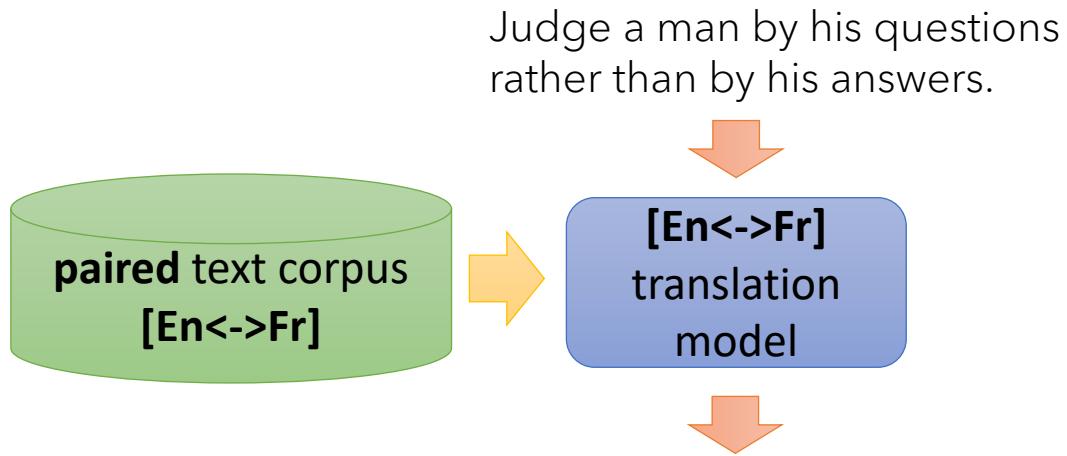


The world has over **6000** languages

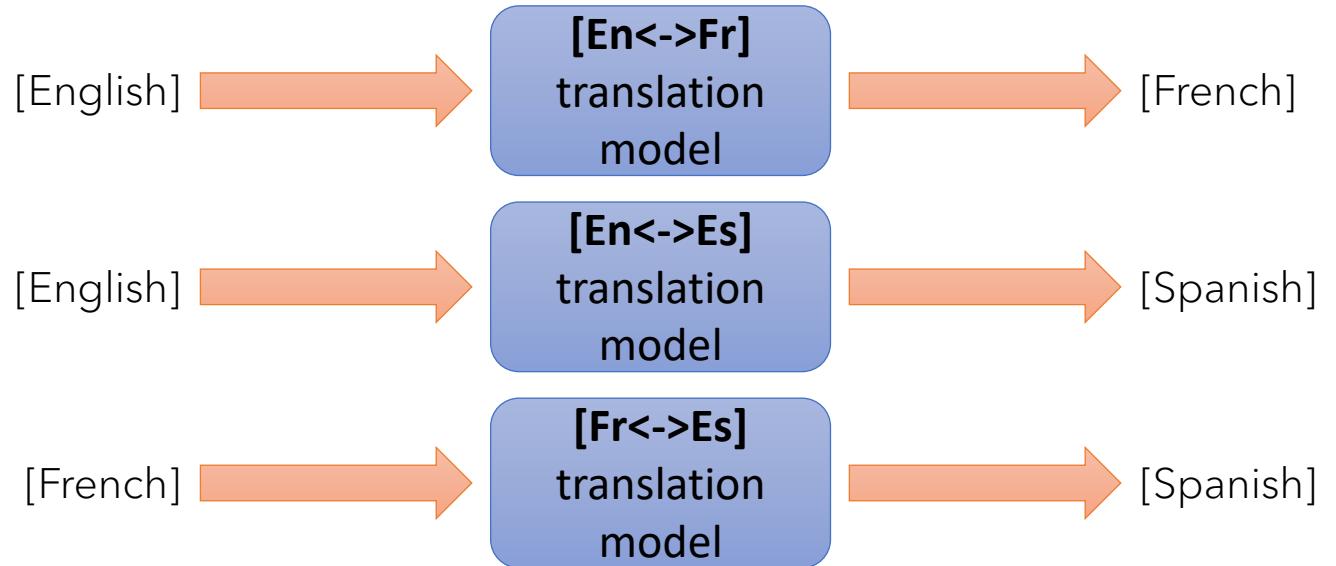
Automated translation systems require **paired** data

[En] I think, therefore I am. <-> [Fr] Je pense, donc je suis.

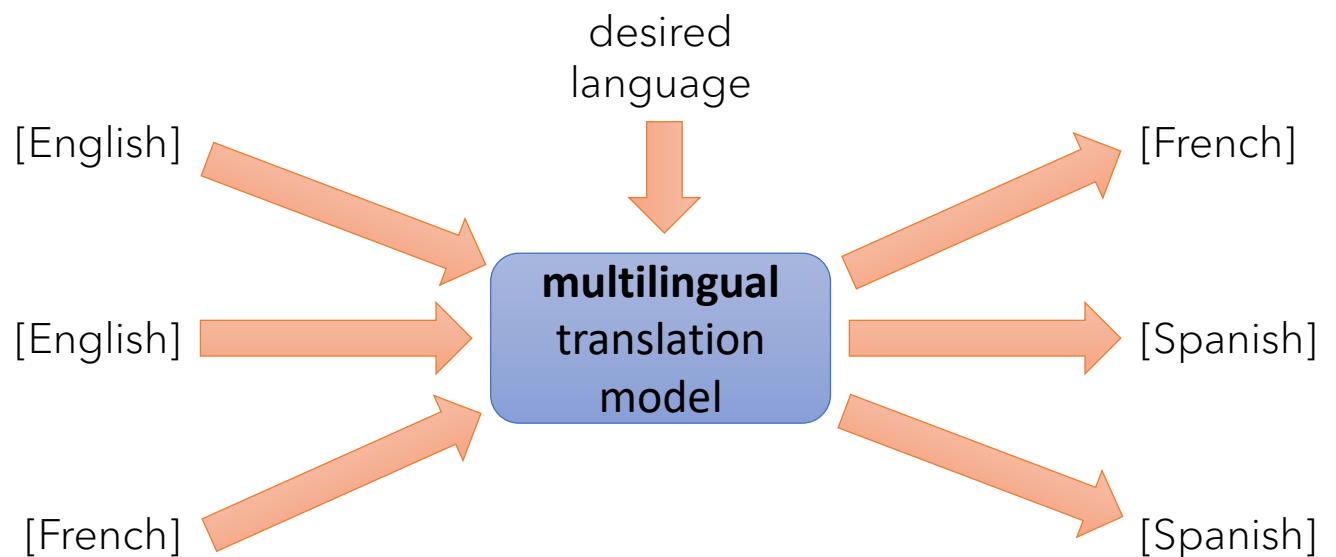


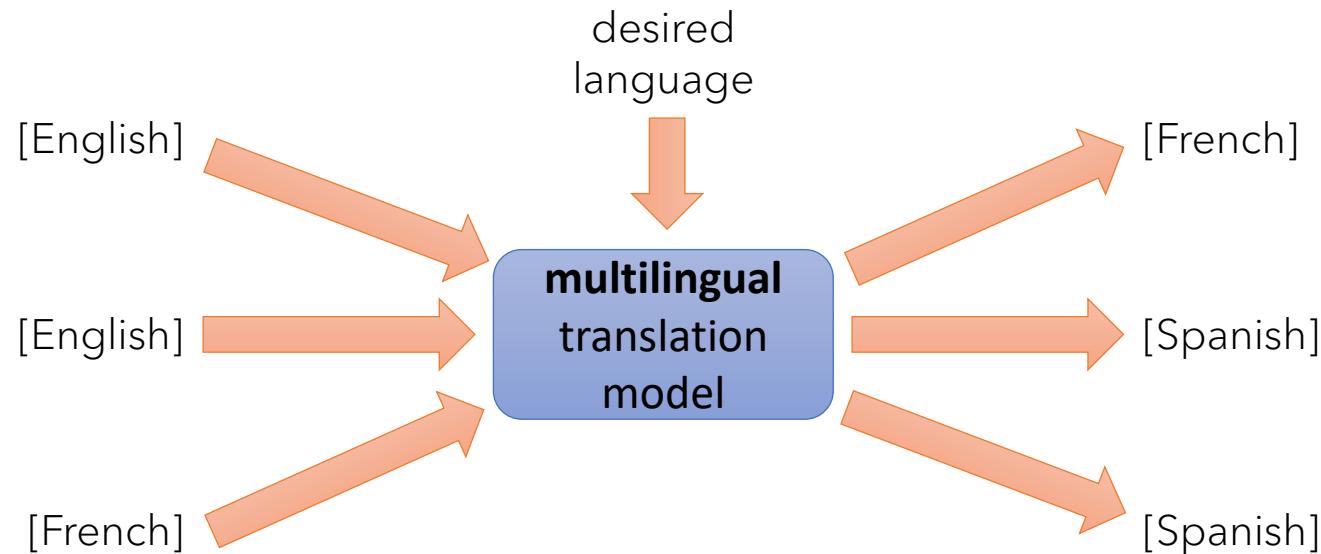
How many **paired** sentences are there for translating **Maltese to Tibetan?**

“Standard” machine translation:



“Multilingual” machine translation:





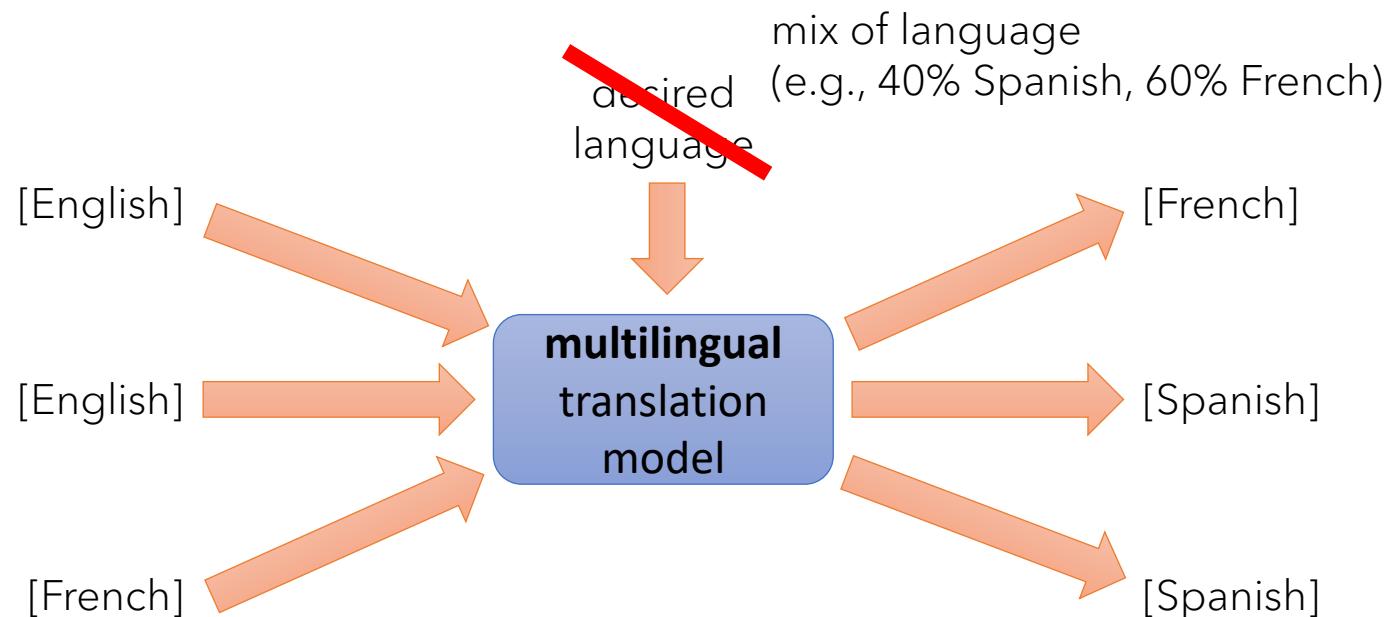
What did they find?

Improved efficiency:

Translating into and out of rare languages works **better** if the model is also trained on more common languages

Zero-shot machine translation:

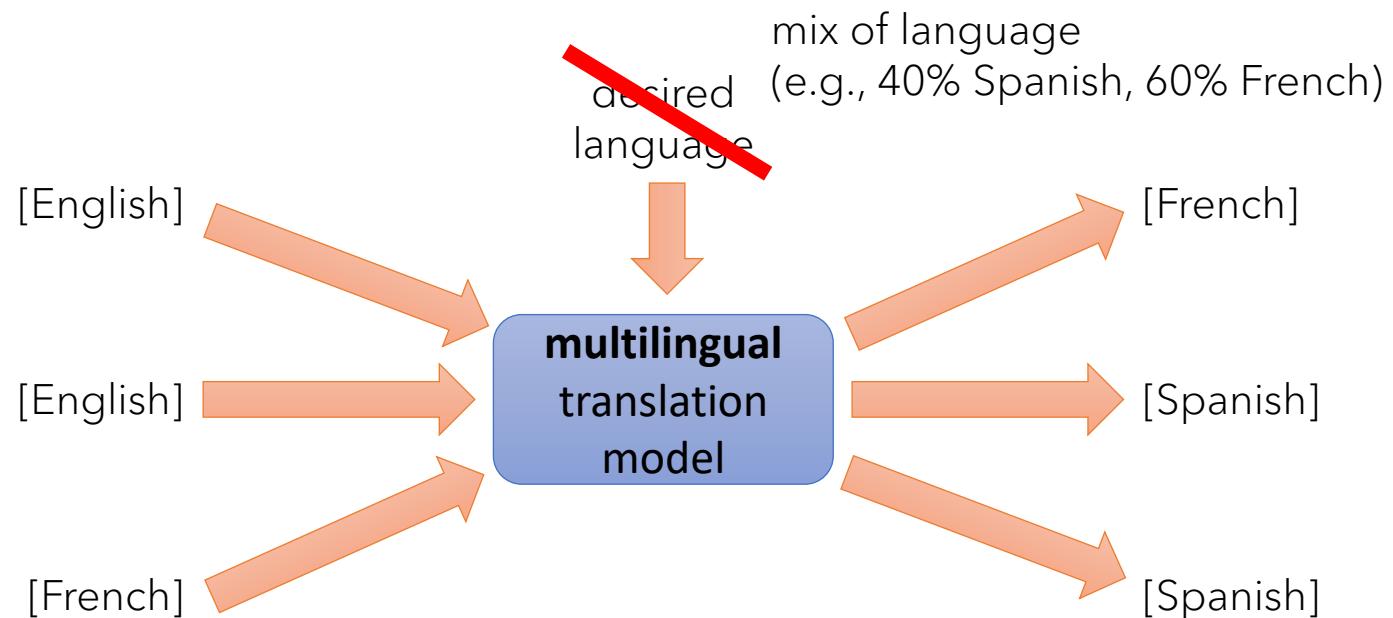
E.g., train on **English -> French**, **French -> English**, and **English -> Spanish**, and be able to translate **French -> Spanish**



Translating English to mix of Spanish and Portuguese:

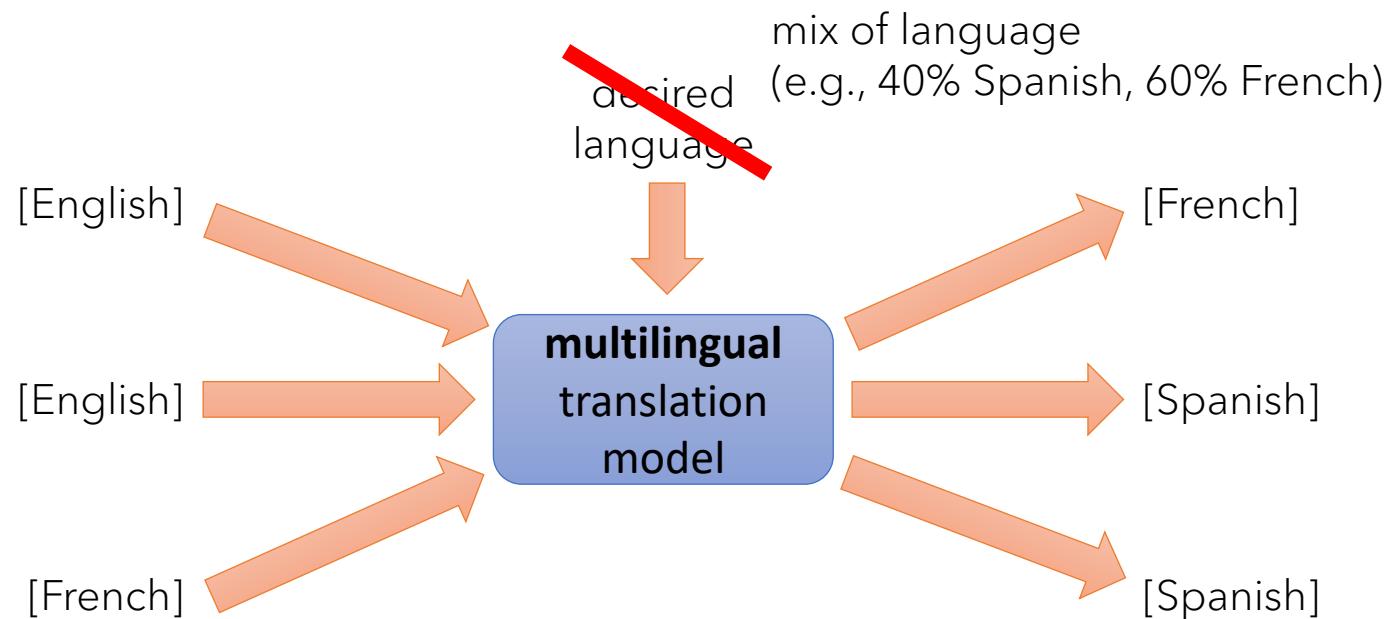
Spanish/Portuguese:	Here the other guinea-pig cheered, and was suppressed.
$w_{pt} = 0.00$	Aquí el otro conejillo de indias animó, y fue suprimido.
$w_{pt} = 0.30$	Aquí el otro conejillo de indias animó, y fue suprimido.
$w_{pt} = 0.40$	Aquí, o outro porquinho-da-índia alegrou, e foi suprimido.
$w_{pt} = 0.42$	Aqui o outro porquinho-da-índia alegrou, e foi suprimido.
$w_{pt} = 0.70$	Aqui o outro porquinho-da-índia alegrou, e foi suprimido.
$w_{pt} = 0.80$	Aqui a outra cobaia animou, e foi suprimida.
$w_{pt} = 1.00$	Aqui a outra cobaia animou, e foi suprimida.

“Portuguese” weight
(Spanish weight = 1-w) →



Translating English to mix of Japanese and Korean:

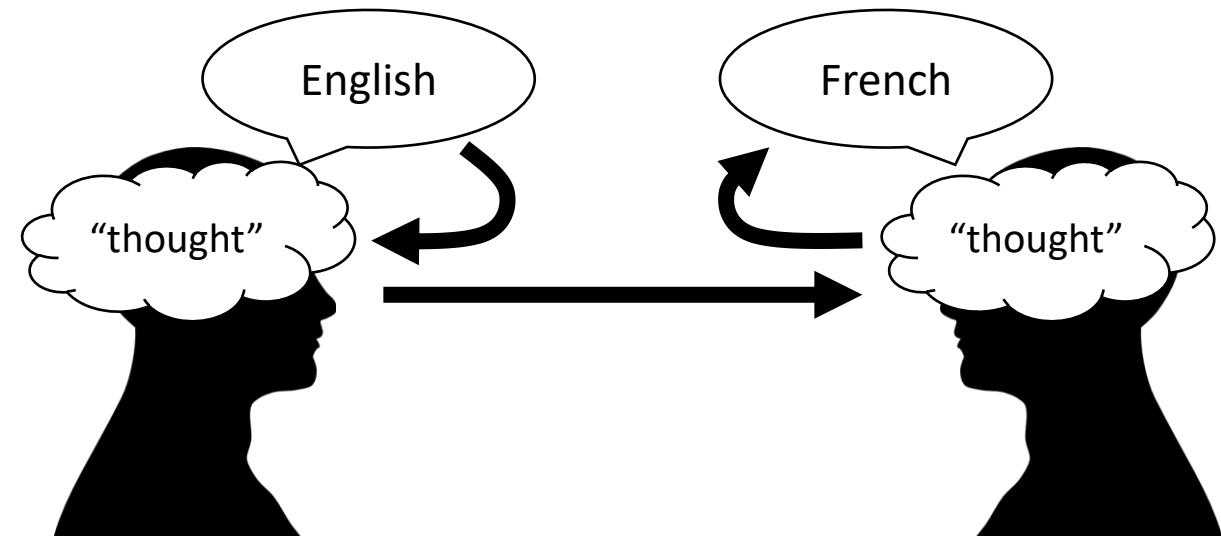
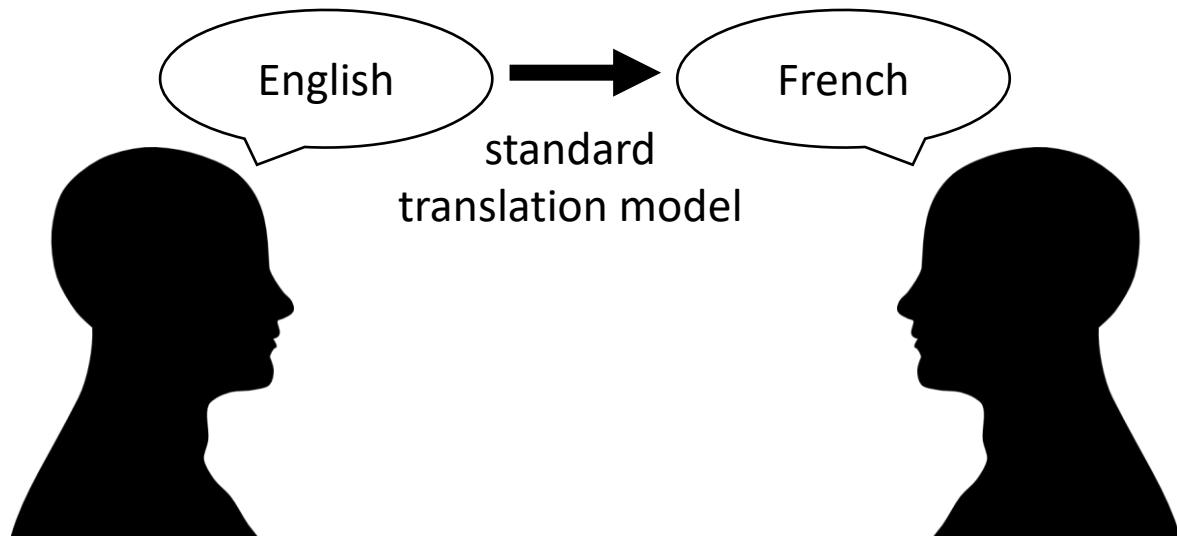
Japanese/Korean:	I must be getting somewhere near the centre of the earth.
$w_{ko} = 0.00$	私は地球の中心の近くにどこかに行っているに違いない。
$w_{ko} = 0.40$	私は地球の中心近くのどこかに着いているに違いない。
$w_{ko} = 0.56$	私は地球の中心の近くのどこかになっているに違いない。
$w_{ko} = 0.58$	私は地図の中心のガッキーに着いています。
$w_{ko} = 0.60$	私は地図のセンターガッキーに着いています。
$w_{ko} = 0.70$	私は地図の中心に着いています。
$w_{ko} = 0.90$	私は地図の中心に着いています。
$w_{ko} = 1.00$	私は地図の中心に着いています。



Translating English to mix of Russian and Belarusian:

Russian/Belarusian:	I wonder what they'll do next!	
$w_{be} = 0.00$	Интересно, что они сделают дальше!	
$w_{be} = 0.20$	Интересно, что они сделают дальше!	
$w_{be} = 0.30$	<u>Цікаво</u> , что они будут делать дальше!	
$w_{be} = 0.44$	<u>Цікаво</u> , що вони будуть робити далі!	
$w_{be} = 0.46$	<u>Цікаво</u> , що вони будуть робити далі!	
$w_{be} = 0.48$	<u>Цікаво</u> , што яны зробяць далей!	
$w_{be} = 0.50$	Цікава, што яны будуць рабіць далей!	
$w_{be} = 1.00$	Цікава, што яны будуць рабіць далей!	
		Neither Russian nor Belarusian!

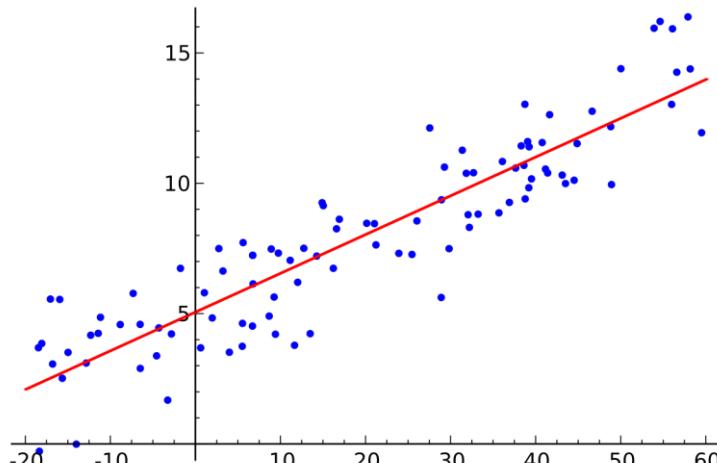
What's going on?



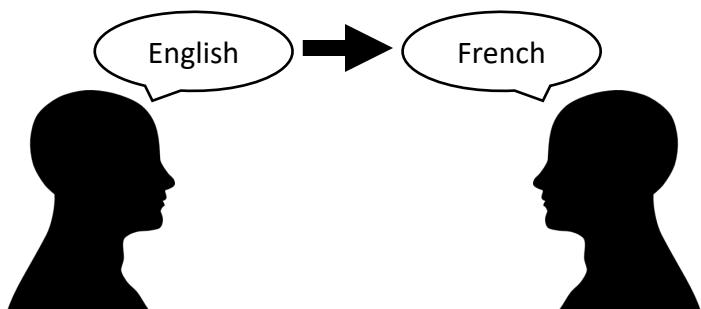
the “thought” is a **representation**!

Representation learning

“Classic” view of machine learning:



predict y from x



but what is x ?

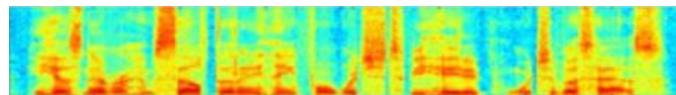
Il est encore plus facile de juger de l'esprit d'un homme par ses questions que par ses réponses.



Handling such complex inputs requires **representations**



The power of deep learning lies in its ability to **learn** such **representations** automatically from data



Deep Learning

Designing, Visualizing and Understanding Deep Neural Networks

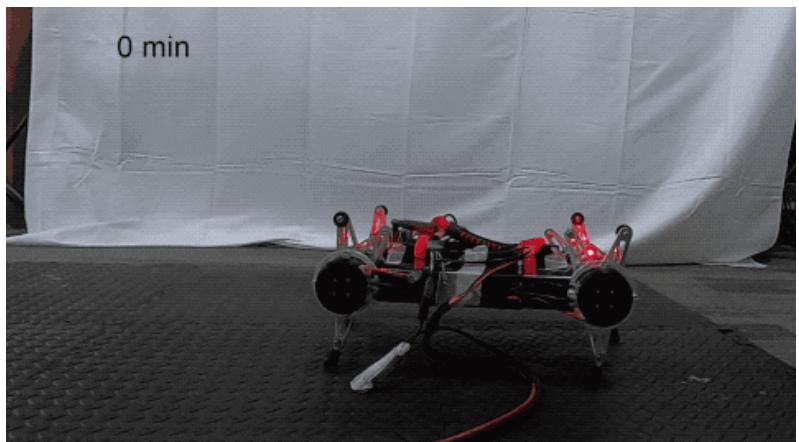
CS W182/282A

Instructor: Sergey Levine
UC Berkeley



Course overview

- **Broad overview of deep learning topics**
 - Neural network architectures
 - Optimization algorithms
 - Applications: vision, NLP
 - Reinforcement learning
 - Advanced topics
- **Four homework programming assignments**
 - Neural network basics
 - Convolutional and recurrent networks
 - Natural language processing
 - Reinforcement learning
- **Two midterm exams**
 - Format TBD, but most likely will be a take-home exam
- **Final project (group project, 2-3 people)**
 - Most important part of the course
 - CS182: choose vision, NLP, or reinforcement learning
 - CS282: self-directed and open-ended project



Course policies

Grading:

- 30% midterms
- 40% programming homeworks
- 30% final project

Late policy:

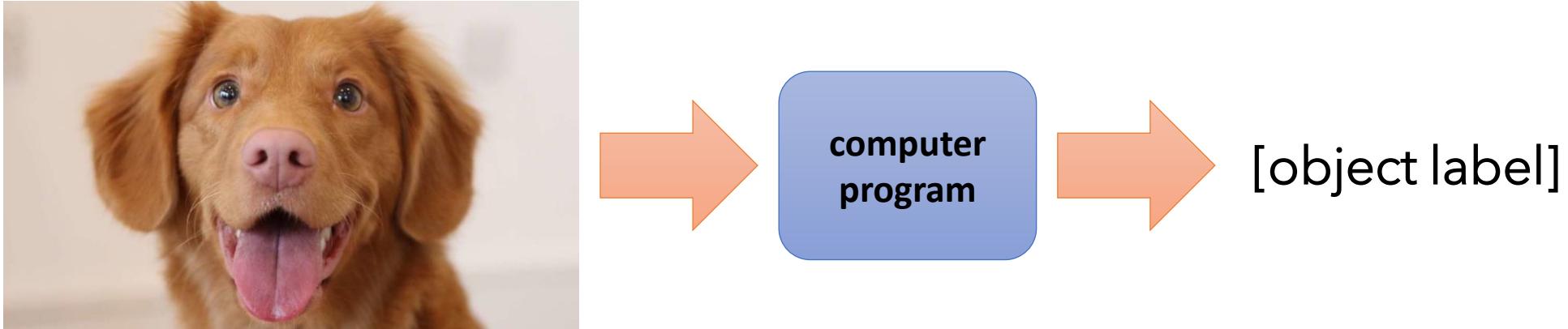
- 5 slip days
- strict late policy, no slack beyond slip days
- no slip days for final project (due to grades deadline)

Prerequisites:

- Excellent knowledge of calculus linear algebra
 - especially:** multi-variate derivatives, matrix operations, solving linear systems
- CS70 or STAT134, excellent knowledge of probability theory (including continuous random variables)
- CS189, or a very strong statistics background
- CS61B or equivalent, able to program in Python

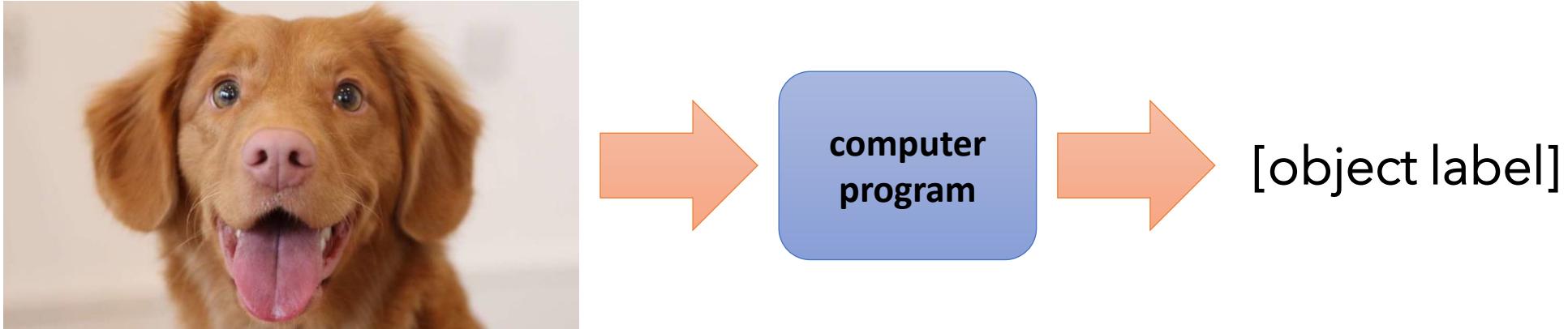
What is machine learning?
What is deep learning?

What is machine learning?



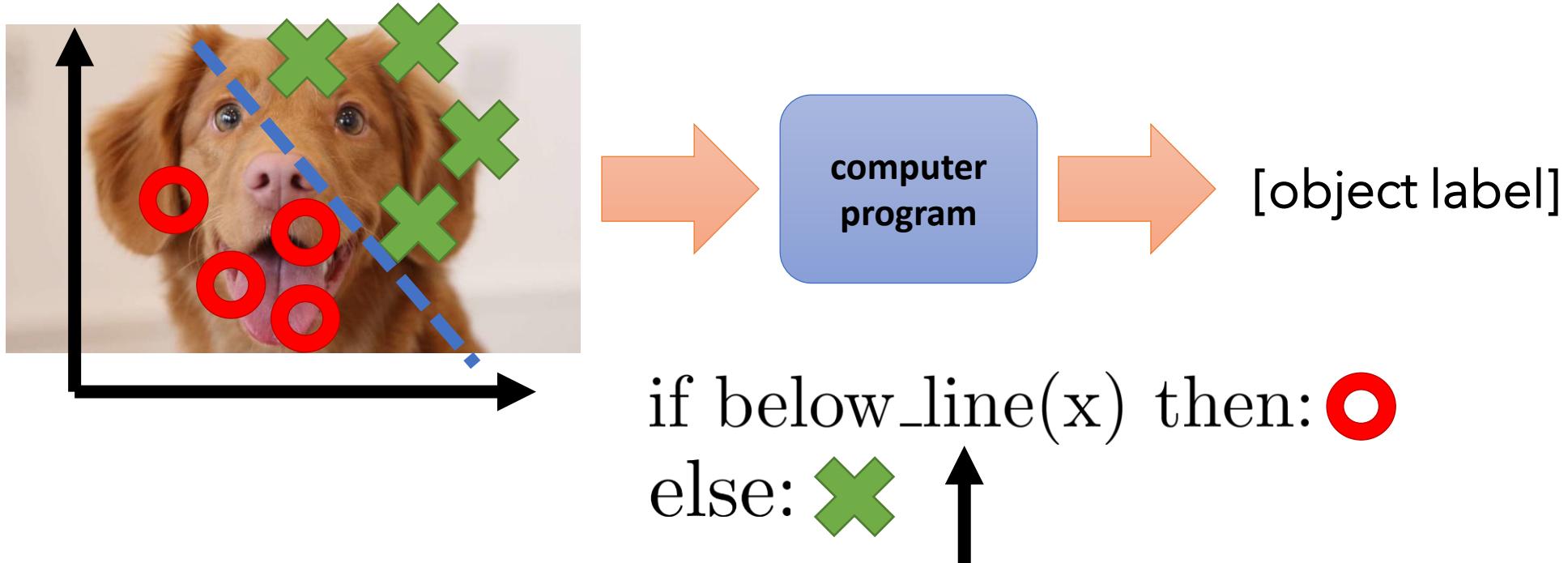
- How do we implement this program?
- A function is a set of **rules** for transforming **inputs** into **outputs**
- Sometimes we can define the rules by hand – this is called programming
- What if we don't know the rules?
- What if the rules are too complex? Too many exceptions & special cases?

What is machine learning?



- Instead of defining the **input -> output** relationship by hand, define a program that acquires this relationship from **data**
- **Key idea:** if the rules that describe how **inputs** map to **outputs** are complex and full of special cases & exceptions, it is easier to provide **data or examples** than to implement those rules
- **Question:** Does this also apply to human and animal learning?

What are we learning?



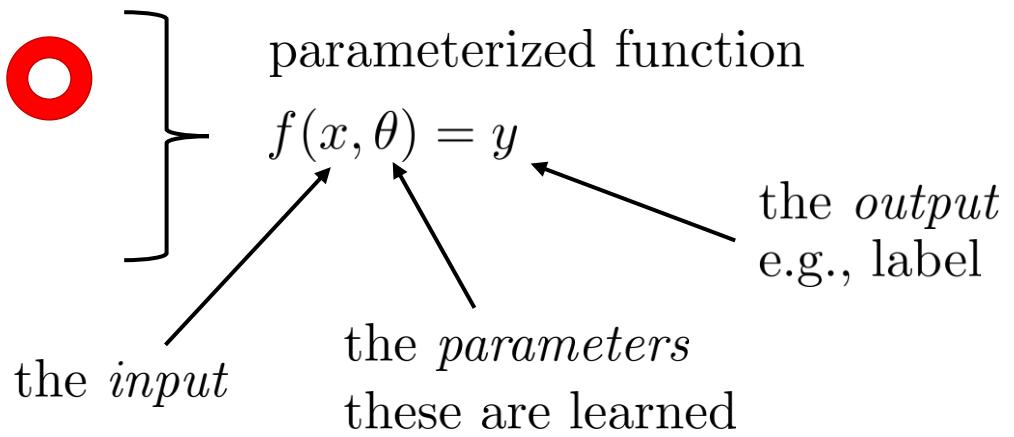
this describes a line $\longrightarrow x_1\theta_1 + x_2\theta_2 + \theta_3 \leq 0$

$$\text{learn } (\theta_1, \theta_2, \theta_3) = \vec{\theta} \quad \vec{x}^T \vec{\theta} \leq 0$$

so that our *parameterized* program (function) gives the right answer!

In general...

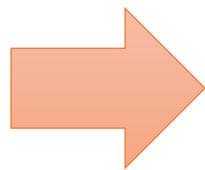
```
if below_line(x) then: ○  
else: ✗
```



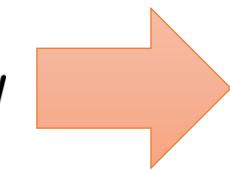
can also write as: $f_\theta(x) = y$

crucially, $f_\theta(x)$ can be almost any expression of x and θ !

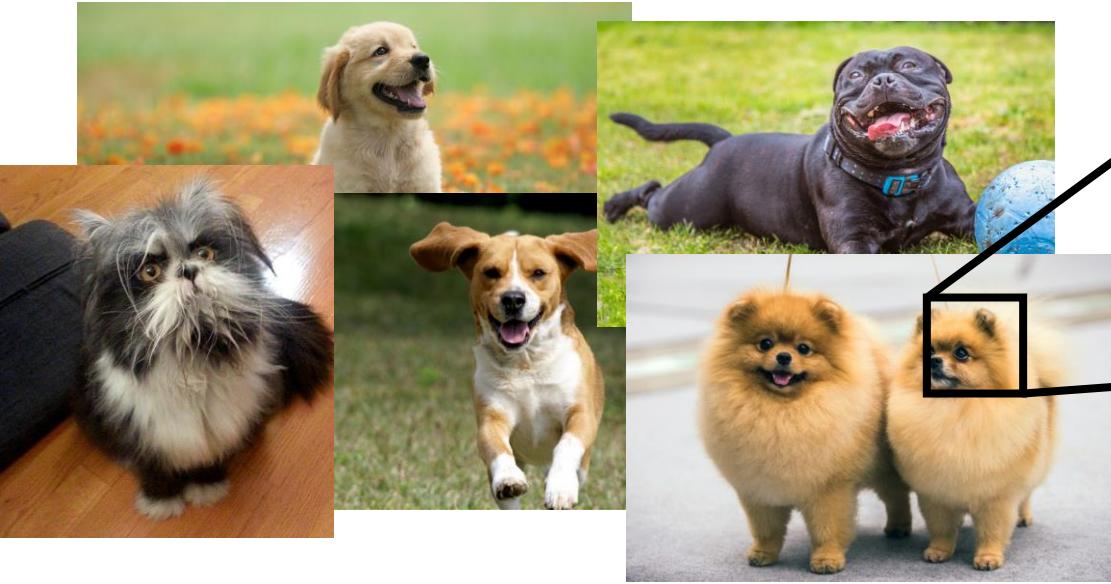
But what parameterization do we use?



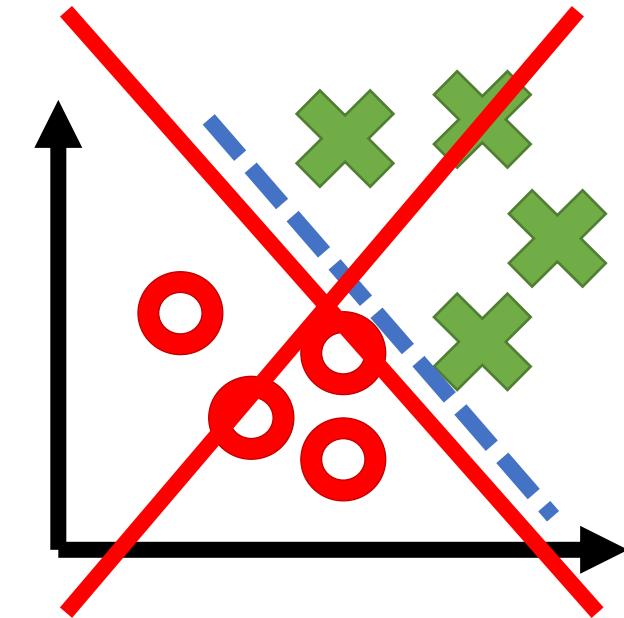
$$f_{\theta}(x) = y$$



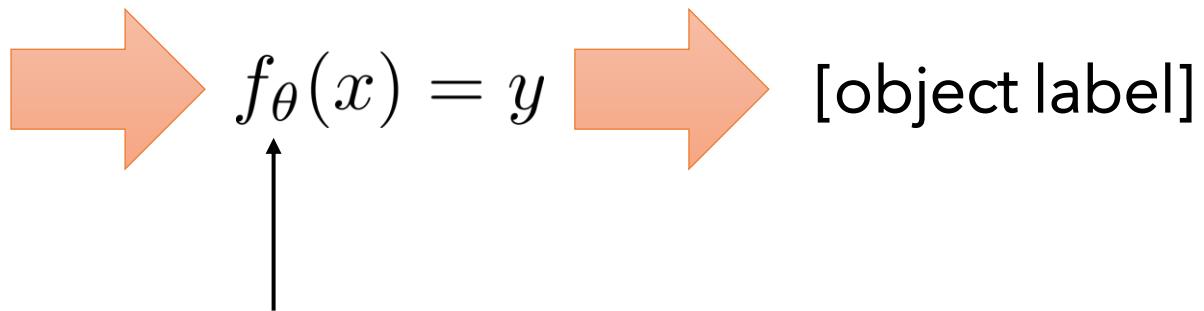
[object label]



0.2	0.1	0.3	0.3
0.2	0.5	0.3	0.3
0.3	0.1	0.2	0.2
0.3	0.1	0.2	0.2

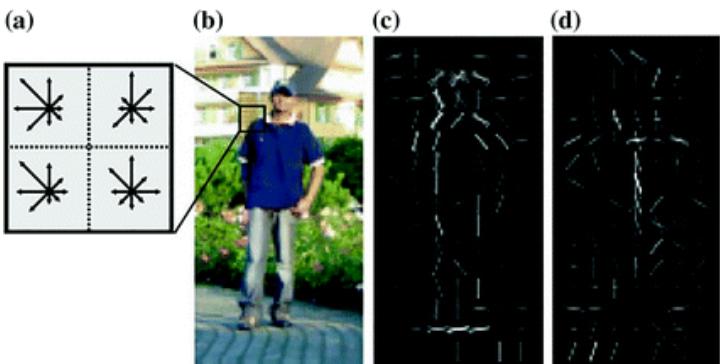


“Shallow” learning



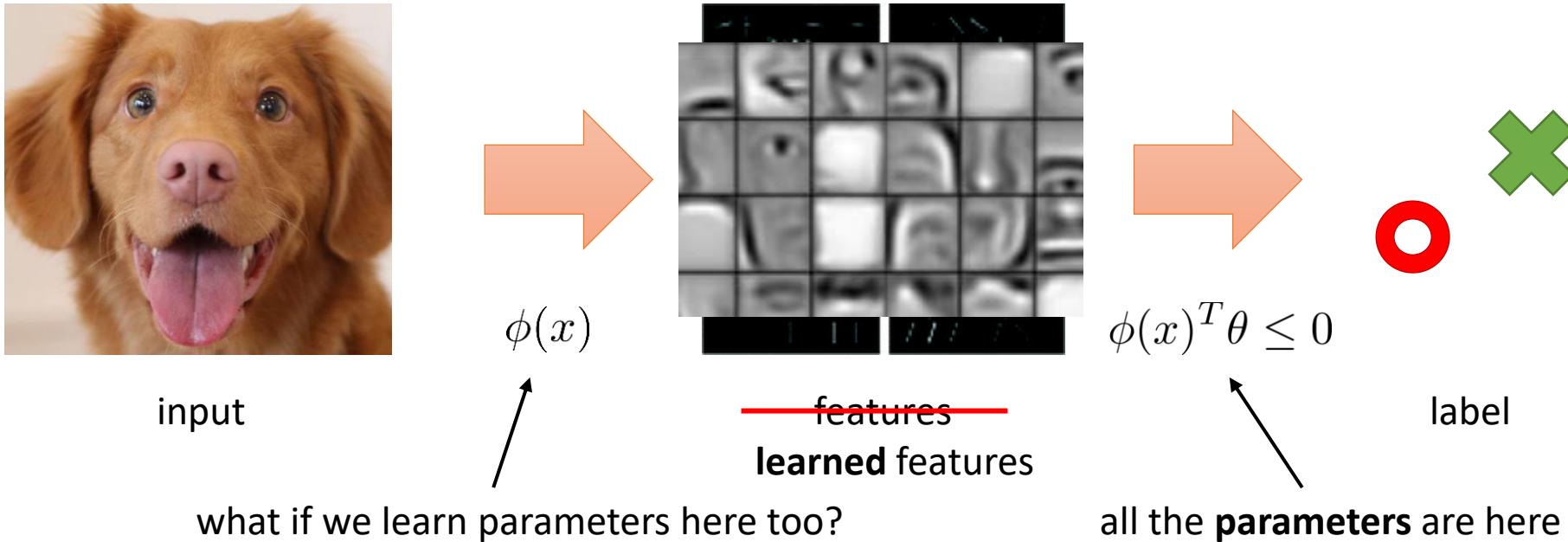
fixed function for extracting *features* from x

$$\phi(x)^T \theta \leq 0$$



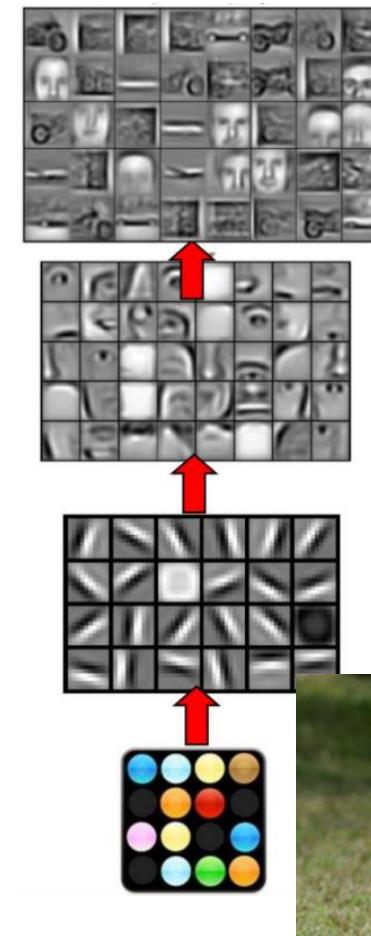
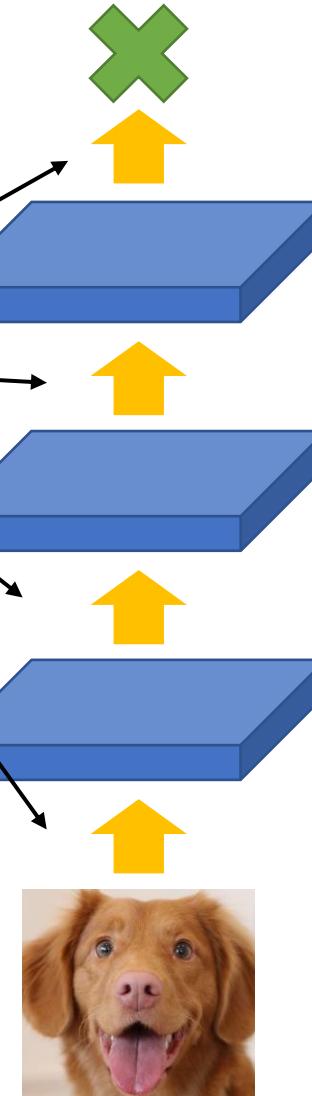
- Kind of a “compromise” solution: don’t hand-program the rules, but hand-program the features
- Learning on top of the features can be simple (just like the 2D example from before!)
- Coming up with good features is very hard!

From shallow learning to deep learning



Multiple layers of representations?

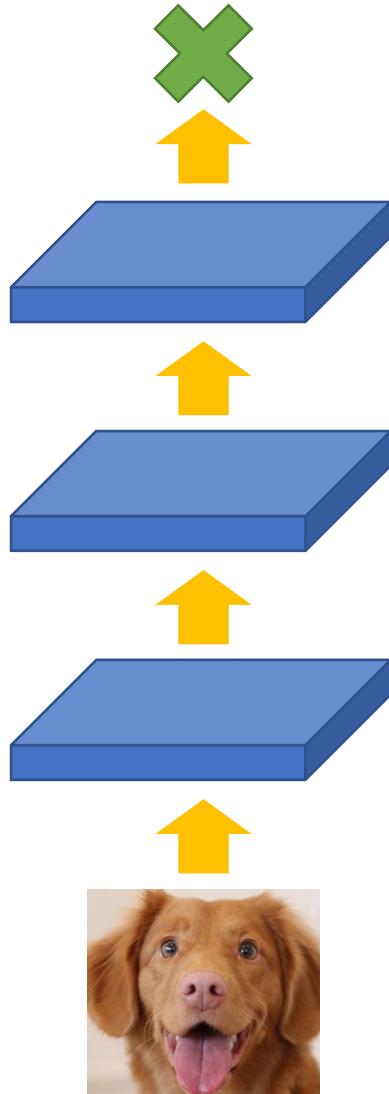
each arrow represents a simple **parameterized** transformation (function) of the preceding layer



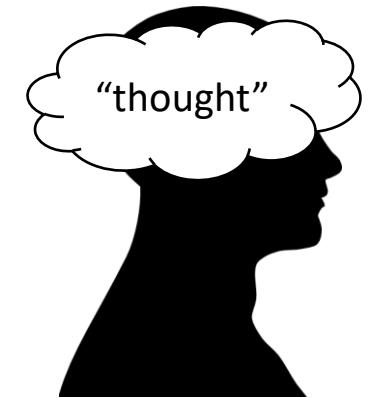
Higher level representations are:

- More abstract
- More invariant to nuisances
- Easier for predicting label

So, what is deep learning?



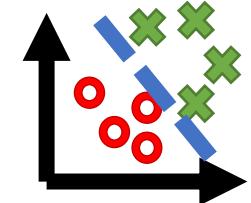
- Machine learning with **multiple layers of learned representations**
- The **function** that represents the transformation from input to internal representation to output is usually a deep neural network
 - This is a bit circular, because almost all **multi-layer parametric** functions with **learned parameters** can be called neural networks (more on this later)
- The parameters for every layer are usually (**but not always!**) trained with respect to the overall task objective (**e.g., accuracy**)
 - This is sometimes referred to as **end-to-end** learning



What makes deep learning work?

1950

1950: Turing describes how learning could be a path to machine intelligence



1960

1957: Rosenblatt's **perceptron** proposed as a practical learning method

1970

1969: Minsky & Papert publish book describing fundamental limitations of neural networks
most (but not all) mainstream research focuses on "shallow" learning

1980

1986: Backpropagation as a practical method for training deep nets
1989: LeNet (neural network for handwriting recognition)

1990

Huge wave of interest in ML community in probabilistic methods, convex optimization, but mostly in shallow models

2000

~2006: deep neural networks start gaining more attention

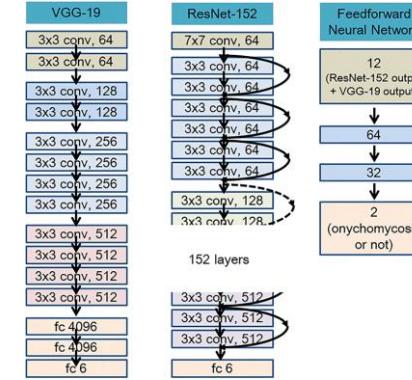
2010

2012: Krizhevsky's AlexNet paper beats all other methods on ImageNet

*what the heck
happened here?*

What makes deep learning work?

1) Big models with many layers



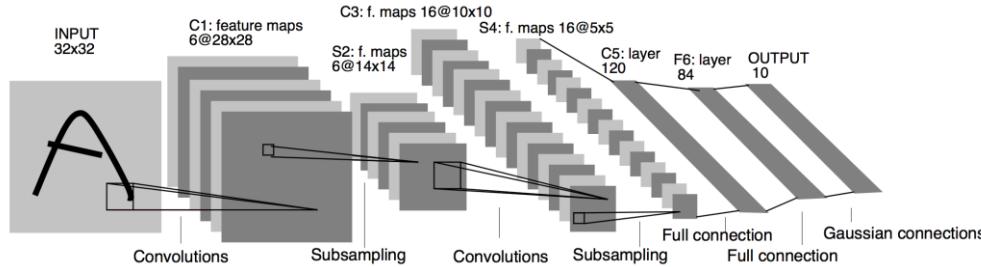
2) Large datasets with many examples



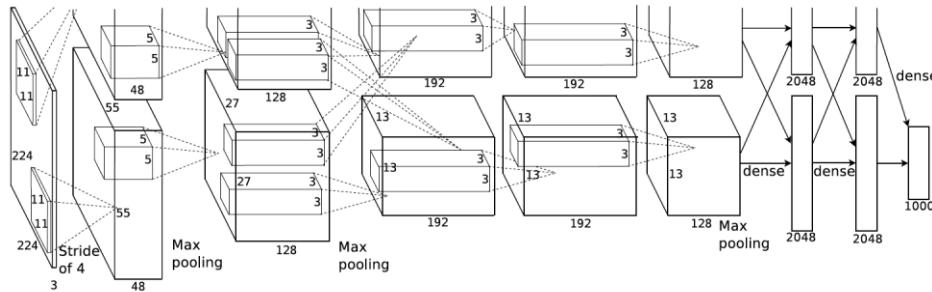
3) Enough **compute** to handle all this



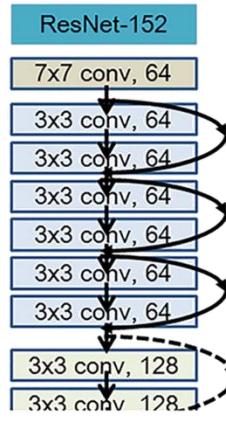
Model scale: is more layers better?



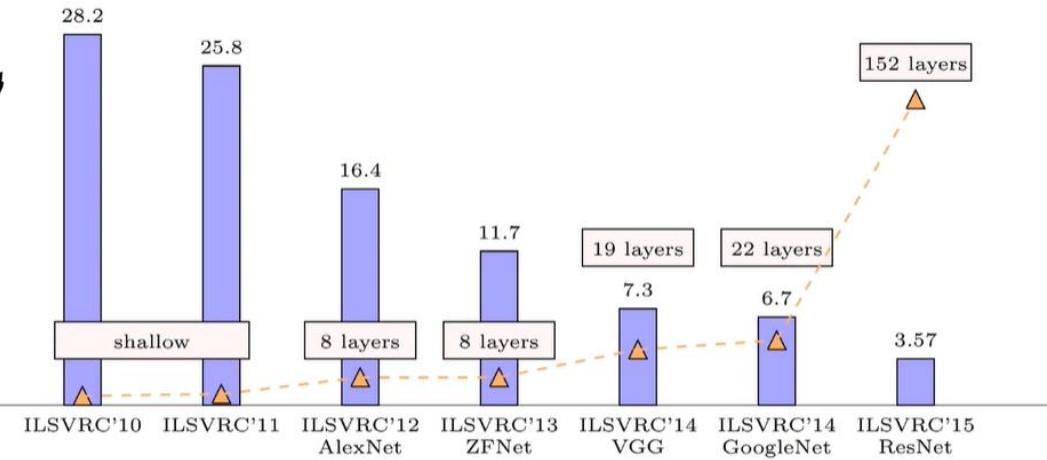
LeNet, 7 layers (1989)



Krizhevsky's model (AlexNet) for ImageNet, 8 layers (2012)

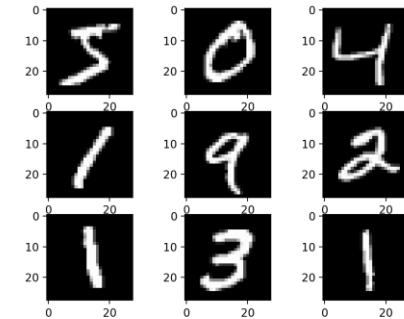


ResNet-152: 152 layers (2015)



How big are the datasets?

MNIST (handwritten characters), 1990s - today: 60,000 images



CalTech 101, 2003: ~9,000 images



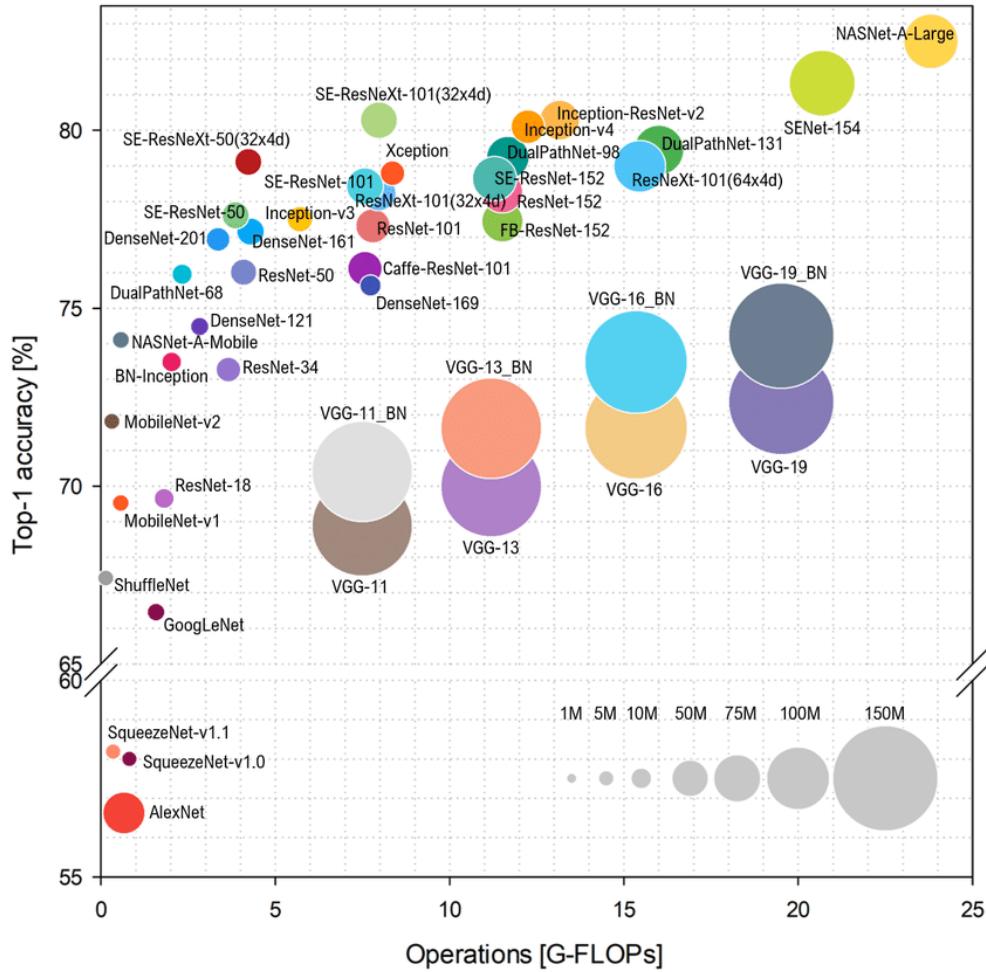
CIFAR 10, 2009: ~60,000 images



ILSVRC (ImageNet), 2009: 1.5 million images



How does it scale with compute?



What about NLP?

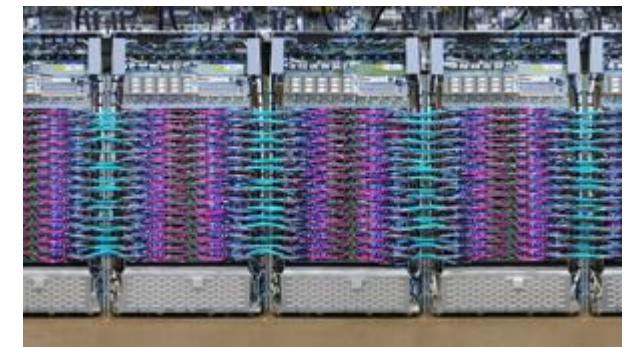
how long does it take to train BERT

All News Shopping Images

About 21,700,000 results (0.78 seconds)

about 54 hours

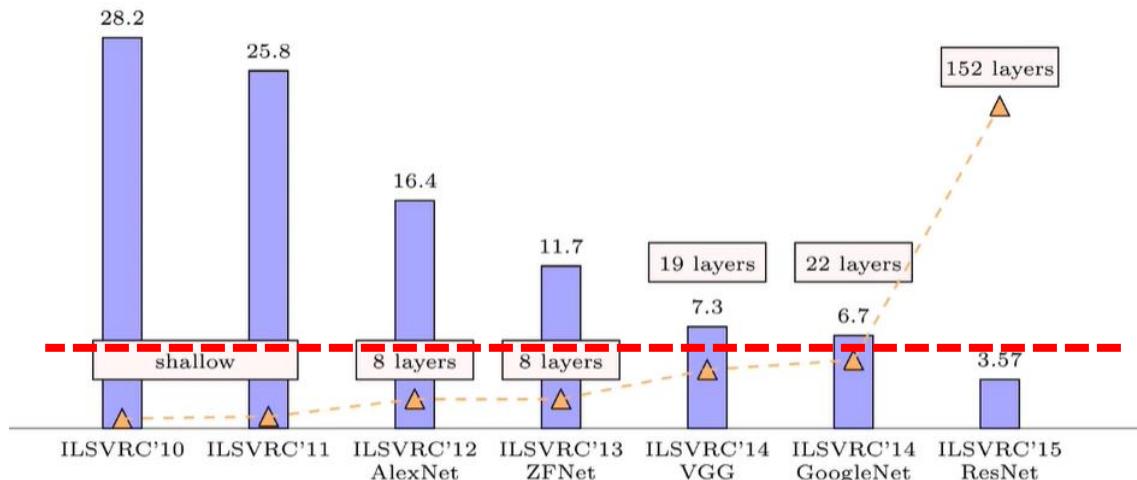
On what?? on this:



about 16 TPUs
(this photo shows a few thousand of these)

So... it's really expensive?

- **One perspective:** deep learning is not such a good idea, because it requires huge models, huge amounts of data, and huge amounts of compute
- **Another perspective:** deep learning is great, because as we add more data, more layers, and more compute, the models get better and better!



human performance:
about 5% error

...which human?



Andrej Karpathy blog

About

What I learned from competing against a ConvNet on
ImageNet

Sep 2, 2014

The underlying themes

- **Acquire representations** by using **high-capacity** models and lots of **data**, without requiring manual engineering of features or representations
 - Automation: we don't need to **know** what the good features are, we can have the model figure it out from data
 - Better performance: when representations are learned end-to-end, they are better tailored to the current task
- **Learning vs. inductive bias** (“nature vs. nurture”): models that get most of their performance from their data rather than from designer insight
 - **Inductive bias**: what we build into the model to make it learn effectively (we can never fully get rid of this!)
 - Should we build in **knowledge**, or better machinery for learning and scale?
- **Algorithms that scale**: This often refers to methods that can get better and better as we add more data, representational capacity, and compute

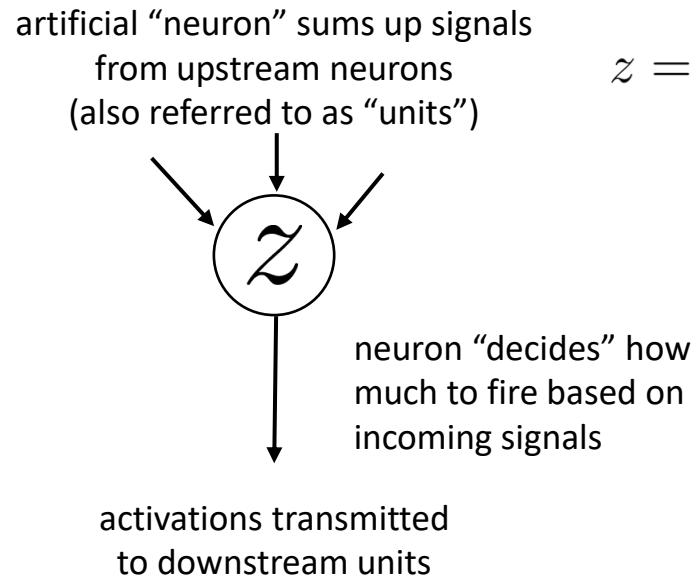
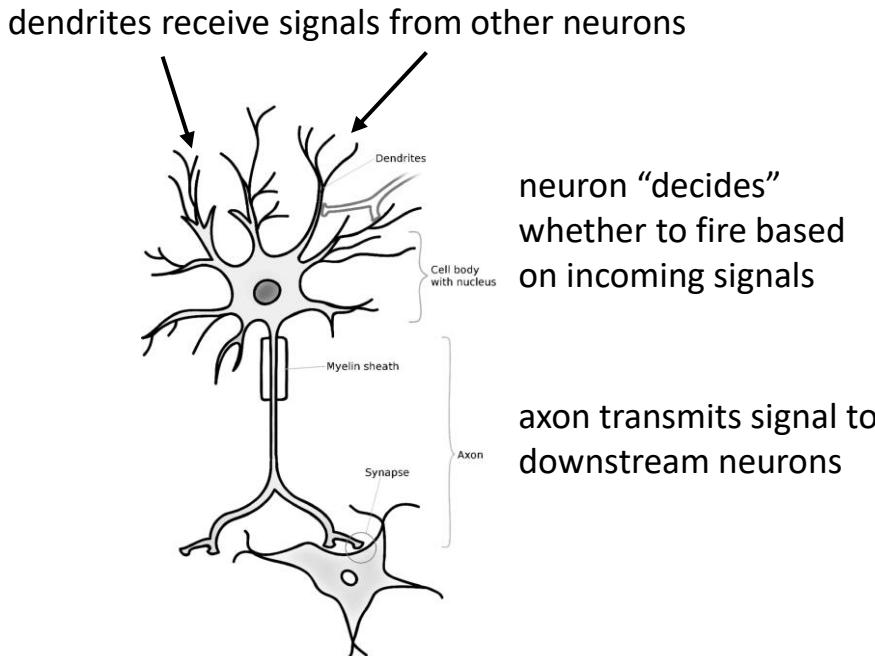
Model capacity: (informally) how many different functions a particular model class can represent (e.g., all linear decision boundaries vs. non-linear boundaries).

Inductive bias: (informally) built-in knowledge or biases in a model designed to help it learn. All such knowledge is “bias” in the sense that it makes some solutions more likely and some less likely.

Scaling: (informally) ability for an algorithm to work better as more data and model capacity is added.

Why do we call them neural nets?

Early on, neural networks were proposed as a rudimentary model of neurons in the brain



$$z = \sum_i a_i$$

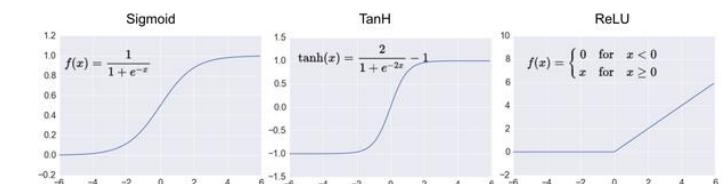
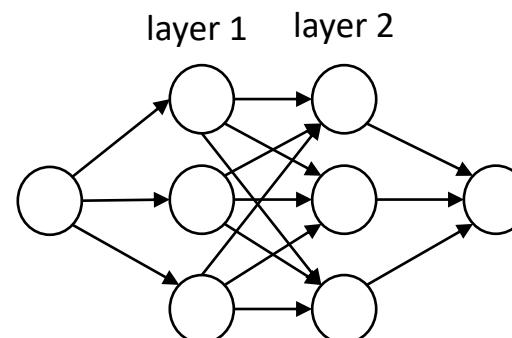
upstream activations

$$a = \sigma(z)$$

"activation function"

Is this a good model for real neurons?

- Crudely models *some* neuron function
- Missing many other important anatomical details
- Don't take it too seriously



What does deep learning have to do with the brain?

Unsupervised learning models of primary cortical receptive fields and receptive field plasticity

Andrew Saxe, Maneesh Bhand, Ritvik Mudur, Bipin Suresh, Andrew Y. Ng

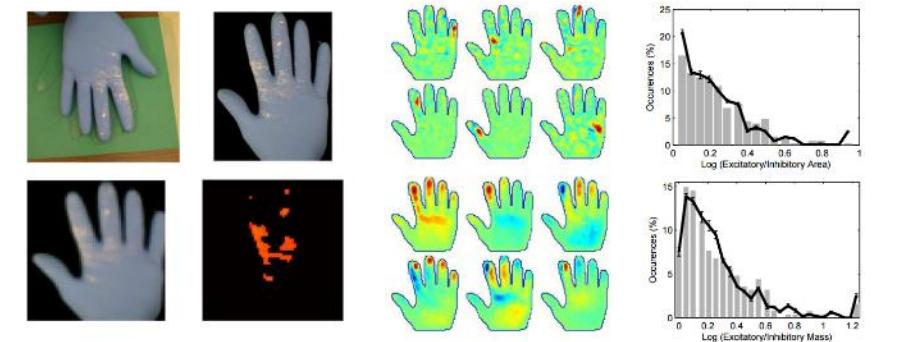
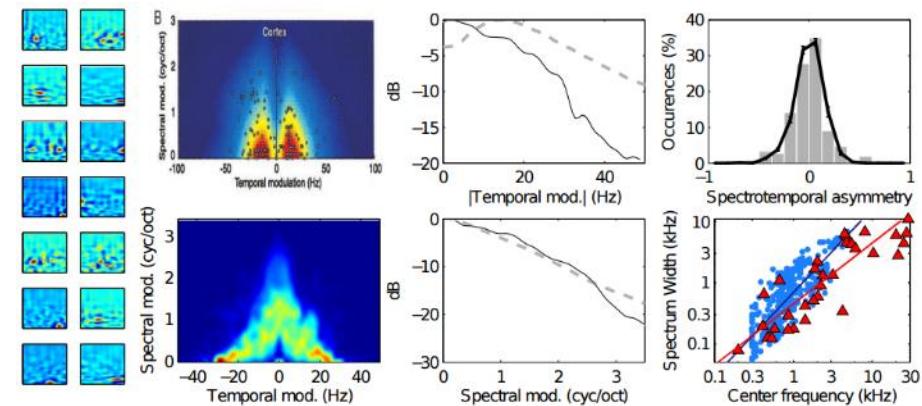
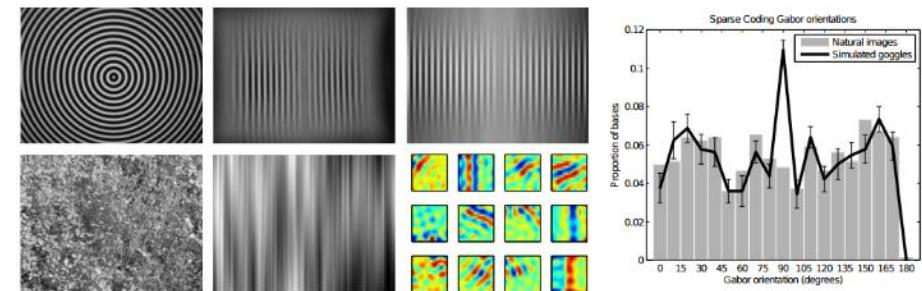
Department of Computer Science

Stanford University

{asaxe, mbhand, rmudur, bipins, ang}@cs.stanford.edu

Does this mean that the brain does deep learning?

Or does it mean that any sufficiently powerful learning machine will basically derive the same solution?



Introduction to Machine Learning

Designing, Visualizing and Understanding Deep Neural Networks

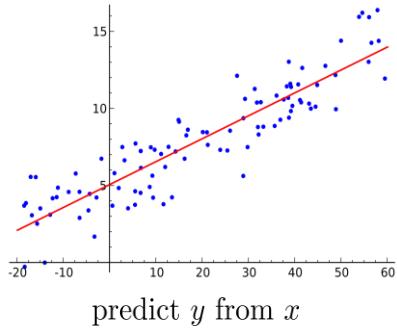
CS W182/282A

Instructor: Sergey Levine
UC Berkeley



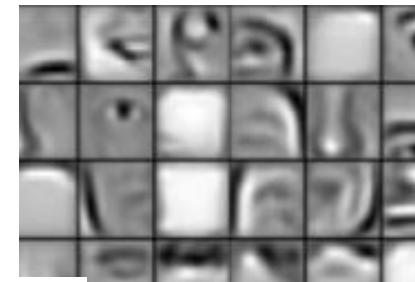
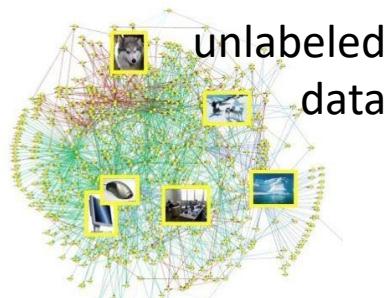
How do we formulate learning problems?

Different types of learning problems



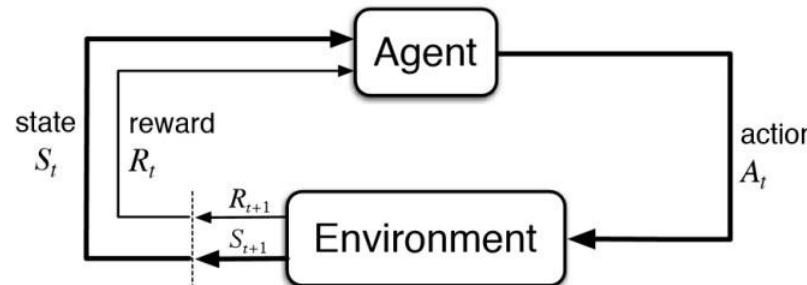
$$f_{\theta}(x) = y \rightarrow [\text{object label}]$$

supervised learning



representation

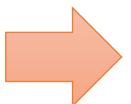
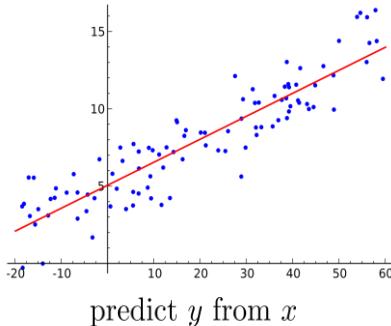
unsupervised learning



reinforcement learning

Supervised learning

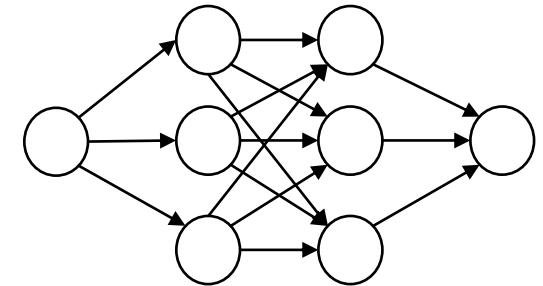
Given: $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ learn $f_\theta(x) \approx y$



$$f_\theta(x) = y$$



[object label]



Questions to answer:

how do we represent $f_\theta(x)$?

$$f_\theta(x) = \theta_1 x_1 + \theta_2 x_2 + \theta_3$$

$$f_\theta(x) = \theta_1 x + \theta_2 x^2 + \theta_3 x^3$$

how do we measure difference between $f_\theta(x_i)$ and y_i ?

$$\|f_\theta(x_i) - y_i\|^2$$

$$\delta(f_\theta(x_i) \neq y_i)$$

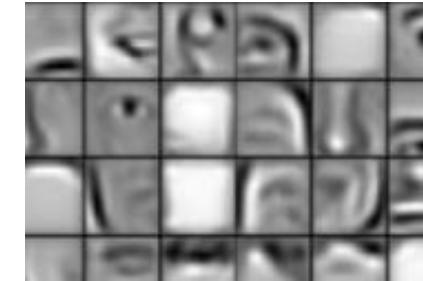
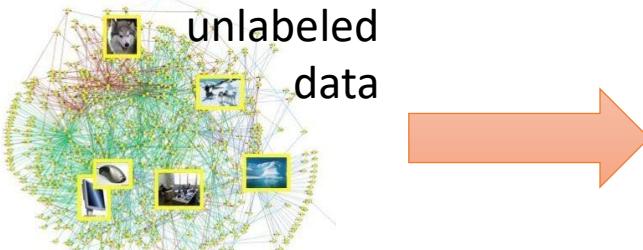
how do we find the best setting of θ ?

gradient descent

random search

least squares

Unsupervised learning



representation

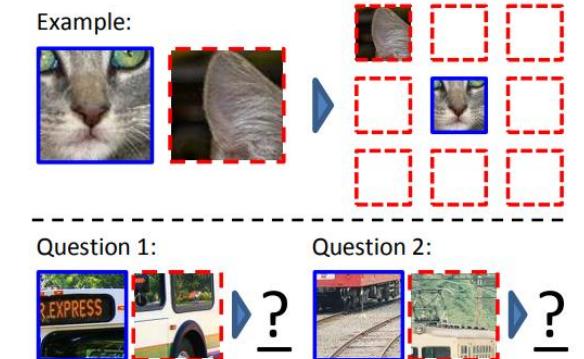
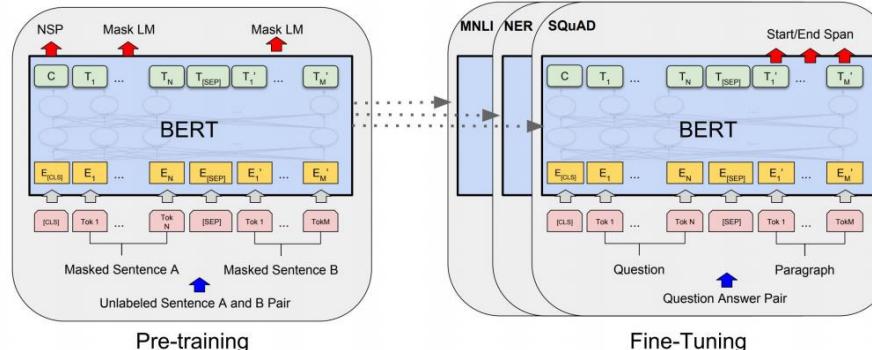
what does that mean?

generative modeling:

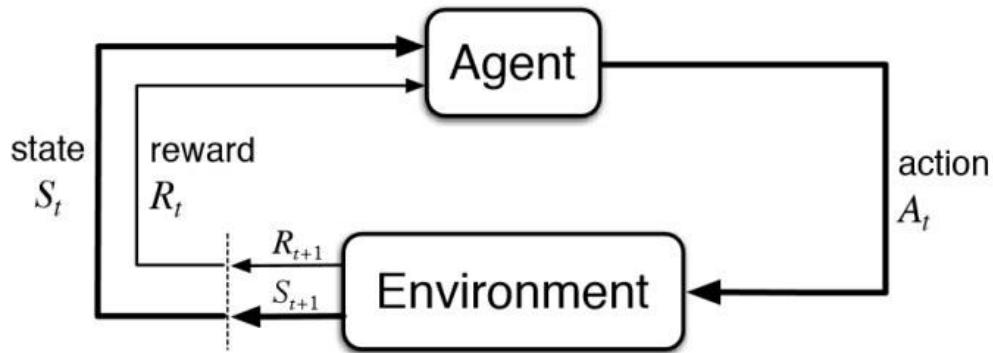


GANs
VAEs
pixel RNN, etc.

self-supervised
representation learning:



Reinforcement learning



choose $f_\theta(s_t) = a_t$

to maximize $\sum_{t=1}^H r(s_t, a_t)$

actually subsumes (generalizes) supervised learning!

supervised learning: get $f_\theta(x_i)$ to match y_i

reinforcement learning: get $f_\theta(s_t)$ to maximize reward (could be anything)



Actions: muscle contractions
Observations: sight, smell
Rewards: food

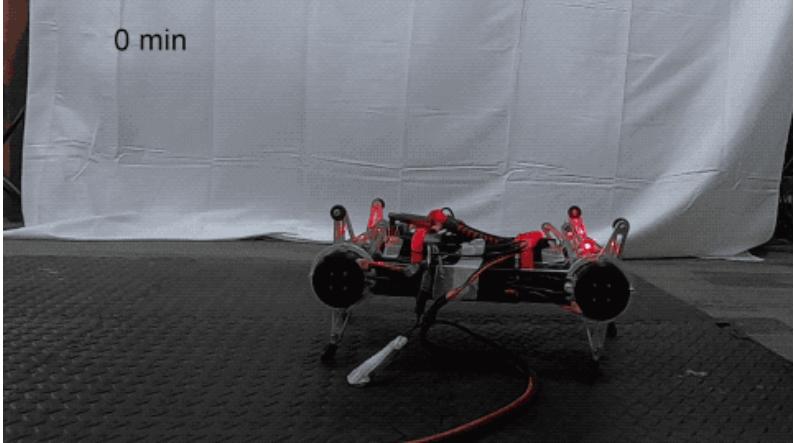


Actions: motor current or torque
Observations: camera images
Rewards: task success measure (e.g., running speed)



Actions: what to purchase
Observations: inventory levels
Rewards: profit

Reinforcement learning



Haarnoja et al., 2019

But many other application areas too!

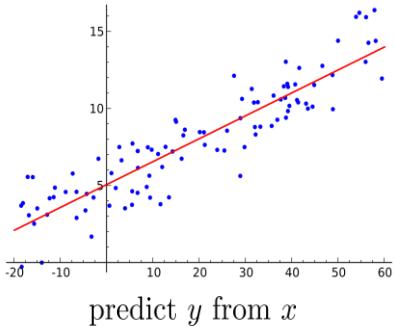
- Education (recommend which topic to study next)
- YouTube recommendations!
- Ad placement
- Healthcare (recommending treatments)



Let's start with supervised learning...

Supervised learning

Given: $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ learn $f_\theta(x) \approx y$



$$\text{predict } y \text{ from } x \rightarrow f_\theta(x) = y \rightarrow [\text{object label}]$$

The overwhelming majority of machine learning that is used in industry is supervised learning

- Encompasses all prediction/recognition models trained from ground truth data
- Multi-billion \$/year industry!
- Simple basic principles

Example supervised learning problems

Given: $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ learn $f_\theta(x) \approx y$

Predict...

category of object

sentence in French

presence of disease

text of a phrase

y

Based on...

image

sentence in English

X-ray image

audio utterance

x

Prediction is difficult

	0	1	2	3	4	5	6	7	8	9
5?	0%	0%	0%	0%	0%	90%	8%	0%	2%	0%
9?	4%	0%	0%	0%	11%	0%	4%	0%	6%	75%
3?	5%	0%	0%	40%	0%	30%	20%	0%	5%	0%
4?	5%	0%	0%	0%	50%	0%	3%	0%	2%	40%
0?	70%	0%	20%	0%	0%	0%	0%	0%	10%	0%

Predicting probabilities

Often makes more sense than predicting discrete labels

We'll see later why it is also **easier** to learn, due to smoothness

Intuitively, we can't change a discrete label "a tiny bit," it's all or nothing

But we **can** change a probability "a tiny bit"

Given: $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$

learn $f_\theta(x) \approx y$ $p_\theta(y|x)$



$$p_\theta(y|x)$$

Conditional probabilities

x random variable representing the **input**

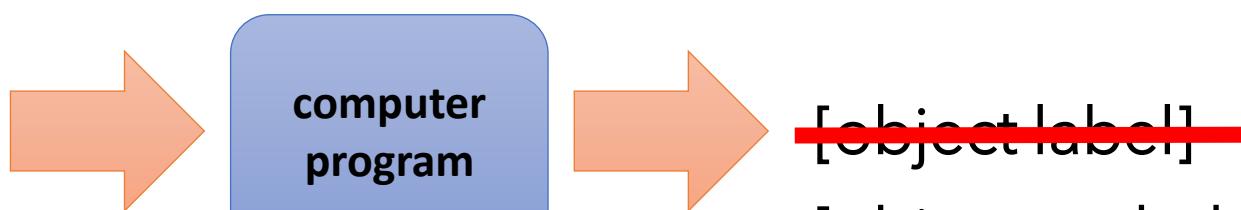
why is it a **random** variable?

y random variable representing the **output**

$$p(x, y) = p(x)p(y|x) \quad \text{chain rule}$$

$$p(y|x) = \frac{p(x, y)}{p(x)} \quad \text{definition of conditional probability}$$

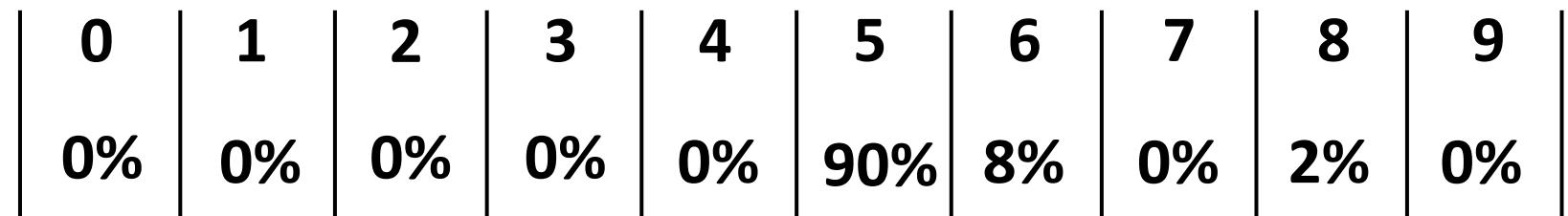
How do we represent it?



[object label]

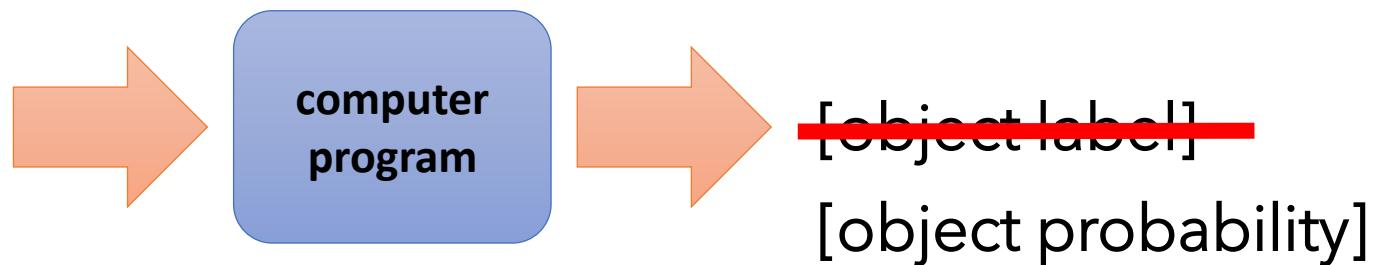
[object probability]

3



10 possible labels, output 10 numbers
(that are positive and sum to 1.0)

How do we represent it?



how about:

$$p(y = \text{dog}|x) = x^T \theta_{\text{dog}}$$

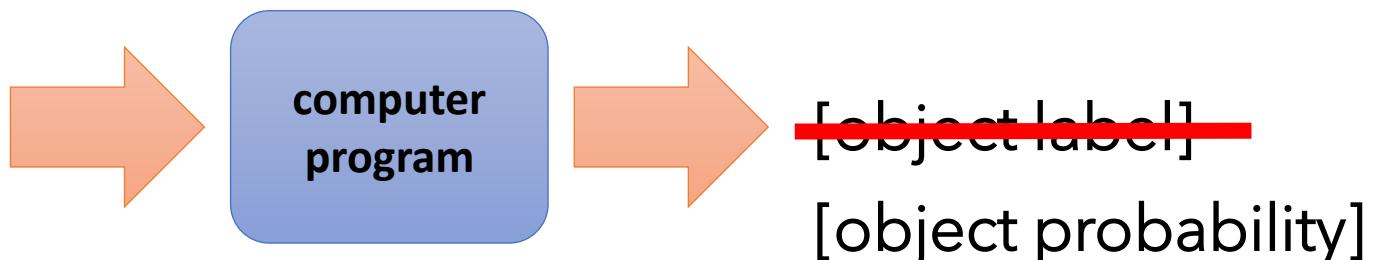
$$p(y = \text{cat}|x) = x^T \theta_{\text{cat}}$$

$$\vec{\theta} = \{\theta_{\text{dog}}, \theta_{\text{cat}}\}$$

if below_line(x) then: ○
else: ✖

(that are positive and sum to 1.0)

How do we represent it?



how about:

$$f_{\text{dog}}(x) = x^T \theta_{\text{dog}}$$

$$f_{\text{cat}}(x) = x^T \theta_{\text{cat}}$$

$$\vec{\theta} = \{\theta_{\text{dog}}, \theta_{\text{cat}}\}$$

if below_line(x)
else: ✗

why any function?

$$p(y|x) = \text{softmax}(f_{\text{dog}}(x), f_{\text{cat}}(x))$$



could be any (ideally one to one & onto) function that takes these inputs and outputs probabilities that are **positive** and **sum to 1**

How do we represent it?

how about:

$$f_{\text{dog}}(x) = x^T \theta_{\text{dog}}$$

$$f_{\text{cat}}(x) = x^T \theta_{\text{cat}}$$

$$\vec{\theta} = \{\theta_{\text{dog}}, \theta_{\text{cat}}\}$$

$$p(y|x) = \text{softmax}(f_{\text{dog}}(x), f_{\text{cat}}(x))$$



could be any (ideally one to one & onto)
function that takes these inputs and outputs
probabilities that are **positive** and **sum to 1**

how to make a number z positive?

$$z^2 \quad |z| \quad \max(0, z) \quad \exp(z)$$

especially convenient because it's one to one & onto
maps entire real number line to entire set of positive reals
(but don't overthink it, any one of these would work)

how to make a bunch of numbers sum to 1?

$$\frac{z_1}{z_1 + z_2} \quad \frac{z_1}{\sum_{i=1}^n z_i}$$

How do we represent it?

how about:

$$f_{\text{dog}}(x) = x^T \theta_{\text{dog}}$$

$$p(y|x) = \text{softmax}(f_{\text{dog}}(x), f_{\text{cat}}(x))$$

$$f_{\text{cat}}(x) = x^T \theta_{\text{cat}}$$

$$\vec{\theta} = \{\theta_{\text{dog}}, \theta_{\text{cat}}\}$$

$$\text{softmax}_{\text{dog}}(f_{\text{dog}}(x), f_{\text{cat}}(x)) = \frac{\exp(f_{\text{dog}}(x))}{\exp(f_{\text{dog}}(x)) + \exp(f_{\text{cat}}(x))}$$

makes it positive

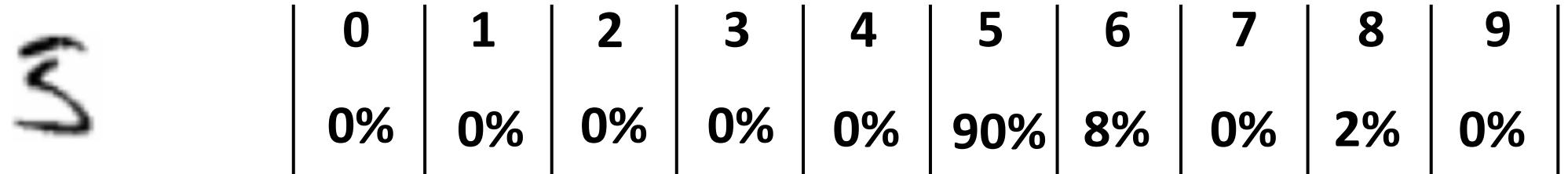
makes it sum to 1

There is nothing magical about this

It's not the only way to do it

Just need to get the numbers to be positive and sum to 1!

The softmax in general



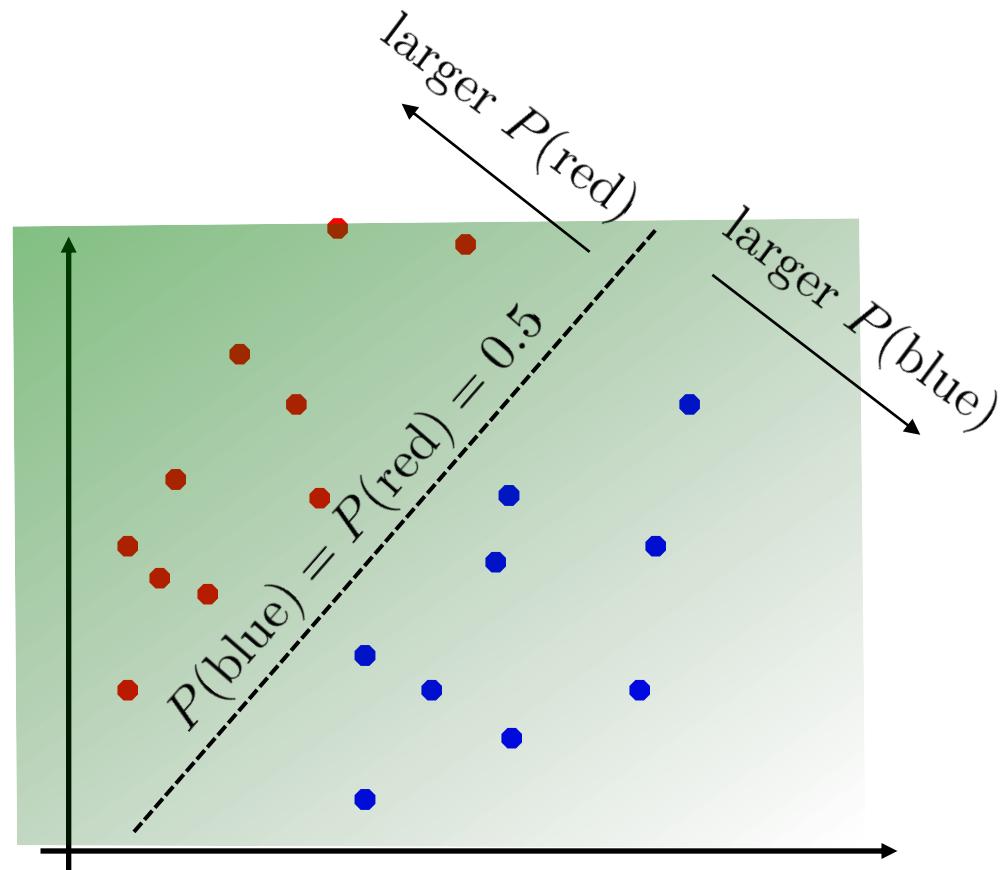
N possible labels

$p(y|x)$ – vector with N elements

$f_\theta(x)$ – vector-valued function with N outputs

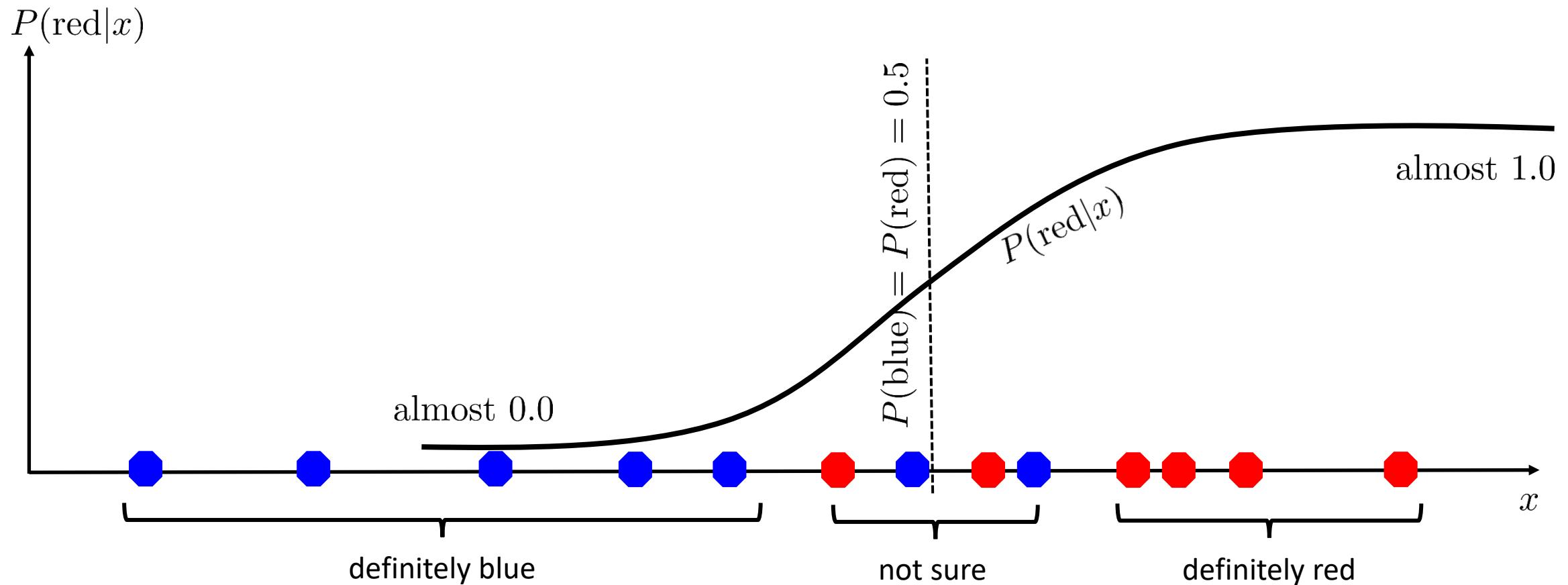
$$p(y = i|x) = \text{softmax}(f_\theta(x))[i] = \frac{\exp(f_{\theta,i}(x))}{\sum_{j=1}^N \exp(f_{\theta,j}(x))}$$

An illustration: 2D case



As $\theta_y^T x$ gets bigger, $p(y|x)$ gets bigger

An illustration: 1D case

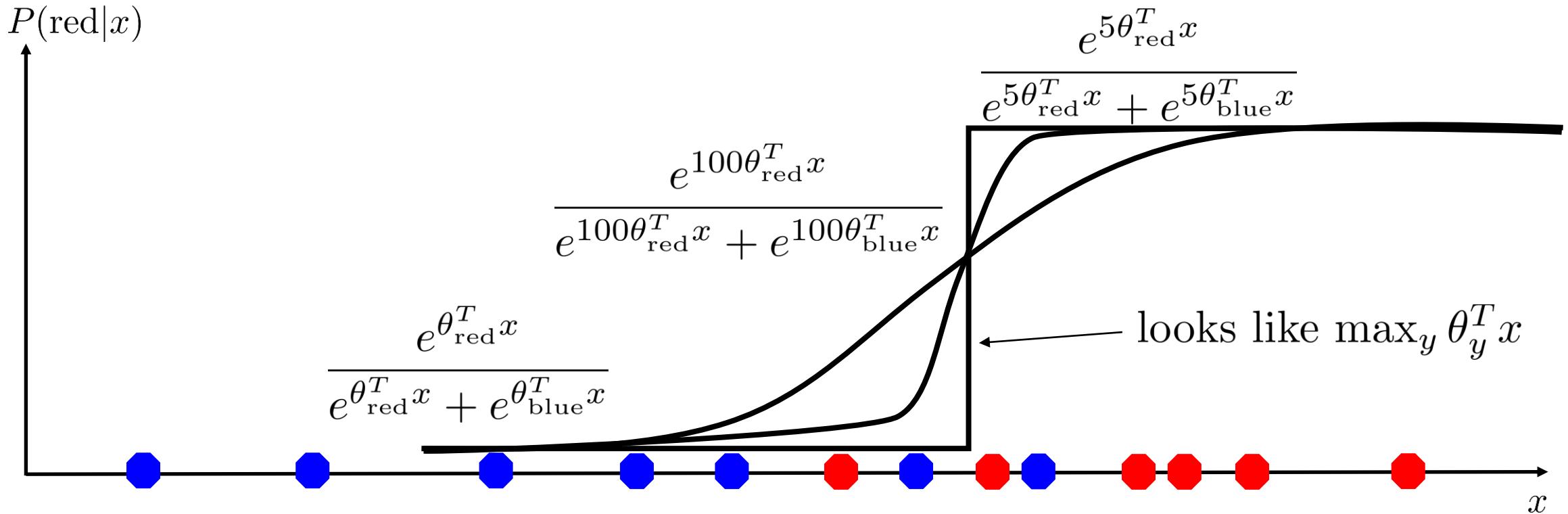


$$P(\text{red}|x) = \frac{e^{\theta_{\text{red}}^T x}}{e^{\theta_{\text{red}}^T x} + e^{\theta_{\text{blue}}^T x}}$$

probability increases exponentially as we move away from boundary

normalizer

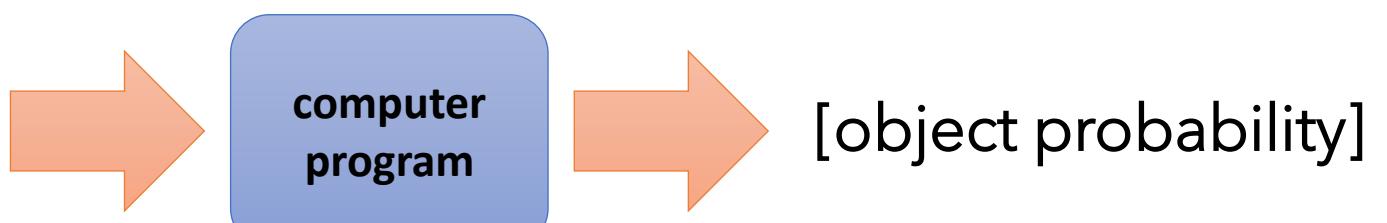
Why is it called a softmax?



$$P(\text{red}|x) = \frac{e^{\theta_{\text{red}}^T x}}{e^{\theta_{\text{red}}^T x} + e^{\theta_{\text{blue}}^T x}}$$

Loss functions

So far...



$$f_{\text{dog}}(x) = x^T \theta_{\text{dog}}$$

$$p(y|x) = \text{softmax}(f_{\text{dog}}(x), f_{\text{cat}}(x))$$

$$f_{\text{cat}}(x) = x^T \theta_{\text{cat}}$$

$$p(y = i|x) = \text{softmax}(f_{\theta}(x))[i] = \frac{\exp(f_{\theta,i}(x))}{\sum_{j=1}^N \exp(f_{\theta,j}(x))}$$

$\vec{\theta} = \{\theta_{\text{dog}}, \theta_{\text{cat}}\}$

this has learned parameters

How do we select $\vec{\theta}$?

The machine learning method

for solving any problem ever

1. Define your **model class**

How do **represent** the “program”

We (mostly) did this in the last section

(though we’ll spend a lot more time on this later)

2. Define your **loss function**

How to measure if one **model** in your **model class** is better than another?

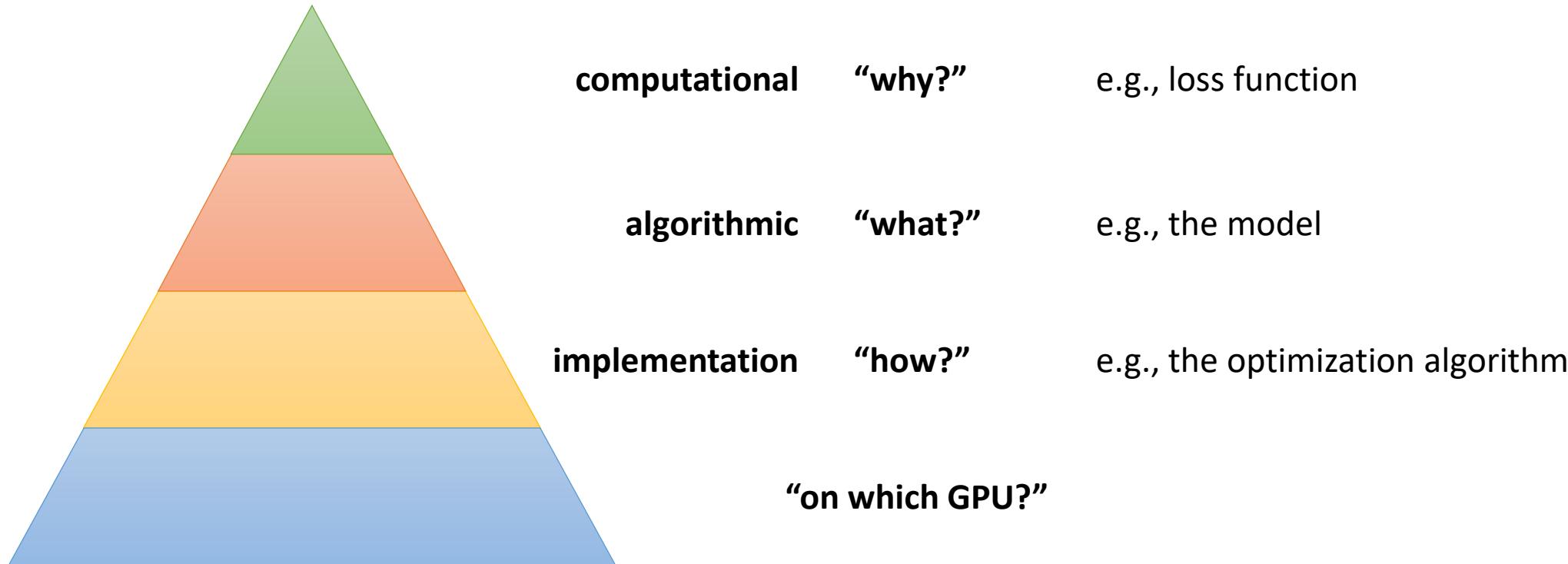
3. Pick your **optimizer**

How to **search** the **model class** to find the model that minimizes the **loss function**?

4. Run it on a big GPU



Aside: Marr's levels of analysis



There are many variants on this basic idea...

The machine learning method for solving any problem ever

1. Define your **model class**

How do **represent** the “program”

2. Define your **loss function**

We (mostly) did this in the last section

(though we'll spend a lot more time on this later)

3. Pick your **optimizer**

How to **measure** if one **model** in your **model class** is better than another?

4. Run it on a big GPU



How is the dataset “generated”?



$$\sim p(x)$$

probability distribution
over photos

$$\text{“dog”} \sim p(y|x)$$

conditional probability
distribution over labels

$$\text{result: } (x, y) \sim p(x, y)$$

How is the dataset “generated”?

$$(x, y) \sim p(x, y)$$

Training set: $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$

what is $p(\mathcal{D})$?

every (x_i, y_i) independent of each (x_j, y_j)
when is this true? when is this false?

key assumption: *independent* and *identically distributed* (i.i.d.)

exactly the same for all i

$$(x_i, y_i) \sim \underline{p(x, y)}$$

when i.i.d.: $p(\mathcal{D}) = \prod_i p(x_i, y_i)$

How is the dataset “generated”?

when i.i.d.: $p(\mathcal{D}) = \prod_i p(x_i, y_i) = \prod_i p(x_i)p(y_i|x_i)$

we are learning $p_\theta(y|x)$ it's a “model” of the true $p(y|x)$

a good model should make the data look probable

idea: choose θ such that

$$p(\mathcal{D}) = \prod_i p(x_i)p_\theta(y_i|x_i)$$

is maximized

what's the problem?



How is the dataset “generated”?

$$p(\mathcal{D}) = \prod_i p(x_i) p_\theta(y_i|x_i)$$



multiplying together many numbers ≤ 1

$$\log p(\mathcal{D}) = \sum_i \log p(x_i) + \log p_\theta(y_i|x_i) = \sum_i \log p_\theta(y_i|x_i) + \text{const}$$

$$\theta^* \leftarrow \arg \max_{\theta} \sum_i \log p_\theta(y_i|x_i)$$

maximum likelihood estimation (MLE)

$$\theta^* \leftarrow \arg \min_{\theta} - \sum_i \log p_\theta(y_i|x_i)$$

negative log-likelihood (NLL)
this is our **loss function!**

Loss functions

In general:

the *loss function* quantifies how *bad* θ is

we want the *least bad* (best) θ

Examples:

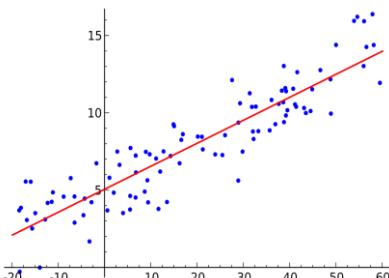
negative log-likelihood: $-\sum_i \log p_\theta(y_i|x_i)$

also called *cross-entropy* why?

zero-one loss: $\sum_i \delta(f_\theta(x_i) \neq y_i)$

mean squared error: $\sum_i \frac{1}{2} \|f_\theta(x_i) - y_i\|^2$

actually just negative log-likelihood! why?



aside: cross-entropy

how similar are two distributions, p_θ and p ?

$$H(p, p_\theta) = - \sum_y p(y|x_i) \log p_\theta(y|x_i)$$

assume $y_i \sim p(y|x_i)$

$$H(p, p_\theta) \approx - \log p_\theta(y_i|x_i)$$

Optimization

The machine learning method for solving any problem ever

1. Define your **model class**

$$f_{\text{dog}}(x) = x^T \theta_{\text{dog}} \quad p_{\theta}(y|x) = \text{softmax}(f_{\text{dog}}(x), f_{\text{cat}}(x))$$
$$f_{\text{cat}}(x) = x^T \theta_{\text{cat}}$$

2. Define your **loss function**

negative log-likelihood: $-\sum_i \log p_{\theta}(y_i|x_i)$

3. Pick your **optimizer**

4. Run it on a big GPU

The loss “landscape”

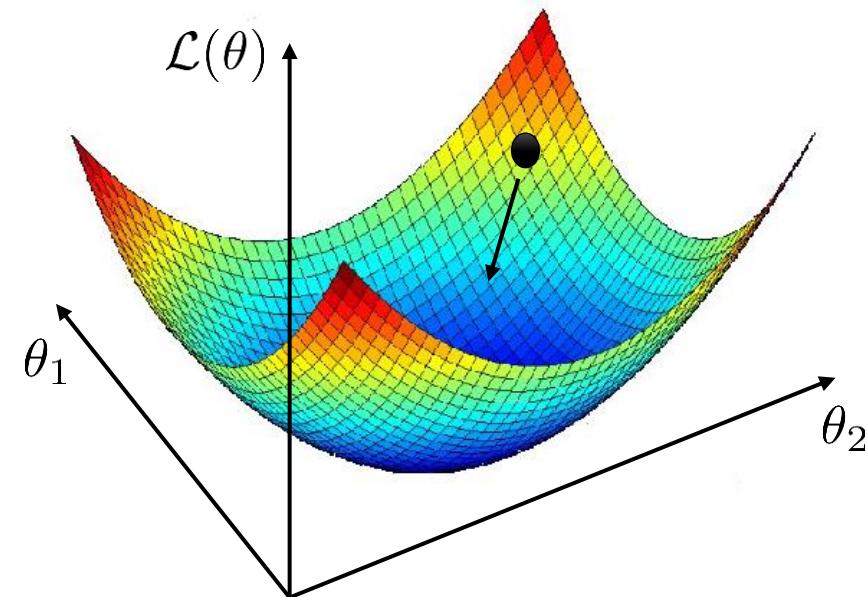
$$\theta^* \leftarrow \arg \min_{\theta} - \underbrace{\sum_i \log p_{\theta}(y_i | x_i)}_{\mathcal{L}(\theta)}$$

let's say θ is 2D

An algorithm:

1. Find a *direction* v where $\mathcal{L}(\theta)$ decreases
2. $\theta \leftarrow \theta + \alpha v$

some small constant
called “learning rate” or
“step size”

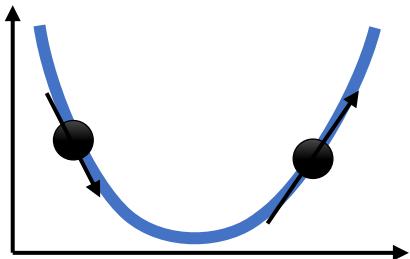


Gradient descent

An algorithm:

- 1. Find a *direction* v where $\mathcal{L}(\theta)$ decreases
- 2. $\theta \leftarrow \theta + \alpha v$

Which way does $\mathcal{L}(\theta)$ decrease?



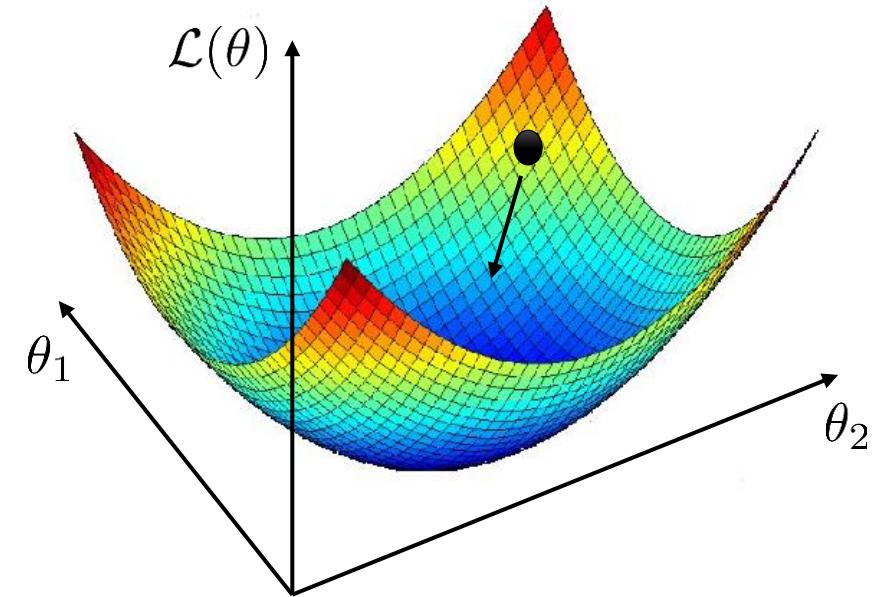
negative slope = go to the right

positive slope = go to the left

in general:

for each dimension, go in the direction
opposite the slope **along that dimension**

$$v_1 = -\frac{d\mathcal{L}(\theta)}{d\theta_1} \quad v_2 = -\frac{d\mathcal{L}(\theta)}{d\theta_2} \quad \text{etc.}$$



gradient:

$$\nabla_{\theta} \mathcal{L}(\theta) = \begin{pmatrix} \frac{d\mathcal{L}(\theta)}{d\theta_1} \\ \frac{d\mathcal{L}(\theta)}{d\theta_2} \\ \vdots \\ \frac{d\mathcal{L}(\theta)}{d\theta_n} \end{pmatrix}$$

Gradient descent

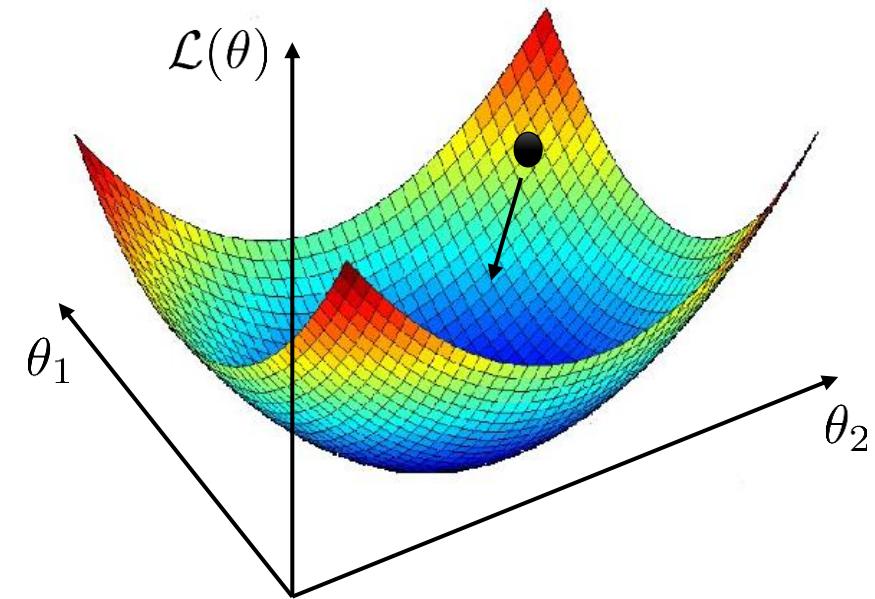
An algorithm:

- 1. Find a *direction* v where $\mathcal{L}(\theta)$ decreases
- 2. $\theta \leftarrow \theta + \alpha v$

Gradient descent:

- 1. Compute $\nabla_{\theta} \mathcal{L}(\theta)$
- 2. $\theta \leftarrow \theta - \alpha \nabla_{\theta} \mathcal{L}(\theta)$

We'll go into a lot more detail about gradient descent and related methods in a later lecture!



$$\nabla_{\theta} \mathcal{L}(\theta) = \begin{pmatrix} \frac{d\mathcal{L}(\theta)}{d\theta_1} \\ \frac{d\mathcal{L}(\theta)}{d\theta_2} \\ \vdots \\ \frac{d\mathcal{L}(\theta)}{d\theta_n} \end{pmatrix}$$

The machine learning method for solving any problem ever

1. Define your **model class**

$$f_{\text{dog}}(x) = x^T \theta_{\text{dog}} \quad p_{\theta}(y|x) = \text{softmax}(f_{\text{dog}}(x), f_{\text{cat}}(x))$$
$$f_{\text{cat}}(x) = x^T \theta_{\text{cat}}$$

2. Define your **loss function**

negative log-likelihood: $-\sum_i \log p_{\theta}(y_i|x_i)$

3. Pick your **optimizer**

Gradient descent:

- 
1. Compute $\nabla_{\theta} \mathcal{L}(\theta)$
 2. $\theta \leftarrow \theta - \alpha \nabla_{\theta} \mathcal{L}(\theta)$

4. Run it on a big GPU

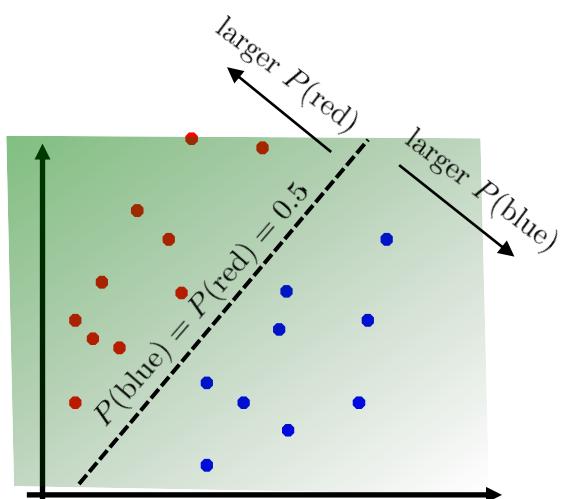
Logistic regression

$$f_{\theta}(x) = \begin{bmatrix} x^T \theta_{y_1} \\ x^T \theta_{y_2} \\ \vdots \\ x^T \theta_{y_m} \end{bmatrix}$$

$$f_{\theta}(x) = x^T \theta$$

matrix

$$x^T \times \begin{array}{|c|c|c|} \hline \theta_{y_1} & \theta_{y_2} & \theta_{y_3} \\ \hline \end{array} = \begin{bmatrix} x^T \theta_{y_1} \\ x^T \theta_{y_2} \\ \vdots \\ x^T \theta_{y_m} \end{bmatrix}$$



$$p_{\theta}(y = i|x) = \text{softmax}(f_{\theta}(x))[i] = \frac{\exp(f_{\theta,i}(x))}{\sum_{j=1}^m \exp(f_{\theta,j}(x))}$$

Gradient descent:

- 1. Compute $\nabla_{\theta} \mathcal{L}(\theta)$
- 2. $\theta \leftarrow \theta - \alpha \nabla_{\theta} \mathcal{L}(\theta)$

$$\mathcal{L}(\theta) = - \sum_{i=1}^n \log p_{\theta}(y_i|x_i)$$

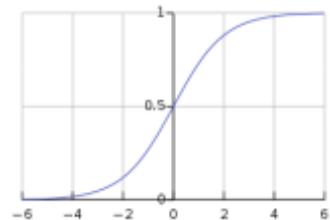
Special case: binary classification

What if we have only two classes?

$$P(y_1|x) = \frac{e^{\theta_{y_1}^T x}}{e^{\theta_{y_1}^T x} + e^{\theta_{y_2}^T x}}$$

$$P(y_1|x) = \frac{e^{\theta_{y_1}^T x}}{e^{\theta_{y_1}^T x} + e^{\theta_{y_2}^T x}}$$

Let $\theta_+ = \theta_{y_1} - \theta_{y_2}$



This is a bit *redundant*

Why?

$$P(y_1|x) + P(y_2|x) = 1$$

if we know $P(y_1|x)$, we know $P(y_2|x)$

$$\begin{aligned} & \text{multiply top and bottom by } e^{-\theta_{y_1}^T x} \\ & \frac{e^{\theta_{y_1}^T x} e^{-\theta_{y_1}^T x}}{(e^{\theta_{y_1}^T x} + e^{\theta_{y_2}^T x}) e^{-\theta_{y_1}^T x}} = \frac{e^{\theta_{y_1}^T x - \theta_{y_1}^T x}}{e^{\theta_{y_1}^T x - \theta_{y_1}^T x} + e^{\theta_{y_2}^T x - \theta_{y_1}^T x}} = 1 \\ & = 1 \end{aligned}$$

$$= \frac{1}{1 + e^{-\theta_+^T x}}$$

this is called the logistic equation
also referred to as a sigmoid

Empirical risk and true risk

zero-one loss: $\sum_i \delta(f_\theta(x_i) \neq y_i)$ 1 if wrong, 0 if right

Risk: probability you will get it wrong
expected value of our loss quantifies this
can be generalized to other losses
(e.g., NLL)



$\sim p(x)$

Risk = $E_{x \sim p(x), y \sim p(y|x)} [\mathcal{L}(x, y, \theta)]$ $y \sim p(y|x)$ how likely is it that $f_\theta(x)$ is wrong?

During training, we can't sample $x \sim p(x)$, we just have \mathcal{D}

Empirical risk = $\frac{1}{n} \sum_{i=1}^n \mathcal{L}(x_i, y_i, \theta) \approx E_{x \sim p(x), y \sim p(y|x)} [\mathcal{L}(x, y, \theta)]$

is this a **good** approximation?

Empirical risk minimization

$$\text{Empirical risk} = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(x_i, y_i, \theta) \approx E_{x \sim p(x), y \sim p(y|x)} [\mathcal{L}(x, y, \theta)]$$

Supervised learning is (usually) *empirical* risk minimization

Is this the same as *true* risk minimization?

Overfitting: when the empirical risk is low, but the true risk is high

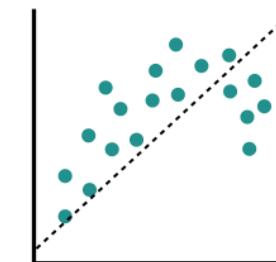
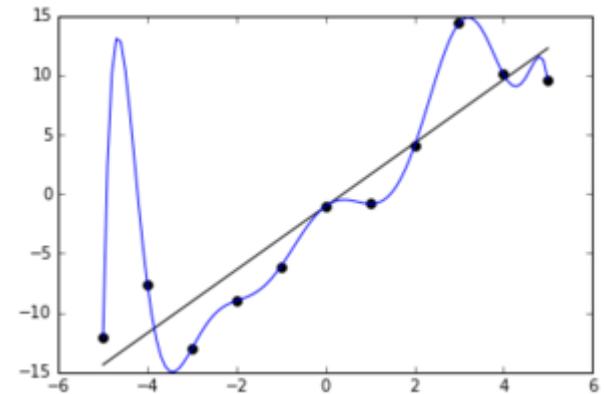
can happen if the dataset is too small

can happen if the model is too powerful (has too many parameters/capacity)

Underfitting: when the empirical risk is high, and the true risk is high

can happen if the model is too weak (has too few parameters/capacity)

can happen if your optimizer is not configured well (e.g., wrong learning rate)



This is very important, and we will discuss this in much more detail later!

Summary

1. Define your **model class**

$$f_{\text{dog}}(x) = x^T \theta_{\text{dog}} \quad p_{\theta}(y|x) = \text{softmax}(f_{\text{dog}}(x), f_{\text{cat}}(x))$$
$$f_{\text{cat}}(x) = x^T \theta_{\text{cat}}$$

2. Define your **loss function**

negative log-likelihood: $-\sum_i \log p_{\theta}(y_i|x_i)$

3. Pick your **optimizer**

Gradient descent:

- 
1. Compute $\nabla_{\theta} \mathcal{L}(\theta)$
 2. $\theta \leftarrow \theta - \alpha \nabla_{\theta} \mathcal{L}(\theta)$

4. Run it on a big GPU

Bias, Variance, and Regularization

Designing, Visualizing and Understanding Deep Neural Networks

CS W182/282A

Instructor: Sergey Levine
UC Berkeley



Will we get the right answer?

Empirical risk and true risk

zero-one loss: $\sum_i \delta(f_\theta(x_i) \neq y_i)$ 1 if wrong, 0 if right

Risk: probability you will get it wrong
expected value of our loss quantifies this
can be generalized to other losses
(e.g., NLL)



$\sim p(x)$

Risk = $E_{x \sim p(x), y \sim p(y|x)} [\mathcal{L}(x, y, \theta)]$ $y \sim p(y|x)$ how likely is it that $f_\theta(x)$ is wrong?

During training, we can't sample $x \sim p(x)$, we just have \mathcal{D}

Empirical risk = $\frac{1}{n} \sum_{i=1}^n \mathcal{L}(x_i, y_i, \theta) \approx E_{x \sim p(x), y \sim p(y|x)} [\mathcal{L}(x, y, \theta)]$

is this a **good** approximation?

Empirical risk minimization

$$\text{Empirical risk} = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(x_i, y_i, \theta) \approx E_{x \sim p(x), y \sim p(y|x)} [\mathcal{L}(x, y, \theta)]$$

Supervised learning is (usually) *empirical* risk minimization

Is this the same as *true* risk minimization?

Overfitting: when the empirical risk is low, but the true risk is high

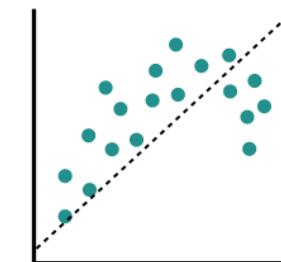
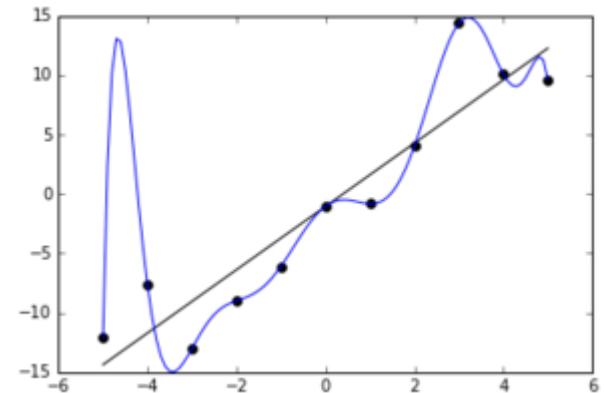
can happen if the dataset is too small

can happen if the model is too powerful (has too many parameters/capacity)

Underfitting: when the empirical risk is high, and the true risk is high

can happen if the model is too weak (has too few parameters/capacity)

can happen if your optimizer is not configured well (e.g., wrong learning rate)

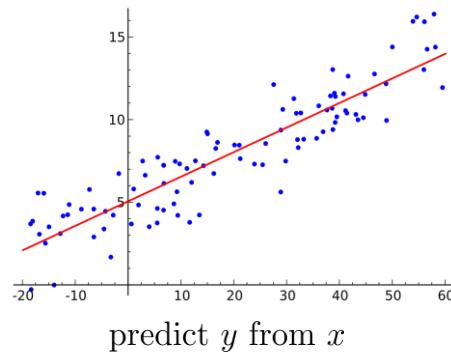
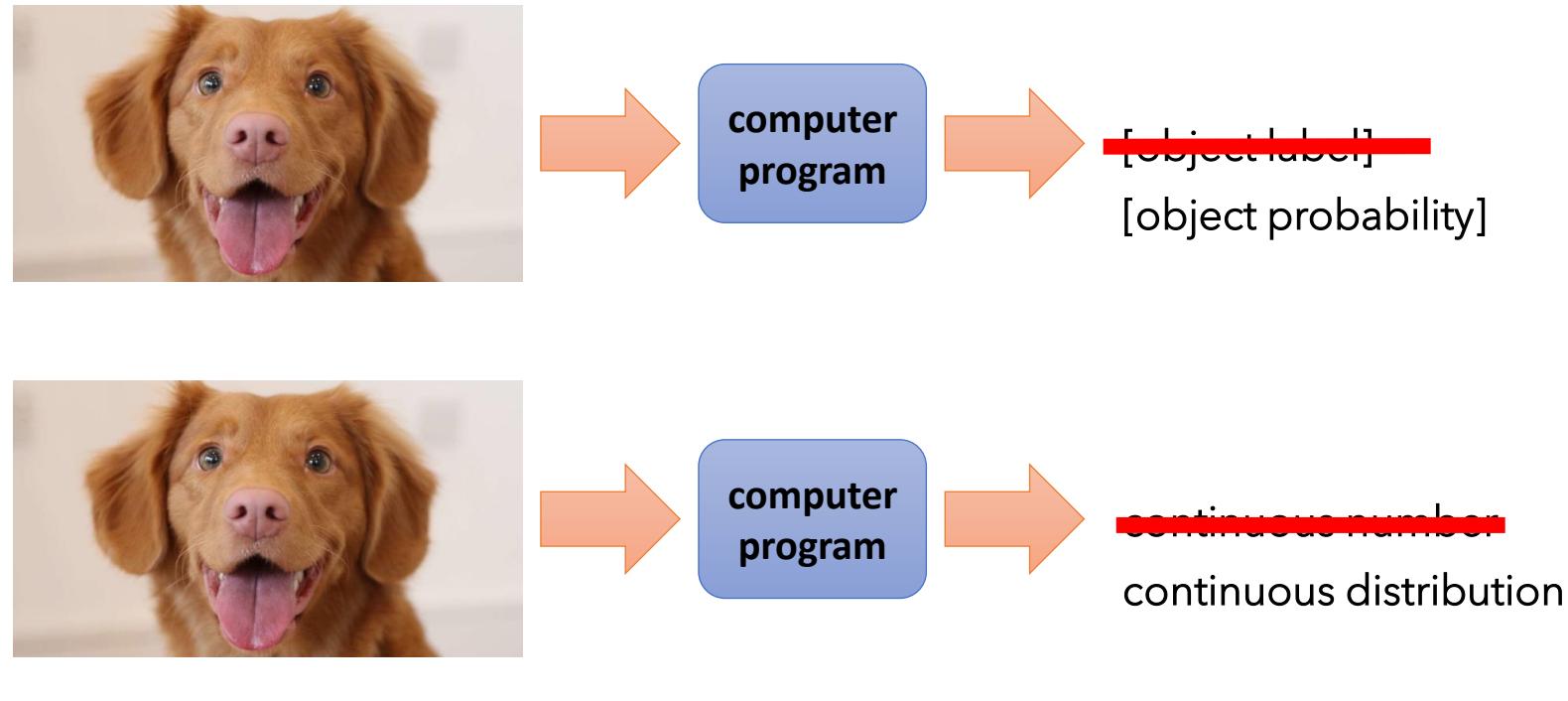


Let's analyze error!

Last time, we discussed **classification**

This time, we'll focus on **regression**

All this stuff applies to classification too,
it's just simpler to derive for regression

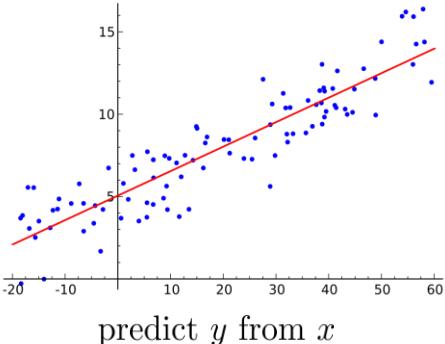


$$\log p_{\theta}(y|x) = \mathcal{N}(f_{\theta}(x), \Sigma_{\theta}(x)) = -\frac{1}{2}(f_{\theta}(x) - y)\Sigma_{\theta}(x)^{-1}(f_{\theta}(x) - y) - \frac{1}{2}\log |\Sigma_{\theta}(x)| + \text{const}$$

normal (Gaussian) distribution

$$\text{if } \Sigma_{\theta}(x) = \mathbf{I} \quad = -\frac{1}{2}\|f_{\theta}(x) - y\|^2 + \text{const}$$

Let's analyze error!



$$\log p_{\theta}(y|x) = -\frac{1}{2} \|f_{\theta}(x) - y\|^2 + \text{const} \quad \text{if } \Sigma_{\theta}(x) = \mathbf{I}$$

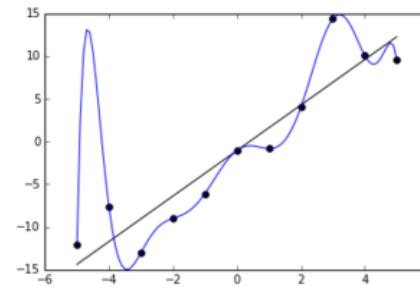
Also the same as the mean squared error (MSE) loss!

$$\mathcal{L}(\theta, x, y) = -\frac{1}{2} \|f_{\theta}(x) - y\|^2 \quad \leftarrow \text{a bit easier to analyze, but we can analyze other losses too}$$

Overfitting: when the empirical risk is low, but the true risk is high

can happen if the dataset is too small

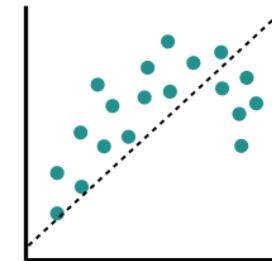
can happen if the model is too powerful (has too many parameters/capacity)



Underfitting: when the empirical risk is high, and the true risk is high

can happen if the model is too weak (has too few parameters/capacity)

can happen if your optimizer is not configured well (e.g., wrong learning rate)



Let's analyze error!

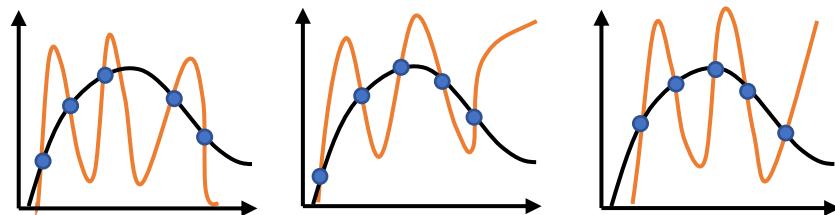
$$\mathcal{L}(\theta, x, y) = -\frac{1}{2} \|\mathbf{f}_\theta(x) - y\|^2$$

Let's try to understand **overfitting** and **underfitting** more formally

Question: how does the error change for different **training sets**?

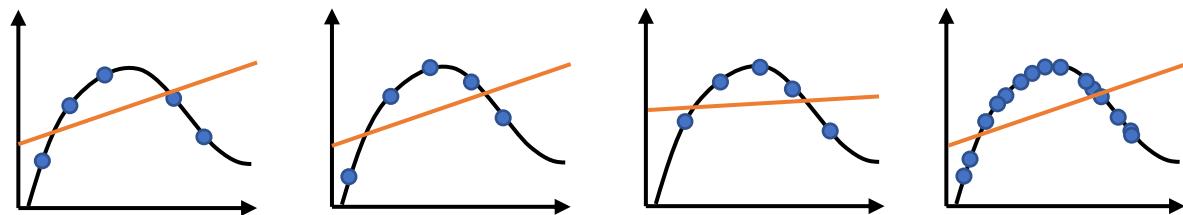
Why is this question important?

overfitting



- The training data is fitted well
- The true function is fitted poorly
- **The learned function looks different each time!**

underfitting



- The training data is fitted poorly
- The true function is fitted poorly
- **The learned function looks similar, even if we pool together all the datasets!**

Let's analyze error!

$$\mathcal{L}(\theta, x, y) = -\frac{1}{2} ||f_\theta(x) - y||^2$$

What is the **expected** error, given a distribution over datasets?



$$\begin{aligned}\mathcal{D} &= \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\} \\ &\sim p(x) \quad p(\mathcal{D}) = \prod_{i=1}^N p(x_i)p(y_i|x_i)\end{aligned}$$

$$y \sim p(y|x)$$

$$E_{\mathcal{D} \sim p(\mathcal{D})} [||f_\theta(x) - y||^2] = \sum_{\mathcal{D}} p(\mathcal{D}) ||f_{\mathcal{D}}(x) - f(x)||^2$$

expected value of error w.r.t. data distribution

sum over all possible datasets

Let's analyze error!

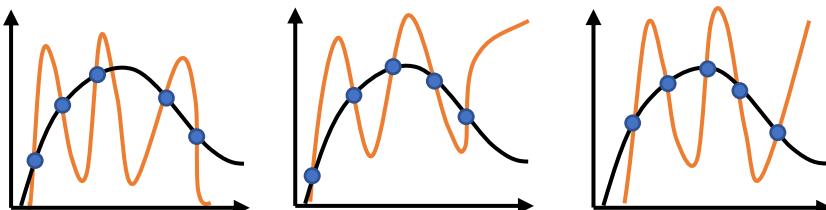
$$E_{\mathcal{D} \sim p(\mathcal{D})} [||f_{\mathcal{D}}(x) - f(x)||^2] = \sum_{\mathcal{D}} p(\mathcal{D}) ||f_{\mathcal{D}}(x) - f(x)||^2$$

Why do we care about this quantity?

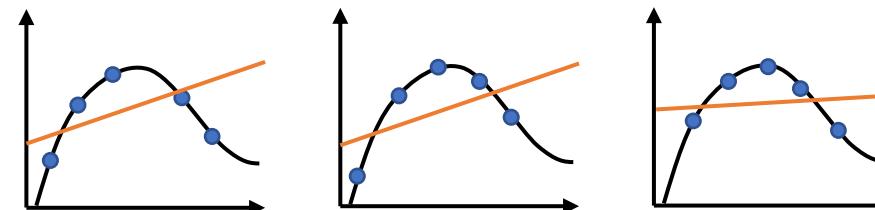
We want to understand how well our **algorithm** does independently of the **particular (random) choice of dataset**

This is very important if we want to **improve** our algorithm!

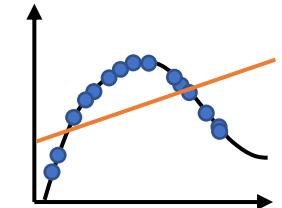
overfitting



underfitting



Bias-variance tradeoff



$$E_{\mathcal{D} \sim p(\mathcal{D})} [||f_{\mathcal{D}}(x) - f(x)||^2] \quad \text{let } \bar{f}(x) = E_{\mathcal{D} \sim p(\mathcal{D})}[f_{\mathcal{D}}(x)]$$

$$= E_{\mathcal{D} \sim p(\mathcal{D})} [||f_{\mathcal{D}}(x) - \bar{f}(x) + \bar{f}(x) - f(x)||^2]$$

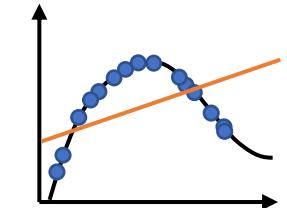
$$= E_{\mathcal{D} \sim p(\mathcal{D})} [||(f_{\mathcal{D}}(x) - \bar{f}(x)) + (\bar{f}(x) - f(x))||^2]$$

$$= E_{\mathcal{D} \sim p(\mathcal{D})} [||f_{\mathcal{D}}(x) - \bar{f}(x)||^2] + E_{\mathcal{D} \sim p(\mathcal{D})} [||\bar{f}(x) - f(x)||^2]$$

$$\underline{- E_{\mathcal{D} \sim p(\mathcal{D})} [2(f_{\mathcal{D}}(x) - \bar{f}(x))^T (\bar{f}(x) - f(x))]}$$

$$0 \quad \overbrace{\hspace{10em}}^{2E[(f_{\mathcal{D}}(x) - \bar{f}(x))]^T (\bar{f}(x) - f(x))}$$

Bias-variance tradeoff

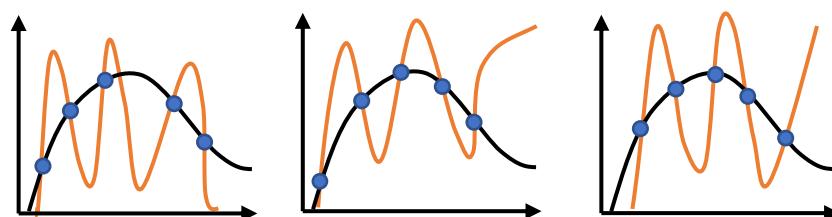


$$E_{\mathcal{D} \sim p(\mathcal{D})} [||f_{\mathcal{D}}(x) - f(x)||^2] \quad \text{let } \bar{f}(x) = E_{\mathcal{D} \sim p(\mathcal{D})}[f_{\mathcal{D}}(x)]$$

$$= E_{\mathcal{D} \sim p(\mathcal{D})} [||f_{\mathcal{D}}(x) - \bar{f}(x)||^2] + E_{\mathcal{D} \sim p(\mathcal{D})} [||\bar{f}(x) - f(x)||^2]$$

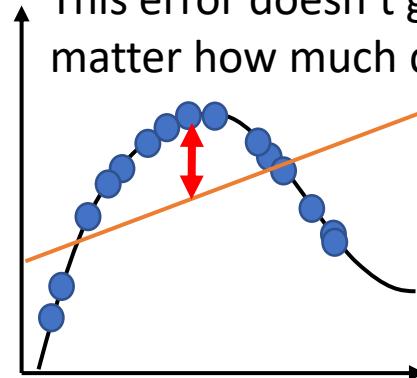
Variance

Regardless of what the true function is, how much does our prediction change with dataset?

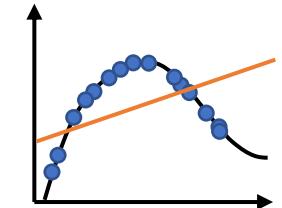


$||\bar{f}(x) - f(x)||^2$
Bias²

This error doesn't go away no matter how much data we have!



Bias-variance tradeoff



$$\begin{aligned} & E_{\mathcal{D} \sim p(\mathcal{D})} [\| f_{\mathcal{D}}(x) - f(x) \|^2] \quad \text{let } \bar{f}(x) = E_{\mathcal{D} \sim p(\mathcal{D})} [f_{\mathcal{D}}(x)] \\ &= E_{\mathcal{D} \sim p(\mathcal{D})} [\| f_{\mathcal{D}}(x) - \bar{f}(x) \|^2] + E_{\mathcal{D} \sim p(\mathcal{D})} [\| \bar{f}(x) - f(x) \|^2] \\ &= \text{Variance} + \text{Bias}^2 \end{aligned}$$

If **variance** is too high, we have too little data/too complex a function class/etc. => this is **overfitting**

If **bias** is too high, we have an insufficiently complex function class => this is **underfitting**

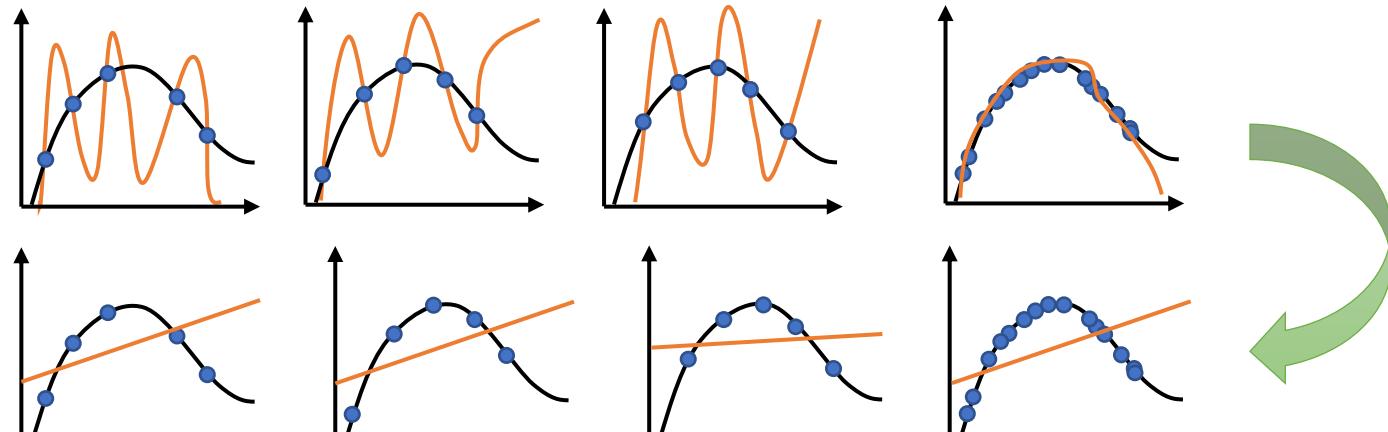
How do we **regulate** the bias-variance tradeoff?

Regularization

How to regulate bias/variance?

Get more data

addresses **variance**



has no effect on **bias**

Change your model class

e.g., 12th degree polynomials to linear functions

Can we “smoothly” restrict the model class?

Can we construct a “continuous knob” for complexity?

Regularization

Regularization: something we add to the loss function to reduce variance

Bayesian interpretation: could be regarded as a prior on parameters

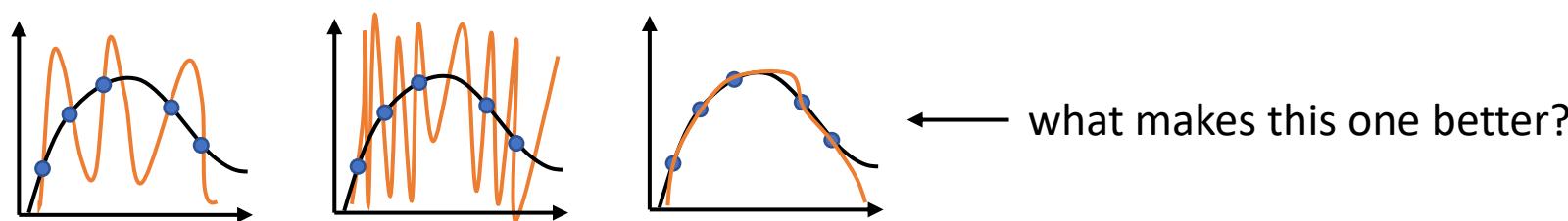
(but this is not the only interpretation!)

High level intuition:

When we have **high variance**, it's because the data doesn't give enough information to identify parameters

If there is not enough information in the **data**, can we give **more information** through the loss function?

If we provide enough **information to disambiguate** between (almost) equally good models, we can pick the best one



all of these solutions have zero training error

The Bayesian perspective

Regularization: something we add to the loss function to reduce variance

Bayesian interpretation: could be regarded as a prior on parameters **(but this is not the only interpretation!)**

Question: Given \mathcal{D} , what is the most likely θ ? $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$

$$p(\theta|\mathcal{D}) = \frac{p(\theta, \mathcal{D})}{p(\mathcal{D})} \propto p(\theta, \mathcal{D}) = p(\mathcal{D}|\theta)p(\theta) \longleftarrow \text{what is this part?}$$



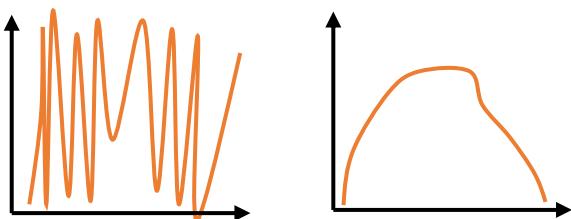
we've seen this part before!

Can we pick a prior that makes the *smoother* function more likely?

$$p(\mathcal{D}|\theta) = \prod_i p(x_i)p_\theta(y_i|x_i)$$



remember: this is just shorthand for
 $p(y_i|x_i, \theta)$



The Bayesian perspective

Regularization: something we add to the loss function to reduce variance

Bayesian interpretation: could be regarded as a prior on parameters **(but this is not the only interpretation!)**

Question: Given \mathcal{D} , what is the most likely θ ? $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$

$$p(\theta|\mathcal{D}) = \frac{p(\theta, \mathcal{D})}{p(\mathcal{D})} \propto p(\theta, \mathcal{D}) = p(\mathcal{D}|\theta)p(\theta)$$

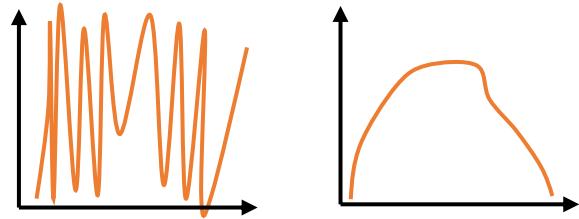
New loss function:

we **choose** this bit

$$\mathcal{L}(\theta) = - \left(\sum_{i=1}^N \log p(y_i|x_i, \theta) \right) - \log p(\theta)$$

Example: regularized linear regression

Can we pick a prior that makes the *smoother* function more likely?



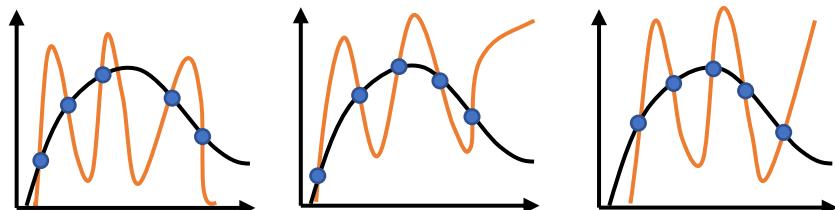
$$\mathcal{L}(\theta) = - \left(\sum_{i=1}^N \log p(y_i|x_i, \theta) \right) - \log p(\theta)$$

what kind of distribution assigns higher probabilities to **small** numbers?

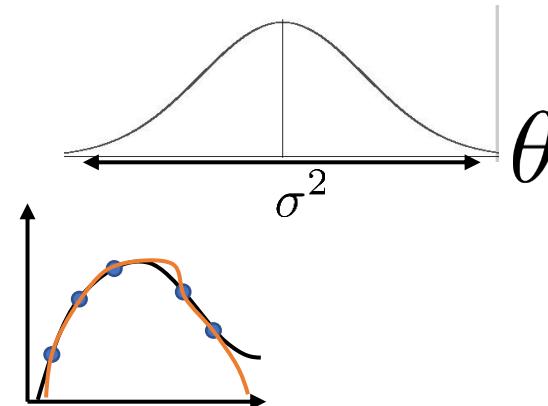
Simple idea: $p(\theta) = \mathcal{N}(0, \sigma^2)$

example: linear regression with polynomial features

$$f_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 \dots$$



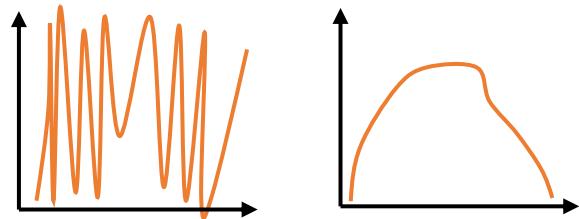
this kind of thing typically requires large coefficients



if we only allow **small** coefficients, best fit might be more like this

Example: regularized linear regression

Can we pick a prior that makes the *smoother* function more likely?



$$\mathcal{L}(\theta) = - \left(\sum_{i=1}^N \log p(y_i|x_i, \theta) \right) - \log p(\theta)$$

what kind of distribution assigns higher probabilities to **small** numbers?

Simple idea: $p(\theta) = \mathcal{N}(0, \sigma^2)$

$$\log p(\theta) = \sum_{i=1}^D -\frac{1}{2} \frac{\theta_i^2}{\sigma^2} - \underbrace{\log \sigma - \frac{1}{2} \log 2\pi}_{\text{doesn't influence } \theta}$$

$$= -\lambda \|\theta\|^2 + \text{const}$$

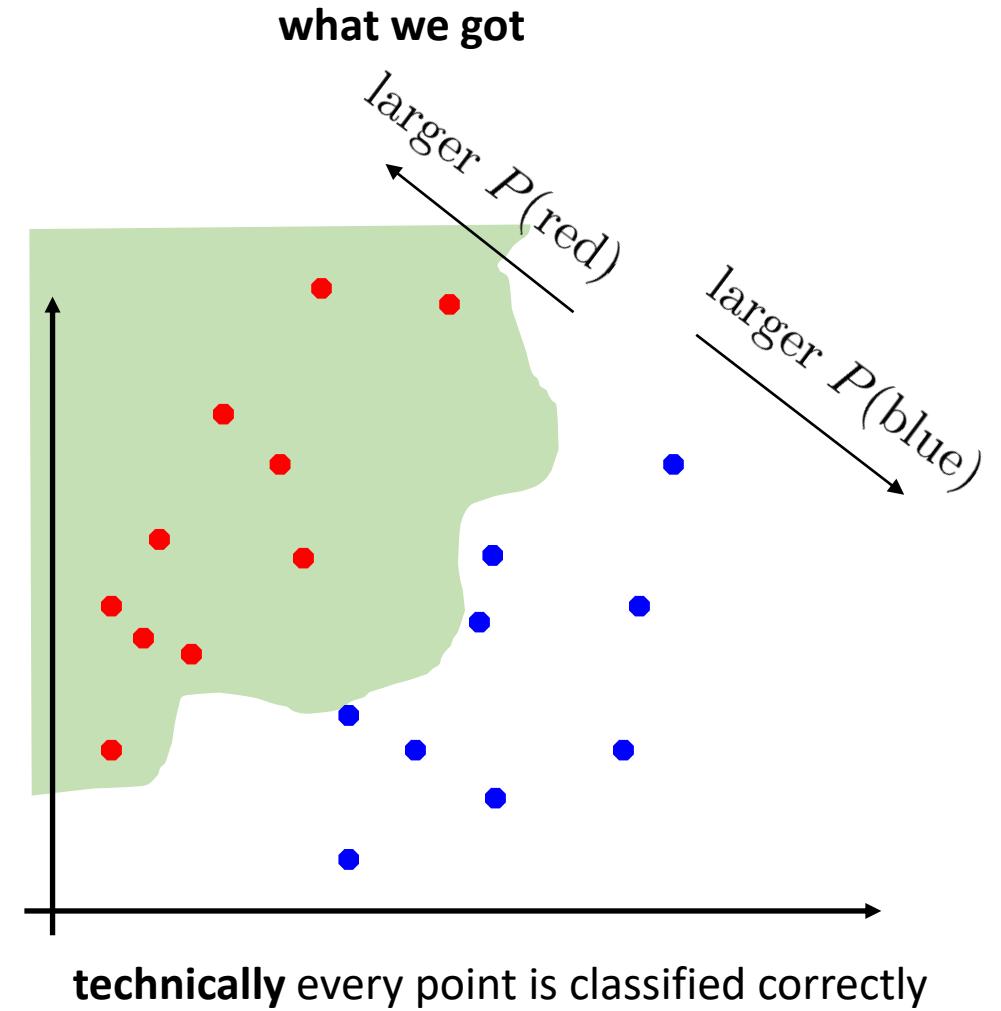
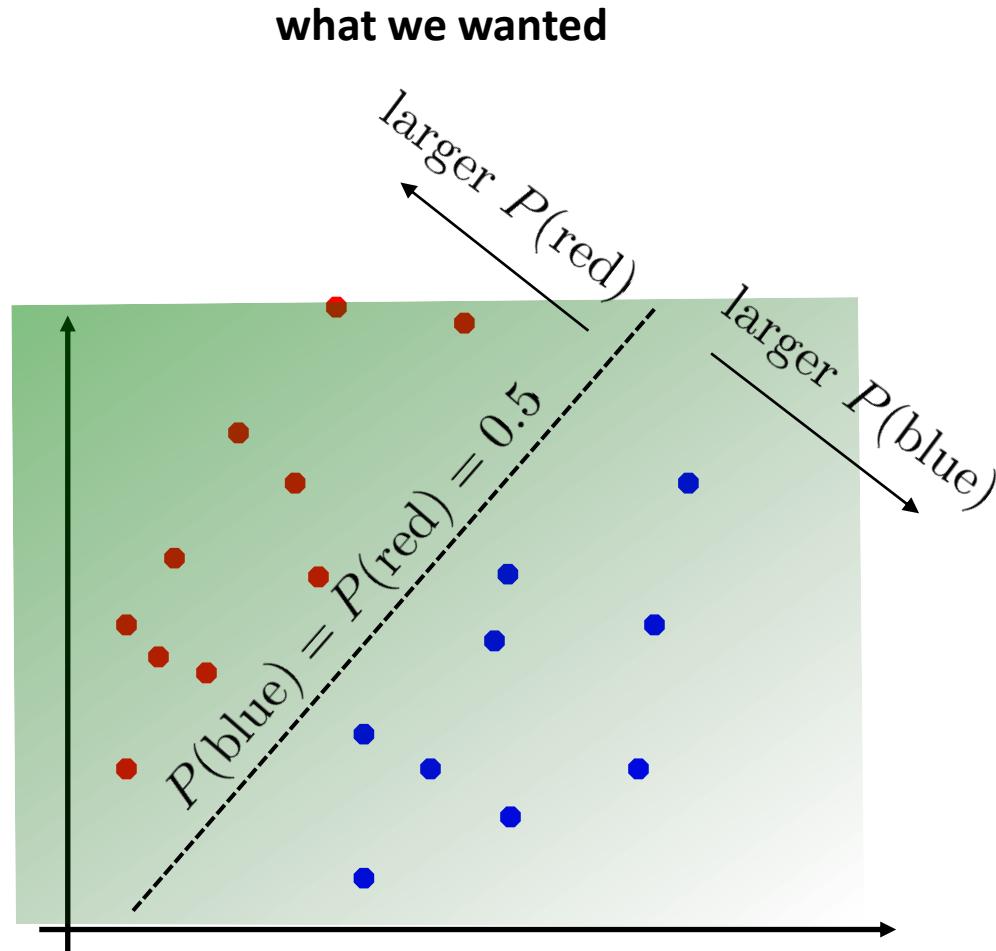
$$\lambda = \frac{1}{2\sigma^2} \quad \text{"hyperparameter"}$$

$$\mathcal{L}(\theta) = \left(\sum_{i=1}^N \|f_\theta(x_i) - y_i\|^2 \right) + \lambda \|\theta\|^2$$

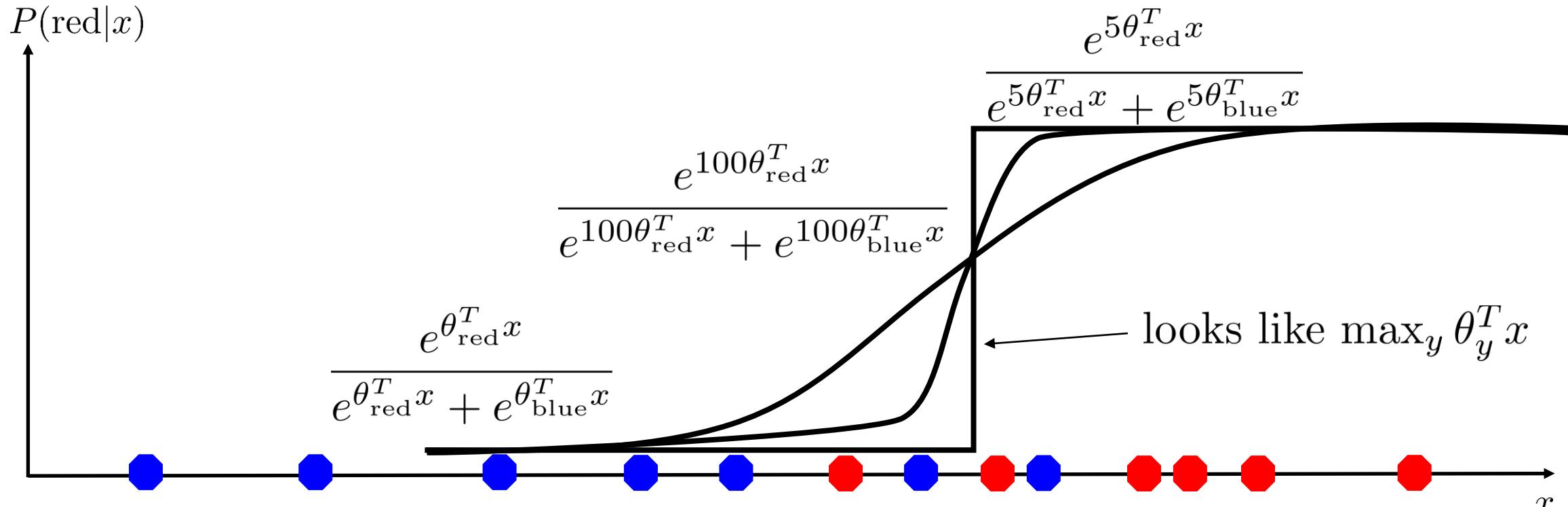
if $\Sigma_\theta(x) = \mathbf{I}$

(but we don't care, we'll just select it directly)

Example: regularized logistic regression



Example: regularized logistic regression

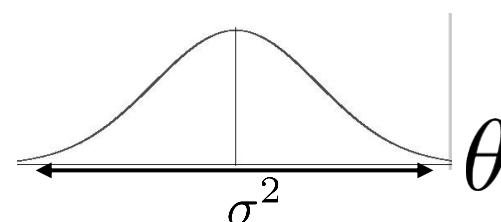


$$P(\text{red}|x) = \frac{e^{\theta_{\text{red}}^T x}}{e^{\theta_{\text{red}}^T x} + e^{\theta_{\text{blue}}^T x}}$$

Larger θ values \Rightarrow sharper probabilities

Solution: prefer **smaller** θ !

Simple idea: $p(\theta) = \mathcal{N}(0, \sigma^2)$

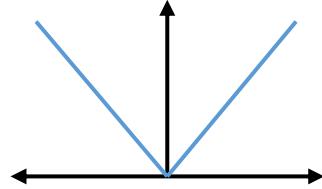


Example: regularized logistic regression

$$\mathcal{L}(\theta) = - \left(\sum_{i=1}^N \log p(y_i|x_i, \theta) \right) + \lambda \|\theta\|^2$$

Other examples of regularizers (we'll discuss some of these later):

$$\lambda \sum_{i=1}^D |\theta_i|$$



creates a preference for zeroing out dimensions!

Dropout: a special type of regularizer for neural networks

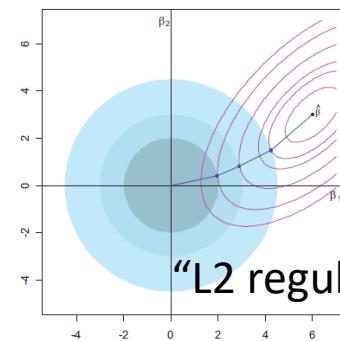
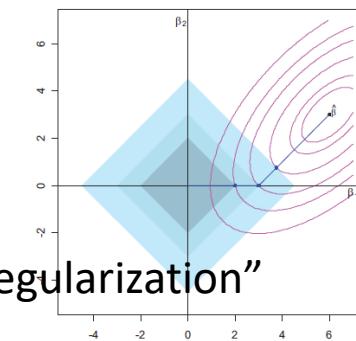
Gradient penalty: a special type of regularizer for GANs

...lots of other choices

same prior, but now for a **classification** problem
this is sometimes called **weight decay**

$$\lambda \sum_{i=1}^D |\theta_i|$$

$$\lambda \sum_{i=1}^D \theta_i^2$$



Other perspectives

Regularization: something we add to the loss function to reduce variance

Bayesian perspective: the regularizer is prior knowledge about parameters

Numerical perspective: the regularizer makes underdetermined problems well-determined

Optimization perspective: the regularizer makes the loss landscape easier to search

paradoxically, regularizers can sometimes reduce **underfitting** if it was due to poor optimization!
especially common with GANs

In machine learning, any “heuristic” term added to the loss
that doesn’t depend on data is generally called a regularizer

$$= -\lambda \|\theta\|^2 + \text{const}$$

“hyperparameter”

Regularizers introduce **hyperparameters** that we have to
select in order for them to work well

Training sets and test sets

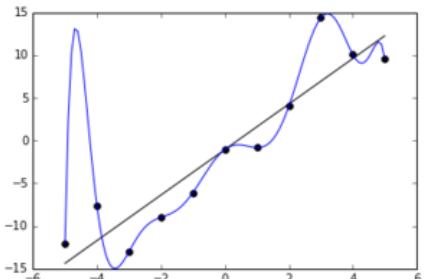
Some questions...

How do we **know** if we are overfitting or underfitting?

How do we select which algorithm to use?

How do we select **hyperparameters**?

One idea: choose whatever makes the **loss** low



$$\mathcal{L}(\theta) = 0$$

Can't diagnose **overfitting** by looking at the training loss!

The machine learning workflow

the dataset



training set

use this for training

$$\mathcal{L}(\theta, \mathcal{D}_{\text{train}})$$

validation set

reserve this for...

- ...selecting hyperparameters
- ...adding/removing features
- ...tweaking your model class

$$\mathcal{L}(\theta, \mathcal{D}_{\text{val}})$$

1. Train θ with $\mathcal{L}(\theta, \mathcal{D}_{\text{train}})$
if $\mathcal{L}(\theta, \mathcal{D}_{\text{train}})$ not low enough
you are **underfitting**
decrease regularization
improve your optimizer
2. Look at $\mathcal{L}(\theta, \mathcal{D}_{\text{val}})$
if $\mathcal{L}(\theta, \mathcal{D}_{\text{val}}) >> \mathcal{L}(\theta, \mathcal{D}_{\text{train}})$
you are **overfitting**
increase regularization

The machine learning workflow

the dataset



used to select...

θ (via optimization)

optimizer hyperparameters (e.g., learning rate α)

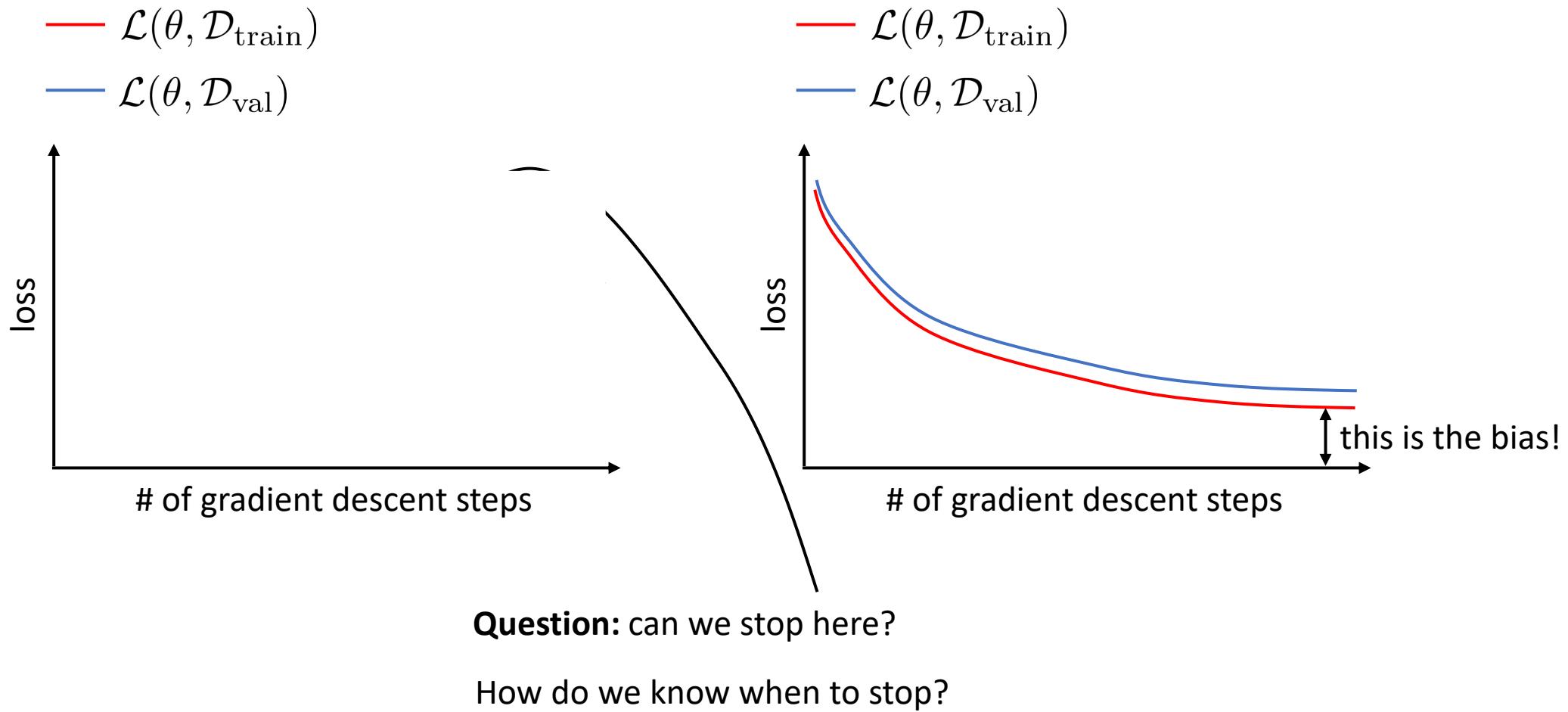
used to select...

model class (e.g., logistic regression vs. neural net)

regularizer hyperparameters

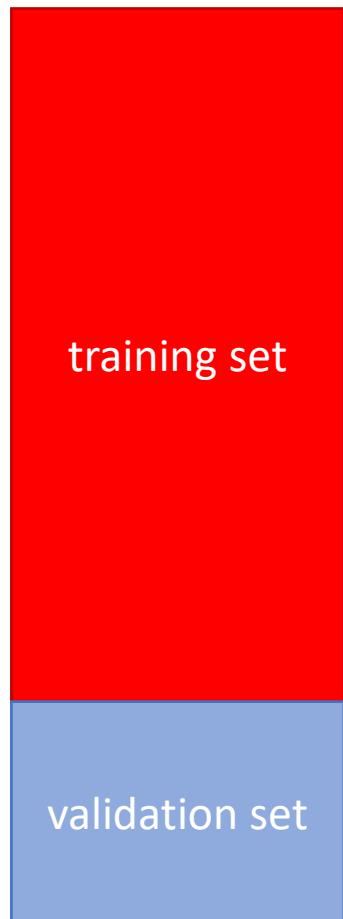
which features to use

Learning curves



The final exam

the dataset



1. Train θ with $\mathcal{L}(\theta, \mathcal{D}_{\text{train}})$
if $\mathcal{L}(\theta, \mathcal{D}_{\text{train}})$ not low enough
you are **underfitting**
decrease regularization
improve your optimizer
2. Look at $\mathcal{L}(\theta, \mathcal{D}_{\text{val}})$
if $\mathcal{L}(\theta, \mathcal{D}_{\text{val}}) >> \mathcal{L}(\theta, \mathcal{D}_{\text{train}})$
you are **overfitting**
increase regularization

We followed the recipe, **now what?**

How good is our final classifier?

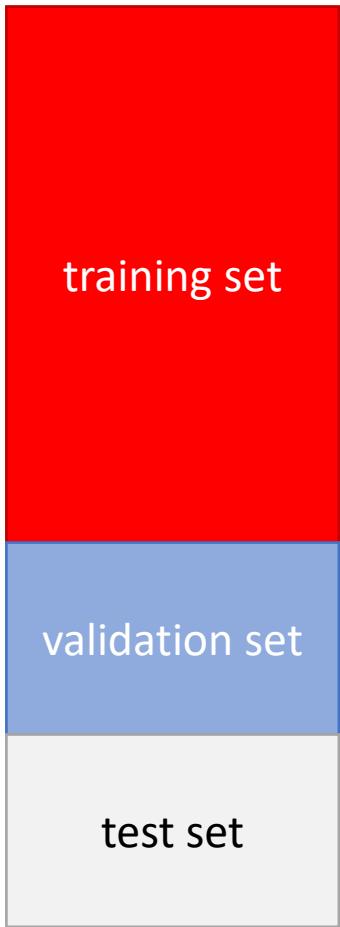
$$\mathcal{L}(\theta, \mathcal{D}_{\text{val}})?$$

That's **no good** – we already used the validation set to pick hyperparameters!

What if we reserve **another** set for a **final exam** (a kind of... validation validation set!)

The machine learning workflow

the dataset



used to select...

θ (via optimization)

optimizer hyperparameters (e.g., learning rate α)

used to select...

model class (e.g., logistic regression vs. neural net)

regularizer hyperparameters

which features to use

Used **only** to report final performance

Summary and takeaways

➤ Where do errors come from?

- Variance: too much capacity, not enough information in the data to find the right parameters
- Bias: too little capacity, not enough representational power to represent the true function
- Error = Variance + Bias²
- Overfitting = too much variance
- Underfitting = too much bias

➤ How can we trade off bias and variance?

- Select your model class carefully
- Select your features carefully
- Regularization: stuff we add to the loss to reduce variance

➤ How do we select hyperparameters?

- Training/validation split
- Training set is for optimization (learning)
- Validation set is for selecting hyperparameters
- Test set is for **reporting final results and nothing else!**



Optimization

Designing, Visualizing and Understanding Deep Neural Networks

CS W182/282A

Instructor: Sergey Levine
UC Berkeley



How does gradient descent work?

The loss “landscape”

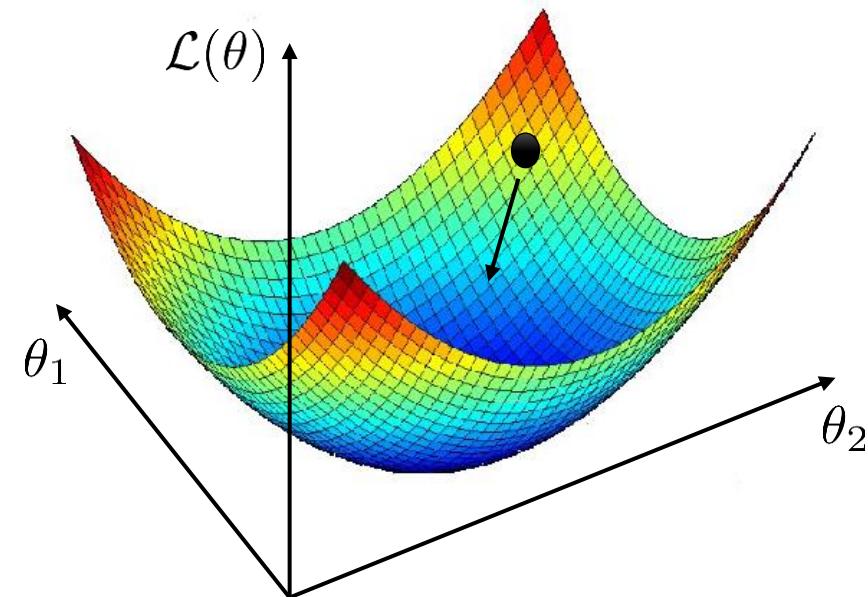
$$\theta^* \leftarrow \arg \min_{\theta} - \underbrace{\sum_i \log p_{\theta}(y_i | x_i)}_{\mathcal{L}(\theta)}$$

let's say θ is 2D

An algorithm:

1. Find a *direction* v where $\mathcal{L}(\theta)$ decreases
2. $\theta \leftarrow \theta + \alpha v$

some small constant
called “learning rate” or
“step size”

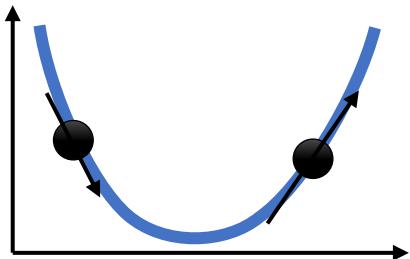


Gradient descent

An algorithm:

- 1. Find a *direction* v where $\mathcal{L}(\theta)$ decreases
- 2. $\theta \leftarrow \theta + \alpha v$

Which way does $\mathcal{L}(\theta)$ decrease?



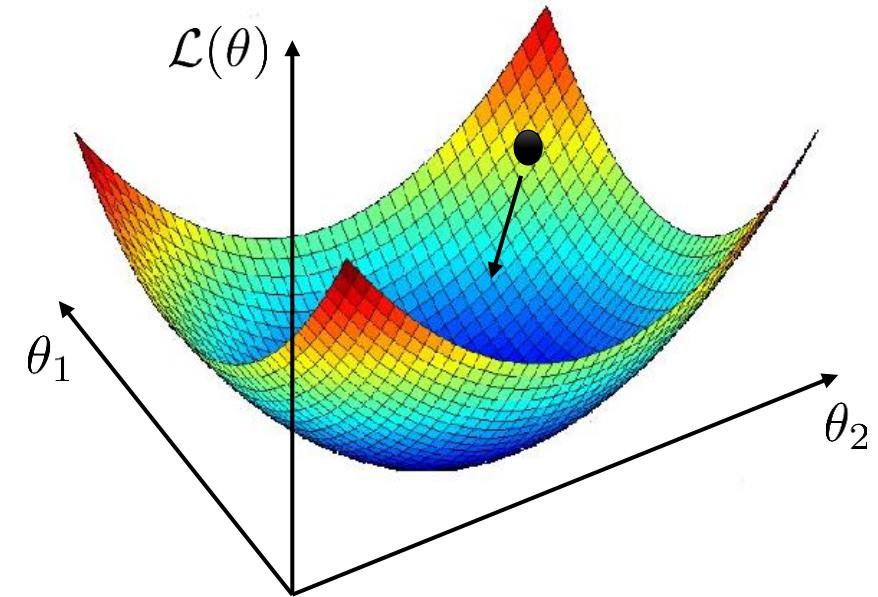
negative slope = go to the right

positive slope = go to the left

in general:

for each dimension, go in the direction
opposite the slope **along that dimension**

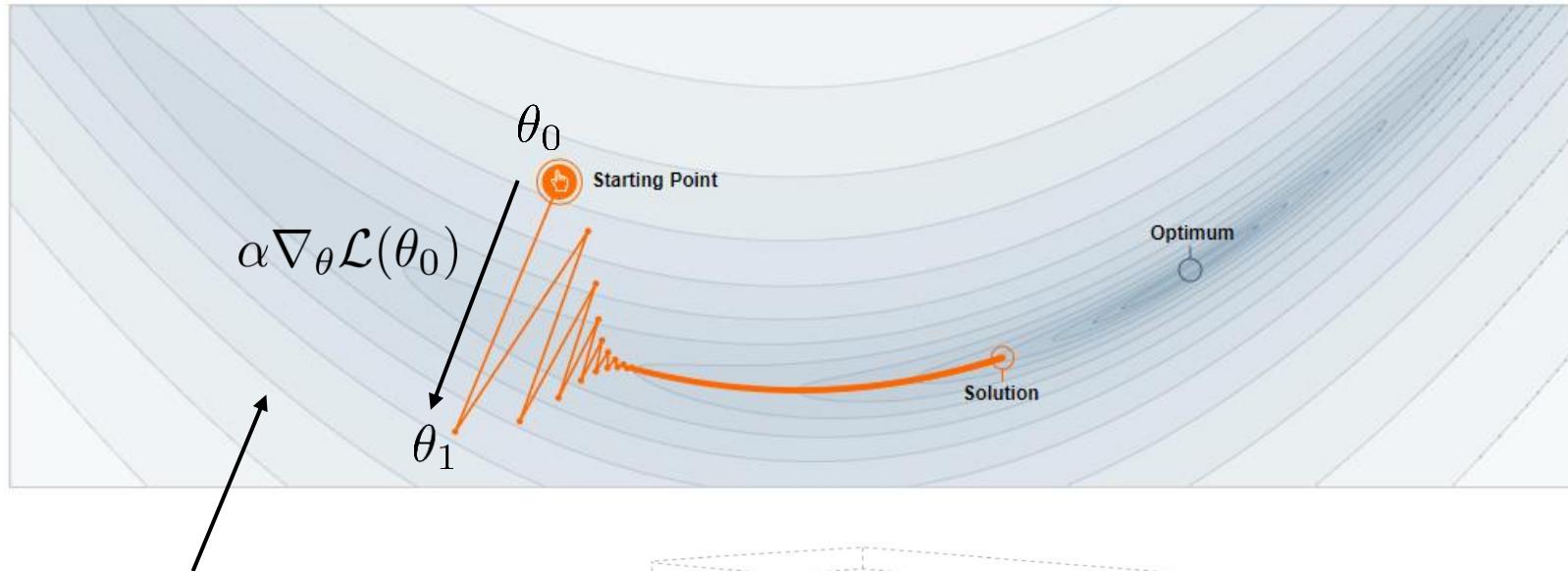
$$v_1 = -\frac{d\mathcal{L}(\theta)}{d\theta_1} \quad v_2 = -\frac{d\mathcal{L}(\theta)}{d\theta_2} \quad \text{etc.}$$



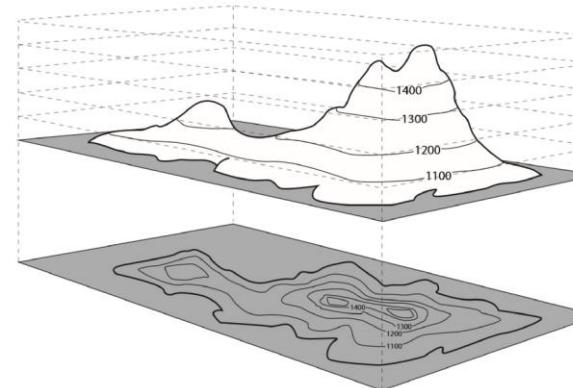
gradient:

$$\nabla_{\theta} \mathcal{L}(\theta) = \begin{pmatrix} \frac{d\mathcal{L}(\theta)}{d\theta_1} \\ \frac{d\mathcal{L}(\theta)}{d\theta_2} \\ \vdots \\ \frac{d\mathcal{L}(\theta)}{d\theta_n} \end{pmatrix}$$

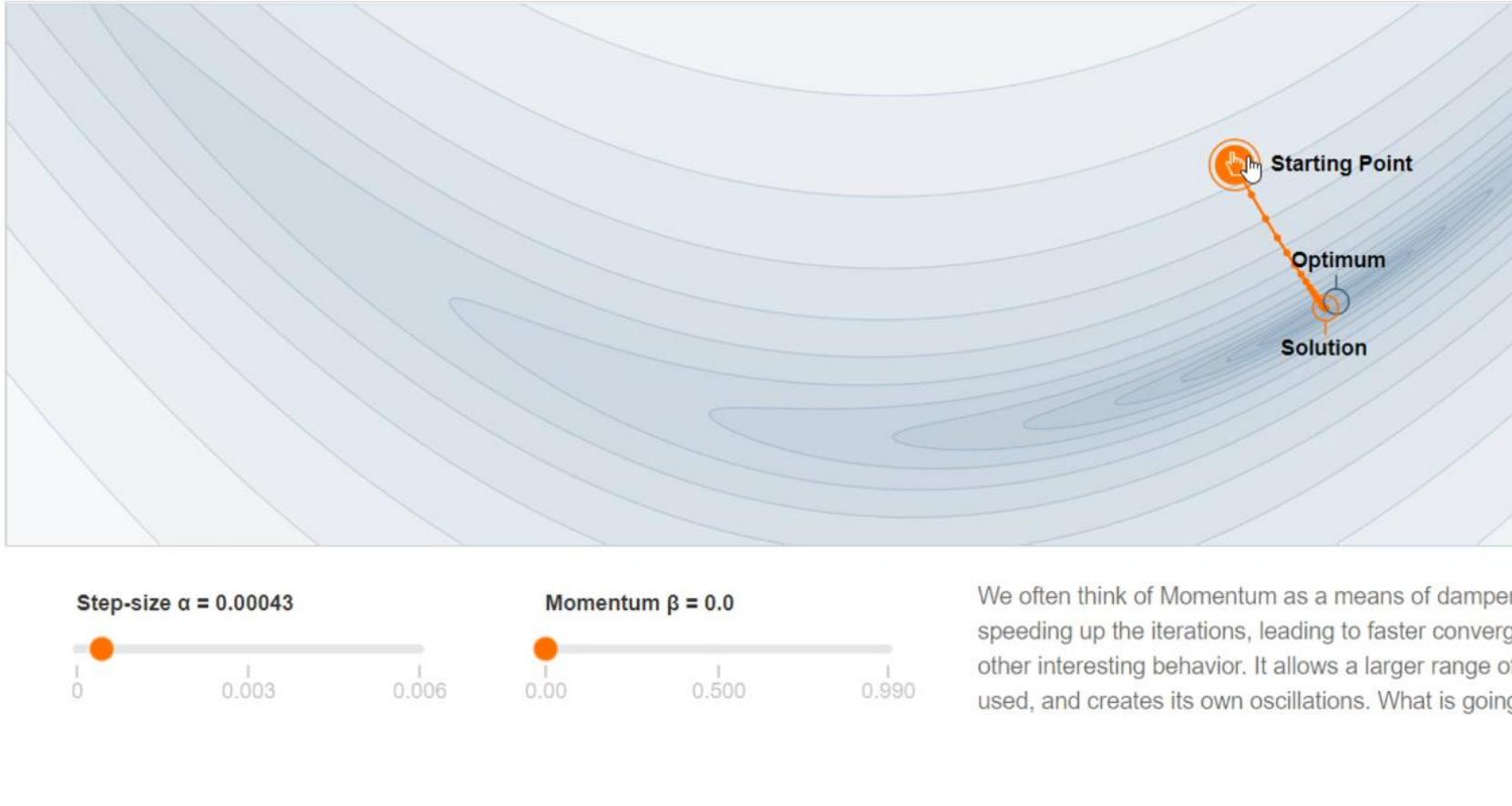
Visualizing gradient descent



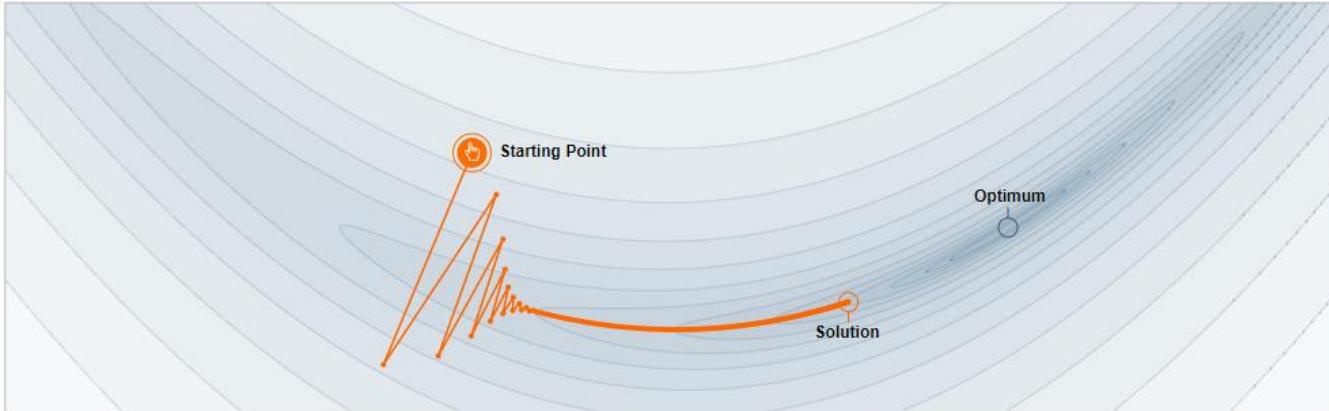
level set contours
for all θ values along a line
 $\mathcal{L}(\theta)$ takes on the same value



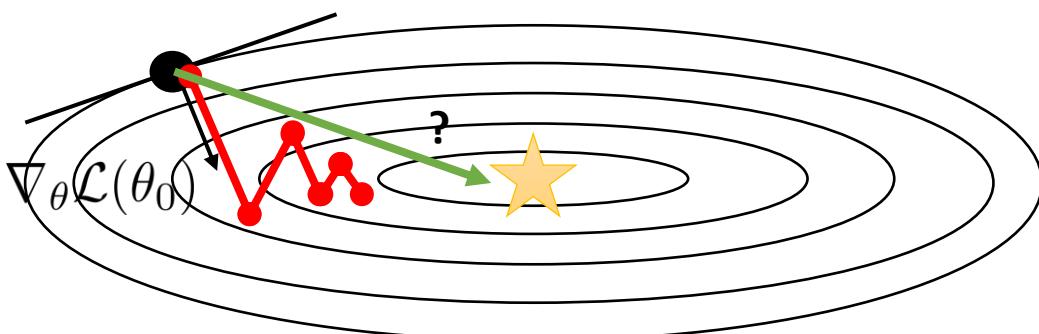
Demo time!



What's going on?

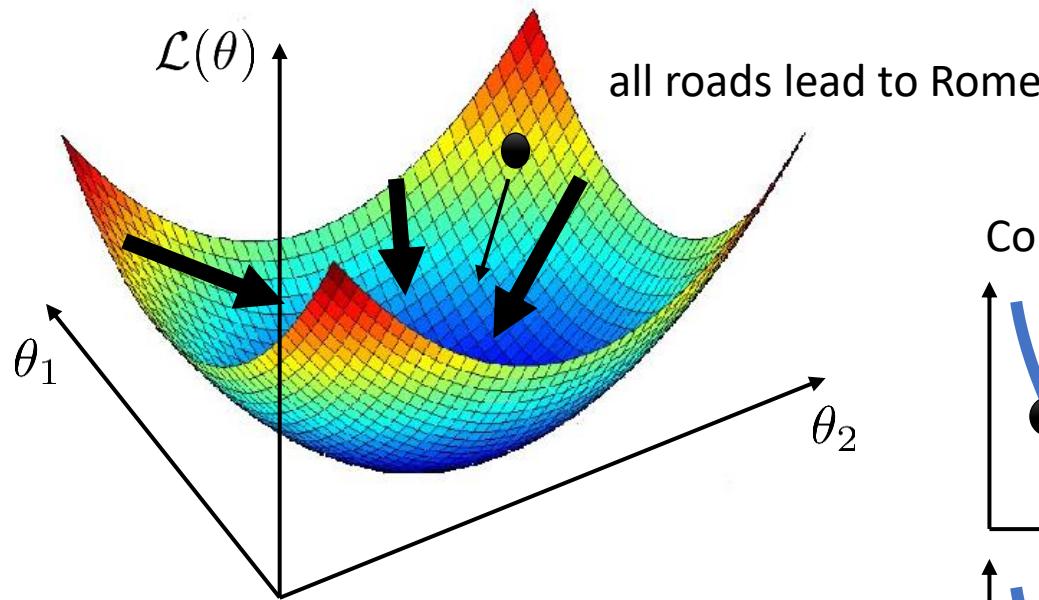


we don't always move toward the optimum!



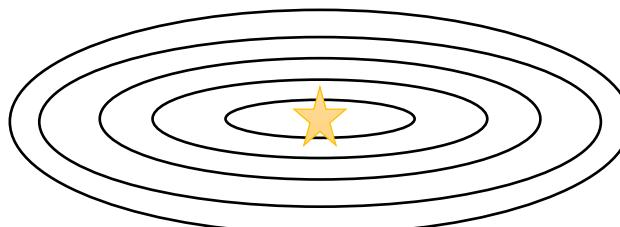
the steepest direction is not always best!
more on this later...

The loss surface



This is a *very nice* loss surface Why?

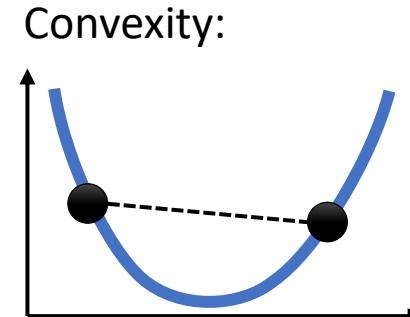
Is our loss actually this nice?



Logistic regression:

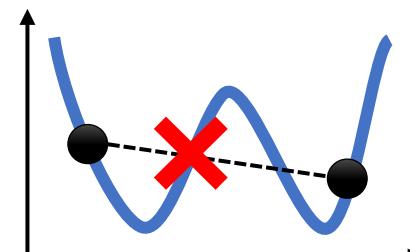
$$p_{\theta}(y = i|x) = \frac{\exp(x^T \theta_i)}{\sum_{j=1}^m \exp(x^T \theta_j)}$$

Negative likelihood loss for **logistic regression** is guaranteed to be **convex**
(this is **not** an obvious or trivial statement!)

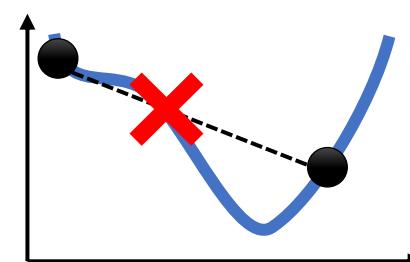


a function is convex if a line segment between any two points lies entirely “above” the graph

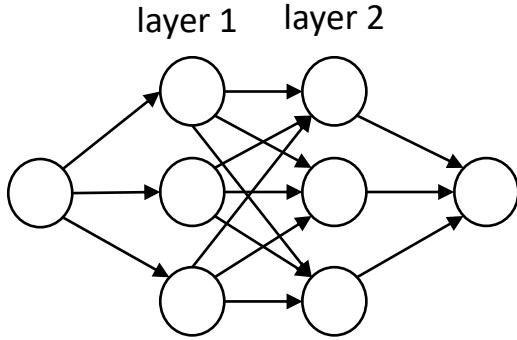
convex functions are “nice” in the sense that simple algorithms like gradient descent have strong guarantees



the **doesn't** mean that gradient descent works well for all convex functions!



The loss surface... ...of a neural network



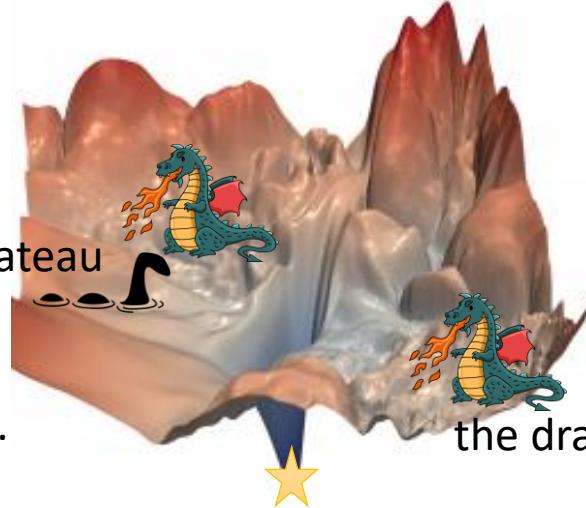
pretty hard to visualize, because neural networks have very large numbers of parameters

but let's give it a try!

Visualizing the Loss Landscape of Neural Nets

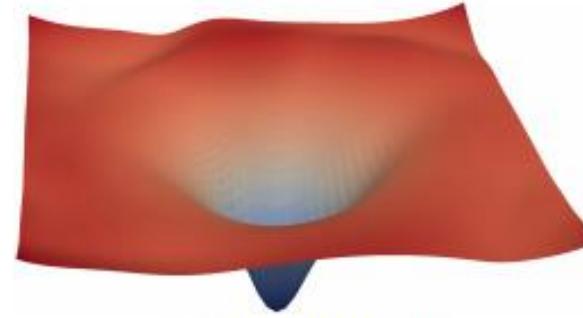
Hao Li¹, Zheng Xu¹, Gavin Taylor², Christoph Studer³, Tom Goldstein¹
¹University of Maryland, College Park ²United States Naval Academy ³Cornell University
`{haoli, xuzh, tomg}@cs.umd.edu, taylor@usna.edu, studer@cornell.edu`

the monster of the plateau

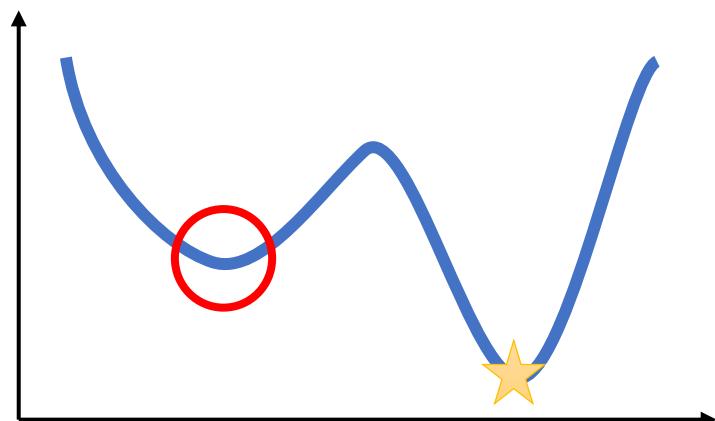


the dragon of local optima

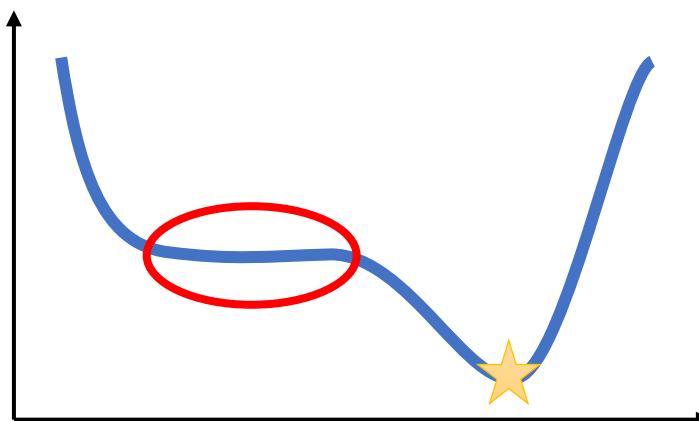
...though some networks are better!



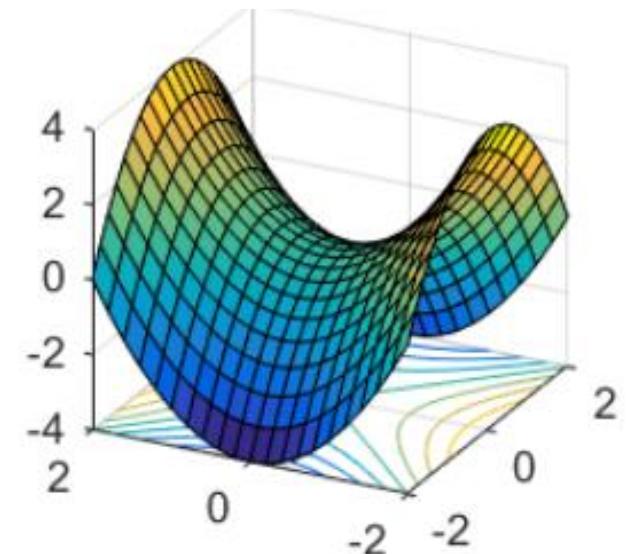
The geography of a loss landscape



the local optimum

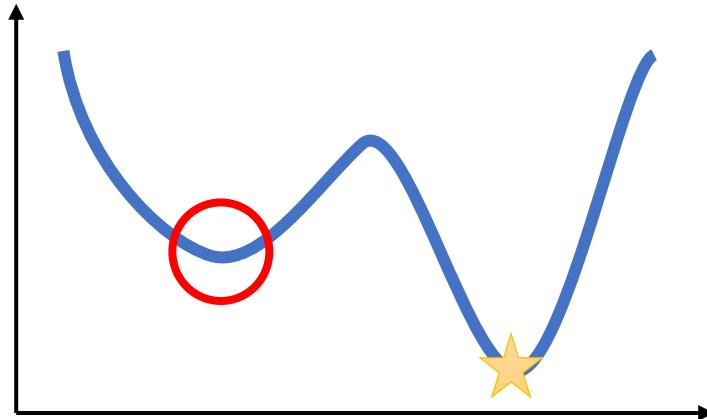


the plateau



the saddle point

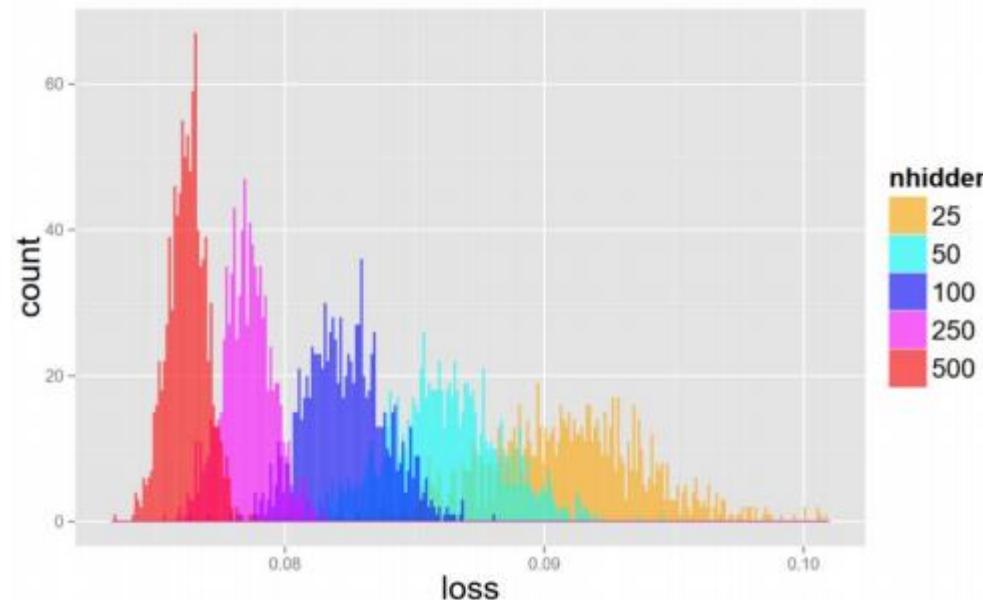
Local optima



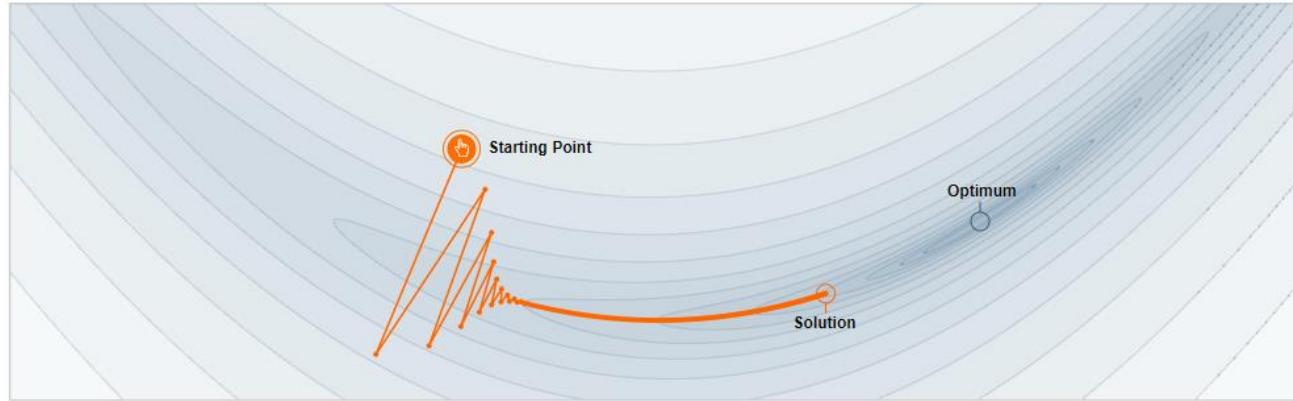
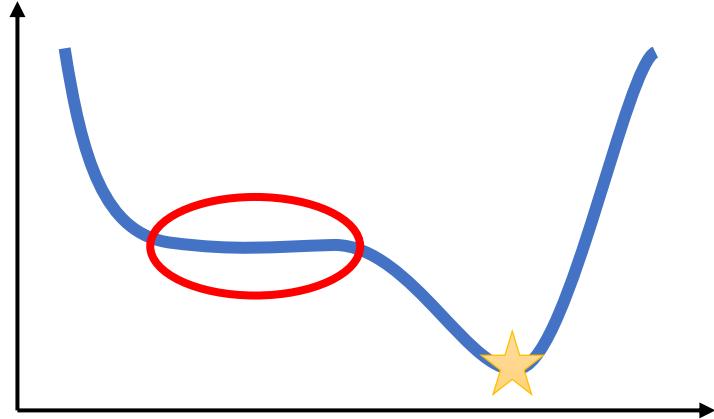
a bit surprisingly, this becomes less of an issue as the number of parameters increases!

for big networks, local optima exist, but tend to be not much worse than global optima

the most obvious issue with non-convex loss landscapes
one of the big reasons people used to worry about neural networks!
very scary in principle, since gradient descent could converge to a solution that is arbitrarily worse than the global optimum!



Plateaus

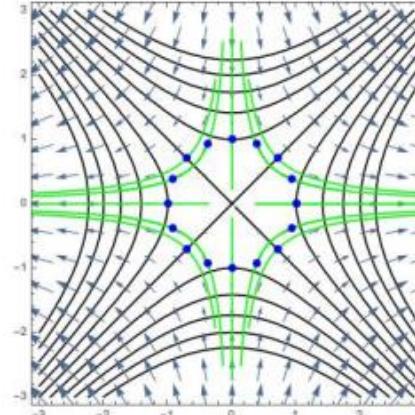
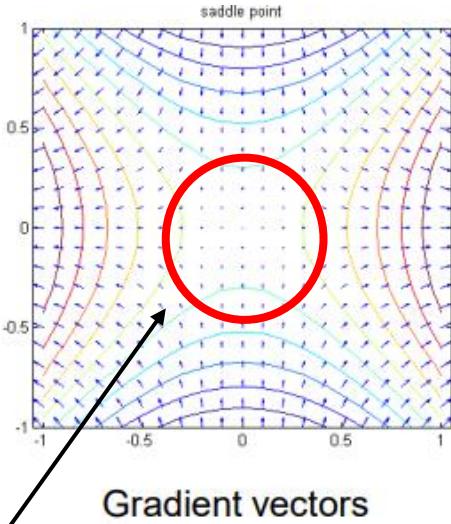
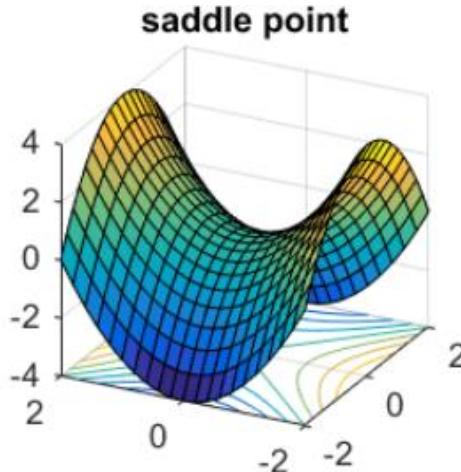


Can't just choose tiny learning rates to prevent oscillation!

Need learning rates to be large enough not to get stuck in a plateau

We'll learn about **momentum**, which really helps with this

Saddle points

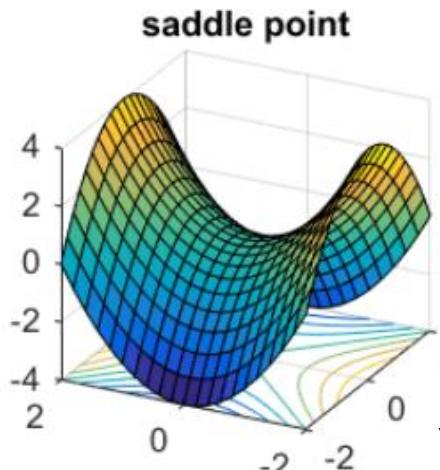


the gradient here is very small
it takes a long time to get out of saddle points

this seems like a **very** special structure,
does it really happen **that** often?

Yes! in fact, most critical points in neural
net loss landscapes are saddle points

Saddle points



Critical points:

any point where $\nabla_{\theta}\mathcal{L}(\theta) = 0$

is it a **maximum**, **minimum**, or **saddle**?

Hessian matrix:

$$\begin{bmatrix} \frac{d^2\mathcal{L}}{d\theta_1 d\theta_1} & \frac{d^2\mathcal{L}}{d\theta_1 d\theta_2} & \frac{d^2\mathcal{L}}{d\theta_1 d\theta_3} \\ \frac{d^2\mathcal{L}}{d\theta_2 d\theta_1} & \frac{d^2\mathcal{L}}{d\theta_2 d\theta_2} & \frac{d^2\mathcal{L}}{d\theta_2 d\theta_3} \\ \frac{d^2\mathcal{L}}{d\theta_3 d\theta_1} & \frac{d^2\mathcal{L}}{d\theta_3 d\theta_2} & \frac{d^2\mathcal{L}}{d\theta_3 d\theta_3} \end{bmatrix}$$

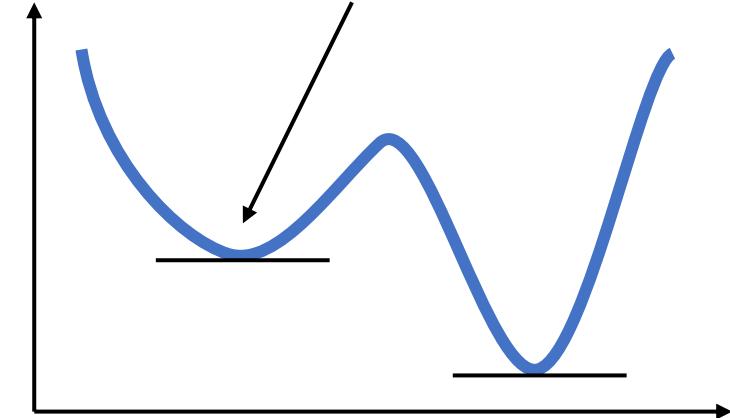
In higher dimensions:

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

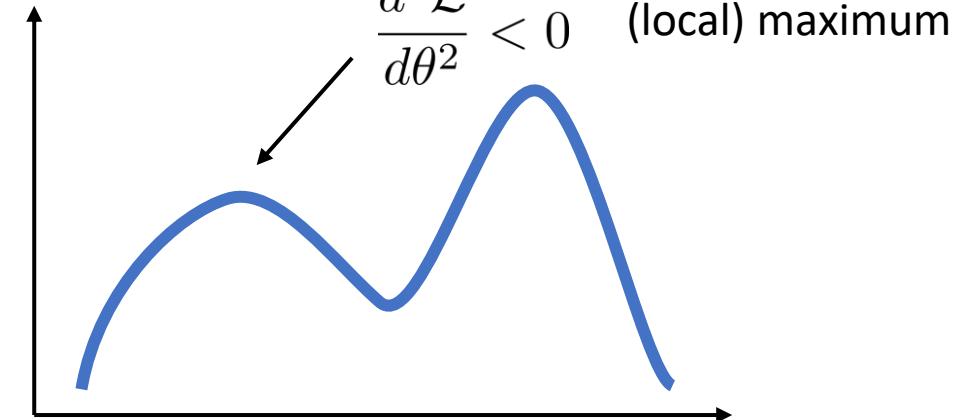
only maximum or minimum if all diagonal entries are positive or negative!

how often is that the case?

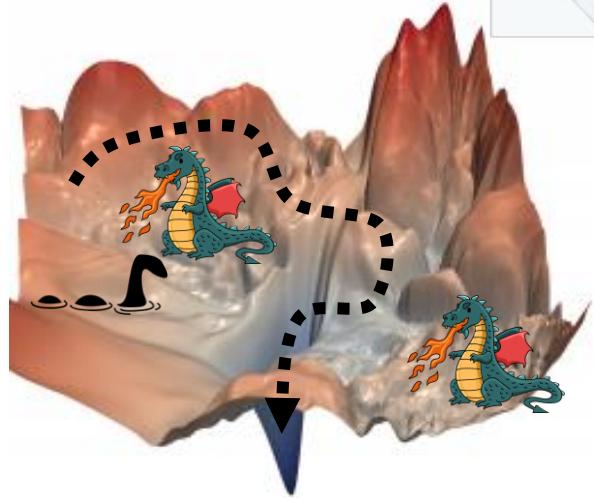
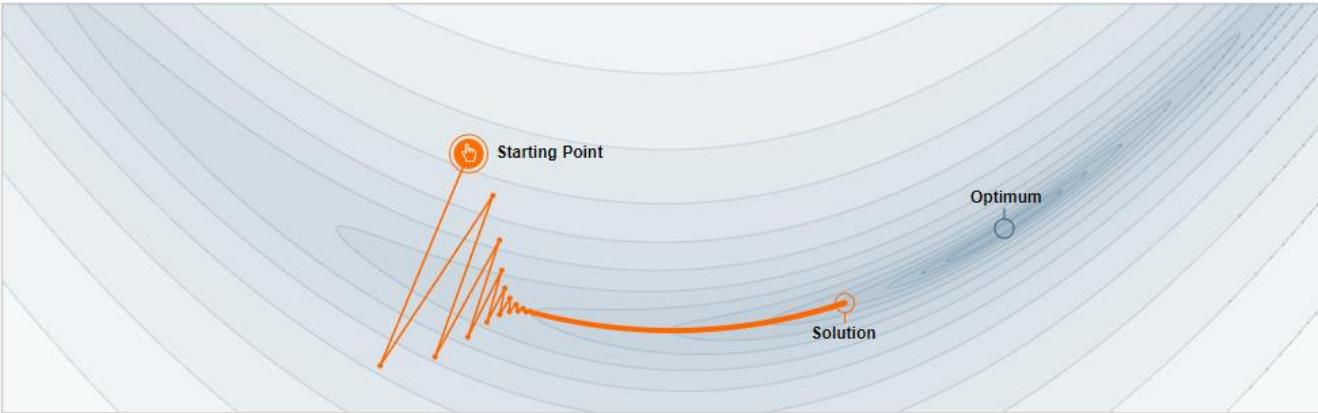
$\frac{d^2\mathcal{L}}{d\theta^2} > 0$ (local) minimum



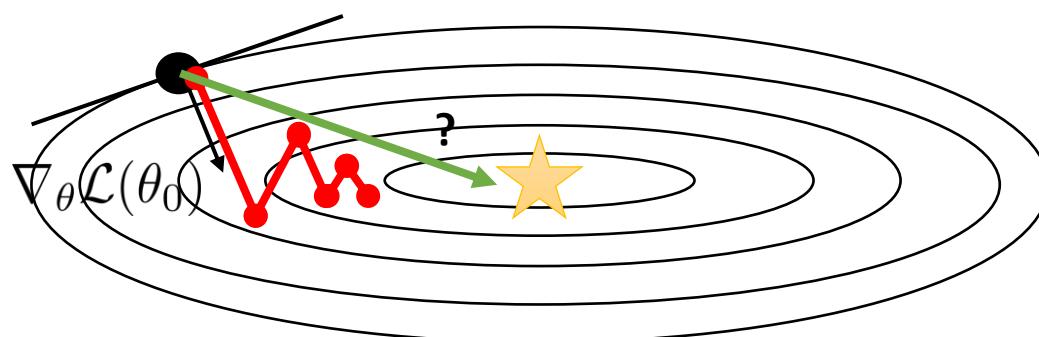
$\frac{d^2\mathcal{L}}{d\theta^2} < 0$ (local) maximum



Which way do we go?



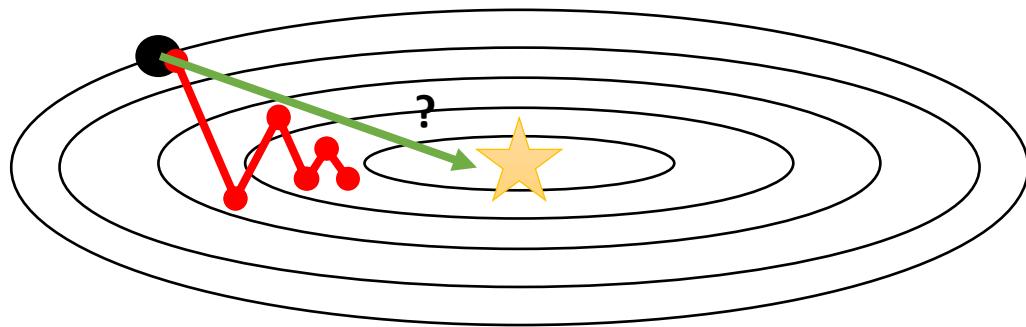
we don't always move toward the optimum!



the steepest direction is not always best!
more on this later...

Improvement directions

A better direction...



can we find this direction?

yes, with Newton's method!

we won't use Newton's method (can't afford it)

but it's an "ideal" to aspire to

Newton's method

Taylor expansion:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2$$

multivariate case:

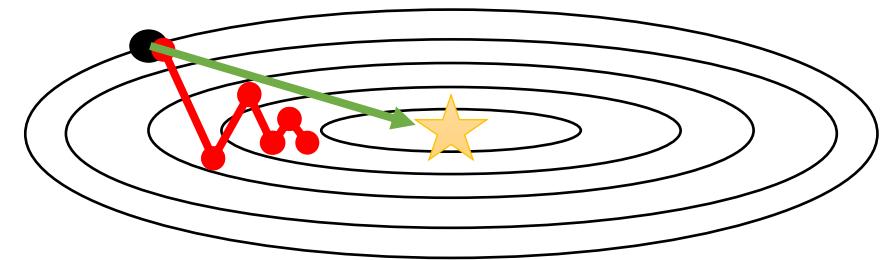
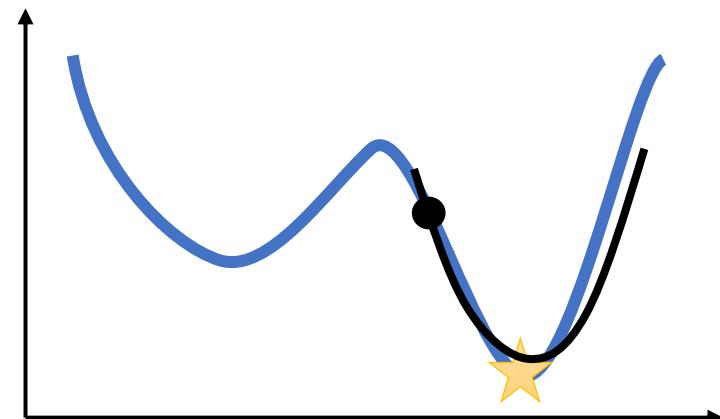
$$\mathcal{L}(\theta) \approx \mathcal{L}(\theta_0) + \underbrace{\nabla_{\theta}\mathcal{L}(\theta_0)}_{\text{gradient}}(\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)^T \underbrace{\nabla_{\theta}^2\mathcal{L}(\theta_0)}_{\text{Hessian}}(\theta - \theta_0)$$

$$\begin{bmatrix} \frac{d^2\mathcal{L}}{d\theta_1 d\theta_1} & \frac{d^2\mathcal{L}}{d\theta_1 d\theta_2} & \frac{d^2\mathcal{L}}{d\theta_1 d\theta_3} \\ \frac{d^2\mathcal{L}}{d\theta_2 d\theta_1} & \frac{d^2\mathcal{L}}{d\theta_2 d\theta_2} & \frac{d^2\mathcal{L}}{d\theta_2 d\theta_3} \\ \frac{d^2\mathcal{L}}{d\theta_3 d\theta_1} & \frac{d^2\mathcal{L}}{d\theta_3 d\theta_2} & \frac{d^2\mathcal{L}}{d\theta_3 d\theta_3} \end{bmatrix}$$

can optimize this analytically!

set derivative to zero and solve:

$$\theta^* \leftarrow \theta_0 - (\nabla_{\theta}^2\mathcal{L}(\theta_0))^{-1}\nabla_{\theta}\mathcal{L}(\theta_0)$$



Tractable acceleration

Why is Newton's method not a viable way to improve neural network optimization?

gradient descent: $\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_{\theta} \mathcal{L}(\theta_k)$ runtime? $\mathcal{O}(n)$

Hessian

$$\begin{bmatrix} \frac{d^2 \mathcal{L}}{d\theta_1 d\theta_1} & \frac{d^2 \mathcal{L}}{d\theta_1 d\theta_2} & \frac{d^2 \mathcal{L}}{d\theta_1 d\theta_3} \\ \frac{d^2 \mathcal{L}}{d\theta_2 d\theta_1} & \frac{d^2 \mathcal{L}}{d\theta_2 d\theta_2} & \frac{d^2 \mathcal{L}}{d\theta_2 d\theta_3} \\ \frac{d^2 \mathcal{L}}{d\theta_3 d\theta_1} & \frac{d^2 \mathcal{L}}{d\theta_3 d\theta_2} & \frac{d^2 \mathcal{L}}{d\theta_3 d\theta_3} \end{bmatrix} \quad \begin{matrix} \uparrow n \\ \downarrow n \end{matrix}$$

$$\theta^* \leftarrow \theta_0 - (\nabla_{\theta}^2 \mathcal{L}(\theta_0))^{-1} \nabla_{\theta} \mathcal{L}(\theta_0)$$

runtime?
 $\mathcal{O}(n^3)$

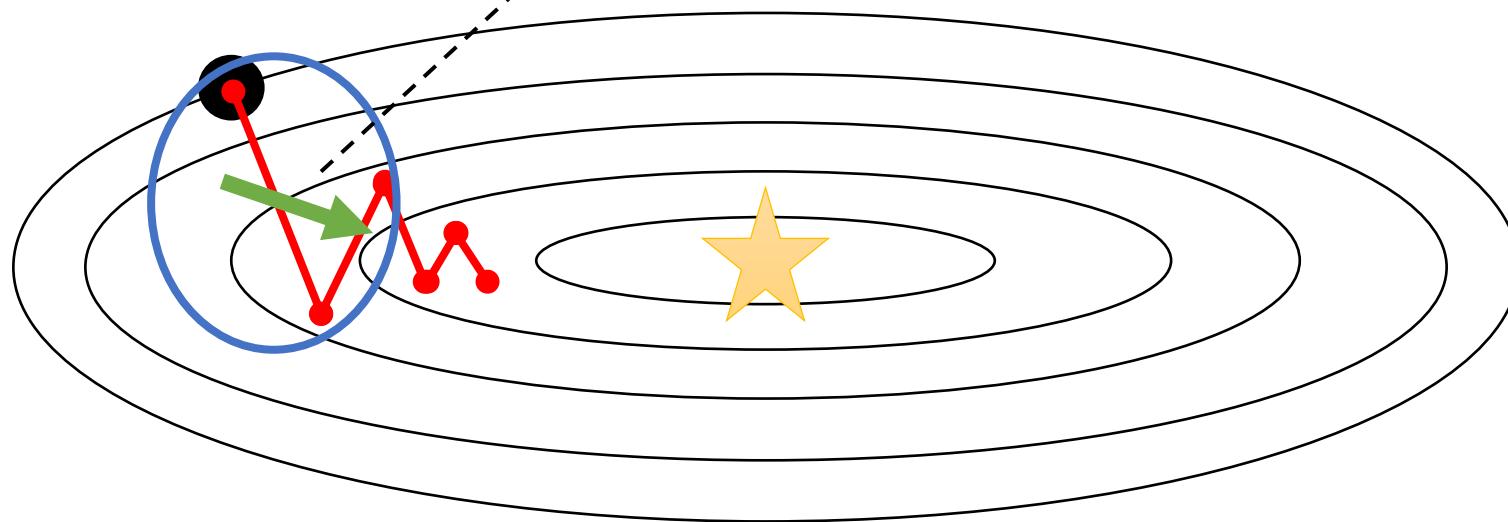
$$\nabla_{\theta} \mathcal{L}(\theta) = \begin{pmatrix} \frac{d\mathcal{L}(\theta)}{d\theta_1} \\ \frac{d\mathcal{L}(\theta)}{d\theta_2} \\ \vdots \\ \frac{d\mathcal{L}(\theta)}{d\theta_n} \end{pmatrix} \quad \begin{matrix} \uparrow n \\ \downarrow n \end{matrix}$$

if using naïve approach, though fancy methods can be much faster if they avoid forming the Hessian explicitly

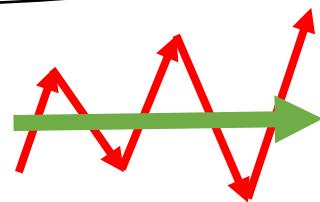
because of this, we would really prefer methods that don't require second derivatives, but somehow "accelerate" gradient descent instead

Momentum

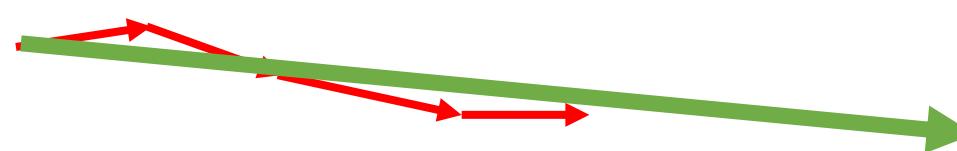
averaging together successive gradients
seems to yield a much better direction!



Intuition: if successive gradient steps point in **different** directions, we should **cancel off** the directions that disagree



if successive gradient steps point in **similar** directions, we
should **go faster** in that direction



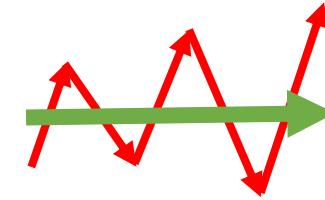
Momentum

update rule:

$$\theta_{k+1} = \theta_k - \alpha g_k$$

before: $g_k = \nabla_{\theta} \mathcal{L}(\theta_k)$

now: $g_k = \nabla_{\theta} \mathcal{L}(\theta_k) + \underbrace{\mu g_{k-1}}$



“blend in” previous direction

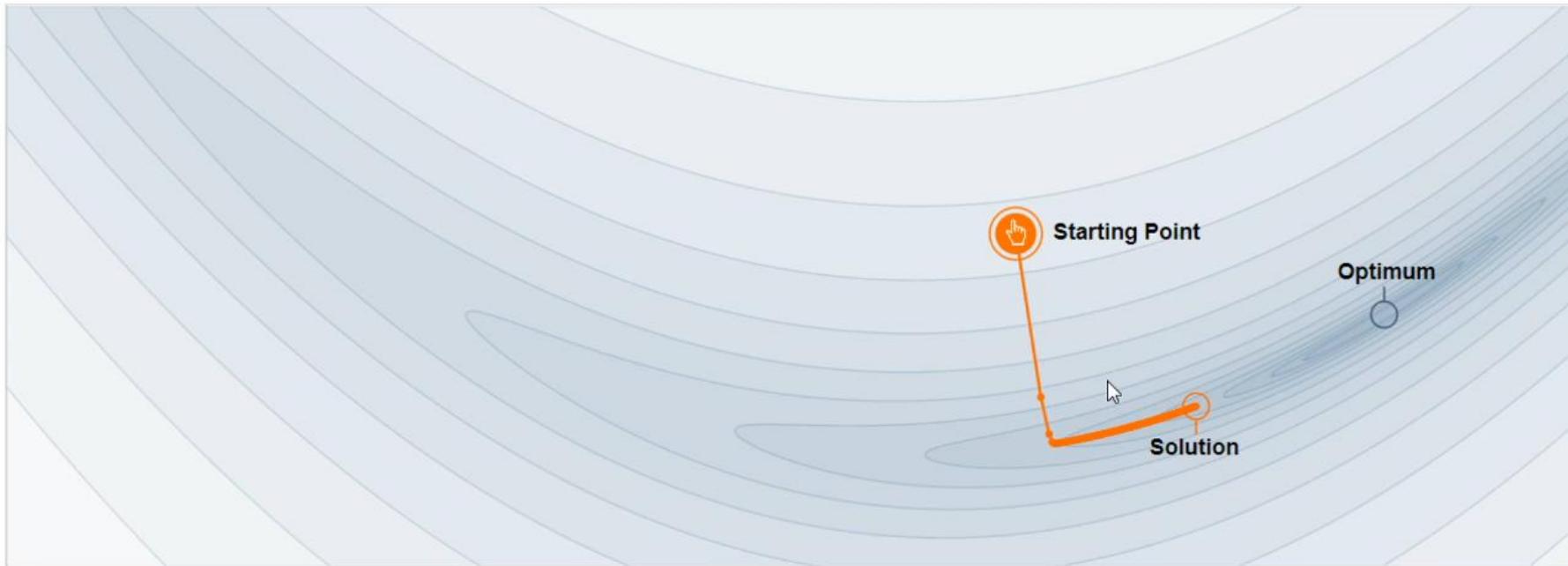
this is a **very** simple update rule

in practice, it brings some of the benefits of
Newton’s method, at virtually no cost

this kind of momentum method has few guarantees

a closely related idea is “Nesterov accelerated gradient,”
which **does** carry very appealing guarantees (in practice we
usually just momentum)

Momentum Demo



Step-size $\alpha = 0.0021$

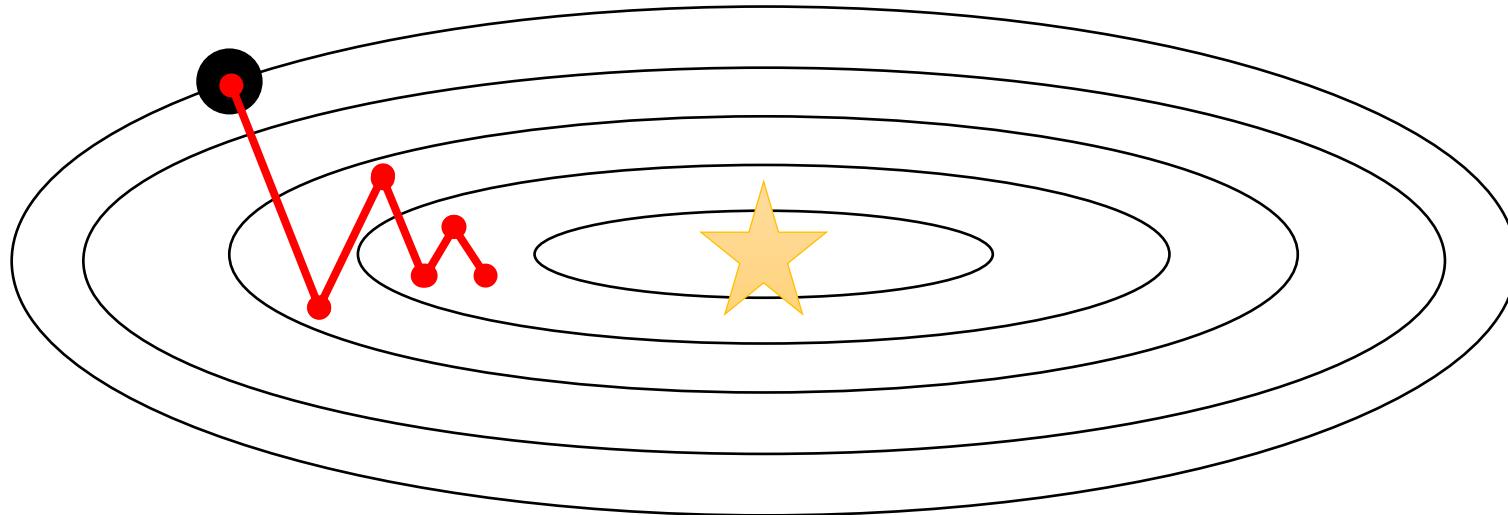


Momentum $\beta = 0.0$



We often think of Momentum as a means of dampening the oscillations of gradient descent, speeding up the iterations, leading to faster convergence. However, it can also lead to other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

Gradient scale



Intuition: the **sign** of the gradient tells us which way to go along each dimension, but the magnitude is not so great

Even worse: overall magnitude of the gradient can change drastically over the course of optimization, making learning rates hard to tune

Idea: “normalize” out the magnitude of the gradient **along each dimension**

$$\nabla_{\theta} \mathcal{L}(\theta) = \begin{bmatrix} \frac{d\mathcal{L}(\theta)}{d\theta_1} \\ \frac{d\mathcal{L}(\theta)}{d\theta_2} \\ \vdots \\ \frac{d\mathcal{L}(\theta)}{d\theta_n} \end{bmatrix}$$

$$\mathcal{L}(\theta) = \|f_{\theta}(x) - y\|^2$$

$$\nabla_{\theta} \mathcal{L}(\theta) = (f_{\theta}(x) - y)^T \underbrace{\frac{df}{d\theta}}_{\text{huge when far from optimum}}$$

Algorithm: RMSProp

Estimate per-dimension magnitude (running average):

$$s_k \leftarrow \beta s_{k-1} + (1 - \beta)(\nabla_{\theta}\mathcal{L}(\theta_k))^2 \quad \text{this is } \textit{roughly} \text{ the squared length of each dimension}$$

$$\theta_{k+1} = \theta_k - \alpha \frac{\nabla_{\theta}\mathcal{L}(\theta_k)}{\sqrt{s_k}} \quad \text{each dimension is divided by its magnitude}$$

Algorithm: AdaGrad

Estimate per-dimension cumulative magnitude:

$$s_k \leftarrow s_{k-1} + (\nabla_{\theta} \mathcal{L}(\theta_k))^2$$

$$\theta_{k+1} = \theta_k - \alpha \frac{\nabla_{\theta} \mathcal{L}(\theta_k)}{\sqrt{s_k}}$$

RMSProp:

$$s_k \leftarrow \beta s_{k-1} + (1 - \beta)(\nabla_{\theta} \mathcal{L}(\theta_k))^2$$

How does AdaGrad and RMSProp compare?

AdaGrad has some appealing guarantees for **convex** problems

Learning rate effectively “decreases” over time, which is good for convex problems

But this only works if we find the optimum quickly before the rate decays too much

RMSProp tends to be much better for deep learning (and most non-convex problems)

Algorithm: Adam

Basic idea: combine **momentum** and **RMSProp**

$$m_k = (1 - \beta_1) \nabla_{\theta} \mathcal{L}(\theta_k) + \beta_1 m_{k-1}$$

first moment estimate (“momentum-like”)

$$v_k = (1 - \beta_2)(\nabla_{\theta} \mathcal{L}(\theta_k))^2 + \beta_2 v_{k-1}$$

second moment estimate

$$\hat{m}_k = \frac{m_k}{1 - \beta_1^k}$$

why?

$$\begin{aligned} m_0 &= 0 \\ v_0 &= 0 \end{aligned}$$

so early on these values will be small, and this correction “blows them up” a bit for small k

$$\hat{v}_k = \frac{v_k}{1 - \beta_2^k}$$

good default settings:

$$\alpha = 0.001$$

$$\beta_1 = 0.9$$

$$\beta_2 = 0.999$$

$$\theta_{k+1} = \theta_k - \alpha \frac{\hat{m}_k}{\sqrt{\hat{v}_k} + \epsilon}$$

$$\epsilon = 10^{-8}$$

small number to prevent division by zero

Stochastic optimization

Why is gradient descent expensive?

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{i=1}^N \log p_\theta(y_i|x_i) \approx -E_{p_{\text{data}}(x,y)}[\log p_\theta(y_i|x_i)]$$

requires summing over **all** datapoints in the dataset

$$\approx -\frac{1}{B} \sum_{j=1}^B \log p_\theta(y_{i_j}|x_{i_j})$$

could simply use **fewer** samples, and still have a correct (unbiased) estimator

$$B \ll N$$



ILSVRC (ImageNet), 2009: 1.5 million images

Stochastic gradient descent

with minibatches

1. Sample $\mathcal{B} \subset \mathcal{D}$ draw **B** datapoints at random from dataset of size **N**
2. Estimate $g_k \leftarrow -\nabla_{\theta} \frac{1}{B} \sum_{i=1}^B \log p(y_i|x_i, \theta) \approx \nabla_{\theta} \mathcal{L}(\theta)$ (where sum is over elements in \mathcal{B})
3. $\theta_{k+1} \leftarrow \theta_k - \alpha g_k$ can also use momentum, ADAM, etc.

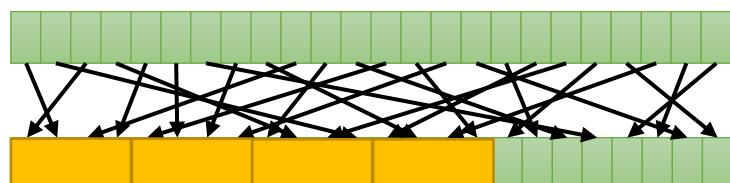
each iteration samples a different **minibatch**

Stochastic gradient descent **in practice**:

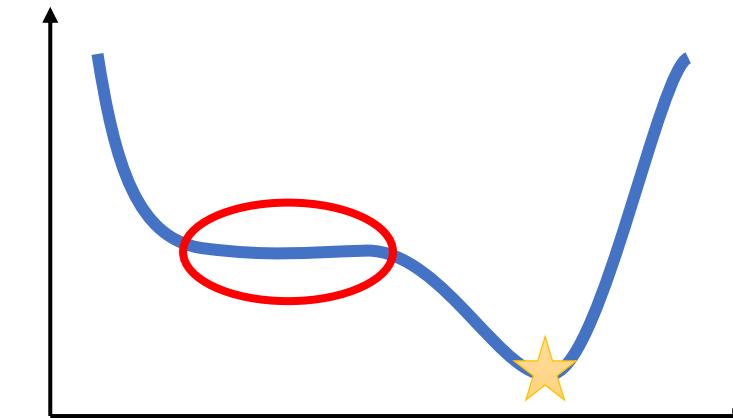
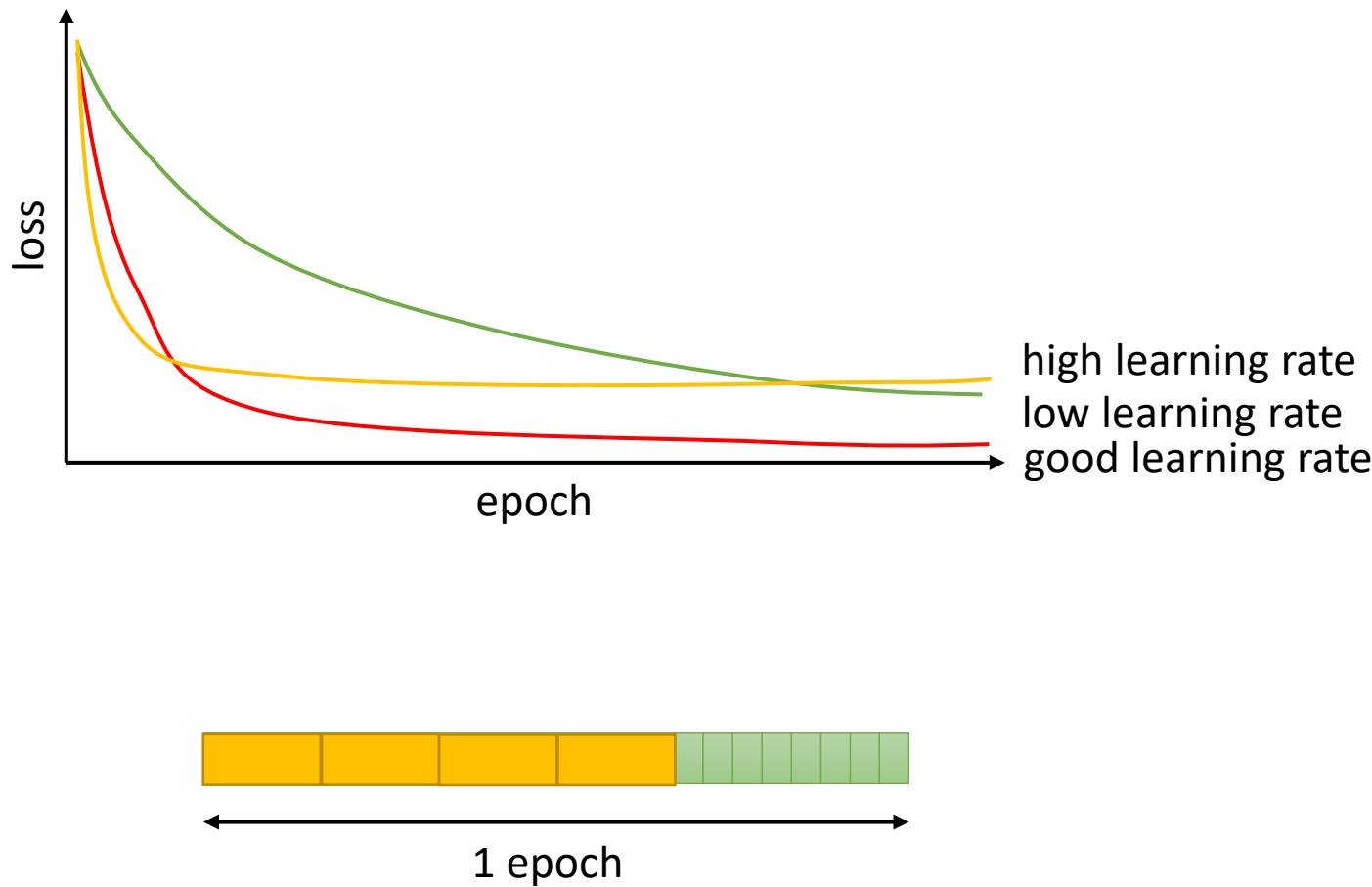
sampling randomly is slow due to random memory access

instead, shuffle the dataset (like a deck of cards...) once, in advance

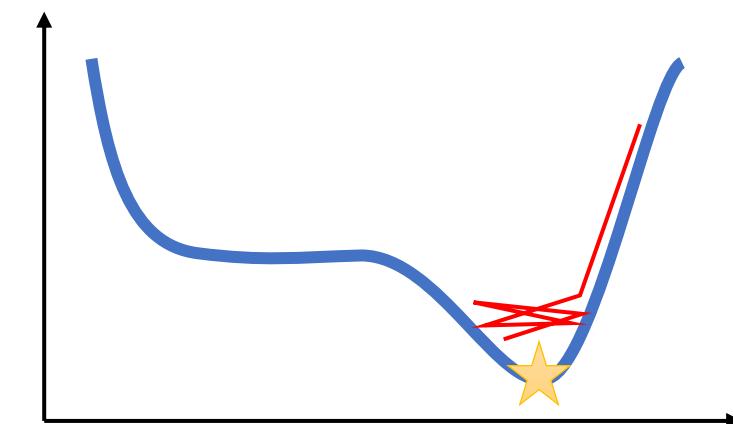
then just construct batches out of consecutive groups of **B** datapoints



Learning rates

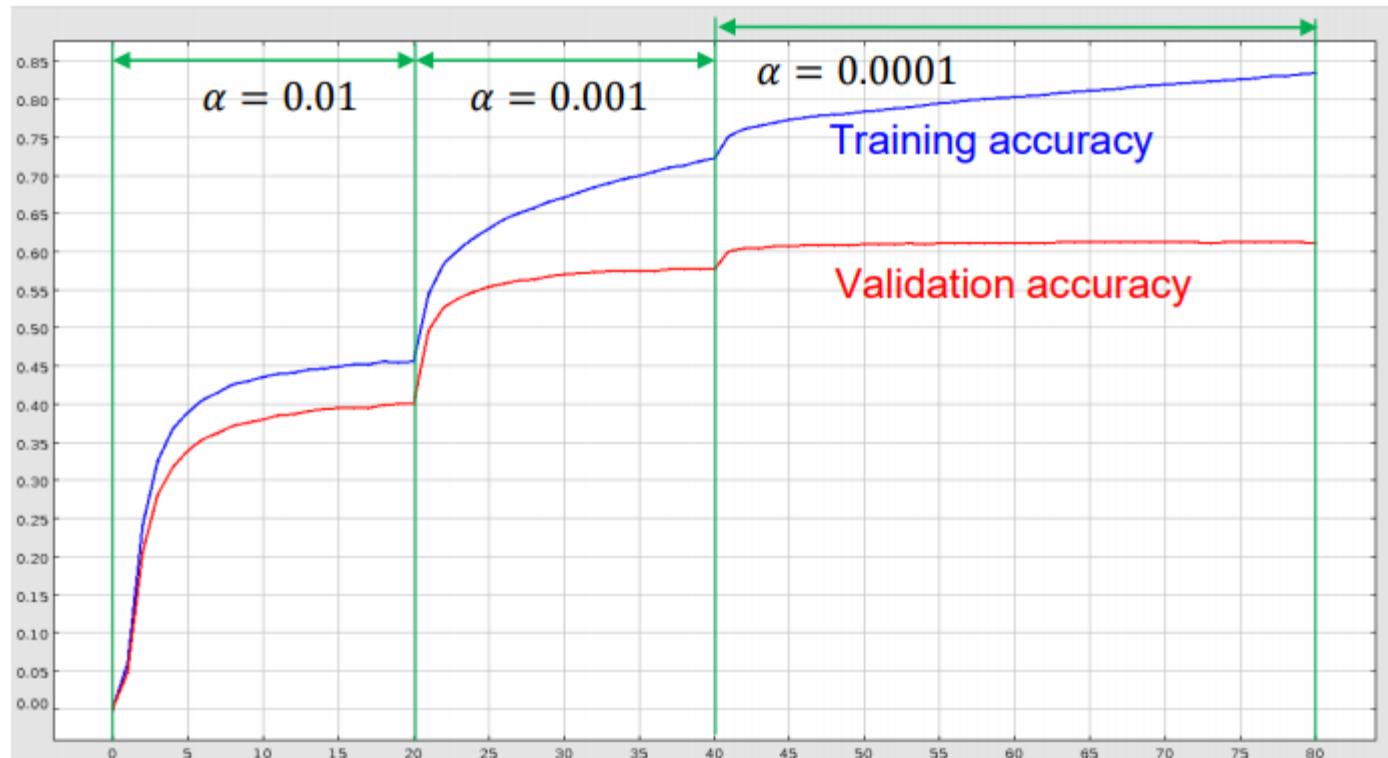


Low learning rates **can** result in convergence to worse values!
This is a bit counter-intuitive



Decaying learning rates

AlexNet trained on ImageNet



Learning rate decay schedules usually needed for best performance with SGD (+momentum)

Often not needed with ADAM

Opinions differ, some people think SGD + momentum is better than ADAM if you want the very best performance (but ADAM is easier to tune)

Tuning (stochastic) gradient descent

Hyperparameters:

batch size: B larger batches = less noisy gradients, usually “safer” but more expensive

learning rate: α best to use the biggest rate that still works, decay over time

momentum: μ Adam parameters: β_1, β_2

0.99 is good keep the defaults (usually)

What to tune hyperparameters on?

Technically we want to tune this on the **training** loss, since it is a parameter of the optimization

Often tuned on **validation** loss

Relationship between stochastic gradient and regularization is complex – some people consider it to be a good regularizer!
(this suggests we should use validation loss)

Backpropagation

Designing, Visualizing and Understanding Deep Neural Networks

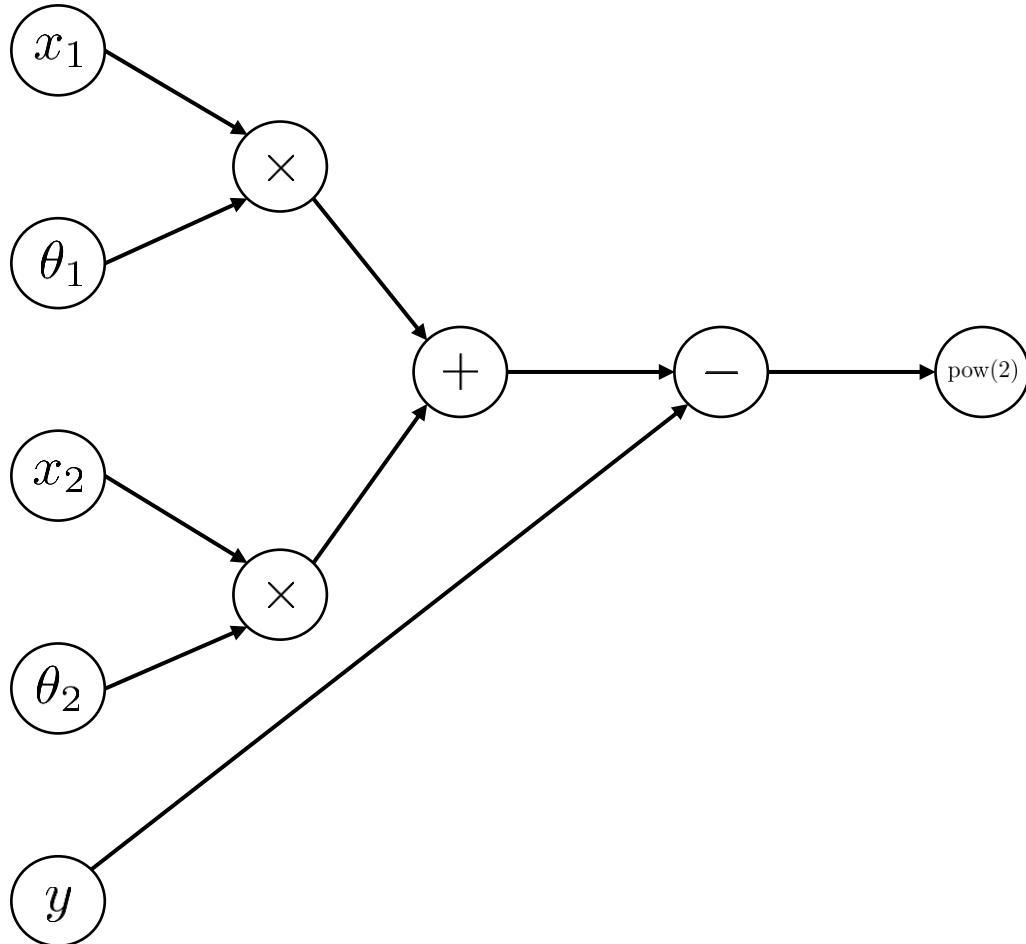
CS W182/282A

Instructor: Sergey Levine
UC Berkeley



Neural networks

Drawing computation graphs



what **expression** does this compute?
equivalently, what **program** does this correspond to?

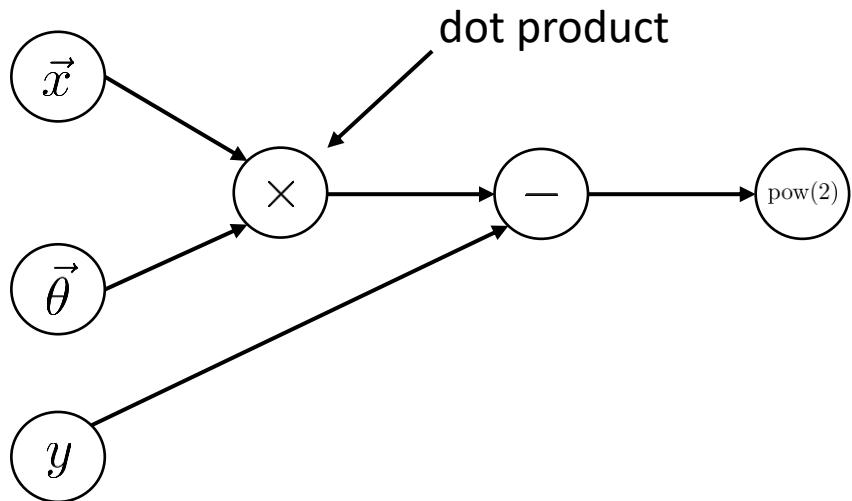
$$\| (x_1 \theta_1 + x_2 \theta_2) - y \|^2$$

this is a **MSE loss** with a **linear regression** model

neural networks are computation graphs
if we design **generic tools** for computation graphs, we
can train **many kinds** of neural networks

Drawing computation graphs

a simpler way to draw the same thing:



what **expression** does this compute?
equivalently, what **program** does this correspond to?

$$\|(\vec{x}\theta + y) - \vec{y}\|^2$$

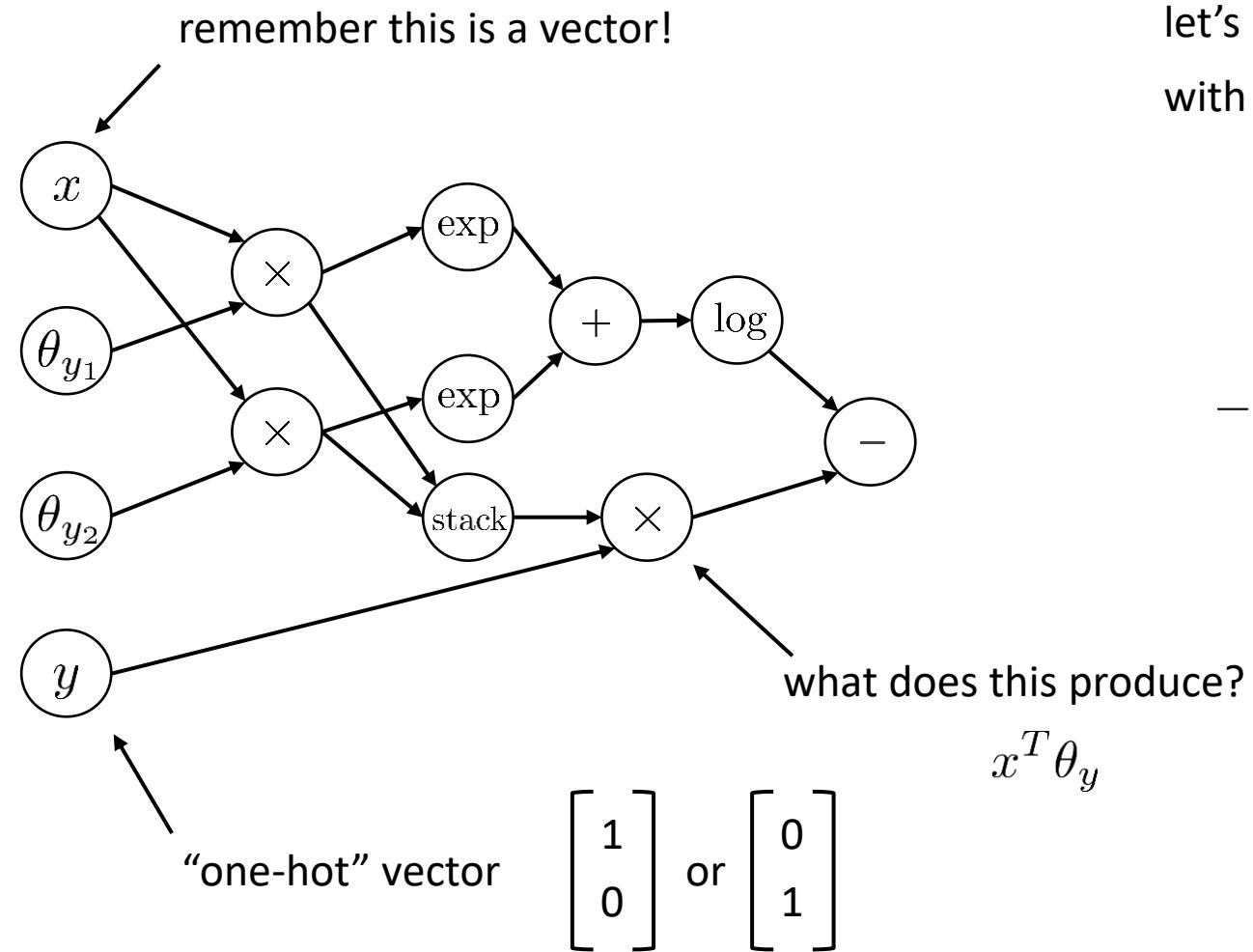
this is a **MSE loss** with a **linear regression** model

neural networks are computation graphs

if we design **generic tools** for computation graphs, we can train **many kinds** of neural networks

I'll drop the $\vec{\cdot}$ decorator from now on...

Logistic regression



let's draw the computation graph for **logistic regression**
with the negative log-likelihood loss

$$p_\theta(y|x) = \frac{\exp(x^T \theta_y)}{\sum_{y'} \exp(x^T \theta_{y'})}$$

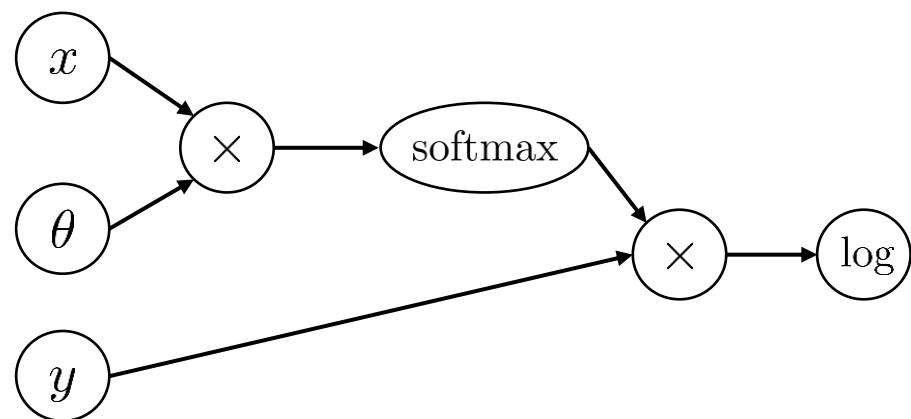
$$-\log p_\theta(y|x) = -x^T \theta_y + \log \sum_{y'} \exp(x^T \theta_{y'})$$

Logistic regression

a simpler way to draw the same thing:

$$f_\theta(x) = \begin{bmatrix} x^T \theta_{y_1} \\ x^T \theta_{y_2} \\ \vdots \\ x^T \theta_{y_m} \end{bmatrix} \quad f_\theta(x) = \theta x$$

matrix



$$p_\theta(y|x) = \frac{\exp(x^T \theta_y)}{\sum_{y'} \exp(x^T \theta_{y'})}$$

$$-\log p_\theta(y|x) = -x^T \theta_y + \log \sum_{y'} \exp(x^T \theta_{y'})$$

$$\begin{bmatrix} \theta_{y_1} \\ \theta_{y_2} \\ \theta_{y_3} \end{bmatrix} \times \begin{bmatrix} x \end{bmatrix} = \begin{bmatrix} x^T \theta_{y_1} \\ x^T \theta_{y_2} \\ \vdots \\ x^T \theta_{y_m} \end{bmatrix}$$

$$p_\theta(y = i|x) = \text{softmax}(f_\theta(x))[i] = \frac{\exp(f_{\theta,i}(x))}{\sum_{j=1}^m \exp(f_{\theta,j}(x))}$$

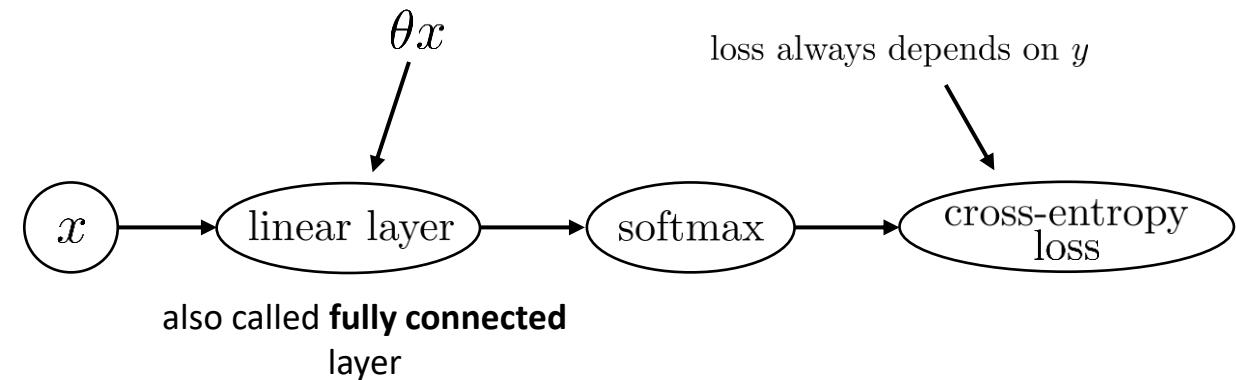
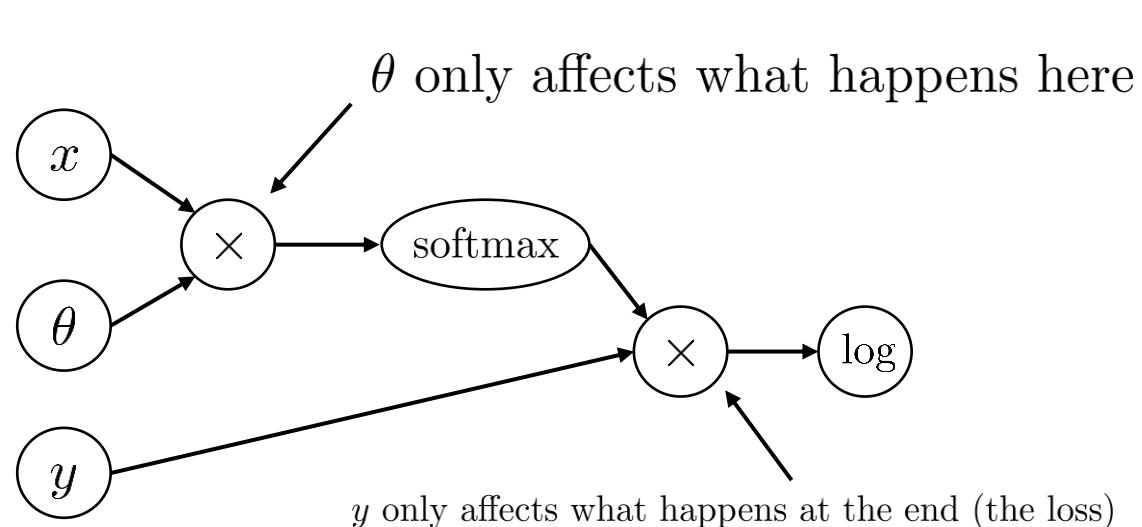
Drawing it even *more* concisely

Notice that we have **two types** of variables:

data (e.g., x, y), which serves as input or target output

parameters (e.g., θ) the parameters *usually* affect one specific operation

(though there is often *parameter sharing*, e.g., conv nets – more on this later)

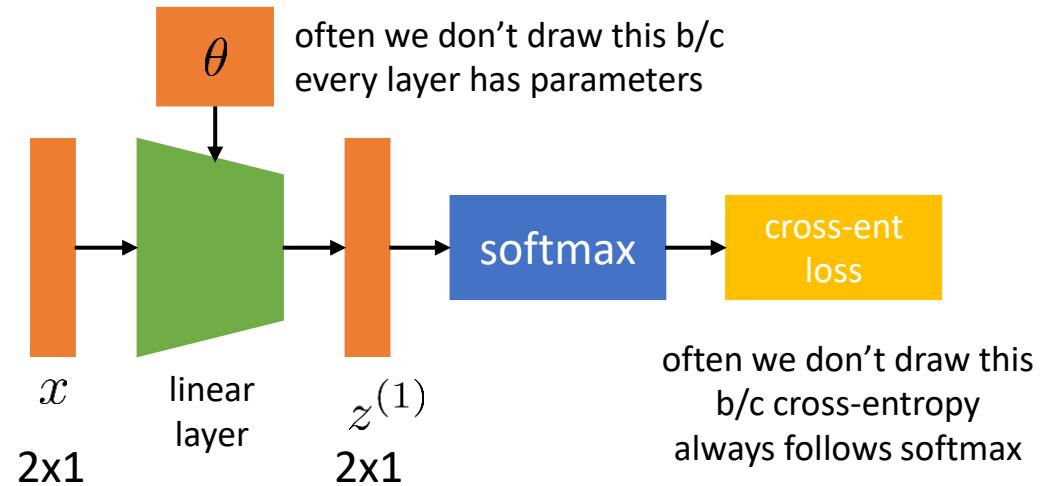


Neural network diagrams

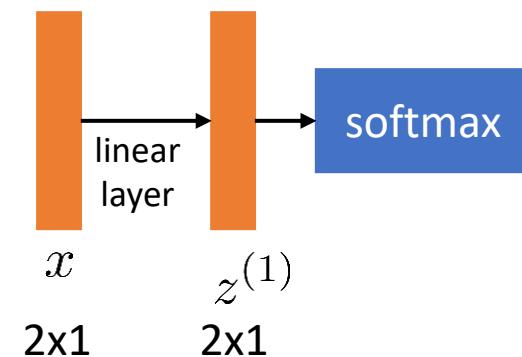
(simplified) computation graph diagram



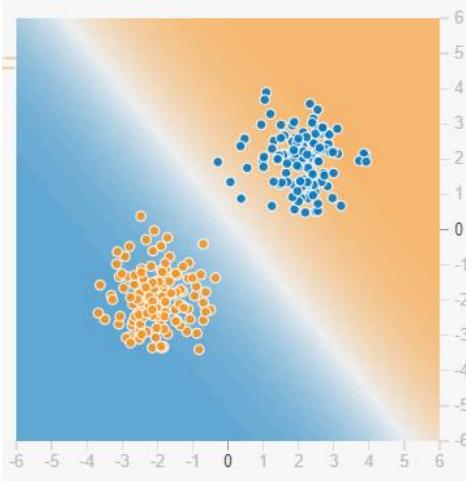
neural network diagram



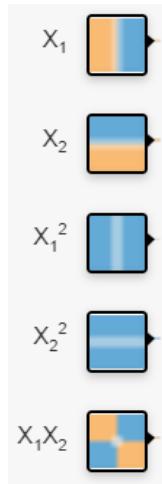
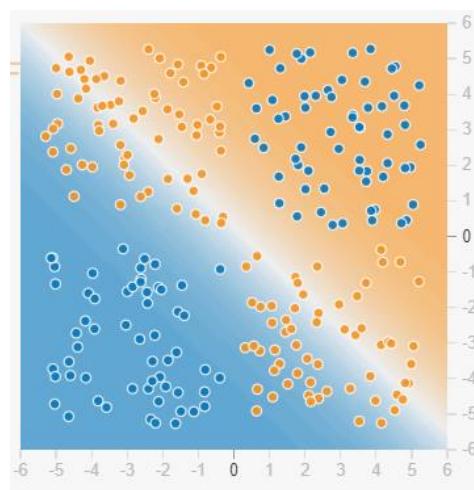
simplified drawing:



Logistic regression with features



$\text{softmax}(x^T \theta)$



pop quiz: what is the dimensionality of θ ?

$$\phi(x) = \begin{pmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1 x_2 \end{pmatrix}$$

$\text{softmax}(\phi(x)^T \theta)$

Learning the features

Problem: how do we represent the learned features?

Idea: what if each feature is a (binary) logistic regression output?

$$\phi_1(x) = \text{softmax}(x^T w_1^{(1)}) = \frac{1}{1 + \exp(-x^T w_1^{(1)})}$$

$$\phi(x) = \begin{pmatrix} \text{softmax}(x^T w_1^{(1)}) \\ \text{softmax}(x^T w_2^{(1)}) \\ \text{softmax}(x^T w_3^{(1)}) \end{pmatrix} = \sigma(W^{(1)}x)$$

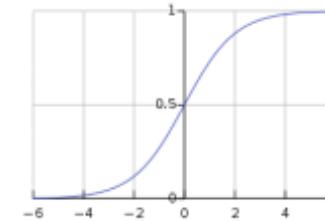
per-element sigmoid

not the same as softmax

each feature is independent

$w_1^{(1)}$

which layer
which feature
= rows of weight **matrix**



$$W^{(1)} = \begin{bmatrix} w_1^{(1)} \\ w_2^{(1)} \\ w_3^{(1)} \end{bmatrix}$$

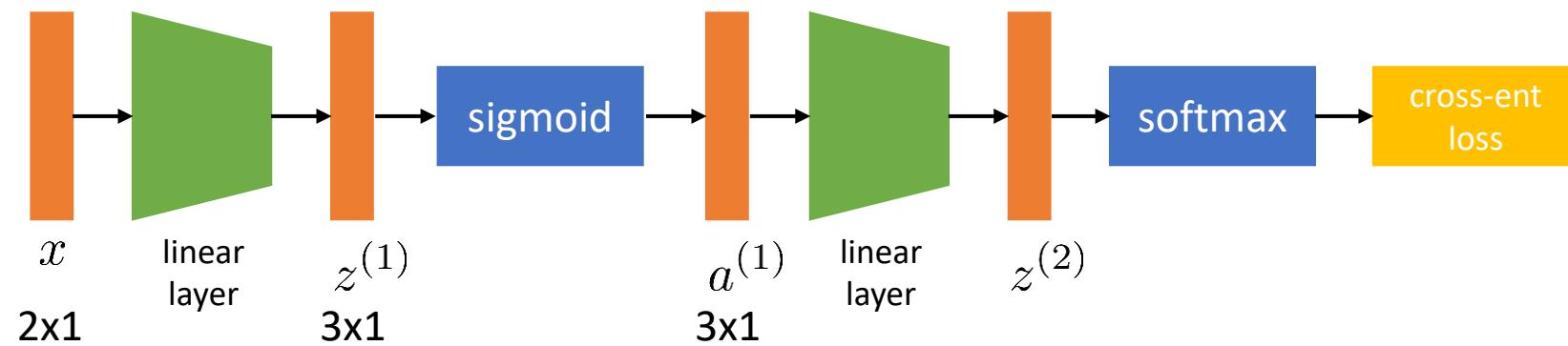
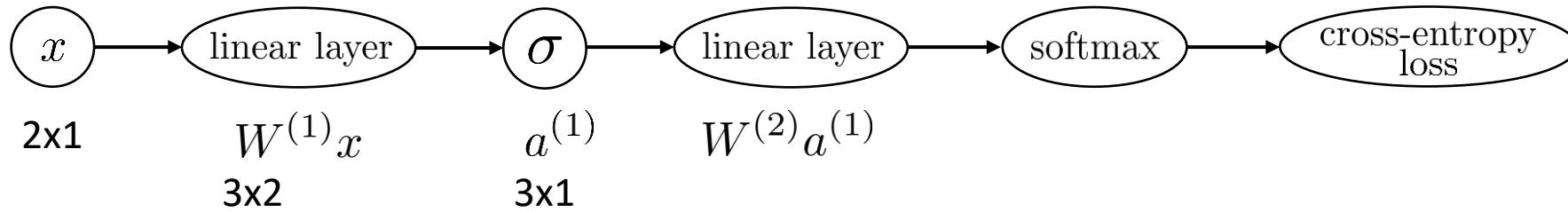
aside: I'll switch to use w or W instead of θ here

θ – all parameters of the model

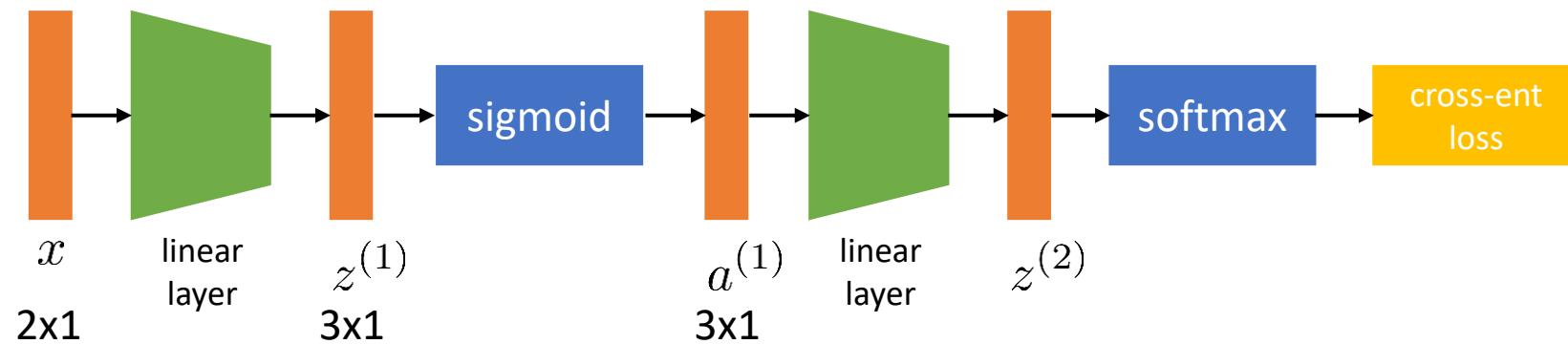
$w_1^{(1)}$ – weights (a.k.a. parameters) of feature 1 at layer 1

Let's draw this!

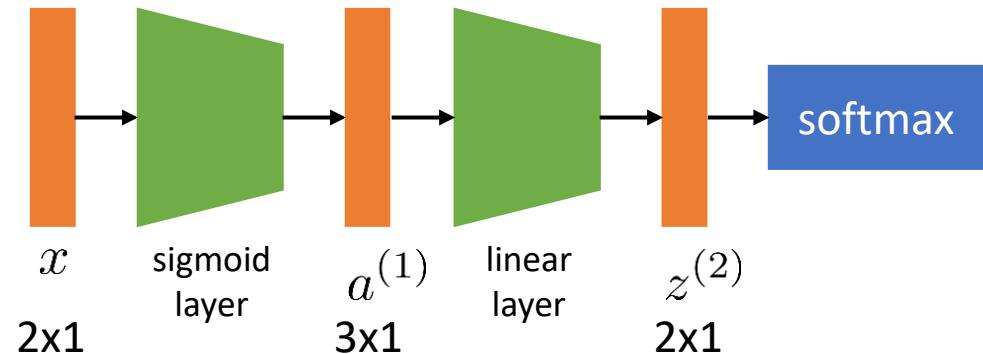
$$\phi(x) = \begin{pmatrix} \text{softmax}(x^T w_1^{(1)}) \\ \text{softmax}(x^T w_2^{(1)}) \\ \text{softmax}(x^T w_3^{(1)}) \end{pmatrix} = \sigma(W^{(1)}x) \quad p(y|x) = \text{softmax}(\phi(x)^T \theta)$$



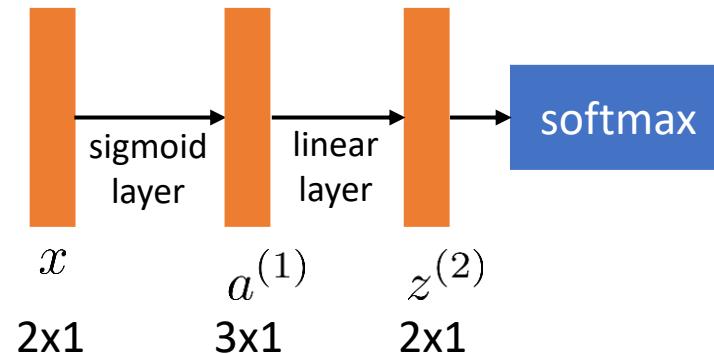
Simpler drawing



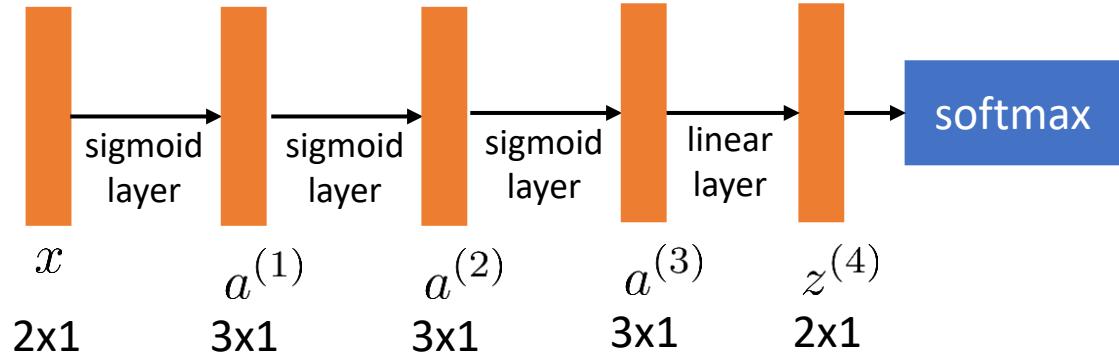
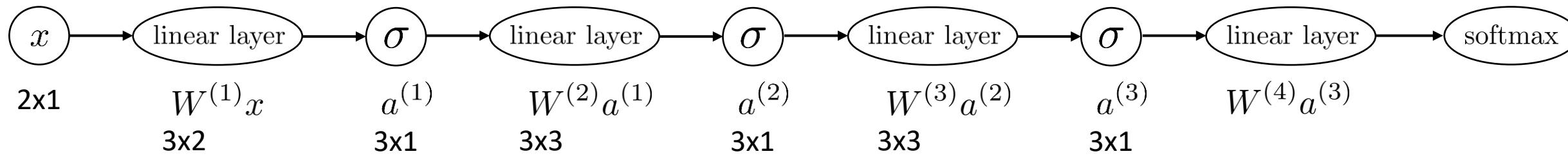
simpler way to draw the same thing:



even simpler:

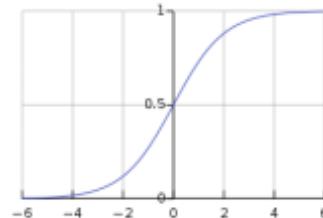


Doing it multiple times



Activation functions

$$\phi_1(x) = \text{softmax}(x^T w_1^{(1)}) = \frac{1}{1 + \exp(-x^T w_1^{(1)})}$$



we don't have to use a **sigmoid!**

a wide range of non-linear functions will work
these are called **activation functions**

we'll discuss specific choices later
why **non-linear?**

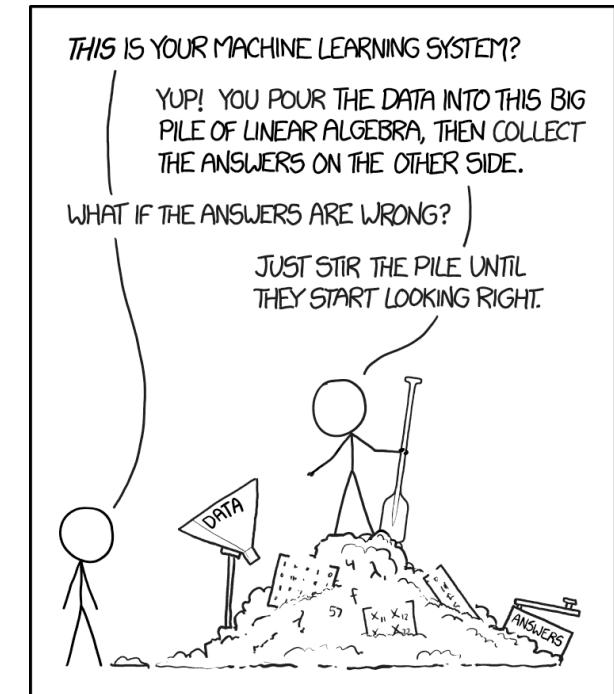
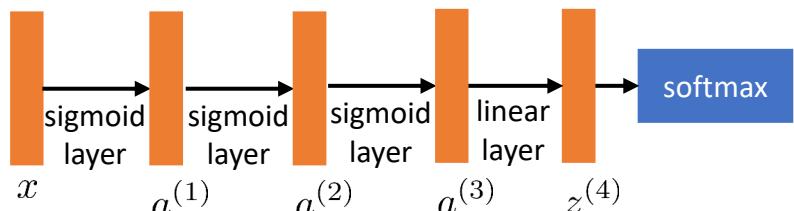
$$a^{(2)} = \sigma(W^{(2)}\sigma(W^{(1)}x))$$

if $\sigma(z) = z$, then...

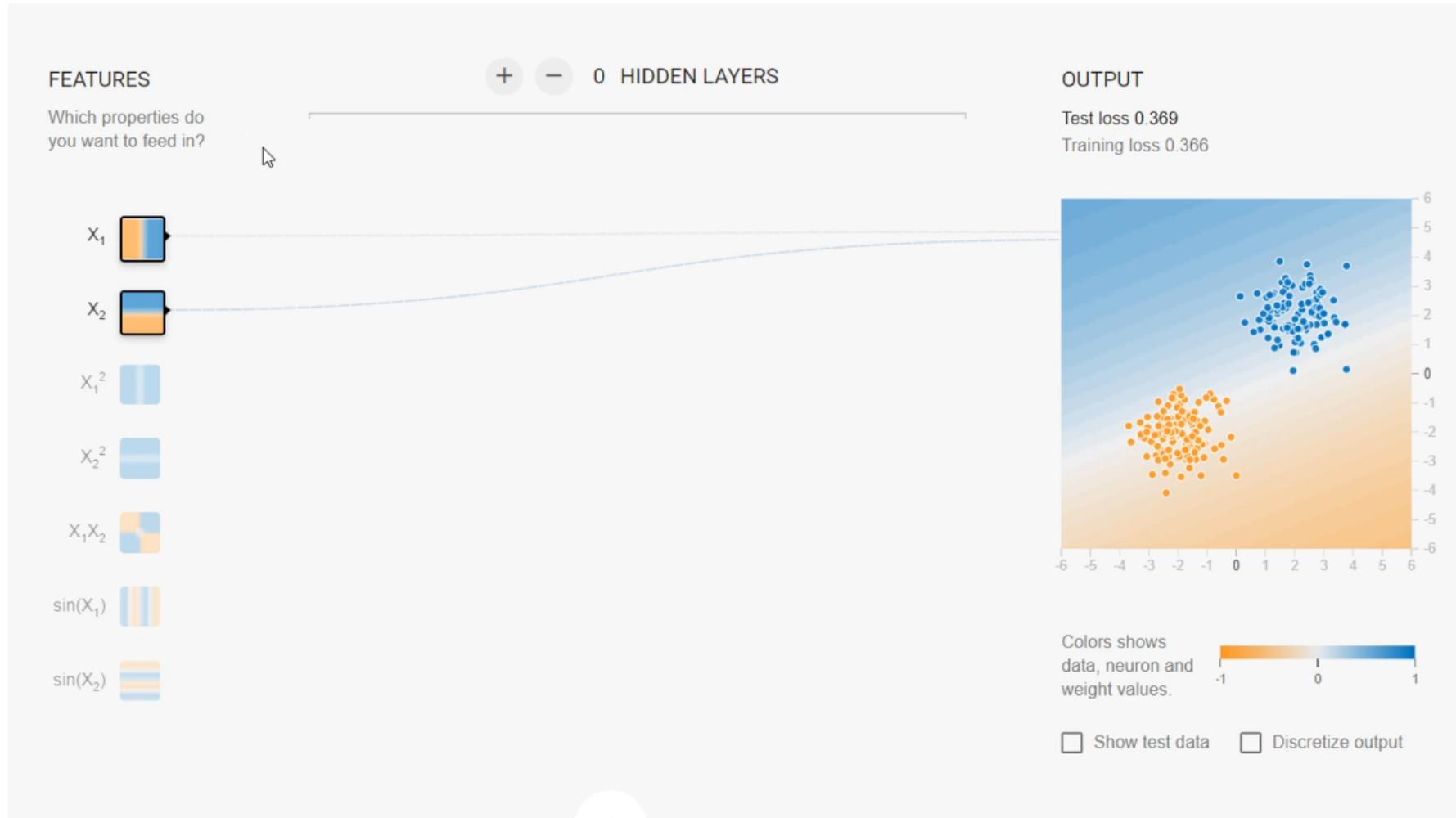
$$a^{(2)} = W^{(2)}W^{(1)}x = Mx$$

multiple linear layers = one linear layer

enough layers = we can represent anything (so long as they're nonlinear)

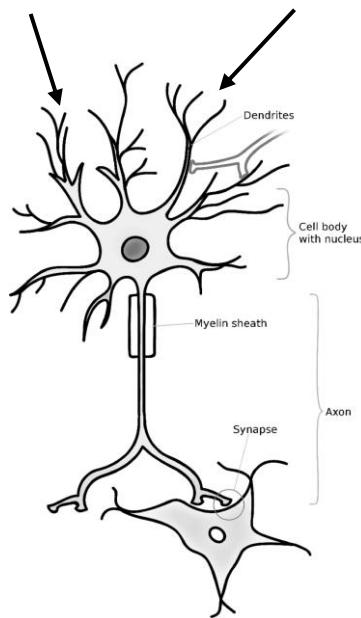


Demo time!



Aside: what's so neural about it?

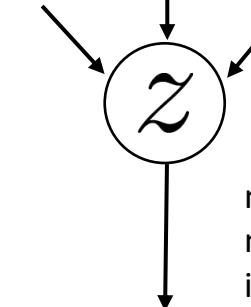
dendrites receive signals from other neurons



neuron “decides”
whether to fire based
on incoming signals

axon transmits signal to
downstream neurons

artificial “neuron” sums up signals
from upstream neurons
(also referred to as “units”)



neuron “decides” how
much to fire based on
incoming signals

activations transmitted
to downstream units

$$z = \sum_i a_i$$

upstream activations

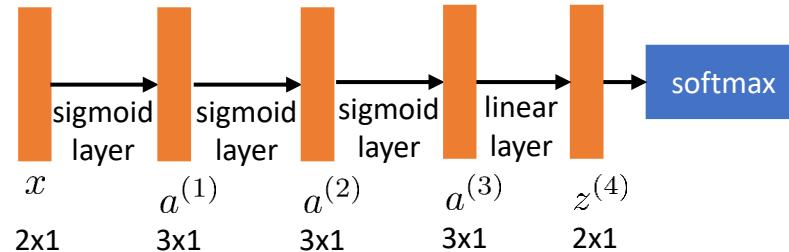
$$a = \sigma(z)$$

activation function

Training neural networks

What do we need?

1. Define your **model class**



2. Define your **loss function**

negative log-likelihood, just like before

3. Pick your **optimizer**

stochastic gradient descent
what do we need?

4. Run it on a big GPU

$$\nabla_{\theta} \mathcal{L}(\theta) = \begin{pmatrix} \frac{d\mathcal{L}(\theta)}{d\theta_1} \\ \frac{d\mathcal{L}(\theta)}{d\theta_2} \\ \vdots \\ \frac{d\mathcal{L}(\theta)}{d\theta_n} \end{pmatrix}$$

Aside: chain rule

Chain rule:

$$x \xrightarrow{g} y \xrightarrow{f} z$$

$$\frac{d}{dx} f(g(x)) = \frac{dz}{dx} = \frac{dy}{dx} \frac{dz}{dy}$$

↑ ↓
Jacobian of g Jacobian of f

Row or column?

In this lecture:



$y \in R^m$

$$\frac{dz}{dy} \in R^m$$

$$\frac{dy}{dx} \in R^{n \times m}$$

$$\left(\frac{dy}{dx} \right)_{ij} = \frac{dy_j}{dx_i}$$

In some textbooks:

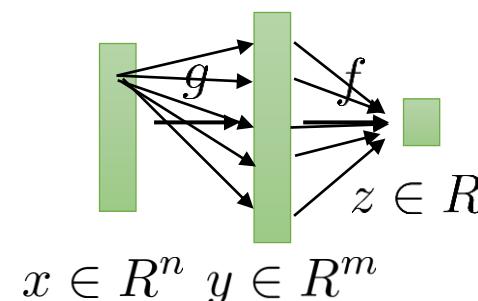


$$y \in R^m$$

$$\frac{dz}{dy} \in R^m$$

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

Just two different conventions!



High-dimensional chain rule

$$\frac{d}{dx_i} f(g(x)) = \sum_{j=1}^m \frac{dy_j}{dx_i} \frac{dz}{dy_j} = \frac{dy}{dx_i} \frac{dz}{dy}$$

↑ ↑
row $1 \times m$ col $m \times 1$
sum over all dimensions of y

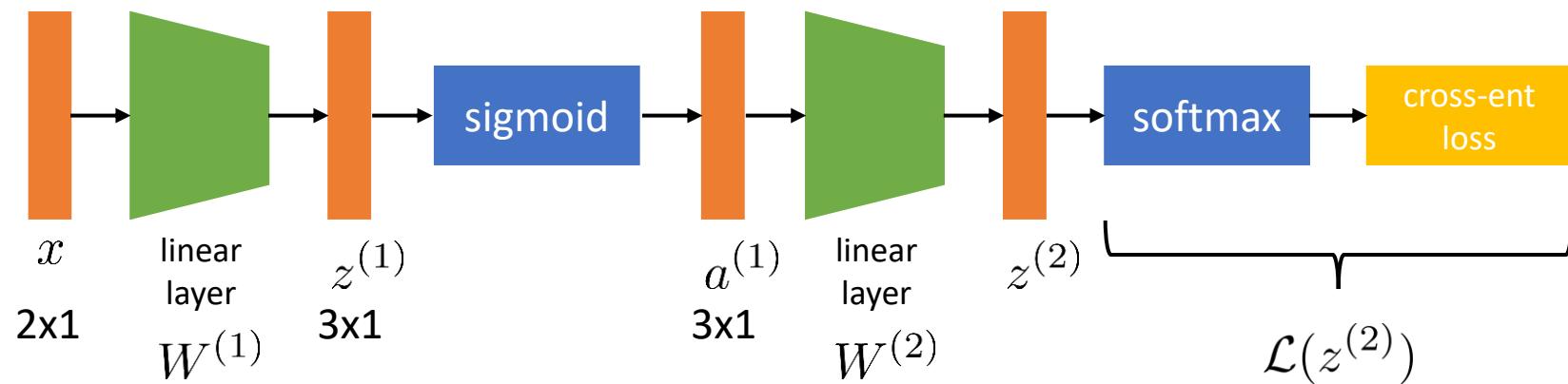
$$\frac{d}{dx} f(g(x)) = \frac{dy}{dx} \frac{dz}{dy}$$

↑ ↑
mat $n \times m$ col $m \times 1$

Chain rule for neural networks

A neural network is just a composition of functions

So we can use chain rule to compute gradients!



$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}}$$

$$\frac{d\mathcal{L}}{dW^{(2)}} = \frac{dz^{(2)}}{dW^{(2)}} \frac{d\mathcal{L}}{dz^{(2)}}$$

Does it work?

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}}$$

We **can** calculate each of these Jacobians!

Example:

$$z^{(2)} = W^{(2)}a^{(1)}$$

$$\frac{dz^{(2)}}{da^{(1)}} = W^{(2)T}$$

Why might this be a **bad** idea?

if each $z^{(i)}$ or $a^{(i)}$ has about n dims...

each Jacobian is about $n \times n$ dimensions

matrix multiplication is $O(n^3)$

do we care?

AlexNet has layers with 4096 units...

Doing it more efficiently

this product is cheap: $O(n^2)$

this product is expensive

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}}$$

$n \times n$

$n \times 1$

this is **always** true because
the loss is scalar-valued!

Idea: start on the right

compute $\frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}} = \delta$ first

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \delta$$

this product is cheap: $O(n^2)$

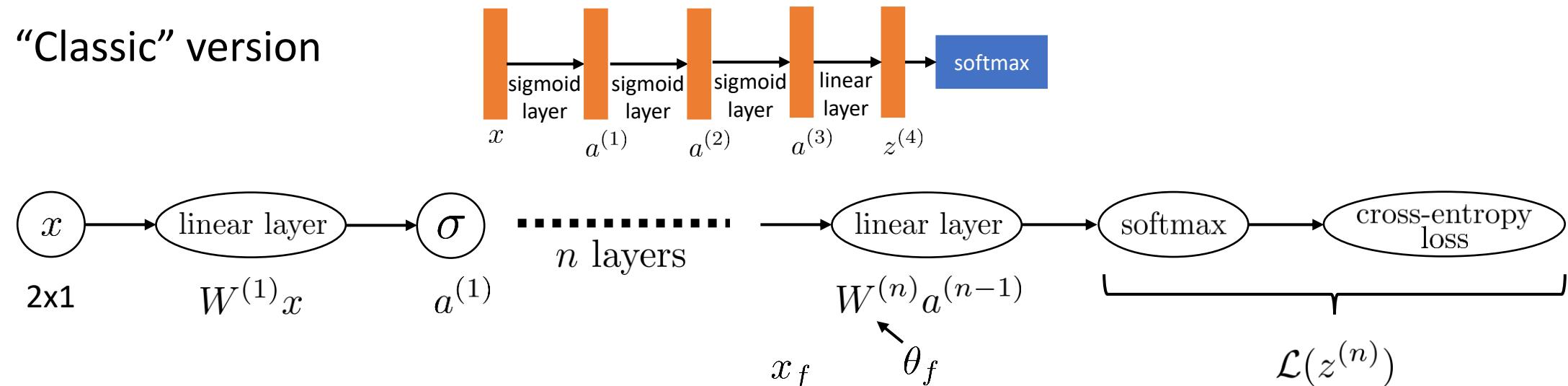
compute $\frac{da^{(1)}}{dz^{(1)}} \delta = \gamma$

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \gamma$$

this product is cheap: $O(n^2)$

The backpropagation algorithm

“Classic” version



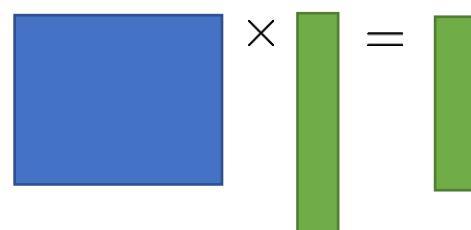
forward pass: calculate each $a^{(i)}$ and $z^{(i)}$ $a^{(n-1)} \longrightarrow f \longrightarrow z^{(n-1)}$

backward pass:

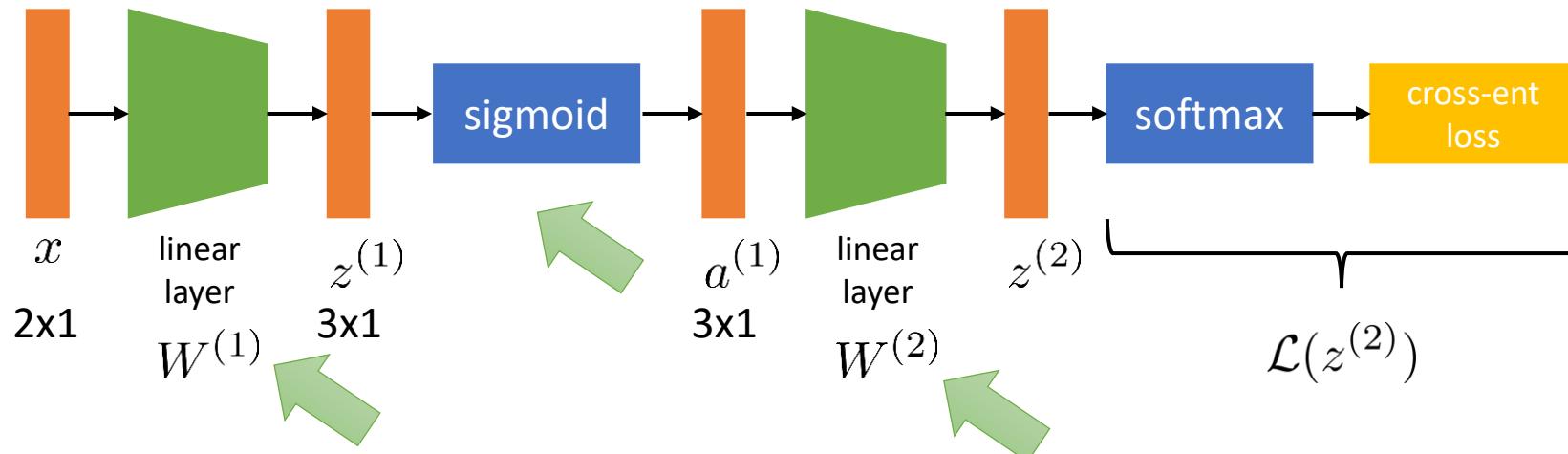
initialize $\delta = \frac{d\mathcal{L}}{dz^{(n)}}$

for each f with input x_f & params θ_f from end to start:

$$\begin{aligned}\frac{d\mathcal{L}}{d\theta_f} &\leftarrow \frac{df}{d\theta_f} \delta \\ \delta &\leftarrow \frac{df}{dx_f} \delta\end{aligned}$$



Let's walk through it...



forward pass: calculate each $a^{(i)}$ and $z^{(i)}$

backward pass:

→ initialize $\delta = \frac{d\mathcal{L}}{dz^{(n)}}$

for each f with input x_f & params θ_f from end to start:

→ $\frac{d\mathcal{L}}{d\theta_f} \leftarrow \frac{df}{d\theta_f} \delta$

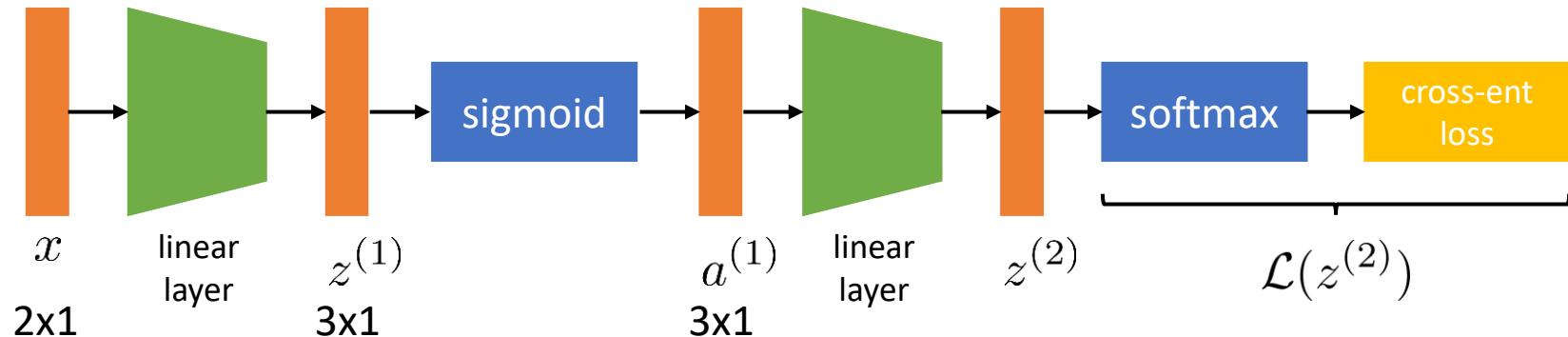
→ $\delta \leftarrow \frac{df}{dx_f} \delta$

$$\frac{d\mathcal{L}}{dW^{(2)}} = \underbrace{\frac{dz^{(2)}}{dW^{(2)}}}_{\delta} \underbrace{\frac{d\mathcal{L}}{dz^{(2)}}}_{\delta}$$

$$\frac{d\mathcal{L}}{dW^{(1)}} = \underbrace{\frac{dz^{(1)}}{dW^{(1)}}}_{\delta} \underbrace{\frac{da^{(1)}}{dz^{(1)}}}_{\delta} \underbrace{\frac{dz^{(2)}}{da^{(1)}}}_{\delta} \underbrace{\frac{d\mathcal{L}}{dz^{(2)}}}_{\delta}$$

Practical implementation

Neural network architecture details



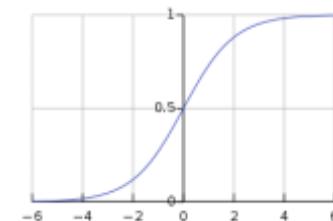
Some things we should figure out:

How many layers?

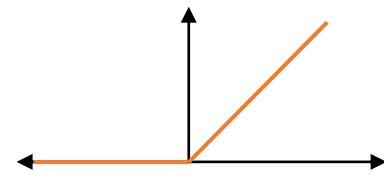
How big are the layers?

What type of **activation function**?

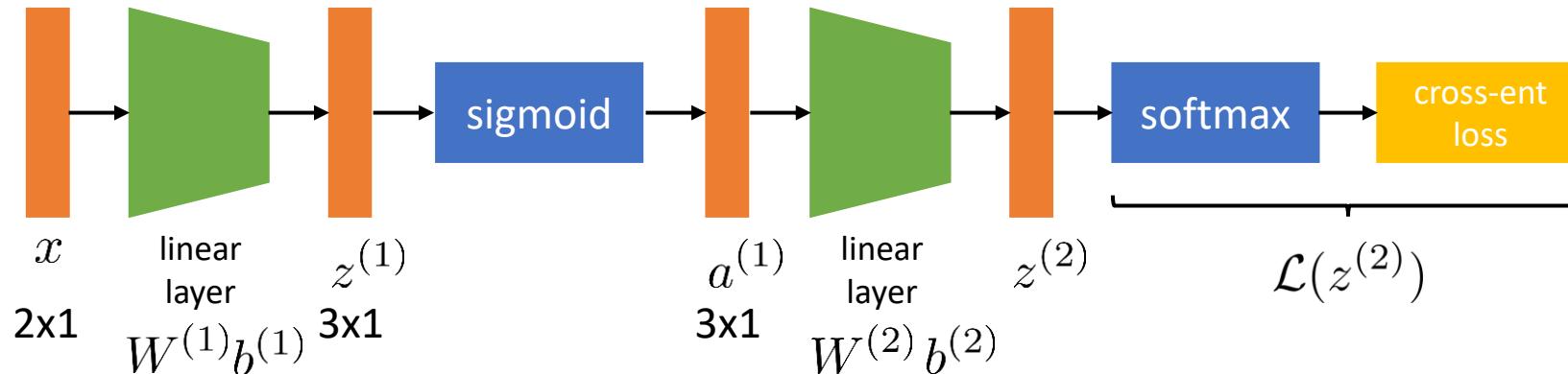
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



$$\text{ReLU}(x) = \max(0, x)$$



Bias terms



Linear layer:

$$z^{(i+1)} = W^{(i)} a^{(i)}$$

problem: if $a^{(i)} = \vec{0}$, we always get 0...

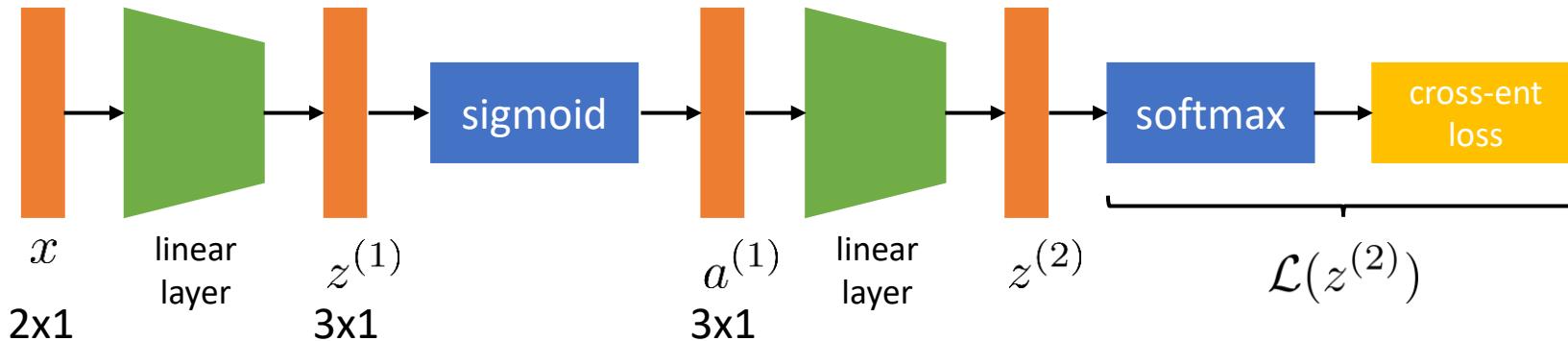
Solution: add a “bias”:

has nothing to do with bias/variance bias

$$z^{(i+1)} = W^{(i)} a^{(i)} + b^{(i)}$$

additional parameters in each linear layer

What else do we need for backprop?



forward pass: calculate each $a^{(i)}$ and $z^{(i)}$

backward pass:

initialize $\delta = \frac{d\mathcal{L}}{dz^{(n)}}$

for each f with input x_f & params θ_f from end to start:

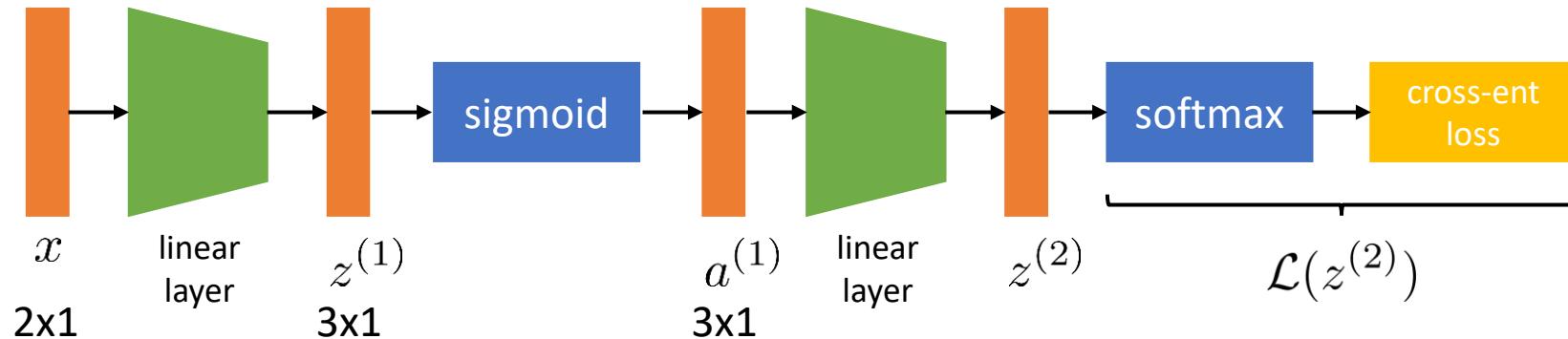
$$\frac{d\mathcal{L}}{d\theta_f} \leftarrow \frac{df}{d\theta_f} \delta$$

$$\delta \leftarrow \frac{df}{dx_f} \delta$$

for each function, we need to compute:

$$\begin{array}{ll} \frac{df}{d\theta_f} \delta & \frac{df}{dx_f} \delta \\ \text{linear layer} & \\ \text{softmax + cross-entropy} & \\ \text{sigmoid} & \\ \text{ReLU} & \end{array}$$

Backpropagation recipes: linear layer



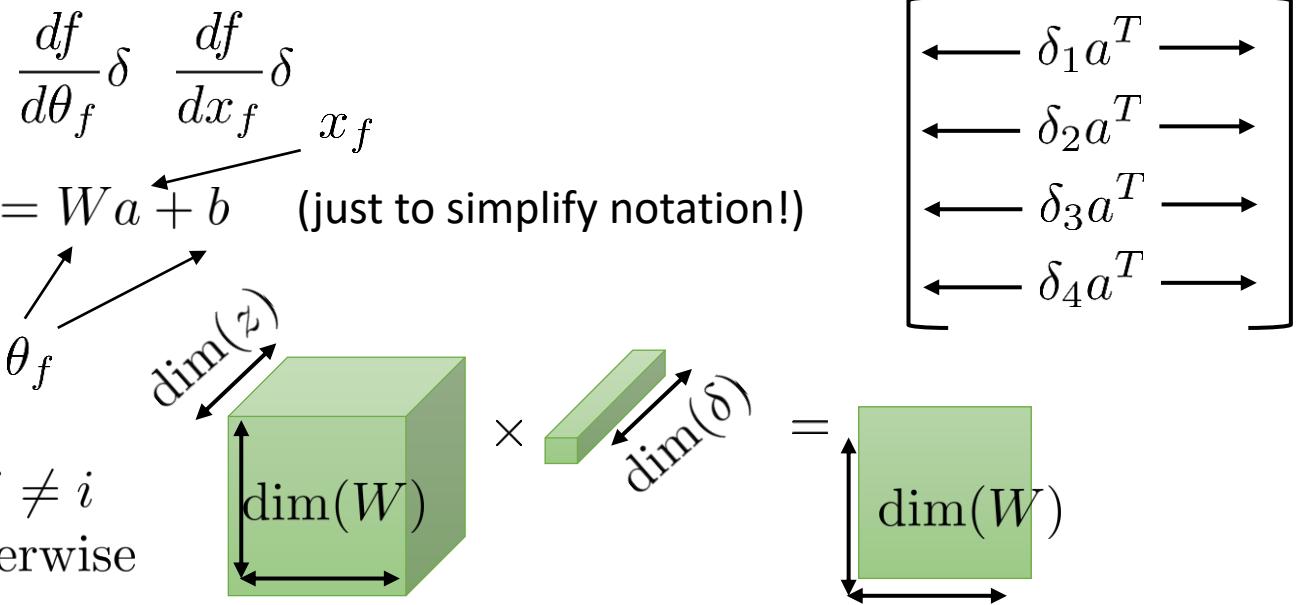
$$\frac{dz_3}{dW} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 \end{bmatrix} \times \delta_3$$

for each function, we need to compute: $\frac{df}{d\theta_f} \delta$ $\frac{df}{dx_f} \delta$

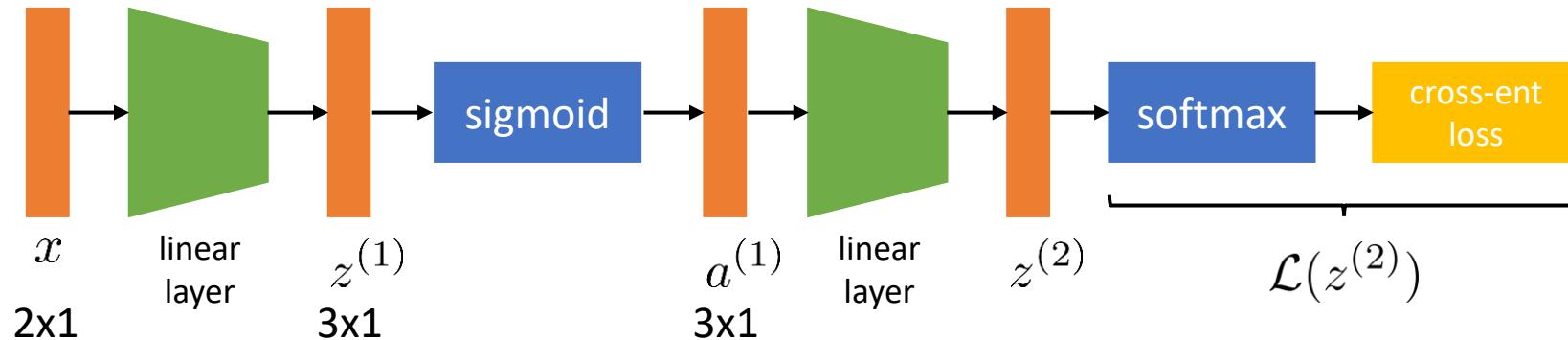
linear layer: $z^{(i+1)} = W^{(i)}a^{(i)} + b^{(i)}$ $z = Wa + b$ (just to simplify notation!)

$$\frac{dz}{dW} \delta = \sum_i \frac{dz_i}{dW} \delta_i = \delta a^T$$

$$z_i = \sum_k W_{ik} a_k + b_i \quad \frac{dz_i}{dW_{jk}} = \begin{cases} 0 & \text{if } j \neq i \\ a_k & \text{otherwise} \end{cases}$$



Backpropagation recipes: linear layer



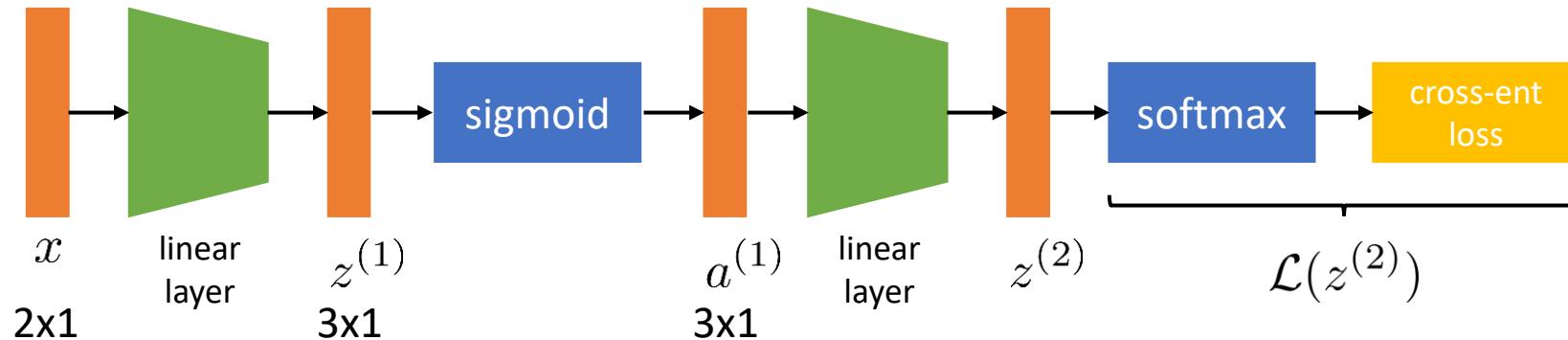
for each function, we need to compute: $\frac{df}{d\theta_f} \delta \quad \frac{df}{dx_f} \delta$

linear layer: $z^{(i+1)} = W^{(i)}a^{(i)} + b^{(i)}$ $z = Wa + b$ (just to simplify notation!)

$$\frac{dz}{db}\delta = \delta$$

$$z_i = \sum_k W_{ik}a_k + b_i \quad \frac{dz_i}{db_j} = \text{Ind}(i=j) \quad \frac{dz}{db} = \mathbf{I}$$

Backpropagation recipes: linear layer



for each function, we need to compute: $\frac{df}{d\theta_f} \delta \quad \frac{df}{dx_f} \delta$

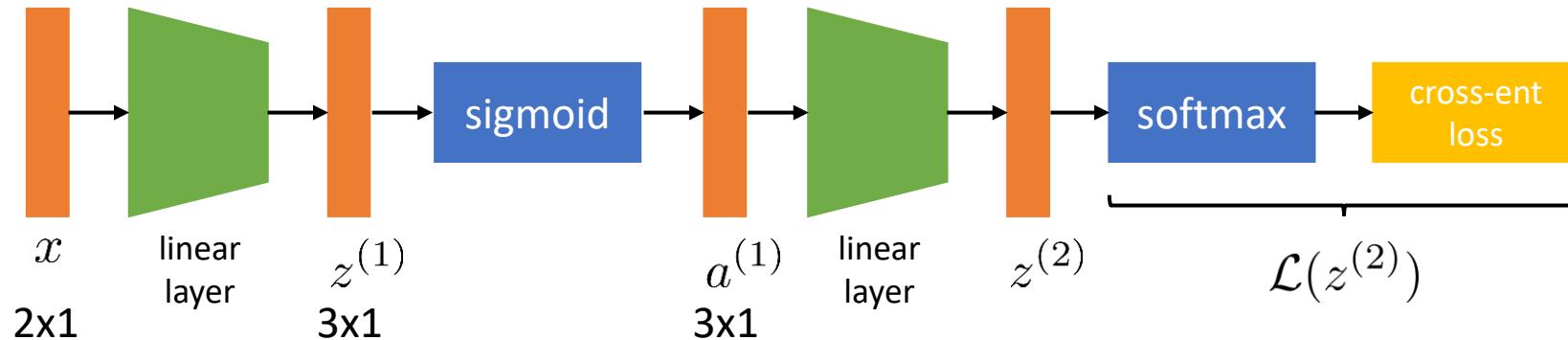
linear layer: $z^{(i+1)} = W^{(i)}a^{(i)} + b^{(i)}$ $z = Wa + b$ (just to simplify notation!)

$$\frac{dz}{da} \delta = W^T \delta$$

$$z_i = \sum_k W_{ik} a_k + b_i \quad \frac{dz_i}{da_k} = W_{ik} \quad \frac{dz}{da} = W^T$$

$$\left(\frac{dy}{dx} \right)_{ij} = \frac{dy_j}{dx_i}$$

Backpropagation recipes: linear layer

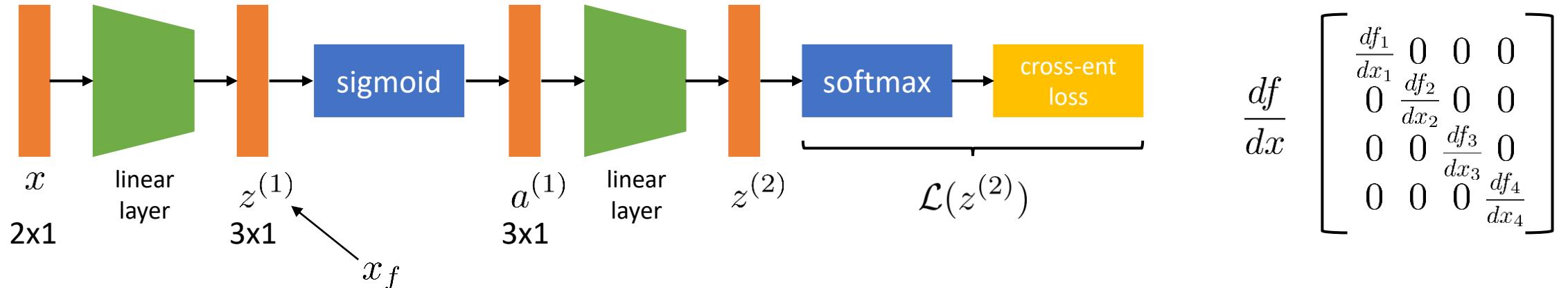


for each function, we need to compute: $\frac{df}{d\theta_f} \delta \quad \frac{df}{dx_f} \delta$

linear layer: $z^{(i+1)} = W^{(i)}a^{(i)} + b^{(i)}$ $z = Wa + b$ (just to simplify notation!)

$$\frac{dz}{da} \delta = W^T \delta \quad \underbrace{\frac{dz}{dW} \delta = \delta a^T}_{\left[\begin{array}{c} \frac{df}{dx_f} \delta \\ \frac{df}{d\theta_f} \delta \end{array} \right]} \quad \frac{dz}{db} \delta = \delta$$

Backpropagation recipes: sigmoid



for each function, we need to compute: $\frac{df}{d\theta_f} \delta \quad \frac{df}{dx_f} \delta$

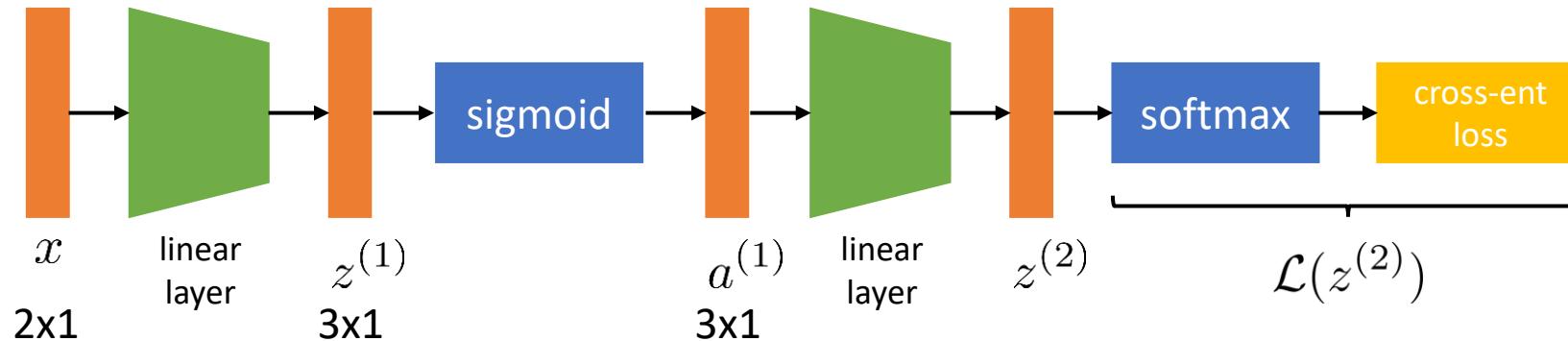
$$\sigma(z_i) = \frac{1}{1 + \exp(-z_i)}$$

$$\frac{df_i}{dz_i} = \underbrace{\frac{\exp(-z_i)}{1 + \exp(-z_i)}}_{\frac{1}{1 + \exp(-z_i)}} \underbrace{\frac{1}{1 + \exp(-z_i)}}_{(1 - \sigma(z_i))\sigma(z_i)} = (1 - \sigma(z_i))\sigma(z_i)$$

$$\left(\frac{df}{dz} \delta \right)_i = (1 - \sigma(z_i))\sigma(z_i)\delta_i$$

$$\underbrace{\frac{1 + \exp(-z_i)}{1 + \exp(-z_i)} - \frac{1 - \sigma(z_i)}{1 + \exp(-z_i)}}_{1 - \sigma(z_i)}$$

Backpropagation recipes: ReLU

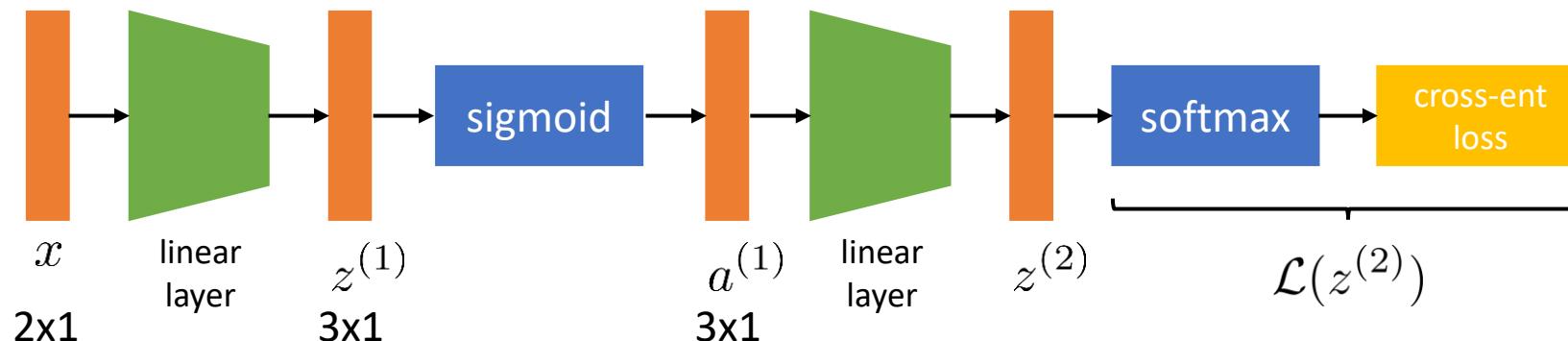


for each function, we need to compute: $\frac{df}{d\theta_f} \delta \quad \frac{df}{dx_f} \delta$

$$f_i(z_i) = \max(0, z_i) \quad \frac{df_i}{dz_i} = \text{Ind}(z_i \geq 0)$$

$$\left(\frac{df}{dz} \delta \right)_i = \text{Ind}(z_i \geq 0) \delta_i$$

Summary



forward pass: calculate each $a^{(i)}$ and $z^{(i)}$

for each function, we need to compute:

backward pass:

$$\text{initialize } \delta = \frac{d\mathcal{L}}{dz^{(n)}}$$

for each f with input x_f & params θ_f from end to start:

$$\frac{d\mathcal{L}}{d\theta_f} \leftarrow \frac{df}{d\theta_f} \delta$$

$$\delta \leftarrow \frac{df}{dx_f} \delta$$

$$\begin{aligned} & \frac{df}{d\theta_f} \delta & \frac{df}{dx_f} \delta \\ & \text{linear layer} & \\ & \text{softmax + cross-entropy} & \\ & \text{sigmoid} & \\ & \text{ReLU} & \end{aligned}$$

Convolutional Networks

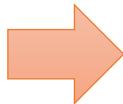
Designing, Visualizing and Understanding Deep Neural Networks

CS W182/282A

Instructor: Sergey Levine
UC Berkeley



Neural network with images



[object label]

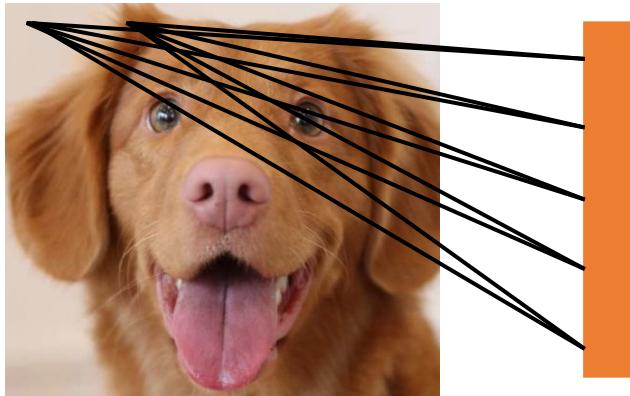
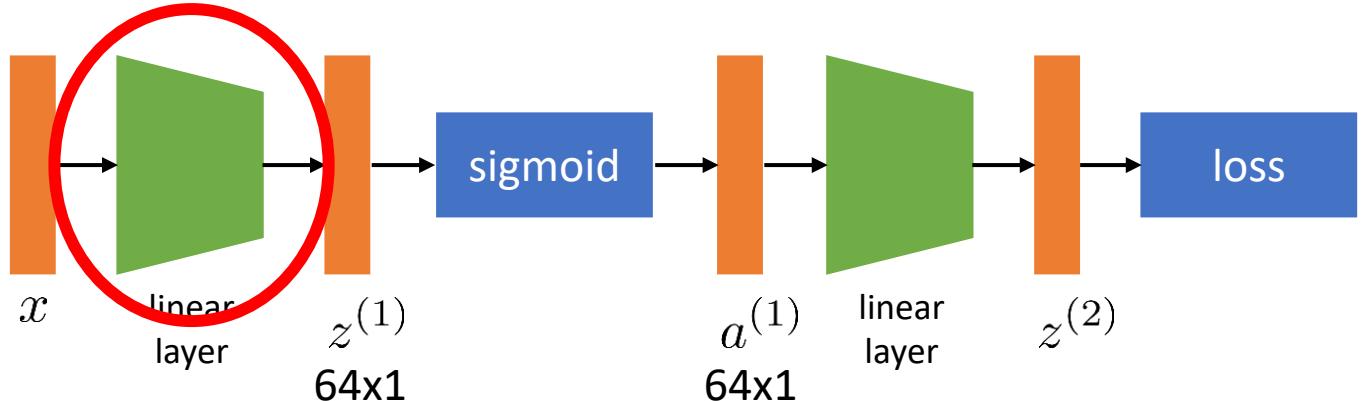


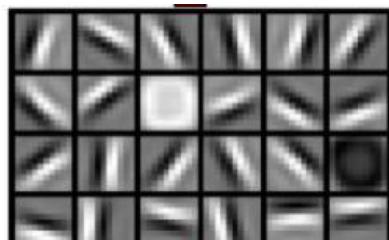
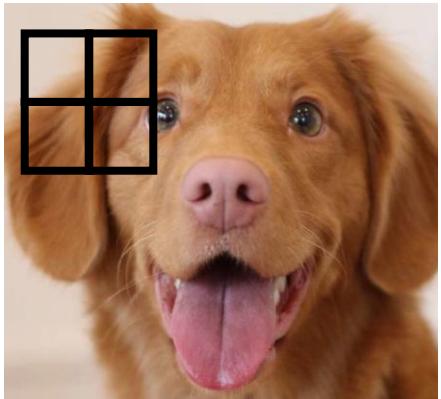
image is $128 \times 128 \times 3 = 49,152$

$z^{(1)}$ is 64-dim

$64 \times 49,152 \approx 3,000,000$

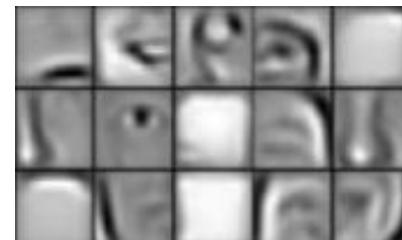
We need a better way!

An idea...



Layer 1:

edge detectors?



Layer 2:

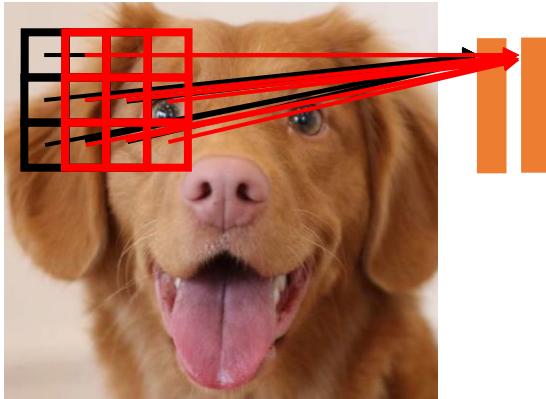
ears? noses?

Observation: many useful image features are **local**

to tell if a particular patch of image contains a feature, enough to look at the local patch

An idea...

Observation: many useful image features are **local**

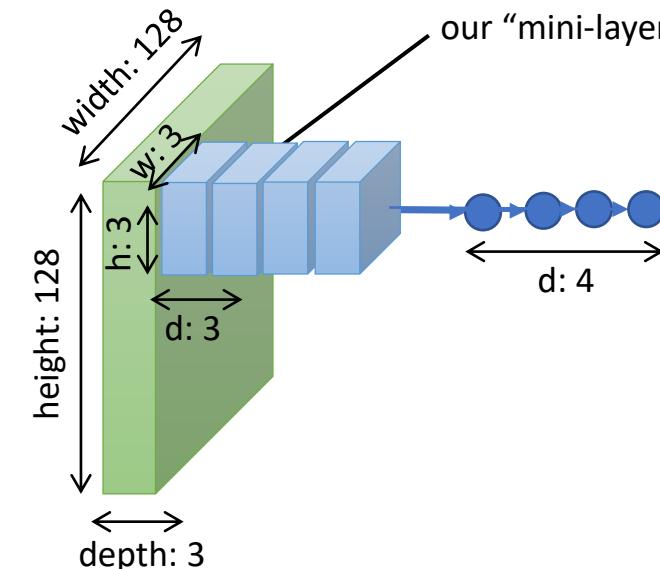


patch is $3 \times 3 \times 3 = 27$

$z^{(1)}$ is 64-dim

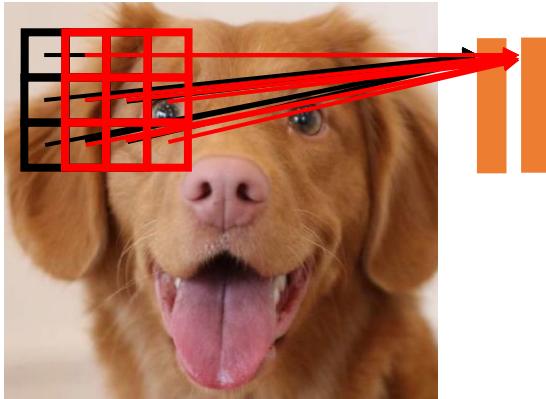
$$64 \times 27 = 1728$$

We get a **different** output at each image location!



An idea...

Observation: many useful image features are **local**

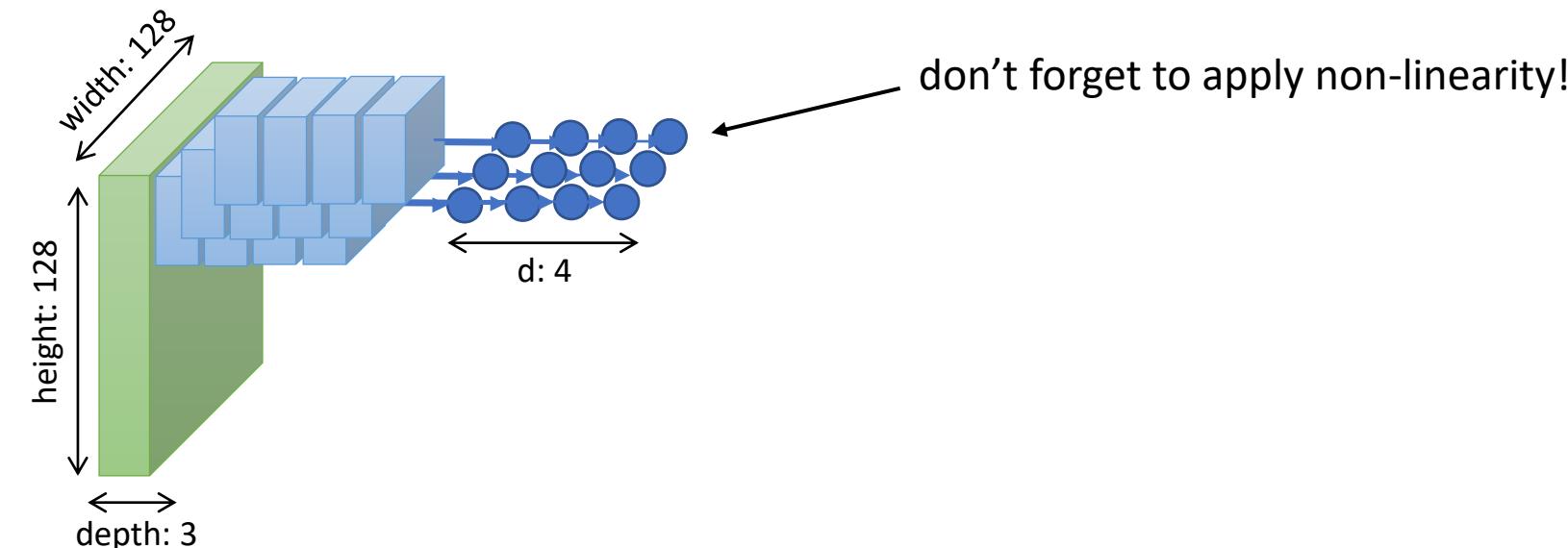


patch is $3 \times 3 \times 3 = 27$

$z^{(1)}$ is 64-dim

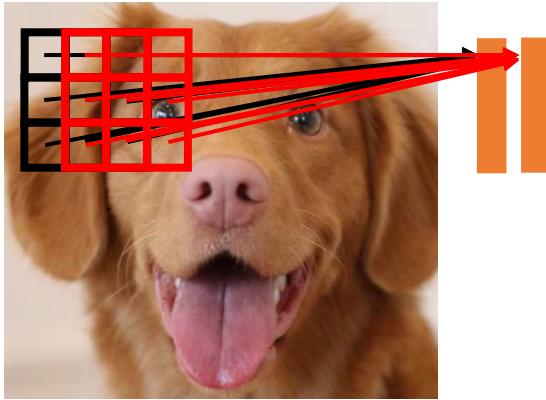
$$64 \times 27 = 1728$$

We get a **different** output at each image location!



An idea...

Observation: many useful image features are **local**

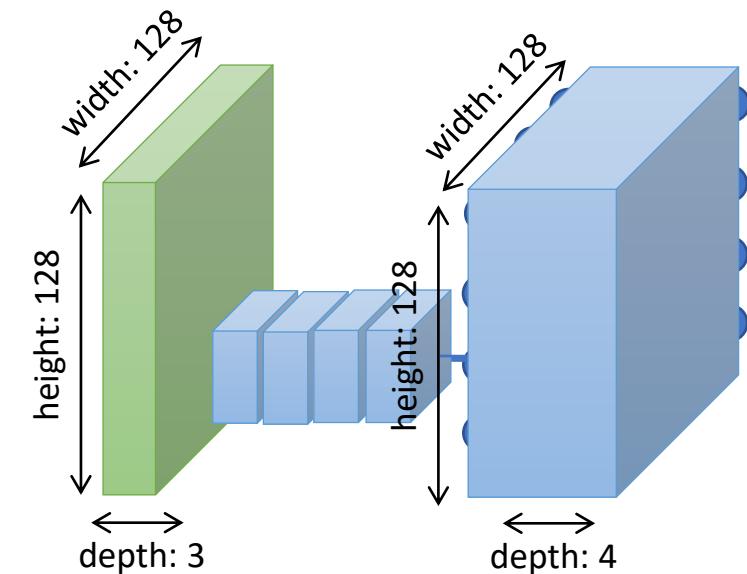


patch is $3 \times 3 \times 3 = 27$

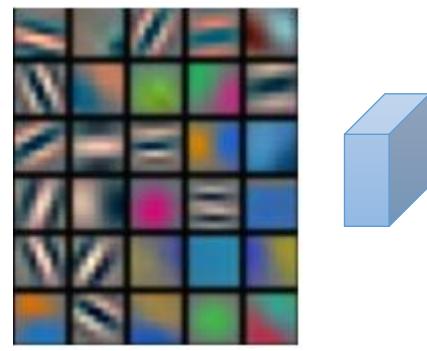
$z^{(1)}$ is 64-dim

$$64 \times 27 = 1728$$

We get a **different** output at each image location!

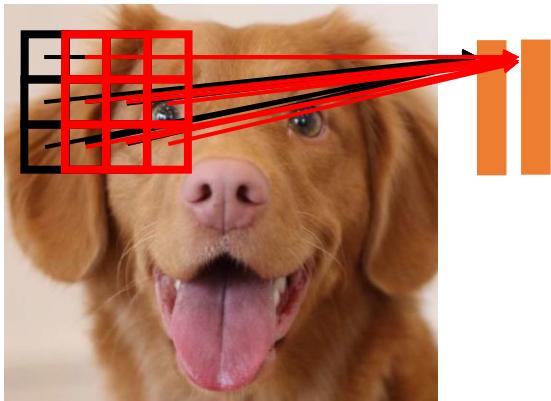


What do they look like?



An idea...

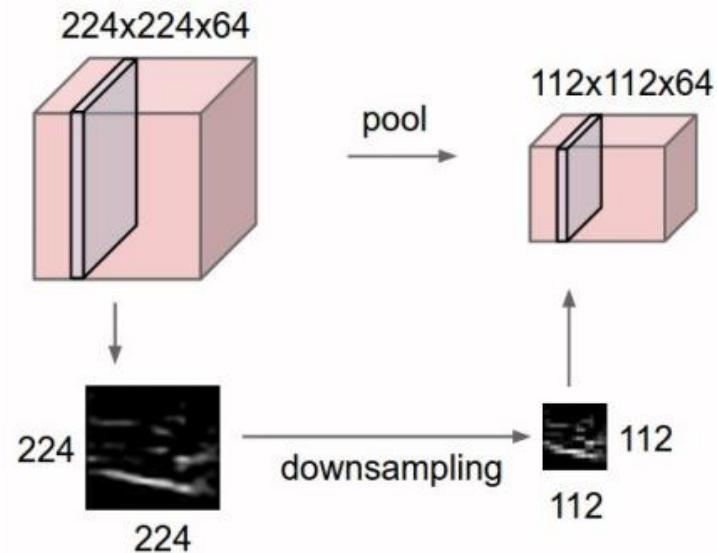
Observation: many useful image features are **local**



patch is $3 \times 3 \times 3 = 27$

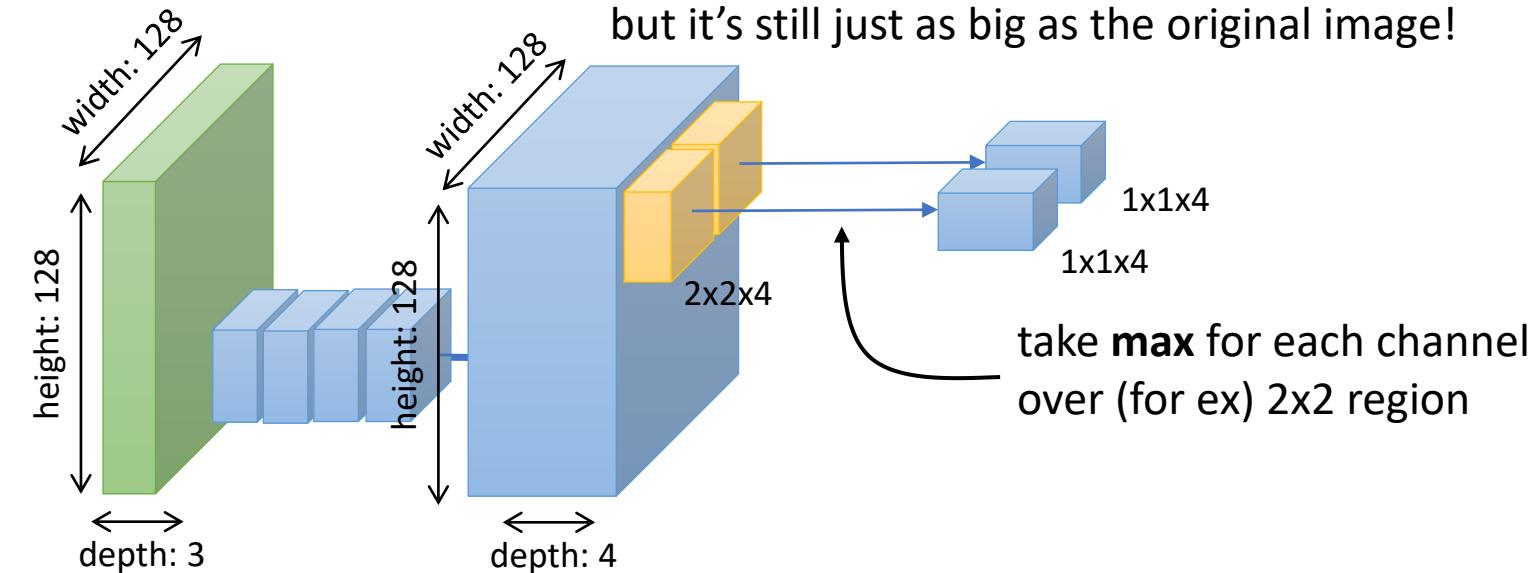
$z^{(1)}$ is 64-dim

$$64 \times 27 = 1728$$



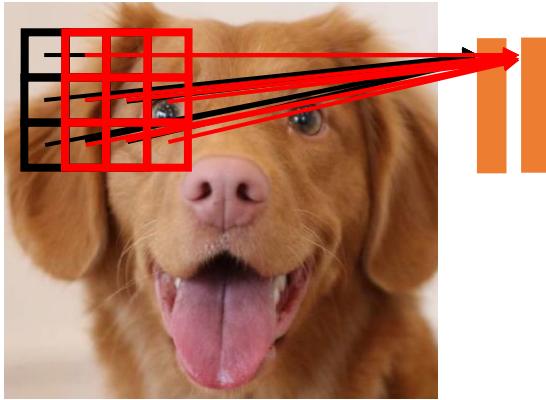
We get a **different** output at each image location!

but it's still just as big as the original image!



An idea...

Observation: many useful image features are **local**

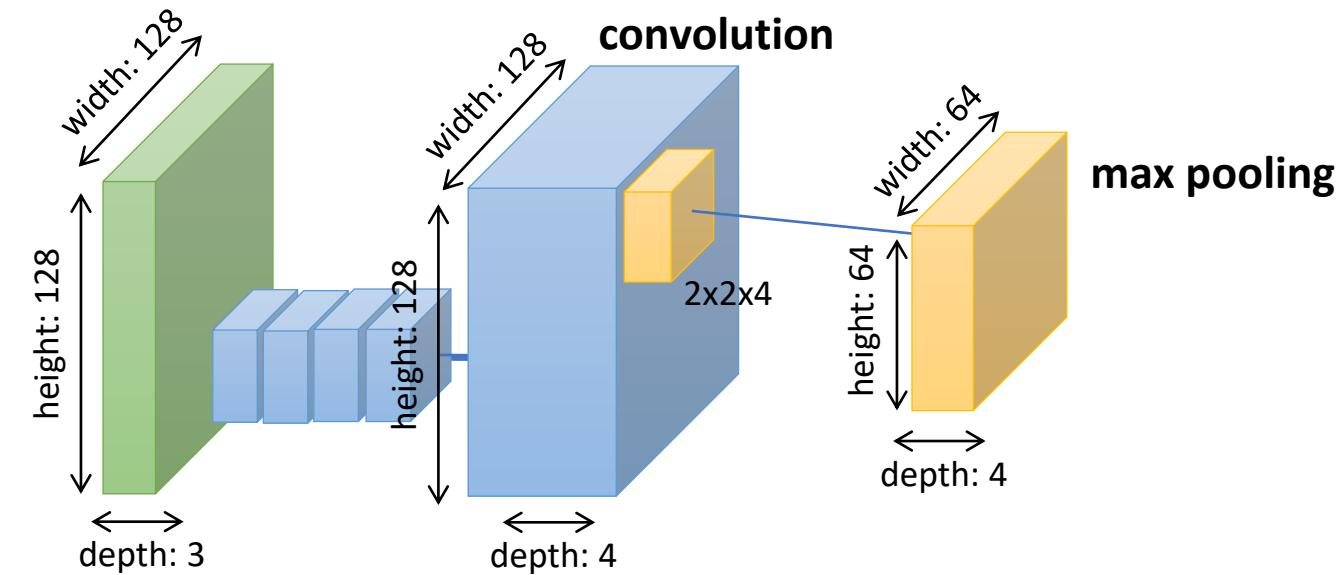


patch is $3 \times 3 \times 3 = 27$

$z^{(1)}$ is 64-dim

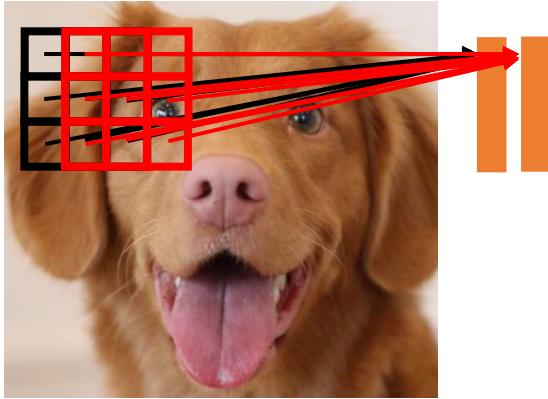
$$64 \times 27 = 1728$$

We get a **different** output at each image location!



An idea...

Observation: many useful image features are **local**

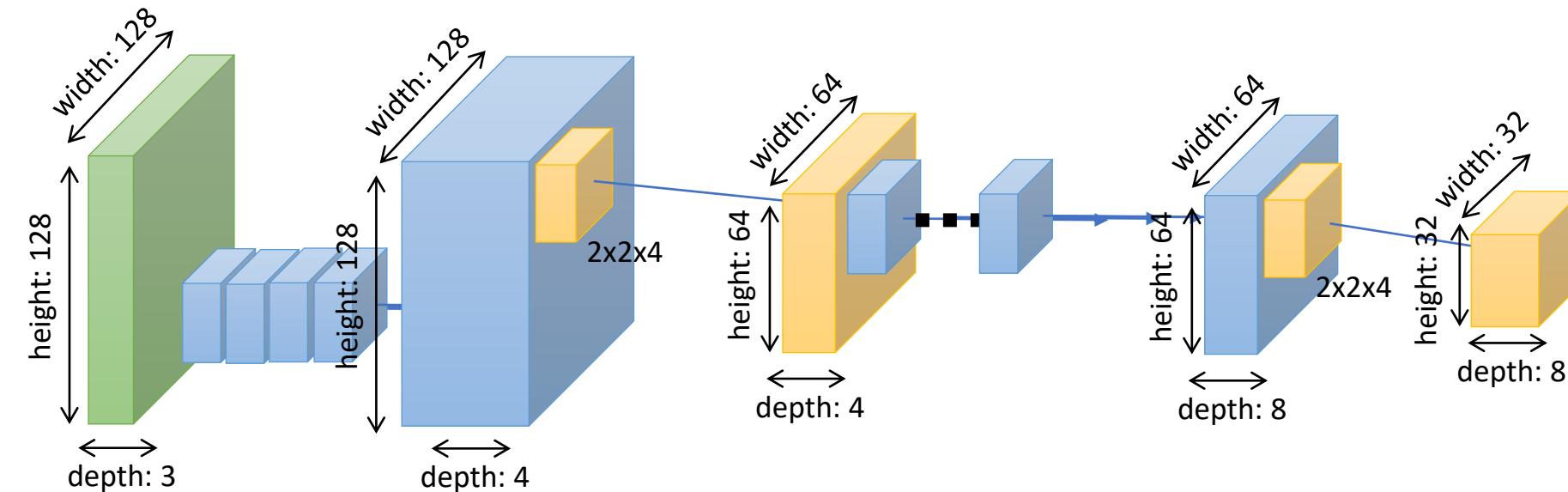


patch is $3 \times 3 \times 3 = 27$

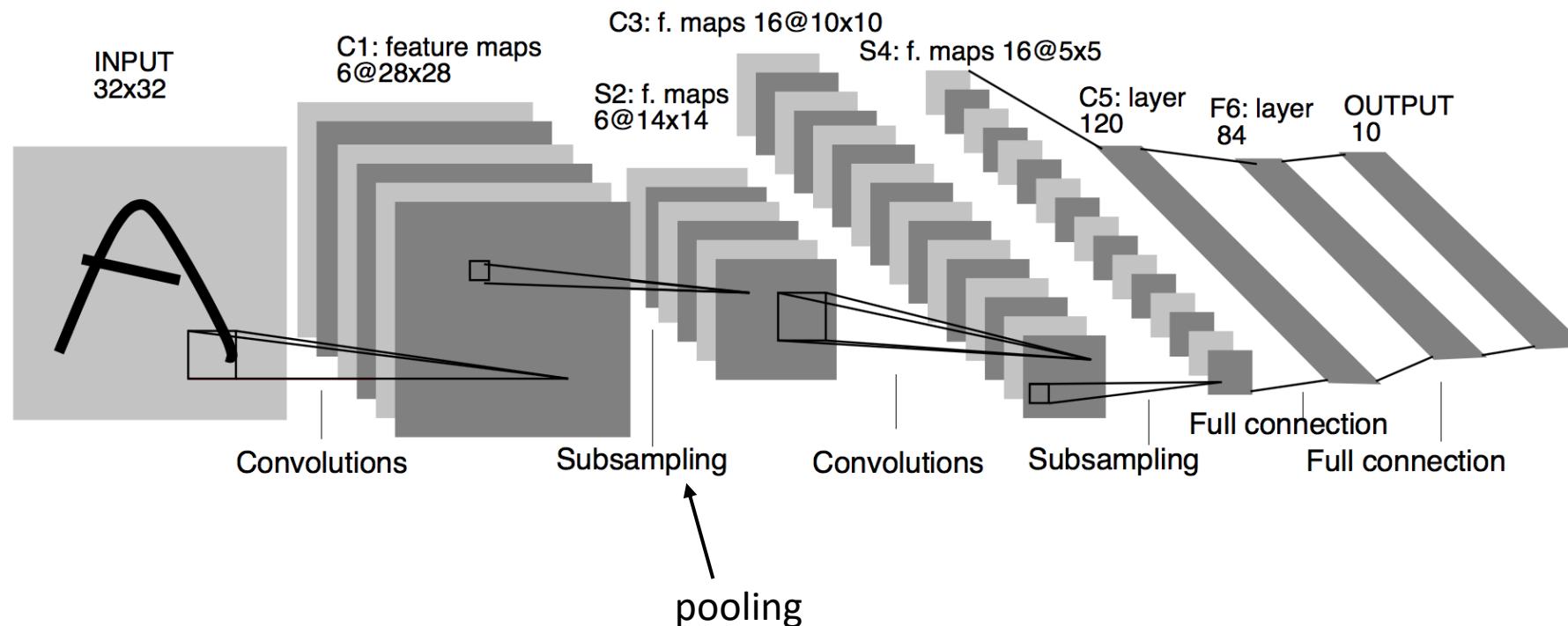
$z^{(1)}$ is 64-dim

$$64 \times 27 = 1728$$

We get a **different** output at each image location!



What does a real conv net look like?

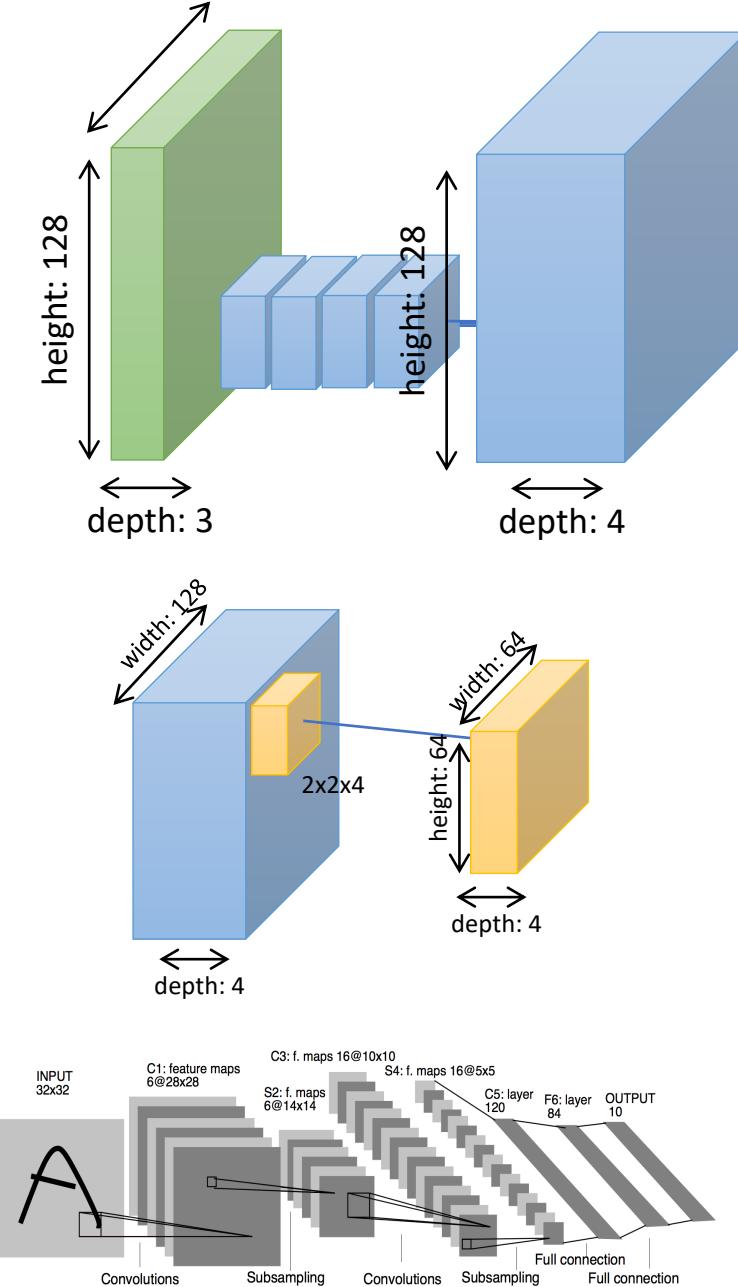


"LeNet" network for handwritten digit recognition

Implementing convolutional layers

Summary

- **Convolutional layer**
 - A way to avoid needing millions of parameters with images
 - Each layer is “local”
 - Each layer produces an “image” with (roughly) the same width & height, and number of channels = number of filters
- **Pooling**
 - If we ever want to get down to a single output, we must reduce resolution as we go
 - Max pooling: downsample the “image” at each layer, taking the max in each region
 - This makes it robust to small translation changes
- **Finishing it up**
 - At the end, we get something small enough that we can “flatten” it (turn it into a vector), and feed into a standard fully connected layer



ND arrays/tensors

all these operations will involve N -dimensional arrays

often used synonymously with *tensor*

input image: HEIGHT \times WIDTH \times CHANNELS

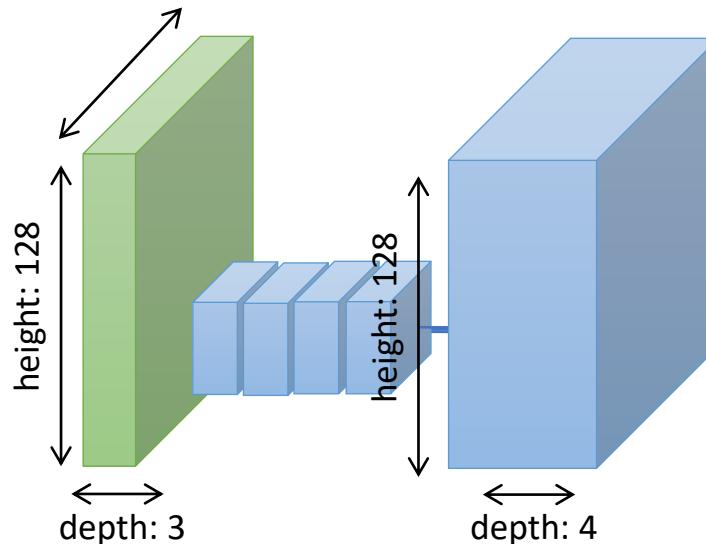
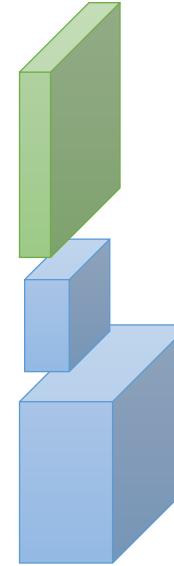
filter: FLT.HEIGHT \times FLT.WIDTH \times OUTPUT CHAN \times INPUT CHAN

activations: HEIGHT \times WIDTH \times LAYER.CHANNELS

The “inner” (rightmost) dimensions work just like vectors/matrices

Matching “outer” dimensions (e.g., height/width) are treated as “broadcast” (i.e., elementwise operations)

Convolution operations performs a tiny matrix multiply at each position (like a tiny linear layer at each position)



Convolutional layer in equations

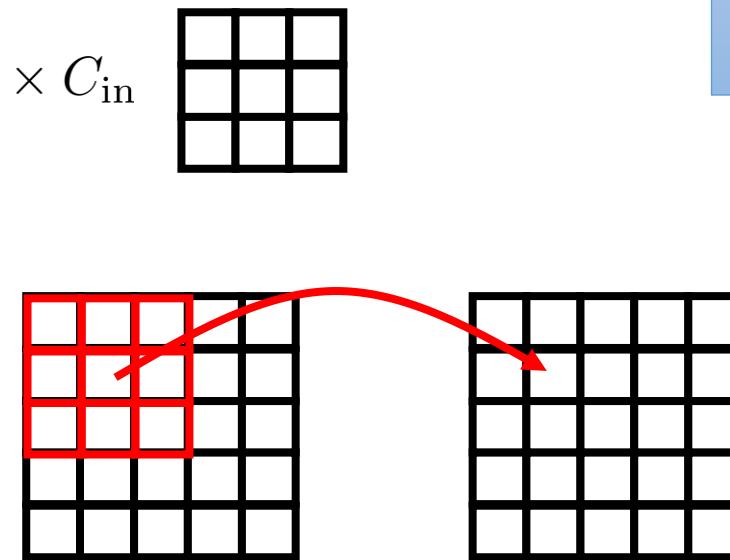
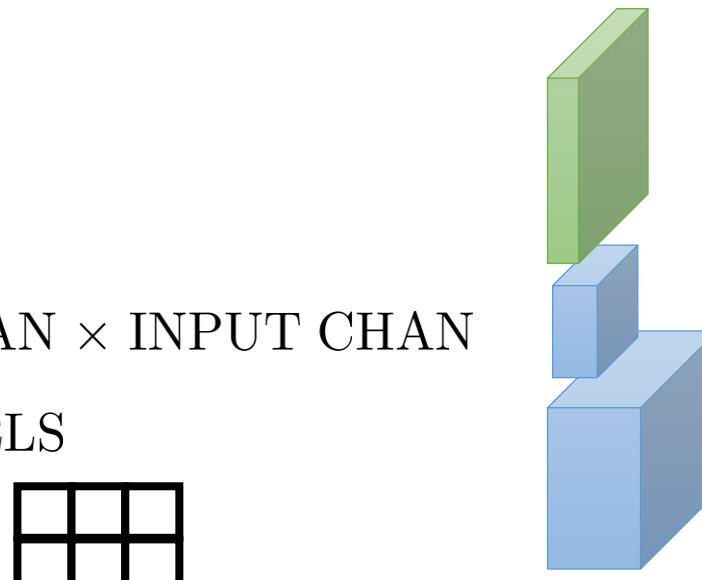
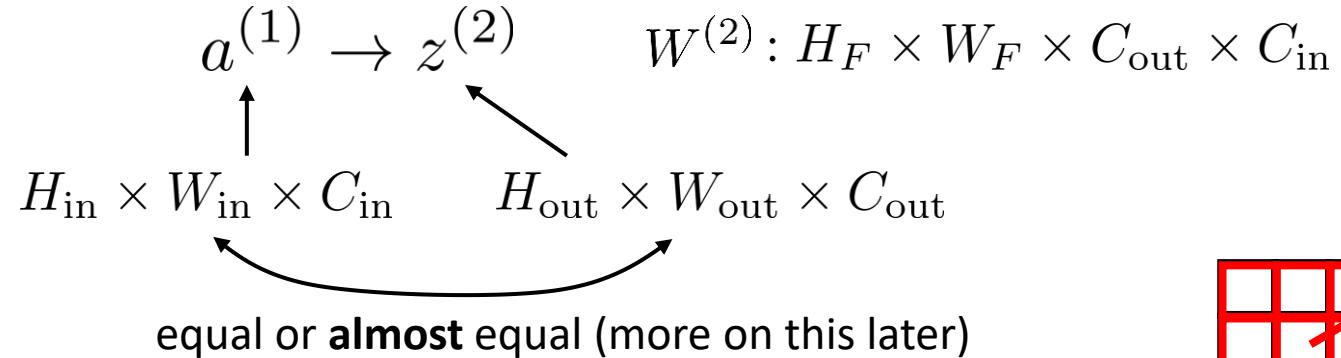
all these operations will involve N -dimensional arrays

often used synonymously with *tensor*

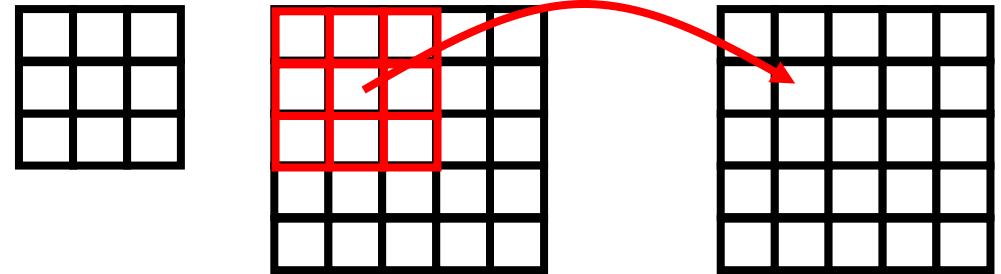
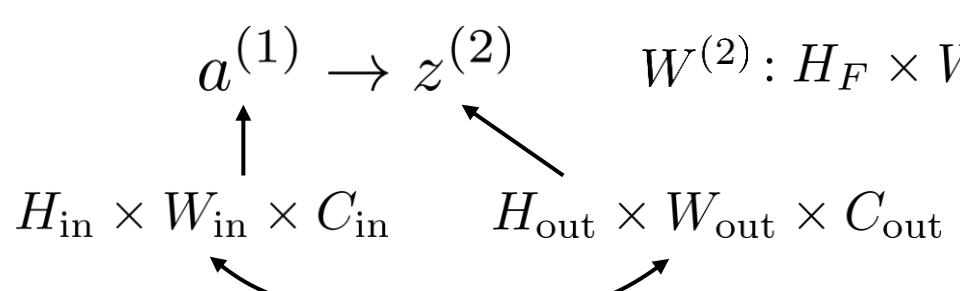
input image: HEIGHT \times WIDTH \times CHANNELS

filter: FLT.HEIGHT \times FLT.WIDTH \times OUTPUT CHAN \times INPUT CHAN

activations: HEIGHT \times WIDTH \times LAYER.CHANNELS



Convolutional layer in equations



equal or **almost** equal (more on this later)

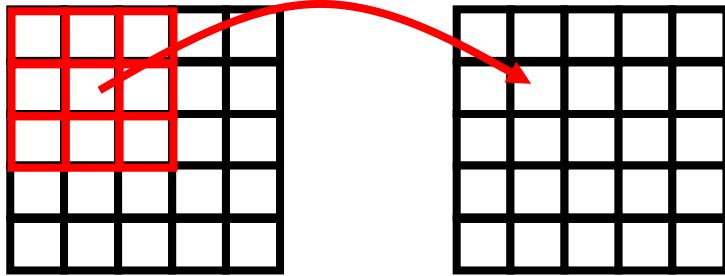
$$z^{(2)}[i, j, k] = \sum_{l=0}^{H_F-1} \sum_{m=0}^{H_W-1} \sum_{n=0}^{C_{\text{in}}-1} W^{(2)}[l, m, k, n] a^{(1)}[i + l - (H_F - 1)/2, j + m - (H_W - 1)/2, n]$$

$$z^{(2)}[i, j] = \sum_{l=0}^{H_F-1} \sum_{m=0}^{H_W-1} W^{(2)}[l, m] a^{(1)}[i + l - (H_F - 1)/2, j + m - (H_W - 1)/2]$$

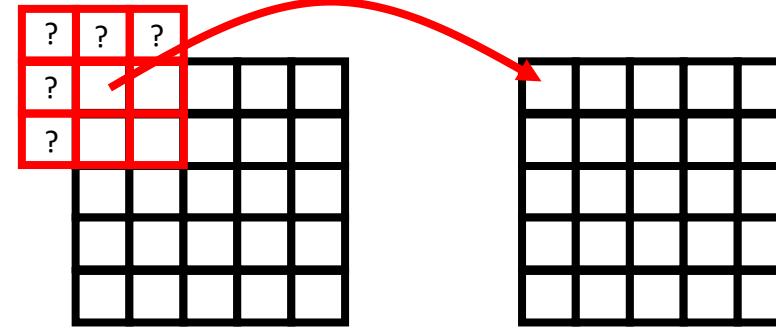
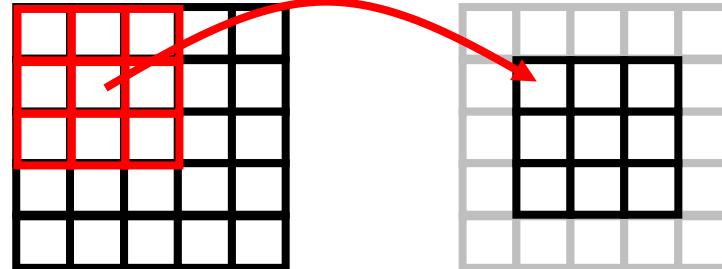
$$a^{(2)}[i, j, k] = \sigma(z^{(2)}[i, j, k]) \quad \text{Activation function applied per element, just like before}$$

Simple principle, but a bit complicated to write

Padding and edges



Option 1: cut off the edges



Problem: our activations shrink with every layer

Pop quiz:

input is $32 \times 32 \times 3$

filter is $5 \times 5 \times 6$

what is the output in this case?

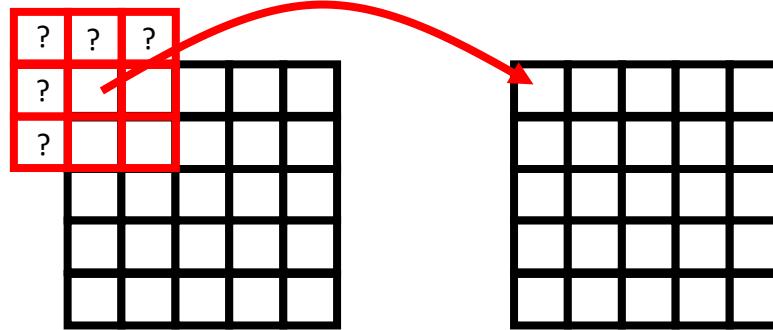
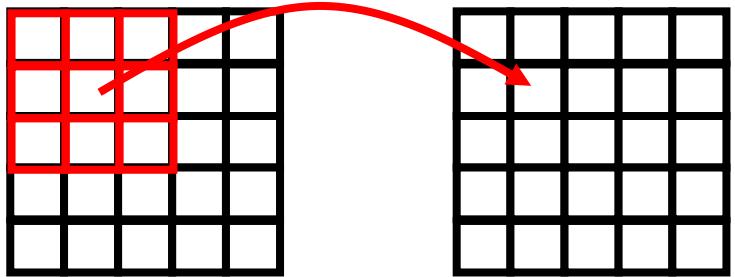
“radius” is $(H_F - 1)/2$ on each side = 2

$$H_{\text{out}} = H_{\text{in}} - ((H_F - 1)/2) \times 2 = 28$$

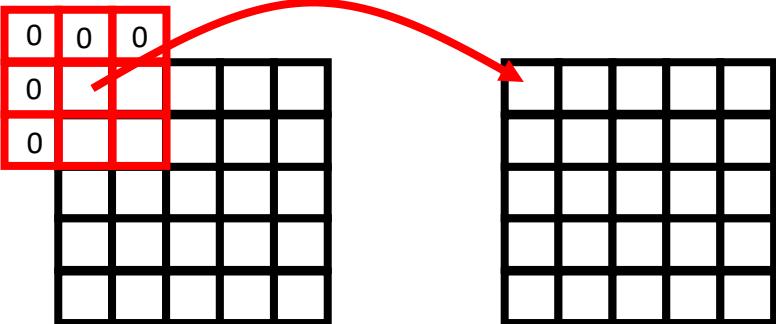
$$28 \times 28 \times 6$$

Some people don't like this

Padding and edges



Option 2: zero pad



Detail: remember to subtract the image mean first
(fancier contrast normalization often used in practice)

Advantage: simple, size is preserved

Disadvantage: weird effect at boundary
(this is usually not a problem, hence
why this method is so popular)

Strided convolutions

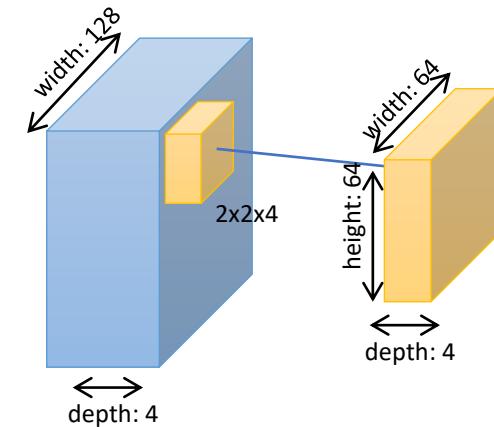
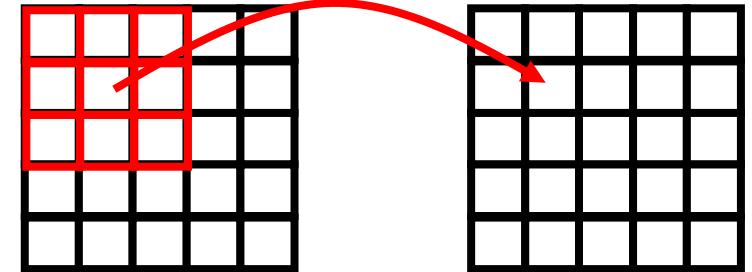
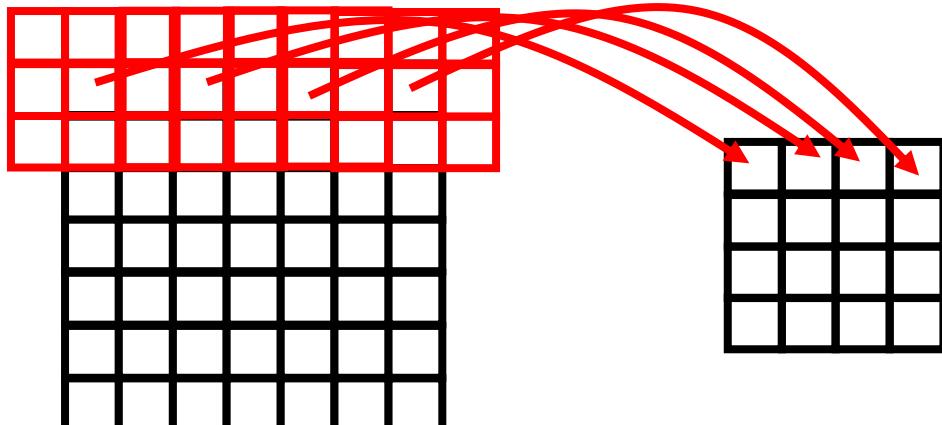
standard conv net structure at each layer:

1. Apply conv, $H \times W \times C_{\text{in}} \rightarrow H \times W \times C_{\text{out}}$
2. Apply activation func σ , $H \times W \times C_{\text{out}} \rightarrow H \times W \times C_{\text{out}}$
3. Apply pooling (width N), $H \times W \times C_{\text{out}} \rightarrow H/N \times W/N \times C_{\text{out}}$

this can be very expensive computationally

$C_{\text{out}} \times C_{\text{in}}$ matrix multiply at each position in $H \times W$ image!

Idea: what if skip over some positions?



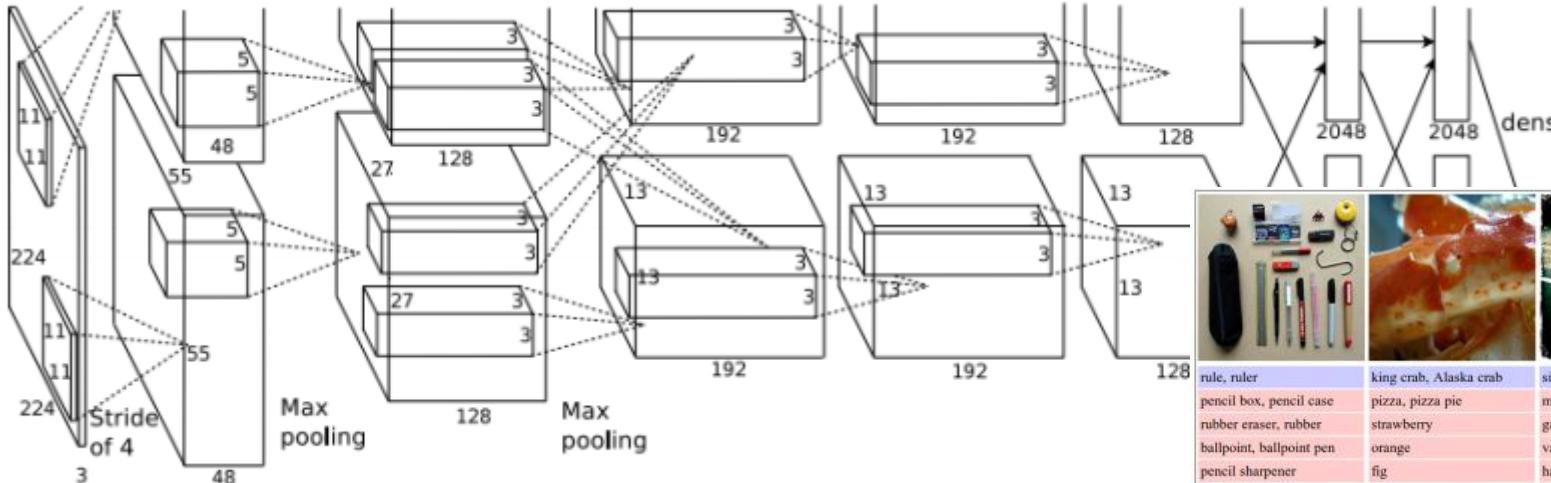
Amount of skipping is called the **stride**

Some people think that strided convolutions are just as good as conv + pooling

Examples of convolutional neural networks

AlexNet

[Krizhevsky et al. 2012]



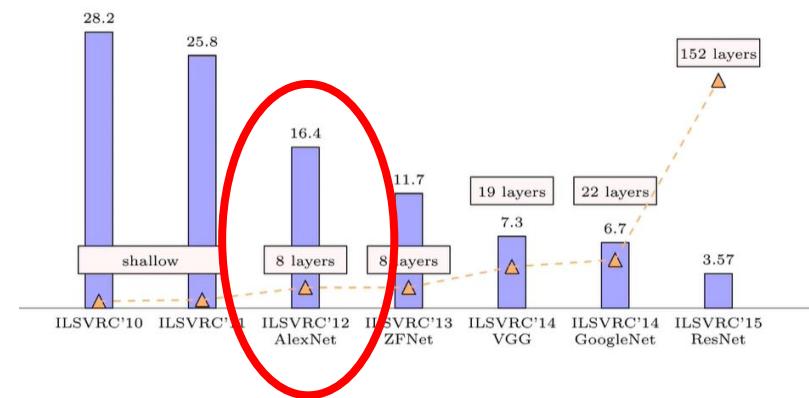
ILSVRC (ImageNet), 2009: 1.5 million images
1000 categories



pencil box, pencil case	pizza, pizza pie	maze, labyrinth	pill bottle	stethoscope	vase	schipperke
rubber eraser, rubber	strawberry	gar, garfish	water bottle	whistle	pitcher, ewer	groenendael
ballpoint, ballpoint pen	orange	valley, vale	lotion	ice lolly, lolly	coffeepot	doormat, welcome mat
pencil sharpener	fig	hammerhead	hair spray	hair spray	mask	teddy, teddy bear
carpenter's kit, tool kit	ice cream, icecream	sea snake	beer bottle	maypole	cup	jigsaw puzzle

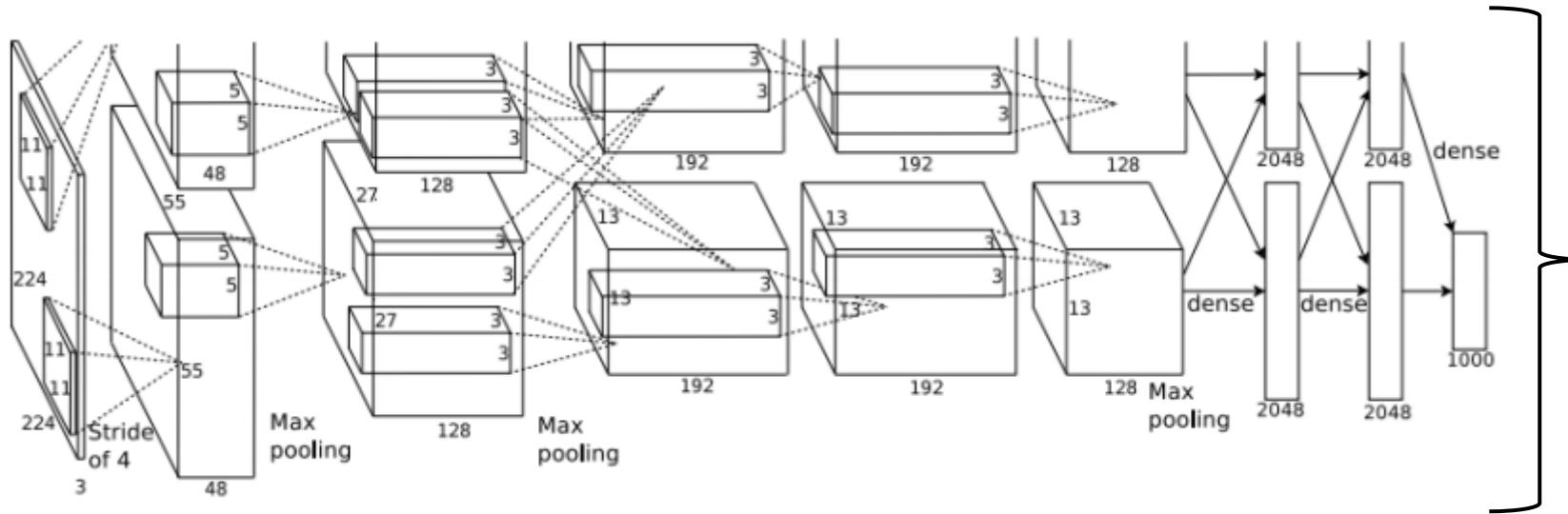
Why is this model important?

- “Classic” medium-depth convolutional network design (a bit like a modernized version of LeNet)
- Widely known for being the first neural network to attain state-of-the-art results on the ImageNet large-scale visual recognition challenge (ILSVRC)

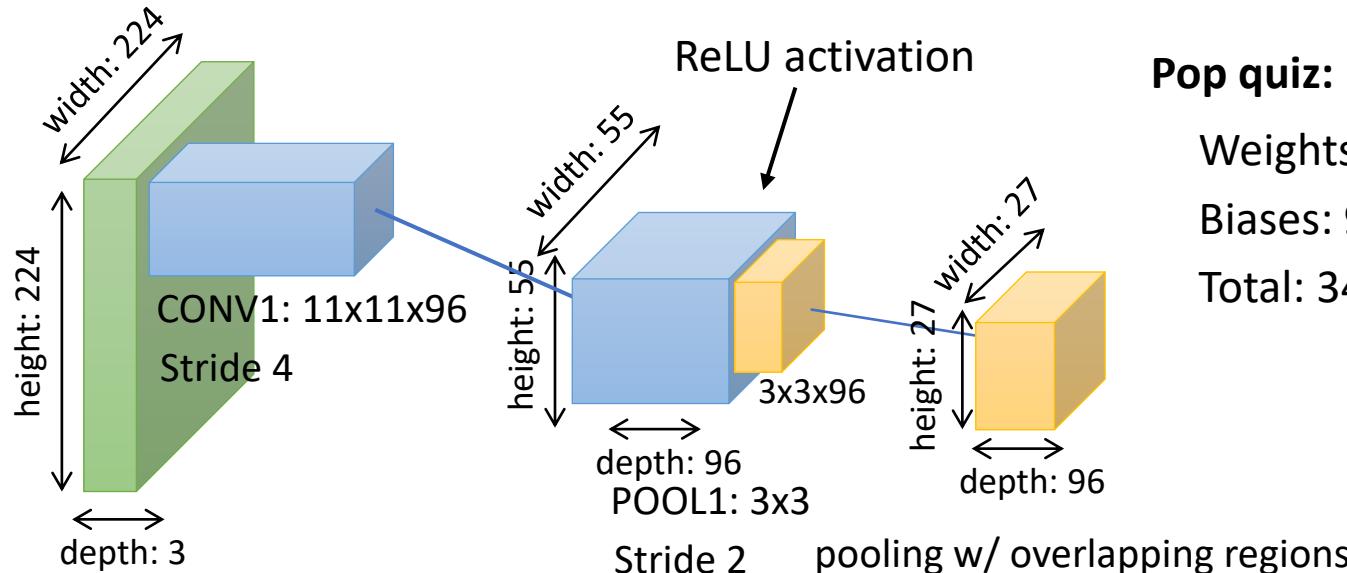


AlexNet

[Krizhevsky et al. 2012]



trained on two GPUs, hence
why the diagram is “split”
... we don't worry about this
sort of thing these days



Pop quiz: how many parameters in CONV1?

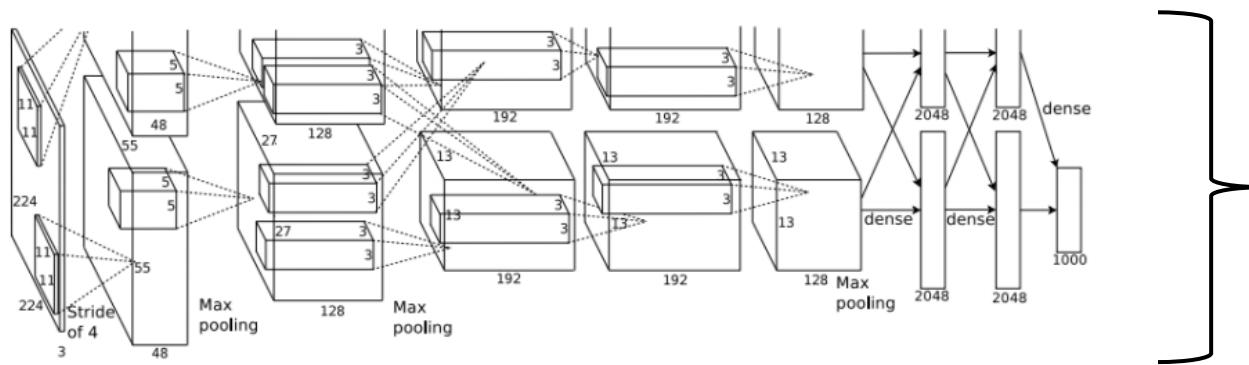
Weights: $11 \times 11 \times 3 \times 96 = 34,848$

Biases: 96

Total: 34,944

AlexNet

[Krizhevsky et al. 2012]



trained on two GPUs, hence
why the diagram is “split”

... we don't worry about this
sort of thing these days

CONV1: 11x11x96, Stride 4, maps 224x224x3 -> 55x55x96 [without zero padding]

POOL1: 3x3x96, Stride 2, maps 55x55x96 -> 27x27x96

NORM1: Local normalization layer [not widely used anymore, but we'll talk about normalization later]

CONV2: 5x5x256, Stride 1, maps 27x27x96 -> 27x27x256 [**with** zero padding]

POOL2: 3x3x256, Stride 2, maps 27x27x256 -> 13x13x256

NORM2: Local normalization layer

CONV3: 3x3x384, Stride 1, maps 13x13x256 -> 13x13x384 [**with** zero padding]

CONV4: 3x3x384, Stride 1, maps 13x13x384 -> 13x13x384 [**with** zero padding]

CONV5: 3x3x384, Stride 1, maps 13x13x384 -> 13x13x256 [**with** zero padding]

POOL3: 3x3x256, Stride 2, maps 13x13x256 -> 6x6x256

FC6: 6x6x256 -> 9,216 -> 4,096 [matrix is 4,096 x 9,216]

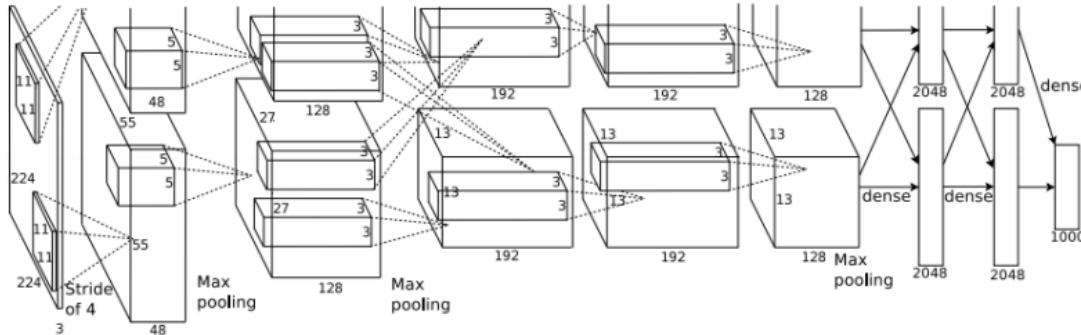
FC7: 4,096 -> 4,096

FC8: 4,096 -> 1,000

SOFTMAX

AlexNet

[Krizhevsky et al. 2012]



CONV1: 11x11x96, Stride 4, maps 224x224x3 -> 55x55x96 [without zero padding]

POOL1: 3x3x96, Stride 2, maps 55x55x96 -> 27x27x96

NORM1: Local normalization layer

CONV2: 5x5x256, Stride 1, maps 27x27x96 -> 27x27x256 [**with** zero padding]

POOL2: 3x3x256, Stride 2, maps 27x27x256 -> 13x13x256

NORM2: Local normalization layer

CONV3: 3x3x384, Stride 1, maps 13x13x256 -> 13x13x384 [**with** zero padding]

CONV4: 3x3x384, Stride 1, maps 13x13x384 -> 13x13x384 [**with** zero padding]

CONV5: 3x3x256, Stride 1, maps 13x13x256 -> 13x13x256 [**with** zero padding]

POOL3: 3x3x256, Stride 2, maps 13x13x256 -> 6x6x256

FC6: 6x6x256 -> 9,216 -> 4,096 [matrix is 4,096 x 9,216]

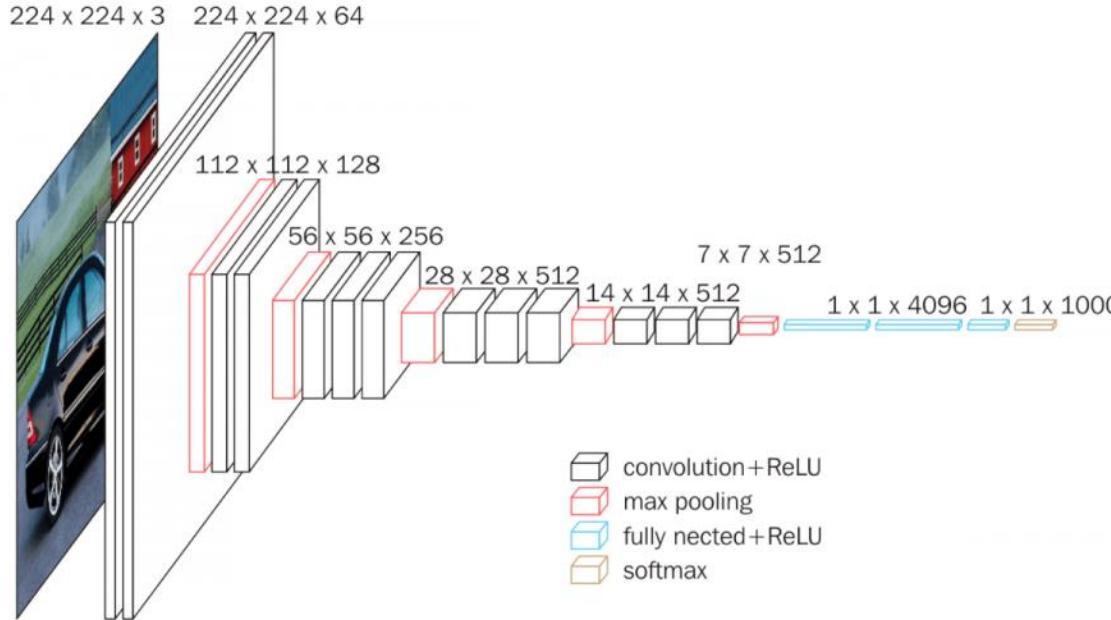
FC7: 4,096 -> 4,096

FC8: 4,096 -> 1,000

SOFTMAX

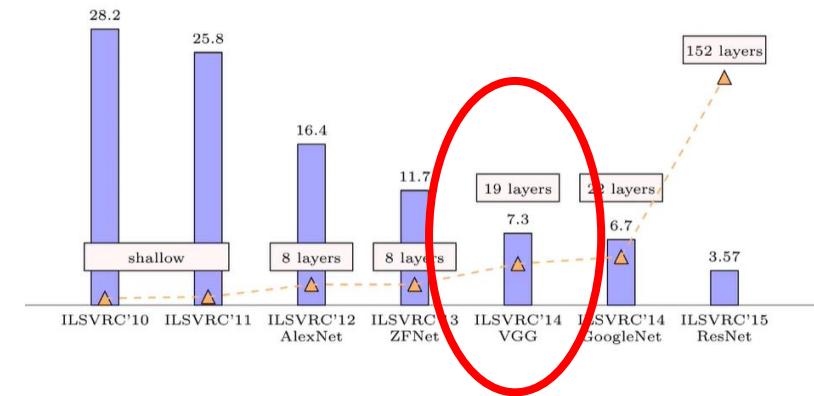
- Don't forget: ReLU nonlinearities after every CONV or FC layer (except the last one!)
- Trained with regularization (we'll learn about these later):
 - Data augmentation
 - Dropout
- Local normalization (not used much anymore, but there are other types of normalization we do use)

VGG



Why is this model important?

- Still often used today
- Big increase in **depth** over previous best model
- Start seeing “homogenous” stacks of multiple convolutions interspersed with resolution reduction



VGG

CONV: 3x3x64, maps 224x224x3 -> 224x224x64

CONV: 3x3x64, maps 224x224x64 -> 224x224x64

POOL: 2x2, maps 224x224x64 -> 112x112x64

CONV: 3x3x128, maps 112x112x64 -> 112x112x128

CONV: 3x3x128, maps 112x112x128 -> 112x112x128

POOL: 2x2, maps 112x112x128 -> 56x56x128

CONV: 3x3x256, maps 56x56x128 -> 56x56x256

CONV: 3x3x256, maps 56x56x256 -> 56x56x256

CONV: 3x3x256, maps 56x56x256 -> 56x56x256

POOL: 2x2, maps 56x56x256 -> 28x28x256

CONV: 3x3x512, maps 28x28x256 -> 28x28x512

CONV: 3x3x512, maps 28x28x512 -> 28x28x512

CONV: 3x3x512, maps 28x28x512 -> 28x28x512

POOL: 2x2, maps 28x28x512 -> 14x14x512

CONV: 3x3x512, maps 14x14x512 -> 14x14x512

CONV: 3x3x512, maps 14x14x512 -> 14x14x512

CONV: 3x3x512, maps 14x14x512 -> 14x14x512

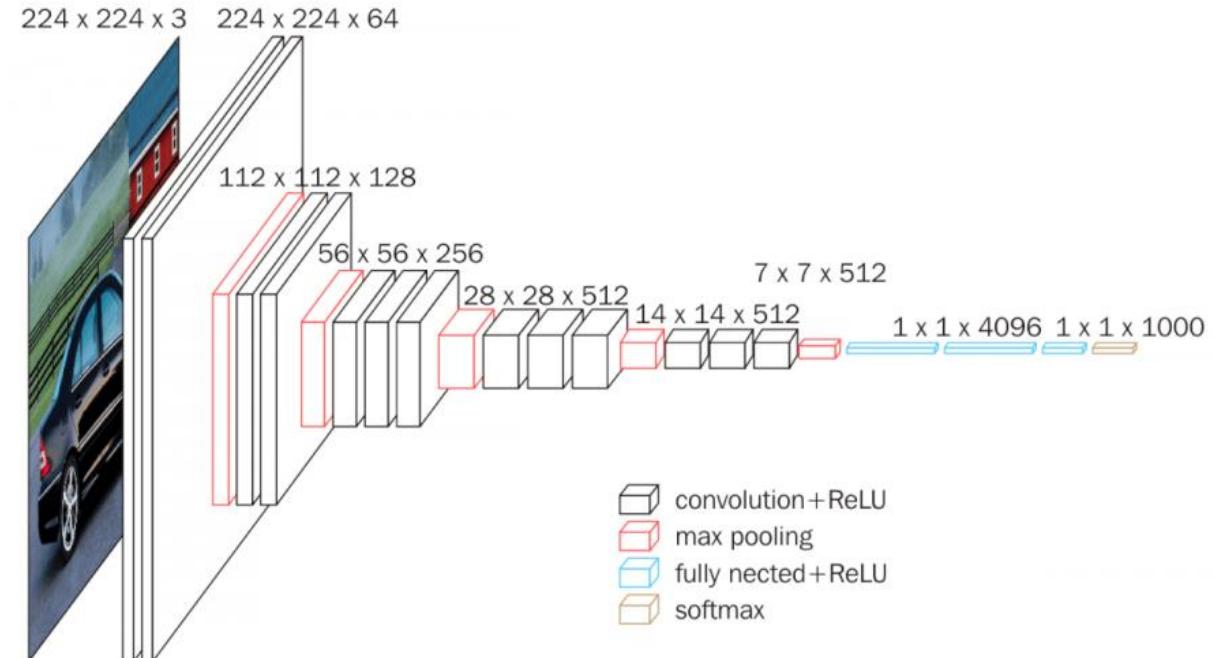
POOL: 2x2, maps 14x14x512 -> 7x7x512

FC: 7x7x512 -> 25,088 -> 4,096 ← almost all parameters are here

FC: 4,096 -> 4,096

FC: 4,096 -> 1,000

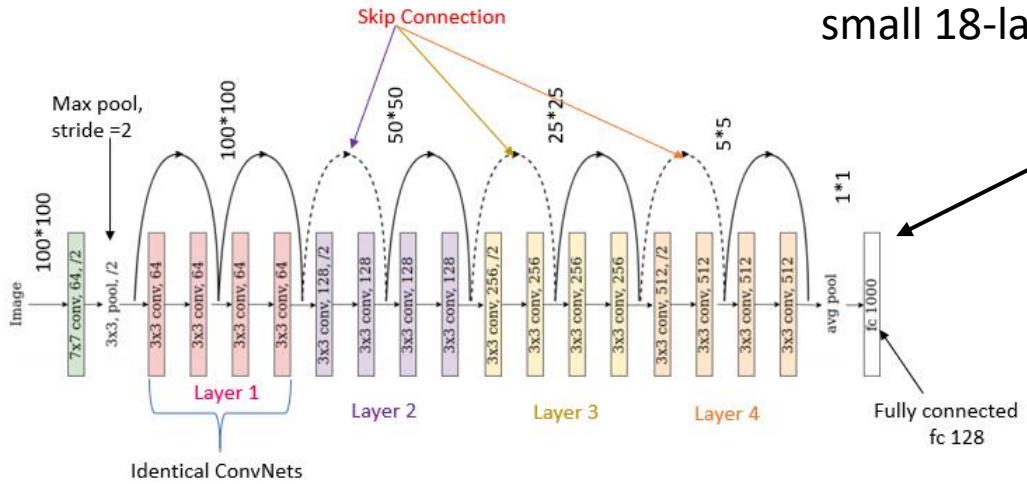
SOFTMAX



- More layers = more processing, which is why we see repeated blocks
- Which parts use the most memory?
- Which parts have the most parameters?

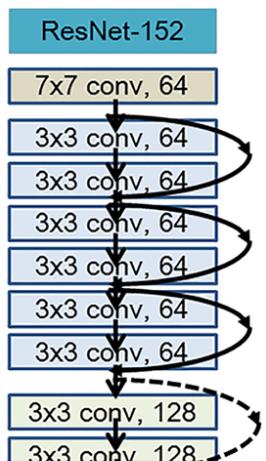
ResNet

152 layers!

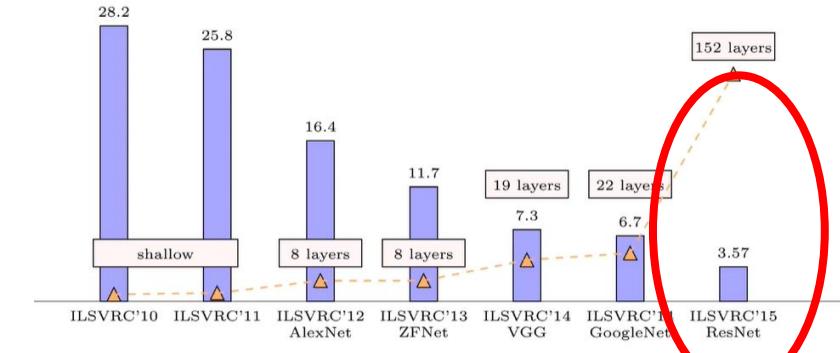
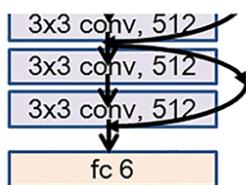


small 18-layer prototype (ResNet-18)

don't bother with huge FC layer at the end, just average pool over all positions and have one linear layer



152 layers



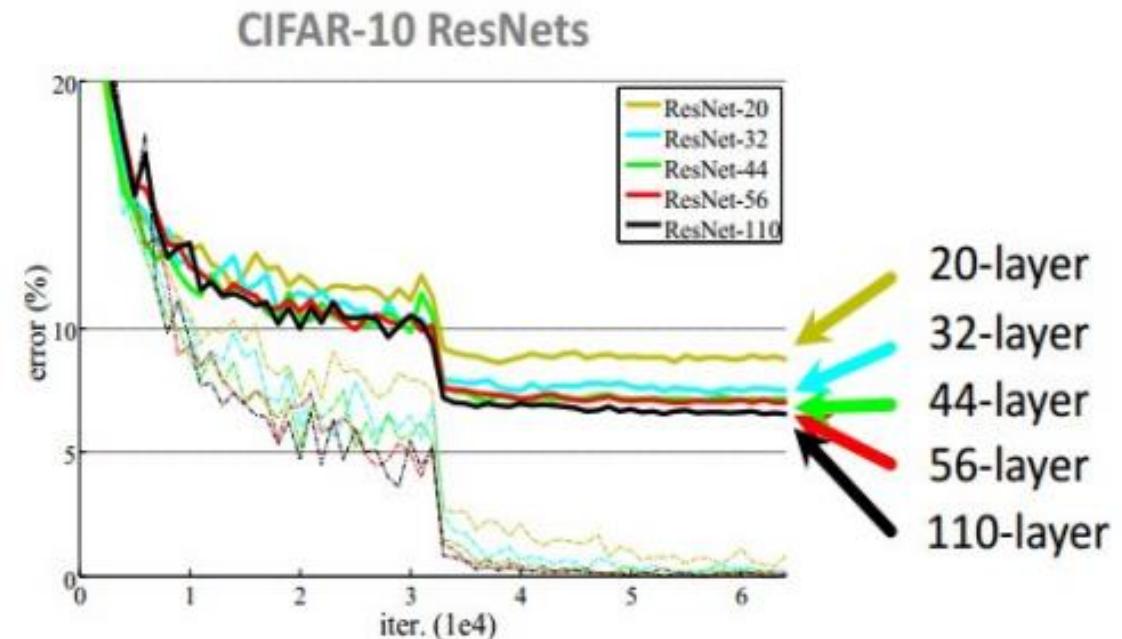
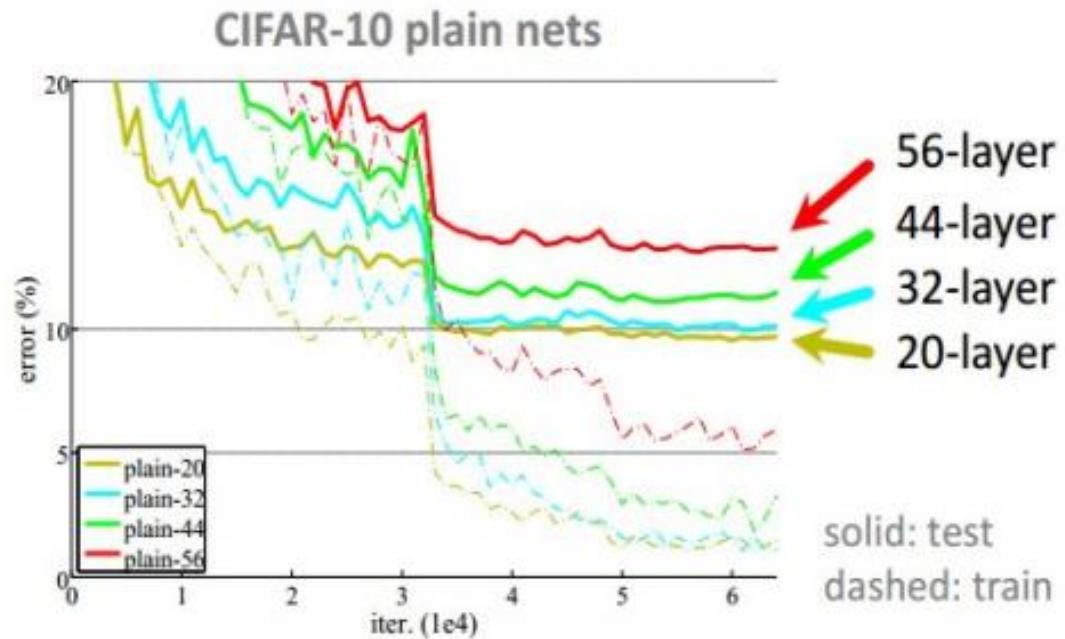
AlexNet, 8 layers
(ILSVRC 2012)

VGG, 19 layers
(ILSVRC 2014)

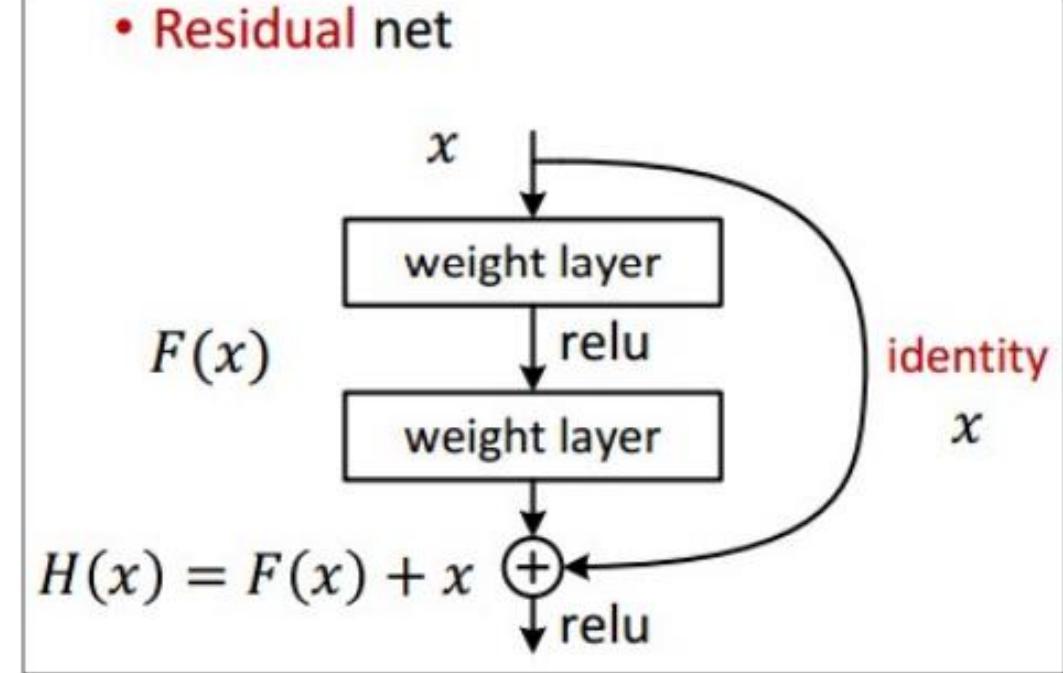
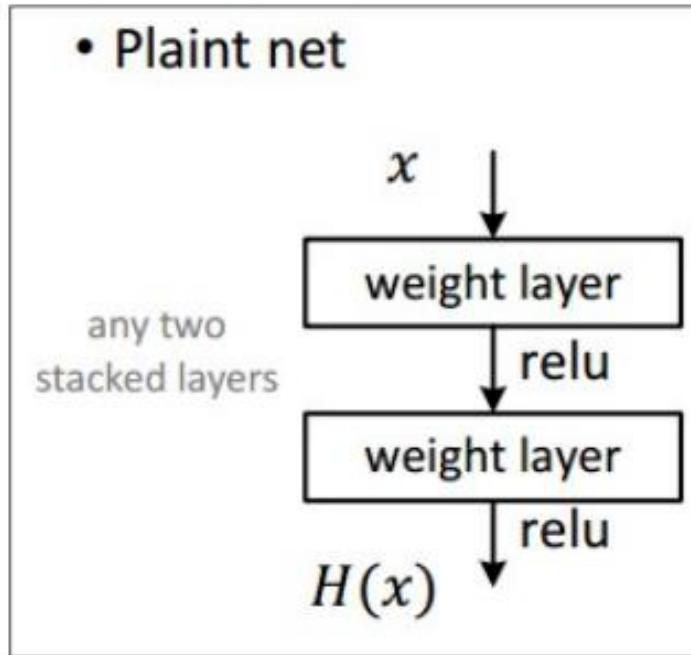
ResNet, 152 layers
(ILSVRC 2015)

ResNet

CIFAR-10 experiments



What's the main idea?



Why is this a good idea?

Why are deep networks hard to train?

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}}$$

$$\frac{d\mathcal{L}}{dW^{(1)}} = J_1 J_2 J_3 \dots J_n \frac{d\mathcal{L}}{dz^{(n)}}$$

If we multiply many many numbers together, what will we get?

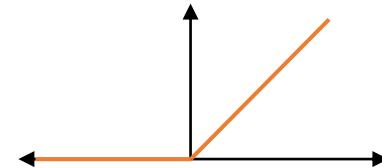
If most of the numbers are < 1 , we get 0

If most of the numbers are > 1 , we get infinity

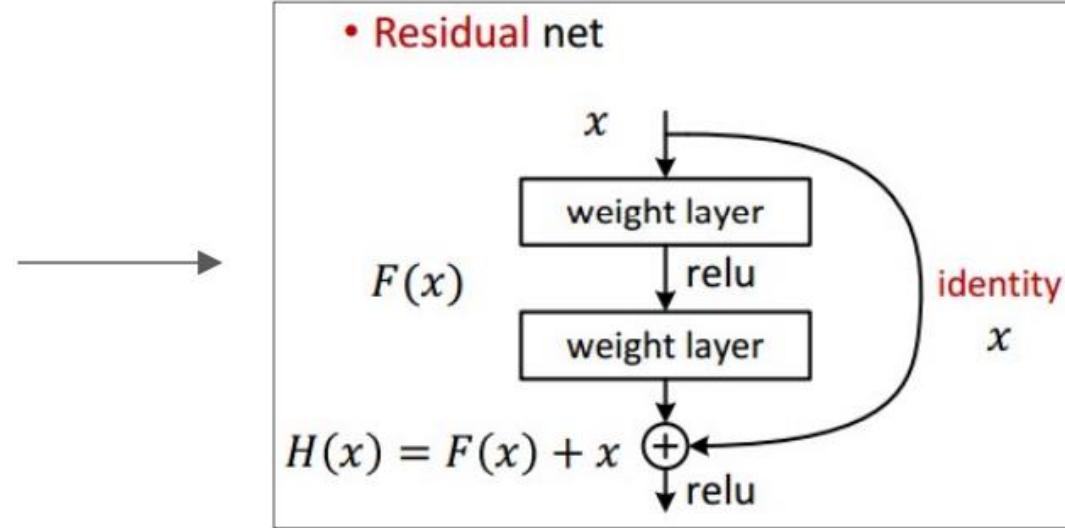
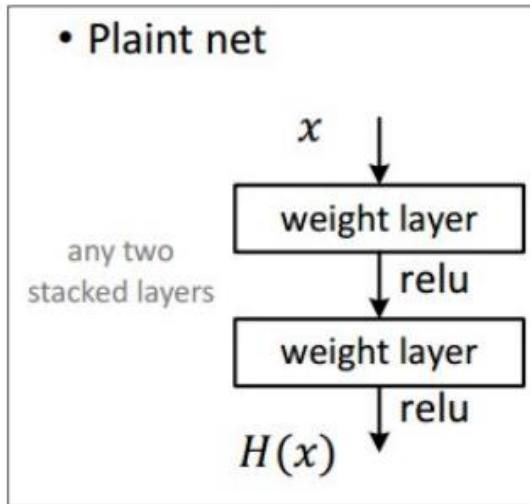
We only get a reasonable answer if the numbers are all close to 1!

For matrices, this means we want $J_i \approx \mathbf{I}$

$$\text{ReLU: } \left(\frac{df}{dz} \right)_i = \text{Ind}(z_i \geq 0)$$



So why is this a good idea?



$$\frac{dH}{dx}$$

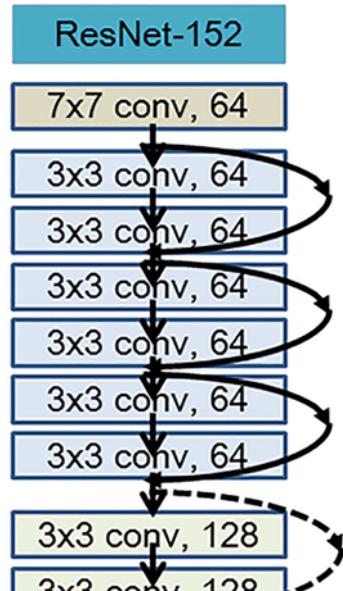
could be **big** or **small**
not close to \mathbf{I}

$$\frac{dH}{dx} = \frac{dF}{dx} + \mathbf{I}$$

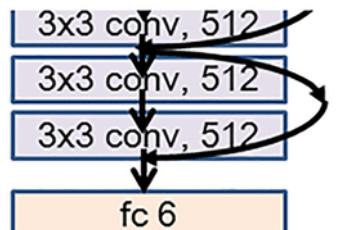
If weights are not too big,
this will be small(ish)



ResNet



152 layers



- “Generic” blocks with many layers, interspersed with a few pooling operations
- No giant FC layer at the end, just mean pool over all x/y positions and a small(ish) FC layer to go into the softmax
- Residual layers to provide for good gradient flow

Getting Neural Nets to Train

Designing, Visualizing and Understanding Deep Neural Networks

CS W182/282A

Instructor: Sergey Levine
UC Berkeley



This lecture

Help! My network doesn't train

If you follow everything I described in the previous lectures...

And you implement everything correctly...

And you train everything for a long time...

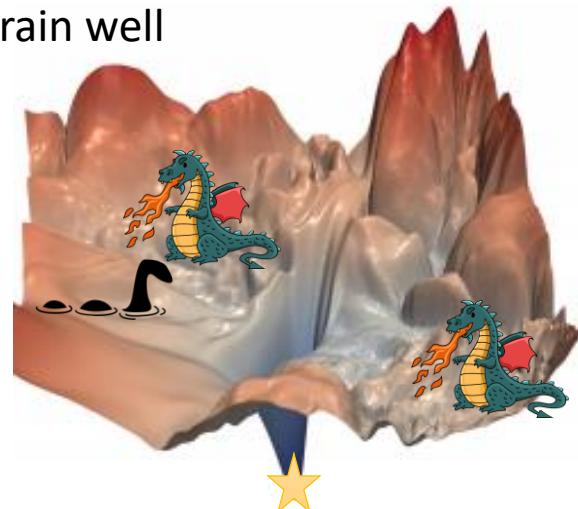
There is a good chance it **still** won't work

Neural networks are messy

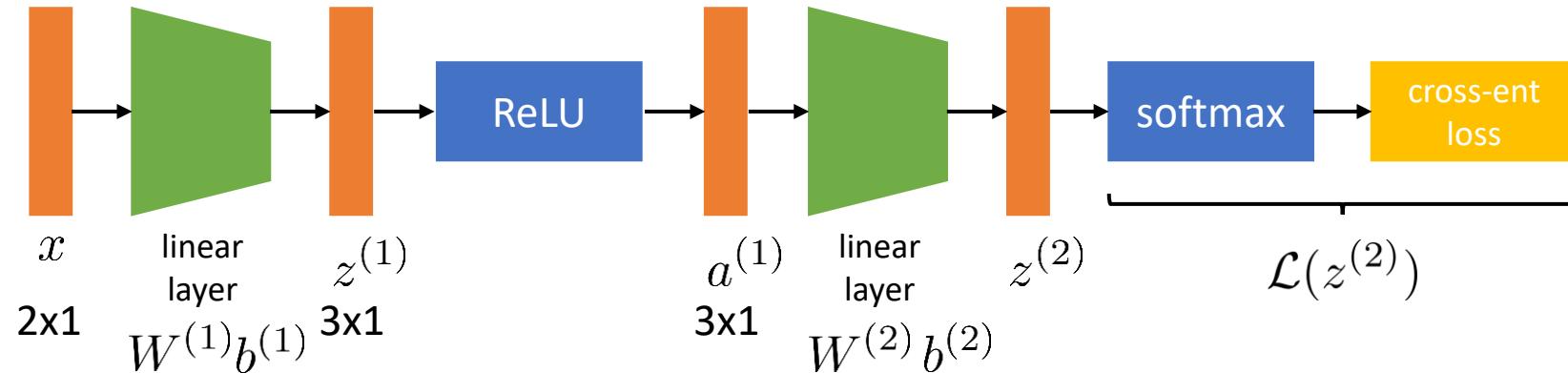
They require lots of "tricks" to train well

We'll discuss these tricks today

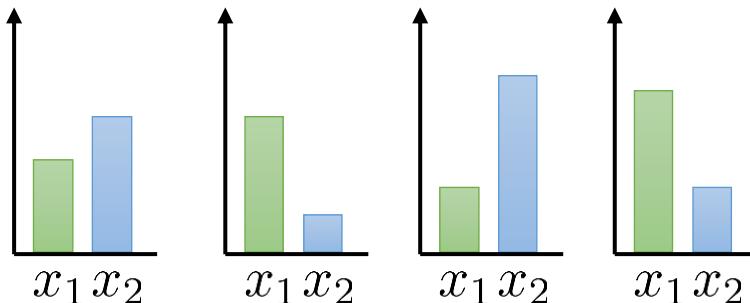
- Normalizing inputs and outputs
- Normalizing **activations** (batch normalization)
- Initialization of weight matrices & bias vectors
- Gradient clipping
- Best practices for hyperparameter optimization
- Ensembling, dropout



The dangers of big inputs, activations, and outputs

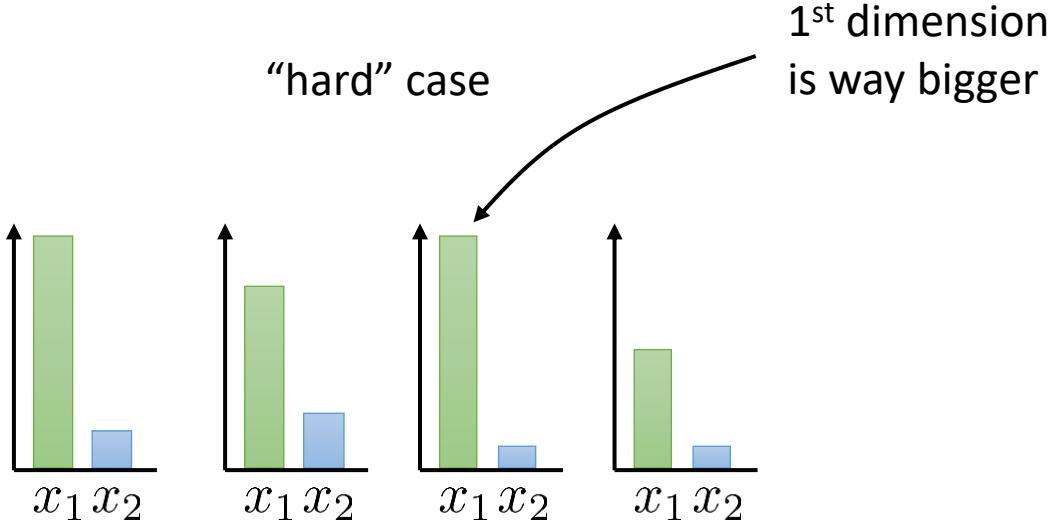


“easy” case



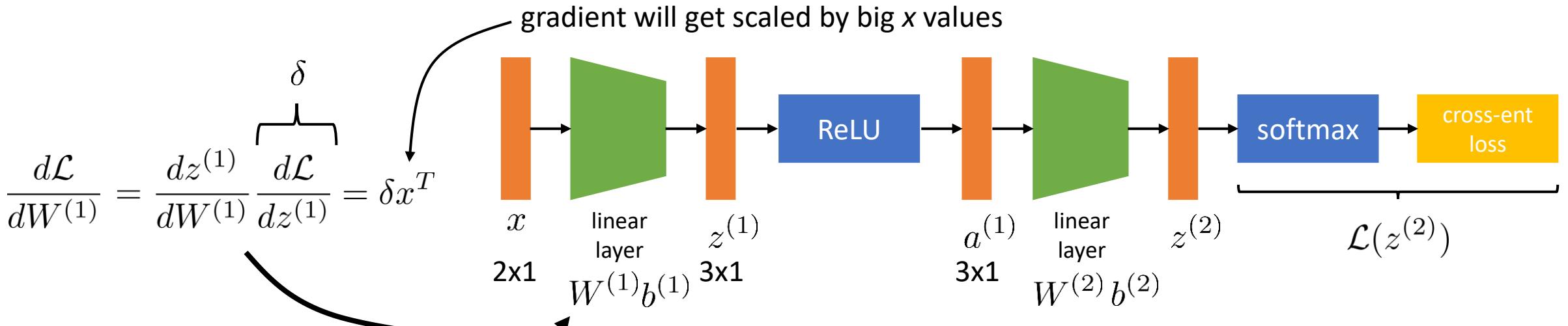
Why?

“hard” case

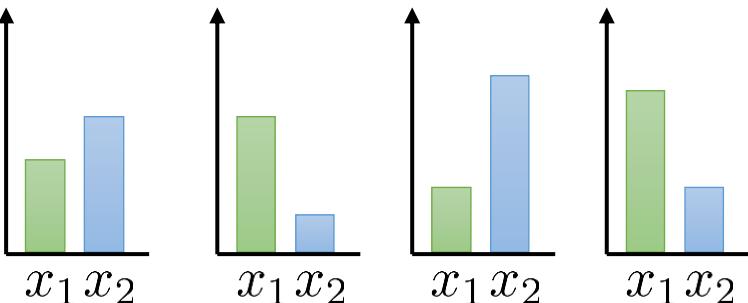


1st dimension
is way bigger

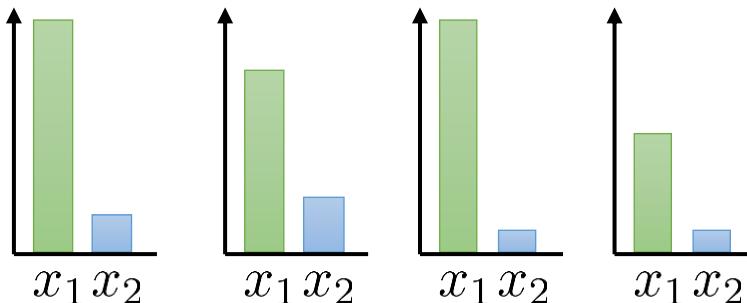
The dangers of big inputs, activations, and outputs



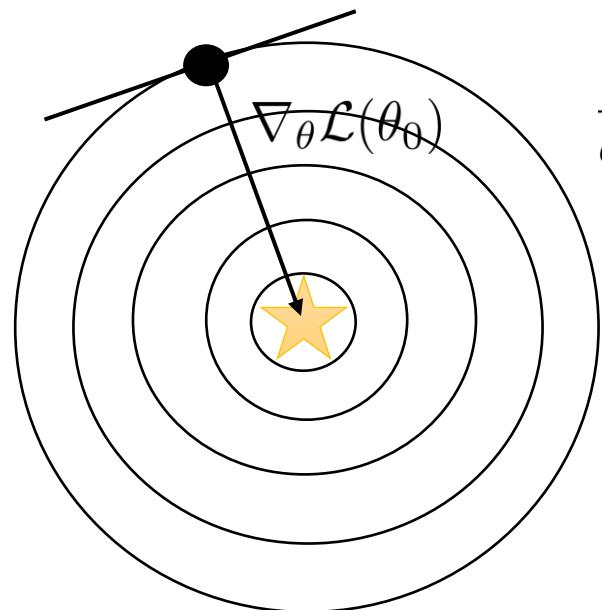
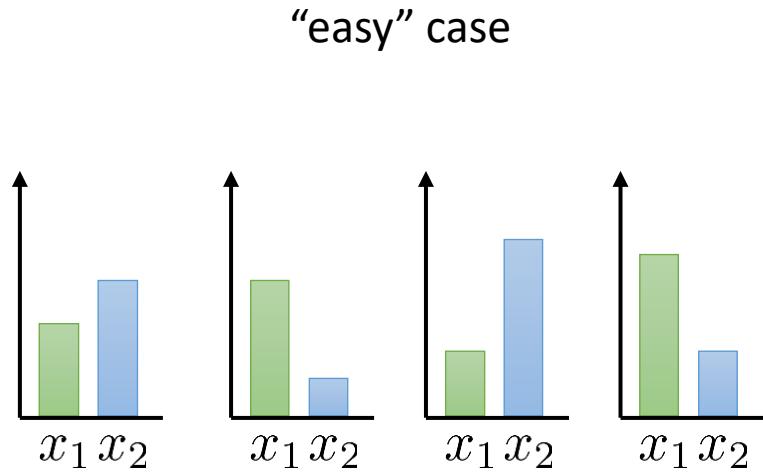
“easy” case



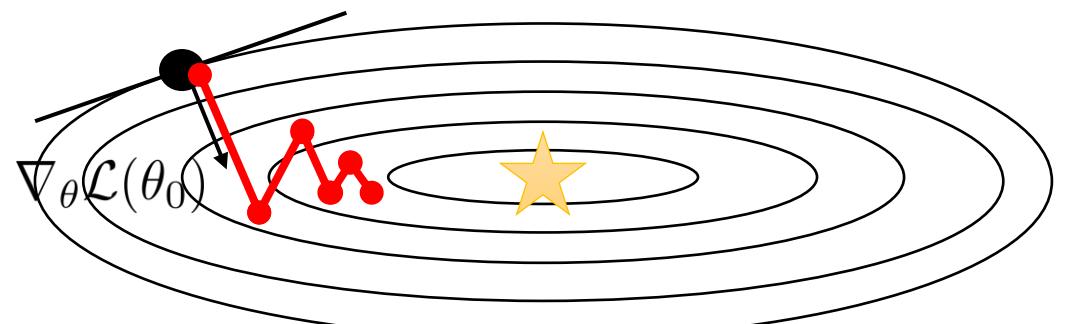
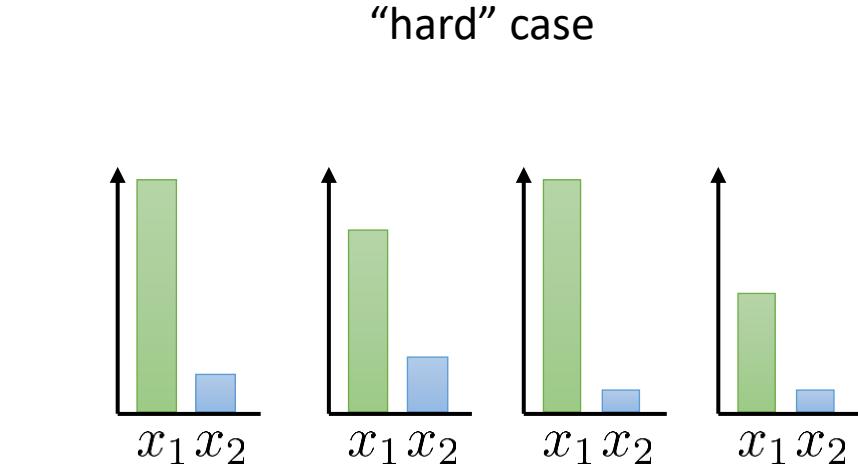
“hard” case



The dangers of big inputs, activations, and outputs



$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{d\mathcal{L}}{dz^{(1)}} = \delta x^T$$



The dangers of big inputs, activations, and outputs

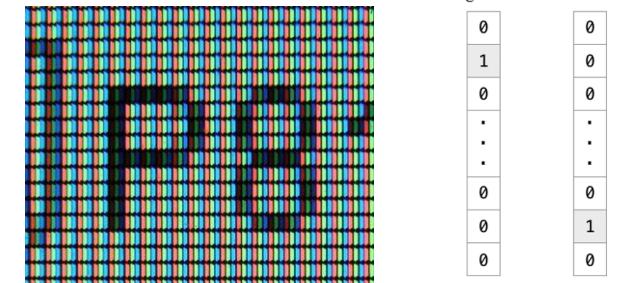
In **general**...

we really want all entries in x to be roughly on the same scale

sometimes not a problem:

images: all pixels are roughly in $[0, 1]$ or $\{0, \dots, 255\}$

discrete inputs (e.g., NLP): all inputs are one-hot (zero or one)



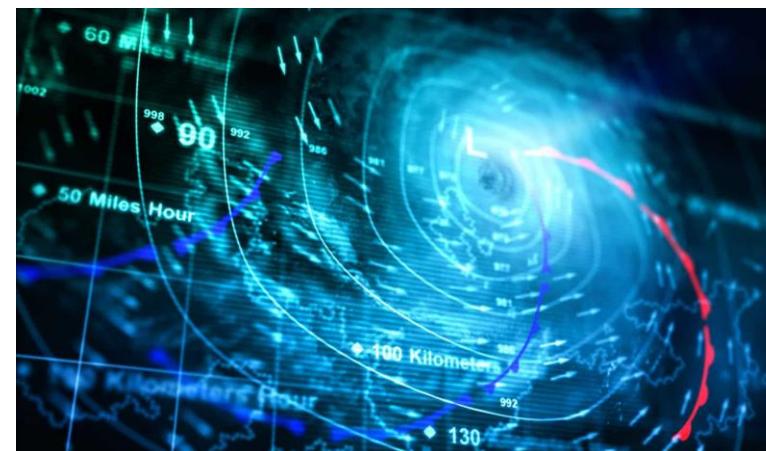
sometimes a **huge** problem:

forecasting the weather?

temperature: somewhere 40-100?

humidity: somewhere 0.3 - 0.6?

etc.



The dangers of big inputs, activations, and outputs

What can we do?

and outputs! (if doing regression)
...and activations??

Standardization: transform inputs so they have $\mu = 0, \sigma = 1$



To make $\mu = 0$: $\bar{x}_i = x_i - E[x]$ $E[x] \approx \frac{1}{N} \sum_{i=1}^N x_i$ all operations are per-dimension

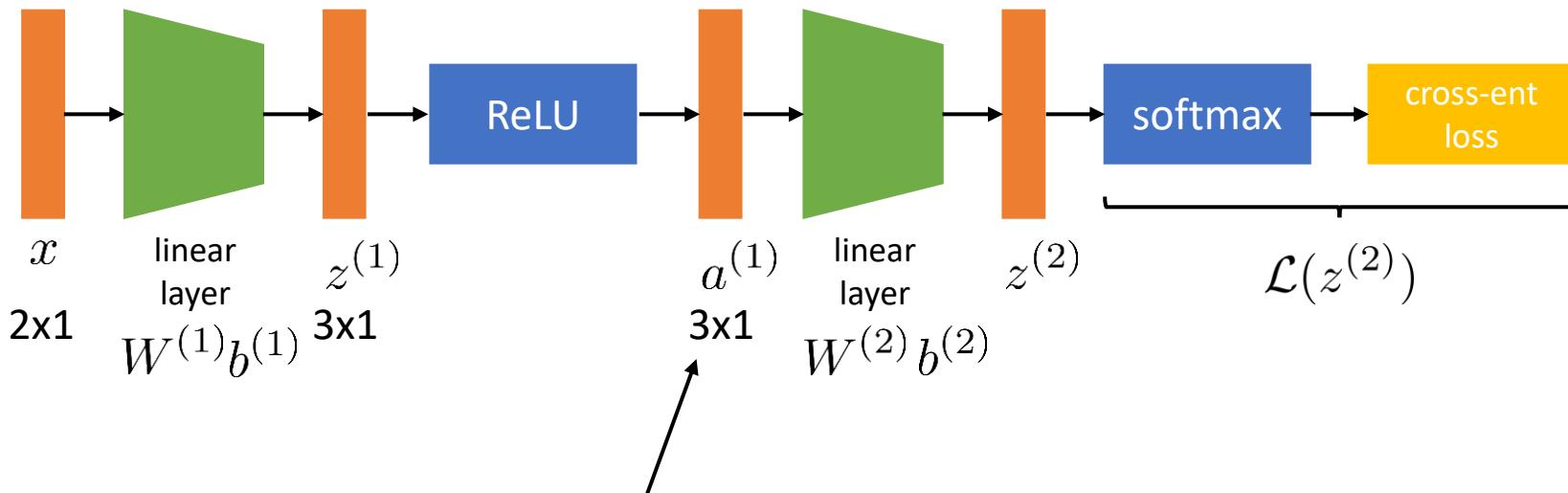
To *also* make $\sigma = 1$: $\bar{x}_i = \frac{x_i - E[x]}{\sqrt{E[(x_i - E[x])^2]}}$ ← standard deviation

Standardizing activations?

What can we do?

and outputs! (if doing regression)
...and activations??

Standardization: transform inputs so they have $\mu = 0, \sigma = 1$



what if we start getting really different scales for each dimension **here?**

can we just standardize these activations too?

basically yes, but now the mean and standard deviation changes during training...

Standardizing activations?

Basic idea:

$$z_i^{(1)} = W^{(1)}x_i + b^{(1)}$$

$$a_i^{(1)} = \text{ReLU}(z_i^{(1)})$$

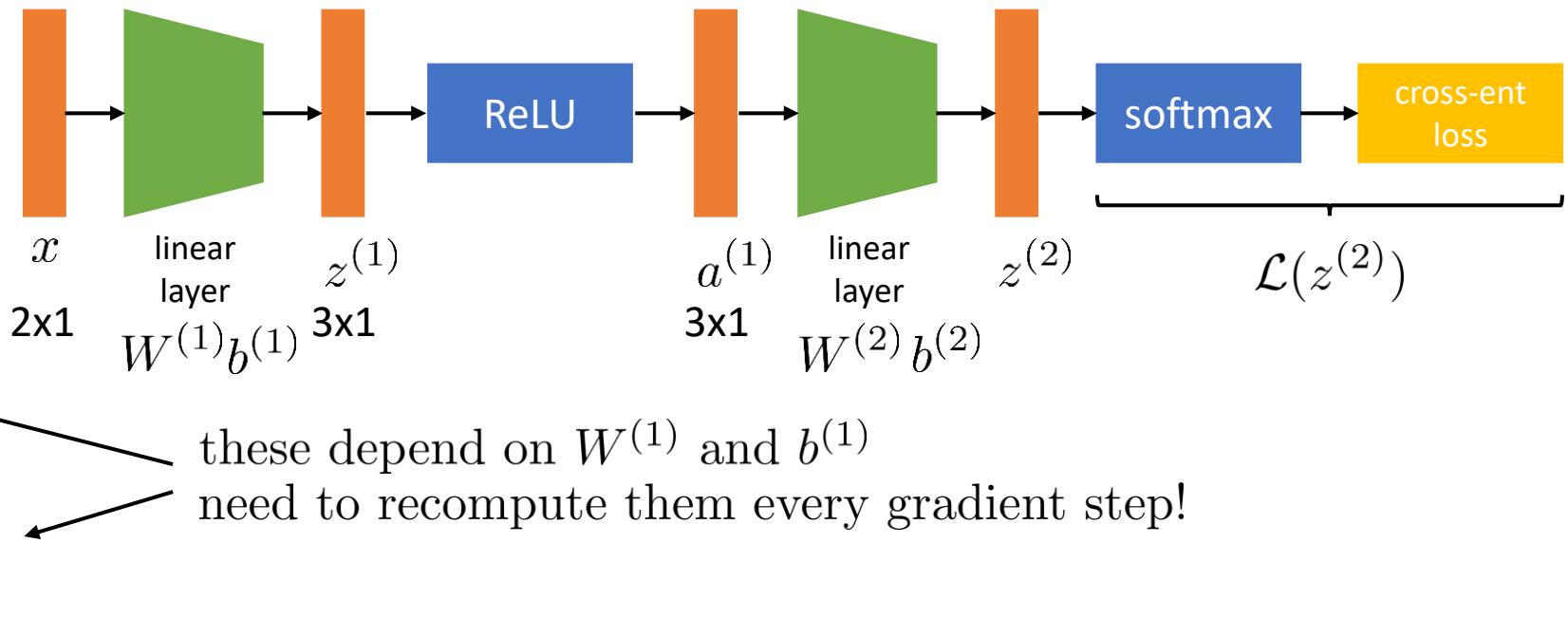
$$\mu^{(1)} = \frac{1}{N} \sum_{i=1}^N a_i^{(1)}$$

$$\sigma^{(1)} = \sqrt{\frac{1}{N} \sum_{i=1}^N (a_i^{(1)} - \mu^{(1)})^2}$$

$$\bar{a}_i^{(1)} = \frac{a_i^{(1)} - \mu^{(1)}}{\sigma^{(1)}}$$

$$z_i^{(2)} = W^{(2)}\bar{a}_i^{(1)} + b^{(2)}$$

etc...



This seems very expensive, since we don't want to evaluate all points in the dataset every gradient step

Batch normalization (basic version)

Basic idea:

$$z_i^{(1)} = W^{(1)}x_i + b^{(1)}$$

$$a_i^{(1)} = \text{ReLU}(z_i^{(1)})$$

$$\mu^{(1)} = \frac{1}{N} \sum_{i=1}^N a_i^{(1)}$$

$$\sigma^{(1)} = \sqrt{\frac{1}{N} \sum_{i=1}^N (a_i^{(1)} - \mu^{(1)})^2}$$

$$\bar{a}_i^{(1)} = \frac{a_i^{(1)} - \mu^{(1)}}{\sigma^{(1)}}$$

$$z_i^{(2)} = W^{(2)}\bar{a}_i^{(1)} + b^{(2)}$$

etc...

This seems very expensive, since we don't want to evaluate all points in the dataset every gradient step

$$\left. \begin{aligned} \mu^{(1)} &\approx \frac{1}{B} \sum_{j=1}^B a_{i_j}^{(1)} \\ \sigma^{(1)} &= \sqrt{\frac{1}{B} \sum_{j=1}^B (a_{i_j}^{(1)} - \mu^{(1)})^2} \end{aligned} \right\} \text{compute mean and std only over the current batch}$$

Batch normalization (real version)

Basic idea:

$$z_i^{(1)} = W^{(1)}x_i + b^{(1)}$$

$$a_i^{(1)} = \text{ReLU}(z_i^{(1)})$$

$$\mu^{(1)} = \frac{1}{N} \sum_{i=1}^N a_i^{(1)}$$

$$\sigma^{(1)} = \sqrt{\frac{1}{N} \sum_{i=1}^N (a_i^{(1)} - \mu^{(1)})^2}$$

$$\bar{a}_i^{(1)} = \frac{a_i^{(1)} - \mu^{(1)}}{\sigma^{(1)}}$$

$$z_i^{(2)} = W^{(2)}\bar{a}_i^{(1)} + b^{(2)}$$

etc...

This seems very expensive, since we don't want to evaluate all points in the dataset every gradient step

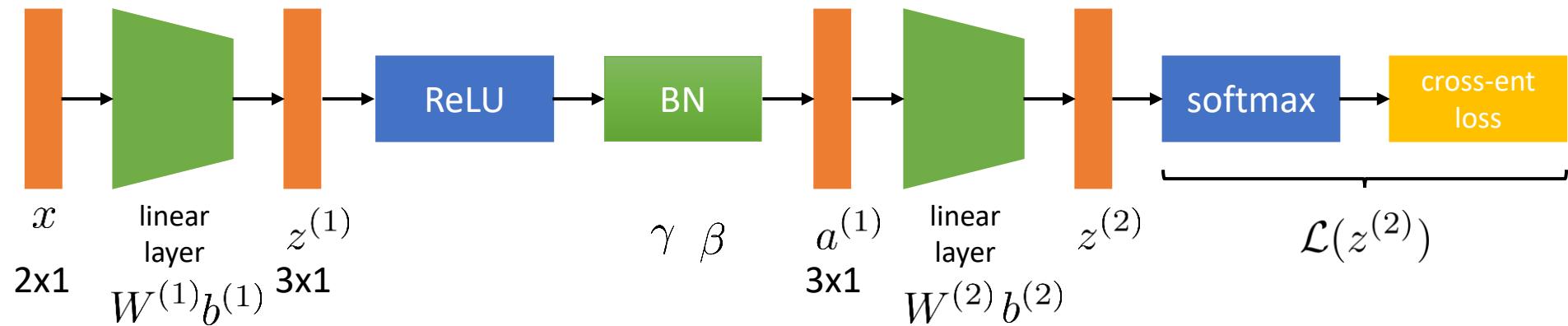
$$\left. \begin{array}{l} \mu^{(1)} \approx \frac{1}{B} \sum_{j=1}^B a_{i_j}^{(1)} \\ \sigma^{(1)} \approx \sqrt{\frac{1}{B} \sum_{j=1}^B (a_{i_j}^{(1)} - \mu^{(1)})^2} \end{array} \right\} \text{compute mean and std only over the current batch}$$

$$\bar{a}_i^{(1)} = \frac{a_i^{(1)} - \mu^{(1)}}{\sigma^{(1)}} \gamma + \beta$$



learnable scale and bias
same dim as $\bar{a}_i^{(1)}$

Batch normalization “layer”



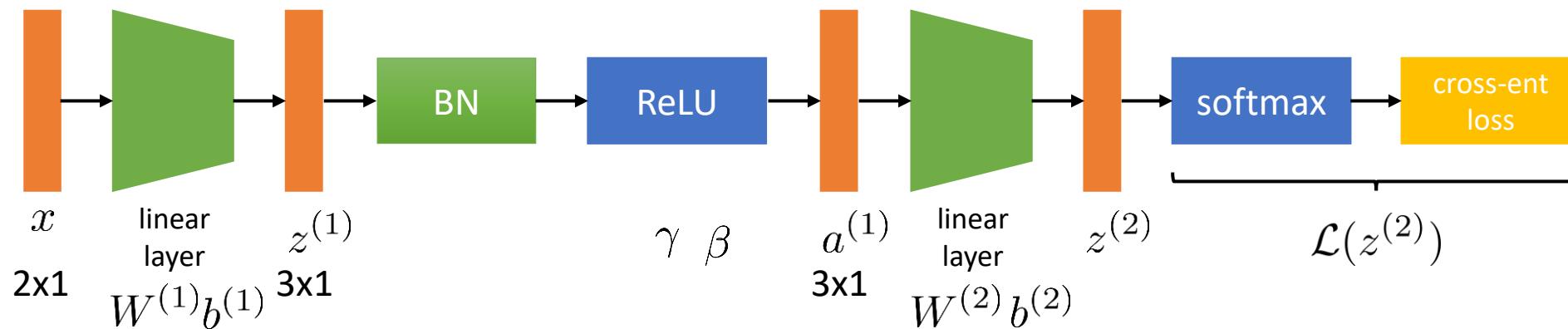
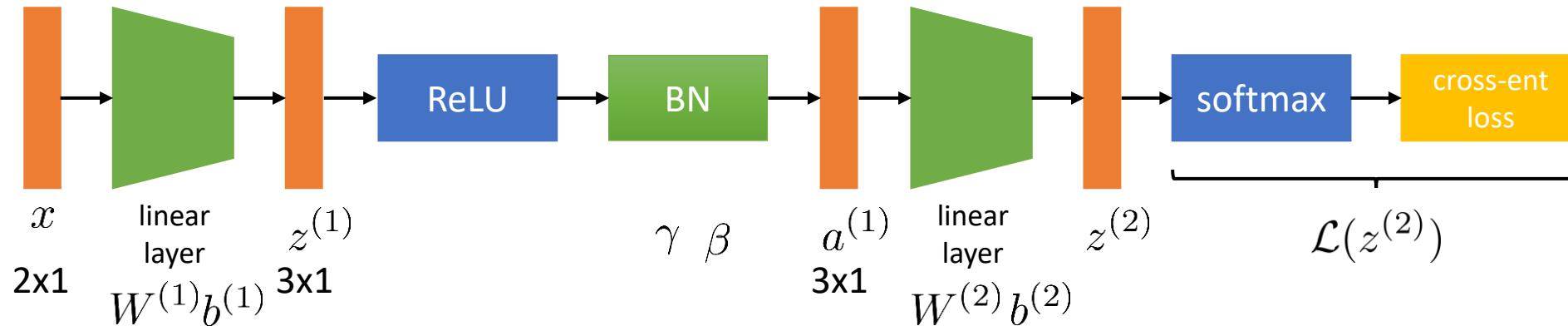
$$\mu^{(1)} \approx \frac{1}{B} \sum_{j=1}^B a_{i_j}^{(1)} \quad \sigma^{(1)} \approx \sqrt{\frac{1}{B} \sum_{j=1}^B (a_{i_j}^{(1)} - \mu^{(1)})^2} \quad \bar{a}_i^{(1)} = \frac{a_i^{(1)} - \mu^{(1)}}{\sigma^{(1)}} \gamma + \beta$$

How to train?

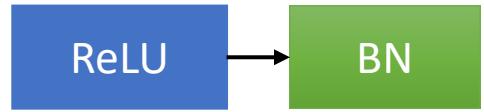
Just use backpropagation!

Exercise: figure out the derivatives w.r.t. parameters and input!

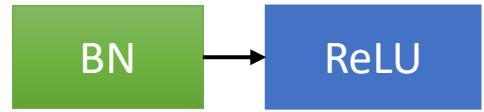
Where to put back normalization?



Where to put back normalization?



- Scale and bias seemingly should be subsumed by next linear layer?
- All ReLU outputs are positive

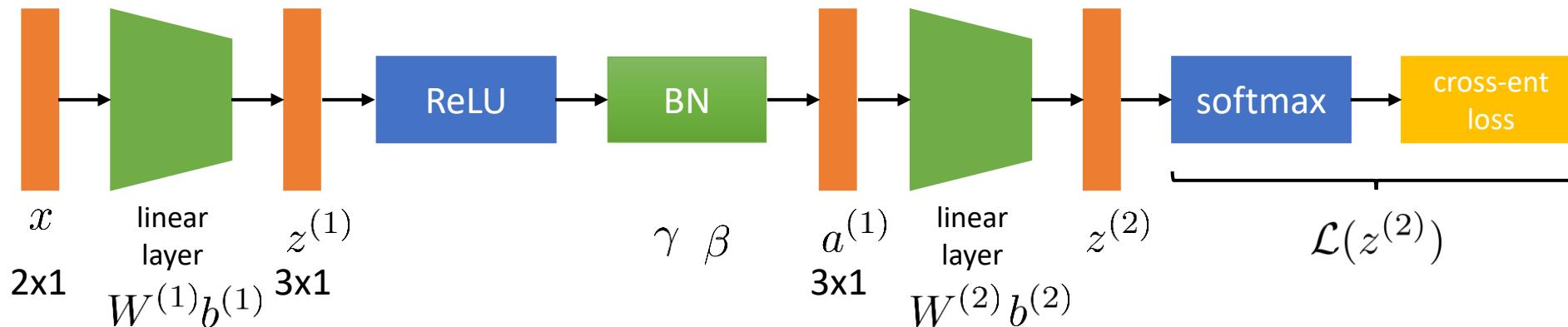


- The “classic” version
- Just appears to be a transformation on the preceding linear layer?

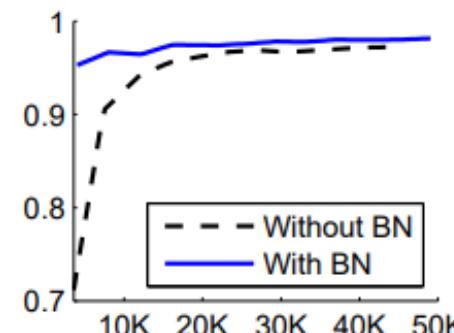
No one seems to agree on what the right way to do it is, try a few options and see what works (but both often work)

A few considerations about batch norm

$$\mu^{(1)} \approx \frac{1}{B} \sum_{j=1}^B a_{i_j}^{(1)} \quad \sigma^{(1)} \approx \sqrt{\frac{1}{B} \sum_{j=1}^B (a_{i_j}^{(1)} - \mu^{(1)})^2} \quad \bar{a}_i^{(1)} = \frac{a_i^{(1)} - \mu^{(1)}}{\sigma^{(1)}} \gamma + \beta$$

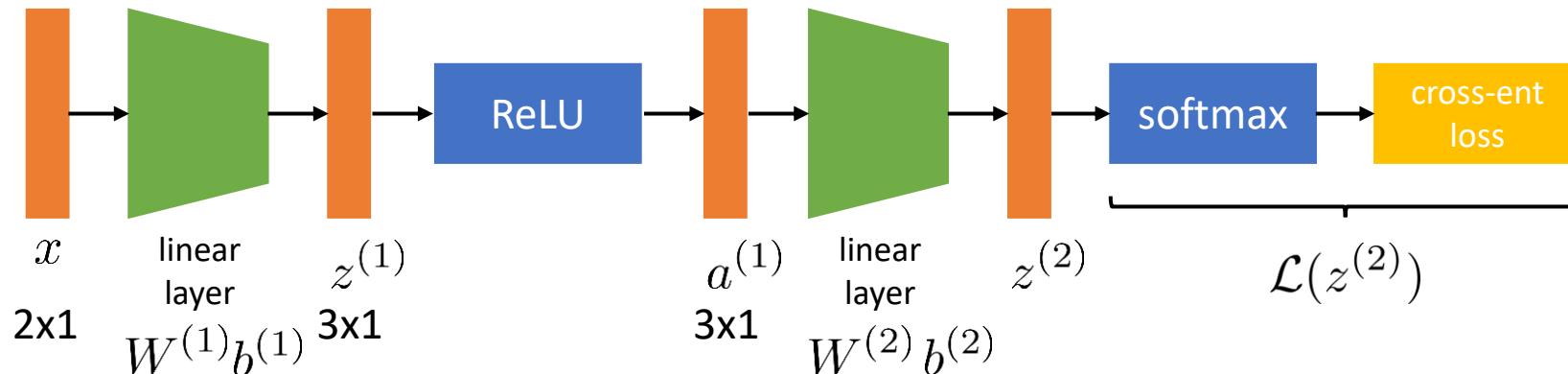


- Often we can use a **larger** learning rate with batch norm
- Models with batch norm can train **much** faster
- Generally requires less regularization (e.g., doesn't need dropout)
- Very good idea in many cases



Weight initialization

General themes



- We want the overall **scale** of activations in the network not to be **too big or too small** for our initial (randomized) weights, so that the gradients propagate well
- **Basic initialization methods:** ensure that activations are on a reasonable scale, and the scale of activations doesn't grow or shrink in later layers as we increase the number of layers
- **More advanced initialization methods:** try to do something about eigenvalues of Jacobians

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}}$$

If we multiply many many numbers together, what will we get?

If most of the numbers are < 1 , we get 0

If most of the numbers are > 1 , we get infinity

We only get a reasonable answer if the numbers are all close to 1!

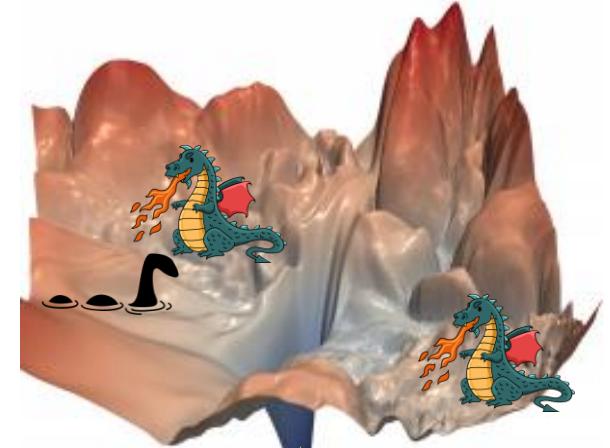
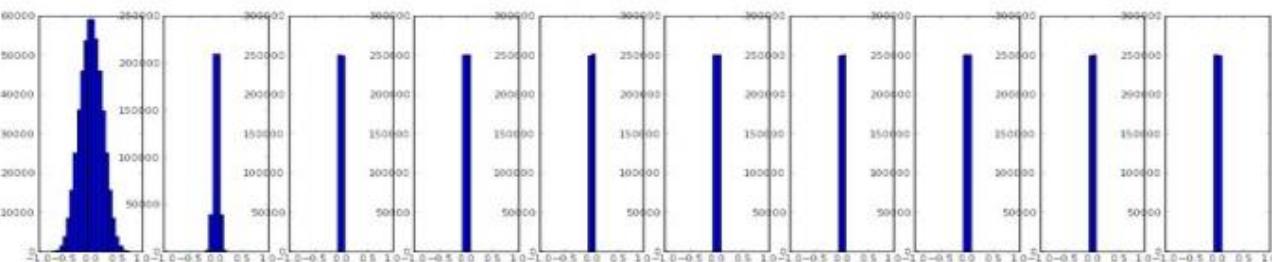
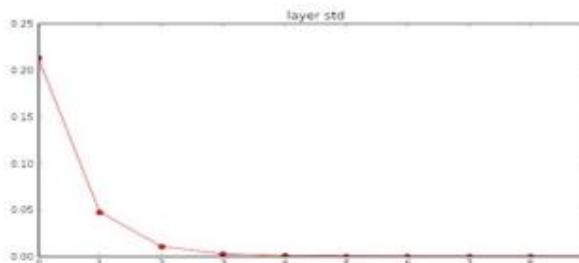
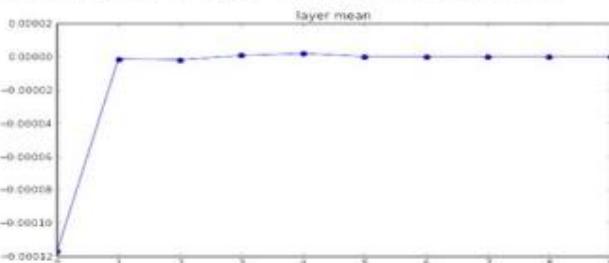
$$\frac{d\mathcal{L}}{dW^{(1)}} = J_1 J_2 J_3 \dots J_n \frac{d\mathcal{L}}{dz^{(n)}}$$

Basic initialization

Simple choice: Gaussian random weights

$$W_{jk}^{(i)} \sim \mathcal{N}(0, 0.0001)$$

```
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
```



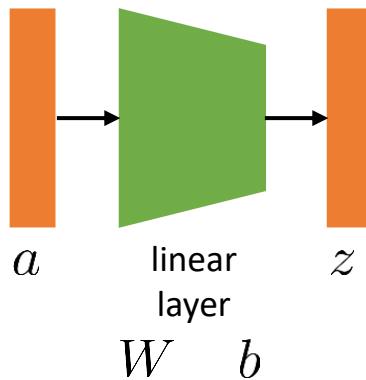
Ideally we could
just initialize here

But we have no idea where that is!

Goal is **not** to start at a good solution, but to
have well-behaved gradients & activations

Why is this bad? $\frac{d\mathcal{L}}{dW^{(i)}} = \frac{dz^{(i)}}{dW^{(i)}} \frac{d\mathcal{L}}{dz^{(i)}} = \delta a^{(i-1)T}$

Basic initialization



$$W_{ij} \sim \mathcal{N}(0, \sigma_W^2)$$

$$b_i \approx 0$$

reasonable choice!
if we standardize x , then
 $x \sim \mathcal{N}(0, 1)$

what is (roughly) the magnitude of z_i ?

$$z_i = \sum_j W_{ij} a_j + \cancel{b_i} \quad b_i \approx 0$$

assume $a_j \sim \mathcal{N}(0, \sigma_a)$
everything is (roughly) 0-mean

$$E[z_i^2] = \sum_j E[W_{ij}^2] E[a_j^2] = D_a \sigma_W^2 \sigma_a^2$$

if $D_a \sigma_W^2 > 1$, magnitude grows with each layer!
if $D_a \sigma_W^2 < 1$, magnitude shrinks with each layer!

what if we choose $\sigma_W^2 = 1/D_a$?
↑
dimensionality of a

Basic initialization

$$E[z_i^2] = \sum_j E[W_{ij}^2]E[a_j^2] = D_a \sigma_W^2 \sigma_a^2$$

↑
dimensionality of a

if $D_a \sigma_W^2 > 1$, magnitude grows with each layer!
if $D_a \sigma_W^2 < 1$, magnitude shrinks with each layer!
what if we choose $\sigma_W^2 = 1/D_a$?

basic principle: get std of W_{ij} to be about $1/\sqrt{D_a}$

this sometimes referred to as “Xavier initialization”

```
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 0.486651
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357188
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.
```

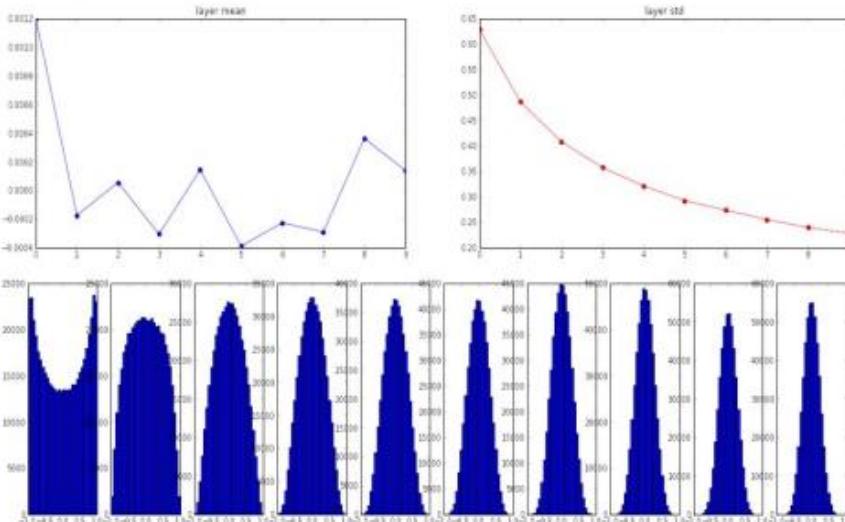


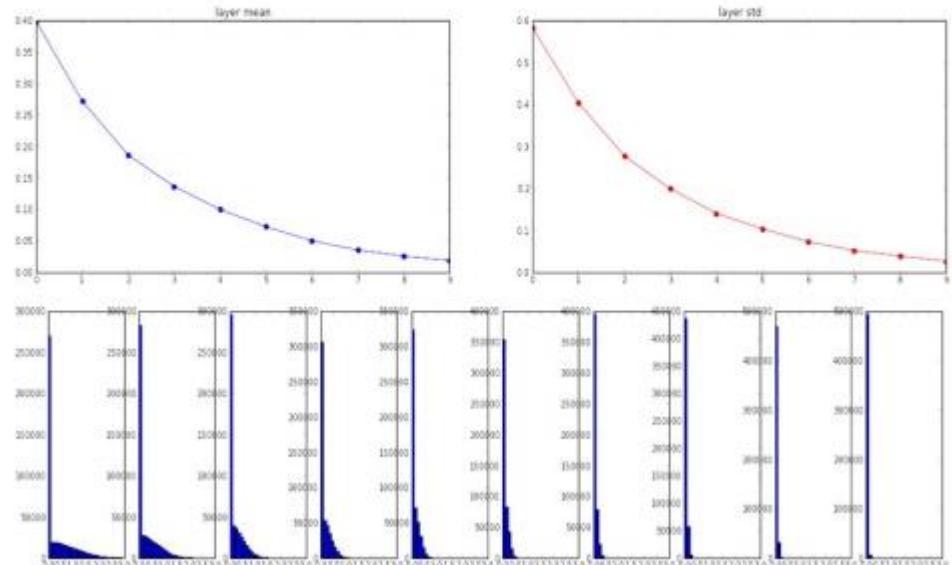
Image from: Fei-Fei Li & Andrej Karpathy

Little detail: ReLUs

$$E[z_i^2] = \sum_j E[W_{ij}^2]E[a_j^2] = D_a \sigma_W^2 \sigma_a^2$$

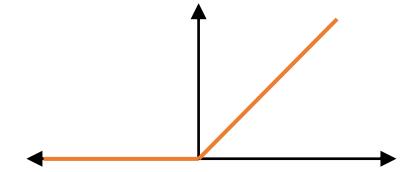
basic principle: get std of W_{ij} to be about $1/\sqrt{D_a}$

```
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 0.186676 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.148299
hidden layer 6 had mean 0.072234 and std 0.103288
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 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076
```



This was all without nonlinearities!

problem: $a_j = \text{ReLU}(z_j)$



“negative half” of 0-mean activations is removed!
variance is cut in half!

might not seem like much...
but it adds up!

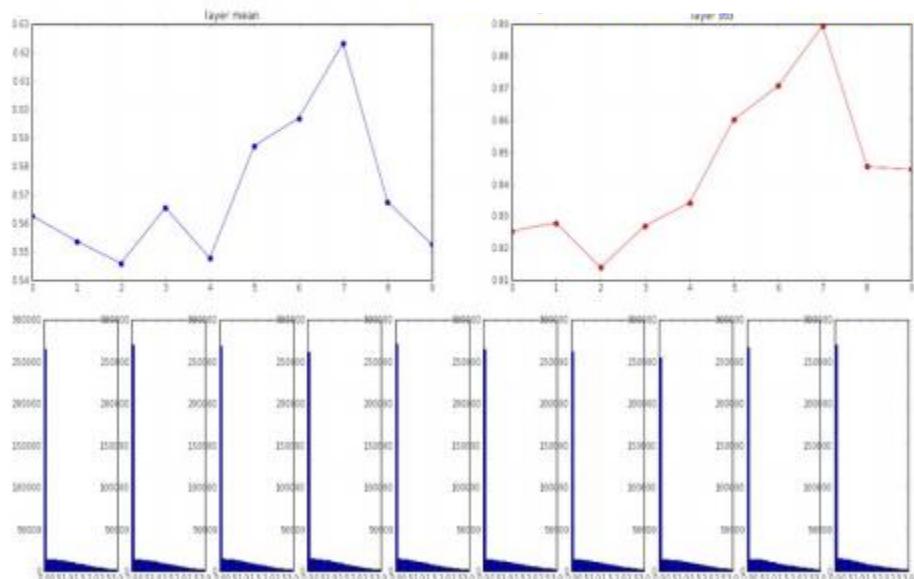
Image from: Fei-Fei Li & Andrej Karpathy

Little detail: ReLUs

$$E[z_i^2] = \sum_j E[W_{ij}^2]E[a_j^2] = D_a \sigma_W^2 \sigma_a^2$$

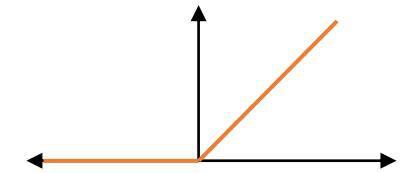
basic principle: get std of W_{ij} to be about $1/\sqrt{D_a}$

```
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
```



This was all without nonlinearities!

problem: $a_j = \text{ReLU}(z_j)$



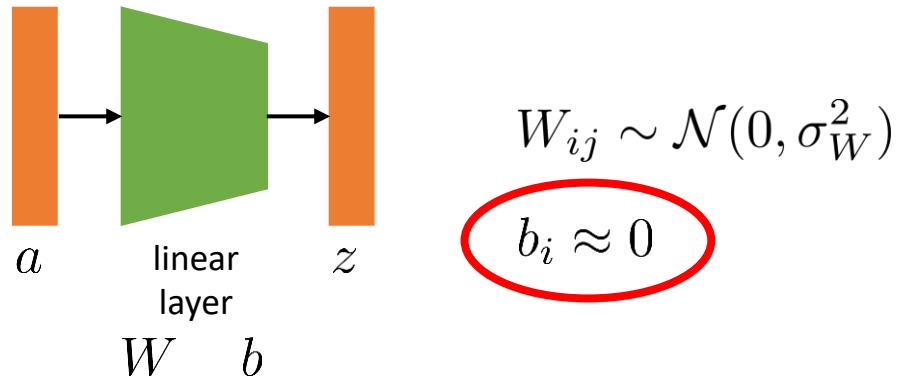
“negative half” of 0-mean activations is removed!
variance is cut in half!

might not seem like much...

proposed by He et al. for ResNet
makes big difference 150+ layers...

Image from: Fei-Fei Li & Andrej Karpathy

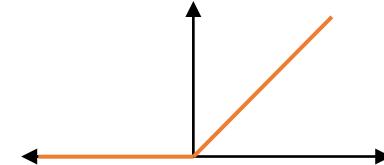
Littler detail: ReLUs & biases



$$W_{ij} \sim \mathcal{N}(0, \sigma_W^2)$$

$$b_i \approx 0$$

problem: $a_j = \text{ReLU}(z_j)$



half of our units (on average) will be “dead”!

often initialize $b_i = 0.1$ (or small constant)

Advanced initialization

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}}$$

If we multiply many many numbers together, what will we get?

If most of the numbers are < 1 , we get 0

If most of the numbers are > 1 , we get infinity

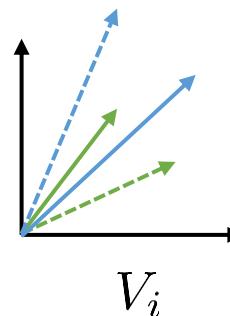
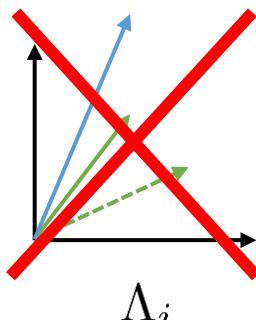
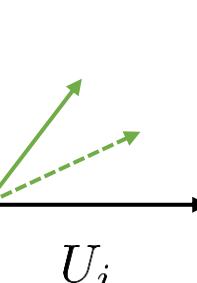
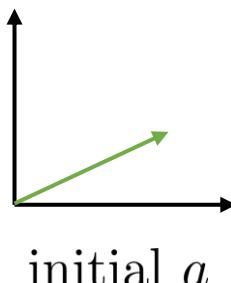
We only get a reasonable answer if the numbers are all close to 1!

for each J_i , we can write: $J_i = U_i \Lambda_i V_i$

e.g., using singular value decomposition

scale-preserving transformations
(i.e., orthonormal bases)

diagonal matrix with same eigenvalues as J_i



initial a

U_i

Λ_i

V_i

Advanced initialization

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}}$$

If we multiply many many numbers together, what will we get?

If most of the numbers are < 1 , we get 0

$$\frac{d\mathcal{L}}{dW^{(1)}} = J_1 J_2 J_3 \dots J_n \frac{d\mathcal{L}}{dz^{(n)}}$$

If most of the numbers are > 1 , we get infinity

We only get a reasonable answer if the numbers are all close to 1!

for each $W^{(i)}$, we can write: $W^{(i)} = U^{(i)} \Lambda^{(i)} V^{(i)}$ e.g., using singular value decomposition

$W^{(i)} \leftarrow U^{(i)} V^{(i)}$ just need to force this to be identity matrix

even simpler:

`a = get_rng().normal(0.0, 1.0, flat_shape)` arbitrary random matrix (doesn't really matter how)

`u, _, v = np.linalg.svd(a, full_matrices=False)`

pick the one with the correct shape

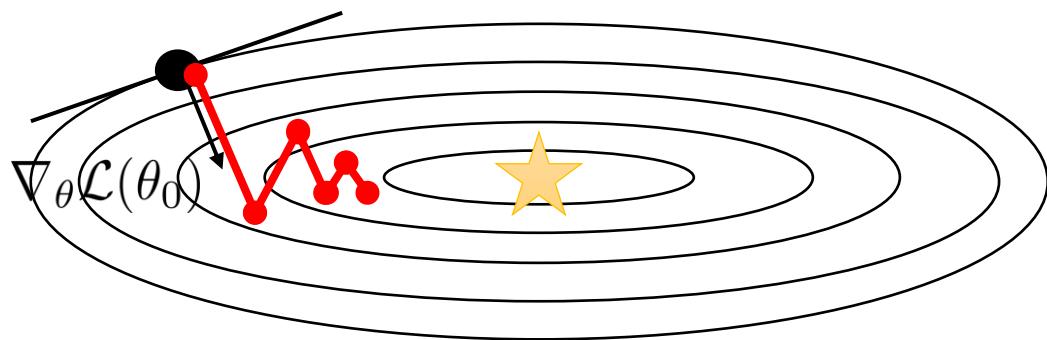
`q = u if u.shape == flat_shape else v`

needed if non-square

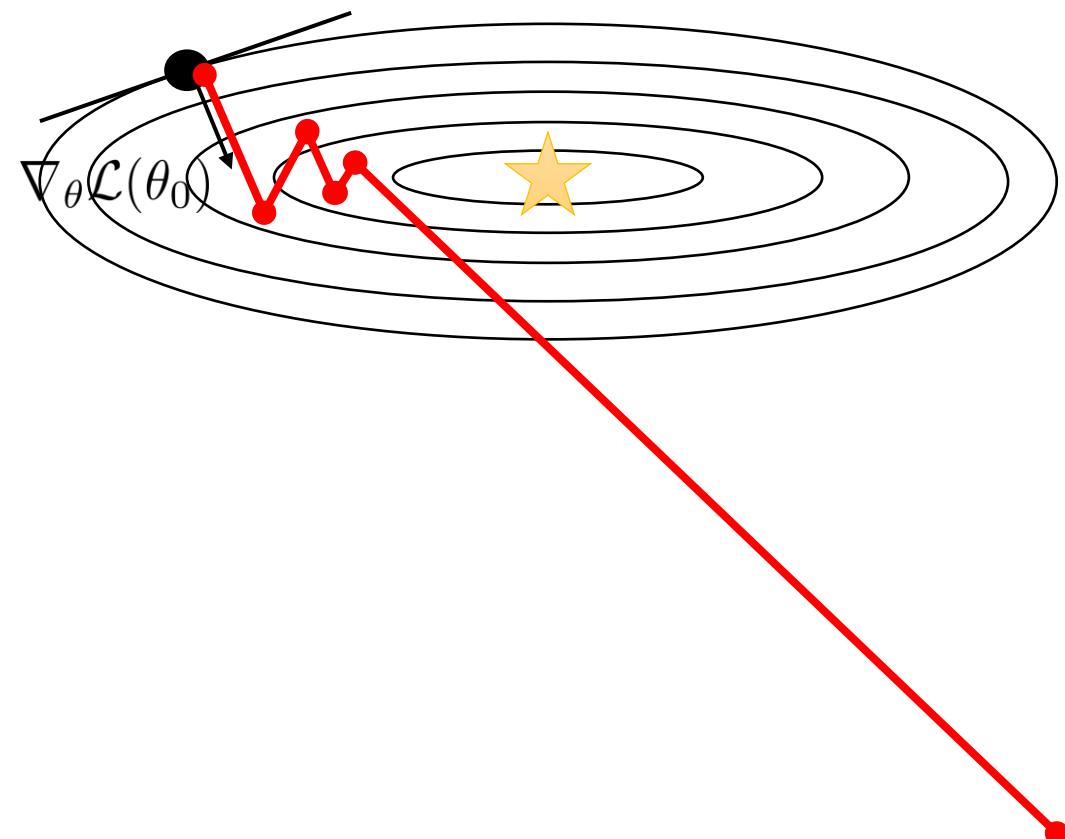
guaranteed orthonormal

Last bit: Gradient clipping

what we hope happens:



what actually happens:
because deep learning, that's why



- Took a step that was too big in the wrong place
- Something got divided by something small (e.g., in batch norm, softmax, etc.)
- Just got really unlucky

Clipping the monster gradients

per-element clipping:

$$\bar{g}_i \leftarrow \max(\min(g_i, c_i), -c_i)$$

norm clipping:

$$\bar{g}_i \leftarrow g \frac{\min(\|g\|, c)}{\|g\|}$$

how to choose c ?

run a few epochs (assuming it doesn't explode)

see what “healthy” magnitudes look like

Ensembles & dropout

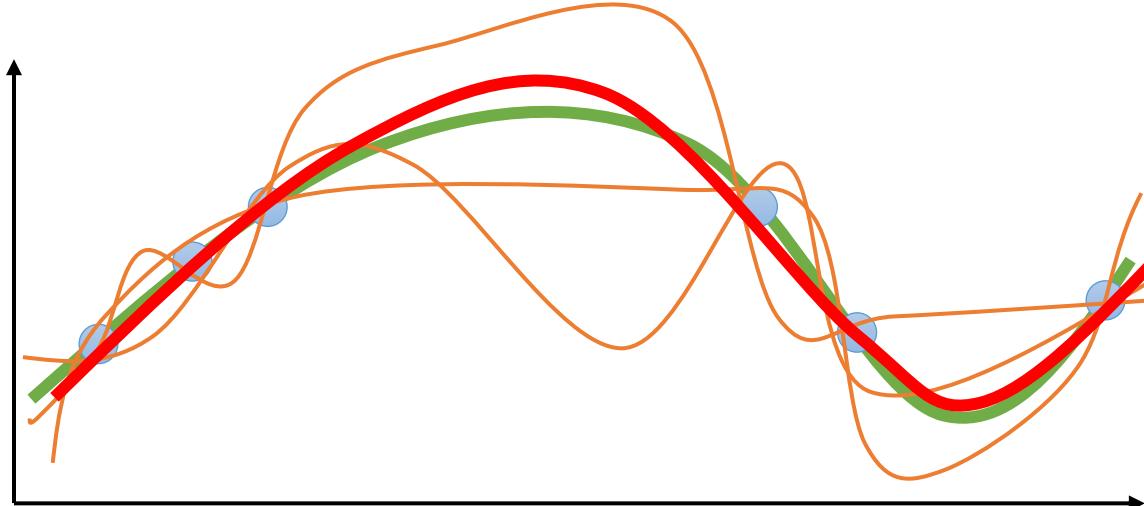
What if my model makes a mistake?

Problem: neural networks have many parameters, often have high variance

Not **nearly** as high as we would expect from basic learning theory
(i.e., overfitting is usually **not** catastrophic), but still...

Interesting idea: when we have multiple high-variance learners, maybe they'll **agree** on the right answer, but **disagree** on the wrong answer

Said another way: there are many more ways to be wrong than to be right



Ensembles in theory

$$\text{Variance} = E_{\mathcal{D} \sim p(\mathcal{D})} [||f_{\mathcal{D}}(x) - \bar{f}(x)||^2]$$

$$\bar{f}(x) = E_{\mathcal{D} \sim p(\mathcal{D})} [f_{\mathcal{D}}(x)] \approx \frac{1}{M} \sum_{i=1}^M f_{\mathcal{D}_j}(x)$$

can we actually estimate this thing?

where do we get **M** different datasets??

Can we **cook up** multiple independent datasets from a single one?

Simple approach: just chop a big dataset into M ~~non overlapping~~ parts

$$\mathcal{D} = \{(x_i, y_i)\}$$

overlapping but **independently sampled**

turns out we actually don't need this!

for each \mathcal{D}_j pick N indices randomly in $\{1, \dots, N\}$ $i_{j,1}, \dots, i_{j,N}$

$$\mathcal{D}_j = \{(x_{i_{j,1}}, y_{i_{j,1}}), (x_{i_{j,2}}, y_{i_{j,2}}), \dots, (x_{i_{j,N}}, y_{i_{j,N}})\}$$

Ensembles in theory

$$\mathcal{D} = \{(x_i, y_i)\}$$

for each \mathcal{D}_j pick N indices randomly in $\{1, \dots, N\}$ $i_{j,1}, \dots, i_{j,N}$

$$\mathcal{D}_j = \{(x_{i_{j,1}}, y_{i_{j,1}}), (x_{i_{j,2}}, y_{i_{j,2}}), \dots, (x_{i_{j,N}}, y_{i_{j,N}})\}$$

This is called resampling **with replacement**

\mathcal{D}	x_1	x_2	x_3
---------------	-------	-------	-------

\mathcal{D}_1	x_2	x_3	x_3		2		3		3
-----------------	-------	-------	-------	--	---	--	---	---	---

\mathcal{D}_2	x_3	x_1	x_3		3		1		3
-----------------	-------	-------	-------	---	---	---	---	--	---

train separate models on each \mathcal{D}_j

Ensembles in theory

$$\mathcal{D} = \{(x_i, y_i)\}$$

for each \mathcal{D}_j pick N indices randomly in $\{1, \dots, N\}$ $i_{j,1}, \dots, i_{j,N}$

$$\mathcal{D}_j = \{(x_{i_{j,1}}, y_{i_{j,1}}), (x_{i_{j,2}}, y_{i_{j,2}}), \dots, (x_{i_{j,N}}, y_{i_{j,N}})\}$$

train separate models on each \mathcal{D}_j

$$p_{\theta_1}(y|x), \dots, p_{\theta_M}(y|x)$$

how do we predict?

principled approach: average the probabilities:

$$p(y|x) = \frac{1}{M} \sum_{j=1}^M p_{\theta_j}(y|x)$$

simple approach: majority vote

Ensembles in practice

There is already a lot of randomness in neural network training

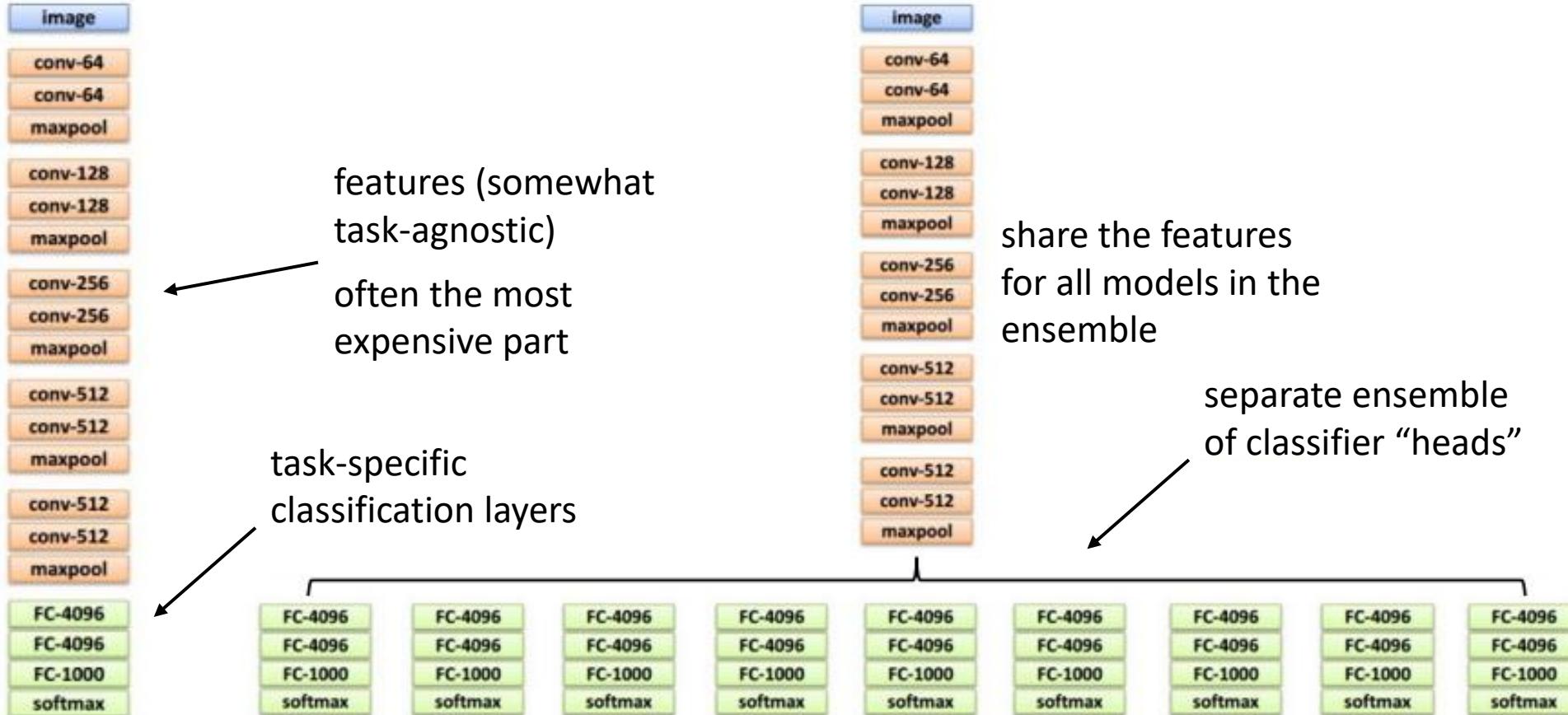
- Random initialization
- Random minibatch shuffling
- Stochastic gradient descent

In practice we get much of the same benefit **without** resampling

train M models $p_{\theta_j}(y|x)$ on the same \mathcal{D}

$$p(y|x) = \frac{1}{M} \sum_{j=1}^M p_{\theta_j}(y|x) \quad \text{or majority vote}$$

Even faster ensembles



Even fasterer ensembles

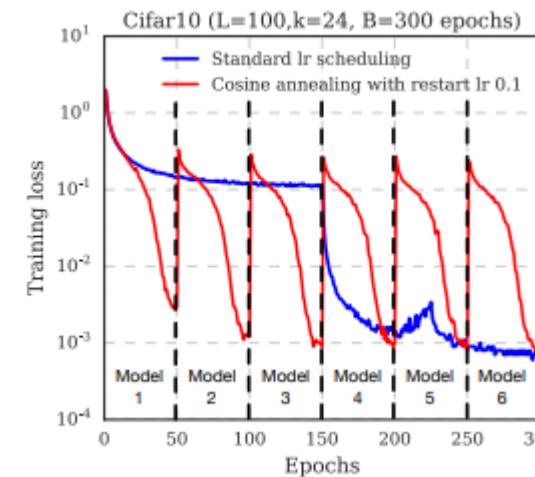
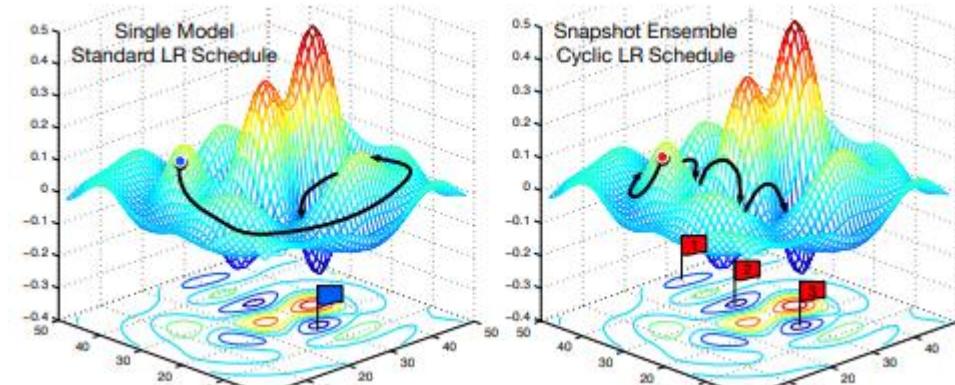
snapshot ensembles:

save out parameter snapshots over the course of SGD optimization, use each snapshot as a model in the ensemble

advantage: don't need to have a bunch of separate training runs

...but need to set things up carefully so that the snapshots are actually different

combining predictions: could average probabilities or vote, or **just average the parameter vectors together**



Some comparisons

Model	Prediction method	Test Accuracy
Baseline (10 epochs)	Single model	0.837
True ensemble of 10 models	Average predictions	0.855
True ensemble of 10 models	Voting	0.851
Snapshots (25) over 10 epochs	Average predictions	0.865
Snapshots (25) over 10 epochs	Voting	0.861
Snapshots (25) over 10 epochs	Parameter averaging	0.864

your mileage may vary

Really really big ensembles?

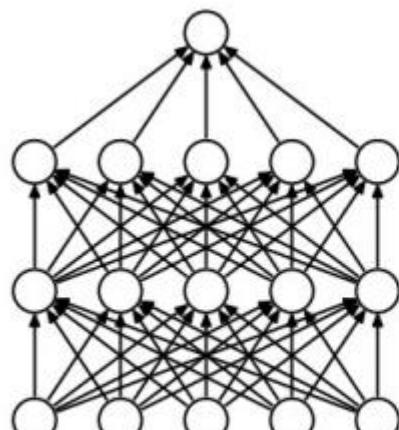
The bigger the ensemble is, the better it works (usually)

But making huge ensembles is expensive

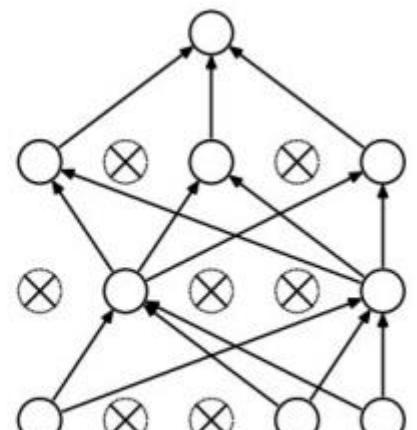
Can we make multiple models **out of a single neural network?**

Dropout

randomly set some activations to zero in the forward pass



(a) Standard Neural Net

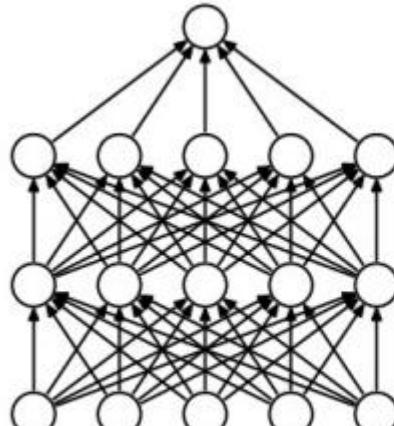


(b) After applying dropout.

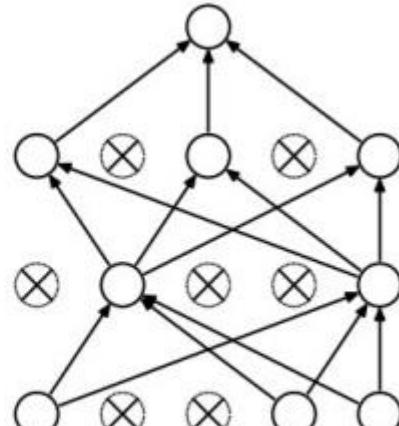
“new” network made
out of the old one

Dropout

randomly set some activations to zero in the forward pass



(a) Standard Neural Net



(b) After applying 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)
```

Andrej Karpathy

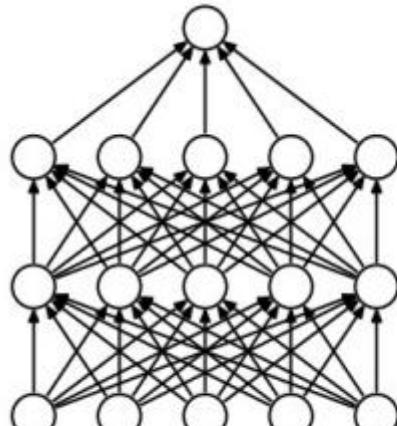
Implementation:

for each $a_j^{(i)}$, set it to $a_j^{(i)} m_{ij}$

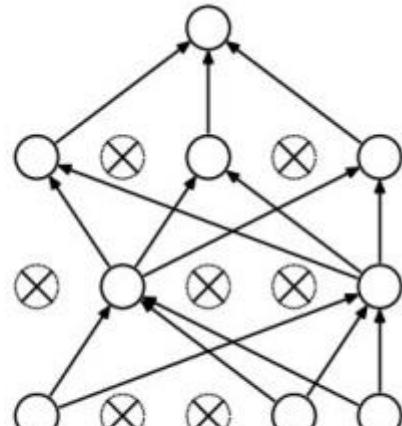
$m_{ij} \sim \text{Bernoulli}(0.5)$ 1.0 with probability 50%, 0.0 otherwise

Dropout

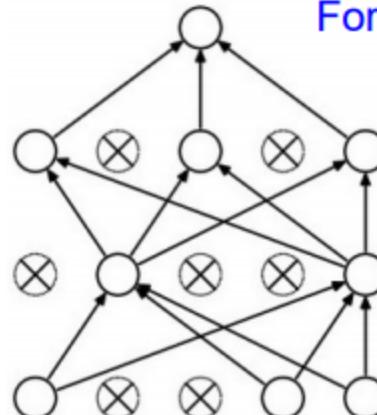
randomly set some activations to zero in the forward pass



(a) Standard Neural Net



(b) After applying dropout.



Forces the network to have a redundant representation.



How could this possibly work?

Can think of every **dropout mask** as defining a different model

Hence this looks like a **huge** ensemble

How huge?

At test time...

During training:

for each $a_j^{(i)}$, set it to $a_j^{(i)} m_{ij}$

$m_{ij} \sim \text{Bernoulli}(0.5)$

```
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!

Andrej Karpathy

At test time:

want to combine all the models

could just generate many dropout masks

what if we stop dropping out at test time?

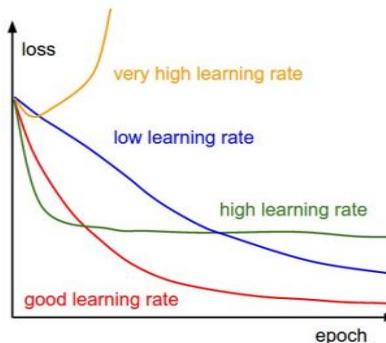
before: on average $\frac{1}{2}$ of dimensions are forced to 0

now: none of them are, so $W^{(i)} a^{(i)}$ will be $\approx 2 \times$ bigger

solution: $\bar{W}^{(i)} = \frac{1}{2} W^{(i)}$ (divide all weights by 2!)

Hyperparameters

- With all these tricks, we have a **lot** of hyperparameters
- Some of these affect **optimization** (training)
 - Learning rate
 - Momentum
 - Initialization
 - Batch normalization
- Some of these affect **generalization** (validation)
 - Ensembling
 - Dropout
 - Architecture (# and size of layers)
- How do we pick these?
 - Recognize which is which: this can really matter!
 - Bad learning rate, momentum, initialization etc. shows up **very** early on in the training process
 - Effect of architecture usually only apparent after training is done
 - Coarse to fine: start with broad sweep, then zero in
 - Consider random hyperparameter search instead of grid



Example: short (5 epoch) log-space LR & weight decay sweep

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)
```

note it's best to optimize in log space!

```
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)
```

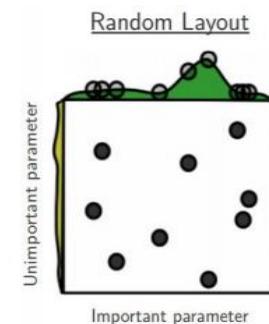
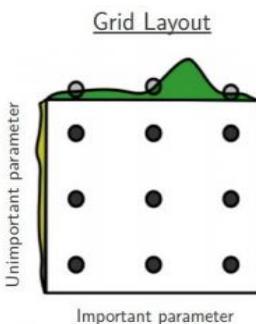
Andrey Karpathy

```
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)
```

```
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)
```



Computer Vision

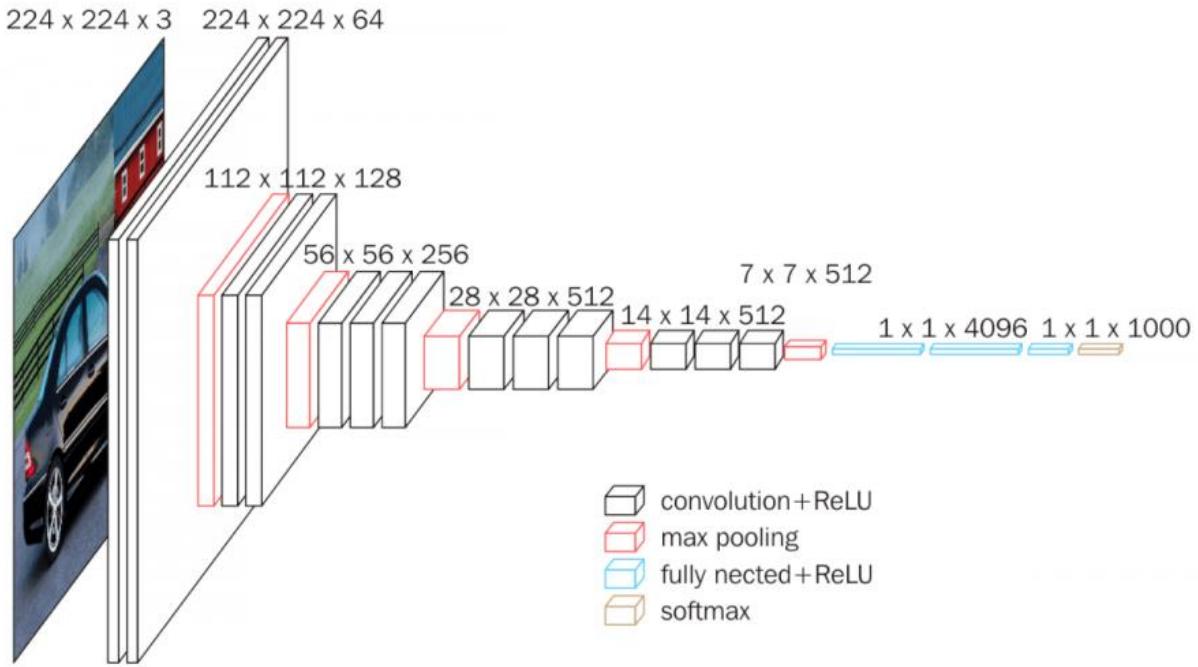
Designing, Visualizing and Understanding Deep Neural Networks

CS W182/282A

Instructor: Sergey Levine
UC Berkeley



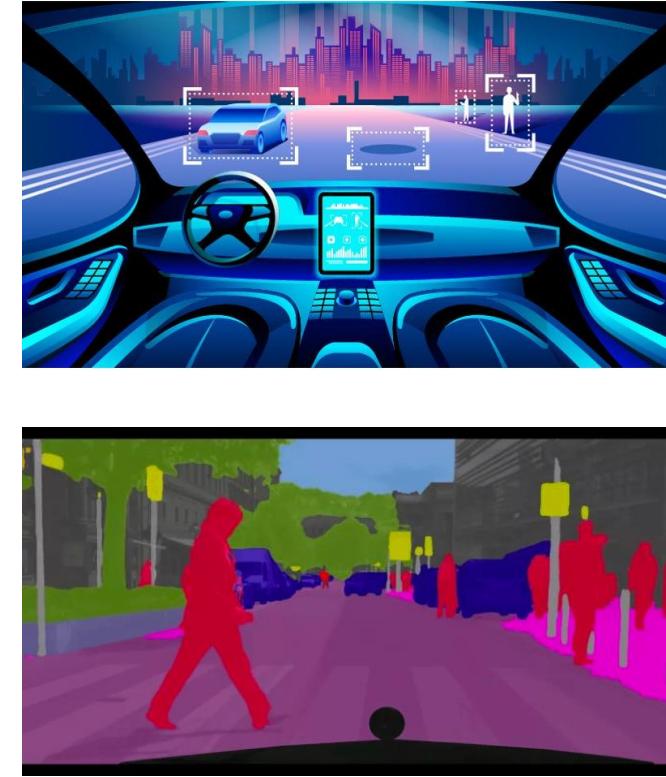
So far...



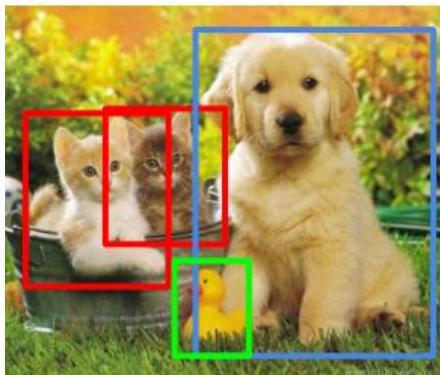
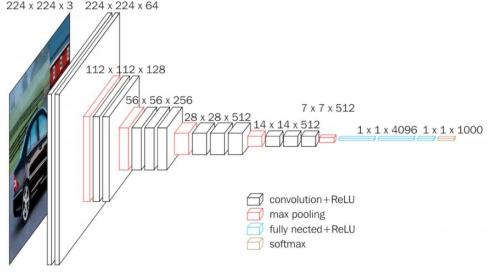
convolutional networks: map image to output value



e.g., semantic category (“bicycle”)



Standard computer vision problems

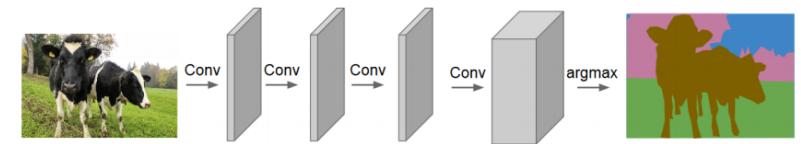


object classification

object localization

object detection

semantic segmentation
a.k.a. scene understanding



Object localization setup

Before: $\mathcal{D} = \{(x_i, y_i)\}$

image class label (categorical)

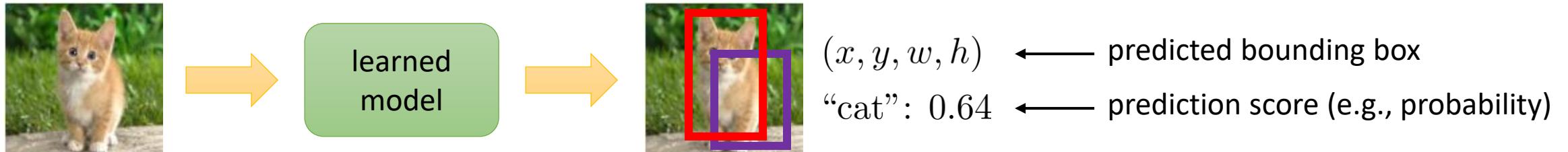


Now: $\mathcal{D} = \{(x_i, y_i)\}$

image $y_i = (\ell_i, x_i, y_i, w_i, h_i)$

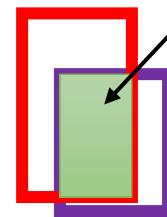


Measuring localization accuracy

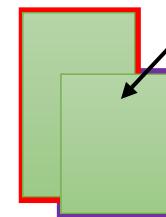


Did we get it right?

Intersection over Union (IoU)



intersection area (**I**)



union area (**U**)

$$\text{IoU} = I / U$$

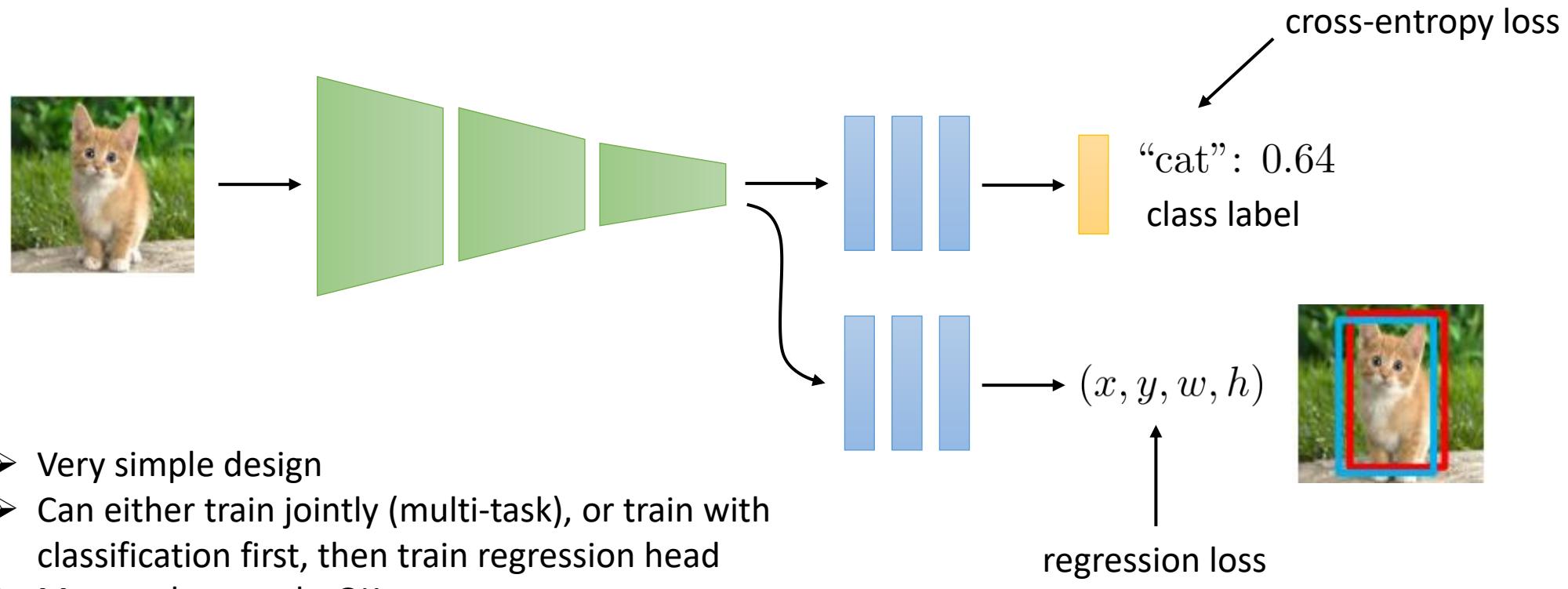
Different datasets have different protocols, but one reasonable one is: **correct if IoU > 0.5**

If also outputting class label (usually the case): **correct if IoU > 0.5 and class is correct**

This is **not** a loss function! Just an evaluation standard

Object localization as regression

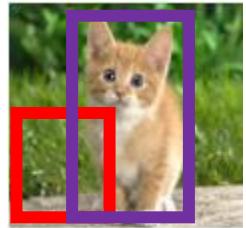
$$\mathcal{D} = \{(x_i, y_i)\} \quad y_i = (\ell_i, x_i, y_i, w_i, h_i)$$



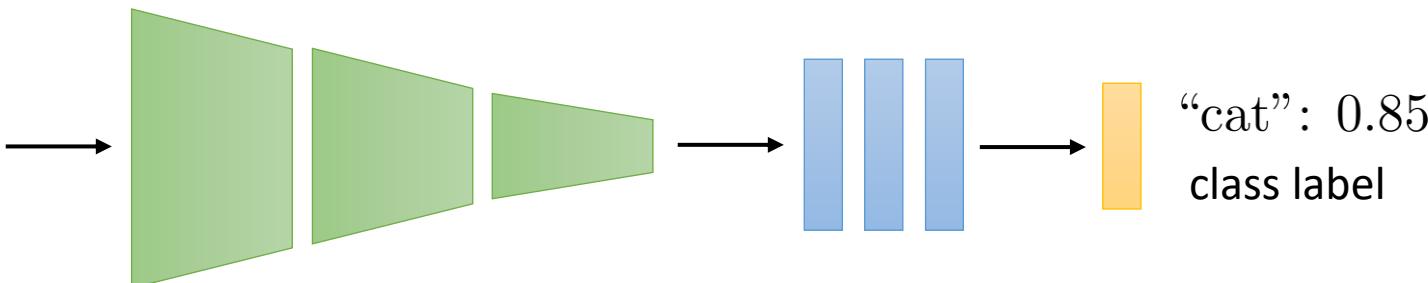
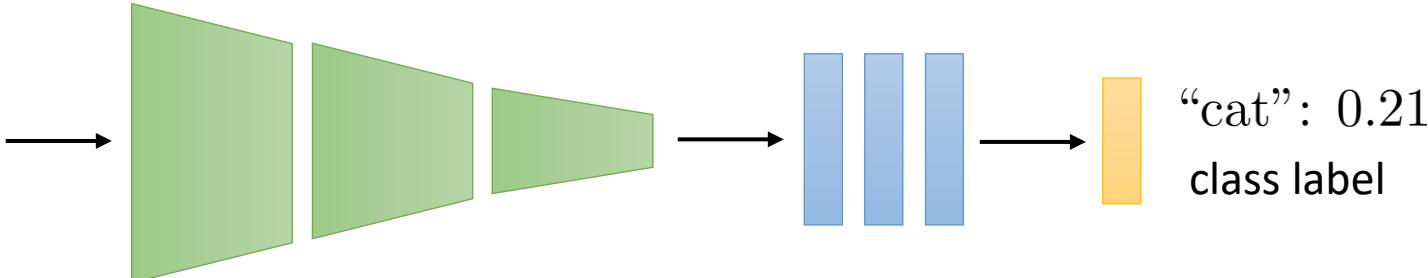
- Very simple design
- Can either train jointly (multi-task), or train with classification first, then train regression head
- More or less works OK
- By itself, this is **not** the way it's usually done!
 - We'll see why shortly

Sliding windows

$$\mathcal{D} = \{(x_i, y_i)\} \quad y_i = (\ell_i, x_i, y_i, w_i, h_i)$$

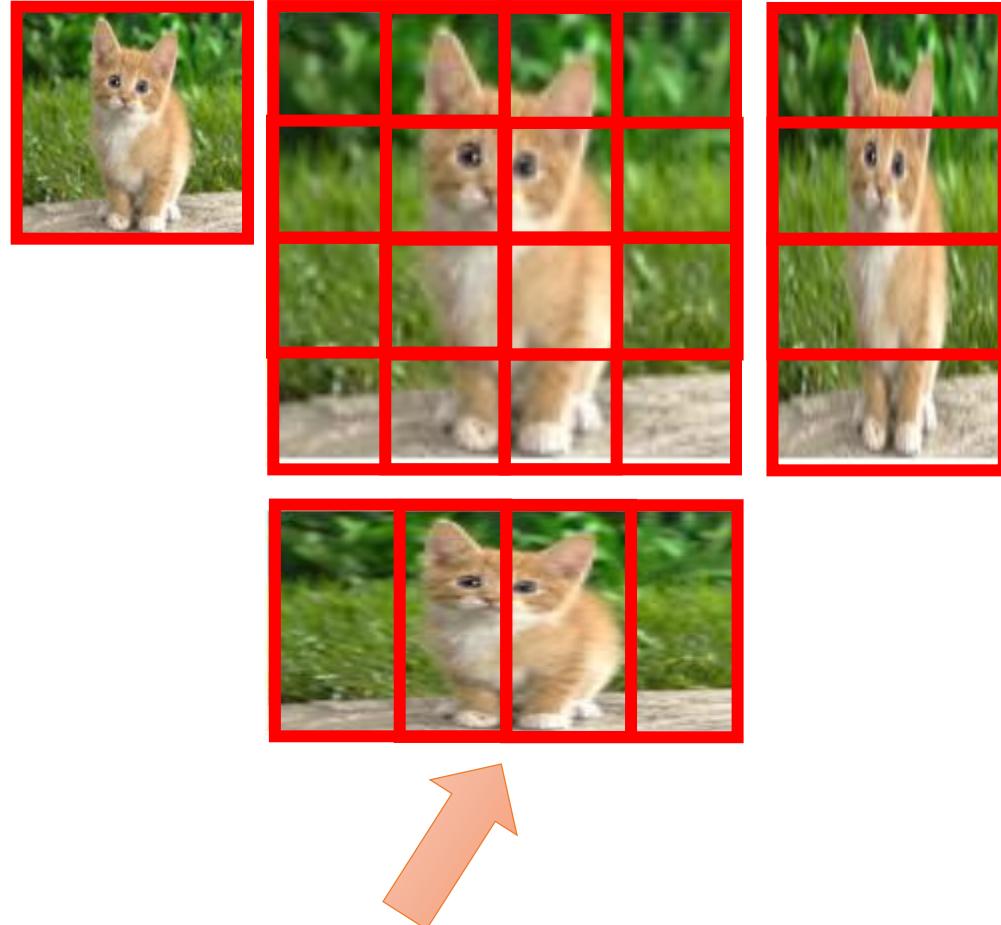


What if we classify **every** patch in the image?

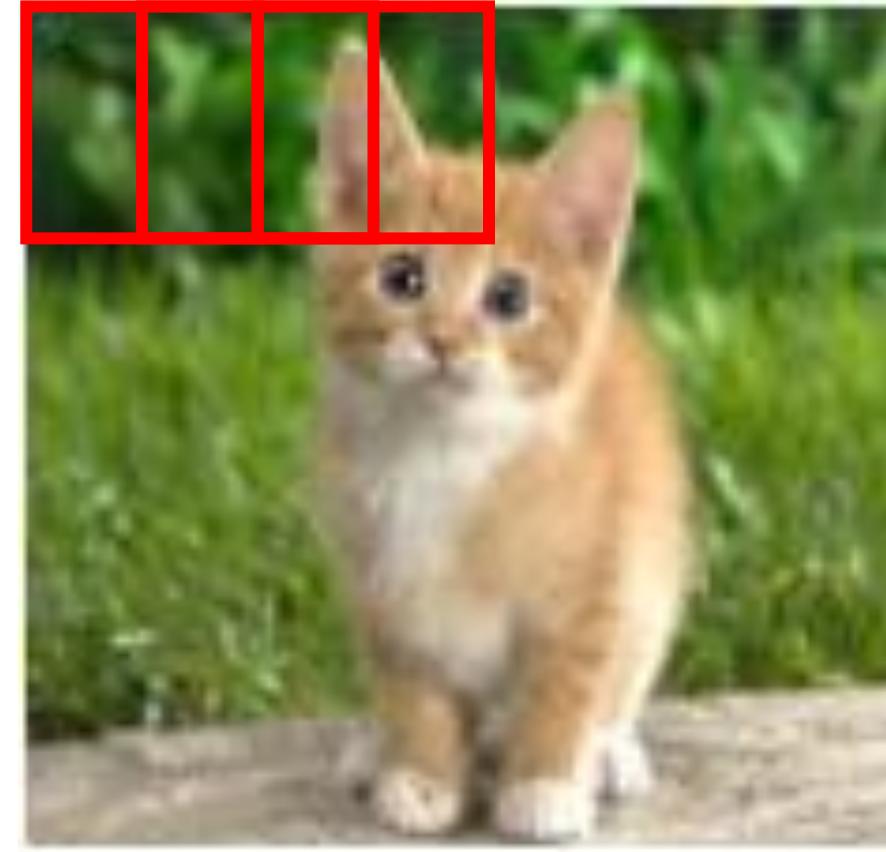


Sliding windows

$$\mathcal{D} = \{(x_i, y_i)\} \quad y_i = (\ell_i, x_i, y_i, w_i, h_i)$$

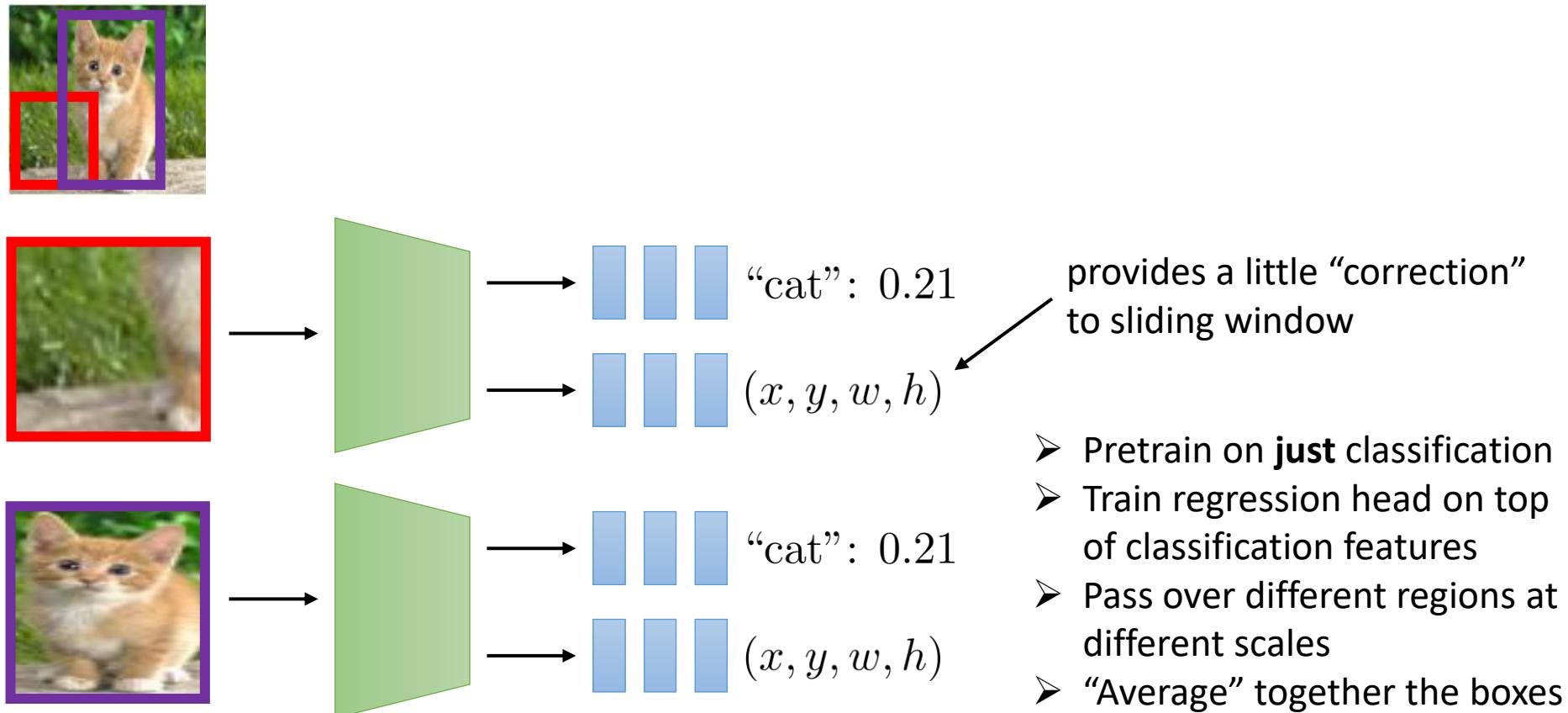


could just take the box with the **highest** class probability
more generally: **non-maximal suppression**

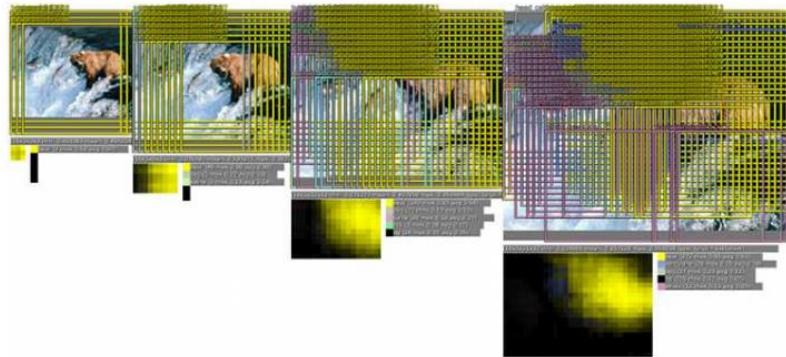


A practical approach: OverFeat

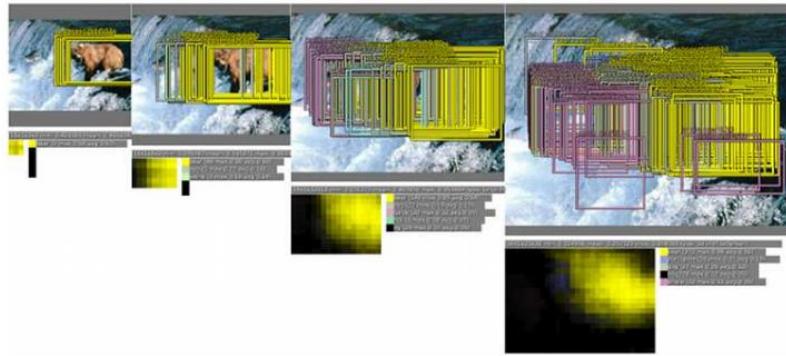
$$\mathcal{D} = \{(x_i, y_i)\} \quad y_i = (\ell_i, x_i, y_i, w_i, h_i)$$



A practical approach: OverFeat



Sliding window **classification** outputs at each scale/position (**yellow** = bear)



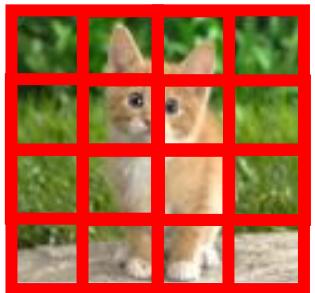
Predicted box x, y, w, h at each scale/position (**yellow** = bear)



Final combined bounding box prediction
(**yellow** = bear)

Sliding windows & reusing calculations

Problem: sliding window is very expensive! (36 windows = 36x the compute cost)

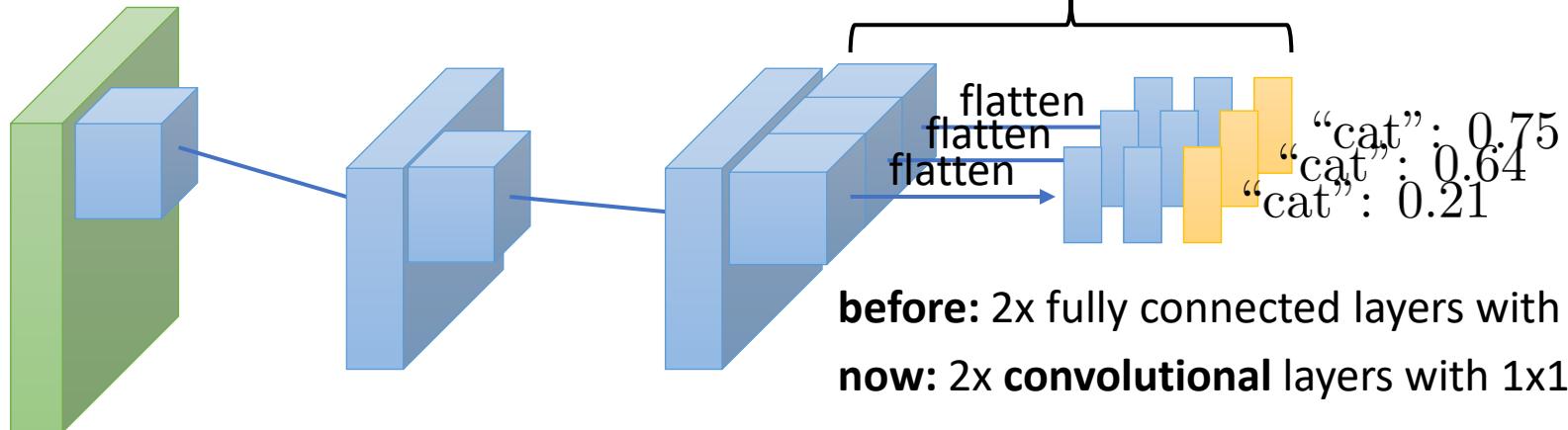


This looks **a lot** like convolution...

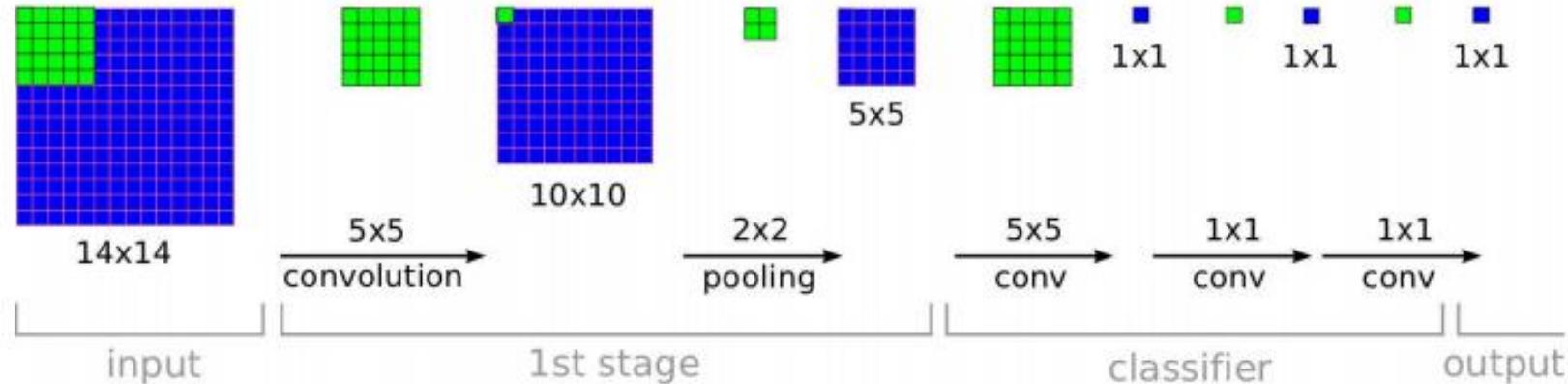
Can we just **reuse** calculations across windows?

“Convolutional classification”

these are just convolutional layers!

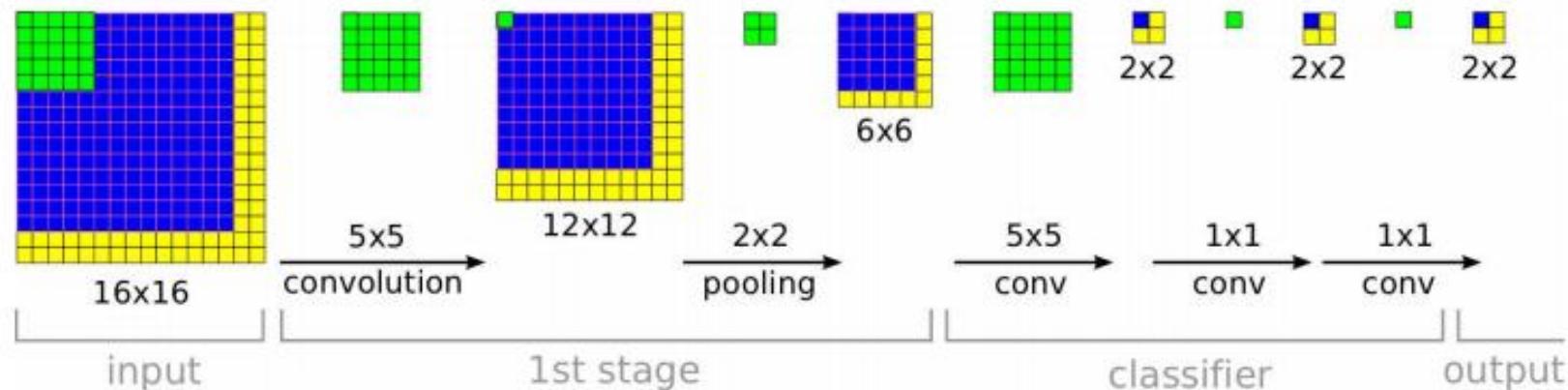


Sliding windows & reusing calculations



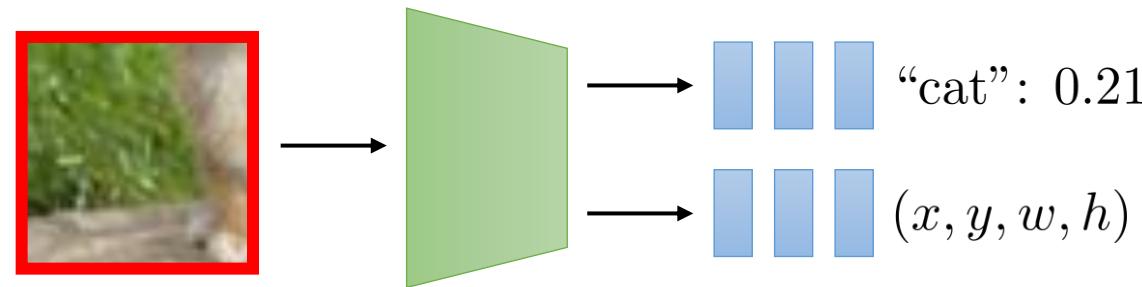
This kind of calculation reuse is extremely powerful for localization problems with conv nets

We'll see variants of this idea in every method we'll cover today!

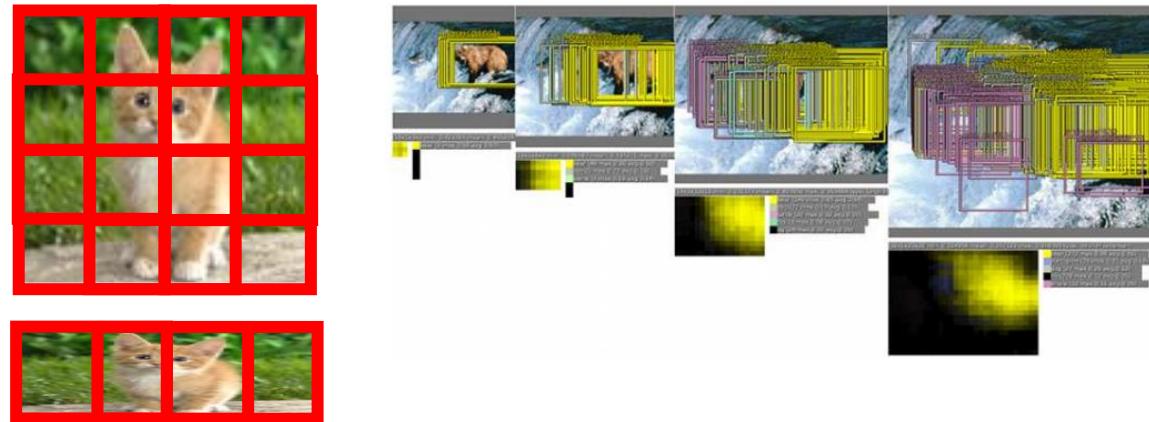


Summary

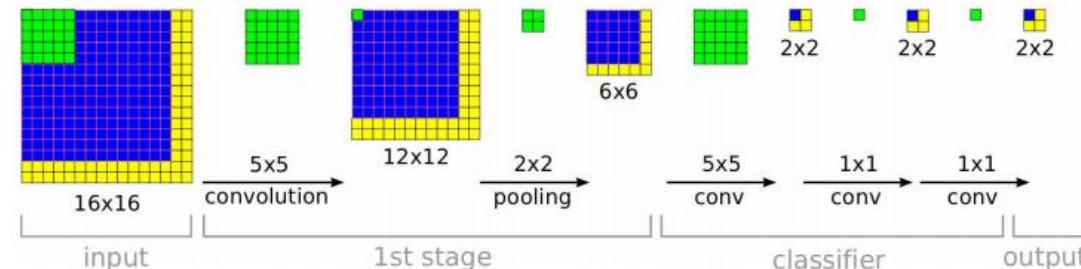
Building block: conv net that outputs class and bounding box coordinates



Evaluate this network at multiple scales and for many different crops, each one producing a probability and bounding box



Implement the sliding window as just another convolution, with 1×1 convolutions for the classifier/regressor at the end, to save on computation



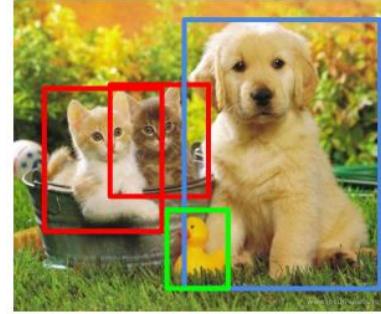
Object detection architectures

The problem setup

Before

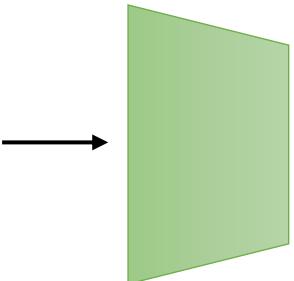


Now



$(x_i,$

number of objects n_i different for each image x_i !



→ “cat”: 0.21

→ (x, y, w, h) ???

How do we get multiple outputs?

Sliding window: each window can be a different object

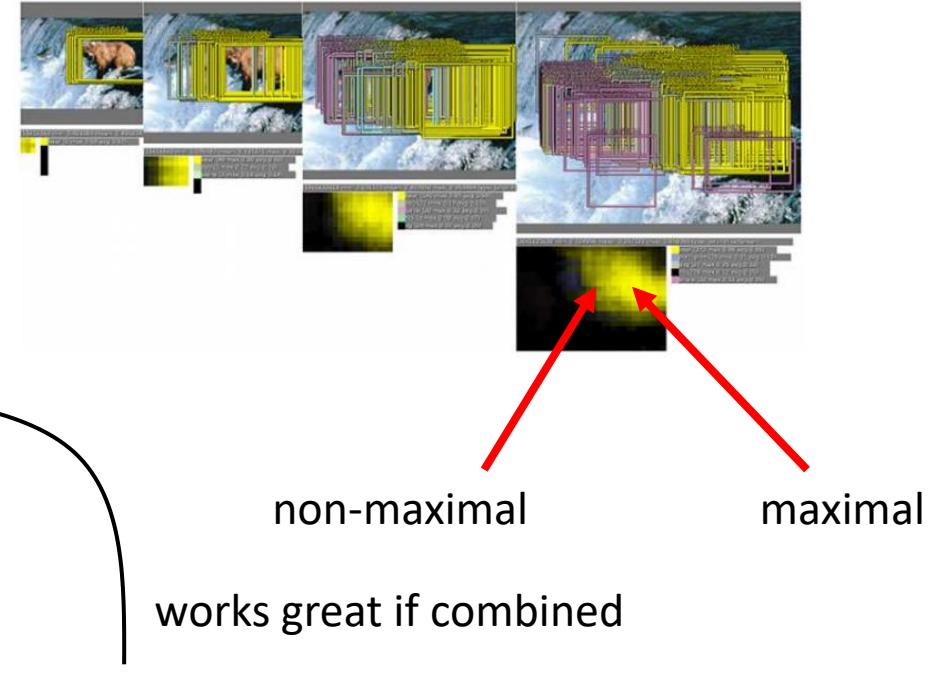
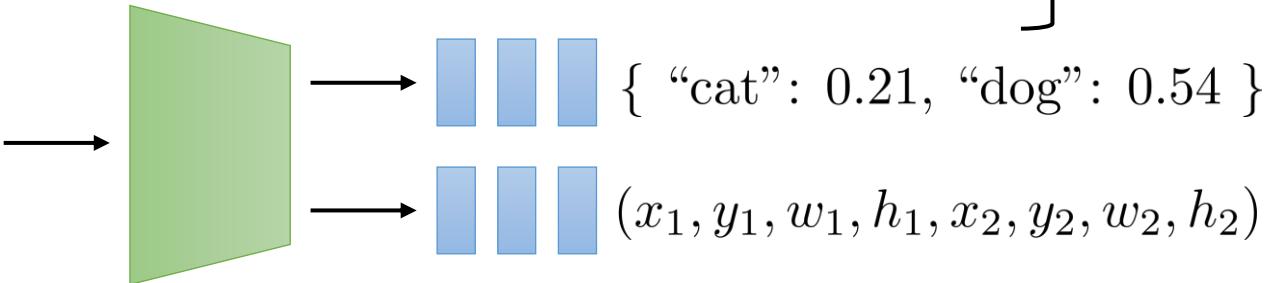
Instead of selecting the window with the highest probability (or merging windows), just output an object in each window above some threshold

Big problem: a high-scoring window probably has **other** high-scoring windows nearby

Non-maximal suppression: (informally) kill off any detections that have other higher-scoring detections of the same class nearby

Actually output multiple things: output is a list of bounding boxes

Obvious problem: need to pick number, usually pretty small



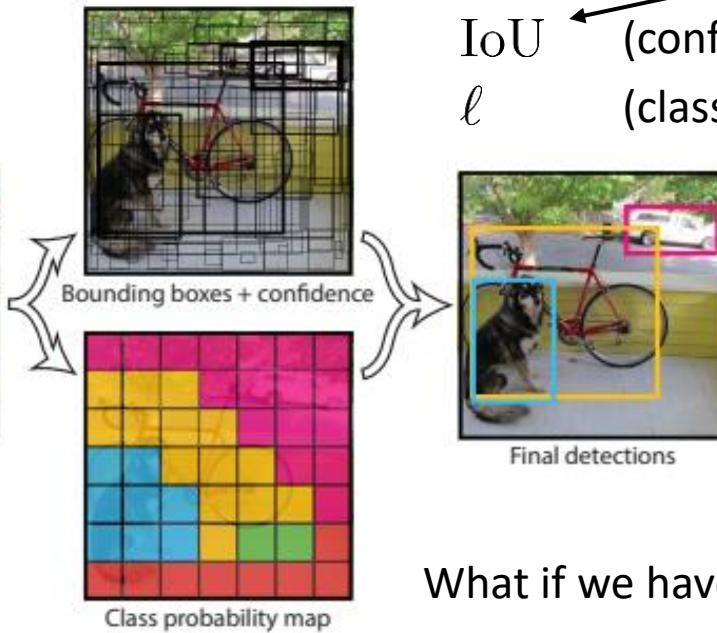
Case study: you only ~~live~~ once (YOLO) look

Actually, you look a few times (49 times to be exact...)

different output for each of the 7×7 (49) grid cells
(a bit like sliding window)



use the same trick as before to reuse computation (cost is not 49x higher!)



for each cell, output:

(x, y, w, h)
IoU
 ℓ

zero if no object
output B of these

some training details:

need to assign which output is “responsible” for each true object during training

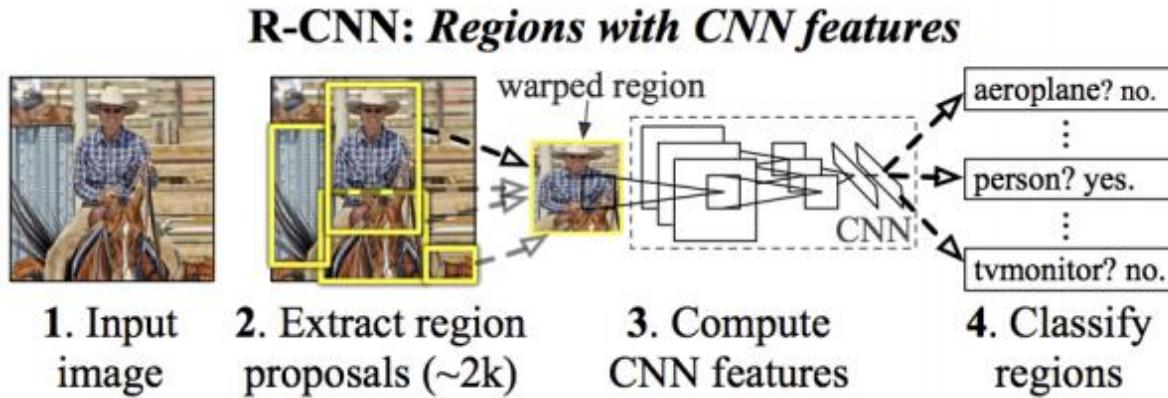
just use the “best-fit” object in that cell (i.e., the one with highest IoU)

What if we have too many objects?

Well, nothing... we just miss them

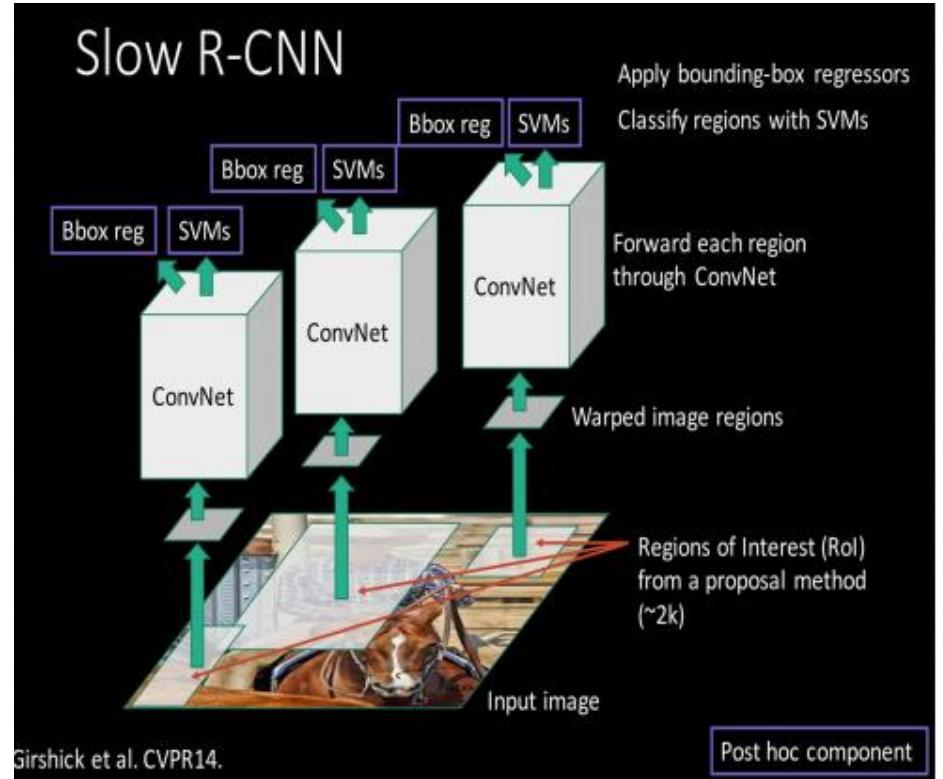
CNNs + Region proposals

A smarter “sliding window”: region of interest proposals



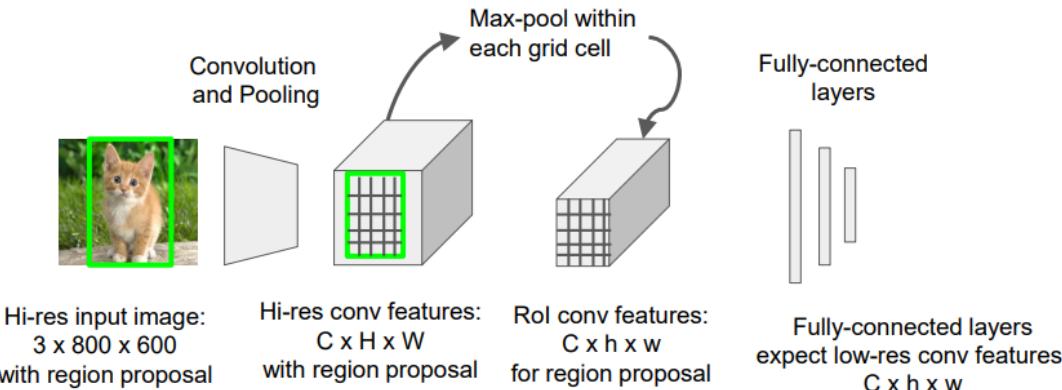
This is really slow

But we already know how to fix this!

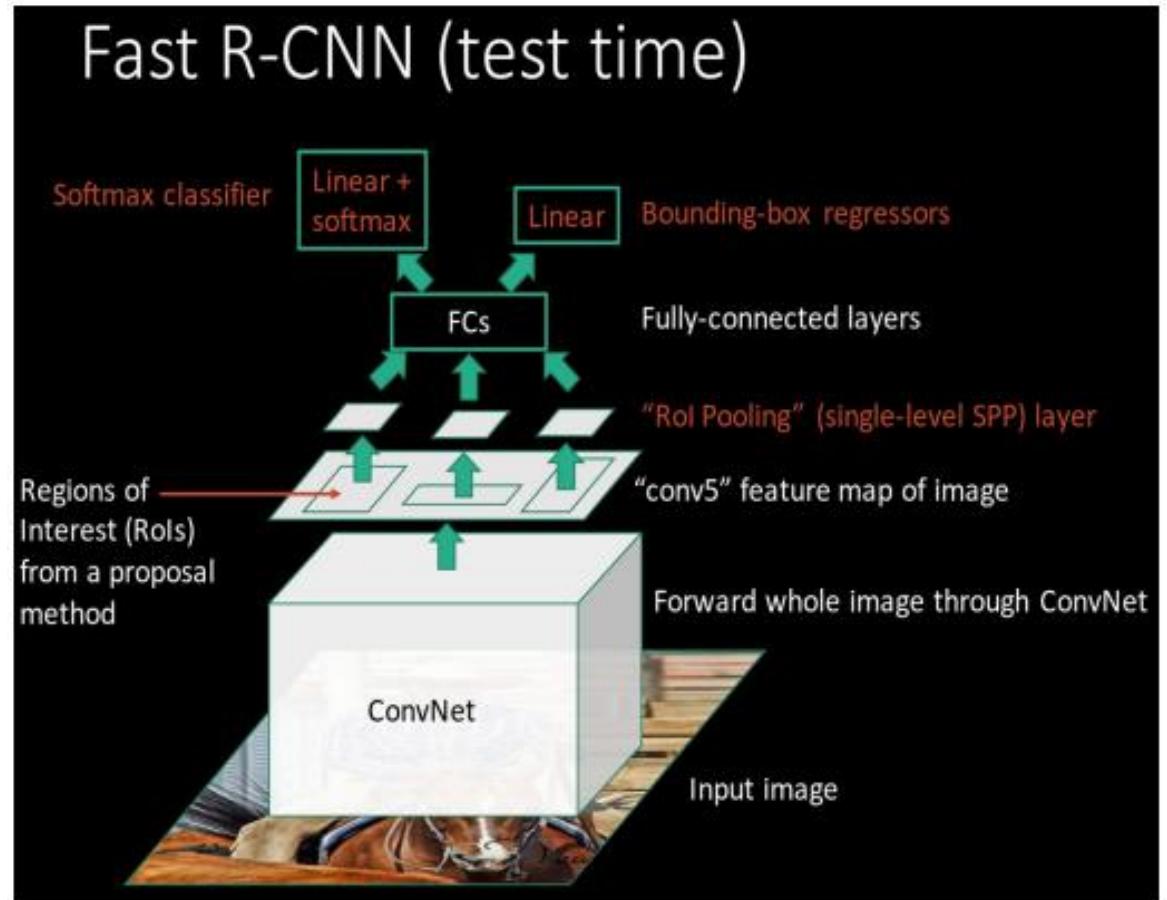
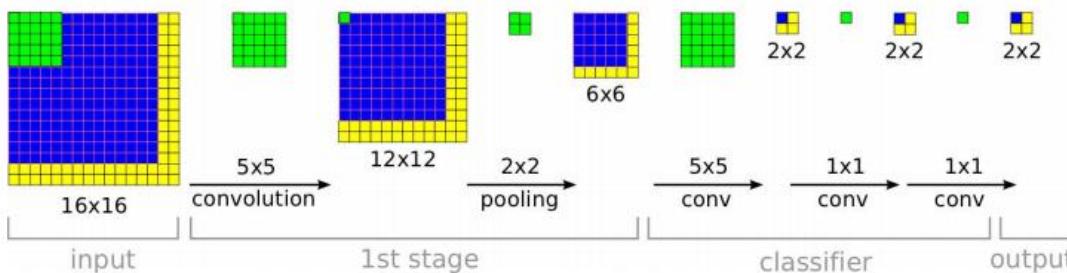


CNNs + Region proposals

A smarter “sliding window”: region of interest proposals



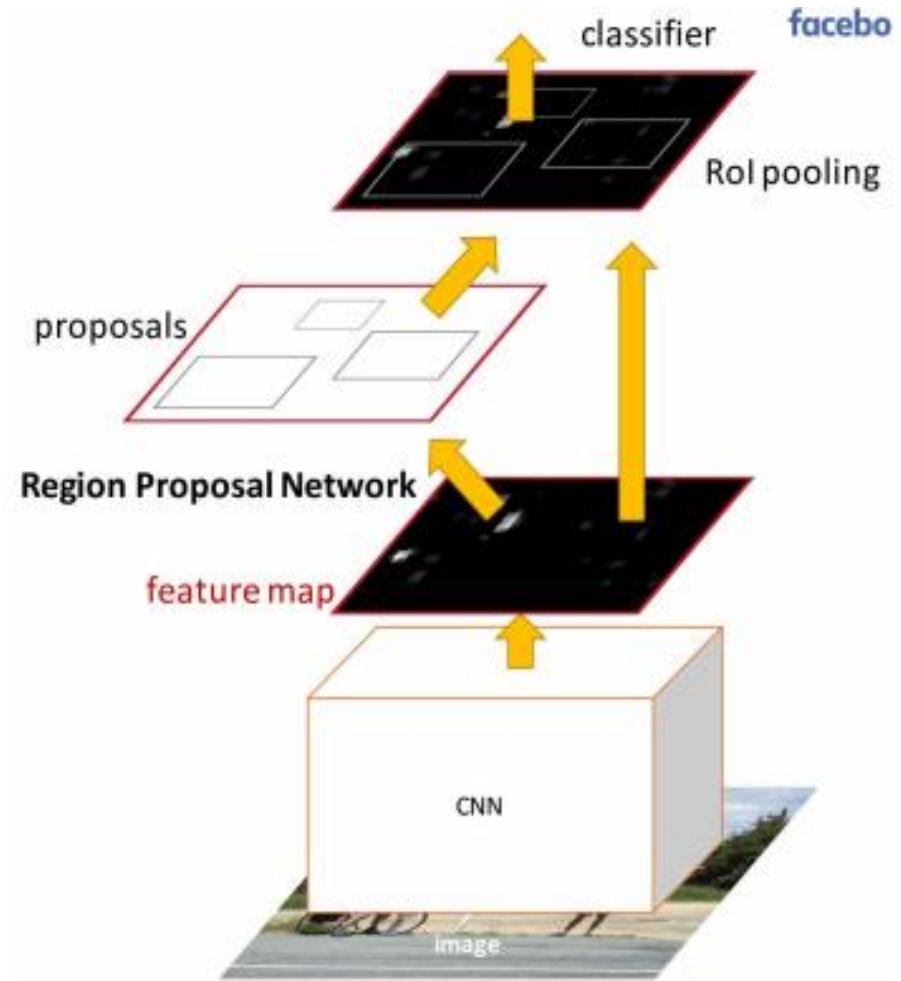
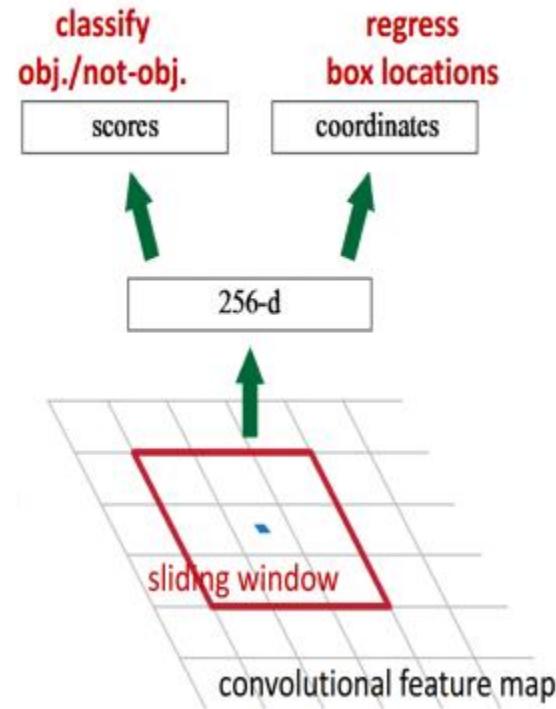
Compare this to evaluating every location:



CNNs + Region proposals

How to train region of interest proposals?

Very similar design to what we saw before (e.g., OverFeat, YOLO), but now for predicting if **any** object is present around that location



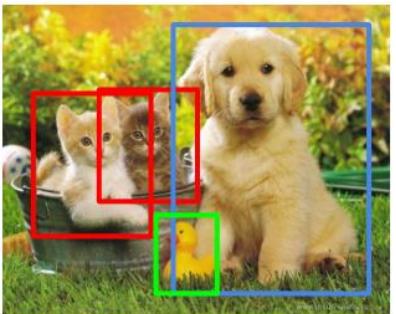
Suggested readings

- Redmon et al. “**You Only Look Once: Unified, Real-Time Object Detection.**” 2015
 - Just regress to different bounding boxes in each cell
 - A few follow-ups (e.g., YOLO v5) that work better
- Girschick et al. “**Fast R-CNN.**” 2015
 - Uses region of interest proposals instead of sliding window/convolution
- Ren et al. “**Faster R-CNN.**” 2015
 - Same as above with a few improvements, like region of interest proposal learning
- Liu et al. **SSD: Single Shot MultiBox Detector.** 2015
 - Directly “classifies” locations with class and bounding box shape

Segmentation architectures

The problem setup

Before



Now

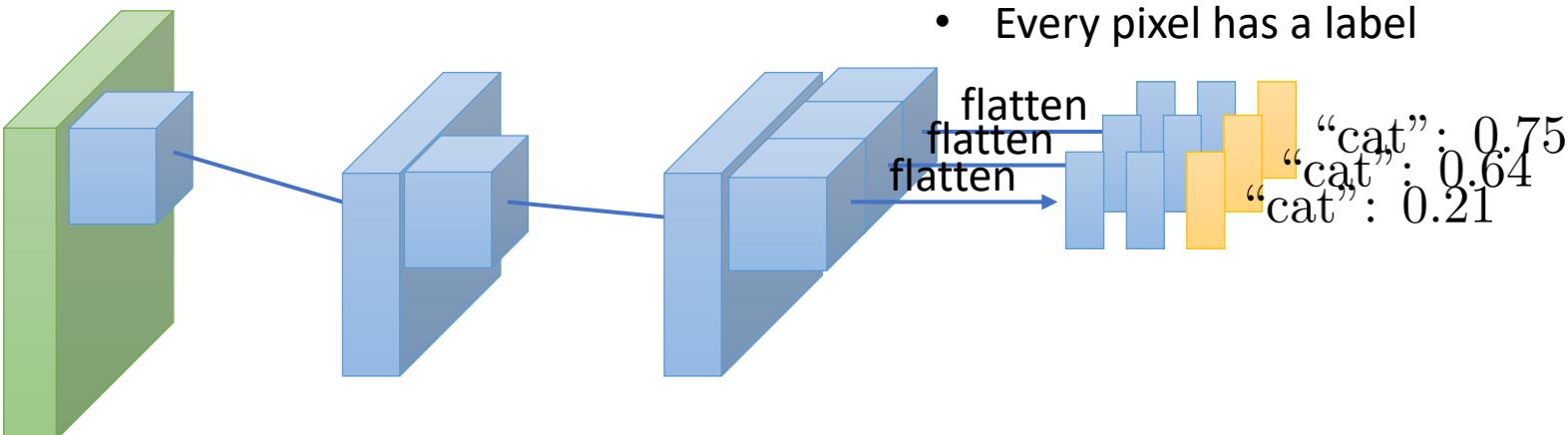


Simple solution:
“per pixel” classifier

Label **every single** pixel with its class

Actually simpler in some sense:

- No longer variable # of outputs
- Every pixel has a label

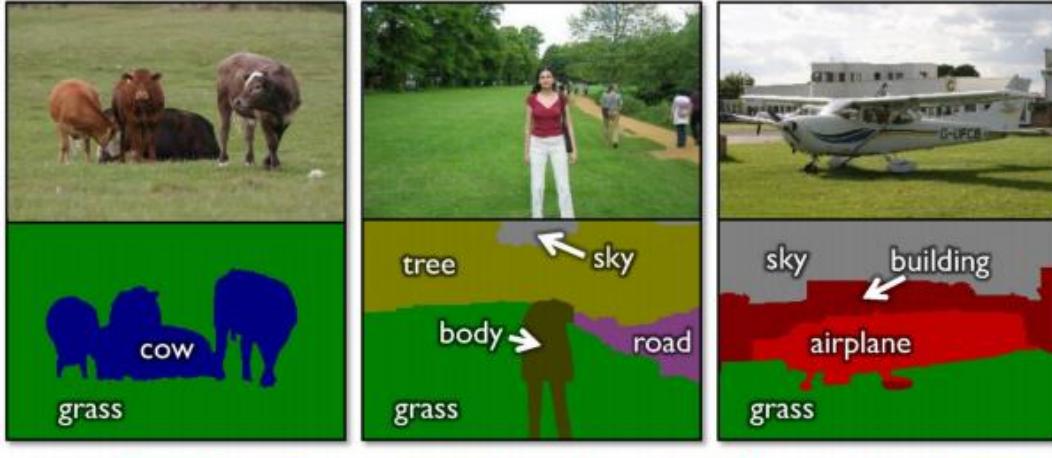


Problem:

We want the output to have the same resolution as the input!

Not hard if we never downsample (i.e., zero padding, stride 1, no pooling), but that is very expensive

The problem setup

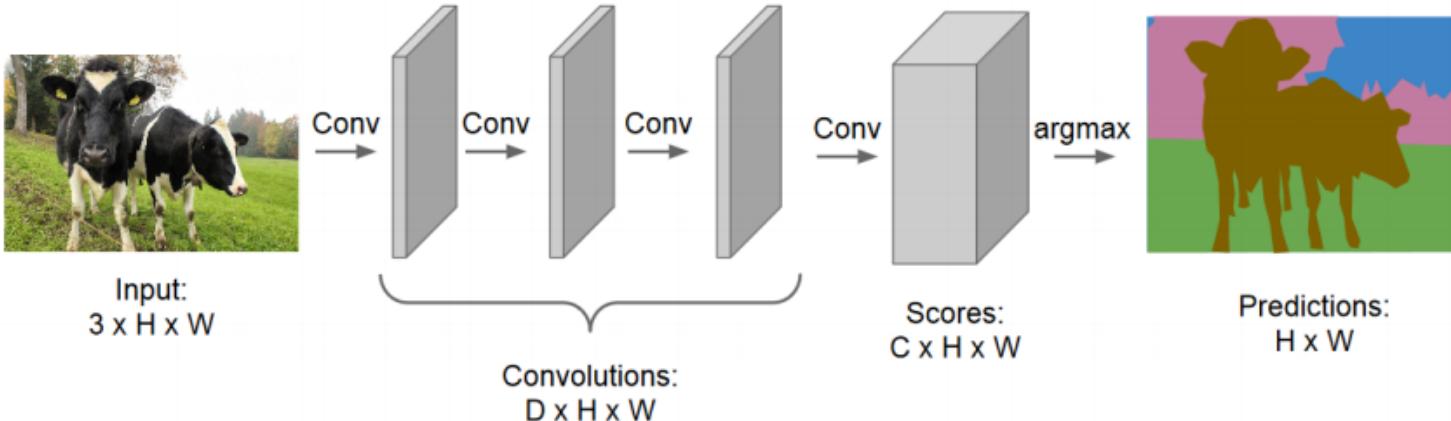


object classes	building	grass	tree	cow	sheep	sky	airplane	water	face	car
bicycle	flower	sign	bird	book	chair	road	cat	dog	body	boat

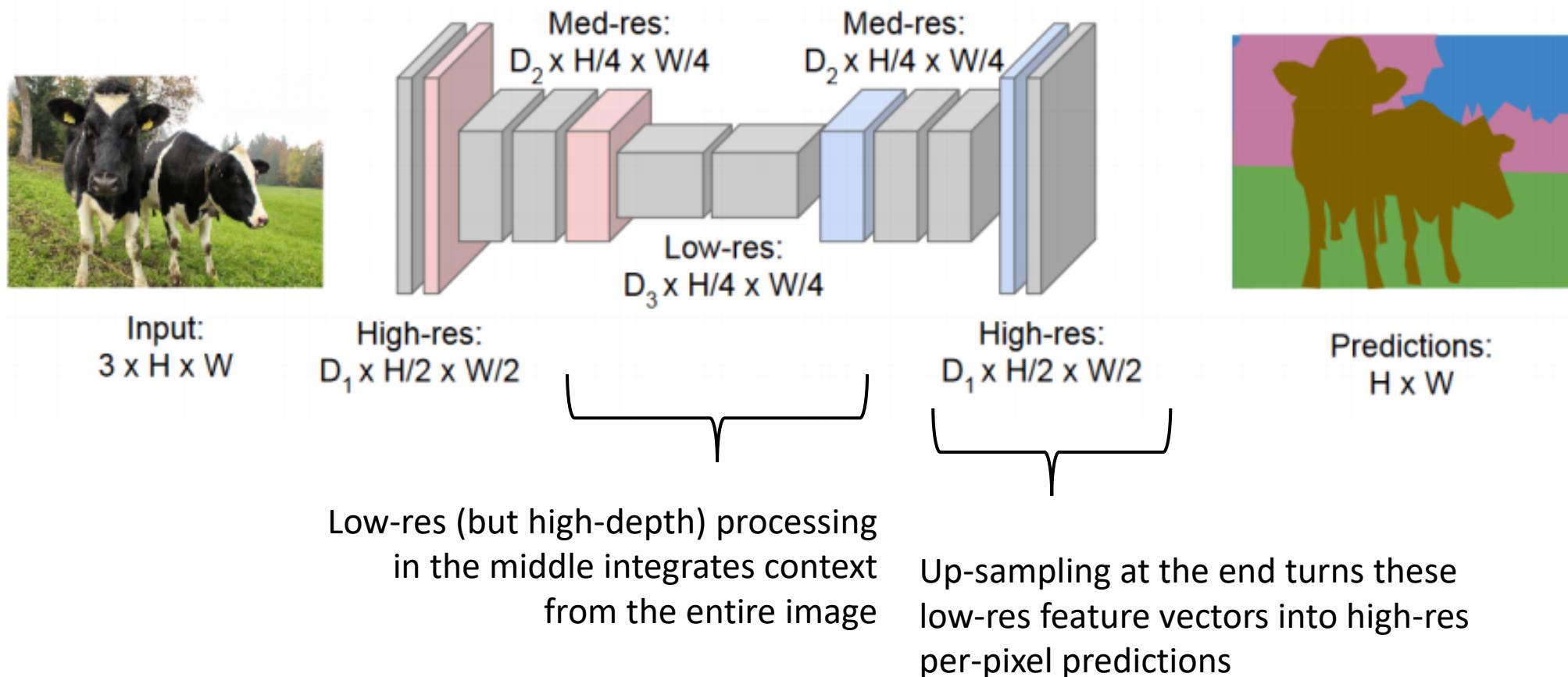
Classify every point with a class

Don't worry for now about instances
(e.g., two adjacent cows are just one "cow blob,"
and that's OK for some reason)

The challenge: design a network architecture
that makes this "per-pixel classification" problem
computationally tractable



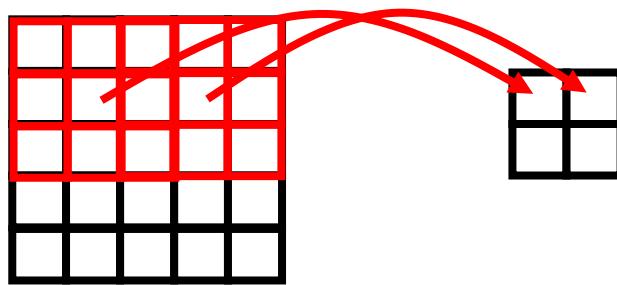
Fully convolutional networks



Up-sampling/transpose convolution

Normal convolutions: reduce resolution with **stride**

Stride = 2



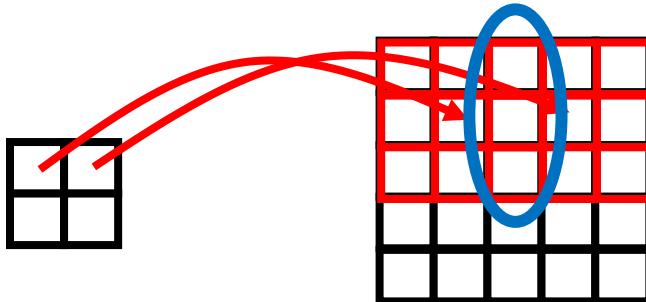
input: $H_f \times W_f \times C_{\text{in}}$

output: $1 \times 1 \times C_{\text{out}}$

filter: $H_f \times W_f \times C_{\text{in}} \times C_{\text{out}}$

Transpose convolutions: increase resolution with **fractional “stride”**

Stride = 1/2



we have two sets of values here! just average them

input: $1 \times 1 \times C_{\text{in}}$

output: $H_f \times W_f \times C_{\text{out}}$

filter: $C_{\text{in}} \times H_f \times W_f \times C_{\text{out}}$

Un-pooling

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

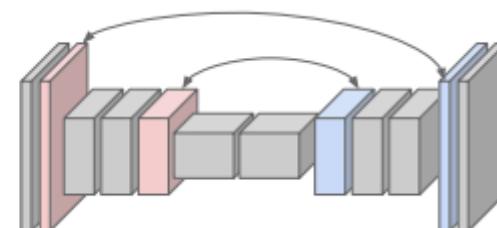
1	2
3	4

Rest of the network

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

Output: 4 x 4

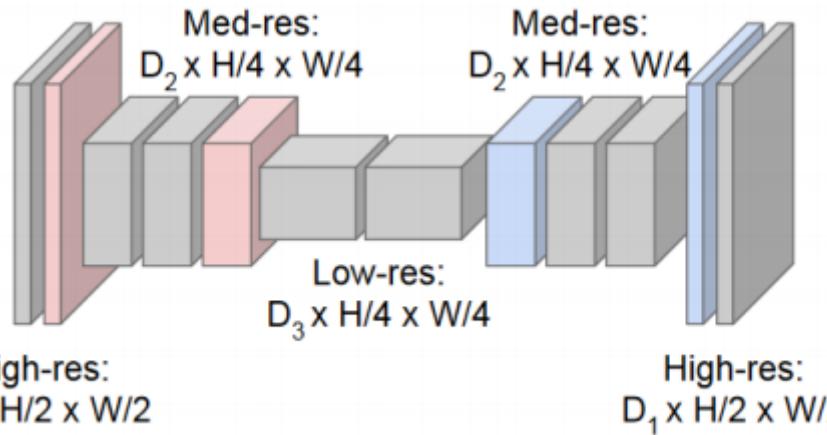
Corresponding pairs of
downsampling and
upsampling layers



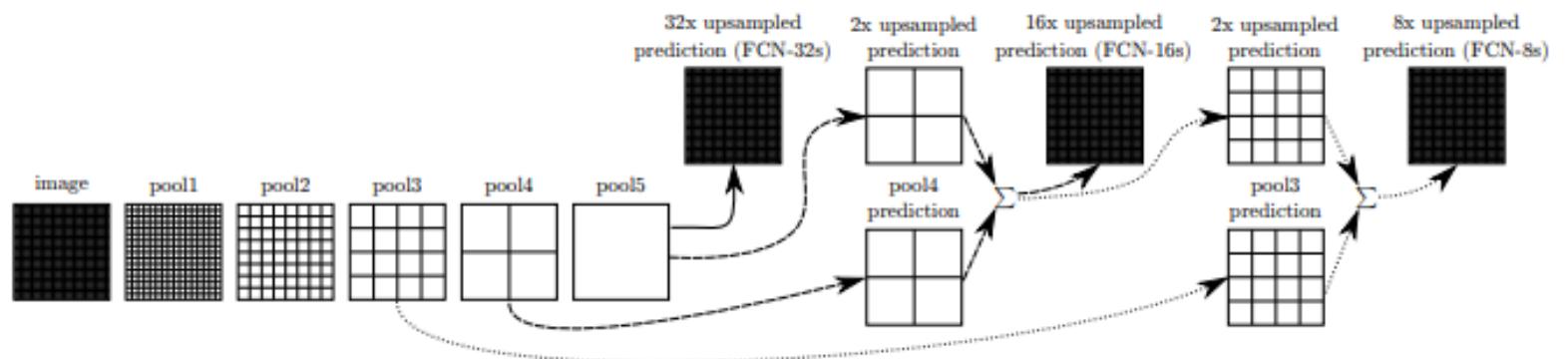
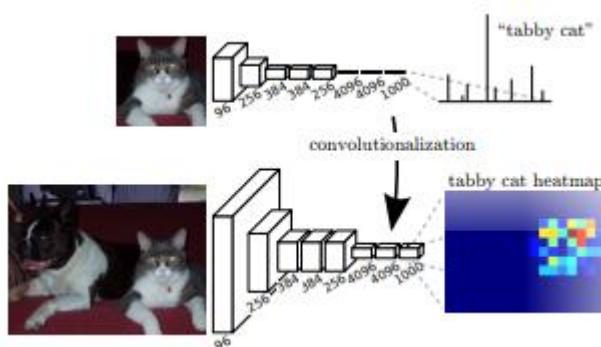
Bottleneck architecture



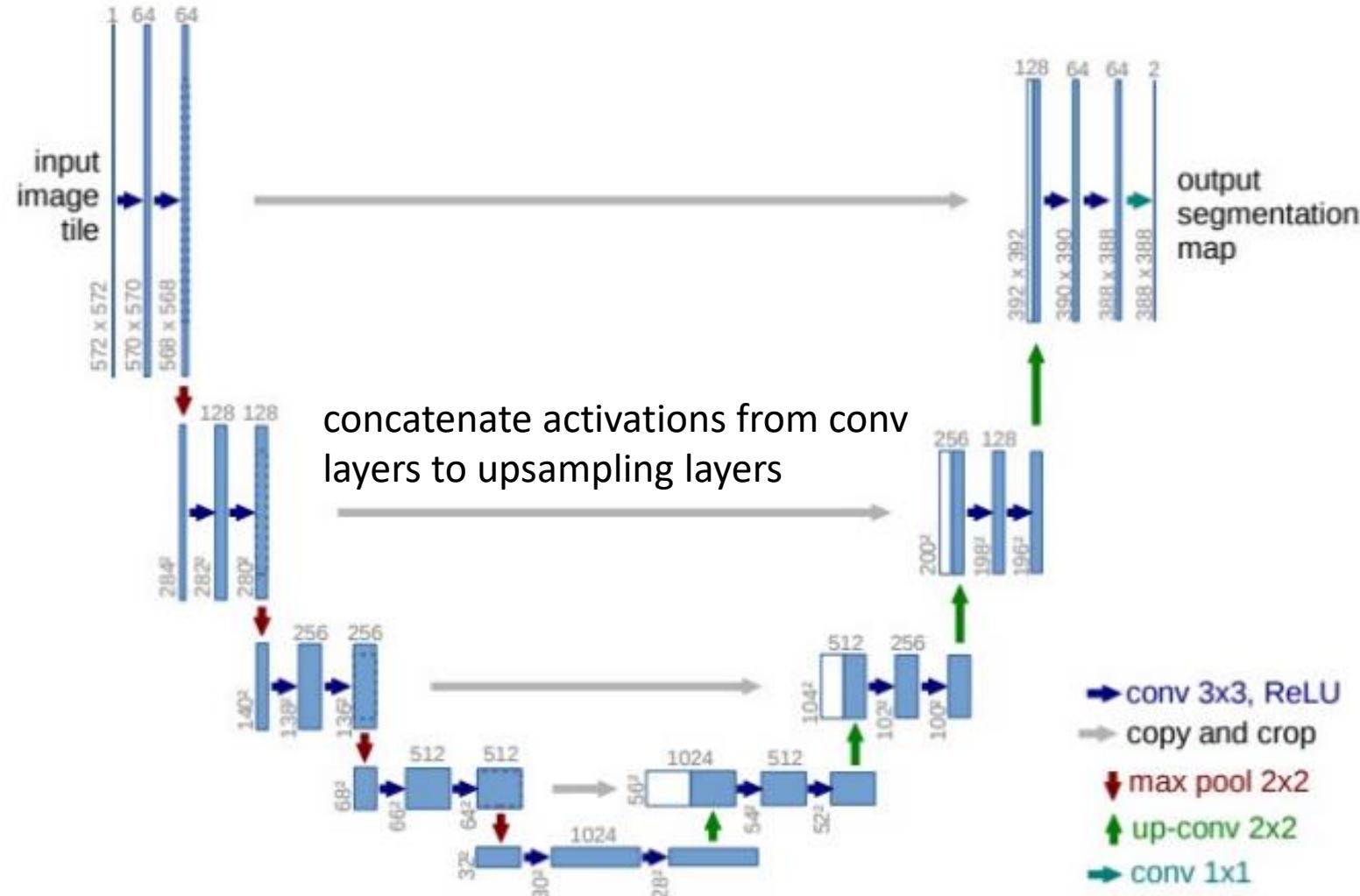
Input:
 $3 \times H \times W$



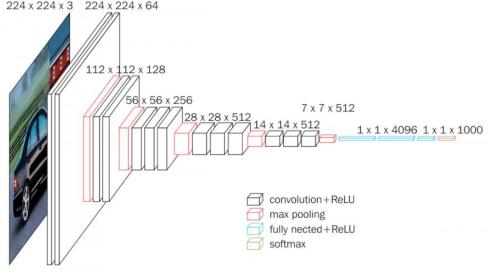
Predictions:
 $H \times W$



U-Net architecture



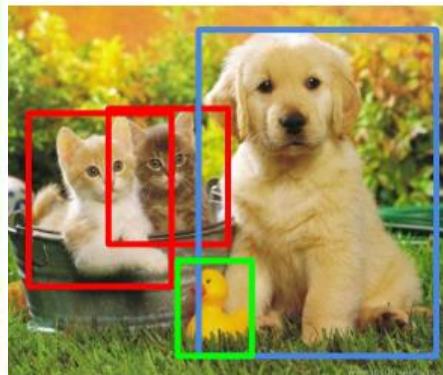
Standard computer vision problems



object classification



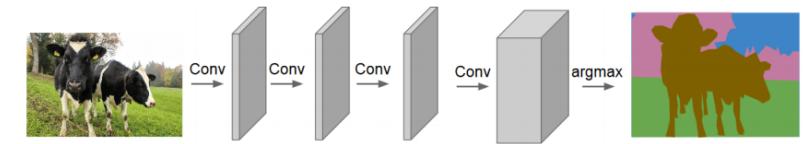
object localization



object detection



semantic segmentation
a.k.a. scene understanding



Generating Images from CNNs

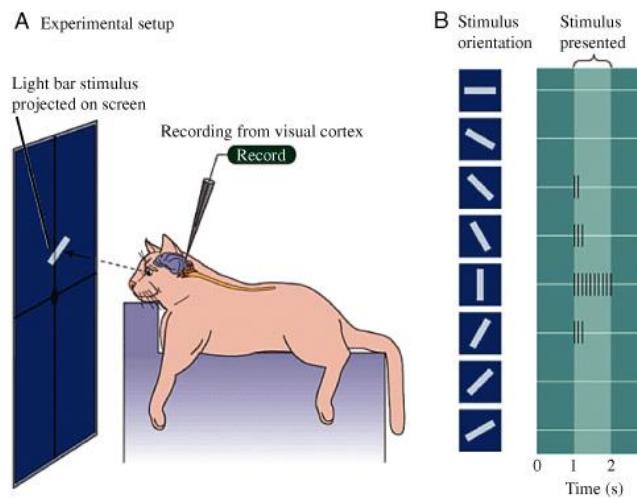
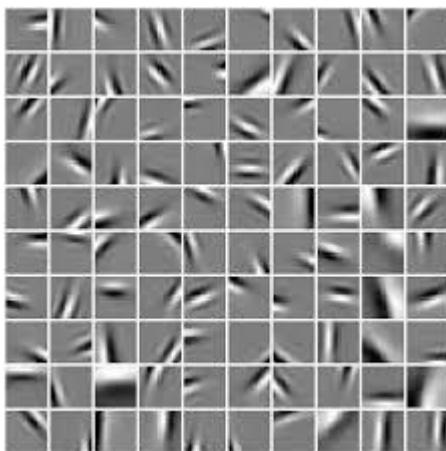
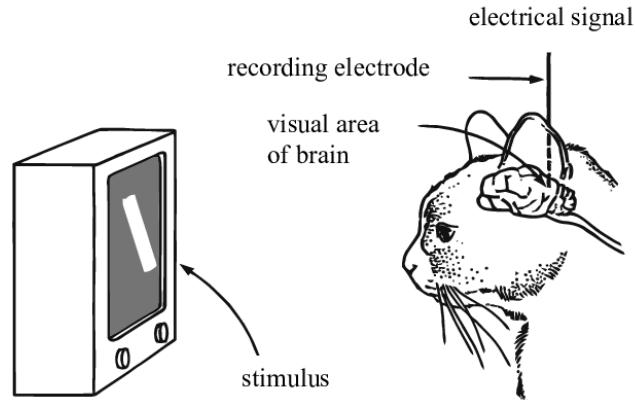
Designing, Visualizing and Understanding Deep Neural Networks

CS W182/282A

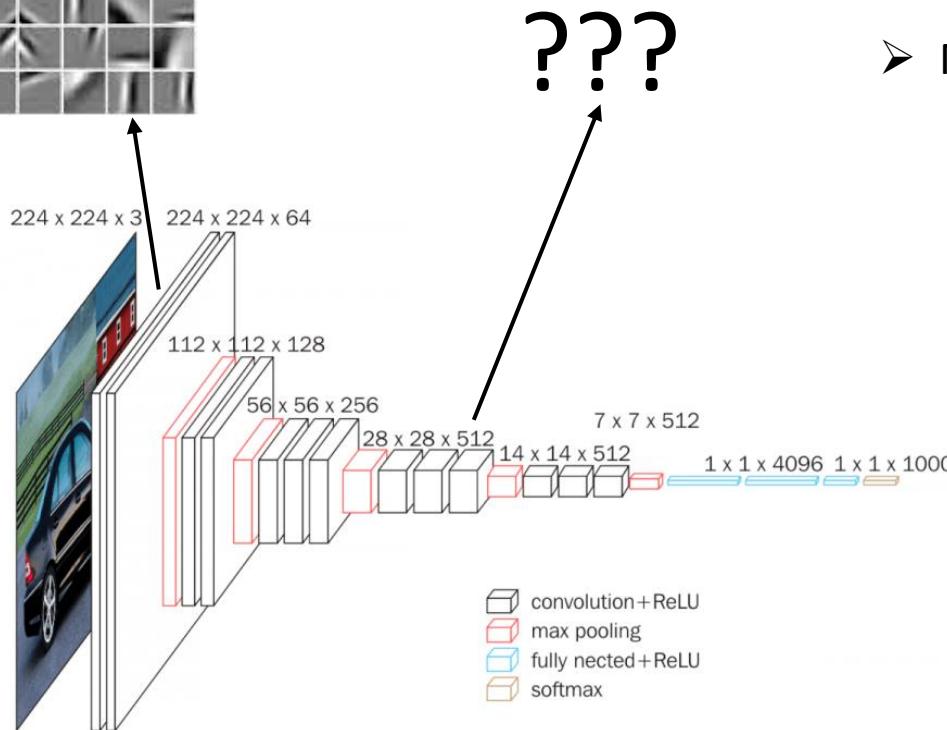
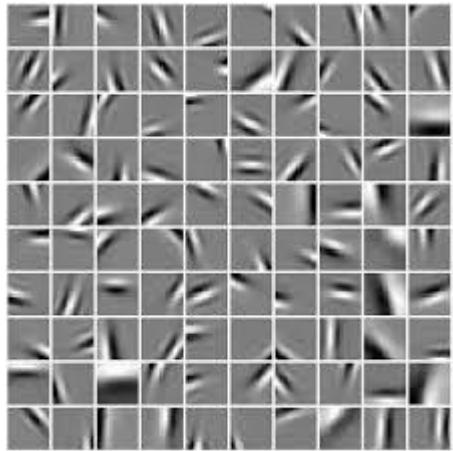
Instructor: Sergey Levine
UC Berkeley



What does the brain see?



What do convolutional networks “see”?



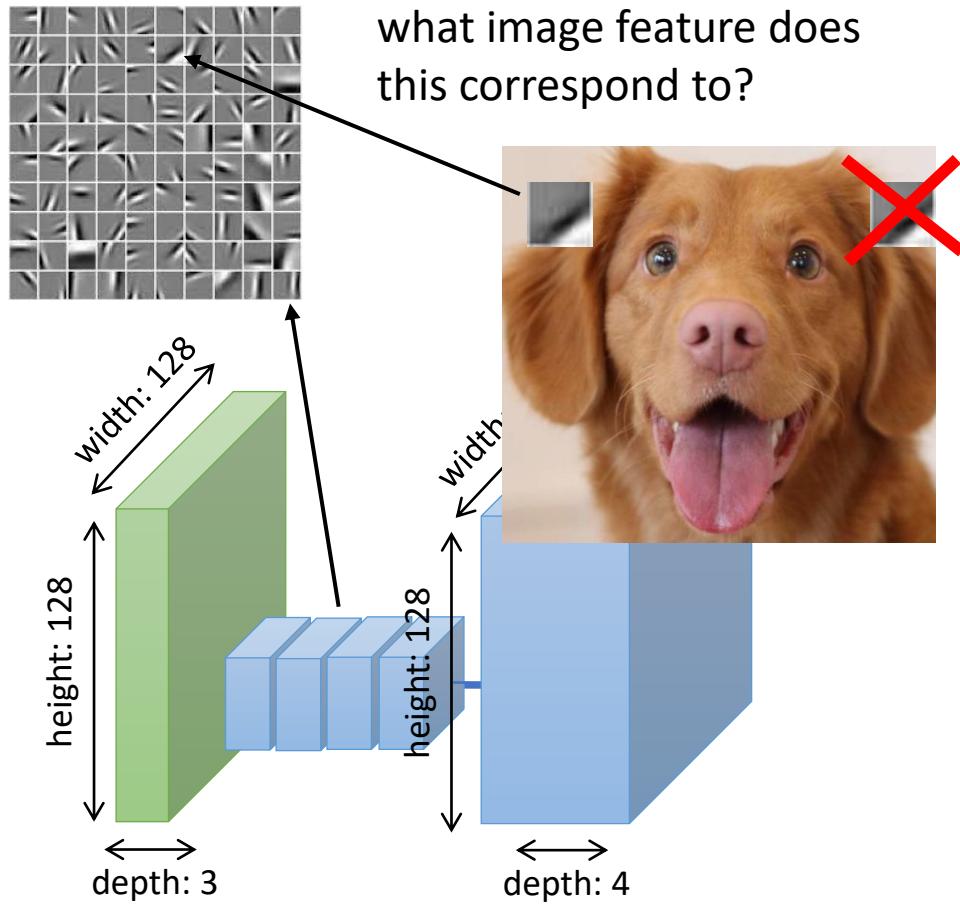
Why are we interested in this?

- Interpret what neural nets pay attention to
- Understand when they'll work and when they won't
- Compare different models and architectures
- Manipulate images based on conv net responses

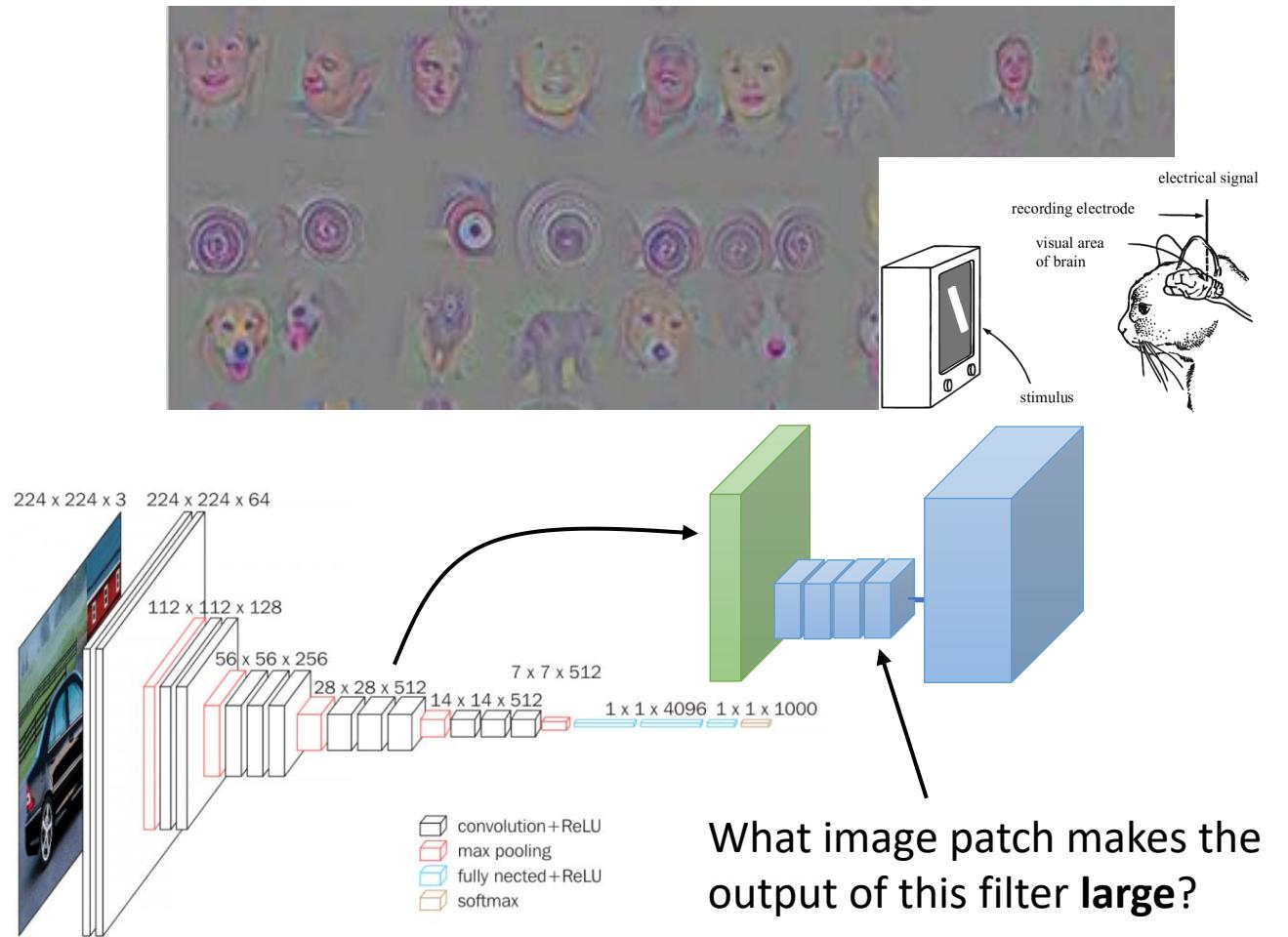


What do we visualize?

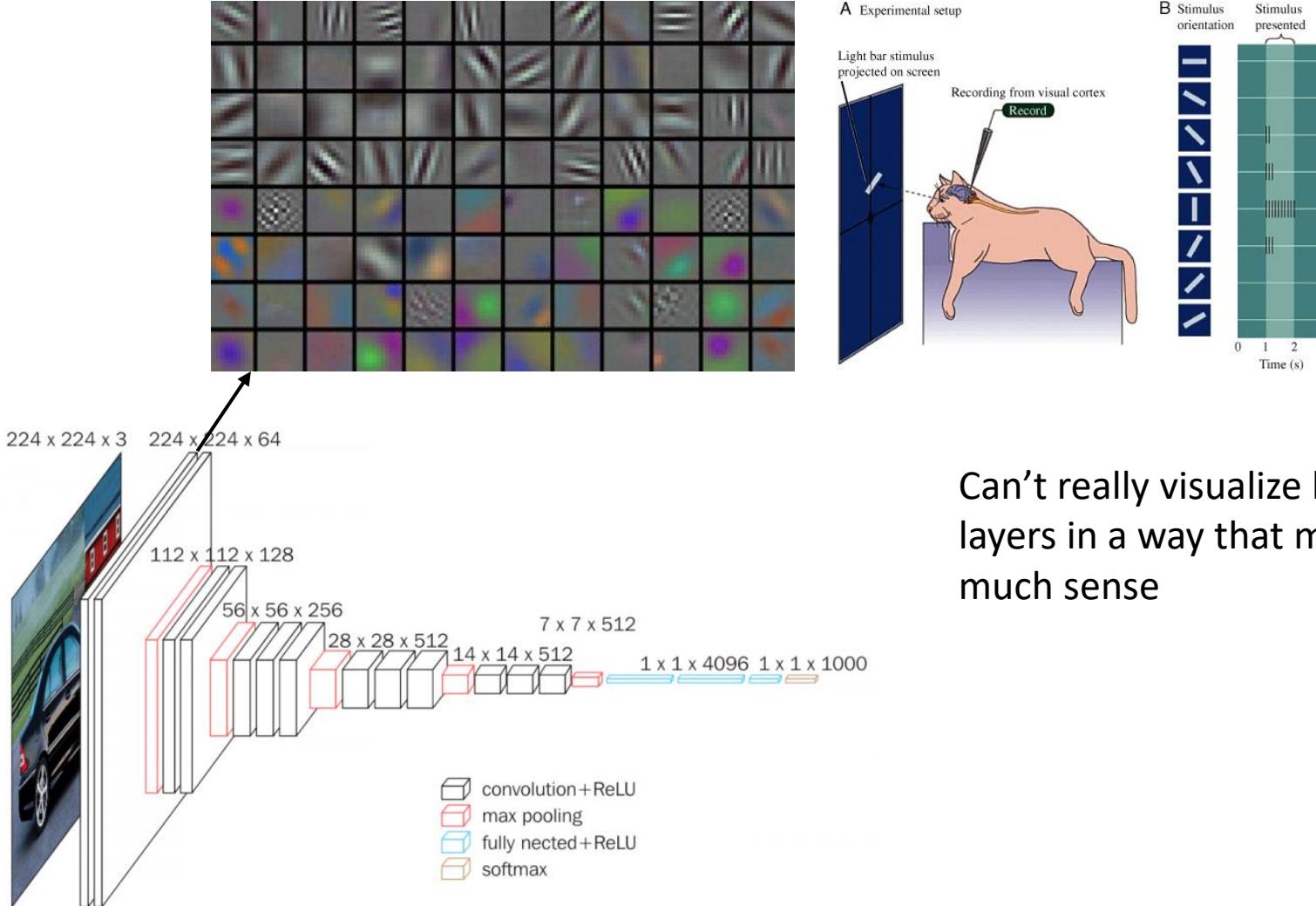
Option 1: visualize the **filter** itself



Option 2: visualize stimuli that activate a “neuron”



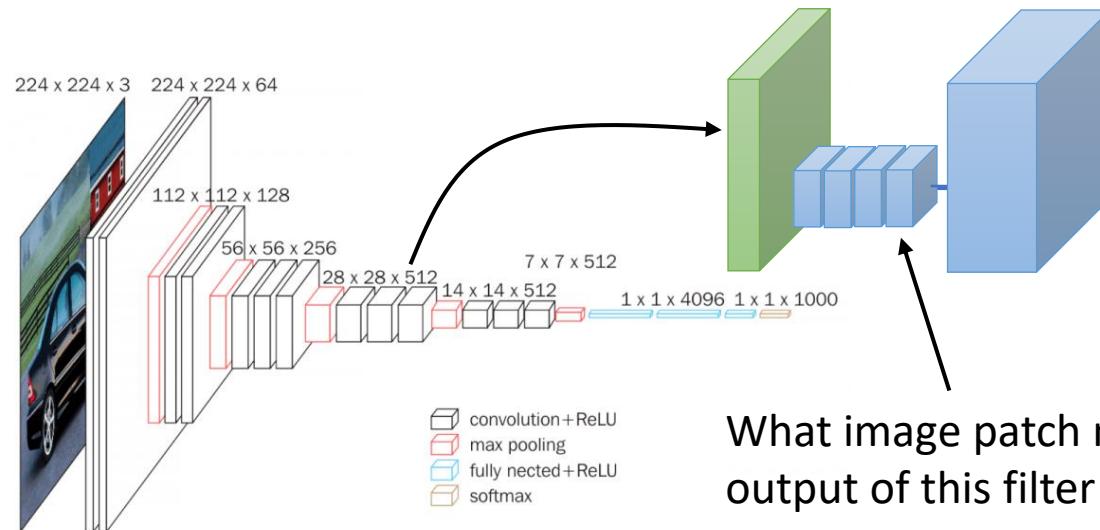
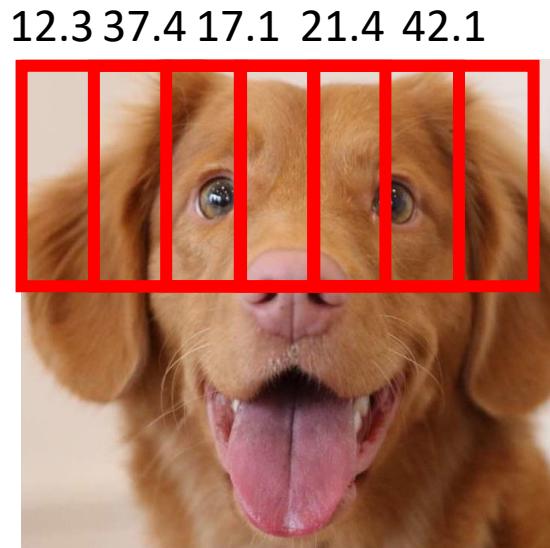
Visualizing filters



Can't really visualize higher layers in a way that makes much sense

Visualizing neuron responses

Idea 1: look for images that maximally “excite” specific units



What image patch makes the output of this filter **large**?

Visualizing neuron responses



Figure 4: Top regions for six pool_5 units. Receptive fields and activation values are drawn in white. Some units are aligned to concepts, such as people (row 1) or text (4). Other units capture texture and material properties, such as dot arrays (2) and specular reflections (6).

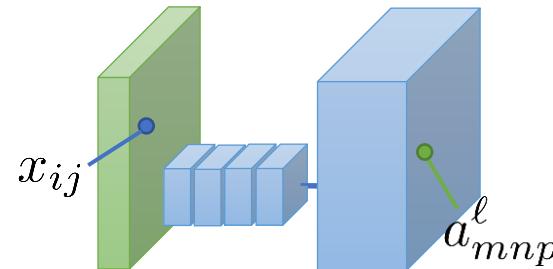
Using gradients for visualization

Idea 1: See which image pixels maximally influence the value at some unit

what does “influence” mean?

given a pixel x_{ij} and a unit a_{mnp}^ℓ

how much does changing x_{ij} change a_{mnp}^ℓ ?



$\frac{da_{mnp}^\ell}{dx_{ij}}$ how do we get this quantity?
backpropagation!

1. set δ to be same size as a^ℓ
2. set $\delta_{mnp} = 1$, all other entries to 0
3. backprop from layer ℓ to the image
4. last δ gives us $\frac{da_{mnp}^\ell}{dx}$

forward pass: calculate each $a^{(i)}$ and $z^{(i)}$

backward pass: $\frac{da_{mnp}^\ell}{da^\ell} \leftarrow$ zero in each position except mnp (which is 1)
initialize $\delta = \frac{d\mathcal{L}}{dz^{(\ell)}}$

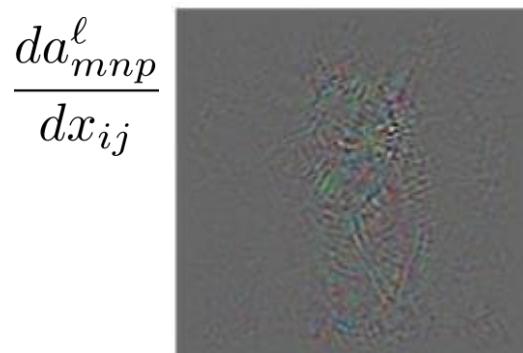
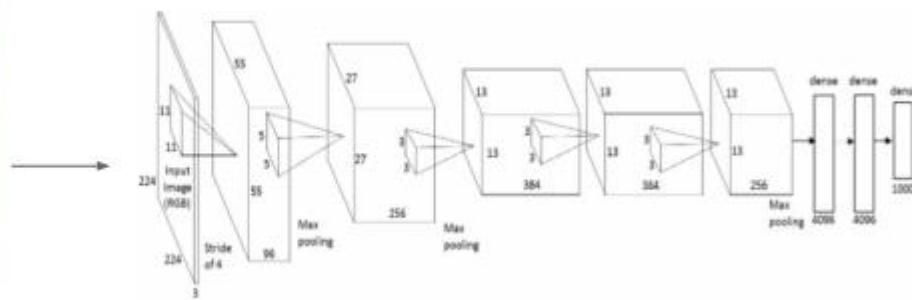
for each f with input x_f & params θ_f from end to start:

$$\begin{aligned}\frac{d\mathcal{L}}{d\theta_f} &\leftarrow \frac{df}{d\theta_f} \delta \\ \delta &\leftarrow \frac{df}{dx_f} \delta\end{aligned}$$

Using gradients for visualization

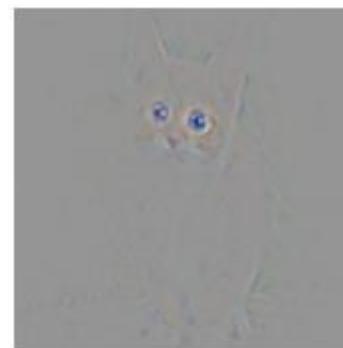
Idea 1: See which image pixels maximally influence the value at some unit

$$\frac{da_{mnp}^\ell}{dx_{ij}}$$



slightly modified
backprop gives
us this:

“guided backpropagation”



basic idea behind “guided backpropagation”:

- Just using backprop is not very interpretable, because many units in the network contribute **positive or negative** gradients
- If we keep just the **positive** gradients, we'll avoid some of the complicated negative contributions, and get a cleaner signal

heuristically change backward step in ReLU:

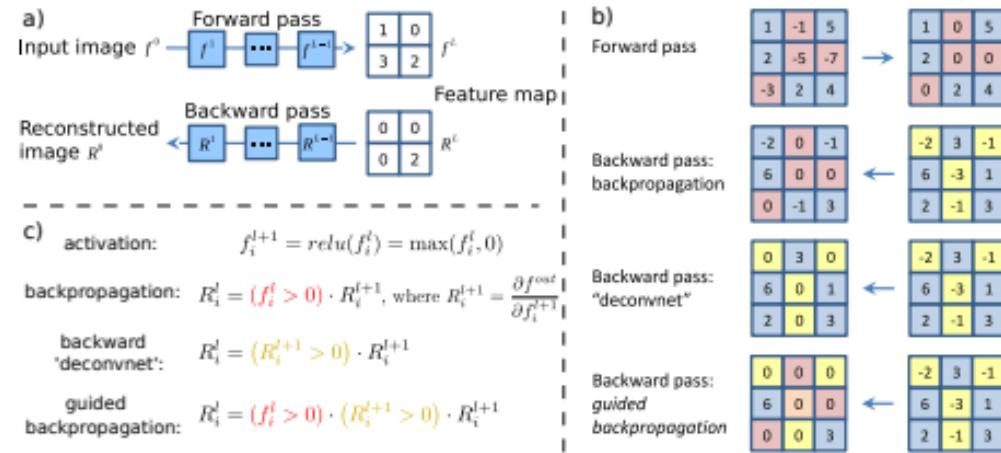
for each entry δ_{ijk} , set it to $\max(0, \delta_{ijk})$
“zero out negative gradients at each layer”

Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller. **Striving for Simplicity: The All Convolutional Net.** 2014.

Using gradients for visualization

Idea 1: See which image pixels maximally influence the value at some unit

$$\frac{da_m^l}{dx_{ij}}$$



basic idea behind “guided backpropagation”:

- Just using backprop is not very interpretable, because many units in the network contribute **positive or negative** gradients
- If we keep just the **positive** gradients, we’ll avoid some of the complicated negative contributions, and get a cleaner signal

heuristically change backward step in ReLU:
 for each entry δ_{ijk} , set it to $\max(0, \delta_{ijk})$
 “zero out negative gradients at each layer”

Figure 1: Schematic of visualizing the activations of high layer neurons. a) Given an input image, we perform the forward pass to the layer we are interested in, then set to zero all activations except one and propagate back to the image to get a reconstruction. b) Different methods of propagating back through a ReLU nonlinearity. c) Formal definition of different methods for propagating a output activation *out* back through a ReLU unit in layer l ; note that the ‘deconvnet’ approach and guided backpropagation do not compute a true gradient but rather an imputed version.

Using gradients for visualization

Idea 1: See which image pixels maximally influence the value at some unit

guided backpropagation



corresponding image crops



CONV6

guided backpropagation



corresponding image crops



CONV9

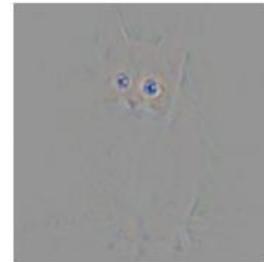
Visualizing Features with Backpropagation

Using gradients for visualization

Idea 2: “Optimize” the image to maximally activate a particular unit

Before: just compute

$$\frac{da_{mnp}^\ell}{dx_{ij}}$$



Now: compute

$$g = \frac{da_{mnp}^\ell}{dx}$$

$$x \leftarrow x + \alpha g$$



what optimization problem is this solving?

$$x \leftarrow \arg \max_x a_{mnp}^\ell(x)$$

activation a_{mnp}^ℓ for image x

$$\text{more generally: } x \leftarrow \arg \max_x S(x)$$

some activation or class label

Using gradients for visualization

Idea 2: “Optimize” the image to maximally activate a particular unit

$$x \leftarrow \arg \max_x S(x) + R(x)$$



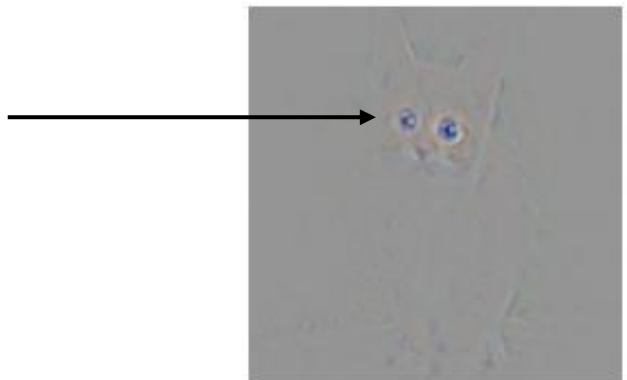
image regularizer to
prevent crazy images

simple choice: $R(x) = \lambda \|x\|_2^2$

problem: it's too easy to produce “crazy” images

making the eyes more blue
increases activation

what if we set blue
channel to 1,000,000,000?



Visualizing “classes”

$$x \leftarrow \arg \max_x S(x) + R(x)$$



simple choice: $R(x) = \lambda ||x||_2^2$

Which unit to maximize?

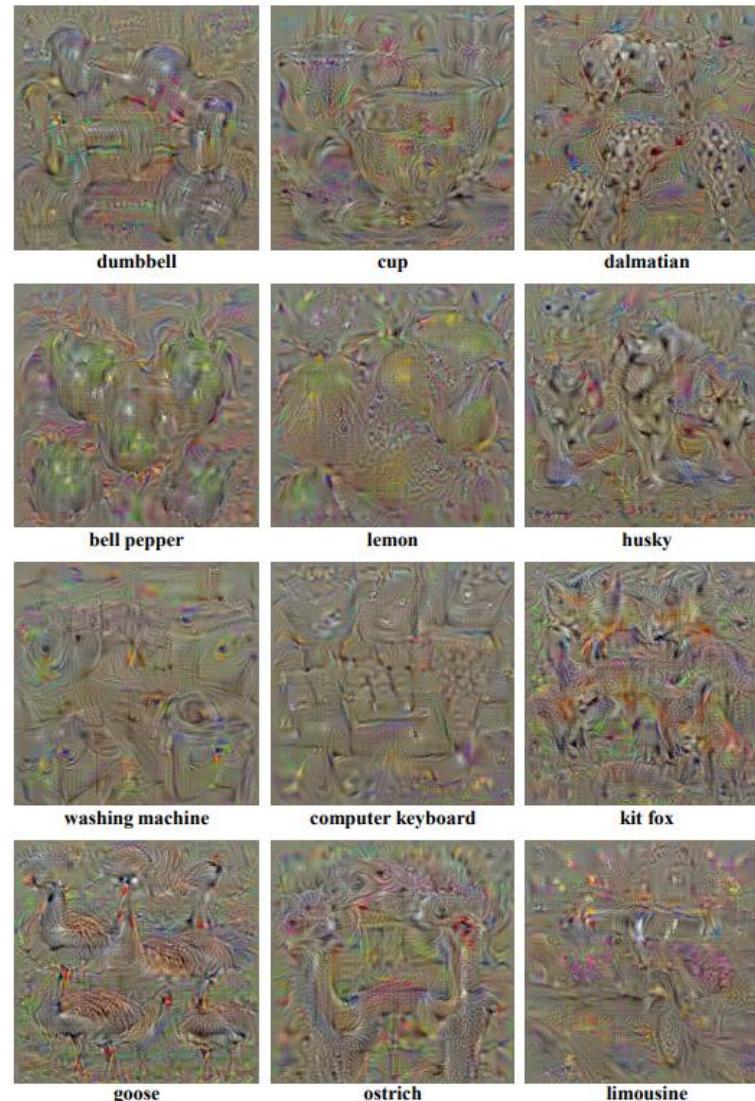
Let's try maximizing class labels first

Important detail: not maximizing **class probabilities**, but the activations **right before** the softmax

$$p(y|x) = \text{softmax}(\underbrace{W^\ell a^\ell(x)}_{\text{not this}} + \underbrace{b^\ell}_{\text{this}})$$

not this

this



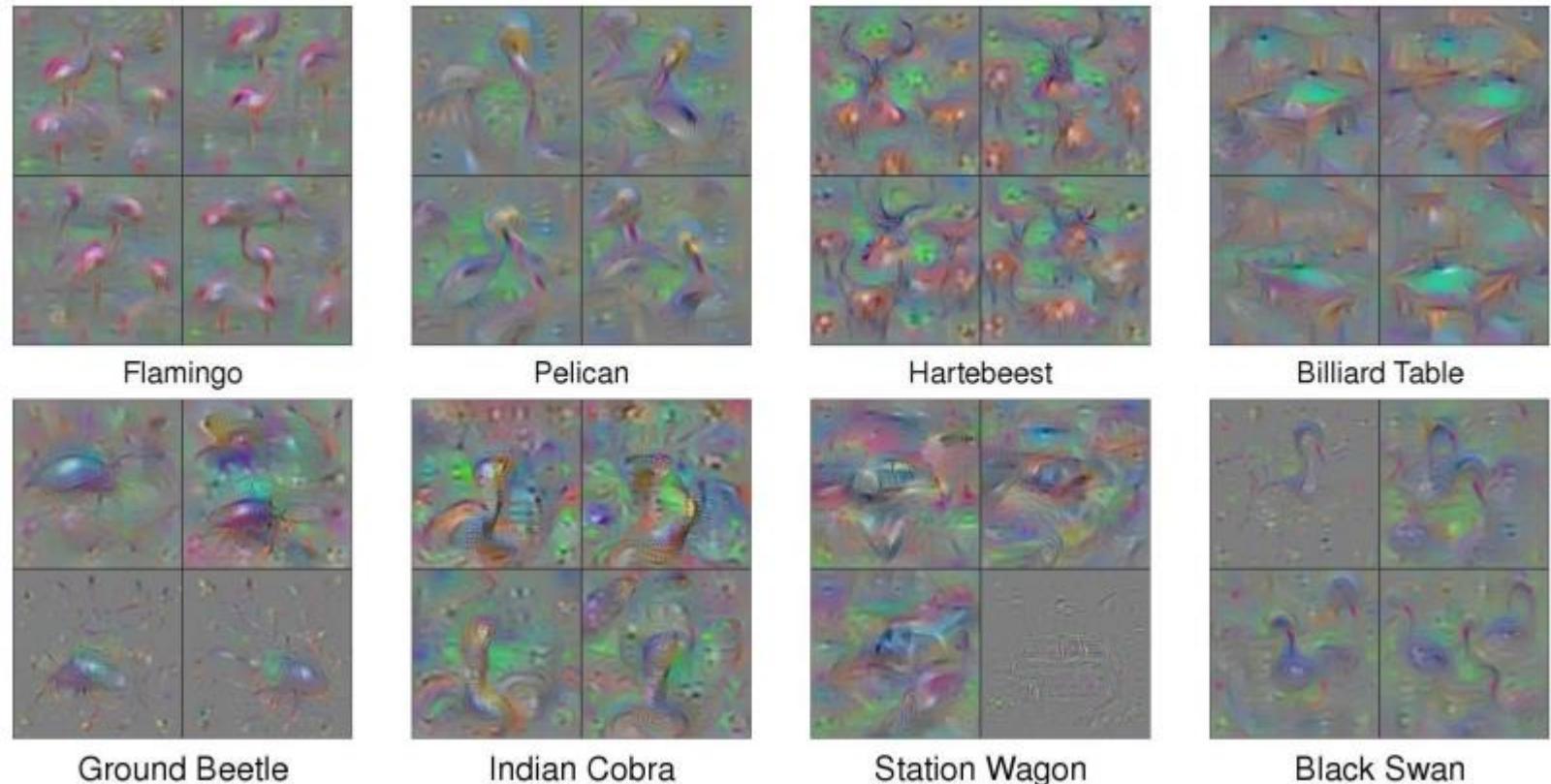
Visualizing “classes”

$$x \leftarrow \arg \max_x S(x) + R(x)$$



slightly more nuanced
regularizer

1. Update image with gradient
2. Blur the image a little
3. Zero out any pixel with small value
4. Repeat

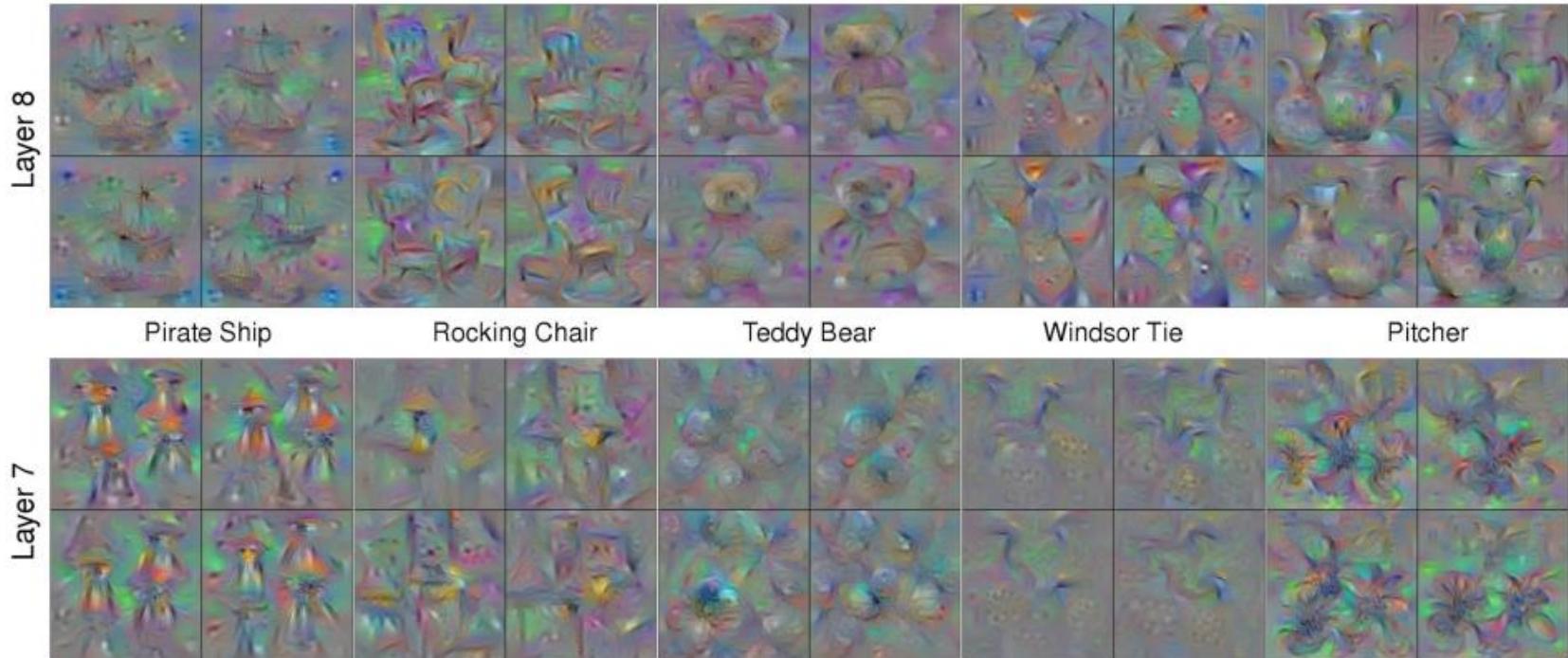


Visualizing units

$$x \leftarrow \arg \max_x S(x) + R(x)$$

↑
slightly more nuanced
regularizer

1. Update image with gradient
2. Blur the image a little
3. Zero out any pixel with small value
4. Repeat

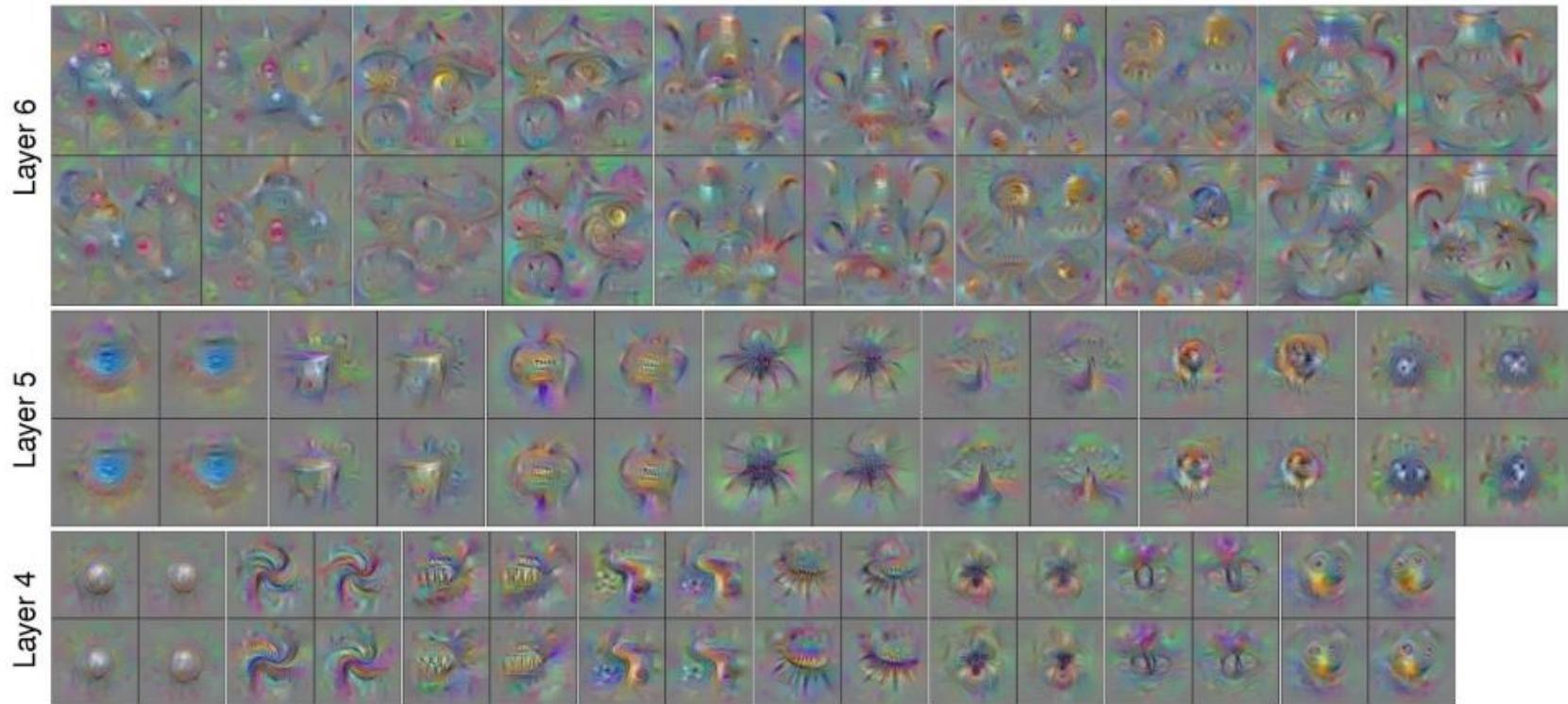


Visualizing units

$$x \leftarrow \arg \max_x S(x) + R(x)$$

↑
slightly more nuanced
regularizer

1. Update image with gradient
2. Blur the image a little
3. Zero out any pixel with small value
4. Repeat

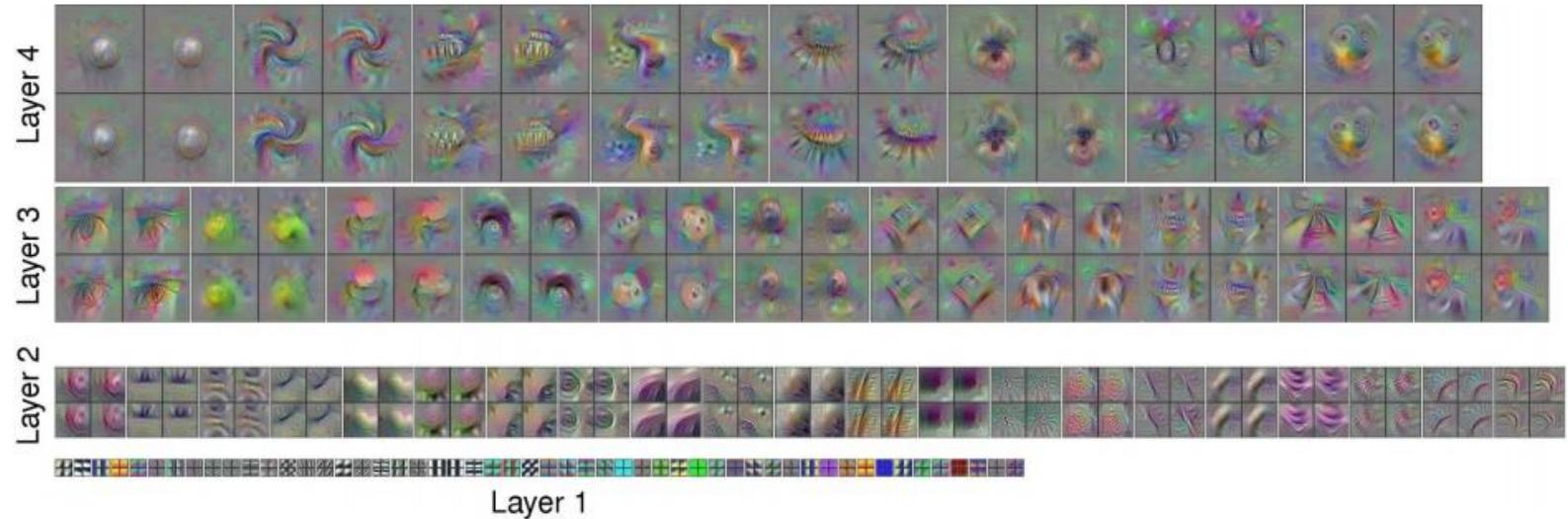


Visualizing units

$$x \leftarrow \arg \max_x S(x) + R(x)$$

↑
slightly more nuanced
regularizer

1. Update image with gradient
2. Blur the image a little
3. Zero out any pixel with small value
4. Repeat



Deep Dream & Style Transfer

Using backprop to *modify* pictures?

Before:



Now:



What is channel 17 in layer conv5 looking at?



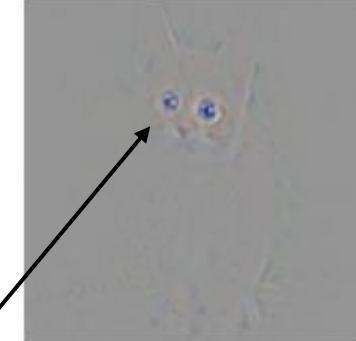
How are these questions related?

Intuition: Monet paintings have particular feature distributions, we can “transport” these distributions to other images!

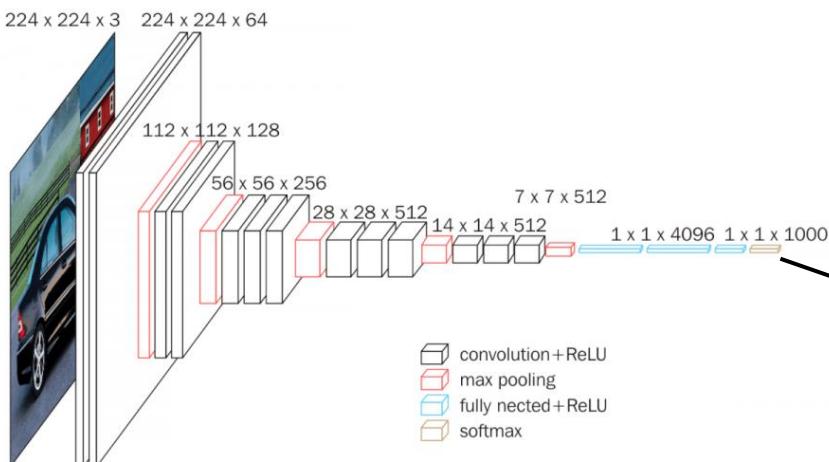


What would it look like if it were a Monet painting?

Looking for patterns in clouds

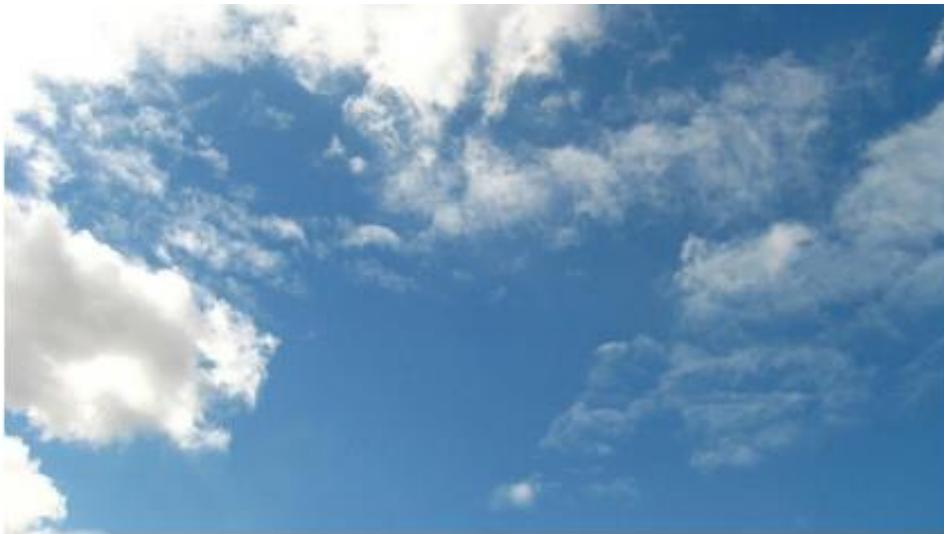


making the eyes more blue will
make this more cat-like



“looks kind of like a dog”
“where is the dog?”
“let me show you!”

DeepDream



1. Pick a layer
2. Run forward pass to compute activations at that layer
3. Set delta to be **equal** to the activations
4. Backprop and apply the gradient
5. Repeat

DeepDream



inception_4c/output



DeepDream

```
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)
```

DeepDream: set $dx = x$:)

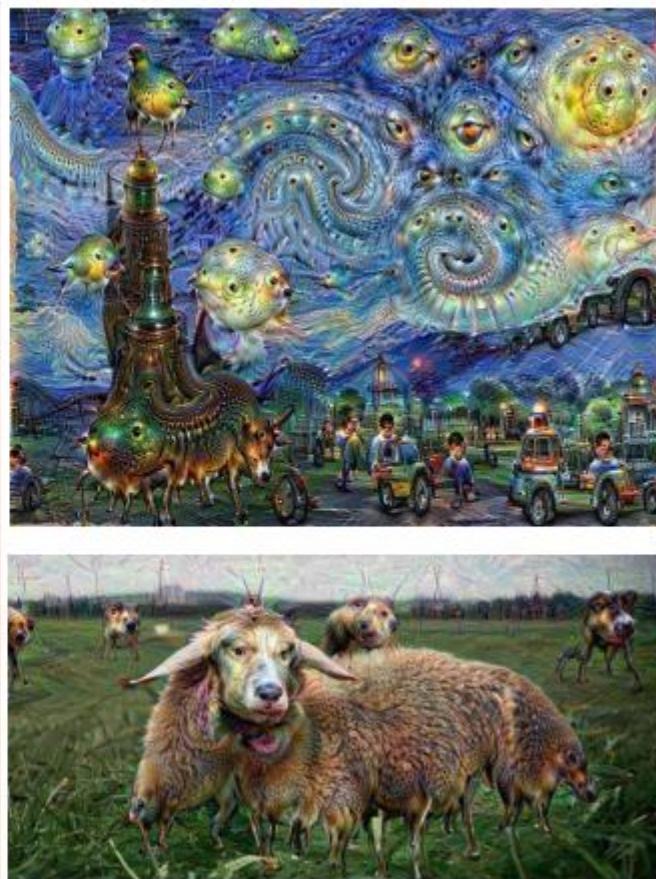
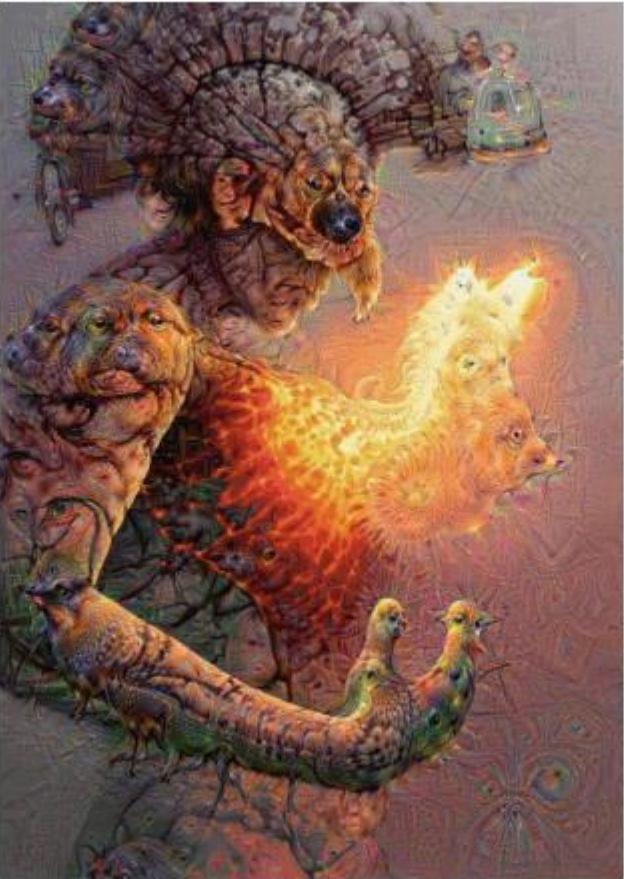
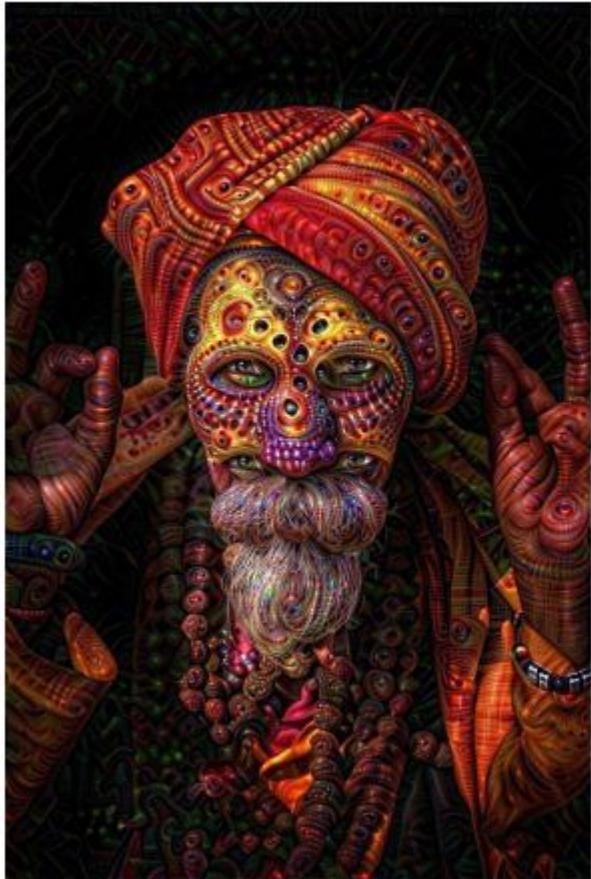
Remember this:

$$x \leftarrow \arg \max_x S(x) + R(x)$$

jitter regularizer

“image update”

DeepDream



DeepDream <https://github.com/google/deepdream>

Another idea

Another idea: instead of exaggerating the features in a single image, what if we make feature of one image look more like the features in **another**?



how do we **quantify** style?

style

→ relationships between features
e.g., lots of angles and straight lines



content

→ spatial positions of features
e.g., where curves, edges, and corners are located



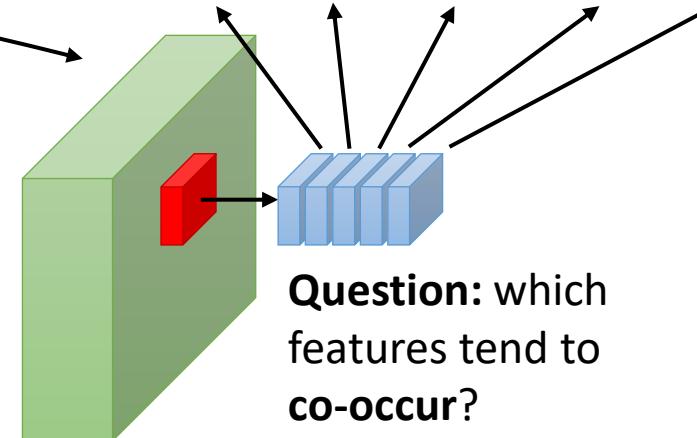
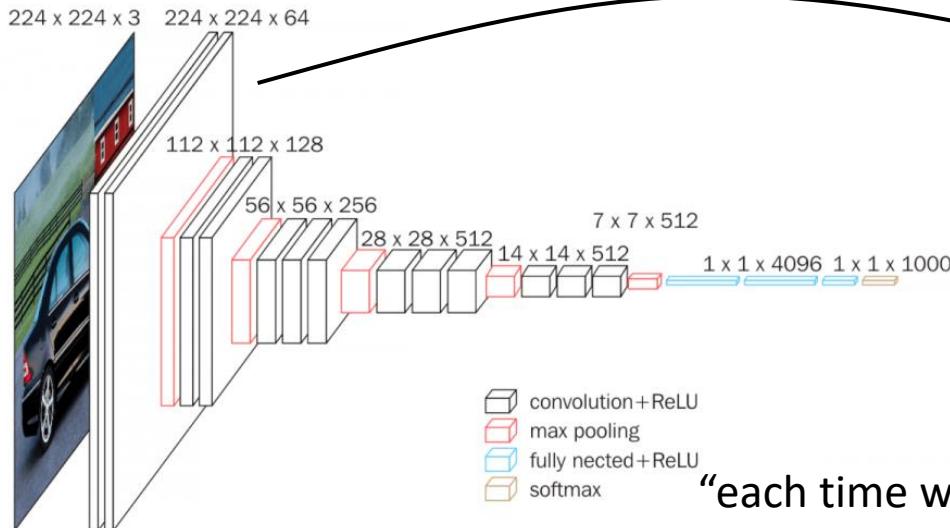
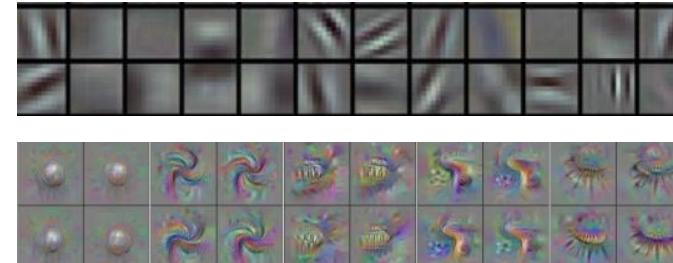
This really works???

How do we quantify style?



style

relationships between features
e.g., lots of angles and straight lines



“each time we see a straight line, we also see this texture”
“each time we see a curve, we also see flat shading”

How do we quantify this?

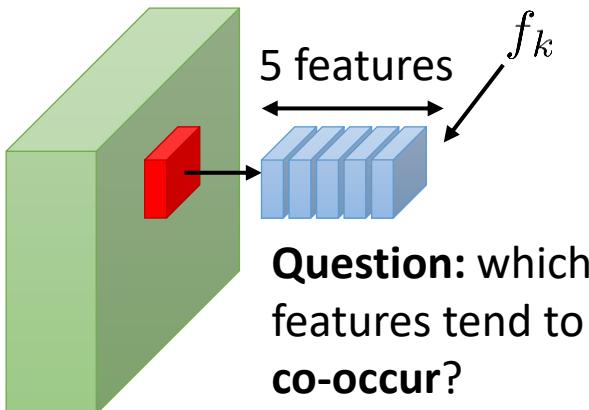
How do we quantify style?



style

relationships between
features
e.g., lots of angles and
straight lines

estimate by averaging over all positions in the image



feature covariance: $\text{Cov}_{km} = E[f_k f_m]$

form Gram matrix: $G_{km} = \text{Cov}_{km}$

If features have this covariance, then we have the right style

Surprising but true!

“each time we see a straight line, we also see this texture”

“each time we see a curve, we also see flat shading”

How do we quantify style?



style

relationships between
features
e.g., lots of angles and
straight lines

feature covariance: $\text{Cov}_{km} = E[f_k f_m]$

form Gram matrix: $G_{km} = \text{Cov}_{km}$ ← this comes from the style source image

new image: $x \leftarrow \arg \min_x \mathcal{L}_{\text{style}}(x) + \mathcal{L}_{\text{content}}(x)$

G^ℓ : source image Gram matrix at layer ℓ

$A^\ell(x)$: new image Gram matrix at layer ℓ

$$\mathcal{L}_{\text{style}}(x) = \sum_{\ell} \sum_{km} (G_{km}^\ell - A_{km}^\ell(x))^2 w_\ell$$

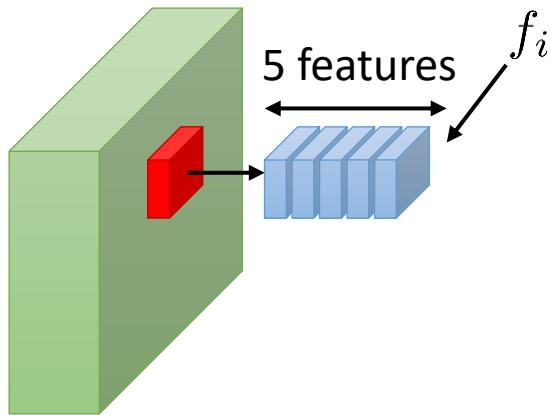
Different weight on each
layer, to prioritize relative
contribution to (desired)
style of different levels of
abstraction

How do we quantify content?



content

spatial positions of
features
e.g., where curves,
edges, and corners
are located



Pick specific layer for matching content

Just directly match the features!

$$\mathcal{L}_{\text{content}}(x) = \sum_{ij} \sum_k (f_{ijk}^\ell(x_{\text{content}}) - f_{ijk}^\ell(x))^2$$

$$\mathcal{L}_{\text{style}}(x) = \sum_\ell \sum_{km} (G_{km}^\ell - A_{km}^\ell(x))^2 w_\ell$$

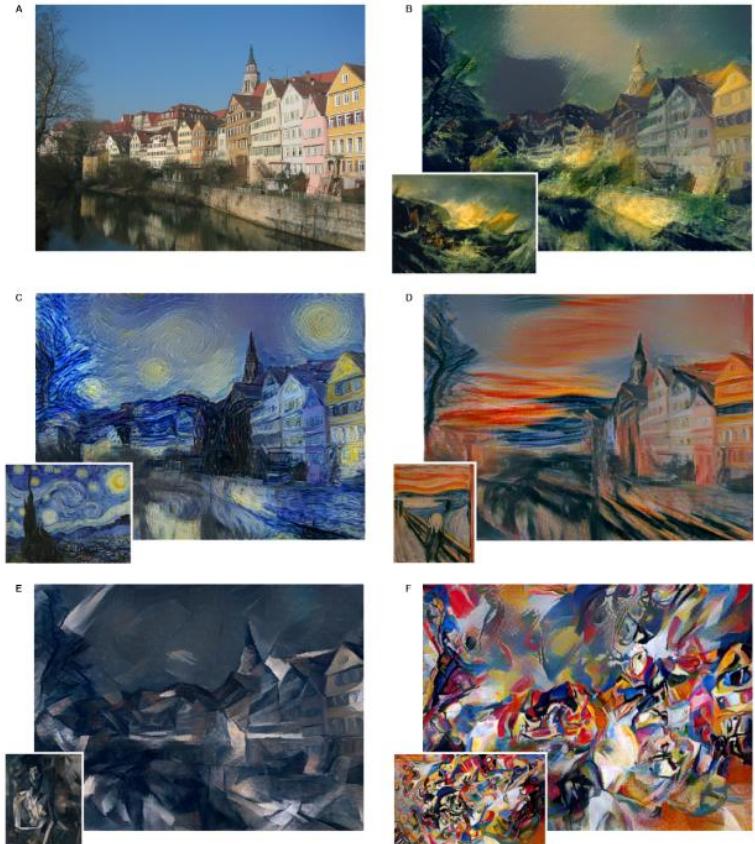
new image: $x \leftarrow \arg \min_x \mathcal{L}_{\text{style}}(x) + \mathcal{L}_{\text{content}}(x)$

Style transfer

new image: $x \leftarrow \arg \min_x \mathcal{L}_{\text{style}}(x) + \mathcal{L}_{\text{content}}(x)$

$$\mathcal{L}_{\text{content}}(x) = \sum_{ij} \sum_k (f_{ijk}^\ell(x_{\text{content}}) - f_{ijk}^\ell(x))^2$$

$$\mathcal{L}_{\text{style}}(x) = \sum_\ell \sum_{km} (G_{km}^\ell - A_{km}^\ell(x))^2 w_\ell$$



make your own easily on deepart.io

Recurrent Networks

Designing, Visualizing and Understanding Deep Neural Networks

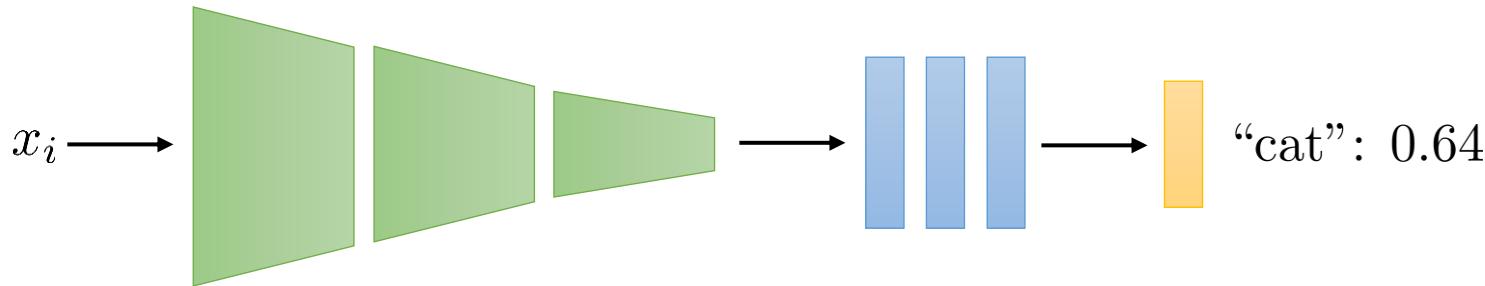
CS W182/282A

Instructor: Sergey Levine
UC Berkeley



What if we have variable-size inputs?

Before:



Now:

$$x_1 = (x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4})$$

$$x_2 = (x_{2,1}, x_{2,2}, x_{2,3})$$

$$x_3 = (x_{3,1}, x_{3,2}, x_{3,3}, x_{3,4}, x_{3,5})$$

Examples:

classifying sentiment for a phrase (sequence of words)

recognizing phoneme from sound (sequence of sounds)

classifying the activity in a video (sequence of images)

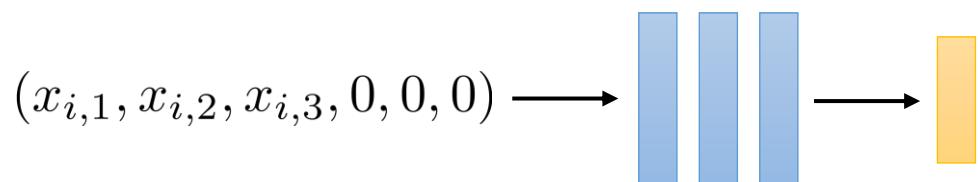
What if we have variable-size inputs?

$$x_1 = (x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4})$$

$$x_2 = (x_{2,1}, x_{2,2}, x_{2,3})$$

$$x_3 = (x_{3,1}, x_{3,2}, x_{3,3}, x_{3,4}, x_{3,5})$$

Simple idea: zero-pad up to length of longest sequence



+ very simple, and can work in a pinch

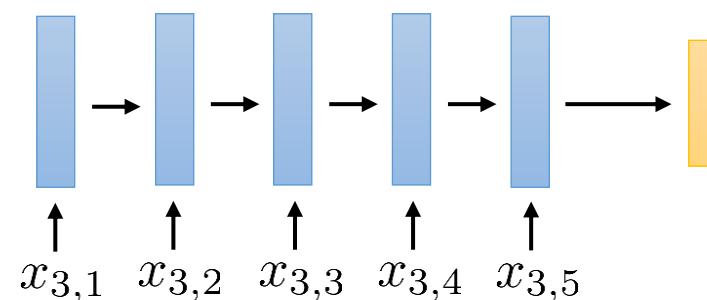
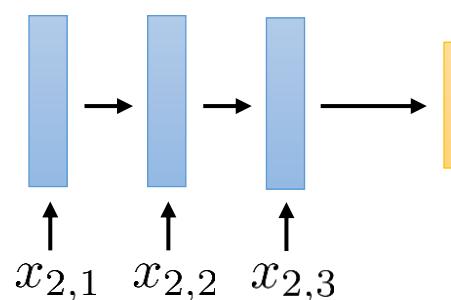
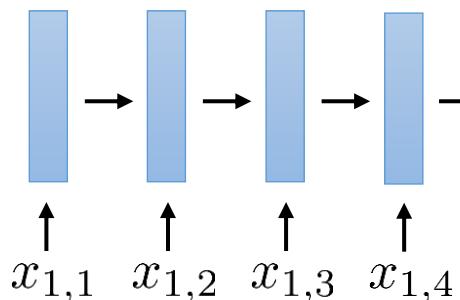
- doesn't scale very well for very long sequences

One input per layer?

$$x_1 = (x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4})$$

$$x_2 = (x_{2,1}, x_{2,2}, x_{2,3})$$

$$x_3 = (x_{3,1}, x_{3,2}, x_{3,3}, x_{3,4}, x_{3,5})$$



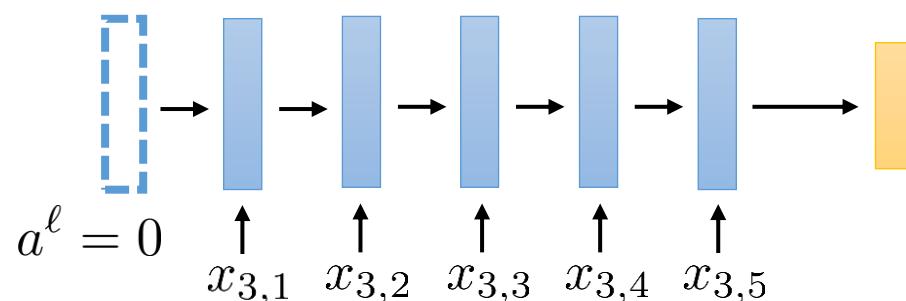
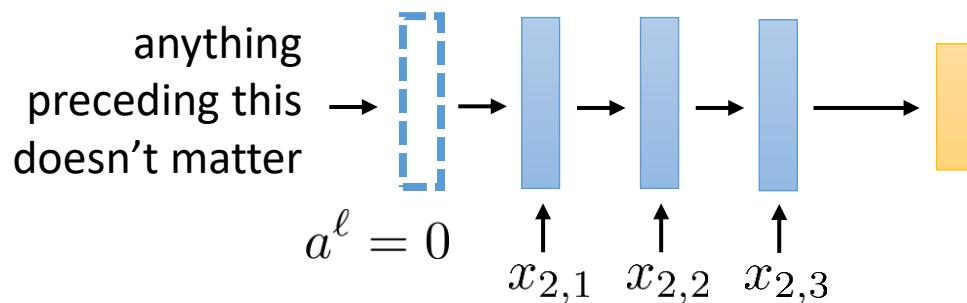
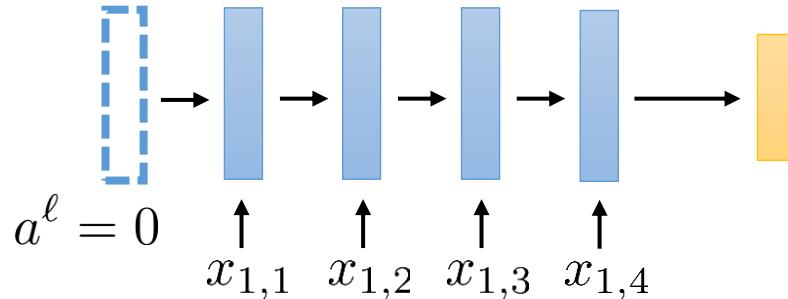
each layer:

$$\bar{a}^{\ell-1} = \begin{bmatrix} a^{\ell-1} \\ x_{i,t} \end{bmatrix} \quad z^\ell = W^\ell \bar{a}^{\ell-1} + b^\ell \quad a^\ell = \sigma(z^\ell)$$

Note: this doesn't actually work very well in practice, we'll discuss this more later

Obvious question: what happens to the missing layers?

Variable layer count?



This is more efficient than always 0-padding the sequence up to max length

Each layer is much smaller than the giant first layer we would need if we feed in the whole sequence at the first layer

The shorter the sequence, the fewer layers we have to evaluate

But the total number of weight matrices increases with max sequence length!

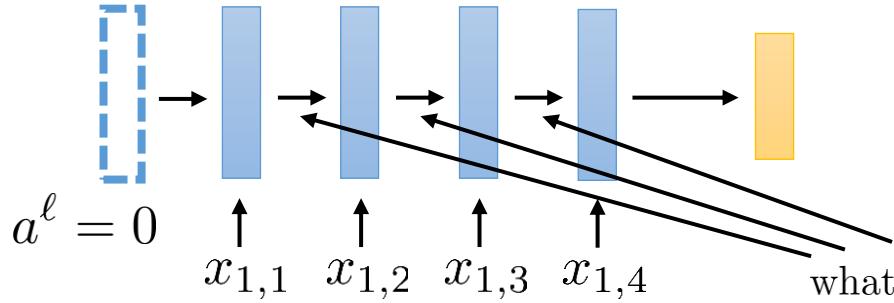
each layer:

$$\bar{a}^{\ell-1} = \begin{bmatrix} a^{\ell-1} \\ x_{i,t} \end{bmatrix}$$

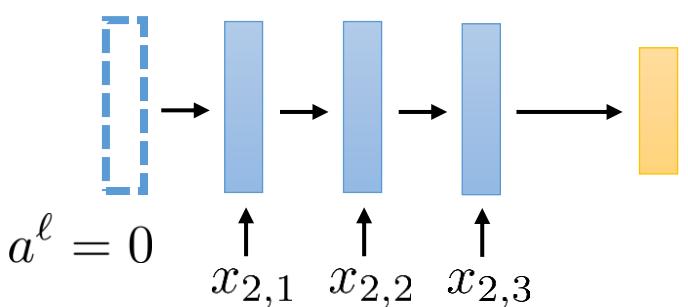
$$z^\ell = W^\ell \bar{a}^{\ell-1} + b^\ell$$

$$a^\ell = \sigma(z^\ell)$$

Can we share weight matrices?



what if W^ℓ is *the same* for all these layers?



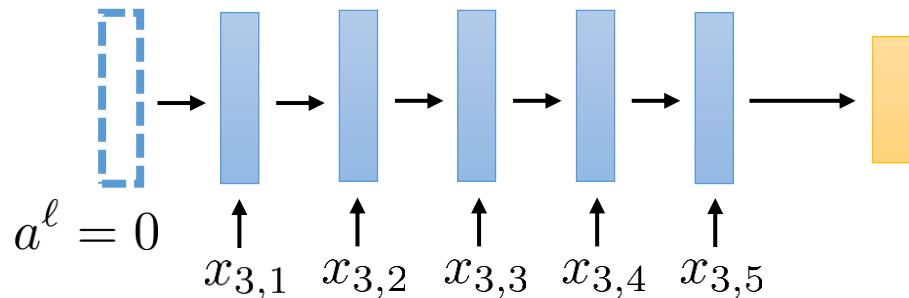
i.e., $W^{\ell_i} = W^{\ell_j}$ for all i, j

$b^{\ell_i} = b^{\ell_j}$ for all i, j

we can have as many “layers” as we want!

this is called a **recurrent** neural network (RNN)

could also call this a “variable-depth”
network perhaps?



each layer:

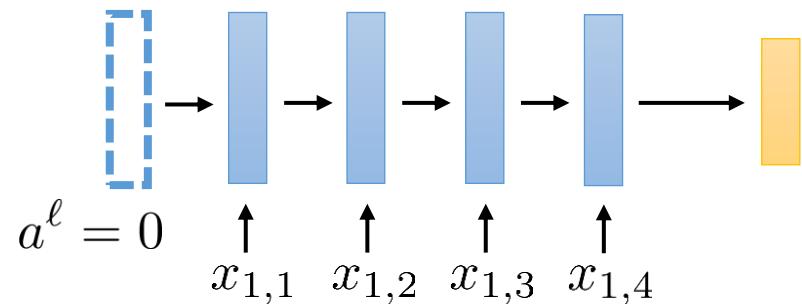
$$\bar{a}^{\ell-1} = \begin{bmatrix} a^{\ell-1} \\ x_{i,t} \end{bmatrix}$$

$$z^\ell = W^\ell \bar{a}^{\ell-1} + b^\ell$$

$$a^\ell = \sigma(z^\ell)$$

Aside: RNNs and time

What we just learned:

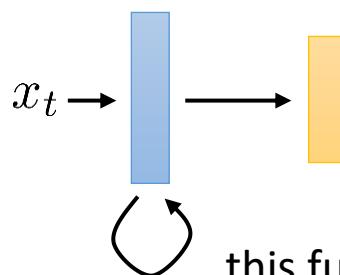


“a recurrent neural network extends a standard neural network along the time dimension”

(or some other assertion of this sort)

This is technically true, but somewhat unhelpful for actually understanding how RNNs work, and makes them seem more mystical than they are

What you often see in textbooks/classes:



RNNs are just neural networks that share weights across multiple layers, take an input at each layer, and have a variable number of layers

this funny thing represents the fact that this layer also gets its own “previous” value as input

How do we train this?

Backpropagation:

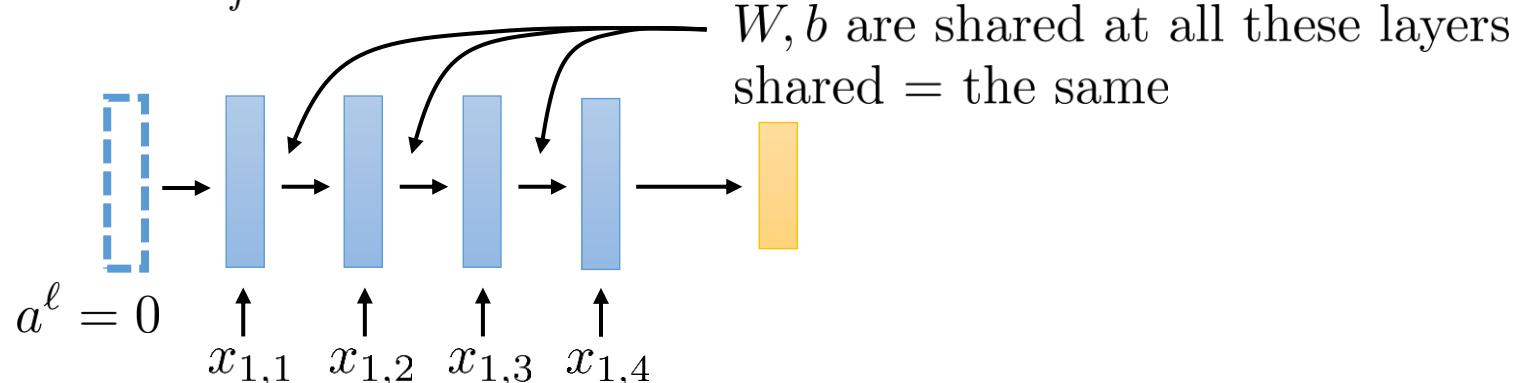
forward pass: calculate each $a^{(i)}$ and $z^{(i)}$

backward pass:

$$\text{initialize } \delta = \frac{d\mathcal{L}}{dz^{(n)}}$$

for each f with input x_f & params θ_f from end to start:

$$\begin{aligned} \frac{d\mathcal{L}}{d\theta_f} &\leftarrow \frac{df}{d\theta_f} \delta && \text{taken literally, gradient at } \ell - 1 \text{ will "overwrite" gradient at } \ell \\ \delta &\leftarrow \frac{df}{dx_f} \delta && \frac{d\mathcal{L}}{d\theta_f} += \frac{df}{d\theta_f} \delta && \text{"accumulate" the gradient during the backward pass} \end{aligned}$$



To convince yourself that this is true:

$$f(x) = g(x, h(x)) \quad \text{how does this resemble role of } W \text{ in the RNN?}$$

$$\frac{d}{dx} f(x) = \frac{dg}{dx} + \frac{dh}{dx} \frac{dg}{dh}$$

↑
derivative through first argument

derivative through second argument
(via chain rule)

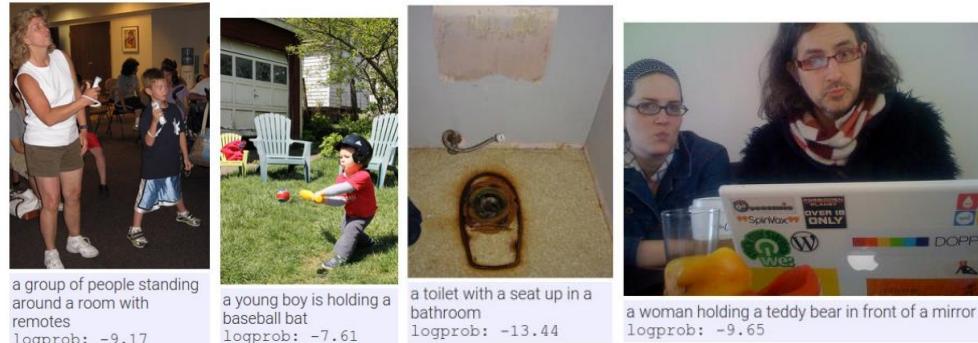
What if we have variable-size outputs?

Examples:

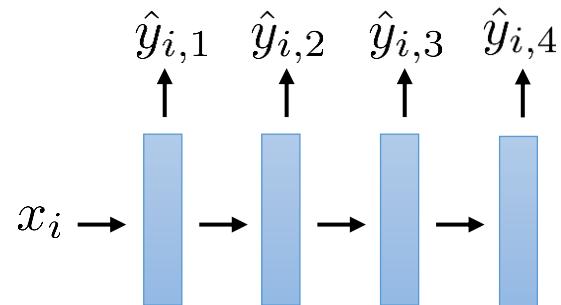
generating a text caption for an image

predicting a sequence of future video frames

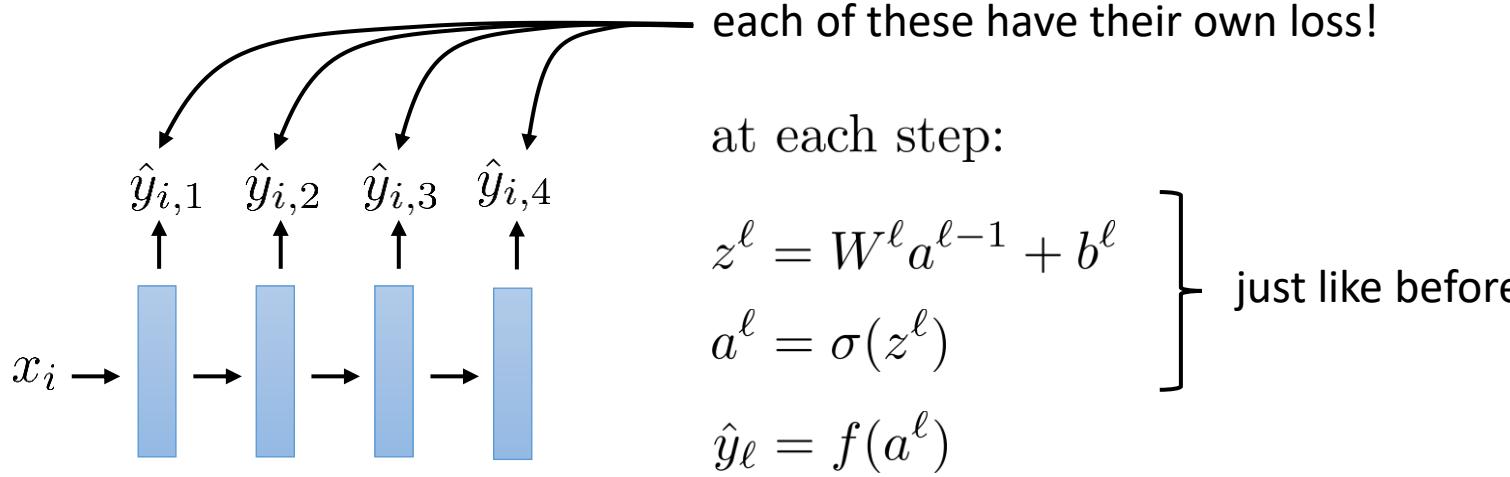
generating an audio sequence



frames with yellow labels are predictions



An output at every layer

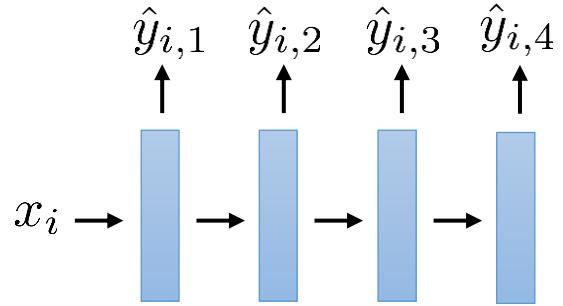


we have a loss on *each* \hat{y}_ℓ
(e.g., cross-entropy)

some kind of readout function
“decoder”
could be as simple as a linear layer + softmax

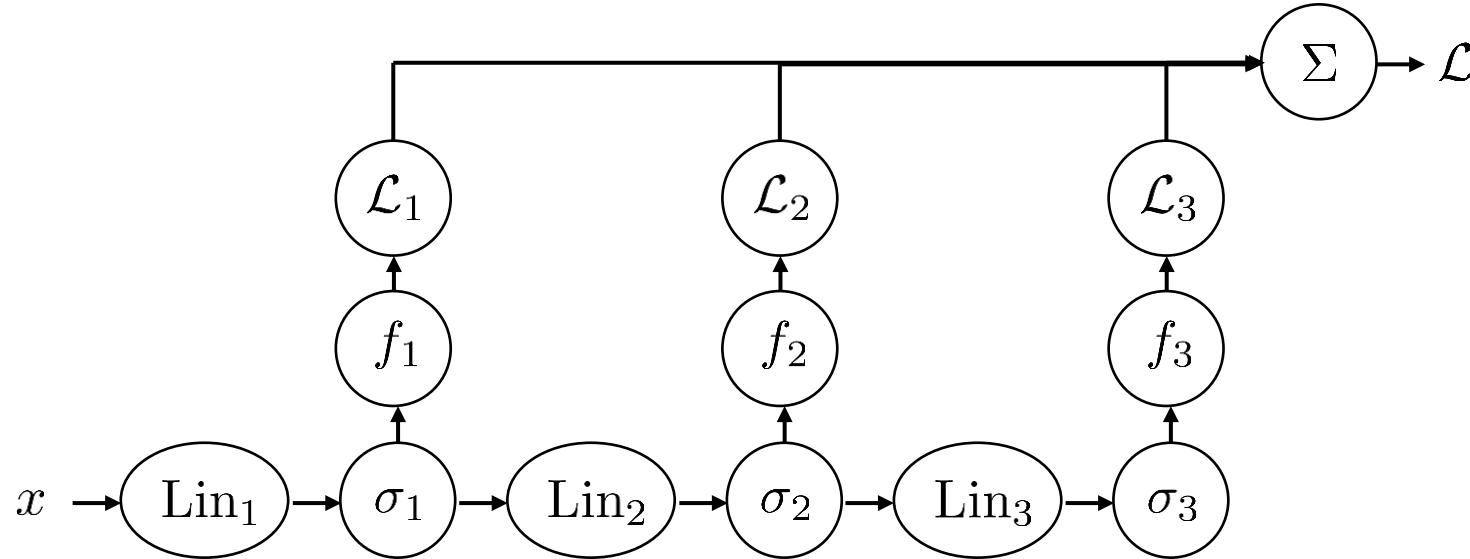
$$\mathcal{L}(\hat{y}_{1:T}) = \sum_\ell \mathcal{L}_\ell(\hat{y}_\ell)$$

Let's draw the computation graph!



$$\begin{aligned} z^\ell &= W^\ell a^{\ell-1} + b^\ell \\ a^\ell &= \sigma(z^\ell) \\ \hat{y}_\ell &= f(a^\ell) \end{aligned}$$

$$\mathcal{L}(\hat{y}_{1:T}) = \sum_\ell \mathcal{L}_\ell(\hat{y}_\ell)$$

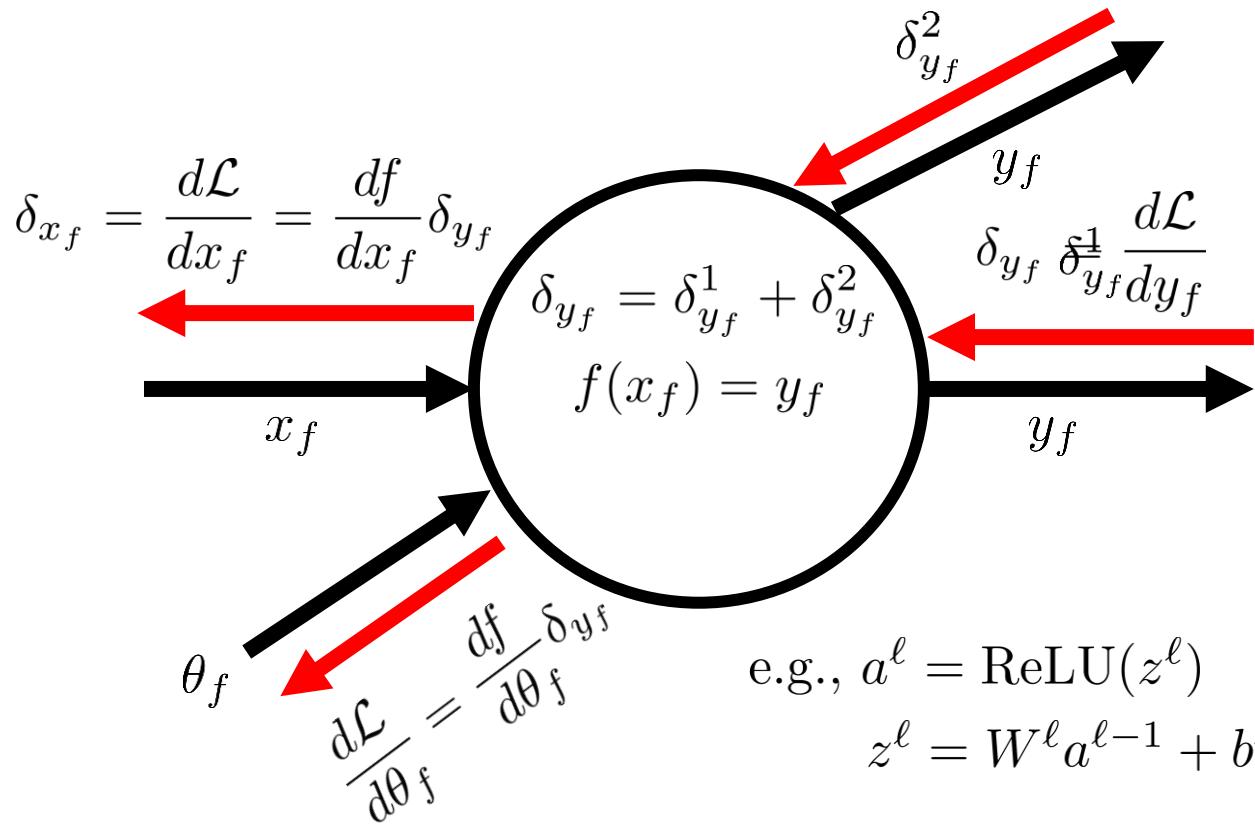


not completely obvious how to do backprop on this!

Graph-structured backpropagation

Also called reverse-mode automatic differentiation

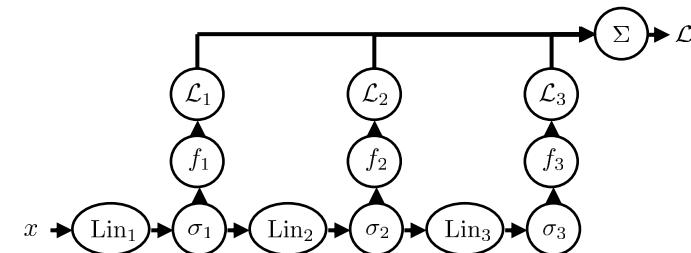
do the following at each layer $f(x_f) \rightarrow y_f$
starting with the last function, where $\delta = 1$



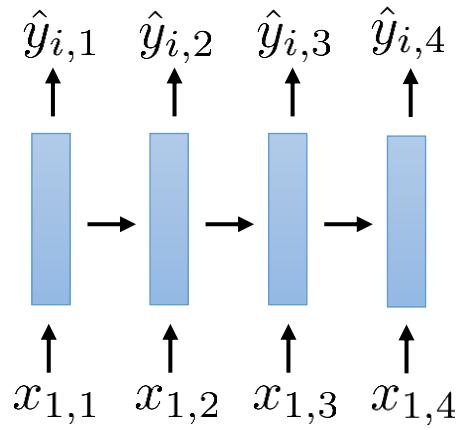
Very simple rule:

For each node with multiple descendants
in the computational graph:

Simply add up the delta vectors coming
from all of the descendants



Inputs and outputs at each step?



at each step:

$$\left. \begin{aligned} \bar{a}^{\ell-1} &= \begin{bmatrix} a^{\ell-1} \\ x_{i,t} \end{bmatrix} \\ z^\ell &= W^\ell \bar{a}^{\ell-1} + b^\ell \\ a^\ell &= \sigma(z^\ell) \end{aligned} \right\} \text{just like before}$$

$$\hat{y}_\ell = f(a^\ell)$$

Examples:

generating a text caption for an image

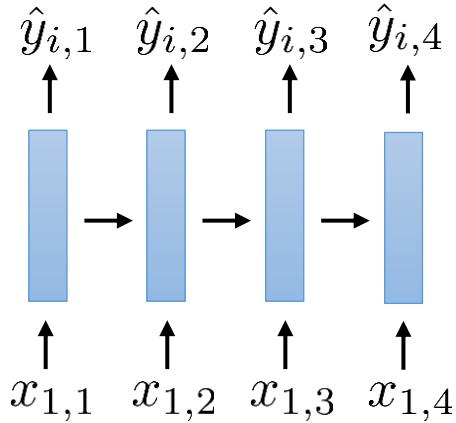
a bit subtle why there are
inputs at each time step!
we'll discuss this later

translating some text into a different language

though there are much
better ways to do it!

What makes RNNs difficult to train?

RNNs are extremely deep networks



imagine our sequence length was 1000+
that's like backpropagating through 1000+ layers!

Intuitively:

vanishing gradients = gradient signal from later steps never reaches the earlier steps

very bad – this prevents the RNN from “remembering” things from the beginning!

“vanishing gradients”

big problem!

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}}$$

If we multiply many many numbers together, what will we get?

If most of the numbers are < 1 , we get 0

$$\frac{d\mathcal{L}}{dW^{(1)}} = J_1 J_2 J_3 \dots J_n \frac{d\mathcal{L}}{dz^{(n)}}$$

If most of the numbers are > 1 , we get infinity

We only get a reasonable answer if the numbers are all close to 1!

“exploding gradients”

could fix with gradient clipping

Promoting better gradient flow

Basic idea: (similar to what we saw before) we would really like the gradients to be close to 1

which gradients?

each layer:

$$\bar{a}_{t-1} = \begin{bmatrix} a_{t-1} \\ x_t \end{bmatrix} \quad z_t = W\bar{a}_{t-1} + b \quad a_t = \sigma(z_t)$$



$$a_t = q(a_{t-1}, x_t) \quad \text{"RNN dynamics"}$$

dynamics Jacobian $\frac{dq}{da_{t-1}} \approx \mathbf{I}$ ← best gradient flow

not always good – only good when we want to **remember**
sometimes we may want to **forget**

Promoting better gradient flow

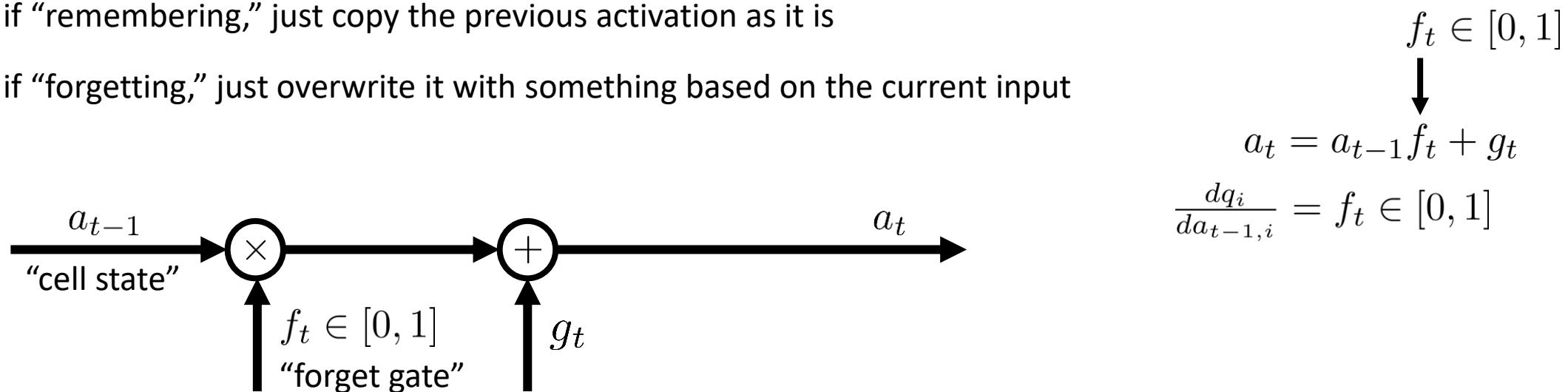
Basic idea: (similar to what we saw before) we would really like the gradients to be close to 1

Intuition: want $\frac{dq_i}{da_{t-1,i}} \approx 1$ if we choose to *remember* $a_{t-1,i}$

for each unit, we have a little “neural circuit” that decides whether to remember or overwrite

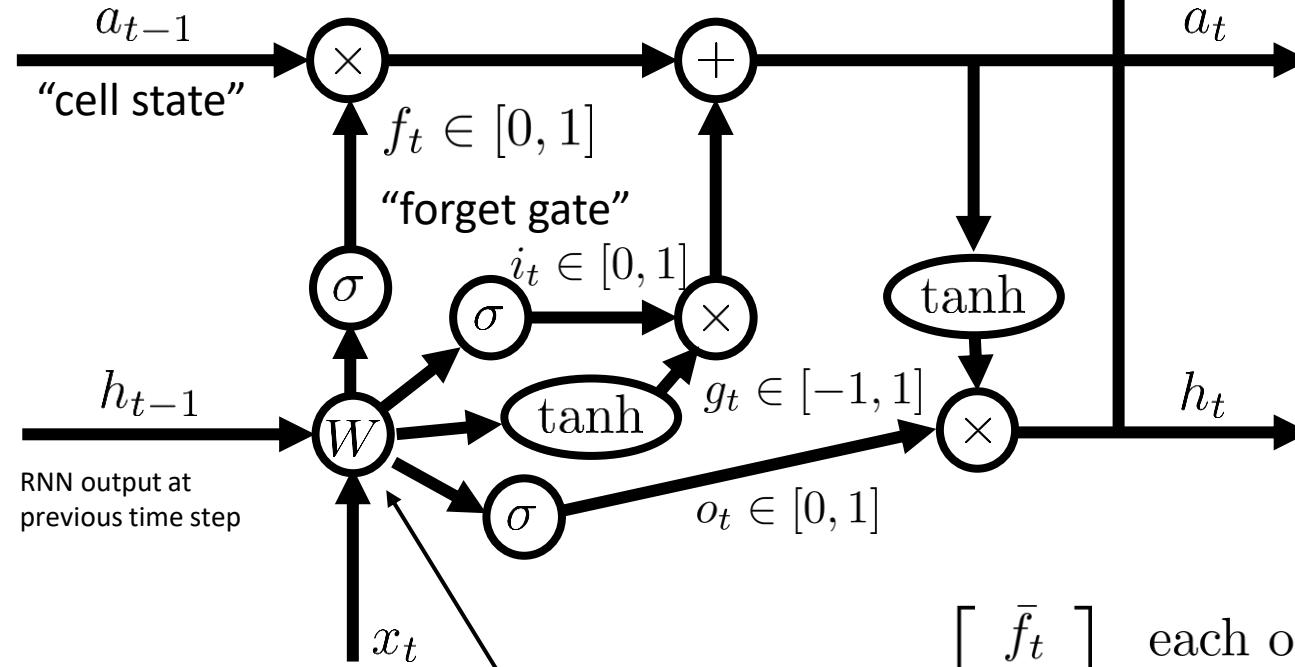
if “remembering,” just copy the previous activation as it is

if “forgetting,” just overwrite it with something based on the current input



LSTM cells

Long short-term memory



output is **4x** larger in dimensionality than RNN cell!

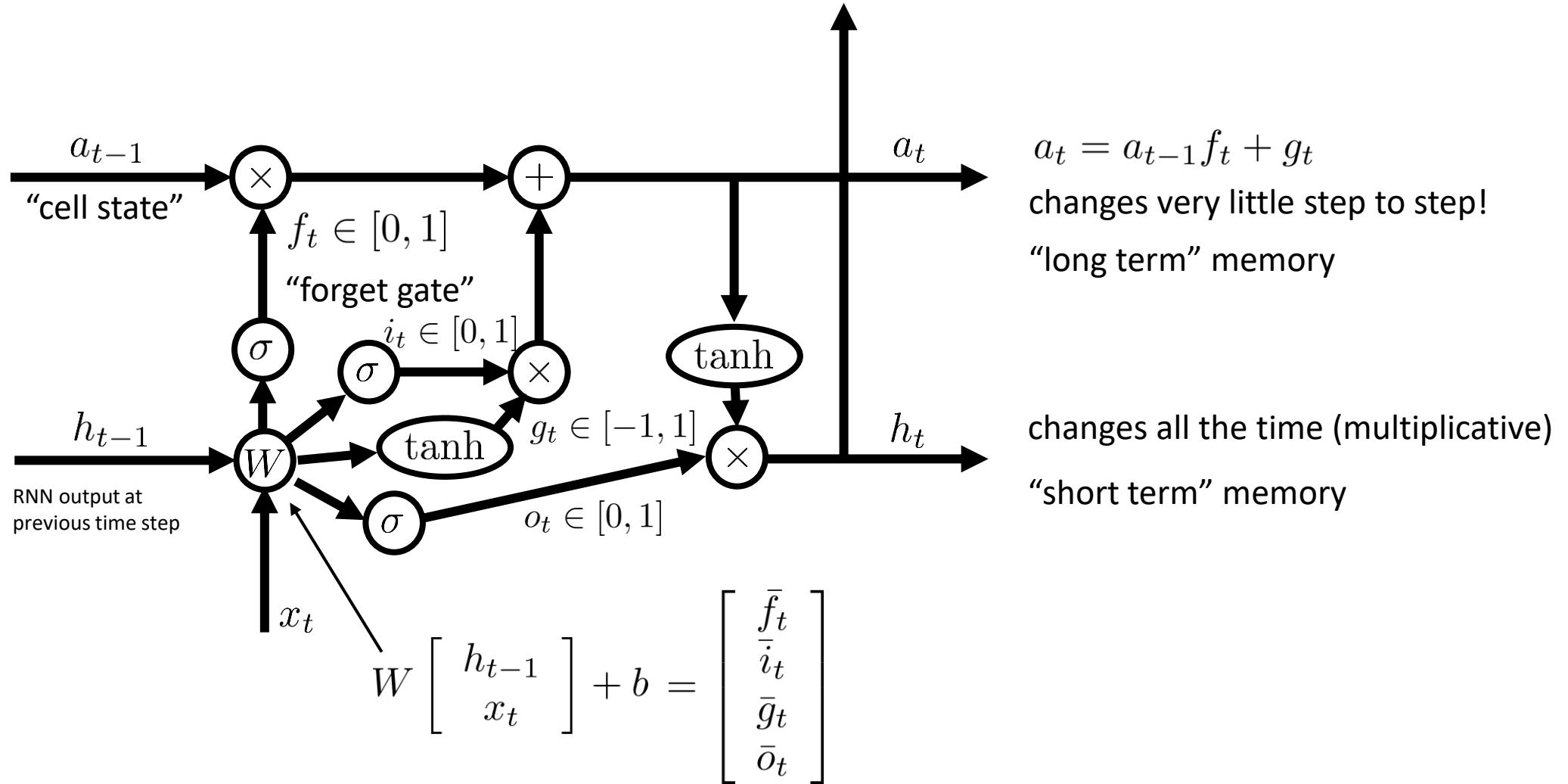
$$W \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b = \begin{bmatrix} \bar{f}_t \\ \bar{i}_t \\ \bar{g}_t \\ \bar{o}_t \end{bmatrix}$$

each of these is a vector with same dims as h_{t-1}

Isn't this all a little arbitrary?

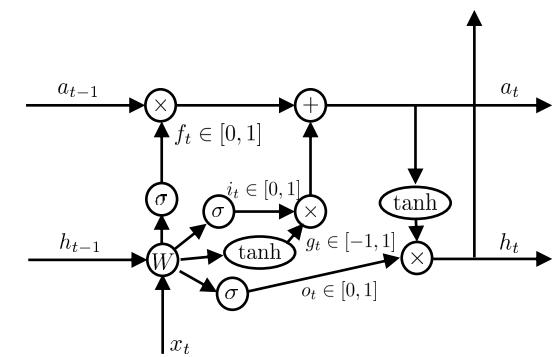
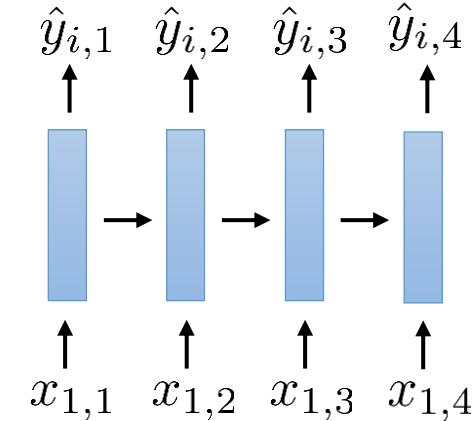
Well, yes, but it ends up working quite well in practice, and much better than a naïve RNN!

Why do LSTMs train better?



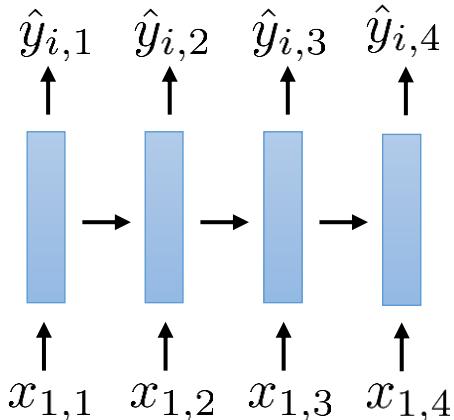
Some practical notes

- In practice, RNNs almost always have both an input and an output at each step (we'll see why in the next section)
- In practice, naïve RNNs like in part 1 almost never work
- LSTM units are OK – they work fine in many cases, and dramatically improve over naïve RNNs
 - Still require way more hyperparameter tuning than standard fully connected or convolutional networks
- Some alternatives (that we'll learn about later) can work better for sequences
 - Temporal convolutions
 - Transformers (temporal attention)
- LSTM cells are annoyingly complicated, but once implemented, they can be used the same as any other type of layer (hurray for abstraction!)
- There are some variants of the LSTM that are a bit simpler and work just as well
 - Gated recurrent unit (GRU)



Using RNNs

Autoregressive models and structured prediction



most RNNs used in practice look like this

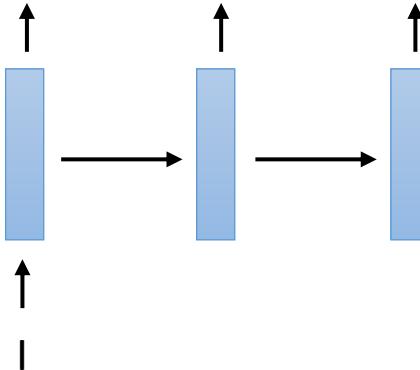
why?

most problems that require multiple outputs have strong **dependencies** between these outputs

this is sometimes referred to as **structured** prediction

Example: text generation

think: 0.3 therefore: 0.3 I: 0.3
like: 0.3 machine: 0.3 learning: 0.3
am: 0.4 not: 0.4 just: 0.4



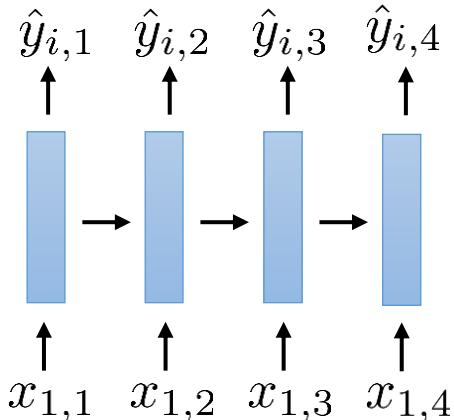
I think therefore I am

I like machine learning

I am not just a neural network

we get a nonsense output
even though the network
had exactly the right
probabilities!

Autoregressive models and structured prediction



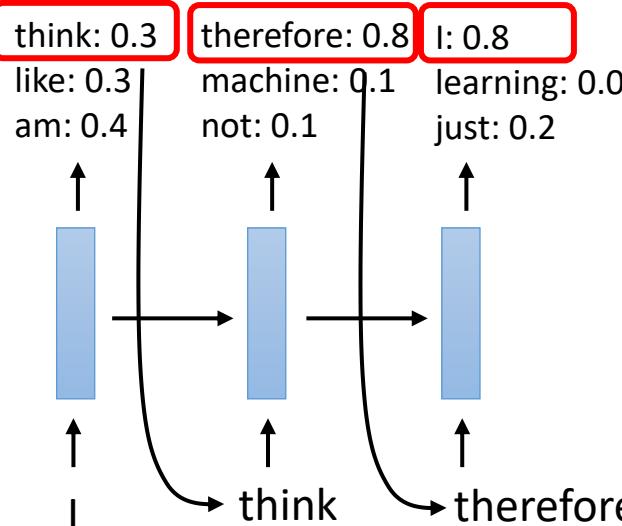
most RNNs used in practice look like this

why?

most problems that require multiple outputs have strong **dependencies** between these outputs

this is sometimes referred to as **structured** prediction

Example: text generation



I think therefore I am

I like machine learning

I am not just a neural network

Key idea: past outputs should influence future outputs!

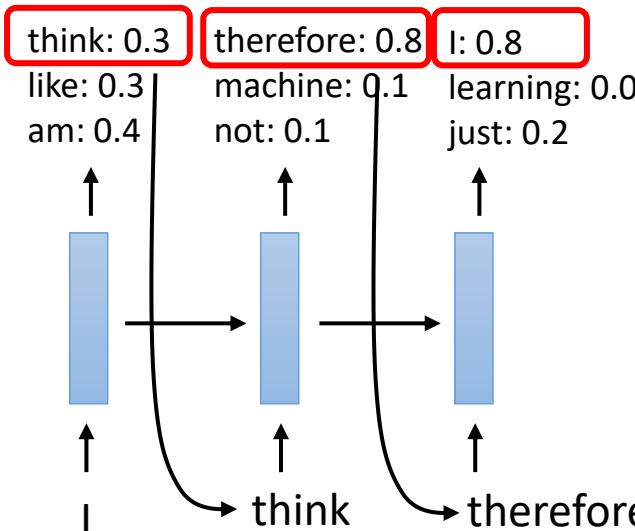
we get a nonsense output even though the network had exactly the right probabilities!

Autoregressive models and structured prediction

How do we train it?

Basic version: just set inputs to be entire training sequences, and ground truth outputs to be those same sequences (offset by one step)

Example: text generation



$$x_{1:5} = ("I", "think", "therefore", "I", "am")$$

$$y_{1:5} = ("think", "therefore", "I", "am", \text{stop_token})$$

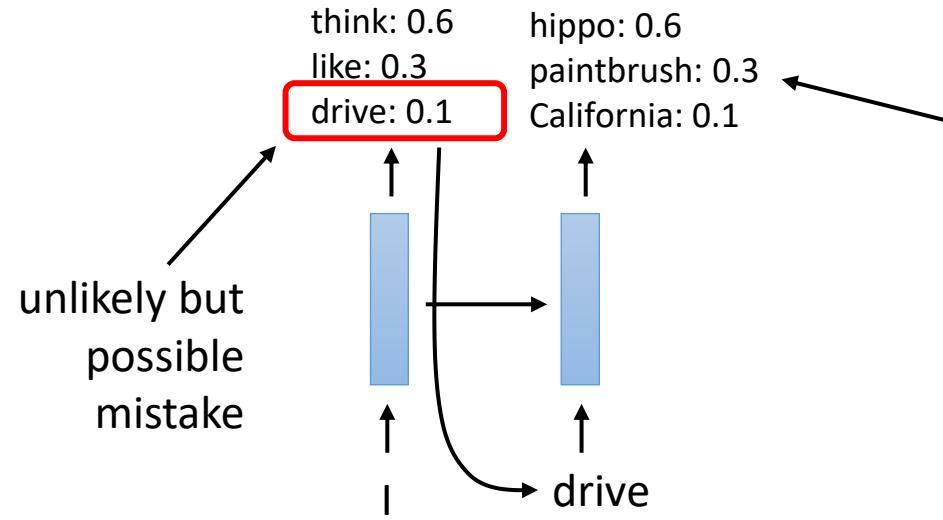
This teaches the network to output "am" if it sees "I think therefore I"

I think therefore I am

I like machine learning

I am not just a neural network

Aside: distributional shift



complete nonsense,
because the network never
saw inputs remotely like this

The problem: this is a training/test discrepancy:
the network always saw **true** sequences as
inputs, but at test-time it gets as input its own
(potentially incorrect) predictions

we got unlucky, but now the
model is completely confused
it never saw “I drive” before

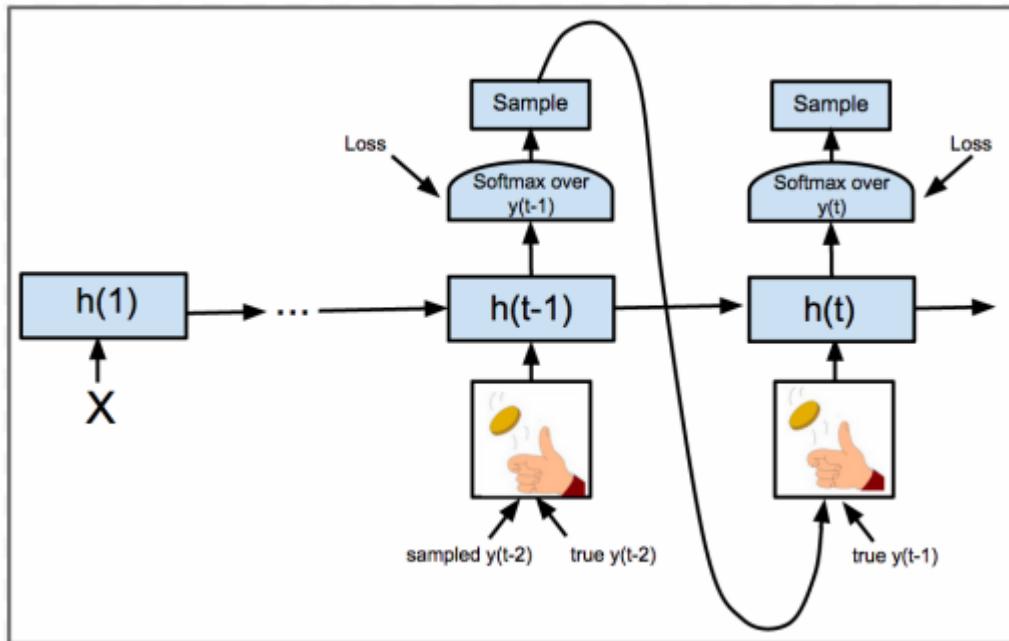
This is called **distributional shift**, because the
input distribution **shifts** from true strings (at
training) to synthetic strings (at test time)

Even **one** random mistake can completely
scramble the output!

Aside: scheduled sampling

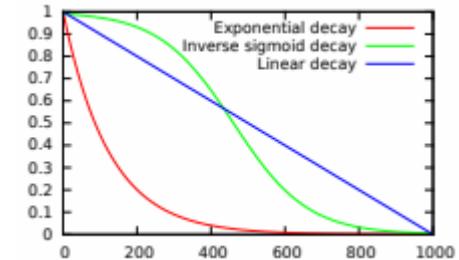
An old trick from reinforcement learning adapted to training RNNs

schedules for probability of using ground truth input token



At the beginning of training, mostly feed in ground truth tokens as input, since model predictions are mostly nonsense

At the end of training, mostly feed in the model's own predictions, to mitigate distribution shift

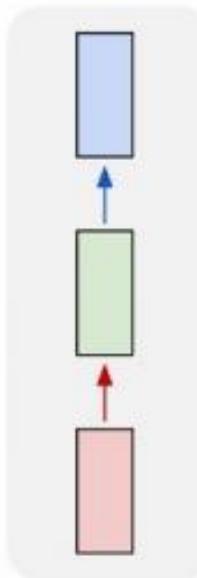


Randomly decide whether to give the network a ground truth token as input during training, or its own previous prediction

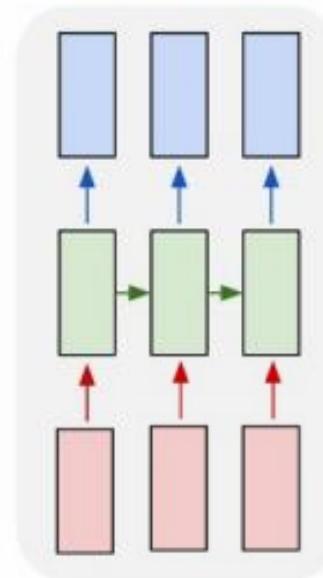
Different ways to use RNNs

in reality, we almost always use autoregressive generation like this

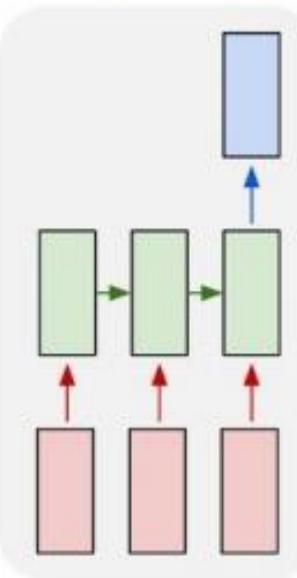
one to one



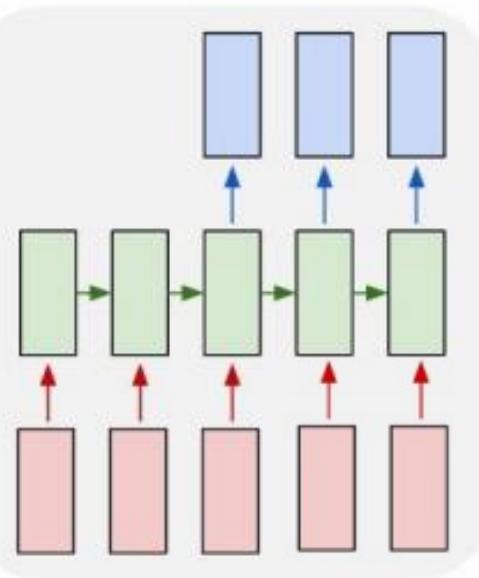
one to many



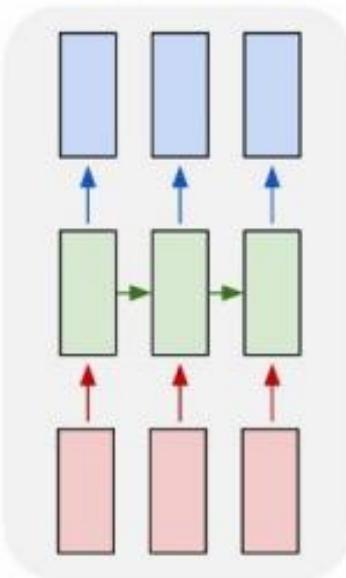
many to one



many to many



many to many



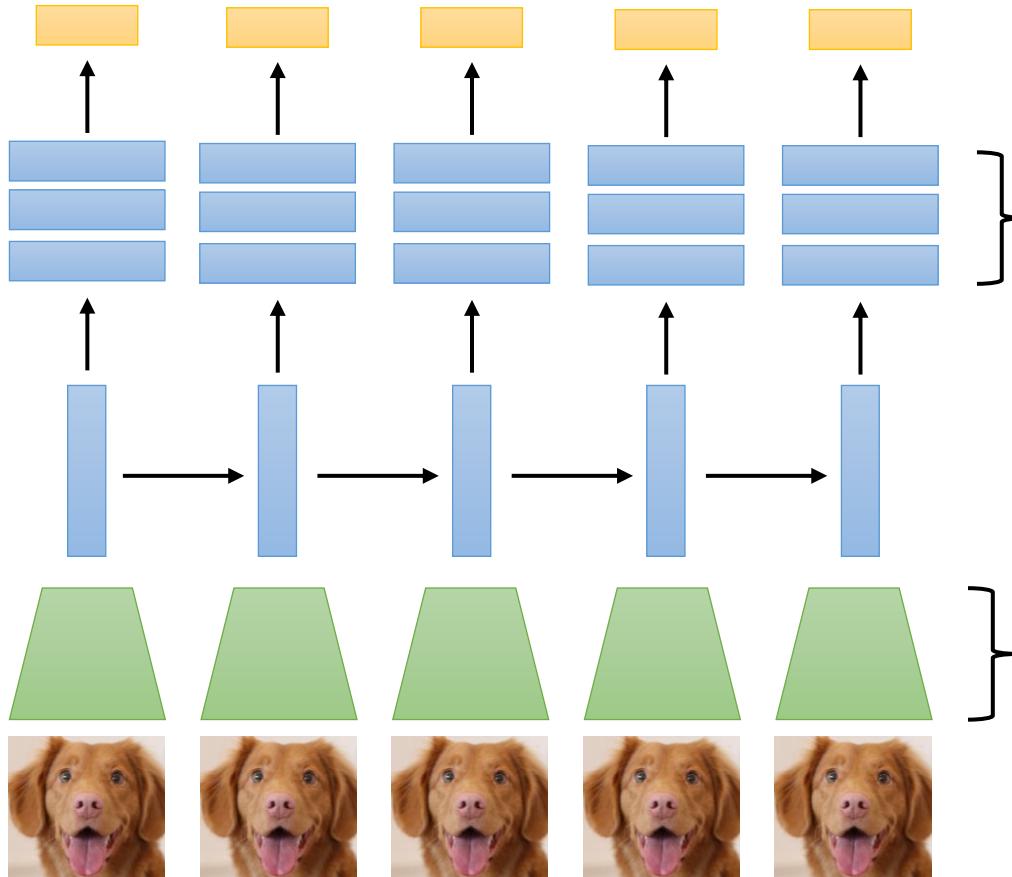
e.g., activity recognition

e.g., image captioning

e.g., frame-level video annotation

e.g., machine translation

RNN encoders and decoders



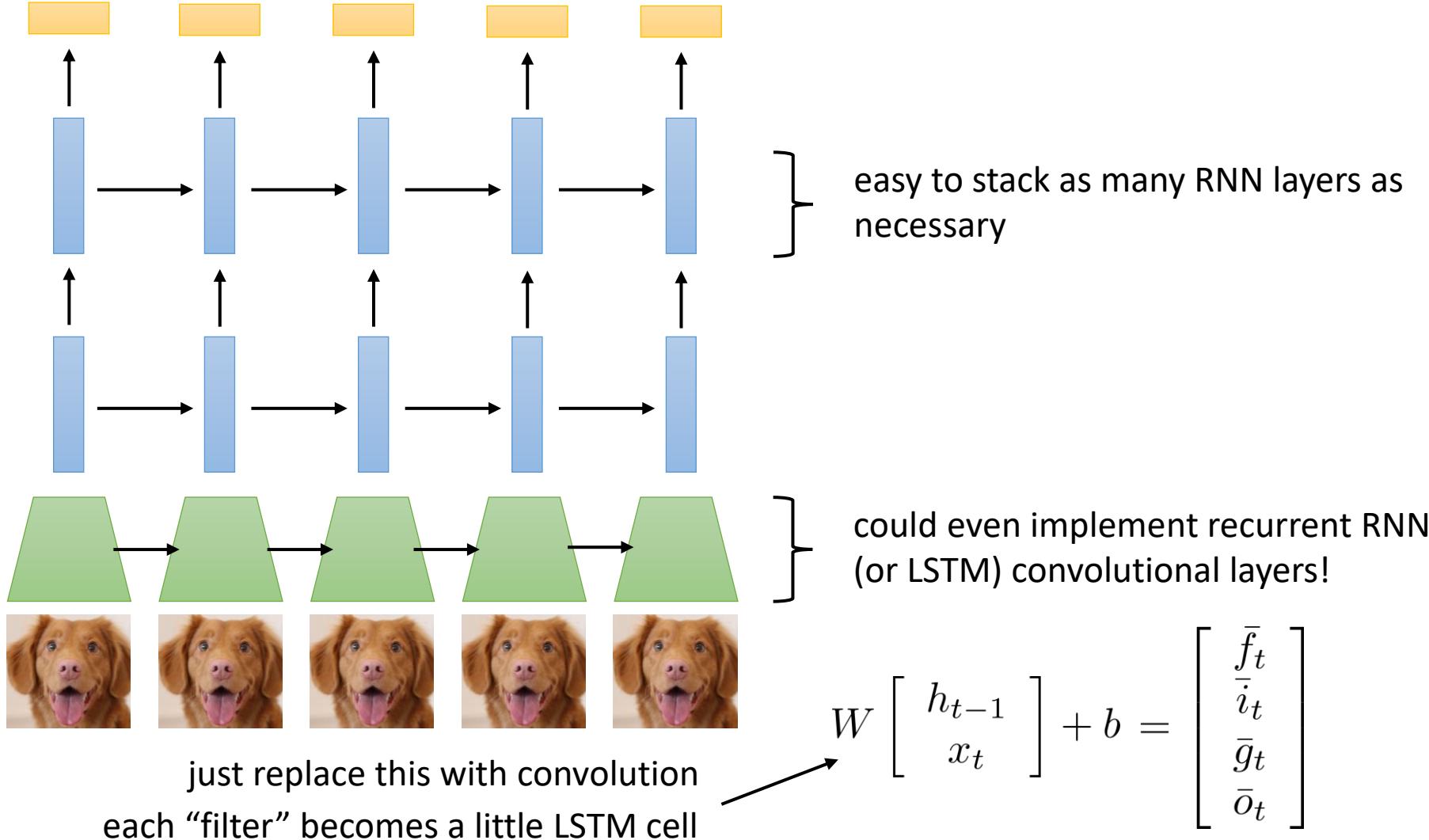
output is processed by a (non-recurrent)
decoder before getting outputted

a bit less common, since we could just
use more RNN layers

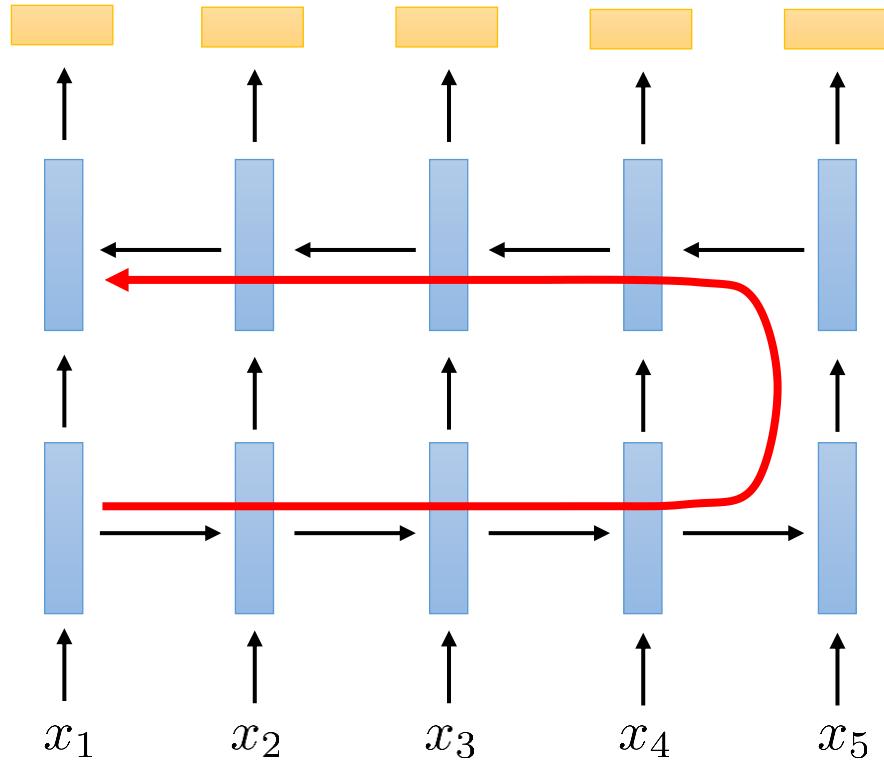
input is processed by a (non-recurrent)
encoder before going into the RNN

very common with image inputs

RNNs with many layers



Bidirectional models



Example: speech recognition

Problem: the word at a particular time step might be hard to guess without looking at the rest of the utterance!

(for example, can't tell if a word is finished until hearing the ending)

This is an even bigger problem in machine translation, but there we use slightly different types of models

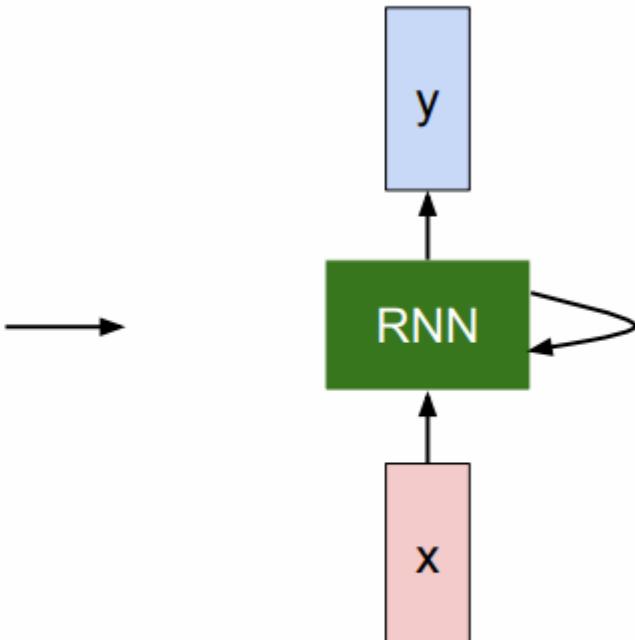
Some (vivid) examples

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 deserved 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.



Some (vivid) examples

at first:

tyntd-iafhatawiaoahrdemot lytdws e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e
plia tkldrgd t o idoe ns,smtt h ne etie h,hregtrs nigtike,aoaennts 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.

Some (vivid) examples

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.

Some (vivid) examples

The Stacks Project: open source algebraic geometry textbook

The Screenshot shows the homepage of The Stacks Project. The main content area is titled "Browse chapters" and lists chapters under the "Preliminaries" part. Each chapter entry includes links for "online", "TeX source", and "view pdf". To the right, a sidebar titled "Parts" lists numbered sections: 1. Preliminaries, 2. Schemes, 3. Topics in Scheme Theory, 4. Algebraic Spaces, 5. Topics in Geometry, 6. Deformation Theory, 7. Algebraic Stacks, and 8. Miscellany. Below this is a "Statistics" section with the following text and bullet points:

The Stacks project now consists of

- o 455910 lines of code
- o 14221 tags (56 inactive tags)
- o 2366 sections

Latex source

<http://stacks.math.columbia.edu/>
The stacks project is licensed under the [GNFL Free Documentation License](#).

Some (vivid) examples

For $\bigoplus_{n=1,\dots,m} \mathcal{L}_{m,n}$ where $\mathcal{L}_{m,n} = 0$, 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 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

$$\widetilde{M}^\bullet = \mathcal{T}^\bullet \otimes_{\text{Spec}(k)} \mathcal{O}_{S,s} - i_X^{-1}\mathcal{F}$$

is a unique morphism of algebraic stacks. Note that

$$\text{Arrows} = (\text{Sch}/S)_{fppf}^{\text{opp}}, (\text{Sch}/S)_{fppf}$$

and

$$V = \Gamma(S, \mathcal{O}) \longrightarrow (U, \text{Spec}(A))$$

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 .

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 $\text{Proj}_X(\mathcal{A}) = \text{Spec}(B)$ 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 restoocomposes of this implies that $\mathcal{F}_{x_0} = \mathcal{F}_{x_0} = \mathcal{F}_{X,\dots,0}$.

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 $\mathcal{J}_{n,0} \circ \overline{A}_2$ 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

Some (vivid) examples

OpenAI GPT-2 generated text

[source](#)

Input: In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

Output: The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.

Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

Basically the same principle, but uses a different type of model that we'll learn about later

Summary

- Recurrent neural networks (RNNs): neural networks that can process variable-length inputs and outputs
 - Could think of them as networks with an input & output at each layer
 - Variable depth
 - Depth = time
- Training RNNs is very hard
 - Vanishing and exploding gradients
 - Can use special cells (LSTM, GRU)
 - Generally need to spend more time tuning hyperparameters
- In practice, we almost always have both inputs **and** outputs at each step
 - This is because we usually want structured prediction
 - Can use scheduled sampling to handle distributional shift
- Many variants for various purposes
 - Sequence to sequence models (more on this later)
 - Bidirectional models
 - Can even “RNN-ify” convolutional layers!

Sequence to Sequence Models

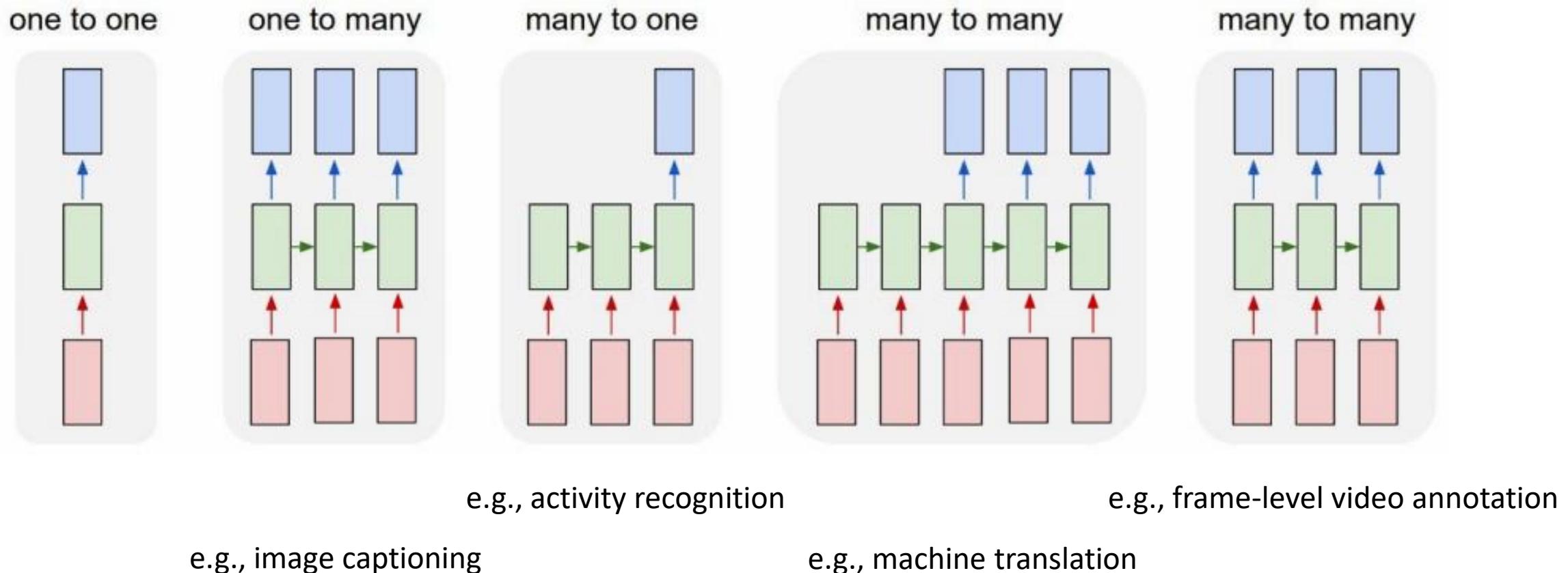
Designing, Visualizing and Understanding Deep Neural Networks

CS W182/282A

Instructor: Sergey Levine
UC Berkeley



Last time: RNNs and LSTMs



A basic neural language model

training data: natural sentences

I think therefore I am

I like machine learning

I am not just a neural network

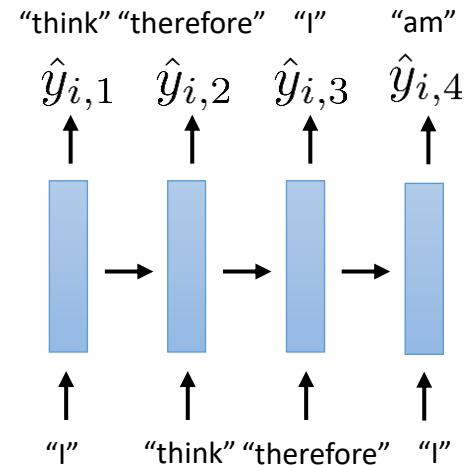
in reality there could be several million of these

how are these represented?

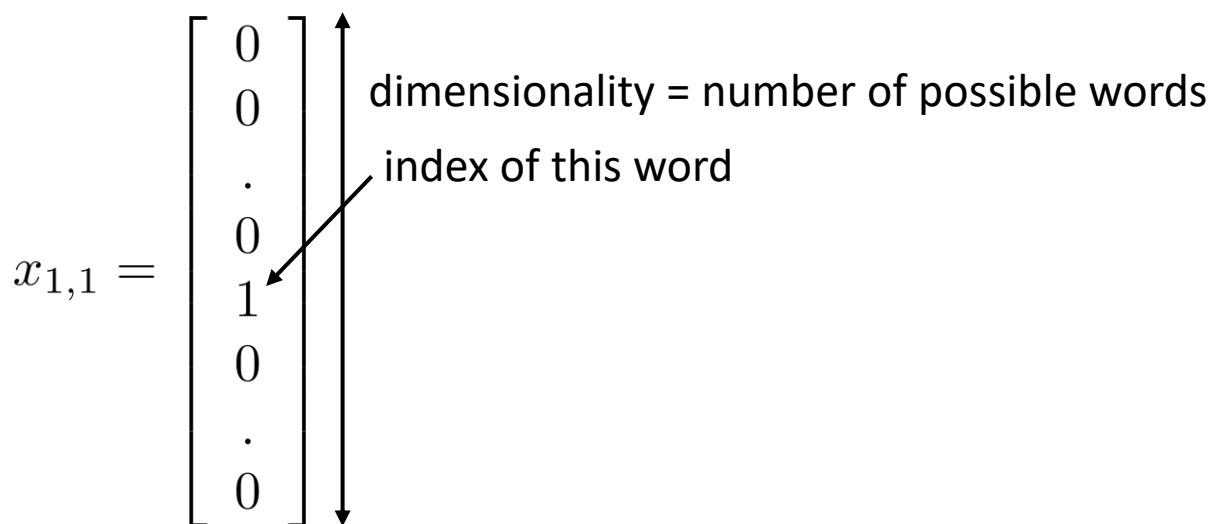
tokenize the sentence (each word is a token)

simplest: one-hot vector

more complex: word embeddings (we'll cover this later)

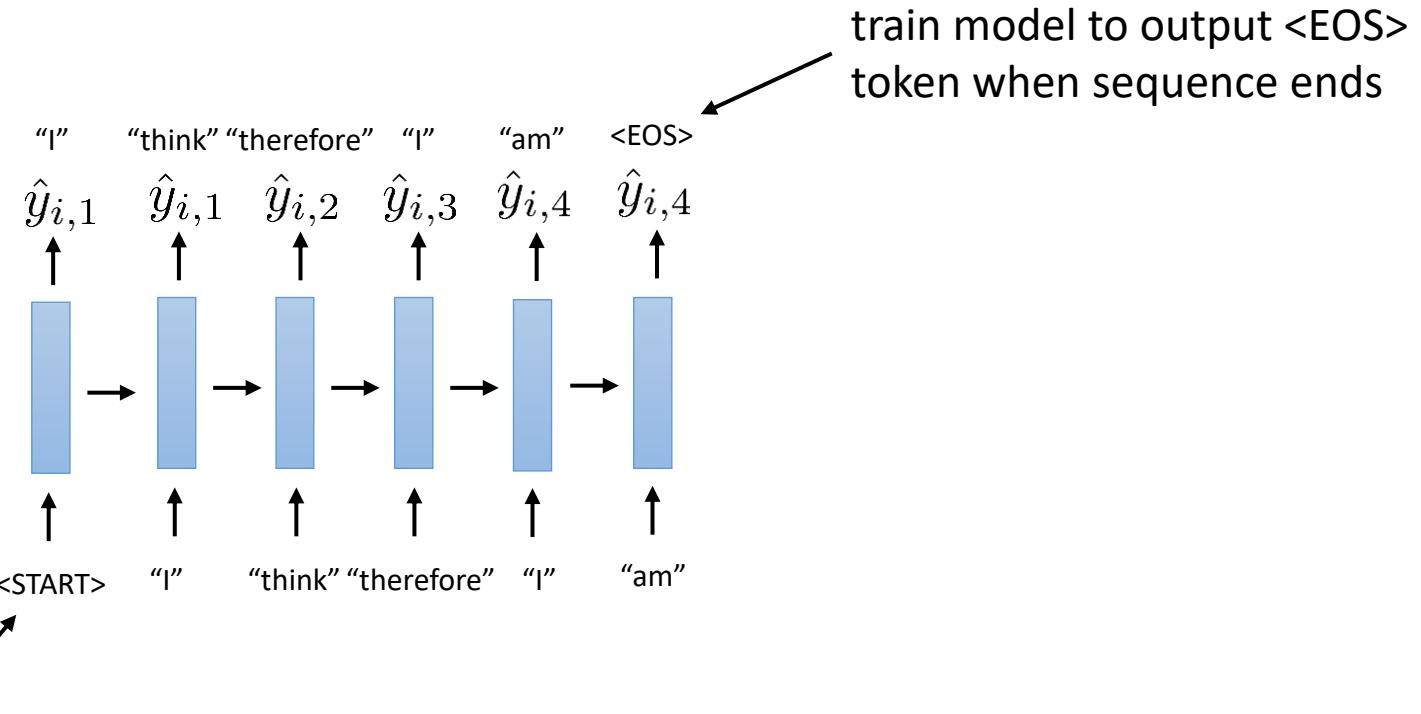


We'll talk about real
language models
much more later



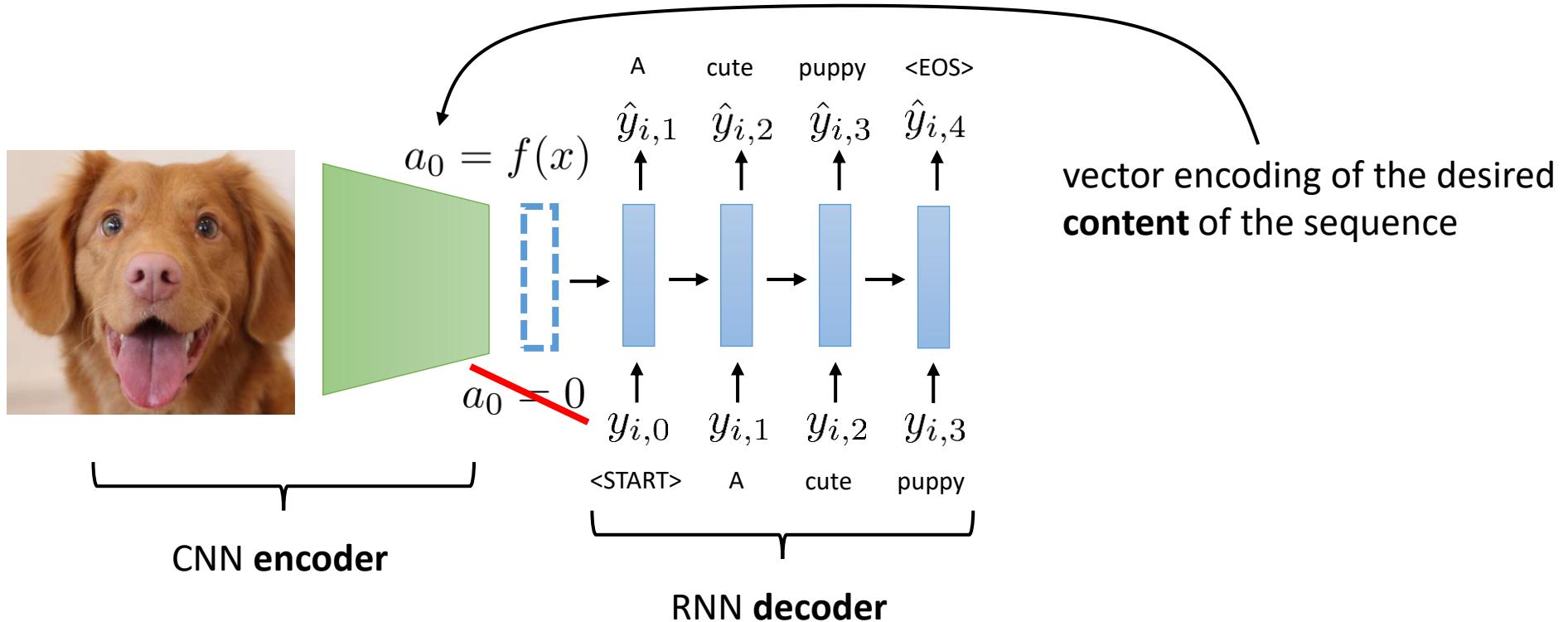
A few details

Question: how do we use such a model to **complete** a sequence (e.g., give it “I think...” and have it finish it?)



If we want to come up with an entirely new sequence, start with a special <START> token

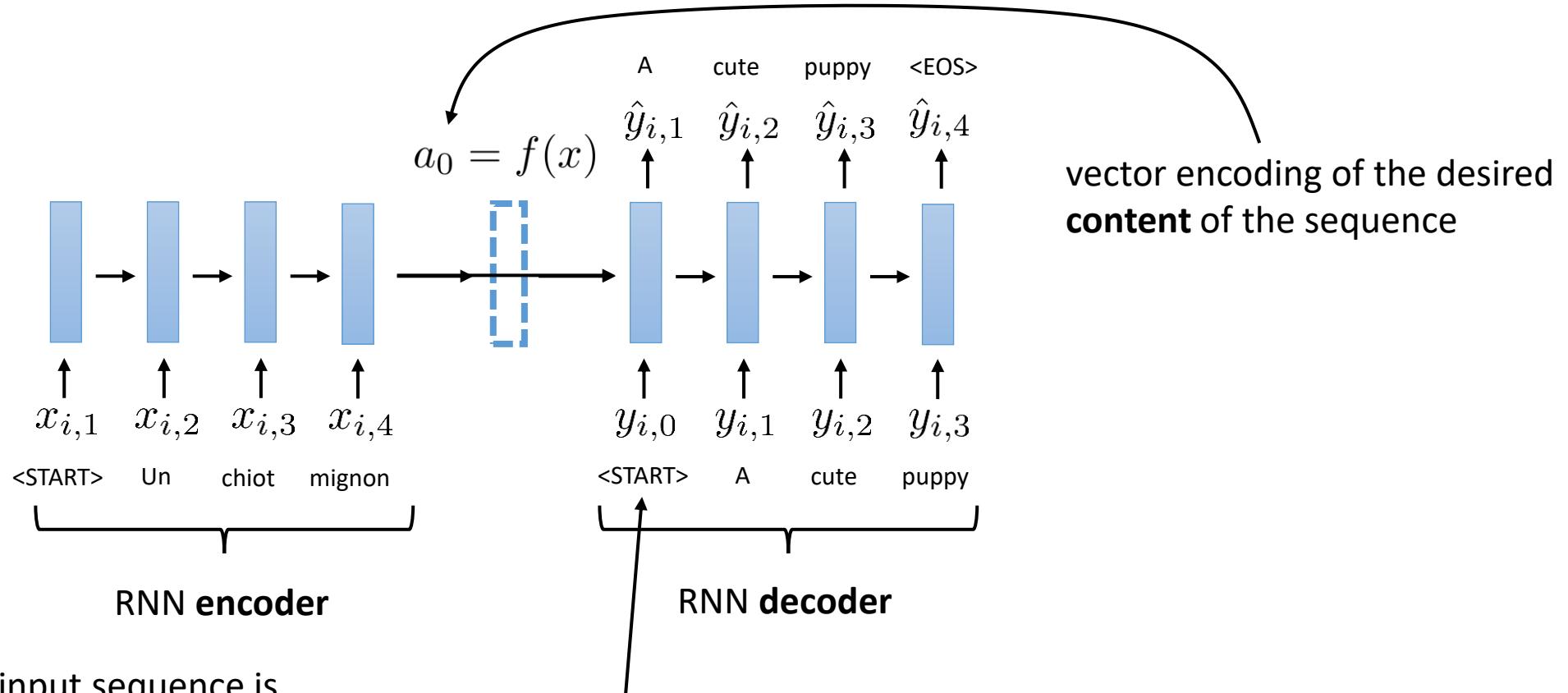
A conditional language model



What do we expect the training data to look like?

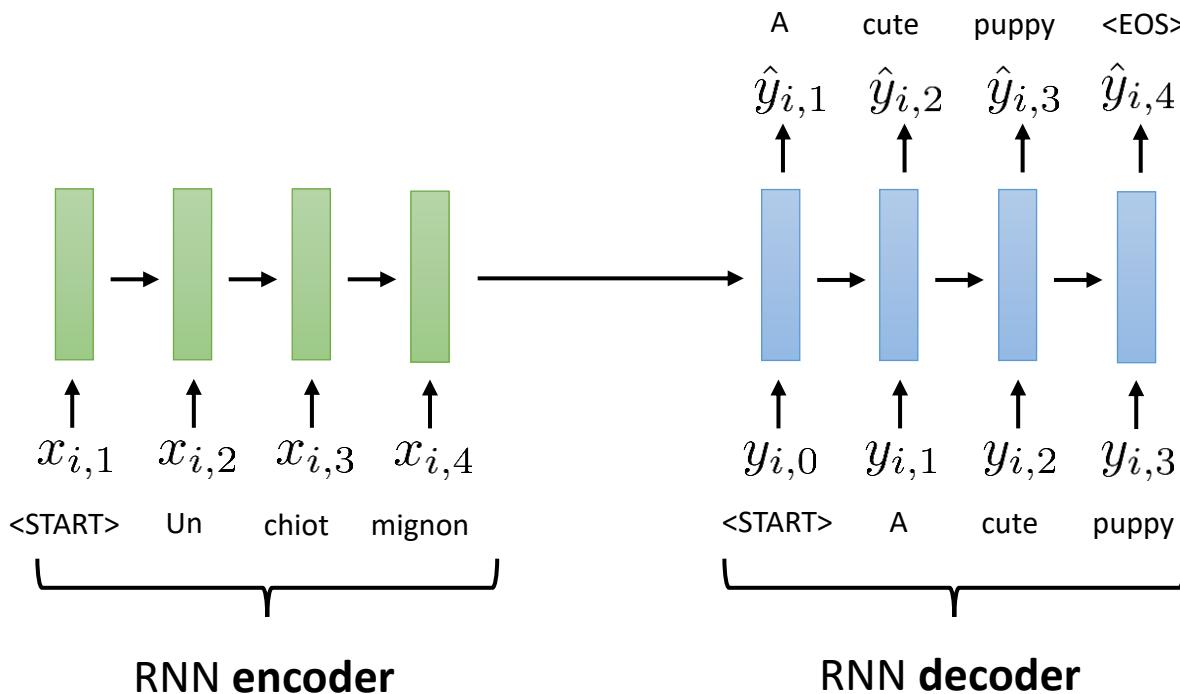
How do we tell the RNN **what** to generate?

What if we condition on *another* sequence?



why?

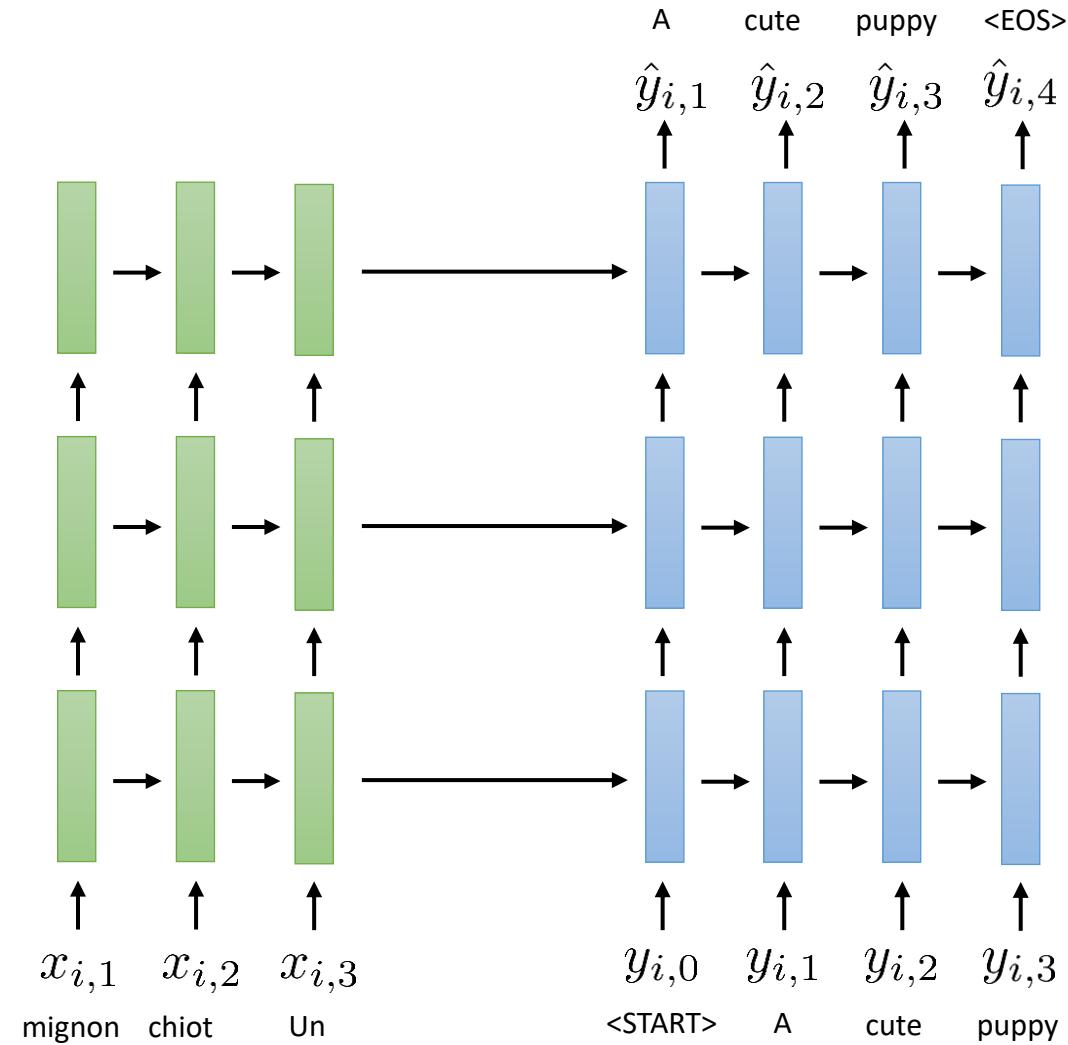
Sequence to sequence models



typically two **separate** RNNs (with different weights)

trained **end-to-end** on paired data (e.g., pairs of French & English sentences)

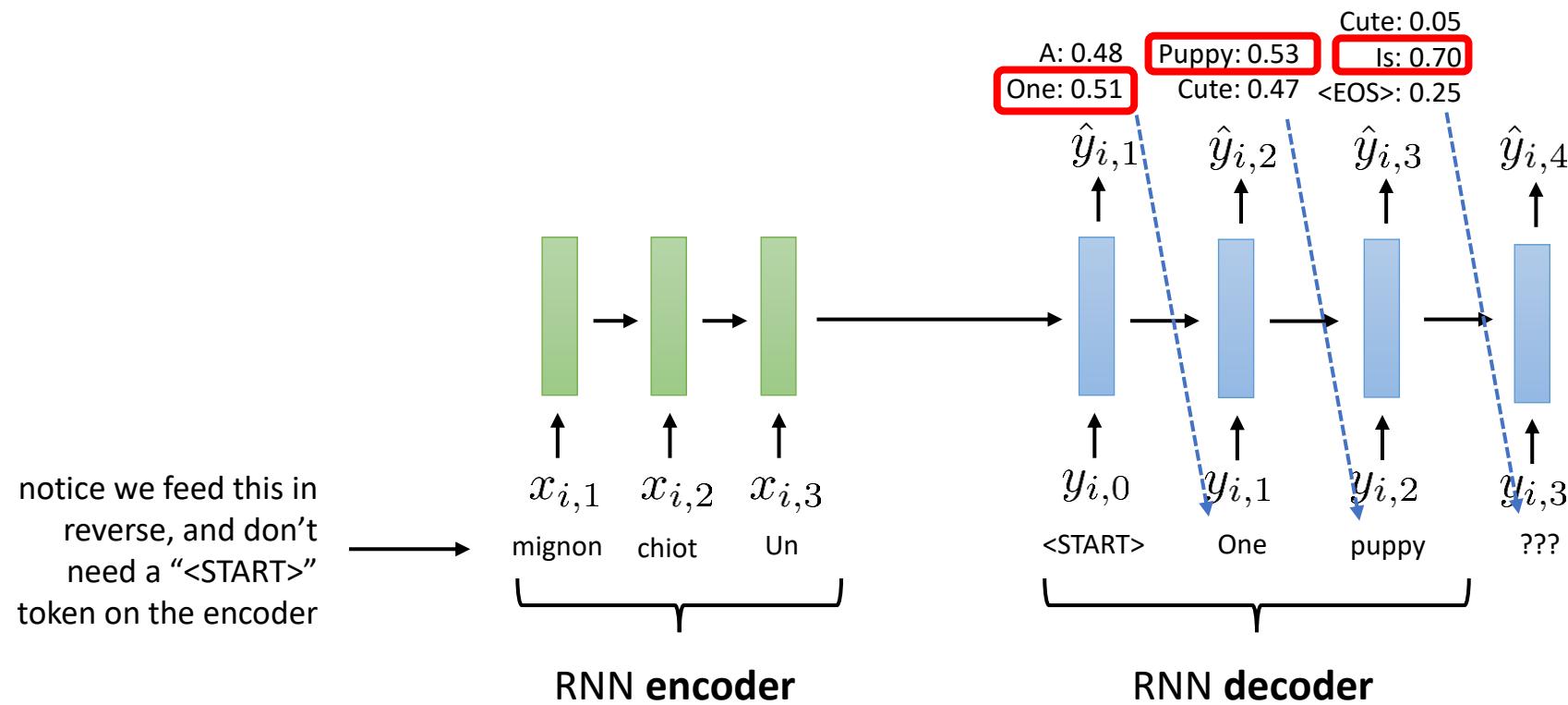
A more realistic example



- Multiple RNN layers
 - Each RNN layer uses LSTM cells (or GRU)
 - Trained end-to-end on pairs of sequences
 - Sequences can be different lengths
- Not just for cute puppies!
- Translate **one language** into **another language**
 - Summarize a **long sentence** into a **short sentence**
 - Respond to a **question** with an **answer**
 - Code generation? **text to Python code**
- For more, see: Ilya Sutskever, Oriol Vinyals, Quoc V. Le.
Sequence to Sequence Learning with Neural Networks. 2014.

Decoding with beam search

Decoding the most likely sequence



What we *should* have done

what does each output represent?

$$p(y_{i,t} | x_{i,1:T}, y_{i,0:t-1})$$

depends on whole
input sequence

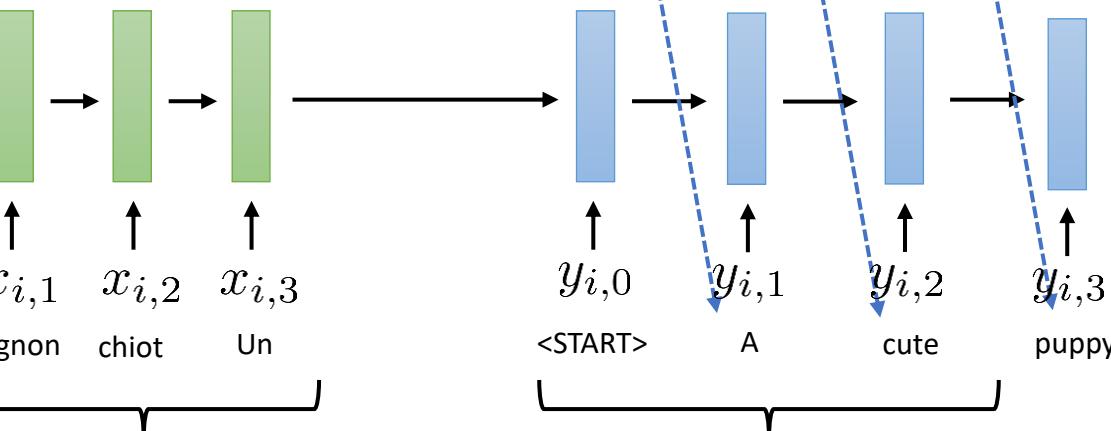
and output
sequence so far!

notice we feed this in
reverse, and don't
need a "<START>"
token on the encoder

slightly lower
probability here

much higher
probability here

A: 0.48	Puppy: 0.47	Puppy: 0.95
One: 0.51	Cute: 0.53	Is: 0.01
<EOS>: 0.04		

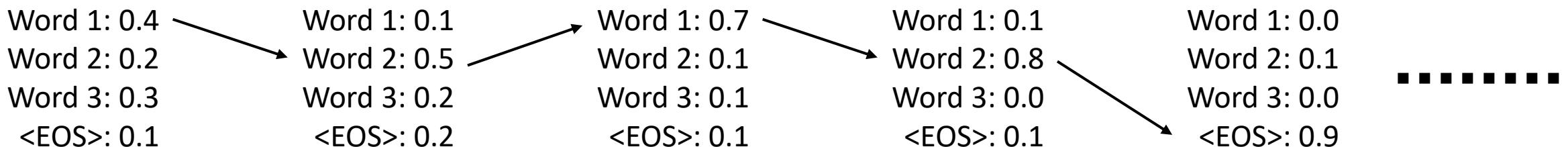


$$p(y_{i,1:T_y} | x_{i,1:T}) = \prod_{t=1}^{T_y} p(y_{i,t} | x_{i,1:T}, y_{i,0:t-1})$$

probabilities at each time step

If we want to maximize the product of **all** probabilities, we should not just greedily select the highest probability on the first step!

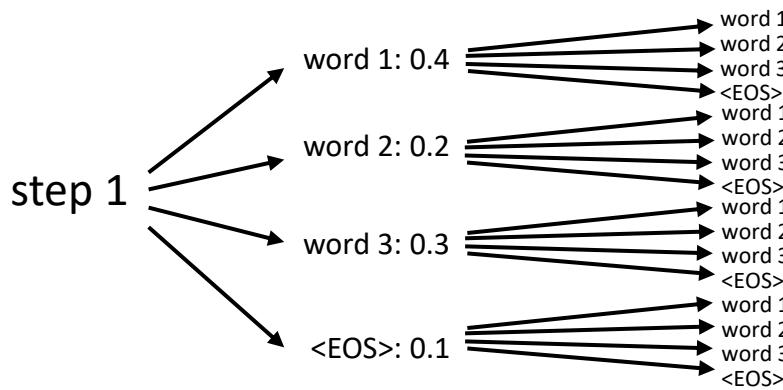
How many possible decodings are there?



for M words, in general there are M^T sequences
of length T

any one of these might be the optimal one!

Decoding is a **search** problem



We could use *any* tree search algorithm

But exact search in this case is **very**
expensive

Fortunately, the **structure** of this problem
makes some simple **approximate search**
methods work **very well**

Decoding with approximate search

Basic intuition: while choosing the **highest-probability** word on the first step may not be optimal, choosing a **very low-probability** word is very unlikely to lead to a good result

Equivalently: we can't be greedy, but we can be *somewhat* greedy

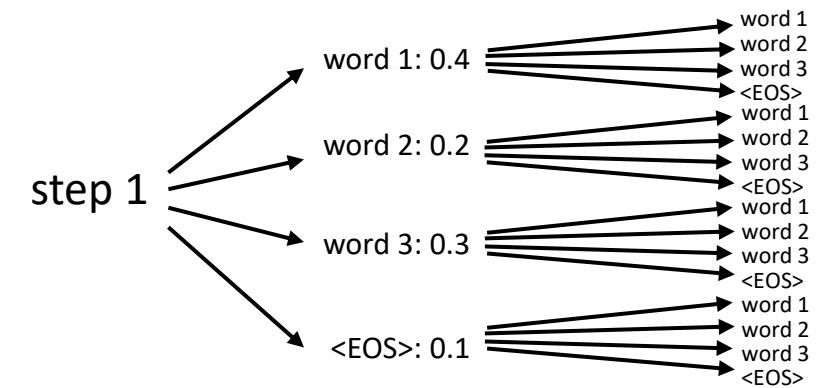
This is not true in general! This is a guess based on what we know about sequence decoding.

Beam search intuition: store the **k** best sequences **so far**, and update each of them.

special case of **k = 1** is just greedy decoding

often use **k** around 5-10

Decoding is a **search** problem



Beam search example

$$p(y_{i,1:T_y} | x_{i,1:T}) = \prod_{t=1}^{T_y} p(y_{i,t} | x_{i,1:T}, y_{i,0:t-1})$$

$$\log p(y_{i,1:T_y} | x_{i,1:T}) = \sum_{t=1}^{T_y} \log p(y_{i,t} | x_{i,1:T}, y_{i,0:t-1})$$

in practice, we **sum up** the log probabilities as we go (to avoid underflow)

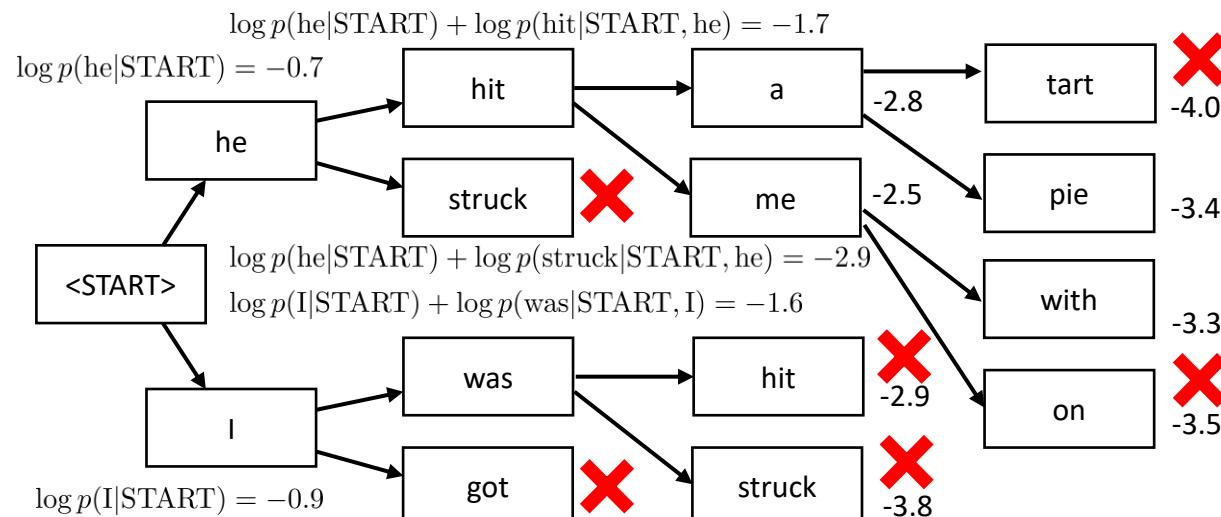
Example (CS224n, Christopher Manning):

translate (Fr->En): il a m'entarté

k = 2 (track the 2 most likely hypotheses)

(he hit me with a pie)

no perfectly equivalent English word, makes this hard

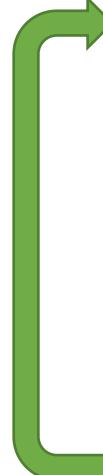


Beam search summary

$$\log p(y_{i,1:T_y} | x_{i,1:T}) = \sum_{t=1}^{T_y} \log p(y_{i,t} | x_{i,1:T}, y_{i,0:t-1})$$

at each time step t :

there are k of these

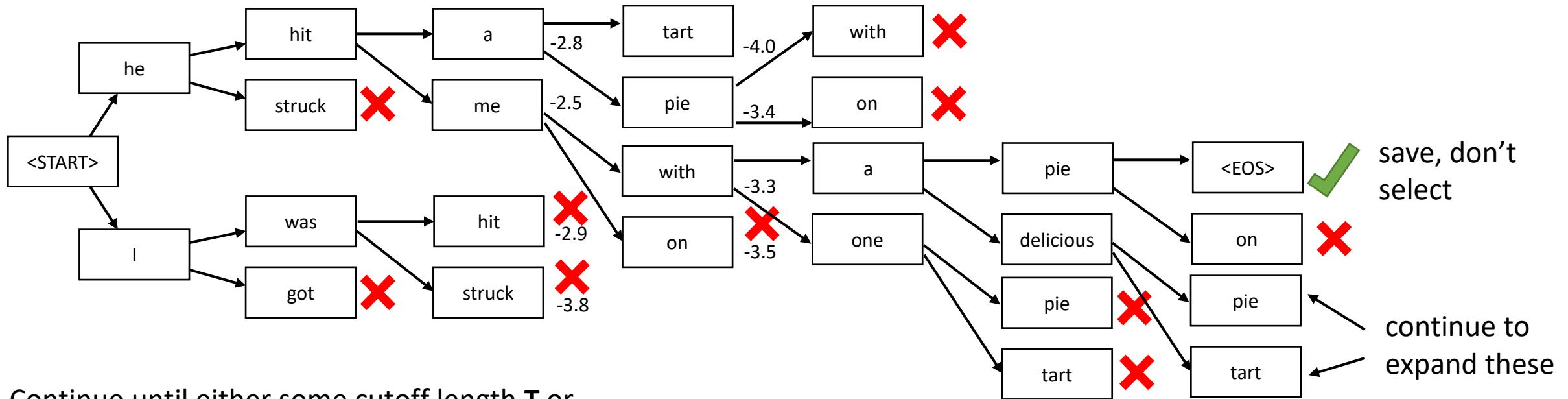
- 
1. for each hypothesis $y_{1:t-1,i}$ that we are tracking:
find the top k tokens $y_{t,i,1}, \dots, y_{t,i,k}$
 2. sort the resulting k^2 length t sequences by their *total* log-probability
 3. keep the top k
 4. advance each hypothesis to time $t + 1$
- very easy, we get this from the softmax log-probs

When do we stop decoding?

Let's say one of the highest-scoring hypotheses ends in <END>

Save it, along with its score, but do **not** pick it to expand further (there is nothing to expand)

Keep expanding the **k** remaining best hypotheses



Continue until either some cutoff length **T** or
until we have **N** hypotheses that end in <EOS>

Which sequence do we pick?

At the end we might have something like this:

he hit me with a pie

$\log p = -4.5$

he threw a pie

$\log p = -3.2$

I was hit with a pie that he threw

$\log p = -7.2$

this is best, right?

$$\log p(y_{i,1:T}|x_{i,1:T}) = \sum_{t=1}^T \log p(y_{i,t}|x_{i,1:T}, y_{i,0:t-1})$$

Problem: $p < 1$ always, hence $\log p < 0$ always

The longer the sequence the lower its total score (more negative numbers added together)

Simple “fix”:

just divide by sequence length

$$\text{score}(y_{i,1:T}|x_{i,1:T}) = \frac{1}{T} \sum_{t=1}^T \log p(y_{i,t}|x_{i,1:T}, y_{i,0:t-1})$$

Beam search summary

$$\text{score}(y_{i,1:T}|x_{i,1:T}) = \frac{1}{T} \sum_{t=1}^T \log p(y_{i,t}|x_{i,1:T}, y_{i,0:t-1})$$

at each time step t :

- 
1. for each hypothesis $y_{1:t-1,i}$ that we are tracking:
find the top k tokens $y_{t,i,1}, \dots, y_{t,i,k}$
 2. sort the resulting k^2 length t sequences by their *total* log-probability
 3. save any sequences that end in EOS
 4. keep the top k
 5. advance each hypothesis to time $t + 1$ if $t < H$

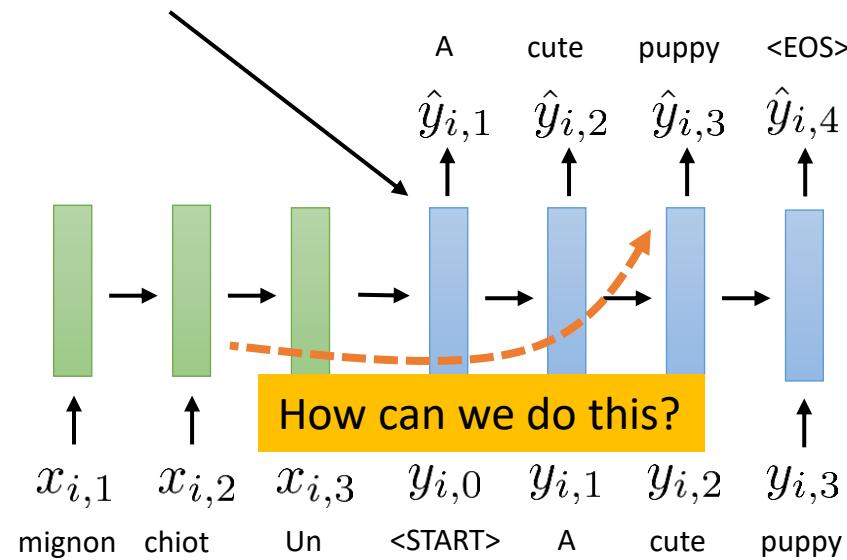
return saved sequence with highest score

Attention

The bottleneck problem

all information about the source sequence
is contained in these activations

this forms a bottleneck



Idea: what if we could somehow “peek” at the source sentence while decoding?

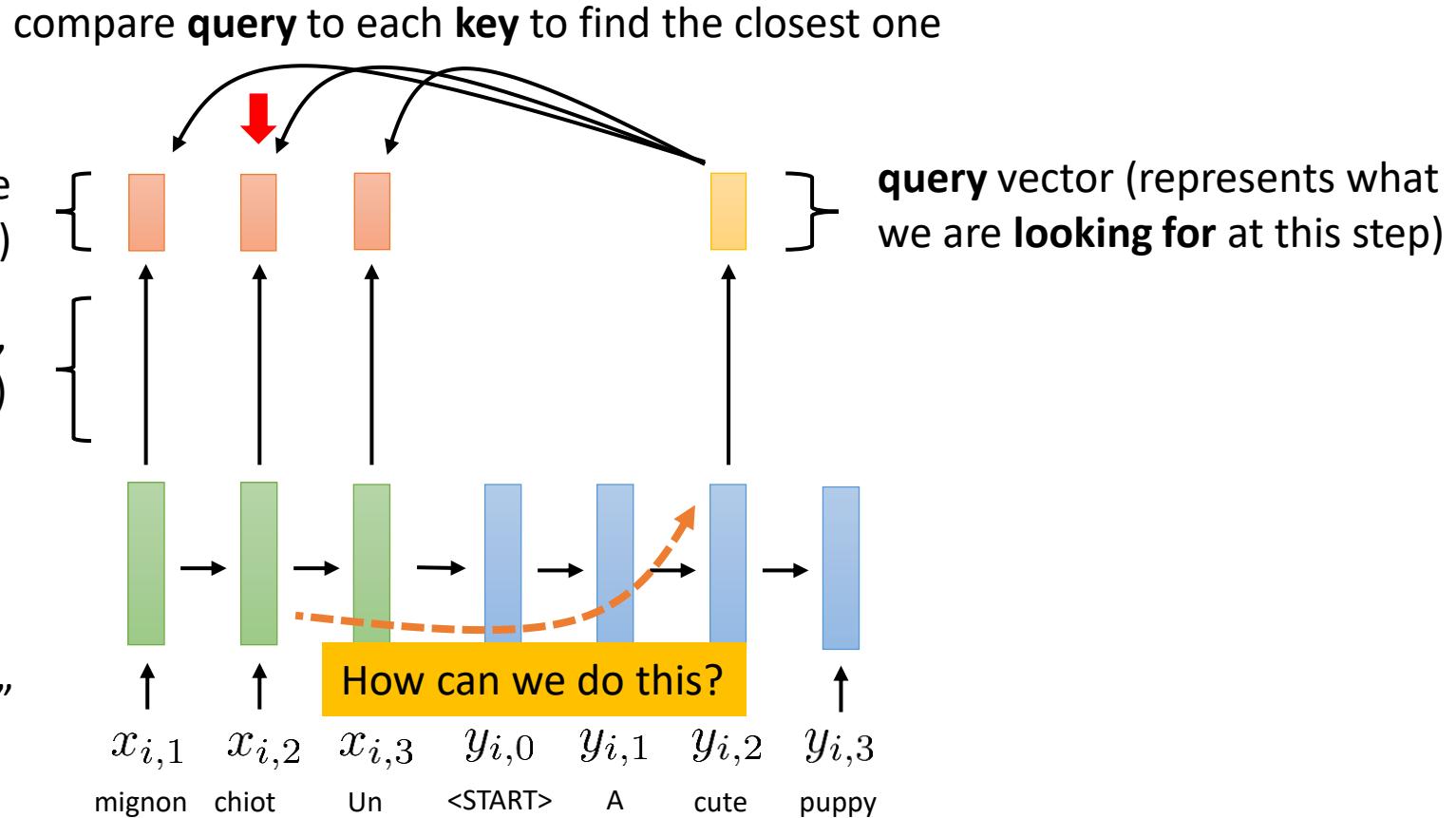
Can we “peek” at the input?

key vector (represents what type of info is present at this step)

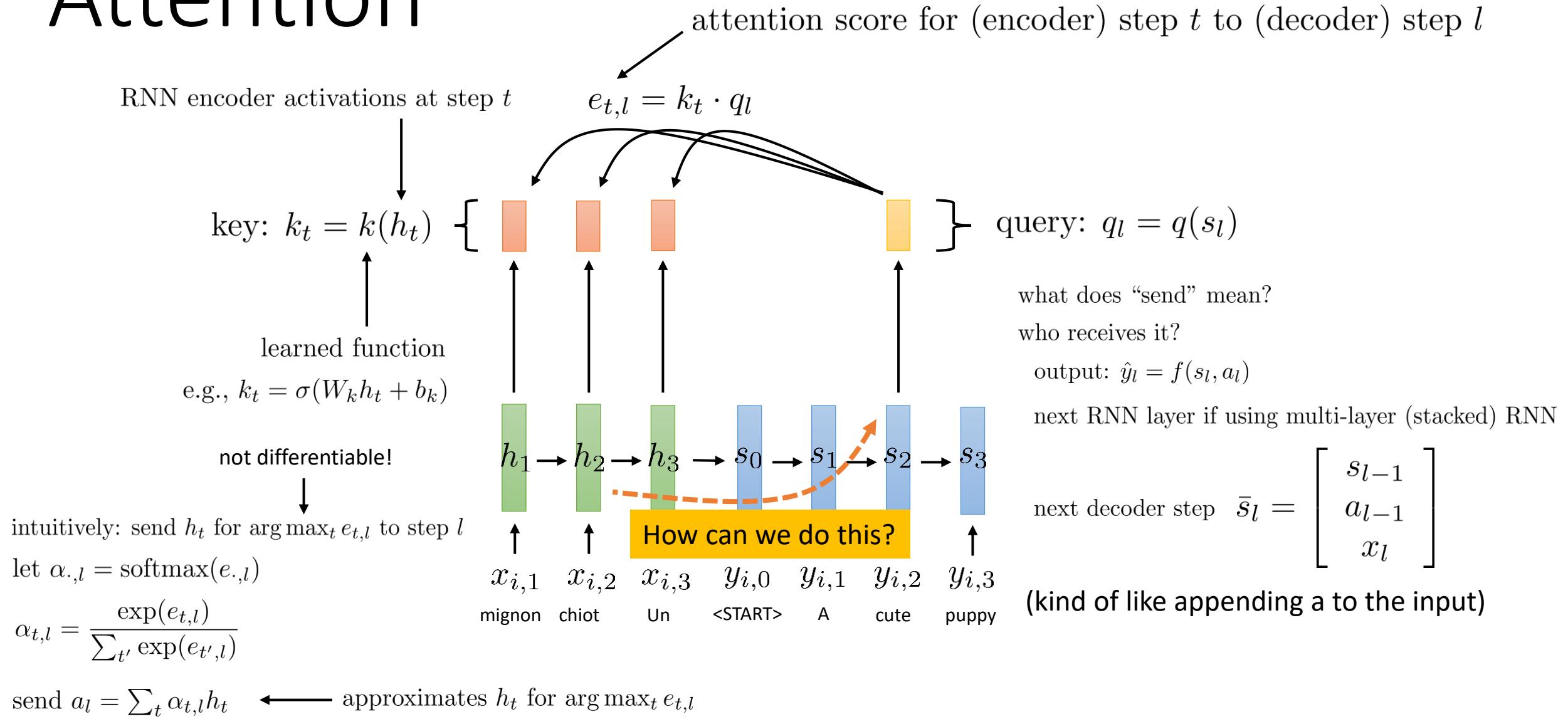
some function (e.g., linear layer + ReLU)

(crude) intuition: key might encode “the subject of the sentence,” and query might ask for “the subject of the sentence”

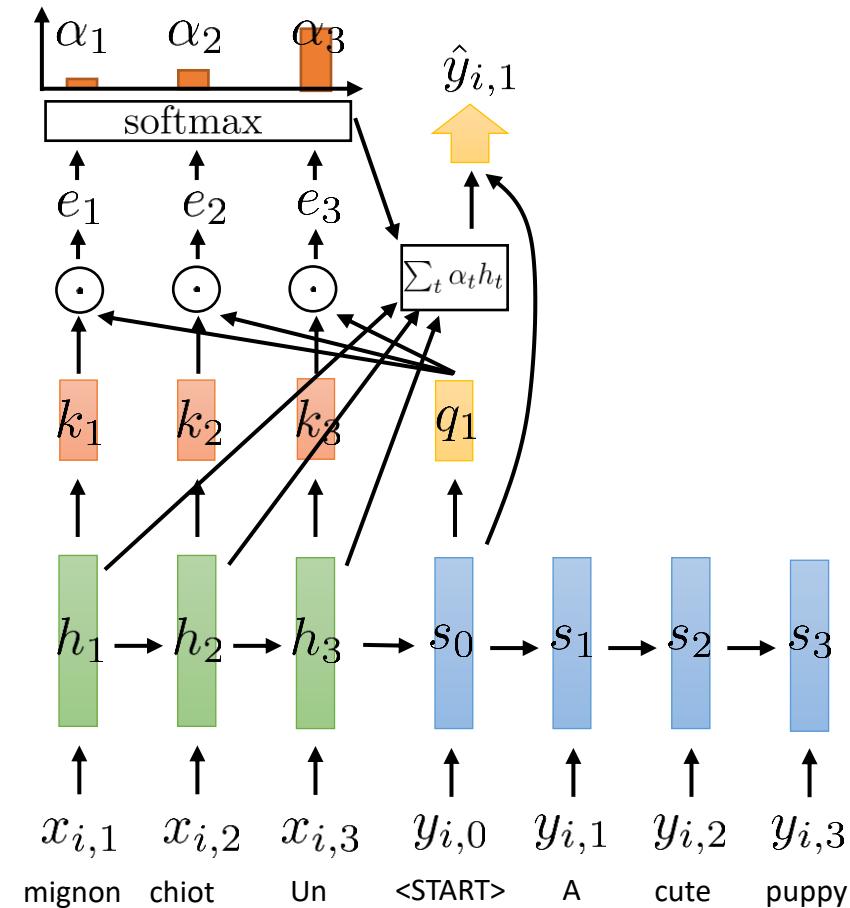
In reality what keys and queries mean is learned – we do not have to select it manually!



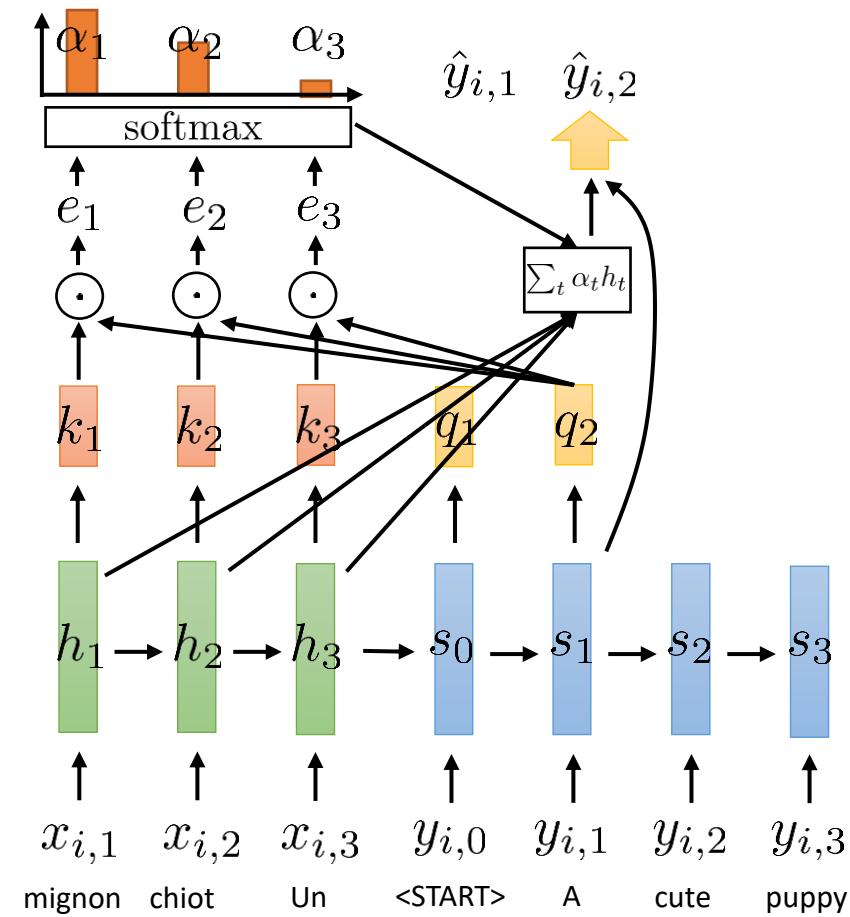
Attention



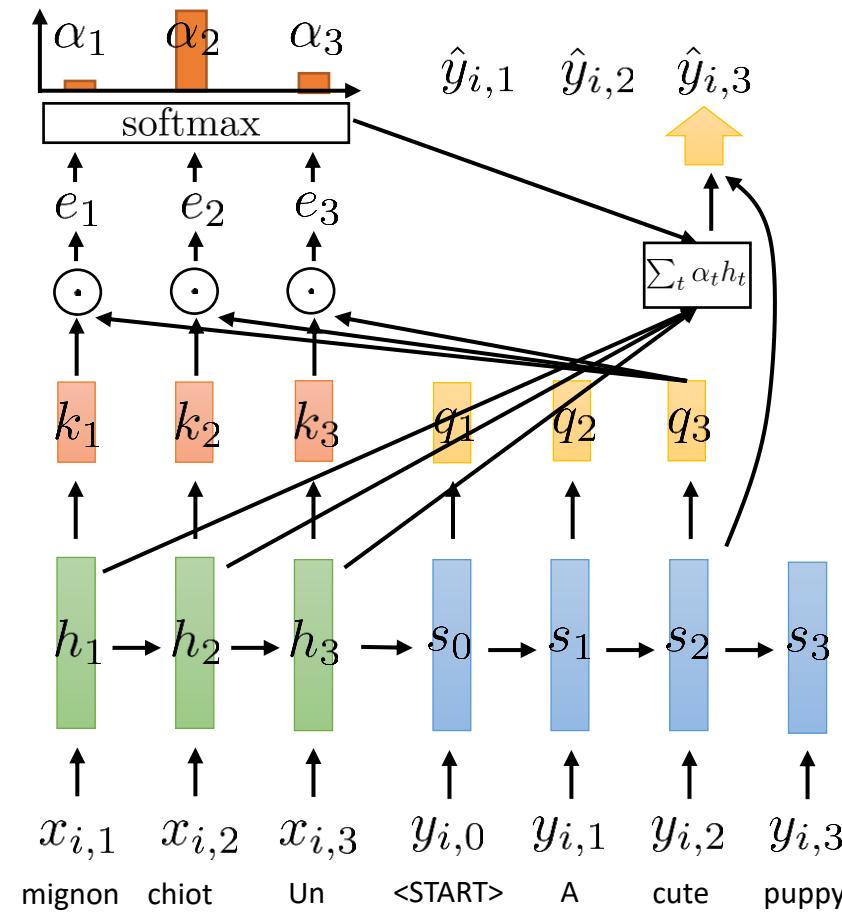
Attention Walkthrough (Example)



Attention Walkthrough (Example)



Attention Walkthrough (Example)



Attention Equations

Encoder-side:

$$k_t = k(h_t)$$

Decoder-side:

$$q_l = q(s_l)$$

$$e_{t,l} = k_t \cdot q_l$$

$$\alpha_{t,l} = \frac{\exp(e_{t,l})}{\sum_{t'} \exp(e_{t',l})}$$

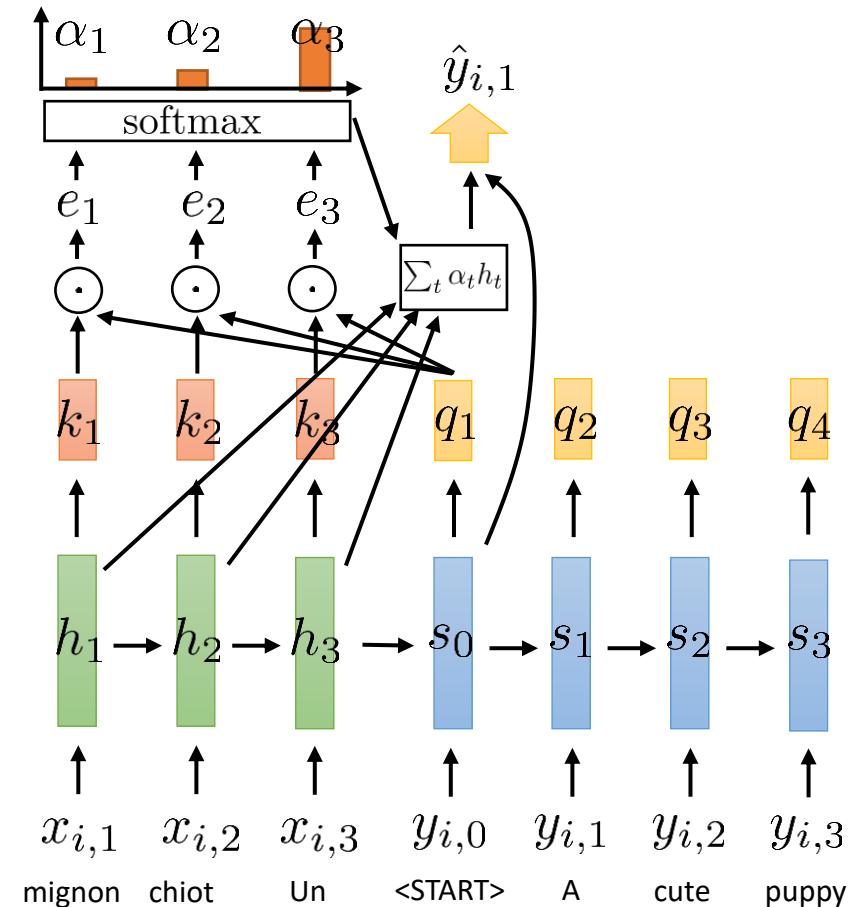
$$a_l = \sum_t \alpha_t h_t$$

Could use this in various ways:

concatenate to hidden state: $\begin{bmatrix} s_{l-1} \\ a_{l-1} \\ x_l \end{bmatrix}$

use for readout, e.g.: $\hat{y}_l = f(s_t, a_l)$

concatenate as input to next RNN layer



Attention Variants

Simple key-query choice: k and q are identity functions

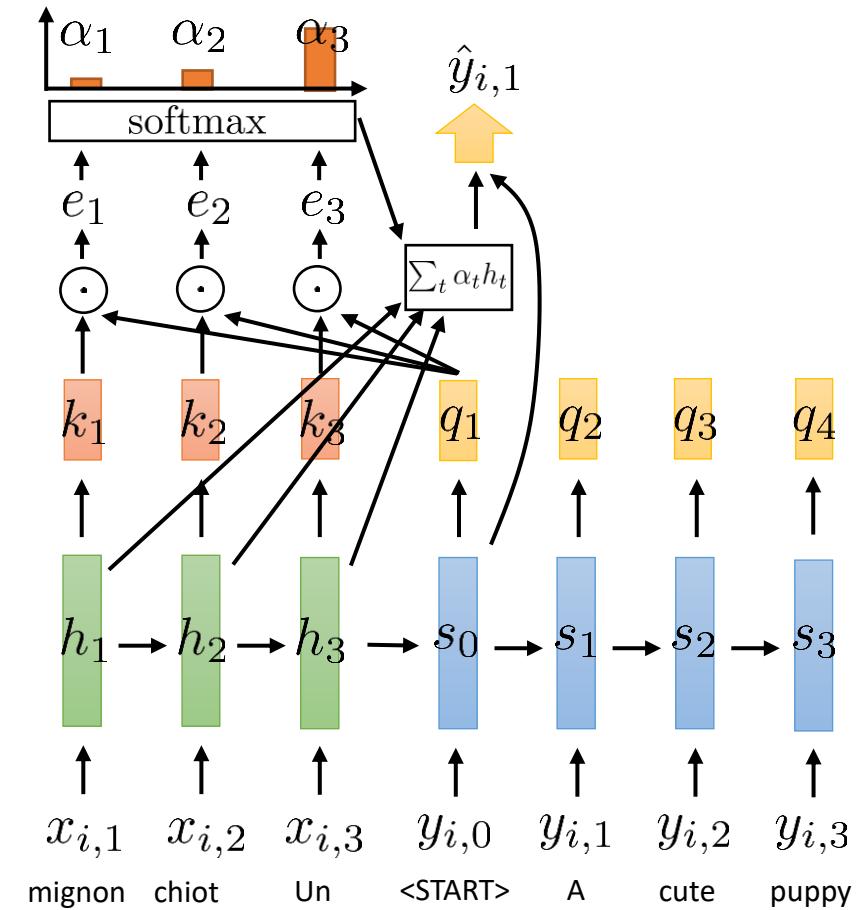
$$k_t = h_t \quad q_l = s_l$$

Decoder-side:

$$e_{t,l} = h_t \cdot s_l$$

$$\alpha_{t,l} = \frac{\exp(e_{t,l})}{\sum_{t'} \exp(e_{t',l})}$$

$$a_l = \sum_t \alpha_t h_t$$



Attention Variants

Linear multiplicative attention:

$$k_t = W_k h_t \quad q_l = W_q s_l$$

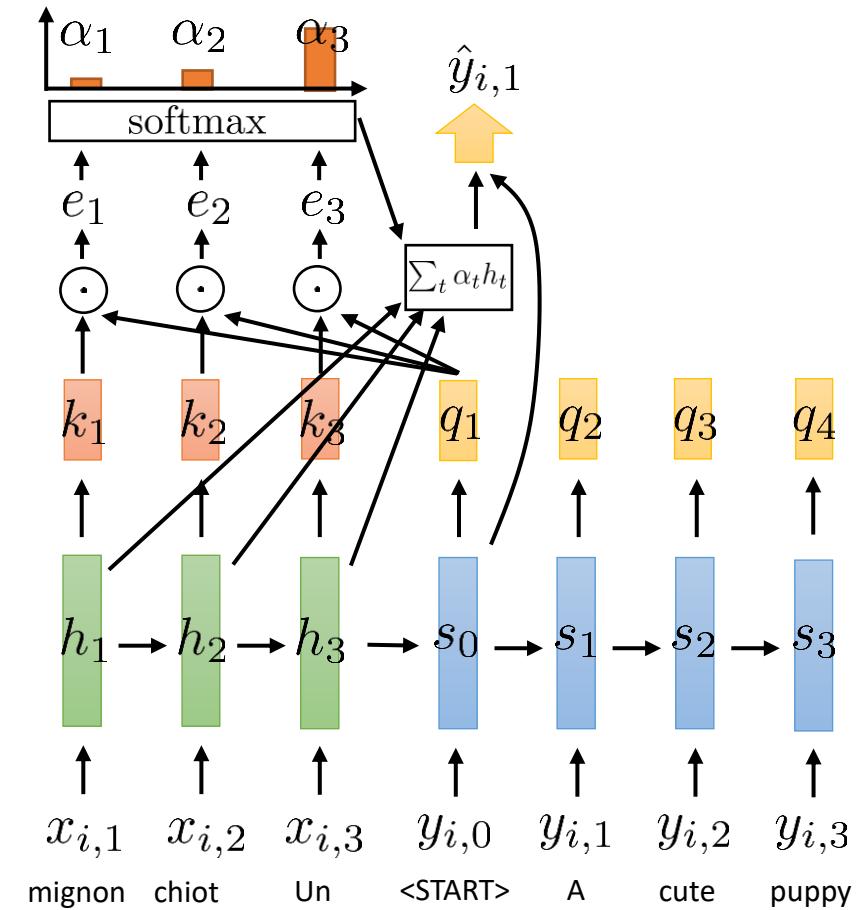
Decoder-side:

$$e_{t,l} = h_t^T W_k^T W_q s_l = h_t^T W_e s_l$$

$$\alpha_{t,l} = \frac{\exp(e_{t,l})}{\sum_{t'} \exp(e_{t',l})}$$

$$a_l = \sum_t \alpha_t h_t$$

just learn this matrix



Attention Variants

Learned value encoding:

Encoder-side:

$$k_t = k(h_t)$$

Decoder-side:

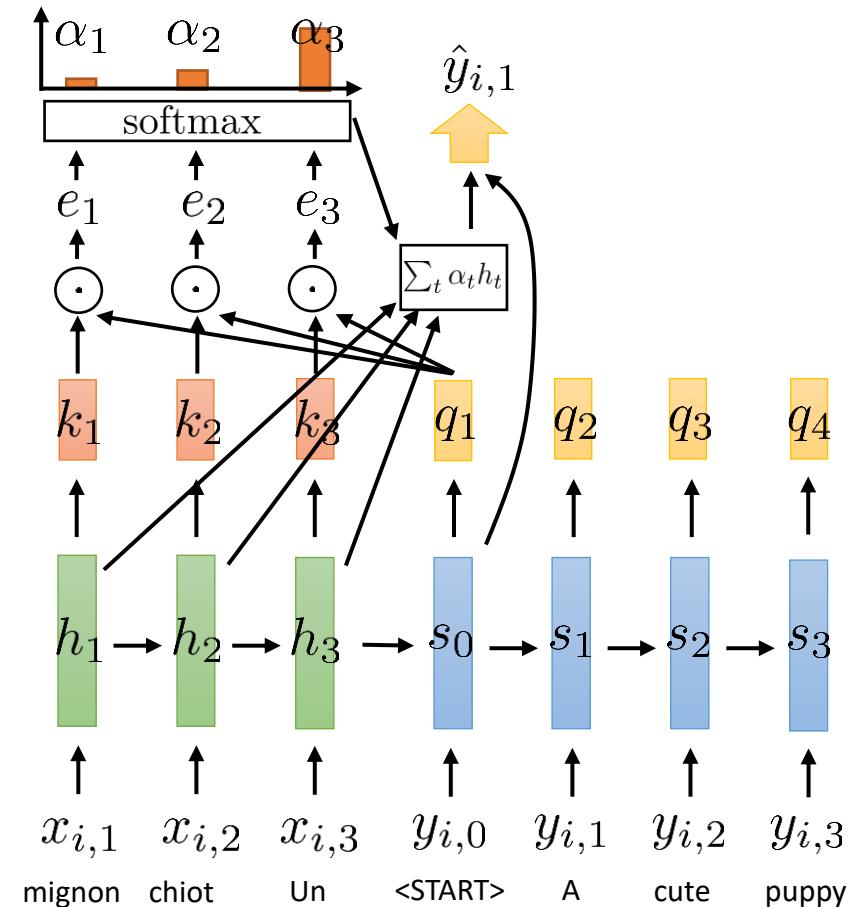
$$q_l = q(s_l)$$

$$e_{t,l} = k_t \cdot q_l$$

$$\alpha_{t,l} = \frac{\exp(e_{t,l})}{\sum_{t'} \exp(e_{t',l})}$$

$$a_l = \sum_t \alpha_t v(h_t)$$

some learned function



Attention Summary

Every encoder step t produces a key k_t

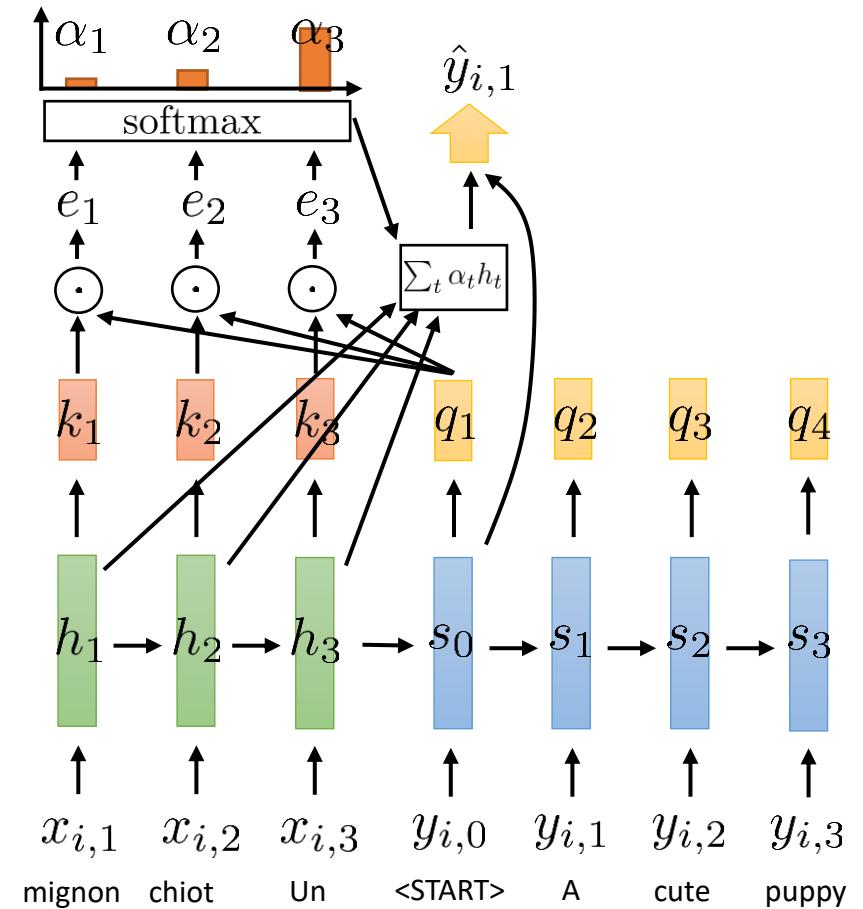
Every decoder step l produces a query q_l

Decoder gets “sent” encoder activation h_t
corresponding to largest value of $k_t \cdot q_l$

actually gets $\sum_t \alpha_t h_t$

Why is this **good**?

- Attention is **very** powerful, because now all decoder steps are connected to **all** encoder steps!
- Connections go from $O(T)$ to $O(1)$
- Gradients are much better behaved ($O(1)$ propagation length)
- Becomes very important for very long sequences
- Bottleneck is much less important
- This works much better in practice



Transformers

Designing, Visualizing and Understanding Deep Neural Networks

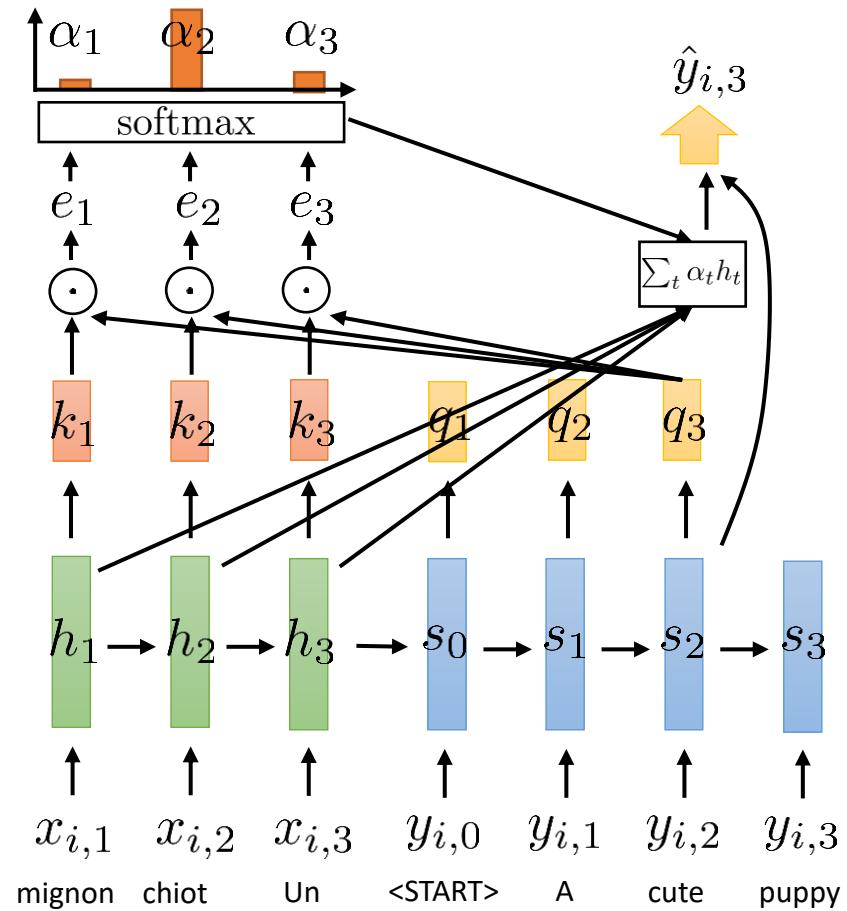
CS W182/282A

Instructor: Sergey Levine
UC Berkeley



Is Attention All We Need?

Attention



If we have **attention**, do we even need recurrent connections?

Can we **transform** our RNN into a **purely attention-based** model?

Attention can access **every** time step

Can in principle do **everything** that recurrence can, and more!

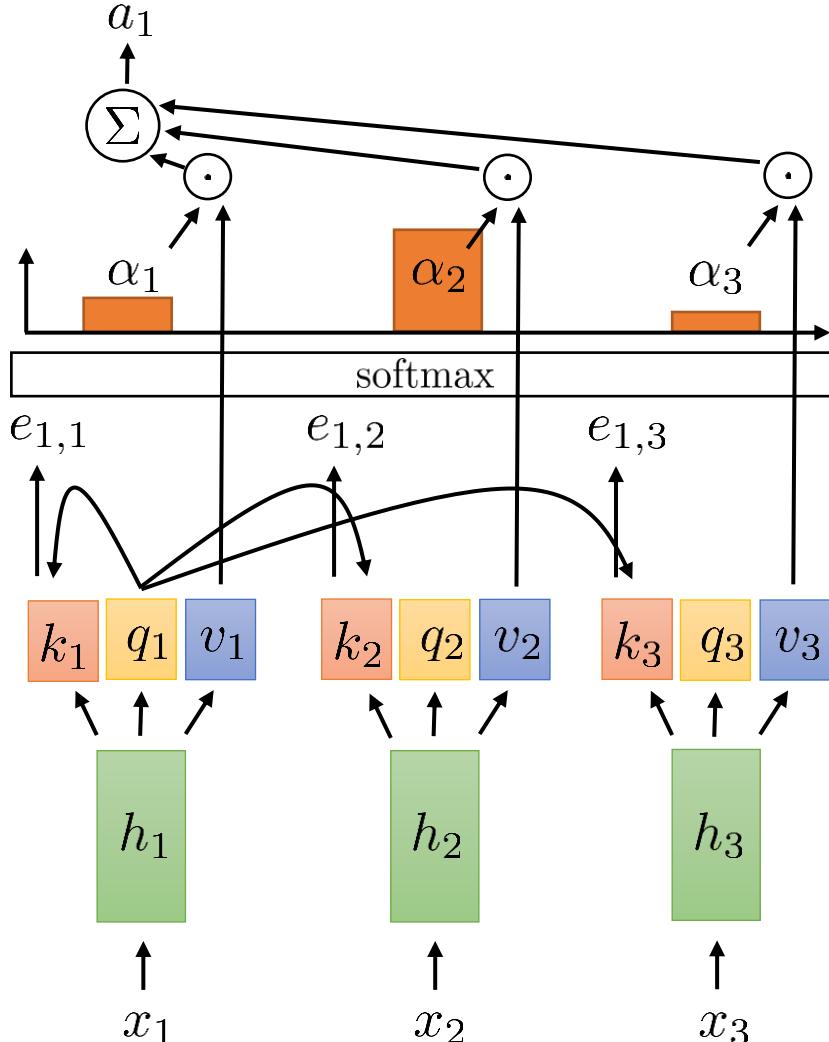
This has a few issues we must overcome:

Problem 1: now step $l = 2$ can't access s_1 or s_0

The encoder has no temporal dependencies at all!

We **must** fix this first

Self-Attention



$$a_l = \sum_t \alpha_{l,t} v_t$$

$$\alpha_{l,t} = \exp(e_{l,t}) / \sum_{t'} \exp(e_{l,t'})$$

$$e_{l,t} = q_l \cdot k_t$$

we'll see why this is important soon

$$v_t = v(h_t) \quad \text{before just had } v(h_t) = h_t, \text{ now e.g. } v(h_t) = W_v h_t$$

$$k_t = k(h_t) \text{ (just like before)} \quad \text{e.g., } k_t = W_k h_t$$

$$q_t = q(h_t) \quad \text{e.g., } q_t = W_q h_t$$

this is *not* a recurrent model!

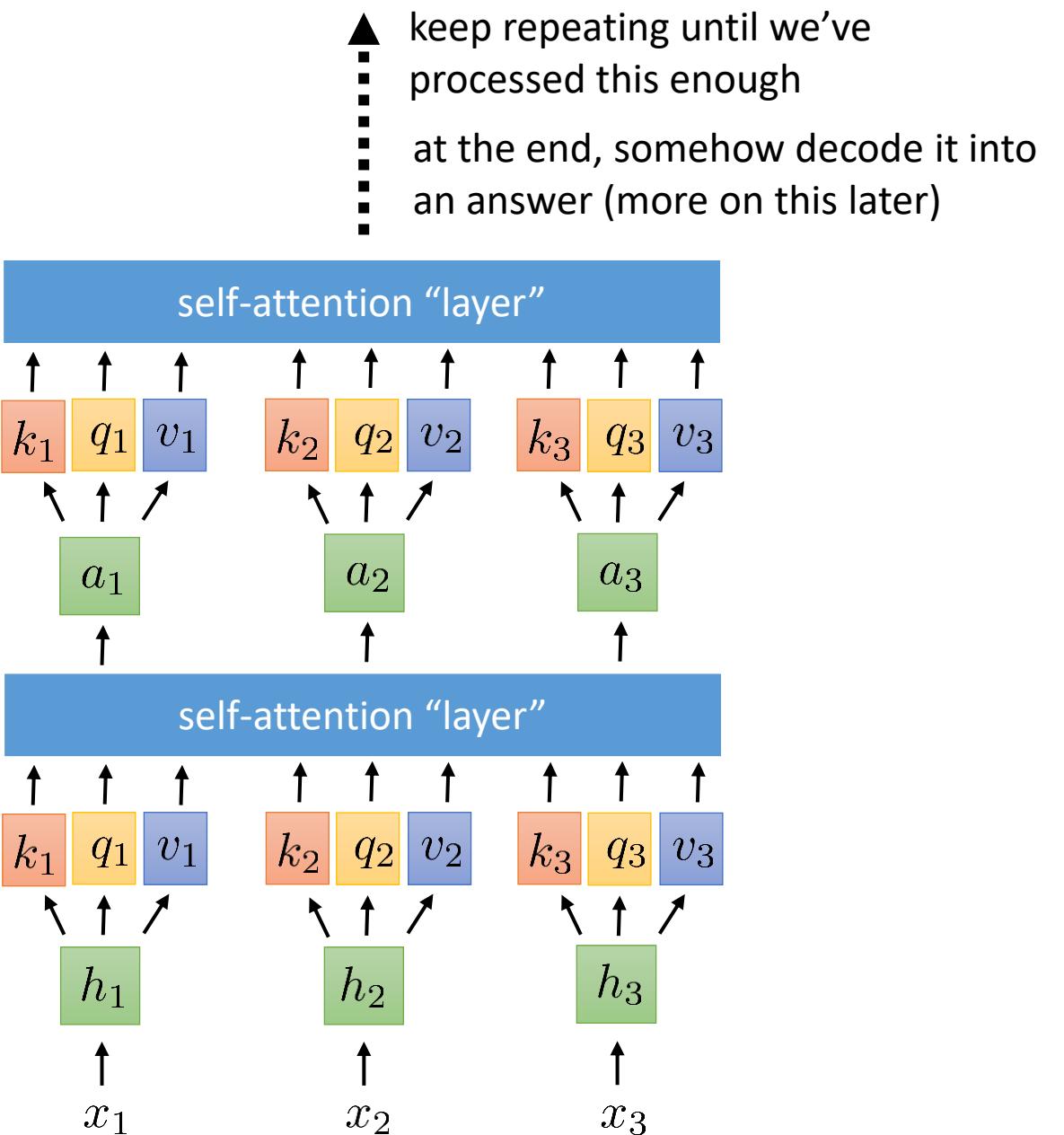
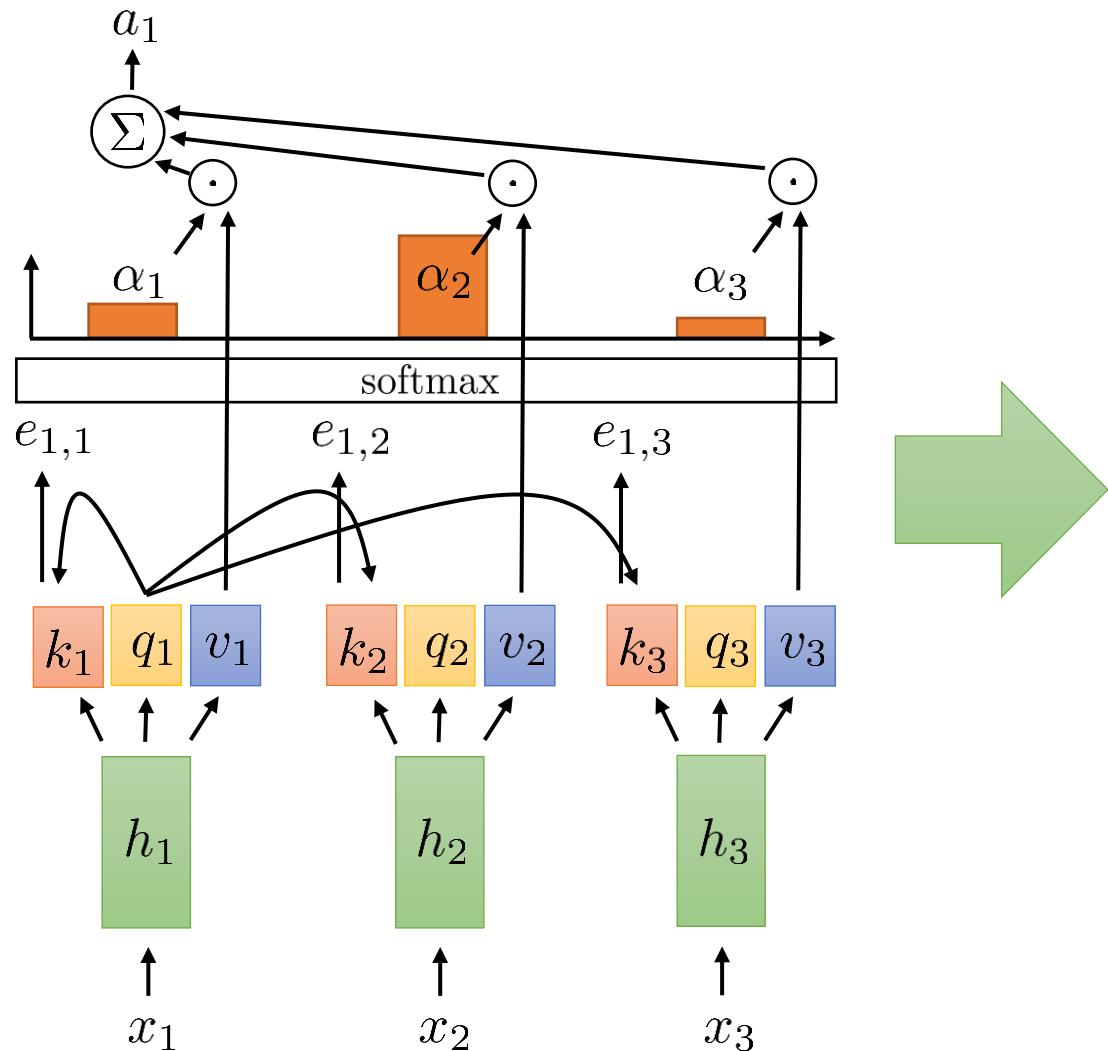
but still weight sharing:

$$h_t = \sigma(Wx_t + b)$$

shared weights at all time steps

(or any other nonlinear function)

Self-Attention



keep repeating until we've
processed this enough
at the end, somehow decode it into
an answer (more on this later)

From Self-Attention to Transformers

The basic concept of **self-attention** can be used to develop a very powerful type of sequence model, called a **transformer**

But to make this actually work, we need to develop a few additional components to address some fundamental limitations

1. Positional encoding addresses lack of sequence information
2. Multi-headed attention allows querying multiple positions at each layer
3. Adding nonlinearities so far, each successive layer is *linear* in the previous one
4. Masked decoding how to prevent attention lookups into the future?

$$a_l = \sum_t \alpha_{l,t} v_t$$
$$v_t = W_v h_t$$

Sequence Models with Self-Attention

From Self-Attention to Transformers

The basic concept of **self-attention** can be used to develop a very powerful type of sequence model, called a **transformer**

But to make this actually work, we need to develop a few additional components to address some fundamental limitations

1. Positional encoding

addresses lack of sequence information

2. Multi-headed attention

allows querying multiple positions at each layer

3. Adding nonlinearities

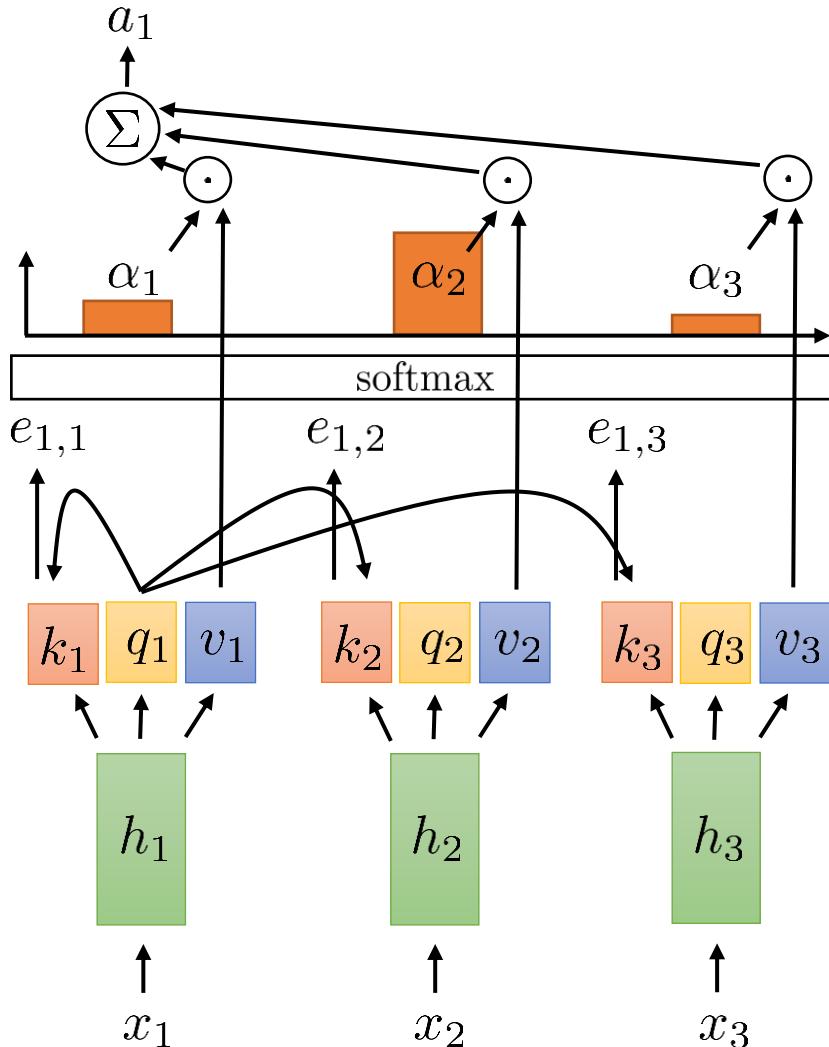
so far, each successive layer is *linear* in the previous one

4. Masked decoding

how to prevent attention lookups into the future?

$$a_l = \sum_t \alpha_{l,t} v_t$$
$$v_t = W_v h_t$$

Positional encoding: what is the order?



what we see:

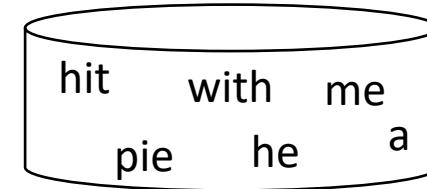
he hit me with a pie

what naïve self-attention sees:

a pie hit me with he

a hit with me he pie

he pie me with a hit



most alternative orderings are nonsense, but some change the meaning

in general the position of words in a sentence carries information!

Idea: add some information to the representation at the beginning that indicates where it is in the sequence!

$$h_t = f(x_t, t)$$

some function

Positional encoding: sin/cos

Naïve positional encoding: just append t to the input

$$\bar{x}_t = \begin{bmatrix} x_t \\ t \end{bmatrix}$$

This is not a great idea, because **absolute** position is less important than **relative** position

I walk my dog every day



every single day I walk my dog



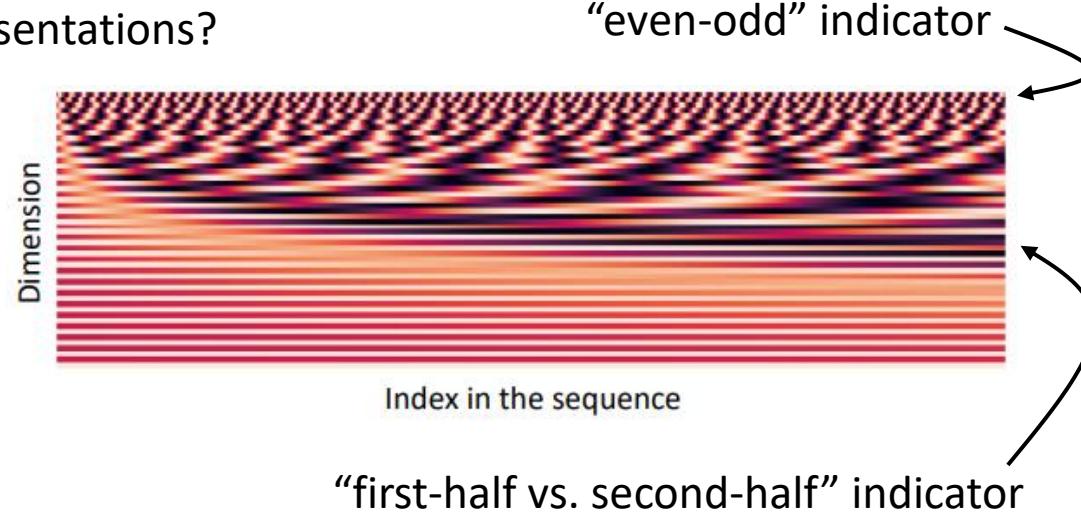
The fact that “my dog” is right after “I walk” is the important part, not its absolute position

we want to represent **position** in a way that tokens with similar **relative** position have similar **positional encoding**

Idea: what if we use **frequency-based** representations?

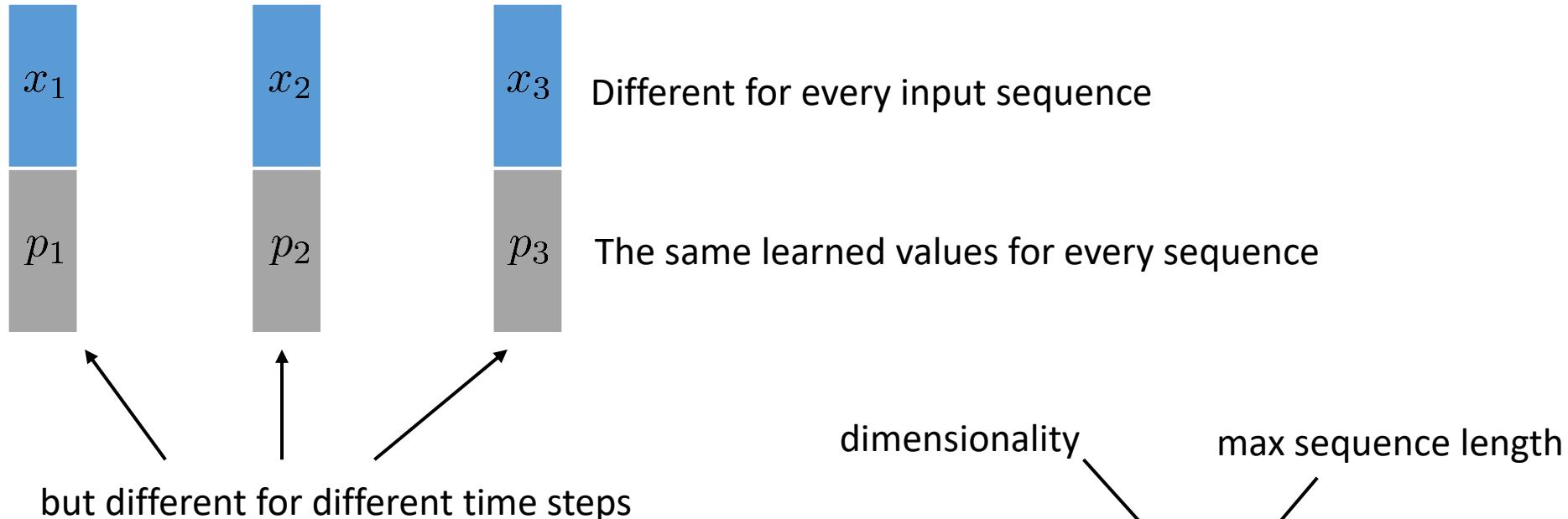
$$p_t = \begin{bmatrix} \sin(t/10000^{2*1/d}) \\ \cos(t/10000^{2*1/d}) \\ \sin(t/10000^{2*2/d}) \\ \cos(t/10000^{2*2/d}) \\ \dots \\ \sin(t/10000^{2*\frac{d}{2}/d}) \\ \cos(t/10000^{2*\frac{d}{2}/d}) \end{bmatrix}$$

dimensionality of positional encoding



Positional encoding: learned

Another idea: just learn a positional encoding



How many values do we need to learn?

$$P = [p_1, p_2, \dots, p_T] \in R^{d \times T}$$

+ more flexible (and perhaps more optimal) than sin/cos encoding

+ a bit more complex, need to pick a max sequence length (and can't generalize beyond it)

How to incorporate positional encoding?

At each step, we have x_t and p_t

Simple choice: just concatenate them

$$\bar{x}_t = \begin{bmatrix} x_t \\ p_t \end{bmatrix}$$

More often: just add after **embedding** the input

input to self-attention is $\text{emb}(x_t) + p_t$



some learned function (e.g., some fully connected layers with linear layers + nonlinearities)

From Self-Attention to Transformers

The basic concept of **self-attention** can be used to develop a very powerful type of sequence model, called a **transformer**

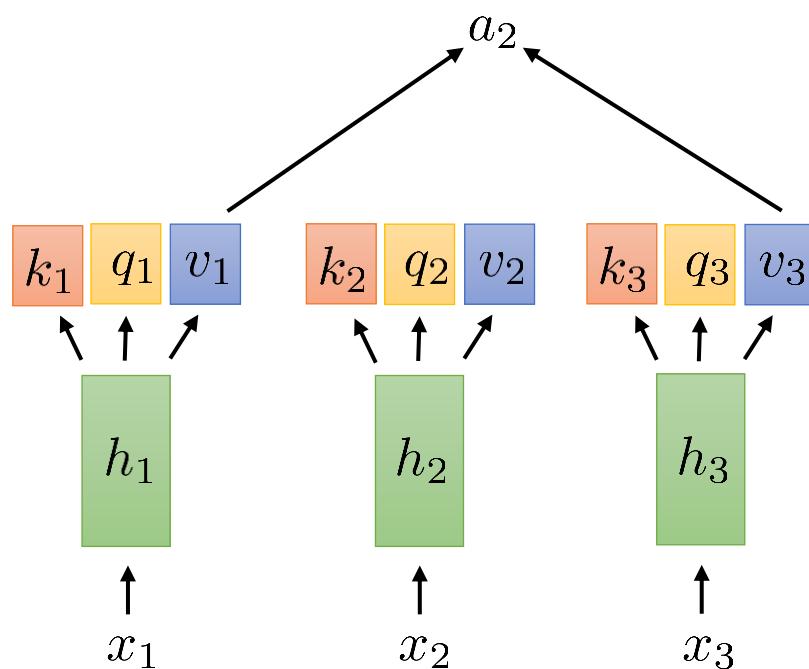
But to make this actually work, we need to develop a few additional components to address some fundamental limitations

1. Positional encoding addresses lack of sequence information
2. Multi-headed attention allows querying multiple positions at each layer
3. Adding nonlinearities so far, each successive layer is *linear* in the previous one
4. Masked decoding how to prevent attention lookups into the future?

$$a_l = \sum_t \alpha_{l,t} v_t$$
$$v_t = W_v h_t$$

Multi-head attention

Since we are relying **entirely** on attention now, we might want to incorporate **more than one** time step



$$a_l = \sum_t \alpha_{l,t} v_t$$

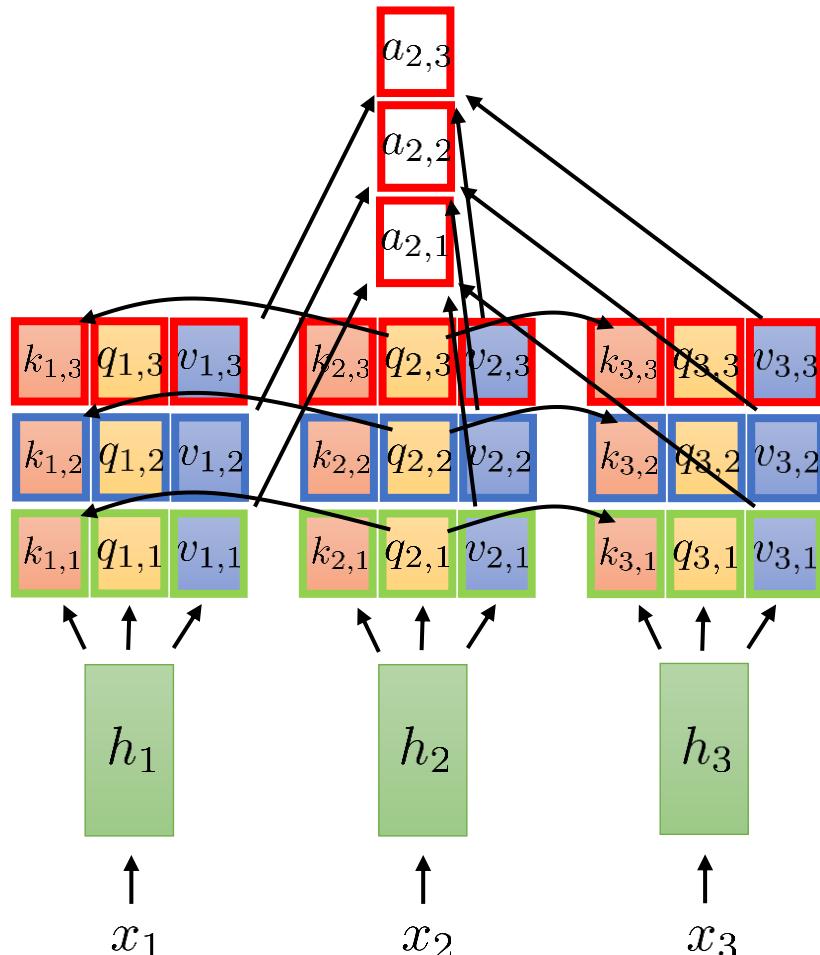
because of softmax, this will
be dominated by one value

$$e_{l,t} = q_l \cdot k_t$$

hard to specify that you want two
different things (e.g., the subject
and the object in a sentence)

Multi-head attention

Idea: have multiple keys, queries, and values for every time step!



full attention vector formed by concatenation:

$$a_2 = \begin{bmatrix} a_{2,1} \\ a_{2,2} \\ a_{2,3} \end{bmatrix}$$

compute weights **independently** for each head

$$e_{l,t,i} = q_{l,i} \cdot k_{l,i}$$

$$\alpha_{l,t,i} = \exp(e_{l,t,i}) / \sum_{t'} \exp(e_{l,t',i})$$

$$a_{l,i} = \sum_t \alpha_{l,t,i} v_{t,i}$$

around **8** heads seems to work
pretty well for big models

From Self-Attention to Transformers

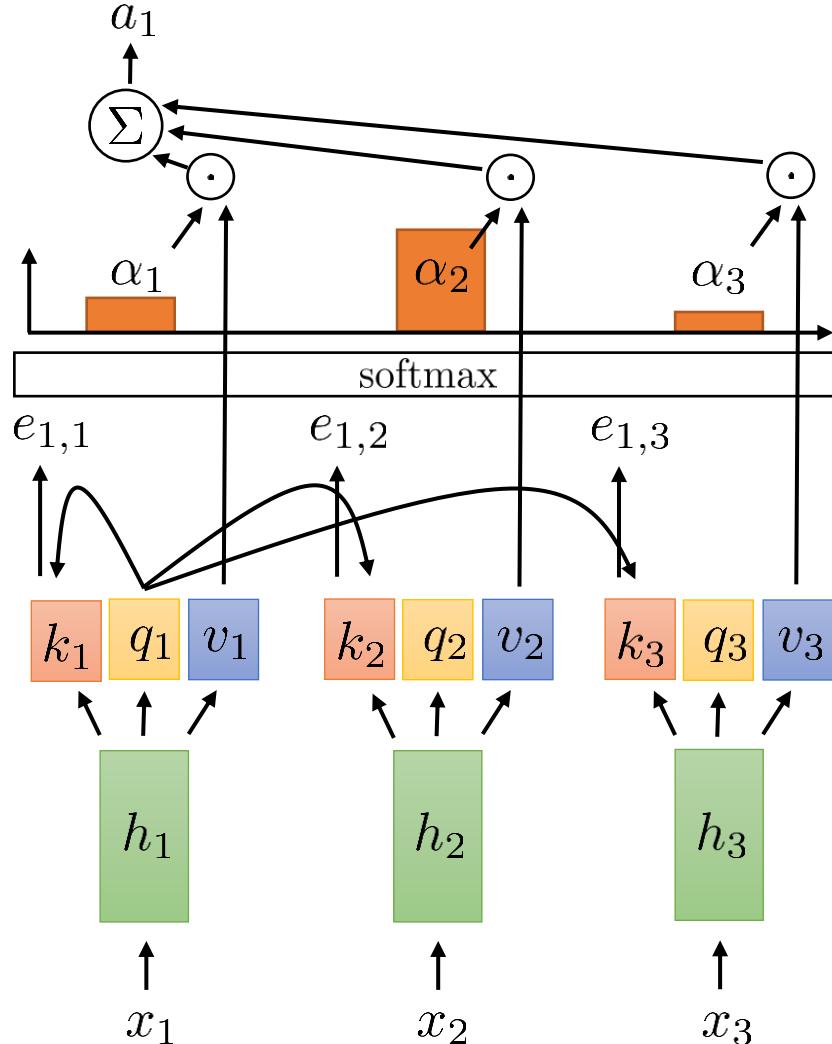
The basic concept of **self-attention** can be used to develop a very powerful type of sequence model, called a **transformer**

But to make this actually work, we need to develop a few additional components to address some fundamental limitations

1. Positional encoding addresses lack of sequence information
2. Multi-headed attention allows querying multiple positions at each layer
3. Adding nonlinearities so far, each successive layer is *linear* in the previous one
4. Masked decoding how to prevent attention lookups into the future?

$$a_l = \sum_t \alpha_{l,t} v_t$$
$$v_t = W_v h_t$$

Self-Attention is Linear



$$k_t = W_k h_t \quad q_t = W_q h_t \quad v_t = W_v h_t$$

$$\alpha_{l,t} = \exp(e_{l,t}) / \sum_{t'} \exp(e_{l,t'})$$

$$e_{l,t} = q_l \cdot k_t$$

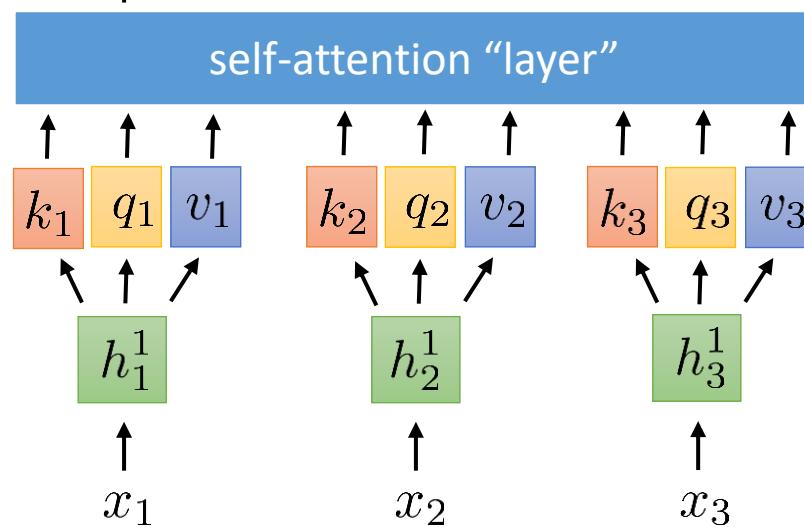
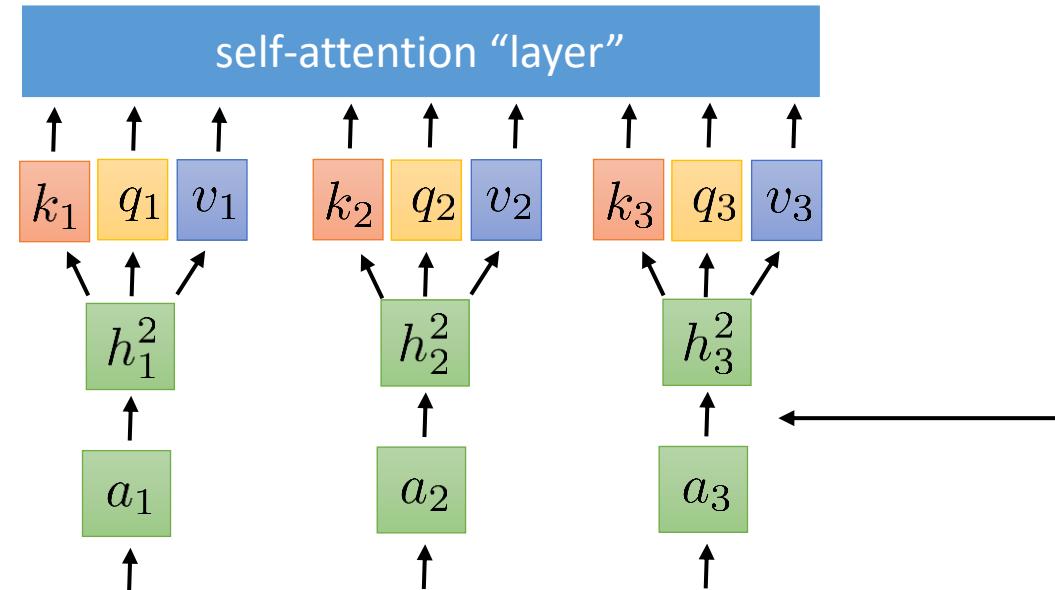
$$a_l = \sum_t \alpha_{l,t} v_t = \sum_t \alpha_{l,t} W_v h_t = W_v \sum_t \alpha_{l,t} h_t$$

linear transformation non-linear weights

Every self-attention “layer” is a linear transformation of the previous layer
(with non-linear weights)

This is not very expressive

Alternating self-attention & nonlinearity



some non-linear (learned) function
e.g., $h_t^\ell = \sigma(W^\ell a_t^\ell + b^\ell)$

just a neural net applied at every position
after every self-attention layer!

Sometimes referred to as "position-wise feedforward network"

We'll describe some specific
commonly used choices shortly

From Self-Attention to Transformers

The basic concept of **self-attention** can be used to develop a very powerful type of sequence model, called a **transformer**

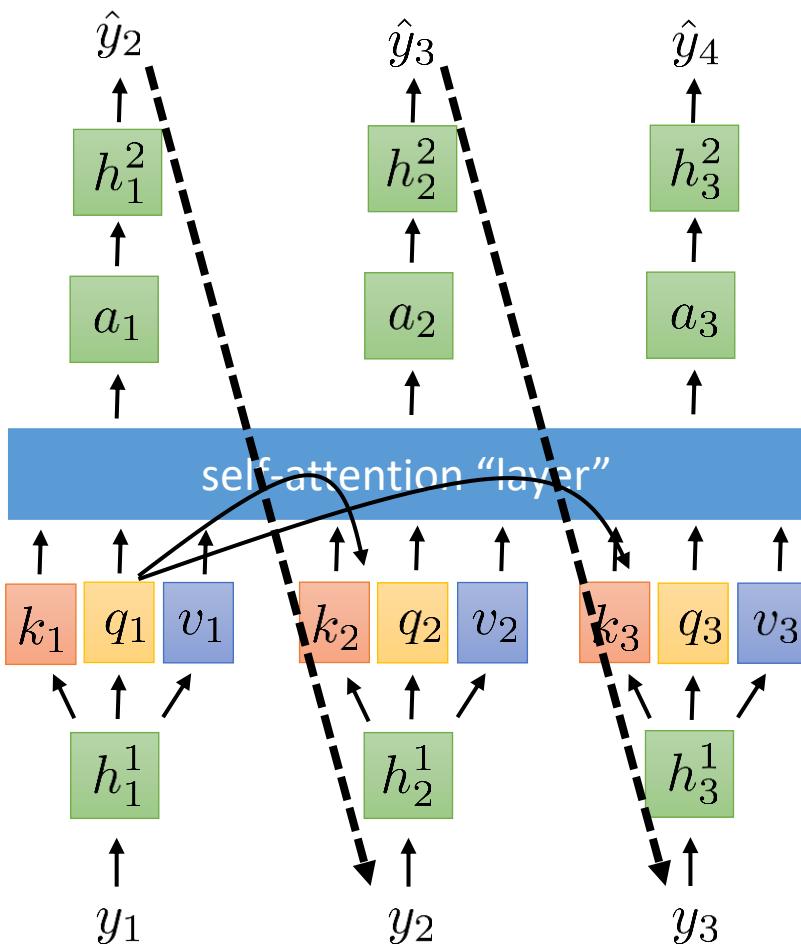
But to make this actually work, we need to develop a few additional components to address some fundamental limitations

1. Positional encoding addresses lack of sequence information
2. Multi-headed attention allows querying multiple positions at each layer
3. Adding nonlinearities so far, each successive layer is *linear* in the previous one
4. Masked decoding how to prevent attention lookups into the future?

$$a_l = \sum_t \alpha_{l,t} v_t$$
$$v_t = W_v h_t$$

Self-attention can see the future!

A **crude** self-attention “language model”:



(in reality, we would have many alternating self-attention layers and position-wise feedforward networks, not just one)

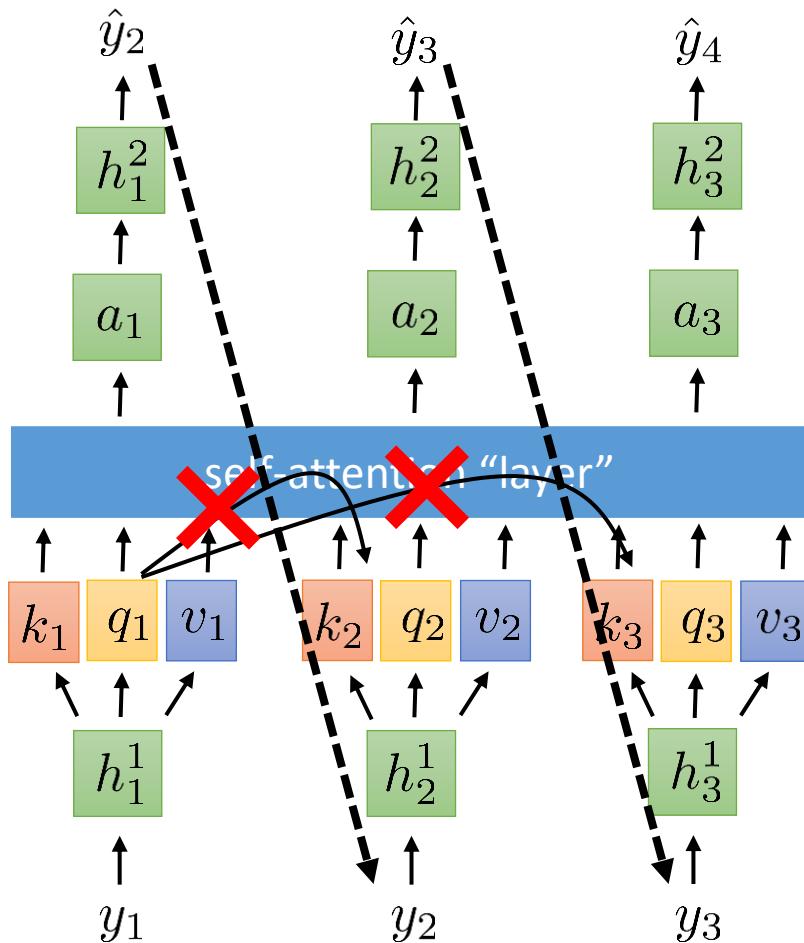
Big problem: self-attention at step 1 can look at the value at steps 2 & 3, which is based on the **inputs** at steps 2 & 3

At test time (when decoding), the **inputs** at steps 2 & 3 will be based on the output at step 1...

...which requires knowing the **input** at steps 2 & 3

Masked attention

A **crude** self-attention “language model”:



At test time (when decoding), the **inputs** at steps 2 & 3 will be based on the output at step 1...

...which requires knowing the **input** at steps 2 & 3

Must allow self-attention into the **past**...

...but not into the **future**

Easy solution:

$$e_{l,t} = q_l \cdot k_t$$

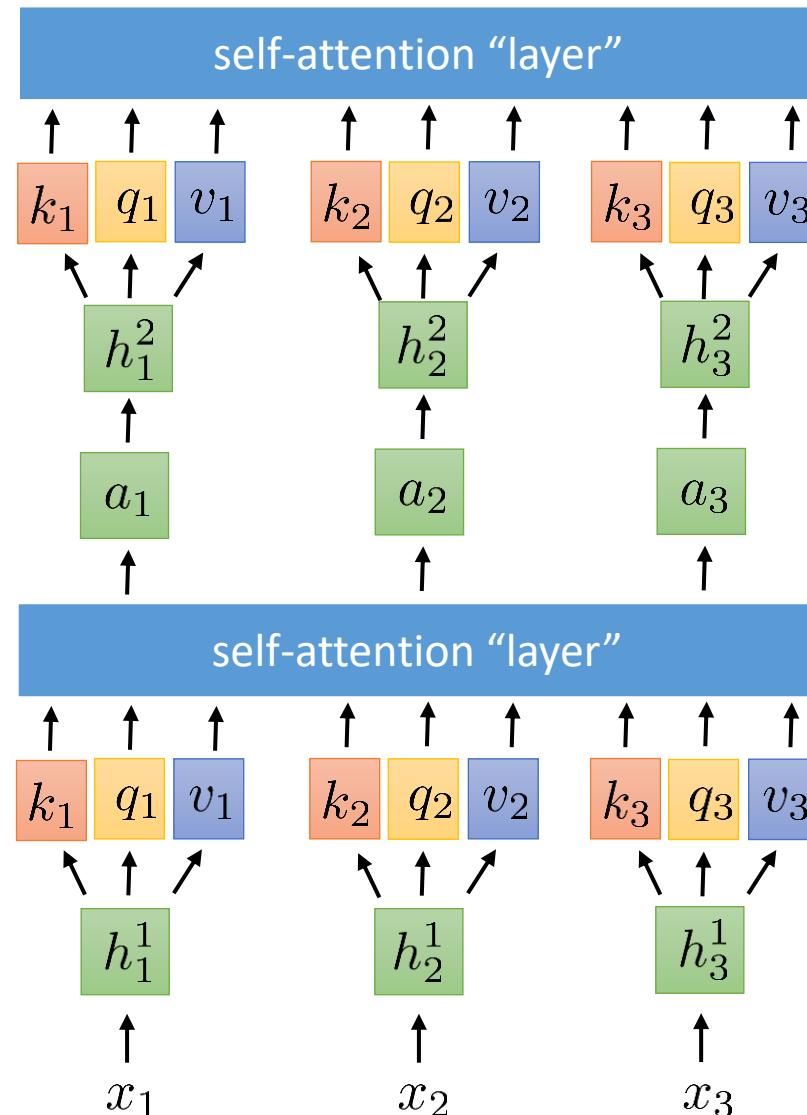
$$e_{l,t} = \begin{cases} q_l \cdot k_t & \text{if } l \geq t \\ -\infty & \text{otherwise} \end{cases}$$

in practice:

just replace $\exp(e_{l,t})$ with 0 if $l < t$

inside the softmax

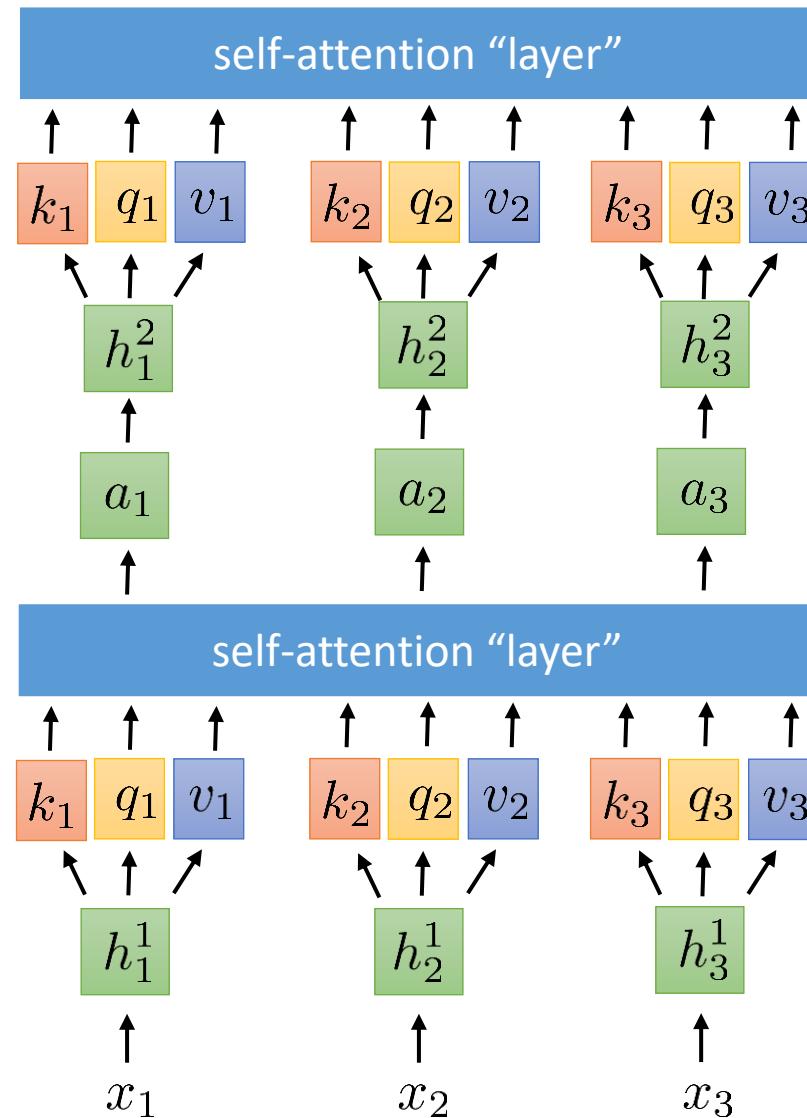
Implementation summary



- We can implement a **practical** sequence model based **entirely** on self-attention
- Alternate self-attention "layers" with nonlinear position-wise feedforward networks (to get nonlinear transformations)
- Use positional encoding (on the input or input embedding) to make the model aware of relative positions of tokens
- Use multi-head attention
- Use masked attention if you want to use the model for decoding

The Transformer

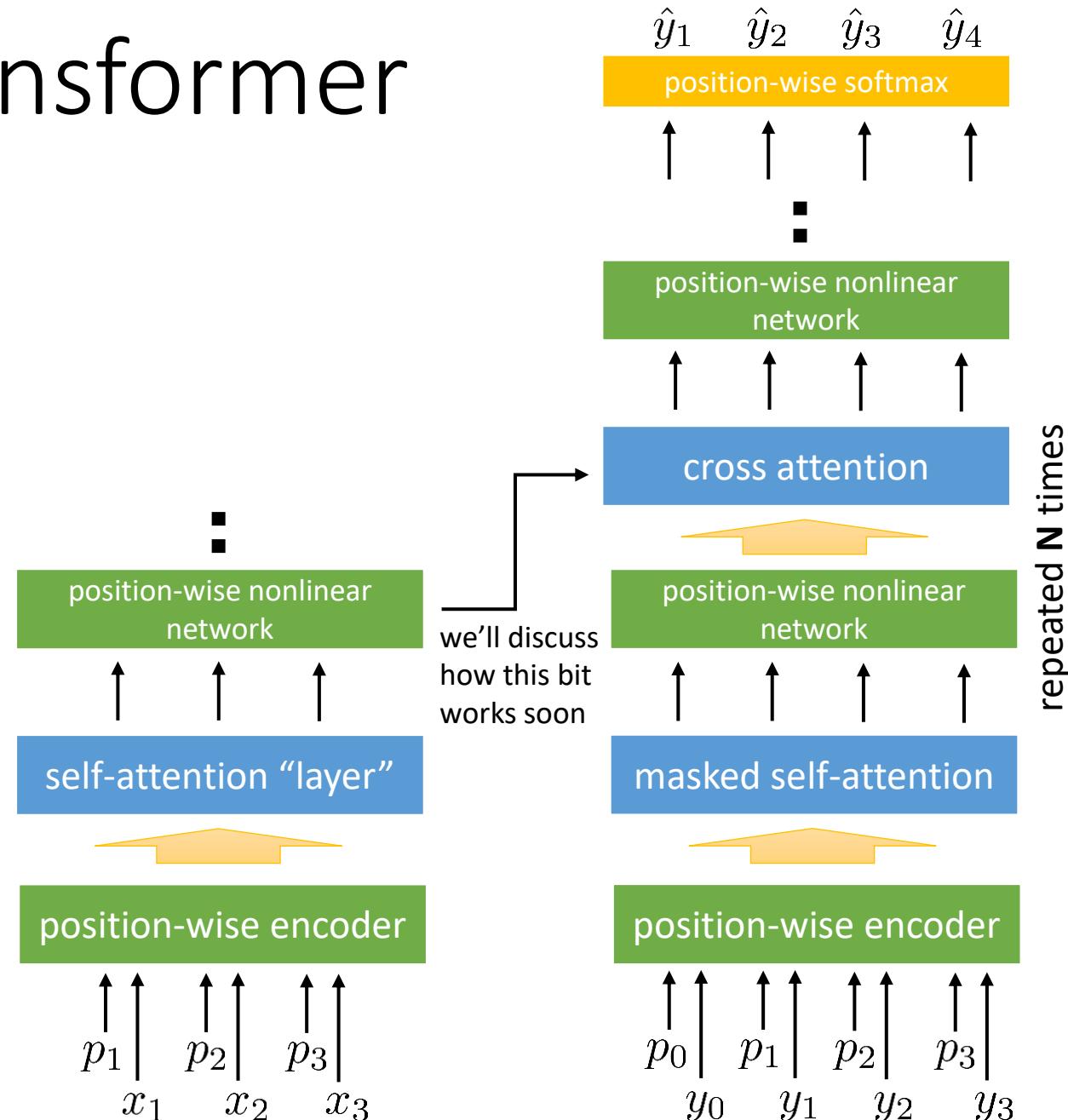
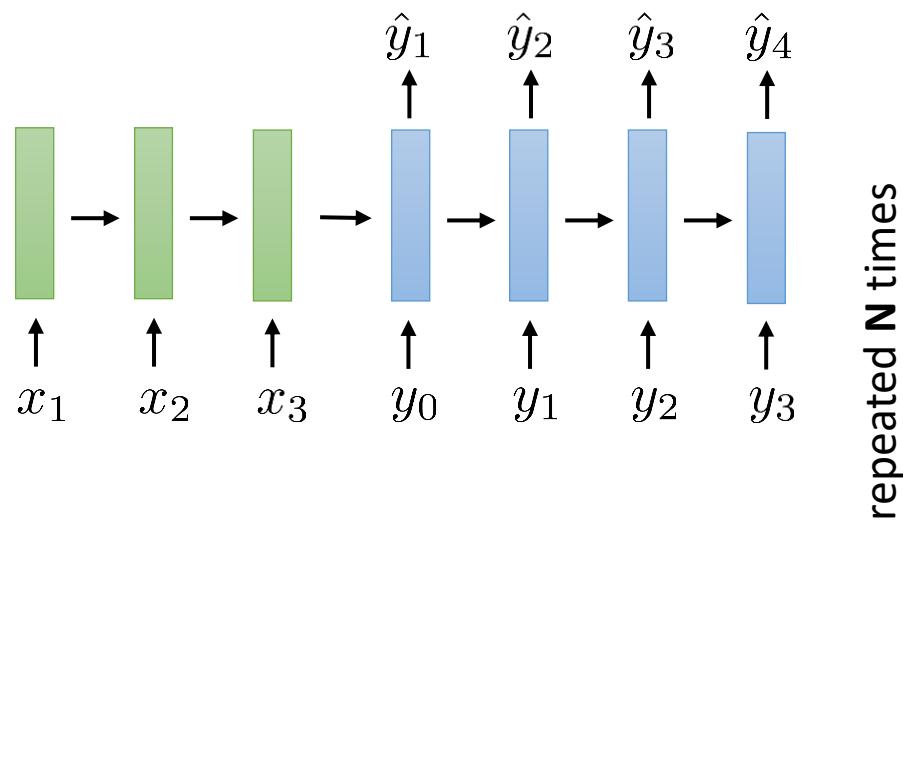
Sequence to sequence with self-attention



- There are a number of model designs that use successive self-attention and position-wise nonlinear layers to process sequences
- These are generally called “Transformers” because they transform one sequence into another at **each** layer
 - See Vaswani et al. **Attention Is All You Need.** 2017
- The “classic” transformer (Vaswani et al. 2017) is a **sequence to sequence** model
- A number of well-known follow works also use transformers for language modeling (BERT, GPT, etc.)

The “classic” transformer

As compared to a sequence
to sequence RNN model



Combining encoder and decoder values

“Cross-attention”

Much more like the **standard** attention
from the previous lecture

$$\text{query: } q_l^\ell = W_q^\ell s_l^\ell \quad \begin{matrix} \text{output of position-wise nonlinear network} \\ \text{at (decoder) layer } \ell, \text{ step } l \end{matrix}$$

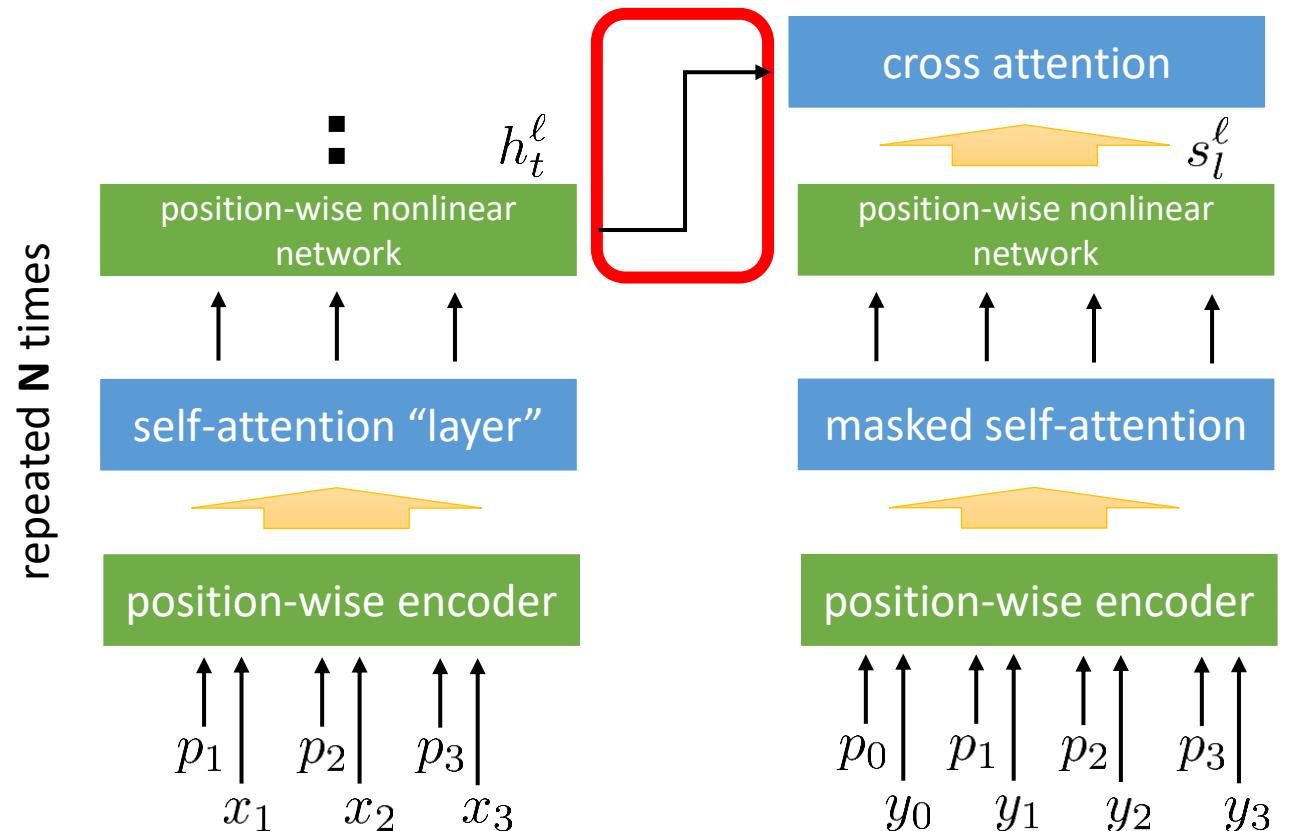
$$\text{key: } k_t^\ell = W_k^\ell h_t^\ell \quad \begin{matrix} \text{output of position-wise nonlinear network} \\ \text{at (encoder) layer } \ell, \text{ step } t \end{matrix}$$

$$\text{value: } v_t^\ell = W_k^\ell h_t^\ell$$

$$e_{l,t}^\ell = q_l^\ell \cdot k_t^\ell$$

$$\alpha_{l,t}^\ell = \frac{\exp(e_{l,t}^\ell)}{\sum_{t'} \exp(e_{l,t'}^\ell)}$$

$$c_l^\ell = \sum_t \alpha_{l,t}^\ell v_t^\ell \quad \text{cross attention output}$$



in reality, cross-attention is
also multi-headed!

One last detail: layer normalization

Main idea: batch normalization is very helpful, but hard to use with sequence models

Sequences are different lengths, makes normalizing across the batch hard

Sequences can be very long, so we sometimes have small batches

Simple solution: “layer normalization” – like batch norm, but not across the batch

<p>Batch norm</p> <p>$d\text{-dim}$</p> $\mu = \frac{1}{B} \sum_{i=1}^B a_i$ $\sigma = \sqrt{\frac{1}{B} \sum_{i=1}^B (a_i - \mu)^2}$ $\bar{a}_i = \frac{a_i - \mu}{\sigma} \gamma + \beta$	<p>$d\text{-dimensional vectors}$</p> <p>for each sample in batch</p> <p>Layer norm</p> <p>a</p> <p>1-dim</p> $\mu = \frac{1}{d} \sum_{j=1}^d a_j$ $\sigma = \sqrt{\frac{1}{d} \sum_{j=1}^d (a_j - \mu)^2}$ $\bar{a} = \frac{a - \mu}{\sigma} \gamma + \beta$
---	--

Putting it all together

The Transformer

multi-head attention keys and values
 $k_{t,1}^\ell, \dots, k_{t,m}^\ell$ and $v_{t,1}^\ell, \dots, v_{t,m}^\ell$
6 layers, each with $d = 512$

$$\bar{h}_t^\ell = \text{LayerNorm}(\bar{a}_t^\ell + h_t^\ell)$$

passed to next layer $\ell + 1$

$$h_t^\ell = W_2^\ell \text{ReLU}(W_1^\ell \bar{a}_t^\ell + b_1^\ell) + b_2^\ell$$

2-layer neural net at each position

$$\bar{a}_t^\ell = \text{LayerNorm}(\bar{h}_t^{\ell-1} + a_t^\ell)$$

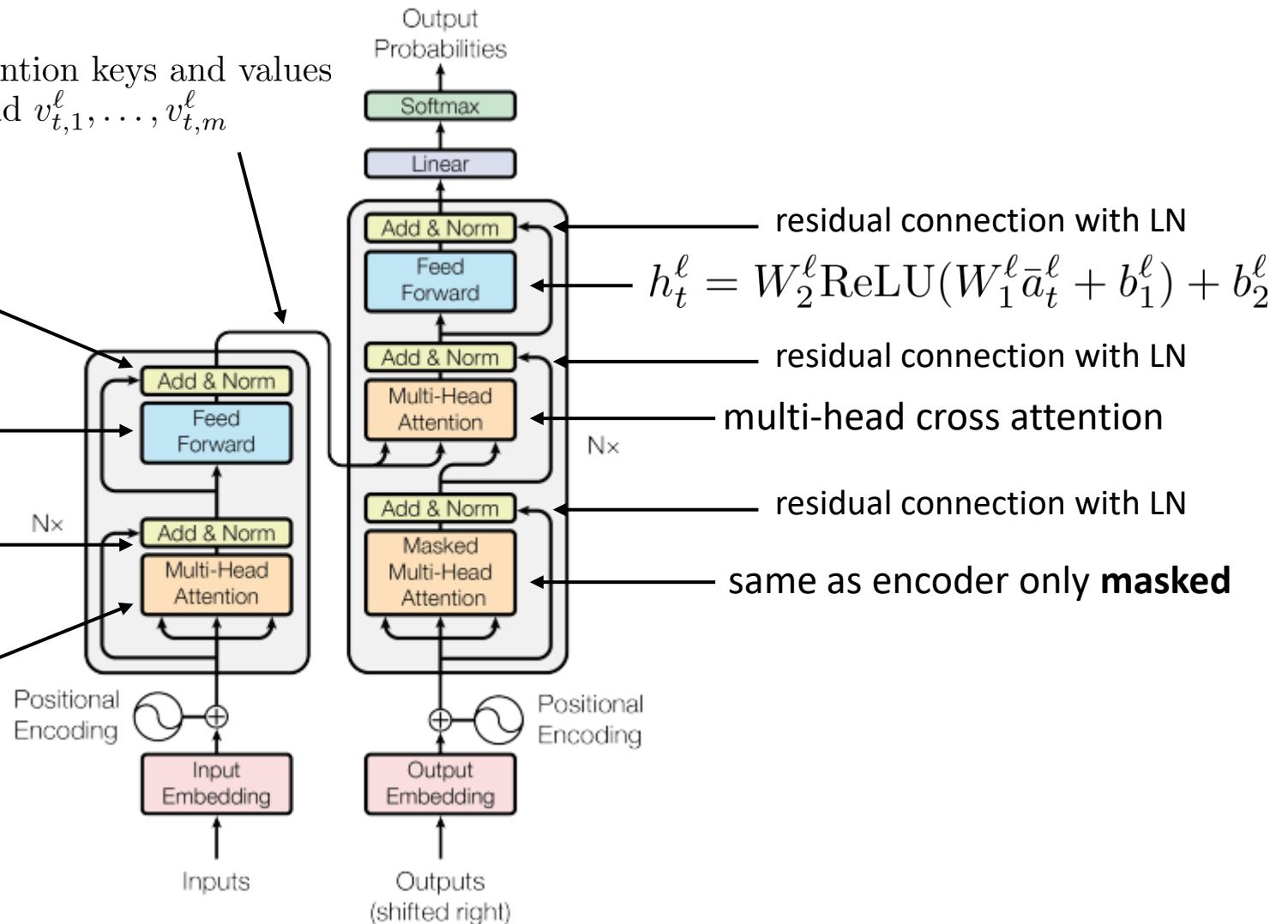
essentially a residual connection with LN

input: $\bar{h}_t^{\ell-1}$

output: a_t^ℓ

concatenates attention from all heads

Decoder decodes one position at a time with masked attention



Why transformers?

Downsides:

- Attention computations are technically $O(n^2)$
- Somewhat more complex to implement (positional encodings, etc.)

Benefits:

- + Much better long-range connections
- + Much easier to parallelize
- + In practice, can make it much deeper (more layers) than RNN

The benefits seem to **vastly** outweigh the downsides, and transformers work **much** better than RNNs (and LSTMs) in many cases

Arguably one of the most important sequence modeling improvements of the past decade

Why transformers?

In practice, this means we can use larger models for the same cost

larger model = better performance

much faster training

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	41.29	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$3.3 \cdot 10^{18}$	
Transformer (big)	28.4	41.8	$2.3 \cdot 10^{19}$	

great translation results

Text summarization

previous state of the art seq2seq model

Model	Test perplexity	ROUGE-L
seq2seq-attention, $L = 500$	5.04952	12.7
Transformer-ED, $L = 500$	2.46645	34.2
Transformer-D, $L = 4000$	2.22216	33.6
Transformer-DMCA, no MoE-layer, $L = 11000$	2.05159	36.2
Transformer-DMCA, MoE-128, $L = 11000$	1.92871	37.9
Transformer-DMCA, MoE-256, $L = 7500$	1.90325	38.8

lower is better (this metric is similar to 1/likelihood)

We'll learn more about the power of transformers as **language models** next time!

Applications: NLP

Designing, Visualizing and Understanding Deep Neural Networks

CS W182/282A

Instructor: Sergey Levine
UC Berkeley



The Big Idea: Unsupervised Pretraining

Deep learning works best when we have a lot of data



The big challenge: how can we use **freely available** and **unlabeled** text data to help us apply deep learning methods to NLP?

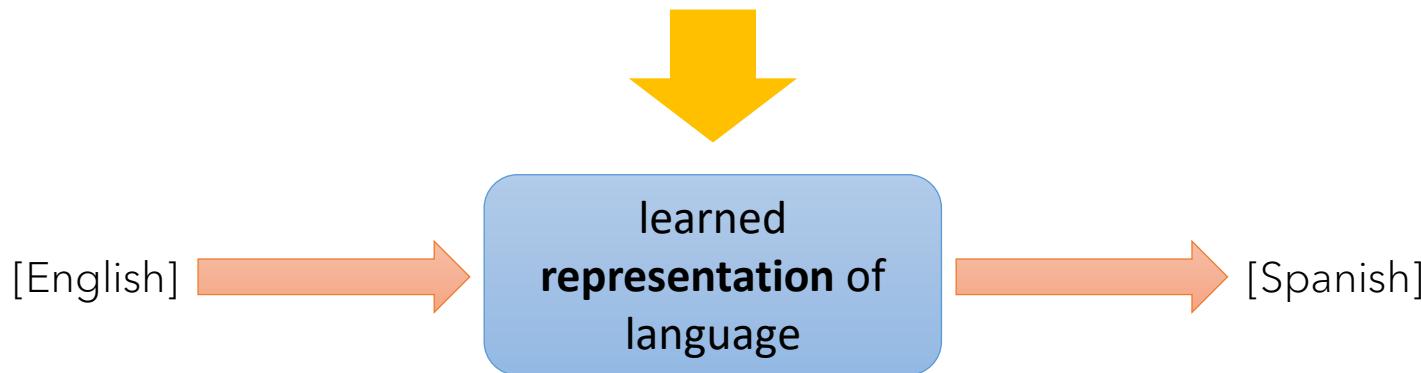
Good news: there is plenty of data of text out there!



Bad news: most of it is unlabeled

1,000s of times more data **without** labels (i.e., valid English text in books, news, web) vs. labeled/paired data (e.g., English/French translations)

What can we do with unlabeled data?



How can we represent these... representations?

local non-contextual
representations
word embeddings

global context-dependent
representations
pretrained language models

sentence embeddings

Start simple: how do we represent words?

$$x_{1,1} = \begin{bmatrix} 0 \\ 0 \\ \cdot \\ 0 \\ 1 \\ 0 \\ \cdot \\ 0 \end{bmatrix}$$

dimensionality = number of possible words
index of this word
not great, not terrible...

This means basically nothing by itself

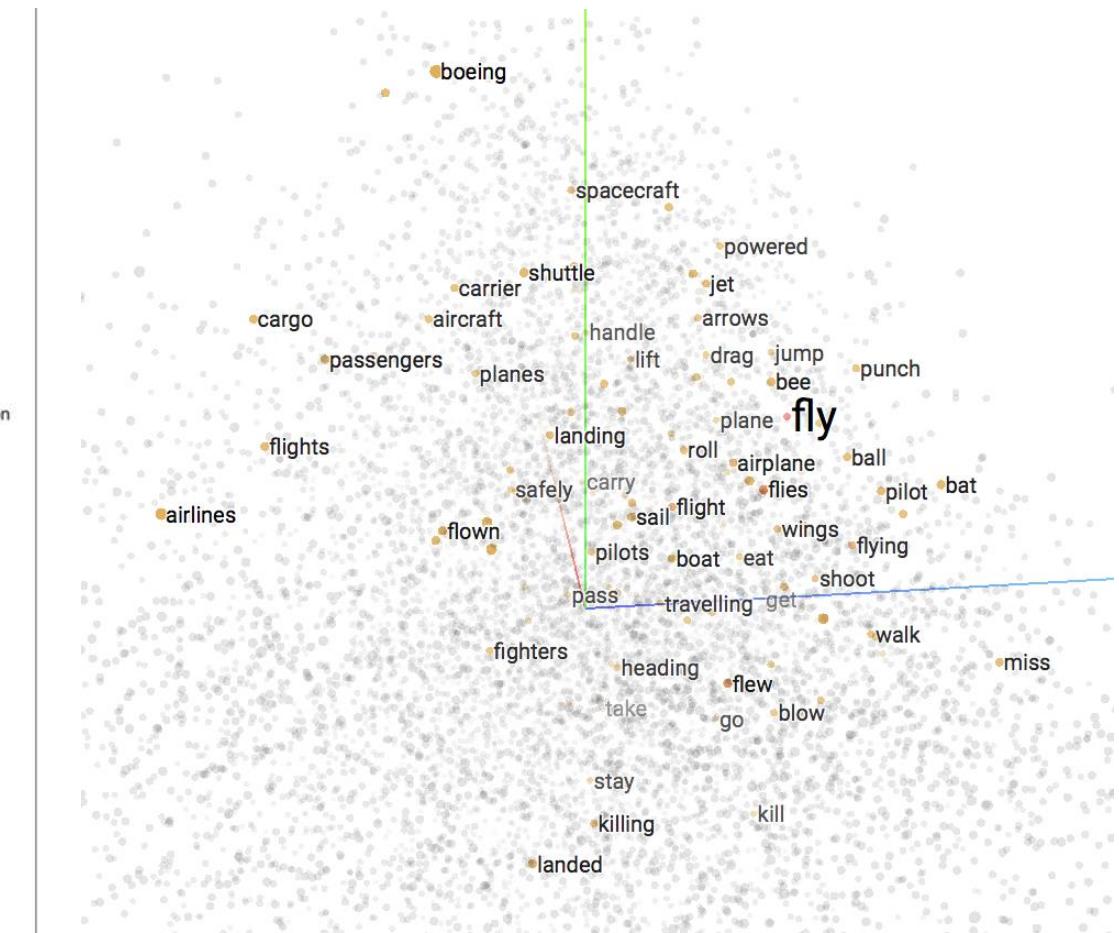
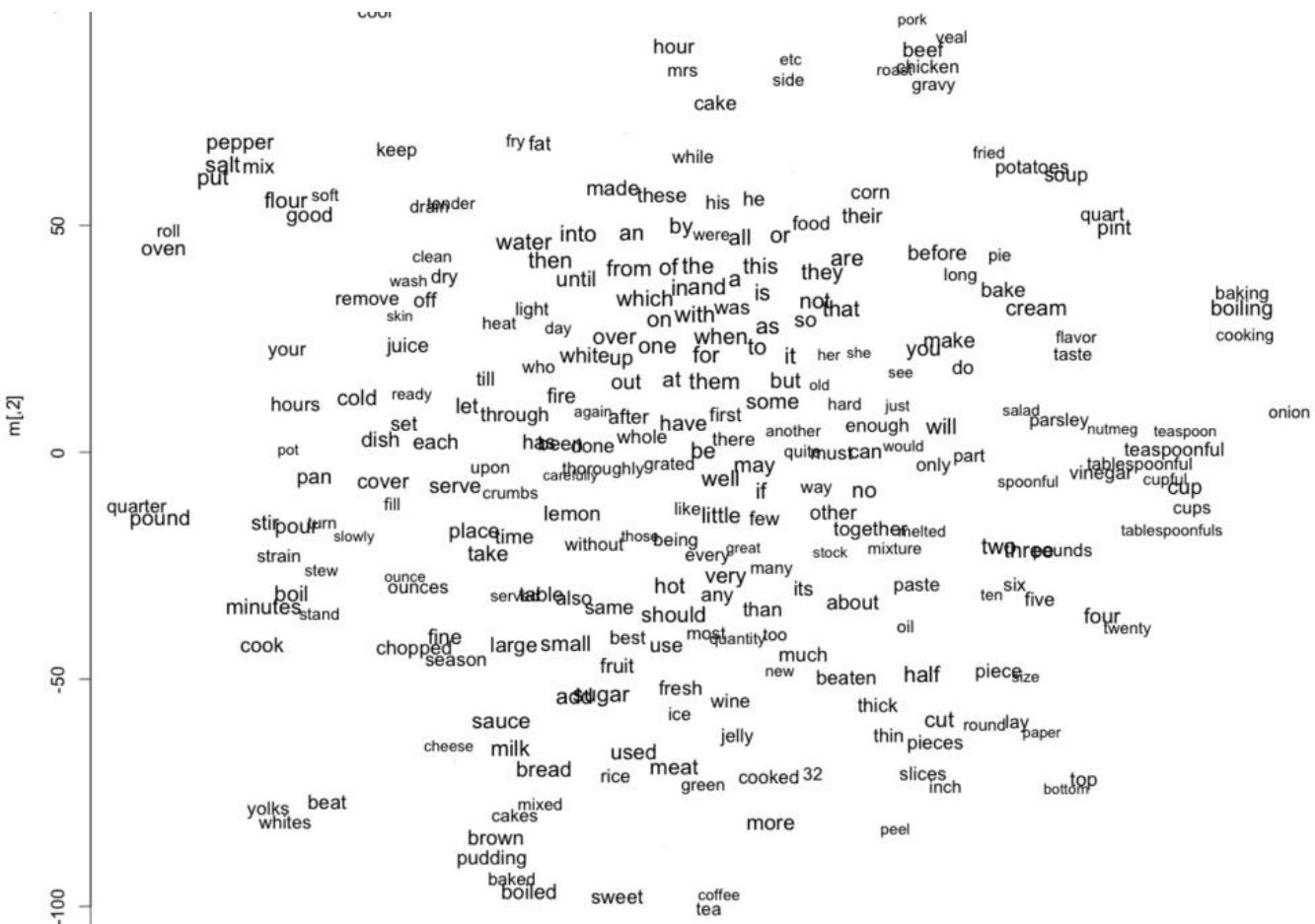


The pixels mean something!
Not much (not a great metric space), but they mean something

Maybe if we had a more meaningful representation of words, then learning downstream tasks would be much easier!

Meaningful = vectors corresponding to **similar** words should be close together

Some examples of good embeddings



How do we learn embeddings?

Basic idea: the meaning of a word is determined by what **other** words occur in close proximity to it

Put another way: the more interchangeable words are, the more similar they are

Example: Seattle hotel vs. Seattle motel

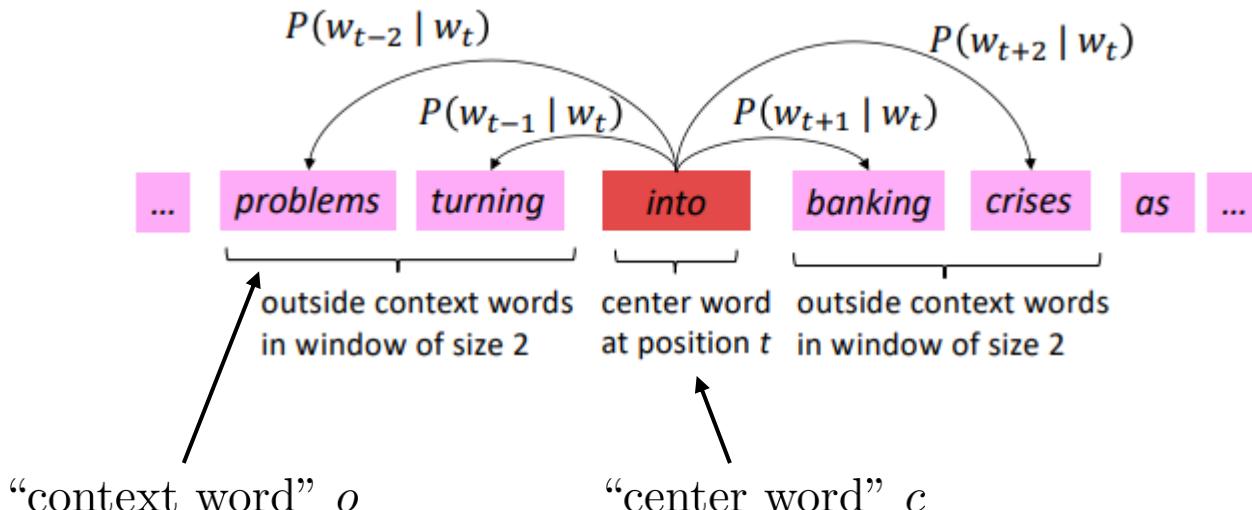
*...government debt problems turning into **banking** crises as happened in 2009...*
*...saying that Europe needs unified **banking** regulation to replace the hodgepodge...*
*...India has just given its **banking** system a shot in the arm...*

These **context words** will represent **banking**

Basic principle: pick a representation for each word such that its neighbors are “close” under this representation

More formally...

Can we **predict** the neighbors of a word from its **embedding value**?



how to train?

$$\arg \max_{u_1, \dots, u_n, v_1, \dots, v_n} \sum_{c,o} \log p(o|c)$$

u and v vectors for all possible words

all possible c and o combinations
e.g., for each word c , pick all words o that are within 5 step

$$p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

(learned) vector representation of o

(learned) vector representation of c

all possible words (vocabulary)

looks a bit like a **logistic regression** model

word2vec

$$\arg \max_{u_1, \dots, u_n, v_1, \dots, v_n} \sum_{c,o} \log p(o|c) \quad p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

$$\theta = \begin{bmatrix} v_{\text{aardvark}} \\ v_a \\ \dots \\ v_{\text{zebra}} \\ u_{\text{aardvark}} \\ u_a \\ \dots \\ u_{\text{zebra}} \end{bmatrix}$$

- Why two vectors? Makes optimization easier
- What to do at the end? Average them
- This then gives us a representation of words!

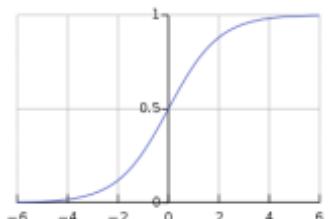
Making word2vec tractable

$$\arg \max_{u_1, \dots, u_n, v_1, \dots, v_n} \sum_{c,o} \log p(o|c) \quad p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

Problem: the vocabulary might be **huge**

denominator might be **really costly** to compute

Another idea: what if we instead have a **binary** classification problem (“is this the right word or not”)?



$$p(o \text{ is the right word}|c) = \sigma(u_o^T v_c) = \frac{1}{1 + \exp(-u_o^T v_c)}$$

This is not enough! Why?

$$p(o \text{ is the wrong word}|c) = \sigma(-u_o^T v_c) = \frac{1}{1 + \exp(u_o^T v_c)}$$

$$\arg \max_{u_1, \dots, u_n, v_1, \dots, v_n} \sum_{c,o} \left(\log p(o \text{ is right}|c) + \sum_w \log p(w \text{ is wrong}|c) \right)$$

randomly chosen “negatives”

Making word2vec tractable: summary

$$p(o \text{ is the right word}|c) = \sigma(u_o^T v_c) = \frac{1}{1 + \exp(-u_o^T v_c)}$$

$$p(o \text{ is the wrong word}|c) = \sigma(-u_o^T v_c) = \frac{1}{1 + \exp(u_o^T v_c)}$$

$$\arg \max_{u_1, \dots, u_n, v_1, \dots, v_n} \sum_{c,o} \left(\log p(o \text{ is right}|c) + \sum_w \log p(w \text{ is wrong}|c) \right)$$

$$\arg \max_{u_1, \dots, u_n, v_1, \dots, v_n} \sum_{c,o} \left(\log \sigma(u_o^T v_c) + \sum_w \log \sigma(-u_w^T v_c) \right)$$

Intuition: push v_c toward u_o and away from other vectors u_w

word2vec examples

$$\arg \max_{u_1, \dots, u_n, v_1, \dots, v_n} \sum_{c,o} \log p(o|c)$$

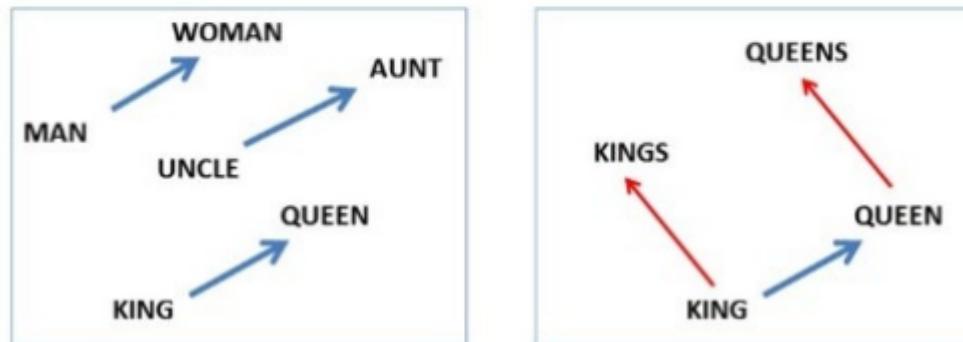
$$p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

Algebraic relations:

$$\text{vec("woman")}-\text{vec("man")} \simeq \text{vec("aunt")}-\text{vec("uncle")}$$

$$\text{vec("woman")}-\text{vec("man")} \simeq \text{vec("queen")}-\text{vec("king")}$$

This is a little bit idealized, most relationships are not nearly this "nice"



word2vec examples

$$\arg \max_{u_1, \dots, u_n, v_1, \dots, v_n} \sum_{c,o} \log p(o|c)$$

$$p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

Word2vec model computed from 6 billion word corpus of news articles

Type of relationship	Word Pair 1		Word Pair 2	
Common capital city	Athens	Greece	Oslo	Norway
All capital cities	Astana	Kazakhstan	Harare	Zimbabwe
Currency	Angola	kwanza	Iran	rial
City-in-state	Chicago	Illinois	Stockton	California
Man-Woman	brother	sister	grandson	granddaughter
Adjective to adverb	apparent	apparently	rapid	rapidly
Opposite	possibly	impossibly	ethical	unethical
Comparative	great	greater	tough	tougher
Superlative	easy	easiest	lucky	luckiest
Present Participle	think	thinking	read	reading
Nationality adjective	Switzerland	Swiss	Cambodia	Cambodian
Past tense	walking	walked	swimming	swam
Plural nouns	mouse	mice	dollar	dollars
Plural verbs	work	works	speak	speaks

word2vec examples

$$\arg \max_{u_1, \dots, u_n, v_1, \dots, v_n} \sum_{c,o} \log p(o|c)$$

$$p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Word2vec summary

$$\arg \max_{u_1, \dots, u_n, v_1, \dots, v_n} \sum_{c,o} \log p(o|c) \quad p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

What do we do with this?

- Use as word representation in place of one-hot vectors
- Much more meaningful for downstream applications
- Can train word embeddings on large unlabeled corpus, and then use it as input into a supervised model trained on a much smaller corpus
- Could think of it as a simple type of **unsupervised pretraining**

Pretrained Language Models

Contextual representations

Word embeddings associate a vector with each word

This can make for a much **better** representation than just a one-hot vector!

However, the vector **does not** change if the word is used in different ways!

Let's play baseball

I saw a play yesterday

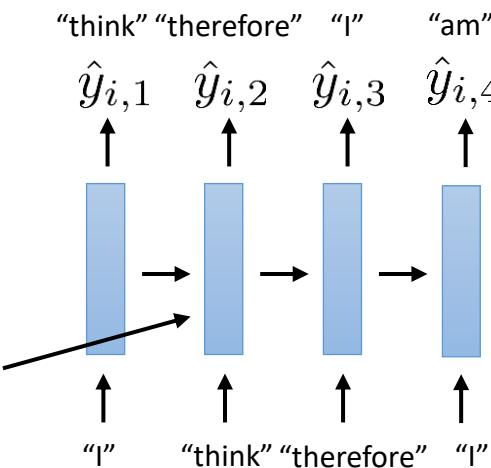
same word2vec representation, even though they mean different things!

Can we learn representations that **depend on context**?

High level idea:

1. Train a **language model**
2. Run it on a sentence
3. Use its **hidden state**

use the hidden state as
the representation for
downstream tasks



Question 1: how to train the best language model for this?

Question 2: how to use this language model for downstream tasks?

The age of Sesame Street characters



ELMo: bidirectional LSTM model used for context-dependent embeddings



BERT: transformer language model used for context-dependent embeddings



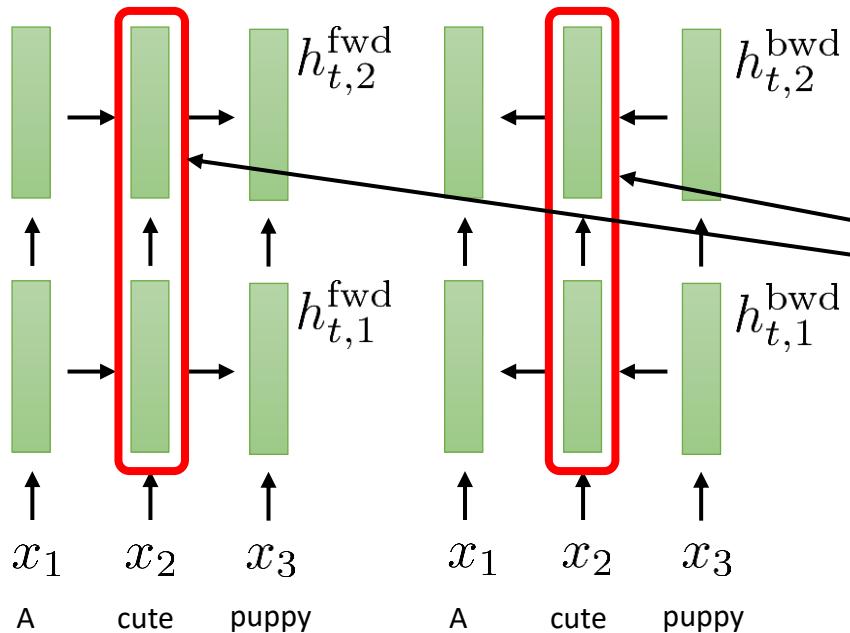
Credit: Jay Alammar: <http://jalammar.github.io/illustrated-bert/>

ELMo

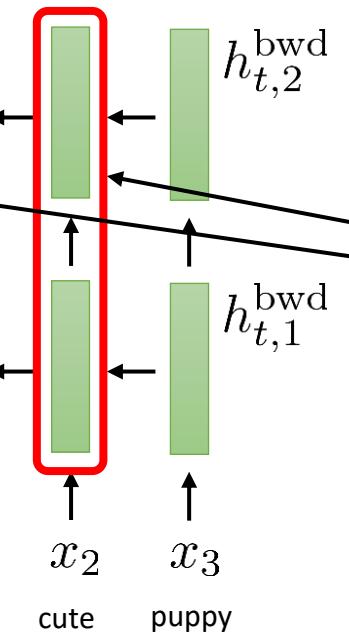


Both the forward and backward LM are trained as language models

Forward LM



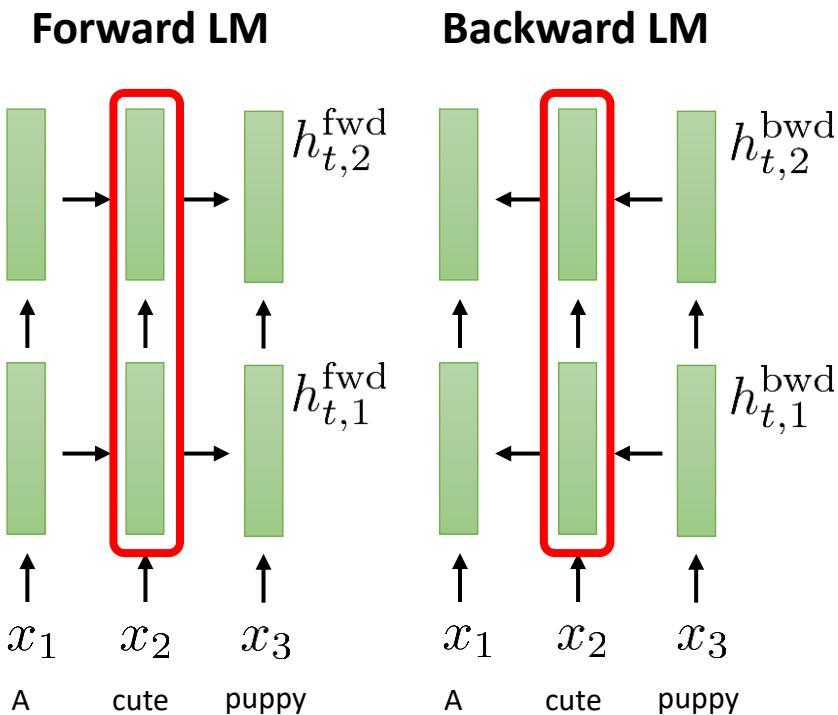
Backward LM



Predict the next (or previous) word

together, all these hidden states form
a representation of the word “cute”

Using ELMo



simple version: $\text{ELMO}_t = [h_{t,2}^{\text{fwd}}, h_{t,2}^{\text{bwd}}]$

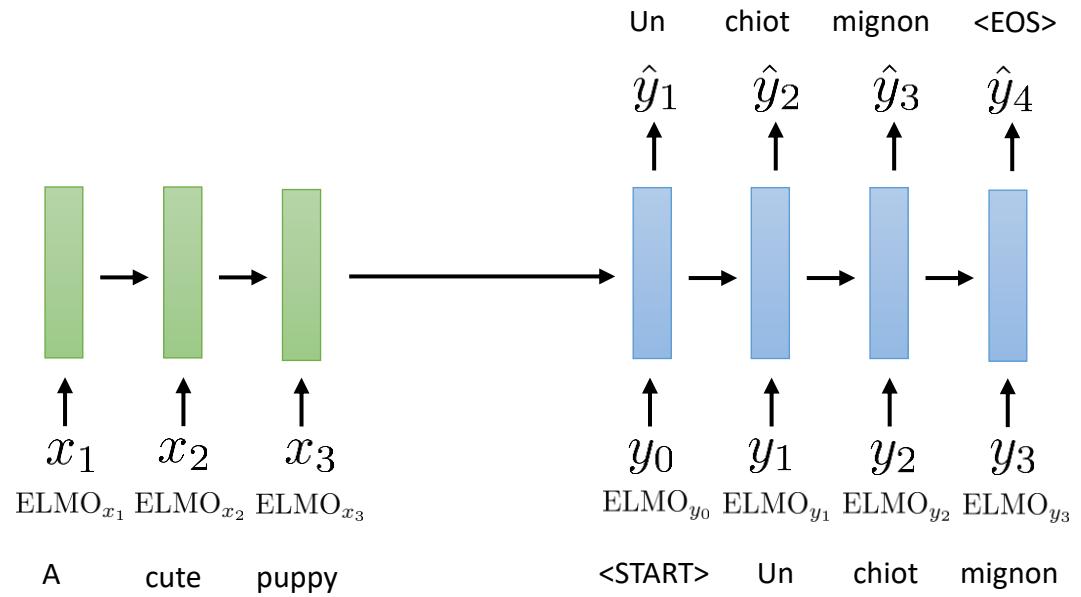
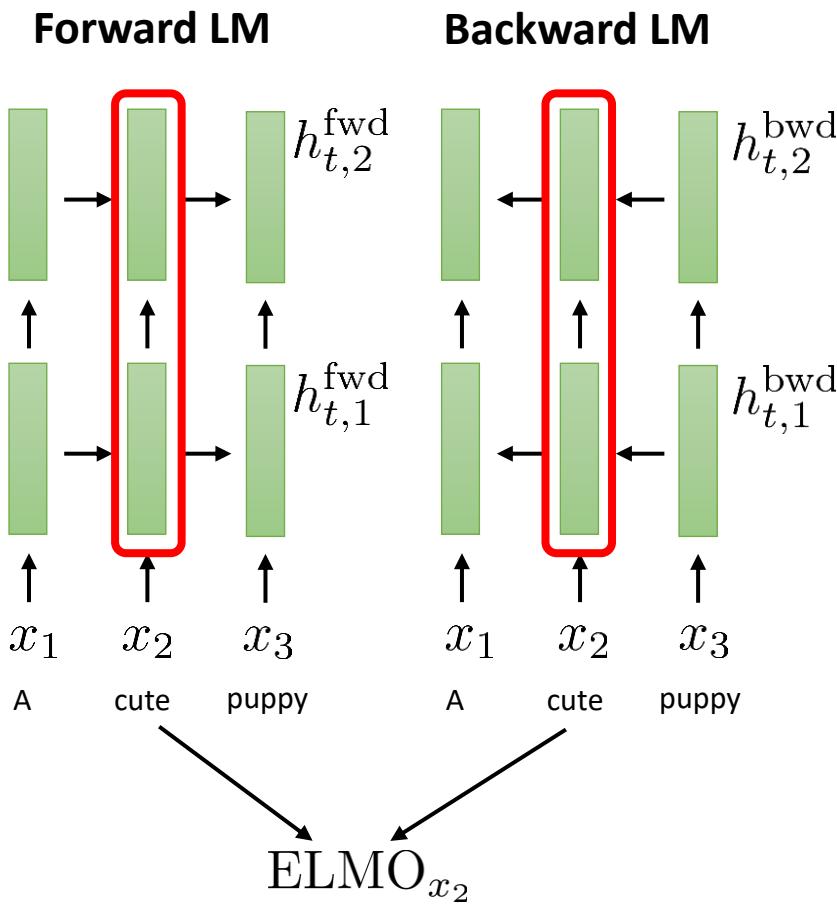
complex version: $\text{ELMO}_t = \gamma \sum_{i=1}^L w_i [h_{t,i}^{\text{fwd}}, h_{t,i}^{\text{bwd}}]$

top layer hidden states

learned task-specific weights

learned as part of **downstream** task!

Using ELMo

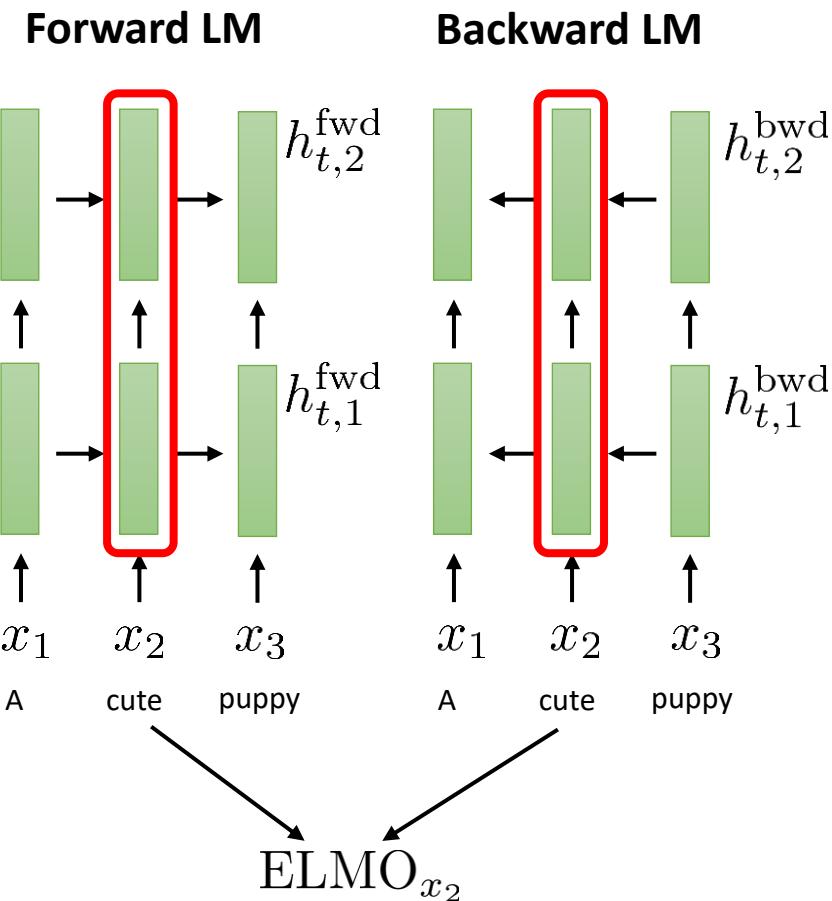


This is just an example, the actual ELMo paper **does not** test on translation, but does test:

- Question answering
- Textual entailment
- Semantic role labeling
- Coreference resolution
- Named entity extraction
- Sentiment analysis

And LMs help with all of these!

ELMo Summary



- Train **forward** and **backward** language models on a large corpus of **unlabeled** text data
- Use the (concatenated) forward and backward LSTM states to represent the word **in context**
- Concatenate the **ELMo representation** to the word embedding (or one-hot vector) as an **input** into a downstream task-specific sequence model
 - This provides a **context specific and semantically meaningful** representation of each token

BERT and Friends

The age of Sesame Street characters



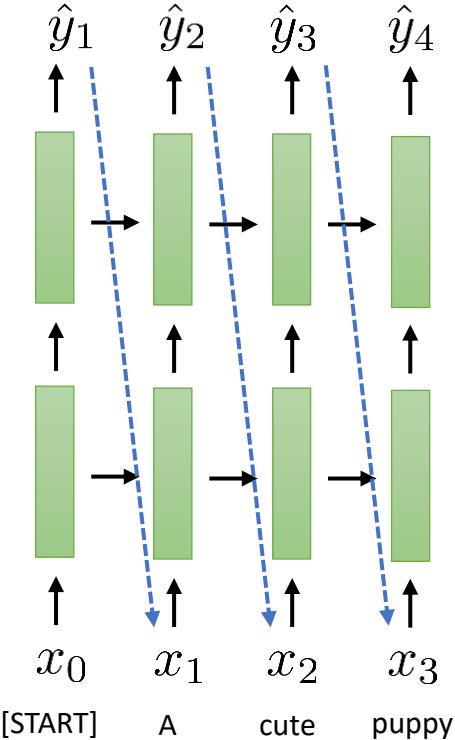
ELMo: bidirectional LSTM model used for context-dependent embeddings



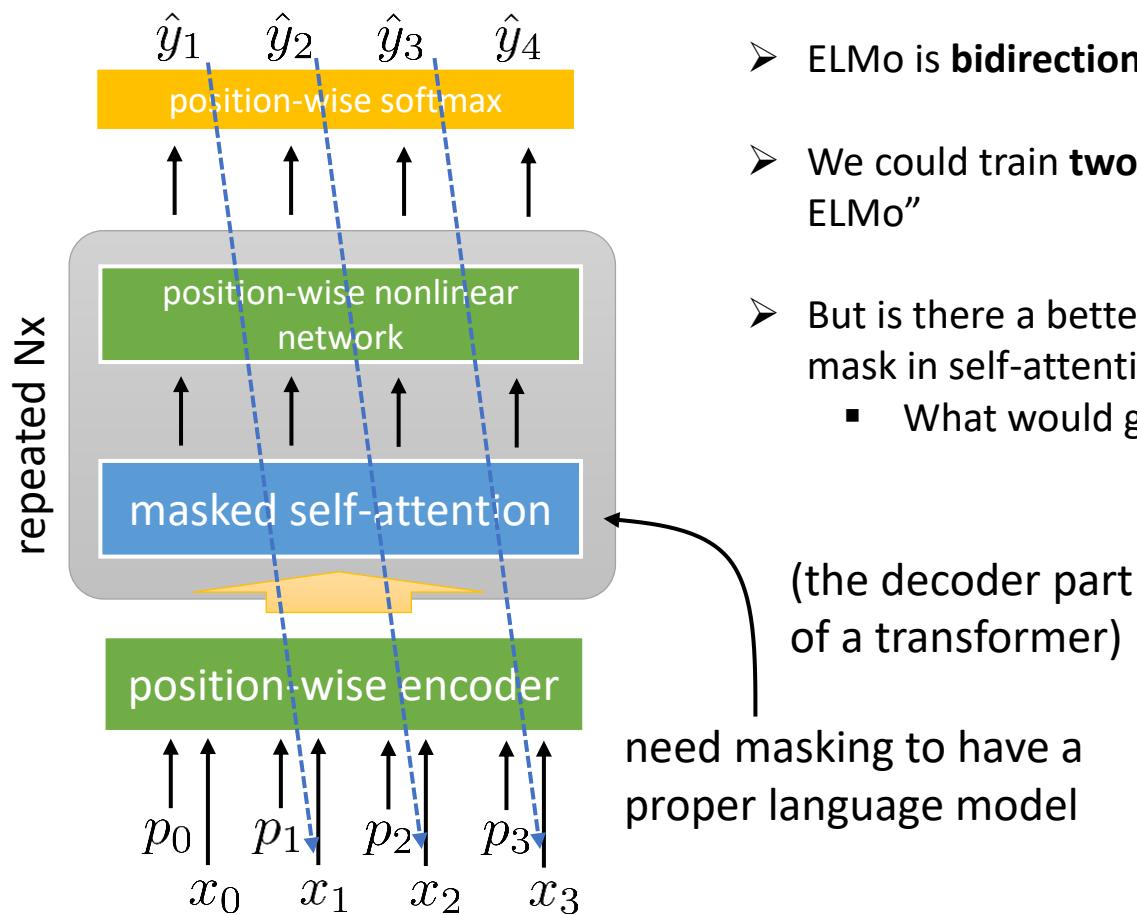
BERT: transformer language model used for context-dependent embeddings

Can we use a transformer instead?

Before:

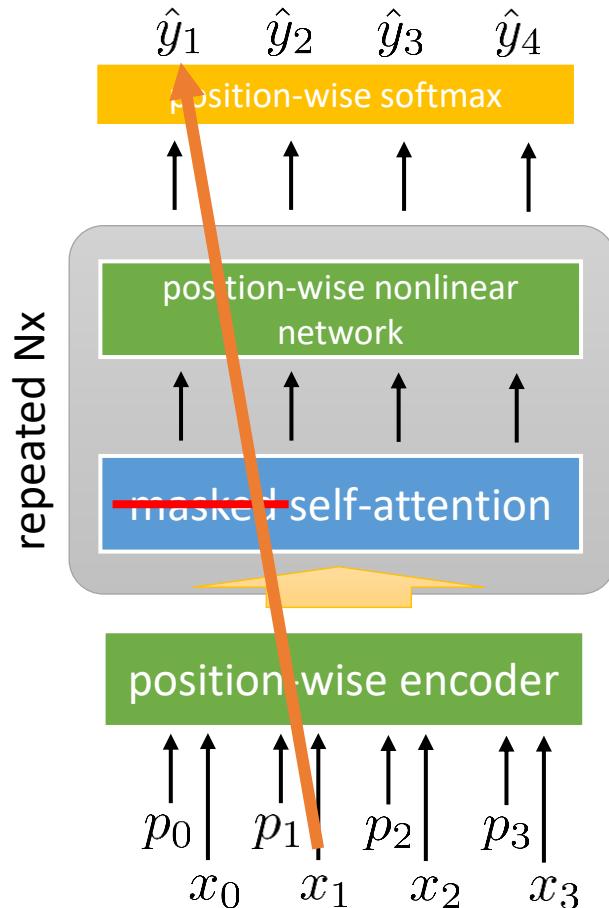


Now:



- This model has a **direction** (forward, depends on masking used in self-attention)
- ELMo is **bidirectional**, this isn't
- We could train **two transformers**, and make "transformer ELMo"
- But is there a better way? Can we simply remove the mask in self-attention and have **one** transformer?
 - What would go wrong?

Bidirectional transformer LMs

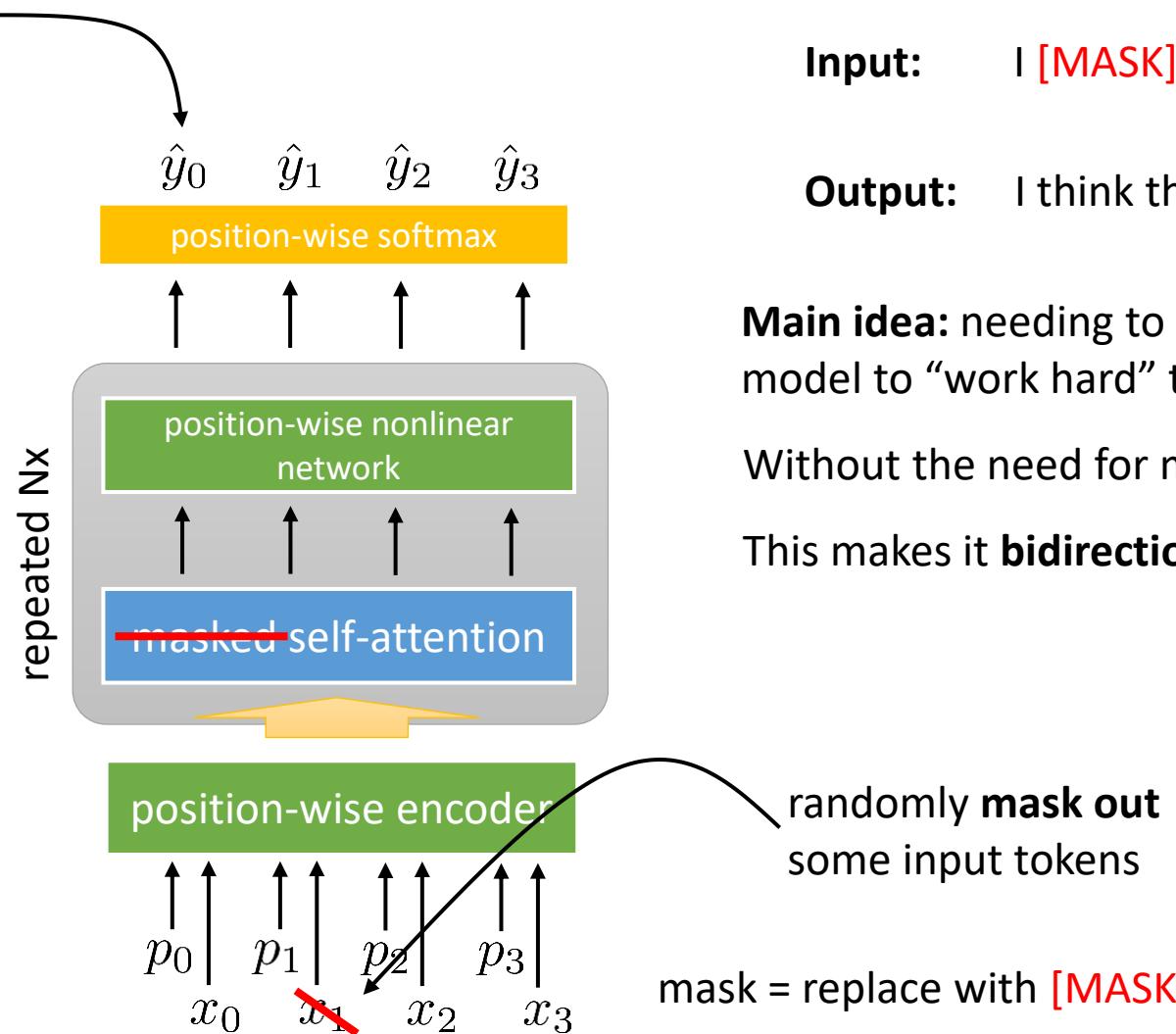


It's trivially easy to get the right answer, since self-attention can access the “right answer” at time **t** from the input at time **t+1**!

Bidirectional transformer LMs



no need to shift things by one anymore (no masking)



BERT is essentially the “encoder” part of a transformer with 15% of inputs replaced with [MASK]

Input: I [MASK] therefore I [MASK]

Output: I think therefore I am

Main idea: needing to predict missing words forces the model to “work hard” to learn a good representation

Without the need for masked self-attention!

This makes it **bidirectional**

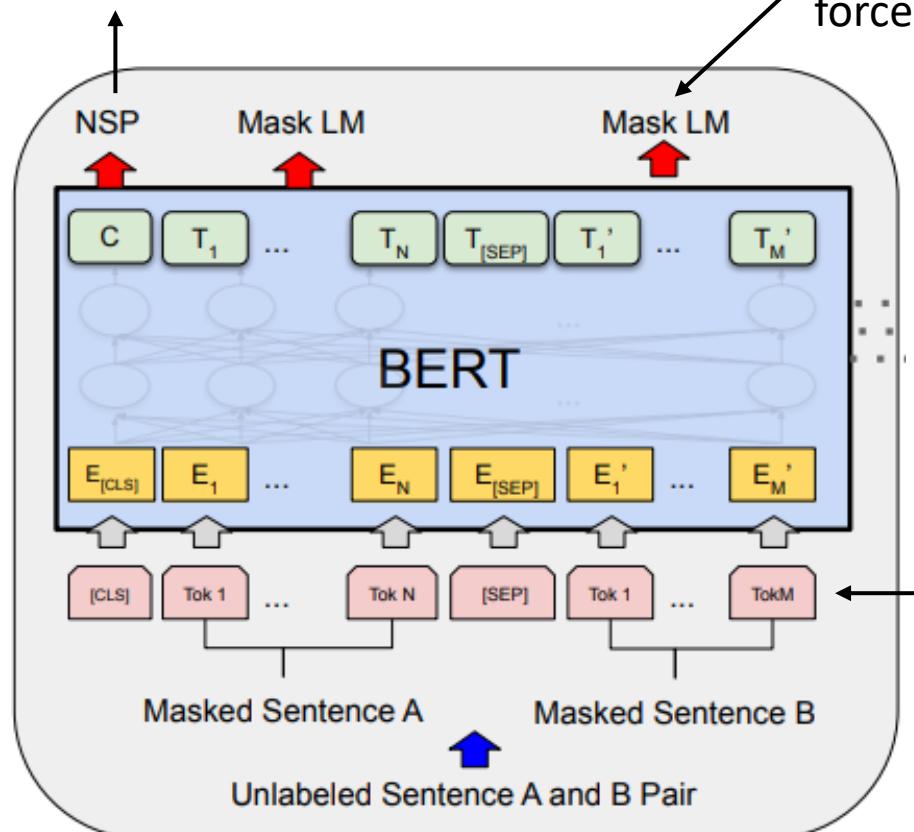
Training BERT



binary classifier output:

does **first sentence** follow the
second sentence, or precede it?

this forces
learning
sentence-level
representations



reconstruct all tokens at each time step
(must predict actual token in place of [MASK])

forces learning context-dependent **word-level** representations

pairs of sentences in the data
are transformed in two ways:

1. Randomly replace 15% of the tokens with [MASK]
2. Randomly swap the order of the sentences 50% of the time

input consists of **two** sentences
why?

many downstream tasks require
processing two sentences:
question answering
natural language inference

Using BERT



binary classifier output:

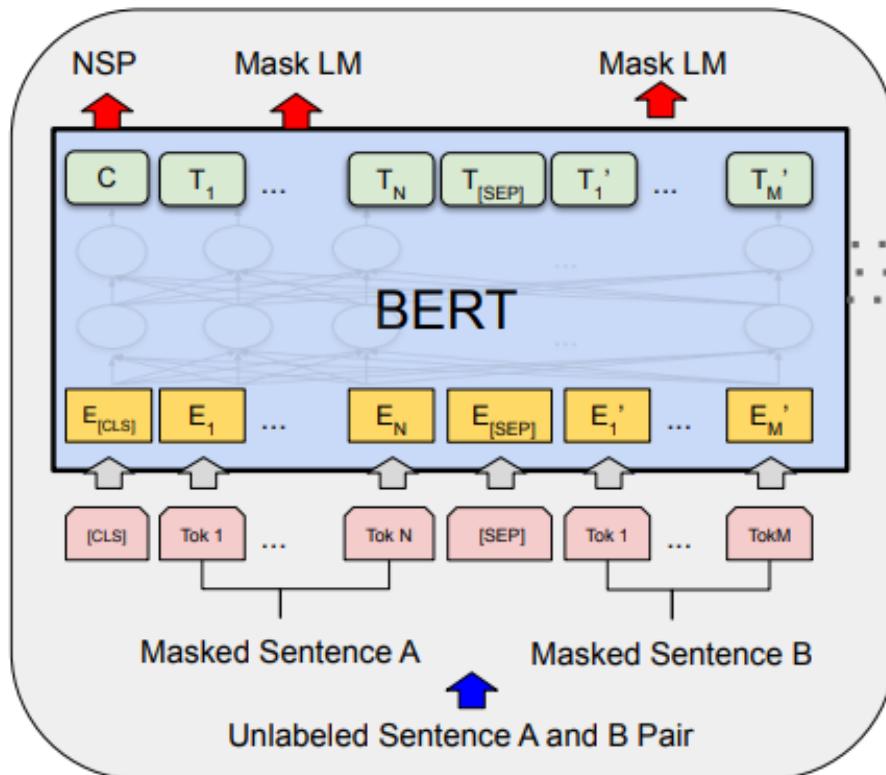
~~A before B vs. A after B~~

task classification output

entailment classification

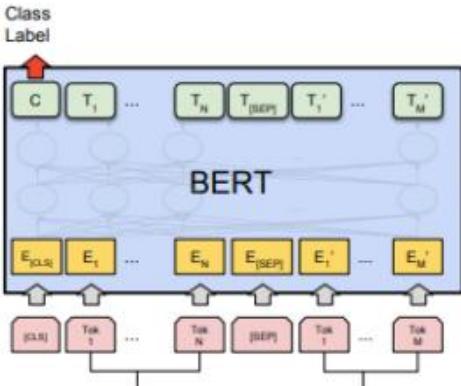
semantic equivalence (e.g., Quora question pair)

sentiment classification

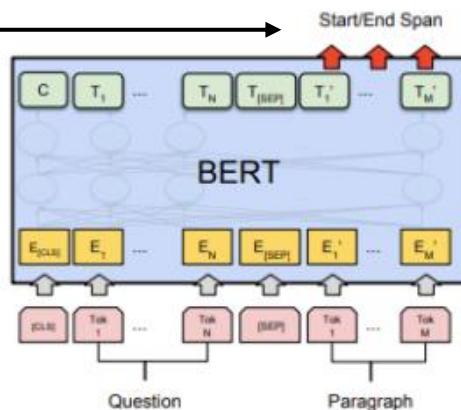


1. Put a cross-entropy loss on **only** the first output (replaces the sentence order classifier)
2. Finetune the **whole** model end-to-end on the new task

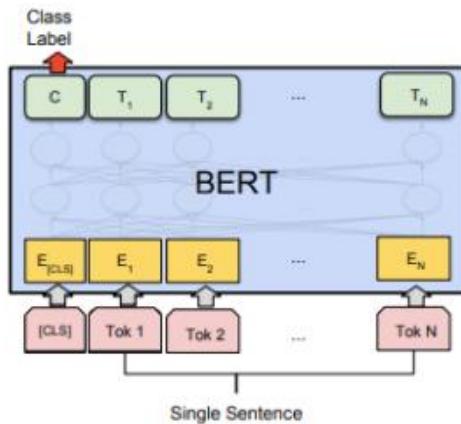
Using BERT



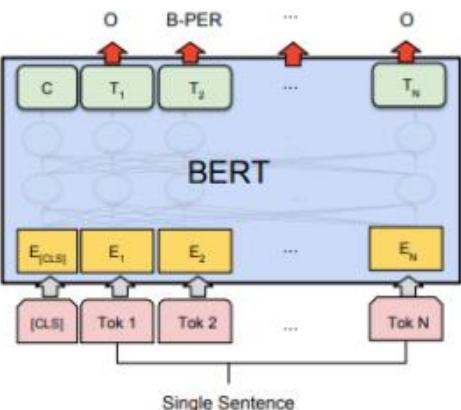
(a) Sentence Pair Classification Tasks:
MNLI, QQP, QNLI, STS-B, MRPC,
RTE, SWAG



(c) Question Answering Tasks:
SQuAD v1.1



(b) Single Sentence Classification Tasks:
SST-2, CoLA



(d) Single Sentence Tagging Tasks:
CoNLL-2003 NER

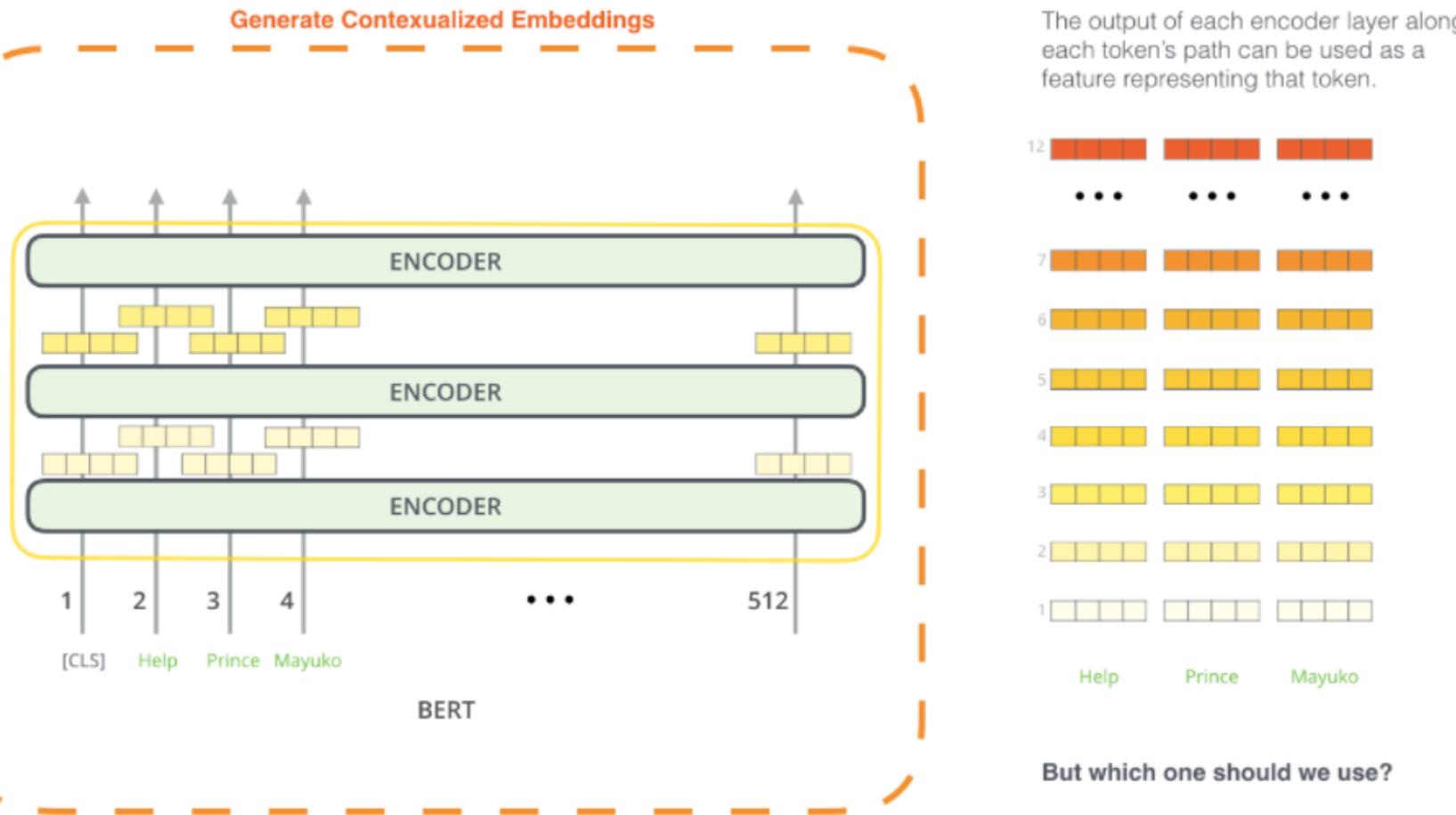
classification tasks

highlight which
span of paragraph
contains answer

← finetune named entity
label for **each** position
(person name, location,
other categories)

Using BERT to get features

We can also pull out features, just like with ELMo!

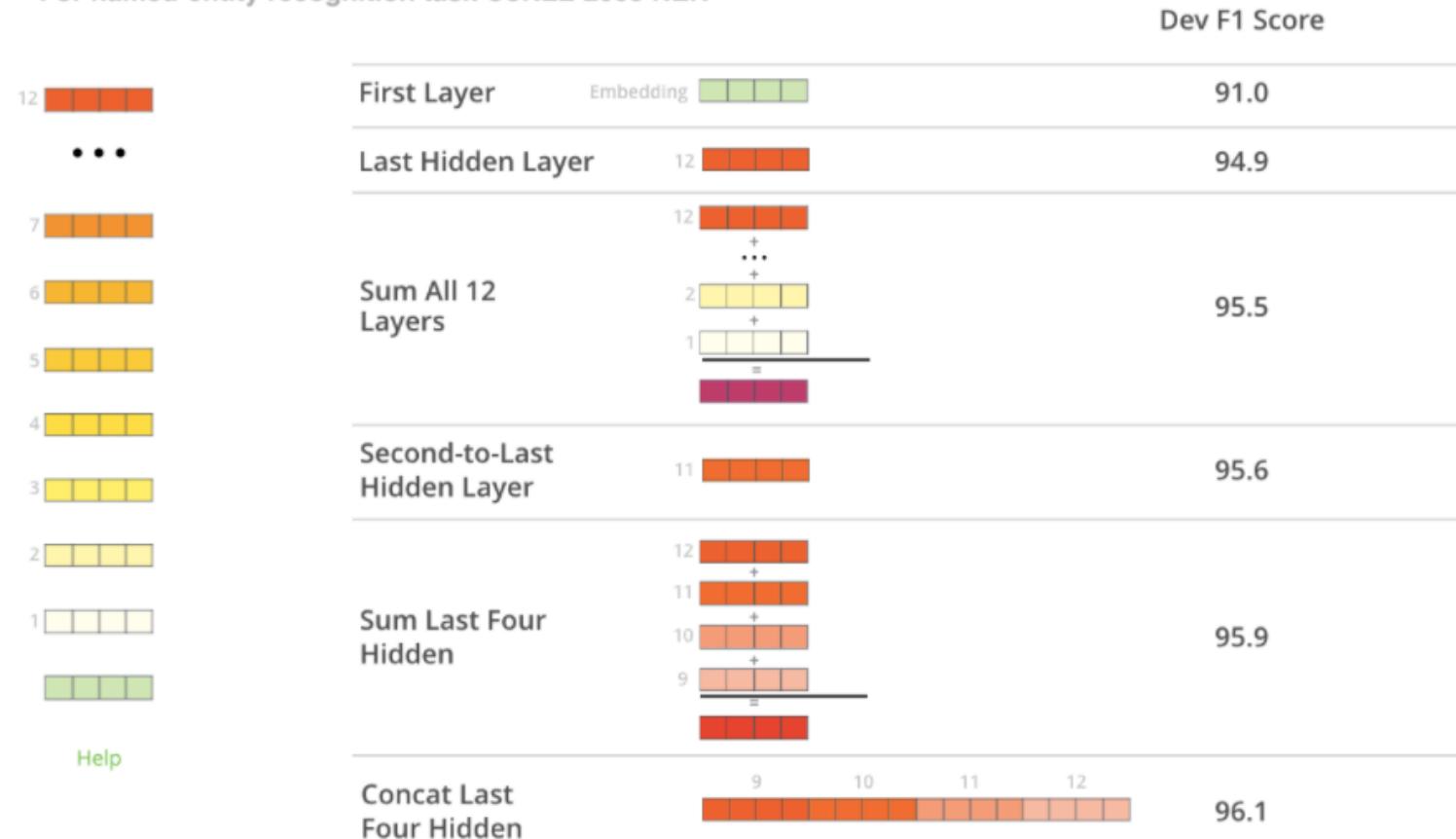


Using BERT to get features

We can also pull out features, just like with ELMo!

What is the best contextualized embedding for “**Help**” in that context?

For named-entity recognition task CoNLL-2003 NER



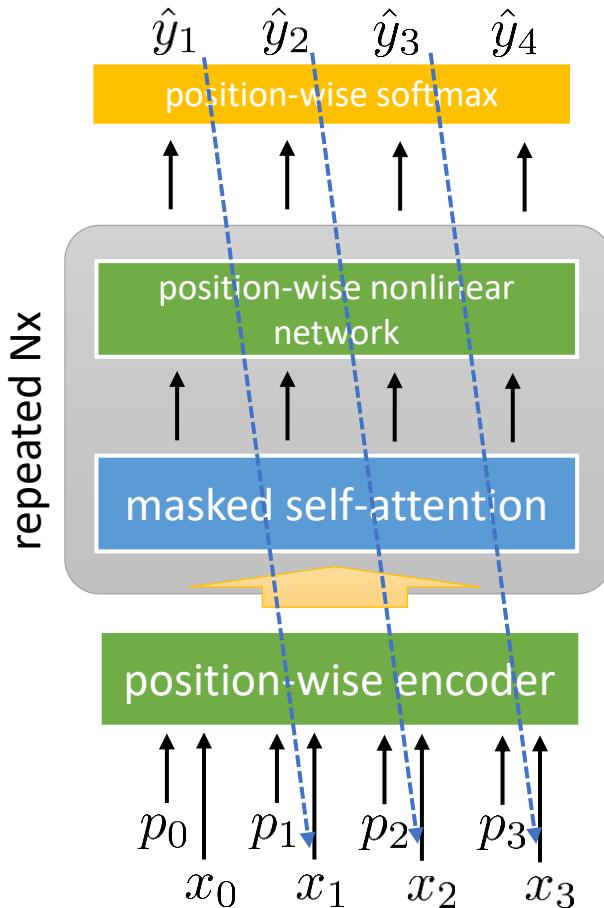
BERT results are extremely good

GLUE test result (battery of varied natural language understanding tasks)

	System	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	Average
		392k	363k	108k	67k	8.5k	5.7k	3.5k	2.5k	-
12 layers	Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
	BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
24 layers	OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
	BERT _{BASE}	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
	BERT _{LARGE}	86.7/85.9	72.1	92.7	94.9	60.5	86.5	89.3	70.1	82.1

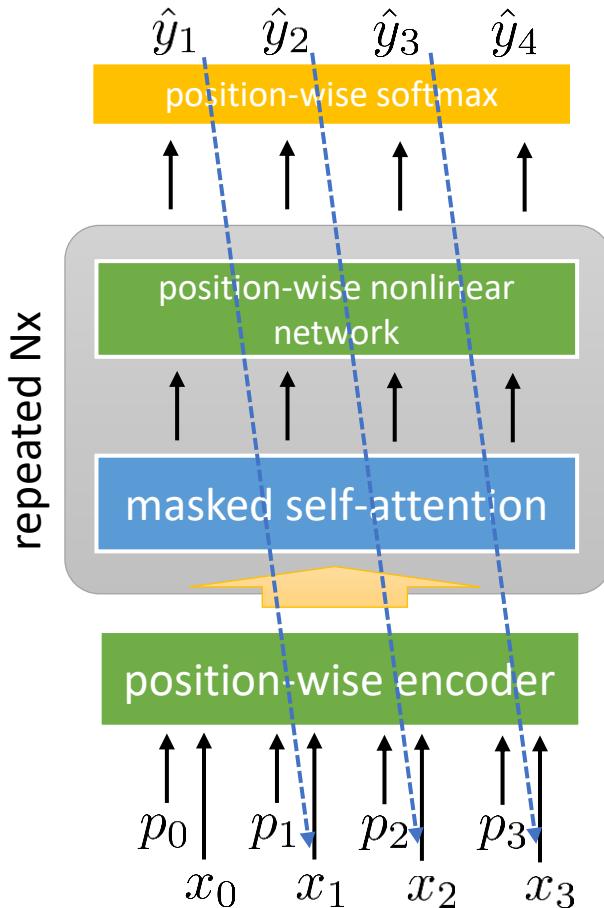
Since then, it has been applied to nearly
every NLP task you can imagine, and often
makes a **huge** difference in performance

GPT et al.



- One-directional (forward) transformer models do have one **big** advantage over BERT. Can you guess what it is?
- **Generation** is not really possible with BERT, but a forward (masked attention) model can do it!
- GPT (GPT-2, GPT-3, etc.) is a classic example of this

GPT et al.



OpenAI GPT-2 generated text

[source](#)

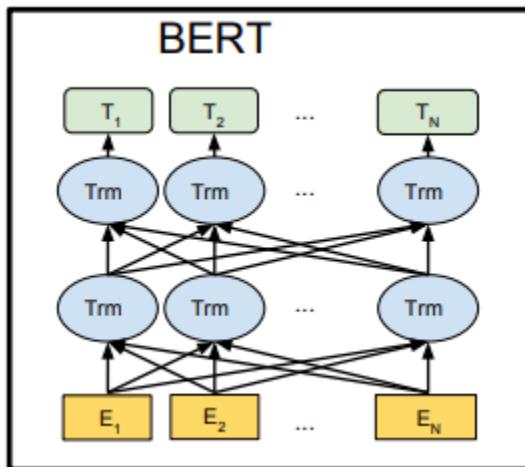
Input: In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

Output: The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.

Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

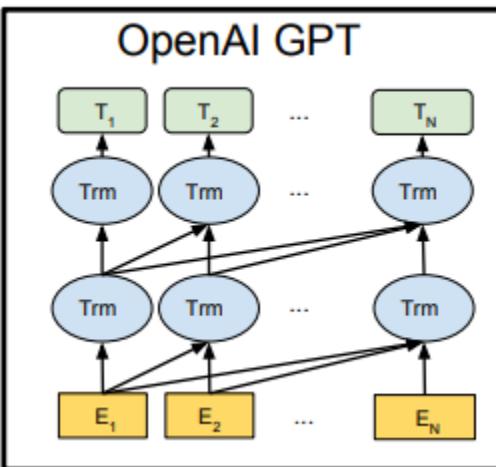
Pretrained language models summary



bidirectional transformer

+ great representations

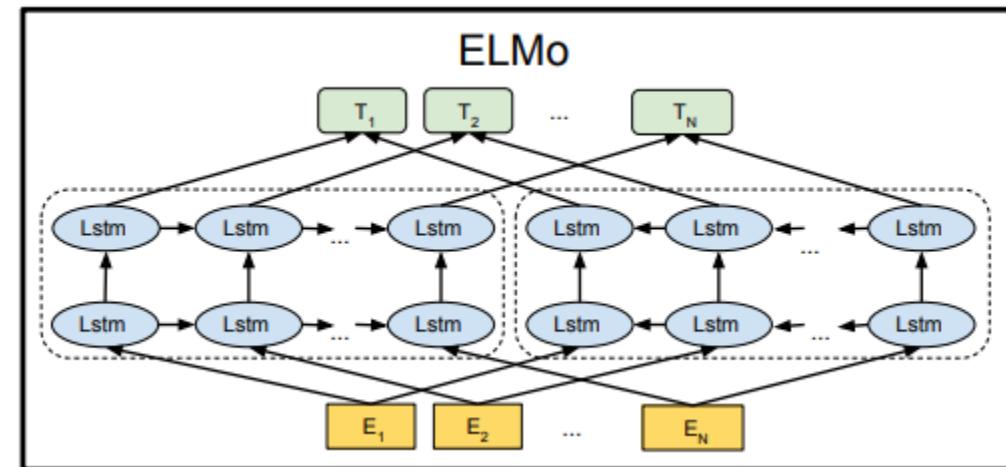
- can't generate text



one-directional transformer

+ can generate text

- OK representations



bidirectional LSTM

- OK representations

(largely supplanted by BERT)

Pretrained language models summary

- Language models can be trained on **very large and unlabeled** datasets of text (e.g., Wikipedia text). Often these are 100s or even 1000s of millions of sentences!
- Internal **learned representations** depend on context: the meaning of a word is informed by the **whole sentence**!
- Can even get us representations of entire sentences (e.g., the first output token for BERT)
- Can be used to either **extract representations** to replace standard word embeddings...
- ...or directly finetuned on downstream tasks (which means we modify all the weights in the whole language model, rather than just using pretrained model hidden states)

This is **very important** in modern NLP because **it works extremely well!**

Learning-Based Control & Imitation

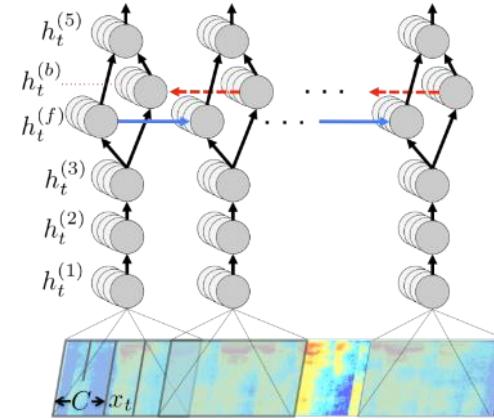
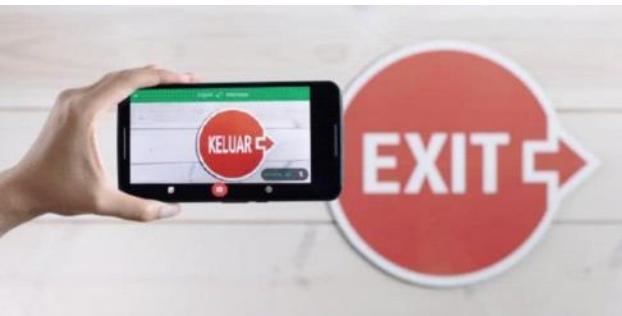
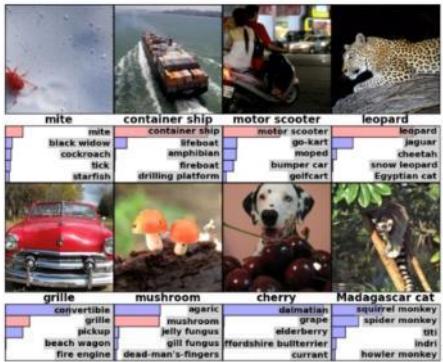
Designing, Visualizing and Understanding Deep Neural Networks

CS W182/282A

Instructor: Sergey Levine
UC Berkeley



So far: learning to *predict*



What about
learning to
control?



From prediction to control: challenges



$$\text{i.i.d.: } p(\mathcal{D}) = \prod_i p(y_i|x_i)p(x_i)$$

output y_1 does not change x_2

this is **very** important, because it allows us to just focus on getting the highest **average** accuracy over the whole dataset

making the wrong choice here is a disaster



making the wrong choice here is perhaps OK

From *prediction* to *control*: challenges



Ground truth labels:



“puppy”

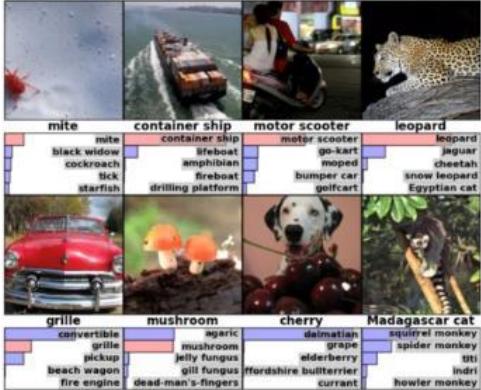


Abstract goals:

“drive to the grocery store”

> what steering command is that?

From *prediction* to *control*: challenges



- i.i.d. distributed data (each datapoint is independent)
- ground truth supervision
- objective is to predict the right label

These are not **just** issues for control: in many cases, real-world deployment of ML has these same **feedback** issues

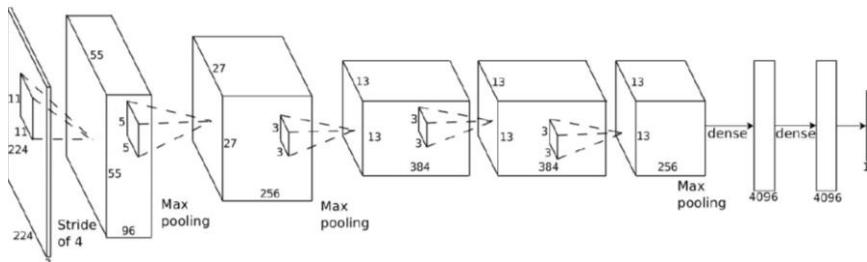
Example: decisions made by a traffic prediction system might affect the route that people take, which changes traffic



- each decision can change future inputs (not independent)
- supervision may be high-level (e.g., a goal)
- objective is to accomplish the task

We will **build up** toward a **reinforcement learning** system that addresses all of these issues, but we'll do so one piece at a time...

Terminology



used to be x

\mathbf{o}_t

s_t – state

\mathbf{o}_t – observation

a_t – action

$\pi_{\theta}(\mathbf{a}|\mathbf{o})$
used to be $p_{\theta}(y|x)$

$\pi_{\theta}(\mathbf{a}_t|\mathbf{o}_t)$ – policy

$\pi_{\theta}(\mathbf{a}_t|\mathbf{s}_t)$ – policy (fully observed)

used to be y

This distinction will very important later, but is not so important today



\mathbf{o}_t – observation

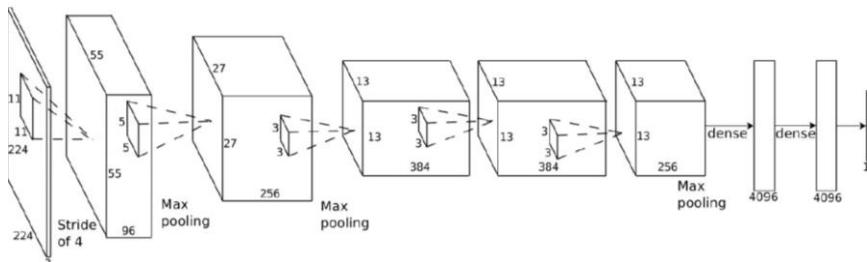


s_t – state

Terminology



\mathbf{o}_t



$\pi_\theta(\mathbf{a}_t | \mathbf{o}_t)$



\mathbf{a}_t

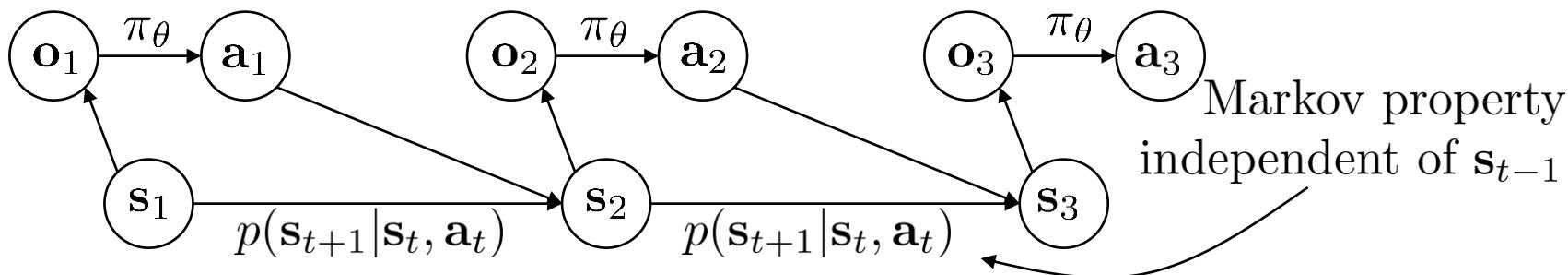
\mathbf{s}_t – state

\mathbf{o}_t – observation

\mathbf{a}_t – action

$\pi_\theta(\mathbf{a}_t | \mathbf{o}_t)$ – policy

$\pi_\theta(\mathbf{a}_t | \mathbf{s}_t)$ – policy (fully observed)



Aside: notation

s_t – state

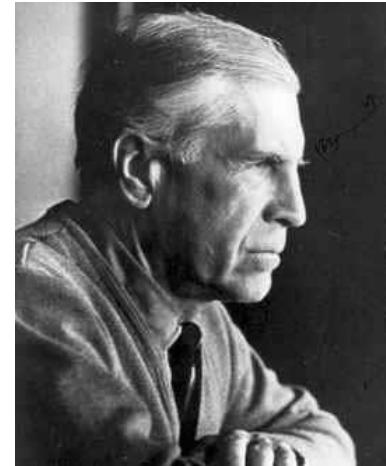
a_t – action

x_t – state

u_t – action управление



Richard Bellman

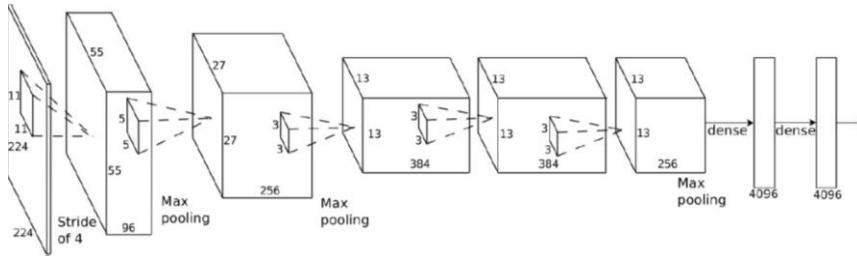


Lev Pontryagin

Imitation Learning



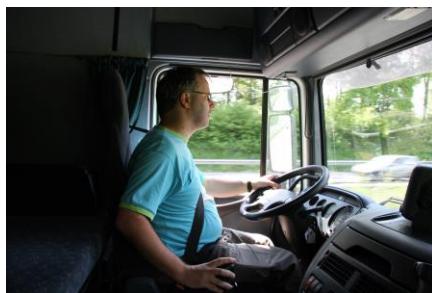
\mathbf{o}_t



$$\pi_{\theta}(\mathbf{a}_t | \mathbf{o}_t)$$

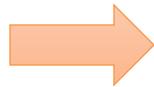


\mathbf{a}_t

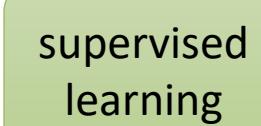


\mathbf{o}_t

\mathbf{a}_t



training
data

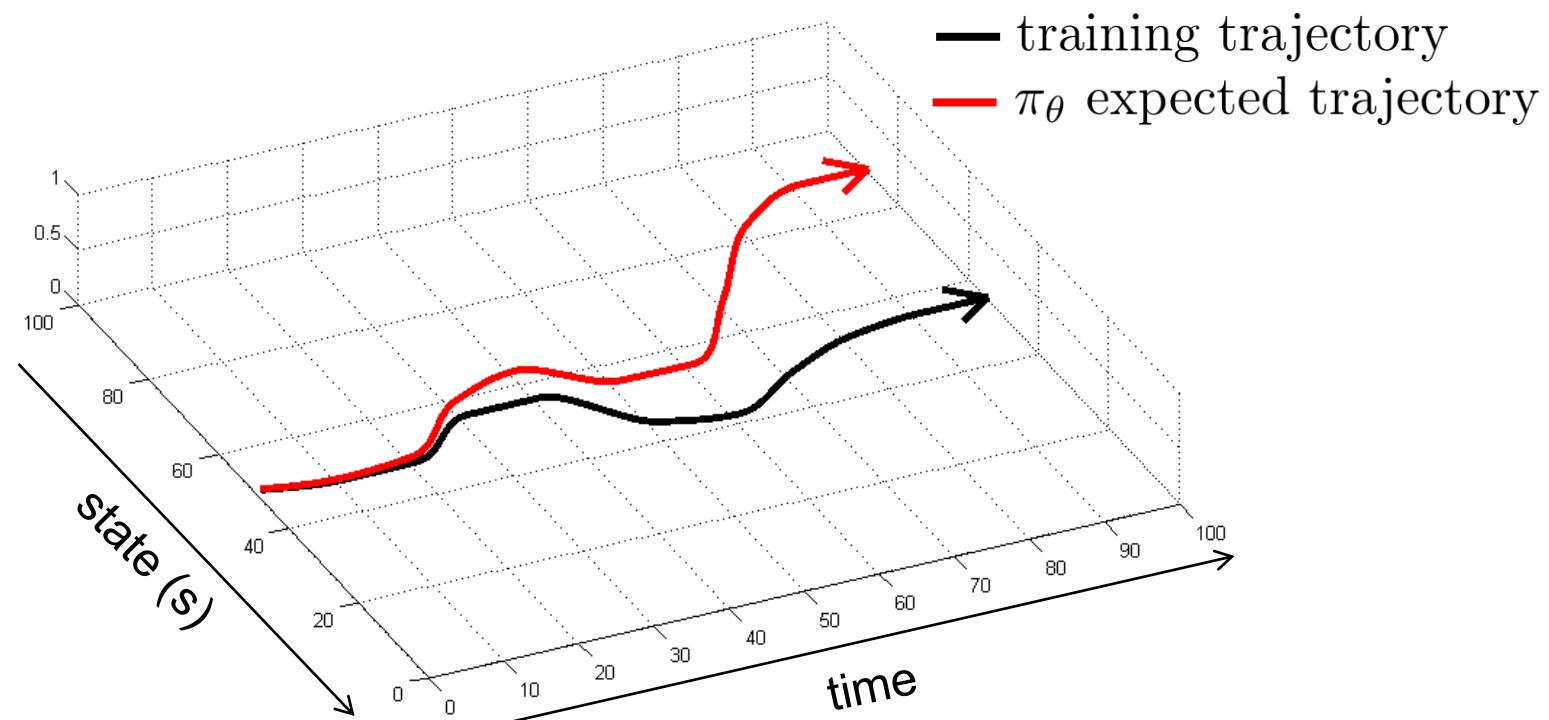


$$\pi_{\theta}(\mathbf{a}_t | \mathbf{o}_t)$$

behavioral cloning

Does it work?

No!



Where have we seen this before?

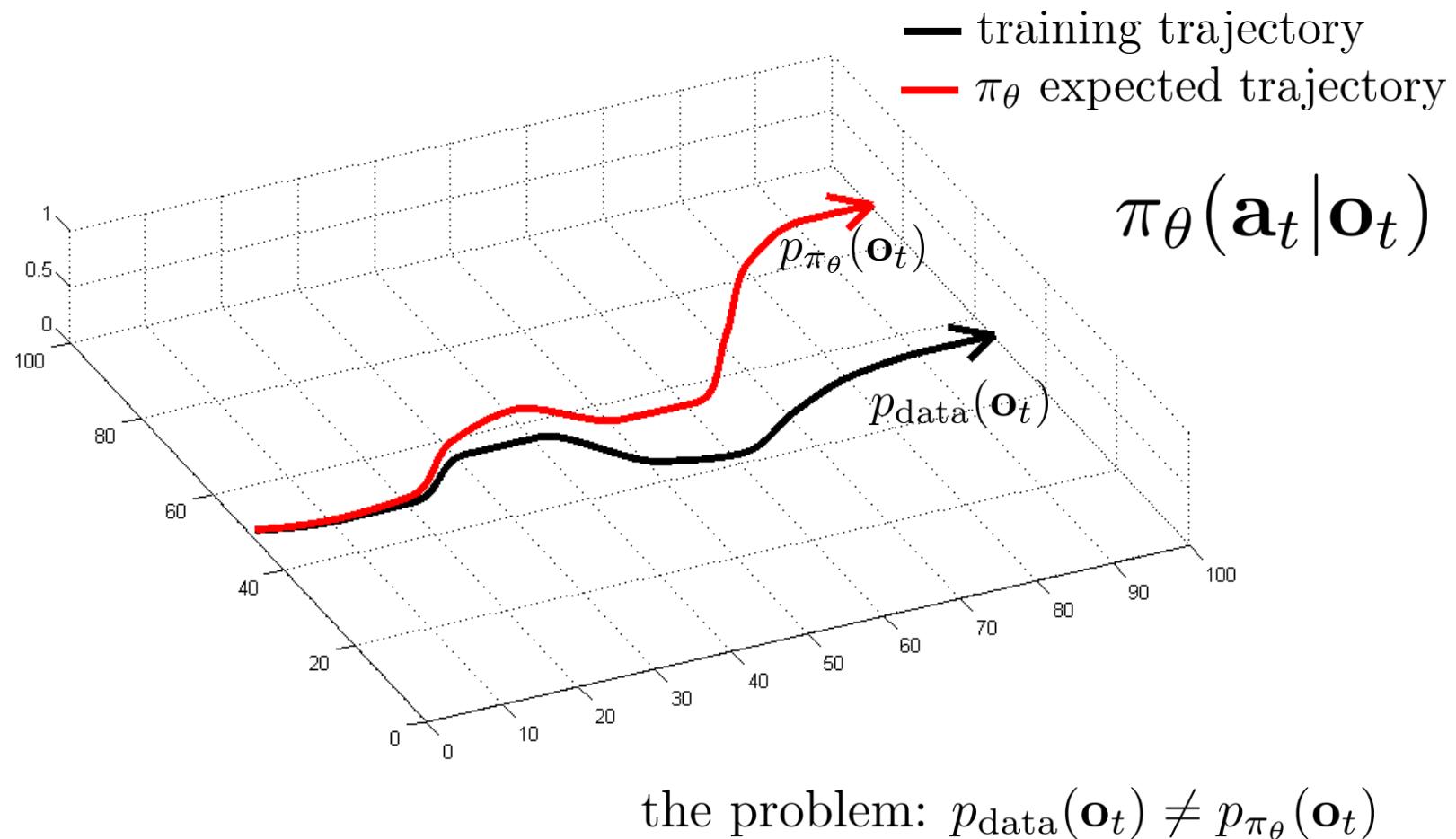
Does it work?

Yes!



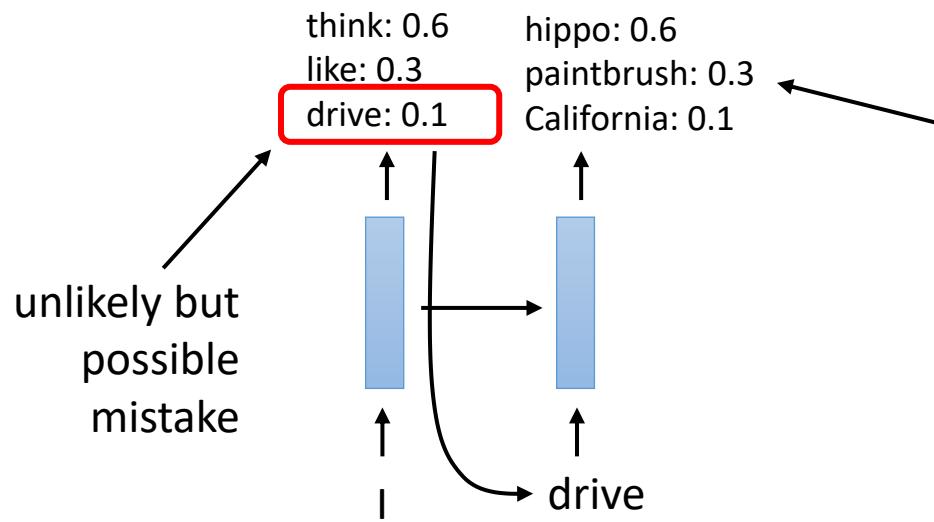
Getting behavioral cloning to work

What is the problem?



What is the problem?

the problem: $p_{\text{data}}(\mathbf{o}_t) \neq p_{\pi_\theta}(\mathbf{o}_t)$



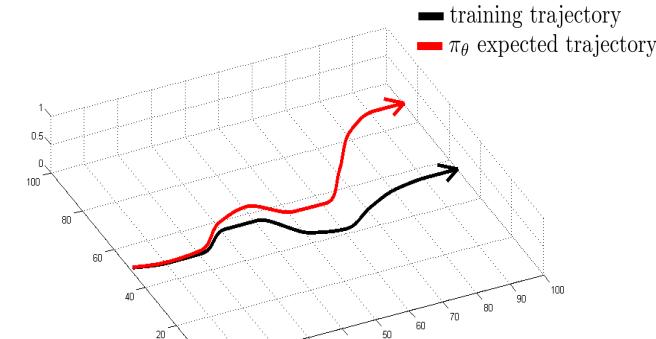
we got unlucky, but now the model is completely confused
it never saw “I drive” before

complete nonsense,
because the network never
saw inputs remotely like this

This is the same problem!

The problem: this is a training/test discrepancy:
the network always saw **true** sequences as
inputs, but at test-time it gets as input its own
(potentially incorrect) predictions

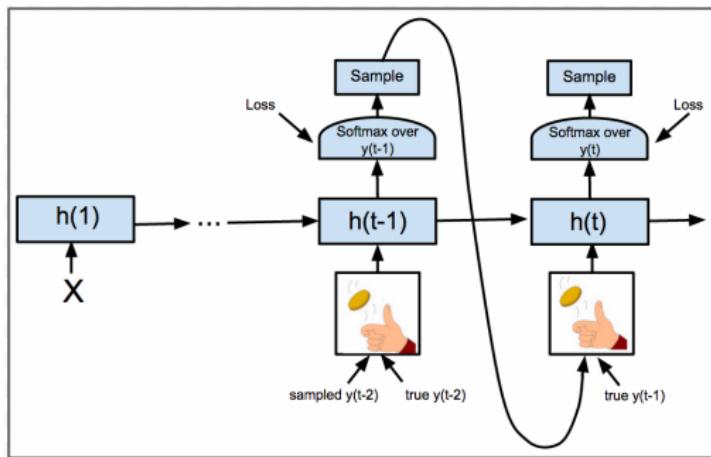
This is called **distributional shift**, because the
input distribution **shifts** from true strings (at
training) to synthetic strings (at test time)



Why not use the same solution?

the problem: $p_{\text{data}}(\mathbf{o}_t) \neq p_{\pi_\theta}(\mathbf{o}_t)$

Before: scheduled sampling

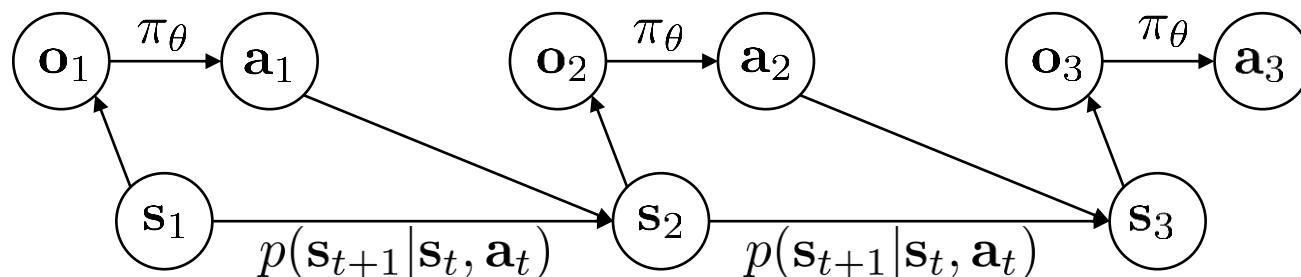


Now: control

we *could* take the predicted action $\mathbf{a}_t \sim \pi_\theta(\mathbf{a}_t | \mathbf{o}_t)$ and observe the resulting \mathbf{o}_{t+1}

but this requires interacting with the world!
why?

we don't know $p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)$!



Can we mitigate the problem?

the problem: $p_{\text{data}}(\mathbf{o}_t) \neq p_{\pi_\theta}(\mathbf{o}_t)$

if $\pi_\theta(\mathbf{a}_t|\mathbf{o}_t)$ is *very* accurate
maybe $p_{\text{data}}(\mathbf{o}_t) \approx p_\theta(\mathbf{o}_t)$

Why **might** we fail to fit the expert?

- 
1. Non-Markovian behavior
 2. Multimodal behavior

If we see the same thing twice, we do the same thing twice, regardless of what happened before

$$\pi_\theta(\mathbf{a}_t|\mathbf{o}_t)$$

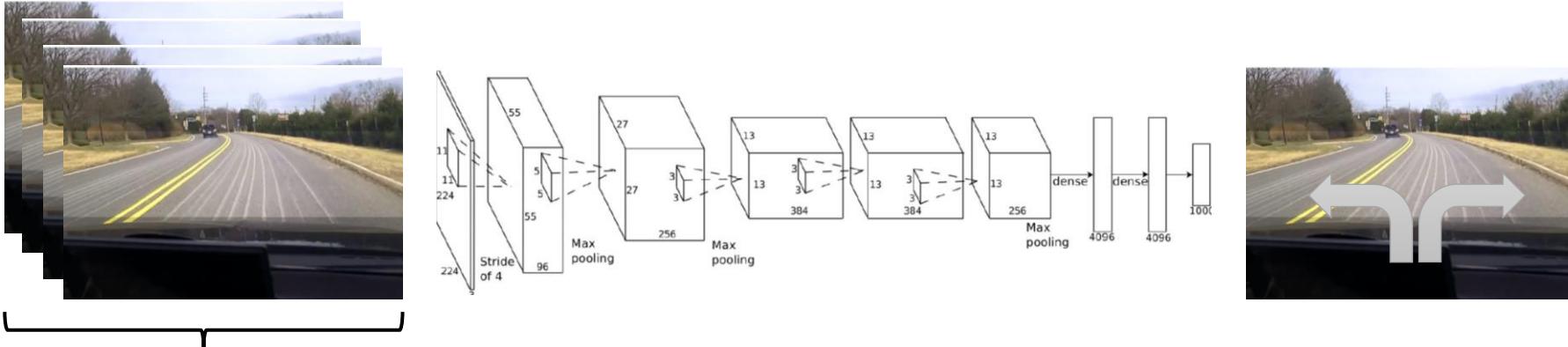
behavior depends only on current observation

$$\pi_\theta(\mathbf{a}_t|\mathbf{o}_1, \dots, \mathbf{o}_t)$$

behavior depends on all past observations

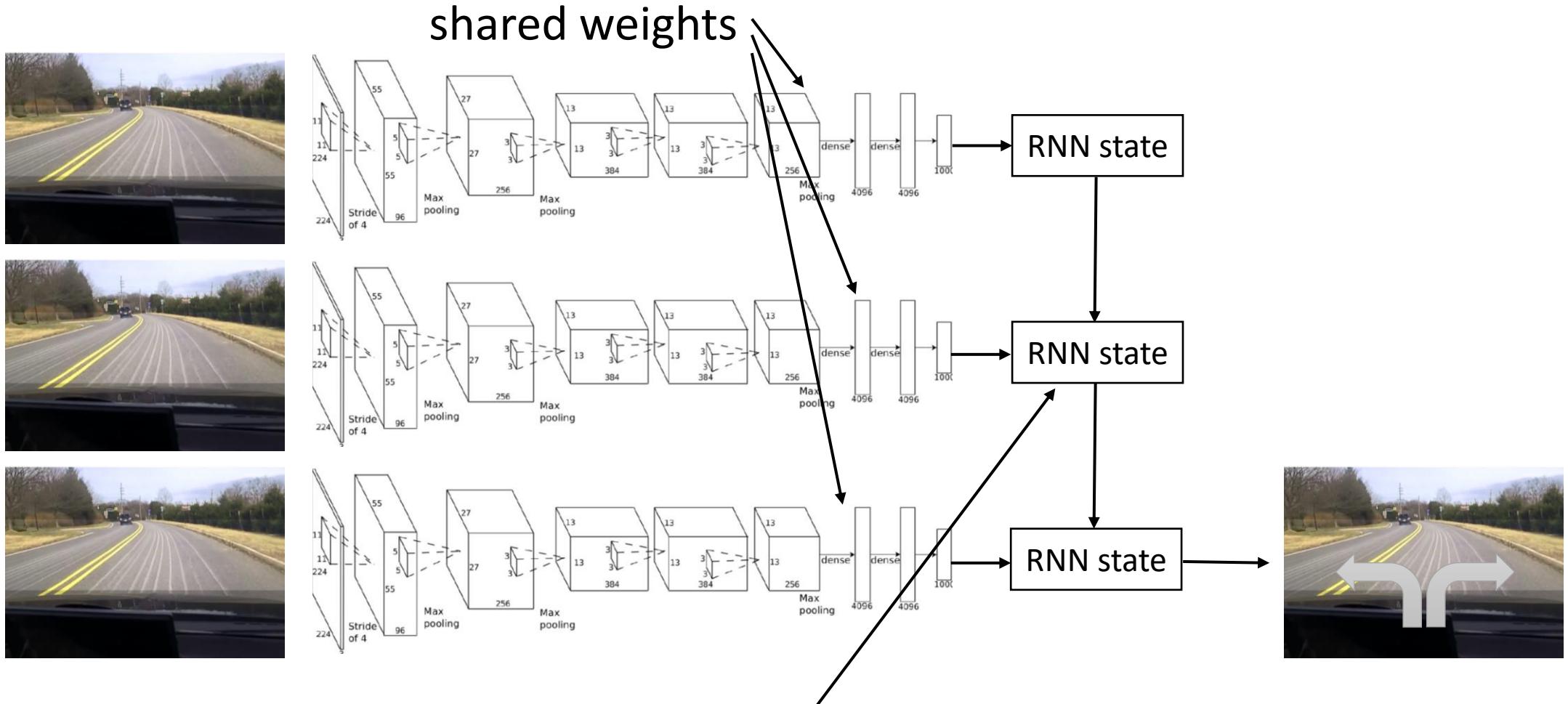
Often very unnatural for human demonstrators

How can we use the whole history?



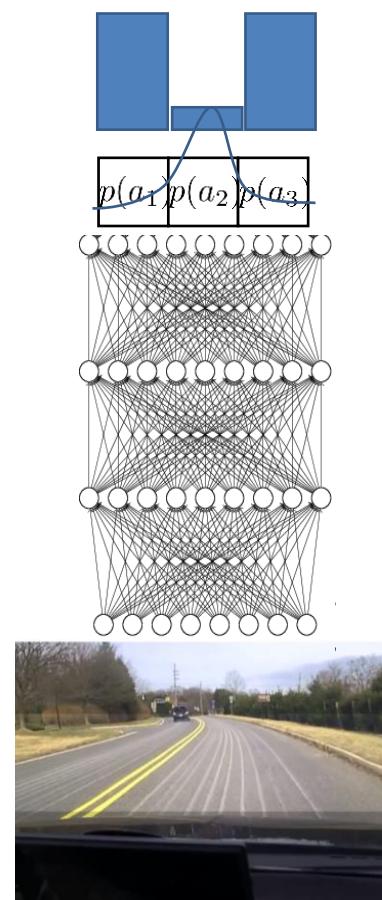
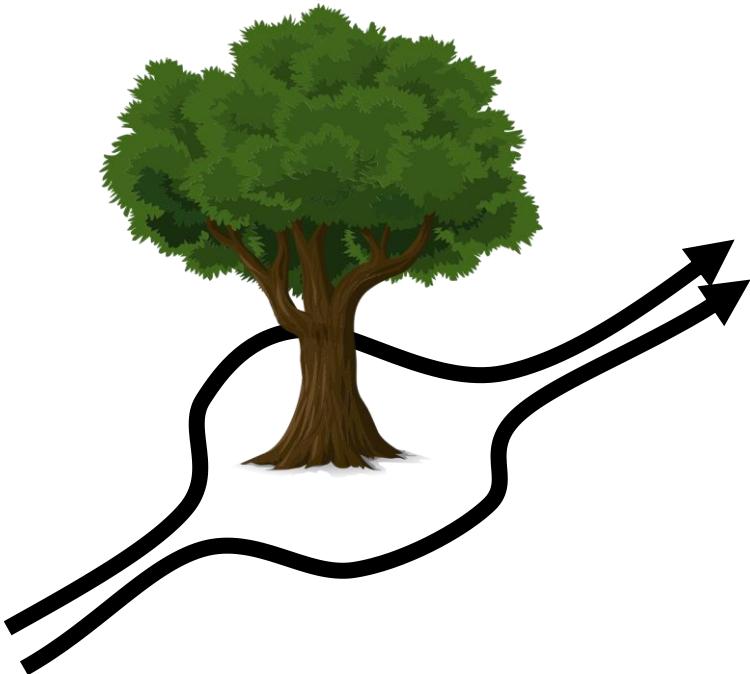
variable number of frames,
too many weights

How can we use the whole history?



Why might we fail to fit the expert?

- 1. Non-Markovian behavior
- 2. Multimodal behavior



- 1. Output mixture of Gaussians
- 2. Latent variable models
- 3. Autoregressive discretization

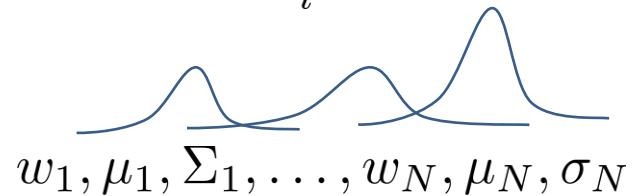


Why might we fail to fit the expert?

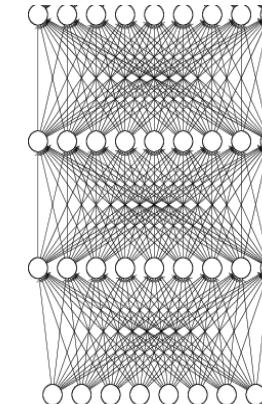


1. Output mixture of Gaussians
2. Latent variable models
3. Autoregressive discretization

$$\pi(\mathbf{a}|\mathbf{o}) = \sum_i w_i \mathcal{N}(\mu_i, \Sigma_i)$$



$w_1, \mu_1, \Sigma_1, \dots, w_N, \mu_N, \sigma_N$

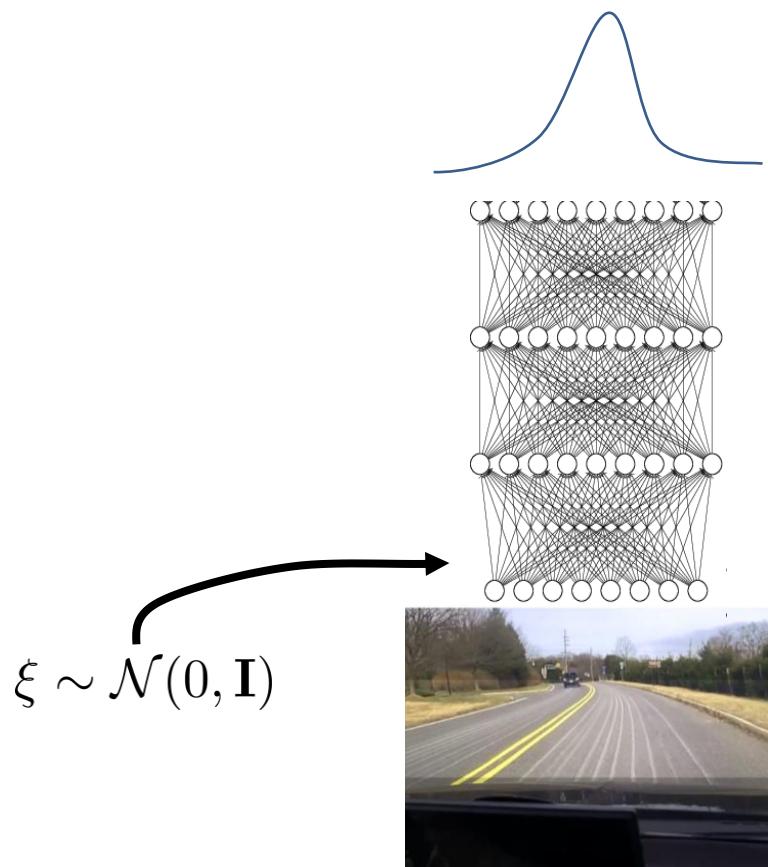


Why might we fail to fit the expert?

1. Output mixture of Gaussians
2. Latent variable models
3. Autoregressive discretization

Look up some of these:

- Conditional variational autoencoder
- Normalizing flow/realNVP
- Stein variational gradient descent

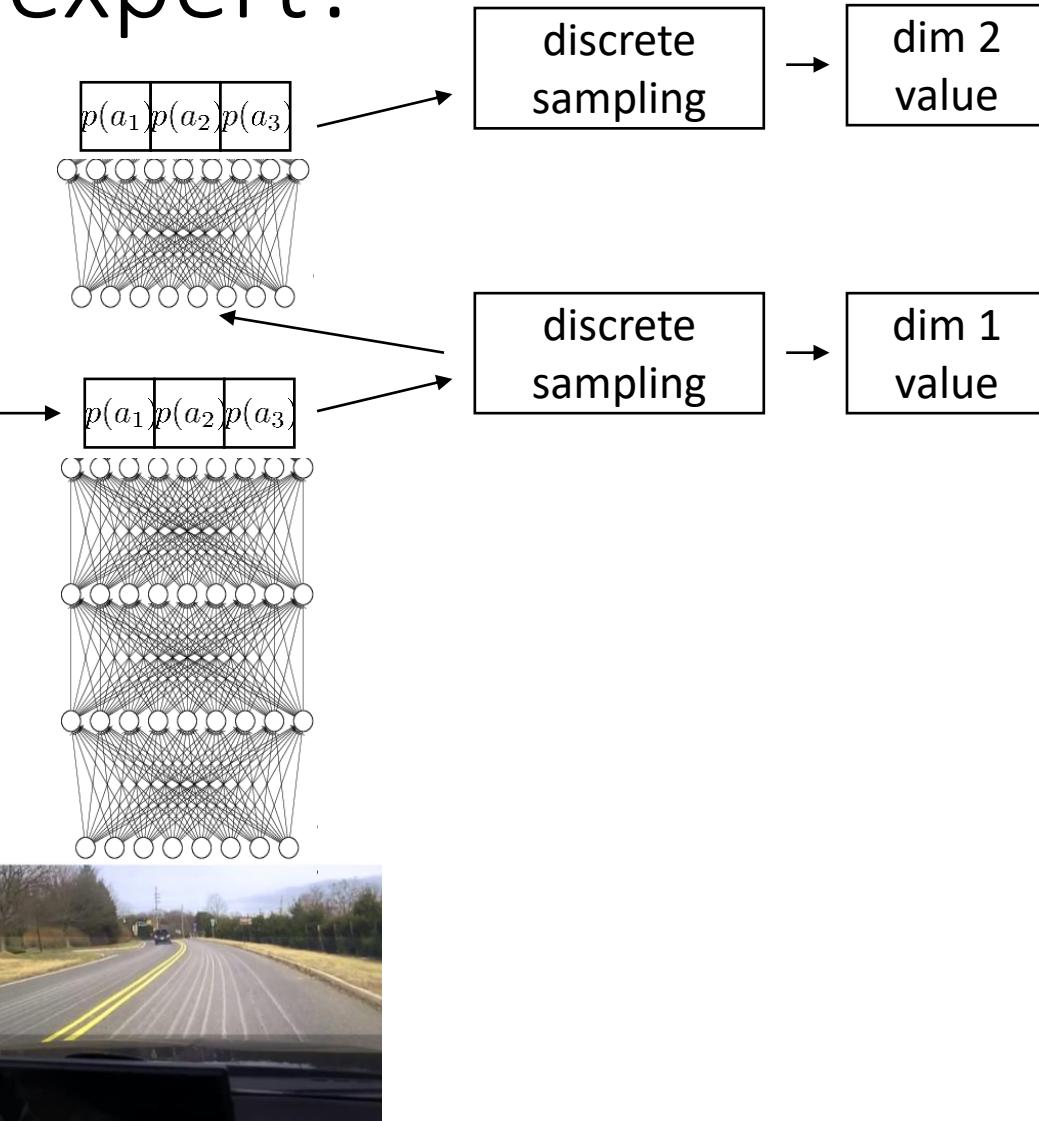


Why might we fail to fit the expert?

1. Output mixture of Gaussians
2. Latent variable models
3. Autoregressive discretization

We'll learn more about better ways to model multi-modal distributions when we cover generative models later

(discretized) distribution over dimension 1 **only**

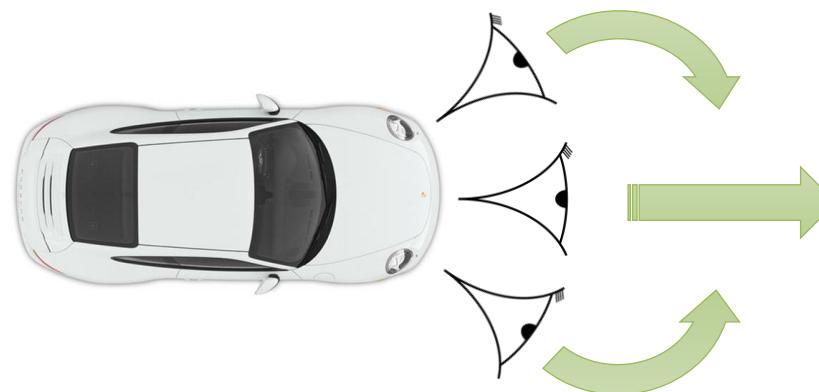
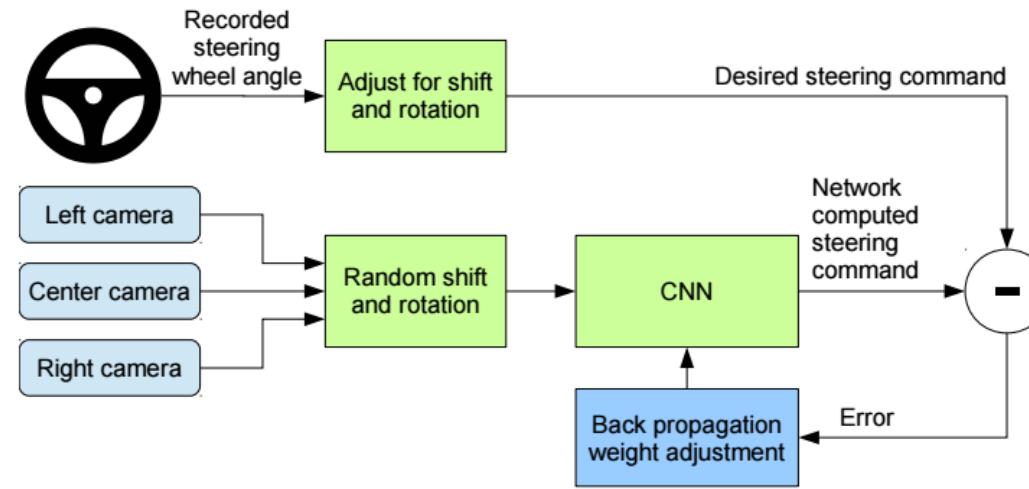


Does it work?

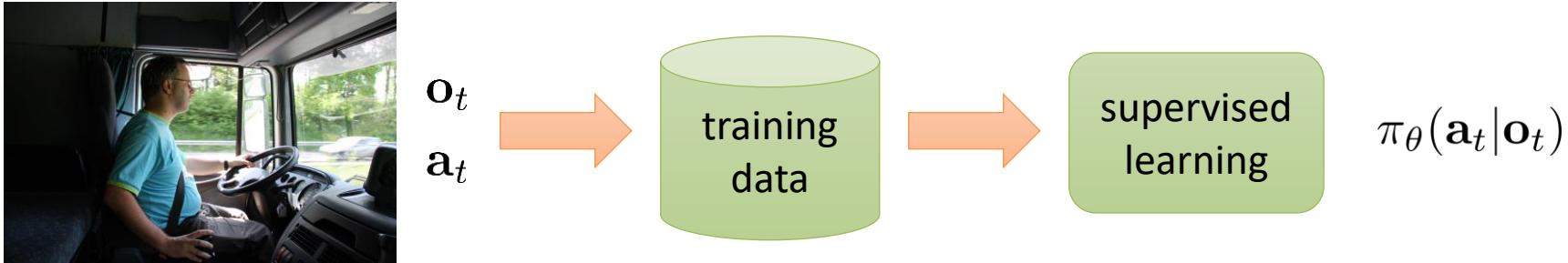
Yes!



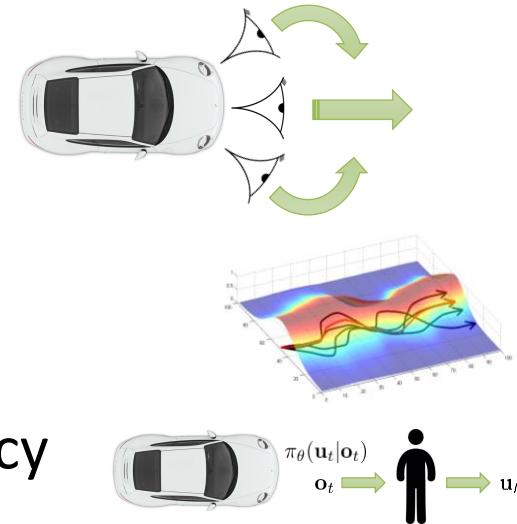
Why did that work?



Summary

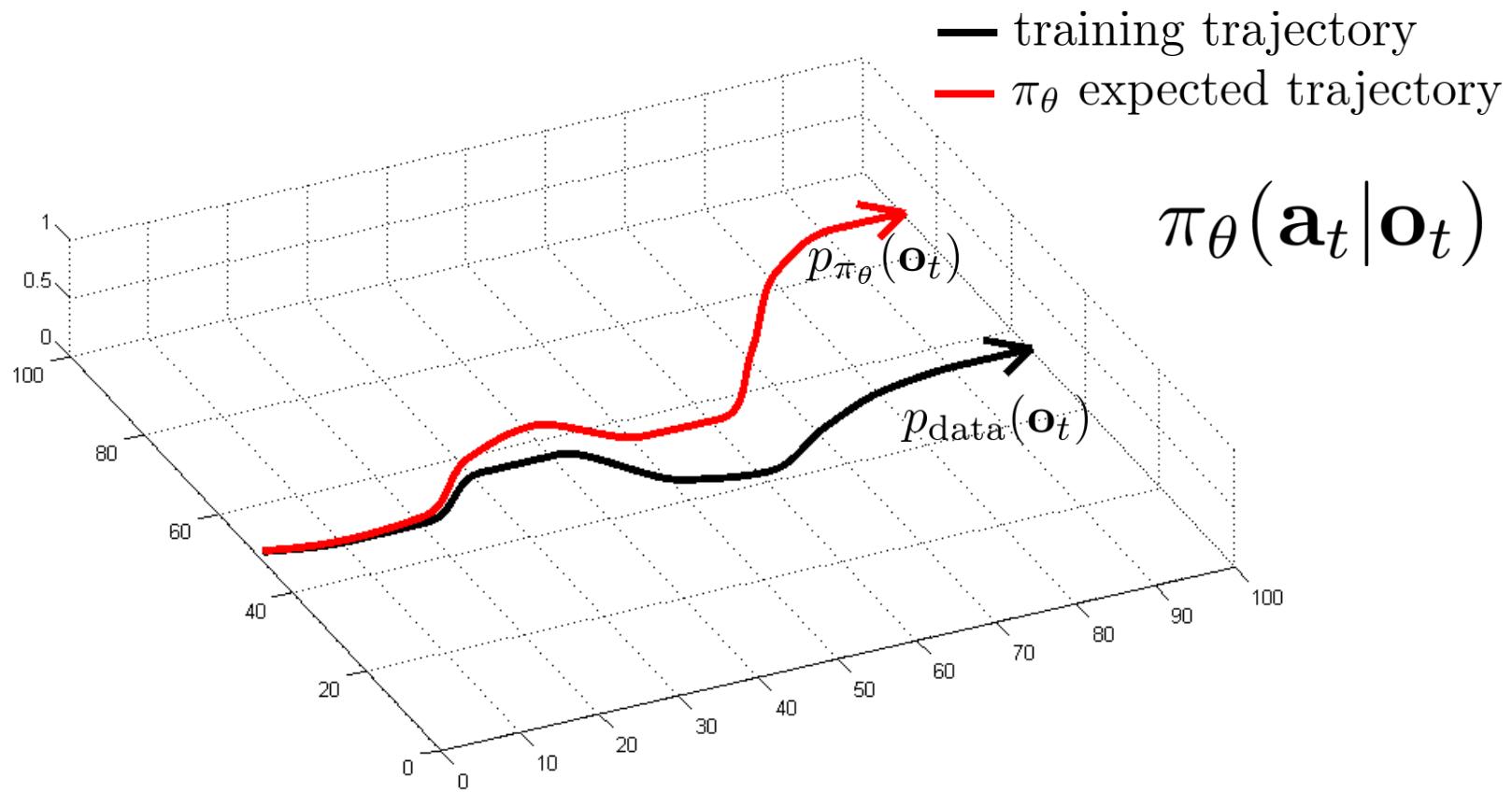


- In principle it should not work
 - Distribution mismatch problem
- Sometimes works well
 - Hacks (e.g. left/right images)
 - Models with memory (i.e., RNNs)
 - Better distribution modeling
 - Generally taking care to get high accuracy



A (perhaps) better approach

Can we make it work more often?



can we make $p_{\text{data}}(\mathbf{o}_t) = p_{\pi_\theta}(\mathbf{o}_t)$?

Can we make it work more often?

can we make $p_{\text{data}}(\mathbf{o}_t) = p_{\pi_\theta}(\mathbf{o}_t)$?

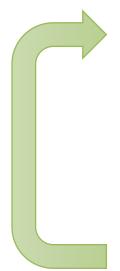
idea: instead of being clever about $p_{\pi_\theta}(\mathbf{o}_t)$, be clever about $p_{\text{data}}(\mathbf{o}_t)$!

DAgger: Dataset Aggregation

goal: collect training data from $p_{\pi_\theta}(\mathbf{o}_t)$ instead of $p_{\text{data}}(\mathbf{o}_t)$

how? just run $\pi_\theta(\mathbf{a}_t | \mathbf{o}_t)$

but need labels \mathbf{a}_t !

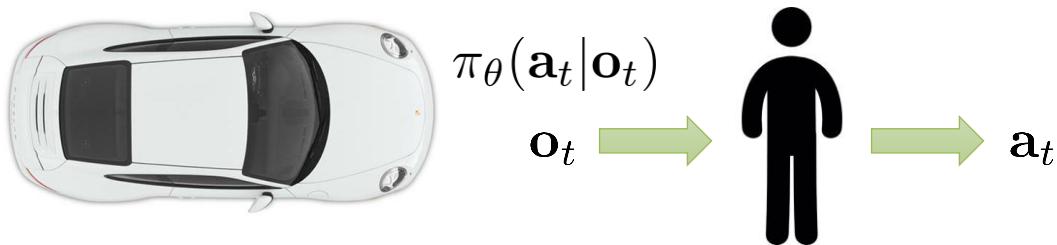
- 
1. train $\pi_\theta(\mathbf{a}_t | \mathbf{o}_t)$ from human data $\mathcal{D} = \{\mathbf{o}_1, \mathbf{a}_1, \dots, \mathbf{o}_N, \mathbf{a}_N\}$
 2. run $\pi_\theta(\mathbf{a}_t | \mathbf{o}_t)$ to get dataset $\mathcal{D}_\pi = \{\mathbf{o}_1, \dots, \mathbf{o}_M\}$
 3. Ask human to label \mathcal{D}_π with actions \mathbf{a}_t
 4. Aggregate: $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_\pi$

DAgger Example

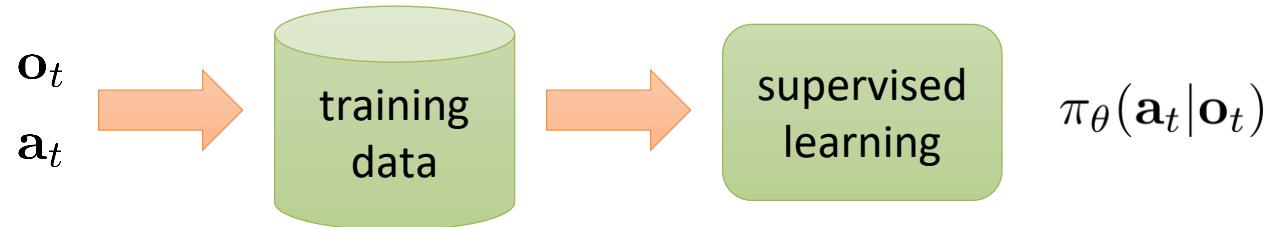


What's the problem?

1. train $\pi_\theta(\mathbf{a}_t|\mathbf{o}_t)$ from human data $\mathcal{D} = \{\mathbf{o}_1, \mathbf{a}_1, \dots, \mathbf{o}_N, \mathbf{a}_N\}$
2. run $\pi_\theta(\mathbf{a}_t|\mathbf{o}_t)$ to get dataset $\mathcal{D}_\pi = \{\mathbf{o}_1, \dots, \mathbf{o}_M\}$
3. Ask human to label \mathcal{D}_π with actions \mathbf{a}_t
4. Aggregate: $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_\pi$



Summary and takeaways



- In principle it should not work
 - Distribution mismatch problem
 - **DAgger can address this, but requires costly data collection and labeling**
- Sometimes works well
 - Requires a bit of (heuristic) hacks, and very good (high-accuracy) models

My recommendation: try behavioral cloning first, but prepare to be disappointed

Next time



- i.i.d. distributed data (each datapoint is independent)
- ground truth supervision
- objective is to predict the right label



- each decision can change future inputs (not independent)
- supervision may be high-level (e.g., a goal)
- objective is to accomplish the task

We'll tackle these issues with **reinforcement learning**

Reinforcement Learning

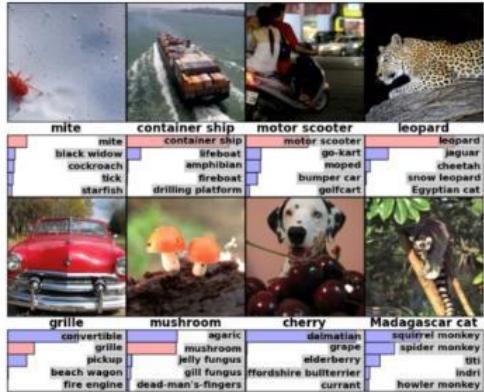
Designing, Visualizing and Understanding Deep Neural Networks

CS W182/282A

Instructor: Sergey Levine
UC Berkeley



From prediction to control



- i.i.d. distributed data (each datapoint is independent)
- ground truth supervision
- objective is to predict the right label

These are not **just** issues for control: in many cases, real-world deployment of ML has these same **feedback** issues

Example: decisions made by a traffic prediction system might affect the route that people take, which changes traffic

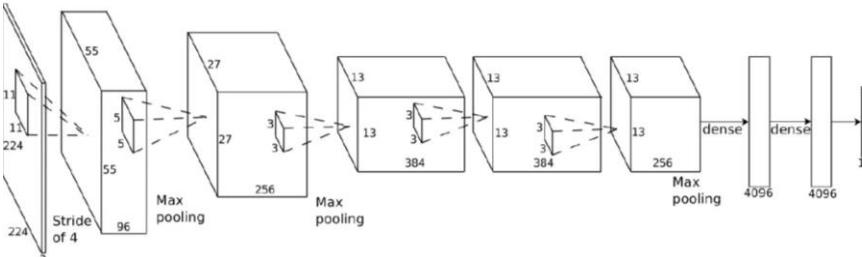


- each decision can change future inputs (not independent)
- supervision may be high-level (e.g., a goal)
- objective is to accomplish the task

How do we specify what we want?



\mathbf{x}_t



$$\pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t)$$



\mathbf{a}_t

which action is better or worse?

$r(\mathbf{s}, \mathbf{a})$: reward function

tells us which states and actions are better

note that we switched to states here,
we'll come back to observations later

We want to maximize
reward **over all time**, not
just greedily!



high reward



low reward

Definitions

Markov decision process

$$\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{T}, r\}$$

\mathcal{S} – state space

states $s \in \mathcal{S}$ (discrete or continuous)

\mathcal{A} – action space

actions $a \in \mathcal{A}$ (discrete or continuous)

r – reward function

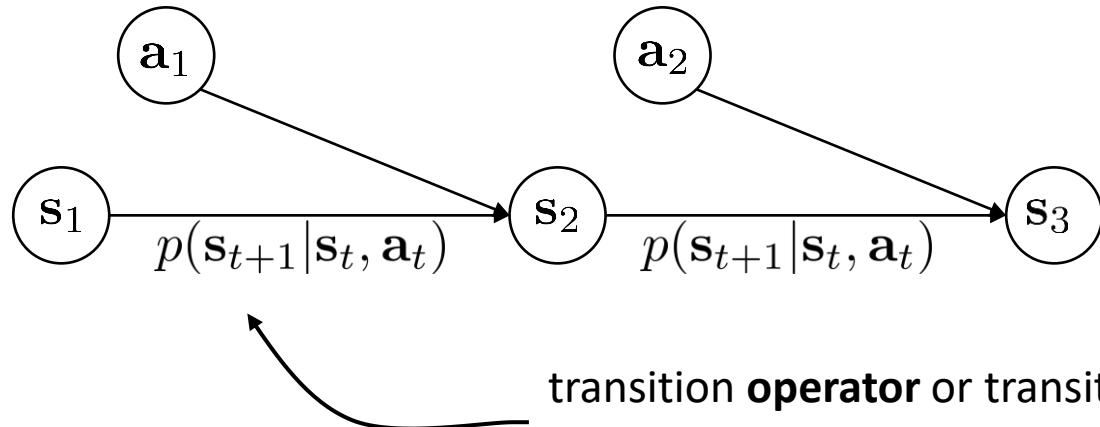
$$r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$$



Andrey Markov



Richard Bellman



Definitions

Markov decision process

$$\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{T}, r\}$$

\mathcal{S} – state space

states $s \in \mathcal{S}$ (discrete or continuous)

\mathcal{A} – action space

actions $a \in \mathcal{A}$ (discrete or continuous)

r – reward function

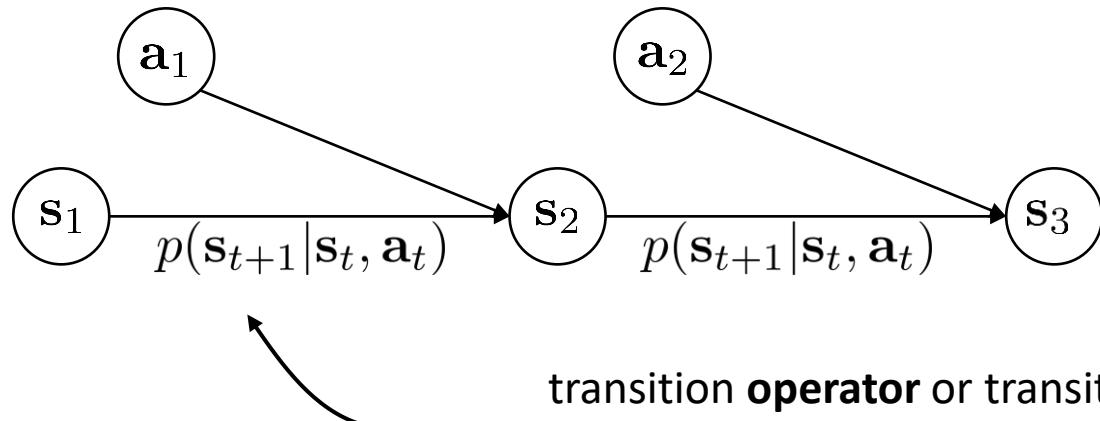
$$r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$$



Andrey Markov



Richard Bellman



transition **operator** or transition **probability**
or **dynamics** (all synonyms)
also referred to as $\mathcal{T}(s_{t+1}|s_t, a_t)$

Definitions

partially observed Markov decision process

$$\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \mathcal{E}, r\}$$

\mathcal{S} – state space

states $s \in \mathcal{S}$ (discrete or continuous)

\mathcal{A} – action space

actions $a \in \mathcal{A}$ (discrete or continuous)

\mathcal{O} – observation space

observations $o \in \mathcal{O}$ (discrete or continuous)

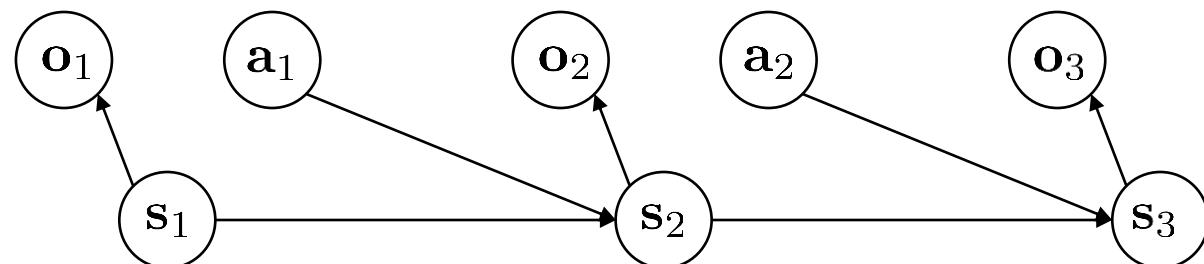
\mathcal{T} – transition operator (like before)

\mathcal{E} – emission probability $p(o_t|s_t)$

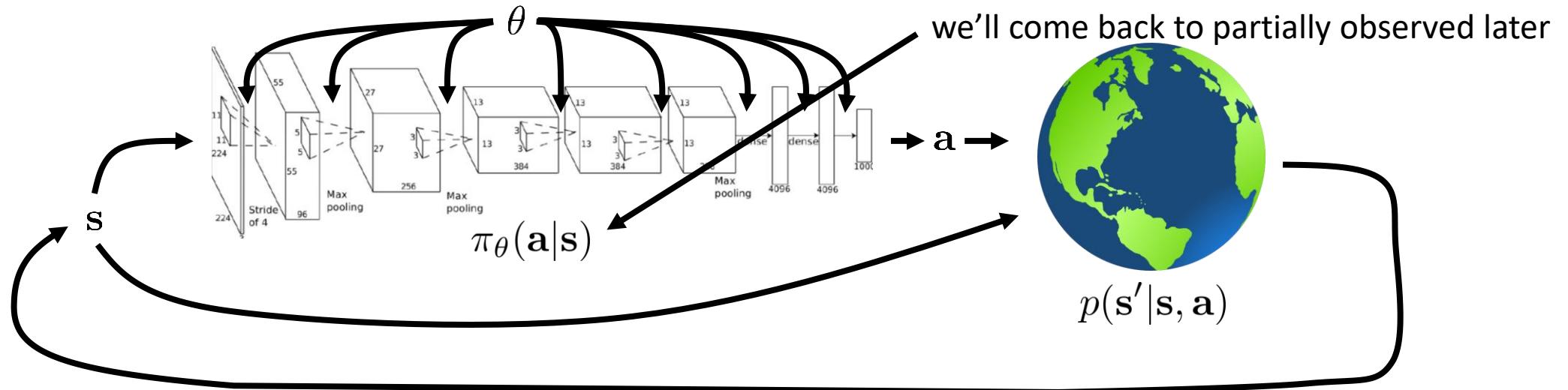
r – reward function

$$r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$$

For now, we'll stick to regular
(fully observed) MDPs, but you
should know that this exists



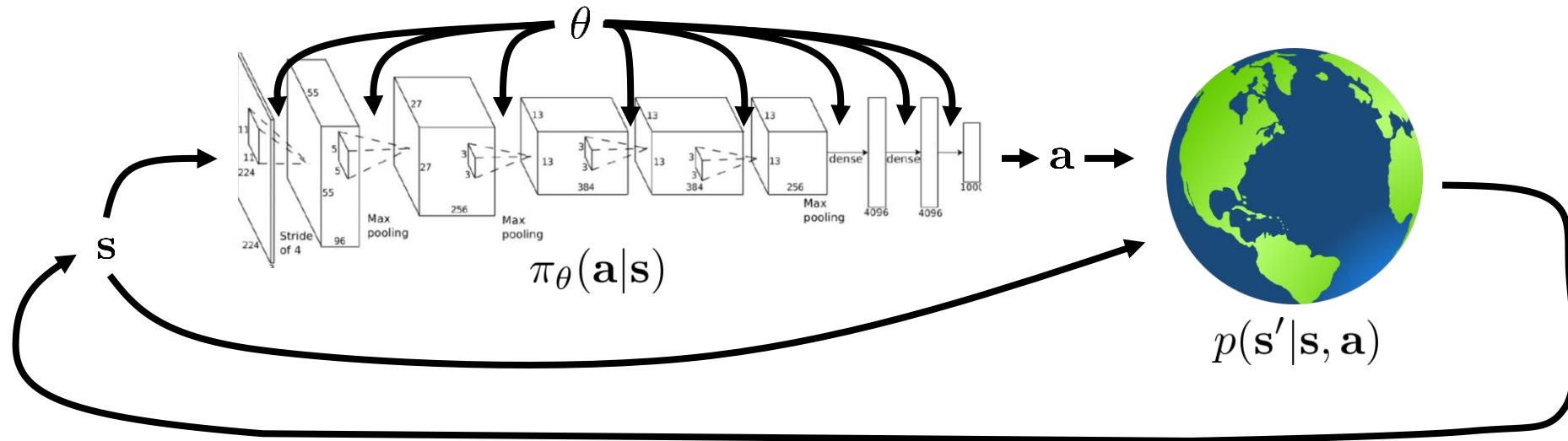
The goal of reinforcement learning



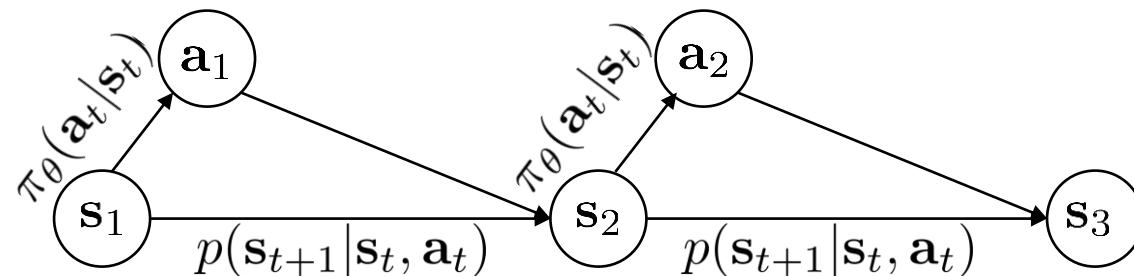
$$p_\theta(s_1, a_1, \dots, s_T, a_T) = p(s_1) \underbrace{\prod_{t=1}^T \pi_\theta(a_t | s_t)}_{p_\theta(\tau)} p(s_{t+1} | s_t, a_t)$$

$$\theta^\star = \arg \max_\theta E_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

The goal of reinforcement learning

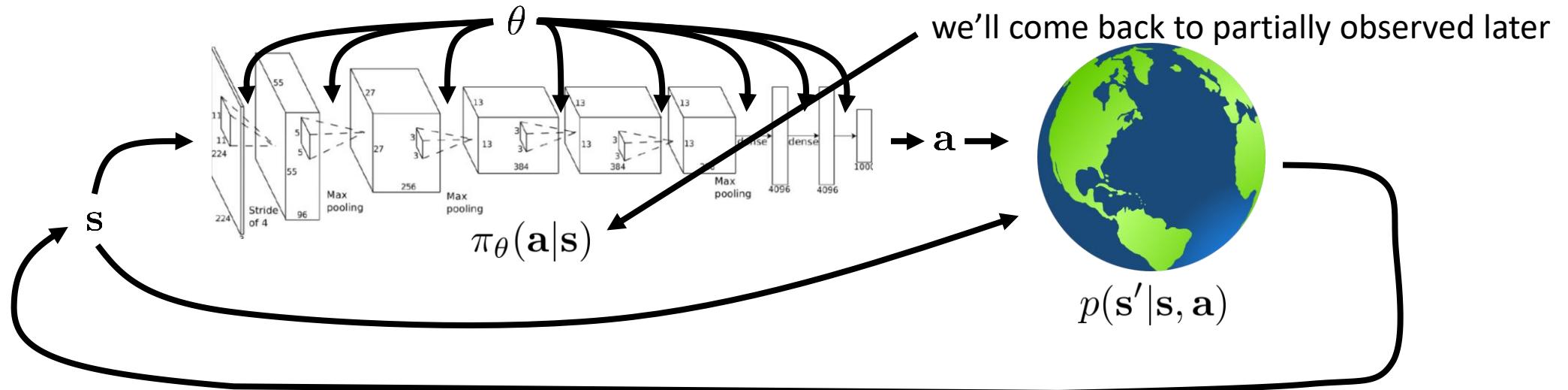


$$p_\theta(s_1, a_1, \dots, s_T, a_T) = p(s_1) \underbrace{\prod_{t=1}^T \pi_\theta(a_t | s_t)}_{p_\theta(\tau)} \underbrace{p(s_{t+1} | s_t, a_t)}_{\text{Markov chain on } (s, a)}$$



Wall of math time (policy gradients)

The goal of reinforcement learning



$$p_\theta(s_1, a_1, \dots, s_T, a_T) = p(s_1) \underbrace{\prod_{t=1}^T \pi_\theta(a_t | s_t)}_{p_\theta(\tau)} p(s_{t+1} | s_t, a_t)$$

$$\theta^\star = \arg \max_\theta E_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

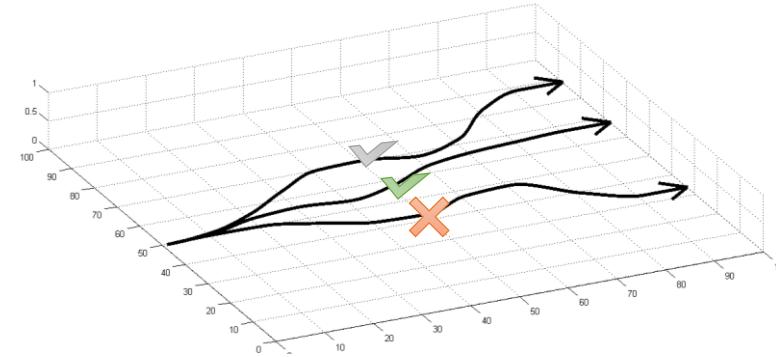
Evaluating the objective

$$\theta^* = \arg \max_{\theta} E_{\tau \sim p_{\theta}(\tau)} \left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right]$$

$J(\theta)$

$$J(\theta) = E_{\tau \sim p_{\theta}(\tau)} \left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right] \approx \frac{1}{N} \sum_i \sum_t r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})$$

sum over samples from π_{θ}



Direct policy differentiation

$$\theta^* = \arg \max_{\theta} E_{\tau \sim p_{\theta}(\tau)} \left[\underbrace{\sum_t r(\mathbf{s}_t, \mathbf{a}_t)}_{J(\theta)} \right]$$

a convenient identity

$$\underline{\pi_{\theta}(\tau) \nabla_{\theta} \log \pi_{\theta}(\tau)} = \pi_{\theta}(\tau) \frac{\nabla_{\theta} \pi_{\theta}(\tau)}{\pi_{\theta}(\tau)} = \underline{\nabla_{\theta} \pi_{\theta}(\tau)}$$

$$J(\theta) = E_{\tau \sim \pi_{\theta}(\tau)} [r(\tau)] = \int \pi_{\theta}(\tau) r(\tau) d\tau$$
$$\sum_{t=1}^T r(\mathbf{s}_t, \mathbf{a}_t)$$

$$\nabla_{\theta} J(\theta) = \int \underline{\nabla_{\theta} \pi_{\theta}(\tau)} r(\tau) d\tau = \int \underline{\pi_{\theta}(\tau) \nabla_{\theta} \log \pi_{\theta}(\tau)} r(\tau) d\tau = E_{\tau \sim \pi_{\theta}(\tau)} [\nabla_{\theta} \log \pi_{\theta}(\tau) r(\tau)]$$

Direct policy differentiation

$$\theta^* = \arg \max_{\theta} J(\theta)$$

$$J(\theta) = E_{\tau \sim \pi_\theta(\tau)}[r(\tau)]$$

$$\nabla_{\theta} J(\theta) = E_{\tau \sim \pi_\theta(\tau)}[\nabla_{\theta} \log \pi_{\theta}(\tau) r(\tau)]$$

$$\nabla_{\theta} \left[\cancel{\log p(\mathbf{s}_1)} + \sum_{t=1}^T \log \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t) + \cancel{\log p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)} \right]$$

$$\nabla_{\theta} J(\theta) = E_{\tau \sim \pi_\theta(\tau)} \left[\left(\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t) \right) \left(\sum_{t=1}^T r(\mathbf{s}_t, \mathbf{a}_t) \right) \right]$$

log of both sides

$$\pi_{\theta}(\mathbf{s}_1, \mathbf{a}_1, \dots, \mathbf{s}_T, \mathbf{a}_T) = p(\mathbf{s}_1) \prod_{t=1}^T \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t) p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)$$

$$\log \pi_{\theta}(\tau) = \log p(\mathbf{s}_1) + \sum_{t=1}^T \log \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t) + \log p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)$$

Evaluating the policy gradient

recall: $J(\theta) = E_{\tau \sim p_\theta(\tau)} \left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right] \approx \frac{1}{N} \sum_i \sum_t r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})$

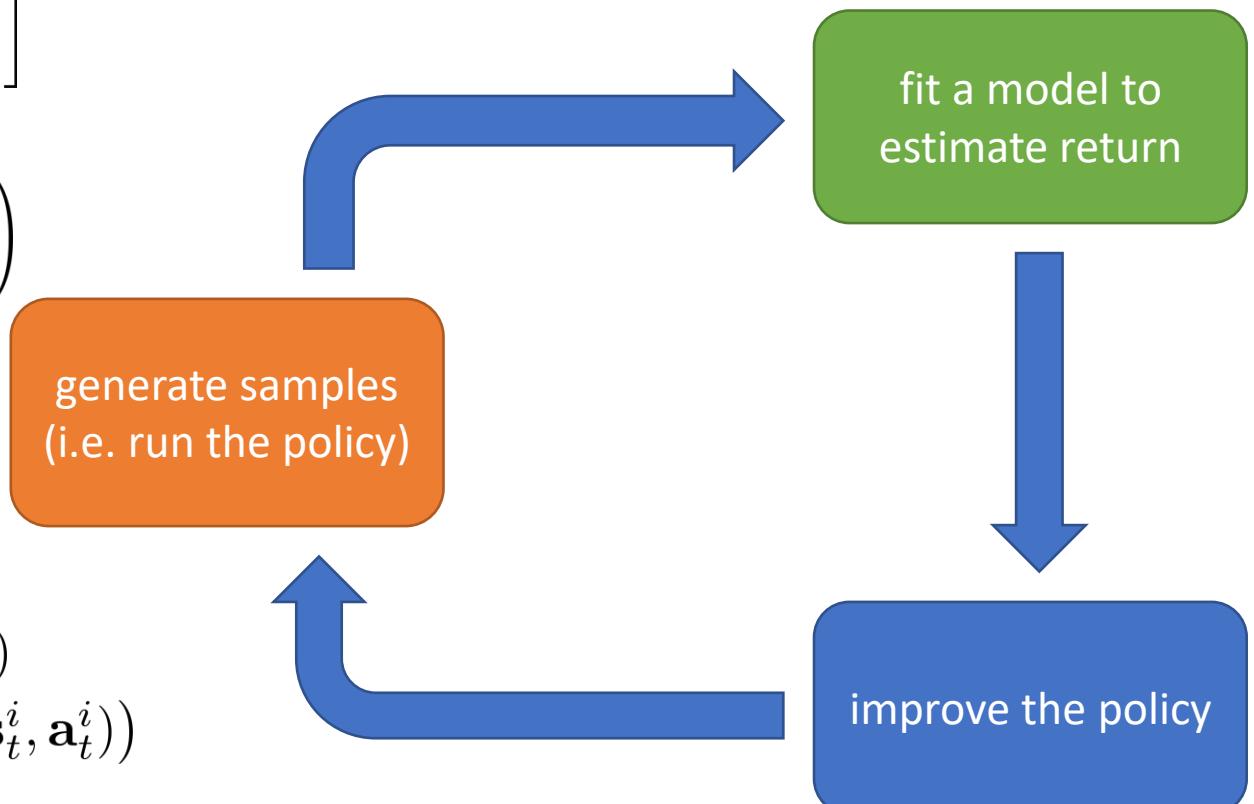
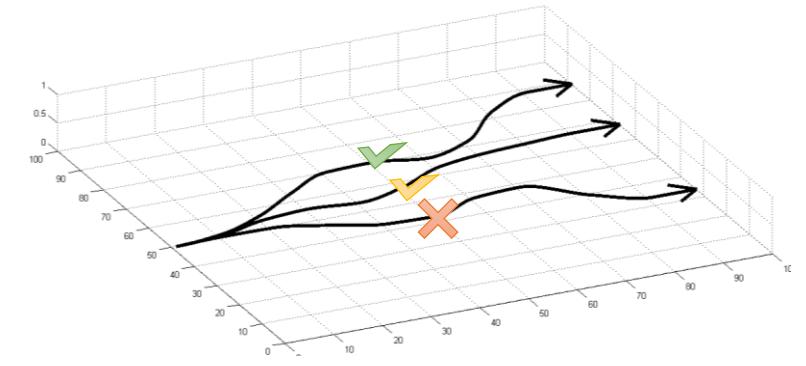
$$\nabla_\theta J(\theta) = E_{\tau \sim \pi_\theta(\tau)} \left[\left(\sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{a}_t | \mathbf{s}_t) \right) \left(\sum_{t=1}^T r(\mathbf{s}_t, \mathbf{a}_t) \right) \right]$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \right) \left(\sum_{t=1}^T r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right)$$

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

REINFORCE algorithm:

- 1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t | \mathbf{s}_t)$ (run the policy)
- 2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i | \mathbf{s}_t^i) \right) \left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i) \right)$
- 3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$



Evaluating the policy gradient

recall: $J(\theta) = E_{\tau \sim p_\theta(\tau)} \left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right] \approx \frac{1}{N} \sum_i \sum_t r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})$

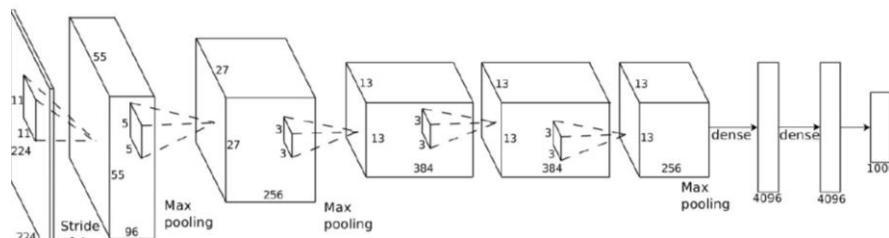
$$\nabla_\theta J(\theta) = E_{\tau \sim \pi_\theta(\tau)} \left[\left(\sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{a}_t | \mathbf{s}_t) \right) \left(\sum_{t=1}^T r(\mathbf{s}_t, \mathbf{a}_t) \right) \right]$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \right) \left(\sum_{t=1}^T r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right)$$

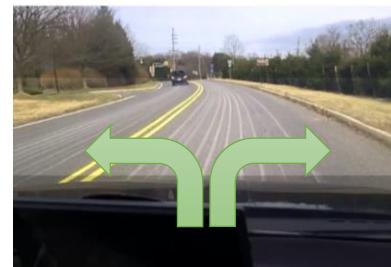
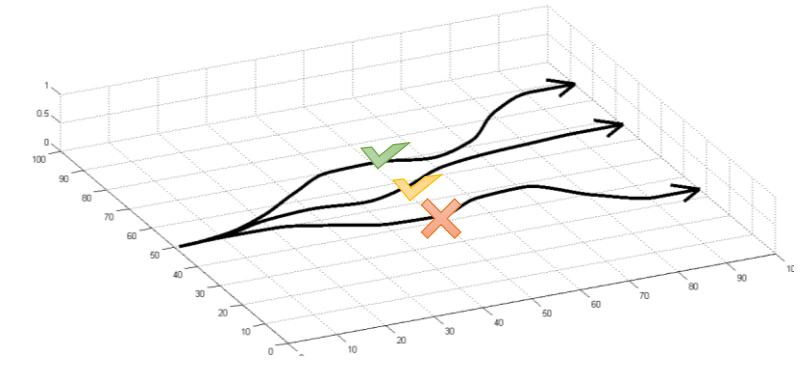
what is this?



\mathbf{s}_t



$\pi_\theta(\mathbf{a}_t | \mathbf{s}_t)$



\mathbf{a}_t

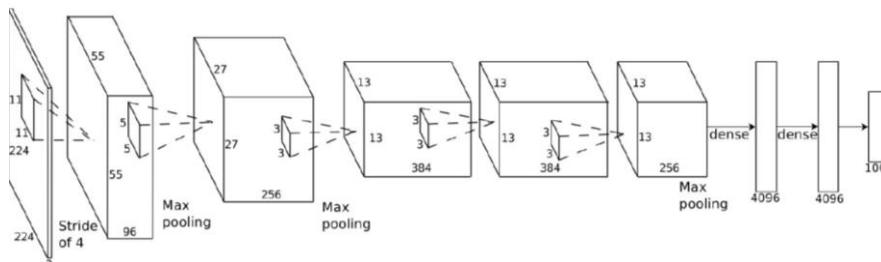
Comparison to maximum likelihood

policy gradient: $\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \right) \left(\sum_{t=1}^T r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right)$

maximum likelihood: $\nabla_{\theta} J_{\text{ML}}(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \right)$



\mathbf{s}_t



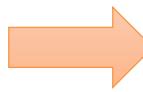
$\pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t)$



\mathbf{a}_t



\mathbf{s}_t
 \mathbf{a}_t



supervised
learning

$\pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t)$

What did we just do?

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \right) \left(\sum_{t=1}^T r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right)$$

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \underbrace{\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\tau_i)}_{r(\tau_i)} + \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t})$$

maximum likelihood: $\nabla_{\theta} J_{\text{ML}}(\theta) \approx \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} \log \pi_{\theta}(\tau_i)$

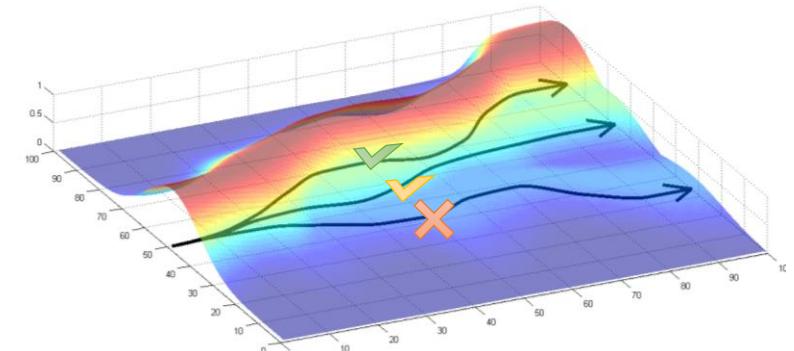
good stuff is made more likely

bad stuff is made less likely

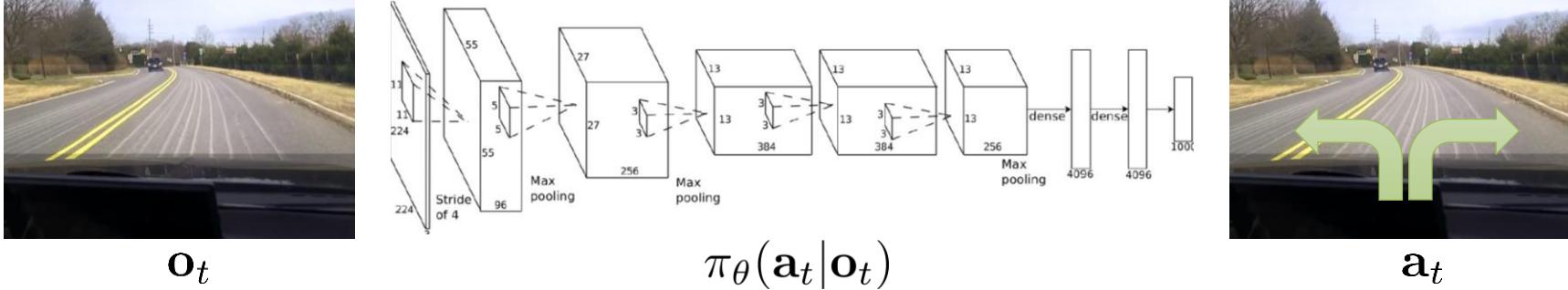
simply formalizes the notion of “trial and error”!

REINFORCE algorithm:

- 1. sample $\{\tau^i\}$ from $\pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t)$ (run it on the robot)
- 2. $\nabla_{\theta} J(\theta) \approx \sum_i (\sum_t \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_t^i | \mathbf{s}_t^i)) (\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i))$
- 3. $\theta \leftarrow \theta + \alpha \nabla_{\theta} J(\theta)$



Partial observability



$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} | \mathbf{o}_{i,t}) \right) \left(\sum_{t=1}^T r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right)$$

Markov property is not actually used!

Can use policy gradient in partially observed MDPs without modification

Making policy gradient actually work

A better estimator

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \right) \left(\sum_{t=1}^T r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right)$$

Causality: policy at time t' cannot affect reward at time t when $t < t'$

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \underbrace{\left(\sum_{t'=1}^T r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}) \right)}_{t' \neq t}$$

“reward to go”

$$\hat{Q}_{i,t}$$

Baselines

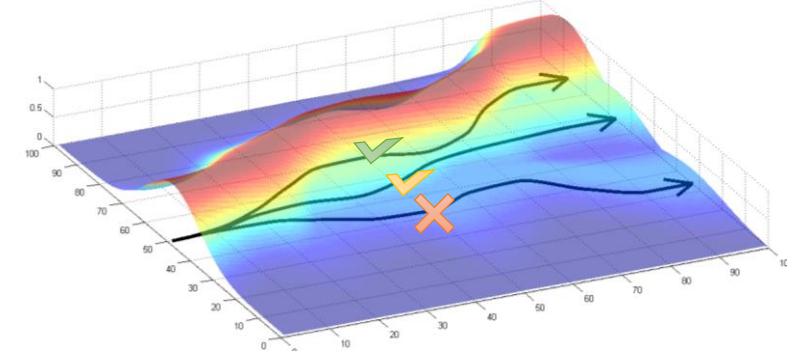
a convenient identity

$$\pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) = \nabla_\theta \pi_\theta(\tau)$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \nabla_\theta \log \pi_\theta(\tau) [\gamma(\tau) - b]$$

$$b = \frac{1}{N} \sum_{i=1}^N r(\tau)$$

but... are we *allowed* to do that??



$$E[\nabla_\theta \log \pi_\theta(\tau) b] = \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) b d\tau = \int \nabla_\theta \pi_\theta(\tau) b d\tau = b \nabla_\theta \int \pi_\theta(\tau) d\tau = b \nabla_\theta 1 = 0$$

subtracting a baseline is *unbiased* in expectation!

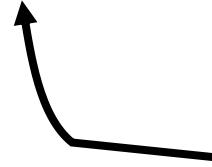
average reward is *not* the best baseline, but it's pretty good!

Policy gradient is on-policy

$$\theta^* = \arg \max_{\theta} J(\theta)$$

$$J(\theta) = E_{\tau \sim \pi_\theta(\tau)}[r(\tau)]$$

$$\nabla_{\theta} J(\theta) = E_{\tau \sim \pi_\theta(\tau)}[\nabla_{\theta} \log \pi_{\theta}(\tau) r(\tau)]$$



this is trouble...

- Neural networks change only a little bit with each gradient step
- On-policy learning can be extremely inefficient!

can't just skip this!

REINFORCE algorithm:

- 
1. sample $\{\tau^i\}$ from $\pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t)$ (run it on the robot)
 2. $\nabla_{\theta} J(\theta) \approx \sum_i (\sum_t \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_t^i | \mathbf{s}_t^i)) (\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i))$
 3. $\theta \leftarrow \theta + \alpha \nabla_{\theta} J(\theta)$

Off-policy learning & importance sampling

$$\theta^* = \arg \max_{\theta} J(\theta)$$

$$J(\theta) = E_{\tau \sim \pi_\theta(\tau)}[r(\tau)]$$

what if we don't have samples from $\pi_\theta(\tau)$?

(we have samples from some $\bar{\pi}(\tau)$ instead)

$$J(\theta) = E_{\tau \sim \bar{\pi}(\tau)} \left[\frac{\pi_\theta(\tau)}{\bar{\pi}(\tau)} r(\tau) \right]$$

$$\pi_\theta(\tau) = p(\mathbf{s}_1) \prod_{t=1}^T \pi_\theta(\mathbf{a}_t | \mathbf{s}_t) p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)$$

$$\frac{\pi_\theta(\tau)}{\bar{\pi}(\tau)} = \frac{p(\mathbf{s}_1) \prod_{t=1}^T \pi_\theta(\mathbf{a}_t | \mathbf{s}_t) p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)}{p(\mathbf{s}_1) \prod_{t=1}^T \bar{\pi}(\mathbf{a}_t | \mathbf{s}_t) p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)} = \frac{\prod_{t=1}^T \pi_\theta(\mathbf{a}_t | \mathbf{s}_t)}{\prod_{t=1}^T \bar{\pi}(\mathbf{a}_t | \mathbf{s}_t)}$$

importance sampling

$$E_{x \sim p(x)}[f(x)] = \int p(x)f(x)dx$$

$$= \int \frac{q(x)}{q(x)}p(x)f(x)dx$$

$$= \int q(x)\frac{p(x)}{q(x)}f(x)dx$$

$$= E_{x \sim q(x)} \left[\frac{p(x)}{q(x)}f(x) \right]$$

Deriving the policy gradient with IS

$$\theta^* = \arg \max_{\theta} J(\theta)$$

$$J(\theta) = E_{\tau \sim \pi_\theta(\tau)}[r(\tau)]$$

a convenient identity

$$\pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) = \nabla_\theta \pi_\theta(\tau)$$

can we estimate the value of some *new* parameters θ' ?

$$J(\theta') = E_{\tau \sim \pi_\theta(\tau)} \left[\frac{\pi_{\theta'}(\tau)}{\pi_\theta(\tau)} r(\tau) \right]$$

the only bit that depends on θ'

$$\nabla_{\theta'} J(\theta') = E_{\tau \sim \pi_\theta(\tau)} \left[\frac{\nabla_{\theta'} \pi_{\theta'}(\tau)}{\pi_\theta(\tau)} r(\tau) \right] = E_{\tau \sim \pi_\theta(\tau)} \left[\frac{\cancel{\pi_{\theta'}(\tau)}}{\cancel{\pi_\theta(\tau)}} \nabla_{\theta'} \log \pi_{\theta'}(\tau) r(\tau) \right]$$

now estimate locally, at $\theta = \theta'$: $\nabla_\theta J(\theta) = E_{\tau \sim \pi_\theta(\tau)} [\nabla_\theta \log \pi_\theta(\tau) r(\tau)]$

The off-policy policy gradient

$$\theta^* = \arg \max_{\theta} J(\theta)$$

$$J(\theta) = E_{\tau \sim \pi_\theta(\tau)}[r(\tau)]$$

$$\frac{\pi_{\theta'}(\tau)}{\pi_\theta(\tau)} = \frac{\prod_{t=1}^T \pi_{\theta'}(\mathbf{a}_t | \mathbf{s}_t)}{\prod_{t=1}^T \pi_\theta(\mathbf{a}_t | \mathbf{s}_t)}$$

$$\nabla_{\theta'} J(\theta') = E_{\tau \sim \pi_\theta(\tau)} \left[\frac{\pi_{\theta'}(\tau)}{\pi_\theta(\tau)} \nabla_{\theta'} \log \pi_{\theta'}(\tau) r(\tau) \right] \quad \text{when } \theta \neq \theta'$$

$$= E_{\tau \sim \pi_\theta(\tau)} \left[\left(\prod_{t=1}^T \frac{\pi_{\theta'}(\mathbf{a}_t | \mathbf{s}_t)}{\pi_\theta(\mathbf{a}_t | \mathbf{s}_t)} \right) \left(\sum_{t=1}^T \nabla_{\theta'} \log \pi_{\theta'}(\mathbf{a}_t | \mathbf{s}_t) \right) \left(\sum_{t=1}^T r(\mathbf{s}_t, \mathbf{a}_t) \right) \right] \text{ what about causality?}$$

$$= E_{\tau \sim \pi_\theta(\tau)} \left[\sum_{t=1}^T \nabla_{\theta'} \log \pi_{\theta'}(\mathbf{a}_t | \mathbf{s}_t) \underbrace{\left(\prod_{t'=1}^t \frac{\pi_{\theta'}(\mathbf{a}_{t'} | \mathbf{s}_{t'})}{\pi_\theta(\mathbf{a}_{t'} | \mathbf{s}_{t'})} \right)}_{\text{future actions don't affect current weight}} \left(\sum_{t'=t}^T r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \left(\prod_{t''=t}^{t'} \frac{\pi_{\theta'}(\mathbf{a}_{t''} | \mathbf{s}_{t''})}{\pi_\theta(\mathbf{a}_{t''} | \mathbf{s}_{t''})} \right) \right) \right]$$

future actions don't affect current weight

if we ignore this, we get
a policy iteration algorithm
(more on this in a later lecture)

A first-order approximation for IS (preview)

$$\nabla_{\theta'} J(\theta') = E_{\tau \sim \pi_\theta(\tau)} \left[\sum_{t=1}^T \nabla_{\theta'} \log \pi_{\theta'}(\mathbf{a}_t | \mathbf{s}_t) \left(\prod_{t'=1}^t \frac{\pi_{\theta'}(\mathbf{a}_{t'} | \mathbf{s}_{t'})}{\pi_\theta(\mathbf{a}_{t'} | \mathbf{s}_{t'})} \right) \left(\sum_{t'=t}^T r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \right) \right]$$

exponential in T ...

let's write the objective a bit differently...

on-policy policy gradient: $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \hat{Q}_{i,t}$

$(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \sim \pi_\theta(\mathbf{s}_t, \mathbf{a}_t)$

off-policy policy gradient: $\nabla_{\theta'} J(\theta') \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \frac{\pi_{\theta'}(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})}{\pi_\theta(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})} \nabla_{\theta'} \log \pi_{\theta'}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \hat{Q}_{i,t}$

This simplification is very
commonly used in practice!

$$= \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \cancel{\frac{\pi_{\theta'}(\mathbf{s}_{i,t})}{\pi_\theta(\mathbf{s}_{i,t})}} \frac{\pi_{\theta'}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t})}{\pi_\theta(\mathbf{a}_{i,t} | \mathbf{s}_{i,t})} \nabla_{\theta'} \log \pi_{\theta'}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \hat{Q}_{i,t}$$

ignore this part

Policy gradient with automatic differentiation

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \underline{\nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \hat{Q}_{i,t}}$$



pretty inefficient to compute these explicitly!

How can we compute policy gradients with automatic differentiation?

We need a graph such that its gradient is the policy gradient!

maximum likelihood: $\nabla_{\theta} J_{\text{ML}}(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t})$ $J_{\text{ML}}(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t})$

Just implement “pseudo-loss” as a weighted maximum likelihood:

$$\tilde{J}(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \hat{Q}_{i,t}$$



cross entropy (discrete) or squared error (Gaussian)

Policy gradient with automatic differentiation

Pseudocode example (with discrete actions):

Maximum likelihood:

```
# Given:  
# actions - (N*T) x Da tensor of actions  
# states - (N*T) x Ds tensor of states  
# Build the graph:  
logits = policy.predictions(states) # This should return (N*T) x Da tensor of action logits  
negative_likelihoods = tf.nn.softmax_cross_entropy_with_logits(labels=actions, logits=logits)  
loss = tf.reduce_mean(negative_likelihoods)  
gradients = loss.gradients(loss, variables)
```

Policy gradient with automatic differentiation

Pseudocode example (with discrete actions):

Policy gradient:

```
# Given:  
# actions - (N*T) x Da tensor of actions  
# states - (N*T) x Ds tensor of states  
# q_values - (N*T) x 1 tensor of estimated state-action values  
# Build the graph:  
logits = policy.predictions(states) # This should return (N*T) x Da tensor of action logits  
negative_likelihoods = tf.nn.softmax_cross_entropy_with_logits(labels=actions, logits=logits)  
weighted_negative_likelihoods = tf.multiply(negative_likelihoods, q_values)  
loss = tf.reduce_mean(weighted_negative_likelihoods)  
gradients = loss.gradients(loss, variables)
```

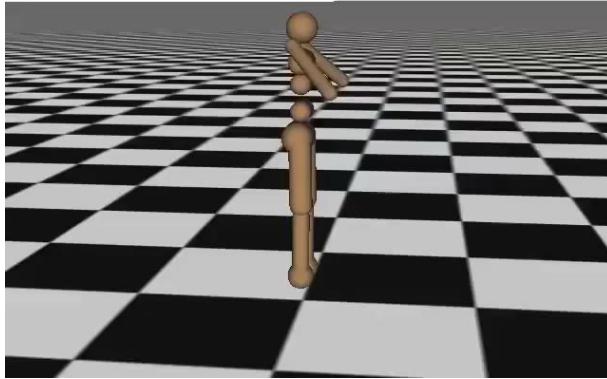
$$\tilde{J}(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \log \pi_\theta(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \hat{Q}_{i,t}$$

Policy gradient in practice

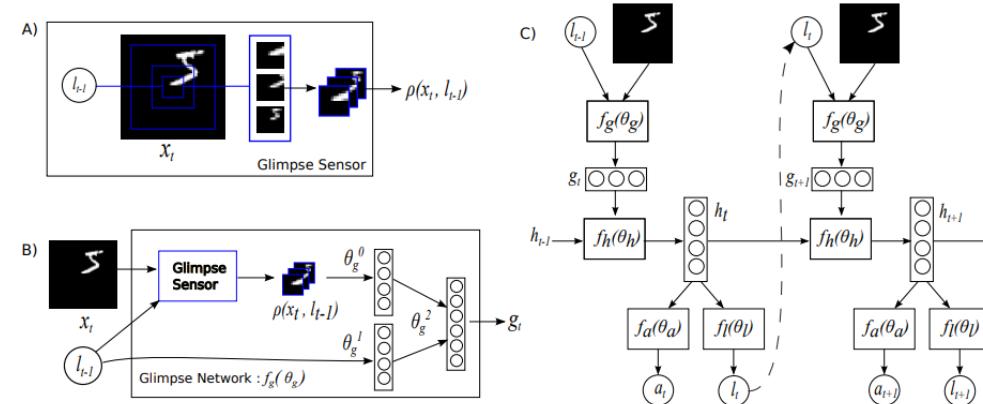
- Remember that the gradient has high variance
 - This isn't the same as supervised learning!
 - Gradients will be really noisy!
- Consider using much larger batches
- Tweaking learning rates is very hard
 - Adaptive step size rules like ADAM can be OK-ish
 - We'll learn about policy gradient-specific learning rate adjustment methods later!

What are policy gradients used for?

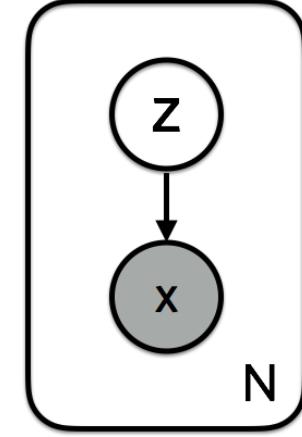
Iteration 0



learning for control



“hard” attention: network selects where to look
(Mnih et al. Recurrent Models of Visual Attention)



Discrete latent variable models
(we'll learn about these later!)

Long story short: policy gradient (REINFORCE) can be used in any setting where we have to **differentiate** through a stochastic but non-differentiable operation

Review

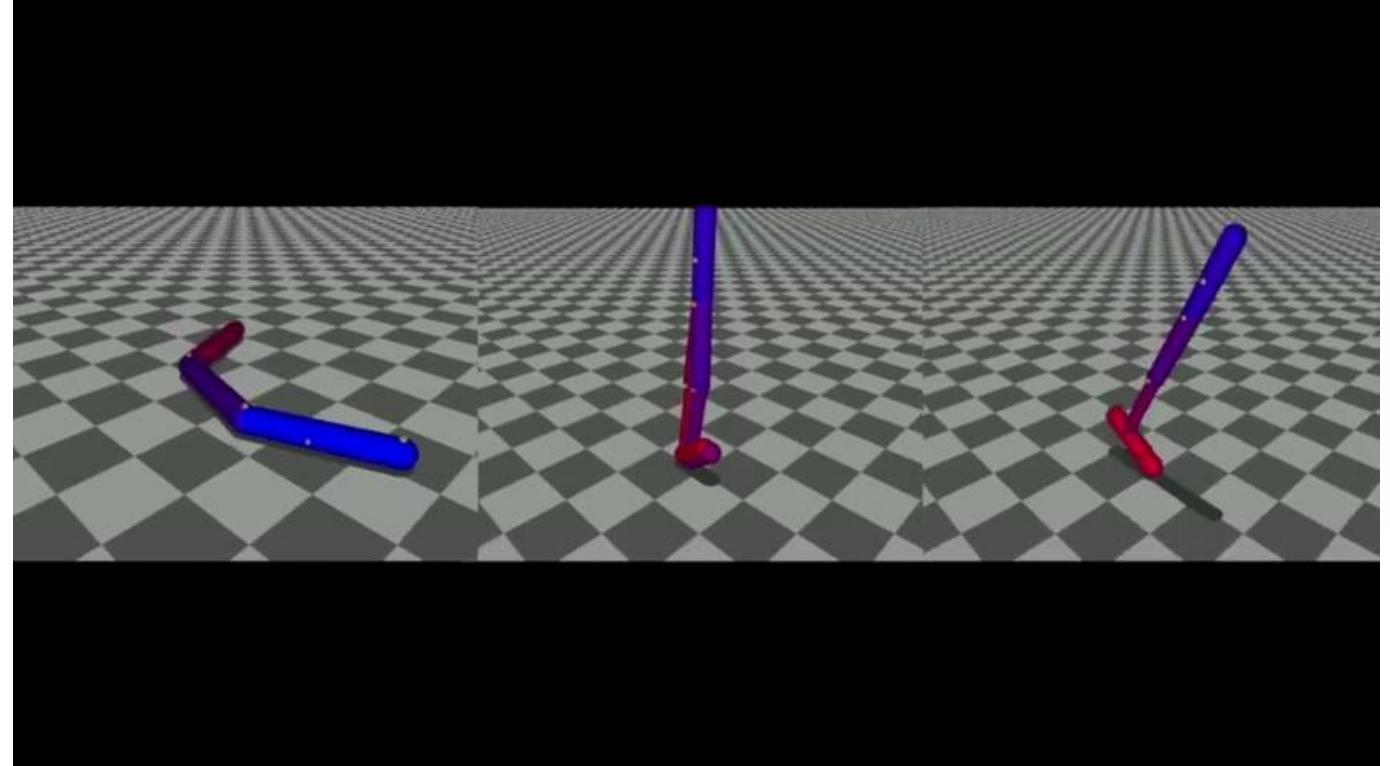
- Evaluating the RL objective
 - Generate samples
- Evaluating the policy gradient
 - Log-gradient trick
 - Generate samples
- Policy gradient is on-policy
- Can derive off-policy variant
 - Use importance sampling
 - Exponential scaling in T
 - Can ignore state portion (approximation)
- Can implement with automatic differentiation – need to know what to backpropagate
- Practical considerations: batch size, learning rates, optimizers

REINFORCE algorithm:

- 
1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t | \mathbf{s}_t)$ (run it on the robot)
 2. $\nabla_\theta J(\theta) \approx \sum_i (\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i | \mathbf{s}_t^i)) (\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i))$
 3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

Example: trust region policy optimization

- Natural gradient with automatic step adjustment
- Discrete and continuous actions



Policy gradients suggested readings

- Classic papers
 - Williams (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning: introduces REINFORCE algorithm
 - Baxter & Bartlett (2001). Infinite-horizon policy-gradient estimation: temporally decomposed policy gradient (not the first paper on this! see actor-critic section later)
 - Peters & Schaal (2008). Reinforcement learning of motor skills with policy gradients: very accessible overview of optimal baselines and natural gradient
- Deep reinforcement learning policy gradient papers
 - Levine & Koltun (2013). Guided policy search: deep RL with importance sampled policy gradient (unrelated to later discussion of guided policy search)
 - Schulman, L., Moritz, Jordan, Abbeel (2015). Trust region policy optimization: deep RL with natural policy gradient and adaptive step size
 - Schulman, Wolski, Dhariwal, Radford, Klimov (2017). Proximal policy optimization algorithms: deep RL with importance sampled policy gradient

Reinforcement Learning

Designing, Visualizing and Understanding Deep Neural Networks

CS W182/282A

Instructor: Sergey Levine
UC Berkeley



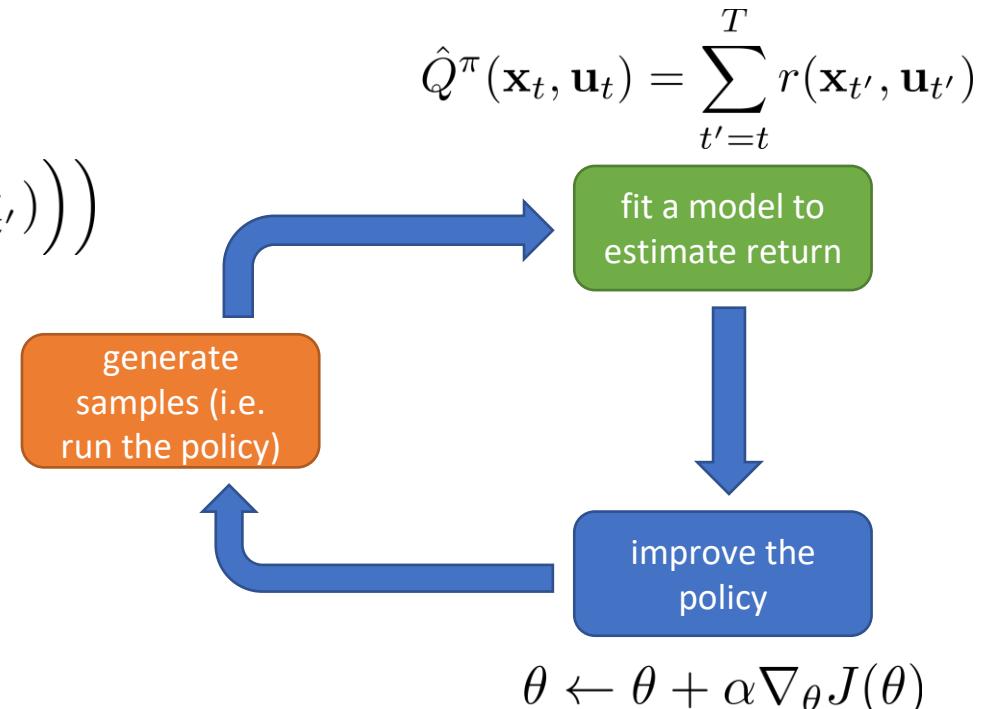
Recap: policy gradients

REINFORCE algorithm:

1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t | \mathbf{s}_t)$ (run the policy)
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i | \mathbf{s}_t^i) \left(\sum_{t'=t}^T r(\mathbf{s}_{t'}^i, \mathbf{a}_{t'}^i) \right) \right)$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \hat{Q}_{i,t}^\pi$$

“reward to go”



Improving the policy gradient

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \left(\underbrace{\sum_{t'=1}^T r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'})}_{\text{"reward to go"}} \right)$$

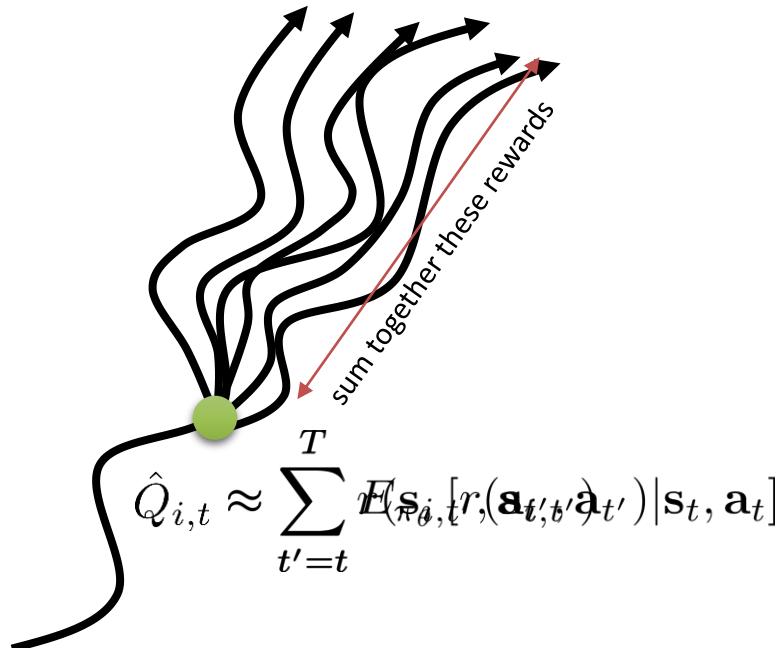
“reward to go”

$$\hat{Q}_{i,t}$$

$\hat{Q}_{i,t}$: estimate of expected reward if we take action $\mathbf{a}_{i,t}$ in state $\mathbf{s}_{i,t}$
can we get a better estimate?

$Q(\mathbf{s}_t, \mathbf{a}_t) = \sum_{t'=t}^T E_{\pi_{\theta}} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t, \mathbf{a}_t]$: true *expected* reward-to-go

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) Q(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})$$



What about the baseline?

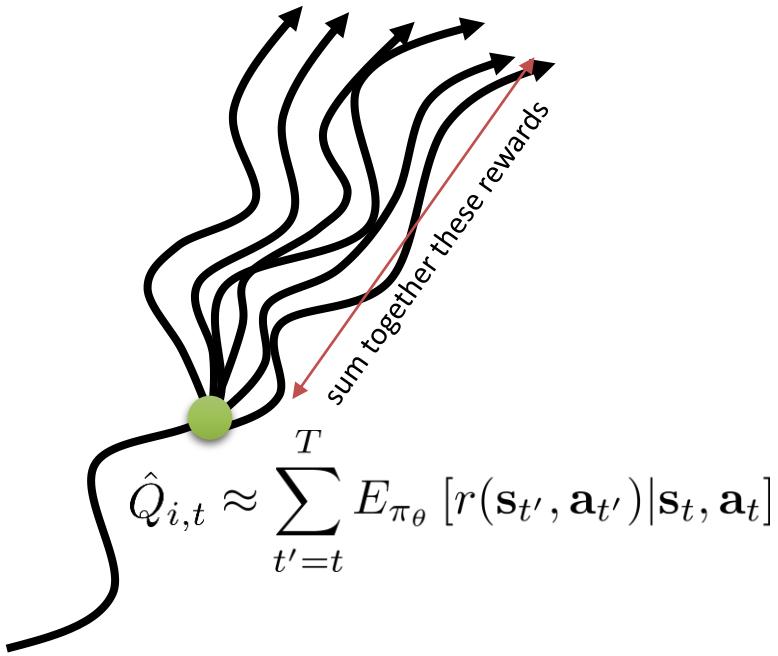
$Q(\mathbf{s}_t, \mathbf{a}_t) = \sum_{t'=t}^T E_{\pi_\theta} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t, \mathbf{a}_t]$: true *expected* reward-to-go

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) (Q(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) - V(\mathbf{s}_{i,t}))$$

$$b_t = \text{average}_{i=1}^N Q(\text{reward}, \mathbf{a}_{i,t})$$

average what?

$$V(\mathbf{s}_t) = E_{\mathbf{a}_t \sim \pi_\theta(\mathbf{a}_t | \mathbf{s}_t)} [Q(\mathbf{s}_t, \mathbf{a}_t)]$$



State & state-action value functions

$$Q^\pi(s_t, a_t) = \sum_{t'=t}^T E_{\pi_\theta}[r(s_{t'}, a_{t'}) | s_t, a_t]: \text{total reward from taking } a_t \text{ in } s_t$$

fit Q^π , V^π , or A^π

$$V^\pi(s_t) = E_{a_t \sim \pi_\theta(a_t | s_t)}[Q^\pi(s_t, a_t)]: \text{total reward from } s_t$$

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t): \text{how much better } a_t \text{ is}$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_{i,t} | s_{i,t}) A^\pi(s_{i,t}, a_{i,t})$$



$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_{i,t} | s_{i,t}) \underbrace{\left(\sum_{t'=1}^T r(s_{i,t'}, a_{i,t'}) - b \right)}_{\text{unbiased, but high variance single-sample estimate}}$$

unbiased, but high variance single-sample estimate

Value function fitting

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) = \sum_{t'=t}^T E_{\pi_\theta} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t, \mathbf{a}_t]$$

$$V^\pi(\mathbf{s}_t) = E_{\mathbf{a}_t \sim \pi_\theta(\mathbf{a}_t | \mathbf{s}_t)} [Q^\pi(\mathbf{s}_t, \mathbf{a}_t)]$$

$$A^\pi(\mathbf{s}_t, \mathbf{a}_t) = Q^\pi(\mathbf{s}_t, \mathbf{a}_t) - V^\pi(\mathbf{s}_t)$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) A^\pi(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})$$

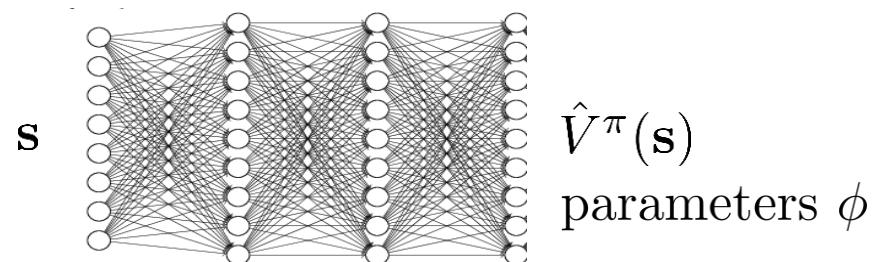
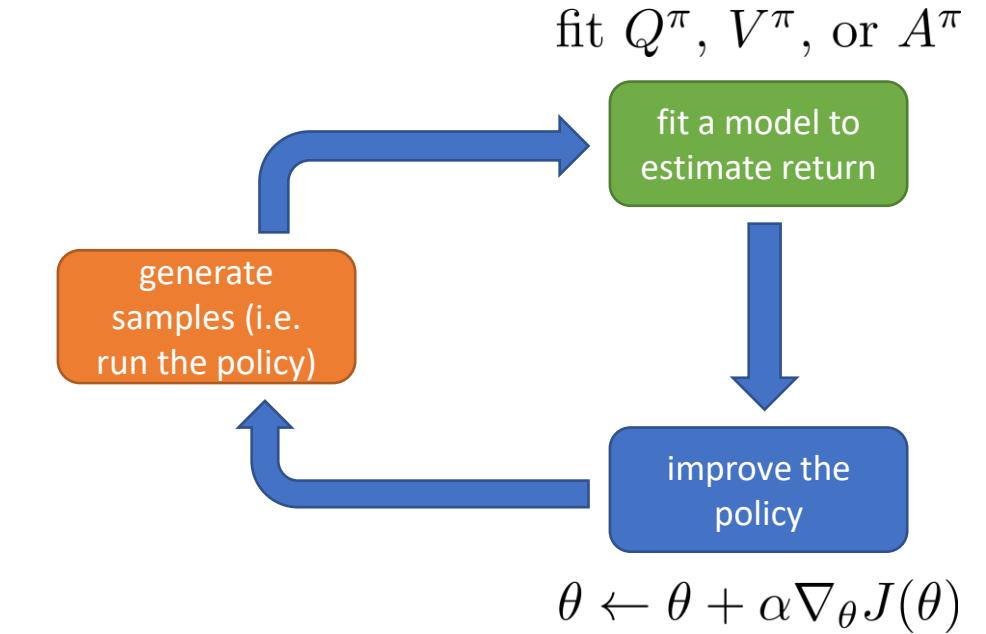
fit *what* to *what*?

Q^π, V^π, A^π ?

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) \approx \underbrace{\sum_{t'=t}^T E_{\pi_\theta} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) + \mathbb{E}_{\mathbf{s}_{t+1} \sim p(\cdot | \mathbf{s}_t, \mathbf{a}_t)} [V^\pi(\mathbf{s}_{t+1})]]}_{\text{fit } V^\pi(\mathbf{s}_t)}$$

$$A^\pi(\mathbf{s}_t, \mathbf{a}_t) \approx r(\mathbf{s}_t, \mathbf{a}_t) + V^\pi(\mathbf{s}_{t+1}) - V^\pi(\mathbf{s}_t)$$

let's just fit $V^\pi(\mathbf{s})$!



Policy evaluation

$$V^\pi(\mathbf{s}_t) = \sum_{t'=t}^T E_{\pi_\theta} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t]$$

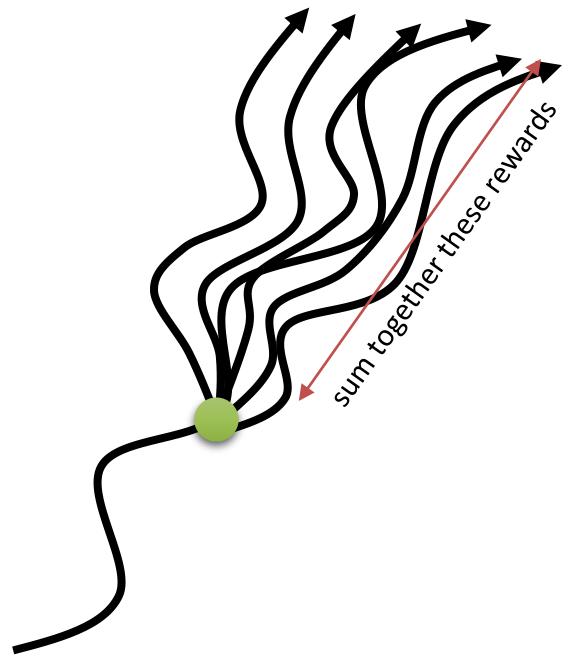
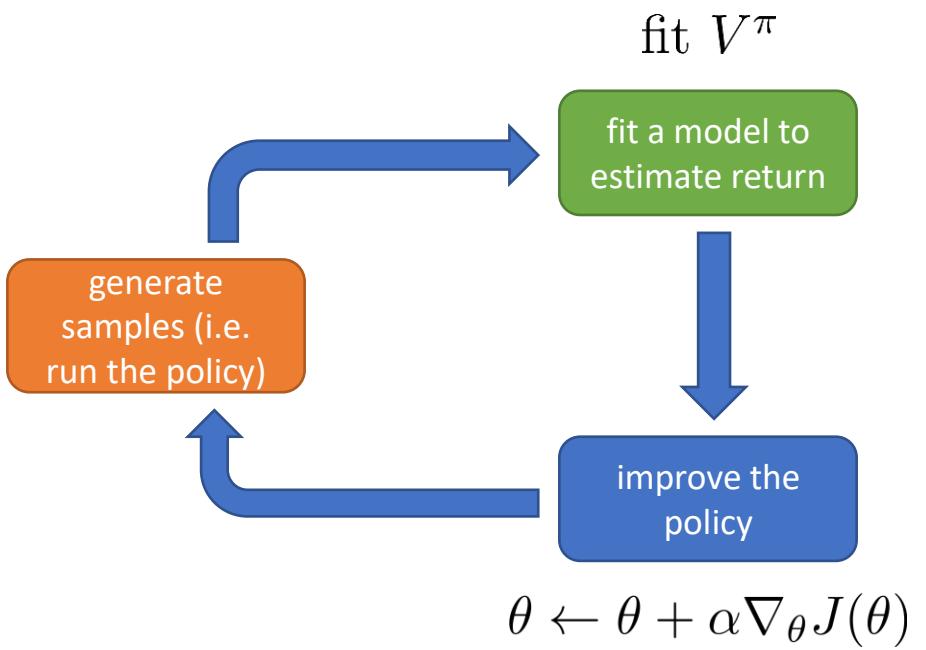
$$J(\theta) = E_{\mathbf{s}_1 \sim p(\mathbf{s}_1)} [V^\pi(\mathbf{s}_1)]$$

how can we perform policy evaluation?

Monte Carlo policy evaluation (this is what policy gradient does)

$$V^\pi(\mathbf{s}_t) \approx \sum_{t'=t}^T r(\mathbf{s}_{t'}, \mathbf{a}_{t'})$$

$$V^\pi(\mathbf{s}_t) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t'=t}^T r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \quad (\text{requires us to reset the simulator})$$



Monte Carlo evaluation with function approximation

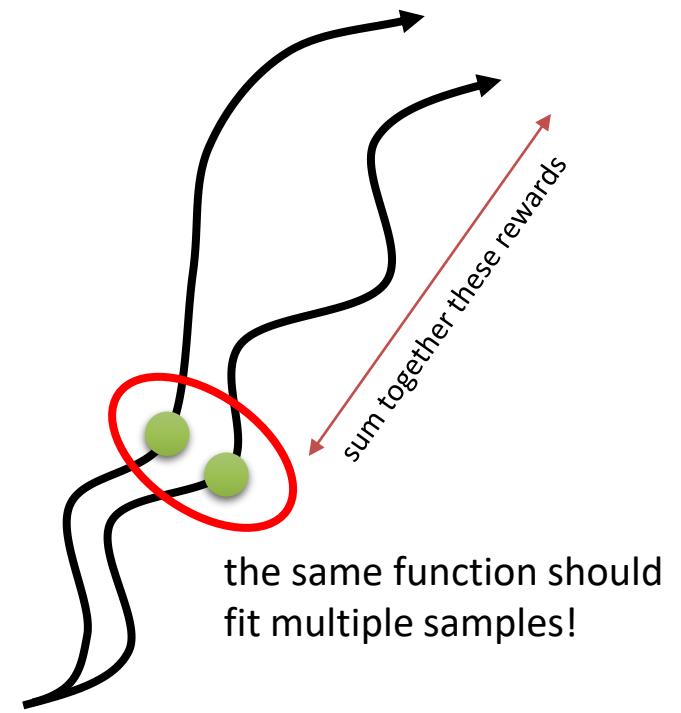
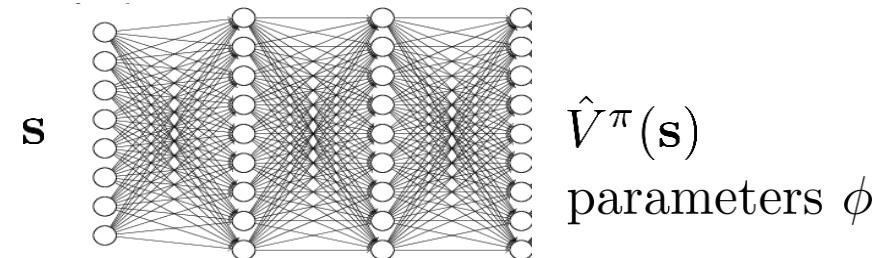
$$V^\pi(\mathbf{s}_t) \approx \sum_{t'=t}^T r(\mathbf{s}_{t'}, \mathbf{a}_{t'})$$

not as good as this: $V^\pi(\mathbf{s}_t) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t'=t}^T r(\mathbf{s}_{t'}, \mathbf{a}_{t'})$

but still pretty good!

training data: $\left\{ \left(\mathbf{s}_{i,t}, \underbrace{\sum_{t'=t}^T r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'})}_{y_{i,t}} \right) \right\}$

supervised regression: $\mathcal{L}(\phi) = \frac{1}{2} \sum_i \left\| \hat{V}_\phi^\pi(\mathbf{s}_i) - y_i \right\|^2$



Can we do better?

ideal target: $y_{i,t} = \sum_{t'=t}^T E_{\pi_\theta} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_{i,t}] \approx r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) + \sum_{t'=t+1}^T \underbrace{E_{\pi_\theta} [r(\mathbf{s}_{t',t+1}, \mathbf{a}_{t'}) | \mathbf{s}_{i,t+1}]}_{\hat{V}_\phi^\pi(\mathbf{s}_{i,t+1})}$

Monte Carlo target: $y_{i,t} = \sum_{t'=t}^T r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'})$

directly use previous fitted value function!

training data: $\left\{ \left(\mathbf{s}_{i,t}, \underbrace{r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) + \hat{V}_\phi^\pi(\mathbf{s}_{i,t+1})}_{y_{i,t}} \right) \right\}$

supervised regression: $\mathcal{L}(\phi) = \frac{1}{2} \sum_i \left\| \hat{V}_\phi^\pi(\mathbf{s}_i) - y_i \right\|^2$

sometimes referred to as a “bootstrapped” estimate

Policy evaluation examples

TD-Gammon, Gerald Tesauro 1992

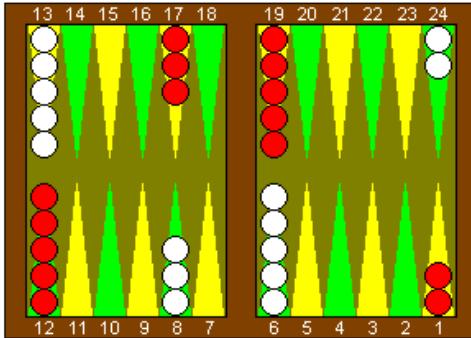


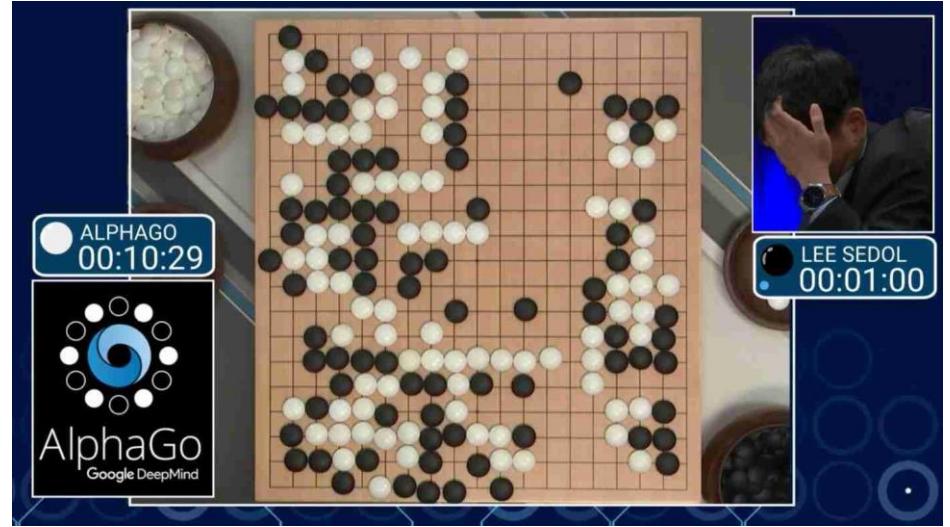
Figure 2. An illustration of the normal opening position in backgammon. TD-Gammon has sparked a near-universal conversion in the way experts play certain opening rolls. For example, with an opening roll of 4-1, most players have now switched from the traditional move of 13-9, 6-5, to TD-Gammon's preference, 13-9, 24-23. TD-Gammon's analysis is given in Table 2.

reward: game outcome

value function $\hat{V}_\phi^\pi(\mathbf{s}_t)$:

expected outcome given board state

AlphaGo, Silver et al. 2016



reward: game outcome

value function $\hat{V}_\phi^\pi(\mathbf{s}_t)$:

expected outcome given board state

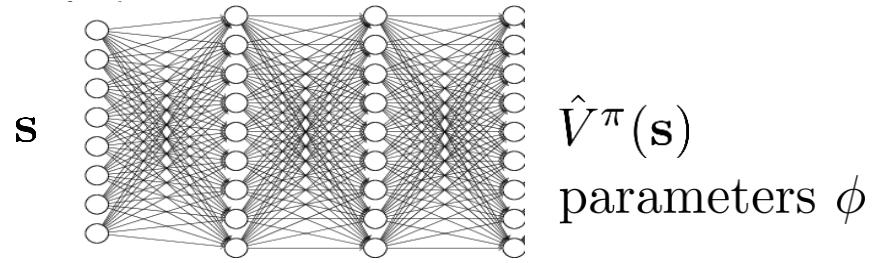
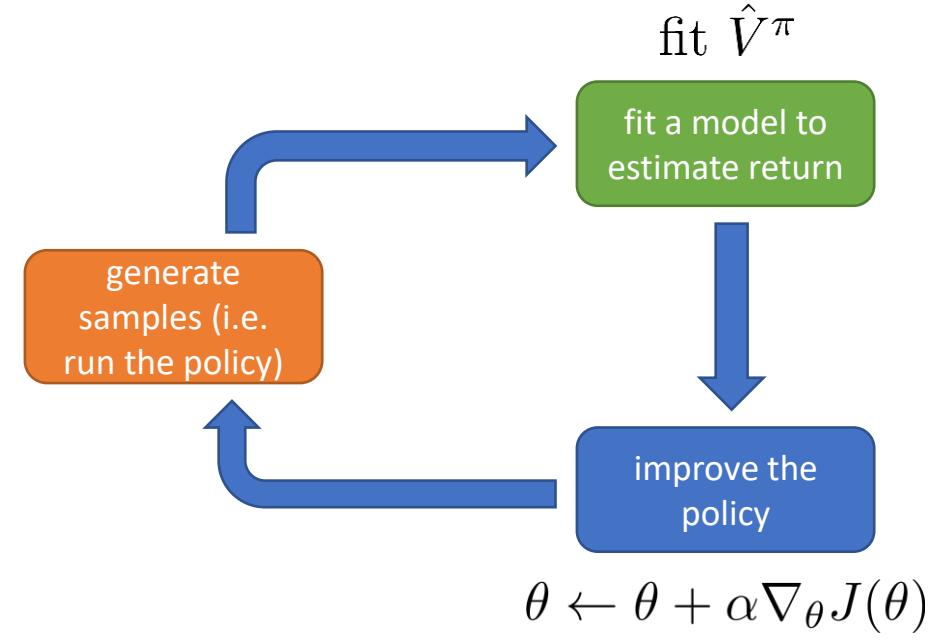
An actor-critic algorithm

batch actor-critic algorithm:

1. sample $\{\mathbf{s}_i, \mathbf{a}_i\}$ from $\pi_\theta(\mathbf{a}|\mathbf{s})$ (run it on the robot)
2. fit $\hat{V}_\phi^\pi(\mathbf{s})$ to sampled reward sums
3. evaluate $\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \hat{V}_\phi^\pi(\mathbf{s}'_i) - \hat{V}_\phi^\pi(\mathbf{s}_i)$
4. $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i|\mathbf{s}_i) \hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i)$
5. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

$$V^\pi(\mathbf{s}_{y,t}) \approx \sum_{t' \neq t}^T E_{\pi_\theta} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t]$$

$$\mathcal{L}(\phi) = \frac{1}{2} \sum_i \left\| \hat{V}_\phi^\pi(\mathbf{s}_i) - y_i \right\|^2$$



$$V^\pi(\mathbf{s}_t) = \sum_{t'=t}^T E_{\pi_\theta} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t]$$

Aside: discount factors

$$y_{i,t} \approx r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) + \hat{V}_\phi^\pi(\mathbf{s}_{i,t+1})$$

$$\mathcal{L}(\phi) = \frac{1}{2} \sum_i \left\| \hat{V}_\phi^\pi(\mathbf{s}_i) - y_i \right\|^2$$

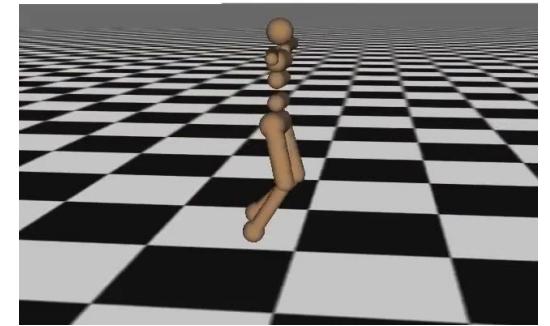
what if T (episode length) is ∞ ?

\hat{V}_ϕ^π can get infinitely large in many cases



episodic tasks

Iteration 2000



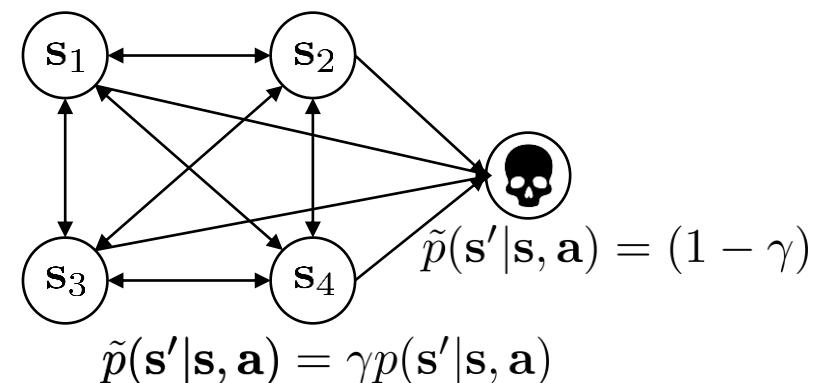
continuous/cyclical tasks

simple trick: better to get rewards sooner than later

$$y_{i,t} \approx r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) + \gamma \hat{V}_\phi^\pi(\mathbf{s}_{i,t+1})$$

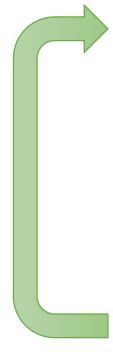
↑
discount factor $\gamma \in [0, 1]$ (0.99 works well)

γ changes the MDP:

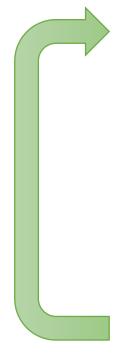


Actor-critic algorithms (with discount)

batch actor-critic algorithm:

- 
1. sample $\{\mathbf{s}_i, \mathbf{a}_i\}$ from $\pi_\theta(\mathbf{a}|\mathbf{s})$ (run it on the robot)
 2. fit $\hat{V}_\phi^\pi(\mathbf{s})$ to sampled reward sums
 3. evaluate $\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \hat{V}_\phi^\pi(\mathbf{s}'_i) - \hat{V}_\phi^\pi(\mathbf{s}_i)$
 4. $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i|\mathbf{s}_i) \hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i)$
 5. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

online actor-critic algorithm:

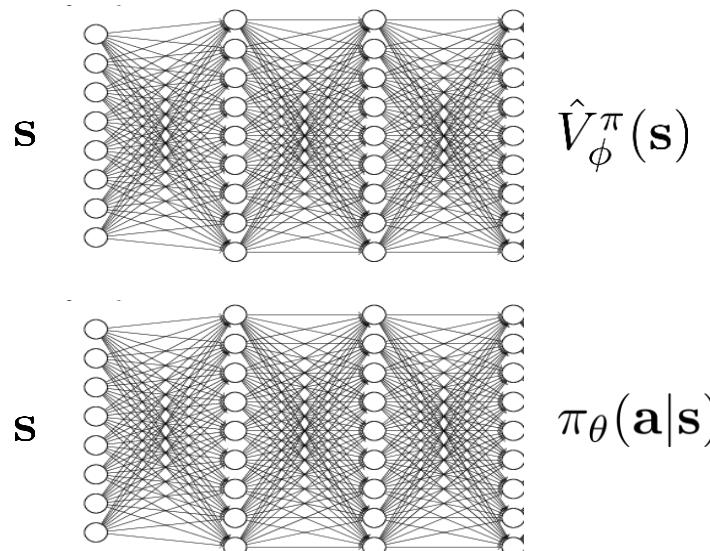
- 
1. take action $\mathbf{a} \sim \pi_\theta(\mathbf{a}|\mathbf{s})$, get $(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$
 2. update \hat{V}_ϕ^π using target $r + \gamma \hat{V}_\phi^\pi(\mathbf{s}')$
 3. evaluate $\hat{A}^\pi(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma \hat{V}_\phi^\pi(\mathbf{s}') - \hat{V}_\phi^\pi(\mathbf{s})$
 4. $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(\mathbf{a}|\mathbf{s}) \hat{A}^\pi(\mathbf{s}, \mathbf{a})$
 5. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

Architecture design

online actor-critic algorithm:

1. take action $\mathbf{a} \sim \pi_\theta(\mathbf{a}|\mathbf{s})$, get $(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$
2. update \hat{V}_ϕ^π using target $r + \gamma \hat{V}_\phi^\pi(\mathbf{s}')$
3. evaluate $\hat{A}^\pi(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma \hat{V}_\phi^\pi(\mathbf{s}') - \hat{V}_\phi^\pi(\mathbf{s})$
4. $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(\mathbf{a}|\mathbf{s}) \hat{A}^\pi(\mathbf{s}, \mathbf{a})$
5. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

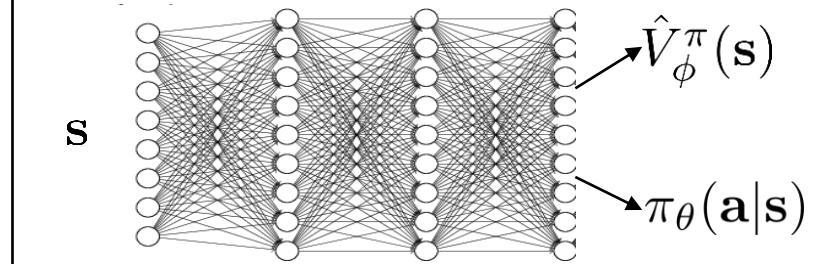
two network design



+ simple & stable

- no shared features between actor & critic

shared network design



Can we use just a value function?

Can we omit policy gradient completely?

$A^\pi(s_t, a_t)$: how much better is a_t than the average action according to π

$\arg \max_{a_t} A^\pi(s_t, a_t)$: best action from s_t , if we then follow π

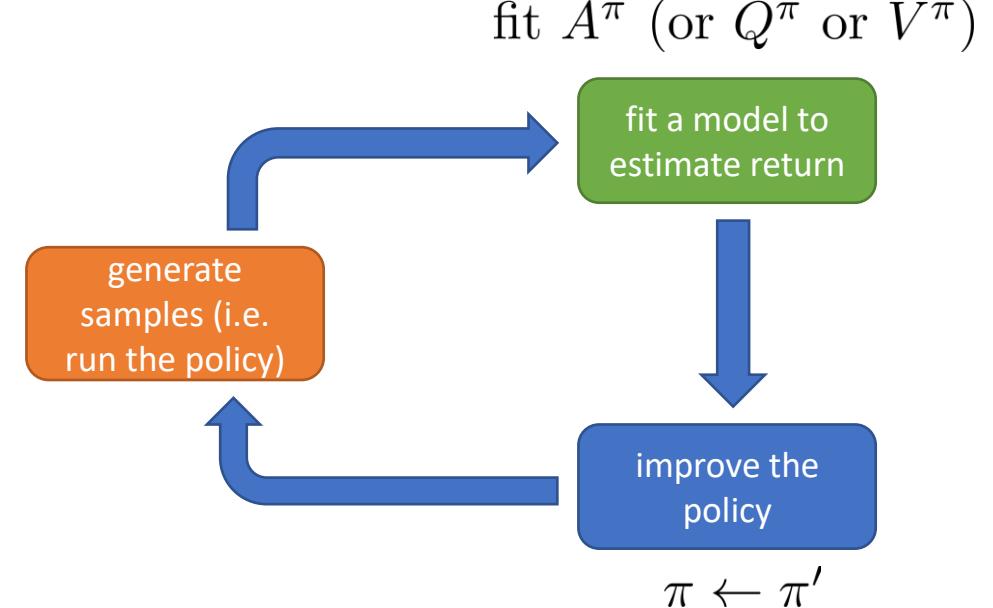
at *least* as good as any $a_t \sim \pi(a_t|s_t)$

regardless of what $\pi(a_t|s_t)$ is!

forget policies, let's just do this!

$$\pi'(a_t|s_t) = \begin{cases} 1 & \text{if } a_t = \arg \max_{a_t} A^\pi(s_t, a_t) \\ 0 & \text{otherwise} \end{cases}$$

as good as π
(probably better)



Policy iteration

High level idea:

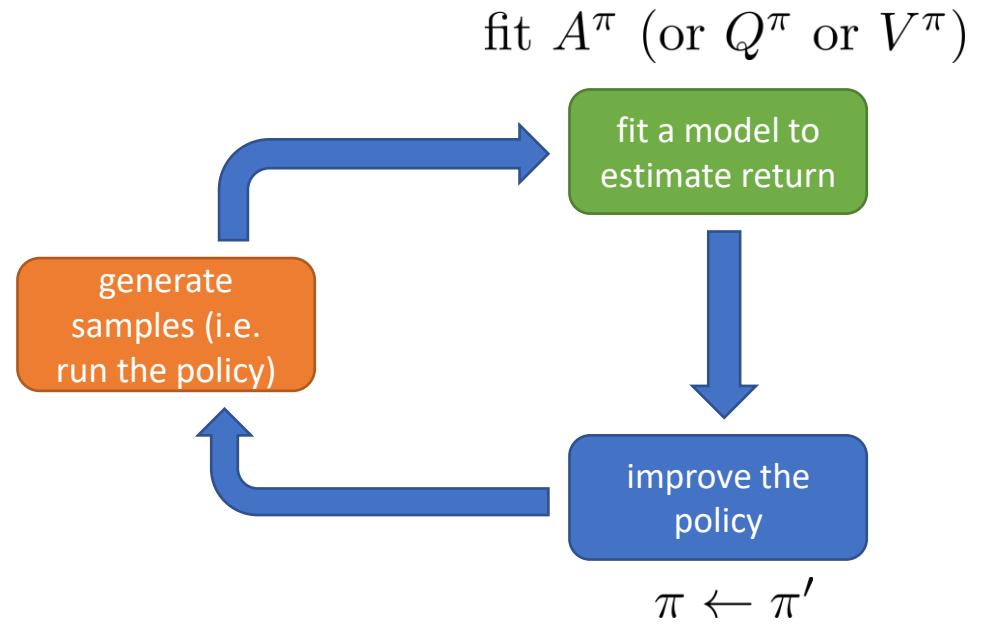
policy iteration algorithm:

- 
1. evaluate $A^\pi(\mathbf{s}, \mathbf{a})$ ← how to do this?
 2. set $\pi \leftarrow \pi'$

$$\pi'(\mathbf{a}_t | \mathbf{s}_t) = \begin{cases} 1 & \text{if } \mathbf{a}_t = \arg \max_{\mathbf{a}_t} A^\pi(\mathbf{s}_t, \mathbf{a}_t) \\ 0 & \text{otherwise} \end{cases}$$

as before: $A^\pi(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma E[V^\pi(\mathbf{s}')]$ – $V^\pi(\mathbf{s})$

let's evaluate $V^\pi(\mathbf{s})$!



Dynamic programming

Let's assume we know $p(\mathbf{s}'|\mathbf{s}, \mathbf{a})$, and \mathbf{s} and \mathbf{a} are both discrete (and small)

0.2	0.3	0.4	0.3
0.3	0.3	0.5	0.3
0.4	0.4	0.6	0.4
0.5	0.5	0.7	0.5

16 states, 4 actions per state
can store full $V^\pi(\mathbf{s})$ in a table!
 \mathcal{T} is $16 \times 16 \times 4$ tensor

bootstrapped update: $V^\pi(\mathbf{s}) \leftarrow E_{\mathbf{a} \sim \pi(\mathbf{a}|\mathbf{s})}[r(\mathbf{s}, \mathbf{a}) + \gamma E_{\mathbf{s}' \sim p(\mathbf{s}'|\mathbf{s}, \mathbf{a})}[V^\pi(\mathbf{s}')]]$



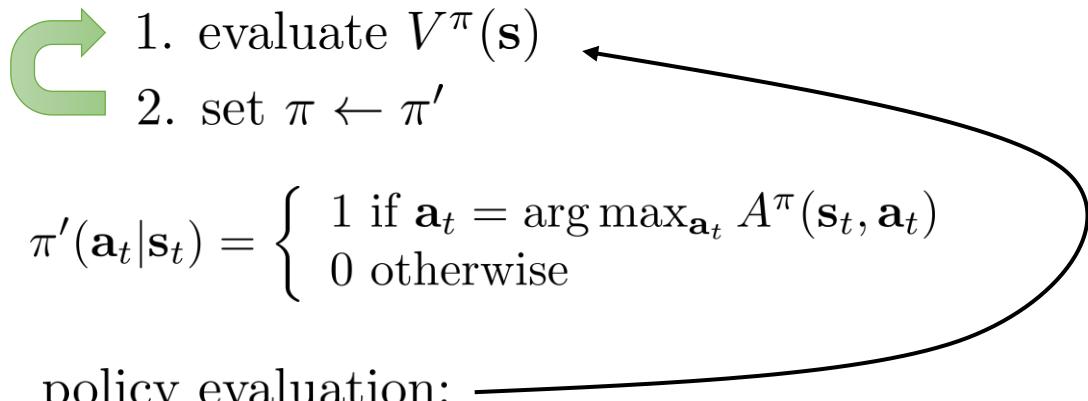
just use the current estimate here

$$\pi'(\mathbf{a}_t|\mathbf{s}_t) = \begin{cases} 1 & \text{if } \mathbf{a}_t = \arg \max_{\mathbf{a}_t} A^\pi(\mathbf{s}_t, \mathbf{a}_t) \\ 0 & \text{otherwise} \end{cases} \longrightarrow \text{deterministic policy } \pi(\mathbf{s}) = \mathbf{a}$$

simplified: $V^\pi(\mathbf{s}) \leftarrow r(\mathbf{s}, \pi(\mathbf{s})) + \gamma E_{\mathbf{s}' \sim p(\mathbf{s}'|\mathbf{s}, \pi(\mathbf{s}))}[V^\pi(\mathbf{s}')]$

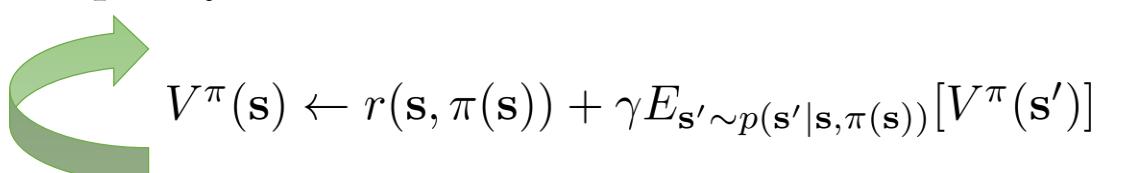
Policy iteration with dynamic programming

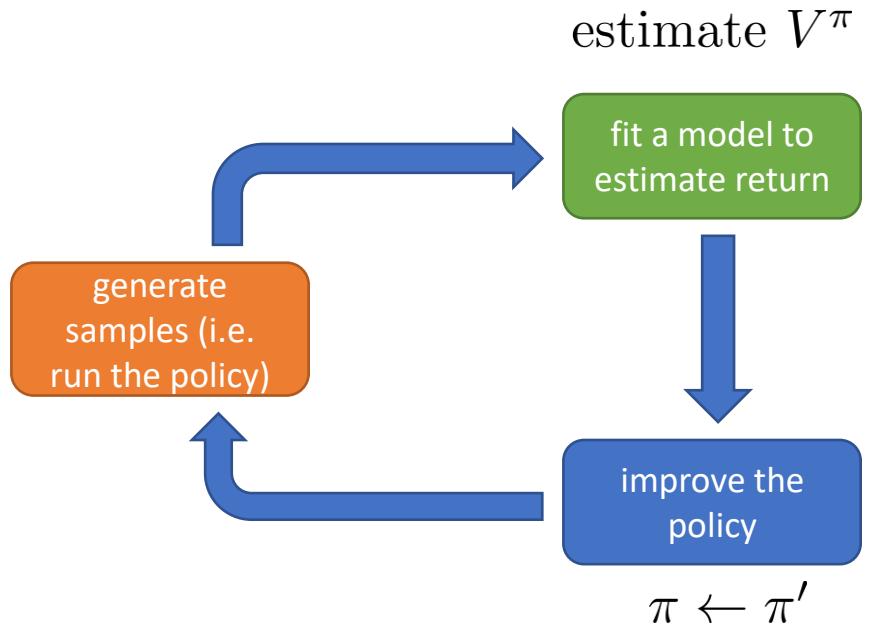
policy iteration:

- 
1. evaluate $V^\pi(\mathbf{s})$
 2. set $\pi \leftarrow \pi'$

$$\pi'(\mathbf{a}_t | \mathbf{s}_t) = \begin{cases} 1 & \text{if } \mathbf{a}_t = \arg \max_{\mathbf{a}_t} A^\pi(\mathbf{s}_t, \mathbf{a}_t) \\ 0 & \text{otherwise} \end{cases}$$

policy evaluation:


$$V^\pi(\mathbf{s}) \leftarrow r(\mathbf{s}, \pi(\mathbf{s})) + \gamma E_{\mathbf{s}' \sim p(\mathbf{s}' | \mathbf{s}, \pi(\mathbf{s}))}[V^\pi(\mathbf{s}')]$$



0.2	0.3	0.4	0.3
0.3	0.3	0.5	0.3
0.4	0.4	0.6	0.4
0.5	0.5	0.7	0.5

16 states, 4 actions per state

can store full $V^\pi(\mathbf{s})$ in a table!

\mathcal{T} is $16 \times 16 \times 4$ tensor

Even simpler dynamic programming

$$\pi'(\mathbf{a}_t | \mathbf{s}_t) = \begin{cases} 1 & \text{if } \mathbf{a}_t = \arg \max_{\mathbf{a}_t} A^\pi(\mathbf{s}_t, \mathbf{a}_t) \\ 0 & \text{otherwise} \end{cases}$$

$$A^\pi(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma E[V^\pi(\mathbf{s}')] - V^\pi(\mathbf{s})$$

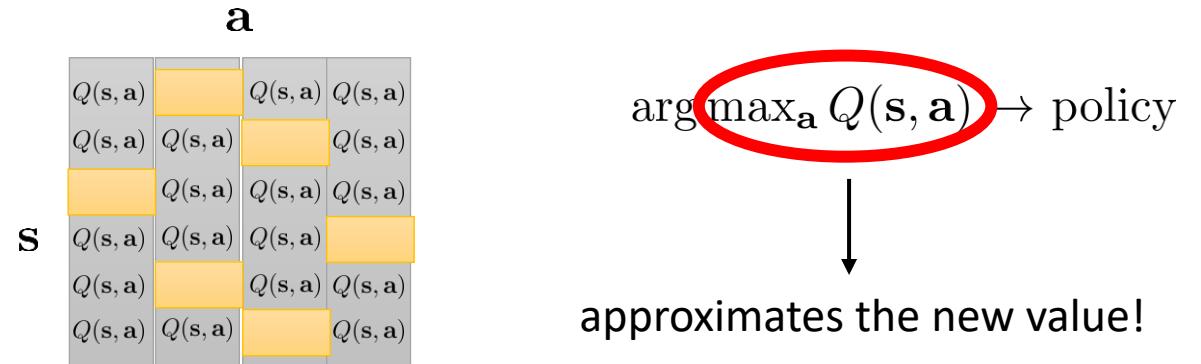
$$\arg \max_{\mathbf{a}_t} A^\pi(\mathbf{s}_t, \mathbf{a}_t) = \arg \max_{\mathbf{a}_t} Q^\pi(\mathbf{s}_t, \mathbf{a}_t)$$

$$Q^\pi(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma E[V^\pi(\mathbf{s}')]$$
 (a bit simpler)

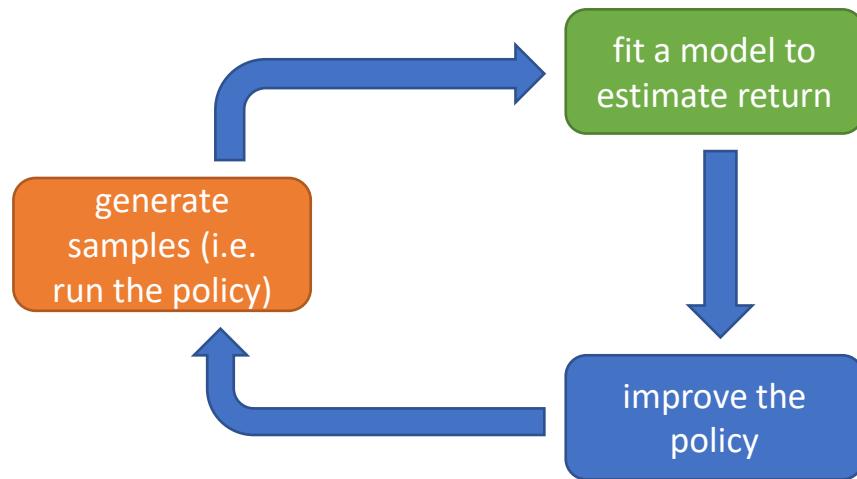
skip the policy and compute values directly!

value iteration algorithm:

- ➡ 1. set $Q(\mathbf{s}, \mathbf{a}) \leftarrow r(\mathbf{s}, \mathbf{a}) + \gamma E[V(\mathbf{s}')] \quad$
- 2. set $V(\mathbf{s}) \leftarrow \max_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a}) \quad$



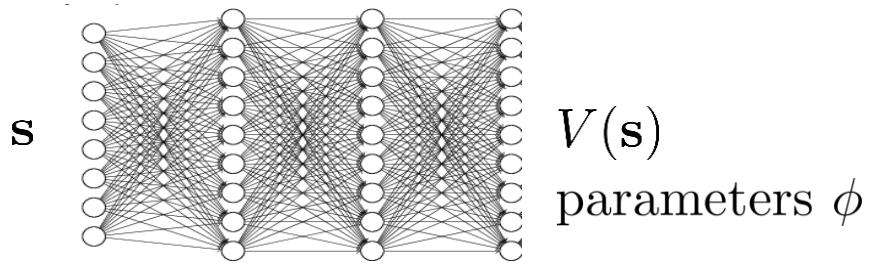
$$Q^\pi(\mathbf{s}, \mathbf{a}) \leftarrow r(\mathbf{s}, \mathbf{a}) + \gamma E_{\mathbf{s}' \sim p(\mathbf{s}'|\mathbf{s}, \mathbf{a})}[V^\pi(\mathbf{s}')]$$



Fitted value iteration

how do we represent $V(\mathbf{s})$?

big table, one entry for each discrete \mathbf{s}
neural net function $V : \mathcal{S} \rightarrow \mathbb{R}$



$$\mathbf{s} = 0 : V(\mathbf{s}) = 0.2$$

$$\mathbf{s} = 1 : V(\mathbf{s}) = 0.3$$

$$\mathbf{s} = 2 : V(\mathbf{s}) = 0.5$$



curse of
dimensionality

$$|\mathcal{S}| = (255^3)^{200 \times 200}$$

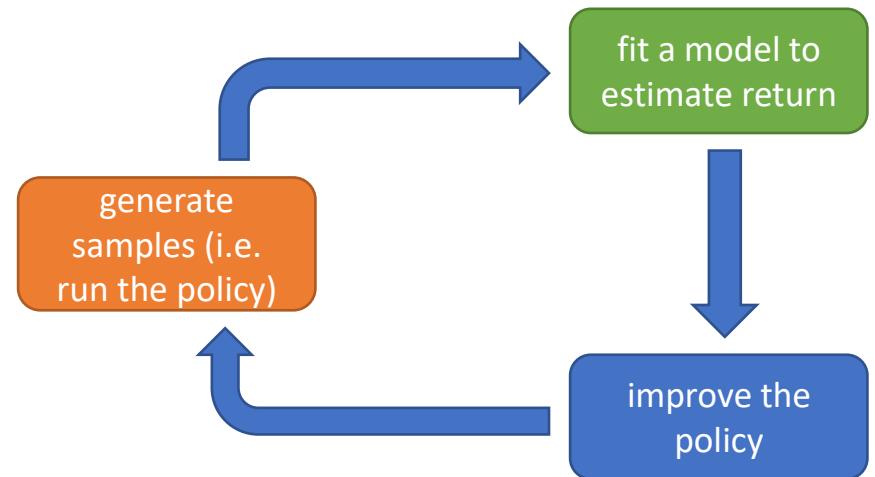
(more than atoms in the universe)

$$Q^\pi(\mathbf{s}, \mathbf{a}) \leftarrow r(\mathbf{s}, \mathbf{a}) + \gamma E_{\mathbf{s}' \sim p(\mathbf{s}' | \mathbf{s}, \mathbf{a})}[V^\pi(\mathbf{s}')]$$

$$\mathcal{L}(\phi) = \frac{1}{2} \left\| V_\phi(\mathbf{s}) - \max_{\mathbf{a}} Q^\pi(\mathbf{s}, \mathbf{a}) \right\|^2$$

fitted value iteration algorithm:

- C 1. set $\mathbf{y}_i \leftarrow \max_{\mathbf{a}_i} (r(\mathbf{s}_i, \mathbf{a}_i) + \gamma E[V_\phi(\mathbf{s}'_i)])$
2. set $\phi \leftarrow \arg \min_\phi \frac{1}{2} \sum_i \|V_\phi(\mathbf{s}_i) - \mathbf{y}_i\|^2$



$$V^\pi(\mathbf{s}) \leftarrow \max_{\mathbf{a}} Q^\pi(\mathbf{s}, \mathbf{a})$$

What if we don't know the transition dynamics?

fitted value iteration algorithm:

1. set $\mathbf{y}_i \leftarrow \max_{\mathbf{a}_i} (r(\mathbf{s}_i, \mathbf{a}_i) + \gamma E[V_\phi(\mathbf{s}'_i)])$
2. set $\phi \leftarrow \arg \min_\phi \frac{1}{2} \sum_i \|V_\phi(\mathbf{s}_i) - \mathbf{y}_i\|^2$

need to know outcomes
for different actions!

Back to policy iteration...

policy iteration:

1. evaluate $Q^\pi(\mathbf{s}, \mathbf{a})$
2. set $\pi \leftarrow \pi'$

$$\pi'(\mathbf{a}_t | \mathbf{s}_t) = \begin{cases} 1 & \text{if } \mathbf{a}_t = \arg \max_{\mathbf{a}_t} Q^\pi(\mathbf{s}_t, \mathbf{a}_t) \\ 0 & \text{otherwise} \end{cases}$$

policy evaluation:

$$V^\pi(\mathbf{s}) \leftarrow r(\mathbf{s}, \pi(\mathbf{s})) + \gamma E_{\mathbf{s}' \sim p(\mathbf{s}'|\mathbf{s}, \pi(\mathbf{s}))}[V^\pi(\mathbf{s}')]$$
$$Q^\pi(\mathbf{s}, \mathbf{a}) \leftarrow r(\mathbf{s}, \mathbf{a}) + \gamma E_{\mathbf{s}' \sim p(\mathbf{s}'|\mathbf{s}, \mathbf{a})}[Q^\pi(\mathbf{s}', \pi(\mathbf{s}'))]$$

can fit this using samples

Can we do the “max” trick again?

policy iteration:

- 
1. evaluate $V^\pi(\mathbf{s})$
 2. set $\pi \leftarrow \pi'$

fitted value iteration algorithm:

- 
1. set $\mathbf{y}_i \leftarrow \max_{\mathbf{a}_i} (r(\mathbf{s}_i, \mathbf{a}_i) + \gamma E[V_\phi(\mathbf{s}'_i)])$
 2. set $\phi \leftarrow \arg \min_\phi \frac{1}{2} \sum_i \|V_\phi(\mathbf{s}_i) - \mathbf{y}_i\|^2$

forget policy, compute value directly

can we do this with Q-values **also**, without knowing the transitions?

fitted Q iteration algorithm:

- 
1. set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma E[V_\phi(\mathbf{s}'_i)]$
 2. set $\phi \leftarrow \arg \min_\phi \frac{1}{2} \sum_i \|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|^2$
- approximate $E[V(\mathbf{s}'_i)] \approx \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
- doesn't require simulation of actions!

+ works even for off-policy samples (unlike actor-critic)

+ only one network, no high-variance policy gradient

- no convergence guarantees for non-linear function approximation (more on this later)

Fitted Q-iteration

full fitted Q-iteration algorithm:

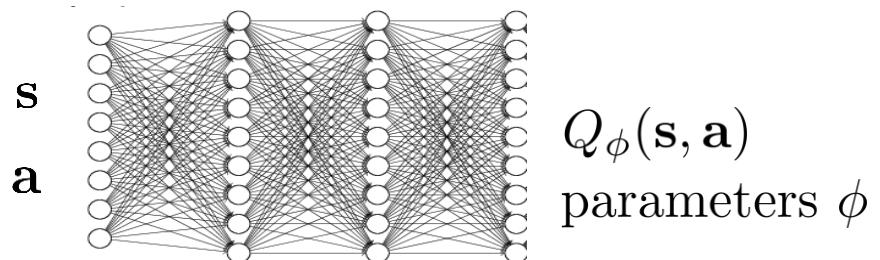
- 1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy
- 2. set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
- 3. set $\phi \leftarrow \arg \min_\phi \frac{1}{2} \sum_i \|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|^2$

parameters

dataset size N , collection policy

iterations K

gradient steps S



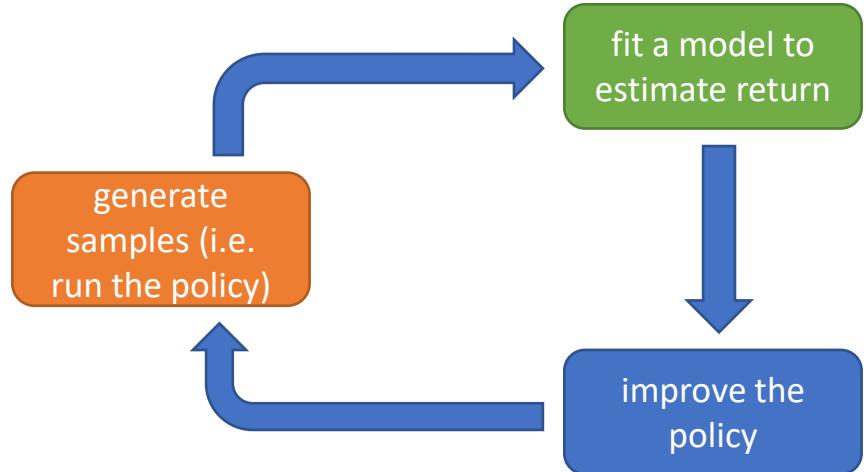
Q-Learning

Online Q-learning algorithms

full fitted Q-iteration algorithm:

- 
- 
1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy
 2. set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
 3. set $\phi \leftarrow \arg \min_\phi \frac{1}{2} \sum_i \|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|^2$

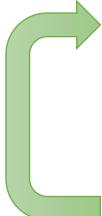
$$Q_\phi(\mathbf{s}, \mathbf{a}) \leftarrow r(\mathbf{s}, \mathbf{a}) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}', \mathbf{a}')$$



$$\mathbf{a} = \arg \max_{\mathbf{a}} Q_\phi(\mathbf{s}, \mathbf{a})$$

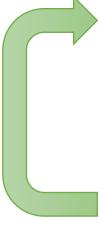
off policy, so many choices here!

online Q iteration algorithm:

- 
1. take some action \mathbf{a}_i and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
 2. $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
 3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$

Exploration with Q-learning

online Q iteration algorithm:

- 
1. take some action \mathbf{a}_i and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
 2. $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
 3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$

final policy:

$$\pi(\mathbf{a}_t | \mathbf{s}_t) = \begin{cases} 1 & \text{if } \mathbf{a}_t = \arg \max_{\mathbf{a}_t} Q_\phi(\mathbf{s}_t, \mathbf{a}_t) \\ 0 & \text{otherwise} \end{cases}$$

why is this a bad idea for step 1?

$$\pi(\mathbf{a}_t | \mathbf{s}_t) = \begin{cases} 1 - \epsilon & \text{if } \mathbf{a}_t = \arg \max_{\mathbf{a}_t} Q_\phi(\mathbf{s}_t, \mathbf{a}_t) \\ \epsilon / (|\mathcal{A}| - 1) & \text{otherwise} \end{cases}$$

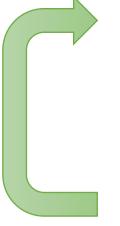
“epsilon-greedy”

$$\pi(\mathbf{a}_t | \mathbf{s}_t) \propto \exp(Q_\phi(\mathbf{s}_t, \mathbf{a}_t))$$

“Boltzmann exploration”

What's wrong?

online Q iteration algorithm:

- 
1. take some action \mathbf{a}_i and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
 2. $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
 3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$
- these are correlated!
- isn't this just gradient descent? that converges, right?

Q-learning is *not* gradient descent!

$$\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - (r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)))$$

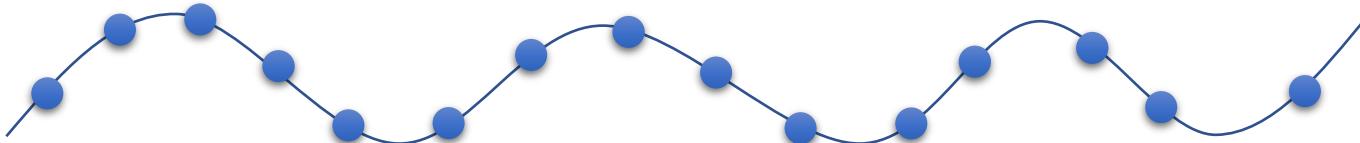
no gradient through target value

Correlated samples in online Q-learning

online Q iteration algorithm:

- 1. take some action \mathbf{a}_i and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
- 2. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$

- sequential states are strongly correlated
- target value is always changing



Replay buffers

online Q iteration algorithm:

- 1. take some action \mathbf{a}_i and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
- 2. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$

special case with $K = 1$, and one gradient step

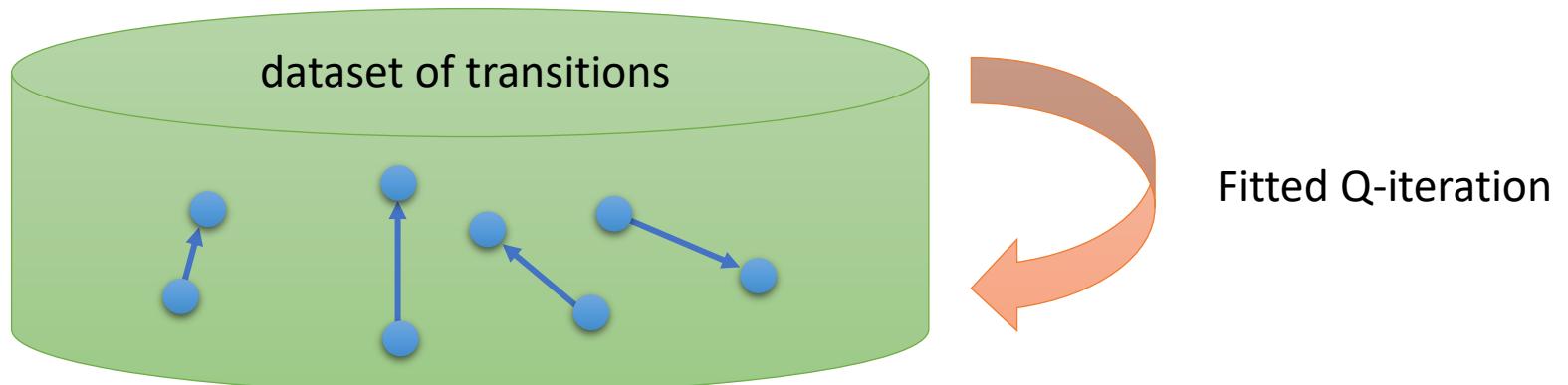
full fitted Q-iteration algorithm:

- 1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy
- 2. set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
- 3. set $\phi \leftarrow \arg \min_\phi \frac{1}{2} \sum_i \|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|^2$

any policy will work! (with broad support)

just load data from a buffer here

still use one gradient step



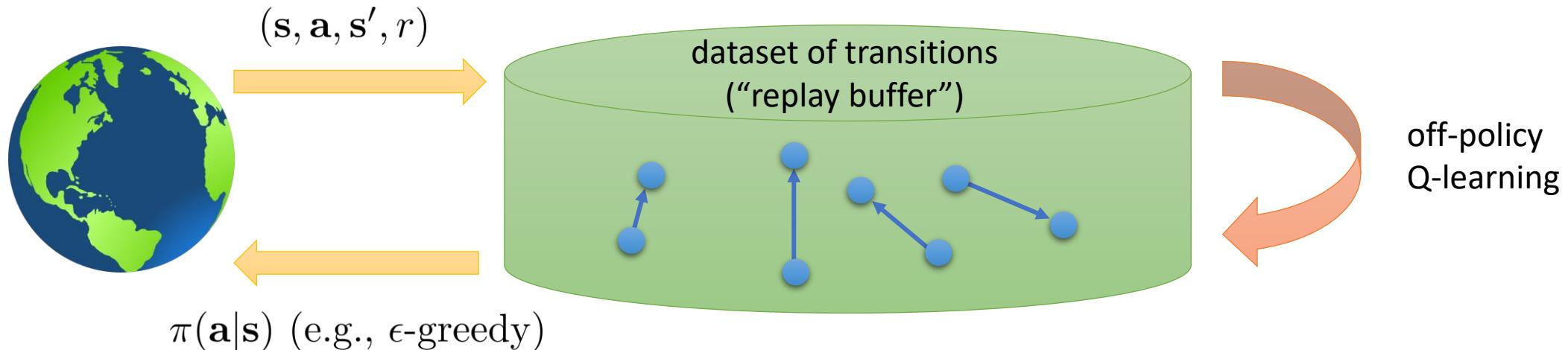
Replay buffers

Q-learning with a replay buffer:

- 1. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from \mathcal{B} + samples are no longer correlated
- 2. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$
+ multiple samples in the batch (low-variance gradient)

but where does the data come from?

need to periodically feed the replay buffer...

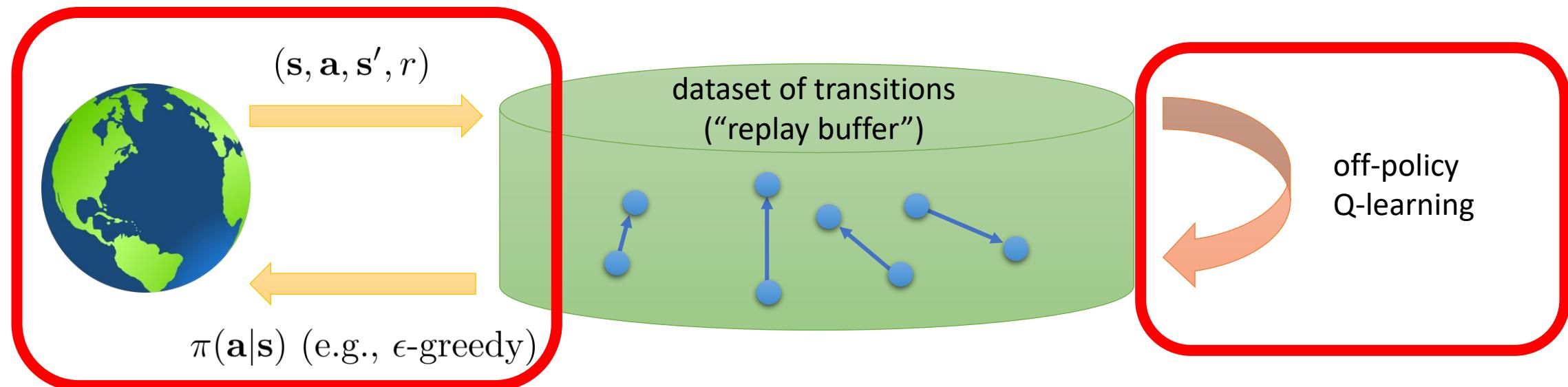


Putting it together

full Q-learning with replay buffer:

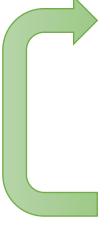
- 1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to \mathcal{B}
- 2. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from \mathcal{B}
- 3. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$

K = 1 is common, though larger K more efficient



What's wrong?

online Q iteration algorithm:

- 
1. take some action \mathbf{a}_i and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
 2. $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
 3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$
- use replay buffer**
- these are correlated!**

Q-learning is *not* gradient descent!

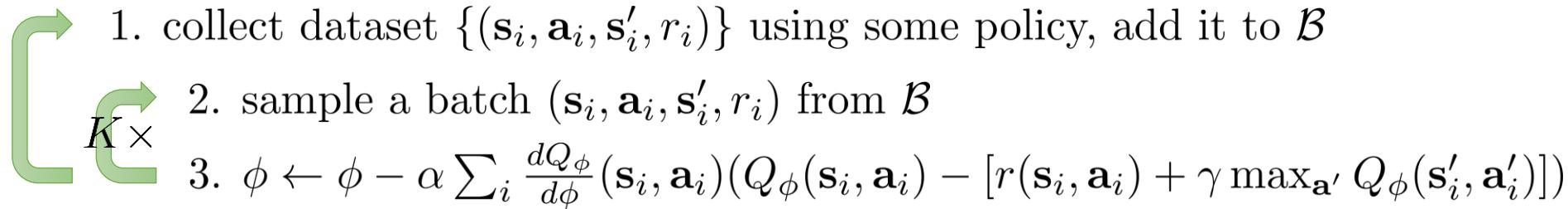
$$\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - (r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)))$$

no gradient through target value

This is still a problem!

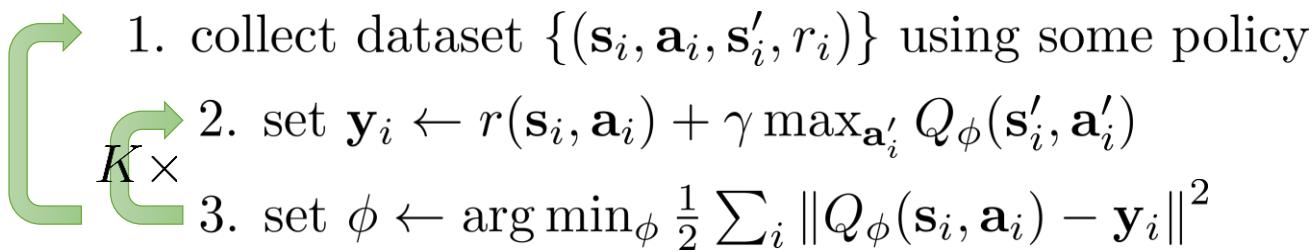
Q-Learning and Regression

full Q-learning with replay buffer:

- 
1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to \mathcal{B}
 2. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from \mathcal{B}
 3. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$

one gradient step, moving target

full fitted Q-iteration algorithm:

- 
1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy
 2. set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
 3. set $\phi \leftarrow \arg \min_\phi \frac{1}{2} \sum_i \|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|^2$

perfectly well-defined, stable regression

Q-Learning with target networks

supervised regression

Q-learning with replay buffer and target network:

-
1. save target network parameters: $\phi' \leftarrow \phi$
 2. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to \mathcal{B}
 3. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from \mathcal{B}
 4. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}'_i, \mathbf{a}'_i)])$
- targets don't change in inner loop!**

“Classic” deep Q-learning algorithm (DQN)

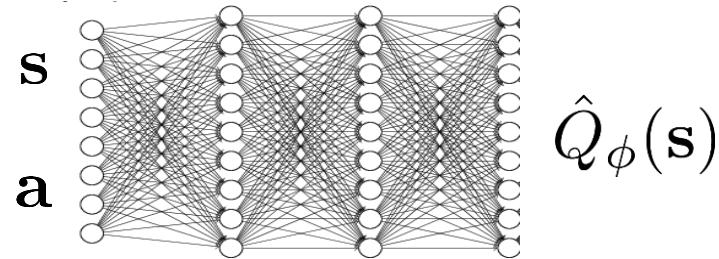
Q-learning with replay buffer and target network:

-
1. save target network parameters: $\phi' \leftarrow \phi$
2. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to \mathcal{B}
3. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from \mathcal{B}
4. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}'_i, \mathbf{a}'_i)])$

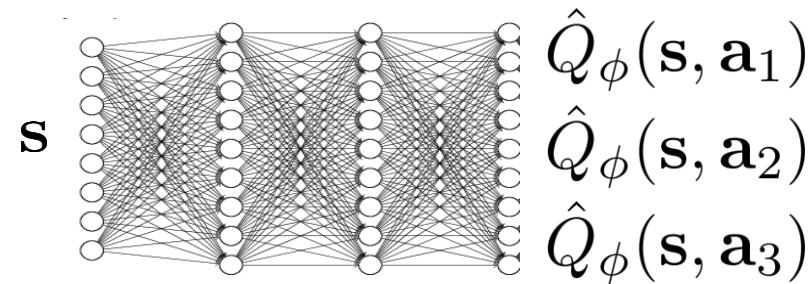
“classic” deep Q-learning algorithm:

-
1. take some action \mathbf{a}_i and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to \mathcal{B}
2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from \mathcal{B} uniformly
3. compute $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$ using *target* network $Q_{\phi'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
5. update ϕ' : copy ϕ every N steps

Representing the Q-function



more common with continuous actions

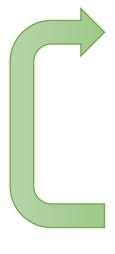


more common with discrete actions

Back to actor-critic

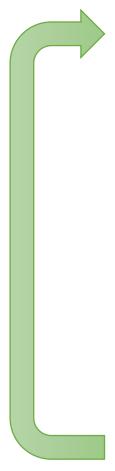
online Q iteration algorithm:

off policy, so many choices here!

- 
1. take some action \mathbf{a}_i and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
 2. $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
 3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$

with **continuous actions**, this is very inconvenient (but not impossible)

Idea: use actor-critic, but with Q-functions (to train off-policy)

- 
1. take some action \mathbf{a}_i and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to \mathcal{B}
 2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from \mathcal{B} uniformly
 3. compute $y_j = r_j + \gamma E_{\mathbf{a}'_j \sim \pi'_\theta(\mathbf{a}'_j | \mathbf{s}'_j)}[Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)]$ using *target* ϕ' and θ'
 4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
 5. $\theta \leftarrow \theta + \beta \sum_j \nabla_\theta E_{\mathbf{a} \sim \pi_\theta(\mathbf{a} | \mathbf{s}_j)}[Q_\phi(\mathbf{s}_j, \mathbf{a})]$ ← **policy gradient**
 6. update ϕ' and θ' every N steps

Simple practical tips for Q-learning

- Q-learning takes some care to stabilize
 - Test on easy, reliable tasks first, make sure your implementation is correct

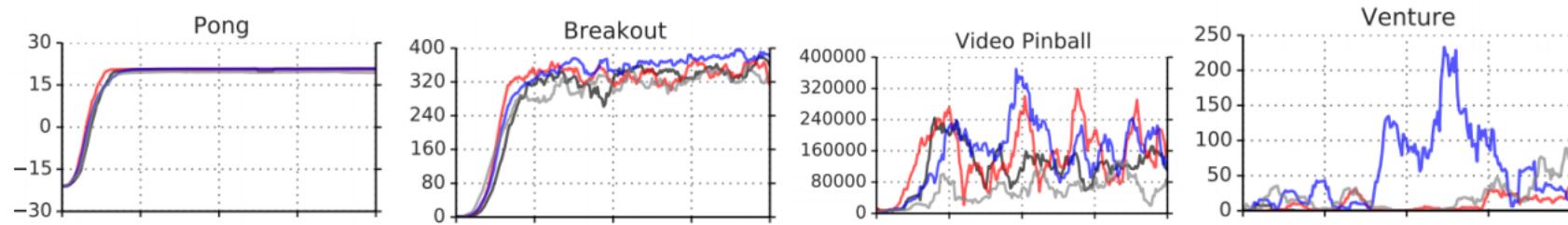


Figure: From T. Schaul, J. Quan, I. Antonoglou, and D. Silver. “Prioritized experience replay”. *arXiv preprint arXiv:1511.05952* (2015), Figure 7

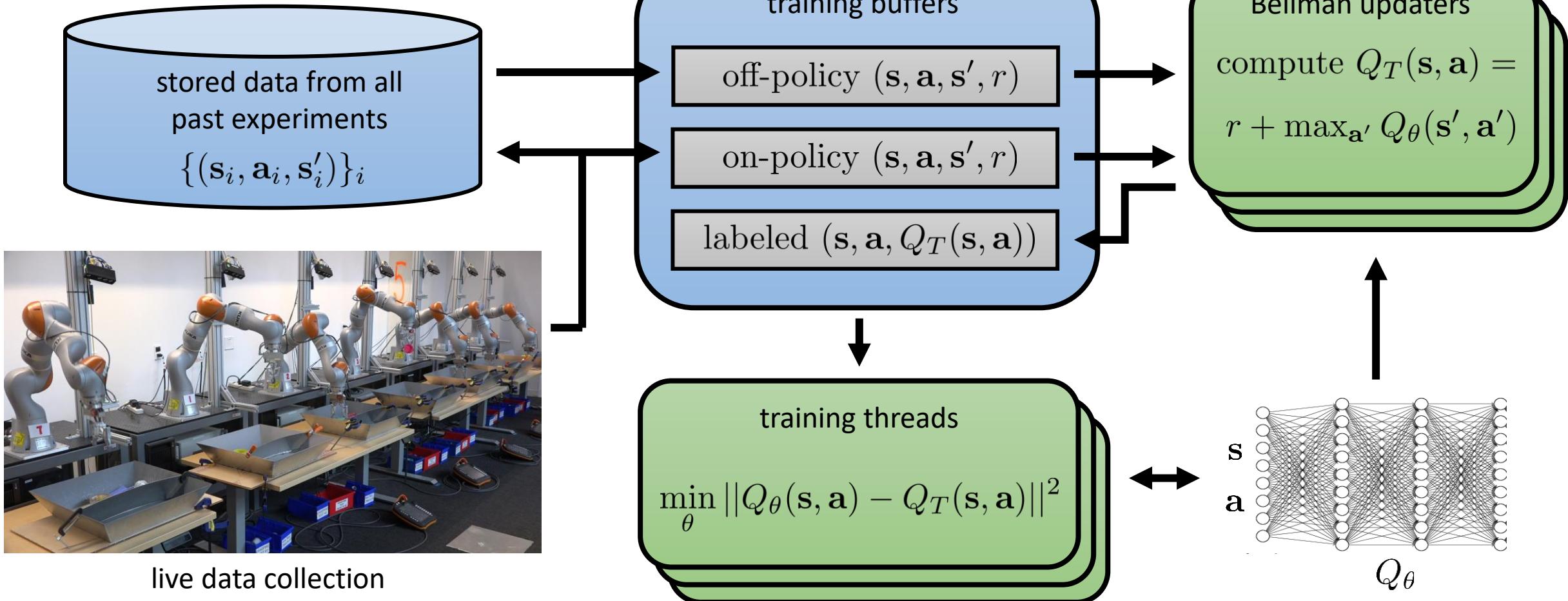
- Large replay buffers help improve stability
 - Looks more like fitted Q-iteration
- It takes time, be patient – might be no better than random for a while
- Start with high exploration (ϵ) and gradually reduce

Q-learning with convolutional networks

- “Human-level control through deep reinforcement learning,” Mnih et al. ‘13
- Q-learning with convolutional networks
- Uses replay buffer and target network
- One-step backup
- One gradient step
- Can be improved a lot with double Q-learning (and other tricks)



Large-scale Q-learning with continuous actions (QT-Opt)



Q-learning suggested readings

- Classic papers
 - Watkins. (1989). Learning from delayed rewards: introduces Q-learning
 - Riedmiller. (2005). Neural fitted Q-iteration: batch-mode Q-learning with neural networks
- Deep reinforcement learning Q-learning papers
 - Lange, Riedmiller. (2010). Deep auto-encoder neural networks in reinforcement learning: early image-based Q-learning method using autoencoders to construct embeddings
 - Mnih et al. (2013). Human-level control through deep reinforcement learning: Q-learning with convolutional networks for playing Atari.
 - Van Hasselt, Guez, Silver. (2015). Deep reinforcement learning with double Q-learning: a very effective trick to improve performance of deep Q-learning.
 - Lillicrap et al. (2016). Continuous control with deep reinforcement learning: continuous Q-learning with actor network for approximate maximization.
 - Gu, Lillicrap, Stuskever, L. (2016). Continuous deep Q-learning with model-based acceleration: continuous Q-learning with action-quadratic value functions.
 - Wang, Schaul, Hessel, van Hasselt, Lanctot, de Freitas (2016). Dueling network architectures for deep reinforcement learning: separates value and advantage estimation in Q-function.

Generative Modeling

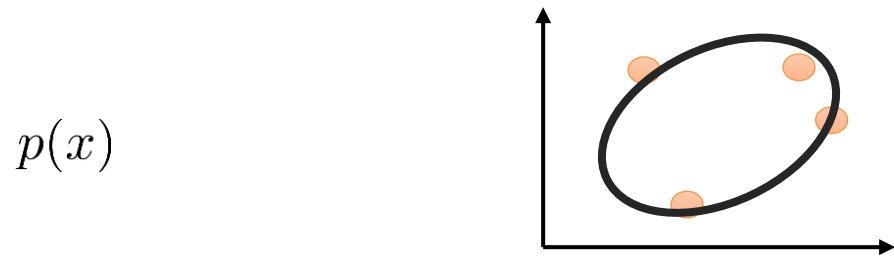
Designing, Visualizing and Understanding Deep Neural Networks

CS W182/282A

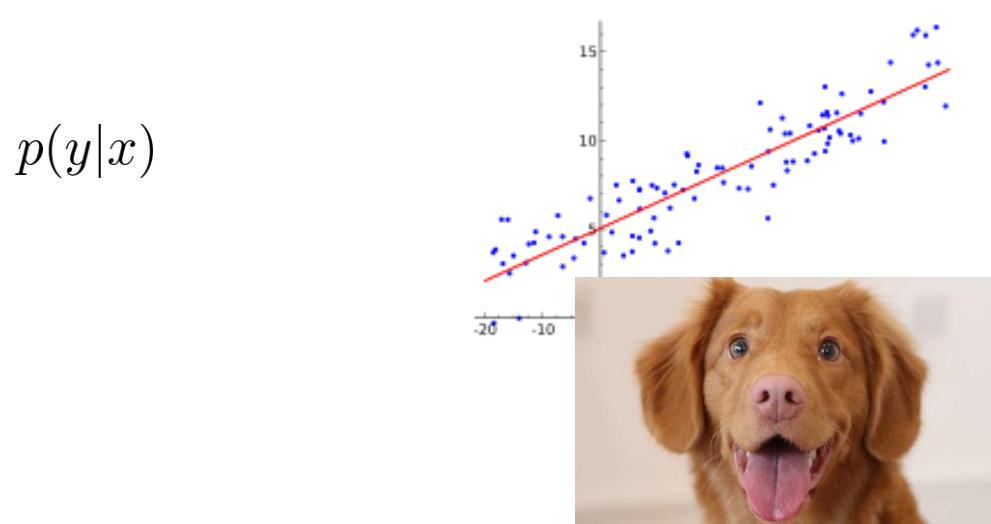
Instructor: Sergey Levine
UC Berkeley



Probabilistic models

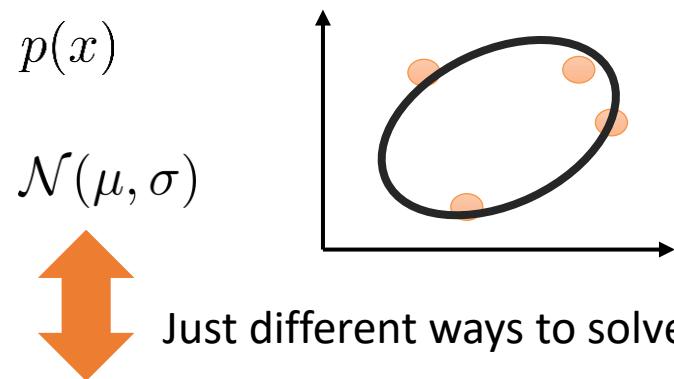


Why would we want to do this?



$$f_{\theta}(x) = y \rightarrow [\text{object label}]$$

Generative models

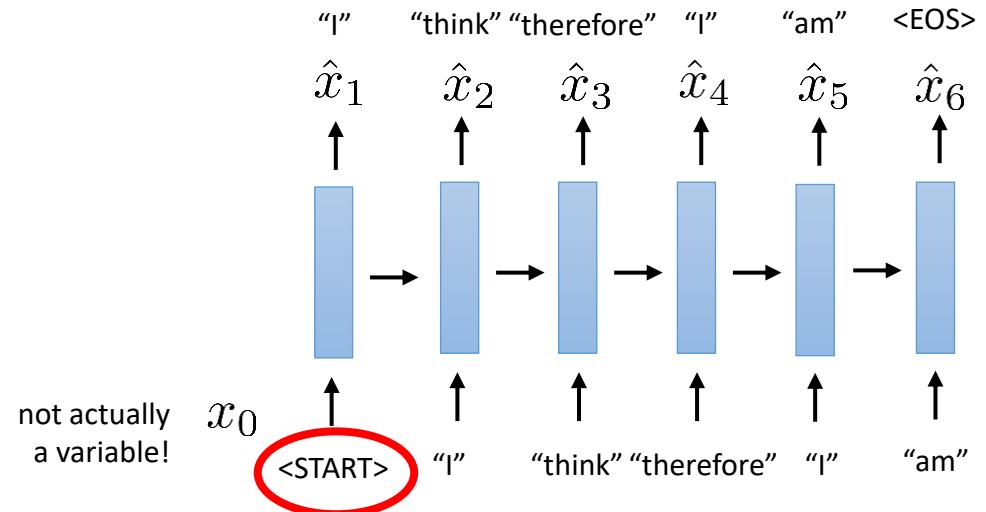


$$p(x) = p(x_1)p(x_2)p(x_3)p(x_4)p(x_5)p(x_6)p(x_7)p(x_8)p(x_9)p(x_{10})x_{0:5}$$

Today: can we go from “language models” to “everything models”?

This is called **unsupervised learning**

Just different ways to solve the same problem!

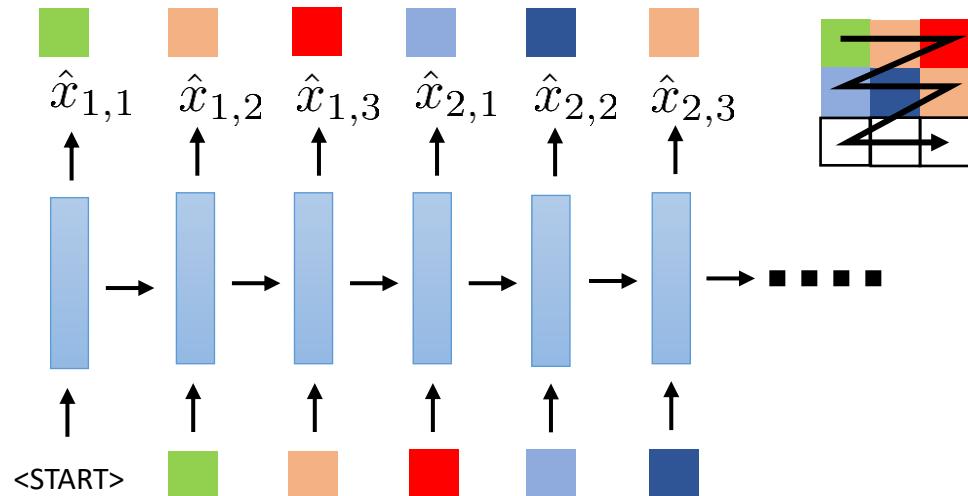
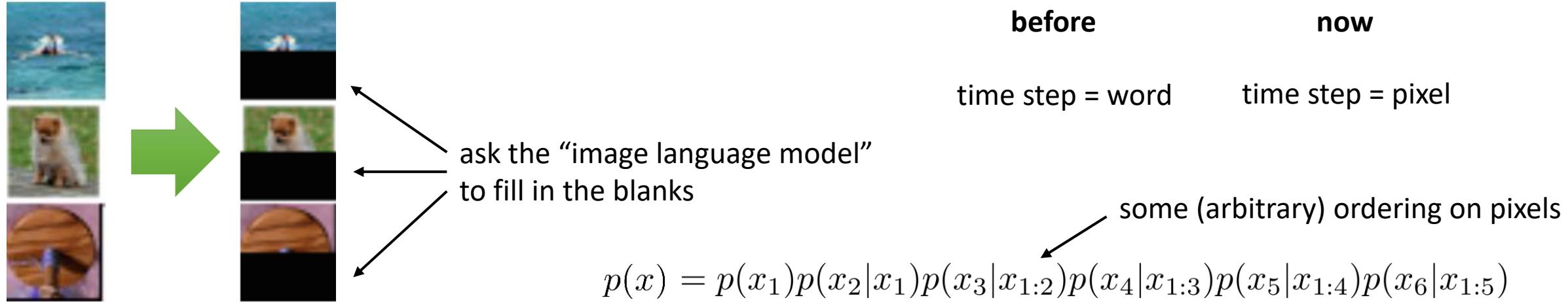


Why would we want to do this?

Same reasons as language modeling!

- Unsupervised pretraining on lots of data
 - Representation learning
 - Pretraining for later finetuning
 - Actually generating things!

Can we “language model” images?



This is basically the main idea, but there are some details we need to figure out!

- How to order the pixels?
- What kind of model to use?

Autoregressive generative models

Main principle for training:

1. Divide up x into dimensions x_1, \dots, x_n
2. Discretize each x_i into k values
3. Model $p(x)$ via the chain rule
$$p(x) = p(x_1)p(x_2|x_1)p(x_3|x_{1:2})p(x_4|x_{1:3})p(x_5|x_{1:4})p(x_6|x_{1:5})$$
4. Use your favorite sequence model to model $p(x)$

each of these is just a softmax

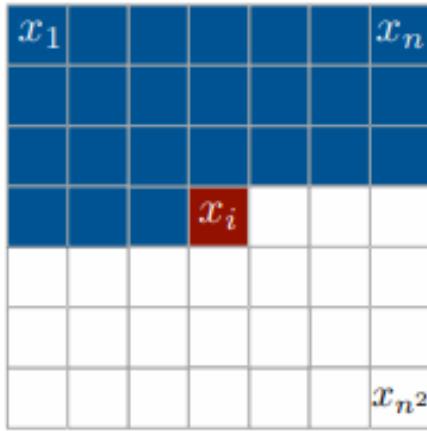
Using autoregressive generative models:

Sampling: ancestral sampling in sequence (x_1 , then x_2 , etc.)

Completion: feed in actual values for known x_i values

Representations: same idea as ELMo or BERT

PixelRNN



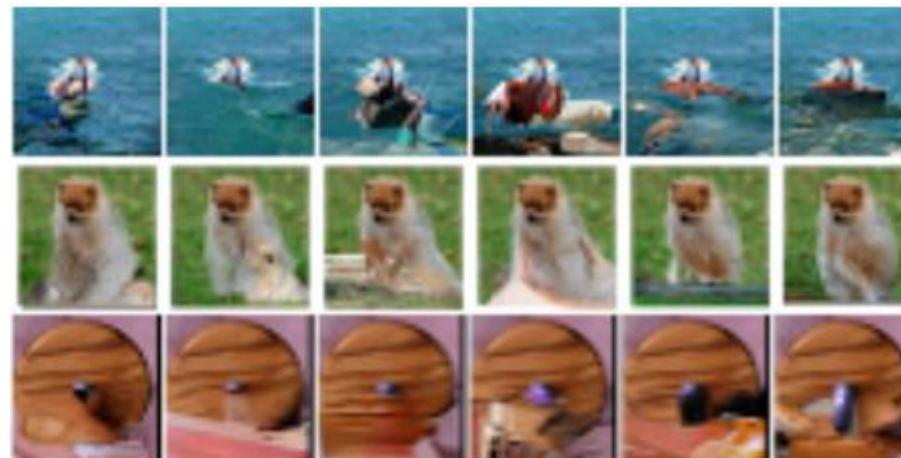
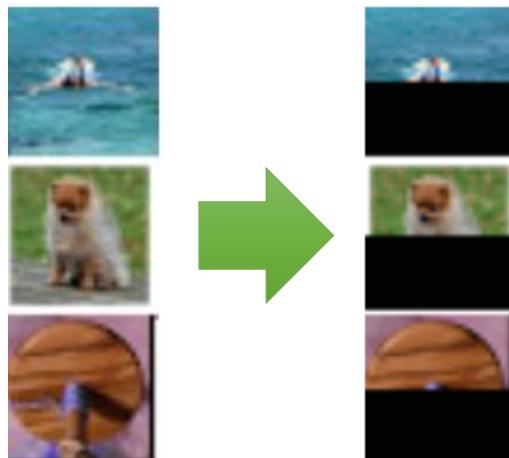
Pixels generated one at a time,
left-to-right, top-to-bottom:

$$p(\mathbf{x}) = \prod_{i=1}^{n^2} p(x_i | x_1, \dots, x_{i-1})$$

Generate one color channel at a time:

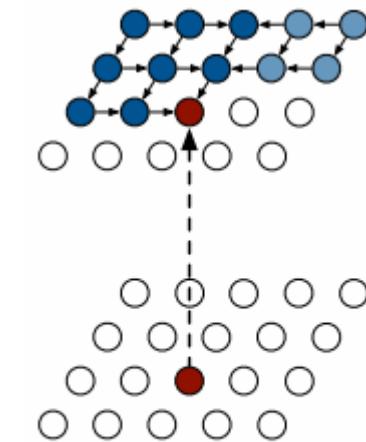
$$p(x_{i,R} | \mathbf{x}_{<i}) p(x_{i,G} | \mathbf{x}_{<i}, x_{i,R}) p(x_{i,B} | \mathbf{x}_{<i}, x_{i,R}, x_{i,G})$$

256-way softmax



Some practical considerations:

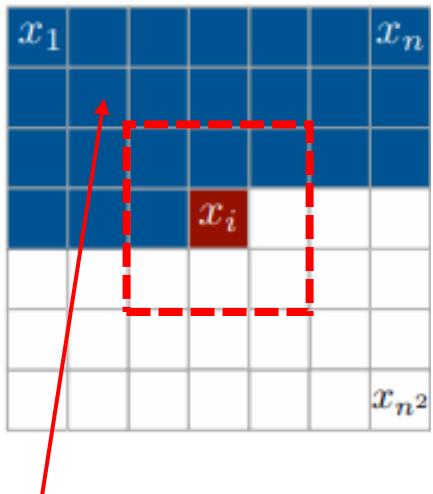
- It's very slow
- Row-by-row LSTMs might struggle to capture spatial context (pixels right above are "far away")
- Many practical improvements and better architectures are possible!



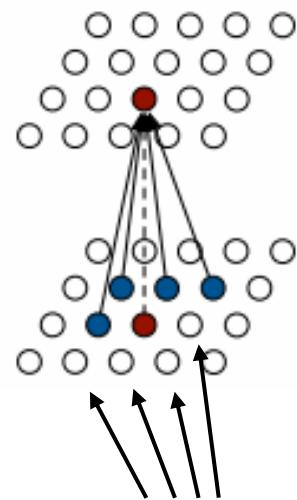
Diagonal BiLSTM

PixelCNN

Idea: make this much faster by not building a full RNN over all pixels, but just using a convolution to determine the value of a pixel based on its neighborhood



this pixel still influences x_i !
why?



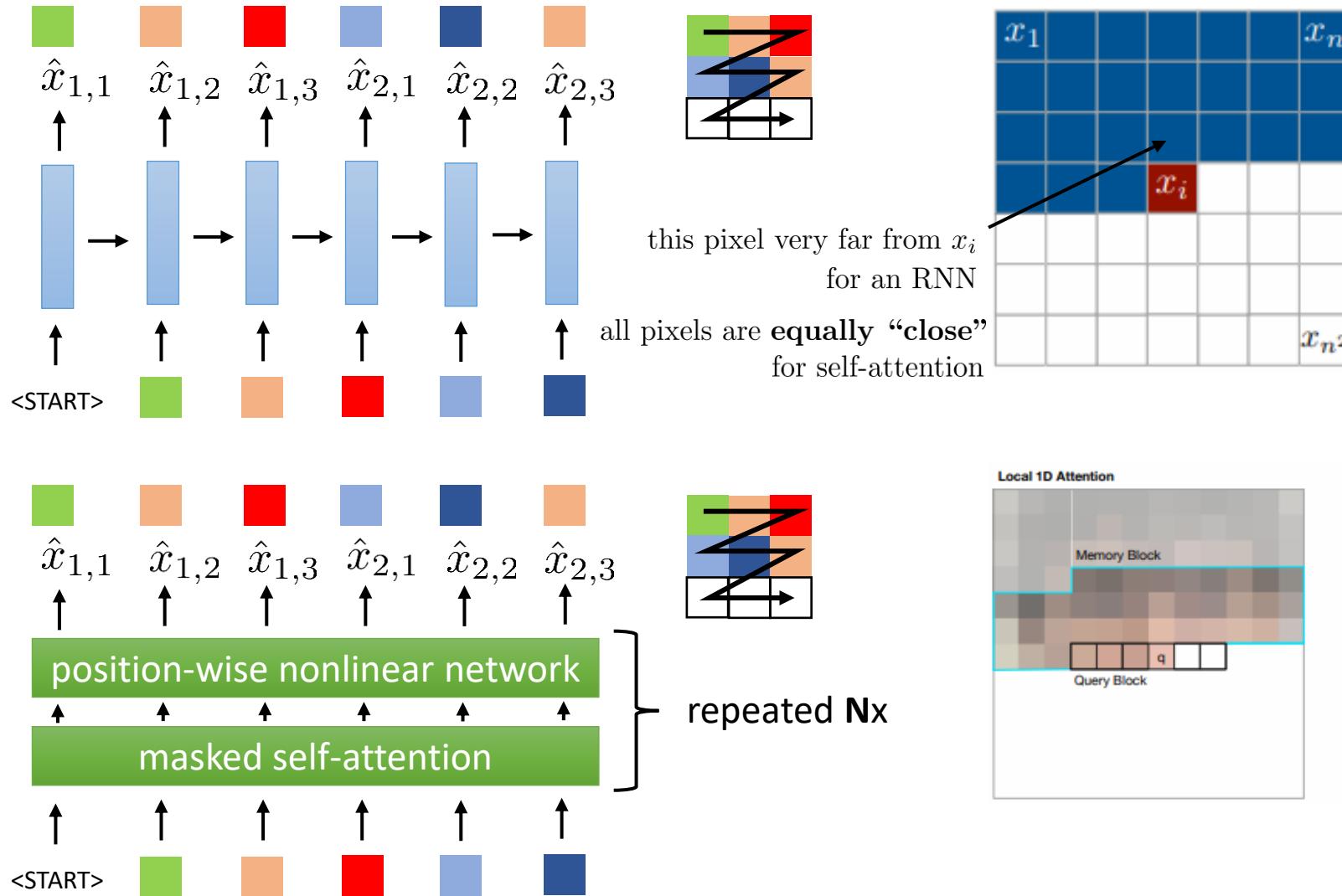
these are **masked out** because
they haven't been generated yet

Question: can we parallelize this?

During **training**?

During **generation**?

Pixel Transformer



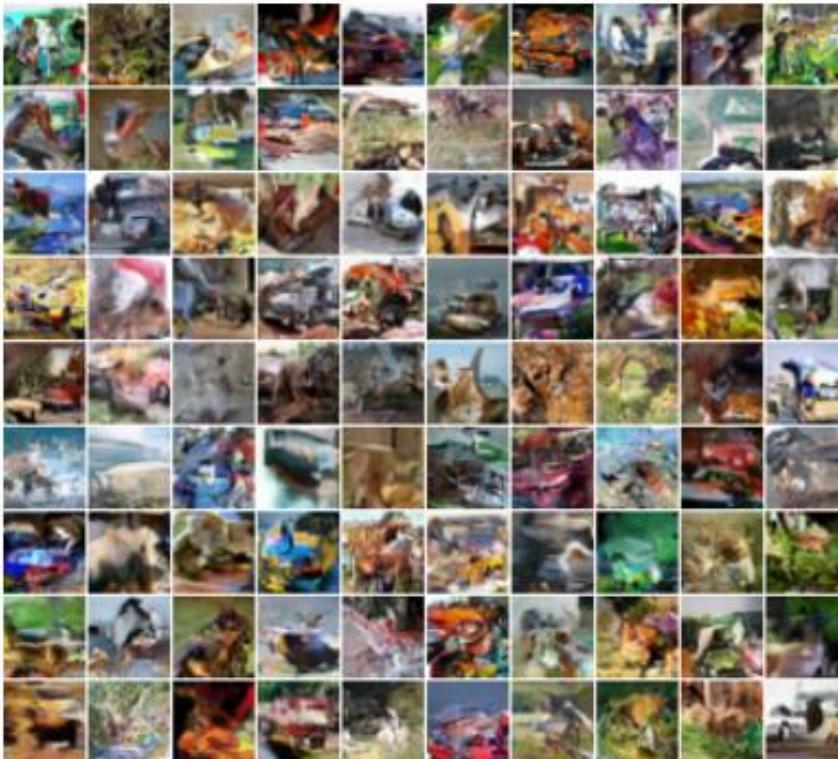
Problem: the number of pixels can be **huge**

attention can become prohibitively expensive

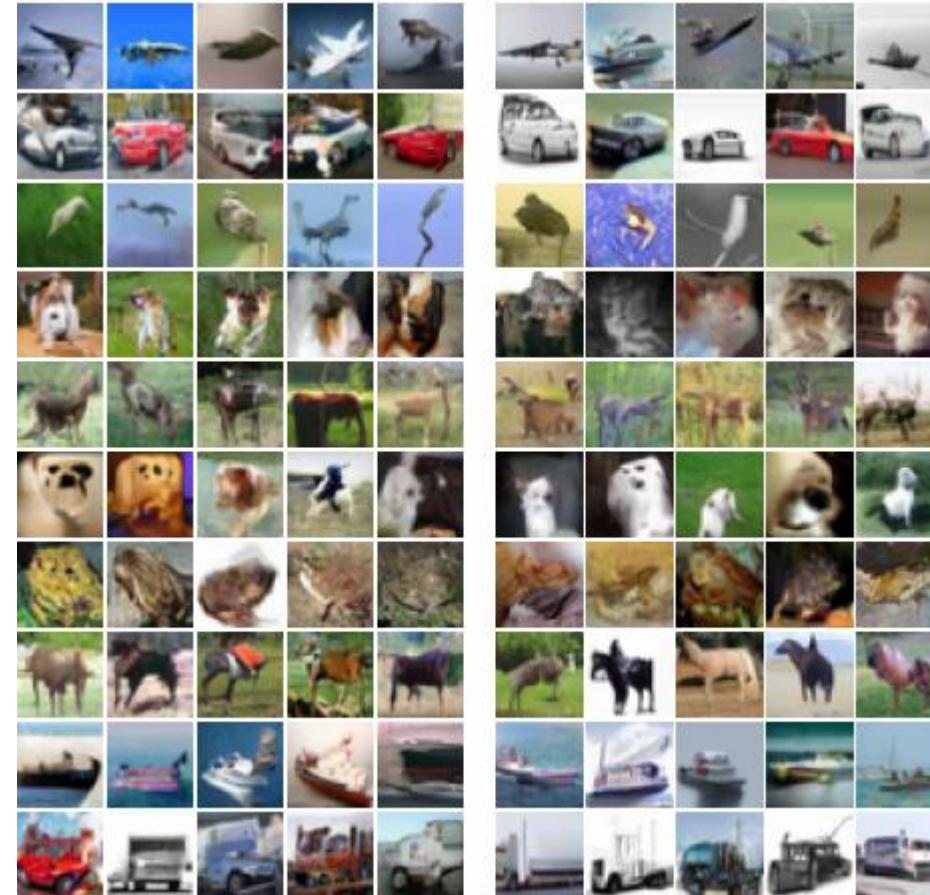
Idea: only compute attention for pixels that are not too far away
(looks a little like PixelCNN)

PixelRNN vs. Pixel Transformer

PixelRNN



Transformer



All models trained
on CIFAR-10

Conditional autoregressive models

What if we want to generate something **conditioned** on another piece of information?

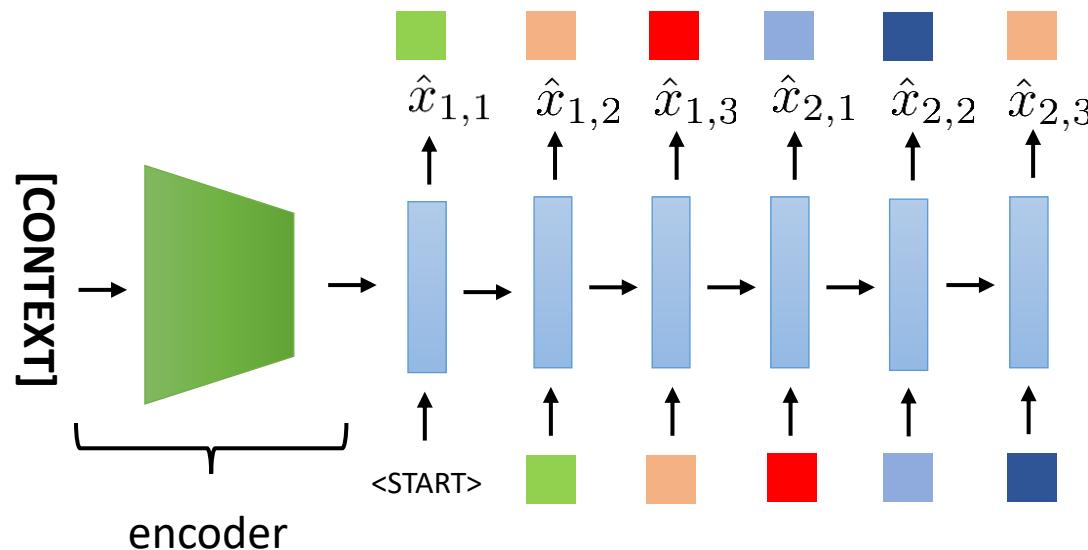
Examples:

- Generate images of specific types of objects (e.g., categories)
- Generate distributions over actions for imitation learning conditioned on the observation
- Many other examples!

Just like conditional language models!

Encoder can be **extremely simple**
(e.g., generate images of a class)

Encoder can be **extremely complex**
(e.g., multimodal policy in IL)



Conditional autoregressive models

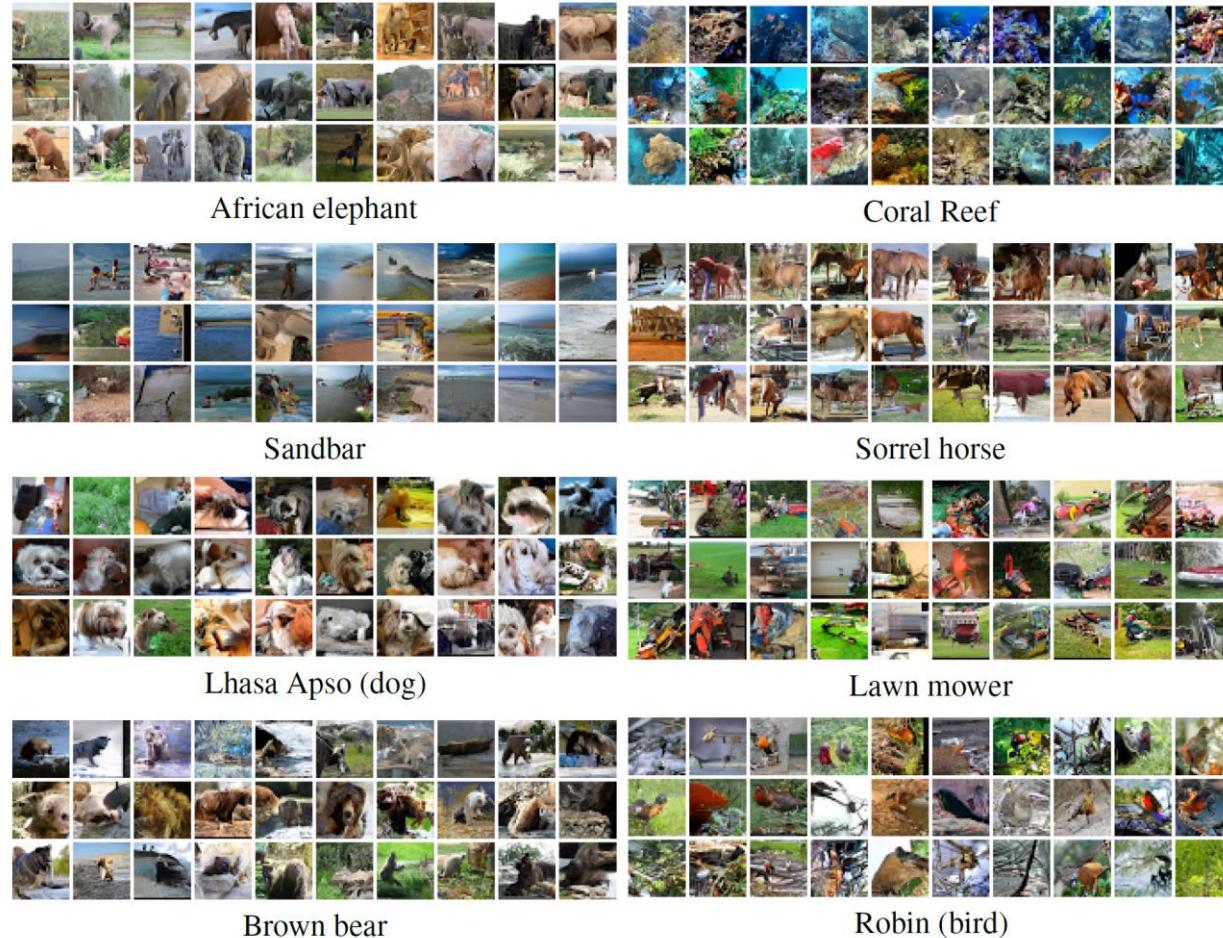


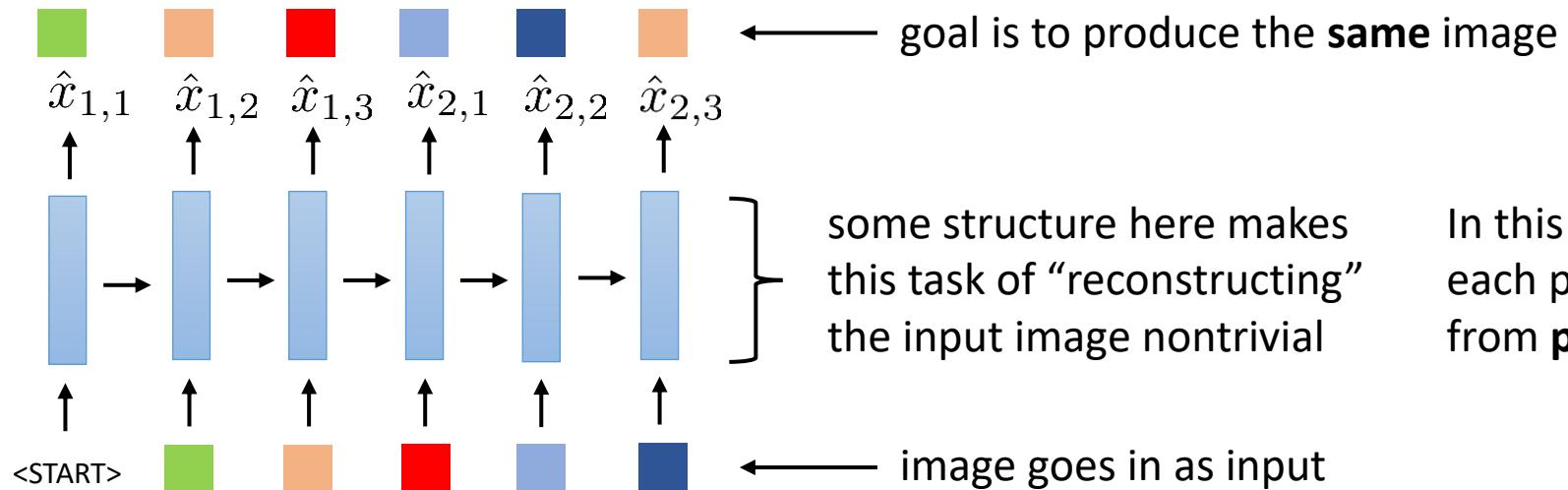
Figure 3: Class-Conditional samples from the Conditional PixelCNN.

Tradeoffs and considerations

- Autoregressive generative models are “language models” for other types of data
 - Though more accurate to say that language models are just a special type of autoregressive generative model
- Can represent autoregressive models in many different ways
 - RNNs (e.g., LSTMs)
 - Local context models like PixelCNNs
 - Transformers
- Tradeoffs compared to other models we'll learn about:
 - + provide full distribution with probabilities
 - + conceptually very simple
 - very slow for large datapoints (e.g., images)
 - generally limited in image resolution

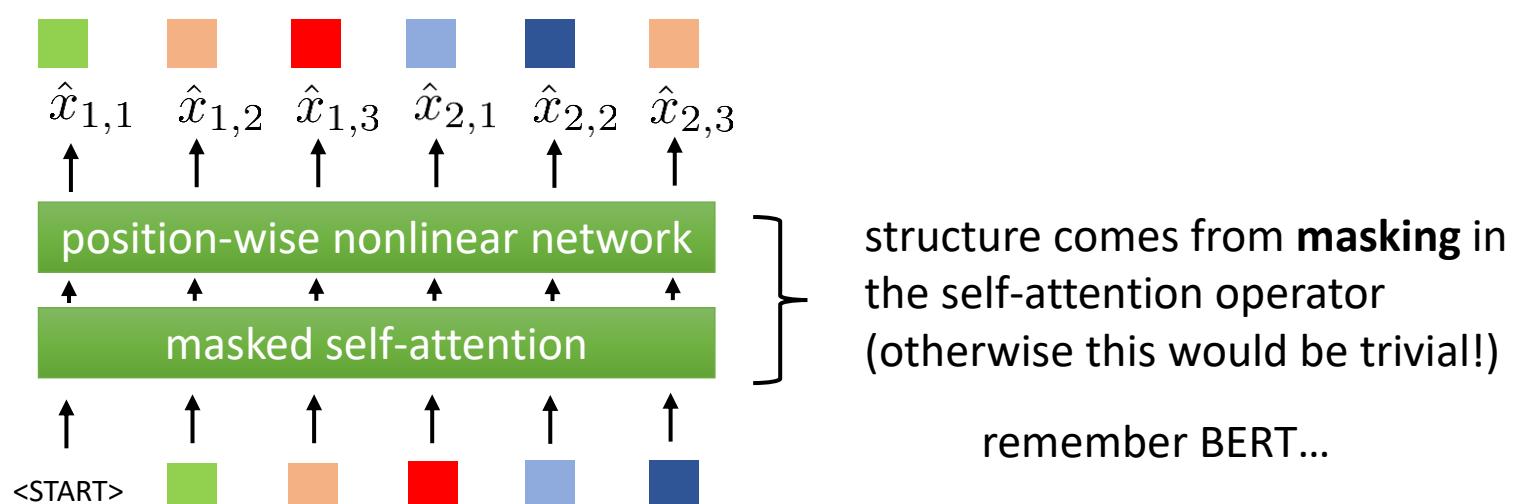
Autoencoders

A 30,000 ft view...



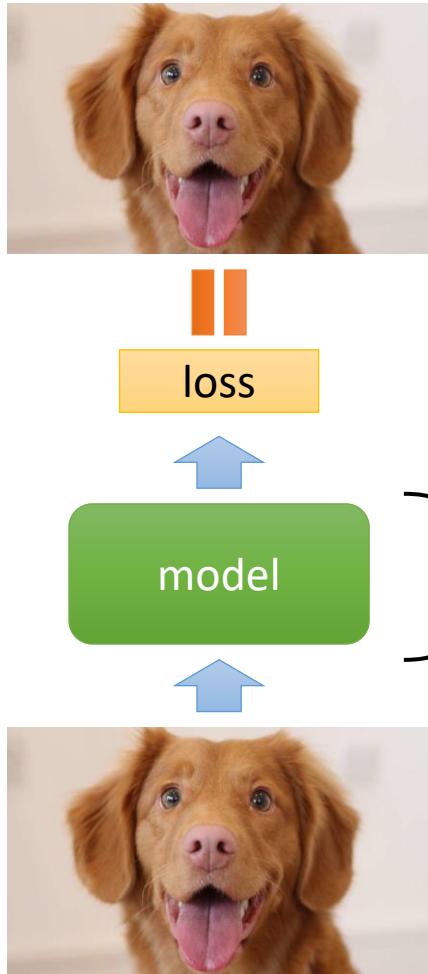
some structure here makes
this task of “reconstructing”
the input image nontrivial

In this case, it’s the fact that
each pixel must be constructed
from **preceding** pixels



A 30,000 ft view...

A general design for generative models?



Examples of structure that we've seen:

- RNN/LSTM sequence models that must predict a pixel's value based only on “previous” pixels
- “PixelCNN” models that must predict a pixel's value based on a (masked) neighborhood
- Pixel transformer, which must make predictions based on **masked** self-attention

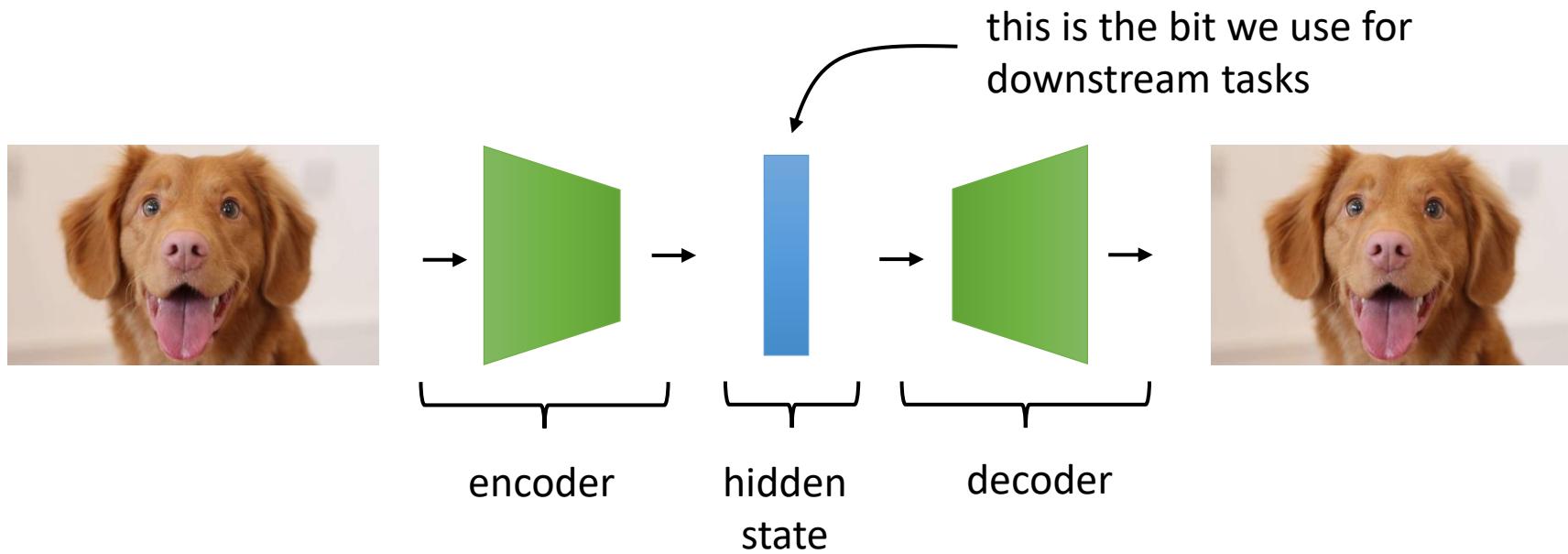
This is all **spatial** structure, can we use
more abstract structure instead?

The autoencoder principle

Basic idea: train a network that **encodes** an image into some **hidden state**, and then **decodes** that image **as accurately as possible** from that **hidden state**

Such a network is called an **autoencoder**

Forcing structure: something about the design of the model, or in the data processing or regularization, must force the autoencoder to learn a **structured representation**



The types of autoencoders

Forcing structure: something about the design of the model, or in the data processing or regularization, must force the autoencoder to learn a **structured** representation

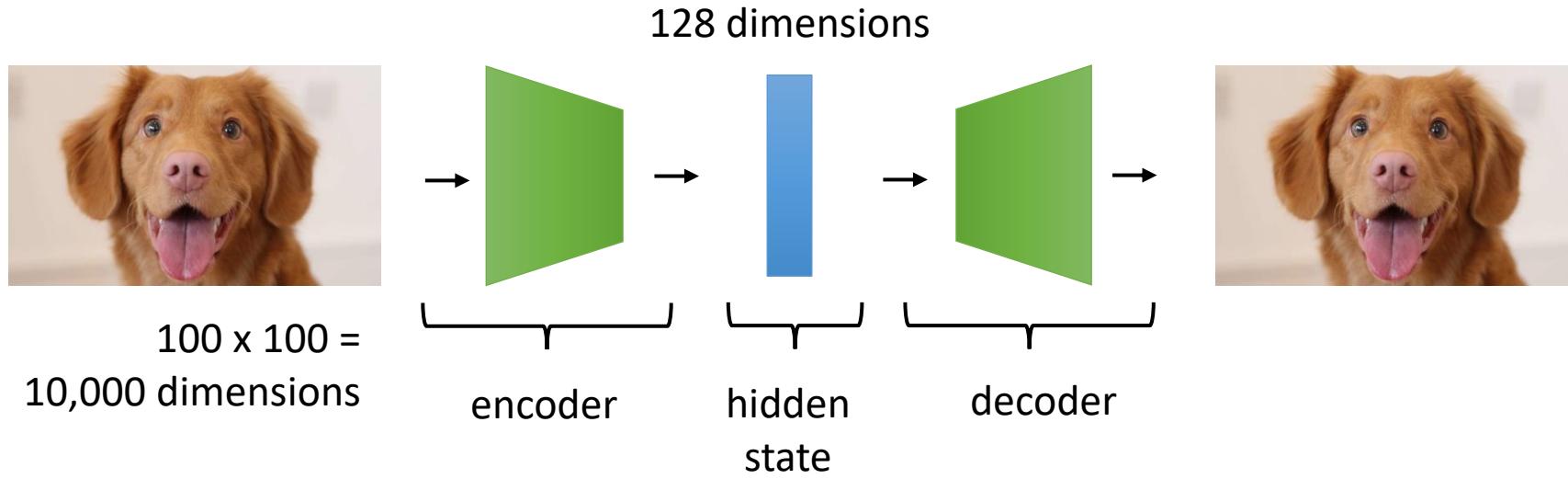
Dimensionality: make the **hidden state** smaller than the **input/output**, so that the network must **compress** it

Sparsity: force the **hidden state** to be sparse (most entries are zero), so that the network must **compress** the input

Denoising: corrupt the **input** with **noise**, forcing the autoencoder to learn to distinguish **noise from signal**

Probabilistic modeling: force the **hidden state** to agree with a **prior distribution** (this will be covered next time)

(Classic) Bottleneck autoencoder

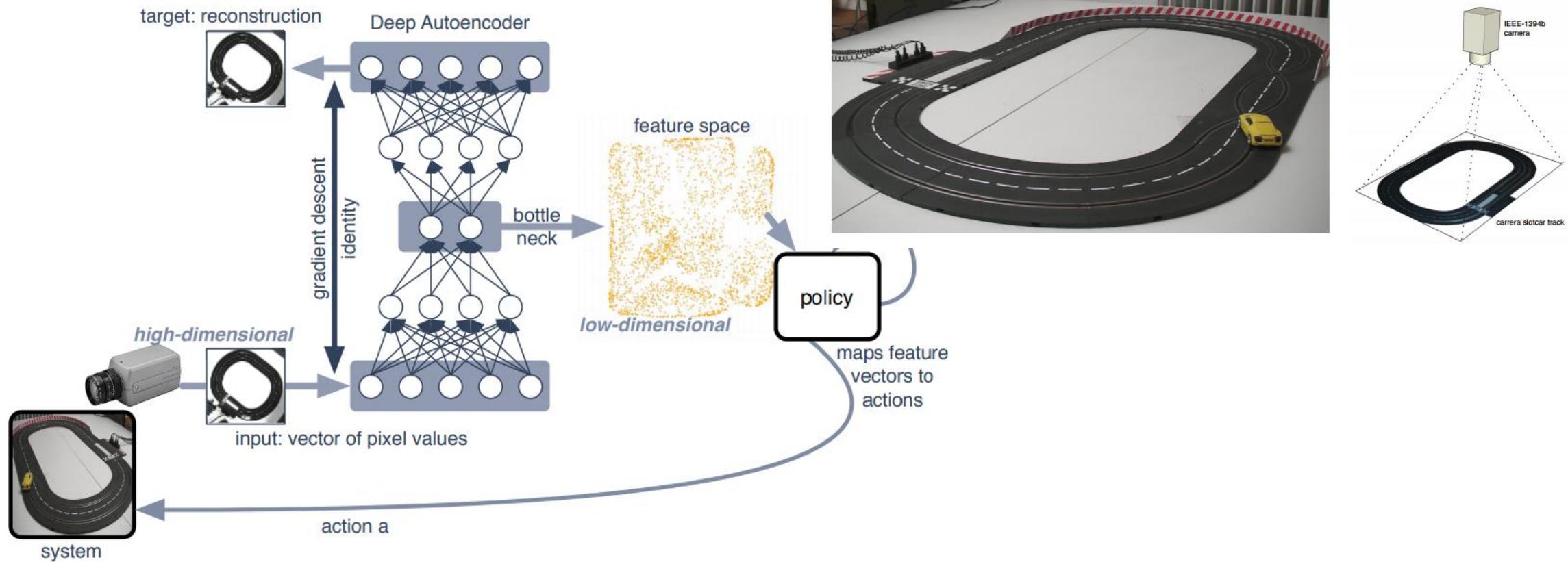


This has some interesting properties:

- If both encoder and decoder are **linear** (which is usually not very interesting), this exactly recovers PCA
- Can be viewed as “non-linear dimensionality reduction” – could be useful simply because dimensionality is lower and we can use various algorithms that are only tractable in low-dimensional spaces (e.g., discretization)

Today, this design is rather antiquated and rarely used,
but good to know about historically

Bottleneck autoencoder example



Sparse autoencoder

Idea: can we describe the input with a small set of “attributes”?

This might be a more **compressed** and **structured** representation



Pixel (0,0): #FE057D

Pixel (0,1): #FD0263

Pixel (0,2): #E1065F

⋮

Idea: “sparse” representations are going to be more structured!

NOT structured

“dense”: most values non-zero



has_ears: 1

has_wings: 0

has_wheels: 0

⋮

very structured!

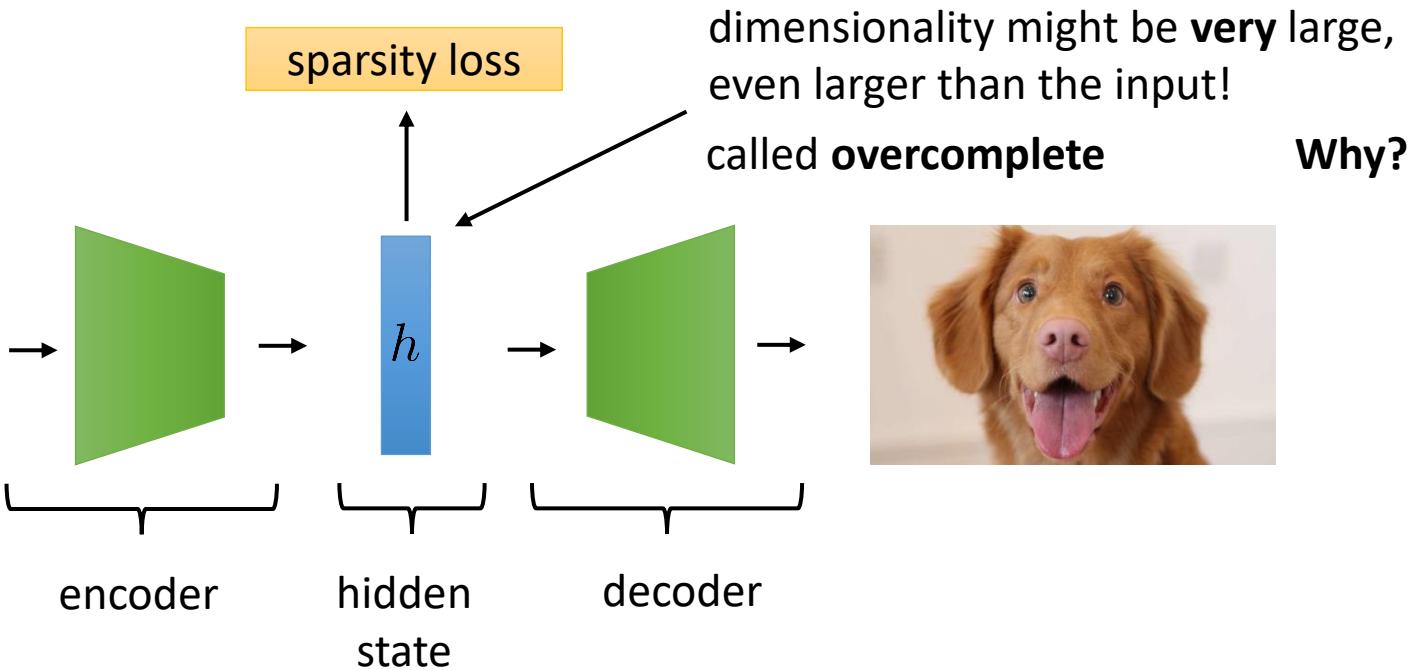
“sparse”: most values are zero

there are many possible “attributes,” and most images don’t have most of the attributes

Aside:

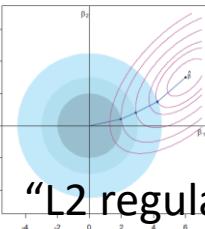
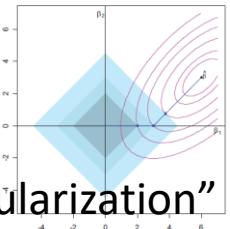
This idea originated in neuroscience, where researchers believe that the brain uses **sparse** representations (see “sparse coding”)

Sparse autoencoder



ℓ_1 norm of h
gradient is always $\text{sgn}(h_j)$

simple sparsity loss: $\sum_{j=1}^D |h_j|$



"L1 regularization"

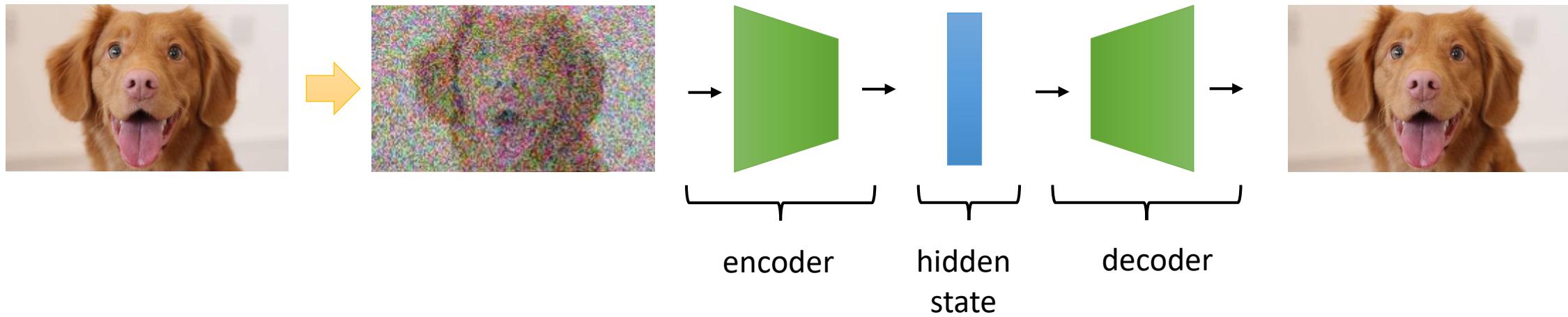
"L2 regularization"

There are other kinds of sparsity losses/models:

- Lifetime sparsity
- Spike and slab models

Denoising autoencoder

Idea: a good model that has learned meaningful structure should “fill in the blanks”



There are **many variants** on this basic idea, and this is one of
the most widely used simple autoencoder designs

The types of autoencoders

Forcing structure: something about the design of the model, or in the data processing or regularization, must force the autoencoder to learn a **structured** representation

Dimensionality: make the **hidden state** smaller than the **input/output**, so that the network must **compress** it

- + very simple to implement
- simply reducing dimensionality often does not provide the structure we want

Sparsity: force the **hidden state** to be sparse (most entries are zero), so that the network must **compress** the input

- + principled approach that can provide a “disentangled” representation
- harder in practice, requires choosing the regularizer and adjusting hyperparameters

Denoising: corrupt the **input** with **noise**, forcing the autoencoder to learn to distinguish **noise from signal**

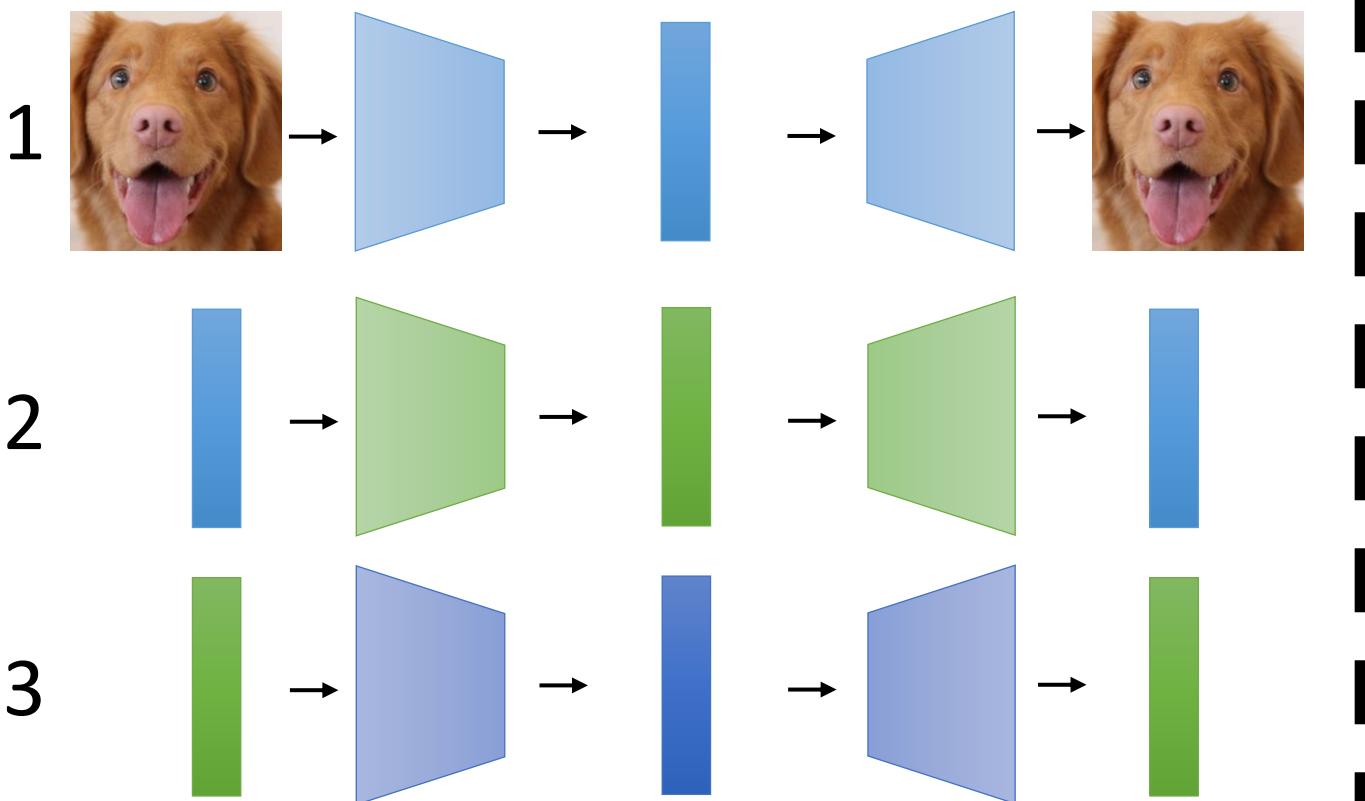
- + very simple to implement
- not clear which layer to choose for the bottleneck, many ad-hoc choices (e.g., how much noise to add)

Probabilistic modeling: force the **hidden state** to agree with a **prior distribution** (this will be covered next time)

We'll discuss this design in much more detail in the next lecture!

Layerwise pretraining

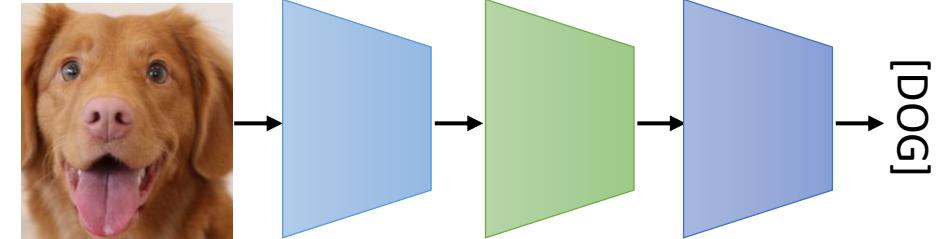
The early days of deep learning...



For a while (2006-2009 or so), this was one of the dominant ways to train **deep** networks

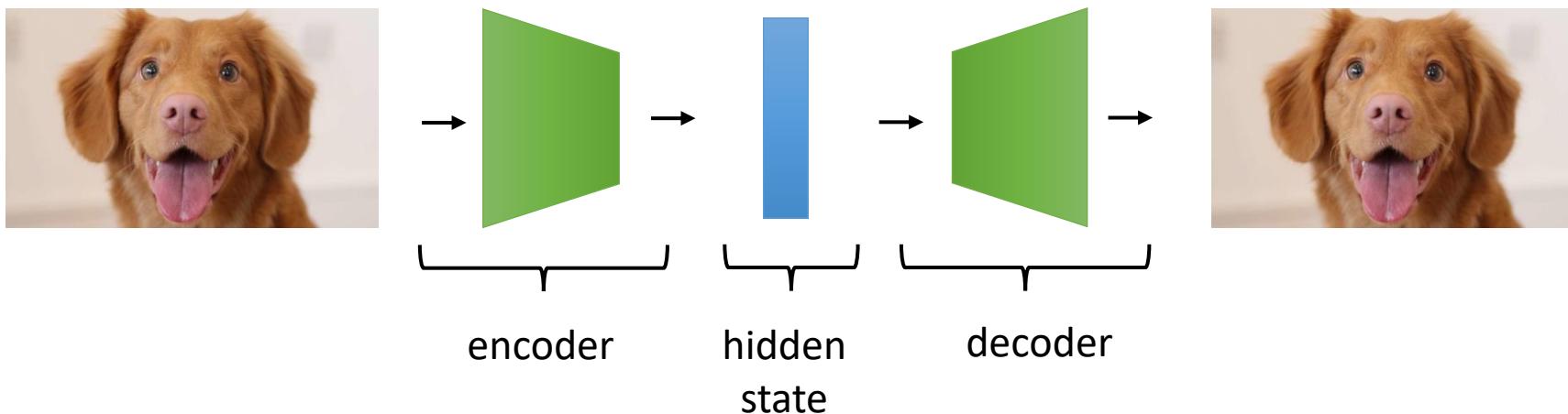
Then we got a lot better at training deep networks end-to-end (ReLU, batch norm, better hyperparameter tuning), and largely stopped doing this

Correspondingly, autoencoders became less important, but they are still useful!



Autoencoders today

- Much less widely used these days because there are better alternatives
 - **Representation learning:** VAEs, contrastive learning
 - **Generation:** GANs, VAEs, autoregressive models
- Still a viable option for “quick and dirty” representation learning that is very fast and can work OK
- **Big problem:** sampling (generation) from an autoencoder is hard, which limits its uses
 - The variational autoencoder (VAE) addresses this, and is the most widely used autoencoder today – we will cover this next time!

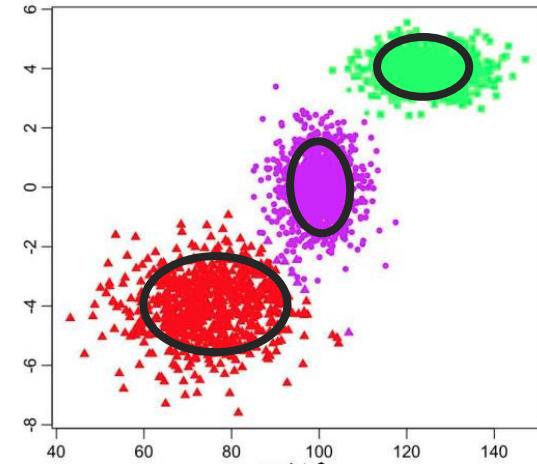


Latent Variable Models

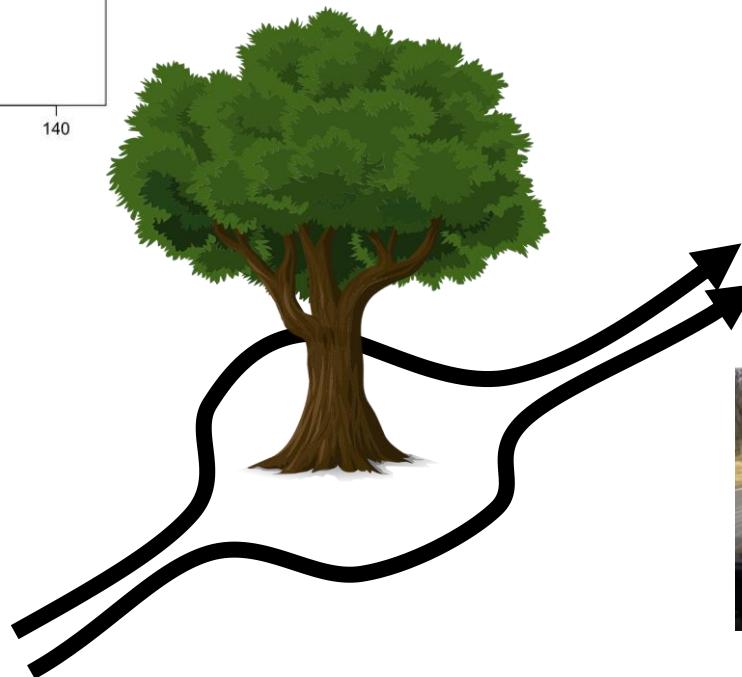
Latent variable models

$$p(x) = \sum_z p(x|z)p(z)$$

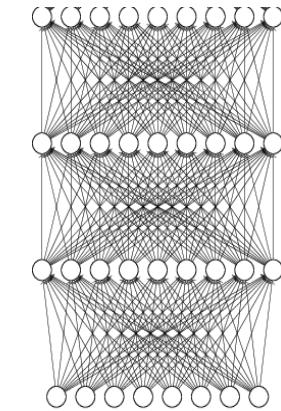
↑
mixture
element



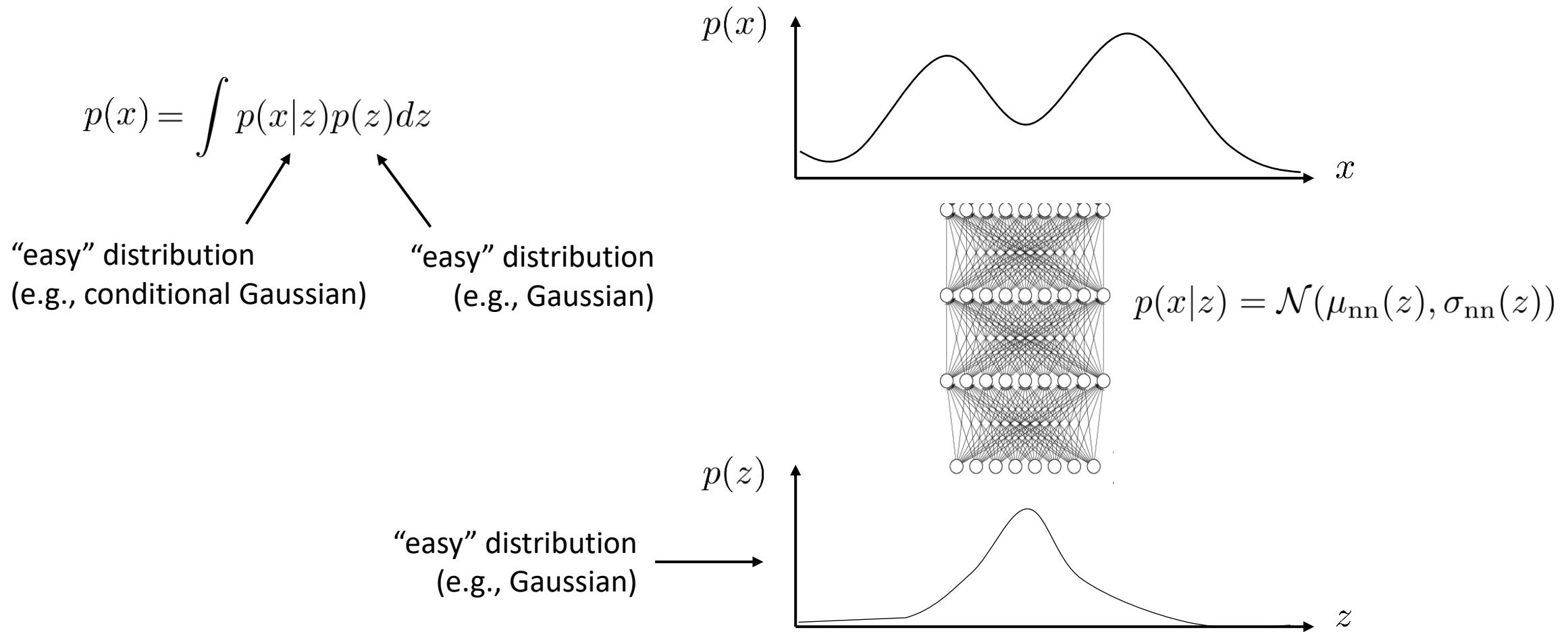
$$p(y|x) = \sum_z p(y|x, z)p(z)$$



$$w_1, \mu_1, \Sigma_1, \dots, w_N, \mu_N, \sigma_N$$



Latent variable models in general



How do we train latent variable models?

the model: $p_\theta(x)$

the data: $\mathcal{D} = \{x_1, x_2, x_3, \dots, x_N\}$

maximum likelihood fit:

$$\theta \leftarrow \arg \max_{\theta} \frac{1}{N} \sum_i \log p_\theta(x_i) \quad p(x) = \int p(x|z)p(z)dz$$

$$\theta \leftarrow \arg \max_{\theta} \frac{1}{N} \sum_i \log \left(\int p_\theta(x_i|z)p(z)dz \right)$$



completely intractable

Estimating the log-likelihood

alternative: *expected* log-likelihood:

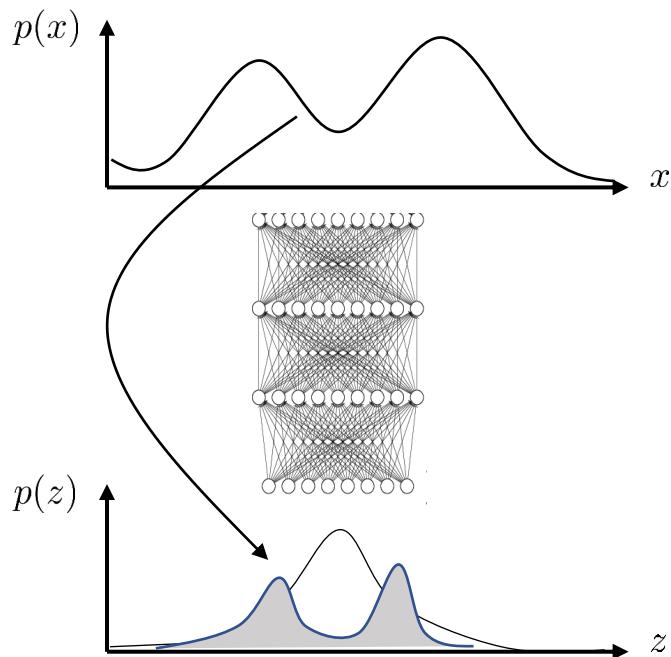
$$\theta \leftarrow \arg \max_{\theta} \frac{1}{N} \sum_i E_{z \sim p(z|x_i)} [\log p_{\theta}(x_i, z)]$$

but... how do we calculate $p(z|x_i)$?

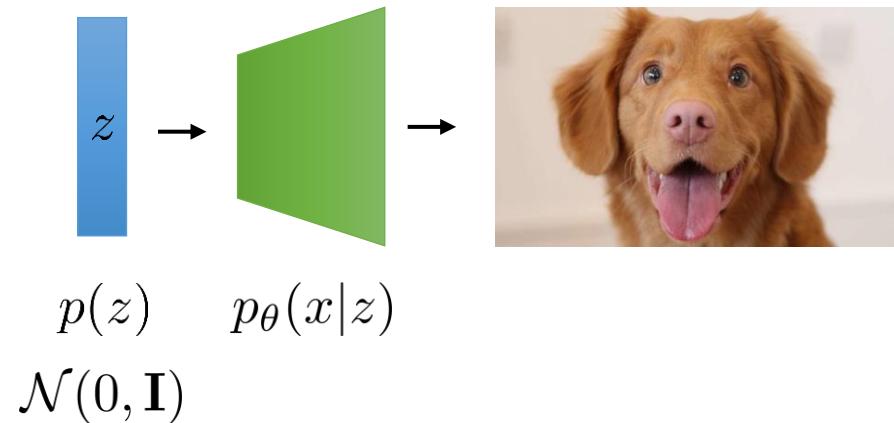
this is called **probabilistic inference**

intuition: “guess” most likely z given x_i ,
and pretend it’s the right one

...but there are many possible values of z
so use the distribution $p(z|x_i)$



Latent variable models in deep learning



A latent variable deep generative model is (usually) just a model that turns random numbers into valid samples (e.g., images)

Please don't tell anyone I said this, it destroys the mystique

There are many types of such models: VAEs, GANs, normalizing flows, etc.

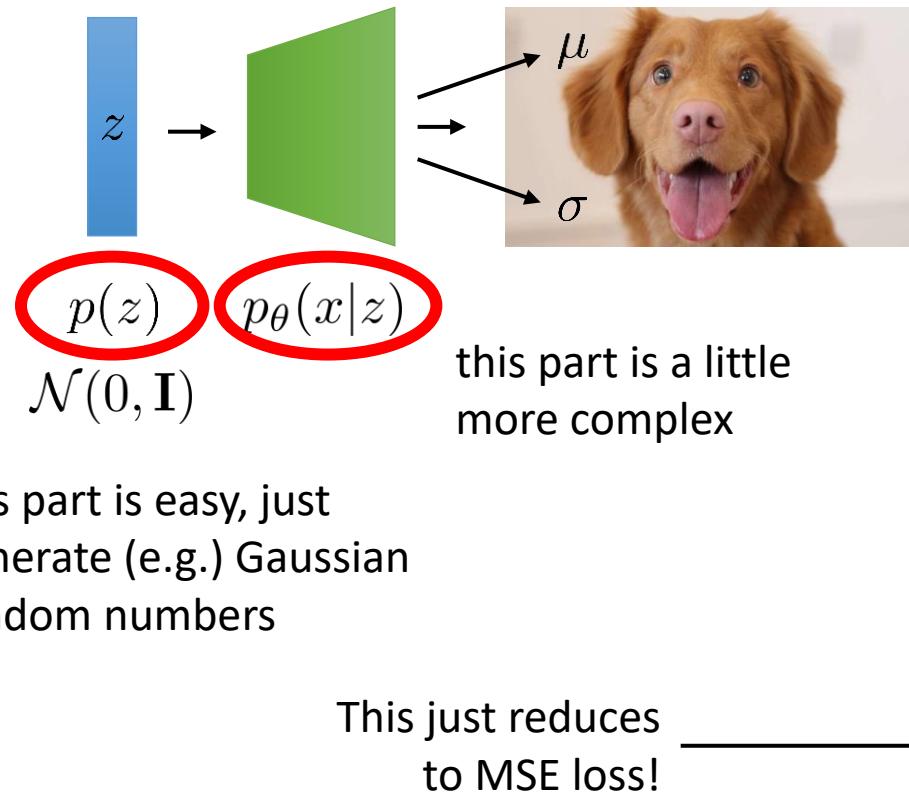
Using the model for **generation**:

1. sample $z \sim p(z)$ “generate a vector of random numbers”
2. sample $x \sim p(x|z)$ “turn that vector of random numbers into an image”

Today: how do we represent and use this

Next time: how do we train this

Representing latent variable models



How do we represent the distribution over x ?

Option 1: Pixels are continuous-valued

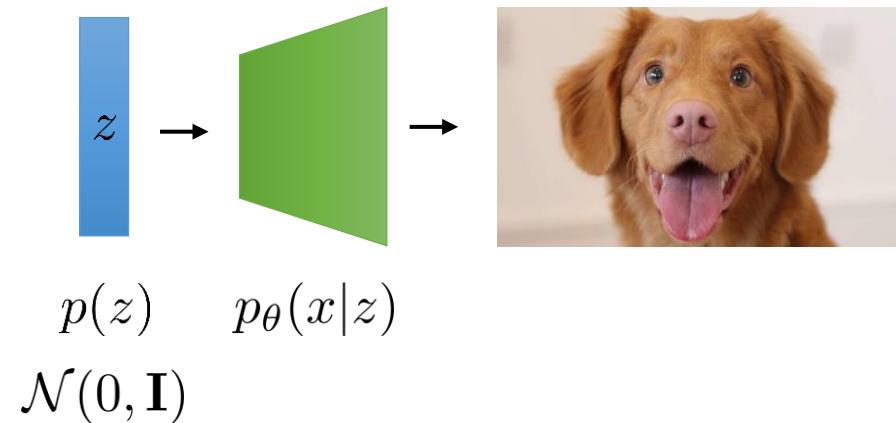
$$p_\theta(x|z) = \mathcal{N}(\mu_\theta(z); \sigma_\theta(z))$$

mean is a neural net function

variance is (optionally) a neural net function

easy choice: let σ just be a constant either a learned constant (independent of z) or chosen manually (e.g., 1)

Representing latent variable models



How do we represent the distribution over x ?

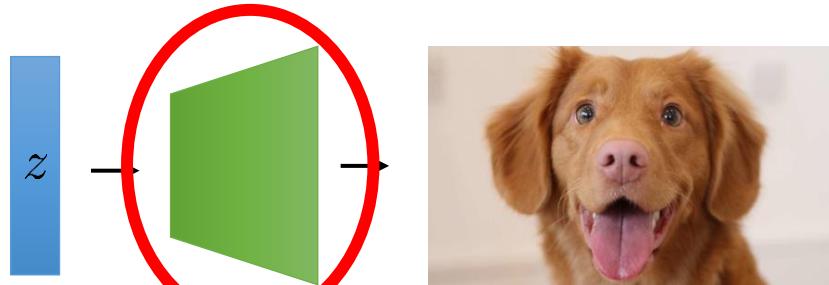
Option 2: Pixels are discrete-valued

Could just a 256-way softmax, just like in PixelRNN or PixelCNN!
(this works very well, but is a little bit slow)

Other choices (not covered in this lecture): discretized logistic, binary cross-entropy

especially common for best performing models

Representing latent variable models

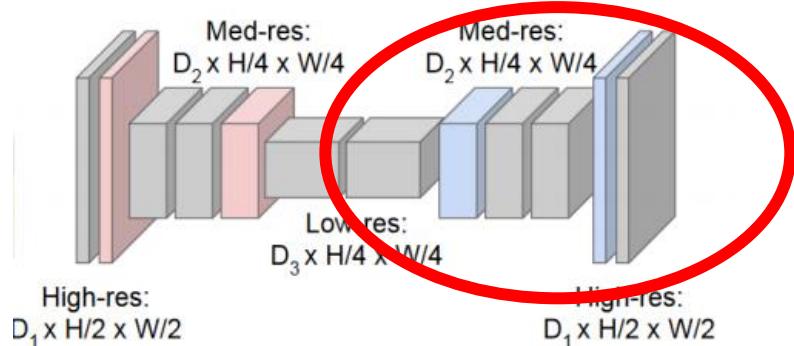


$p(z)$ $p_{\theta}(x|z)$ what architecture should we use?

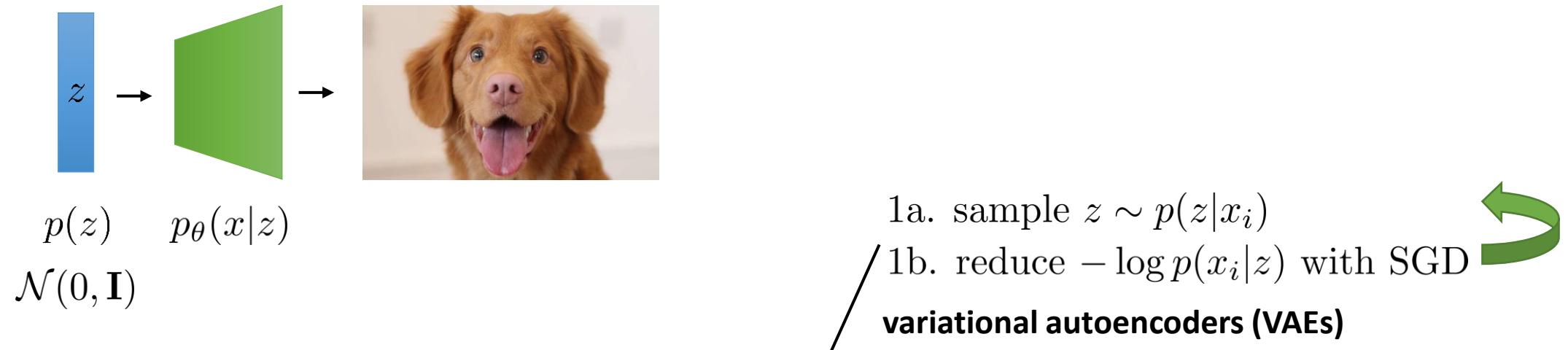
$\mathcal{N}(0, \mathbf{I})$

Easy choice: just a big fully connected network (linear layers + ReLU)
works well for tiny images (e.g., MNIST) or non-image data

Better choice: transpose convolutions



Training latent variable models



Three basic choices:

1. Perform inference to figure out $p(z|x_i)$ for each training image x_i

Then minimize *expected* NLL $E_{p(z|x_i)}[-\log p(x_i|z)]$

2. Use an *invertible* mapping z to x **normalizing flows**

3. Match the distribution $E_{z \sim p(z)}[p_\theta(x|z)]$ to data distribution **generative adversarial networks (GANs)**

Latent Variable Models

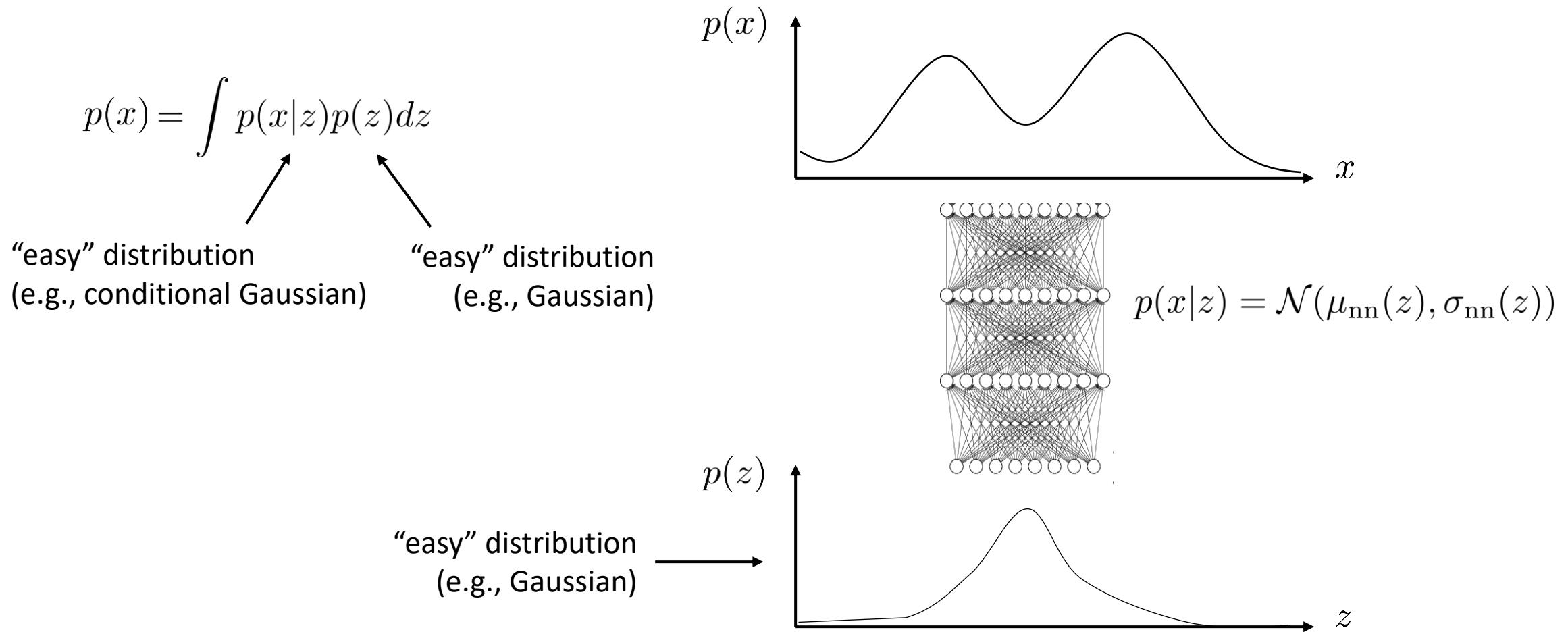
Designing, Visualizing and Understanding Deep Neural Networks

CS W182/282A

Instructor: Sergey Levine
UC Berkeley



Latent variable models in general



Estimating the log-likelihood

expected log-likelihood:

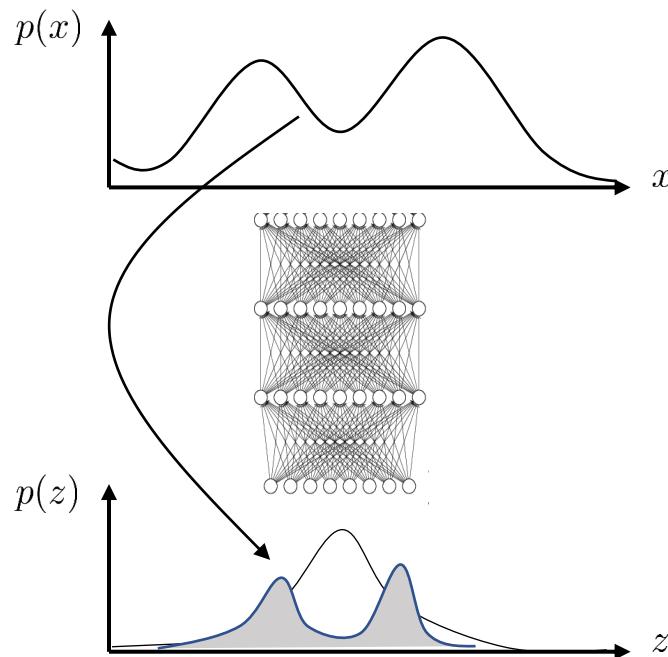
$$\theta \leftarrow \arg \max_{\theta} \frac{1}{N} \sum_i E_{z \sim p(z|x_i)} [\log p_{\theta}(x_i, z)]$$

but... how do we calculate $p(z|x_i)$?

this is called **probabilistic inference**

intuition: “guess” most likely z given x_i ,
and pretend it’s the right one

...but there are many possible values of z
so use the distribution $p(z|x_i)$



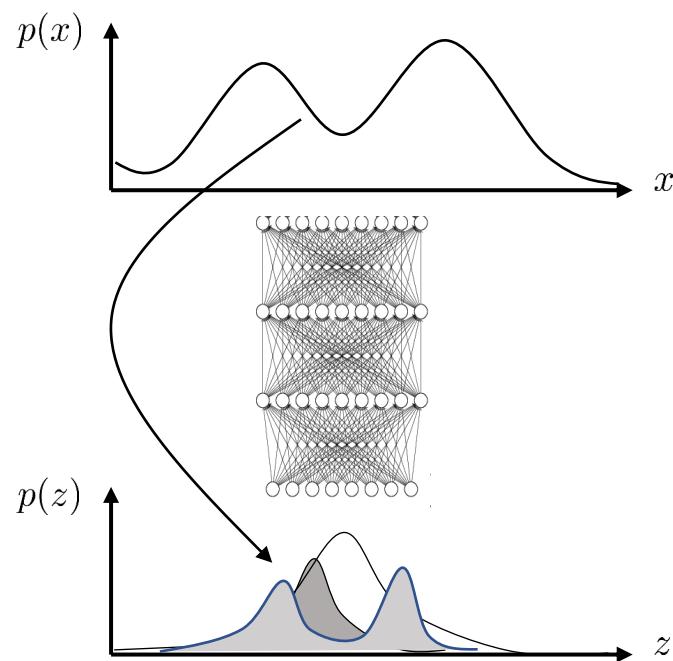
The variational approximation

but... how do we calculate $p(z|x_i)$?

can bound $\log p(x_i)!$

$$\begin{aligned}\log p(x_i) &= \log \int_z p(x_i|z)p(z) \\ &= \log \int_z p(x_i|z)p(z) \frac{q_i(z)}{q_i(z)} \\ &= \log E_{z \sim q_i(z)} \left[\frac{p(x_i|z)p(z)}{q_i(z)} \right]\end{aligned}$$

what if we approximate with $q_i(z) = \mathcal{N}(\mu_i, \sigma_i)$



The variational approximation

but... how do we calculate $p(z|x_i)$?

can bound $\log p(x_i)$!

$$\log p(x_i) = \log \int_z p(x_i|z)p(z)$$

$$= \log \int_z p(x_i|z)p(z) \frac{q_i(z)}{q_i(z)}$$

$$= \log E_{z \sim q_i(z)} \left[\frac{p(x_i|z)p(z)}{q_i(z)} \right]$$

$$\geq E_{z \sim q_i(z)} \left[\log \frac{p(x_i|z)p(z)}{q_i(z)} \right] = E_{z \sim q_i(z)} [\log p(x_i|z) + \log p(z)] + H_{\mathcal{A}q_i}(z) [\log q_i(z)]$$

Jensen's inequality

$$\log E[y] \geq E[\log y]$$

maximizing this maximizes $\log p(x_i)$



A brief aside...

Entropy:

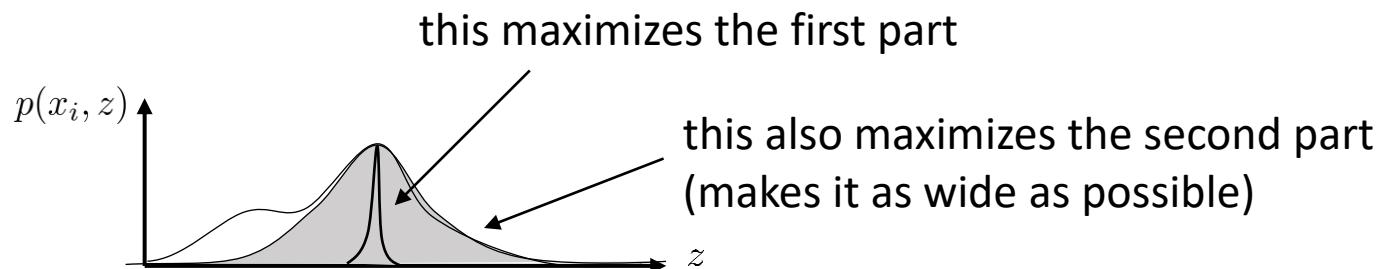
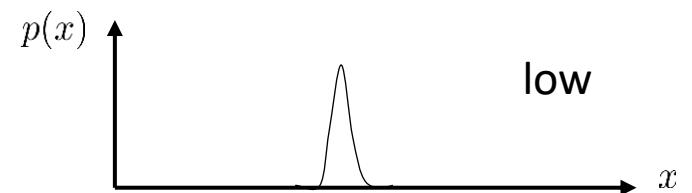
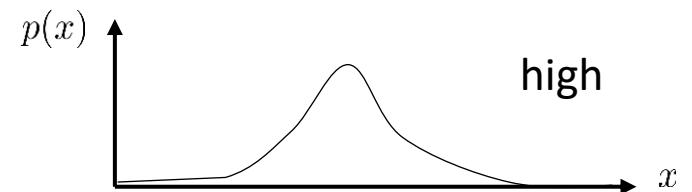
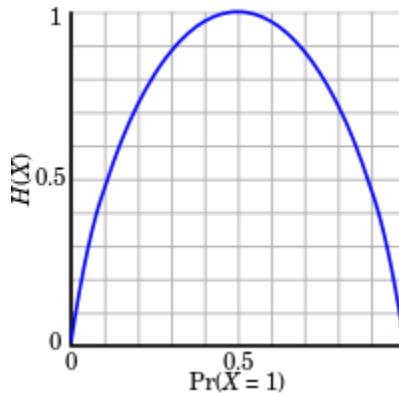
$$\mathcal{H}(p) = -E_{x \sim p(x)}[\log p(x)] = - \int_x p(x) \log p(x) dx$$

Intuition 1: how *random* is the random variable?

Intuition 2: how large is the log probability in expectation *under itself*

what do we expect this to do?

$$E_{z \sim q_i(z)}[\log p(x_i|z) + \log p(z)] + \mathcal{H}(q_i)$$



A brief aside...

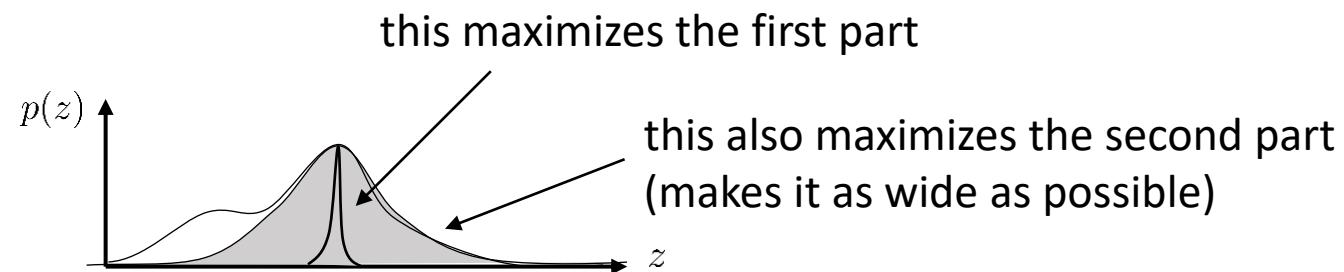
KL-Divergence:

$$D_{\text{KL}}(q \| p) = E_{x \sim q(x)} \left[\log \frac{q(x)}{p(x)} \right] = E_{x \sim q(x)} [\log q(x)] - E_{x \sim q(x)} [\log p(x)] = -E_{x \sim q(x)} [\log p(x)] - \mathcal{H}(q)$$

Intuition 1: how *different* are two distributions?

Intuition 2: how small is the expected log probability of one distribution under another, minus entropy?

why entropy?



The variational approximation

$$\log p(x_i) \geq \overbrace{E_{z \sim q_i(z)}[\log p(x_i|z) + \log p(z)] + \mathcal{H}(q_i)}^{\mathcal{L}_i(p, q_i)}$$

what makes a good $q_i(z)$?

intuition: $q_i(z)$ should approximate $p(z|x_i)$

approximate in what sense?

compare in terms of KL-divergence: $D_{\text{KL}}(q_i(z)\|p(z|x))$

why?

$$\begin{aligned} D_{\text{KL}}(q_i(z)\|p(z|x_i)) &= E_{z \sim q_i(z)} \left[\log \frac{q_i(z)}{p(z|x_i)} \right] = E_{z \sim q_i(z)} \left[\log \frac{q_i(z)p(x_i)}{p(x_i, z)} \right] \\ &= -E_{z \sim q_i(z)}[\log p(x_i|z) + \log p(z)] + E_{z \sim q_i(z)}[\log q_i(z)] + E_{z \sim q_i(z)}[\log p(x_i)] \\ &= -E_{z \sim q_i(z)}[\log p(x_i|z) + \log p(z)] - \mathcal{H}(q_i) + \log p(x_i) \\ &= -\mathcal{L}_i(p, q_i) + \log p(x_i) \end{aligned}$$

$$\log p(x_i) = D_{\text{KL}}(q_i(z)\|p(z|x_i)) + \mathcal{L}_i(p, q_i)$$

$$\log p(x_i) \geq \mathcal{L}_i(p, q_i)$$

The variational approximation

$$\log p(x_i) \geq E_{z \sim q_i(z)} [\log p(x_i|z) + \log p(z)] + \mathcal{H}(q_i)$$

$$\log p(x_i) = D_{\text{KL}}(q_i(z) \| p(z|x_i)) + \mathcal{L}_i(p, q_i)$$

$$\log p(x_i) \geq \mathcal{L}_i(p, q_i)$$

$$\begin{aligned} D_{\text{KL}}(q_i(z) \| p(z|x_i)) &= E_{z \sim q_i(z)} \left[\log \frac{q_i(z)}{p(z|x_i)} \right] = E_{z \sim q_i(z)} \left[\log \frac{q_i(z)p(x_i)}{p(x_i, z)} \right] \\ &= -E_{z \sim q_i(z)} [\log p(x_i|z) + \log p(z)] - \mathcal{H}(q_i) + \log p(x_i) \end{aligned}$$

—————
 $-\mathcal{L}_i(p, q_i)$ independent of q_i !

\Rightarrow maximizing $\mathcal{L}_i(p, q_i)$ w.r.t. q_i minimizes KL-divergence!

How do we use this?

$$\log p(x_i) \geq E_{z \sim q_i(z)}[\log p_\theta(x_i|z) + \log p(z)] + \mathcal{H}(q_i)$$

$$\theta \leftarrow \arg \max_{\theta} \frac{1}{N} \sum_i \log p_\theta(x_i)$$

for each x_i (or mini-batch):

calculate $\nabla_{\theta} \mathcal{L}_i(p, q_i)$:

sample $z \sim q_i(z)$

$$\nabla_{\theta} \mathcal{L}_i(p, q_i) \approx \nabla_{\theta} \log p_\theta(x_i|z)$$

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} \mathcal{L}_i(p, q_i)$$

update q_i to maximize $\mathcal{L}_i(p, q_i)$

$$\theta \leftarrow \arg \max_{\theta} \frac{1}{N} \sum_i \mathcal{L}_i(p, q_i)$$

let's say $q_i(z) = \mathcal{N}(\mu_i, \sigma_i)$

use gradient $\nabla_{\mu_i} \mathcal{L}_i(p, q_i)$ and $\nabla_{\sigma_i} \mathcal{L}_i(p, q_i)$

gradient ascent on μ_i, σ_i

how?

What's the problem?

for each x_i (or mini-batch):

calculate $\nabla_{\theta} \mathcal{L}_i(p, q_i)$:

sample $z \sim q_i(z)$

$$\nabla_{\theta} \mathcal{L}_i(p, q_i) \approx \nabla_{\theta} \log p_{\theta}(x_i|z)$$

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} \mathcal{L}_i(p, q_i)$$

update q_i to maximize $\mathcal{L}_i(p, q_i)$

let's say $q_i(z) = \mathcal{N}(\mu_i, \sigma_i)$

use gradient $\nabla_{\mu_i} \mathcal{L}_i(p, q_i)$ and $\nabla_{\sigma_i} \mathcal{L}_i(p, q_i)$

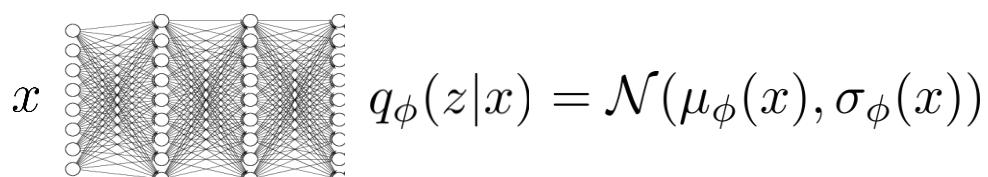
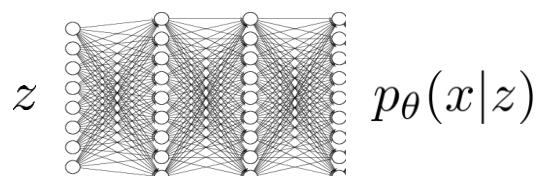
gradient ascent on μ_i, σ_i

How many parameters are there?

$$|\theta| + (|\mu_i| + |\sigma_i|) \times N$$

intuition: $q_i(z)$ should approximate $p(z|x_i)$

what if we learn a *network* $q_i(z) = q(z|x_i) \approx p(z|x_i)$?



Amortized Variational Inference

What's the problem?

for each x_i (or mini-batch):

calculate $\nabla_{\theta} \mathcal{L}_i(p, q_i)$:

sample $z \sim q_i(z)$

$$\nabla_{\theta} \mathcal{L}_i(p, q_i) \approx \nabla_{\theta} \log p_{\theta}(x_i|z)$$

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} \mathcal{L}_i(p, q_i)$$

update q_i to maximize $\mathcal{L}_i(p, q_i)$

let's say $q_i(z) = \mathcal{N}(\mu_i, \sigma_i)$

use gradient $\nabla_{\mu_i} \mathcal{L}_i(p, q_i)$ and $\nabla_{\sigma_i} \mathcal{L}_i(p, q_i)$

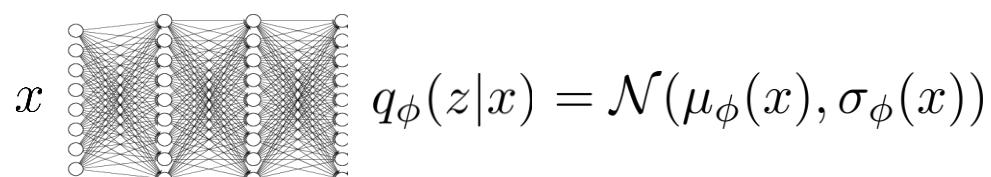
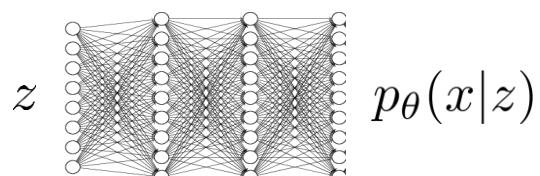
gradient ascent on μ_i, σ_i

How many parameters are there?

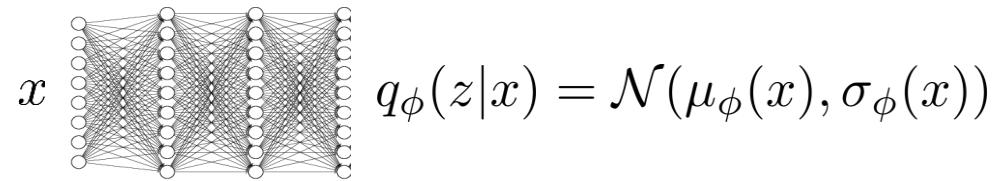
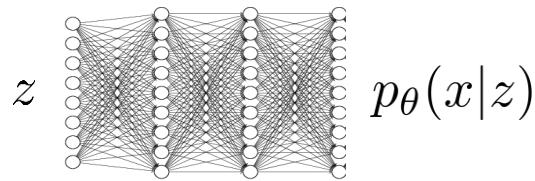
$$|\theta| + (|\mu_i| + |\sigma_i|) \times N$$

intuition: $q_i(z)$ should approximate $p(z|x_i)$

what if we learn a *network* $q_i(z) = q(z|x_i) \approx p(z|x_i)$?



Amortized variational inference



for each x_i (or mini-batch):

calculate $\nabla_\theta \mathcal{L}(p_\theta(x_i|z), q_\phi(z|x_i))$:

sample $z \sim q_\phi(z|x_i)$

$\nabla_\theta \mathcal{L} \approx \nabla_\theta \log p_\theta(x_i|z)$

$\theta \leftarrow \theta + \alpha \nabla_\theta \mathcal{L}$

$\phi \leftarrow \phi + \alpha \nabla_\phi \mathcal{L}$

$$\log p(x_i) \geq \underbrace{E_{z \sim q_\phi(z|x_i)} [\log p_\theta(x_i|z) + \log p(z)] + \mathcal{H}(q_\phi(z|x_i))}_{\mathcal{L}(p_\theta(x_i|z), q_\phi(z|x_i))}$$

how do we calculate this?

Amortized variational inference

for each x_i (or mini-batch):

calculate $\nabla_{\theta} \mathcal{L}(p_{\theta}(x_i|z), q_{\phi}(z|x_i))$:

sample $z \sim q_{\phi}(z|x_i)$

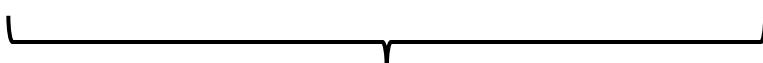
$\nabla_{\theta} \mathcal{L} \approx \nabla_{\theta} \log p_{\theta}(x_i|z)$

$\theta \leftarrow \theta + \alpha \nabla_{\theta} \mathcal{L}$

$\phi \leftarrow \phi + \alpha \nabla_{\phi} \mathcal{L}$

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

$$\mathcal{L}_i = E_{z \sim q_{\phi}(z|x_i)} [\log p_{\theta}(x_i|z) + \log p(z)] + \mathcal{H}(q_{\phi}(z|x_i))$$



$$J(\phi) = E_{z \sim q_{\phi}(z|x_i)} [r(x_i, z)]$$

can just use policy gradient!

What's wrong with this gradient?

$$\nabla J(\phi) \approx \frac{1}{M} \sum_j \nabla_{\phi} \log q_{\phi}(z_j|x_i) r(x_i, z_j)$$

look up formula for
entropy of a Gaussian



The reparameterization trick

Is there a better way?

$$J(\phi) = E_{z \sim q_\phi(z|x_i)}[r(x_i, z)]$$

$$= E_{\epsilon \sim \mathcal{N}(0,1)}[r(x_i, \mu_\phi(x_i) + \epsilon \sigma_\phi(x_i))]$$

estimating $\nabla_\phi J(\phi)$:

sample $\epsilon_1, \dots, \epsilon_M$ from $\mathcal{N}(0, 1)$ (a single sample works well!)

$$\nabla_\phi J(\phi) \approx \frac{1}{M} \sum_j \nabla_\phi r(x_i, \mu_\phi(x_i) + \epsilon_j \sigma_\phi(x_i))$$

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

$$z = \mu_\phi(x) + \epsilon \sigma_\phi(x)$$



$$\epsilon \sim \mathcal{N}(0, 1)$$

independent of ϕ !

most autodiff software (e.g., TensorFlow)
will compute this for you!

Another way to look at it...

$$\mathcal{L}_i = E_{z \sim q_\phi(z|x_i)} [\log p_\theta(x_i|z) + \log p(z)] + \mathcal{H}(q_\phi(z|x_i))$$

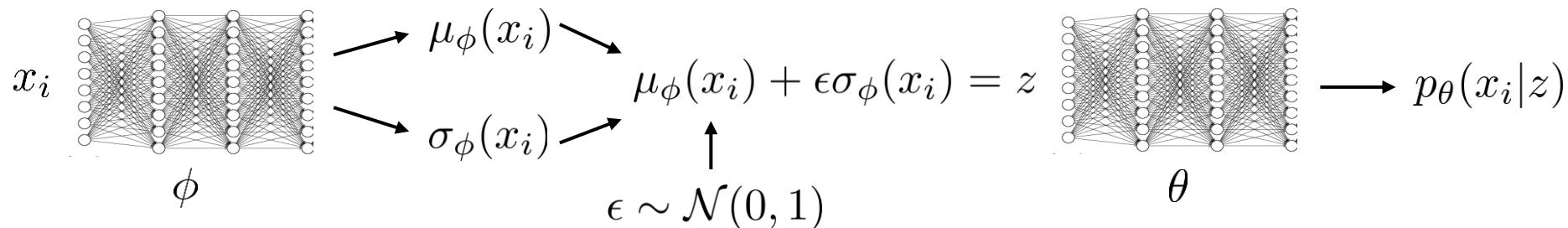
$$= E_{z \sim q_\phi(z|x_i)} [\log p_\theta(x_i|z)] + \underbrace{E_{z \sim q_\phi(z|x_i)} [\log p(z)] + \mathcal{H}(q_\phi(z|x_i))}_{-D_{\text{KL}}(q_\phi(z|x_i) \| p(z))}$$

$-D_{\text{KL}}(q_\phi(z|x_i) \| p(z))$ ← this often has a convenient analytical form (e.g., KL-divergence for Gaussians)

$$= E_{z \sim q_\phi(z|x_i)} [\log p_\theta(x_i|z)] - D_{\text{KL}}(q_\phi(z|x_i) \| p(z))$$

$$= E_{\epsilon \sim \mathcal{N}(0,1)} [\log p_\theta(x_i|\mu_\phi(x_i) + \epsilon\sigma_\phi(x_i))] - D_{\text{KL}}(q_\phi(z|x_i) \| p(z))$$

$$\approx \log p_\theta(x_i|\mu_\phi(x_i) + \epsilon\sigma_\phi(x_i)) - D_{\text{KL}}(q_\phi(z|x_i) \| p(z))$$



Reparameterization trick vs. policy gradient

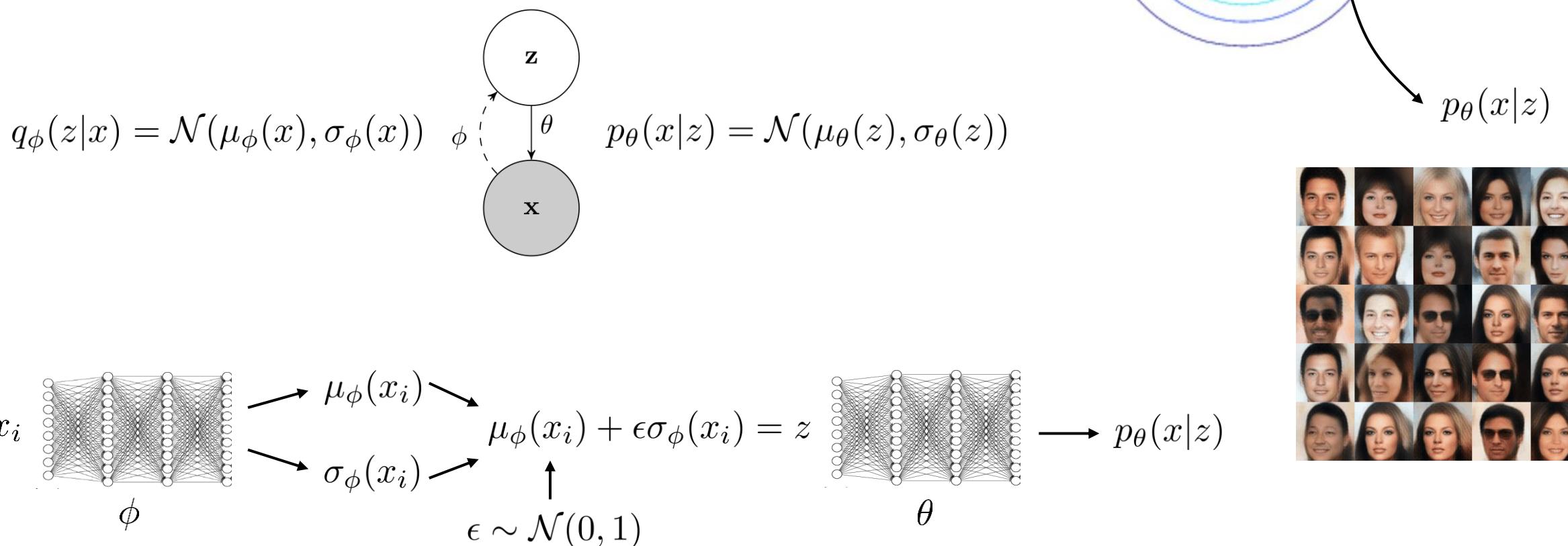
- Policy gradient
 - Can handle both discrete and continuous latent variables
 - High variance, requires multiple samples & small learning rates
- Reparameterization trick
 - Only continuous latent variables
 - Very simple to implement
 - Low variance

$$J(\phi) \approx \frac{1}{M} \sum_j \nabla_\phi \log q_\phi(z_j | x_i) r(x_i, z_j)$$

$$\nabla_\phi J(\phi) \approx \frac{1}{M} \sum_j \nabla_\phi r(x_i, \mu_\phi(x_i) + \epsilon_j \sigma_\phi(x_i))$$

Variational Autoencoders

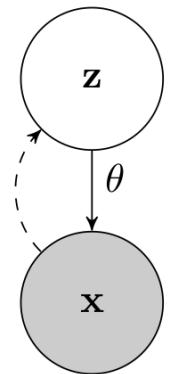
The *variational* autoencoder



$$\max_{\theta, \phi} \frac{1}{N} \sum_i \log p_\theta(x_i | \mu_\phi(x_i) + \epsilon\sigma_\phi(x_i)) - D_{\text{KL}}(q_\phi(z|x_i) \| p(z))$$

Using the variational autoencoder

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



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

$$p(x) = \int p(x|z)p(z)dz$$

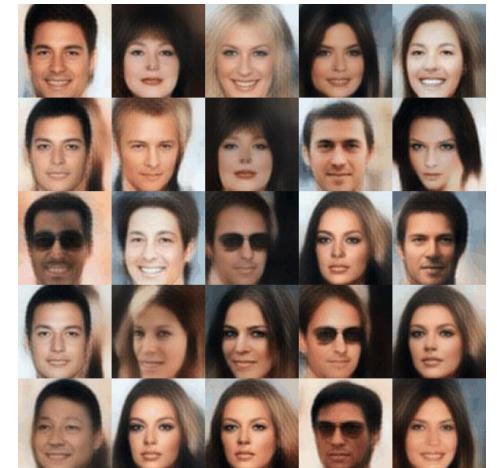
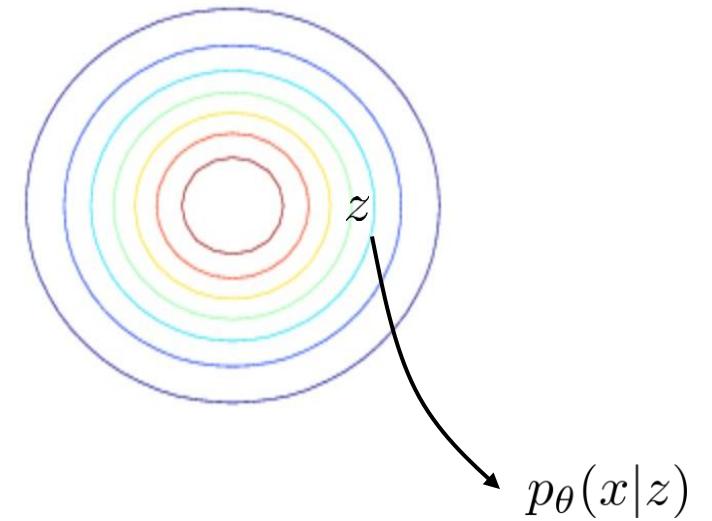
why does this work?

sampling:

$$z \sim p(z)$$

$$x \sim p(x|z)$$

$$\mathcal{L}_i = E_{z \sim q_\phi(z|x_i)}[\log p_\theta(x_i|z)] - D_{\text{KL}}(q_\phi(z|x_i) \| p(z))$$



Conditional models

$$\mathcal{L}_i = E_{z \sim q_\phi(z|x_i, y_i)} [\log p_\theta(y_i|x_i, z) + \log p(z|x_i)] + \mathcal{H}(q_\phi(z|x_i, y_i))$$



just like before, only now generating y_i
and *everything* is conditioned on x_i

at test time:

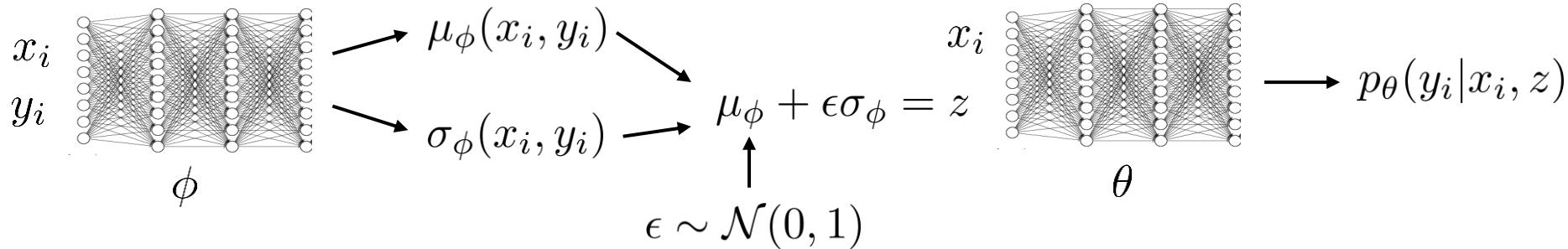
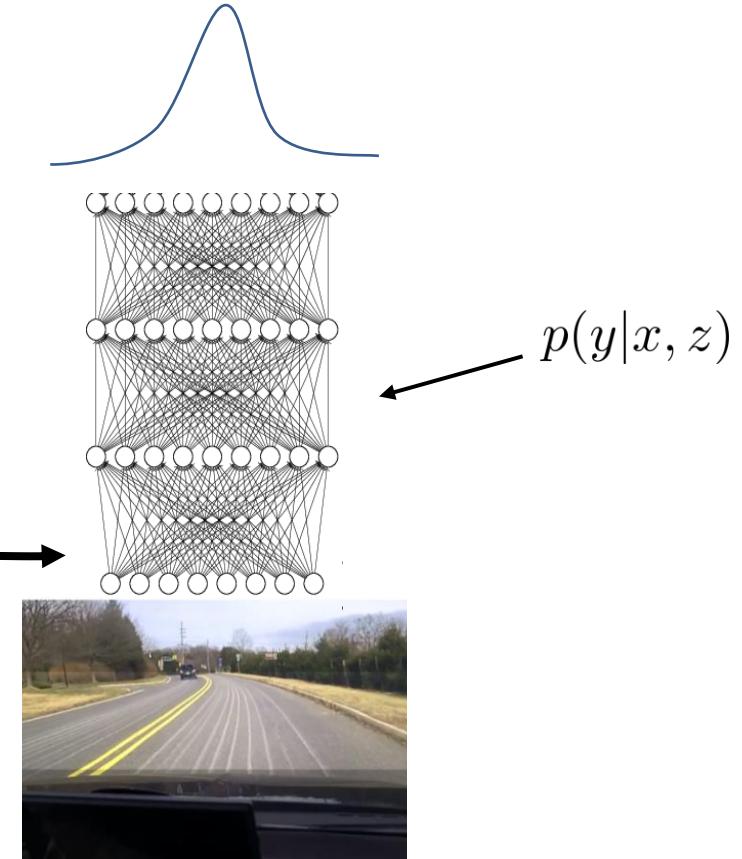
$$z \sim p(z|x_i)$$

$$y \sim p(y|x_i, z)$$

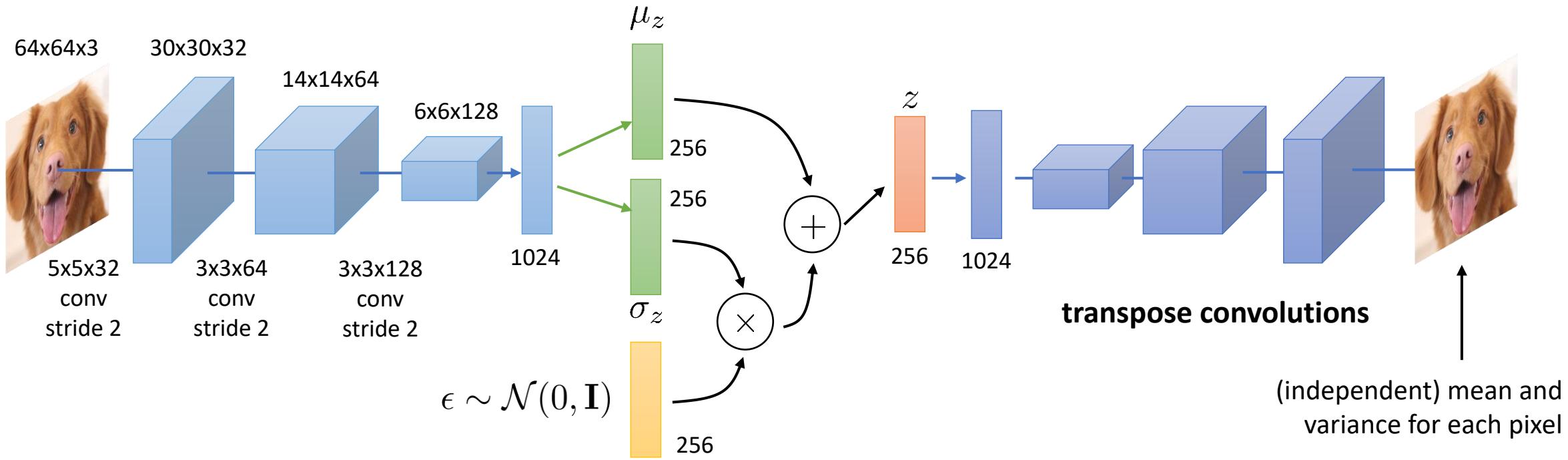
can *optionally* depend on x

$$z \sim \mathcal{N}(0, \mathbf{I})$$

$$p(z)$$



VAEs with convolutions



Question: can we design a **fully convolutional** VAE?

Yes, but be careful with the latent codes!

$p(z) = \mathcal{N}(0, \mathbf{I})$ \longleftarrow implies all z dimensions are independent

VAEs in practice

Common issue: very tempting for VAEs (especially **conditional** VAEs) to ignore the latent codes, or generate poor samples

↑
why?

Problem 1: latent code is ignored

$$p_{\theta}(x|z) \rightarrow p(x)$$

what does this look like? blurry “average” image
 when *reconstructing*
 $z \sim q_{\phi}(z|x)$ $x \sim p_{\theta}(x|z)$

Problem 2: latent code is not *compressed*

$$q_{\phi}(z|x) \text{ very far from } p(z)$$

what does this look like? garbage images
 when *sampling*
 $z \sim p(z)$ $x \sim p_{\theta}(x|z)$

too low no info in z $D_{\text{KL}}(q_{\phi}(z|x)\|p(z))$

too high too much info in z

↑
need to control this quantity
carefully to get good results!

VAEs in practice

Problem 1: latent code is ignored

too low no info in z

$$D_{\text{KL}}(q_{\phi}(z|x) \| p(z))$$

Problem 2: latent code is not *compressed*

too high too much info in z



need to control this quantity
carefully to get good results!

$$\max_{\theta, \phi} \frac{1}{N} \sum_i \log p_{\theta}(x_i | \mu_{\phi}(x_i) + \epsilon \sigma_{\phi}(x_i)) - \beta D_{\text{KL}}(q_{\phi}(z|x_i) \| p(z))$$



multiplier to adjust regularizer strength

adjust β manually to get good reconstructions **and** good samples

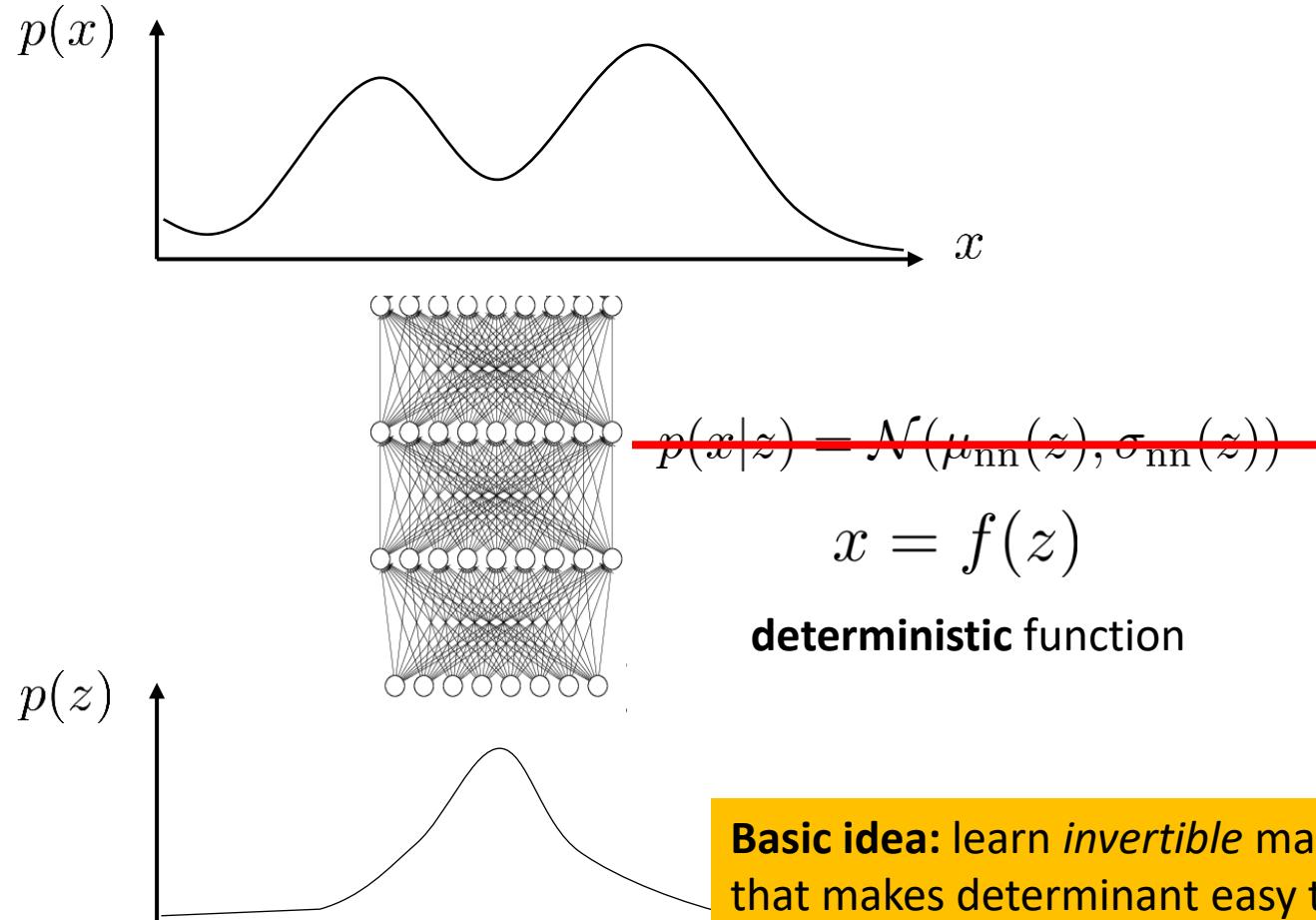
could **schedule** β

start low (to get VAE to use z to reconstruct)

end high (to get samples to be good)

Invertible Models and Normalizing Flows

A simpler kind of model



Why is this such a big deal?

change of variables formula:

$$p(x) = p(z) \left| \det \left(\frac{df(z)}{dz} \right) \right|^{-1}$$

where $z = f^{-1}(x)$

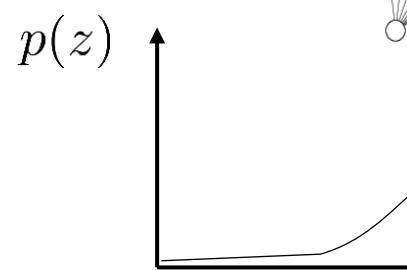
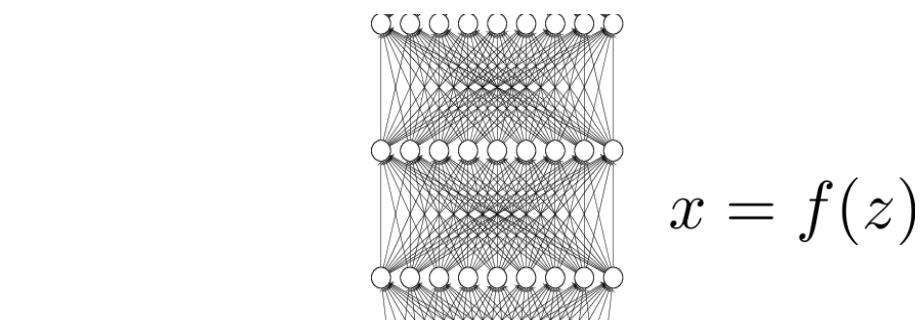
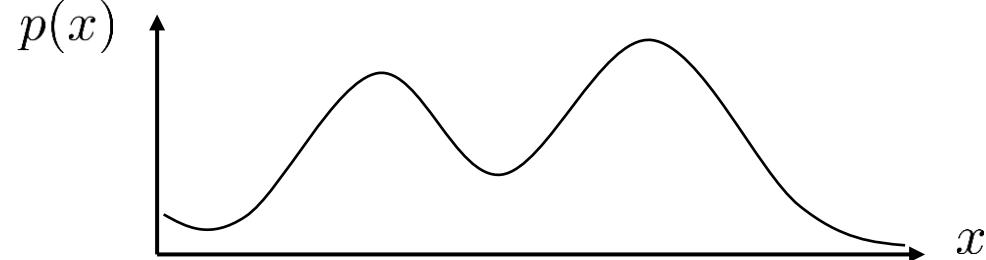


correction for change in local density due to f

Basic idea: learn *invertible* mapping from z to x that makes determinant easy to compute

No more need for lower bounds! Can get exact probabilities/likelihoods!

Normalizing flow models



A **normalizing flow** model consists of multiple layers of invertible transformations

We need to figure out how to make an invertible layer, and then compose many of them to make a deep network

Training objective:

$$\max_{\theta} \frac{1}{N} \sum_{i=1}^N \log p(x_i)$$



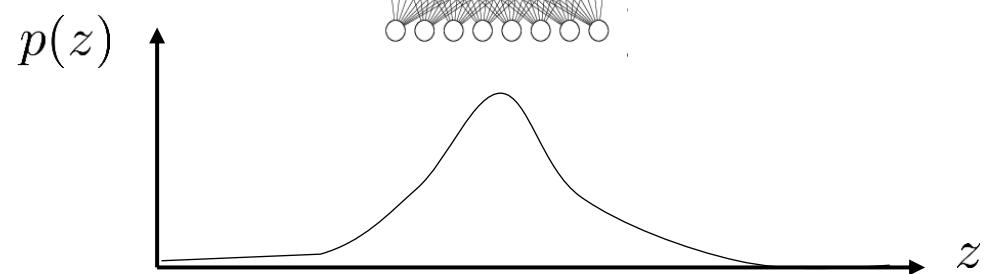
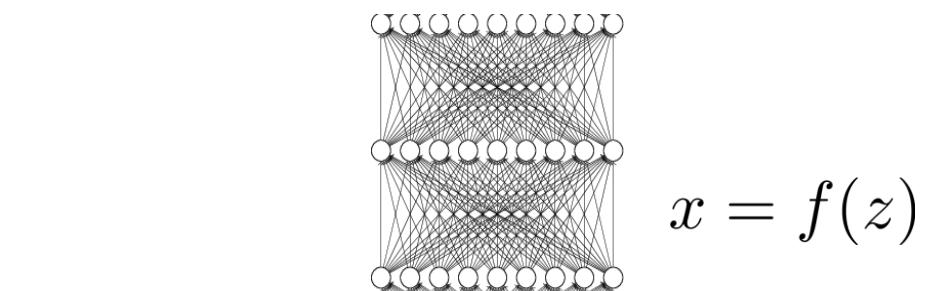
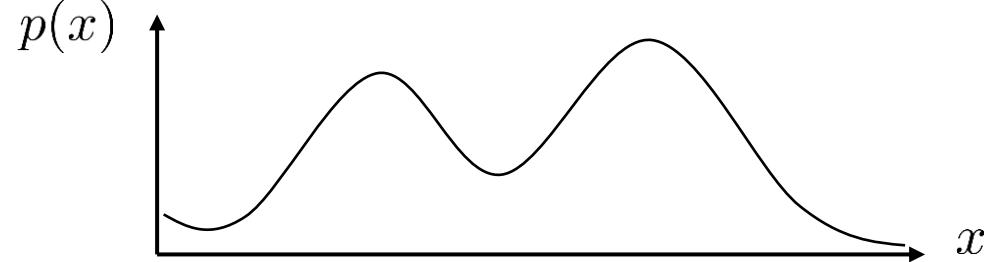
$$\max_{\theta} \frac{1}{N} \sum_{i=1}^N \log p(f^{-1}(x_i)) - \log \left| \det \left(\frac{df(z)}{dz} \right) \right|$$

choose a **special** architecture that makes these easy to compute

$$p(x) = p(z) \left| \det \left(\frac{df(z)}{dz} \right) \right|^{-1}$$

where $z = f^{-1}(x)$

Normalizing flow models



$$\max_{\theta} \frac{1}{N} \sum_{i=1}^N \log p(f^{-1}(x_i)) - \log \left| \det \left(\frac{df(z)}{dz} \right) \right|$$

$$f(z) = f_4(f_3(f_2(f_1(z))))$$

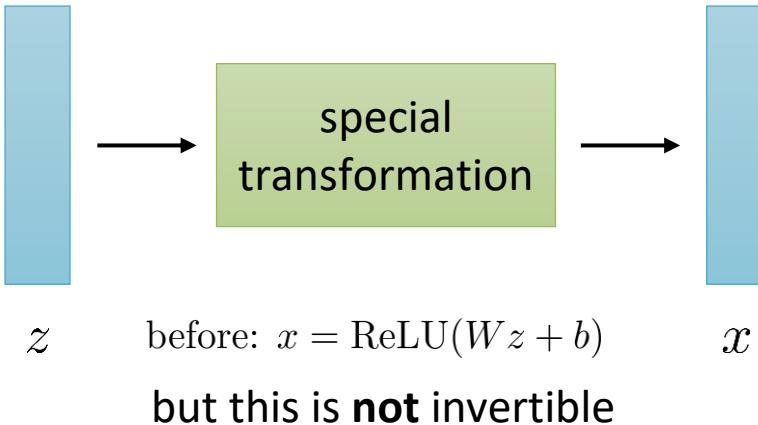
If each **layer** is invertible, the whole thing is invertible

Oftentimes, invertible layers also have very convenient determinants

Log-determinant of whole model is just the sum of log-determinants of the layers

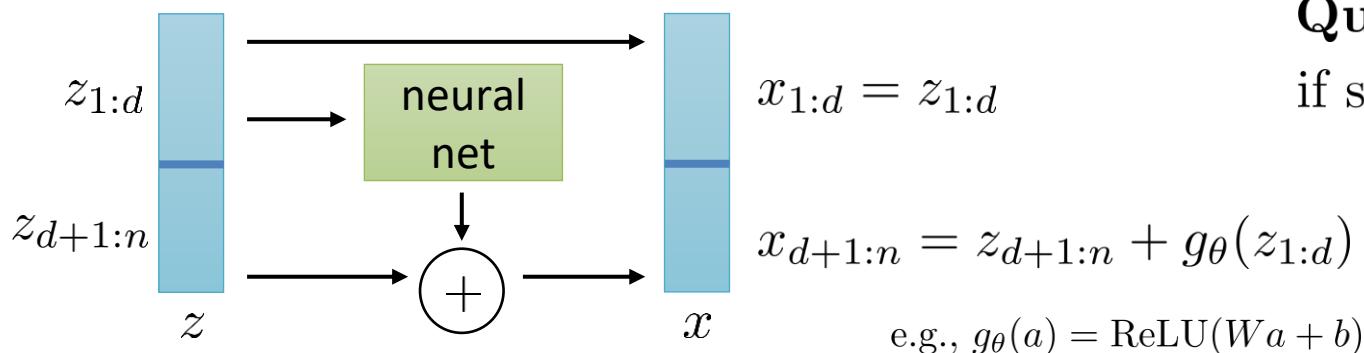
Goal: design an invertible layer, and then compose many of them to create a fully invertible neural net

NICE: Nonlinear Independent Components Estimation



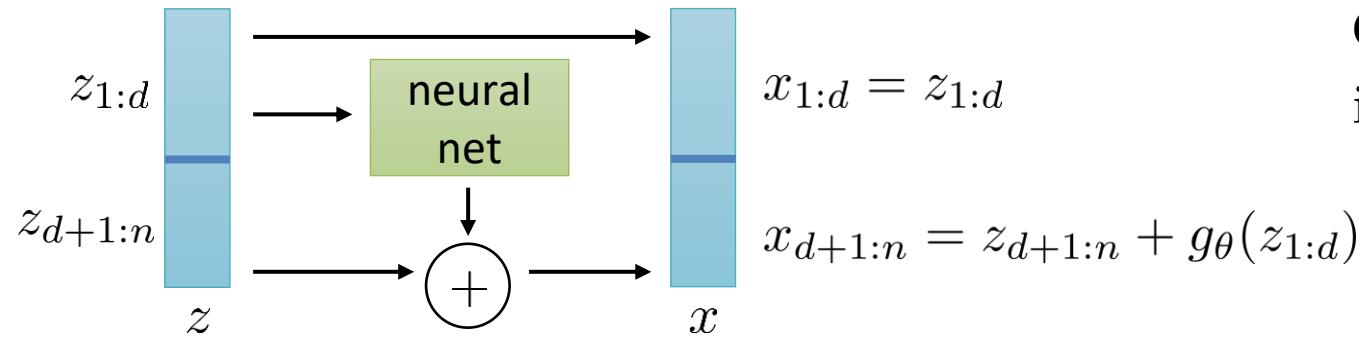
Idea: what if we force **part** of the layer to keep all the information so that we can then recover anything that was changed by the nonlinear transformation?

Important: here I describe the case for **one** layer, but in reality we'll have many layers!

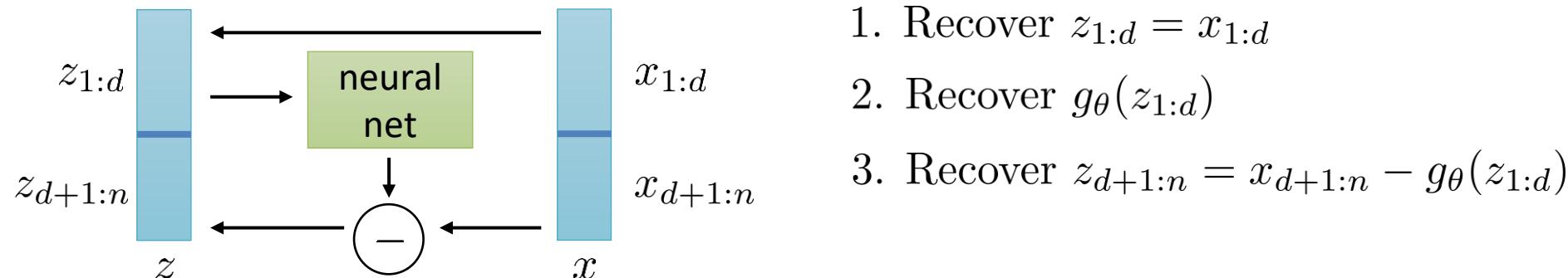


Question: if we have x , can we recover z ?
if so, then this layer is **invertible**

NICE: Nonlinear Independent Components Estimation

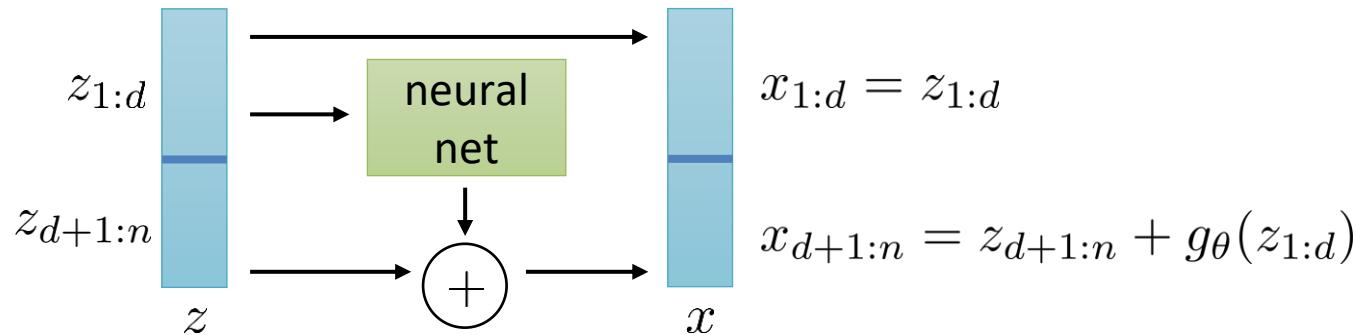


Question: if we have x , can we recover z ?
if so, then this layer is **invertible**



1. Recover $z_{1:d} = x_{1:d}$
2. Recover $g_\theta(z_{1:d})$
3. Recover $z_{d+1:n} = x_{d+1:n} - g_\theta(z_{1:d})$

What about the Jacobian?



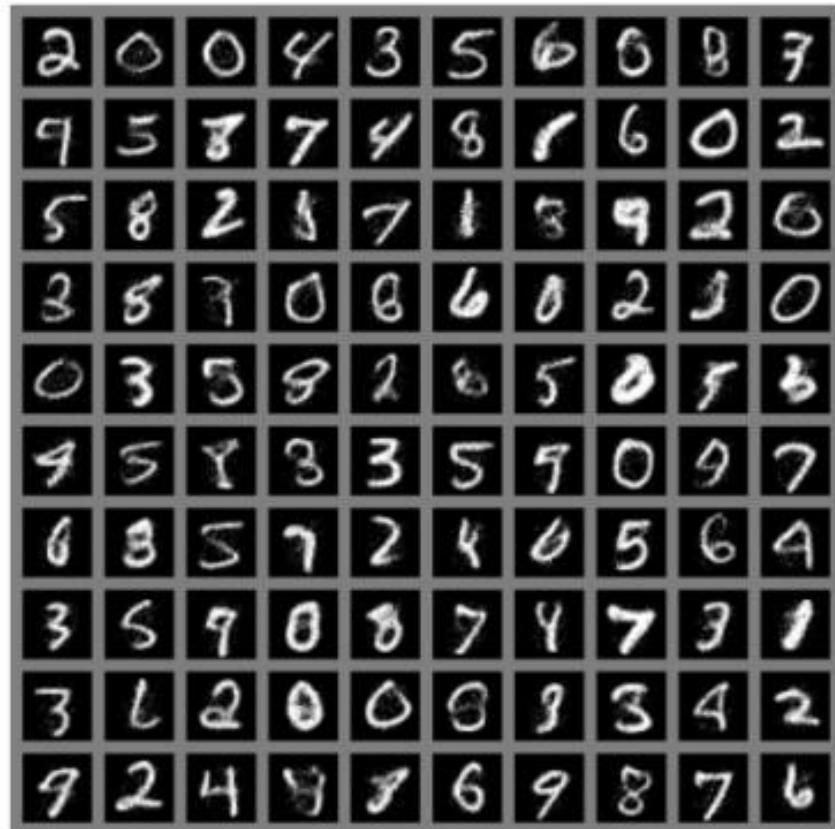
$$\left| \det \left(\frac{df(z)}{dz} \right) \right| = 1$$

This is very simple and convenient

But it's representationally a bit limiting

$$\frac{df(z)}{dz} = \begin{bmatrix} \frac{\partial x_{1:d}}{\partial z_{1:d}} & \frac{\partial x_{1:d}}{\partial z_{d+1:n}} \\ \frac{\partial x_{d+1:n}}{\partial z_{1:d}} & \frac{\partial x_{d+1:n}}{\partial z_{d+1:n}} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & 0 \\ \frac{\partial g_\theta}{\partial z_{1:d}} & \mathbf{I} \end{bmatrix}$$

NICE: Nonlinear Independent Components Estimation



(a) Model trained on MNIST



(b) Model trained on TFD

NICE: Nonlinear Independent Components Estimation

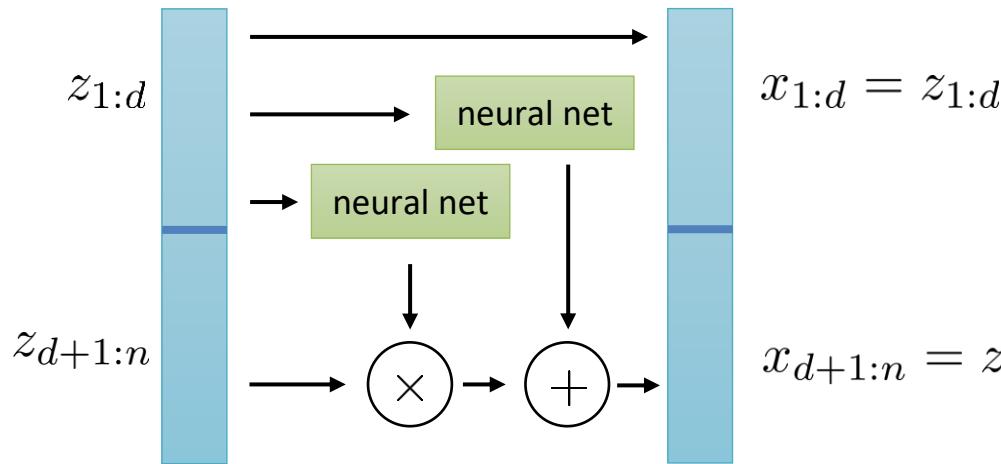


(c) Model trained on SVHN



(d) Model trained on CIFAR-10

Real-NVP: Non-Volume Preserving Transformation



Inverse can be derived in the same way as before:

1. Recover $z_{1:d} = x_{1:d}$
2. Recover $g_\theta(z_{1:d})$ and $h_\theta(z_{1:d})$
3. Recover $z_{d+1:n} = (x_{d+1:n} - g_\theta(z_{1:d})) / \exp(h_\theta(z_{1:d}))$

$$x_{d+1:n} = z_{d+1:n} \times \exp(h_\theta(z_{1:d})) + g_\theta(z_{1:d})$$

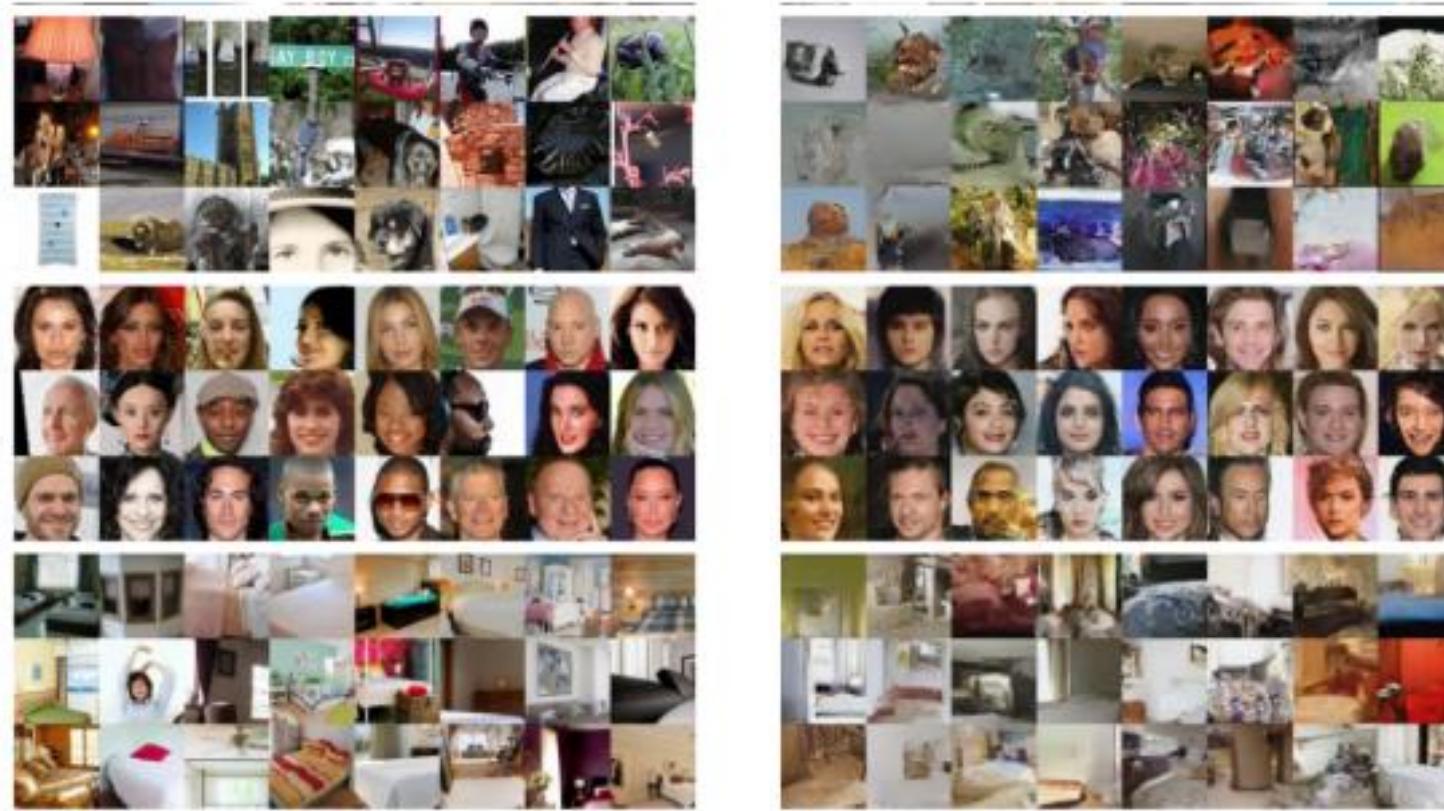
elementwise product

$$\left| \det \left(\frac{df(z)}{dz} \right) \right| = \prod_{i=d+1}^n \exp(h_\theta(z_{1:d})_i)$$

$$\frac{df(z)}{dz} = \begin{bmatrix} \mathbf{I} & 0 \\ \frac{dx_{d+1:n}}{dz_{1:d}} & \text{diag}(\exp(h_\theta(z_{1:d}))) \end{bmatrix}$$

This is significantly more expressive

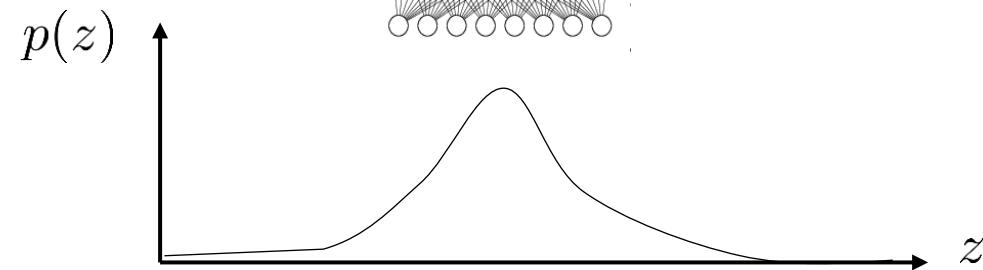
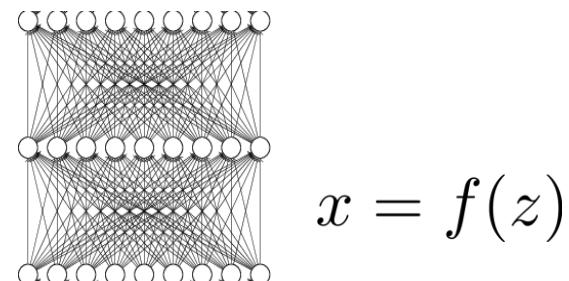
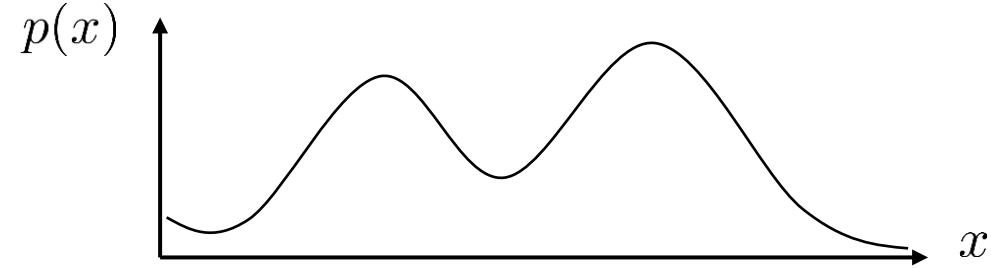
Real-NVP Samples



Material based on Grover & Ermon CS236

Dinh et al. **Density estimation using Real-NVP**. 2016.

Concluding Remarks



- + can get exact probabilities/likelihoods
- + no need for lower bounds
- + conceptually simpler (perhaps)
- requires special architecture
- Z must have same dimensionality as X

Generative Adversarial Networks

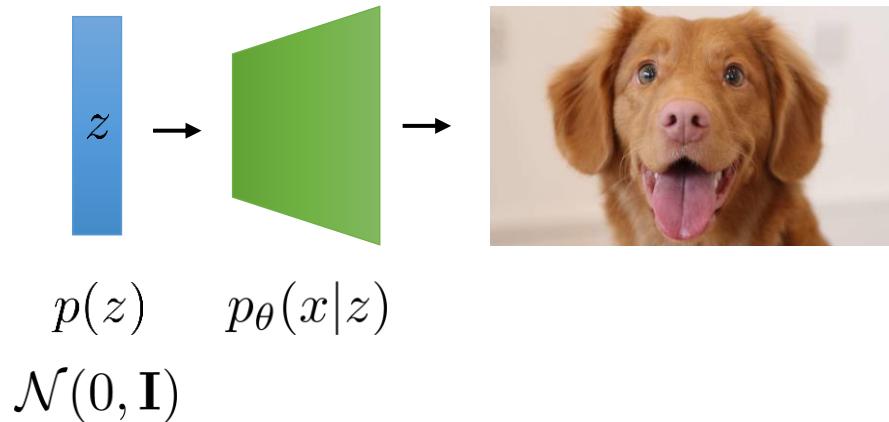
Designing, Visualizing and Understanding Deep Neural Networks

CS W182/282A

Instructor: Sergey Levine
UC Berkeley



Back to latent variable models



Idea: instead of training an encoder, can we just train the whole model to generate images that look similar to real images *at the population level?*

Using the model for **generation**:

1. sample $z \sim p(z)$ “generate a vector of random numbers”
2. sample $x \sim p(x|z)$ “turn that vector of random numbers into an image”

Matching distributions at the population level

no two faces are the same, but they look similar **at the population level**

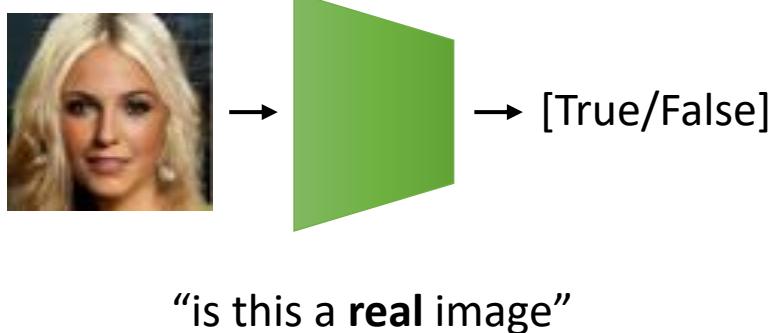


which set of faces is real?

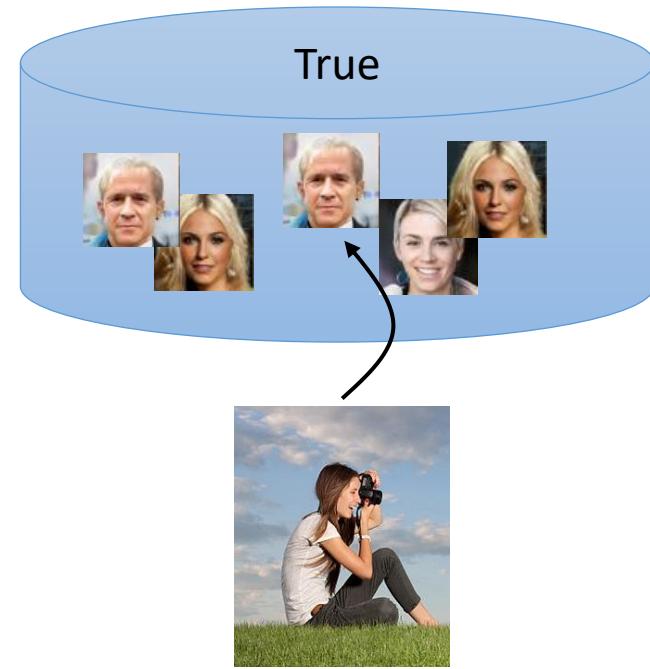
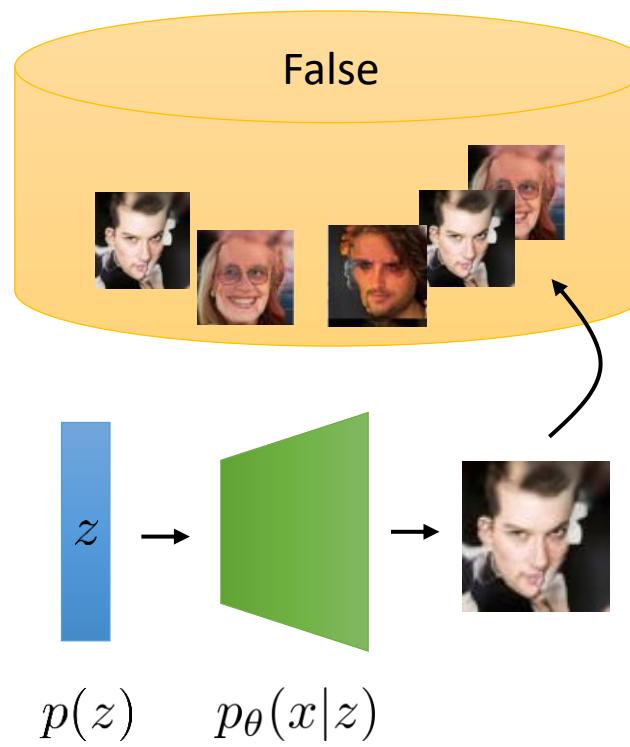
it's a trick question...

The only winning move is to generate

Idea: train a **network** to guess which images are real and which are fake!



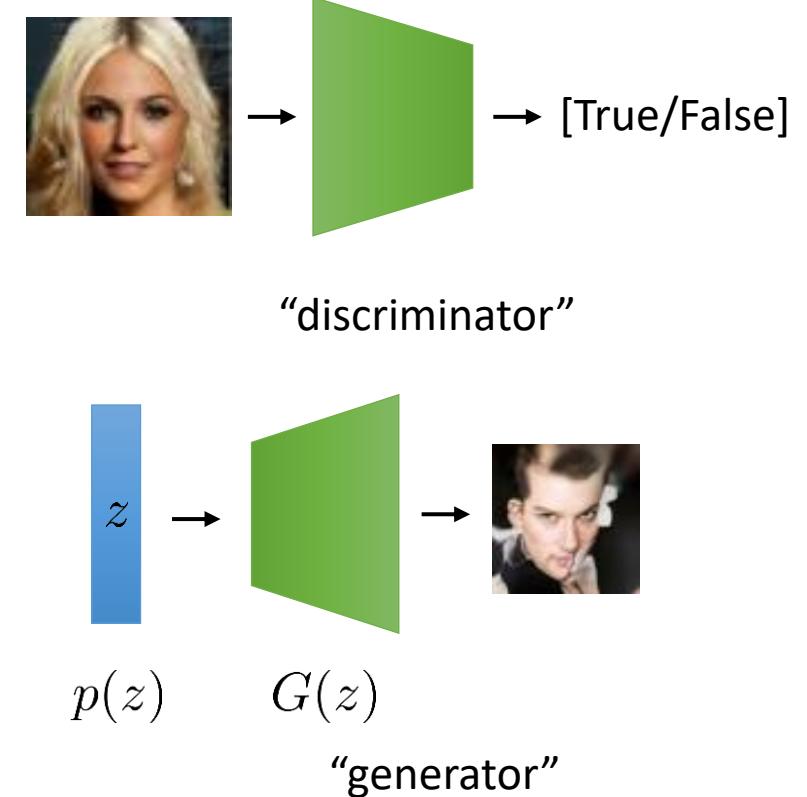
This model can then serve as a loss function for the generator!



The only winning move is to generate

1. get a “True” dataset $\mathcal{D}_T = \{(x_i)\}$
2. get a generator $G_\theta(z)$ (how?)
3. sample a “False” dataset $\mathcal{D}_F: z \sim p(z), x = G(z)$
4. train a discriminator $D_\phi(x) = p_\phi(y|x)$ using \mathcal{D}_T and \mathcal{D}_F
5. use $-\log D(x)$ as “loss” to train $G(z)$

if only done once, too easy for $G(z)$ to “fool” $D(x)$

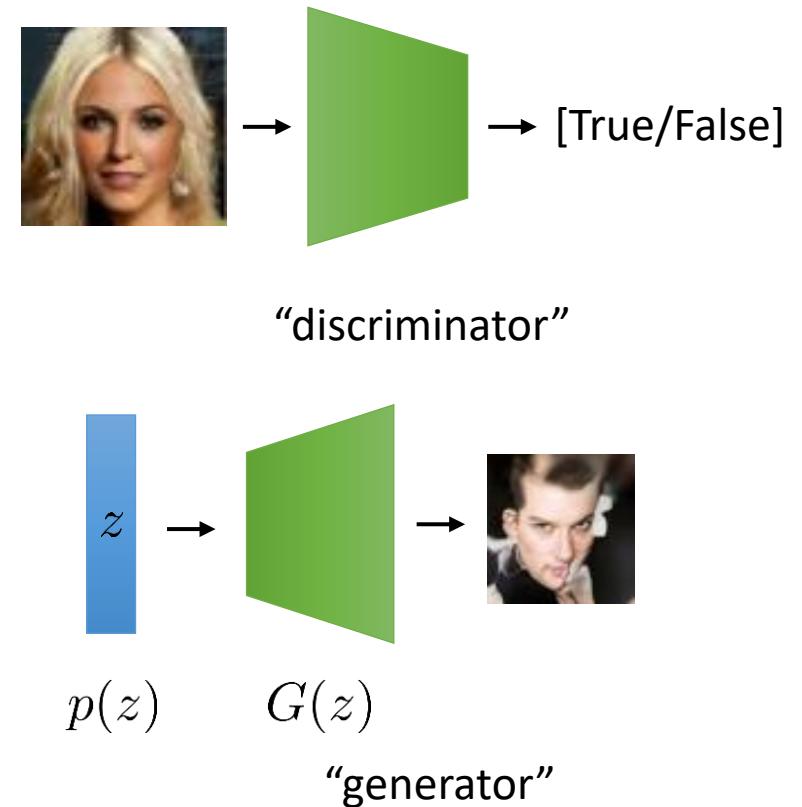


this **almost** works, but has two major problems

The only winning move is to generate

1. get a “True” dataset $\mathcal{D}_T = \{(x_i)\}$
2. get a generator $G_\theta(z)$ random initialization!
3. sample a “False” dataset \mathcal{D}_F : $z \sim p(z), x = G(z)$
4. update $D_\phi(x) = p_\phi(y|x)$ using \mathcal{D}_T and \mathcal{D}_F (1 SGD step)
5. use $-\log D(x)$ as “loss” to update $G(z)$ (1 SGD step)
(in reality there are a variety of different losses, but similar idea...)

this is called a **generative adversarial network (GAN)**



Why do GANs learn distributions?

1. get a “True” dataset $\mathcal{D}_T = \{(x_i)\}$
2. get a generator $G_\theta(z)$ random initialization!
3. sample a “False” dataset \mathcal{D}_F : $z \sim p(z), x = G(z)$
4. update $D_\phi(x) = p_\phi(y|x)$ using \mathcal{D}_T and \mathcal{D}_F (1 SGD step)
5. use $-\log D(x)$ as “loss” to update $G(z)$ (1 SGD step)
(in reality there are a variety of different losses, but similar idea...)

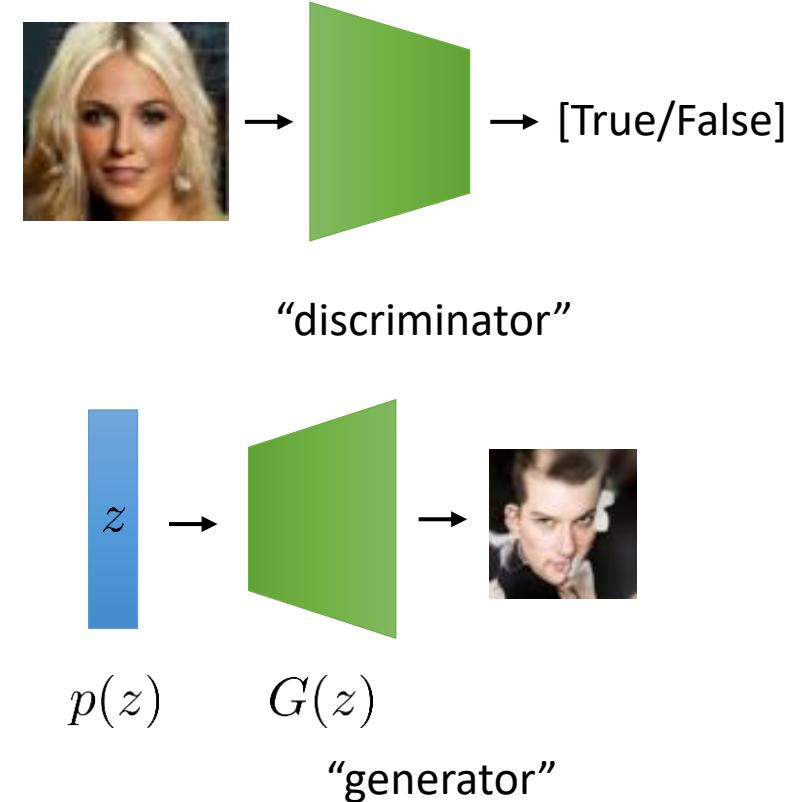
what does $G(z)$ want to do?

make $D(x) = 0.5$ for all generated x “can’t tell if real or fake”

How to do this?

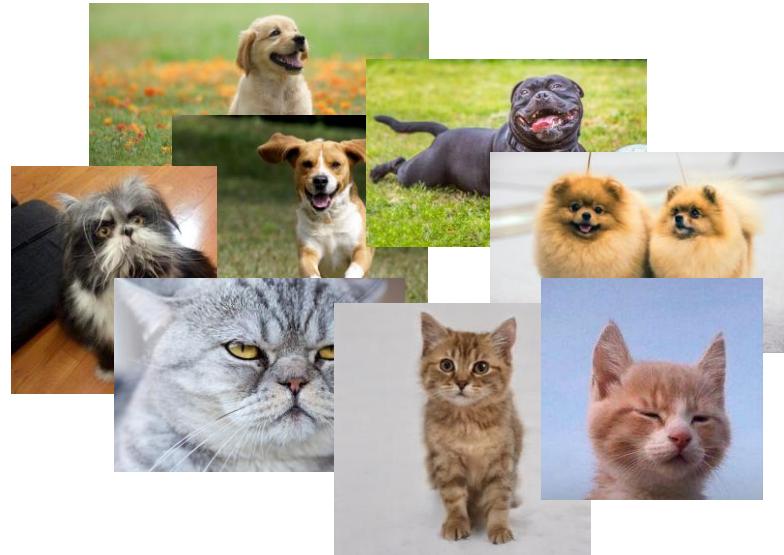
- Generate images that look realistic (obviously)
- Generate **all** possible realistic images

why?

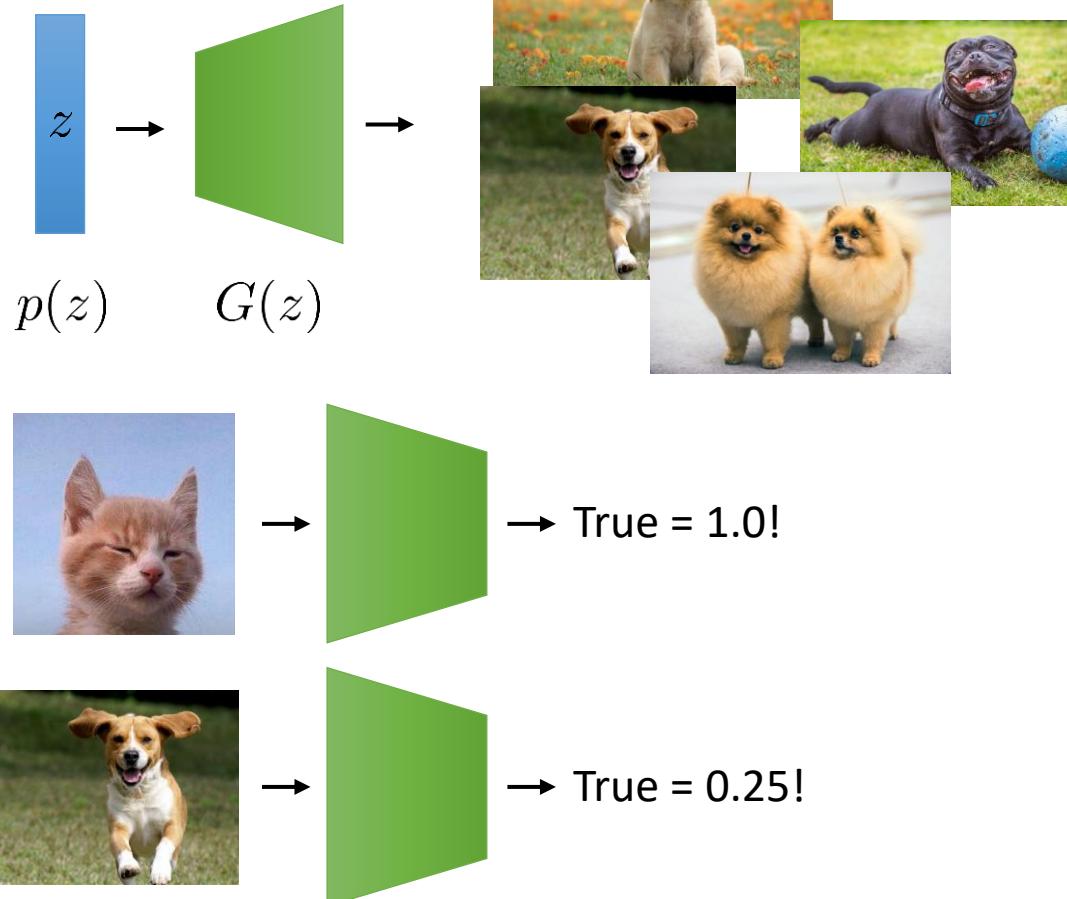


Why do GANs learn distributions?

- Generate **all** possible realistic images



why?



The generator will do **better** if it not only generates realistic pictures, but if it generates **all** realistic pictures

Small GANs

real pictures



a)



b)



c)



d)

High-res GANs



Figure 5: 1024×1024 images generated using the CELEBA-HQ dataset. See Appendix F for a larger set of results, and the accompanying video for latent space interpolations.

Big GANs

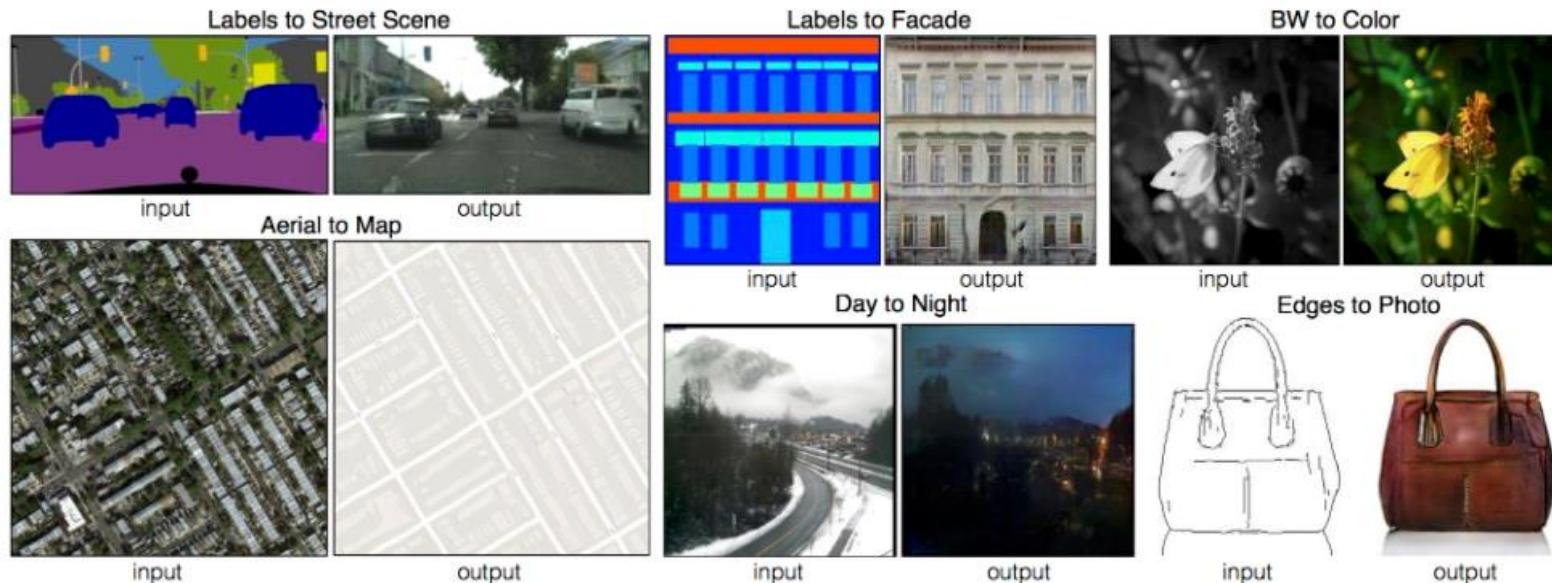


Figure 1: Class-conditional samples generated by our model.

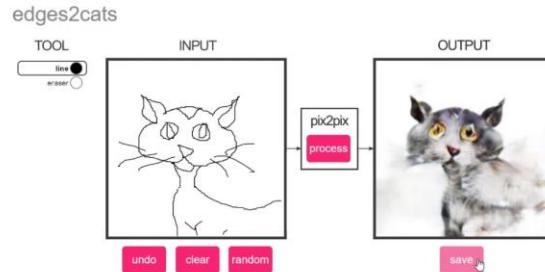


Figure 4: Samples from our BigGAN model with truncation threshold 0.5 (a-c) and an example of class leakage in a partially trained model (d).

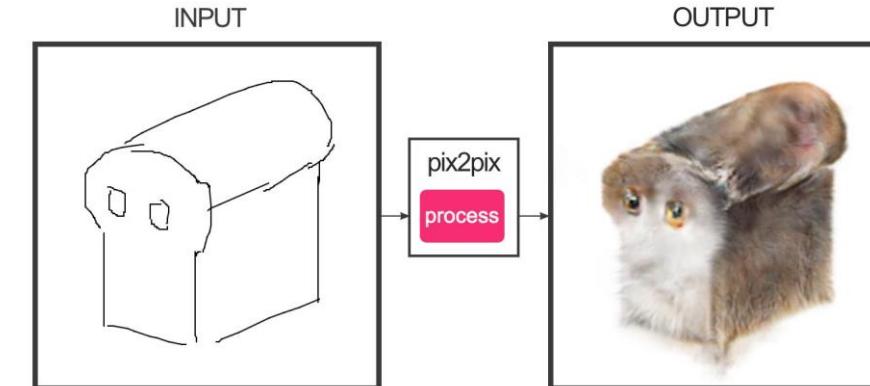
Turning bread into cat...



Example results on several image-to-image translation problems. In each case we use the same architecture and objective, simply training on different data.



Trained on about 2k stock cat photos and edges automatically generated from those photos.
Generates cat-colored objects, some with nightmare faces. The best one I've seen yet was a [cat-beholder](#).



Generative Adversarial Networks

The GAN game

1. get a “True” dataset $\mathcal{D}_T = \{(x_i)\}$
2. get a generator $G_\theta(z)$ random initialization!
3. sample a “False” dataset \mathcal{D}_F : $z \sim p(z), x = G(z)$
4. update $D_\phi(x) = p_\phi(y|x)$ using \mathcal{D}_T and \mathcal{D}_F (1 SGD step)
5. use $D(x)$ to update $G(z)$ (1 SGD step)

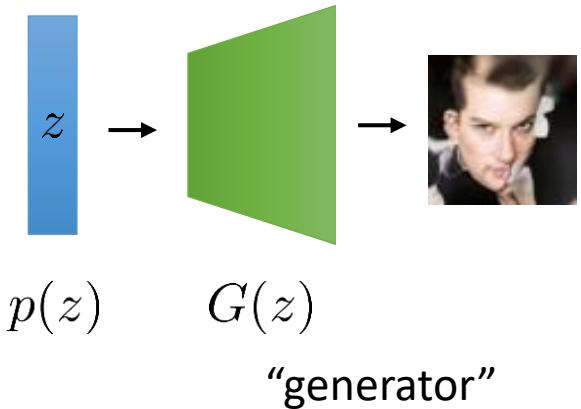
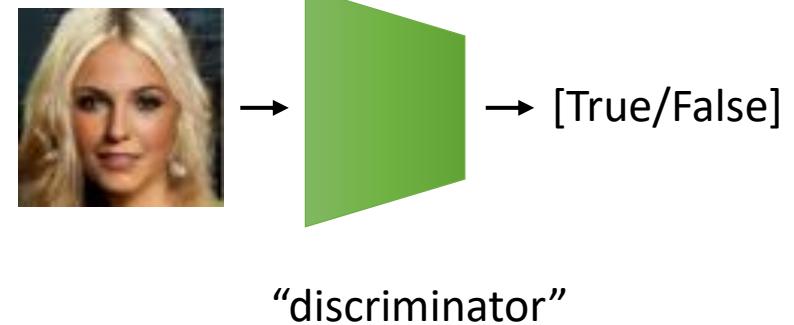
“classic” GAN 2-player game:

$$\min_G \max_D V(D, G) = E_{x \sim p_{\text{data}}(x)}[\log D(x)] + E_{z \sim p(z)}[\log(1 - D(G(z)))]$$

$$\approx \frac{1}{N} \sum_{i=1}^N \log D(x_i) \quad x_i \in \mathcal{D}_T \quad \approx \frac{1}{N} \sum_{j=1}^N \log(1 - D(x_j))$$

$$x_j = G(z_j)$$

random numbers



The GAN game

$$\min_{\theta} \max_{\phi} V(\theta, \phi) = E_{x \sim p_{\text{data}}(x)}[\log D_{\phi}(x)] + E_{z \sim p(z)}[\log(1 - D_{\phi}(G_{\theta}(z)))]$$

$$\begin{aligned} \phi &\leftarrow \phi + \alpha \nabla_{\phi} V(\theta, \phi) \approx \nabla_{\phi} \left(\frac{1}{N} \sum_{i=1}^N \log D_{\phi}(x_i) + \frac{1}{N} \sum_{j=1}^N \log(1 - D_{\phi}(x_j)) \right) \\ \theta &\leftarrow \theta - \alpha \nabla_{\theta} V(\theta, \phi) \\ &\approx \nabla_{\theta} \left(\frac{1}{N} \sum_{j=1}^N \log(1 - D_{\phi}(G_{\theta}(z_j))) \right) \end{aligned}$$

random numbers

↑

this is just cross-entropy loss!

Two important details:

How to make this work with **stochastic** gradient descent/ascent?

How to compute the gradients?

(both are actually pretty simple)

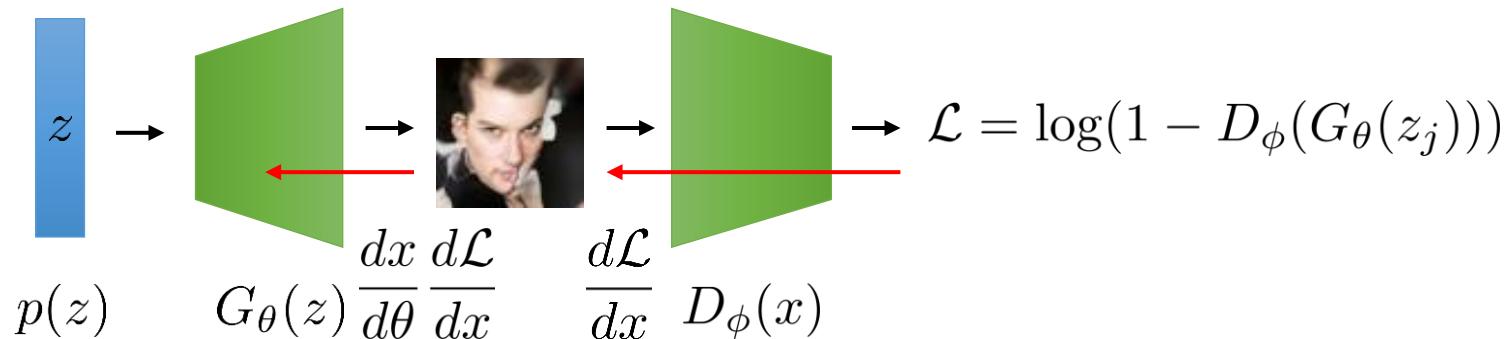
The GAN game

$$\min_{\theta} \max_{\phi} V(\theta, \phi) = E_{x \sim p_{\text{data}}(x)}[\log D_{\phi}(x)] + E_{z \sim p(z)}[\log(1 - D_{\phi}(G_{\theta}(z)))]$$

Green curved arrow:

$$\begin{aligned}\phi &\leftarrow \phi + \alpha \nabla_{\phi} V(\theta, \phi) \\ \theta &\leftarrow \theta - \alpha \nabla_{\theta} V(\theta, \phi)\end{aligned}$$

$$\nabla_{\theta} \left(\frac{1}{N} \sum_{j=1}^N \log(1 - D_{\phi}(G_{\theta}(z_j))) \right)$$



Just backpropagate from the discriminator into the generator!

What does the GAN optimize?

$$\min_G \max_D V(D, G) = E_{x \sim p_{\text{data}}(x)}[\log D(x)] + E_{z \sim p(z)}[\log(1 - D(G(z)))]$$

what can we say about $G(z)$ at convergence?

idea: express $D(x)$ in closed form as function of $G(z)$

$$D_G^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)}$$

\uparrow

$$x = G(z) \quad z \sim p(z)$$

now what is the objective for G ?

$V(D_G^*, G) =$  entirely a function of G

$$E_{p_{\text{data}}(x)}[\log p_{\text{data}}(x) - \log(p_{\text{data}}(x) + p_G(x))] +$$

$$E_{p_G(x)}[\log p_G(x) - \log(p_{\text{data}}(x) + p_G(x))]$$

$$\mathcal{D}_T = \{x_i \sim p(x)\} \quad \mathcal{D}_F = \{x_j \sim q(x)\}$$

$$D^* = \arg \max_D E_p[\log D(x)] + E_q[\log(1 - D(x))]$$

$$\nabla_D = E_p \left[\frac{1}{D(x)} \right] - E_q \left[\frac{1}{1 - D(x)} \right] = 0$$

$$\text{plug in } D^*(x) = \frac{p(x)}{p(x) + q(x)}$$

optimal
discriminator

$$\sum_x p(x) \frac{p(x) + q(x)}{p(x)} - \sum_x q(x) \frac{p(x) + q(x)}{q(x)} = 0$$

What does the GAN optimize?

$$V(D_G^*, G) =$$

$$E_{p_{\text{data}}(x)}[\log p_{\text{data}}(x) - \log(p_{\text{data}}(x) + p_G(x))] +$$

what funny expressions...

$$E_{p_G(x)}[\log p_G(x) - \log(p_{\text{data}}(x) + p_G(x))]$$

$$\text{let } q(x) = \frac{p_{\text{data}}(x) + p_G(x)}{2}$$

accounts for the $\frac{1}{2}$ factors

$$V(D_G^*, G) = E_{p_{\text{data}}(x)}[\log p_{\text{data}}(x) - \log q(x)] + E_{p_G(x)}[\log p_G(x) - \log q(x)] - \log 4$$

$$\underbrace{\phantom{E_{p_{\text{data}}(x)}[\log p_{\text{data}}(x) - \log q(x)] + E_{p_G(x)}[\log p_G(x) - \log q(x)]}}_{D_{\text{KL}}(p_{\text{data}} \| q(x))} \quad \underbrace{\phantom{E_{p_{\text{data}}(x)}[\log p_{\text{data}}(x) - \log q(x)] + E_{p_G(x)}[\log p_G(x) - \log q(x)]}}_{D_{\text{KL}}(p_G \| q(x))}$$

$$= D_{\text{JS}}(p_{\text{data}} \| p_G)$$

Jensen-Shannon divergence

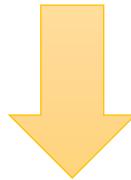
This has some interesting properties:

goes to zero if the distributions match
symmetric (unlike KL-divergence)

This means the GAN really is trying
to match the data distribution!

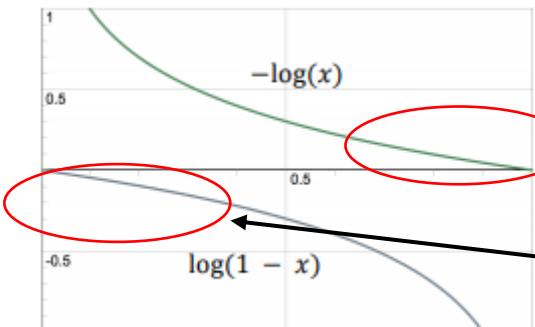
A small practical aside

$$\min_{\theta} \max_{\phi} V(\theta, \phi) = E_{x \sim p_{\text{data}}(x)}[\log D_{\phi}(x)] + E_{z \sim p(z)}[\log(1 - D_{\phi}(G_{\theta}(z)))]$$



generator loss should be $E_{z \sim p(z)}[\log(1 - D_{\phi}(G_{\theta}(z)))]$ “minimize probability that image is fake”

in practice, we often use $E_{z \sim p(z)}[-\log D_{\phi}(G_{\theta}(z))]$ “maximize probability that image is real”
(though there are other variants too!)

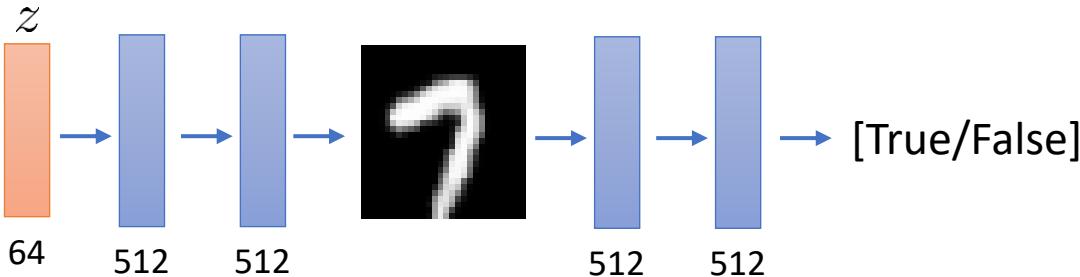


small gradient when generator is good
(i.e., probability of real is high)

small gradient when generator is bad
(i.e., probability of fake is high)

GAN architectures

some made-up architectures



a real architecture (BigGAN)

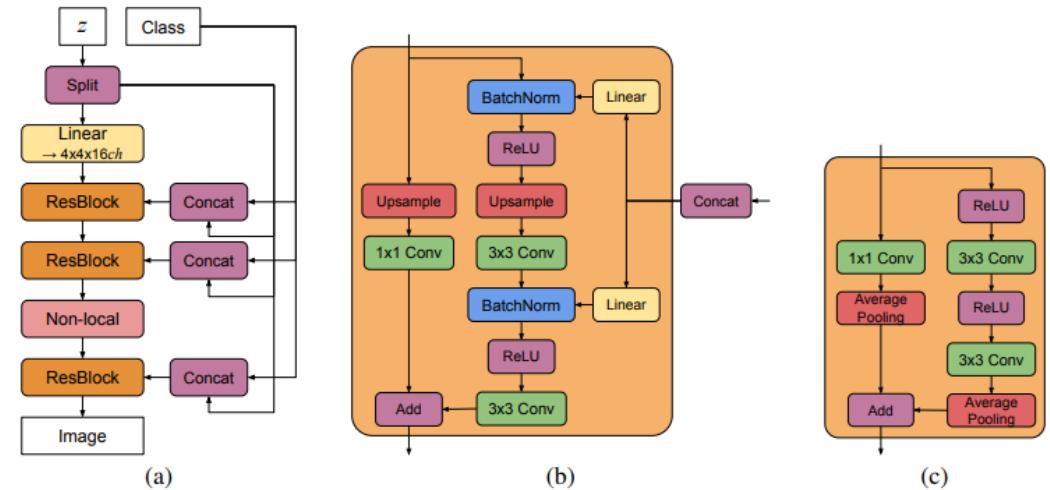
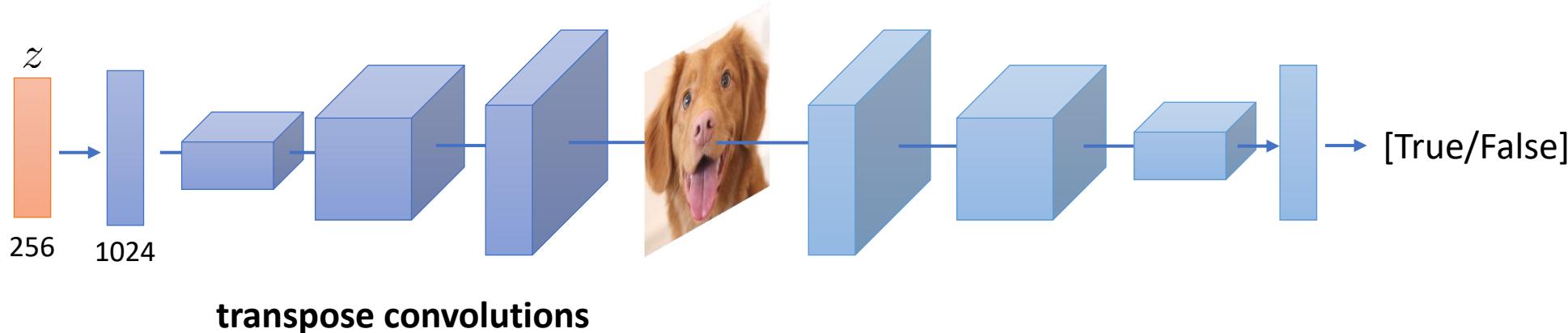
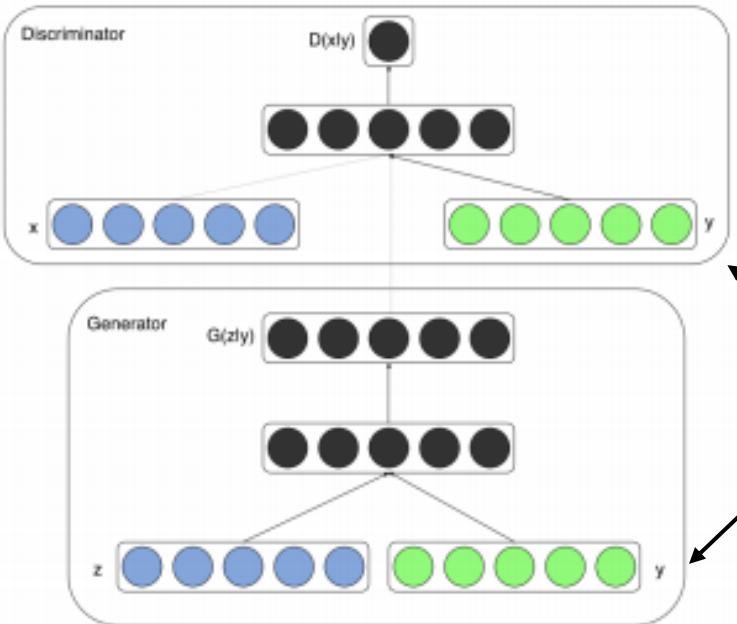


Figure 15: (a) A typical architectural layout for BigGAN's \mathbf{G} ; details are in the following tables. (b) A Residual Block (*ResBlock up*) in BigGAN's \mathbf{G} . (c) A Residual Block (*ResBlock down*) in BigGAN's \mathbf{D} .



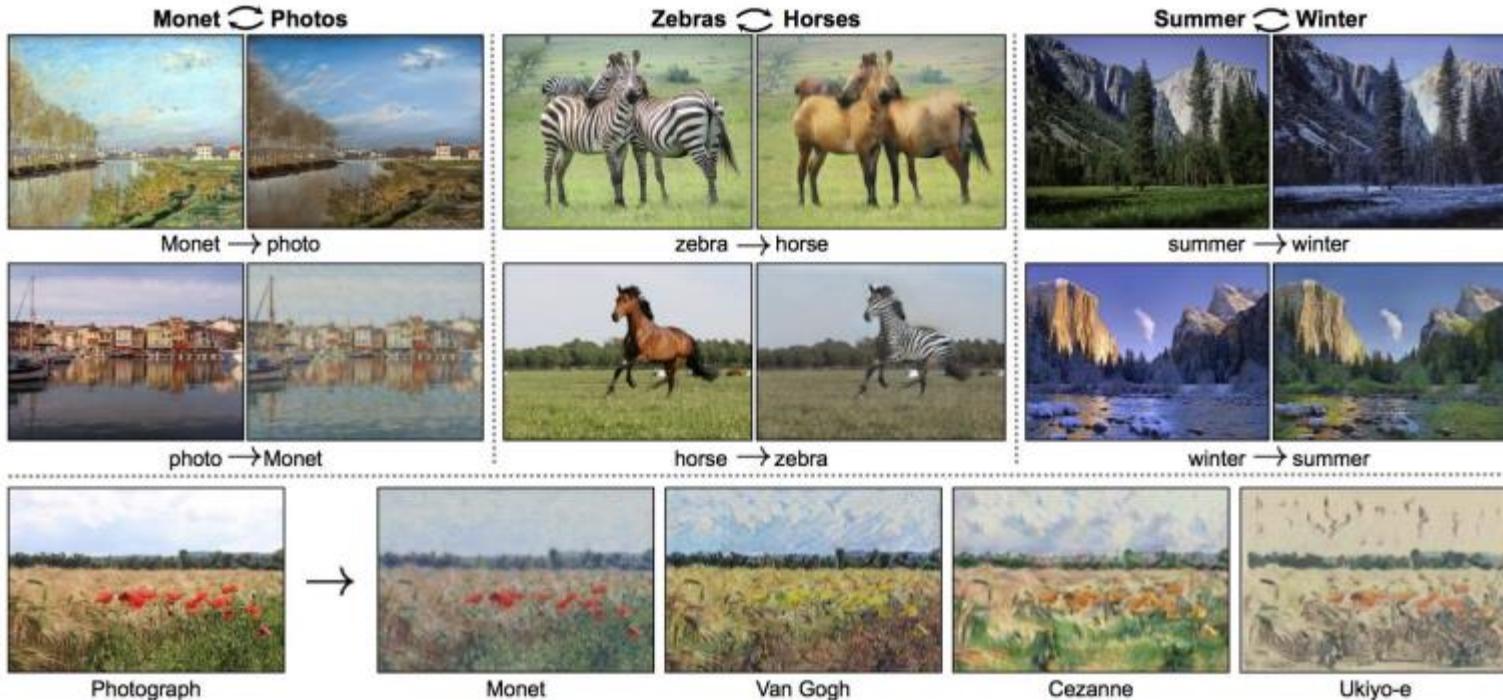
Conditional GANs



append conditioning (e.g., class label)
to **both** generator and discriminator



CycleGAN



Problem: we don't know which images "go together"

CycleGAN

two (conditional) generators:

G : turn X into Y (e.g., horse into zebra)

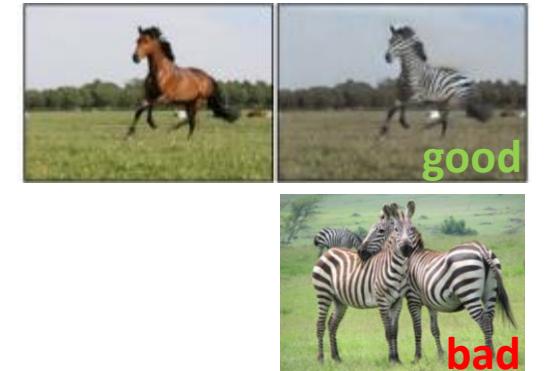
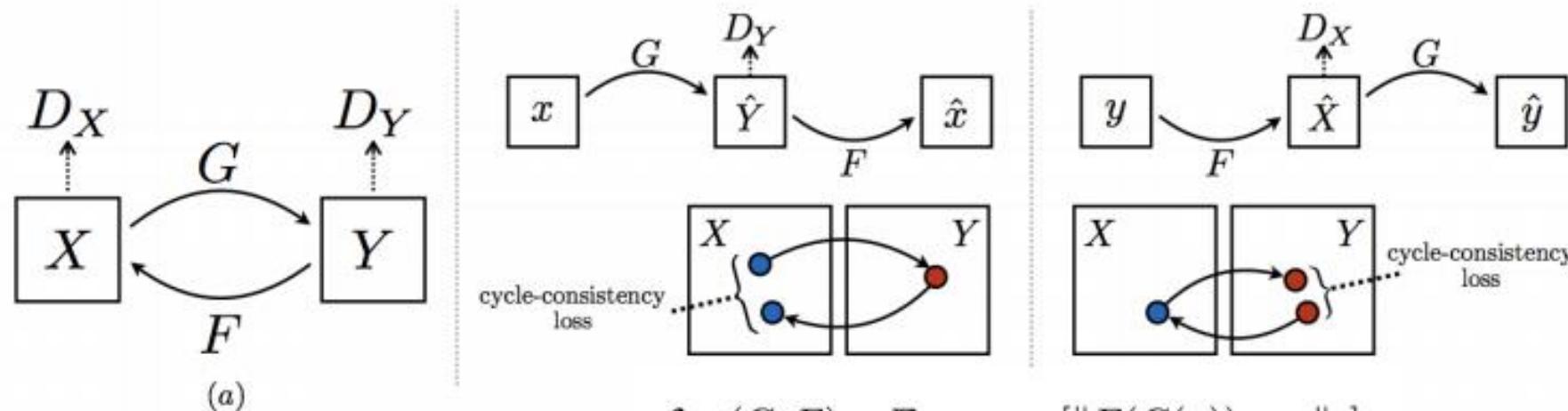
F : turn Y into X (e.g., zebra into horse)

two discriminators:

D_X : is it a realistic horse?

D_Y : is it a realistic zebra?

Problem: why should the “translated” zebra look anything like the original horse?



If I turn this horse into a zebra, and
then turn that zebra back into a
horse, I should get the same horse!

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



Summary

- The GAN is a 2-player game $\min_G \max_D V(D, G) = E_{x \sim p_{\text{data}}(x)}[\log D(x)] + E_{z \sim p(z)}[\log(1 - D(G(z)))]$
- We can derive the **optimal** discriminator, and from this determine what objective is minimized at the Nash equilibrium
 - Note that this does **not** guarantee that we'll actually find the Nash equilibrium!
- We can train either fully connected **or** convolutional GANs
- We can turn horses into zebras

$$D_G^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)}$$

$$V(D_G^*, G) = D_{\text{JS}}(p_{\text{data}} \| p_G)$$

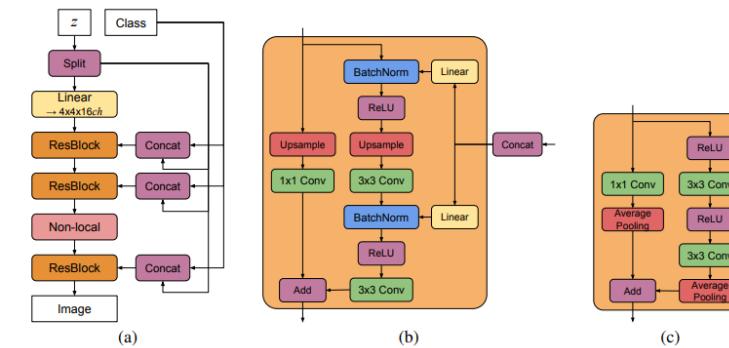
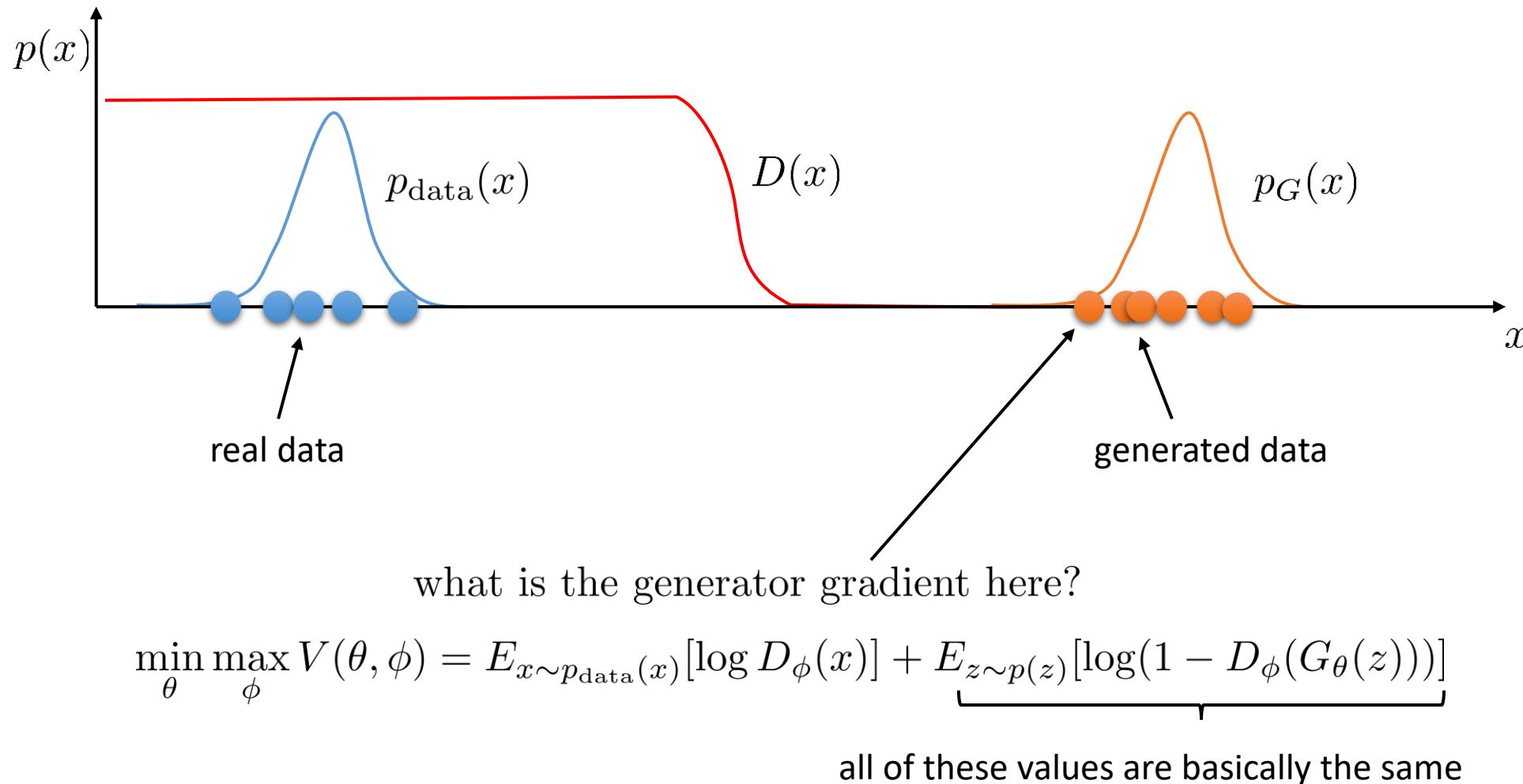


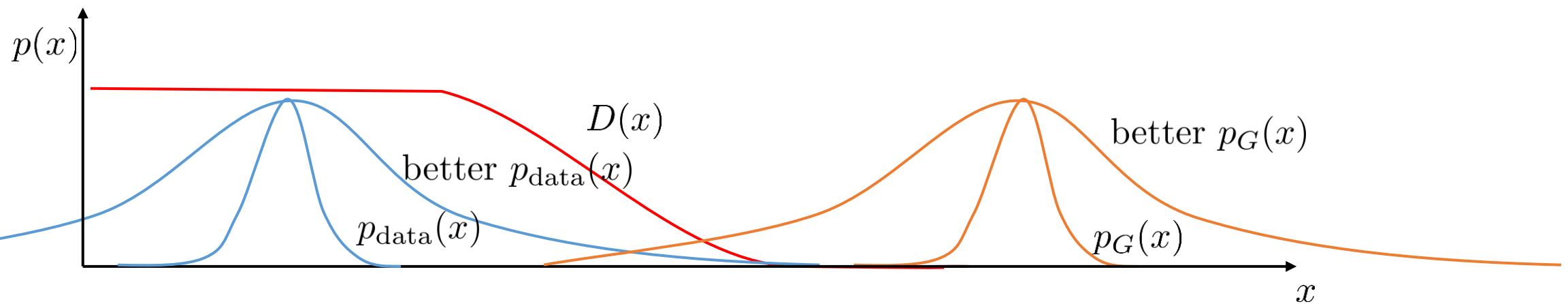
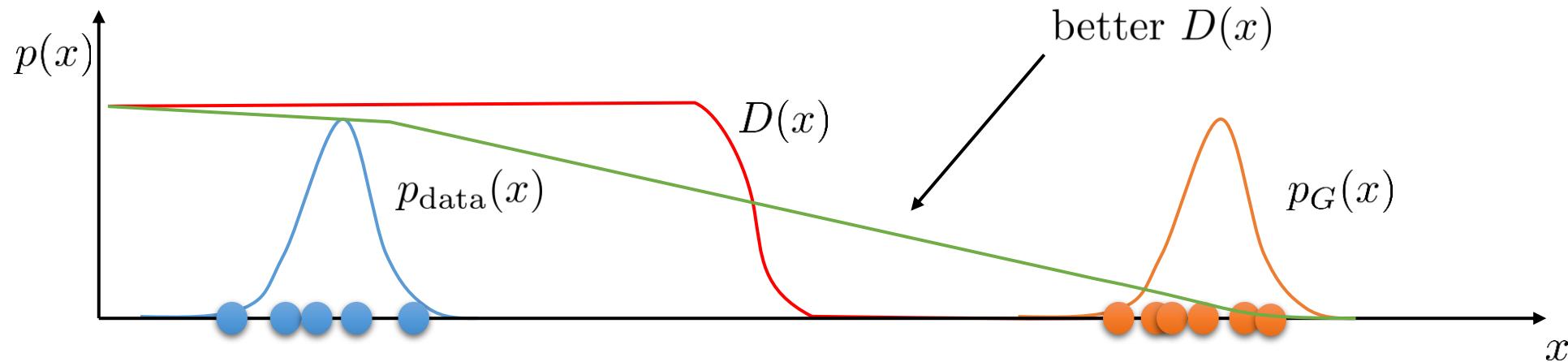
Figure 15: (a) A typical architectural layout for BigGAN's G ; details are in the following tables. (b) A Residual Block ($ResBlock\ up$) in BigGAN's G . (c) A Residual Block ($ResBlock\ down$) in BigGAN's D .

Improved GAN Training

Why is training GANs hard?



How can we make it better?



Improved GAN techniques

(in no particular order)

Least-squares GAN (LSGAN)

discriminator outputs real-valued number

Wasserstein GAN (WGAN)

discriminator is constrained to be Lipschitz-continuous

Gradient penalty

discriminator is constrained to be continuous even harder

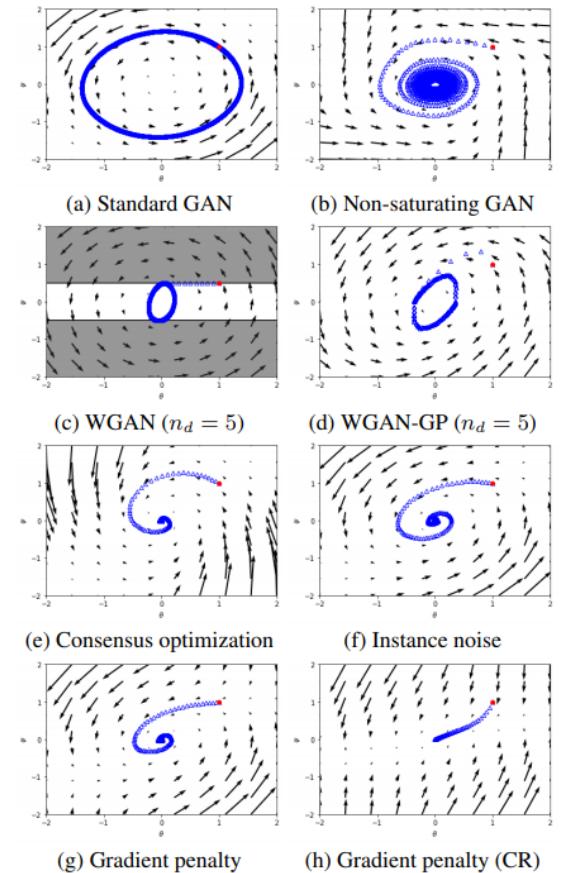
Spectral norm

discriminator is **really** constrained to be continuous

Instance noise

add noise to the data and generated samples

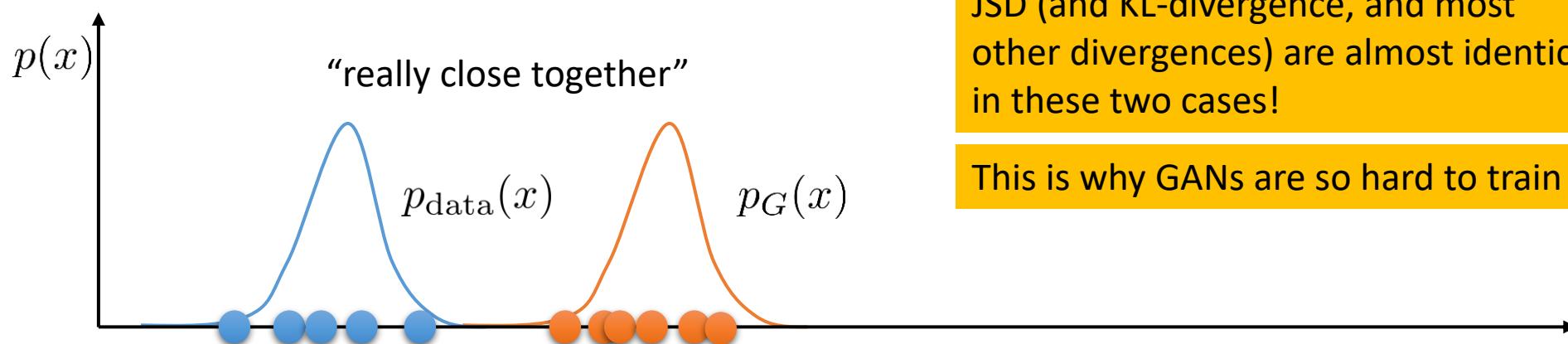
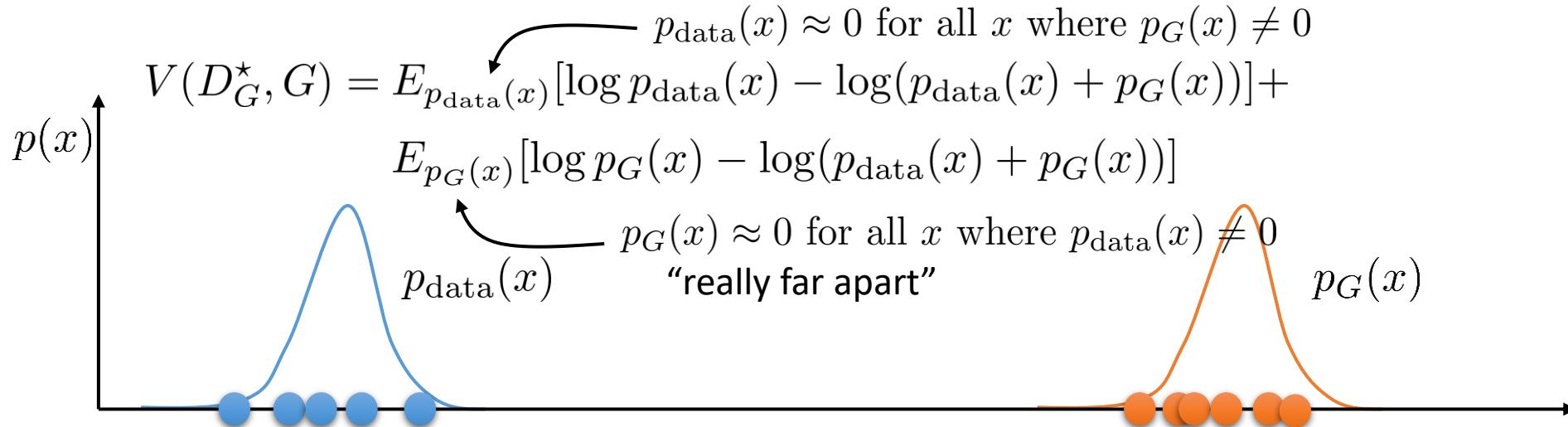
these are pretty good choices today



Mescheder et al. **Which Training Methods for GANs do actually Converge?** 2108.

Wasserstein GAN (WGAN)

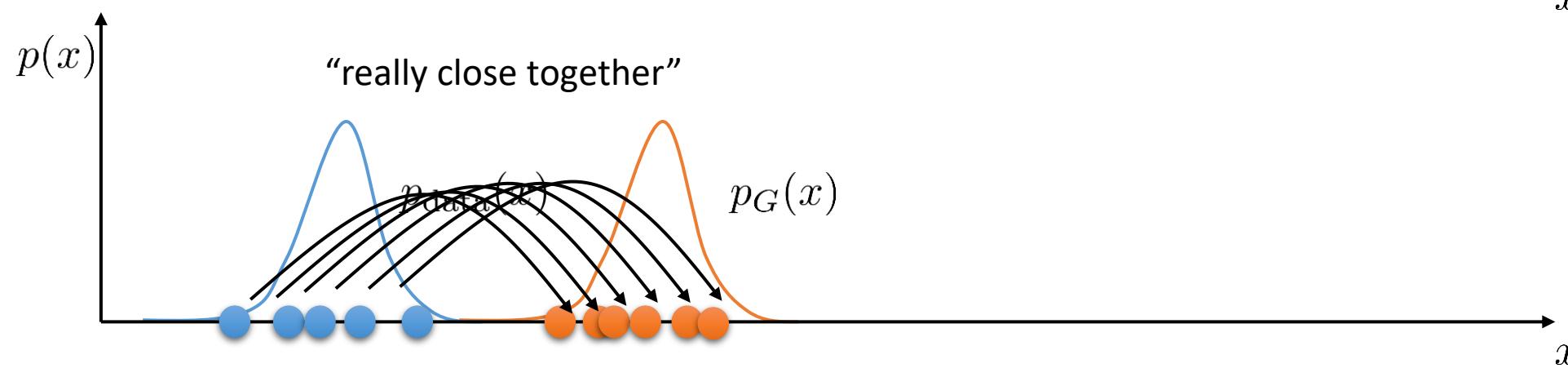
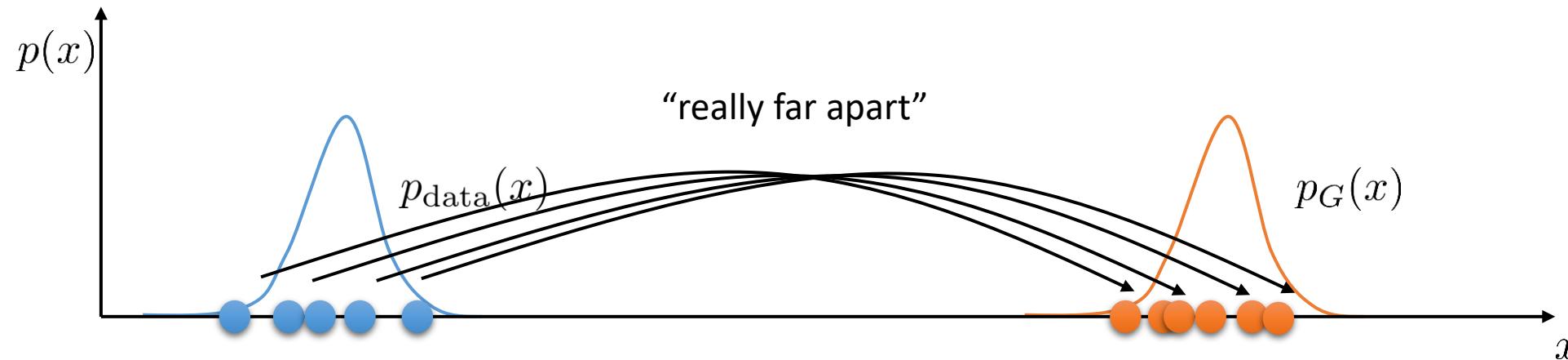
High-level intuition: the JS divergence used by the classic GAN doesn't account for "distance"



Wasserstein GAN (WGAN)

A better metric: consider how far apart (in Euclidean space) all the “bits” of probability are

More precisely: optimal transport (“Earth mover’s distance”) – how far do you have to go to “transport” one distribution into another



Wasserstein GAN (WGAN)

A better metric: consider how far apart (in Euclidean space) all the “bits” of probability are

More precisely: optimal transport (“Earth mover’s distance”) – how far do you have to go to “transport” one distribution into another

Formal definition: (don’t worry too much if this is hard to understand, not actually necessary to *implement* WGAN)

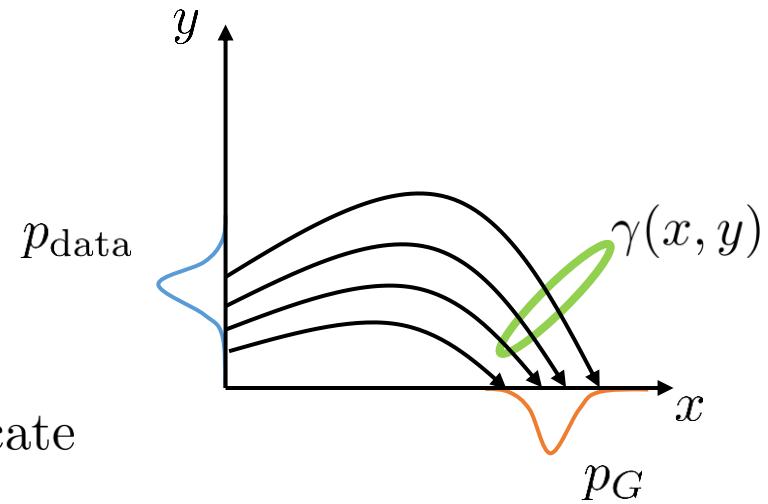
$$W(p_{\text{data}}, p_G) = \inf_{\gamma} E_{(x,y) \sim \gamma(x,y)} [||x - y||]$$



$\gamma(x, y)$ is a *distribution* over x, y

with marginals $\gamma_X(x) = p_{\text{data}}(x)$ and $\gamma_Y(y) = p_G(y)$

intuition: correlations between x and y in $\gamma(x, y)$ indicate which x should be “transported” to which y



Wasserstein GAN (WGAN)

A better metric: consider how far apart (in Euclidean space) all the “bits” of probability are

More precisely: optimal transport (“Earth mover’s distance”) – how far do you have to go to “transport” one distribution into another

$$W(p_{\text{data}}, p_G) = \inf_{\gamma} E_{(x,y) \sim \gamma(x,y)} [\|x - y\|]$$

could learn γ directly
but this is very hard

$p_{\text{data}}(x)$ is unknown
 $\gamma(x, y)$ is really complex

cool theorem based on Kantorovich-Rubinstein duality:

$$W(p_{\text{data}}, p_G) = \sup_{\|f\|_L \leq 1} E_{p_{\text{data}}} [f(x)] - E_{p_G(x)} [f(x)]$$



set of all 1-Lipschitz scalar functions

$$|f(x) - f(y)| \leq |x - y|$$

equivalent to saying function has bounded slope
i.e., it should not be too steep

expressed as difference of expectations
under $p_G(x)$ and $p_{\text{data}}(x)$
just like a regular GAN!

How?

Wasserstein GAN (WGAN)

A better metric: consider how far apart (in Euclidean space) all the “bits” of probability are

More precisely: optimal transport (“Earth mover’s distance”) – how far do you have to go to “transport” one distribution into another

$$W(p_{\text{data}}, p_G) = \sup_{\|f\|_L \leq 1} E_{p_{\text{data}}}[f(x)] - E_{p_G(x)}[f(x)]$$



doesn’t guarantee 1-Lipschitz
unless we pick bounds very carefully

does guarantee K-Lipschitz
for some finite K

set of all 1-Lipschitz scalar functions

$$|f(x) - f(y)| \leq |x - y|$$

idea: if f is a neural net with ReLU activations, can bound the weights W

1-layer: $f_\theta(x) = \text{ReLU}(W_1x + b_1) \quad \theta = \{W_1, b_1\}$

if $W_{1,i,j} \in [-0.01, 0.01]$, slope can’t be bigger than $0.01 \times D$

2-layer: $f_\theta(x) = W_2 \text{ReLU}(W_1x + b_1) + b_2 \quad \theta = \{W_1, b_1, W_2, b_2\}$

if $W_{\ell,i,j} \in [-0.01, 0.01]$, slope is bounded (but greater than 0.01!)

Wasserstein GAN (WGAN)

A better metric: consider how far apart (in Euclidean space) all the “bits” of probability are

More precisely: optimal transport (“Earth mover’s distance”) – how far do you have to go to “transport” one distribution into another

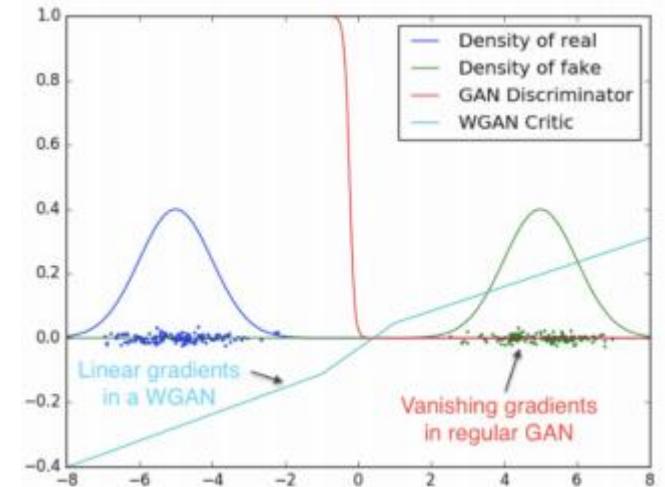
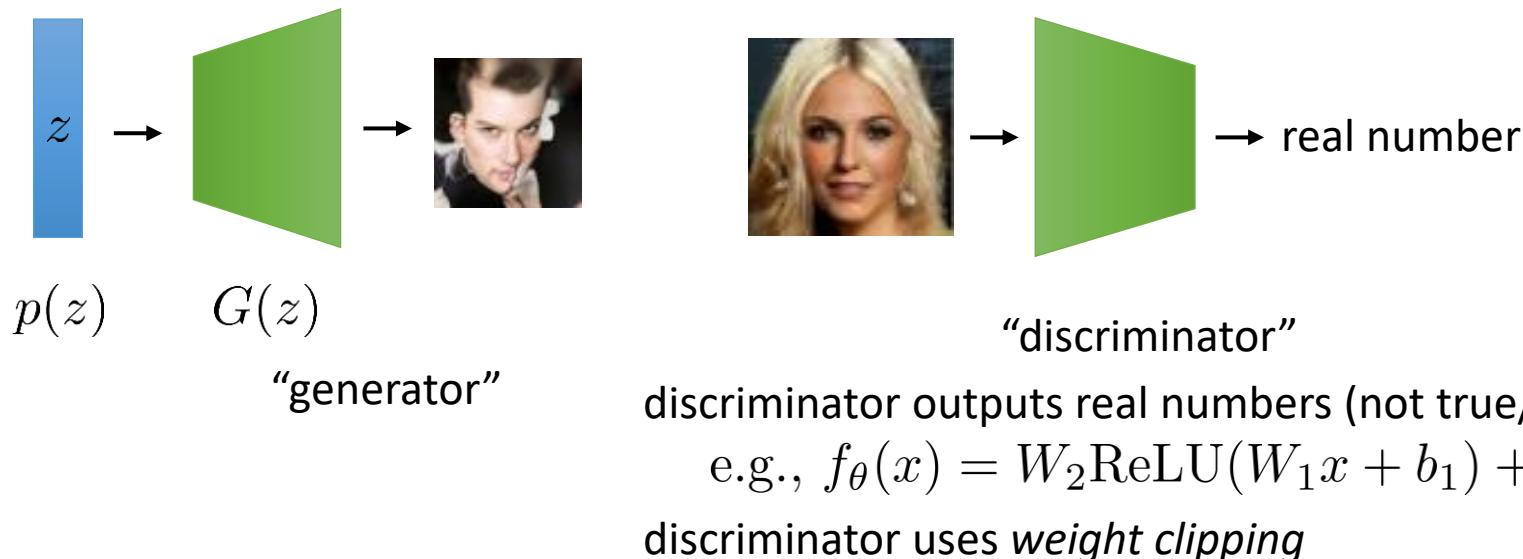
$$W(p_{\text{data}}, p_G) = \sup_{\|f\|_L \leq 1} E_{p_{\text{data}}}[f(x)] - E_{p_G(x)}[f(x)]$$

Weight clipping is a clearly terrible way to enforce a Lipschitz constraint. If the clipping parameter is large, then it can take a long time for any weights to reach their limit, thereby making it harder to train the critic till optimality. If the clipping is small, this can easily lead to vanishing gradients when the number of layers is big, or batch normalization is not used (such as in RNNs). We experimented with simple variants (such as projecting the weights to a sphere) with little difference, and we stuck with weight clipping due to its simplicity and already good performance. However, we do leave the topic of enforcing Lipschitz constraints in a neural network setting for further investigation, and we actively encourage interested researchers to improve on this method.

Wasserstein GAN (WGAN)

$$W(p_{\text{data}}, p_G) = \sup_{\|f\|_L \leq 1} E_{p_{\text{data}}}[f(x)] - E_{p_G(x)}[f(x)]$$

1. update f_θ using gradient of $E_{x \sim p_{\text{data}}}[f_\theta(x)] - E_{z \sim p(z)}[f_\theta(G(z))]$
2. clip all weight matrices in θ to $[-c, c]$
3. update generator to *maximize* $E_{z \sim p(z)}[f_\theta(G(z))]$



Better discriminator regularization

Weight clipping is a clearly terrible way to enforce a Lipschitz constraint.

Gradient penalty: Want bounded slope? We'll give you bounded slope!

1-Lipschitz:
 $|f(x) - f(y)| \leq |x - y|$

update f_θ using gradient of

$$E_{x \sim p_{\text{data}}}[f_\theta(x) - \underbrace{\lambda(||\nabla_x f_\theta(x)||_2 - 1)^2}_{\text{make norm of gradient close to 1}}] - E_{z \sim p(z)}[f_\theta(G(z))]$$

Spectral norm

Idea: bound the Lipschitz constant in terms of singular values of each W_ℓ

$$f(x) = f_3 \circ f_2 \circ f_1(x)$$

neural net layers (e.g., linear, conv, ReLU, etc.)



$$\|f(x)\|_{\text{Lip}} = \|f_3 \circ f_2 \circ f_1\|_{\text{Lip}} \leq \|f_3\|_{\text{Lip}} \cdot \|f_2\|_{\text{Lip}} \cdot \|f_1\|_{\text{Lip}}$$

Lipschitz constant

to get intuition for why this is true, imagine these are **linear** functions

$\text{ReLU}(x) = \max(0, x) \Rightarrow$ max slope is 1! that's easy, how about linear layers?

max slope of $Wx + b$ is **spectral norm**:

$$\sigma(W) = \max_{h:h \neq 0} \frac{\|Wh\|}{\|h\|} = \max_{\|h\| \leq 1} \|Wh\|$$

largest singular value of W

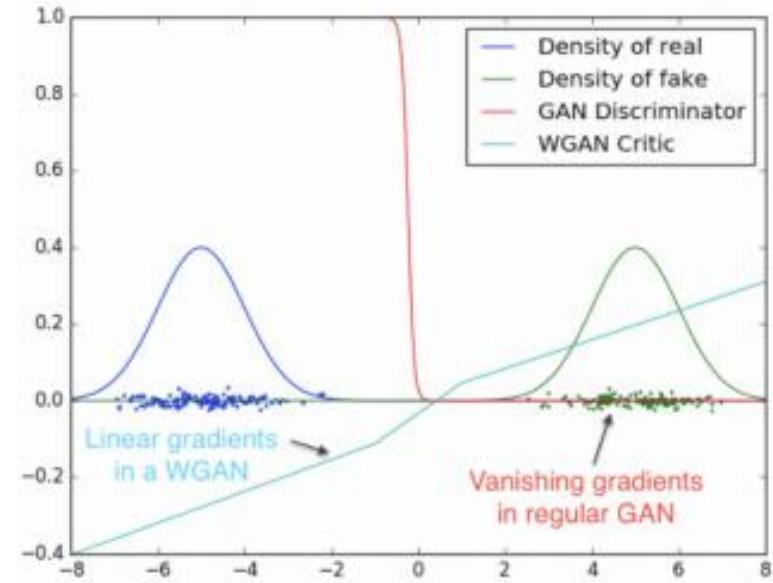


Method: after each grad step, force $W_\ell \leftarrow \frac{W_\ell}{\sigma(W_\ell)}$

See paper for how to
implement this efficiently

GAN training summary

- GAN training is really hard, because the discriminator can provide poor gradients
- Various “tricks” can make this much more practical
 - “Smooth” real-valued discriminators: LSGAN, WGAN, WGAN-GP, spectral norm
 - Instance noise
- The theory behind these tricks is quite complex, but the methods in practice are very simple
- Such GANs are **much** easier to train



Adversarial Examples

Designing, Visualizing and Understanding Deep Neural Networks

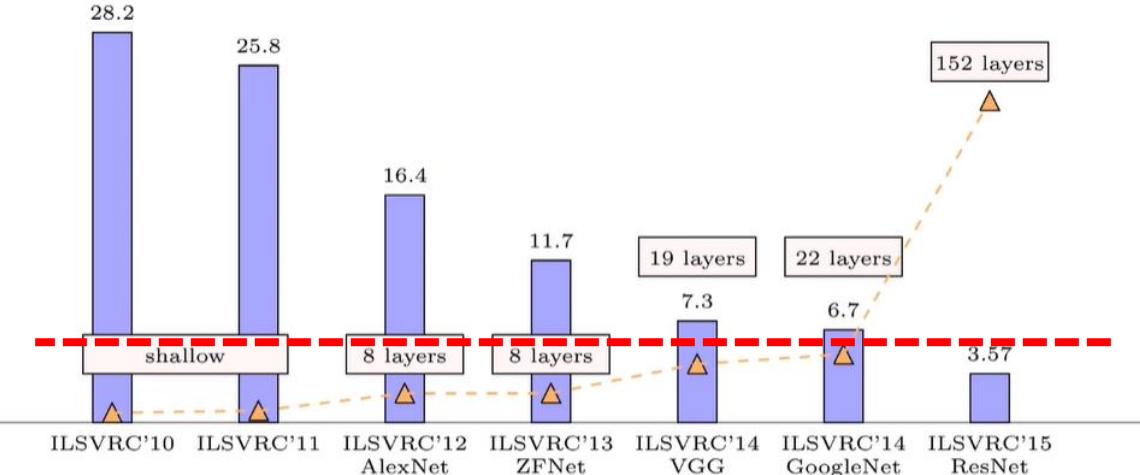
CS W182/282A

Instructor: Sergey Levine
UC Berkeley



Do deep nets generalize?

What a strange question!



human performance:
about 5% error

but what about the **mistakes**? What kinds of mistakes are they?

Do deep nets generalize?

Even the mistakes make sense (sometimes)!

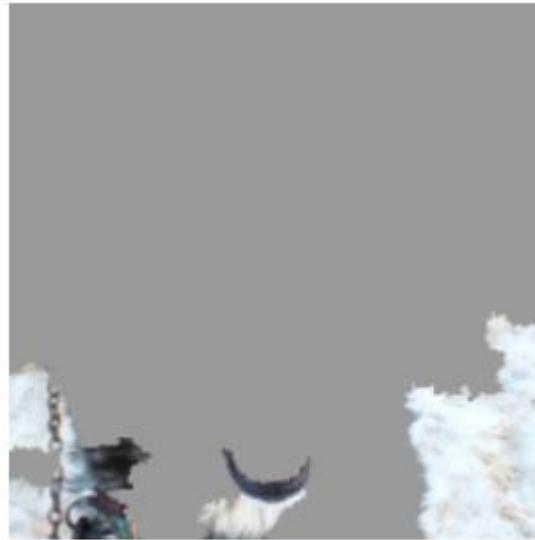


Mosque

Do deep nets generalize?



(a) Husky classified as wolf



(b) Explanation

Figure 11: Raw data and explanation of a bad model's prediction in the “Husky vs Wolf” task.

A metaphor for machine learning...



Clever Hans

or: when the training/test paradigm goes wrong

Everything might be “working as intended”,
but we might still not get what we want!

Distribution shift

One source of trouble: the test inputs might come from a different distribution than training inputs
often especially problematic if the training data has spurious correlations



traffic sign classification dataset



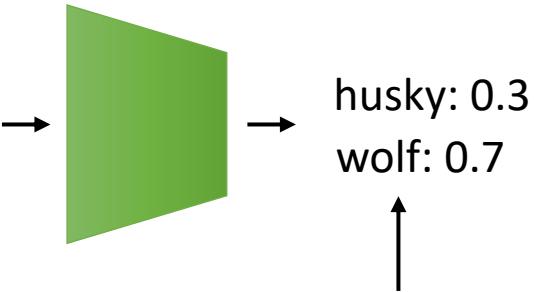
traffic sign in reality

Some more realistic examples:

- Medical imaging: different hospitals have different machines
- Even worse, different hospitals have different positive rates (e.g., some hospitals get more sick patients)
- Induces machine \Leftrightarrow label correlation
- Selection biases: center crop, canonical pose, etc.
- Feedback: the use of the ML system causes users to change their behavior, thus changing the input distribution
 - Classic example: spam classification

Calibration

Definition: the predicted probabilities reflect the actual frequencies of the predicted events

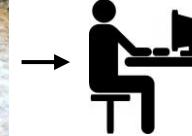
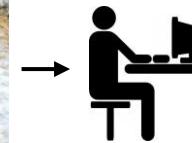


“7 times out of 10 times, a person would say this is a wolf”

how is the data generated?



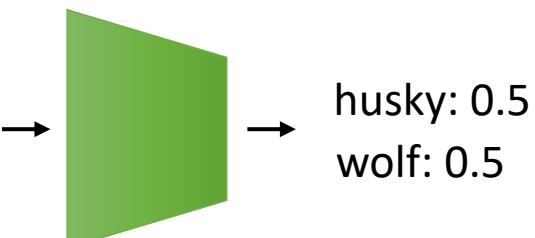
$$x \sim p(x)$$



“husky”

$$y \sim p(y|x)$$

out of distribution:



“I don’t know what this is”

Does this happen?

Usually not, such models typically give **confident** but **wrong** predictions on OOD inputs (but not always!)

why?

Are **in-distribution** predictions calibrated?

Usually not, but there are many methods for improving calibration

Adversarial examples

Adversarial examples

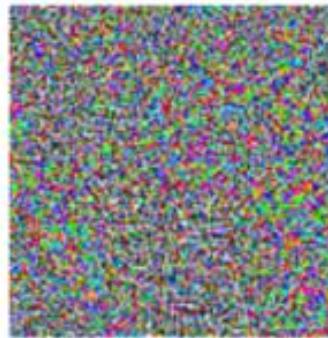
A particularly vivid illustration of how learned models may or may not generalize correctly

this is **not** random noise –
special pattern design to “fool”
the model



“panda”
57.7% confidence

+ .007 ×



“nematode”
8.2% confidence

=



“gibbon”
99.3 % confidence

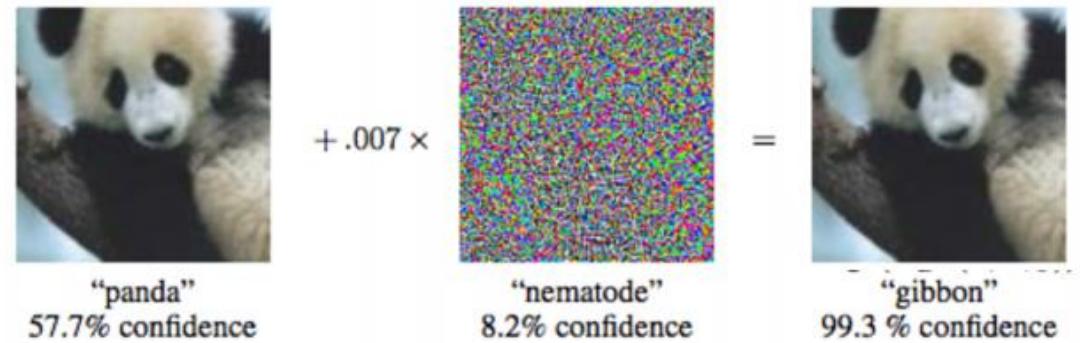
What's going on here? very special patterns, almost imperceptible to people, can change a model's classification drastically

Why do we care? **The direct issue:** this is a potential way to “attack” learned classifiers
The bigger issue: this implies some strange things about generalization

Some facts about adversarial examples

We'll discuss many of these facts in detail, but let's get the full picture first:

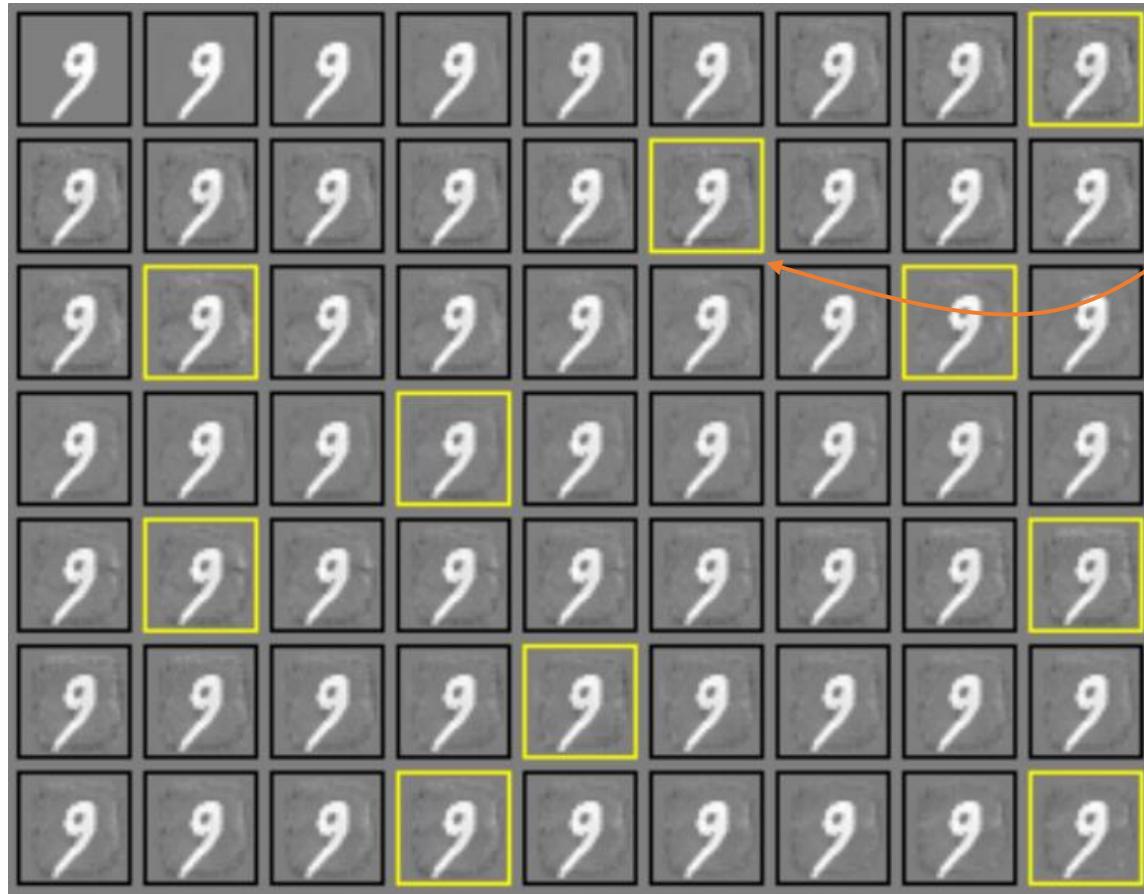
- It's not just for gibbons. Can turn basically **anything** into **anything else** with enough effort
- It is **not** easy to defend against, obvious fixes can help, but nothing provides a bulletproof defense (that we know of)
- Adversarial examples can transfer across different networks (e.g., the same adversarial example can fool both AlexNet and ResNet)
- Adversarial examples can work in the real world, not just special and very precise pixel patterns
- Adversarial examples are **not** specific to (artificial) neural networks, virtually all learned models are susceptible to them



← speed limit: 45
photo is not altered in any way!
but the sign is

including your brain,
which is a type of
learned model

A problem with deep nets?



linear model (logistic regression)

adversarial examples appear to be a general phenomenon for most learned models (and all high-capacity models that we know of)

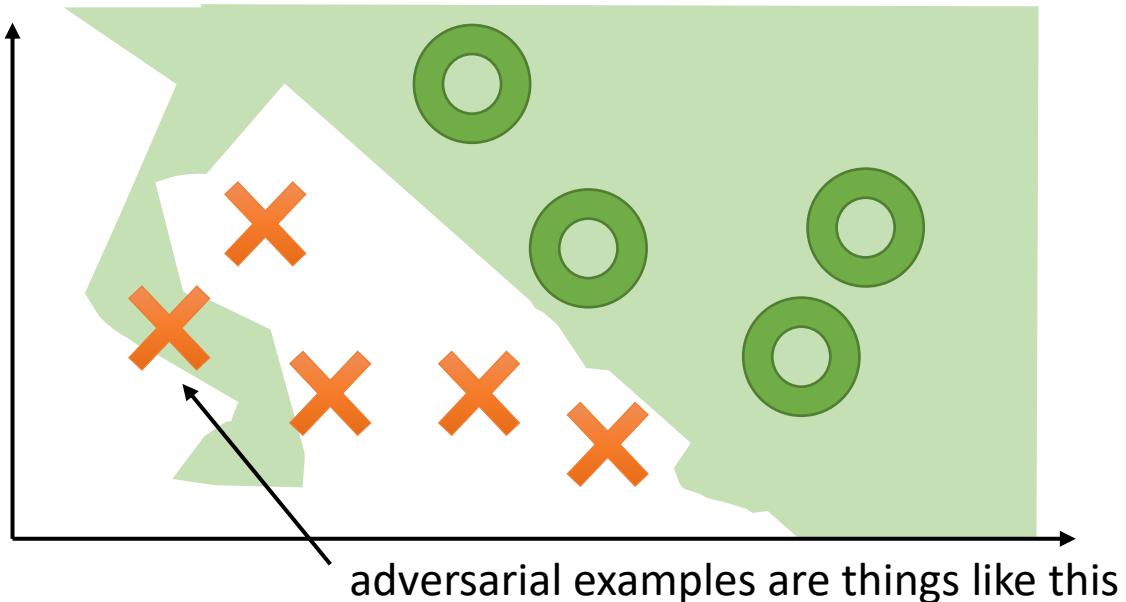
Is it due to overfitting?

Overfitting hypothesis: because neural nets have a huge number of parameters, they tend to overfit, making it easy to find inputs that produce crazy outputs

Implication: to fix adversarial examples, stop using neural nets

most evidence suggests that this hypothesis is false

The mental model:

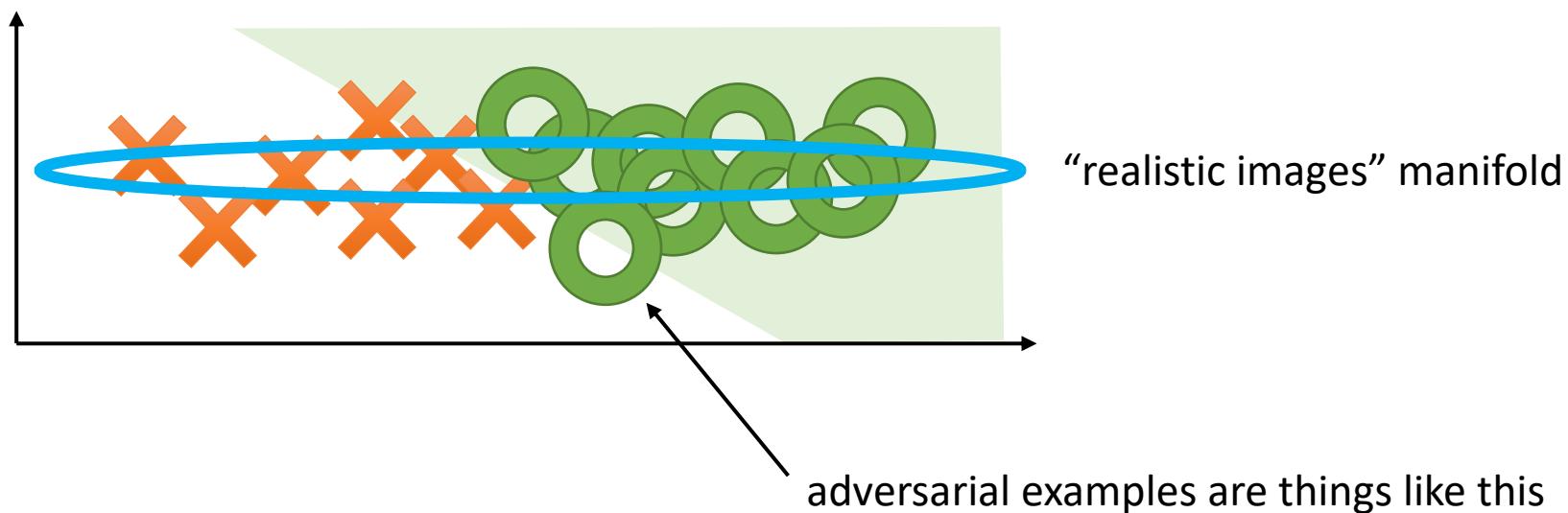


- If this were true, we would expect different models to have very different adversarial examples (high variance)
 - This is conclusively not the case
- If this were true, we would expect low capacity models (e.g., linear models) not to have this issue
 - Low capacity models also have this
- If this were true, we would expect highly nonlinear decision boundaries around adversarial examples
 - This appears to not be true

Linear models hypothesis

Linear models hypothesis: because neural networks (and many other models!) tend to be locally linear, they extrapolate in somewhat counterintuitive ways when moving away from the data

this has a somewhat counterintuitive meaning in high dimensions

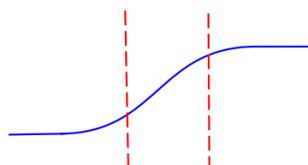


why so linear?

Rectified linear unit



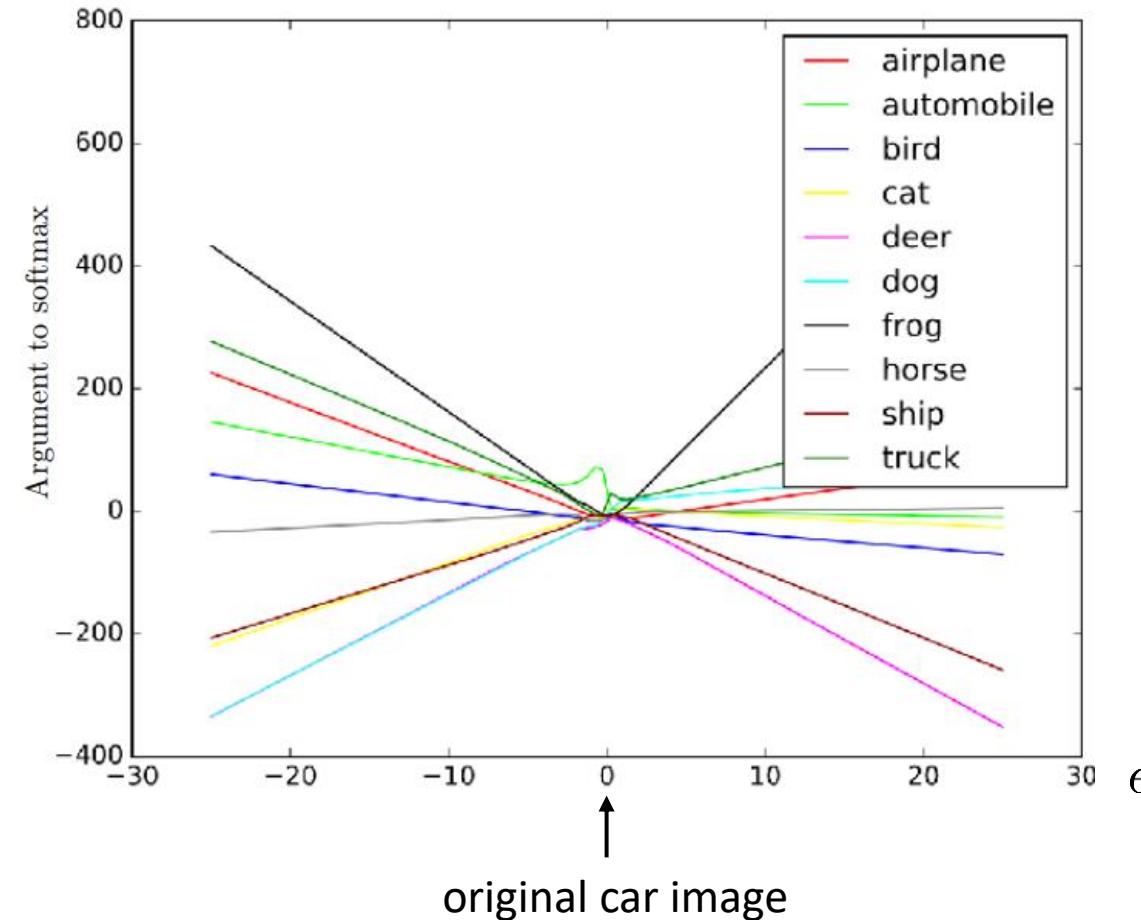
Carefully tuned sigmoid



- Consistent with transferability of adversarial examples
- Reducing “overfitting” doesn’t fix the problem

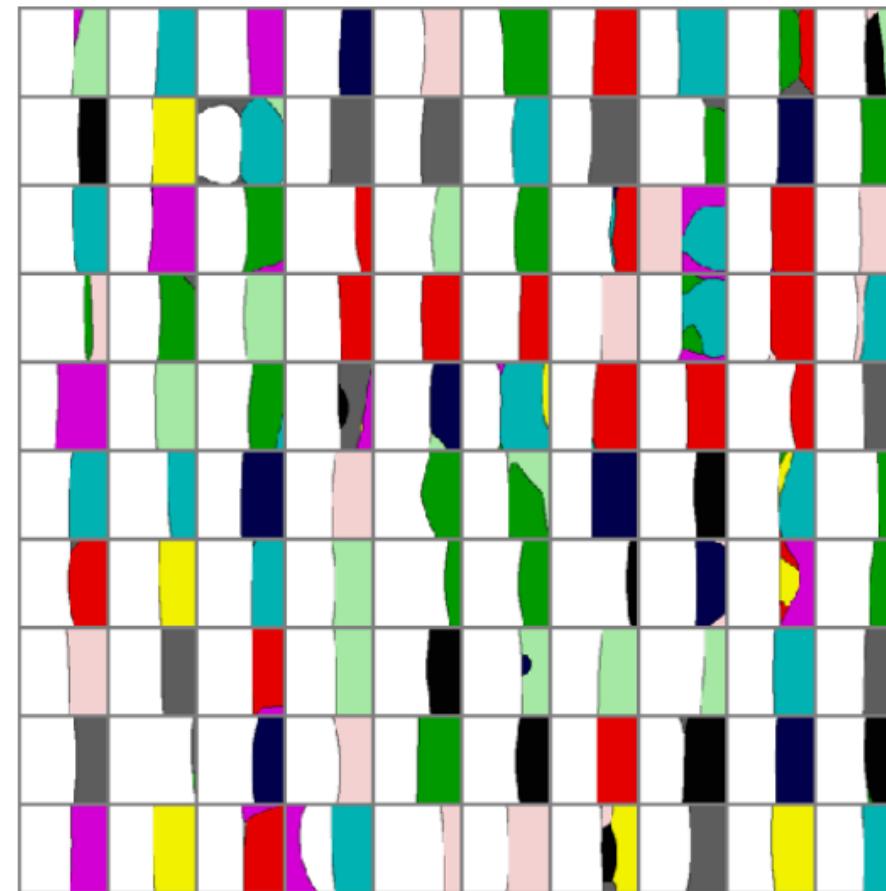
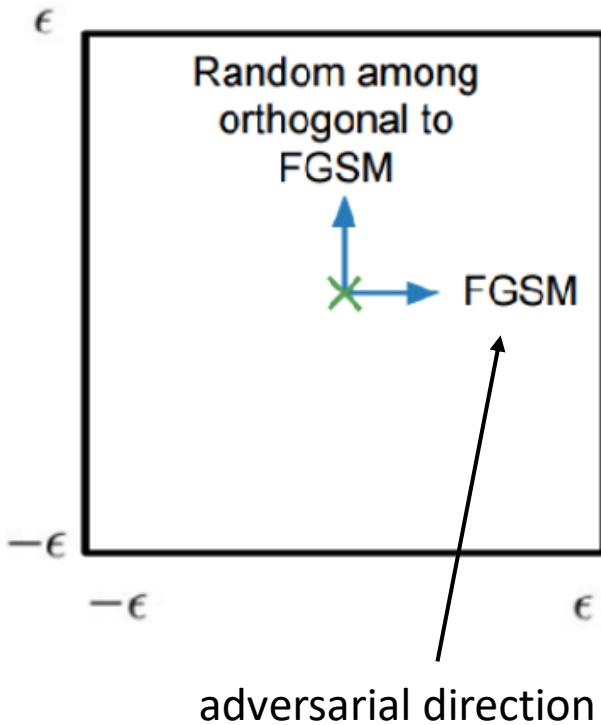
Linear models hypothesis

Experiment 1: vary images along one vector, and see how predictions change



Linear models hypothesis

Experiment 2: vary images along two directions: an adversarial one, and a random one



not much variation **orthogonal** to adversarial direction

clean “shift” on **one side** for adversarial direction, suggesting a mostly linear decision boundary

Real-world adversarial examples

Distance/Angle	Subtle Poster	Subtle Poster Right Turn	Camouflage Graffiti	Camouflage Art (LISA-CNN)	Camouflage Art (GTSRB-CNN)
5' 0°					
5' 15°					
10' 0°					
10' 30°					
40' 0°					
Targeted-Attack Success	100%	73.33%	66.67%	100%	80%

all of these turn into 45 mph speed limit signs

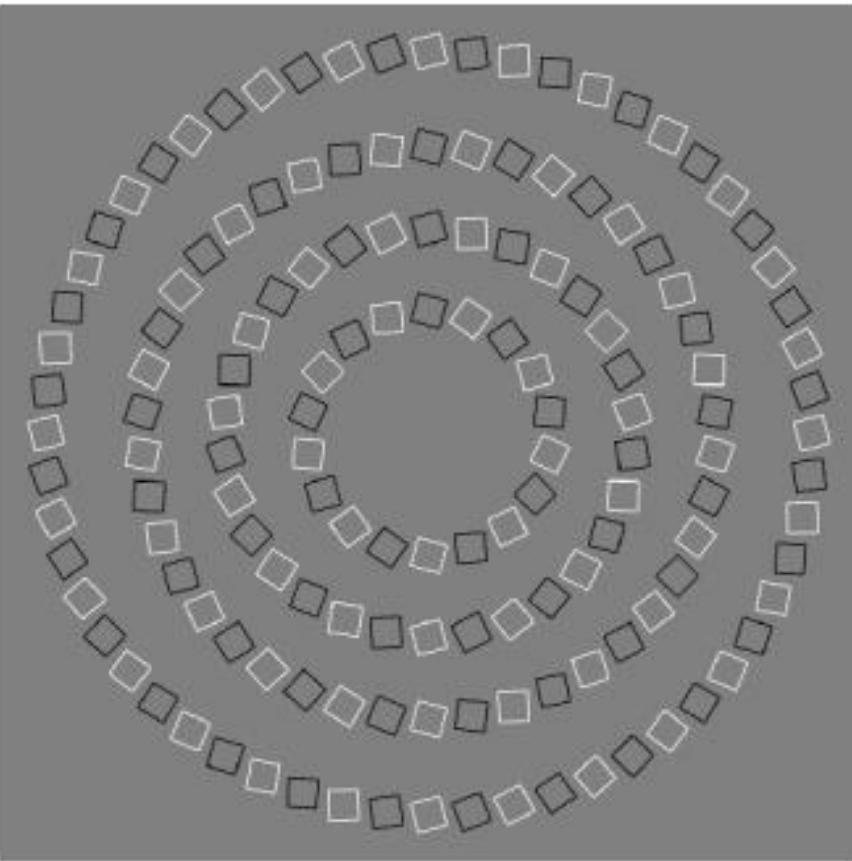


█ classified as turtle █ classified as rifle
█ classified as other



█ classified as baseball █ classified as espresso
█ classified as other

Human adversarial examples?



These are
concentric
circles,
not
intertwined
spirals.

(Pinna and Gregory, 2002)

(Goodfellow 2016)

Human adversarial examples?

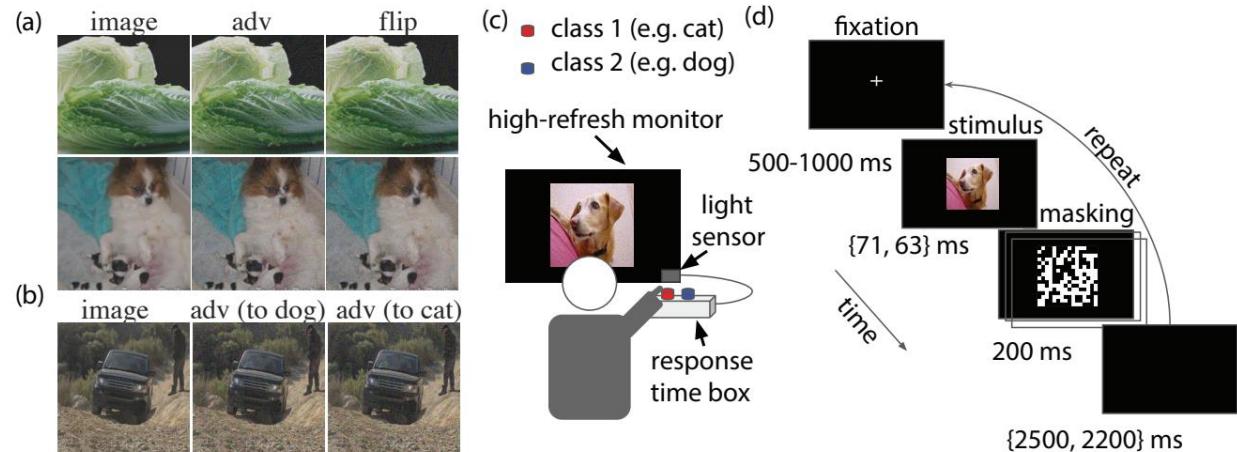


Figure 1: **Experiment setup and task.** (a) examples images from the conditions (image, adv, and flip). Top: adv targeting broccoli class. bottom: adv targeting cat class. See definition of conditions at Section 3.2.2 (b) example images from the false experiment condition. (c) Experiment setup and recording apparatus. (d) Task structure and timings. The subject is asked to repeatedly identify which of two classes (e.g. dog vs. cat) a briefly presented image belongs to. The image is either adversarial, or belongs to one of several control conditions. See Section 3.2 for details.

What does this have to do with generalization?

Linear hypothesis is relevant not just for adversarial examples, but for understanding how neural nets do (and don't) generalize

When you train a model to classify cats vs. dogs, it is not actually learning what cats and dogs look like, it is learning about the patterns **in your dataset**

From there, it will extrapolate in potentially weird ways

Put another way, adversarial examples are not **bugs** they are **features** of your learning algorithm

Literally: Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, Aleksander Madry. **Adversarial Examples Are Not Bugs, They Are Features**. 2019.

Basic idea: neural nets pay attention to “adversarial directions” because it **helps** them to get the right answer on the training data!



Summary

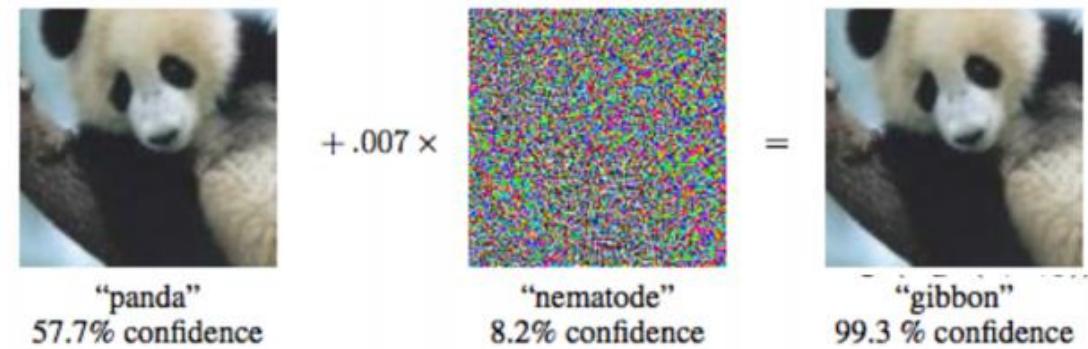
- Neural nets generalize very well on tests sets drawn from the **same distribution** as the training set
- They sometimes do this by being a smart horse
 - This is not their fault! It's your fault for asking the wrong question
- They are often not **well-calibrated**, especially on out-of-distribution inputs
- A related (but not the same!) problem is that we can almost always synthesize **adversarial examples** by modifying normal images to "fool" a neural network into producing an incorrect label
- Adversarial examples are most likely **not** a symptom of overfitting
 - They are conserved across different models, and affect low-capacity models
- There is reason to believe they are actually due to excessively linear (simple) models attempting to extrapolate + distribution shift



(a) Husky classified as wolf

(b) Explanation

Figure 11: Raw data and explanation of a bad model's prediction in the "Husky vs Wolf" task.



Adversarial attacks

A formal definition

Caveat: formally defining an adversarial attack is helpful for mathematicians, but can hide some important real-world considerations

relation: $R(x, x')$

original image altered image

example: $R_\infty(x, x') = \|x - x'\|_\infty$

each pixel changed by at most ϵ

attack: $x^* \leftarrow \arg \max_{x': R(x, x') \leq \epsilon} \mathcal{L}_\theta(x', y)$

pick image x' close to x

e.g., $\mathcal{L}_\theta(x, y) = -\log p_\theta(y|x)$

maximize the loss of your choice

defense: $\theta^* \leftarrow \arg \min_\theta \sum_{(x, y) \in D} \max_{x': R(x, x') \leq \epsilon} \mathcal{L}_\theta(x', y)$

robust loss

Real attackers don't care about your definitions



speed limit: 45
what is ϵ ?

Fast gradient sign method (FGSM)

A **very simple** approximate method for an infinity norm relation

$$R(x, x') = \|x - x'\|_\infty$$

$$\text{attack: } x^* \leftarrow \arg \max_{x': R(x, x') \leq \epsilon} \mathcal{L}_\theta(x', y)$$

$$\text{"first order" assumption: } \mathcal{L}(x', y) \approx \mathcal{L}(x, y) + (x' - x)^T \nabla_x \mathcal{L}$$

ordinarily, we might think that this would make for a very **weak** attack,
but we saw before how neural nets seem to behave locally linearly!

$$\text{attack: } x^* \leftarrow \arg \max_{x': \|x - x'\|_\infty \leq \epsilon} (x' - x)^T \nabla_x \mathcal{L}$$

optional solution: move each dimension of x in direction of $\nabla_x \mathcal{L}$ by ϵ

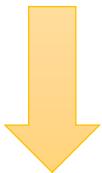
$$x^* = x + \epsilon \text{sign}(\nabla_x \mathcal{L})$$

this works **very well** against standard (naïve)
neural nets

it can be defeated with simple defenses, but
more advanced attacks can be more
resilient

A more general formulation

attack: $x^* \leftarrow \arg \max_{x': R(x, x') \leq \epsilon} \mathcal{L}_\theta(x', y)$



Lagrange multiplier
could be chosen heuristically
or optimized with e.g. dual gradient descent

$$x^* \leftarrow \arg \max_{x'} \mathcal{L}_\theta(x', y) - \lambda R(x, x')$$

optimize to convergence, for example with ADAM

$$\delta = x' - x$$

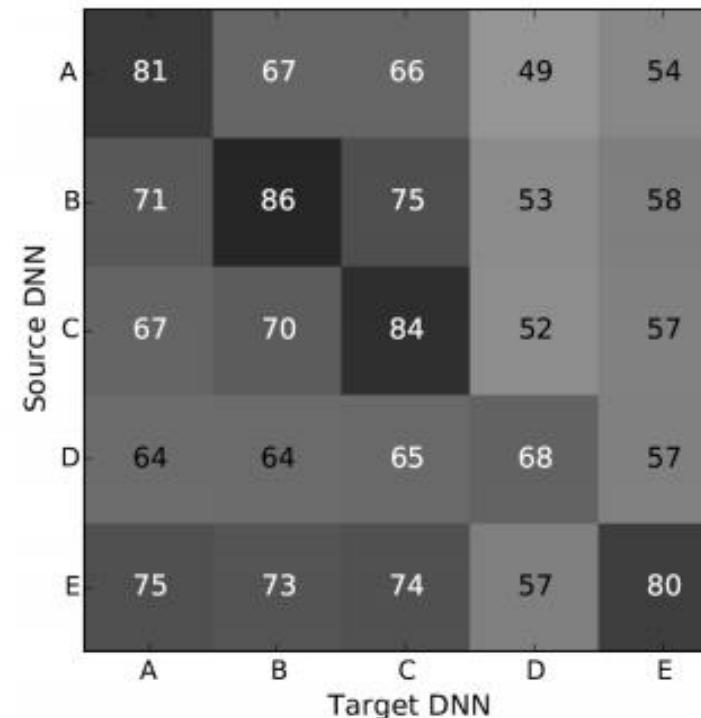
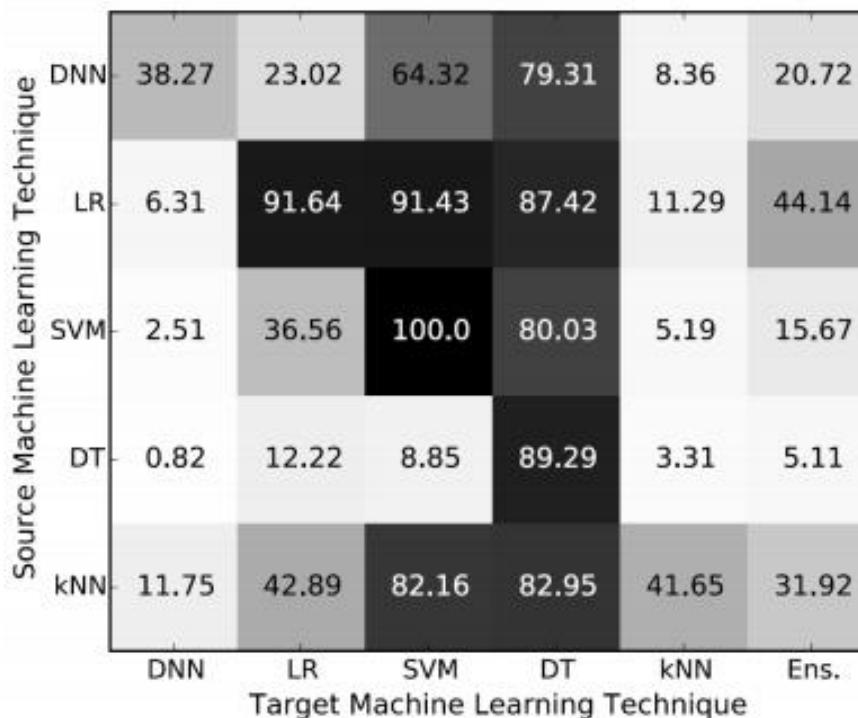
$$\delta^* \leftarrow \arg \max_\delta \mathcal{L}_\theta(x + \delta, y) - \lambda \|\delta\|_\infty$$

In general can use a variety of losses
here, including perceptual losses

In general, such attacks are very hard to defeat

Transferability of adversarial attacks

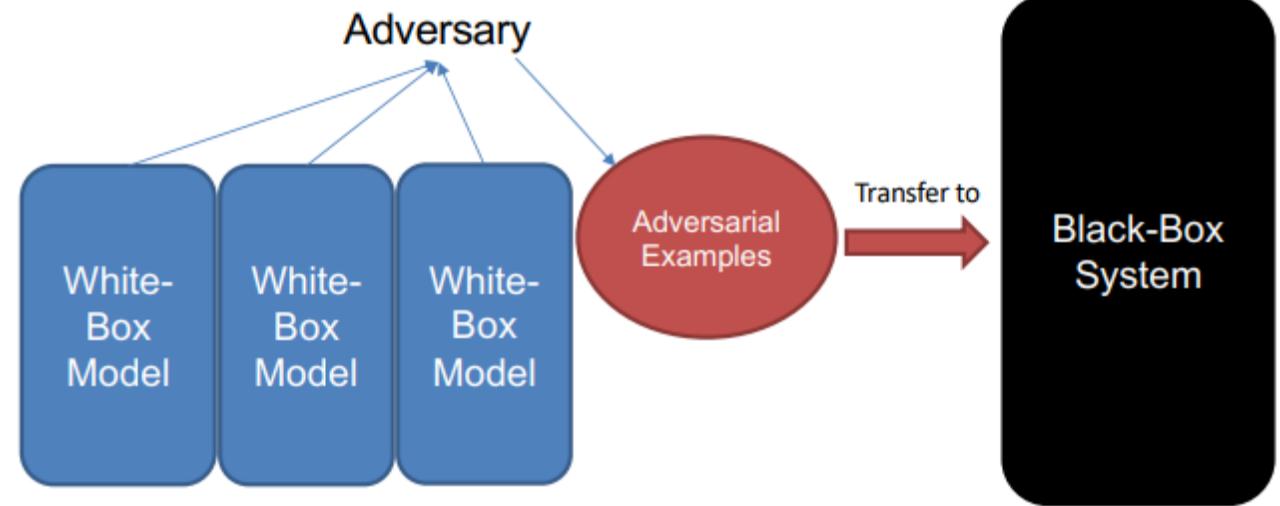
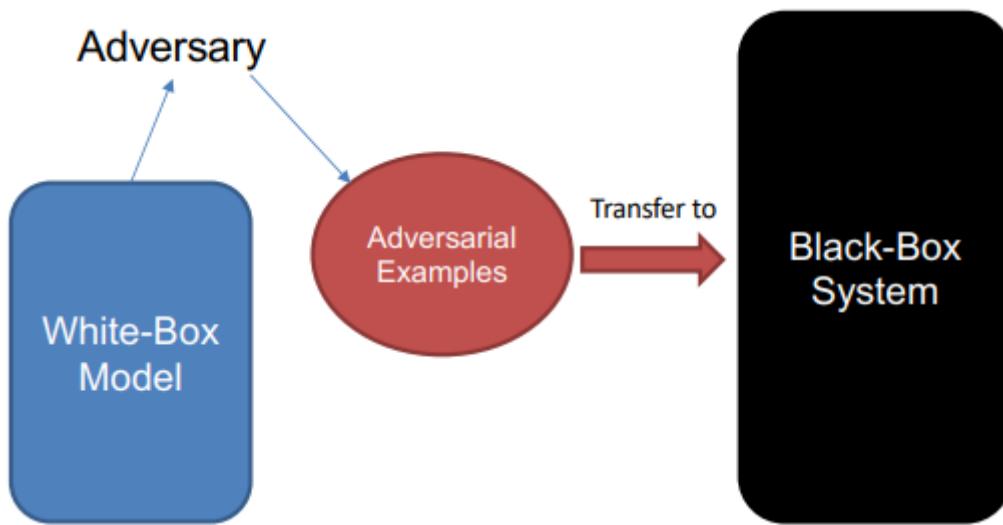
Oftentimes it just works



In particular, this means that we often don't need direct gradient access to a neural net we are actually attacking – we can just use **another** neural net to construct our adversarial example!

% success rate at fooling one model when trained on another

Zero-shot black-box attack



Finite differences gradient estimation

It's possible to estimate the gradient with a moderate number of queries to a model (e.g., on a web server) without being able to actually directly access its gradient

$$x^* = x + \epsilon \text{sign}(\nabla_x \mathcal{L})$$

all we need is the sign of the gradient

for each dimension i of x :

$$\begin{aligned} &\text{get } v_i \leftarrow \mathcal{L}(x + 10^{-3} e_i, y) \\ &\nabla_x \mathcal{L} \approx (v - \mathcal{L}(x, y)) / (10^{-3}) \end{aligned}$$

a small number

i^{th} canonical vector

If you really want to do this, there are fancy tricks to even further reduce how many queries are needed to estimate the gradient

Defending against adversarial attacks?

There are **many** different methods in the literature for “robustifying” models against adversarial examples

$$\text{defense: } \theta^* \leftarrow \arg \min_{\theta} \underbrace{\sum_{(x,y) \in D} \max_{x': R(x,x') \leq \epsilon} \mathcal{L}_{\theta}(x', y)}_{\text{robust loss}}$$

Simple recipe: adversarial training

- 1. sample minibatch $\{(x_i, y_i)\}$ from dataset \mathcal{D}
- 2. for each x_i , compute adversarial x'_i
- 3. take SGD step: $\theta \leftarrow \theta - \alpha \sum_i \nabla_{\theta} \mathcal{L}_{\theta}(x'_i, y_i)$

e.g., FGSM: $x'_i \leftarrow x_i + \epsilon \text{sign}(\nabla_{x_i} \mathcal{L}_{\theta}(x_i, y_i))$

usually also add original loss grad $\nabla_{\theta} \mathcal{L}_{\theta}(x_i, y_i)$

Usually doesn't come for free:

increases robustness to adversarial attacks (lower % fooling rate)

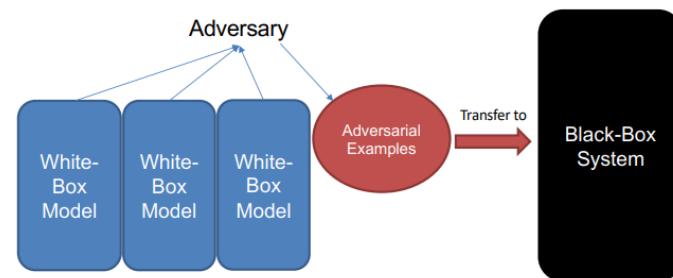
decreases overall accuracy on the test set (compared to naïve network)

Summary

- **Fast gradient sign method:** a simple and convenient way to construct an attack with an infinity-norm constraint (i.e., each pixel can change by at most a small amount)
- **Better attack methods:** use many steps of gradient descent to optimize an image subject to a constraint
- **Black box attack** without access to a model's gradients
 - **Construct** your own model (or an ensemble), attack those, and then transfer the adversarial example in zero shot
 - **Estimate** the gradient using queries (e.g., finite differences)
- **Defenses:** heavily studied topic! But very hard

$$x^* = x + \epsilon \text{sign}(\nabla_x \mathcal{L})$$

$$\delta^* \leftarrow \arg \max_{\delta} \mathcal{L}_{\theta}(x + \delta, y) - \lambda \|\delta\|_{\infty}$$



Meta-Learning

Designing, Visualizing and Understanding Deep Neural Networks

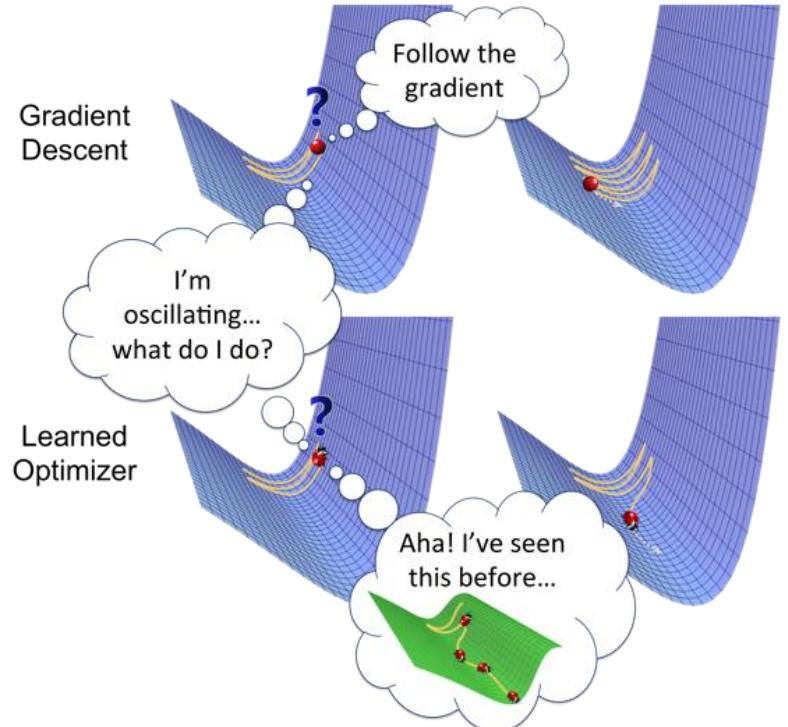
CS W182/282A

Instructor: Sergey Levine
UC Berkeley



What is meta-learning?

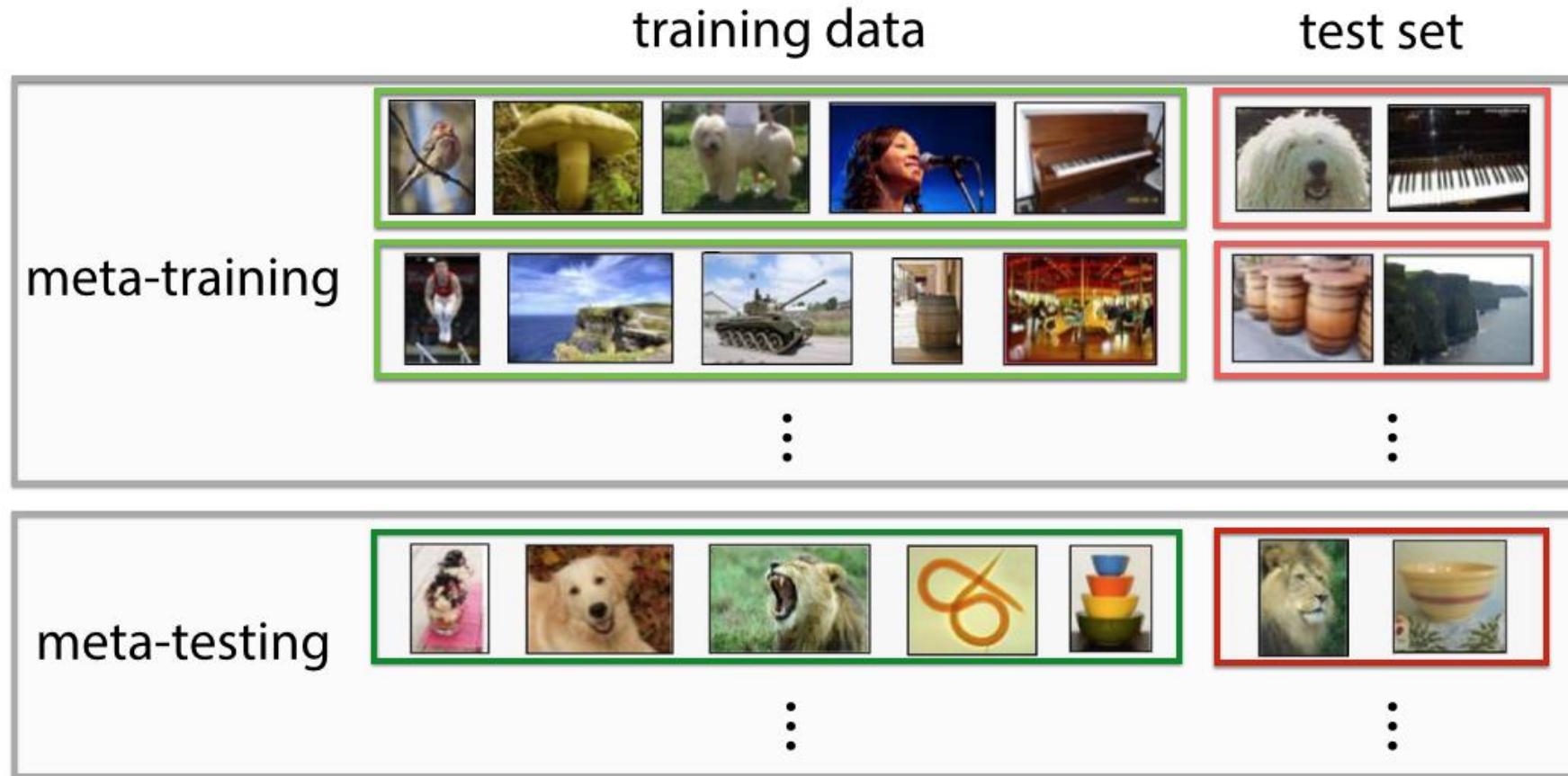
- If you've learned 100 tasks already, can you figure out how to *learn* more efficiently?
 - Now having multiple tasks is a huge advantage!
- Meta-learning = *learning to learn*
- In practice, very closely related to multi-task learning
- Many formulations
 - Learning an optimizer
 - Learning an RNN that ingests experience
 - Learning a representation



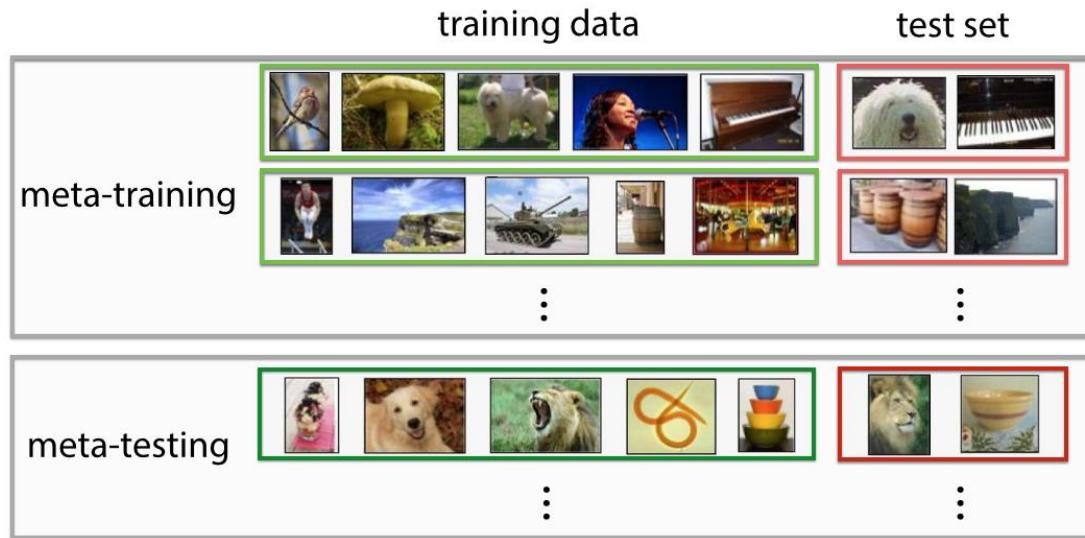
Why is meta-learning a good idea?

- Deep learning works very well, but requires **large** datasets
- In many cases, we only have a small amount of data available (e.g., some specific computer vision task), but we might have lots of data of a similar type for other tasks (e.g., other object classification tasks)
- How does a *meta-learner* help with this?
 - Use plentiful prior tasks to meta-train a model that can learn a new task quickly with only a few examples
 - Collect a small amount of labeled data for the new task
 - Learn a model on this new dataset that generalizes broadly

Meta-learning with supervised learning

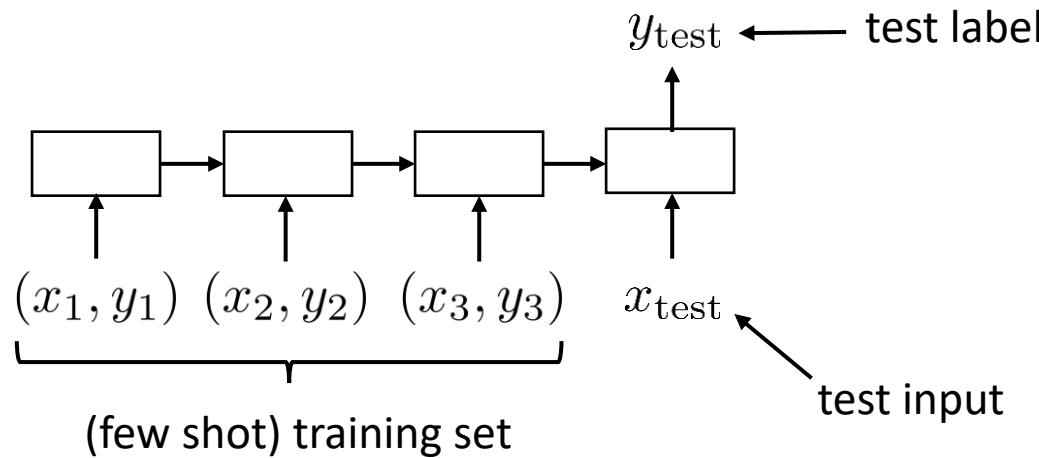


Meta-learning with supervised learning

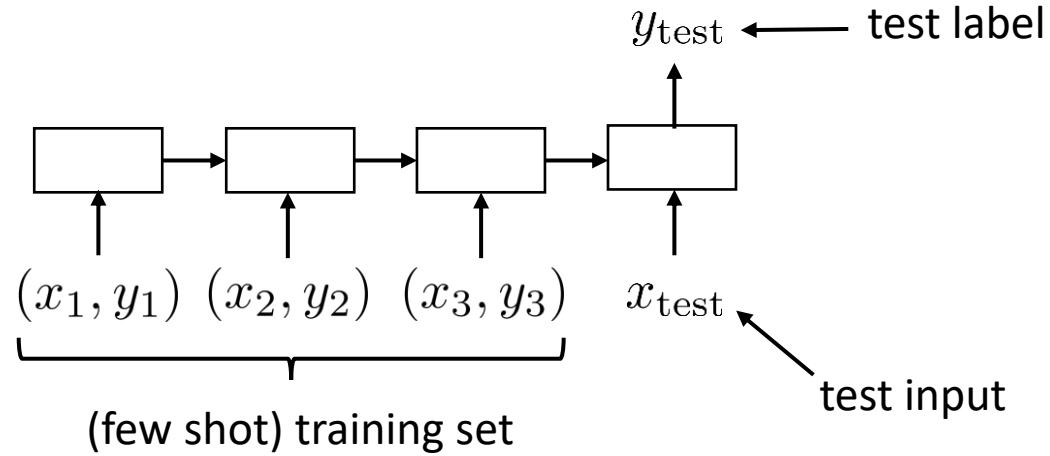


supervised learning: $f(x) \rightarrow y$
input (e.g., image) output (e.g., label)

supervised meta-learning: $f(\mathcal{D}^{\text{tr}}, x) \rightarrow y$
training set



What is being “learned”?



supervised meta-learning: $f(\mathcal{D}^{\text{tr}}, x) \rightarrow y$

“Generic” learning:

$$\begin{aligned}\theta^* &= \arg \min_{\theta} \mathcal{L}(\theta, \mathcal{D}^{\text{tr}}) \\ &= f_{\text{learn}}(\mathcal{D}^{\text{tr}})\end{aligned}$$

“Generic” meta-learning:

$$\begin{aligned}\theta^* &= \arg \min_{\theta} \sum_{i=1}^n \mathcal{L}(\phi_i, \mathcal{D}_i^{\text{ts}}) \\ &\text{where } \phi_i = f_{\theta}(\mathcal{D}_i^{\text{tr}})\end{aligned}$$

What is being “learned”?

“Generic” learning:

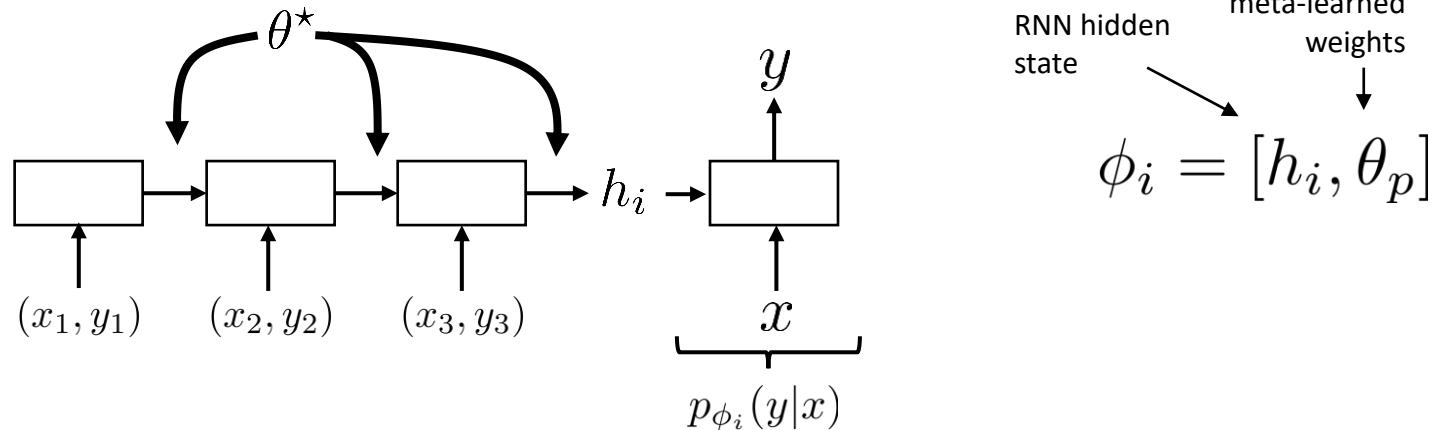
$$\theta^* = \arg \min_{\theta} \mathcal{L}(\theta, \mathcal{D}^{\text{tr}})$$

$$= f_{\text{learn}}(\mathcal{D}^{\text{tr}})$$

“Generic” meta-learning:

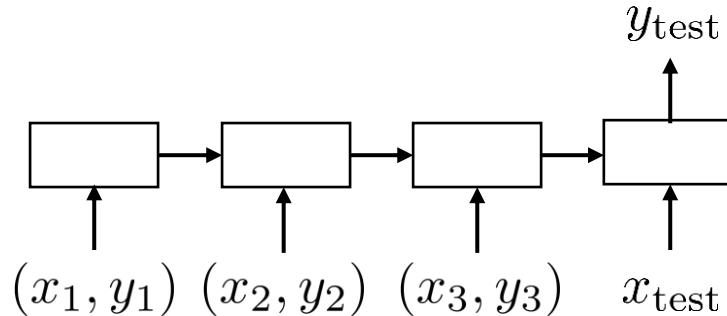
$$\theta^* = \arg \min_{\theta} \sum_{i=1}^n \mathcal{L}(\phi_i, \mathcal{D}_i^{\text{ts}})$$

$$\text{where } \phi_i = f_{\theta}(\mathcal{D}_i^{\text{tr}})$$

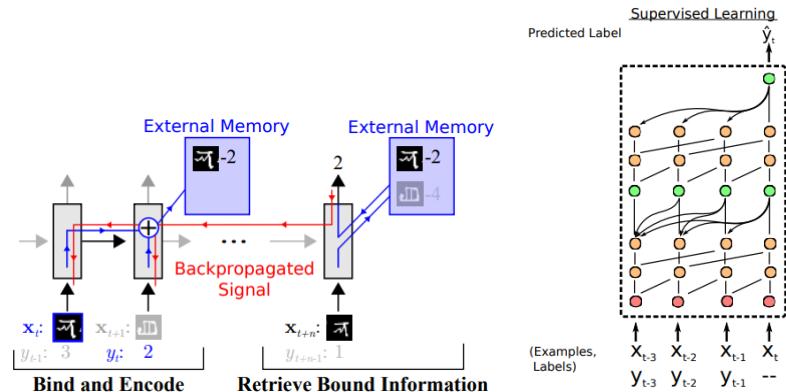


Meta-learning methods

black-box meta-learning

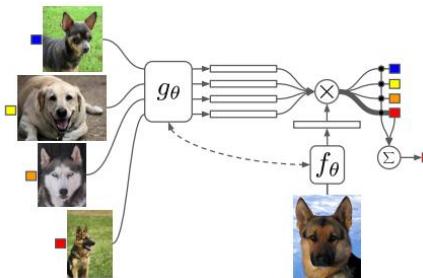


some kind of network that can read in an entire (few-shot) training set

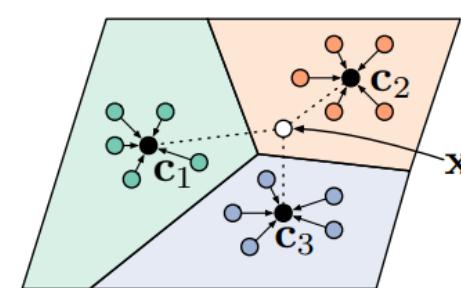


Santoro et al. **Meta-Learning with Memory-Augmented Neural Networks**. 2016.

non-parametric meta-learning

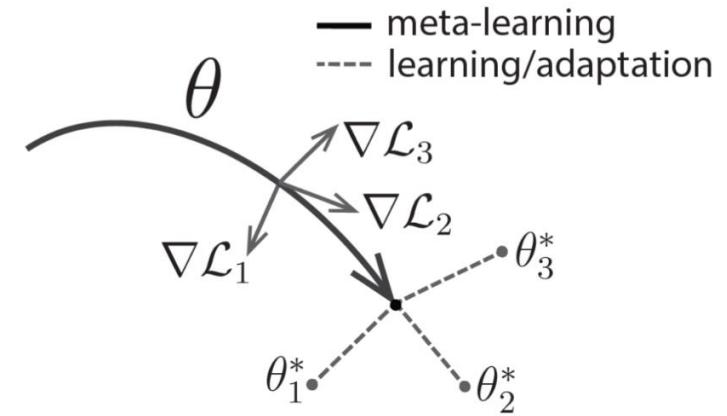


Vinyals et al. **Matching Networks for One Shot Learning**. 2017.



Snell et al. **Prototypical Networks for Few-shot Learning**. 2018.

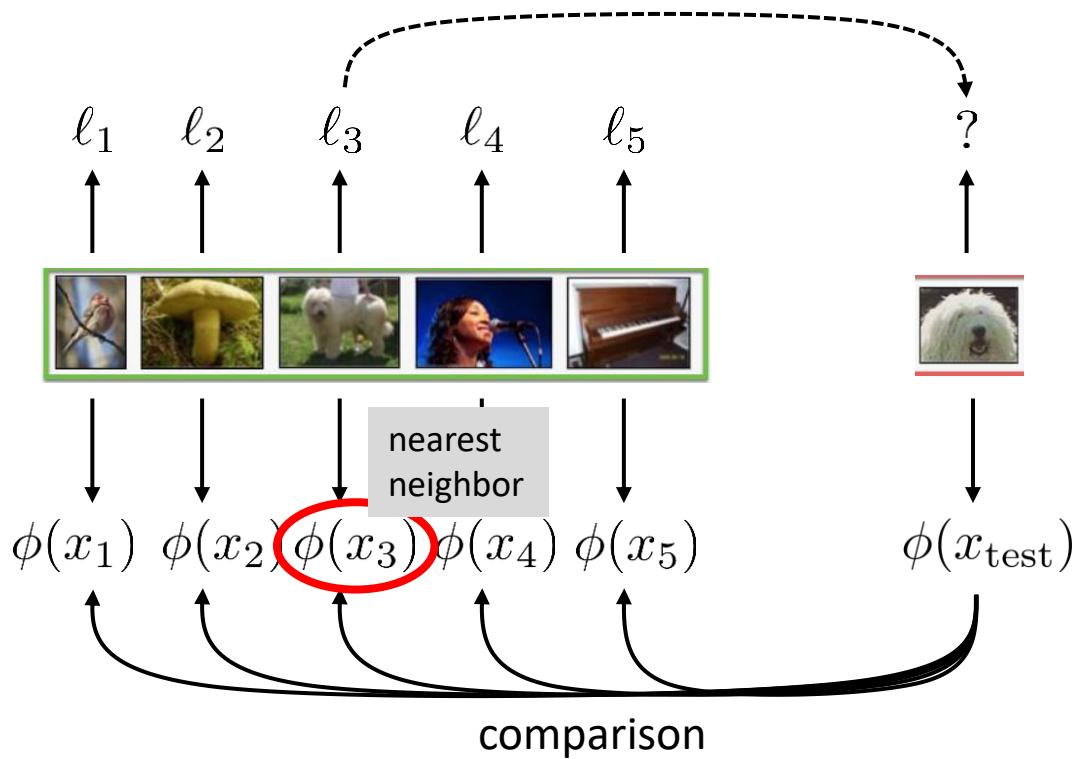
gradient-based meta-learning



Finn et al. **Model-Agnostic Meta-Learning**. 2018.

Non-Parametric & Gradient-Based Meta-Learning

Basic idea



$$\theta^* = \arg \min_{\theta} \sum_{i=1}^n \mathcal{L}(f_{\theta}(\mathcal{D}_i^{\text{tr}}), \mathcal{D}_i^{\text{ts}}) = - \sum_{i=1}^n \sum_{j=1}^m \log p_{\theta}(y_j^{\text{ts}} | x_j^{\text{ts}}, \mathcal{D}_i^{\text{tr}})$$

learned (soft) nearest
neighbor classifier

why does this work?

that is, why does the nearest neighbor have the right class?

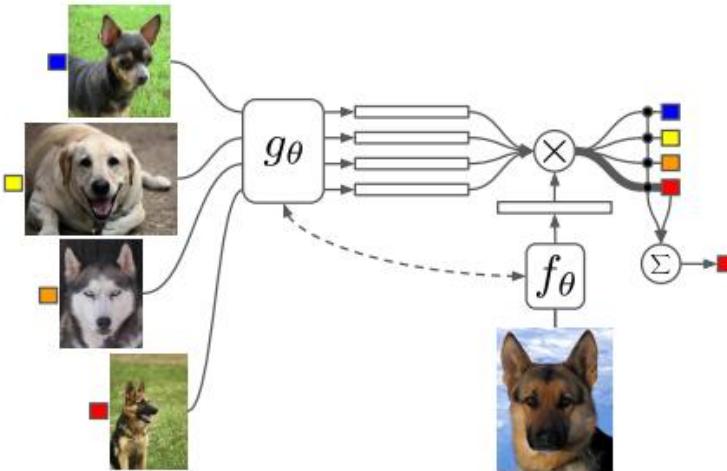
because we **meta-train** the features so that this produces the right answer!

$$p_{\text{nearest}}(x_k^{\text{tr}} | x_j^{\text{ts}}) \propto \exp(\phi(x_k^{\text{tr}})^T \phi(x_j^{\text{ts}}))$$

$$p_{\theta}(y_j^{\text{ts}} | x_j^{\text{ts}}, \mathcal{D}_i^{\text{tr}}) = \underbrace{\sum_{k: y_k^{\text{tr}} = y_j^{\text{ts}}} p_{\text{nearest}}(x_k^{\text{tr}} | x_j^{\text{ts}})}_{\text{all training points that have this label}}$$

all training points that have this label

Matching networks

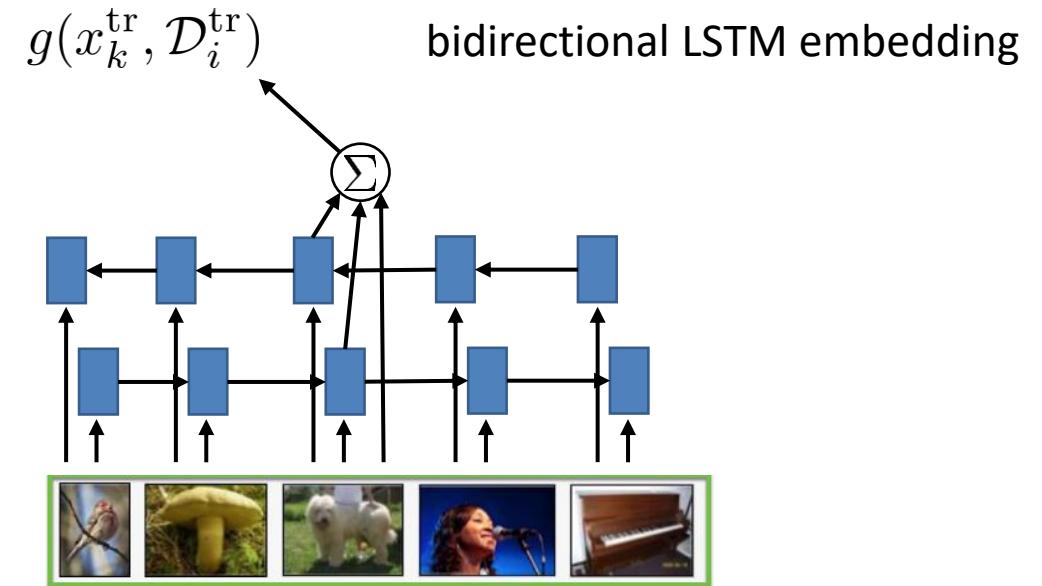


$$p_\theta(y_j^{\text{ts}} | x_j^{\text{ts}}, \mathcal{D}_i^{\text{tr}}) = \sum_{k: y_k^{\text{tr}} = y_j^{\text{ts}}} p_{\text{nearest}}(x_k^{\text{tr}} | x_j^{\text{ts}})$$

$$p_{\text{nearest}}(x_k^{\text{tr}} | x_j^{\text{ts}}) \propto \exp(g(x_k^{\text{tr}}, \mathcal{D}_i^{\text{tr}})^T f(x_j^{\text{ts}}, \mathcal{D}_i^{\text{tr}}))$$

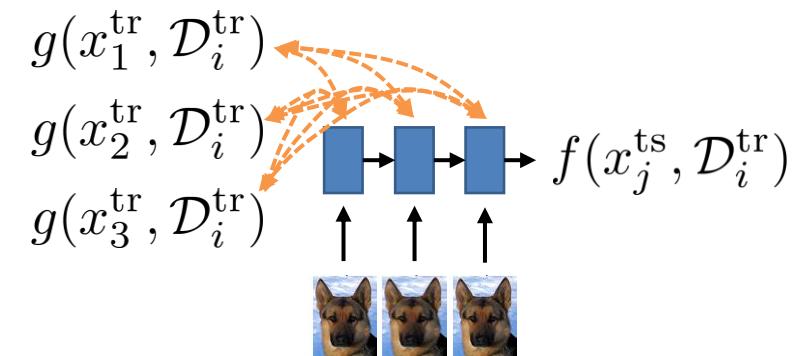
different nets to embed x^{tr} and x^{ts}

both f and g conditioned on entire set $\mathcal{D}_i^{\text{tr}}$

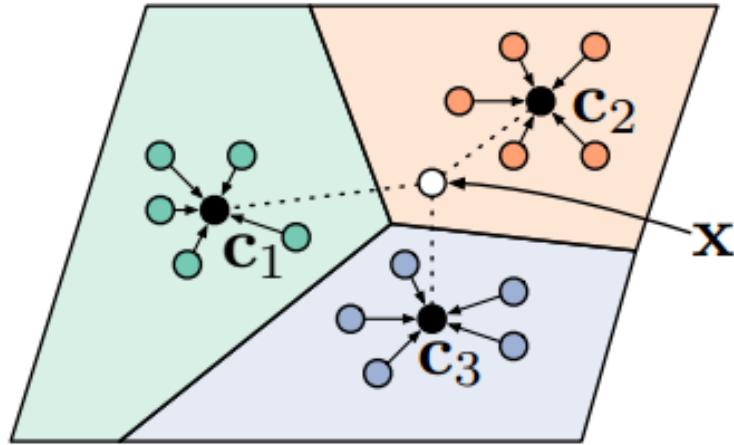


$$f(x_j^{\text{ts}}, \mathcal{D}_i^{\text{tr}})$$

attentional LSTM embedding



Prototypical networks



Two simple ideas compared to matching networks:

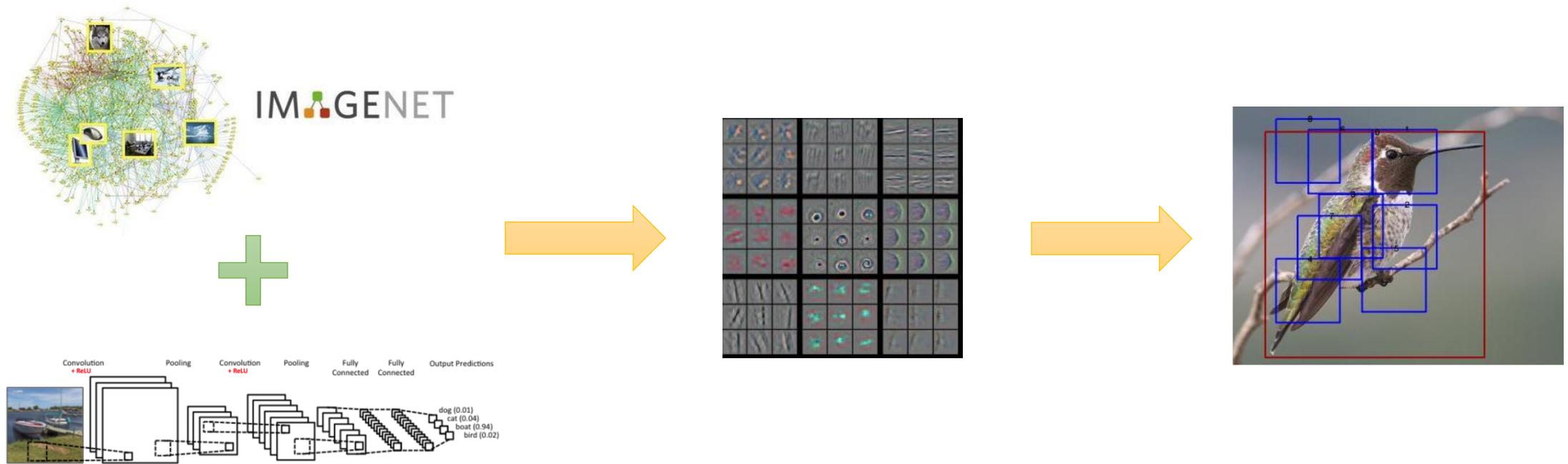
1. Instead of “soft nearest neighbor,” construct prototype for each class

$$p_{\theta}(y|x_j^{\text{ts}}, \mathcal{D}_i^{\text{tr}}) \propto \exp(c_y^T f(x_j^{\text{ts}})) \quad c_y = \frac{1}{N_y} \sum_{k:y_k^{\text{tr}}=y} g(x_k^{\text{tr}})$$

2. Get rid of all the complex junk

~~— bidirectional LSTM embedding~~
~~— attentional LSTM embedding~~

Back to representations...



is pretraining a *type* of meta-learning?
better features = faster learning of new task!

Meta-learning as an optimization problem

$$\theta^* = \arg \min_{\theta} \sum_{i=1}^n \mathcal{L}(\phi_i, \mathcal{D}_i^{\text{ts}})$$

where $\phi_i = f_{\theta}(\mathcal{D}_i^{\text{tr}})$

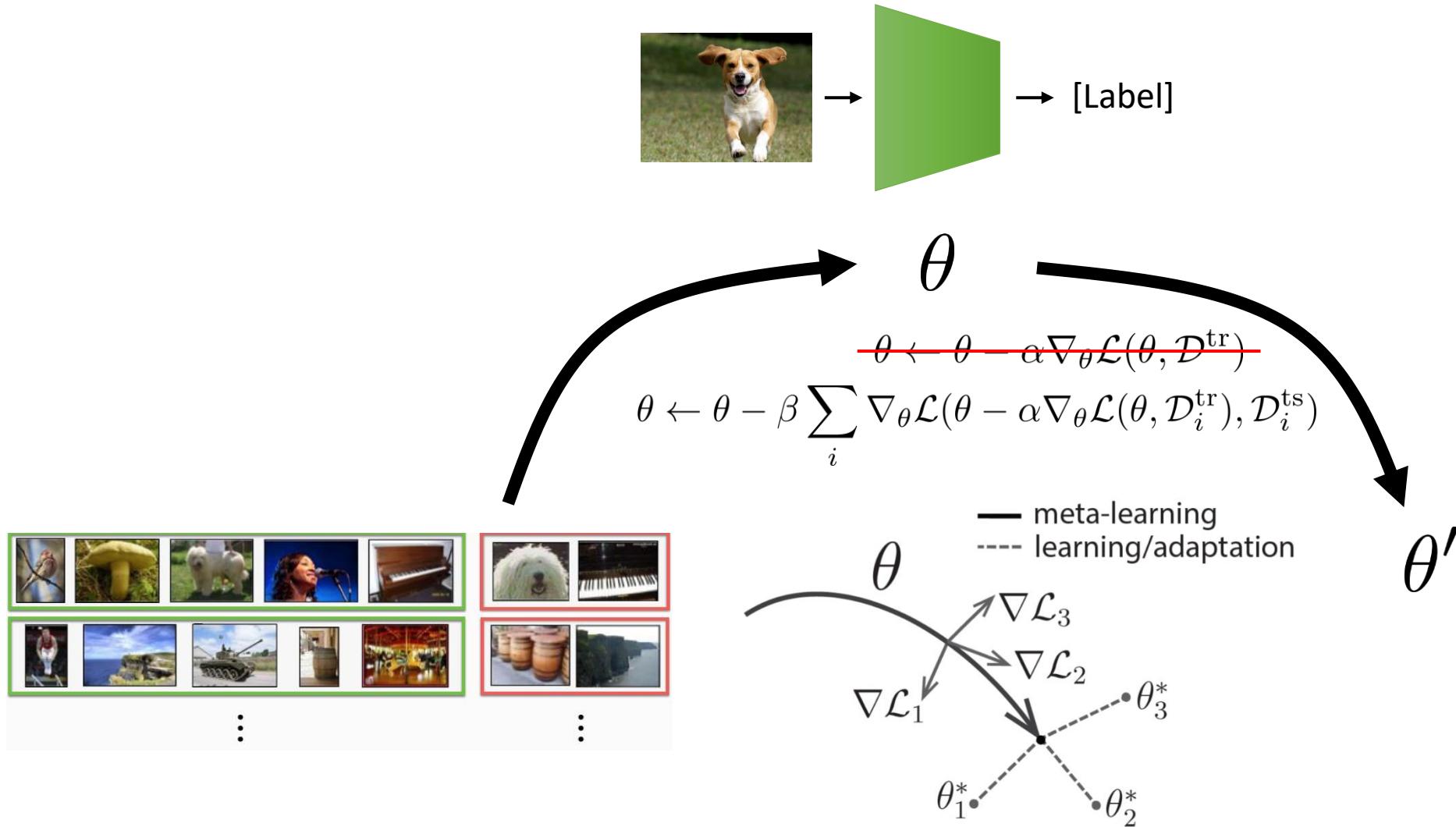
what if $f_{\theta}(\mathcal{D}_i^{\text{tr}})$ is just a *finetuning* algorithm?

$$f_{\theta}(\mathcal{D}_i^{\text{tr}}) = \theta - \alpha \nabla_{\theta} \mathcal{L}(\theta, \mathcal{D}_i^{\text{tr}})$$

(could take a few gradient steps in general)

This can be trained the same way as any other neural network, by implementing gradient descent as a computation graph and then running backpropagation *through* gradient descent!

MAML in pictures



What did we just do??

supervised learning: $f(x) \rightarrow y$

supervised meta-learning: $f(\mathcal{D}^{\text{tr}}, x) \rightarrow y$

model-agnostic meta-learning: $f_{\text{MAML}}(\mathcal{D}^{\text{tr}}, x) \rightarrow y$

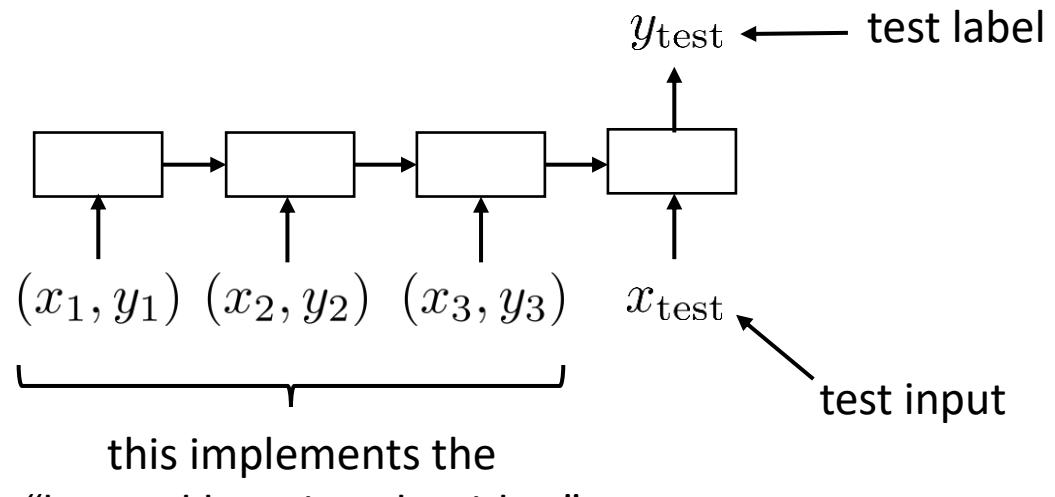
$$f_{\text{MAML}}(\mathcal{D}^{\text{tr}}, x) = f_{\theta'}(x)$$

$$\theta' = \theta - \alpha \sum_{(x,y) \in \mathcal{D}^{\text{tr}}} \nabla_{\theta} \mathcal{L}(f_{\theta}(x), y)$$

Just another computation graph...
Can implement with any autodiff
package (e.g., TensorFlow)
But has favorable inductive bias...

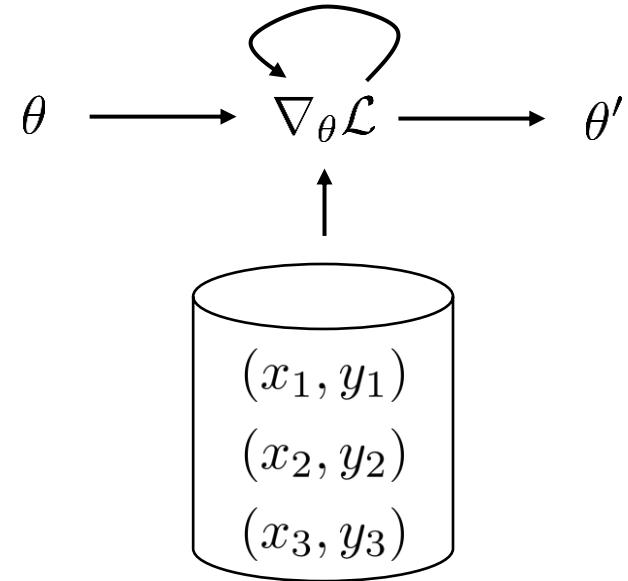
Why does it work?

black-based meta-learning



- Does it converge?
 - Kind of?
- What does it converge to?
 - Who knows...
- What to do if it's not good enough?
 - Nothing...

MAML



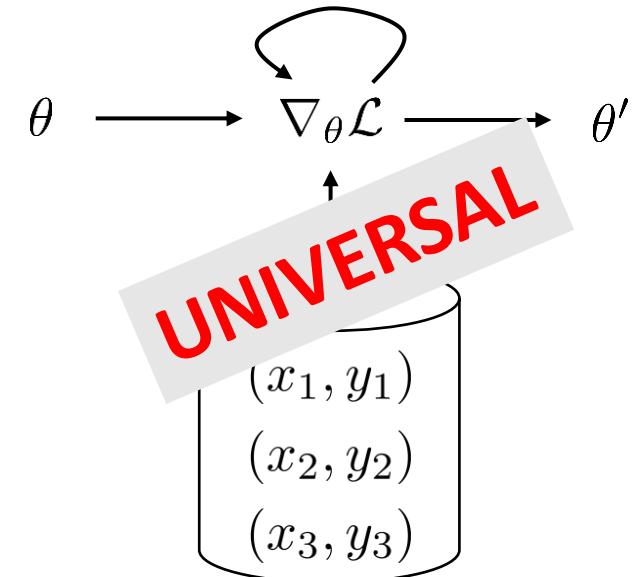
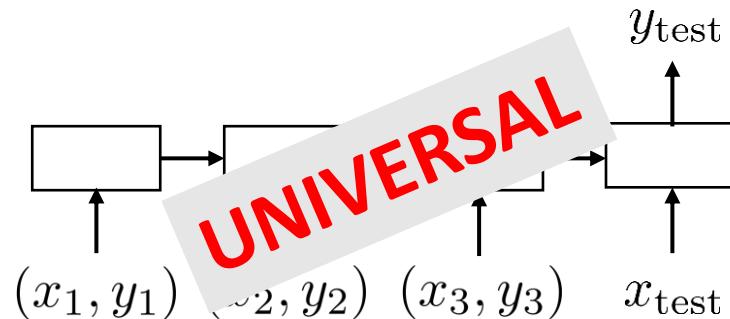
- Does it converge?
 - Yes (it's gradient descent...)
- What does it converge to?
 - A local optimum (it's gradient descent...)
- What to do if it's not good enough?
 - Keep taking gradient steps (it's gradient descent...)

Universality

Did we lose anything?

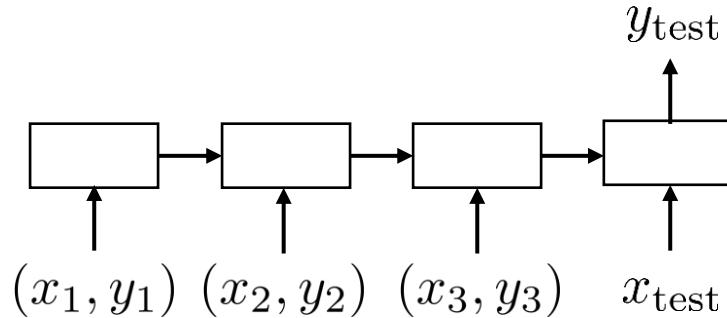
Universality: meta-learning can learn any “algorithm”

more precisely, can represent any function $f(\mathcal{D}_{\text{train}}, x)$

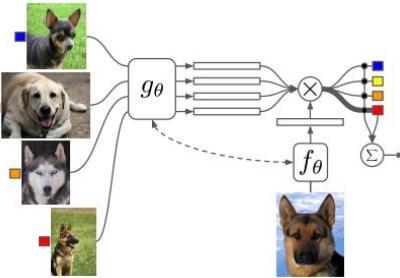


Summary

black-box meta-learning



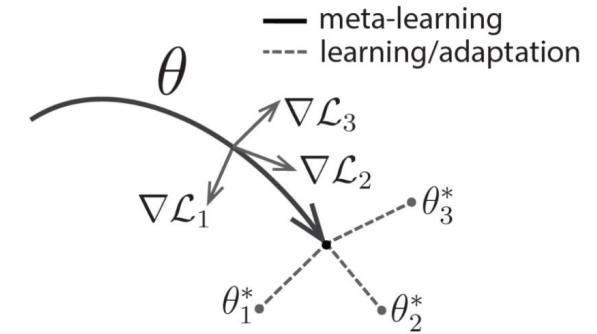
non-parametric meta-learning



Vinyals et al. **Matching Networks for One Shot Learning**. 2017.

- + conceptually very simple
- + benefits from advances in sequence models (e.g., transformers)
- minimal inductive bias (i.e., **everything** has to be meta-learned)
- hard to scale to “medium” shot (we get long “sequences”)

gradient-based meta-learning



Finn et al. **Model-Agnostic Meta-Learning**. 2018.

- + can work very well by combining some inductive bias with easy end-to-end optimization
- restricted to classification, hard to extend to other settings like regression or reinforcement learning
- somewhat specialized architectures

- + easy to apply to any architecture or loss function (inc. RL, regression)
- + good generalization to out-of-domain tasks
- meta-training optimization problem is harder, requires more tuning
- requires second derivatives

Meta-Reinforcement Learning

The meta reinforcement learning problem

“Generic” learning:

$$\begin{aligned}\theta^* &= \arg \min_{\theta} \mathcal{L}(\theta, \mathcal{D}^{\text{tr}}) \\ &= f_{\text{learn}}(\mathcal{D}^{\text{tr}})\end{aligned}$$

Reinforcement learning:

$$\begin{aligned}\theta^* &= \arg \max_{\theta} E_{\pi_{\theta}(\tau)}[R(\tau)] \\ &= f_{\text{RL}}(\mathcal{M}) \quad \mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{P}, r\}\end{aligned}$$

MDP

“Generic” meta-learning:

$$\begin{aligned}\theta^* &= \arg \min_{\theta} \sum_{i=1}^n \mathcal{L}(\phi_i, \mathcal{D}_i^{\text{ts}}) \\ \text{where } \phi_i &= f_{\theta}(\mathcal{D}_i^{\text{tr}})\end{aligned}$$

Meta-reinforcement learning:

$$\begin{aligned}\theta^* &= \arg \max_{\theta} \sum_{i=1}^n E_{\pi_{\phi_i}(\tau)}[R(\tau)] \\ \text{where } \phi_i &= f_{\theta}(\mathcal{M}_i)\end{aligned}$$

MDP for task i

The meta reinforcement learning problem

$$\theta^* = \arg \max_{\theta} \sum_{i=1}^n E_{\pi_{\phi_i}(\tau)}[R(\tau)]$$

where $\phi_i = f_{\theta}(\mathcal{M}_i)$

assumption: $\mathcal{M}_i \sim p(\mathcal{M})$

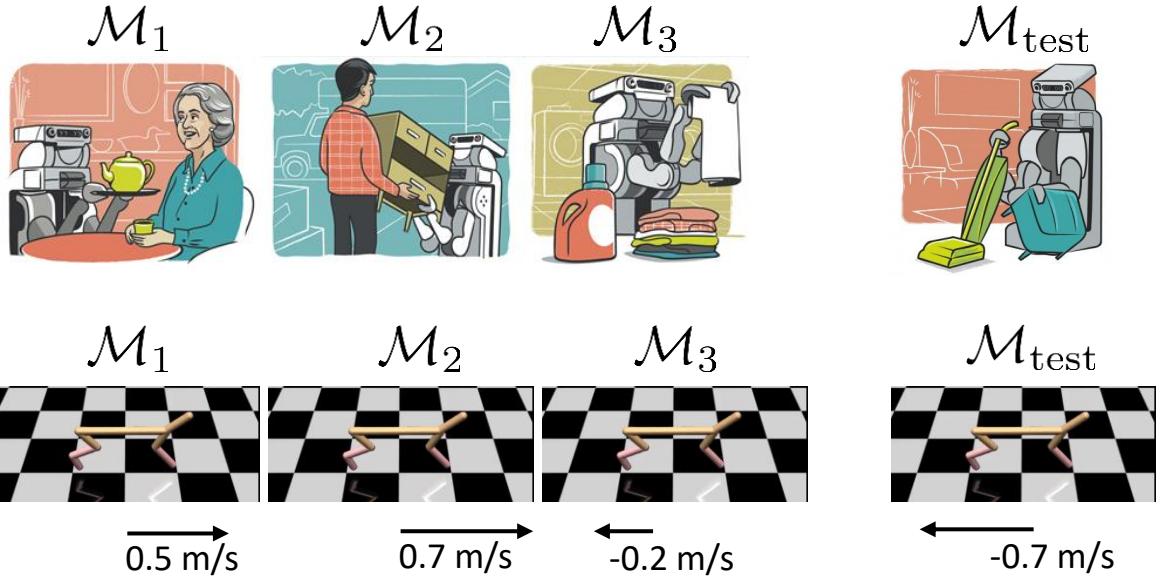
meta test-time:

sample $\mathcal{M}_{\text{test}} \sim p(\mathcal{M})$, get $\phi_i = f_{\theta}(\mathcal{M}_{\text{test}})$

$\{\mathcal{M}_1, \dots, \mathcal{M}_n\}$

↑
meta-training MDPs

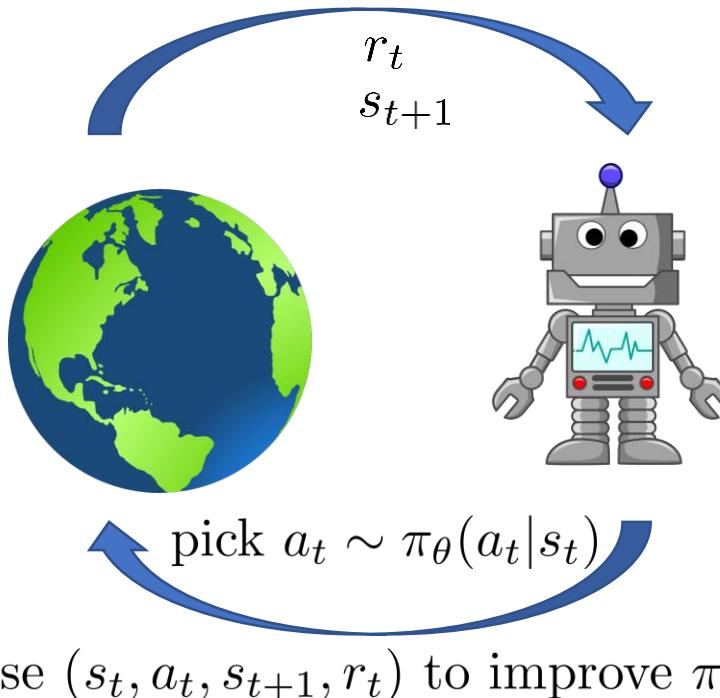
Some examples:



Meta-RL with recurrent policies

$$\theta^* = \arg \max_{\theta} \sum_{i=1}^n E_{\pi_{\phi_i}(\tau)}[R(\tau)]$$

where $\phi_i = f_{\theta}(\mathcal{M}_i)$



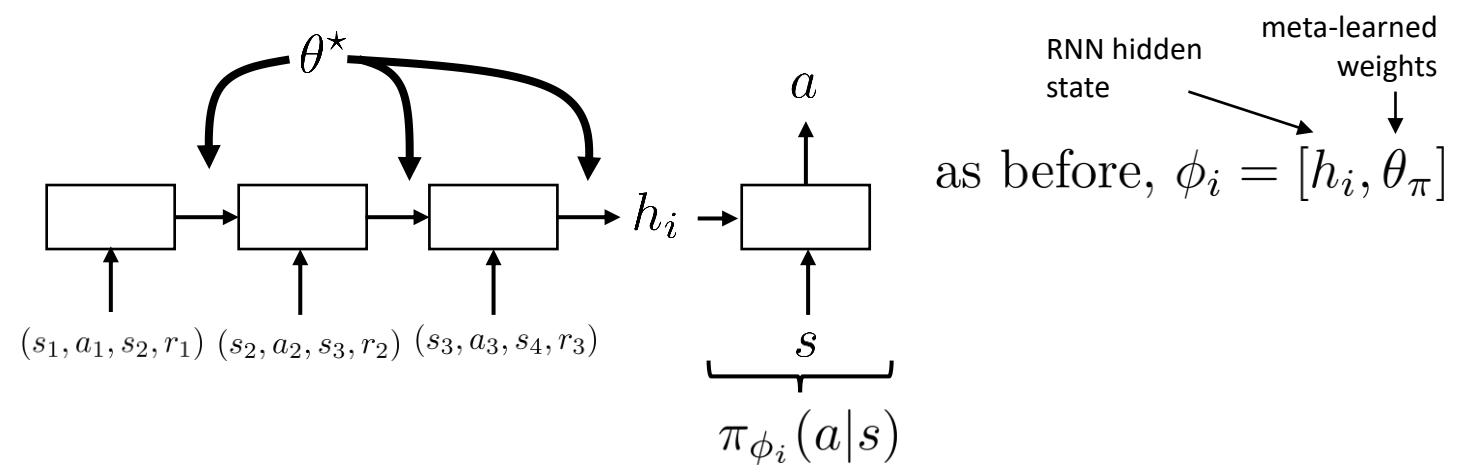
main question: how to implement $f_{\theta}(\mathcal{M}_i)$?

what should $f_{\theta}(\mathcal{M}_i)$ do?

1. improve policy with experience from \mathcal{M}_i

$\{(s_1, a_1, s_2, r_1), \dots, (s_T, a_T, s_{T+1}, r_T)\}$

2. (new in RL): choose how to interact, i.e. choose a_t
meta-RL must also *choose* how to *explore*!



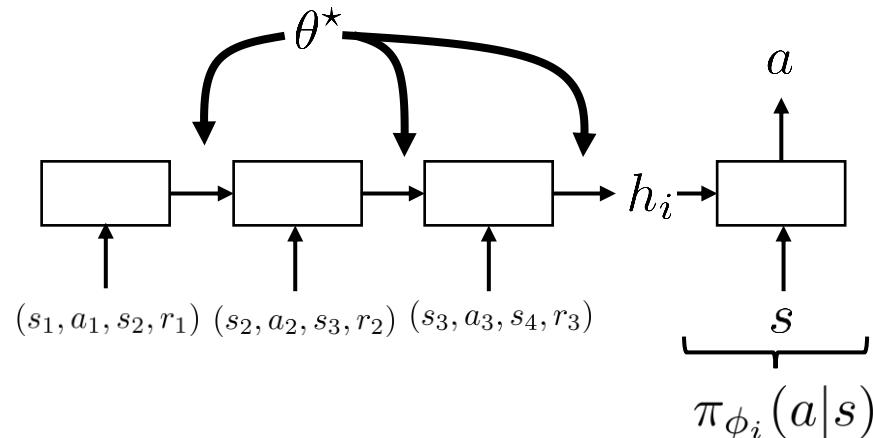
Meta-RL with recurrent policies

$$\theta^* = \arg \max_{\theta} \sum_{i=1}^n E_{\pi_{\phi_i}(\tau)}[R(\tau)]$$

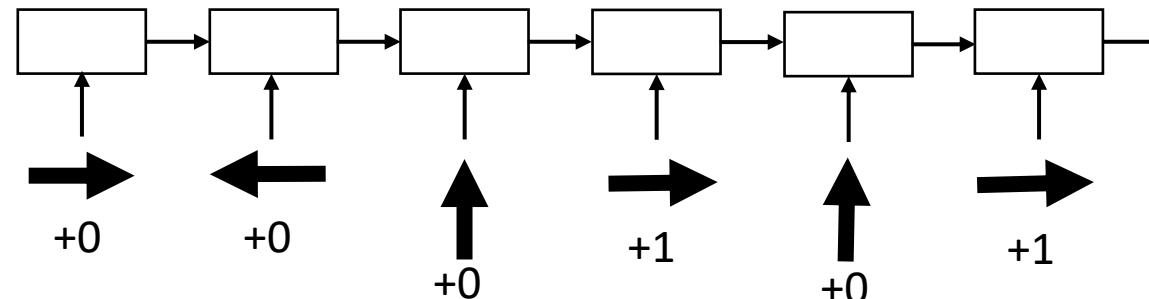
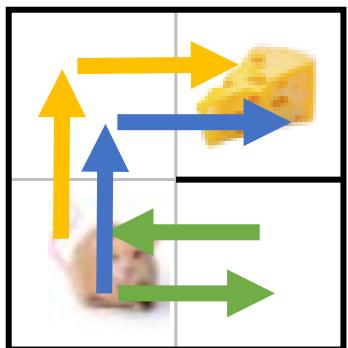
where $\phi_i = f_{\theta}(\mathcal{M}_i)$

so... we just train an RNN policy?

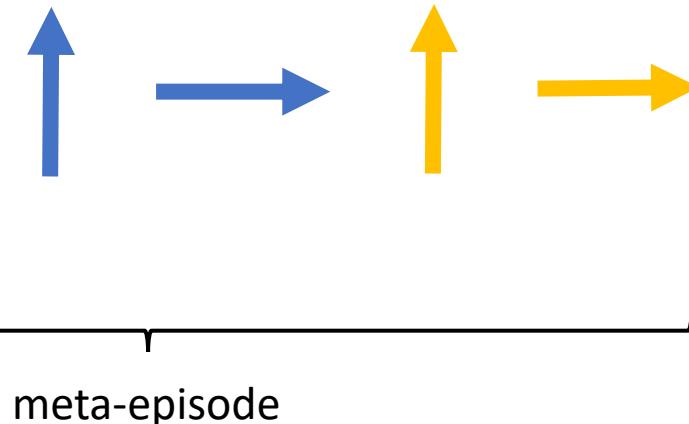
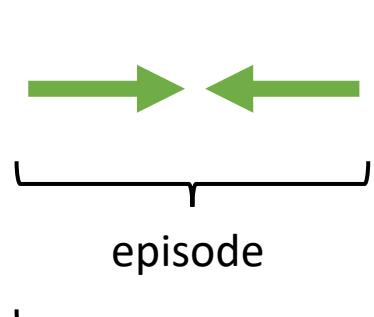
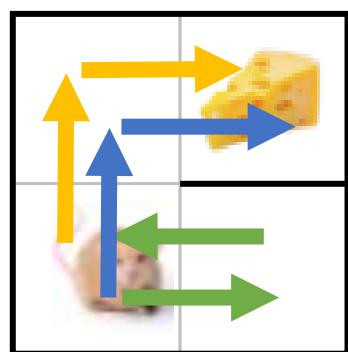
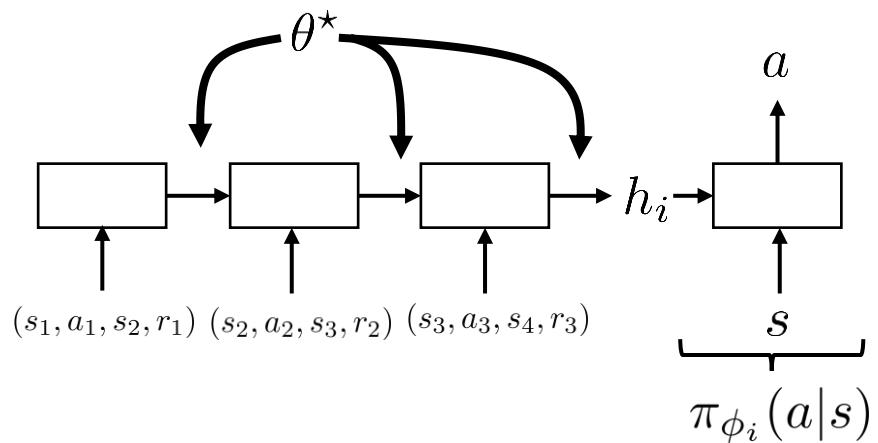
yes!



crucially, RNN hidden state is **not reset between episodes!**



Why recurrent policies *learn to explore*



1. improve policy with experience from \mathcal{M}_i
 $\{(s_1, a_1, s_2, r_1), \dots, (s_T, a_T, s_{T+1}, r_T)\}$
2. (new in RL): choose how to interact, i.e. choose a_t
meta-RL must also *choose* how to *explore*!

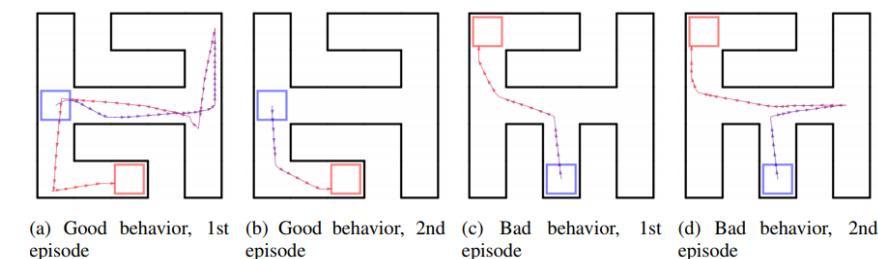
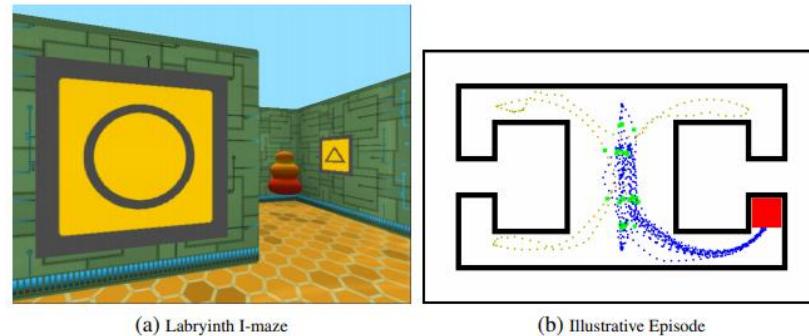
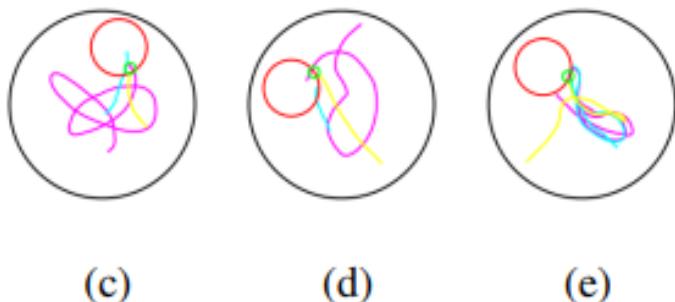
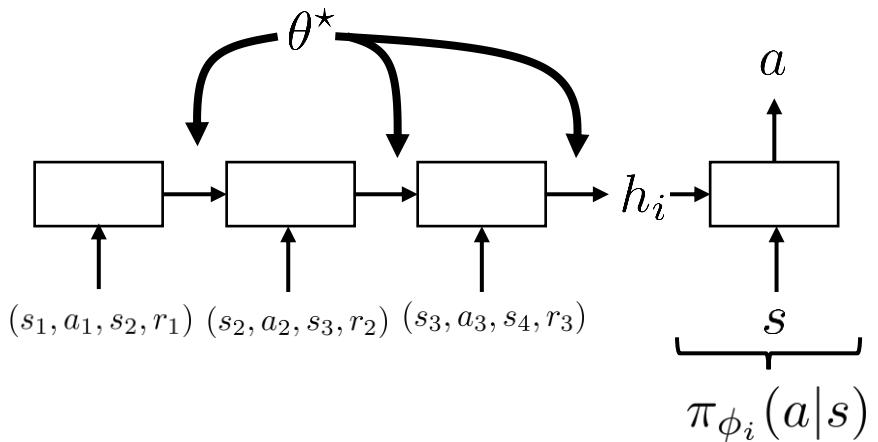
$$\theta^* = \arg \max_{\theta} E_{\pi_{\theta}} \left[\sum_{t=0}^T r(s_t, a_t) \right]$$

optimizing total reward over the entire **meta**-episode with RNN policy **automatically** learns to explore!

Meta-RL with recurrent policies

$$\theta^* = \arg \max_{\theta} \sum_{i=1}^n E_{\pi_{\phi_i}(\tau)}[R(\tau)]$$

where $\phi_i = f_{\theta}(\mathcal{M}_i)$

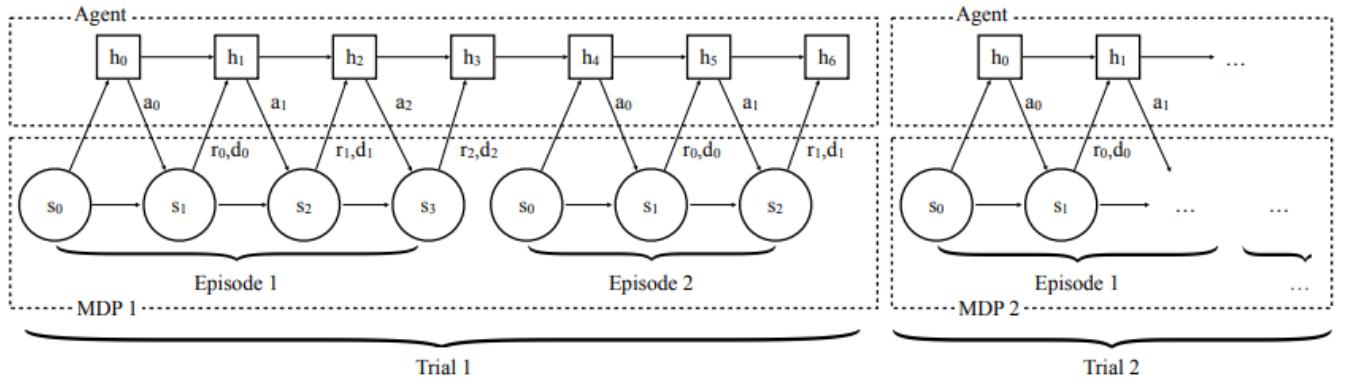


Heess, Hunt, Lillicrap, Silver. **Memory-based control with recurrent neural networks.** 2015.

Wang, Kurth-Nelson, Tirumala, Soyer, Leibo, Munos, Blundell, Kumaran, Botvinick. **Learning to Reinforcement Learning.** 2016.

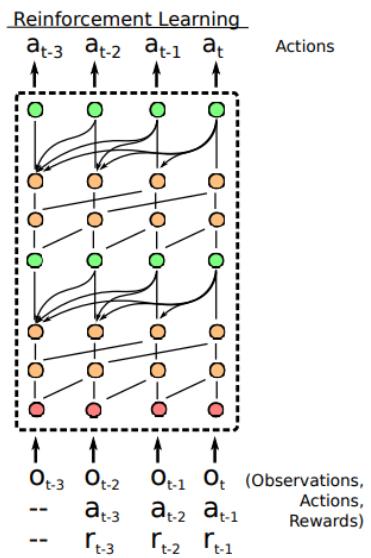
Duan, Schulman, Chen, Bartlett, Sutskever, Abbeel. **RL2: Fast Reinforcement Learning via Slow Reinforcement Learning.** 2016.

Architectures for meta-RL



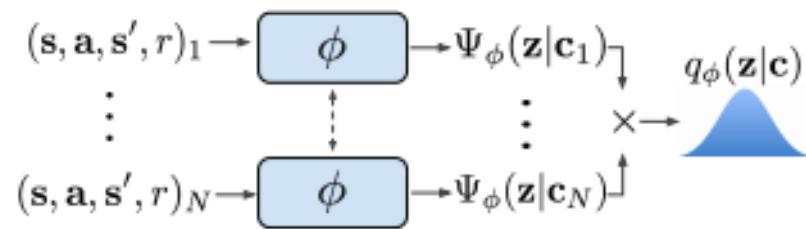
standard RNN (LSTM) architecture

Duan, Schulman, Chen, Bartlett, Sutskever, Abbeel. **RL2: Fast Reinforcement Learning via Slow Reinforcement Learning.** 2016.



attention + temporal convolution

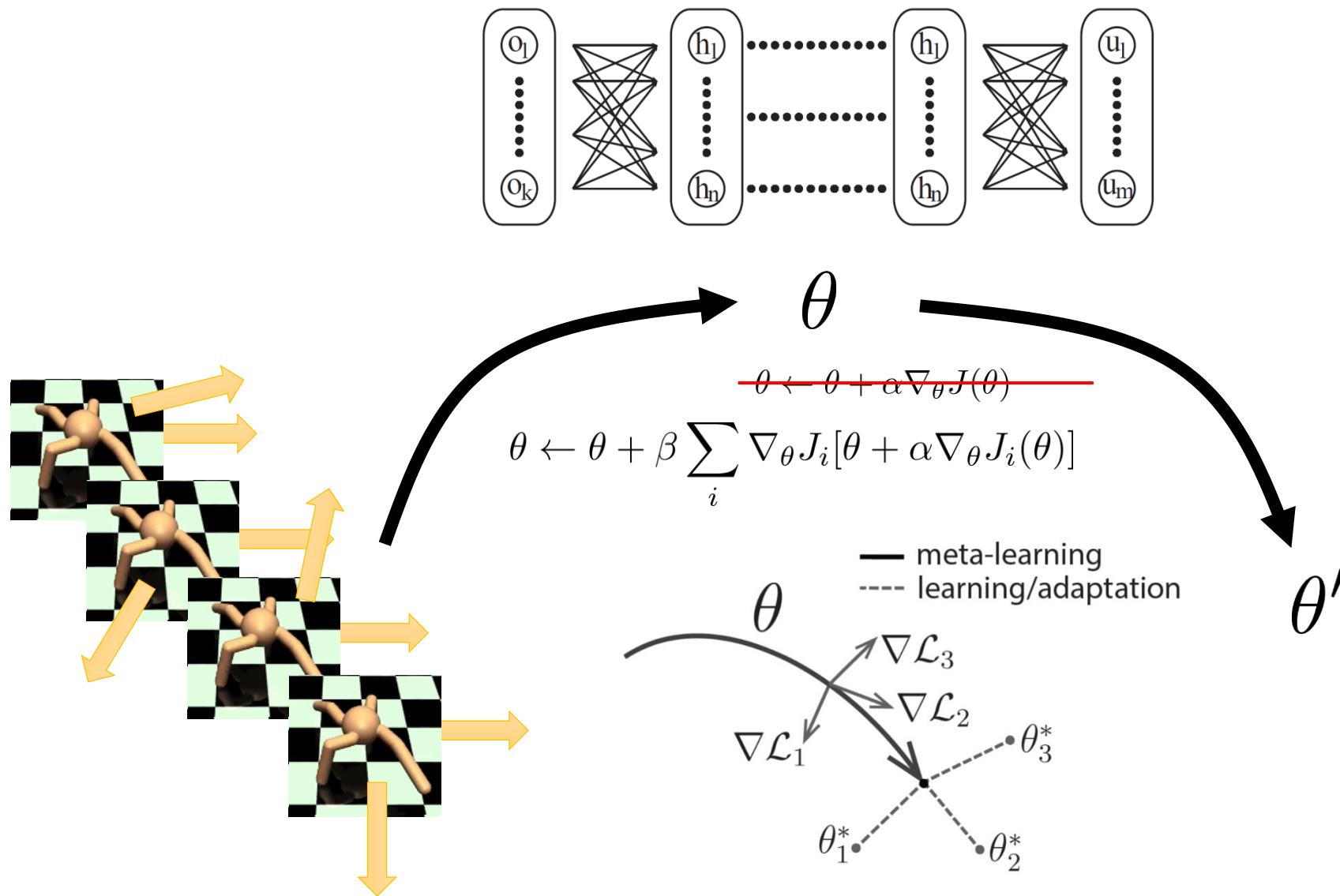
Mishra, Rohaninejad, Chen, Abbeel. **A Simple Neural Attentive Meta-Learner.**



parallel permutation-invariant context encoder

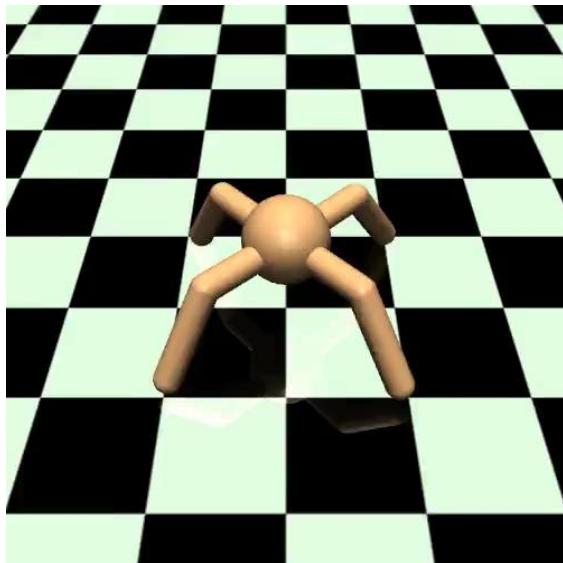
Rakelly*, Zhou*, Quillen, Finn, Levine. **Efficient Off-Policy Meta-Reinforcement learning via Probabilistic Context Variables.**

MAML for RL

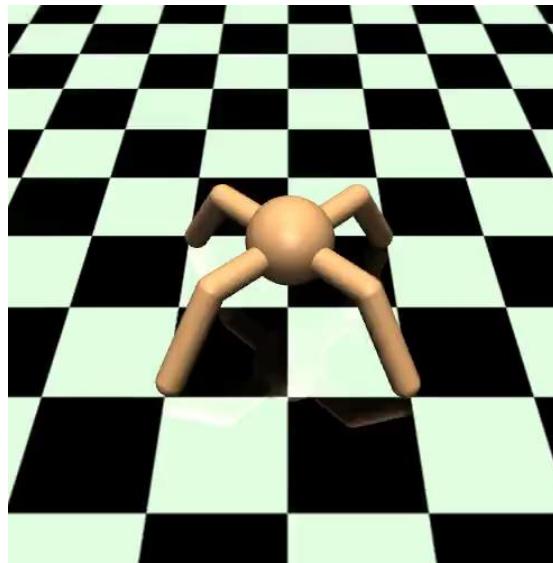


MAML for RL videos

after MAML training



after 1 gradient step
(forward reward)



after 1 gradient step
(backward reward)

