



# Lecture 12

## Mechanisms of Training and Running Neural Nets

# Announcements

- Pset 4 due today
- Pset 5 out today

# 12. Mechanisms of Training and Running Neural Nets

- Data
- Model
- Optimization
- Evaluation, Execution, and Debugging



Lots of slides adapted from **Evan Shelhamer's** "DIY Deep Learning: Advice on Weaving Nets"



# Data

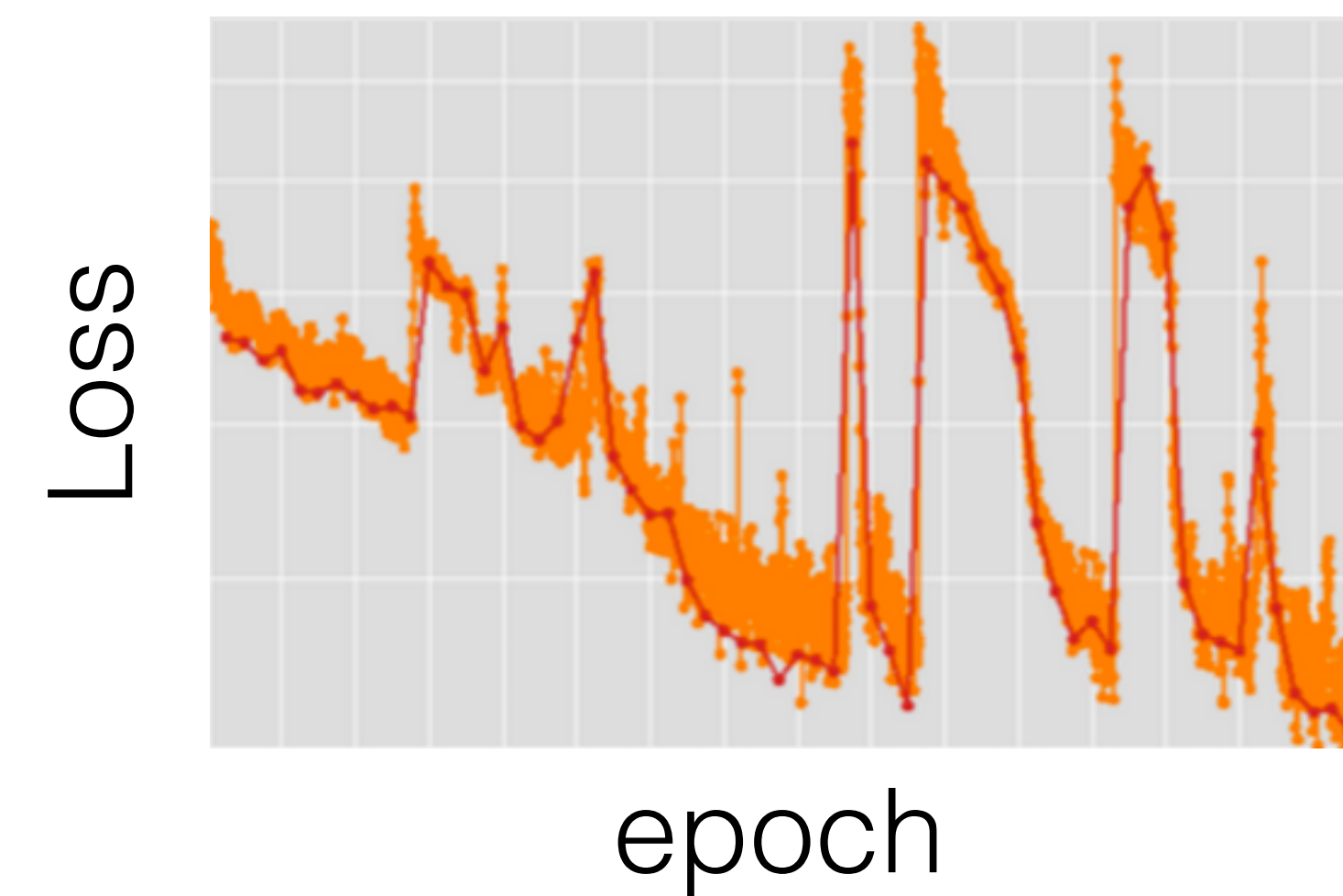
Machine learning == Data-driven intelligence







**“Become friends with every pixel”**







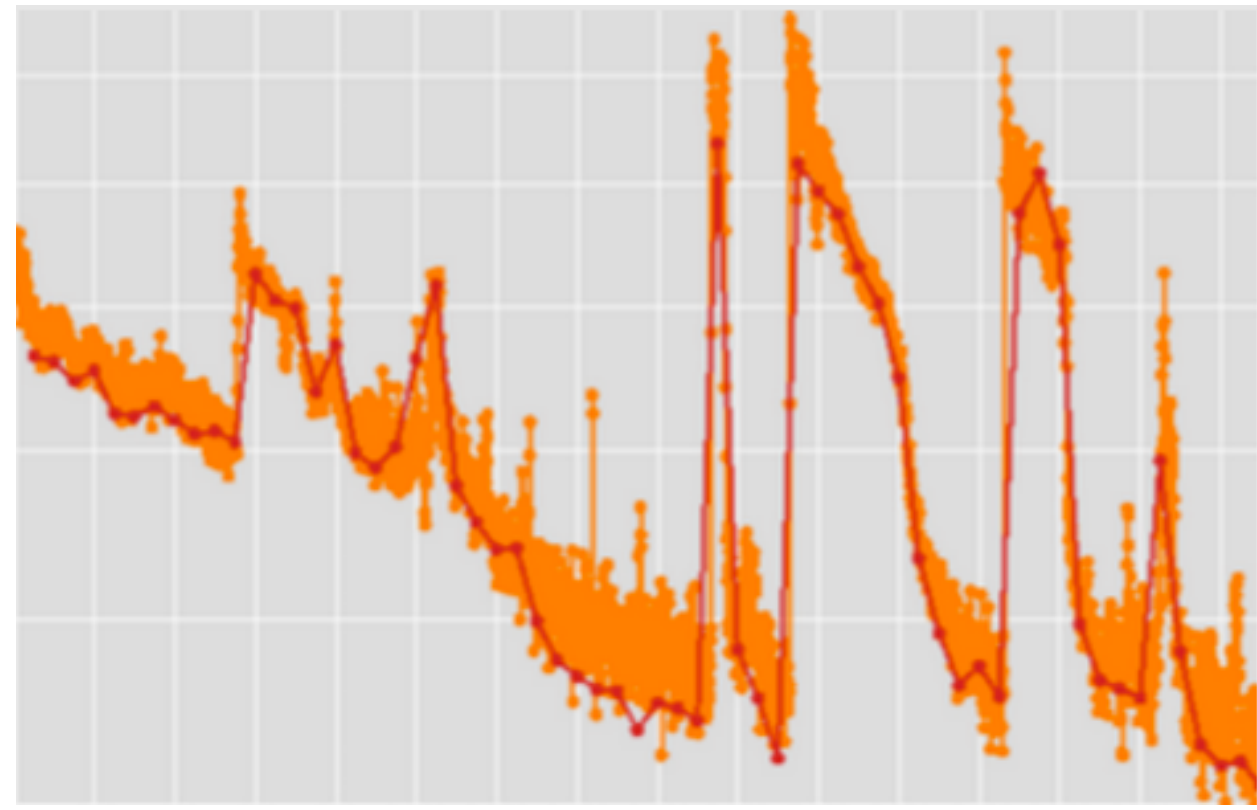
**“Become friends with every pixel”**

*Look at your data*



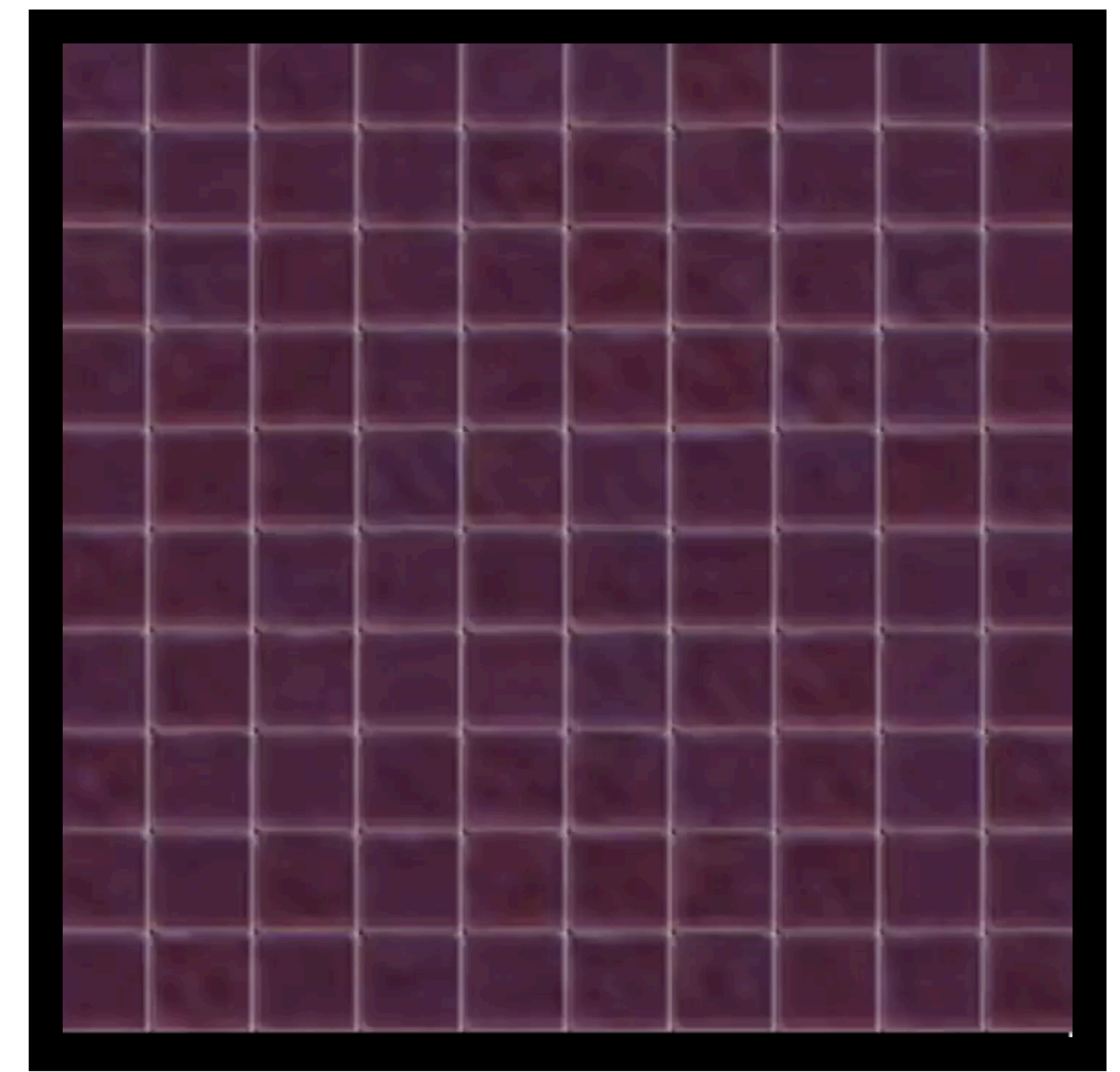


Loss

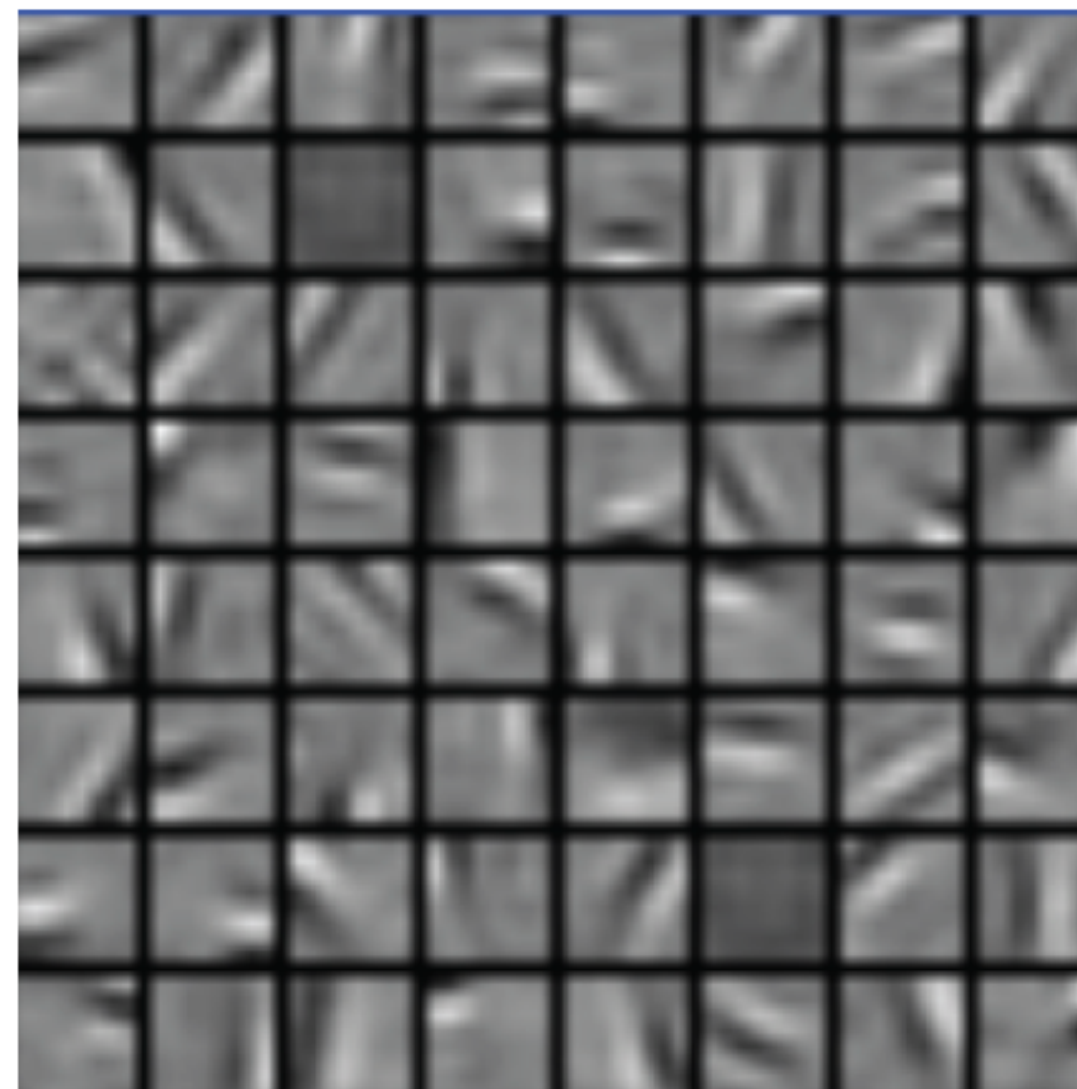


epoch

*Look at the output*



*Look at the filters*



*Visualize neurons*





# 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



# 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



# 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



# Data

pre-processing:

- **standardize:**

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

- 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



# 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  $64 \times 64 \times 64 \times 64$  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?



# Data

resample the data to decorrelate: that is, remember to **shuffle**

- when you shuffle your inputs,  $x$ , make sure your targets,  $y$ , are shuffled in the same order!

consider selecting miniature train and test sets for development and testing: these should be chosen once and fixed throughout optimization + evaluation

- it's heartbreaking to wait an entire epoch and then and only then have your experiment-to-be crash



# Data

check if you can do the task, as it is given to the model

- take windowed data at receptive field size

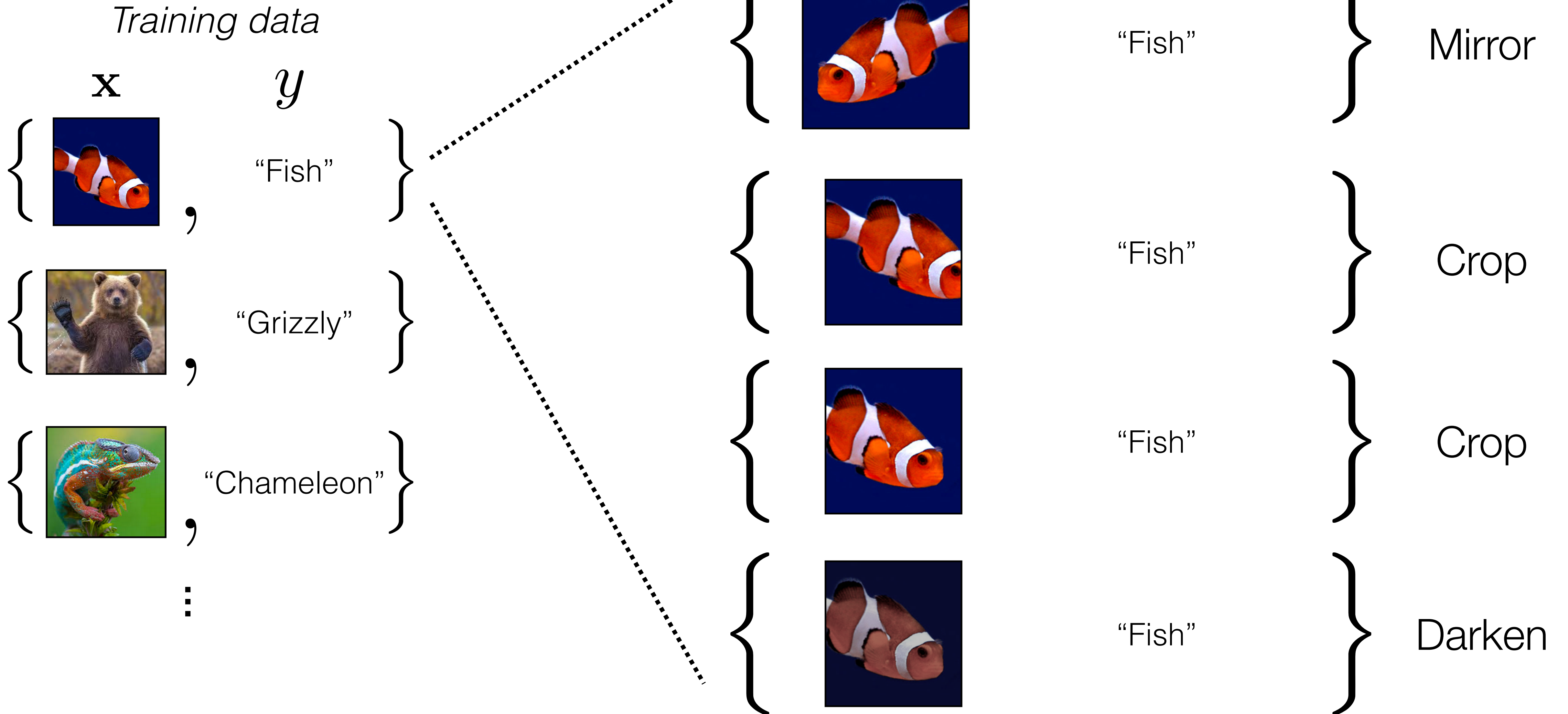


... see if it's a reasonable perceptual task for a human



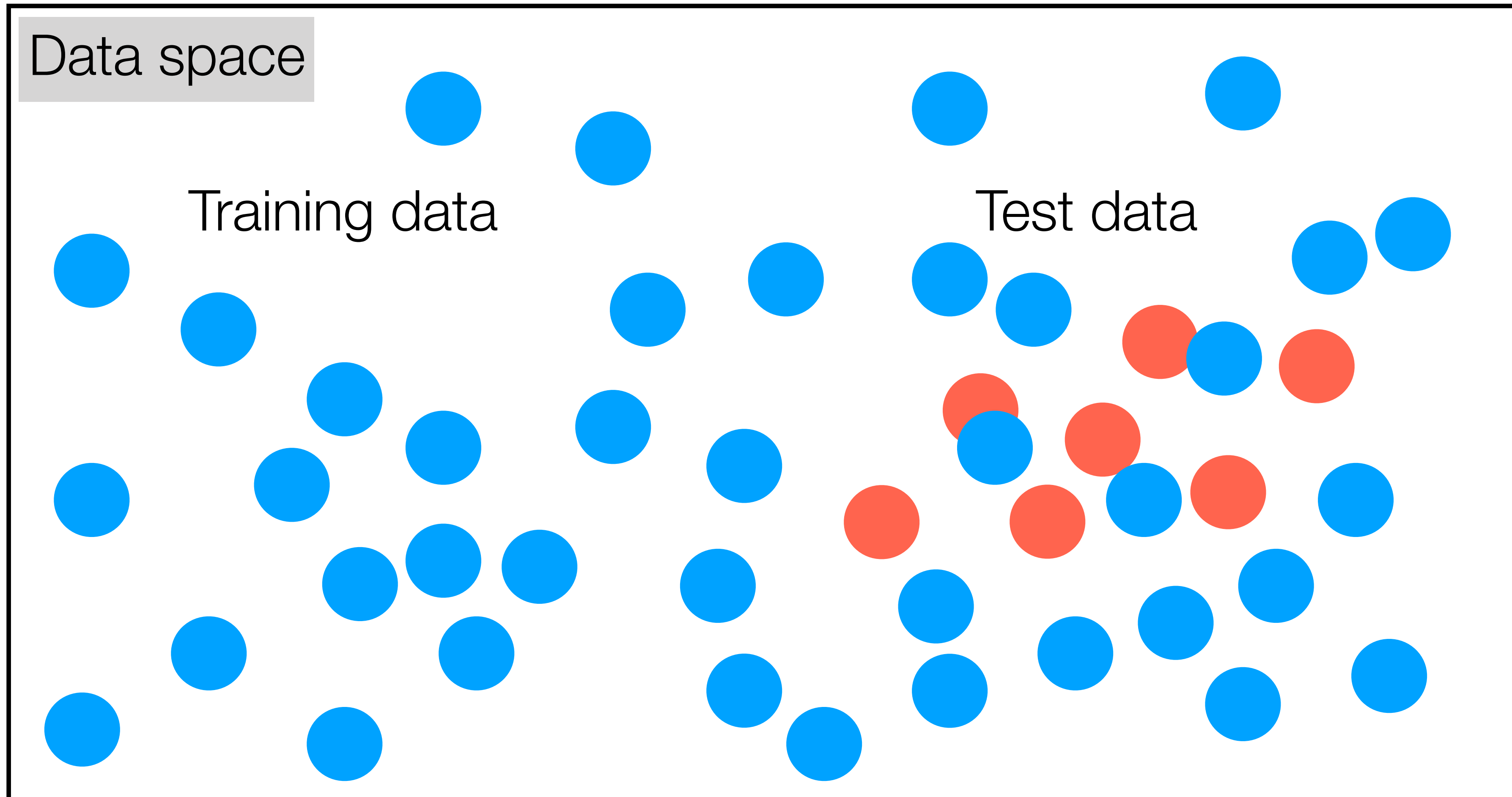
# Data

# Data augmentation





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

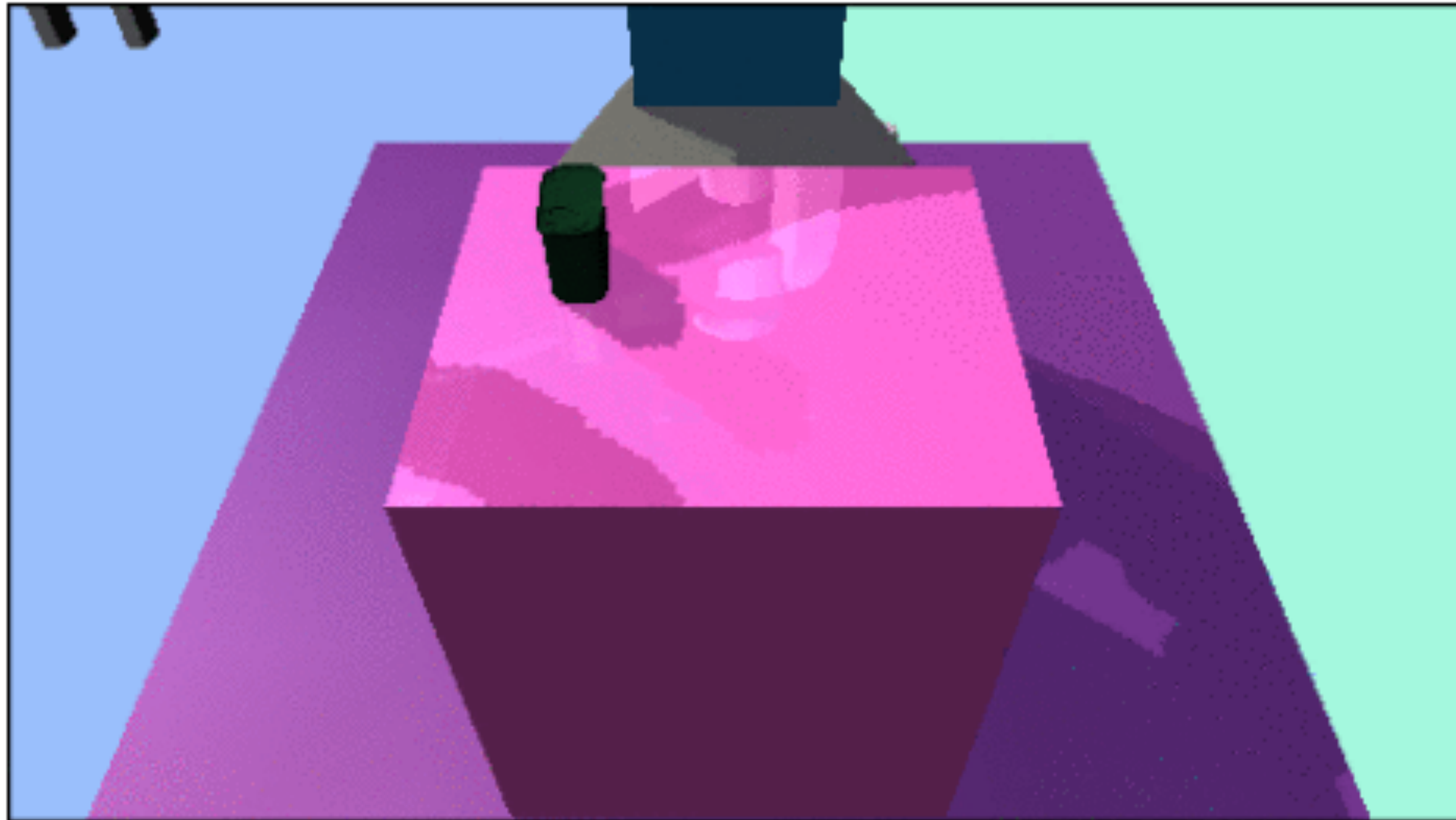


This is called **domain randomization** or **data augmentation**

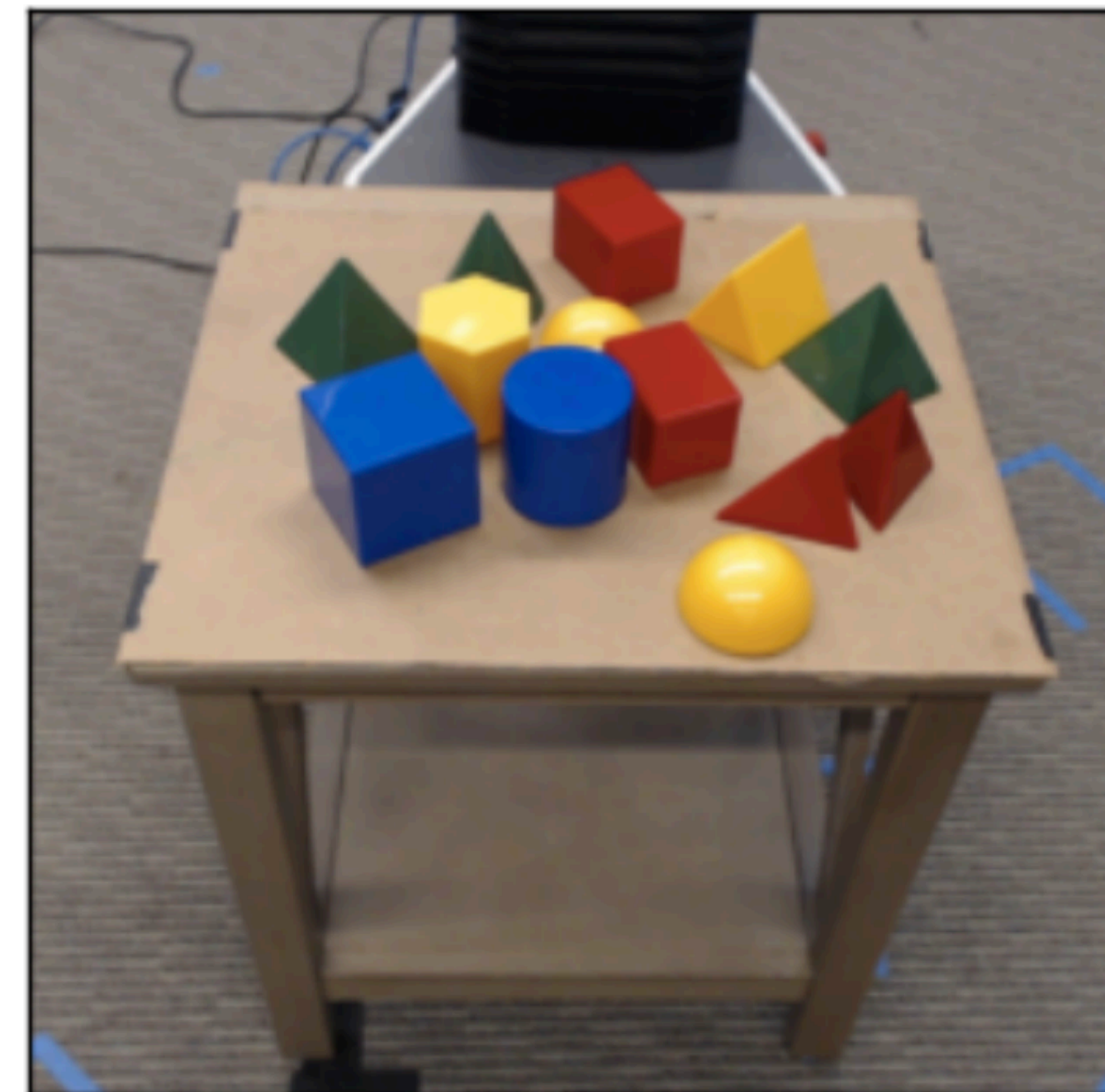


# Domain randomization

Training data



Test data



[Sadeghi & Levine 2016]

Above example is from [Tobin et al. 2017]



*source domain*

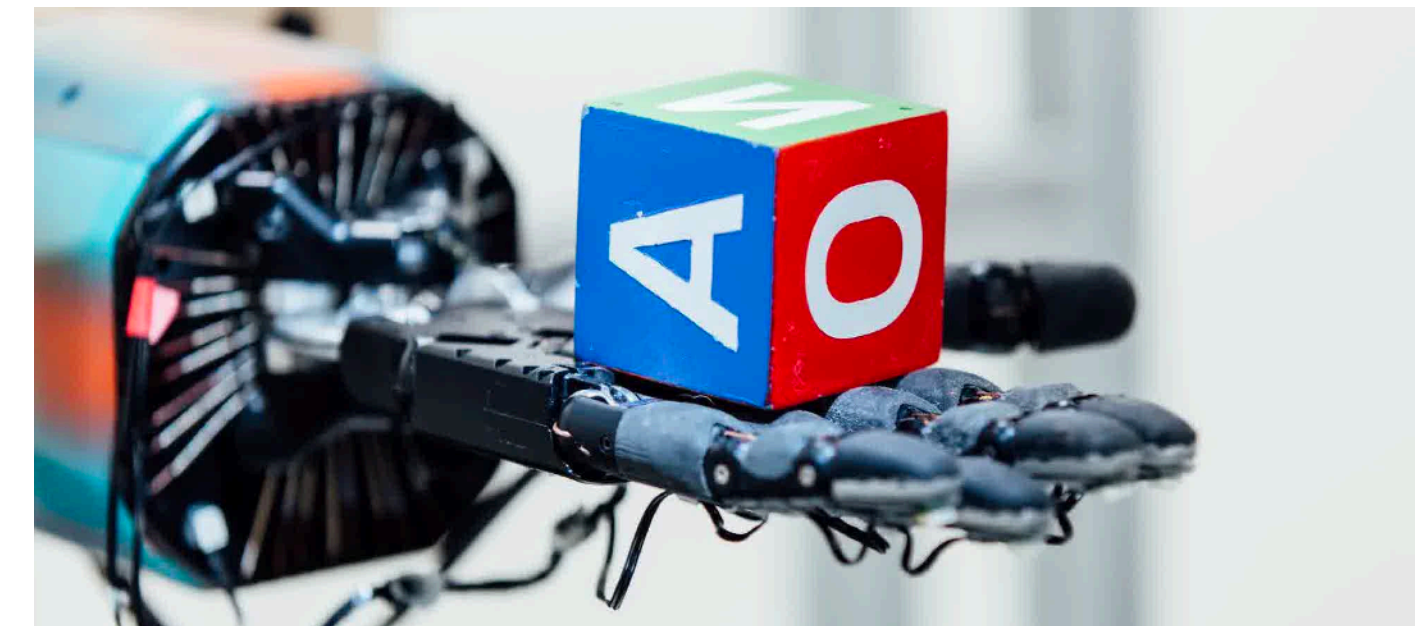
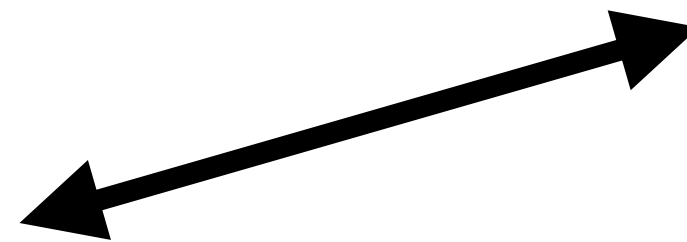
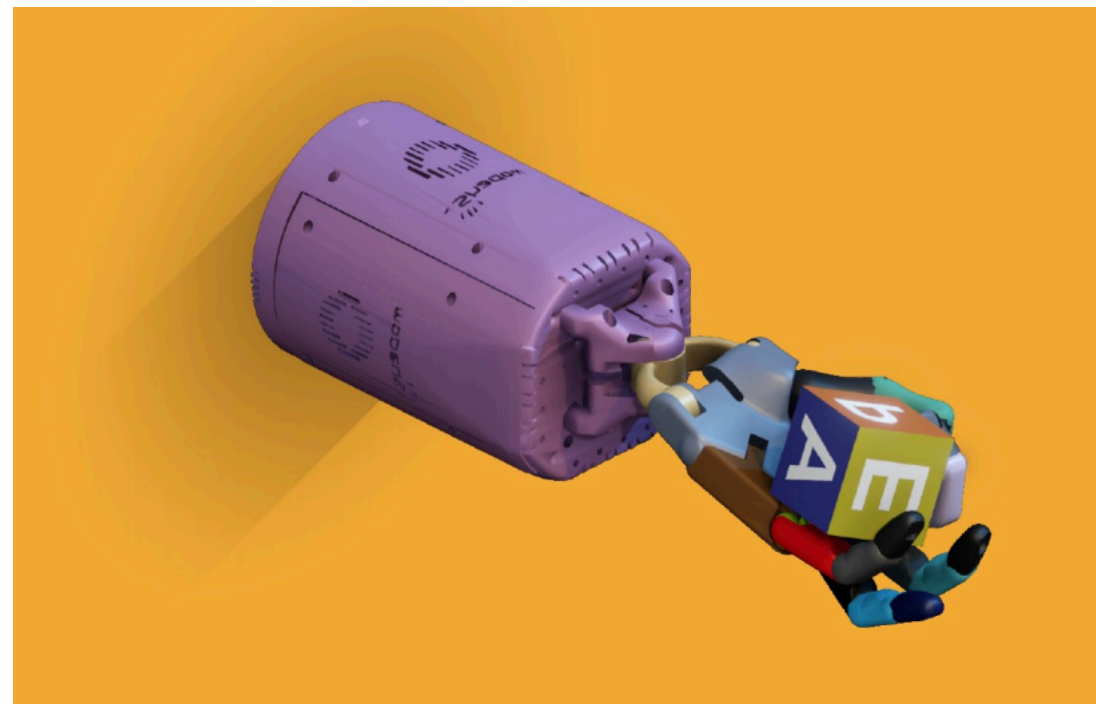
*target domain*

(where we actual use our model)

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

Space of images

Source data



Target data



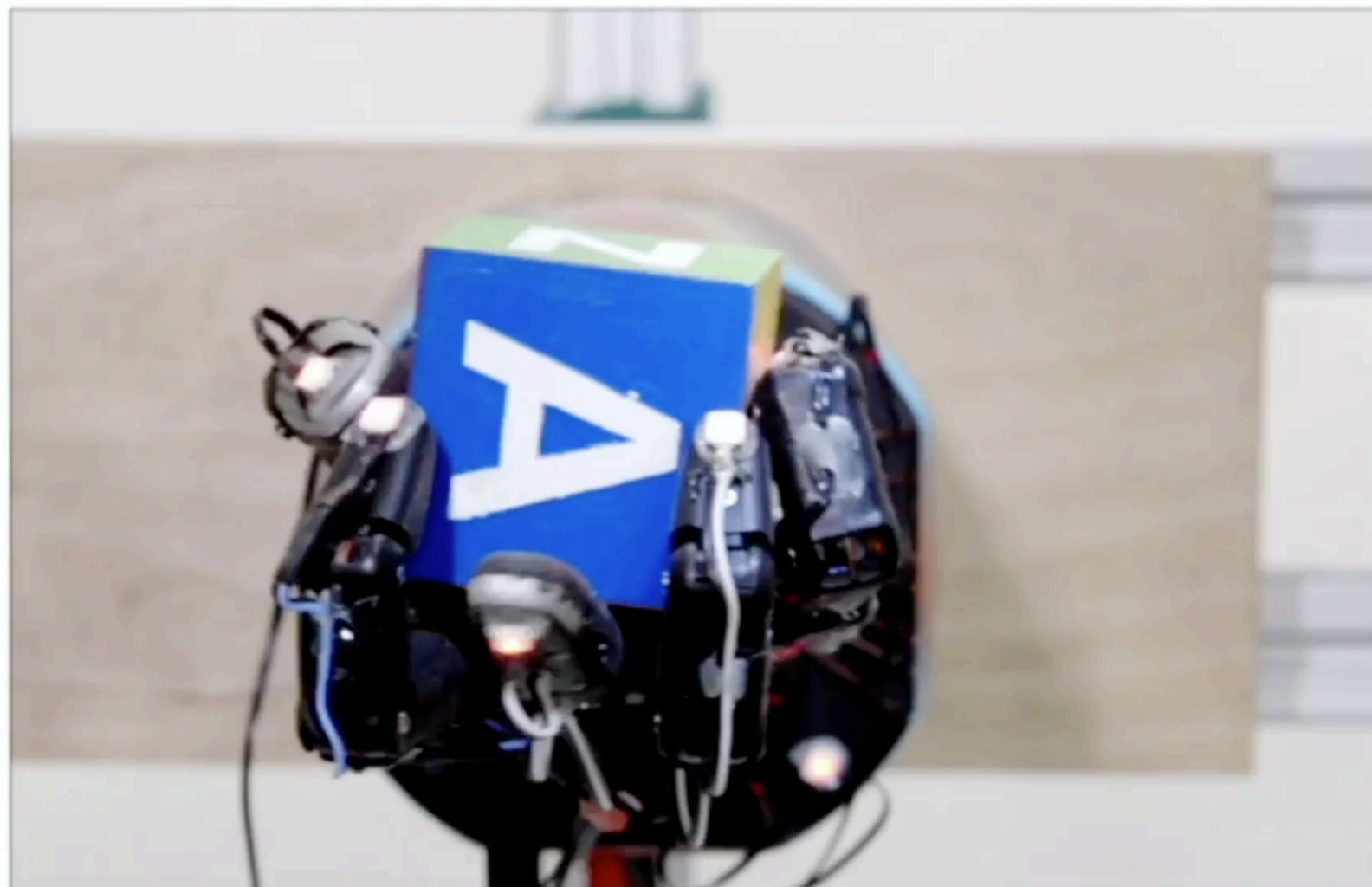
# OpenAI Dactyl



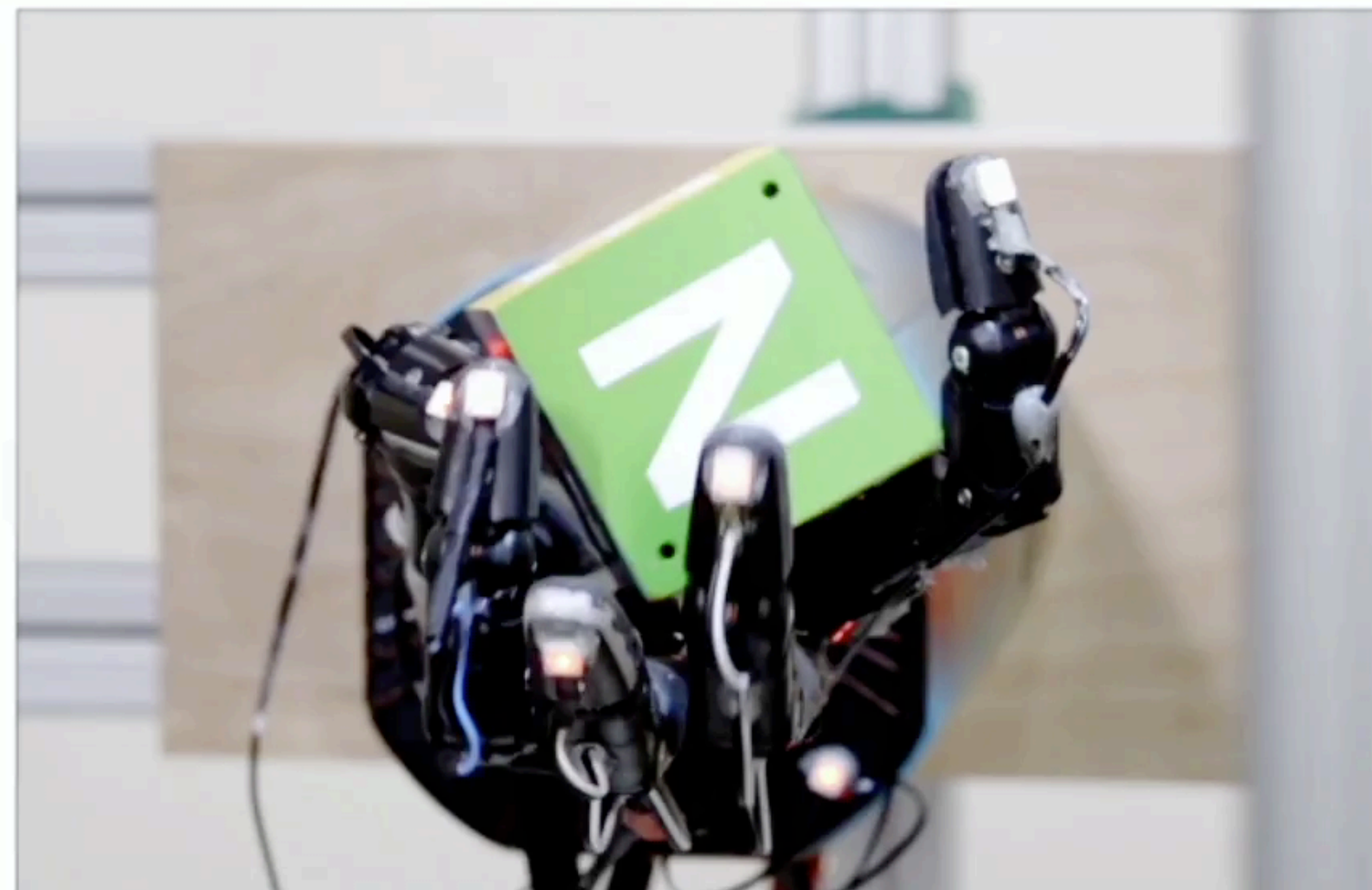
[\[https://openai.com/blog/learning-dexterity/\]](https://openai.com/blog/learning-dexterity/)



# OpenAI Dactyl



**FINGER PIVOTING**



**SLIDING**



**FINGER GAITING**

# Model

**Keep it as simple as possible!**

sure, there are sophisticated models out there

but they were made by either

- 1) going step-by-step, from simple to complex, or
- 2) suffering, madness, and the gnashing of teeth



# Model

**Keep it as simple as possible!**

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)
- My personal opinion: simplicity is highly undervalued in the community; if you focus on simplicity you will have an unfair advantage

# Model

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

if you have a classification problem, you might try:

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

if you have a detection problem, you might try:

```
git clone https://github.com/roytseng-tw/mask-rcnn.pytorch.git
```

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



# Model

**double-check the model** actually is what you thought you defined

```
# walk the model for inspection
for name_, module_ in model.named_modules():
    if name_ == 'name' or isinstance(module_, nn.Conv2d):
        [...]
```

# Model

**simple baselines catch many issues** and swiftly too

- learn a linear model on random features by freezing all the parameters at their initialization except for the final output layer
- zero out the data by `x.zero()` and check that the results are worse



# Model

## **Transform your problem into a “solved” problem**

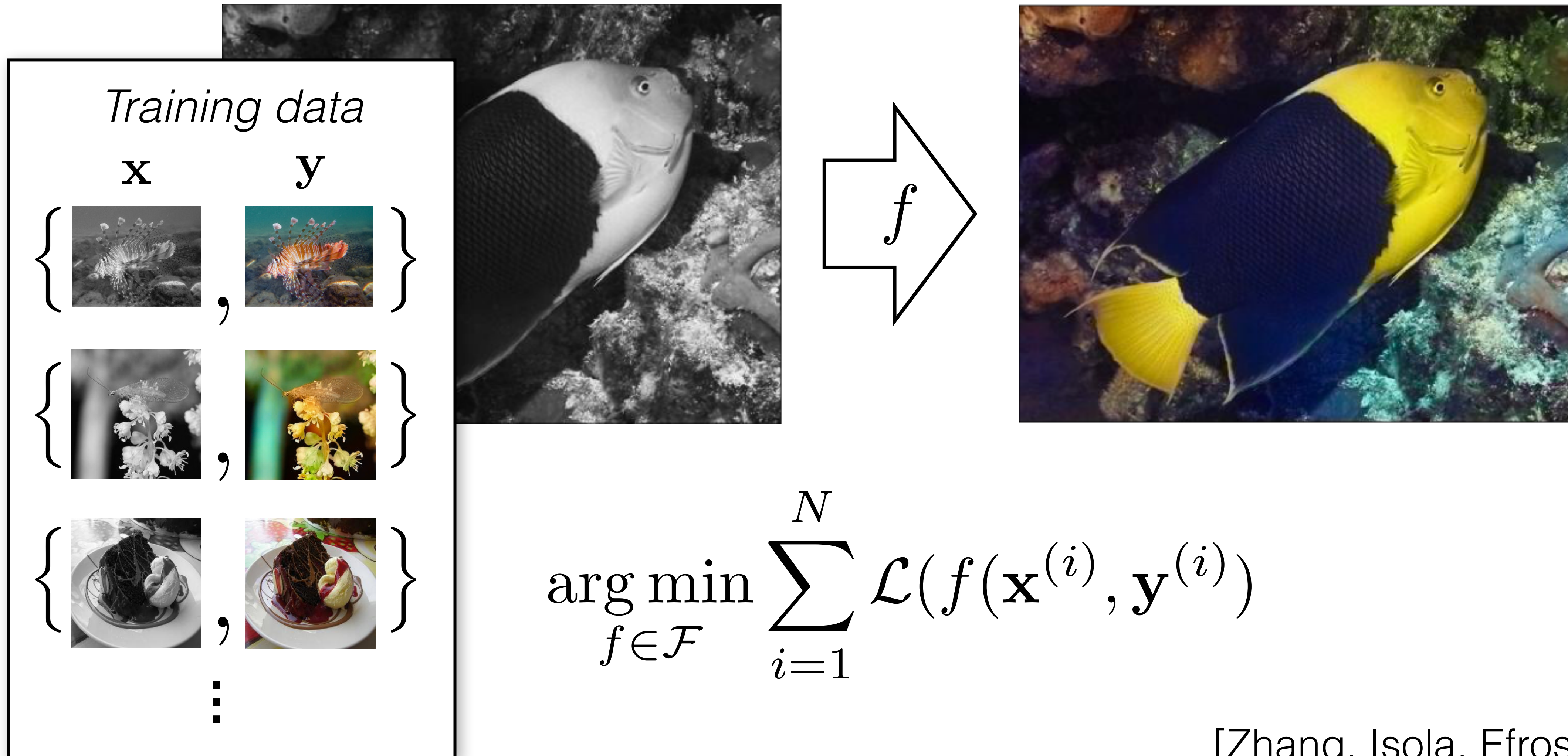
- Case study: transforming image *colorization* to image *classification*

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

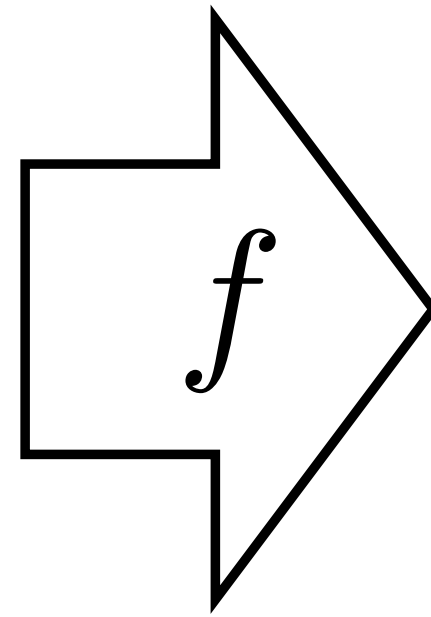
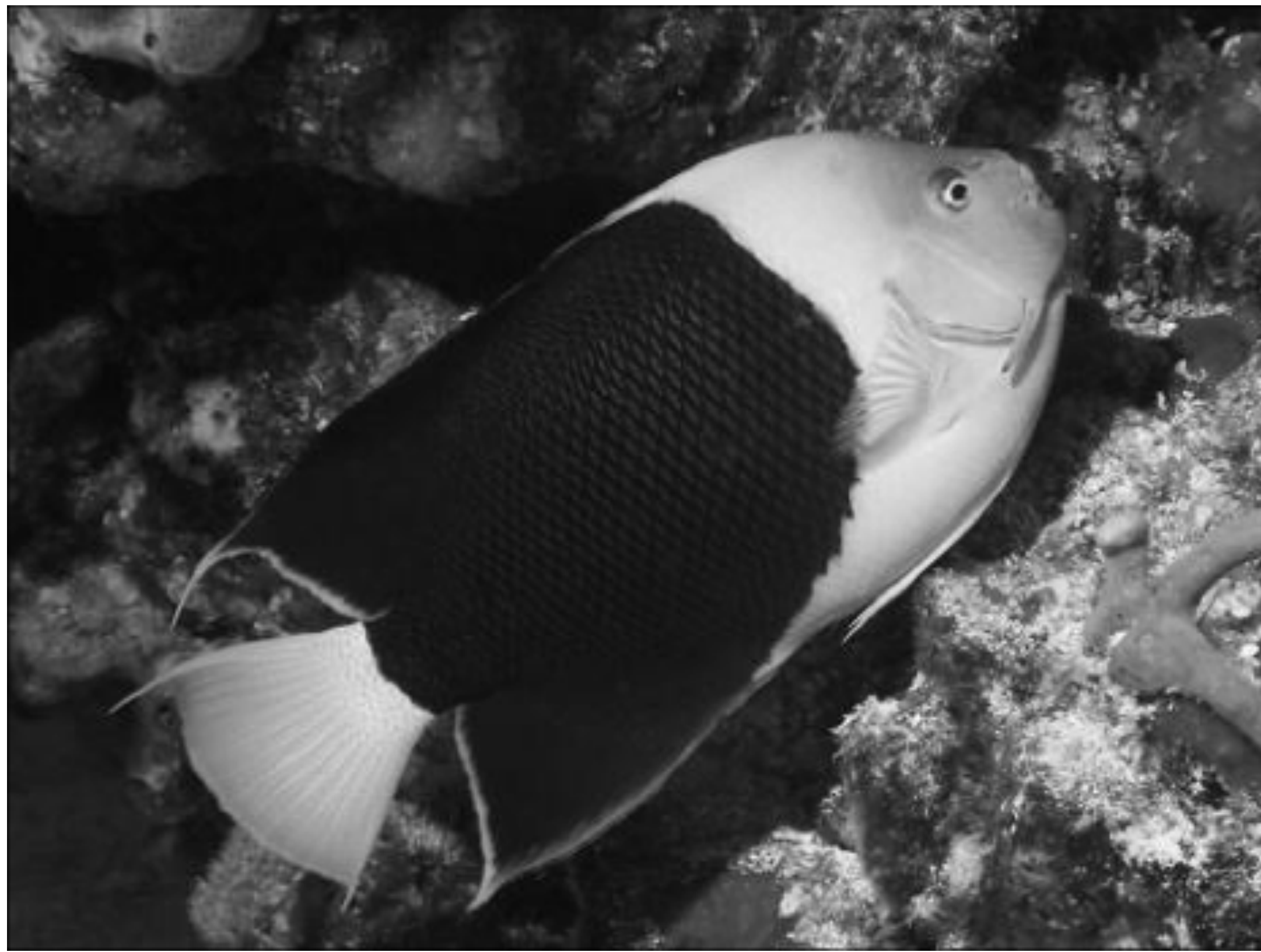
# Image colorization

Input  $\mathbf{x}$

Output  $\mathbf{y}$







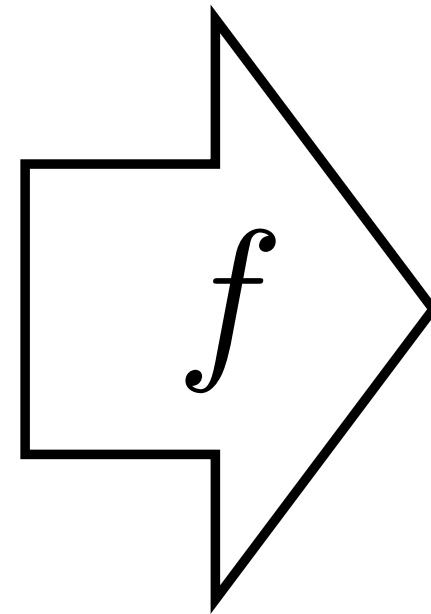
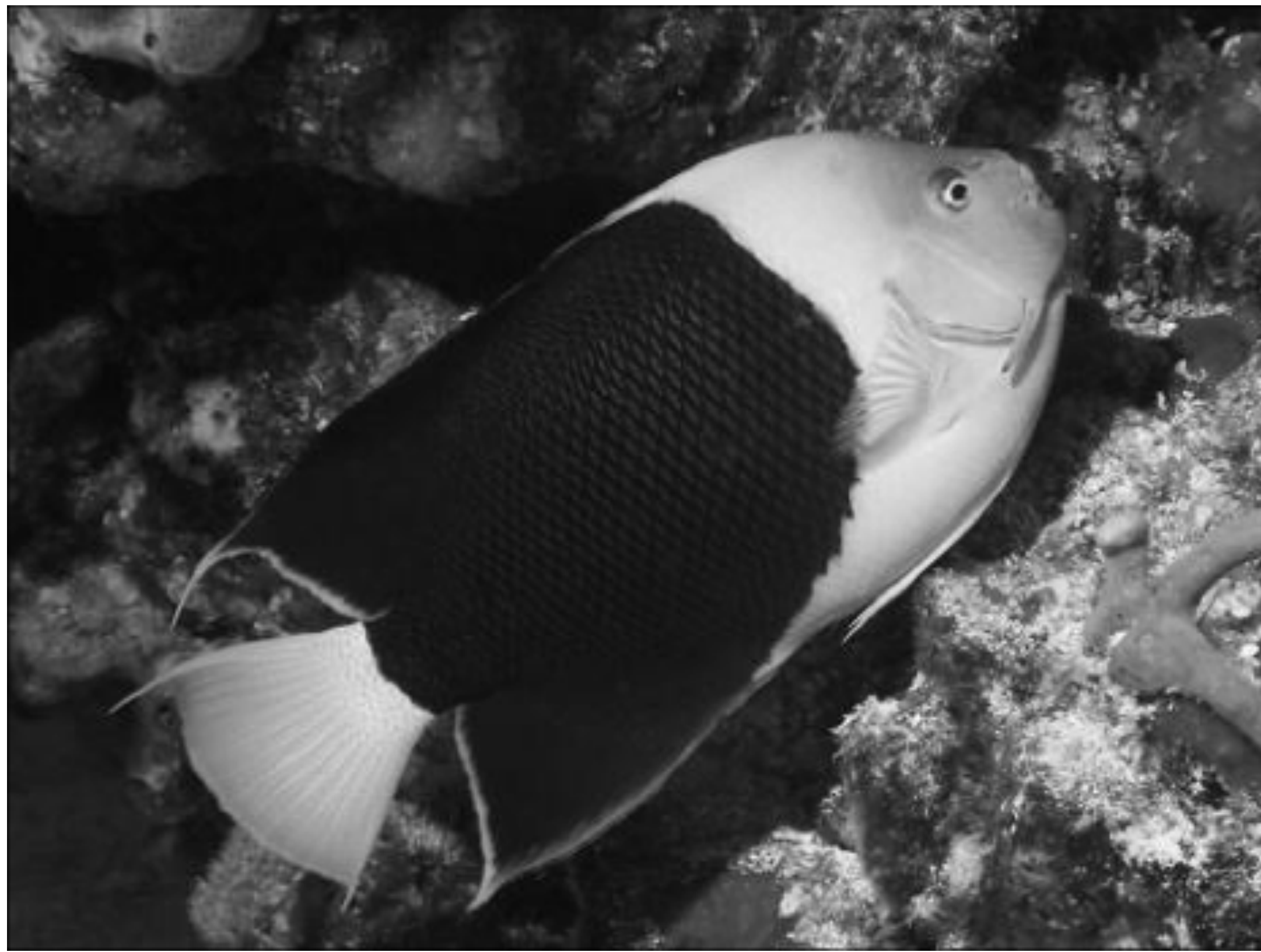
Grayscale image: **L channel**

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

Color information: **ab channels**

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





Grayscale image: **L channel**

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

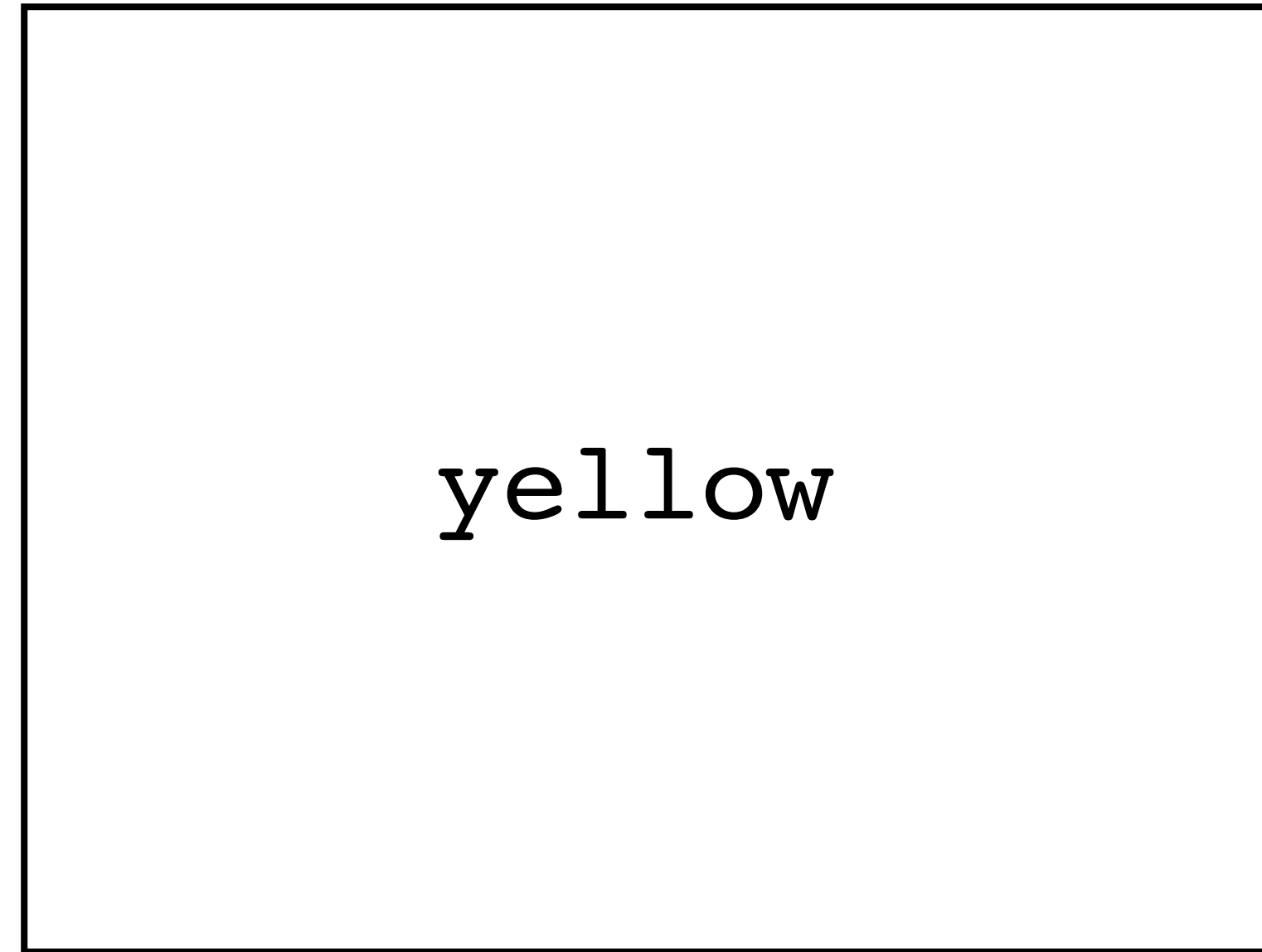
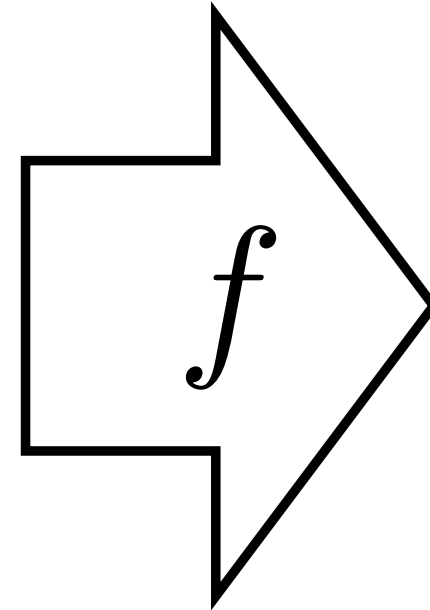
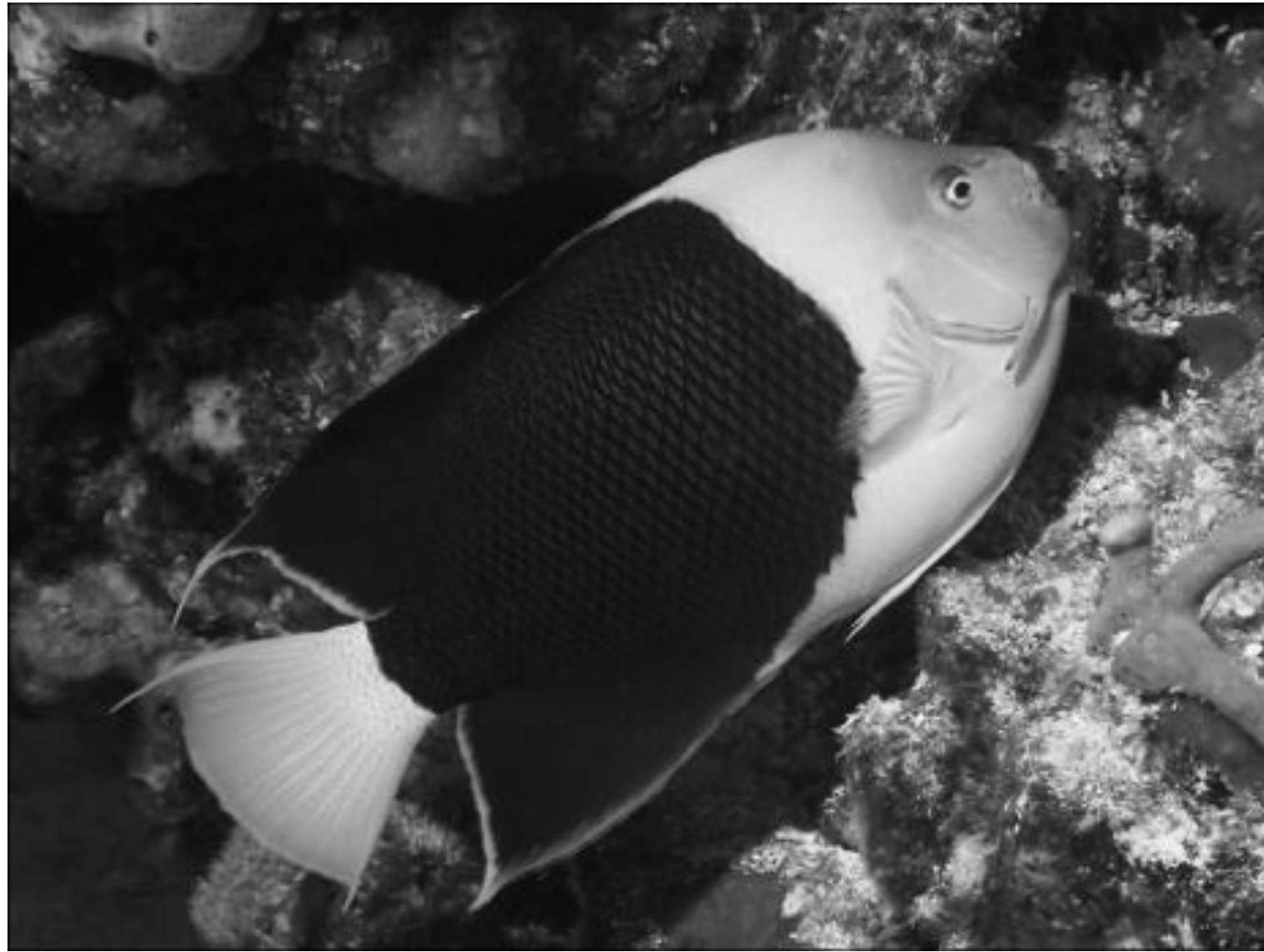
Color information: **ab channels**

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

[Zhang, Isola, Efros, ECCV 2016]

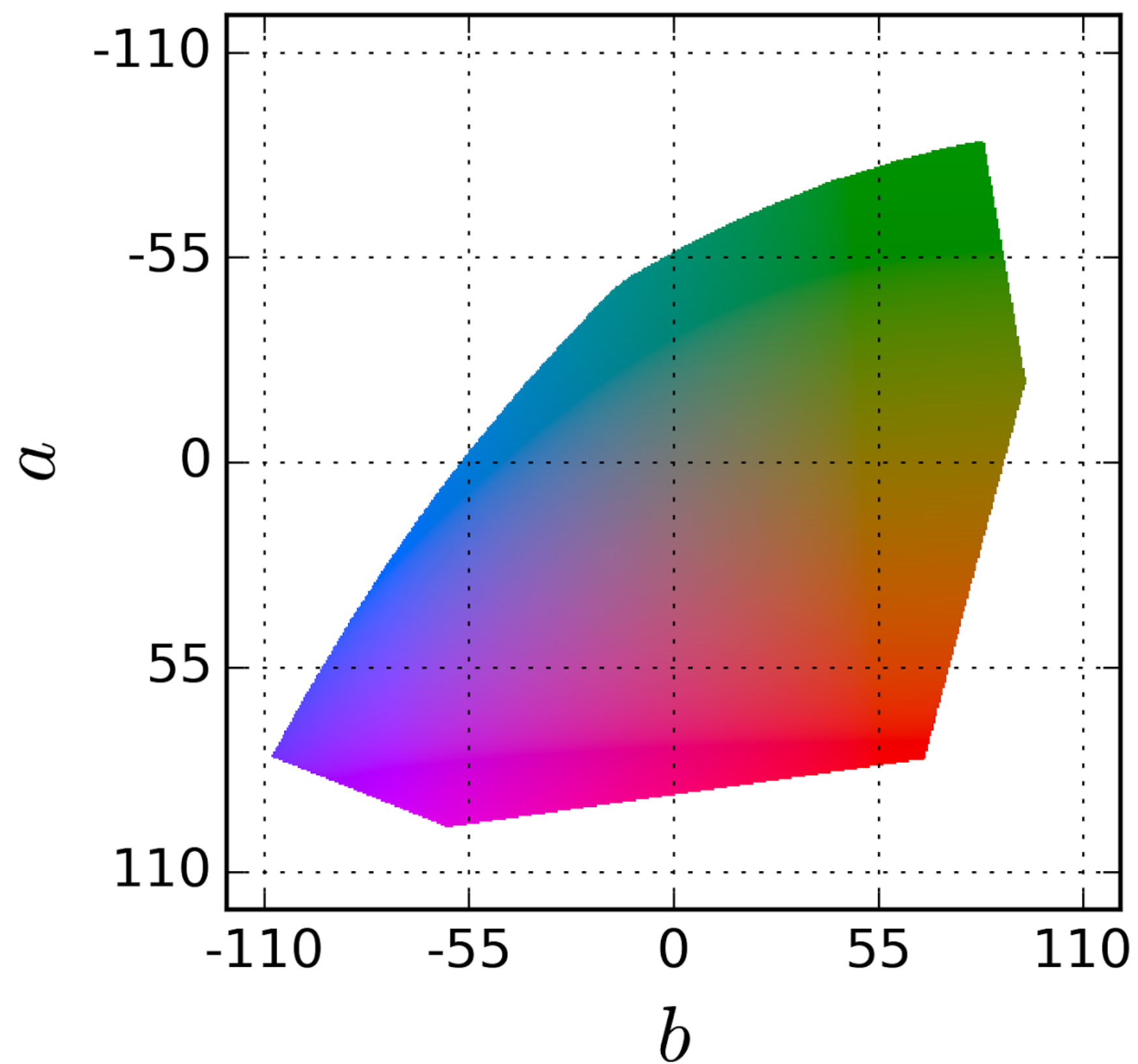


# Colorization $\rightarrow$ Classification



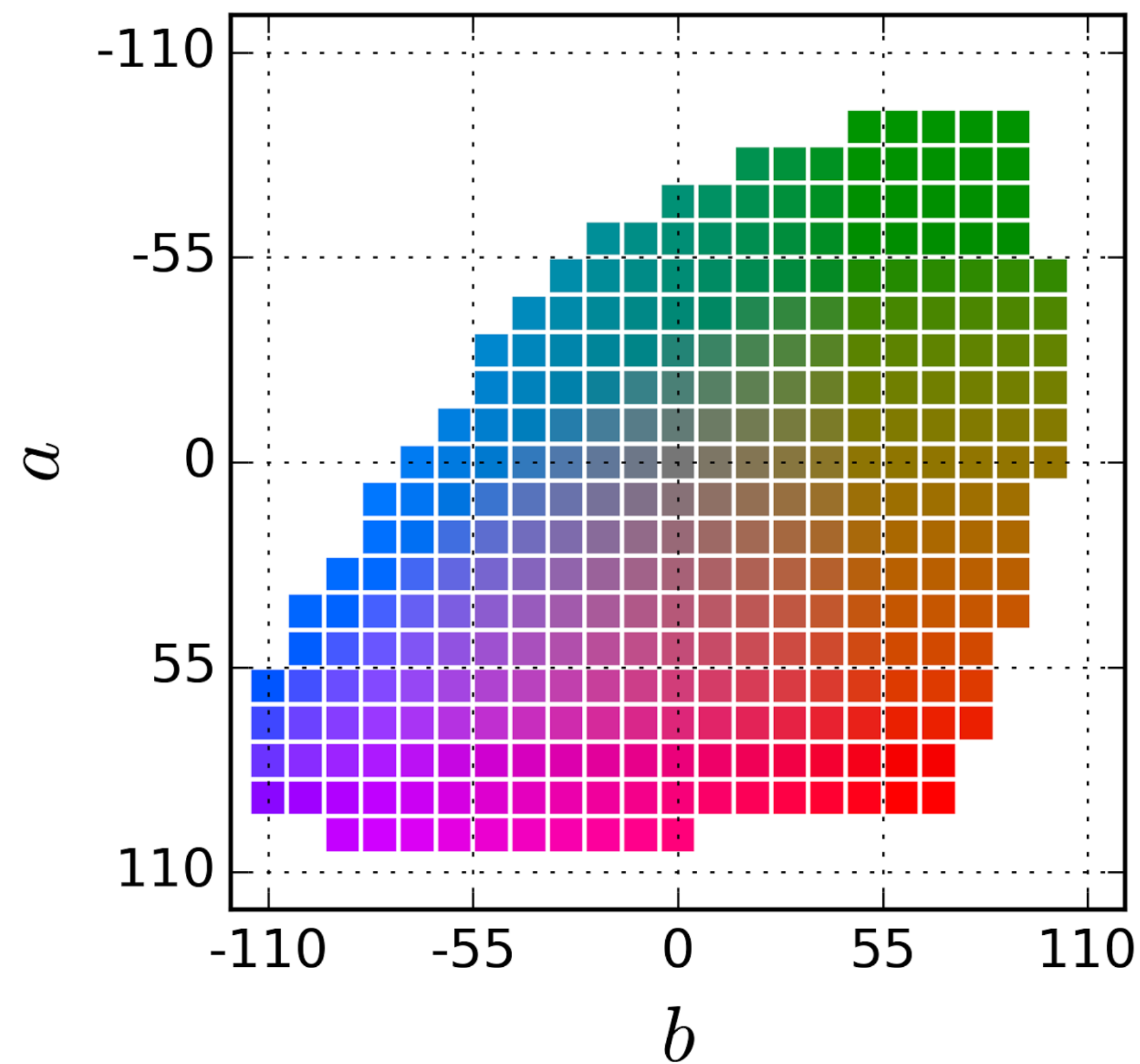
# Colors $\rightarrow$ Classes

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

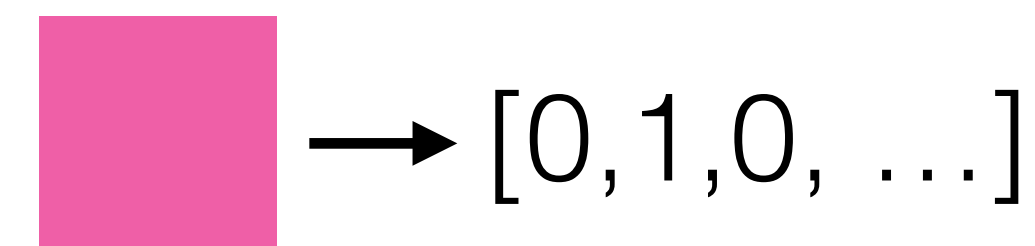
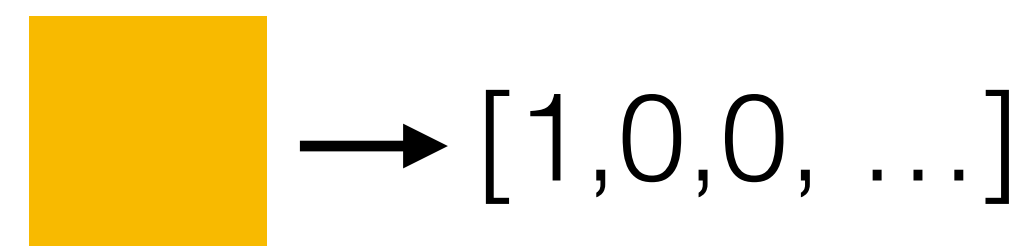
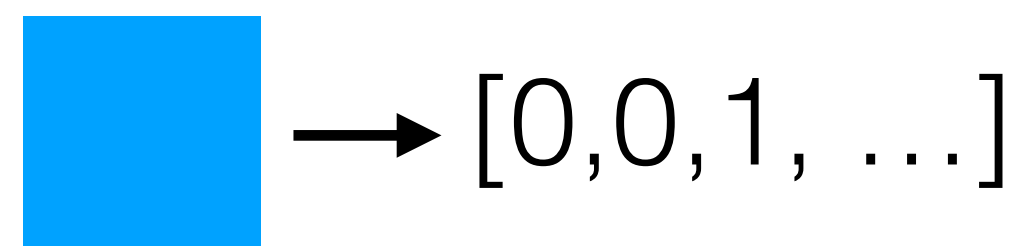


one-hot representation of  $K$  discrete classes

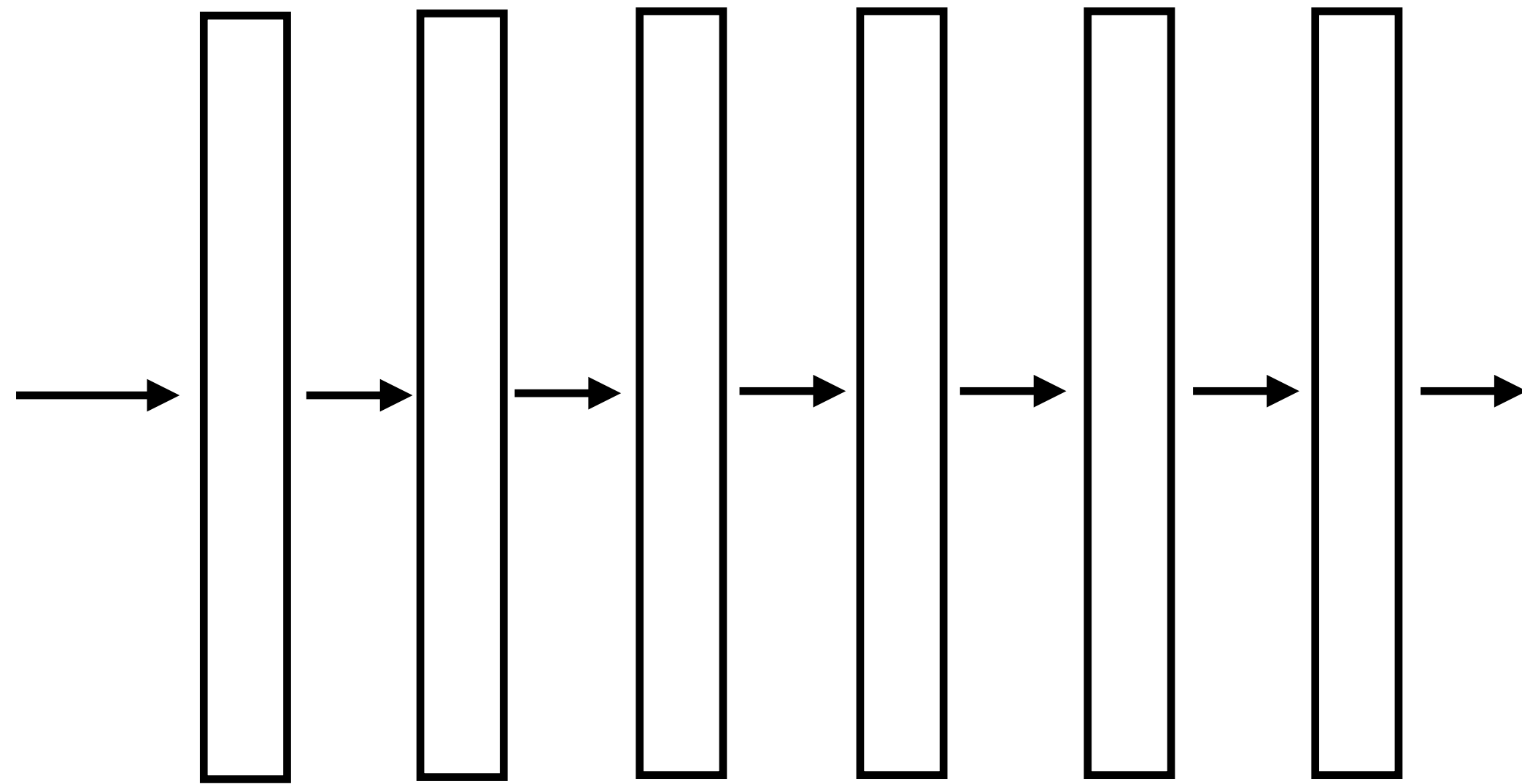
$$\mathbf{y} \in \mathbb{R}^{H \times W \times K}$$



One hot codes:



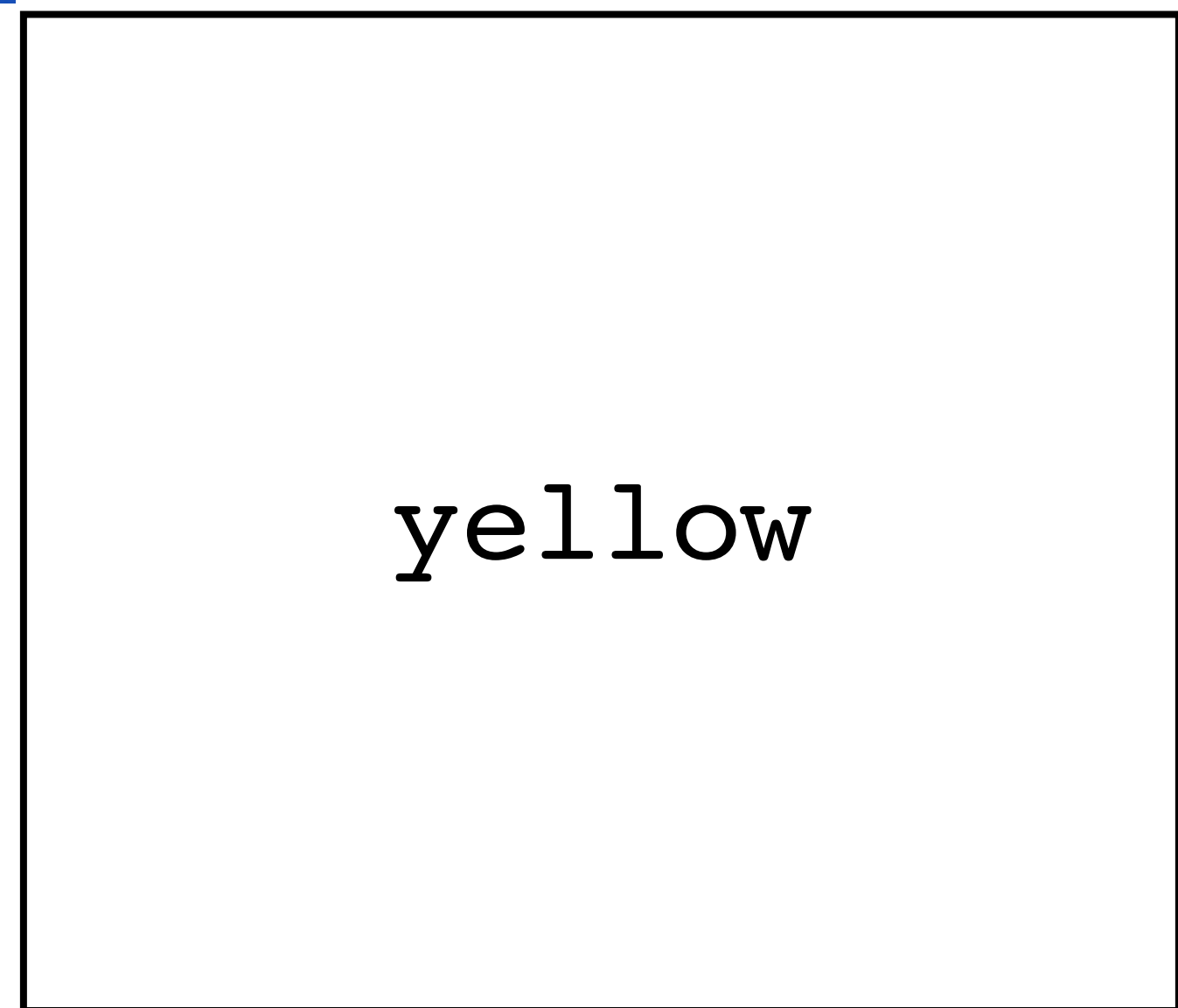
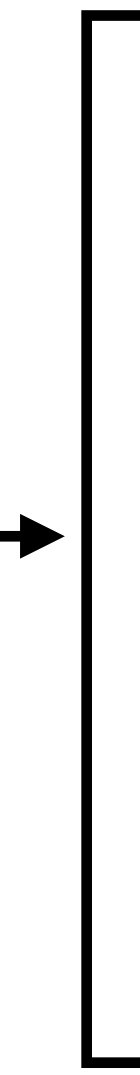
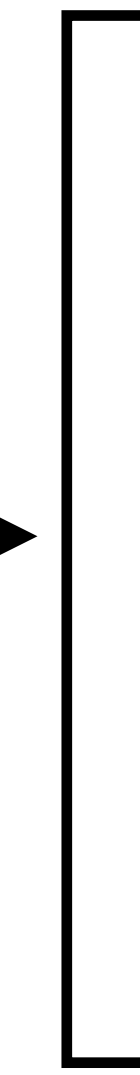
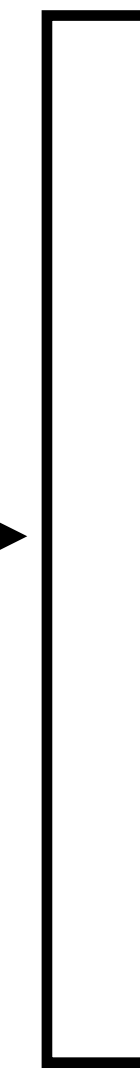
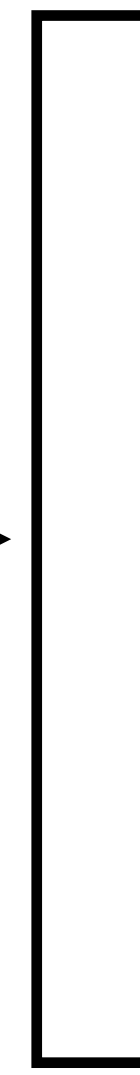
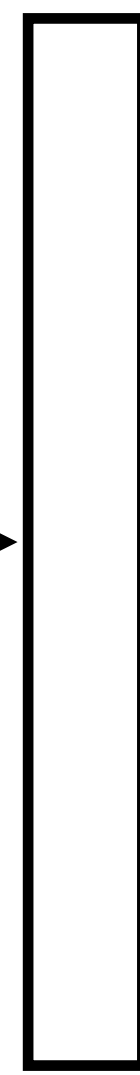
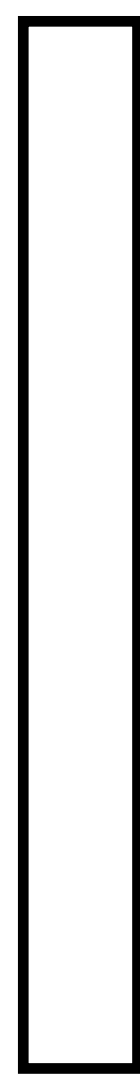




“rockfish”



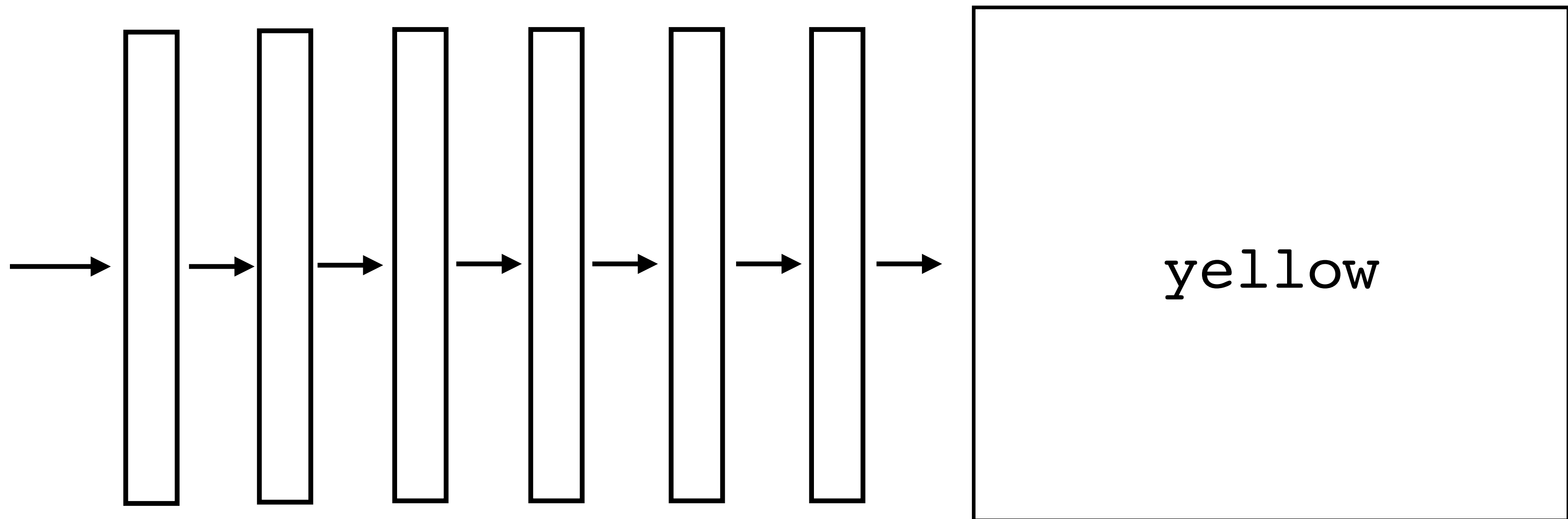
...



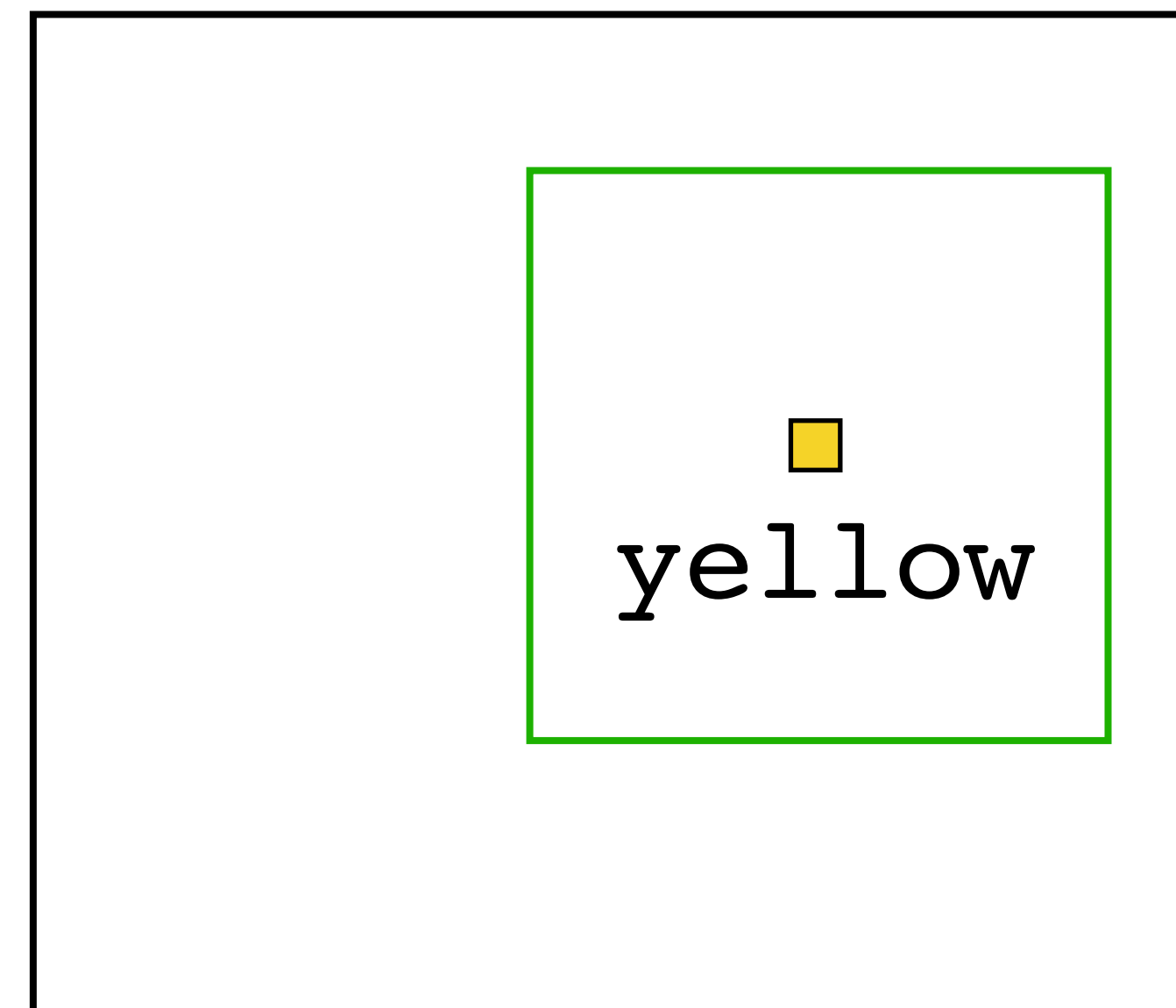
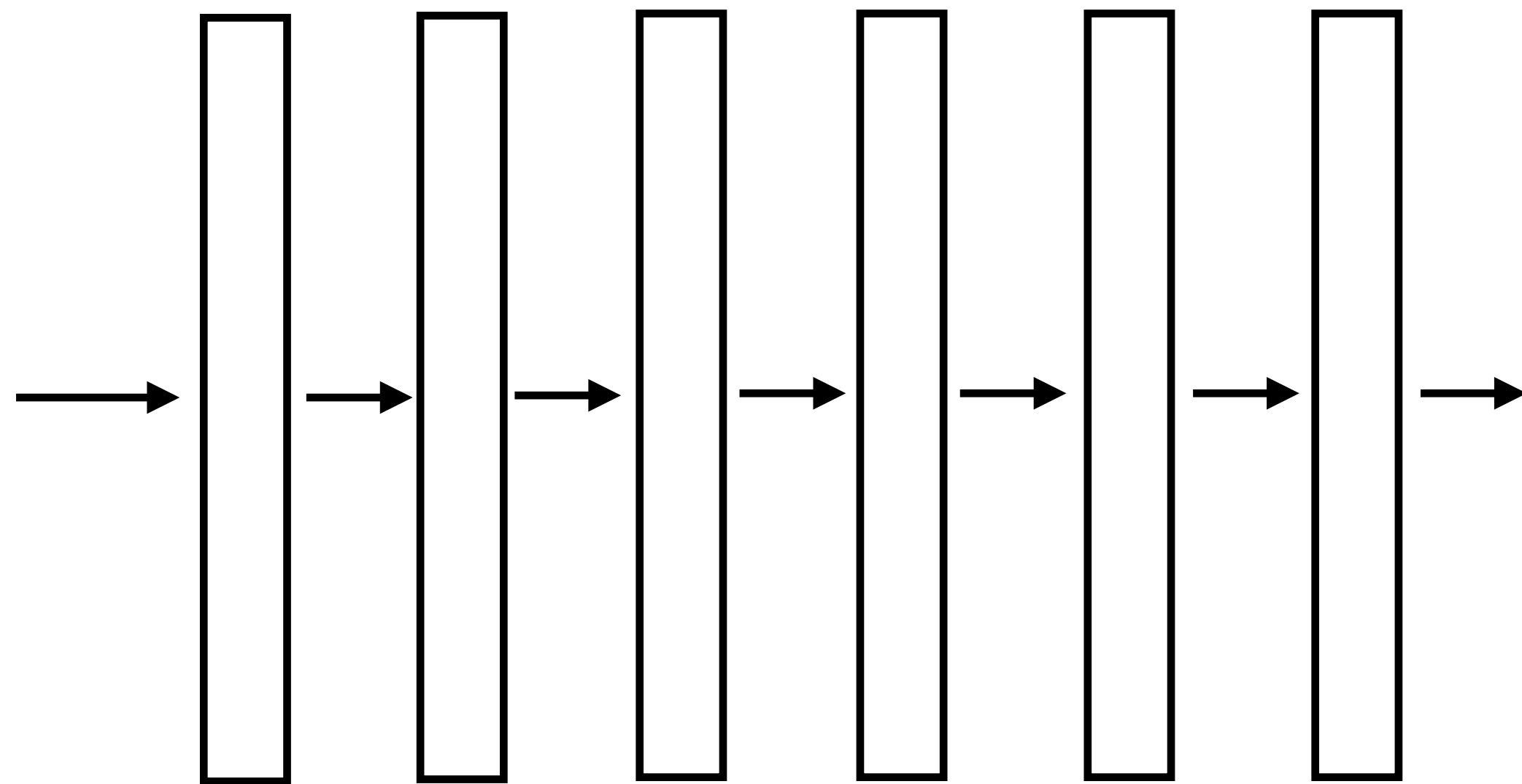
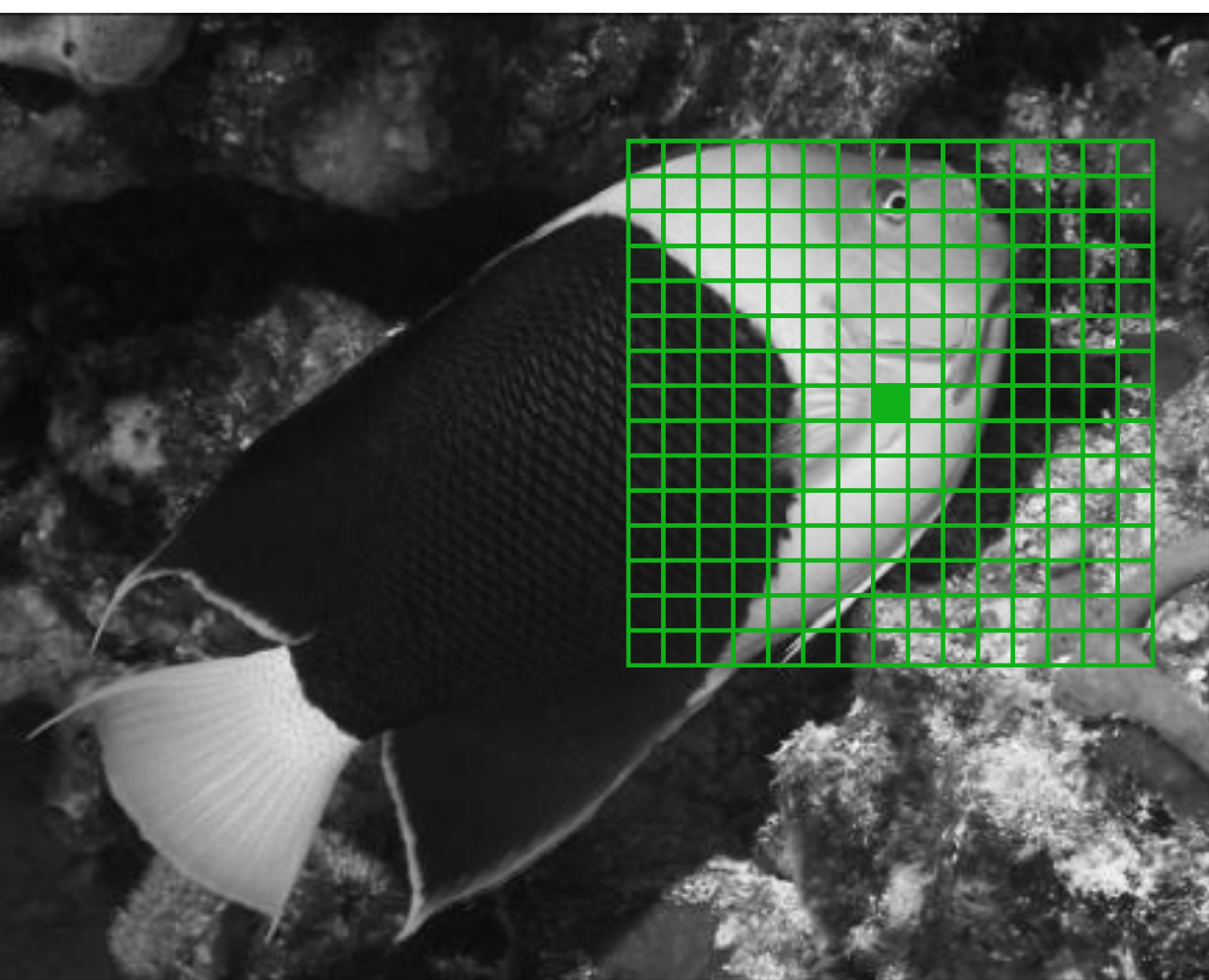
...



# Image classification —> Pixel classification

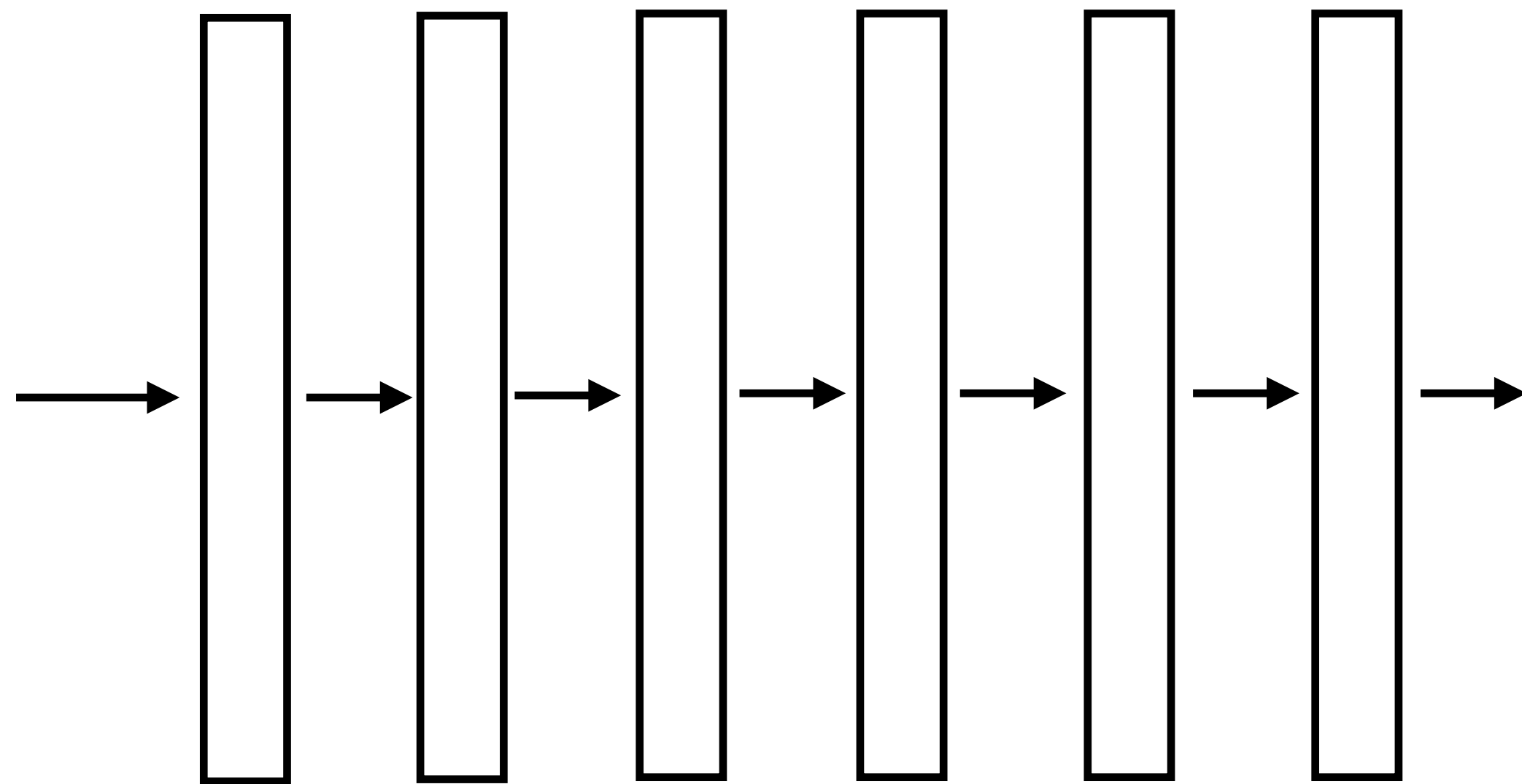
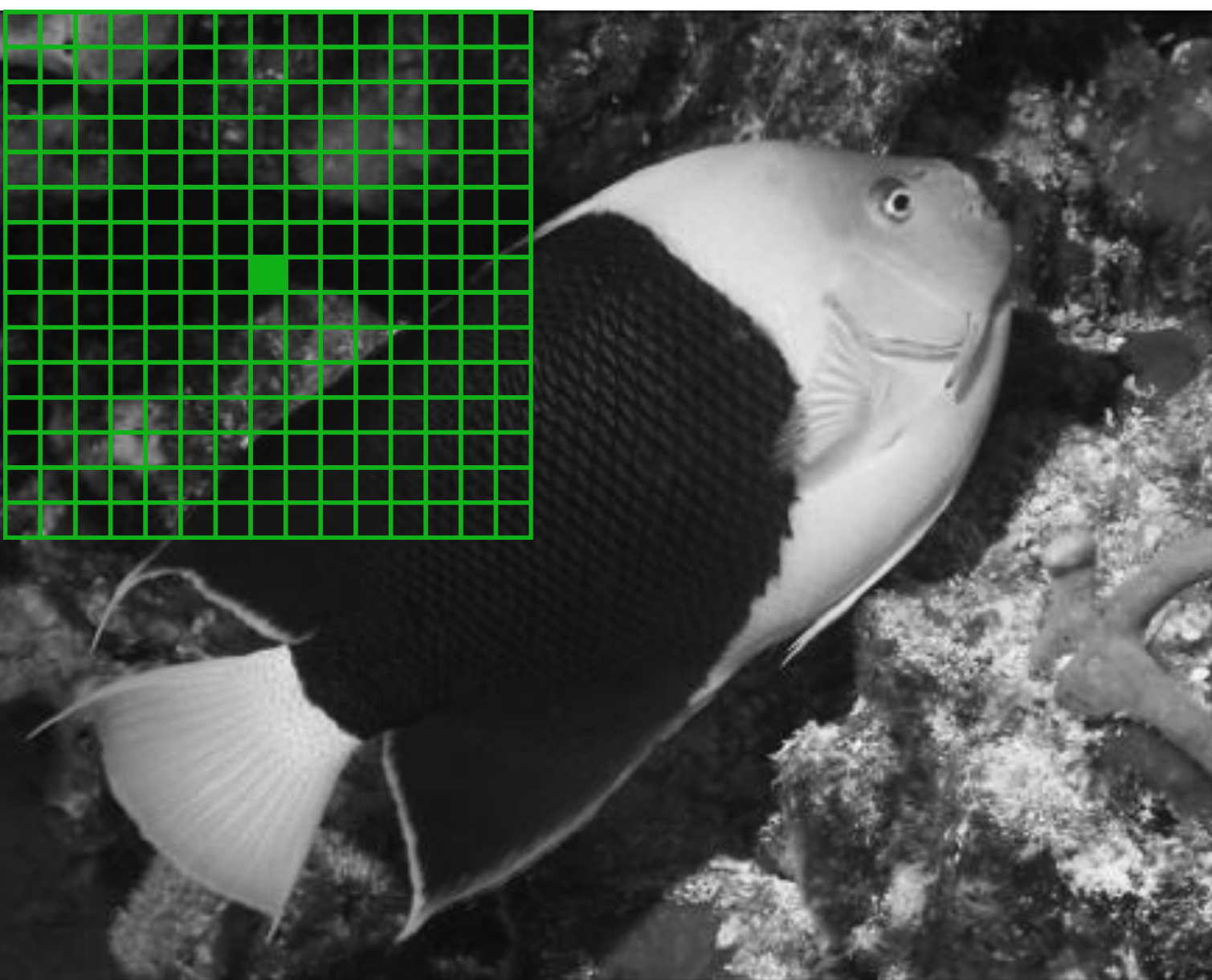


# Image classification $\longrightarrow$ Pixel classification





# Image classification $\longrightarrow$ Pixel classification



# Model

## Recipe for deep learning in a new domain

1. Transform your data into numbers (e.g., one-hot vectors)
2. Transform your goal into an numerical measure (e.g., cross-entropy loss)
3. Use a generic optimizer (SGD) and an standard architecture (e.g., CNN) to solve the learning problem



# 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, roughly in order

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

# Optimization

**we are still learning** how to optimize deep nets

much progress is being made but it's nevertheless a dark forest

explore if you like, but balance exploration by exploitation of a known good setting, or the closest setting you can find



# 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

# 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



# Optimization

the **learning rate** and the **batch size** are more-or-less the cardinal hyperparameters

- choose the learning rate on a small set (see [Bottou](#))
- simplify your life and **use a constant rate**, that is, no schedule until everything else is figured out
- schedule according to epochs (== number of passes through the data), not number of iterations of SGD
- set batch size to be as large as will fit in memory

# Optimization

**clear gradients** after each iteration or else they accumulate (in Pytorch)

remember `opt.zero_grad()`!

remarkably, with adaptive optimizers and enough time a model can still learn if the gradients are never cleared out... but it's really sensitive



# 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

# 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



# Evaluation

switch to **evaluation mode** by `model.eval()` (Pytorch)

no, really

and check the mode by `model.training`

# Evaluation

**know the output!** metrics and summaries can obscure all kinds of issues

- inspect input/output pairs, across min/max/quartiles of the eval metrics
- keep an eye on the output and loss for a chosen set across iterations to have a sense of the learning dynamics (could be the whole validation set)



Image #

Input

Ground Truth

L1

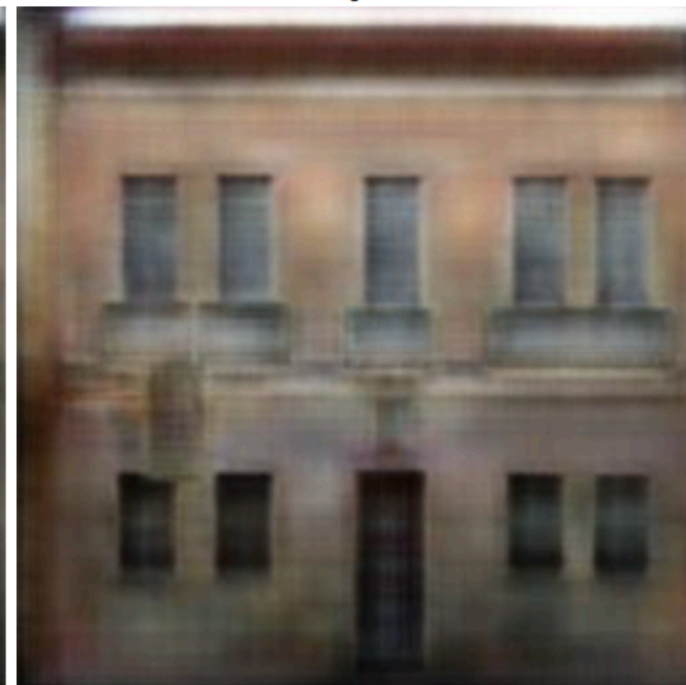
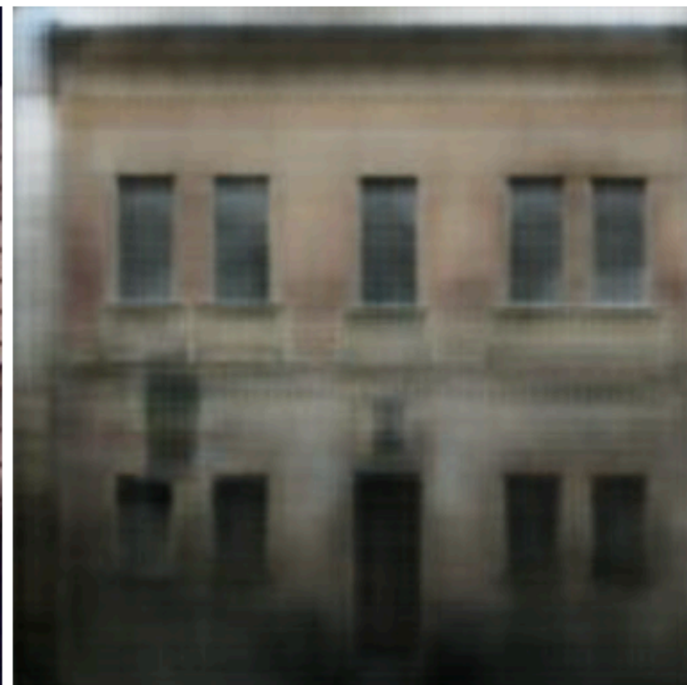
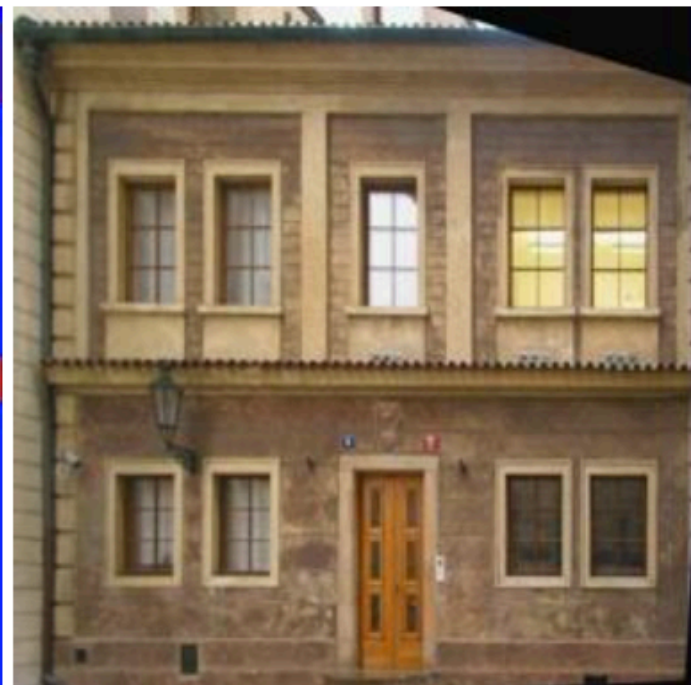
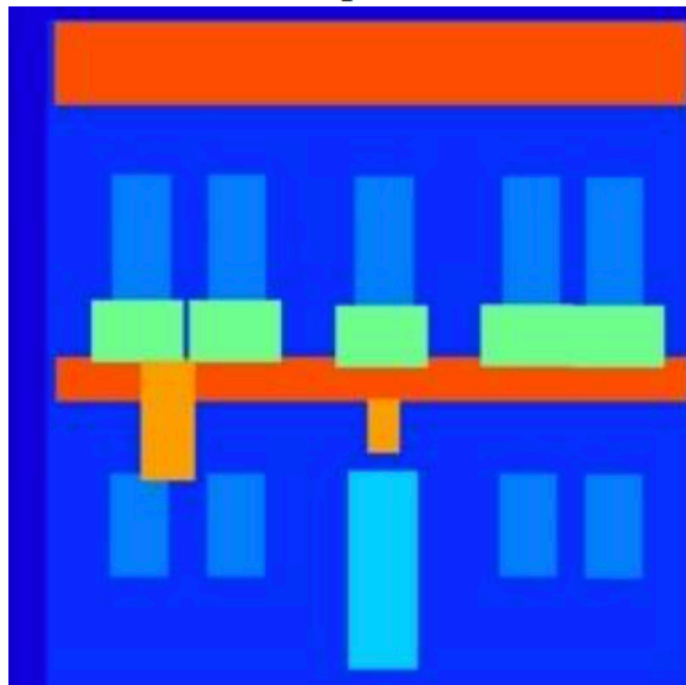
0layers

1layers

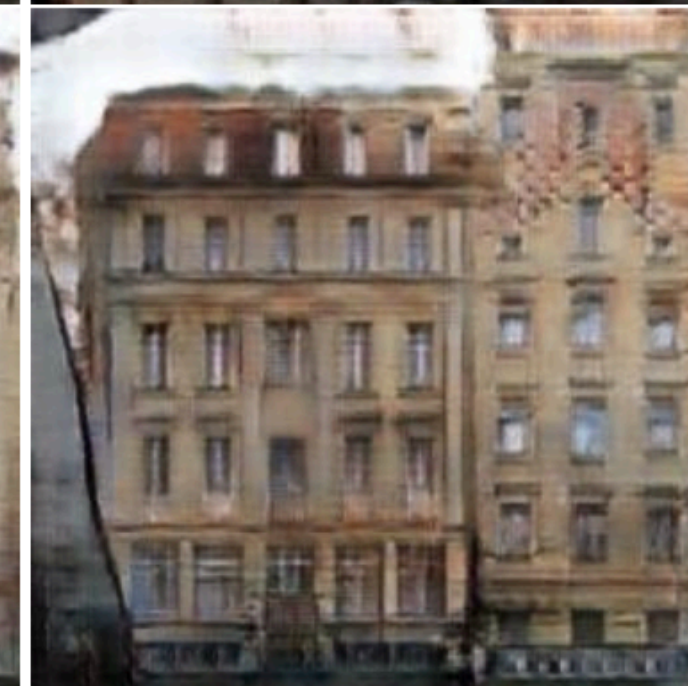
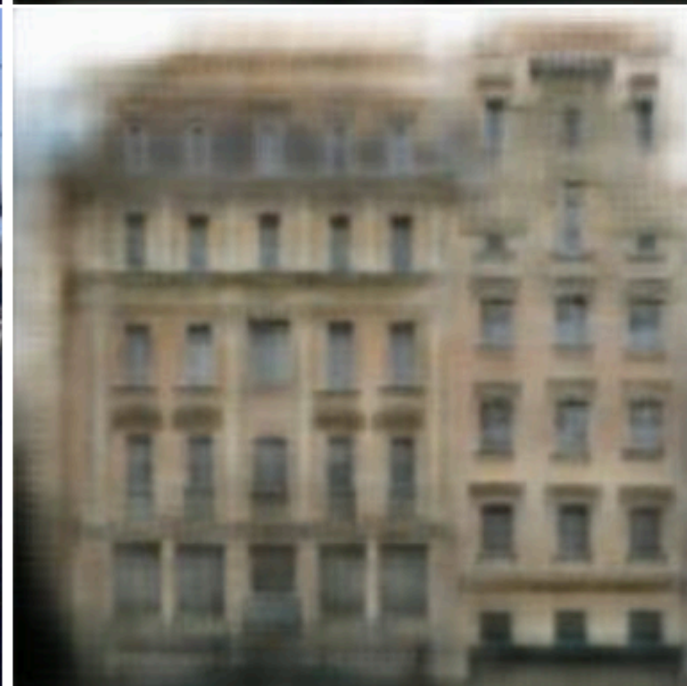
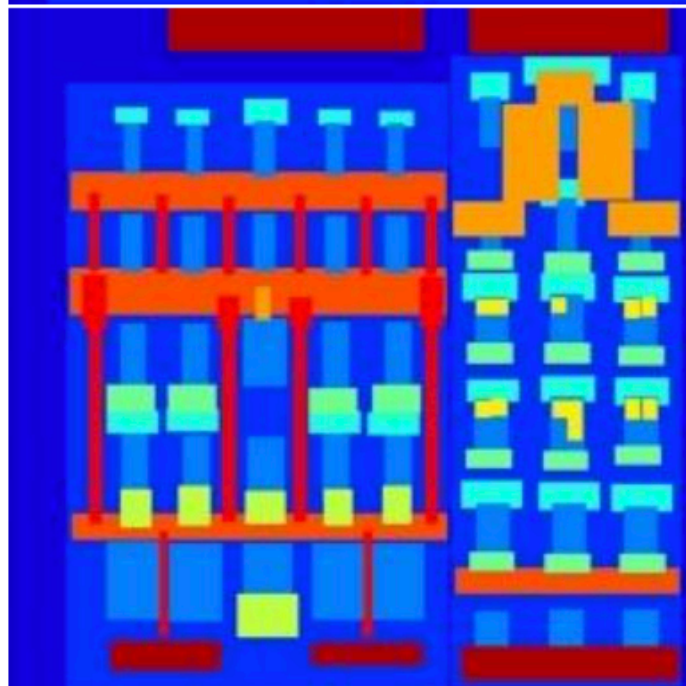
3layers

6layers

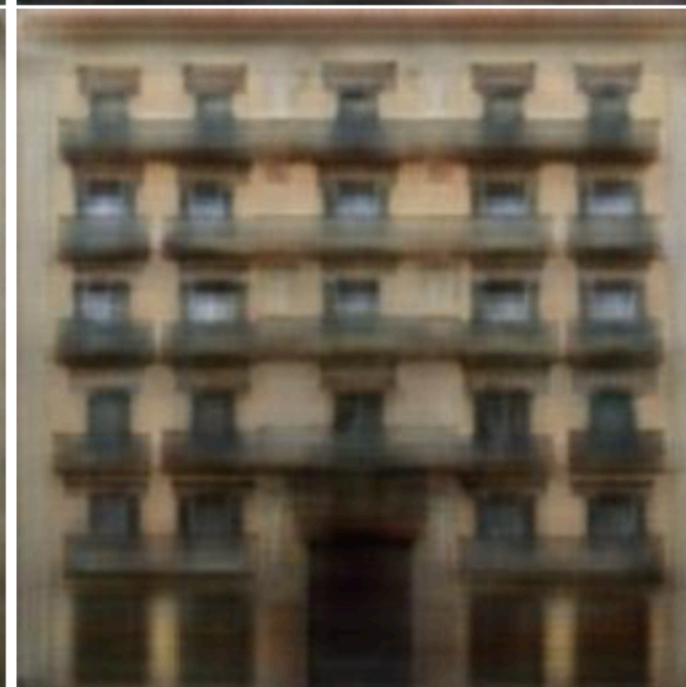
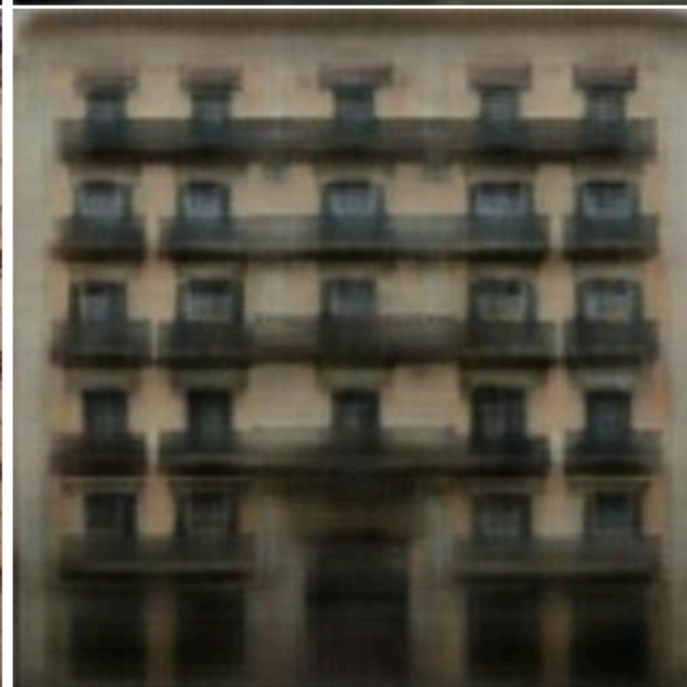
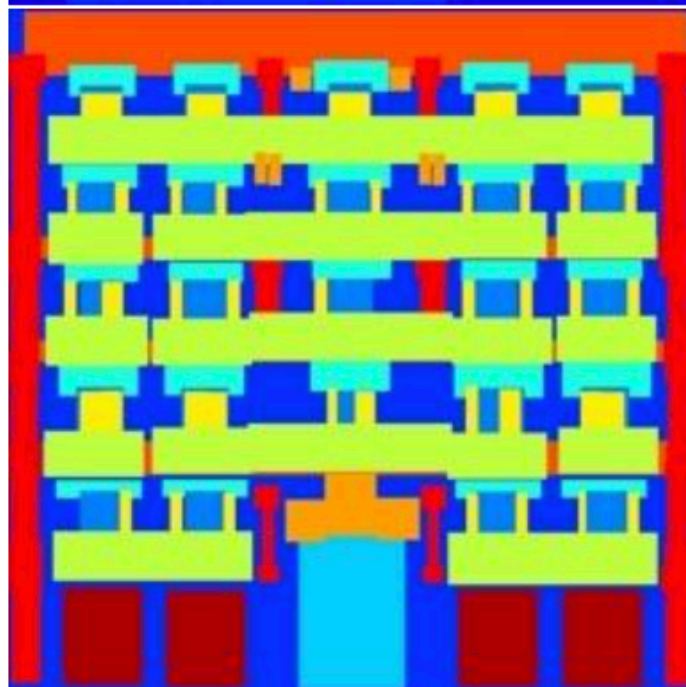
1



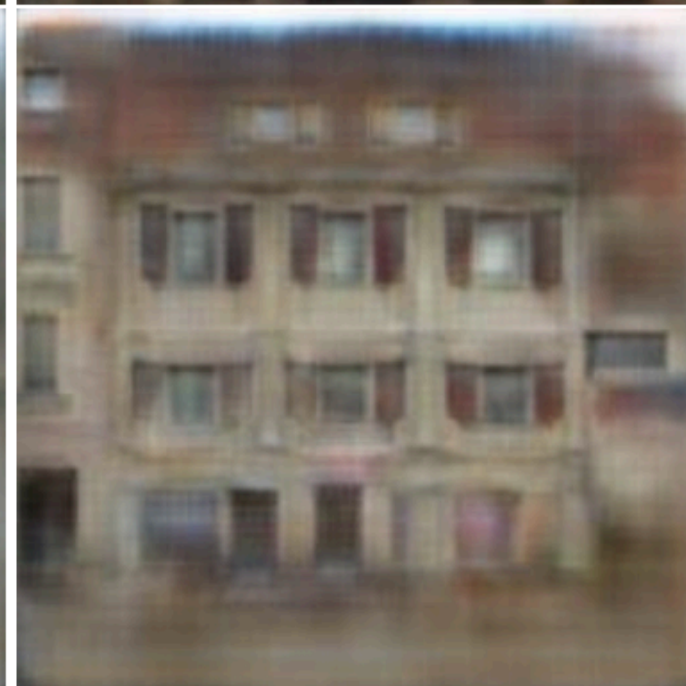
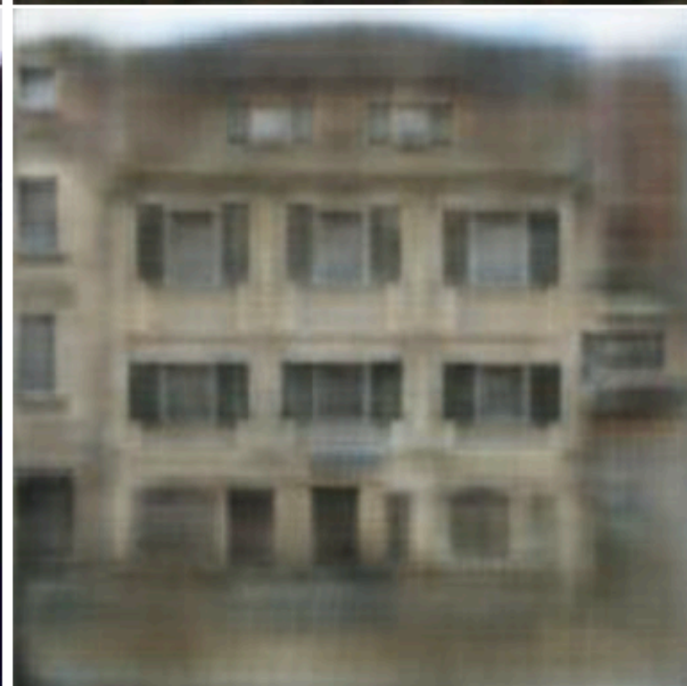
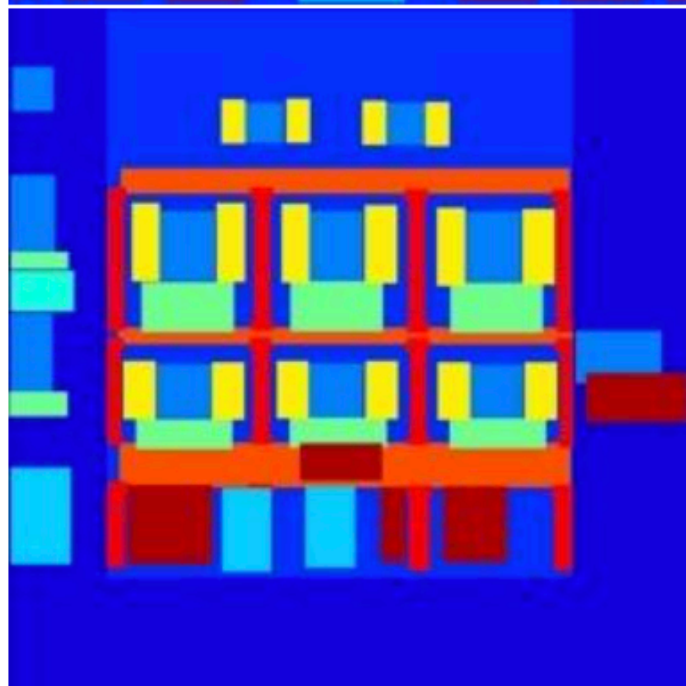
2



3



4





# Evaluation

**separate evaluation** from optimization

- save checkpoints at intervals and evaluate them offline
- there are the perils of nondeterminism, batch norm, etc when mixing training and testing
- plus it only slows down training



# Tuning

## **try the scientific method**

- change one thing at a time

## **not the computer scientific method**

- change everything every time

(joke courtesy Dave Patterson)

# Tuning

to tune hyperparameters (architectural design choices, optimizer parameters, etc) **random search** is better than grid search

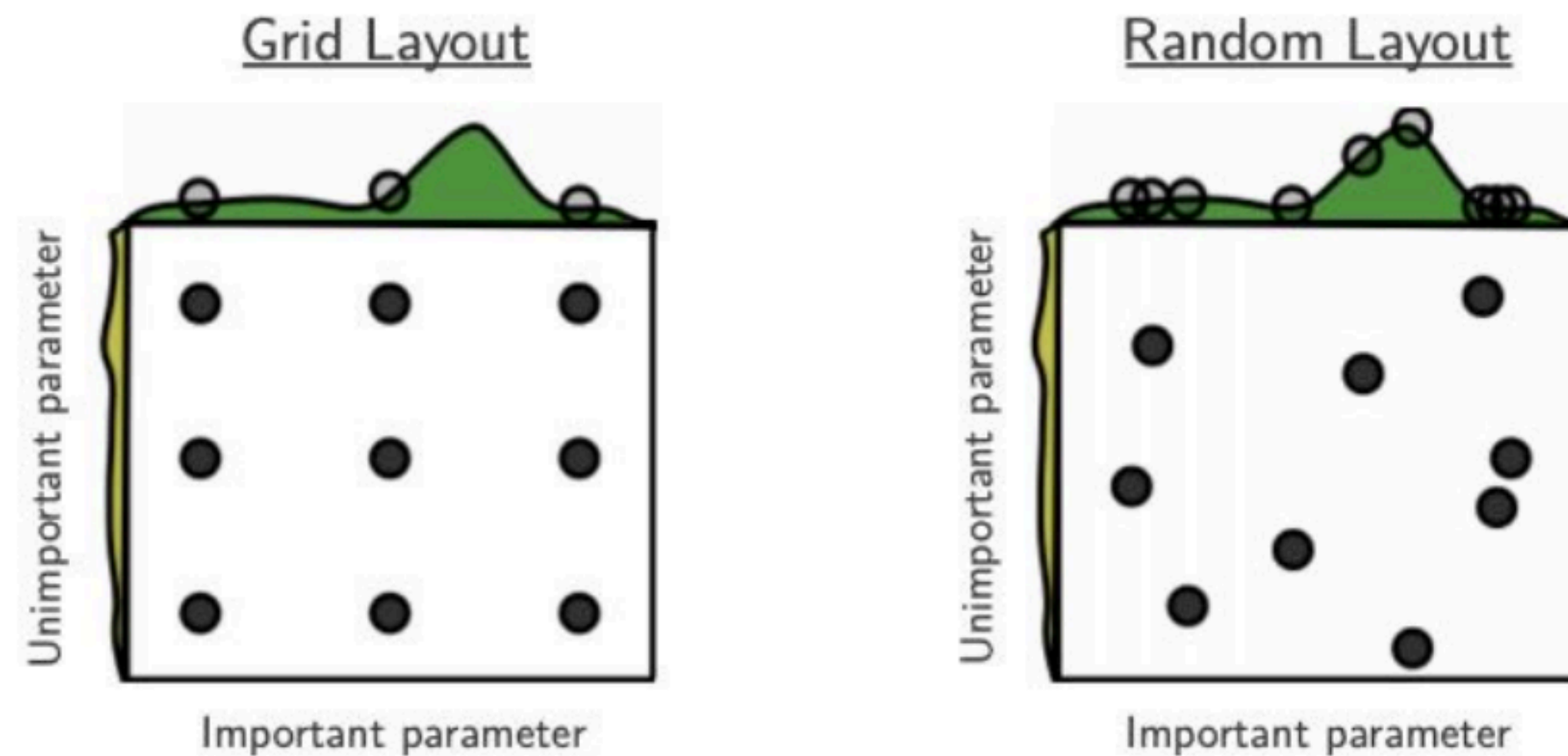


Figure 1: Grid and random search of nine trials for optimizing a function  $f(x,y) = g(x) + h(y) \approx g(x)$  with low effective dimensionality. Above each square  $g(x)$  is shown in green, and left of each square  $h(y)$  is shown in yellow. With grid search, nine trials only test  $g(x)$  in three distinct places. With random search, all nine trials explore distinct values of  $g$ . This failure of grid search is the rule rather than the exception in high dimensional hyper-parameter optimization.



# Execution

**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!

# Execution

**log everything!** especially arguments

```
import logging
def setup_logging(logfile):
    FORMAT = '%(asctime)s.%(msecs)03d] %(message)s'
    DATEFMT = '%Y-%m-%d %H:%M:%S'
    logging.root.handlers = [] # clear logging from elsewhere
    logging.basicConfig(level=logging.INFO, format=FORMAT, datefmt=DATEFMT,
                        handlers=[
                            logging.FileHandler(logfile),
                            logging.StreamHandler(sys.stdout)
                        ])
    logger = logging.getLogger()
    return logger

args = parser.parse_args()
log = setup_logging('log')
log.info(f"args: {vars(args)}")
```



# Testing 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>

# Testing and debugging

**check gradients** using finite difference methods

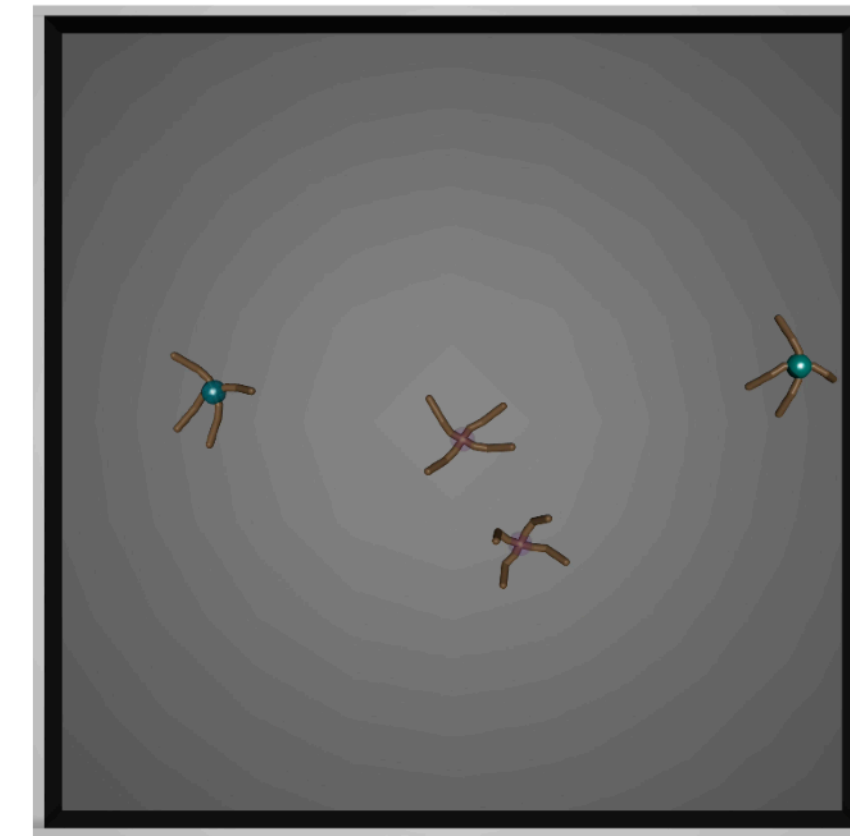
- [gradcheck](#) is the checker bundled into PyTorch
- [cs231](#) has gradient checking rules of thumb
- [Tim Vieira](#) has further tips for accurate and thorough checking



# 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

# 12. Mechanisms of Training and Running Neural Nets

- Data
- Model
- Optimization
- Evaluation, Execution, and Debugging



Lots of slides adapted from **Evan Shelhamer's** “DIY Deep Learning: Advice on Weaving Nets”