Lecture 9: Hacker's guide to DL

Speaker: Phillip Isola

9. Hacker's guide to DL

- Data
- Model
- Optimization
- Evaluation, Experimentation, and Debugging
- Compute

Disclaimer:

This lecture is my personal opinions and anecdotes!

Acknowledgements:
Lots of slides adapted from Evan Shelhamer's "DIY Deep Learning: Advice on Weaving Nets." Builds on advice from Andrej Karpathy (http://karpathy.github.io/2019/04/25/recipe/), feedback from Isolab members and MIT community, slides from Dylan Hadfield-Menell, twitter feedback (https://twitter.com/phillip_isola/status/1576965425384263680?s=20&t=3eLg6JBYVSkacUtNlz83pA)

Part of the story of deep learning has been the (temporary) success of hacking over theory

UNDERSTANDING DEEP LEARNING REQUIRES RETHINKING GENERALIZATION

Chiyuan Zhang*
Massachusetts Institute of Technology
chiyuan@mit.edu

Samy Bengio
Google Brain
bengio@google.com

Moritz Hardt
Google Brain
mrtz@google.com

Benjamin Recht†
University of California, Berkeley
brecht@berkeley.edu

Oriol Vinyals
Google DeepMind
vinyals@google.com

ABSTRACT

Despite their massive size, successful deep artificial neural networks can exhibit a remarkably small difference between training and test performance. Conventional wisdom attributes small generalization error either to properties of the model family, or to the regularization techniques used during training.

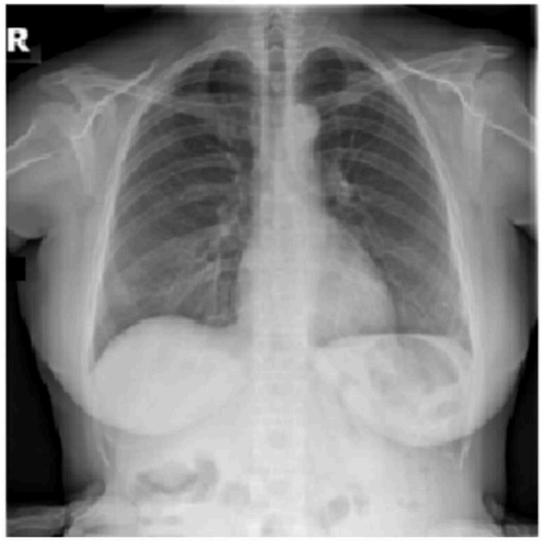
Through extensive systematic experiments, we show how these traditional approaches fail to explain why large neural networks generalize well in practice. Specifically, our experiments establish that state-of-the-art convolutional networks for image classification trained with stochastic gradient methods easily fit a random labeling of the training data. This phenomenon is qualitatively unaffected by explicit regularization, and occurs even if we replace the true images by completely unstructured random noise. We corroborate these experimental findings with a theoretical construction showing that simple depth two neural networks already have perfect finite sample expressivity as soon as the number of parameters exceeds the number of data points as it usually does in practice.

We interpret our experimental findings by comparison with traditional models.

fast.ai—Making neural nets uncool again

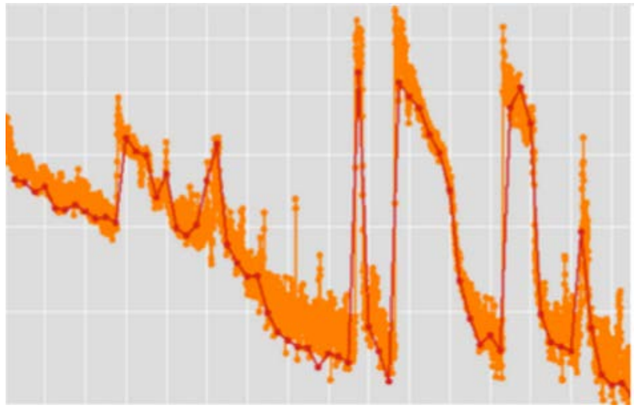
- **Courses**: Practical Deep Learning for Coders; From Deep Learning Foundations to Stable Diffusion
- **Software**: fastai for PyTorch; nbdev
- **Book**: Practical Deep Learning for Coders with fastai and PyTorch
- **In the news**: The Economist; The New York Times; MIT Tech Review
- **Corporate partner program**: Get help with fast.ai technologies & courses from the partner program

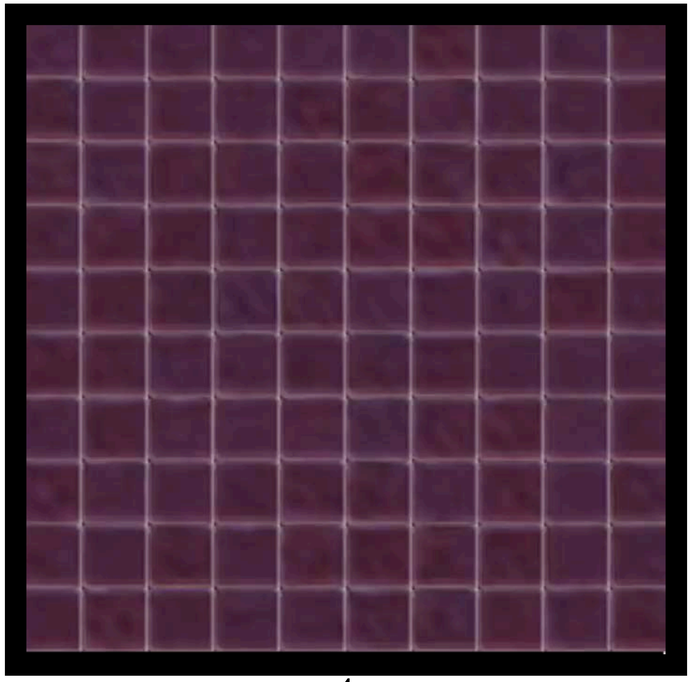
Look at the input

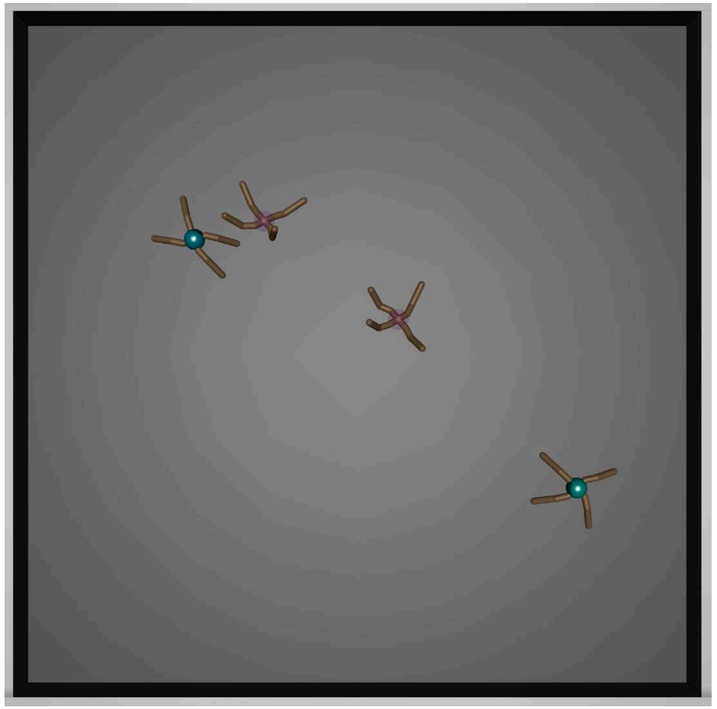


DeGrave, Janizek, Lee, 2020

Look at the output







Data

Look at the data!

inspect the distribution of inputs and targets

- inspect random selection of inputs and targets to have a general sense
- histogram input dimensions to see range and variability
- histogram targets to see range and imbalance
- select, sort, and inspect by type of target or whatever else

[slide adapted from Evan Shelhamer]

Data

Inspect the inliers, outliers, and neighbors:

- visualize distribution and data, especially **outliers**, to uncover dataset issues
- look at **nearest neighbors**
- examples:
  - rare grayscale images in color dataset, huge images that should have been rescaled, corrupted class labels that had been cast to uint8

[slide adapted from Evan Shelhamer]

Data

pre-processing: the data as it is loaded is not always the data as it is stored!

- inspect the data as it is given to the model by `output = model(data)`

original

DeCAF

Caffe

[slide adapted from Evan Shelhamer]

Data

Most important function in deep learning:

```
def inspect_data(X):
  print('type:', X.type())
  print('shape:', X.shape)
  print('requires grad:', X.requires_grad)
  print('numerical range: [{:.2f}, {:.2f}]'.format(X.min(), X.max()))
  print('mean and var: {:.2f}, {:.2f}'.format(X.mean(), X.var()))
```

A library that gives this kind of info by default: https://github.com/xl0/lovely-tensors

Data

pre-processing:

- **standardize**:

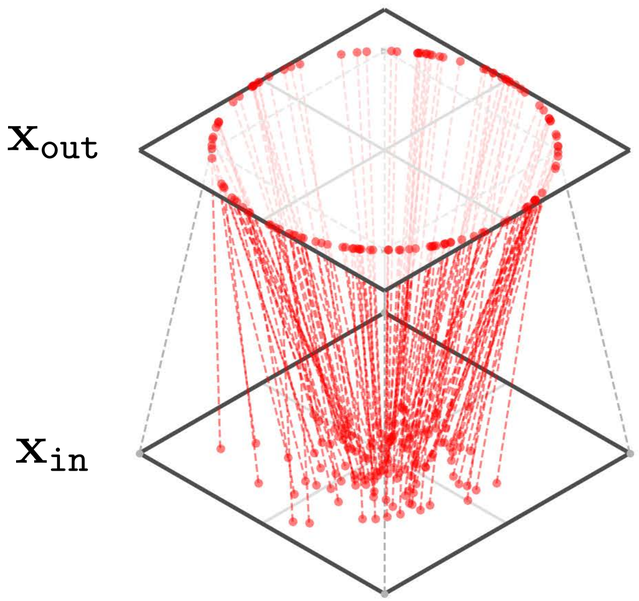
$$x_k \leftarrow \frac{x_k - \mathbb{E}[x_k]}{\sqrt{\texttt{Var}[x_k]}} \quad \forall k$$

  - Squashes all your data dimensions into the same standard range
  - This makes it so that, a priori, no one dimension is valued more than any other
  - Important when different measurements have vastly different scales or units

Beware of low dimensions

RMS-norm

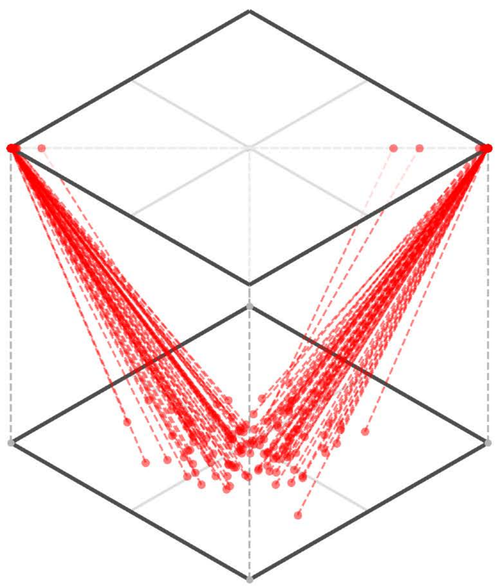
$$x_{\text{out}} = \texttt{RMS-norm}(x_{\text{in}})$$



layernorm

$$x[k] = x_{\text{in}}[k] - \frac{1}{k}\sum_{k} x_{\text{in}}[k]$$

$$x_{\text{out}} = \texttt{RMS-norm}(x)$$



In high dimensions, normalization layers can make entries ~N(0,1), whose typical set is ~surface of hypersphere.

<— Not so in low dimensions.

Many normalization layers behave badly in low dimensions.

—> Avoid low dimensions! All tensor dimensions should be big numbers: [BxNxMxC] data batches, [NxM] weights

Data

pre-processing:

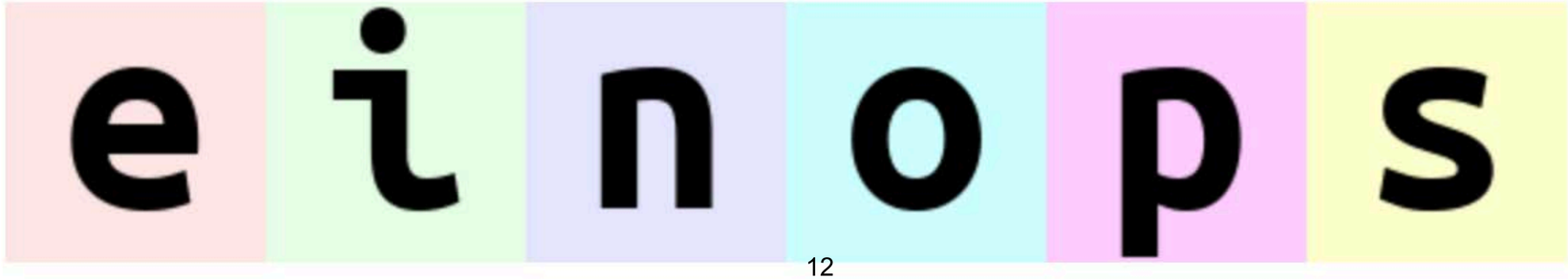
- **summary statistics**: check the min/max and mean/variance to catch mistakes like loading values in the range [0,255] when the model expects values in the range [0,1].
- **shape**: are you certain of each dimension and its size?
  - sanity check with dummy data of prime dimensions: there are no common factors, so mistaken reshaping/flattening/permuting will be more obvious. example: a 64x64x64x64 array can be permuted without knowing
- **type**: check for casting, especially to lower precision
  - what's -1 for a byte? how does standardization change integer data?

[slide adapted from Evan Shelhamer]

Data

- A lot of your code will just be reshaping tensors
- What does `reshape(X, (X.shape(0)*X.shape(1))` do? Is it column order or row order?
- Tools like **einops** can make it much easier to avoid mistakes
  - https://github.com/arogozhnikov/einops/tree/master/docs

```
# or compose a new dimension of batch and width
rearrange(ims, 'b h w c -> h (b w) c')
```



Data

Training data

$\mathbf{x}$ y

 , "Fish"

 , "Grizzly"

 , "Chameleon"

⋮

Image © source unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see https://ocw.mit.edu/help/faq-fair-use/

Data augmentation

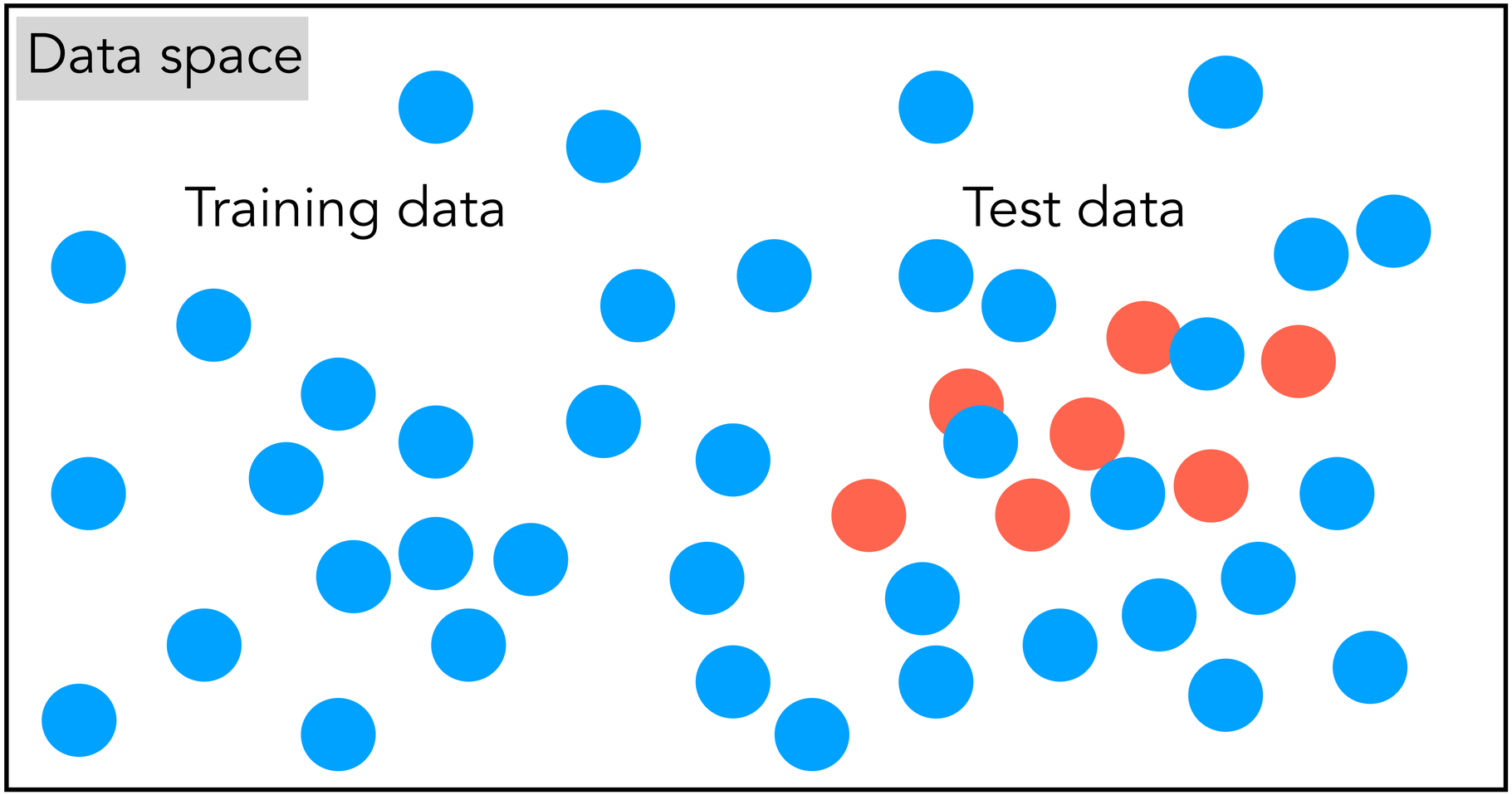
 "Fish" — Mirror

 "Fish" — Crop

 "Fish" — Crop

 "Fish" — Darken

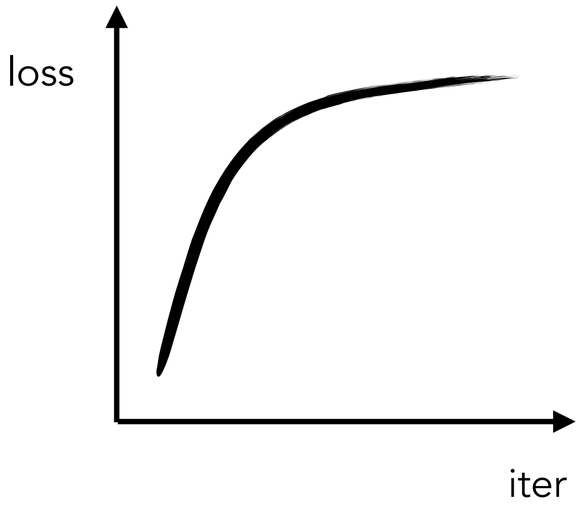
Idea: Train on randomly perturbed data, so that test set just looks like another random perturbation



This is called **domain randomization** or **data augmentation**

What does a good training curve look like?

Bad!
Your data is too easy



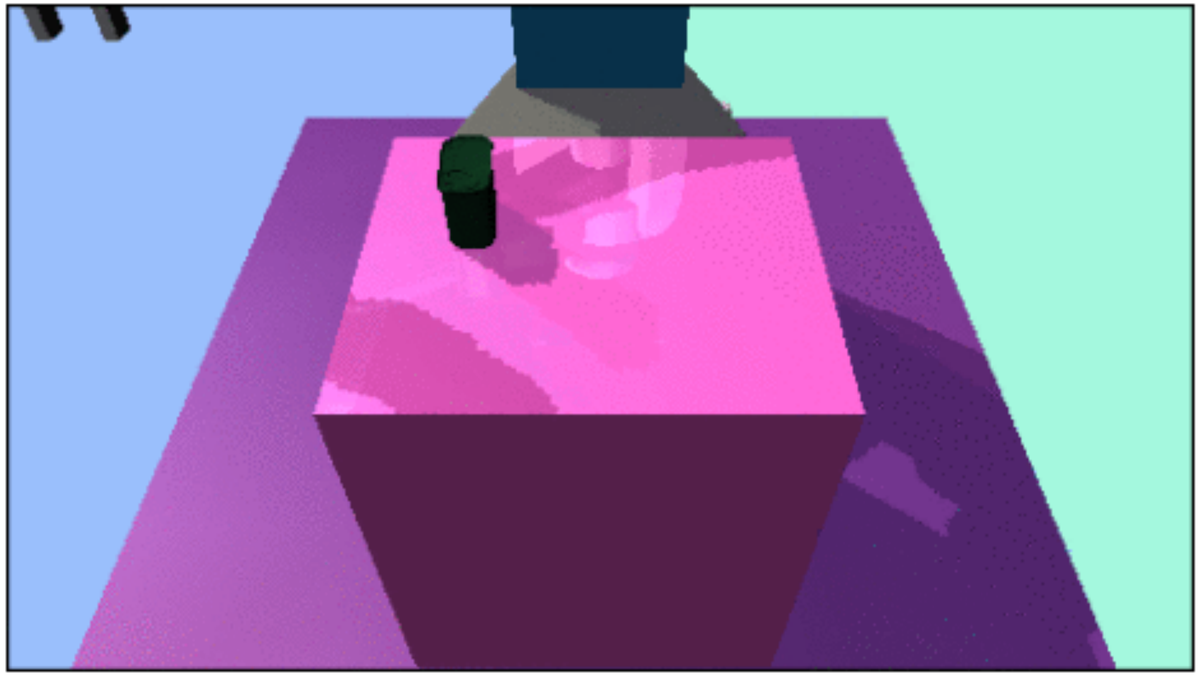
Bad!
You aren't fitting your data

Good
Fitting a hard problem

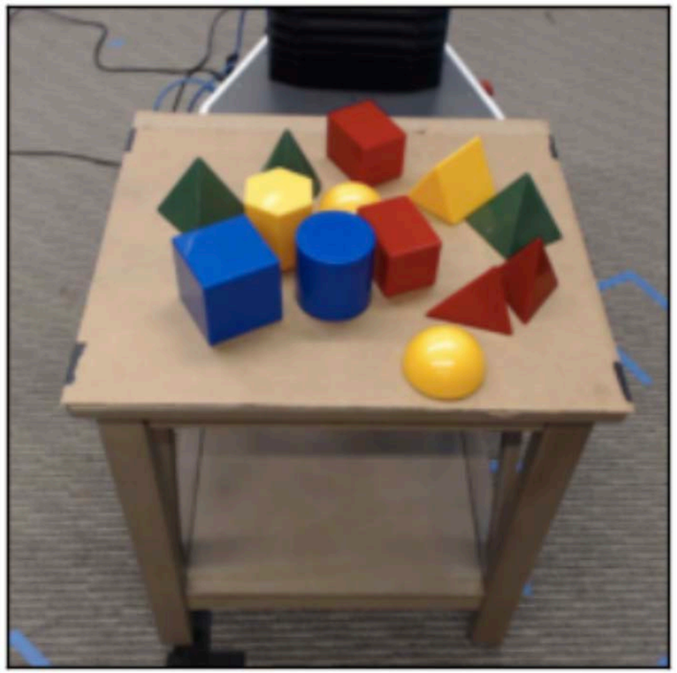
You roughly want to select data and parameters as: `max_data min_params loss(data, params)`

Domain randomization

Training data



Test data



[Sadeghi & Levine 2016]

Above example is from [Tobin, Fong, Ray et al. 2017]

source domain

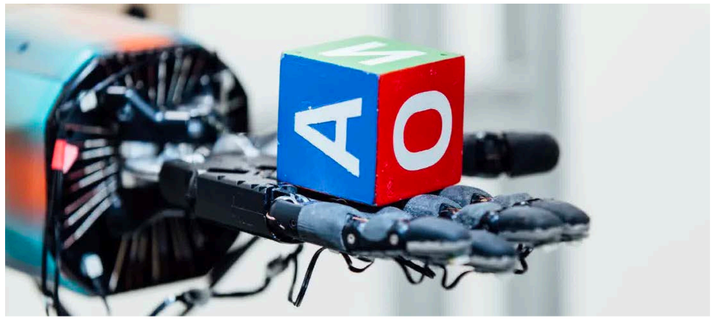
target domain

(where we actual use our model)

Domain gap between $p_{\texttt{source}}$ and $p_{\texttt{target}}$ will cause us to fail to generalize.

Space of images

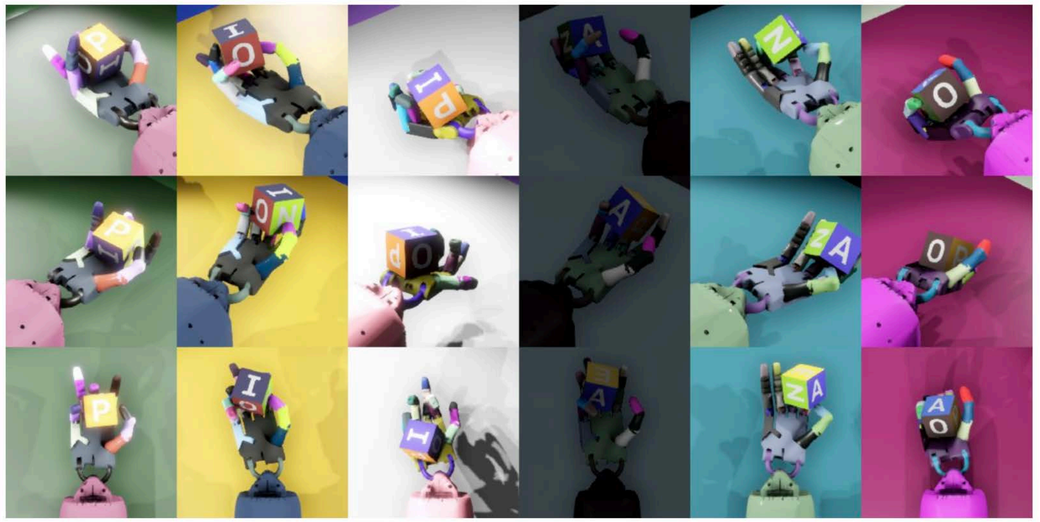
Source data

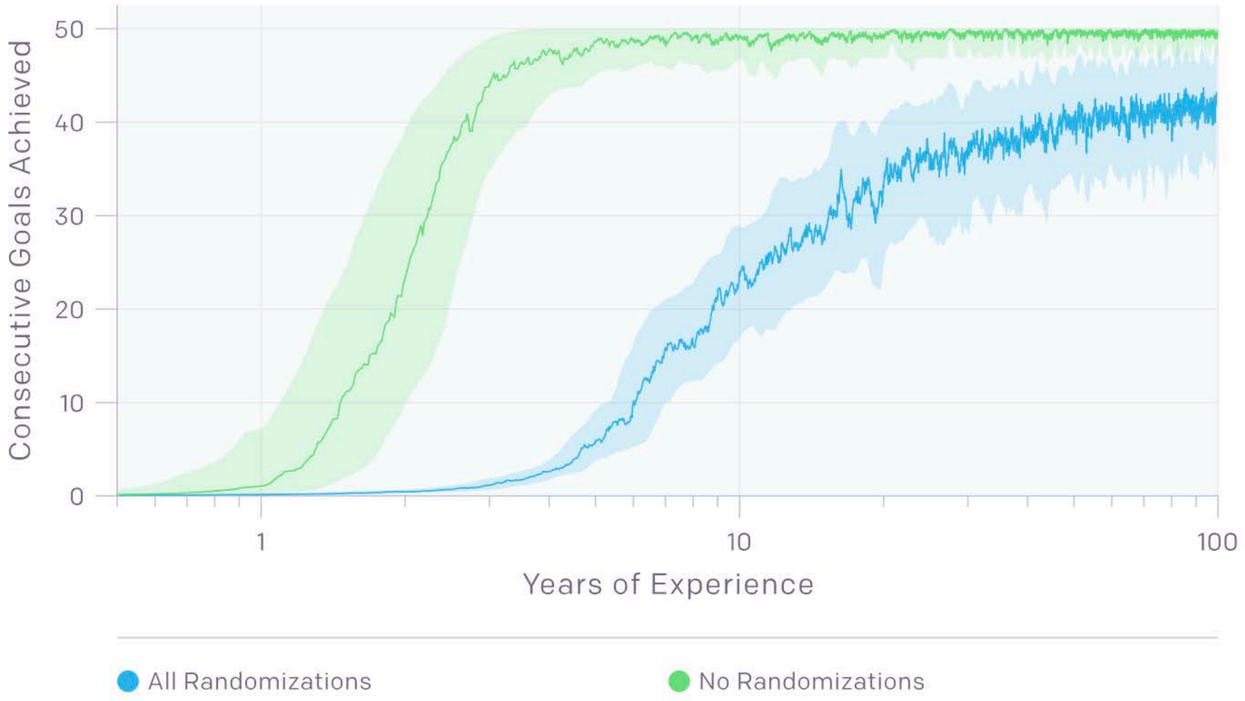


Target data

Table 1: Ranges of physics parameter randomizations.

Parameter	Scaling factor range	Additive term range
object dimensions	uniform([0.95, 1.05])	
object and robot link masses	uniform([0.5, 1.5])	
surface friction coefficients	uniform([0.7, 1.3])	
robot joint damping coefficients	loguniform([0.3, 3.0])	
actuator force gains (P term)	loguniform([0.75, 1.5])	
joint limits		$\mathcal{N}(0, 0.15)$ rad
gravity vector (each coordinate)		$\mathcal{N}(0, 0.4)$ m/s^2

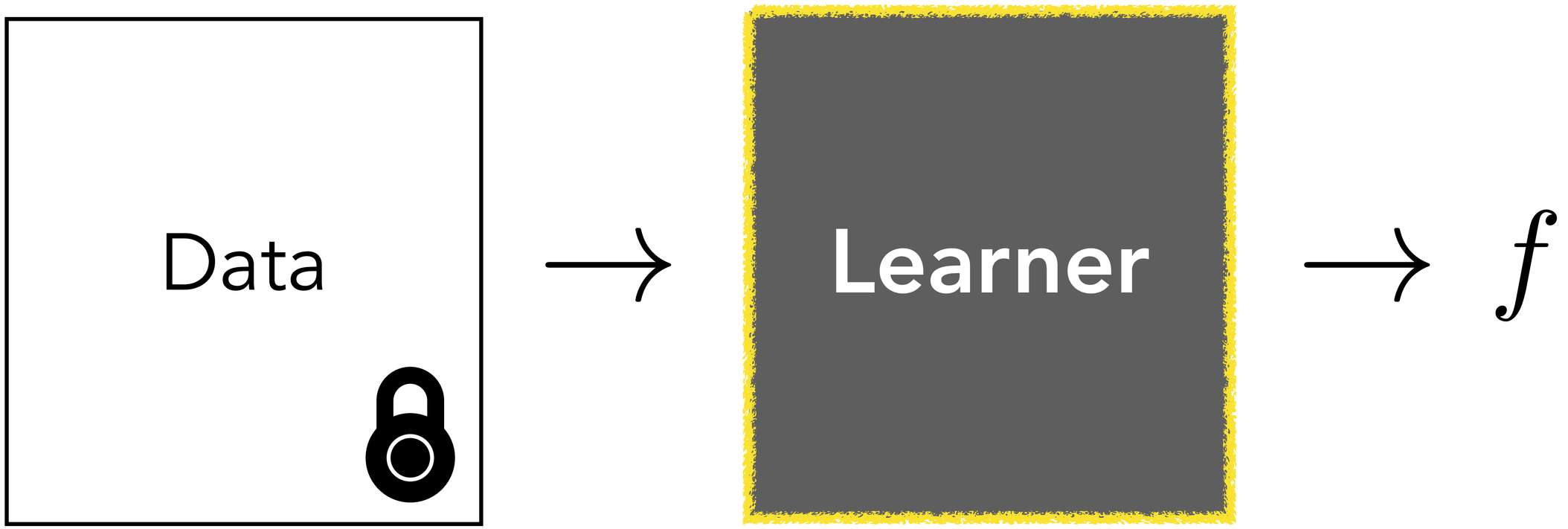




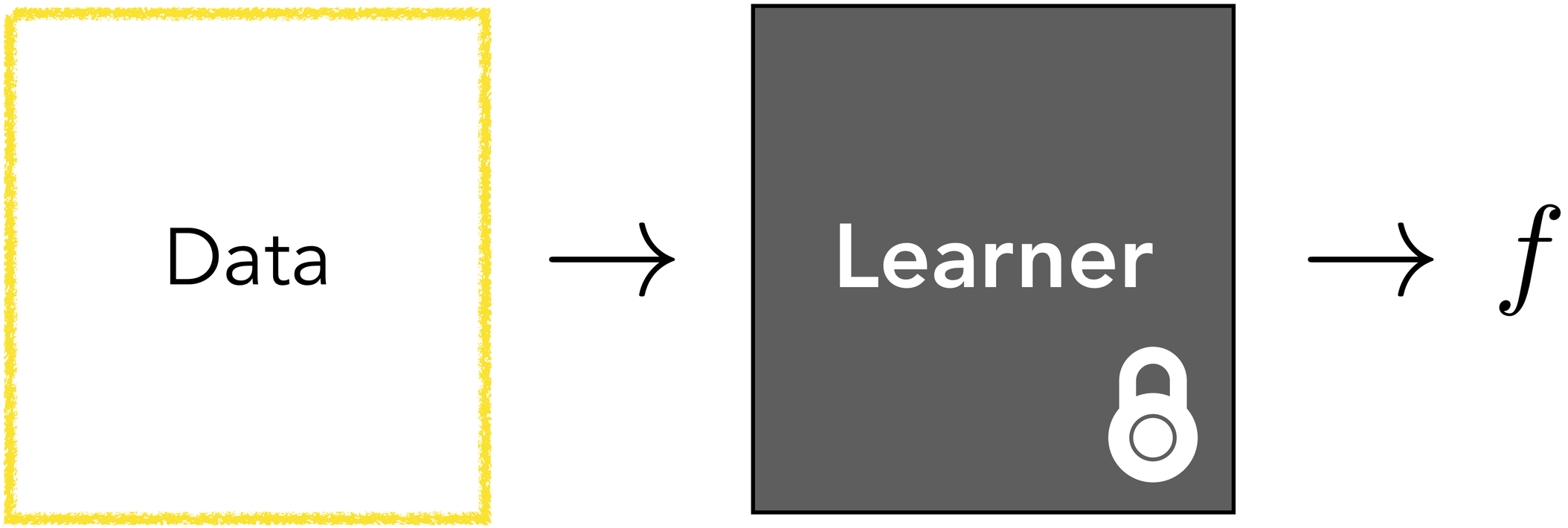
- High train accuracy can mean problem is too easy
- Add more data to make problem harder

[https://openai.com/blog/learning-dexterity/]

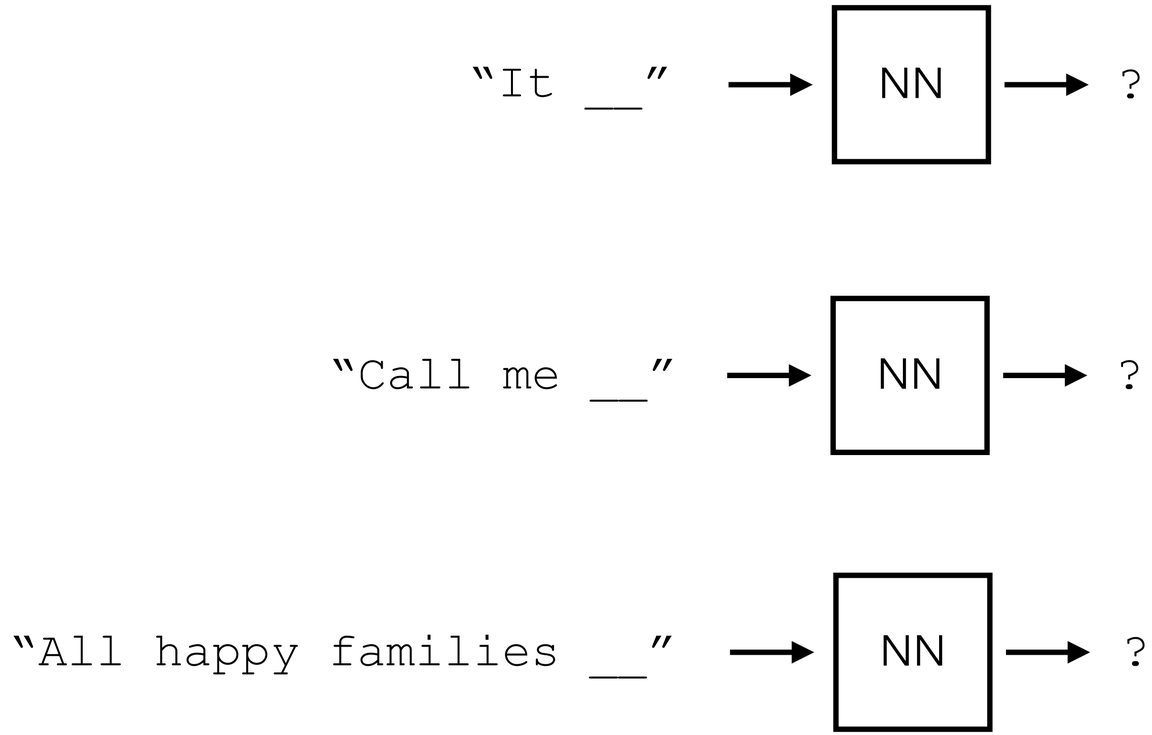
In the academy we typically take data as fixed, and design models that learn from it



In industry, it's usually then other way around. We use a standard learning algorithm, and get to collect data to instruct it



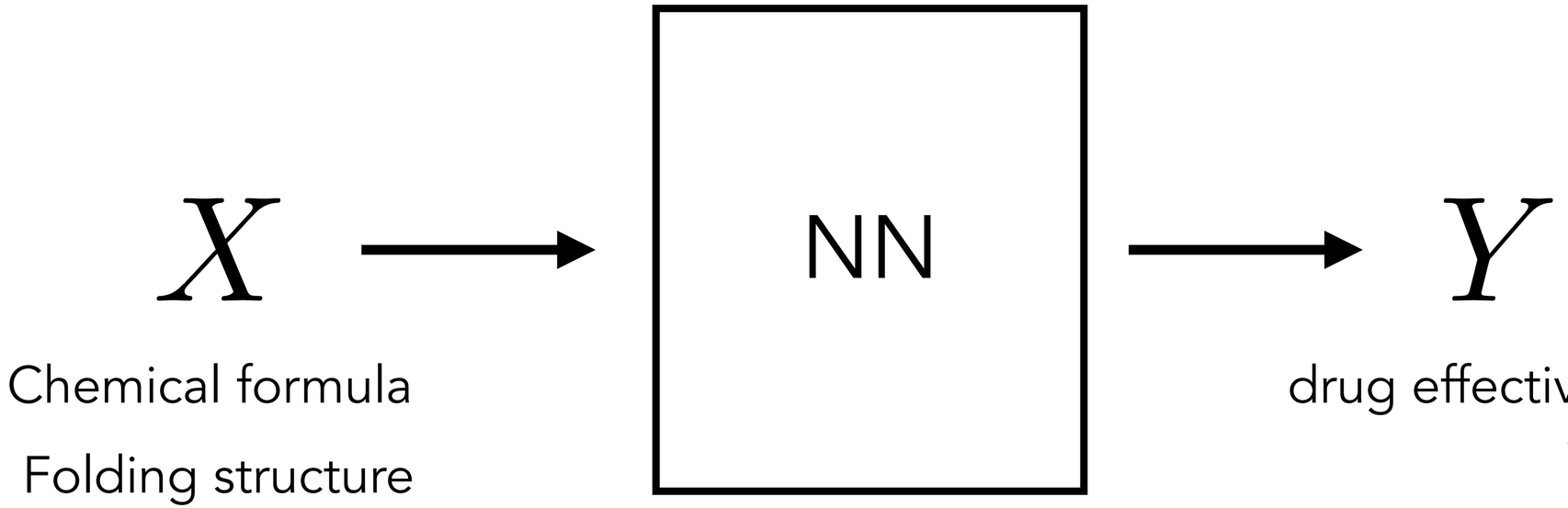
Which is the hardest prediction problem?



Prediction gets *easier* the longer the input!

You can change your data to make the learning work better!

Suppose you are designing a pharmaceutical drug



Chemical formula

Folding structure

Patient age

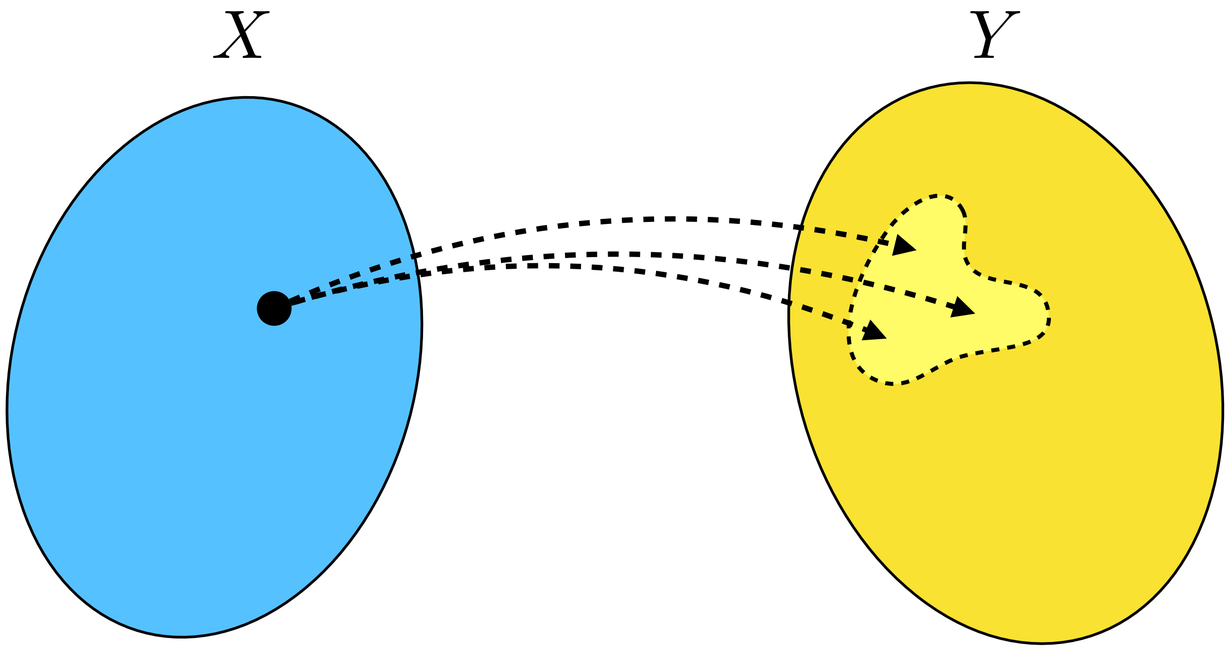
Patient biopsy

… the universe

drug effectiveness

We will treat our prediction targets as fixed (given)

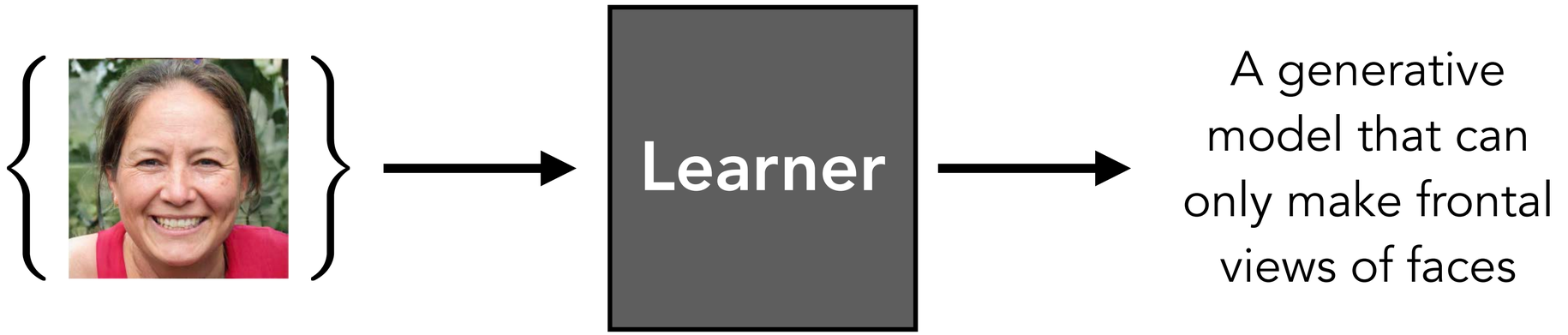
Adding info to X reduces uncertainty over Y



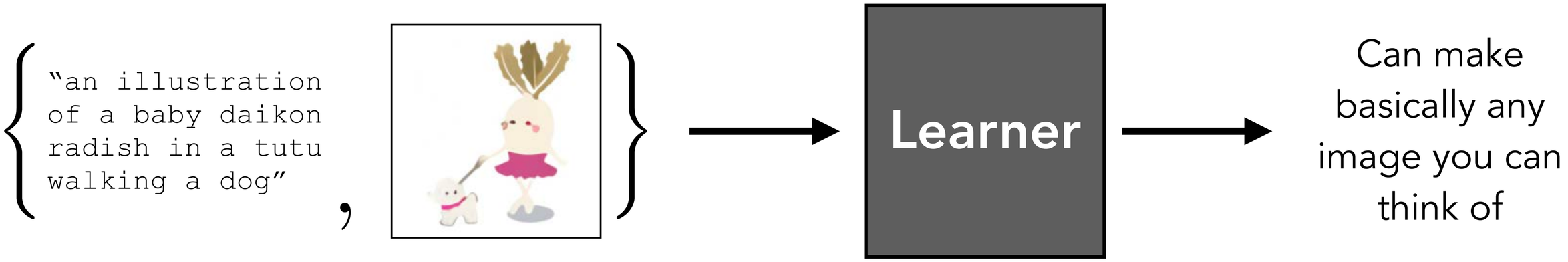
- It's really hard to model a complicated *distribution*, P(Y|X), over all the possible values of Y for some given observation X (we will get to this in the generative modeling lectures).
- Standard NN regression outputs a single point prediction for each X.
- The hack is to put so much info in X that P(Y|X) looks like a single point!

Evolution of image generation

2019: StyleGAN2



2021: DALL-E



You can change your data to make the learning work better!

- Use **big** data
  - Big as in lots of {x,y} training pairs
  - Big as in x is a big object, replete with information (+ high-dimensional)
  - (Big as in y is a big object too)

Model

Keep it as simple as possible!

do your first experiment with the simplest possible model w/ and w/o your idea

Why keep it simple?

- easy to build, debug, share
- tractable to understand, make robust, build theories around
- *simple models also work better* (Occam's razor, Solomonoff Induction)
- if you focus on simplicity you will have an unfair advantage

Model

start with a standard and popular model (popularity matters more than performance)

if you have an image classification problem, you might try:

```
model = torch.hub.load('pytorch/vision:v0.9.0', 'resnet18')
```

└ https://pytorch.org/hub/pytorch_vision_resnet/ 

if you have text problem, you might try:

```
>>> from transformers import pipeline, set_seed
>>> generator = pipeline('text-generation', model='gpt2')
```

← https://huggingface.co/ 

find popular models and code here: https://paperswithcode.com/

Model

stand on the shoulders of giants

use pretrained models

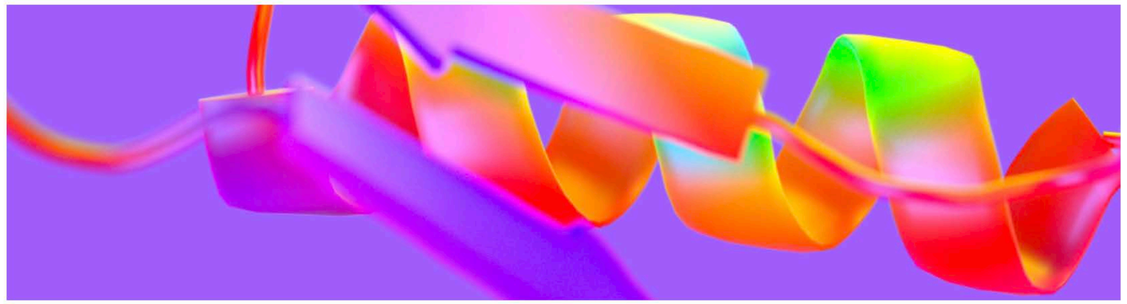
(but be aware of their flaws and limitations)

Stable Diffusion

Stable Diffusion was made possible thanks to a collaboration with Stability AI and Runway and builds upon our previous work:

High-Resolution Image Synthesis with Latent Diffusion Models
Robin Rombach*, Andreas Blattmann*, Dominik Lorenz, Patrick Esser, Björn Ommer
CVPR '22 Oral | GitHub | arXiv | Project page

README.md



AlphaFold

This package provides an implementation of the inference pipeline of AlphaFold v2.0. This is a completely new model that was entered in CASP14 and published in Nature. For simplicity, we refer to this model as AlphaFold throughout the rest of this document.

Model

Transform your problem into a “solved” problem

- Case study: transforming image *colorization* to image *classification*

[c.f. the strategy of “polynomial-time reduction”]

Image colorization

Input $\mathbf{x}$

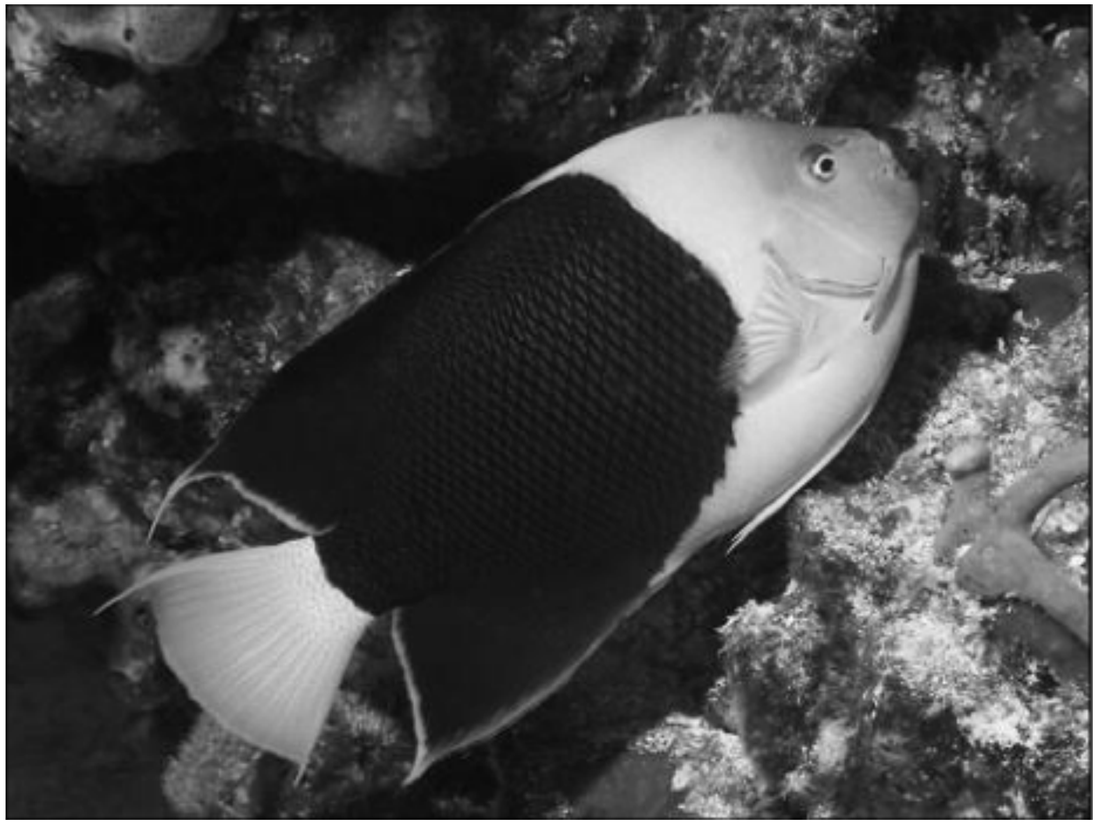
Output $\mathbf{y}$

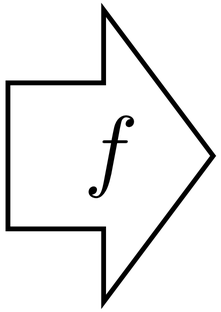
Training data

$\mathbf{x}$ $\mathbf{y}$

$$\arg\min_{f \in \mathcal{F}} \sum_{i=1}^{N} \mathcal{L}\left(f\left(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\right)\right.$$

[Zhang, Isola, Efros, ECCV 2016]





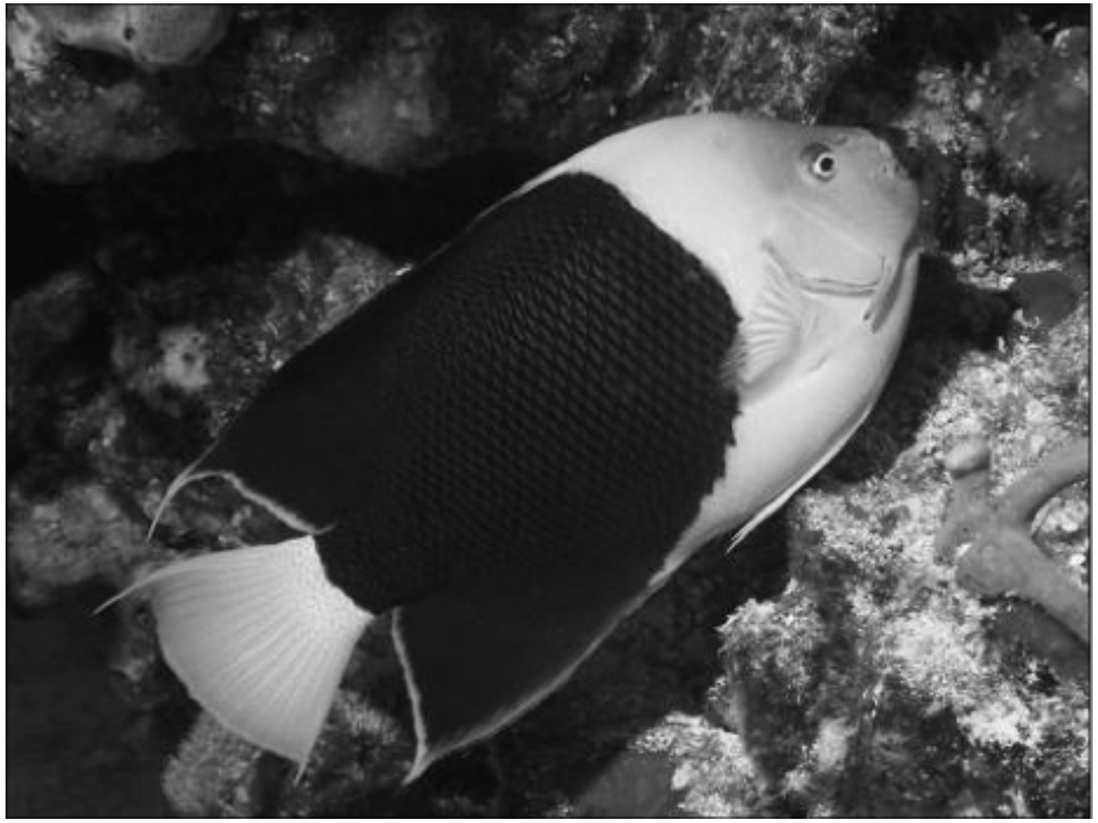
Grayscale image: **L channel**

$$\mathbf{x} \in \mathbb{R}^{H \times W \times 1}$$

Color information: **ab channels**

$$\mathbf{y} \in \mathbb{R}^{H \times W \times 2}$$

[Zhang, Isola, Efros, ECCV 2016]



f

Grayscale image: **L channel**

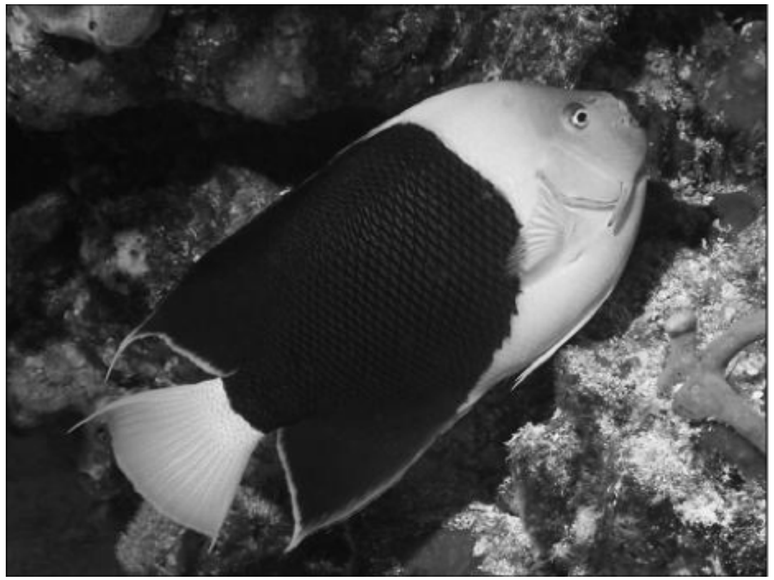
$$\mathbf{x} \in \mathbb{R}^{H \times W \times 1}$$

Color information: **ab channels**

$$\mathbf{y} \in \mathbb{R}^{H \times W \times 2}$$

[Zhang, Isola, Efros, ECCV 2016]

Colorization → Classification



f

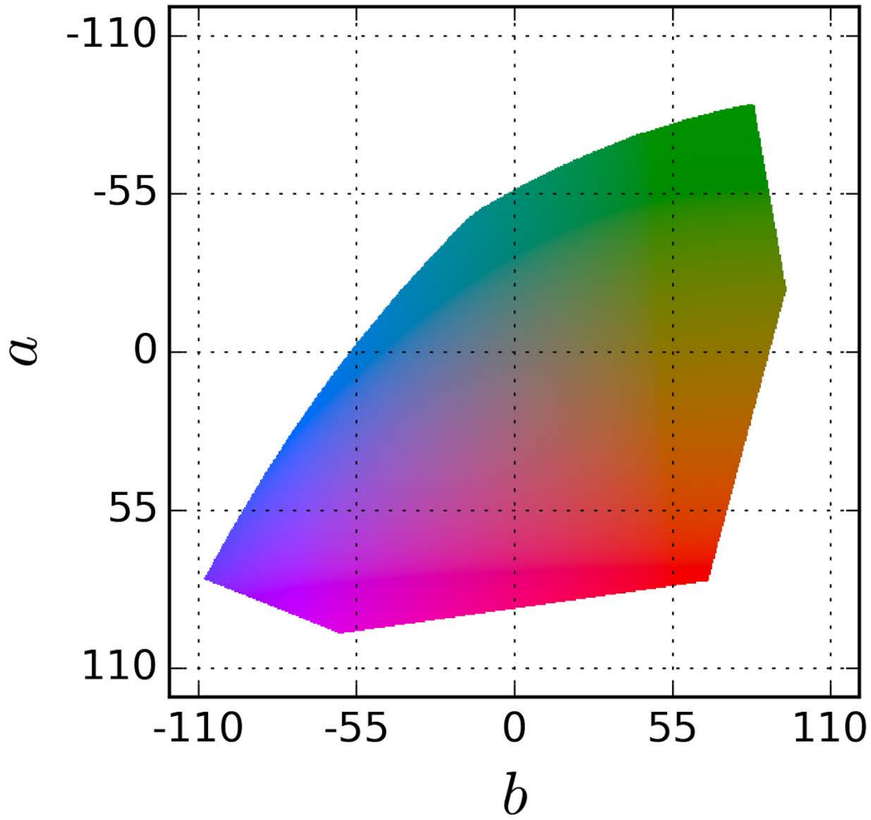
yellow

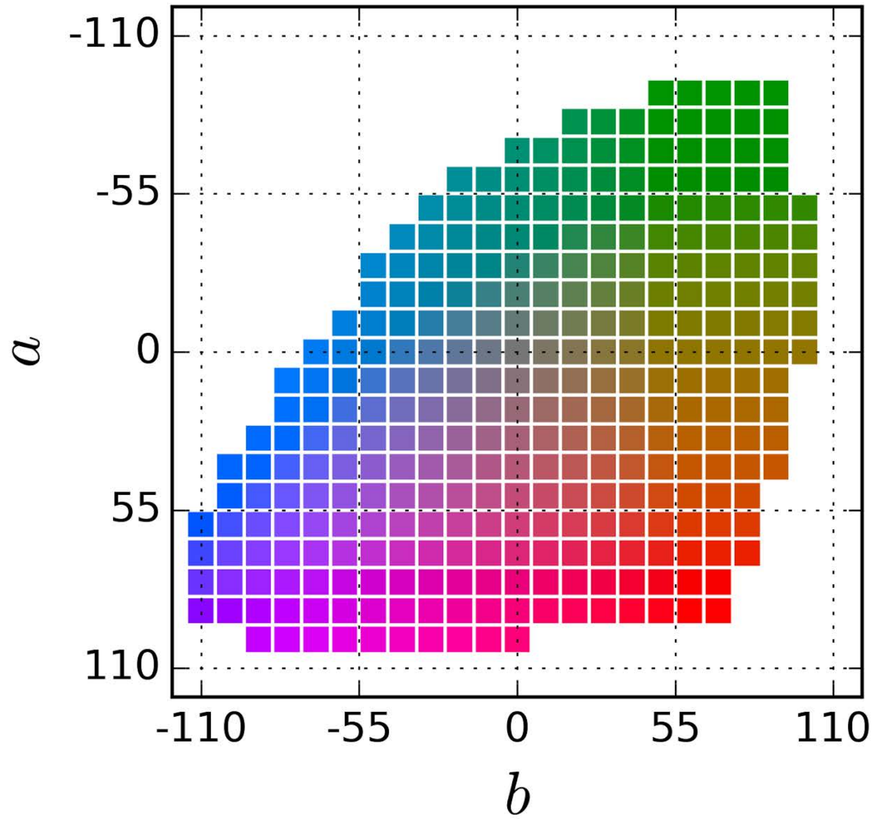
Colors → Classes

one-hot representation of K discrete classes

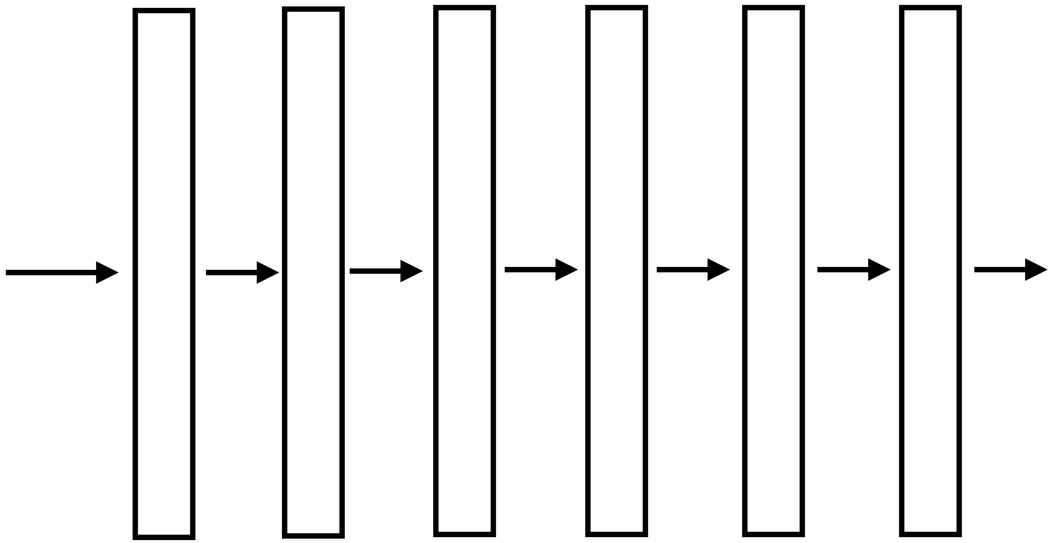
$\mathbf{y} \in \mathbb{R}^{H \times W \times 2}$

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





One hot codes: → [0,0,1, …] → [1,0,0, …] → [0,1,0, …]



rockfish

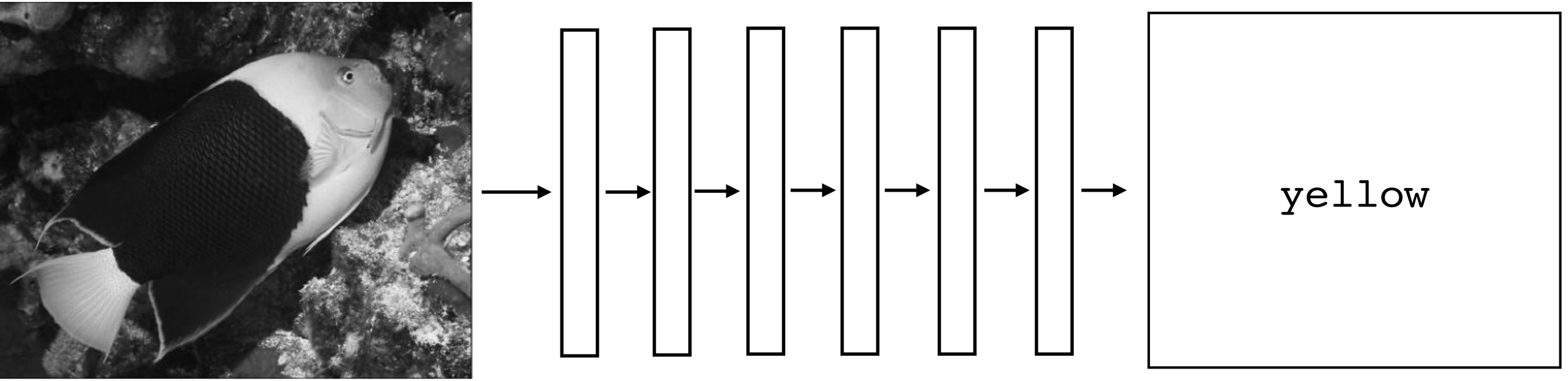


Image classification → Pixel classification

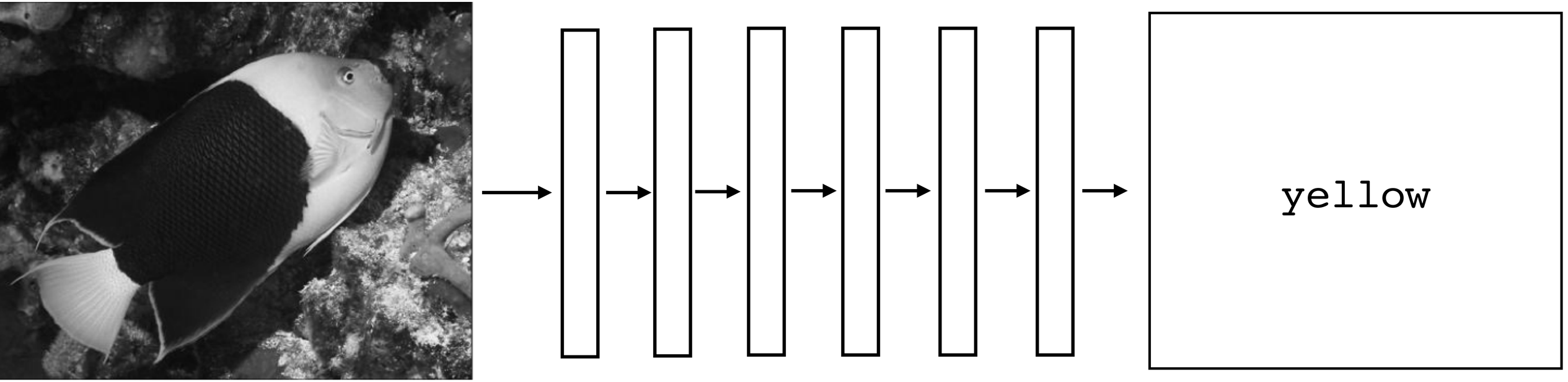


Image classification → Pixel classification

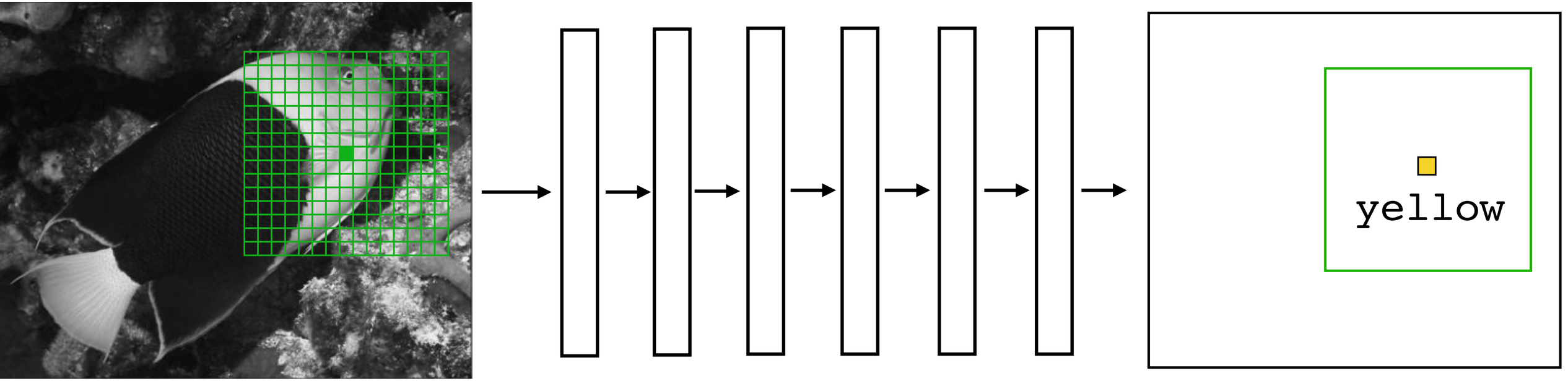
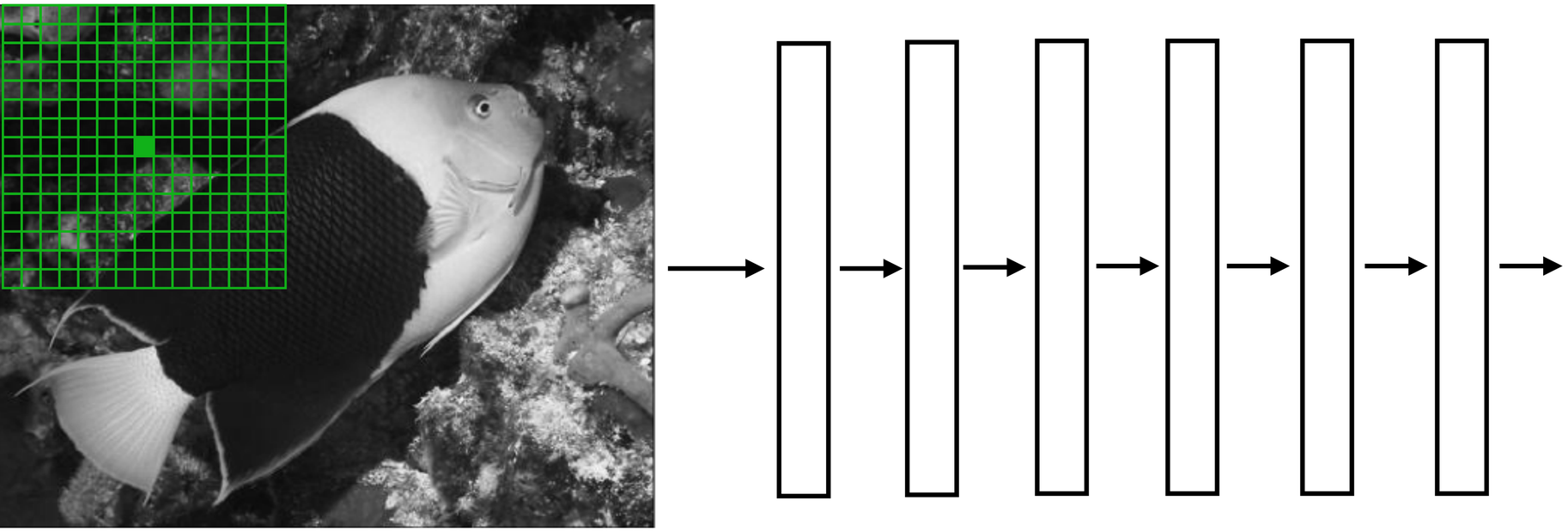


Image classification → Pixel classification



Model

- Formulate your problem as **softmax regression** (a.k.a. classification)
  - cross-entropy loss, 1-hot labels
  - Why?
    1. No restriction on shape of predictive distribution [up to quantization] (this is *not* the case for least-squares regression, which assumes Gaussian predictions)
    2. Discrete classes are easy to label
    3. All labels are equidistant under 1-hot representation

Model

good default choices ca 2024

Recipe for deep learning in a new domain

1. Transform your data into numbers (one-hot vectors)

2. Transform your goal into an numerical measure (cross-entropy loss)

3. Use a generic optimizer (Adam) and an standard architecture (transformer) to solve the learning problem

Model

good default choices ca 2024

Don’t use batch norm

- Introduces a strong dependency on batch size (now batch size becomes an even more critical hyperparameter)
- Different behavior at train and test time
- Makes distributed computing hard — requires communication between all elements in a batch
- Use layer norm instead

Longer rant I wrote a few years ago:

(Disclaimer that this is my personal opinion)

Batchnorm *can* be a useful tool, but in my experience it's usually more trouble than it's worth. Below are some things that make working with batchnorm a headache. You can work around all these issues… or you can just not use batchnorm 😃.

1. Behavior at train time and test time is different. Forgot to set model.eval()? You will have a bug. More generally, differences in train vs test behavior make it harder to anticipate test behavior from training behavior. You might be in for surprises.
2. It only works if batch size is sufficiently big. Suppose your batch size is 1. Then if you have an activation vector $\mathbf{z}$ and subtract the mean over the batch, you get $\mathbf{z} - \mathbf{z} = 0$. The model just zeroed out the activation vector and your net won't work. Worse, the variance is undefined for batch size 1 and that could cause bugs too. For small batch size the variance could be a very poor estimate of the true variance of the activations and cause numerical and optimization issues. [I had this bug in the original version of the pix2pix paper, and it made the baseline work worse than it should have (which might not have been a bad thing for making the paper popular…). See change log in appendix here: https://arxiv.org/abs/1611.07004]
3. Suppose your batch size is large, but all activations in a batch happen to have the same value. Again the variance is undefined and the activations get zeroed out by subtracting the mean. [This was the "bug" that the SPADE paper tried to fix: https://arxiv.org/abs/1903.07291]
4. Want to train a really big model in parallel across many machines? With SGD and no batchnorm you can run forward and back propagation independently on subbatches distributed across the machines, then only communicate between them once to aggregate the gradients. With batchnorm, the activations for one subbatch will depend on the activations for another subbatch – you need to communicate all these activations between the machines. Naively you need to do this *after each layer of your network*, to compute the inputs to the next layer, which is a huge communication overhead, but in practice people use various (bug-prone) speedups.
5. More generally, batchnorm introduces a dependency between different elements of your batch. Different training examples are not processed iid. This not only gives implementation headaches but also makes the theoretical analysis harder. This might be one reason why the theory of why batchnorm works is still not really resolved.
6. There are lots of tricks and hacks to fix the above issues – these tricks add more potential bugs and complexity. Sometimes the fix is worse than the original problem. In distributed training sometimes people run batchnorm independently on each machine. That fixes the communication problem, but now you have a maybe worse problem: your results change dramatically depending on how many machines are in your cluster (since the number of machines determines the size of the subbatches when you distribute across machines and subbatch size has a big effect on how well batchnorm works).

Feel free to add to this list if anyone has more – I think the list could go on and on.

Model

Remember that often the easiest way to get better performance is:

1) **Scale** your data: more (diverse) training examples

2) **Scale** your model: more layers, more channels

3) **Scale** your compute: train for longer

In the current era, I would say these are the top three factors that determine success

… but working at small scale forces efficiency, and *then* when you do scale up, you get more bang for your buck

Model

Once you get your system working, you are only halfway done

Second half is to remove everything nonessential

“Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away”
— Antoine de Saint Exupéry

Copilots

Colab AI

Good for boilerplate code, visualization, getting syntax right. Ever improving.

Don’t use it for your psets but you can use it for your final projects. You *should* learn how to use these tools effectively.

Think first, then ask an LLM for help.

Don’t trust the code without verification.

Copilots

General advice: the more documentation you provide, the better the completion will be.

PH

```
def backward_D_basic(self, netD, real, fake):
    """Calculate GAN loss for the discriminator

    Parameters:
      netD (network)     -- the discriminator D
      real (tensor array) -- real images
      fake (tensor array) -- images generated by a generator

    Return the discriminator loss.
    We also call loss_D.backward() to calculate the gradients.
    """
```

python Copy code

```python
import torch

def backward_D_basic(netD, real, fake):
    # Calculate the discriminator's predictions for real and fake images
    real_pred = netD(real)
    fake_pred = netD(fake)

    # Create labels for real and fake images (1 for real, 0 for fake)
    real_labels = torch.ones_like(real_pred)
    fake_labels = torch.zeros_like(fake_pred)

    # Calculate the binary cross-entropy loss for real and fake images
    criterion = torch.nn.BCELoss()

    real_loss = criterion(real_pred, real_labels)
    fake_loss = criterion(fake_pred, fake_labels)

    # Calculate the total discriminator loss
    loss_D = real_loss + fake_loss

    # Backpropagate to compute gradients
    loss_D.backward()

    return loss_D
```

Regenera

Optimization

figure out optimization on **one/few/many datapoints**, in that order

- overfit to a data point
- then fit a batch
- and finally try fitting the dataset (or a miniature version of it)

first make sure you can fit train set, then consider generalization to test set

[slide adapted from Evan Shelhamer]

Optimization

sanity check the loss against a suitable reference value

- classification with cross-entropy loss: uniform distribution
  - get to know log loss numbers:

    -0.69 = ln(0.5) [chance on binary classification]

    -2.3 = ln(0.1) [chance on 10-way classification]

- regression with squared loss: mean of targets (or even just zero)

and if your loss is constant, double check for zero initialization of the weights

[slide adapted from Evan Shelhamer]

Optimization

Most important hyperparameters: **learning rate** and **batch size**

- first, **use a constant rate;** don't schedule until everything else is figured out
- schedule according to number of iterations of SGD, not epochs
- use biggest batch size that will fit in memory
- always retune lr when *anything* changes in your model (most changes to model change scale of gradients, which changes the **effective lr**)

Until Jeremy solves lr-free optimization

may look like your model is training much faster, but really you just scaled the effective lr

Optimization

Be careful with the concept of “epochs”

- There are no epochs in the wild
- Trend toward single-epoch training in LLMs
- Don’t tie lr schedule to epochs
  - makes it hard to compare learning curves between experiments
  - be careful with cosine lr (looks like it is converging when it is not)

Method	Architecture	Param.	Head	Epochs	Top-1	Top-5
InstDis [73]	ResNet-50	24	Linear	200	56.5	-
Local Agg. [83]	ResNet-50	24	Linear	200	58.8	-
CMC [66]	ResNet-50*	12	Linear	240	60.0	82.3
MoCo [28]	ResNet-50	24	Linear	200	60.6	-
PIRL [49]	ResNet-50	24	Linear	800	63.6	-
CPC v2 [31]	ResNet-50	24	-	-	63.8	85.3
SimCLR [10]	ResNet-50	24	MLP	1000	69.3	89.0
InfoMin Aug. (Ours)	ResNet-50	24	MLP	200	70.1	89.4
InfoMin Aug. (Ours)	ResNet-50	24	MLP	800	**73.0**	**91.1**

[Tian, Sun, Poole, et al., 2020]

Optimization

checkpoint features + gradients to trade space for time and fit large models

- can then accumulate gradients across checkpoints
- can resume training if your computer crashes
- have a “paper trail” to debug later

[slide adapted from Evan Shelhamer]

Optimization

live on the edge and try extreme settings (but just a little bit)

- If optimization never diverges, your learning rate is too low

in the style of *Umeshism*

- If you've never missed a flight, you're spending too much time in airports

[slide adapted from Evan Shelhamer]

Optimization

Use exponentially moving averages (EMA) $\theta^t_{\texttt{EMA}}$ $\beta\theta^{t-1}_{\texttt{EMA}} + (1-\beta)\theta^{t-1}$

- Replace a quantity with a weighted average of its previous values, with weight exponentially decaying over time
- Time averages (EMA) can achieve a similar effect as “space” averages (e.g., average gradients over batch)
- Useful for many quantities in deep learning, including gradients (where it is known as momentum), weights, data, activations, targets, etc.
  - (Basically for any variable in DL, try replacing it with its EMA version and it may be better)

Optimization

optimizers:

- Adam (or AdamW) is good for prototyping (generally just works)
- SGD may be slightly better for performance (but requires more tuning of hyperparameters)
- Clip gradients to improve stability

Evaluation

switch to **evaluation mode** by model.eval() (PyTorch)

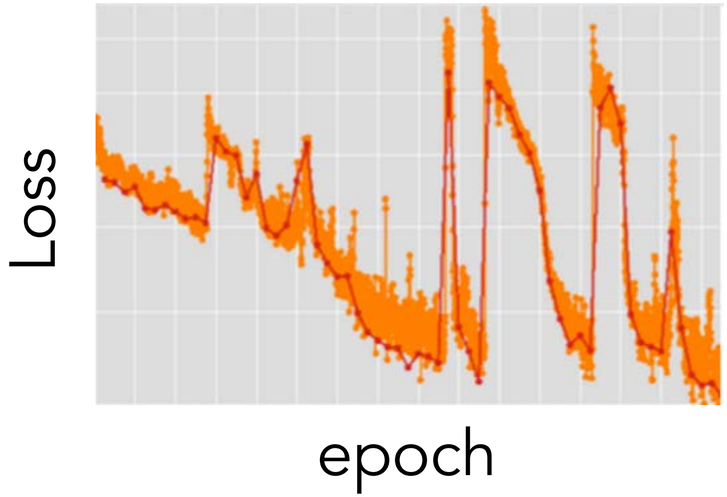
no, really

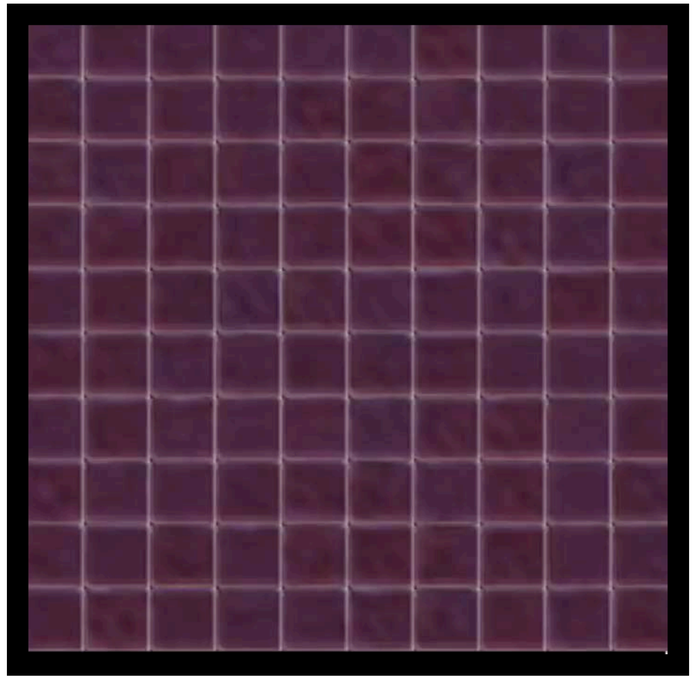
and check the mode by model.training

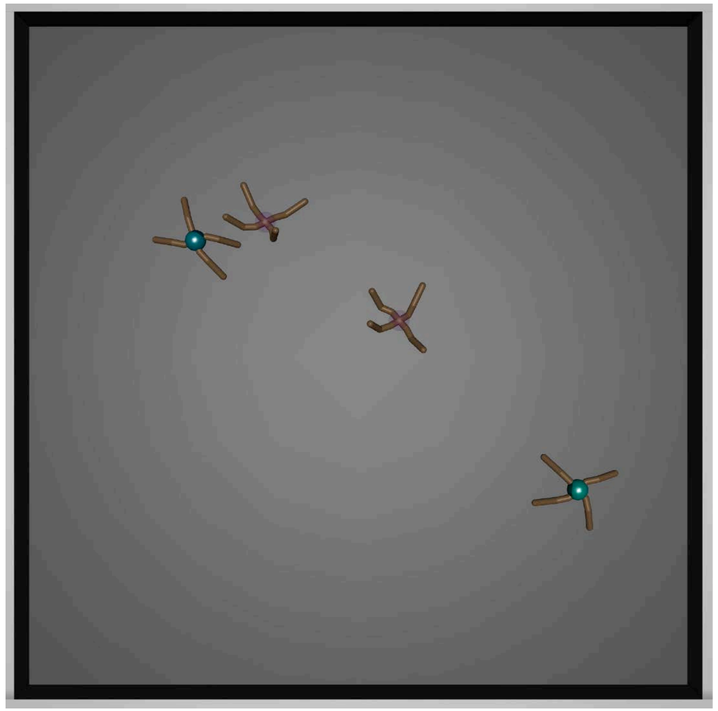
[slide adapted from Evan Shelhamer]

Evaluation

Look at the output





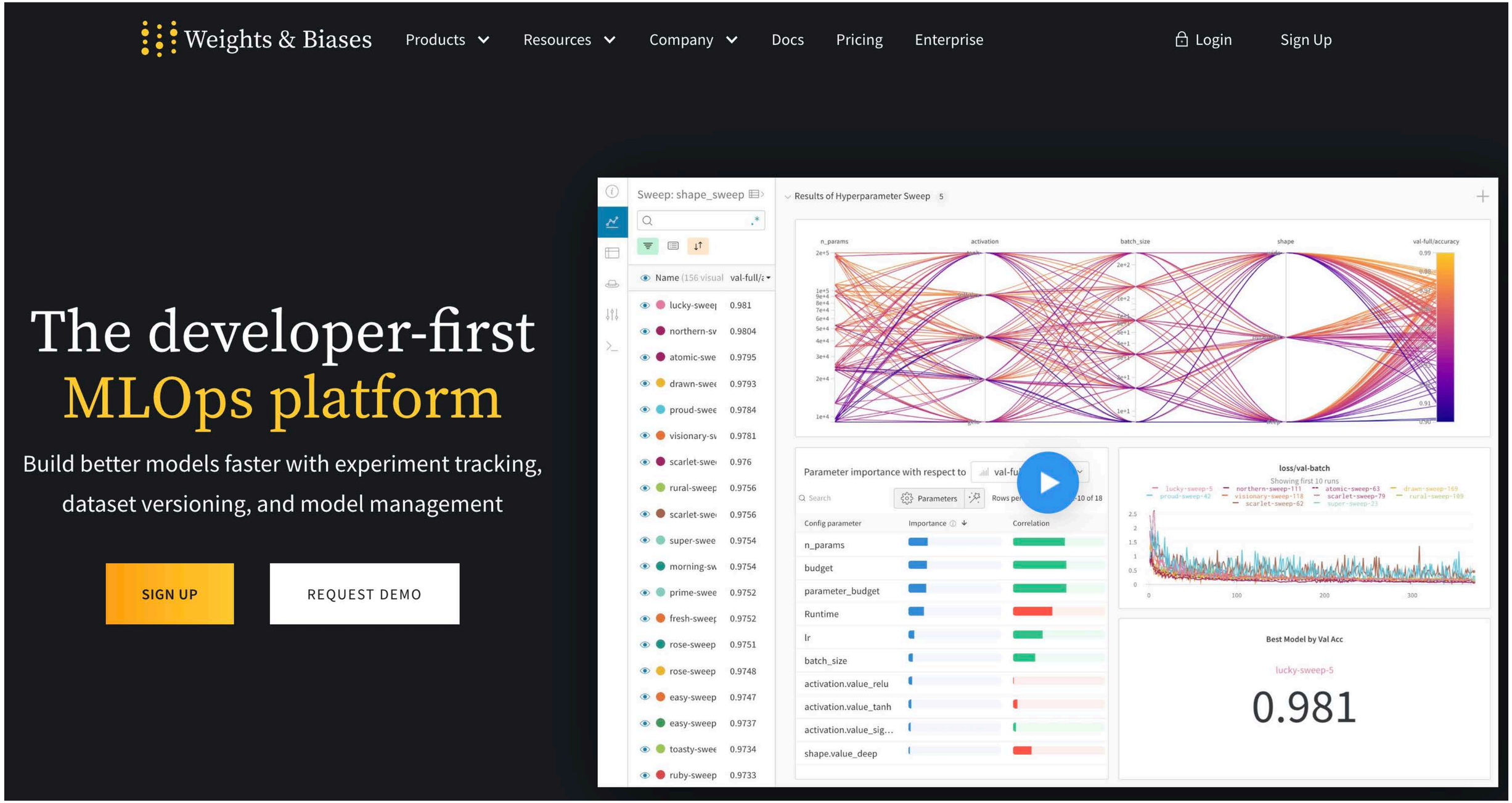


Evaluation

WandB and **Tensorboard** can be your friends (?). Or roll your own logs/viz.

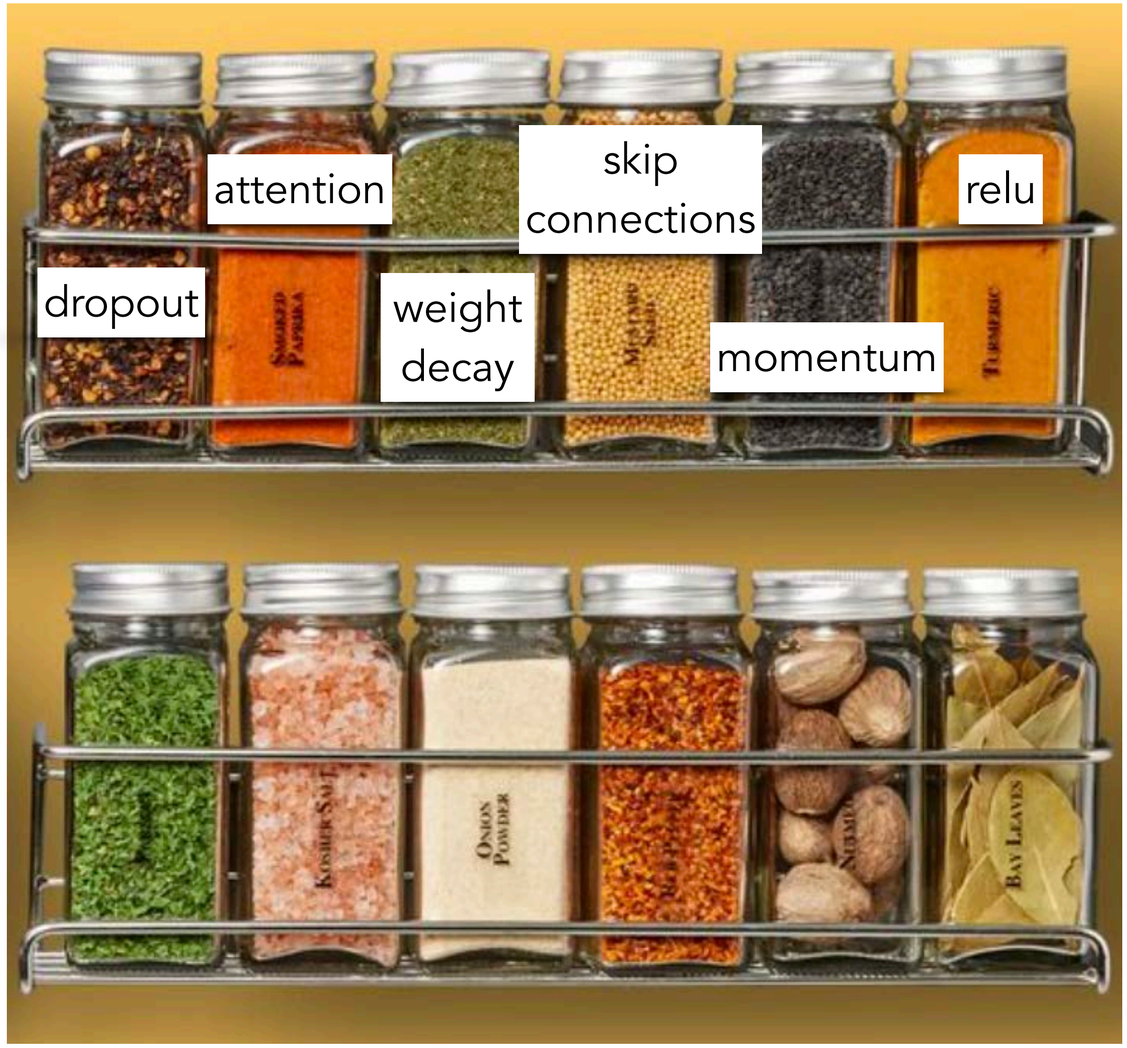
- When in doubt, log it
- If you're logging it, make it easy to see the results

[slide adapted from Dylan Hadfield-Menell]



https://wandb.ai/site

Tuning



Cayenne pepper is all you need?

No! Each spice has its use. But combination matters. And don’t over spice.

Experimentation and debugging

don't be finger-bound! script the optimization + evaluation of your models

every character you type is a chance to make a mistake

also scripting makes the work reproducible!

use config files (e.g., yaml) to manage experiments; log all arguments

[slide adapted from Evan Shelhamer]

Experimentation and debugging

debug with the default python debugger: **pdb**

```
import pdb; pdb.set_trace()
```

https://www.digitalocean.com/community/tutorials/how-to-use-the-python-debugger

For finding nans during debugging:

```
torch.autograd.set_detect_anomaly(True)
```

[slide adapted from Evan Shelhamer]

Common bugs

```
RuntimeError: a view of a leaf Variable that requires grad is being used in an
in-place operation.
```

```
x = torch.ones(2,2, requires_grad=True)

# fails
x += 1

# works
x = x + 1

# fails
x[0,0] = 1

# works (but what should the gradient be?)
y = x.clone()
y[0,0] = 1
```

A leaf variable is one that you directly create, that is not the result of any differentiable operation.

These are the leaves, the inputs, to the computation graph.

Common bugs

```
Out of memory
```

At inference time, don’t store gradients:

```
with torch.no_grad():
  Y = model.forward(X)
```

Clear memory where appropriate:

```
torch.cuda.empty_cache()
del variable_name
```

Common bugs

`Timing your code`

GPU calls may run asynchronously, so if you want to time an operation, make sure to synchronize first:

```
torch.cuda.synchronize()

timer.start()
Y = model.forward(X)
timer.stop()
```

Common bugs

```
RuntimeError: Trying to backward through the graph a second time, but the
buffers have already been freed. Specify retain_graph=True when calling
backward the first time.
```

```
x = torch.randn(1, requires_grad=True)
y = x ** 2

# First backward pass
y.backward(retain_graph=True)

# Second backward pass (this works now)
y.backward()
```

PyTorch frees computational graph after calling backward().

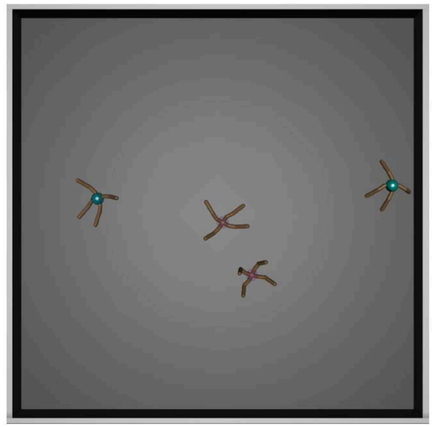
If you see this error it's likely you are doing something you don't want to be doing.

But sometimes you do want to call backward twice on same computation graph, or subparts of it, in which case just set retain_graph=True.

Compute

more hardware, more problems don't parallelize immediately

- make your model work on a single device first
- attempt to parallelize on a single machine
- only then go to a multi machine set
- and check that iterations/time actually improves



see [Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour] for good advice

Compute

Saturate your GPUs

- Check GPU utilization (memory and flops): `nvtop` `nvidia-smi`
- Increase batch size until ~100% utilization

Compute

Include this at the top of your scripts:

```
torch.cudnn.benchmark = True
```

Try AMP (https://developer.nvidia.com/automatic-mixed-precision)

```
scaler = GradScaler()
with autocast():
    output = model(input)
    loss = loss_fn(output, target)
scaler.scale(loss).backward()
scaler.step(optimizer)
scaler.update()
```

Try torch.compile (https://pytorch.org/tutorials/intermediate/torch_compile_tutorial.html)

MIT group with presentations / tutorials on cutting edge practice of training big models

https://scale-ml.org/

Scale ML

▸ We are a cross-lab MIT AI graduate student collective focusing on **Algorithms That Learn and Scale**.
▸ The group is open to all with an academic email - however if you are still interested shoot us an email or message us via Twitter. We currently host bi-weekly seminars and will have hands on sessions and research socials in the future.
▸ Our snacks 🍰 are currently funded by generous donations from Pulkit Agrawal and Yoon Kim.
▸ Please contact the organizers for inquires
▸ Join our next seminar on Zoom or in-person:
Click here to join the mailing list

Discussion Schedule

10/30	u-μP: The Unit-Scaled Maximal Update Parametrization	Charlie Blake (Graphcore)
10/16	Transformers and Turing Machines	Eran Malach (Harvard)
09/04	A New Perspective on Shampoo's Preconditioner	Nikhil Vyas (Harvard)
08/22	1B parameter model training. (hands on session)	Aniruddha Nrusimha (MIT)
08/12	How to scale models with Modula in NumPy. (hands on session)	Jeremy Bernstein (MIT)
07/24	FineWeb: Creating a large dataset for pretraining LLMse	Guilherme Penedo (Hugging Face)
07/17	Hardware-aware Algorithms for Language Modeling	Tri Dao (Princeton)

6.7960 Deep Learning
Fall 2024