &\quad + \max(0, -1.7 - 3.2 + 1) \\
 &= \max(0, 2.9) + \max(0, -3.9) \\
 &= 2.9 + 0 \\
 &= 2.9
 \end{aligned}$$

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1
Losses:	2.9	0	

Multiclass SVM loss:

Given an example (x_i, y_i) where x_i is the image and where y_i is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

$$\begin{aligned}
 L_i &= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \\
 &= \max(0, 1.3 - 4.9 + 1) \\
 &\quad + \max(0, 2.0 - 4.9 + 1) \\
 &= \max(0, -2.6) + \max(0, -1.9) \\
 &= 0 + 0 \\
 &= 0
 \end{aligned}$$

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1
Losses:	2.9	0	12.9

Multiclass SVM loss:

Given an example (x_i, y_i) where x_i is the image and where y_i is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

$$\begin{aligned}
 L_i &= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \\
 &= \max(0, 2.2 - (-3.1) + 1) \\
 &\quad + \max(0, 2.5 - (-3.1) + 1) \\
 &= \max(0, 6.3) + \max(0, 6.6) \\
 &= 6.3 + 6.6 \\
 &= 12.9
 \end{aligned}$$

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1
Losses:	2.9	0	12.9

Multiclass SVM loss:

Given an example (x_i, y_i) where x_i is the image and where y_i is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Loss over full dataset is average:

$$L = \frac{1}{N} \sum_{i=1}^N L_i$$

$$\begin{aligned} L &= (2.9 + 0 + 12.9)/3 \\ &= 5.27 \end{aligned}$$

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1
Losses:	2.9	0	12.9

Multiclass SVM loss:

Given an example (x_i, y_i) where x_i is the image and where y_i is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Q: What happens to loss if car scores change a bit?

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1
Losses:	2.9	0	12.9

Multiclass SVM loss:

Given an example (x_i, y_i) where x_i is the image and where y_i is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Q2: what is the min/max possible loss?

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1
Losses:	2.9	0	12.9

Multiclass SVM loss:

Given an example (x_i, y_i) where x_i is the image and where y_i is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Q3: At initialization W is small so all $s \approx 0$. What is the loss?

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1
Losses:	2.9	0	12.9

Multiclass SVM loss:

Given an example (x_i, y_i) where x_i is the image and where y_i is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Q4: What if the sum was over all classes?
(including $j = y_i$)

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1
Losses:	2.9	0	12.9

Multiclass SVM loss:

Given an example (x_i, y_i) where x_i is the image and where y_i is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Q5: What if we used mean instead of sum?

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1
Losses:	2.9	0	12.9

Multiclass SVM loss:

Given an example (x_i, y_i) where x_i is the image and where y_i is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Q6: What if we used

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)^2$$

Multiclass SVM Loss: Example code

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

```
def L_i_vectorized(x, y, W):
    scores = W.dot(x)
    margins = np.maximum(0, scores - scores[y] + 1)
    margins[y] = 0
    loss_i = np.sum(margins)
    return loss_i
```

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

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1)$$

E.g. Suppose that we found a W such that $L = 0$.
Is this W unique?

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

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1)$$

E.g. Suppose that we found a W such that $L = 0$.
Is this W unique?

No! $2W$ is also has $L = 0!$

Suppose: 3 training examples, 3 classes.

With some W the scores $f(x, W) = Wx$ are:



cat	3.2	1.3	2.2	
car	5.1	4.9	2.5	
frog	-1.7	2.0	-3.1	
Losses:	2.9	0		

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Before:

$$\begin{aligned} &= \max(0, 1.3 - 4.9 + 1) \\ &\quad + \max(0, 2.0 - 4.9 + 1) \\ &= \max(0, -2.6) + \max(0, -1.9) \\ &= 0 + 0 \\ &= 0 \end{aligned}$$

With W twice as large:

$$\begin{aligned} &= \max(0, 2.6 - 9.8 + 1) \\ &\quad + \max(0, 4.0 - 9.8 + 1) \\ &= \max(0, -6.2) + \max(0, -4.8) \\ &= 0 + 0 \\ &= 0 \end{aligned}$$

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

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1)$$

E.g. Suppose that we found a W such that $L = 0$.
Is this W unique?

No! $2W$ is also has $L = 0$!

How do we choose between W and $2W$?

Regularization

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

Data loss: Model predictions
should match training data

Regularization

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


Data loss: Model predictions should match training data

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

Regularization

λ = regularization strength
(hyperparameter)

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss: Model predictions should match training data}} + \underbrace{\lambda R(W)}_{\text{Regularization: Prevent the model from doing } \textit{too well} \text{ on training data}}$$

Data loss: Model predictions should match training data

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

Regularization

λ = regularization strength
(hyperparameter)

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



Data loss: Model predictions should match training data

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

Simple examples

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

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

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

Regularization

λ = regularization strength
(hyperparameter)

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



Data loss: Model predictions should match training data

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

Simple examples

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

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

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

More complex:

Dropout

Batch normalization

Stochastic depth, fractional pooling, etc

Regularization

λ = regularization strength
(hyperparameter)

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


Data loss: Model predictions should match training data

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

Why regularize?

- Express preferences over weights
- Make the model *simple* so it works on test data
- Improve optimization by adding curvature

Regularization: Expressing Preferences

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

L2 Regularization

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

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

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

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

Regularization: Expressing Preferences

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

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

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

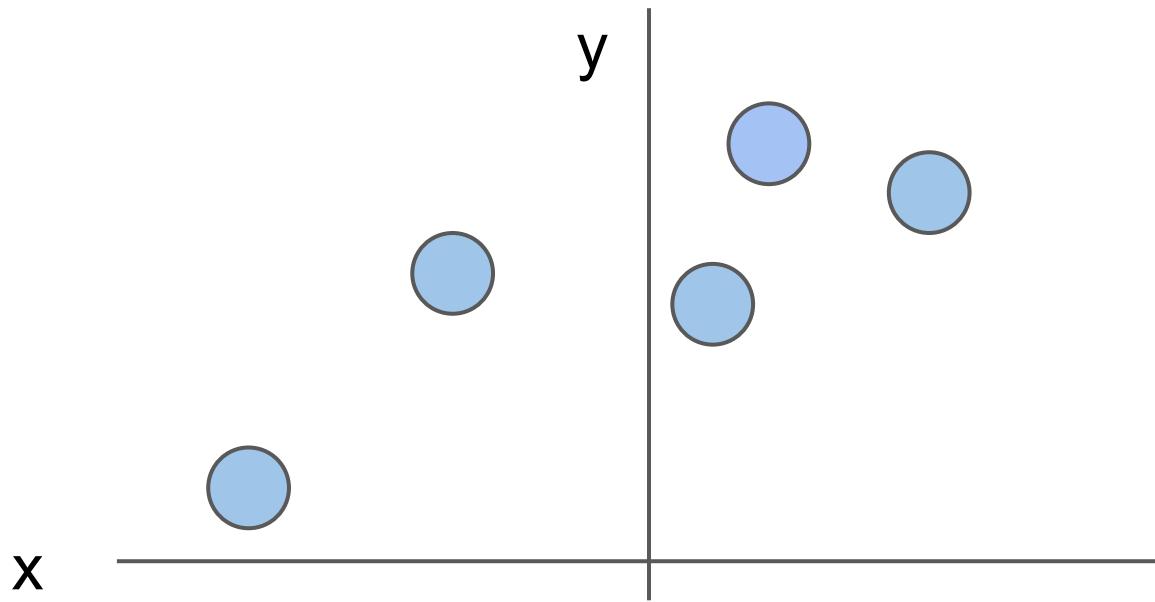
L2 Regularization

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

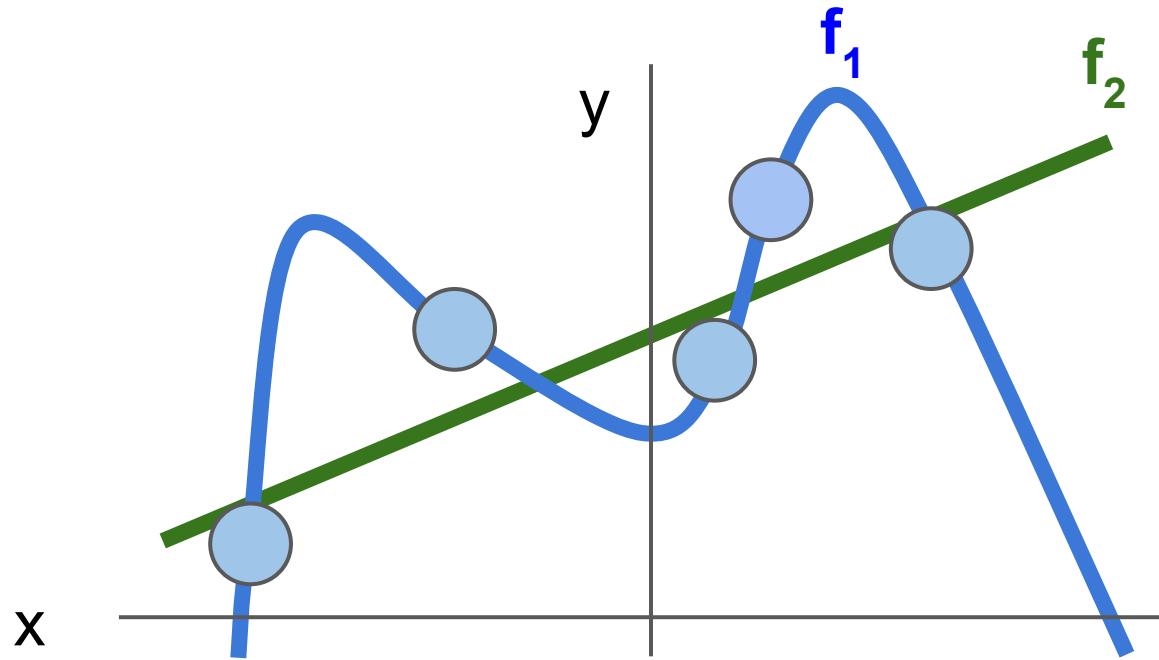
L2 regularization likes to
“spread out” the weights

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

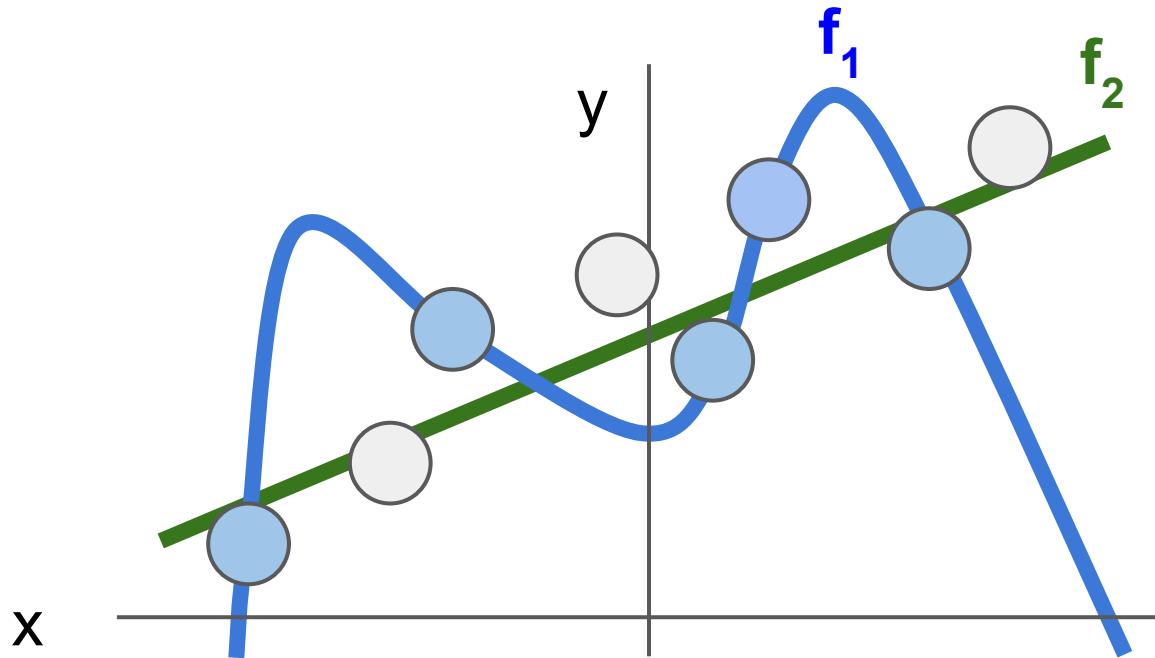
Regularization: Prefer Simpler Models



Regularization: Prefer Simpler Models



Regularization: Prefer Simpler Models



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

Softmax Classifier (Multinomial Logistic Regression)

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



cat	3.2
car	5.1
frog	-1.7

Softmax Classifier (Multinomial Logistic Regression)

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



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

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

Softmax
Function

cat	3.2
car	5.1
frog	-1.7

Softmax Classifier (Multinomial Logistic Regression)



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

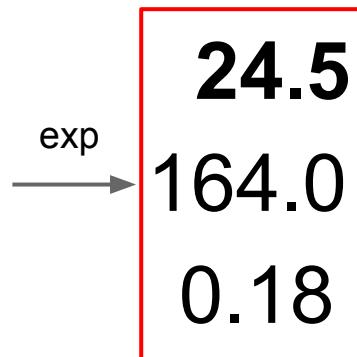
$$s = f(x_i; W)$$

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

Softmax
Function

Probabilities
must be ≥ 0

cat	3.2
car	5.1
frog	-1.7



unnormalized
probabilities

Softmax Classifier (Multinomial Logistic Regression)

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



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

Probabilities
must be ≥ 0

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

Softmax
Function

cat	3.2
car	5.1
frog	-1.7

exp

24.5
164.0
0.18

unnormalized
probabilities

normalize

0.13
0.87
0.00

probabilities

Softmax Classifier (Multinomial Logistic Regression)

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



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

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

Softmax Function

cat
car
frog

3.2
5.1
-1.7

Unnormalized
log-probabilities / logits

exp

24.5
164.0
0.18

unnormalized
probabilities

normalize

0.13
0.87
0.00

probabilities

Softmax Classifier (Multinomial Logistic Regression)

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



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

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$
 Softmax Function

Probabilities
must be ≥ 0

Probabilities
must sum to 1

cat
car
frog

3.2
5.1
-1.7

Unnormalized
log-probabilities / logits

exp

24.5
164.0
0.18

unnormalized
probabilities

normalize

0.13
0.87
0.00

probabilities

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

$$\rightarrow L_i = -\log(0.13) \\ = 0.89$$

Softmax Classifier (Multinomial Logistic Regression)

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



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

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$
 Softmax Function

Probabilities
must be ≥ 0

Probabilities
must sum to 1

cat
car
frog

3.2
5.1
-1.7

Unnormalized
log-probabilities / logits

exp

24.5
164.0
0.18

unnormalized
probabilities

normalize

0.13
0.87
0.00

probabilities

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

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

Maximum Likelihood Estimation
Choose probabilities to maximize
the likelihood of the observed data
(See CS 229 for details)

Softmax Classifier (Multinomial Logistic Regression)



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

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

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

Softmax Function

Probabilities
must be ≥ 0

Probabilities
must sum to 1

cat
car
frog

3.2
5.1
-1.7

Unnormalized
log-probabilities / logits

exp

24.5
164.0
0.18

unnormalized
probabilities

normalize

0.13
0.87
0.00

probabilities

compare

1.00
0.00
0.00

Correct
probs

Softmax Classifier (Multinomial Logistic Regression)

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



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

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

Softmax Function

cat
car
frog

3.2
5.1
-1.7

Unnormalized
log-probabilities / logits

exp

24.5
164.0
0.18

unnormalized
probabilities

normalize

0.13
0.87
0.00

probabilities

compare

Kullback–Leibler
divergence

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

1.00
0.00
0.00

Correct
probs

Softmax Classifier (Multinomial Logistic Regression)



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

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

Probabilities
must be ≥ 0

cat
car
frog

3.2
5.1
-1.7

Unnormalized
log-probabilities / logits

24.5
164.0
0.18

unnormalized
probabilities

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

Softmax
Function

Probabilities
must sum to 1

0.13
0.87
0.00

probabilities

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

1.00
0.00
0.00

Cross Entropy

$$H(P, Q) = H(p) + D_{KL}(P||Q)$$

Correct
probs

Softmax Classifier (Multinomial Logistic Regression)

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



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

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

Softmax
Function

Maximize probability of correct class

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

Putting it all together:

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

cat	3.2
car	5.1
frog	-1.7

Softmax Classifier (Multinomial Logistic Regression)



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

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

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

Softmax
Function

cat	3.2
car	5.1
frog	-1.7

Maximize probability of correct class

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

Putting it all together:

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

Q: What is the min/max possible loss L_i ?

Softmax Classifier (Multinomial Logistic Regression)



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

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

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

Softmax Function

Maximize probability of correct class

cat	3.2
car	5.1
frog	-1.7

Putting it all together:

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

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

Q: What is the min/max possible loss L_i ?
A: min 0, max infinity

Softmax Classifier (Multinomial Logistic Regression)

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



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

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

Softmax
Function

Maximize probability of correct class

Putting it all together:

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

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

cat	3.2
car	5.1
frog	-1.7

Q2: At initialization all s will be approximately equal; what is the loss?

Softmax Classifier (Multinomial Logistic Regression)



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

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

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

Softmax Function

Maximize probability of correct class

Putting it all together:

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

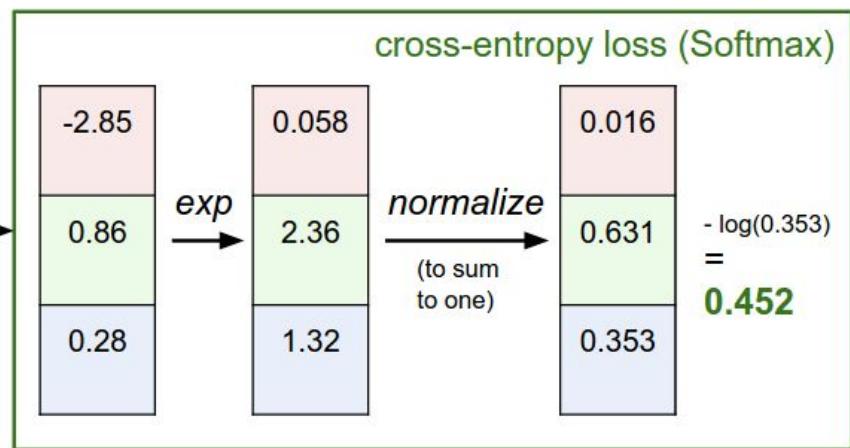
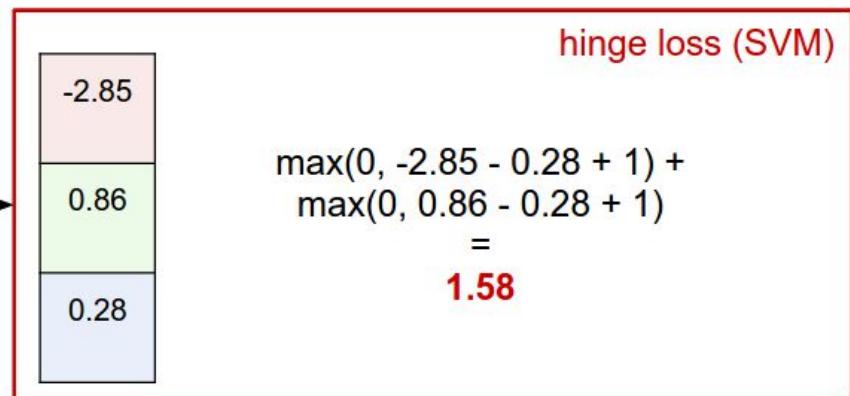
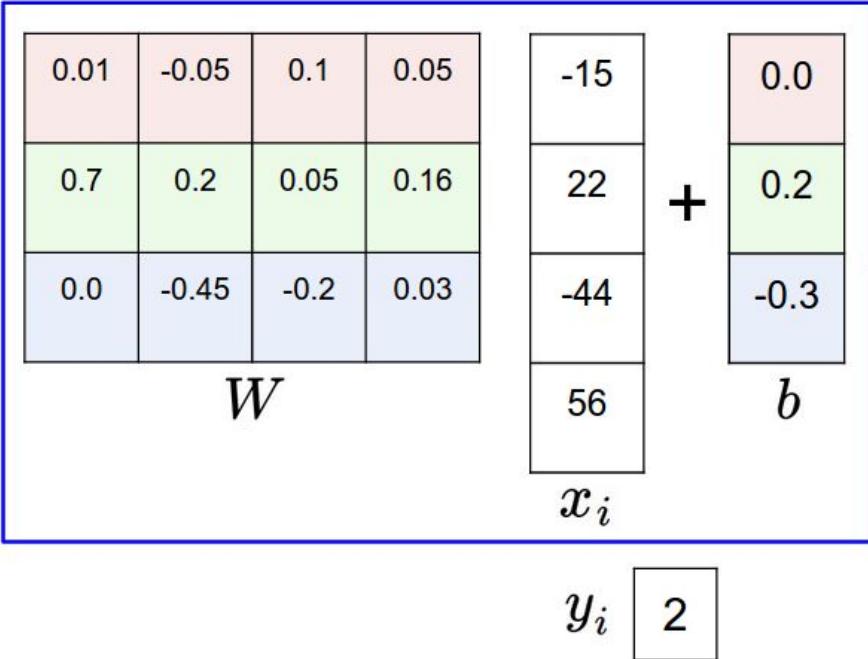
$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

cat	3.2
car	5.1
frog	-1.7

Q2: At initialization all s will be approximately equal; what is the loss?
A: $\log(C)$, eg $\log(10) \approx 2.3$

Softmax vs. SVM

matrix multiply + bias offset



Softmax vs. SVM

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Softmax vs. SVM

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

assume scores:

[10, -2, 3]

[10, 9, 9]

[10, -100, -100]

and $y_i = 0$

Q: Suppose I take a datapoint and I jiggle a bit (changing its score slightly). What happens to the loss in both cases?

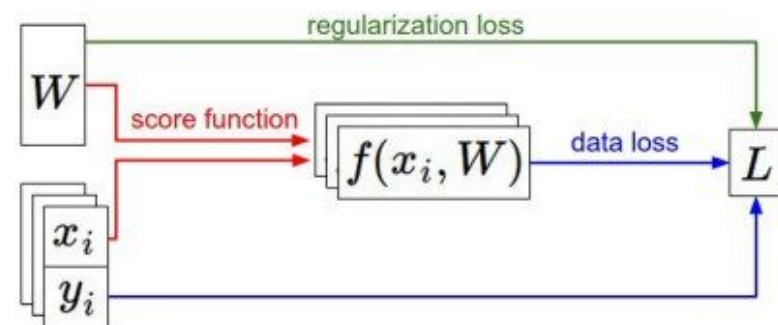
Recap

- We have some dataset of (x, y)
- We have a **score function**: $s = f(x; W) = Wx$ e.g.
- We have a **loss function**:

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{sj}}\right) \quad \text{Softmax}$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \quad \text{SVM}$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + R(W) \quad \text{Full loss}$$



Recap

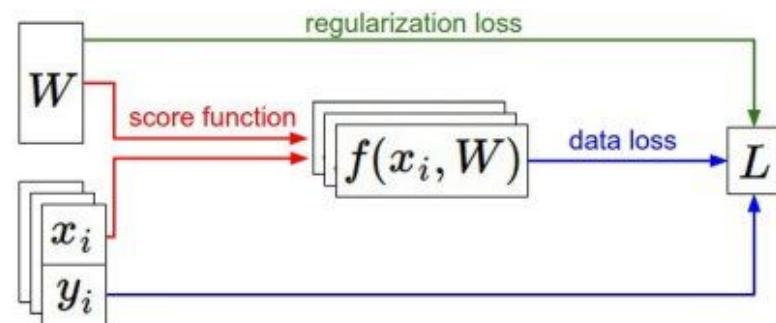
How do we find the best W ?

- We have some dataset of (x, y)
- We have a **score function**: $s = f(x; W) = Wx$ e.g.
- We have a **loss function**:

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{sj}}\right) \quad \text{Softmax}$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \quad \text{SVM}$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + R(W) \quad \text{Full loss}$$



Optimization



[This image](#) is CC0 1.0 public domain



[Walking man image is CC0 1.0 public domain](#)

Strategy #1: A first very bad idea solution: Random search

```
# assume X_train is the data where each column is an example (e.g. 3073 x 50,000)
# assume Y_train are the labels (e.g. 1D array of 50,000)
# assume the function L evaluates the loss function

bestloss = float("inf") # Python assigns the highest possible float value
for num in xrange(1000):
    W = np.random.randn(10, 3073) * 0.0001 # generate random parameters
    loss = L(X_train, Y_train, W) # get the loss over the entire training set
    if loss < bestloss: # keep track of the best solution
        bestloss = loss
        bestW = W
    print 'in attempt %d the loss was %f, best %f' % (num, loss, bestloss)

# prints:
# in attempt 0 the loss was 9.401632, best 9.401632
# in attempt 1 the loss was 8.959668, best 8.959668
# in attempt 2 the loss was 9.044034, best 8.959668
# in attempt 3 the loss was 9.278948, best 8.959668
# in attempt 4 the loss was 8.857370, best 8.857370
# in attempt 5 the loss was 8.943151, best 8.857370
# in attempt 6 the loss was 8.605604, best 8.605604
# ... (truncated: continues for 1000 lines)
```

Lets see how well this works on the test set...

```
# Assume X_test is [3073 x 10000], Y_test [10000 x 1]
scores = Wbest.dot(Xte_cols) # 10 x 10000, the class scores for all test examples
# find the index with max score in each column (the predicted class)
Yte_predict = np.argmax(scores, axis = 0)
# and calculate accuracy (fraction of predictions that are correct)
np.mean(Yte_predict == Yte)
# returns 0.1555
```

15.5% accuracy! not bad!
(SOTA is ~95%)

Strategy #2: Follow the slope



Strategy #2: Follow the slope

In 1-dimension, the derivative of a function:

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

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

The slope in any direction is the **dot product** of the direction with the gradient
The direction of steepest descent is the **negative gradient**

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,
?,...]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

W + h (first dim):

[0.34 + 0.0001,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25322

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,...]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

W + h (first dim):

[0.34 + 0.0001,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25322

gradient dW:

[-2.5,

? ,

? ,

$$\frac{(1.25322 - 1.25347)}{0.0001} = -2.5$$

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

? ,

? ,...]

current W:	W + h (second dim):	gradient dW:
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]	[0.34, -1.11 + 0.0001 , 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]	[-2.5, ?, ?, ?, ?, ?, ?, ?, ?, ?,...]

loss 1.25347 **loss 1.25353**

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss **1.25347**

W + h (second dim):

[0.34,
-1.11 + **0.0001**,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss **1.25353**

gradient dW:

[-2.5,
0.6,
?,
?,

$$\frac{(1.25353 - 1.25347)}{0.0001} = 0.6$$

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

?,...]

current W:	W + h (third dim):	gradient dW:
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]	[0.34, -1.11, 0.78 + 0.0001 , 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]	[-2.5, 0.6, ?, ?, ?, ?, ?, ?, ?, ?,...]

loss 1.25347 **loss 1.25347**

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

W + h (third dim):

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

gradient dW:

[-2.5,
0.6,
0,
?,
?]

$$\frac{(1.25347 - 1.25347)}{0.0001} = 0$$

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

?, ...]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

W + h (third dim):

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

gradient dW:

[-2.5,
0.6,
0,
?,
?]

[...,]

Numeric Gradient

- Slow! Need to loop over all dimensions
- Approximate

This is silly. The loss is just a function of W :

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

want $\nabla_W L$

This is silly. The loss is just a function of W:

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

want $\nabla_W L$

Use calculus to compute an analytic gradient



[This image](#) is in the public domain



[This image](#) is in the public domain

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

gradient dW:

[-2.5,
0.6,
0,
0.2,
0.7,
-0.5,
1.1,
1.3,
-2.1,...]

dW = ...
(some function
data and W)



In summary:

- Numerical gradient: approximate, slow, easy to write
- Analytic gradient: exact, fast, error-prone

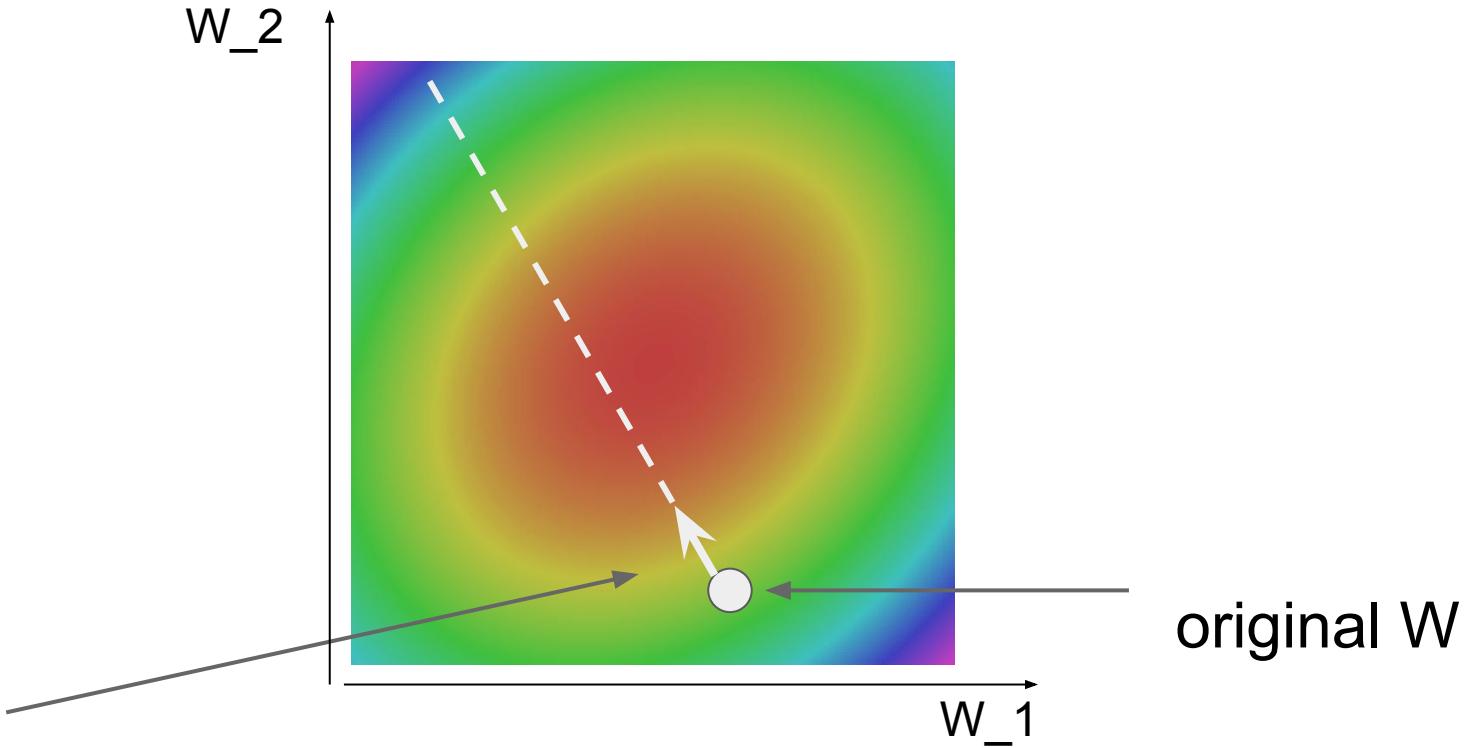
=>

In practice: Always use analytic gradient, but check implementation with numerical gradient. This is called a **gradient check**.

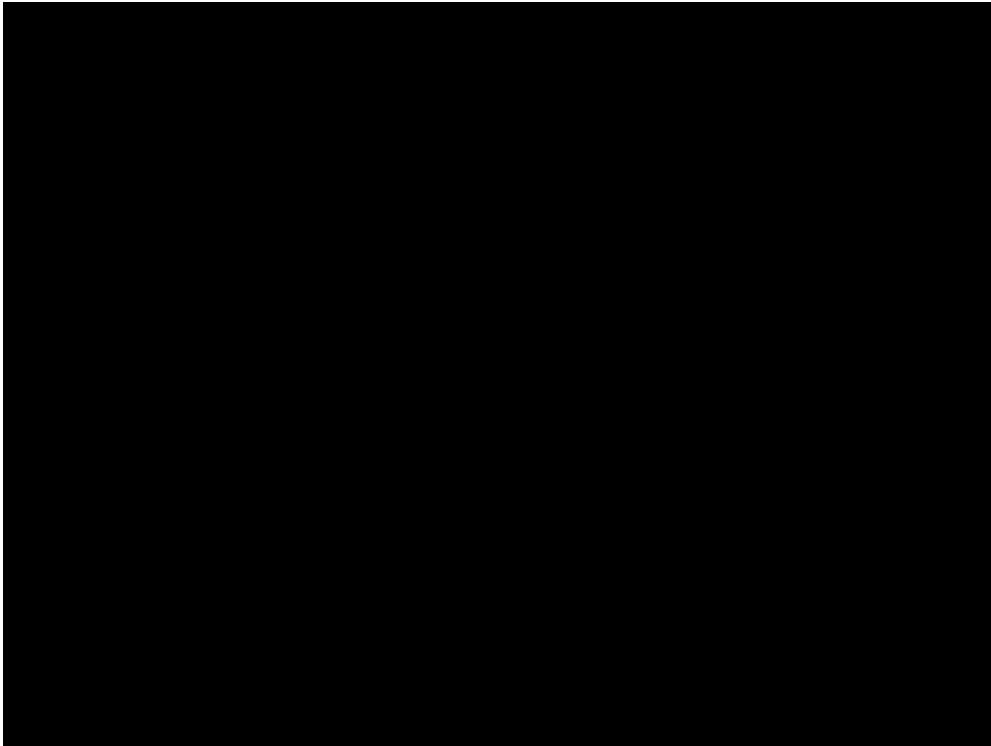
Gradient Descent

```
# Vanilla Gradient Descent

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



negative gradient direction



Stochastic Gradient Descent (SGD)

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

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

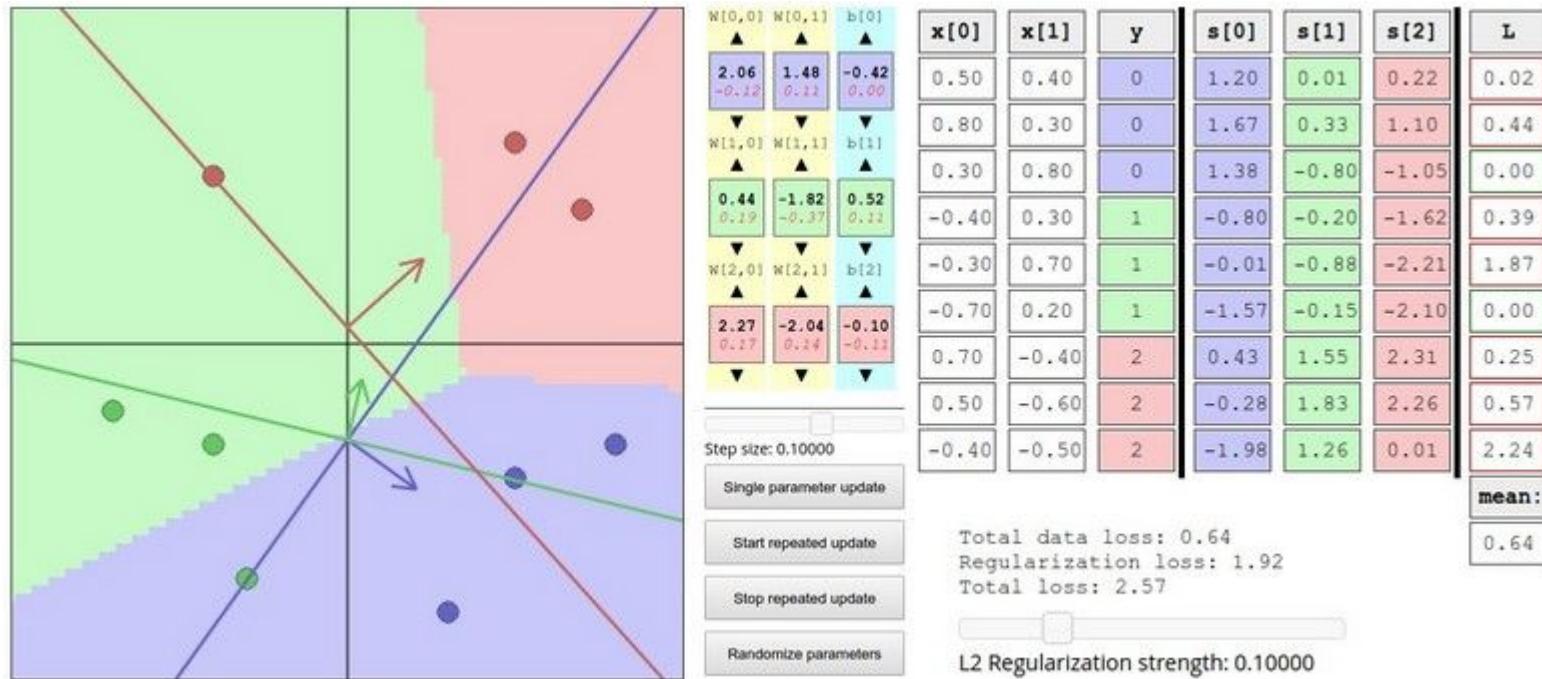
Full sum expensive
when N is large!

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

```
# Vanilla Minibatch Gradient Descent
```

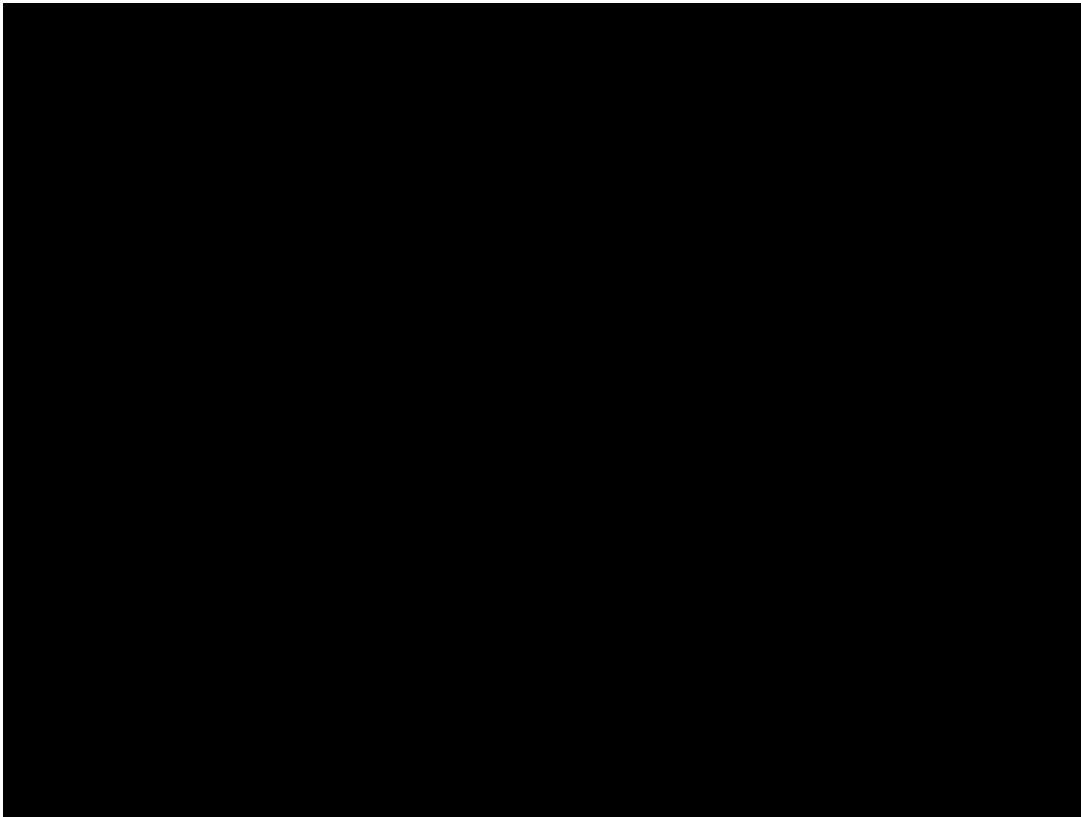
```
while True:  
    data_batch = sample_training_data(data, 256) # sample 256 examples  
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)  
    weights += - step_size * weights_grad # perform parameter update
```

Interactive Web Demo time....



<http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/>

Interactive Web Demo time....

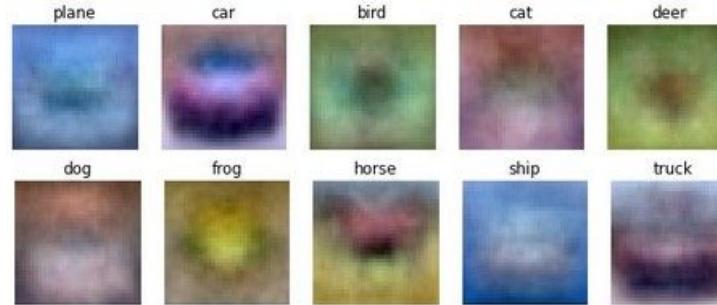


Aside: Image Features



$$f(x) = Wx$$

Class
scores



Aside: Image Features

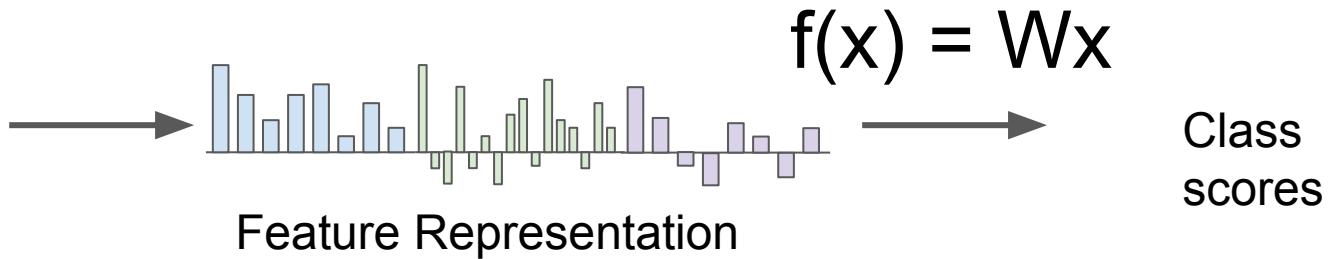
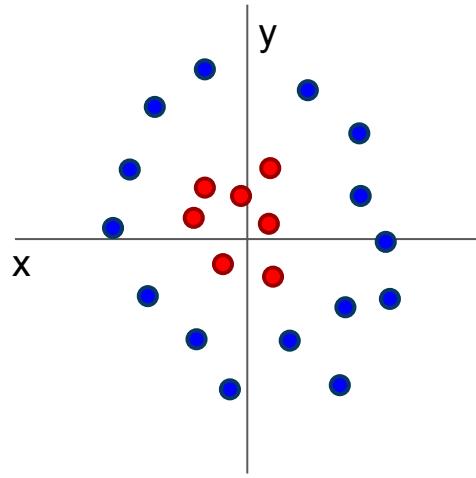
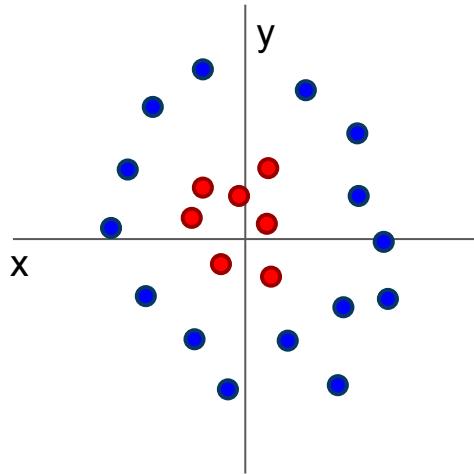


Image Features: Motivation



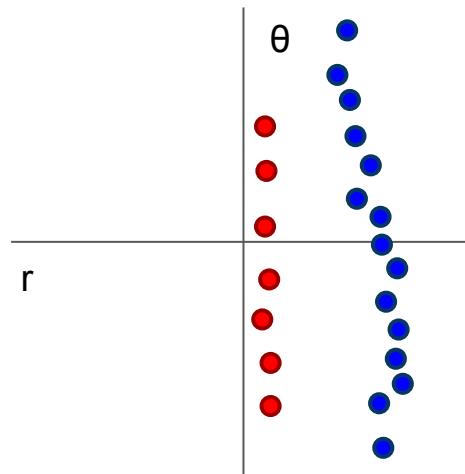
Cannot separate red
and blue points with
linear classifier

Image Features: Motivation



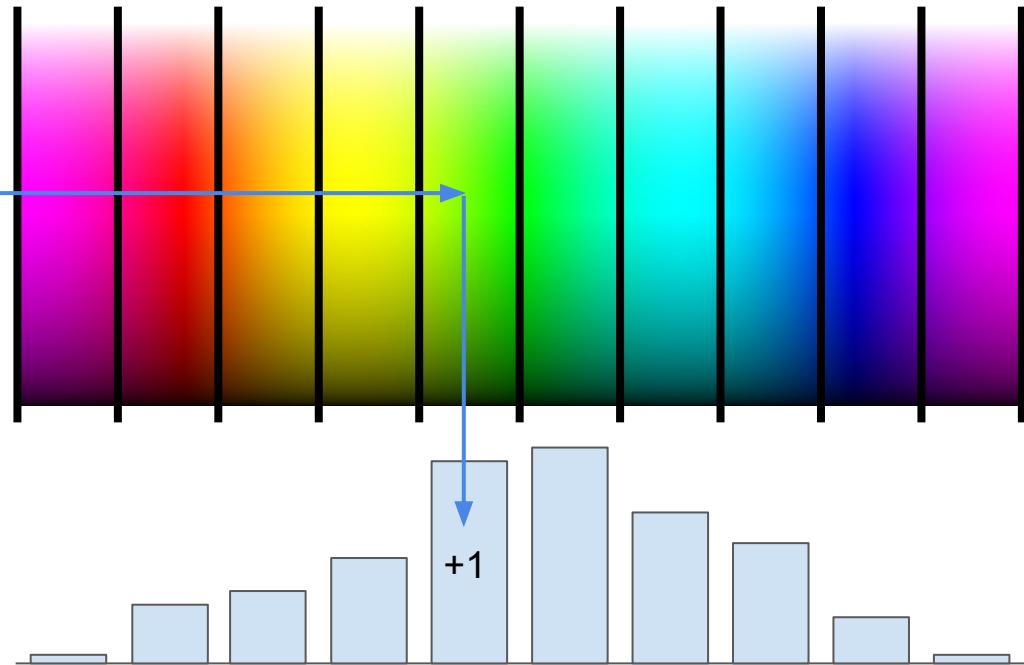
Cannot separate red
and blue points with
linear classifier

$$f(x, y) = (r(x, y), \theta(x, y))$$



After applying feature
transform, points can
be separated by linear
classifier

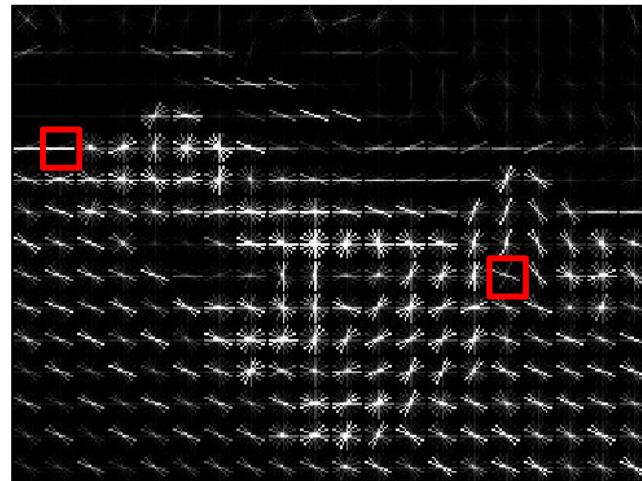
Example: Color Histogram



Example: Histogram of Oriented Gradients (HoG)



Divide image into 8x8 pixel regions
Within each region quantize edge
direction into 9 bins



Example: 320x240 image gets divided
into 40x30 bins; in each bin there are
9 numbers so feature vector has
 $30*40*9 = 10,800$ numbers

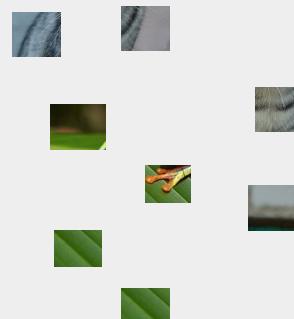
Lowe, "Object recognition from local scale-invariant features", ICCV 1999
Dalal and Triggs, "Histograms of oriented gradients for human detection," CVPR 2005

Example: Bag of Words

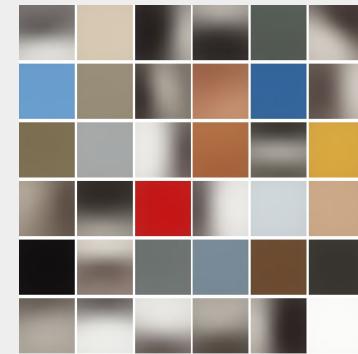
Step 1: Build codebook



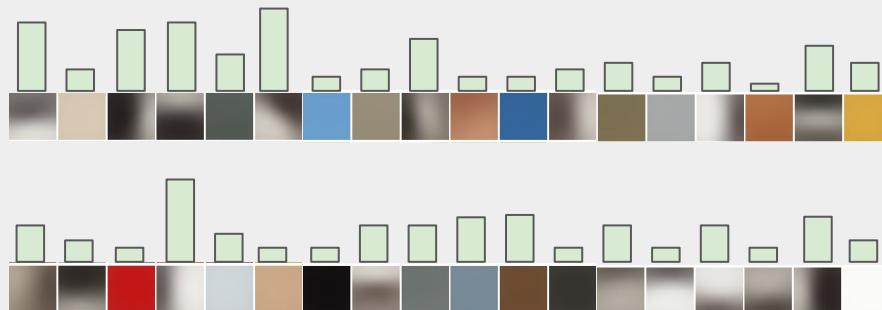
Extract random patches



Cluster patches to form “codebook” of “visual words”



Step 2: Encode images



Fei-Fei and Perona, "A bayesian hierarchical model for learning natural scene categories", CVPR 2005

Aside: Image Features

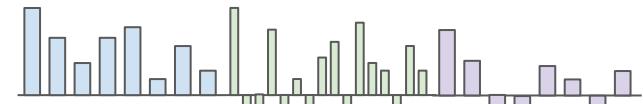
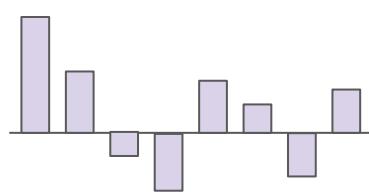
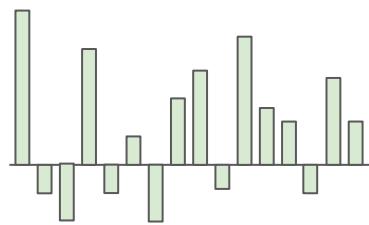
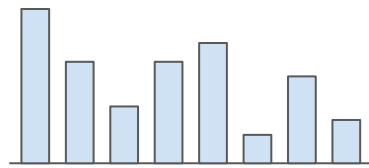
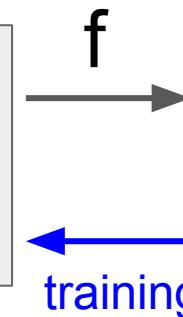
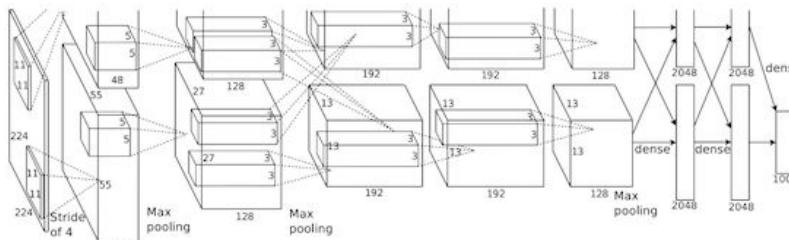


Image features vs ConvNets



10 numbers giving scores for classes



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

training

10 numbers giving scores for classes

Next time:

Introduction to neural networks

Backpropagation

Lecture 4: Backpropagation and Neural Networks

Administrative

Assignment 1 due Wednesday April 18, 11:59pm

Administrative

All office hours this week will use queuestatus

Where we are...

$$s = f(x; W) = Wx$$

scores function

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

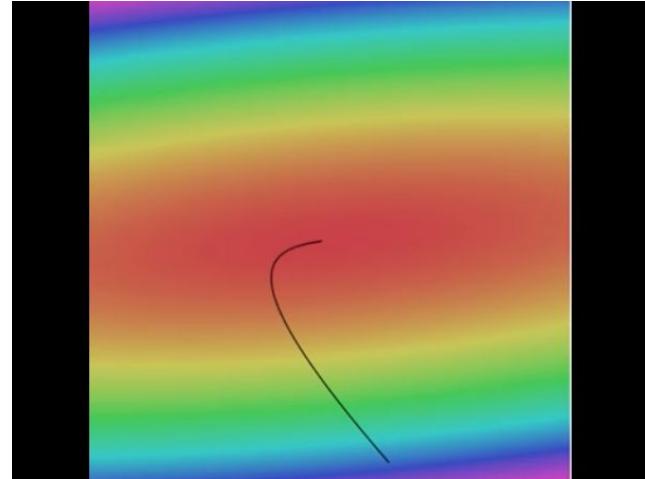
SVM loss

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \sum_k W_k^2$$

data loss + regularization

want $\boxed{\nabla_W L}$

Optimization



```
# Vanilla Gradient Descent

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

Landscape image is [CC0 1.0](#) public domain
Walking man image is [CC0 1.0](#) public domain

Gradient descent

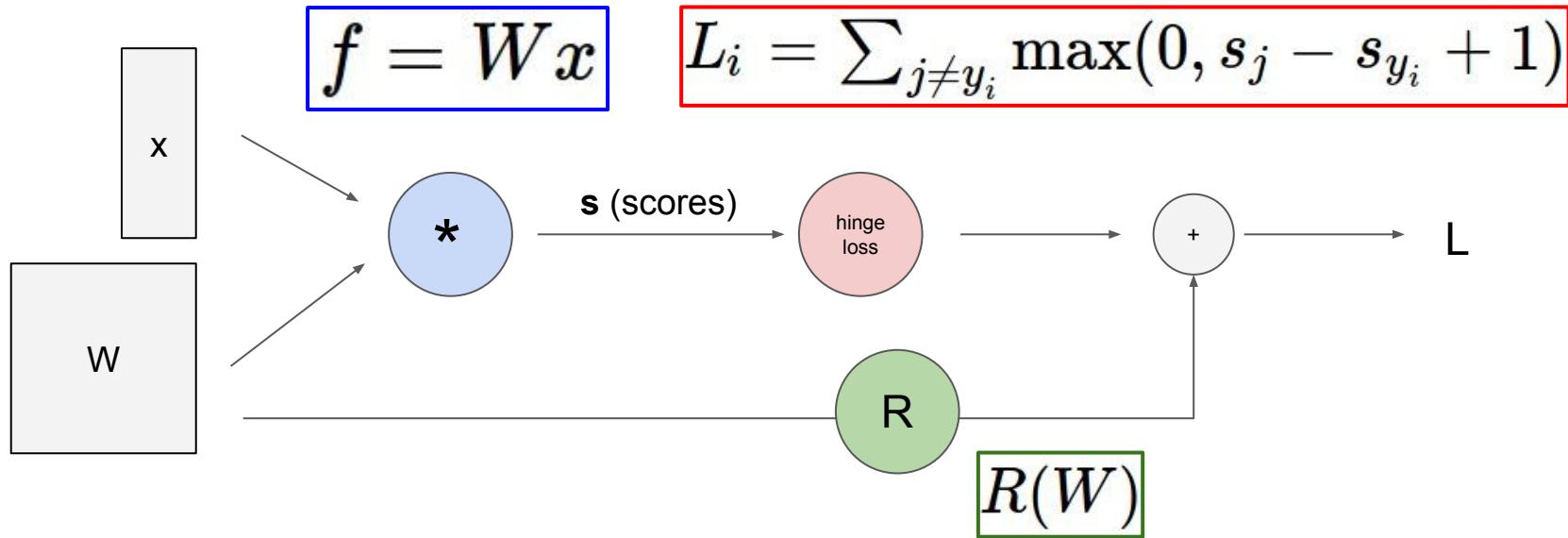
$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

Numerical gradient: slow :, approximate :, easy to write :)

Analytic gradient: fast :), exact :), error-prone :(

In practice: Derive analytic gradient, check your implementation with numerical gradient

Computational graphs



Convolutional network (AlexNet)

input image

weights

loss

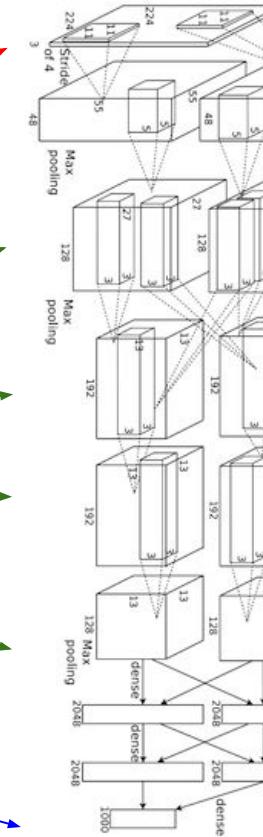


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

Neural Turing Machine

input image

loss

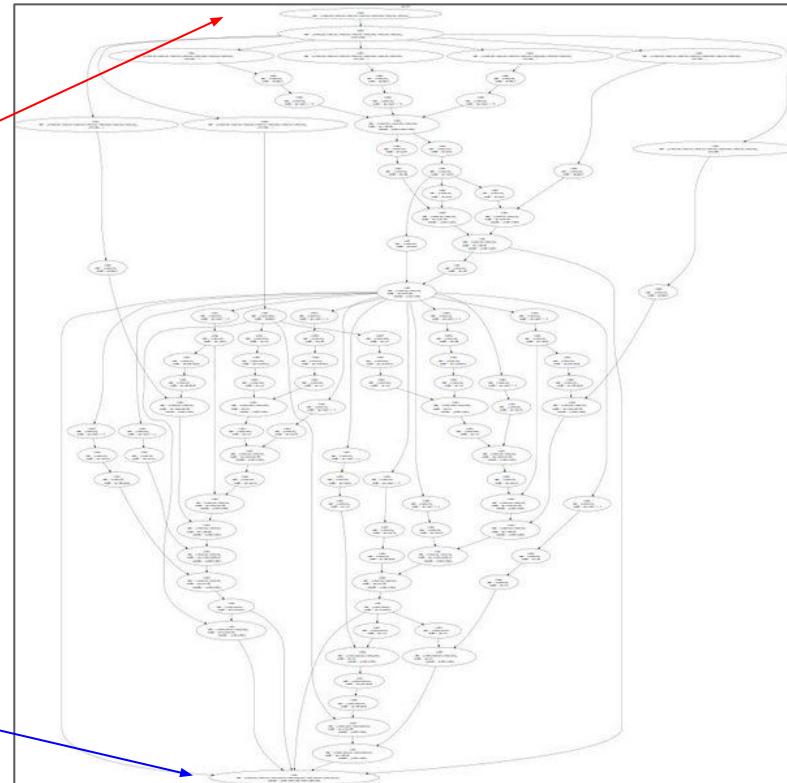


Figure reproduced with permission from a [Twitter post](#) by Andrej Karpathy.

Neural Turing Machine

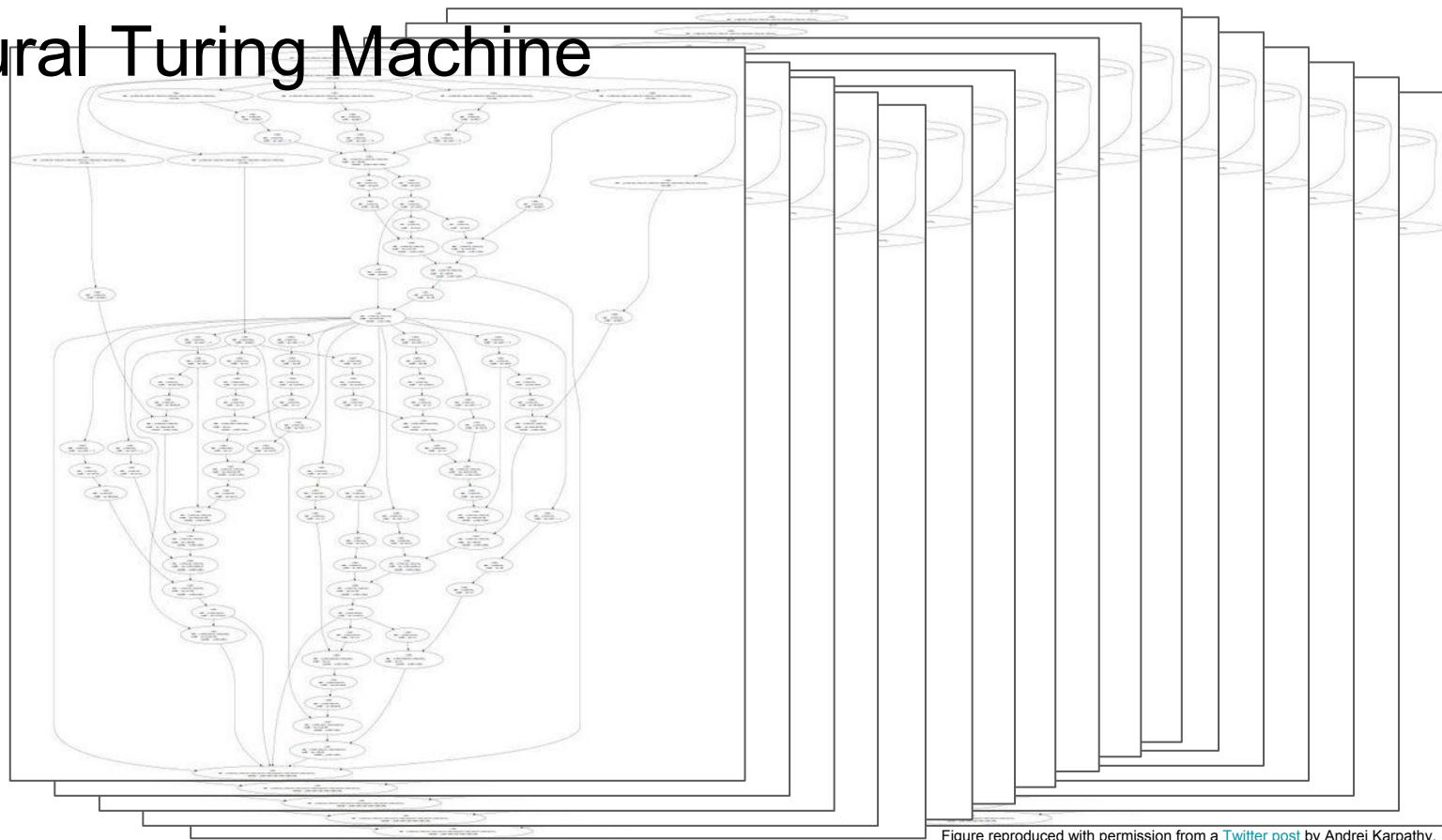


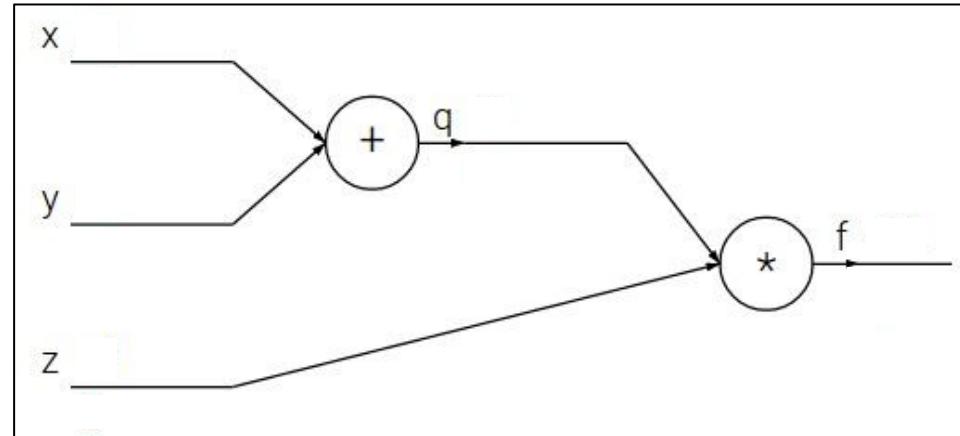
Figure reproduced with permission from a [Twitter post](#) by Andrej Karpathy.

Backpropagation: a simple example

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

Backpropagation: a simple example

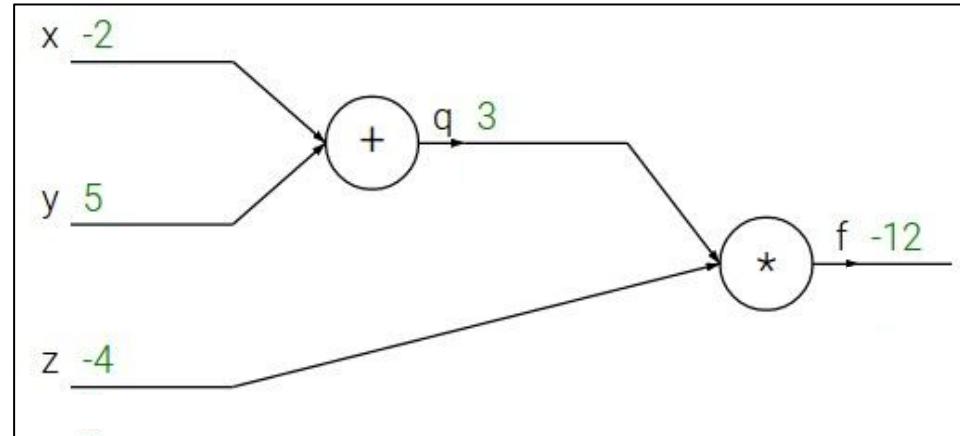
$$f(x, y, z) = (x + y)z$$



Backpropagation: a simple example

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

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



Backpropagation: a simple example

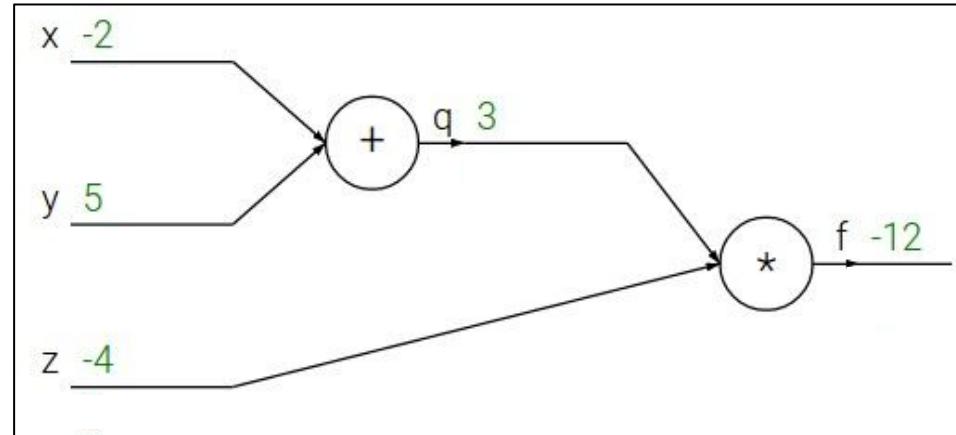
$$f(x, y, z) = (x + y)z$$

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

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

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



Backpropagation: a simple example

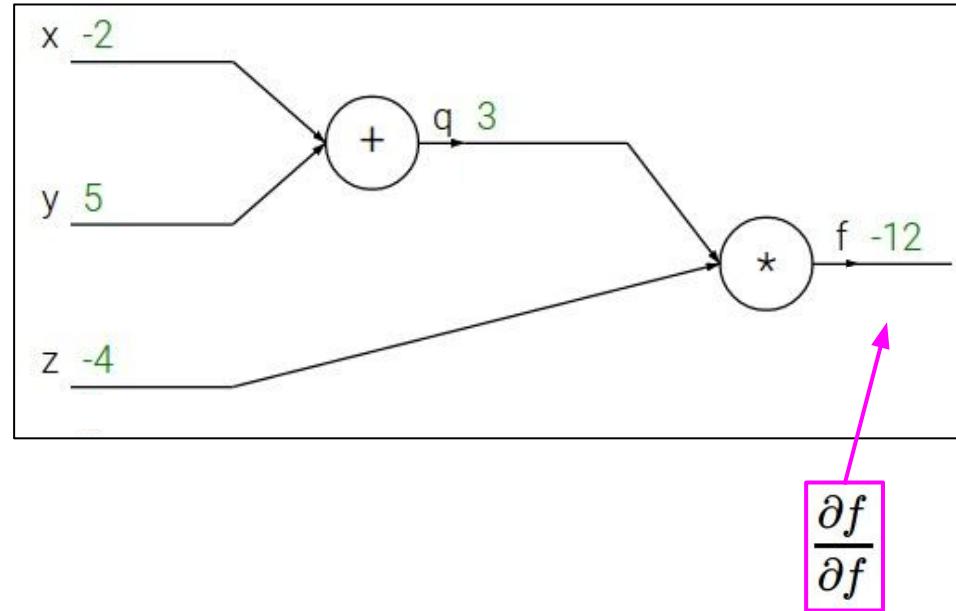
$$f(x, y, z) = (x + y)z$$

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

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

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



Backpropagation: a simple example

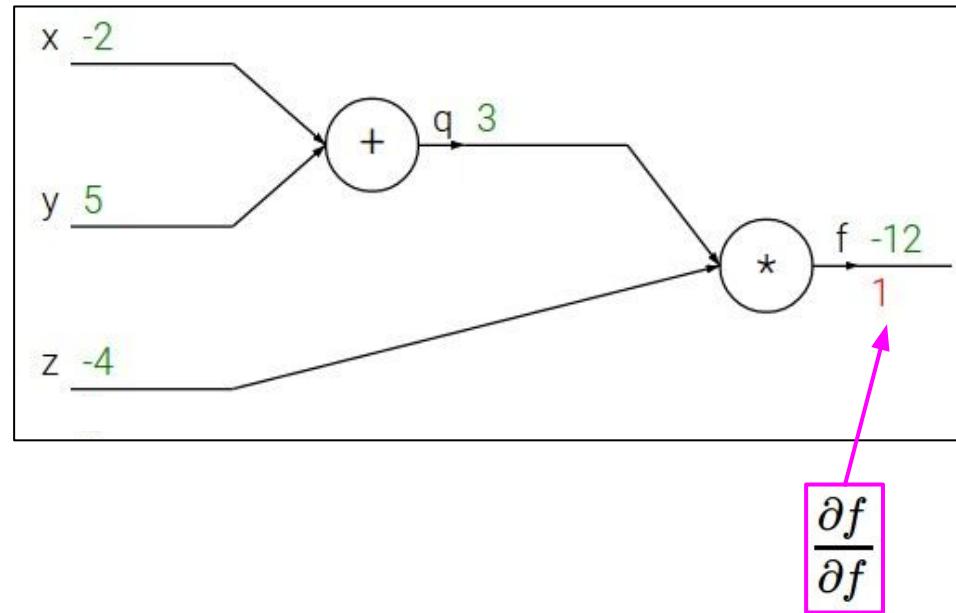
$$f(x, y, z) = (x + y)z$$

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

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

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



Backpropagation: a simple example

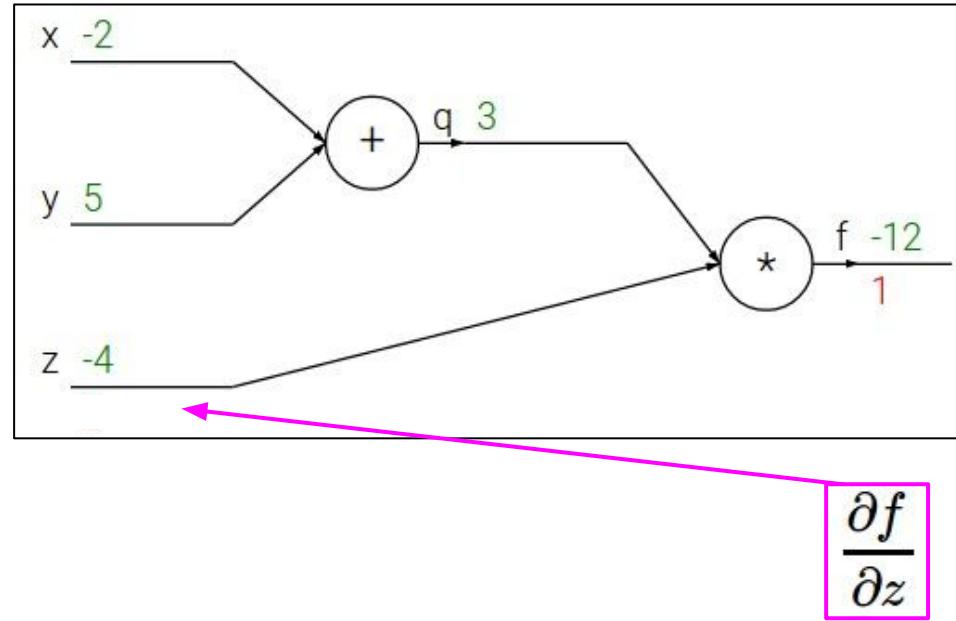
$$f(x, y, z) = (x + y)z$$

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

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

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



Backpropagation: a simple example

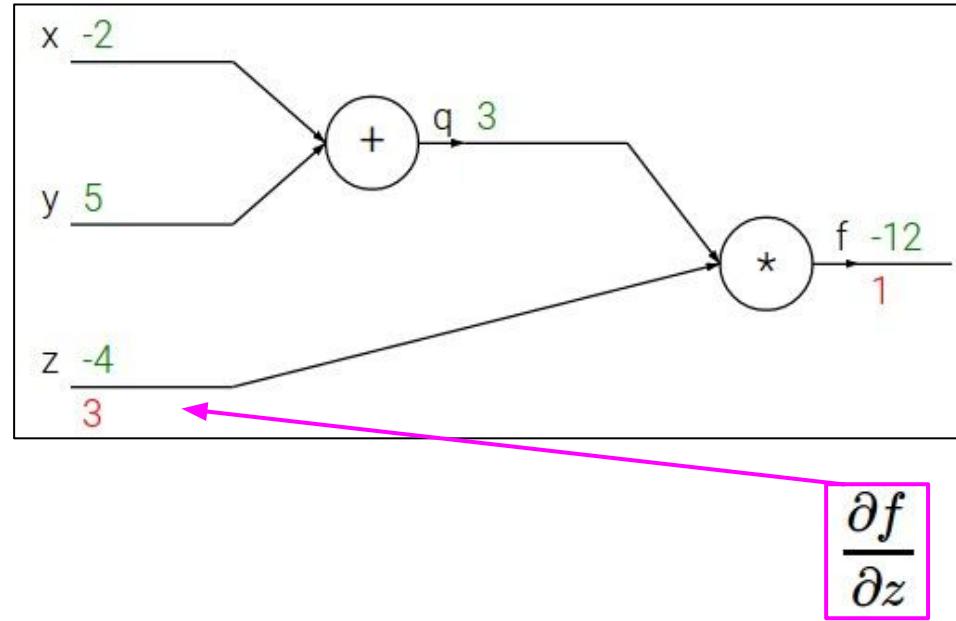
$$f(x, y, z) = (x + y)z$$

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

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

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



$$\frac{\partial f}{\partial z}$$

Backpropagation: a simple example

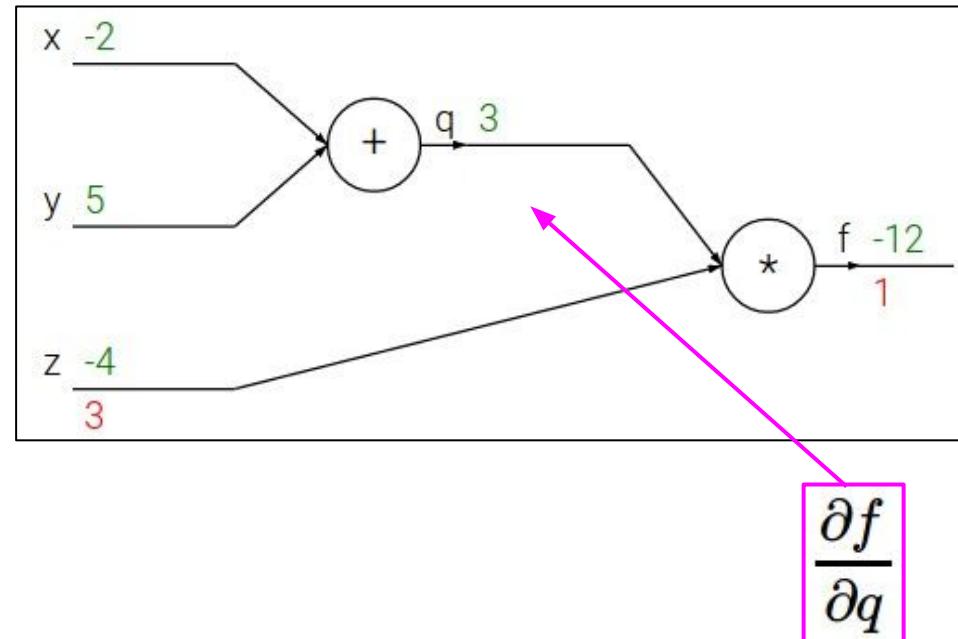
$$f(x, y, z) = (x + y)z$$

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

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

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



Backpropagation: a simple example

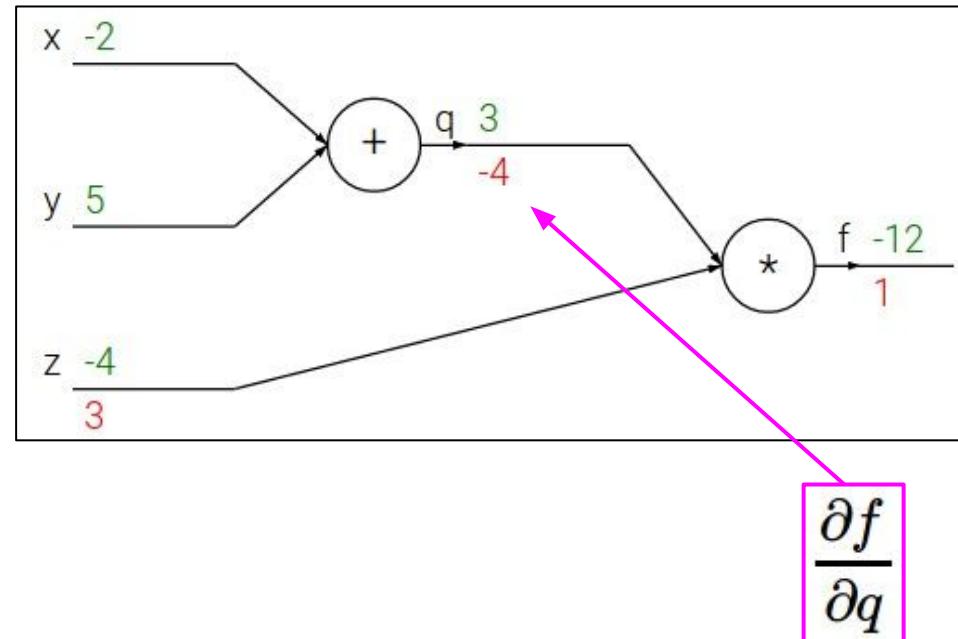
$$f(x, y, z) = (x + y)z$$

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

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

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



Backpropagation: a simple example

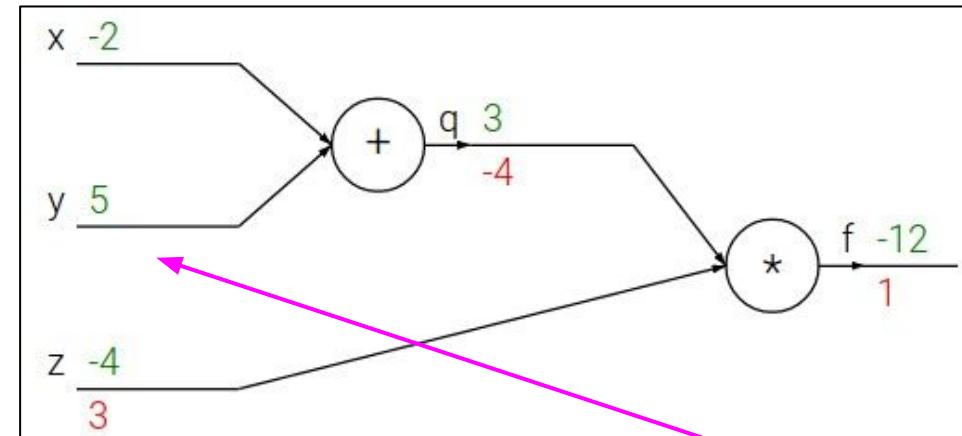
$$f(x, y, z) = (x + y)z$$

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

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

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



Chain rule:

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

Upstream
gradient

Local
gradient

$$\frac{\partial f}{\partial y}$$

Backpropagation: a simple example

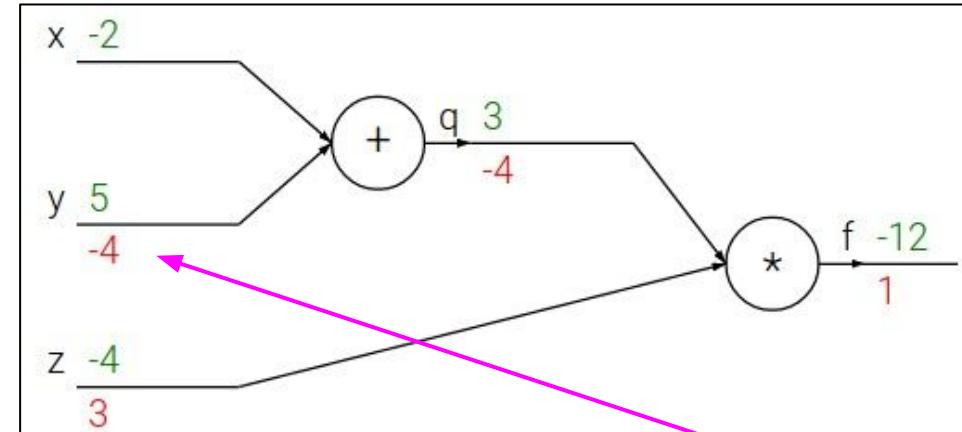
$$f(x, y, z) = (x + y)z$$

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

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

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



Chain rule:

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

Upstream
gradient

Local
gradient

$$\frac{\partial f}{\partial y}$$

Backpropagation: a simple example

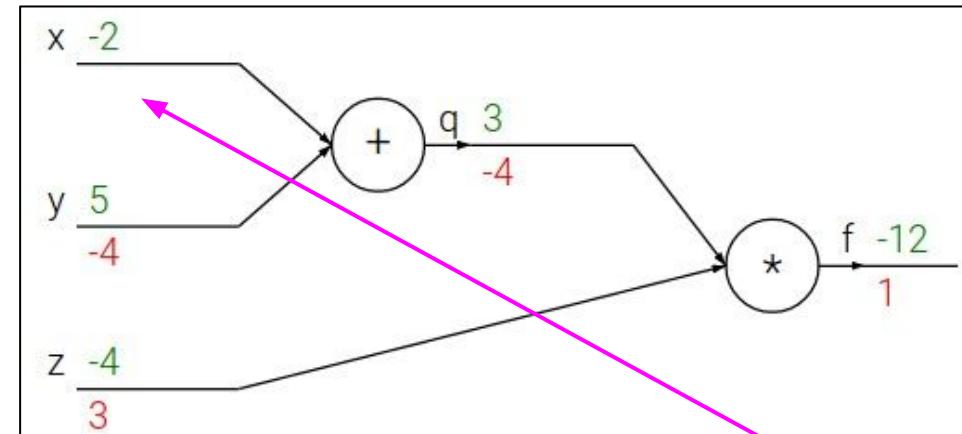
$$f(x, y, z) = (x + y)z$$

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

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

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



Chain rule:

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

Upstream
gradient

Local
gradient

Backpropagation: a simple example

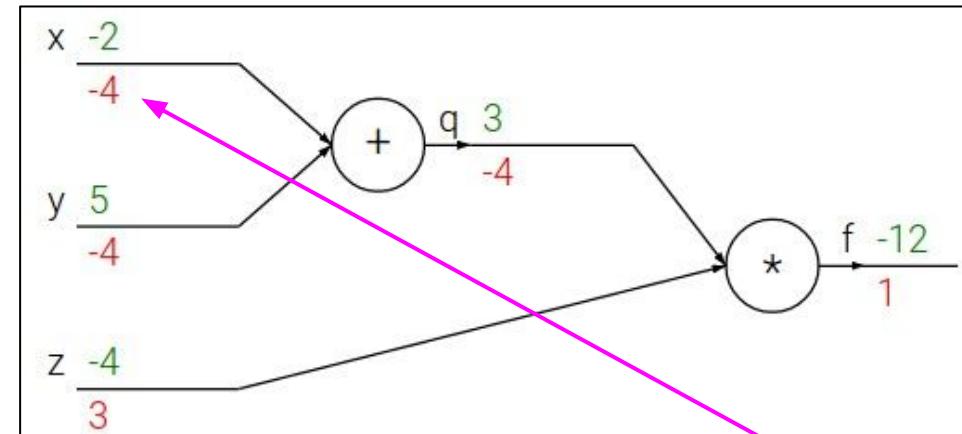
$$f(x, y, z) = (x + y)z$$

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

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

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

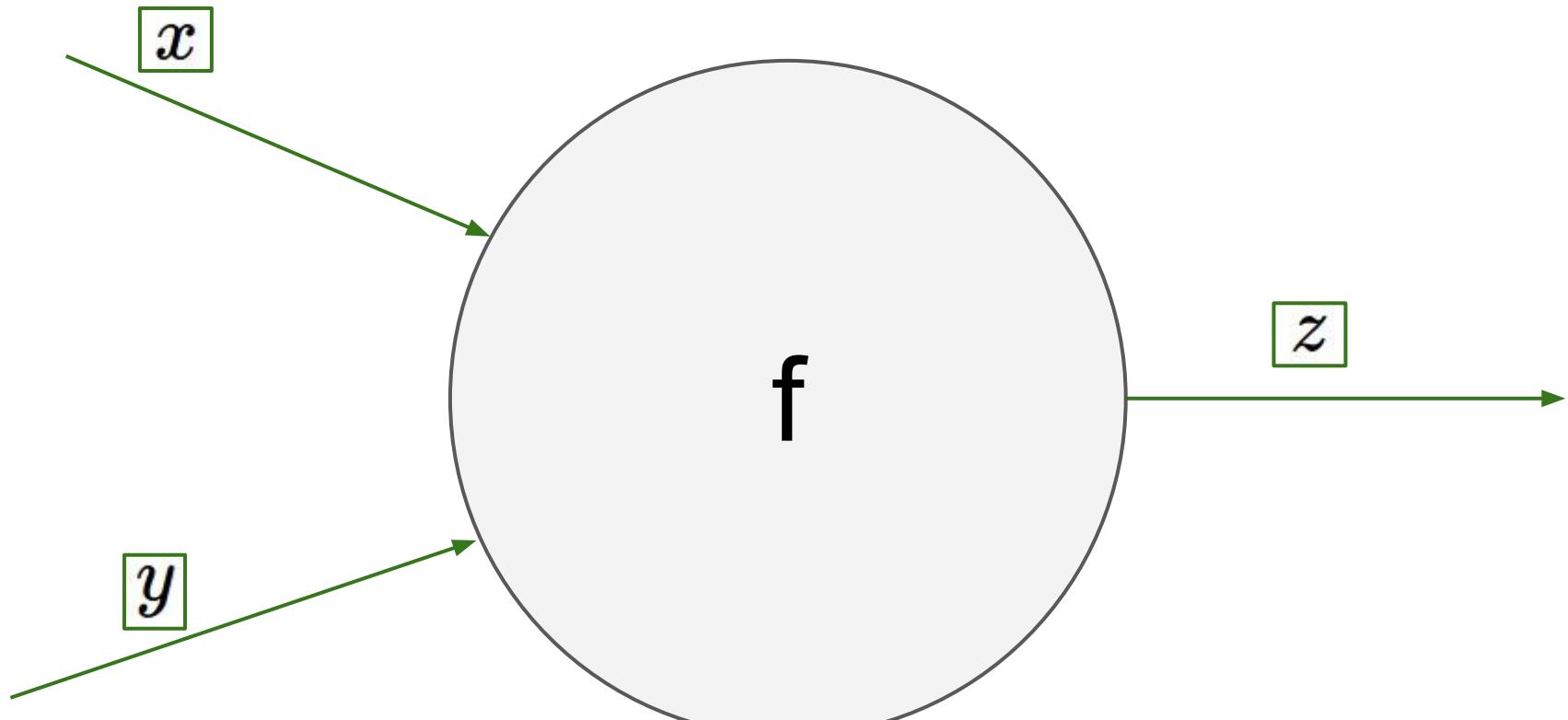


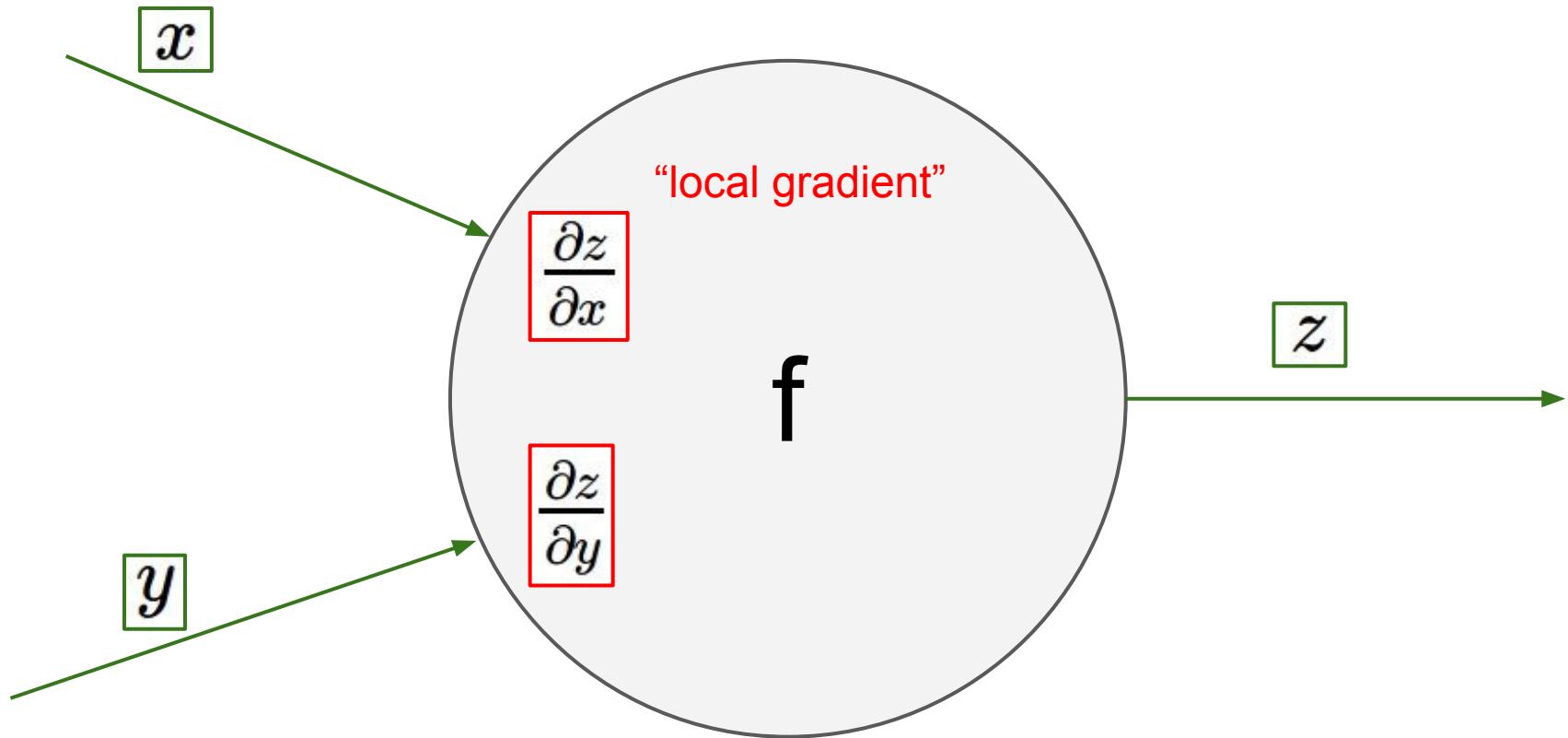
Chain rule:

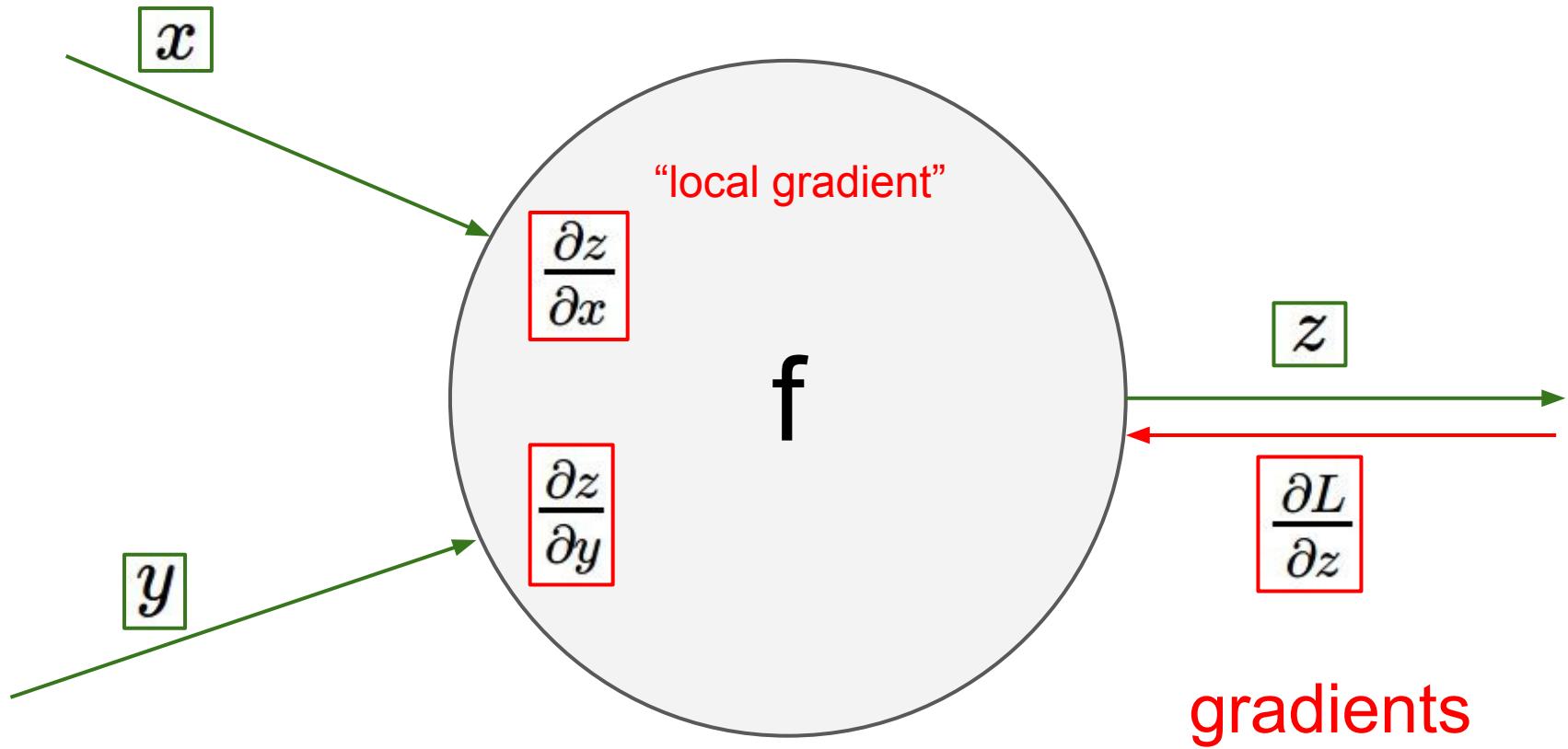
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

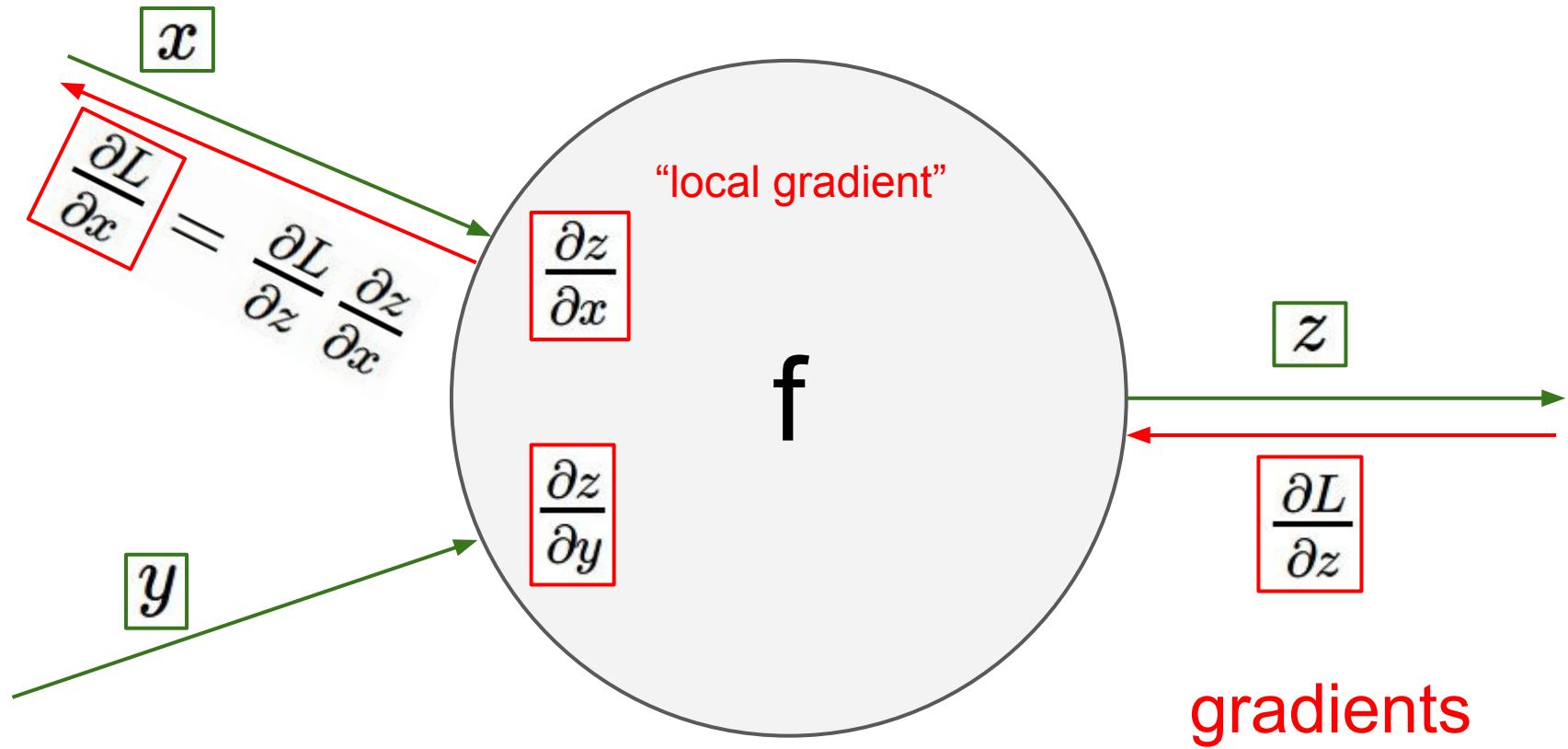
Upstream
gradient

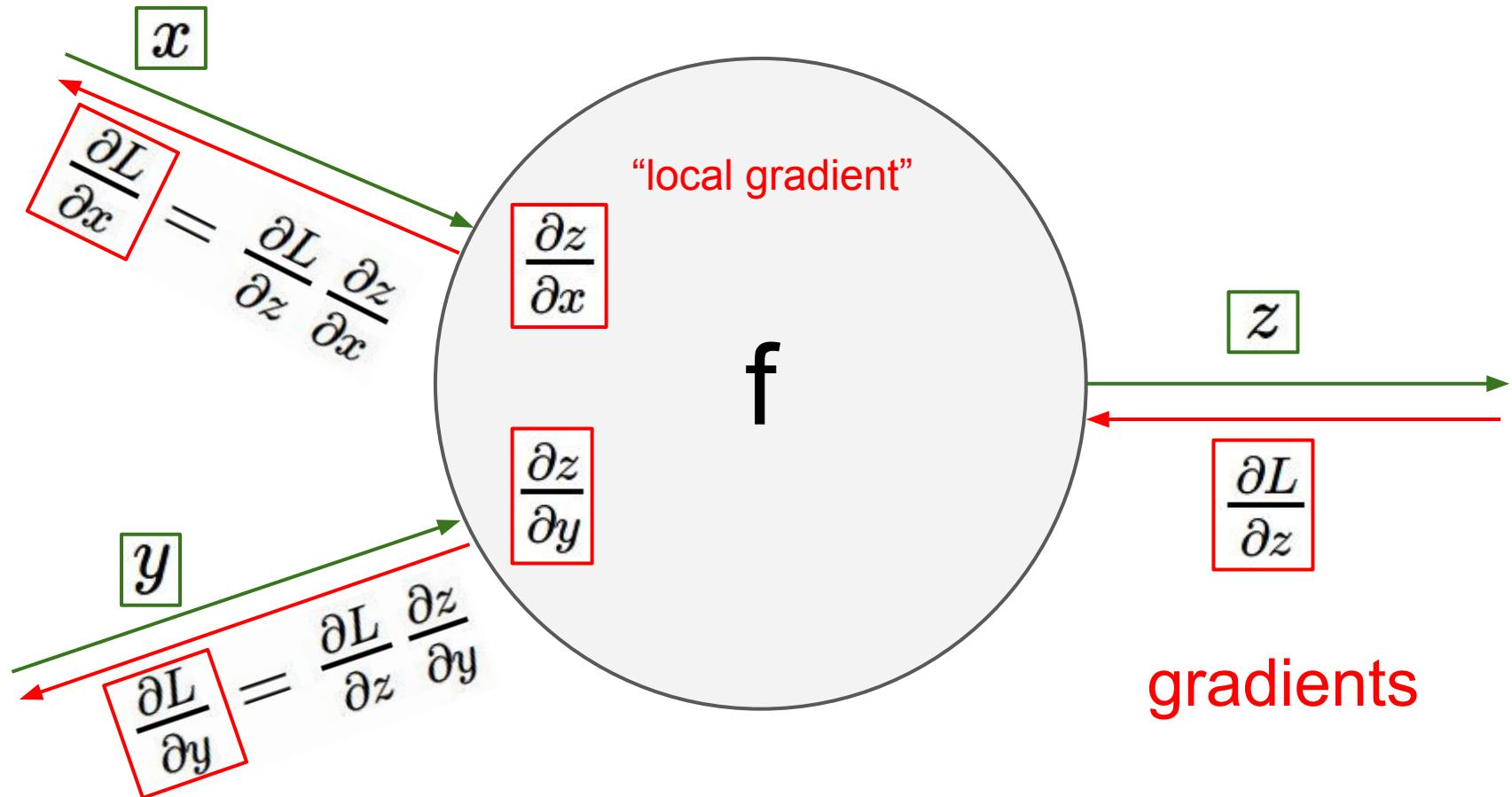
Local
gradient

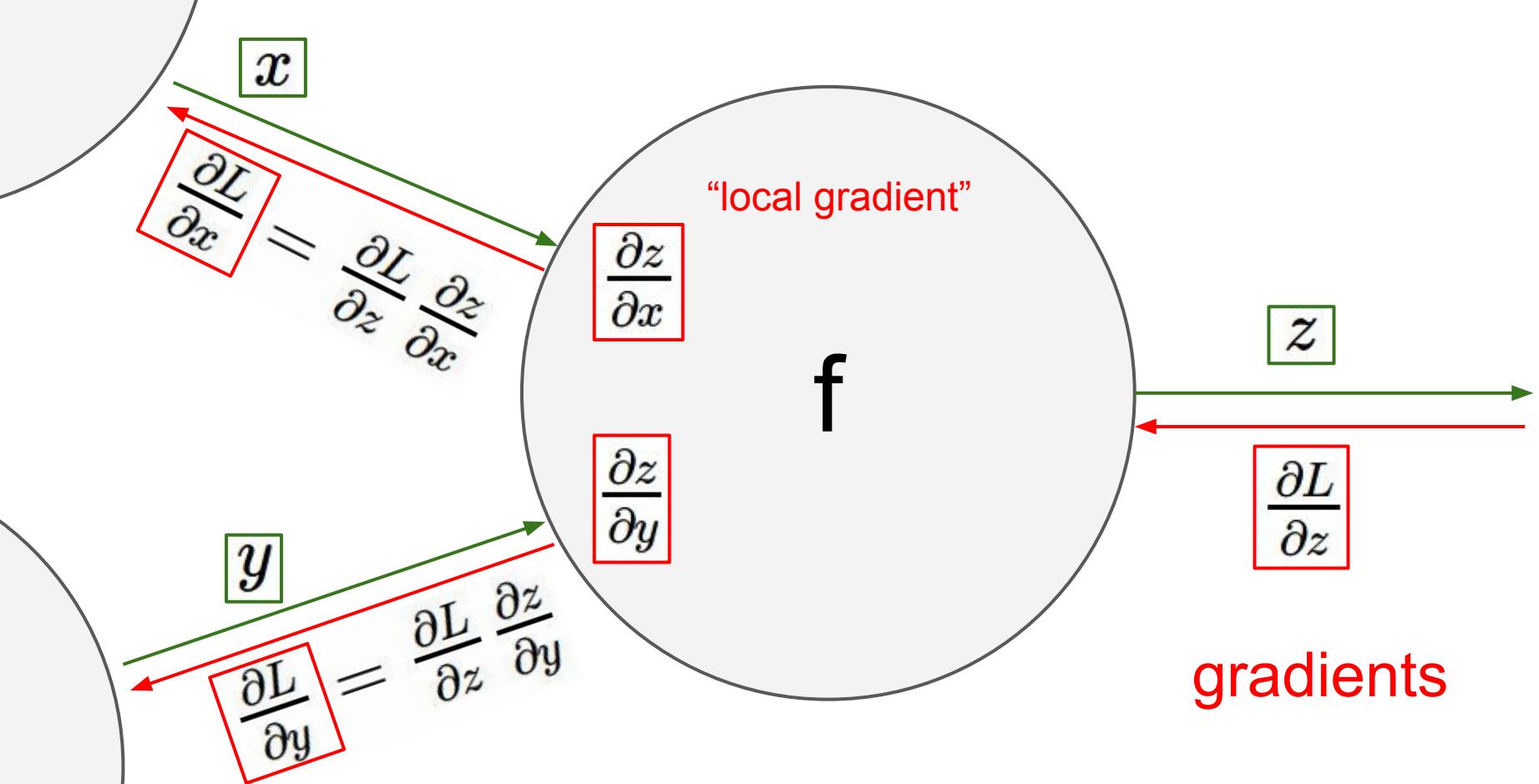






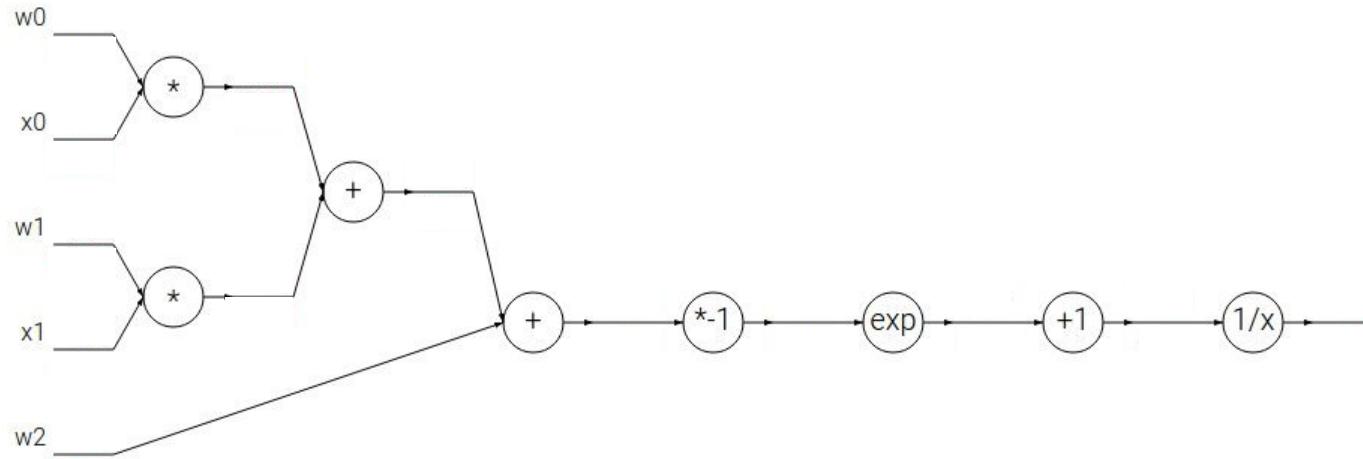






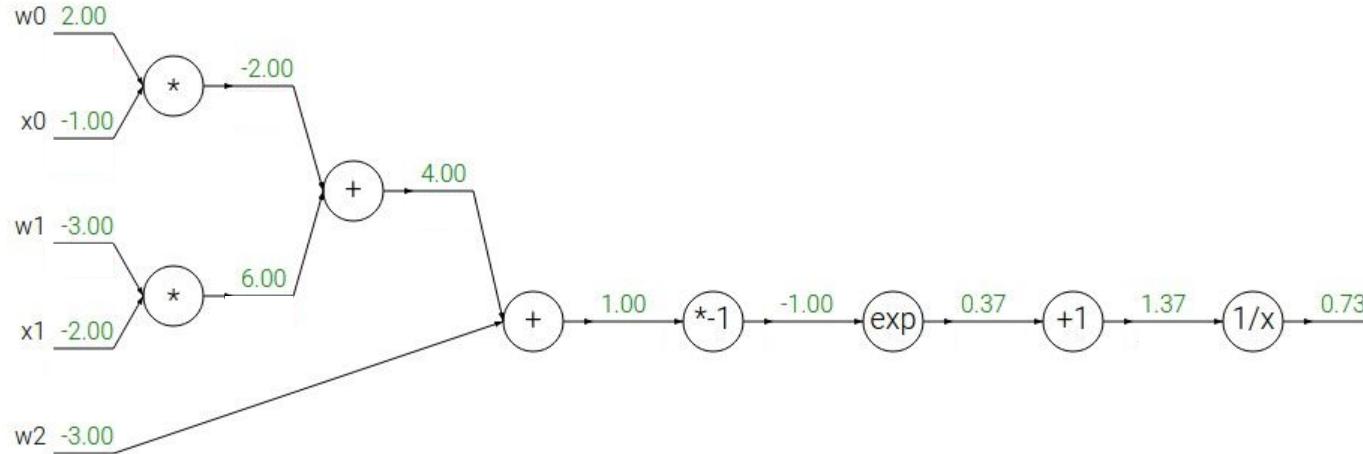
Another example:

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



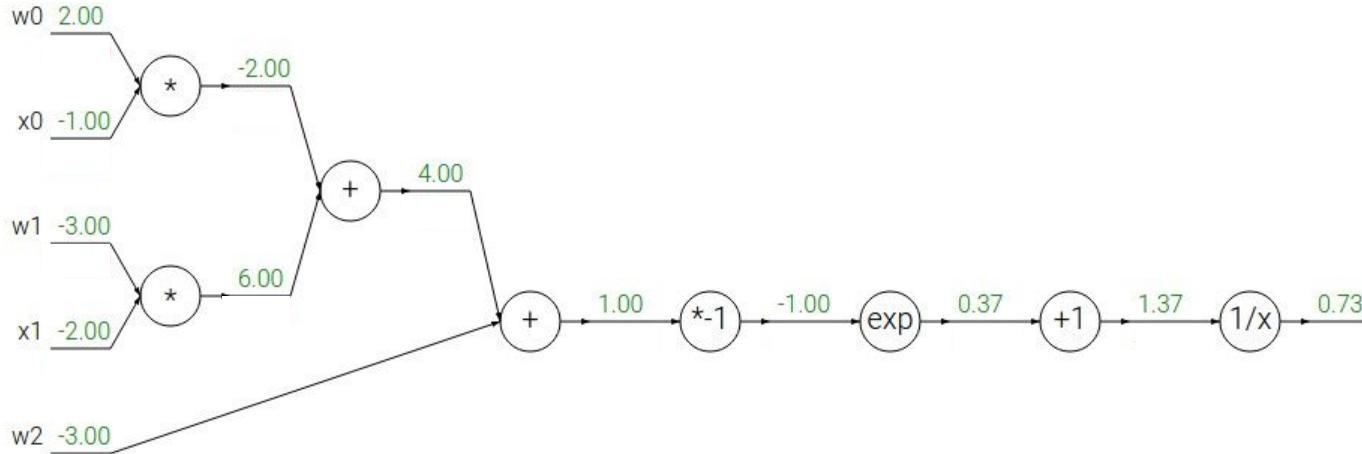
Another example:

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



Another example:

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



$$f(x) = e^x$$

\rightarrow

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

\rightarrow

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

$$f_c(x) = c + x$$

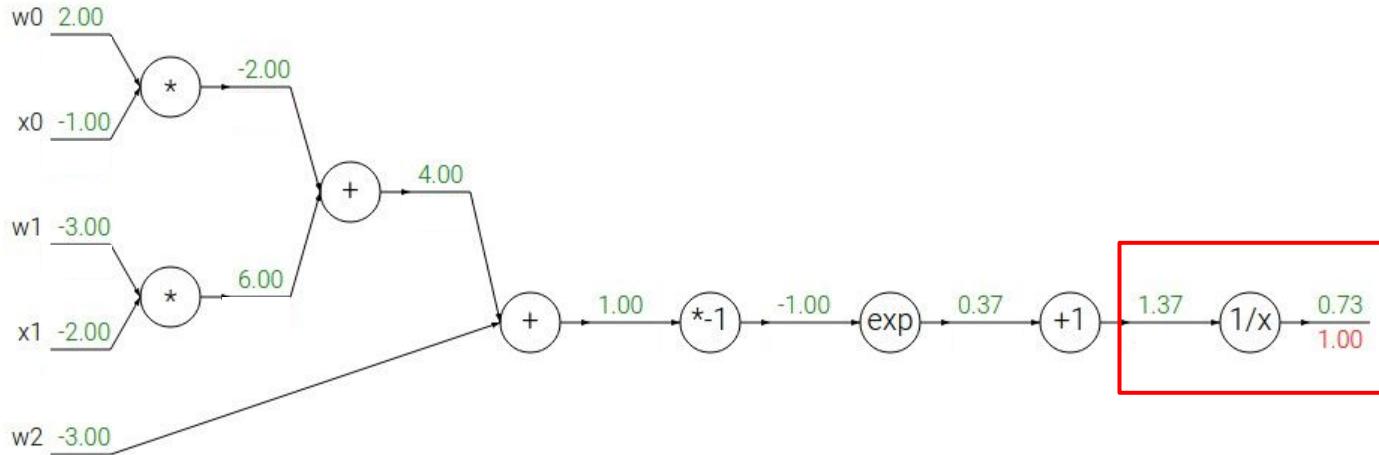
\rightarrow

$$\frac{df}{dx} = -1/x^2$$

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

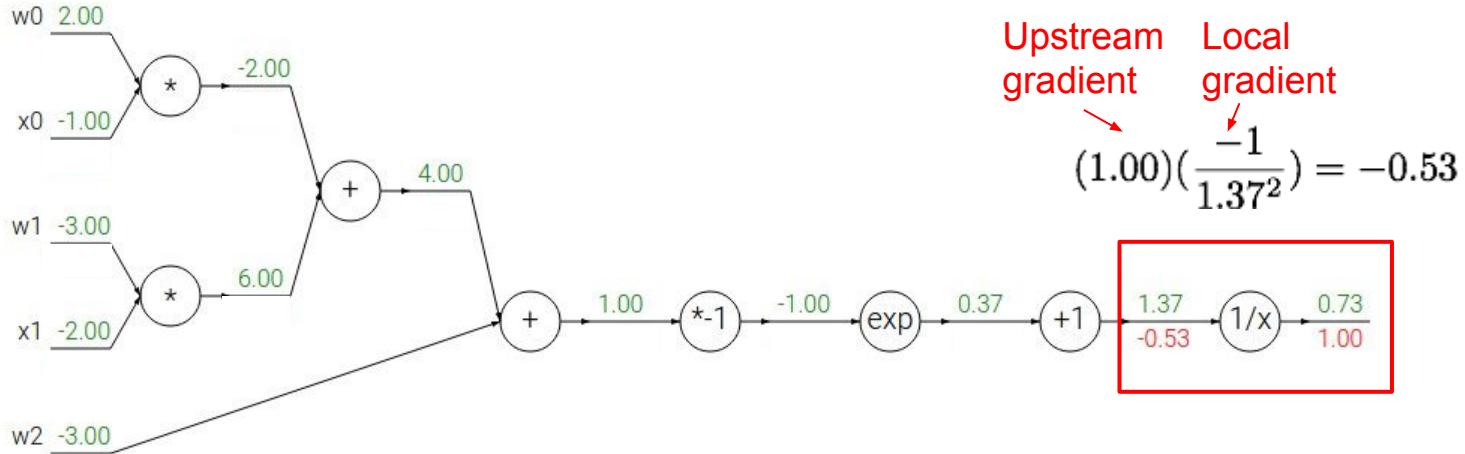
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

\rightarrow

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

\rightarrow

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

\rightarrow

$$\frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x$$

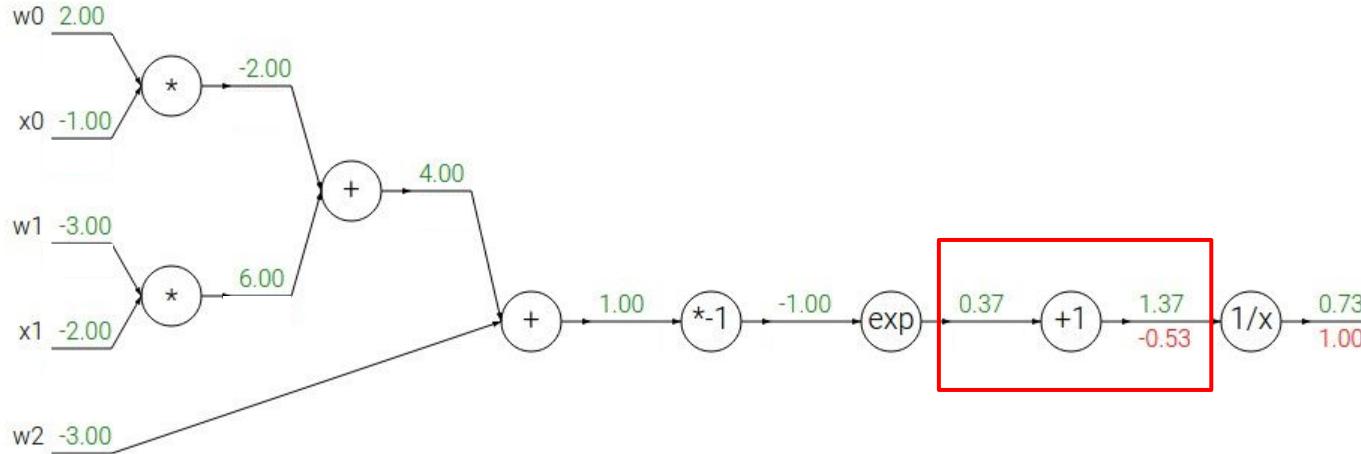
Upstream gradient

\rightarrow

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

\rightarrow

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

\rightarrow

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

\rightarrow

$$\frac{df}{dx} = -1/x^2$$

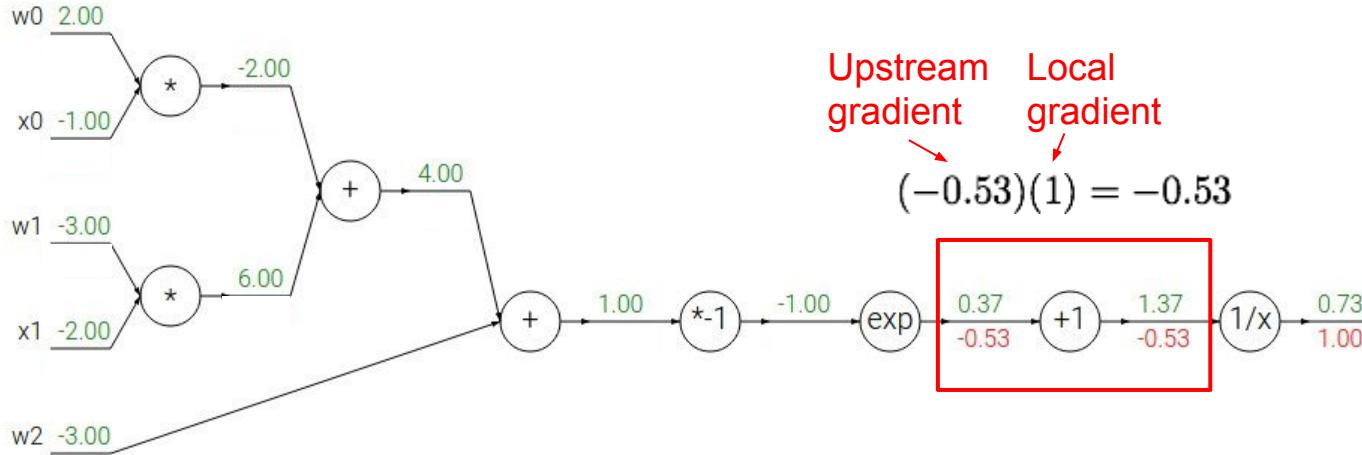
$$f_c(x) = c + x$$

\rightarrow

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

\rightarrow

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

\rightarrow

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

\rightarrow

$$\frac{df}{dx} = -1/x^2$$

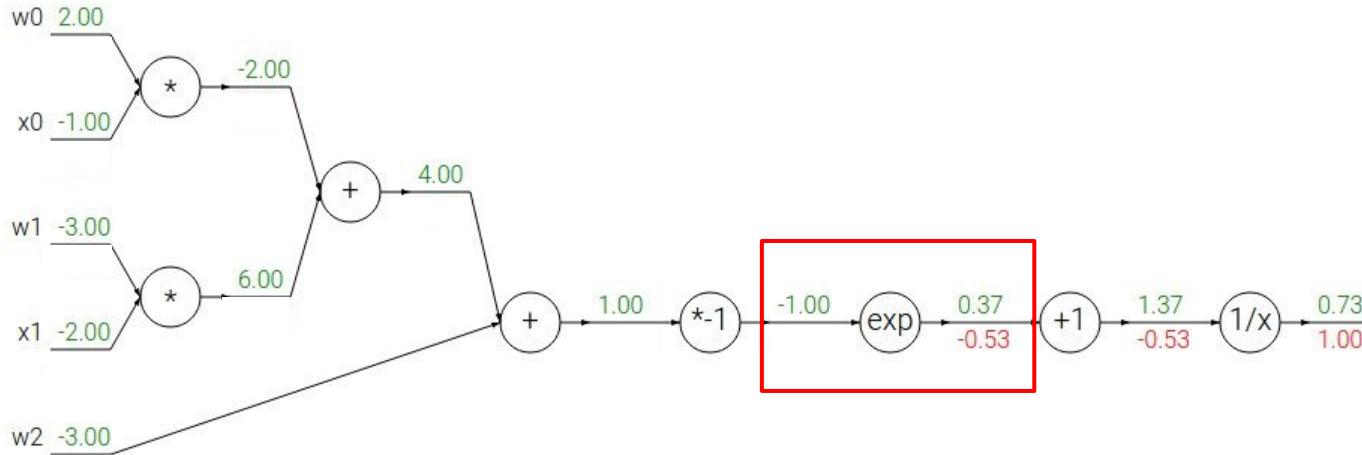
$$f_c(x) = c + x$$

\rightarrow

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

\rightarrow

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

\rightarrow

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

$$f_c(x) = c + x$$

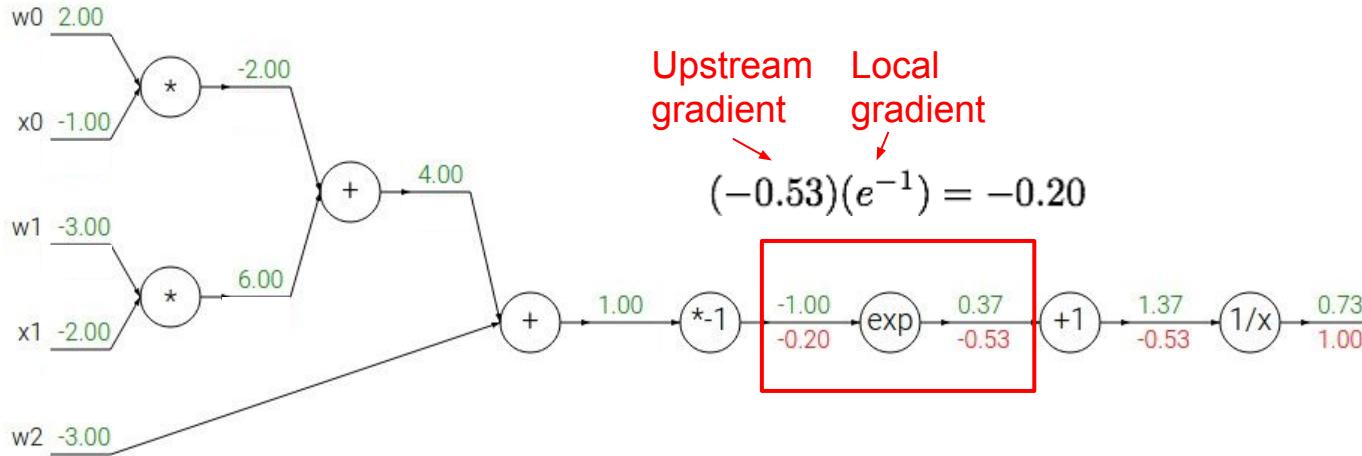
\rightarrow

$$\frac{df}{dx} = -1/x^2$$

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

\rightarrow

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

\rightarrow

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

\rightarrow

$$\frac{df}{dx} = -1/x^2$$

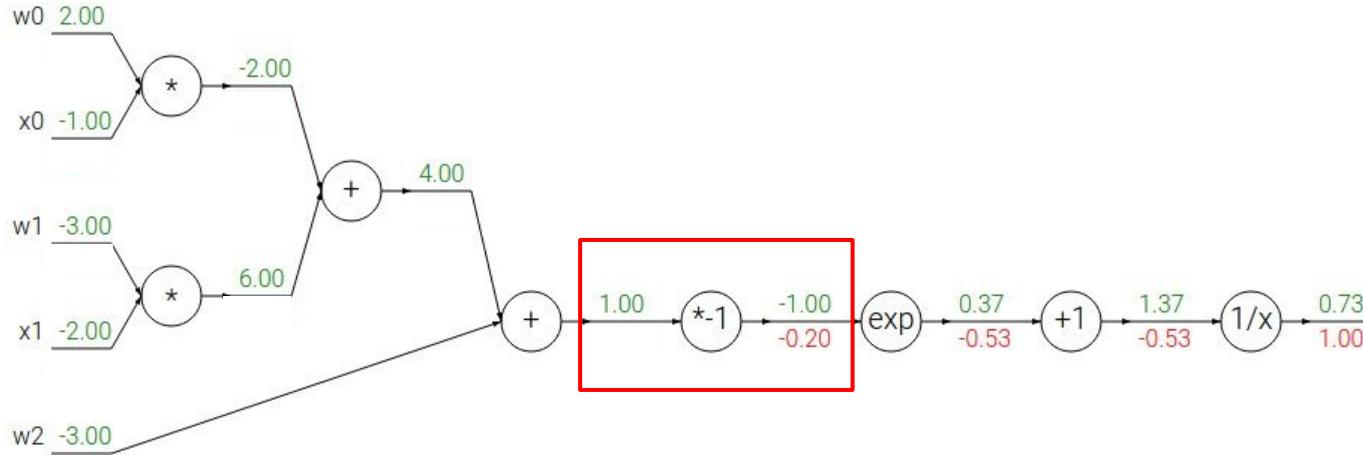
$$f_c(x) = c + x$$

\rightarrow

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

\rightarrow

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

\rightarrow

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

\rightarrow

$$\frac{df}{dx} = -1/x^2$$

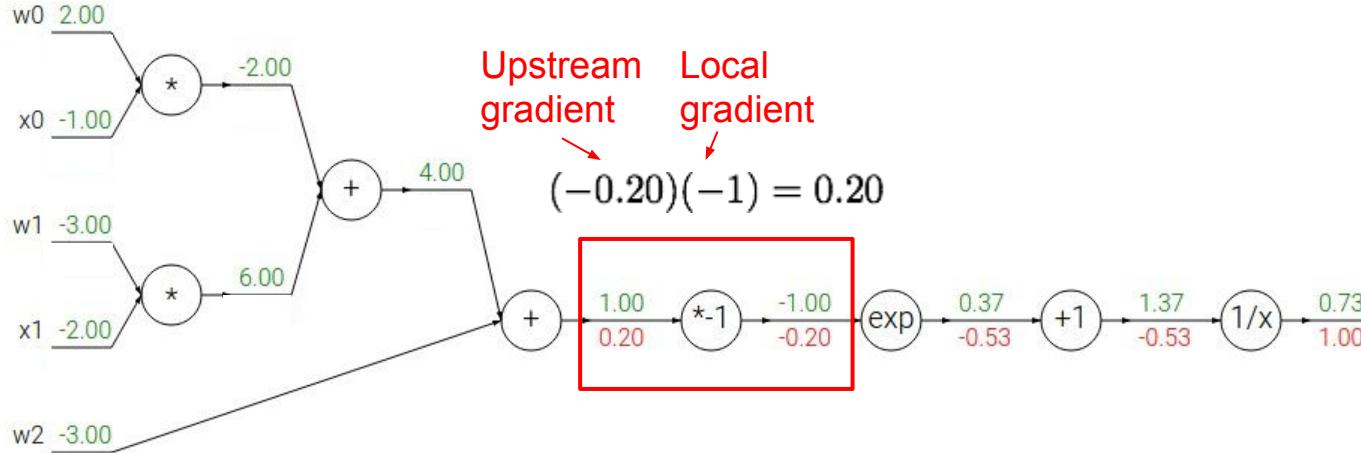
$$f_c(x) = c + x$$

\rightarrow

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

\rightarrow

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

\rightarrow

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

\rightarrow

$$\frac{df}{dx} = -1/x^2$$

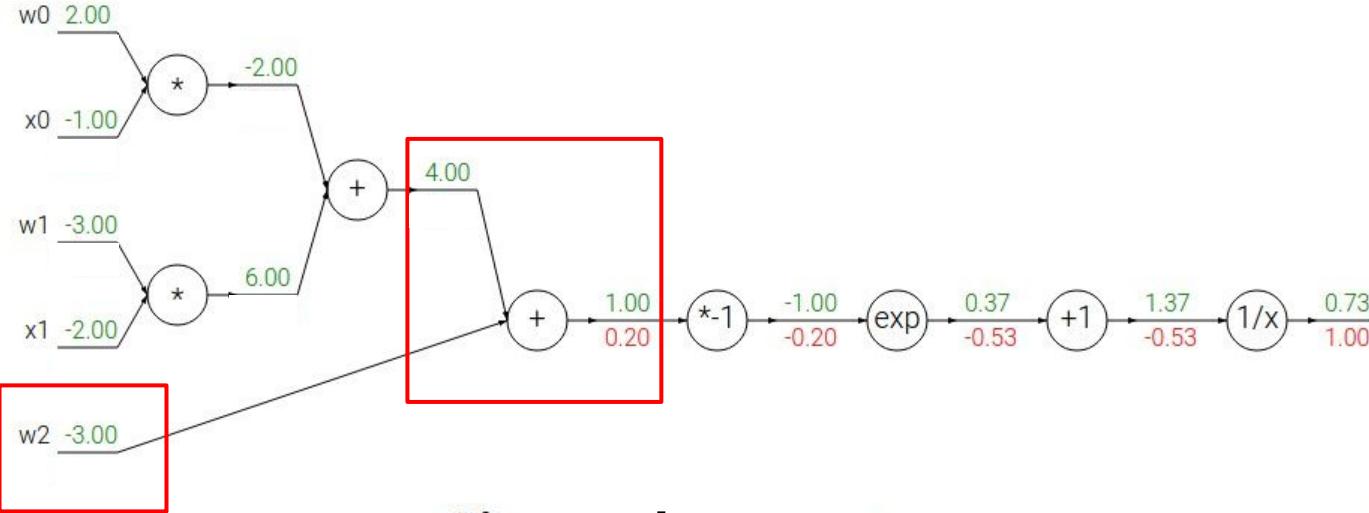
$$f_c(x) = c + x$$

\rightarrow

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

\rightarrow

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

\rightarrow

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

$$f_c(x) = c + x$$

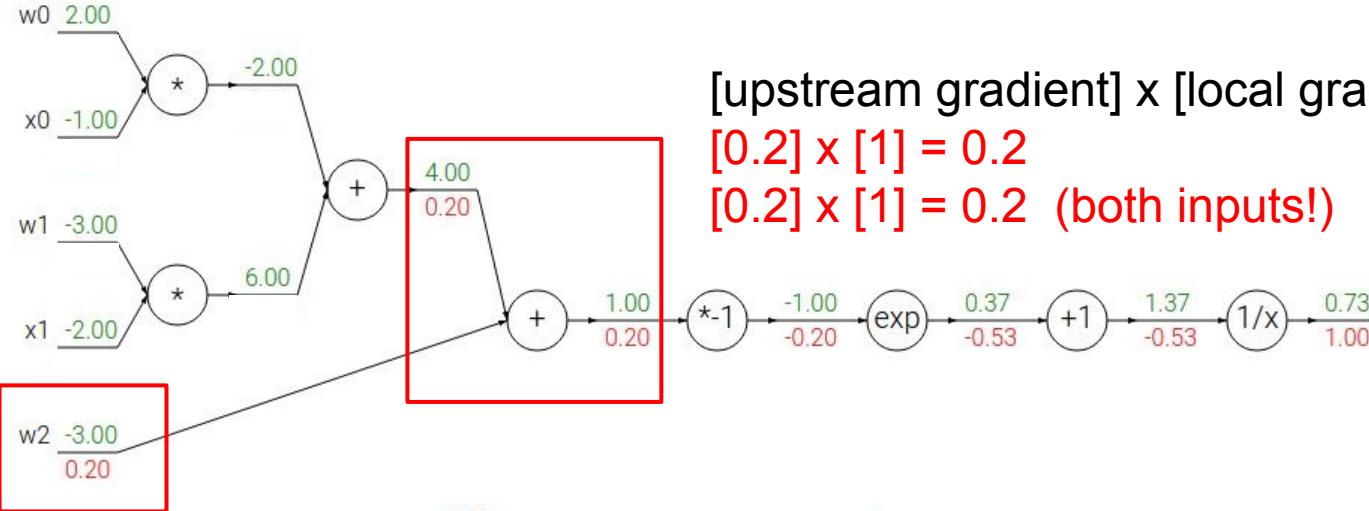
\rightarrow

$$\frac{df}{dx} = -1/x^2$$

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

\rightarrow

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

\rightarrow

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

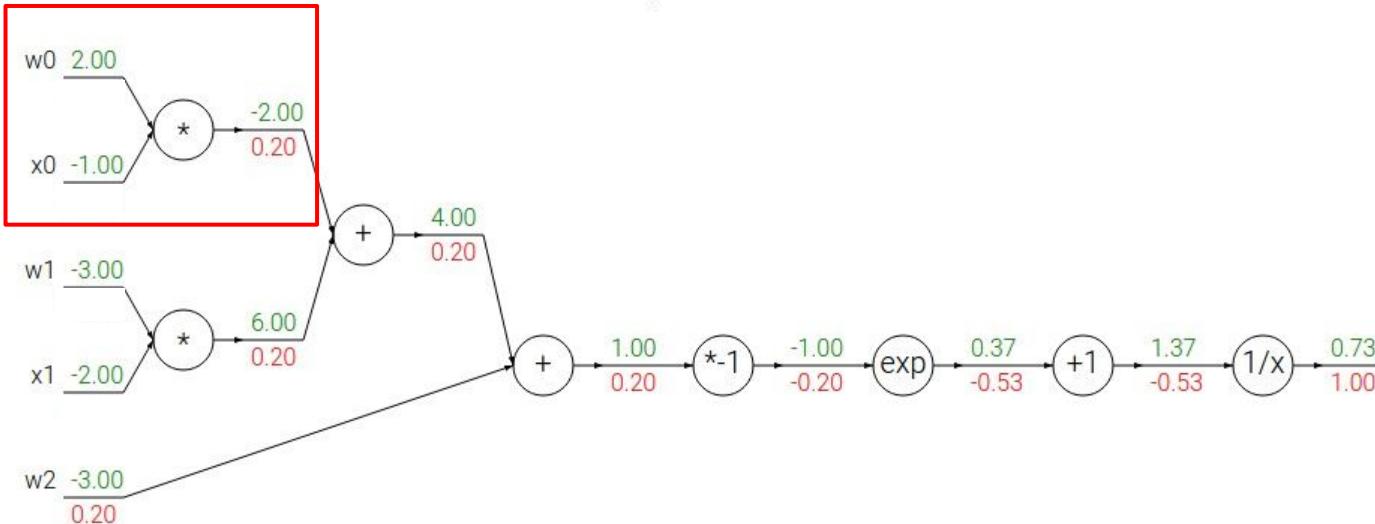
$$f_c(x) = c + x$$

$$\frac{df}{dx} = -1/x^2$$

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

\rightarrow

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

\rightarrow

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

$$f_c(x) = c + x$$

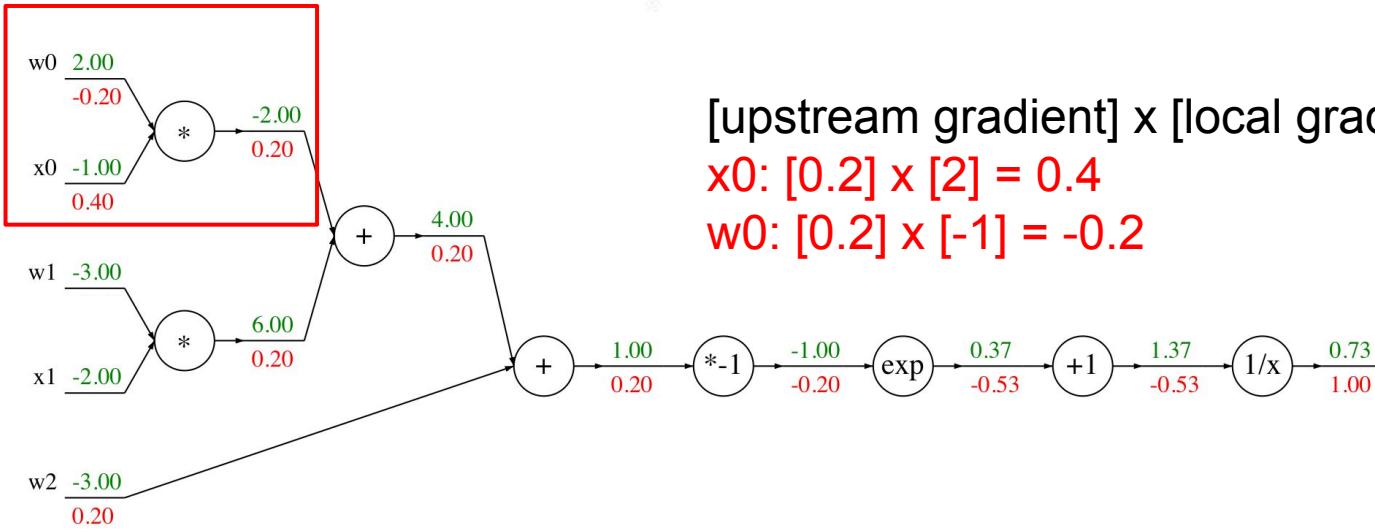
\rightarrow

$$\frac{df}{dx} = -1/x^2$$

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

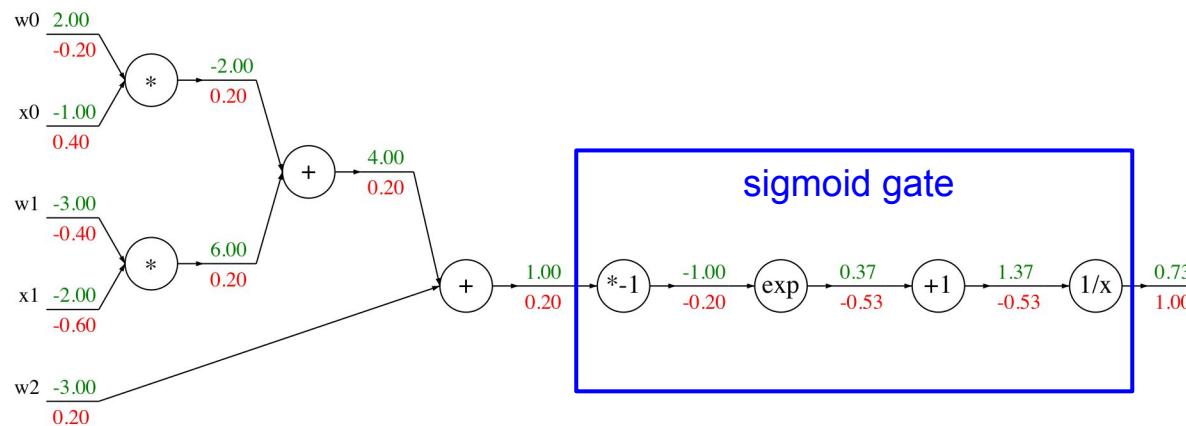
$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!

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

sigmoid function

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left(\frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x))\sigma(x)$$



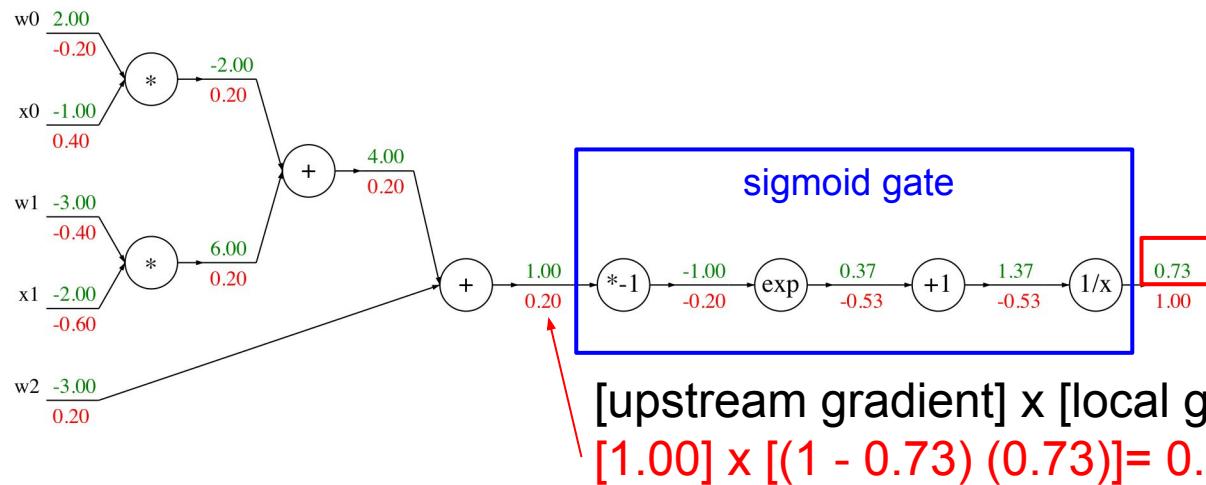
$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!

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

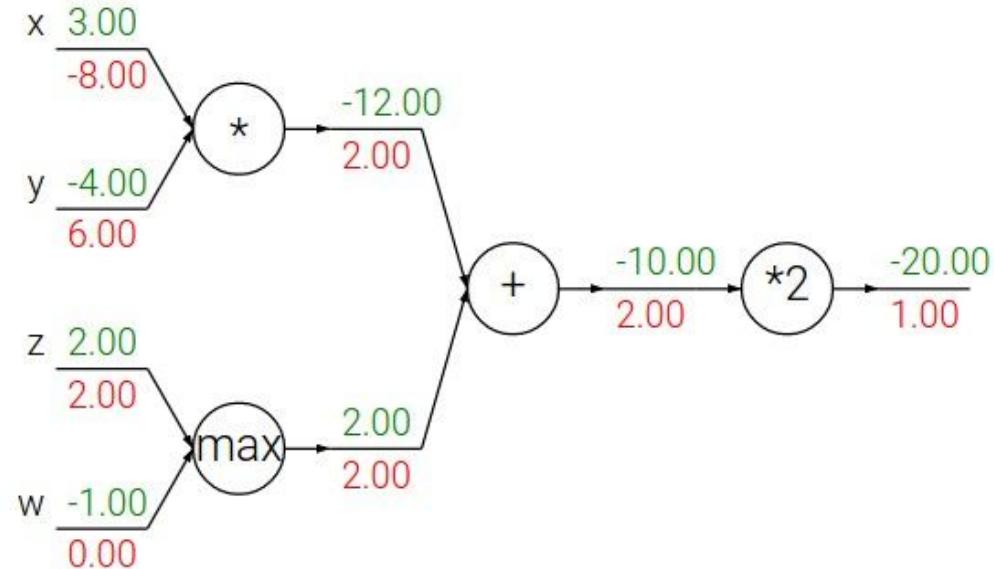
sigmoid function

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left(\frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x))\sigma(x)$$



Patterns in backward flow

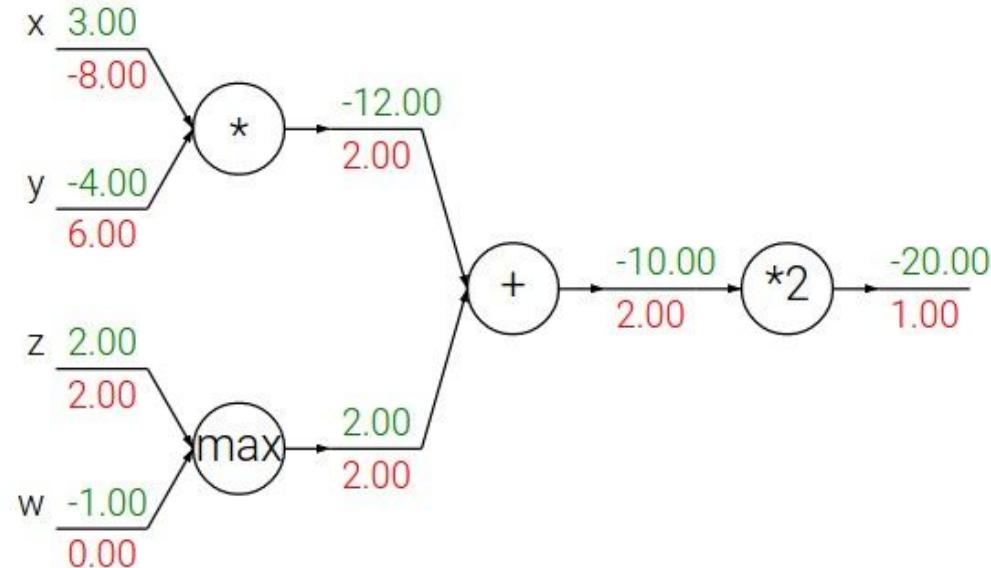
add gate: gradient distributor



Patterns in backward flow

add gate: gradient distributor

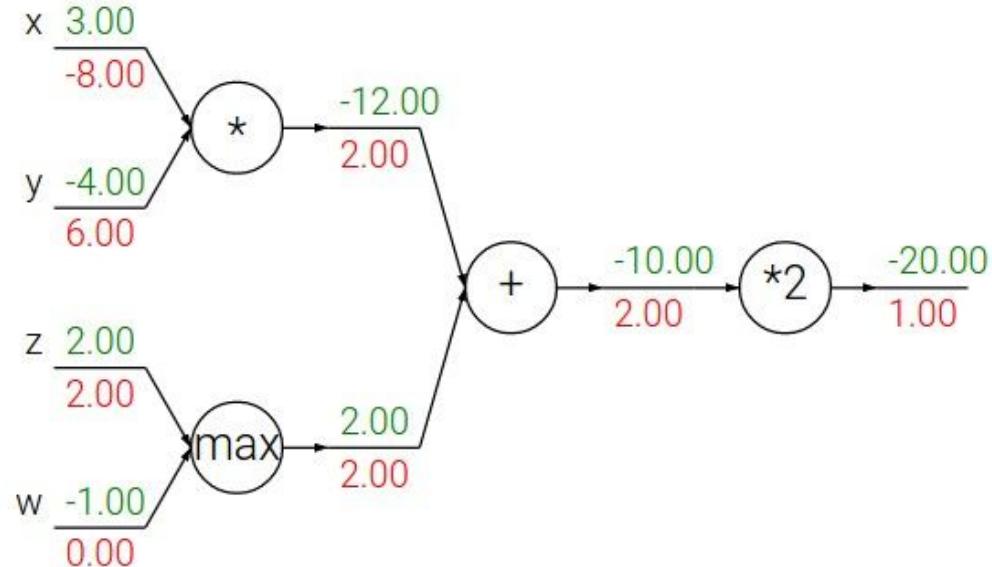
Q: What is a **max** gate?



Patterns in backward flow

add gate: gradient distributor

max gate: gradient router

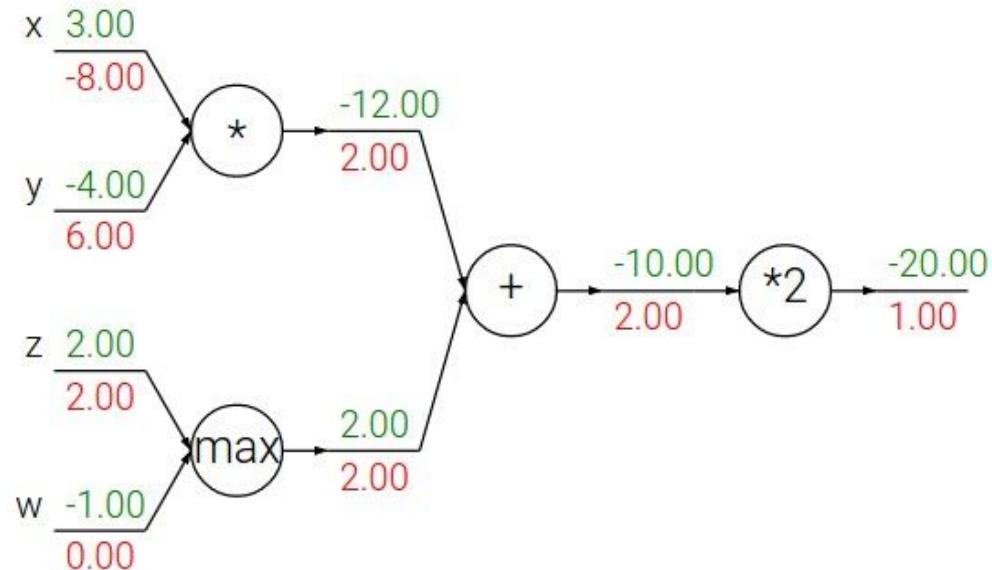


Patterns in backward flow

add gate: gradient distributor

max gate: gradient router

Q: What is a **mul** gate?

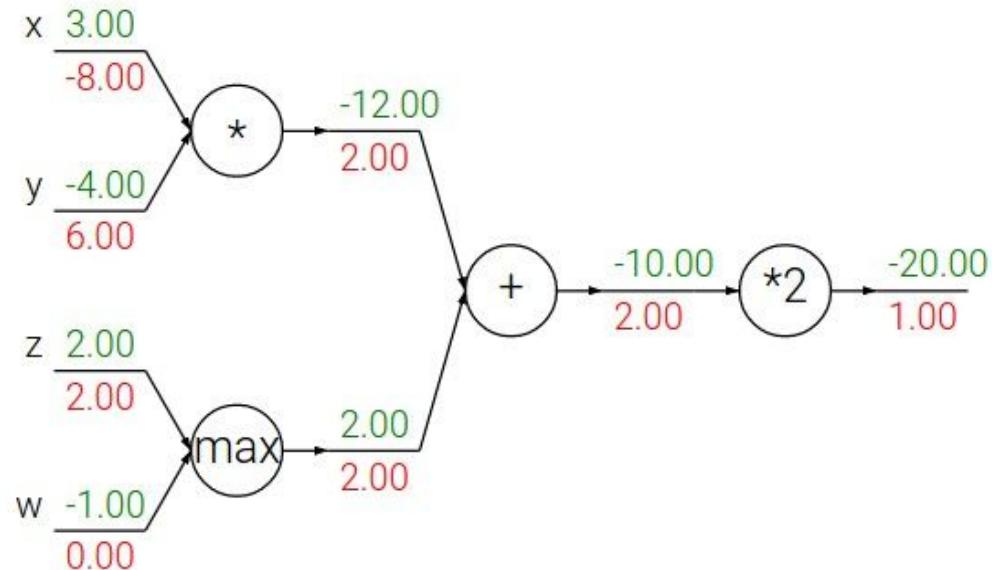


Patterns in backward flow

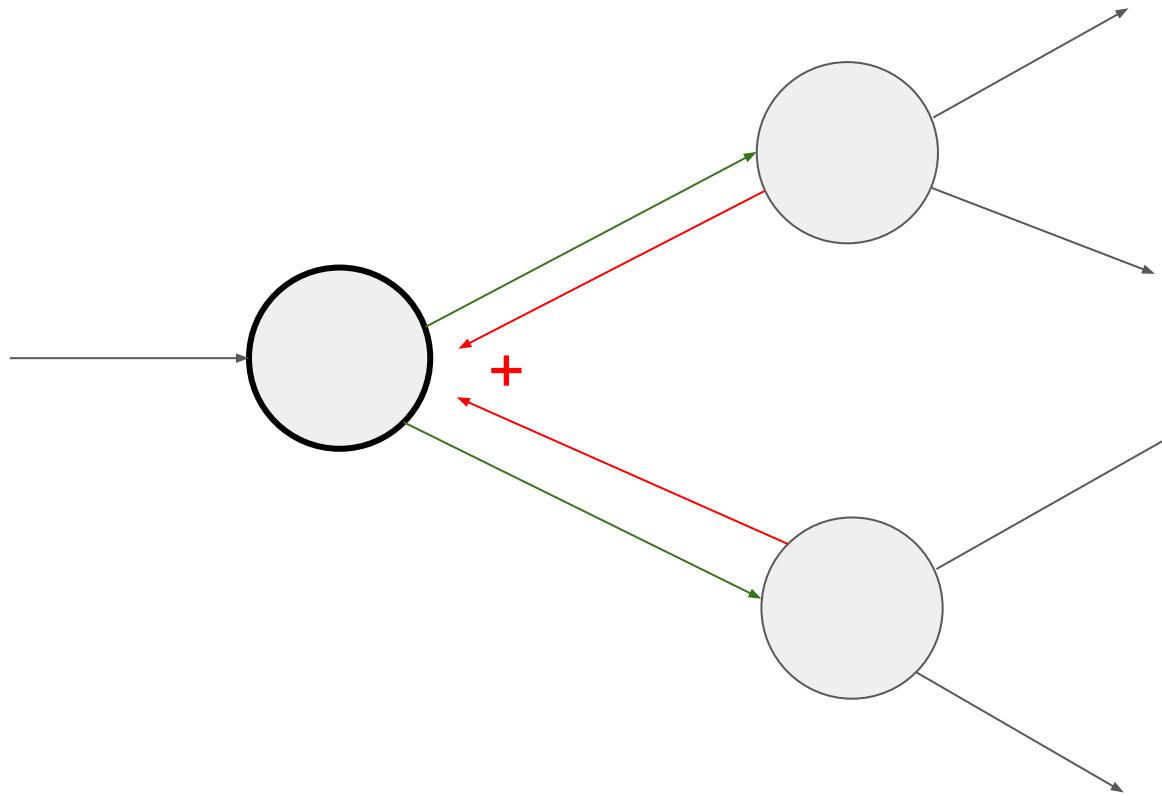
add gate: gradient distributor

max gate: gradient router

mul gate: gradient switcher



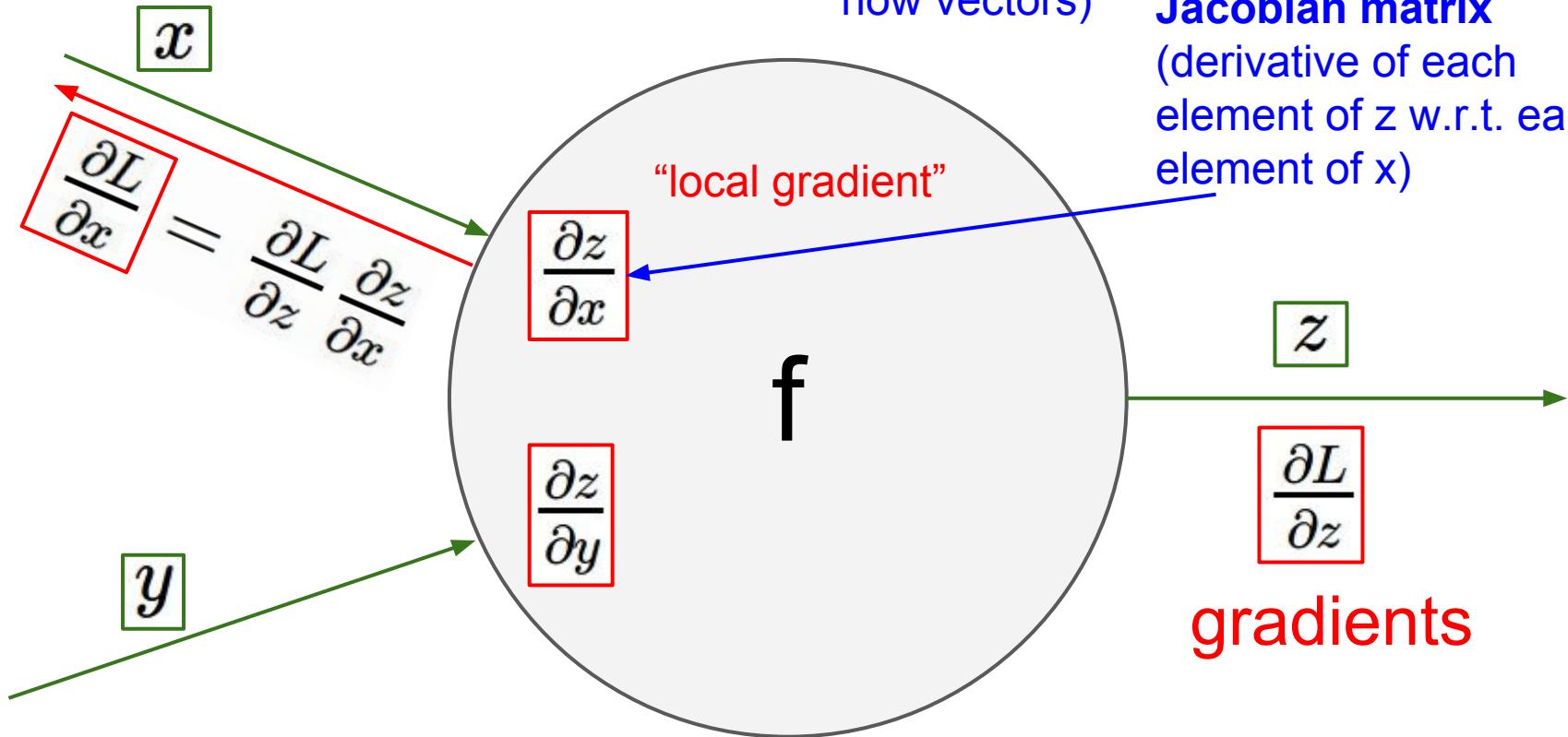
Gradients add at branches



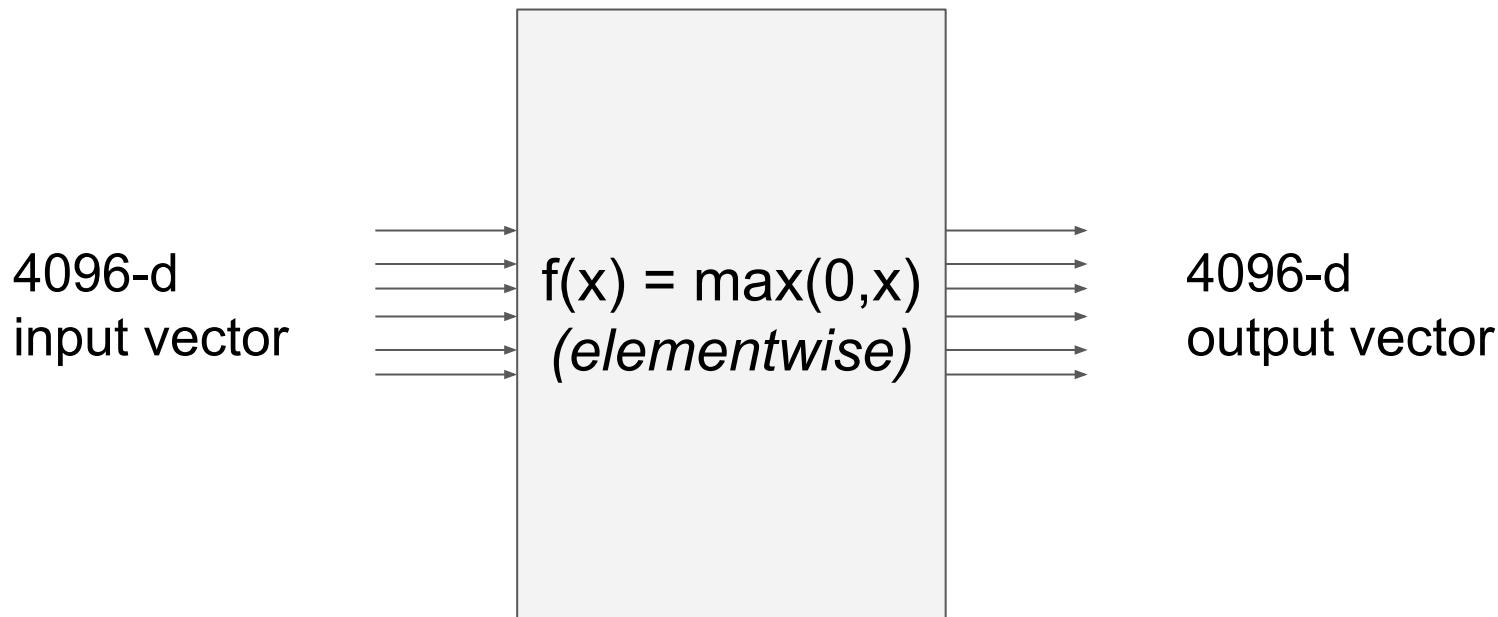
Gradients for vectorized code

(x, y, z are now vectors)

This is now the **Jacobian matrix** (derivative of each element of z w.r.t. each element of x)



Vectorized operations

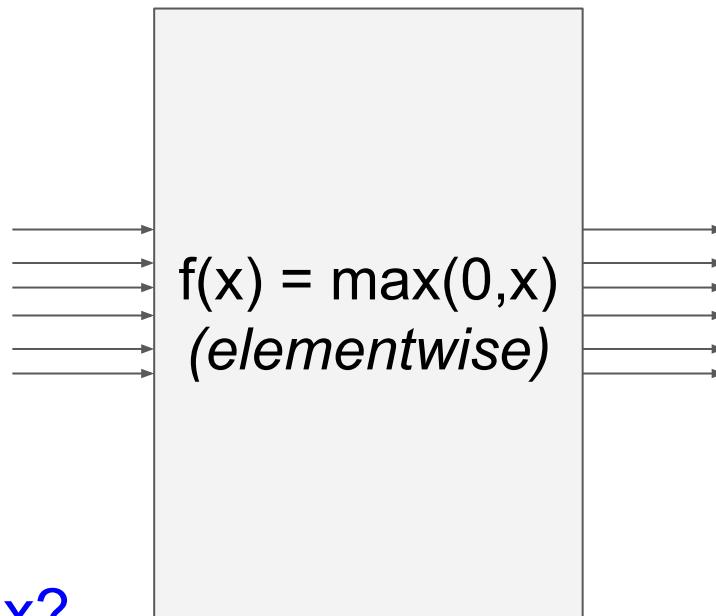


Vectorized operations

$$\frac{\partial L}{\partial x} = \boxed{\frac{\partial f}{\partial x}} \frac{\partial L}{\partial f}$$

Jacobian matrix

4096-d
input vector



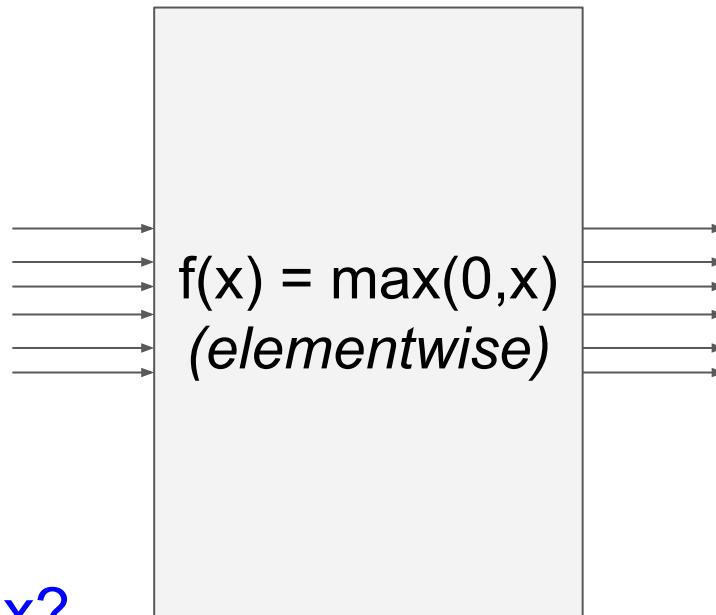
Q: what is the
size of the
Jacobian matrix?

Vectorized operations

$$\frac{\partial L}{\partial x} = \boxed{\frac{\partial f}{\partial x}} \frac{\partial L}{\partial f}$$

Jacobian matrix

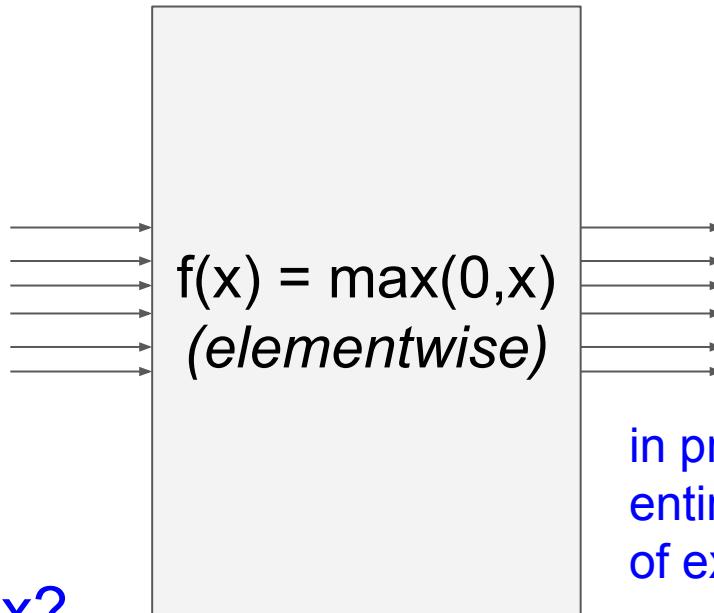
4096-d
input vector



Q: what is the
size of the
Jacobian matrix?
[4096 x 4096!]

Vectorized operations

4096-d
input vector



Q: what is the
size of the
Jacobian matrix?
[4096 x 4096!]

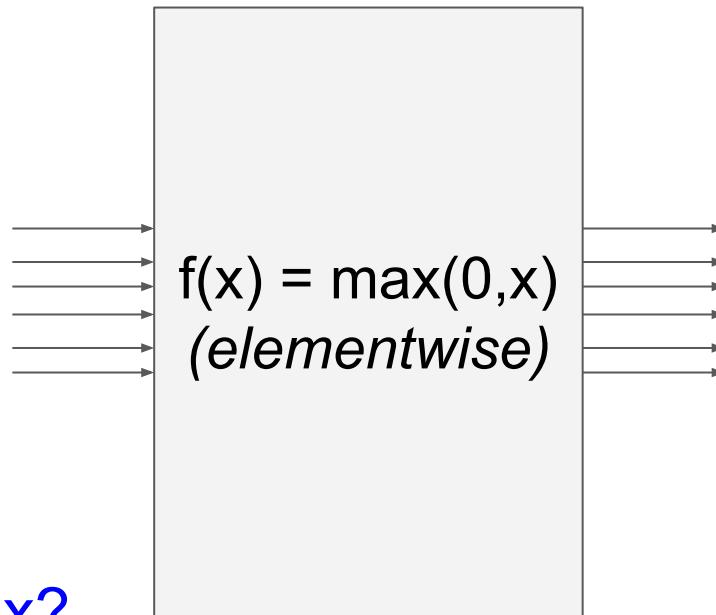
in practice we process an
entire minibatch (e.g. 100)
of examples at one time:
i.e. Jacobian would technically be a
[409,600 x 409,600] matrix :\
\\

Vectorized operations

$$\frac{\partial L}{\partial x} = \boxed{\frac{\partial f}{\partial x}} \frac{\partial L}{\partial f}$$

Jacobian matrix

4096-d
input vector



Q: what is the
size of the
Jacobian matrix?
[4096 x 4096!]

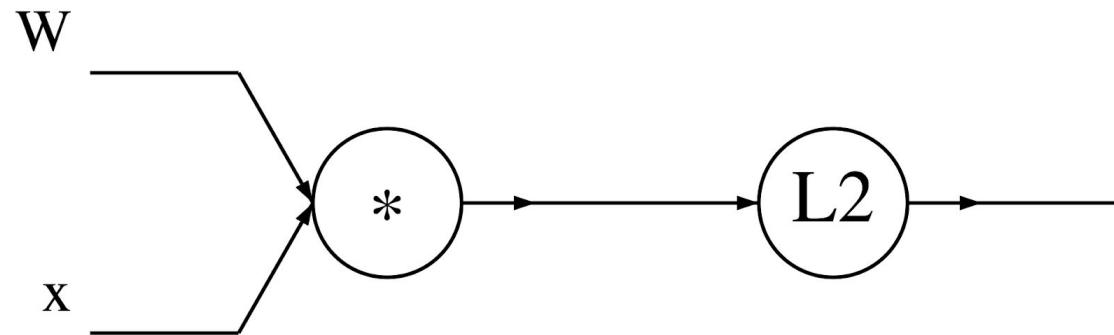
Q2: what does it
look like?

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$

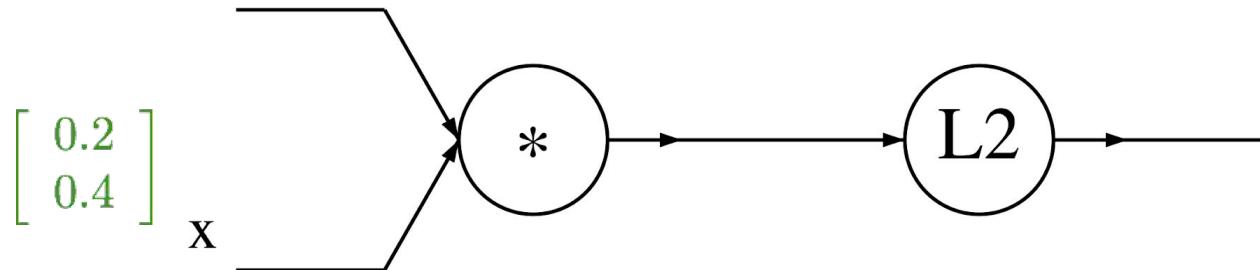
\downarrow \downarrow
 $\in \mathbb{R}^n$ $\in \mathbb{R}^{n \times n}$

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$



A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$

$$\begin{bmatrix} 0.1 & 0.5 \\ -0.3 & 0.8 \end{bmatrix} W$$

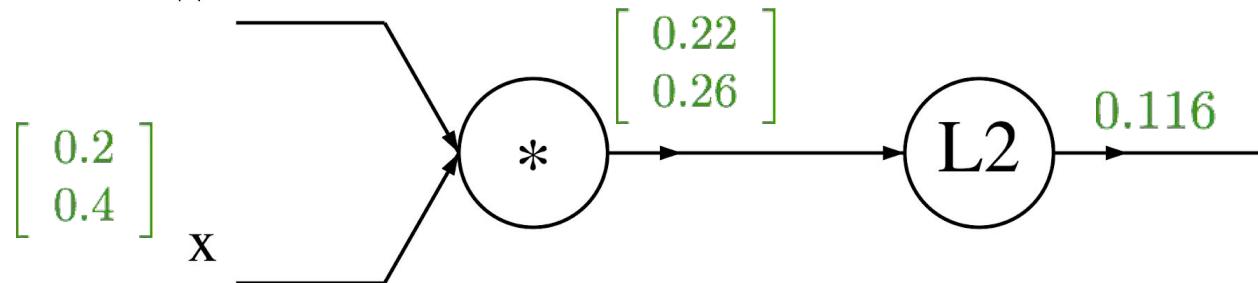


$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$

$$\begin{bmatrix} 0.1 & 0.5 \\ -0.3 & 0.8 \end{bmatrix} W$$

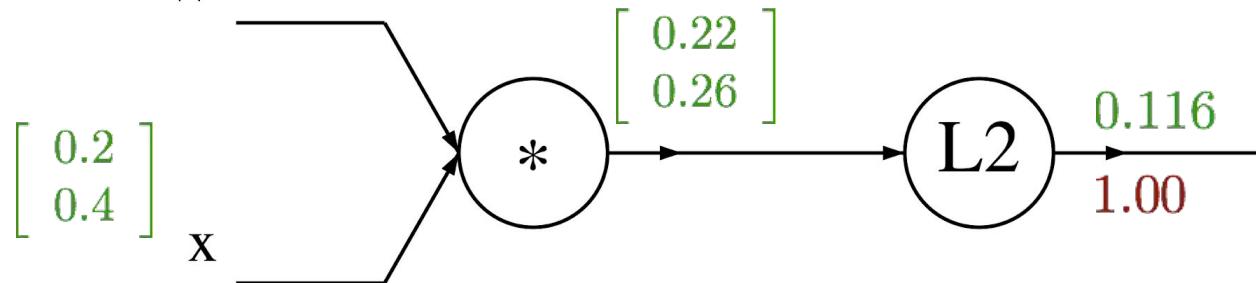


$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$

$$\begin{bmatrix} 0.1 & 0.5 \\ -0.3 & 0.8 \end{bmatrix} W$$

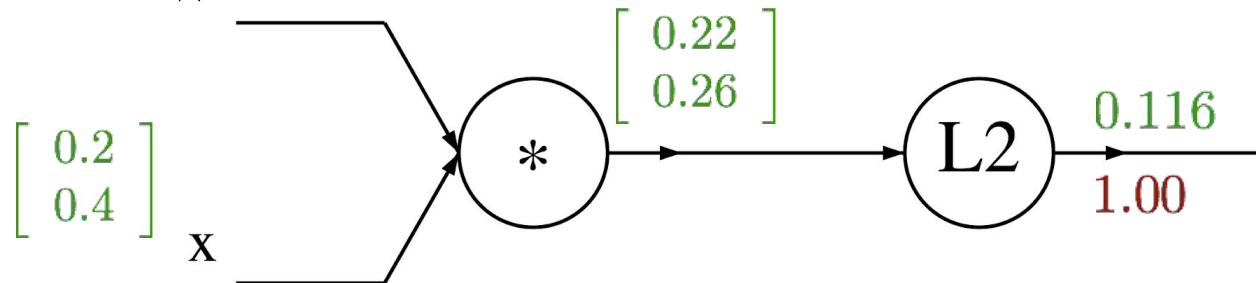


$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$

$$\begin{bmatrix} 0.1 & 0.5 \\ -0.3 & 0.8 \end{bmatrix} W$$



$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

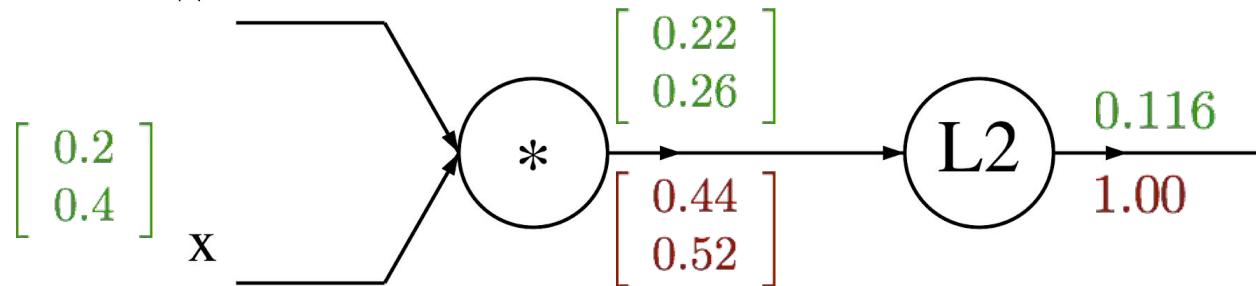
$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

$$\frac{\partial f}{\partial q_i} = 2q_i$$

$$\nabla_q f = 2q$$

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$

$$\begin{bmatrix} 0.1 & 0.5 \\ -0.3 & 0.8 \end{bmatrix} W$$



$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

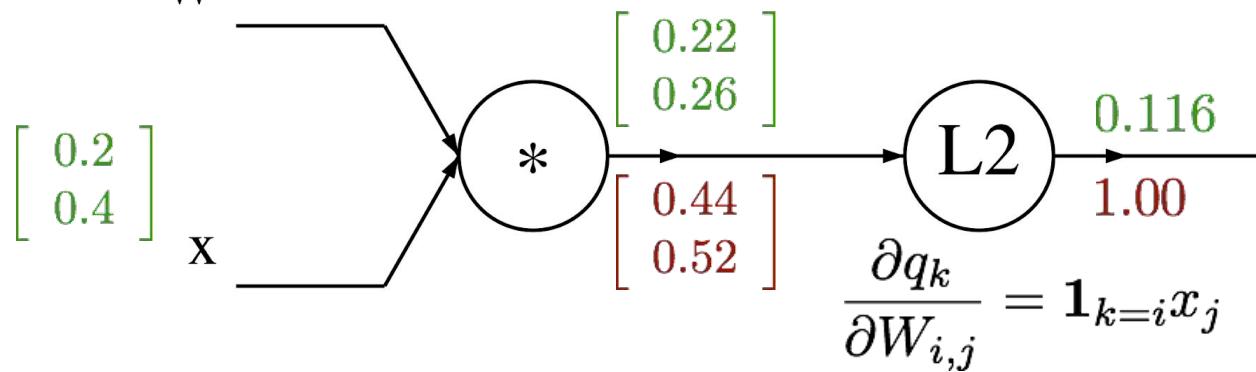
$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

$$\frac{\partial f}{\partial q_i} = 2q_i$$

$$\nabla_q f = 2q$$

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$

$$\begin{bmatrix} 0.1 & 0.5 \\ -0.3 & 0.8 \end{bmatrix} W$$

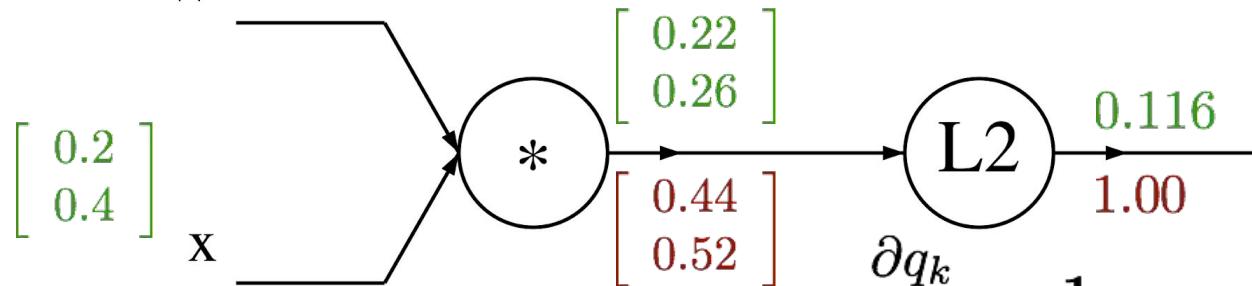


$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$

$$\begin{bmatrix} 0.1 & 0.5 \\ -0.3 & 0.8 \end{bmatrix} W$$



$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

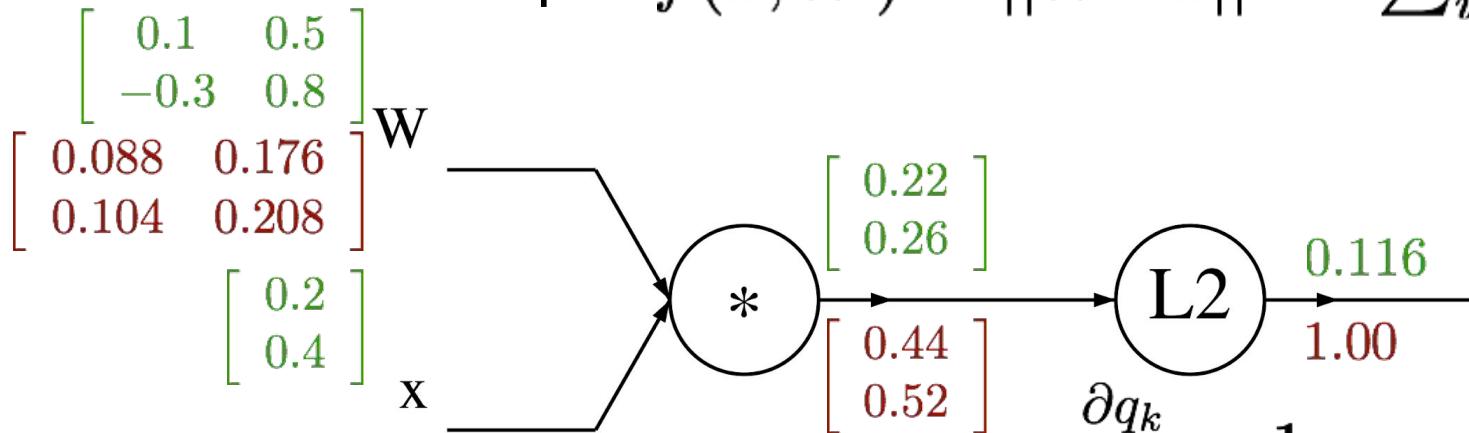
$$\frac{\partial q_k}{\partial W_{i,j}} = \mathbf{1}_{k=i} x_j$$

$$\frac{\partial f}{\partial W_{i,j}} = \sum_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial W_{i,j}}$$

$$= \sum_k (2q_k)(\mathbf{1}_{k=i} x_j)$$

$$= 2q_i x_j$$

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$



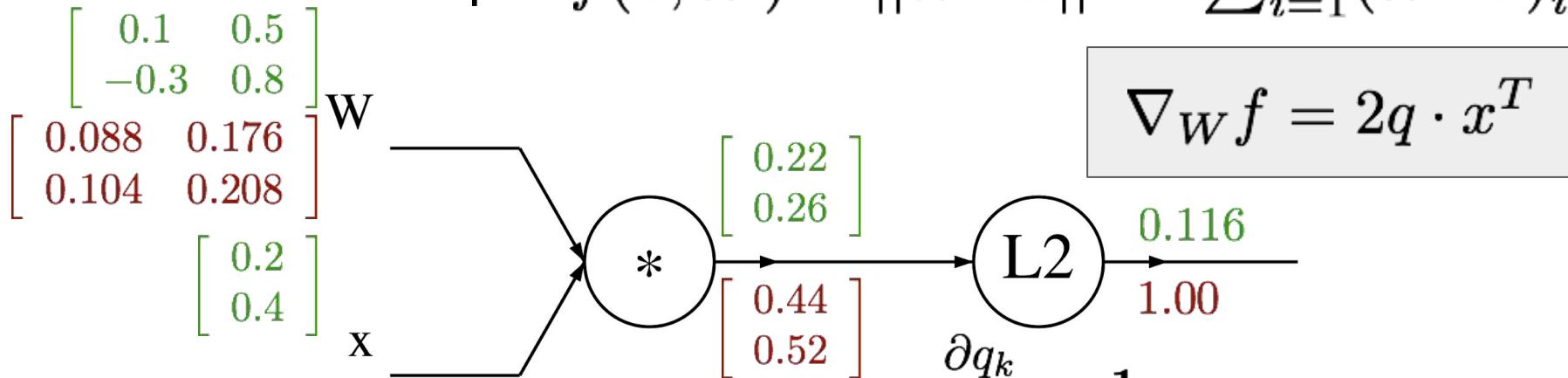
$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

$$\frac{\partial q_k}{\partial W_{i,j}} = \mathbf{1}_{k=i} x_j$$

$$\begin{aligned} \frac{\partial f}{\partial W_{i,j}} &= \sum_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial W_{i,j}} \\ &= \sum_k (2q_k)(\mathbf{1}_{k=i} x_j) \\ &= 2q_i x_j \end{aligned}$$

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$



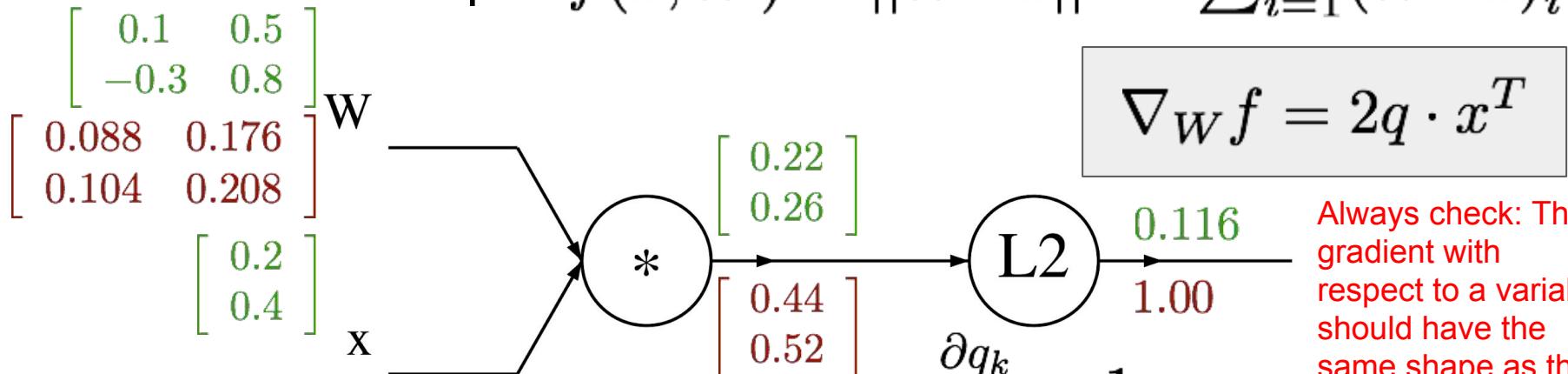
$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

$$\frac{\partial q_k}{\partial W_{i,j}} = \mathbf{1}_{k=i} x_j$$

$$\begin{aligned} \frac{\partial f}{\partial W_{i,j}} &= \sum_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial W_{i,j}} \\ &= \sum_k (2q_k)(\mathbf{1}_{k=i} x_j) \\ &= 2q_i x_j \end{aligned}$$

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$



$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

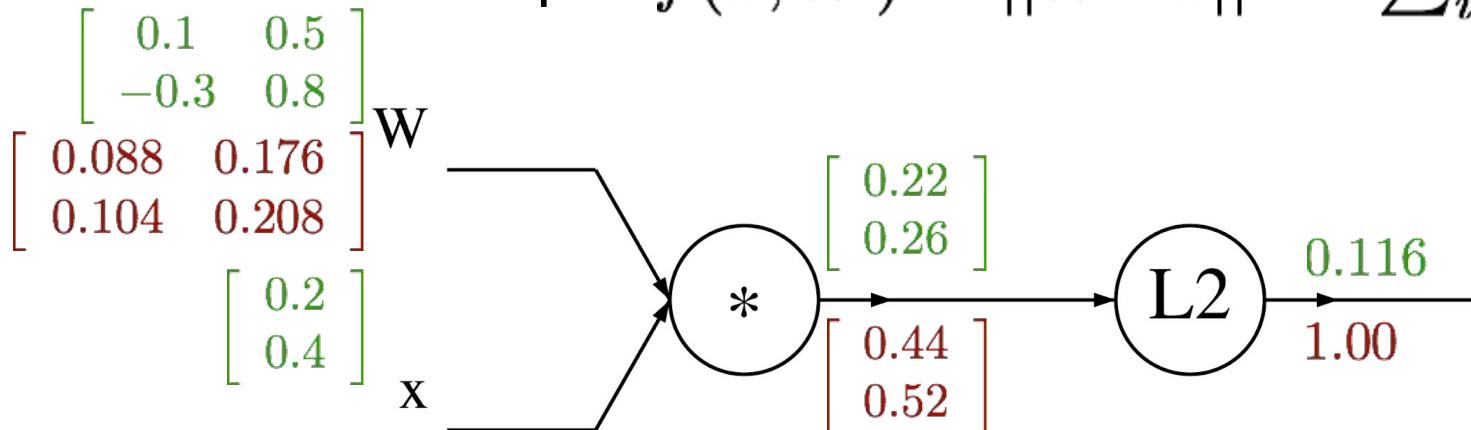
$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

$$\frac{\partial q_k}{\partial W_{i,j}} = \mathbf{1}_{k=i} x_j$$

$$\begin{aligned} \frac{\partial f}{\partial W_{i,j}} &= \sum_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial W_{i,j}} \\ &= \sum_k (2q_k)(\mathbf{1}_{k=i} x_j) \\ &= 2q_i x_j \end{aligned}$$

Always check: The gradient with respect to a variable should have the same shape as the variable

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$

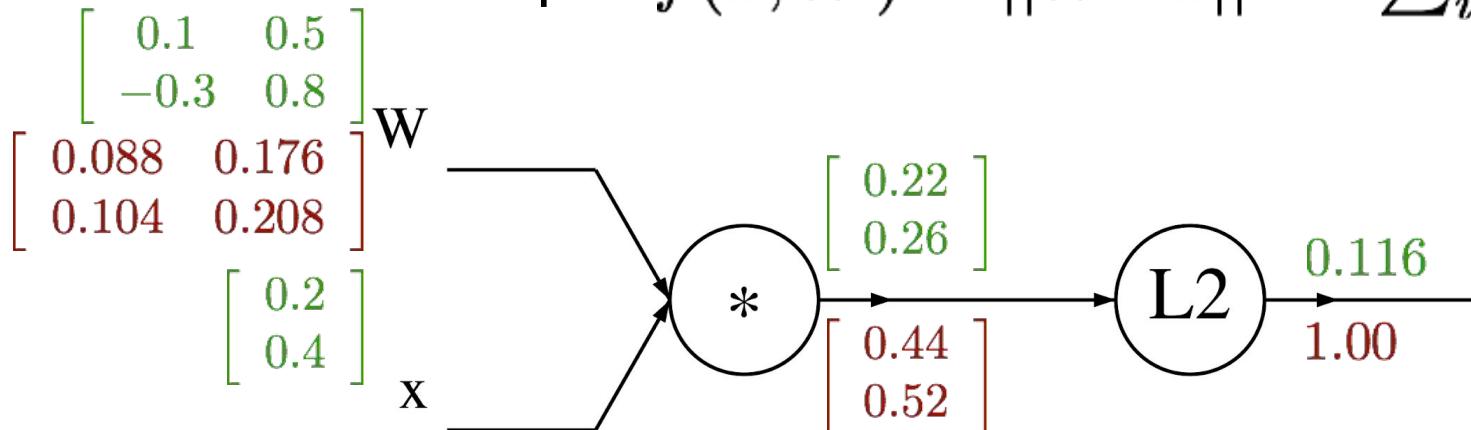


$$\frac{\partial q_k}{\partial x_i} = W_{k,i}$$

$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$

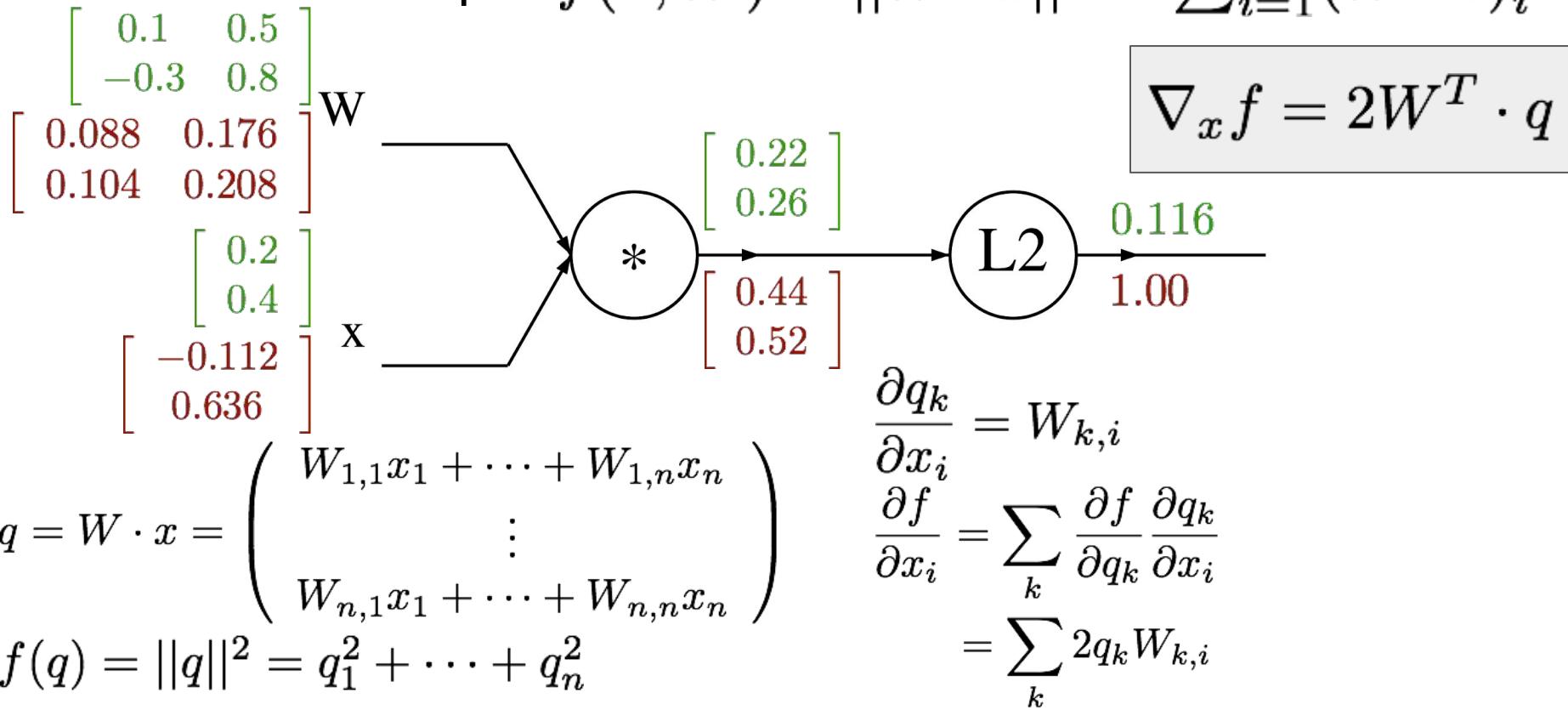


$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

$$\begin{aligned}\frac{\partial q_k}{\partial x_i} &= W_{k,i} \\ \frac{\partial f}{\partial x_i} &= \sum_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial x_i} \\ &= \sum_k 2q_k W_{k,i}\end{aligned}$$

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$



$$\begin{aligned}\frac{\partial q_k}{\partial x_i} &= W_{k,i} \\ \frac{\partial f}{\partial x_i} &= \sum_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial x_i} \\ &= \sum_k 2q_k W_{k,i}\end{aligned}$$

In discussion section: A matrix example...

$$z_1 = XW_1$$

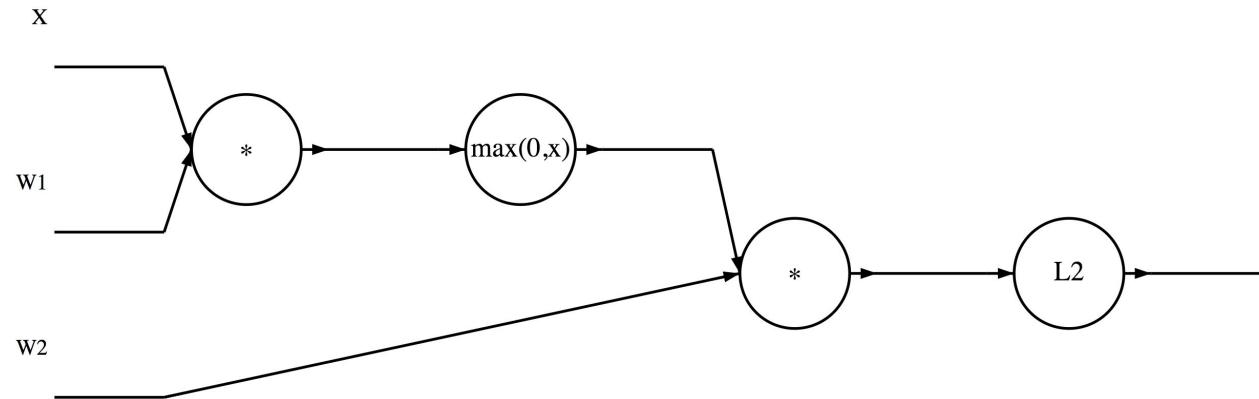
$$h_1 = \text{ReLU}(z_1)$$

$$\hat{y} = h_1 W_2$$

$$L = \|\hat{y}\|_2^2$$

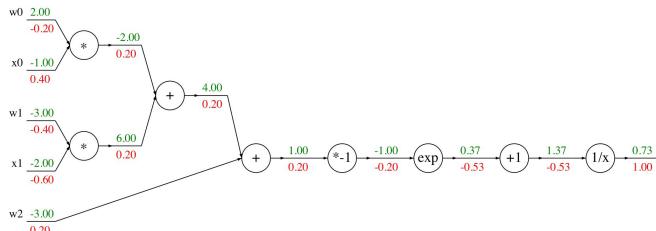
$$\frac{\partial L}{\partial W_2} = ?$$

$$\frac{\partial L}{\partial W_1} = ?$$



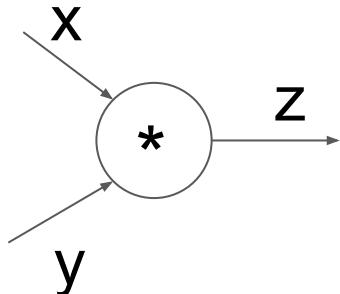
Modularized implementation: forward / backward API

Graph (or Net) object (*rough pseudo code*)



```
class ComputationalGraph(object):  
    ...  
    def forward(self, inputs):  
        # 1. [pass inputs to input gates...]  
        # 2. forward the computational graph:  
        for gate in self.graph.nodes_topologically_sorted():  
            gate.forward()  
        return loss # the final gate in the graph outputs the loss  
    def backward(self):  
        for gate in reversed(self.graph.nodes_topologically_sorted()):  
            gate.backward() # little piece of backprop (chain rule applied)  
        return inputs_gradients
```

Modularized implementation: forward / backward API



$(x,y,z$ are scalars)

```
class MultiplyGate(object):  
    def forward(x,y):  
        z = x*y  
        self.x = x # must keep these around!  
        self.y = y  
        return z  
    def backward(dz):  
        dx = self.y * dz # [dz/dx * dL/dz]  
        dy = self.x * dz # [dz/dy * dL/dz]  
        return [dx, dy]
```

Local gradient Upstream gradient variable

Example: Caffe layers

Branch: master	cafe / src / caffe / layers /	Create new file	Upload files	Find file	History
	shelhamer committed on GitHub Merge pull request #4630 from BiGene/load_hdf5_fix ...				Latest commit e687a71 21 days ago
..					
absval_layer.cpp	dismantle layer headers				a year ago
absval_layer.cu	dismantle layer headers				a year ago
accuracy_layer.cpp	dismantle layer headers				a year ago
argmax_layer.cpp	dismantle layer headers				a year ago
base_conv_layer.cpp	enable dilated deconvolution				a year ago
base_data_layer.cpp	Using default from proto for prefetch				3 months ago
base_data_layer.cu	Switched multi-GPU to NCCL				3 months ago
batch_norm_layer.cpp	Add missing spaces besides equal signs in batch_norm_layer.cpp				4 months ago
batch_norm_layer.cu	dismantle layer headers				a year ago
batch_reindex_layer.cpp	dismantle layer headers				a year ago
batch_reindex_layer.cu	dismantle layer headers				a year ago
bias_layer.cpp	Remove incorrect cast of gemm int arg to Dtype in BiasLayer				a year ago
bias_layer.cu	Separation and generalization of ChannelwiseAffineLayer into BiasLayer				a year ago
bnll_layer.cpp	dismantle layer headers				a year ago
bnll_layer.cu	dismantle layer headers				a year ago
concat_layer.cpp	dismantle layer headers				a year ago
concat_layer.cu	dismantle layer headers				a year ago
contrastive_loss_layer.cpp	dismantle layer headers				a year ago
contrastive_loss_layer.cu	dismantle layer headers				a year ago
conv_layer.cpp	add support for 2D dilated convolution				a year ago
conv_layer.cu	dismantle layer headers				a year ago
crop_layer.cpp	remove redundant operations in Crop layer (#5138)				2 months ago
crop_layer.cu	remove redundant operations in Crop layer (#5138)				2 months ago
cudnn_conv_layer.cpp	dismantle layer headers				a year ago
cudnn_conv_layer.cu	Add cuDNN v5 support, drop cuDNN v3 support				11 months ago
cudnn_lcn_layer.cpp	dismantle layer headers				a year ago
cudnn_lcn_layer.cu	dismantle layer headers				a year ago
cudnn_lrn_layer.cpp	dismantle layer headers				a year ago
cudnn_lrn_layer.cu	dismantle layer headers				a year ago
cudnn_pooling_layer.cpp	dismantle layer headers				a year ago
cudnn_pooling_layer.cu	dismantle layer headers				a year ago
cudnn_relu_layer.cpp	Add cuDNN v5 support, drop cuDNN v3 support				11 months ago
cudnn_relu_layer.cu	Add cuDNN v5 support, drop cuDNN v3 support				11 months ago
cudnn_sigmoid_layer.cpp	Add cuDNN v5 support, drop cuDNN v3 support				11 months ago
cudnn_sigmoid_layer.cu	Add cuDNN v5 support, drop cuDNN v3 support				11 months ago
cudnn_softmax_layer.cpp	dismantle layer headers				a year ago
cudnn_softmax_layer.cu	dismantle layer headers				a year ago
cudnn_tanh_layer.cpp	Add cuDNN v5 support, drop cuDNN v3 support				11 months ago
cudnn_tanh_layer.cu	Add cuDNN v5 support, drop cuDNN v3 support				11 months ago
data_layer.cpp	Switched multi-GPU to NCCL				3 months ago
deconv_layer.cpp	enable dilated deconvolution				a year ago
deconv_layer.cu	dismantle layer headers				a year ago
dropout_layer.cpp	supporting N-D Blobs in Dropout layer Reshape				a year ago
dropout_layer.cu	dismantle layer headers				a year ago
dummy_data_layer.cpp	dismantle layer headers				a year ago
eltwise_layer.cpp	dismantle layer headers				a year ago
eltwise_layer.cu	dismantle layer headers				a year ago
elu_layer.cpp	ELU layer with basic tests				a year ago
elu_layer.cu	ELU layer with basic tests				a year ago
embed_layer.cpp	dismantle layer headers				a year ago
embed_layer.cu	dismantle layer headers				a year ago
euclidean_loss_layer.cpp	dismantle layer headers				a year ago
euclidean_loss_layer.cu	dismantle layer headers				a year ago
exp_layer.cpp	Solving issue with exp layer with base e				a year ago
exp_layer.cu	dismantle layer headers				a year ago

Caffe is licensed under [BSD 2-Clause](#)

Caffe Sigmoid Layer

```
1 #include <cmath>
2 #include <vector>
3
4 #include "caffe/layers/sigmoid_layer.hpp"
5
6 namespace caffe {
7
8     template <typename Dtype>
9     inline Dtype sigmoid(Dtype x) {
10         return 1. / (1. + exp(-x));
11     }
12
13     template <typename Dtype>
14     void SigmoidLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>>& bottom,
15                                             const vector<Blob<Dtype>>& top) {
16         const Dtype* bottom_data = bottom[0]->cpu_data();
17         Dtype* top_data = top[0]->mutable_cpu_data();
18         const int count = bottom[0]->count();
19         for (int i = 0; i < count; ++i) {
20             top_data[i] = sigmoid(bottom_data[i]);
21         }
22     }
23
24     template <typename Dtype>
25     void SigmoidLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>>& top,
26                                              const vector<bool>& propagate_down,
27                                              const vector<Blob<Dtype>>& bottom) {
28         if (propagate_down[0]) {
29             const Dtype* top_data = top[0]->cpu_data();
30             const Dtype* top_diff = top[0]->cpu_diff();
31             Dtype* bottom_diff = bottom[0]->mutable_cpu_diff();
32             const int count = bottom[0]->count();
33             for (int i = 0; i < count; ++i) {
34                 const Dtype sigmoid_x = top_data[i];
35                 bottom_diff[i] = top_diff[i] * sigmoid_x * (1. - sigmoid_x);
36             }
37         }
38     }
39
40 #ifdef CPU_ONLY
41 STUB_GPU(SigmoidLayer);
42#endif
43
44 INSTANTIATE_CLASS(SigmoidLayer);
45
46
47 } // namespace caffe
```

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

$$(1 - \sigma(x)) \sigma'(x) * \text{top_diff}$$
 (chain rule)

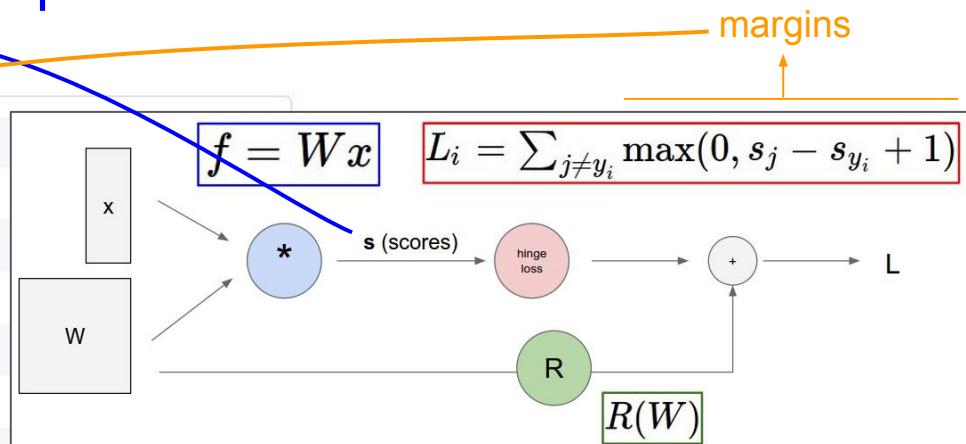
[Caffe](#) is licensed under [BSD 2-Clause](#)

In Assignment 1: Writing SVM / Softmax

Stage your forward/backward computation!

E.g. for the SVM:

```
# receive W (weights), X (data)
# forward pass (we have 6 lines)
scores = #...
margins = #... ←
data_loss = #...
reg_loss = #...
loss = data_loss + reg_loss
# backward pass (we have 5 lines)
dmargins = # ... (optionally, we go direct to dscores)
dscores = #...
dW = #...
```



Summary so far...

- neural nets will be very large: impractical to write down gradient formula by hand for all parameters
- **backpropagation** = recursive application of the chain rule along a computational graph to compute the gradients of all inputs/parameters/intermediates
- implementations maintain a graph structure, where the nodes implement the **forward()** / **backward()** API
- **forward**: compute result of an operation and save any intermediates needed for gradient computation in memory
- **backward**: apply the chain rule to compute the gradient of the loss function with respect to the inputs

Next: Neural Networks

Neural networks: without the brain stuff

(Before) Linear score function: $f = Wx$

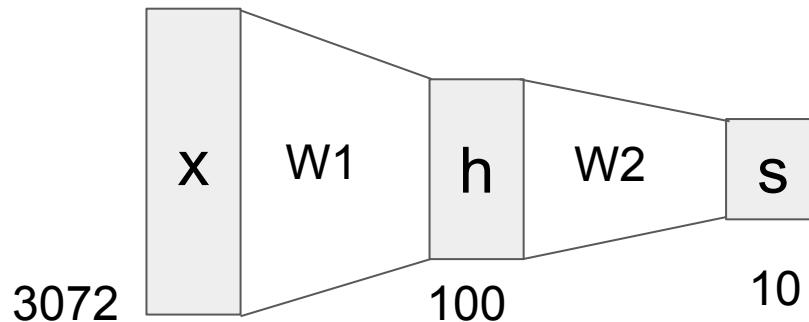
Neural networks: without the brain stuff

- (Before) Linear score function: $f = Wx$
- (Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

Neural networks: without the brain stuff

(Before) Linear score function: $f = Wx$

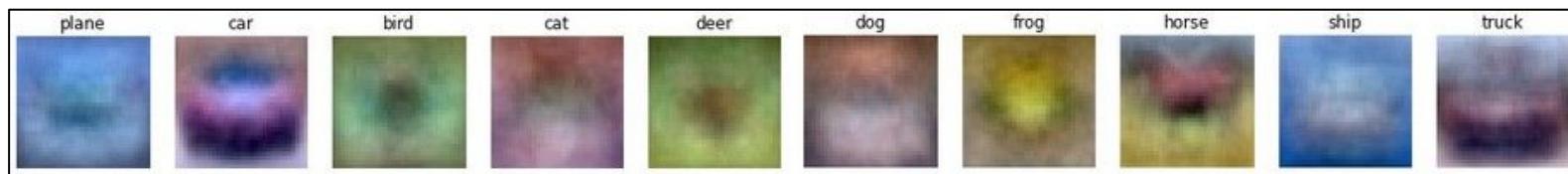
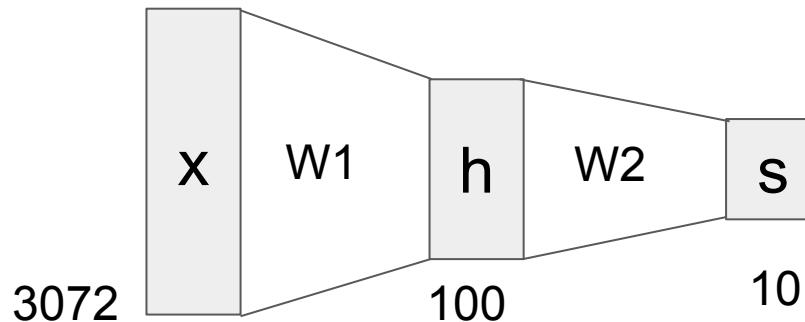
(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$



Neural networks: without the brain stuff

(Before) Linear score function: $f = Wx$

(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$



Neural networks: without the brain stuff

- (Before) Linear score function: $f = Wx$
- (Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$
or 3-layer Neural Network
 $f = W_3 \max(0, W_2 \max(0, W_1 x))$

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14 grad_y_pred = 2.0 * (y_pred - y)
15 grad_w2 = h.T.dot(grad_y_pred)
16 grad_h = grad_y_pred.dot(w2.T)
17 grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19 w1 -= 1e-4 * grad_w1
20 w2 -= 1e-4 * grad_w2
```

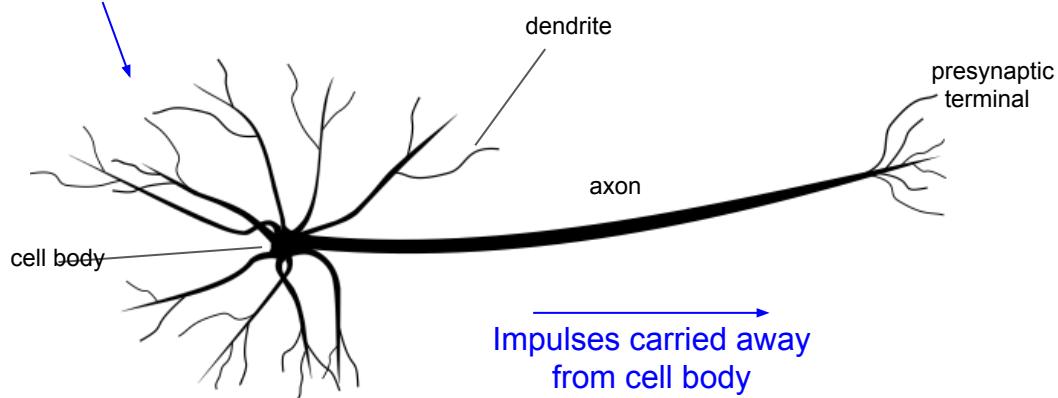
In HW: Writing a 2-layer net

```
# receive W1,W2,b1,b2 (weights/biases), X (data)
# forward pass:
h1 = #... function of X,W1,b1
scores = #... function of h1,W2,b2
loss = #... (several lines of code to evaluate Softmax loss)
# backward pass:
dscores = #...
dh1,dW2,db2 = #...
dW1,db1 = #...
```



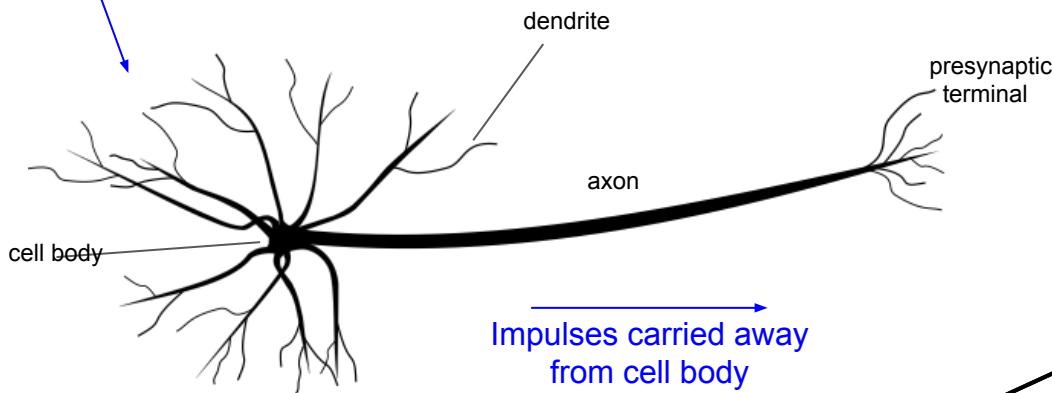
This image by [Fotis Bobolas](#) is
licensed under [CC-BY 2.0](#)

Impulses carried toward cell body



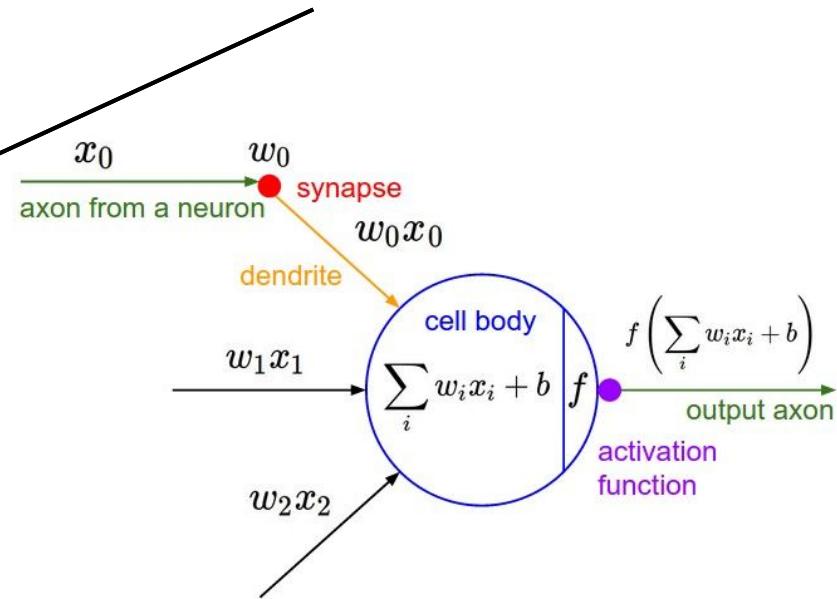
[This image](#) by Felipe Perucho
is licensed under [CC-BY 3.0](#)

Impulses carried toward cell body

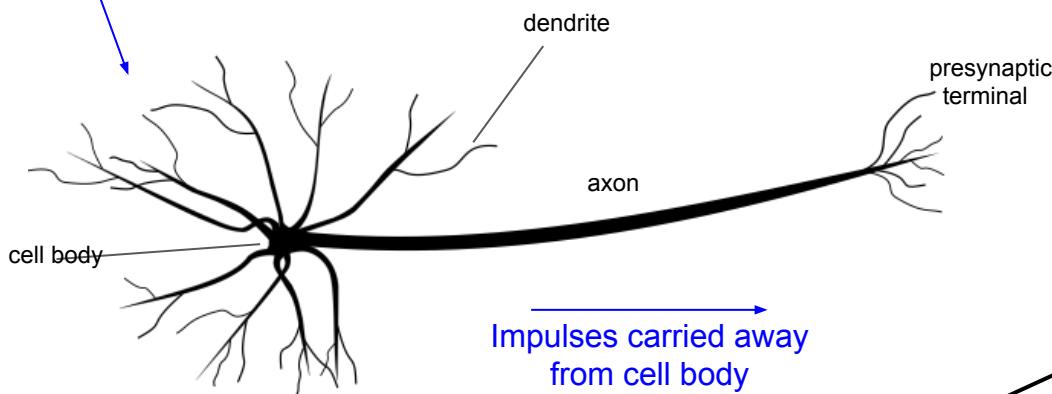


This image by Felipe Perucho
is licensed under CC-BY 3.0

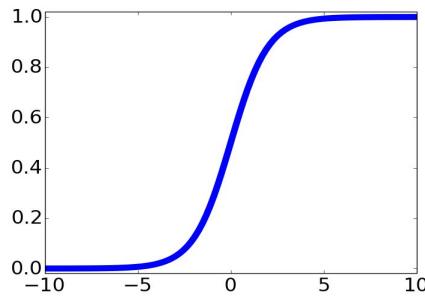
Impulses carried away
from cell body



Impulses carried toward cell body

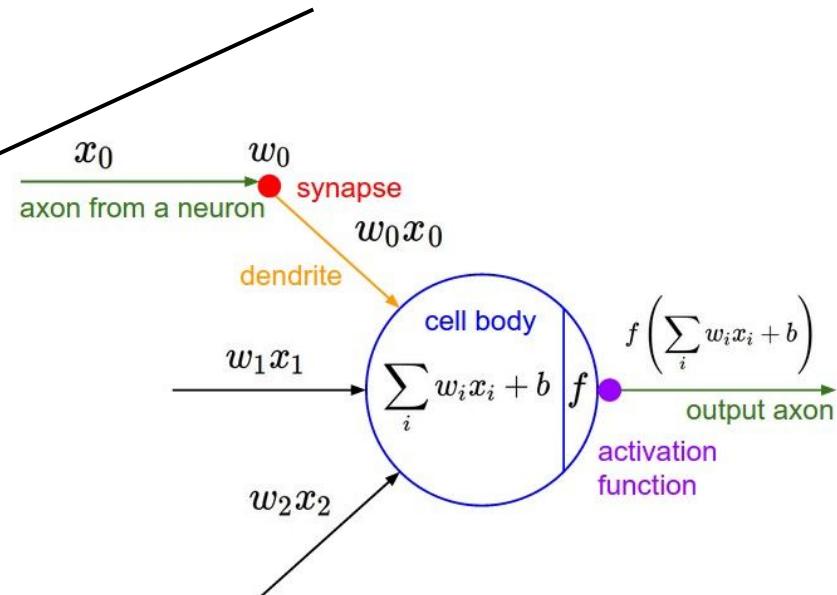


This image by Felipe Perucho
is licensed under CC-BY 3.0

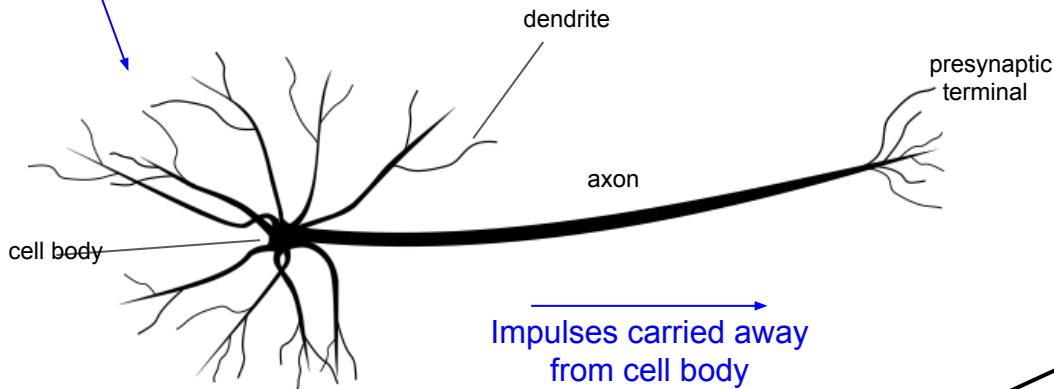


sigmoid activation function

$$\frac{1}{1 + e^{-x}}$$

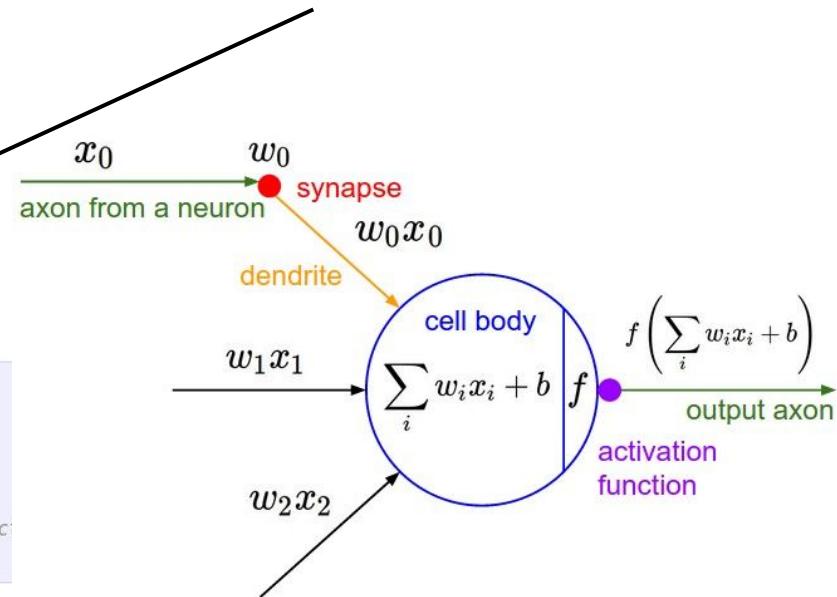


Impulses carried toward cell body



This image by Felipe Perucco
is licensed under CC-BY 3.0

```
class Neuron:  
    ...  
    def neuron_tick(inputs):  
        """ assume inputs and weights are 1-D numpy arrays and bias is a number """  
        cell_body_sum = np.sum(inputs * self.weights) + self.bias  
        firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation function  
        return firing_rate
```



Be very careful with your brain analogies!

Biological Neurons:

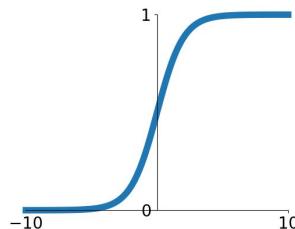
- Many different types
- Dendrites can perform complex non-linear computations
- Synapses are not a single weight but a complex non-linear dynamical system
- Rate code may not be adequate

[Dendritic Computation. London and Häusser]

Activation functions

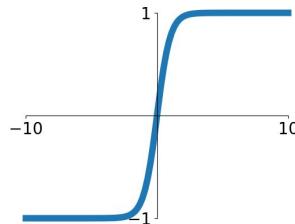
Sigmoid

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



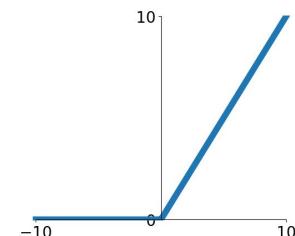
tanh

$$\tanh(x)$$



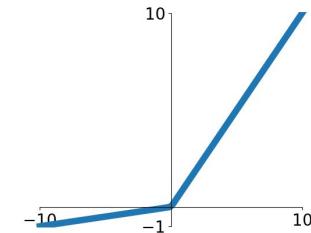
ReLU

$$\max(0, x)$$



Leaky ReLU

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

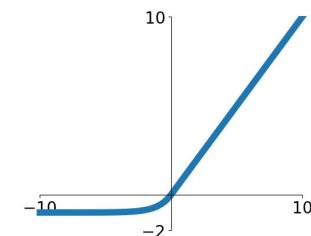


Maxout

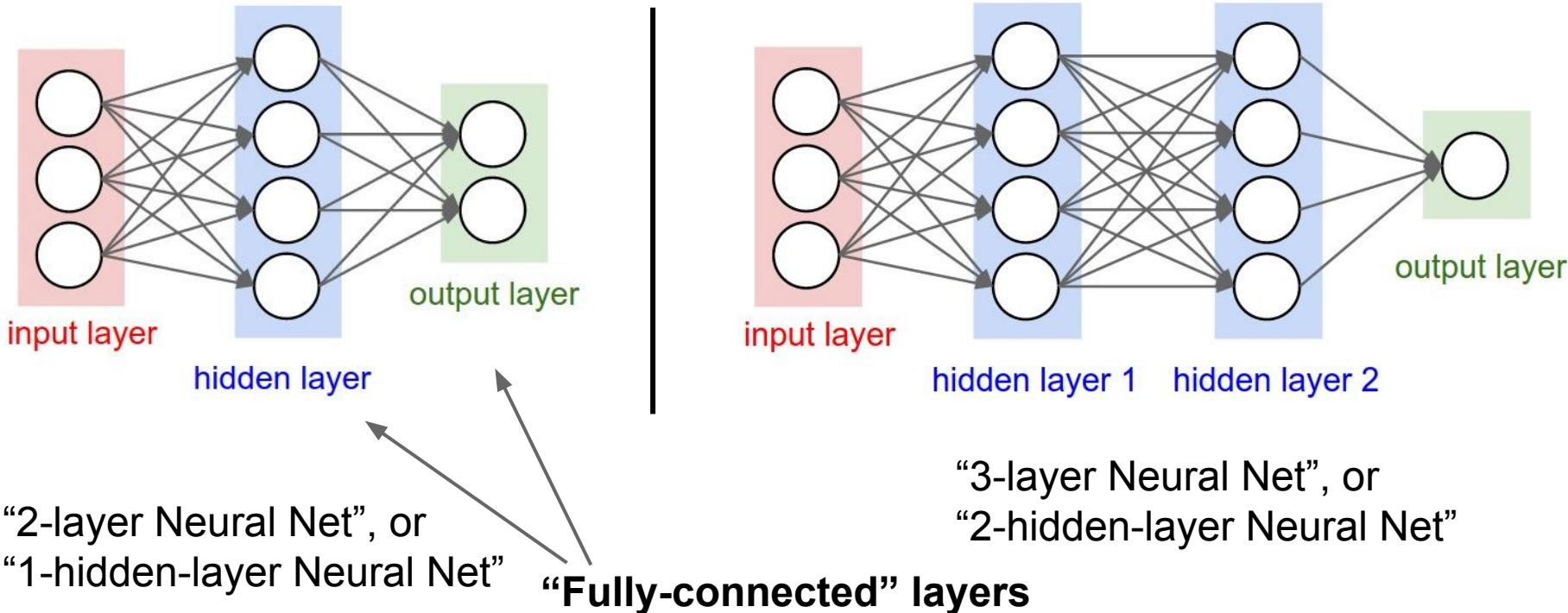
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

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



Neural networks: Architectures

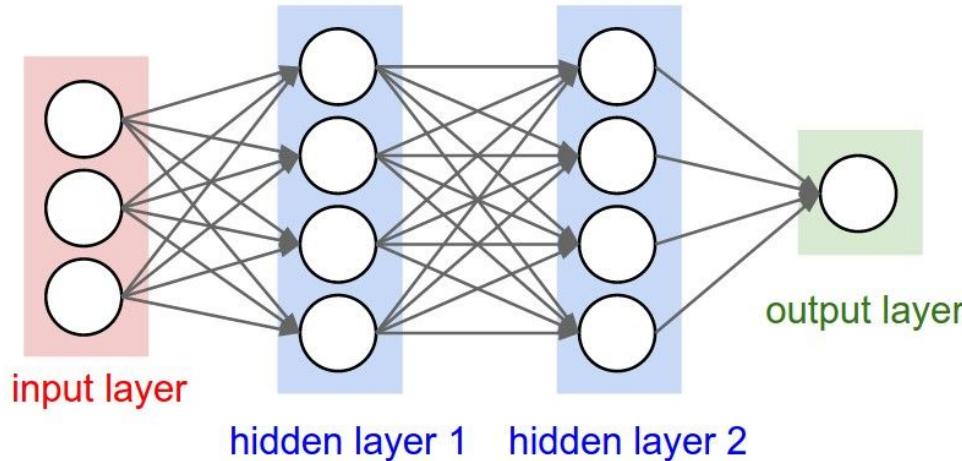


Example feed-forward computation of a neural network

```
class Neuron:  
    # ...  
    def neuron_tick(inputs):  
        """ assume inputs and weights are 1-D numpy arrays and bias is a number """  
        cell_body_sum = np.sum(inputs * self.weights) + self.bias  
        firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation function  
        return firing_rate
```

We can efficiently evaluate an entire layer of neurons.

Example feed-forward computation of a neural network



```
# forward-pass of a 3-layer neural network:  
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)  
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)  
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)  
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)  
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

Summary

- We arrange neurons into fully-connected layers
- The abstraction of a **layer** has the nice property that it allows us to use efficient vectorized code (e.g. matrix multiplies)
- Neural networks are not really *neural*
- Next time: Convolutional Neural Networks

Lecture 5: Convolutional Neural Networks

Administrative

Assignment 1 due **Wednesday April 18, 11:59pm**
Assignment 2 will also be released Wednesday

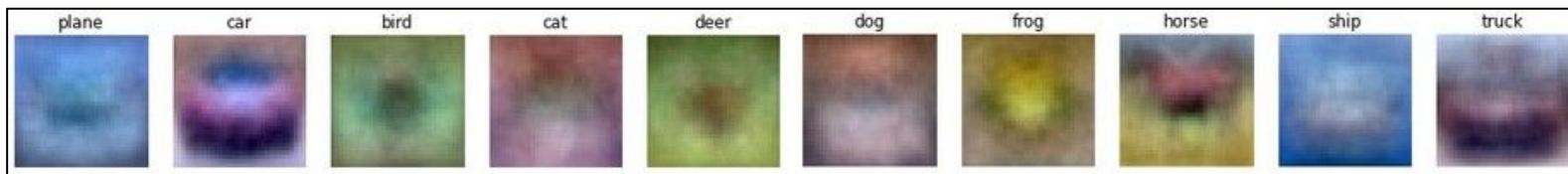
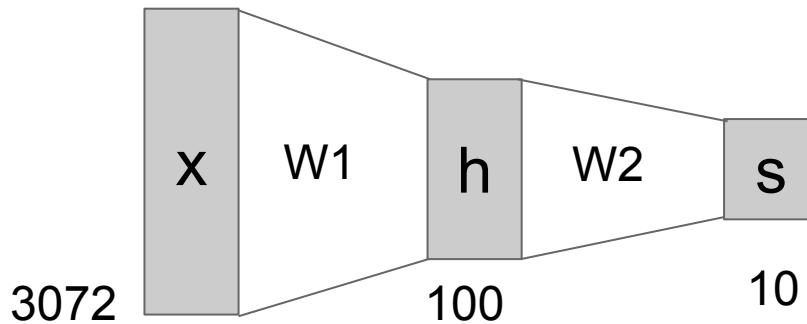
Last time: Neural Networks

Linear score function:

$$f = Wx$$

2-layer Neural Network

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



Next: Convolutional Neural Networks

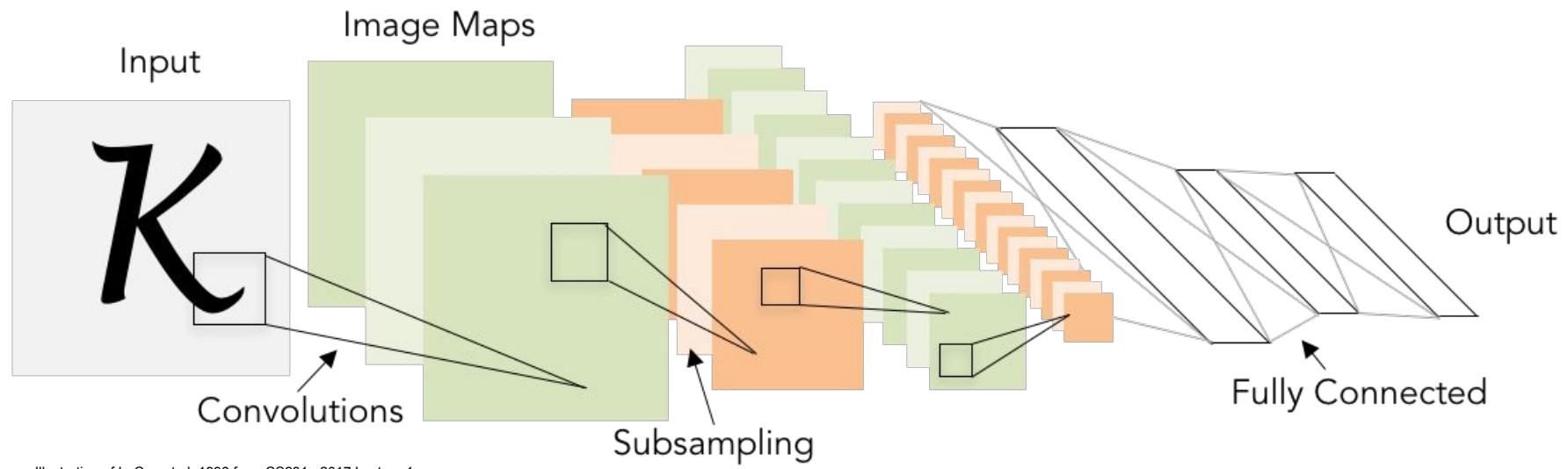


Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

A bit of history...

The **Mark I Perceptron** machine was the first implementation of the perceptron algorithm.

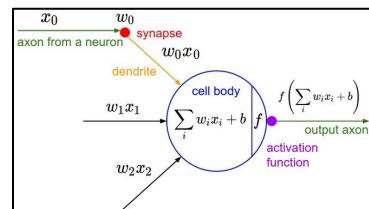
The machine was connected to a camera that used 20×20 cadmium sulfide photocells to produce a 400-pixel image.

recognized letters of the alphabet

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

update rule:

$$w_i(t+1) = w_i(t) + \alpha(d_j - y_j(t))x_{j,i}$$

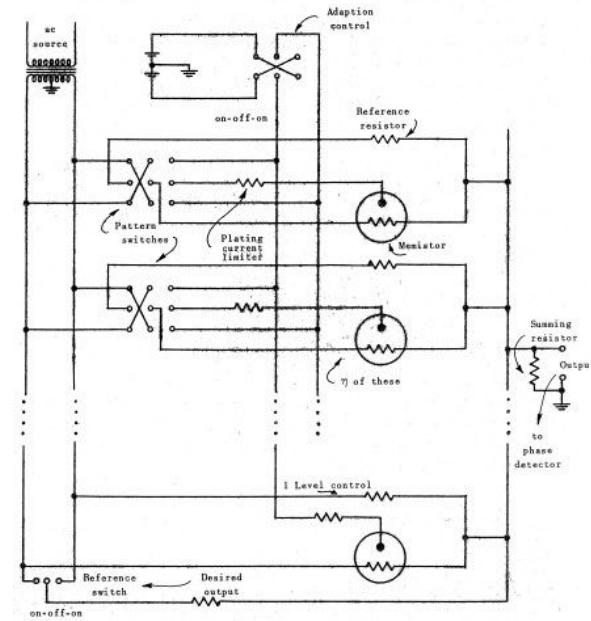
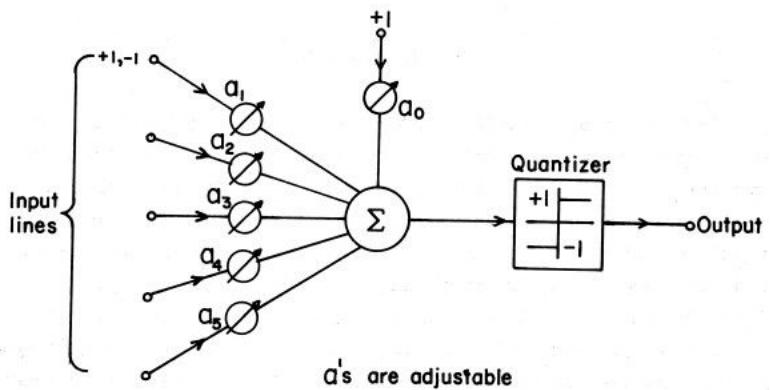


Frank Rosenblatt, ~1957: Perceptron



[This image](#) by Rocky Acosta is licensed under [CC-BY 3.0](#)

A bit of history...

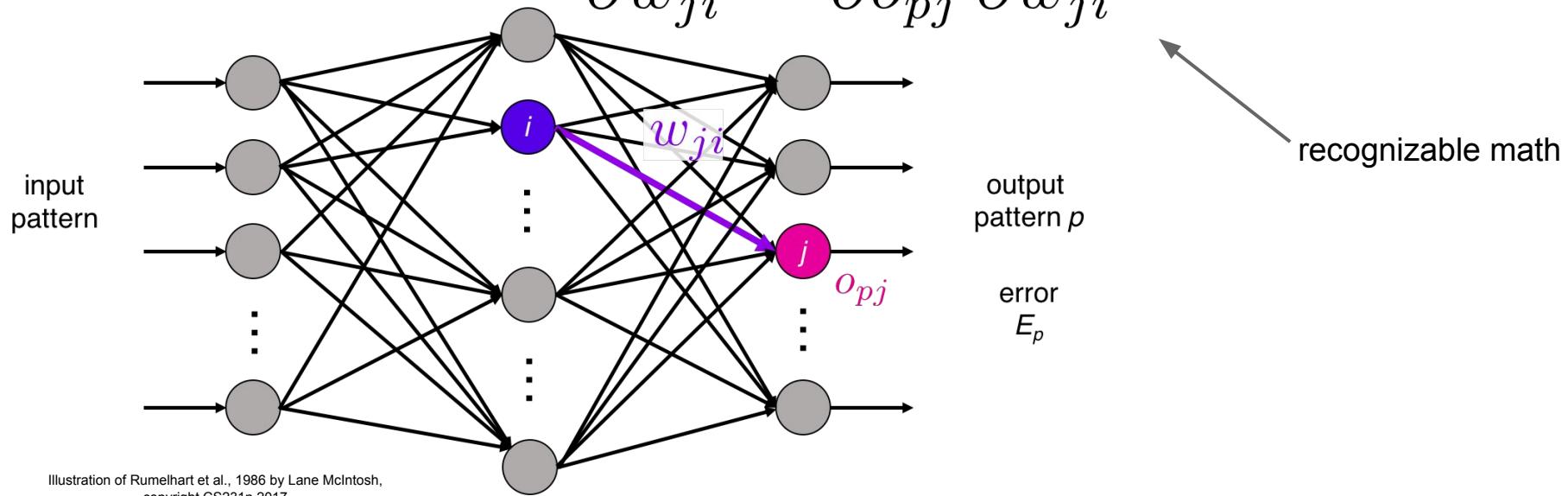


Widrow and Hoff, ~1960: Adaline/Madaline

These figures are reproduced from [Widrow 1960, Stanford Electronics Laboratories Technical Report](#) with permission from [Stanford University Special Collections](#).

A bit of history...

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



Rumelhart et al., 1986: First time back-propagation became popular

A bit of history...

[Hinton and Salakhutdinov 2006]

Reinvigorated research in
Deep Learning

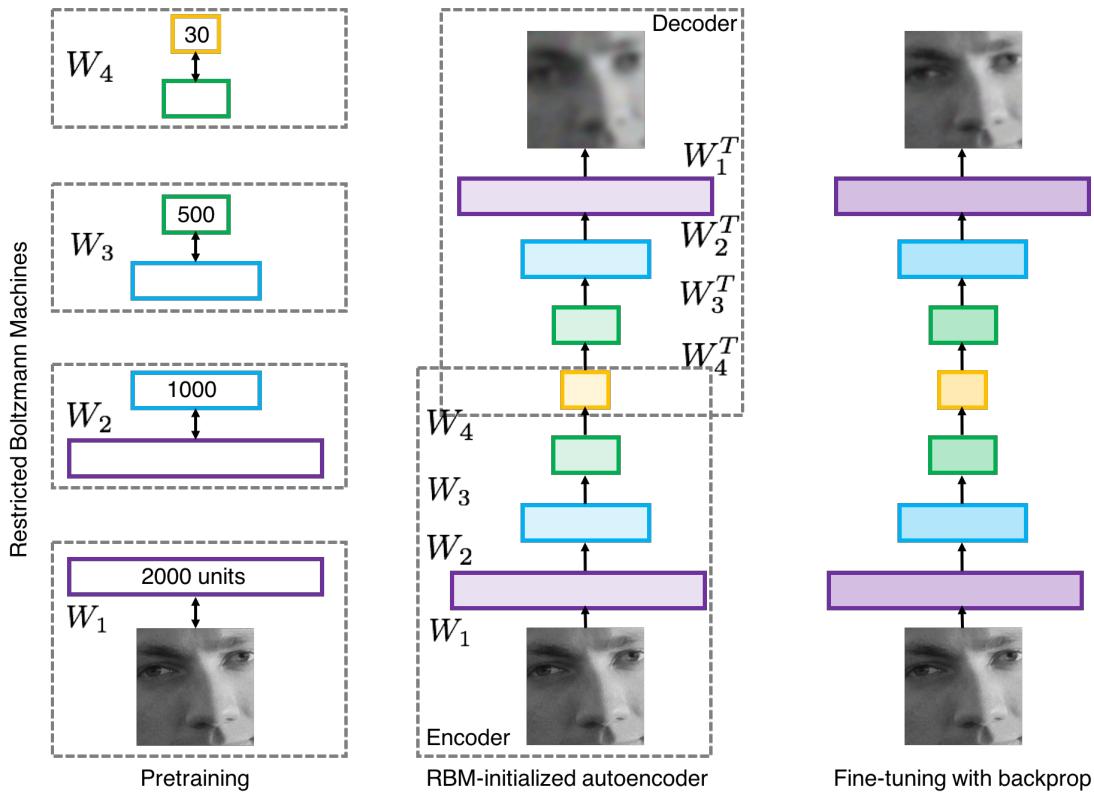


Illustration of Hinton and Salakhutdinov 2006 by Lane McIntosh, copyright CS231n 2017

First strong results

Acoustic Modeling using Deep Belief Networks

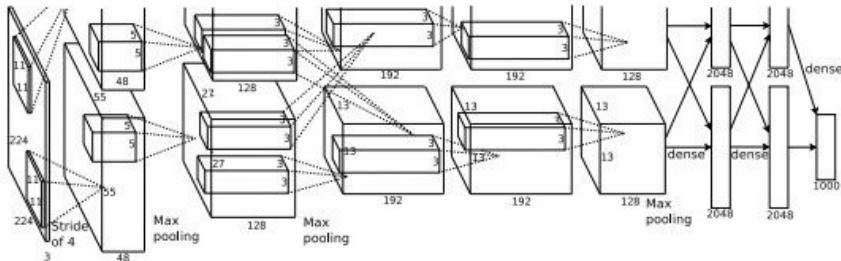
Abdel-rahman Mohamed, George Dahl, Geoffrey Hinton, 2010

Context-Dependent Pre-trained Deep Neural Networks for Large Vocabulary Speech Recognition

George Dahl, Dong Yu, Li Deng, Alex Acero, 2012

Imagenet classification with deep convolutional neural networks

Alex Krizhevsky, Ilya Sutskever, Geoffrey E Hinton, 2012



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

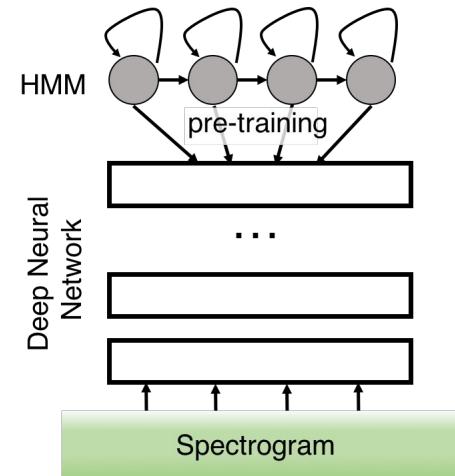


Illustration of Dahl et al. 2012 by Lane McIntosh, copyright CS231n 2017

A bit of history:

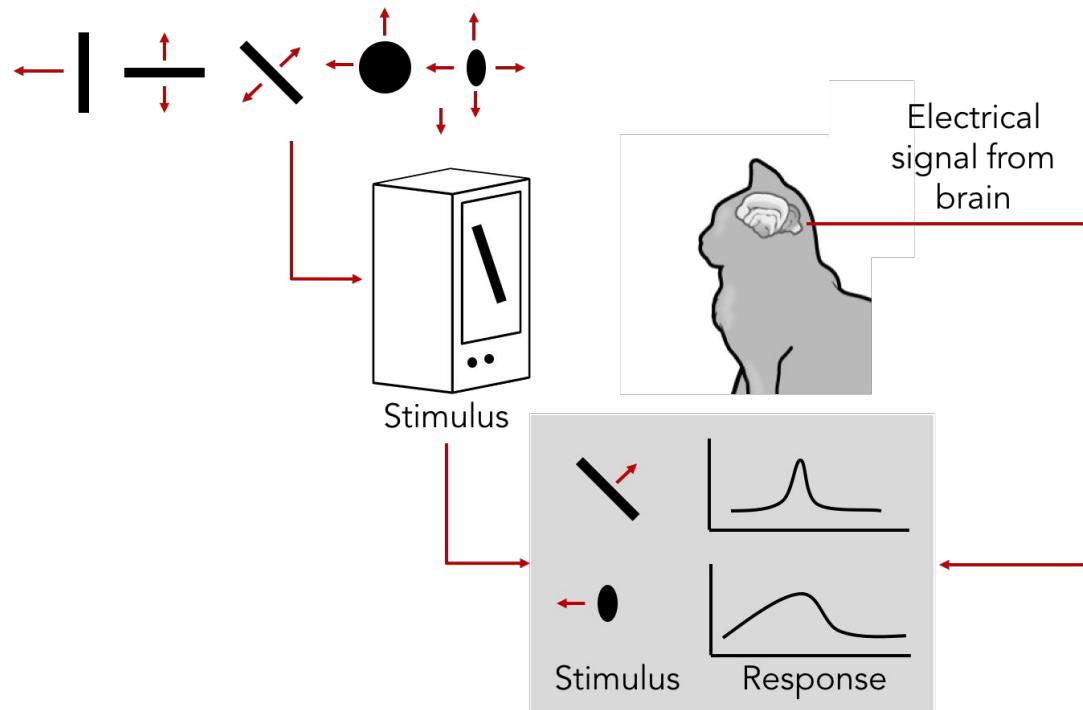
**Hubel & Wiesel,
1959**

RECEPTIVE FIELDS OF SINGLE
NEURONES IN
THE CAT'S STRIATE CORTEX

1962

RECEPTIVE FIELDS, BINOCULAR
INTERACTION
AND FUNCTIONAL ARCHITECTURE IN
THE CAT'S VISUAL CORTEX

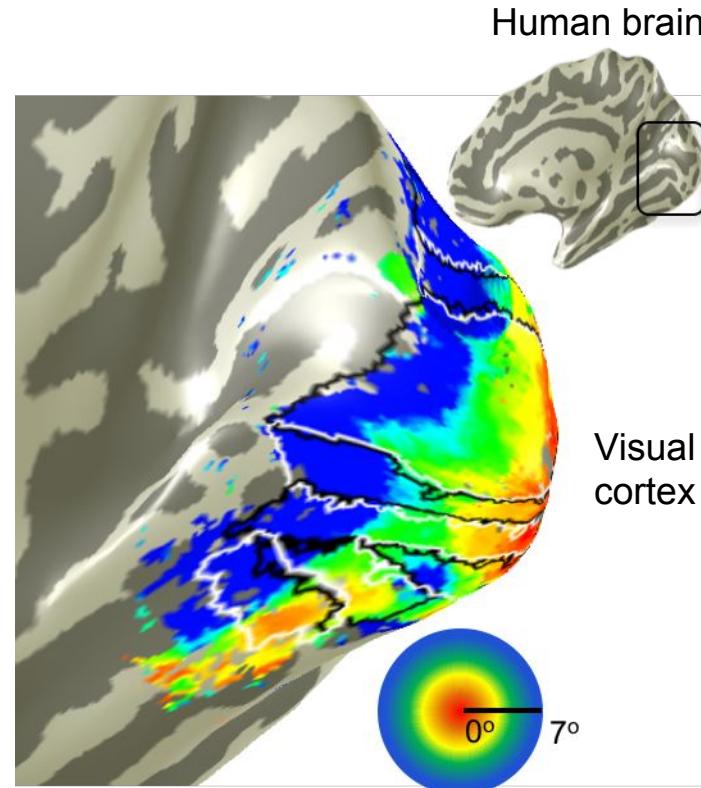
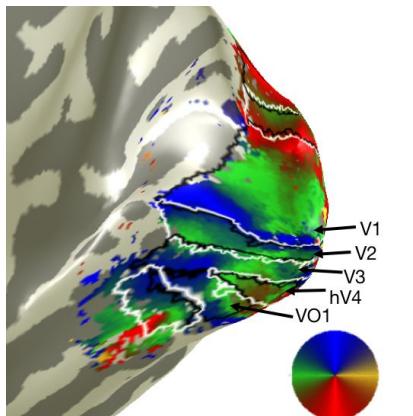
1968...



[Cat image](#) by CNX OpenStax is licensed
under CC BY 4.0; changes made

A bit of history

Topographical mapping in the cortex:
nearby cells in cortex represent
nearby regions in the visual field



Retinotopy images courtesy of Jesse Gomez in the Stanford Vision & Perception Neuroscience Lab.

Hierarchical organization

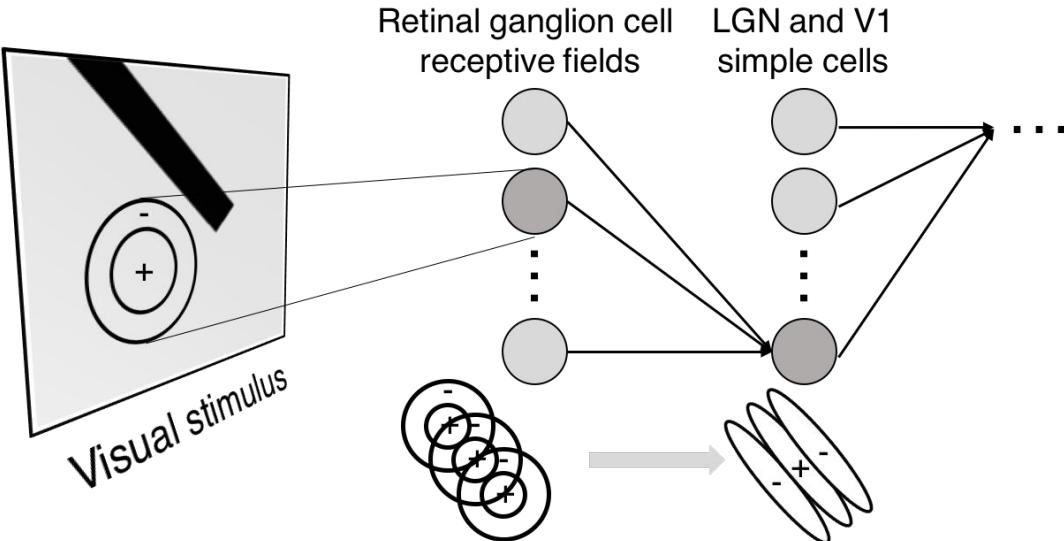
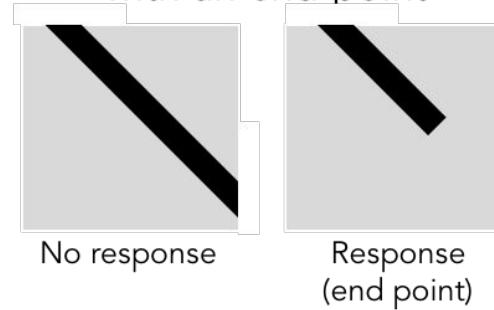


Illustration of hierarchical organization in early visual pathways by Lane McIntosh, copyright CS231n 2017

Simple cells:
Response to light orientation

Complex cells:
Response to light orientation and movement

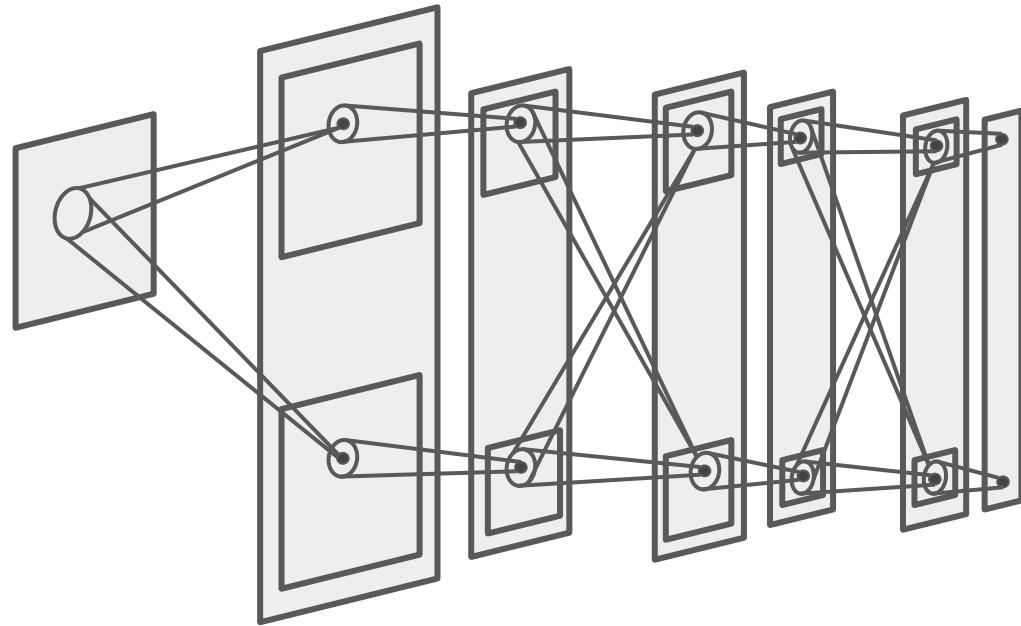
Hypercomplex cells:
response to movement with an end point



A bit of history:

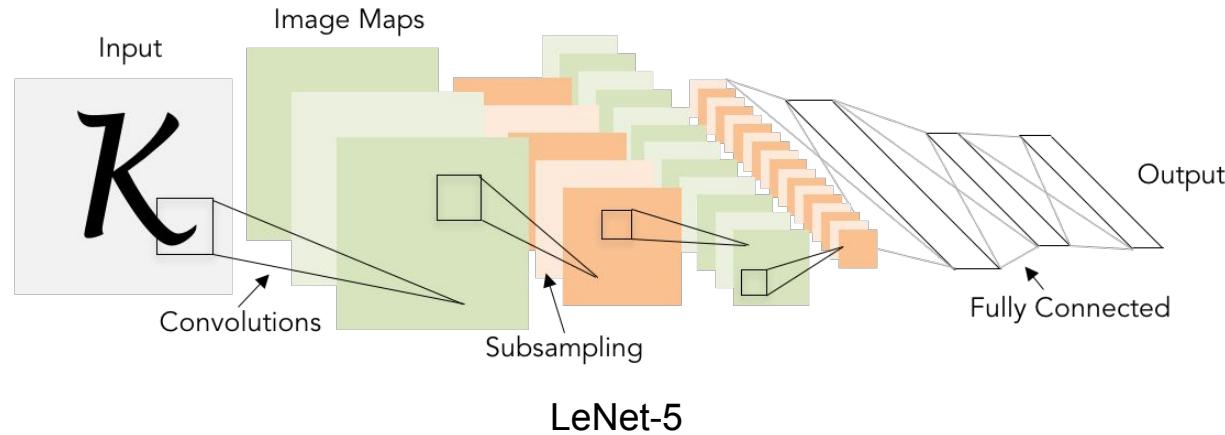
Neocognitron [Fukushima 1980]

“sandwich” architecture (SCSCSC...)
simple cells: modifiable parameters
complex cells: perform pooling



A bit of history: Gradient-based learning applied to document recognition

[LeCun, Bottou, Bengio, Haffner 1998]



A bit of history: ImageNet Classification with Deep Convolutional Neural Networks *[Krizhevsky, Sutskever, Hinton, 2012]*

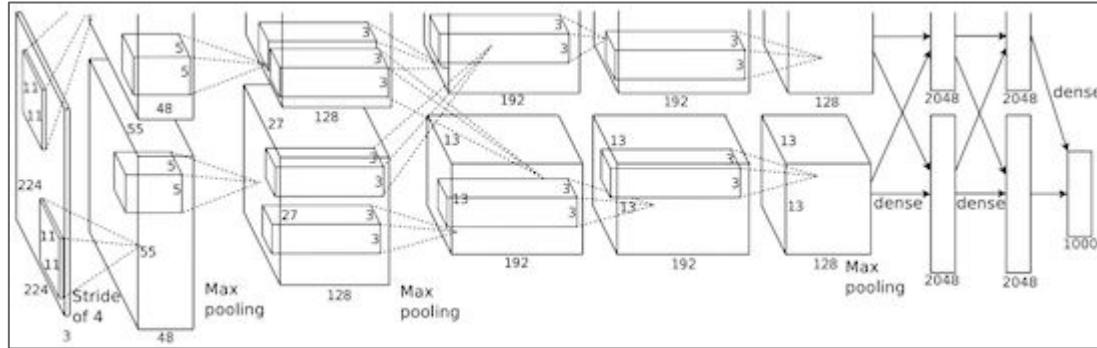


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

“AlexNet”

Fast-forward to today: ConvNets are everywhere

Classification



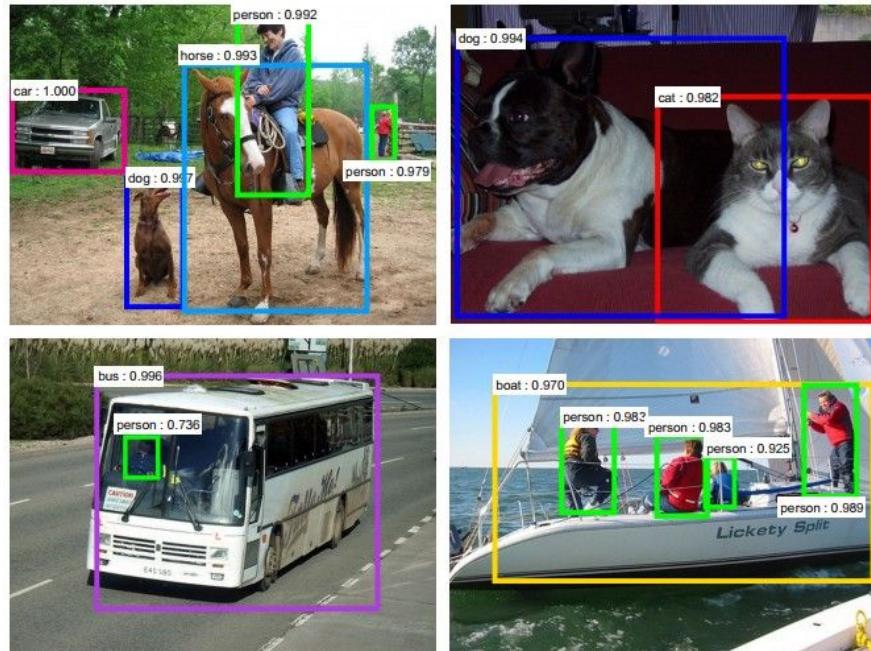
Retrieval



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

Fast-forward to today: ConvNets are everywhere

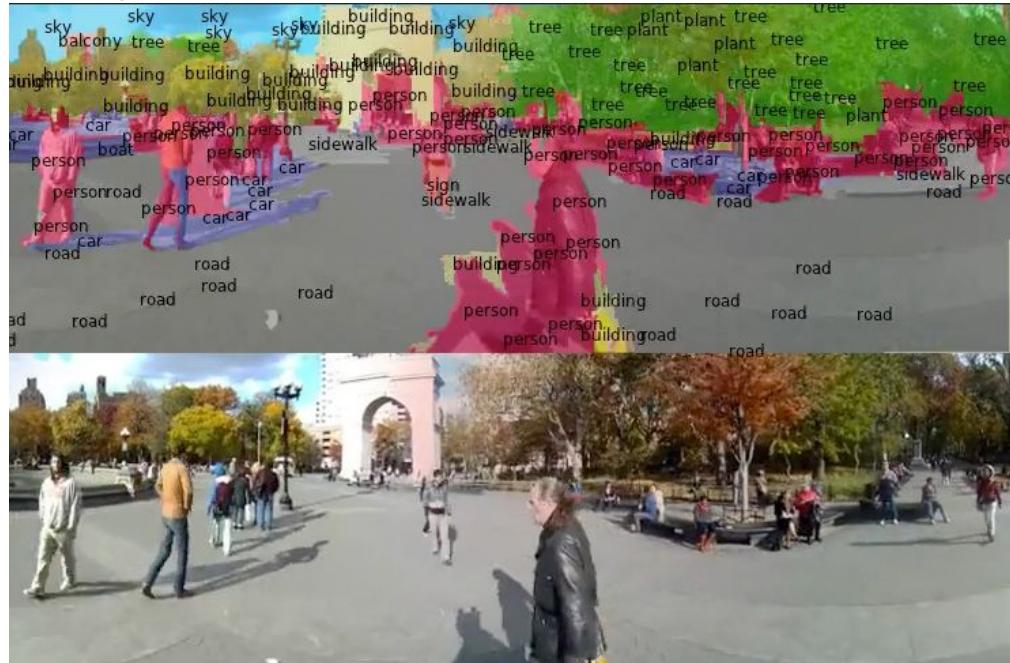
Detection



Figures copyright Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun, 2015. Reproduced with permission.

[Faster R-CNN: Ren, He, Girshick, Sun 2015]

Segmentation



Figures copyright Clement Farabet, 2012.
Reproduced with permission.

[Farabet et al., 2012]

Fast-forward to today: ConvNets are everywhere



self-driving cars

Photo by Lane McIntosh. Copyright CS231n 2017.



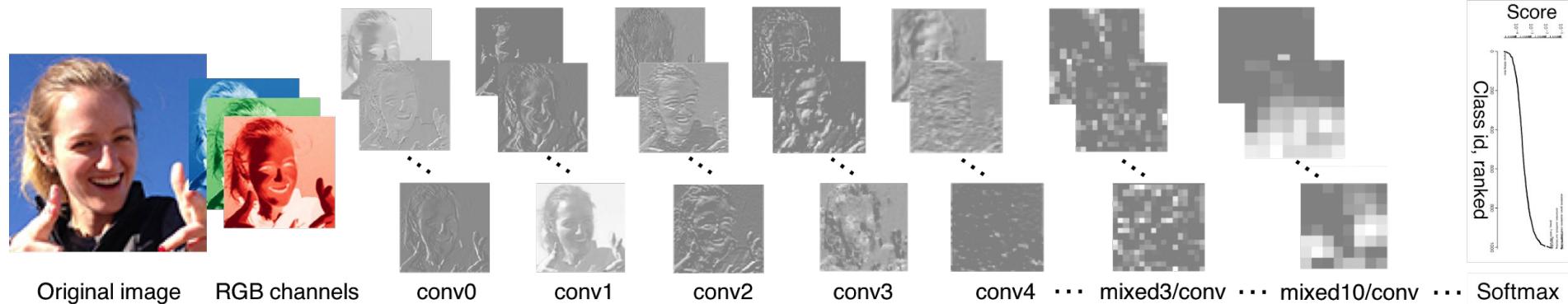
[This image](#) by GBPublic_PR is licensed under [CC-BY 2.0](#)

NVIDIA Tesla line

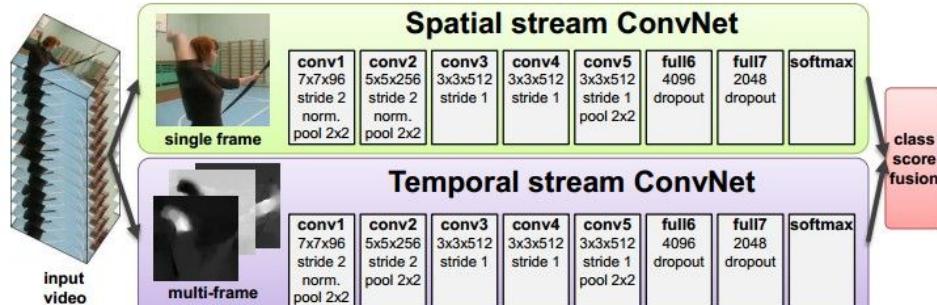
(these are the GPUs on rye01.stanford.edu)

Note that for embedded systems a typical setup would involve NVIDIA Tegras, with integrated GPU and ARM-based CPU cores.

Fast-forward to today: ConvNets are everywhere



Activations of [inception-v3 architecture](#) [Szegedy et al. 2015] to image of Emma McIntosh, used with permission. Figure and architecture not from Taigman et al. 2014.



[Simonyan et al. 2014]

Figures copyright Simonyan et al., 2014.
Reproduced with permission.

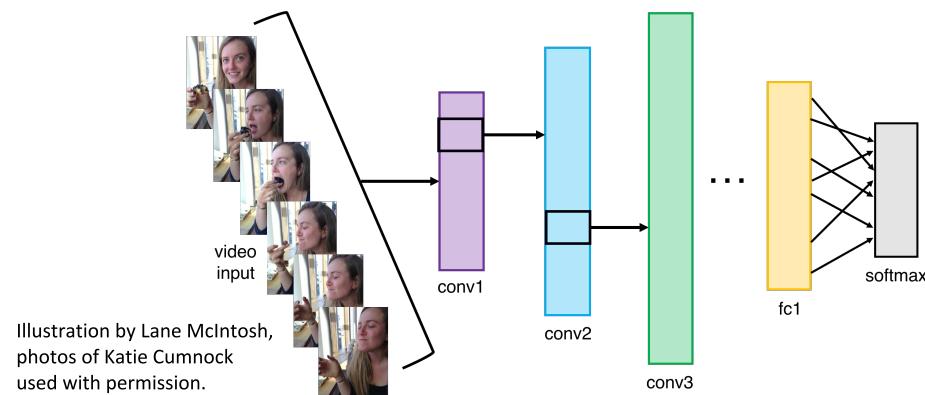


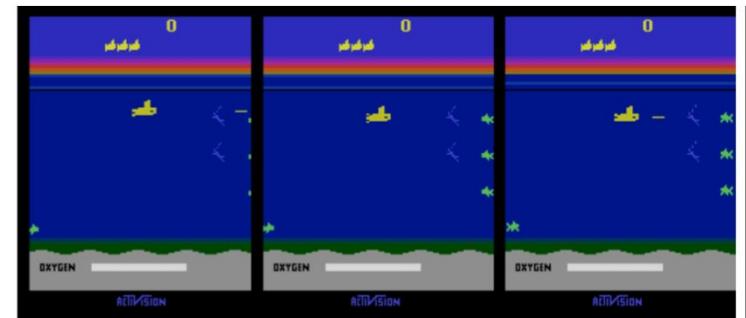
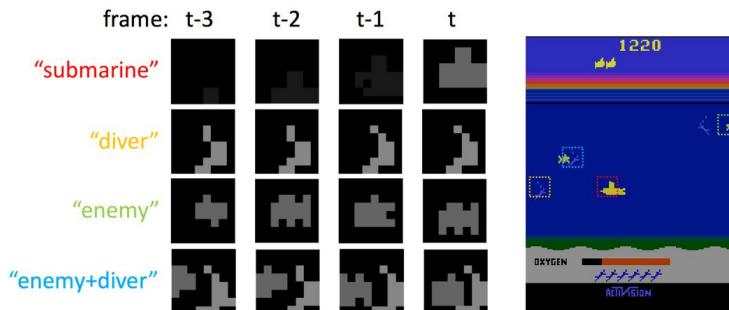
Illustration by Lane McIntosh,
photos of Katie Cumnock
used with permission.

Fast-forward to today: ConvNets are everywhere



Images are examples of pose estimation, not actually from Toshev & Szegedy 2014. Copyright Lane McIntosh.

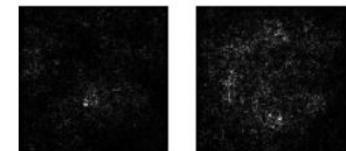
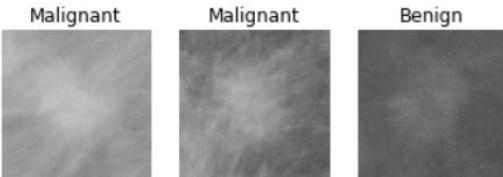
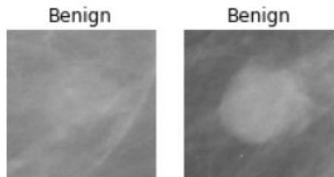
[Toshev, Szegedy 2014]



[Guo et al. 2014]

Figures copyright Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard Lewis, and Xiaoshi Wang, 2014. Reproduced with permission.

Fast-forward to today: ConvNets are everywhere



[Levy et al. 2016]

Figure copyright Levy et al. 2016.
Reproduced with permission.



[Dieleman et al. 2014]

From left to right: [public domain by NASA](#), usage [permitted](#) by
ESA/Hubble, [public domain by NASA](#), and [public domain](#).



Photos by Lane McIntosh.
Copyright CS231n 2017.

[Sermanet et al. 2011]
[Ciresan et al.]

[This image](#) by Christin Khan is in the public domain and originally came from the U.S. NOAA.



Whale recognition, Kaggle Challenge

Photo and figure by Lane McIntosh; not actual example from Mnih and Hinton, 2010 paper.



Mnih and Hinton, 2010

No errors



A white teddy bear sitting in the grass



A man riding a wave on top of a surfboard

Minor errors



A man in a baseball uniform throwing a ball



A cat sitting on a suitcase on the floor

Somewhat related



A woman is holding a cat in her hand



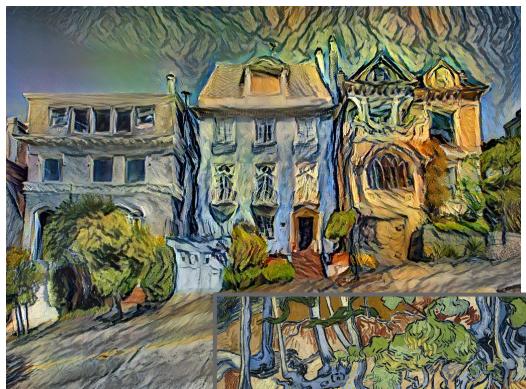
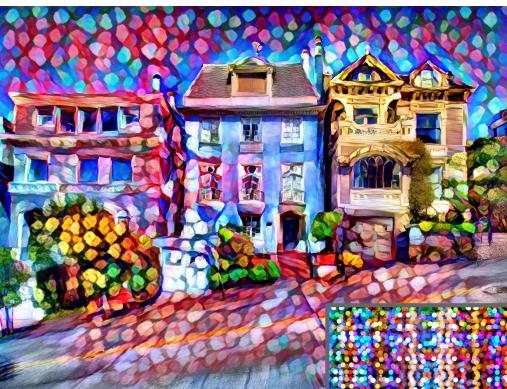
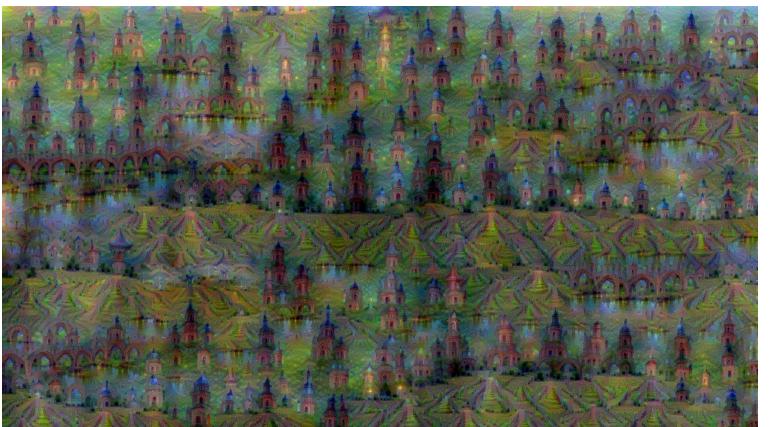
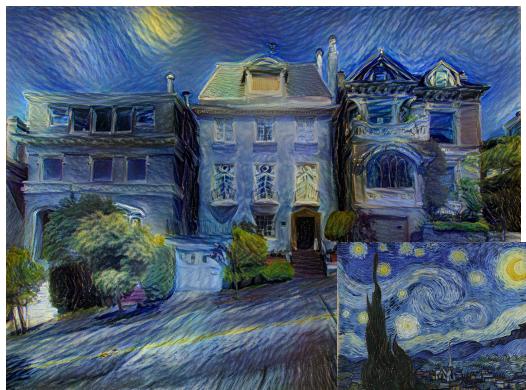
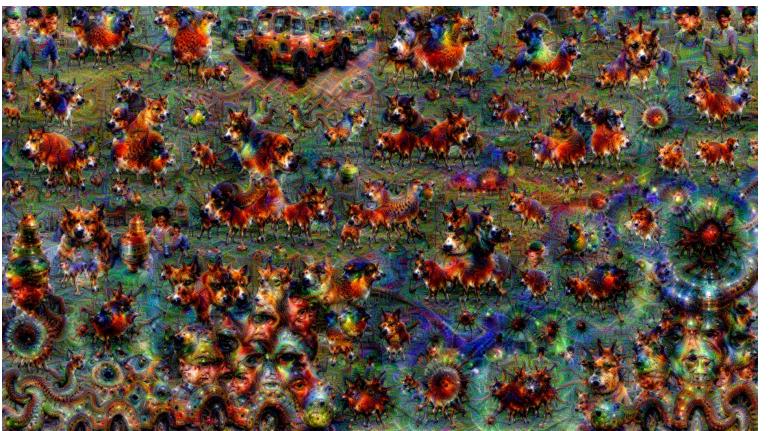
A woman standing on a beach holding a surfboard

Image Captioning

[Vinyals et al., 2015]
[Karpathy and Fei-Fei, 2015]

All images are CC0 Public domain:
<https://pixabay.com/en/luggage-antique-cat-1643010/>
<https://pixabay.com/en/teddy-plush-bears-cute-teddy-bear-1623436/>
<https://pixabay.com/en/surf-wave-summer-sport-litoral-1668716/>
<https://pixabay.com/en/woman-female-model-portrait-adult-983967/>
<https://pixabay.com/en/handstand-lake-meditation-496008/>
<https://pixabay.com/en/baseball-player-shortstop-infield-1045263/>

Captions generated by Justin Johnson using [Neuraltalk2](#)



Figures copyright Justin Johnson, 2015. Reproduced with permission. Generated using the Inceptionism approach from a [blog post](#) by Google Research.

[Original image](#) is CC0 public domain

[Starry Night](#) and [Tree Roots](#) by Van Gogh are in the public domain

[Bokeh image](#) is in the public domain

Stylized images copyright Justin Johnson, 2017;
reproduced with permission

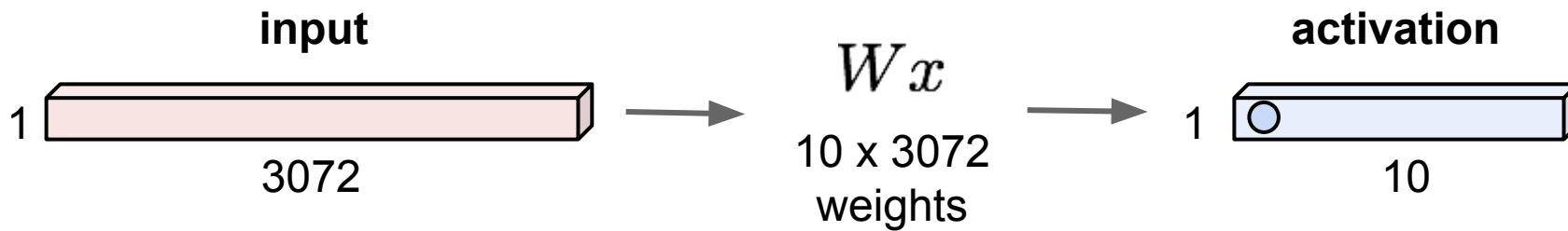
Gatys et al, "Image Style Transfer using Convolutional Neural Networks", CVPR 2016
Gatys et al, "Controlling Perceptual Factors in Neural Style Transfer", CVPR 2017

Convolutional Neural Networks

(First without the brain stuff)

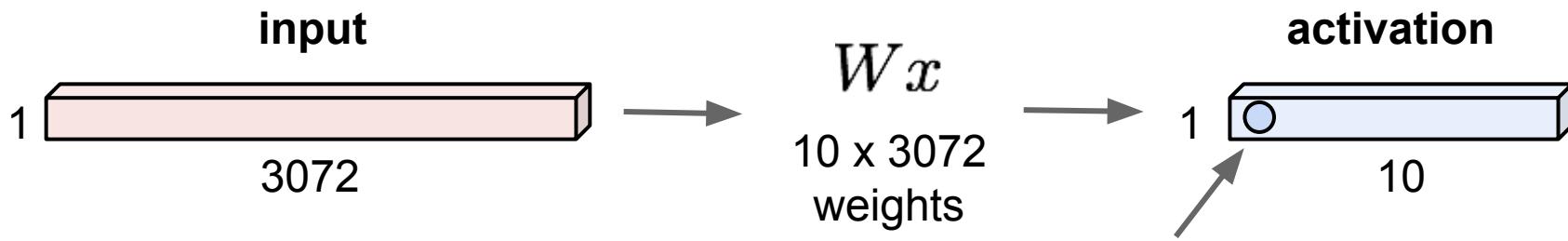
Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1



Fully Connected Layer

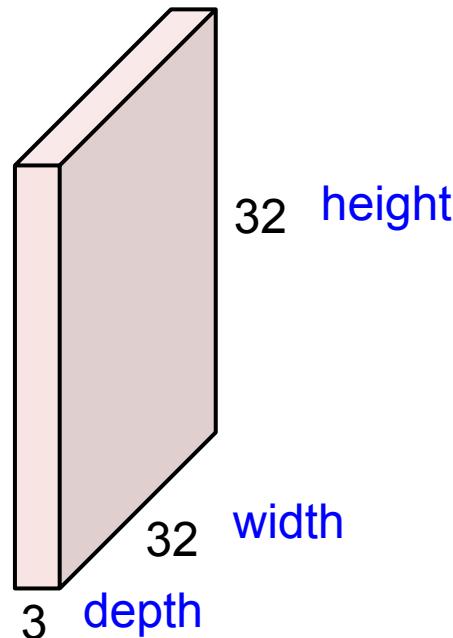
32x32x3 image -> stretch to 3072 x 1



1 number:
the result of taking a dot product
between a row of W and the input
(a 3072-dimensional dot product)

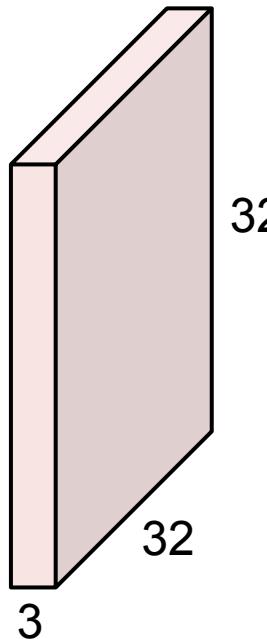
Convolution Layer

32x32x3 image -> preserve spatial structure



Convolution Layer

32x32x3 image



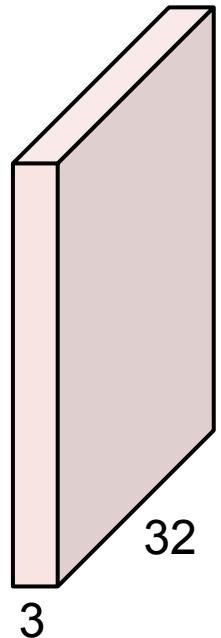
5x5x3 filter



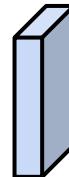
Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

Convolution Layer

32x32x3 image



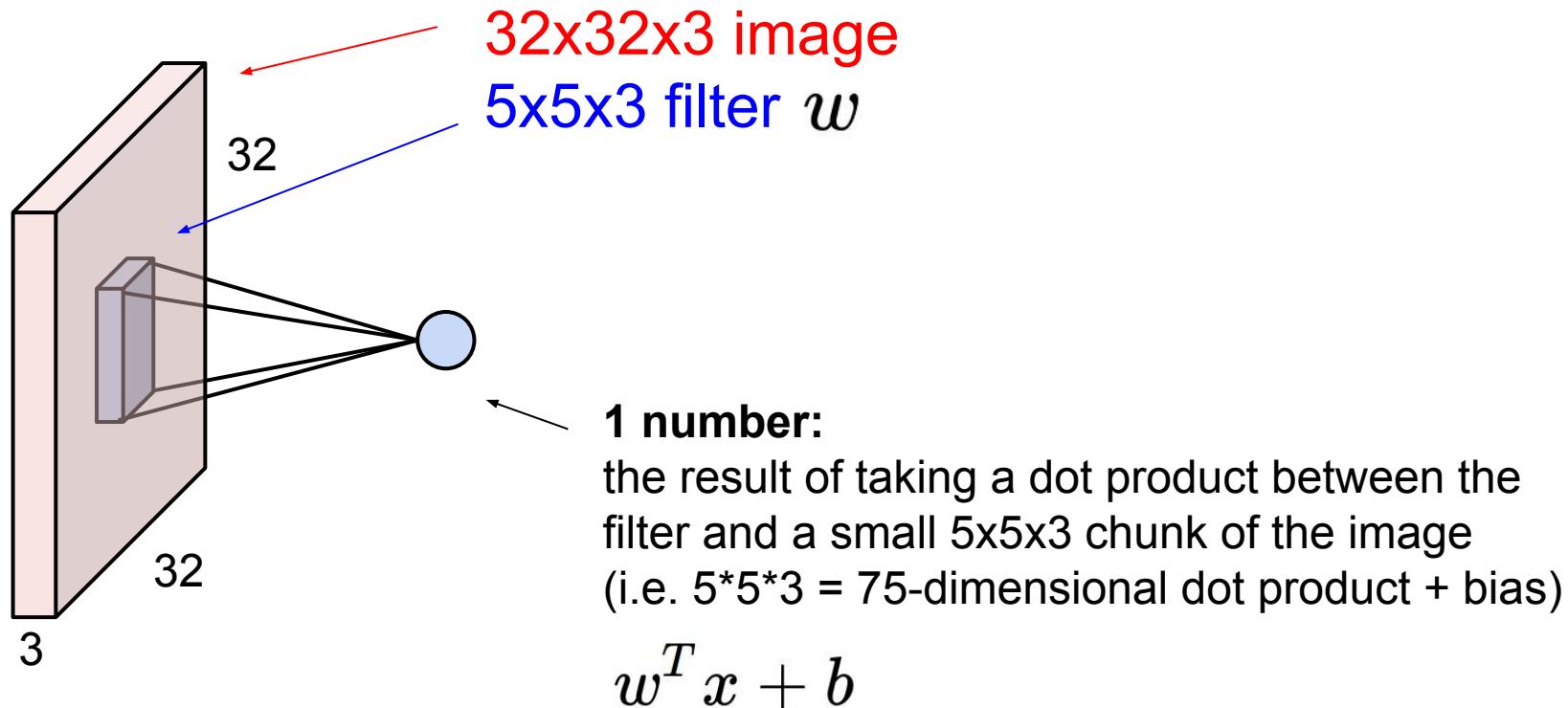
5x5x3 filter



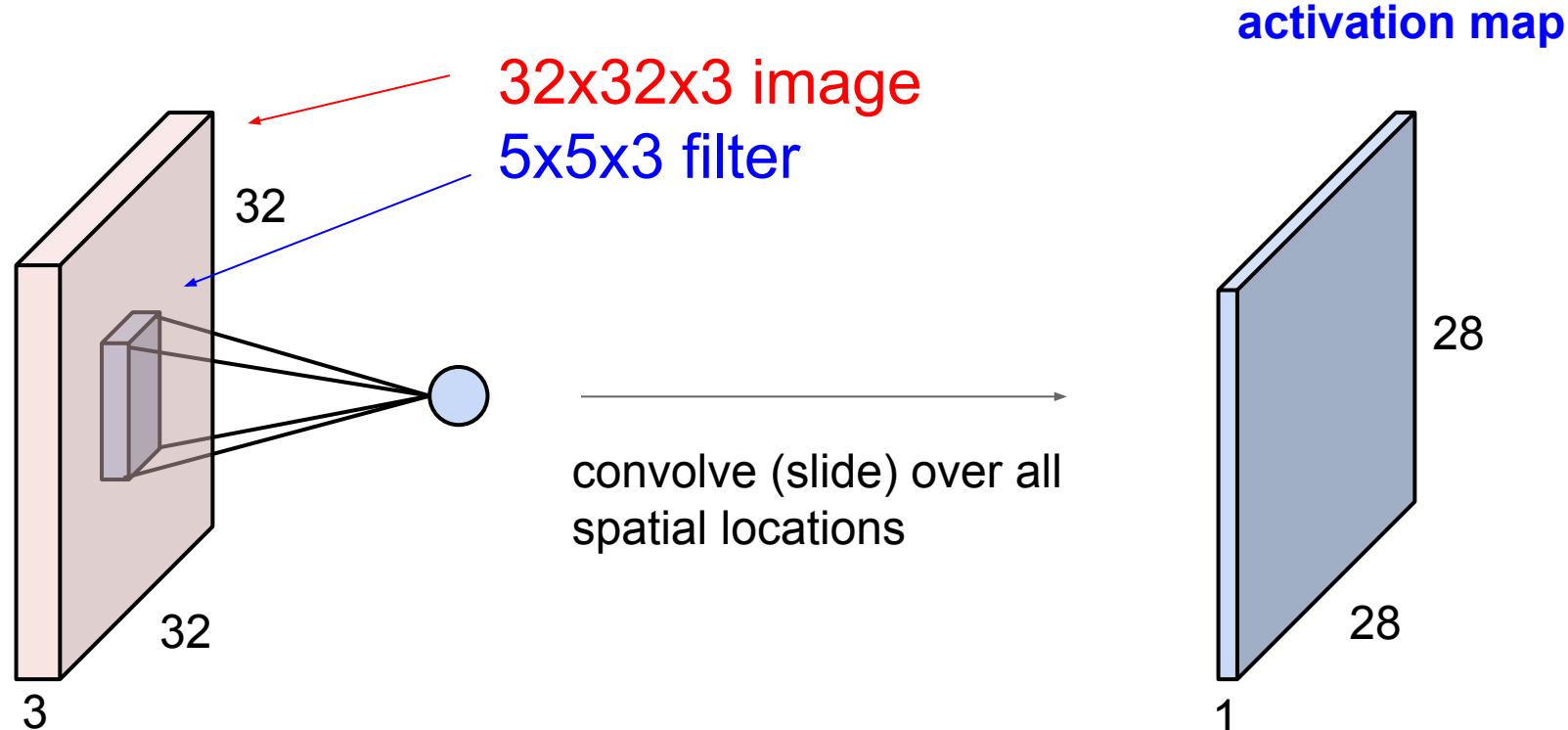
Filters always extend the full depth of the input volume

Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

Convolution Layer

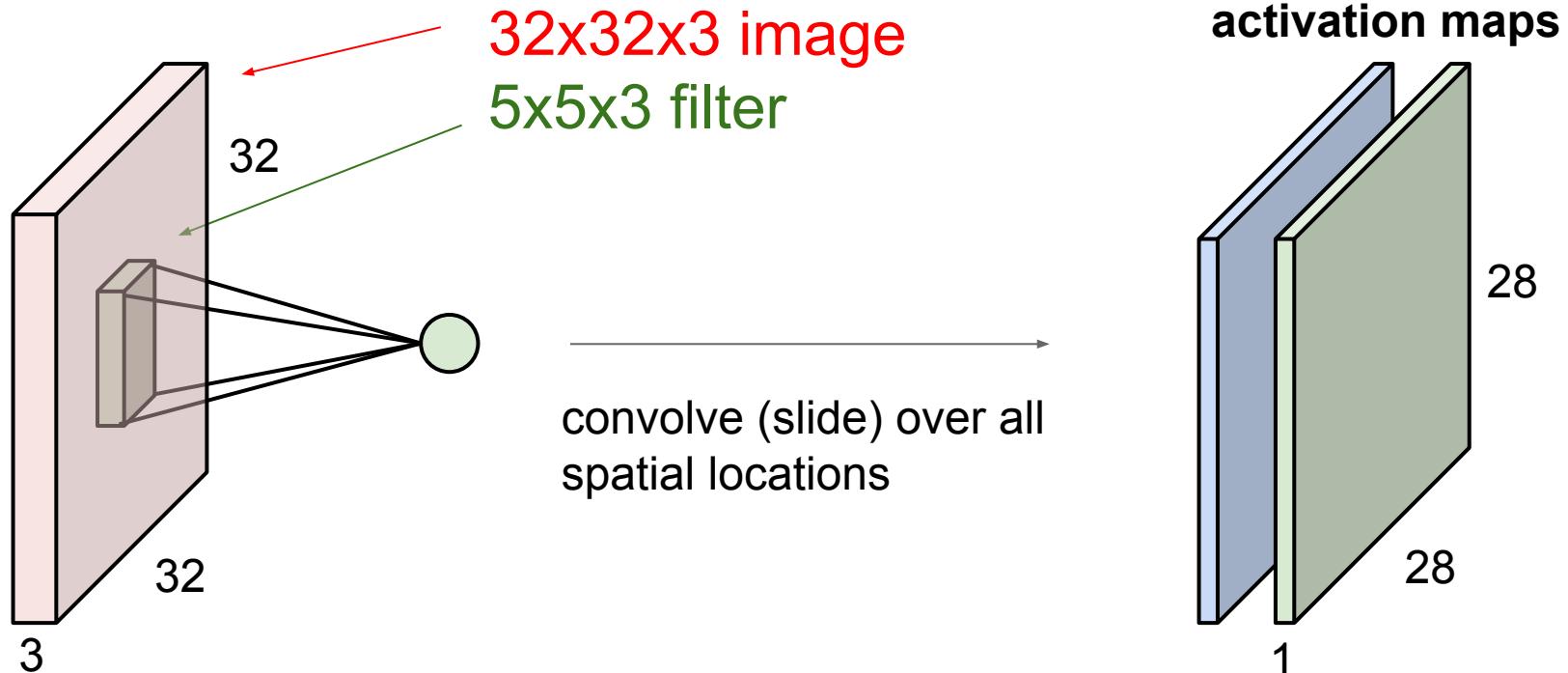


Convolution Layer

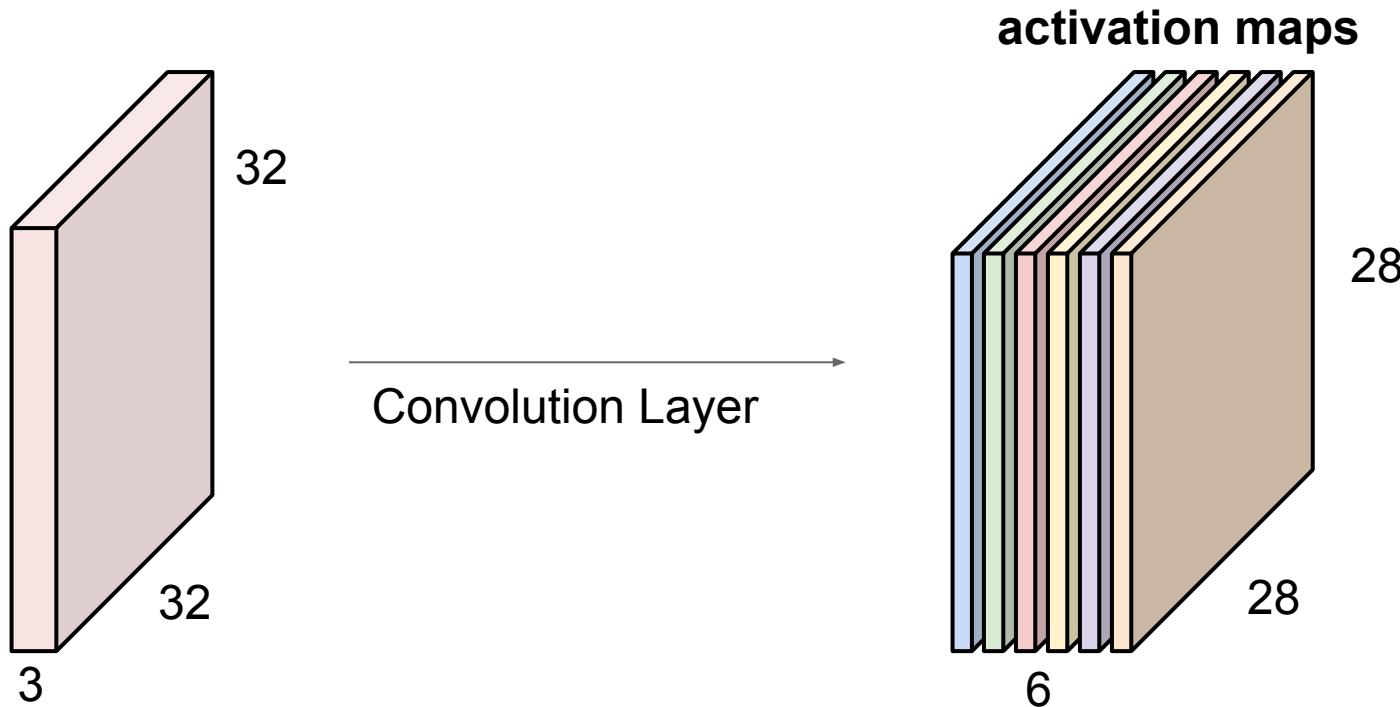


Convolution Layer

consider a second, green filter

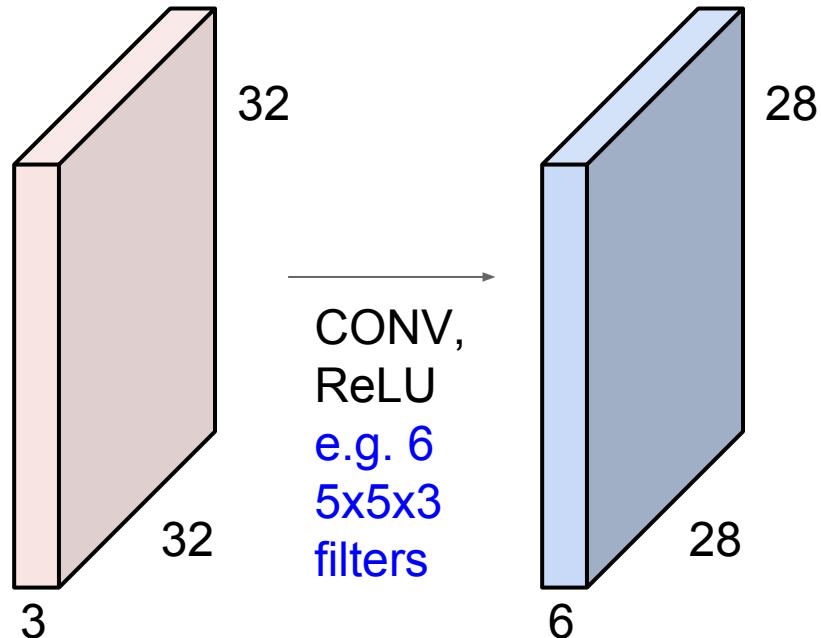


For example, if we had 6 5×5 filters, we'll get 6 separate activation maps:

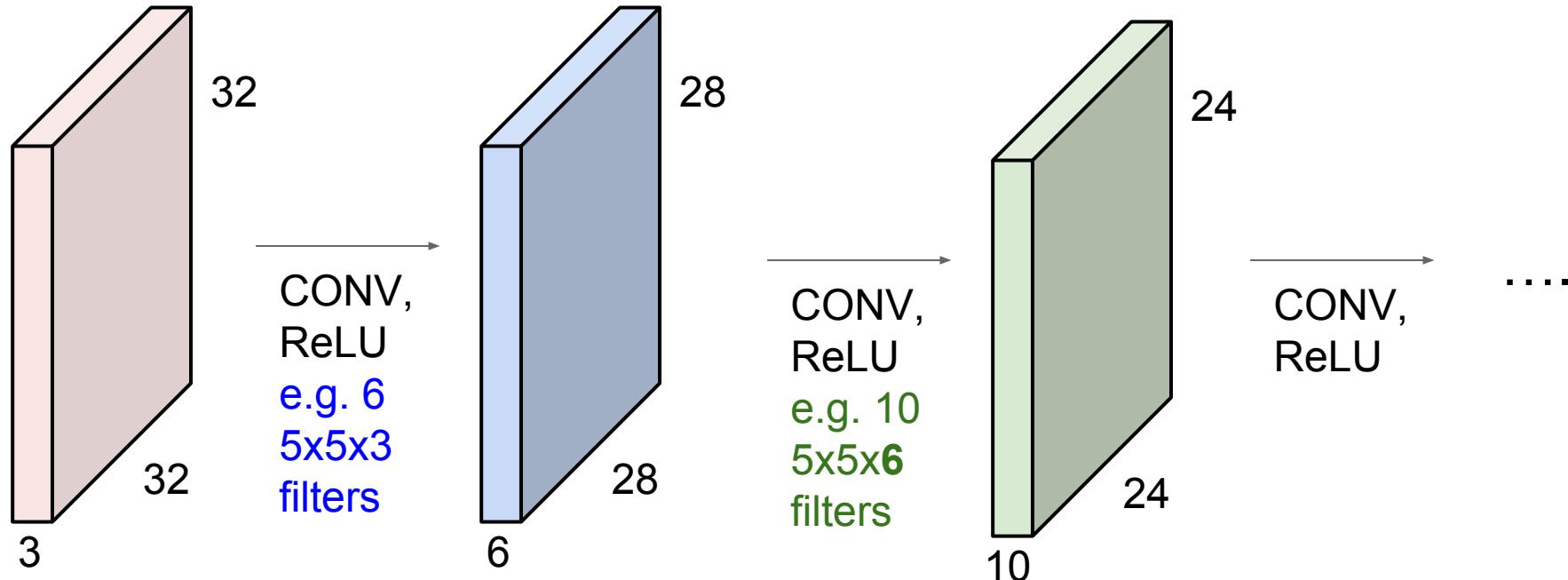


We stack these up to get a “new image” of size $28 \times 28 \times 6$!

Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



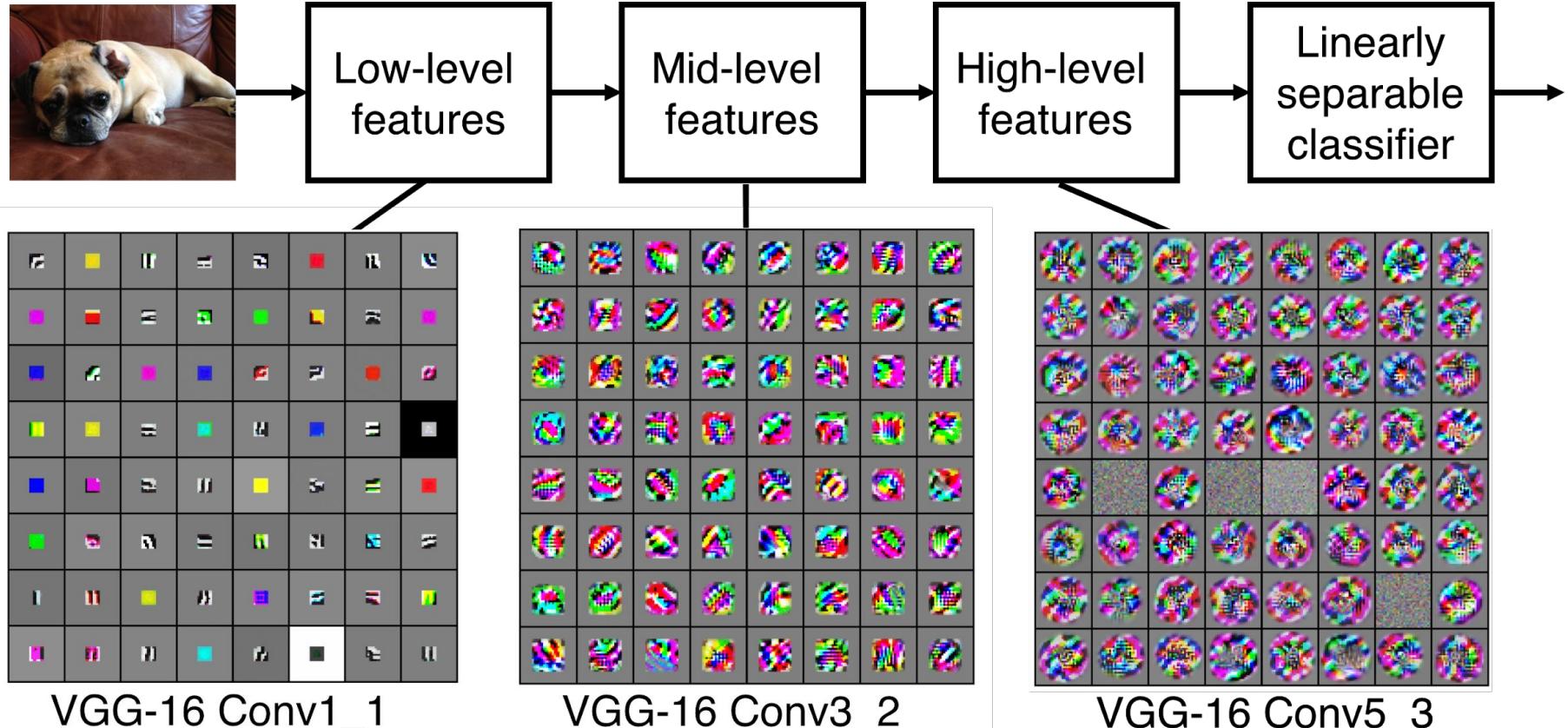
Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



Preview

[Zeiler and Fergus 2013]

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

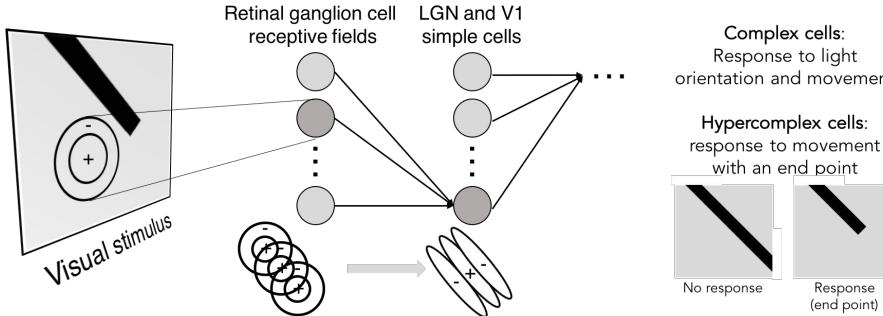
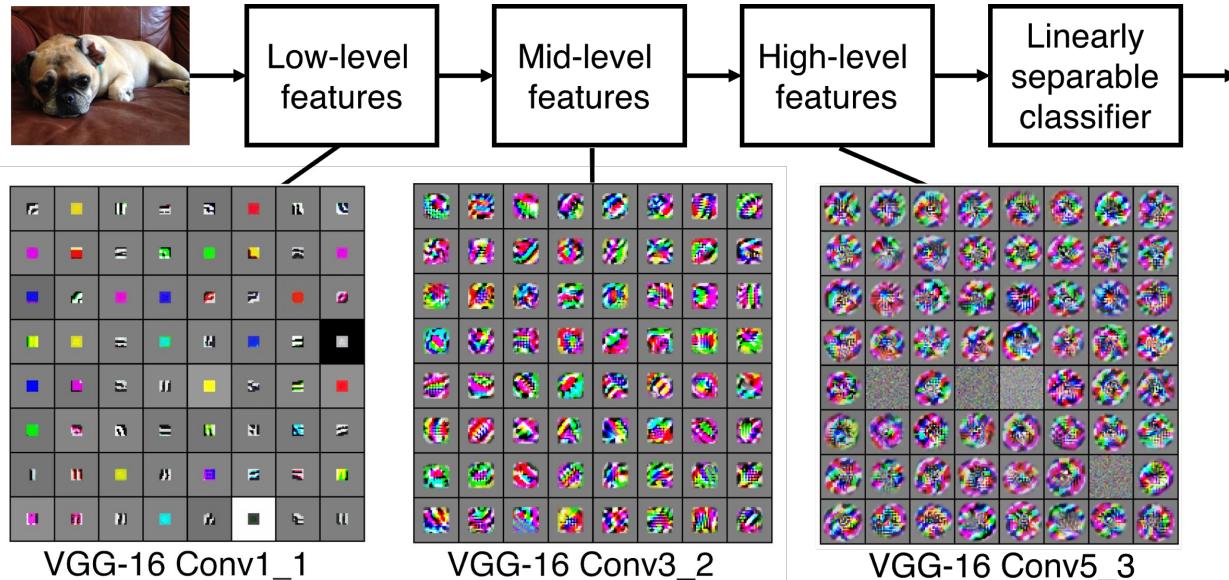


VGG-16 Conv1_1

VGG-16 Conv3_2

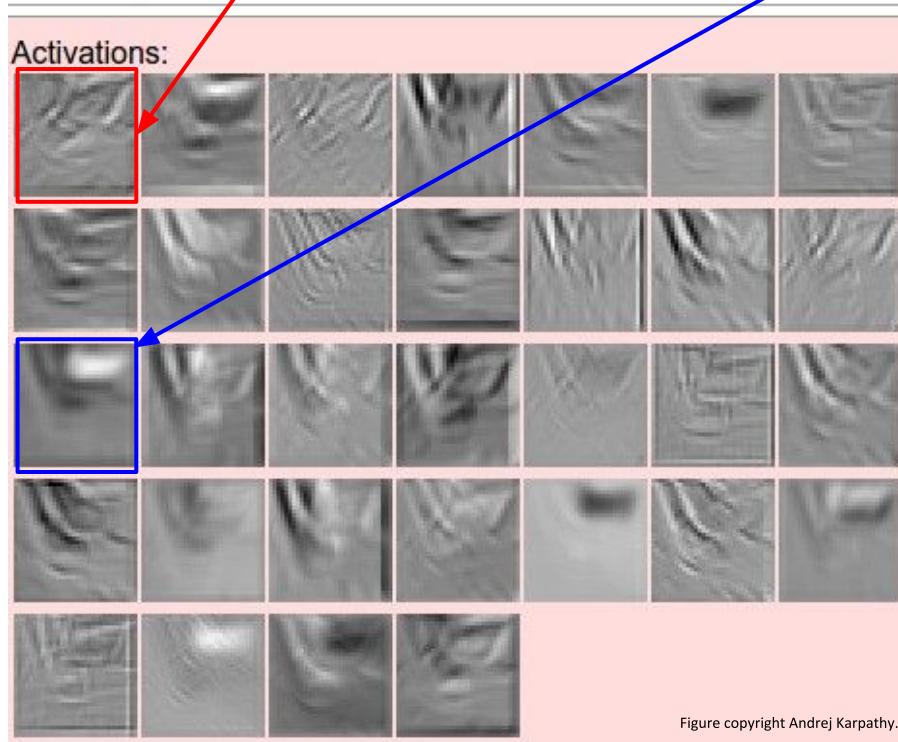
VGG-16 Conv5_3

Preview





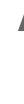
one filter =>
one activation map



example 5x5 filters
(32 total)

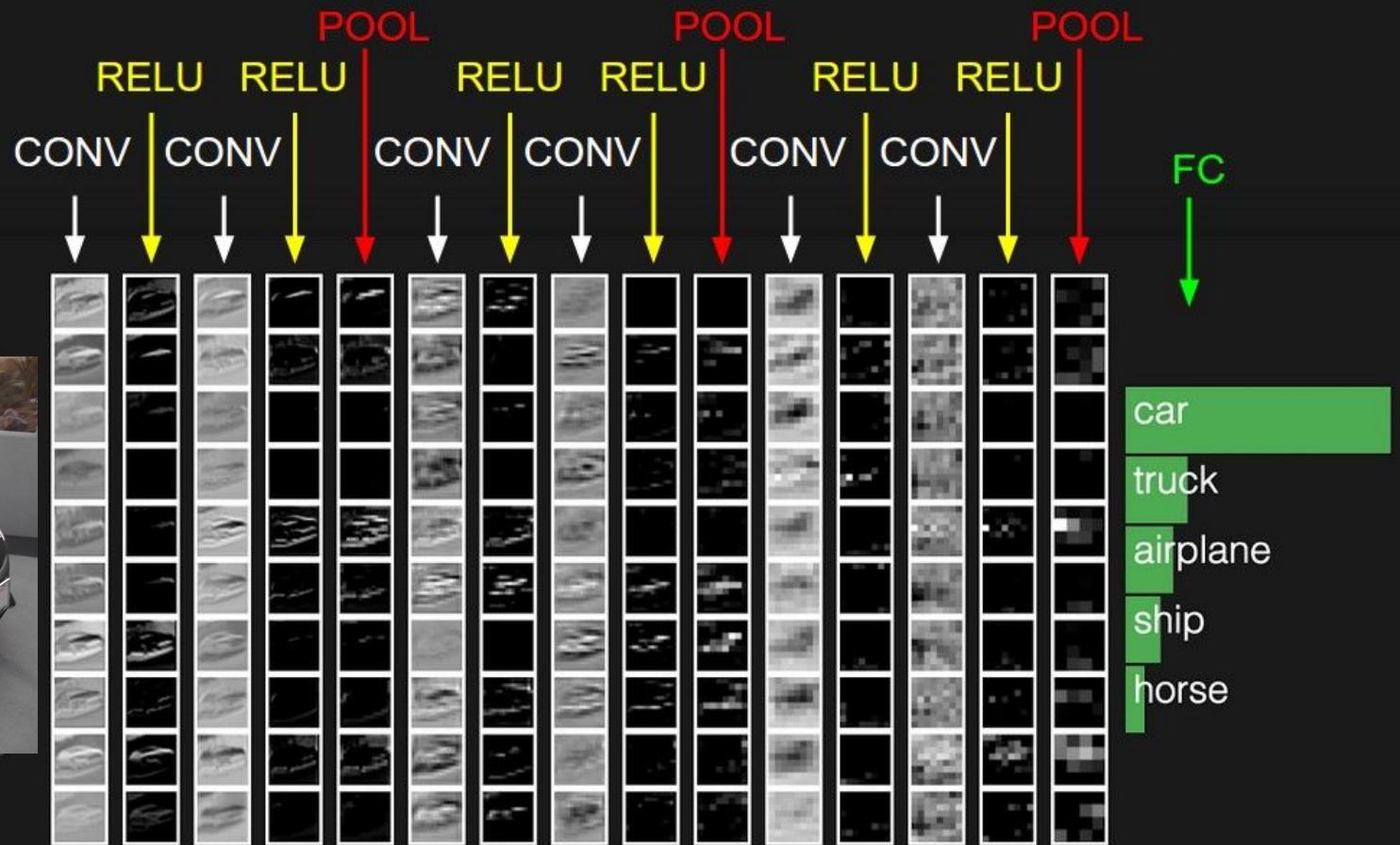
We call the layer convolutional
because it is related to convolution
of two signals:

$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1, n_2] \cdot g[x - n_1, y - n_2]$$

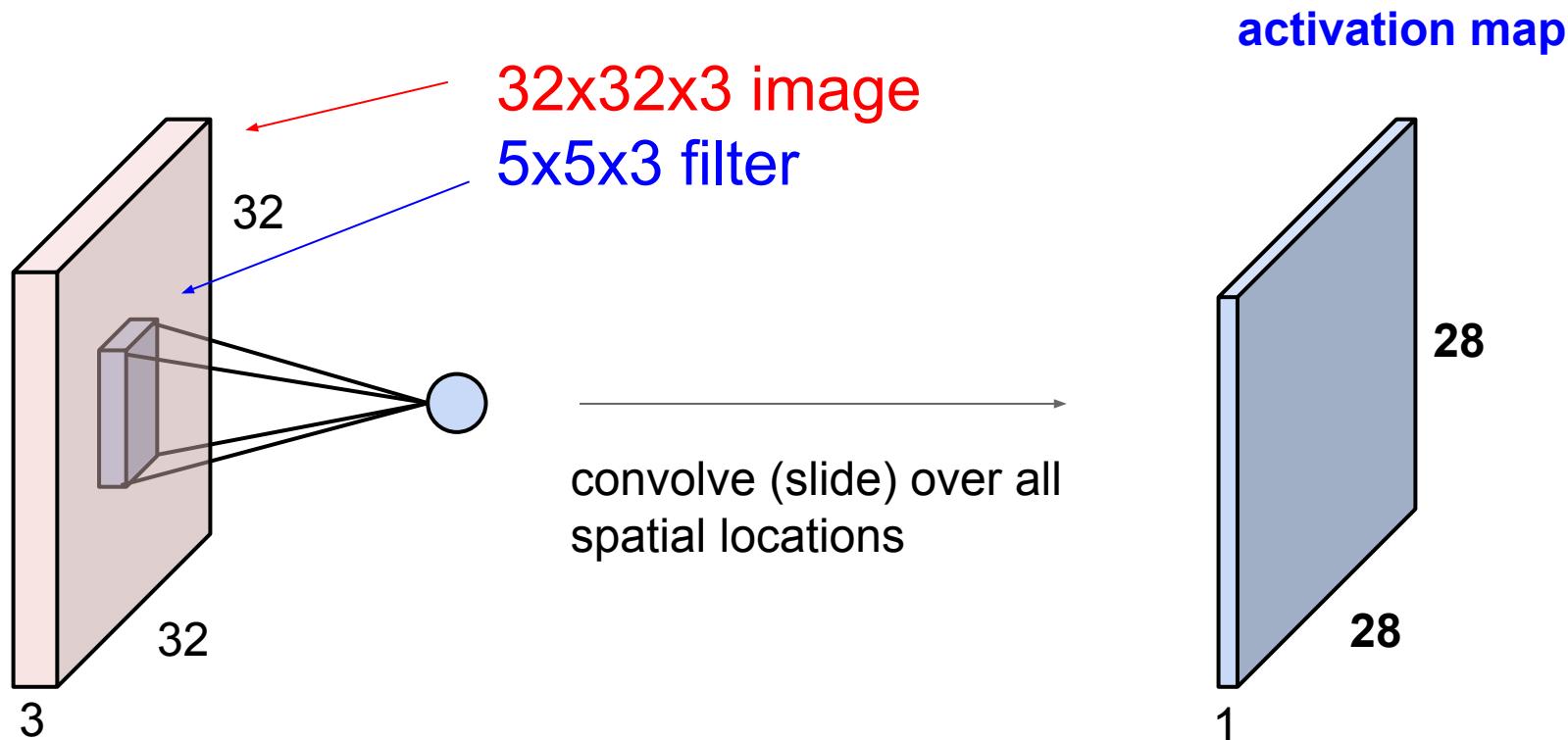


elementwise multiplication and sum of
a filter and the signal (image)

preview:

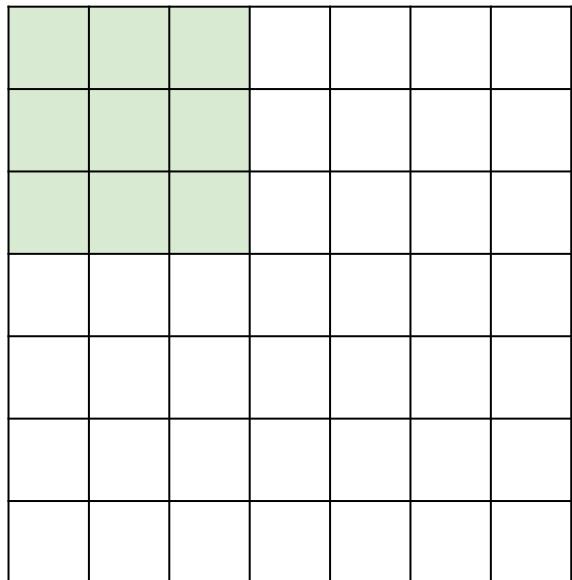


A closer look at spatial dimensions:



A closer look at spatial dimensions:

7

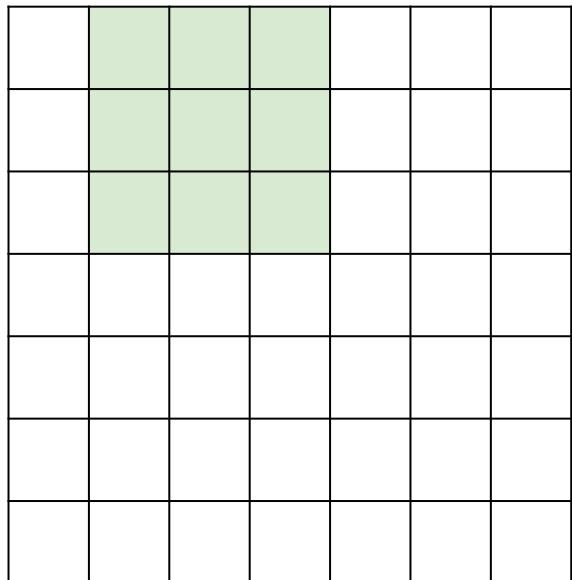


7x7 input (spatially)
assume 3x3 filter

7

A closer look at spatial dimensions:

7

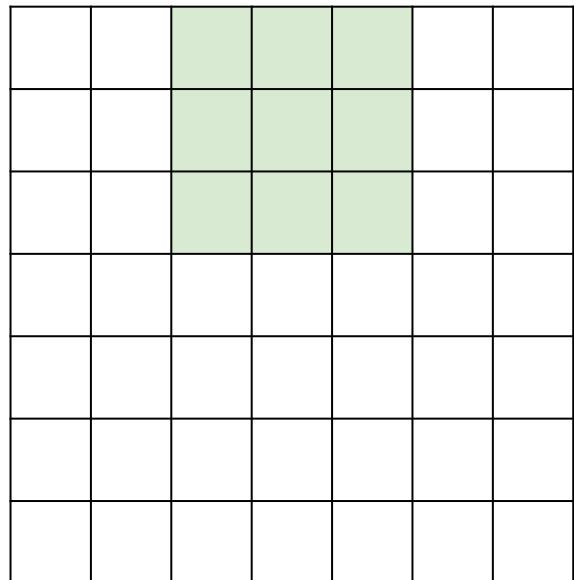


7x7 input (spatially)
assume 3x3 filter

7

A closer look at spatial dimensions:

7

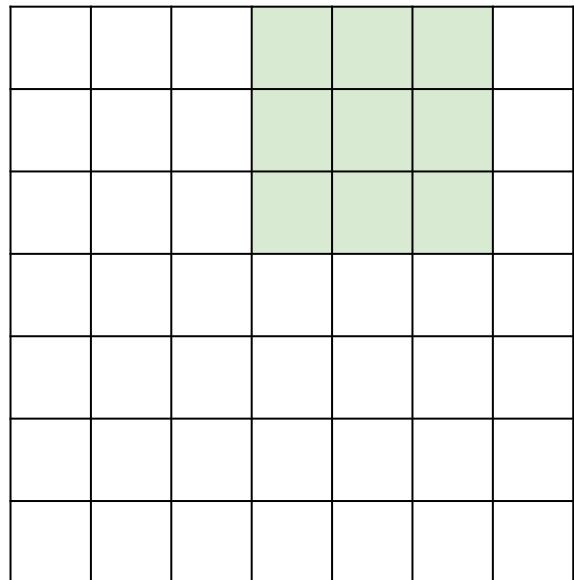


7x7 input (spatially)
assume 3x3 filter

7

A closer look at spatial dimensions:

7

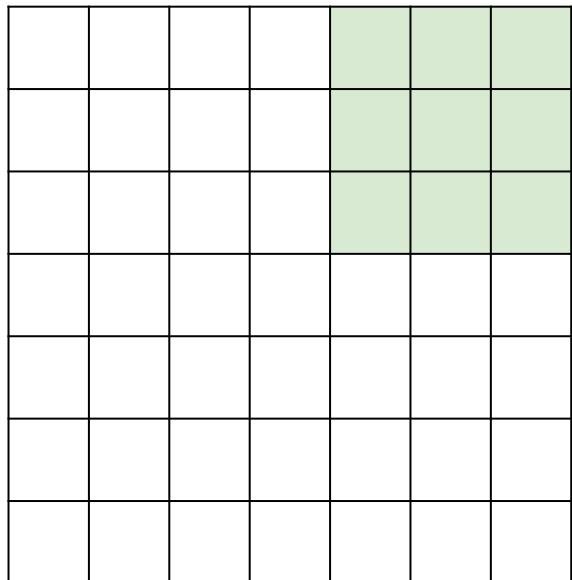


7x7 input (spatially)
assume 3x3 filter

7

A closer look at spatial dimensions:

7

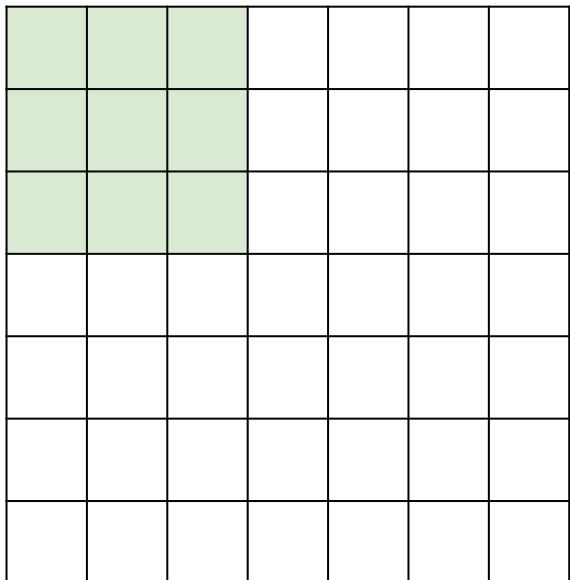


7x7 input (spatially)
assume 3x3 filter

=> 5x5 output

A closer look at spatial dimensions:

7

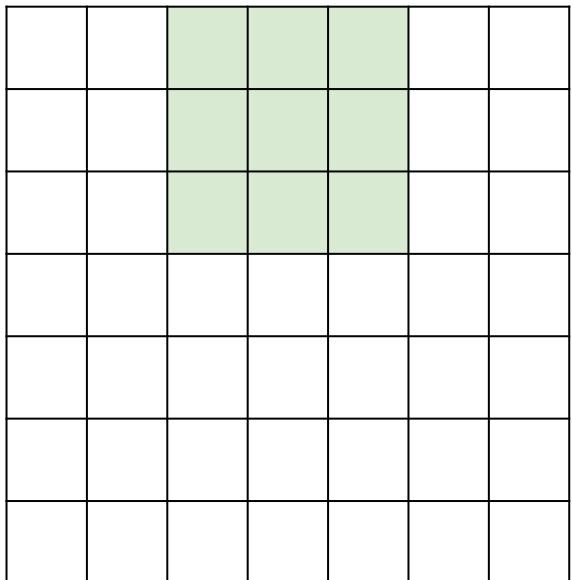


7

7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:

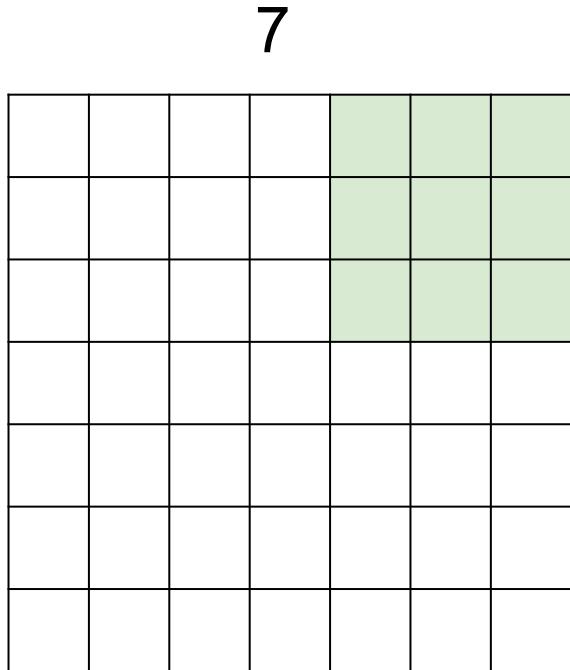
7



7

7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

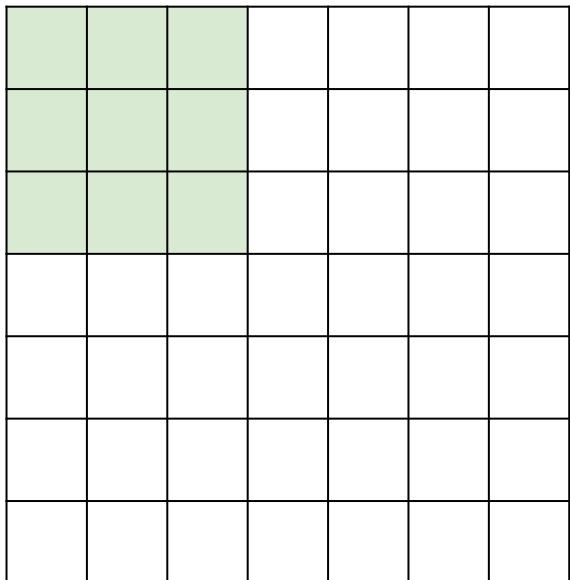
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

A closer look at spatial dimensions:

7

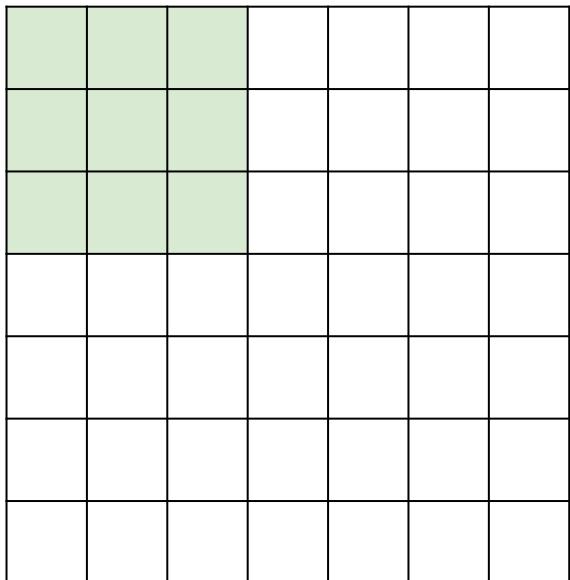


7

7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

A closer look at spatial dimensions:

7



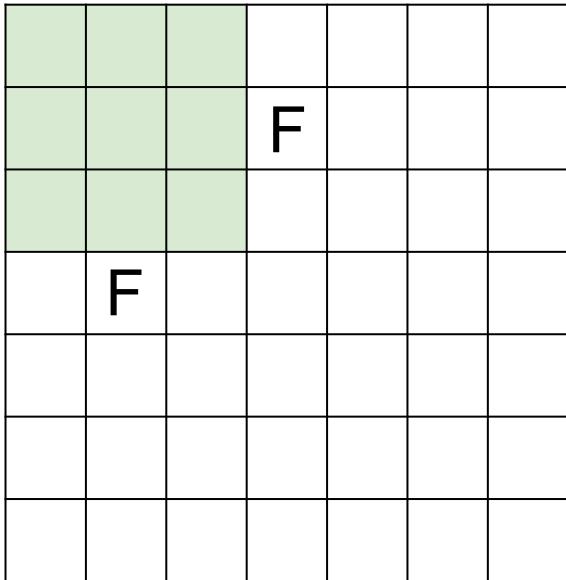
7

7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

doesn't fit!

cannot apply 3x3 filter on
7x7 input with stride 3.

N



N

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

e.g. $N = 7, F = 3$:

$$\text{stride } 1 \Rightarrow (7 - 3)/1 + 1 = 5$$

$$\text{stride } 2 \Rightarrow (7 - 3)/2 + 1 = 3$$

$$\text{stride } 3 \Rightarrow (7 - 3)/3 + 1 = 2.33 : \backslash$$

In practice: Common to zero pad the border

0	0	0	0	0	0		
0							
0							
0							
0							

e.g. input 7x7

3x3 filter, applied with stride 1

pad with 1 pixel border => what is the output?

(recall:)

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

In practice: Common to zero pad the border

0	0	0	0	0	0		
0							
0							
0							
0							

e.g. input 7x7

3x3 filter, applied with stride 1

pad with 1 pixel border => what is the output?

7x7 output!

In practice: Common to zero pad the border

0	0	0	0	0	0		
0							
0							
0							
0							

e.g. input 7x7

3x3 filter, applied with stride 1

pad with 1 pixel border => what is the output?

7x7 output!

in general, common to see CONV layers with stride 1, filters of size $F \times F$, and zero-padding with $(F-1)/2$. (will preserve size spatially)

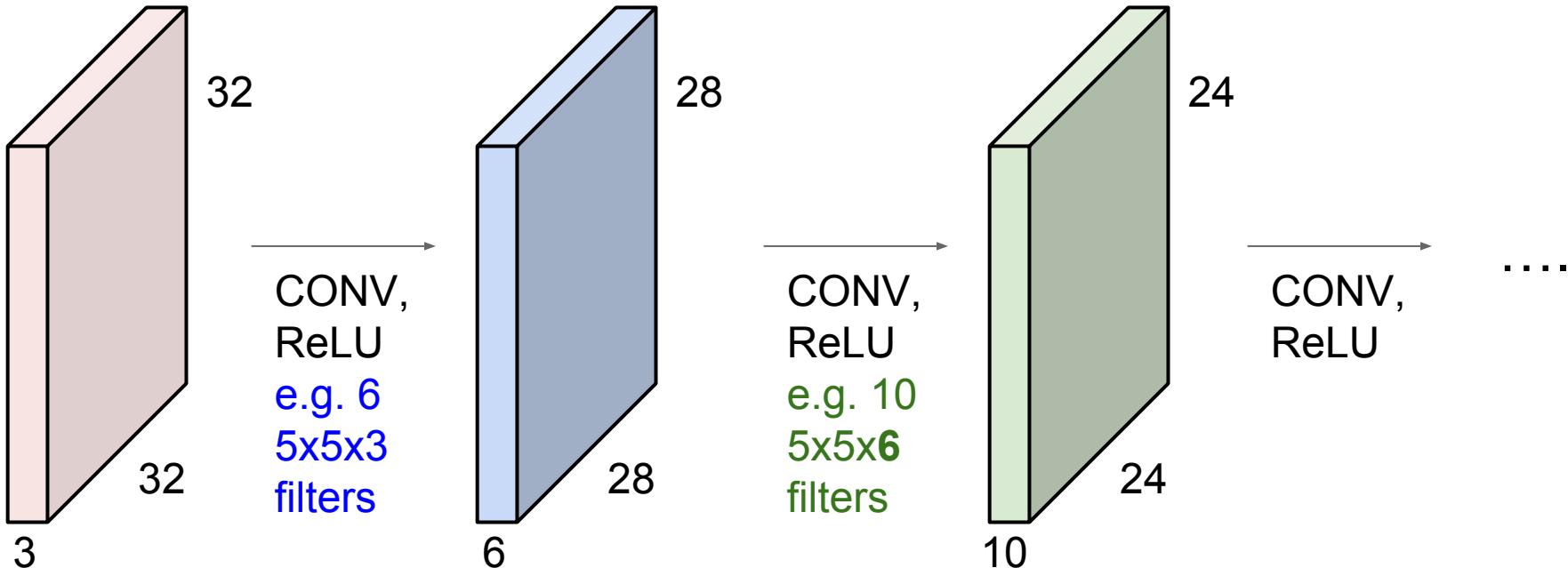
e.g. $F = 3 \Rightarrow$ zero pad with 1

$F = 5 \Rightarrow$ zero pad with 2

$F = 7 \Rightarrow$ zero pad with 3

Remember back to...

E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially!
(32 -> 28 -> 24 ...). Shrinking too fast is not good, doesn't work well.

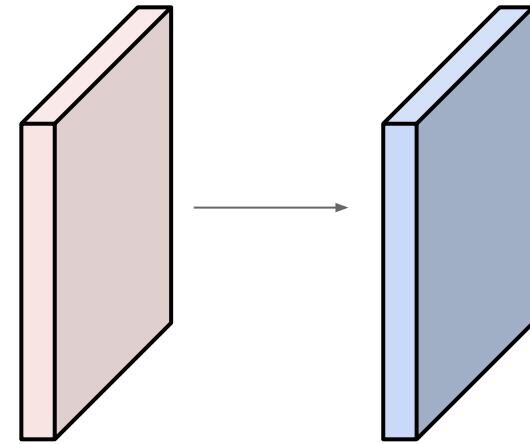


Examples time:

Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

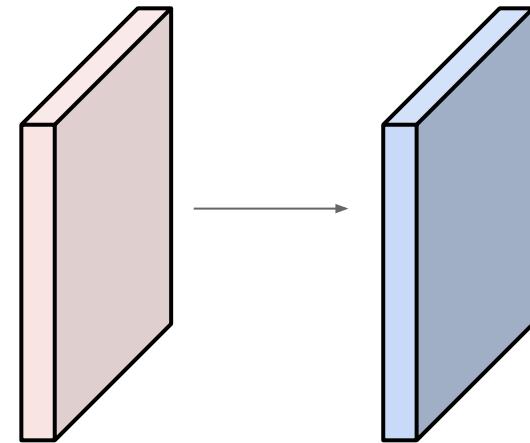
Output volume size: ?



Examples time:

Input volume: **32x32x3**

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



Output volume size:

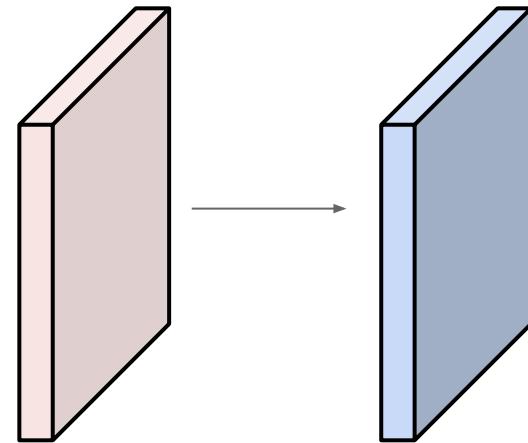
$(32+2*2-5)/1+1 = 32$ spatially, so

32x32x10

Examples time:

Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

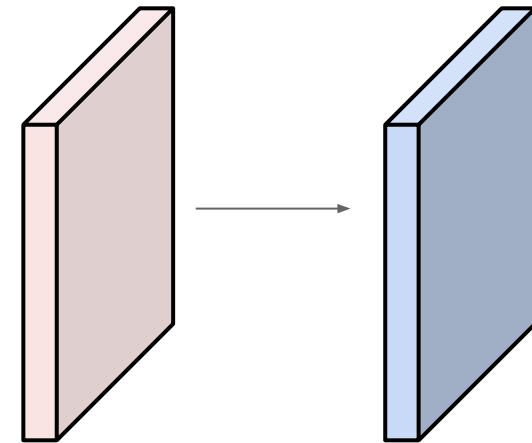


Number of parameters in this layer?

Examples time:

Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2



Number of parameters in this layer?

each filter has $5*5*3 + 1 = 76$ params (+1 for bias)
=> $76*10 = 760$

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
 - Number of filters K ,
 - their spatial extent F ,
 - the stride S ,
 - the amount of zero padding P .
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
 - $W_2 = (W_1 - F + 2P)/S + 1$
 - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
 - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d -th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d -th filter over the input volume with a stride of S , and then offset by d -th bias.

Common settings:

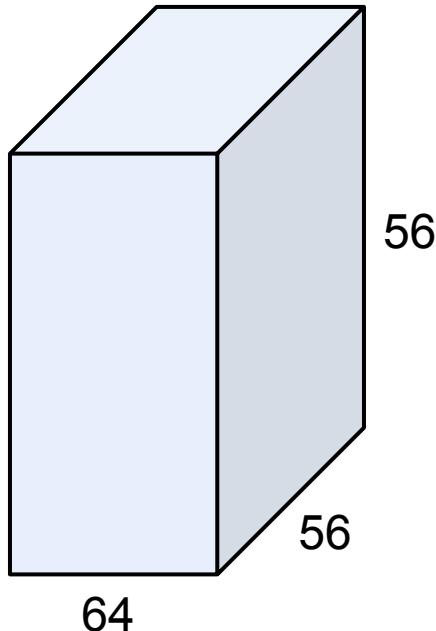
Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
 - Number of filters K ,
 - their spatial extent F ,
 - the stride S ,
 - the amount of zero padding P .
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
 - $W_2 = (W_1 - F + 2P)/S + 1$
 - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
 - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d -th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d -th filter over the input volume with a stride of S , and then offset by d -th bias.

$$K = (\text{powers of 2, e.g. } 32, 64, 128, 512)$$

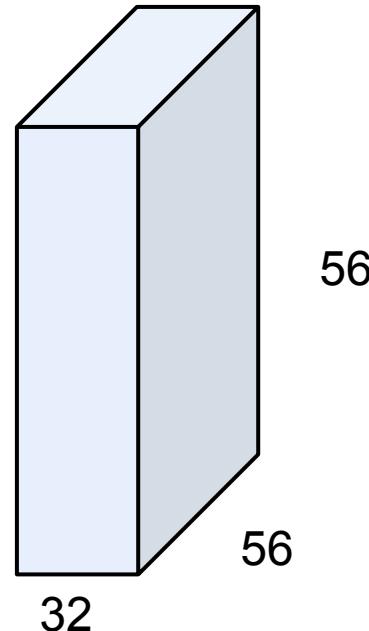
- $F = 3, S = 1, P = 1$
- $F = 5, S = 1, P = 2$
- $F = 5, S = 2, P = ?$ (whatever fits)
- $F = 1, S = 1, P = 0$

(btw, 1x1 convolution layers make perfect sense)



1x1 CONV
with 32 filters

(each filter has size
1x1x64, and performs a
64-dimensional dot
product)



Example: CONV layer in Torch

SpatialConvolution

```
module = nn.SpatialConvolution(nInputPlane, nOutputPlane, kW, KH, [dW], [dH], [padW], [padH])
```

Applies a 2D convolution over an input image composed of several input planes. The `input` tensor in `forward(input)` is expected to be a 3D tensor (`nInputPlane x height x width`).

The parameters are the following:

- `nInputPlane` : The number of expected input planes in the image given into `forward()`.
- `nOutputPlane` : The number of output planes the convolution layer will produce.
- `kW` : The kernel width of the convolution
- `KH` : The kernel height of the convolution
- `dW` : The step of the convolution in the width dimension. Default is `1`.
- `dH` : The step of the convolution in the height dimension. Default is `1`.
- `padW` : The additional zeros added per width to the input planes. Default is `0`, a good number is `(kW-1)/2`.
- `padH` : The additional zeros added per height to the input planes. Default is `padW`, a good number is `(KH-1)/2`.

Note that depending of the size of your kernel, several (of the last) columns or rows of the input image might be lost. It is up to the user to add proper padding in images.

If the input image is a 3D tensor `nInputPlane x height x width`, the output image size will be `nOutputPlane x oheight x owidth` where

```
owidth = floor((width + 2*padW - kW) / dW + 1)
oheight = floor((height + 2*padH - KH) / dH + 1)
```

[Torch](#) is licensed under [BSD 3-clause](#).

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
 - Number of filters K ,
 - their spatial extent F ,
 - the stride S ,
 - the amount of zero padding P .

Example: CONV layer in Caffe

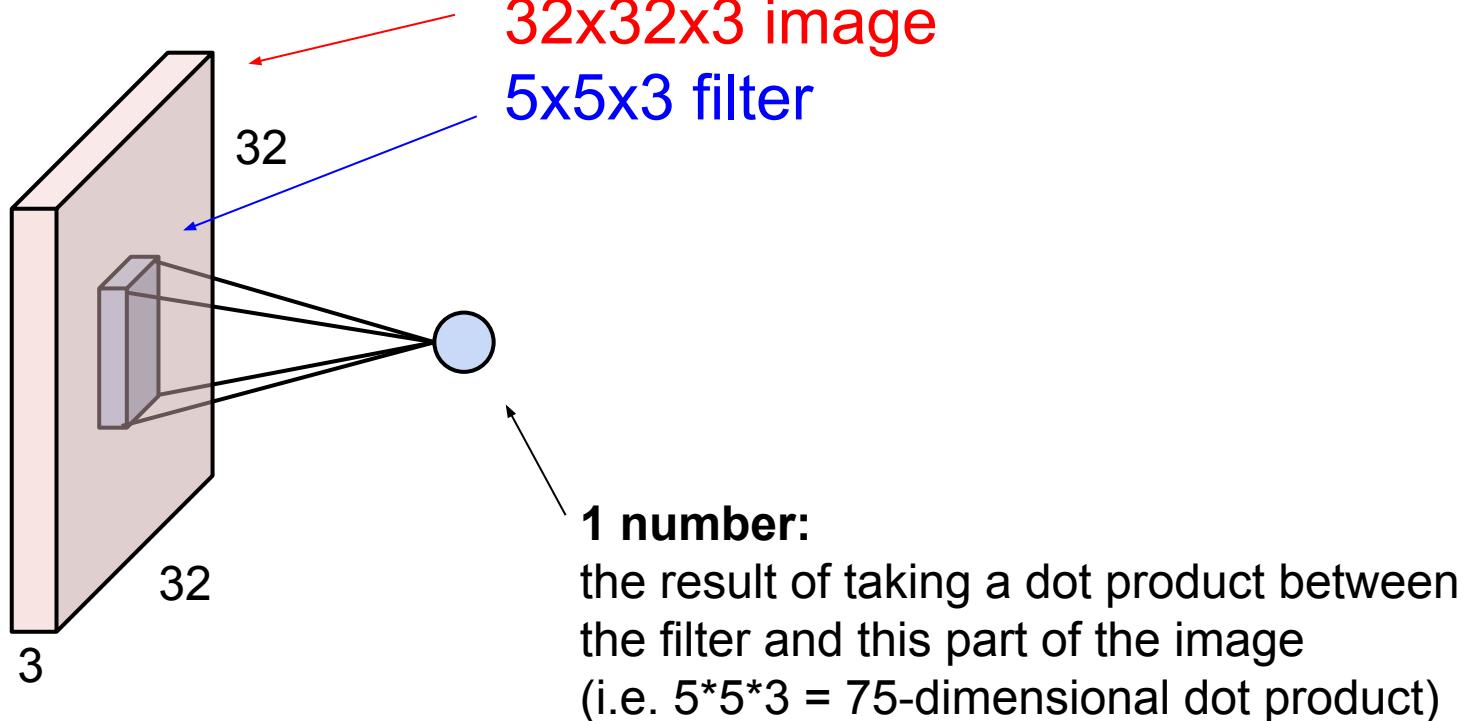
```
layer {
    name: "convl"
    type: "Convolution"
    bottom: "data"
    top: "convl"
    # learning rate and decay multipliers for the filters
    param { lr_mult: 1 decay_mult: 1 }
    # learning rate and decay multipliers for the biases
    param { lr_mult: 2 decay_mult: 0 }
    convolution_param {
        num_output: 96      # learn 96 filters
        kernel_size: 11     # each filter is 11x11
        stride: 4           # step 4 pixels between each filter application
        weight_filler {
            type: "gaussian" # initialize the filters from a Gaussian
            std: 0.01          # distribution with stdev 0.01 (default mean: 0)
        }
        bias_filler {
            type: "constant" # initialize the biases to zero (0)
            value: 0
        }
    }
}
```

Summary. To summarize, the Conv Layer:

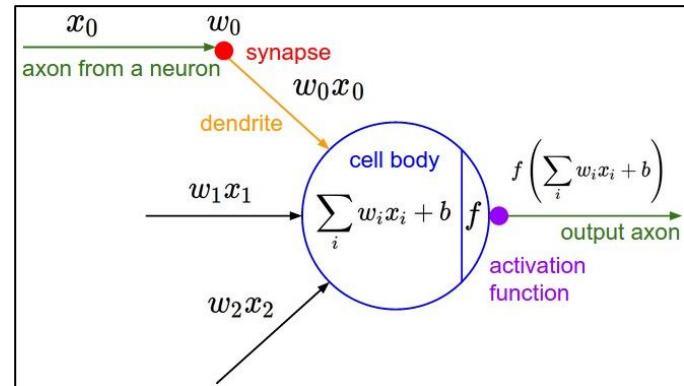
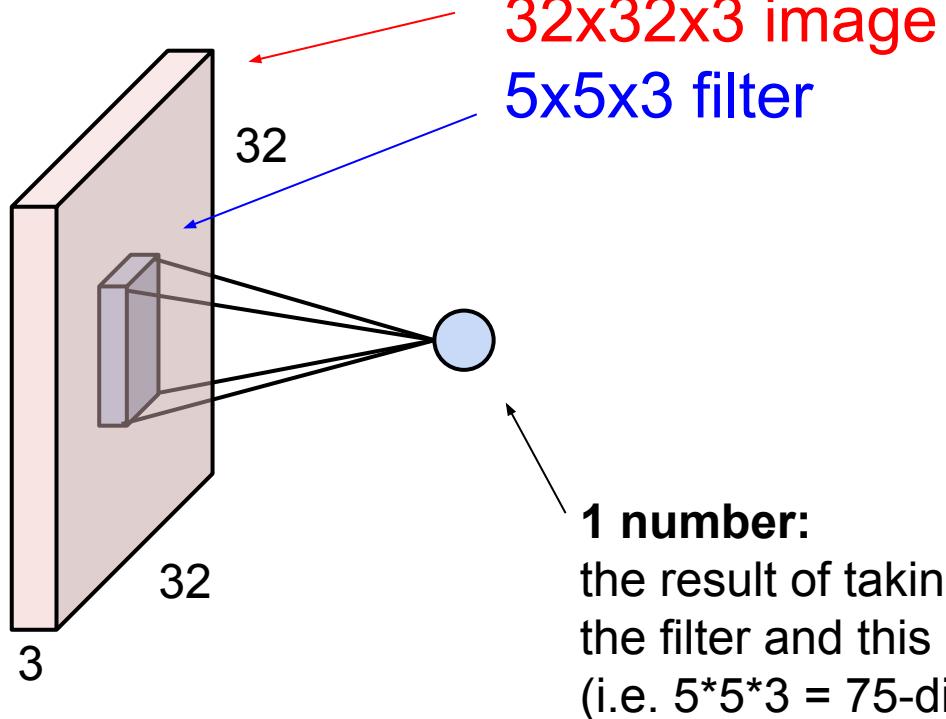
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
 - Number of filters K ,
 - their spatial extent F ,
 - the stride S ,
 - the amount of zero padding P .

[Caffe](#) is licensed under [BSD 2-Clause](#).

The brain/neuron view of CONV Layer

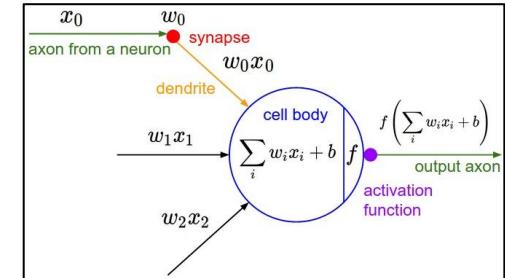
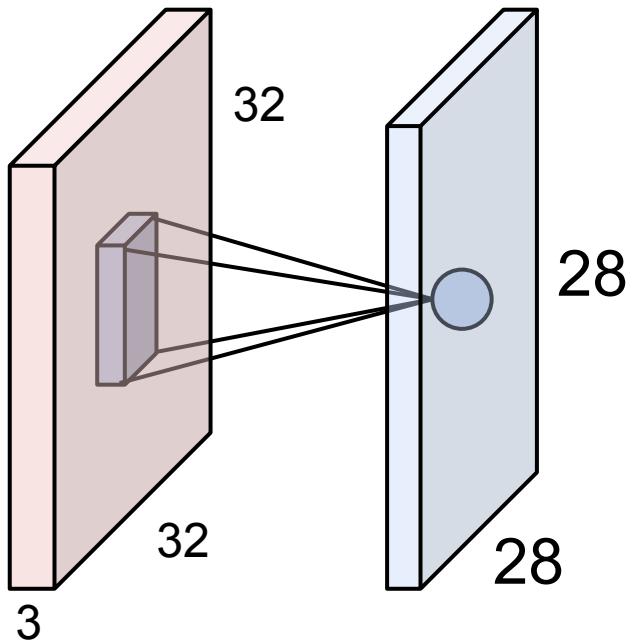


The brain/neuron view of CONV Layer



It's just a neuron with local connectivity...

The brain/neuron view of CONV Layer

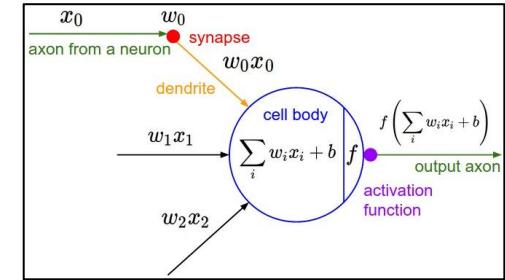
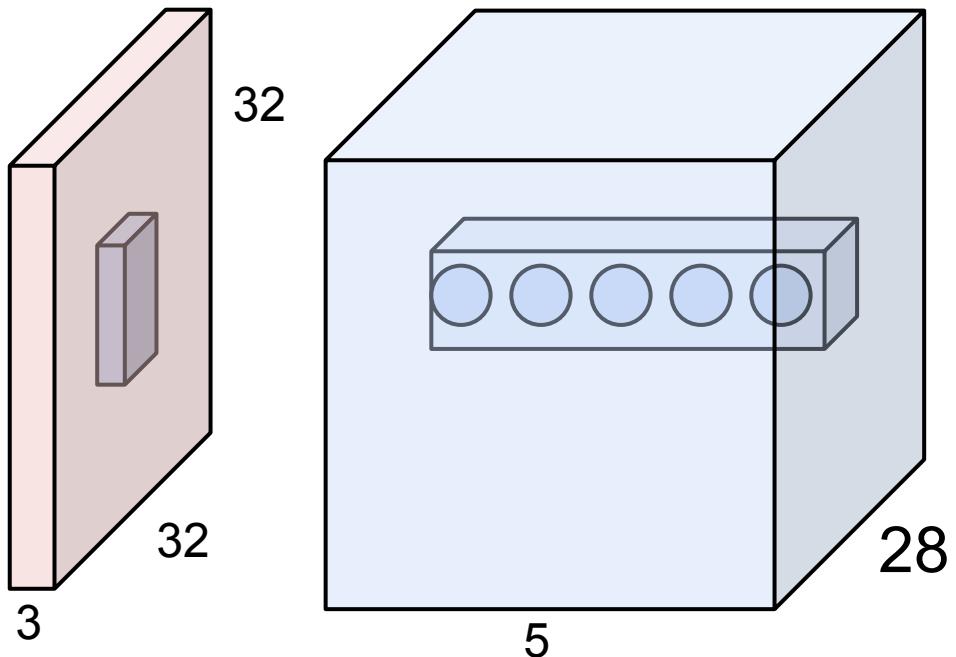


An activation map is a 28x28 sheet of neuron outputs:

1. Each is connected to a small region in the input
2. All of them share parameters

“5x5 filter” -> “5x5 receptive field for each neuron”

The brain/neuron view of CONV Layer



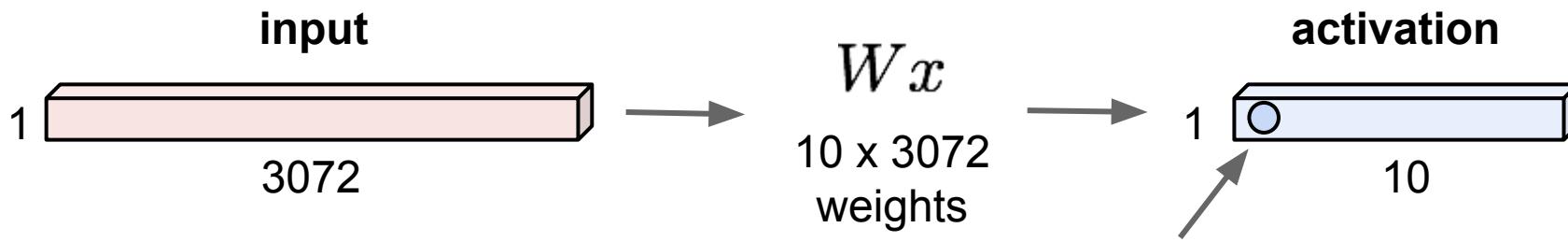
E.g. with 5 filters,
CONV layer consists of
neurons arranged in a 3D grid
($28 \times 28 \times 5$)

There will be 5 different
neurons all looking at the same
region in the input volume

Reminder: Fully Connected Layer

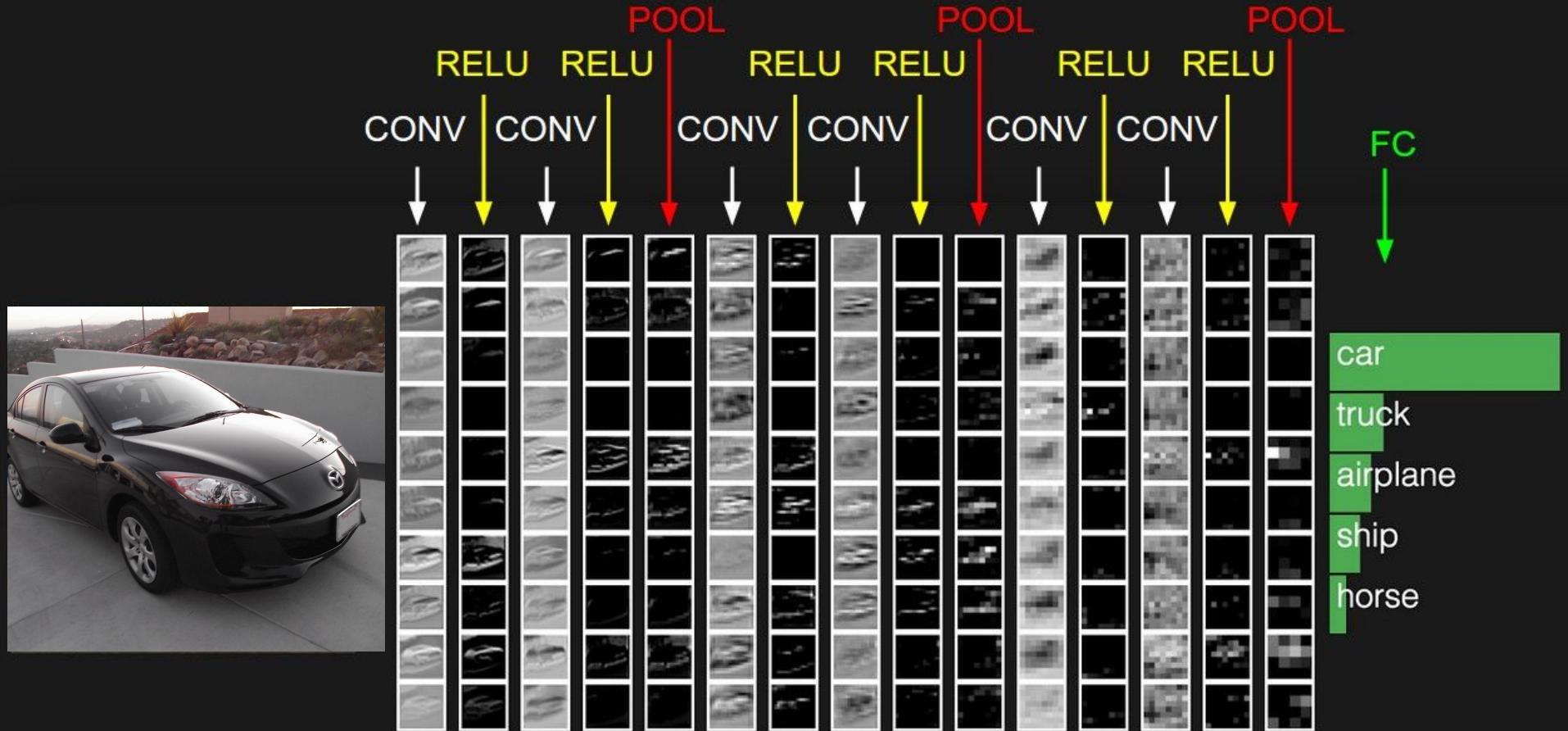
32x32x3 image -> stretch to 3072 x 1

Each neuron
looks at the full
input volume



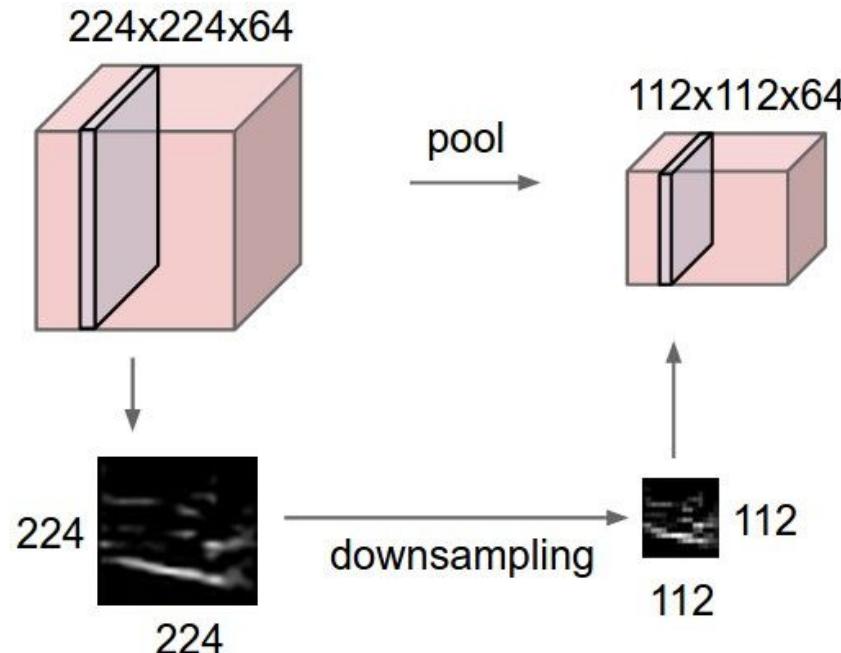
1 number:
the result of taking a dot product
between a row of W and the input
(a 3072-dimensional dot product)

two more layers to go: POOL/FC

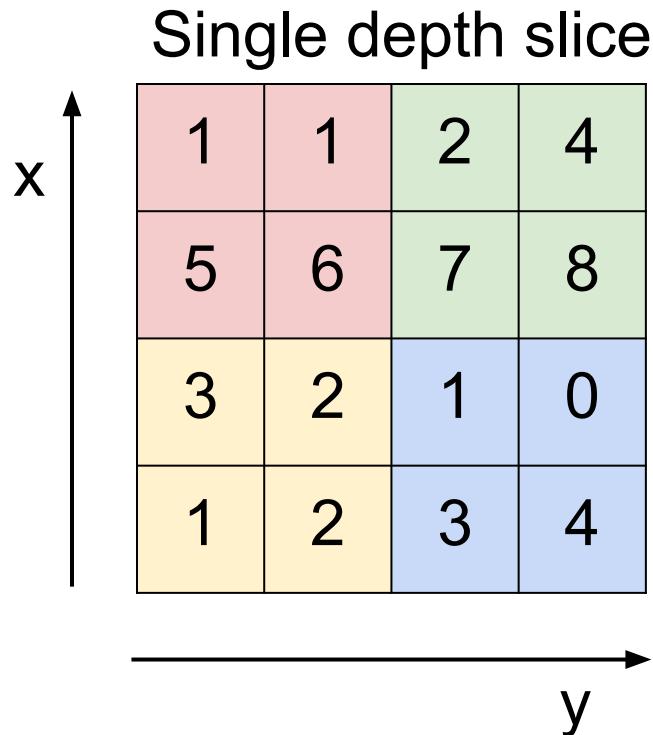


Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently:



MAX POOLING



- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires three hyperparameters:
 - their spatial extent F ,
 - the stride S ,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
 - $W_2 = (W_1 - F)/S + 1$
 - $H_2 = (H_1 - F)/S + 1$
 - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers

Common settings:

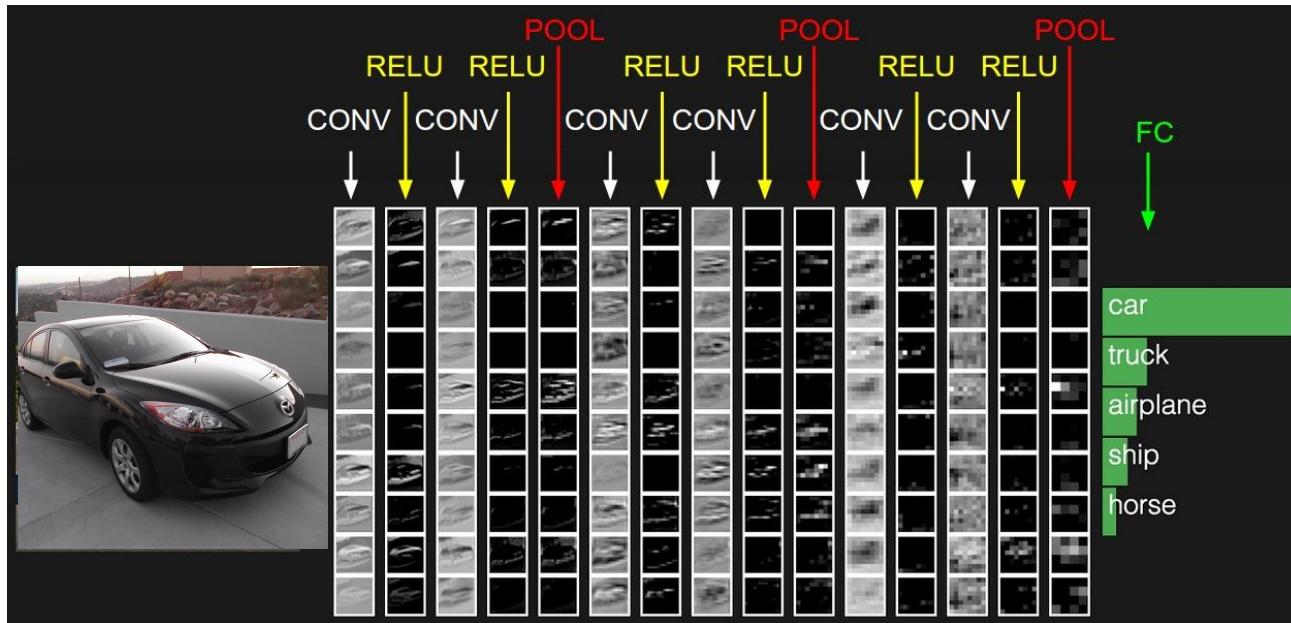
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires three hyperparameters:
 - their spatial extent F ,
 - the stride S ,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
 - $W_2 = (W_1 - F)/S + 1$
 - $H_2 = (H_1 - F)/S + 1$
 - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers

$F = 2, S = 2$

$F = 3, S = 2$

Fully Connected Layer (FC layer)

- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks



[ConvNetJS demo: training on CIFAR-10]

ConvNetJS CIFAR-10 demo

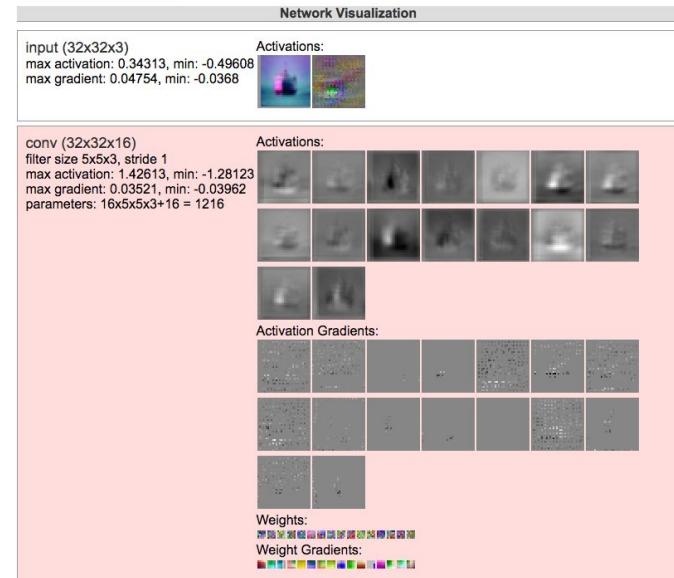
Description

This demo trains a Convolutional Neural Network on the [CIFAR-10 dataset](#) in your browser, with nothing but Javascript. The state of the art on this dataset is about 90% accuracy and human performance is at about 94% (not perfect as the dataset can be a bit ambiguous). I used [this python script](#) to parse the [original files](#) (python version) into batches of images that can be easily loaded into page DOM with img tags.

This dataset is more difficult and it takes longer to train a network. Data augmentation includes random flipping and random image shifts by up to 2px horizontally and vertically.

By default, in this demo we're using Adadelta which is one of per-parameter adaptive step size methods, so we don't have to worry about changing learning rates or momentum over time. However, I still included the text fields for changing these if you'd like to play around with SGD+Momentum trainer.

Report questions/bugs/suggestions to [@karpathy](#).



<http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html>

Summary

- ConvNets stack CONV,POOL,FC layers
- Trend towards smaller filters and deeper architectures
- Trend towards getting rid of POOL/FC layers (just CONV)
- Typical architectures look like
 $[(CONV-RELU)^*N-POOL?]^*M-(FC-RELU)^*K,SOFTMAX$
where N is usually up to ~5, M is large, $0 \leq K \leq 2$.
 - but recent advances such as ResNet/GoogLeNet challenge this paradigm

Lecture 6: Training Neural Networks, Part I

Administrative

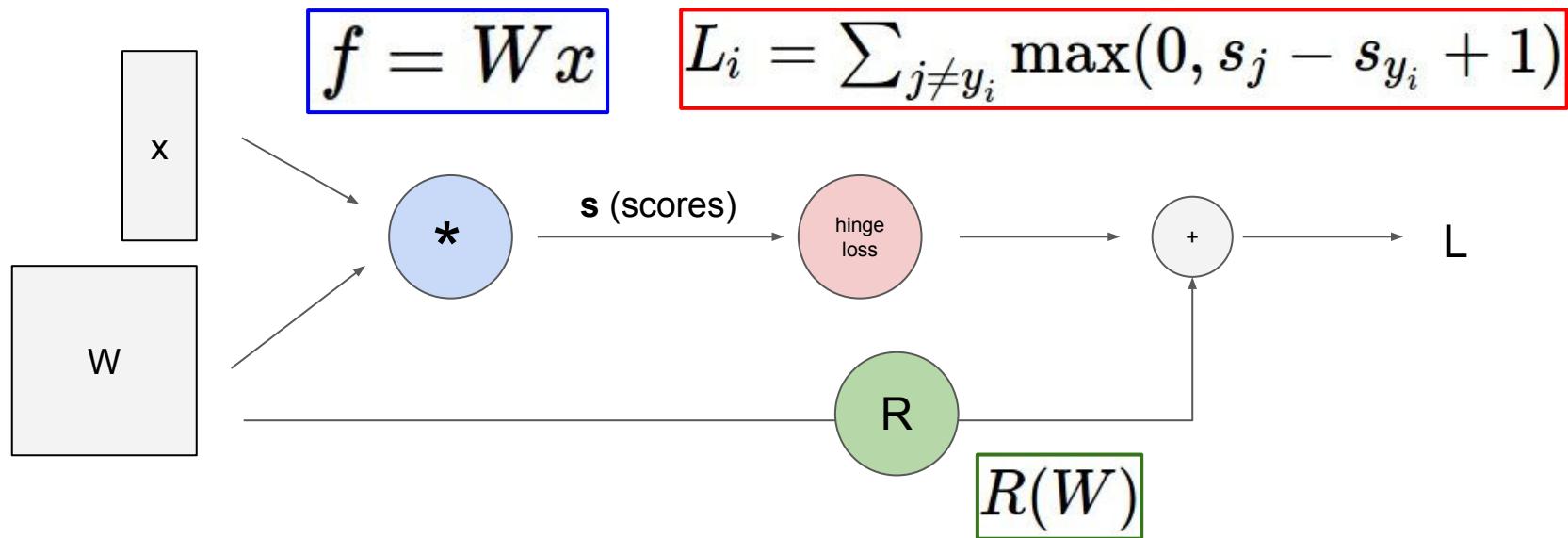
Assignment 1 was due yesterday.

Assignment 2 is out, due Wed May 2. Q5 will be released in a few days, keep an eye out for announcements.

Project proposal due Wed April 25.

Where we are now...

Computational graphs



Where we are now...

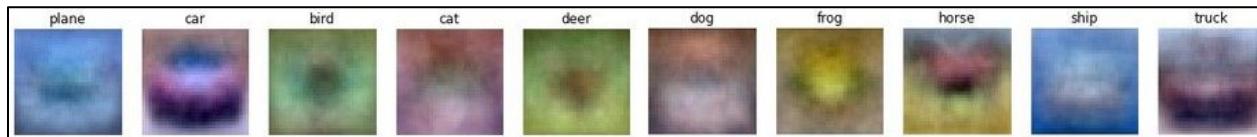
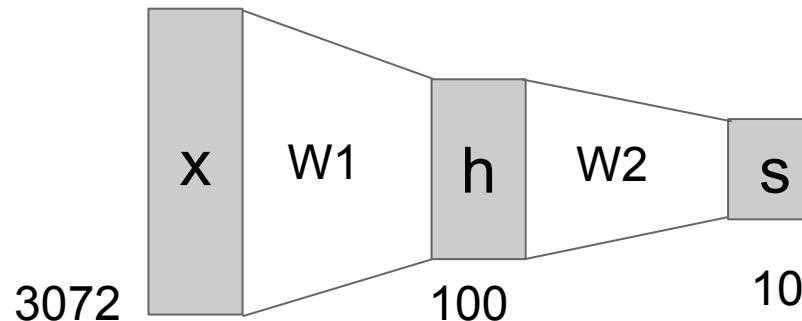
Neural Networks

Linear score function:

$$f = Wx$$

2-layer Neural Network

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



Where we are now...

Convolutional Neural Networks

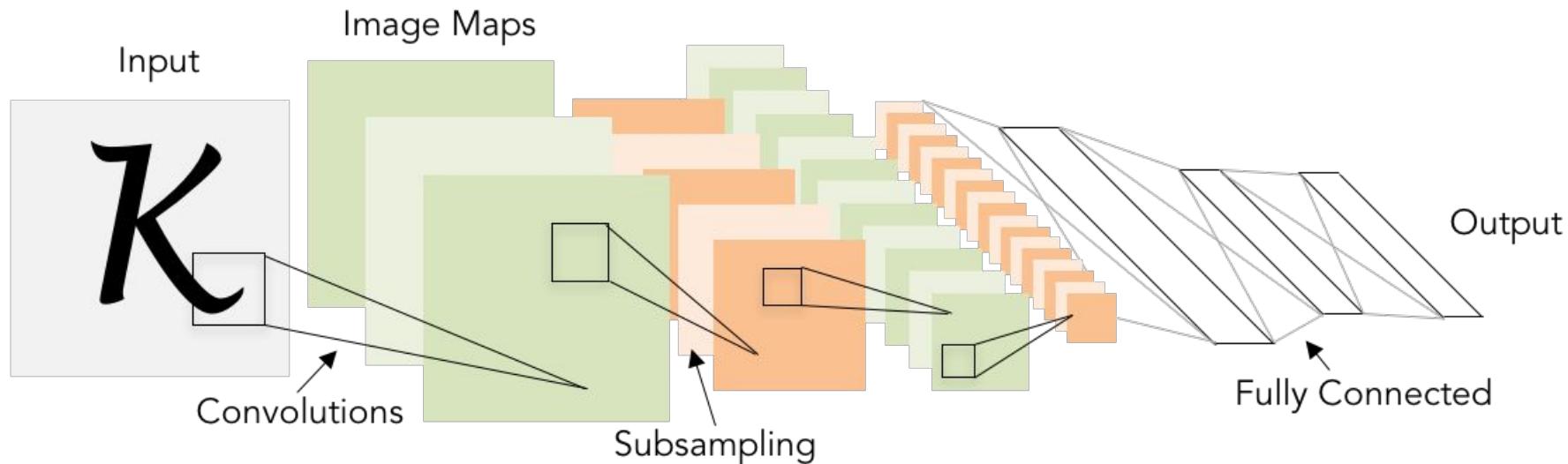
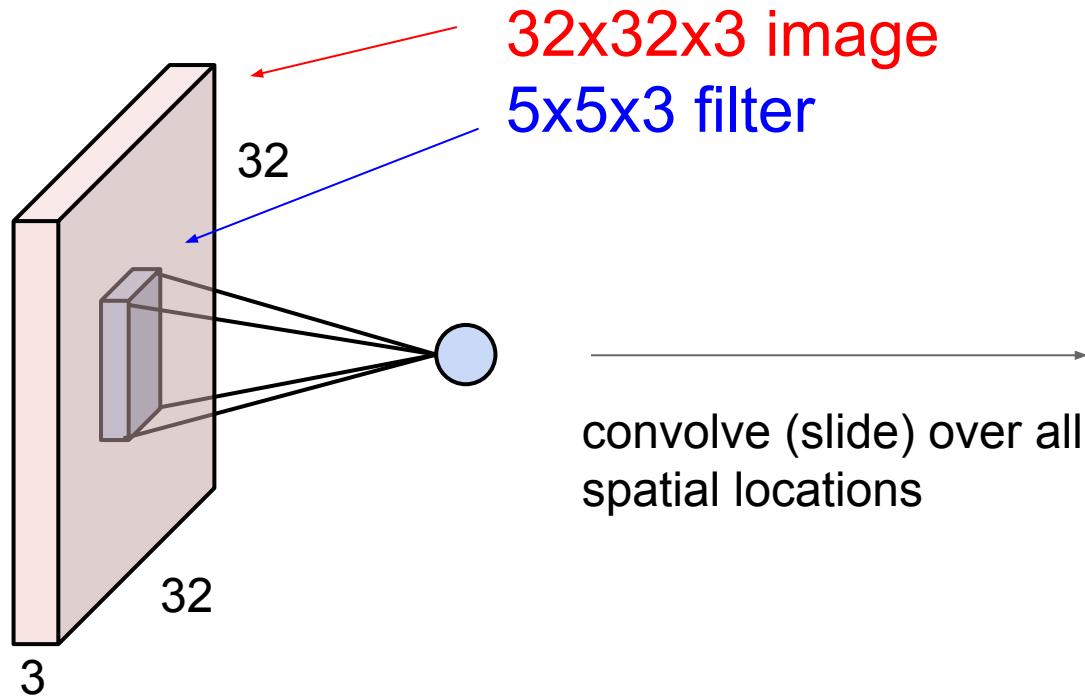


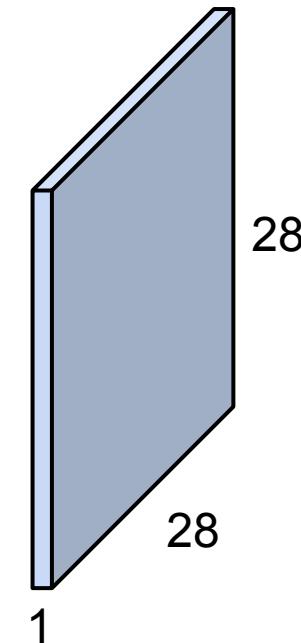
Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

Where we are now...

Convolutional Layer

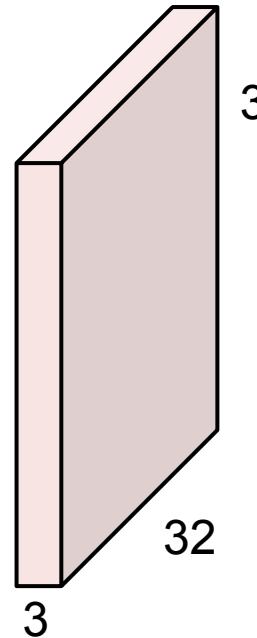


activation map



Where we are now...

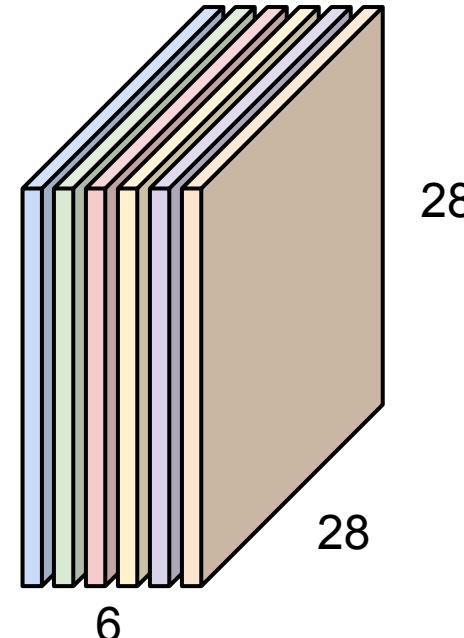
Convolutional Layer



Convolution Layer

For example, if we had 6 5×5 filters, we'll get 6 separate activation maps:

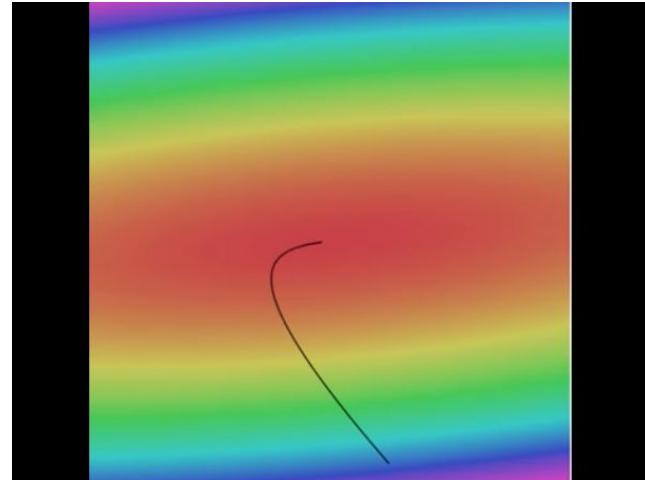
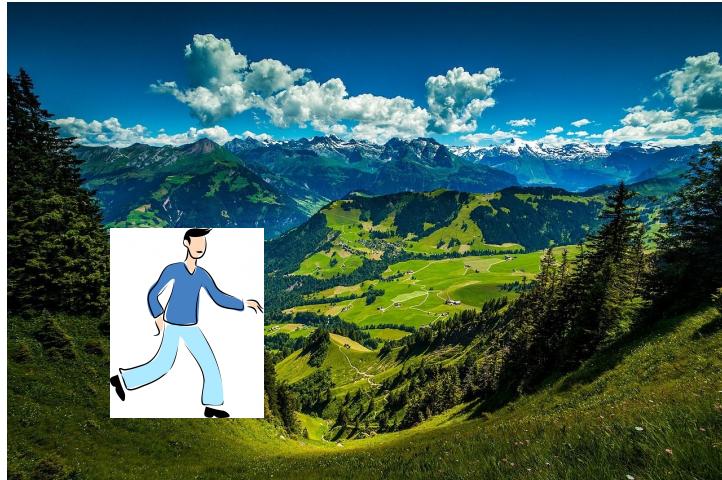
activation maps



We stack these up to get a “new image” of size $28 \times 28 \times 6$!

Where we are now...

Learning network parameters through optimization



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

Landscape image is CC0 1.0 public domain
Walking man image is CC0 1.0 public domain

Where we are now...

Mini-batch SGD

Loop:

1. **Sample** a batch of data
2. **Forward** prop it through the graph
(network), get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient

Next: Training Neural Networks

Overview

1. One time setup

activation functions, preprocessing, weight initialization, regularization, gradient checking

2. Training dynamics

*babysitting the learning process,
parameter updates, hyperparameter optimization*

3. Evaluation

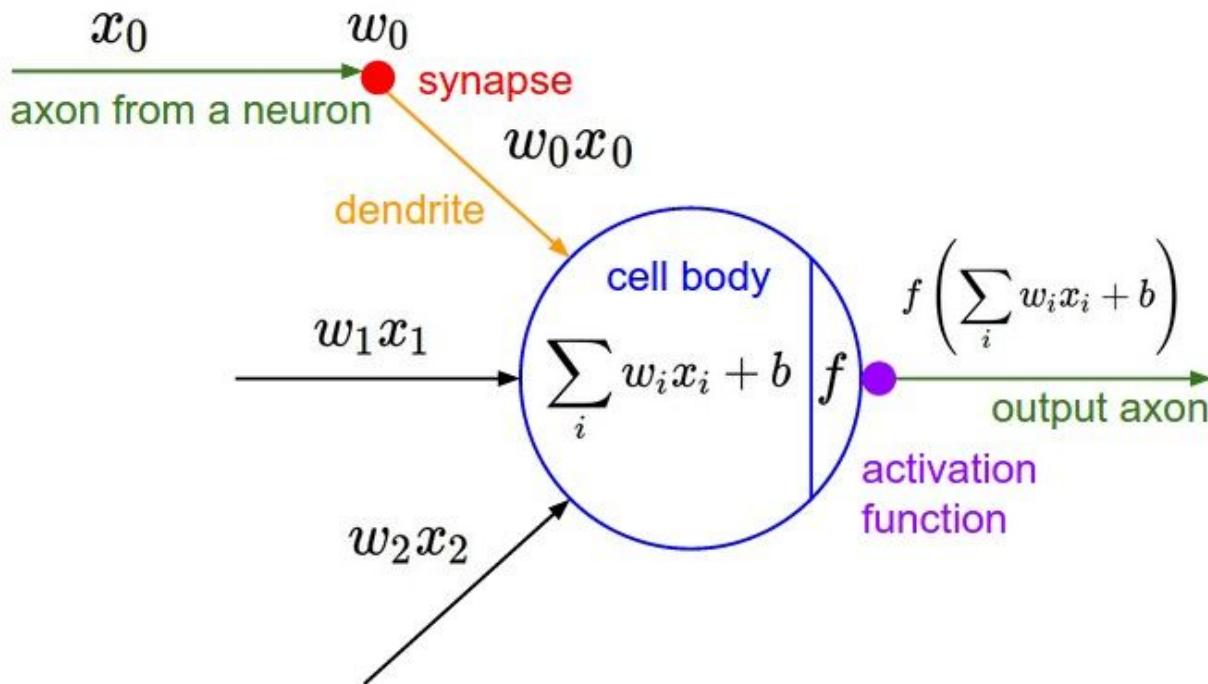
model ensembles

Part 1

- Activation Functions
- Data Preprocessing
- Weight Initialization
- Batch Normalization
- Babysitting the Learning Process
- Hyperparameter Optimization

Activation Functions

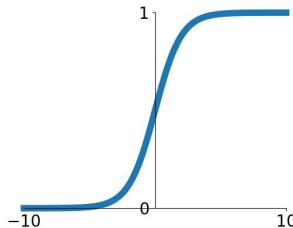
Activation Functions



Activation Functions

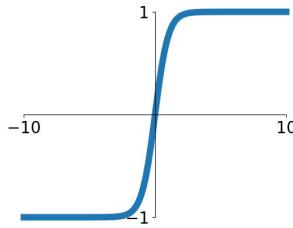
Sigmoid

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



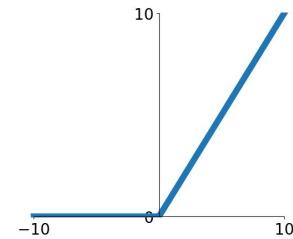
tanh

$$\tanh(x)$$



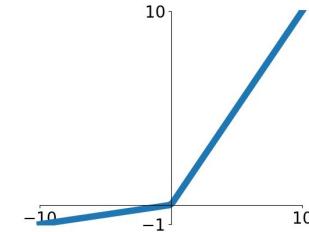
ReLU

$$\max(0, x)$$



Leaky ReLU

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

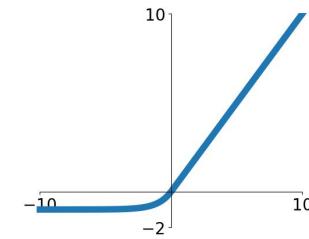


Maxout

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

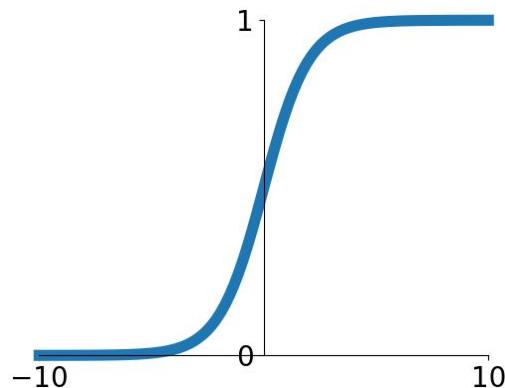
ELU

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



Activation Functions

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



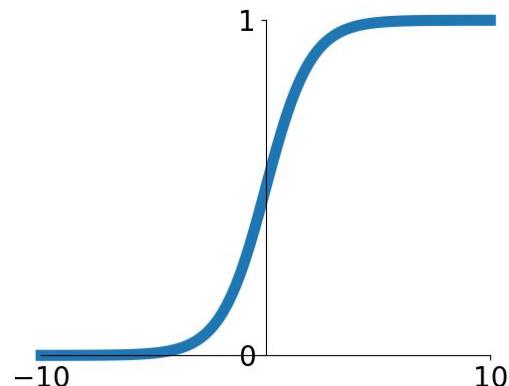
Sigmoid

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

Activation Functions

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

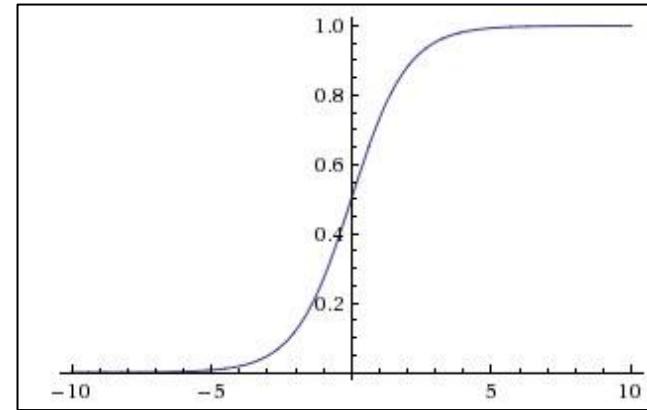
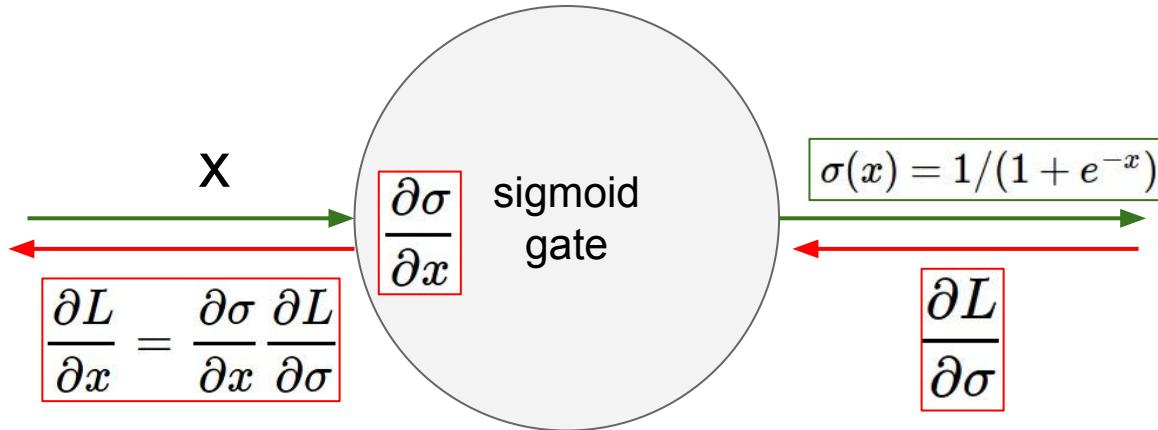
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron



Sigmoid

3 problems:

1. Saturated neurons “kill” the gradients



What happens when $x = -10$?

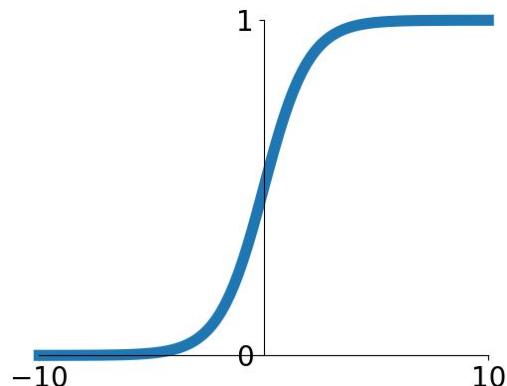
What happens when $x = 0$?

What happens when $x = 10$?

Activation Functions

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

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

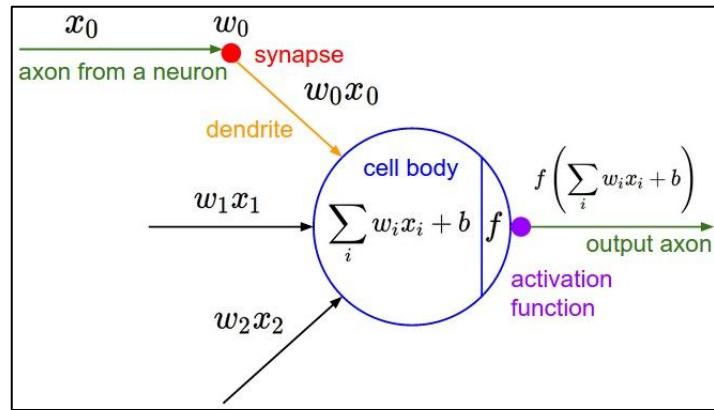


Sigmoid

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered

Consider what happens when the input to a neuron (x) is always positive:

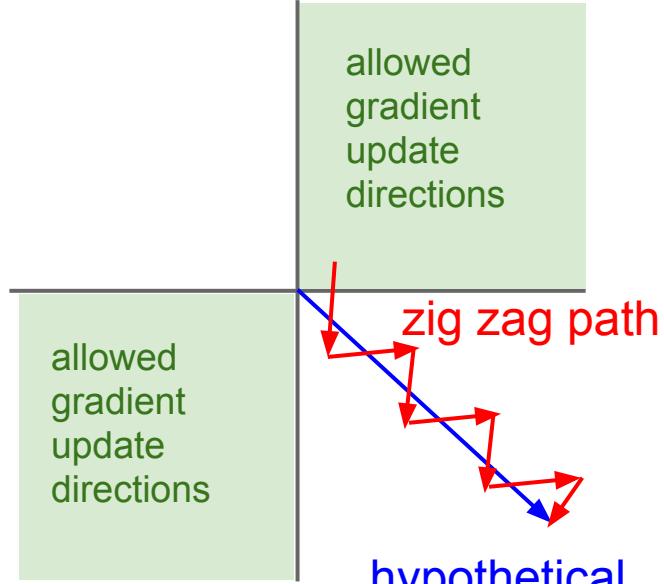


$$f \left(\sum_i w_i x_i + b \right)$$

What can we say about the gradients on w ?

Consider what happens when the input to a neuron is always positive...

$$f \left(\sum_i w_i x_i + b \right)$$



What can we say about the gradients on w ?

Always all positive or all negative :(

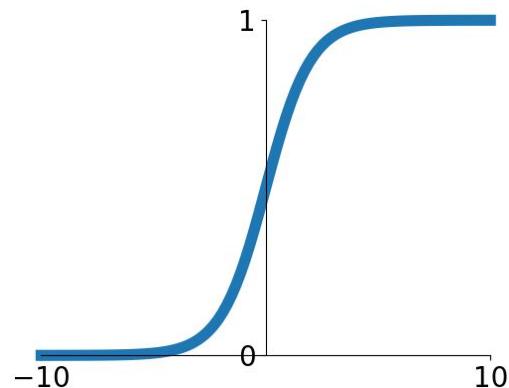
(this is also why you want zero-mean data!)

hypothetical
optimal w
vector

Activation Functions

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

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

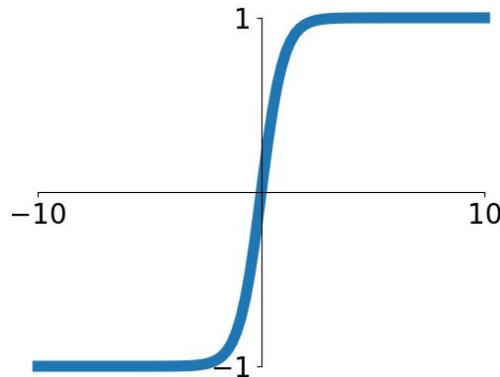


Sigmoid

3 problems:

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

Activation Functions

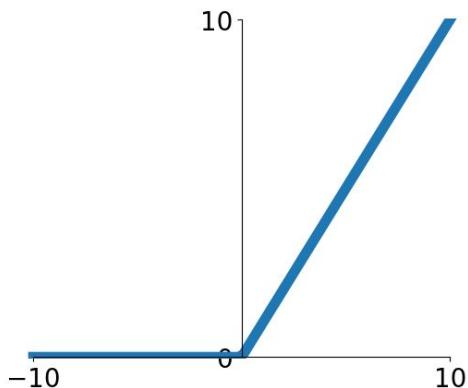


tanh(x)

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

[LeCun et al., 1991]

Activation Functions

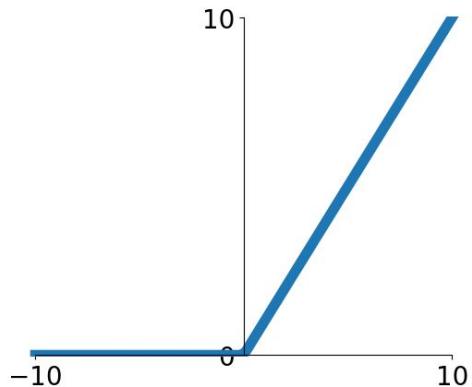


- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

ReLU
(Rectified Linear Unit)

[Krizhevsky et al., 2012]

Activation Functions

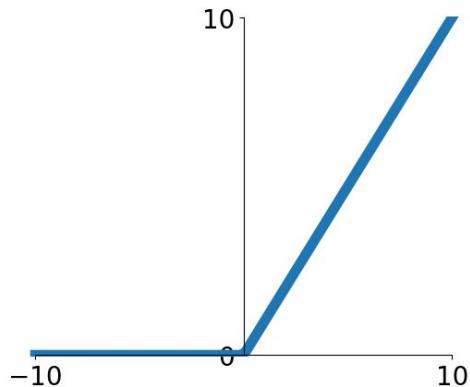


ReLU
(Rectified Linear Unit)

- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

- Not zero-centered output

Activation Functions

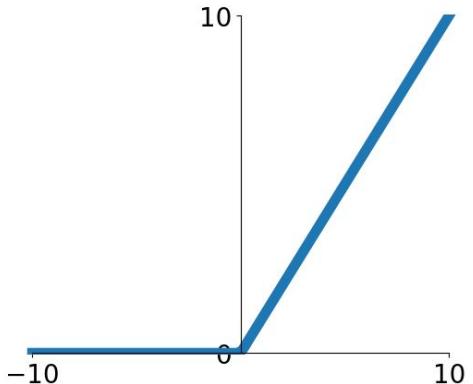
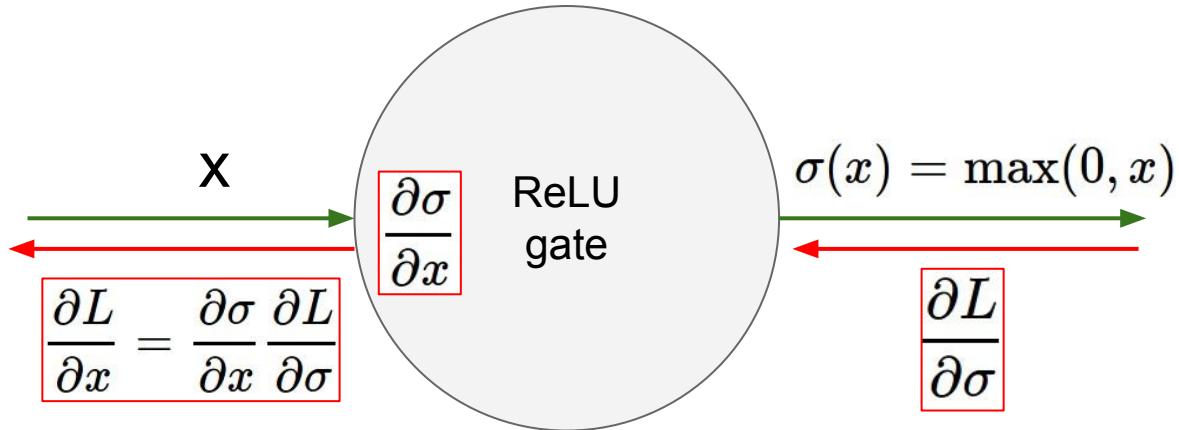


ReLU
(Rectified Linear Unit)

- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

- Not zero-centered output
- An annoyance:

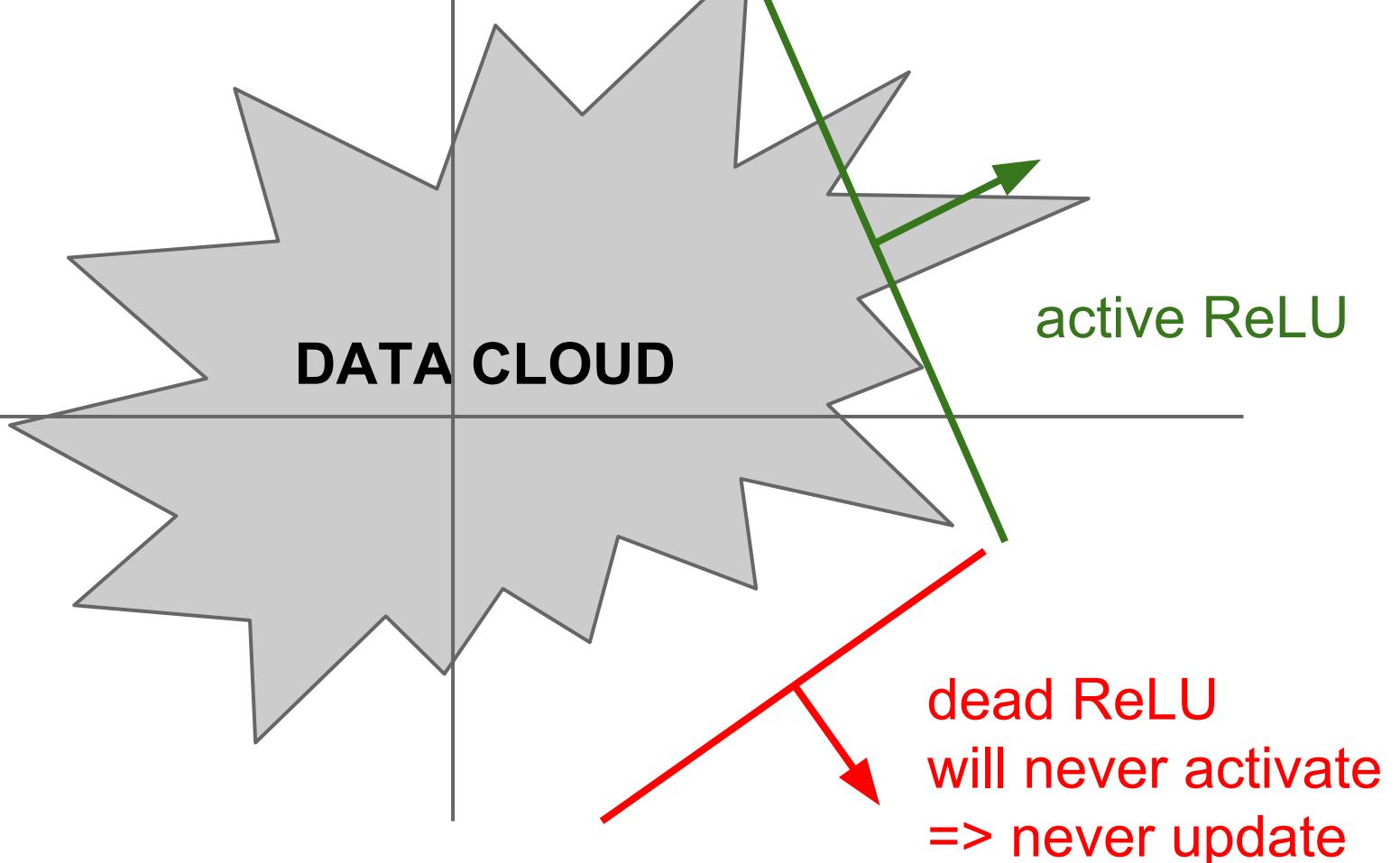
hint: what is the gradient when $x < 0$?

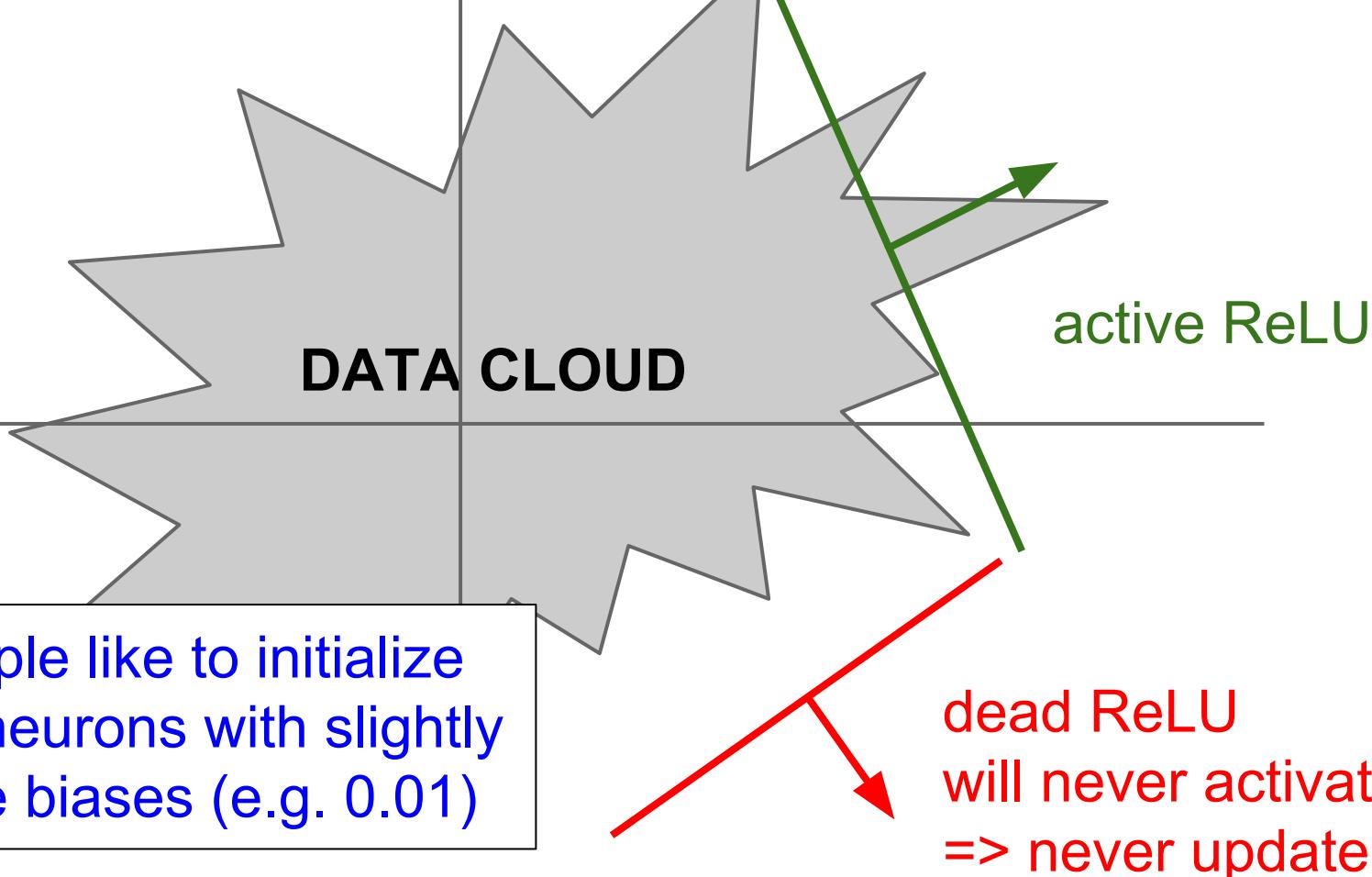


What happens when $x = -10$?

What happens when $x = 0$?

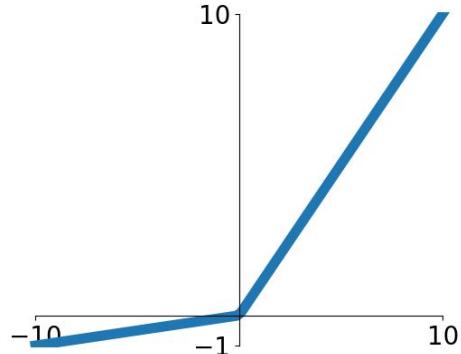
What happens when $x = 10$?





Activation Functions

[Mass et al., 2013]
[He et al., 2015]



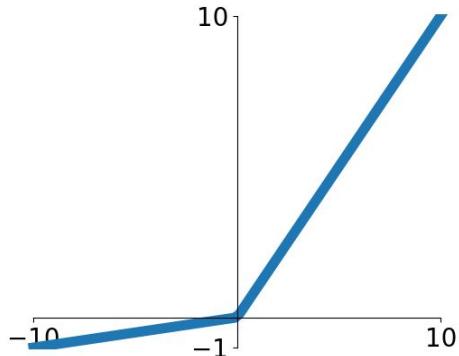
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Activation Functions

[Mass et al., 2013]
[He et al., 2015]



Leaky ReLU

$$f(x) = \max(0.01x, x)$$

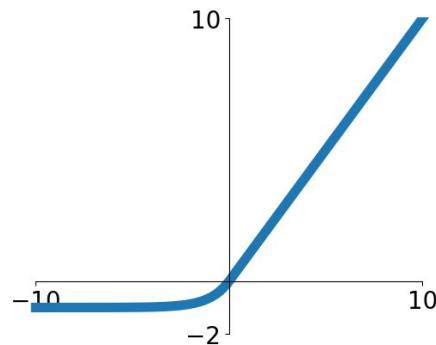
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

Parametric Rectifier (PReLU)

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

backprop into α
(parameter)

Exponential Linear Units (ELU)



- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

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

- Computation requires $\exp()$

Maxout “Neuron”

[Goodfellow et al., 2013]

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

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

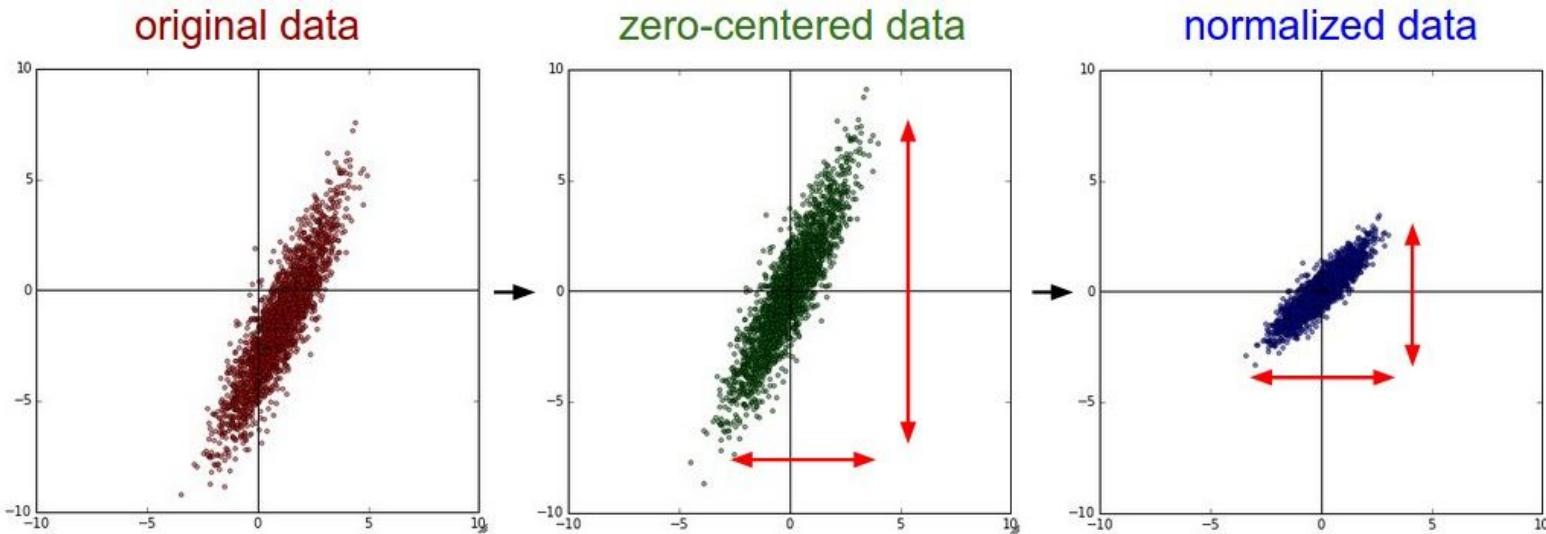
Problem: doubles the number of parameters/neuron :(

TLDR: In practice:

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don't expect much
- Don't use sigmoid

Data Preprocessing

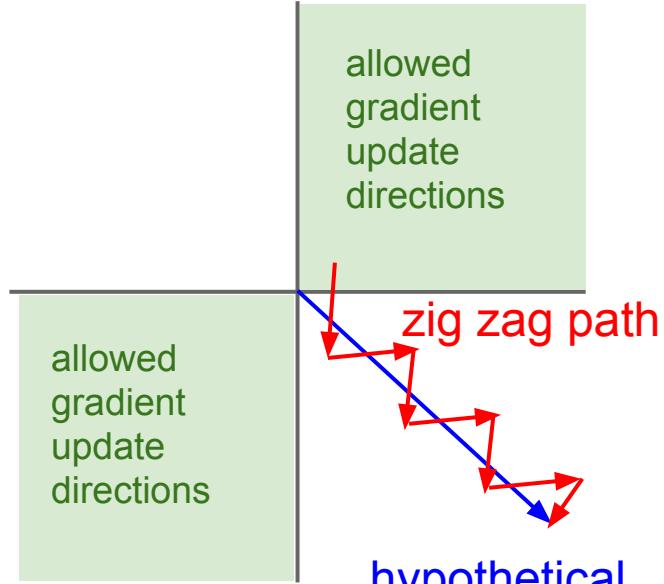
Step 1: Preprocess the data



(Assume X [NxD] is data matrix,
each example in a row)

Remember: Consider what happens when the input to a neuron is always positive...

$$f \left(\sum_i w_i x_i + b \right)$$

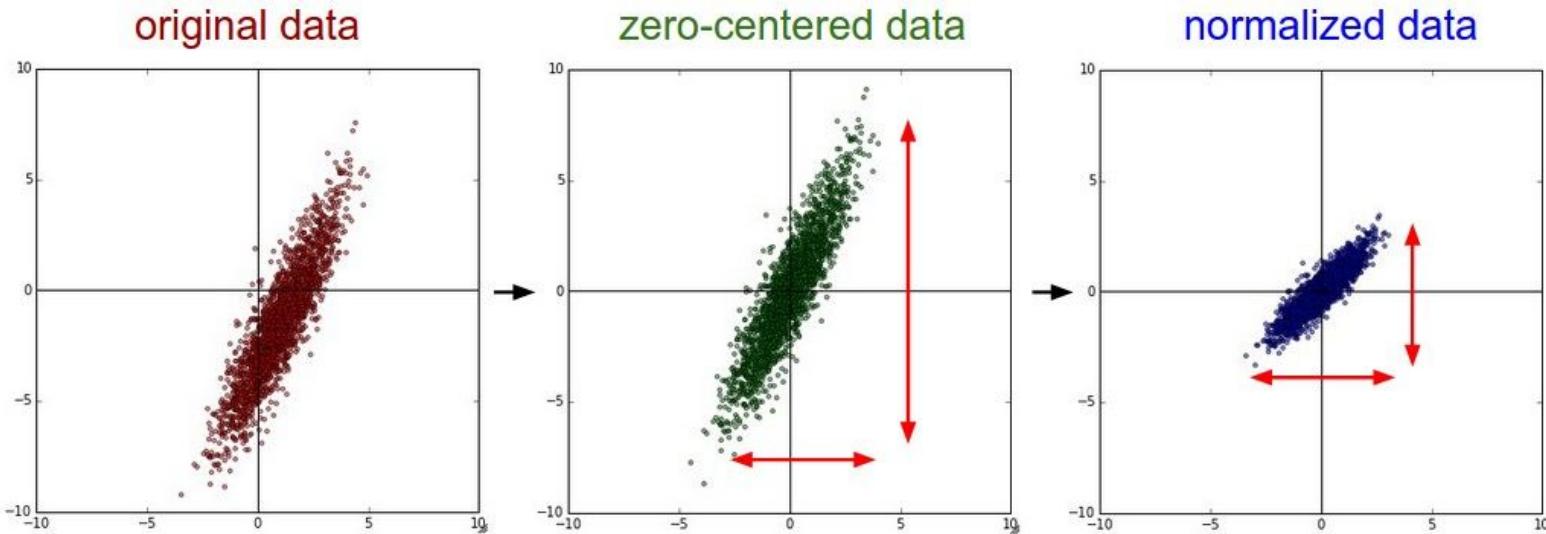


What can we say about the gradients on w ?

Always all positive or all negative :(

(this is also why you want zero-mean data!)

Step 1: Preprocess the data



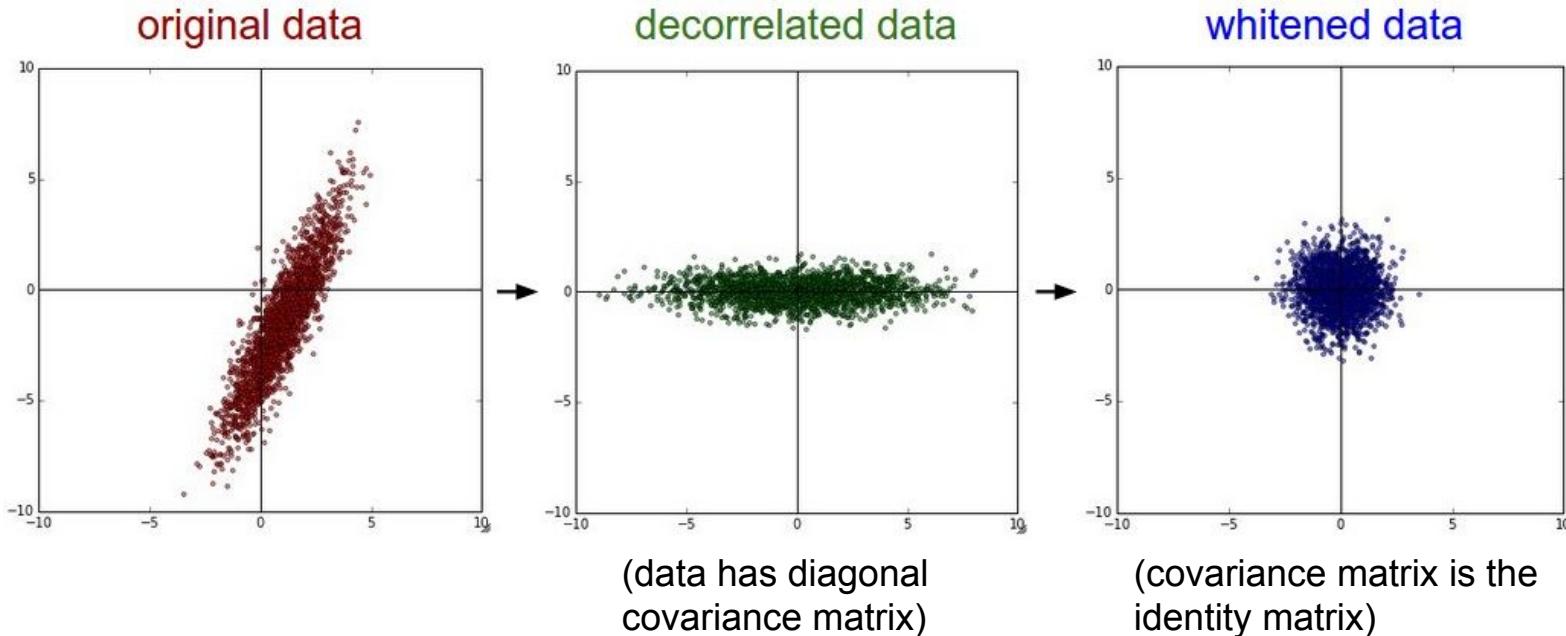
```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

(Assume X [NxD] is data matrix,
each example in a row)

Step 1: Preprocess the data

In practice, you may also see **PCA** and **Whitening** of the data



TLDR: In practice for Images: center only

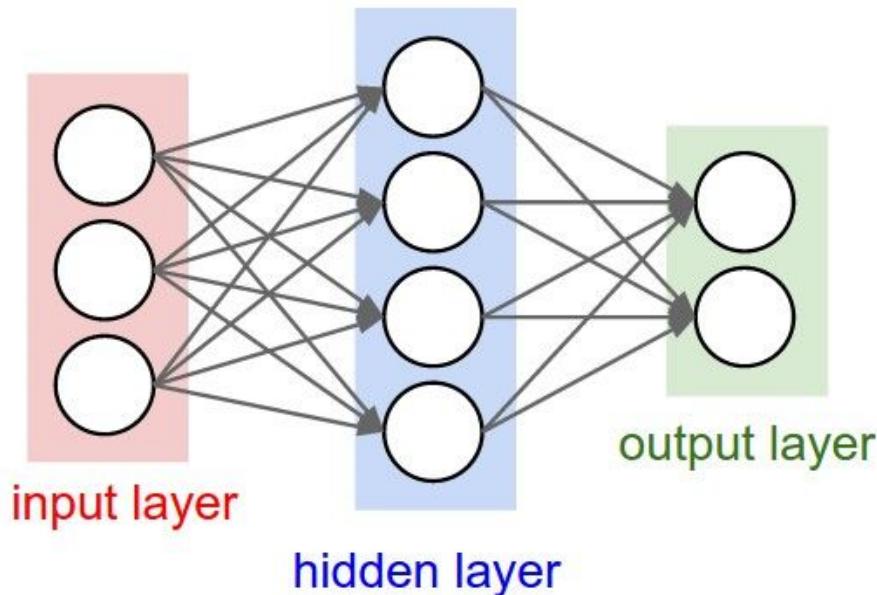
e.g. consider CIFAR-10 example with [32,32,3] images

- 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)

Not common to normalize variance, to do PCA or whitening

Weight Initialization

- Q: what happens when $W=\text{constant init}$ is used?



- First idea: **Small random numbers**
(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

- First idea: **Small random numbers**
(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Works ~okay for small networks, but problems with deeper networks.

Lets look at some activation statistics

E.g. 10-layer net with 500 neurons on each layer, using tanh non-linearities, and initializing as described in last slide.

```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)

act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

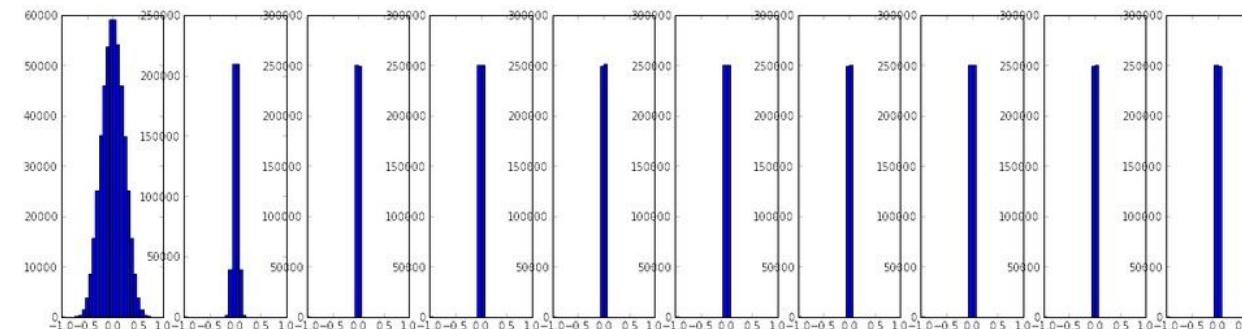
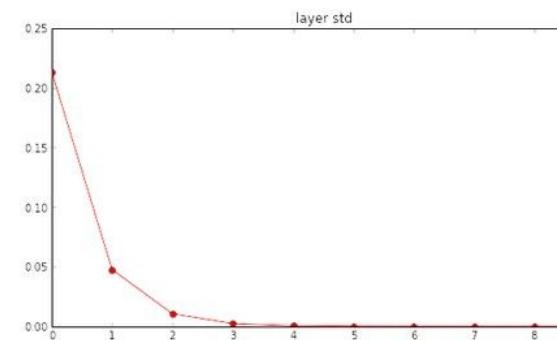
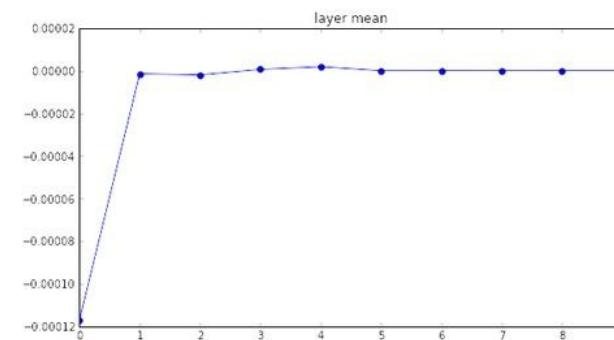
    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer

# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.iteritems()]
layer_stds = [np.std(H) for i,H in Hs.iteritems()]
for i,H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i])

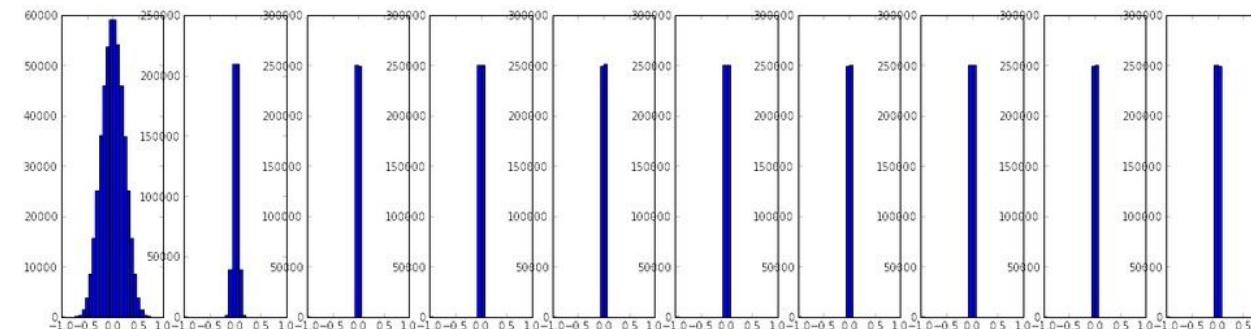
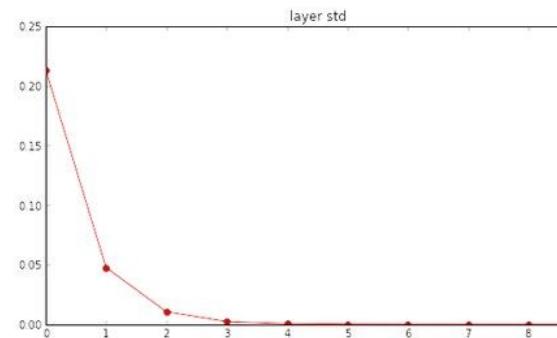
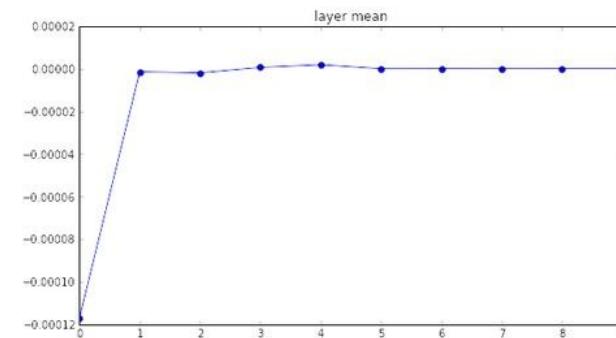
# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i,H in Hs.iteritems():
    plt.subplot(1,len(Hs),i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```

```
input layer had mean 0.000927 and std 0.998388  
hidden layer 1 had mean -0.000117 and std 0.213081  
hidden layer 2 had mean -0.000001 and std 0.047551  
hidden layer 3 had mean -0.000002 and std 0.010630  
hidden layer 4 had mean 0.000001 and std 0.002378  
hidden layer 5 had mean 0.000002 and std 0.000532  
hidden layer 6 had mean -0.000000 and std 0.000119  
hidden layer 7 had mean 0.000000 and std 0.000026  
hidden layer 8 had mean -0.000000 and std 0.000006  
hidden layer 9 had mean 0.000000 and std 0.000001  
hidden layer 10 had mean -0.000000 and std 0.000000
```



```
input layer had mean 0.000927 and std 0.998388  
hidden layer 1 had mean -0.000117 and std 0.213081  
hidden layer 2 had mean -0.000001 and std 0.047551  
hidden layer 3 had mean -0.000002 and std 0.010630  
hidden layer 4 had mean 0.000001 and std 0.002378  
hidden layer 5 had mean 0.000002 and std 0.000532  
hidden layer 6 had mean -0.000000 and std 0.000119  
hidden layer 7 had mean 0.000000 and std 0.000026  
hidden layer 8 had mean -0.000000 and std 0.000006  
hidden layer 9 had mean 0.000000 and std 0.000001  
hidden layer 10 had mean -0.000000 and std 0.000000
```



All activations become zero!

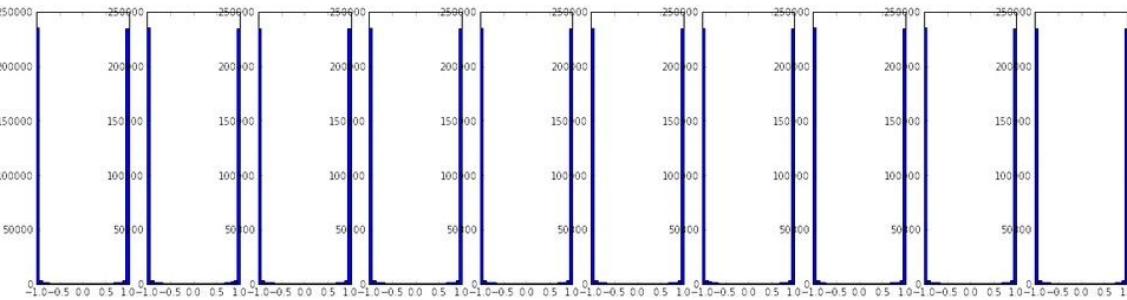
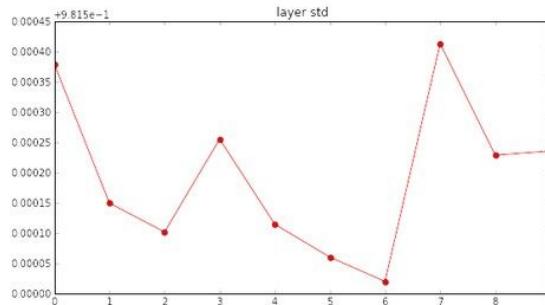
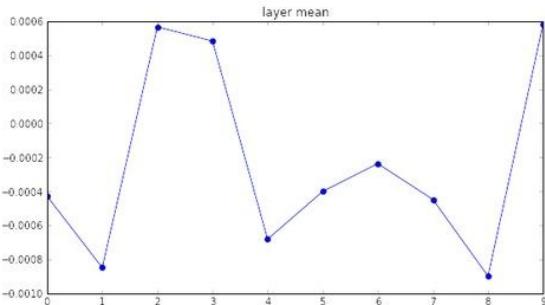
Q: think about the backward pass.
What do the gradients look like?

Hint: think about backward pass for a W^*X gate.

```
w = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

```
input layer had mean 0.001800 and std 1.001311  
hidden layer 1 had mean -0.000430 and std 0.981879  
hidden layer 2 had mean -0.000849 and std 0.981649  
hidden layer 3 had mean 0.000566 and std 0.981601  
hidden layer 4 had mean 0.000483 and std 0.981755  
hidden layer 5 had mean -0.000682 and std 0.981614  
hidden layer 6 had mean -0.000401 and std 0.981560  
hidden layer 7 had mean -0.000237 and std 0.981520  
hidden layer 8 had mean -0.000448 and std 0.981913  
hidden layer 9 had mean -0.000899 and std 0.981728  
hidden layer 10 had mean 0.000584 and std 0.981736
```

*1.0 instead of *0.01

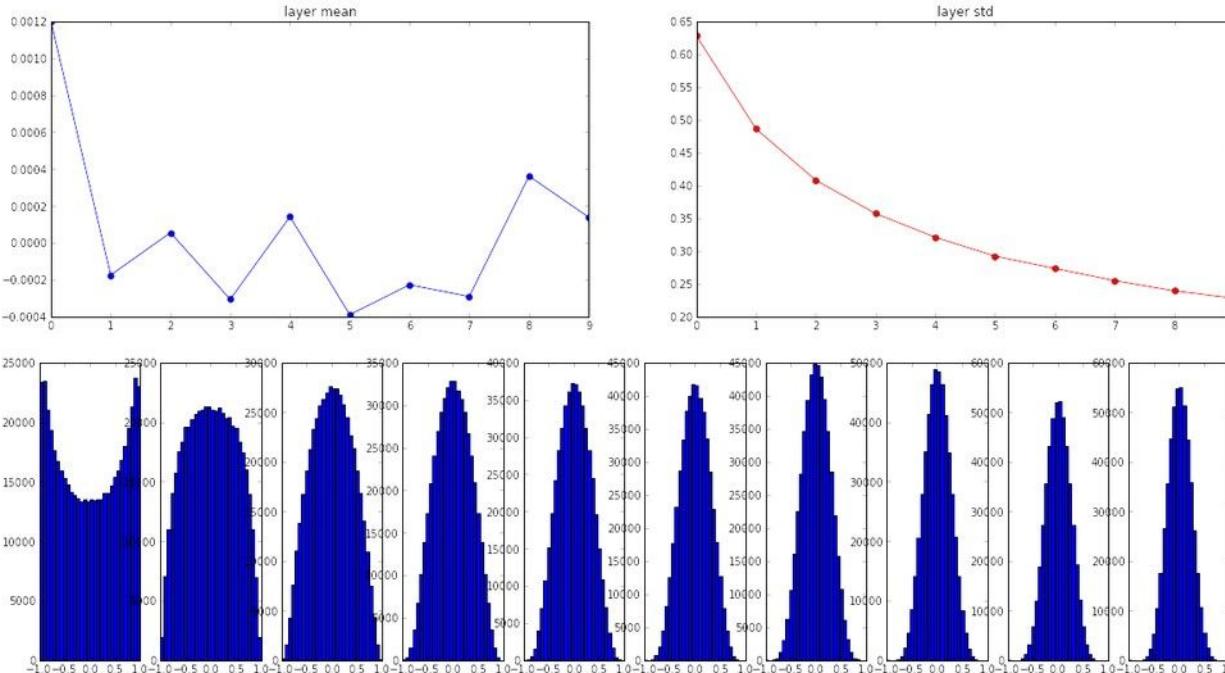


Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

```
input layer had mean 0.001800 and std 1.001311  
hidden layer 1 had mean 0.001198 and std 0.627953  
hidden layer 2 had mean -0.000175 and std 0.486051  
hidden layer 3 had mean 0.000055 and std 0.407723  
hidden layer 4 had mean -0.000306 and std 0.357108  
hidden layer 5 had mean 0.000142 and std 0.320917  
hidden layer 6 had mean -0.000389 and std 0.292116  
hidden layer 7 had mean -0.000228 and std 0.273387  
hidden layer 8 had mean -0.000291 and std 0.254935  
hidden layer 9 had mean 0.000361 and std 0.239266  
hidden layer 10 had mean 0.000139 and std 0.228008
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

“Xavier initialization”
[Glorot et al., 2010]

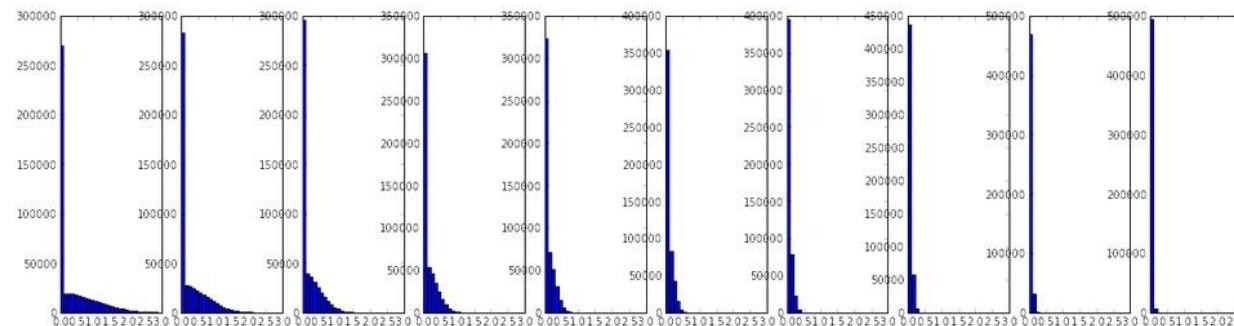
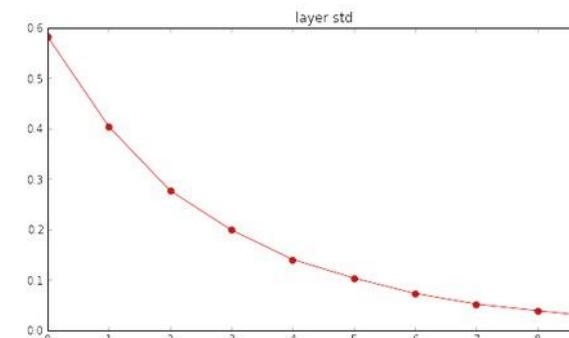
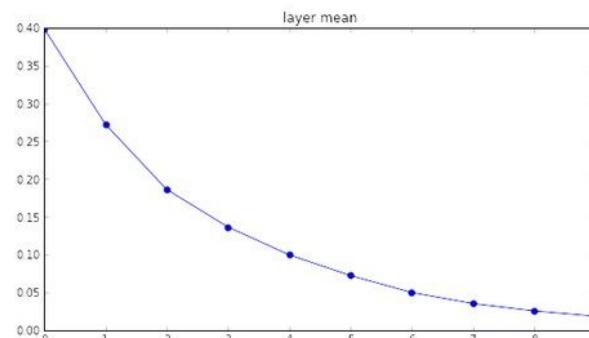


Reasonable initialization.
(Mathematical derivation
assumes linear activations)

```
input layer had mean 0.000501 and std 0.999444  
hidden layer 1 had mean 0.398623 and std 0.582273  
hidden layer 2 had mean 0.272352 and std 0.403795  
hidden layer 3 had mean 0.186076 and std 0.276912  
hidden layer 4 had mean 0.136442 and std 0.198685  
hidden layer 5 had mean 0.099568 and std 0.140299  
hidden layer 6 had mean 0.072234 and std 0.103280  
hidden layer 7 had mean 0.049775 and std 0.072748  
hidden layer 8 had mean 0.035138 and std 0.051572  
hidden layer 9 had mean 0.025494 and std 0.038583  
hidden layer 10 had mean 0.018408 and std 0.026697
```

```
w = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

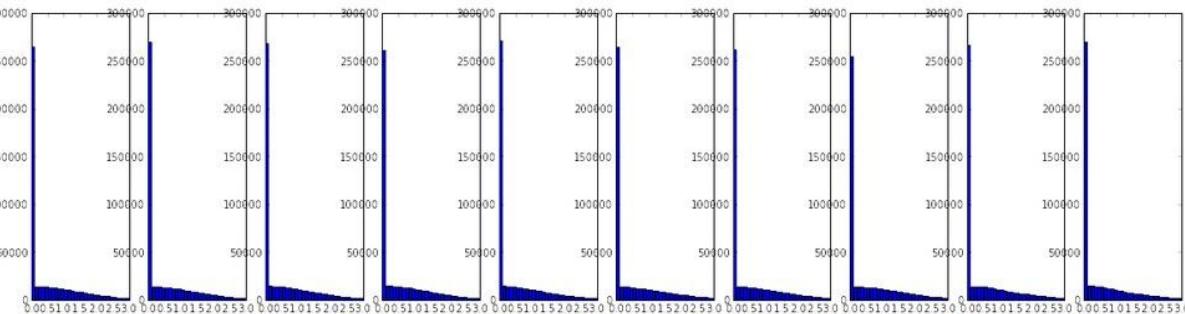
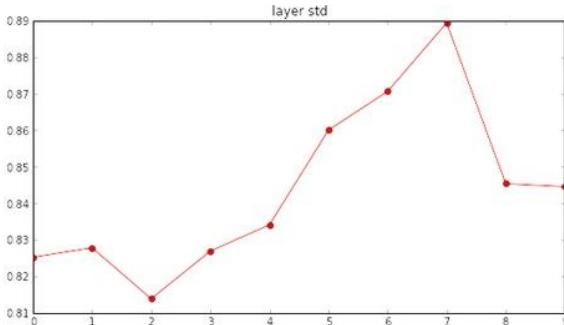
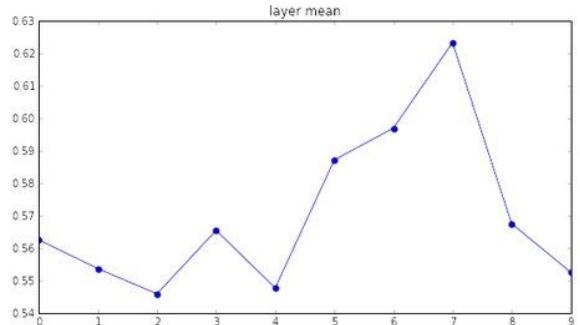
but when using the ReLU nonlinearity it breaks.



```
input layer had mean 0.000501 and std 0.999444  
hidden layer 1 had mean 0.562488 and std 0.825232  
hidden layer 2 had mean 0.553614 and std 0.827835  
hidden layer 3 had mean 0.545867 and std 0.813855  
hidden layer 4 had mean 0.565396 and std 0.826902  
hidden layer 5 had mean 0.547678 and std 0.834092  
hidden layer 6 had mean 0.587103 and std 0.860035  
hidden layer 7 had mean 0.596867 and std 0.870610  
hidden layer 8 had mean 0.623214 and std 0.889348  
hidden layer 9 had mean 0.567498 and std 0.845357  
hidden layer 10 had mean 0.552531 and std 0.844523
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(2/fan_in) # layer initialization
```

He et al., 2015
(note additional 2/)



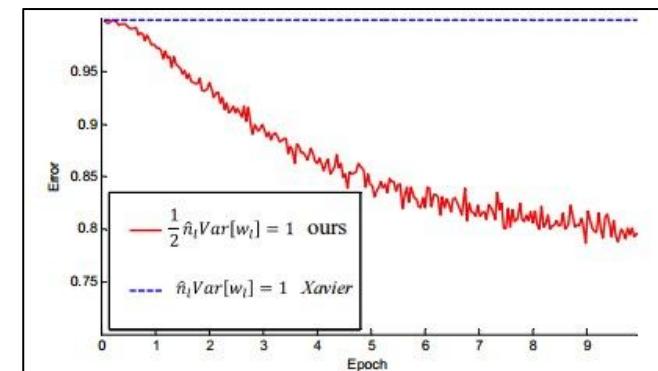
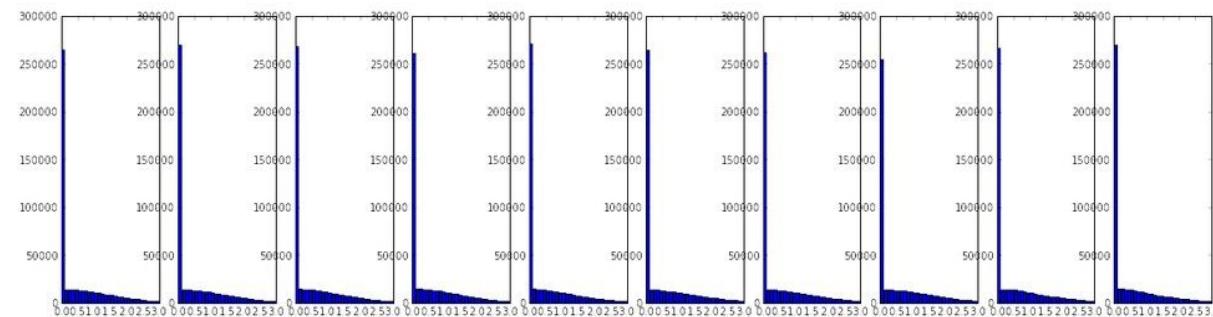
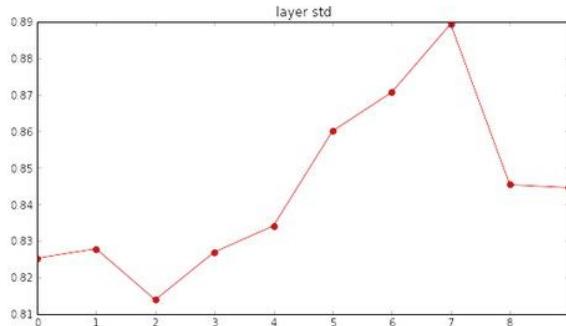
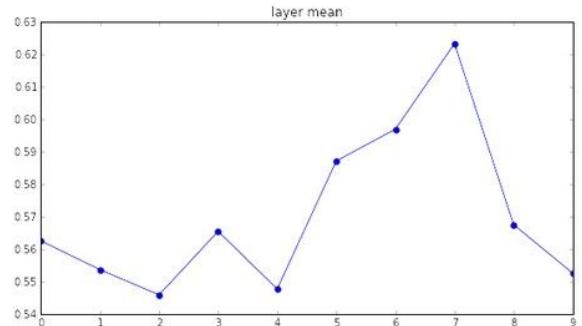
```

input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523

```

`W = np.random.randn(fan_in, fan_out) / np.sqrt(2/fan_in) # layer initialization`

He et al., 2015
(note additional 2/)



Proper initialization is an active area of research...

Understanding the difficulty of training deep feedforward neural networks

by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by

Saxe et al, 2013

Random walk initialization for training very deep feedforward networks by Sussillo and

Abbott, 2014

Delving deep into rectifiers: Surpassing human-level performance on ImageNet

classification by He et al., 2015

Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015

All you need is a good init, Mishkin and Matas, 2015

...

Batch Normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

“you want zero-mean unit-variance activations? just make them so.”

consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

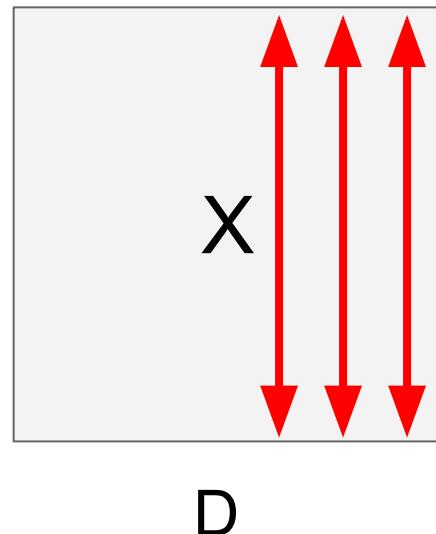
$$\hat{x}^{(k)} = \frac{x^{(k)} - \text{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

this is a vanilla
differentiable function...

Batch Normalization

[Ioffe and Szegedy, 2015]

“you want zero-mean unit-variance activations? just make them so.”



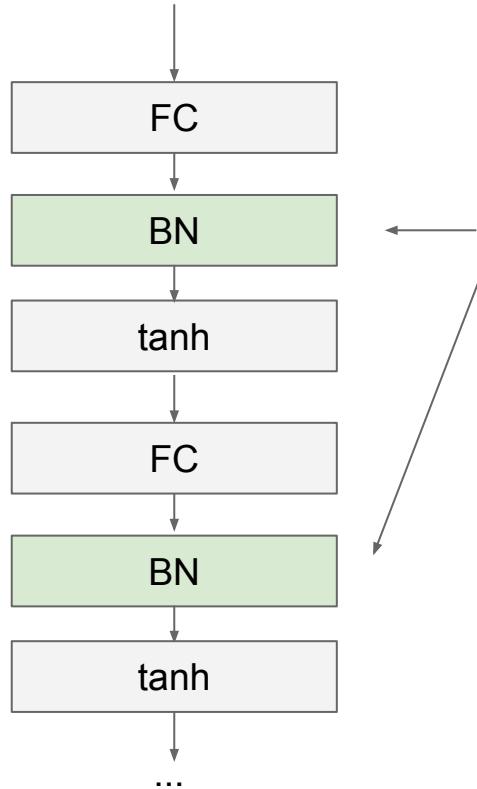
1. compute the empirical mean and variance independently for each dimension.

2. Normalize

$$\hat{x}^{(k)} = \frac{x^{(k)} - \text{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

Batch Normalization

[Ioffe and Szegedy, 2015]

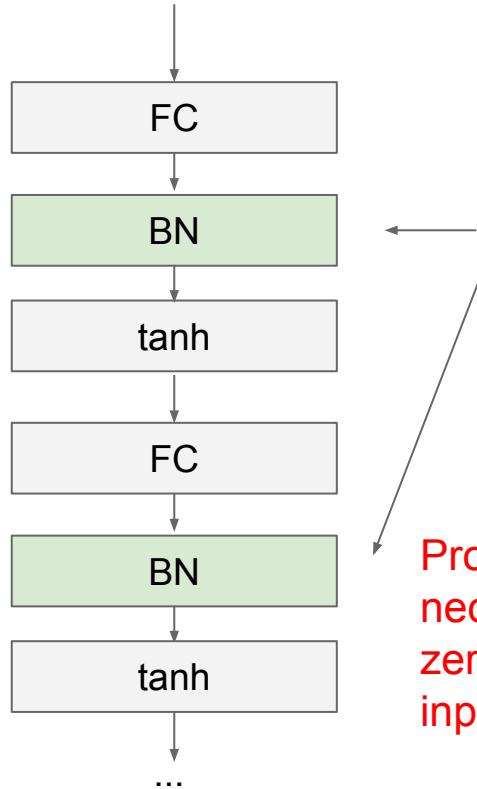


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\hat{x}^{(k)} = \frac{x^{(k)} - \text{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

Batch Normalization

[Ioffe and Szegedy, 2015]



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

Problem: do we necessarily want a zero-mean unit-variance input?

$$\hat{x}^{(k)} = \frac{x^{(k)} - \text{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \text{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \text{E}[x^{(k)}]$$

to recover the identity mapping.

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

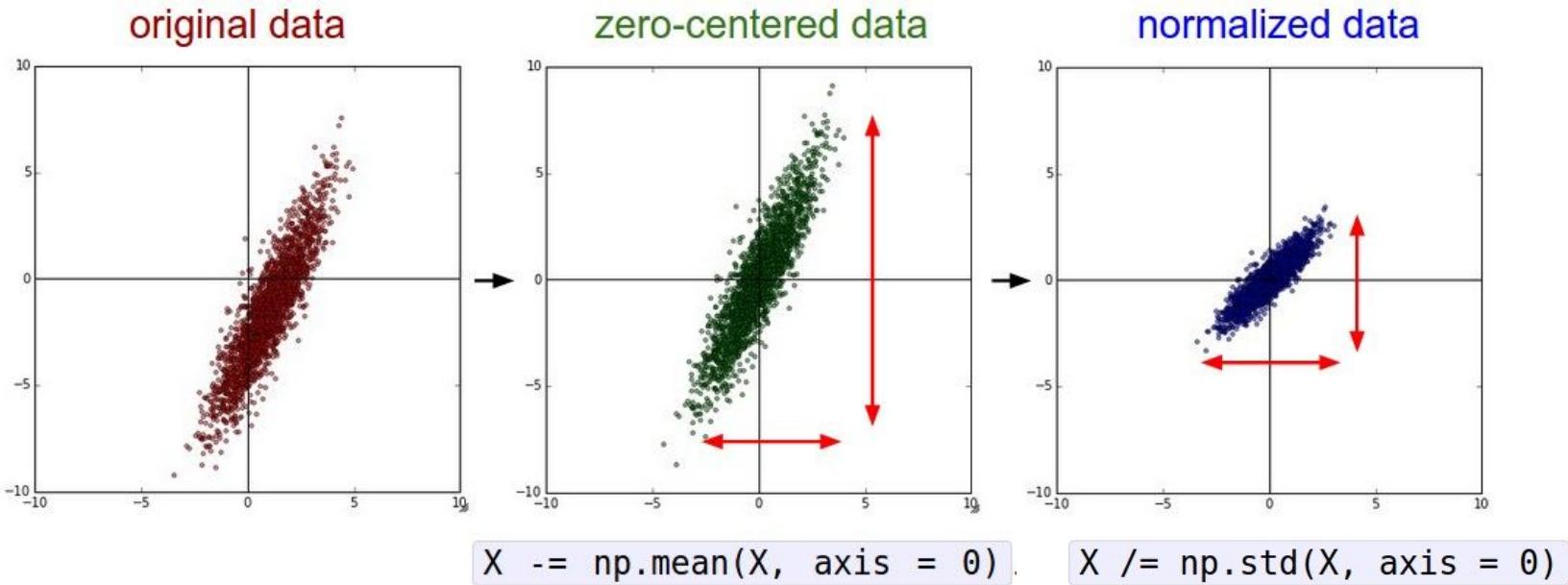
Note: at test time BatchNorm layer functions differently:

The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

(e.g. can be estimated during training with running averages)

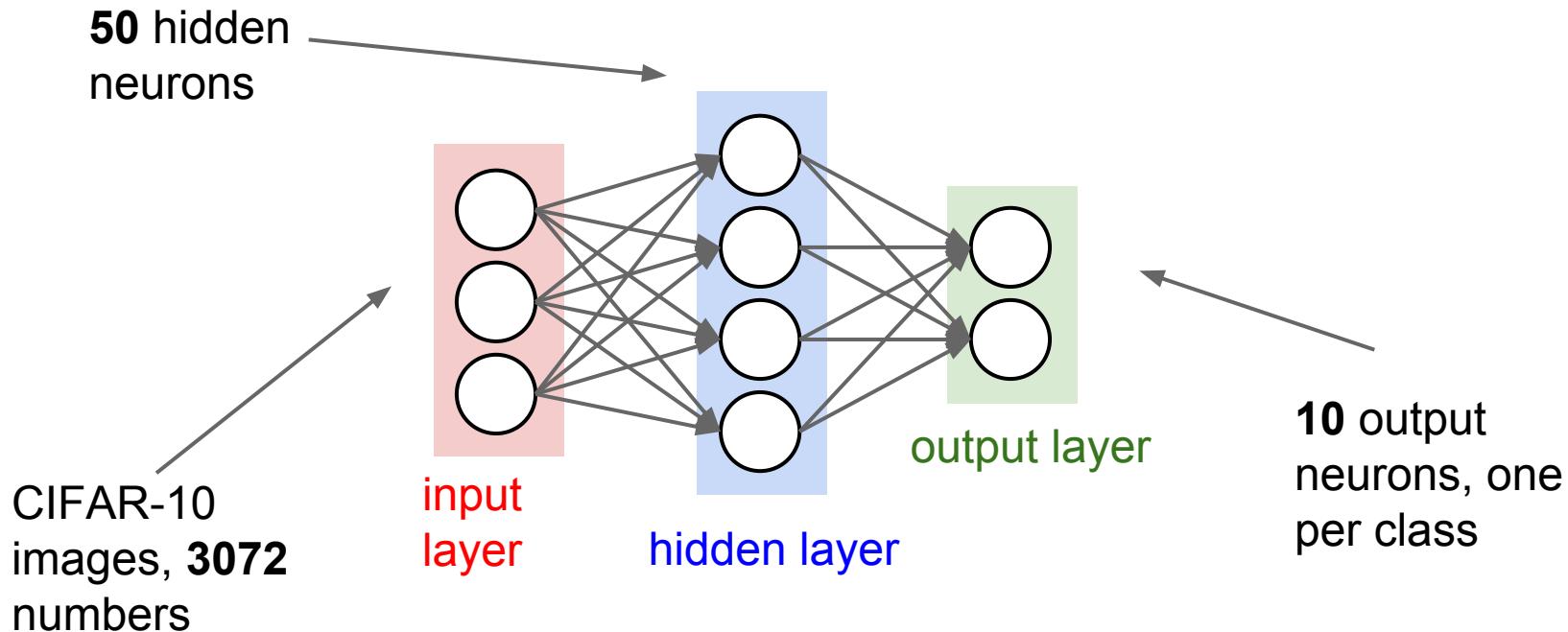
Babysitting the Learning Process

Step 1: Preprocess the data



(Assume X [NxD] is data matrix,
each example in a row)

Step 2: Choose the architecture: say we start with one hidden layer of 50 neurons:



Double check that the loss is reasonable:

```
def init_two_layer_model(input_size, hidden_size, output_size):
    # initialize a model
    model = {}
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
    model['b1'] = np.zeros(hidden_size)
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
    model['b2'] = np.zeros(output_size)
    return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 0.0) # disable regularization
print loss
```

2.30261216167

loss ~2.3.

“correct” for
10 classes

returns the loss and the
gradient for all parameters

“correct” for
10 classes

Double check that the loss is reasonable:

```
def init_two_layer_model(input_size, hidden_size, output_size):
    # initialize a model
    model = {}
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
    model['b1'] = np.zeros(hidden_size)
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
    model['b2'] = np.zeros(output_size)
    return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 1e3)      crank up regularization
print loss
```

3.06859716482

loss went up, good. (sanity check)

Lets try to train now...

Tip: Make sure that you can overfit very small portion of the training data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

The above code:

- take the first 20 examples from CIFAR-10
- turn off regularization ($\text{reg} = 0.0$)
- use simple vanilla ‘sgd’

Lets try to train now...

Tip: Make sure that you can overfit very small portion of the training data

Very small loss,
train accuracy 1.00,
nice!

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 20 / 200: cost 1.325750, train: 0.550000, val 0.550000, lr 1.000000e-03
...
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
    model, two_layer_net,
    num_epochs=10, reg=0.000001,
    update='sgd', learning_rate_decay=1,
    sample_batches = True,
    learning_rate=1e-6, verbose=True)
```

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
    model, two_layer_net,
    num_epochs=10, reg=0.000001,
    update='sgd', learning_rate_decay=1,
    sample_batches=True,
    learning_rate=1e-6, verbose=True)

Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
    model, two_layer_net,
    num_epochs=10, reg=0.000001,
    update='sgd', learning_rate_decay=1,
    sample_batches=True,
    learning_rate=1e-6, verbose=True)

Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing: Learning rate is probably too low

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
    model, two_layer_net,
    num_epochs=10, reg=0.000001,
    update='sgd', learning_rate_decay=1,
    sample_batches=True,
    learning_rate=1e-6, verbose=True)

Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing: Learning rate is probably too low

Notice train/val accuracy goes to 20% though, what's up with that? (remember this is softmax)

Lets try to train now...

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                   model, two_layer_net,
                                   num_epochs=10, reg=0.00001,
                                   update='sgd', learning_rate_decay=1,
                                   sample_batches = True,
```

Start with small regularization and find learning rate that makes the loss go down.

Now let's try learning rate 1e6.



loss not going down:
learning rate too low

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                   model, two_layer_net,
                                   num_epochs=10, reg=0.00001,
                                   update='sgd', learning_rate_decay=1,
                                   sample_batches = True,
                                   learning_rate=1e-6, verbose=True)

/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:50: RuntimeWarning: divide by zero en
countered in log
    data_loss = -np.sum(np.log(probs[range(N), y])) / N
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:48: RuntimeWarning: invalid value enc
ountered in subtract
    probs = np.exp(scores - np.max(scores, axis=1, keepdims=True))
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

loss not going down:
learning rate too low
loss exploding:
learning rate too high

cost: NaN almost
always means high
learning rate...

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
    model, two_layer_net,
    num_epochs=10, reg=0.000001,
    update='sgd', learning_rate_decay=1,
    sample_batches = True,
    learning_rate=3e-3, verbose=True)

Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

3e-3 is still too high. Cost explodes....

loss not going down:
learning rate too low
loss exploding:
learning rate too high

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 ... 1e-5]

Hyperparameter Optimization

Cross-validation strategy

coarse -> fine cross-validation in stages

First stage: only a few epochs to get rough idea of what params work

Second stage: longer running time, finer search

... (repeat as necessary)

Tip for detecting explosions in the solver:

If the cost is ever $> 3 * \text{original cost}$, break out early

For example: run coarse search for 5 epochs

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5) ←
    lr = 10**uniform(-3, -6)

    trainer = ClassifierTrainer()
    model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
    trainer = ClassifierTrainer()
    best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                              model, two_layer_net,
                                              num_epochs=5, reg=reg,
                                              update='momentum', learning_rate_decay=0.9,
                                              sample_batches = True, batch_size = 100,
                                              learning rate=lr, verbose=False)
```

note it's best to optimize
in log space!

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

→ nice

Now run finer search...

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)
```

adjust range

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-4, 0)
    lr = 10**uniform(-3, -4)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good
for a 2-layer neural net
with 50 hidden neurons.

Now run finer search...

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)
```

adjust range

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-4, 0)
    lr = 10**uniform(-3, -4)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100) ←
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good
for a 2-layer neural net
with 50 hidden neurons.

But this best
cross-validation result is
worrying. Why?

Random Search vs. Grid Search

*Random Search for
Hyper-Parameter Optimization
Bergstra and Bengio, 2012*

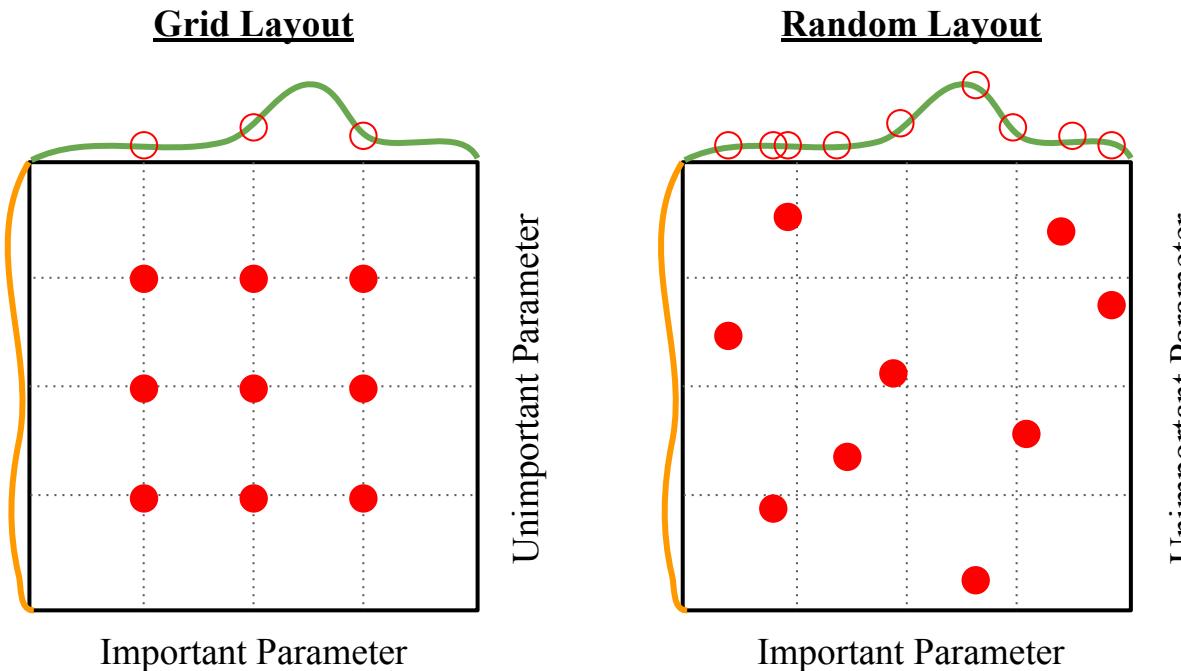


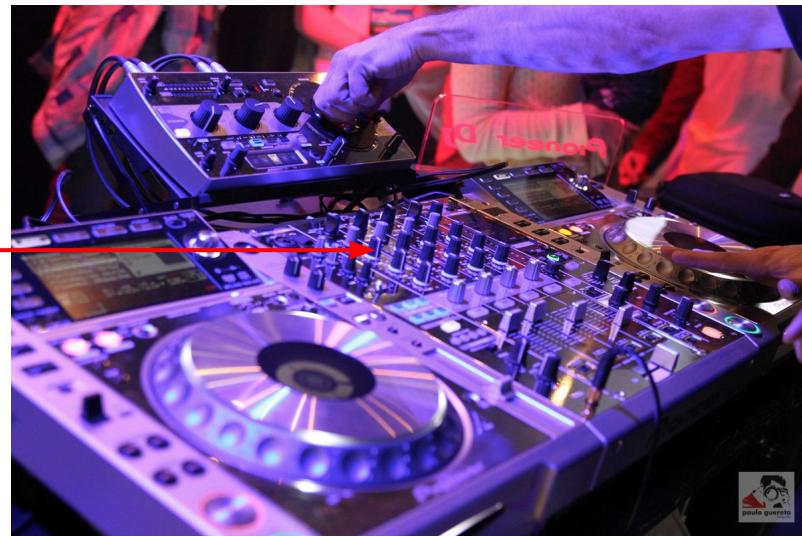
Illustration of Bergstra et al., 2012 by Shayne
Longpre, copyright CS231n 2017

Hyperparameters to play with:

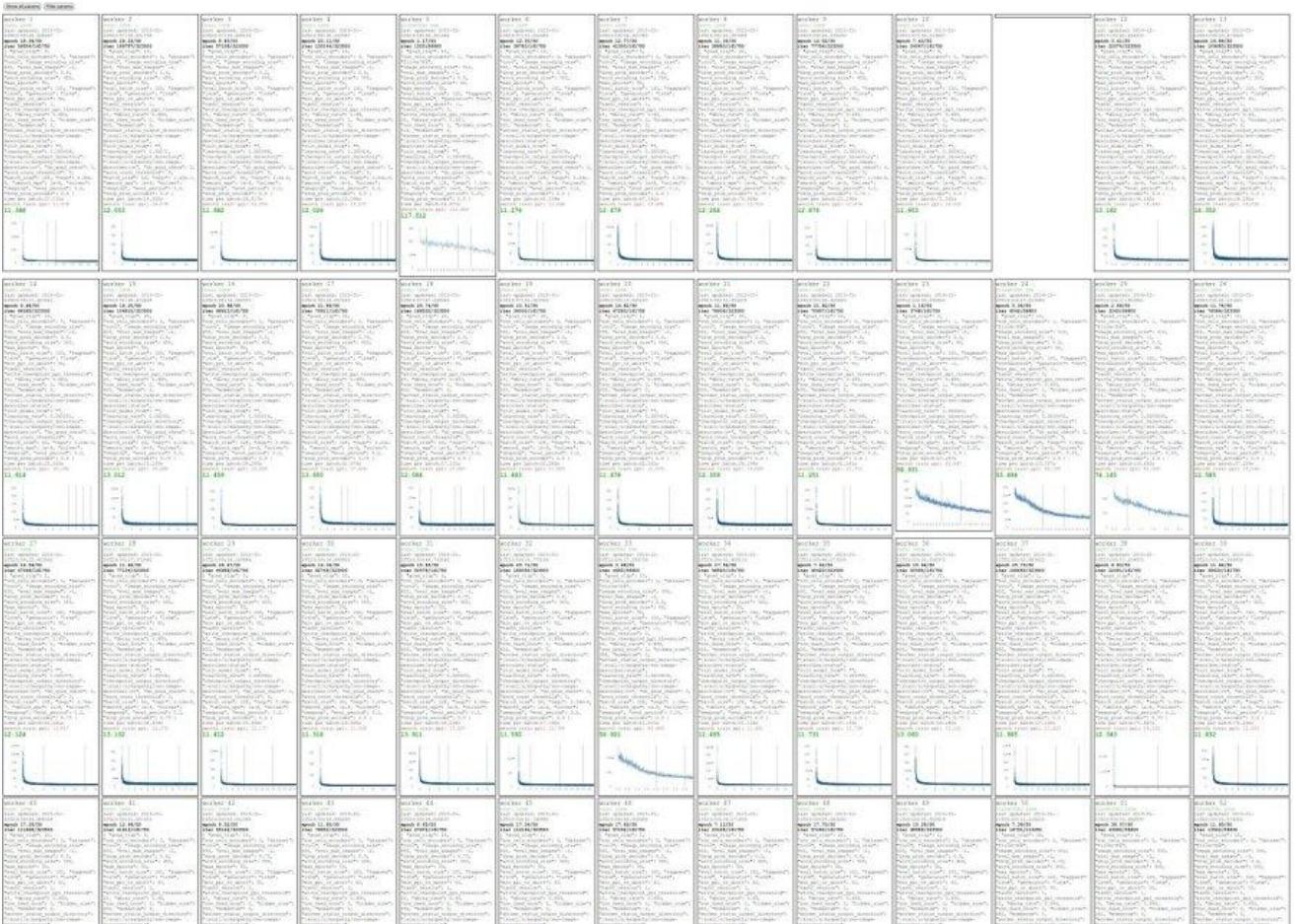
- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

neural networks practitioner
music = loss function

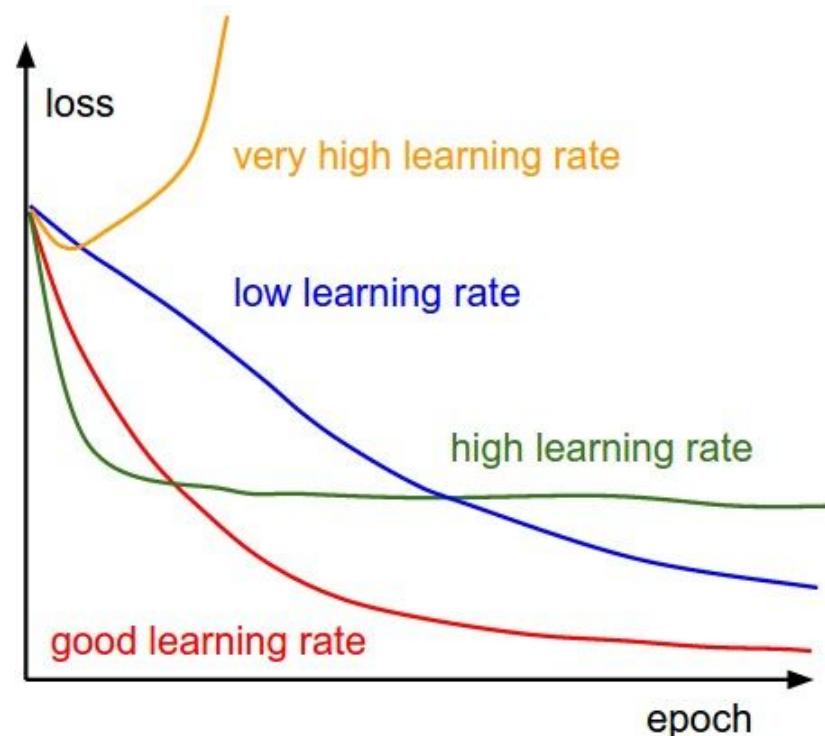
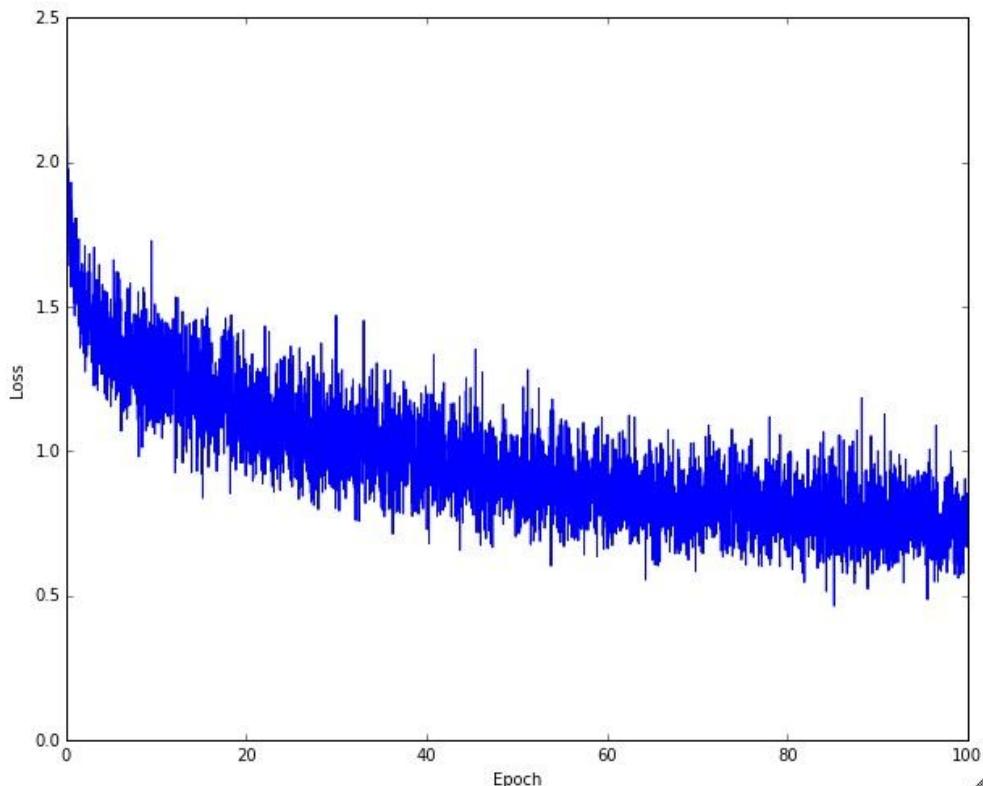
[This image](#) by Paolo Guereta is licensed under [CC-BY 2.0](#)

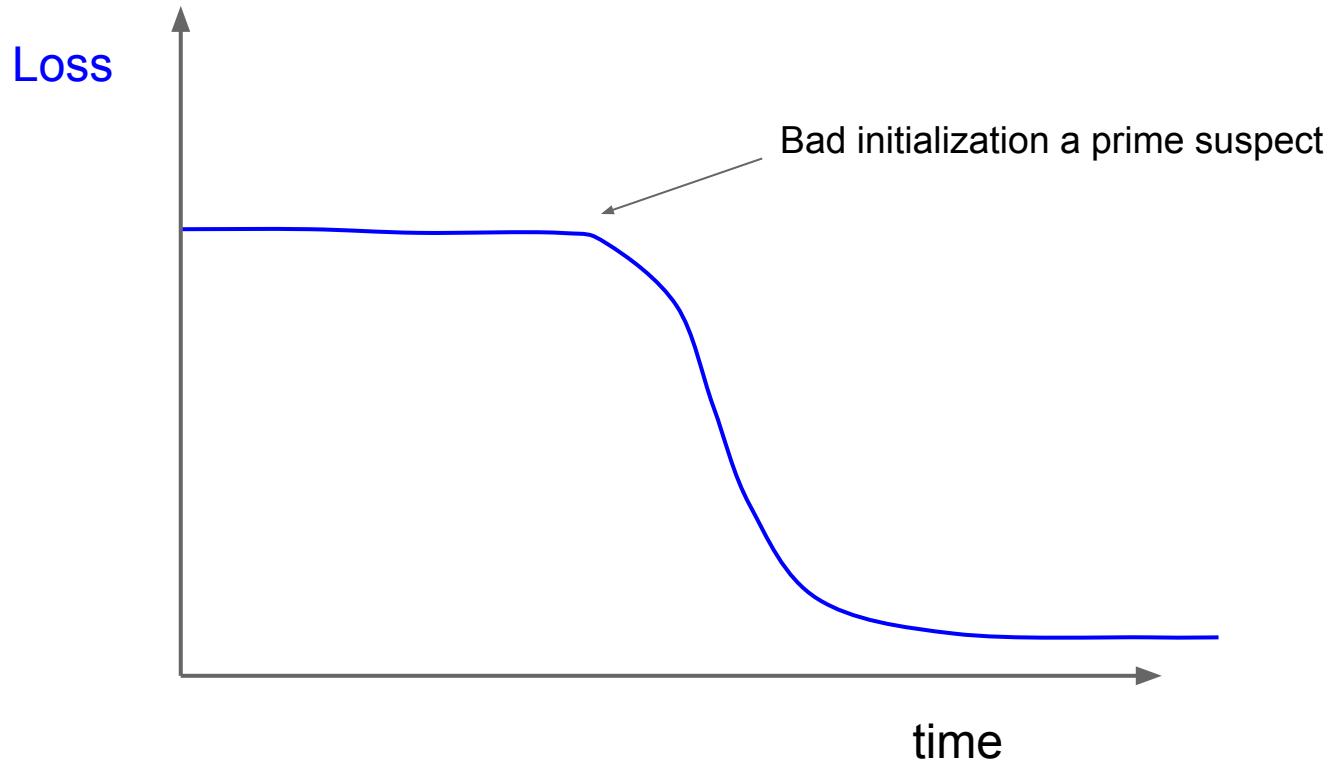


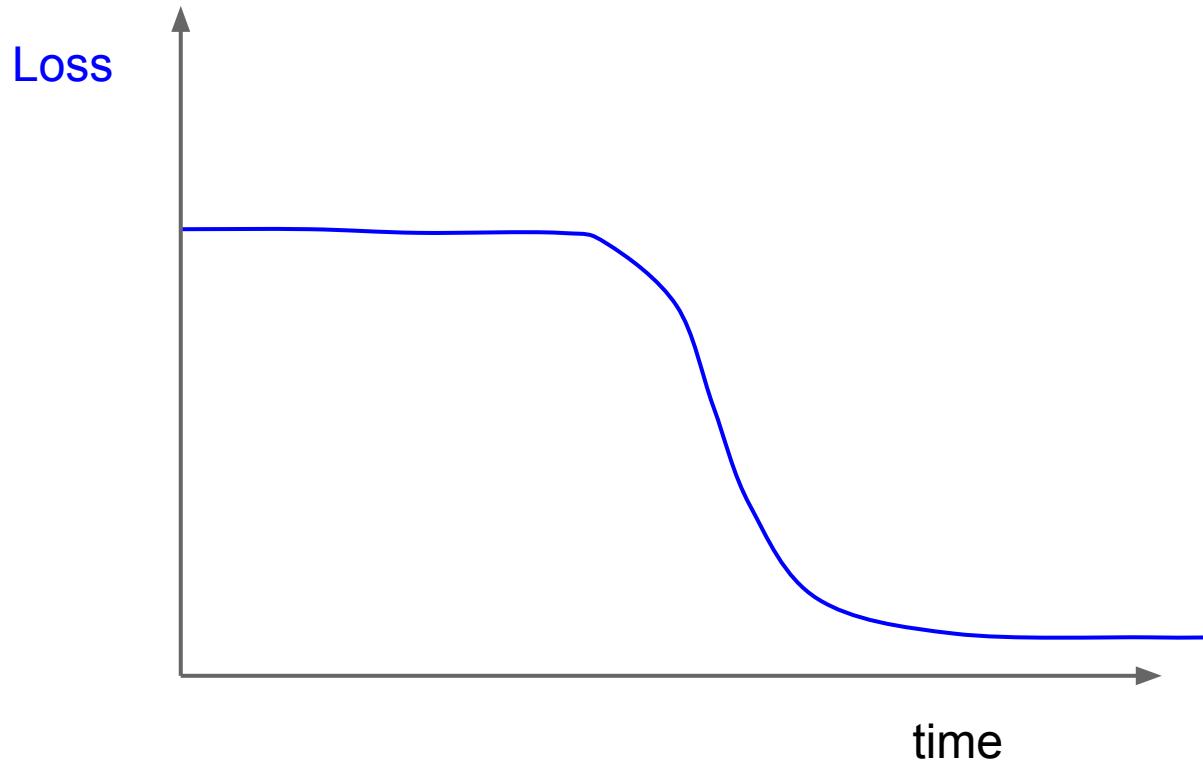
Cross-validation “command center”



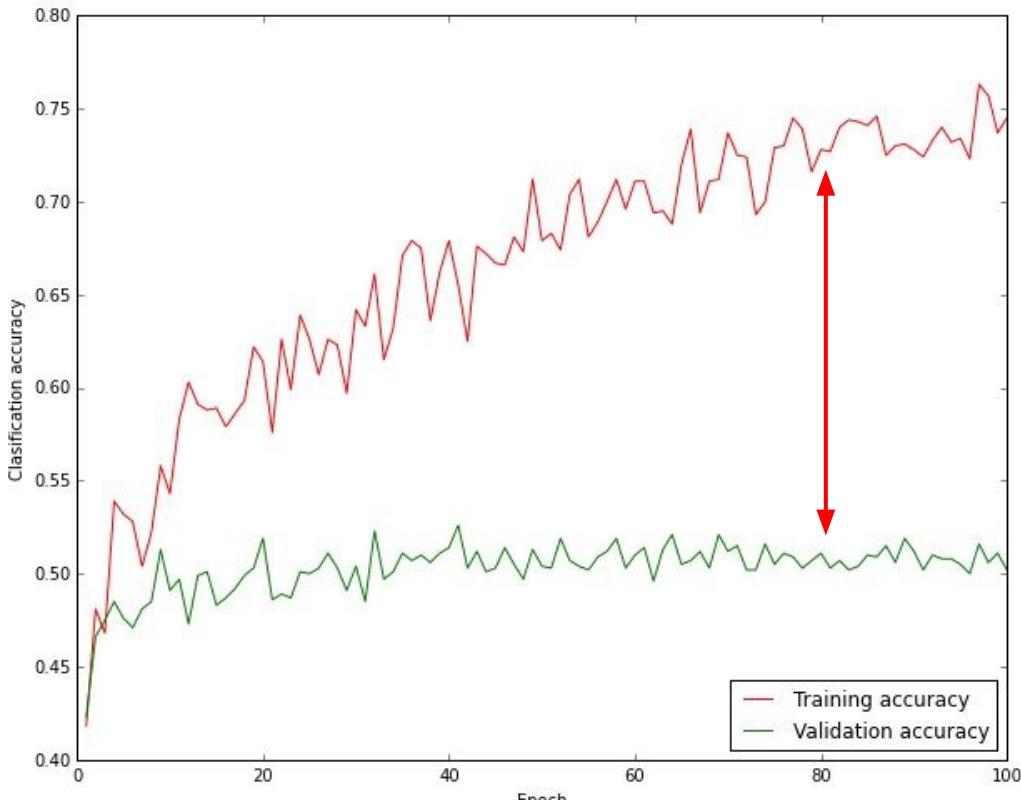
Monitor and visualize the loss curve







Monitor and visualize the accuracy:



big gap = overfitting
=> increase regularization strength?

no gap
=> increase model capacity?

Track the ratio of weight updates / weight magnitudes:

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

ratio between the updates and values: $\sim 0.0002 / 0.02 = 0.01$ (about okay)
want this to be somewhere around 0.001 or so

Summary

TLDRs

We looked in detail at:

- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier/He init)
- Batch Normalization (use)
- Babysitting the Learning process
- Hyperparameter Optimization
(random sample hyperparams, in log space when appropriate)

Next time:

Training Neural Networks, Part 2

- Parameter update schemes
- Learning rate schedules
- Gradient checking
- Regularization (Dropout etc.)
- Evaluation (Ensembles etc.)
- Transfer learning / fine-tuning

Lecture 7: Training Neural Networks, Part 2

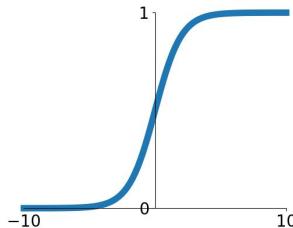
Administrative

- Assignment 1 is being graded, stay tuned
- Project proposals due tomorrow by 11:59pm on Gradescope
- Assignment 2 is out, due Wednesday 5/2 11:59pm

Last time: Activation Functions

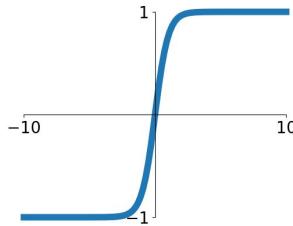
Sigmoid

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



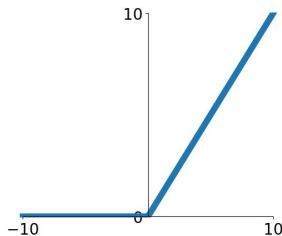
tanh

$$\tanh(x)$$

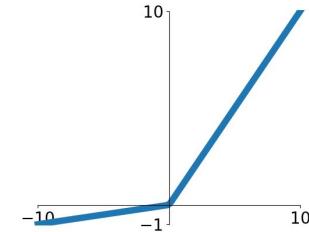


ReLU

$$\max(0, x)$$

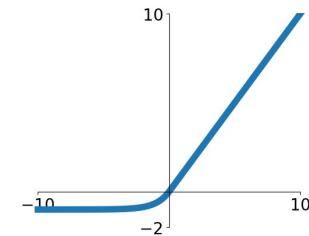


Leaky ReLU
 $\max(0.1x, x)$



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

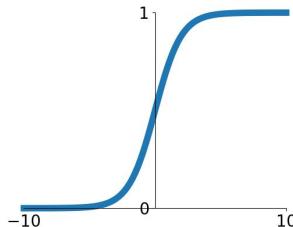
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Last time: Activation Functions

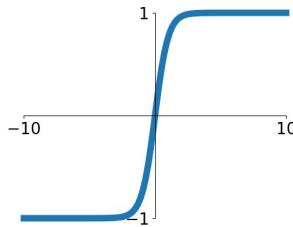
Sigmoid

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



tanh

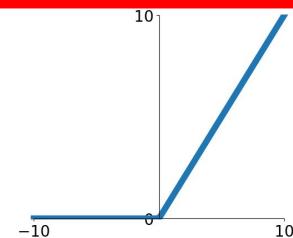
$$\tanh(x)$$



ReLU

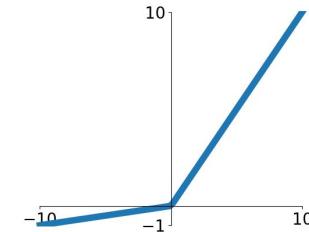
$$\max(0, x)$$

Good default choice



Leaky ReLU

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

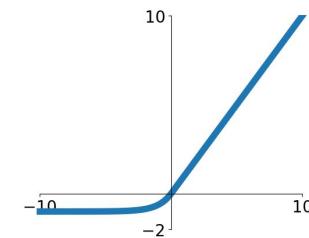


Maxout

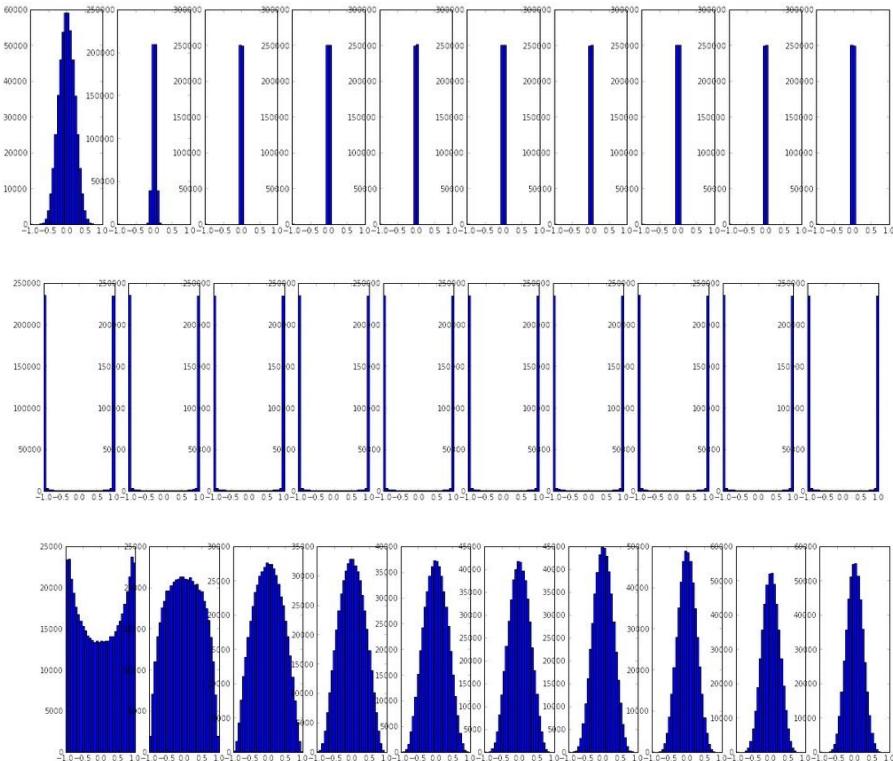
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

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



Last time: Weight Initialization



Initialization too small:

Activations go to zero, gradients also zero,
No learning

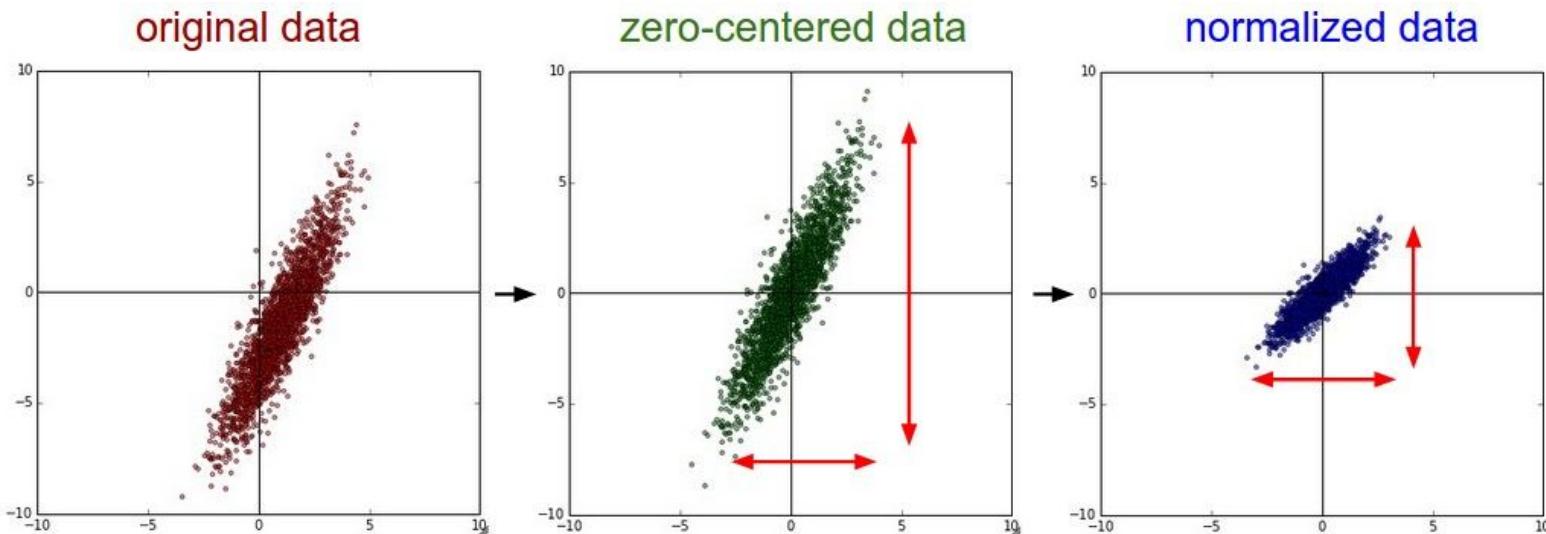
Initialization too big:

Activations saturate (for tanh),
Gradients zero, no learning

Initialization just right:

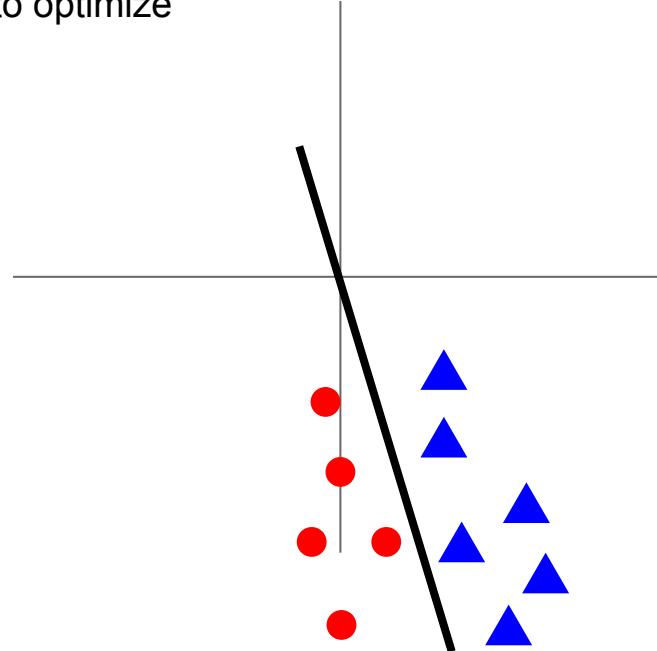
Nice distribution of activations at all layers,
Learning proceeds nicely

Last time: Data Preprocessing

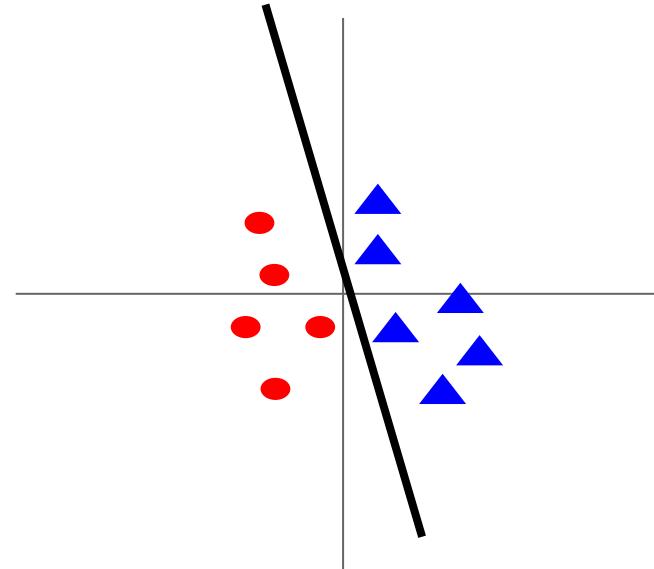


Last time: Data Preprocessing

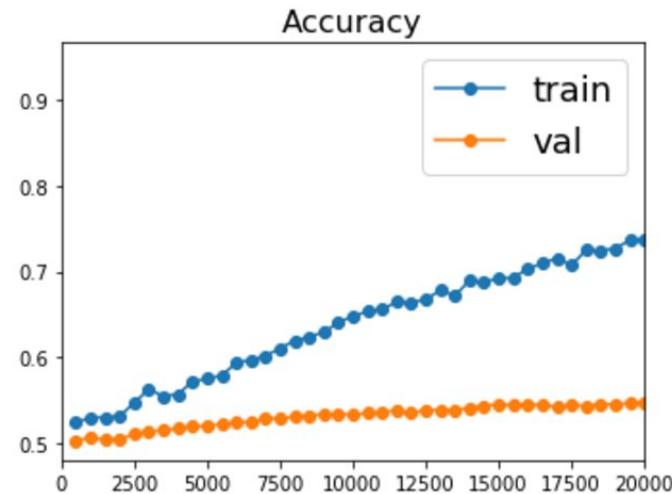
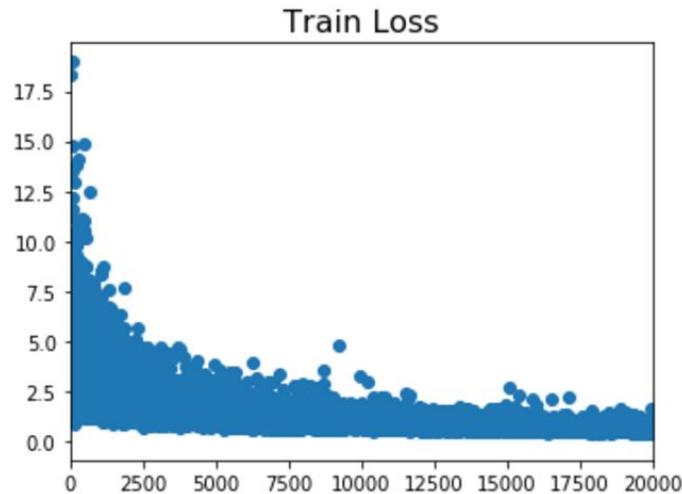
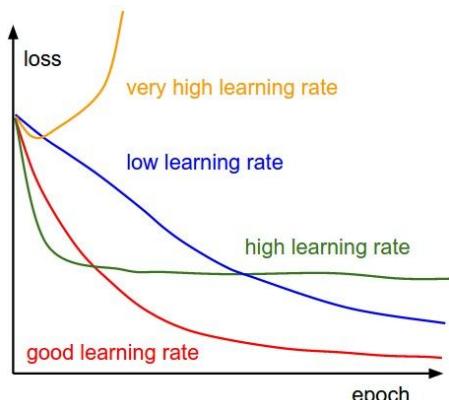
Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize

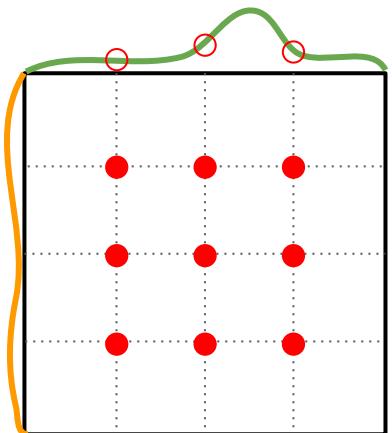


Last time: Babysitting Learning



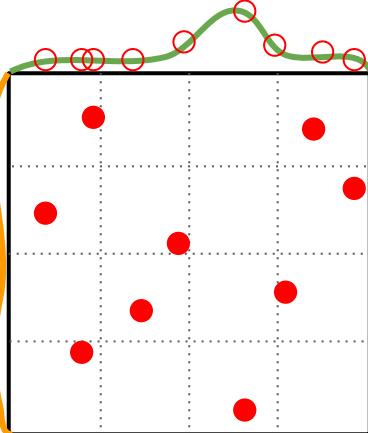
Last time: Hyperparameter Search

Grid Layout



Important
Parameter

Random Layout



Important
Parameter

Unimportant
Parameter

Unimportant
Parameter

Coarse to fine search

```
val acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val acc: 0.280000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val acc: 0.441000, lr: 1.758259e-04, reg: 2.110807e-04, (7 / 100)
val acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

```
val acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val acc: 0.512000, lr: 8.686827e-04, reg: 1.349727e-02, (2 / 100)
val acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val acc: 0.475000, lr: 2.021162e-04, reg: 2.287870e-01, (11 / 100)
val acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

Today

- More normalization
- Fancier optimization
- Regularization
- Transfer Learning

Last time: Batch Normalization

Input: $x : N \times D$

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Learnable params:

$$\gamma, \beta : D$$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Intermediates: $\mu, \sigma : D$
 $\hat{x} : N \times D$

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

Output: $y : N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Last time: Batch Normalization

Estimate mean and variance from minibatch;
Can't do this at test-time

Input: $x : N \times D$

Learnable params:

$\gamma, \beta : D$

Intermediates: $\mu, \sigma : D$
 $\hat{x} : N \times D$

Output: $y : N \times D$

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

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

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Batch Normalization: Test Time

Input: $x : N \times D$

$\mu_j =$ (Running) average of values
seen during training

Learnable params:

$\gamma, \beta : D$

$\sigma_j^2 =$ (Running) average of values
seen during training

Intermediates: $\mu, \sigma : D$
 $\hat{x} : N \times D$

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

Output: $y : N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Batch Normalization for ConvNets

Batch Normalization for
fully-connected networks

$$\begin{aligned} \mathbf{x}: & \quad \mathbf{N} \times \mathbf{D} \\ \text{Normalize} & \quad \downarrow \\ \boldsymbol{\mu}, \sigma: & \quad 1 \times \mathbf{D} \\ \boldsymbol{\gamma}, \beta: & \quad 1 \times \mathbf{D} \\ \mathbf{y} = & \quad \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \sigma + \beta \end{aligned}$$

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

$$\begin{aligned} \mathbf{x}: & \quad \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W} \\ \text{Normalize} & \quad \downarrow \quad \downarrow \quad \downarrow \\ \boldsymbol{\mu}, \sigma: & \quad 1 \times \mathbf{C} \times 1 \times 1 \\ \boldsymbol{\gamma}, \beta: & \quad 1 \times \mathbf{C} \times 1 \times 1 \\ \mathbf{y} = & \quad \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \sigma + \beta \end{aligned}$$

Layer Normalization

Batch Normalization for
fully-connected networks

Layer Normalization for
fully-connected networks
Same behavior at train and test!
Can be used in recurrent networks

$\mathbf{x}: N \times D$

Normalize



$\mu, \sigma: 1 \times D$

$\gamma, \beta: 1 \times D$

$$\mathbf{y} = \gamma(\mathbf{x} - \boldsymbol{\mu}) / \sigma + \boldsymbol{\beta}$$

$\mathbf{x}: N \times D$

Normalize



$\mu, \sigma: N \times 1$

$\gamma, \beta: 1 \times D$

$$\mathbf{y} = \gamma(\mathbf{x} - \boldsymbol{\mu}) / \sigma + \boldsymbol{\beta}$$

Ba, Kiros, and Hinton, "Layer Normalization", arXiv 2016

Instance Normalization

Batch Normalization for convolutional networks

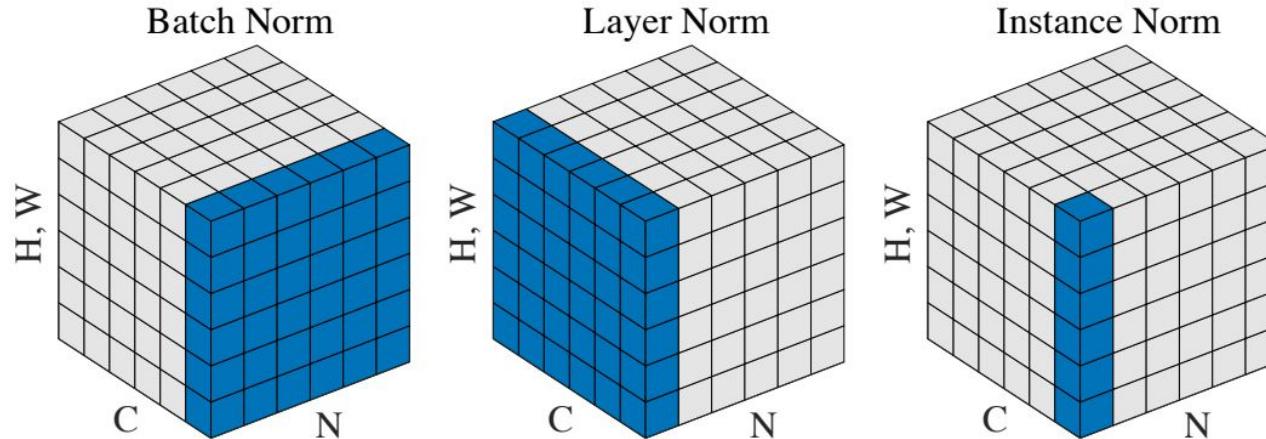
$$\begin{aligned} \mathbf{x}: & \quad \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W} \\ \text{Normalize} & \quad \downarrow \quad \downarrow \quad \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma}: & \quad \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \boldsymbol{\gamma}, \boldsymbol{\beta}: & \quad \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \mathbf{y} = & \quad \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta} \end{aligned}$$

Instance Normalization for convolutional networks
Same behavior at train / test!

$$\begin{aligned} \mathbf{x}: & \quad \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W} \\ \text{Normalize} & \quad \downarrow \quad \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma}: & \quad \mathbf{N} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \boldsymbol{\gamma}, \boldsymbol{\beta}: & \quad \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \mathbf{y} = & \quad \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta} \end{aligned}$$

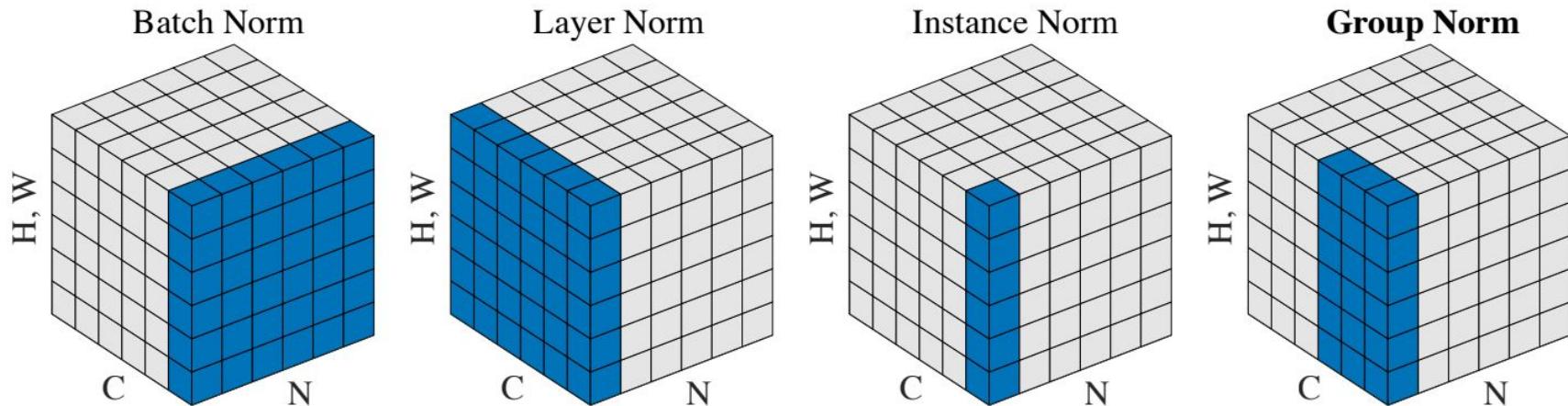
Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017

Comparison of Normalization Layers



Wu and He, "Group Normalization", arXiv 2018

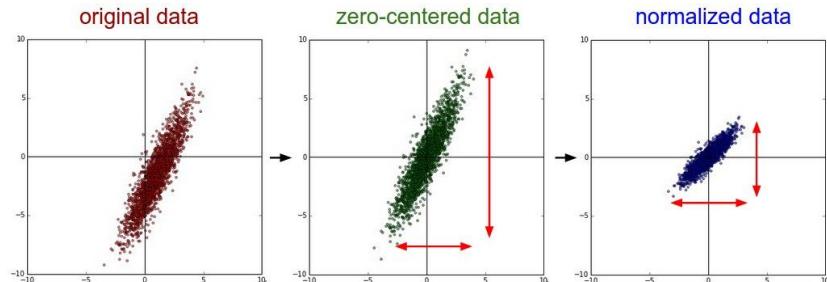
Group Normalization



Wu and He, "Group Normalization", arXiv 2018 (Appeared 3/22/2018)

Decorrelated Batch Normalization

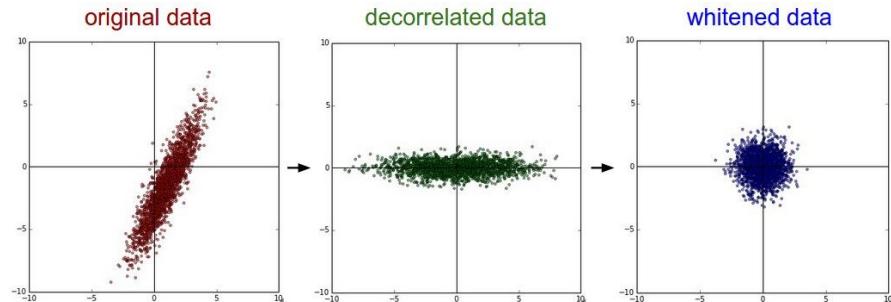
Batch Normalization



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

BatchNorm normalizes the data, but cannot correct for correlations among the input features

Decorrelated Batch Normalization



$$\hat{x}_i = \Sigma^{-\frac{1}{2}}(x_i - \mu)$$

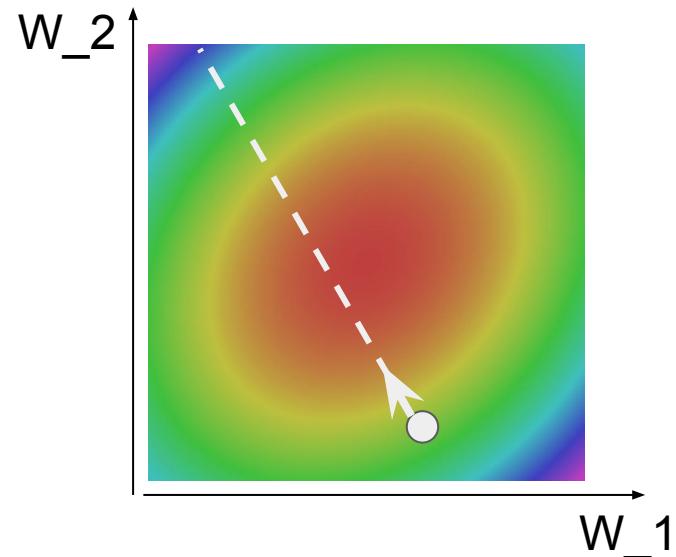
DBN **whitens** the data using the full covariance matrix of the minibatch; this corrects for correlations

Huang et al, "Decorrelated Batch Normalization", arXiv 2018 (Appeared 4/23/2018)

Optimization

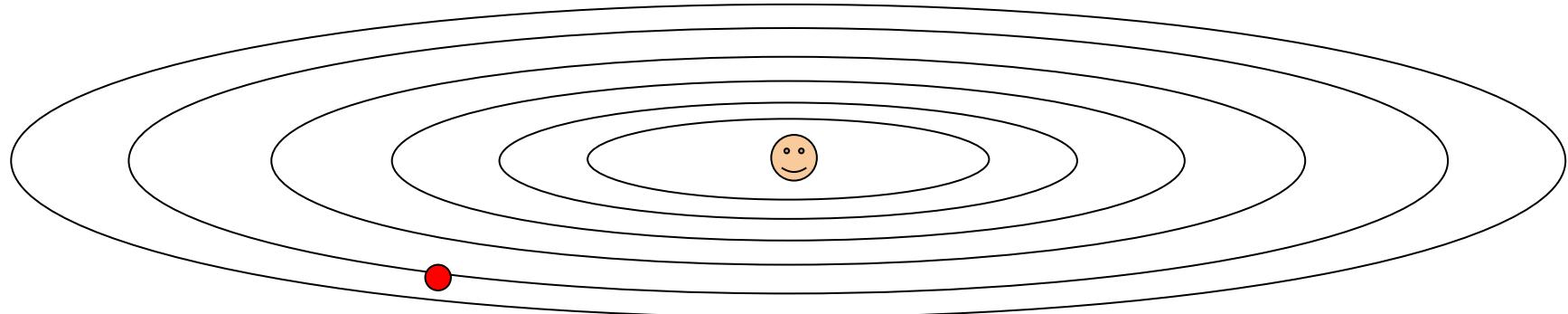
```
# Vanilla Gradient Descent

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



Optimization: Problems with SGD

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?



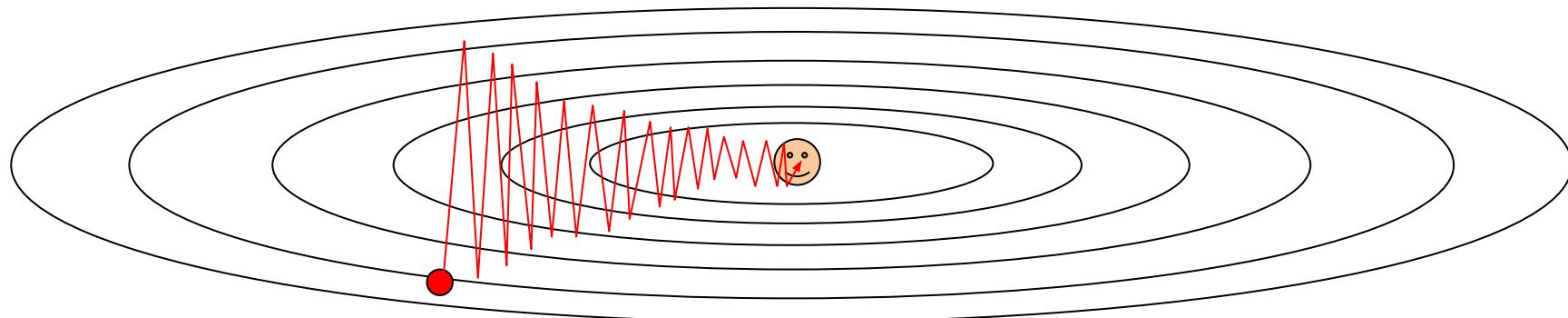
Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Optimization: Problems with SGD

What if loss changes quickly in one direction and slowly in another?

What does gradient descent do?

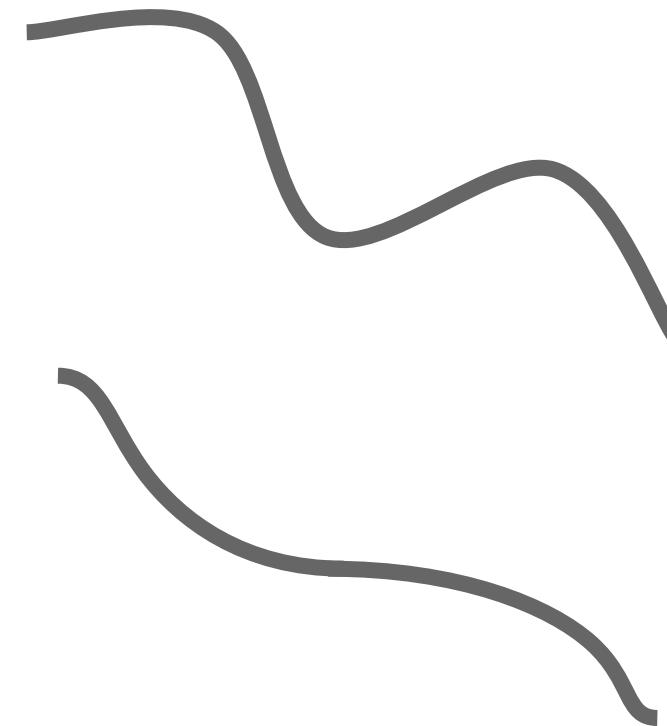
Very slow progress along shallow dimension, jitter along steep direction



Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Optimization: Problems with SGD

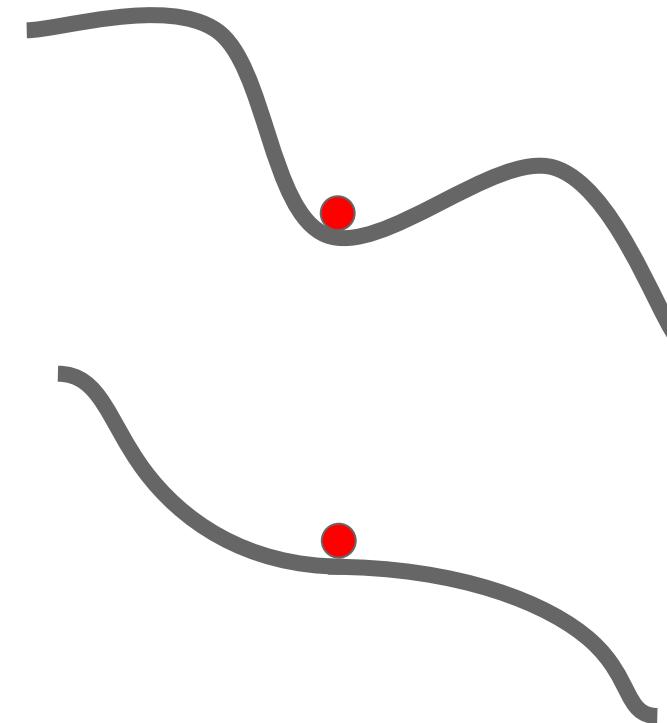
What if the loss
function has a
local minima or
saddle point?



Optimization: Problems with SGD

What if the loss
function has a
local minima or
saddle point?

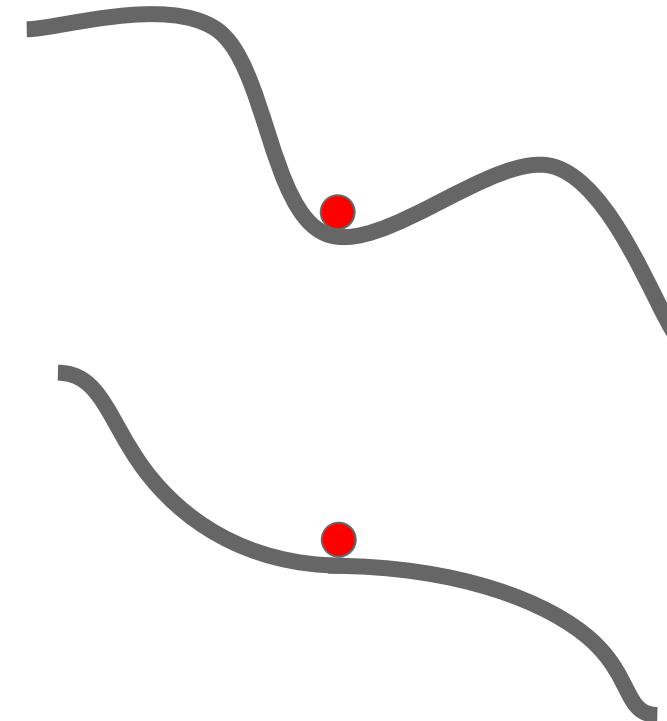
Zero gradient,
gradient descent
gets stuck



Optimization: Problems with SGD

What if the loss
function has a
local minima or
saddle point?

Saddle points much
more common in
high dimension



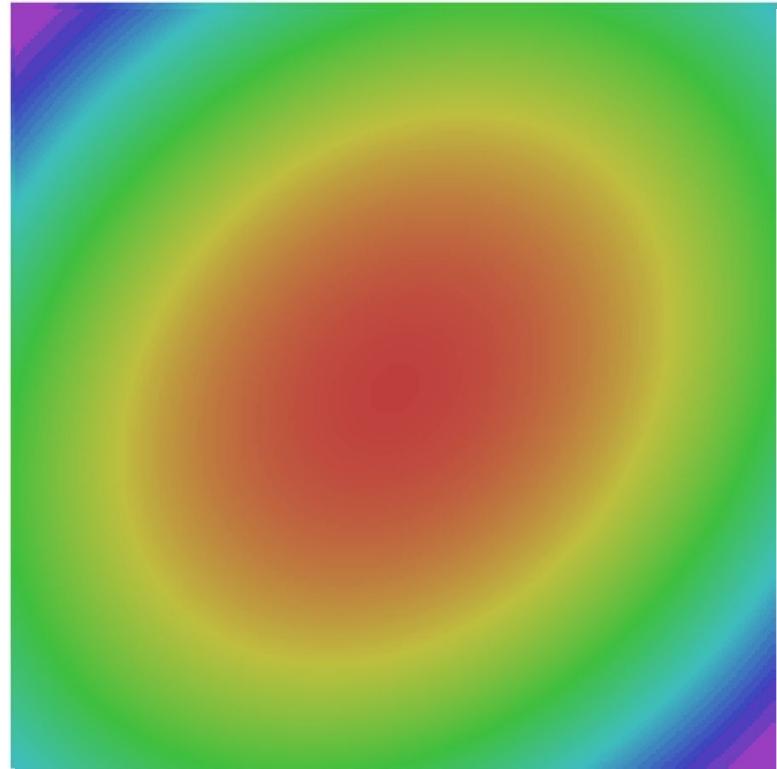
Dauphin et al, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", NIPS 2014

Optimization: Problems with SGD

Our gradients come from
minibatches so they can be noisy!

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

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



SGD + Momentum

SGD

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

```
while True:  
    dx = compute_gradient(x)  
    x -= learning_rate * dx
```

SGD+Momentum

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

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0  
while True:  
    dx = compute_gradient(x)  
    vx = rho * vx + dx  
    x -= learning_rate * vx
```

- 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

SGD + Momentum

SGD+Momentum

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

$$x_{t+1} = x_t + v_{t+1}$$

```
vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx - learning_rate * dx
    x += vx
```

SGD+Momentum

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

$$x_{t+1} = x_t - \alpha v_{t+1}$$

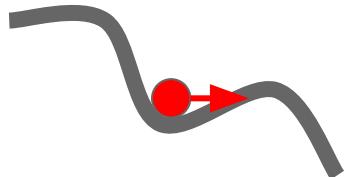
```
vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx + dx
    x -= learning_rate * vx
```

You may see SGD+Momentum formulated different ways,
but they are equivalent - give same sequence of x

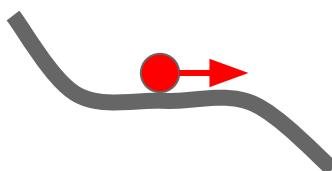
Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

SGD + Momentum

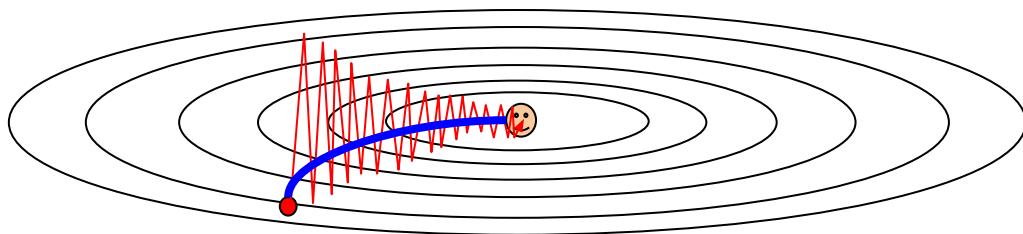
Local Minima



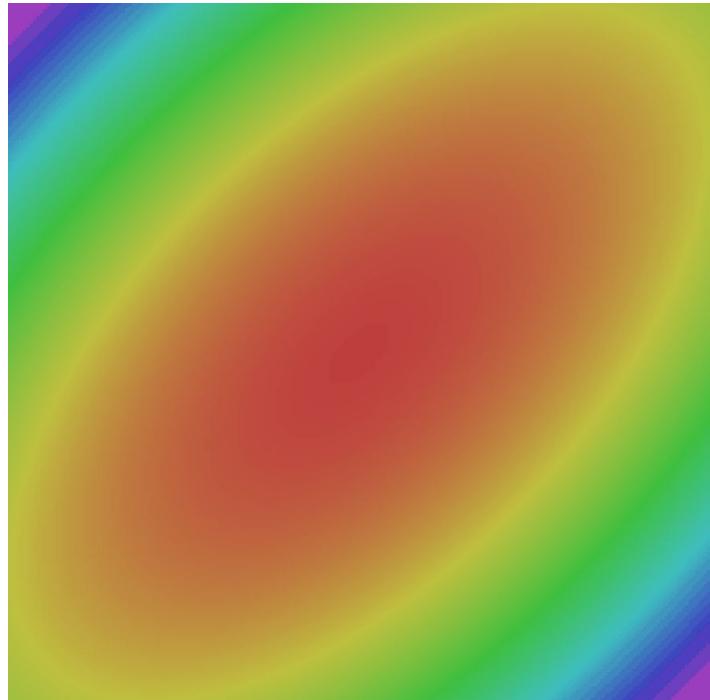
Saddle points



Poor Conditioning



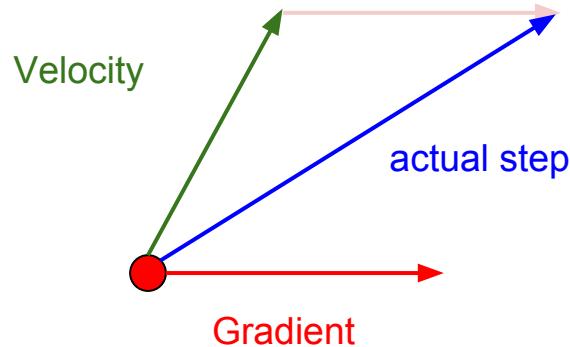
Gradient Noise



— SGD — SGD+Momentum

SGD+Momentum

Momentum update:



Combine gradient at current point with velocity to get step used to update weights

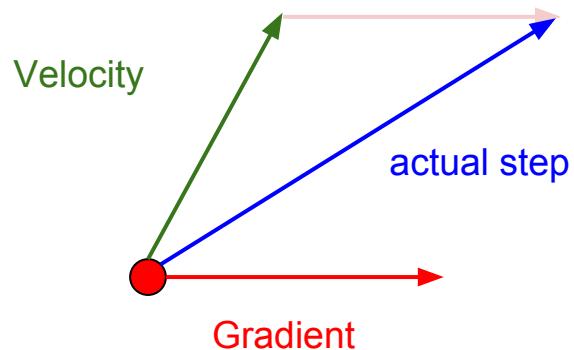
Nesterov, "A method of solving a convex programming problem with convergence rate $O(1/k^2)$ ", 1983

Nesterov, "Introductory lectures on convex optimization: a basic course", 2004

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

Nesterov Momentum

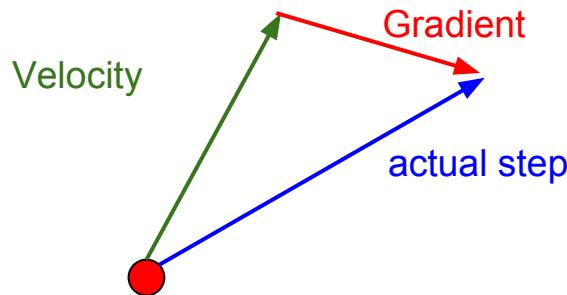
Momentum update:



Combine gradient at current point with velocity to get step used to update weights

Nesterov, "A method of solving a convex programming problem with convergence rate $O(1/k^2)$ ", 1983
Nesterov, "Introductory lectures on convex optimization: a basic course", 2004
Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

Nesterov Momentum

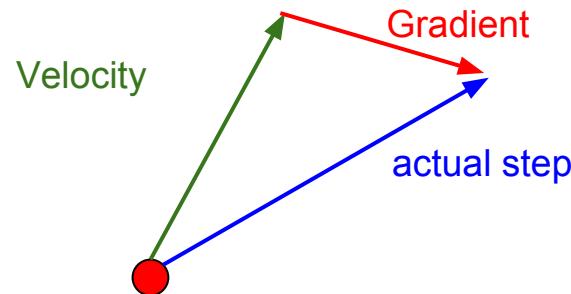


"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov Momentum

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

$$x_{t+1} = x_t + v_{t+1}$$



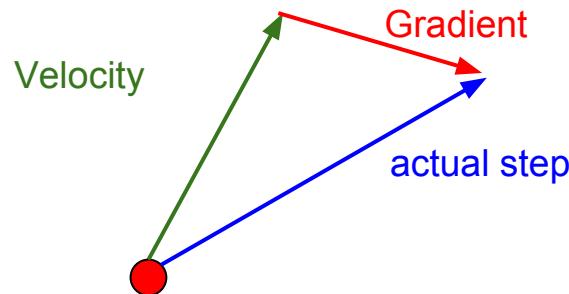
“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov Momentum

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

$$x_{t+1} = x_t + v_{t+1}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$



“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov Momentum

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

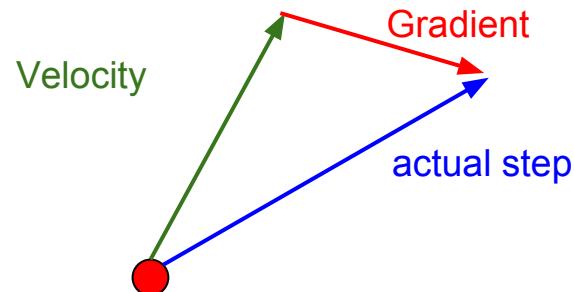
$$x_{t+1} = x_t + v_{t+1}$$

Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$

$$\begin{aligned}\tilde{x}_{t+1} &= \tilde{x}_t - \rho v_t + (1 + \rho)v_{t+1} \\ &= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)\end{aligned}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$



“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov Momentum

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

$$x_{t+1} = x_t + v_{t+1}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

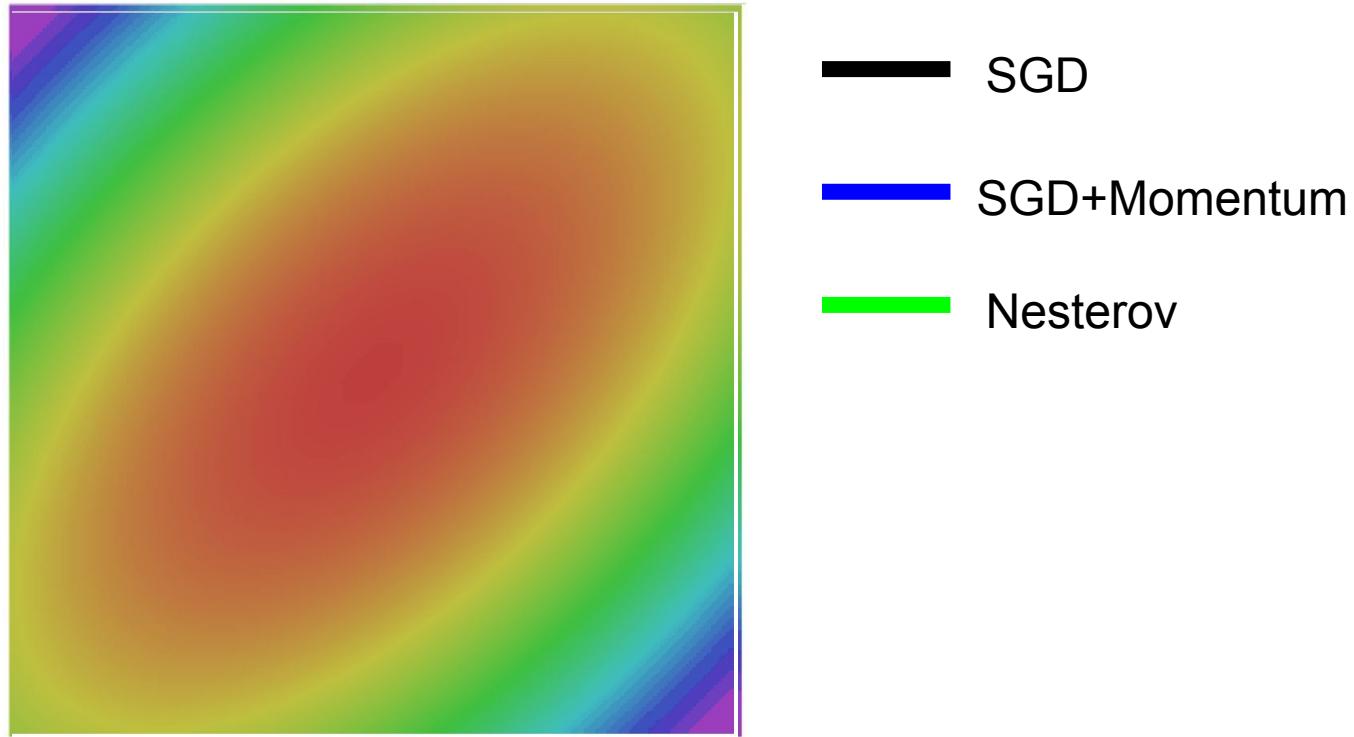
$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$

$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1 + \rho)v_{t+1}$$

$$= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$

```
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```

Nesterov Momentum



AdaGrad

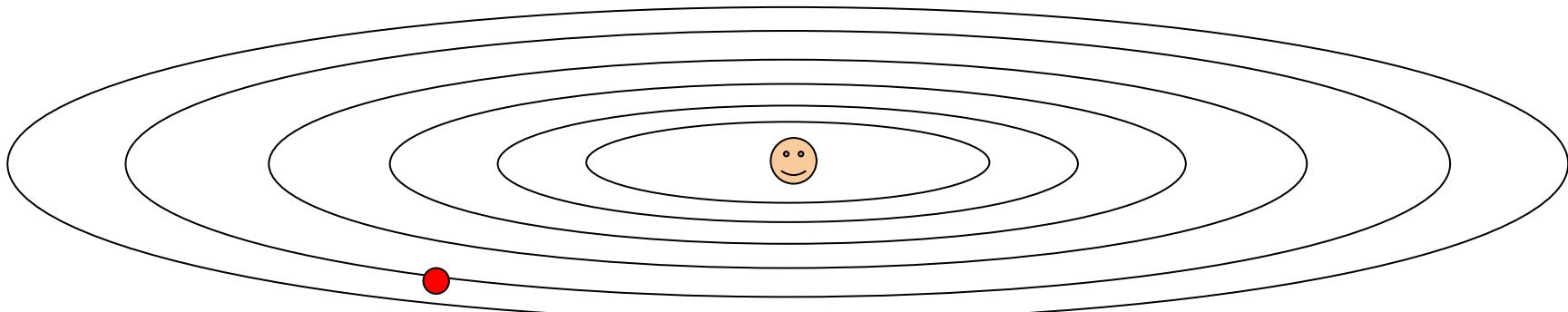
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

“Per-parameter learning rates”
or “adaptive learning rates”

AdaGrad

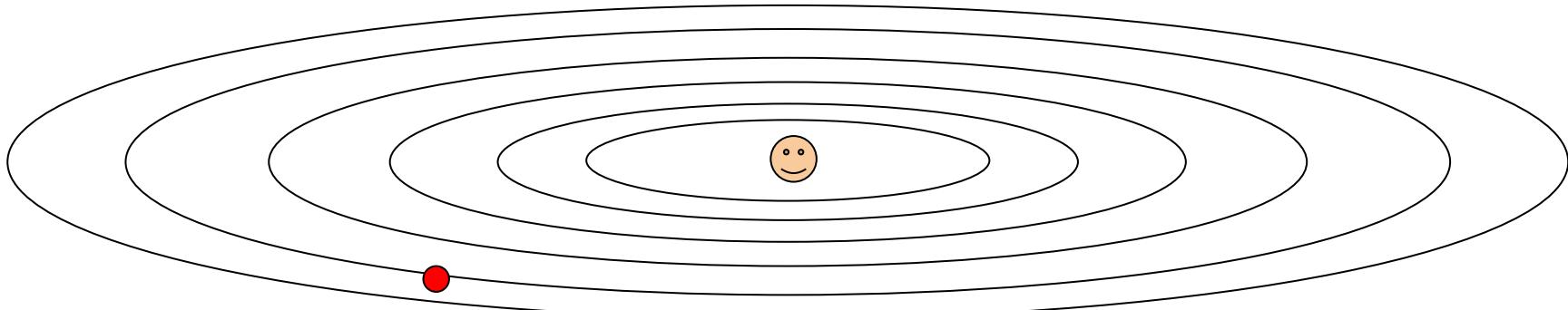
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q: What happens with AdaGrad?

AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

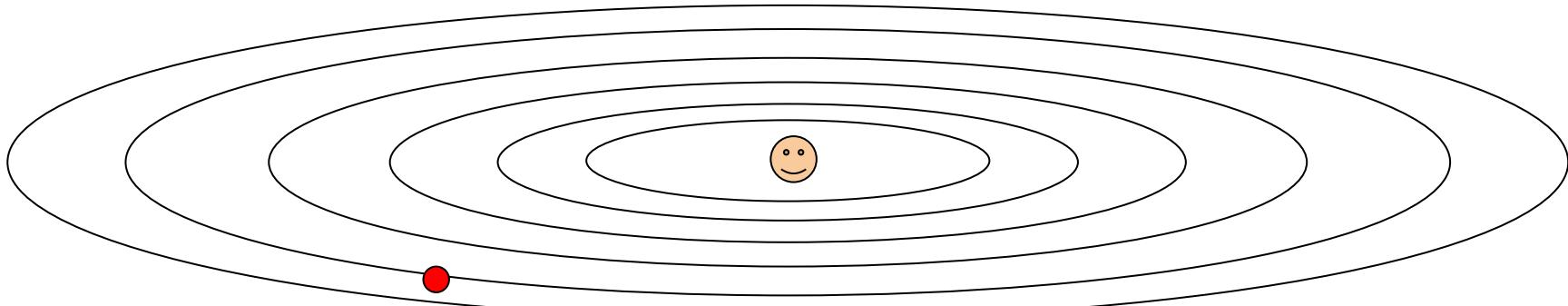


Q: What happens with AdaGrad?

Progress along “steep” directions is damped;
progress along “flat” directions is accelerated

AdaGrad

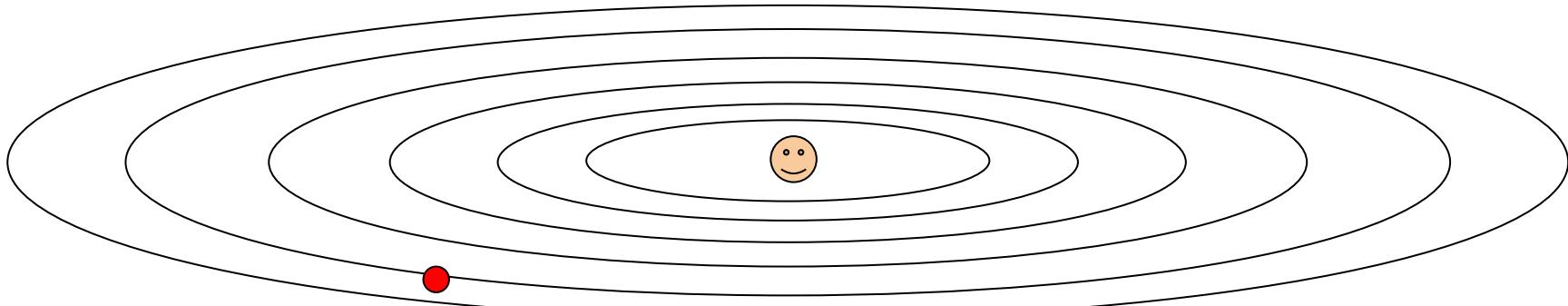
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q2: What happens to the step size over long time?

AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q2: What happens to the step size over long time?

Decays to zero

RMSProp

AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

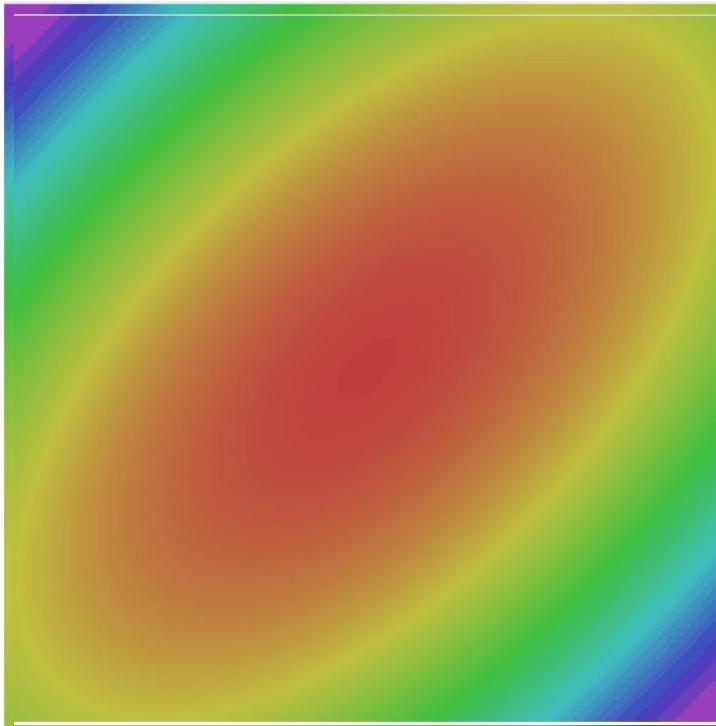


RMSProp

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012

RMSProp



- SGD
- SGD+Momentum
- RMSProp

Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

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

Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

AdaGrad / RMSProp

Sort of like RMSProp with momentum

Q: What happens at first timestep?

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

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Bias correction for the fact that
first and second moment
estimates start at zero

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

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Bias correction for the fact that
first and second moment
estimates start at zero

Adam with $\beta_1 = 0.9$,
 $\beta_2 = 0.999$, and $\text{learning_rate} = 1e-3$ or $5e-4$
is a great starting point for many models!

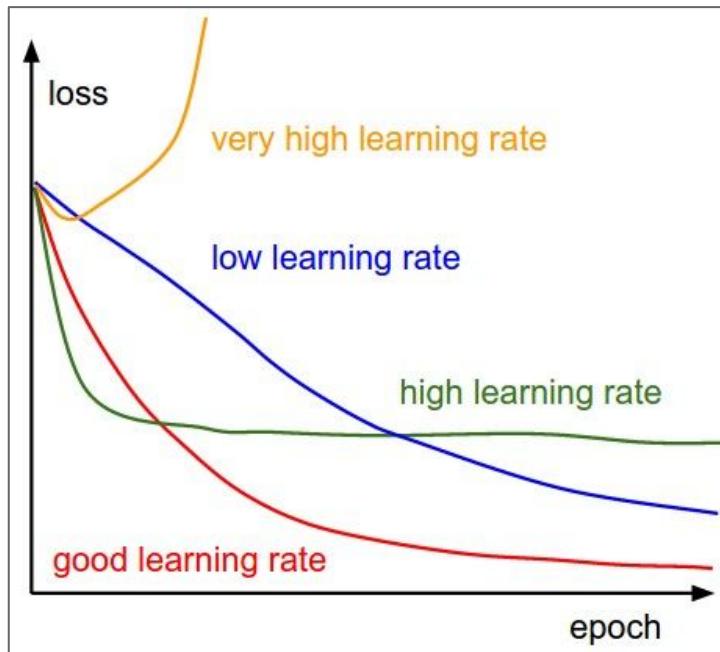
Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Adam



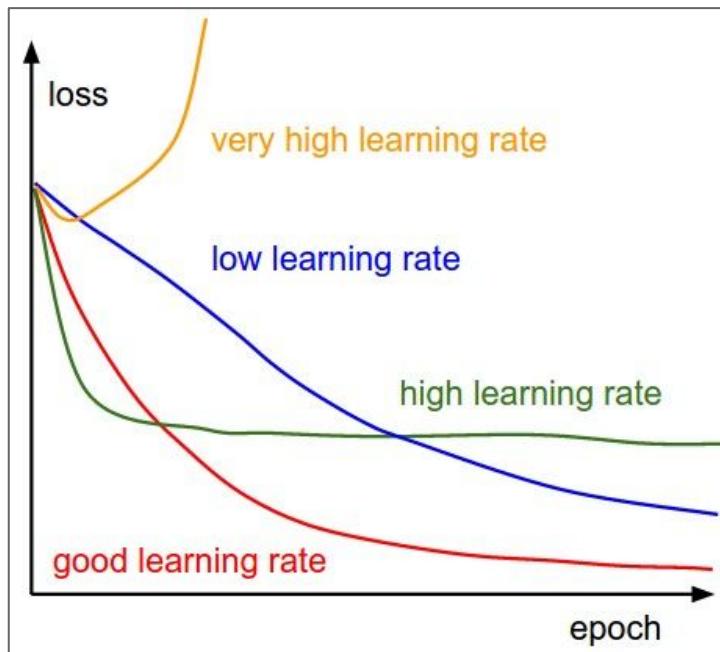
- SGD
- SGD+Momentum
- RMSProp
- Adam

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



=> Learning rate decay over time!

step decay:

e.g. decay learning rate by half every few epochs.

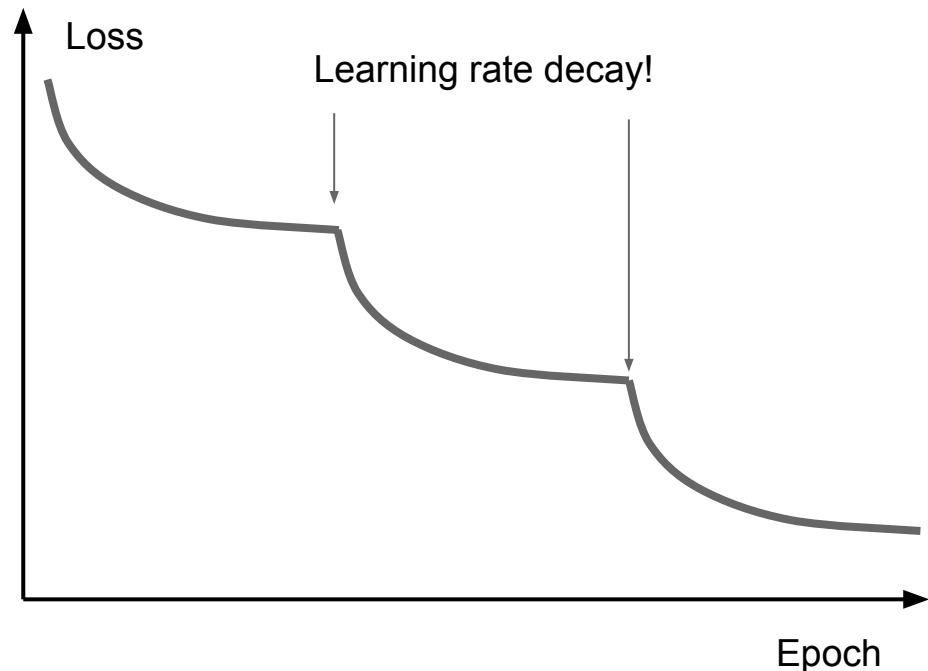
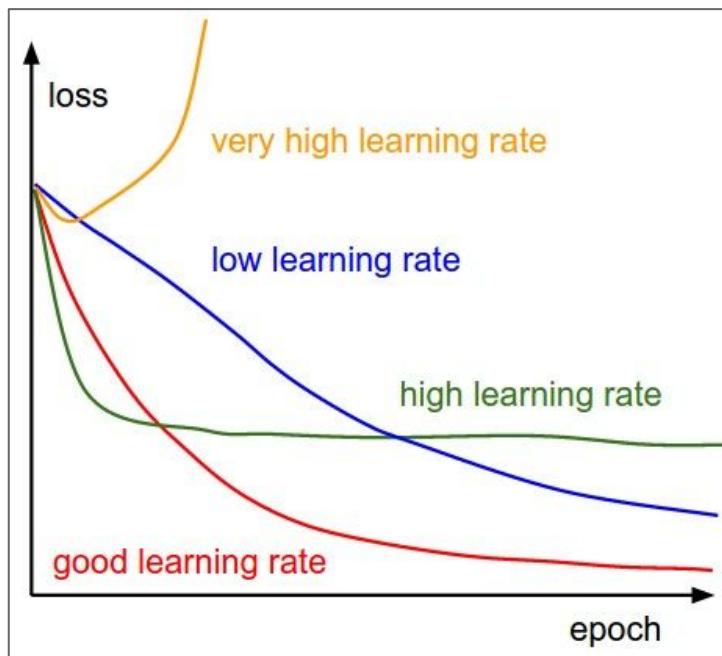
exponential decay:

$$\alpha = \alpha_0 e^{-kt}$$

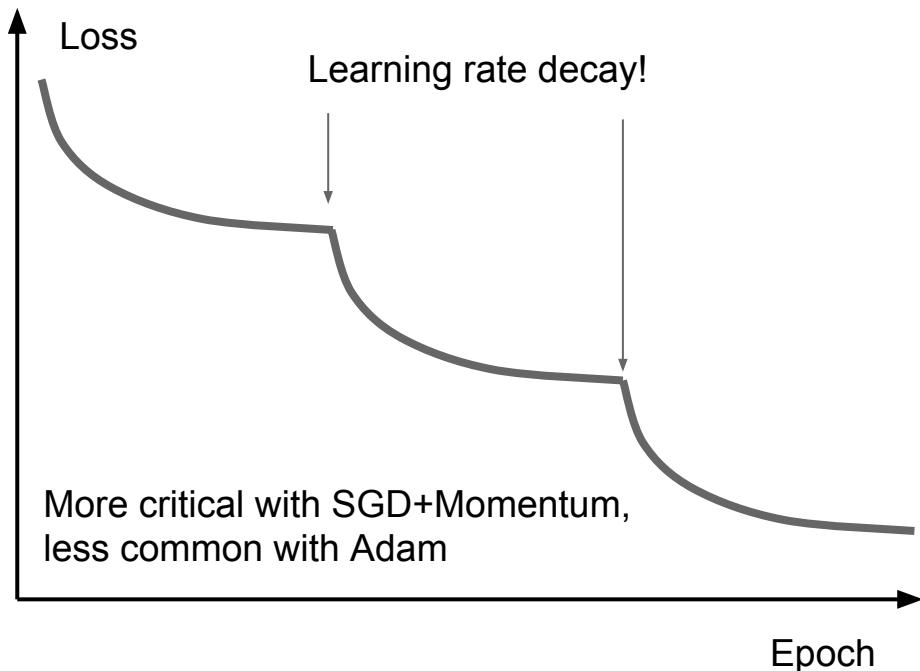
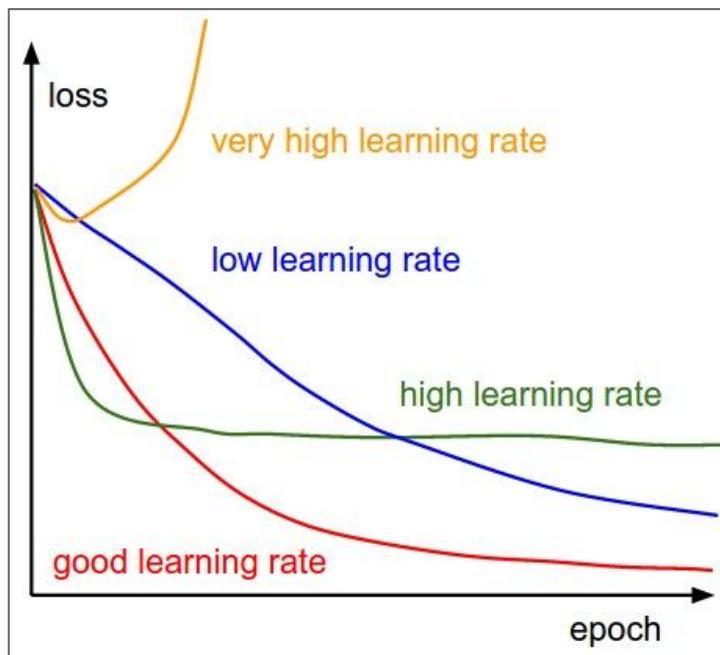
1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

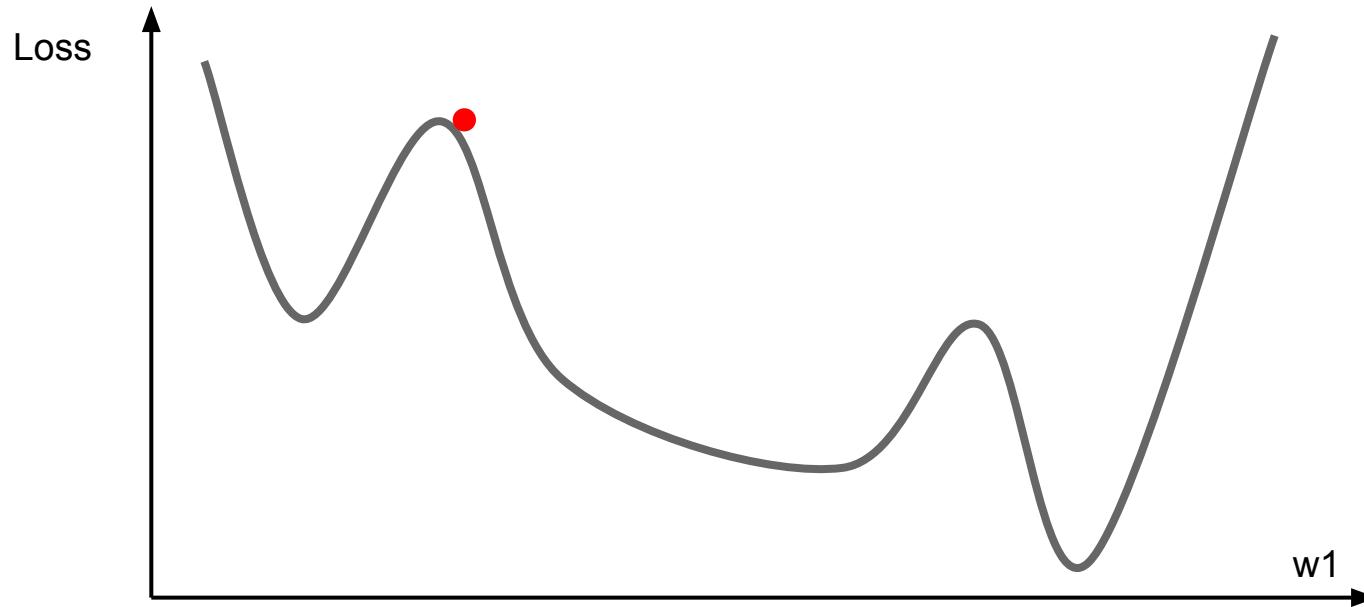
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

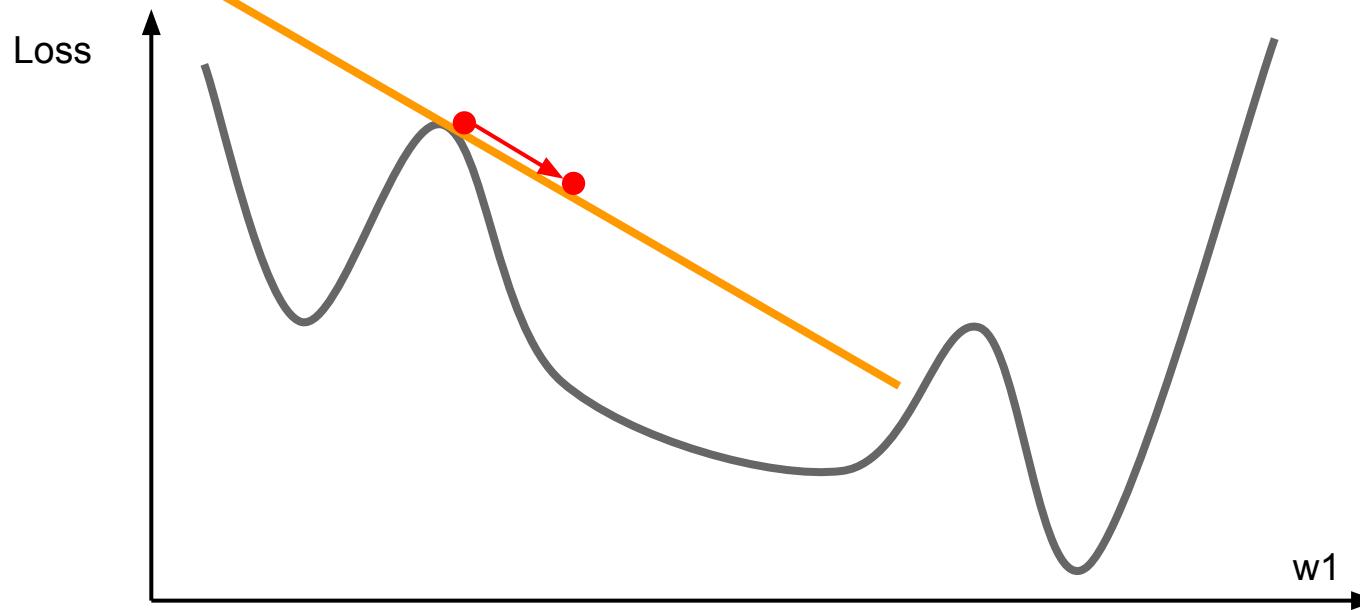


First-Order Optimization



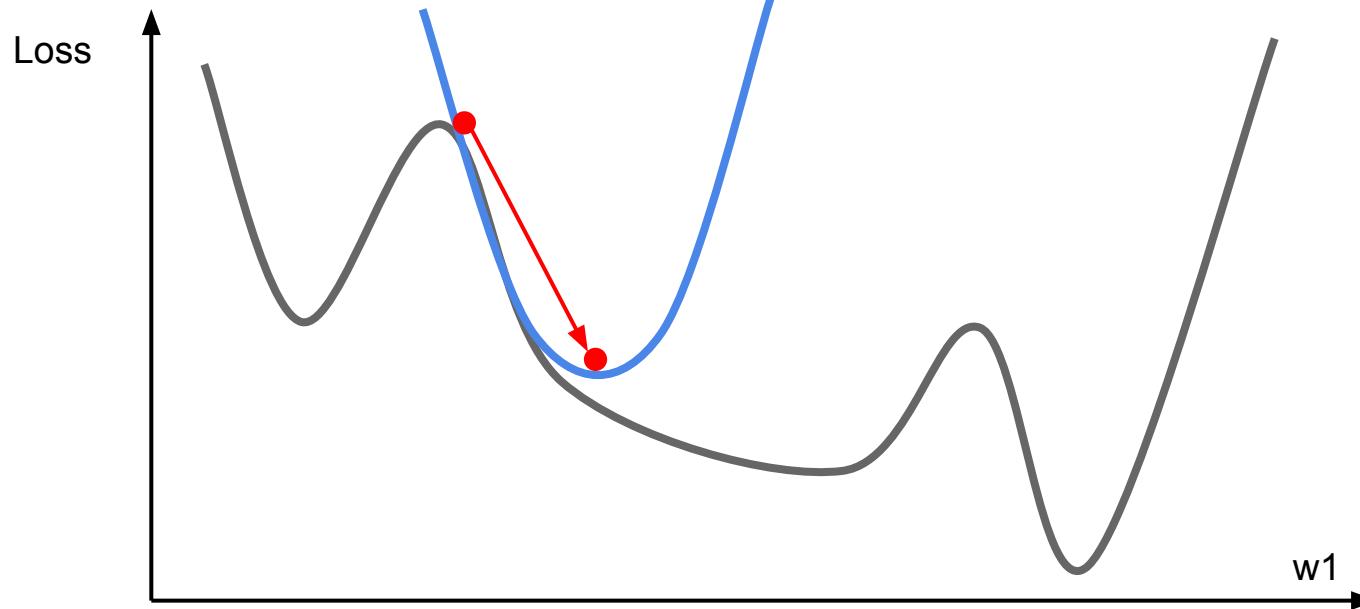
First-Order Optimization

- (1) Use gradient form linear approximation
- (2) Step to minimize the approximation



Second-Order Optimization

- (1) Use gradient **and Hessian** to form **quadratic approximation**
- (2) Step to the **minima** of the approximation



Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Q: What is nice about this update?

Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

No hyperparameters!

No learning rate!

(Though you might use one in practice)

Q: What is nice about this update?

Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Q2: Why is this bad for deep learning?

Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Hessian has $O(N^2)$ elements
Inverting takes $O(N^3)$
 $N = (\text{Tens or Hundreds of}) \text{ Millions}$

Q2: Why is this bad for deep learning?

Second-Order Optimization

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

- Quasi-Newton methods (**BGFS** most popular):
instead of inverting the Hessian ($O(n^3)$), approximate inverse Hessian with rank 1 updates over time ($O(n^2)$ each).
- **L-BFGS** (Limited memory BFGS):
Does not form/store the full inverse Hessian.

L-BFGS

- **Usually works very well in full batch, deterministic mode**
i.e. if you have a single, deterministic $f(x)$ then L-BFGS will probably work very nicely
- **Does not transfer very well to mini-batch setting.** Gives bad results. Adapting second-order methods to large-scale, stochastic setting is an active area of research.

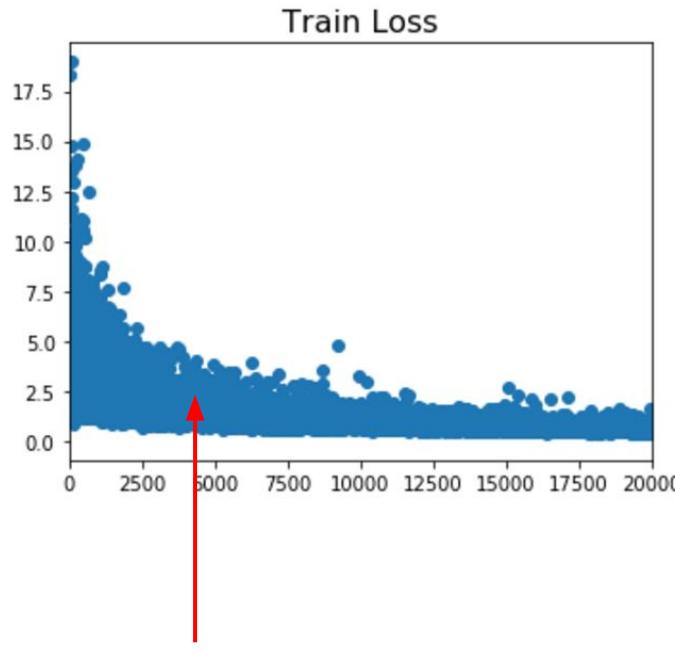
Le et al, "On optimization methods for deep learning, ICML 2011"

Ba et al, "Distributed second-order optimization using Kronecker-factored approximations", ICLR 2017

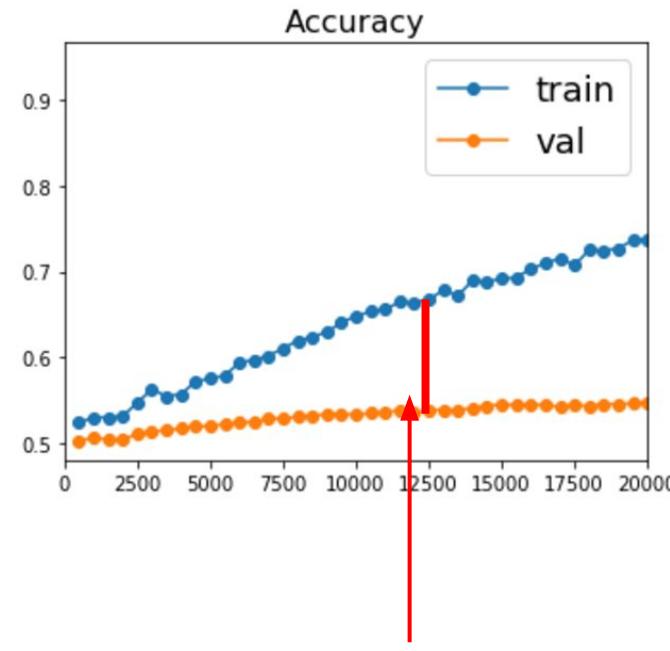
In practice:

- **Adam** is a good default choice in many cases
- **SGD+Momentum** with learning rate decay often outperforms Adam by a bit, but requires more tuning
- If you can afford to do full batch updates then try out **L-BFGS** (and don't forget to disable all sources of noise)

Beyond Training Error

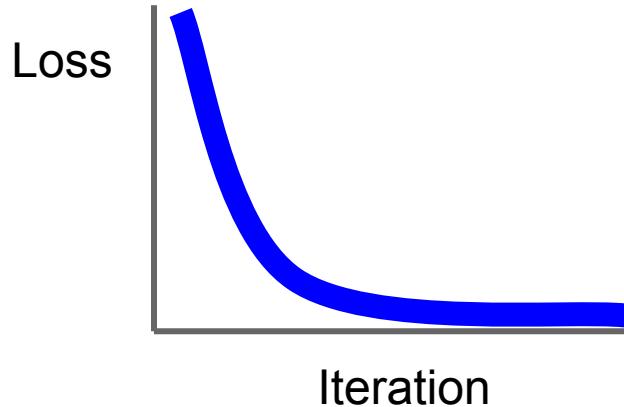


Better optimization algorithms help reduce training loss

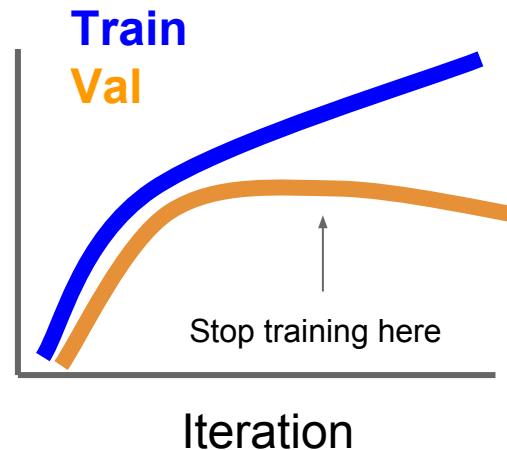


But we really care about error on new data - how to reduce the gap?

Early Stopping



Accuracy



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

Model Ensembles

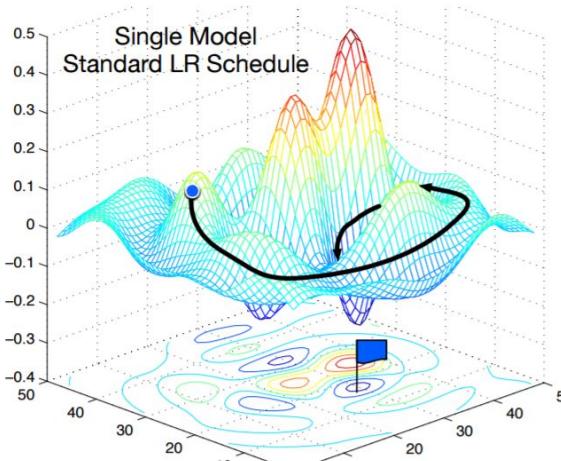
1. Train multiple independent models
2. At test time average their results

(Take average of predicted probability distributions, then choose argmax)

Enjoy 2% extra performance

Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!



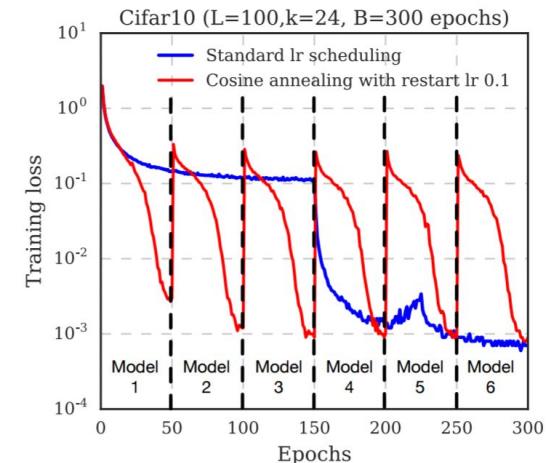
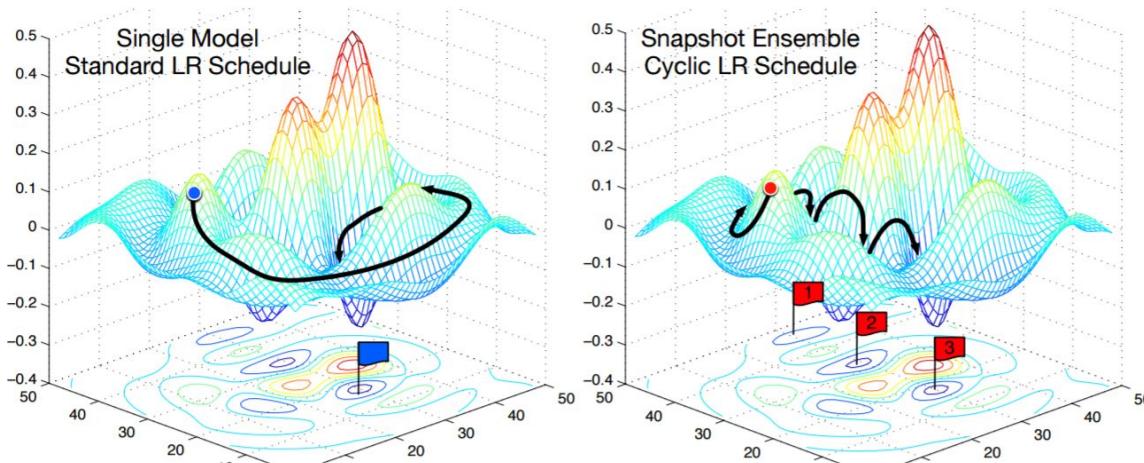
Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016

Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017

Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!



Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016

Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017

Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

Cyclic learning rate schedules can make this work even better!

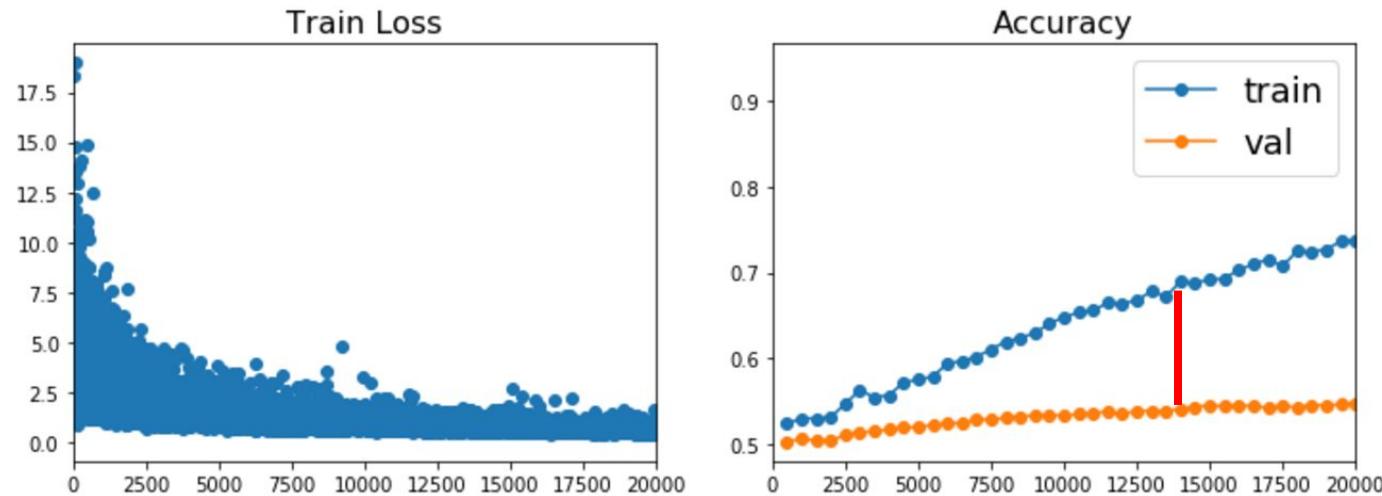
Model Ensembles: Tips and Tricks

Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time (Polyak averaging)

```
while True:  
    data_batch = dataset.sample_data_batch()  
    loss = network.forward(data_batch)  
    dx = network.backward()  
    x += - learning_rate * dx  
    x_test = 0.995*x_test + 0.005*x # use for test set
```

Polyak and Juditsky, "Acceleration of stochastic approximation by averaging", SIAM Journal on Control and Optimization, 1992.

How to improve single-model performance?



Regularization

Regularization: Add term to loss

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use:

L2 regularization

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

L1 regularization

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

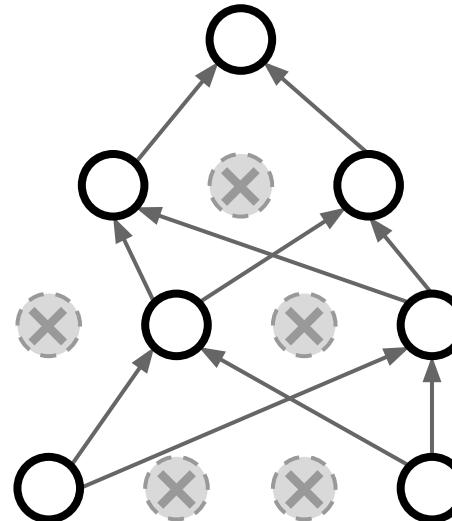
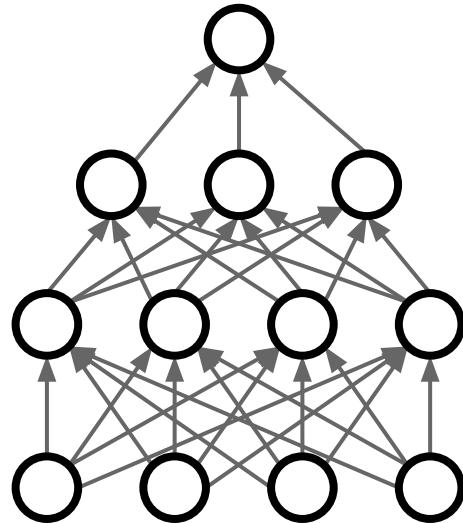
Elastic net (L1 + L2)

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

Regularization: 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

Regularization: Dropout

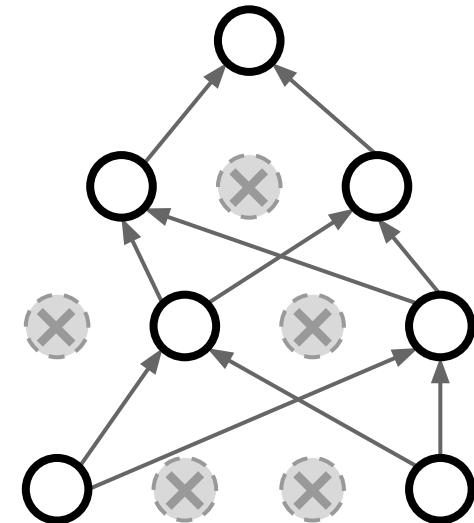
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

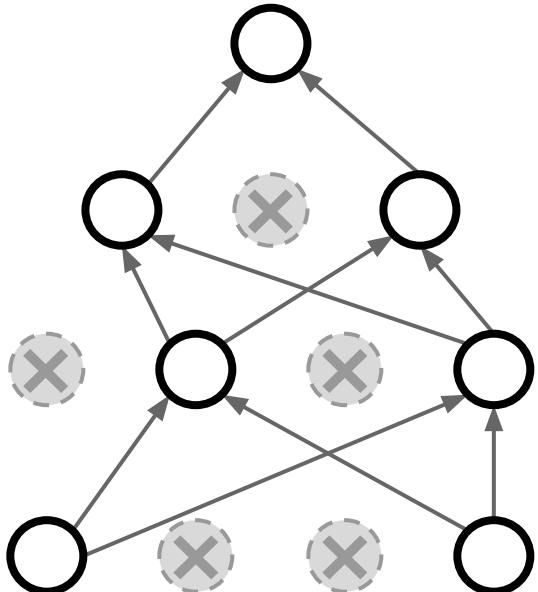
    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout



Regularization: Dropout

How can this possibly be a good idea?

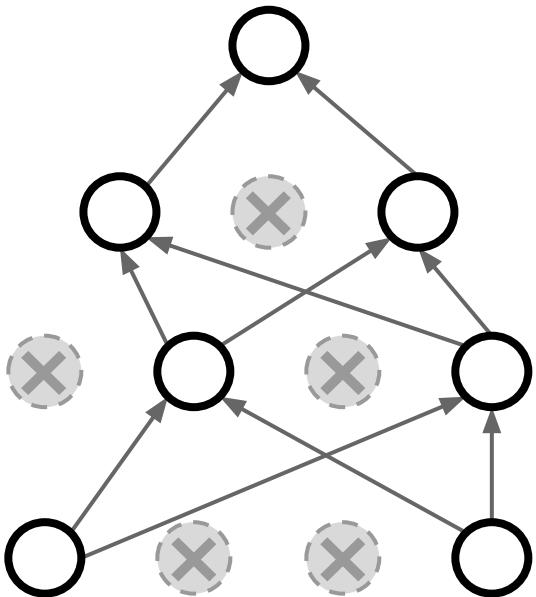


Forces the network to have a redundant representation;
Prevents co-adaptation of features



Regularization: Dropout

How can this possibly be a good idea?



Another interpretation:

Dropout is training a large **ensemble** of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!

Only $\sim 10^{82}$ atoms in the universe...

Dropout: Test time

Dropout makes our output random!

$$\boxed{y} = f_W(\boxed{x}, \boxed{z})$$

Output
(label) Input
(image) Random
mask

Want to “average out” the randomness at test-time

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

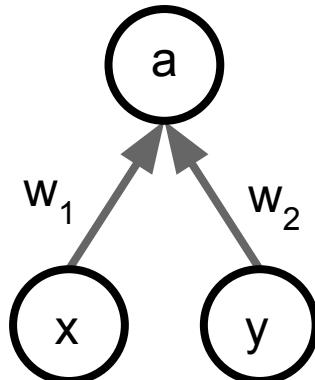
But this integral seems hard ...

Dropout: Test time

Want to approximate
the integral

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Consider a single neuron.

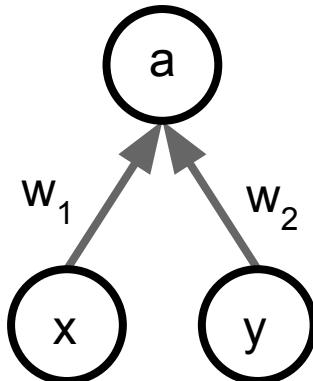


Dropout: Test time

Want to approximate
the integral

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Consider a single neuron.

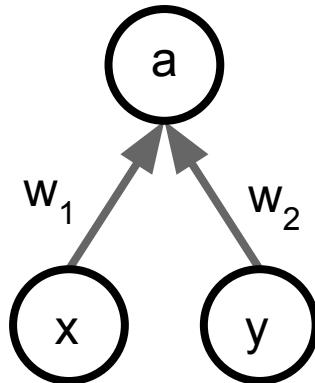


At test time we have: $E[a] = w_1x + w_2y$

Dropout: Test time

Want to approximate
the integral

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$



Consider a single neuron.

At test time we have: $E[a] = w_1x + w_2y$

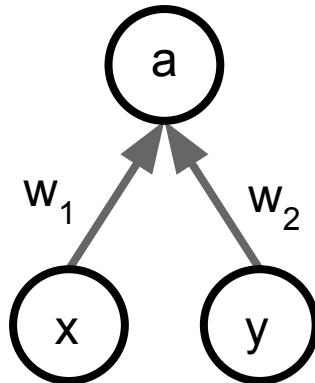
During training we have:

$$\begin{aligned} E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ &= \frac{1}{2}(w_1x + w_2y) \end{aligned}$$

Dropout: Test time

Want to approximate
the integral

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$



Consider a single neuron.

At test time we have: $E[a] = w_1x + w_2y$

During training we have:

$$\begin{aligned} E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ &= \frac{1}{2}(w_1x + w_2y) \end{aligned}$$

At test time, multiply
by dropout probability

Dropout: Test time

```
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

At test time all neurons are active always

=> We must scale the activations so that for each neuron:

output at test time = expected output at training time

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """
```

```
p = 0.5 # probability of keeping a unit active. higher = less dropout
```

```
def train_step(X):  
    """ X contains the data """
```

```
# forward pass for example 3-layer neural network
```

```
H1 = np.maximum(0, np.dot(W1, X) + b1)
```

```
U1 = np.random.rand(*H1.shape) < p # first dropout mask
```

```
H1 *= U1 # drop!
```

```
H2 = np.maximum(0, np.dot(W2, H1) + b2)
```

```
U2 = np.random.rand(*H2.shape) < p # second dropout mask
```

```
H2 *= U2 # drop!
```

```
out = np.dot(W3, H2) + b3
```

```
# backward pass: compute gradients... (not shown)
```

```
# perform parameter update... (not shown)
```

```
def predict(X):
```

```
# ensembled forward pass
```

```
H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
```

```
H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
```

```
out = np.dot(W3, H2) + b3
```

Dropout Summary

drop in forward pass

scale at test time

More common: “Inverted dropout”

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```

test time is unchanged!



Regularization: A common pattern

Training: Add some kind
of randomness

$$y = f_W(x, z)$$

Testing: Average out randomness
(sometimes approximate)

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Regularization: A common pattern

Training: Add some kind of randomness

$$y = f_W(x, z)$$

Testing: Average out randomness (sometimes approximate)

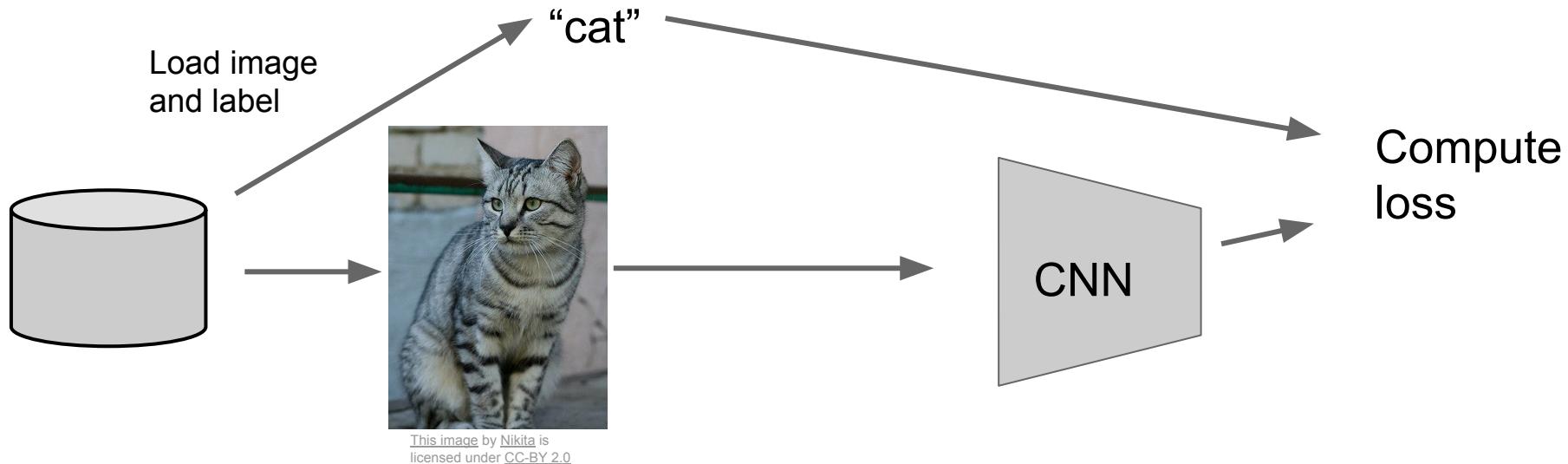
$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Example: Batch Normalization

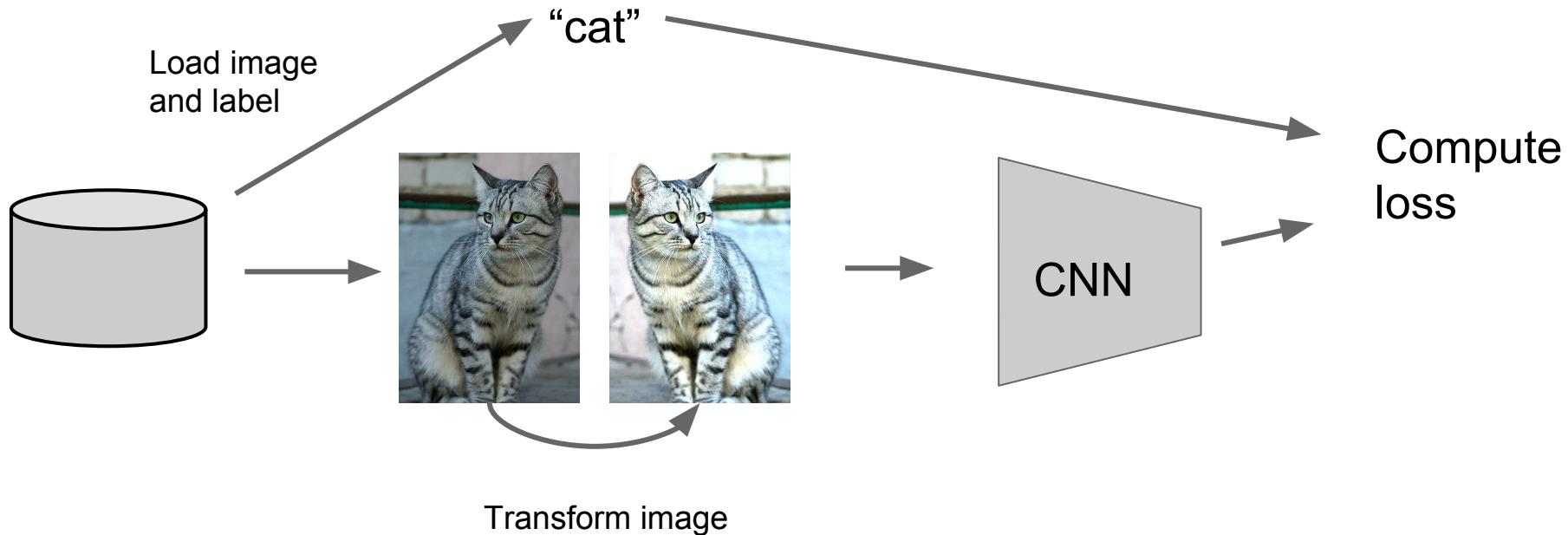
Training: Normalize using stats from random minibatches

Testing: Use fixed stats to normalize

Regularization: Data Augmentation



Regularization: Data Augmentation



Data Augmentation

Horizontal Flips



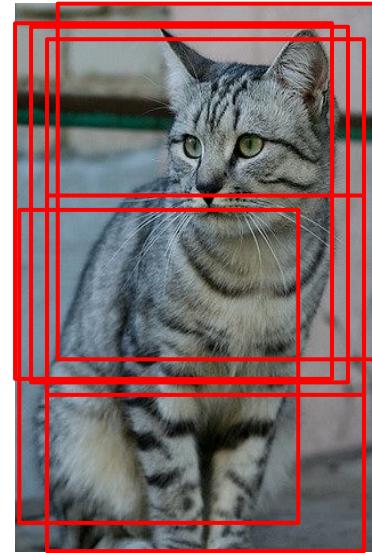
Data Augmentation

Random crops and scales

Training: sample random crops / scales

ResNet:

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



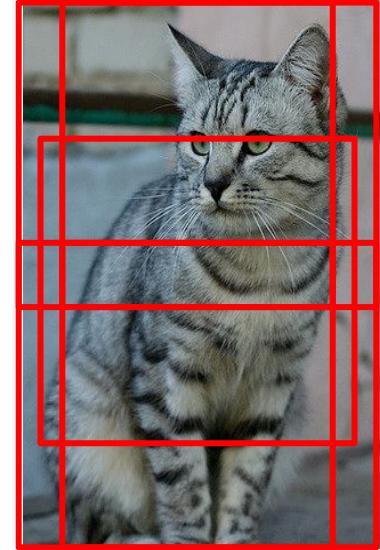
Data Augmentation

Random crops and scales

Training: sample random crops / scales

ResNet:

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



Testing: average a fixed set of crops

ResNet:

1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224×224 crops: 4 corners + center, + flips

Data Augmentation

Color Jitter

Simple: Randomize
contrast and brightness



Data Augmentation

Color Jitter

Simple: Randomize
contrast and brightness



More Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a “color offset” along principal component directions
3. Add offset to all pixels of a training image

(As seen in *[Krizhevsky et al. 2012]*, ResNet, etc)

Data Augmentation

Get creative for your problem!

Random mix/combinations of :

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

Regularization: A common pattern

Training: Add random noise

Testing: Marginalize over the noise

Examples:

Dropout

Batch Normalization

Data Augmentation

Regularization: A common pattern

Training: Add random noise

Testing: Marginalize over the noise

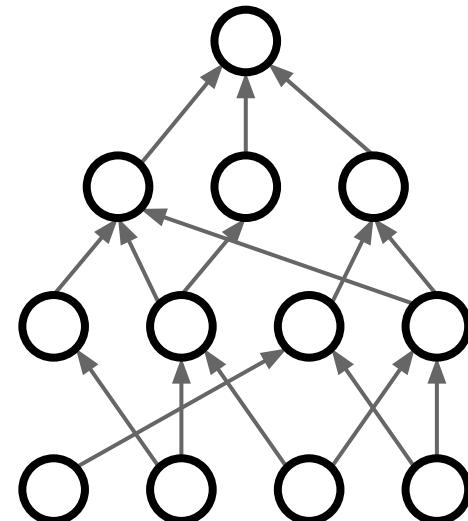
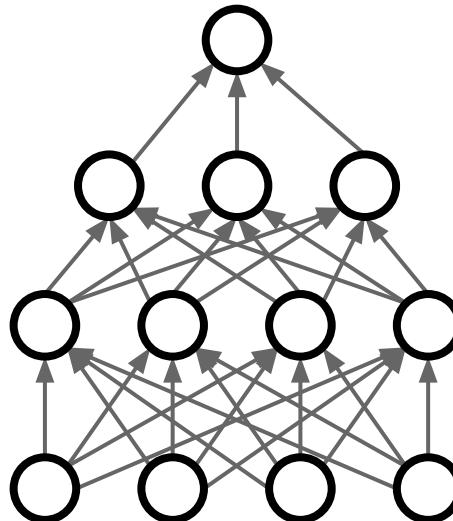
Examples:

Dropout

Batch Normalization

Data Augmentation

DropConnect



Wan et al, "Regularization of Neural Networks using DropConnect", ICML 2013

Regularization: A common pattern

Training: Add random noise

Testing: Marginalize over the noise

Examples:

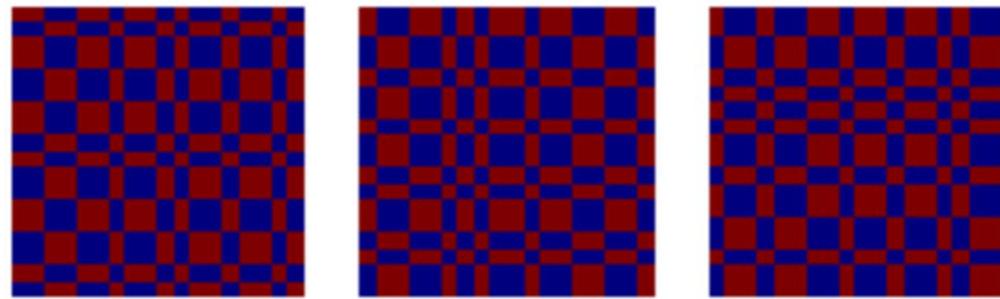
Dropout

Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling



Graham, "Fractional Max Pooling", arXiv 2014

Regularization: A common pattern

Training: Add random noise

Testing: Marginalize over the noise

Examples:

Dropout

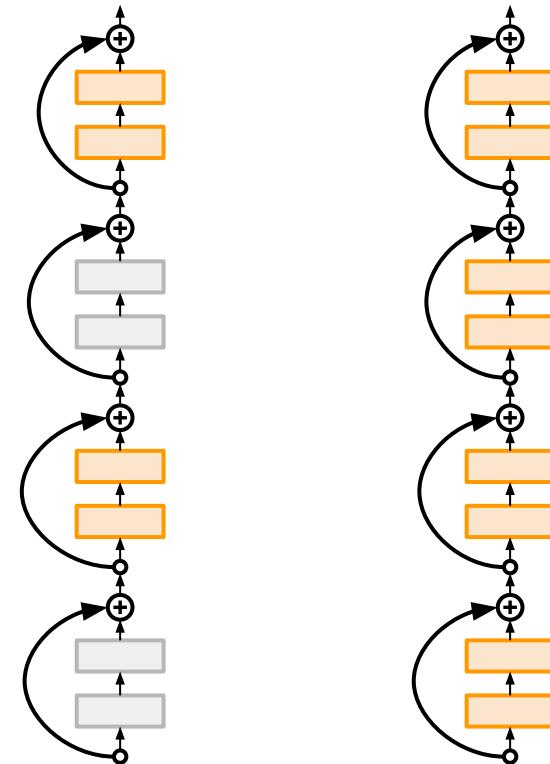
Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth



Huang et al, "Deep Networks with Stochastic Depth", ECCV 2016

Transfer Learning

“You need a lot of data if you want to
train/use CNNs”

Transfer Learning

“You need a lot of data if you want to
train/use CNNs”

BUSTED

Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

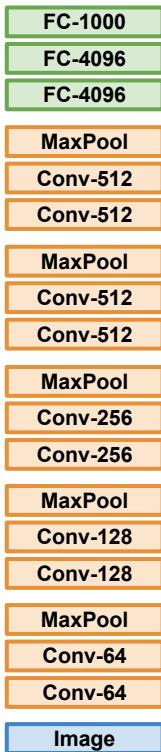
1. Train on Imagenet



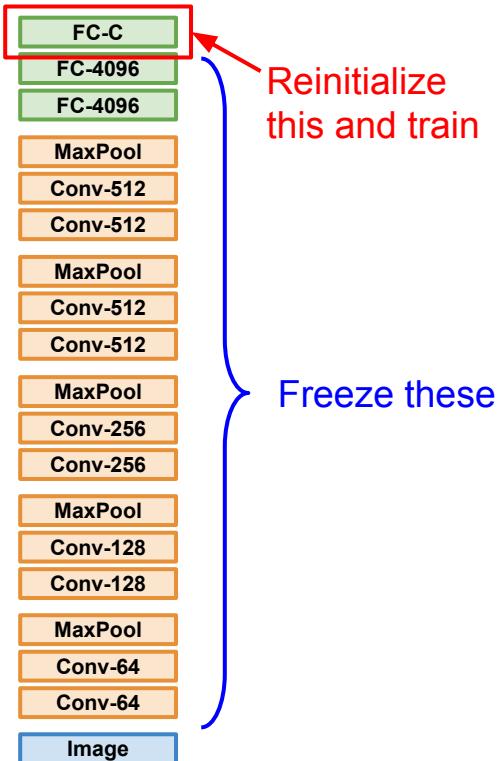
Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

1. Train on Imagenet



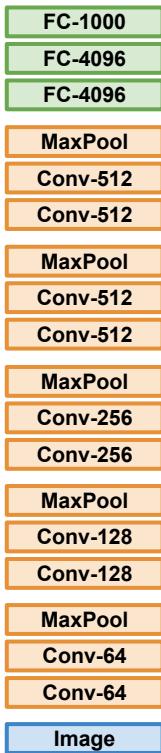
2. Small Dataset (C classes)



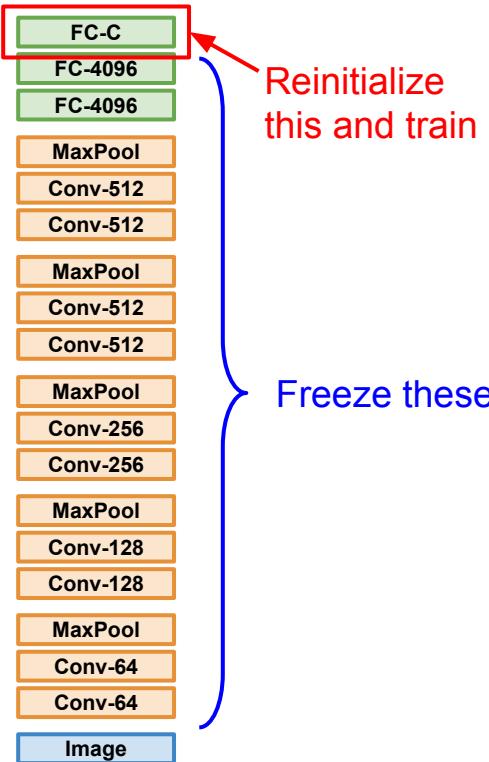
Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

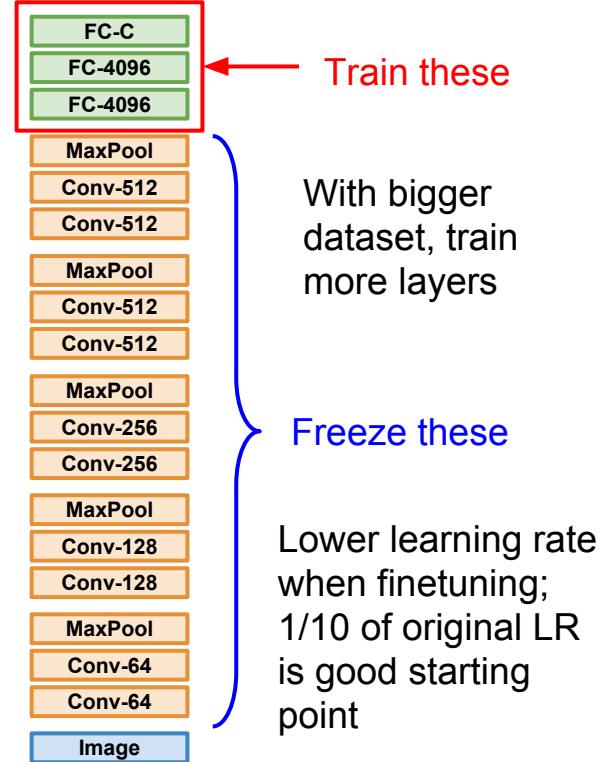
1. Train on Imagenet

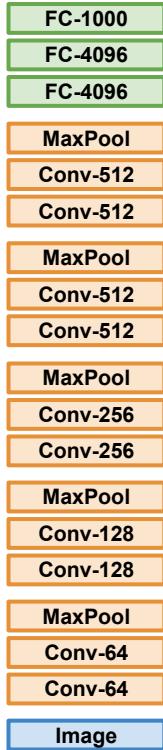


2. Small Dataset (C classes)

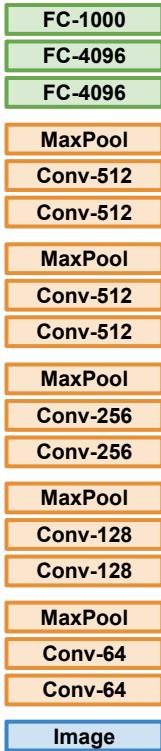


3. Bigger dataset

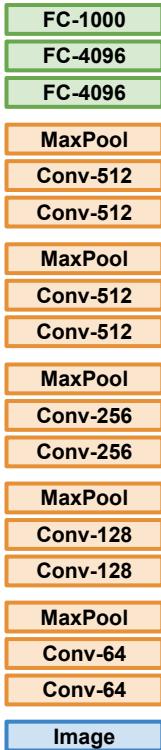




	very similar dataset	very different dataset
very little data	?	?
quite a lot of data	?	?



	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	?
quite a lot of data	Finetune a few layers	?



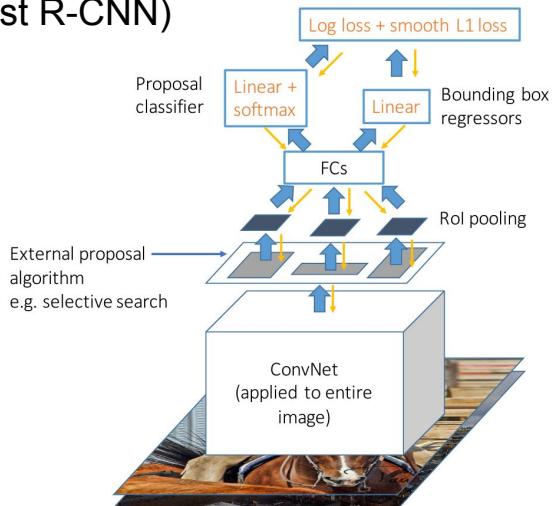
More specific

More generic

	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	You're in trouble... Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

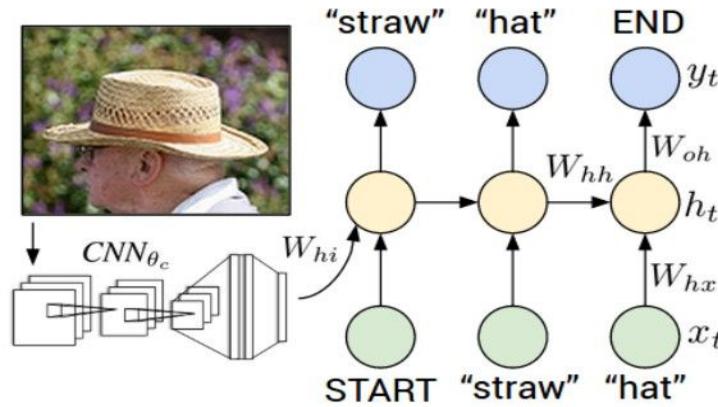
Transfer learning with CNNs is pervasive... (it's the norm, not an exception)

Object Detection
(Fast R-CNN)



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

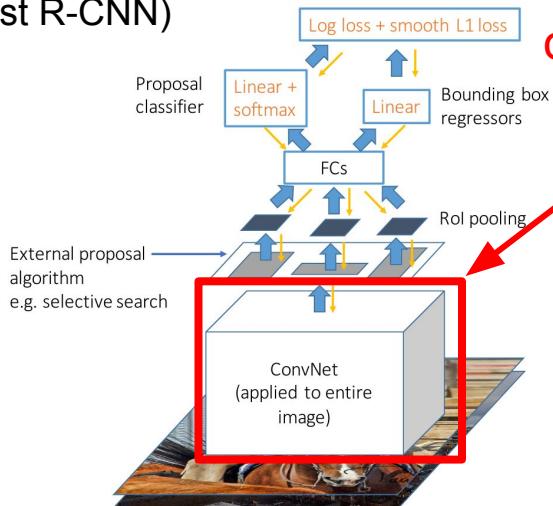
Image Captioning: CNN + RNN



Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

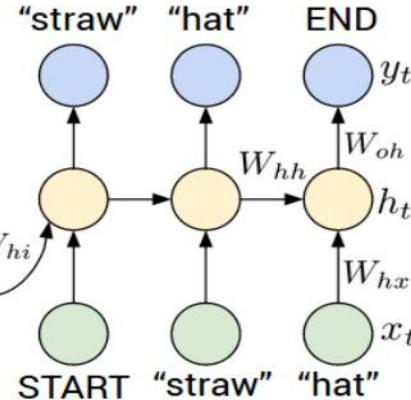
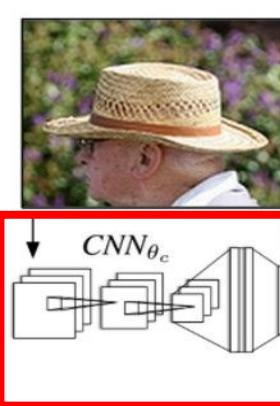
Transfer learning with CNNs is pervasive... (it's the norm, not an exception)

Object Detection
(Fast R-CNN)



CNN pretrained
on ImageNet

Image Captioning: CNN + RNN

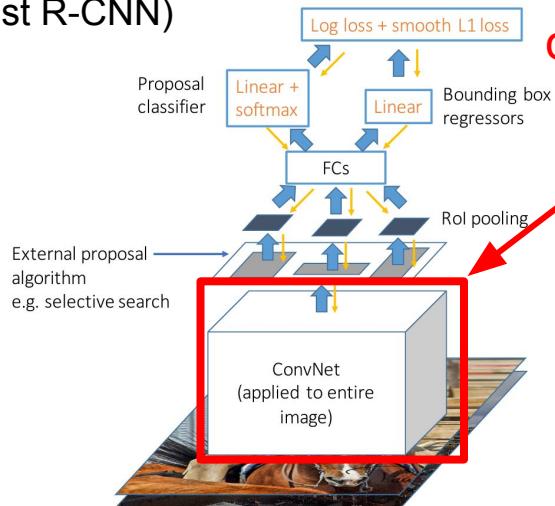


Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

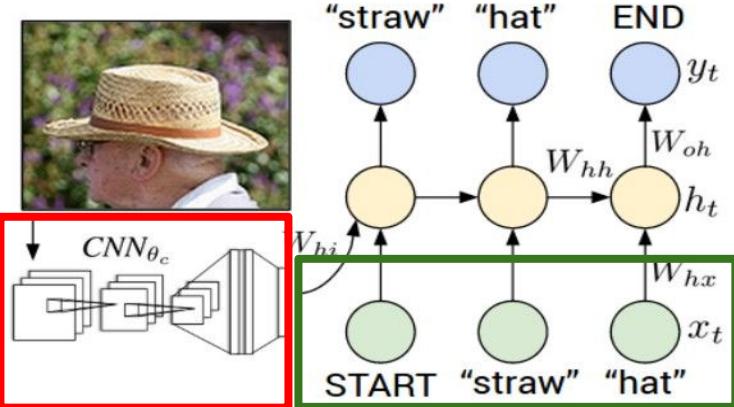
Transfer learning with CNNs is pervasive... (it's the norm, not an exception)

Object Detection
(Fast R-CNN)



CNN pretrained
on ImageNet

Image Captioning: CNN + RNN



Word vectors pretrained
with word2vec

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

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

Takeaway for your projects and beyond:

Have some dataset of interest but it has < ~1M images?

1. Find a very large dataset that has similar data, train a big ConvNet there
2. Transfer learn to your dataset

Deep learning frameworks provide a “Model Zoo” of pretrained models so you don’t need to train your own

Caffe: <https://github.com/BVLC/caffe/wiki/Model-Zoo>

TensorFlow: <https://github.com/tensorflow/models>

PyTorch: <https://github.com/pytorch/vision>

Summary

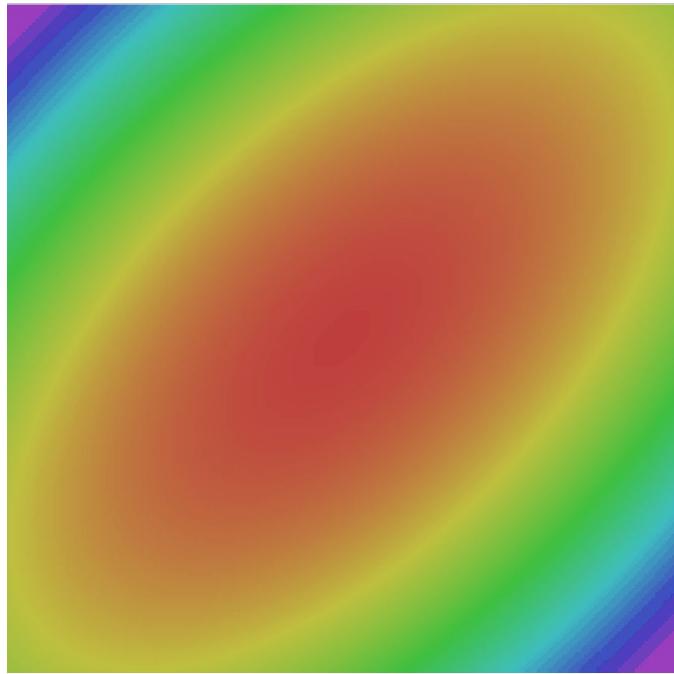
- Lots of Batch Normalization variants
- Optimization
 - Momentum, RMSProp, Adam, etc
- Regularization
 - Dropout, etc
- Transfer learning
 - Use this for your projects!

Next time: Deep Learning Software!

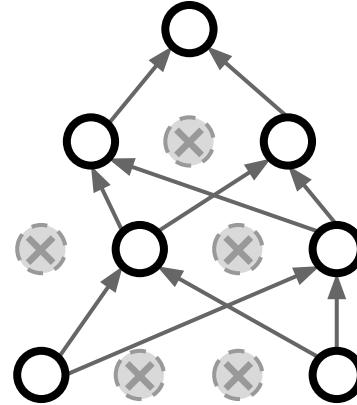
Lecture 8: Hardware and Software

Last time

Optimization: SGD+Momentum,
Nesterov, RMSProp, Adam



Regularization: Dropout

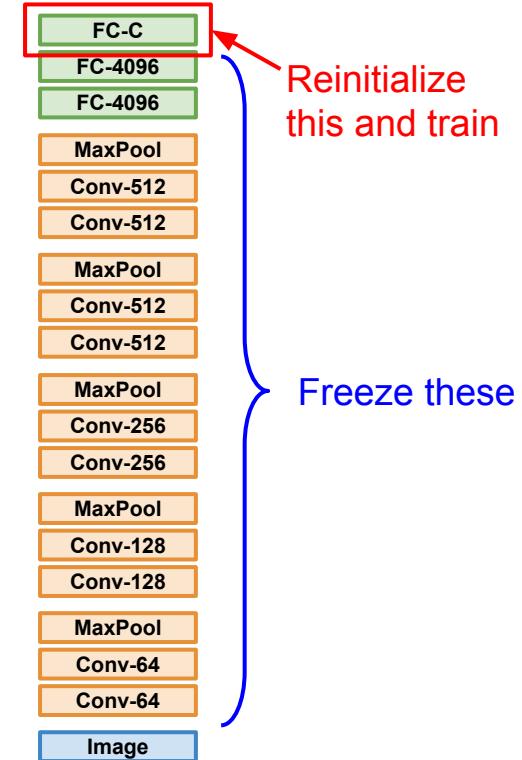


Regularization: Add noise, then
marginalize out

$$\text{Train} \quad y = f_W(x, z)$$

$$\text{Test} \quad y = f(x) = E_z[f(x, z)]$$

**Transfer
Learning**



Today

- Deep learning hardware
 - CPU, GPU, TPU
- Deep learning software
 - PyTorch and TensorFlow
 - Static vs Dynamic computation graphs

Deep Learning Hardware

My computer

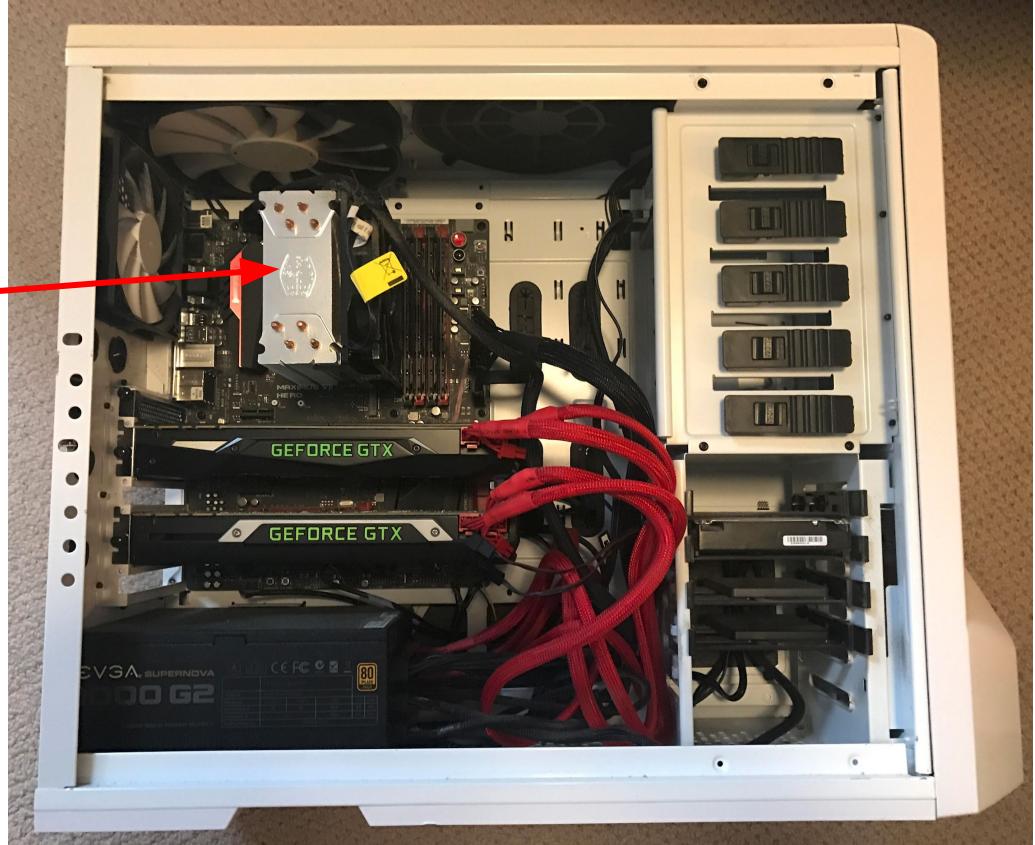


Spot the CPU!

(central processing unit)



[This image is licensed under CC-BY 2.0](#)



Spot the GPUs!

(graphics processing unit)



This image is in the public domain



NVIDIA

vs

AMD

NVIDIA

vs

AMD

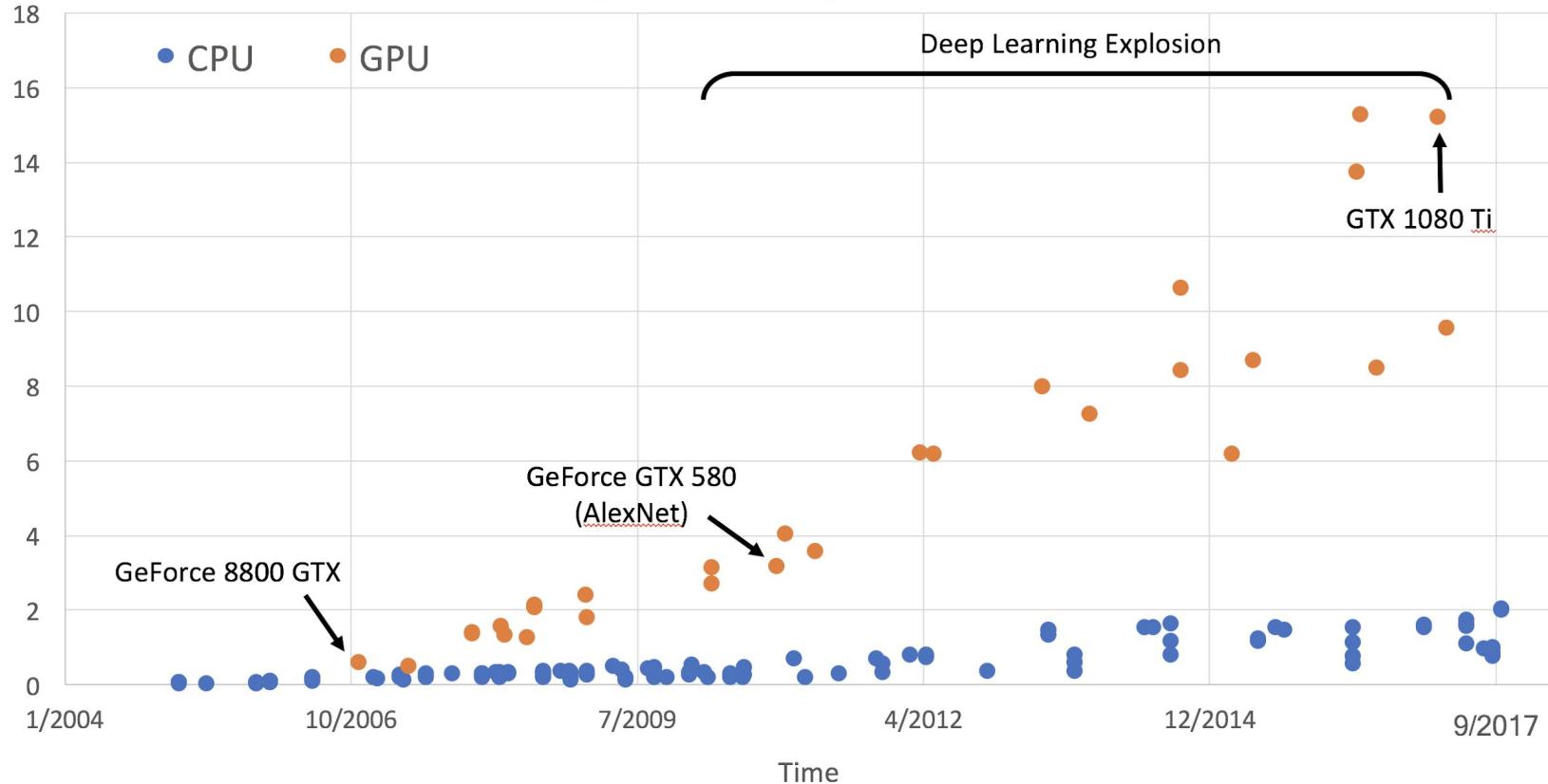
CPU vs GPU

	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	\$339	~540 GFLOPs FP32
GPU (NVIDIA GTX 1080 Ti)	3584	1.6 GHz	11 GB GDDR5 X	\$699	~11.4 TFLOPs FP32

CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

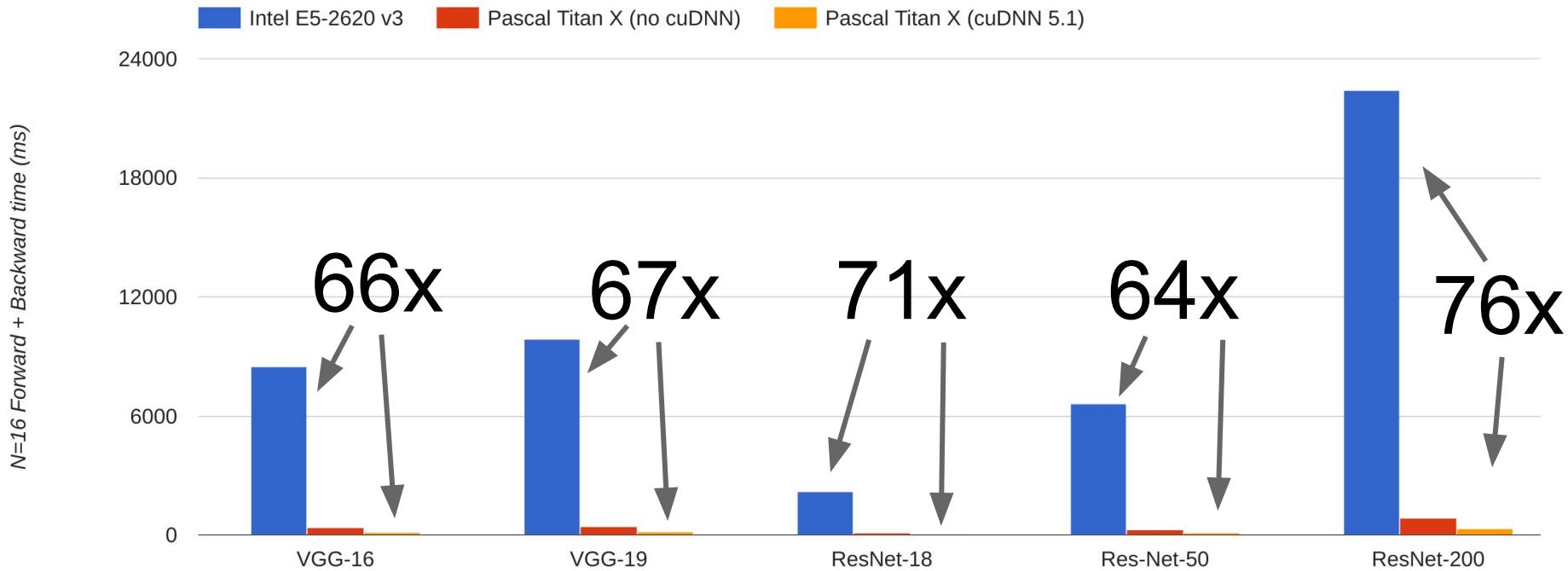
GPU: More cores, but each core is much slower and “dumber”; great for parallel tasks

GigaFLOPs per Dollar



CPU vs GPU in practice

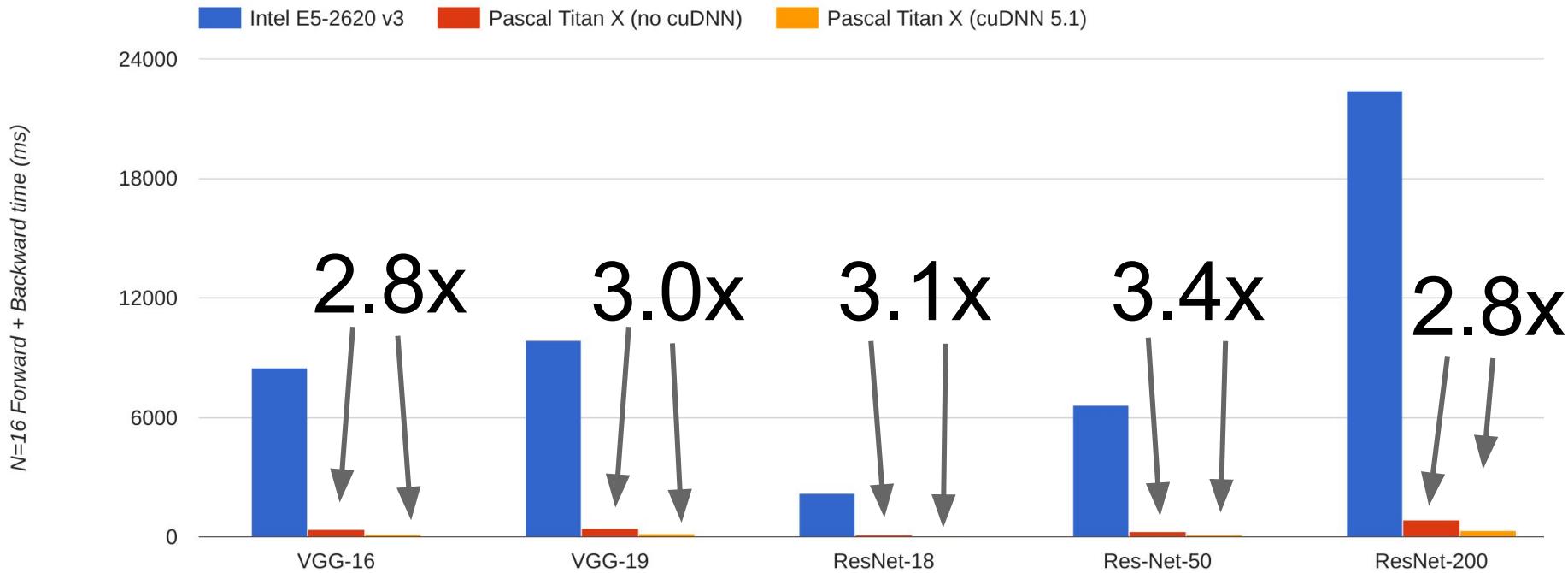
(CPU performance not well-optimized, a little unfair)



Data from <https://github.com/jcjohnson/cnn-benchmarks>

CPU vs GPU in practice

cuDNN much faster than
“unoptimized” CUDA



Data from <https://github.com/jcjohnson/cnn-benchmarks>

CPU vs GPU

	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	\$339	~540 GFLOPs FP32
GPU (NVIDIA GTX 1080 Ti)	3584	1.6 GHz	11 GB GDDR5 X	\$699	~11.4 TFLOPs FP32
TPU NVIDIA TITAN V	5120 CUDA, 640 Tensor	1.5 GHz	12GB HBM2	\$2999	~14 TFLOPs FP32 ~112 TFLOP FP16
TPU Google Cloud TPU	?	?	64 GB HBM	\$6.50 per hour	~180 TFLOP

CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

GPU: More cores, but each core is much slower and “dumber”; great for parallel tasks

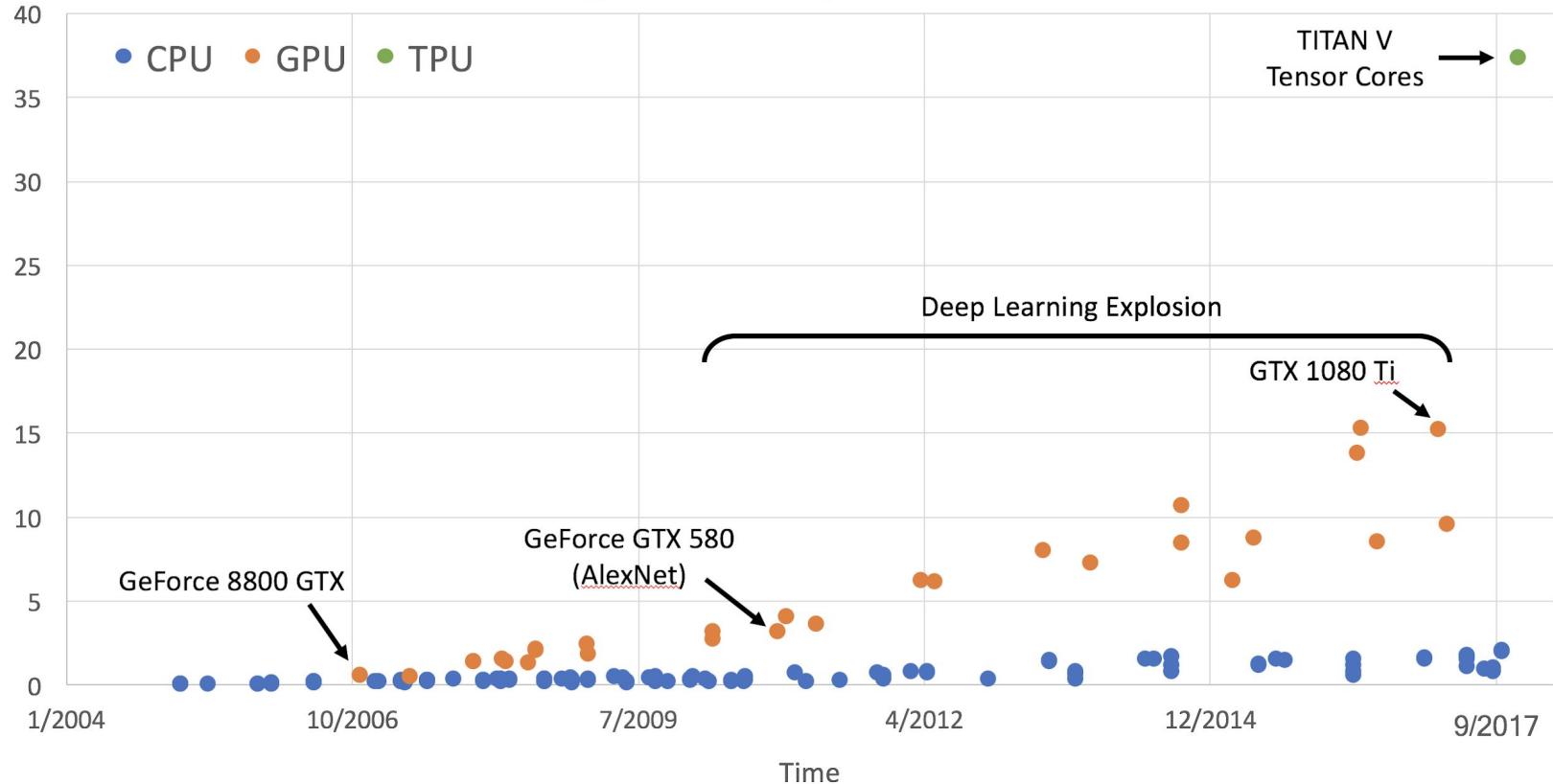
TPU: Specialized hardware for deep learning

CPU vs GPU

	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	\$339	~540 GFLOPs FP32
GPU (NVIDIA GTX 1080 Ti)	3584	1.6 GHz	11 GB GDDR5 X	\$699	~11.4 TFLOPs FP32
TPU NVIDIA TITAN V	5120 CUDA, 640 Tensor	1.5 GHz	12GB HBM2	\$2999	~14 TFLOPs FP32 ~112 TFLOP FP16
TPU Google Cloud TPU	?	?	64 GB HBM	\$6.50 per hour	~180 TFLOP

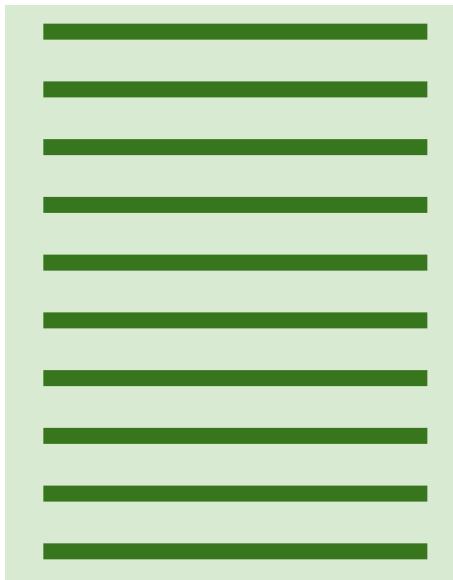
NOTE: TITAN V isn't technically a "TPU" since that's a Google term, but both have hardware specialized for deep learning

GigaFLOPs per Dollar

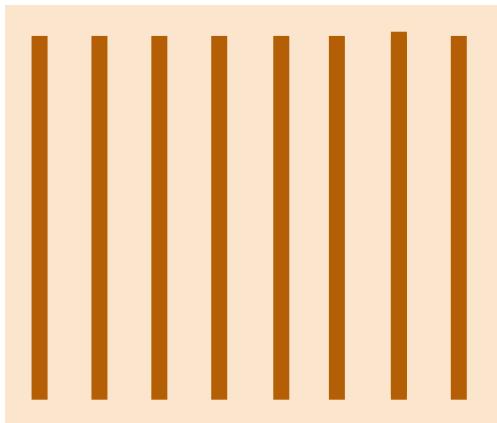


Example: Matrix Multiplication

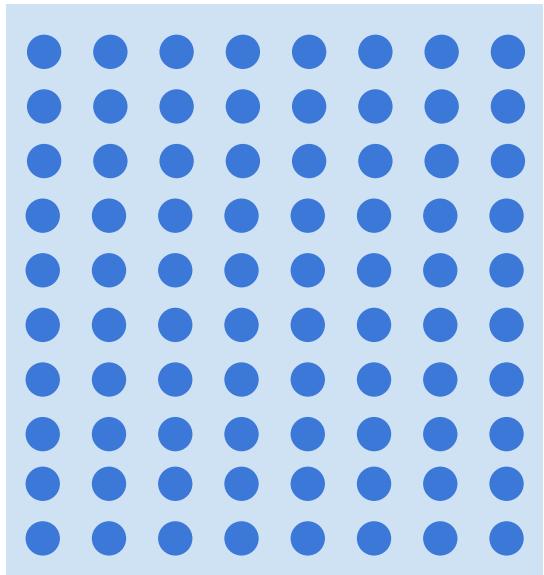
$A \times B$



$B \times C$



$A \times C$



=

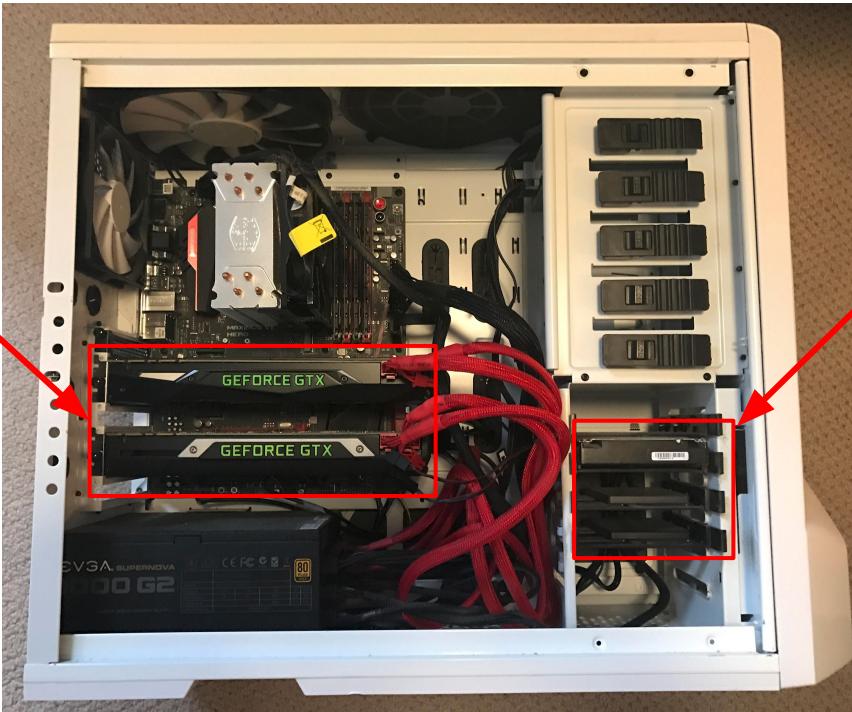
Programming GPUs

- CUDA (NVIDIA only)
 - Write C-like code that runs directly on the GPU
 - Optimized APIs: cuBLAS, cuFFT, cuDNN, etc
- OpenCL
 - Similar to CUDA, but runs on anything
 - Usually slower on NVIDIA hardware
- HIP <https://github.com/ROCm-Developer-Tools/HIP>
 - New project that automatically converts CUDA code to something that can run on AMD GPUs
- Udacity: Intro to Parallel Programming
<https://www.udacity.com/course/cs344>
 - For deep learning just use existing libraries

CPU / GPU Communication

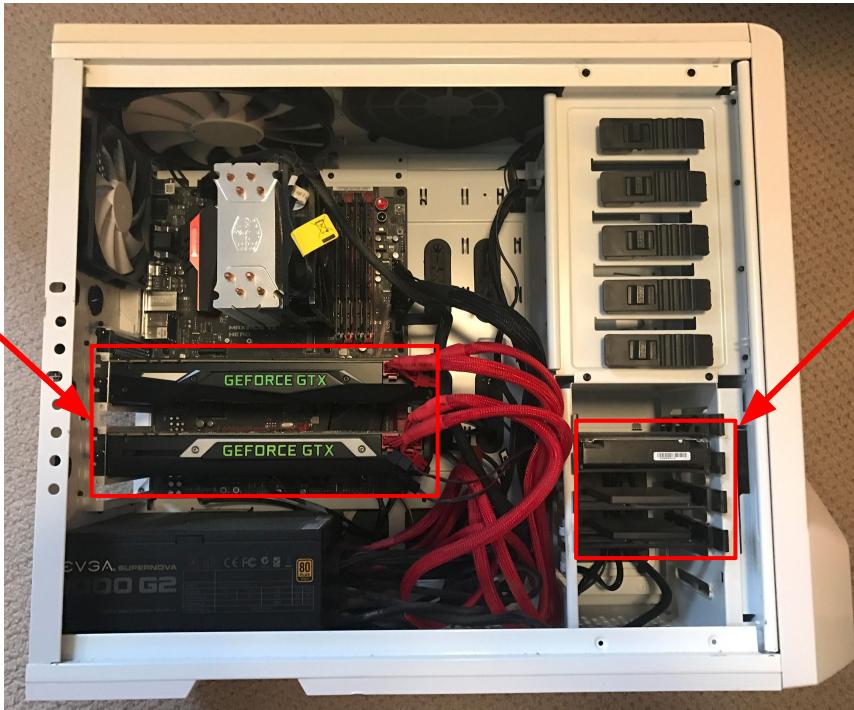
Model
is here

Data is here



CPU / GPU Communication

Model
is here



Data is here

If you aren't careful, training can bottleneck on reading data and transferring to GPU!

Solutions:

- Read all data into RAM
- Use SSD instead of HDD
- Use multiple CPU threads to prefetch data

Deep Learning Software

A zoo of frameworks!

Caffe
(UC Berkeley)



Caffe2
(Facebook)

Torch
(NYU / Facebook)



PyTorch
(Facebook)

Theano
(U Montreal)



TensorFlow
(Google)

PaddlePaddle
(Baidu)

MXNet
(Amazon)

Developed by U Washington, CMU, MIT, Hong Kong U, etc but main framework of choice at AWS

Chainer

CNTK
(Microsoft)

Deeplearning4j

And others...

A zoo of frameworks!

Caffe
(UC Berkeley)



Caffe2
(Facebook)

Torch
(NYU / Facebook)



PyTorch
(Facebook)

Theano
(U Montreal)



TensorFlow
(Google)

We'll focus on these

PaddlePaddle
(Baidu)

MXNet
(Amazon)

Developed by U Washington, CMU, MIT, Hong Kong U, etc but main framework of choice at AWS

Chainer

CNTK
(Microsoft)

Deeplearning4j

And others...

A zoo of frameworks!

Caffe
(UC Berkeley)



Caffe2
(Facebook)

Torch
(NYU / Facebook)



PyTorch
(Facebook)

Theano
(U Montreal)



TensorFlow
(Google)

I've mostly used these

PaddlePaddle
(Baidu)

MXNet
(Amazon)

Developed by U Washington, CMU, MIT, Hong Kong U, etc but main framework of choice at AWS

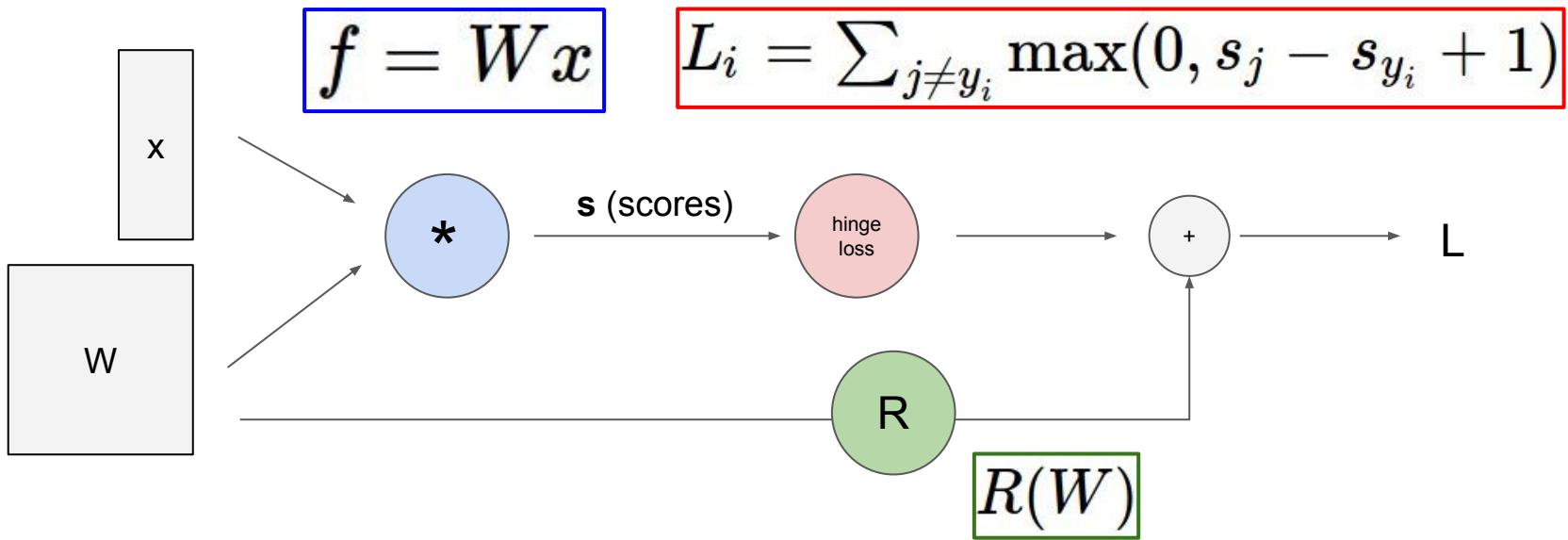
Chainer

CNTK
(Microsoft)

Deeplearning4j

And others...

Recall: Computational Graphs



Recall: Computational Graphs

input image

weights

loss

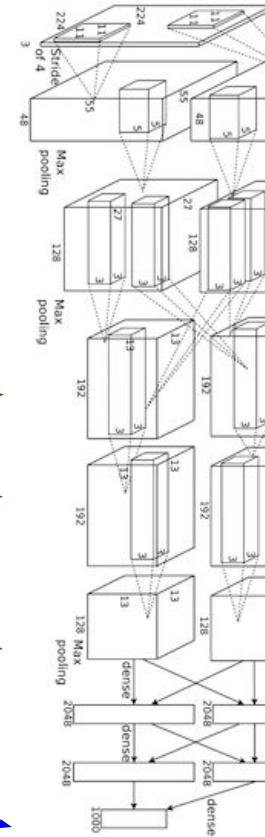


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

Recall: Computational Graphs

input image

loss

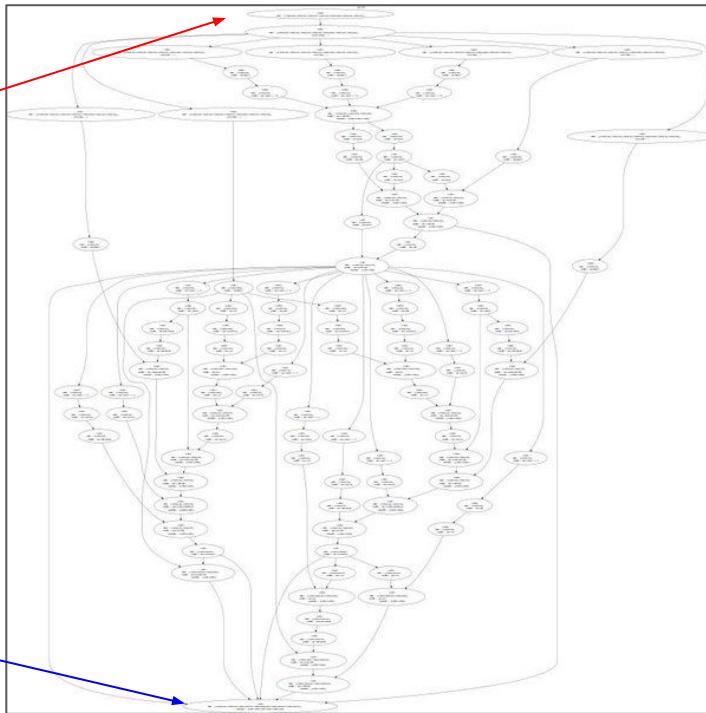


Figure reproduced with permission from a [Twitter post](#) by Andrej Karpathy.

The point of deep learning frameworks

- (1) Quick to develop and test new ideas
- (2) Automatically compute gradients
- (3) Run it all efficiently on GPU (wrap cuDNN, cuBLAS, etc)

Computational Graphs

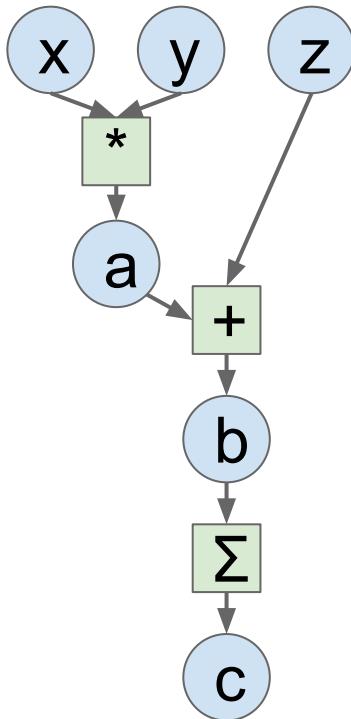
Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)
```



Computational Graphs

Numpy

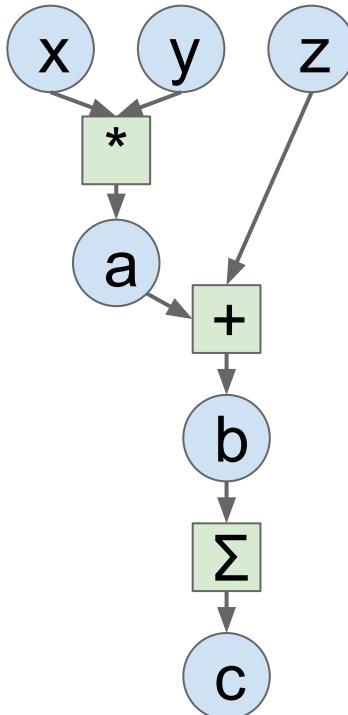
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



Computational Graphs

Numpy

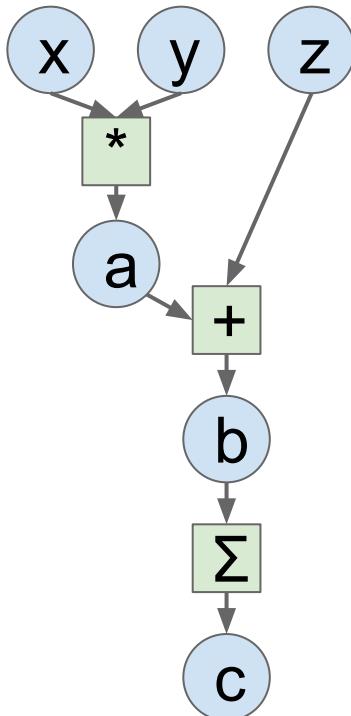
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



Good:

- Clean API, easy to write numeric code

Bad:

- Have to compute our own gradients
- Can't run on GPU

Computational Graphs

Numpy

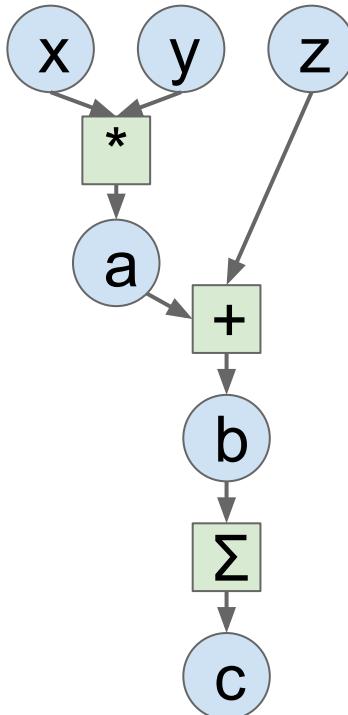
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

```
import torch

N, D = 3, 4
x = torch.randn(N, D)
y = torch.randn(N, D)
z = torch.randn(N, D)

a = x * y
b = a + z
c = torch.sum(b)
```

Looks exactly like numpy!

Computational Graphs

Numpy

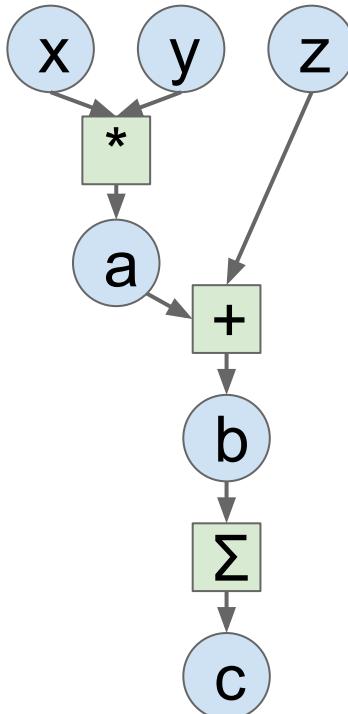
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

```
import torch

N, D = 3, 4
x = torch.randn(N, D, requires_grad=True)
y = torch.randn(N, D)
z = torch.randn(N, D)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
print(x.grad)
```

PyTorch handles gradients for us!

Computational Graphs

Numpy

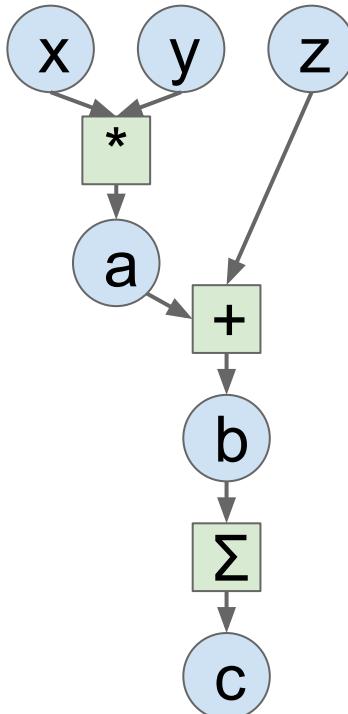
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

```
import torch
device = 'cuda:0'
N, D = 3, 4
x = torch.randn(N, D, requires_grad=True,
               device=device)
y = torch.randn(N, D, device=device)
z = torch.randn(N, D, device=device)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
print(x.grad)
```

Trivial to run on GPU - just construct arrays on a different device!

PyTorch

(More detail)

PyTorch: Fundamental Concepts

Tensor: Like a numpy array, but can run on GPU

Autograd: Package for building computational graphs out of Tensors, and automatically computing gradients

Module: A neural network layer; may store state or learnable weights

PyTorch: Versions

For this class we are using **PyTorch version 0.4** which was released Tuesday 4/24

This version makes a lot of changes to some of the core APIs around autograd, Tensor construction, Tensor datatypes / devices, etc

Be careful if you are looking at older PyTorch code!

PyTorch: Tensors

Running example: Train
a two-layer ReLU
network on random data
with L2 loss

```
import torch

device = torch.device('cpu')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

PyTorch Tensors are just like numpy arrays, but they can run on GPU.

PyTorch Tensor API looks almost exactly like numpy!

Here we fit a two-layer net using PyTorch Tensors:

```
import torch

device = torch.device('cpu')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

Create random tensors
for data and weights



```
import torch

device = torch.device('cpu')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

Forward pass: compute predictions and loss



```
import torch

device = torch.device('cpu')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

Backward pass:
manually compute
gradients



```
import torch

device = torch.device('cpu')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

Gradient descent
step on weights

```
import torch

device = torch.device('cpu')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

To run on GPU, just use a different device!

```
import torch  
device = torch.device('cuda:0')  
  
N, D_in, H, D_out = 64, 1000, 100, 10  
x = torch.randn(N, D_in, device=device)  
y = torch.randn(N, D_out, device=device)  
w1 = torch.randn(D_in, H, device=device)  
w2 = torch.randn(H, D_out, device=device)  
  
learning_rate = 1e-6  
for t in range(500):  
    h = x.mm(w1)  
    h_relu = h.clamp(min=0)  
    y_pred = h_relu.mm(w2)  
    loss = (y_pred - y).pow(2).sum()  
  
    grad_y_pred = 2.0 * (y_pred - y)  
    grad_w2 = h_relu.t().mm(grad_y_pred)  
    grad_h_relu = grad_y_pred.mm(w2.t())  
    grad_h = grad_h_relu.clone()  
    grad_h[h < 0] = 0  
    grad_w1 = x.t().mm(grad_h)  
  
    w1 -= learning_rate * grad_w1  
    w2 -= learning_rate * grad_w2
```

PyTorch: Autograd

Creating Tensors with
requires_grad=True enables
autograd

Operations on Tensors with
requires_grad=True cause PyTorch
to build a computational graph

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd

We will not want gradients
(of loss) with respect to data

Do want gradients with
respect to weights

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd

Forward pass looks exactly the same as before, but we don't need to track intermediate values - PyTorch keeps track of them for us in the graph

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd

Compute gradient of loss
with respect to w1 and w2

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd

Make gradient step on weights, then zero them. Torch.no_grad means “don’t build a computational graph for this part”

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch methods that end in underscore
modify the Tensor in-place; methods that
don't return a new Tensor

PyTorch: New Autograd Functions

Define your own autograd functions by writing forward and backward functions for Tensors

Very similar to modular layers in A2! Use ctx object to “cache” values for the backward pass, just like cache objects from A2

```
class MyReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_y):
        x, = ctx.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input
```

PyTorch: New Autograd Functions

Define your own autograd functions by writing forward and backward functions for Tensors

Very similar to modular layers in A2! Use ctx object to “cache” values for the backward pass, just like cache objects from A2

Define a helper function to make it easy to use the new function

```
class MyReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_y):
        x, = ctx.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input

def my_relu(x):
    return MyReLU.apply(x)
```

PyTorch: New Autograd Functions

```
class MyReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_y):
        x, = ctx.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input

def my_relu(x):
    return MyReLU.apply(x)
```

Can use our new autograd
function in the forward pass

```
N, D_in, H, D_out = 64, 1000, 100, 10

x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = my_relu(x.mm(w1)).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: New Autograd Functions

```
def my_relu(x):
    return x.clamp(min=0)
```

In practice you almost never need to define new autograd functions! Only do it when you need custom backward. In this case we can just use a normal Python function

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = my_relu(x.mm(w1)).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: nn

Higher-level wrapper for working with neural nets

Use this! It will make your life easier

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
```

PyTorch: nn

Define our model as a sequence of layers; each layer is an object that holds learnable weights



```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
```

PyTorch: nn

Forward pass: feed data to model, and compute loss

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
```

PyTorch: nn

Forward pass: feed data to model, and compute loss

torch.nn.functional has useful helpers like loss functions

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
```

PyTorch: nn

Backward pass: compute gradient with respect to all model weights (they have `requires_grad=True`)

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
```

PyTorch: nn

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()
```

```
with torch.no_grad():
    for param in model.parameters():
        param -= learning_rate * param.grad
model.zero_grad()
```

Make gradient step on
each model parameter
(with gradients disabled)



PyTorch: optim

Use an **optimizer** for different update rules

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)

for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    optimizer.step()
    optimizer.zero_grad()
```

PyTorch: optim

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)

for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    optimizer.step()
    optimizer.zero_grad()
```

After computing gradients, use
optimizer to update params
and zero gradients



```
optimizer.step()  
optimizer.zero_grad()
```

Aside: Lua Torch

Direct ancestor of PyTorch
(they used to share a lot of C
backend)

Written in Lua, not Python

Torch has Tensors and Modules
like PyTorch, but no full-featured
autograd; much more painful to
work with

More details: Check 2016 slides

```
require 'torch'
require 'nn'
require 'optim'

local N, D, H, C = 64, 256, 512, 10

local model = nn.Sequential()
model:add(nn.Linear(D, H))
model:add(nn.ReLU())
model:add(nn.Linear(H, C))
local loss_fn = nn.CrossEntropyCriterion()

local x = torch.randn(N, D)
local y = torch.Tensor(N):random(C)
local weights, grad_weights = model:getParameters()

local function f(w)
    assert(w == weights)
    local scores = model:forward(x)
    local loss = loss_fn:forward(scores, y)

    grad_weights:zero()
    local grad_scores = loss_fn:backward(scores, y)
    local grad_x = model:backward(x, grad_scores)

    return loss, grad_weights
end

local state = {learningRate=1e-3}
for t = 1, 100 do
    optim.adam(f, weights, state)
end
```

PyTorch: nn

Define new Modules

A PyTorch **Module** is a neural net layer; it inputs and outputs Tensors

Modules can contain weights or other modules

You can define your own Modules using autograd!

```
import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

PyTorch: nn

Define new Modules

Define our whole model
as a single Module

```
import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

PyTorch: nn

Define new Modules

Initializer sets up two children (Modules can contain modules)

```
import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

PyTorch: nn

Define new Modules

Define forward pass using child modules

No need to define backward - autograd will handle it

```
import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

PyTorch: nn

Define new Modules

Construct and train an instance of our model

```
import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
```

```
model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

PyTorch: nn

Define new Modules

Very common to mix and match
custom Module subclasses and
Sequential containers

```
import torch

class ParallelBlock(torch.nn.Module):
    def __init__(self, D_in, D_out):
        super(ParallelBlock, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, D_out)
        self.linear2 = torch.nn.Linear(D_in, D_out)
    def forward(self, x):
        h1 = self.linear1(x)
        h2 = self.linear2(x)
        return (h1 * h2).clamp(min=0)

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
    ParallelBlock(D_in, H),
    ParallelBlock(H, H),
    torch.nn.Linear(H, D_out))

optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

PyTorch: nn

Define new Modules

Define network component
as a Module subclass



```
import torch

class ParallelBlock(torch.nn.Module):
    def __init__(self, D_in, D_out):
        super(ParallelBlock, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, D_out)
        self.linear2 = torch.nn.Linear(D_in, D_out)
    def forward(self, x):
        h1 = self.linear1(x)
        h2 = self.linear2(x)
        return (h1 * h2).clamp(min=0)

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
    ParallelBlock(D_in, H),
    ParallelBlock(H, H),
    torch.nn.Linear(H, D_out))

optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

PyTorch: nn

Define new Modules

Stack multiple instances of the component in a sequential



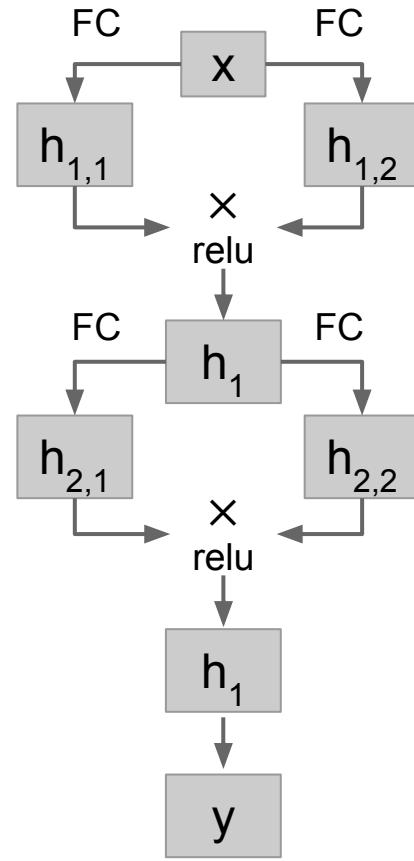
```
import torch

class ParallelBlock(torch.nn.Module):
    def __init__(self, D_in, D_out):
        super(ParallelBlock, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, D_out)
        self.linear2 = torch.nn.Linear(D_in, D_out)
    def forward(self, x):
        h1 = self.linear1(x)
        h2 = self.linear2(x)
        return (h1 * h2).clamp(min=0)

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
    ParallelBlock(D_in, H),
    ParallelBlock(H, H),
    torch.nn.Linear(H, D_out))

optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```



```

import torch

class ParallelBlock(torch.nn.Module):
    def __init__(self, D_in, D_out):
        super(ParallelBlock, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, D_out)
        self.linear2 = torch.nn.Linear(D_in, D_out)

    def forward(self, x):
        h1 = self.linear1(x)
        h2 = self.linear2(x)
        return (h1 * h2).clamp(min=0)

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
    ParallelBlock(D_in, H),
    ParallelBlock(H, H),
    torch.nn.Linear(H, D_out))

optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
  
```

PyTorch: DataLoaders

A **DataLoader** wraps a **Dataset** and provides minibatching, shuffling, multithreading, for you

When you need to load custom data, just write your own Dataset class

```
import torch
from torch.utils.data import TensorDataset, DataLoader

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# loader = DataLoader(TensorDataset(x, y), batch_size=8)
model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)
for epoch in range(20):
    for x_batch, y_batch in loader:
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred, y_batch)

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

PyTorch: DataLoaders

Iterate over loader to form minibatches

```
import torch
from torch.utils.data import TensorDataset, DataLoader

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

loader = DataLoader(TensorDataset(x, y), batch_size=8)
model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)
for epoch in range(20):
    for x_batch, y_batch in loader:
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred, y_batch)

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

PyTorch: Pretrained Models

Super easy to use pretrained models with torchvision
<https://github.com/pytorch/vision>

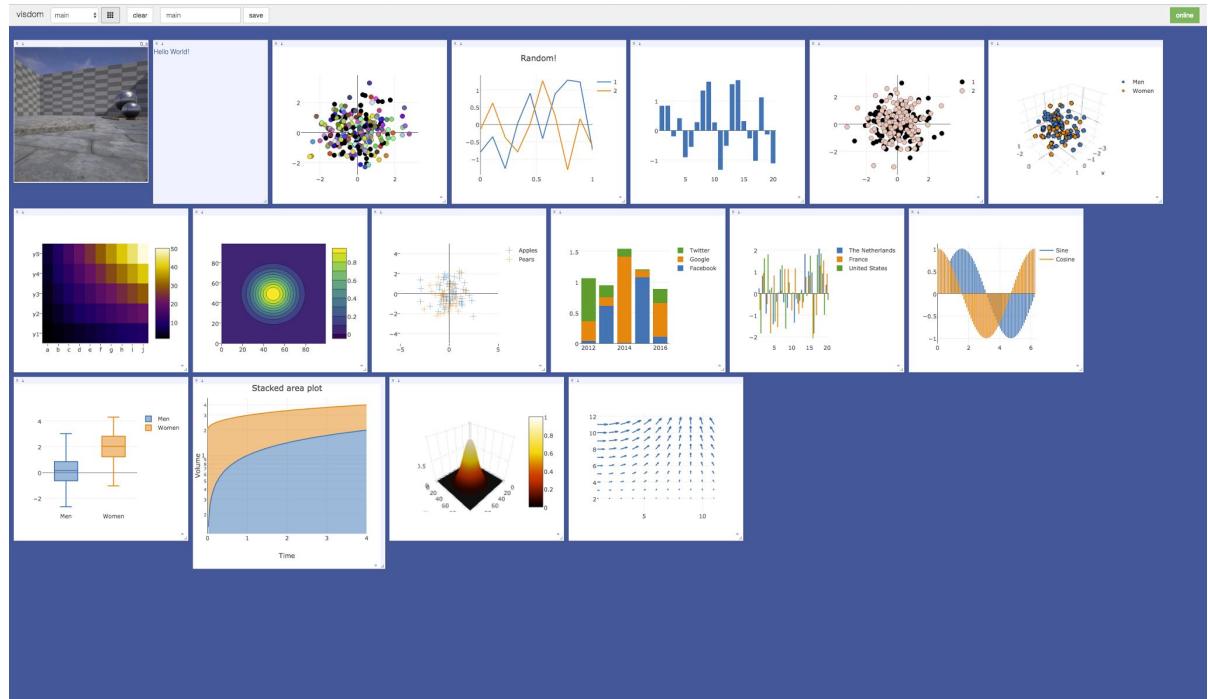
```
import torch
import torchvision

alexnet = torchvision.models.alexnet(pretrained=True)
vgg16 = torchvision.models.vgg16(pretrained=True)
resnet101 = torchvision.models.resnet101(pretrained=True)
```

PyTorch: Visdom

Visualization tool: add logging to your code, then visualize in a browser

Can't visualize computational graph structure (yet?)



<https://github.com/facebookresearch/visdom>

This image is licensed under CC-BY 4.0; no changes were made to the image

PyTorch: Dynamic Computation Graphs

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

PyTorch: Dynamic Computation Graphs

x

w1

w2

y

```
import torch

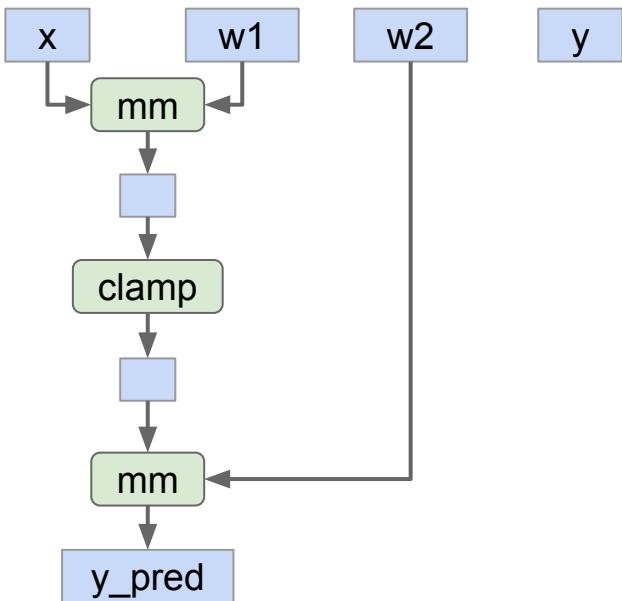
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Create Tensor objects

PyTorch: Dynamic Computation Graphs



```
import torch

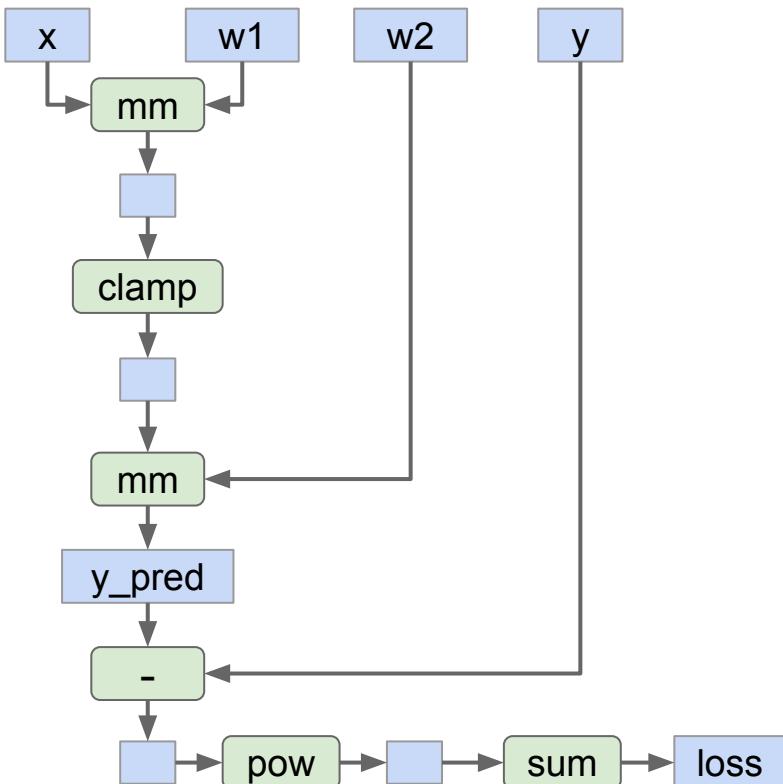
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Build graph data structure AND
perform computation

PyTorch: Dynamic Computation Graphs



```
import torch

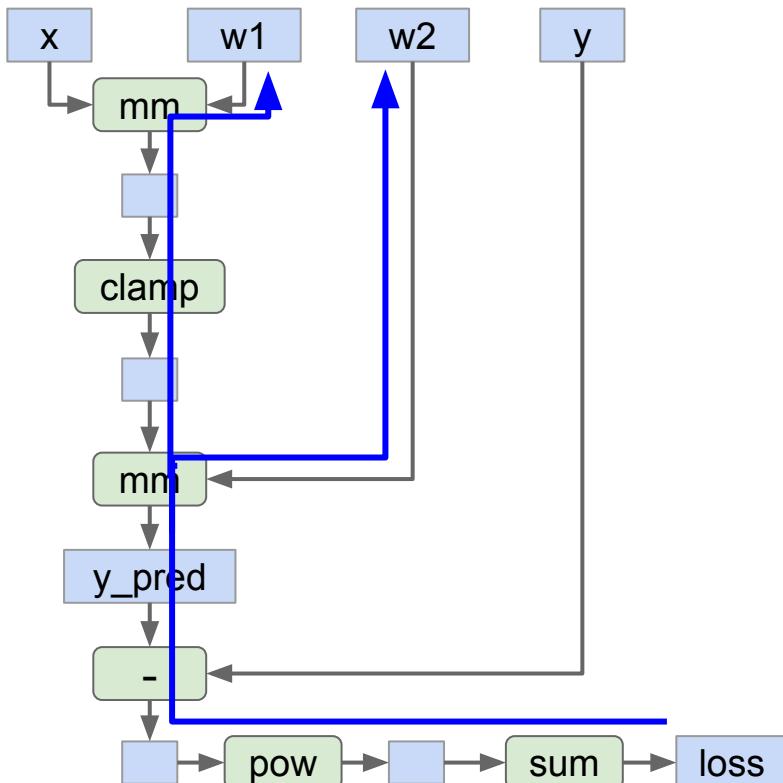
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Build graph data structure AND
perform computation

PyTorch: Dynamic Computation Graphs



```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Search for path between loss and w1, w2
(for backprop) AND perform computation

PyTorch: Dynamic Computation Graphs

x

w1

w2

y

```
import torch

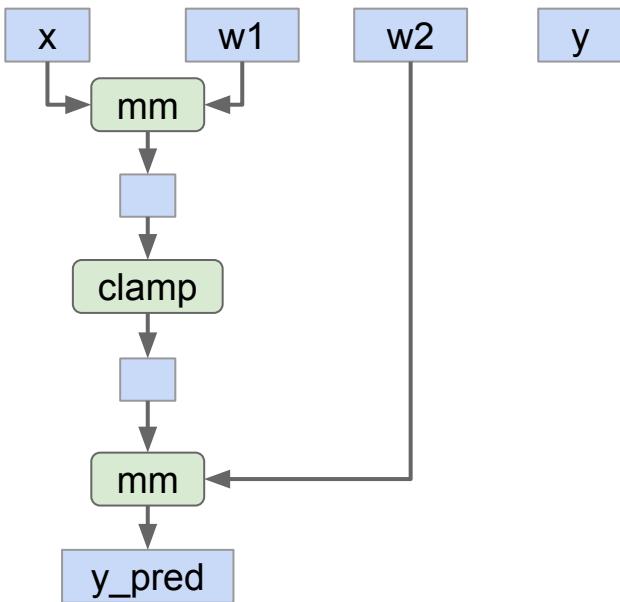
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Throw away the graph, backprop path, and rebuild it from scratch on every iteration

PyTorch: Dynamic Computation Graphs



```
import torch

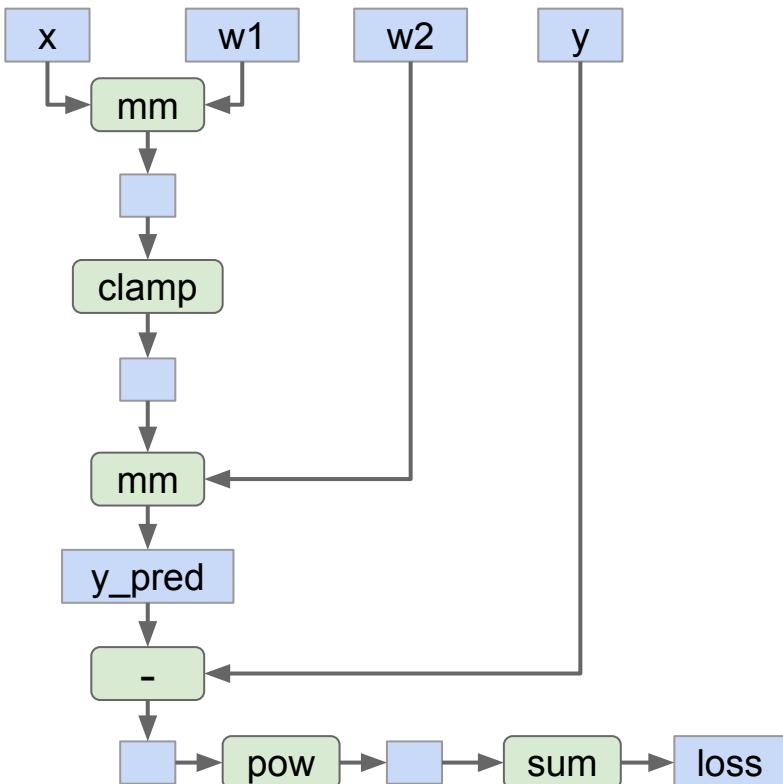
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Build graph data structure AND
perform computation

PyTorch: Dynamic Computation Graphs



```
import torch

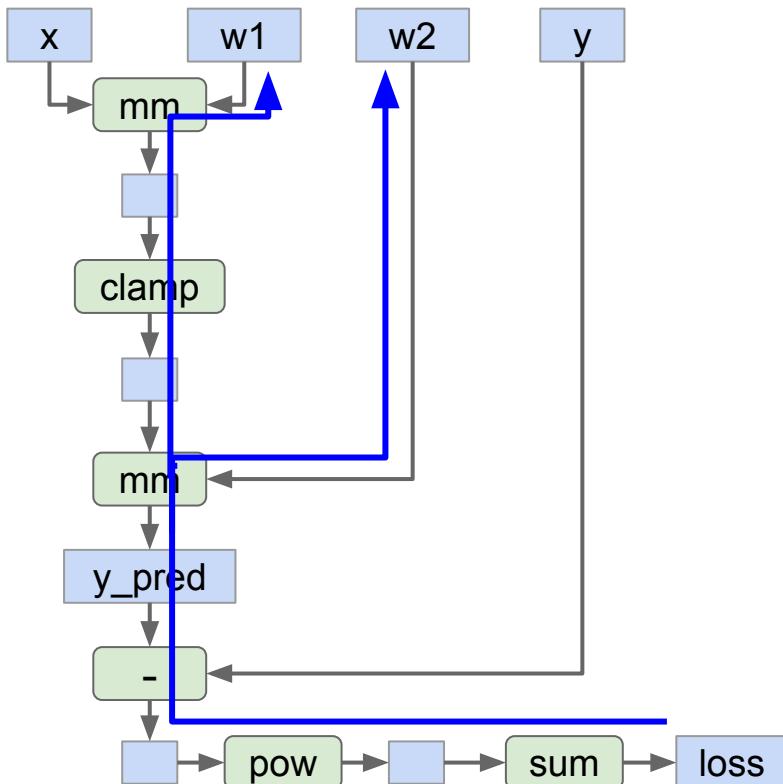
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Build graph data structure AND
perform computation

PyTorch: Dynamic Computation Graphs



```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Search for path between loss and w1, w2
(for backprop) AND perform computation

PyTorch: Dynamic Computation Graphs

Building the graph and
computing the graph happen at
the same time.

Seems inefficient, especially if we
are building the same graph over
and over again...

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Static Computation Graphs

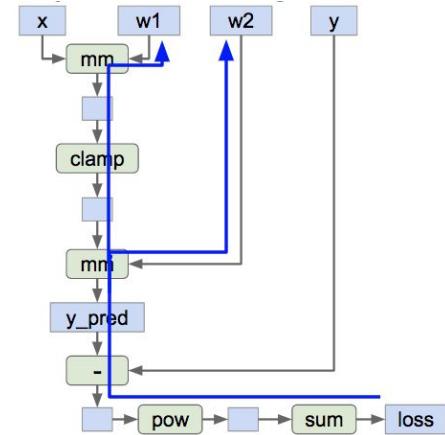
Alternative: **Static** graphs

Step 1: Build computational graph
describing our computation
(including finding paths for
backprop)

Step 2: Reuse the same graph on
every iteration

```
graph = build_graph()

for x_batch, y_batch in loader:
    run_graph(graph, x=x_batch, y=y_batch)
```



TensorFlow

TensorFlow: Neural Net

```
import numpy as np
import tensorflow as tf
```

(Assume imports at the top of each snippet)

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                  feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net

First **define**
computational graph



```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])
```

Then **run** the graph
many times



```
with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                  feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net

Create **placeholders** for
input x, weights w1 and
w2, and targets y

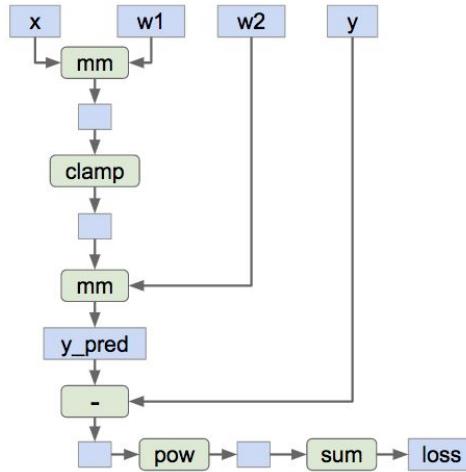
```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                  feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net



Forward pass: compute prediction for y and loss. No computation - just building graph

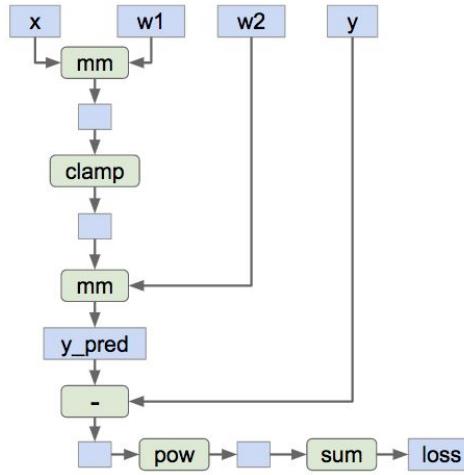
```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D)}
    out = sess.run([loss, grad_w1, grad_w2],
                  feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net



Tell TensorFlow to compute loss of gradient with respect to w_1 and w_2 .
No compute - just building the graph

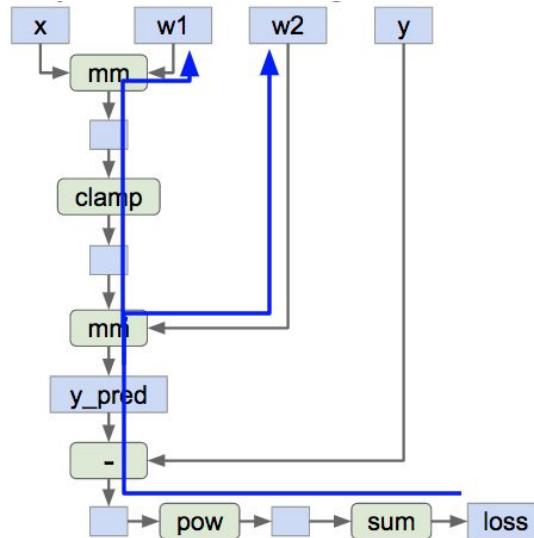
```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D)}
    out = sess.run([loss, grad_w1, grad_w2],
                  feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net



Find paths between loss and w_1 , w_2

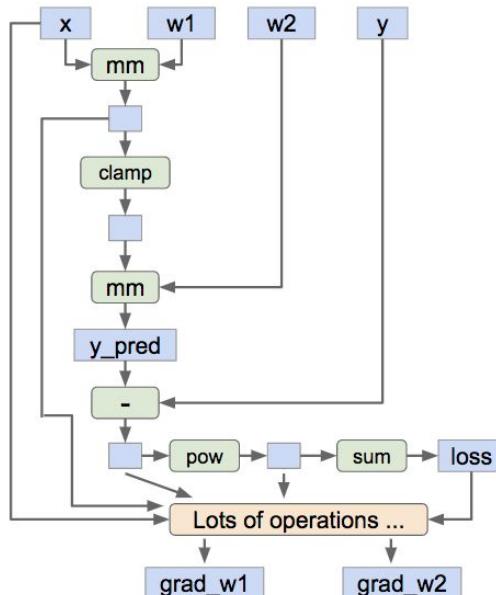
```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                  feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net



Add new operators to the graph which compute grad_{w1} and grad_{w2}

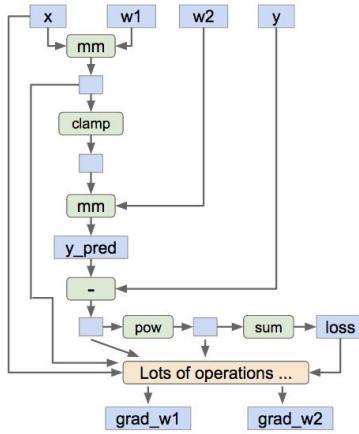
```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                  feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net



Now done building our graph,
so we enter a **session** so we
can actually run the graph

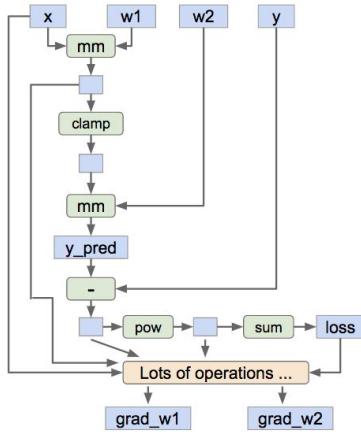
```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D)}
    out = sess.run([loss, grad_w1, grad_w2],
                  feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net



Create numpy arrays that will fill in the placeholders above

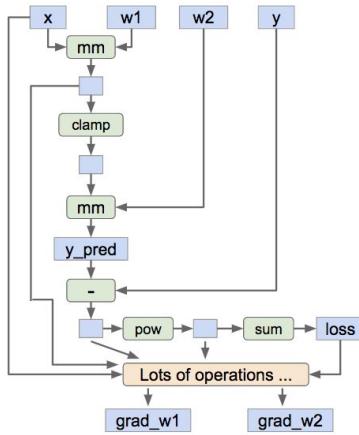
```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                  feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net



Run the graph: feed in the numpy arrays for x, y, w1, and w2; get numpy arrays for loss, grad_w1, and grad_w2

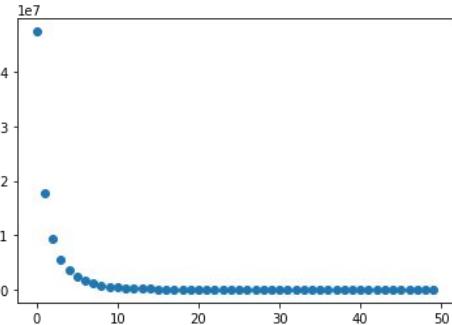
```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                  feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net



Train the network: Run the graph over and over, use gradient to update weights

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    learning_rate = 1e-5
    for t in range(50):
        out = sess.run([loss, grad_w1, grad_w2],
                      feed_dict=values)
        loss_val, grad_w1_val, grad_w2_val = out
        values[w1] -= learning_rate * grad_w1_val
        values[w2] -= learning_rate * grad_w2_val
```

TensorFlow: Neural Net

Problem: copying
weights between CPU /
GPU each step

Train the network: Run
the graph over and over,
use gradient to update
weights

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    learning_rate = 1e-5
    for t in range(50):
        out = sess.run([loss, grad_w1, grad_w2],
                      feed_dict=values)
        loss_val, grad_w1_val, grad_w2_val = out
        values[w1] -= learning_rate * grad_w1_val
        values[w2] -= learning_rate * grad_w2_val
```

TensorFlow: Neural Net

Change w1 and w2 from
placeholder (fed on
each call) to **Variable**
(persists in the graph
between calls)

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(tf.square(diff), axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, = sess.run([loss], feed_dict=values)
```

TensorFlow: Neural Net

Add **assign** operations
to update w1 and w2 as
part of the graph!

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])
```

```
learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
```

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, = sess.run([loss], feed_dict=values)
```

TensorFlow: Neural Net

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

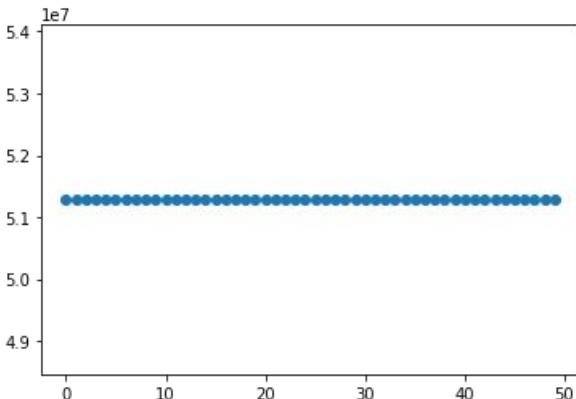
learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, = sess.run([loss], feed_dict=values)
```

Run graph once to
initialize w1 and w2

Run many times to train

TensorFlow: Neural Net



Problem: loss not going down! Assign calls not actually being executed!

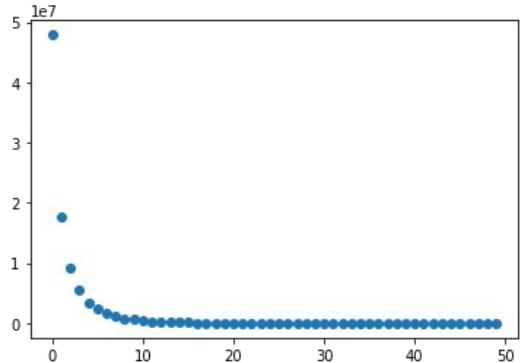
```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, = sess.run([loss], feed_dict=values)
```

TensorFlow: Neural Net



Add dummy graph node
that depends on updates

Tell TensorFlow to
compute dummy node

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
updates = tf.group(new_w1, new_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    losses = []
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                              feed_dict=values)
```

TensorFlow: Optimizer

Can use an **optimizer** to
compute gradients and
update weights

Remember to execute the
output of the optimizer!

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff * diff, axis=1))
```

```
optimizer = tf.train.GradientDescentOptimizer(1e-5)
updates = optimizer.minimize(loss)
```

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    losses = []
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                              feed_dict=values)
```

TensorFlow: LOSS

Use predefined
common losseses

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
loss = tf.losses.mean_squared_error(y_pred, y)

optimizer = tf.train.GradientDescentOptimizer(1e-3)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                              feed_dict=values)
```

TensorFlow: Layers

tf.layers automatically
sets up weight and
(and bias) for us!

Use He
initializer

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))

init = tf.variance_scaling_initializer(2.0)
h = tf.layers.dense(inputs=x, units=H,
                     activation=tf.nn.relu, kernel_initializer=init)
y_pred = tf.layers.dense(inputs=h, units=D,
                         kernel_initializer=init)

loss = tf.losses.mean_squared_error(y_pred, y)

optimizer = tf.train.GradientDescentOptimizer(1e-0)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                              feed_dict=values)
```

Keras: High-Level Wrapper

Keras is a layer on top of
TensorFlow, makes common
things easy to do

(Used to be third-party, now
merged into TensorFlow)

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))

model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
y_pred = model(x)
loss = tf.losses.mean_squared_error(y_pred, y)

optimizer = tf.train.GradientDescentOptimizer(1e-0)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D)}
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                              feed_dict=values)
```

Keras: High-Level Wrapper

Define model as a sequence of layers

Get output by calling the model

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))

model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
y_pred = model(x)
loss = tf.losses.mean_squared_error(y_pred, y)

optimizer = tf.train.GradientDescentOptimizer(1e-0)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D)}
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                              feed_dict=values)
```

Keras: High-Level Wrapper

```
N, D, H = 64, 1000, 100
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), 
                                activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))

model.compile(loss=tf.keras.losses.mean_squared_error,
              optimizer=tf.keras.optimizers.SGD(lr=1e-0))
```

```
x = np.random.randn(N, D)
y = np.random.randn(N, D)
```

Keras can handle the training loop for you!

No sessions or feed_dict

→ `history = model.fit(x, y, epochs=50, batch_size=N)`

TensorFlow: High-Level Wrappers

tf.keras (https://www.tensorflow.org/api_docs/python/tf/keras)

tf.layers (https://www.tensorflow.org/api_docs/python/tf/layers)

TensorFlow: High-Level Wrappers

Keras (<https://keras.io/>)

tf.keras (https://www.tensorflow.org/api_docs/python/tf/keras)

tf.layers (https://www.tensorflow.org/api_docs/python/tf/layers)

TensorFlow: High-Level Wrappers

Keras (<https://keras.io/>)

tf.keras (https://www.tensorflow.org/api_docs/python/tf/keras)

tf.layers (https://www.tensorflow.org/api_docs/python/tf/layers)

tf.estimator (https://www.tensorflow.org/api_docs/python/tf/estimator)

TensorFlow: High-Level Wrappers

Keras (<https://keras.io/>)

Ships with TensorFlow

tf.keras (https://www.tensorflow.org/api_docs/python/tf/keras)

tf.layers (https://www.tensorflow.org/api_docs/python/tf/layers)

tf.estimator (https://www.tensorflow.org/api_docs/python/tf/estimator)

tf.contrib.estimator (https://www.tensorflow.org/api_docs/python/tf/contrib/estimator)

tf.contrib.layers (https://www.tensorflow.org/api_docs/python/tf/contrib/layers)

tf.contrib.slim (<https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim>)

tf.contrib.learn (https://www.tensorflow.org/api_docs/python/tf/contrib/learn)

TensorFlow: High-Level Wrappers

Keras (<https://keras.io/>)

Ships with TensorFlow

tf.keras (https://www.tensorflow.org/api_docs/python/tf/keras)

tf.layers (https://www.tensorflow.org/api_docs/python/tf/layers)

tf.estimator (https://www.tensorflow.org/api_docs/python/tf/estimator)

tf.contrib.estimator (https://www.tensorflow.org/api_docs/python/tf/contrib/estimator)

tf.contrib.layers (https://www.tensorflow.org/api_docs/python/tf/contrib/layers)

tf.contrib.slim (<https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim>)

~~tf.contrib.learn~~ (https://www.tensorflow.org/api_docs/python/tf/contrib/learn) DEPRECATED

TensorFlow: High-Level Wrappers

Keras (<https://keras.io/>)

Ships with TensorFlow

tf.keras (https://www.tensorflow.org/api_docs/python/tf/keras)

tf.layers (https://www.tensorflow.org/api_docs/python/tf/layers)

tf.estimator (https://www.tensorflow.org/api_docs/python/tf/estimator)

tf.contrib.estimator (https://www.tensorflow.org/api_docs/python/tf/contrib/estimator)

tf.contrib.layers (https://www.tensorflow.org/api_docs/python/tf/contrib/layers)

tf.contrib.slim (<https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim>)

tf.contrib.learn (https://www.tensorflow.org/api_docs/python/tf/contrib/learn) DEPRECATED

Sonnet (<https://github.com/deepmind/sonnet>)

By DeepMind

TensorFlow: High-Level Wrappers

Keras (<https://keras.io/>)

Ships with TensorFlow

tf.keras (https://www.tensorflow.org/api_docs/python/tf/keras)

tf.layers (https://www.tensorflow.org/api_docs/python/tf/layers)

tf.estimator (https://www.tensorflow.org/api_docs/python/tf/estimator)

tf.contrib.estimator (https://www.tensorflow.org/api_docs/python/tf/contrib/estimator)

tf.contrib.layers (https://www.tensorflow.org/api_docs/python/tf/contrib/layers)

tf.contrib.slim (<https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim>)

tf.contrib.learn (https://www.tensorflow.org/api_docs/python/tf/contrib/learn) DEPRECATED

Sonnet (<https://github.com/deepmind/sonnet>) By DeepMind

TFLearn (<http://tflearn.org/>)

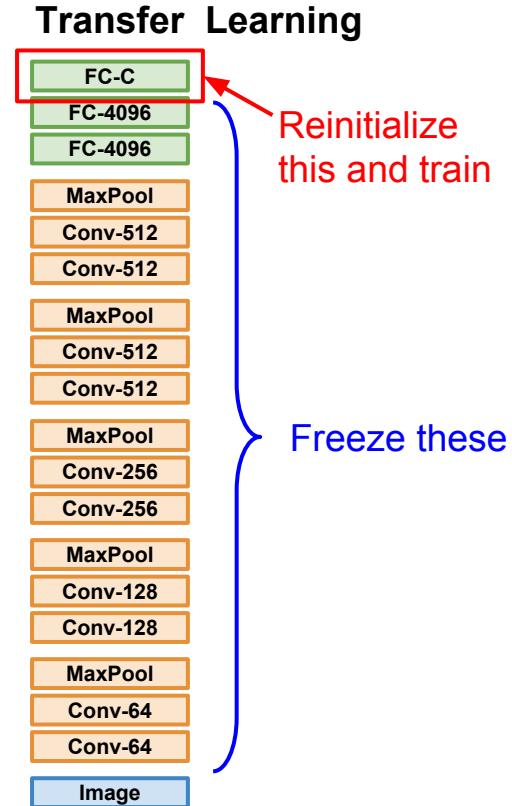
TensorLayer (<http://tensorlayer.readthedocs.io/en/latest/>)

Third-Party

TensorFlow: Pretrained Models

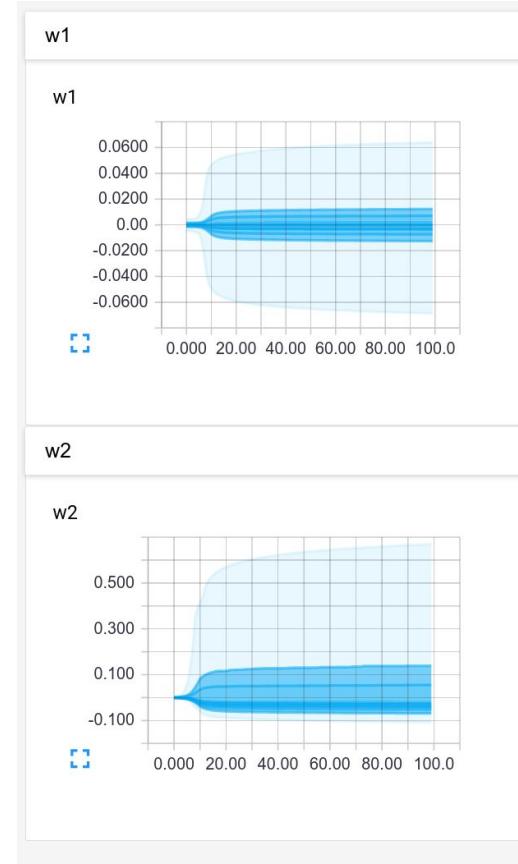
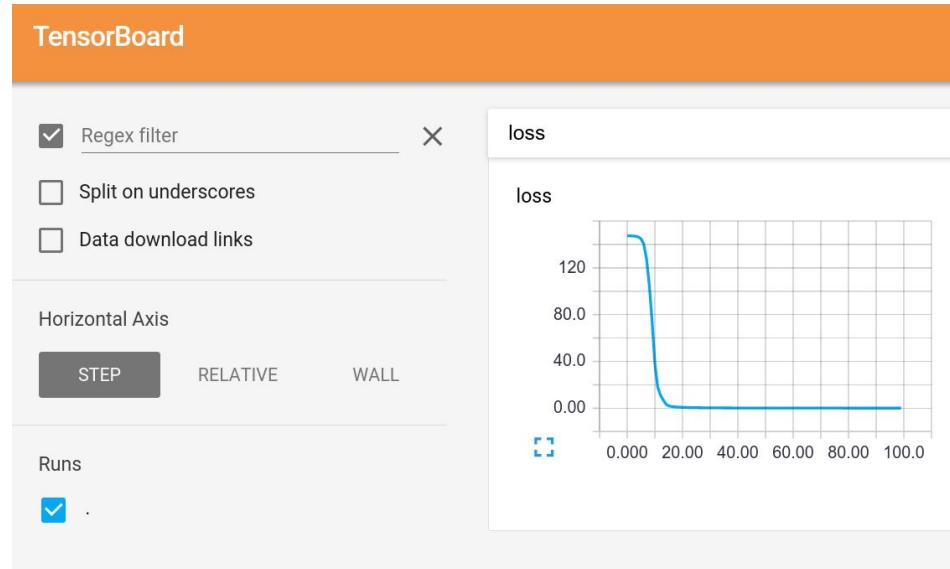
tf.keras: (https://www.tensorflow.org/api_docs/python/tf/keras/applications)

TF-Slim: (<https://github.com/tensorflow/models/tree/master/slim/nets>)

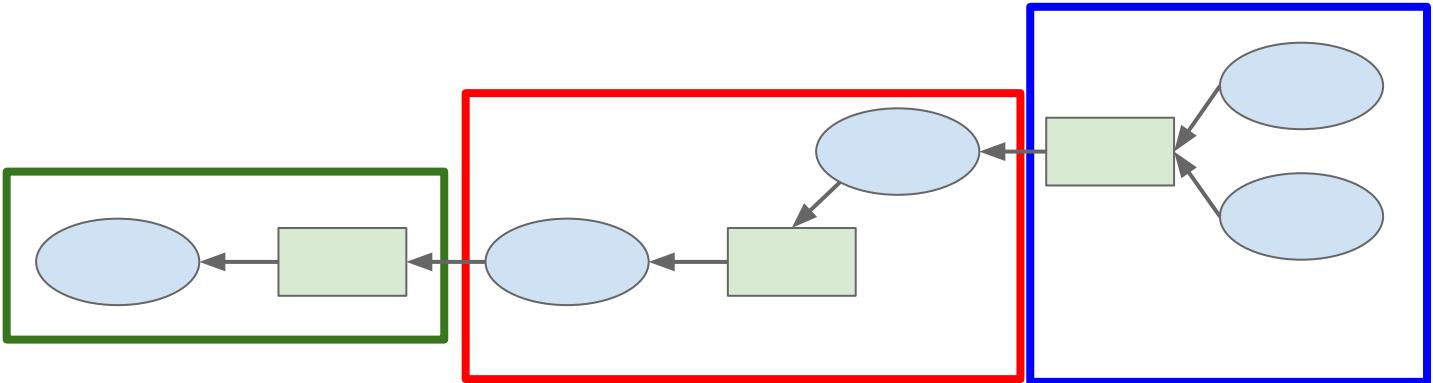


TensorFlow: Tensorboard

Add logging to code to record loss, stats, etc
Run server and get pretty graphs!



TensorFlow: Distributed Version

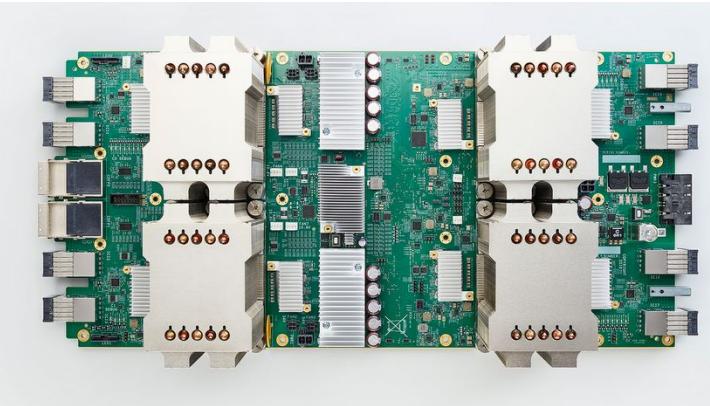


Split one graph
over multiple
machines!



<https://www.tensorflow.org/deploy/distributed>

TensorFlow: Tensor Processing Units



Google Cloud TPU
= 180 TFLOPs of compute!

TensorFlow: Tensor Processing Units



Google Cloud TPU
= 180 TFLOPs of compute!



NVIDIA Tesla V100
= 125 TFLOPs of compute

TensorFlow: Tensor Processing Units



Google Cloud TPU
= 180 TFLOPs of compute!



NVIDIA Tesla V100
= 125 TFLOPs of compute

NVIDIA Tesla P100 = 11 TFLOPs of compute
GTX 580 = 0.2 TFLOPs

TensorFlow: Tensor Processing Units



Google Cloud TPU
= 180 TFLOPs of compute!



Google Cloud TPU Pod
= 64 Cloud TPUs
= 11.5 PFLOPs of compute!

https://www.tensorflow.org/versions/master/programmers_guide/using_tpu

Static vs Dynamic Graphs

TensorFlow: Build graph once, then run many times (**static**)

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
updates = tf.group(new_w1, new_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D)}
    losses = []
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                              feed_dict=values)
```

Build graph

Run each iteration

PyTorch: Each forward pass defines a new graph (**dynamic**)

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

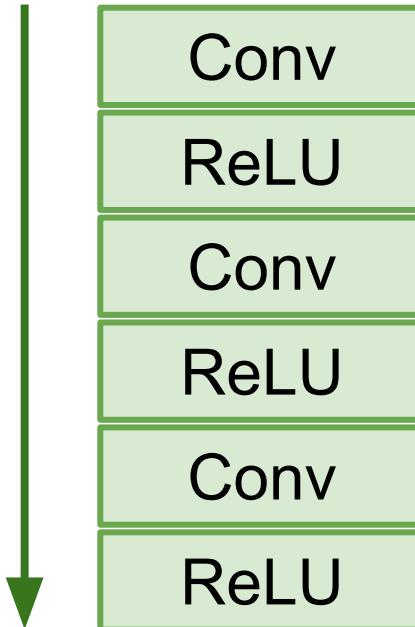
    loss.backward()
```

New graph each iteration

Static vs Dynamic: Optimization

With static graphs,
framework can
optimize the
graph for you
before it runs!

The graph you wrote



Equivalent graph with
fused operations



Static vs Dynamic: Serialization

Static

Once graph is built, can **serialize** it and run it without the code that built the graph!

Dynamic

Graph building and execution are intertwined, so always need to keep code around

Static vs Dynamic: Conditional

$$y = \begin{cases} w_1 * x & \text{if } z > 0 \\ w_2 * x & \text{otherwise} \end{cases}$$

Static vs Dynamic: Conditional

$$y = \begin{cases} w_1 * x & \text{if } z > 0 \\ w_2 * x & \text{otherwise} \end{cases}$$

PyTorch: Normal Python

```
N, D, H = 3, 4, 5

x = torch.randn(N, D, requires_grad=True)
w1 = torch.randn(D, H)
w2 = torch.randn(D, H)

z = 10
if z > 0:
    y = x.mm(w1)
else:
    y = x.mm(w2)
```

Static vs Dynamic: Conditional

$$y = \begin{cases} w_1 * x & \text{if } z > 0 \\ w_2 * x & \text{otherwise} \end{cases}$$

PyTorch: Normal Python

```
N, D, H = 3, 4, 5

x = torch.randn(N, D, requires_grad=True)
w1 = torch.randn(D, H)
w2 = torch.randn(D, H)

z = 10
if z > 0:
    y = x.mm(w1)
else:
    y = x.mm(w2)
```

TensorFlow: Special TF control flow operator!

```
N, D, H = 3, 4, 5
x = tf.placeholder(tf.float32, shape=(N, D))
z = tf.placeholder(tf.float32, shape=None)
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(D, H))
```

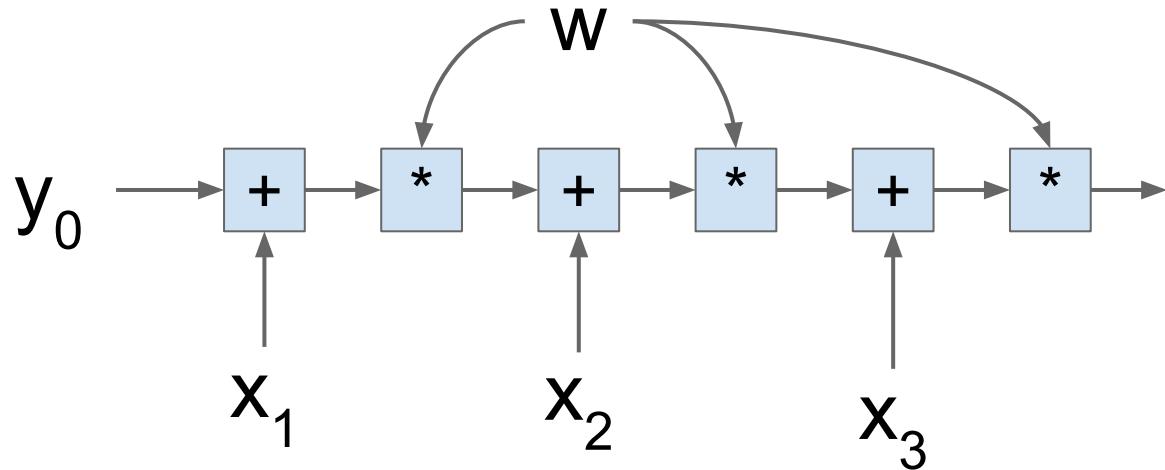
```
def f1(): return tf.matmul(x, w1)
def f2(): return tf.matmul(x, w2)
y = tf.cond(tf.less(z, 0), f1, f2)

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        z: 10,
        w1: np.random.randn(D, H),
        w2: np.random.randn(D, H),
    }
    y_val = sess.run(y, feed_dict=values)
```



Static vs Dynamic: Loops

$$y_t = (y_{t-1} + x_t) * w$$



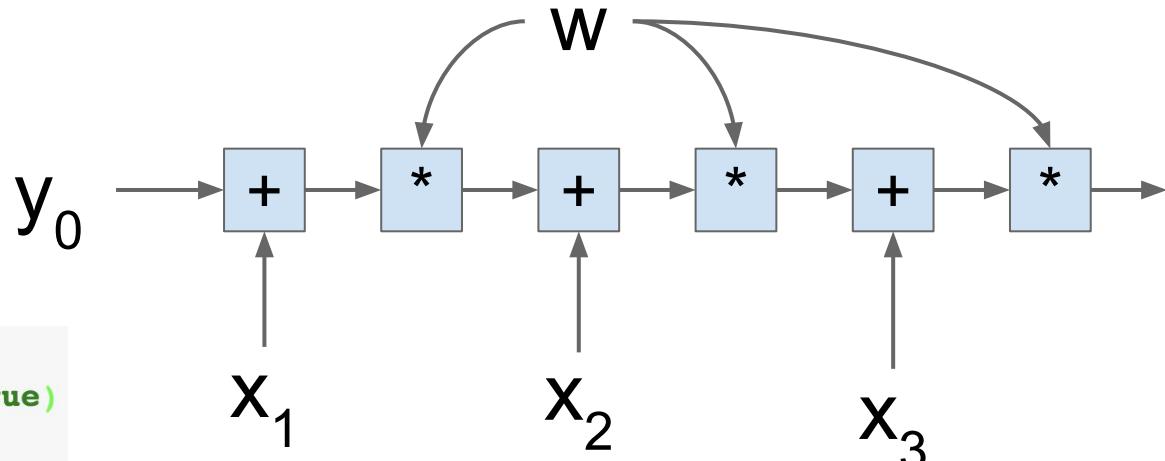
Static vs Dynamic: Loops

$$y_t = (y_{t-1} + x_t) * w$$

PyTorch: Normal Python

```
T, D = 3, 4
y0 = torch.randn(D, requires_grad=True)
x = torch.randn(T, D)
w = torch.randn(D)

y = [y0]
for t in range(T):
    prev_y = y[-1]
    next_y = (prev_y + x[t]) * w
    y.append(next_y)
```



Static vs Dynamic: Loops

$$y_t = (y_{t-1} + x_t) * w$$

PyTorch: Normal Python

```
T, D = 3, 4
y0 = torch.randn(D, requires_grad=True)
x = torch.randn(T, D)
w = torch.randn(D)

y = [y0]
for t in range(T):
    prev_y = y[-1]
    next_y = (prev_y + x[t]) * w
    y.append(next_y)
```

TensorFlow: Special TF control flow

```
T, N, D = 3, 4, 5
x = tf.placeholder(tf.float32, shape=(T, D))
y0 = tf.placeholder(tf.float32, shape=(D,))
w = tf.placeholder(tf.float32, shape=(D,))

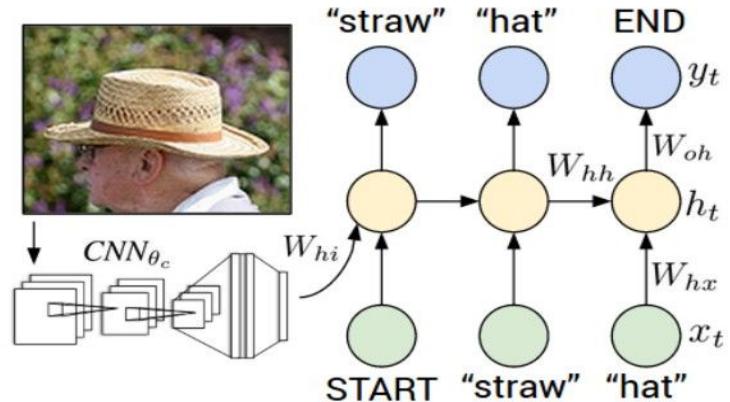
def f(prev_y, cur_x):
    return (prev_y + cur_x) * w

y = tf.foldl(f, x, y0)

with tf.Session() as sess:
    values = {
        x: np.random.randn(T, D),
        y0: np.random.randn(D),
        w: np.random.randn(D),
    }
    y_val = sess.run(y, feed_dict=values)
```

Dynamic Graph Applications

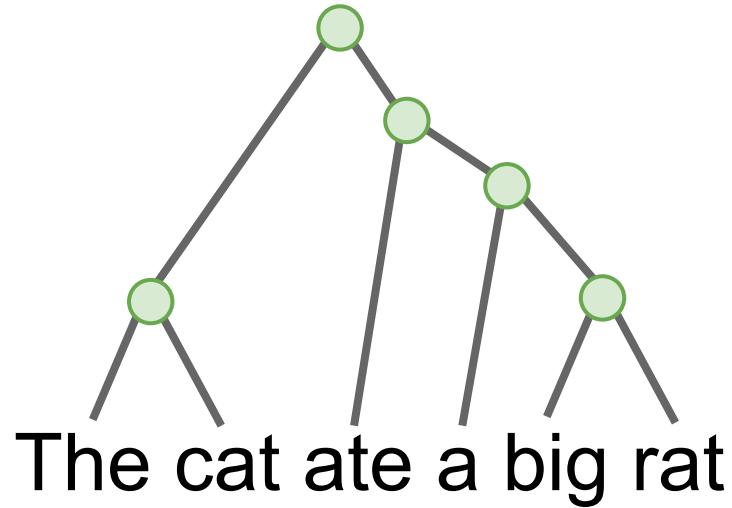
- Recurrent networks



Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

Dynamic Graph Applications

- Recurrent networks
- Recursive networks



Dynamic Graph Applications

- Recurrent networks
- Recursive networks
- Modular Networks

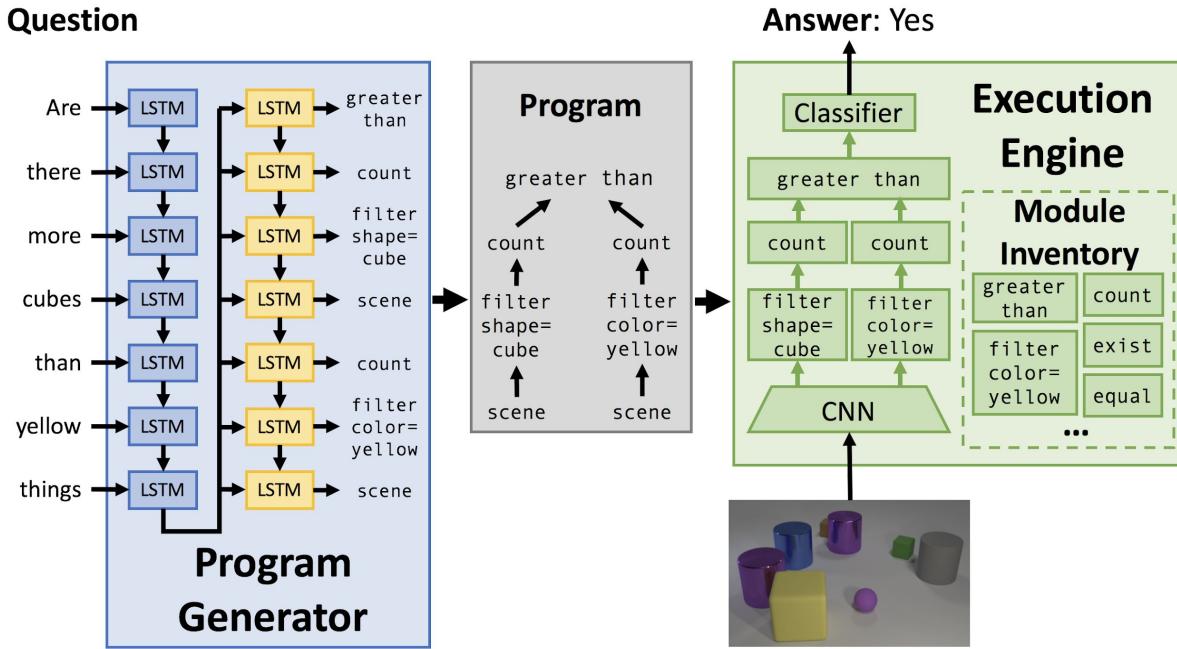


Figure copyright Justin Johnson, 2017. Reproduced with permission.

Andreas et al, "Neural Module Networks", CVPR 2016

Andreas et al, "Learning to Compose Neural Networks for Question Answering", NAACL 2016

Johnson et al, "Inferring and Executing Programs for Visual Reasoning", ICCV 2017

Dynamic Graph Applications

- Recurrent networks
- Recursive networks
- Modular Networks
- (Your creative idea here)

PyTorch vs TensorFlow, Static vs Dynamic

PyTorch
Dynamic Graphs

TensorFlow
Static Graphs

PyTorch vs TensorFlow, Static vs Dynamic

PyTorch
Dynamic Graphs

TensorFlow
Static Graphs

Lines are blurring! PyTorch is adding static features, and TensorFlow is adding dynamic features.

Dynamic TensorFlow: Dynamic Batching

TensorFlow Fold make dynamic graphs easier in TensorFlow through **dynamic batching**

Looks et al, "Deep Learning with Dynamic Computation Graphs", ICLR 2017
<https://github.com/tensorflow/fold>

Dynamic TensorFlow: Eager Execution

TensorFlow 1.7 added
eager execution which
allows dynamic graphs!

```
import tensorflow as tf
import tensorflow.contrib.eager as tfe

tf.enable_eager_execution()

N, D = 3, 4
x = tfe.Variable(tf.random_normal((N, D)))
y = tfe.Variable(tf.random_normal((N, D)))
z = tfe.Variable(tf.random_normal((N, D)))
with tfe.GradientTape() as tape:
    a = x * z
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tape.gradient(c, [x, y, z])
print(grad_x)
```

Dynamic TensorFlow: Eager Execution

Enable eager mode at
the start of the program:
it's a global switch

```
import tensorflow as tf
import tensorflow.contrib.eager as tfe

tf.enable_eager_execution()

N, D = 3, 4
x = tfe.Variable(tf.random_normal((N, D)))
y = tfe.Variable(tf.random_normal((N, D)))
z = tfe.Variable(tf.random_normal((N, D)))
with tfe.GradientTape() as tape:
    a = x * z
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tape.gradient(c, [x, y, z])
print(grad_x)
```

Dynamic TensorFlow: Eager Execution

These calls to
tf.random_normal produce
concrete values! No need
for placeholders / sessions

Wrap values in a
tfe.Variable if we might
want to compute grads for
them

```
import tensorflow as tf
import tensorflow.contrib.eager as tfe

tf.enable_eager_execution()

N, D = 3, 4
x = tfe.Variable(tf.random_normal((N, D)))
y = tfe.Variable(tf.random_normal((N, D)))
z = tfe.Variable(tf.random_normal((N, D)))

with tfe.GradientTape() as tape:
    a = x * z
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tape.gradient(c, [x, y, z])
print(grad_x)
```

Dynamic TensorFlow: Eager Execution

Operations scoped under a GradientTape will build a dynamic graph, similar to PyTorch

```
import tensorflow as tf
import tensorflow.contrib.eager as tfe

tf.enable_eager_execution()

N, D = 3, 4
x = tfe.Variable(tf.random_normal((N, D)))
y = tfe.Variable(tf.random_normal((N, D)))
z = tfe.Variable(tf.random_normal((N, D)))

with tfe.GradientTape() as tape:
    a = x * z
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tape.gradient(c, [x, y, z])
print(grad_x)
```

Dynamic TensorFlow: Eager Execution

```
import tensorflow as tf
import tensorflow.contrib.eager as tfe

tf.enable_eager_execution()

N, D = 3, 4
x = tfe.Variable(tf.random_normal((N, D)))
y = tfe.Variable(tf.random_normal((N, D)))
z = tfe.Variable(tf.random_normal((N, D)))
with tfe.GradientTape() as tape:
    a = x * z
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tape.gradient(c, [x, y, z])
print(grad_x)
```

Use the tape to compute gradients, like .backward() in PyTorch. The print statement works!

Dynamic TensorFlow: Eager Execution

Eager execution still pretty new, not fully supported in all TensorFlow APIs

Try it out!

```
import tensorflow as tf
import tensorflow.contrib.eager as tfe

tf.enable_eager_execution()

N, D = 3, 4
x = tfe.Variable(tf.random_normal((N, D)))
y = tfe.Variable(tf.random_normal((N, D)))
z = tfe.Variable(tf.random_normal((N, D)))
with tfe.GradientTape() as tape:
    a = x * z
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tape.gradient(c, [x, y, z])
print(grad_x)
```

Static PyTorch: Caffe2 <https://caffe2.ai/>

- Deep learning framework developed by Facebook
- Static graphs, somewhat similar to TensorFlow
- Core written in C++
- Nice Python interface
- Can train model in Python, then serialize and deploy without Python
- Works on iOS / Android, etc

Static PyTorch: ONNX Support

ONNX is an open-source standard for neural network models

Goal: Make it easy to train a network in one framework, then run it in another framework

Supported by PyTorch, Caffe2, Microsoft CNTK, Apache MXNet

<https://github.com/onnx/onnx>

Static PyTorch: ONNX Support

You can export a PyTorch model to ONNX

Run the graph on a dummy input, and save the graph to a file

Will only work if your model doesn't actually make use of dynamic graph - must build same graph on every forward pass, no loops / conditionals

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))

dummy_input = torch.randn(N, D_in)
torch.onnx.export(model, dummy_input,
                  'model.proto',
                  verbose=True)
```

Static PyTorch: ONNX Support

```
graph(%0 : Float(64, 1000)
      %1 : Float(100, 1000)
      %2 : Float(100)
      %3 : Float(10, 100)
      %4 : Float(10)) {
    %5 : Float(64, 100) =
    onnx::Gemm[alpha=1, beta=1, broadcast=1,
    transB=1](%0, %1, %2), scope:
    Sequential/Linear[0]
    %6 : Float(64, 100) = onnx::Relu(%5),
    scope: Sequential/ReLU[1]
    %7 : Float(64, 10) = onnx::Gemm[alpha=1,
    beta=1, broadcast=1, transB=1](%6, %3,
    %4), scope: Sequential/Linear[2]
    return (%7);
}
```

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))

dummy_input = torch.randn(N, D_in)
torch.onnx.export(model, dummy_input,
                  'model.proto',
                  verbose=True)
```

After exporting to ONNX, can run the PyTorch model in Caffe2

Static PyTorch: Future???

The screenshot shows a GitHub repository page for `pytorch / pytorch`. The top navigation bar includes links for `Pull requests`, `Issues`, `Marketplace`, and `Explore`. On the right side of the header are icons for notifications, a plus sign, and a user profile.

Key statistics displayed on the page include:

- `Watch`: 763
- `Unstar`: 14,502
- `Fork`: 3,233

Below the header, there are tabs for `Code`, `Issues 829`, `Pull requests 120`, `Projects 3`, `Wiki`, and `Insights`. The `Code` tab is currently selected.

A specific commit is highlighted:
Merge `caffe2` with `pytorch`.
By `ezyang` committed 27 days ago
2 parents `1e9a16c` + `eca84e2` commit `90afedb6e222d430d5c9333ff27adb42aa4bb900`

At the bottom of the commit details, it says "Showing 1,983 changed files with 369,779 additions and 20 deletions." with buttons for "Unified" and "Split".

<https://github.com/pytorch/pytorch/commit/90afedb6e222d430d5c9333ff27adb42aa4bb900>

PyTorch vs TensorFlow, Static vs Dynamic

PyTorch

Dynamic Graphs

Static: ONNX, Caffe2

TensorFlow

Static Graphs

Dynamic: Eager

My Advice:

PyTorch is my personal favorite. Clean API, dynamic graphs make it very easy to develop and debug. Can build model in PyTorch then export to Caffe2 with ONNX for production / mobile

TensorFlow is a safe bet for most projects. Not perfect but has huge community, wide usage. Can use same framework for research and production. Probably use a high-level framework. Only choice if you want to run on TPUs.

Next Time: CNN Architecture Case Studies

Lecture 9: CNN Architectures

Administrative

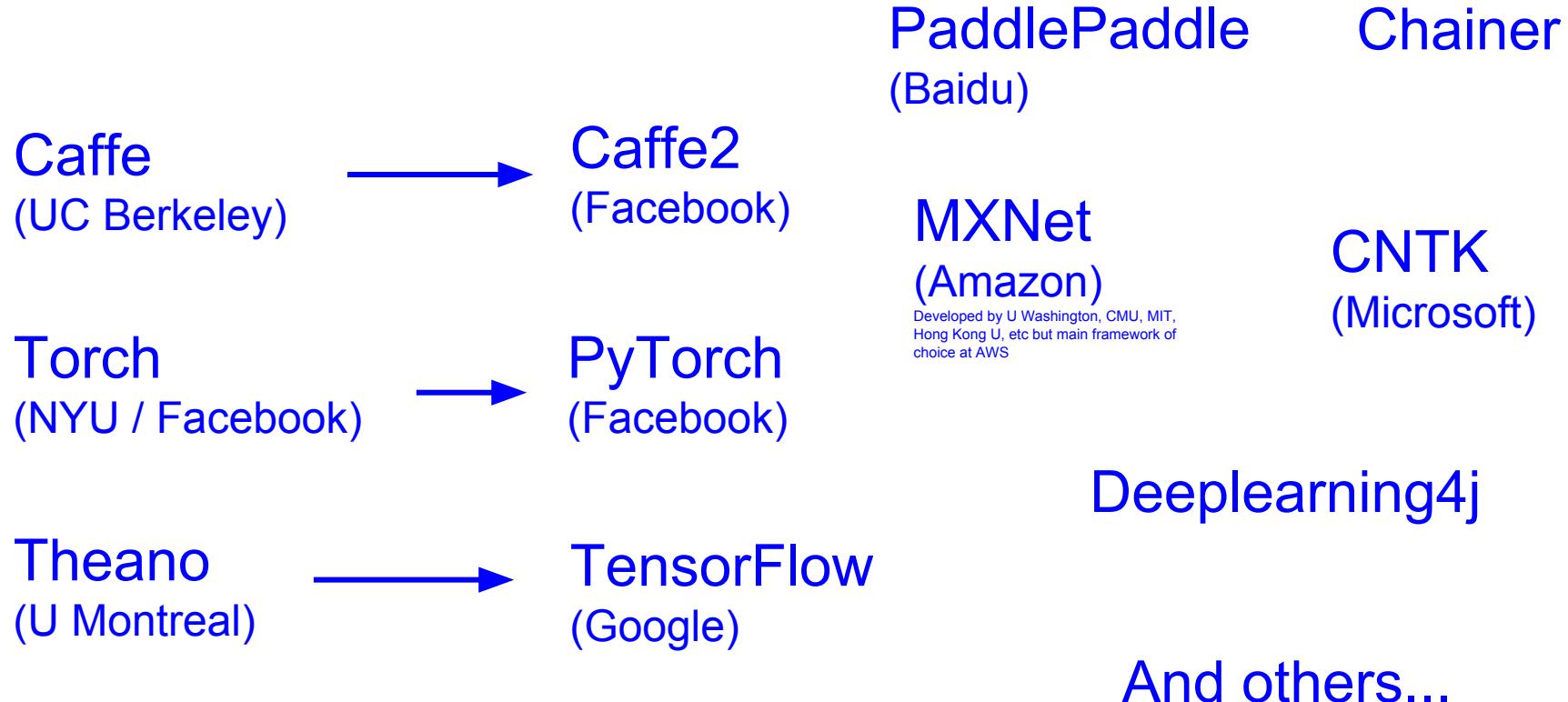
A2 due Wed May 2

Midterm: In-class Tue May 8. Covers material through Lecture 10 (Thu May 3).

Sample midterm released on piazza.

Midterm review session: Fri May 4 discussion section

Last time: Deep learning frameworks



Last time: Deep learning frameworks

- (1) Easily build big computational graphs
- (2) Easily compute gradients in computational graphs
- (3) Run it all efficiently on GPU (wrap cuDNN, cuBLAS, etc)

Today: CNN Architectures

Case Studies

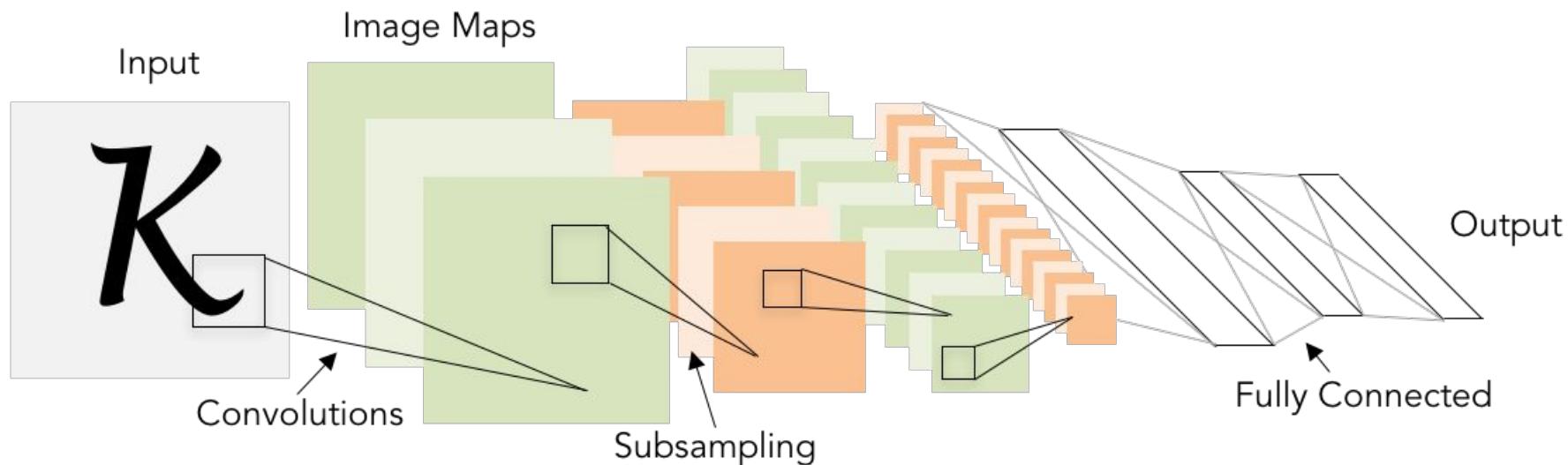
- AlexNet
- VGG
- GoogLeNet
- ResNet

Also....

- NiN (Network in Network)
- Wide ResNet
- ResNeXT
- Stochastic Depth
- Squeeze-and-Excitation Network
- DenseNet
- FractalNet
- SqueezeNet
- NASNet

Review: LeNet-5

[LeCun et al., 1998]



Conv filters were 5x5, applied at stride 1

Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-FC-FC]

Case Study: AlexNet

[Krizhevsky et al. 2012]

Architecture:

CONV1

MAX POOL1

NORM1

CONV2

MAX POOL2

NORM2

CONV3

CONV4

CONV5

Max POOL3

FC6

FC7

FC8

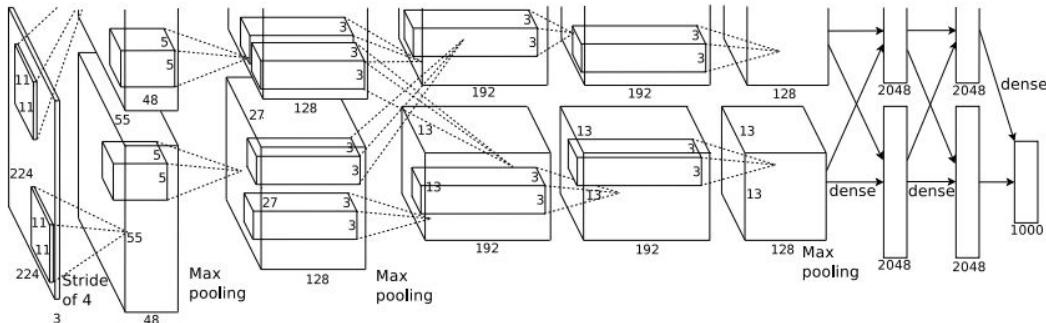
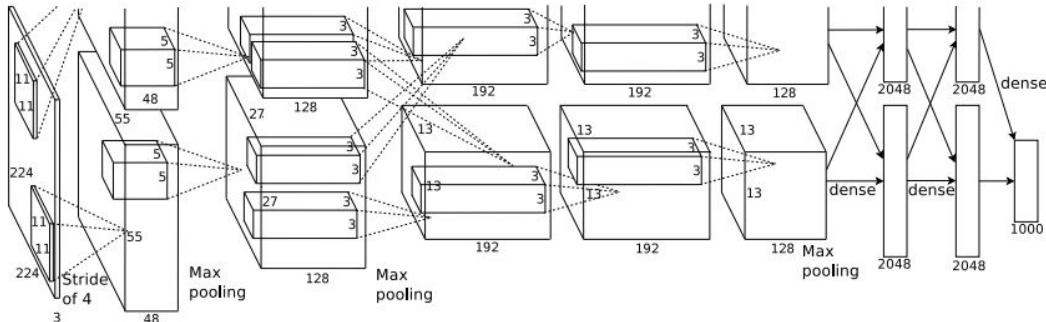


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4

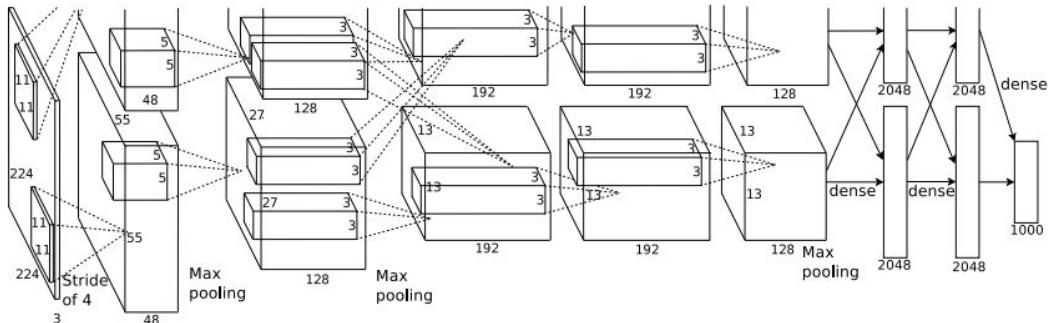
=>

Q: what is the output volume size? Hint: $(227-11)/4+1 = 55$

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

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4

=>

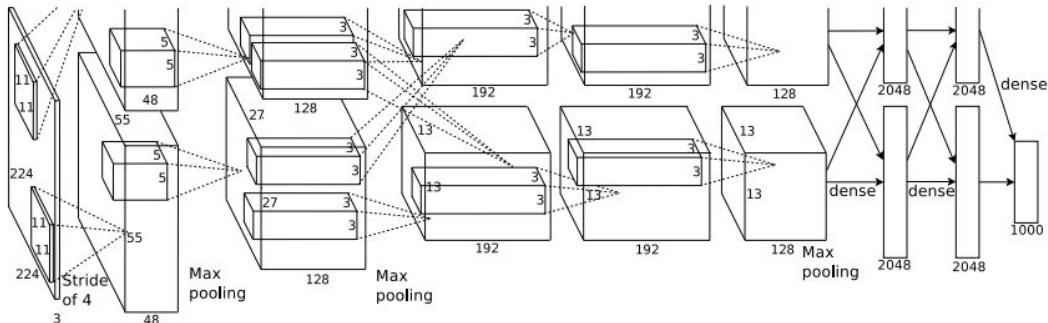
Output volume **[55x55x96]**

Q: What is the total number of parameters in this layer?

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

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4

=>

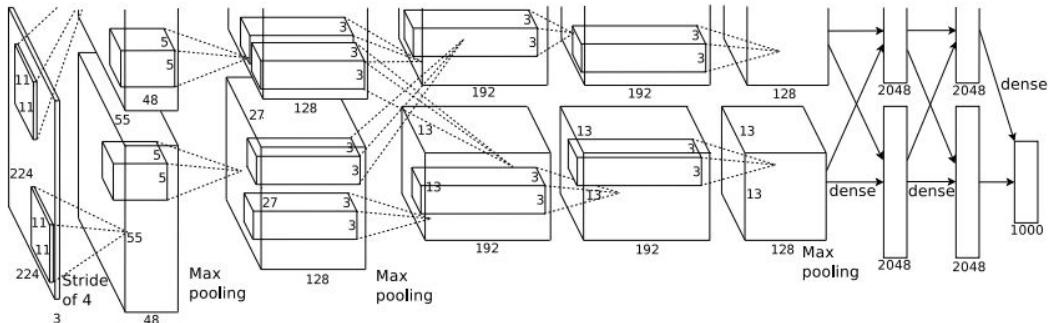
Output volume **[55x55x96]**

Parameters: $(11 \times 11 \times 3) \times 96 = 35\text{K}$

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

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

After CONV1: 55x55x96

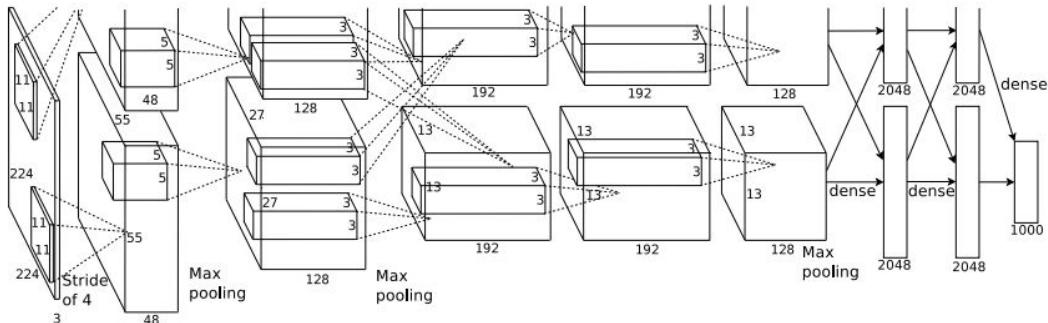
Second layer (POOL1): 3x3 filters applied at stride 2

Q: what is the output volume size? Hint: $(55-3)/2+1 = 27$

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

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

After CONV1: 55x55x96

Second layer (POOL1): 3x3 filters applied at stride 2

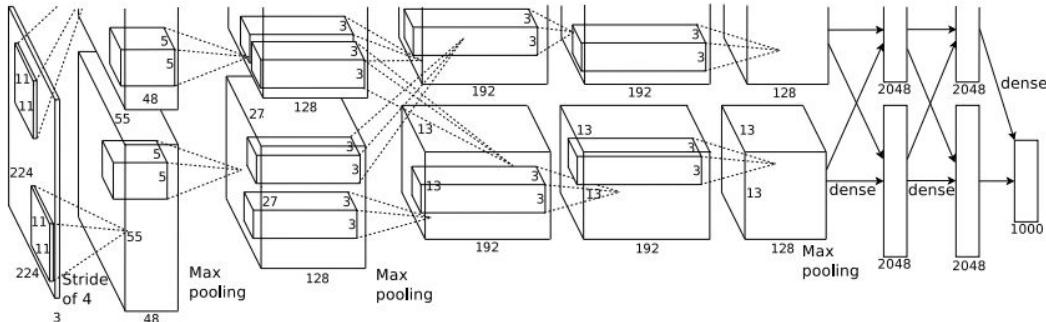
Output volume: 27x27x96

Q: what is the number of parameters in this layer?

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

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

After CONV1: 55x55x96

Second layer (POOL1): 3x3 filters applied at stride 2

Output volume: 27x27x96

Parameters: 0!

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

Case Study: AlexNet

[Krizhevsky et al. 2012]

Input: 227x227x3 images

After CONV1: 55x55x96

After POOL1: 27x27x96

...

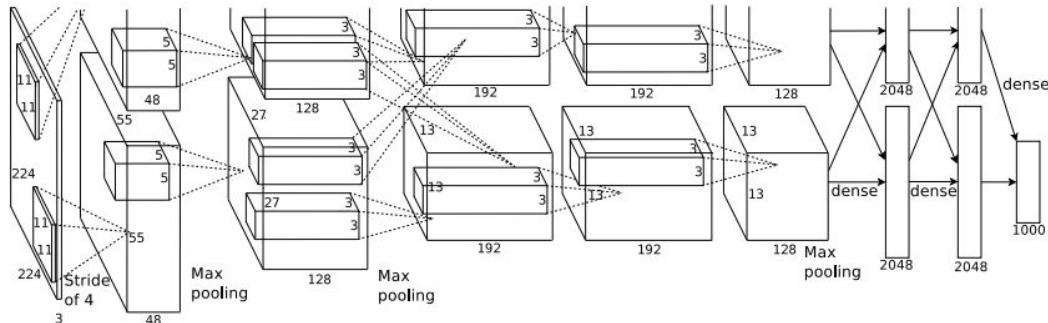


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

Case Study: AlexNet

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0

[27x27x96] MAX POOL1: 3x3 filters at stride 2

[27x27x96] NORM1: Normalization layer

[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2

[13x13x256] MAX POOL2: 3x3 filters at stride 2

[13x13x256] NORM2: Normalization layer

[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1

[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1

[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1

[6x6x256] MAX POOL3: 3x3 filters at stride 2

[4096] FC6: 4096 neurons

[4096] FC7: 4096 neurons

[1000] FC8: 1000 neurons (class scores)

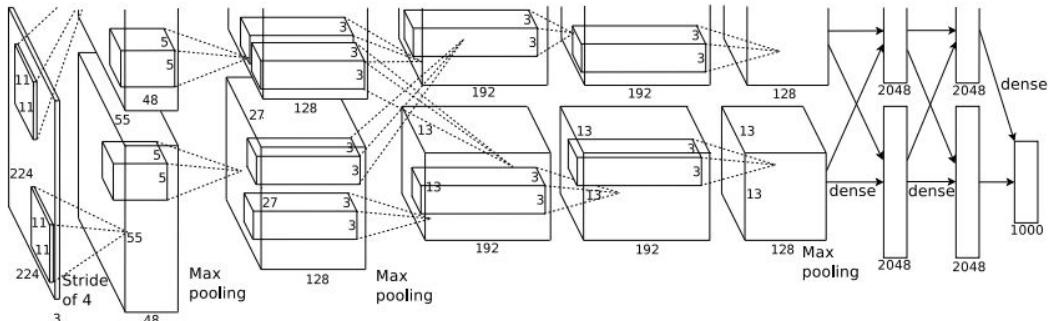


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

Case Study: AlexNet

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0

[27x27x96] MAX POOL1: 3x3 filters at stride 2

[27x27x96] NORM1: Normalization layer

[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2

[13x13x256] MAX POOL2: 3x3 filters at stride 2

[13x13x256] NORM2: Normalization layer

[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1

[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1

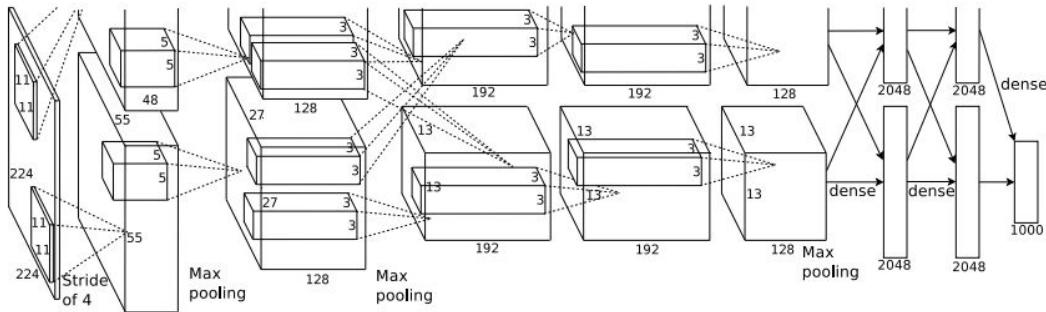
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1

[6x6x256] MAX POOL3: 3x3 filters at stride 2

[4096] FC6: 4096 neurons

[4096] FC7: 4096 neurons

[1000] FC8: 1000 neurons (class scores)



Details/Retrospectives:

- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

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

Case Study: AlexNet

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0

[27x27x96] MAX POOL1: 3x3 filters at stride 2

[27x27x96] NORM1: Normalization layer

[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2

[13x13x256] MAX POOL2: 3x3 filters at stride 2

[13x13x256] NORM2: Normalization layer

[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1

[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1

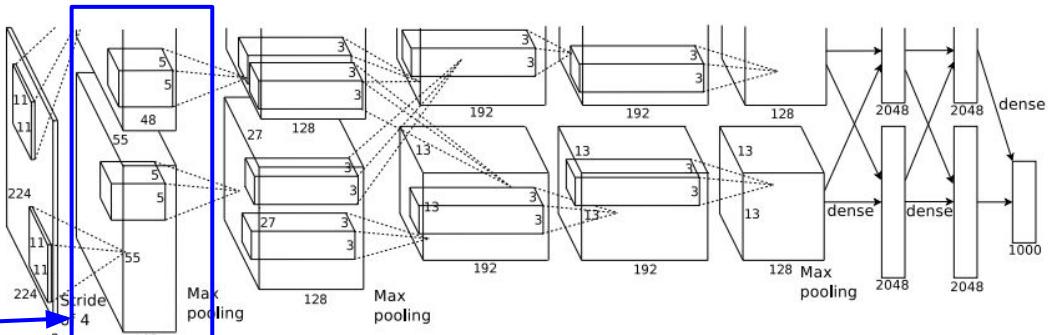
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1

[6x6x256] MAX POOL3: 3x3 filters at stride 2

[4096] FC6: 4096 neurons

[4096] FC7: 4096 neurons

[1000] FC8: 1000 neurons (class scores)



[55x55x48] x 2

Historical note: Trained on GTX 580 GPU with only 3 GB of memory. Network spread across 2 GPUs, half the neurons (feature maps) on each GPU.

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

Case Study: AlexNet

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0

[27x27x96] MAX POOL1: 3x3 filters at stride 2

[27x27x96] NORM1: Normalization layer

[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2

[13x13x256] MAX POOL2: 3x3 filters at stride 2

[13x13x256] NORM2: Normalization layer

[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1

[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1

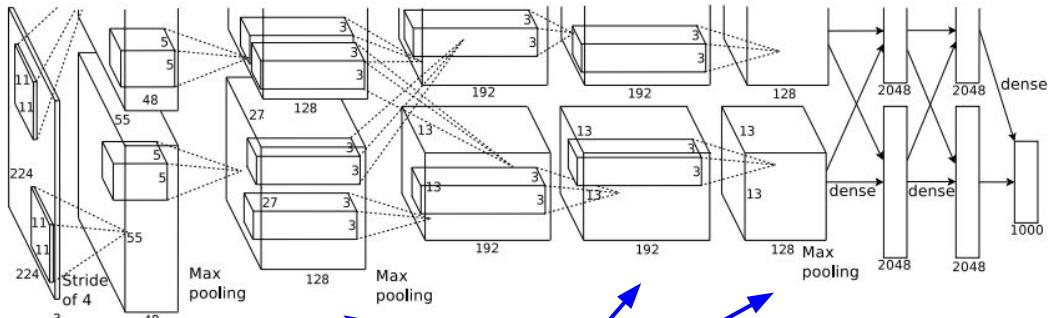
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1

[6x6x256] MAX POOL3: 3x3 filters at stride 2

[4096] FC6: 4096 neurons

[4096] FC7: 4096 neurons

[1000] FC8: 1000 neurons (class scores)



CONV1, CONV2, CONV4, CONV5:
Connections only with feature maps
on same GPU

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

Case Study: AlexNet

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0

[27x27x96] MAX POOL1: 3x3 filters at stride 2

[27x27x96] NORM1: Normalization layer

[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2

[13x13x256] MAX POOL2: 3x3 filters at stride 2

[13x13x256] NORM2: Normalization layer

[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1

[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1

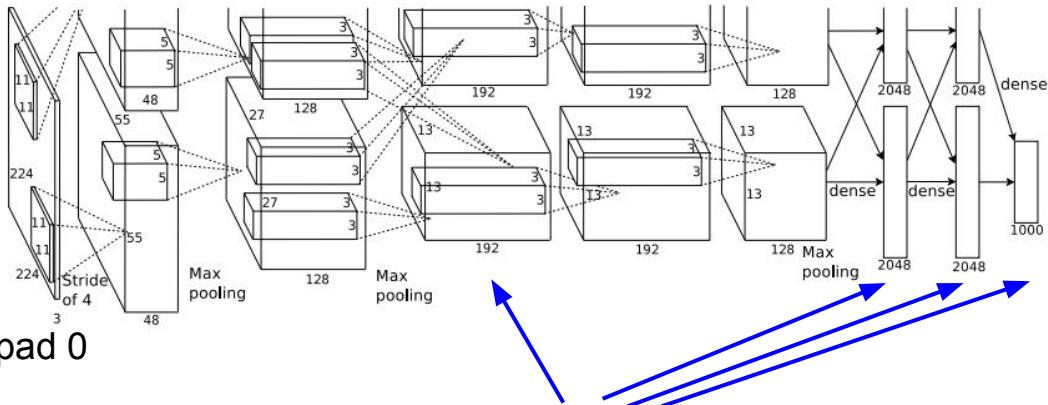
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1

[6x6x256] MAX POOL3: 3x3 filters at stride 2

[4096] FC6: 4096 neurons

[4096] FC7: 4096 neurons

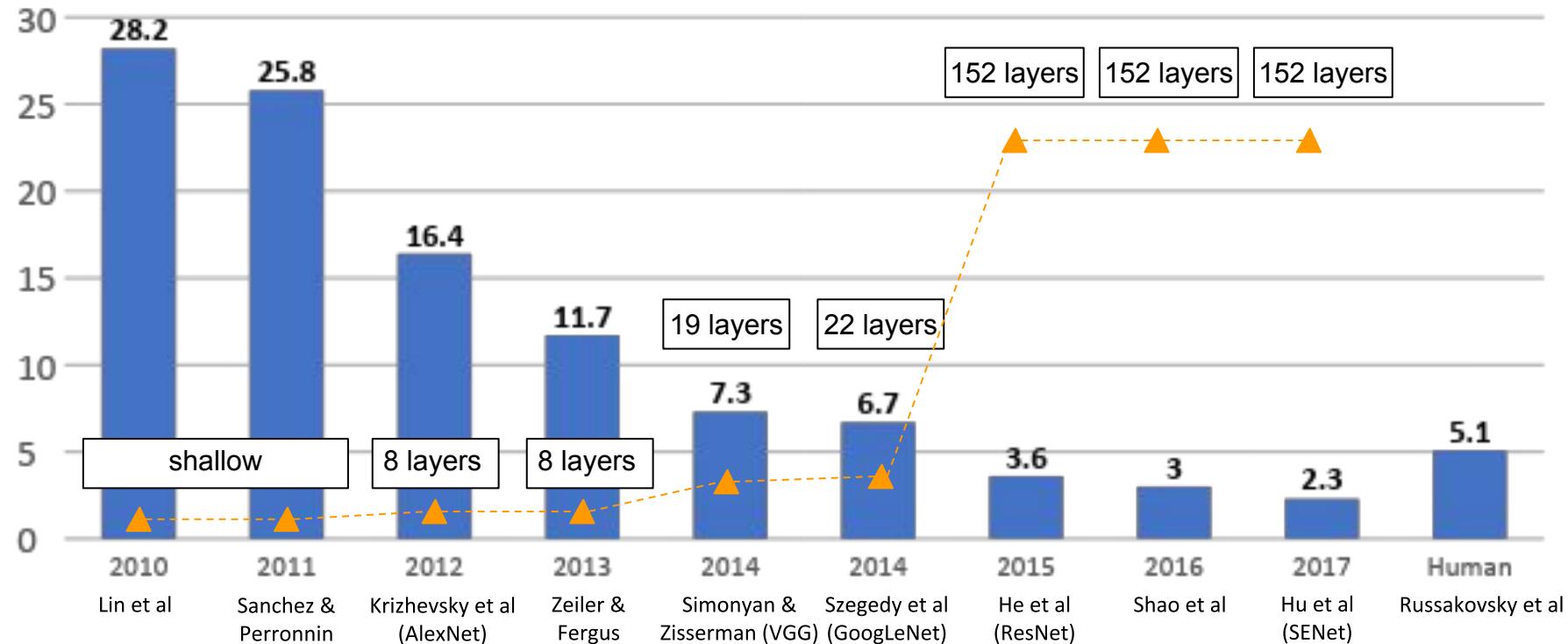
[1000] FC8: 1000 neurons (class scores)



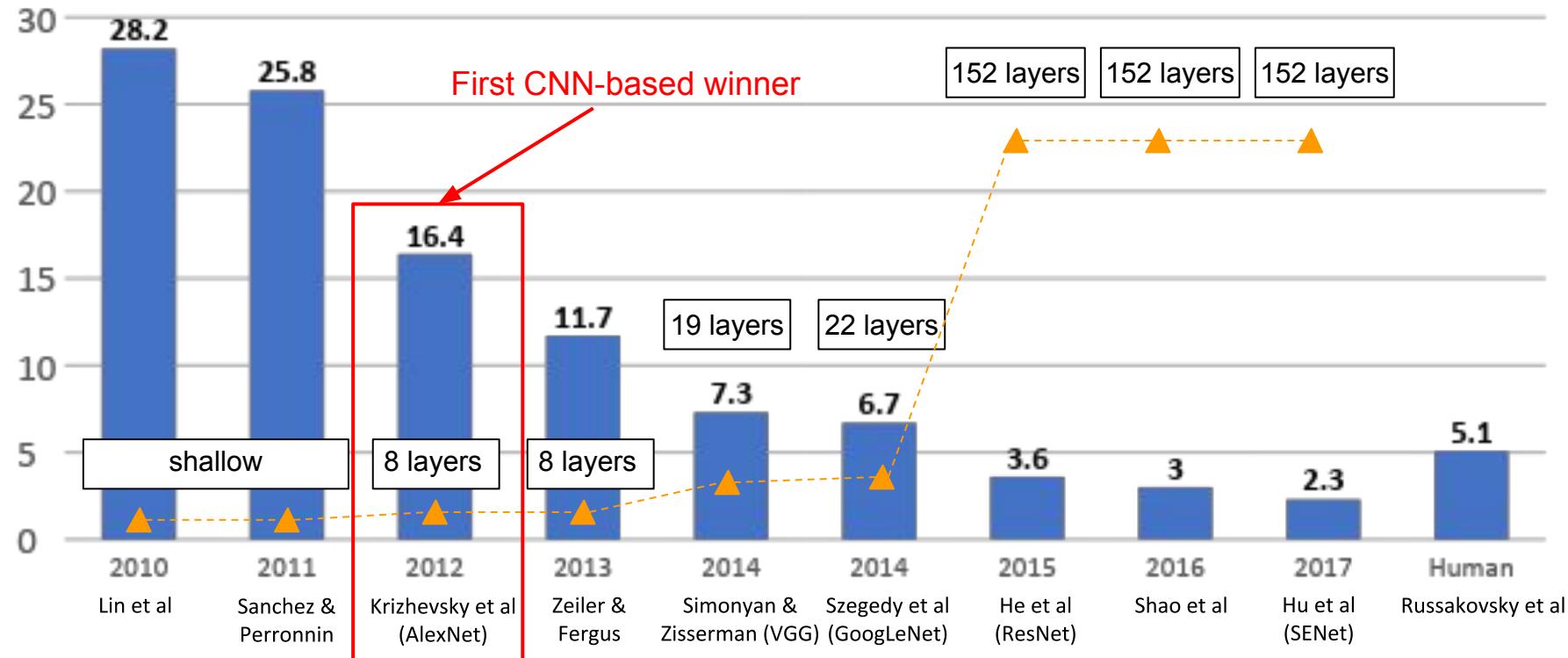
CONV3, FC6, FC7, FC8:
Connections with all feature maps in
preceding layer, communication
across GPUs

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

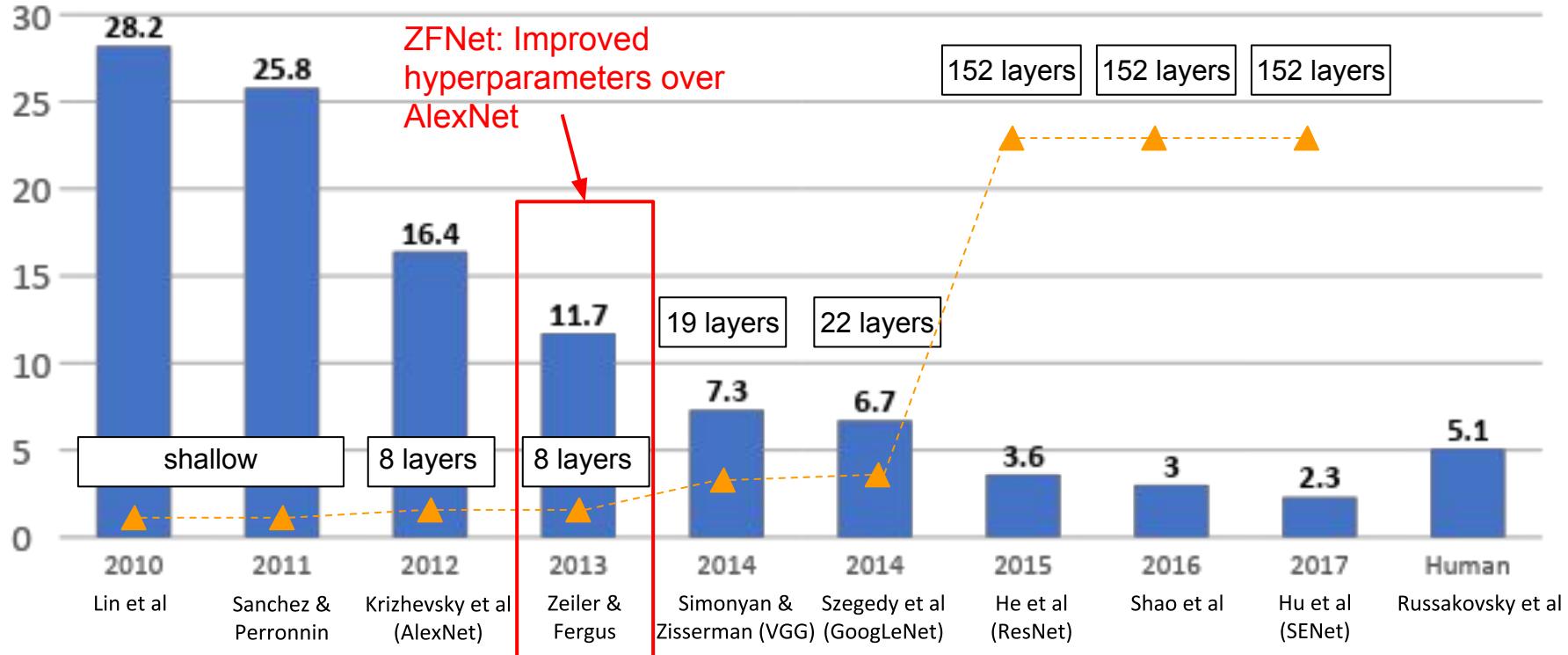
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

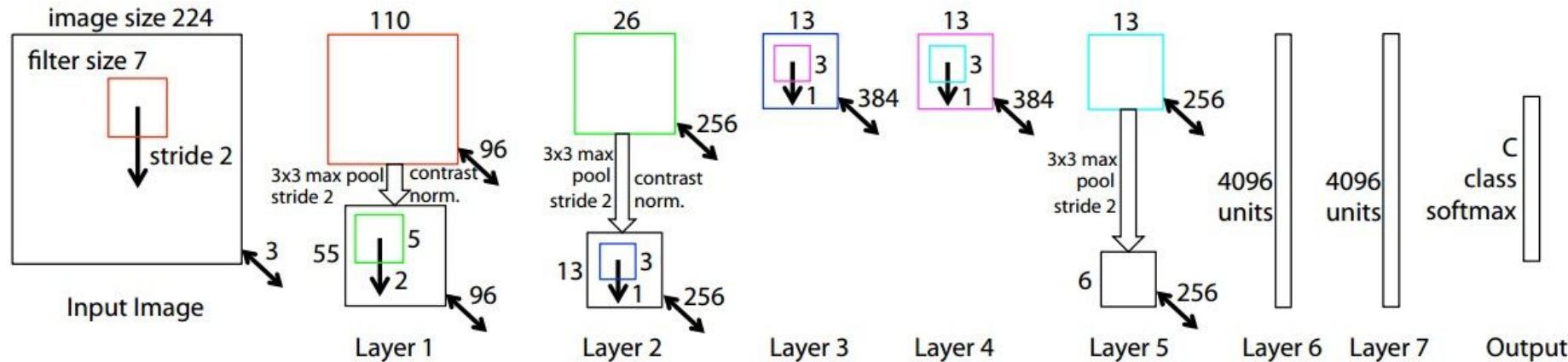


ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



ZFNet

[Zeiler and Fergus, 2013]



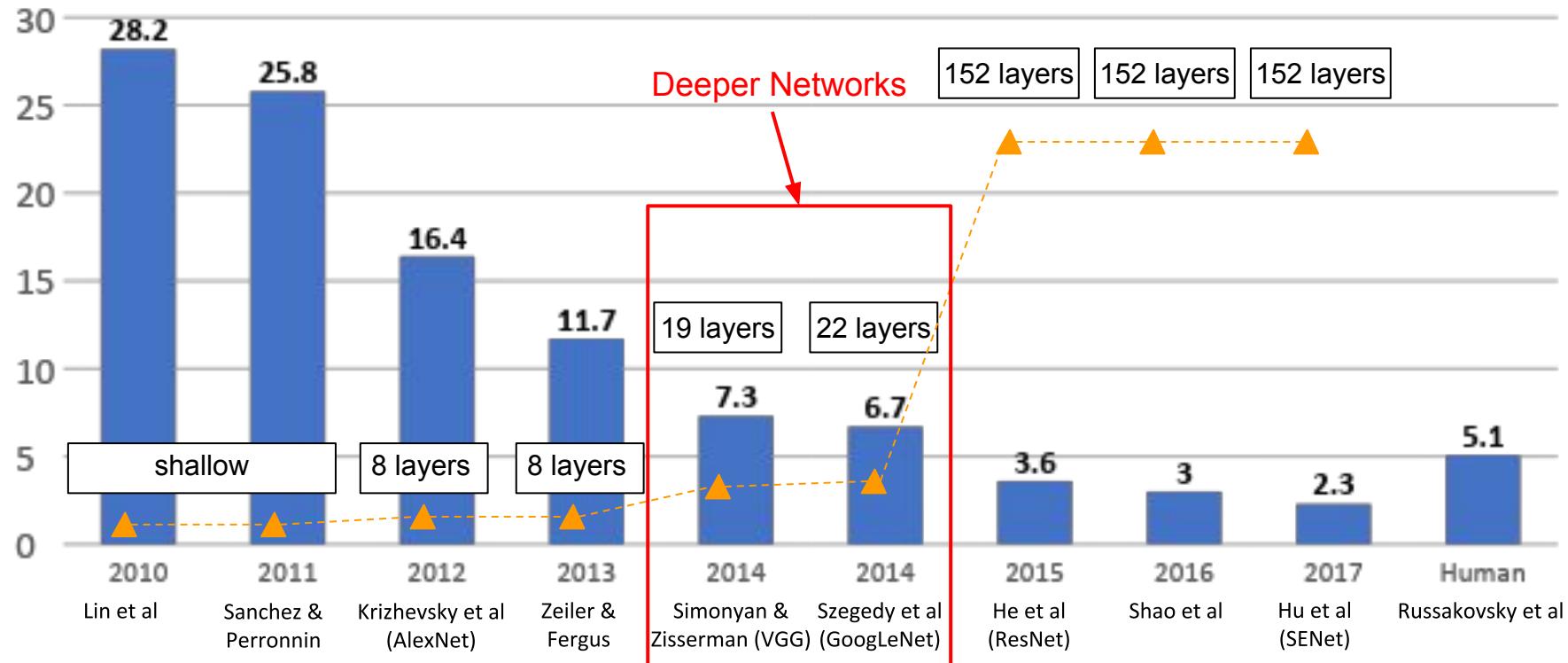
AlexNet but:

CONV1: change from (11x11 stride 4) to (7x7 stride 2)

CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512

ImageNet top 5 error: 16.4% \rightarrow 11.7%

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

8 layers (AlexNet)

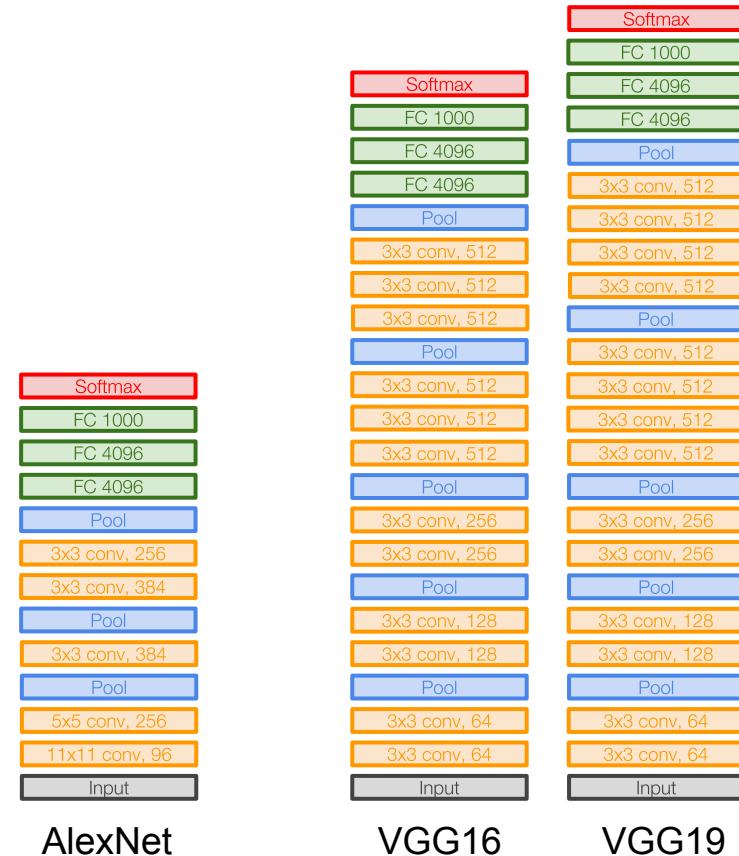
-> 16 - 19 layers (VGG16Net)

Only 3x3 CONV stride 1, pad 1
and 2x2 MAX POOL stride 2

11.7% top 5 error in ILSVRC'13

(ZFNet)

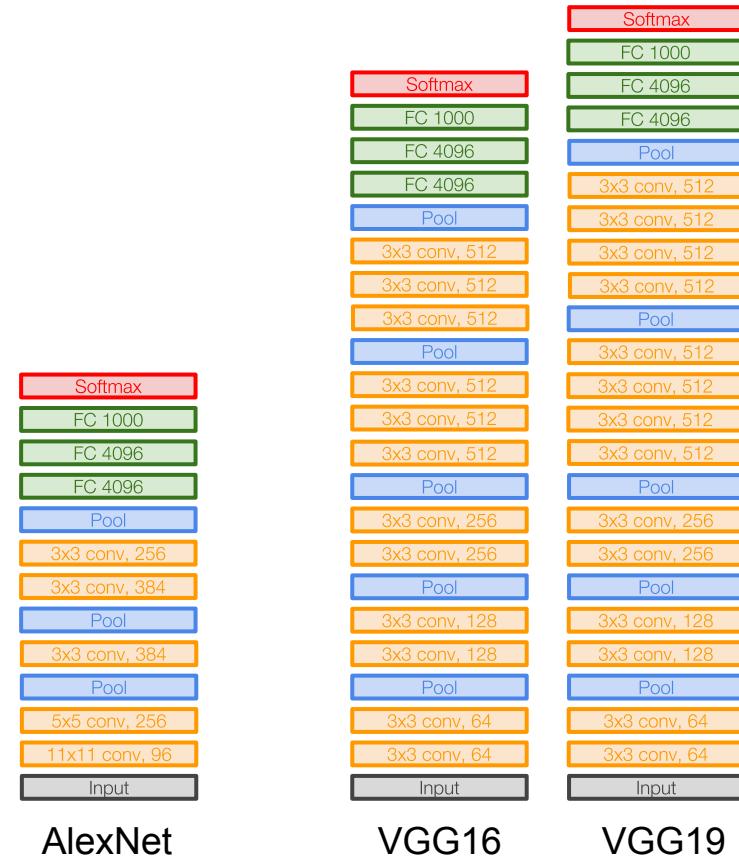
-> 7.3% top 5 error in ILSVRC'14



Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)



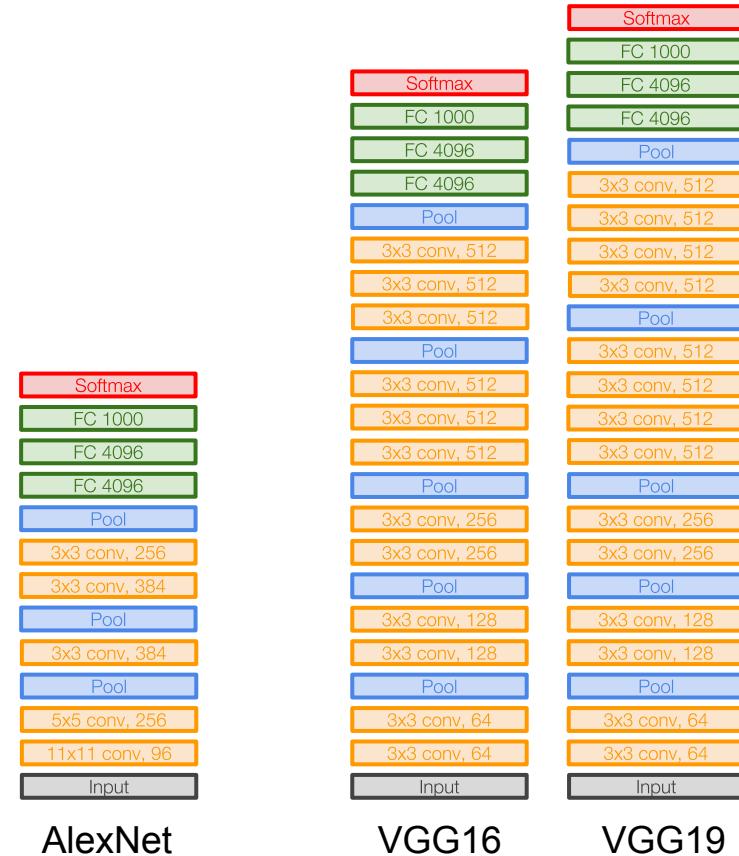
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers
has same **effective receptive field** as
one 7x7 conv layer

Q: What is the effective receptive field of
three 3x3 conv (stride 1) layers?



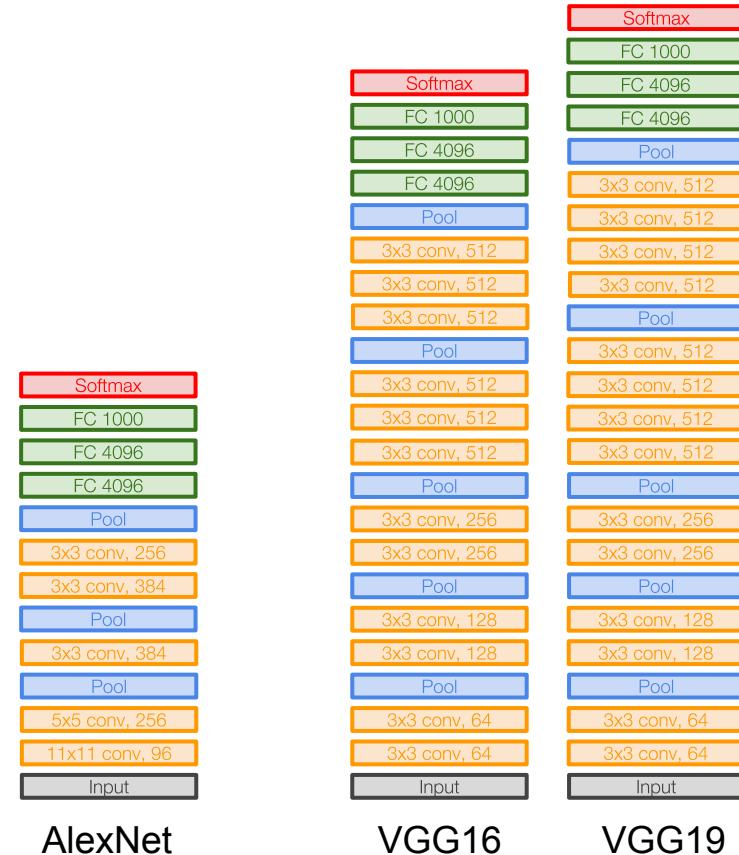
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers
has same **effective receptive field** as
one 7x7 conv layer

[7x7]



Case Study: VGGNet

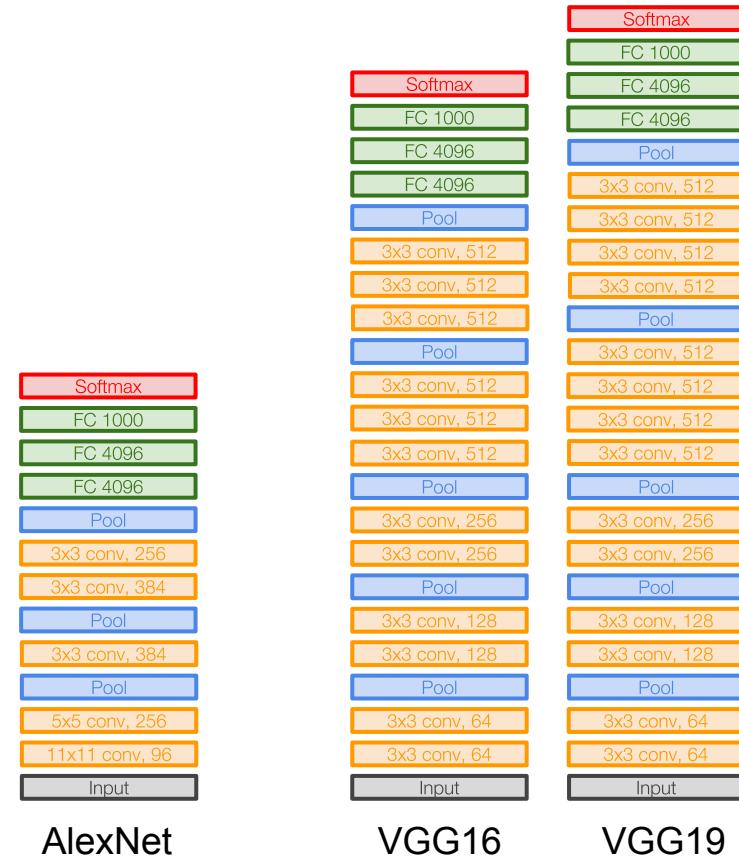
[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers
has same **effective receptive field** as
one 7x7 conv layer

But deeper, more non-linearities

And fewer parameters: $3 * (3^2 C^2)$ vs.
 $7^2 C^2$ for C channels per layer



INPUT: [224x224x3] memory: $224 \times 224 \times 3 = 150K$ params: 0 (not counting biases)

CONV3-64: [224x224x64] memory: $224 \times 224 \times 64 = 3.2M$ params: $(3 \times 3 \times 3) \times 64 = 1,728$

CONV3-64: [224x224x64] memory: $224 \times 224 \times 64 = 3.2M$ params: $(3 \times 3 \times 64) \times 64 = 36,864$

POOL2: [112x112x64] memory: $112 \times 112 \times 64 = 800K$ params: 0

CONV3-128: [112x112x128] memory: $112 \times 112 \times 128 = 1.6M$ params: $(3 \times 3 \times 64) \times 128 = 73,728$

CONV3-128: [112x112x128] memory: $112 \times 112 \times 128 = 1.6M$ params: $(3 \times 3 \times 128) \times 128 = 147,456$

POOL2: [56x56x128] memory: $56 \times 56 \times 128 = 400K$ params: 0

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 128) \times 256 = 294,912$

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 256) \times 256 = 589,824$

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 256) \times 256 = 589,824$

POOL2: [28x28x256] memory: $28 \times 28 \times 256 = 200K$ params: 0

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 256) \times 512 = 1,179,648$

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

POOL2: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: 0

CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

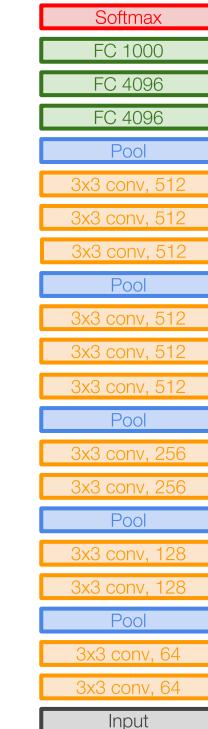
CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

POOL2: [7x7x512] memory: $7 \times 7 \times 512 = 25K$ params: 0

FC: [1x1x4096] memory: 4096 params: $7 \times 7 \times 512 \times 4096 = 102,760,448$

FC: [1x1x4096] memory: 4096 params: $4096 \times 4096 = 16,777,216$

FC: [1x1x1000] memory: 1000 params: $4096 \times 1000 = 4,096,000$



VGG16

INPUT: [224x224x3] memory: $224 \times 224 \times 3 = 150K$ params: 0 (not counting biases)

CONV3-64: [224x224x64] memory: $224 \times 224 \times 64 = 3.2M$ params: $(3 \times 3 \times 3) \times 64 = 1,728$

CONV3-64: [224x224x64] memory: $224 \times 224 \times 64 = 3.2M$ params: $(3 \times 3 \times 64) \times 64 = 36,864$

POOL2: [112x112x64] memory: $112 \times 112 \times 64 = 800K$ params: 0

CONV3-128: [112x112x128] memory: $112 \times 112 \times 128 = 1.6M$ params: $(3 \times 3 \times 64) \times 128 = 73,728$

CONV3-128: [112x112x128] memory: $112 \times 112 \times 128 = 1.6M$ params: $(3 \times 3 \times 128) \times 128 = 147,456$

POOL2: [56x56x128] memory: $56 \times 56 \times 128 = 400K$ params: 0

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 128) \times 256 = 294,912$

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 256) \times 256 = 589,824$

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 256) \times 256 = 589,824$

POOL2: [28x28x256] memory: $28 \times 28 \times 256 = 200K$ params: 0

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 256) \times 512 = 1,179,648$

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

POOL2: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: 0

CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

POOL2: [7x7x512] memory: $7 \times 7 \times 512 = 25K$ params: 0

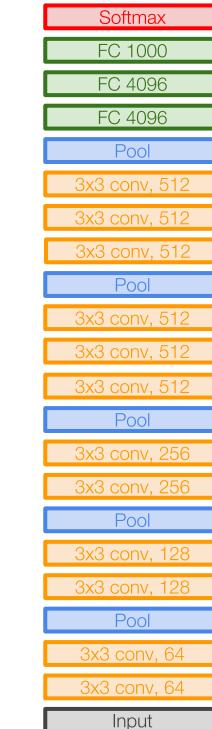
FC: [1x1x4096] memory: 4096 params: $7 \times 7 \times 512 \times 4096 = 102,760,448$

FC: [1x1x4096] memory: 4096 params: $4096 \times 4096 = 16,777,216$

FC: [1x1x1000] memory: 1000 params: $4096 \times 1000 = 4,096,000$

TOTAL memory: $24M \times 4 \text{ bytes} \approx 96\text{MB} / \text{image}$ (for a forward pass)

TOTAL params: 138M parameters



VGG16

INPUT: [224x224x3] memory: $224 \times 224 \times 3 = 150K$ params: 0 (not counting biases)

CONV3-64: [224x224x64] memory: $224 \times 224 \times 64 = 3.2M$ params: $(3 \times 3 \times 3) \times 64 = 1,728$

CONV3-64: [224x224x64] memory: $224 \times 224 \times 64 = 3.2M$ params: $(3 \times 3 \times 64) \times 64 = 36,864$

POOL2: [112x112x64] memory: $112 \times 112 \times 64 = 800K$ params: 0

CONV3-128: [112x112x128] memory: $112 \times 112 \times 128 = 1.6M$ params: $(3 \times 3 \times 64) \times 128 = 73,728$

CONV3-128: [112x112x128] memory: $112 \times 112 \times 128 = 1.6M$ params: $(3 \times 3 \times 128) \times 128 = 147,456$

POOL2: [56x56x128] memory: $56 \times 56 \times 128 = 400K$ params: 0

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 128) \times 256 = 294,912$

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 256) \times 256 = 589,824$

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 256) \times 256 = 589,824$

POOL2: [28x28x256] memory: $28 \times 28 \times 256 = 200K$ params: 0

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 256) \times 512 = 1,179,648$

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

POOL2: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: 0

CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

POOL2: [7x7x512] memory: $7 \times 7 \times 512 = 25K$ params: 0

FC: [1x1x4096] memory: 4096 params: $7 \times 7 \times 512 \times 4096 = 102,760,448$

FC: [1x1x4096] memory: 4096 params: $4096 \times 4096 = 16,777,216$

FC: [1x1x1000] memory: 1000 params: $4096 \times 1000 = 4,096,000$

Note:

Most memory is in early CONV

Most params are in late FC

TOTAL memory: $24M * 4 \text{ bytes} \approx 96\text{MB} / \text{image}$ (only forward! ~ 2 for bwd)

TOTAL params: 138M parameters

INPUT: [224x224x3] memory: $224 \times 224 \times 3 = 150K$ params: 0 (not counting biases)

CONV3-64: [224x224x64] memory: $224 \times 224 \times 64 = 3.2M$ params: $(3 \times 3 \times 3) \times 64 = 1,728$

CONV3-64: [224x224x64] memory: $224 \times 224 \times 64 = 3.2M$ params: $(3 \times 3 \times 64) \times 64 = 36,864$

POOL2: [112x112x64] memory: $112 \times 112 \times 64 = 800K$ params: 0

CONV3-128: [112x112x128] memory: $112 \times 112 \times 128 = 1.6M$ params: $(3 \times 3 \times 64) \times 128 = 73,728$

CONV3-128: [112x112x128] memory: $112 \times 112 \times 128 = 1.6M$ params: $(3 \times 3 \times 128) \times 128 = 147,456$

POOL2: [56x56x128] memory: $56 \times 56 \times 128 = 400K$ params: 0

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 128) \times 256 = 294,912$

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 256) \times 256 = 589,824$

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 256) \times 256 = 589,824$

POOL2: [28x28x256] memory: $28 \times 28 \times 256 = 200K$ params: 0

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 256) \times 512 = 1,179,648$

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

POOL2: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: 0

CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

POOL2: [7x7x512] memory: $7 \times 7 \times 512 = 25K$ params: 0

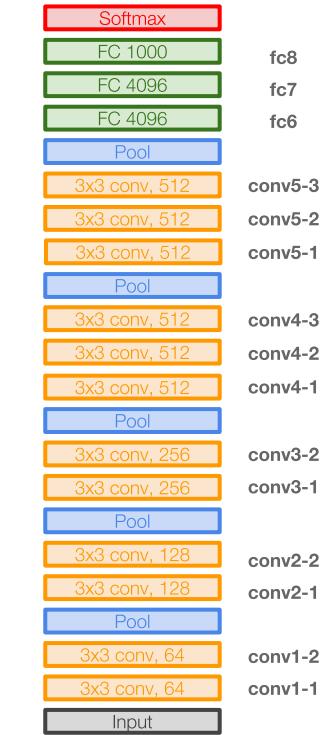
FC: [1x1x4096] memory: 4096 params: $7 \times 7 \times 512 \times 4096 = 102,760,448$

FC: [1x1x4096] memory: 4096 params: $4096 \times 4096 = 16,777,216$

FC: [1x1x1000] memory: 1000 params: $4096 \times 1000 = 4,096,000$

TOTAL memory: $24M * 4 \text{ bytes} \approx 96\text{MB} / \text{image}$ (only forward! ~ 2 for bwd)

TOTAL params: 138M parameters



VGG16

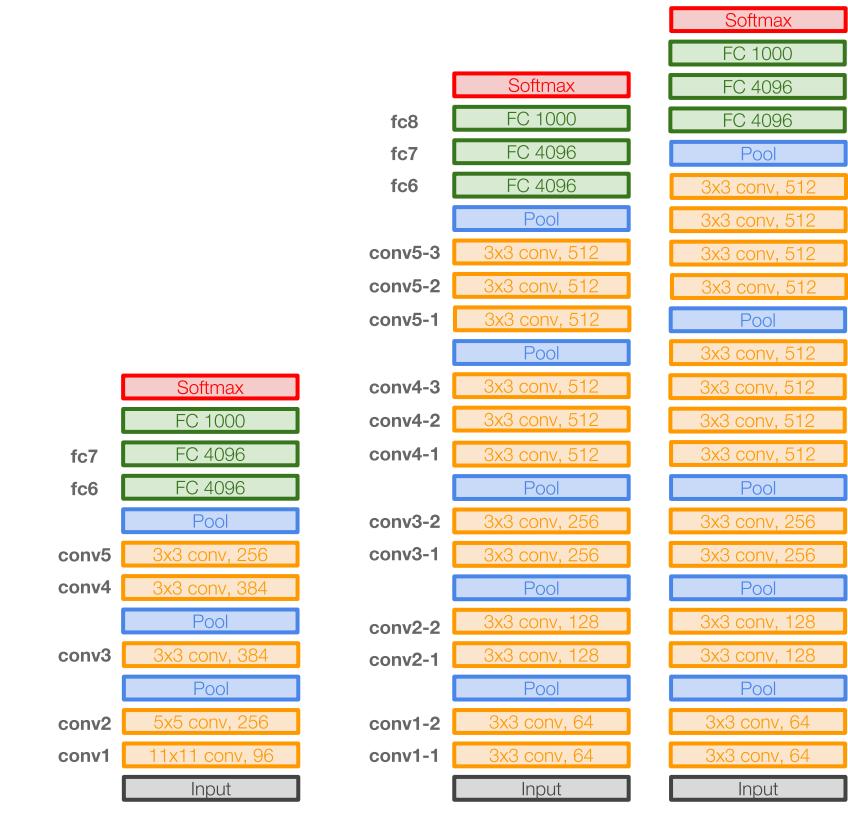
Common names

Case Study: VGGNet

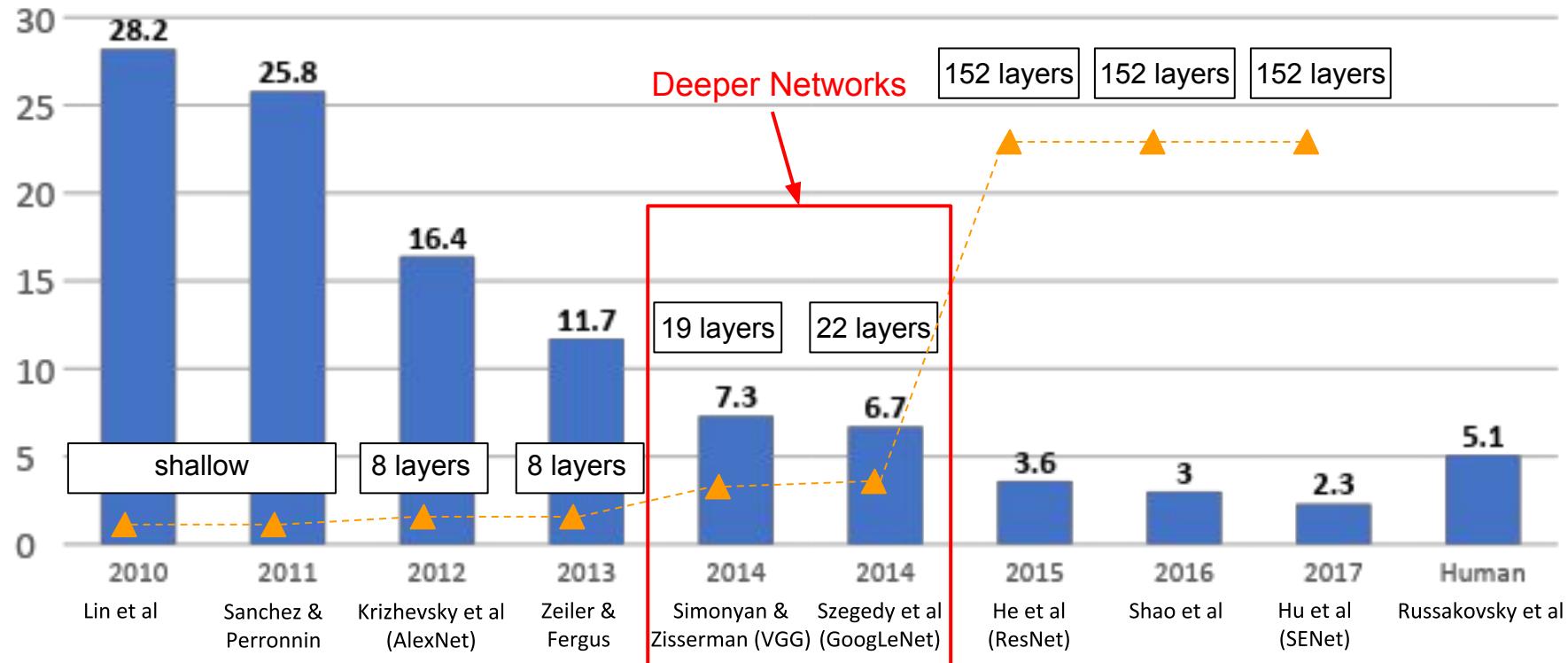
[Simonyan and Zisserman, 2014]

Details:

- ILSVRC'14 2nd in classification, 1st in localization
- Similar training procedure as Krizhevsky 2012
- No Local Response Normalisation (LRN)
- Use VGG16 or VGG19 (VGG19 only slightly better, more memory)
- Use ensembles for best results
- FC7 features generalize well to other tasks



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

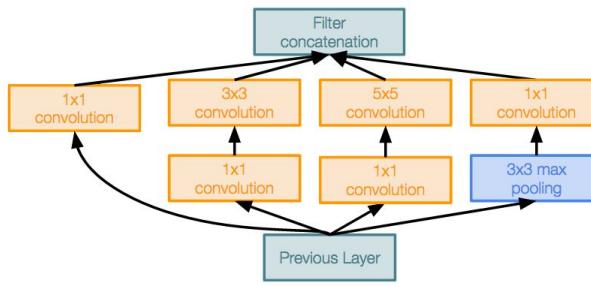


Case Study: GoogLeNet

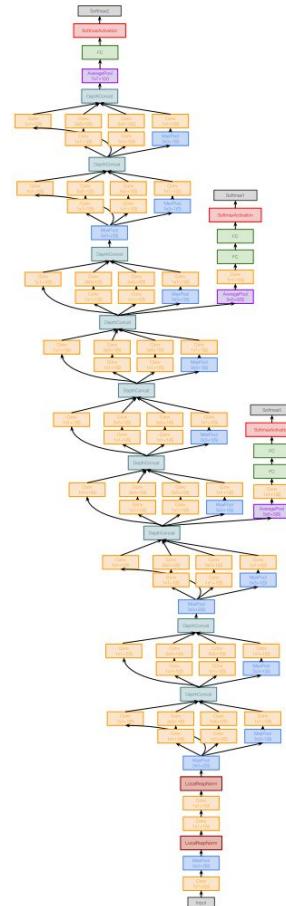
[Szegedy et al., 2014]

Deeper networks, with computational efficiency

- 22 layers
- Efficient “Inception” module
- No FC layers
- Only 5 million parameters!
12x less than AlexNet
- ILSVRC’14 classification winner
(6.7% top 5 error)



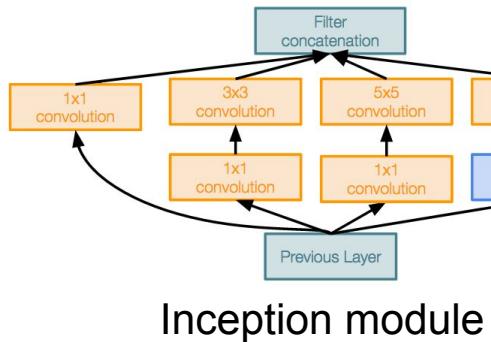
Inception module



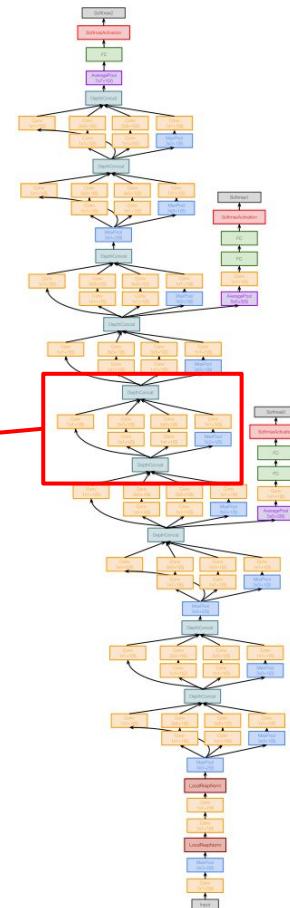
Case Study: GoogLeNet

[Szegedy et al., 2014]

“Inception module”: design a good local network topology (network within a network) and then stack these modules on top of each other

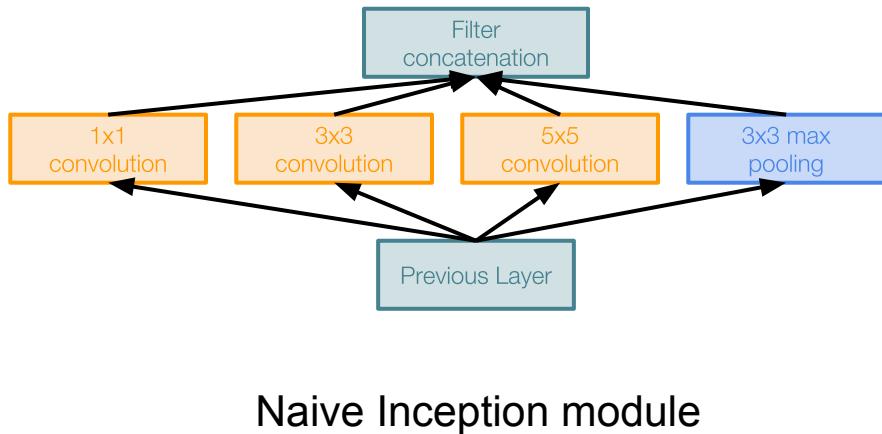


Inception module



Case Study: GoogLeNet

[Szegedy et al., 2014]



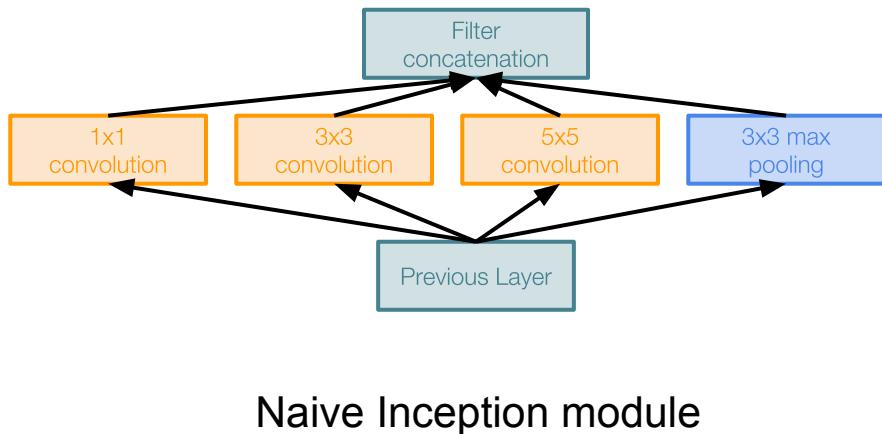
Apply parallel filter operations on the input from previous layer:

- Multiple receptive field sizes for convolution (1×1 , 3×3 , 5×5)
- Pooling operation (3×3)

Concatenate all filter outputs together depth-wise

Case Study: GoogLeNet

[Szegedy et al., 2014]



Apply parallel filter operations on the input from previous layer:

- Multiple receptive field sizes for convolution (1×1 , 3×3 , 5×5)
- Pooling operation (3×3)

Concatenate all filter outputs together depth-wise

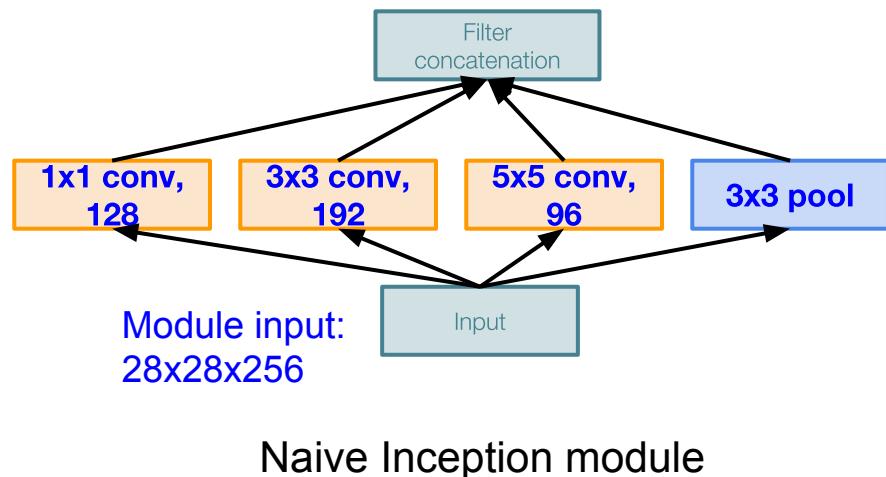
Q: What is the problem with this?
[Hint: Computational complexity]

Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:



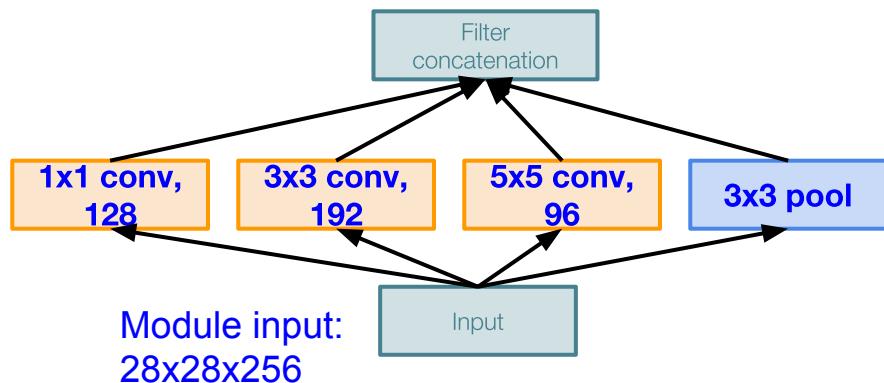
Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q1: What is the output size of the
1x1 conv, with 128 filters?

Q: What is the problem with this?
[Hint: Computational complexity]



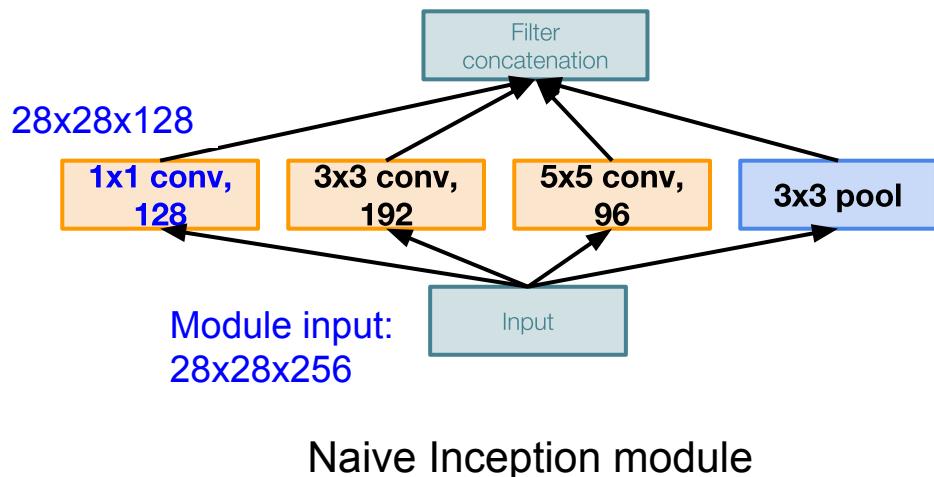
Naive Inception module

Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q1: What is the output size of the
1x1 conv, with 128 filters?



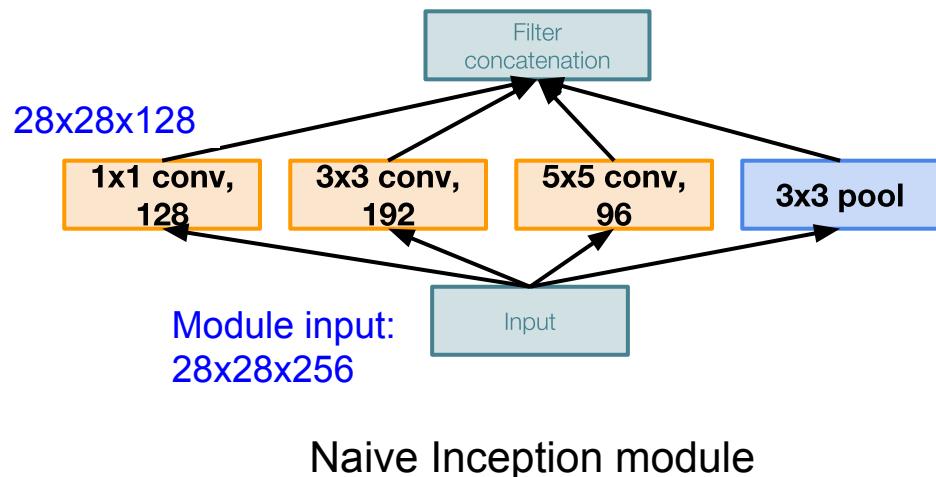
Q: What is the problem with this?
[Hint: Computational complexity]

Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q2: What are the output sizes of all different filter operations?



Q: What is the problem with this?
[Hint: Computational complexity]

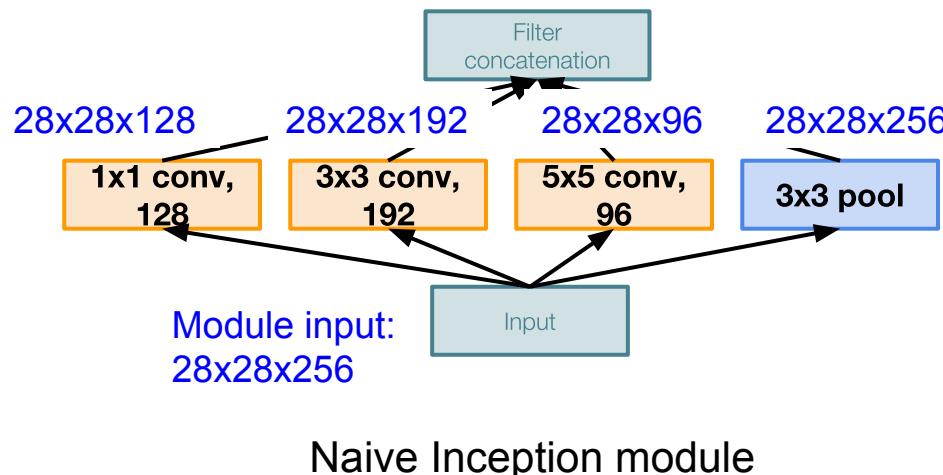
Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q2: What are the output sizes of all different filter operations?



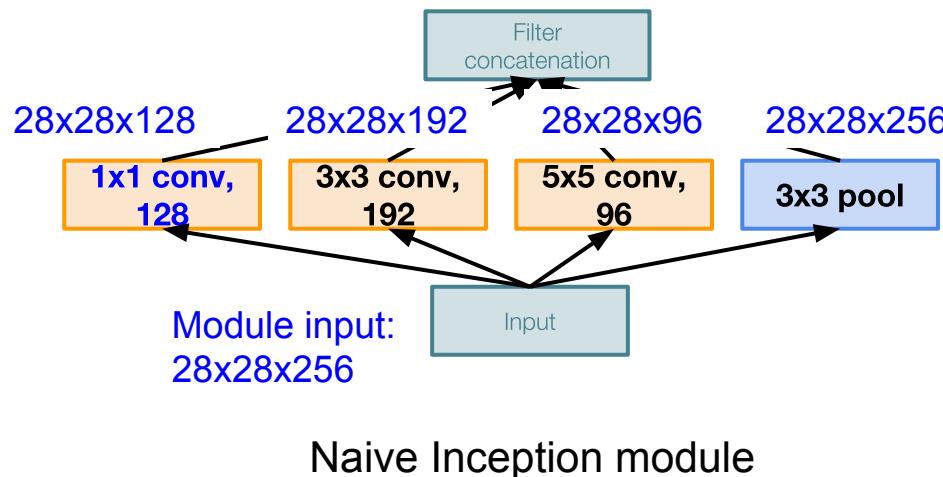
Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q3: What is output size after
filter concatenation?



Naive Inception module

Case Study: GoogLeNet

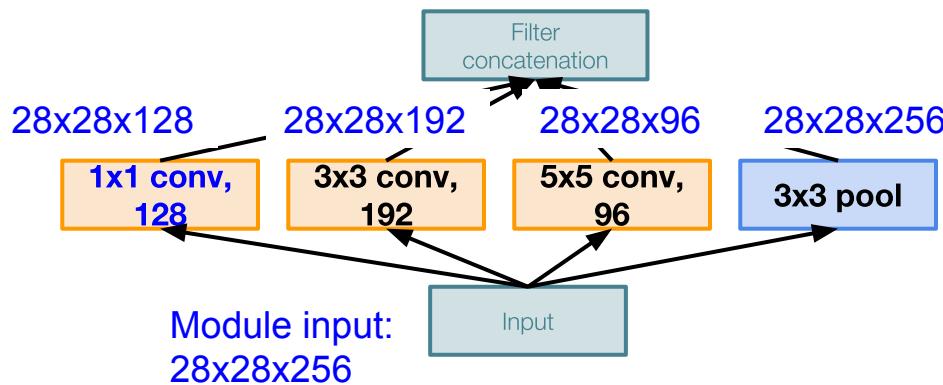
[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q3: What is output size after
filter concatenation?

$$28 \times 28 \times (128 + 192 + 96 + 256) = 28 \times 28 \times 672$$



Naive Inception module

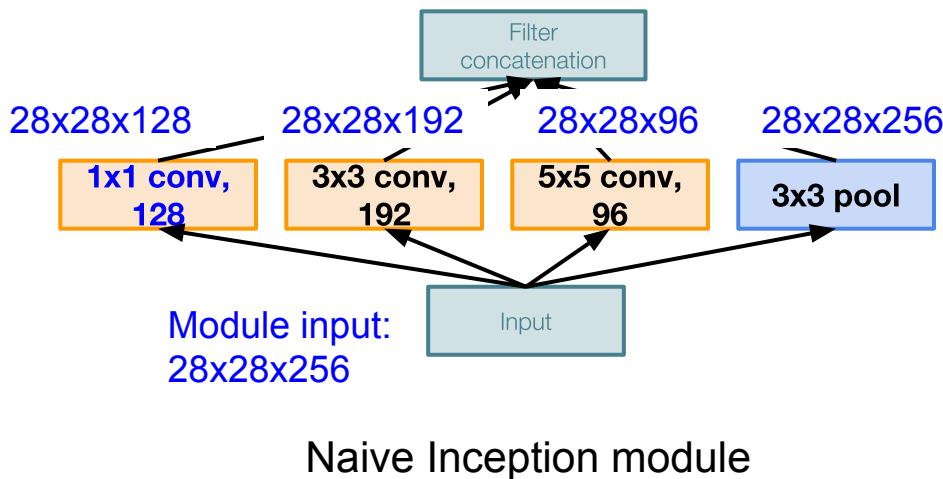
Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3: What is output size after filter concatenation?

$$28 \times 28 \times (128 + 192 + 96 + 256) = 28 \times 28 \times 672$$



Q: What is the problem with this?
[Hint: Computational complexity]

Conv Ops:

[1x1 conv, 128] $28 \times 28 \times 128 \times 1 \times 1 \times 256$

[3x3 conv, 192] $28 \times 28 \times 192 \times 3 \times 3 \times 256$

[5x5 conv, 96] $28 \times 28 \times 96 \times 5 \times 5 \times 256$

Total: 854M ops

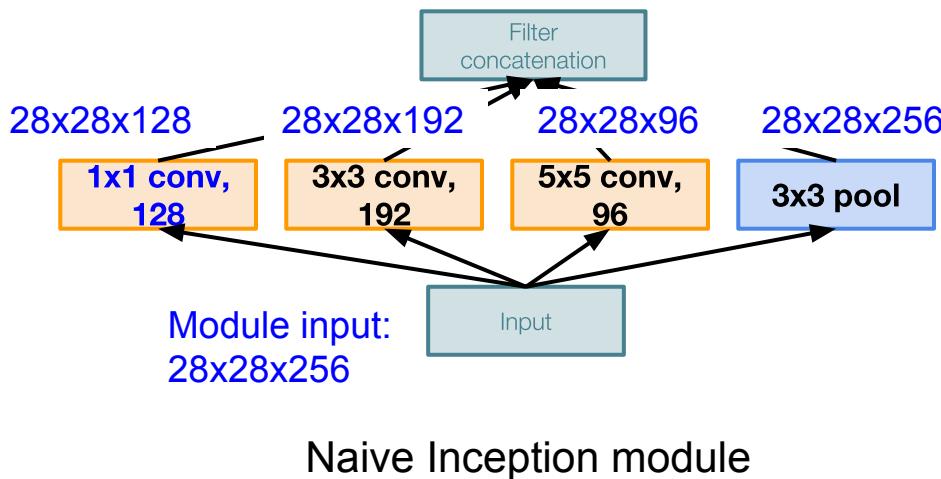
Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3: What is output size after filter concatenation?

$$28 \times 28 \times (128 + 192 + 96 + 256) = 28 \times 28 \times 672$$



Q: What is the problem with this?
[Hint: Computational complexity]

Conv Ops:

- [1x1 conv, 128] $28 \times 28 \times 128 \times 1 \times 1 \times 256$
- [3x3 conv, 192] $28 \times 28 \times 192 \times 3 \times 3 \times 256$
- [5x5 conv, 96] $28 \times 28 \times 96 \times 5 \times 5 \times 256$

Total: 854M ops

Very expensive compute

Pooling layer also preserves feature depth, which means total depth after concatenation can only grow at every layer!

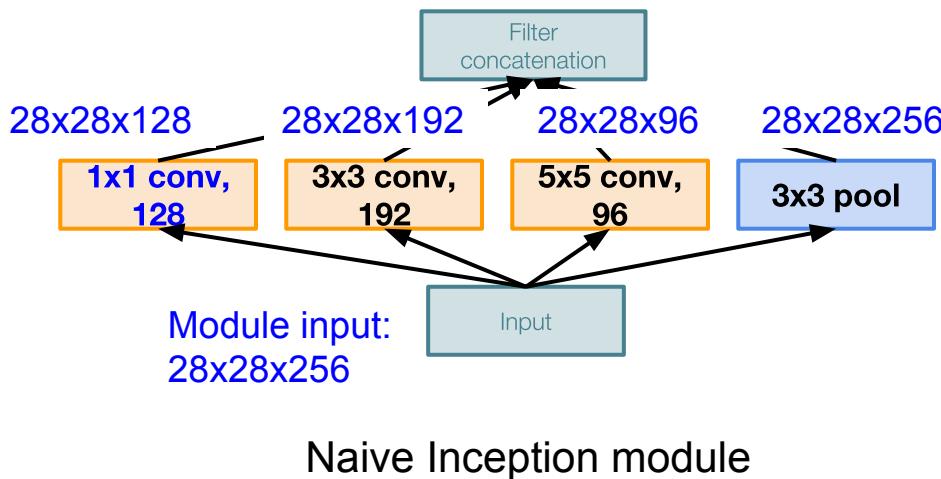
Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3: What is output size after filter concatenation?

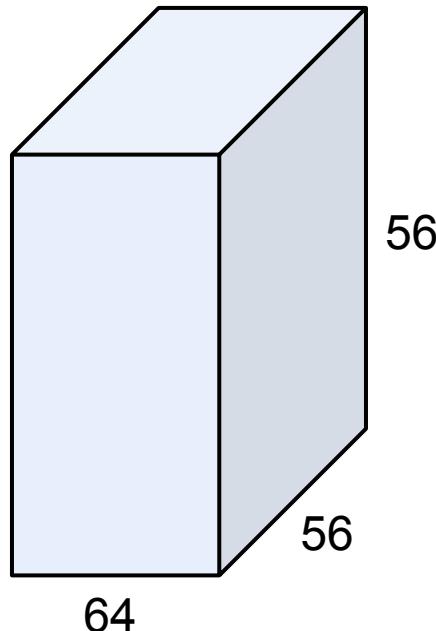
$$28 \times 28 \times (128 + 192 + 96 + 256) = 529k$$



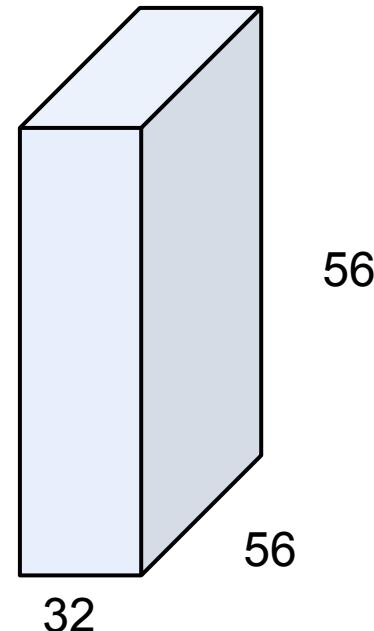
Q: What is the problem with this?
[Hint: Computational complexity]

Solution: “bottleneck” layers that use 1×1 convolutions to reduce feature depth

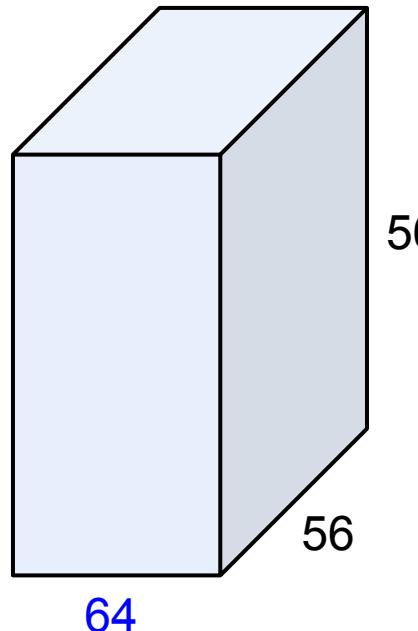
Reminder: 1x1 convolutions



1x1 CONV
with 32 filters
→
(each filter has size
 $1 \times 1 \times 64$, and performs a
64-dimensional dot
product)



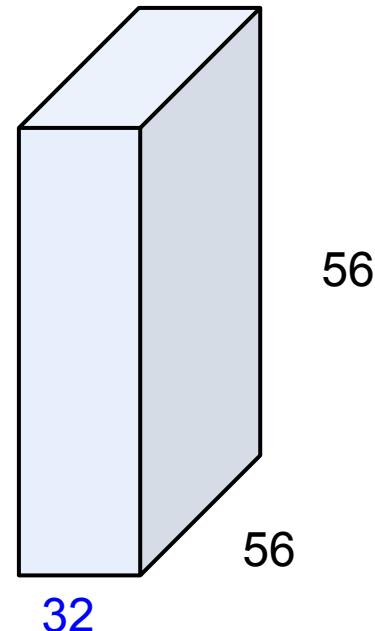
Reminder: 1x1 convolutions



1x1 CONV
with 32 filters

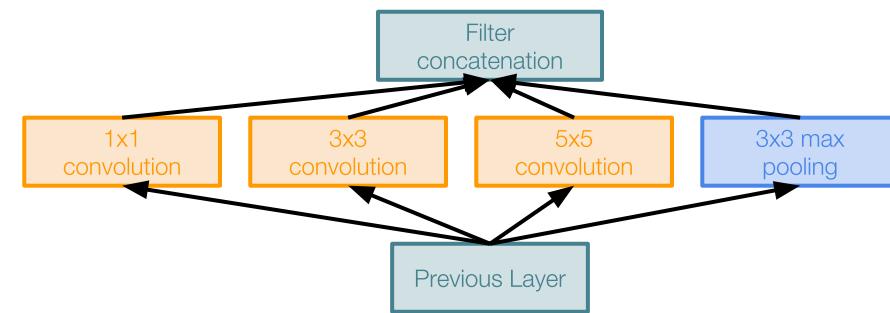
preserves spatial
dimensions, reduces depth!

Projects depth to lower
dimension (combination of
feature maps)

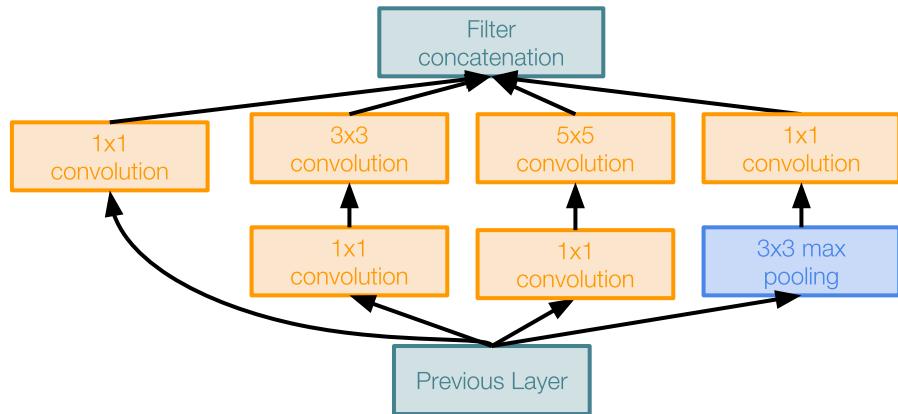


Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

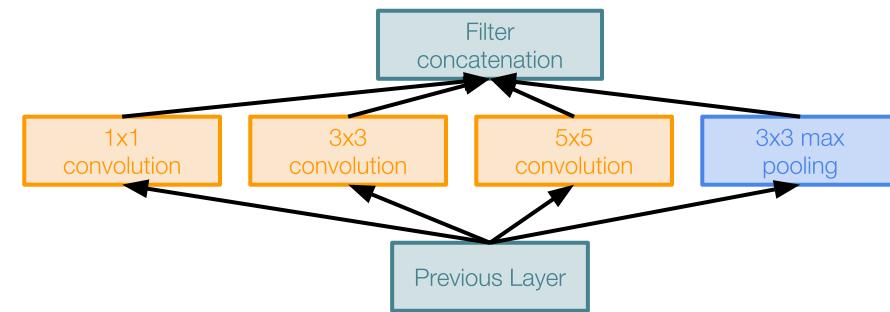


Inception module with dimension reduction

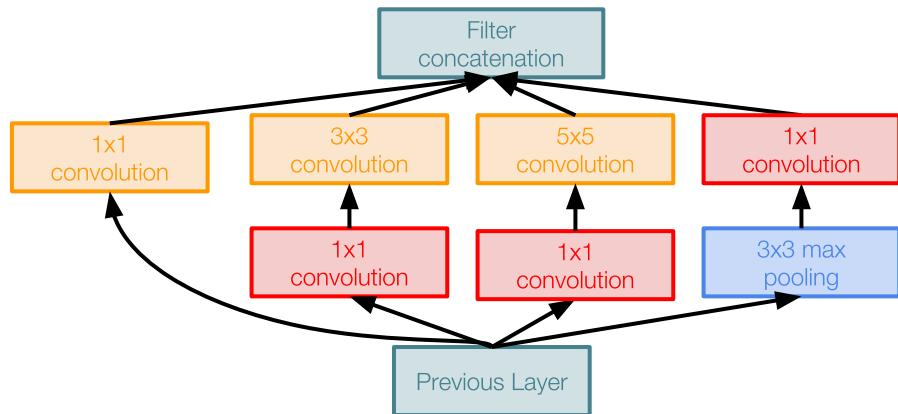
Case Study: GoogLeNet

[Szegedy et al., 2014]

1x1 conv “bottleneck”
layers



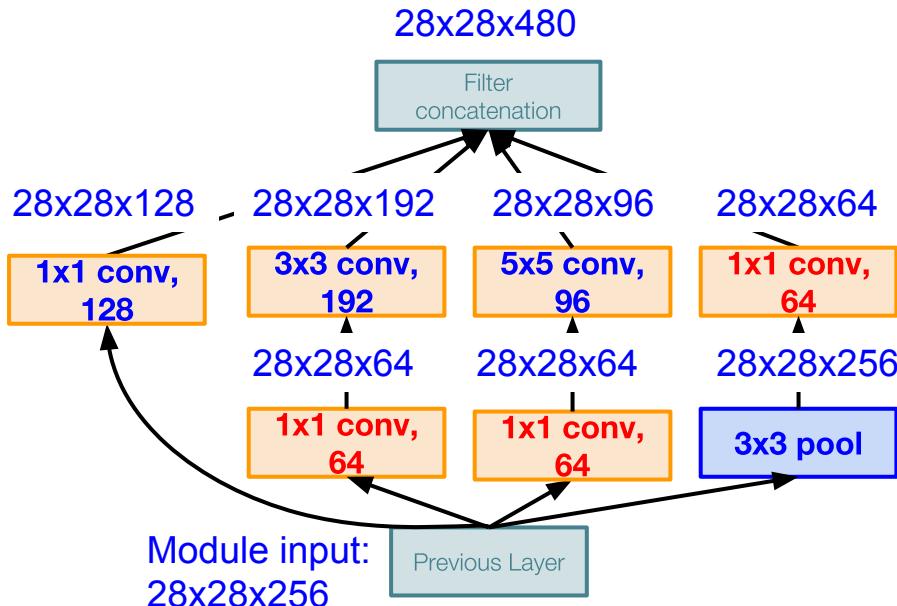
Naive Inception module



Inception module with dimension reduction

Case Study: GoogLeNet

[Szegedy et al., 2014]



Inception module with dimension reduction

Using same parallel layers as naive example, and adding “ 1×1 conv, 64 filter” bottlenecks:

Conv Ops:

- [1×1 conv, 64] $28 \times 28 \times 64 \times 1 \times 1 \times 256$
- [1×1 conv, 64] $28 \times 28 \times 64 \times 1 \times 1 \times 256$
- [1×1 conv, 128] $28 \times 28 \times 128 \times 1 \times 1 \times 256$
- [3×3 conv, 192] $28 \times 28 \times 192 \times 3 \times 3 \times 64$
- [5×5 conv, 96] $28 \times 28 \times 96 \times 5 \times 5 \times 64$
- [1×1 conv, 64] $28 \times 28 \times 64 \times 1 \times 1 \times 256$

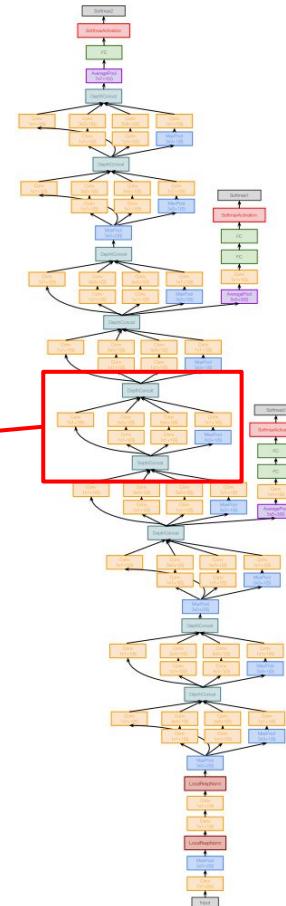
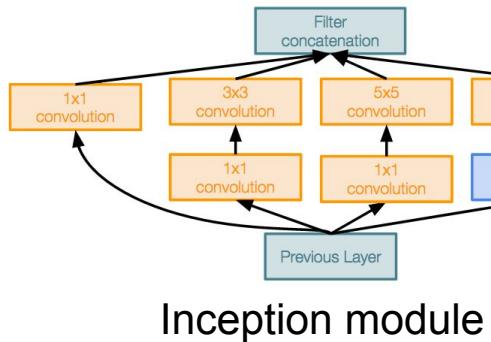
Total: 358M ops

Compared to 854M ops for naive version
Bottleneck can also reduce depth after pooling layer

Case Study: GoogLeNet

[Szegedy et al., 2014]

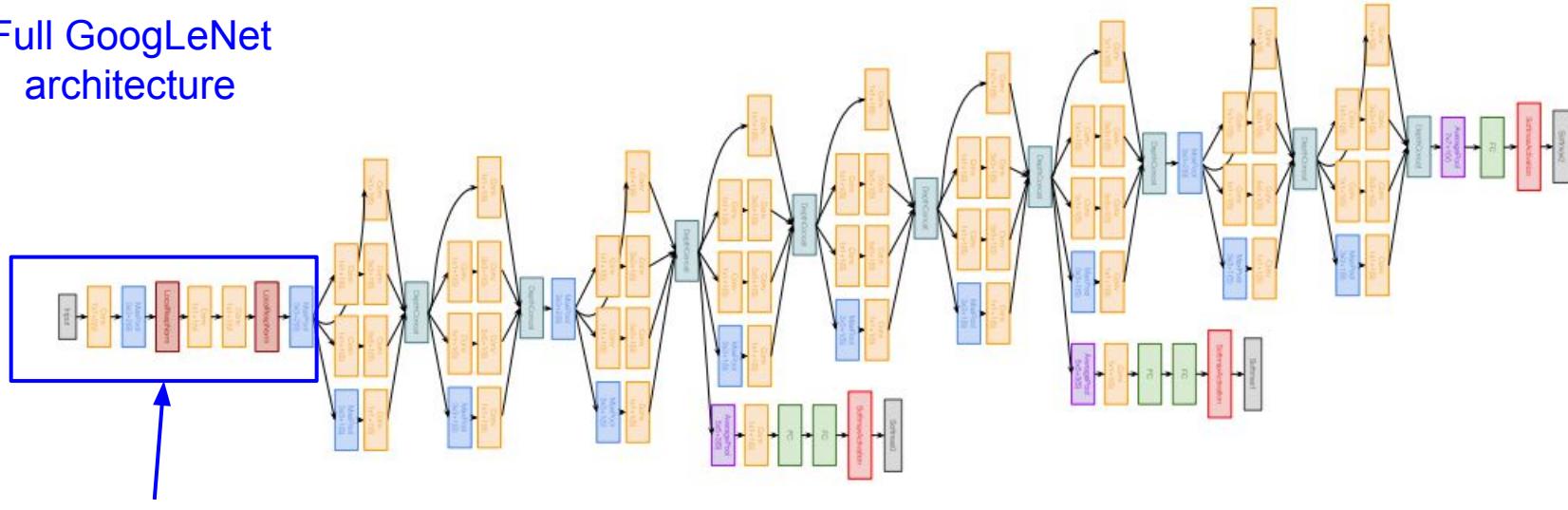
Stack Inception modules
with dimension reduction
on top of each other



Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet
architecture

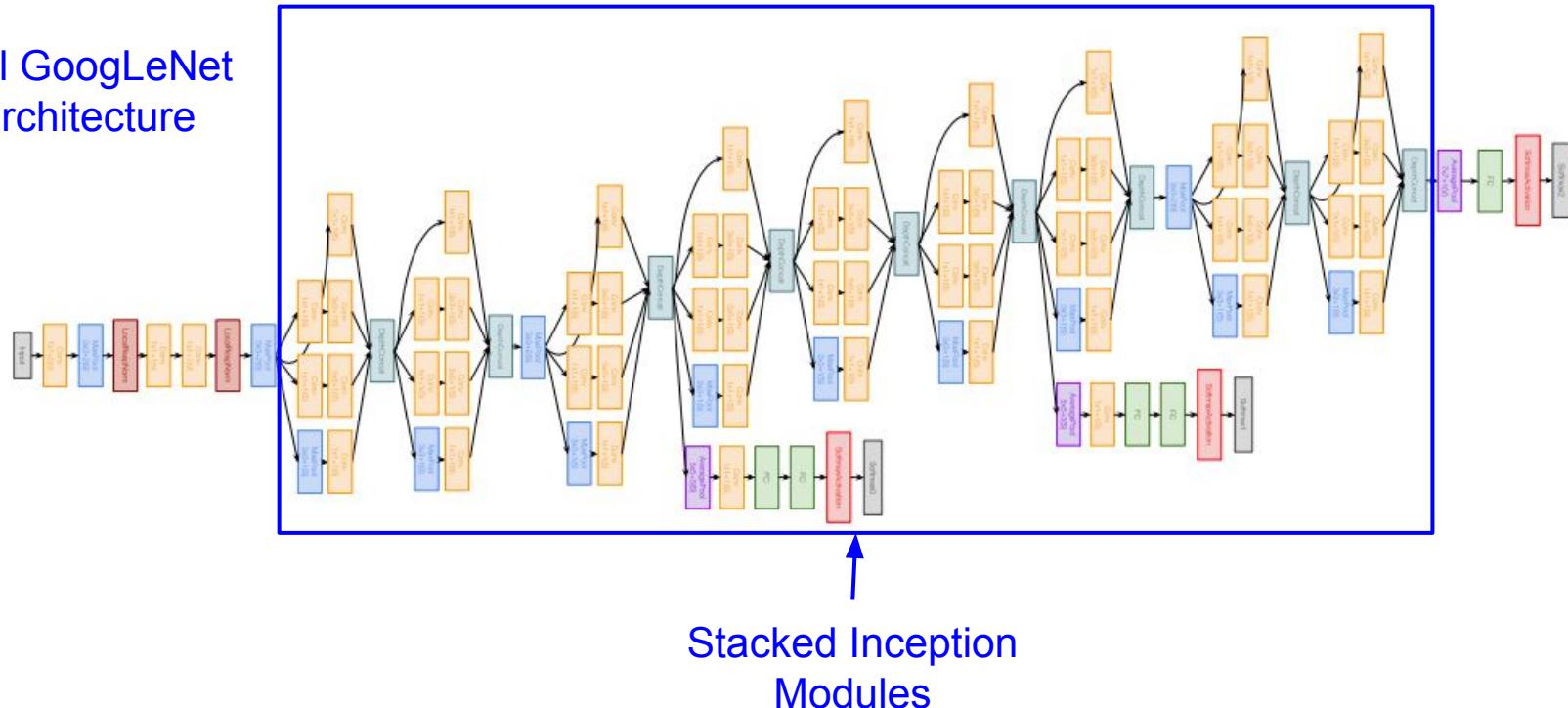


Stem Network:
Conv-Pool-
2x Conv-Pool

Case Study: GoogLeNet

[Szegedy et al., 2014]

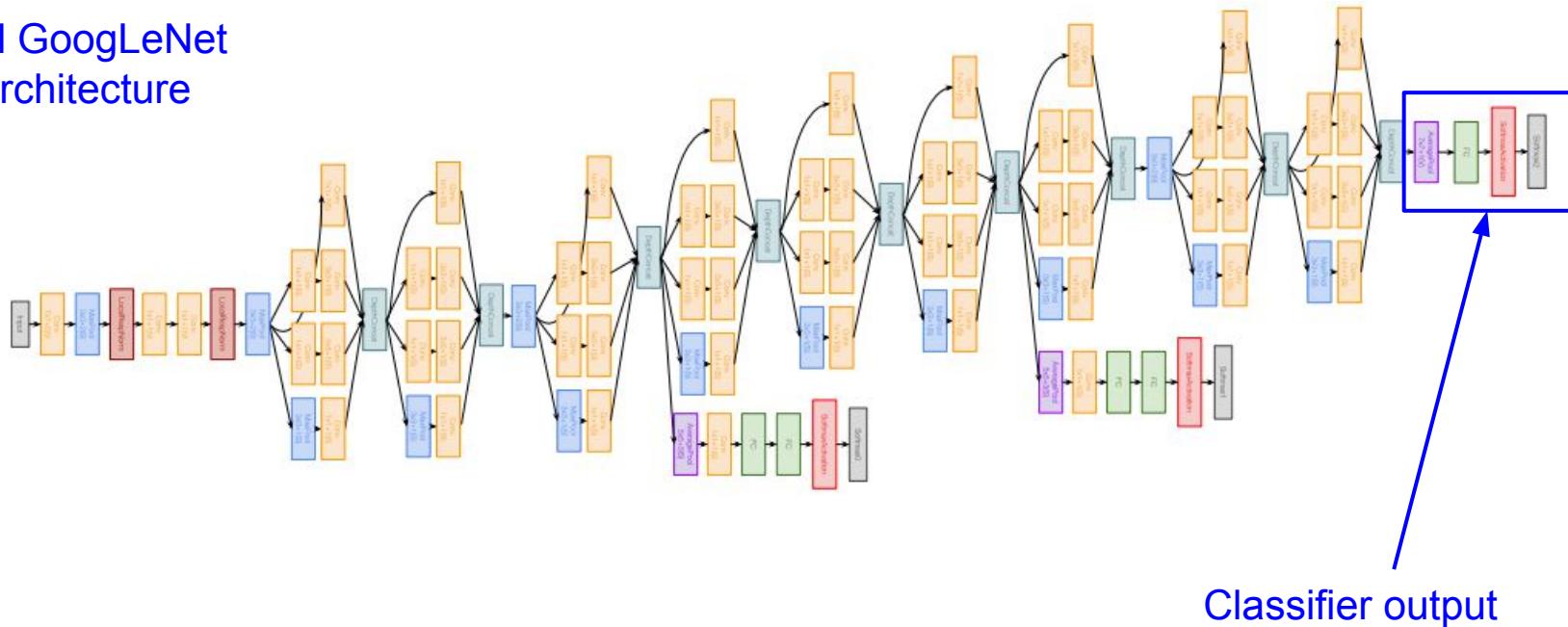
Full GoogLeNet
architecture



Case Study: GoogLeNet

[Szegedy et al., 2014]

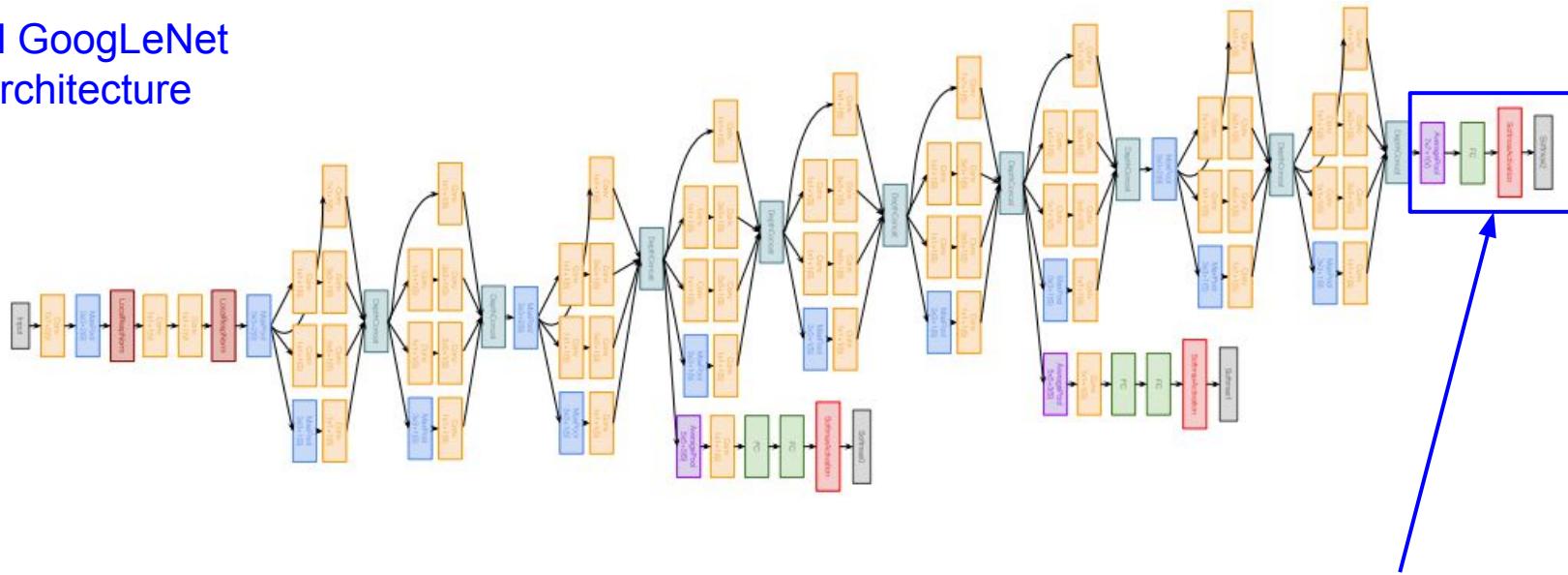
Full GoogLeNet
architecture



Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet
architecture

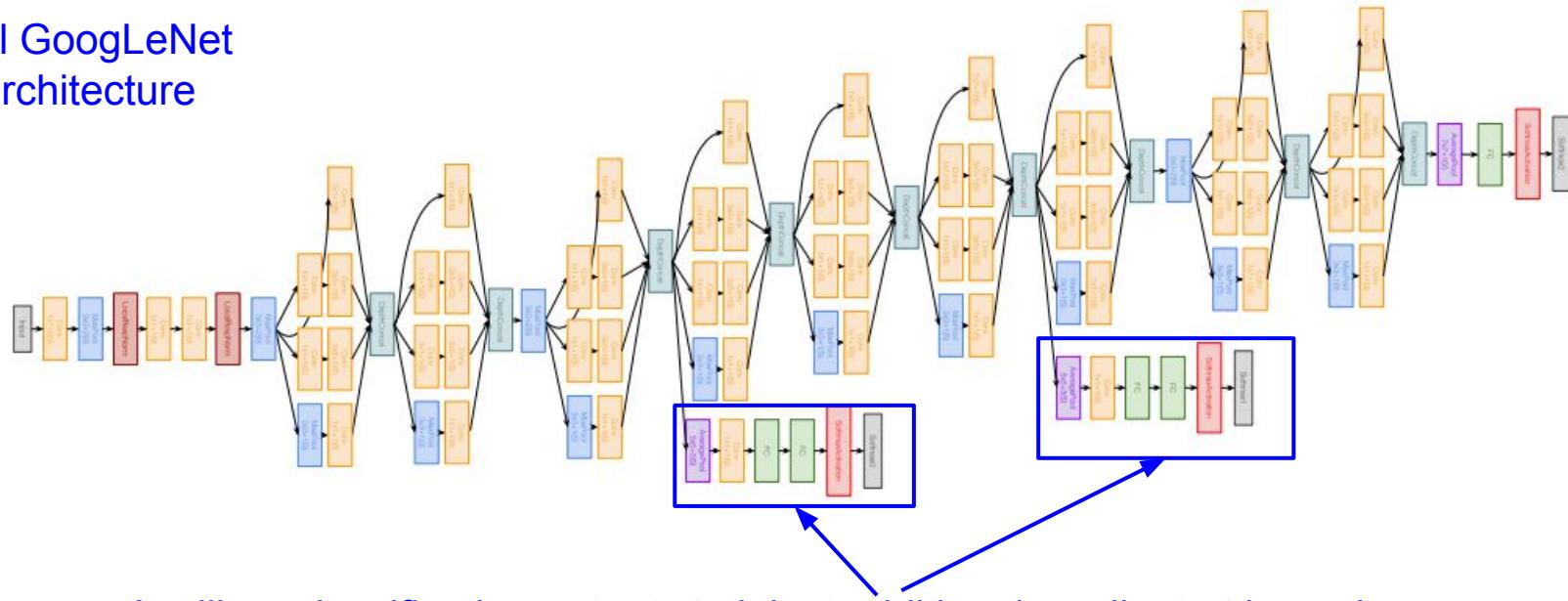


Classifier output
(removed expensive FC layers!)

Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet
architecture

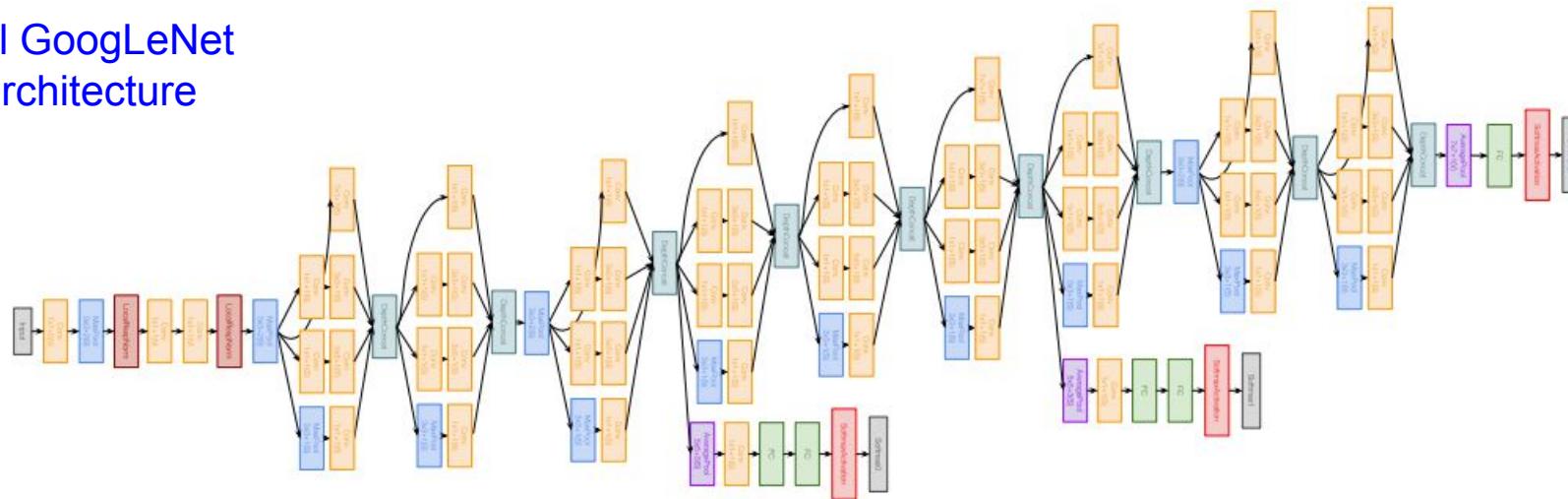


Auxiliary classification outputs to inject additional gradient at lower layers
(AvgPool-1x1Conv-FC-FC-Softmax)

Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet
architecture



22 total layers with weights

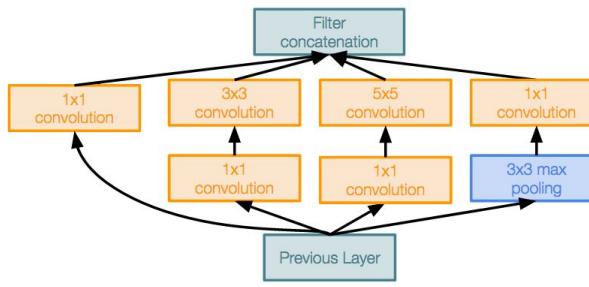
(parallel layers count as 1 layer => 2 layers per Inception module. Don't count auxiliary output layers)

Case Study: GoogLeNet

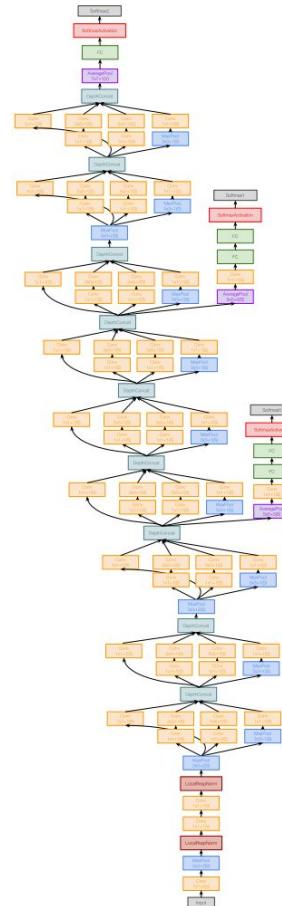
[Szegedy et al., 2014]

Deeper networks, with computational efficiency

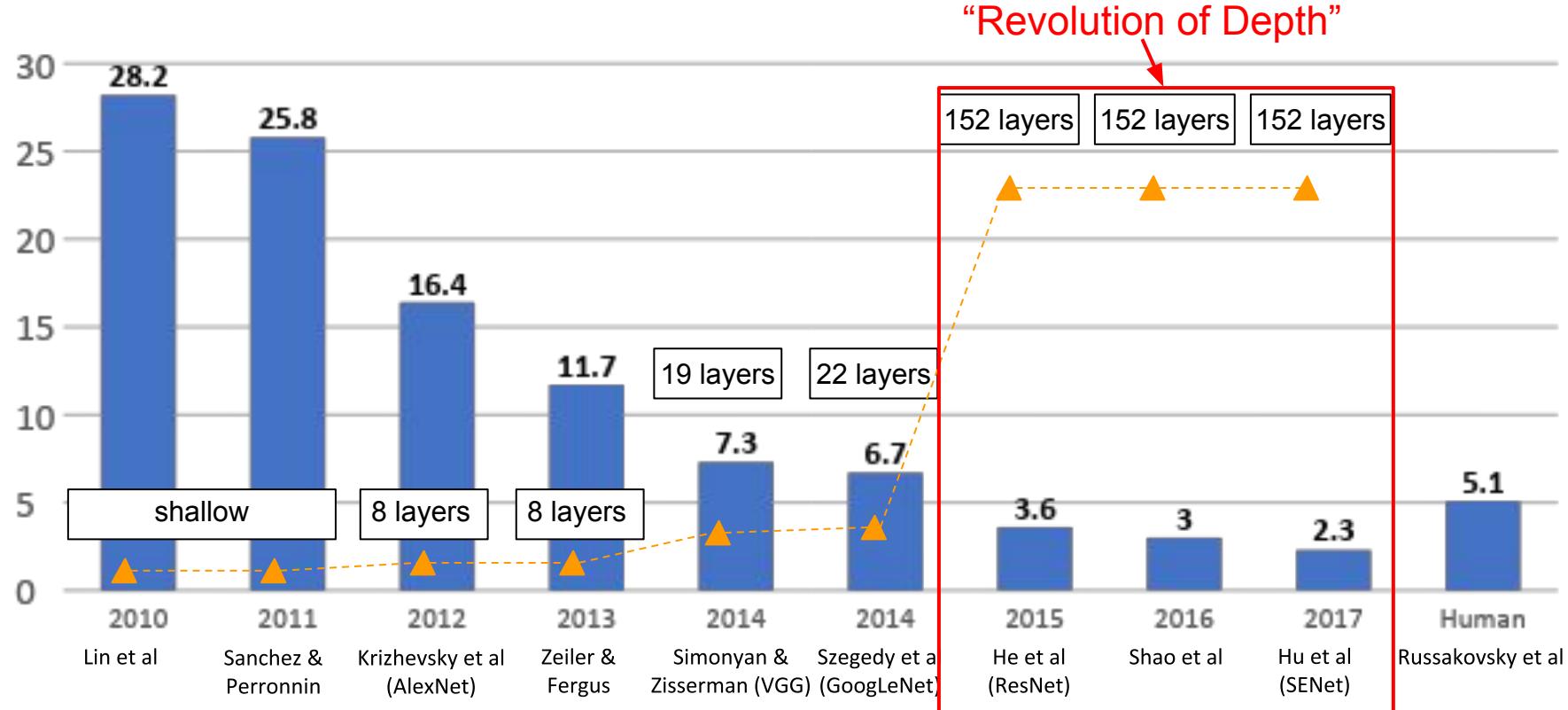
- 22 layers
- Efficient “Inception” module
- No FC layers
- 12x less params than AlexNet
- ILSVRC’14 classification winner (6.7% top 5 error)



Inception module



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

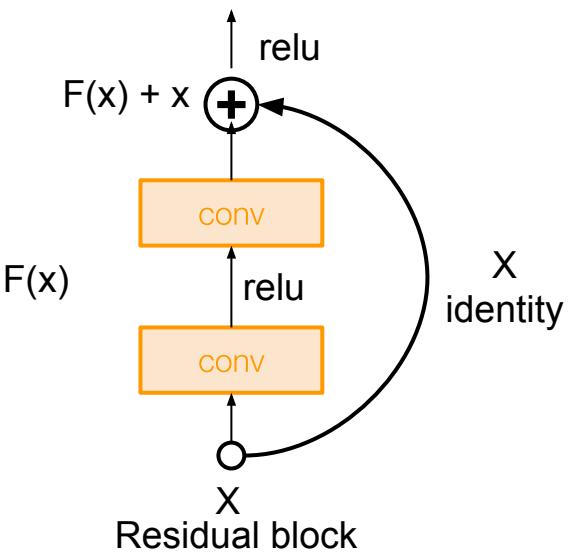


Case Study: ResNet

[He et al., 2015]

Very deep networks using residual connections

- 152-layer model for ImageNet
- ILSVRC'15 classification winner (3.57% top 5 error)
- Swept all classification and detection competitions in ILSVRC'15 and COCO'15!



Case Study: ResNet

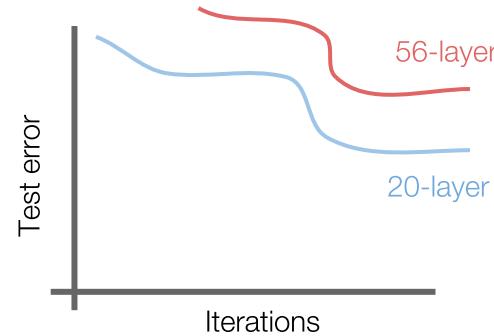
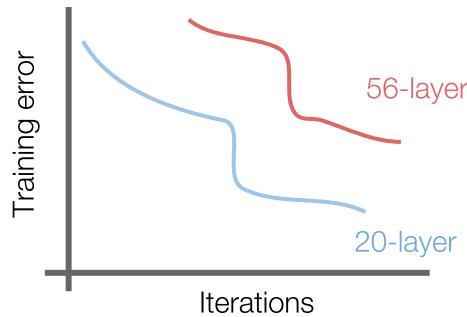
[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?

Case Study: ResNet

[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?

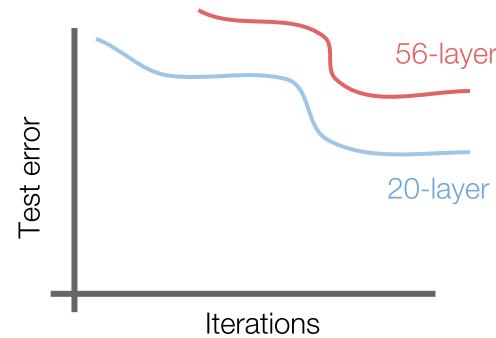
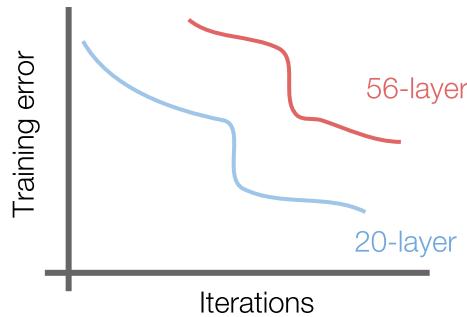


Q: What's strange about these training and test curves?
[Hint: look at the order of the curves]

Case Study: ResNet

[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?



56-layer model performs worse on both training and test error
-> The deeper model performs worse, but it's not caused by overfitting!

Case Study: ResNet

[He et al., 2015]

Hypothesis: the problem is an *optimization* problem, deeper models are harder to optimize

Case Study: ResNet

[He et al., 2015]

Hypothesis: the problem is an *optimization* problem, deeper models are harder to optimize

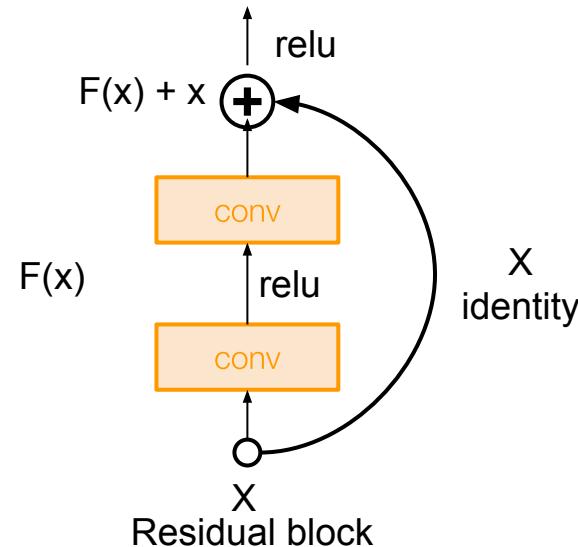
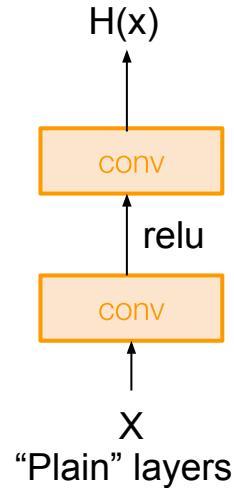
The deeper model should be able to perform at least as well as the shallower model.

A solution by construction is copying the learned layers from the shallower model and setting additional layers to identity mapping.

Case Study: ResNet

[He et al., 2015]

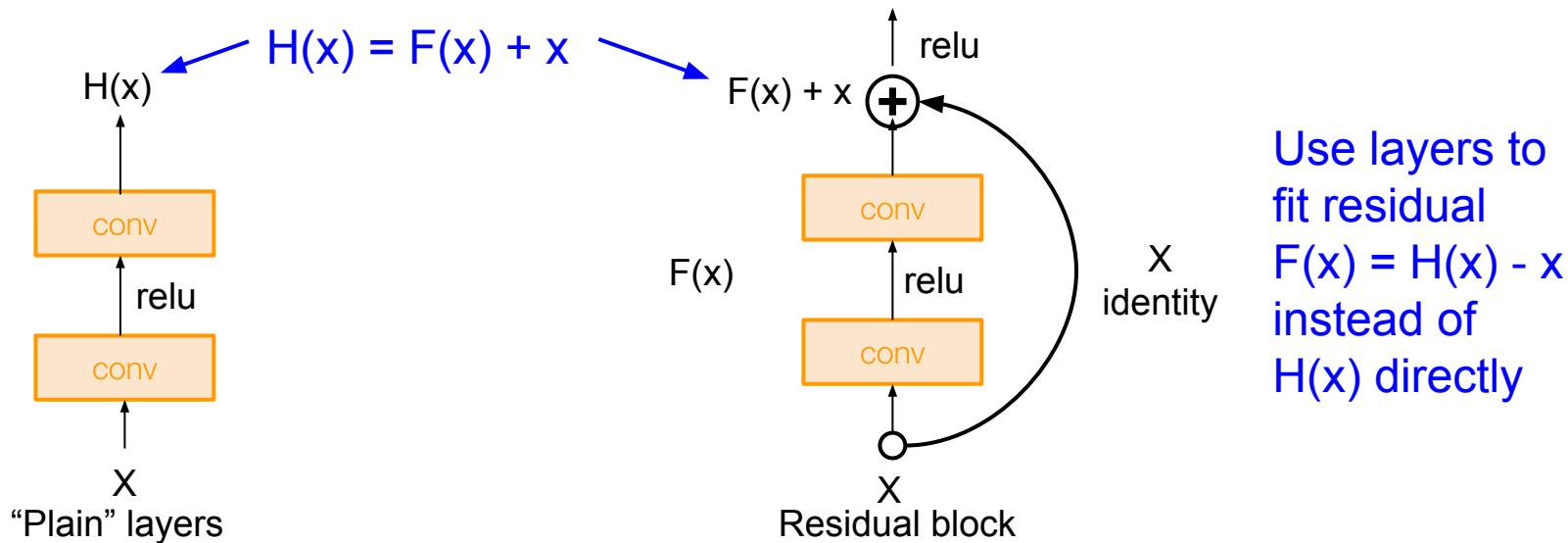
Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



Case Study: ResNet

[He et al., 2015]

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping

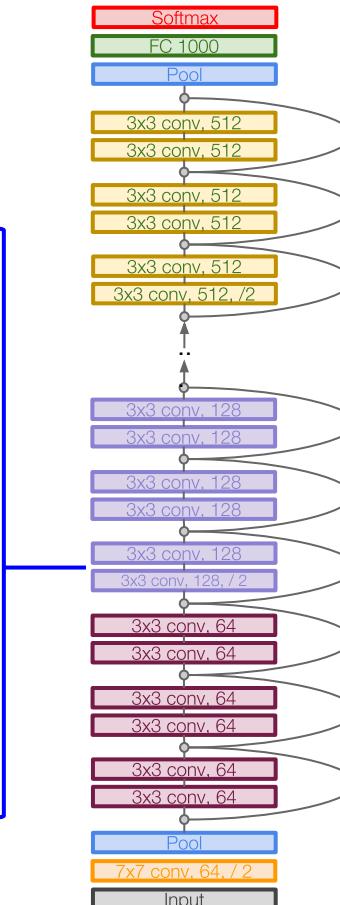
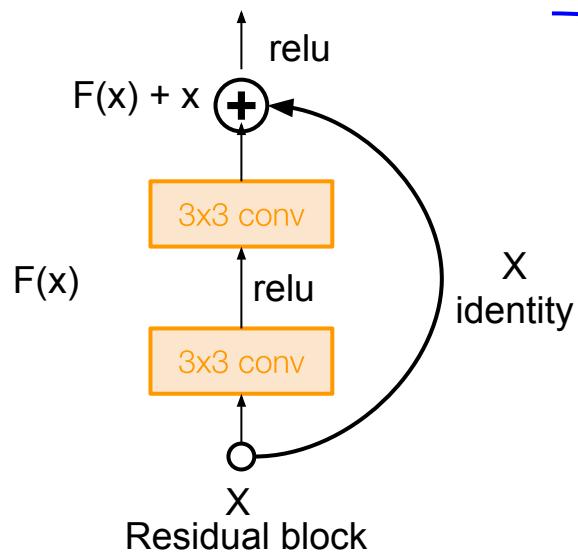


Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers

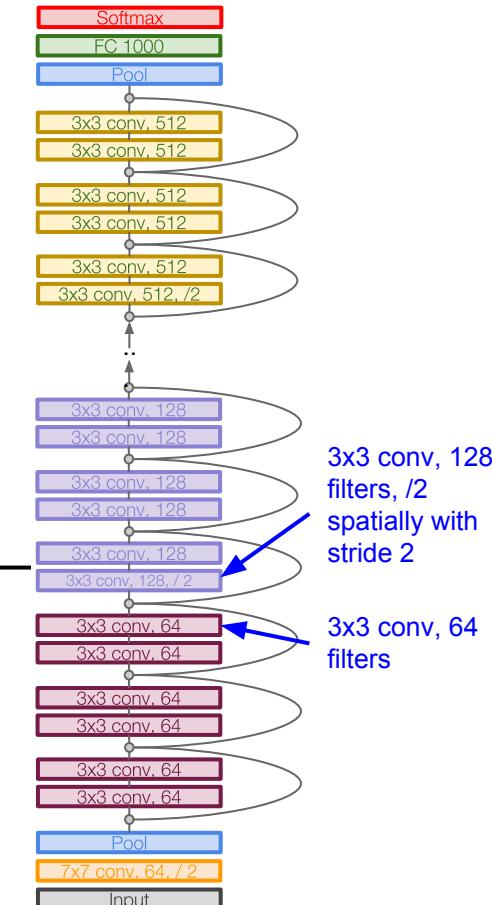
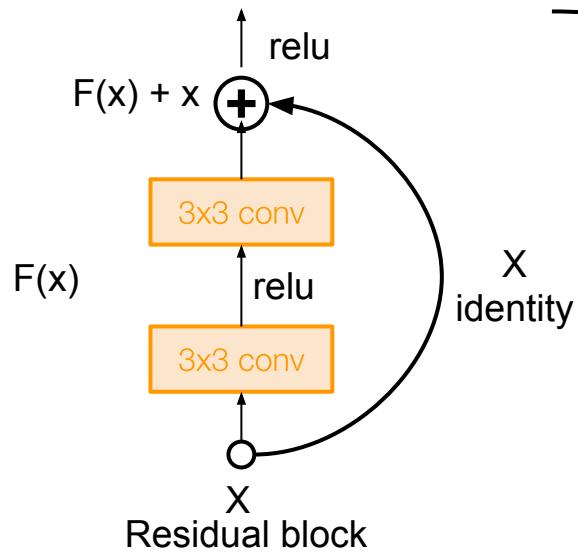


Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)

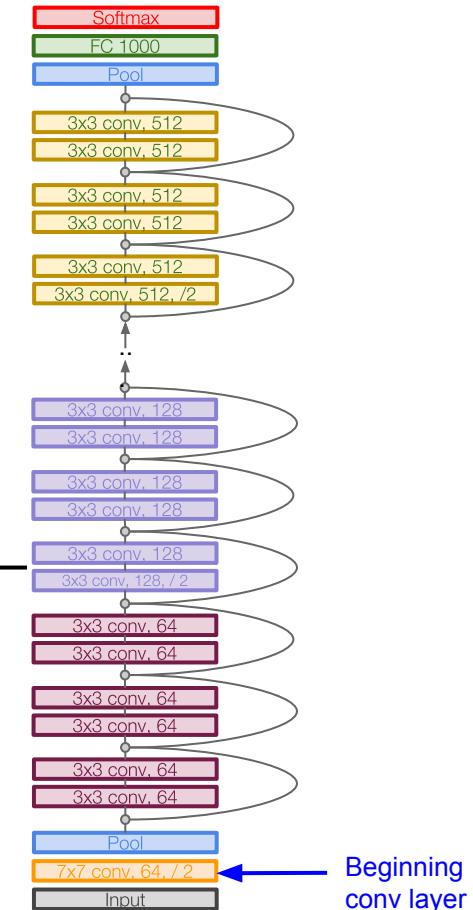
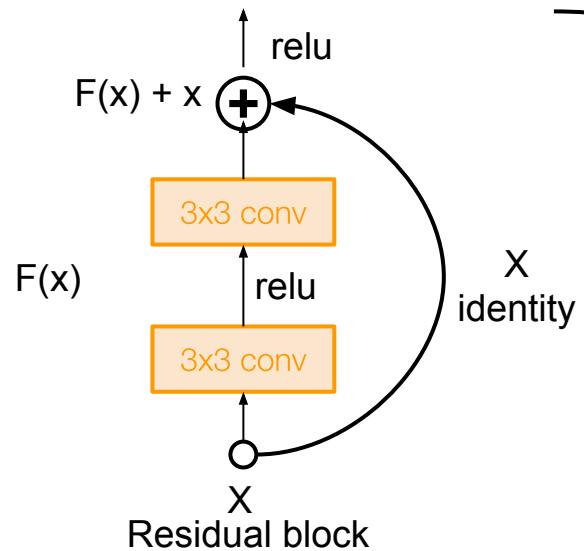


Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- Additional conv layer at the beginning

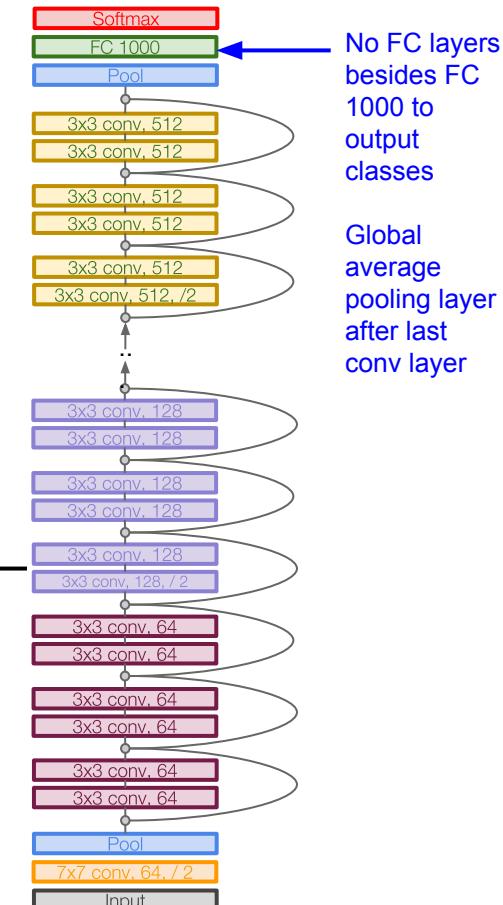
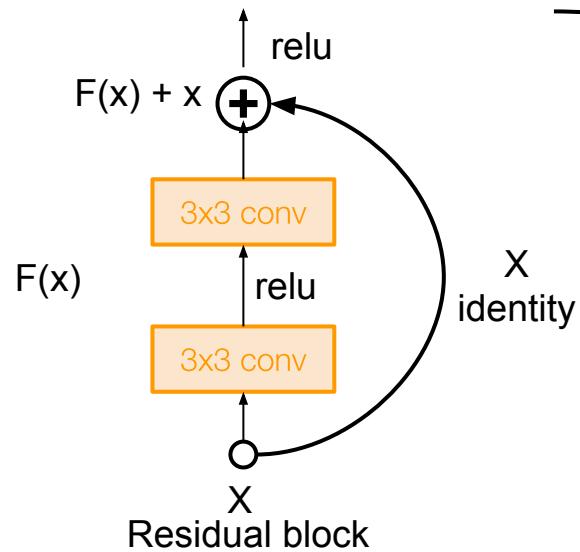


Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

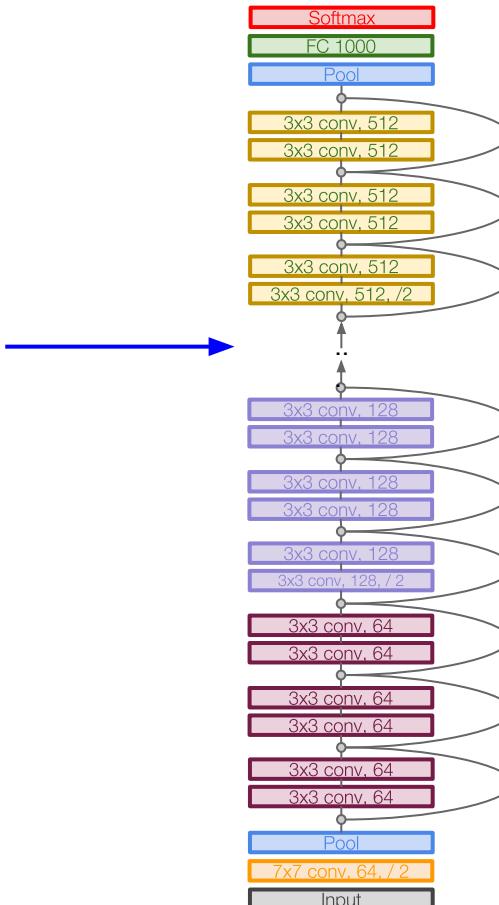
- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- Additional conv layer at the beginning
- No FC layers at the end (only FC 1000 to output classes)



Case Study: ResNet

[He et al., 2015]

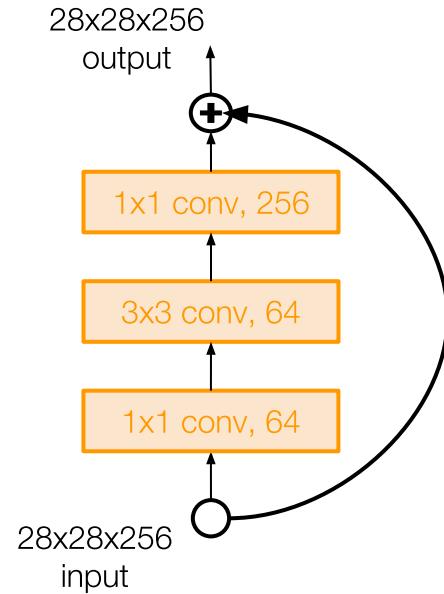
Total depths of 34, 50, 101, or
152 layers for ImageNet



Case Study: ResNet

[He et al., 2015]

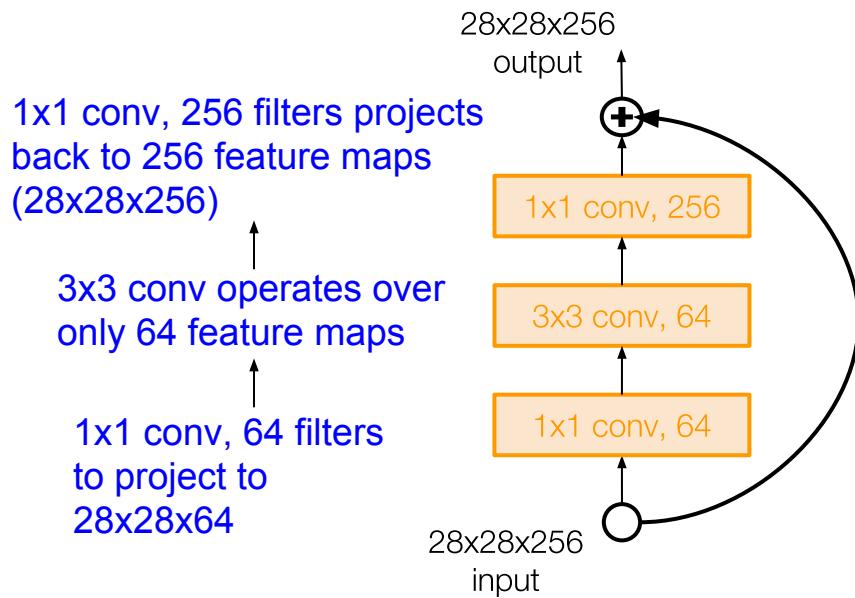
For deeper networks
(ResNet-50+), use “bottleneck”
layer to improve efficiency
(similar to GoogLeNet)



Case Study: ResNet

[He et al., 2015]

For deeper networks
(ResNet-50+), use “bottleneck”
layer to improve efficiency
(similar to GoogLeNet)



Case Study: ResNet

[He et al., 2015]

Training ResNet in practice:

- Batch Normalization after every CONV layer
- Xavier 2/ initialization from He et al.
- SGD + Momentum (0.9)
- Learning rate: 0.1, divided by 10 when validation error plateaus
- Mini-batch size 256
- Weight decay of 1e-5
- No dropout used

Case Study: ResNet

[He et al., 2015]

Experimental Results

- Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
- Deeper networks now achieve lower training error as expected
- Swept 1st place in all ILSVRC and COCO 2015 competitions

MSRA @ ILSVRC & COCO 2015 Competitions

- **1st places** in all five main tracks
 - ImageNet Classification: “Ultra-deep” (quote Yann) **152-layer** nets
 - ImageNet Detection: **16%** better than 2nd
 - ImageNet Localization: **27%** better than 2nd
 - COCO Detection: **11%** better than 2nd
 - COCO Segmentation: **12%** better than 2nd

Case Study: ResNet

[He et al., 2015]

Experimental Results

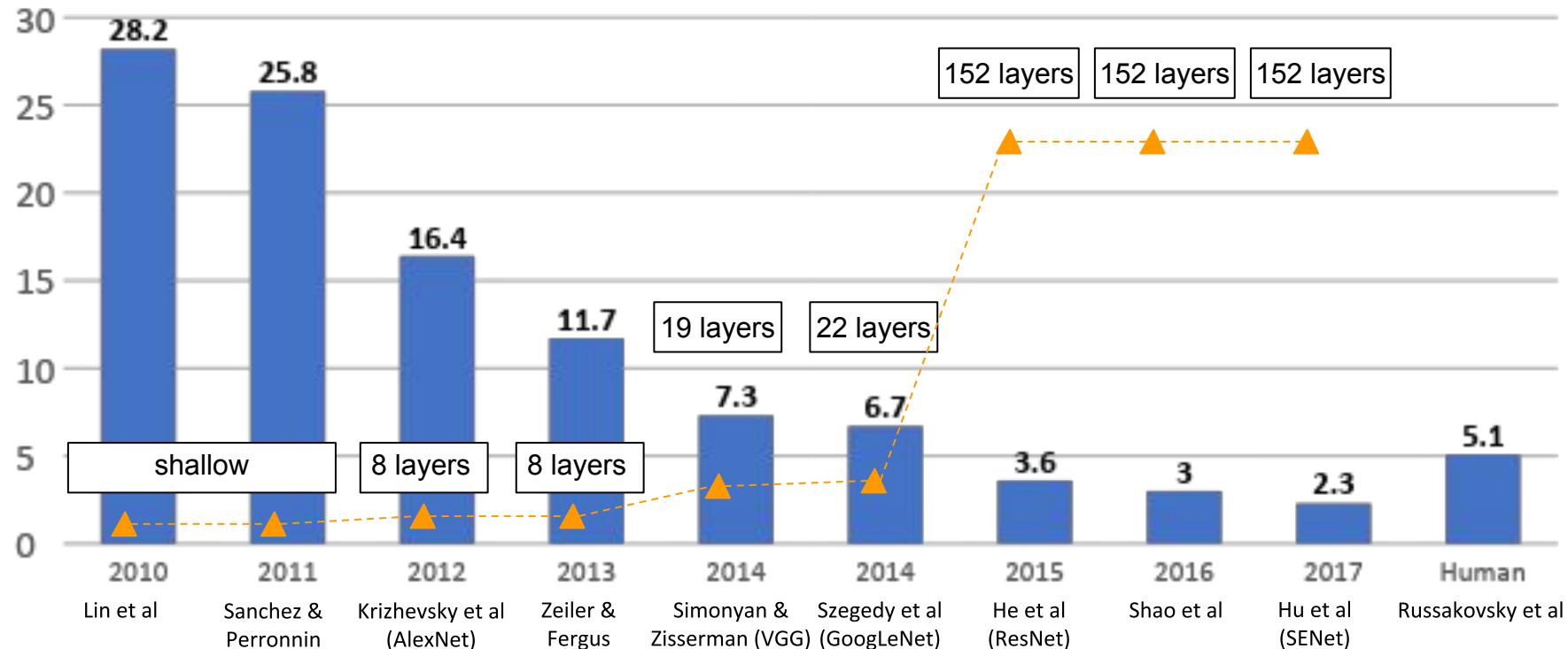
- Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
- Deeper networks now achieve lower training error as expected
- Swept 1st place in all ILSVRC and COCO 2015 competitions

MSRA @ ILSVRC & COCO 2015 Competitions

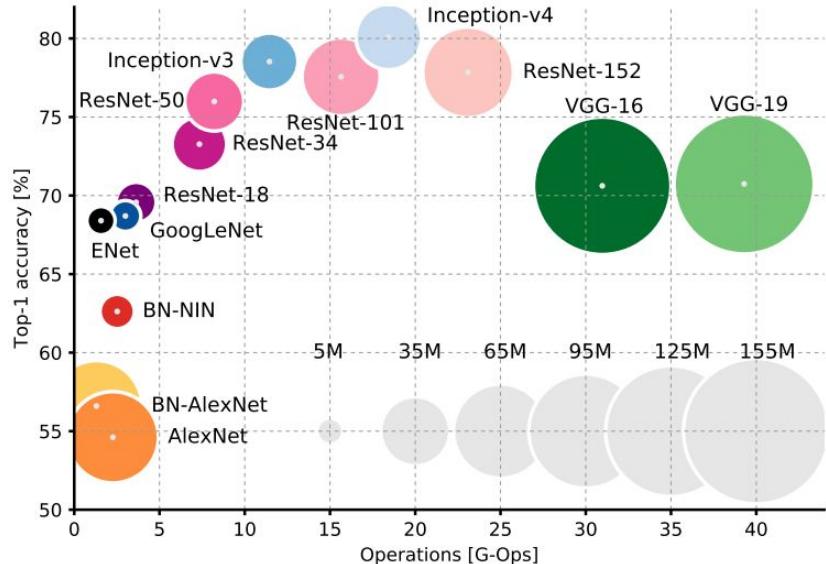
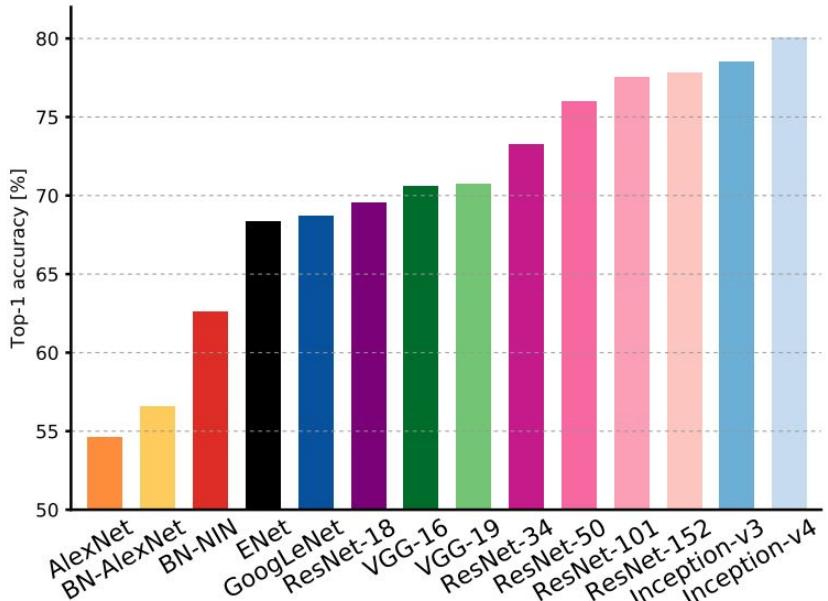
- **1st places** in all five main tracks
 - ImageNet Classification: “Ultra-deep” (quote Yann) 152-layer nets
 - ImageNet Detection: 16% better than 2nd
 - ImageNet Localization: 27% better than 2nd
 - COCO Detection: 11% better than 2nd
 - COCO Segmentation: 12% better than 2nd

ILSVRC 2015 classification winner (3.6% top 5 error) -- better than “human performance”! (Russakovsky 2014)

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Comparing complexity...

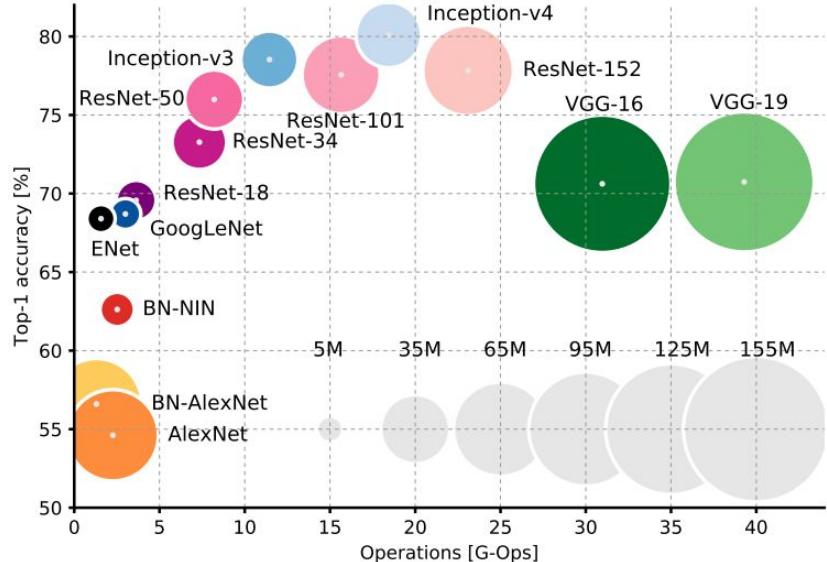
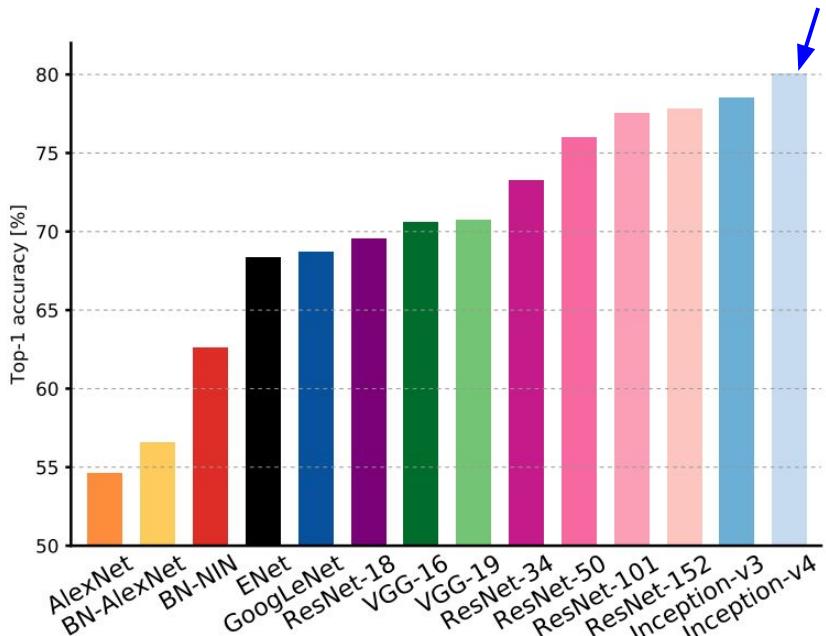


An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Figures copyright Alfredo Canziani, Adam Paszke, Eugenio Culurciello, 2017. Reproduced with permission.

Comparing complexity...

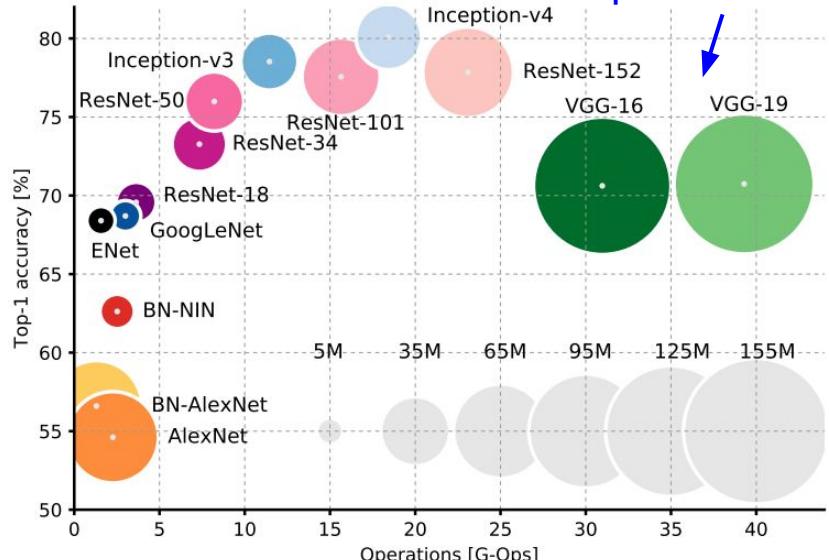
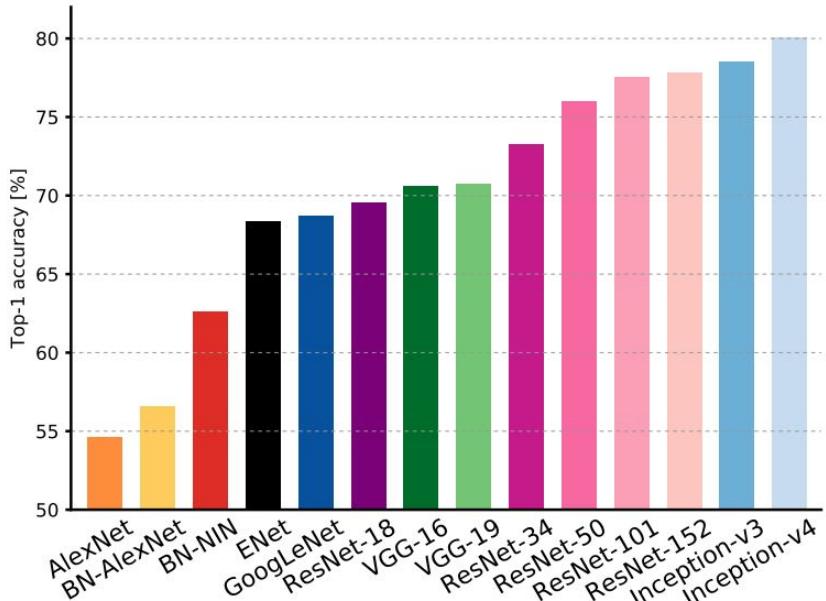
Inception-v4: Resnet + Inception!



An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Figures copyright Alfredo Canziani, Adam Paszke, Eugenio Culurciello, 2017. Reproduced with permission.

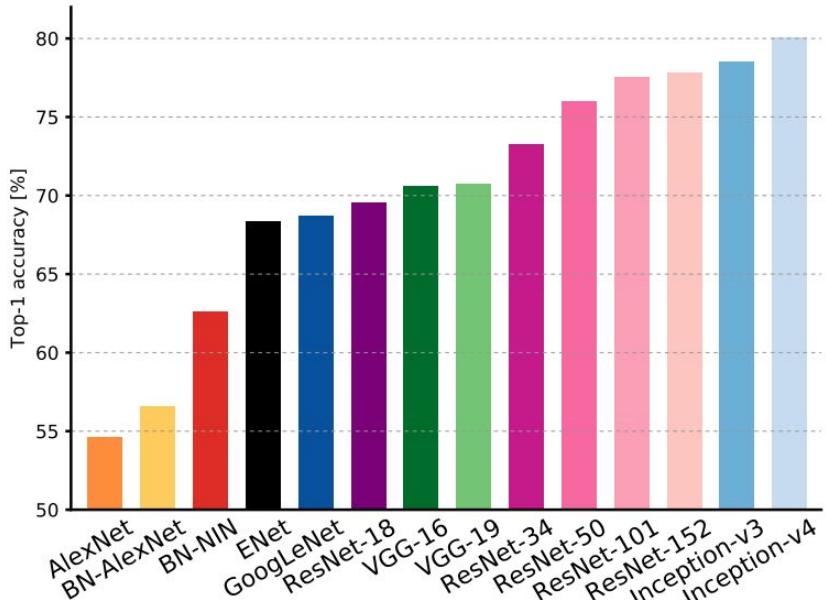
Comparing complexity...



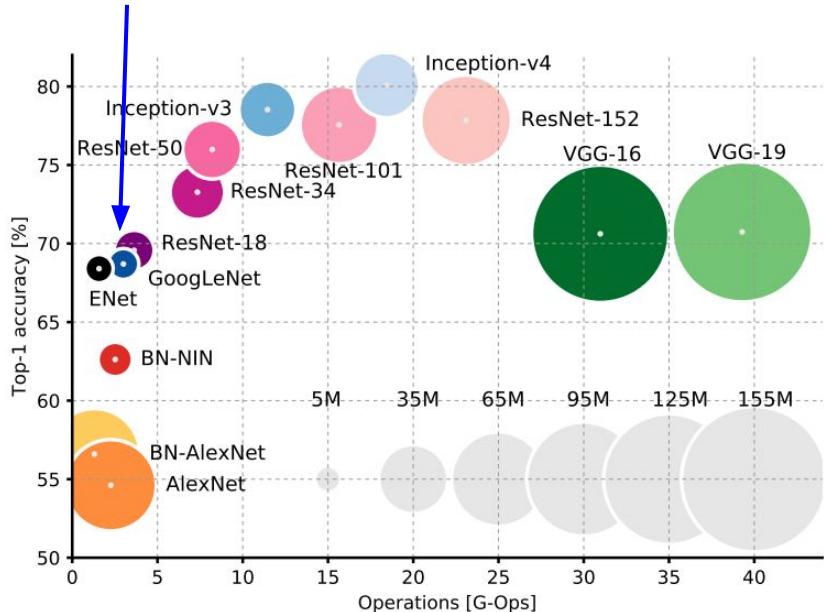
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Figures copyright Alfredo Canziani, Adam Paszke, Eugenio Culurciello, 2017. Reproduced with permission.

Comparing complexity...



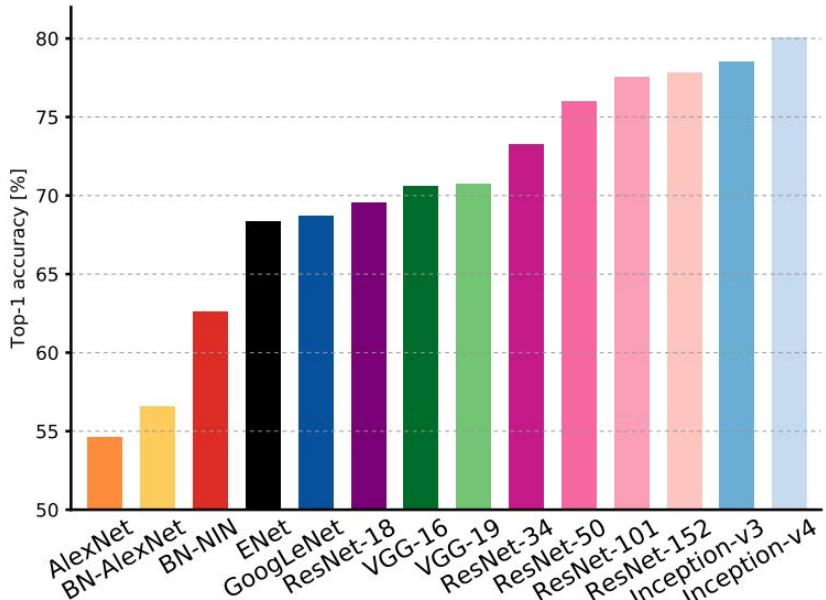
GoogLeNet:
most efficient



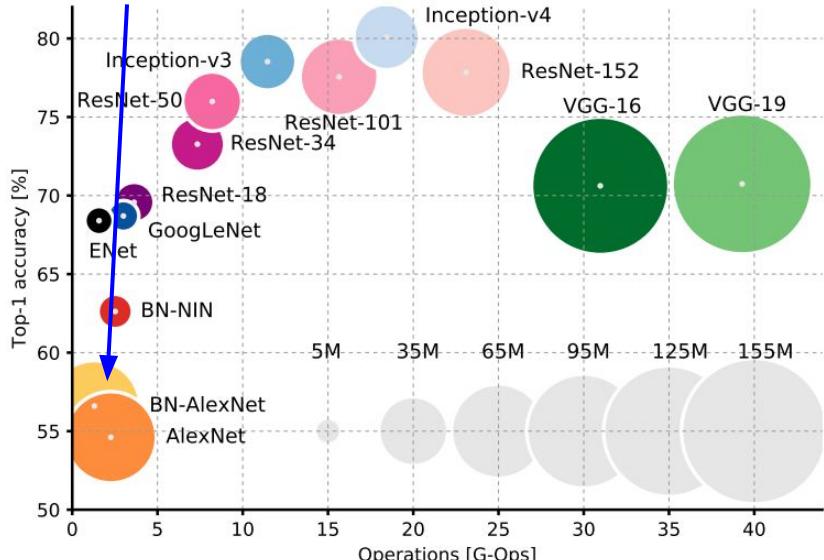
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Figures copyright Alfredo Canziani, Adam Paszke, Eugenio Culurciello, 2017. Reproduced with permission.

Comparing complexity...



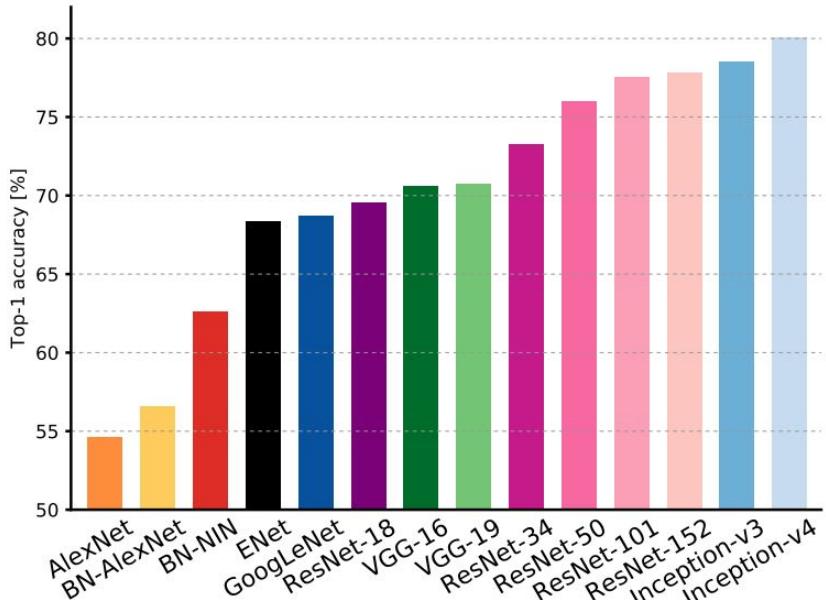
AlexNet:
Smaller compute, still memory
heavy, lower accuracy



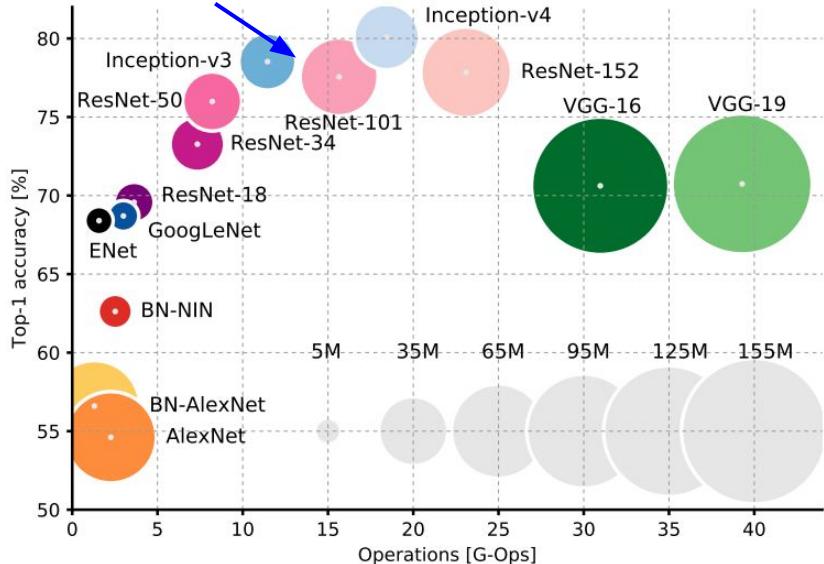
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Figures copyright Alfredo Canziani, Adam Paszke, Eugenio Culurciello, 2017. Reproduced with permission.

Comparing complexity...



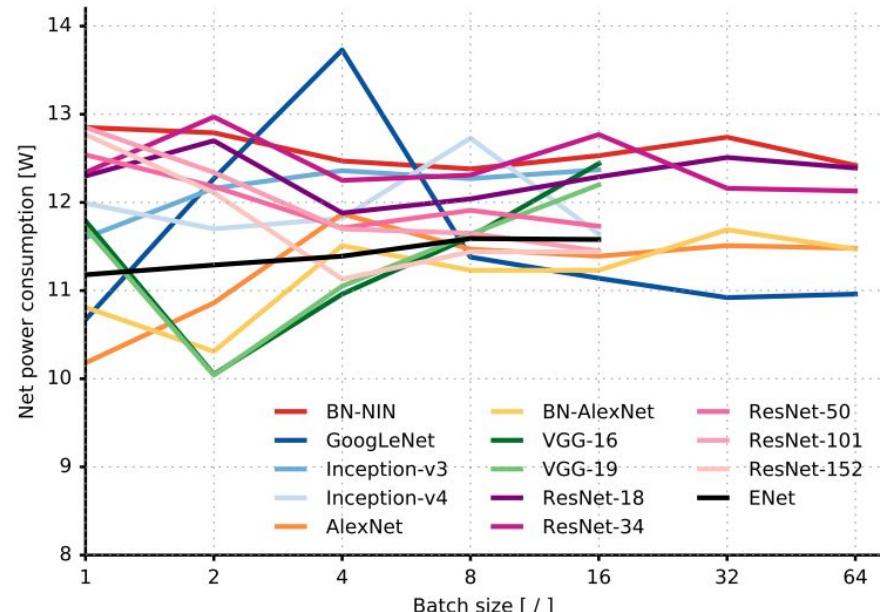
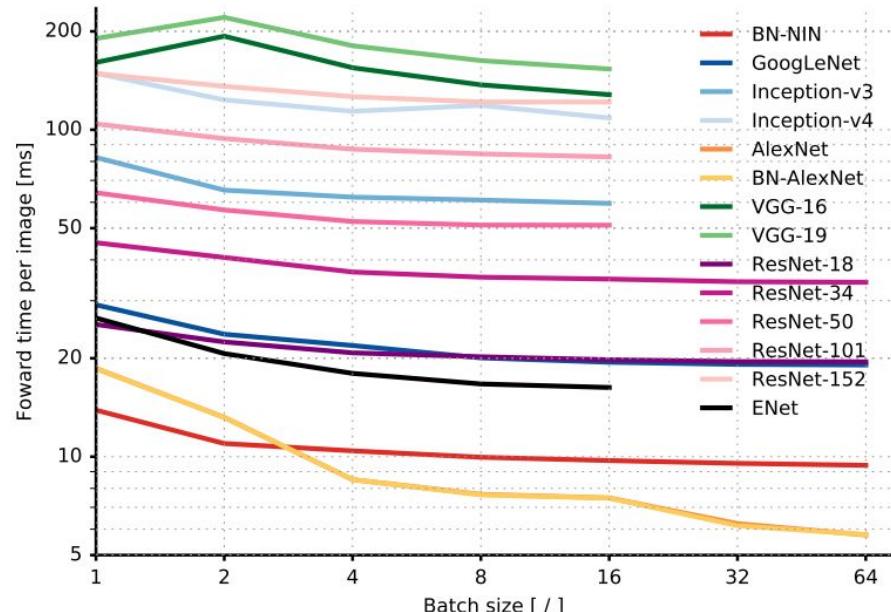
ResNet:
Moderate efficiency depending on
model, highest accuracy



An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Figures copyright Alfredo Canziani, Adam Paszke, Eugenio Culurciello, 2017. Reproduced with permission.

Forward pass time and power consumption



An Analysis of Deep Neural Network Models for Practical Applications, 2017.

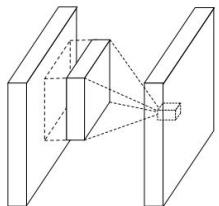
Figures copyright Alfredo Canziani, Adam Paszke, Eugenio Culurciello, 2017. Reproduced with permission.

Other architectures to know...

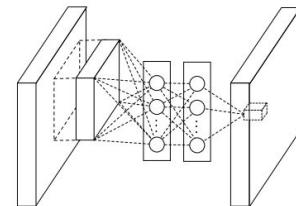
Network in Network (NiN)

[Lin et al. 2014]

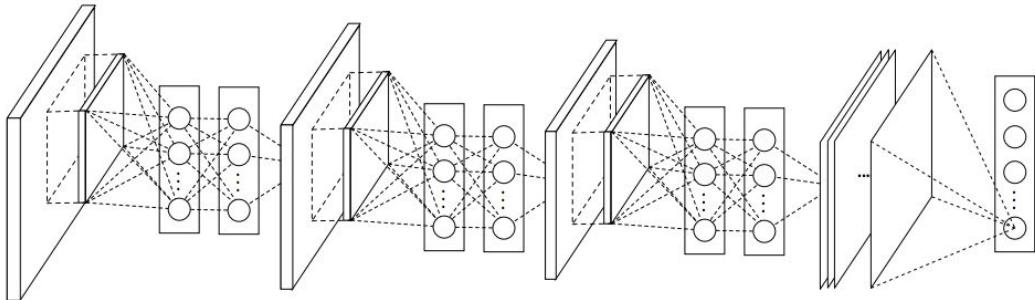
- Mlpconv layer with “micronetwork” within each conv layer to compute more abstract features for local patches
- Micronetwork uses multilayer perceptron (FC, i.e. 1×1 conv layers)
- Precursor to GoogLeNet and ResNet “bottleneck” layers
- Philosophical inspiration for GoogLeNet



(a) Linear convolution layer



(b) Mlpconv layer

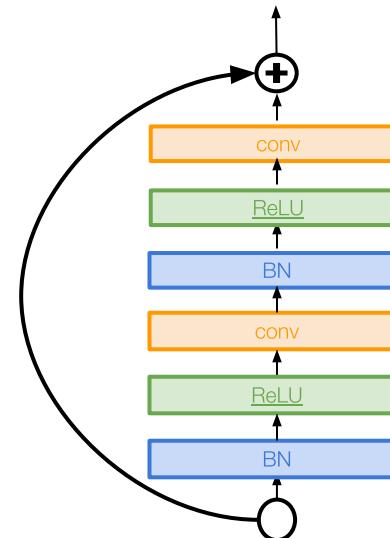


Figures copyright Lin et al., 2014. Reproduced with permission.

Identity Mappings in Deep Residual Networks

[He et al. 2016]

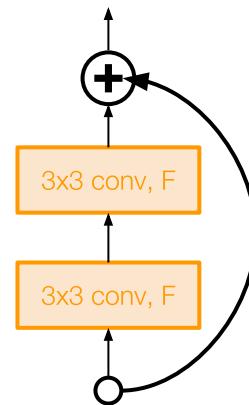
- Improved ResNet block design from creators of ResNet
- Creates a more direct path for propagating information throughout network (moves activation to residual mapping pathway)
- Gives better performance



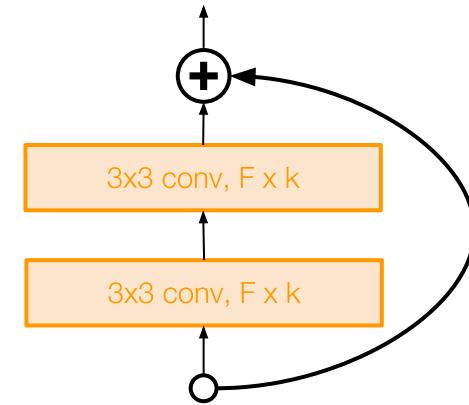
Wide Residual Networks

[Zagoruyko et al. 2016]

- Argues that residuals are the important factor, not depth
- Use wider residual blocks ($F \times k$ filters instead of F filters in each layer)
- 50-layer wide ResNet outperforms 152-layer original ResNet
- Increasing width instead of depth more computationally efficient (parallelizable)



Basic residual block

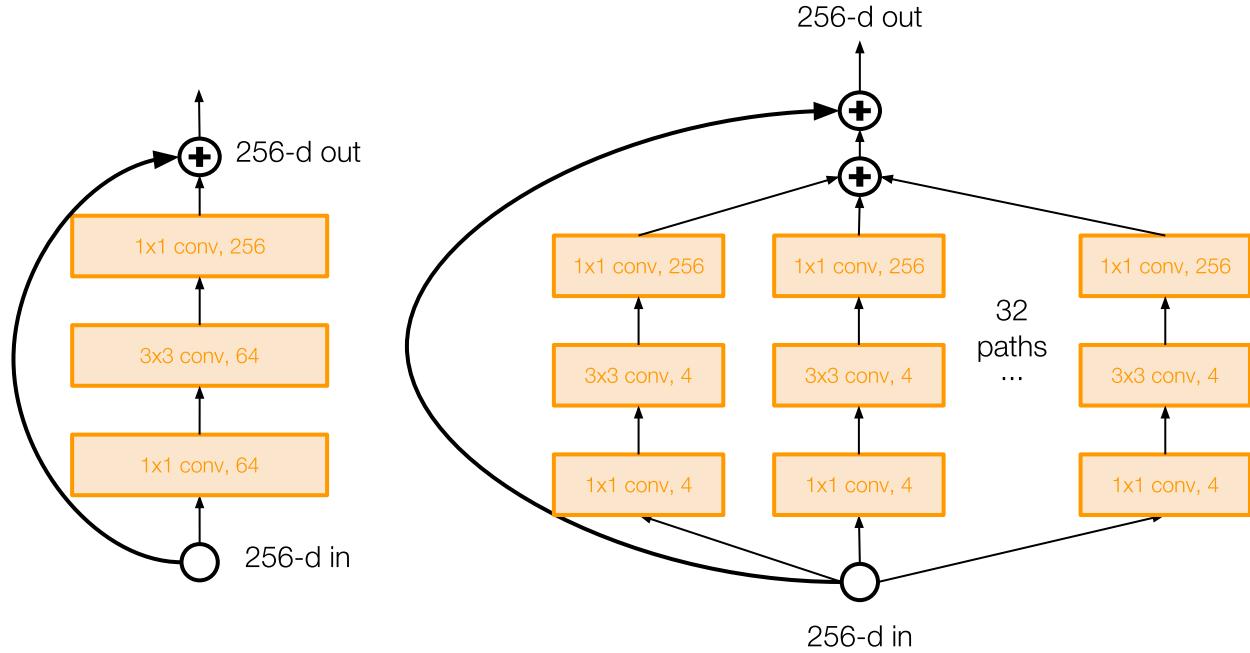


Wide residual block

Improving ResNets... Aggregated Residual Transformations for Deep Neural Networks (ResNeXt)

[Xie et al. 2016]

- Also from creators of ResNet
- Increases width of residual block through multiple parallel pathways (“cardinality”)
- Parallel pathways similar in spirit to Inception module

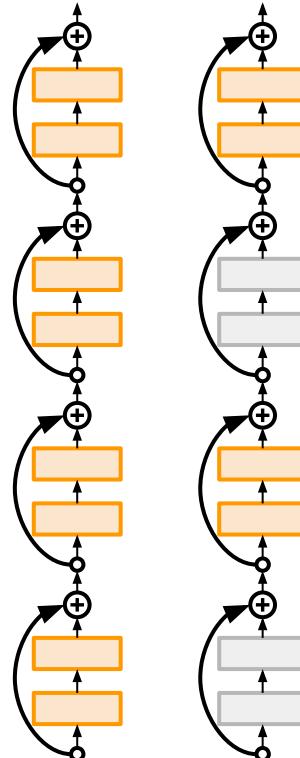


Improving ResNets...

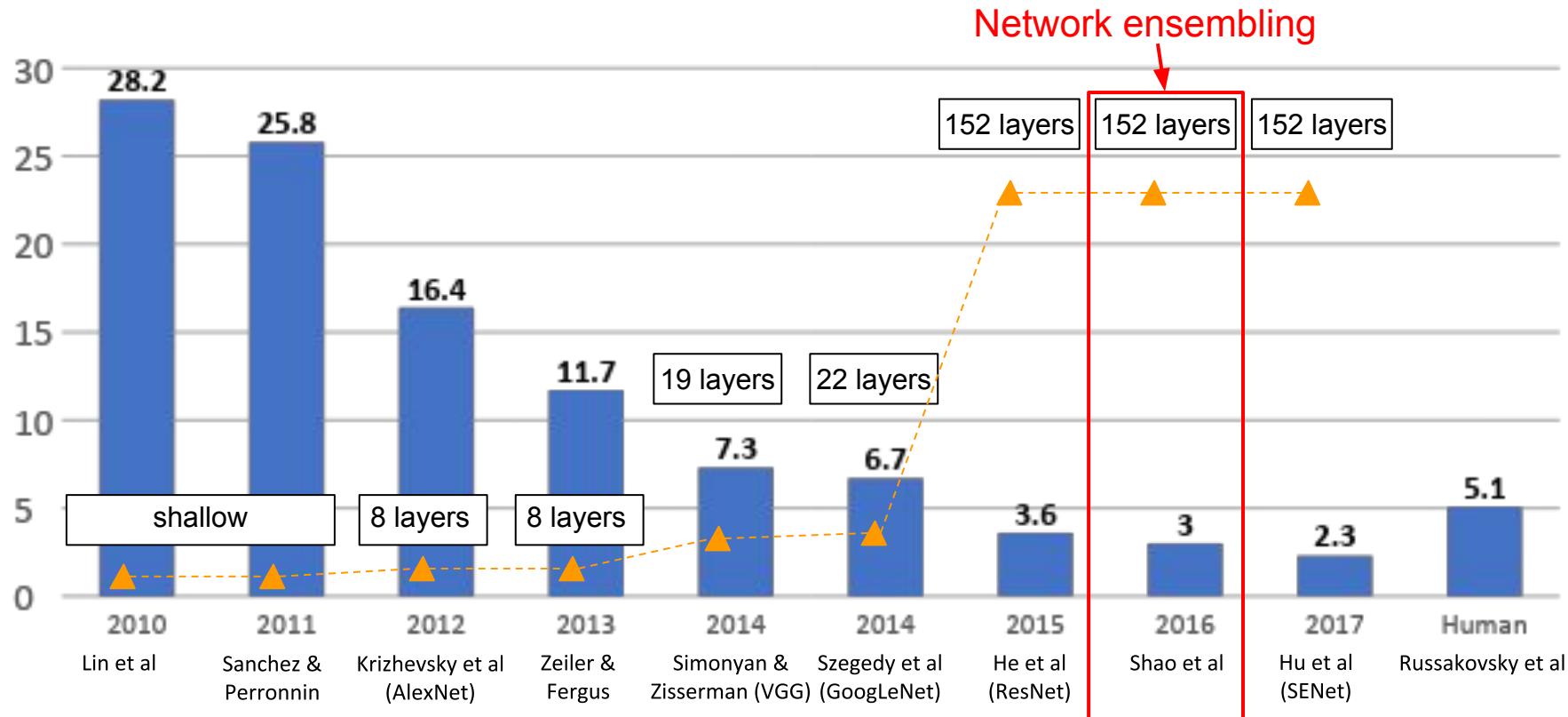
Deep Networks with Stochastic Depth

[Huang et al. 2016]

- Motivation: reduce vanishing gradients and training time through short networks during training
- Randomly drop a subset of layers during each training pass
- Bypass with identity function
- Use full deep network at test time



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Improving ResNets...

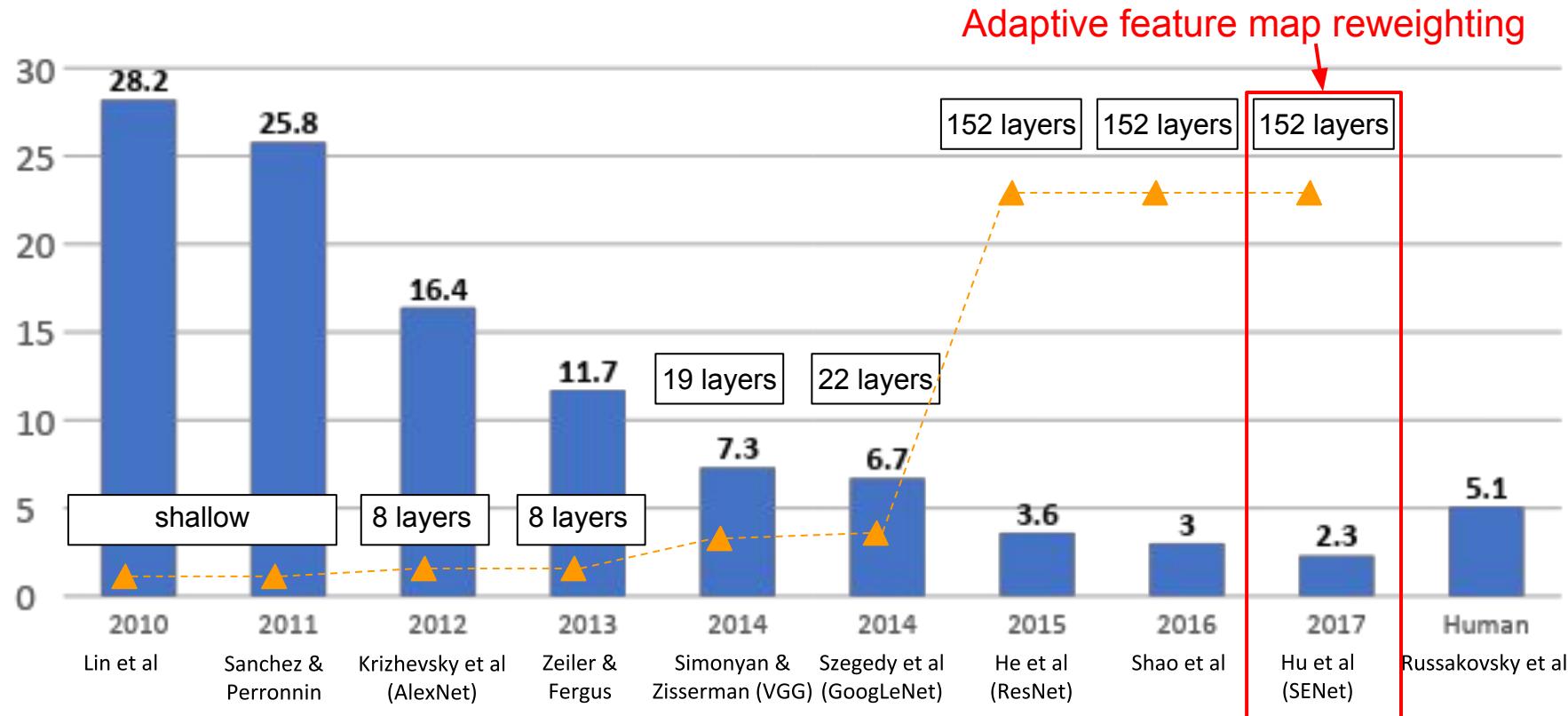
“Good Practices for Deep Feature Fusion”

[Shao et al. 2016]

- Multi-scale ensembling of Inception, Inception-Resnet, Resnet, Wide Resnet models
- ILSVRC'16 classification winner

	Inception-v3	Inception-v4	Inception-Resnet-v2	Resnet-200	Wrn-68-3	Fusion (Val.)	Fusion (Test)
Err. (%)	4.20	4.01	3.52	4.26	4.65	2.92 (-0.6)	2.99

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

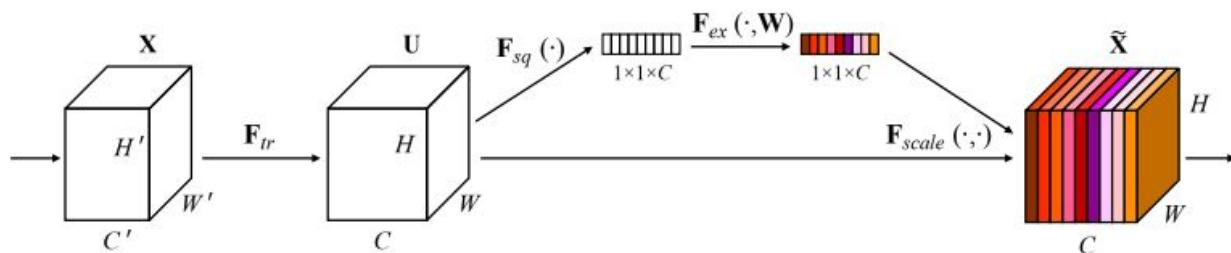
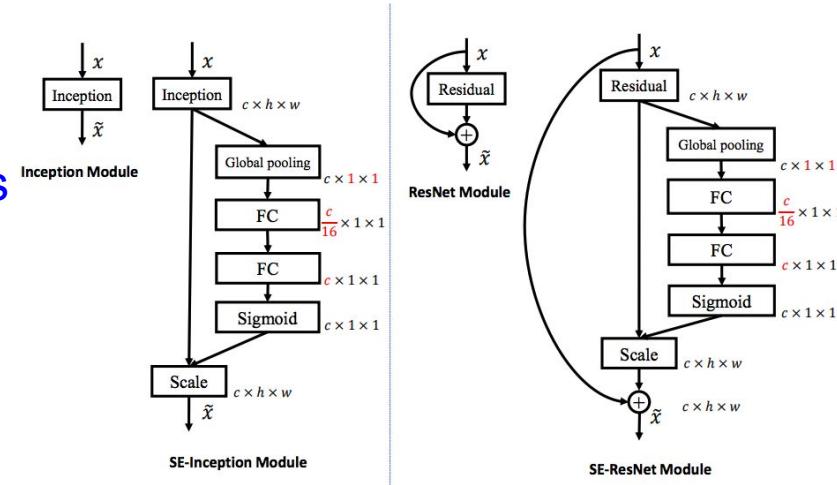


Improving ResNets...

Squeeze-and-Excitation Networks (SENet)

[Hu et al. 2017]

- Add a “feature recalibration” module that learns to adaptively reweight feature maps
- Global information (global avg. pooling layer) + 2 FC layers used to determine feature map weights
- ILSVRC’17 classification winner (using ResNeXt-152 as a base architecture)

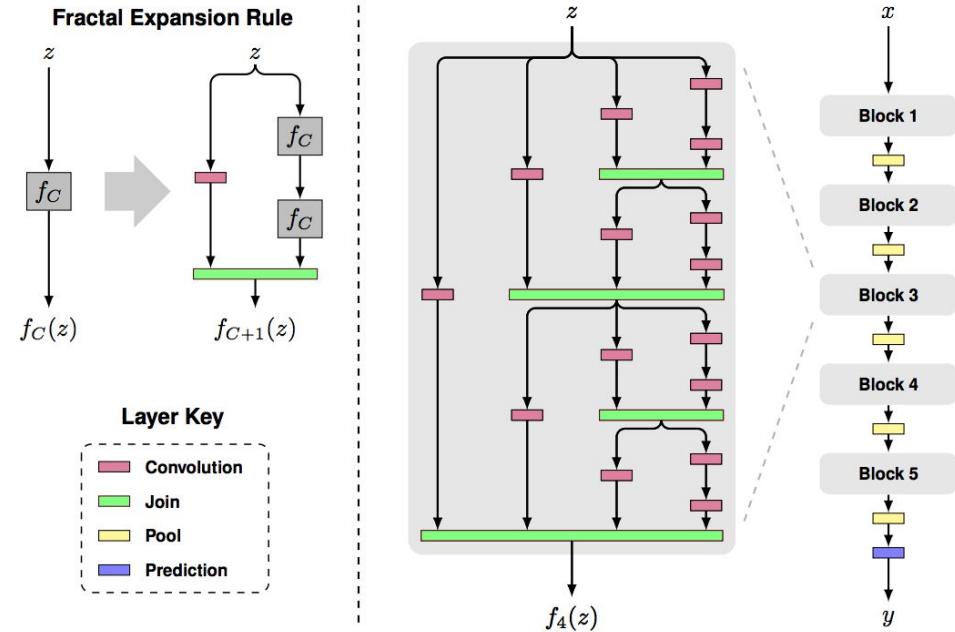


Beyond ResNets...

FractalNet: Ultra-Deep Neural Networks without Residuals

[Larsson et al. 2017]

- Argues that key is transitioning effectively from shallow to deep and residual representations are not necessary
- Fractal architecture with both shallow and deep paths to output
- Trained with dropping out sub-paths
- Full network at test time



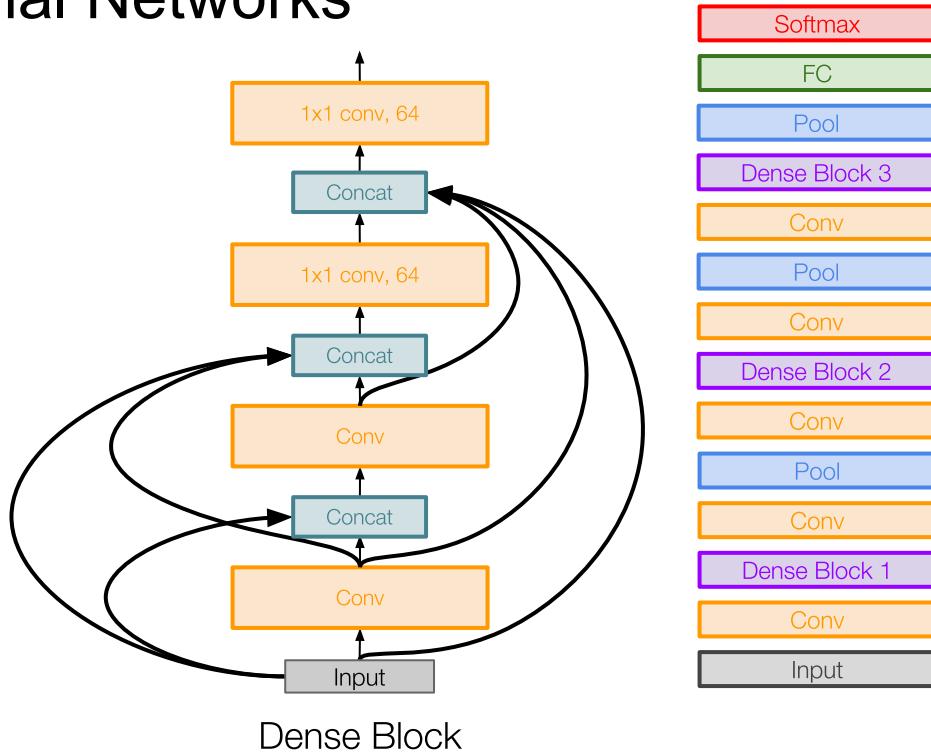
Figures copyright Larsson et al., 2017. Reproduced with permission.

Beyond ResNets...

Densely Connected Convolutional Networks

[Huang et al. 2017]

- Dense blocks where each layer is connected to every other layer in feedforward fashion
- Alleviates vanishing gradient, strengthens feature propagation, encourages feature reuse



Efficient networks...

SqueezeNet: AlexNet-level Accuracy With 50x Fewer Parameters and <0.5Mb Model Size

[Iandola et al. 2017]

- Fire modules consisting of a ‘squeeze’ layer with 1x1 filters feeding an ‘expand’ layer with 1x1 and 3x3 filters
- AlexNet level accuracy on ImageNet with 50x fewer parameters
- Can compress to 510x smaller than AlexNet (0.5Mb)

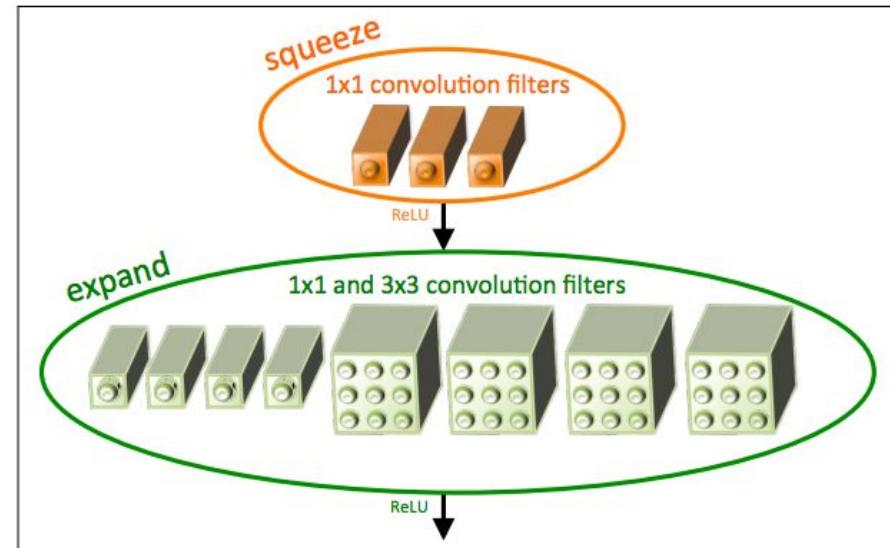


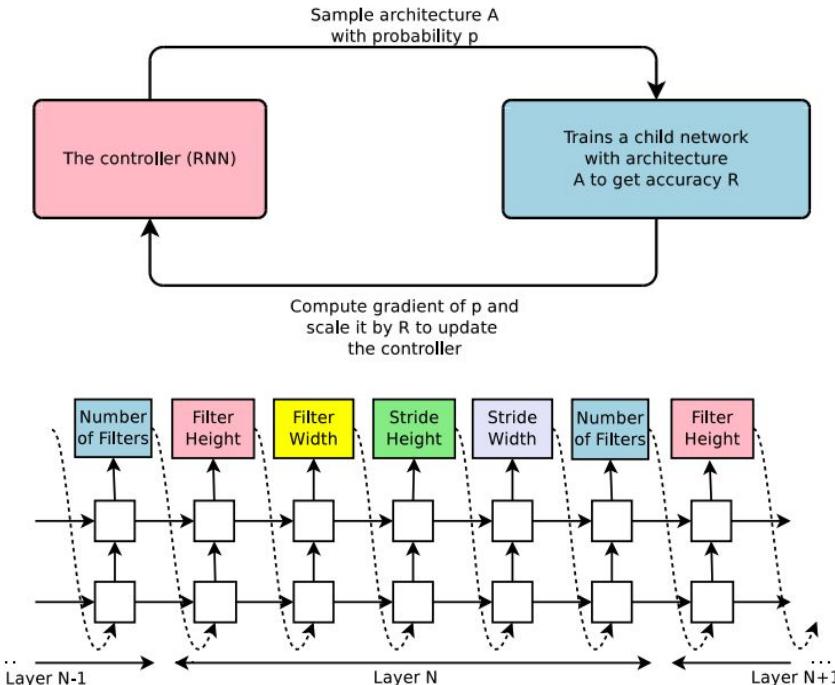
Figure copyright Iandola, Han, Moskewicz, Ashraf, Dally, Keutzer, 2017. Reproduced with permission.

Meta-learning: Learning to learn network architectures...

Neural Architecture Search with Reinforcement Learning (NAS)

[Zoph et al. 2016]

- “Controller” network that learns to design a good network architecture (output a string corresponding to network design)
- Iterate:
 - 1) Sample an architecture from search space
 - 2) Train the architecture to get a “reward” R corresponding to accuracy
 - 3) Compute gradient of sample probability, and scale by R to perform controller parameter update (i.e. increase likelihood of good architecture being sampled, decrease likelihood of bad architecture)

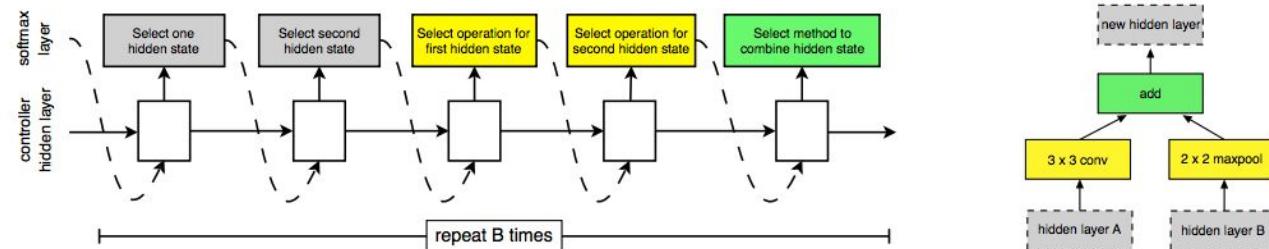
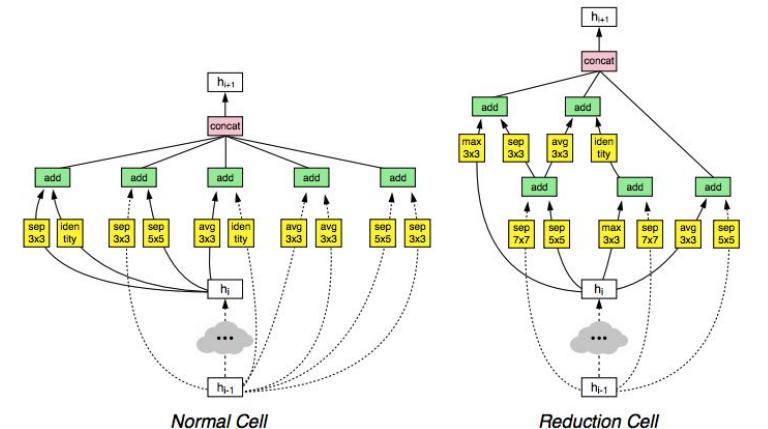


Meta-learning: Learning to learn network architectures...

Learning Transferable Architectures for Scalable Image Recognition

[Zoph et al. 2017]

- Applying neural architecture search (NAS) to a large dataset like ImageNet is expensive
- Design a search space of building blocks (“cells”) that can be flexibly stacked
- NASNet: Use NAS to find best cell structure on smaller CIFAR-10 dataset, then transfer architecture to ImageNet



Summary: CNN Architectures

Case Studies

- AlexNet
- VGG
- GoogLeNet
- ResNet

Also....

- NiN (Network in Network)
- Wide ResNet
- ResNeXT
- Stochastic Depth
- Squeeze-and-Excitation Network
- DenseNet
- FractalNet
- SqueezeNet
- NASNet

Summary: CNN Architectures

- VGG, GoogLeNet, ResNet all in wide use, available in model zoos
- ResNet current best default, also consider SENet when available
- Trend towards extremely deep networks
- Significant research centers around design of layer / skip connections and improving gradient flow
- Efforts to investigate necessity of depth vs. width and residual connections
- Even more recent trend towards meta-learning
- Next time: Recurrent neural networks

Lecture 10: Recurrent Neural Networks

Administrative

A1 regrade deadline is tonight

A2 due yesterday

Redeem your Google Cloud coupons by
Sunday 5/6

Administrative: Midterm

In-class midterm on Tuesday!

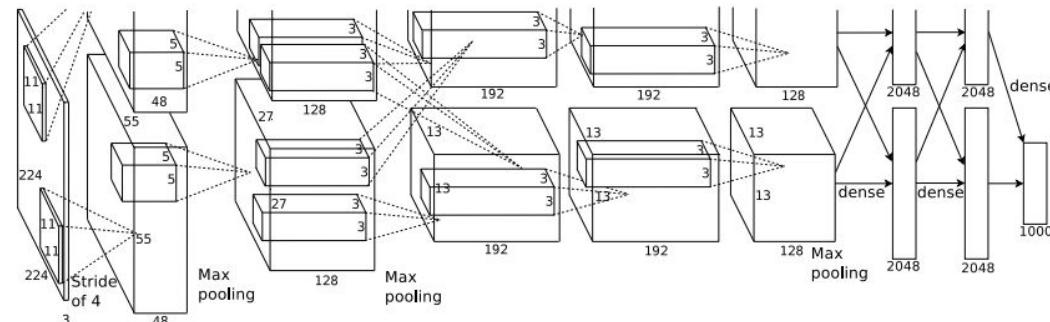
Details on Piazza

If you need an alternate exam time then let us know by tomorrow

Midterm review session tomorrow

Last Time: CNN Architectures

AlexNet



Revolution of Depth

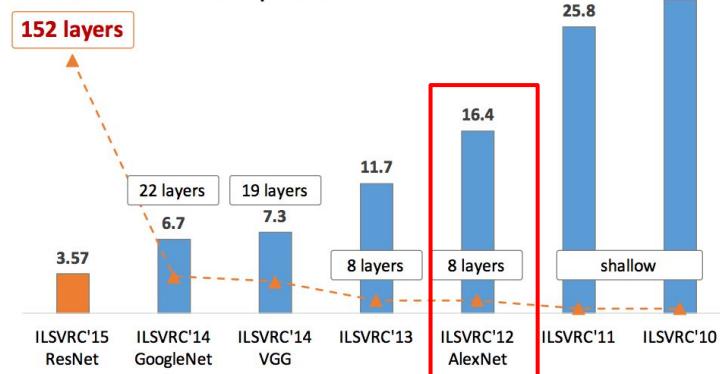
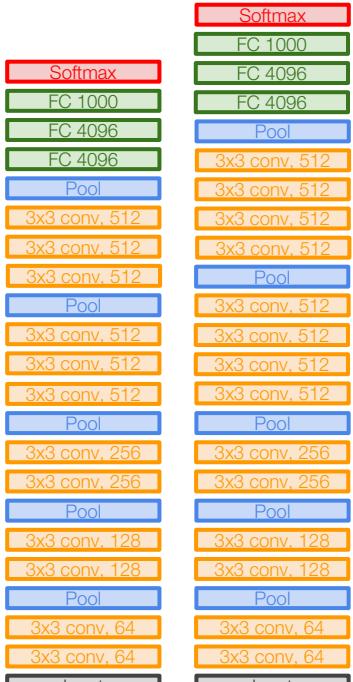


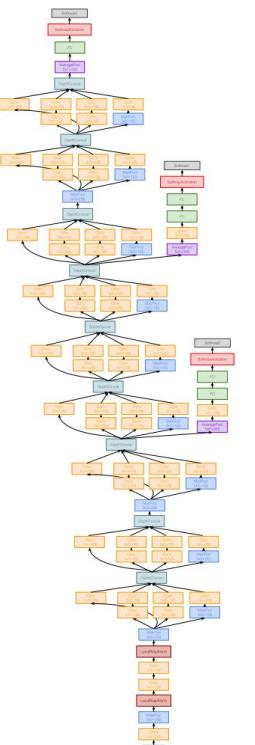
Figure copyright Kaiming He, 2016. Reproduced with permission.

Last Time: CNN Architectures



VGG16

VGG19



GoogLeNet

Revolution of Depth

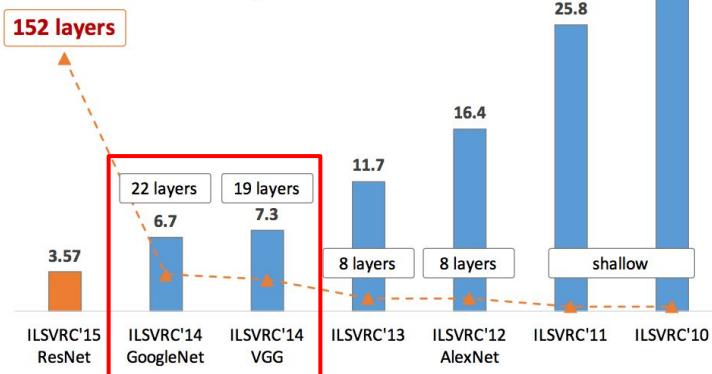
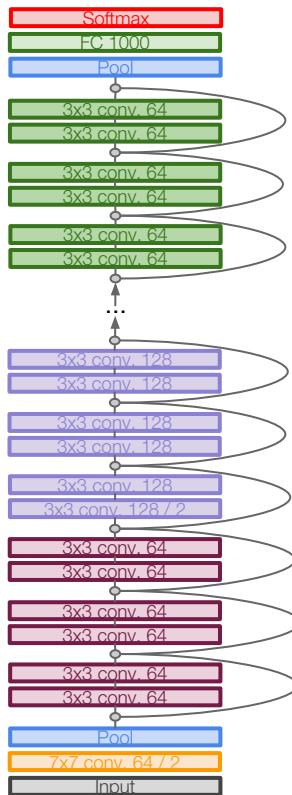
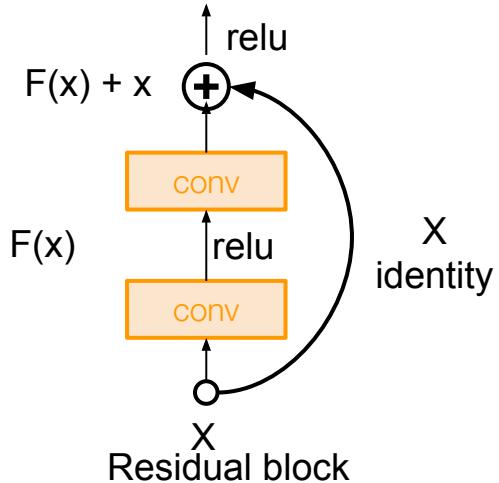


Figure copyright Kaiming He, 2016. Reproduced with permission.

Last Time: CNN Architectures



Revolution of Depth

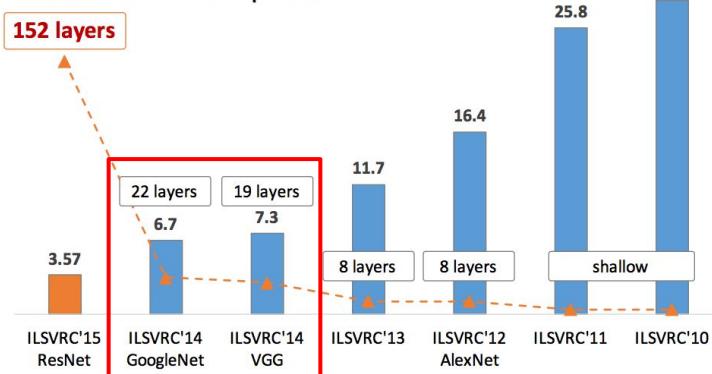
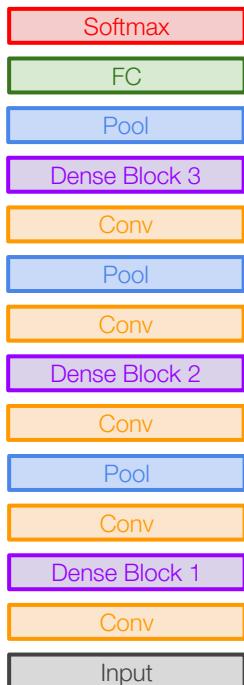
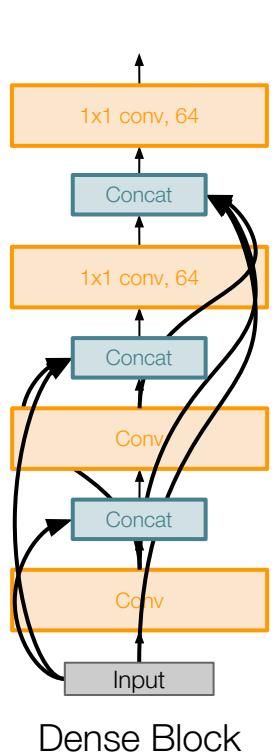


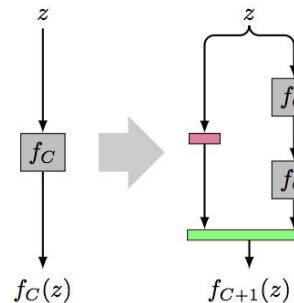
Figure copyright Kaiming He, 2016. Reproduced with permission.

DenseNet



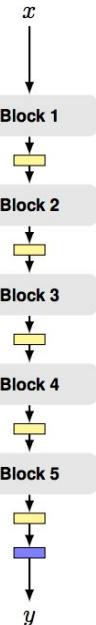
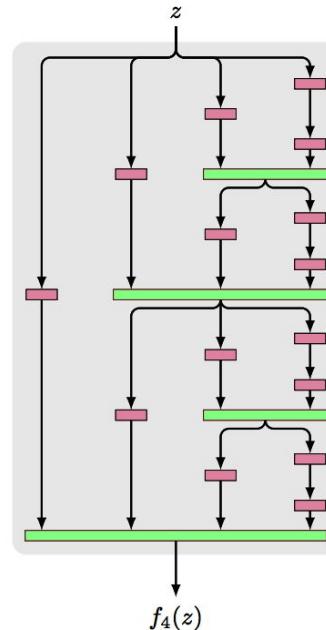
FractalNet

Fractal Expansion Rule



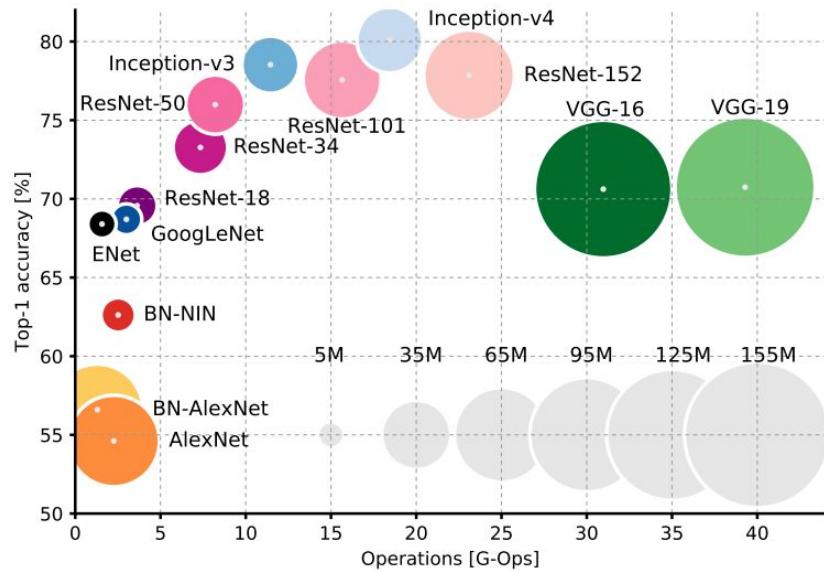
Layer Key

- Convolution (pink rectangle)
- Join (green rectangle)
- Pool (yellow rectangle)
- Prediction (blue rectangle)

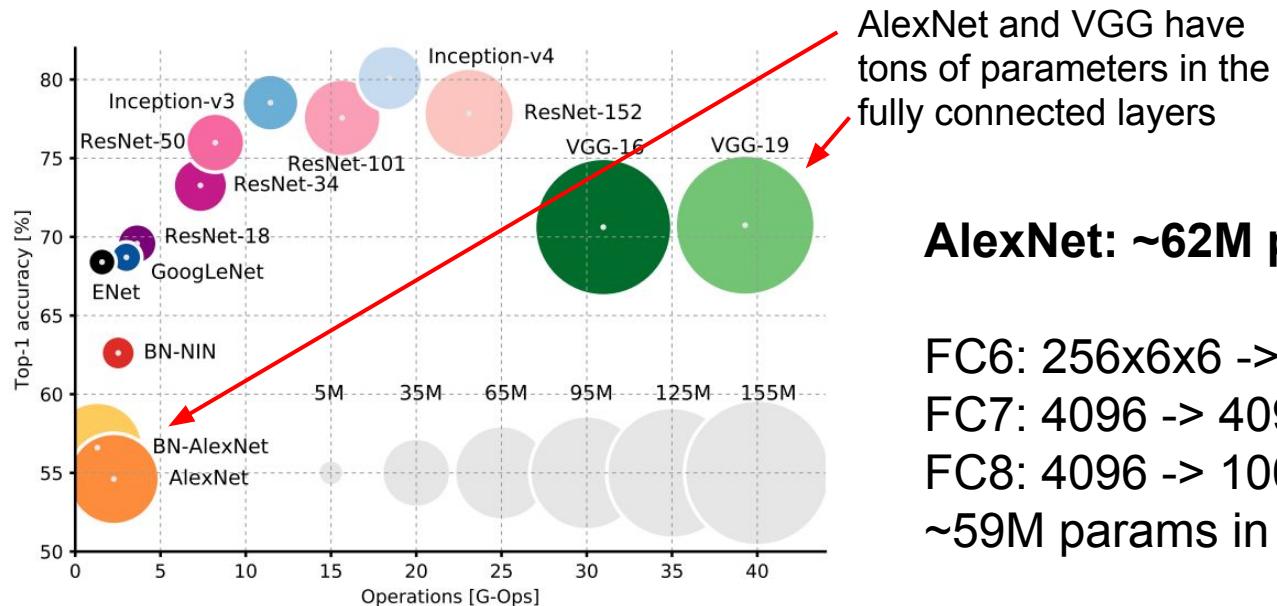


Figures copyright Larsson et al., 2017. Reproduced with permission.

Last Time: CNN Architectures



Last Time: CNN Architectures



AlexNet and VGG have tons of parameters in the fully connected layers

AlexNet: ~62M parameters

FC6: 256x6x6 \rightarrow 4096: 38M params

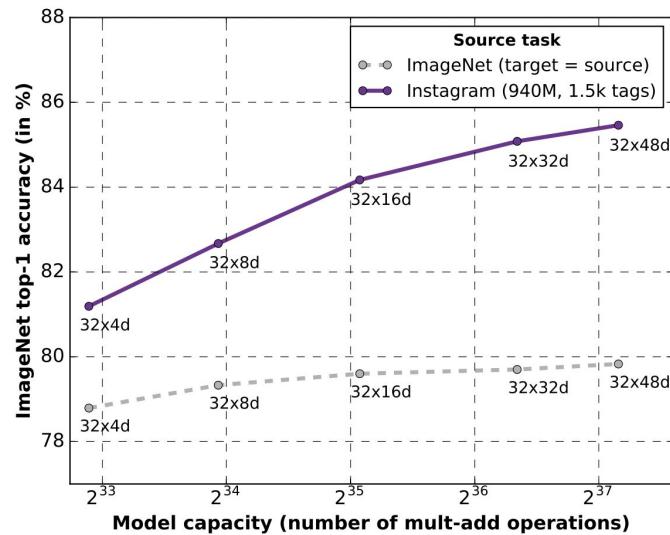
FC7: 4096 \rightarrow 4096: 17M params

FC8: 4096 \rightarrow 1000: 4M params

~59M params in FC layers!

ImageNet pretraining -> Instagram pretraining

Bigger models are saturated
on ImageNet, but with more
data bigger models do better



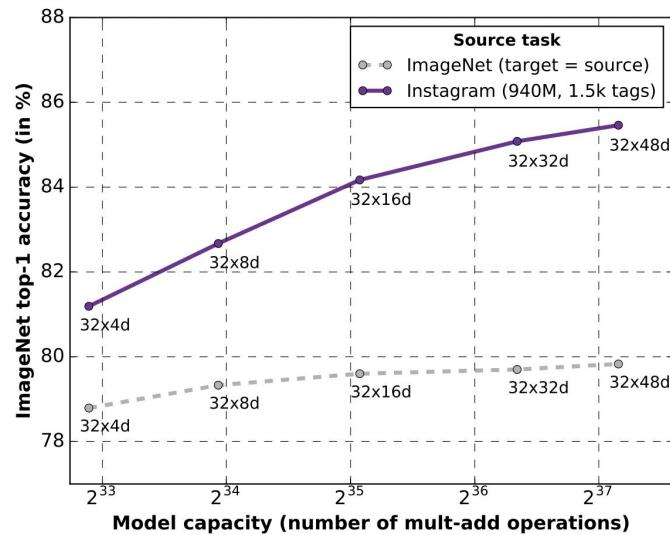
Biggest network was pretrained on
3.5B Instagram images

Trained on 336 GPUs for 22 days

Mahajan et al, "Exploring the Limits of Weakly Supervised Pretraining", arXiv 2018

ImageNet pretraining -> Instagram pretraining

Bigger models are saturated
on ImageNet, but with more
data bigger models do better



Biggest network was pretrained on
3.5B Instagram images

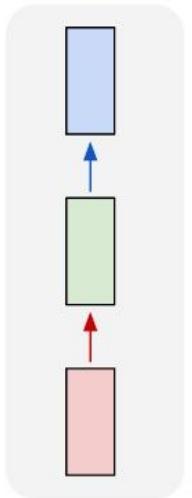
Trained on 336 GPUs for 22 days
≈ \$129,000 on Google Cloud

Mahajan et al, "Exploring the Limits of Weakly Supervised Pretraining", arXiv 2018

Today: Recurrent Neural Networks

“Vanilla” Neural Network

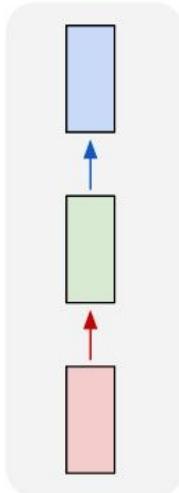
one to one



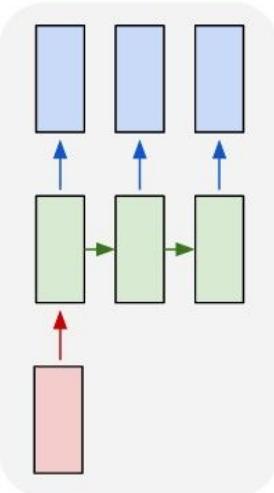
Vanilla Neural Networks

Recurrent Neural Networks: Process Sequences

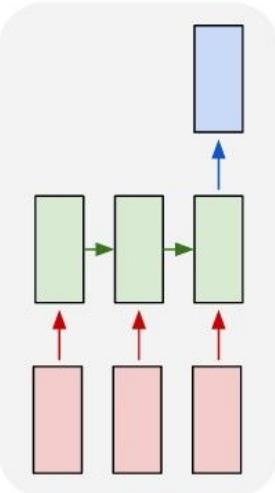
one to one



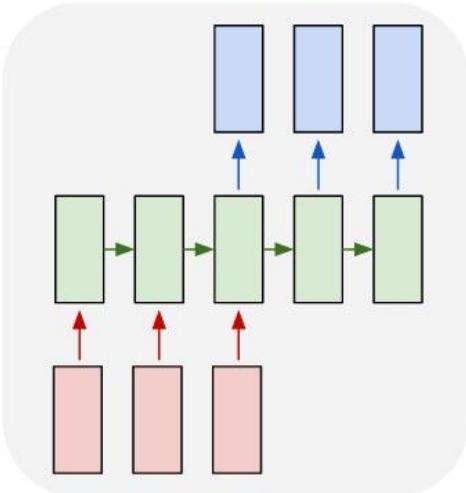
one to many



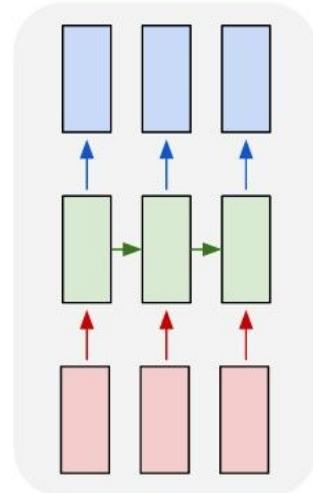
many to one



many to many



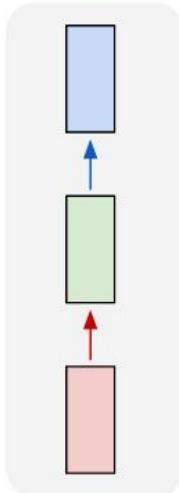
many to many



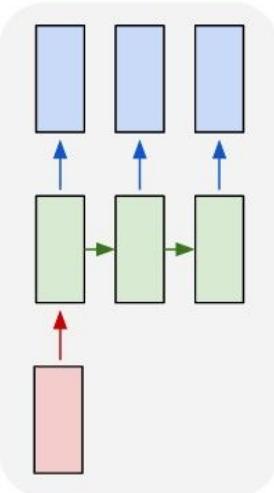
→
e.g. **Image Captioning**
image -> sequence of words

Recurrent Neural Networks: Process Sequences

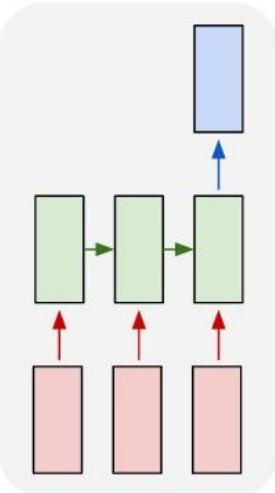
one to one



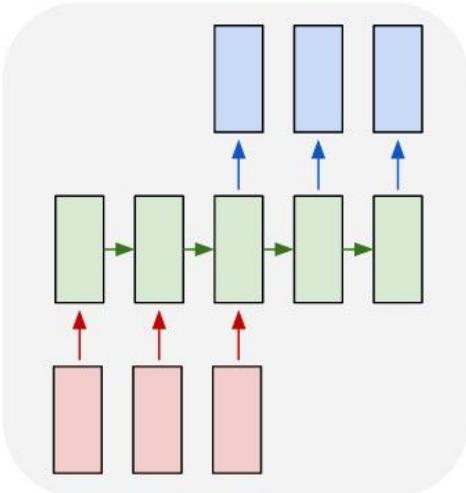
one to many



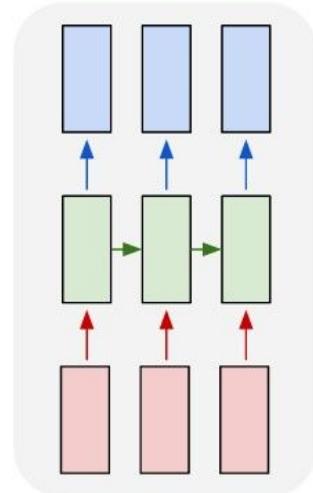
many to one



many to many



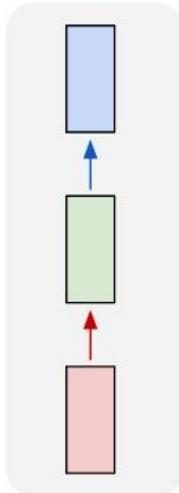
many to many



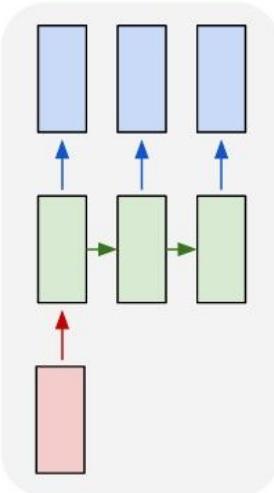
e.g. **Sentiment Classification**
sequence of words -> sentiment

Recurrent Neural Networks: Process Sequences

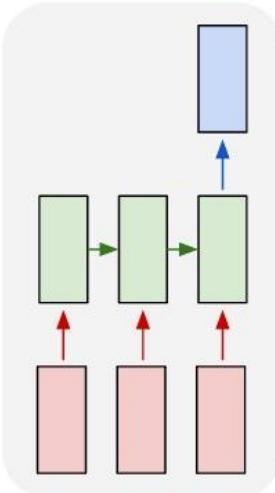
one to one



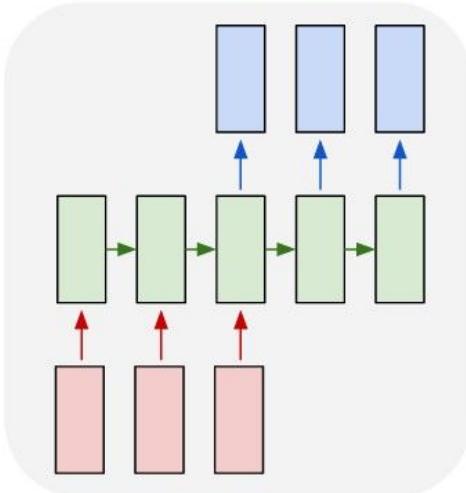
one to many



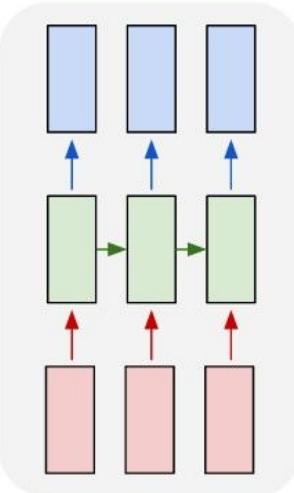
many to one



many to many



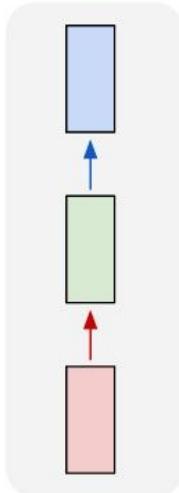
many to many



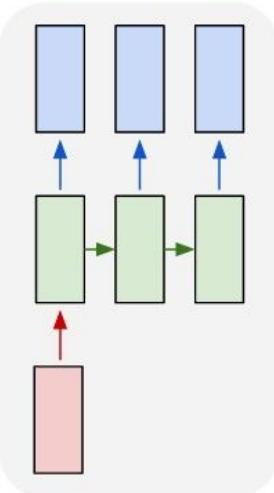
↑
e.g. **Machine Translation**
seq of words -> seq of words

Recurrent Neural Networks: Process Sequences

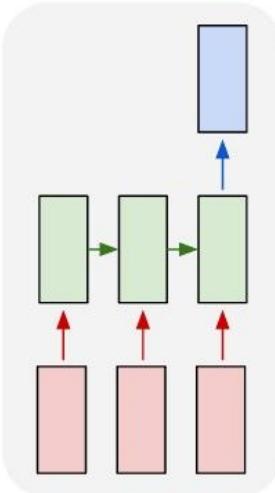
one to one



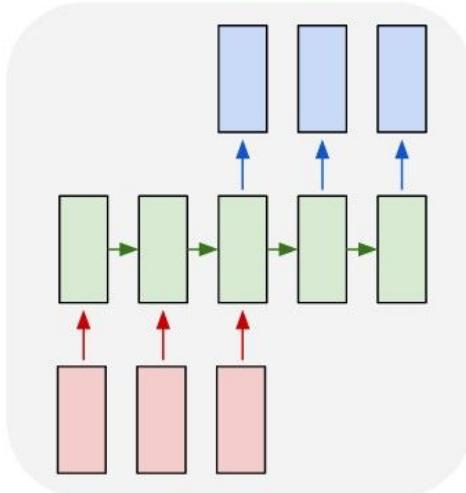
one to many



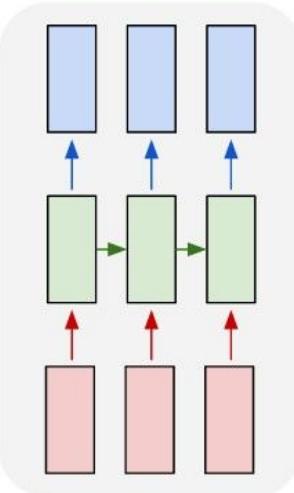
many to one



many to many



many to many



e.g. Video classification on frame level

Sequential Processing of Non-Sequence Data

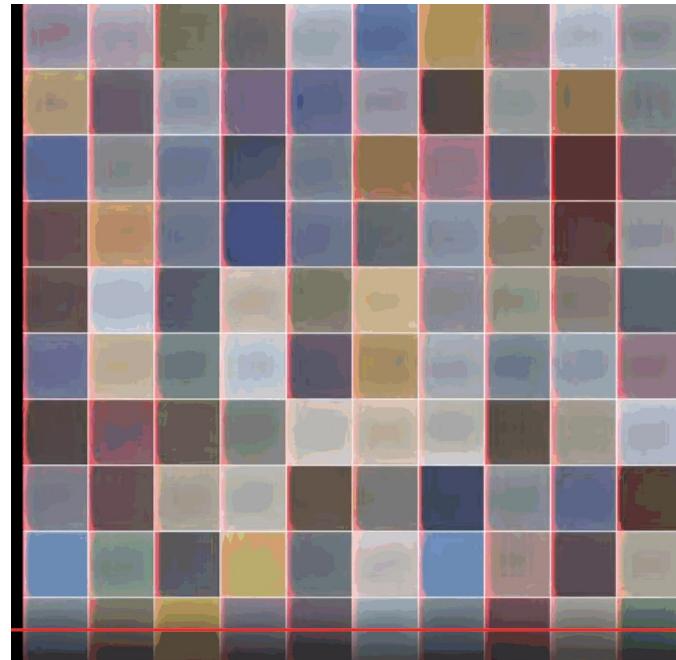
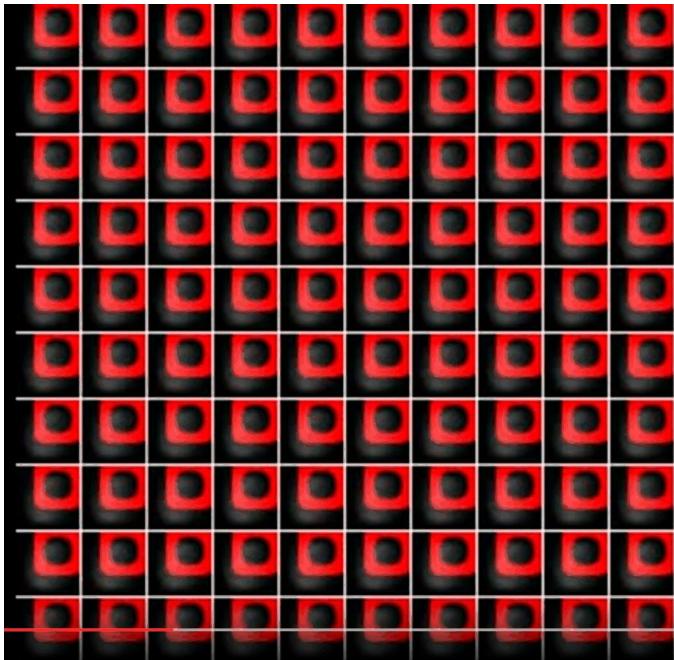
Classify images by taking a series of “glimpses”



Ba, Mnih, and Kavukcuoglu, “Multiple Object Recognition with Visual Attention”, ICLR 2015.
Gregor et al, “DRAW: A Recurrent Neural Network For Image Generation”, ICML 2015
Figure copyright Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra, 2015. Reproduced with permission.

Sequential Processing of Non-Sequence Data

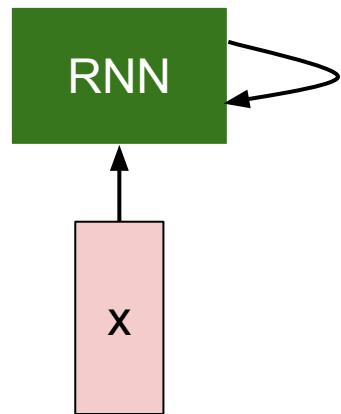
Generate images one piece at a time!



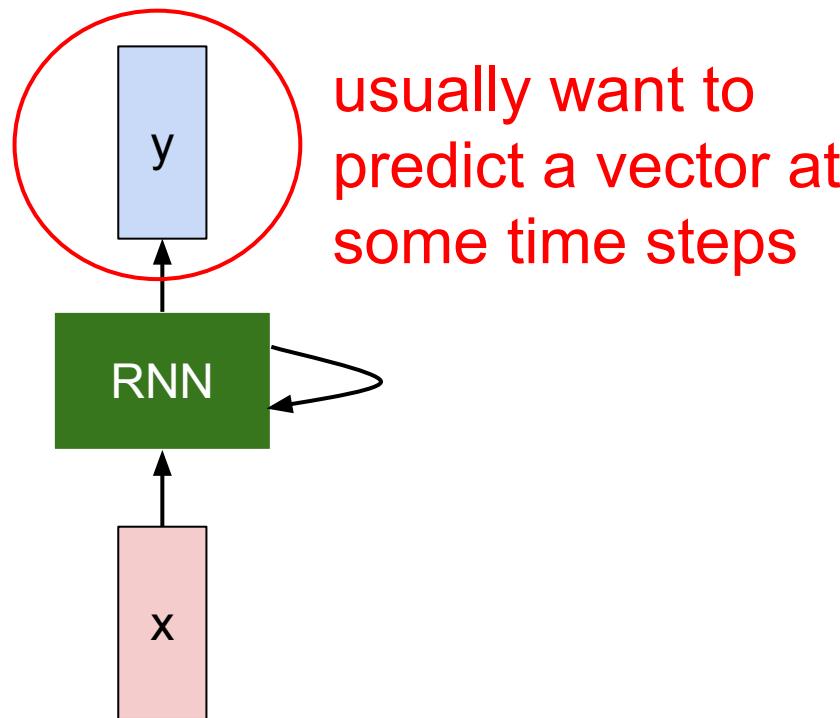
Gregor et al, "DRAW: A Recurrent Neural Network For Image Generation , ICML 2015

Figure copyright Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra, 2015. Reproduced with permission.

Recurrent Neural Network

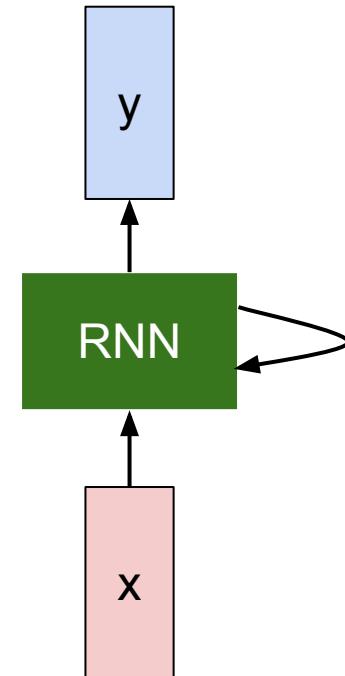


Recurrent Neural Network



Recurrent Neural Network

We can process a sequence of vectors \mathbf{x} by applying a **recurrence formula** at every time step:

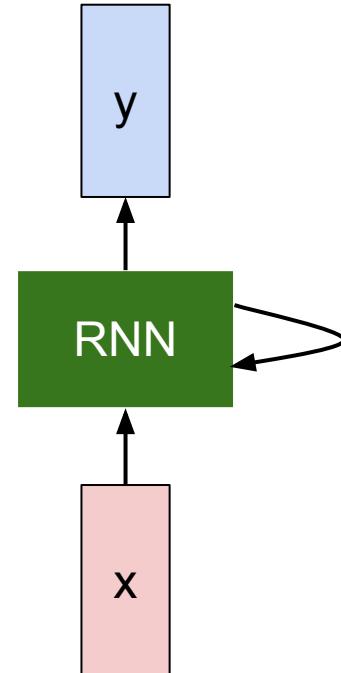


Recurrent Neural Network

We can process a sequence of vectors \mathbf{x} by applying a **recurrence formula** at every time step:

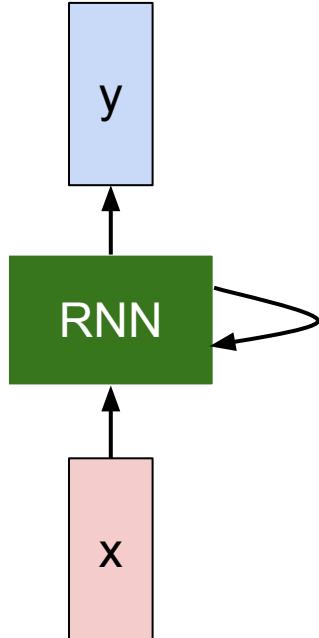
$$h_t = f_W(h_{t-1}, x_t)$$

Notice: the same function and the same set of parameters are used at every time step.



(Simple) Recurrent Neural Network

The state consists of a single “*hidden*” vector \mathbf{h} :



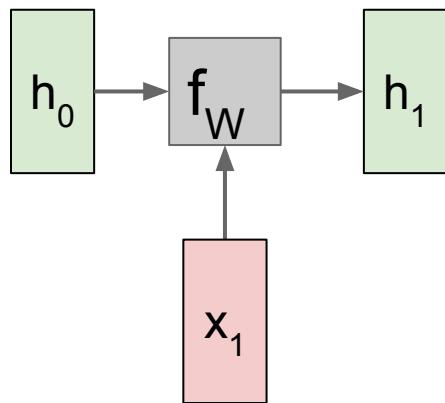
$$\mathbf{h}_t = f_W(\mathbf{h}_{t-1}, \mathbf{x}_t)$$

$$\mathbf{h}_t = \tanh(W_{hh}\mathbf{h}_{t-1} + W_{xh}\mathbf{x}_t)$$

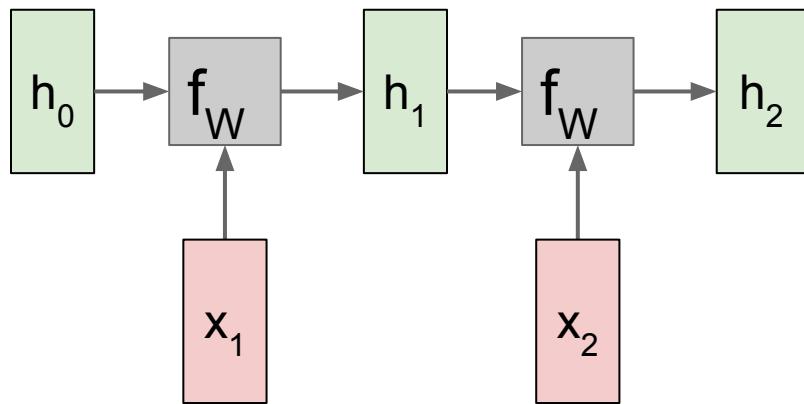
$$y_t = W_{hy}\mathbf{h}_t$$

Sometimes called a “Vanilla RNN” or an
“Elman RNN” after Prof. Jeffrey Elman

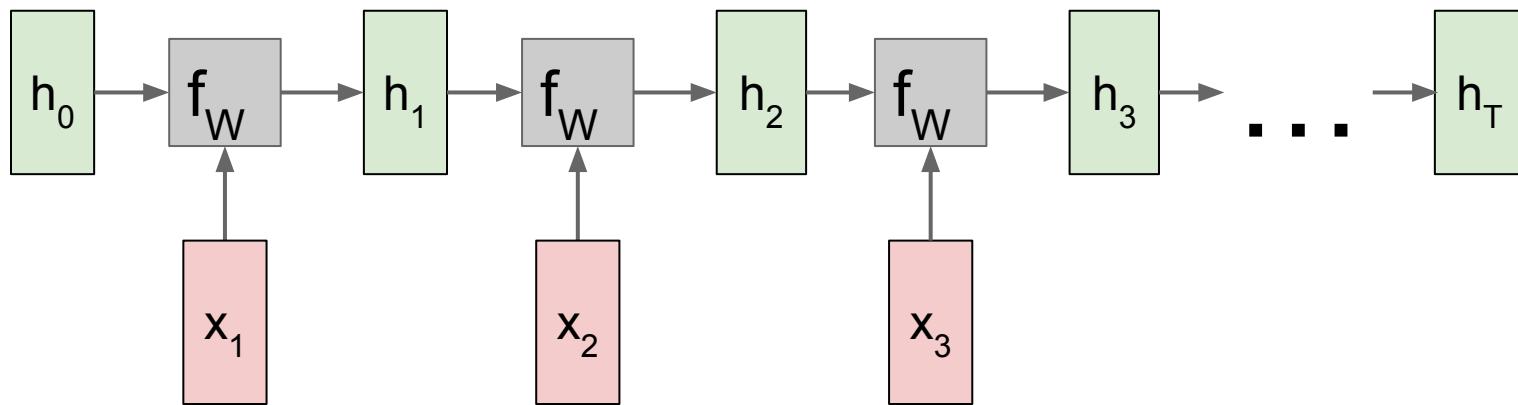
RNN: Computational Graph



RNN: Computational Graph

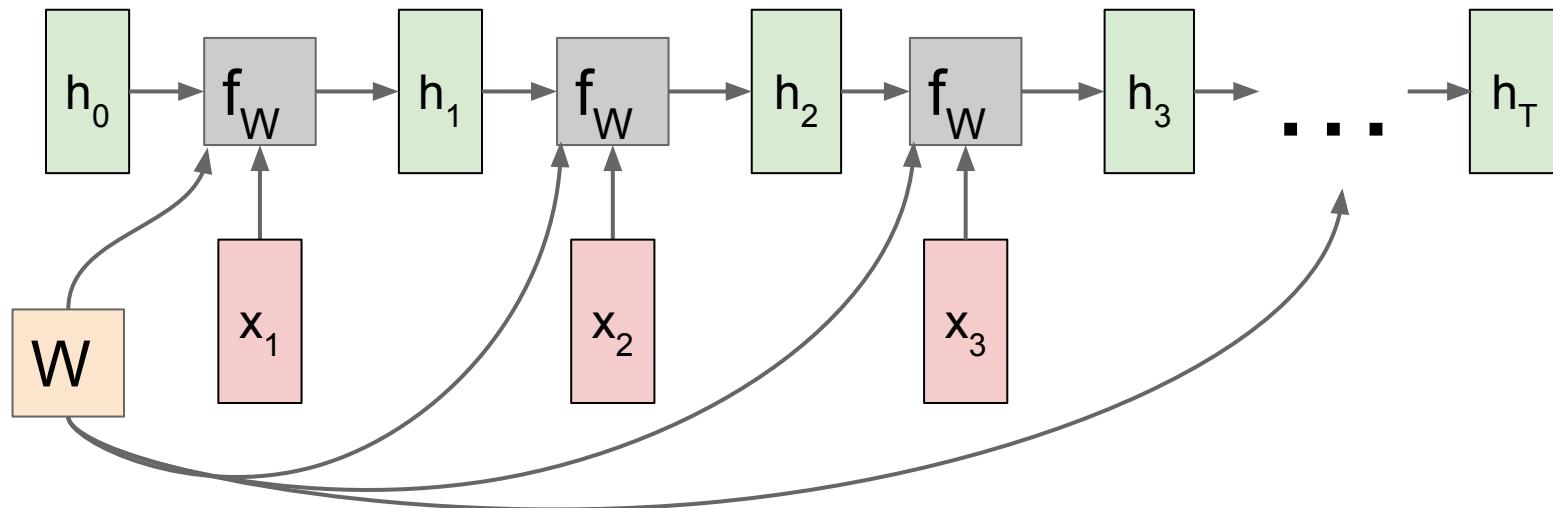


RNN: Computational Graph

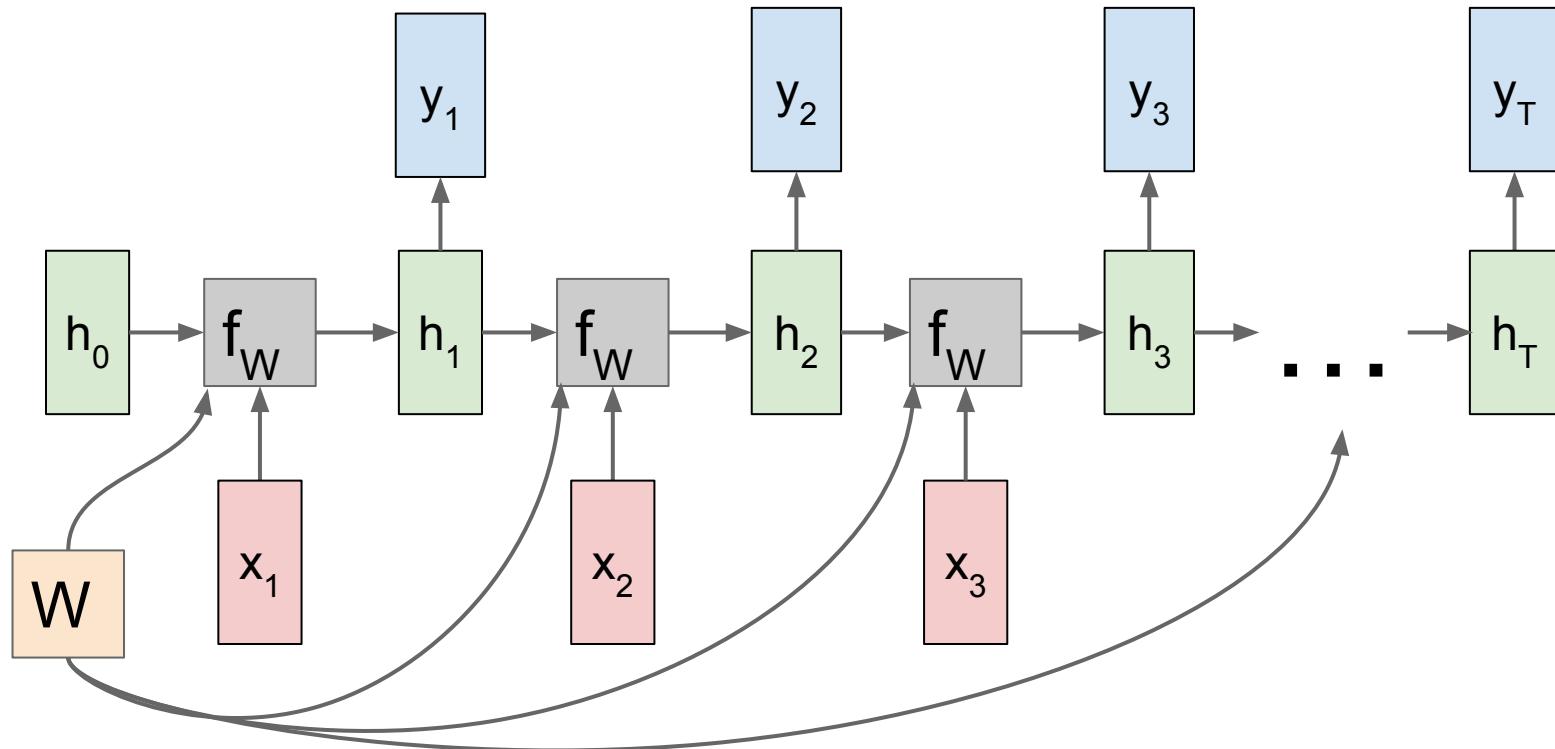


RNN: Computational Graph

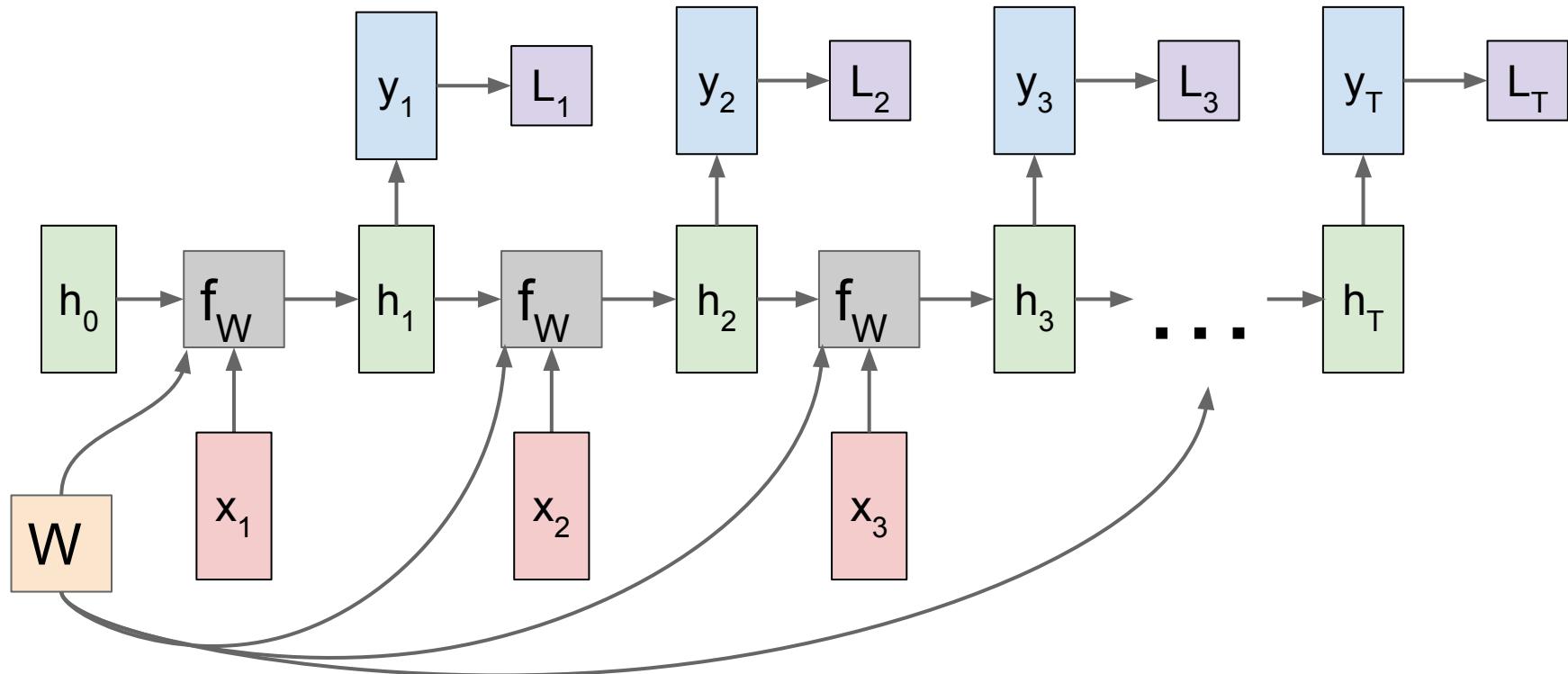
Re-use the same weight matrix at every time-step



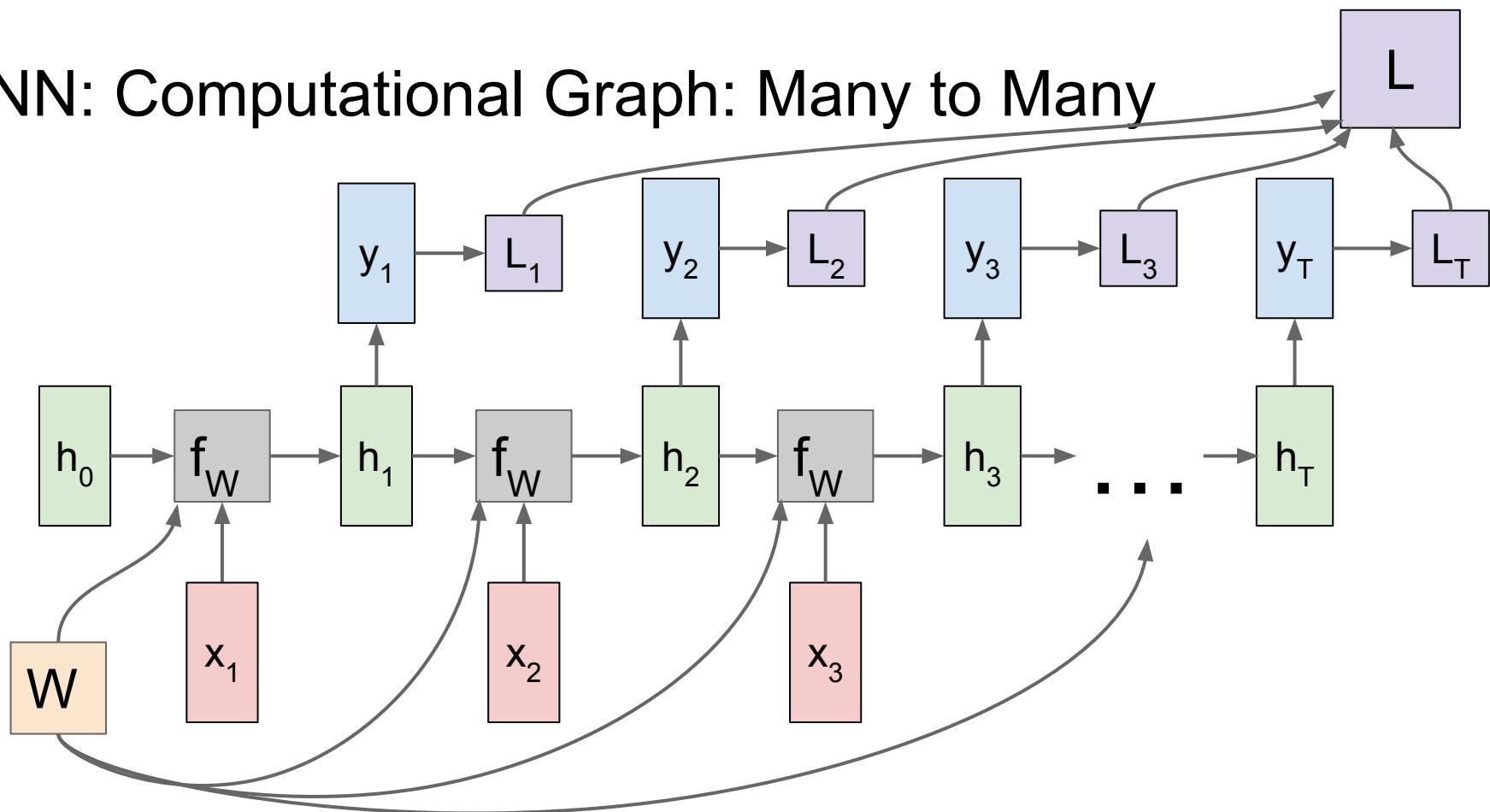
RNN: Computational Graph: Many to Many



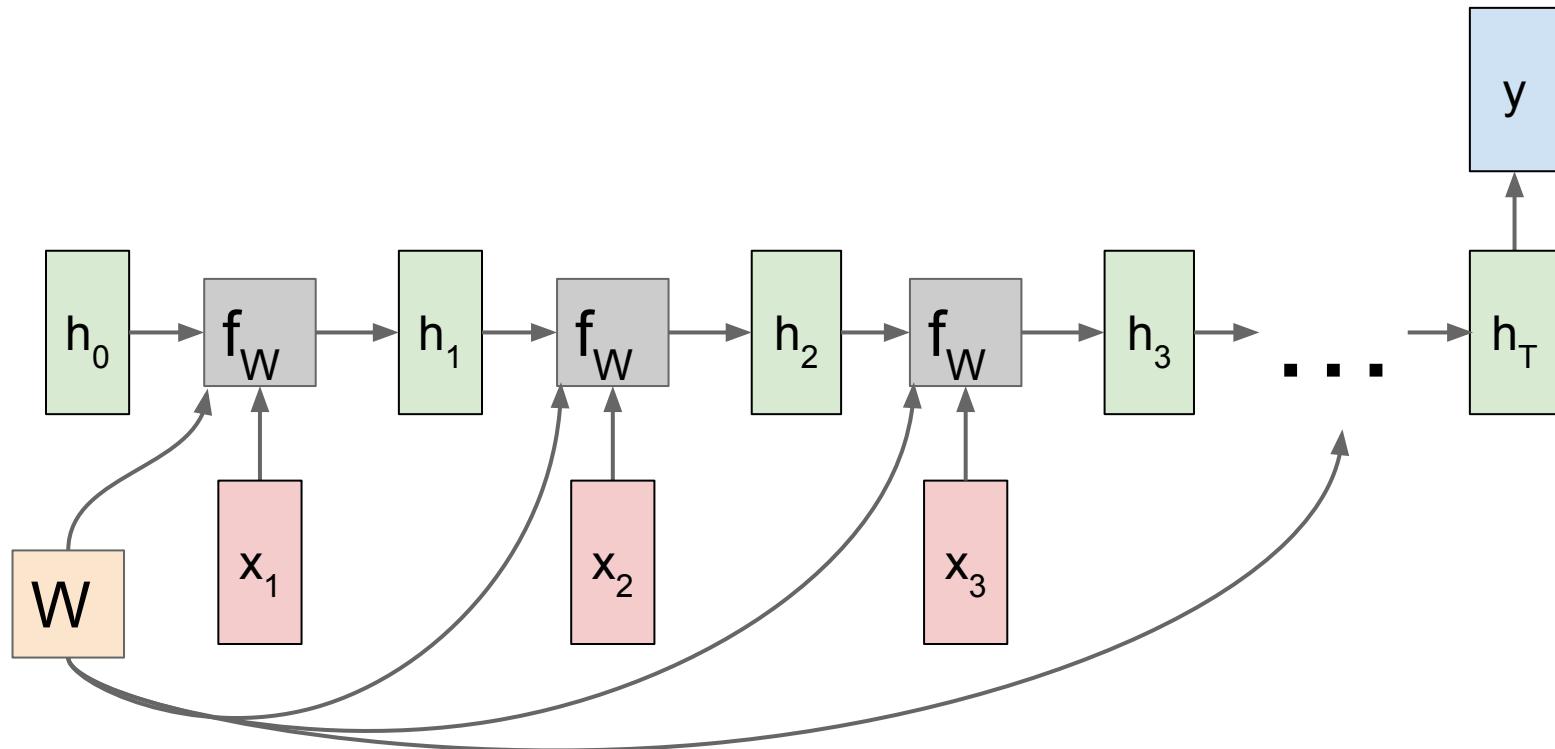
RNN: Computational Graph: Many to Many



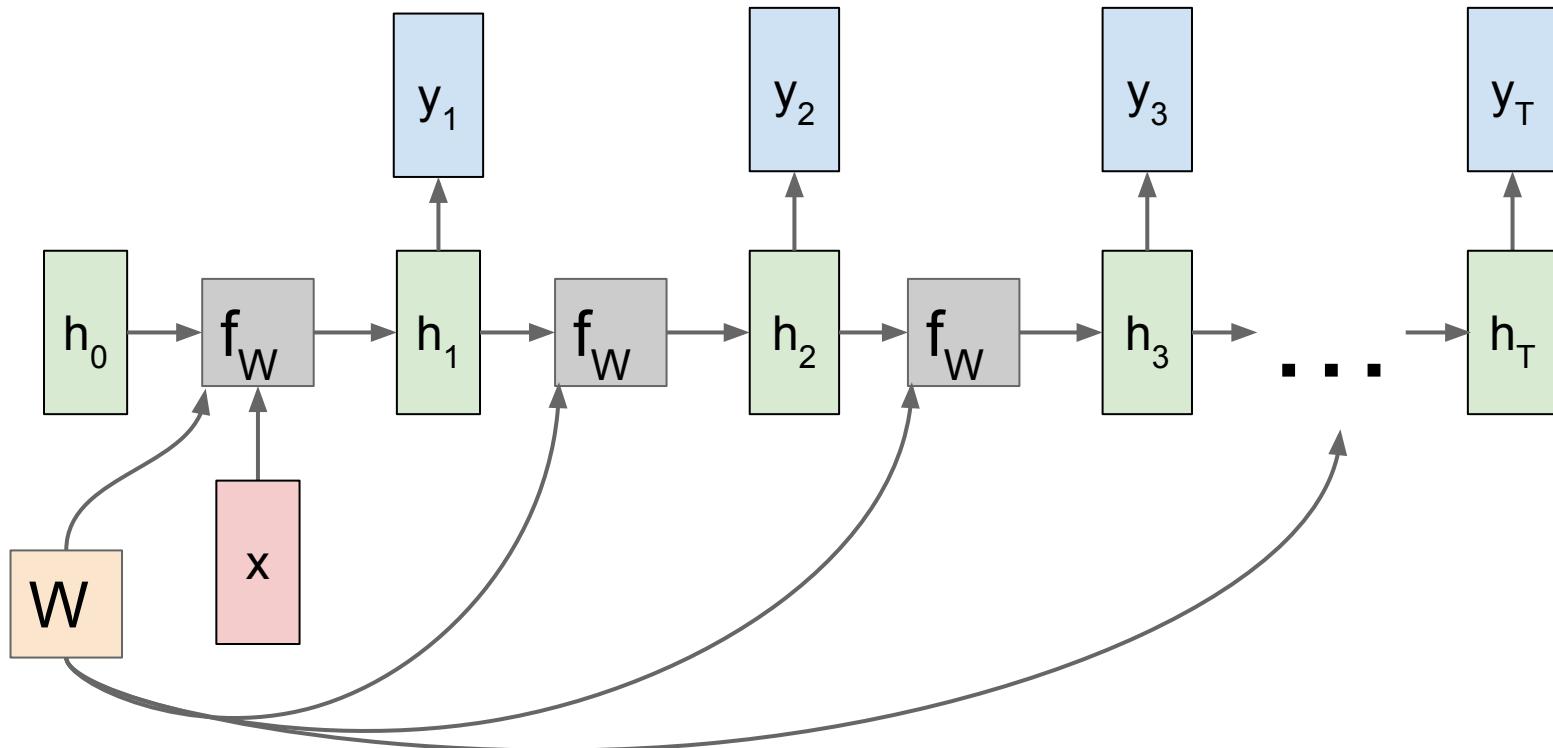
RNN: Computational Graph: Many to Many



RNN: Computational Graph: Many to One

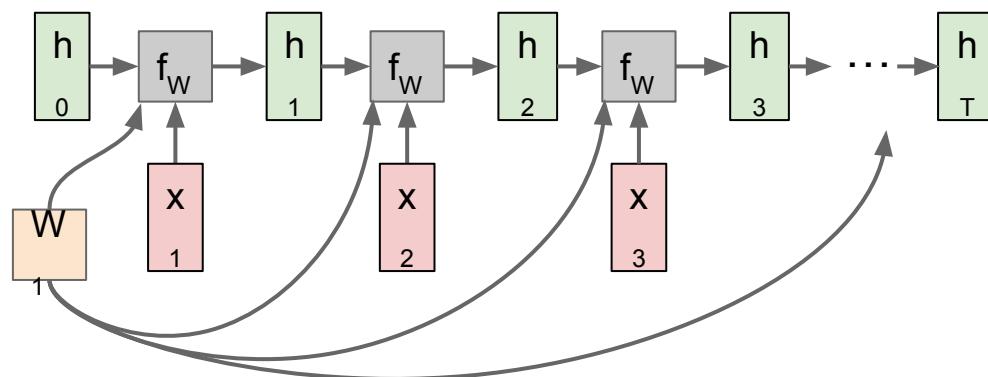


RNN: Computational Graph: One to Many



Sequence to Sequence: Many-to-one + one-to-many

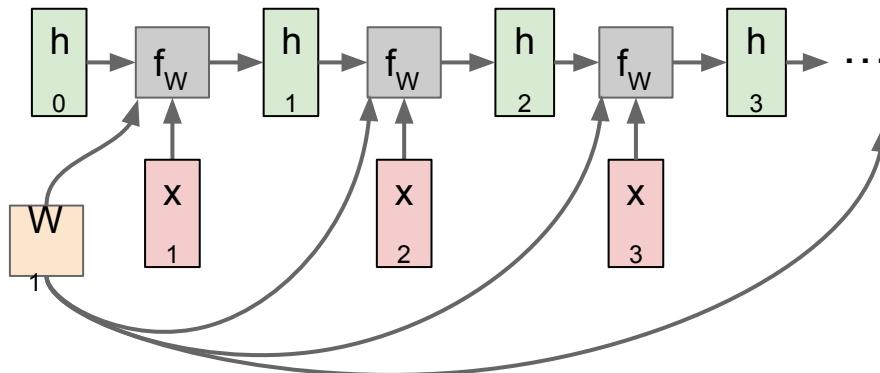
Many to one: Encode input sequence in a single vector



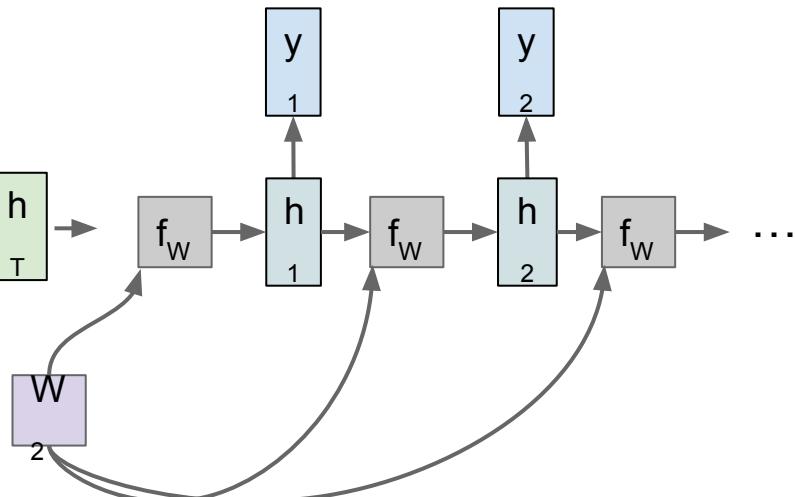
Sutskever et al, "Sequence to Sequence Learning with Neural Networks", NIPS 2014

Sequence to Sequence: Many-to-one + one-to-many

Many to one: Encode input sequence in a single vector



One to many: Produce output sequence from single input vector

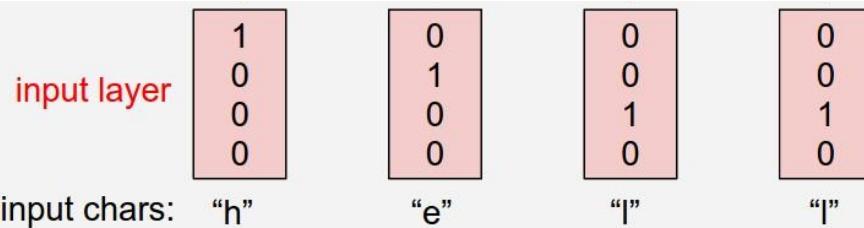


Sutskever et al, "Sequence to Sequence Learning with Neural Networks", NIPS 2014

Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

Example training
sequence:
“hello”

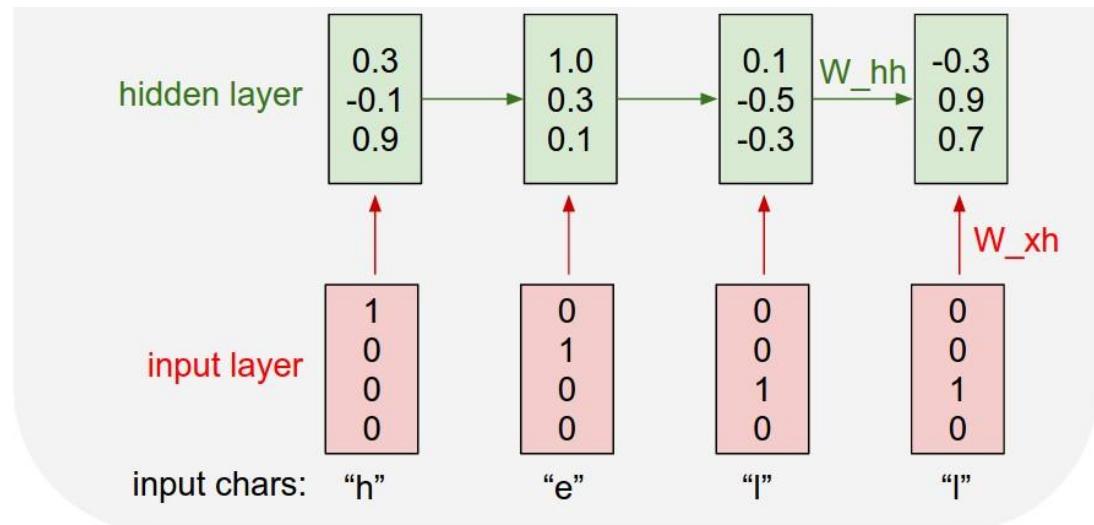


Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

Example training
sequence:
“hello”

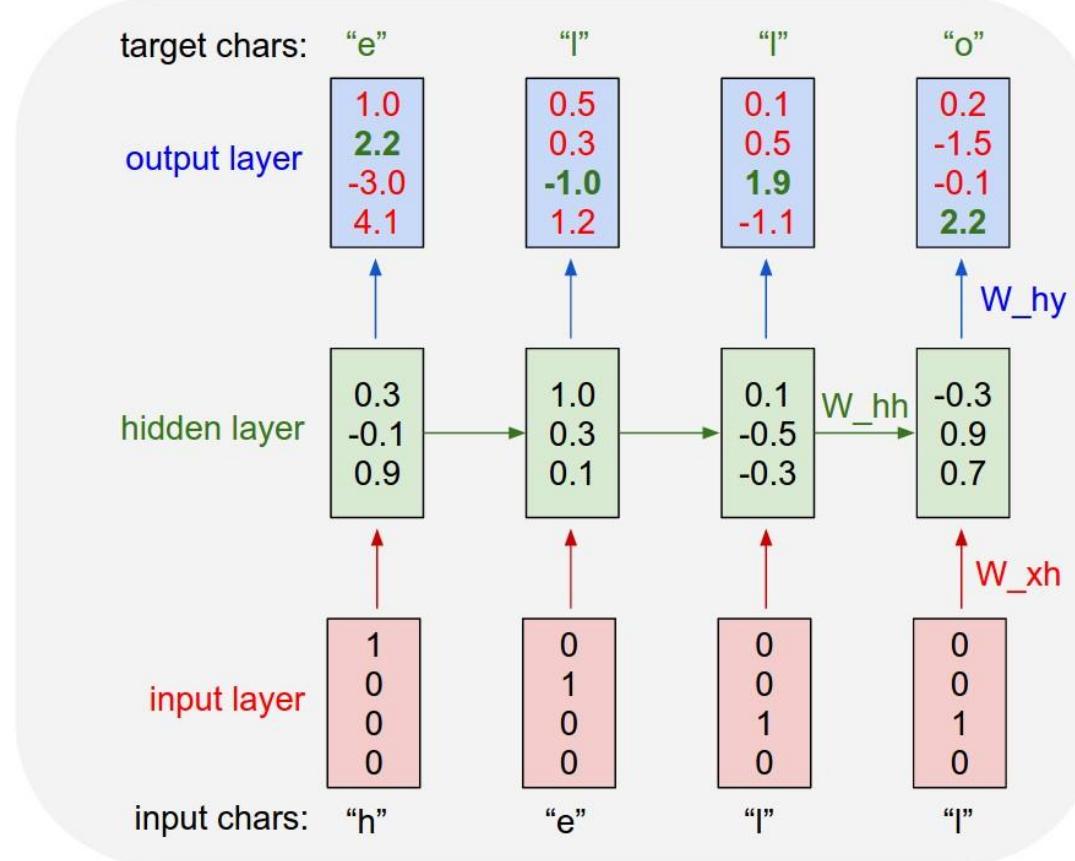
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$



Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

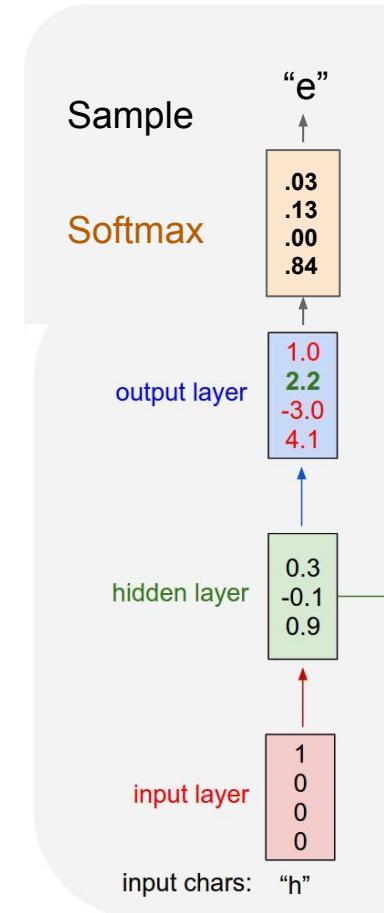
Example training
sequence:
“hello”



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

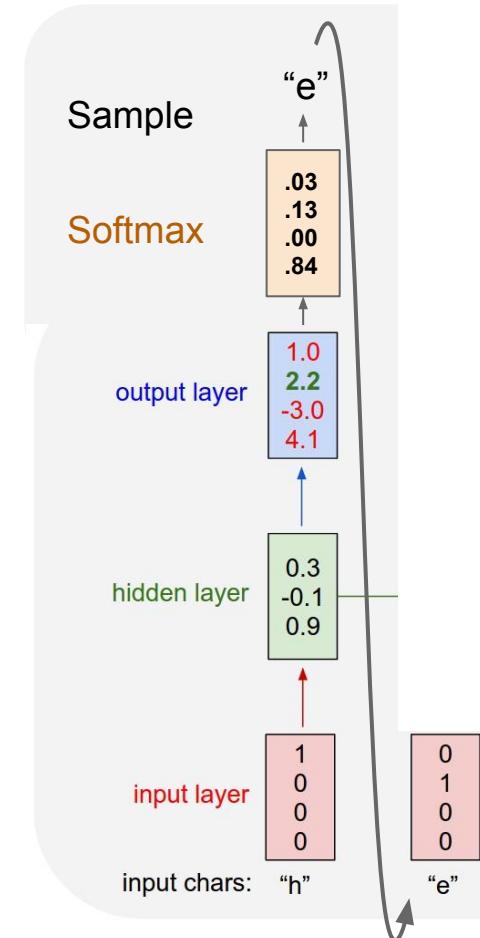
At test-time sample
characters one at a time,
feed back to model



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

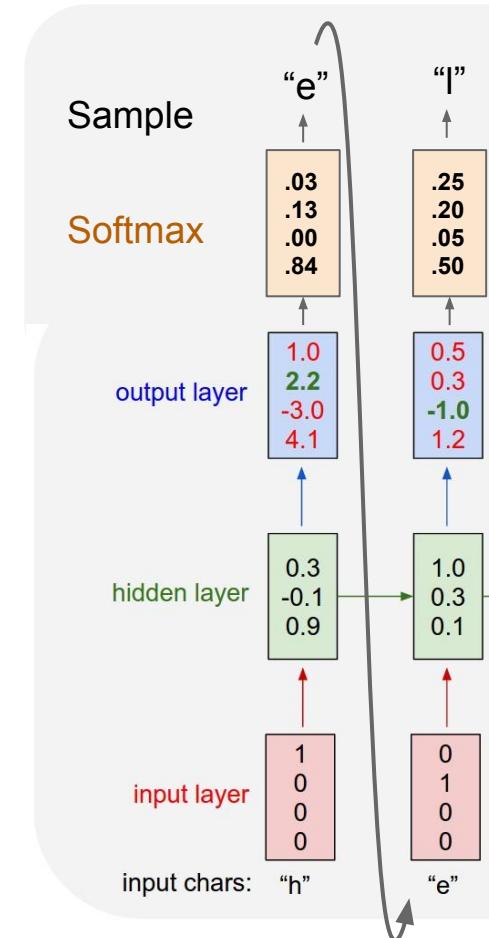
At test-time sample
characters one at a time,
feed back to model



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

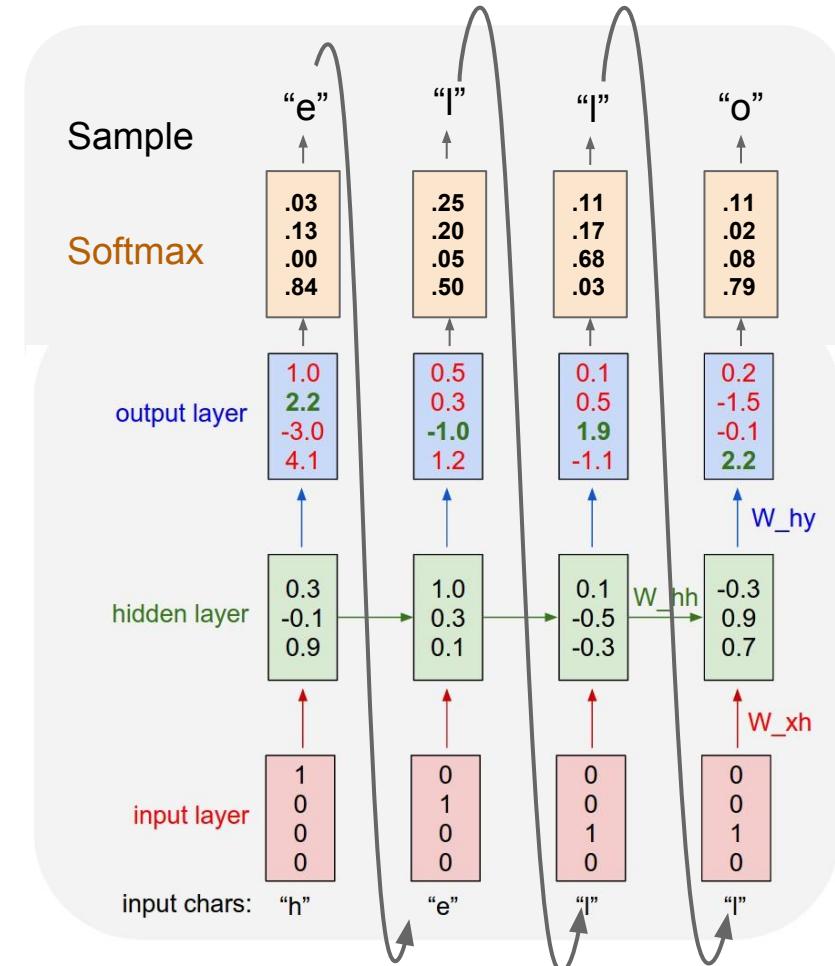
At test-time sample
characters one at a time,
feed back to model



Example: Character-level Language Model Sampling

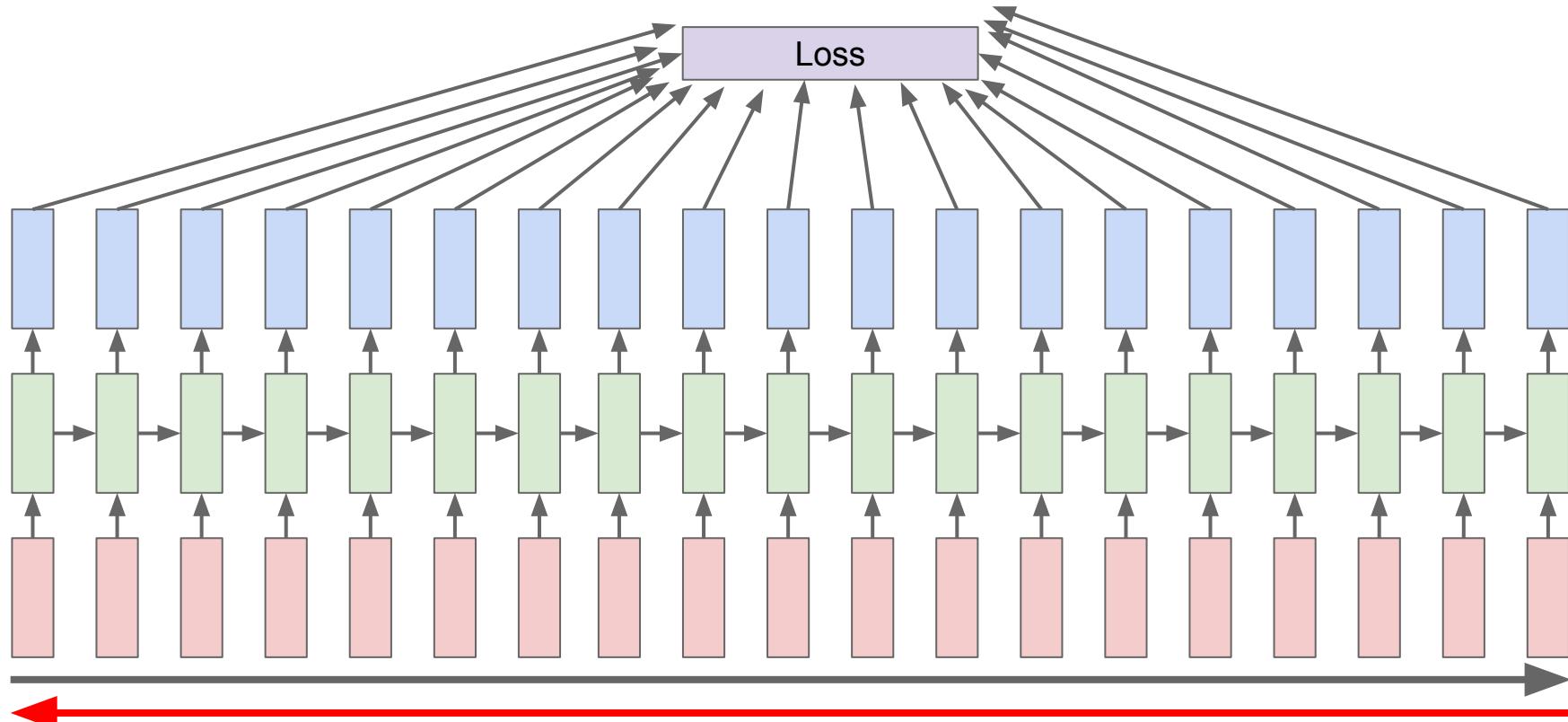
Vocabulary:
[h,e,l,o]

At test-time sample
characters one at a time,
feed back to model

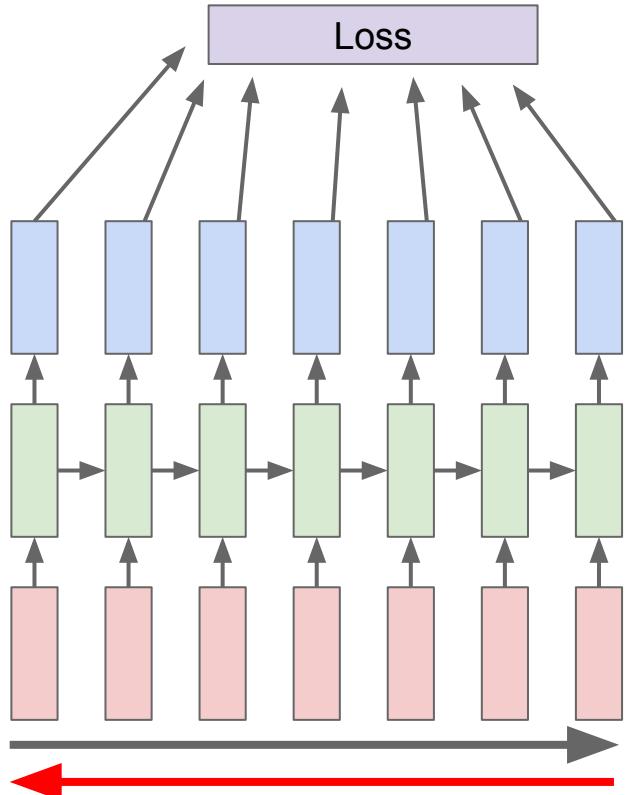


Backpropagation through time

Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient

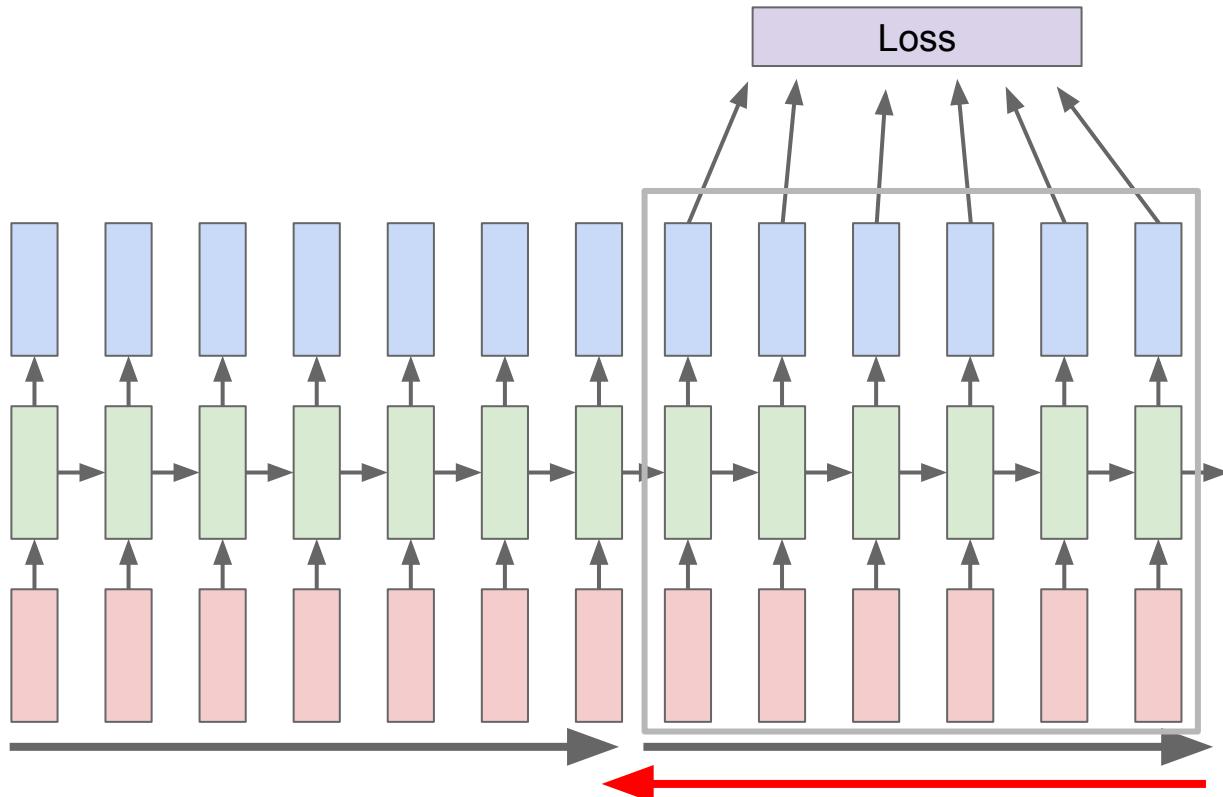


Truncated Backpropagation through time



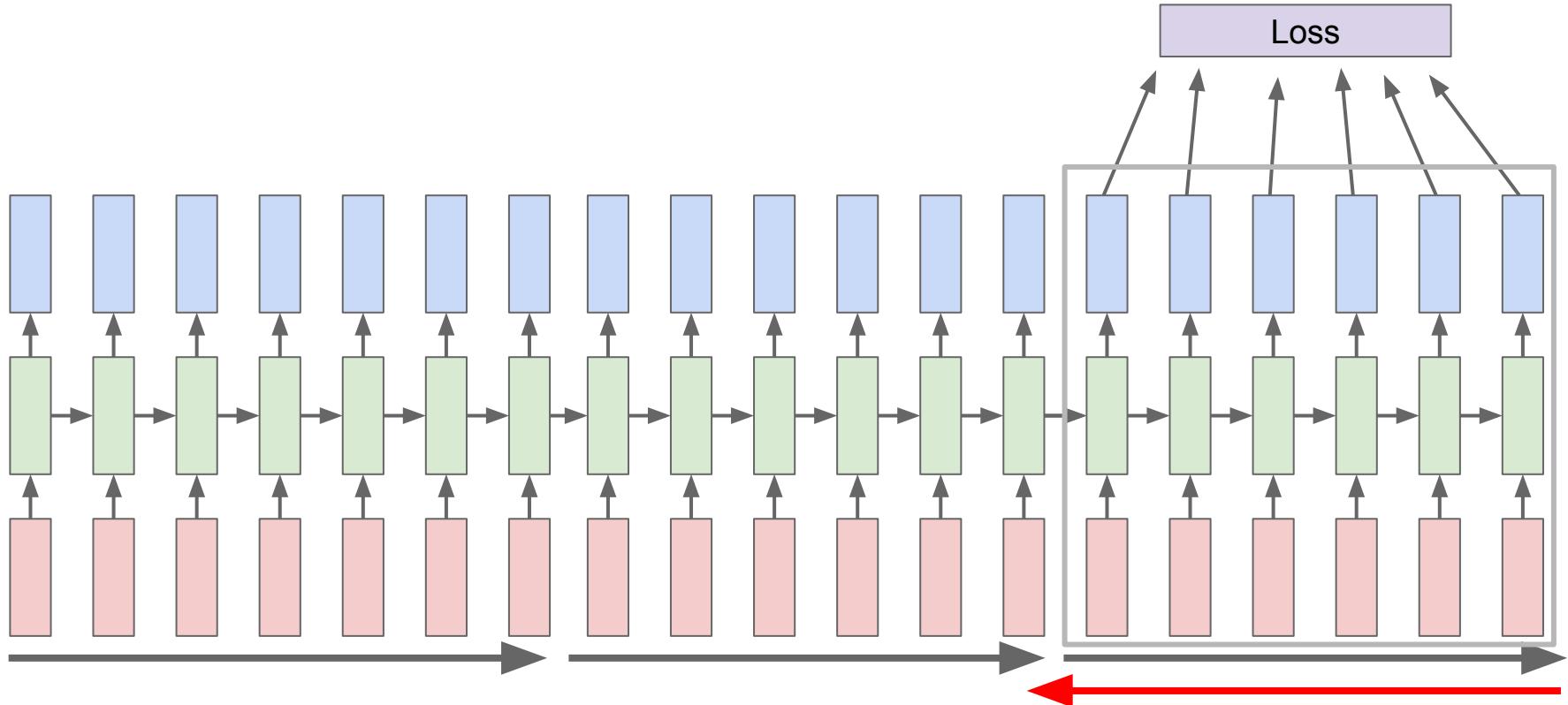
Run forward and backward
through chunks of the
sequence instead of whole
sequence

Truncated Backpropagation through time



Carry hidden states
forward in time forever,
but only backpropagate
for some smaller
number of steps

Truncated Backpropagation through time



min-char-rnn.py gist: 112 lines of Python

```
1  """
2  Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  BSD License
4  """
5  import numpy as np
6
7  # data I/O
8  data = open('input.txt', 'r').read() # should be simple plain text file
9  chars = list(set(data))
10 data_size, vocab_size = len(data), len(chars)
11 print('data has %d characters, %d unique.' % (data_size, vocab_size))
12 char_to_ix = {ch:i for i,ch in enumerate(chars)}
13 ix_to_char = {i:ch for i,ch in enumerate(chars)}
14
15 # hyperparameters
16 hidden_size = 100 # size of hidden layer of neurons
17 seq_length = 25 # number of steps to unroll the RNN for
18 learning_rate = 1e-1
19
20 # model parameters
21 wkh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
23 why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
24 bh = np.zeros((hidden_size, 1)) # hidden bias
25 by = np.zeros((vocab_size, 1)) # output bias
26
27 def lossFun(inputs, targets, hprev):
28     """
29     inputs,targets are both list of integers.
30     hprev is Hx1 array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33     xs, hs, ys, ps = {}, {}, {}, {}
34     hs[-1] = np.copy(hprev)
35     loss = 0
36     # forward pass
37     for t in xrange(len(inputs)):
38         xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
39         xs[t][inputs[t]] = 1
40         hs[t] = np.tanh(np.dot(wkh, xs[t]) + np.dot(whh, hs[t-1]) + bh) # hidden state
41         ys[t] = np.dot(why, hs[t]) - by # unnormalized log probabilities for next chars
42         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
43         loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
44
45         # backward pass: compute gradients going backwards
46         dwhx, dwhh, dwhy = np.zeros_like(wkh), np.zeros_like(whh), np.zeros_like(why)
47         dbh, dby = np.zeros_like(bh), np.zeros_like(by)
48         dhnext = np.zeros_like(hs[0])
49         for t in reversed(xrange(len(inputs))):
50             dy = np.copy(ps[t])
51             dy[targets[t]] -= 1 # backprop into y
52             dyb = np.dot(dy, hs[t].T)
53             dh = np.dot(why.T, dy) + dhnext # backprop into h
54             ddraw = (i - hs[t].T) * dh # backprop through tanh nonlinearity
55             ddbh = ddraw
56             dwhh += np.dot(ddraw, xs[t].T)
57             dwhy += np.dot(ddraw, hs[t-1].T)
58             dhnext = np.dot(why.T, ddraw)
59             for dparam in [dwhx, dwhh, dwhy, dbh, dby]:
60                 np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61
62     return loss, dwhx, dwhh, dwhy, dbh, dby, hs[len(inputs)-1]
```



```
63 def sample(h, seed_ix, n):
64     """
65     sample a sequence of integers from the model
66     h is memory state, seed_ix is seed letter for first time step
67     """
68     x = np.zeros((vocab_size, 1))
69     x[seed_ix] = 1
70     ixes = []
71     for t in xrange(n):
72         h = np.tanh(np.dot(wkh, x) + np.dot(whh, h) + bh)
73         y = np.dot(why, h) + by
74         p = np.exp(y) / np.sum(np.exp(y))
75         ix = np.random.choice(range(vocab_size), p=p.ravel())
76         x = np.zeros((vocab_size, 1))
77         x[ix] = 1
78         ixes.append(ix)
79
80     return ixes
81
82 n, p = 0, 0
83 mxwh, mwhh, mwhy = np.zeros_like(wkh), np.zeros_like(whh), np.zeros_like(why)
84 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
85 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
86 while True:
87     # prepare inputs (we're sweeping from left to right in steps seq_length long)
88     if p+seq_length >= len(data) or n == 0:
89         hprev = np.zeros((hidden_size,1)) # reset RNN memory
90         p = 0 # go from start of data
91     inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
92     targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
93
94     # sample from the model now and then
95     if n % 100 == 0:
96         sample_ix = sample(hprev, inputs[0], 200)
97         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
98         print('----\n%s\n----' % (txt,))
99
100     # forward seq_length characters through the net and fetch gradient
101     loss, dwhx, dwhh, dwhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
102     smooth_loss = smooth_loss * .999 + loss / n # print progress
103
104     # perform parameter update with Adagrad
105     for param, dparam, mem in zip([wkh, whh, why, bh, by],
106                                  [dwhx, dwhh, dwhy, dbh, dby],
107                                  [mxwh, mwhh, mwhy, mbh, mby]):
108         mem += dparam * dparam
109         param += -learning_rate * param / np.sqrt(mem + 1e-8) # adagrad update
110
111     p += seq_length # move data pointer
112     n += 1 # iteration counter
```

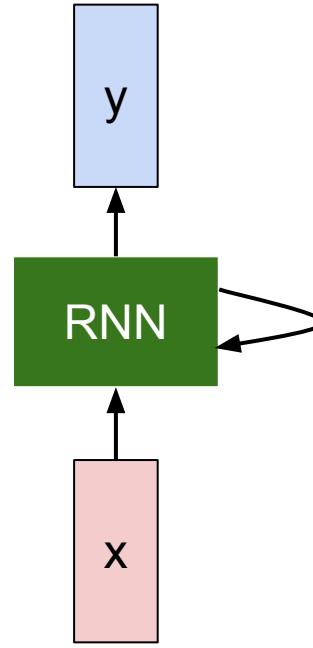
(<https://gist.github.com/karpathy/d4dee566867f8291f086>)

THE SONNETS

by William Shakespeare

From fairest creatures we desire increase,
That thereby beauty's rose might never die,
But as the riper should by time decease,
His tender heir might bear his memory:
But thou, contracted to thine own bright eyes,
Feed'st thy light's flame with self-substantial fuel,
Making a famine where abundance lies,
Thyself thy foe, to thy sweet self too cruel:
Thou that art now the world's fresh ornament,
And only herald to the gaudy spring,
Within thine own bud buriest thy content,
And tender churl mak'st waste in niggarding:
 Pity the world, or else this glutton be,
 To eat the world's due, by the grave and thee.

When forty winters shall besiege thy brow,
And dig deep trenches in thy beauty's field,
Thy youth's proud livery so gazed on now,
Will be a tatter'd weed of small worth held:
Then being asked, where all thy beauty lies,
Where all the treasure of thy lusty days;
To say, within thine own deep sunken eyes,
Were an all-eating shame, and thriftless praise.
How much more praise deserv'd thy beauty's use,
If thou couldst answer 'This fair child of mine
Shall sum my count, and make my old excuse,'
Proving his beauty by succession thine!
 This were to be new made when thou art old,
 And see thy blood warm when thou feel'st it cold.



at first:

tyntd-iafhatawiaoihrdemot lytdws e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e
plia tkldrgd t o idoe ns,smtt h ne etie h,hregtrs nigtike,aoaenns lng

↓ train more

"Tmont thithey" fomesscerliund
Keushey. Thom here
sheulke, anmerenith ol sivh I lalterthend Bleipile shuwyl fil on aseterlome
coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."

↓ train more

Aftair fall unsuch that the hall for Prince Velzonski's that me of
her hearly, and behs to so arwage fiving were to it beloge, pavu say falling misfort
how, and Gogition is so overelical and ofter.

↓ train more

"Why do what that day," replied Natasha, and wishing to himself the fact the
princess, Princess Mary was easier, fed in had oftened him.
Pierre aking his soul came to the packs and drove up his father-in-law women.

PANDARUS:

Alas, I think he shall be come approached and the day
When little strain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:

They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:

Well, your wit is in the care of side and that.

Second Lord:

They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:

Come, sir, I will make did behold your worship.

VIOLA:

I'll drink it.

VIOLA:

Why, Salisbury must find his flesh and thought
That which I am not aps, not a man and in fire,
To show the reining of the raven and the wars
To grace my hand reproach within, and not a fair are hand,
That Caesar and my goodly father's world;
When I was heaven of presence and our fleets,
We spare with hours, but cut thy council I am great,
Murdered and by thy master's ready there
My power to give thee but so much as hell:
Some service in the noble bondman here,
Would show him to her wine.

KING LEAR:

O, if you were a feeble sight, the courtesy of your law,
Your sight and several breath, will wear the gods
With his heads, and my hands are wonder'd at the deeds,
So drop upon your lordship's head, and your opinion
Shall be against your honour.

The Stacks Project: open source algebraic geometry textbook

The screenshot shows the homepage of the Stacks Project. At the top, there is a navigation bar with links: home, about, tags explained, tag lookup, browse, search, bibliography, recent comments, blog, and add slogans. Below the navigation bar, there is a section titled "Browse chapters". This section contains a table with two columns: "Part" and "Chapter". The "Part" column lists "Preliminaries", "1. Introduction", "2. Conventions", "3. Set Theory", "4. Categories", "5. Topology", "6. Sheaves on Spaces", "7. Sites and Sheaves", "8. Stacks", "9. Fields", and "10. Commutative Algebra". The "Chapter" column contains the chapter titles. To the right of the table, there are three buttons: "online", "TeX source", and "view pdf". Below the table, there is a sidebar with two sections: "Parts" and "Statistics". The "Parts" section lists eight numbered items: Preliminaries, Schemes, Topics in Scheme Theory, Algebraic Spaces, Topics in Geometry, Deformation Theory, Algebraic Stacks, and Miscellany. The "Statistics" section provides information about the project's size: 455910 lines of code, 14221 tags (56 inactive tags), and 2366 sections.

Part	Chapter
Preliminaries	1. Introduction
	2. Conventions
	3. Set Theory
	4. Categories
	5. Topology
	6. Sheaves on Spaces
	7. Sites and Sheaves
	8. Stacks
	9. Fields
	10. Commutative Algebra

online [TeX source](#) [view pdf](#)

Parts

- [Preliminaries](#)
- [Schemes](#)
- [Topics in Scheme Theory](#)
- [Algebraic Spaces](#)
- [Topics in Geometry](#)
- [Deformation Theory](#)
- [Algebraic Stacks](#)
- [Miscellany](#)

Statistics

The Stacks project now consists of

- 455910 lines of code
- 14221 tags (56 inactive tags)
- 2366 sections

Latex source

<http://stacks.math.columbia.edu/>

The stacks project is licensed under the [GNU Free Documentation License](#)

For $\bigoplus_{n=1,\dots,m} \mathcal{L}_{m,n} = 0$, hence we can find a closed subset \mathcal{H} in \mathcal{H} and any sets \mathcal{F} on X , U is a closed immersion of S , then $U \rightarrow T$ is a separated algebraic space.

Proof. Proof of (1). It also start we get

$$S = \text{Spec}(R) = U \times_X U \times_X U$$

and the comparicoly in the fibre product covering we have to prove the lemma generated by $\coprod Z \times_U U \rightarrow V$. Consider the maps M along the set of points Sch_{fppf} and $U \rightarrow U$ is the fibre category of S in U in Section, ?? and the fact that any U affine, see Morphisms, Lemma ???. Hence we obtain a scheme S and any open subset $W \subset U$ in $\text{Sh}(G)$ such that $\text{Spec}(R') \rightarrow S$ is smooth or an

$$U = \bigcup U_i \times_{S_i} U_i$$

which has a nonzero morphism we may assume that f_i is of finite presentation over S . We claim that $\mathcal{O}_{X,x}$ is a scheme where $x, x', s'' \in S'$ such that $\mathcal{O}_{X,x'} \rightarrow \mathcal{O}_{X',x'}$ is separated. By Algebra, Lemma ?? we can define a map of complexes $\text{GL}_{S'}(x'/S'')$ and we win. \square

To prove study we see that $\mathcal{F}|_U$ is a covering of \mathcal{X}' , and \mathcal{T}_i is an object of $\mathcal{F}_{X/S}$ for $i > 0$ and \mathcal{F}_p exists and let \mathcal{F}_i be a presheaf of \mathcal{O}_X -modules on \mathcal{C} as a \mathcal{F} -module. In particular $\mathcal{F} = U/\mathcal{F}$ we have to show that

$$\widetilde{M}^\bullet = \mathcal{I}^\bullet \otimes_{\text{Spec}(k)} \mathcal{O}_{S,s} - i_X^{-1} \mathcal{F}$$

is a unique morphism of algebraic stacks. Note that

$$\text{Arrows} = (\text{Sch}/S)_{fppf}^{\text{opp}}, (\text{Sch}/S)_{fppf}$$

and

$$V = \Gamma(S, \mathcal{O}) \rightarrow (U, \text{Spec}(A))$$

is an open subset of X . Thus U is affine. This is a continuous map of X is the inverse, the groupoid scheme S .

Proof. See discussion of sheaves of sets. \square

The result for prove any open covering follows from the less of Example ???. It may replace S by $X_{\text{spaces},\text{étale}}$ which gives an open subspace of X and T equal to S_{Zar} , see Descent, Lemma ???. Namely, by Lemma ?? we see that R is geometrically regular over S .

Lemma 0.1. Assume (3) and (3) by the construction in the description.

Suppose $X = \lim |X|$ (by the formal open covering X and a single map $\underline{\text{Proj}}_X(\mathcal{A}) = \text{Spec}(B)$ over U compatible with the complex

$$\text{Set}(\mathcal{A}) = \Gamma(X, \mathcal{O}_{X,\mathcal{O}_X}).$$

When in this case of to show that $\mathcal{Q} \rightarrow \mathcal{C}_{Z/X}$ is stable under the following result in the second conditions of (1), and (3). This finishes the proof. By Definition ?? (without element is when the closed subschemes are catenary. If T is surjective we may assume that T is connected with residue fields of S . Moreover there exists a closed subspace $Z \subset X$ of X where U in X' is proper (some defining as a closed subset of the uniqueness it suffices to check the fact that the following theorem

(1) f is locally of finite type. Since $S = \text{Spec}(R)$ and $Y = \text{Spec}(R)$.

Proof. This is form all sheaves of sheaves on X . But given a scheme U and a surjective étale morphism $U \rightarrow X$. Let $U \cap U = \coprod_{i=1,\dots,n} U_i$ be the scheme X over S at the schemes $X_i \rightarrow X$ and $U = \lim_i X_i$. \square

The following lemma surjective restrocomposes of this implies that $\mathcal{F}_{x_0} = \mathcal{F}_{x_0} = \mathcal{F}_{x,\dots,x_0}$.

Lemma 0.2. Let X be a locally Noetherian scheme over S , $E = \mathcal{F}_{X/S}$. Set $\mathcal{I} = \mathcal{J}_1 \subset \mathcal{I}'_n$. Since $\mathcal{I}^n \subset \mathcal{I}^n$ are nonzero over $i_0 \leq p$ is a subset of $\mathcal{J}_{n,0} \circ \mathcal{A}_2$ works.

Lemma 0.3. In Situation ???. Hence we may assume $q' = 0$.

Proof. We will use the property we see that p is the next functor (??). On the other hand, by Lemma ?? we see that

$$D(\mathcal{O}_{X'}) = \mathcal{O}_X(D)$$

where K is an F -algebra where δ_{n+1} is a scheme over S . \square

Proof. Omitted. □

Lemma 0.1. Let \mathcal{C} be a set of the construction.

Let \mathcal{C} be a gerber covering. Let \mathcal{F} be a quasi-coherent sheaves of \mathcal{O} -modules. We have to show that

$$\mathcal{O}_{\mathcal{O}_X} = \mathcal{O}_X(\mathcal{L})$$

Proof. This is an algebraic space with the composition of sheaves \mathcal{F} on $X_{\text{étale}}$ we have

$$\mathcal{O}_X(\mathcal{F}) = \{\text{morph}_1 \times_{\mathcal{O}_X} (\mathcal{G}, \mathcal{F})\}$$

where \mathcal{G} defines an isomorphism $\mathcal{F} \rightarrow \mathcal{F}$ of \mathcal{O} -modules. □

Lemma 0.2. This is an integer \mathcal{Z} is injective.

Proof. See Spaces, Lemma ??.

Lemma 0.3. Let S be a scheme. Let X be a scheme and X is an affine open covering. Let $\mathcal{U} \subset \mathcal{X}$ be a canonical and locally of finite type. Let X be a scheme. Let X be a scheme which is equal to the formal complex.

The following to the construction of the lemma follows.

Let X be a scheme. Let X be a scheme covering. Let

$$b : X \rightarrow Y' \rightarrow Y \rightarrow Y \rightarrow Y' \times_X Y \rightarrow X.$$

be a morphism of algebraic spaces over S and Y .

Proof. Let X be a nonzero scheme of X . Let X be an algebraic space. Let \mathcal{F} be a quasi-coherent sheaf of \mathcal{O}_X -modules. The following are equivalent

- (1) \mathcal{F} is an algebraic space over S .
- (2) If X is an affine open covering.

Consider a common structure on X and X the functor $\mathcal{O}_X(U)$ which is locally of finite type. □

This since $\mathcal{F} \in \mathcal{F}$ and $x \in \mathcal{G}$ the diagram

$$\begin{array}{ccccc}
 S & \xrightarrow{\quad} & & & \\
 \downarrow & & & & \\
 \xi & \longrightarrow & \mathcal{O}_{X'} & \xrightarrow{\quad} & \\
 \text{gor}_s & & \uparrow & \searrow & \\
 & & =\alpha' \longrightarrow & & \\
 & & \downarrow & & \\
 & & =\alpha' \longrightarrow & & \\
 & & \downarrow & & \\
 \text{Spec}(K_\psi) & & \text{Mor}_{\text{Sets}} & & d(\mathcal{O}_{X/k}, \mathcal{G}) \\
 & & \downarrow & & \\
 & & X & \xrightarrow{\quad} & \\
 & & \downarrow & & \\
 & & & & d(\mathcal{O}_{X/k}, \mathcal{G})
 \end{array}$$

is a limit. Then \mathcal{G} is a finite type and assume S is a flat and \mathcal{F} and \mathcal{G} is a finite type f_* . This is of finite type diagrams, and

- the composition of \mathcal{G} is a regular sequence,
- $\mathcal{O}_{X'}$ is a sheaf of rings.

Proof. We have see that $X = \text{Spec}(R)$ and \mathcal{F} is a finite type representable by algebraic space. The property \mathcal{F} is a finite morphism of algebraic stacks. Then the cohomology of X is an open neighbourhood of U . □

Proof. This is clear that \mathcal{G} is a finite presentation, see Lemmas ??.

A reduced above we conclude that U is an open covering of \mathcal{C} . The functor \mathcal{F} is a “field”

$$\mathcal{O}_{X,x} \longrightarrow \mathcal{F}_{\bar{x}} \xrightarrow{-1} (\mathcal{O}_{X_{\text{étale}}}) \longrightarrow \mathcal{O}_{X_{\bar{\ell}}}^{-1} \mathcal{O}_{X_{\lambda}}(\mathcal{O}_{X_{\eta}}^{\bar{v}})$$

is an isomorphism of covering of \mathcal{O}_{X_i} . If \mathcal{F} is the unique element of \mathcal{F} such that X is an isomorphism.

The property \mathcal{F} is a disjoint union of Proposition ?? and we can filtered set of presentations of a scheme \mathcal{O}_X -algebra with \mathcal{F} are opens of finite type over S .

If \mathcal{F} is a scheme theoretic image points. □

If \mathcal{F} is a finite direct sum \mathcal{O}_{X_k} is a closed immersion, see Lemma ???. This is a sequence of \mathcal{F} is a similar morphism.



This repository Search

Explore Gist Blog Help



karpathy

+ · ⌂ ⚙ ⌂



torvalds / linux

Watch 3,711

Star 23,054

Fork 9,141

Linux kernel source tree

520,037 commits

1 branch

420 releases

5,039 contributors



branch: master + linux / +



Merge branch 'drm-fixes' of git://people.freedesktop.org/~airlied/linux ...

torvalds authored 9 hours ago

latest commit 4b1786927d ↗

Documentation

Merge git://git.kernel.org/pub/scm/linux/kernel/git/nab/target-pending

6 days ago

arch

Merge branch 'x86-urgent-for-linus' of git://git.kernel.org/pub/scm/l...

a day ago

block

block: discard bdi_unregister() in favour of bdi_destroy()

9 days ago

crypto

Merge git://git.kernel.org/pub/scm/linux/kernel/git/herbert/crypto-2.6

10 days ago

drivers

Merge branch 'drm-fixes' of git://people.freedesktop.org/~airlied/linux

9 hours ago

firmware

firmware/ihex2fw.c: restore missing default in switch statement

2 months ago

fs

vfs: read file_handle only once in handle_to_path

4 days ago

include

Merge branch 'perl-urgent-for-linus' of git://git.kernel.org/pub/scm/...

a day ago

init

init: fix regression by supporting devices with major:minor:offset fo...

a month ago

iio

iio: sensors: Merge 'iio-urgent-for-linus' and 'iio-urgent-for-linus' branches

a month ago



Code



74 Pull requests



Pulse



Graphs

HTTPS clone URL

<https://github.com/torvalds/linux> ↗

You can clone with **HTTPS**,
SSH, or **Subversion**. ⓘ

Clone in Desktop

Download ZIP

```
static void do_command(struct seq_file *m, void *v)
{
    int column = 32 << (cmd[2] & 0x80);
    if (state)
        cmd = (int)(int_state ^ (in_8(&ch->ch_flags) & Cmd) ? 2 : 1);
    else
        seq = 1;
    for (i = 0; i < 16; i++) {
        if (k & (1 << 1))
            pipe = (in_use & UMXTHREAD_UNCCA) +
                ((count & 0x00000000fffffff8) & 0x0000000f) << 8;
        if (count == 0)
            sub(pid, ppc_md.kexec_handle, 0x20000000);
        pipe_set_bytes(i, 0);
    }
    /* Free our user pages pointer to place camera if all dash */
    subsystem_info = &of_changes[PAGE_SIZE];
    rek_controls(offset, idx, &soffset);
    /* Now we want to deliberately put it to device */
    control_check_polarity(&context, val, 0);
    for (i = 0; i < COUNTER; i++)
        seq_puts(s, "policy ");
}
```

Generated C code

```
/*
 * Copyright (c) 2006-2010, Intel Mobile Communications. All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 as published by
 * the Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 *
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#include <linux/kexec.h>
#include <linux/errno.h>
#include <linux/io.h>
#include <linux/platform_device.h>
#include <linux/multi.h>
#include <linux/ckevent.h>

#include <asm/io.h>
#include <asm/prom.h>
#include <asm/e820.h>
#include <asm/system_info.h>
#include <asm/seteew.h>
#include <asm/pgproto.h>
```

```

#include <asm/io.h>
#include <asm/prom.h>
#include <asm/e820.h>
#include <asm/system_info.h>
#include <asm/seteew.h>
#include <asm/pgproto.h>

#define REG_PG      vesa_slot_addr_pack
#define PFM_NOCOMP  AFSR(0, load)
#define STACK_DDR(type)      (func)

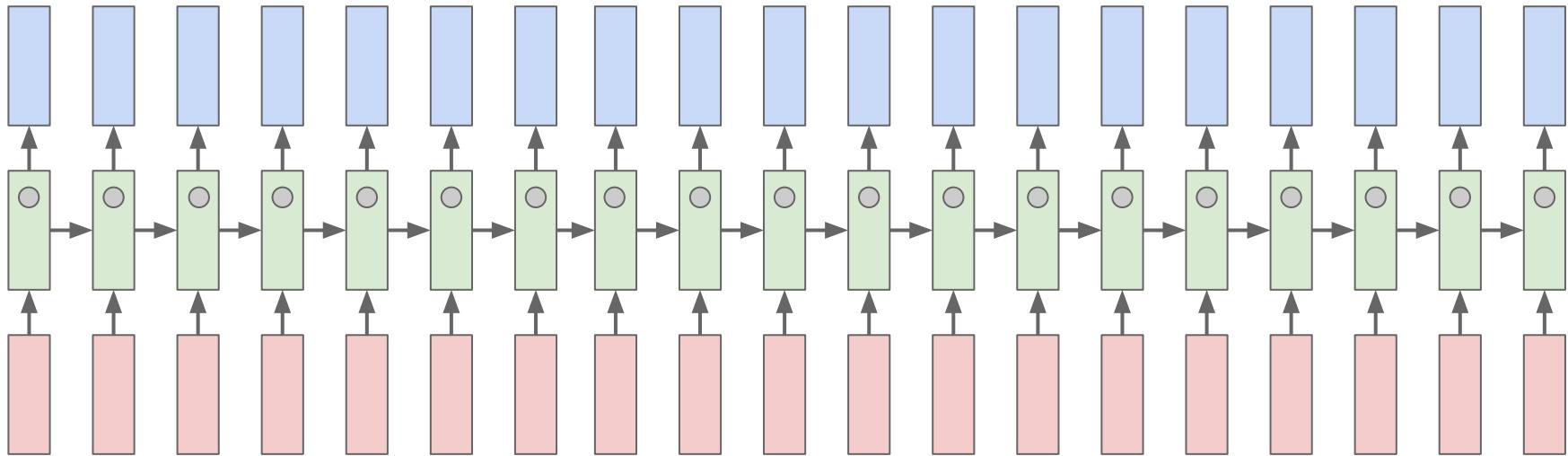
#define SWAP_ALLOCATE(nr)      (e)
#define emulate_sigs()  arch_get_unaligned_child()
#define access_rw(TST)  asm volatile("movd %esp, %0, %3" : : "r" (0)); \
    if (__type & DO_READ)

static void stat_PC_SEC __read_mostly offsetof(struct seq_argsqueue, \
    pC>[1]);

static void
os_prefix(unsigned long sys)
{
#endif CONFIG_PREEMPT
    PUT_PARAM_RAID(2, sel) = get_state_state();
    set_pid_sum((unsigned long)state, current_state_str(),
                (unsigned long)-1->lr_full, low;
}

```

Searching for interpretable cells



Karpathy, Johnson, and Fei-Fei: Visualizing and Understanding Recurrent Networks, ICLR Workshop 2016

Searching for interpretable cells

```
/* Unpack a filter field's string representation from user-space
 * buffer. */
char *audit_unpack_string(void **bufp, size_t *remain, size_t len)
{
    char *str;
    if (!*bufp || (len == 0) || (len > *remain))
        return ERR_PTR(-EINVAL);
    /* of the currently implemented string fields, PATH_MAX
     * defines the longest valid length.
    */
```

Searching for interpretable cells

"You mean to imply that I have nothing to eat out of.... On the contrary, I can supply you with everything even if you want to give dinner parties," warmly replied Chichagov, who tried by every word he spoke to prove his own rectitude and therefore imagined Kutuzov to be animated by the same desire.

Kutuzov, shrugging his shoulders, replied with his subtle penetrating smile: "I meant merely to say what I said."

quote detection cell

Karpathy, Johnson, and Fei-Fei: Visualizing and Understanding Recurrent Networks, ICLR Workshop 2016
Figures copyright Karpathy, Johnson, and Fei-Fei, 2015; reproduced with permission

Searching for interpretable cells

Cell sensitive to position in line:

The sole importance of the crossing of the Berezina lies in the fact that it plainly and indubitably proved the fallacy of all the plans for cutting off the enemy's retreat and the soundness of the only possible line of action--the one Kutuzov and the general mass of the army demanded--namely, simply to follow the enemy up. The French crowd fled at a continually increasing speed and all its energy was directed to reaching its goal. It fled like a wounded animal and it was impossible to block its path. This was shown not so much by the arrangements it made for crossing as by what took place at the bridges. When the bridges broke down, unarmed soldiers, people from Moscow and women with children who were with the French transport, all--carried on by vis inertiae--pressed forward into boats and into the ice-covered water and did not, surrender.

line length tracking cell

Karpathy, Johnson, and Fei-Fei: Visualizing and Understanding Recurrent Networks, ICLR Workshop 2016

Figures copyright Karpathy, Johnson, and Fei-Fei, 2015; reproduced with permission

Searching for interpretable cells

```
static int __dequeue_signal(struct sigpending *pending, sigset_t *mask,
    siginfo_t *info)
{
    int sig = next_signal(pending, mask);
    if (sig) {
        if (current->notifier) {
            if (sigismember(current->notifier_mask, sig)) {
                if (! (current->notifier)(current->notifier_data)) {
                    clear_thread_flag(TIF_SIGPENDING);
                    return 0;
                }
            }
            collect_signal(sig, pending, info);
        }
    }
    return sig;
}
```

if statement cell

Searching for interpretable cells

Cell that turns on inside comments and quotes:

```
/* Duplicate LSM field information. The lsm_rule is opaque, so
 * re-initialized. */
static inline int audit_dupe_lsm_field(struct audit_field *df,
                                       struct audit_field *sf)
{
    int ret = 0;
    char *lsm_str;
    /* our own copy of lsm_str */
    lsm_str = kstrdup(sf->lsm_str, GFP_KERNEL);
    if (unlikely(!lsm_str))
        return -ENOMEM;
    df->lsm_str = lsm_str;
    /* our own (refreshed) copy of lsm_rule */
    ret = security_audit_rule_init(df->type, df->op, df->lsm_str,
                                   (void *) &df->lsm_rule);
    /* Keep currently invalid fields around in case they
     * become valid after a policy reload. */
    if (ret == -EINVAL) {
        pr_warn("audit rule for LSM \\'%s\\' is invalid\n",
               df->lsm_str);
        ret = 0;
    }
    return ret;
}
```

quote/comment cell

Searching for interpretable cells

```
#ifdef CONFIG_AUDITSYSCALL
static inline int audit_match_class_bits(int class, u32 *mask)
{
    int i;
    if (classes[class]) {
        for (i = 0; i < AUDIT_BITMASK_SIZE; i++)
            if (mask[i] & classes[class][i])
                return 0;
    }
    return 1;
}
```

code depth cell

Image Captioning

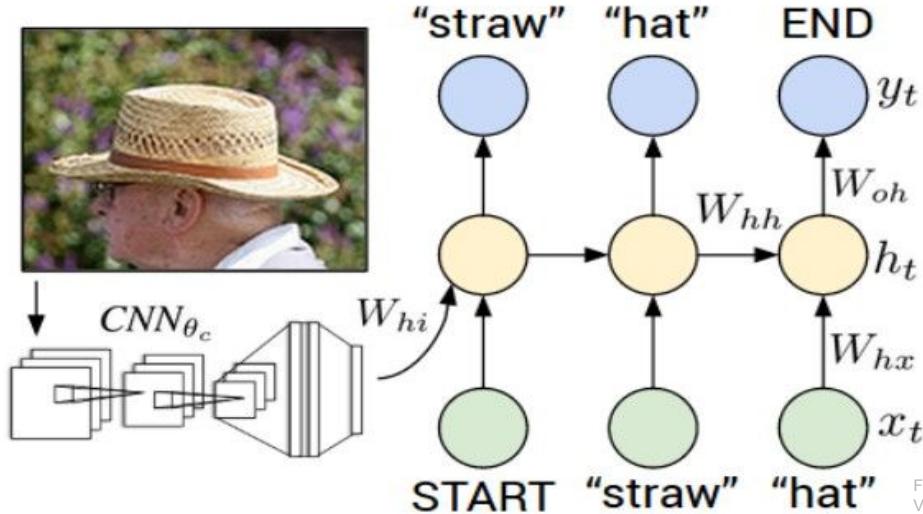


Figure from Karpathy et al, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015; figure copyright IEEE, 2015.
Reproduced for educational purposes.

Explain Images with Multimodal Recurrent Neural Networks, Mao et al.

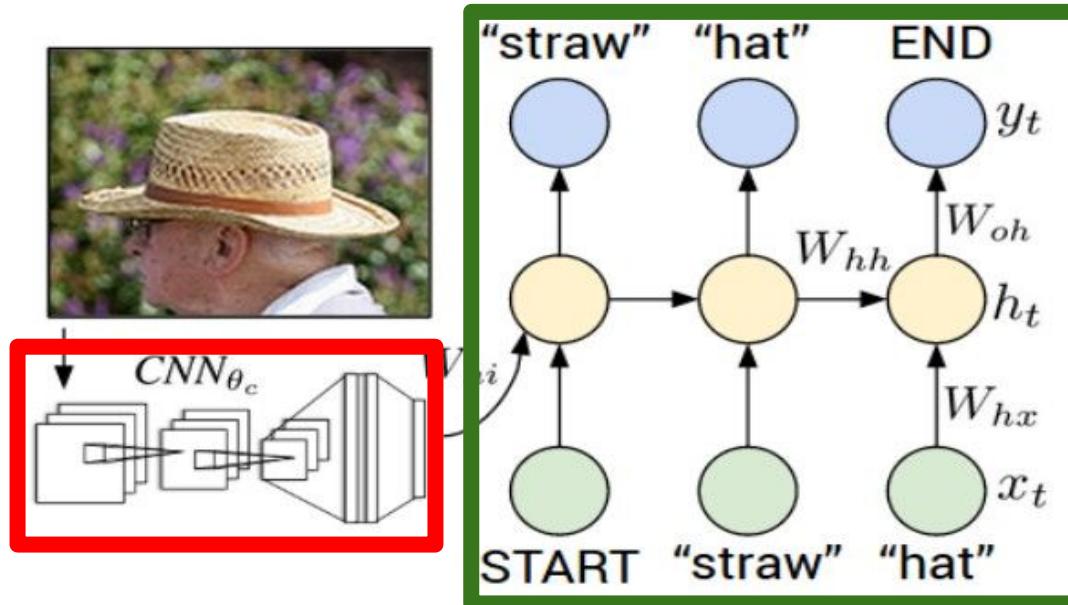
Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei

Show and Tell: A Neural Image Caption Generator, Vinyals et al.

Long-term Recurrent Convolutional Networks for Visual Recognition and Description, Donahue et al.

Learning a Recurrent Visual Representation for Image Caption Generation, Chen and Zitnick

Recurrent Neural Network



Convolutional Neural Network

test image



[This image is CC0 public domain](#)

image



test image



conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

FC-1000

softmax

image



test image



conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

FC-1000

softmax

X

image



conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

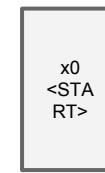
maxpool

FC-4096

FC-4096



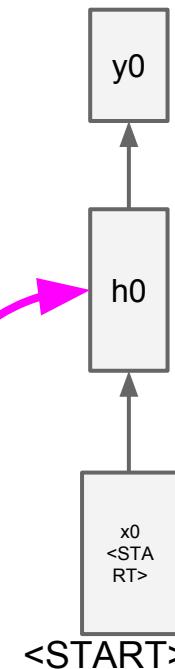
test image



<START>



test image



before:

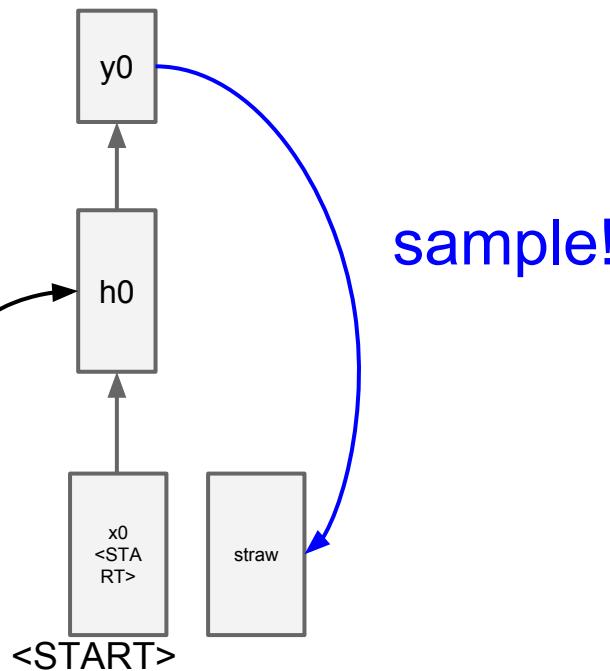
$$h = \tanh(W_{xh} * x + W_{hh} * h)$$

now:

$$h = \tanh(W_{xh} * x + W_{hh} * h + W_{ih} * v)$$



test image



image



test image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

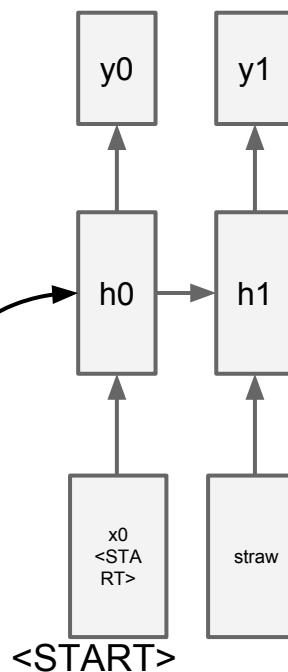
conv-512

conv-512

maxpool

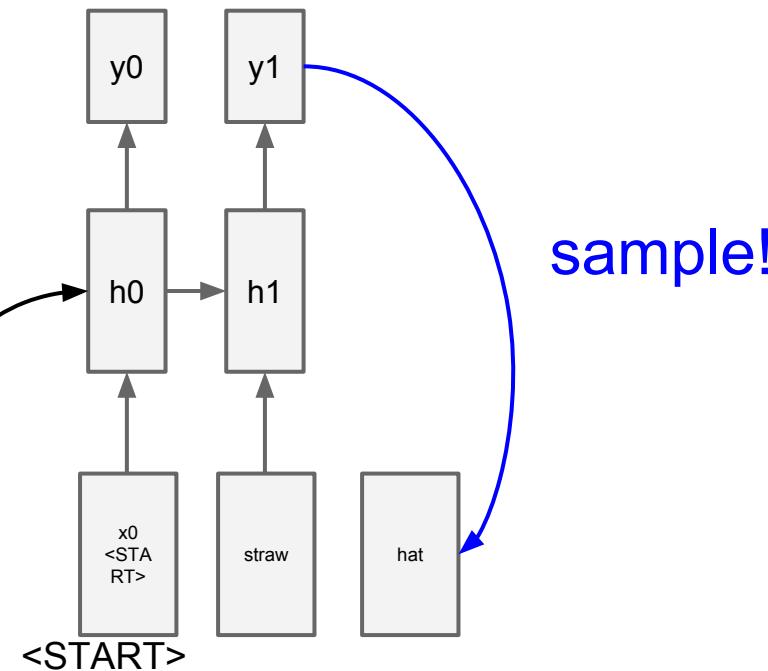
FC-4096

FC-4096





test image



image



test image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

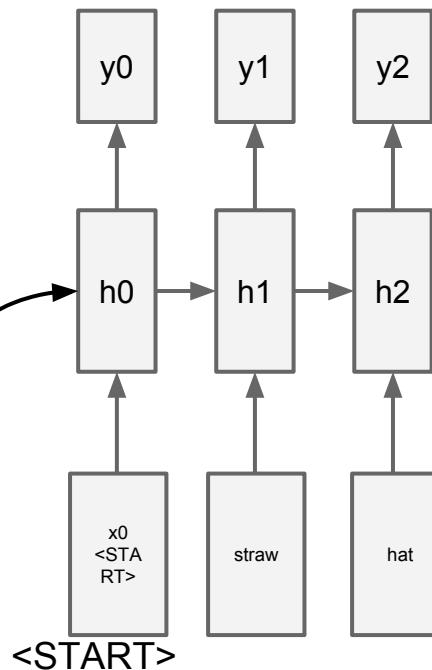
conv-512

conv-512

maxpool

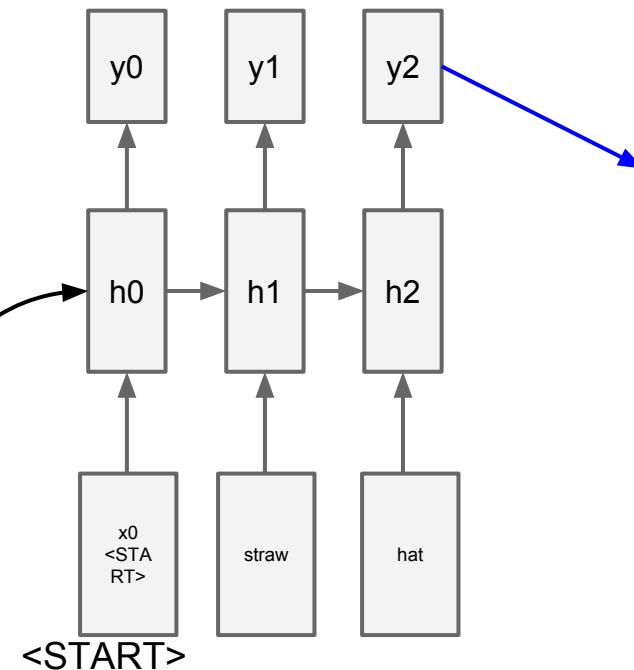
FC-4096

FC-4096





test image



sample
<END> token
=> finish.

Image Captioning: Example Results



A cat sitting on a suitcase on the floor



A cat is sitting on a tree branch



A dog is running in the grass with a frisbee



A white teddy bear sitting in the grass



Two people walking on the beach with surfboards



A tennis player in action on the court



Two giraffes standing in a grassy field



A man riding a dirt bike on a dirt track

Image Captioning: Failure Cases

Captions generated using [neuraltalk2](#)
All images are CC0 Public domain: [fur coat](#), [handstand](#), [spider web](#), [baseball](#)



A woman is holding a cat in her hand



A person holding a computer mouse on a desk



A woman standing on a beach holding a surfboard



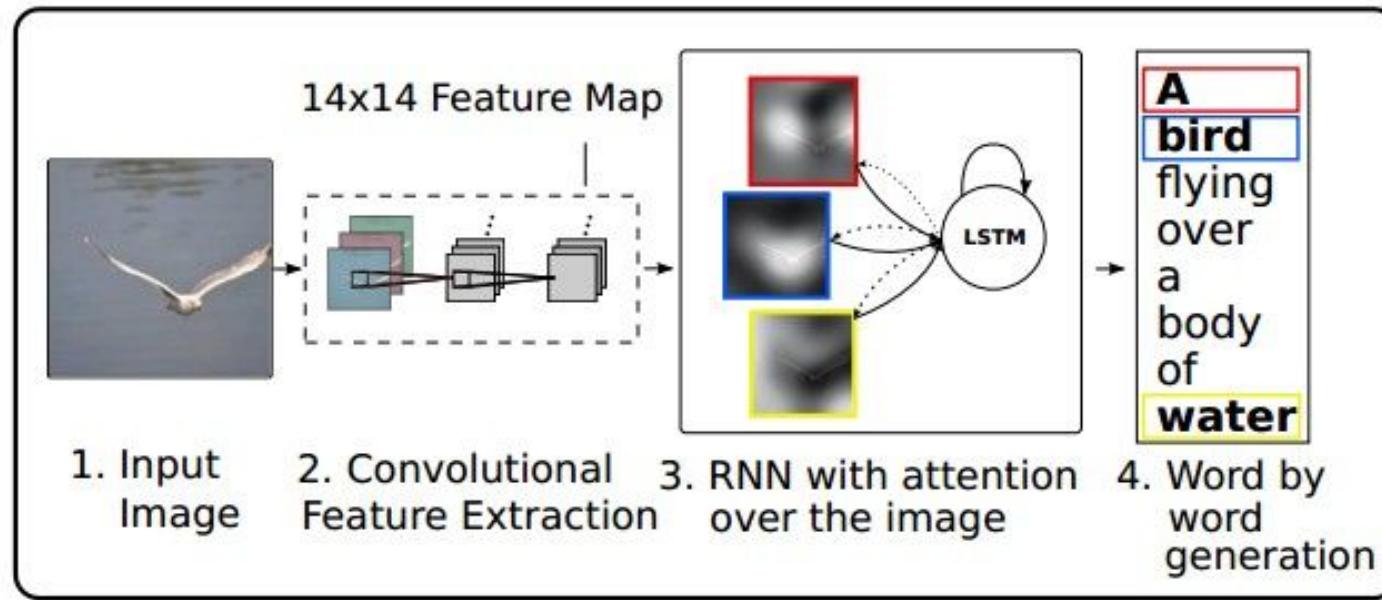
A bird is perched on a tree branch



A man in a baseball uniform throwing a ball

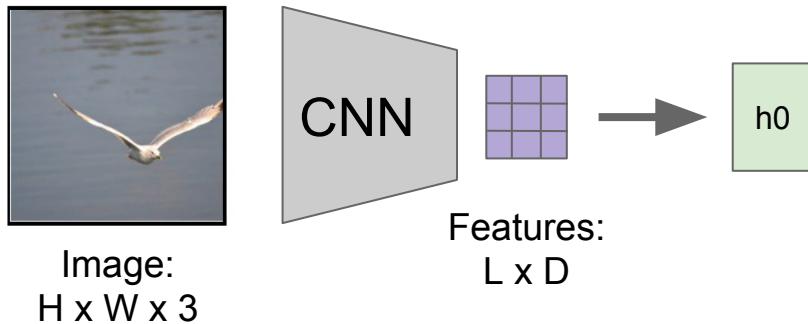
Image Captioning with Attention

RNN focuses its attention at a different spatial location when generating each word



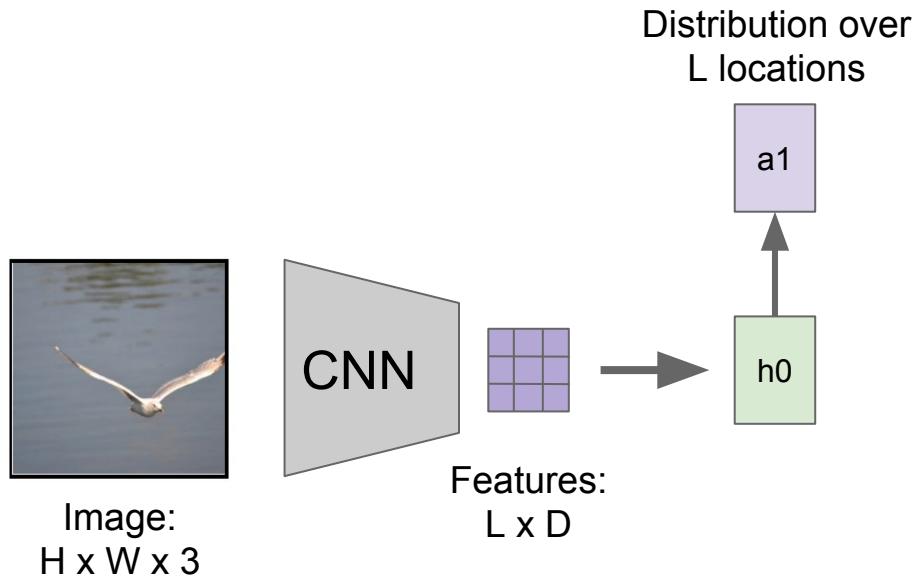
Xu et al, "Show, Attend, and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015
Figure copyright Kelvin Xu, Jimmy Lei Ba, Jamie Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio, 2015. Reproduced with permission.

Image Captioning with Attention



Xu et al, "Show, Attend and Tell: Neural
Image Caption Generation with Visual
Attention", ICML 2015

Image Captioning with Attention

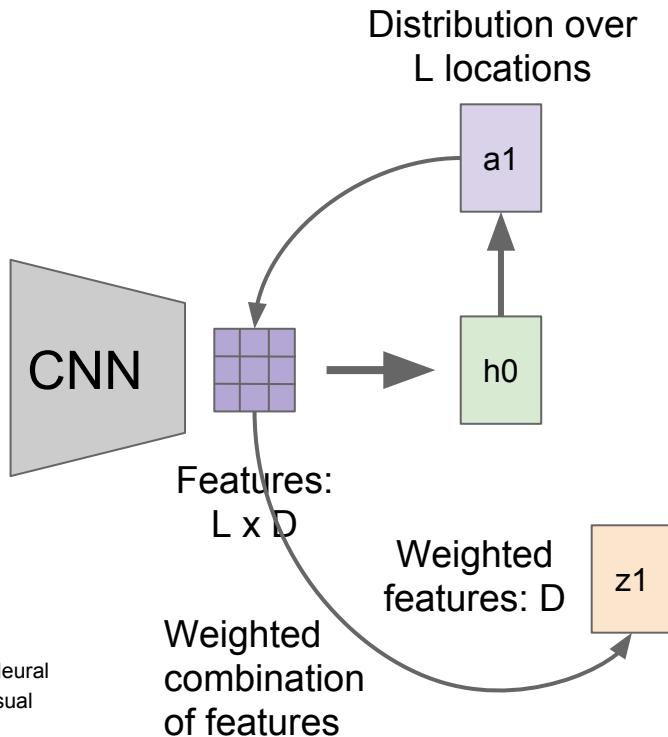


Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning with Attention



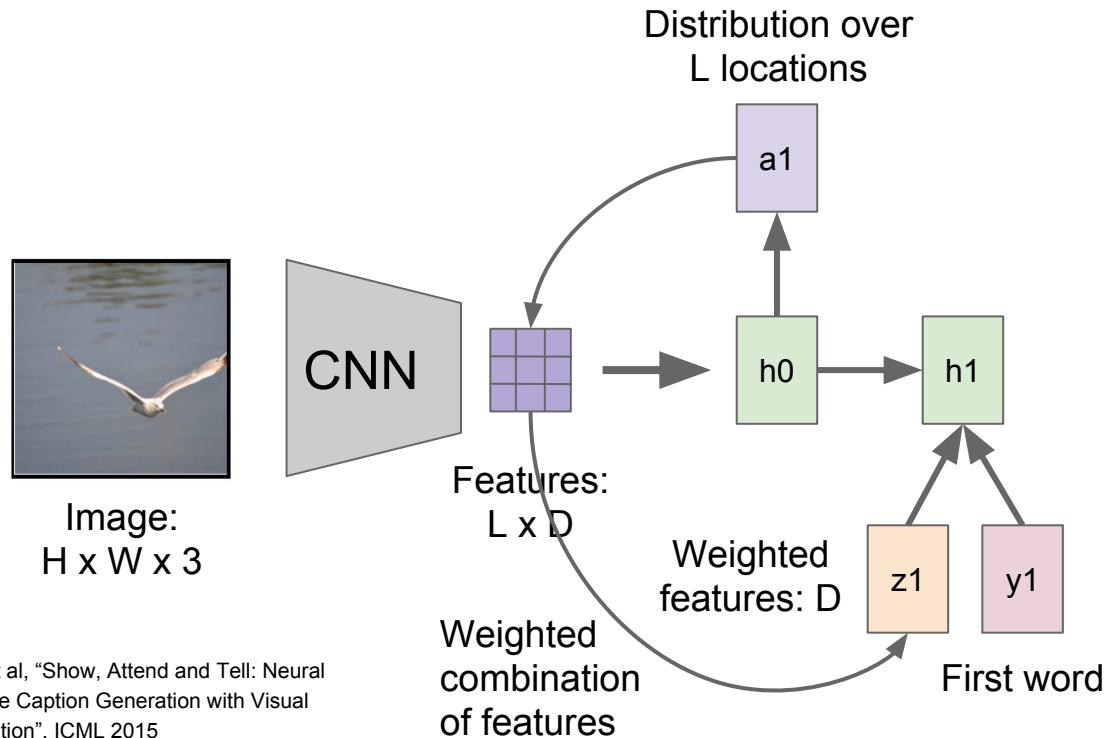
Image:
 $H \times W \times 3$



$$z = \sum_{i=1}^L p_i v_i$$

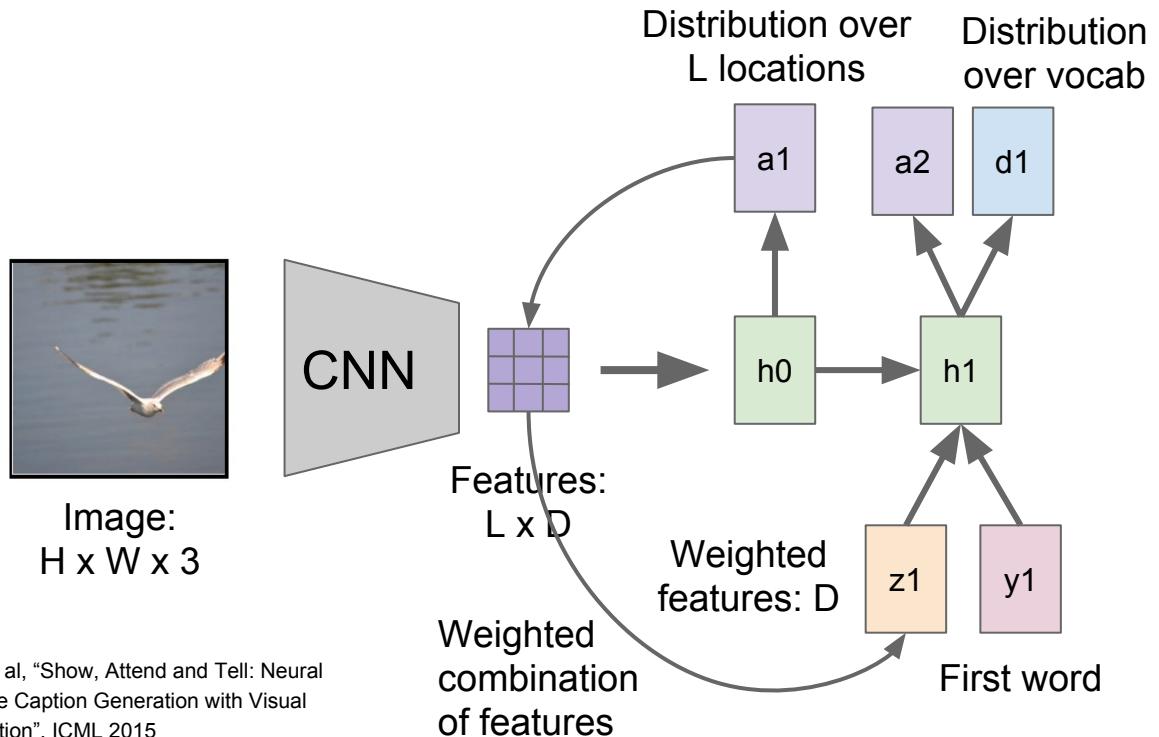
Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning with Attention



Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning with Attention



Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning with Attention

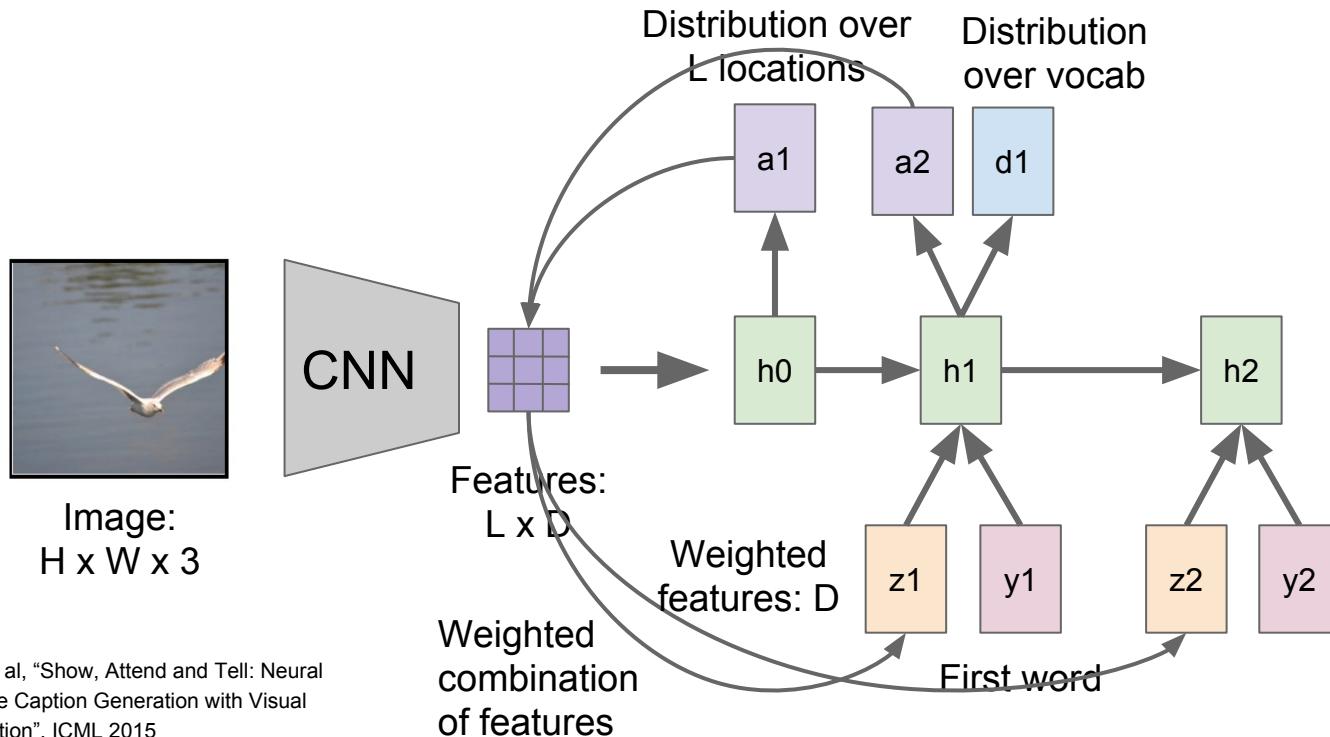
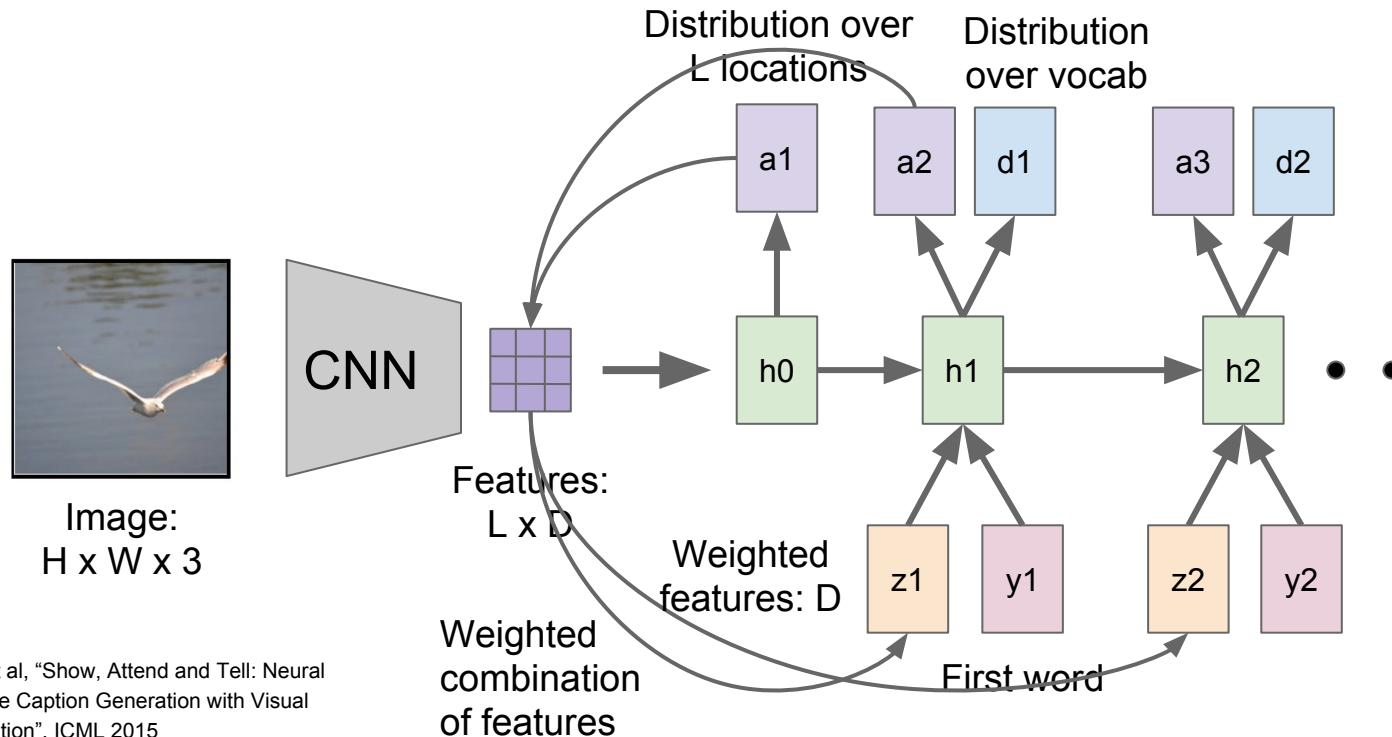
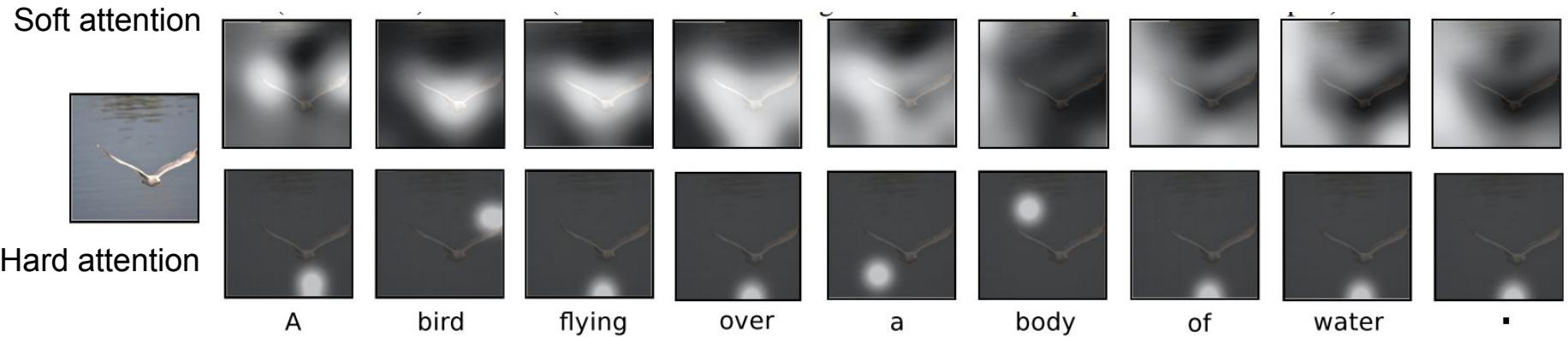


Image Captioning with Attention



Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning with Attention



Xu et al, "Show, Attend, and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Figure copyright Kelvin Xu, Jimmy Lei Ba, Jamie Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio, 2015. Reproduced with permission.

Image Captioning with Attention



A woman is throwing a frisbee in a park.



A dog is standing on a hardwood floor.



A stop sign is on a road with a mountain in the background.



A little girl sitting on a bed with a teddy bear.



A group of people sitting on a boat in the water.



A giraffe standing in a forest with trees in the background.

Xu et al, “Show, Attend, and Tell: Neural Image Caption Generation with Visual Attention”, ICML 2015

Figure copyright Kelvin Xu, Jimmy Lei Ba, Jamie Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio, 2015. Reproduced with permission.

Visual Question Answering



Q: What endangered animal is featured on the truck?

- A: A bald eagle.
- A: A sparrow.
- A: A humming bird.
- A: A raven.



Q: Where will the driver go if turning right?

- A: Onto 24 1/4 Rd.
- A: Onto 25 1/4 Rd.
- A: Onto 23 1/4 Rd.
- A: Onto Main Street.



Q: When was the picture taken?

- A: During a wedding.
- A: During a bar mitzvah.
- A: During a funeral.
- A: During a Sunday church service

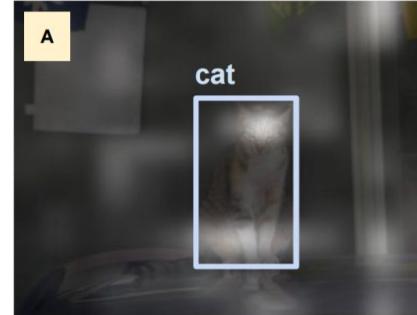
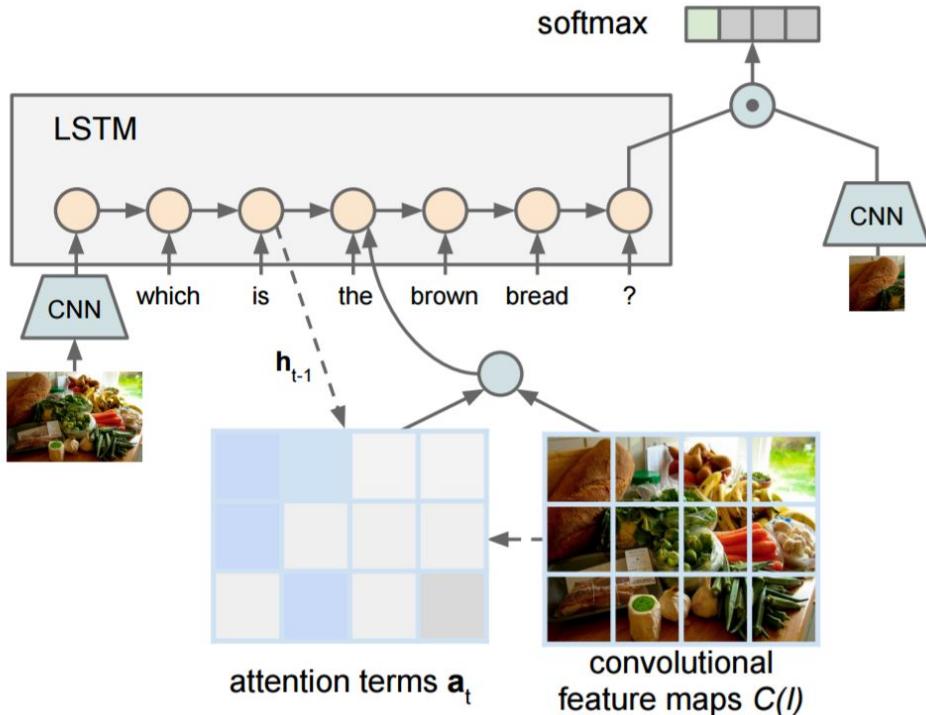


Q: Who is under the umbrella?

- A: Two women.
- A: A child.
- A: An old man.
- A: A husband and a wife.

Agrawal et al, "VQA: Visual Question Answering", ICCV 2015
Zhu et al, "Visual 7W: Grounded Question Answering in Images", CVPR 2016
Figure from Zhu et al, copyright IEEE 2016. Reproduced for educational purposes.

Visual Question Answering: RNNs with Attention



What kind of animal is in the photo?

A **cat**.



Why is the person holding a knife?

To cut the **cake** with.

Zhu et al, "Visual 7W: Grounded Question Answering in Images", CVPR 2016
Figures from Zhu et al, copyright IEEE 2016. Reproduced for educational purposes.

Multilayer RNNs

$$h_t^l = \tanh W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$$h \in \mathbb{R}^n \quad W^l [n \times 2n]$$

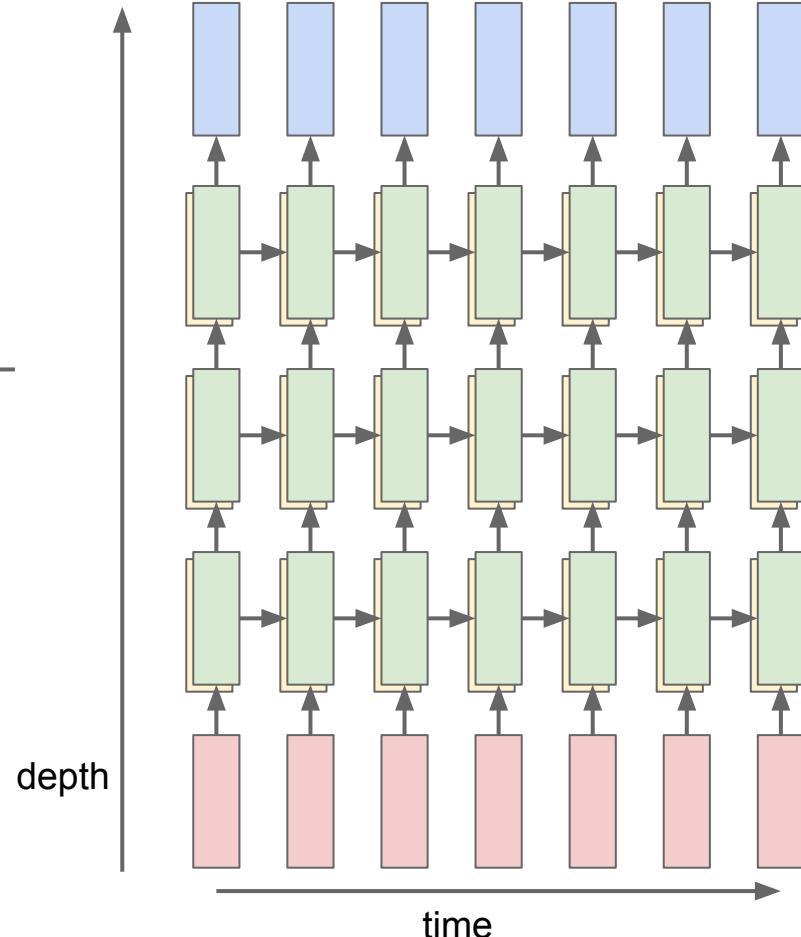
LSTM:

$$W^l [4n \times 2n]$$

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

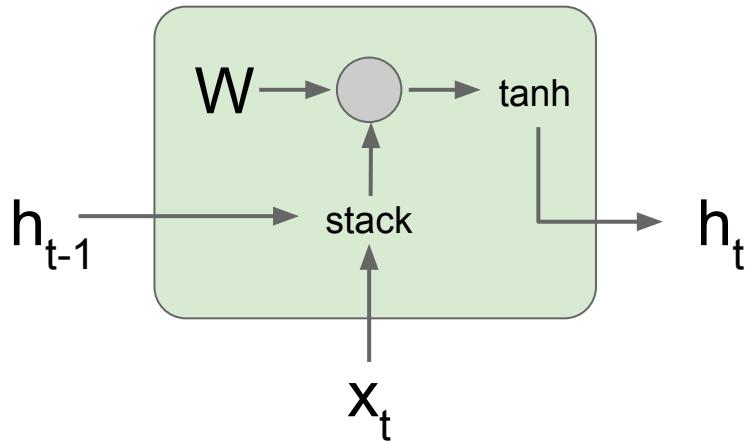
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

$$h_t^l = o \odot \tanh(c_t^l)$$



Vanilla RNN Gradient Flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

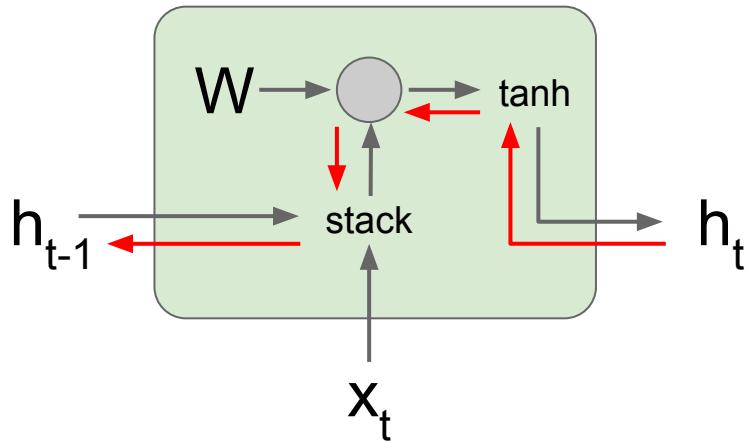


$$\begin{aligned} h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\ &= \tanh \left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \\ &= \tanh \left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \end{aligned}$$

Vanilla RNN Gradient Flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

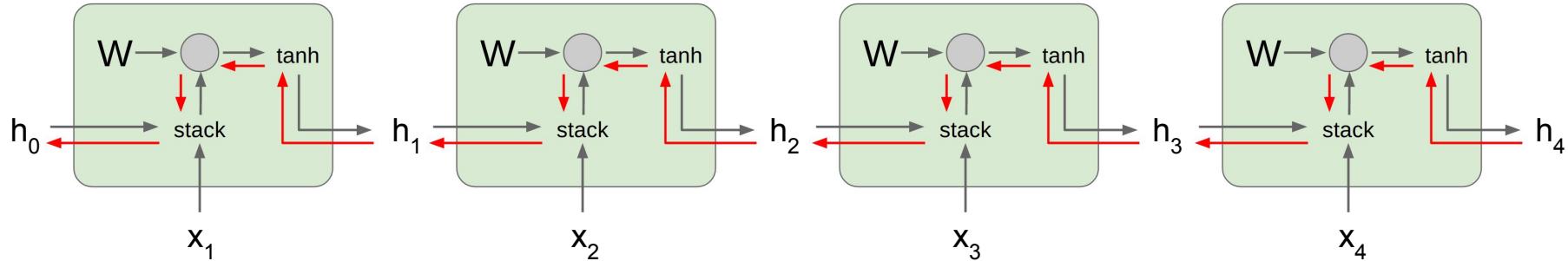
Backpropagation from h_t
to h_{t-1} multiplies by W
(actually W_{hh}^T)



$$\begin{aligned} h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\ &= \tanh \left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \\ &= \tanh \left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \end{aligned}$$

Vanilla RNN Gradient Flow

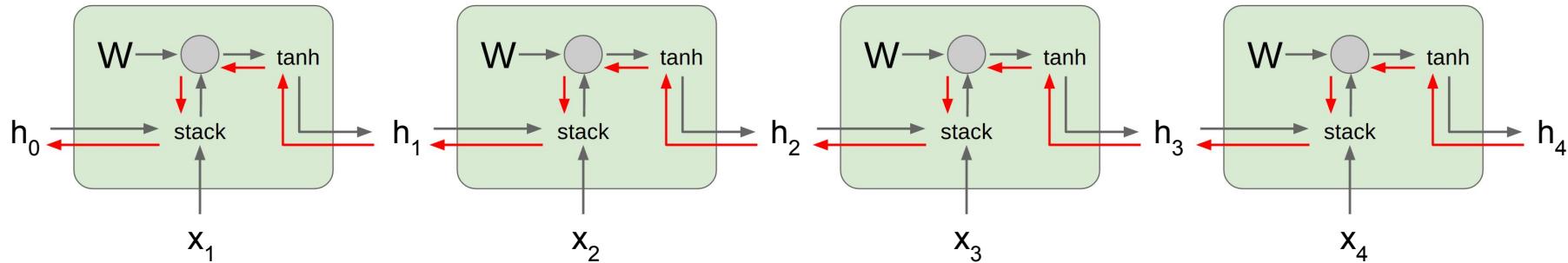
Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



Computing gradient
of h_0 involves many
factors of W
(and repeated tanh)

Vanilla RNN Gradient Flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



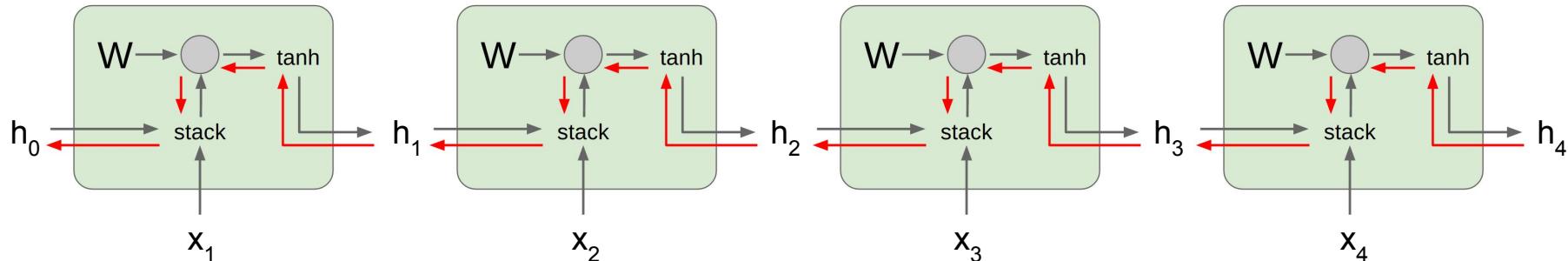
Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value > 1 :
Exploding gradients

Largest singular value < 1 :
Vanishing gradients

Vanilla RNN Gradient Flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value > 1 :
Exploding gradients

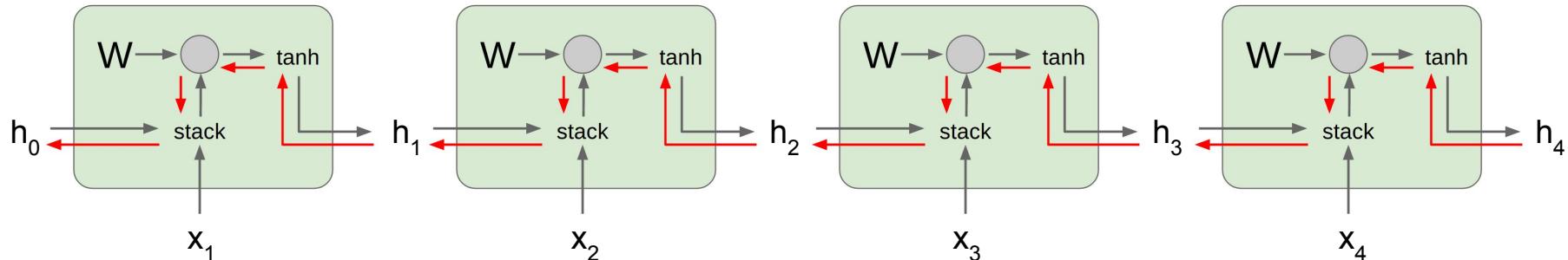
Largest singular value < 1 :
Vanishing gradients

Gradient clipping: Scale gradient if its norm is too big

```
grad_norm = np.sum(grad * grad)
if grad_norm > threshold:
    grad *= (threshold / grad_norm)
```

Vanilla RNN Gradient Flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value > 1 :
Exploding gradients

Largest singular value < 1 :
Vanishing gradients

→ Change RNN architecture

Long Short Term Memory (LSTM)

Vanilla RNN

$$h_t = \tanh \left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)$$

LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

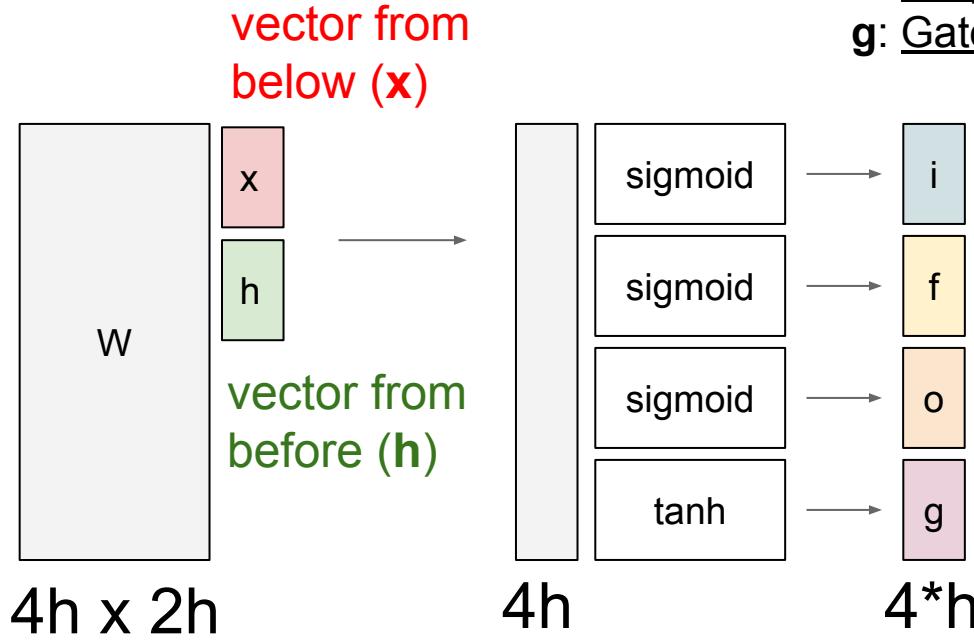
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Hochreiter and Schmidhuber, "Long Short Term Memory", Neural Computation
1997

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



- i: Input gate, whether to write to cell
- f: Forget gate, Whether to erase cell
- o: Output gate, How much to reveal cell
- g: Gate gate (?), How much to write to cell

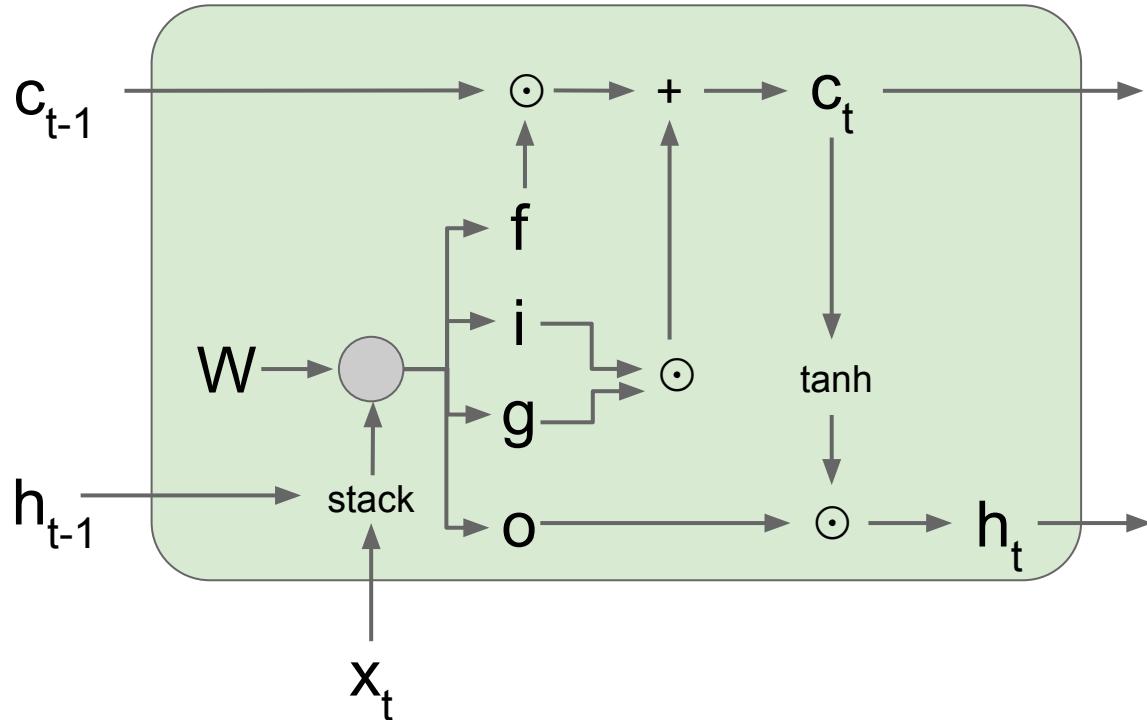
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



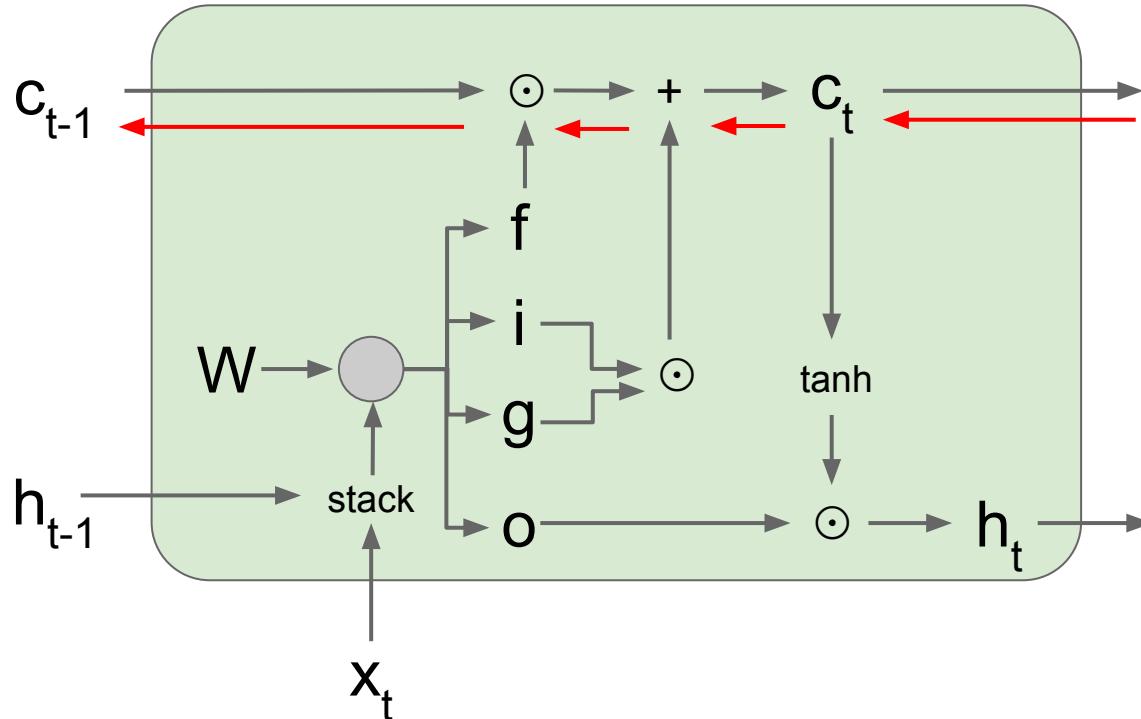
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Long Short Term Memory (LSTM): Gradient Flow

[Hochreiter et al., 1997]



Backpropagation from c_t to c_{t-1} only elementwise multiplication by f , no matrix multiply by W

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

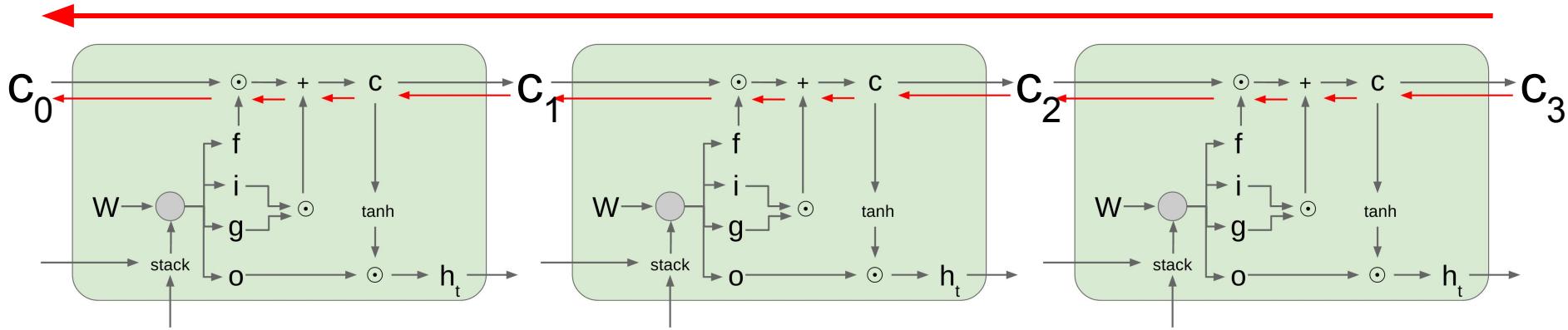
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Long Short Term Memory (LSTM): Gradient Flow

[Hochreiter et al., 1997]

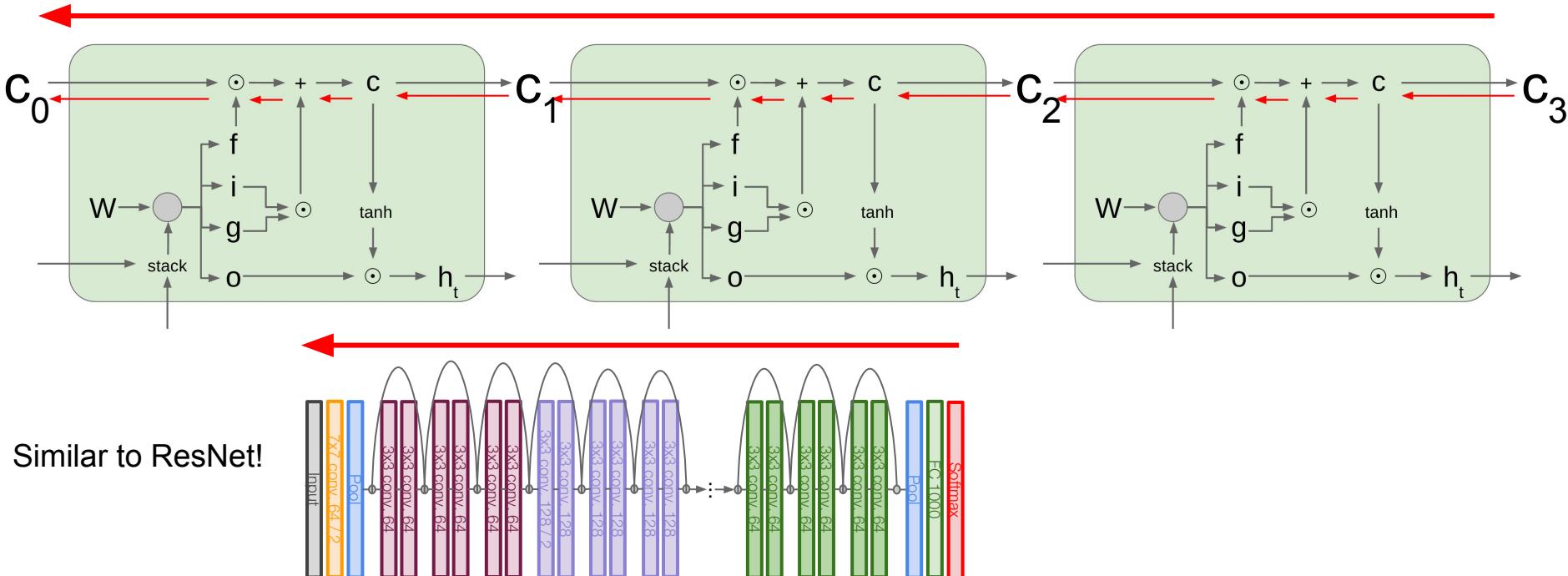
Uninterrupted gradient flow!



Long Short Term Memory (LSTM): Gradient Flow

[Hochreiter et al., 1997]

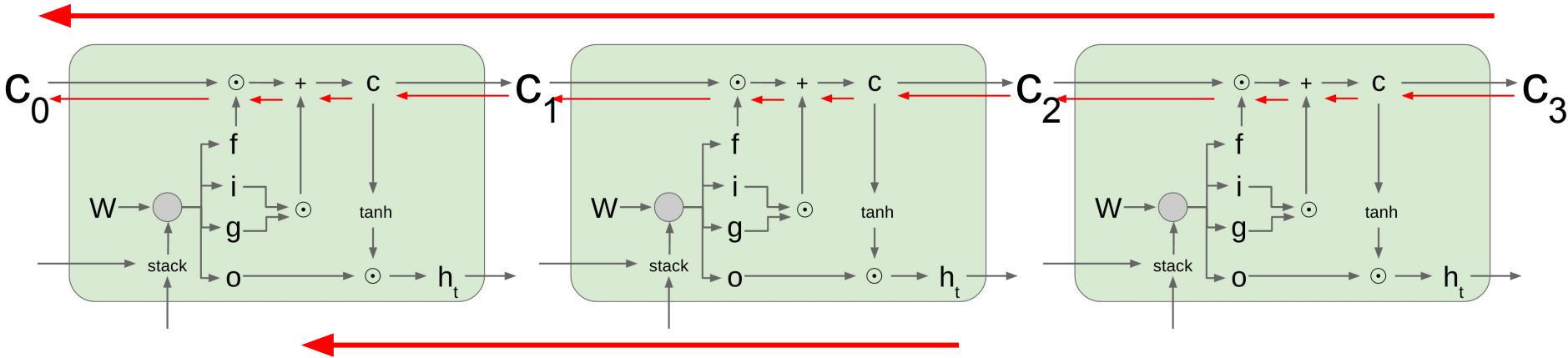
Uninterrupted gradient flow!



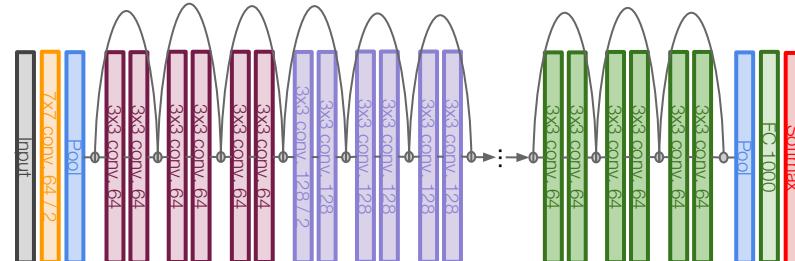
Long Short Term Memory (LSTM): Gradient Flow

[Hochreiter et al., 1997]

Uninterrupted gradient flow!



Similar to ResNet!



In between:
Highway Networks

$$g = T(x, W_T)$$

$$y = g \odot H(x, W_H) + (1 - g) \odot x$$

Srivastava et al., "Highway Networks",
ICML DL Workshop 2015

Other RNN Variants

GRU [*Learning phrase representations using rnn encoder-decoder for statistical machine translation*, Cho et al. 2014]

$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r)$$

$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z)$$

$$\tilde{h}_t = \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h)$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$$

[*LSTM: A Search Space Odyssey*, Greff et al., 2015]

[*An Empirical Exploration of Recurrent Network Architectures*, Jozefowicz et al., 2015]

MUT1:

$$z = \text{sigm}(W_{xz}x_t + b_z)$$

$$r = \text{sigm}(W_{xr}x_t + W_{hr}h_t + b_r)$$

$$h_{t+1} = \tanh(W_{hh}(r \odot h_t) + \tanh(x_t) + b_h) \odot z + h_t \odot (1 - z)$$

MUT2:

$$z = \text{sigm}(W_{xz}x_t + W_{hx}h_t + b_z)$$

$$r = \text{sigm}(x_t + W_{hr}h_t + b_r)$$

$$h_{t+1} = \tanh(W_{hh}(r \odot h_t) + W_{xh}x_t + b_h) \odot z + h_t \odot (1 - z)$$

MUT3:

$$z = \text{sigm}(W_{xz}x_t + W_{hz}\tanh(h_t) + b_z)$$

$$r = \text{sigm}(W_{xr}x_t + W_{hr}h_t + b_r)$$

$$h_{t+1} = \tanh(W_{hh}(r \odot h_t) + W_{xh}x_t + b_h) \odot z + h_t \odot (1 - z)$$

Summary

- RNNs allow a lot of flexibility in architecture design
- Vanilla RNNs are simple but don't work very well
- Common to use LSTM or GRU: their additive interactions improve gradient flow
- Backward flow of gradients in RNN can explode or vanish. Exploding is controlled with gradient clipping. Vanishing is controlled with additive interactions (LSTM)
- Better/simpler architectures are a hot topic of current research
- Better understanding (both theoretical and empirical) is needed.

Next time: Midterm!

Lecture 11: Detection and Segmentation

Administrative

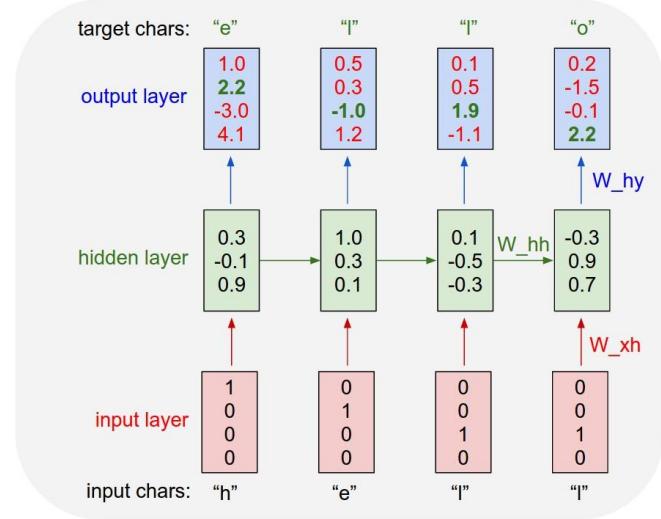
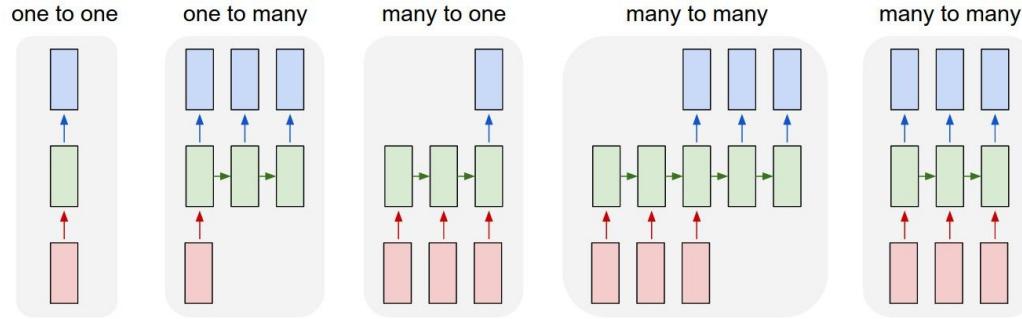
1. Midterms being graded

- **Please don't** discuss midterms until next week - some students not yet taken

2. A2 being graded

3. Project milestones due Wed 5/16

Last Time: Recurrent Networks



Last Time: Recurrent Networks

For $\bigoplus_{n=1,\dots,m} \mathcal{L}_{m,n} = 0$, hence we can find a closed subset \mathcal{H} in \mathcal{H} and any sets \mathcal{F} on X , U is a closed immersion of S , then $U \rightarrow T$ is a separated algebraic space.

Proof. Proof of (1). It also start we get

$$S = \text{Spec}(R) = U \times_X U \times_X U$$

and the comparicoly in the fibre product covering we have to prove the lemma generated by $\coprod Z \times_U U \rightarrow V$. Consider the maps M along the set of points \mathcal{Sch}_{fppf} and $U \rightarrow U$ is the fibre category of S in U in Section ?? and the fact that any U affine, see Morphisms, Lemma ?? . Hence we obtain a scheme S and any open subset $W \subset U$ in $\mathcal{Sh}(G)$ such that $\text{Spec}(R') \rightarrow S$ is smooth or an

$$U = \bigcup U_i \times_{S_i} U_i$$

which has a nonzero morphism we may assume that f_i is of finite presentation over S . We claim that $\mathcal{O}_{X,x}$ is a scheme where $x, x', s'' \in S'$ such that $\mathcal{O}_{X,x'} \rightarrow \mathcal{O}_{X',x'}$ is separated. By Algebra, Lemma ?? we can define a map of complexes $\text{GL}_{S'}(x'/S'')$ and we win. \square

To prove study we see that $\mathcal{F}|_U$ is a covering of \mathcal{X}' , and \mathcal{T}_i is an object of $\mathcal{F}_{X/S}$ for $i > 0$ and \mathcal{F}_0 exists and let \mathcal{F}_i be a presheaf of \mathcal{O}_X -modules on \mathcal{C} as a \mathcal{F} -module. In particular $\mathcal{F} = U/\mathcal{F}$ we have to show that

$$\widetilde{M}^\bullet = \mathcal{I}^\bullet \otimes_{\text{Spec}(k)} \mathcal{O}_{S,s} - i_X^{-1} \mathcal{F}$$

is a unique morphism of algebraic stacks. Note that

$$\text{Arrows} = (\mathcal{Sch}/S)_{fppf}^{\text{opp}}, (\mathcal{Sch}/S)_{fppf}$$

and

$$V = \Gamma(S, \mathcal{O}) \longrightarrow (U, \text{Spec}(A))$$

is an open subset of X . Thus U is affine. This is a continuous map of X is the inverse, the groupoid scheme S .

Proof. See discussion of sheaves of sets. \square

The result for prove any open covering follows from the less of Example ?? . It may replace S by $X_{\text{spaces},\text{étale}}$ which gives an open subspace of X and T equal to S_{Zar} , see Descent, Lemma ?? . Namely, by Lemma ?? we see that R is geometrically regular over S .

PANDARUS:

Alas, I think he shall be come approached and the day
 When little strain would be attain'd into being never fed,
 And who is but a chain and subjects of his death,
 I should not sleep.

Second Senator:

They are away this miseries, produced upon my soul,
 Breaking and strongly should be buried, when I perish
 The earth and thoughts of many states.

DUKE VINCENTIO:

Well, your wit is in the care of side and that.

Second Lord:

They would be ruled after this chamber, and
 my fair nues begun out of the fact, to be conveyed,
 Whose noble souls I'll have the heart of the wars.

Clown:

Come, sir, I will make did behold your worship.

VIOLA:

I'll drink it.

```
static void do_command(struct seq_file *m, void *v)
{
    int column = 32 << (cmd[2] & 0x80);
    if (state)
        cmd = (int)(int_state ^ (in_8(&ch->ch_flags) & Cmd) ? 2 : 1);
    else
        seq = 1;
    for (i = 0; i < 16; i++) {
        if (k & (1 << i))
            pipe = (in_use & UMXTHREAD_UNCCA) +
                ((count & 0x00000000fffffff8) & 0x0000000f) << 8;
        if (count == 0)
            sub(pid, ppc_md.kexec_handle, 0x20000000);
        pipe_set_bytes(i, 0);
    }
    /* Free our user pages pointer to place camera if all dash */
    subsystem_info = &of_changes[PAGE_SIZE];
    rek_controls(offset, idx, &offset);
    /* Now we want to deliberately put it to device */
    control_check_polarity(&context, val, 0);
    for (i = 0; i < COUNTER; i++)
        seq_puts(s, "policy ");
}
```

Last Time: Recurrent Networks

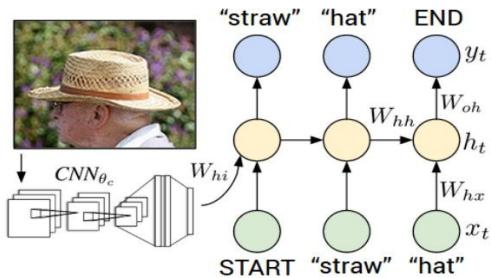


Figure from Karpathy et al, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015; figure copyright IEEE, 2015.

Reproduced for educational purposes.



A cat sitting on a suitcase on the floor



A cat is sitting on a tree branch



A woman is holding a cat in her hand



Two people walking on the beach with surfboards



A tennis player in action on the court



A person holding a computer mouse on a desk

So far: Image Classification



This image is CC0 public domain

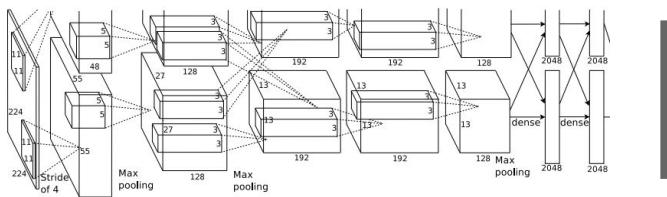


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

Vector:
4096

Fully-Connected:
4096 to 1000
...

Class Scores
Cat: 0.9
Dog: 0.05
Car: 0.01

Today: Detection, Segmentation

Other Computer Vision Tasks

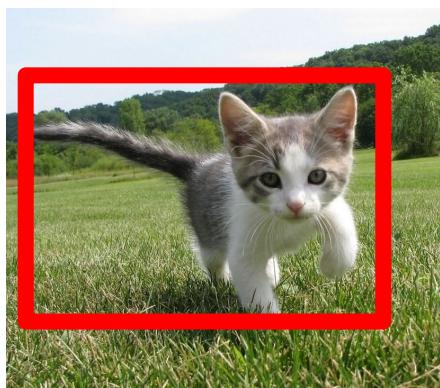
Semantic Segmentation



GRASS, CAT,
TREE, SKY

No objects, just pixels

Classification + Localization



CAT

Single Object

Object Detection



DOG, DOG, CAT

Multiple Object

Instance Segmentation



DOG, DOG, CAT

This image is CC0 public domain

Other Computer Vision Tasks

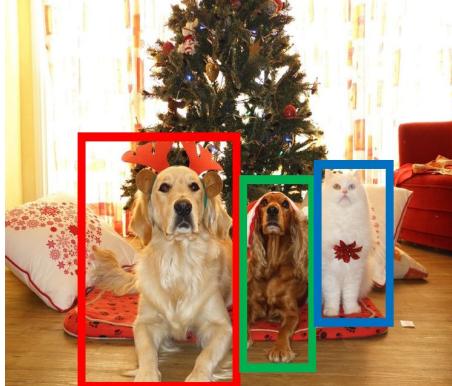
Semantic Segmentation



**GRASS, CAT,
TREE, SKY**

No objects, just pixels

2D Object Detection



DOG, DOG, CAT

Object categories +
2D bounding boxes

3D Object Detection



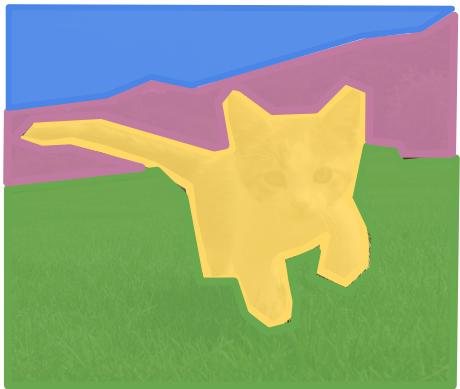
Car

Object categories +
3D bounding boxes

This image is CC0 public domain

Semantic Segmentation

Semantic Segmentation



**GRASS, CAT,
TREE, SKY**

No objects, just pixels

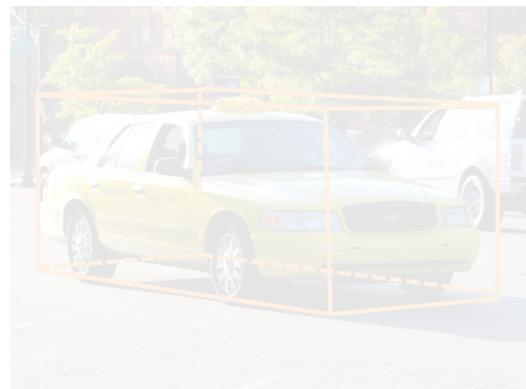
2D Object Detection



DOG, DOG, CAT

Object categories +
2D bounding boxes

3D Object Detection



Car

Object categories +
3D bounding boxes

This image is CC0 public domain

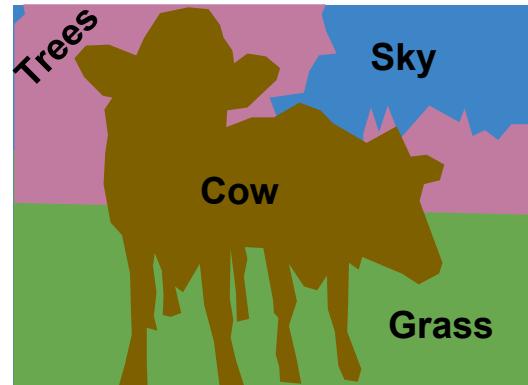
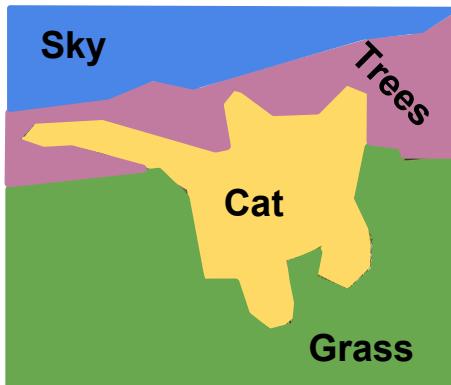
Semantic Segmentation

Label each pixel in the image with a category label

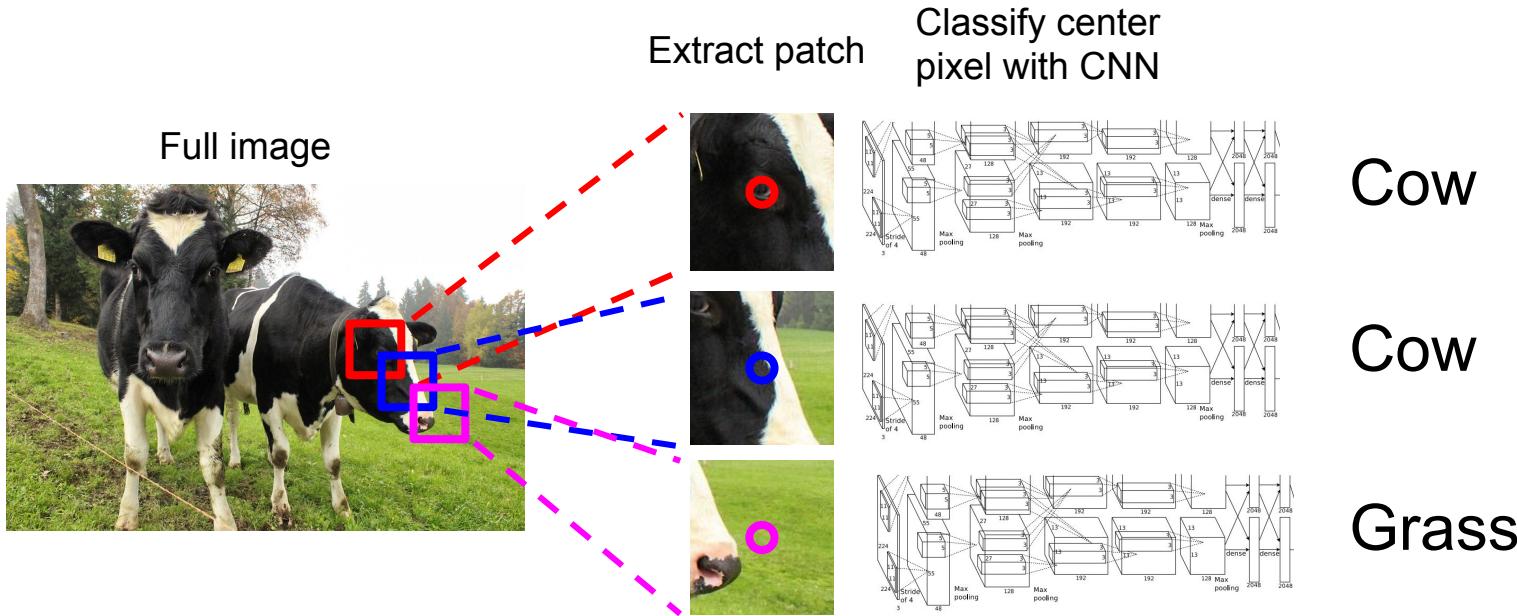
Don't differentiate instances, only care about pixels



[This image is CC0 public domain](#)



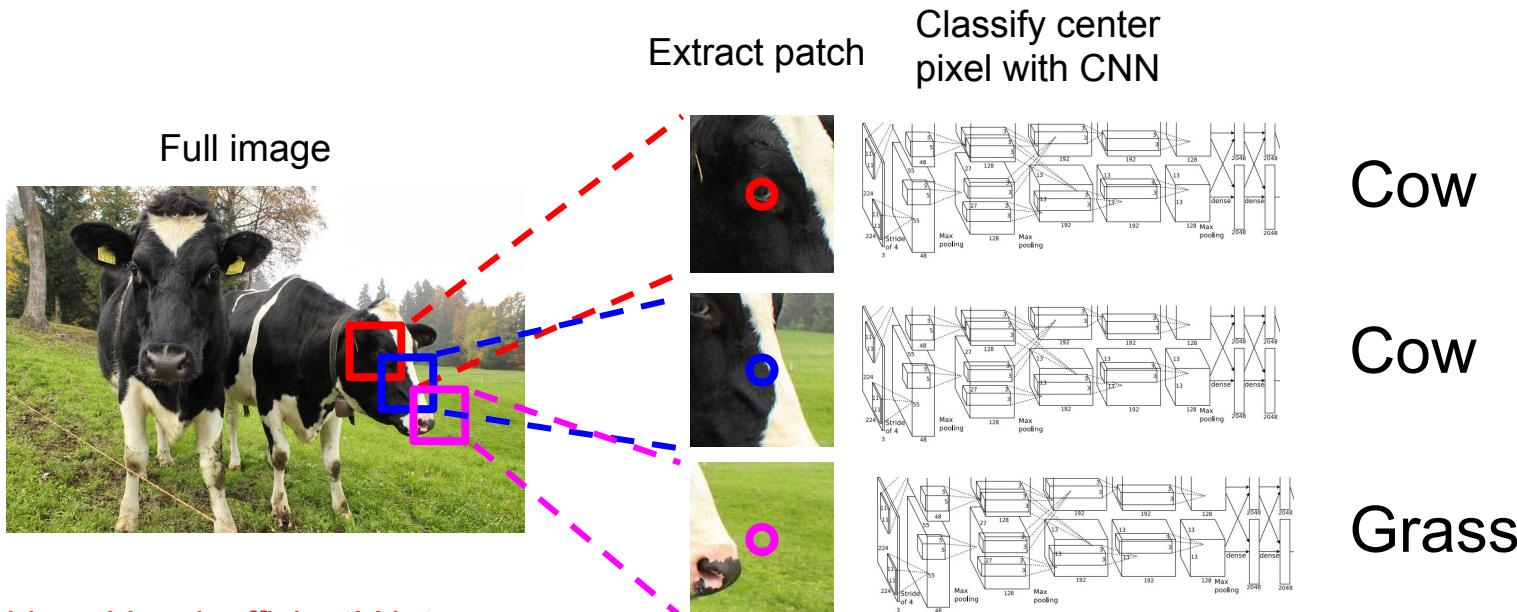
Semantic Segmentation Idea: Sliding Window



Farabet et al, "Learning Hierarchical Features for Scene Labeling," TPAMI 2013

Pinheiro and Collobert, "Recurrent Convolutional Neural Networks for Scene Labeling", ICML 2014

Semantic Segmentation Idea: Sliding Window

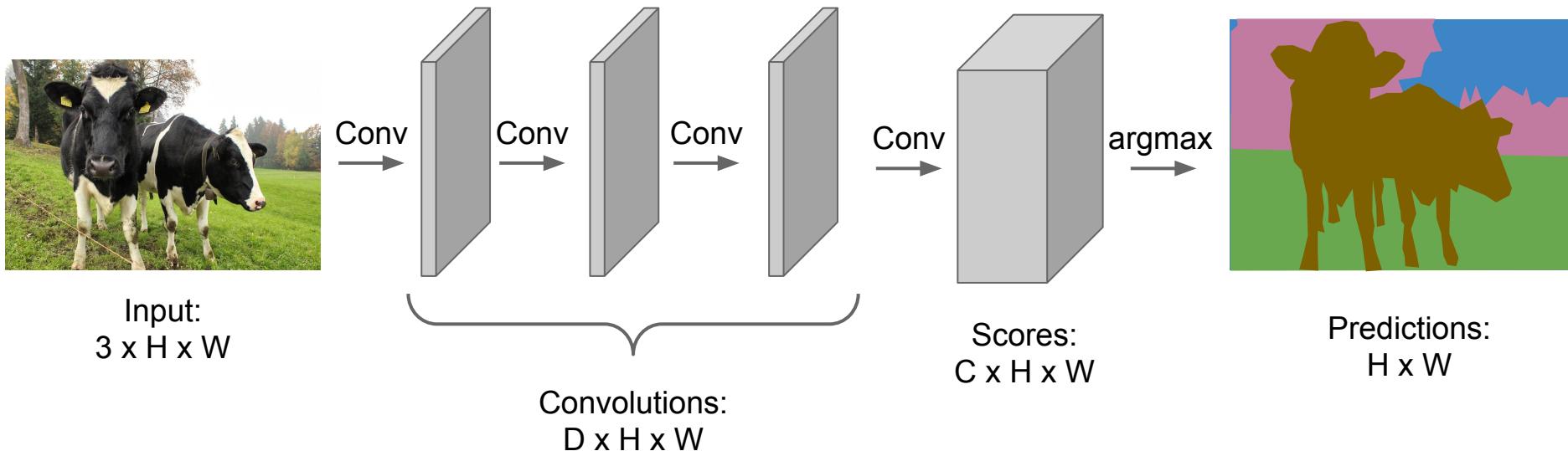


Problem: Very inefficient! Not reusing shared features between overlapping patches

Farabet et al, "Learning Hierarchical Features for Scene Labeling," TPAMI 2013
Pinheiro and Collobert, "Recurrent Convolutional Neural Networks for Scene Labeling", ICML 2014

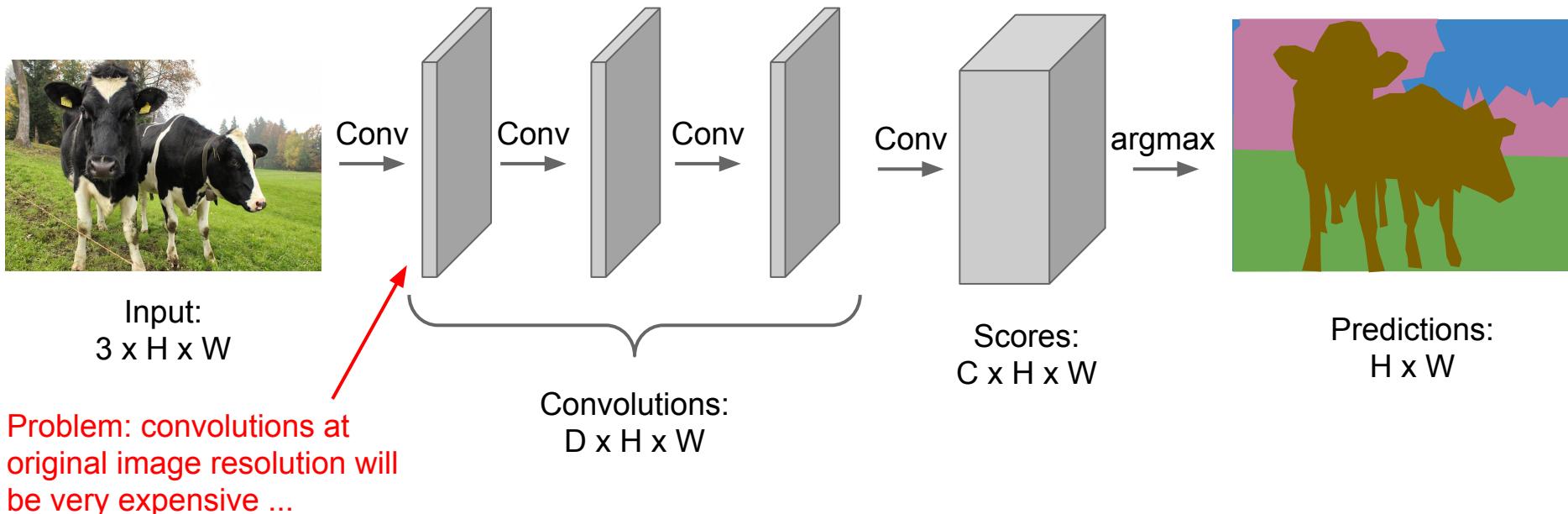
Semantic Segmentation Idea: Fully Convolutional

Design a network as a bunch of convolutional layers
to make predictions for pixels all at once!



Semantic Segmentation Idea: Fully Convolutional

Design a network as a bunch of convolutional layers
to make predictions for pixels all at once!

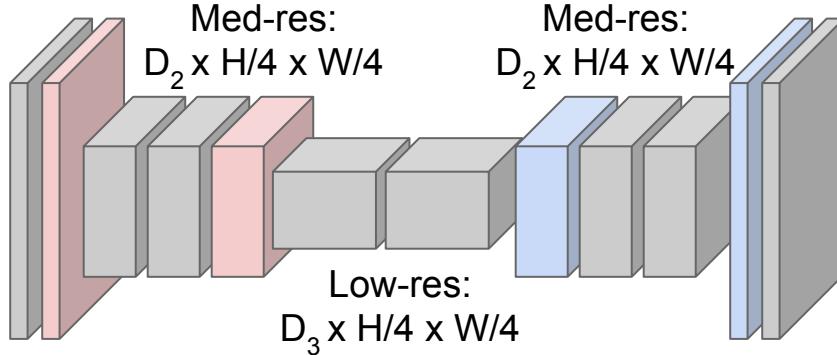


Semantic Segmentation Idea: Fully Convolutional

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



Input:
 $3 \times H \times W$



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

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



Predictions:
 $H \times W$

Long, Shelhamer, and Darrell, "Fully Convolutional Networks for Semantic Segmentation", CVPR 2015

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

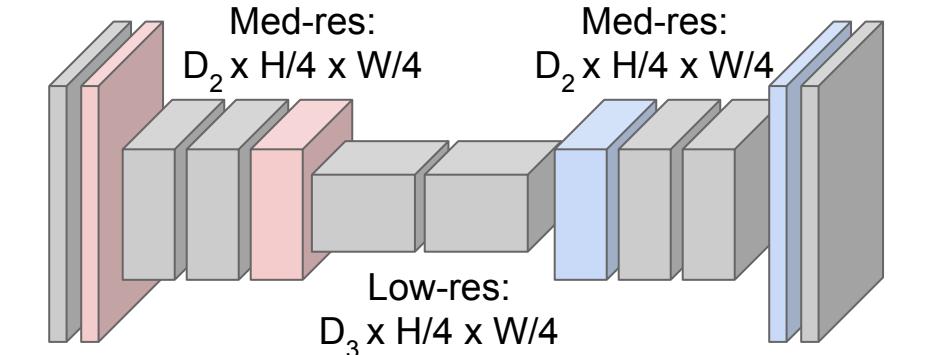
Semantic Segmentation Idea: Fully Convolutional

Downsampling:
Pooling, strided
convolution



Input:
 $3 \times H \times W$

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



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

Upsampling:
???



Predictions:
 $H \times W$

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: “Unpooling”

Nearest Neighbor

1	2
3	4



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

Input: 2 x 2

Output: 4 x 4

“Bed of Nails”

1	2
3	4



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

Output: 4 x 4

In-Network upsampling: “Max Unpooling”

Max Pooling

Remember which element was max!

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

Input: 4 x 4

5	6
7	8

Output: 2 x 2

Max Unpooling

Use positions from pooling layer

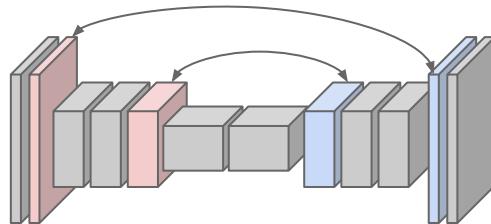
1	2
3	4

Rest of the network

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

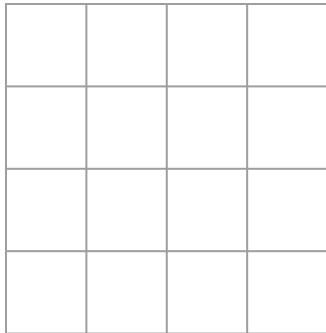
Output: 4 x 4

Corresponding pairs of
downsampling and
upsampling layers

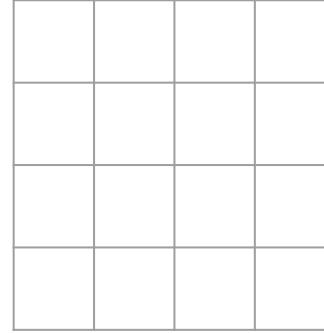


Learnable Upsampling: Transpose Convolution

Recall: Typical 3×3 convolution, stride 1 pad 1



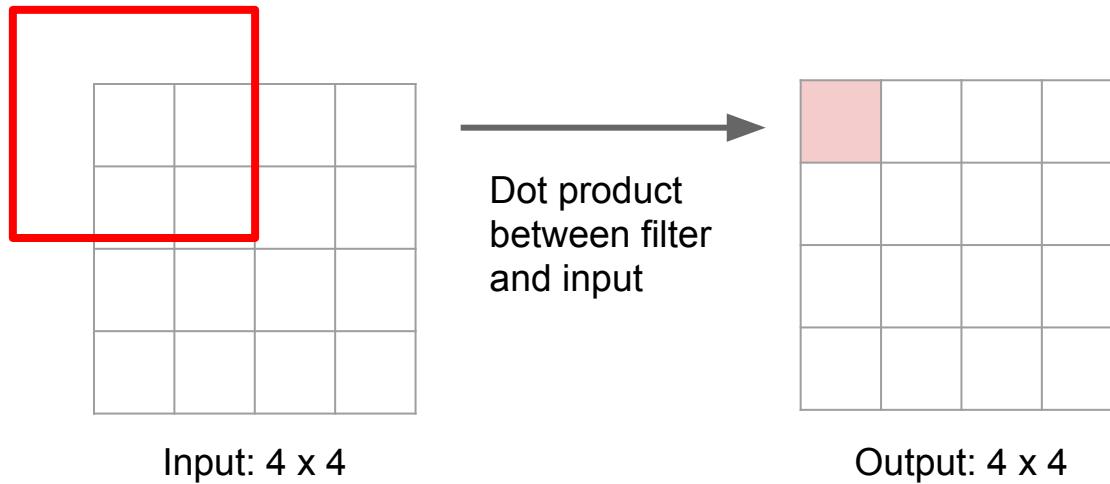
Input: 4×4



Output: 4×4

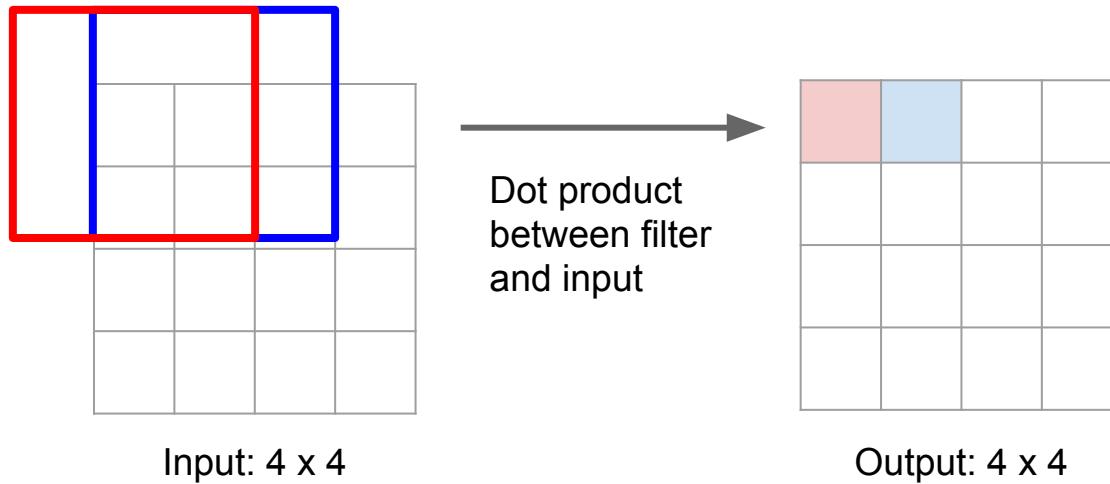
Learnable Upsampling: Transpose Convolution

Recall: Normal 3×3 convolution, stride 1 pad 1



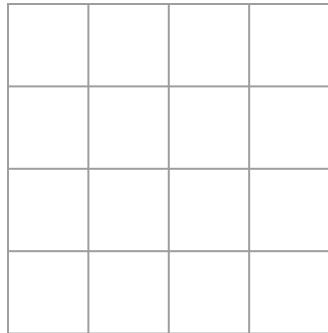
Learnable Upsampling: Transpose Convolution

Recall: Normal 3×3 convolution, stride 1 pad 1

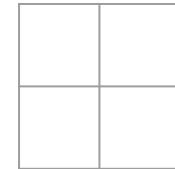


Learnable Upsampling: Transpose Convolution

Recall: Normal 3×3 convolution, stride 2 pad 1



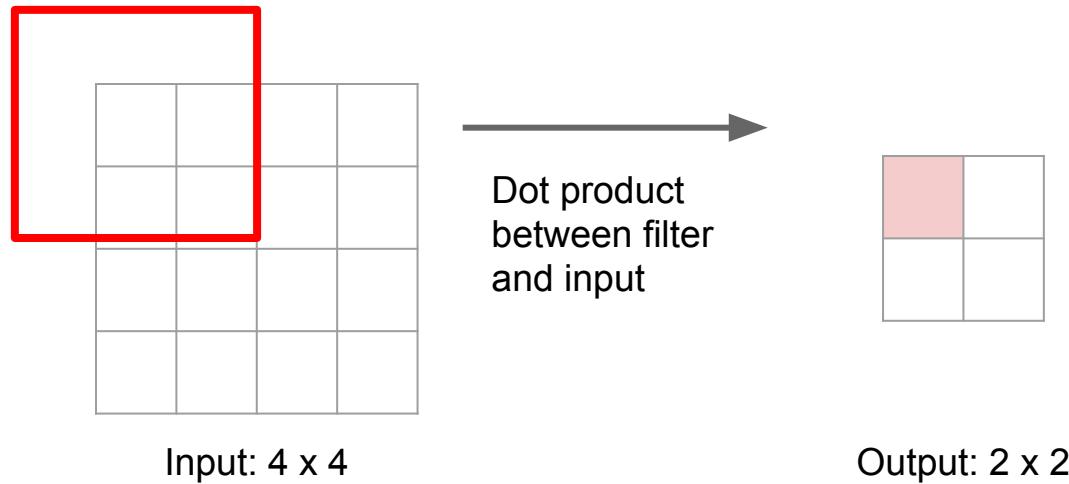
Input: 4×4



Output: 2×2

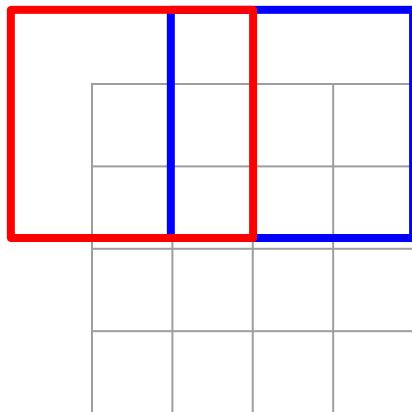
Learnable Upsampling: Transpose Convolution

Recall: Normal 3×3 convolution, stride 2 pad 1



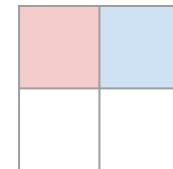
Learnable Upsampling: Transpose Convolution

Recall: Normal 3×3 convolution, stride 2 pad 1



Input: 4×4

Dot product
between filter
and input



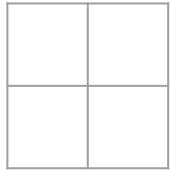
Output: 2×2

Filter moves 2 pixels in
the input for every one
pixel in the output

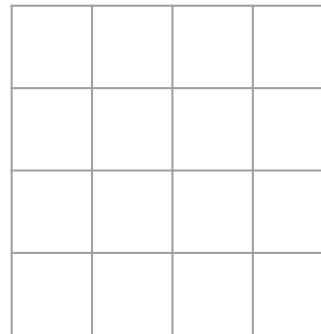
Stride gives ratio between
movement in input and
output

Learnable Upsampling: Transpose Convolution

3×3 **transpose** convolution, stride 2 pad 1



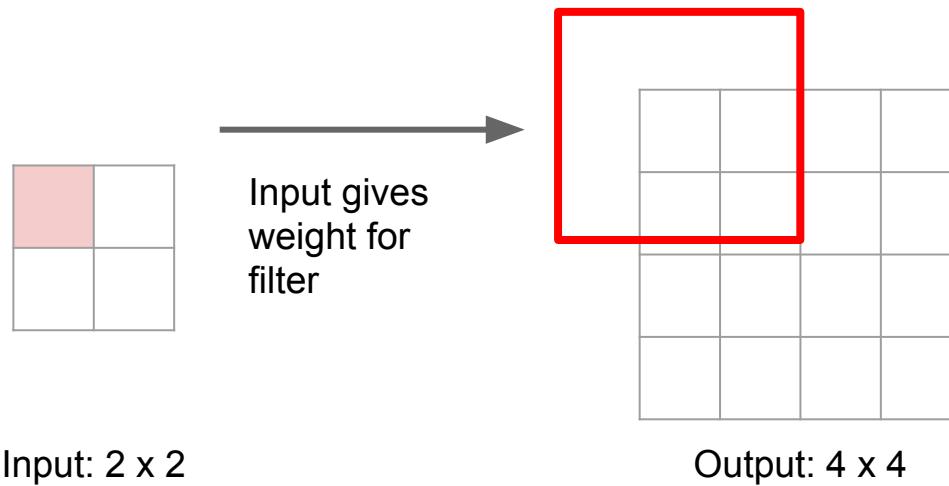
Input: 2×2



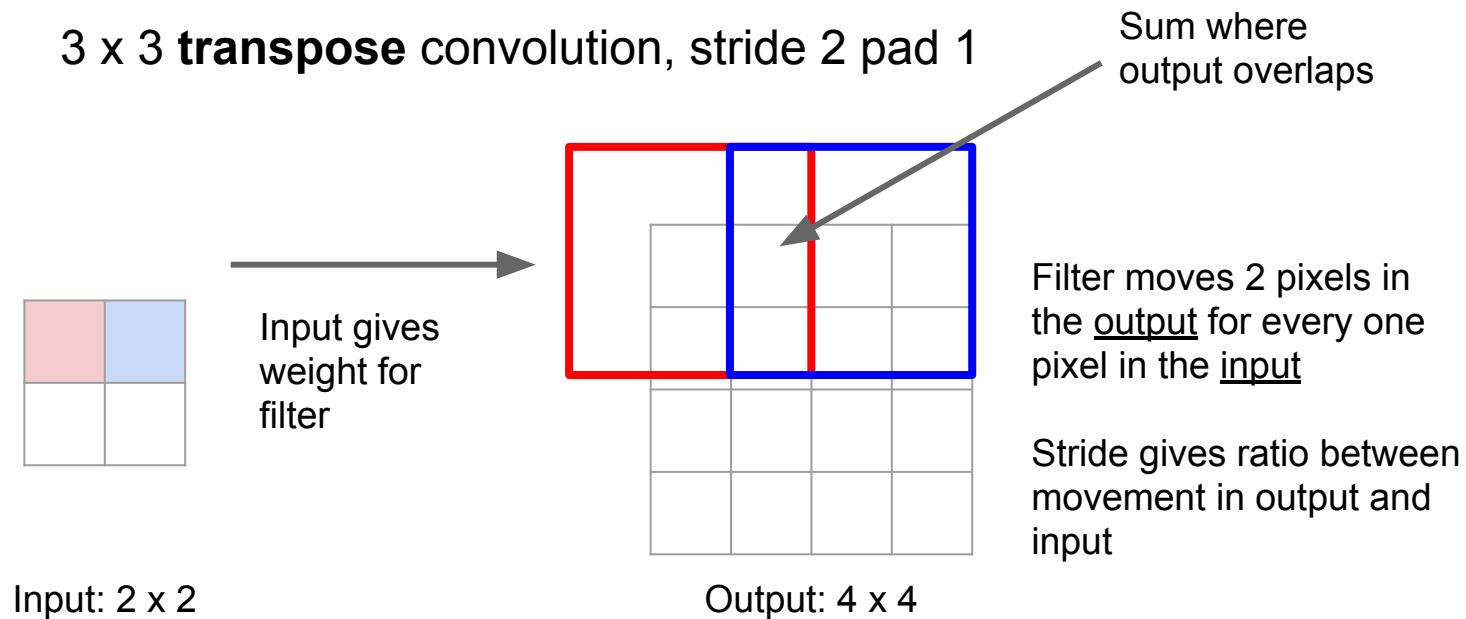
Output: 4×4

Learnable Upsampling: Transpose Convolution

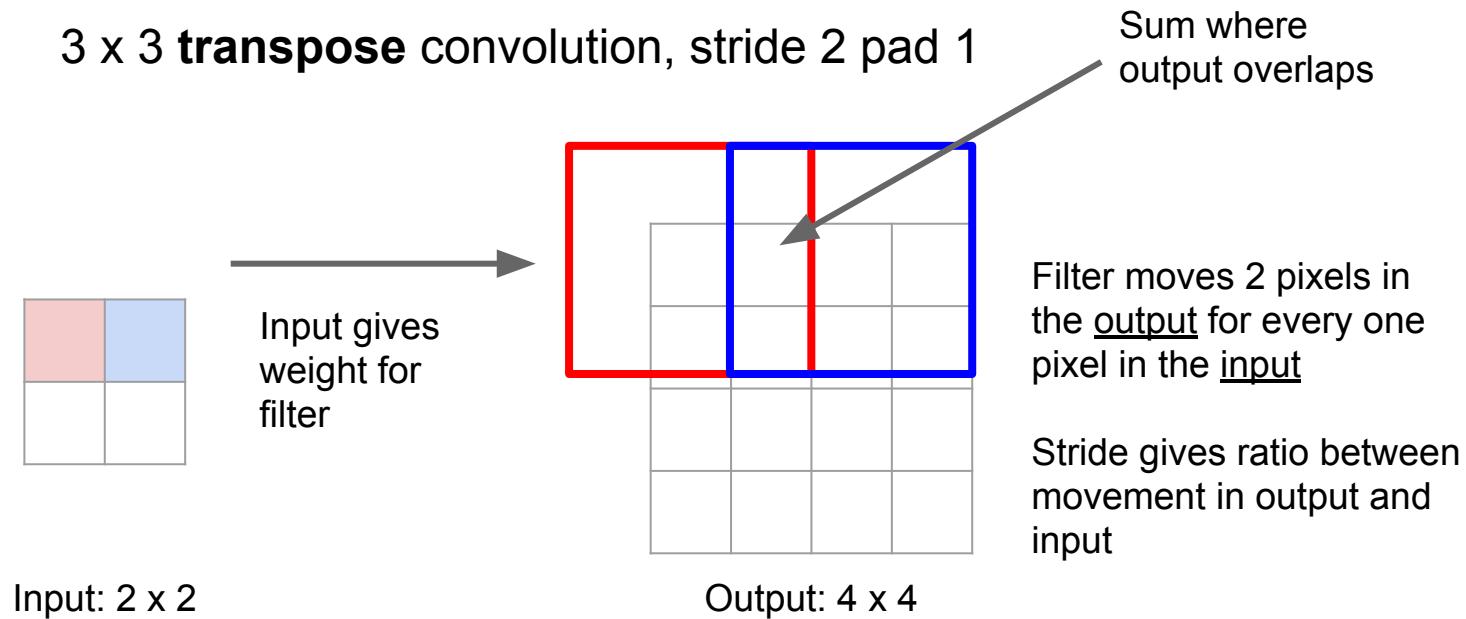
3 x 3 **transpose** convolution, stride 2 pad 1



Learnable Upsampling: Transpose Convolution



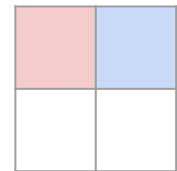
Learnable Upsampling: Transpose Convolution



Learnable Upsampling: Transpose Convolution

Other names:

- Deconvolution (bad)
- Upconvolution
- Fractionally strided convolution
- Backward strided convolution

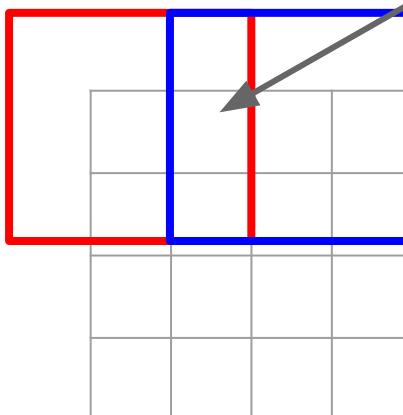


Input: 2 x 2

3 x 3 transpose convolution, stride 2 pad 1



Input gives weight for filter



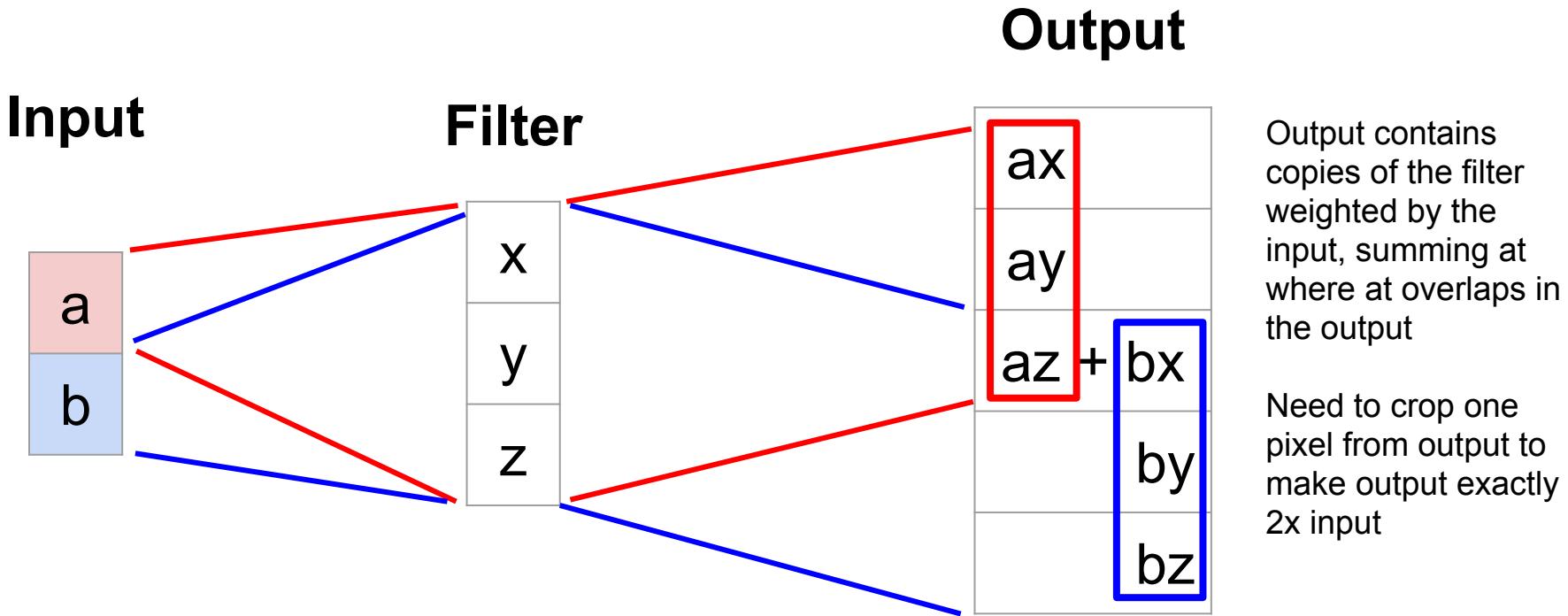
Output: 4 x 4

Filter moves 2 pixels in the output for every one pixel in the input

Stride gives ratio between movement in output and input

Sum where output overlaps

Learnable Upsampling: 1D Example



Convolution as Matrix Multiplication (1D Example)

We can express convolution in terms of a matrix multiplication

$$\vec{x} * \vec{a} = X\vec{a}$$

$$\begin{bmatrix} x & y & x & 0 & 0 & 0 \\ 0 & x & y & x & 0 & 0 \\ 0 & 0 & x & y & x & 0 \\ 0 & 0 & 0 & x & y & x \end{bmatrix} \begin{bmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ay + bz \\ ax + by + cz \\ bx + cy + dz \\ cx + dy \end{bmatrix}$$

Example: 1D conv, kernel size=3, stride=1, padding=1

Convolution as Matrix Multiplication (1D Example)

We can express convolution in terms of a matrix multiplication

$$\vec{x} * \vec{a} = X\vec{a}$$

$$\begin{bmatrix} x & y & x & 0 & 0 & 0 \\ 0 & x & y & x & 0 & 0 \\ 0 & 0 & x & y & x & 0 \\ 0 & 0 & 0 & x & y & x \end{bmatrix} \begin{bmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ay + bz \\ ax + by + cz \\ bx + cy + dz \\ cx + dy \end{bmatrix}$$

Example: 1D conv, kernel size=3, stride=1, padding=1

Convolution transpose multiplies by the transpose of the same matrix:

$$\vec{x} *^T \vec{a} = X^T \vec{a}$$

$$\begin{bmatrix} x & 0 & 0 & 0 \\ y & x & 0 & 0 \\ z & y & x & 0 \\ 0 & z & y & x \\ 0 & 0 & z & y \\ 0 & 0 & 0 & z \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} ax \\ ay + bx \\ az + by + cx \\ bz + cy + dx \\ cz + dy \\ dz \end{bmatrix}$$

When stride=1, convolution transpose is just a regular convolution (with different padding rules)

Convolution as Matrix Multiplication (1D Example)

We can express convolution in terms of a matrix multiplication

$$\vec{x} * \vec{a} = X\vec{a}$$

$$\begin{bmatrix} x & y & x & 0 & 0 & 0 \\ 0 & 0 & x & y & x & 0 \end{bmatrix} \begin{bmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ay + bz \\ bx + cy + dz \end{bmatrix}$$

Example: 1D conv, kernel size=3, stride=2, padding=1

Convolution as Matrix Multiplication (1D Example)

We can express convolution in terms of a matrix multiplication

$$\vec{x} * \vec{a} = X\vec{a}$$

$$\begin{bmatrix} x & y & z & 0 & 0 & 0 \\ 0 & 0 & x & y & z & 0 \end{bmatrix} \begin{bmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ay + bz \\ bx + cy + dz \end{bmatrix}$$

Example: 1D conv, kernel size=3, stride=2, padding=1

Convolution transpose multiplies by the transpose of the same matrix:

$$\vec{x} *^T \vec{a} = X^T \vec{a}$$

$$\begin{bmatrix} x & 0 \\ y & 0 \\ z & x \\ 0 & y \\ 0 & z \\ 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} ax \\ ay \\ az + bx \\ by \\ bz \\ 0 \end{bmatrix}$$

When stride>1, convolution transpose is no longer a normal convolution!

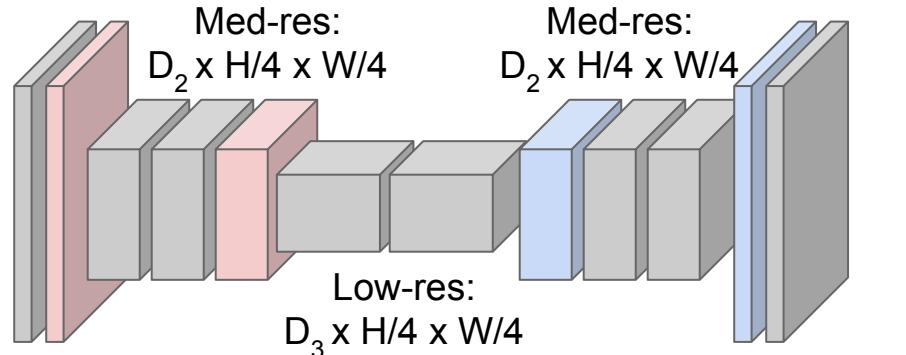
Semantic Segmentation Idea: Fully Convolutional

Downsampling:
Pooling, strided convolution



Input:
 $3 \times H \times W$

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



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

Low-res:
 $D_3 \times H/4 \times W/4$

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

Upsampling:
Unpooling or strided transpose convolution

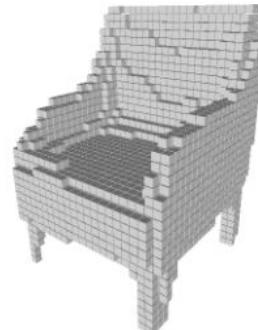


Predictions:
 $H \times W$

Long, Shelhamer, and Darrell, "Fully Convolutional Networks for Semantic Segmentation", CVPR 2015

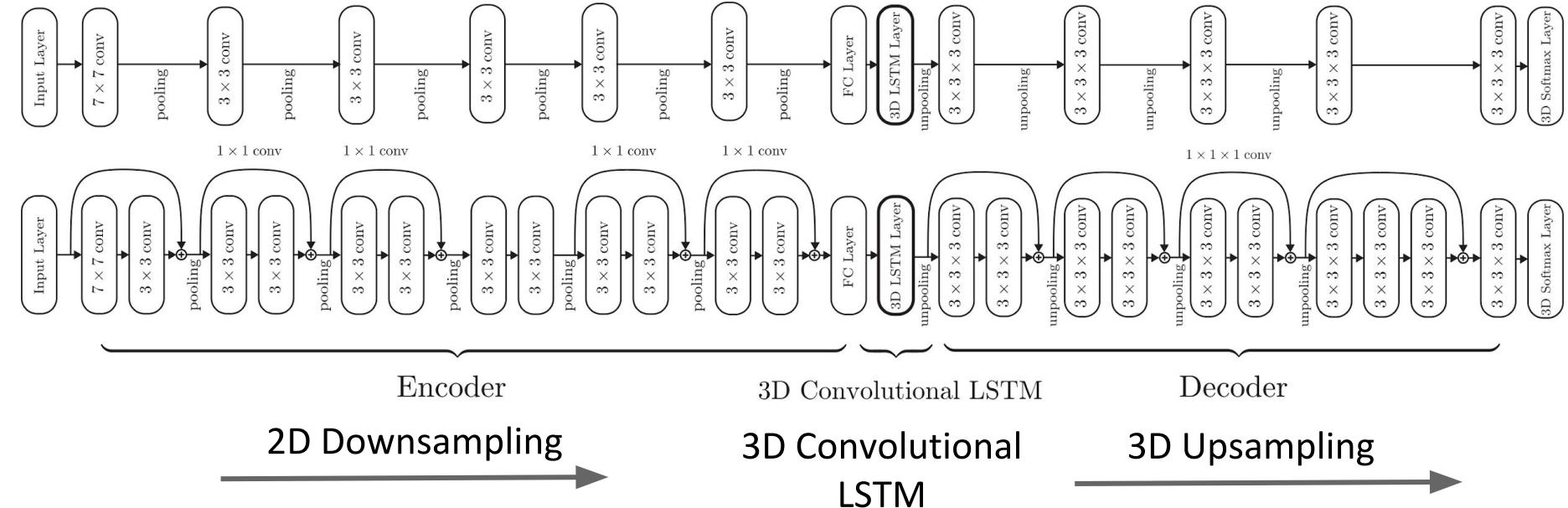
Noh et al, "Learning Deconvolution Network for Semantic Segmentation", ICCV 2015

Aside: Multi-view 3D Reconstruction

Image 1	Image 2	Image 3	Reconstruction
			

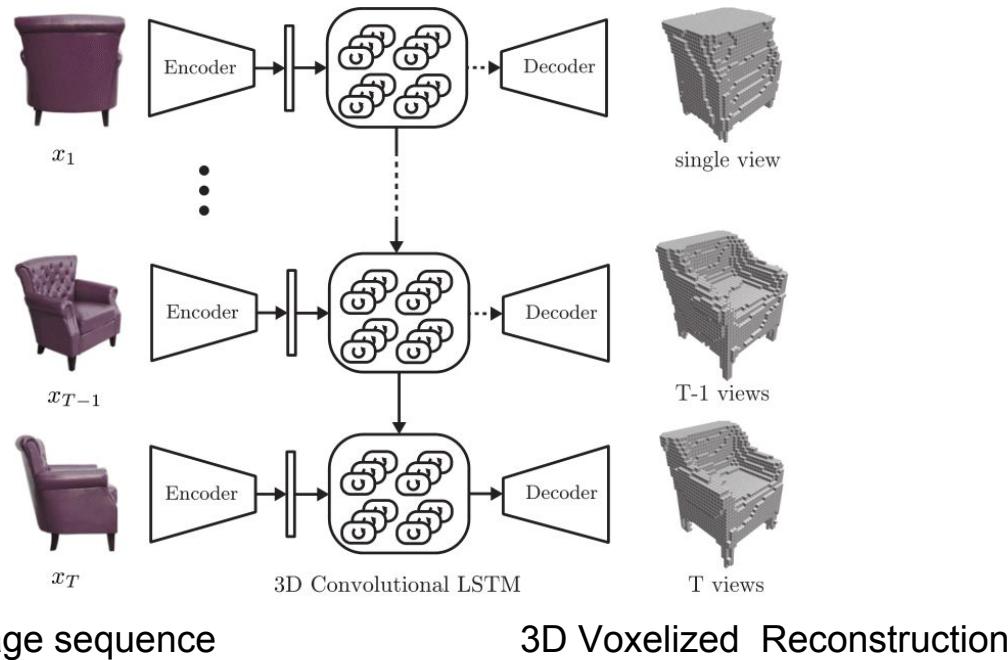
Choy, C. B., Xu, D., Gwak, J., Chen, K., & Savarese, S. (2016, October). 3d-r2n2: A unified approach for single and multi-view 3d object reconstruction. In European Conference on Computer Vision (pp. 628-644). Springer, Cham.

Aside: Multi-view 3D Reconstruction



Choy, C. B., Xu, D., Gwak, J., Chen, K., & Savarese, S. (2016, October). 3d-r2n2: A unified approach for single and multi-view 3d object reconstruction. In European Conference on Computer Vision (pp. 628-644). Springer, Cham.

Aside: Multi-view 3D Reconstruction



Choy, C. B., Xu, D., Gwak, J., Chen, K., & Savarese, S. (2016, October). 3d-r2n2: A unified approach for single and multi-view 3d object reconstruction. In European Conference on Computer Vision (pp. 628-644). Springer, Cham.

2D Object Detection

Semantic
Segmentation



GRASS, CAT,
TREE, SKY

No objects, just pixels

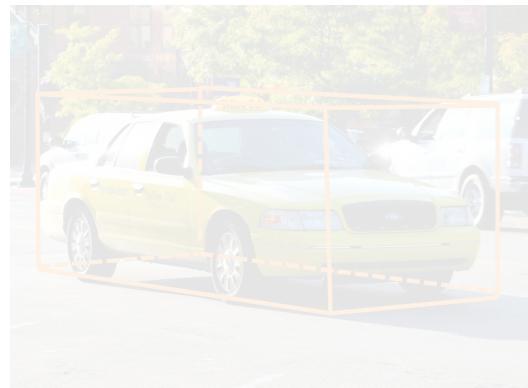
2D Object
Detection



DOG, DOG, CAT

Object categories +
2D bounding boxes

3D Object
Detection



Car

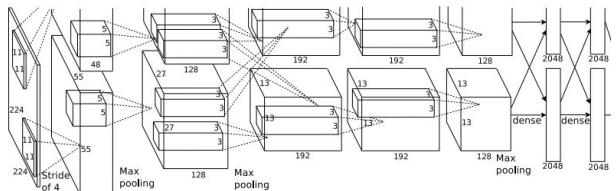
Object categories +
3D bounding boxes

This image is CC0 public domain

Classification + Localization



This image is CC0 public domain



Fully
Connected:
4096 to 1000

Class Scores

Cat: 0.9
Dog: 0.05
Car: 0.01
...

Vector: Fully
Connected:
4096 to 4

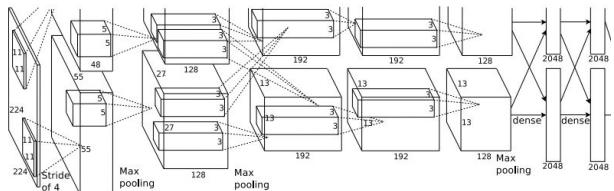
Box Coordinates (x, y, w, h)

Treat localization as a
regression problem!

Classification + Localization



This image is CC0 public domain



Treat localization as a
regression problem!

Fully
Connected:
4096 to 1000

Vector: Fully
Connected:
4096 4096 to 4

Correct label:
Cat

Softmax
Loss

Class Scores
Cat: 0.9
Dog: 0.05
Car: 0.01
...

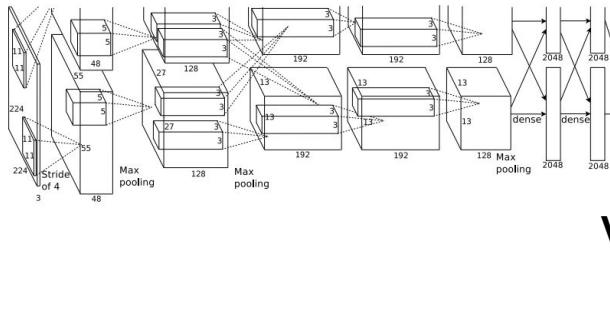
Box
Coordinates → L2 Loss
(x, y, w, h)

Correct box:
(x', y', w', h')

Classification + Localization



This image is CC0 public domain



Treat localization as a
regression problem!

Vector: Fully
Connected:
4096 to 4

Class Scores

Cat: 0.9
Dog: 0.05
Car: 0.01
...

Multitask Loss

Box
Coordinates → L2 Loss
(x, y, w, h)

Correct label:
Cat

Softmax
Loss

+

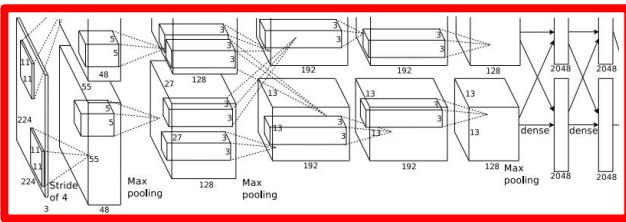
Loss

Correct box:
(x', y', w', h')

Classification + Localization



This image is CC0 public domain



Often pretrained on ImageNet
(Transfer learning)

Treat localization as a
regression problem!

Vector: 4096
Fully Connected: 4096 to 4

Class Scores
Cat: 0.9
Dog: 0.05
Car: 0.01
...

Box
Coordinates → L2 Loss
(x, y, w, h)

Correct label:
Cat

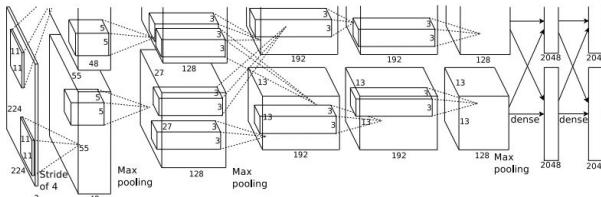
Softmax
Loss

+

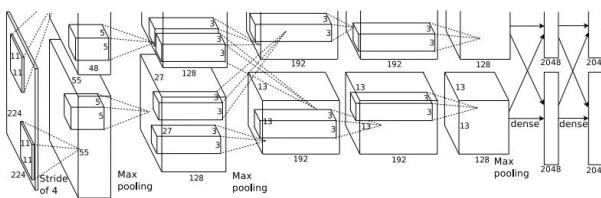
Loss

Correct box:
(x', y', w', h')

Object Detection as Regression?



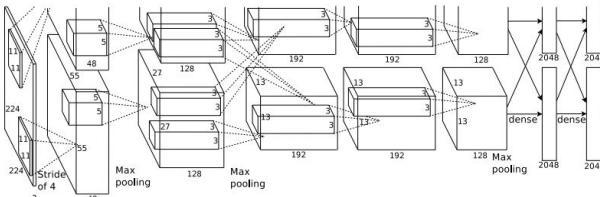
CAT: (x, y, w, h)



DOG: (x, y, w, h)

DOG: (x, y, w, h)

CAT: (x, y, w, h)



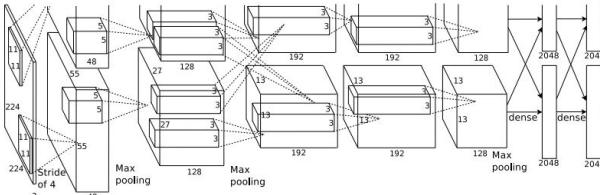
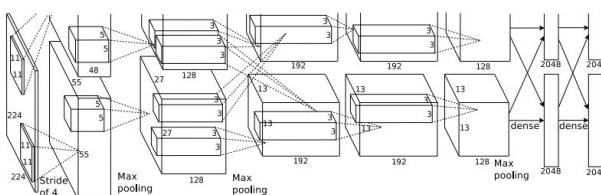
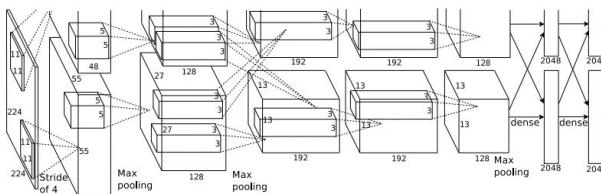
DUCK: (x, y, w, h)

DUCK: (x, y, w, h)

...

Object Detection as Regression?

Each image needs a different number of outputs!



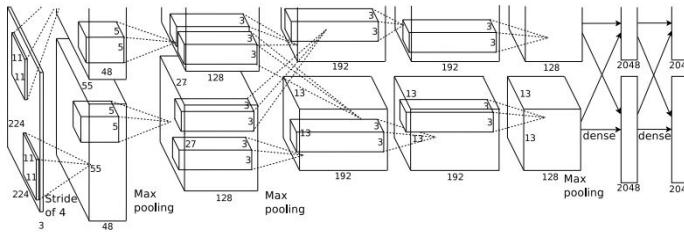
CAT: (x, y, w, h) 4 numbers

DOG: (x, y, w, h)
DOG: (x, y, w, h) 16 numbers
CAT: (x, y, w, h)

DUCK: (x, y, w, h) Many
DUCK: (x, y, w, h) numbers!
....

Object Detection as Classification: Sliding Window

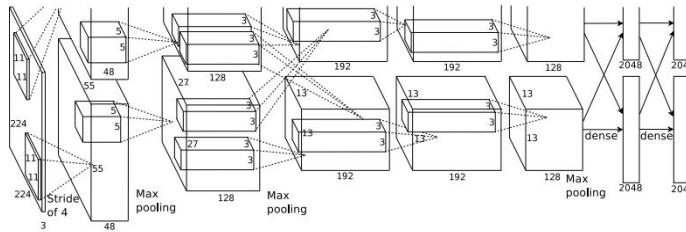
Apply a CNN to many different crops of the image, CNN classifies each crop as object or background



Dog? NO
Cat? NO
Background? YES

Object Detection as Classification: Sliding Window

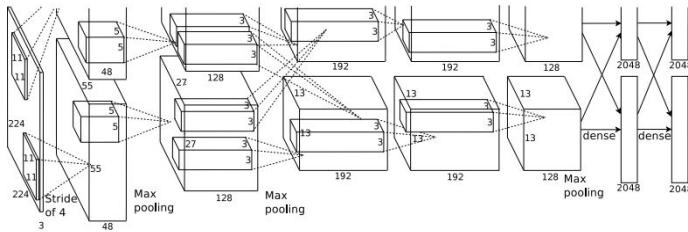
Apply a CNN to many different crops of the image, CNN classifies each crop as object or background



Dog? YES
Cat? NO
Background? NO

Object Detection as Classification: Sliding Window

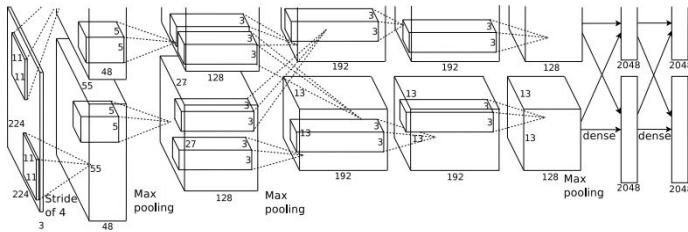
Apply a CNN to many different crops of the image, CNN classifies each crop as object or background



Dog? YES
Cat? NO
Background? NO

Object Detection as Classification: Sliding Window

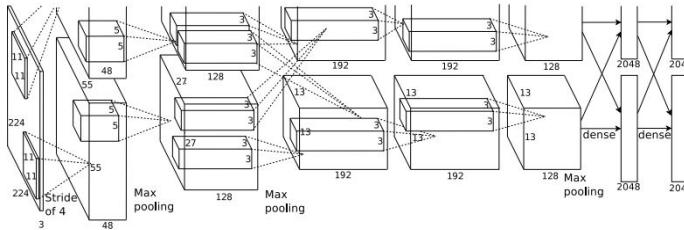
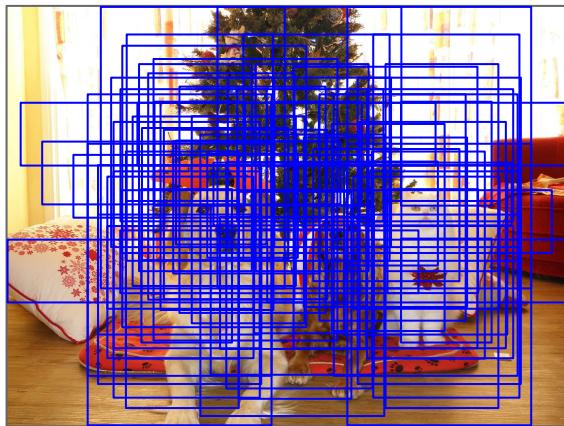
Apply a CNN to many different crops of the image, CNN classifies each crop as object or background



Dog? NO
Cat? YES
Background? NO

Object Detection as Classification: Sliding Window

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

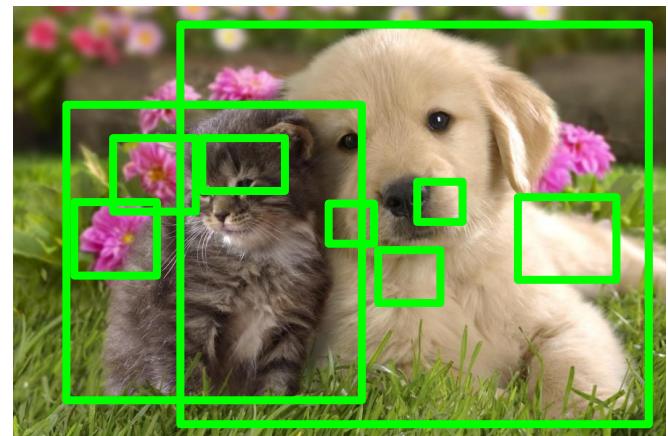


Dog? NO
Cat? YES
Background? NO

Problem: Need to apply CNN to huge number of locations, scales, and aspect ratios, very computationally expensive!

Region Proposals / Selective Search

- Find “blobby” image regions that are likely to contain objects
- 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



Input image

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

R-CNN

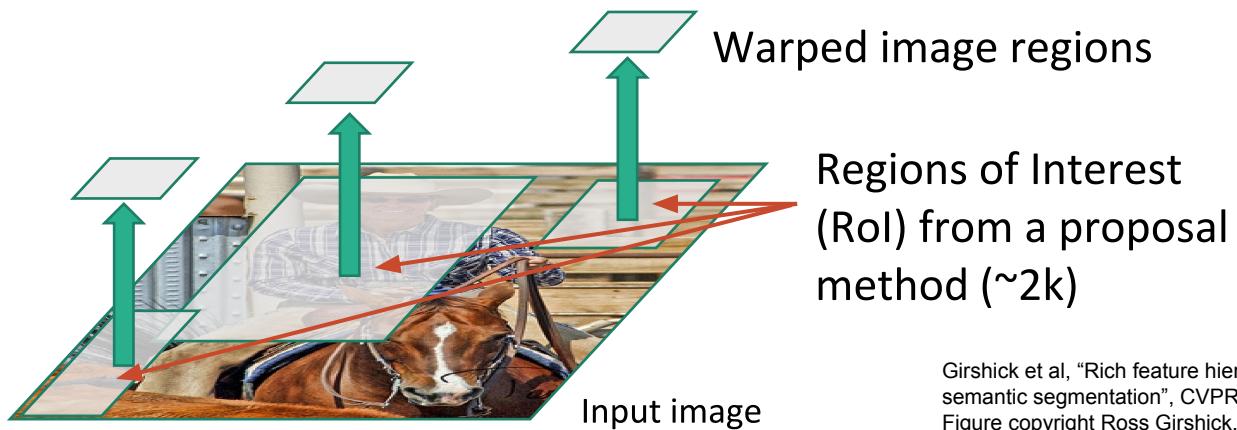


Input image

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

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

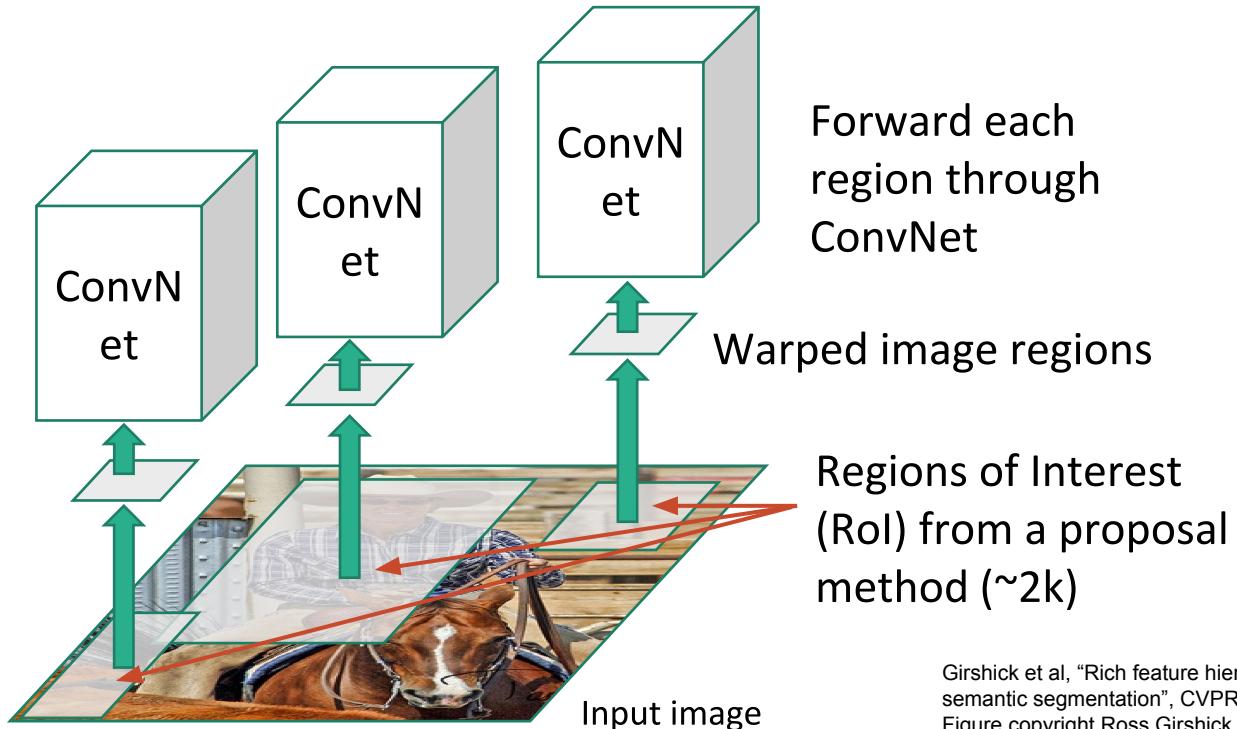
R-CNN



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

Figure copyright Ross Girshick, 2015; [source](#). Reproduced with permission.

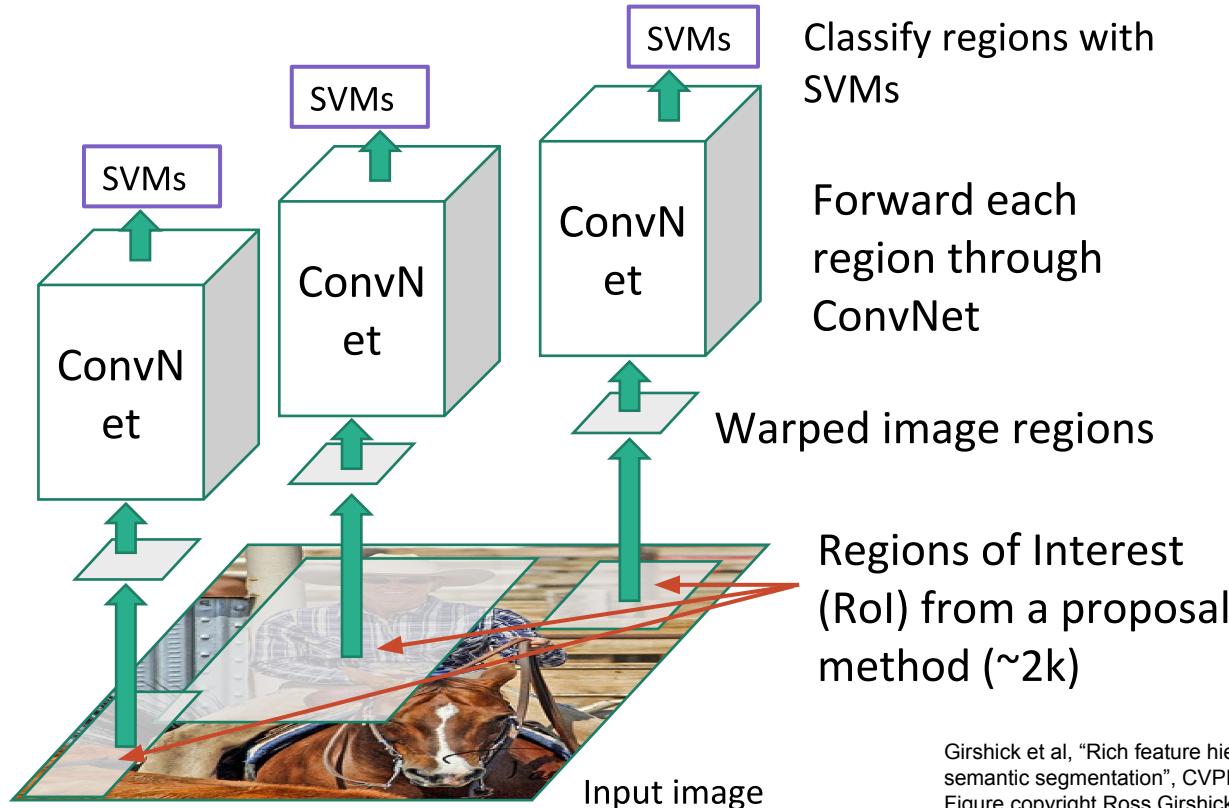
R-CNN



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

Figure copyright Ross Girshick, 2015; [source](#). Reproduced with permission.

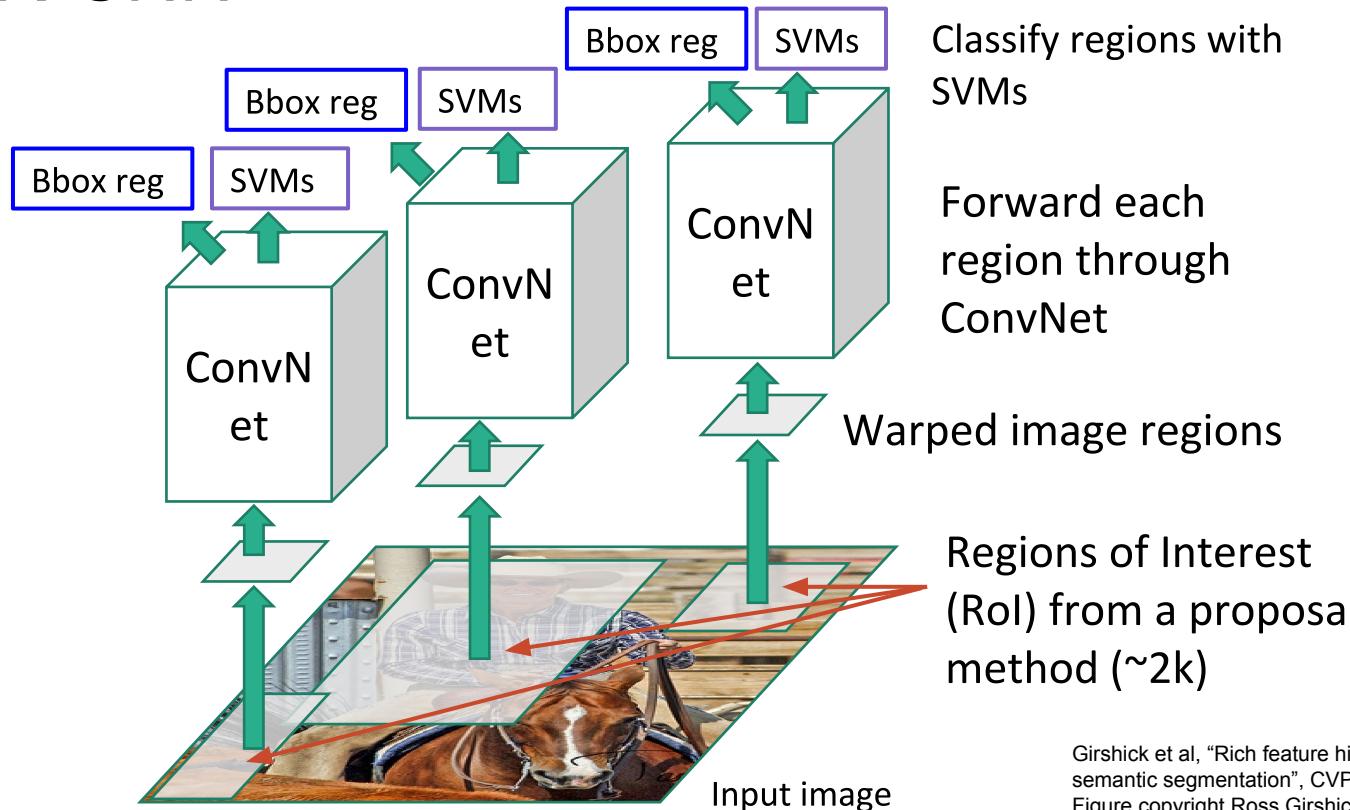
R-CNN



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

R-CNN

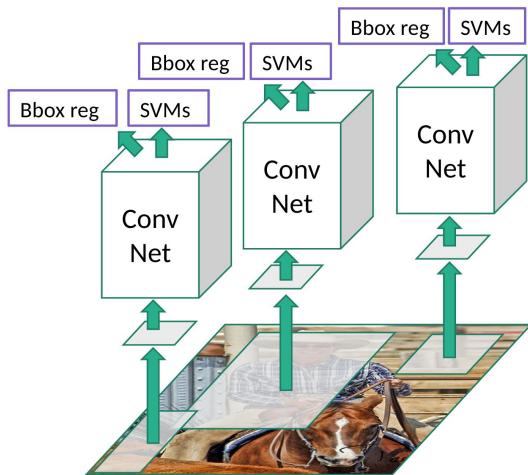
Linear Regression for bounding box offsets



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

R-CNN: Problems

- Ad hoc training objectives
 - Fine-tune network with softmax classifier (log loss)
 - Train post-hoc linear SVMs (hinge loss)
 - Train post-hoc bounding-box regressions (least squares)
- Training is slow (84h), takes a lot of disk space
- Inference (detection) is slow
 - 47s / image with VGG16 [Simonyan & Zisserman. ICLR15]
 - Fixed by SPP-net [He et al. ECCV14]



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

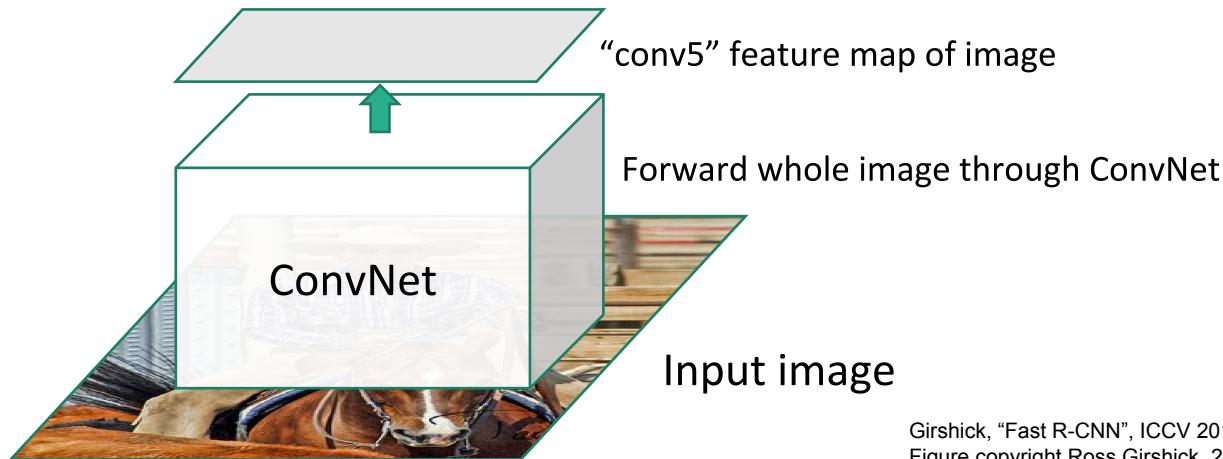
Fast R-CNN



Input image

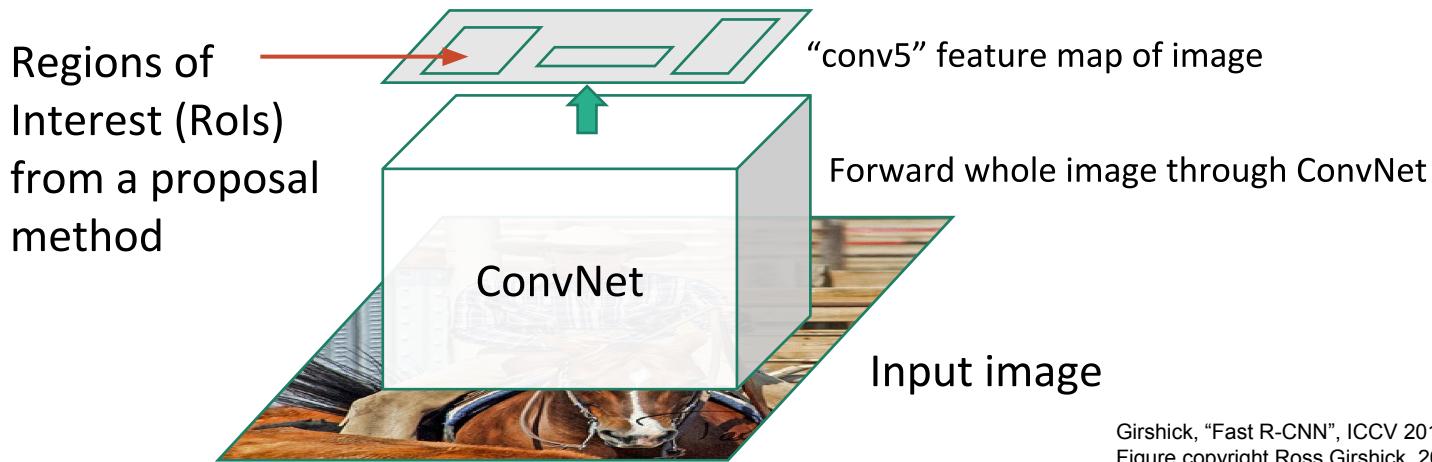
Girshick, "Fast R-CNN", ICCV 2015.
Figure copyright Ross Girshick, 2015; [source](#). Reproduced with permission.

Fast R-CNN



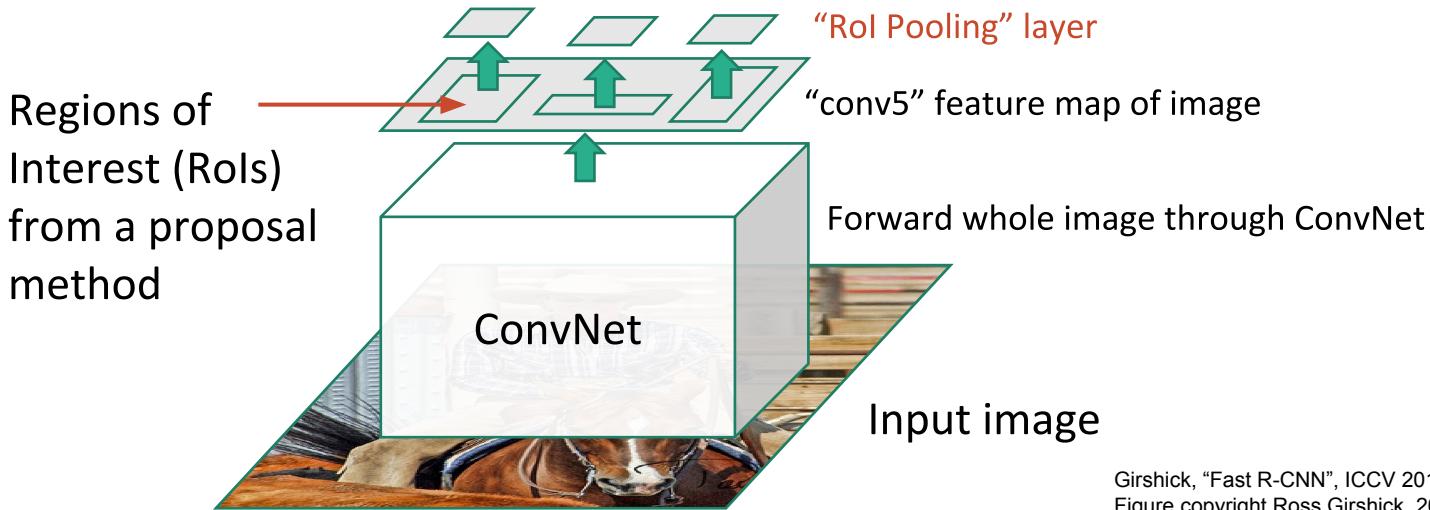
Girshick, "Fast R-CNN", ICCV 2015.
Figure copyright Ross Girshick, 2015; [source](#). Reproduced with permission.

Fast R-CNN



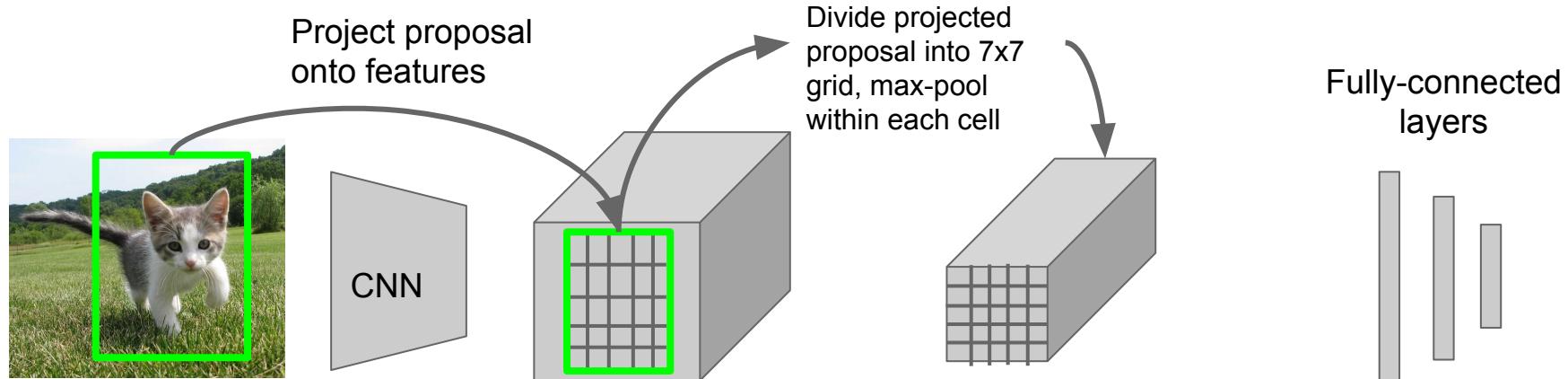
Girshick, "Fast R-CNN", ICCV 2015.
Figure copyright Ross Girshick, 2015; [source](#). Reproduced with permission.

Fast R-CNN



Girshick, "Fast R-CNN", ICCV 2015.
Figure copyright Ross Girshick, 2015; [source](#). Reproduced with permission.

Fast R-CNN: RoI Pooling



Hi-res input image:
 $3 \times 640 \times 480$
with region
proposal

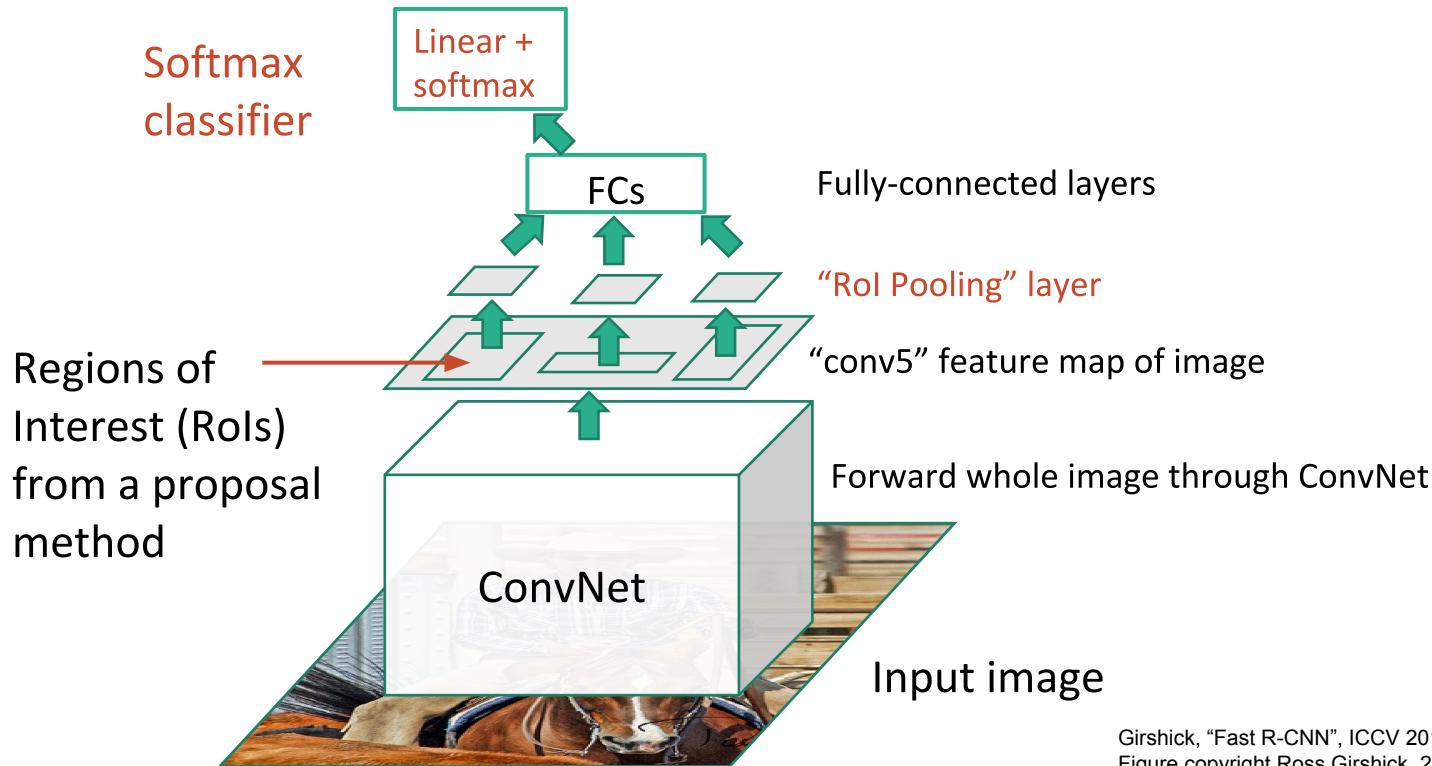
Hi-res conv features:
 $512 \times 20 \times 15$;
Projected region
proposal is e.g.
 $512 \times 18 \times 8$
(varies per proposal)

RoI conv features:
 $512 \times 7 \times 7$
for region proposal

Fully-connected layers expect
low-res conv features:
 $512 \times 7 \times 7$

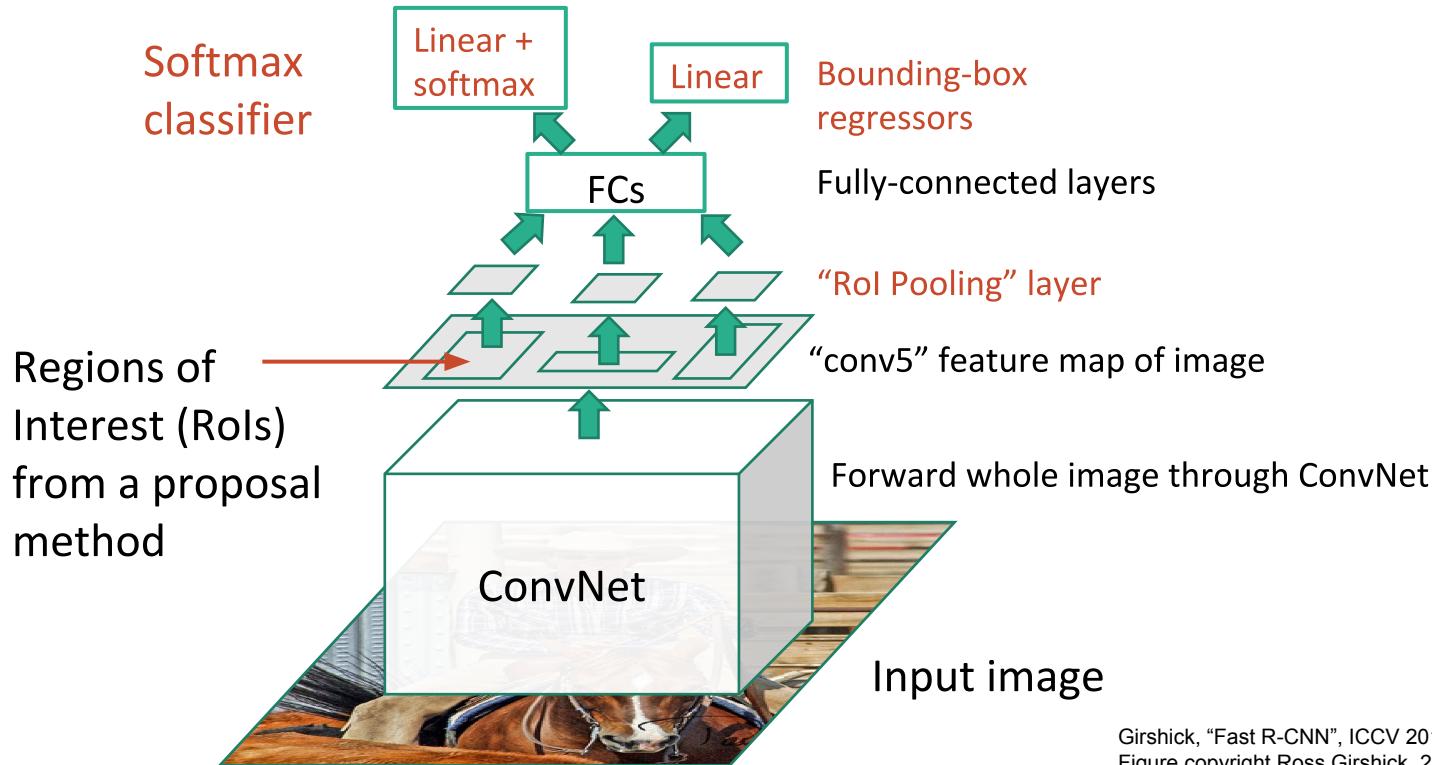
Girshick, "Fast R-CNN", ICCV 2015.

Fast R-CNN



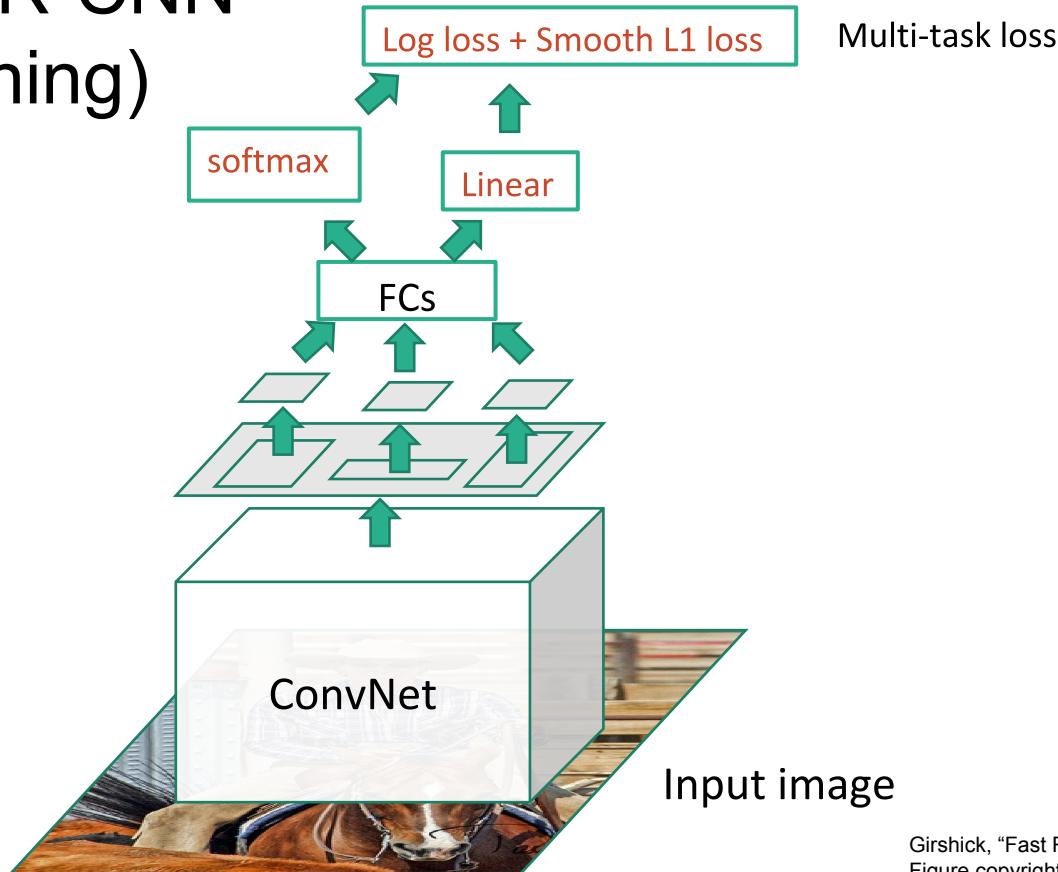
Girshick, "Fast R-CNN", ICCV 2015.
Figure copyright Ross Girshick, 2015; [source](#). Reproduced with permission.

Fast R-CNN



Girshick, "Fast R-CNN", ICCV 2015.
Figure copyright Ross Girshick, 2015; [source](#). Reproduced with permission.

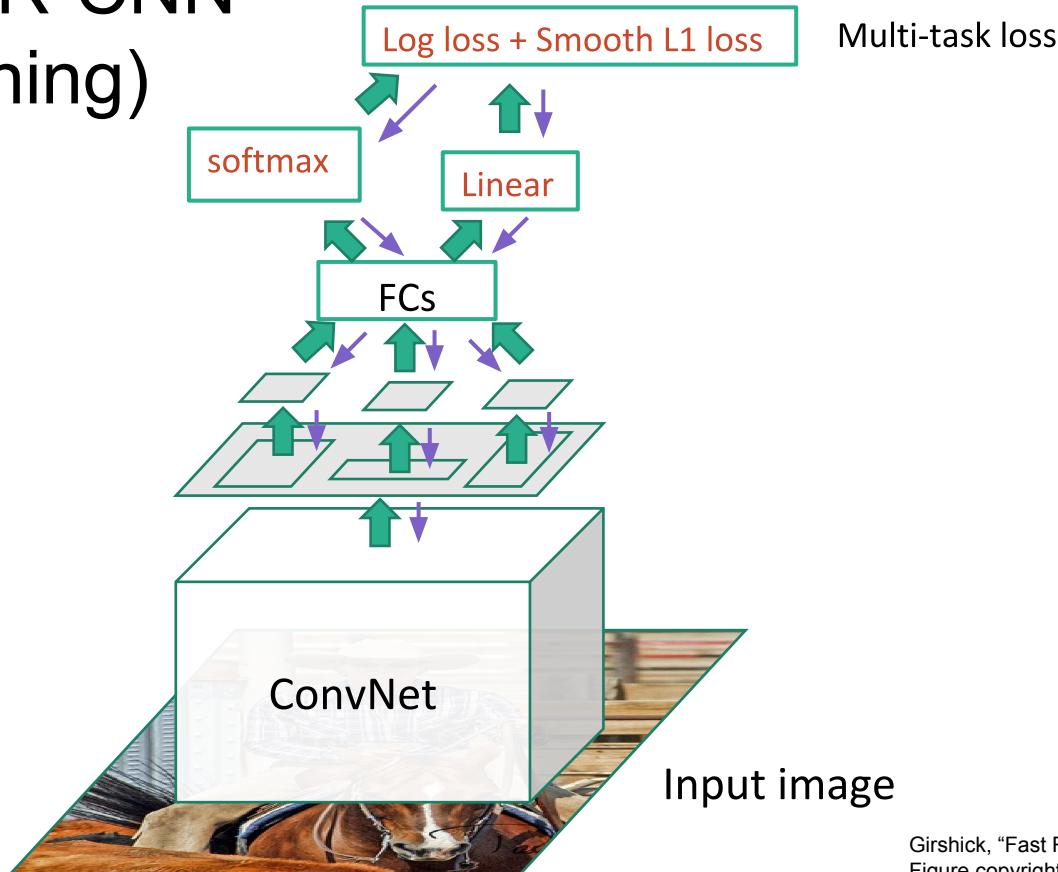
Fast R-CNN (Training)



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

Figure copyright Ross Girshick, 2015; [source](#). Reproduced with permission.

Fast R-CNN (Training)

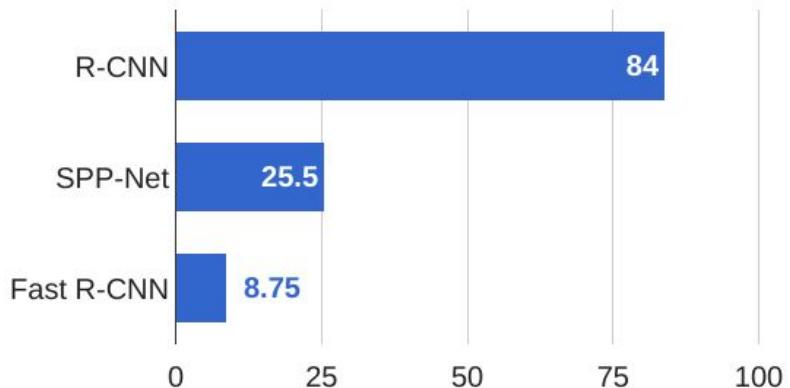


Girshick, "Fast R-CNN", ICCV 2015.

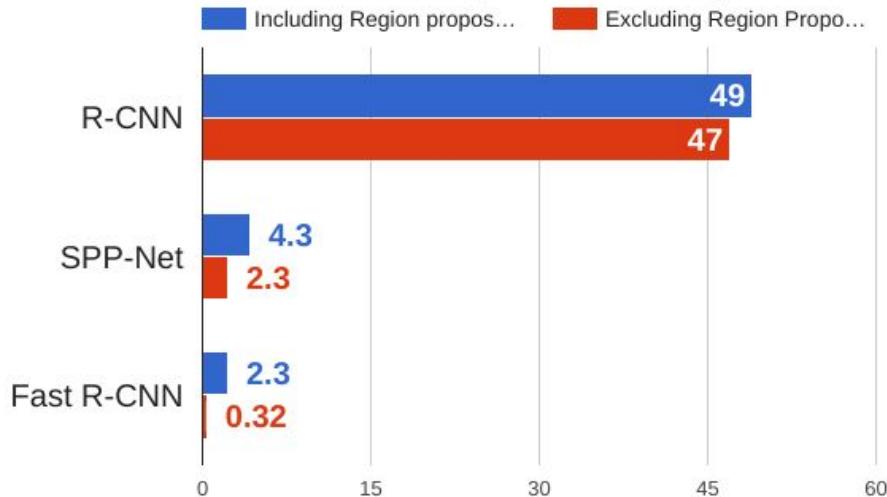
Figure copyright Ross Girshick, 2015; [source](#). Reproduced with permission.

R-CNN vs SPP vs Fast R-CNN

Training time (Hours)



Test time (seconds)



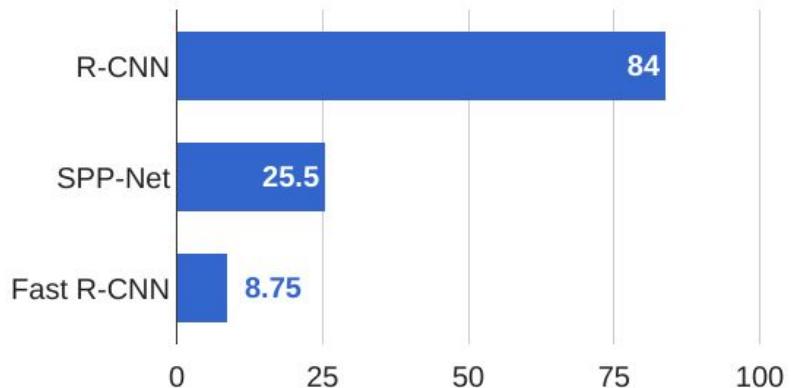
Girshick et al, "Rich feature hierarchies for accurate object detection and semantic segmentation", CVPR 2014.

He et al, "Spatial pyramid pooling in deep convolutional networks for visual recognition", ECCV 2014

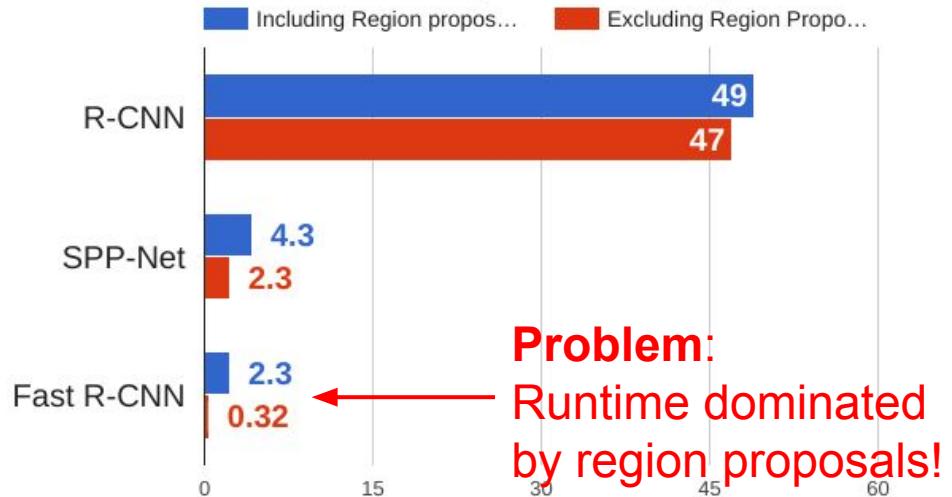
Girshick, "Fast R-CNN", ICCV 2015

R-CNN vs SPP vs Fast R-CNN

Training time (Hours)



Test time (seconds)



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

He et al, "Spatial pyramid pooling in deep convolutional networks for visual recognition", ECCV 2014

Girshick, "Fast R-CNN", ICCV 2015

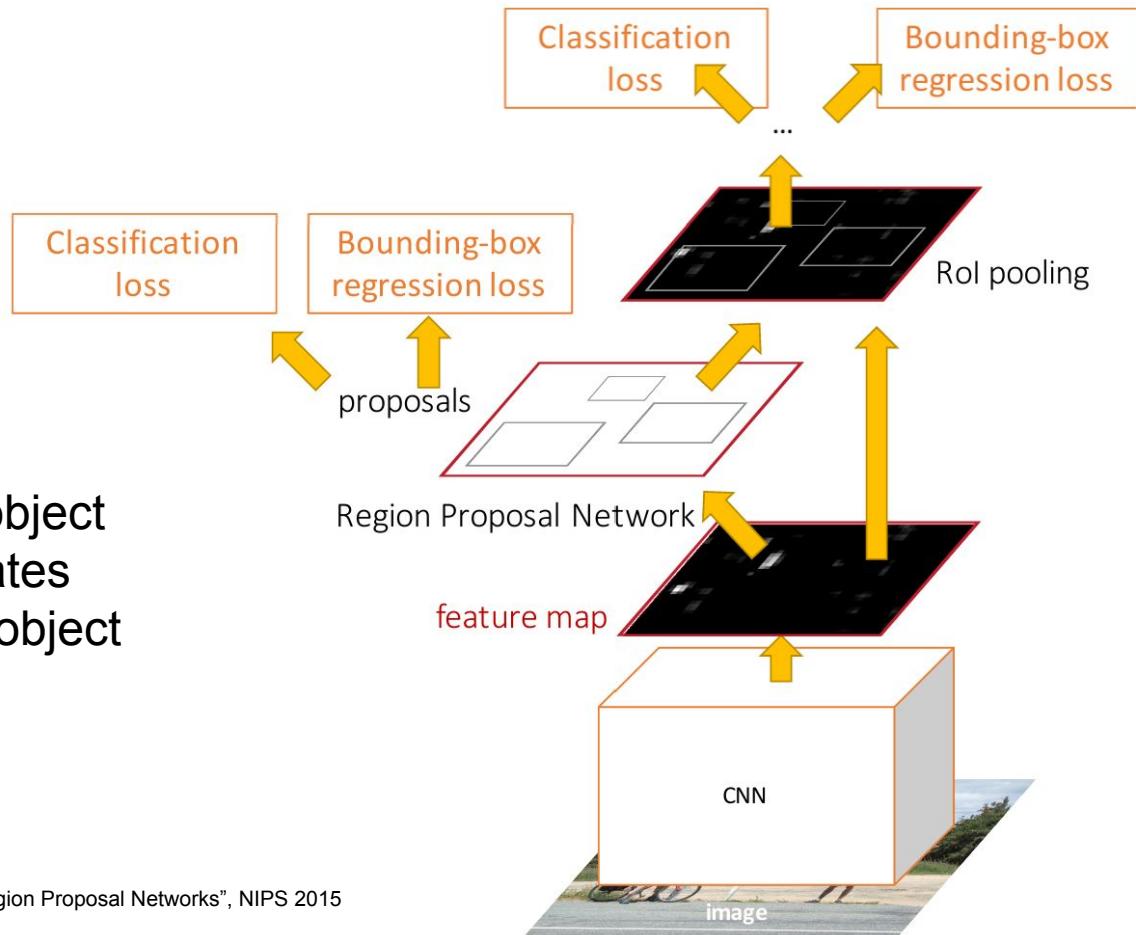
Faster R-CNN:

Make CNN do proposals!

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

Jointly train with 4 losses:

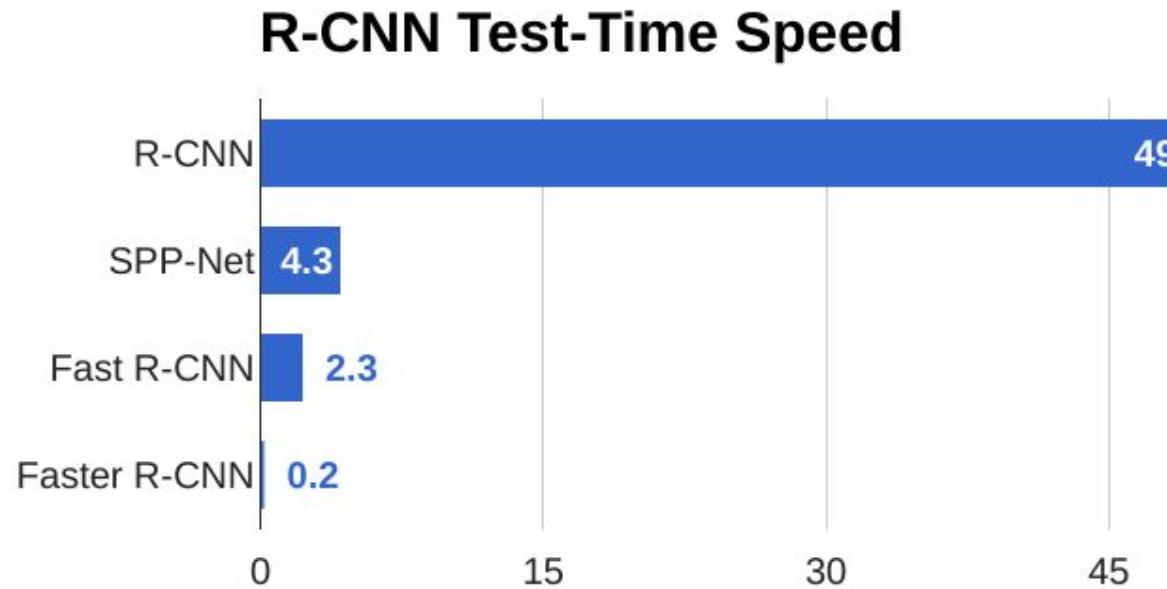
1. RPN classify object / not object
2. RPN regress box coordinates
3. Final classification score (object classes)
4. Final box coordinates



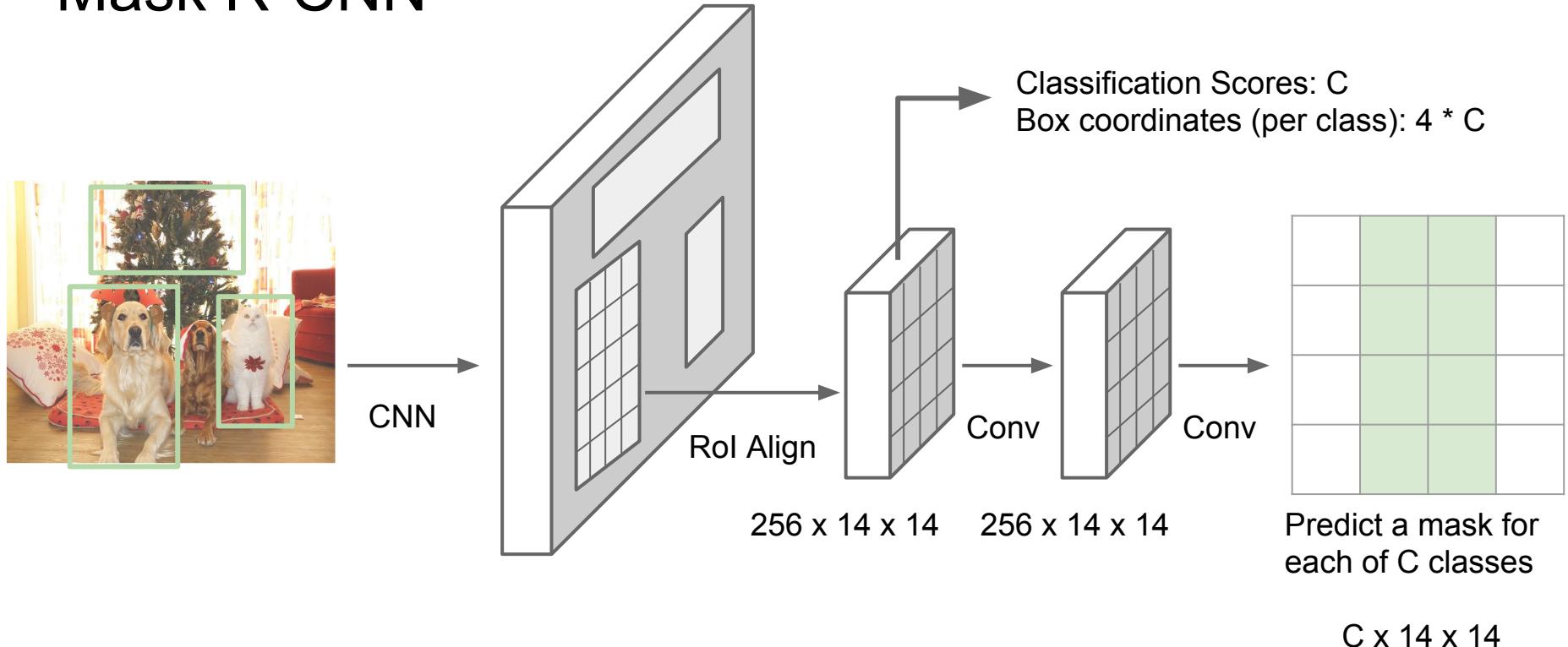
Ren et al, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", NIPS 2015
Figure copyright 2015, Ross Girshick; reproduced with permission

Faster R-CNN:

Make CNN do proposals!

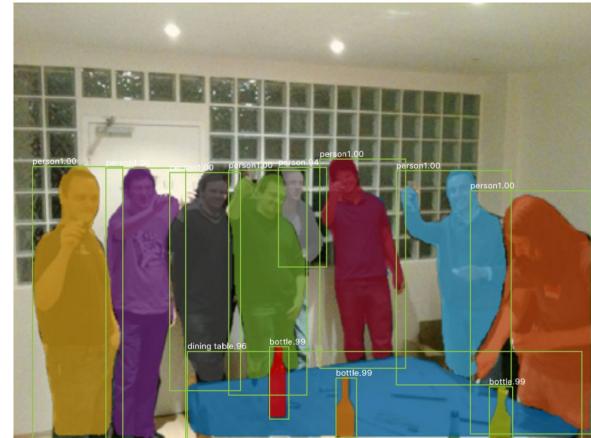
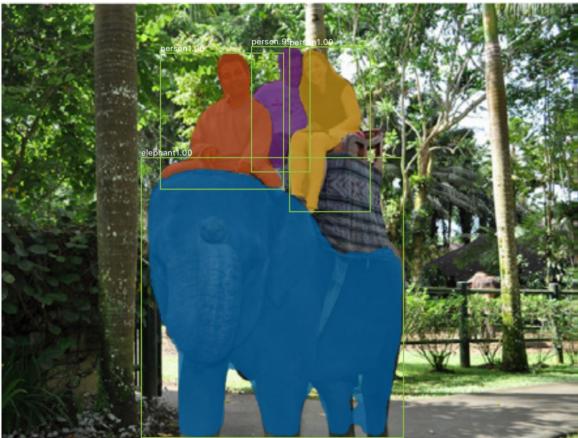


Mask R-CNN



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

Mask R-CNN: Very Good Results!



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

Figures copyright Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick, 2017.
Reproduced with permission.

Mask R-CNN

Also does pose



He et al., "Mask R-CNN", arXiv 2017

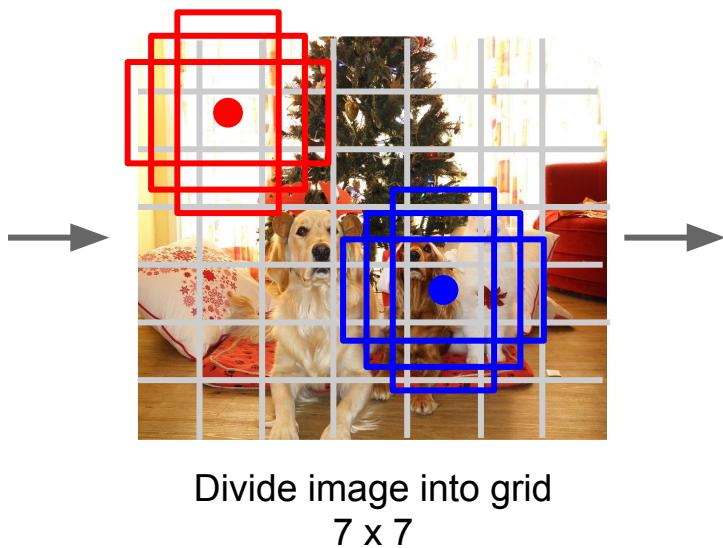
Figures copyright Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick, 2017.

Reproduced with permission.

Detection without Proposals: YOLO / SSD



Input image
 $3 \times H \times W$



Divide image into grid
 7×7

Image a set of **base boxes**
centered at each grid cell
Here $B = 3$

- Within each grid cell:
- Regress from each of the B base boxes to a final box with 5 numbers:
(dx , dy , dh , dw , confidence)
 - Predict scores for each of C classes (including background as a class)

Output:
 $7 \times 7 \times (5 * B + C)$

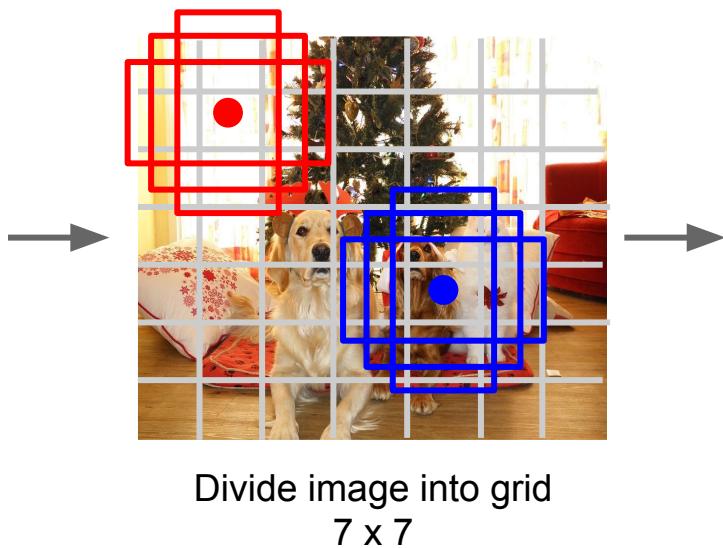
Redmon et al, "You Only Look Once:
Unified, Real-Time Object Detection", CVPR 2016
Liu et al, "SSD: Single-Shot MultiBox Detector", ECCV 2016

Detection without Proposals: YOLO / SSD

Go from input image to tensor of scores with one big convolutional network!



Input image
 $3 \times H \times W$



Divide image into grid
 7×7

Image a set of **base boxes**
centered at each grid cell
Here $B = 3$

- Within each grid cell:
- Regress from each of the B base boxes to a final box with 5 numbers:
(dx , dy , dh , dw , confidence)
 - Predict scores for each of C classes (including background as a class)

Output:
 $7 \times 7 \times (5 * B + C)$

Redmon et al, "You Only Look Once:
Unified, Real-Time Object Detection", CVPR 2016
Liu et al, "SSD: Single-Shot MultiBox Detector", ECCV 2016

Object Detection: Lots of variables ...

Base Network

VGG16
ResNet-101
Inception V2
Inception V3
Inception
ResNet
MobileNet

Object Detection architecture

Faster R-CNN
R-FCN
SSD

Image Size # Region Proposals

...

Takeaways

Faster R-CNN is slower but more accurate

SSD is much faster but not as accurate

Huang et al, "Speed/accuracy trade-offs for modern convolutional object detectors", CVPR 2017

R-FCN: Dai et al, "R-FCN: Object Detection via Region-based Fully Convolutional Networks", NIPS 2016

Inception-V2: Ioffe and Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015

Inception V3: Szegedy et al, "Rethinking the Inception Architecture for Computer Vision", arXiv 2016

Inception ResNet: Szegedy et al, "Inception-V4, Inception-ResNet and the Impact of Residual Connections on Learning", arXiv 2016

MobileNet: Howard et al, "Efficient Convolutional Neural Networks for Mobile Vision Applications", arXiv 2017

Object Detection: Impact of Deep Learning

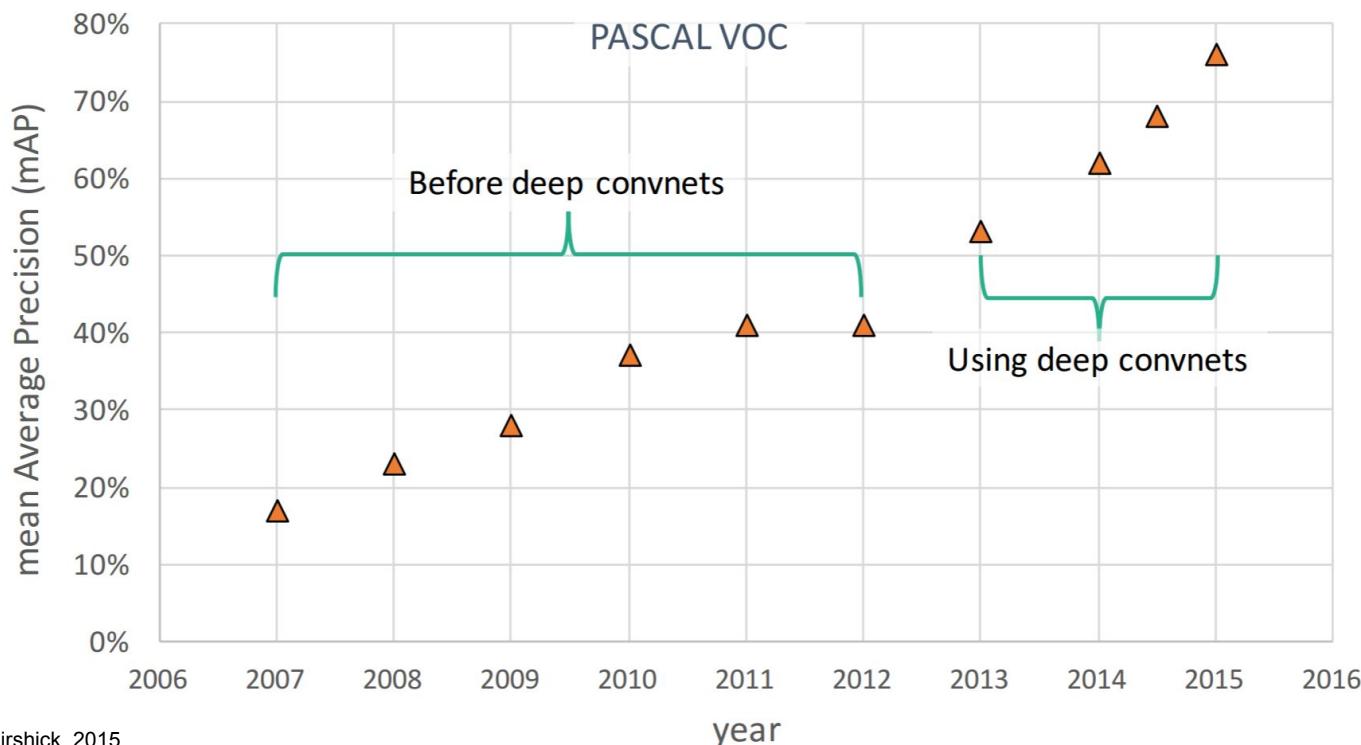


Figure copyright Ross Girshick, 2015.
Reproduced with permission.

Open Source Frameworks

Lots of good implementations on GitHub!

TensorFlow Detection API:

https://github.com/tensorflow/models/tree/master/research/object_detection

Faster RCNN, SSD, RFCN, Mask R-CNN

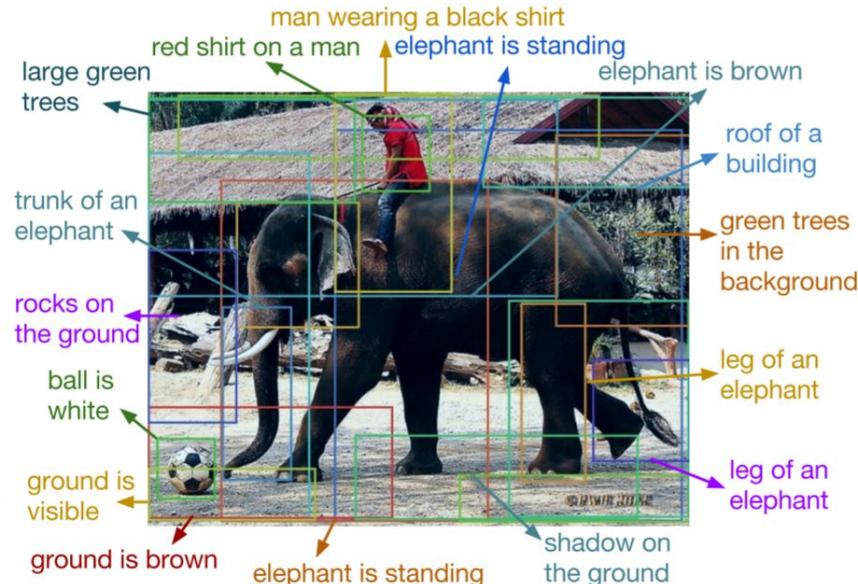
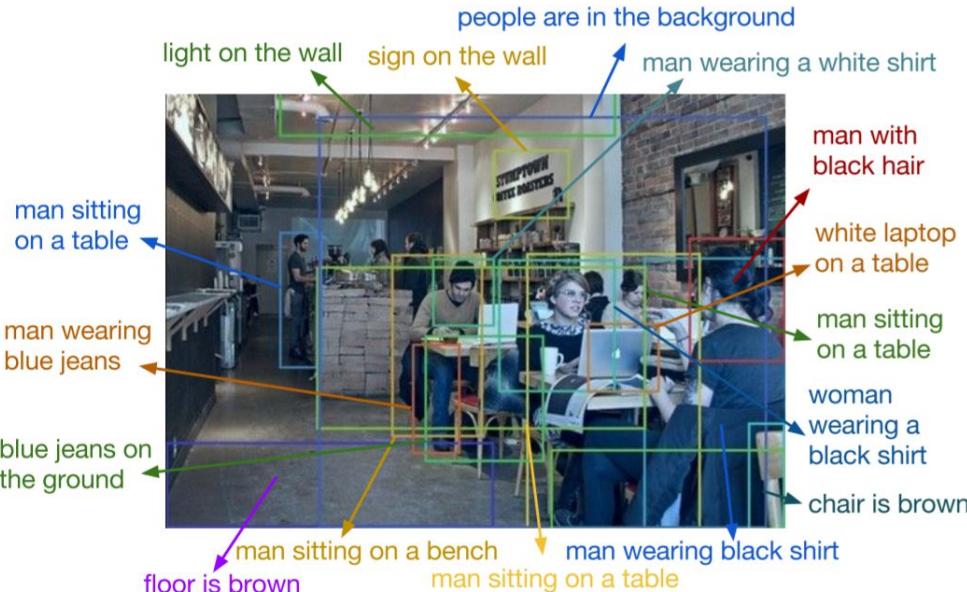
Caffe2 Detectron:

<https://github.com/facebookresearch/Detectron>

Mask R-CNN, RetinaNet, Faster R-CNN, RPN, Fast R-CNN, R-FCN

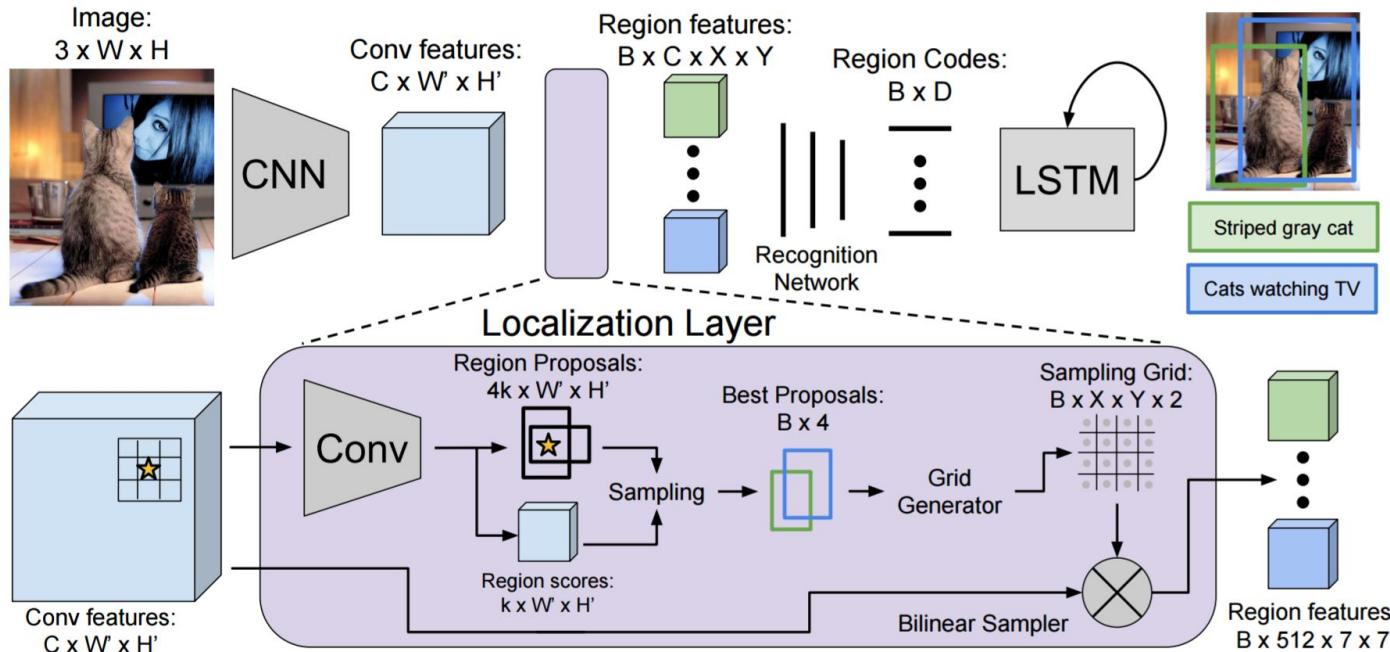
Finetune on your own dataset with pre-trained models

Aside: Object Detection + Captioning = Dense Captioning

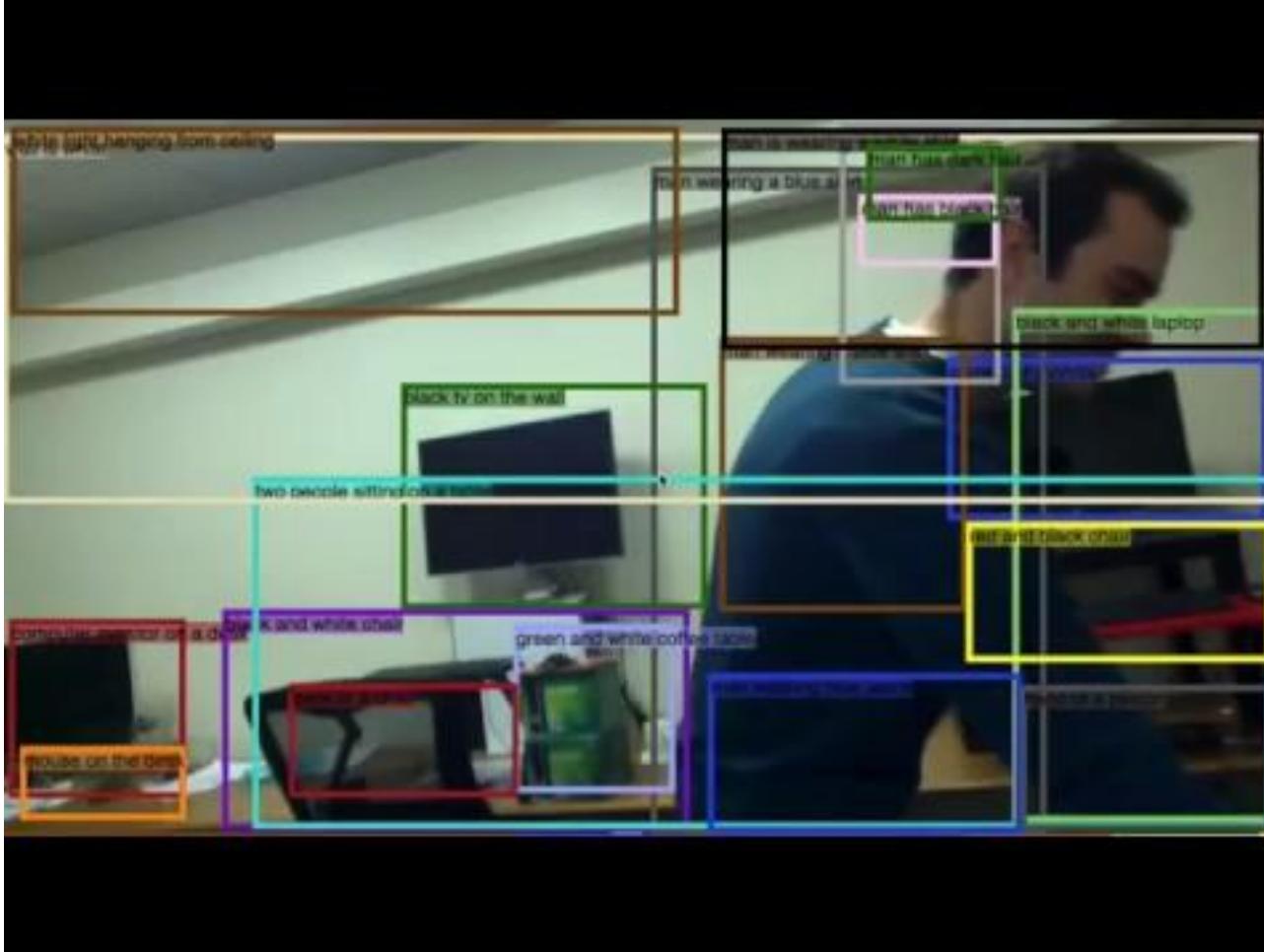


Johnson, Karpathy, and Fei-Fei, "DenseCap: Fully Convolutional Localization Networks for Dense Captioning", CVPR 2016
Figure copyright IEEE, 2016. Reproduced for educational purposes.

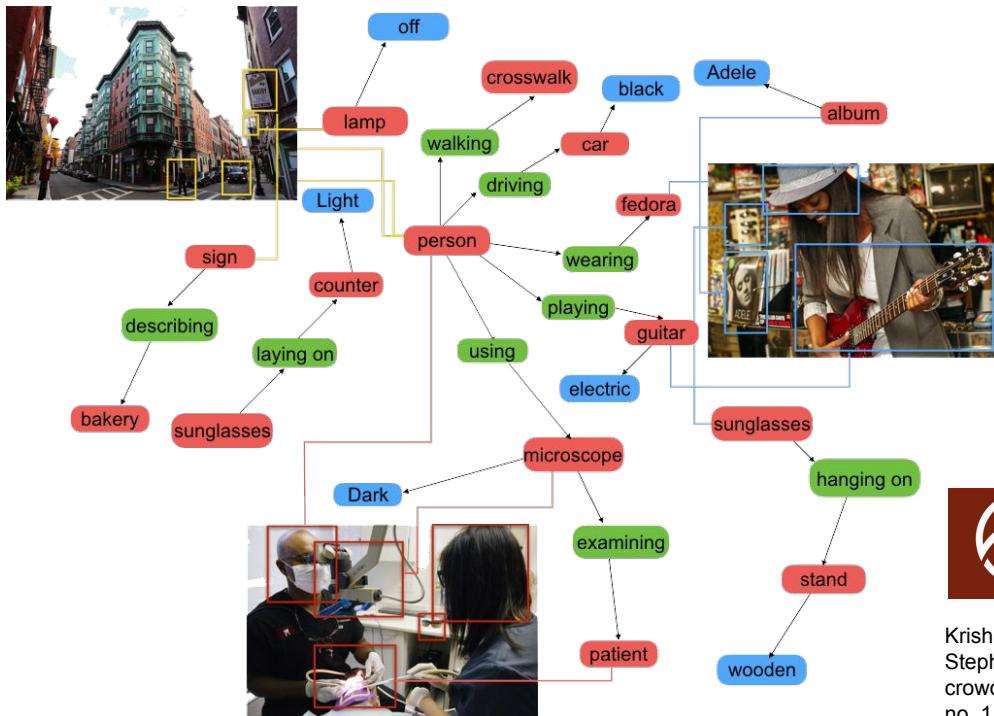
Aside: Object Detection + Captioning = Dense Captioning



Johnson, Karpathy, and Fei-Fei, "DenseCap: Fully Convolutional Localization Networks for Dense Captioning", CVPR 2016
Figure copyright IEEE, 2016. Reproduced for educational purposes.



Aside: Visual Genome



108,077 Images

5.4 Million Region Descriptions

1.7 Million Visual Question Answers

3.8 Million Object Instances

2.8 Million Attributes

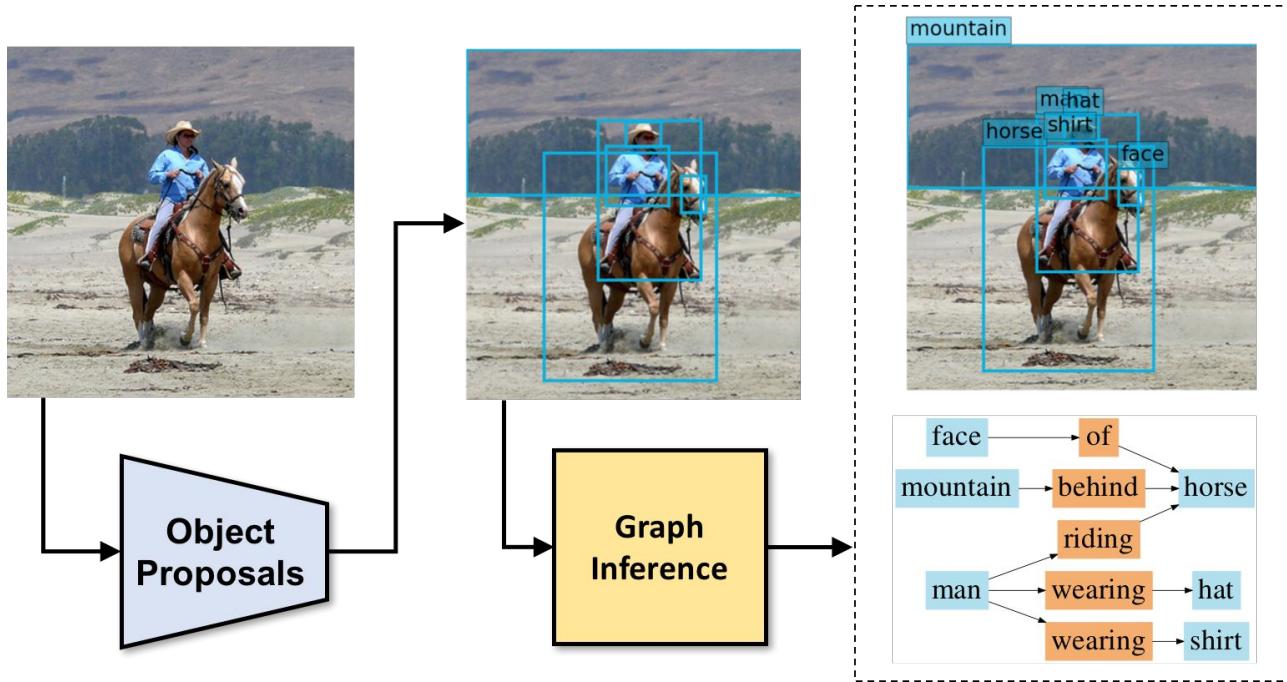
2.3 Million Relationships

Everything Mapped to Wordnet Synsets

 **VISUAL GENOME**

Krishna, Ranjay, Yuke Zhu, Oliver Groth, Justin Johnson, Kenji Hata, Joshua Kravitz, Stephanie Chen et al. "Visual genome: Connecting language and vision using crowdsourced dense image annotations." *International Journal of Computer Vision* 123, no. 1 (2017): 32-73.

Aside: Scene Graph Generation



Xu, Zhu, Choy, and Fei-Fei, "Scene Graph Generation by Iterative Message Passing", CVPR 2017
Figure copyright IEEE, 2018. Reproduced for educational purposes.

3D Object Detection

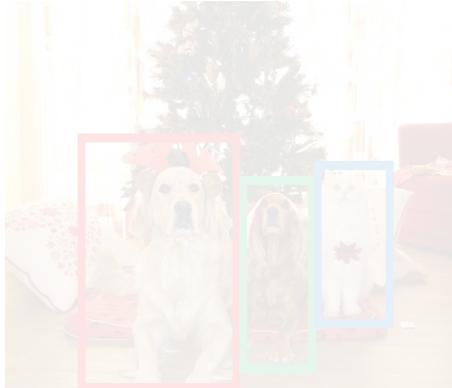
Semantic Segmentation



GRASS, CAT,
TREE, SKY

No objects, just pixels

2D Object Detection



DOG, DOG, CAT

Object categories +
2D bounding boxes

3D Object Detection



Car

Object categories +
3D bounding boxes

This image is CC0 public domain

Simplified Camera Model

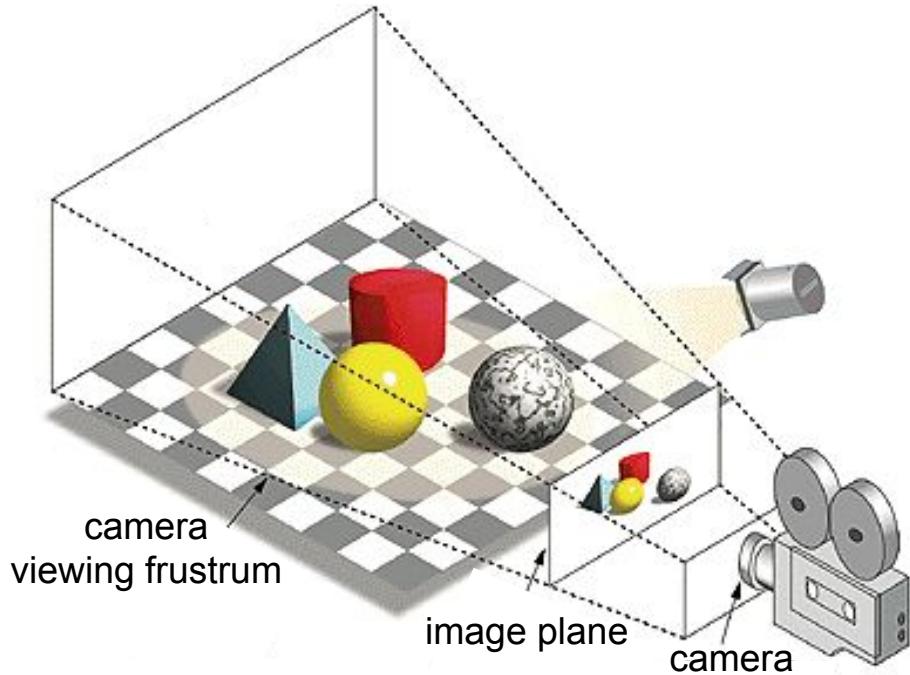
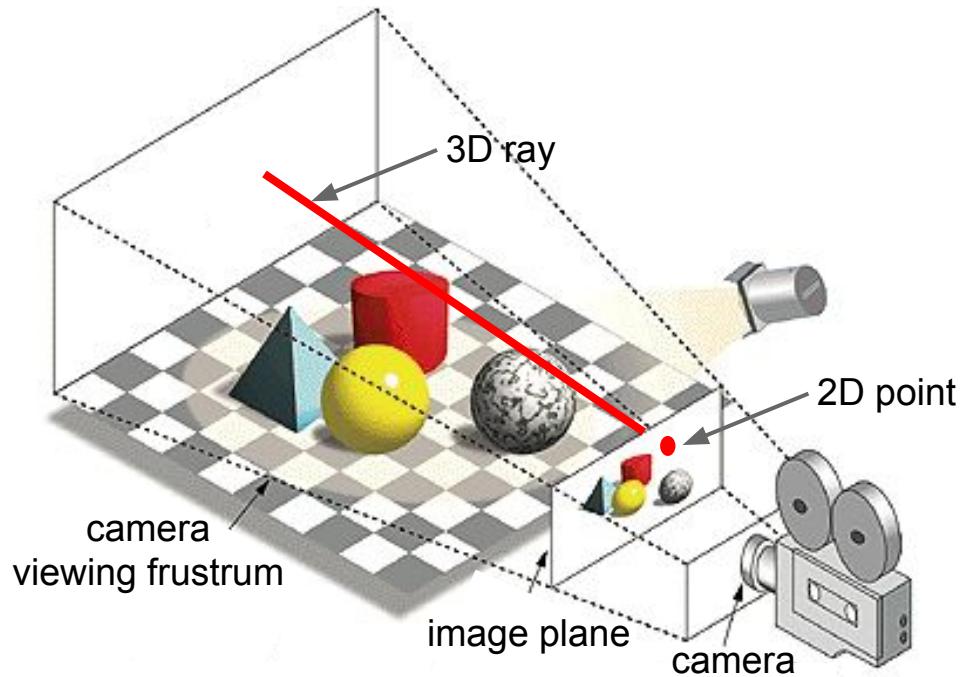


Image source: https://www.pcmag.com/encyclopedia_images/_FRUSTUM.GIF

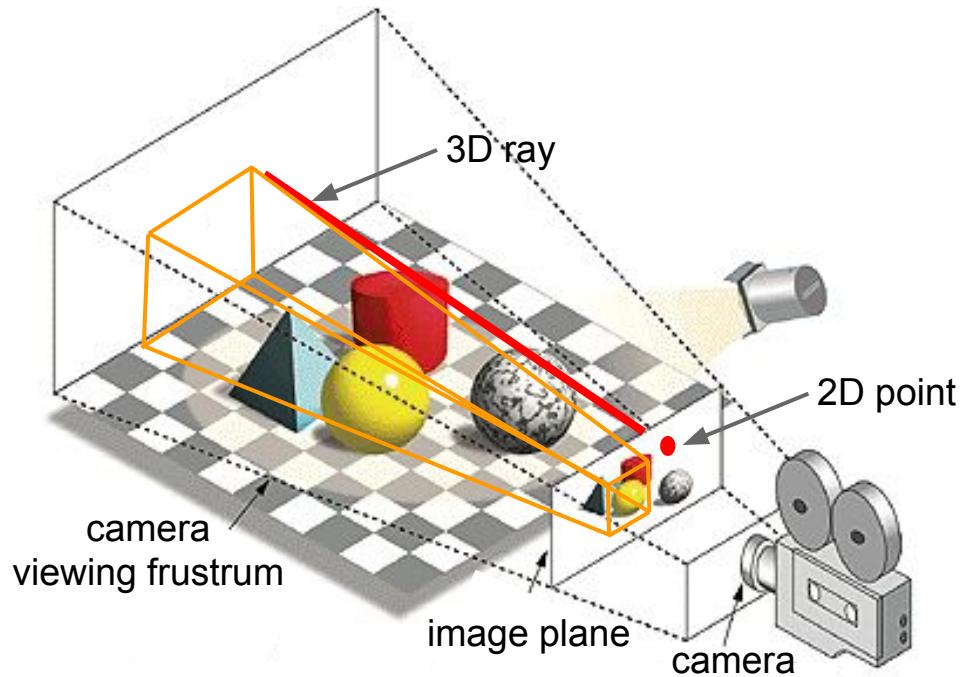
Simplified Camera Model



A point on the image plane corresponds to a **ray** in the 3D space

Image source: https://www.pcmag.com/encyclopedia_images/_FRUSTUM.GIF

Simplified Camera Model

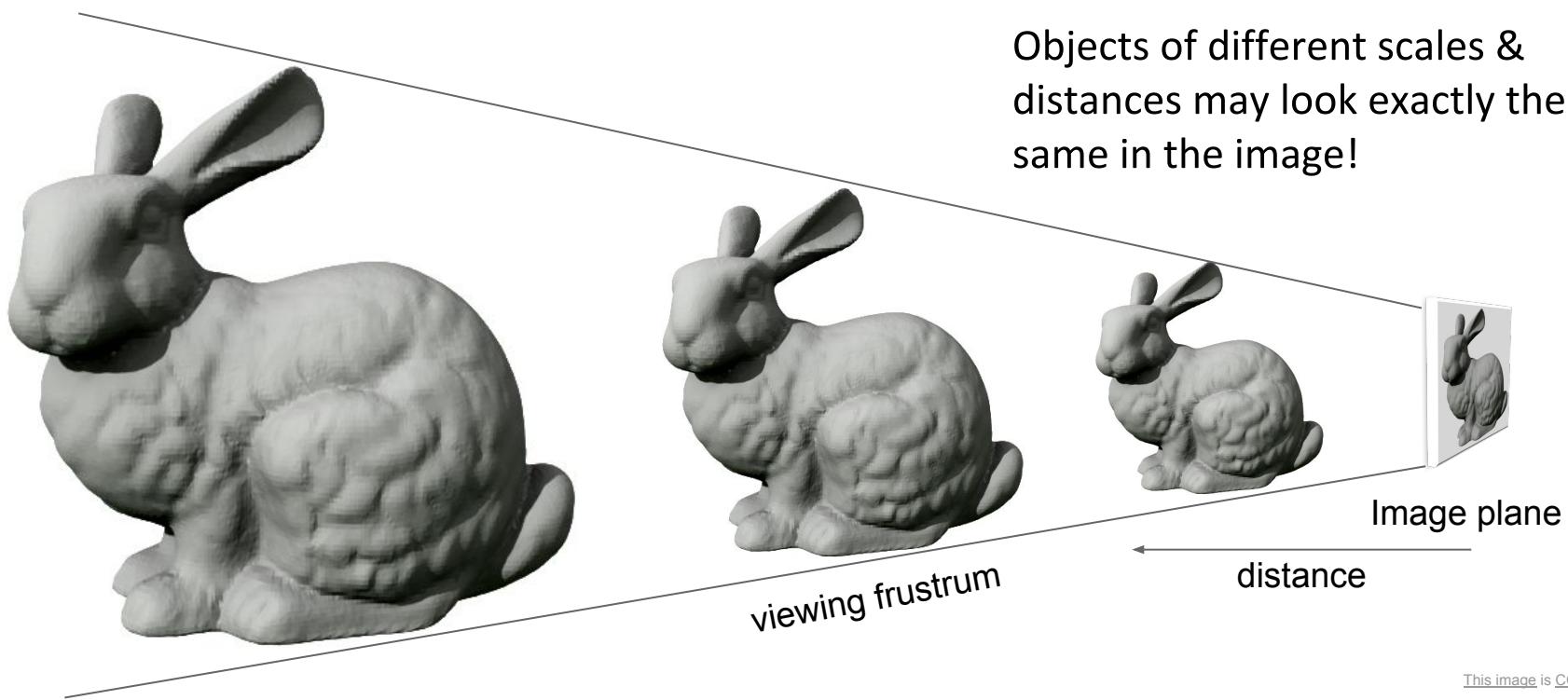


A point on the image plane corresponds to a **ray** in the 3D space

A 2D bounding box on an image is a **frustum** in the 3D space

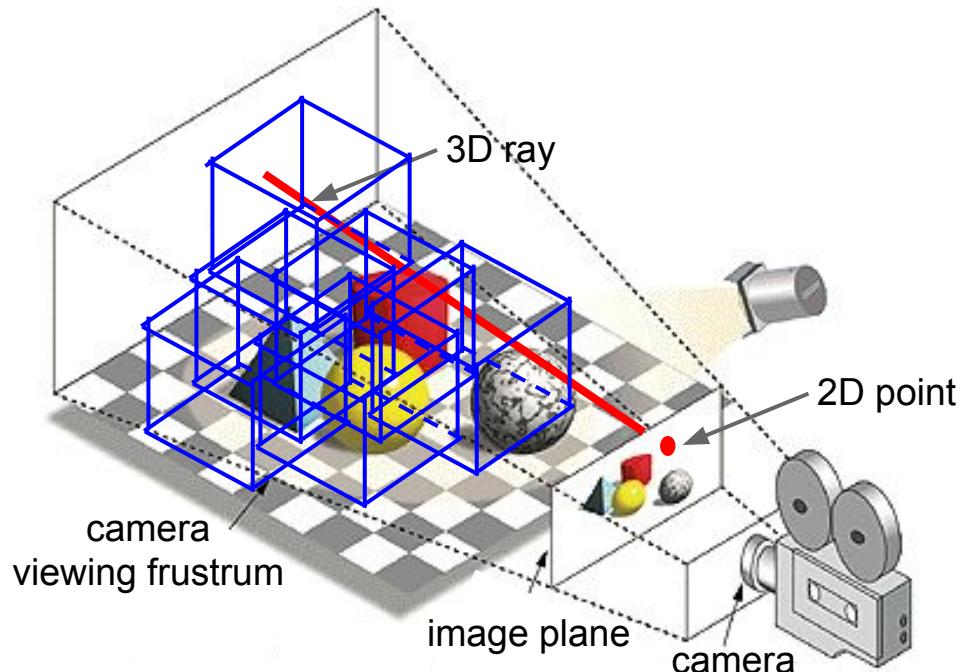
Image source: https://www.pcmag.com/encyclopedia_images/_FRUSTUM.GIF

Scale & Distance Ambiguity



This image is CC0 public domain

Simplified Camera Model



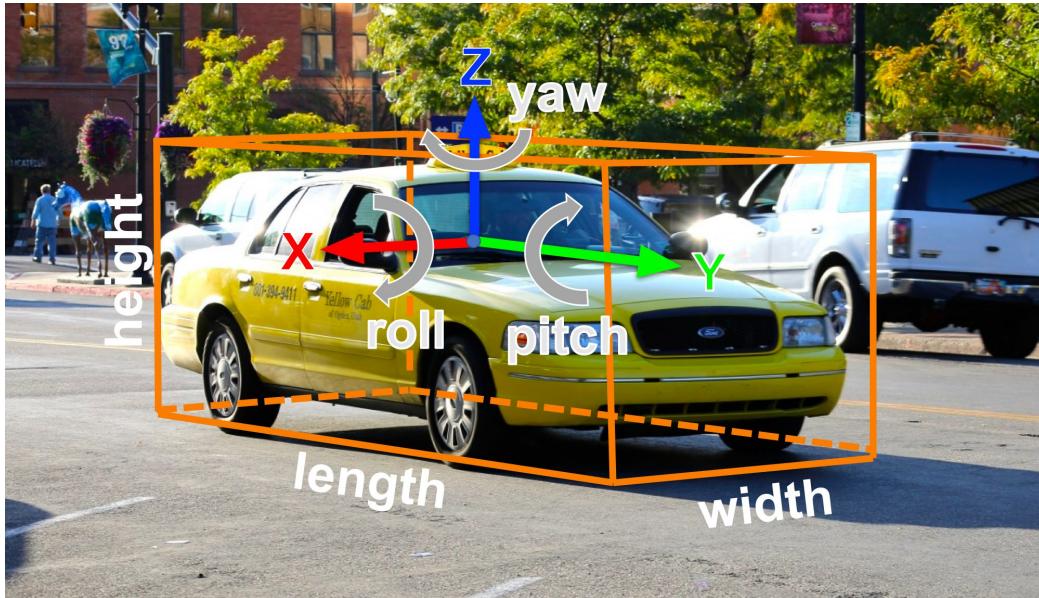
A point on the image plane corresponds to a **ray** in the 3D space

A 2D bounding box on an image is a **frustrum** in the 3D space

Localize an object in 3D:
The object can be anywhere in the **camera viewing frustum!**

Image source: https://www.pcmag.com/encyclopedia_images/_FRUSTUM.GIF

3D Object Detection



2D Object Detection:

2D bounding box

(x, y, w, h)

3D Object Detection:

3D oriented bounding box

$(x, y, z, w, h, l, r, p, y)$

Simplified bbox: no roll & pitch

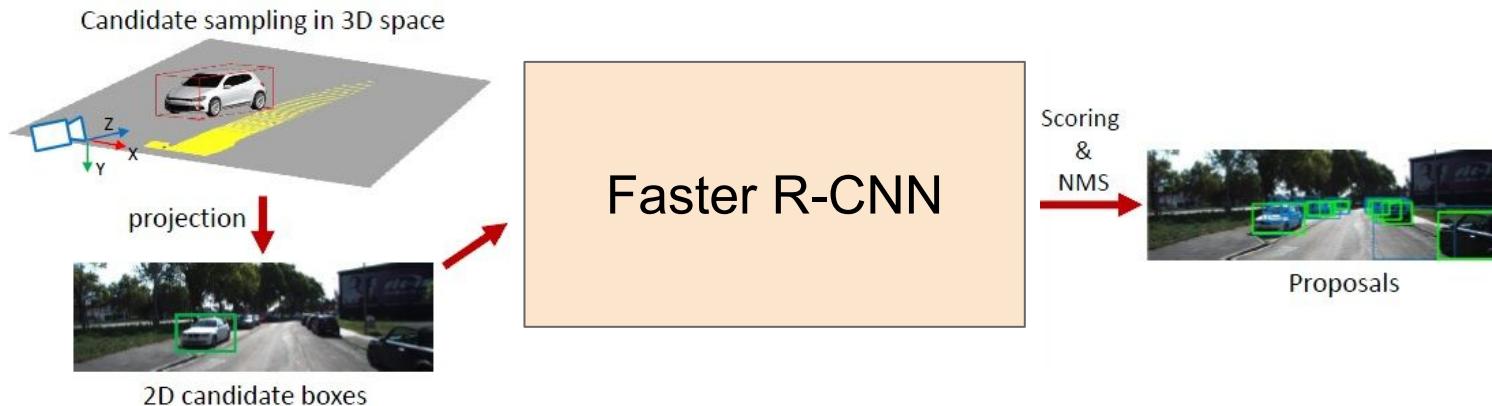
Much harder problem than 2D object detection!

This image is CC0 public domain

3D Object Detection



3D Object Detection: Monocular Camera



- Same idea as Faster RCNN, but proposals are in 3D
- 3D bounding box proposal, regress 3D box parameters + class score

Chen, Xiaozhi, Kaustav Kundu, Ziyu Zhang, Huimin Ma, Sanja Fidler, and Raquel Urtasun. "Monocular 3d object detection for autonomous driving." CVPR 2016.

3D Object Detection: Monocular View



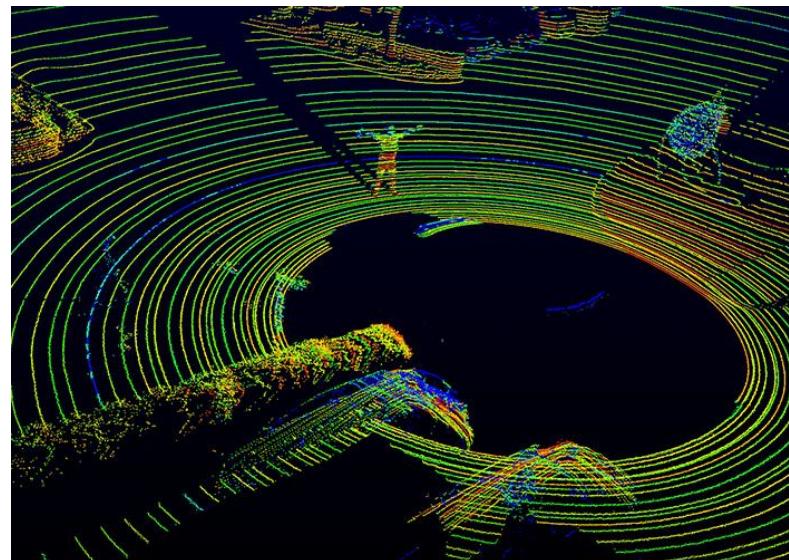
Chen, Xiaozhi, Kaustav Kundu, Ziyu Zhang, Huimin Ma, Sanja Fidler, and Raquel Urtasun. "Monocular 3d object detection for autonomous driving." CVPR 2016.

3D Object Detection: Camera + LiDAR

This image is CC0 public domain

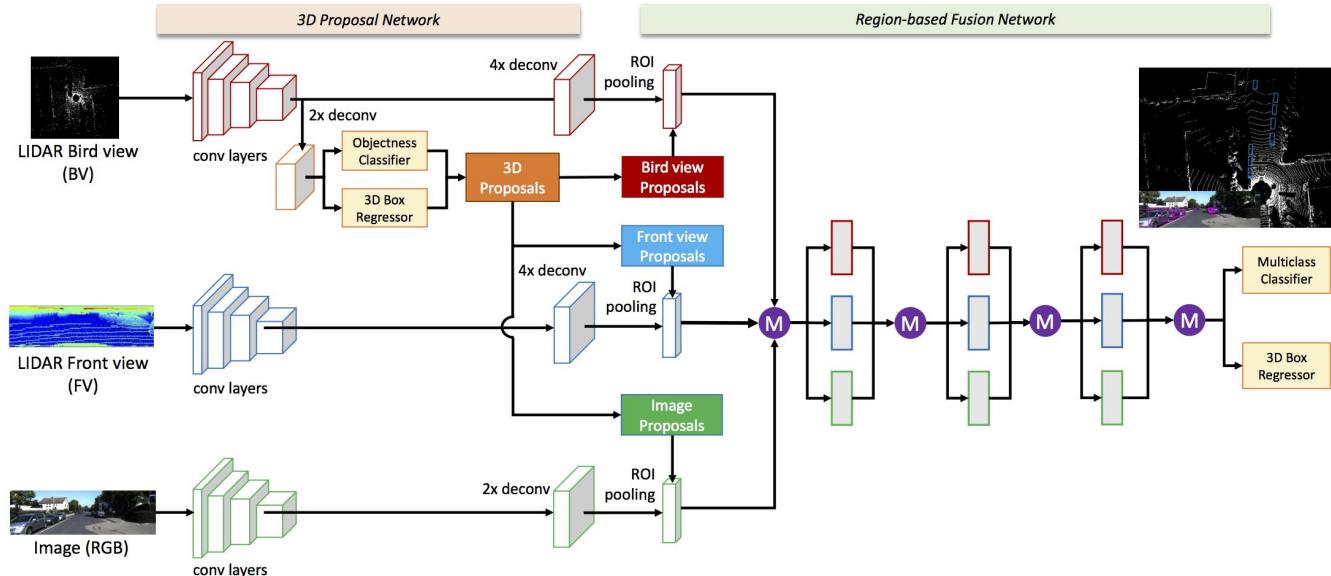


Velodyne (HDL-64e)



3D Point Cloud

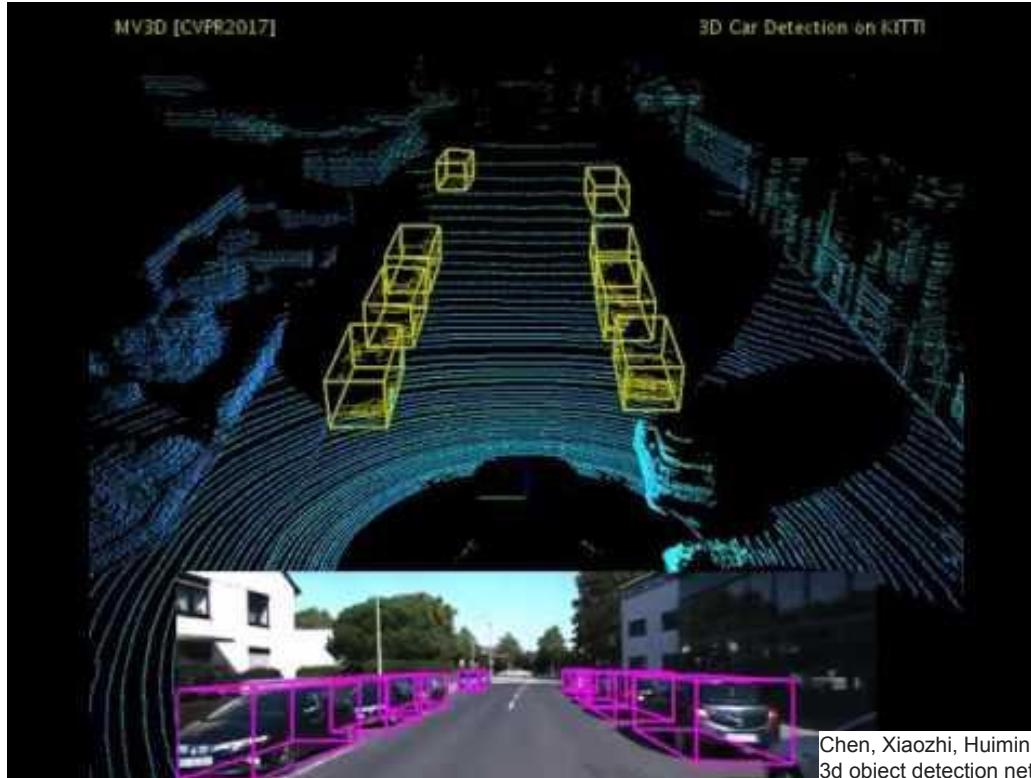
3D Object Detection: Camera + LiDAR



- Combine 3D proposals from multiple views & sensors
- regress 3D box parameters + class score

Chen, Xiaozhi, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. "Multi-view 3d object detection network for autonomous driving." CVPR 2017

3D Object Detection: Camera + LiDAR



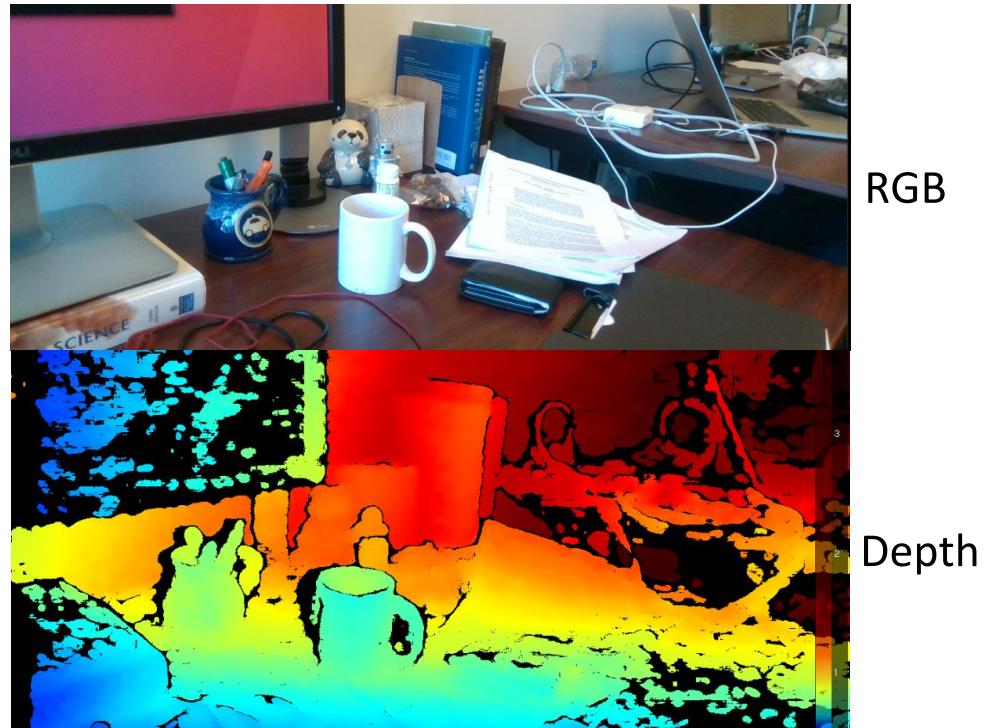
Chen, Xiaozhi, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. "Multi-view 3d object detection network for autonomous driving." CVPR 2017

RGB-Depth Camera

This image is CC0 public domain



Kinect (Xbox One)



RGB

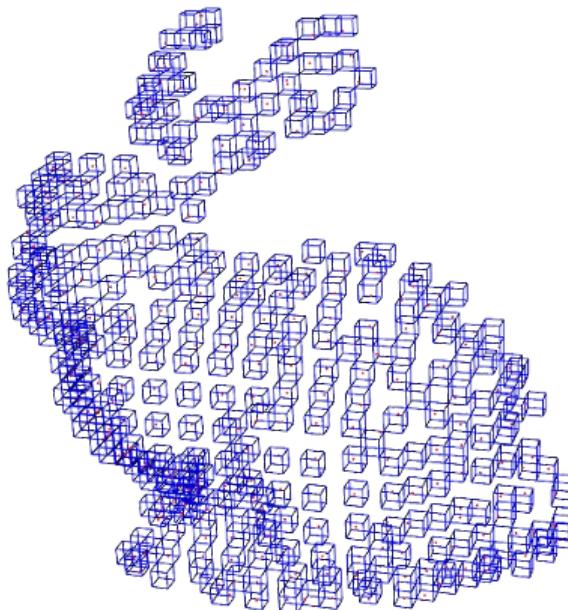
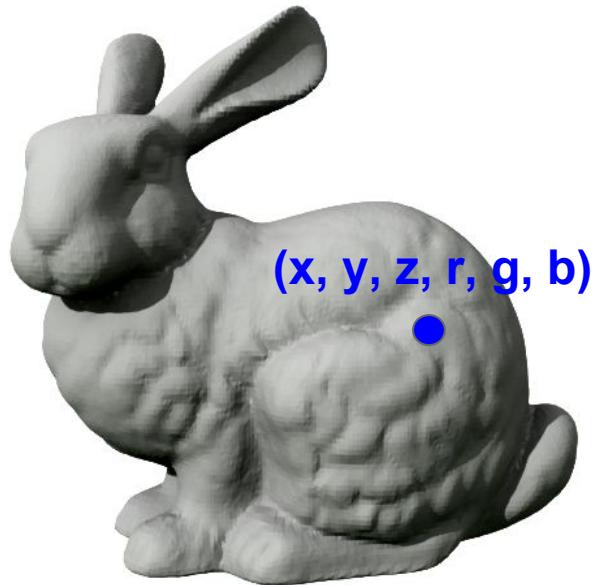
Depth

RGB-Depth Camera



Registered RGB + depth point cloud

Point Cloud Voxelization



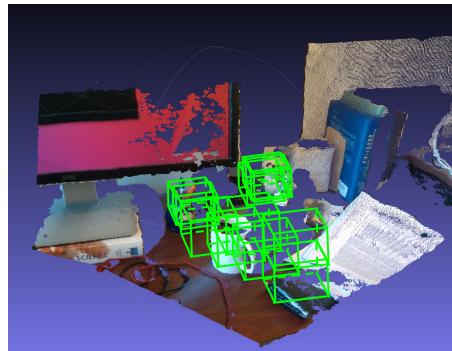
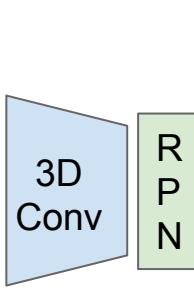
1. Capture RGB-D point cloud of a scene.
2. Partition the 3D space into a regular 3D grid.
3. For each grid cell that has a point fall into it, fill the cell with the RGB value of that point.

A bit like “3D image”

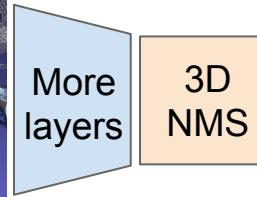
3D Object Detection: RGB-Depth Camera



Voxelized RGB-D point cloud



3D region proposals



Object categories +
3D bounding boxes

“Faster RCNN in 3D”

S. Song, and J. Xiao. Deep Sliding Shapes for Amodal 3D Object Detection in RGB-D Images. CVPR 2016

Recap

Semantic Segmentation



**GRASS, CAT,
TREE, SKY**

No objects, just pixels

2D Object Detection



DOG, DOG, CAT

Object categories +
2D bounding boxes

3D Object Detection



Car

Object categories +
3D bounding boxes

This image is CC0 public domain

Next time:
Visualizing CNN features
DeepDream + Style Transfer

Lecture 12: Generative Models

Administrative

Project Milestone due tomorrow (Wed 5/16)

A3 due next Wed, 5/23

Overview

- Unsupervised Learning
- Generative Models
 - PixelRNN and PixelCNN
 - Variational Autoencoders (VAE)
 - Generative Adversarial Networks (GAN)

Supervised vs Unsupervised Learning

Supervised Learning

Data: (x, y)

x is data, y is label

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

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

Supervised vs Unsupervised Learning

Supervised Learning

Data: (x, y)

x is data, y is label

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

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



→ Cat

Classification

This image is CC0 public domain

Supervised vs Unsupervised Learning

Supervised Learning

Data: (x, y)

x is data, y is label

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

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



DOG, DOG, CAT

Object Detection

This image is CC0 public domain

Supervised vs Unsupervised Learning

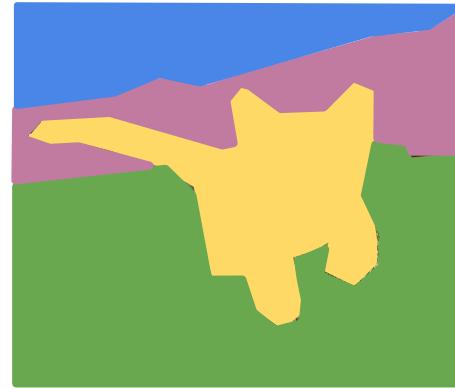
Supervised Learning

Data: (x, y)

x is data, y is label

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

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



GRASS, CAT,
TREE, SKY

Semantic Segmentation

Supervised vs Unsupervised Learning

Supervised Learning

Data: (x, y)

x is data, y is label

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

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



A cat sitting on a suitcase on the floor

Image captioning

Caption generated using [neuraltalk2](#).
[Image is CC0 Public domain](#).

Supervised vs Unsupervised Learning

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.

Supervised vs Unsupervised Learning

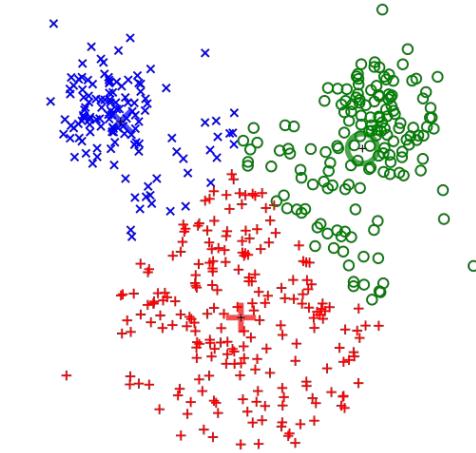
Unsupervised Learning

Data: x

Just data, no labels!

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

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



K-means clustering

This image is CC0 public domain

Supervised vs Unsupervised Learning

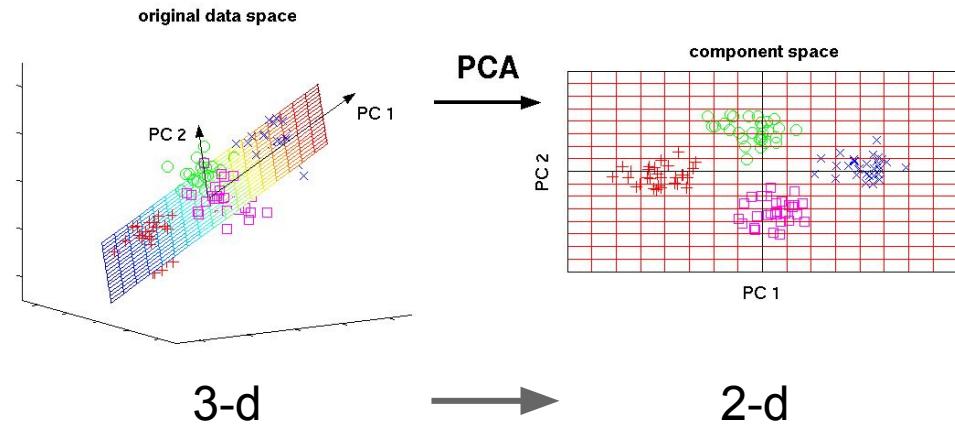
Unsupervised Learning

Data: x

Just data, no labels!

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

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



Principal Component Analysis
(Dimensionality reduction)

This image from Matthias Scholz
is CC0 public domain

Supervised vs Unsupervised Learning

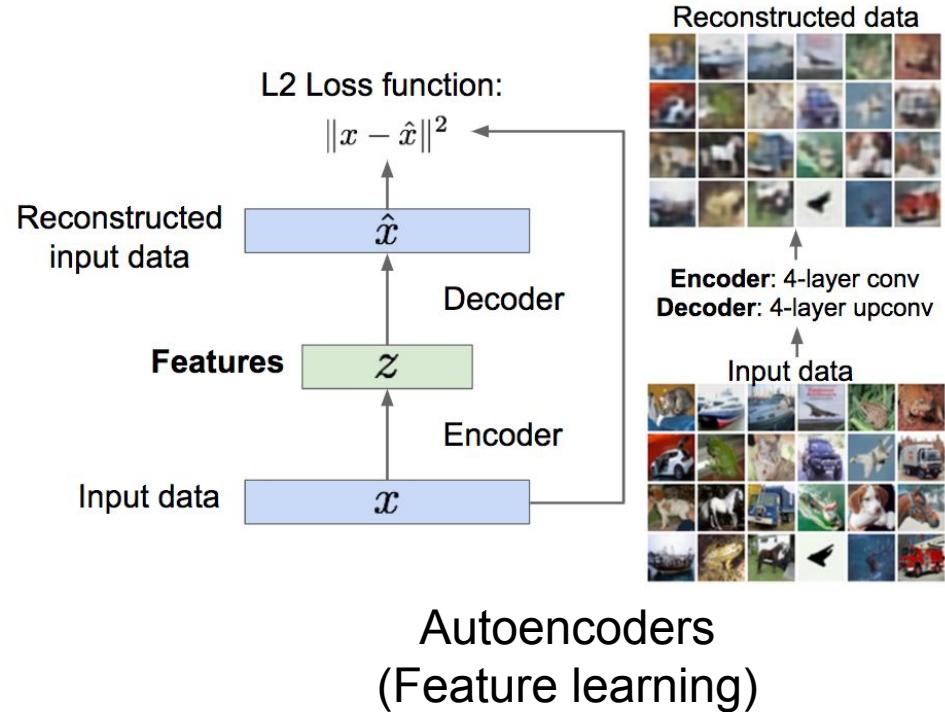
Unsupervised Learning

Data: x

Just data, no labels!

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

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



Supervised vs Unsupervised Learning

Unsupervised Learning

Data: x

Just data, no labels!

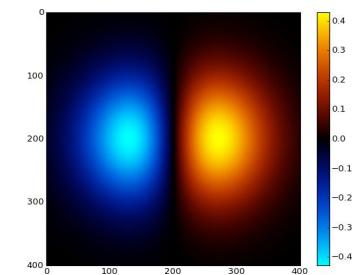
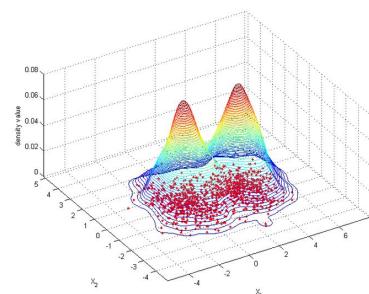
Goal: Learn some underlying hidden *structure* of the data

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



Figure copyright Ian Goodfellow, 2016. Reproduced with permission.

1-d density estimation



2-d density estimation

2-d density images [left](#) and [right](#)
are CC0 public domain

Supervised vs Unsupervised Learning

Supervised Learning

Data: (x, y)

x is data, y is label

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

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

Unsupervised Learning

Data: x

Just data, no labels!

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

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

Supervised vs Unsupervised Learning

Supervised Learning

Data: (x, y)

x is data, y is label

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

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

Unsupervised Learning

Training data is cheap

Data: x

Just data, no labels!

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

Holy grail: Solve
unsupervised learning
 \Rightarrow understand structure
of visual world

Generative Models

Given training data, generate new samples from same distribution



Training data $\sim p_{\text{data}}(x)$

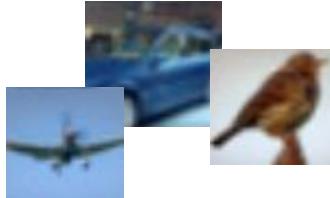


Generated samples $\sim p_{\text{model}}(x)$

Want to learn $p_{\text{model}}(x)$ similar to $p_{\text{data}}(x)$

Generative Models

Given training data, generate new samples from same distribution



Training data $\sim p_{\text{data}}(x)$



Generated samples $\sim p_{\text{model}}(x)$

Want to learn $p_{\text{model}}(x)$ similar to $p_{\text{data}}(x)$

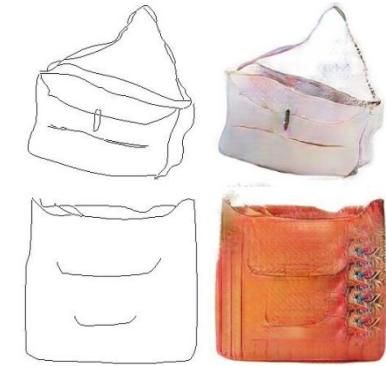
Addresses density estimation, a core problem in unsupervised learning

Several flavors:

- Explicit density estimation: explicitly define and solve for $p_{\text{model}}(x)$
- Implicit density estimation: learn model that can sample from $p_{\text{model}}(x)$ w/o explicitly defining it

Why Generative Models?

- Realistic samples for artwork, super-resolution, colorization, etc.



- Generative models of time-series data can be used for simulation and planning (reinforcement learning applications!)
- Training generative models can also enable inference of latent representations that can be useful as general features

Figures from L-R are copyright: (1) [Alec Radford et al. 2016](#); (2) [David Berthelot et al. 2017; Phillip Isola et al. 2017](#). Reproduced with authors permission.

Taxonomy of Generative Models

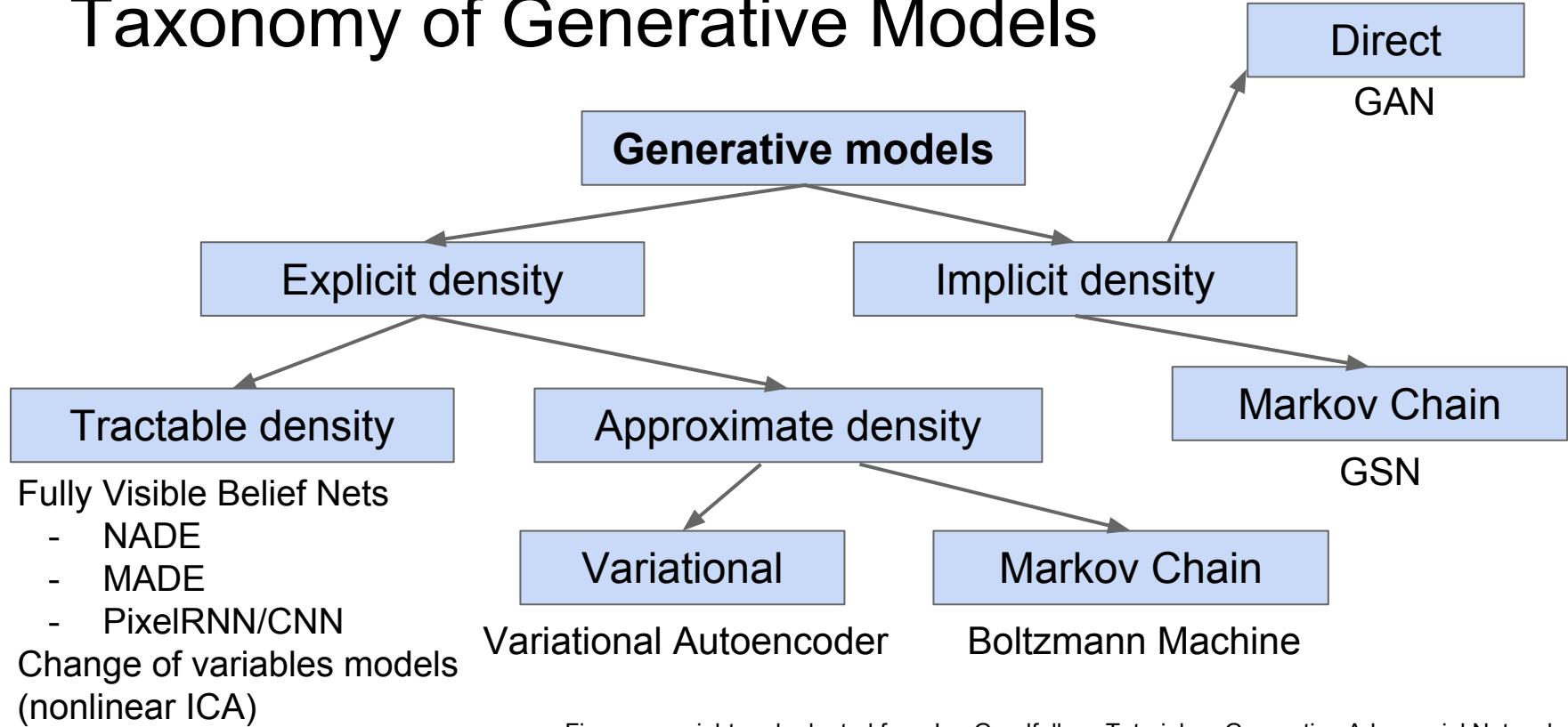


Figure copyright and adapted from Ian Goodfellow, Tutorial on Generative Adversarial Networks, 2017.

Taxonomy of Generative Models

Today: discuss 3 most popular types of generative models today

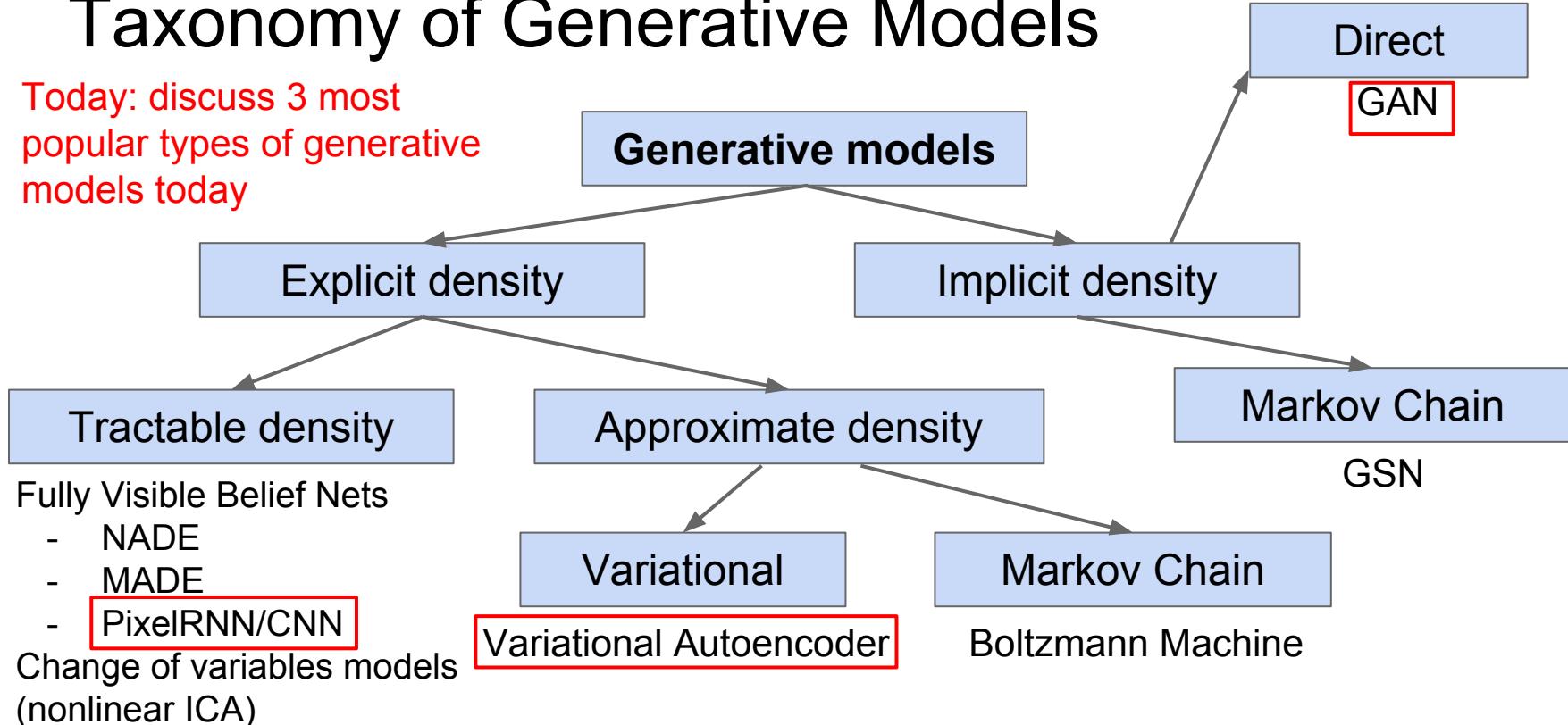


Figure copyright and adapted from Ian Goodfellow, Tutorial on Generative Adversarial Networks, 2017.

PixelRNN and PixelCNN

Fully visible belief network

Explicit density model

Use chain rule to decompose likelihood of an image x into product of 1-d distributions:

$$p(x) = \prod_{i=1}^n p(x_i|x_1, \dots, x_{i-1})$$

Likelihood of image x

Probability of i 'th pixel value given all previous pixels

Then maximize likelihood of training data

Fully visible belief network

Explicit density model

Use chain rule to decompose likelihood of an image x into product of 1-d distributions:

$$p(x) = \prod_{i=1}^n p(x_i|x_1, \dots, x_{i-1})$$

↑ ↑

Likelihood of image x Probability of i 'th pixel value given all previous pixels

Complex distribution over pixel values => Express using a neural network!

Then maximize likelihood of training data

Fully visible belief network

Explicit density model

Use chain rule to decompose likelihood of an image x into product of 1-d distributions:

$$p(x) = \prod_{i=1}^n p(x_i|x_1, \dots, x_{i-1})$$

↑ ↑

Likelihood of image x Probability of i 'th pixel value given all previous pixels

likelihood of training data

Will need to define ordering of “previous pixels”

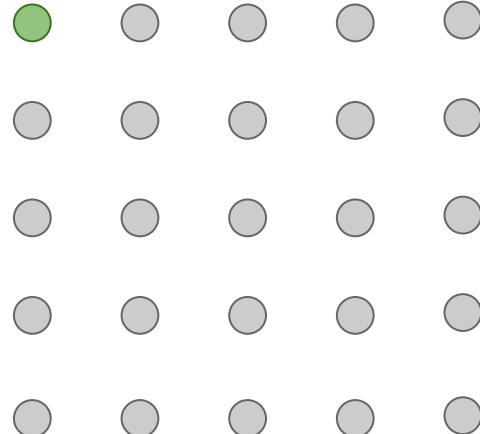
Complex distribution over pixel values => Express using a neural network!

Then maximize likelihood of training data

PixelRNN

[van der Oord et al. 2016]

Generate image pixels starting from corner



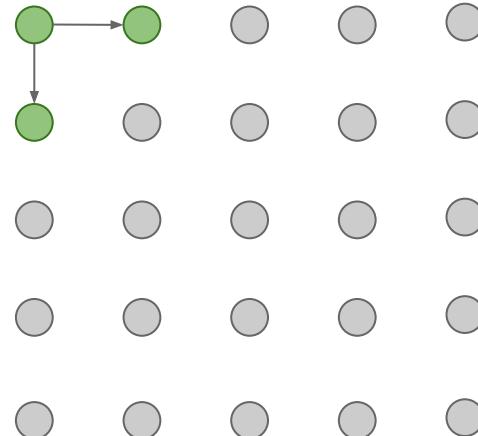
Dependency on previous pixels modeled
using an RNN (LSTM)

PixelRNN

[van der Oord et al. 2016]

Generate image pixels starting from corner

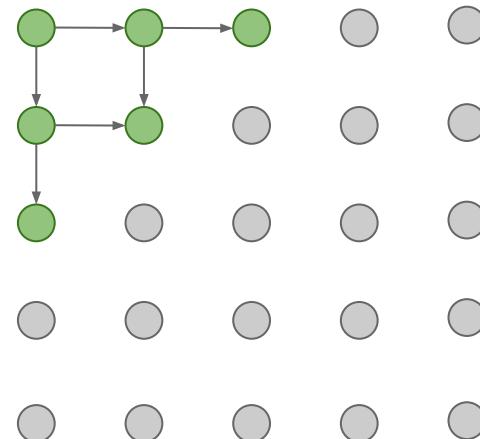
Dependency on previous pixels modeled
using an RNN (LSTM)



PixelRNN [van der Oord et al. 2016]

Generate image pixels starting from corner

Dependency on previous pixels modeled using an RNN (LSTM)



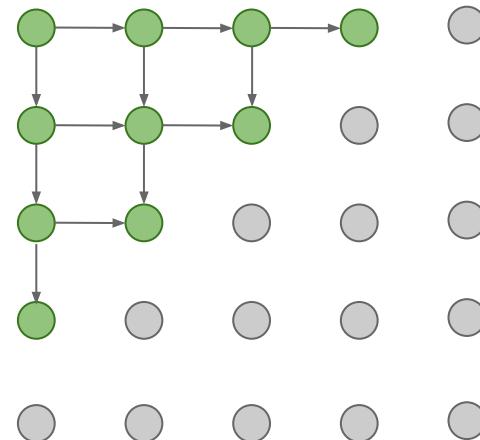
PixelRNN

[van der Oord et al. 2016]

Generate image pixels starting from corner

Dependency on previous pixels modeled
using an RNN (LSTM)

Drawback: sequential generation is slow!



PixelCNN

[van der Oord et al. 2016]

Still generate image pixels starting from corner

Dependency on previous pixels now modeled using a CNN over context region

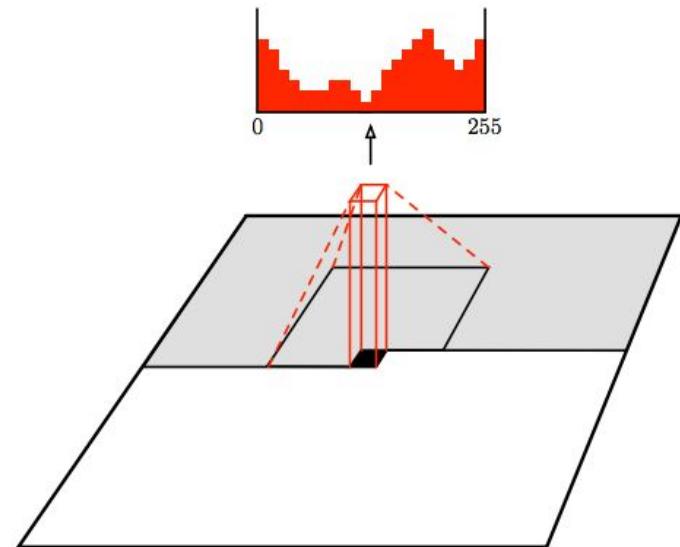


Figure copyright van der Oord et al., 2016. Reproduced with permission.

PixelCNN

[van der Oord et al. 2016]

Still generate image pixels starting from corner

Dependency on previous pixels now modeled using a CNN over context region

Training: maximize likelihood of training images

$$p(x) = \prod_{i=1}^n p(x_i|x_1, \dots, x_{i-1})$$

Softmax loss at each pixel

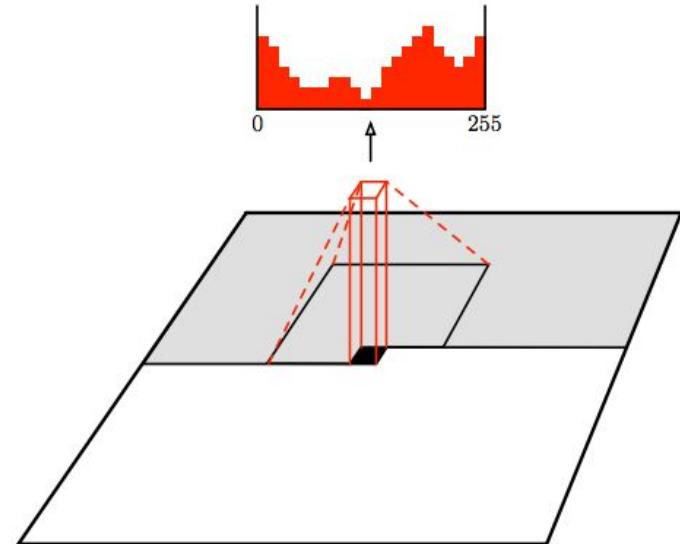


Figure copyright van der Oord et al., 2016. Reproduced with permission.

PixelCNN

[van der Oord et al. 2016]

Still generate image pixels starting from corner

Dependency on previous pixels now modeled using a CNN over context region

Training is faster than PixelRNN
(can parallelize convolutions since context region values known from training images)

Generation must still proceed sequentially
=> still slow

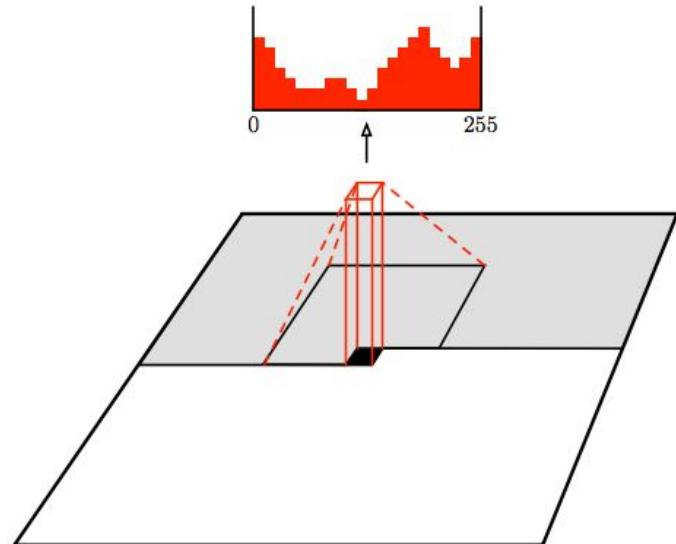
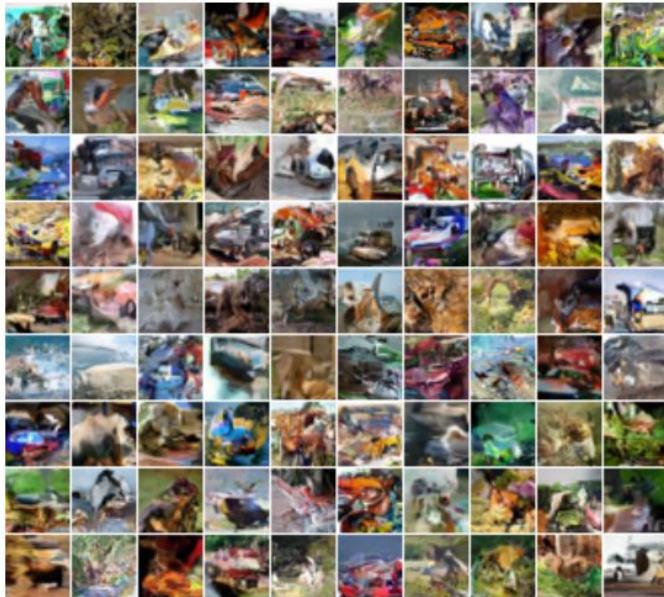


Figure copyright van der Oord et al., 2016. Reproduced with permission.

Generation Samples



32x32 CIFAR-10



32x32 ImageNet

Figures copyright Aaron van der Oord et al., 2016. Reproduced with permission.

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++)

Variational Autoencoders (VAE)

So far...

PixelCNNs define tractable density function, optimize likelihood of training data:

$$p_{\theta}(x) = \prod_{i=1}^n p_{\theta}(x_i|x_1, \dots, x_{i-1})$$

So far...

PixelCNNs define tractable density function, optimize likelihood of training data:

$$p_{\theta}(x) = \prod_{i=1}^n p_{\theta}(x_i|x_1, \dots, x_{i-1})$$

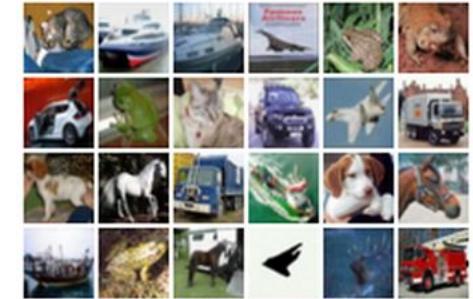
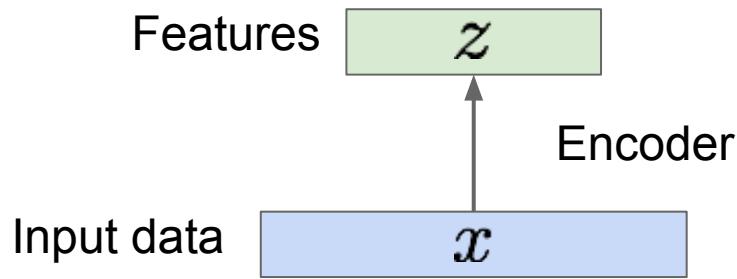
VAEs define intractable density function with latent \mathbf{z} :

$$p_{\theta}(x) = \int p_{\theta}(z)p_{\theta}(x|z)dz$$

Cannot optimize directly, derive and optimize lower bound on likelihood instead

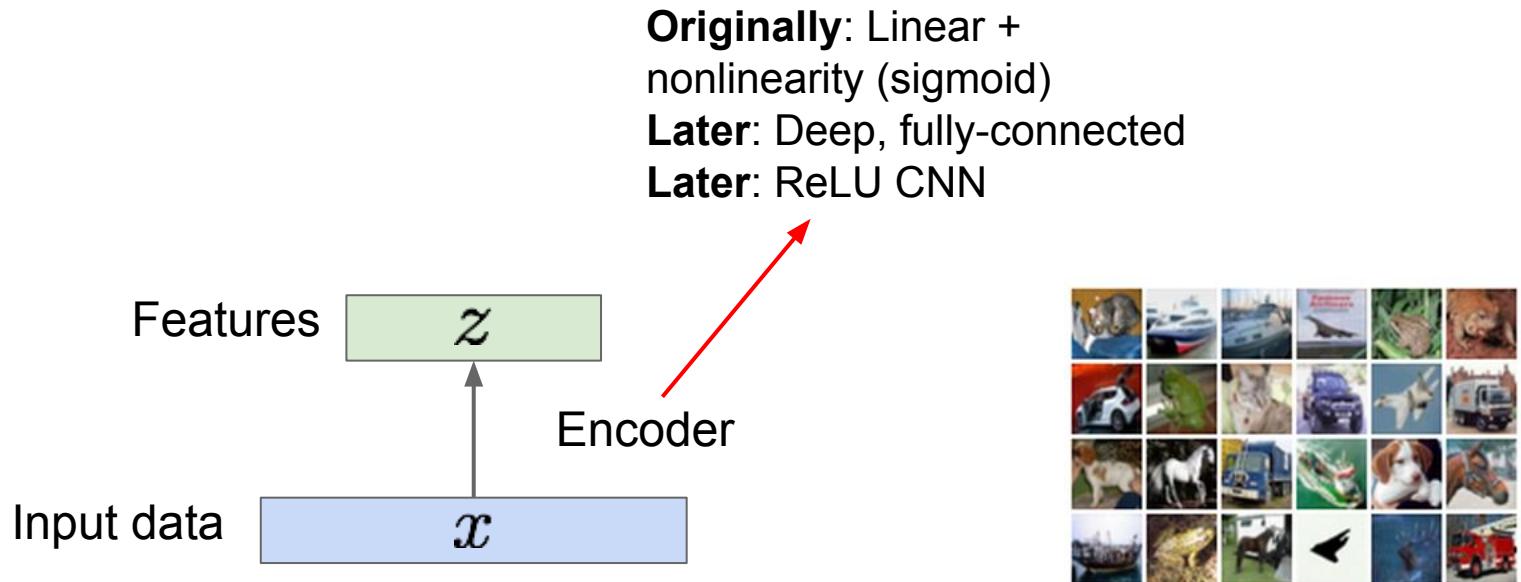
Some background first: Autoencoders

Unsupervised approach for learning a lower-dimensional feature representation from unlabeled training data



Some background first: Autoencoders

Unsupervised approach for learning a lower-dimensional feature representation from unlabeled training data

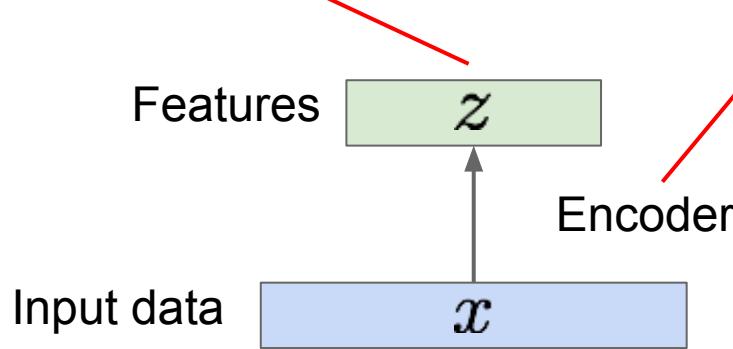


Some background first: Autoencoders

Unsupervised approach for learning a lower-dimensional feature representation from unlabeled training data

z usually smaller than x
(dimensionality reduction)

Q: Why dimensionality reduction?



Originally: Linear +
nonlinearity (sigmoid)
Later: Deep, fully-connected
Later: ReLU CNN



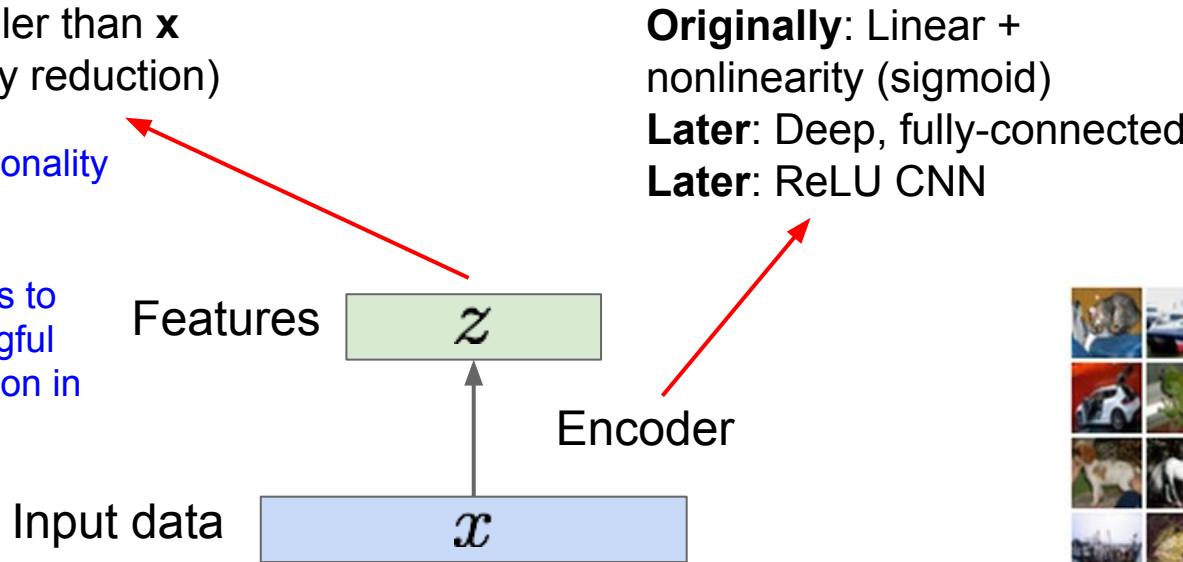
Some background first: Autoencoders

Unsupervised approach for learning a lower-dimensional feature representation from unlabeled training data

z usually smaller than x
(dimensionality reduction)

Q: Why dimensionality reduction?

A: Want features to capture meaningful factors of variation in data

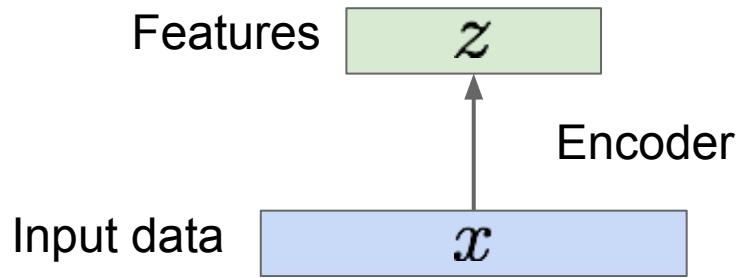


Originally: Linear +
nonlinearity (sigmoid)
Later: Deep, fully-connected
Later: ReLU CNN



Some background first: Autoencoders

How to learn this feature representation?

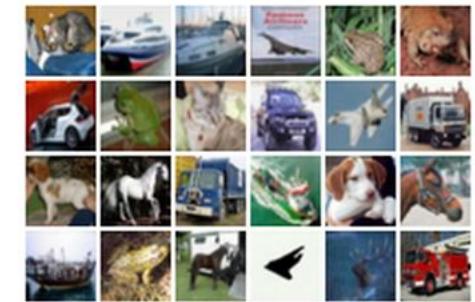
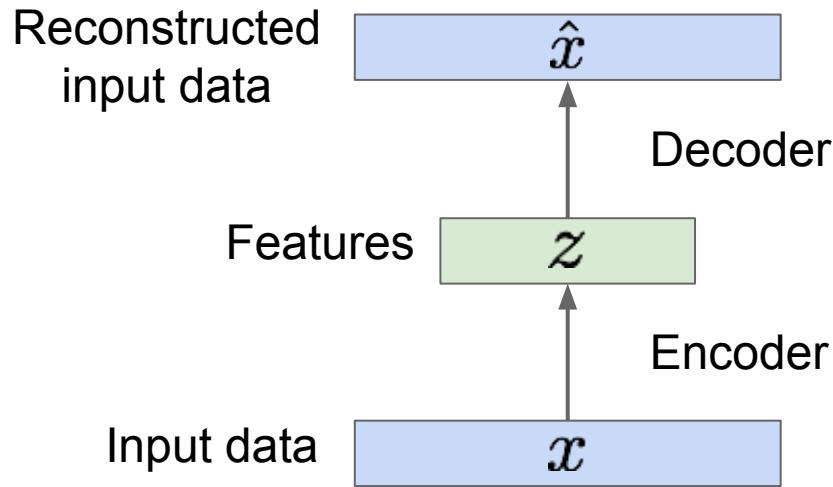


Some background first: Autoencoders

How to learn this feature representation?

Train such that features can be used to reconstruct original data

“Autoencoding” - encoding itself

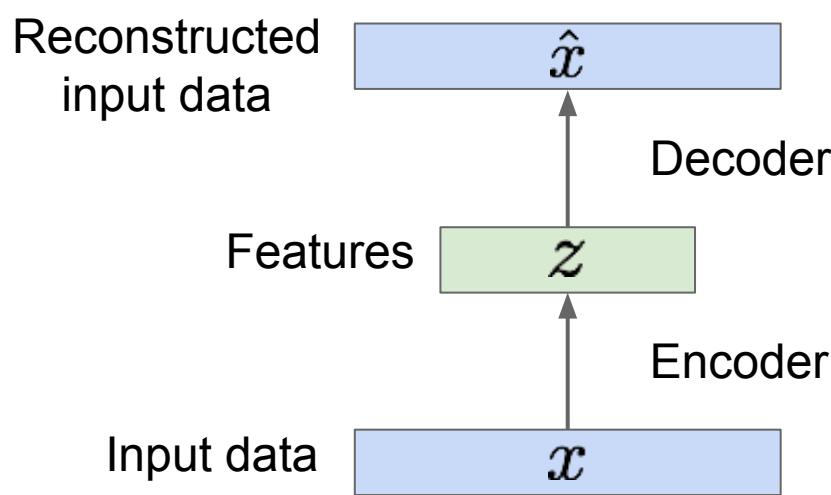


Some background first: Autoencoders

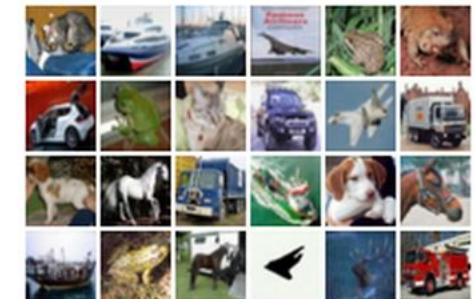
How to learn this feature representation?

Train such that features can be used to reconstruct original data

“Autoencoding” - encoding itself



Originally: Linear +
nonlinearity (sigmoid)
Later: Deep, fully-connected
Later: ReLU CNN (upconv)

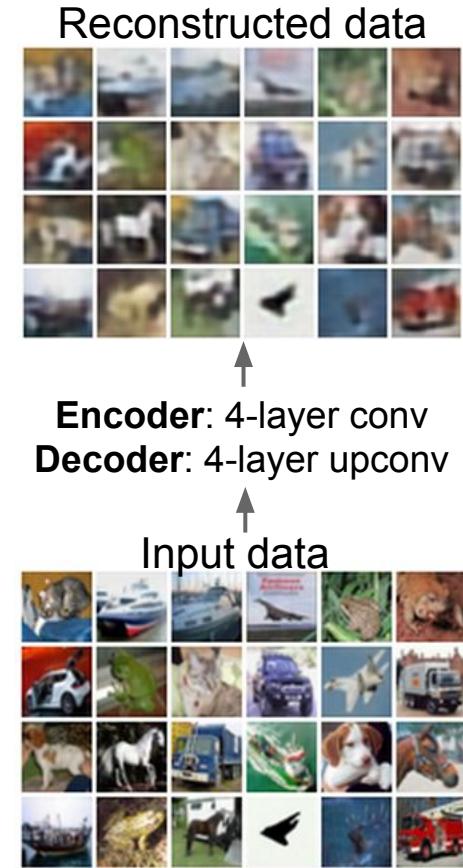
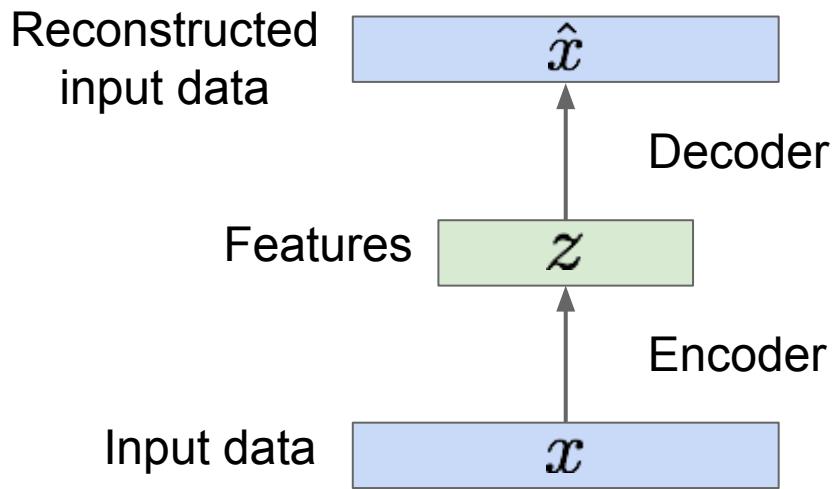


Some background first: Autoencoders

How to learn this feature representation?

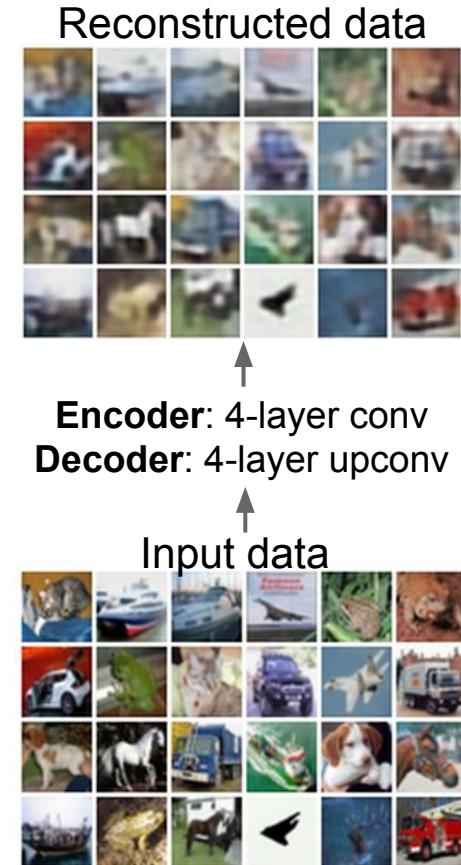
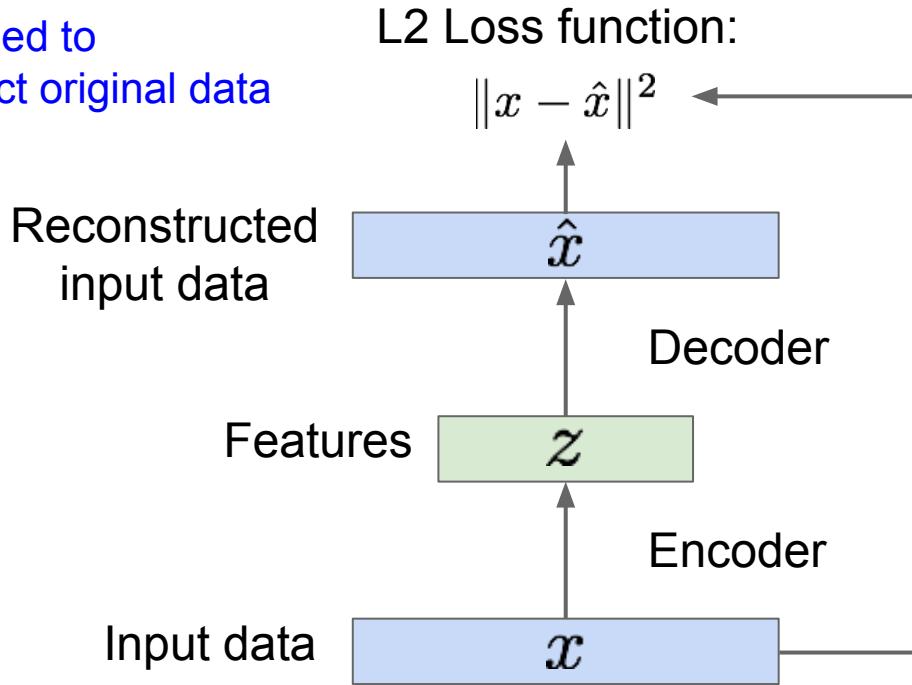
Train such that features can be used to reconstruct original data

“Autoencoding” - encoding itself



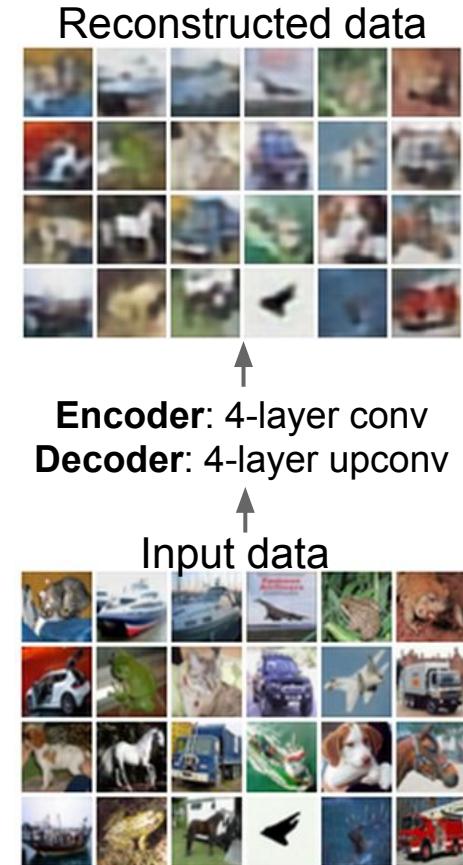
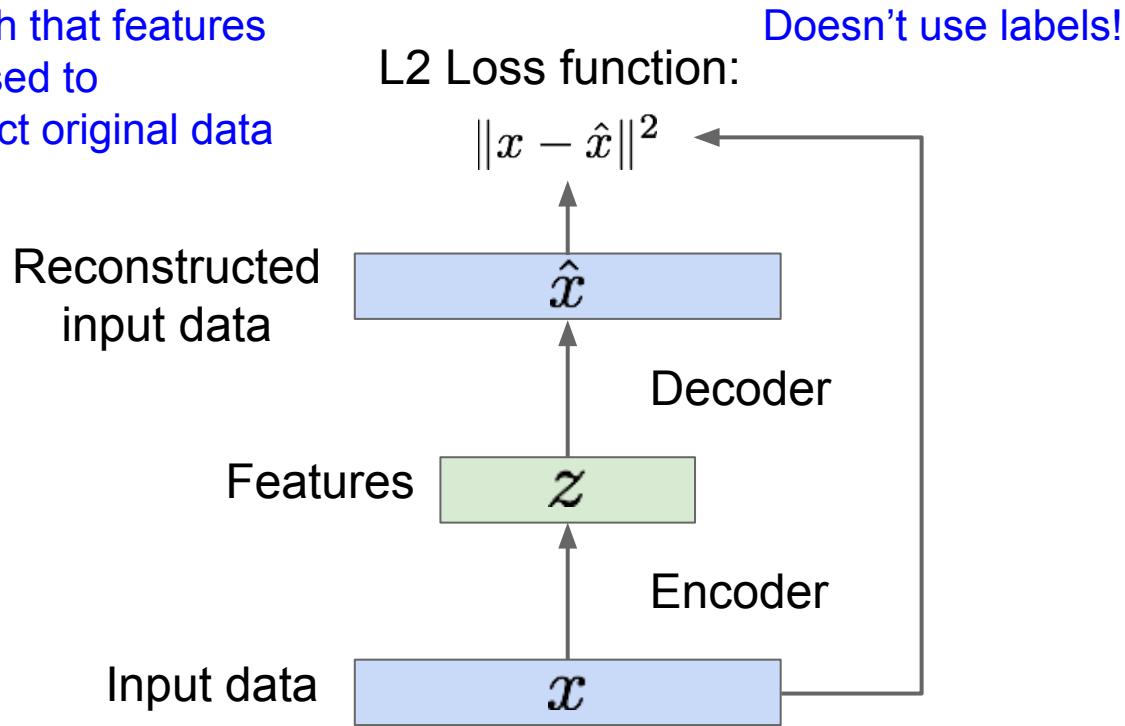
Some background first: Autoencoders

Train such that features
can be used to
reconstruct original data

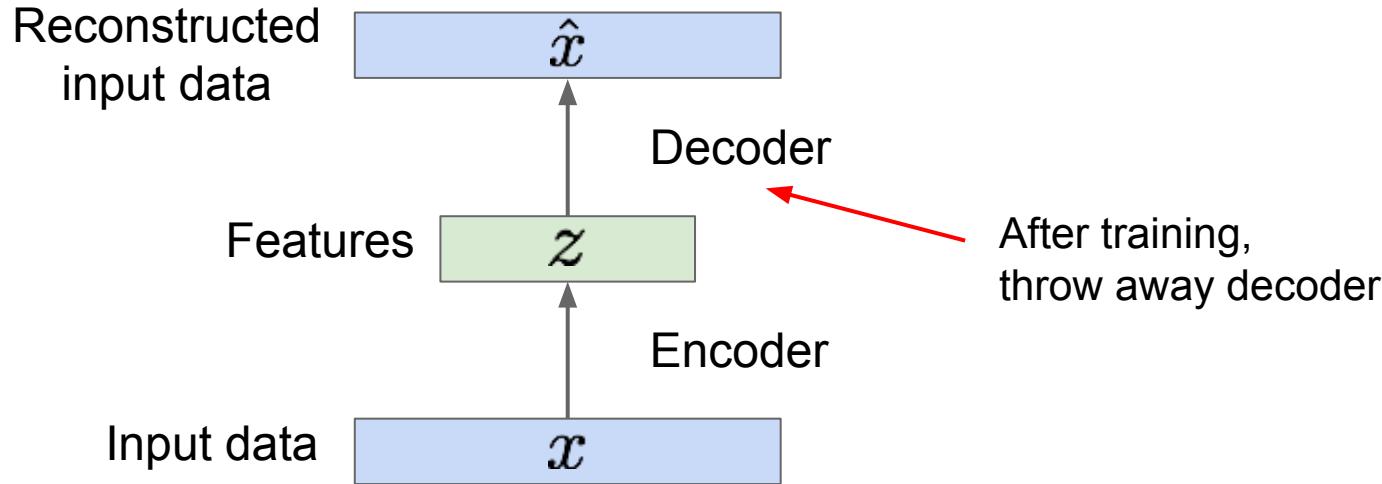


Some background first: Autoencoders

Train such that features
can be used to
reconstruct original data

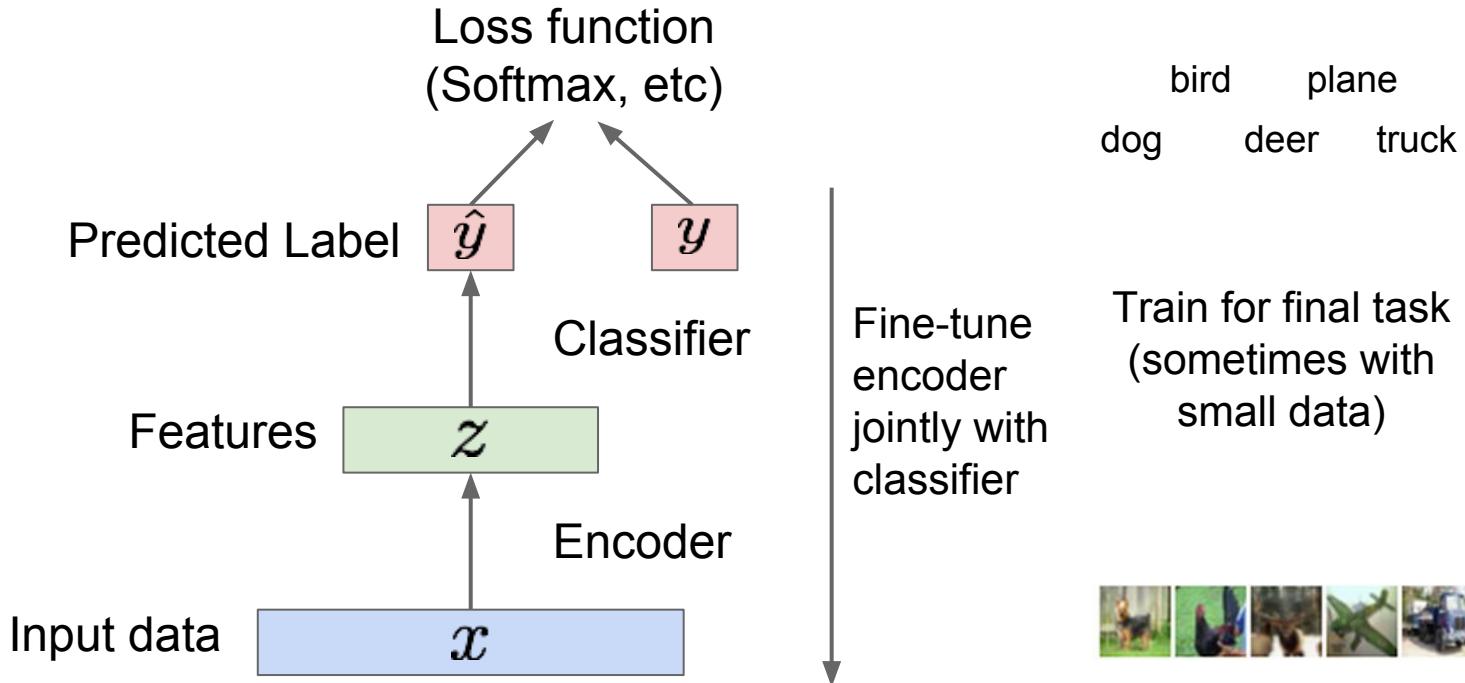


Some background first: Autoencoders

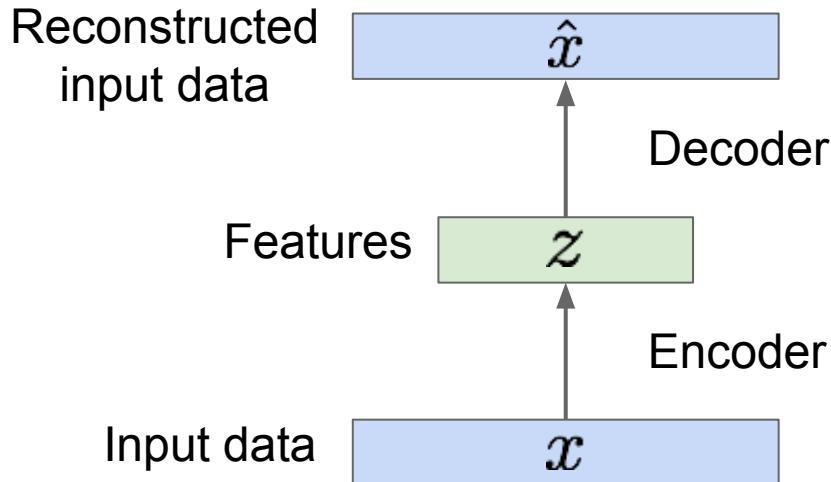


Some background first: Autoencoders

Encoder can be used to initialize a **supervised** model



Some background first: Autoencoders



Autoencoders can reconstruct data, and can learn features to initialize a supervised model

Features capture factors of variation in training data. Can we generate new images from an autoencoder?

Variational Autoencoders

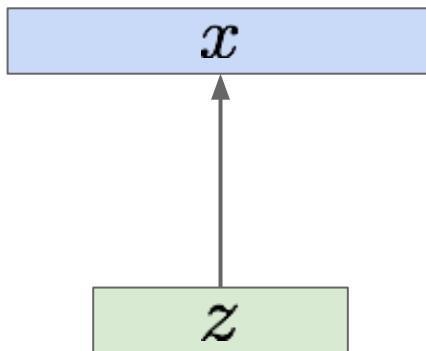
Probabilistic spin on autoencoders - will let us sample from the model to generate data!

Variational Autoencoders

Probabilistic spin on autoencoders - will let us sample from the model to generate data!

Assume training data $\{x^{(i)}\}_{i=1}^N$ is generated from underlying unobserved (latent) representation z

Sample from
true conditional
 $p_{\theta^*}(x \mid z^{(i)})$



Sample from
true prior
 $p_{\theta^*}(z)$

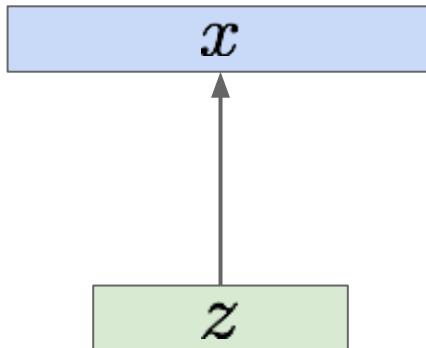
Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders

Probabilistic spin on autoencoders - will let us sample from the model to generate data!

Assume training data $\{x^{(i)}\}_{i=1}^N$ is generated from underlying unobserved (latent) representation z

Sample from
true conditional
 $p_{\theta^*}(x \mid z^{(i)})$



Intuition (remember from autoencoders!):
 x is an image, z is latent factors used to generate x : attributes, orientation, etc.

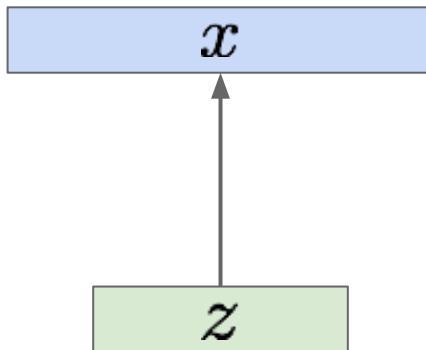
Sample from
true prior
 $p_{\theta^*}(z)$

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders

We want to estimate the true parameters θ^* of this generative model.

Sample from
true conditional
 $p_{\theta^*}(x \mid z^{(i)})$

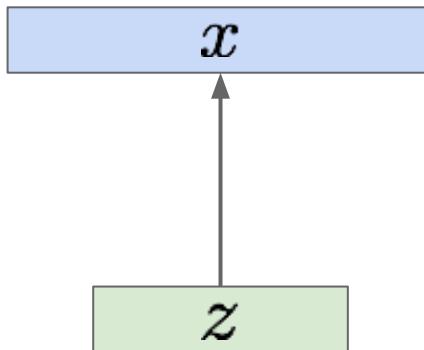


Sample from
true prior
 $p_{\theta^*}(z)$

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders

Sample from
true conditional
 $p_{\theta^*}(x \mid z^{(i)})$



Sample from
true prior
 $p_{\theta^*}(z)$

We want to estimate the true parameters θ^* of this generative model.

How should we represent this model?

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders

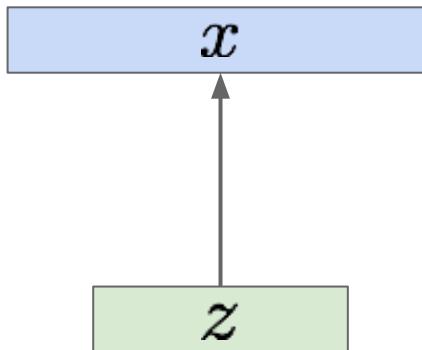
Sample from
true conditional

$$p_{\theta^*}(x \mid z^{(i)})$$

Sample from

true prior

$$p_{\theta^*}(z)$$



We want to estimate the true parameters θ^* of this generative model.

How should we represent this model?

Choose prior $p(z)$ to be simple, e.g. Gaussian. Reasonable for latent attributes, e.g. pose, how much smile.

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

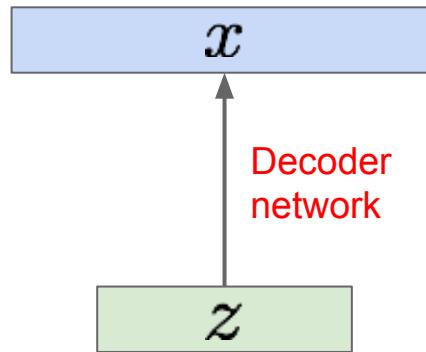
Variational Autoencoders

Sample from
true conditional

$$p_{\theta^*}(x \mid z^{(i)})$$

Sample from
true prior

$$p_{\theta^*}(z)$$



We want to estimate the true parameters θ^* of this generative model.

How should we represent this model?

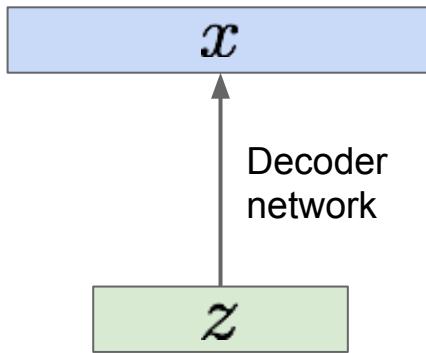
Choose prior $p(z)$ to be simple, e.g.
Gaussian.

Conditional $p(x|z)$ is complex (generates
image) => represent with neural network

Kingma and Welling, “Auto-Encoding Variational Bayes”, ICLR 2014

Variational Autoencoders

Sample from
true conditional
 $p_{\theta^*}(x \mid z^{(i)})$



Sample from
true prior
 $p_{\theta^*}(z)$

We want to estimate the true parameters θ^* of this generative model.

How to train the model?

Variational Autoencoders

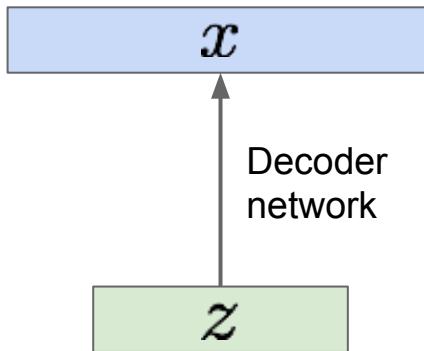
Sample from
true conditional

$$p_{\theta^*}(x \mid z^{(i)})$$

Sample from

true prior

$$p_{\theta^*}(z)$$



We want to estimate the true parameters θ^* of this generative model.

How to train the model?

Remember strategy for training generative models from FVBMs. Learn model parameters to maximize likelihood of training data

$$p_{\theta}(x) = \int p_{\theta}(z)p_{\theta}(x|z)dz$$

Kingma and Welling, “Auto-Encoding Variational Bayes”, ICLR 2014

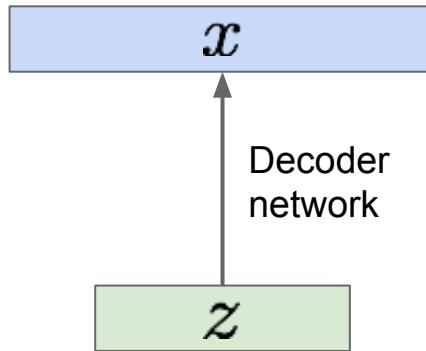
Variational Autoencoders

Sample from
true conditional

$$p_{\theta^*}(x \mid z^{(i)})$$

Sample from
true prior

$$p_{\theta^*}(z)$$



We want to estimate the true parameters θ^* of this generative model.

How to train the model?

Remember strategy for training generative models from FVBMs. Learn model parameters to maximize likelihood of training data

$$p_{\theta}(x) = \int p_{\theta}(z)p_{\theta}(x|z)dz$$

Now with latent z

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

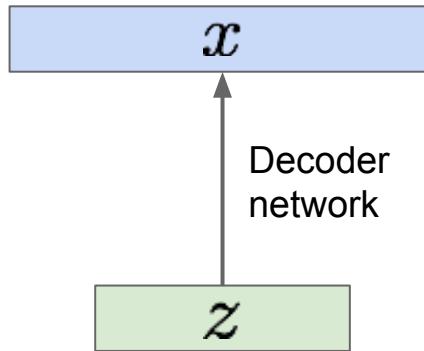
Variational Autoencoders

Sample from
true conditional

$$p_{\theta^*}(x \mid z^{(i)})$$

Sample from
true prior

$$p_{\theta^*}(z)$$



We want to estimate the true parameters θ^* of this generative model.

How to train the model?

Remember strategy for training generative models from FVBMs. Learn model parameters to maximize likelihood of training data

$$p_{\theta}(x) = \int p_{\theta}(z)p_{\theta}(x|z)dz$$

Q: What is the problem with this?

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

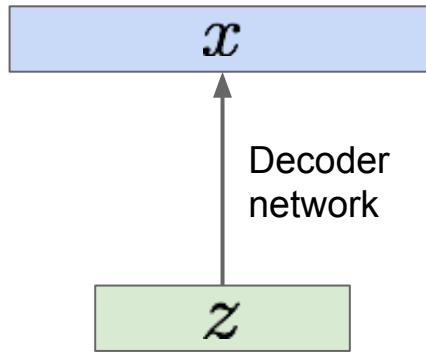
Variational Autoencoders

Sample from
true conditional

$$p_{\theta^*}(x \mid z^{(i)})$$

Sample from
true prior

$$p_{\theta^*}(z)$$



We want to estimate the true parameters θ^* of this generative model.

[How to train the model?](#)

Remember strategy for training generative models from FVBMs. Learn model parameters to maximize likelihood of training data

$$p_{\theta}(x) = \int p_{\theta}(z)p_{\theta}(x|z)dz$$

[Q: What is the problem with this?](#)

Intractable!

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders: Intractability

Data likelihood: $p_\theta(x) = \int p_\theta(z)p_\theta(x|z)dz$

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders: Intractability

Data likelihood: $p_{\theta}(x) = \int p_{\theta}(z)p_{\theta}(x|z)dz$

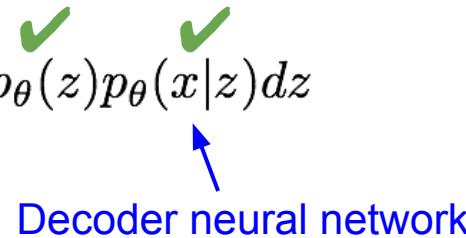
$$p_{\theta}(x) = \int p_{\theta}(z)p_{\theta}(x|z)dz$$

Simple Gaussian prior

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders: Intractability

Data likelihood: $p_{\theta}(x) = \int p_{\theta}(z)p_{\theta}(x|z)dz$



Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders: Intractability

Data likelihood: $p_\theta(x) = \int p_\theta(z)p_\theta(x|z)dz$



Intractible to compute
 $p(x|z)$ for every z !

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders: Intractability



Data likelihood: $p_\theta(x) = \int p_\theta(z)p_\theta(x|z)dz$

Posterior density also intractable: $p_\theta(z|x) = p_\theta(x|z)p_\theta(z)/p_\theta(x)$

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders: Intractability

Data likelihood: $p_\theta(x) = \int p_\theta(z)p_\theta(x|z)dz$

Posterior density also intractable: $p_\theta(z|x) = p_\theta(x|z)p_\theta(z)/p_\theta(x)$

Intractable data likelihood

Variational Autoencoders: Intractability

Data likelihood: $p_\theta(x) = \int p_\theta(z)p_\theta(x|z)dz$

Posterior density also intractable: $p_\theta(z|x) = p_\theta(x|z)p_\theta(z)/p_\theta(x)$

Solution: In addition to decoder network modeling $p_\theta(x|z)$, define additional encoder network $q_\phi(z|x)$ that approximates $p_\theta(z|x)$

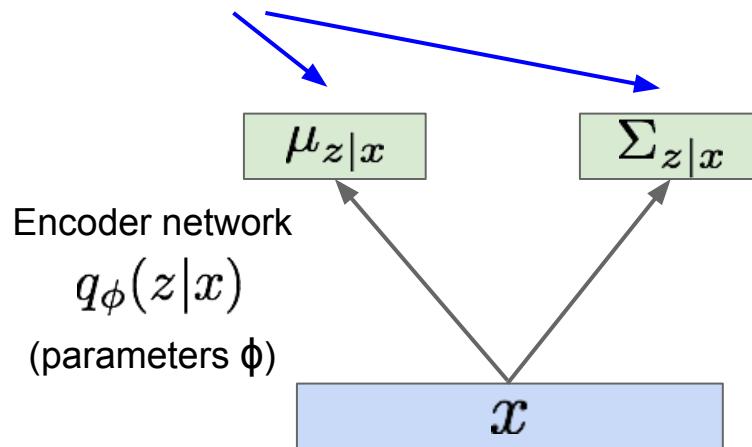
Will see that this allows us to derive a lower bound on the data likelihood that is tractable, which we can optimize

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders

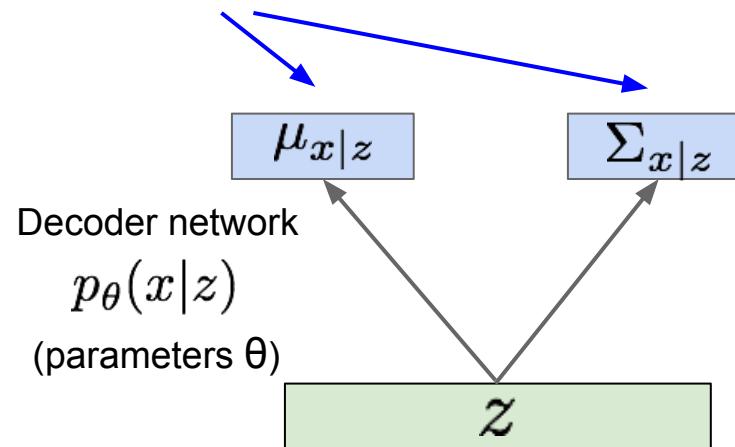
Since we're modeling probabilistic generation of data, encoder and decoder networks are probabilistic

Mean and (diagonal) covariance of $z | x$



Encoder network
 $q_\phi(z|x)$
(parameters ϕ)

Mean and (diagonal) covariance of $x | z$

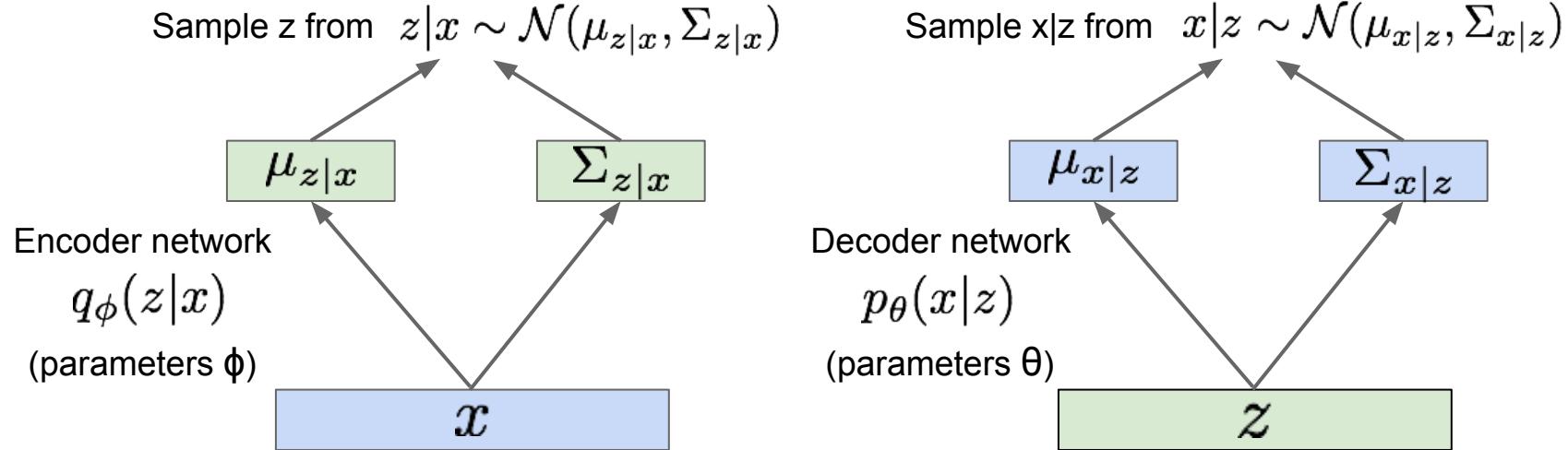


Decoder network
 $p_\theta(x|z)$
(parameters θ)

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders

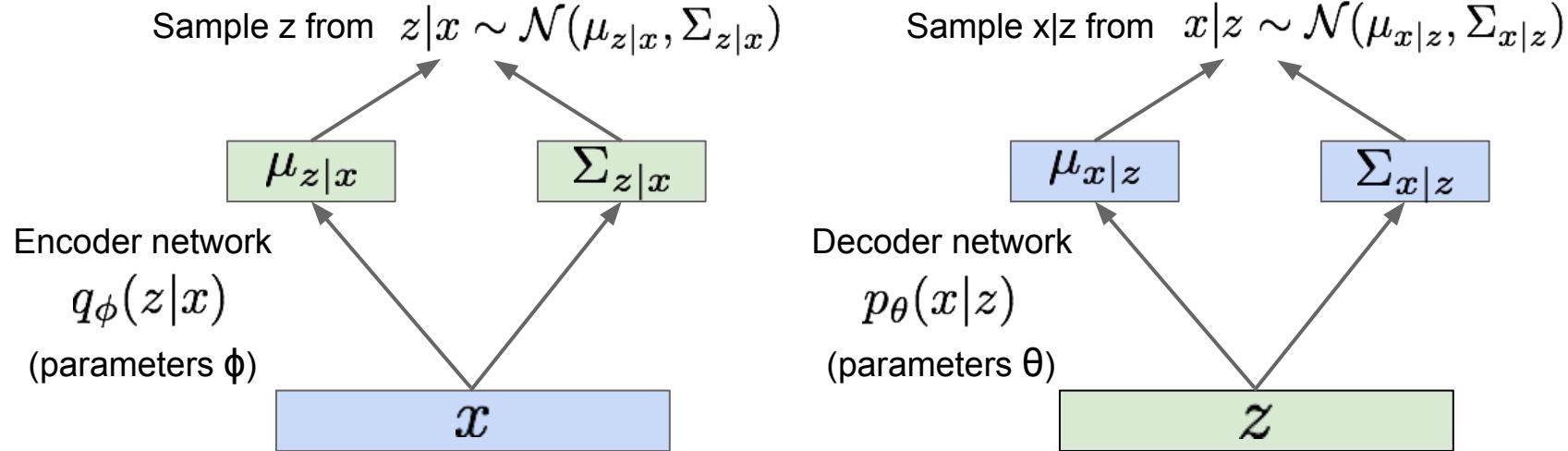
Since we're modeling probabilistic generation of data, encoder and decoder networks are probabilistic



Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders

Since we're modeling probabilistic generation of data, encoder and decoder networks are probabilistic



Encoder and decoder networks also called
“recognition”/“inference” and “generation” networks

Kingma and Welling, “Auto-Encoding Variational Bayes”, ICLR 2014

Variational Autoencoders

Now equipped with our encoder and decoder networks, let's work out the (log) data likelihood:

$$\log p_{\theta}(x^{(i)}) = \mathbf{E}_{z \sim q_{\phi}(z|x^{(i)})} [\log p_{\theta}(x^{(i)})] \quad (p_{\theta}(x^{(i)}) \text{ Does not depend on } z)$$

Variational Autoencoders

Now equipped with our encoder and decoder networks, let's work out the (log) data likelihood:

$$\log p_{\theta}(x^{(i)}) = \mathbf{E}_{z \sim q_{\phi}(z|x^{(i)})} [\log p_{\theta}(x^{(i)})] \quad (p_{\theta}(x^{(i)}) \text{ Does not depend on } z)$$



Taking expectation wrt. z
(using encoder network) will
come in handy later

Variational Autoencoders

Now equipped with our encoder and decoder networks, let's work out the (log) data likelihood:

$$\begin{aligned}\log p_{\theta}(x^{(i)}) &= \mathbf{E}_{z \sim q_{\phi}(z|x^{(i)})} [\log p_{\theta}(x^{(i)})] \quad (p_{\theta}(x^{(i)}) \text{ Does not depend on } z) \\ &= \mathbf{E}_z \left[\log \frac{p_{\theta}(x^{(i)} | z)p_{\theta}(z)}{p_{\theta}(z | x^{(i)})} \right] \quad (\text{Bayes' Rule})\end{aligned}$$

Variational Autoencoders

Now equipped with our encoder and decoder networks, let's work out the (log) data likelihood:

$$\begin{aligned}\log p_{\theta}(x^{(i)}) &= \mathbf{E}_{z \sim q_{\phi}(z|x^{(i)})} \left[\log p_{\theta}(x^{(i)}) \right] \quad (p_{\theta}(x^{(i)}) \text{ Does not depend on } z) \\ &= \mathbf{E}_z \left[\log \frac{p_{\theta}(x^{(i)} | z)p_{\theta}(z)}{p_{\theta}(z | x^{(i)})} \right] \quad (\text{Bayes' Rule}) \\ &= \mathbf{E}_z \left[\log \frac{p_{\theta}(x^{(i)} | z)p_{\theta}(z)}{p_{\theta}(z | x^{(i)})} \frac{q_{\phi}(z | x^{(i)})}{q_{\phi}(z | x^{(i)})} \right] \quad (\text{Multiply by constant})\end{aligned}$$

Variational Autoencoders

Now equipped with our encoder and decoder networks, let's work out the (log) data likelihood:

$$\begin{aligned}\log p_\theta(x^{(i)}) &= \mathbf{E}_{z \sim q_\phi(z|x^{(i)})} [\log p_\theta(x^{(i)})] \quad (p_\theta(x^{(i)}) \text{ Does not depend on } z) \\ &= \mathbf{E}_z \left[\log \frac{p_\theta(x^{(i)} | z)p_\theta(z)}{p_\theta(z | x^{(i)})} \right] \quad (\text{Bayes' Rule}) \\ &= \mathbf{E}_z \left[\log \frac{p_\theta(x^{(i)} | z)p_\theta(z)}{p_\theta(z | x^{(i)})} \frac{q_\phi(z | x^{(i)})}{q_\phi(z | x^{(i)})} \right] \quad (\text{Multiply by constant}) \\ &= \mathbf{E}_z [\log p_\theta(x^{(i)} | z)] - \mathbf{E}_z \left[\log \frac{q_\phi(z | x^{(i)})}{p_\theta(z)} \right] + \mathbf{E}_z \left[\log \frac{q_\phi(z | x^{(i)})}{p_\theta(z | x^{(i)})} \right] \quad (\text{Logarithms})\end{aligned}$$

Variational Autoencoders

Now equipped with our encoder and decoder networks, let's work out the (log) data likelihood:

$$\begin{aligned}\log p_{\theta}(x^{(i)}) &= \mathbf{E}_{z \sim q_{\phi}(z | x^{(i)})} [\log p_{\theta}(x^{(i)})] \quad (p_{\theta}(x^{(i)}) \text{ Does not depend on } z) \\ &= \mathbf{E}_z \left[\log \frac{p_{\theta}(x^{(i)} | z)p_{\theta}(z)}{p_{\theta}(z | x^{(i)})} \right] \quad (\text{Bayes' Rule}) \\ &= \mathbf{E}_z \left[\log \frac{p_{\theta}(x^{(i)} | z)p_{\theta}(z)}{p_{\theta}(z | x^{(i)})} \frac{q_{\phi}(z | x^{(i)})}{q_{\phi}(z | x^{(i)})} \right] \quad (\text{Multiply by constant}) \\ &= \mathbf{E}_z [\log p_{\theta}(x^{(i)} | z)] - \mathbf{E}_z \left[\log \frac{q_{\phi}(z | x^{(i)})}{p_{\theta}(z)} \right] + \mathbf{E}_z \left[\log \frac{q_{\phi}(z | x^{(i)})}{p_{\theta}(z | x^{(i)})} \right] \quad (\text{Logarithms}) \\ &= \mathbf{E}_z [\log p_{\theta}(x^{(i)} | z)] - D_{KL}(q_{\phi}(z | x^{(i)}) || p_{\theta}(z)) + D_{KL}(q_{\phi}(z | x^{(i)}) || p_{\theta}(z | x^{(i)}))\end{aligned}$$

Variational Autoencoders

Now equipped with our encoder and decoder networks, let's work out the (log) data likelihood:

$$\begin{aligned}\log p_\theta(x^{(i)}) &= \mathbf{E}_{z \sim q_\phi(z|x^{(i)})} [\log p_\theta(x^{(i)})] \quad (p_\theta(x^{(i)}) \text{ Does not depend on } z) \\ &= \mathbf{E}_z \left[\log \frac{p_\theta(x^{(i)} | z)p_\theta(z)}{p_\theta(z | x^{(i)})} \right] \quad (\text{Bayes' Rule}) \\ &= \mathbf{E}_z \left[\log \frac{p_\theta(x^{(i)} | z)p_\theta(z)}{p_\theta(z | x^{(i)})} \frac{q_\phi(z | x^{(i)})}{q_\phi(z | x^{(i)})} \right] \quad (\text{Multiply by constant}) \\ &= \mathbf{E}_z [\log p_\theta(x^{(i)} | z)] - \mathbf{E}_z \left[\log \frac{q_\phi(z | x^{(i)})}{p_\theta(z)} \right] + \mathbf{E}_z \left[\log \frac{q_\phi(z | x^{(i)})}{p_\theta(z | x^{(i)})} \right] \quad (\text{Logarithms}) \\ &= \mathbf{E}_z [\log p_\theta(x^{(i)} | z)] - D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z)) + D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z | x^{(i)}))\end{aligned}$$



The expectation wrt. z (using encoder network) let us write nice KL terms

Variational Autoencoders

Now equipped with our encoder and decoder networks, let's work out the (log) data likelihood:

$$\begin{aligned}\log p_\theta(x^{(i)}) &= \mathbf{E}_{z \sim q_\phi(z|x^{(i)})} [\log p_\theta(x^{(i)})] \quad (p_\theta(x^{(i)}) \text{ Does not depend on } z) \\ &= \mathbf{E}_z \left[\log \frac{p_\theta(x^{(i)} | z)p_\theta(z)}{p_\theta(z | x^{(i)})} \right] \quad (\text{Bayes' Rule}) \\ &= \mathbf{E}_z \left[\log \frac{p_\theta(x^{(i)} | z)p_\theta(z)}{p_\theta(z | x^{(i)})} \frac{q_\phi(z | x^{(i)})}{q_\phi(z | x^{(i)})} \right] \quad (\text{Multiply by constant}) \\ &= \mathbf{E}_z [\log p_\theta(x^{(i)} | z)] - \mathbf{E}_z \left[\log \frac{q_\phi(z | x^{(i)})}{p_\theta(z)} \right] + \mathbf{E}_z \left[\log \frac{q_\phi(z | x^{(i)})}{p_\theta(z | x^{(i)})} \right] \quad (\text{Logarithms}) \\ &= \mathbf{E}_z [\log p_\theta(x^{(i)} | z)] - D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z)) + D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z | x^{(i)}))\end{aligned}$$

↑
Decoder network gives $p_\theta(x|z)$, can
compute estimate of this term through
sampling. (Sampling differentiable
through reparam. trick, see paper.)

↑
This KL term (between
Gaussians for encoder and z
prior) has nice closed-form
solution!

↑
 $p_\theta(z|x)$ intractable (saw
earlier), can't compute this KL
term :(But we know KL
divergence always ≥ 0 .

Variational Autoencoders

Now equipped with our encoder and decoder networks, let's work out the (log) data likelihood:

$$\begin{aligned}\log p_\theta(x^{(i)}) &= \mathbf{E}_{z \sim q_\phi(z|x^{(i)})} [\log p_\theta(x^{(i)})] \quad (p_\theta(x^{(i)}) \text{ Does not depend on } z) \\ &= \mathbf{E}_z \left[\log \frac{p_\theta(x^{(i)} | z)p_\theta(z)}{p_\theta(z | x^{(i)})} \right] \quad (\text{Bayes' Rule}) \\ &= \mathbf{E}_z \left[\log \frac{p_\theta(x^{(i)} | z)p_\theta(z)}{p_\theta(z | x^{(i)})} \frac{q_\phi(z | x^{(i)})}{q_\phi(z | x^{(i)})} \right] \quad (\text{Multiply by constant}) \\ &= \mathbf{E}_z [\log p_\theta(x^{(i)} | z)] - \mathbf{E}_z \left[\log \frac{q_\phi(z | x^{(i)})}{p_\theta(z)} \right] + \mathbf{E}_z \left[\log \frac{q_\phi(z | x^{(i)})}{p_\theta(z | x^{(i)})} \right] \quad (\text{Logarithms}) \\ &= \underbrace{\mathbf{E}_z [\log p_\theta(x^{(i)} | z)] - D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z))}_{\mathcal{L}(x^{(i)}, \theta, \phi)} + \underbrace{D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z | x^{(i)}))}_{\geq 0}\end{aligned}$$

Tractable lower bound which we can take gradient of and optimize! ($p_\theta(x|z)$ differentiable, KL term differentiable)

Variational Autoencoders

Now equipped with our encoder and decoder networks, let's work out the (log) data likelihood:

$$\begin{aligned}\log p_\theta(x^{(i)}) &= \mathbf{E}_{z \sim q_\phi(z|x^{(i)})} [\log p_\theta(x^{(i)})] \quad (p_\theta(x^{(i)}) \text{ Does not depend on } z) \\ &= \mathbf{E}_z \left[\log \frac{p_\theta(x^{(i)} | z)p_\theta(z)}{p_\theta(z | x^{(i)})} \right] \quad (\text{Bayes' Rule}) \\ &= \mathbf{E}_z \left[\log \frac{p_\theta(x^{(i)} | z)p_\theta(z)}{p_\theta(z | x^{(i)})} \frac{q_\phi(z | x^{(i)})}{q_\phi(z | x^{(i)})} \right] \quad (\text{Multiply by constant}) \\ &= \mathbf{E}_z [\log p_\theta(x^{(i)} | z)] - \mathbf{E}_z \left[\log \frac{q_\phi(z | x^{(i)})}{p_\theta(z)} \right] + \mathbf{E}_z \left[\log \frac{q_\phi(z | x^{(i)})}{p_\theta(z | x^{(i)})} \right] \quad (\text{Logarithms}) \\ &= \underbrace{\mathbf{E}_z [\log p_\theta(x^{(i)} | z)] - D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z))}_{\mathcal{L}(x^{(i)}, \theta, \phi)} + \underbrace{D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z | x^{(i)}))}_{\geq 0} \end{aligned}$$

$$\log p_\theta(x^{(i)}) \geq \mathcal{L}(x^{(i)}, \theta, \phi)$$

Variational lower bound ("ELBO")

$$\theta^*, \phi^* = \arg \max_{\theta, \phi} \sum_{i=1}^N \mathcal{L}(x^{(i)}, \theta, \phi)$$

Training: Maximize lower bound

Variational Autoencoders

Now equipped with our encoder and decoder networks, let's work out the (log) data likelihood:

Reconstruct
the input data

$$\begin{aligned} \log p_\theta(x^{(i)}) &= \mathbf{E}_{z \sim q_\phi(z|x^{(i)})} [\log p_\theta(x^{(i)})] \quad (p_\theta(x^{(i)}) \text{ Does not depend on } z) \\ &= \mathbf{E}_z \left[\log \frac{p_\theta(x^{(i)} | z)p_\theta(z)}{p_\theta(z | x^{(i)})} \right] \quad (\text{Bayes' Rule}) \\ &= \mathbf{E}_z \left[\log \frac{p_\theta(x^{(i)} | z)p_\theta(z)}{p_\theta(z | x^{(i)})} \frac{q_\phi(z | x^{(i)})}{q_\phi(z | x^{(i)})} \right] \quad (\text{Multiply by constant}) \\ &= \mathbf{E}_z [\log p_\theta(x^{(i)} | z)] - \mathbf{E}_z \left[\log \frac{q_\phi(z | x^{(i)})}{p_\theta(z)} \right] + \mathbf{E}_z \left[\log \frac{q_\phi(z | x^{(i)})}{p_\theta(z | x^{(i)})} \right] \quad (\text{Logarithms}) \\ &= \underbrace{\mathbf{E}_z [\log p_\theta(x^{(i)} | z)]}_{\mathcal{L}(x^{(i)}, \theta, \phi)} - D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z)) + \underbrace{D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z | x^{(i)}))}_{> 0} \end{aligned}$$

Make approximate posterior distribution close to prior

$$\log p_\theta(x^{(i)}) \geq \mathcal{L}(x^{(i)}, \theta, \phi)$$

Variational lower bound ("ELBO")

$$\theta^*, \phi^* = \arg \max_{\theta, \phi} \sum_{i=1}^N \mathcal{L}(x^{(i)}, \theta, \phi)$$

Training: Maximize lower bound

Variational Autoencoders

Putting it all together: maximizing the likelihood lower bound

$$\underbrace{\mathbf{E}_z \left[\log p_\theta(x^{(i)} | z) \right] - D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z))}_{\mathcal{L}(x^{(i)}, \theta, \phi)}$$

Variational Autoencoders

Putting it all together: maximizing the likelihood lower bound

$$\underbrace{\mathbf{E}_z \left[\log p_\theta(x^{(i)} | z) \right] - D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z))}_{\mathcal{L}(x^{(i)}, \theta, \phi)}$$

Let's look at computing the bound (forward pass) for a given minibatch of input data

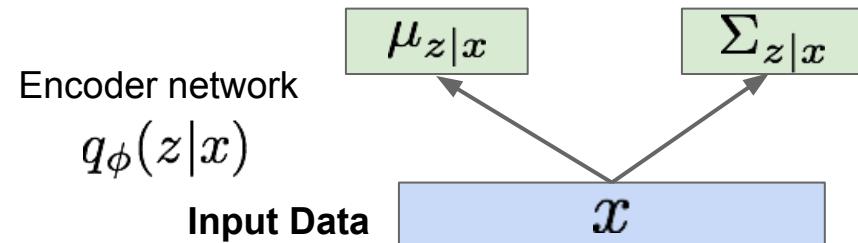
Input Data

x

Variational Autoencoders

Putting it all together: maximizing the likelihood lower bound

$$\underbrace{\mathbf{E}_z \left[\log p_\theta(x^{(i)} | z) \right] - D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z))}_{\mathcal{L}(x^{(i)}, \theta, \phi)}$$



Variational Autoencoders

Putting it all together: maximizing the likelihood lower bound

$$\underbrace{\mathbf{E}_z \left[\log p_\theta(x^{(i)} | z) \right] - D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z))}_{\mathcal{L}(x^{(i)}, \theta, \phi)}$$

Make approximate posterior distribution close to prior

Encoder network

$$q_\phi(z|x)$$

Input Data

$$\mu_{z|x}$$

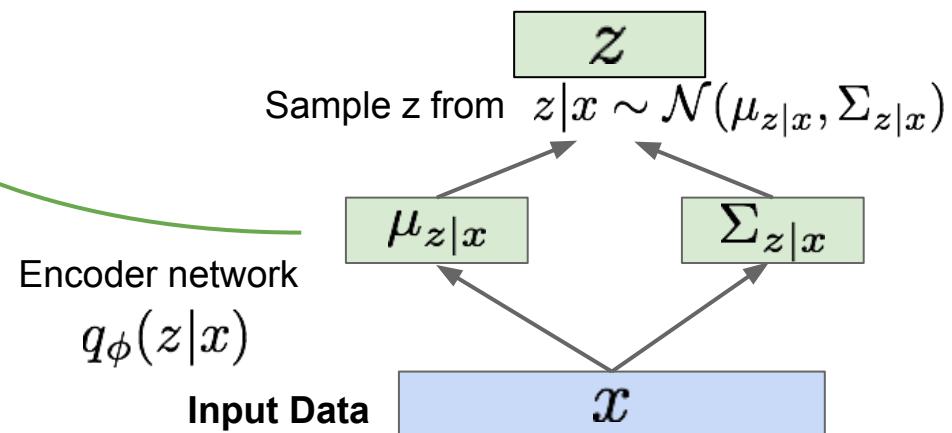
$$\Sigma_{z|x}$$

Variational Autoencoders

Putting it all together: maximizing the likelihood lower bound

$$\underbrace{\mathbf{E}_z \left[\log p_\theta(x^{(i)} | z) \right] - D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z))}_{\mathcal{L}(x^{(i)}, \theta, \phi)}$$

Make approximate posterior distribution close to prior

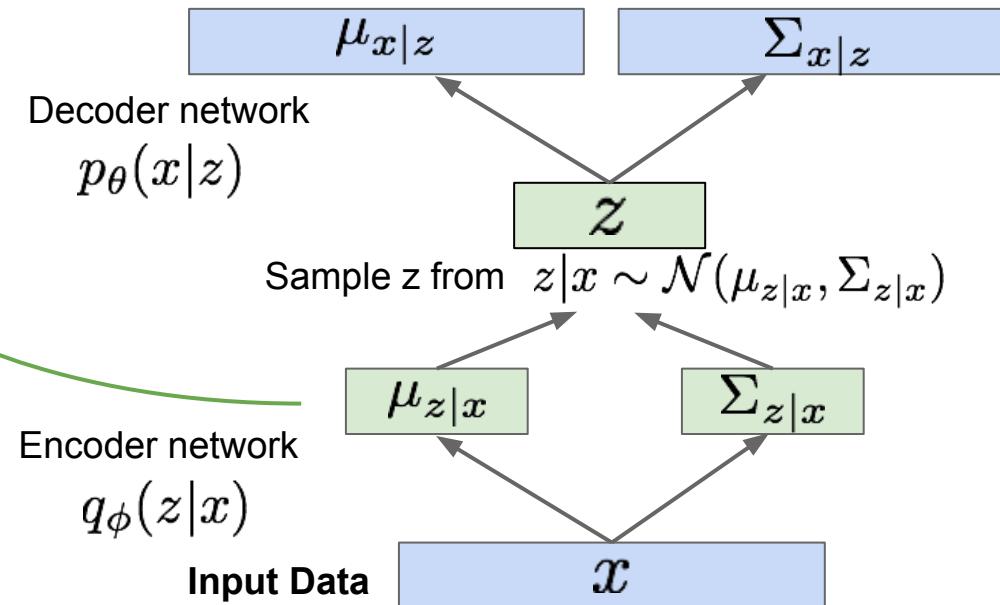


Variational Autoencoders

Putting it all together: maximizing the likelihood lower bound

$$\underbrace{\mathbb{E}_z \left[\log p_\theta(x^{(i)} | z) \right] - D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z))}_{\mathcal{L}(x^{(i)}, \theta, \phi)}$$

Make approximate posterior distribution close to prior



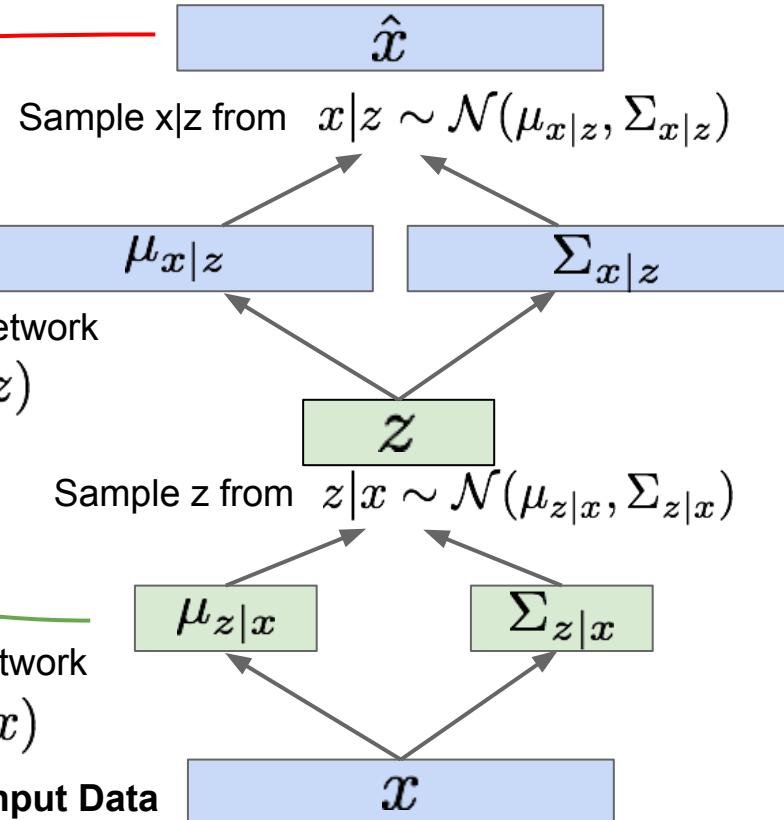
Variational Autoencoders

Putting it all together: maximizing the likelihood lower bound

$$\underbrace{\mathbb{E}_z \left[\log p_\theta(x^{(i)} | z) \right] - D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z))}_{\mathcal{L}(x^{(i)}, \theta, \phi)}$$

Maximize likelihood of original input being reconstructed

Decoder network
 $p_\theta(x|z)$



Make approximate posterior distribution close to prior

Encoder network

$$q_\phi(z|x)$$

Input Data
 x

Variational Autoencoders

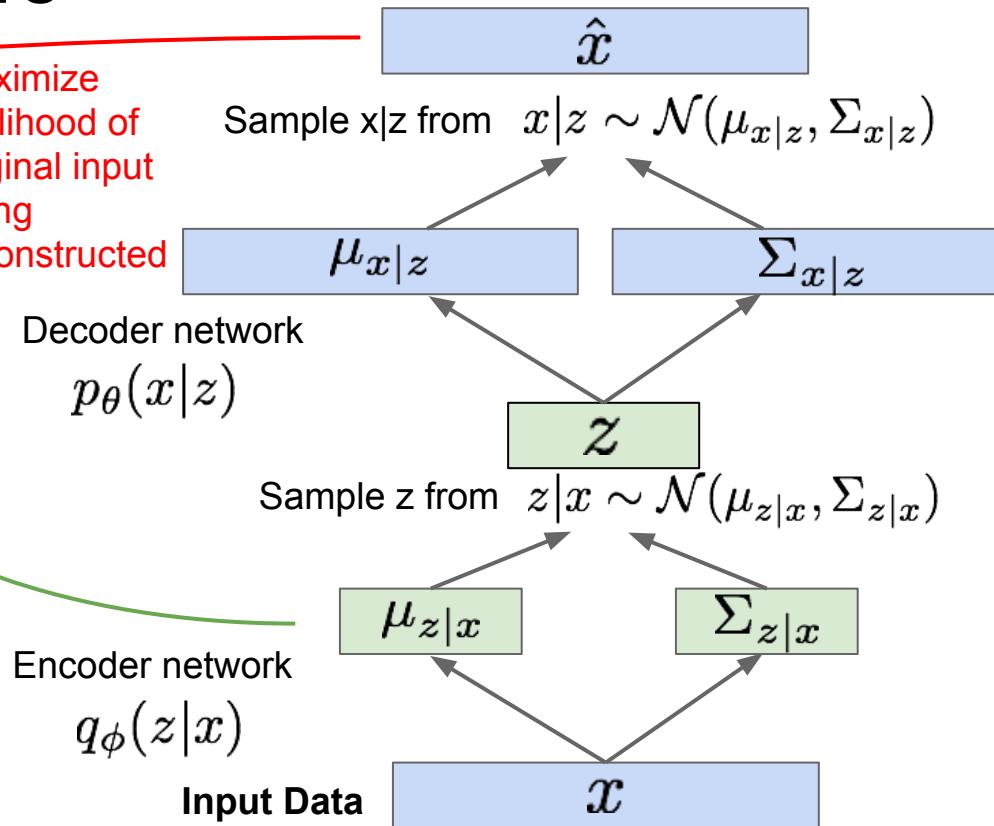
Putting it all together: maximizing the likelihood lower bound

$$\underbrace{\mathbb{E}_z \left[\log p_\theta(x^{(i)} | z) \right] - D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z))}_{\mathcal{L}(x^{(i)}, \theta, \phi)}$$

Maximize likelihood of original input being reconstructed

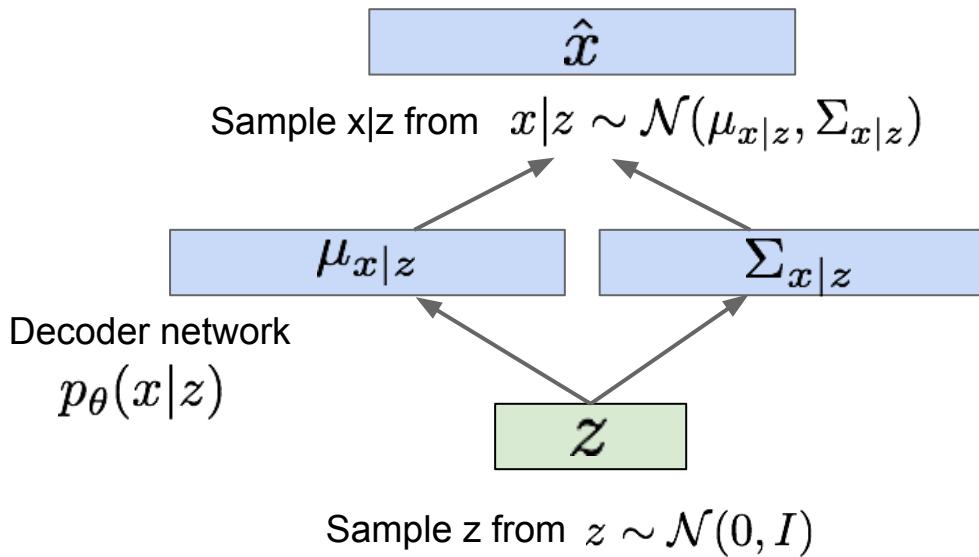
Make approximate posterior distribution close to prior

For every minibatch of input data: compute this forward pass, and then backprop!



Variational Autoencoders: Generating Data!

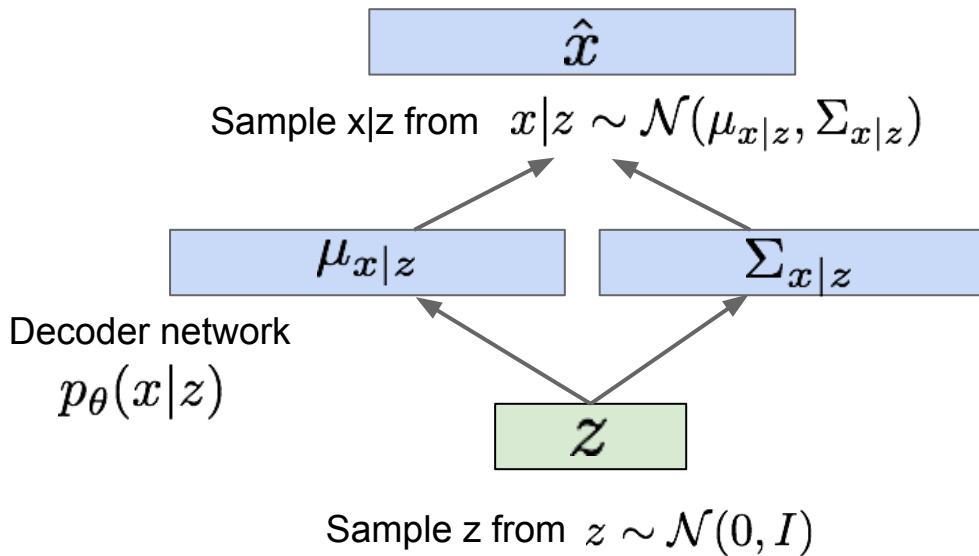
Use decoder network. Now sample z from prior!



Kingma and Welling, “Auto-Encoding Variational Bayes”, ICLR 2014

Variational Autoencoders: Generating Data!

Use decoder network. Now sample z from prior!



Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

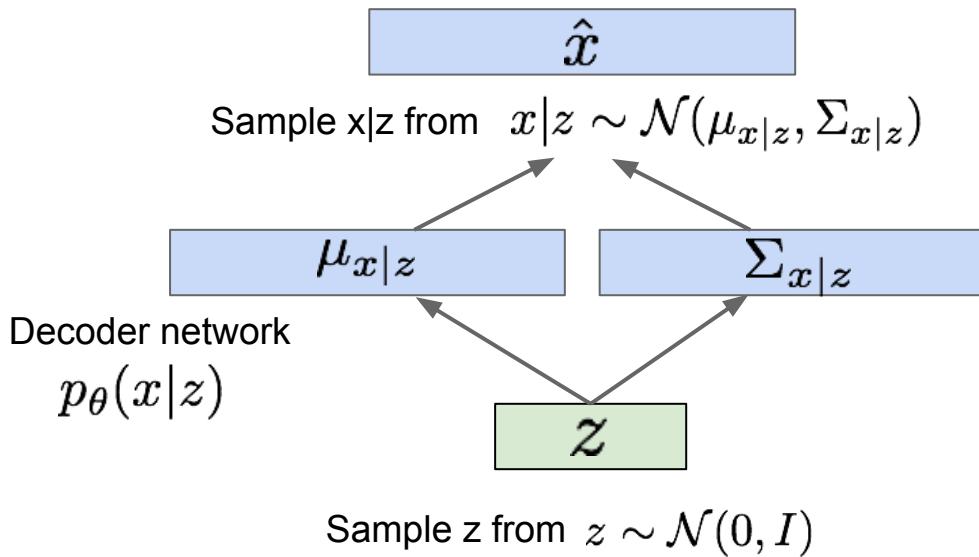
Fei-Fei Li & Justin Johnson & Serena Yeung

Lecture 12 - 92

May 15, 2018

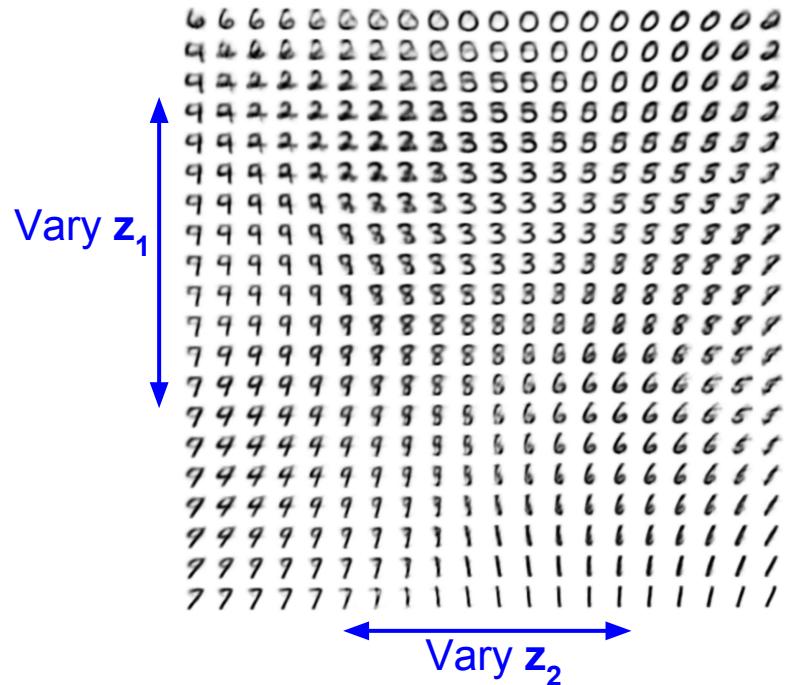
Variational Autoencoders: Generating Data!

Use decoder network. Now sample z from prior!



Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

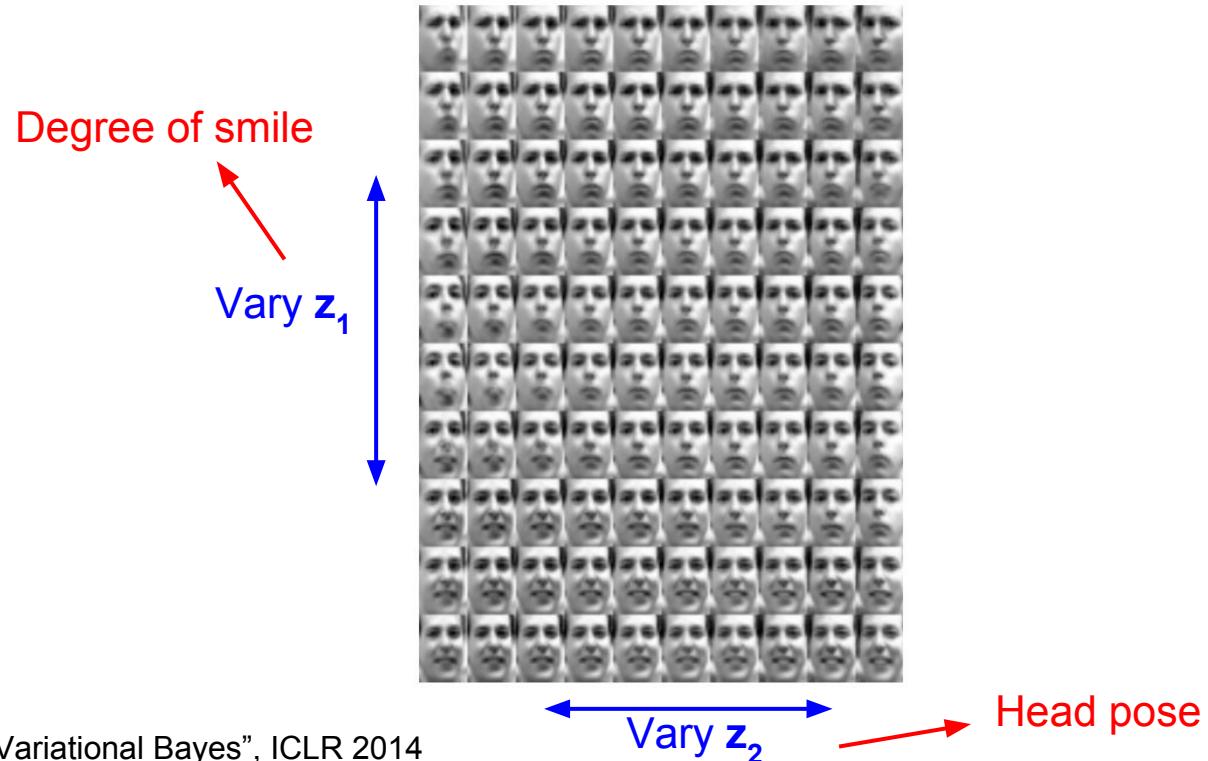
Data manifold for 2-d z



Variational Autoencoders: Generating Data!

Diagonal prior on z
=> independent
latent variables

Different
dimensions of z
encode
interpretable factors
of variation



Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders: Generating Data!

Diagonal prior on \mathbf{z}
=> independent
latent variables

Different
dimensions of \mathbf{z}
encode
interpretable factors
of variation

Also good feature representation that
can be computed using $q_{\phi}(\mathbf{z}|\mathbf{x})$!

Degree of smile
Vary \mathbf{z}_1



Vary \mathbf{z}_2 Head pose

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational Autoencoders: Generating Data!



32x32 CIFAR-10



Labeled Faces in the Wild

Figures copyright (L) Dirk Kingma et al. 2016; (R) Anders Larsen et al. 2017. Reproduced with permission.

Variational Autoencoders

Probabilistic spin to traditional autoencoders => allows generating data

Defines an intractable density => derive and optimize a (variational) lower bound

Pros:

- Principled approach to generative models
- Allows inference of $q(z|x)$, can be useful feature representation for other tasks

Cons:

- Maximizes lower bound of likelihood: okay, but not as good evaluation as PixelRNN/PixelCNN
- Samples blurrier and lower quality compared to state-of-the-art (GANs)

Active areas of research:

- More flexible approximations, e.g. richer approximate posterior instead of diagonal Gaussian
- Incorporating structure in latent variables

Generative Adversarial Networks (GAN)

So far...

PixelCNNs define tractable density function, optimize likelihood of training data:

$$p_{\theta}(x) = \prod_{i=1}^n p_{\theta}(x_i|x_1, \dots, x_{i-1})$$

VAEs define intractable density function with latent \mathbf{z} :

$$p_{\theta}(x) = \int p_{\theta}(z)p_{\theta}(x|z)dz$$

Cannot optimize directly, derive and optimize lower bound on likelihood instead

So far...

PixelCNNs define tractable density function, optimize likelihood of training data:

$$p_{\theta}(x) = \prod_{i=1}^n p_{\theta}(x_i|x_1, \dots, x_{i-1})$$

VAEs define intractable density function with latent \mathbf{z} :

$$p_{\theta}(x) = \int p_{\theta}(z)p_{\theta}(x|z)dz$$

Cannot optimize directly, derive and optimize lower bound on likelihood instead

What if we give up on explicitly modeling density, and just want ability to sample?

So far...

PixelCNNs define tractable density function, optimize likelihood of training data:

$$p_{\theta}(x) = \prod_{i=1}^n p_{\theta}(x_i|x_1, \dots, x_{i-1})$$

VAEs define intractable density function with latent \mathbf{z} :

$$p_{\theta}(x) = \int p_{\theta}(z)p_{\theta}(x|z)dz$$

Cannot optimize directly, derive and optimize lower bound on likelihood instead

What if we give up on explicitly modeling density, and just want ability to sample?

GANs: don't work with any explicit density function!

Instead, take game-theoretic approach: learn to generate from training distribution through 2-player game

Generative Adversarial Networks

Ian Goodfellow et al., “Generative Adversarial Nets”, NIPS 2014

Problem: Want to sample from complex, high-dimensional training distribution. No direct way to do this!

Solution: Sample from a simple distribution, e.g. random noise. Learn transformation to training distribution.

Q: What can we use to represent this complex transformation?

Generative Adversarial Networks

Ian Goodfellow et al., “Generative Adversarial Nets”, NIPS 2014

Problem: Want to sample from complex, high-dimensional training distribution. No direct way to do this!

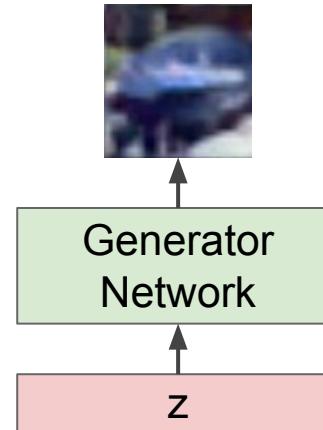
Solution: Sample from a simple distribution, e.g. random noise. Learn transformation to training distribution.

Q: What can we use to represent this complex transformation?

A: A neural network!

Output: Sample from training distribution

Input: Random noise



Training GANs: Two-player game

Ian Goodfellow et al., “Generative Adversarial Nets”, NIPS 2014

Generator network: try to fool the discriminator by generating real-looking images

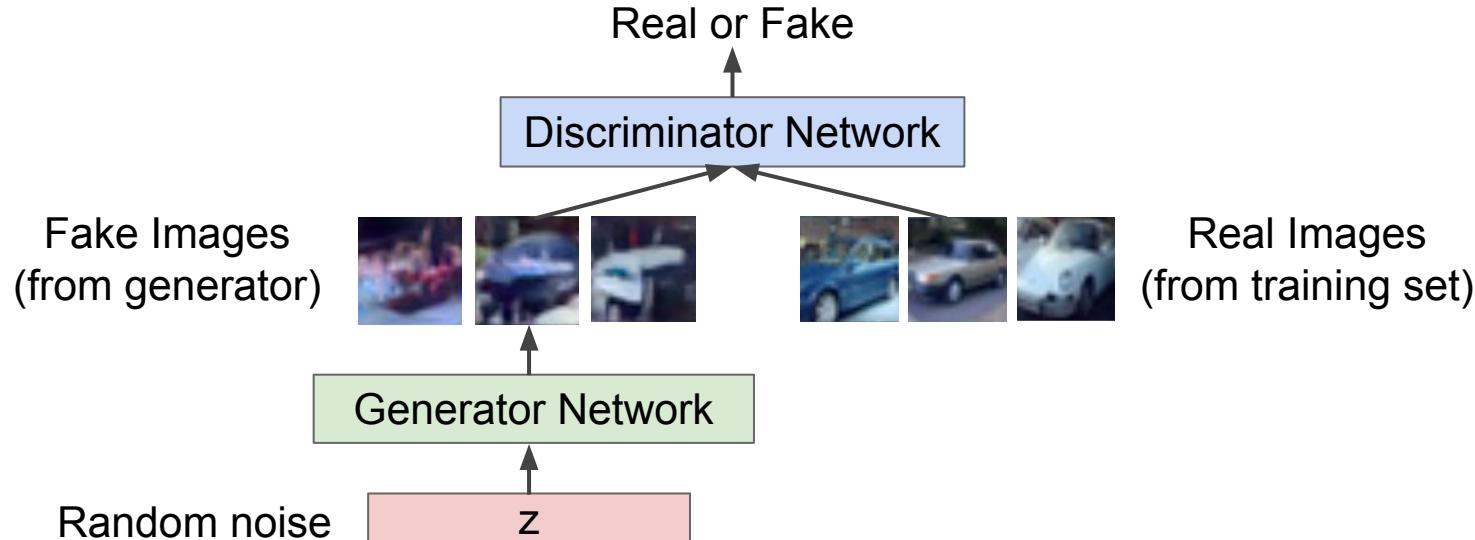
Discriminator network: try to distinguish between real and fake images

Training GANs: Two-player game

Ian Goodfellow et al., “Generative Adversarial Nets”, NIPS 2014

Generator network: try to fool the discriminator by generating real-looking images

Discriminator network: try to distinguish between real and fake images



Fake and real images copyright Emily Denton et al. 2015. Reproduced with permission.

Training GANs: Two-player game

Ian Goodfellow et al., “Generative Adversarial Nets”, NIPS 2014

Generator network: try to fool the discriminator by generating real-looking images

Discriminator network: try to distinguish between real and fake images

Train jointly in **minimax game**

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Training GANs: Two-player game

Ian Goodfellow et al., “Generative Adversarial Nets”, NIPS 2014

Generator network: try to fool the discriminator by generating real-looking images

Discriminator network: try to distinguish between real and fake images

Train jointly in **minimax game**

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\text{Discriminator output for real data } x} + \mathbb{E}_{z \sim p(z)} \log (1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Discriminator output for generated fake data } G(z)}) \right]$$

Discriminator outputs likelihood in (0,1) of real image

Training GANs: Two-player game

Ian Goodfellow et al., “Generative Adversarial Nets”, NIPS 2014

Generator network: try to fool the discriminator by generating real-looking images

Discriminator network: try to distinguish between real and fake images

Train jointly in **minimax game**

Discriminator outputs likelihood in (0,1) of real image

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\text{Discriminator output for real data } x} + \mathbb{E}_{z \sim p(z)} \log(1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Discriminator output for generated fake data } G(z)}) \right]$$

- Discriminator (θ_d) wants to **maximize objective** such that $D(x)$ is close to 1 (real) and $D(G(z))$ is close to 0 (fake)
- Generator (θ_g) wants to **minimize objective** such that $D(G(z))$ is close to 1 (discriminator is fooled into thinking generated $G(z)$ is real)

Training GANs: Two-player game

Ian Goodfellow et al., “Generative Adversarial Nets”, NIPS 2014

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Alternate between:

1. **Gradient ascent** on discriminator

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

2. **Gradient descent** on generator

$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

Training GANs: Two-player game

Ian Goodfellow et al., “Generative Adversarial Nets”, NIPS 2014

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Alternate between:

1. **Gradient ascent** on discriminator

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

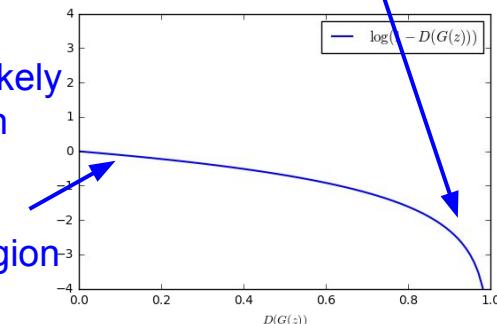
2. **Gradient descent** on generator

$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

In practice, optimizing this generator objective does not work well!

When sample is likely fake, want to learn from it to improve generator. But gradient in this region is relatively flat!

Gradient signal dominated by region where sample is already good



Training GANs: Two-player game

Ian Goodfellow et al., “Generative Adversarial Nets”, NIPS 2014

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Alternate between:

1. **Gradient ascent** on discriminator

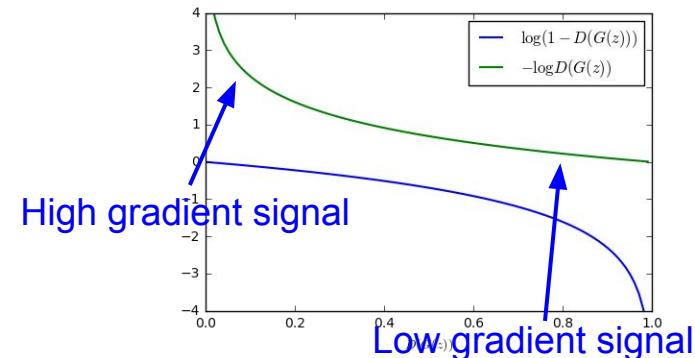
$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

2. Instead: **Gradient ascent** on generator, different objective

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z)))$$

Instead of minimizing likelihood of discriminator being correct, now maximize likelihood of discriminator being wrong.

Same objective of fooling discriminator, but now higher gradient signal for bad samples => works much better! Standard in practice.



Training GANs: Two-player game

Ian Goodfellow et al., “Generative Adversarial Nets”, NIPS 2014

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Alternate between:

1. **Gradient ascent** on discriminator

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

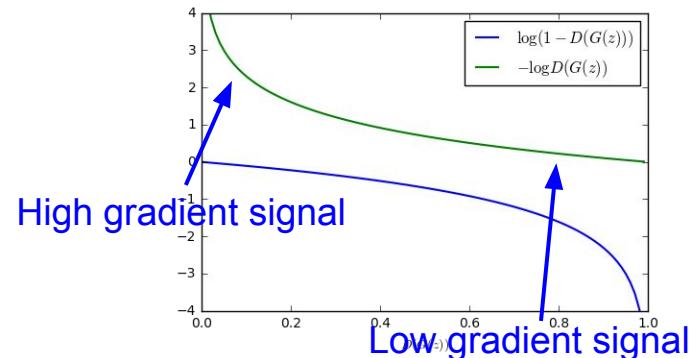
2. Instead: **Gradient ascent** on generator, different objective

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z)))$$

Instead of minimizing likelihood of discriminator being correct, now maximize likelihood of discriminator being wrong.

Same objective of fooling discriminator, but now higher gradient signal for bad samples => works much better! Standard in practice.

Aside: Jointly training two networks is challenging, can be unstable. Choosing objectives with better loss landscapes helps training, is an active area of research.



Training GANs: Two-player game

Ian Goodfellow et al., “Generative Adversarial Nets”, NIPS 2014

Putting it together: GAN training algorithm

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Sample minibatch of m examples $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\mathbf{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D_{\theta_d}(\mathbf{x}^{(i)}) + \log(1 - D_{\theta_d}(G_{\theta_g}(\mathbf{z}^{(i)}))) \right]$$

end for

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Update the generator by ascending its stochastic gradient (improved objective):

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log(D_{\theta_d}(G_{\theta_g}(\mathbf{z}^{(i)})))$$

end for

Training GANs: Two-player game

Ian Goodfellow et al., “Generative Adversarial Nets”, NIPS 2014

Putting it together: GAN training algorithm

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Sample minibatch of m examples $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\mathbf{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D_{\theta_d}(\mathbf{x}^{(i)}) + \log(1 - D_{\theta_d}(G_{\theta_g}(\mathbf{z}^{(i)}))) \right]$$

end for

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Update the generator by ascending its stochastic gradient (improved objective):

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log(D_{\theta_d}(G_{\theta_g}(\mathbf{z}^{(i)})))$$

end for

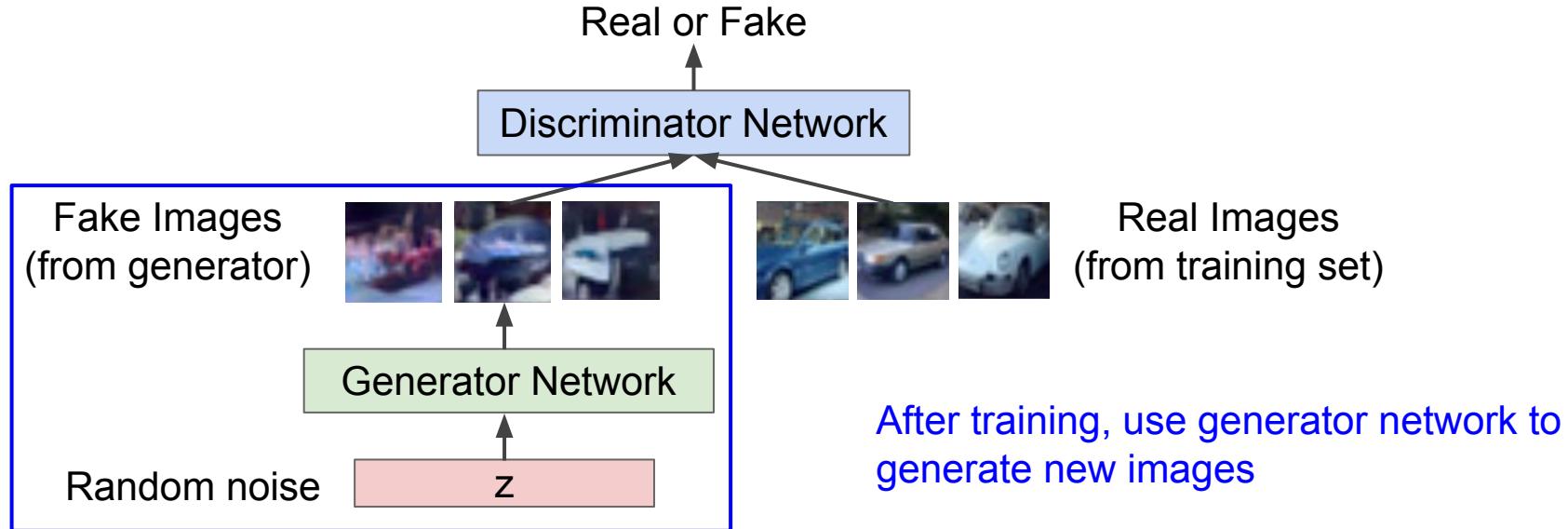
Some find $k=1$ more stable, others use $k > 1$, no best rule.

Recent work (e.g. Wasserstein GAN) alleviates this problem, better stability!

Training GANs: Two-player game

Ian Goodfellow et al., “Generative Adversarial Nets”, NIPS 2014

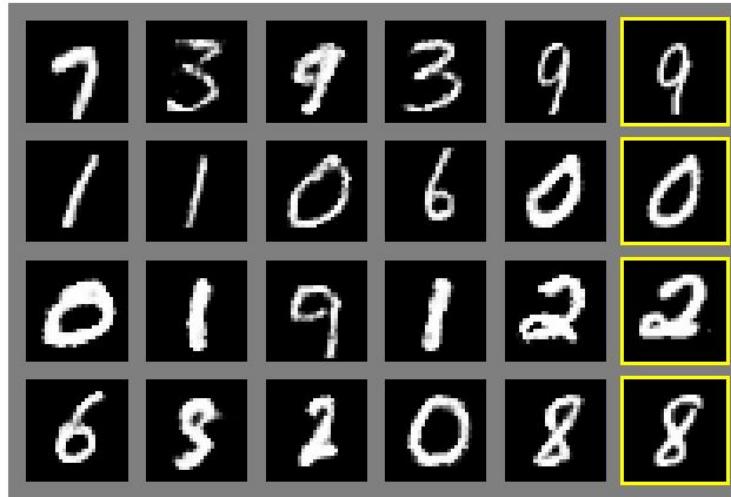
Generator network: try to fool the discriminator by generating real-looking images
Discriminator network: try to distinguish between real and fake images



Fake and real images copyright Emily Denton et al. 2015. Reproduced with permission.

Generative Adversarial Nets

Generated samples

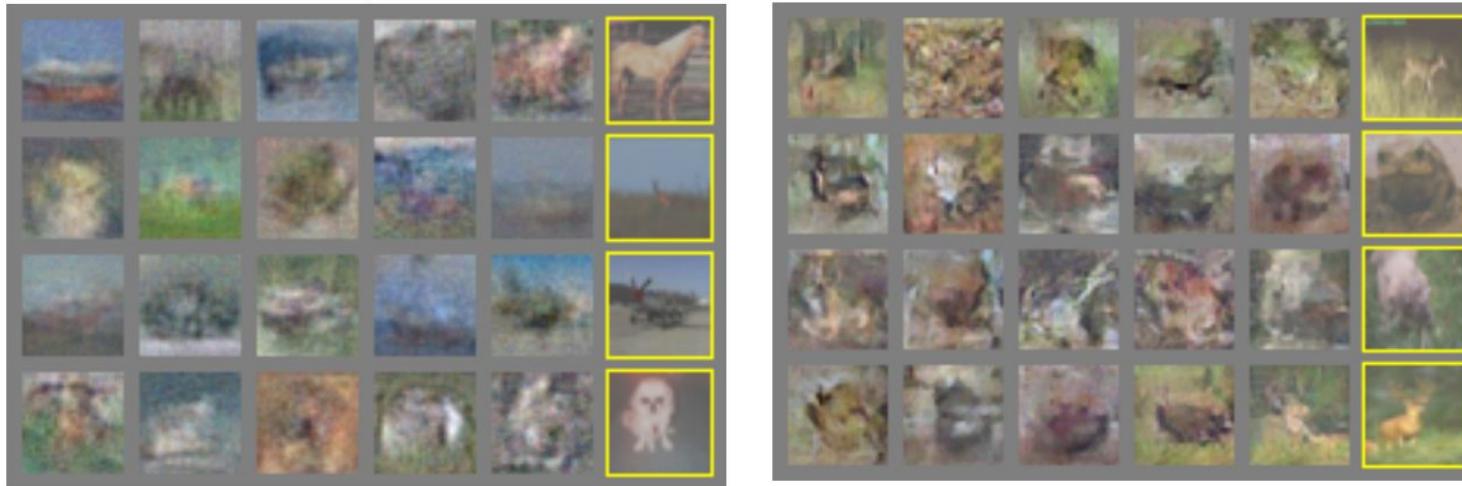


Nearest neighbor from training set

Figures copyright Ian Goodfellow et al., 2014. Reproduced with permission.

Generative Adversarial Nets

Generated samples (CIFAR-10)



Nearest neighbor from training set

Figures copyright Ian Goodfellow et al., 2014. Reproduced with permission.

Generative Adversarial Nets: Convolutional Architectures

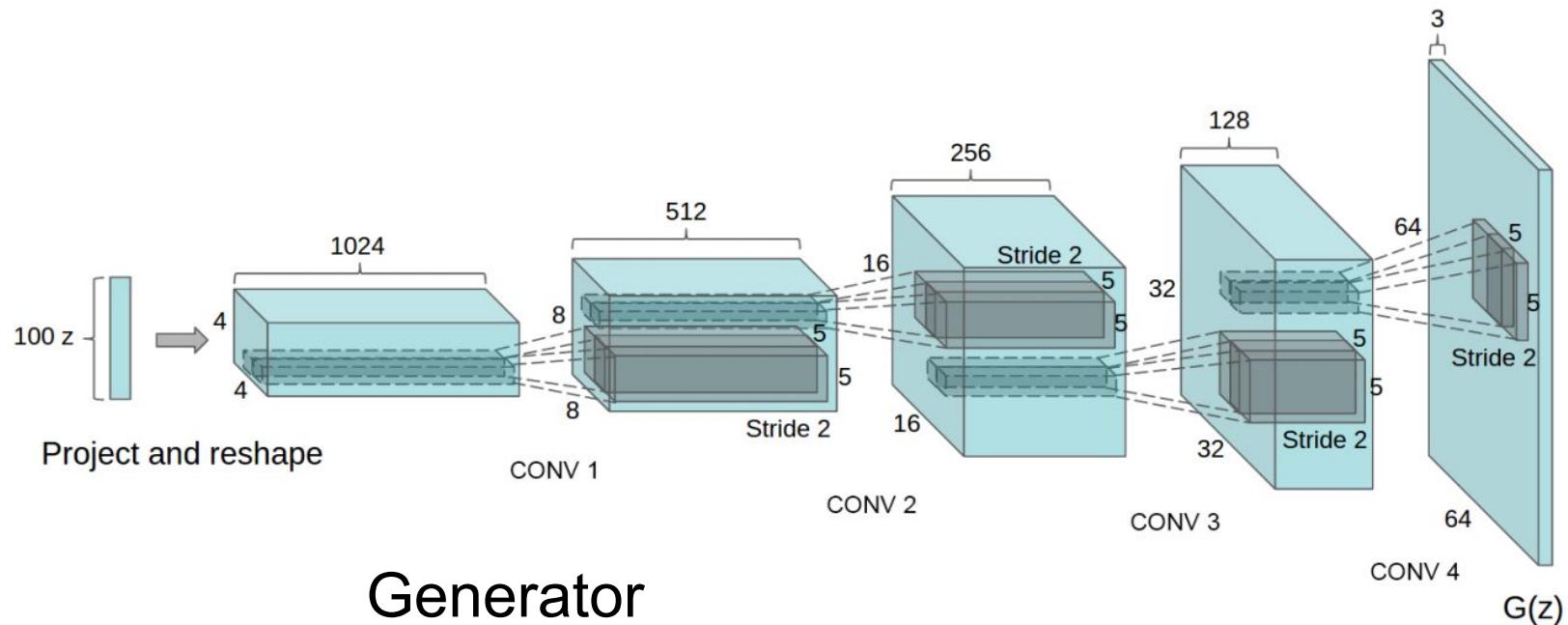
Generator is an upsampling network with fractionally-strided convolutions
Discriminator is a convolutional network

Architecture guidelines for stable Deep Convolutional GANs

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.

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

Generative Adversarial Nets: Convolutional Architectures



Generator

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

Generative Adversarial Nets: Convolutional Architectures

Samples
from the
model look
much
better!



Radford et al,
ICLR 2016

Generative Adversarial Nets: Convolutional Architectures

Interpolating
between
random
points in latent
space



Radford et al,
ICLR 2016

Generative Adversarial Nets: Interpretable Vector Math

Smiling woman



Neutral woman



Neutral man



Samples
from the
model

Radford et al, ICLR 2016

Generative Adversarial Nets: Interpretable Vector Math

Radford et al, ICLR 2016

Smiling woman Neutral woman Neutral man

Samples
from the
model



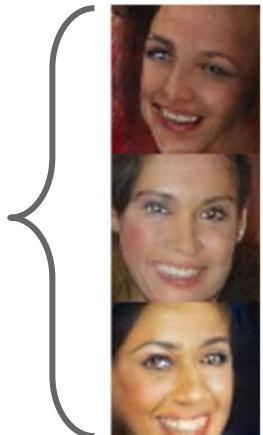
Average Z
vectors, do
arithmetic



Generative Adversarial Nets: Interpretable Vector Math

Smiling woman Neutral woman Neutral man

Samples
from the
model



Average Z
vectors, do
arithmetic



Radford et al, ICLR 2016

Smiling Man

Generative Adversarial Nets: Interpretable Vector Math

Glasses man



Radford et al,
ICLR 2016

No glasses man



No glasses woman



Generative Adversarial Nets: Interpretable Vector Math

Glasses man



No glasses man



No glasses woman



Radford et al,
ICLR 2016

Woman with glasses



2017: Explosion of GANs

“The GAN Zoo”

- GAN - Generative Adversarial Networks
- 3D-GAN - Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling
- acGAN - Face Aging With Conditional Generative Adversarial Networks
- AC-GAN - Conditional Image Synthesis With Auxiliary Classifier GANs
- AdAGAN - AdaGAN: Boosting Generative Models
- AEGAN - Learning Inverse Mapping by Autoencoder based Generative Adversarial Nets
- AffGAN - Amortised MAP Inference for Image Super-resolution
- AL-CGAN - Learning to Generate Images of Outdoor Scenes from Attributes and Semantic Layouts
- ALI - Adversarially Learned Inference
- AM-GAN - Generative Adversarial Nets with Labeled Data by Activation Maximization
- AnoGAN - Unsupervised Anomaly Detection with Generative Adversarial Networks to Guide Marker Discovery
- ArtGAN - ArtGAN: Artwork Synthesis with Conditional Categorical GANs
- b-GAN - b-GAN: Unified Framework of Generative Adversarial Networks
- Bayesian GAN - Deep and Hierarchical Implicit Models
- BEGAN - BEGAN: Boundary Equilibrium Generative Adversarial Networks
- BiGAN - Adversarial Feature Learning
- BS-GAN - Boundary-Seeking Generative Adversarial Networks
- CGAN - Conditional Generative Adversarial Nets
- CaloGAN - CaloGAN: Simulating 3D High Energy Particle Showers in Multi-Layer Electromagnetic Calorimeters with Generative Adversarial Networks
- CCGAN - Semi-Supervised Learning with Context-Conditional Generative Adversarial Networks
- CatGAN - Unsupervised and Semi-supervised Learning with Categorical Generative Adversarial Networks
- CoGAN - Coupled Generative Adversarial Networks
- Context-RNN-GAN - Contextual RNN-GANs for Abstract Reasoning Diagram Generation
- C-RNN-GAN - C-RNN-GAN: Continuous recurrent neural networks with adversarial training
- CS-GAN - Improving Neural Machine Translation with Conditional Sequence Generative Adversarial Nets
- CVAE-GAN - CVAE-GAN: Fine-Grained Image Generation through Asymmetric Training
- CycleGAN - Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks
- DTN - Unsupervised Cross-Domain Image Generation
- DCGAN - Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks
- DiscoGAN - Learning to Discover Cross-Domain Relations with Generative Adversarial Networks
- DR-GAN - Disentangled Representation Learning GAN for Pose-Invariant Face Recognition
- DualGAN - DualGAN: Unsupervised Dual Learning for Image-to-Image Translation
- EBGAN - Energy-based Generative Adversarial Network
- f-GAN - f-GAN: Training Generative Neural Samplers using Variational Divergence Minimization
- FF-GAN - Towards Large-Pose Face Frontalization in the Wild
- GAWWN - Learning What and Where to Draw
- GeneGAN - GeneGAN: Learning Object Transfiguration and Attribute Subspace from Unpaired Data
- Geometric GAN - Geometric GAN
- GoGAN - Gang of GANs: Generative Adversarial Networks with Maximum Margin Ranking
- GP-GAN - GP-GAN: Towards Realistic High-Resolution Image Blending
- IAN - Neural Photo Editing with Introspective Adversarial Networks
- iGAN - Generative Visual Manipulation on the Natural Image Manifold
- IcGAN - Invertible Conditional GANs for image editing
- ID-CGAN - Image De-raining Using a Conditional Generative Adversarial Network
- Improved GAN - Improved Techniques for Training GANs
- InfoGAN - InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets
- LAGAN - Learning Particle Physics by Example: Location-Aware Generative Adversarial Networks for Physics Synthesis
- LAPGAN - Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks

<https://github.com/hindupuravinash/the-gan-zoo>

2017: Explosion of GANs

“The GAN Zoo”

- GAN - Generative Adversarial Networks
- 3D-GAN - Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling
- acGAN - Face Aging With Conditional Generative Adversarial Networks
- AC-GAN - Conditional Image Synthesis With Auxiliary Classifier GANs
- AdAGAN - AdaGAN: Boosting Generative Models
- AEGAN - Learning Inverse Mapping by Autoencoder based Generative Adversarial Nets
- AffGAN - Amortised MAP Inference for Image Super-resolution
- AL-CGAN - Learning to Generate Images of Outdoor Scenes from Attributes and Semantic Layouts
- ALI - Adversarially Learned Inference
- AM-GAN - Generative Adversarial Nets with Labeled Data by Activation Maximization
- AnoGAN - Unsupervised Anomaly Detection with Generative Adversarial Networks to Guide Marker Discovery
- ArtGAN - ArtGAN: Artwork Synthesis with Conditional Categorical GANs
- b-GAN - b-GAN: Unified Framework of Generative Adversarial Networks
- Bayesian GAN - Deep and Hierarchical Implicit Models
- BEGAN - BEGAN: Boundary Equilibrium Generative Adversarial Networks
- BiGAN - Adversarial Feature Learning
- BS-GAN - Boundary-Seeking Generative Adversarial Networks
- CGAN - Conditional Generative Adversarial Nets
- CaloGAN - CaloGAN: Simulating 3D High Energy Particle Showers in Multi-Layer Electromagnetic Calorimeters with Generative Adversarial Networks
- CCGAN - Semi-Supervised Learning with Context-Conditional Generative Adversarial Networks
- CatGAN - Unsupervised and Semi-supervised Learning with Categorical Generative Adversarial Networks
- CoGAN - Coupled Generative Adversarial Networks

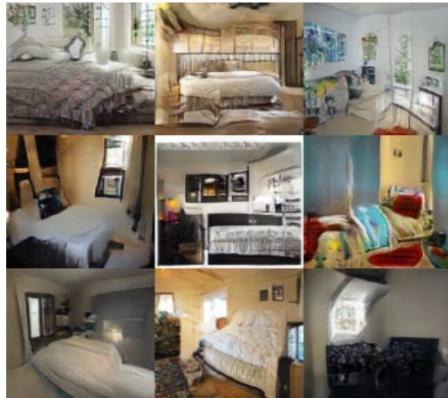
See also: <https://github.com/soumith/ganhacks> for tips and tricks for trainings GANs

- Context-RNN-GAN - Contextual RNN-GANs for Abstract Reasoning Diagram Generation
- C-RNN-GAN - C-RNN-GAN: Continuous recurrent neural networks with adversarial training
- CS-GAN - Improving Neural Machine Translation with Conditional Sequence Generative Adversarial Nets
- CVAE-GAN - CVAE-GAN: Fine-Grained Image Generation through Asymmetric Training
- CycleGAN - Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks
- DTN - Unsupervised Cross-Domain Image Generation
- DCGAN - Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks
- DiscoGAN - Learning to Discover Cross-Domain Relations with Generative Adversarial Networks
- DR-GAN - Disentangled Representation Learning GAN for Pose-Invariant Face Recognition
- DualGAN - DualGAN: Unsupervised Dual Learning for Image-to-Image Translation
- EBGAN - Energy-based Generative Adversarial Network
- f-GAN - f-GAN: Training Generative Neural Samplers using Variational Divergence Minimization
- FF-GAN - Towards Large-Pose Face Frontalization in the Wild
- GAWWN - Learning What and Where to Draw
- GeneGAN - GeneGAN: Learning Object Transfiguration and Attribute Subspace from Unpaired Data
- Geometric GAN - Geometric GAN
- GoGAN - Gang of GANs: Generative Adversarial Networks with Maximum Margin Ranking
- GP-GAN - GP-GAN: Towards Realistic High-Resolution Image Blending
- IAN - Neural Photo Editing with Introspective Adversarial Networks
- iGAN - Generative Visual Manipulation on the Natural Image Manifold
- IcGAN - Invertible Conditional GANs for image editing
- ID-CGAN - Image De-raining Using a Conditional Generative Adversarial Network
- Improved GAN - Improved Techniques for Training GANs
- InfoGAN - InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets
- LAGAN - Learning Particle Physics by Example: Location-Aware Generative Adversarial Networks for Physics Synthesis
- LAPGAN - Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks

<https://github.com/hindupuravinash/the-gan-zoo>

2017: Explosion of GANs

Better training and generation



LSGAN, Zhu 2017.



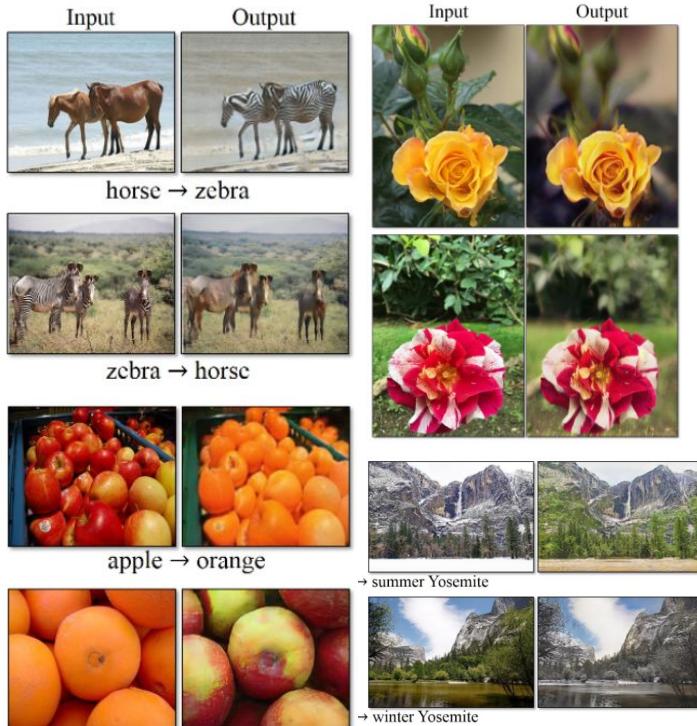
Wasserstein GAN,
Arjovsky 2017.
Improved Wasserstein
GAN, Gulrajani 2017.



Progressive GAN, Karras 2018.

2017: Explosion of GANs

Source->Target domain transfer



CycleGAN. Zhu et al. 2017.

Text -> Image Synthesis

this small bird has a pink breast and crown, and black primaries and secondaries.

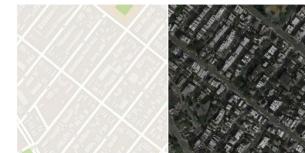
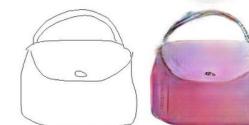


this magnificent fellow is almost all black with a red crest, and white cheek patch.



Reed et al. 2017.

Many GAN applications



Pix2pix. Isola 2017. Many examples at <https://phillipi.github.io/pix2pix/>

GANs

Don't work with an explicit density function

Take game-theoretic approach: learn to generate from training distribution through 2-player game

Pros:

- Beautiful, state-of-the-art samples!

Cons:

- Trickier / more unstable to train
- Can't solve inference queries such as $p(x)$, $p(z|x)$

Active areas of research:

- Better loss functions, more stable training (Wasserstein GAN, LSGAN, many others)
- Conditional GANs, GANs for all kinds of applications

Recap

Generative Models

- PixelRNN and PixelCNN Explicit density model, optimizes exact likelihood, good samples. But inefficient sequential generation.
- Variational Autoencoders (VAE) Optimize variational lower bound on likelihood. Useful latent representation, inference queries. But current sample quality not the best.
- Generative Adversarial Networks (GANs) Game-theoretic approach, best samples! But can be tricky and unstable to train, no inference queries.

Recap

Generative Models

- PixelRNN and PixelCNN Explicit density model, optimizes exact likelihood, good samples. But inefficient sequential generation.
- Variational Autoencoders (VAE) Optimize variational lower bound on likelihood. Useful latent representation, inference queries. But current sample quality not the best.
- Generative Adversarial Networks (GANs) Game-theoretic approach, best samples! But can be tricky and unstable to train, no inference queries.

Also recent work in combinations of these types of models! E.g. Adversarial Autoencoders (Makhzani 2015) and PixelVAE (Gulrajani 2016)

Recap

Generative Models

- PixelRNN and PixelCNN Explicit density model, optimizes exact likelihood, good samples. But inefficient sequential generation.
- Variational Autoencoders (VAE) Optimize variational lower bound on likelihood. Useful latent representation, inference queries. But current sample quality not the best.
- Generative Adversarial Networks (GANs) Game-theoretic approach, best samples! But can be tricky and unstable to train, no inference queries.

Next time: Visualizing and understanding networks

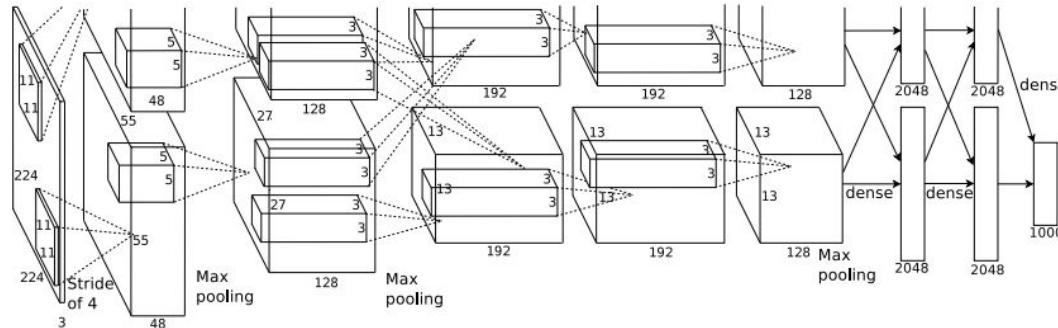
Lecture 13: Visualizing and Understanding

Administrative

- Project milestone was due yesterday 5/16
- Assignment 3 due a week from yesterday 5/23
- Google Cloud:
 - See pinned Piazza post about GPUs
 - Make a private post on Piazza if you need more credits

What's going on inside ConvNets?

This image is CC0 public domain



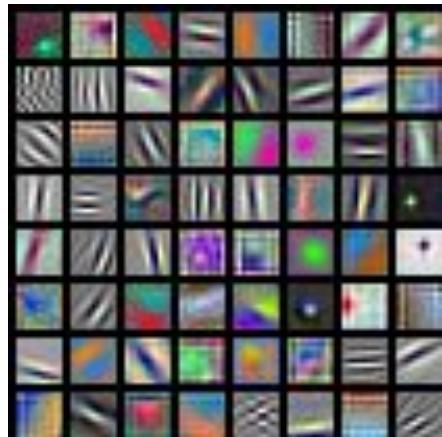
Input Image:
3 x 224 x 224

What are the intermediate features looking for?

Class Scores:
1000 numbers

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

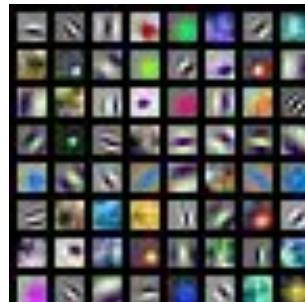
First Layer: Visualize Filters



AlexNet:
 $64 \times 3 \times 11 \times 11$



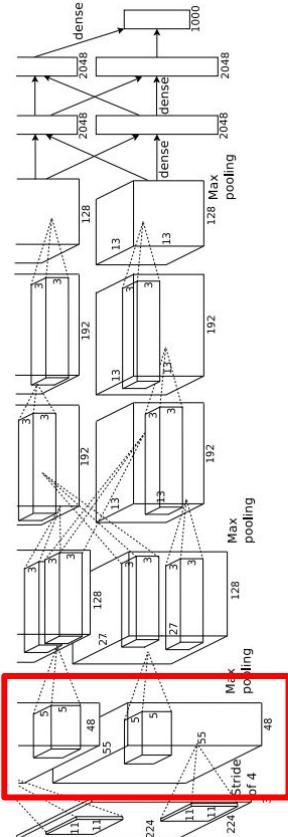
ResNet-18:
 $64 \times 3 \times 7 \times 7$



ResNet-101:
 $64 \times 3 \times 7 \times 7$



DenseNet-121:
 $64 \times 3 \times 7 \times 7$



Krizhevsky, "One weird trick for parallelizing convolutional neural networks", arXiv 2014

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

Huang et al, "Densely Connected Convolutional Networks", CVPR 2017

Visualize the filters/kernels (raw weights)

We can visualize filters at higher layers, but not that interesting

(these are taken from ConvNetJS
CIFAR-10 demo)

Weights:


layer 1 weights

$16 \times 3 \times 7 \times 7$

Weights:


layer 2 weights

$20 \times 16 \times 7 \times 7$

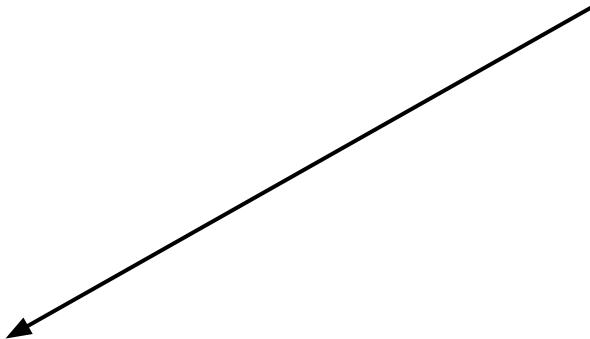
Weights:


layer 3 weights

$20 \times 20 \times 7 \times 7$

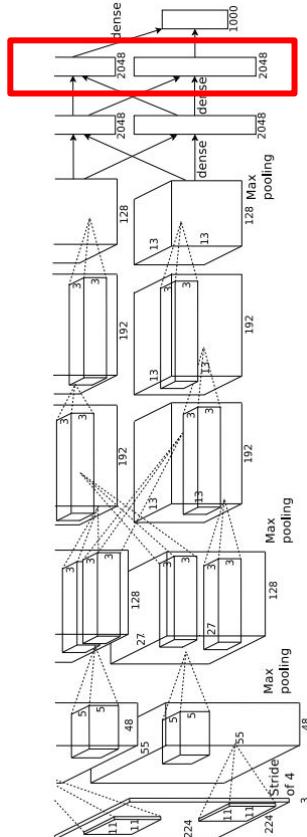
Last Layer

FC7 layer



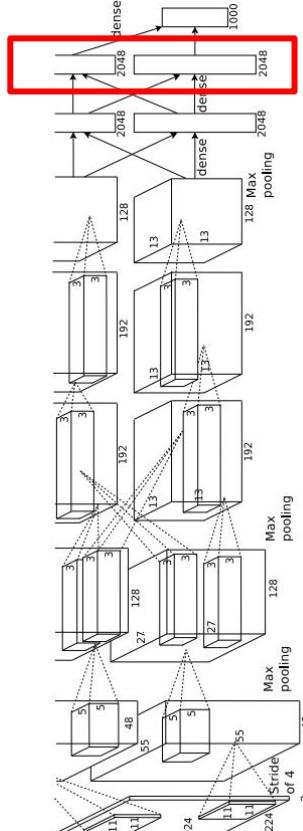
4096-dimensional feature vector for an image
(layer immediately before the classifier)

Run the network on many images, collect the
feature vectors



Last Layer: Nearest Neighbors

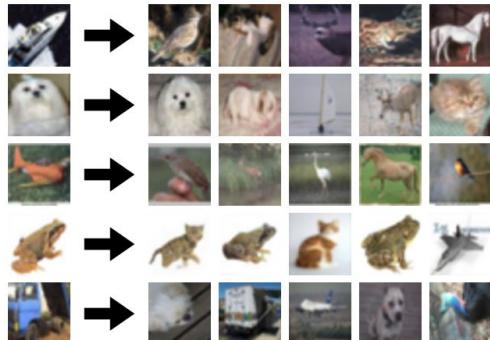
4096-dim vector



Test image L2 Nearest neighbors in feature space



Recall: Nearest neighbors
in pixel space



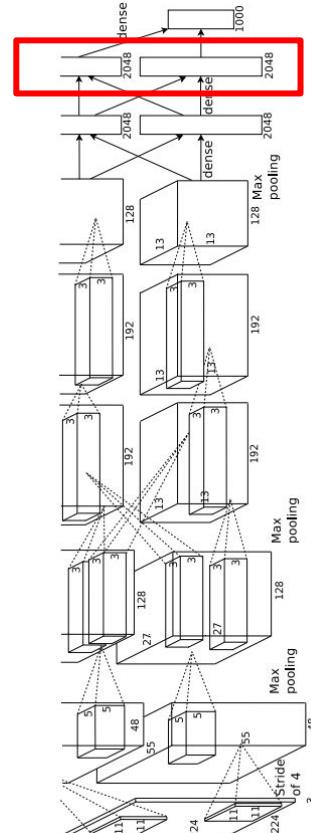
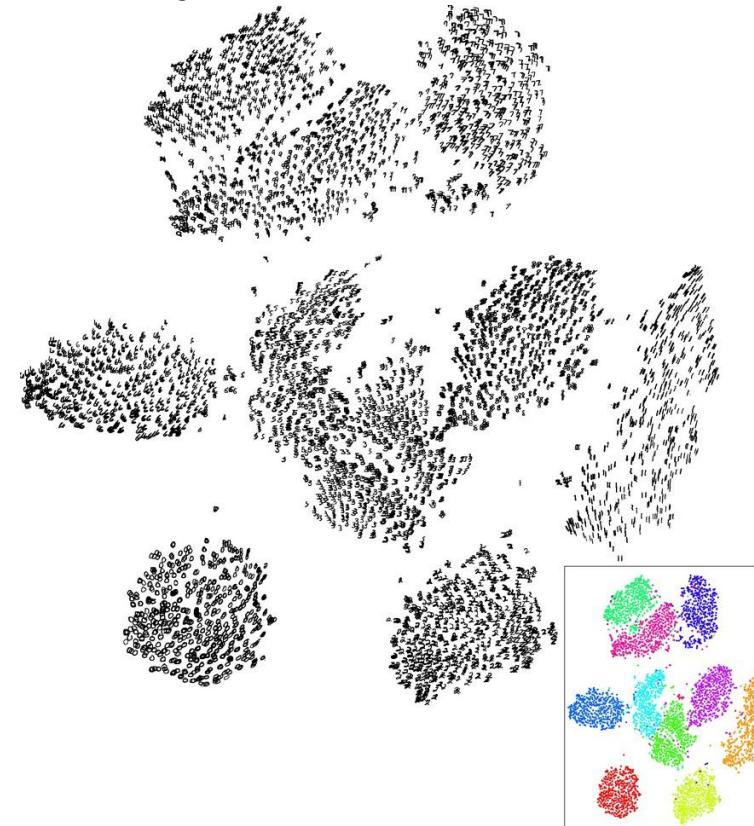
Krizhevsky et al, "ImageNet Classification with Deep Convolutional Neural Networks", NIPS 2012.
Figures reproduced with permission.

Last Layer: Dimensionality Reduction

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

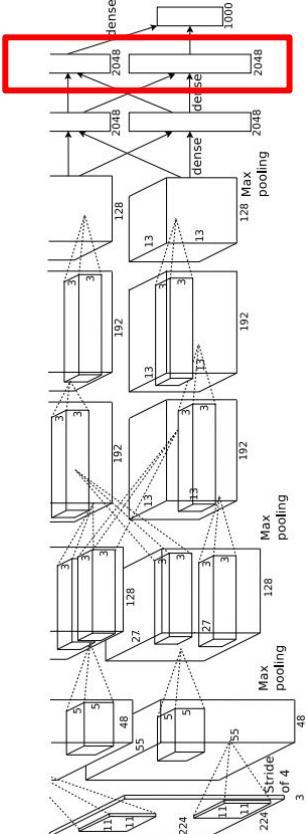
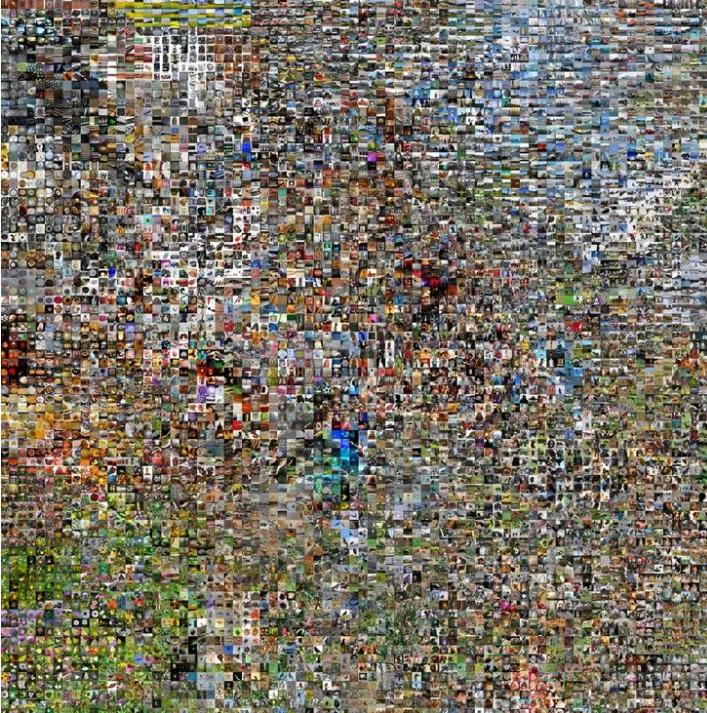
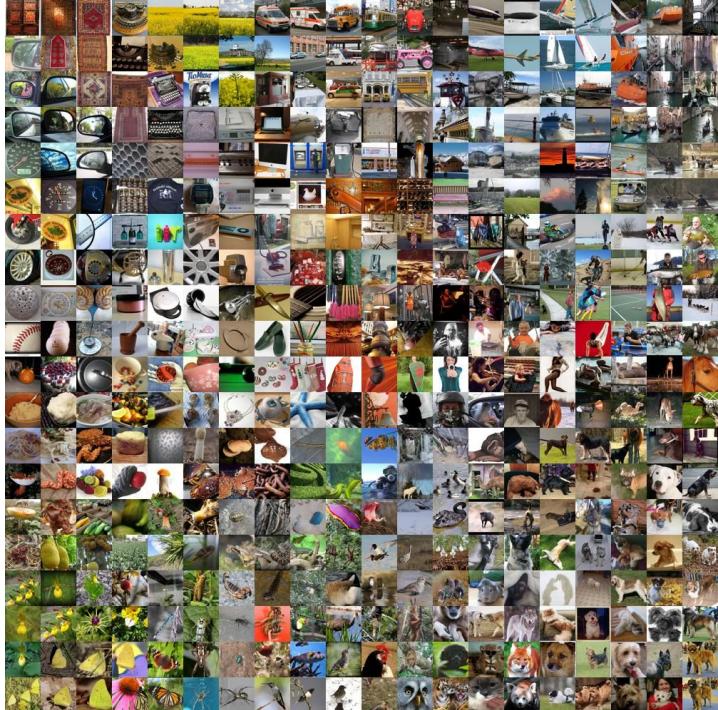
Simple algorithm: Principal Component Analysis (PCA)

More complex: t-SNE



Van der Maaten and Hinton, "Visualizing Data using t-SNE", JMLR 2008
Figure copyright Laurens van der Maaten and Geoff Hinton, 2008. Reproduced with permission.

Last Layer: Dimensionality Reduction

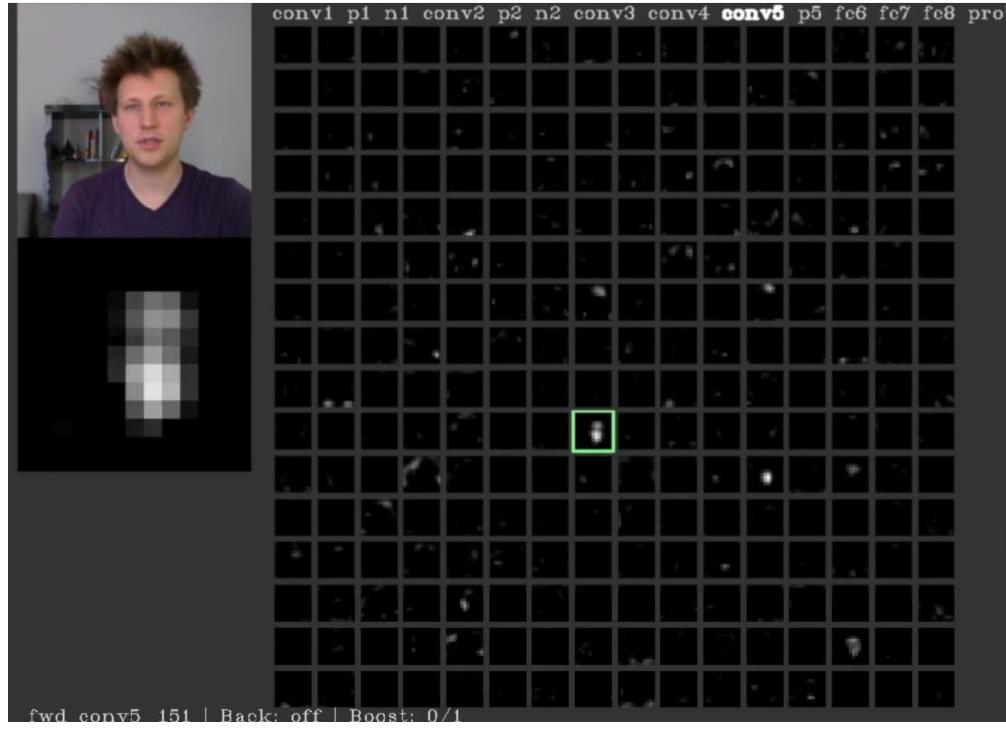


Van der Maaten and Hinton, "Visualizing Data using t-SNE", JMLR 2008
Krizhevsky et al, "ImageNet Classification with Deep Convolutional Neural Networks", NIPS 2012.
Figure reproduced with permission.

See high-resolution versions at
<http://cs.stanford.edu/people/karpathy/cnnembed/>

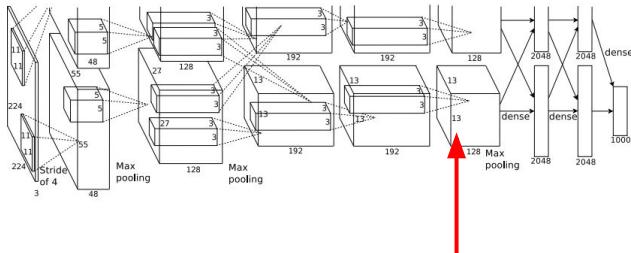
Visualizing Activations

conv5 feature map is
128x13x13; visualize
as 128 13x13
grayscale images



Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML DL Workshop 2014.
Figure copyright Jason Yosinski, 2014. Reproduced with permission.

Maximally Activating Patches



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

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

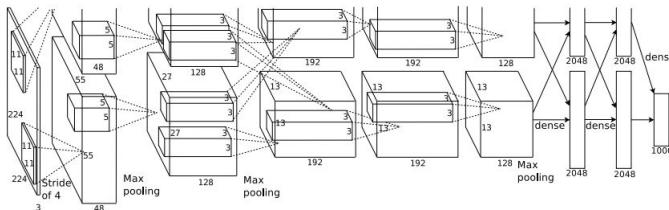
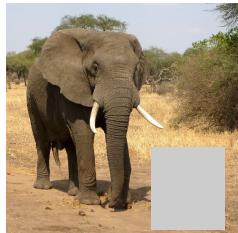
Visualize image patches that correspond
to maximal activations



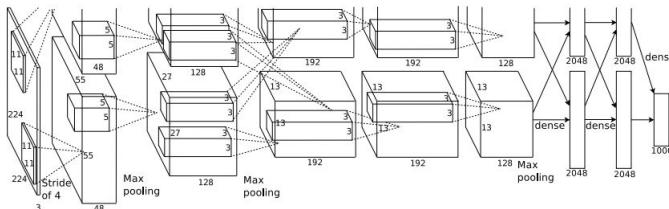
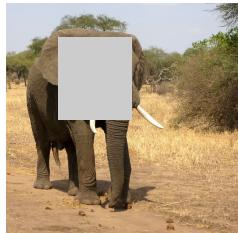
Springenberg et al, "Striving for Simplicity: The All Convolutional Net", ICLR Workshop 2015
Figure copyright Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller, 2015;
reproduced with permission.

Which pixels matter: Saliency vs Occlusion

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



$$P(\text{elephant}) = 0.95$$



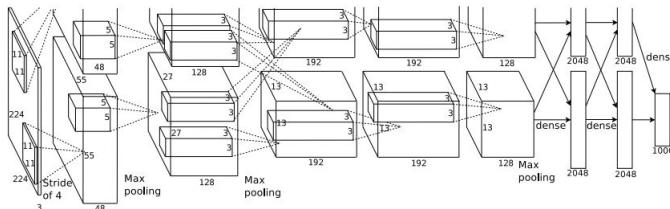
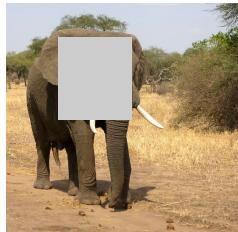
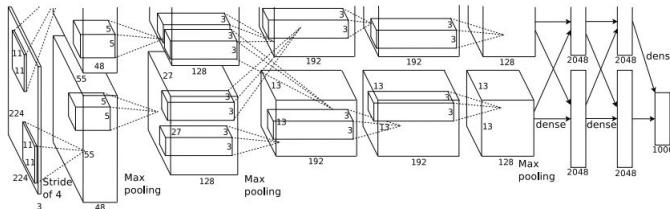
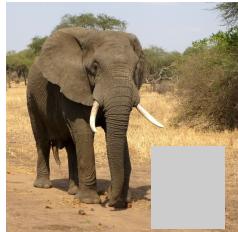
$$P(\text{elephant}) = 0.75$$

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

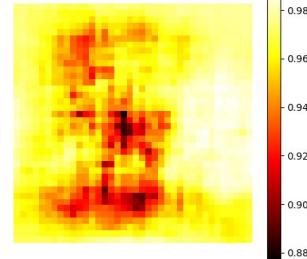
Boat image is CC0 public domain
Elephant image is CC0 public domain
Go-Karts image is CC0 public domain

Which pixels matter: Saliency vs Occlusion

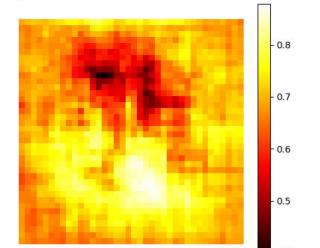
Mask part of the image before feeding to CNN,
check how much predicted probabilities change



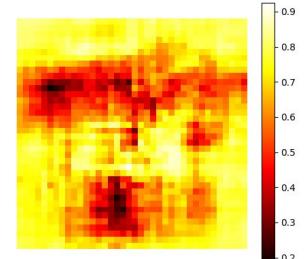
schooner



African elephant, *Loxodonta africana*



go-kart

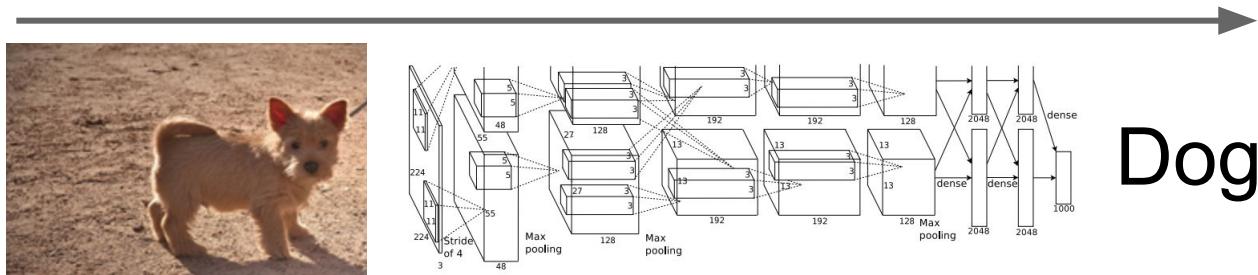


Boat image is CC0 public domain
Elephant image is CC0 public domain
Go-Karts image is CC0 public domain

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

Which pixels matter: Saliency via Backprop

Forward pass: Compute probabilities

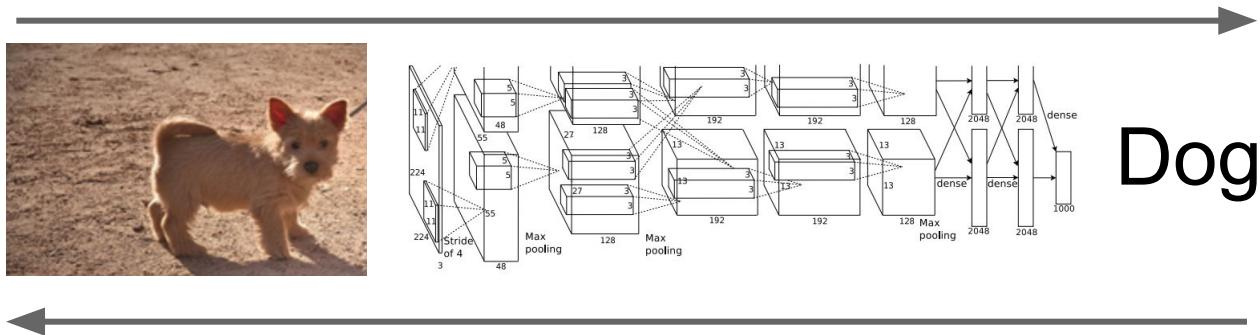


Simonyan, Vedaldi, and Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.

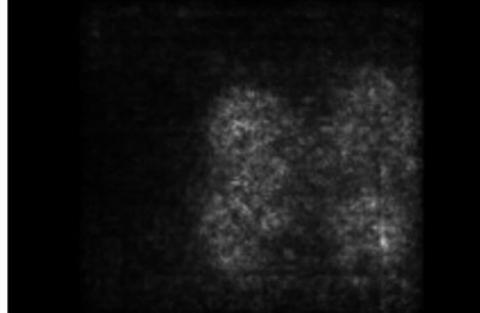
Figures copyright Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman, 2014; reproduced with permission.

Which pixels matter: Saliency via Backprop

Forward pass: Compute probabilities



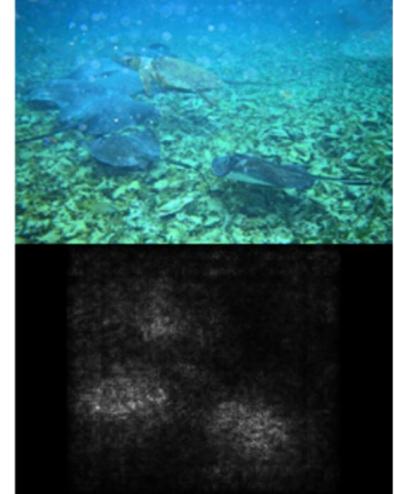
Compute gradient of (unnormalized) class score with respect to image pixels, take absolute value and max over RGB channels



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

Figures copyright Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman, 2014; reproduced with permission.

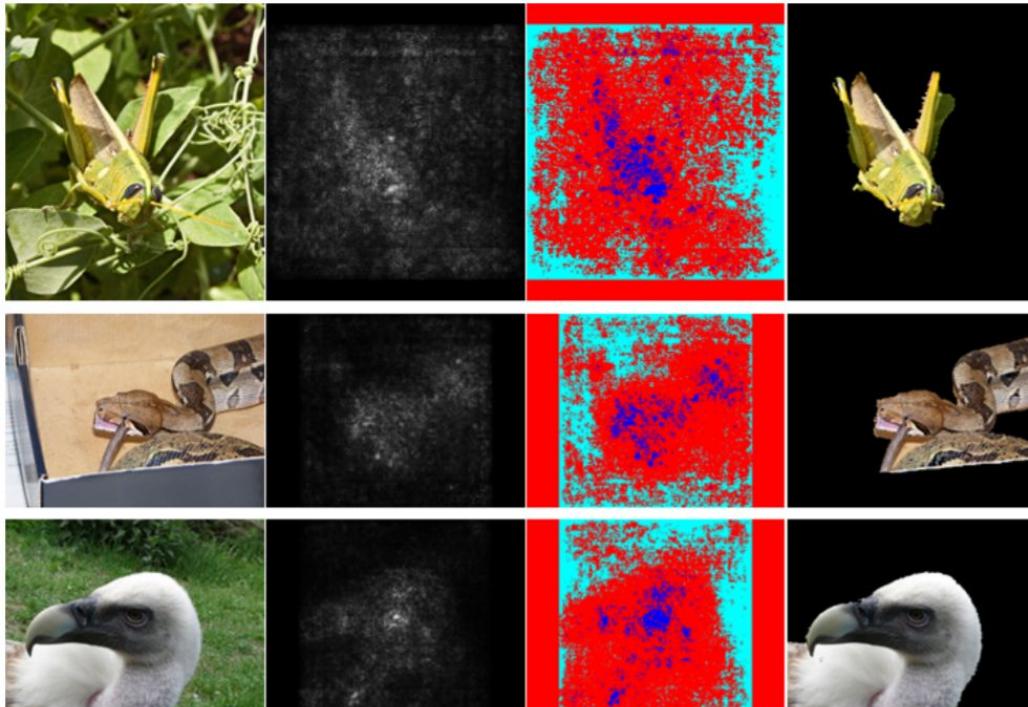
Saliency Maps



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

Figures copyright Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman, 2014; reproduced with permission.

Saliency Maps: Segmentation without supervision



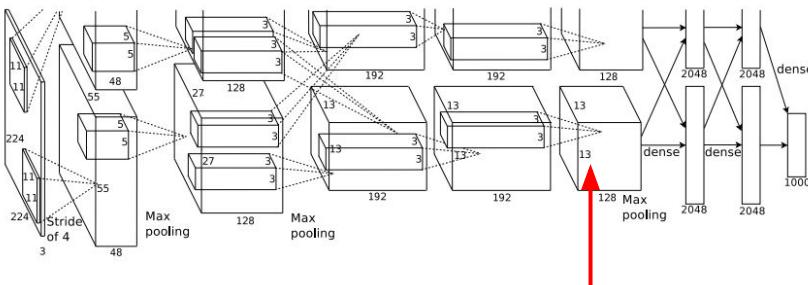
Use GrabCut on
saliency map

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

Figures copyright Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman, 2014; reproduced with permission.

Rother et al, "Grabcut: Interactive foreground extraction using iterated graph cuts", ACM TOG 2004

Intermediate Features via (guided) backprop

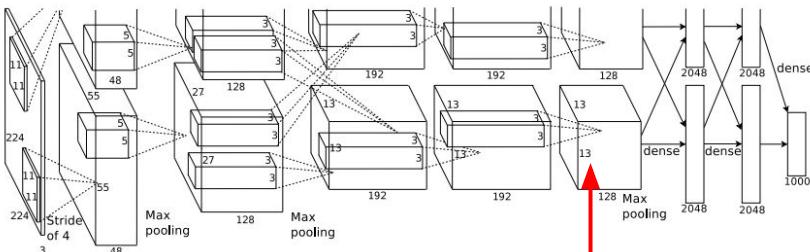


Pick a single intermediate neuron, e.g. one value in $128 \times 13 \times 13$ conv5 feature map

Compute gradient of neuron value with respect to image pixels

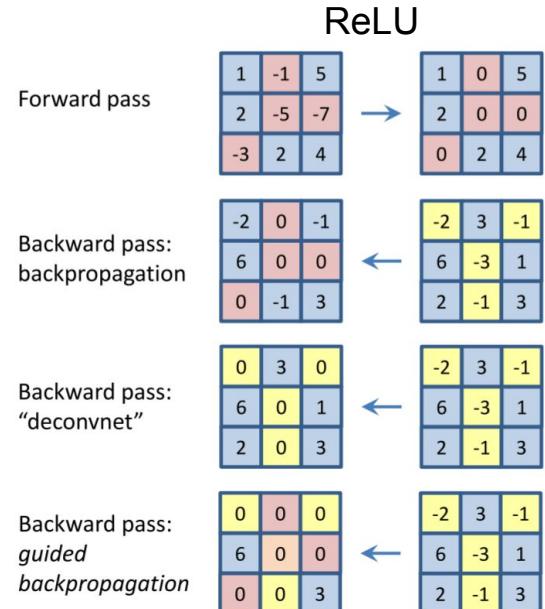
Zeiler and Fergus, "Visualizing and Understanding Convolutional Networks", ECCV 2014
Springenberg et al, "Striving for Simplicity: The All Convolutional Net", ICLR Workshop 2015

Intermediate Features via (guided) backprop



Pick a single intermediate neuron, e.g. one value in $128 \times 13 \times 13$ conv5 feature map

Compute gradient of neuron value with respect to image pixels



Images come out nicer if you only backprop positive gradients through each ReLU (guided backprop)

Figure copyright Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller, 2015; reproduced with permission.

Zeiler and Fergus, "Visualizing and Understanding Convolutional Networks", ECCV 2014
Springenberg et al, "Striving for Simplicity: The All Convolutional Net", ICLR Workshop 2015

Intermediate features via (guided) backprop



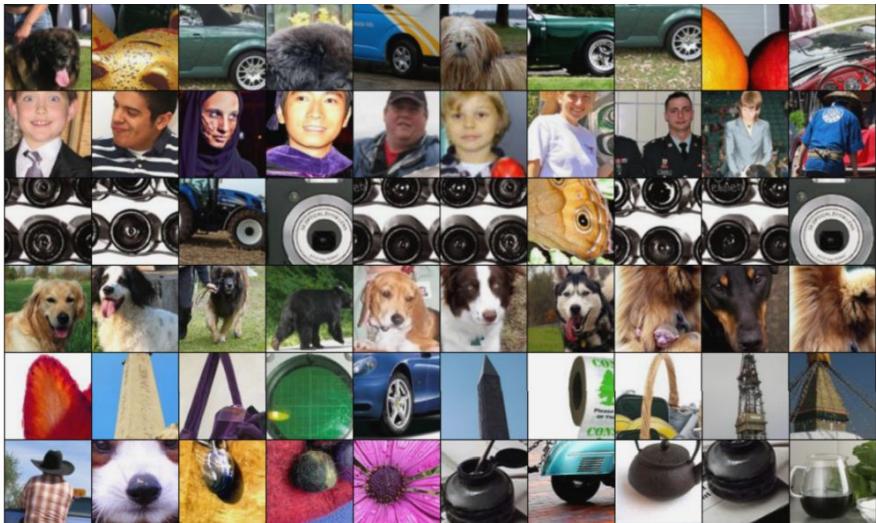
Maximally activating patches
(Each row is a different neuron)



Guided Backprop

Zeiler and Fergus, "Visualizing and Understanding Convolutional Networks", ECCV 2014
Springenberg et al, "Striving for Simplicity: The All Convolutional Net", ICLR Workshop 2015
Figure copyright Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller, 2015; reproduced with permission.

Intermediate features via (guided) backprop



Maximally activating patches (Each row is a different neuron)



Guided Backprop

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

Zeller and Fergus, "Visualizing and Understanding Convolutional Networks", ECCV 2014
Springenberg et al. "Striving for Simplicity: The All Convolutional Net", ICLR Workshop 2015

Figure copyright Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller. 2015; reproduced with permission.

Visualizing CNN features: Gradient Ascent

(Guided) backprop:

Find the part of an image that a neuron responds to

Gradient ascent:

Generate a synthetic image that maximally activates a neuron

$$I^* = \arg \max_I f(I) + R(I)$$

Neuron value

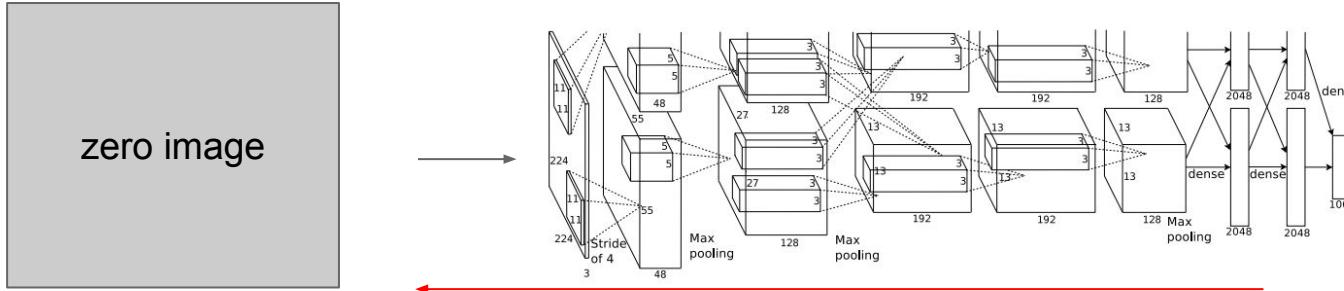
Natural image regularizer

Visualizing CNN features: Gradient Ascent

1. Initialize image to zeros

$$\arg \max_I S_c(I) - \lambda \|I\|_2^2$$

score for class c (before Softmax)



Repeat:

2. Forward image to compute current scores
3. Backprop to get gradient of neuron value with respect to image pixels
4. Make a small update to the image

Visualizing CNN features: Gradient Ascent

$$\arg \max_I S_c(I) - \boxed{\lambda \|I\|_2^2}$$

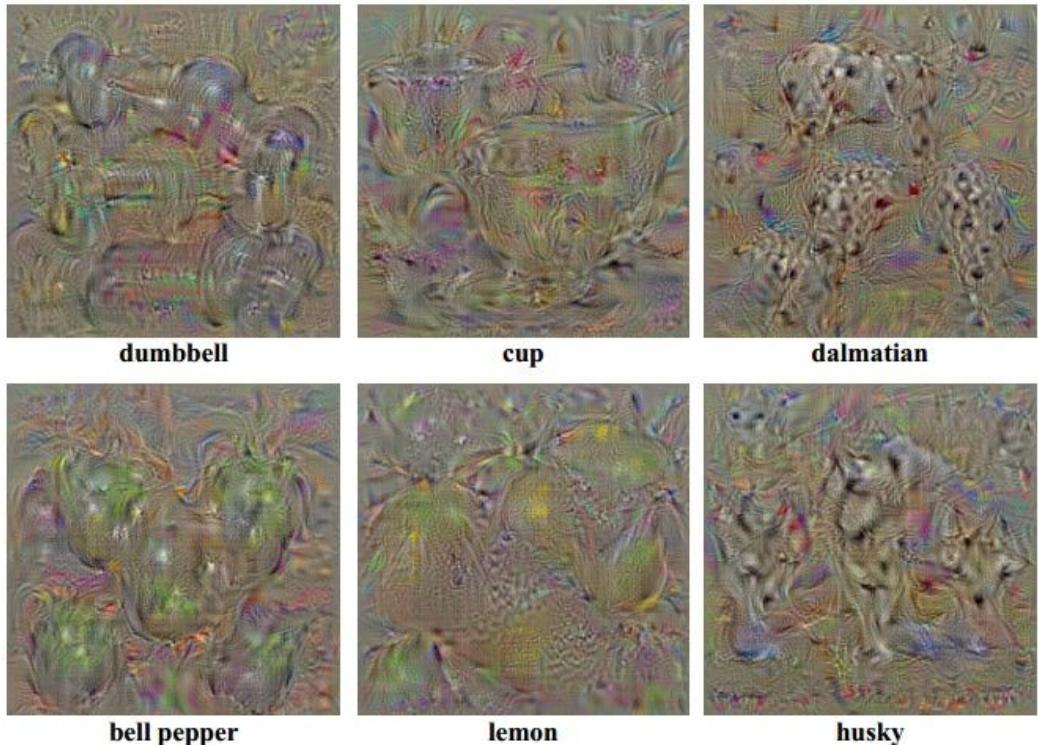
Simple regularizer: Penalize L2
norm of generated image

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

Visualizing CNN features: Gradient Ascent

$$\arg \max_I S_c(I) - \boxed{\lambda \|I\|_2^2}$$

Simple regularizer: Penalize L2 norm of generated image



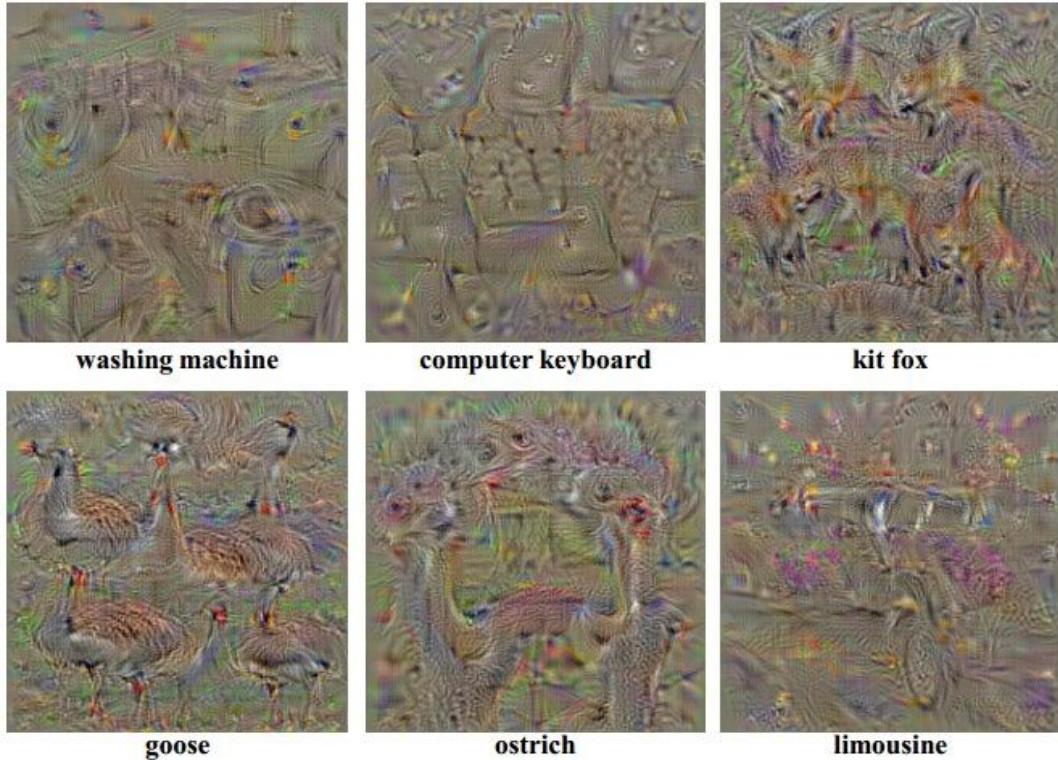
Simonyan, Vedaldi, and Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.

Figures copyright Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman, 2014; reproduced with permission.

Visualizing CNN features: Gradient Ascent

$$\arg \max_I S_c(I) - \boxed{\lambda \|I\|_2^2}$$

Simple regularizer: Penalize L2
norm of generated image



Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML DL Workshop 2014.
Figure copyright Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson, 2014.
Reproduced with permission.

Visualizing CNN features: Gradient Ascent

$$\arg \max_I S_c(I) - \lambda \|I\|_2^2$$

Better regularizer: Penalize L2 norm of image; also during optimization periodically

- (1) Gaussian blur image
- (2) Clip pixels with small values to 0
- (3) Clip pixels with small gradients to 0

Visualizing CNN features: Gradient Ascent

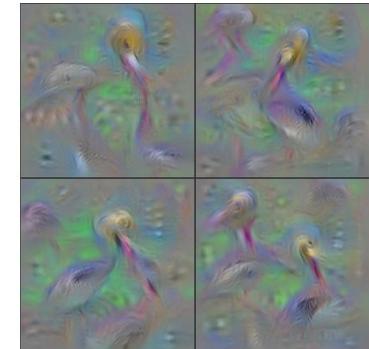
$$\arg \max_I S_c(I) - \lambda \|I\|_2^2$$

Better regularizer: Penalize L2 norm of image; also during optimization periodically

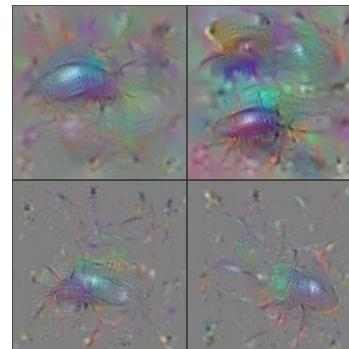
- (1) Gaussian blur image
- (2) Clip pixels with small values to 0
- (3) Clip pixels with small gradients to 0



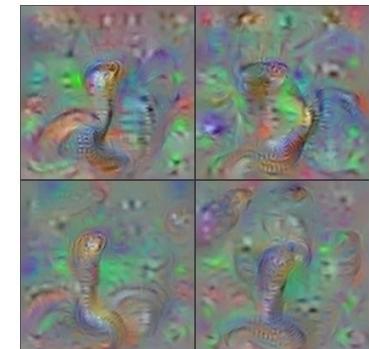
Flamingo



Pelican



Ground Beetle



Indian Cobra

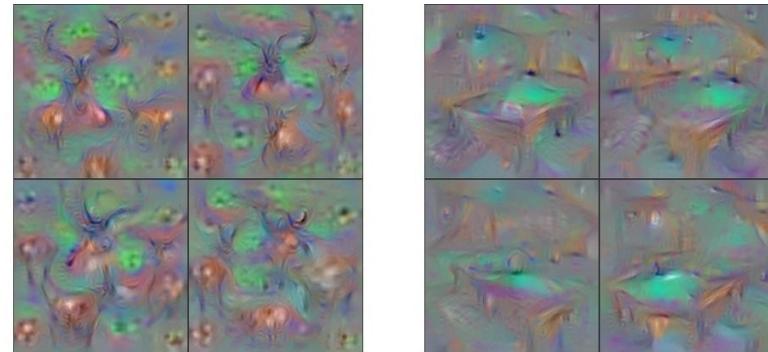
Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML DL Workshop 2014.
Figure copyright Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson, 2014. Reproduced with permission.

Visualizing CNN features: Gradient Ascent

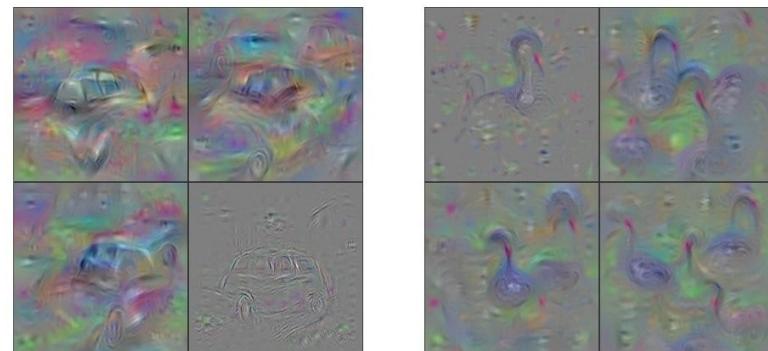
$$\arg \max_I S_c(I) - \lambda \|I\|_2^2$$

Better regularizer: Penalize L2 norm of image; also during optimization periodically

- (1) Gaussian blur image
- (2) Clip pixels with small values to 0
- (3) Clip pixels with small gradients to 0



Hartebeest



Station Wagon

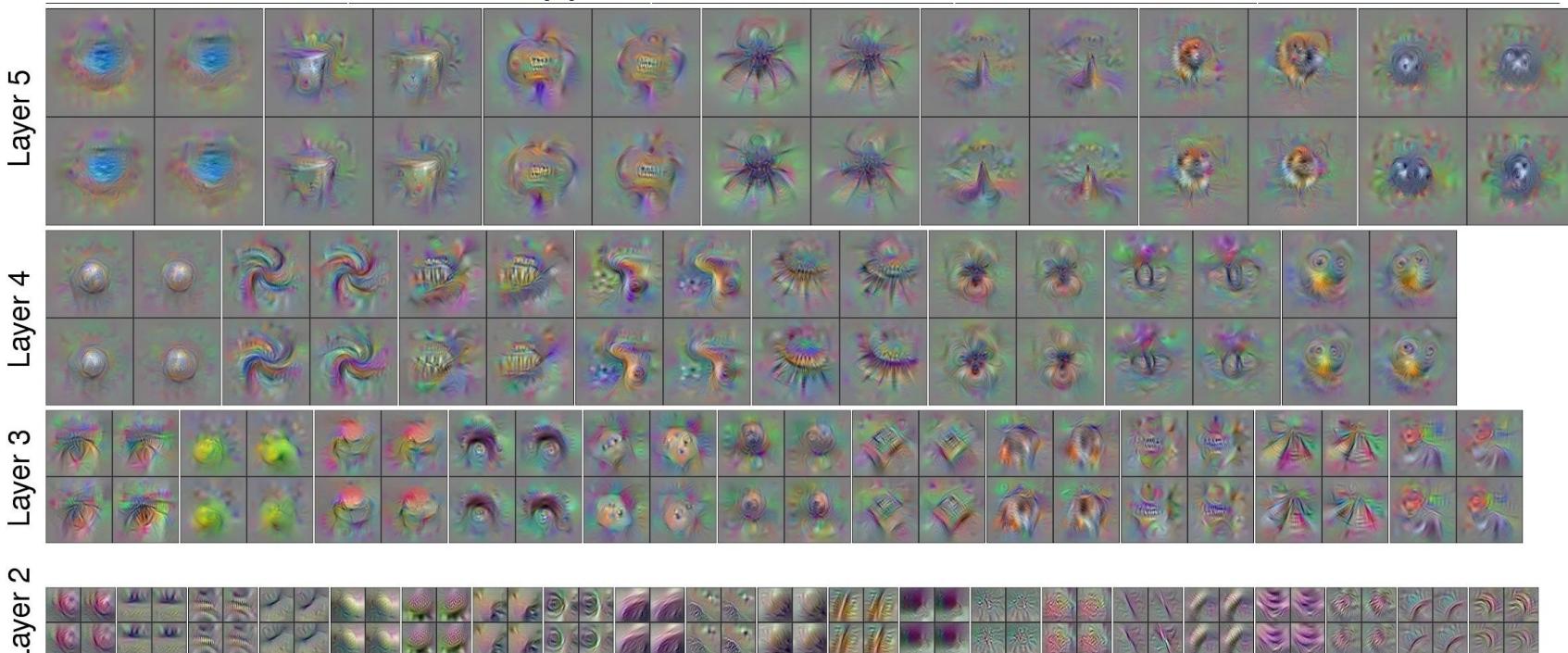
Billiard Table

Black Swan

Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML DL Workshop 2014.
Figure copyright Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson, 2014. Reproduced with permission.

Visualizing CNN features: Gradient Ascent

Use the same approach to visualize intermediate features



Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML DL Workshop 2014.

Figure copyright Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson, 2014. Reproduced with permission.

Visualizing CNN features: Gradient Ascent

Adding “multi-faceted” visualization gives even nicer results:
(Plus more careful regularization, center-bias)

Reconstructions of multiple feature types (facets) recognized by the same “grocery store” neuron



Corresponding example training set images recognized by the same neuron as in the “grocery store” class



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.
Figures copyright Anh Nguyen, Jason Yosinski, and Jeff Clune, 2016; reproduced with permission.

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.

Figures copyright Anh Nguyen, Jason Yosinski, and Jeff Clune, 2016; reproduced with permission.

Visualizing CNN features: Gradient Ascent

Optimize in FC6 latent space instead of pixel space:



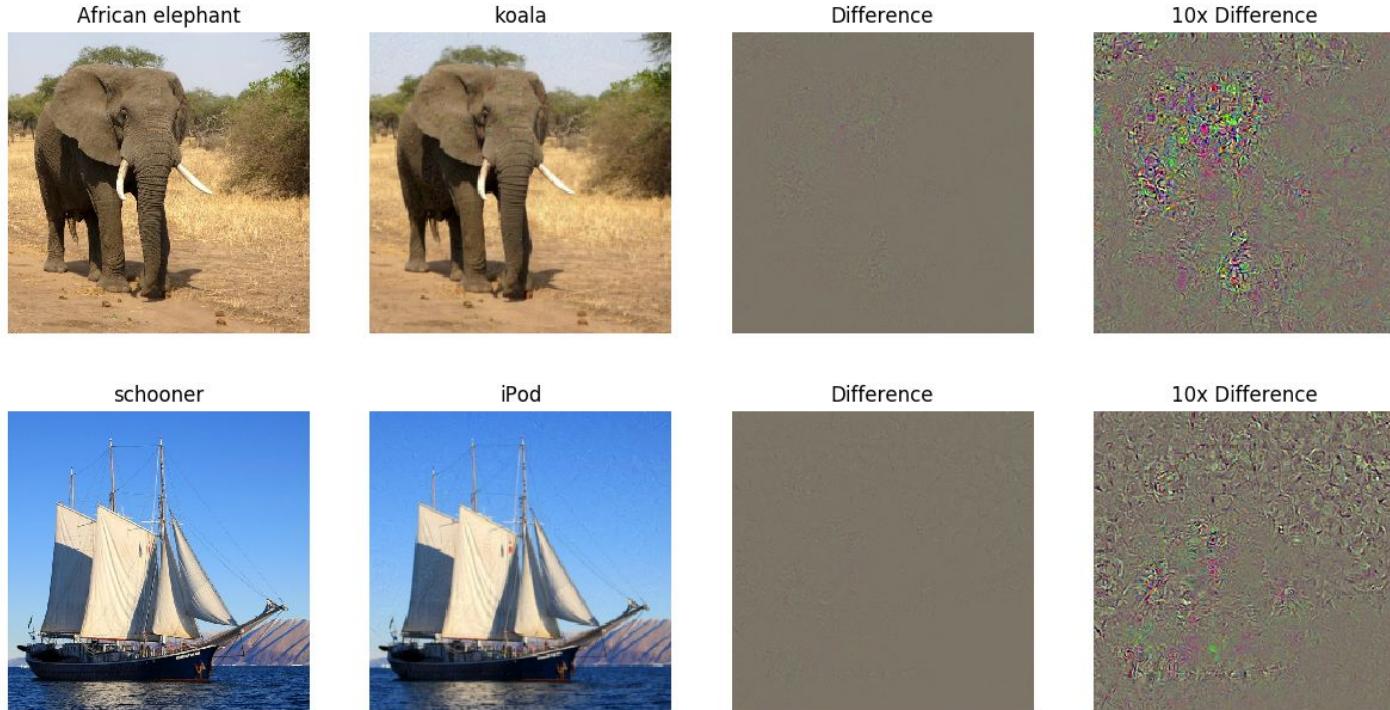
Nguyen et al., "Synthesizing the preferred inputs for neurons in neural networks via deep generator networks," NIPS 2016

Figure copyright Nguyen et al, 2016; reproduced with permission.

Fooling Images / Adversarial Examples

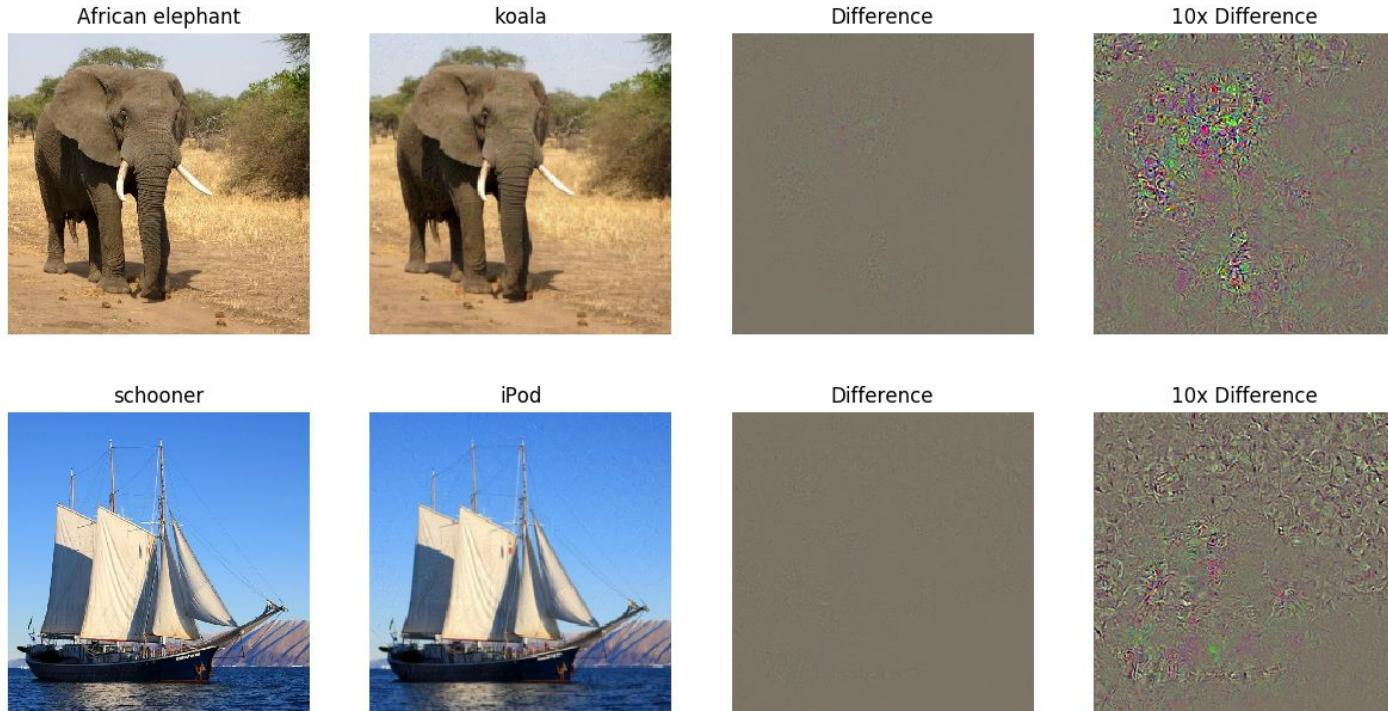
- (1) Start from an arbitrary image
- (2) Pick an arbitrary class
- (3) Modify the image to maximize the class
- (4) Repeat until network is fooled

Fooling Images / Adversarial Examples



[Boat image is CC0 public domain](#)
[Elephant image is CC0 public domain](#)

Fooling Images / Adversarial Examples

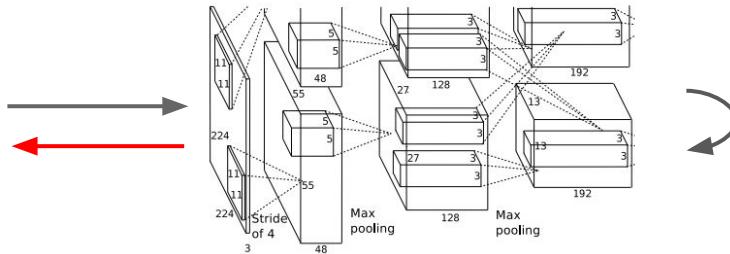


Check out [Ian Goodfellow's lecture](#) from last year

Boat image is CC0 public domain
Elephant image is CC0 public domain

DeepDream: Amplify existing features

Rather than synthesizing an image to maximize a specific neuron, instead try to **amplify** the neuron activations at some layer in the network



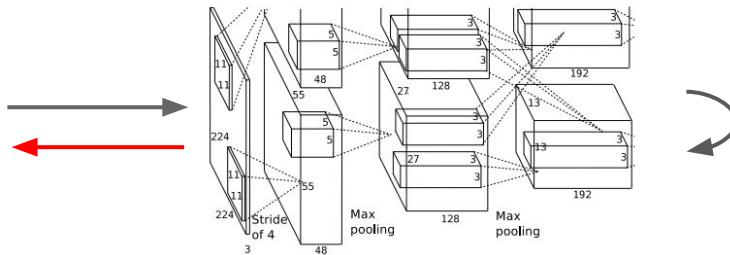
Choose an image and a layer in a CNN; repeat:

1. Forward: compute activations at chosen layer
2. Set gradient of chosen layer *equal to its activation*
3. Backward: Compute gradient on image
4. Update image

Mordvintsev, Olah, and Tyka, "Inceptionism: Going Deeper into Neural Networks", [Google Research Blog](#). Images are licensed under [CC-BY 4.0](#).

DeepDream: Amplify existing features

Rather than synthesizing an image to maximize a specific neuron, instead try to **amplify** the neuron activations at some layer in the network



Choose an image and a layer in a CNN; repeat:

1. Forward: compute activations at chosen layer
2. Set gradient of chosen layer *equal to its activation*
3. Backward: Compute gradient on image
4. Update image

Equivalent to:

$$I^* = \arg \max_I \sum_i f_i(I)^2$$

Mordvintsev, Olah, and Tyka, "Inceptionism: Going Deeper into Neural Networks", [Google Research Blog](#). Images are licensed under [CC-BY 4.0](#).

DeepDream: Amplify existing features

```
def objective_L2(dst):
    dst.diff[:] = dst.data

def make_step(net, step_size=1.5, end='inception_4c/output',
             jitter=32, clip=True, objective=objective_L2):
    '''Basic gradient ascent step.'''
    src = net.blobs['data'] # input image is stored in Net's 'data' blob
    dst = net.blobs[end]

    ox, oy = np.random.randint(-jitter, jitter+1, 2)
    src.data[0] = np.roll(np.roll(src.data[0], ox, -1), oy, -2) # apply jitter shift

    net.forward(end=end)
    objective(dst) # specify the optimization objective
    net.backward(start=end)
    g = src.diff[0]
    # apply normalized ascent step to the input image
    src.data[:] += step_size/np.abs(g).mean() * g

    src.data[0] = np.roll(np.roll(src.data[0], -ox, -1), -oy, -2) # unshift image

    if clip:
        bias = net.transformer.mean['data']
        src.data[:] = np.clip(src.data, -bias, 255-bias)
```

[Code](#) is very simple but it uses a couple tricks:

(Code is licensed under [Apache 2.0](#))

DeepDream: Amplify existing features

```
def objective_L2(dst):
    dst.diff[:] = dst.data

def make_step(net, step_size=1.5, end='inception_4c/output',
             jitter=32, clip=True, objective=objective_L2):
    '''Basic gradient ascent step.'''
    src = net.blobs['data'] # input image is stored in Net's 'data' blob
    dst = net.blobs[end]

    ox, oy = np.random.randint(-jitter, jitter+1, 2)
    src.data[0] = np.roll(np.roll(src.data[0], ox, -1), oy, -2) # apply jitter shift

    net.forward(end=end)
    objective(dst) # specify the optimization objective
    net.backward(start=end)
    g = src.diff[0]
    # apply normalized ascent step to the input image
    src.data[:] += step_size/np.abs(g).mean() * g

    src.data[0] = np.roll(np.roll(src.data[0], -ox, -1), -oy, -2) # unshift image

if clip:
    bias = net.transformer.mean['data']
    src.data[:] = np.clip(src.data, -bias, 255-bias)
```

[Code](#) is very simple but it uses a couple tricks:

(Code is licensed under [Apache 2.0](#))

Jitter image

DeepDream: Amplify existing features

```
def objective_L2(dst):
    dst.diff[:] = dst.data

def make_step(net, step_size=1.5, end='inception_4c/output',
             jitter=32, clip=True, objective=objective_L2):
    '''Basic gradient ascent step.'''
    src = net.blobs['data'] # input image is stored in Net's 'data' blob
    dst = net.blobs[end]

    ox, oy = np.random.randint(-jitter, jitter+1, 2)
    src.data[0] = np.roll(np.roll(src.data[0], ox, -1), oy, -2) # apply jitter shift

    net.forward(end=end)
    objective(dst) # specify the optimization objective
    net.backward(start=end)
    g = src.diff[0]
    # apply normalized ascent step to the input image
    src.data[:] += step_size/np.abs(g).mean() * g

    src.data[0] = np.roll(np.roll(src.data[0], -ox, -1), -oy, -2) # unshift image

    if clip:
        bias = net.transformer.mean['data']
        src.data[:] = np.clip(src.data, -bias, 255-bias)
```

[Code](#) is very simple but it uses a couple tricks:

(Code is licensed under [Apache 2.0](#))

Jitter image

L1 Normalize gradients

DeepDream: Amplify existing features

```
def objective_L2(dst):
    dst.diff[:] = dst.data

def make_step(net, step_size=1.5, end='inception_4c/output',
             jitter=32, clip=True, objective=objective_L2):
    '''Basic gradient ascent step.'''
    src = net.blobs['data'] # input image is stored in Net's 'data' blob
    dst = net.blobs[end]

    ox, oy = np.random.randint(-jitter, jitter+1, 2)
    src.data[0] = np.roll(np.roll(src.data[0], ox, -1), oy, -2) # apply jitter shift

    net.forward(end=end)
    objective(dst) # specify the optimization objective
    net.backward(start=end)
    g = src.diff[0]
    # apply normalized ascent step to the input image
    src.data[:] += step_size/np.abs(g).mean() * g

    src.data[0] = np.roll(np.roll(src.data[0], -ox, -1), -oy, -2) # unshift image

    if clip:
        bias = net.transformer.mean['data']
        src.data[:] = np.clip(src.data, -bias, 255-bias)
```

[Code](#) is very simple but it uses a couple tricks:

(Code is licensed under [Apache 2.0](#))

Jitter image

L1 Normalize gradients

Clip pixel values

Also uses multiscale processing for a fractal effect (not shown)



Sky image is licensed under [CC-BY SA 3.0](#)

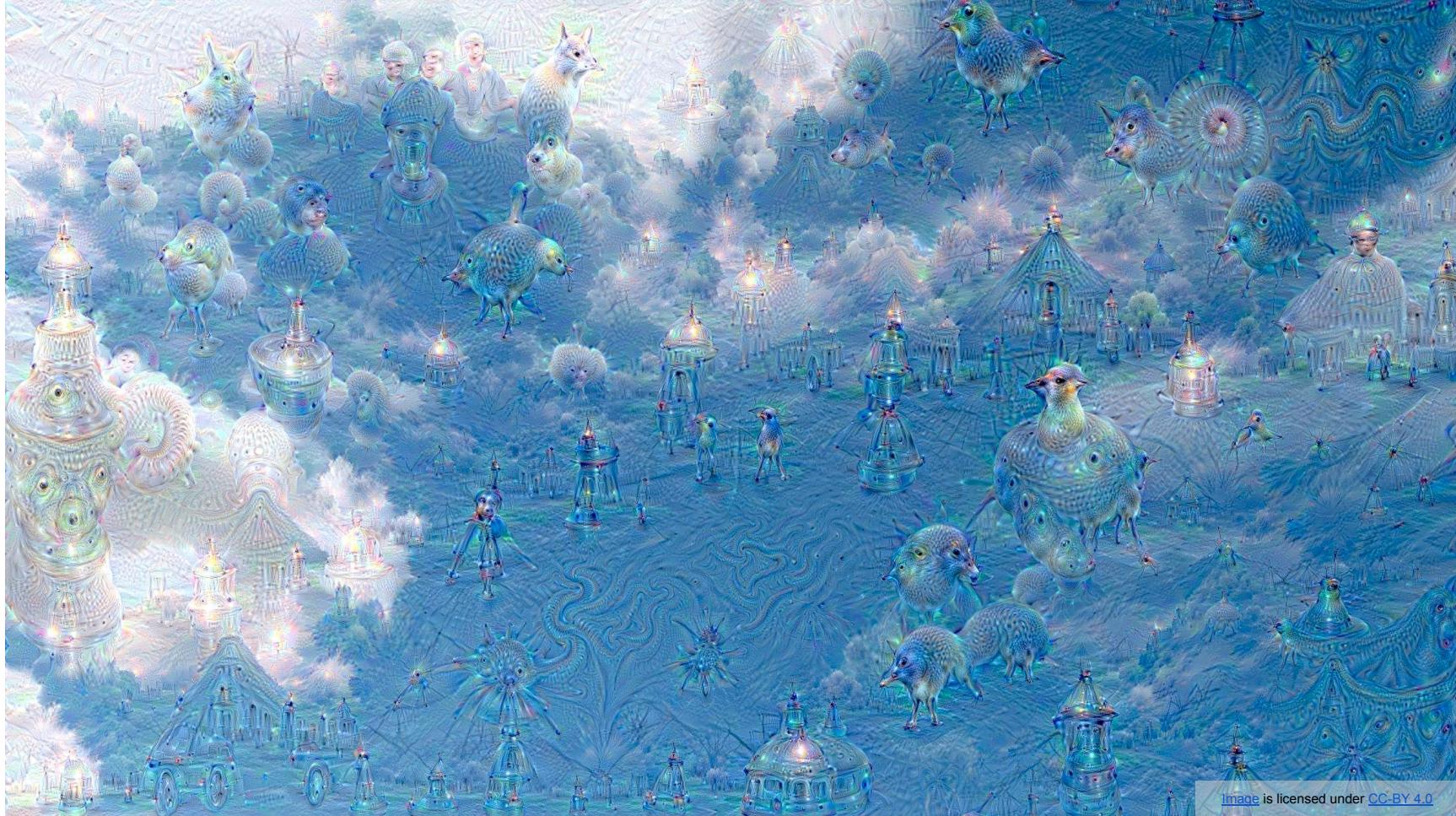


Image is licensed under [CC-BY 4.0](#)

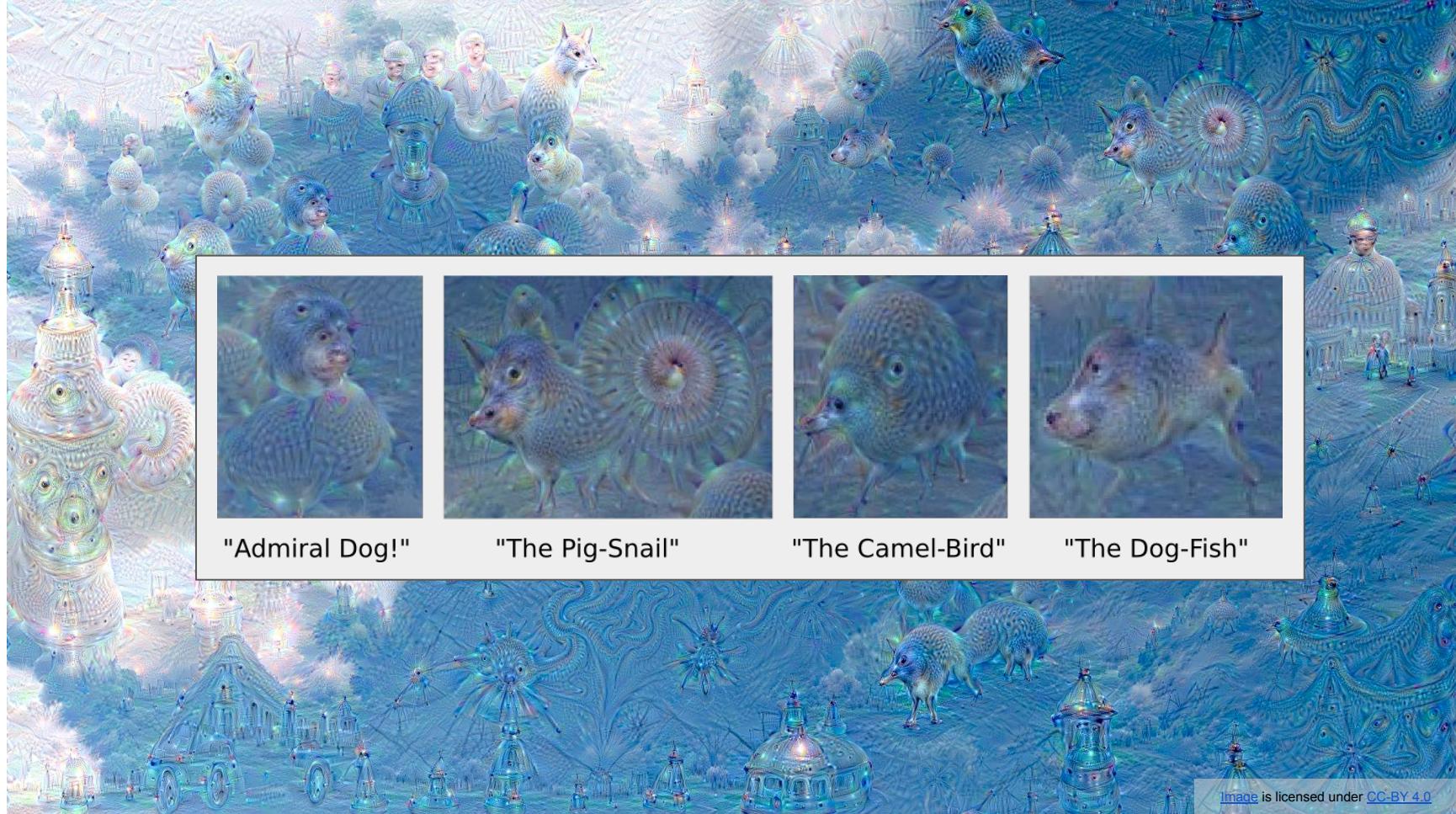
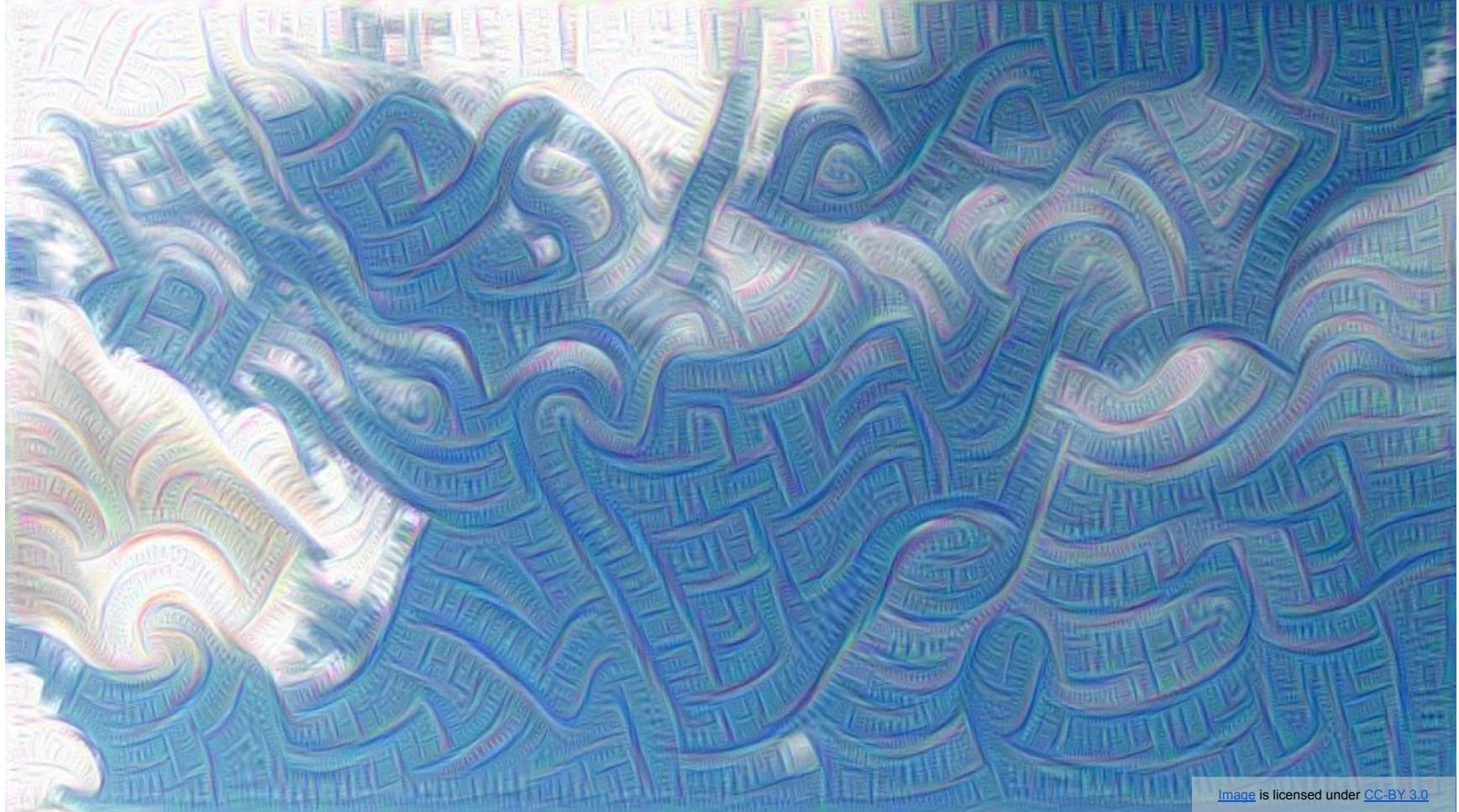


Image is licensed under [CC-BY 4.0](#)



[Image](#) is licensed under [CC-BY 3.0](#)



[Image](#) is licensed under [CC-BY 3.0](#)

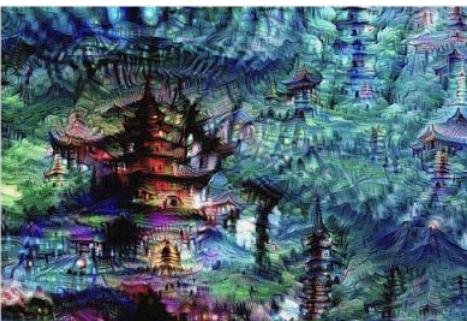
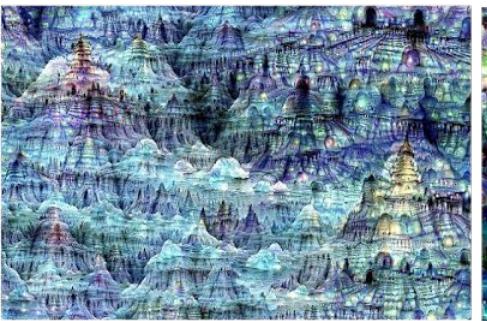
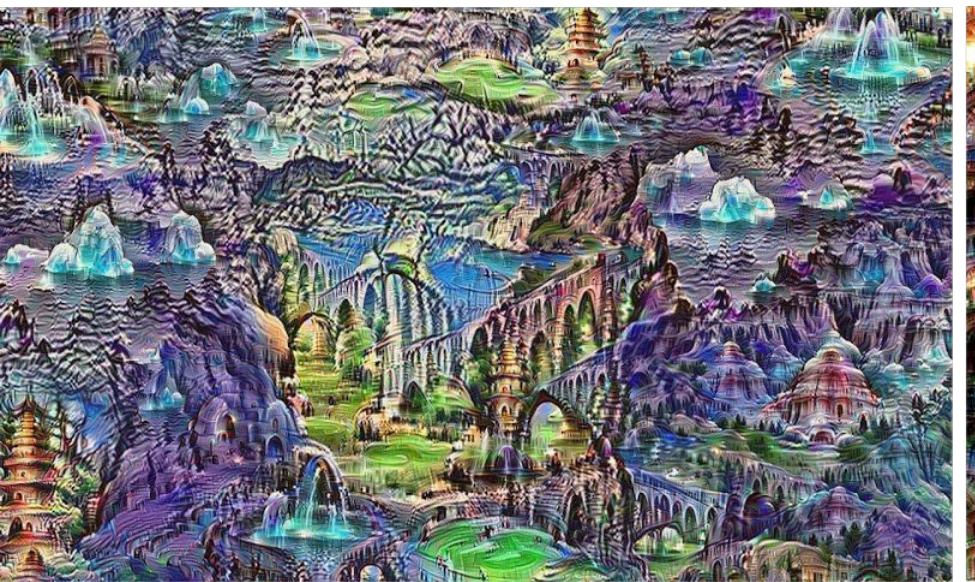


Image is licensed under CC-BY 4.0

Feature Inversion

Given a CNN feature vector for an image, find a new image that:

- Matches the given feature vector
- “looks natural” (image prior regularization)

$$\mathbf{x}^* = \underset{\mathbf{x} \in \mathbb{R}^{H \times W \times C}}{\operatorname{argmin}} \ell(\Phi(\mathbf{x}), \Phi_0) + \lambda \mathcal{R}(\mathbf{x})$$

Given feature vector

Features of new image

$$\ell(\Phi(\mathbf{x}), \Phi_0) = \|\Phi(\mathbf{x}) - \Phi_0\|^2$$

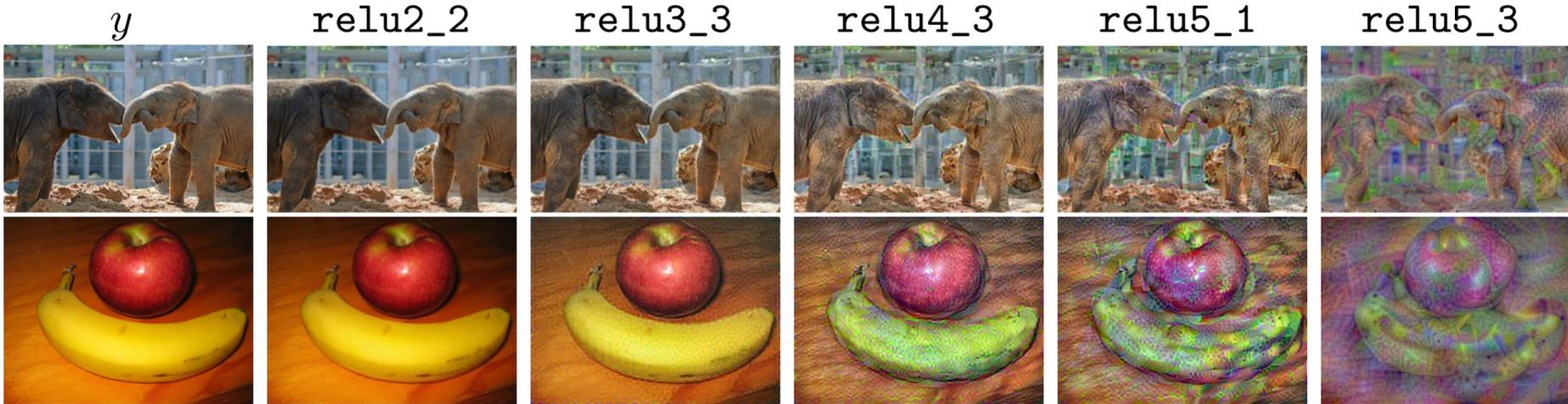
$$\mathcal{R}_{V^\beta}(\mathbf{x}) = \sum_{i,j} \left((x_{i,j+1} - x_{ij})^2 + (x_{i+1,j} - x_{ij})^2 \right)^{\frac{\beta}{2}}$$

Total Variation regularizer
(encourages spatial smoothness)

Mahendran and Vedaldi, “Understanding Deep Image Representations by Inverting Them”, CVPR 2015

Feature Inversion

Reconstructing from different layers of VGG-16



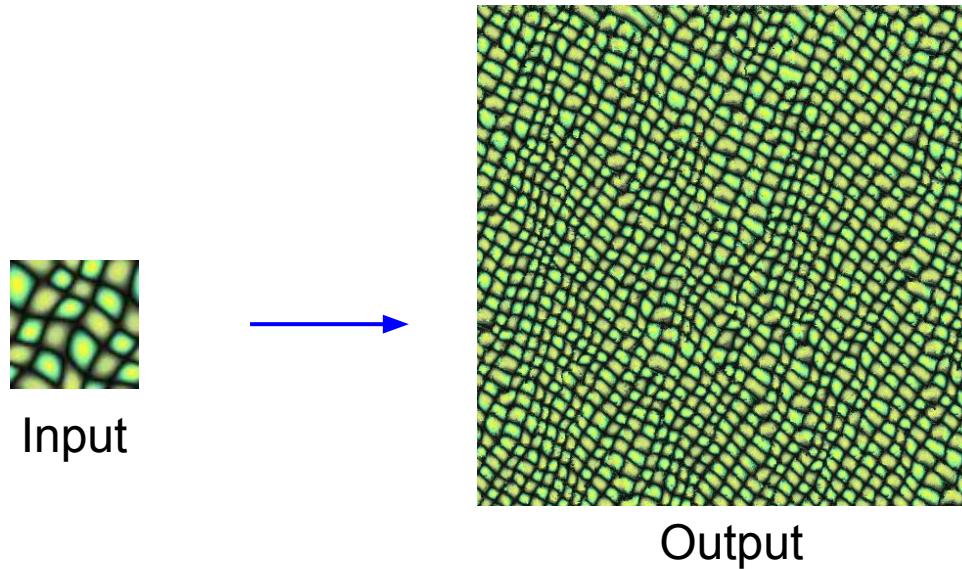
Mahendran and Vedaldi, "Understanding Deep Image Representations by Inverting Them", CVPR 2015

Figure from Johnson, Alahi, and Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution", ECCV 2016. Copyright Springer, 2016.

Reproduced for educational purposes.

Texture Synthesis

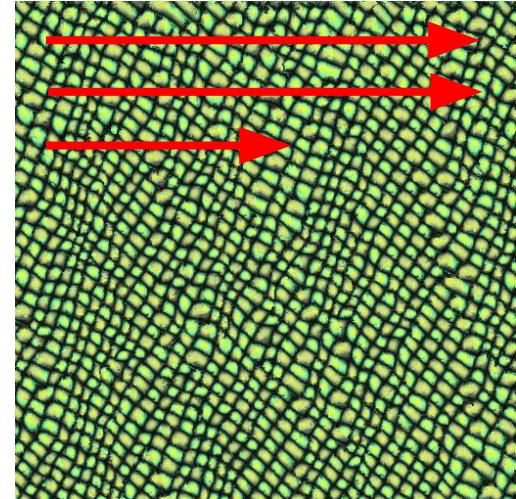
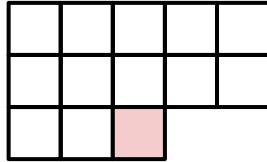
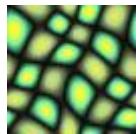
Given a sample patch of some texture, can we generate a bigger image of the same texture?



[Output image](#) is licensed under the [MIT license](#)

Texture Synthesis: Nearest Neighbor

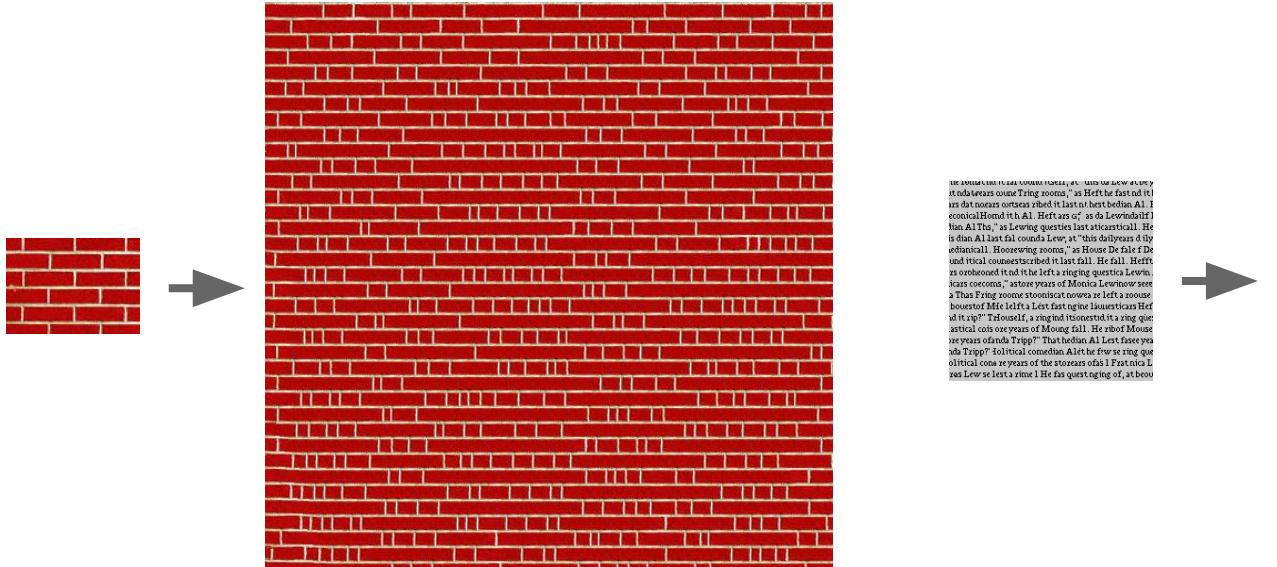
Generate pixels one at a time in scanline order; form neighborhood of already generated pixels and copy nearest neighbor from input



Wei and Levoy, "Fast Texture Synthesis using Tree-structured Vector Quantization", SIGGRAPH 2000

Efros and Leung, "Texture Synthesis by Non-parametric Sampling", ICCV 1999

Texture Synthesis: Nearest Neighbor

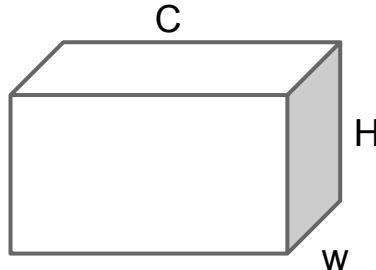
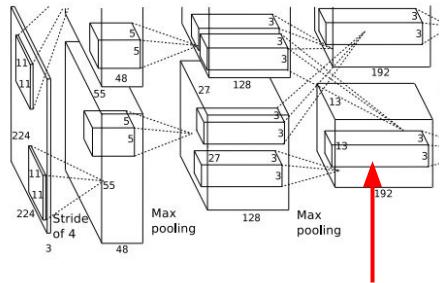


This diagram illustrates the process of texture synthesis using the nearest neighbor method. On the left, a small image of a brick wall is shown. A large arrow points from this image to a much larger, synthesized image of the same brick wall. This visualizes how a local patch is copied to create a larger texture.

Neural Texture Synthesis: Gram Matrix



[This image](#) is in the public domain.

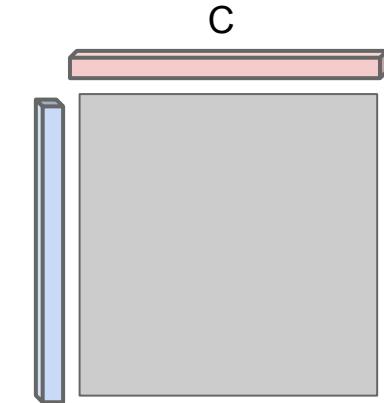
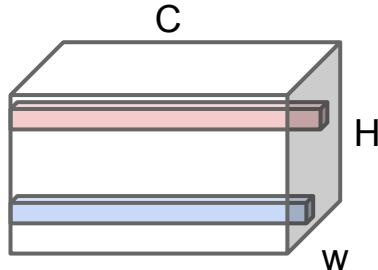
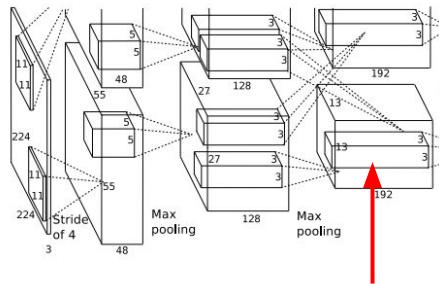


Each layer of CNN gives $C \times H \times W$ tensor of features; $H \times W$ grid of C -dimensional vectors

Neural Texture Synthesis: Gram Matrix



This image is in the public domain.



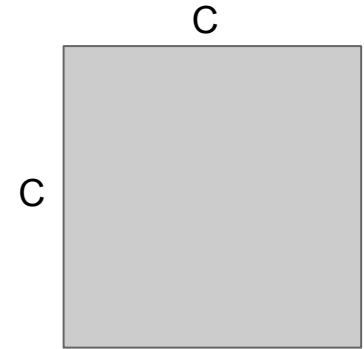
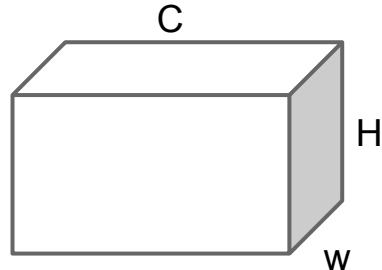
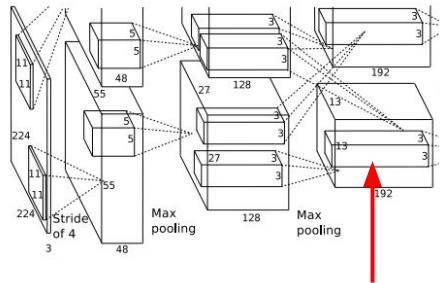
Each layer of CNN gives $C \times H \times W$ tensor of features; $H \times W$ grid of C -dimensional vectors

Outer product of two C -dimensional vectors gives $C \times C$ matrix measuring co-occurrence

Neural Texture Synthesis: Gram Matrix



This image is in the public domain.



Each layer of CNN gives $C \times H \times W$ tensor of features; $H \times W$ grid of C -dimensional vectors

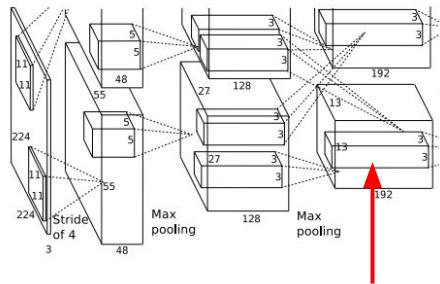
Outer product of two C -dimensional vectors gives $C \times C$ matrix measuring co-occurrence

Average over all HW pairs of vectors, giving **Gram matrix** of shape $C \times C$

Neural Texture Synthesis: Gram Matrix



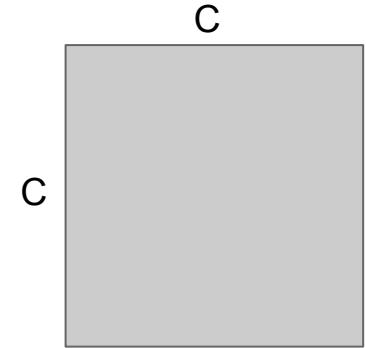
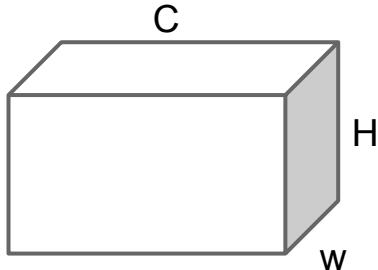
This image is in the public domain.



Each layer of CNN gives $C \times H \times W$ tensor of features; $H \times W$ grid of C -dimensional vectors

Outer product of two C -dimensional vectors gives $C \times C$ matrix measuring co-occurrence

Average over all HW pairs of vectors, giving **Gram matrix** of shape $C \times C$



Efficient to compute; reshape features from

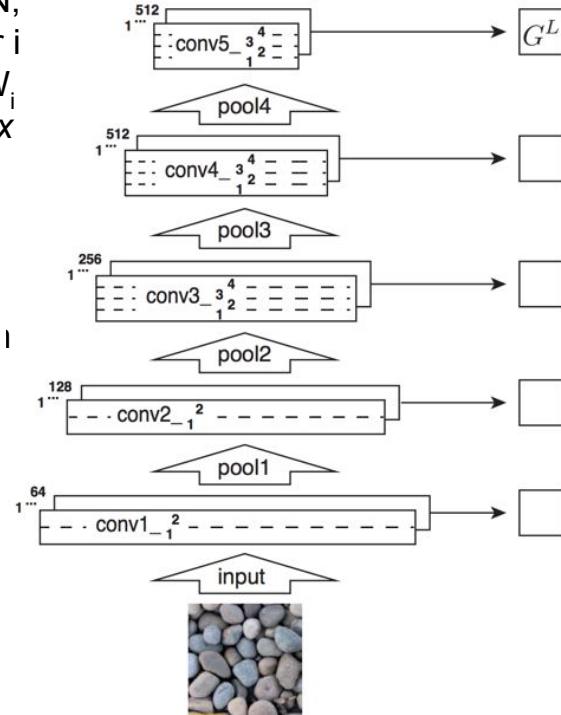
$C \times H \times W$ to $=C \times HW$

then compute $G = FF^T$

Neural Texture Synthesis

1. Pretrain a CNN on ImageNet (VGG-19)
2. Run input texture forward through CNN, record activations on every layer; layer i gives feature map of shape $C_i \times H_i \times W_i$
3. At each layer compute the *Gram matrix* giving outer product of features:

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l \text{ (shape } C_i \times C_i\text{)}$$



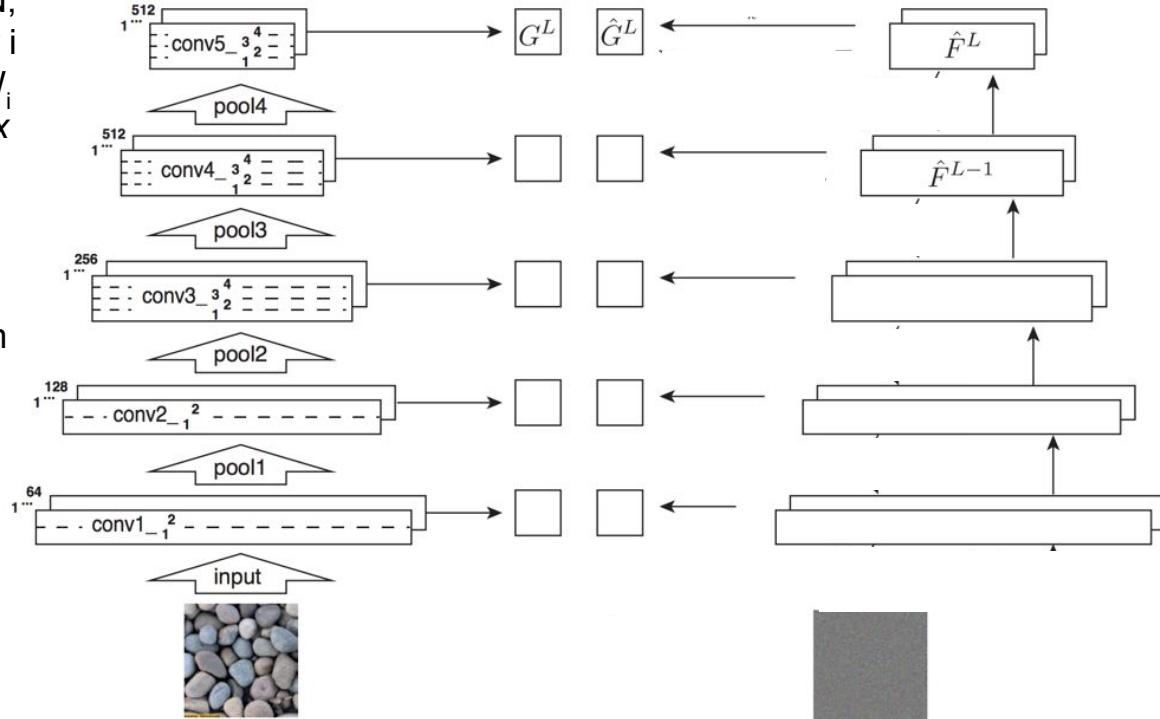
Gatys, Ecker, and Bethge, "Texture Synthesis Using Convolutional Neural Networks", NIPS 2015
Figure copyright Leon Gatys, Alexander S. Ecker, and Matthias Bethge, 2015. Reproduced with permission.

Neural Texture Synthesis

1. Pretrain a CNN on ImageNet (VGG-19)
2. Run input texture forward through CNN, record activations on every layer; layer i gives feature map of shape $C_i \times H_i \times W_i$
3. At each layer compute the *Gram matrix* giving outer product of features:

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l \text{ (shape } C_i \times C_i\text{)}$$

4. Initialize generated image from random noise
5. Pass generated image through CNN, compute Gram matrix on each layer



Gatys, Ecker, and Bethge, "Texture Synthesis Using Convolutional Neural Networks", NIPS 2015
Figure copyright Leon Gatys, Alexander S. Ecker, and Matthias Bethge, 2015. Reproduced with permission.

Neural Texture Synthesis

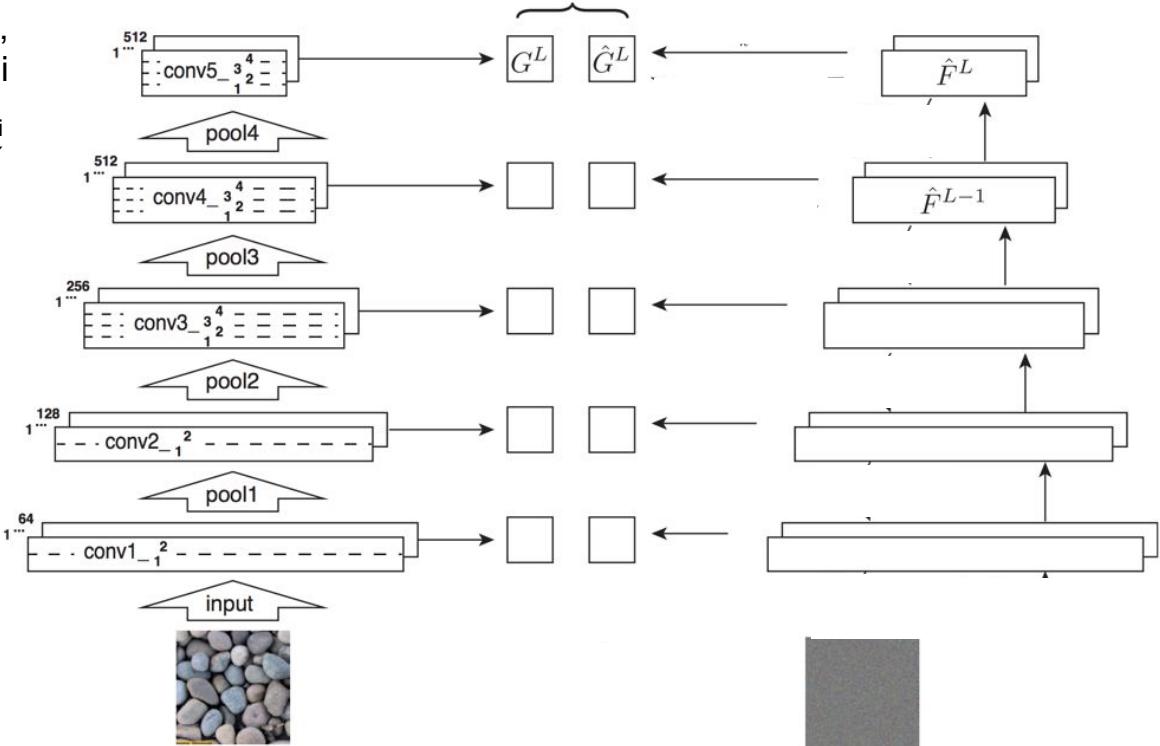
1. Pretrain a CNN on ImageNet (VGG-19)
2. Run input texture forward through CNN, record activations on every layer; layer i gives feature map of shape $C_i \times H_i \times W_i$
3. At each layer compute the *Gram matrix* giving outer product of features:

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l \text{ (shape } C_i \times C_i\text{)}$$

4. Initialize generated image from random noise
5. Pass generated image through CNN, compute Gram matrix on each layer
6. Compute loss: weighted sum of L2 distance between Gram matrices

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G_{ij}^l - \hat{G}_{ij}^l \right)^2$$

$$\mathcal{L}(\vec{x}, \hat{x}) = \sum_{l=0}^L w_l E_l$$



Gatys, Ecker, and Bethge, "Texture Synthesis Using Convolutional Neural Networks", NIPS 2015
 Figure copyright Leon Gatys, Alexander S. Ecker, and Matthias Bethge, 2015. Reproduced with permission.

Neural Texture Synthesis

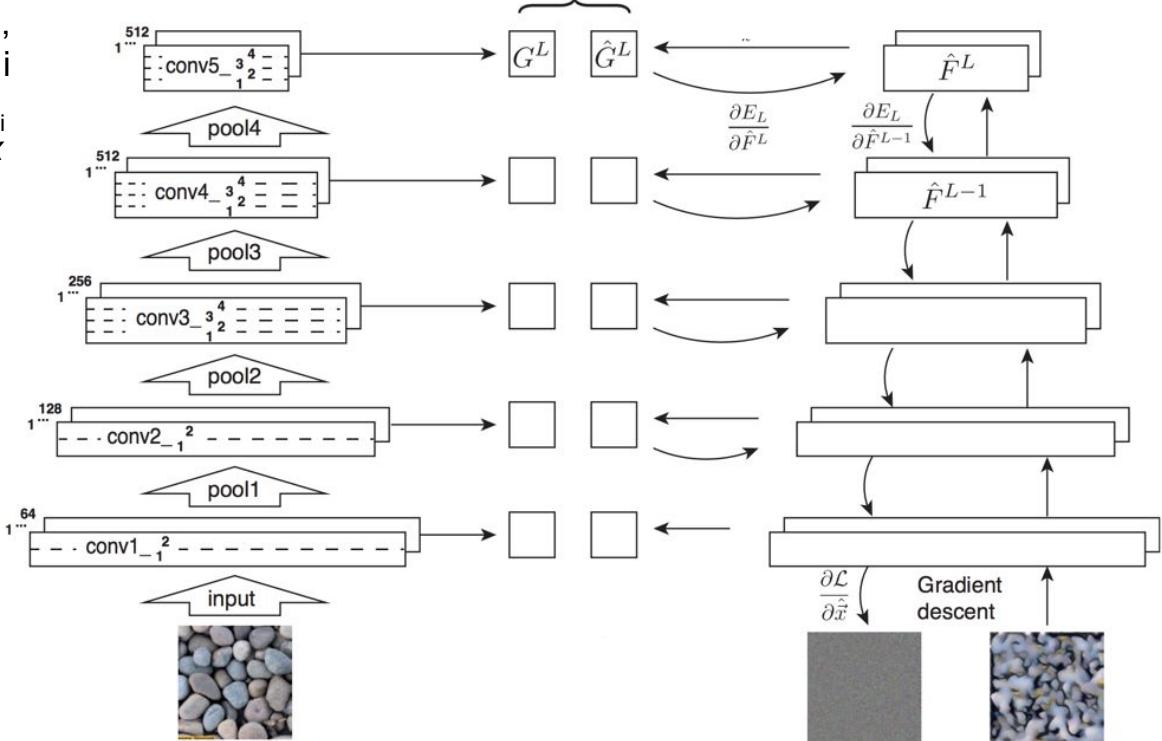
1. Pretrain a CNN on ImageNet (VGG-19)
2. Run input texture forward through CNN, record activations on every layer; layer i gives feature map of shape $C_i \times H_i \times W_i$
3. At each layer compute the *Gram matrix* giving outer product of features:

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l \text{ (shape } C_i \times C_i\text{)}$$

4. Initialize generated image from random noise
5. Pass generated image through CNN, compute Gram matrix on each layer
6. Compute loss: weighted sum of L2 distance between Gram matrices
7. Backprop to get gradient on image
8. Make gradient step on image
9. GOTO 5

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G_{ij}^l - \hat{G}_{ij}^l \right)^2$$

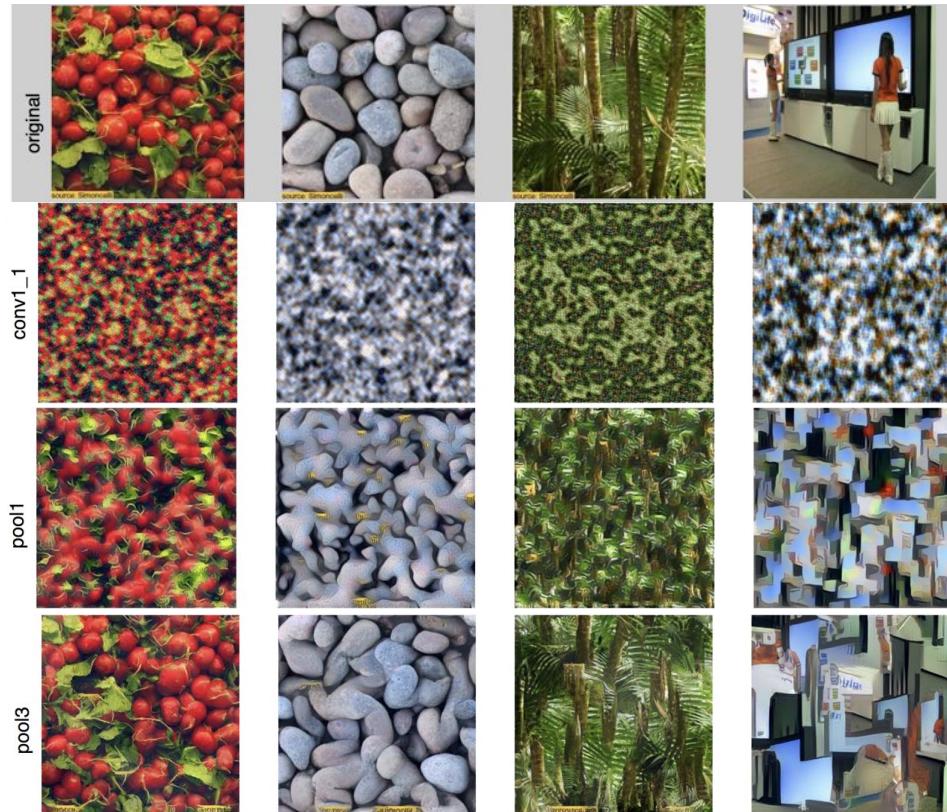
$$\mathcal{L}(\vec{x}, \hat{x}) = \sum_{l=0}^L w_l E_l$$



Gatys, Ecker, and Bethge, "Texture Synthesis Using Convolutional Neural Networks", NIPS 2015
 Figure copyright Leon Gatys, Alexander S. Ecker, and Matthias Bethge, 2015. Reproduced with permission.

Neural Texture Synthesis

Reconstructing texture from higher layers recovers larger features from the input texture



Gatys, Ecker, and Bethge, "Texture Synthesis Using Convolutional Neural Networks", NIPS 2015
Figure copyright Leon Gatys, Alexander S. Ecker, and Matthias Bethge, 2015. Reproduced with permission.

Neural Texture Synthesis: Texture = Artwork

Texture synthesis
(Gram
reconstruction)

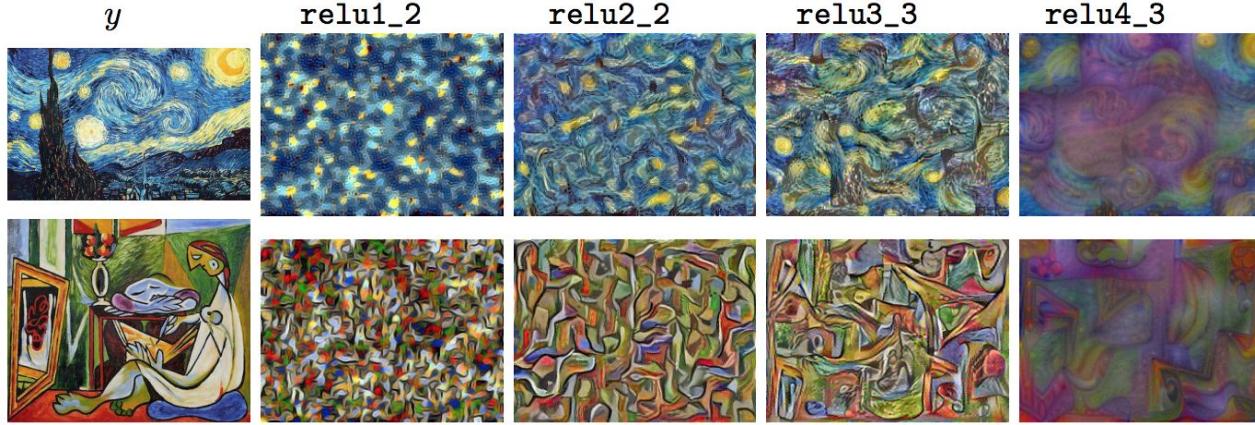
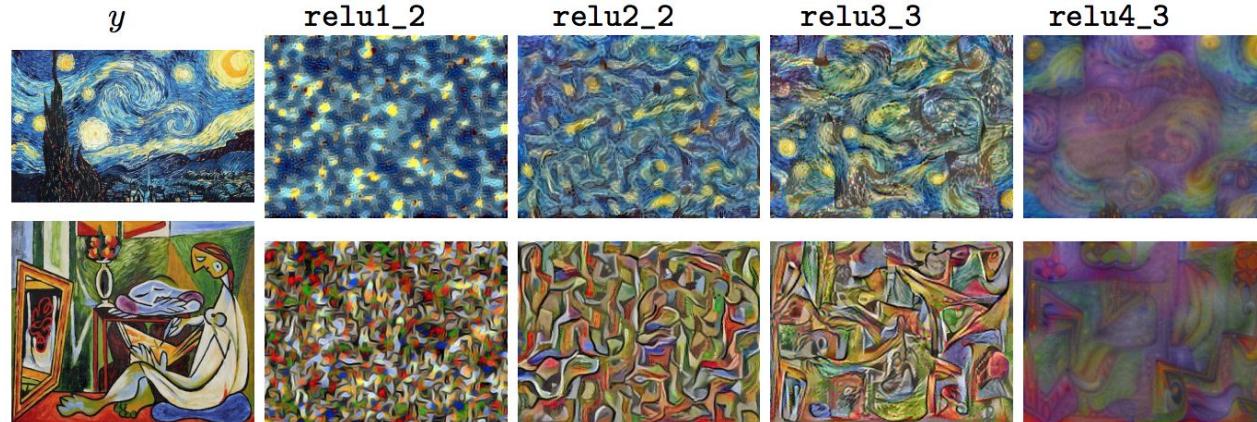


Figure from Johnson, Alahi, and Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution", ECCV 2016. Copyright Springer, 2016.
Reproduced for educational purposes.

Neural Style Transfer: Feature + Gram Reconstruction

Texture synthesis
(Gram
reconstruction)



Feature
reconstruction

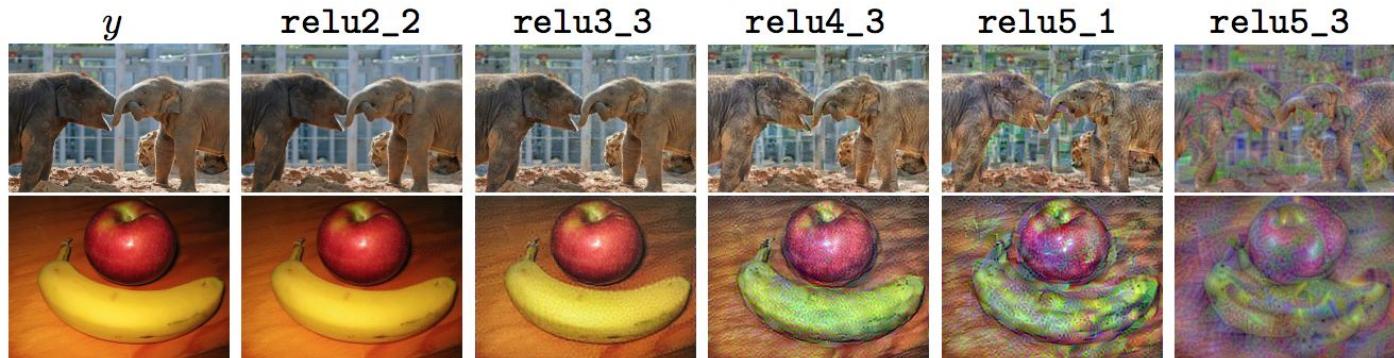


Figure from Johnson, Alahi, and Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution", ECCV 2016. Copyright Springer, 2016.
Reproduced for educational purposes.

Neural Style Transfer

Content Image



[This image is licensed under CC-BY 3.0](#)

+

Style Image



[Starry Night](#) by Van Gogh is in the public domain

Gatys, Ecker, and Bethge, "Texture Synthesis Using Convolutional Neural Networks", NIPS 2015

Neural Style Transfer

Content Image



[This image](#) is licensed under CC-BY 3.0

+

Style Image



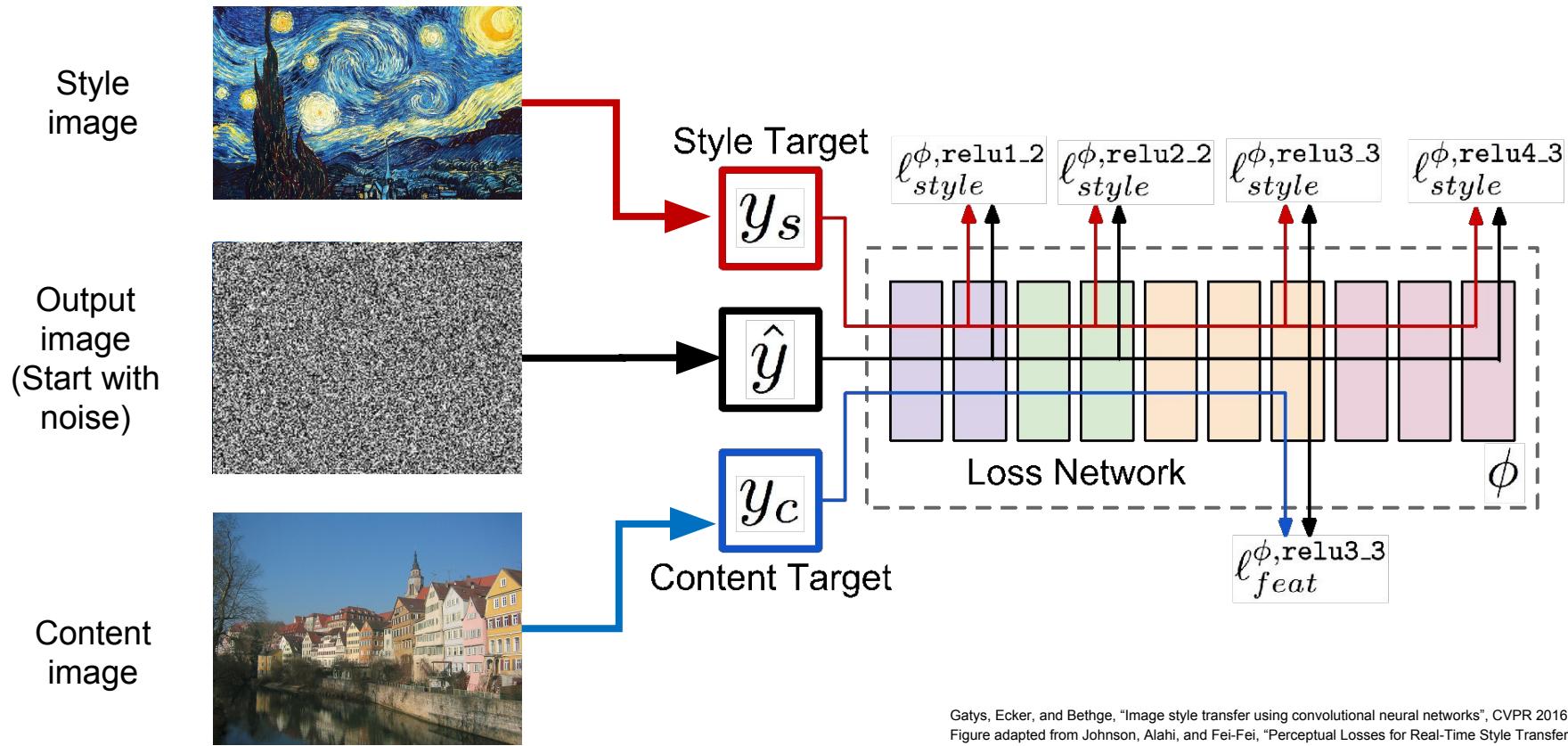
[Starry Night](#) by Van Gogh is in the public domain

=

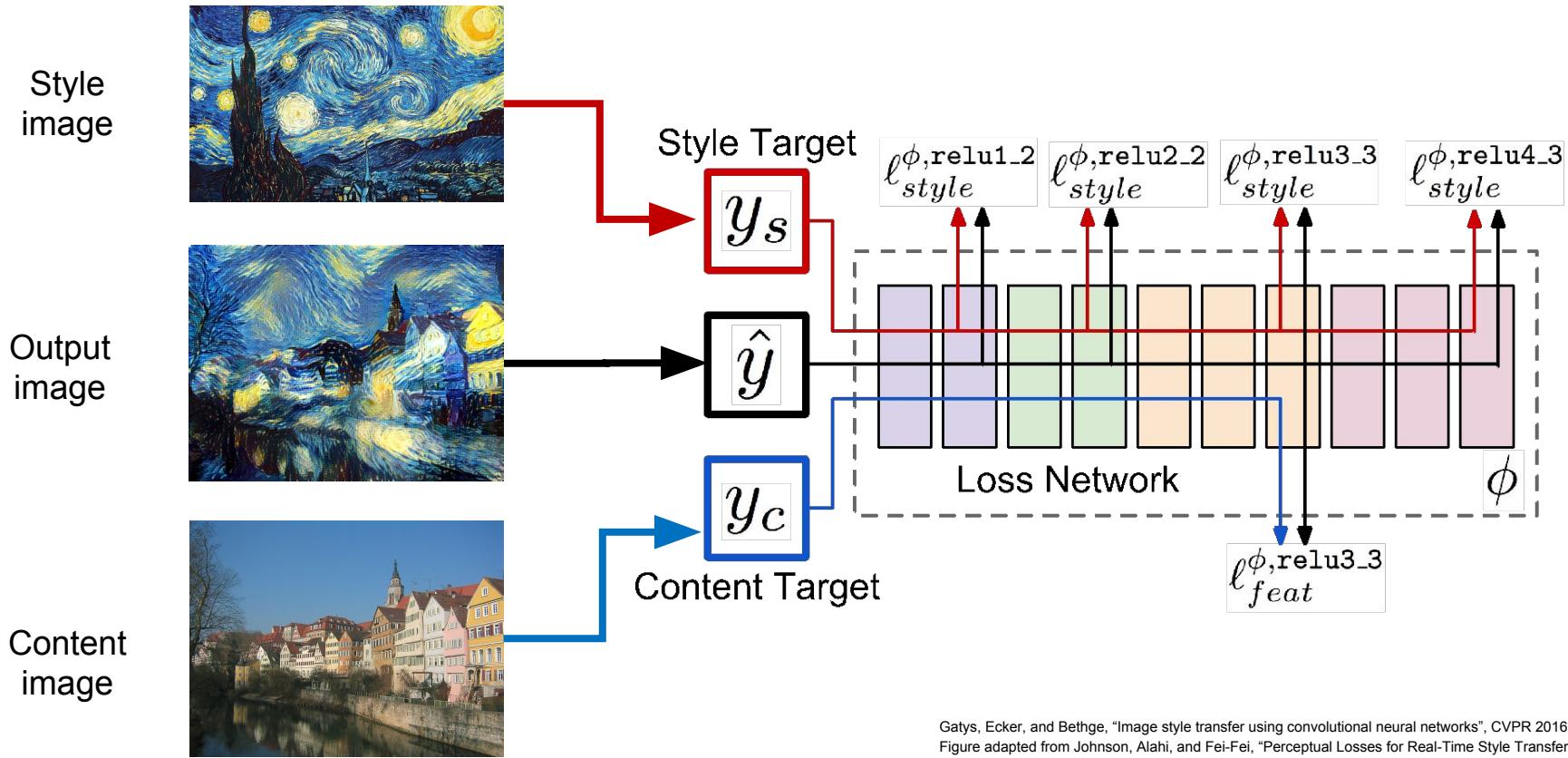


[This image](#) copyright Justin Johnson, 2015. Reproduced with permission.

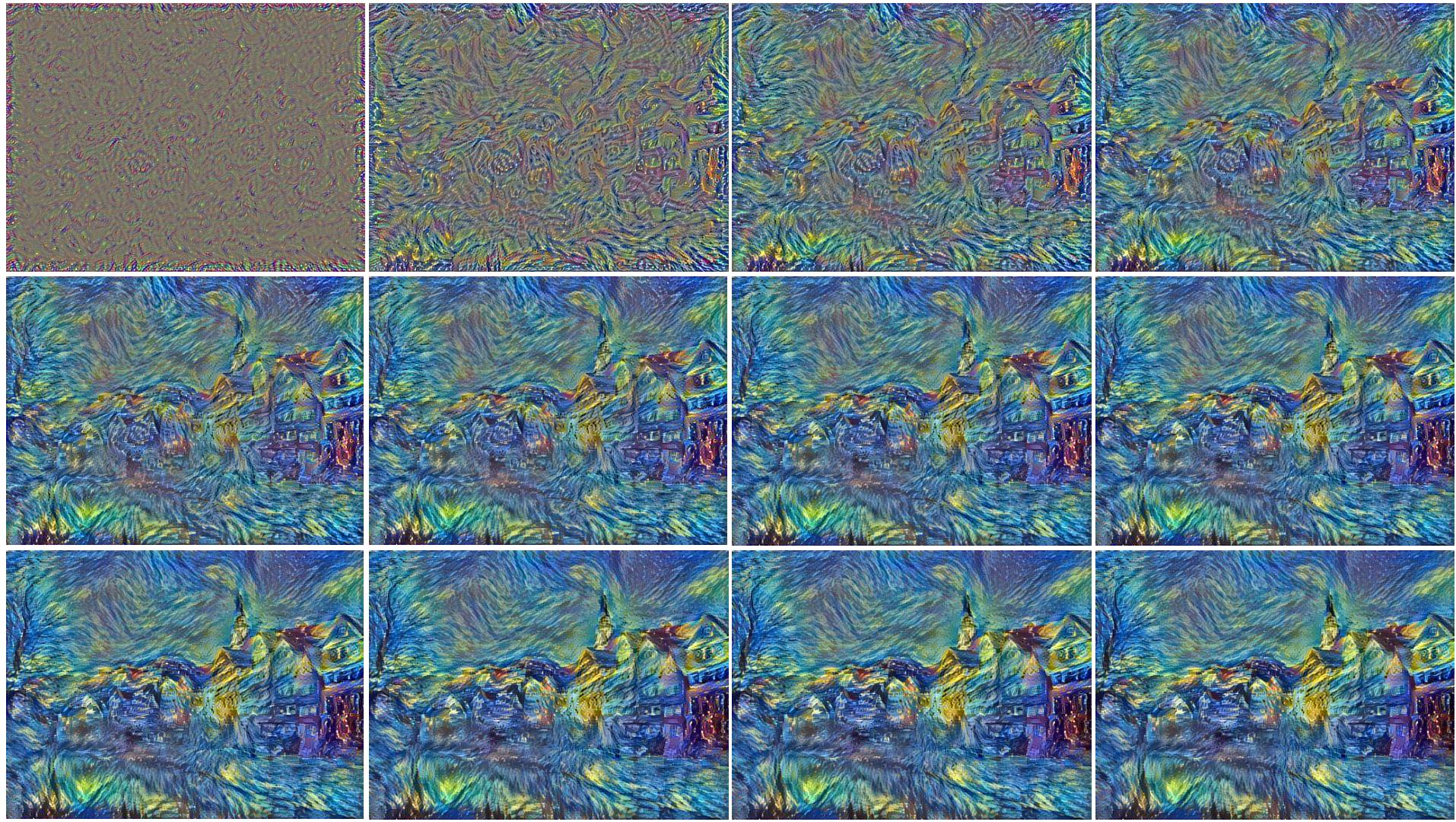
Gatys, Ecker, and Bethge, "Image style transfer using convolutional neural networks", CVPR 2016



Gatys, Ecker, and Bethge, "Image style transfer using convolutional neural networks", CVPR 2016
 Figure adapted from Johnson, Alahi, and Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution", ECCV 2016. Copyright Springer, 2016. Reproduced for educational purposes.



Gatys, Ecker, and Bethge, "Image style transfer using convolutional neural networks", CVPR 2016
 Figure adapted from Johnson, Alahi, and Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution", ECCV 2016. Copyright Springer, 2016. Reproduced for educational purposes.



Neural Style Transfer

Example outputs from
[my implementation](#)
(in Torch)



Gatys, Ecker, and Bethge, "Image style transfer using convolutional neural networks", CVPR 2016
Figure copyright Justin Johnson, 2015.

Neural Style Transfer



More weight to
content loss



More weight to
style loss

Neural Style Transfer

Resizing style image before running style transfer algorithm can transfer different types of features



Larger style
image

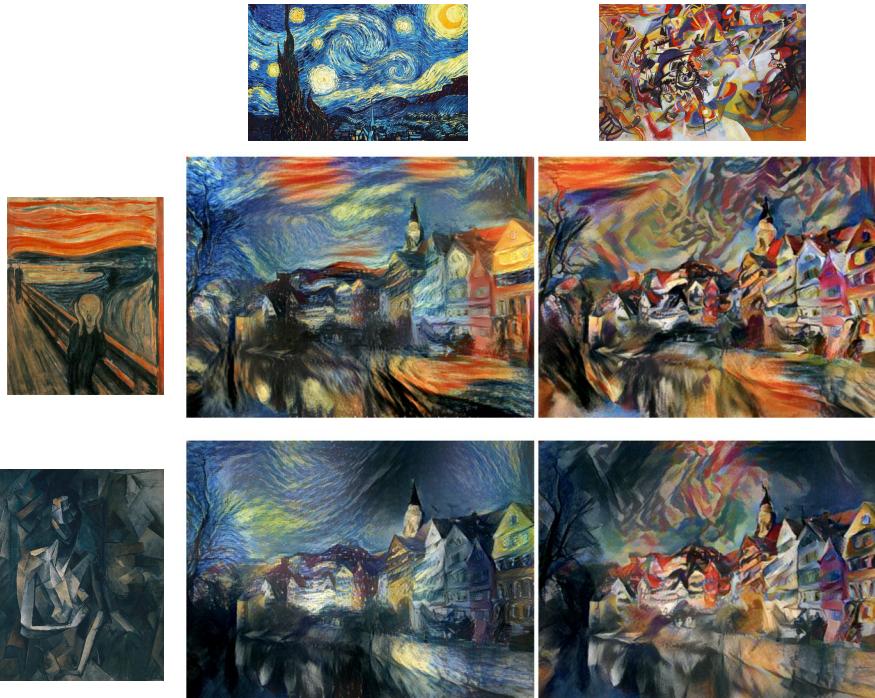


Smaller style
image

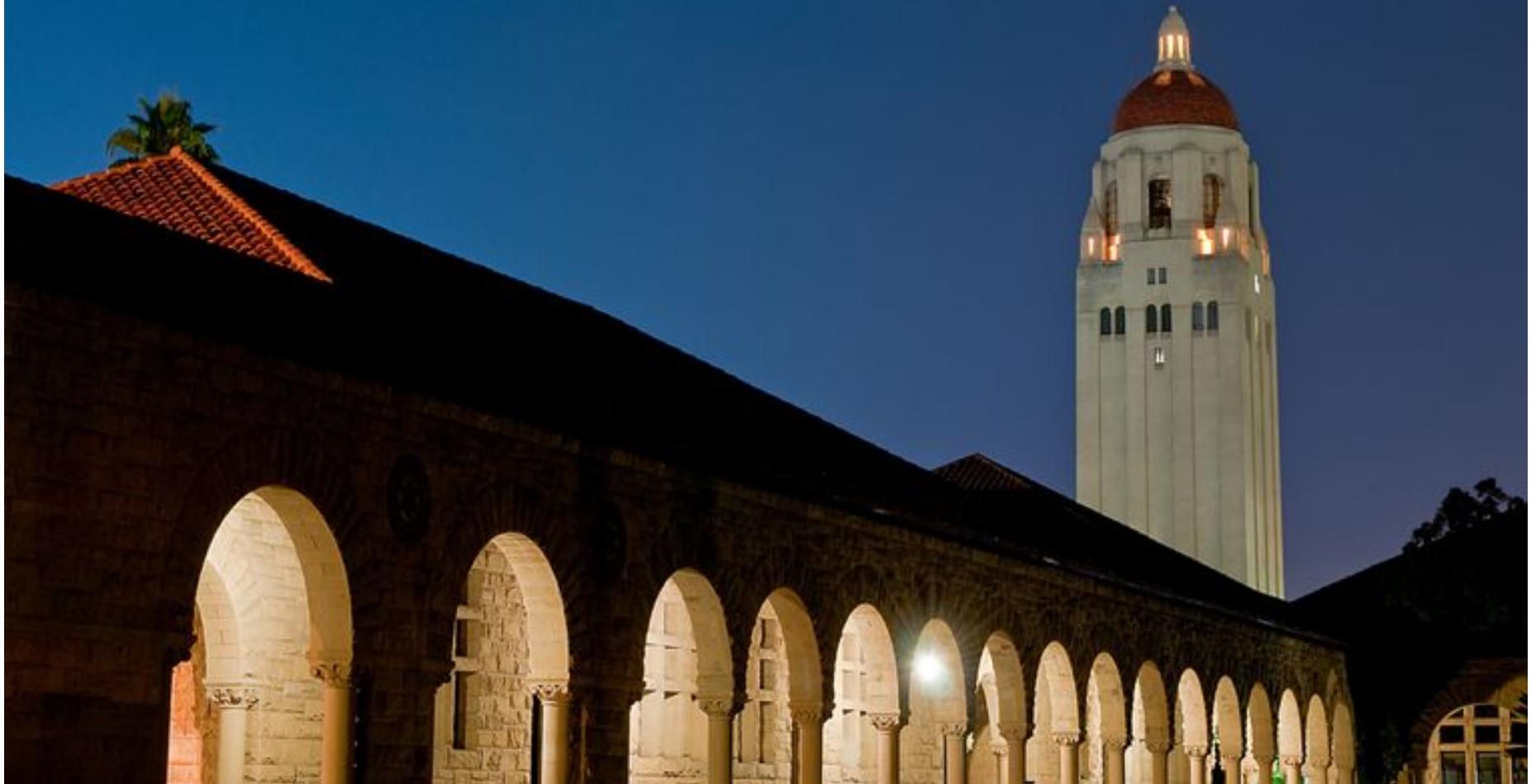
Gatys, Ecker, and Bethge, "Image style transfer using convolutional neural networks", CVPR 2016
Figure copyright Justin Johnson, 2015.

Neural Style Transfer: Multiple Style Images

Mix style from multiple images by taking a weighted average of Gram matrices



Gatys, Ecker, and Bethge, "Image style transfer using convolutional neural networks", CVPR 2016
Figure copyright Justin Johnson, 2015.



Fei-Fei Li & Justin Johnson & Serena Yeung

Lecture 13 - 74 May 17, 2018



Fei-Fei Li & Justin Johnson & Serena Yeung

Lecture 13 - 75 May 17, 2018



Fei-Fei Li & Justin Johnson & Serena Yeung

Lecture 13 - 76 May 17, 2018



Fei-Fei Li & Justin Johnson & Serena Yeung

Lecture 13 - 77 May 17, 2018

Neural Style Transfer

Problem: Style transfer
requires many forward /
backward passes through
VGG; very slow!

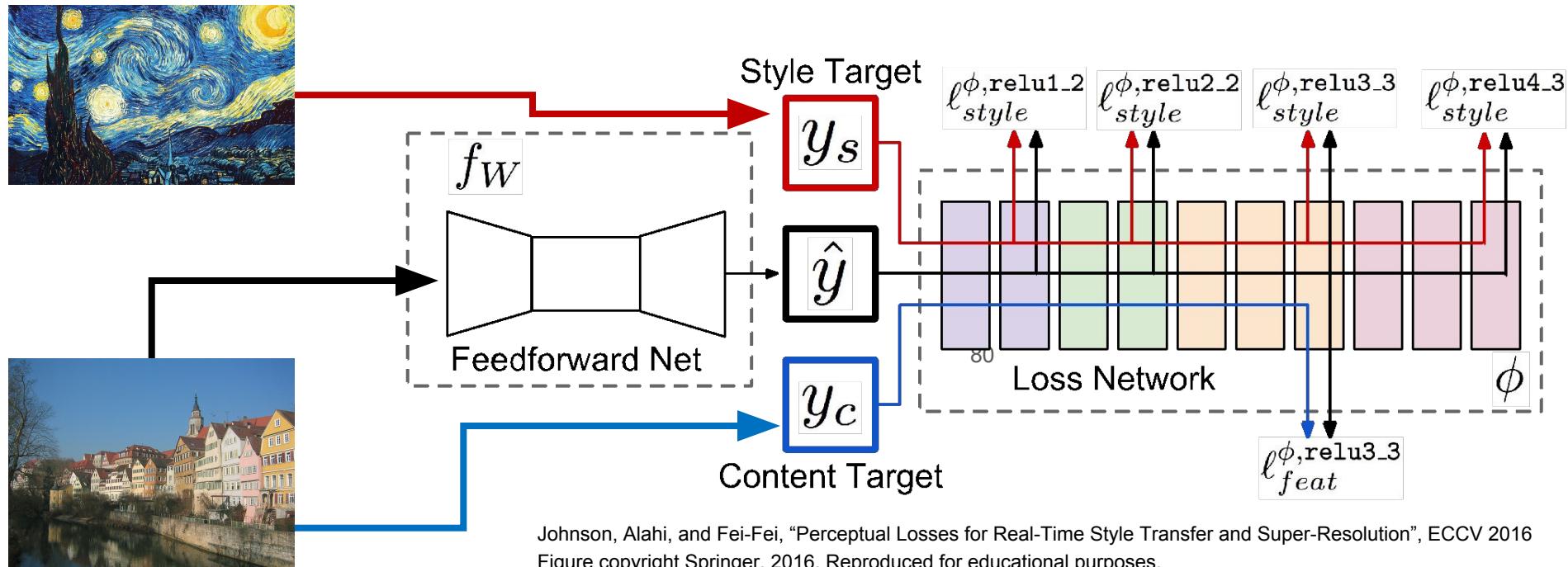
Neural Style Transfer

Problem: Style transfer requires many forward / backward passes through VGG; very slow!

Solution: Train another neural network to perform style transfer for us!

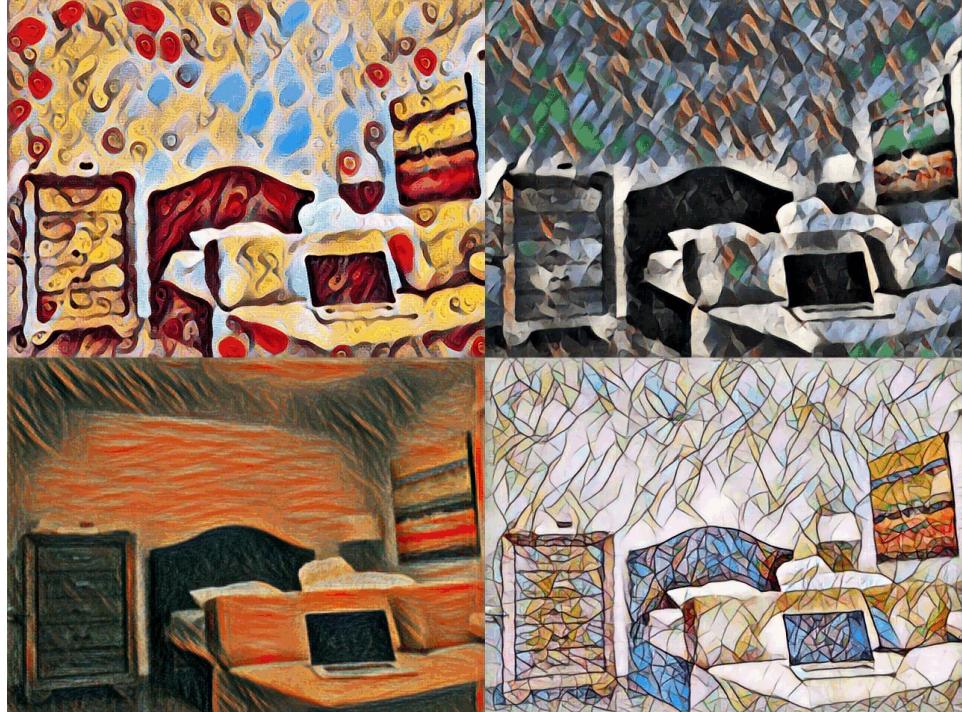
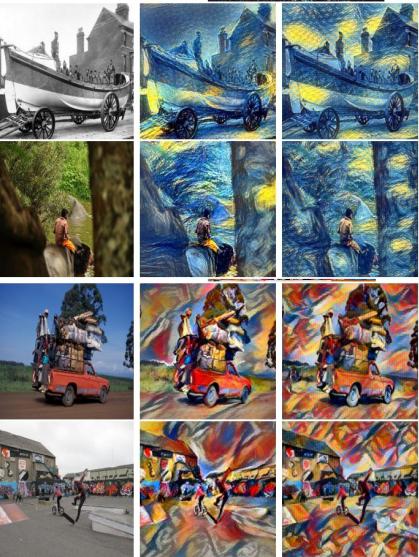
Fast Style Transfer

- (1) Train a feedforward network for each style
- (2) Use pretrained CNN to compute same losses as before
- (3) After training, stylize images using a single forward pass



Johnson, Alahi, and Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution", ECCV 2016
Figure copyright Springer, 2016. Reproduced for educational purposes.

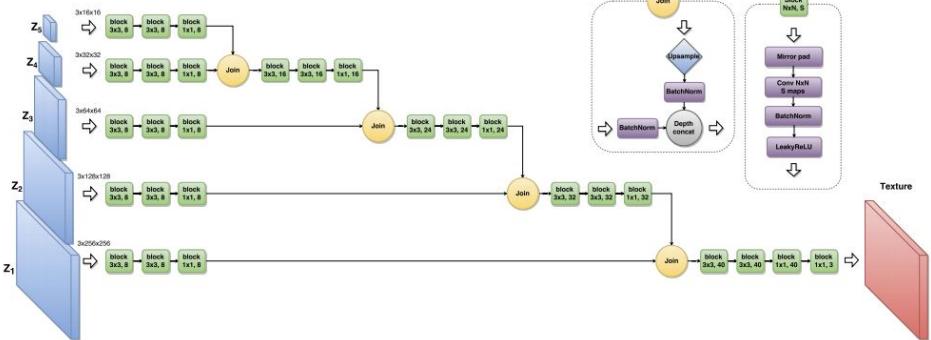
Fast Style Transfer



Johnson, Alahi, and Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution", ECCV 2016
Figure copyright Springer, 2016. Reproduced for educational purposes.

<https://github.com/jcjohnson/fast-neural-style>

Fast Style Transfer



Concurrent work from Ulyanov et al, comparable results

Ulyanov et al, "Texture Networks: Feed-forward Synthesis of Textures and Stylized Images", ICML 2016

Ulyanov et al, "Instance Normalization: The Missing Ingredient for Fast Stylization", arXiv 2016

Figures copyright Dmitry Ulyanov, Vadim Lebedev, Andrea Vedaldi, and Victor Lempitsky, 2016. Reproduced with permission.

Fast Style Transfer



Replacing batch normalization with Instance Normalization improves results

Ulyanov et al. "Texture Networks: Feed-forward Synthesis of Textures and Stylized Images", ICML 2016

Ulyanov et al. "Instance Normalization: The Missing Ingredient for Fast Stylization", arXiv 2016

Figures copyright Dmitry Ulyanov, Vadim Lebedev, Andrea Vedaldi, and Victor Lempitsky, 2016. Reproduced with permission.

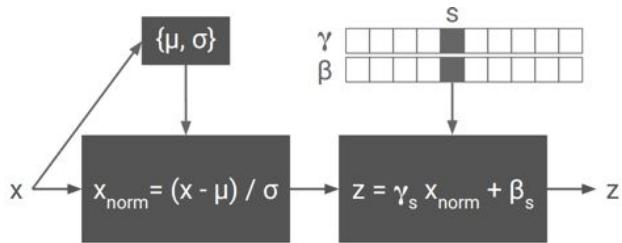
One Network, Many Styles



Dumoulin, Shlens, and Kudlur, "A Learned Representation for Artistic Style", ICLR 2017.
Figure copyright Vincent Dumoulin, Jonathon Shlens, and Manjunath Kudlur, 2016; reproduced with permission.

One Network, Many Styles

Use the same network for multiple styles using conditional instance normalization: learn separate scale and shift parameters per style



Single network can blend styles after training

Dumoulin, Shlens, and Kudlur, "A Learned Representation for Artistic Style", ICLR 2017.
Figure copyright Vincent Dumoulin, Jonathon Shlens, and Manjunath Kudlur, 2016; reproduced with permission.

Summary

Many methods for understanding CNN representations

Activations: Nearest neighbors, Dimensionality reduction, maximal patches, occlusion

Gradients: Saliency maps, class visualization, fooling images, feature inversion

Fun: DeepDream, Style Transfer.

Next time: (Deep) Reinforcement Learning

Lecture 14: Reinforcement Learning

Administrative

- All projects must be registered online (see Piazza for instructions), due Fri 11:59pm
- Midterm and A2 regrade requests also due Fri 11:59pm

So far... Supervised Learning

Data: (x, y)

x is data, y is label

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

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



→ Cat

Classification

This image is CC0 public domain

So far... Unsupervised Learning

Data: x

Just data, no labels!

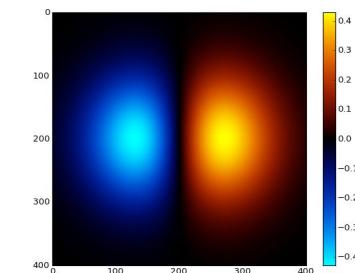
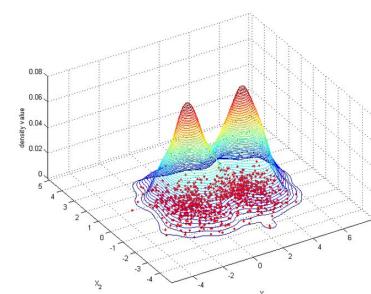
Goal: Learn some underlying hidden *structure* of the data

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



Figure copyright Ian Goodfellow, 2016. Reproduced with permission.

1-d density estimation



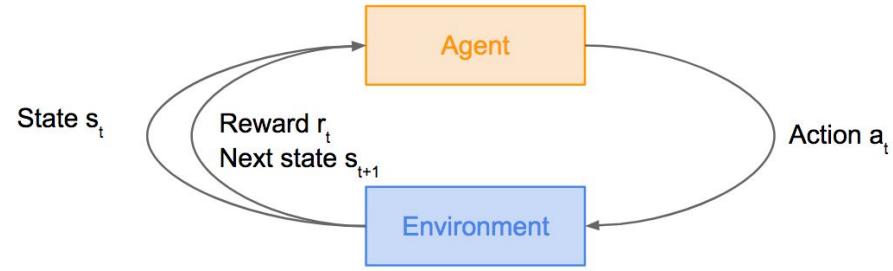
2-d density estimation

2-d density images [left](#) and [right](#)
are [CC0 public domain](#)

Today: Reinforcement Learning

Problems involving an **agent** interacting with an **environment**, which provides numeric **reward** signals

Goal: Learn how to take actions in order to maximize reward



Atari games figure copyright Volodymyr Mnih et al., 2013. Reproduced with permission.

Overview

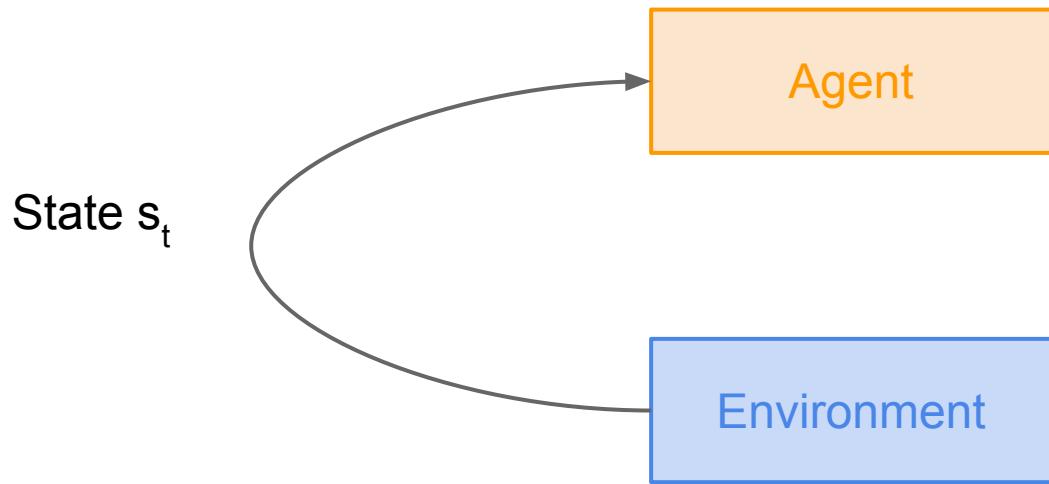
- What is Reinforcement Learning?
- Markov Decision Processes
- Q-Learning
- Policy Gradients

Reinforcement Learning

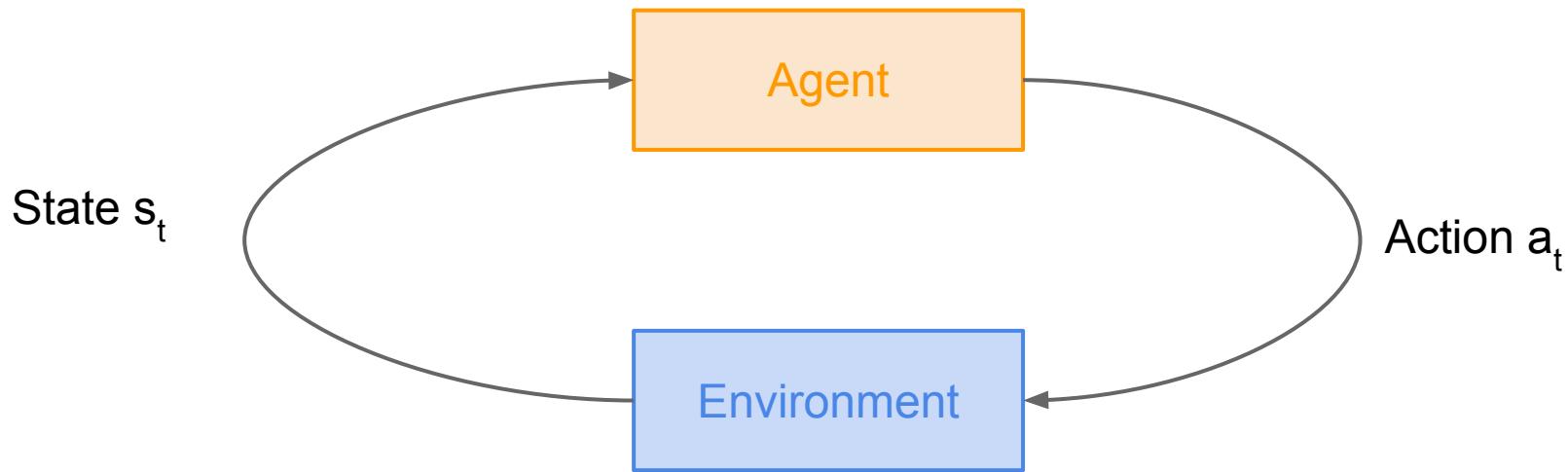
Agent

Environment

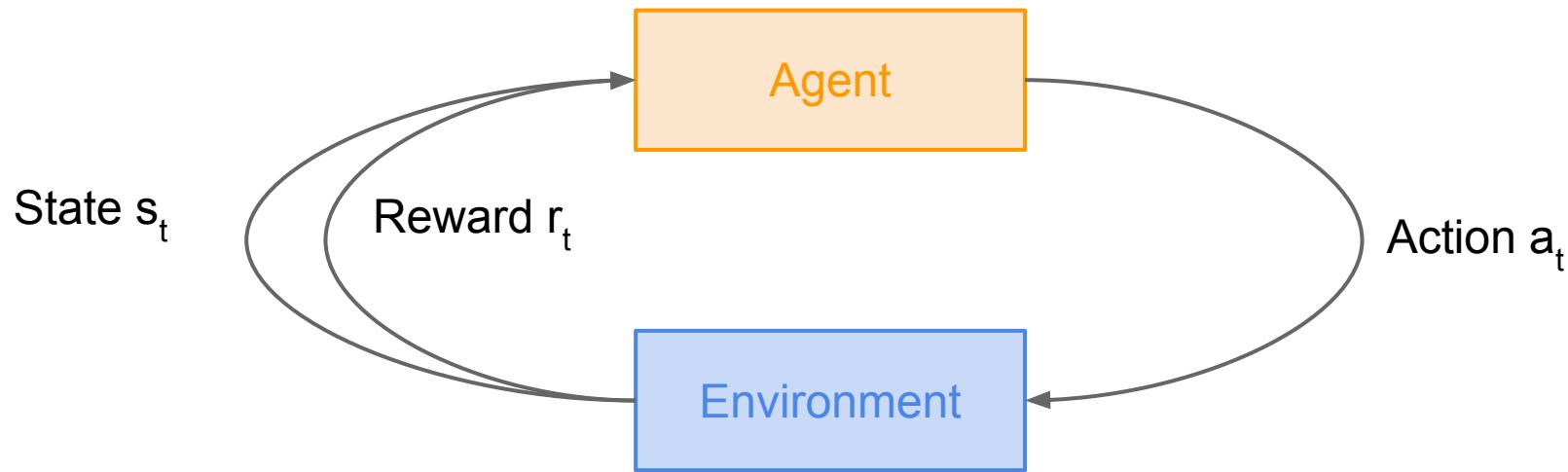
Reinforcement Learning



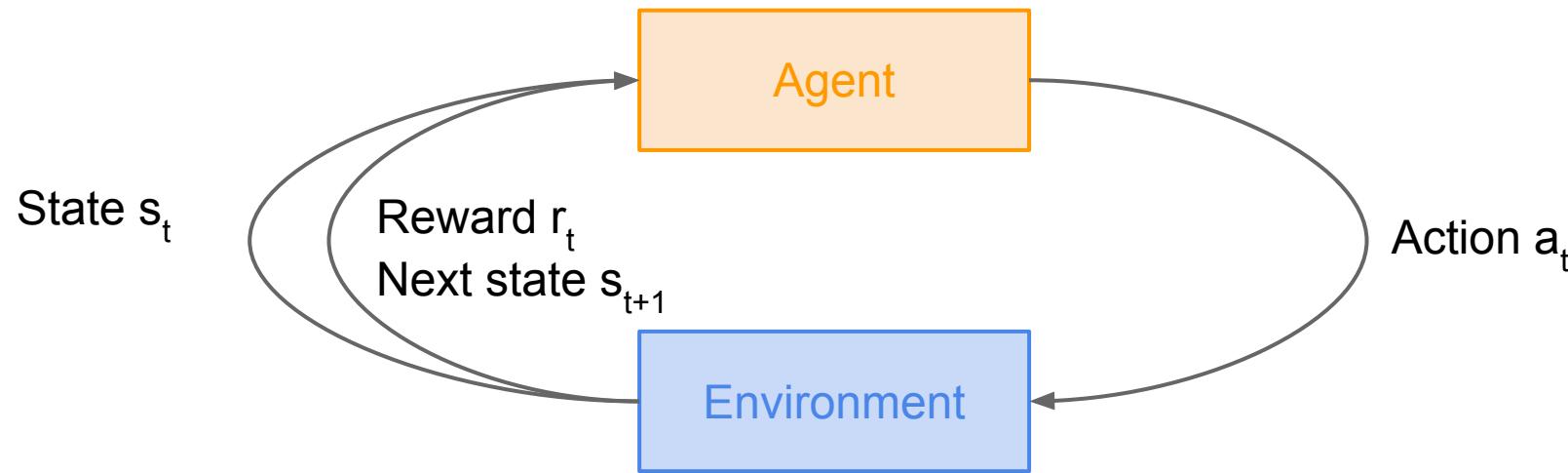
Reinforcement Learning



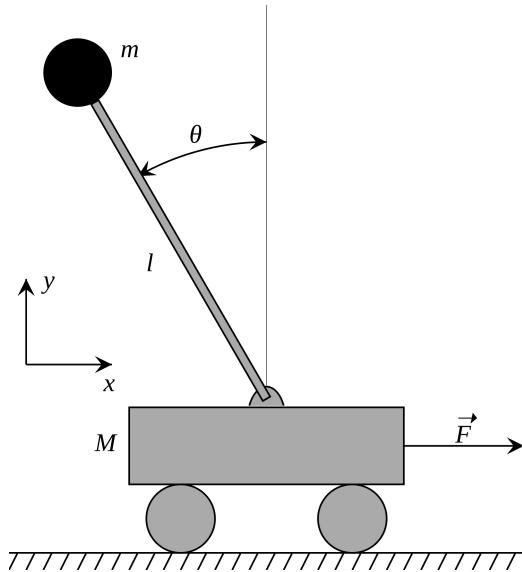
Reinforcement Learning



Reinforcement Learning



Cart-Pole Problem



Objective: Balance a pole on top of a movable cart

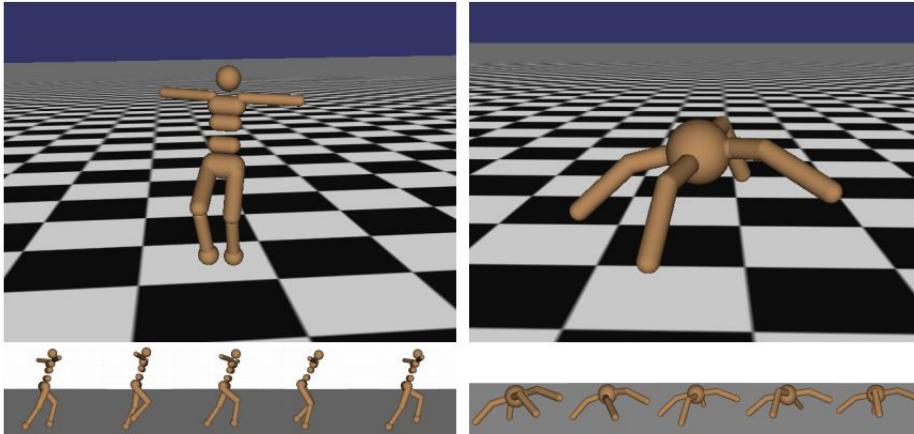
State: angle, angular speed, position, horizontal velocity

Action: horizontal force applied on the cart

Reward: 1 at each time step if the pole is upright

This image is CC0 public domain

Robot Locomotion



Objective: Make the robot move forward

State: Angle and position of the joints

Action: Torques applied on joints

Reward: 1 at each time step upright +
forward movement

Figures copyright John Schulman et al., 2016. Reproduced with permission.

Atari Games



Objective: Complete the game with the highest score

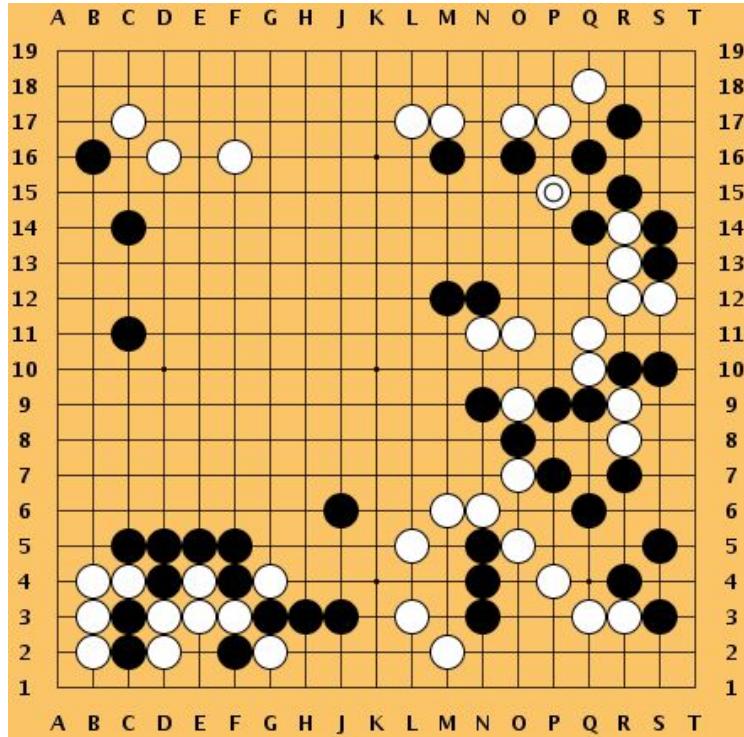
State: Raw pixel inputs of the game state

Action: Game controls e.g. Left, Right, Up, Down

Reward: Score increase/decrease at each time step

Figures copyright Volodymyr Mnih et al., 2013. Reproduced with permission.

Go



Objective: Win the game!

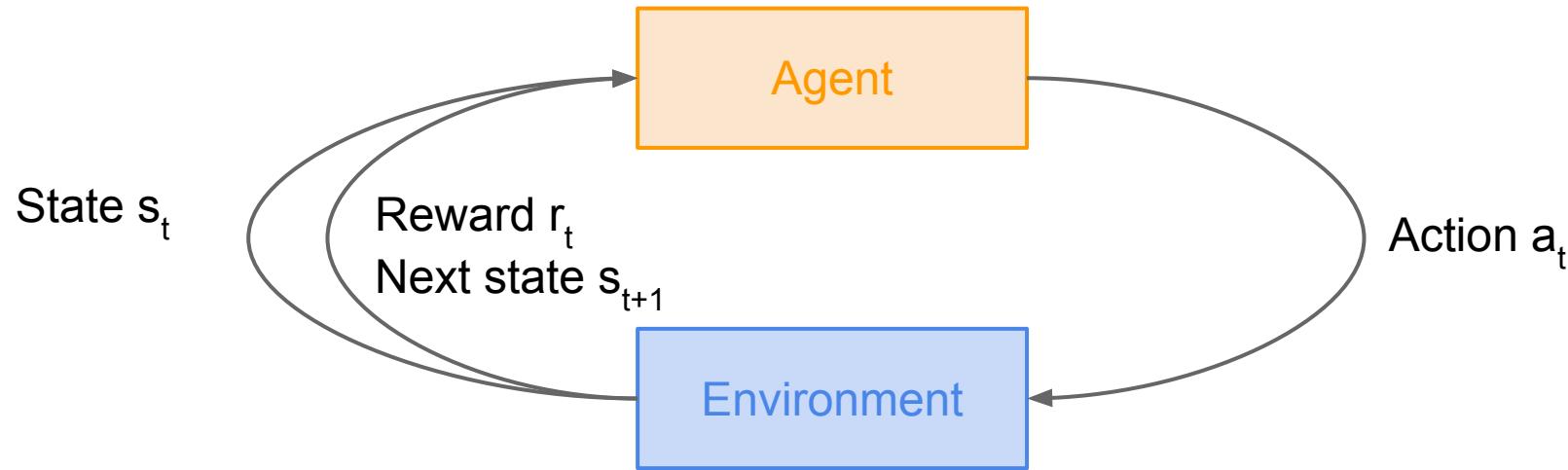
State: Position of all pieces

Action: Where to put the next piece down

Reward: 1 if win at the end of the game, 0 otherwise

This image is CC0 public domain

How can we mathematically formalize the RL problem?



Markov Decision Process

- Mathematical formulation of the RL problem
- **Markov property:** Current state completely characterises the state of the world

Defined by: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$

\mathcal{S} : set of possible states

\mathcal{A} : set of possible actions

\mathcal{R} : distribution of reward given (state, action) pair

\mathbb{P} : transition probability i.e. distribution over next state given (state, action) pair

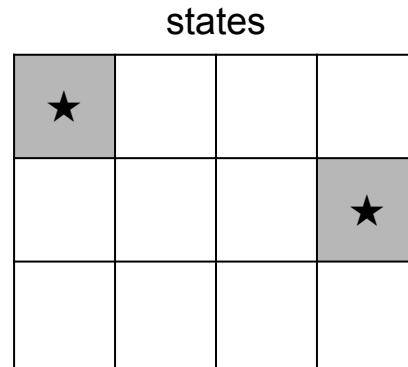
γ : discount factor

Markov Decision Process

- At time step $t=0$, environment samples initial state $s_0 \sim p(s_0)$
- Then, for $t=0$ until done:
 - Agent selects action a_t
 - Environment samples reward $r_t \sim R(\cdot | s_t, a_t)$
 - Environment samples next state $s_{t+1} \sim P(\cdot | s_t, a_t)$
 - Agent receives reward r_t and next state s_{t+1}
- A policy π is a function from S to A that specifies what action to take in each state
- **Objective:** find policy π^* that maximizes cumulative discounted reward:
$$\sum_{t \geq 0} \gamma^t r_t$$

A simple MDP: Grid World

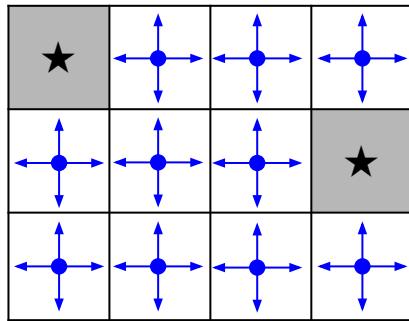
```
actions = {  
    1. right →  
    2. left ←  
    3. up ↑  
    4. down ↓  
}
```



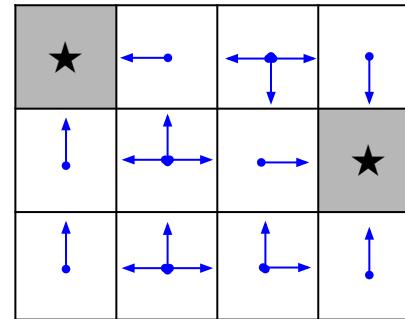
Set a negative “reward”
for each transition
(e.g. $r = -1$)

Objective: reach one of terminal states (greyed out) in
least number of actions

A simple MDP: Grid World



Random Policy



Optimal Policy

The optimal policy π^*

We want to find optimal policy π^* that maximizes the sum of rewards.

How do we handle the randomness (initial state, transition probability...)?

The optimal policy π^*

We want to find optimal policy π^* that maximizes the sum of rewards.

How do we handle the randomness (initial state, transition probability...)?
Maximize the **expected sum of rewards!**

Formally: $\pi^* = \arg \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | \pi \right]$ with $s_0 \sim p(s_0), a_t \sim \pi(\cdot | s_t), s_{t+1} \sim p(\cdot | s_t, a_t)$

Definitions: Value function and Q-value function

Following a policy produces sample trajectories (or paths) $s_0, a_0, r_0, s_1, a_1, r_1, \dots$

Definitions: Value function and Q-value function

Following a policy produces sample trajectories (or paths) $s_0, a_0, r_0, s_1, a_1, r_1, \dots$

How good is a state?

The **value function** at state s , is the expected cumulative reward from following the policy from state s :

$$V^\pi(s) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi \right]$$

Definitions: Value function and Q-value function

Following a policy produces sample trajectories (or paths) $s_0, a_0, r_0, s_1, a_1, r_1, \dots$

How good is a state?

The **value function** at state s , is the expected cumulative reward from following the policy from state s :

$$V^\pi(s) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi \right]$$

How good is a state-action pair?

The **Q-value function** at state s and action a , is the expected cumulative reward from taking action a in state s and then following the policy:

$$Q^\pi(s, a) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]$$

Bellman equation

The optimal Q-value function Q^* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s, a) = \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]$$

Bellman equation

The optimal Q-value function Q^* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s, a) = \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]$$

Q^* satisfies the following **Bellman equation**:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Intuition: if the optimal state-action values for the next time-step $Q^*(s', a')$ are known, then the optimal strategy is to take the action that maximizes the expected value of $r + \gamma Q^*(s', a')$

Bellman equation

The optimal Q-value function Q^* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s, a) = \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]$$

Q^* satisfies the following **Bellman equation**:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Intuition: if the optimal state-action values for the next time-step $Q^*(s', a')$ are known, then the optimal strategy is to take the action that maximizes the expected value of $r + \gamma Q^*(s', a')$

The optimal policy π^* corresponds to taking the best action in any state as specified by Q^*

Solving for the optimal policy

Value iteration algorithm: Use Bellman equation as an iterative update

$$Q_{i+1}(s, a) = \mathbb{E} \left[r + \gamma \max_{a'} Q_i(s', a') | s, a \right]$$

Q_i will converge to Q^* as $i \rightarrow \infty$

Solving for the optimal policy

Value iteration algorithm: Use Bellman equation as an iterative update

$$Q_{i+1}(s, a) = \mathbb{E} \left[r + \gamma \max_{a'} Q_i(s', a') | s, a \right]$$

Q_i will converge to Q^* as $i \rightarrow \infty$

What's the problem with this?

Solving for the optimal policy

Value iteration algorithm: Use Bellman equation as an iterative update

$$Q_{i+1}(s, a) = \mathbb{E} \left[r + \gamma \max_{a'} Q_i(s', a') | s, a \right]$$

Q_i will converge to Q^* as $i \rightarrow \infty$

What's the problem with this?

Not scalable. Must compute $Q(s, a)$ for every state-action pair. If state is e.g. current game state pixels, computationally infeasible to compute for entire state space!

Solving for the optimal policy

Value iteration algorithm: Use Bellman equation as an iterative update

$$Q_{i+1}(s, a) = \mathbb{E} \left[r + \gamma \max_{a'} Q_i(s', a') | s, a \right]$$

Q_i will converge to Q^* as $i \rightarrow \infty$

What's the problem with this?

Not scalable. Must compute $Q(s, a)$ for every state-action pair. If state is e.g. current game state pixels, computationally infeasible to compute for entire state space!

Solution: use a function approximator to estimate $Q(s, a)$. E.g. a neural network!

Solving for the optimal policy: Q-learning

Q-learning: Use a function approximator to estimate the action-value function

$$Q(s, a; \theta) \approx Q^*(s, a)$$

Solving for the optimal policy: Q-learning

Q-learning: Use a function approximator to estimate the action-value function

$$Q(s, a; \theta) \approx Q^*(s, a)$$

If the function approximator is a deep neural network => **deep q-learning!**

Solving for the optimal policy: Q-learning

Q-learning: Use a function approximator to estimate the action-value function

$$Q(s, a; \theta) \approx Q^*(s, a)$$

function parameters (weights)

If the function approximator is a deep neural network => **deep q-learning!**

Solving for the optimal policy: Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Solving for the optimal policy: Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} [(y_i - Q(s, a; \theta_i))^2]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$

Solving for the optimal policy: Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} [(y_i - Q(s, a; \theta_i))^2]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$

Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)) \nabla_{\theta_i} Q(s, a; \theta_i) \right]$$

Solving for the optimal policy: Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} [(y_i - Q(s, a; \theta_i))^2]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$

Iteratively try to make the Q-value close to the target value (y_i) it should have, if Q-function corresponds to optimal Q^* (and optimal policy π^*)

Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right] \nabla_{\theta_i} Q(s, a; \theta_i)$$

Case Study: Playing Atari Games



Objective: Complete the game with the highest score

State: Raw pixel inputs of the game state

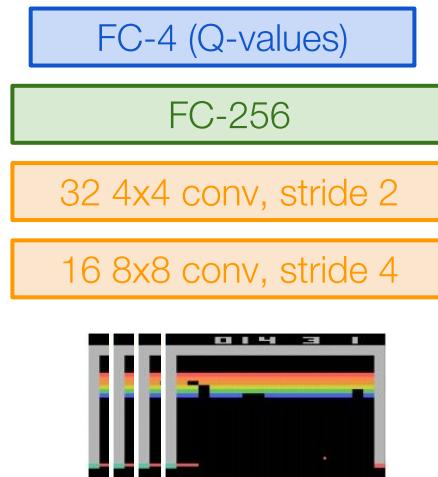
Action: Game controls e.g. Left, Right, Up, Down

Reward: Score increase/decrease at each time step

Figures copyright Volodymyr Mnih et al., 2013. Reproduced with permission.

Q-network Architecture

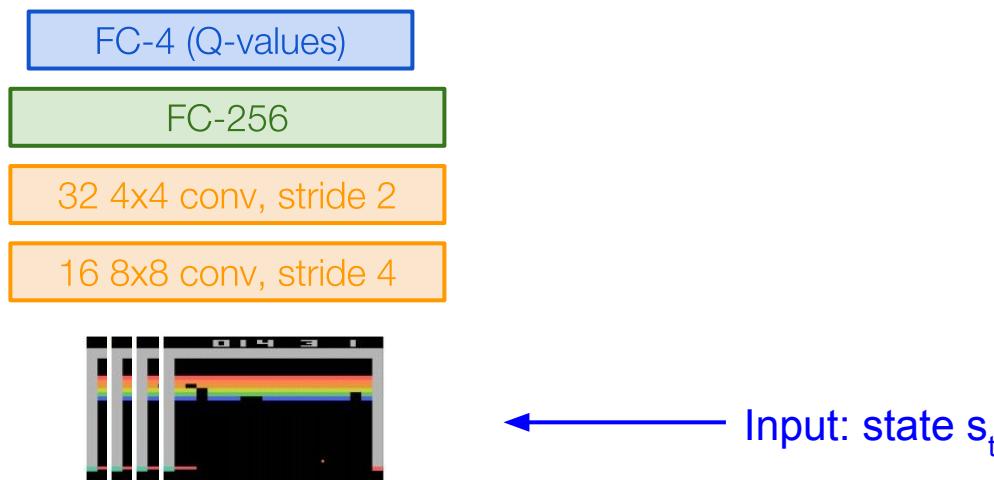
$Q(s, a; \theta)$:
neural network
with weights θ



Current state s_t : 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

Q-network Architecture

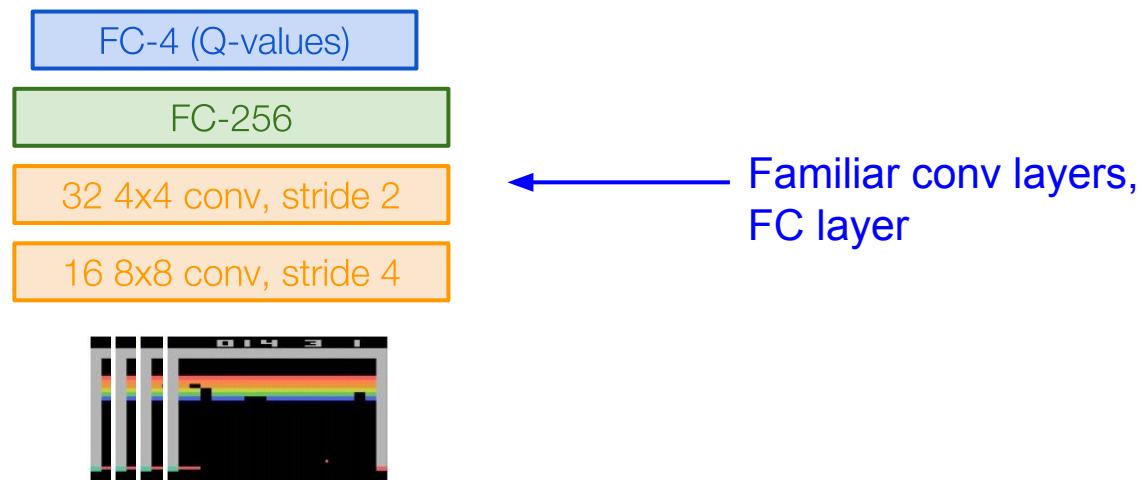
$Q(s, a; \theta)$:
neural network
with weights θ



Current state s_t : 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

Q-network Architecture

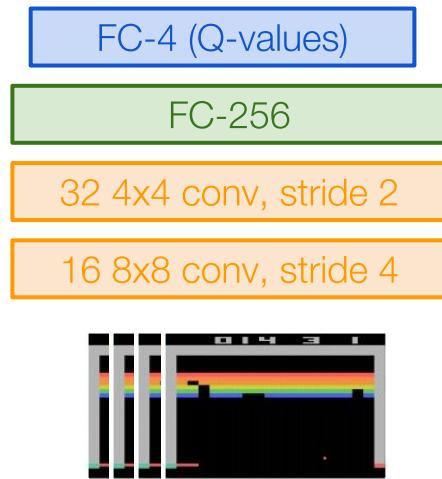
$Q(s, a; \theta)$:
neural network
with weights θ



Current state s_t : 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

Q-network Architecture

$Q(s, a; \theta)$:
neural network
with weights θ

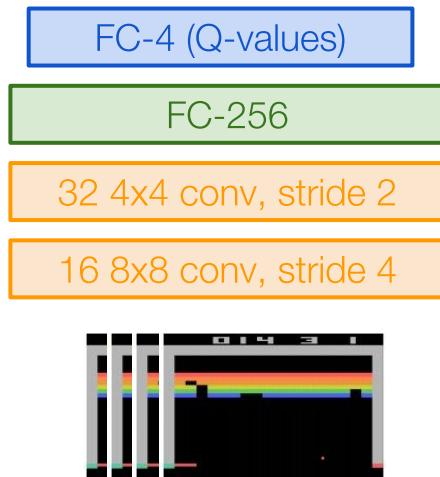


Last FC layer has 4-d output (if 4 actions), corresponding to $Q(s_t, a_1), Q(s_t, a_2), Q(s_t, a_3), Q(s_t, a_4)$

Current state s_t : 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

Q-network Architecture

$Q(s, a; \theta)$:
neural network
with weights θ



Last FC layer has 4-d output (if 4 actions), corresponding to $Q(s_t, a_1)$, $Q(s_t, a_2)$, $Q(s_t, a_3)$, $Q(s_t, a_4)$

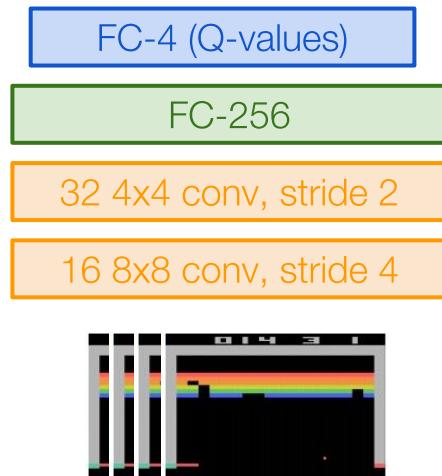
Number of actions between 4-18 depending on Atari game

Current state s_t : 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

Q-network Architecture

$Q(s, a; \theta)$:
neural network
with weights θ

A single feedforward pass
to compute Q-values for all
actions from the current
state => efficient!



Last FC layer has 4-d output (if 4 actions), corresponding to $Q(s_t, a_1), Q(s_t, a_2), Q(s_t, a_3), Q(s_t, a_4)$

Number of actions between 4-18 depending on Atari game

Current state s_t : 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

Training the Q-network: Loss function (from before)

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} [(y_i - Q(s, a; \theta_i))^2]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$

Iteratively try to make the Q-value close to the target value (y_i) it should have, if Q-function corresponds to optimal Q^* (and optimal policy π^*)

Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right] \nabla_{\theta_i} Q(s, a; \theta_i)$$

Training the Q-network: Experience Replay

Learning from batches of consecutive samples is problematic:

- Samples are correlated => inefficient learning
- Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand side) => can lead to bad feedback loops

Training the Q-network: Experience Replay

Learning from batches of consecutive samples is problematic:

- Samples are correlated => inefficient learning
- Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand side) => can lead to bad feedback loops

Address these problems using **experience replay**

- Continually update a **replay memory** table of transitions (s_t, a_t, r_t, s_{t+1}) as game (experience) episodes are played
- Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples

Training the Q-network: Experience Replay

Learning from batches of consecutive samples is problematic:

- Samples are correlated => inefficient learning
- Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand side) => can lead to bad feedback loops

Address these problems using **experience replay**

- Continually update a **replay memory** table of transitions (s_t, a_t, r_t, s_{t+1}) as game (experience) episodes are played
- Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples

Each transition can also contribute
to multiple weight updates
=> greater data efficiency

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

end for

end for

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N ← Initialize replay memory, Q-network
 Initialize action-value function Q with random weights
for episode = 1, M **do**
 Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
 for $t = 1, T$ **do**
 With probability ϵ select a random action a_t
 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
 Execute action a_t in emulator and observe reward r_t and image x_{t+1}
 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}
 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}
 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
 end for
end for

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N
 Initialize action-value function Q with random weights
for episode = 1, M **do** ← Play M episodes (full games)

- Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
- for** $t = 1, T$ **do**
- With probability ϵ select a random action a_t
- otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
- Execute action a_t in emulator and observe reward r_t and image x_{t+1}
- Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
- Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}
- Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}
- Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
- Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

end for

end for

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N
 Initialize action-value function Q with random weights
for episode = 1, M **do**

- Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$ ← Initialize state (starting game screen pixels) at the beginning of each episode
- for** $t = 1, T$ **do**
- With probability ϵ select a random action a_t
- otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
- Execute action a_t in emulator and observe reward r_t and image x_{t+1}
- Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
- Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}
- Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}
- Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
- Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

end for

end for

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N
 Initialize action-value function Q with random weights
for episode = 1, M **do**

- Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
- for** $t = 1, T$ **do**
- With probability ϵ select a random action a_t
- otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
- Execute action a_t in emulator and observe reward r_t and image x_{t+1}
- Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
- Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}
- Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}
- Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
- Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

end for

end for

← For each timestep t of the game

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

end for

end for



With small probability,
select a random
action (explore),
otherwise select
greedy action from
current policy

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

end for

end for



Take the action (a_t) ,
and observe the
reward r_t and next
state s_{t+1}

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

end for

end for



Store transition in
replay memory

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

end for

end for



Experience Replay:
Sample a random minibatch of transitions from replay memory and perform a gradient descent step



<https://www.youtube.com/watch?v=V1eYniJ0Rnk>

Video by Károly Zsolnai-Fehér. Reproduced with permission.

Policy Gradients

What is a problem with Q-learning?

The Q-function can be very complicated!

Example: a robot grasping an object has a very high-dimensional state => hard to learn exact value of every (state, action) pair

Policy Gradients

What is a problem with Q-learning?

The Q-function can be very complicated!

Example: a robot grasping an object has a very high-dimensional state => hard to learn exact value of every (state, action) pair

But the policy can be much simpler: just close your hand

Can we learn a policy directly, e.g. finding the best policy from a collection of policies?

Policy Gradients

Formally, let's define a class of parameterized policies: $\Pi = \{\pi_\theta, \theta \in \mathbb{R}^m\}$

For each policy, define its value:

$$J(\theta) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | \pi_\theta \right]$$

Policy Gradients

Formally, let's define a class of parameterized policies: $\Pi = \{\pi_\theta, \theta \in \mathbb{R}^m\}$

For each policy, define its value:

$$J(\theta) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | \pi_\theta \right]$$

We want to find the optimal policy $\theta^* = \arg \max_{\theta} J(\theta)$

How can we do this?

Policy Gradients

Formally, let's define a class of parameterized policies: $\Pi = \{\pi_\theta, \theta \in \mathbb{R}^m\}$

For each policy, define its value:

$$J(\theta) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t | \pi_\theta \right]$$

We want to find the optimal policy $\theta^* = \arg \max_{\theta} J(\theta)$

How can we do this?

Gradient ascent on policy parameters!

REINFORCE algorithm

Mathematically, we can write:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\tau \sim p(\tau; \theta)} [r(\tau)] \\ &= \int_{\tau} r(\tau)p(\tau; \theta)d\tau \end{aligned}$$

Where $r(\tau)$ is the reward of a trajectory $\tau = (s_0, a_0, r_0, s_1, \dots)$

REINFORCE algorithm

Expected reward:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\tau \sim p(\tau; \theta)} [r(\tau)] \\ &= \int_{\tau} r(\tau)p(\tau; \theta)d\tau \end{aligned}$$

REINFORCE algorithm

Expected reward:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\tau \sim p(\tau; \theta)} [r(\tau)] \\ &= \int_{\tau} r(\tau)p(\tau; \theta)d\tau \end{aligned}$$

Now let's differentiate this:

$$\nabla_{\theta} J(\theta) = \int_{\tau} r(\tau)\nabla_{\theta} p(\tau; \theta)d\tau$$

REINFORCE algorithm

Expected reward:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\tau \sim p(\tau; \theta)} [r(\tau)] \\ &= \int_{\tau} r(\tau)p(\tau; \theta)d\tau \end{aligned}$$

Now let's differentiate this:

$$\nabla_{\theta} J(\theta) = \int_{\tau} r(\tau)\nabla_{\theta} p(\tau; \theta)d\tau$$

Intractable! Gradient of an expectation is problematic when p depends on θ

REINFORCE algorithm

Expected reward:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\tau \sim p(\tau; \theta)} [r(\tau)] \\ &= \int_{\tau} r(\tau)p(\tau; \theta)d\tau \end{aligned}$$

Now let's differentiate this:

$$\nabla_{\theta} J(\theta) = \int_{\tau} r(\tau)\nabla_{\theta}p(\tau; \theta)d\tau$$

Intractable! Gradient of an expectation is problematic when p depends on θ

However, we can use a nice trick:

$$\nabla_{\theta}p(\tau; \theta) = p(\tau; \theta) \frac{\nabla_{\theta}p(\tau; \theta)}{p(\tau; \theta)} = p(\tau; \theta)\nabla_{\theta} \log p(\tau; \theta)$$

REINFORCE algorithm

Expected reward:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\tau \sim p(\tau; \theta)} [r(\tau)] \\ &= \int_{\tau} r(\tau)p(\tau; \theta)d\tau \end{aligned}$$

Now let's differentiate this:

$$\nabla_{\theta} J(\theta) = \int_{\tau} r(\tau)\nabla_{\theta}p(\tau; \theta)d\tau$$

Intractable! Gradient of an expectation is problematic when p depends on θ

However, we can use a nice trick: $\nabla_{\theta}p(\tau; \theta) = p(\tau; \theta) \frac{\nabla_{\theta}p(\tau; \theta)}{p(\tau; \theta)} = p(\tau; \theta)\nabla_{\theta} \log p(\tau; \theta)$
If we inject this back:

$$\begin{aligned} \nabla_{\theta} J(\theta) &= \int_{\tau} (r(\tau)\nabla_{\theta} \log p(\tau; \theta)) p(\tau; \theta)d\tau \\ &= \mathbb{E}_{\tau \sim p(\tau; \theta)} [r(\tau)\nabla_{\theta} \log p(\tau; \theta)] \end{aligned}$$

Can estimate with
Monte Carlo sampling

REINFORCE algorithm

Can we compute those quantities without knowing the transition probabilities?

We have: $p(\tau; \theta) = \prod_{t \geq 0} p(s_{t+1} | s_t, a_t) \pi_\theta(a_t | s_t)$

REINFORCE algorithm

Can we compute those quantities without knowing the transition probabilities?

We have: $p(\tau; \theta) = \prod_{t \geq 0} p(s_{t+1}|s_t, a_t) \pi_\theta(a_t|s_t)$

Thus: $\log p(\tau; \theta) = \sum_{t \geq 0} \log p(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t)$

REINFORCE algorithm

Can we compute those quantities without knowing the transition probabilities?

We have: $p(\tau; \theta) = \prod_{t \geq 0} p(s_{t+1}|s_t, a_t) \pi_\theta(a_t|s_t)$

Thus: $\log p(\tau; \theta) = \sum_{t \geq 0} \log p(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t)$

And when differentiating: $\nabla_\theta \log p(\tau; \theta) = \sum_{t \geq 0} \nabla_\theta \log \pi_\theta(a_t|s_t)$

Doesn't depend on
transition probabilities!

REINFORCE algorithm

$$\begin{aligned}\nabla_{\theta} J(\theta) &= \int_{\tau} (r(\tau) \nabla_{\theta} \log p(\tau; \theta)) p(\tau; \theta) d\tau \\ &= \mathbb{E}_{\tau \sim p(\tau; \theta)} [r(\tau) \nabla_{\theta} \log p(\tau; \theta)]\end{aligned}$$

Can we compute those quantities without knowing the transition probabilities?

We have: $p(\tau; \theta) = \prod_{t \geq 0} p(s_{t+1} | s_t, a_t) \pi_{\theta}(a_t | s_t)$

Thus: $\log p(\tau; \theta) = \sum_{t \geq 0} \log p(s_{t+1} | s_t, a_t) + \log \pi_{\theta}(a_t | s_t)$

And when differentiating: $\nabla_{\theta} \log p(\tau; \theta) = \sum_{t \geq 0} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$

Doesn't depend on
transition probabilities!

Therefore when sampling a trajectory τ , we can estimate $J(\theta)$ with

$$\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

Intuition

Gradient estimator: $\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$

Interpretation:

- If $r(\tau)$ is high, push up the probabilities of the actions seen
- If $r(\tau)$ is low, push down the probabilities of the actions seen

Intuition

Gradient estimator: $\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$

Interpretation:

- If $r(\tau)$ is high, push up the probabilities of the actions seen
- If $r(\tau)$ is low, push down the probabilities of the actions seen

Might seem simplistic to say that if a trajectory is good then all its actions were good. **But in expectation, it averages out!**

Intuition

Gradient estimator: $\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$

Interpretation:

- If $r(\tau)$ is high, push up the probabilities of the actions seen
- If $r(\tau)$ is low, push down the probabilities of the actions seen

Might seem simplistic to say that if a trajectory is good then all its actions were good. **But in expectation, it averages out!**

However, this also suffers from high variance because credit assignment is really hard. Can we help the estimator?

Variance reduction

Gradient estimator: $\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$

Variance reduction

Gradient estimator: $\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$

First idea: Push up probabilities of an action seen, only by the cumulative future reward from that state

$$\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} r_{t'} \right) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

Variance reduction

Gradient estimator: $\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$

First idea: Push up probabilities of an action seen, only by the cumulative future reward from that state

$$\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} r_{t'} \right) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

Second idea: Use discount factor γ to ignore delayed effects

$$\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t' - t} r_{t'} \right) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

Variance reduction: Baseline

Problem: The raw value of a trajectory isn't necessarily meaningful. For example, if rewards are all positive, you keep pushing up probabilities of actions.

What is important then? Whether a reward is better or worse than what you expect to get

Idea: Introduce a baseline function dependent on the state.
Concretely, estimator is now:

$$\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r_{t'} - b(s_t) \right) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

How to choose the baseline?

$$\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r_{t'} - b(s_t) \right) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

A simple baseline: constant moving average of rewards experienced so far from all trajectories

How to choose the baseline?

$$\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r_{t'} - b(s_t) \right) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

A simple baseline: constant moving average of rewards experienced so far from all trajectories

Variance reduction techniques seen so far are typically used in “Vanilla REINFORCE”

How to choose the baseline?

A better baseline: Want to push up the probability of an action from a state, if this action was better than the **expected value of what we should get from that state**.

Q: What does this remind you of?

How to choose the baseline?

A better baseline: Want to push up the probability of an action from a state, if this action was better than the **expected value of what we should get from that state**.

Q: What does this remind you of?

A: Q-function and value function!

How to choose the baseline?

A better baseline: Want to push up the probability of an action from a state, if this action was better than the **expected value of what we should get from that state**.

Q: What does this remind you of?

A: Q-function and value function!

Intuitively, we are happy with an action a_t in a state s_t if $Q^\pi(s_t, a_t) - V^\pi(s_t)$ is large. On the contrary, we are unhappy with an action if it's small.

How to choose the baseline?

A better baseline: Want to push up the probability of an action from a state, if this action was better than the **expected value of what we should get from that state**.

Q: What does this remind you of?

A: Q-function and value function!

Intuitively, we are happy with an action a_t in a state s_t if $Q^\pi(s_t, a_t) - V^\pi(s_t)$ is large. On the contrary, we are unhappy with an action if it's small.

Using this, we get the estimator: $\nabla_\theta J(\theta) \approx \sum_{t \geq 0} (Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)) \nabla_\theta \log \pi_\theta(a_t | s_t)$

Actor-Critic Algorithm

Problem: we don't know Q and V. Can we learn them?

Yes, using Q-learning! We can combine Policy Gradients and Q-learning by training both an **actor** (the policy) and a **critic** (the Q-function).

- The actor decides which action to take, and the critic tells the actor how good its action was and how it should adjust
- Also alleviates the task of the critic as it only has to learn the values of (state, action) pairs generated by the policy
- Can also incorporate Q-learning tricks e.g. experience replay
- **Remark:** we can define by the **advantage function** how much an action was better than expected

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$$

Actor-Critic Algorithm

Initialize policy parameters θ , critic parameters ϕ

For iteration=1, 2 ... **do**

 Sample m trajectories under the current policy

$\Delta\theta \leftarrow 0$

For i=1, ..., m **do**

For t=1, ..., T **do**

$$A_t = \sum_{t' \geq t} \gamma^{t'-t} r_t^i - V_\phi(s_t^i)$$

$$\Delta\theta \leftarrow \Delta\theta + A_t \nabla_\theta \log(a_t^i | s_t^i)$$

$$\Delta\phi \leftarrow \sum_t \sum_i \nabla_\phi ||A_t^i||^2$$

$$\theta \leftarrow \alpha \Delta\theta$$

$$\phi \leftarrow \beta \Delta\phi$$

End for

REINFORCE in action: Recurrent Attention Model (RAM)

Objective: Image Classification

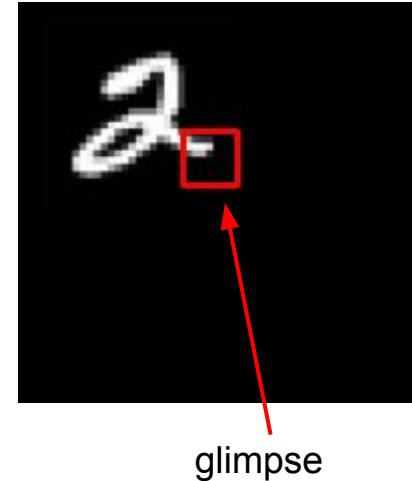
Take a sequence of “glimpses” selectively focusing on regions of the image, to predict class

- Inspiration from human perception and eye movements
- Saves computational resources => scalability
- Able to ignore clutter / irrelevant parts of image

State: Glimpses seen so far

Action: (x,y) coordinates (center of glimpse) of where to look next in image

Reward: 1 at the final timestep if image correctly classified, 0 otherwise



[Mnih et al. 2014]

REINFORCE in action: Recurrent Attention Model (RAM)

Objective: Image Classification

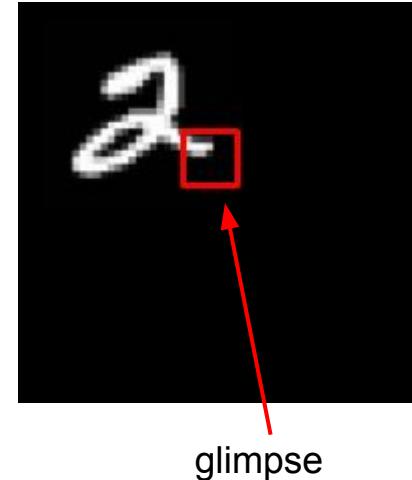
Take a sequence of “glimpses” selectively focusing on regions of the image, to predict class

- Inspiration from human perception and eye movements
- Saves computational resources => scalability
- Able to ignore clutter / irrelevant parts of image

State: Glimpses seen so far

Action: (x,y) coordinates (center of glimpse) of where to look next in image

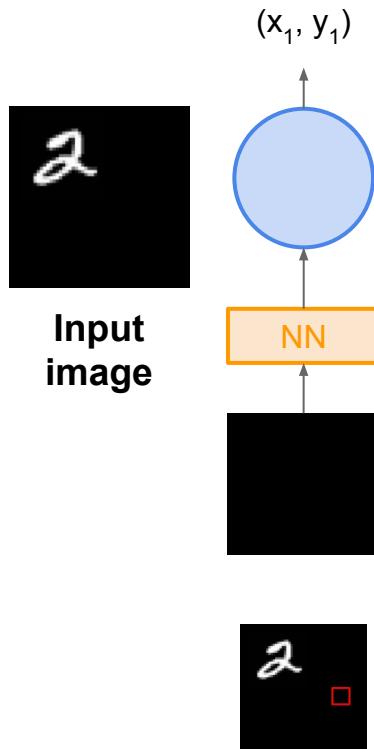
Reward: 1 at the final timestep if image correctly classified, 0 otherwise



Glimpsing is a non-differentiable operation => learn policy for how to take glimpse actions using REINFORCE
Given state of glimpses seen so far, use RNN to model the state and output next action

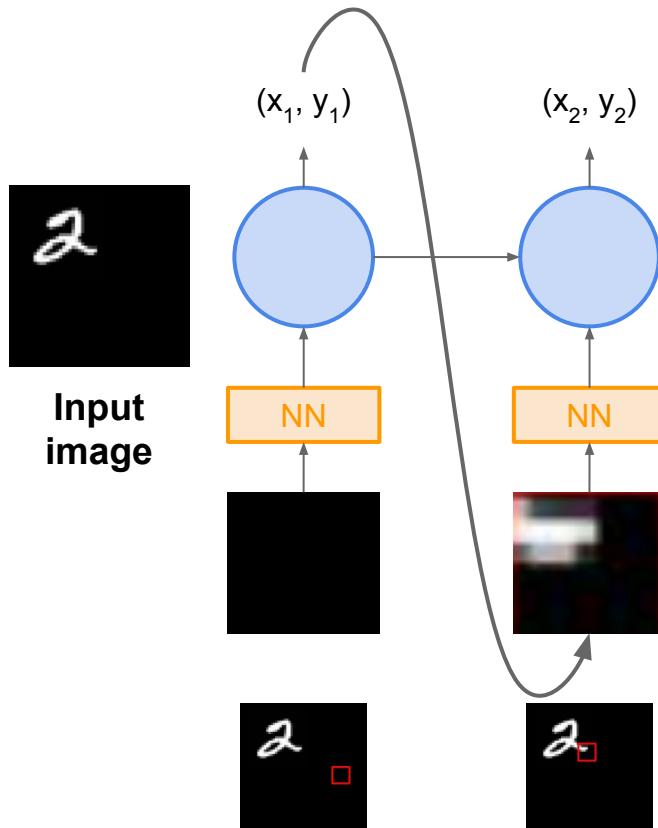
[Mnih et al. 2014]

REINFORCE in action: Recurrent Attention Model (RAM)



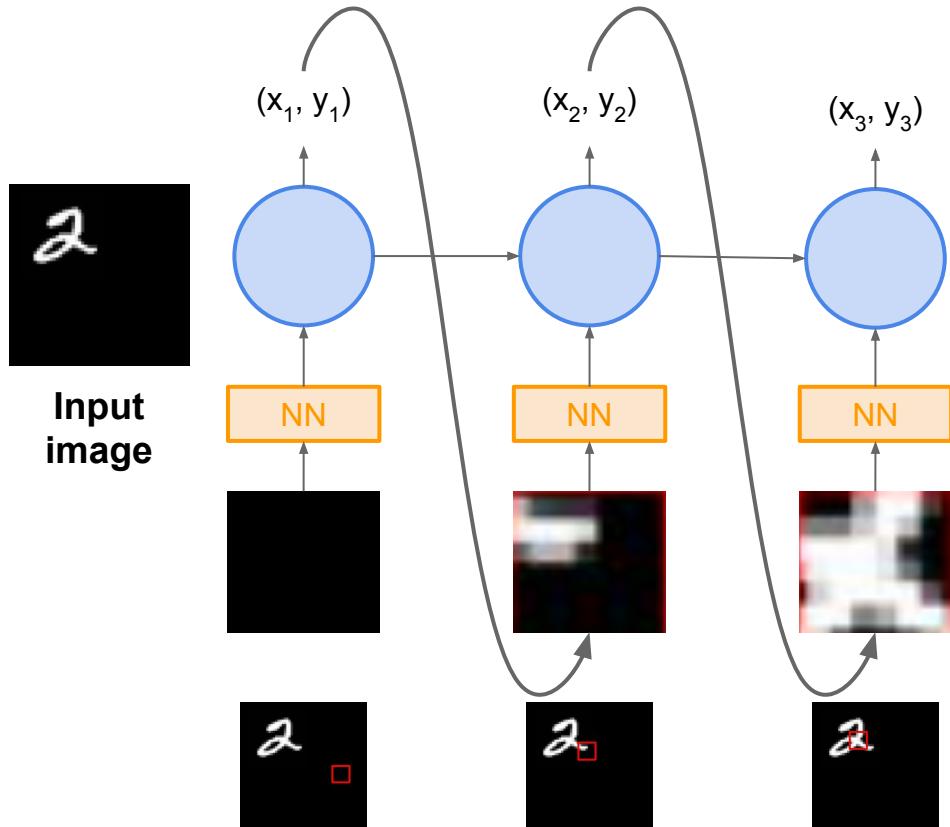
[Mnih et al. 2014]

REINFORCE in action: Recurrent Attention Model (RAM)



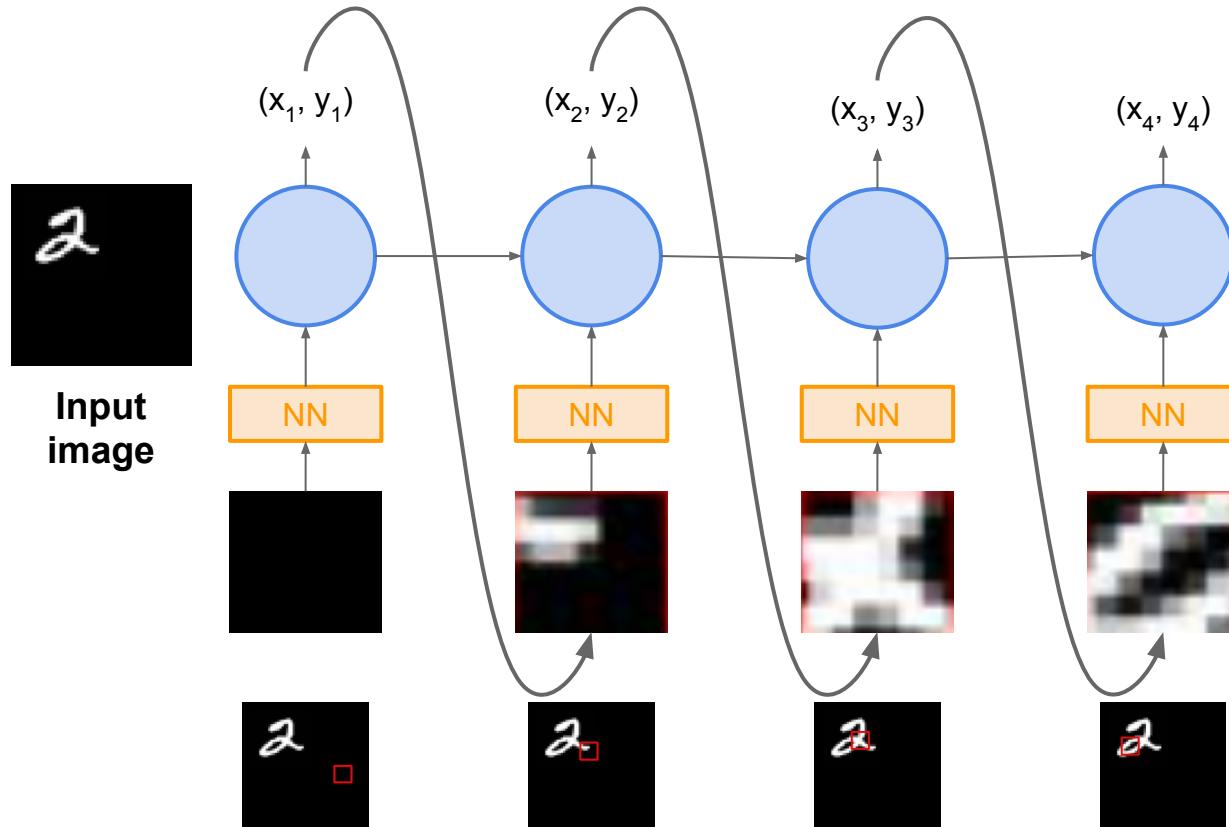
[Mnih et al. 2014]

REINFORCE in action: Recurrent Attention Model (RAM)



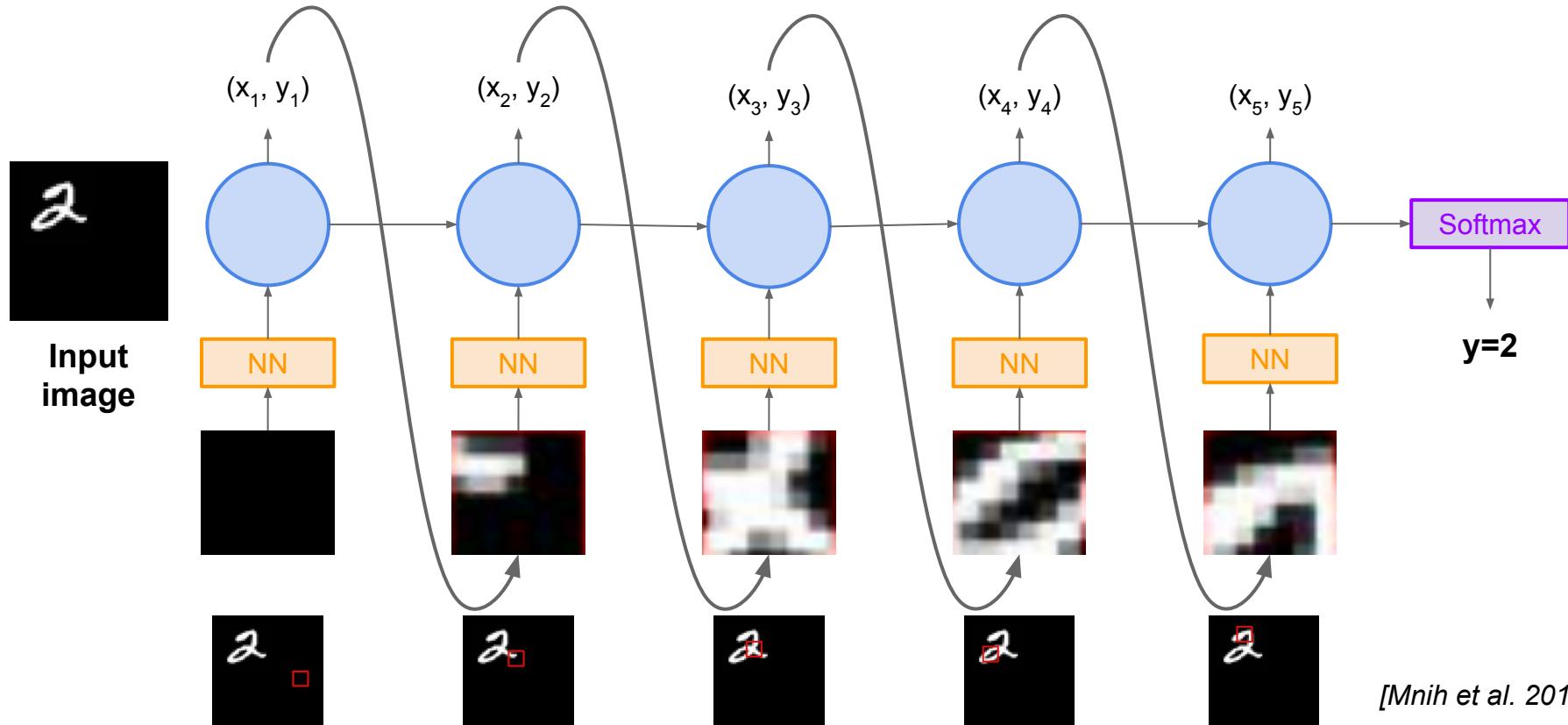
[Mnih et al. 2014]

REINFORCE in action: Recurrent Attention Model (RAM)

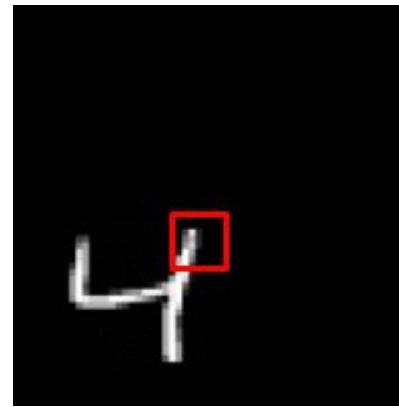
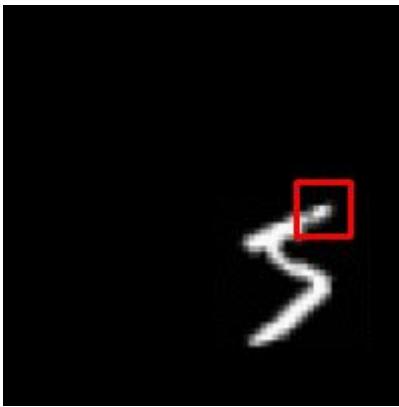
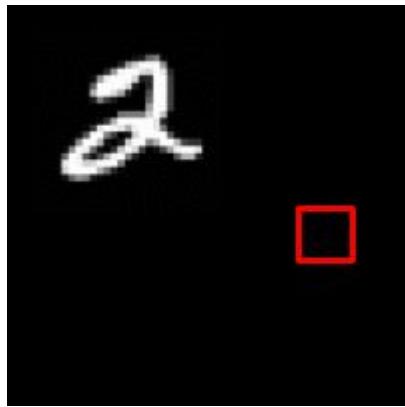


[Mnih et al. 2014]

REINFORCE in action: Recurrent Attention Model (RAM)



REINFORCE in action: Recurrent Attention Model (RAM)



Has also been used in many other tasks including fine-grained image recognition, image captioning, and visual question-answering!

Figures copyright Daniel Levy, 2017. Reproduced with permission.

[Mnih et al. 2014]

More policy gradients: AlphaGo

AlphaGo [Nature 2016]:

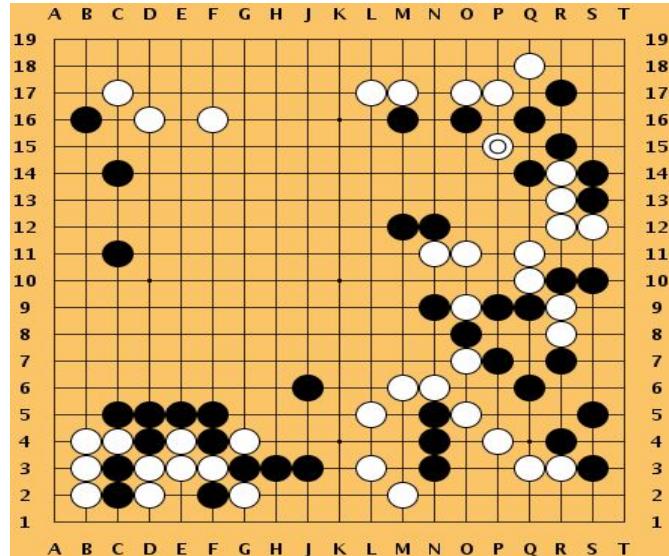
- Required many engineering tricks
- Bootstrapped from human play
- Beat 18-time world champion Lee Sedol

AlphaGo Zero [Nature 2017]:

- Simplified and elegant version of AlphaGo
- No longer bootstrapped from human play
- Beat (at the time) #1 world ranked Ke Jie

Alpha Zero: Dec. 2017

- Generalized to beat world champion programs on chess and shogi as well



This image is CC0 public domain

Summary

- **Policy gradients:** very general but suffer from high variance so requires a lot of samples. **Challenge:** sample-efficiency
- **Q-learning:** does not always work but when it works, usually more sample-efficient. **Challenge:** exploration
- Guarantees:
 - **Policy Gradients:** Converges to a local minima of $J(\theta)$, often good enough!
 - **Q-learning:** Zero guarantees since you are approximating Bellman equation with a complicated function approximator

Next Time

Guest Lecture: Andrej Karpathy