Intro to ML II

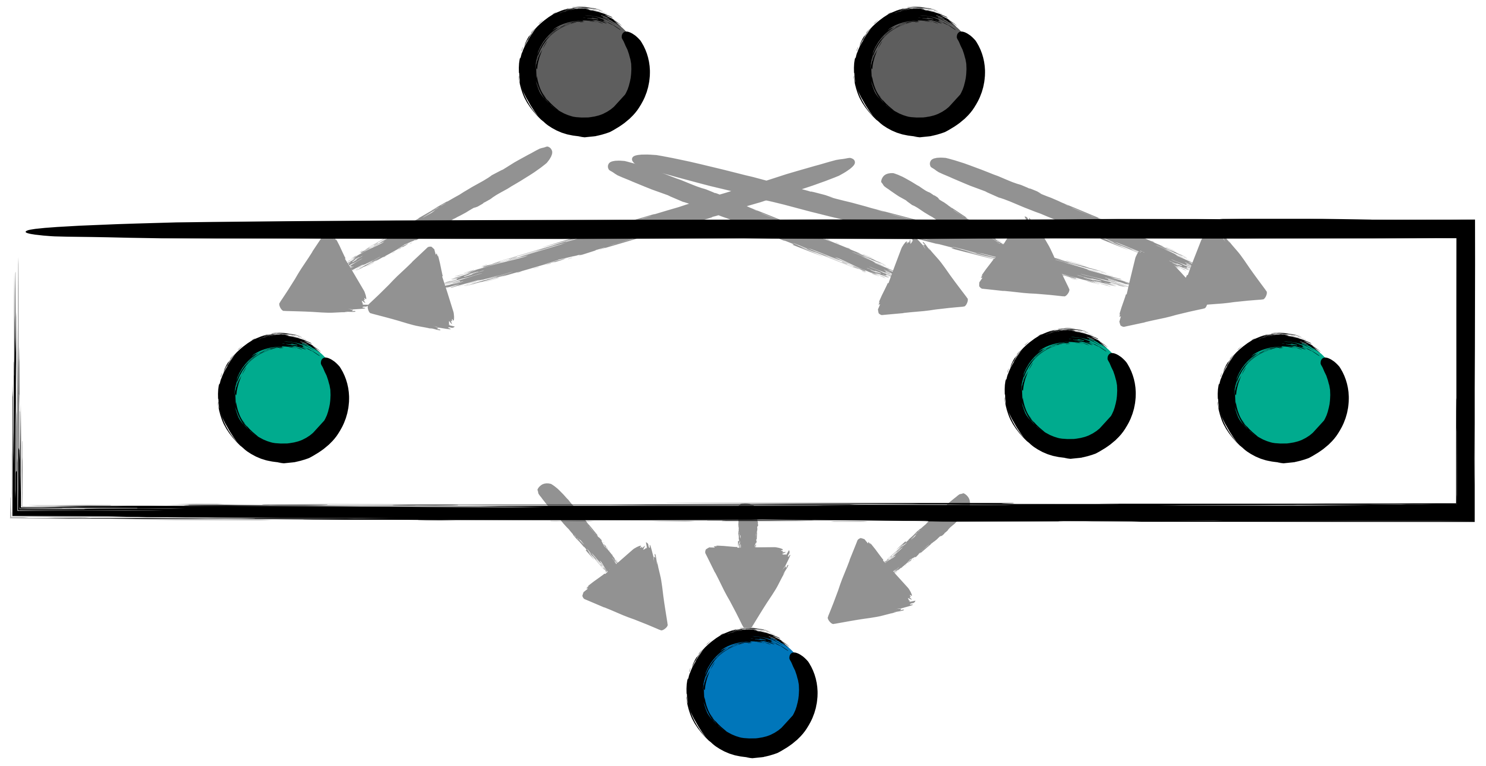
CERN School of Computing 2023

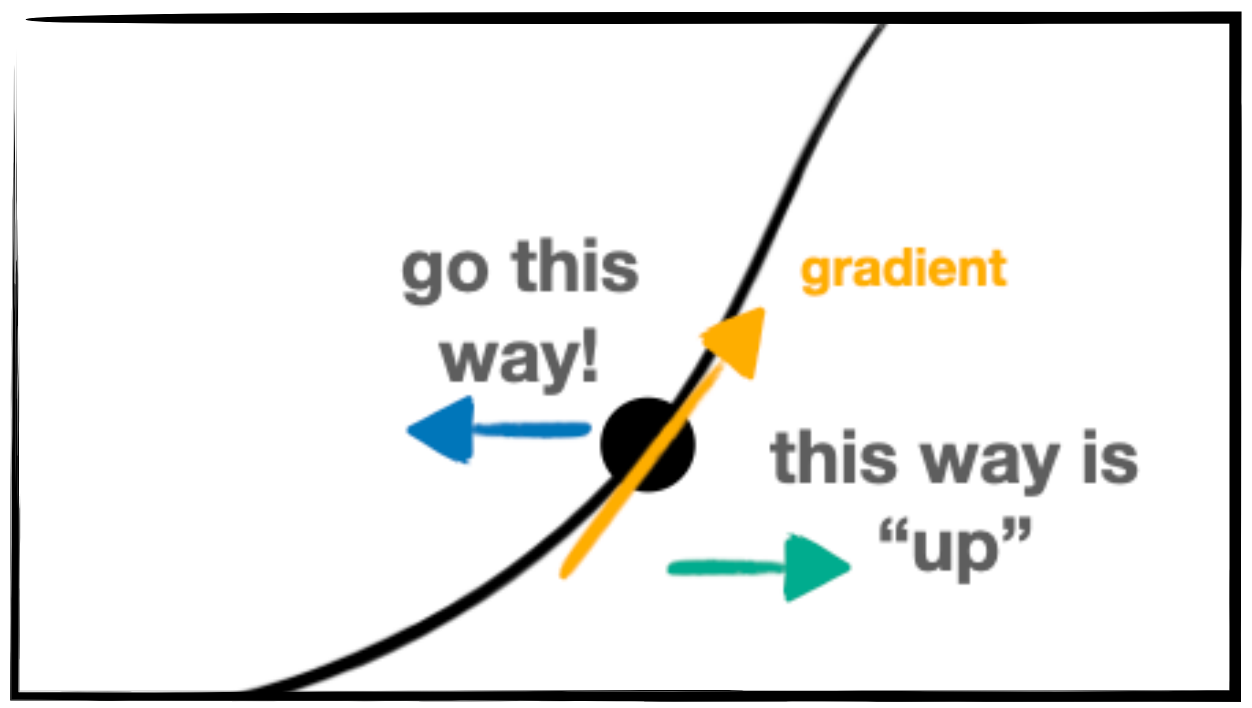
Lukas Heinrich, TUM

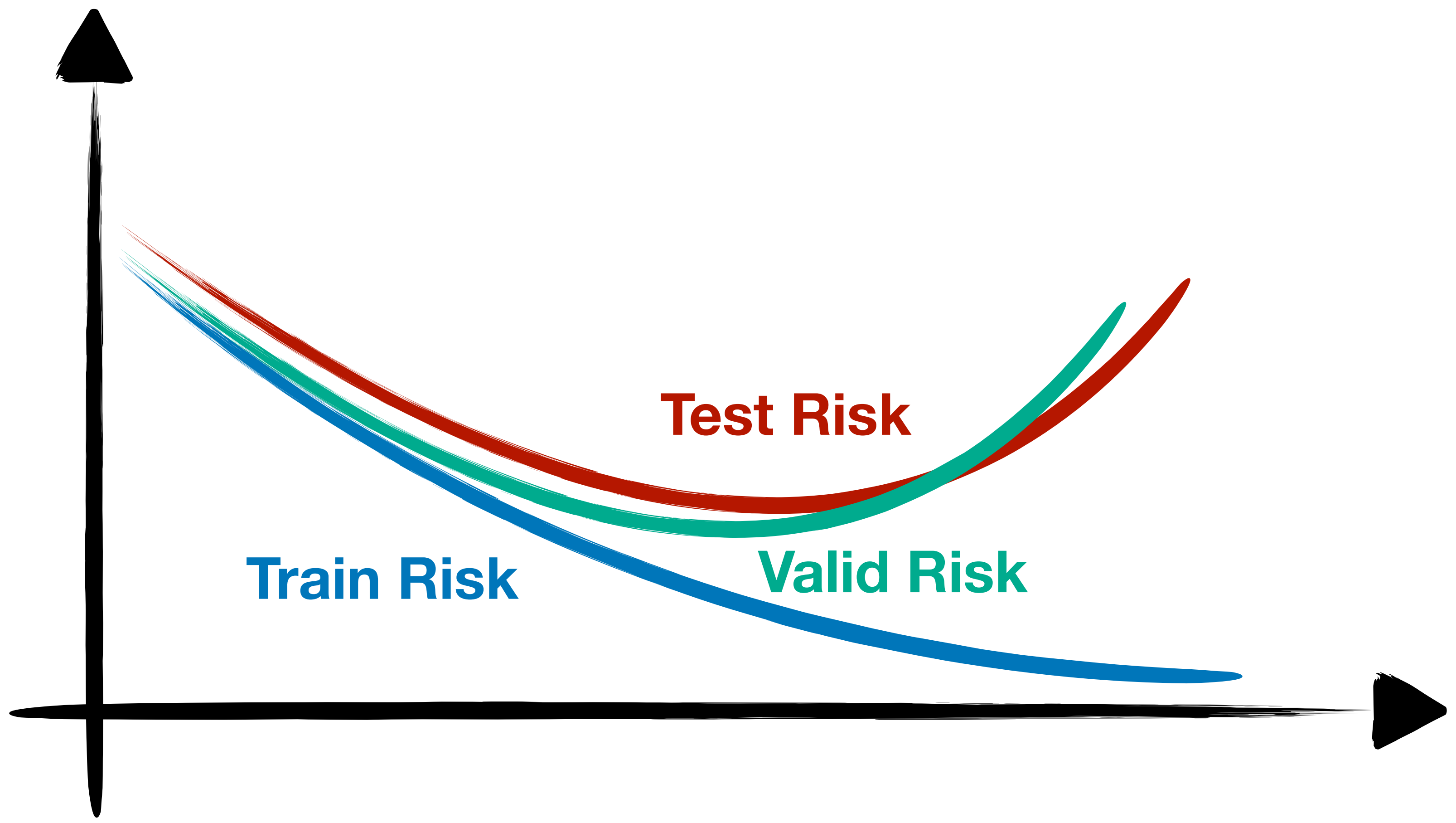
Supervised Learning

What we need

We have the general ingredients for learning







We need to now formulate the actual objectives for the tasks we're interested in

Let's start with supervised learning

Latent Concepts

We interpret the real world data we perceive / measure to be a realization of an “*underlying concept*”

We observe the data but the concept is “latent”

latent | ˈleɪt(ə)nt |

adjective

(of a quality or state) existing but not yet developed or manifest; hidden or concealed: *they have a huge reserve of latent talent*.

Latent Concepts

We interpret the real world data we perceive / measure to be a realization of an "*underlying concept*"

concept: "cat"

realization: pixel values in image

Latent Concepts

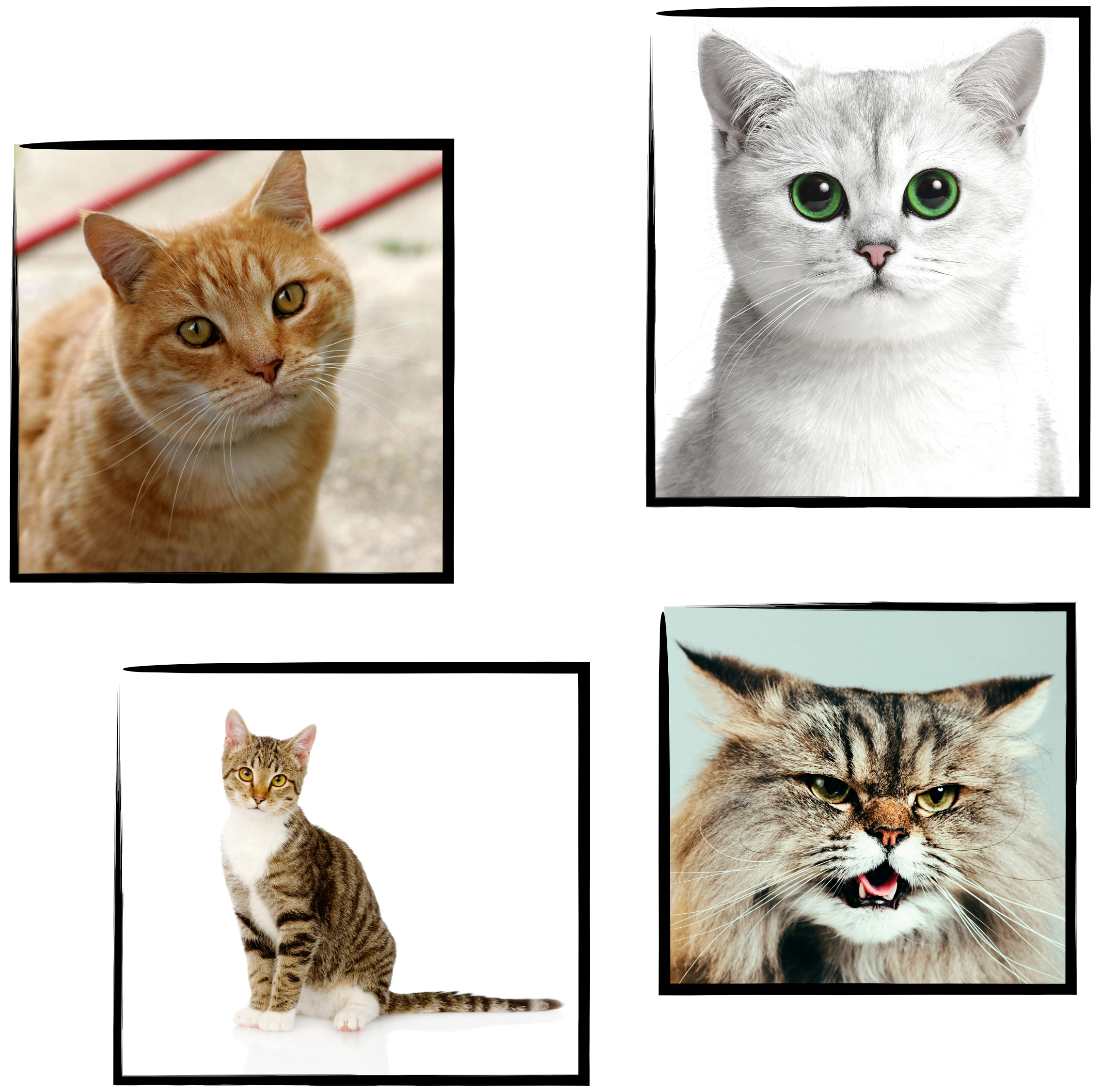
We interpret the real world data we perceive / measure to be a realization of an "*underlying concept*" (*or* "*label*")

concept: "true weight"

realization: reading on the scale

Latent Concepts

In statistical learning, we assume that concept z and realization x are linked through a a **conditional probability:**

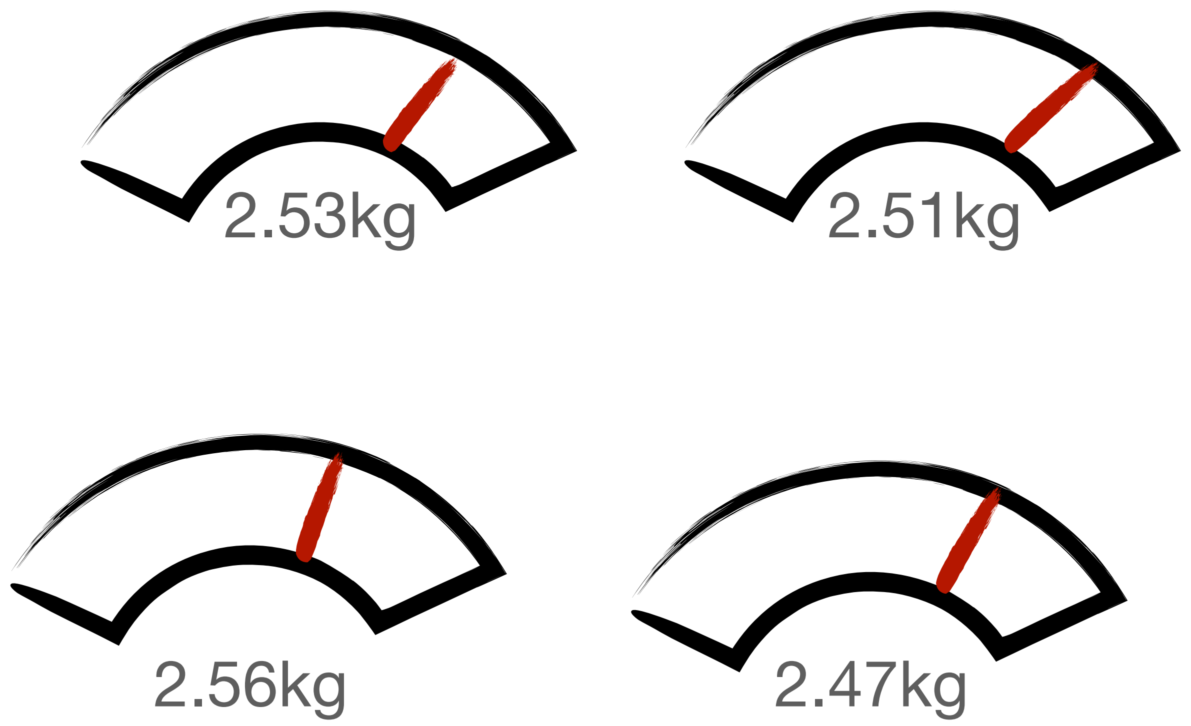


$$x \sim p(x \mid z) \qquad z = \text{cat}$$

many realizations *true value*

Latent Concepts

In statistical learning, we assume that concept z and realization x are linked through a a **conditional probability:**



$$x \sim p(x \mid z)$$

$$z = 2.50 \text{ kg}$$

many realizations

true value

Inference

Classic Goal in Statistics: try to find out ("infer") the latent values given the observed values, i.e. the data x

Bayes' Theorem

$$\underbrace{p(z \mid x)}_{\textit{posterior}} = \frac{\overbrace{p(x \mid z)}^{\textit{likelihood}}}{\underbrace{p(x)}_{\textit{evidence}}} \overbrace{p(z)}^{\textit{prior}}$$

Inference by another name

We name inference based on the **type** of the latent variable

“cat”

$$z \in \{z_0, z_1 \ldots z_n\}$$

finite set = “Classification”

2.50

$$z \in \mathbb{R}$$

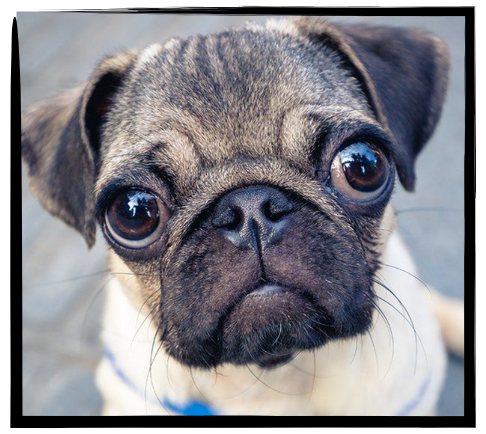
real values: “Regression”

Statistics vs Machine Learning

To do **standard statistics,** we'd need to know what the true data-generating process is $p(x \mid z), p(z)$, but we don't!

x z

"cat"



"dog"

We want:

$$p(\text{animal} \mid \text{image})$$

But we don't even have:

$$p(\text{image} \mid \text{animal}) = ?$$

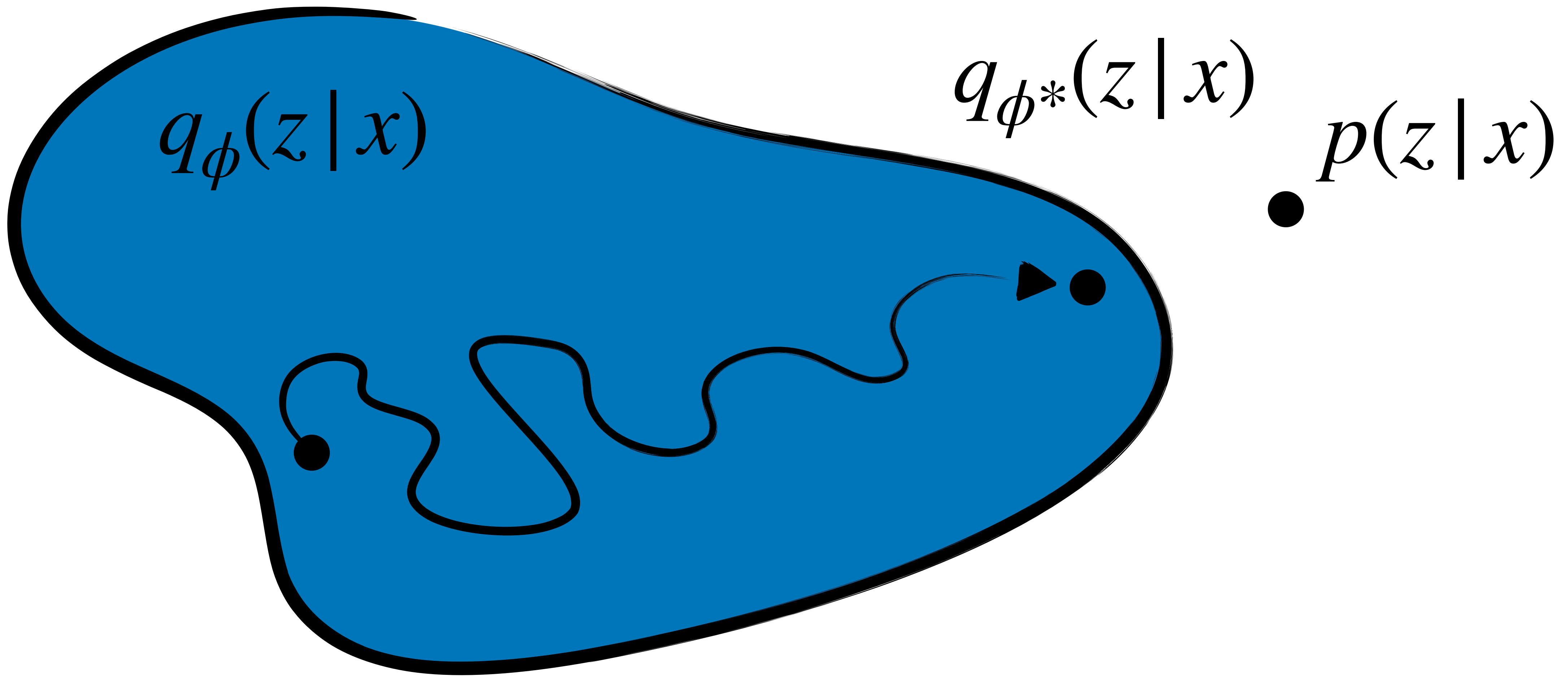
$$p(\text{animal}) = ?$$

Solution

Apply "learning as search". If we don't know $p(z|x)$ maybe we can approximate it?

Look for the best candidate **family of candidate distributions** $q_\phi(z|x)$

"Variational Inference"

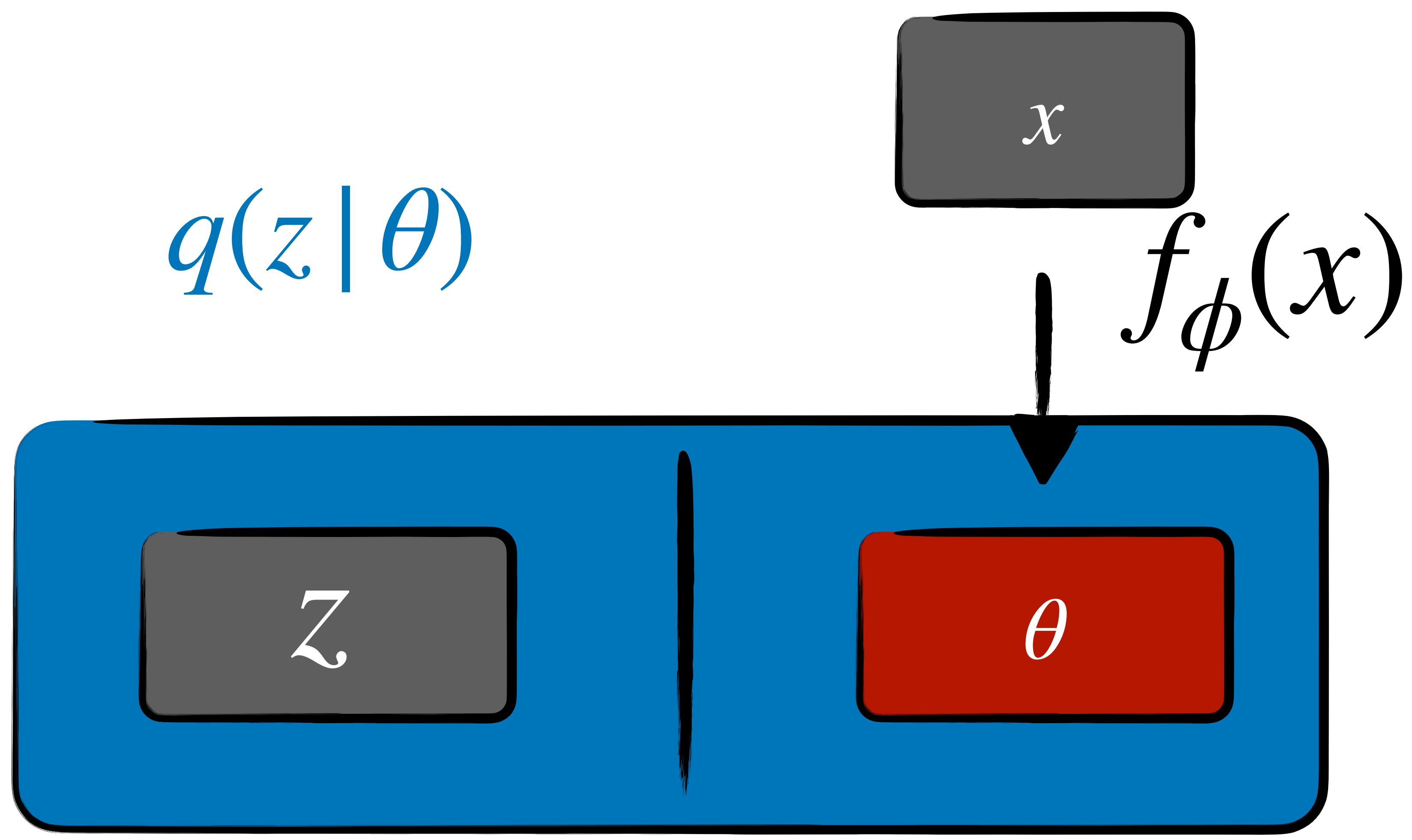


From Functions to Densities

To define a family of distributions, we can use

- well-known densities (e.g. Gaussian, …)
- compute parameters as functions of data: $f_\phi(x)$

$\rightarrow q_\phi(z \mid x) = q(z \mid \theta = f_\phi(x))$



Examples

Distribution Type

Parameters as function of data

Regression

Gaussian

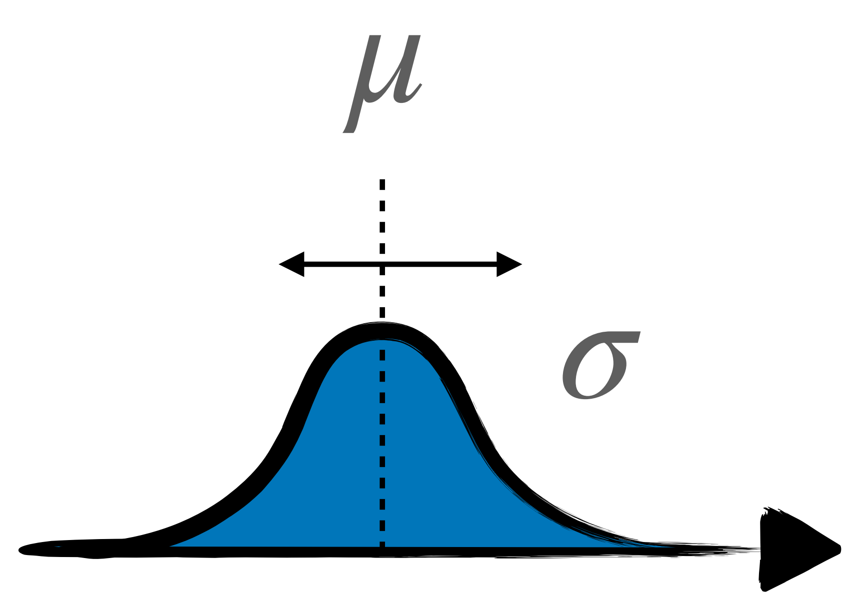
$\mathcal{N}(z \mid \mu, \sigma)$

Mean

$\mu = f_\phi(x)$

Variance

$\log \sigma = g_\phi(x)$

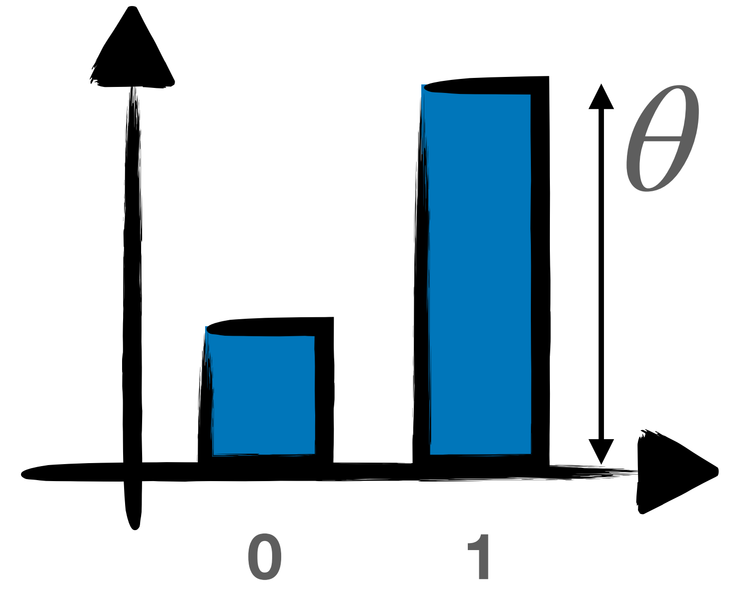


Bernoulli

$\mathrm{Bern}(z \mid \theta)$

Probability of z=1

$\theta = f_\phi(x)$

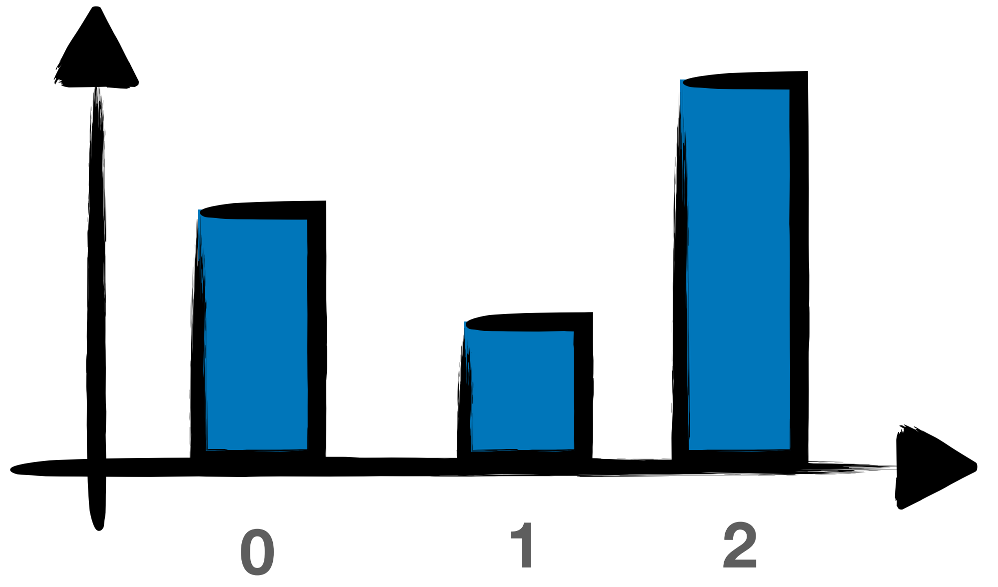


Classification

Categorical

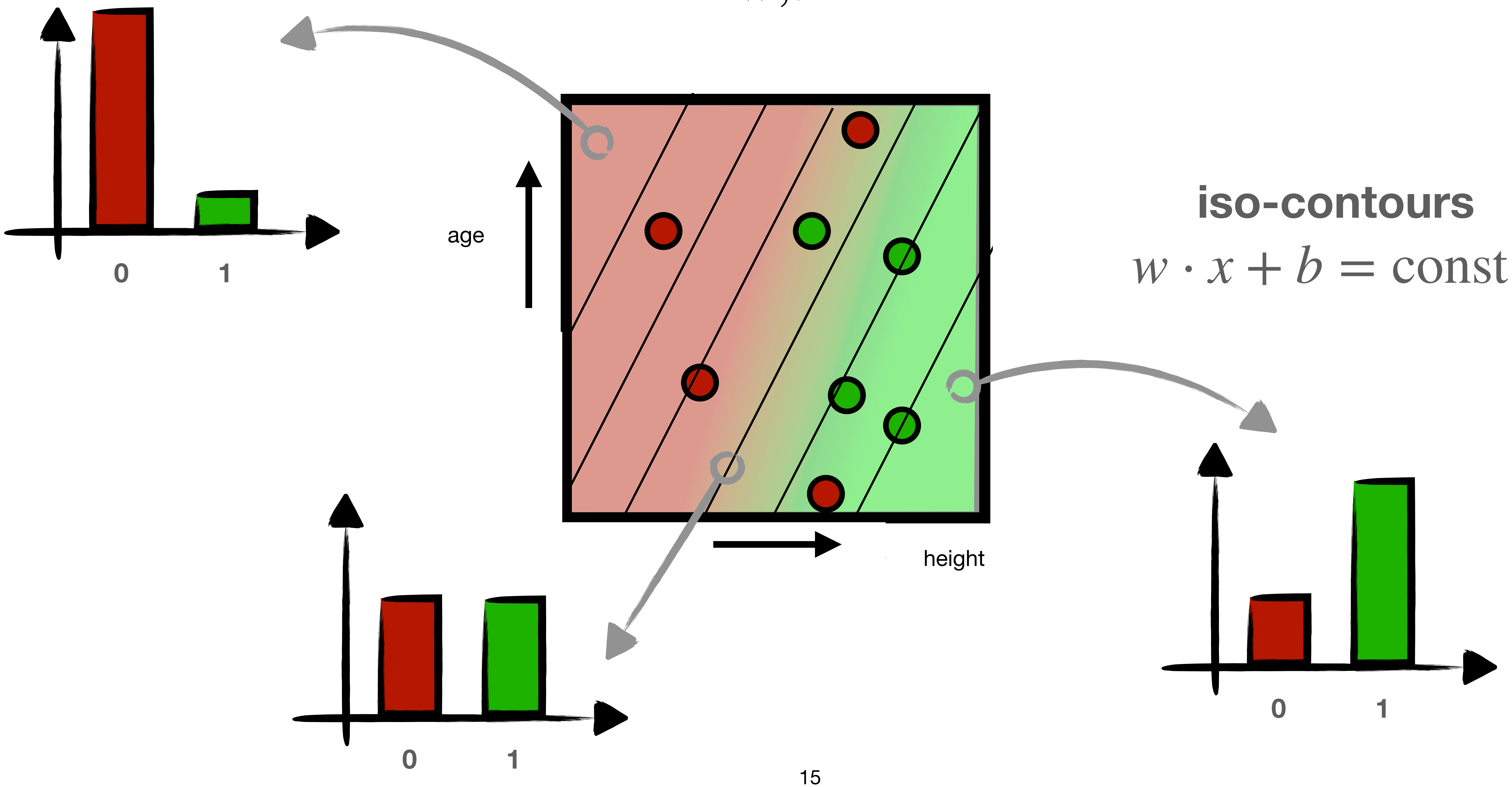
$\mathrm{Cat}(z \mid \{p_1 \ldots p_n\})$

$$\{p_i\}, \text{ s.t. } \sum_i p_i = 1$$



Soft Perceptron was our first example

$$q_{w,b}(z \mid x) = \mathrm{Ber}(z \mid \sigma(wx + b))$$



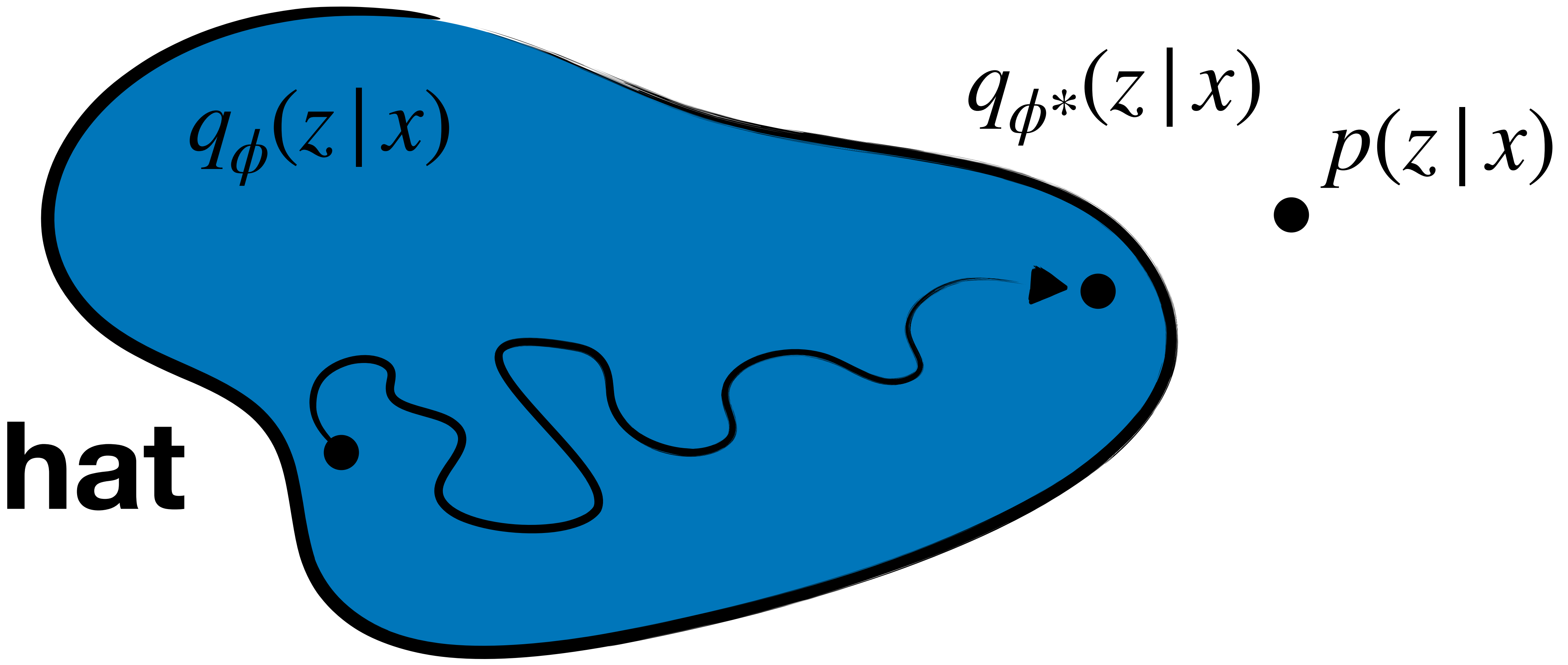
iso-contours

$$w \cdot x + b = \mathrm{const}\,.$$

Defining an Objective

Our goal is to approximate $p(z \mid x)$. Intuitively, we some notion of **distance between distributions** $d(p, q_\phi)$

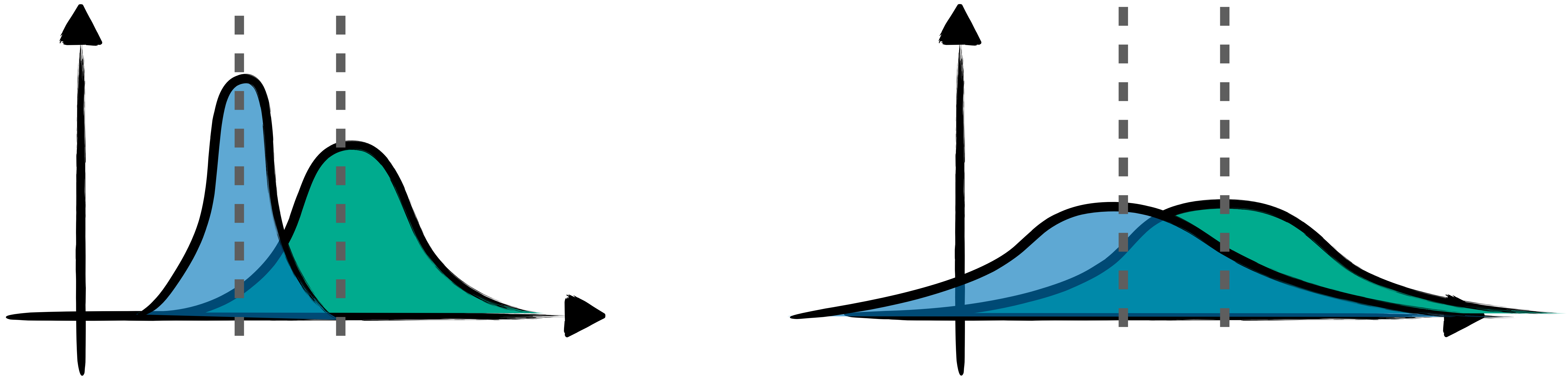
Learning as minimization of that distance



$$\phi^* = \text{argmin}_\phi d(p, q_\phi)$$

Distances between Distributions

Distributions are extended objects, not single point



same distance in means but which pair is "closer"?

A common choice:
"KL Distance"

$$D_{\mathrm{KL}}(p\,||\,q) = \int \mathrm{d}x\; p(x)\;\log\frac{p(x)}{q(x)}$$

Kullback-Leibler Divergence

A natural objective

So a Natural Objective: get good inference performance across all the possible data we might encounter.

I.e. minimize:

$$L(\phi) = \mathbb{E}_p(x) D_{\mathrm{KL}}(p(z \mid x) \,||\, q_\phi(z \mid x))$$

Average over all (likely) data

True Inference Solution

Our Approximation

A natural objective

Ok, but it seems like to compute the objective we already need to know the answer?

$$L(\phi) = \mathbb{E}_{p(x)} D_{\mathrm{KL}}(p(z \mid x) \,||\, q_\phi(z \mid x))$$

$$= \mathbb{E}_{p(x)} \mathbb{E}_{p(z|x)} \log \frac{p(z \mid x)}{q_\phi(z \mid x)}$$

A natural objective

Amazingly it all drops out! In the end, we get something we can estimate purely from data pairs (x_i, z_i) - which we have

$$L(\phi) = \mathbb{E}_x \mathbb{E}_{p(z|x)} \log \frac{p(z \mid x)}{q_\phi(z \mid x)} = - \mathbb{E}_{p(x,z)} \log q_\phi(z \mid x)$$

This is called the "Cross-Entropy" Loss and is most used for supervised learning

Alternative Names

CE sometimes hides under another name / formula:

Gaussians: "Mean Squared Error" (MSE)

$$\mathbb{E}_{p(x,z)}(z - \mu_\phi(x))^2$$

Binary Classification: "Binary Cross Entropy" (BCE)

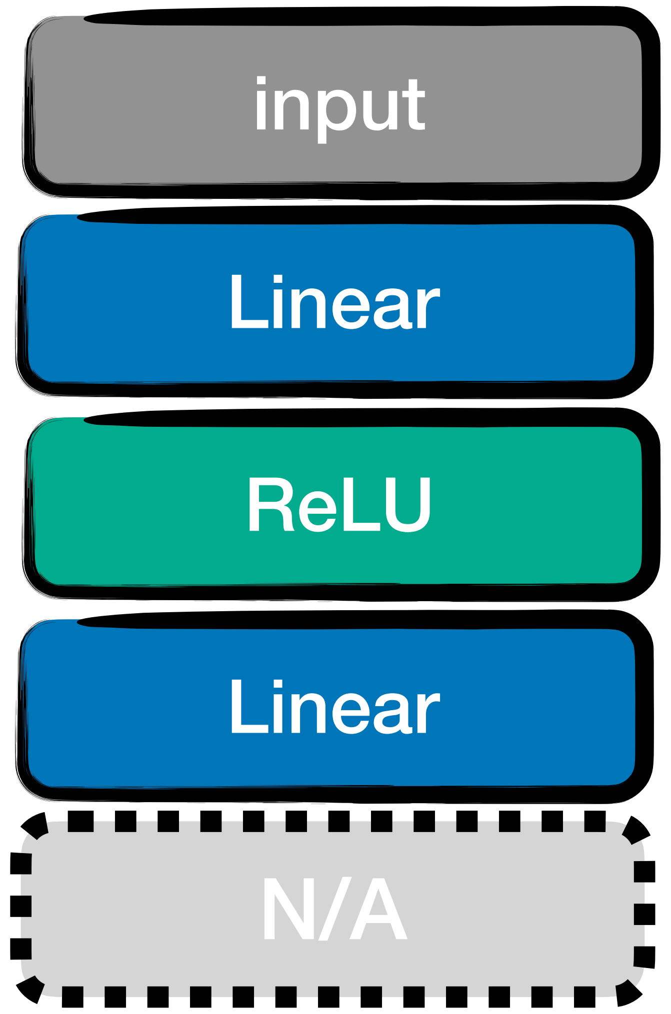
$$\mathbb{E}_{p(x,z)}[z \log \theta_\phi(x) + (1 - z)\log(1 - \theta_\phi(x))]$$

Output Activations

For UFA the type of non-linearity was irrelevant, now we need to at least be careful with the **output activation**

Regression

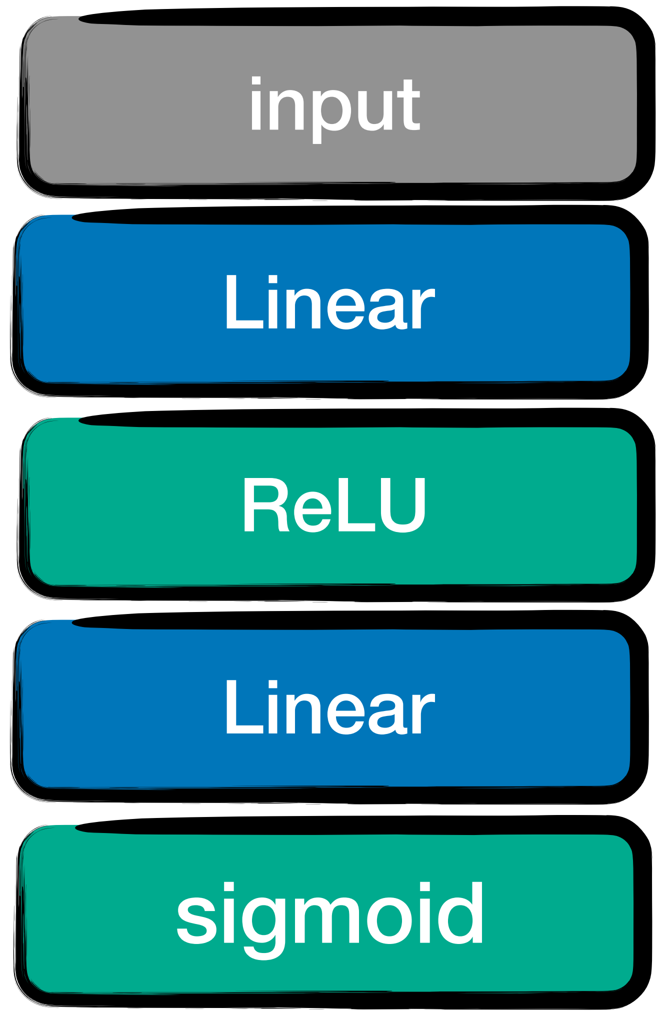
$$\mu_\phi(x) \in \mathbb{R}$$



No activation!

Binary Classification

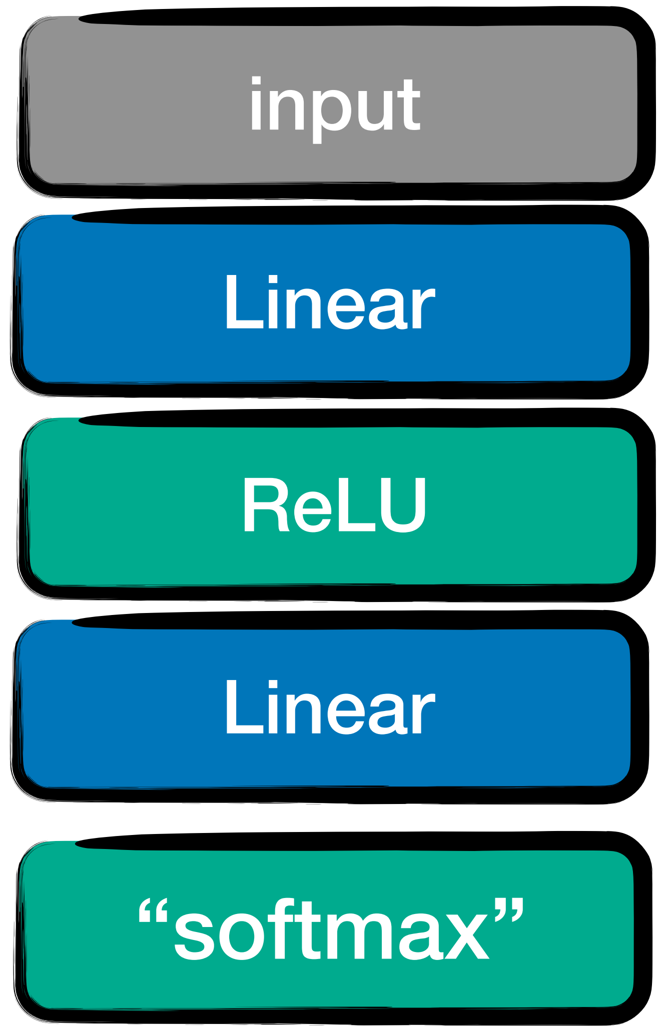
$$\theta(x) \in [0,1]$$



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

Multi-class Classification

$$p_i(x) \geq 0 \text{ s.t. } \sum p_i = 1$$



$$\text{softmax}(x) = \frac{e^{x_i}}{\sum_i e^{x_i}}$$

Deep Learning
Inductive Bias & Friends

Two issues with shallow networks

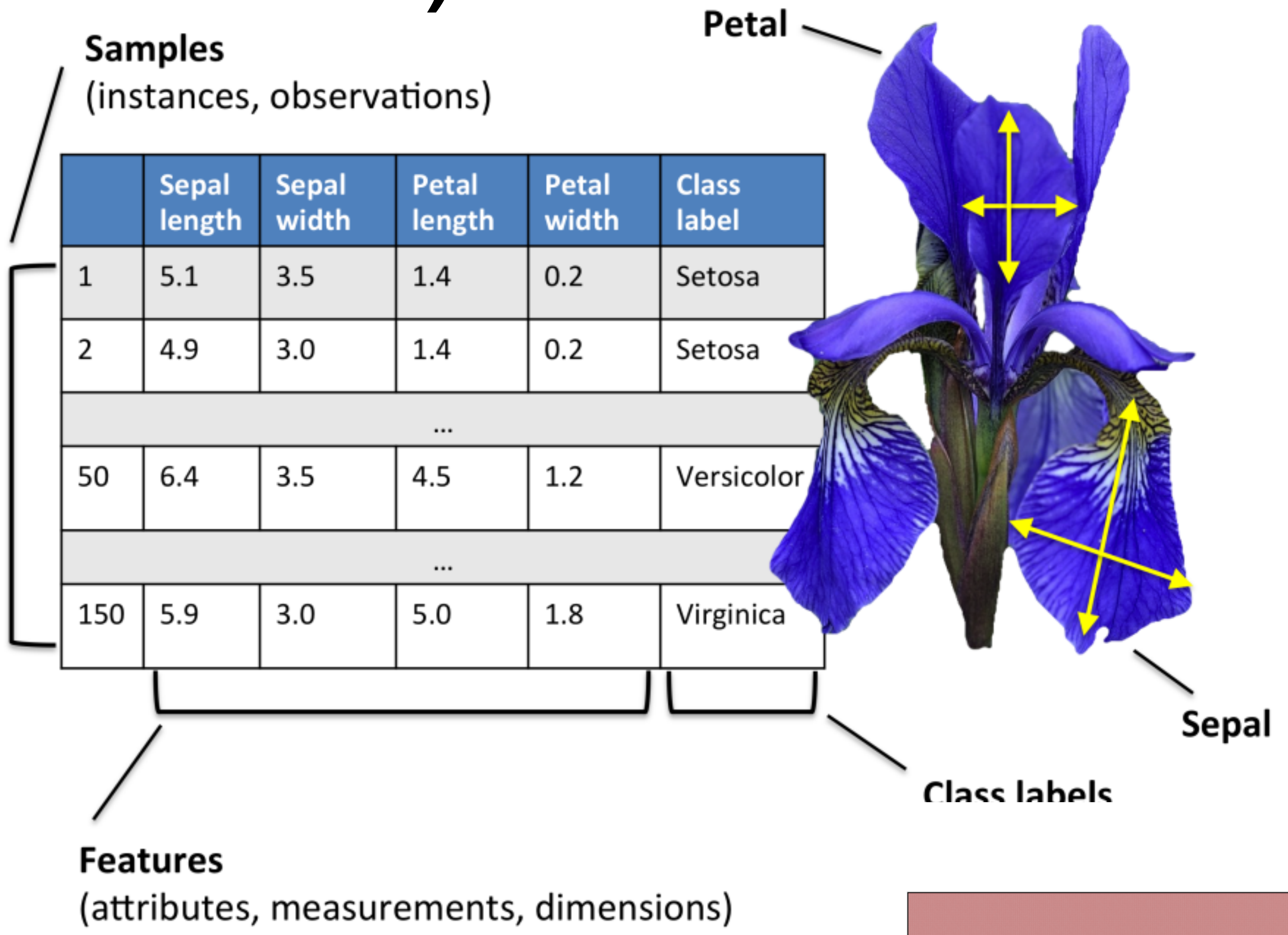
Shallow Networks are great (universal, even!),
but there are two issues:

- to *actually* model complex functions you need a lot of neurons, and i.e. parameters
- sometimes **we know quite a bit** about our target function, should we really search in a **universal space**?

Deep Learning

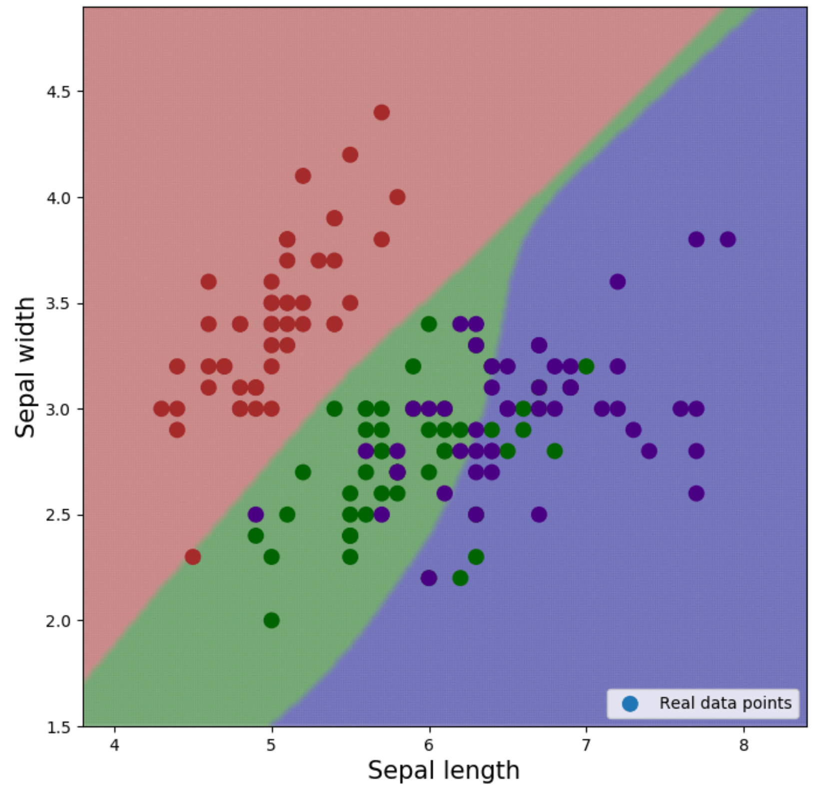
A lot of classic machine learning was done on highly preprocessed data (“engineered features”)

Iris Versicolor Iris Setosa Iris Virginica



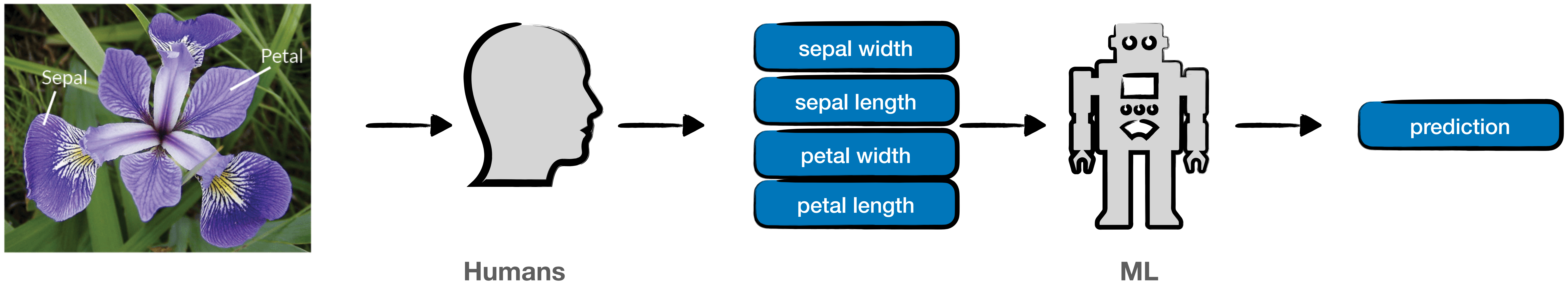
	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
			...		
50	6.4	3.5	4.5	1.2	Versicolor
			...		
150	5.9	3.0	5.0	1.8	Virginica

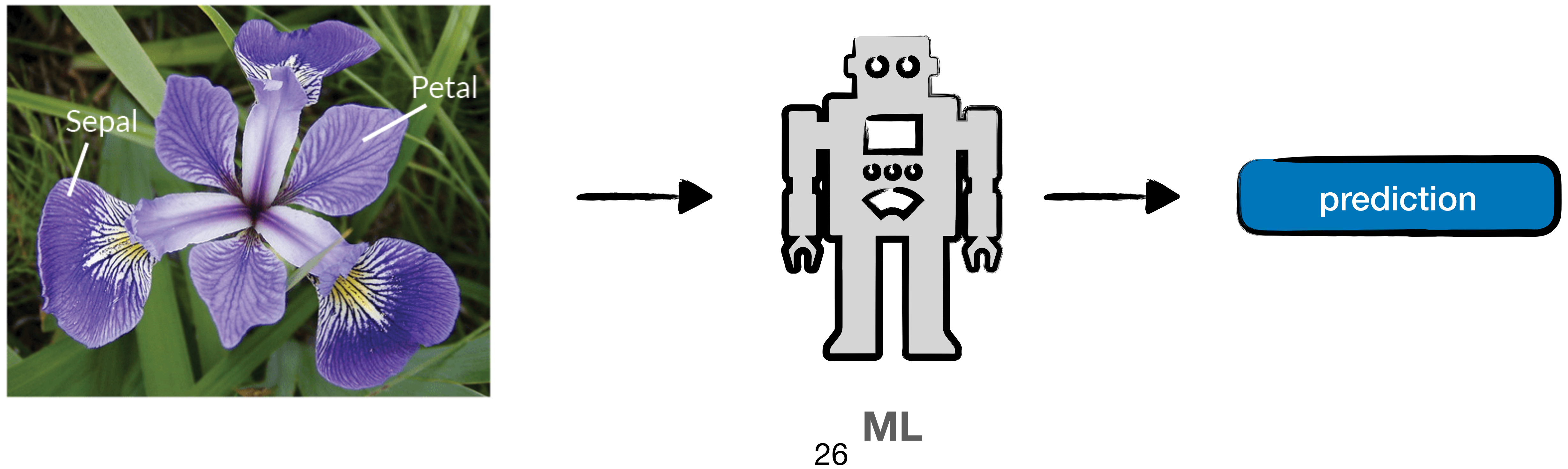
didn’t require (and couldn’t afford) very complex hypothesis spaces



Deep Learning

More ambitious: Can we learn the features as well?





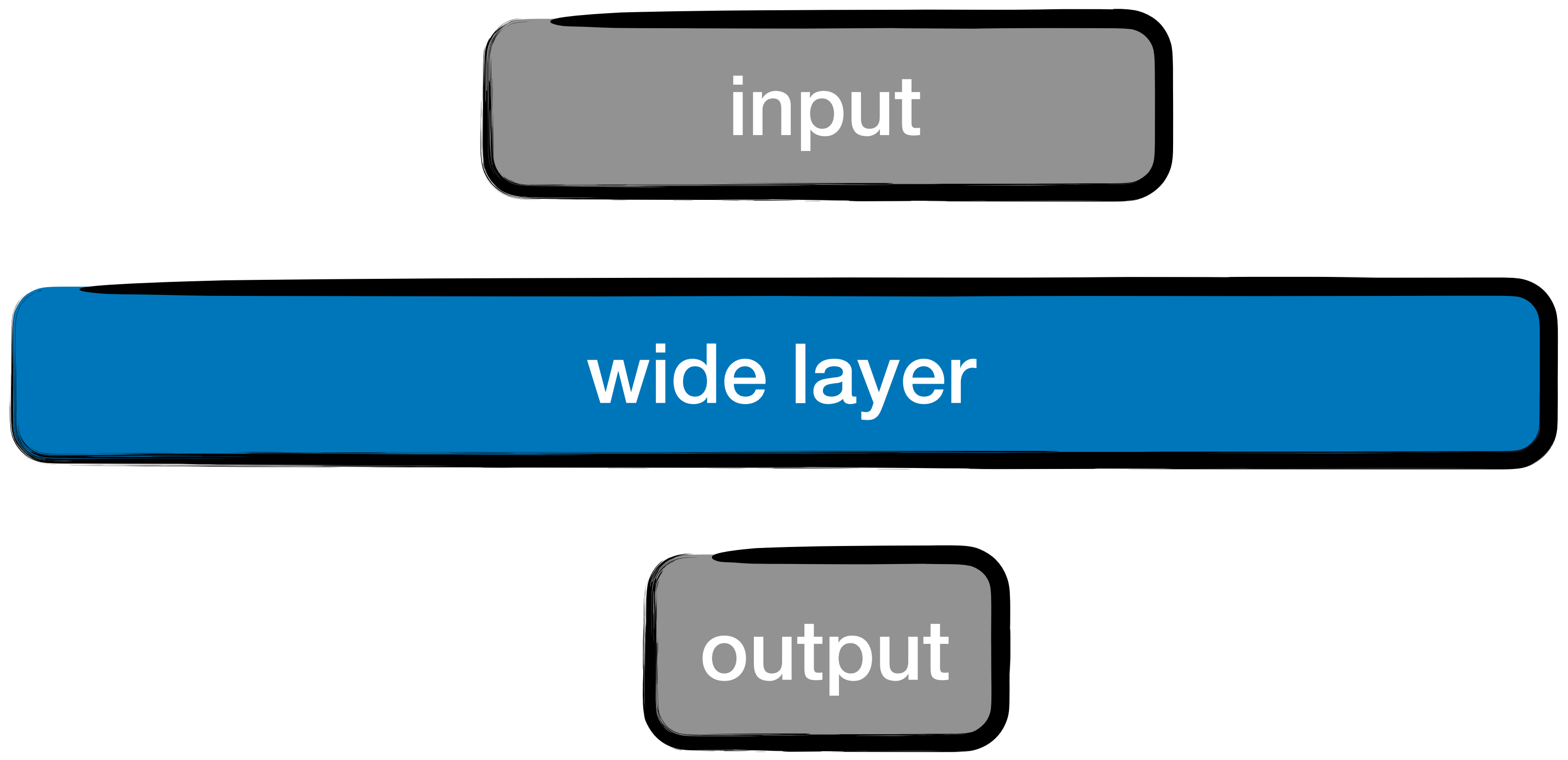
Deep Learning Considerations

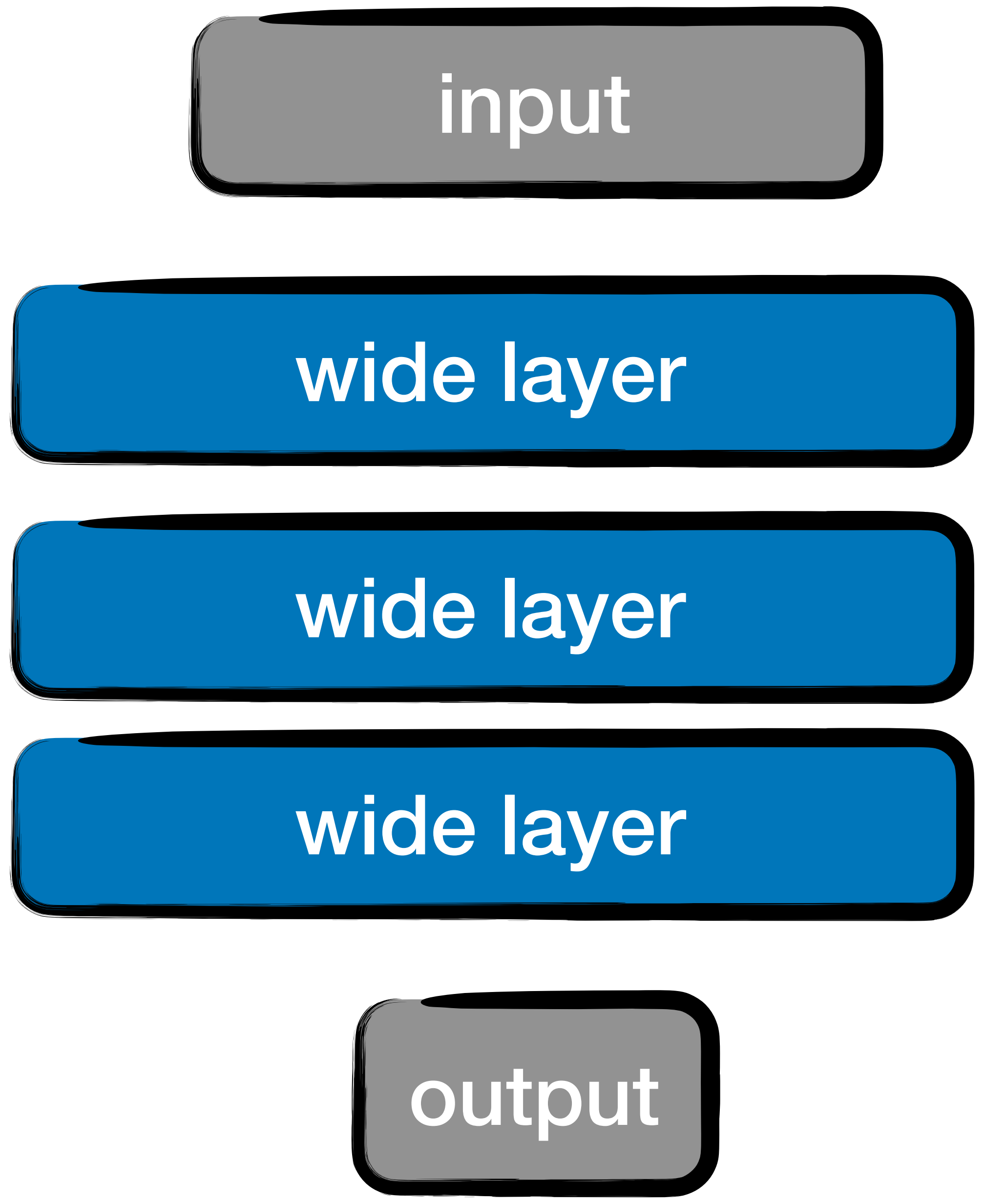
To pull this off you will need

- **much more complex functions** for e.g. pixels → cat|
  →bigger hypothesis sets
- sufficient amount of data to be able to afford them
  → remember bias variance tradeoff
- … or an affective way too constrain the search space
  → inductive bias

Growing Neural Networks

How should we grow our neural networks? Wide of Deep?

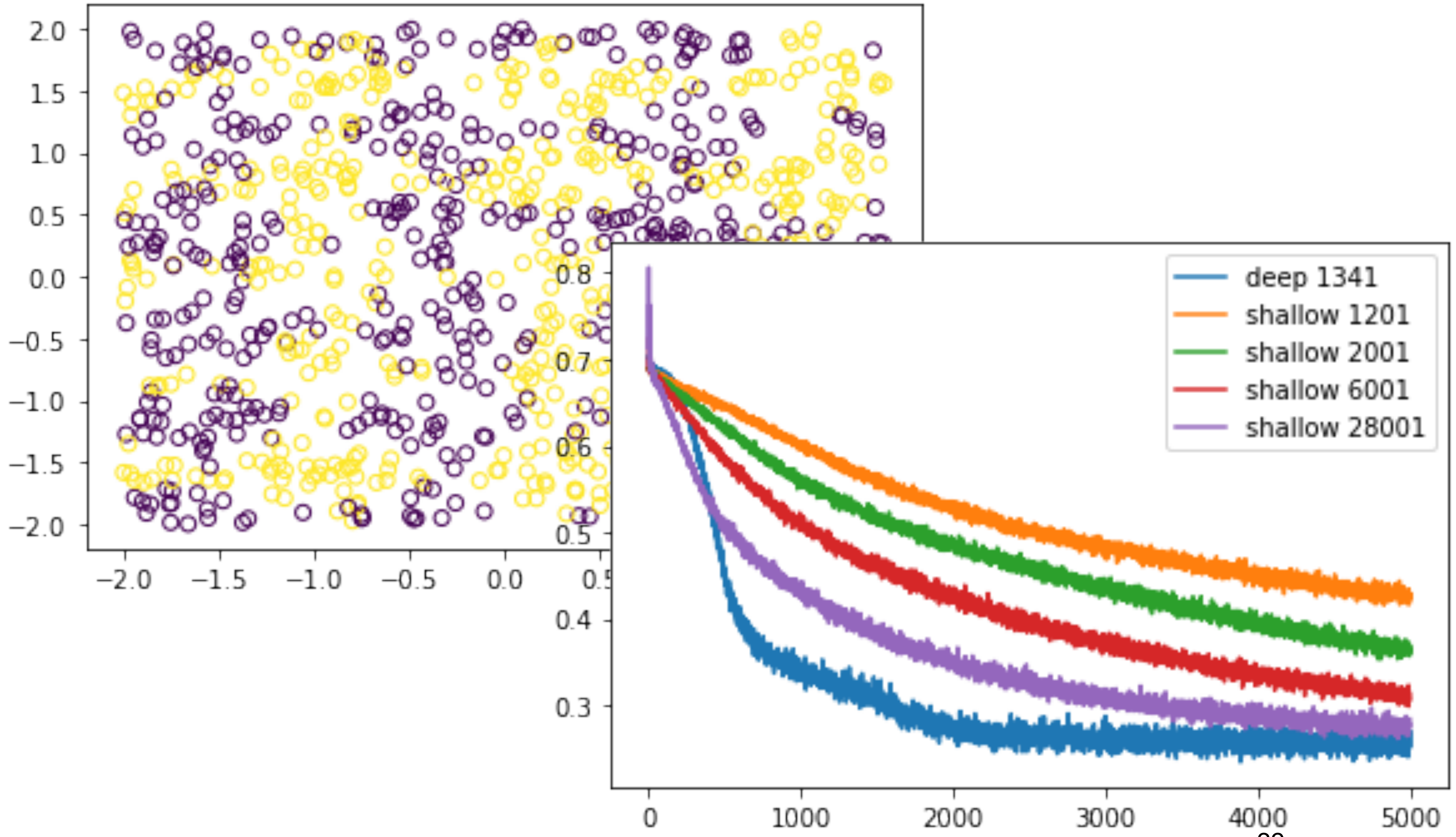


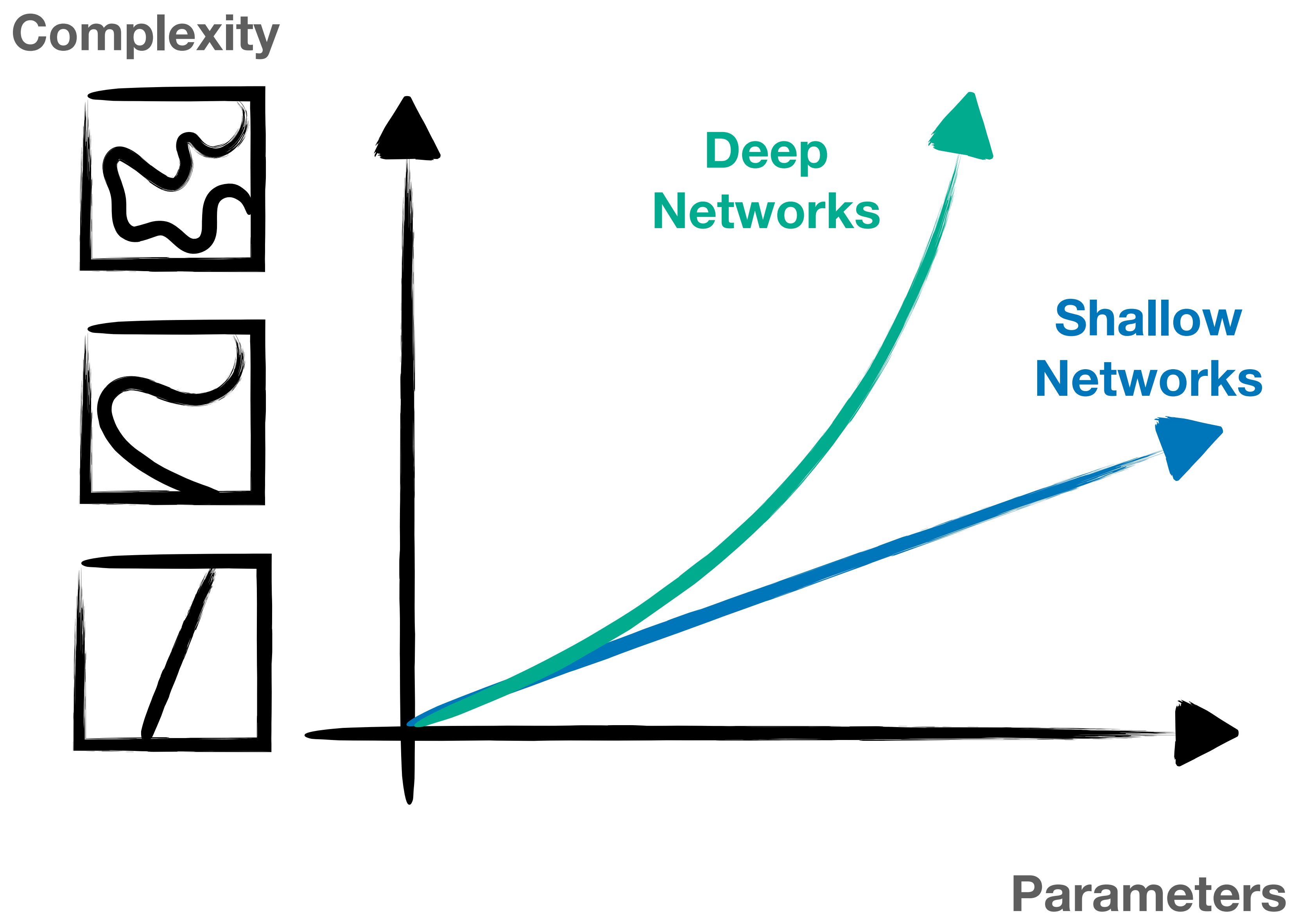


Both add parameters, but what about the actual functions they can approximate?

Benefits of Depth

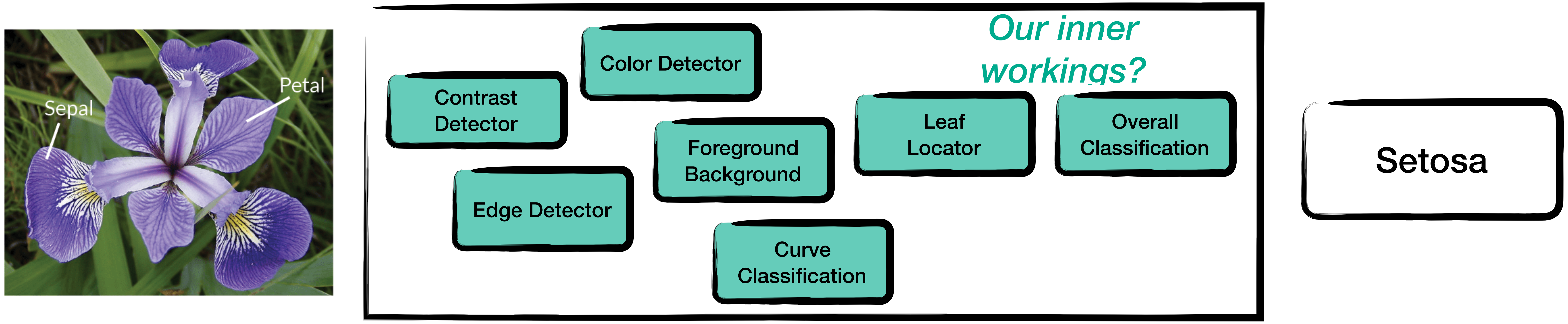
Deep Networks are much more effective at approximating complex functions.

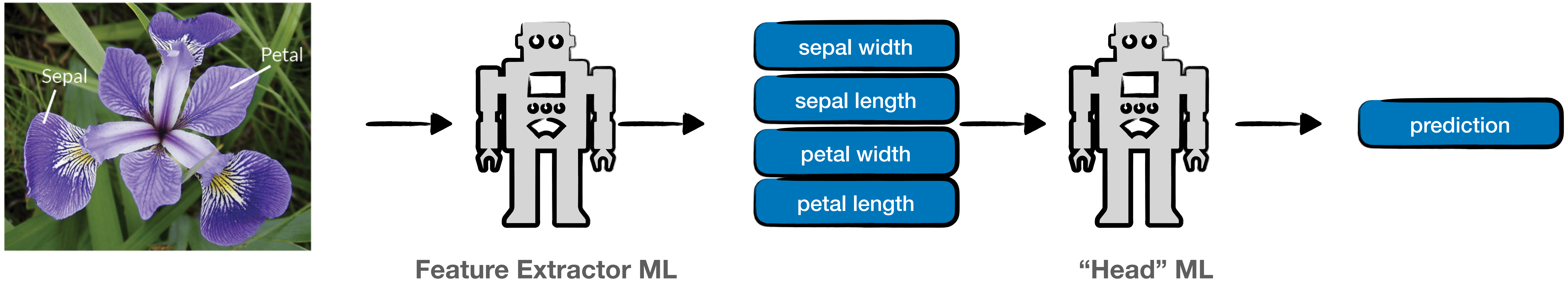




Deep Learning

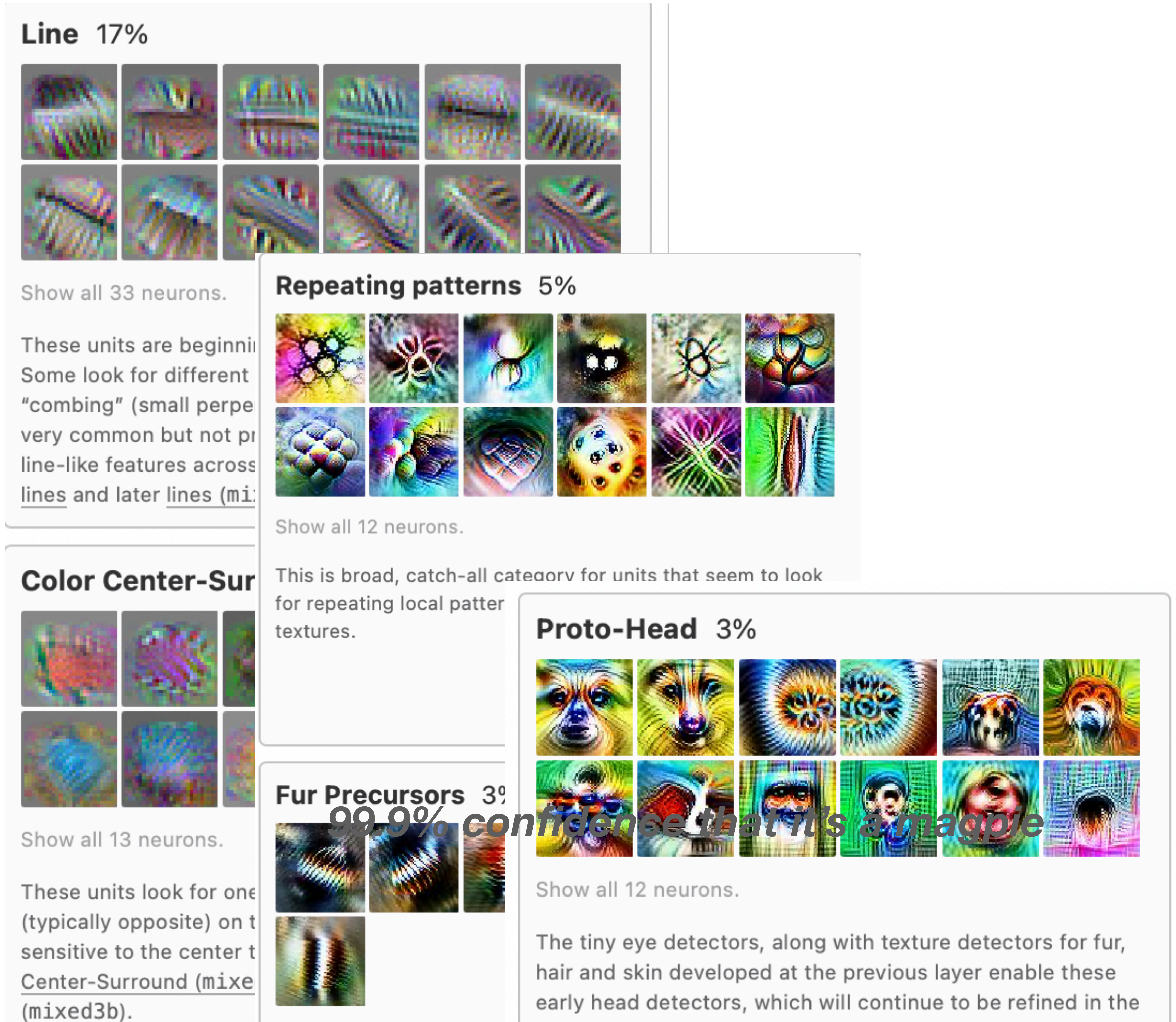
The assumption is that similar to us, effective machine-learned reasoning should go through layers of abstraction



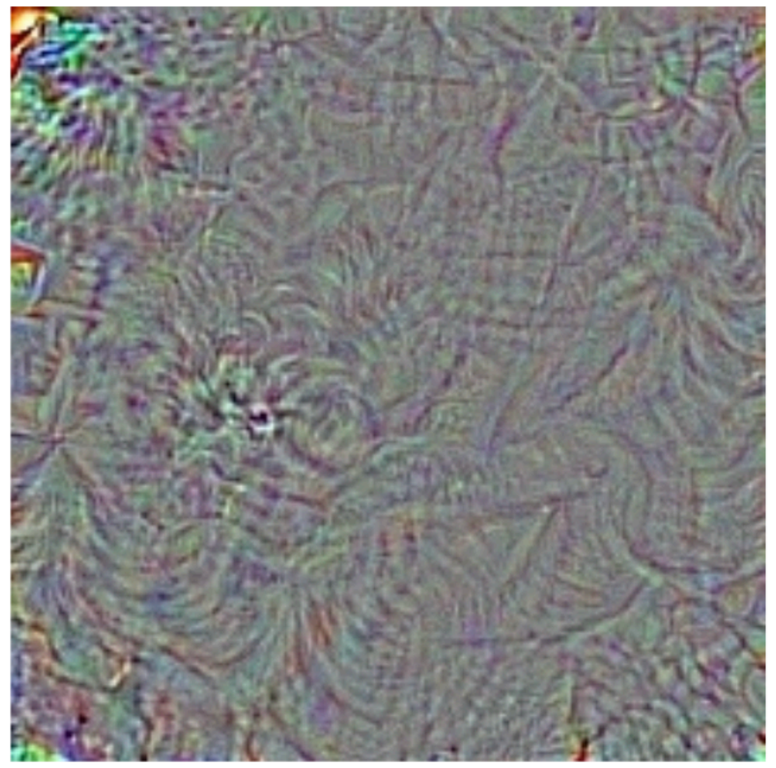


Deep Learning

We do see this, but care is needed to not overinterpret this



99.9% confidence that it's a magpie



ML system has 99% confidence that this is a magpie

… an actual magpie

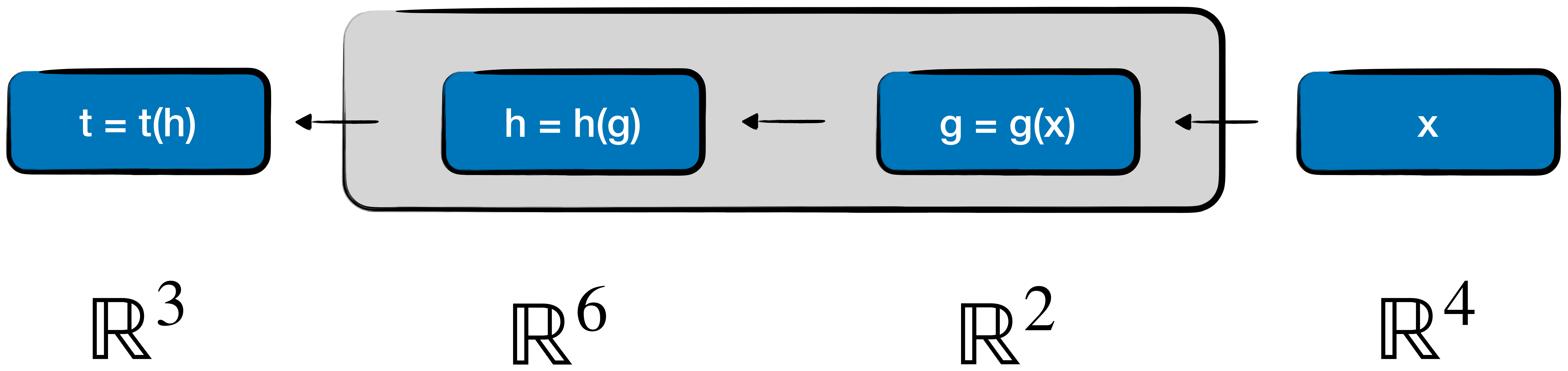
[src]

Gradients of Deep Programs

Gradient Descent needs… Gradients!

As neural networks become bigger & deeper, need to find a way to compute them efficiently

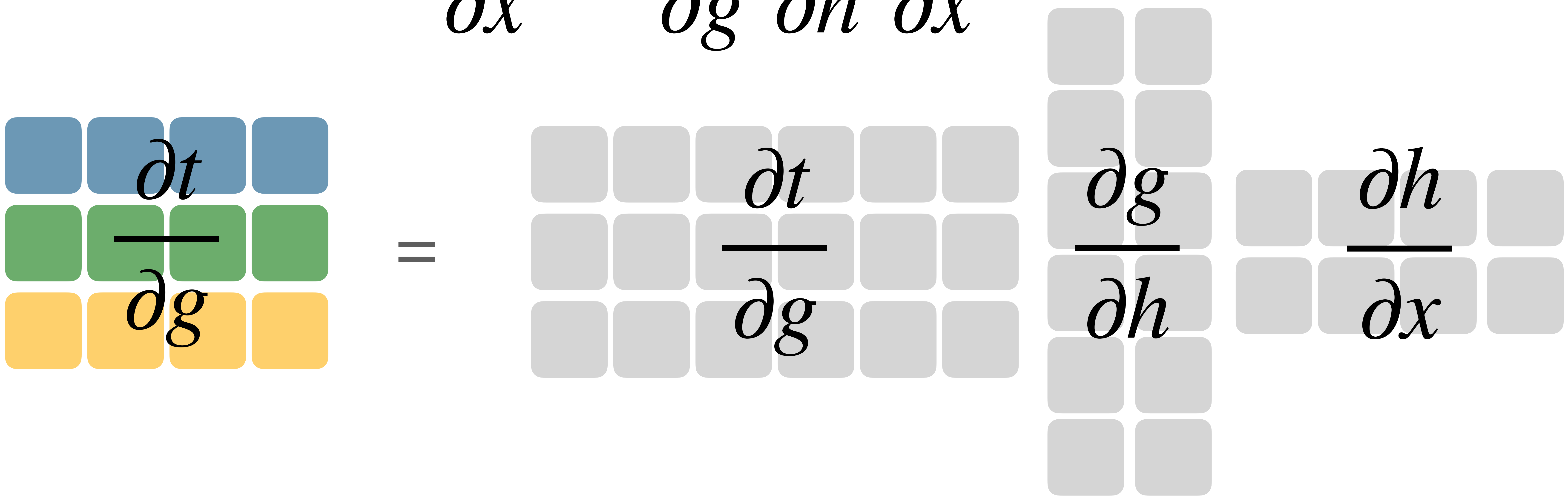
$$f(x) = (t \circ h \circ g)(x) = t(h(g(x))) : \mathbb{R}^4 \to \mathbb{R}^3$$



Gradient Descent needs… Gradients!

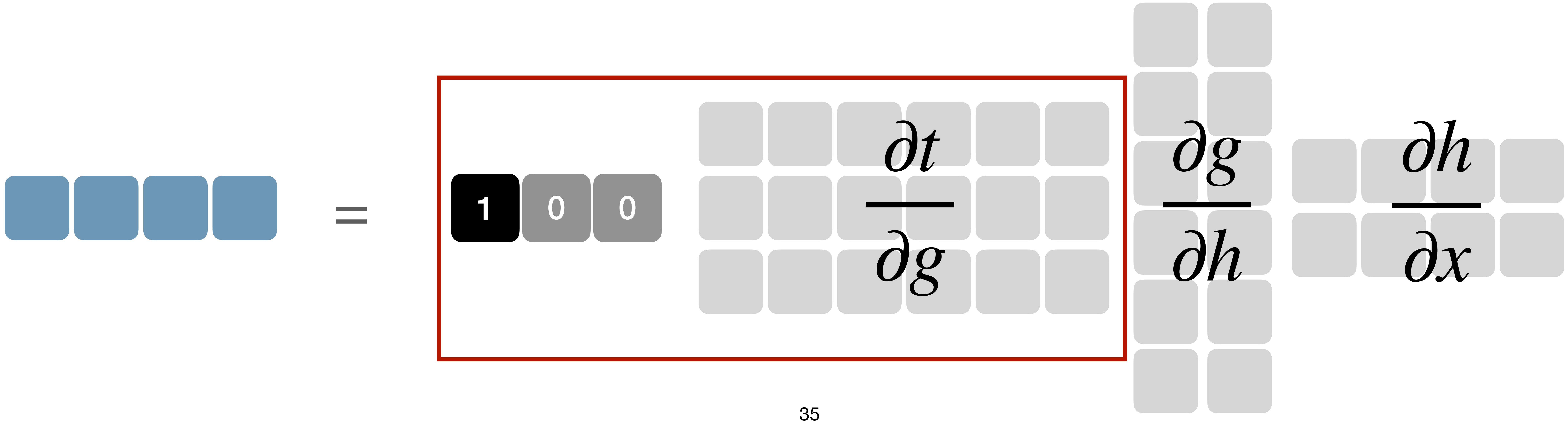
We want to compute the Jacobian of the deep composition of functions. **But a naive approach scales badly**

$$\frac{\partial f}{\partial x} = \frac{\partial t}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial x}$$



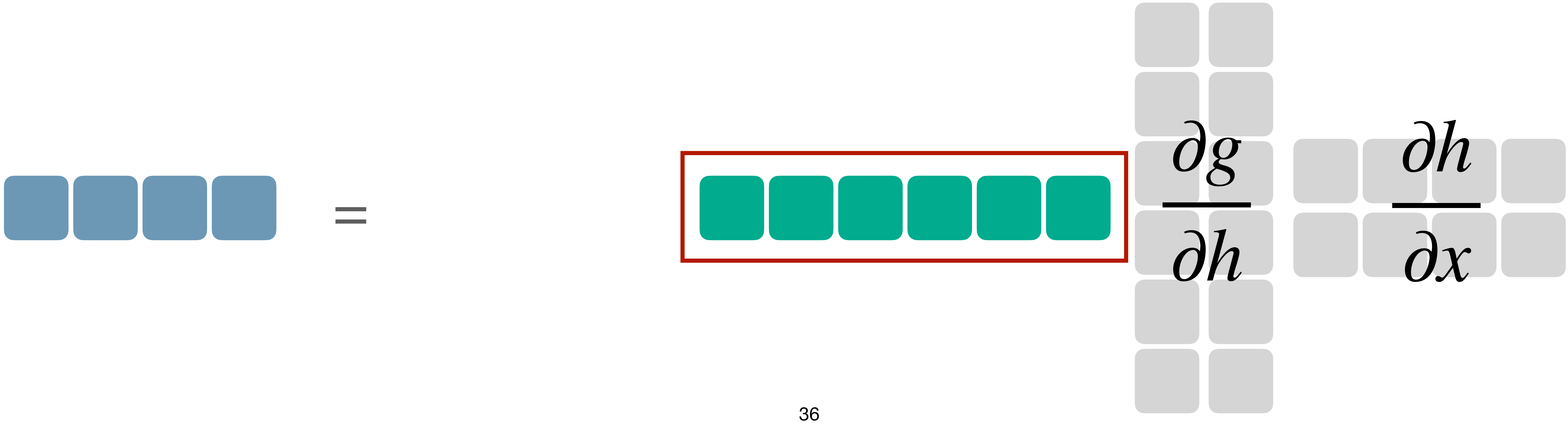
Matrix-Free Computation

Instead of Matrix-Matrix products, we can compute more **cheap vector-Matrix products** and compute a row at a time



Matrix-Free Computation

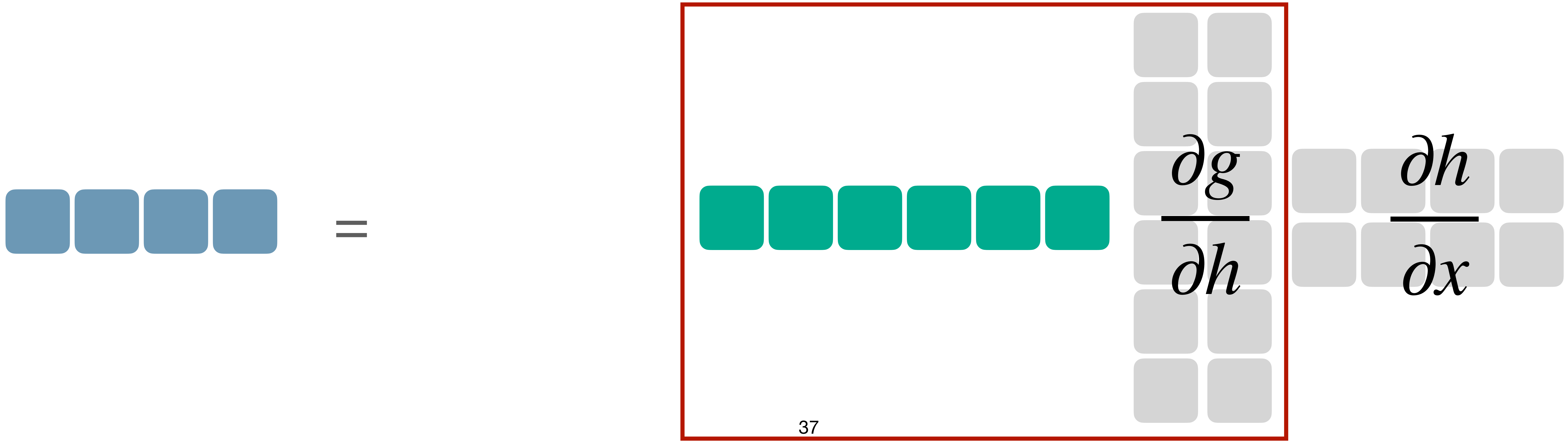
Instead of Matrix-Matrix products, we can compute more **cheap vector-Matrix products** and compute a row at a time



$$\frac{\partial g}{\partial h} \qquad \frac{\partial h}{\partial x}$$

Matrix-Free Computation

Instead of Matrix-Matrix products, we can compute more **cheap vector-Matrix products** and compute a row at a time

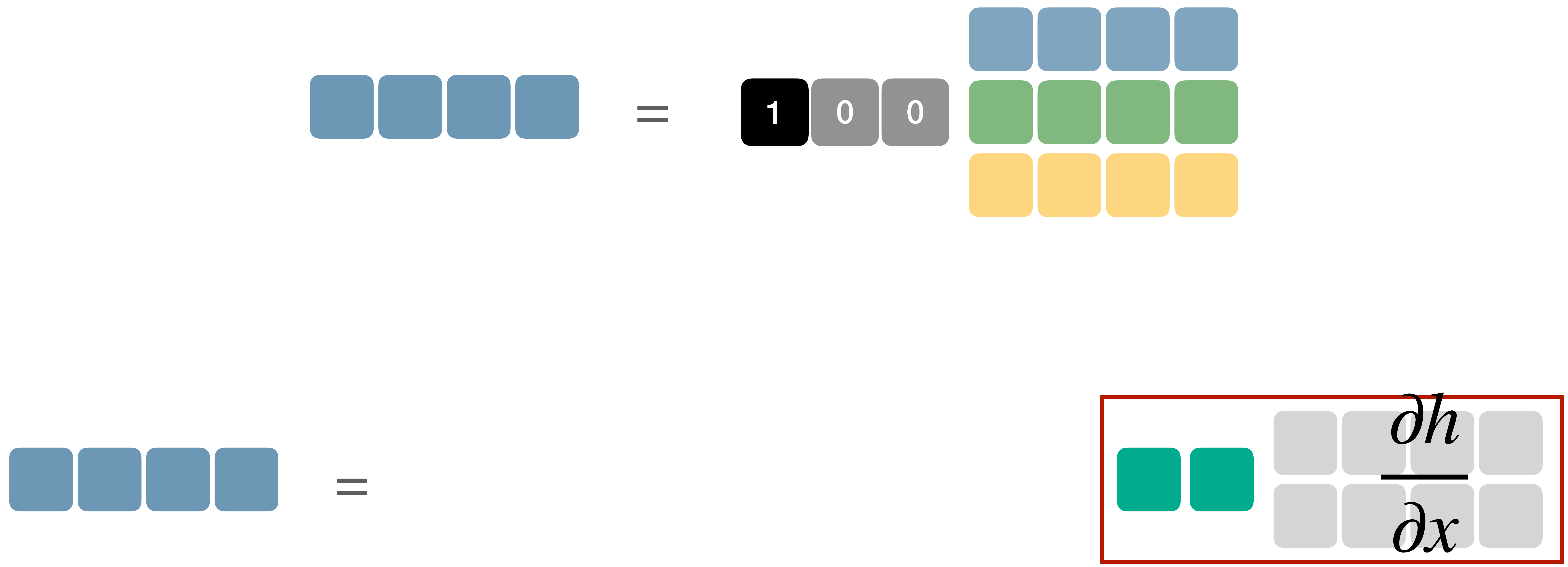


Matrix-Free Computation

Instead of Matrix-Matrix products, we can compute more **cheap vector-Matrix products** and compute a row at a time

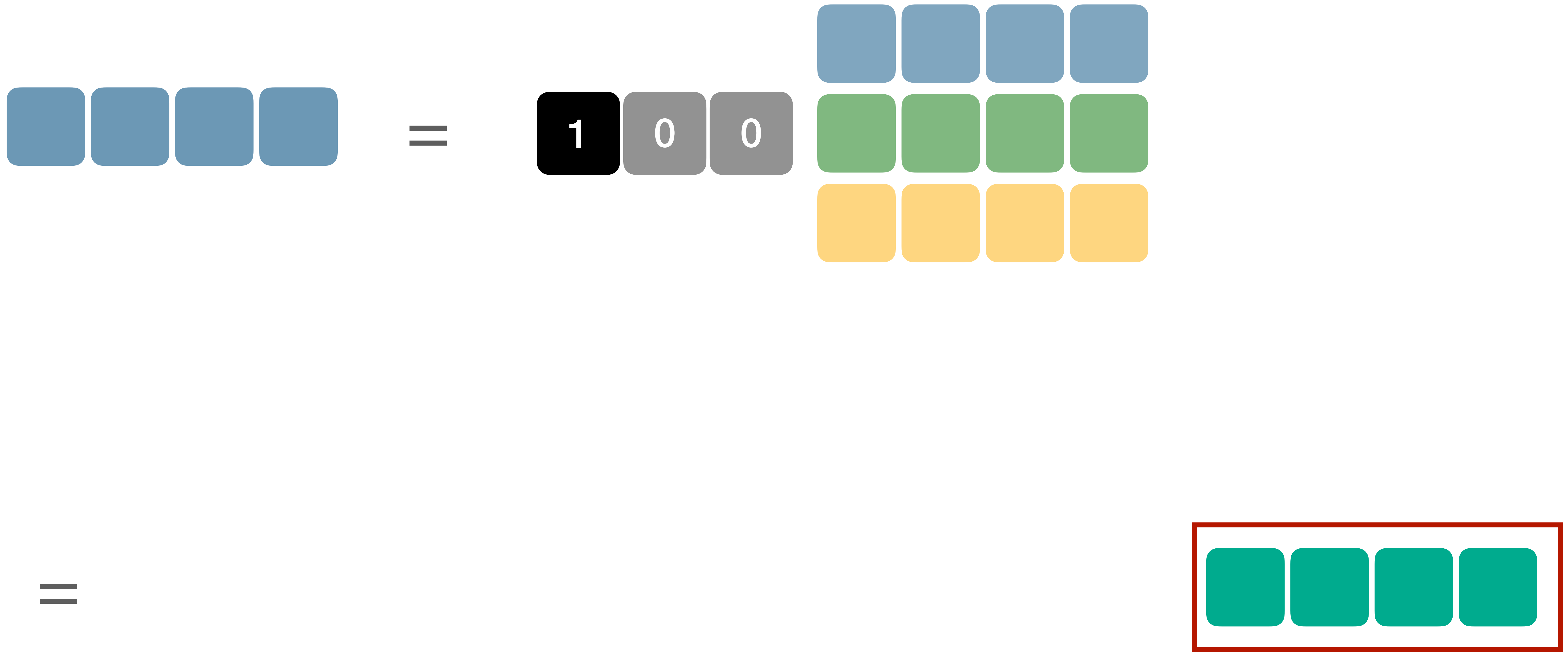
Matrix-Free Computation

Instead of Matrix-Matrix products, we can compute more **cheap vector-Matrix products** and compute a row at a time



Matrix-Free Computation

Instead of Matrix-Matrix products, we can compute more **cheap vector-Matrix products** and compute a row at a time



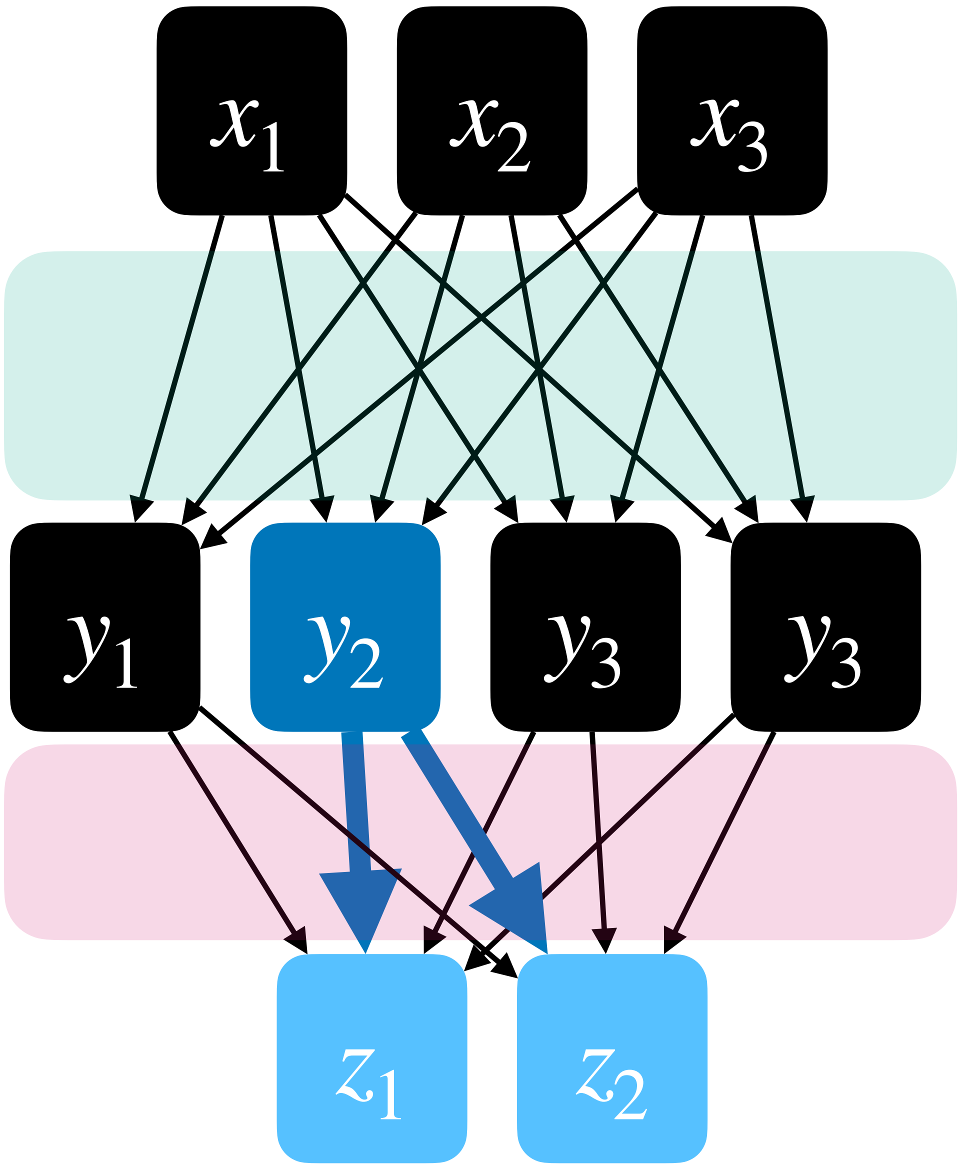
Automatic Differentiation

The approach can be generalized to arbitrary computational graphs: **The Backpropagation Algorithm**

g^z $\quad J_{y\to z} = \frac{\partial z_i}{\partial y_j}$ $\quad g^y = \frac{\partial L}{\partial y}$

$$J_{x\to y} = \frac{\partial y_i}{\partial x_j}$$

$$J_{y\to z} = \frac{\partial z_i}{\partial y_j}$$



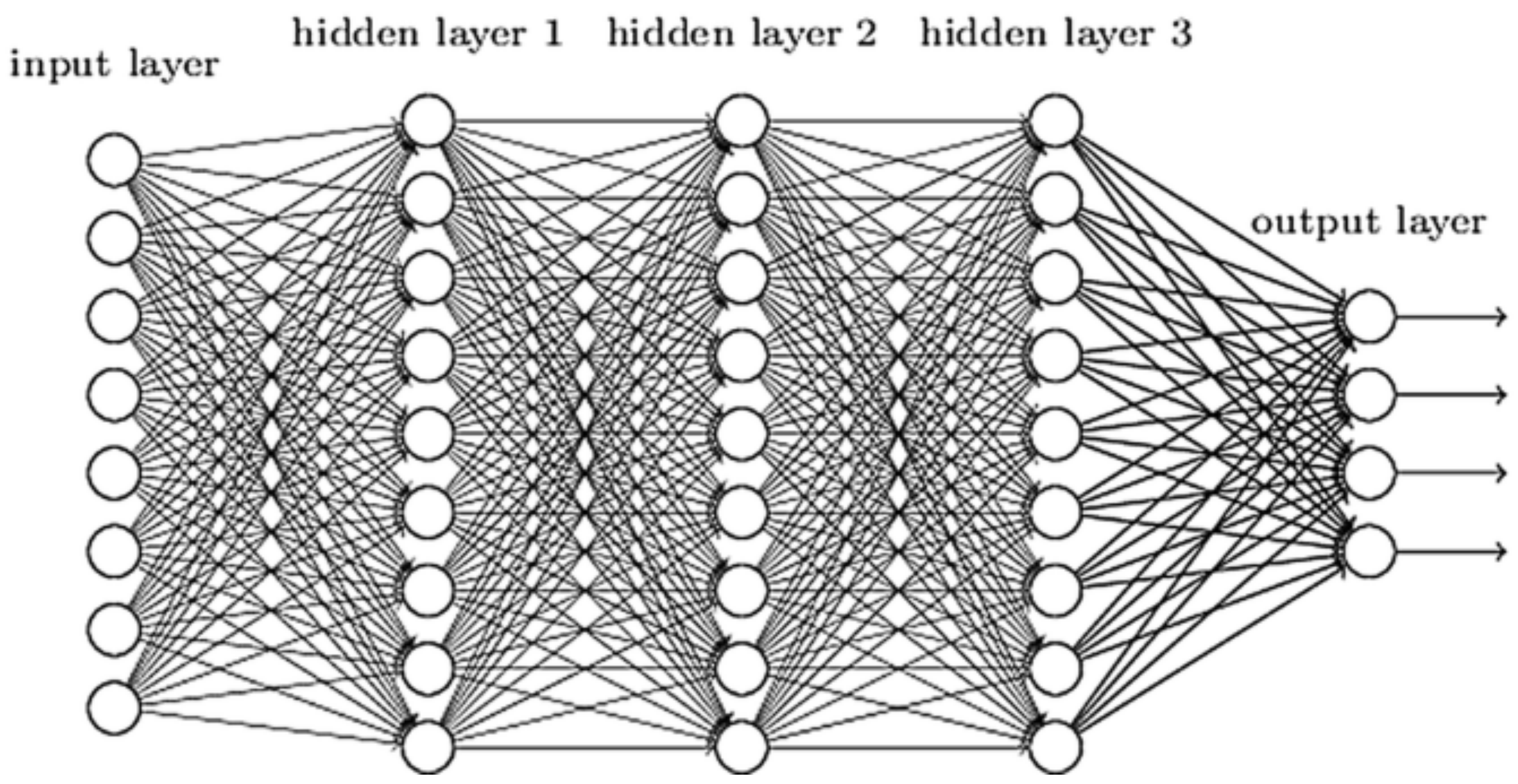
$$g^y = (g^z J)_i = \sum_k g^z_k J_{ki}$$

$$g^y_i = (g^z J)_i = \sum_{c \in \text{children}(y_i)} g^z_c \frac{\partial z_c}{\partial z_i}$$

Putting it all Together

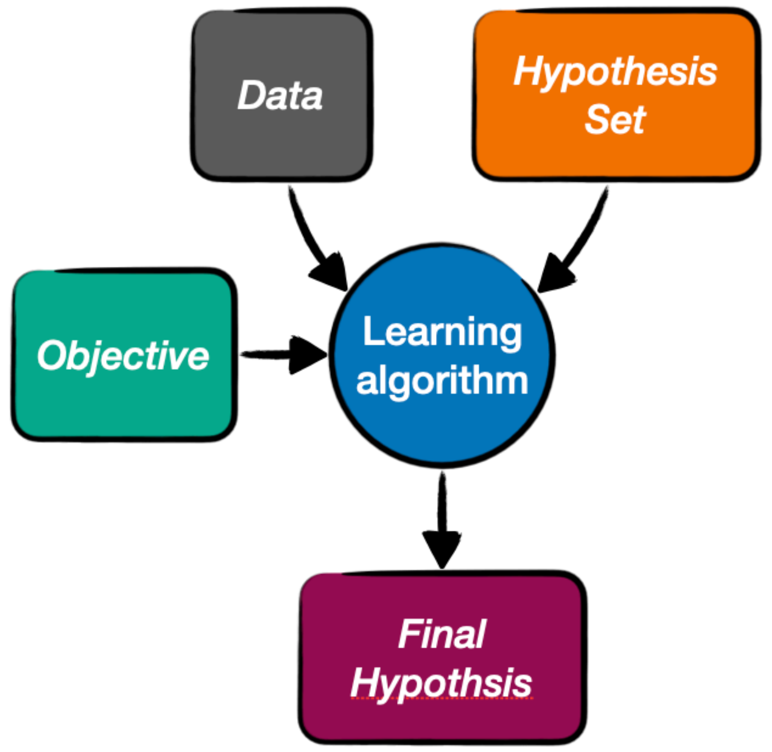
ML Frameworks

ML Frameworks like PyTorch, Tensorflow, JAX put a lot of the pieces together to provide a performance setup




```python
def create_model(input_dim, output_dim):
  return torch.nn.Sequential(
      torch.nn.Linear(input_dim, 9),
      torch.nn.ReLU(),
      torch.nn.Linear(9, 9),
      torch.nn.ReLU(),
      torch.nn.Linear(9, 9),
      torch.nn.ReLU(),
      torch.nn.Linear(9, output_dim),
      torch.nn.Sigmoid()
  )
```

A full training Loop



Data $s \sim p(s)$

```python
def learn(samples):
    features, labels = samples
    model = MyModel()
    loss_func = torch.nn.BCELoss()
    opt = torch.optim.Adam(
        model.parameters(), lr = 1e-3
    )

    for i in range(steps):
        predictions = model(samples)
        loss = loss_func(predictions, labels)
        loss.backward()
        opt.step()
        opt.zero_grad()
    return model
```

Hypothesis Set $f_\phi(x)$

Objective $L(s) = L(f_\phi(x), z)$

Learning Algorithm $\mathrm{argmin}_\phi L_{\mathrm{MC}}$

Gradient Computation $\nabla_\phi L_{\mathrm{MC}}(\phi)$

Trained Model $f_{\hat{\phi}}(x)$

Inductive Bias & Architectures

Beyond Depth

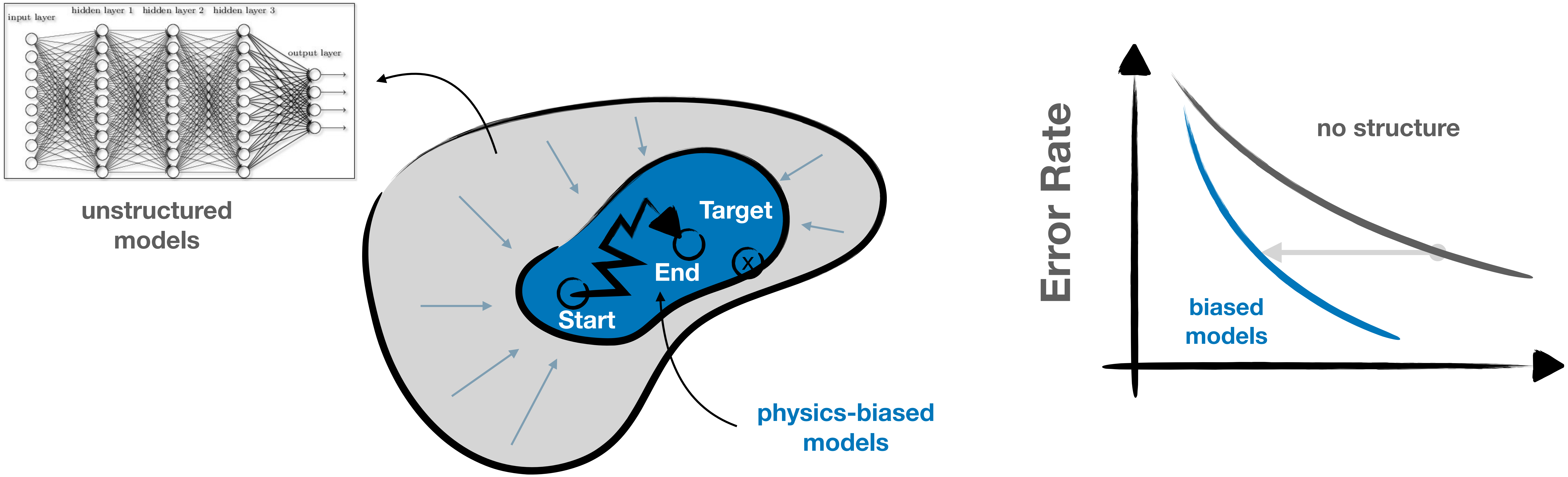
Can we push this further, should we move away from universal function approximators?

- bias variance tradeoff: reduce $\mathscr{H}$ as much as you can

General Idea: $\mathscr{H}$ should match **data modality & task**

Inductive Bias

If we can throw out irrelevant functions, which we know can't be the solution, we **bias** our inductive process towards good solution (here: bias is good)



The Architecture Zoo

Convolutional Neural Nets

Convolutional Neural Networks are (approximately) **translationally invariant**

Is there a cat in the picture?

How about now?

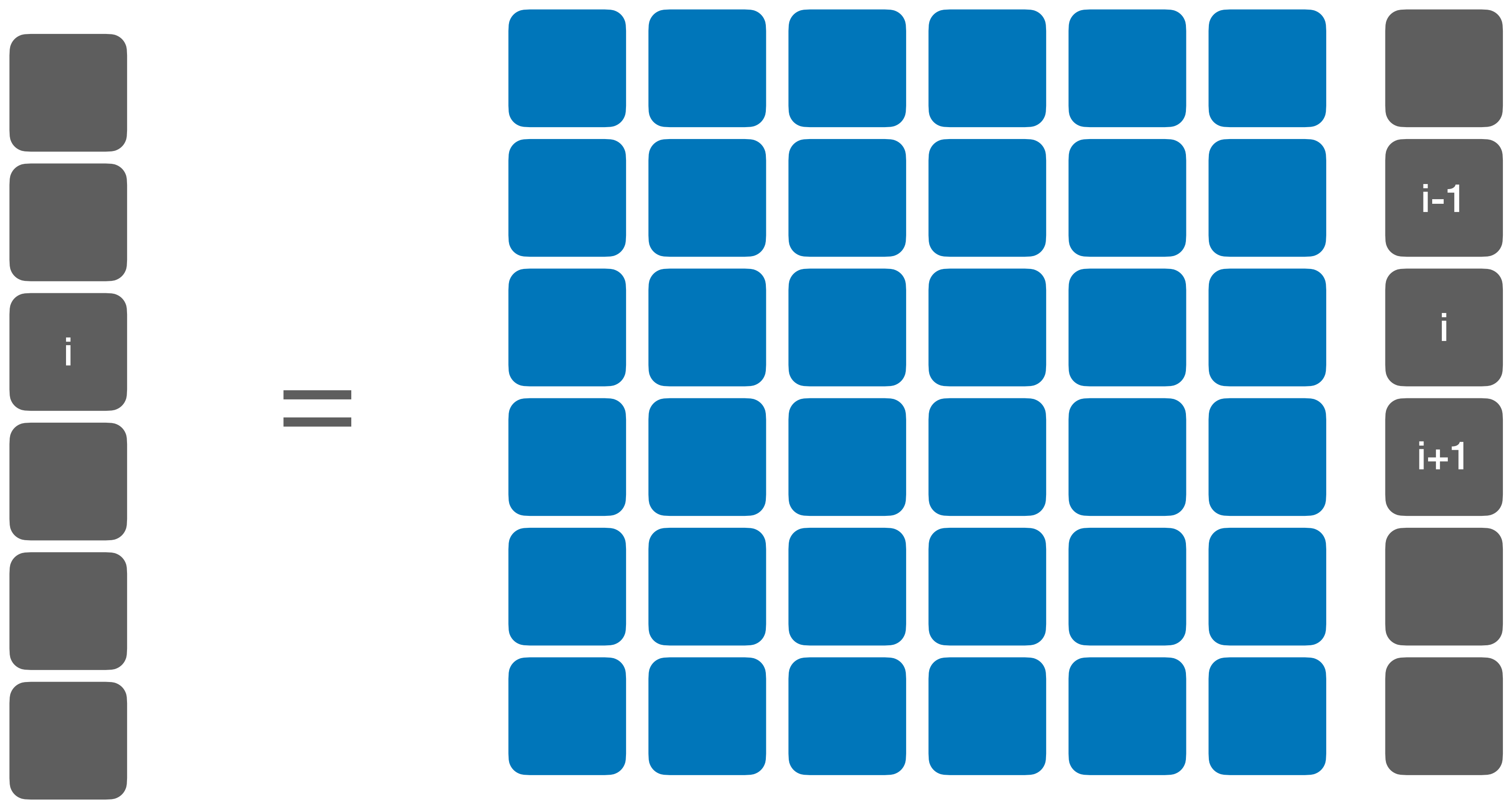
One of the early successes of deep learning in the 80s

Convolutions

Two key ideas lead to convolutions as a building block

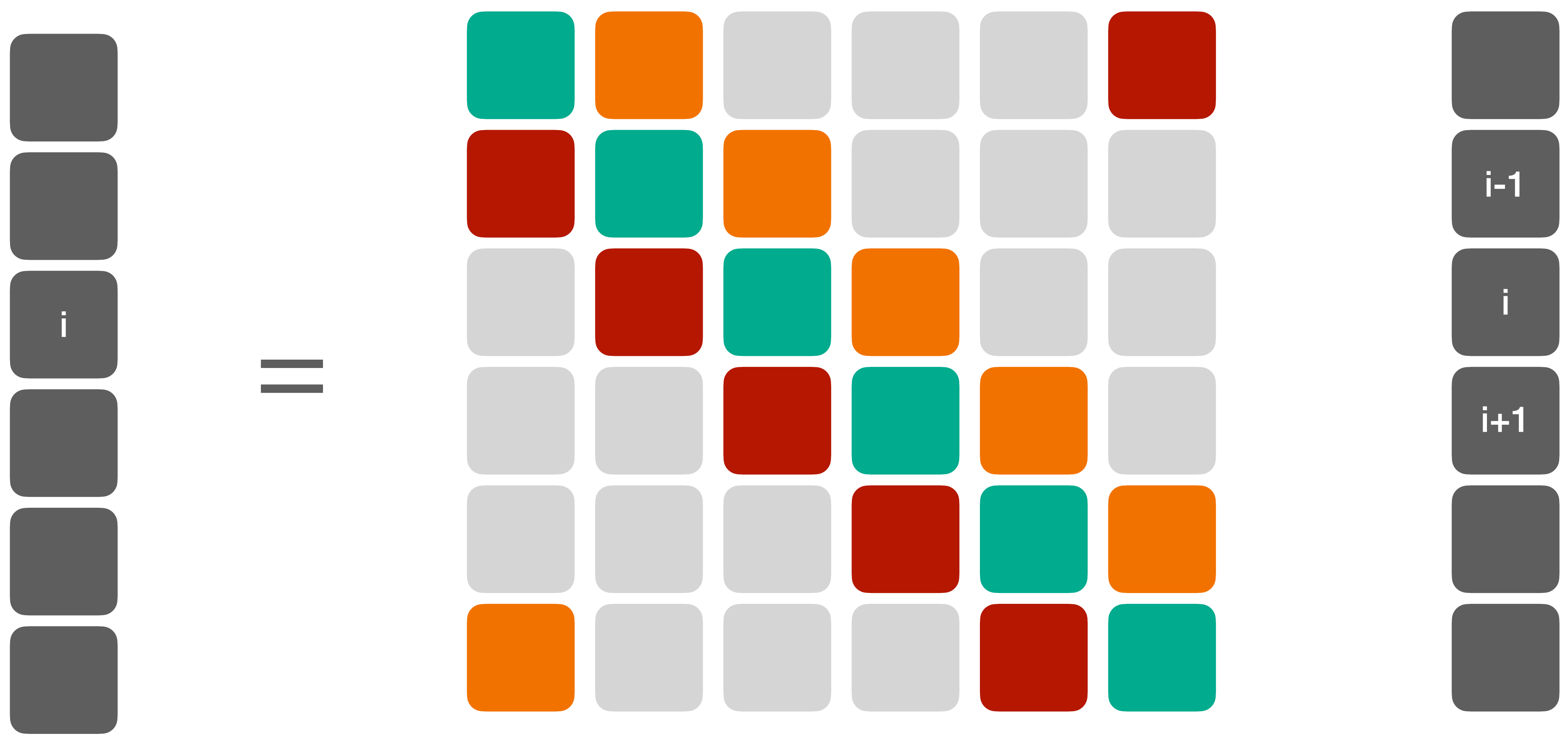
- local connectivity and weight sharing

Standard Linear Layer



$$y_i = \sum_{j=0\ldots N} w_{ij} x_j$$

Convolution



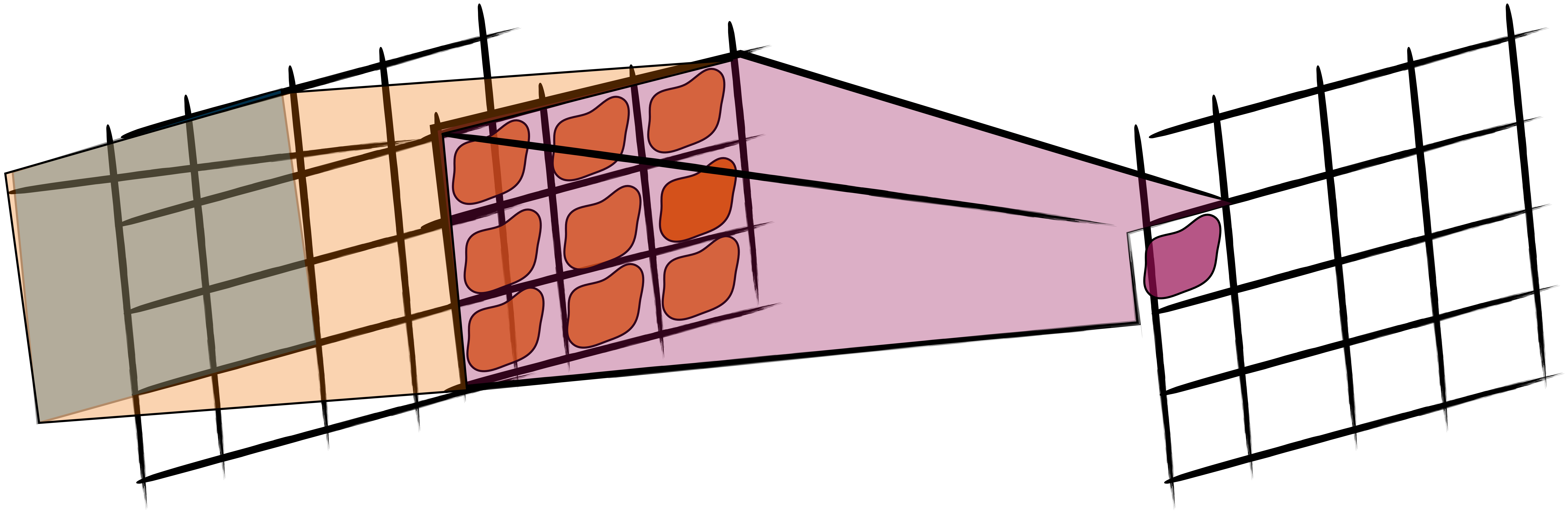
$$y_i = \sum_{o=-1,0,1} w_o x_{i+o}$$

Convolutions

Equivalent to a filter that slides across the input

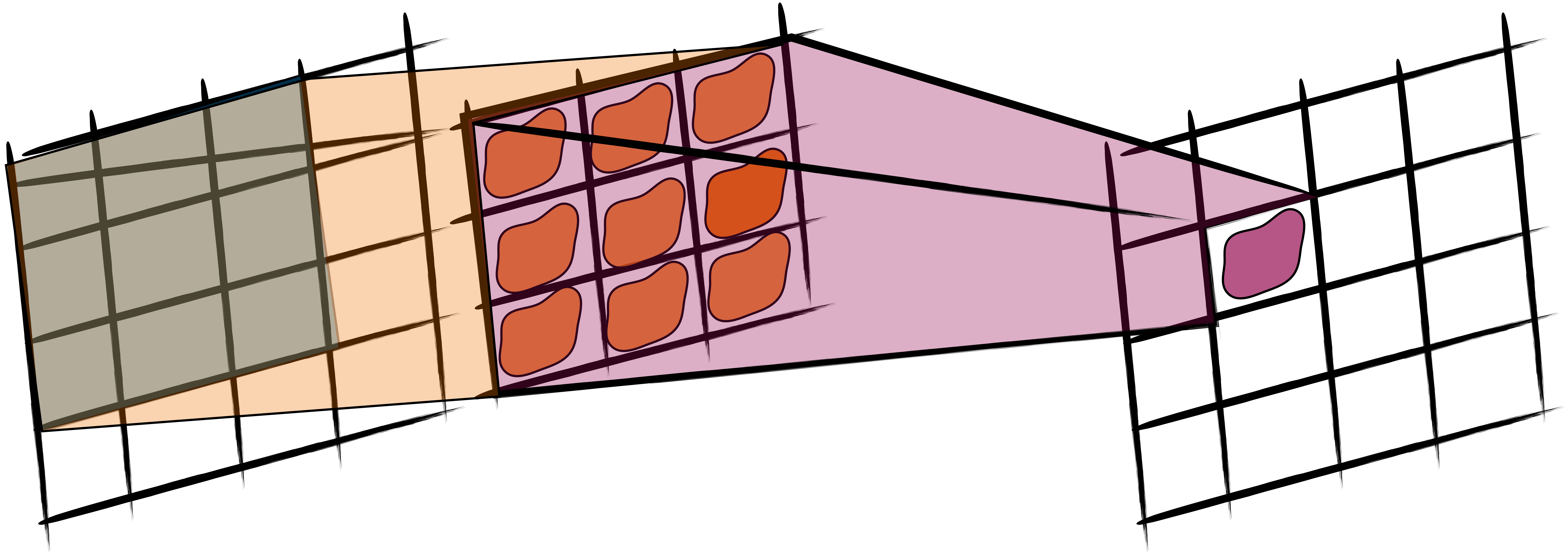
2D Convolutions

We can extend this idea to higher dimensions:



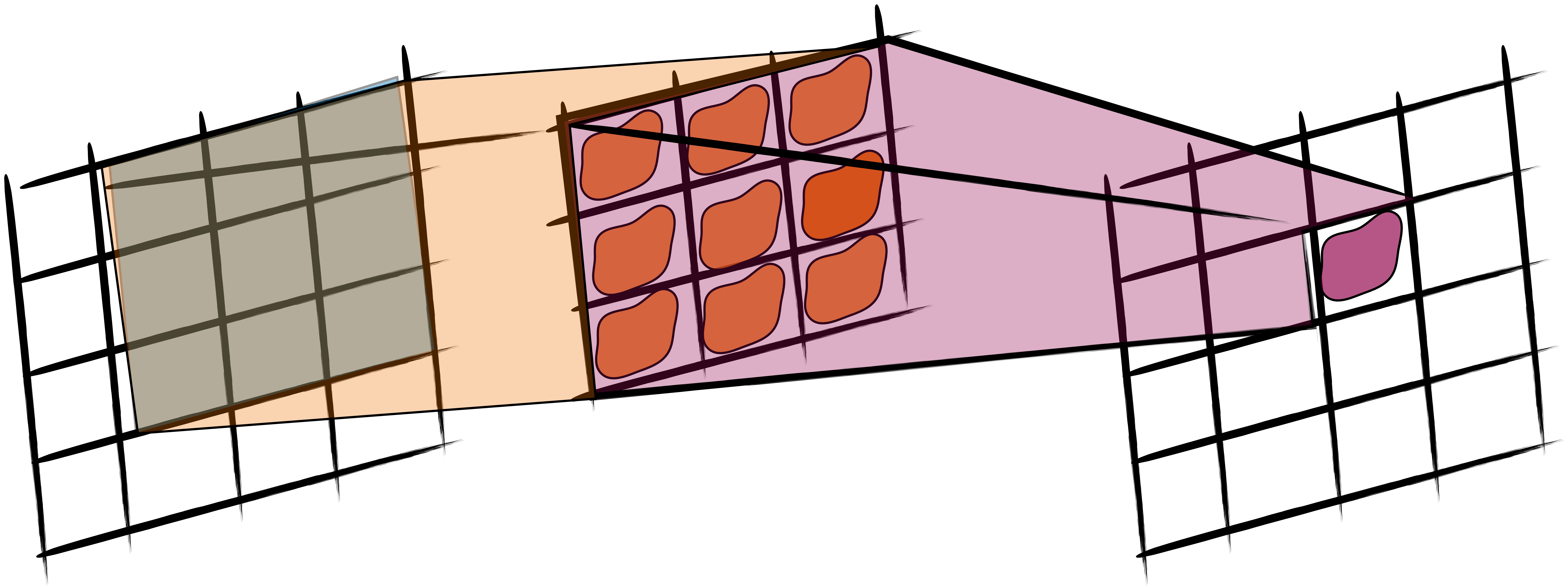
2D Convolutions

We can extend this idea to higher dimensions:



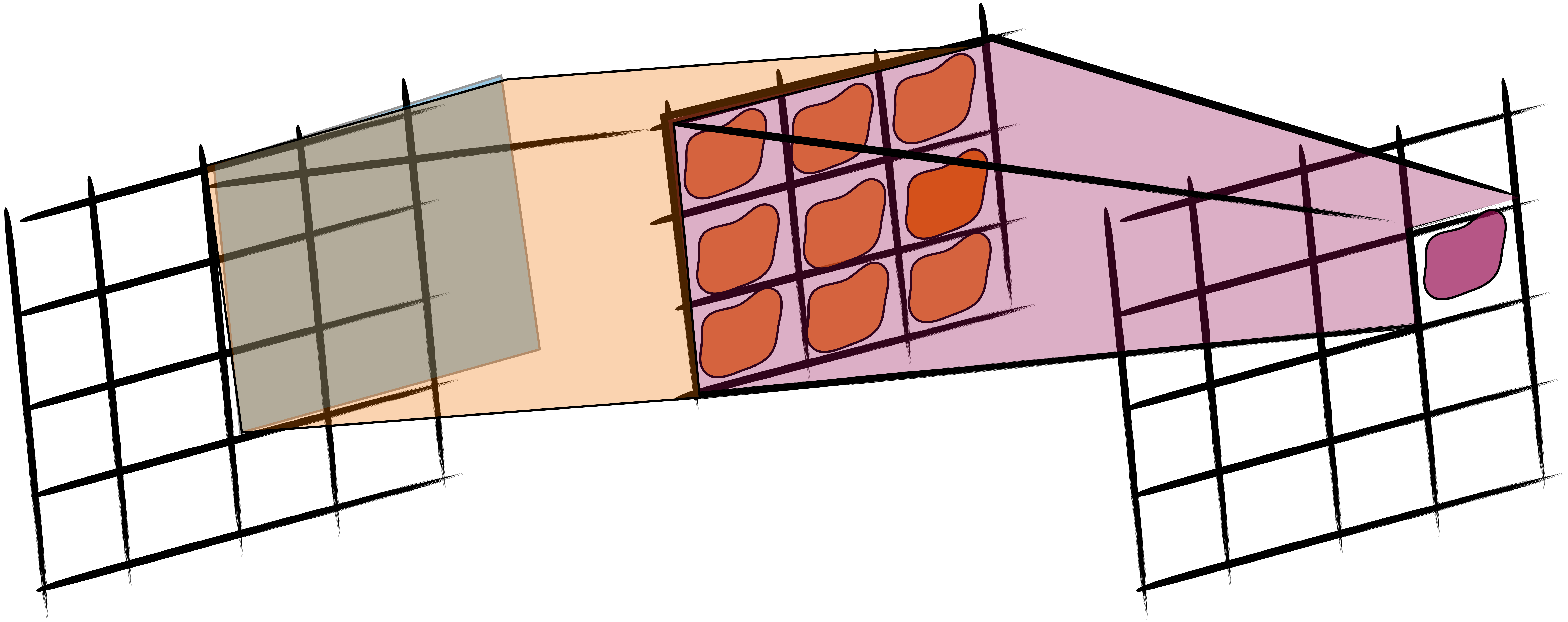
2D Convolutions

We can extend this idea to higher dimensions:



2D Convolutions

We can extend this idea to higher dimensions:

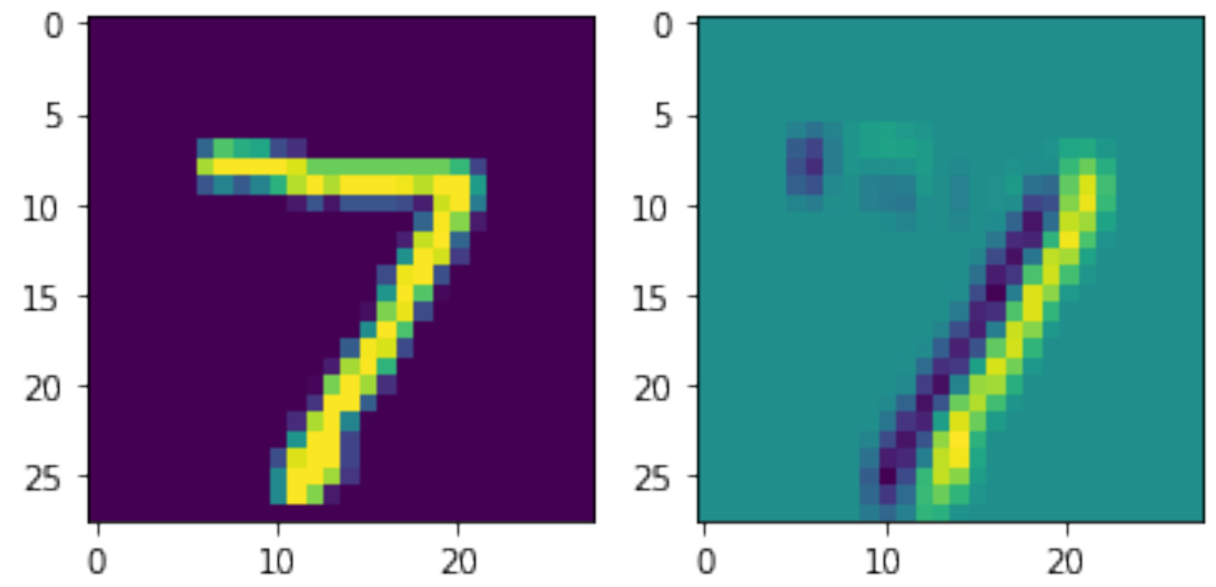


Local Pattern Detectors

The filters are like mini-neural nets extracting features for a local patch: e.g. edges, curves, texture, …

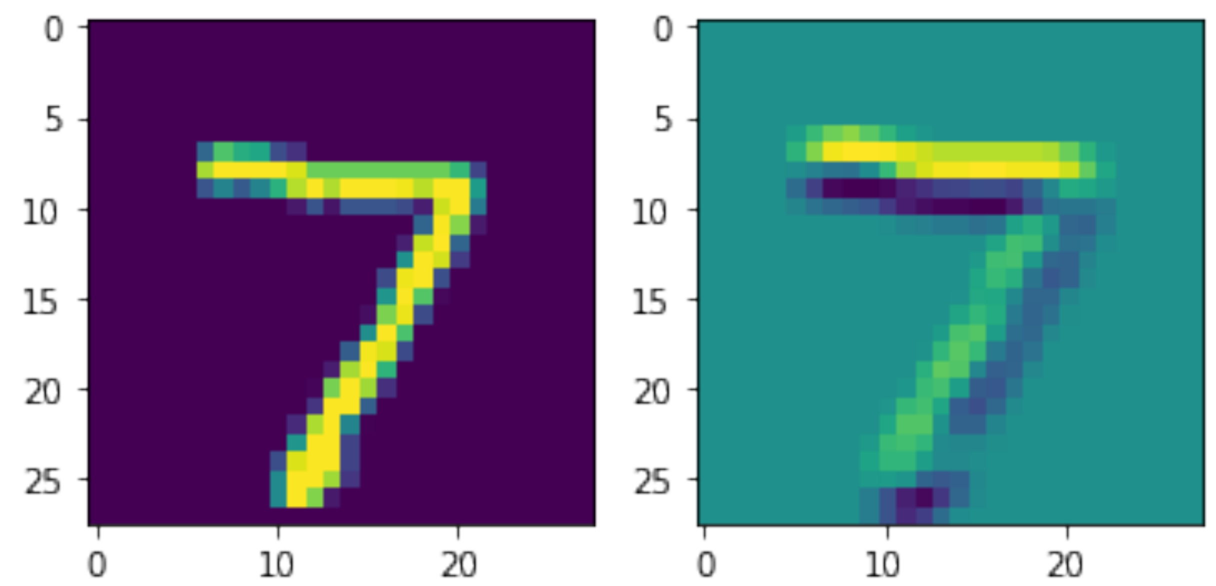
Vertical Edges

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



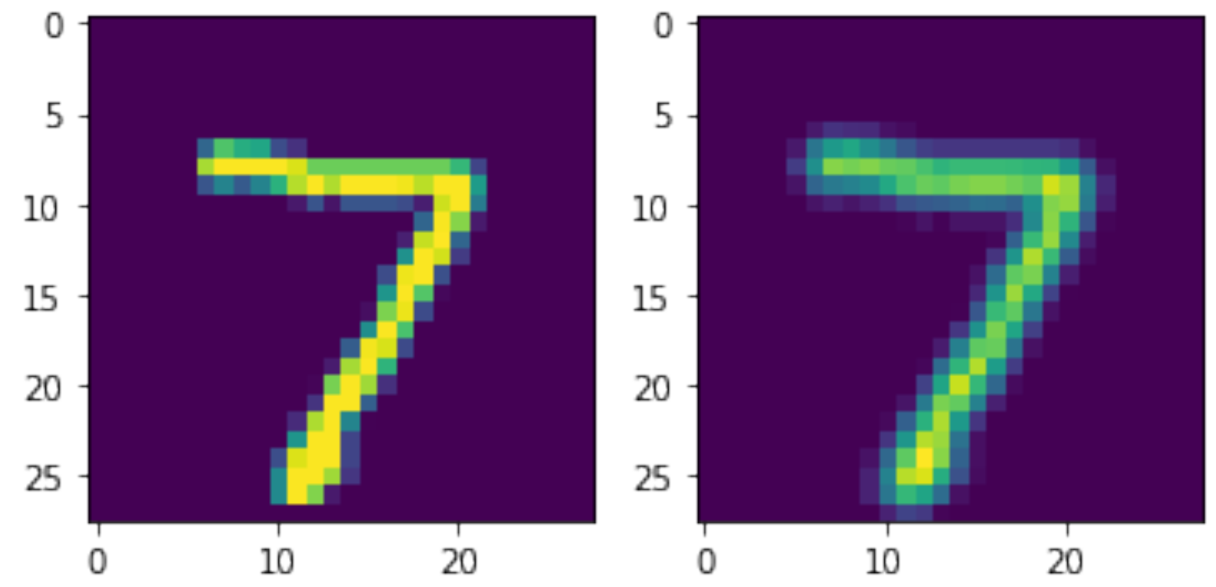
Horizontal Edges

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



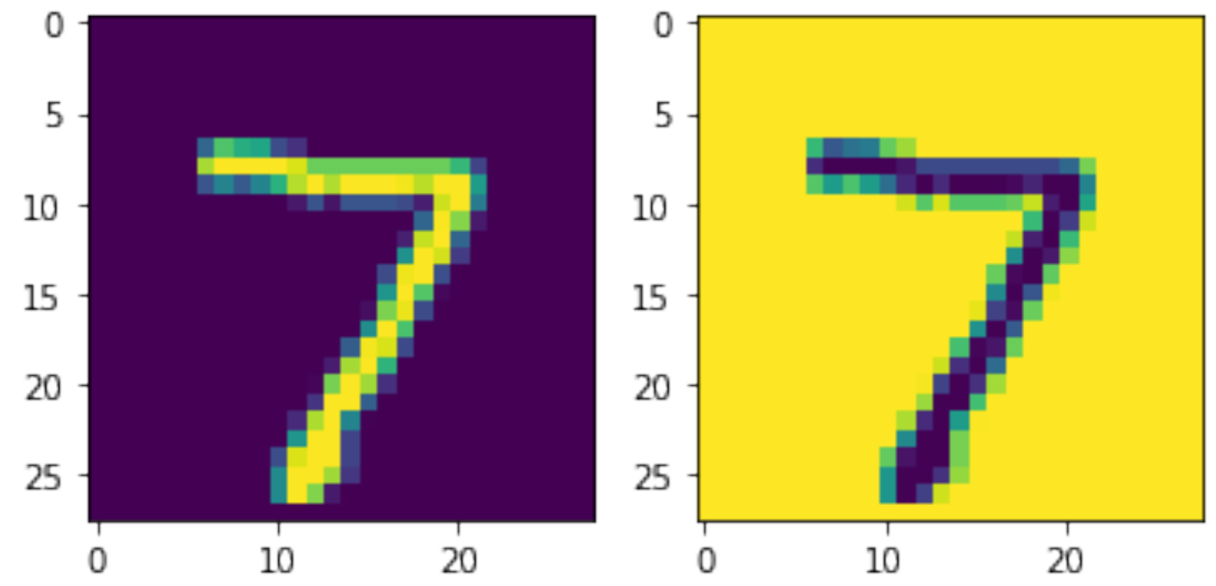
Blurring

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



Inversion

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



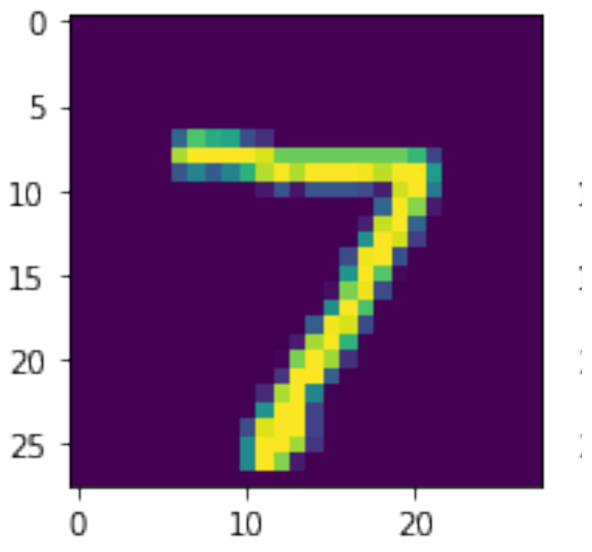
Convolutional Layers

To build up networks, we can extract many features with multiple kernels:

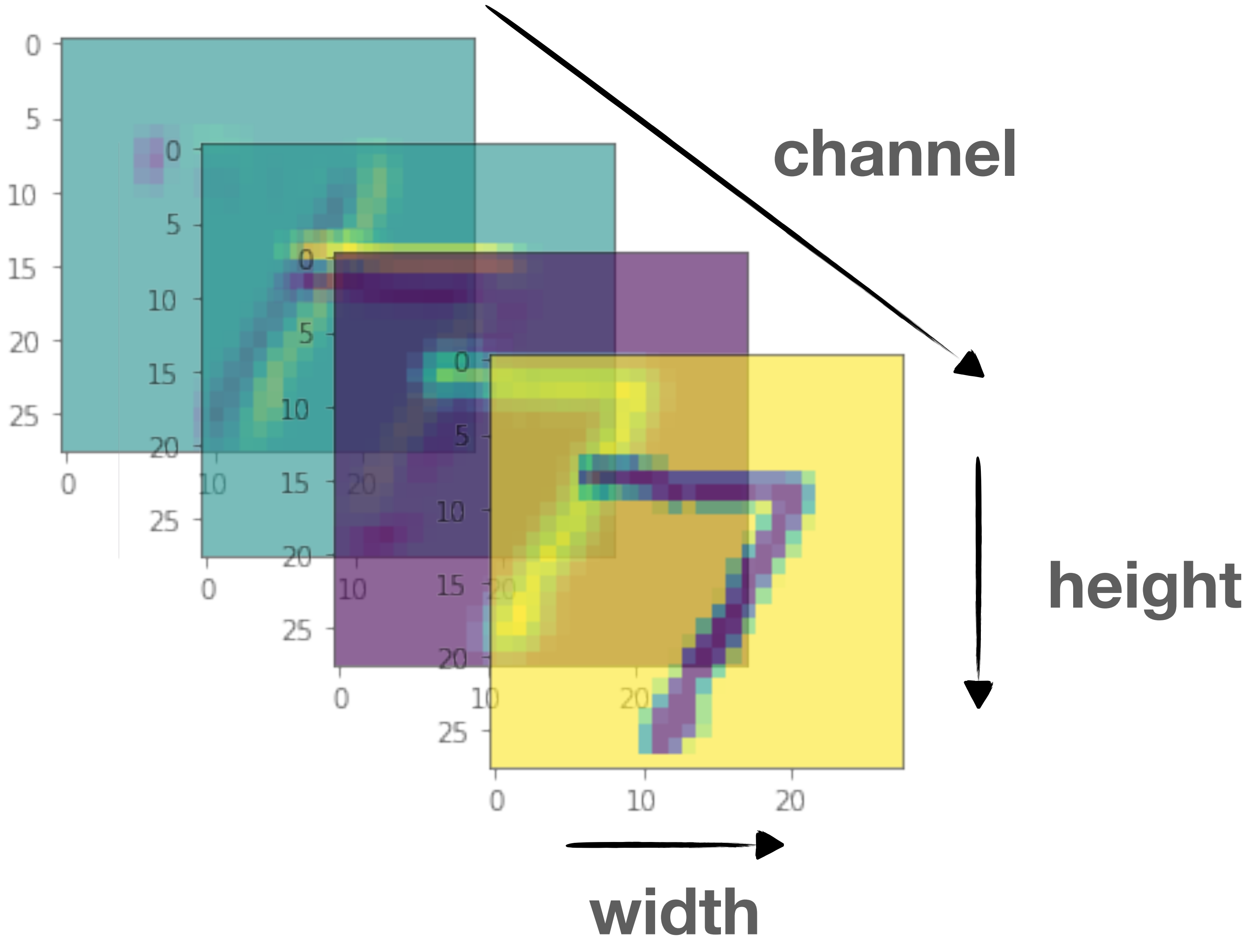
$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

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

Input



Output



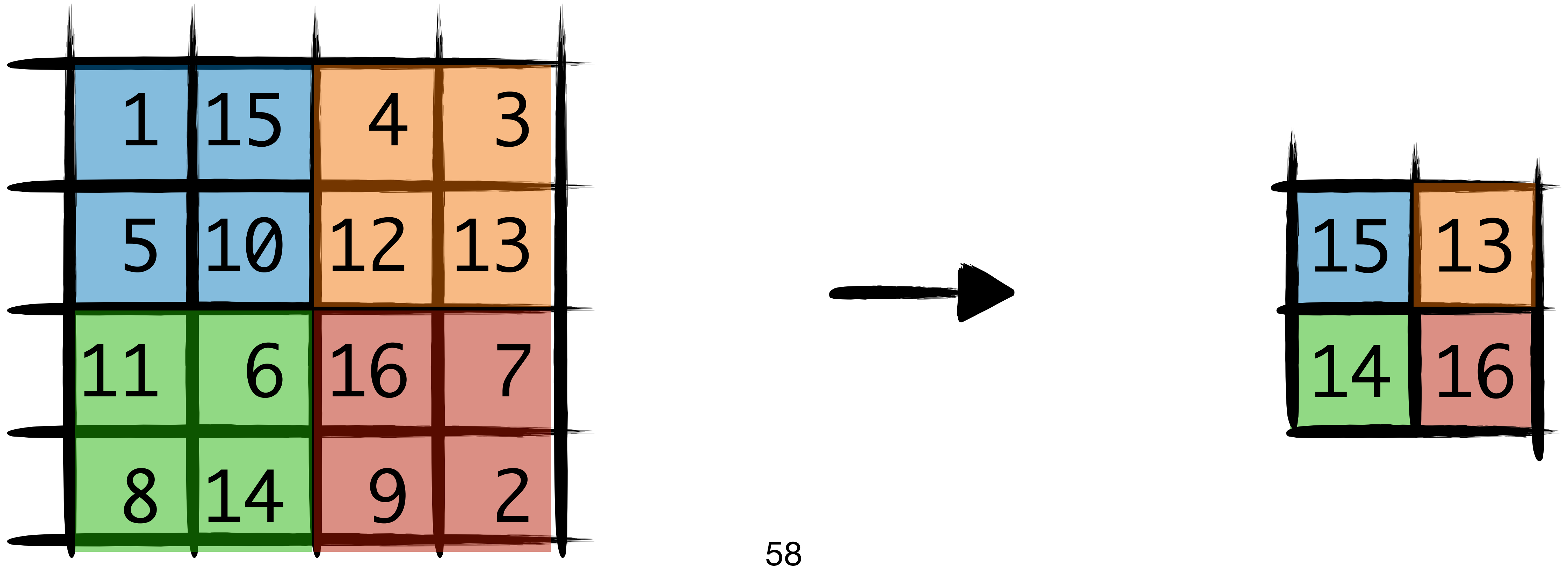
input channels: 1
output channels: 4
kernel size = 3
stride = 1
padding = 1

Pooling

Role of convolutions is to **extract local features**. Pooling **summarizes a local patch** in terms of those features

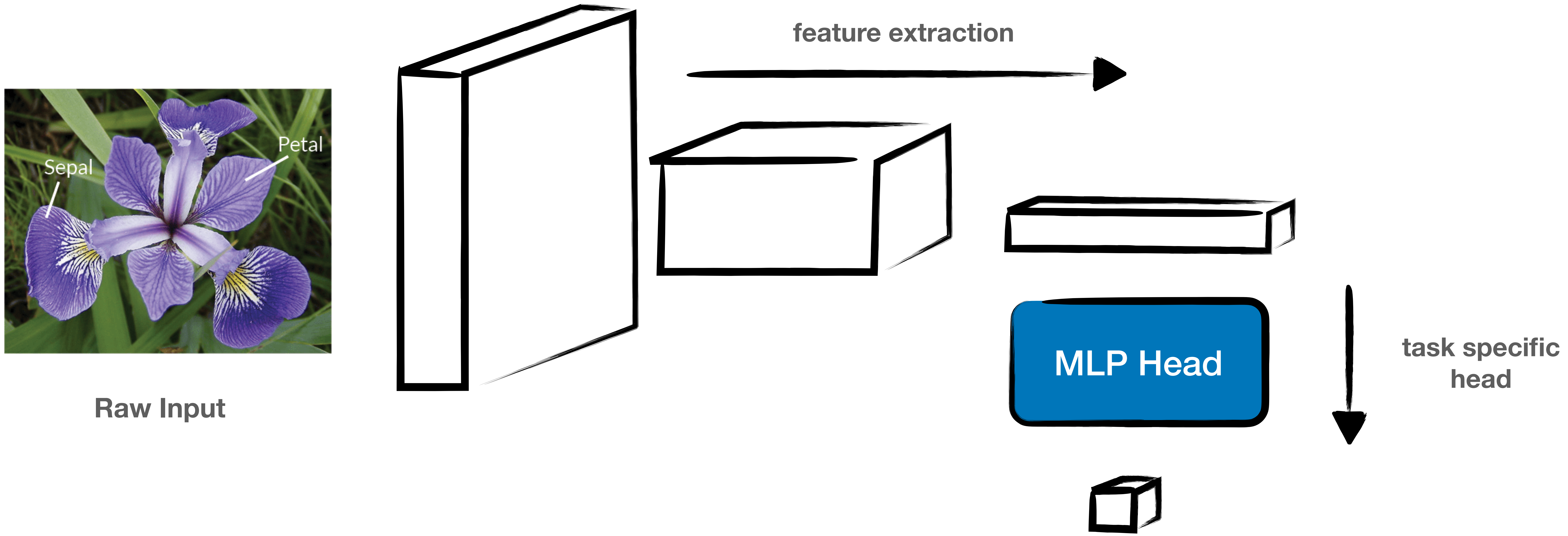
Average Pooling $$y = \frac{1}{N} \sum_{v \in view(y)} x_v$$

Maximum Pooling $$y = \mathrm{argmax}_{\mathrm{v} \in \mathrm{view(y)}} x_v$$



Building CNNs

The full CNN then implements the Deep Learning idea: learned feature extraction, followed by simple MLP head

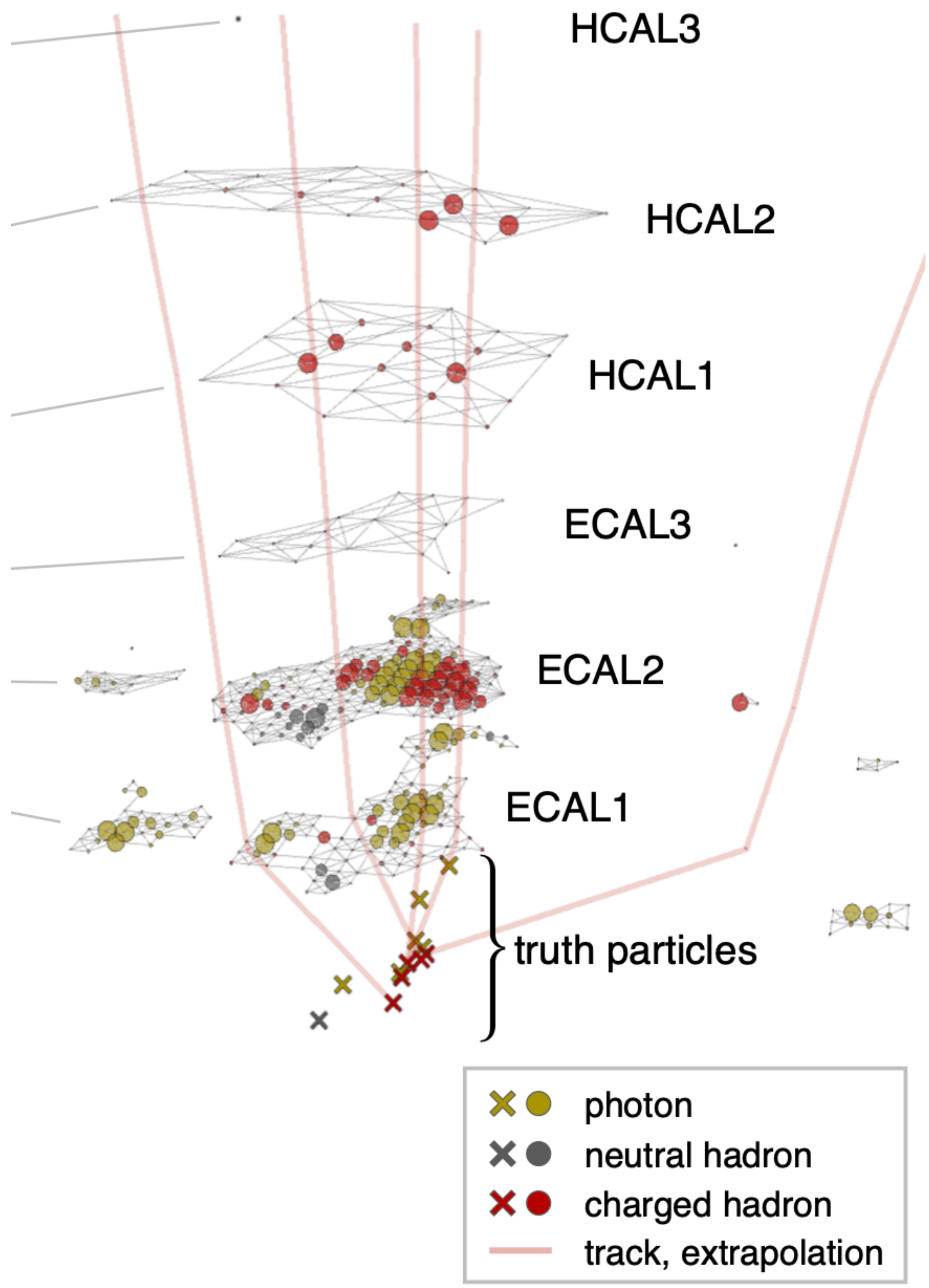


Graph Neural Nets

CNNs excel at data that "lives" on a regular grid. Feature extraction by **combining information from neighborhood.**

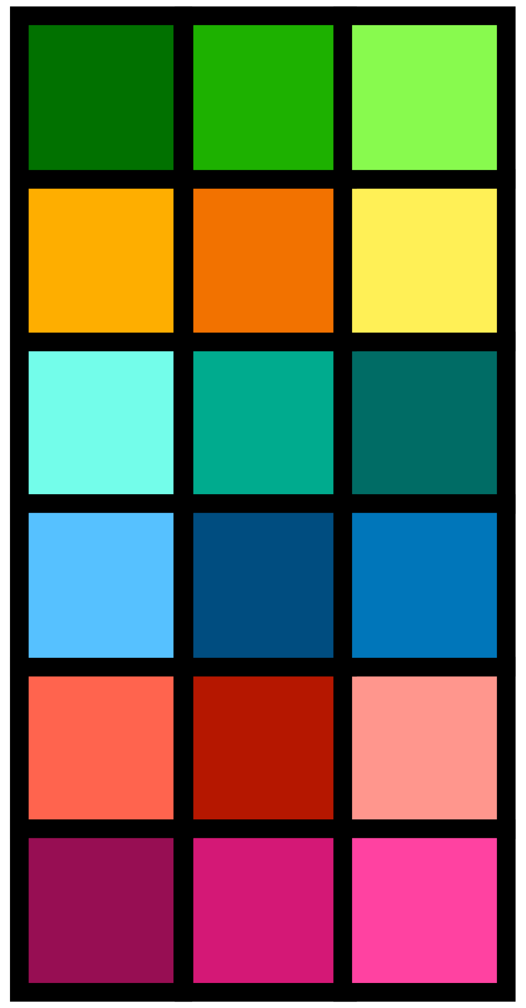
But a lot of data is more irregular.

A local neighborhood defined more by relationships than a grid

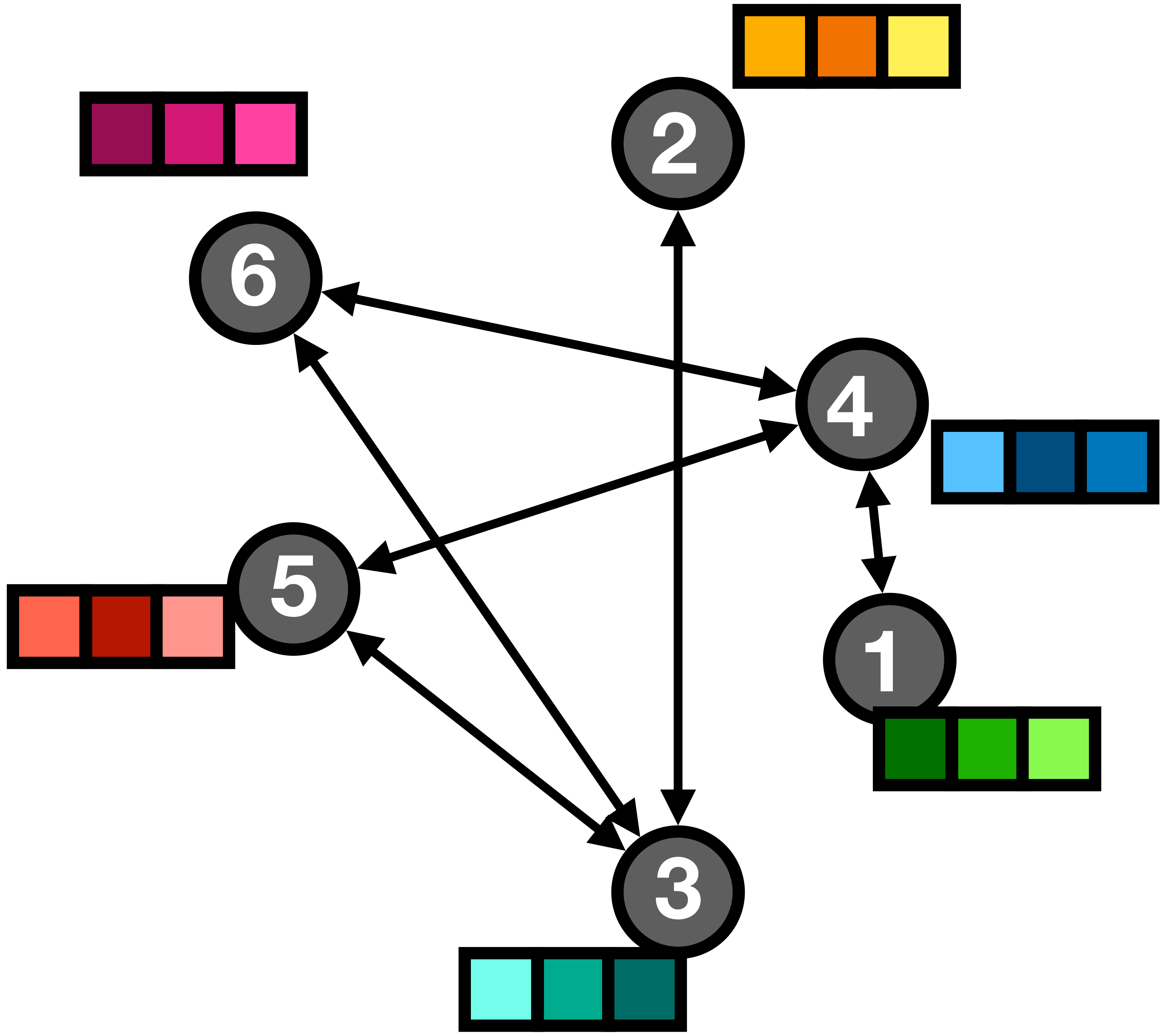


Graph Data

Graphs can still be represented by Matrices, but the processing of graphs must not rely on (arbitrary) order



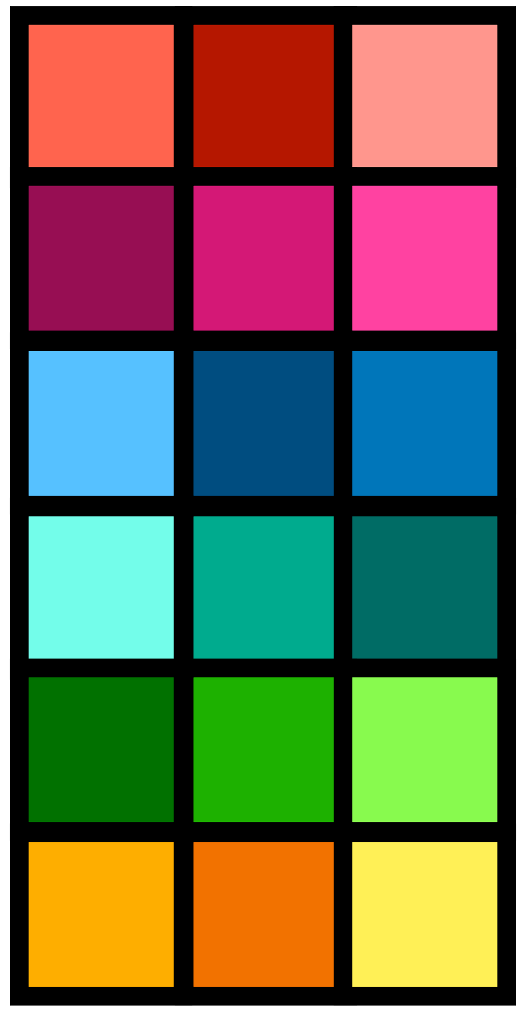
$X \in \mathbb{R}^{n \times f}$



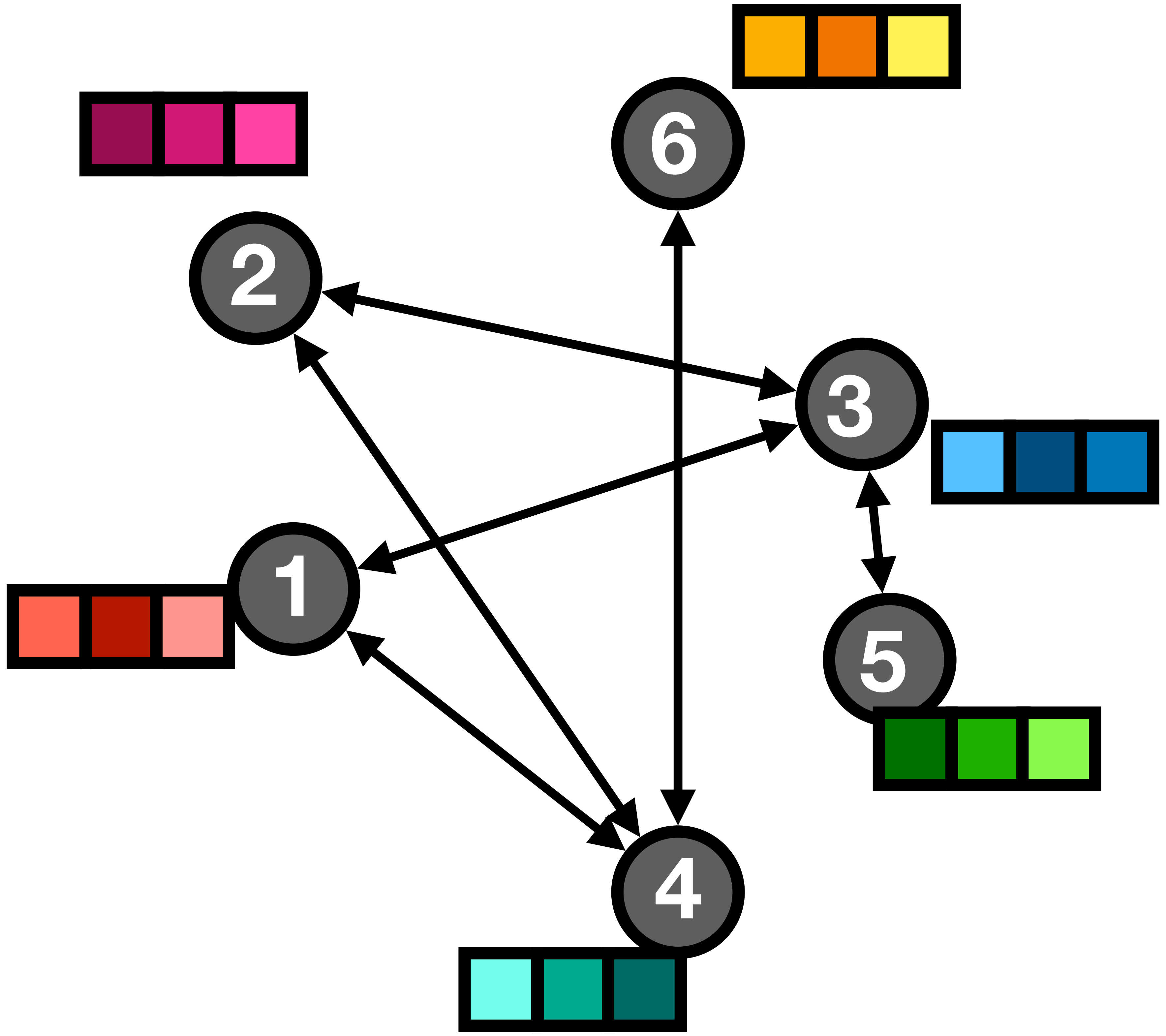
Permutation Invariance

Graph Data

Graphs can still be represented by Matrices, but the processing of graphs must not rely on (arbitrary) order



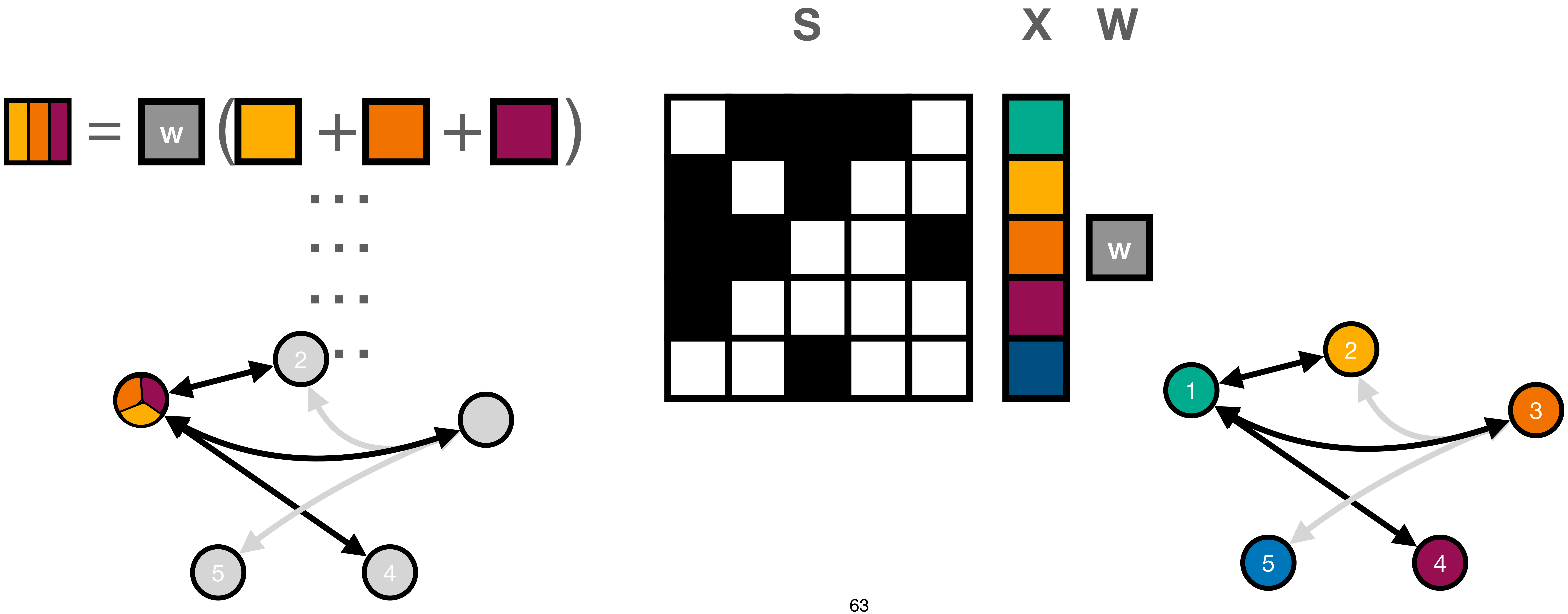
$$X \in \mathbb{R}^{n \times f}$$



Permutation Invariance

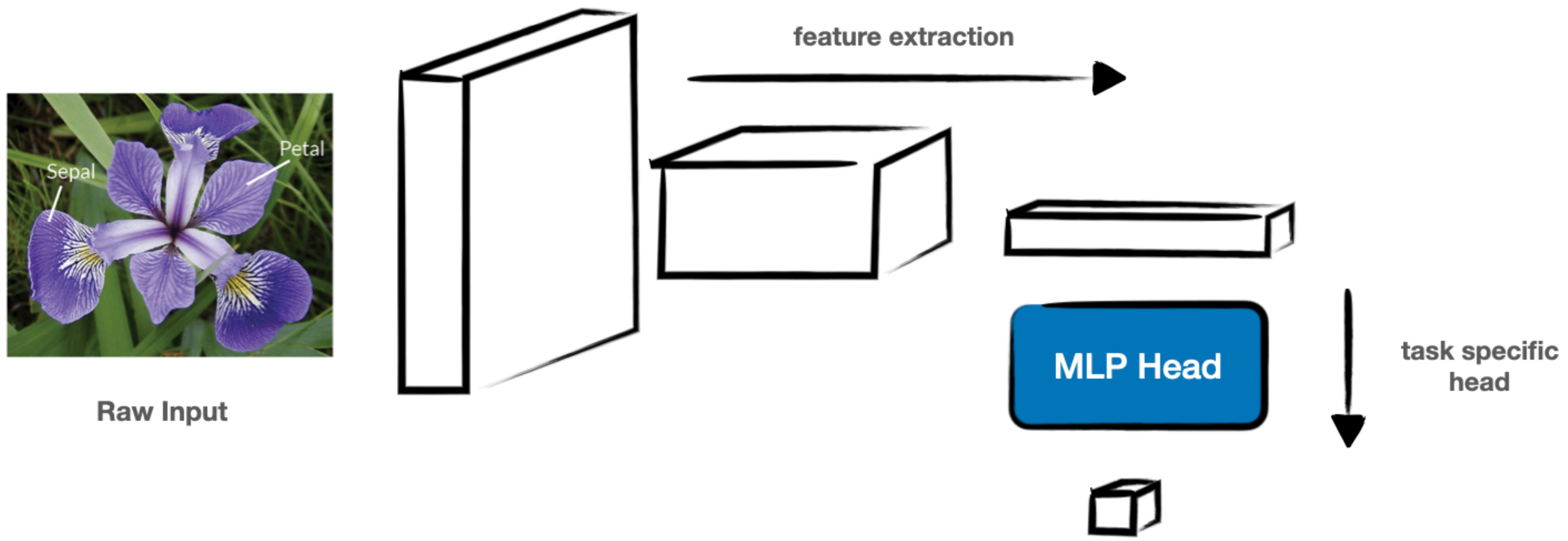
Graph Convolutions

Graph Convolutions generalize CNN convolutions to pass messages from neighbors as defined in the graph

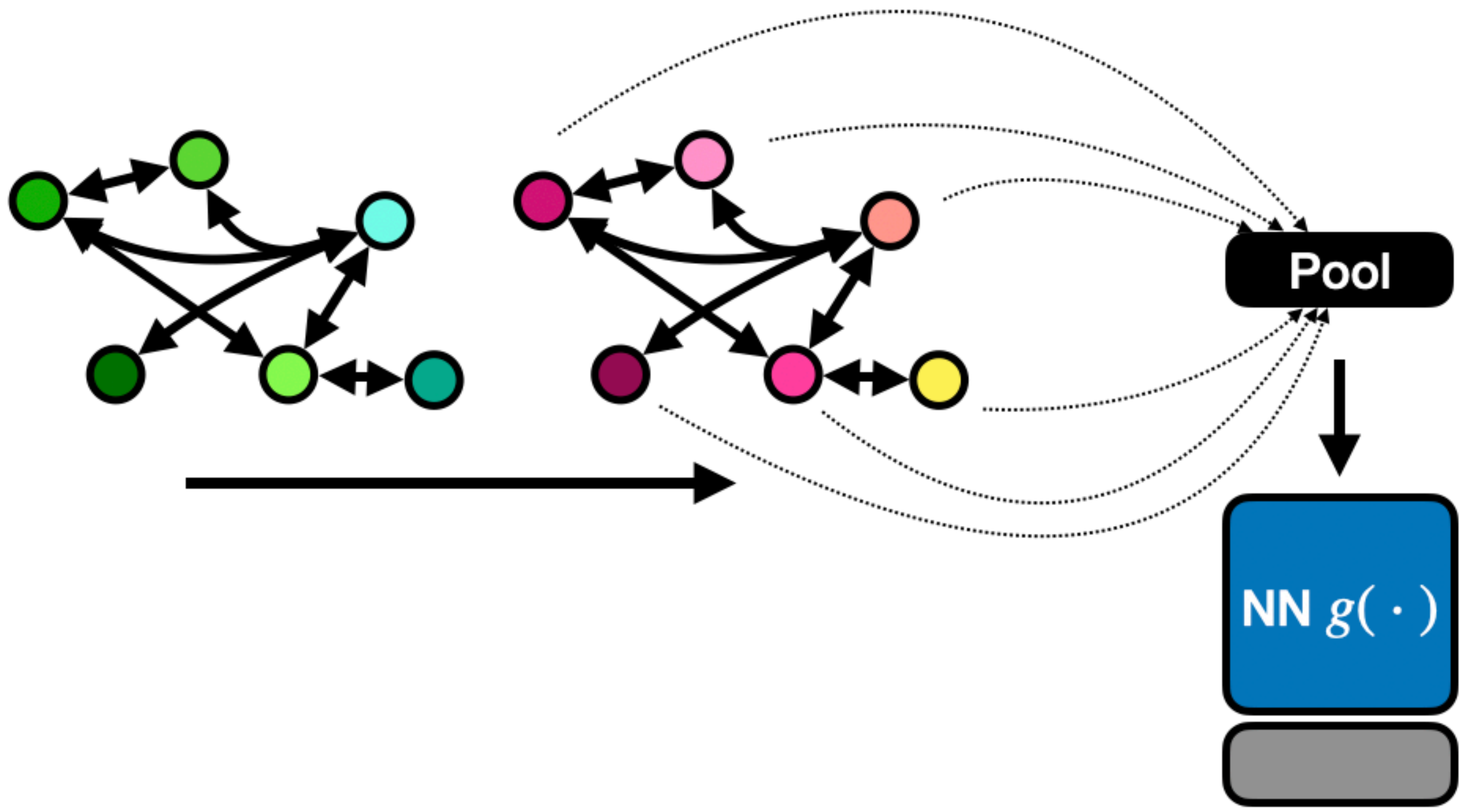


GNNs

As in CNNs, we can stack Graph processing into a stack of feature extraction, and then follow up high-level “head”



CNN



GNN

Dynamic Networks

Neural Nets are functions of the input & network parameters

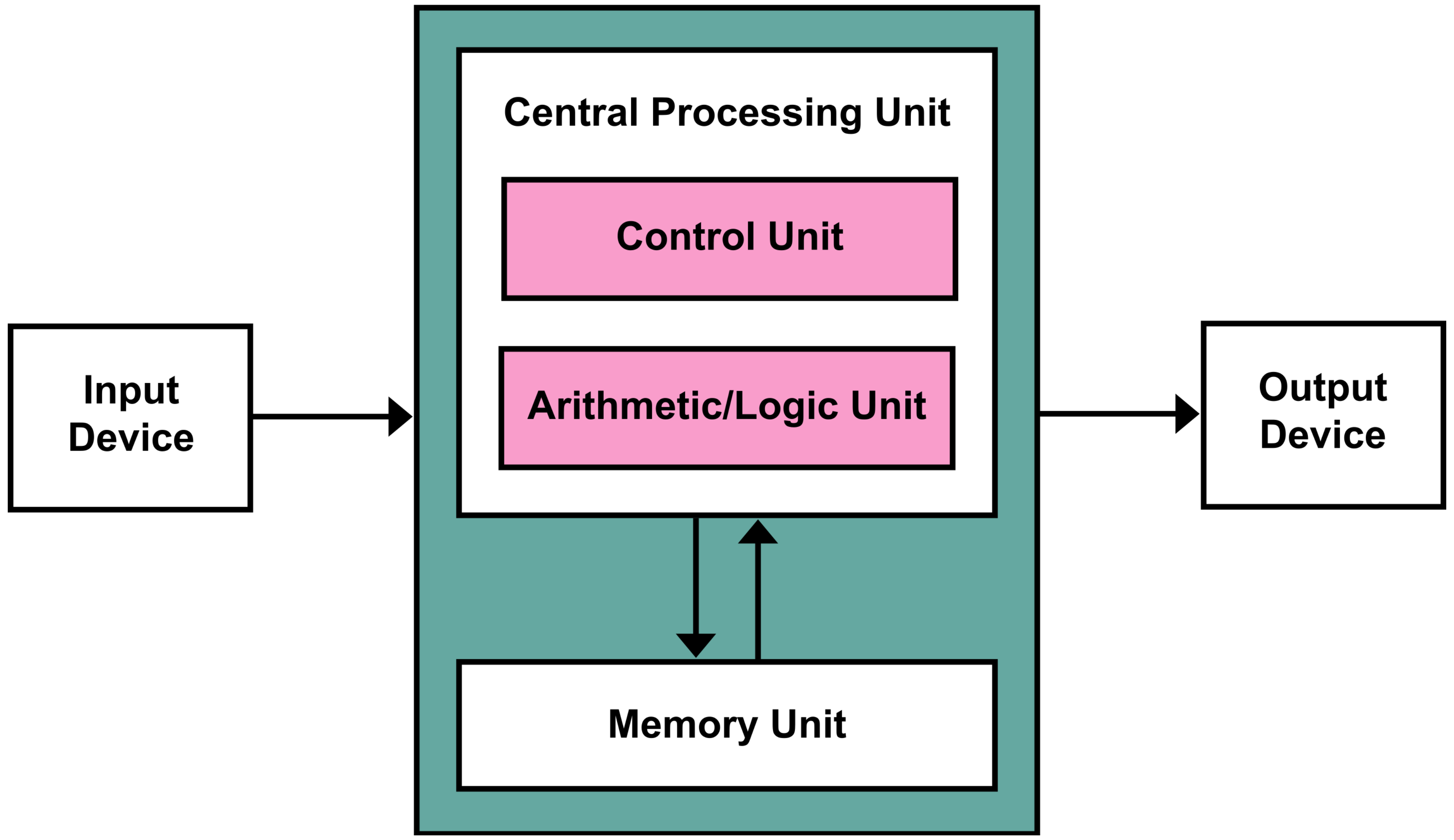
$$f(x, \phi)$$

In their most basic form, both inputs **are static**, but some of the most powerful architectures allow them to be **dynamic**

Recurrent Neural Networks

Often data is variable in size. Recurrent neural networks deal with it like a computer:

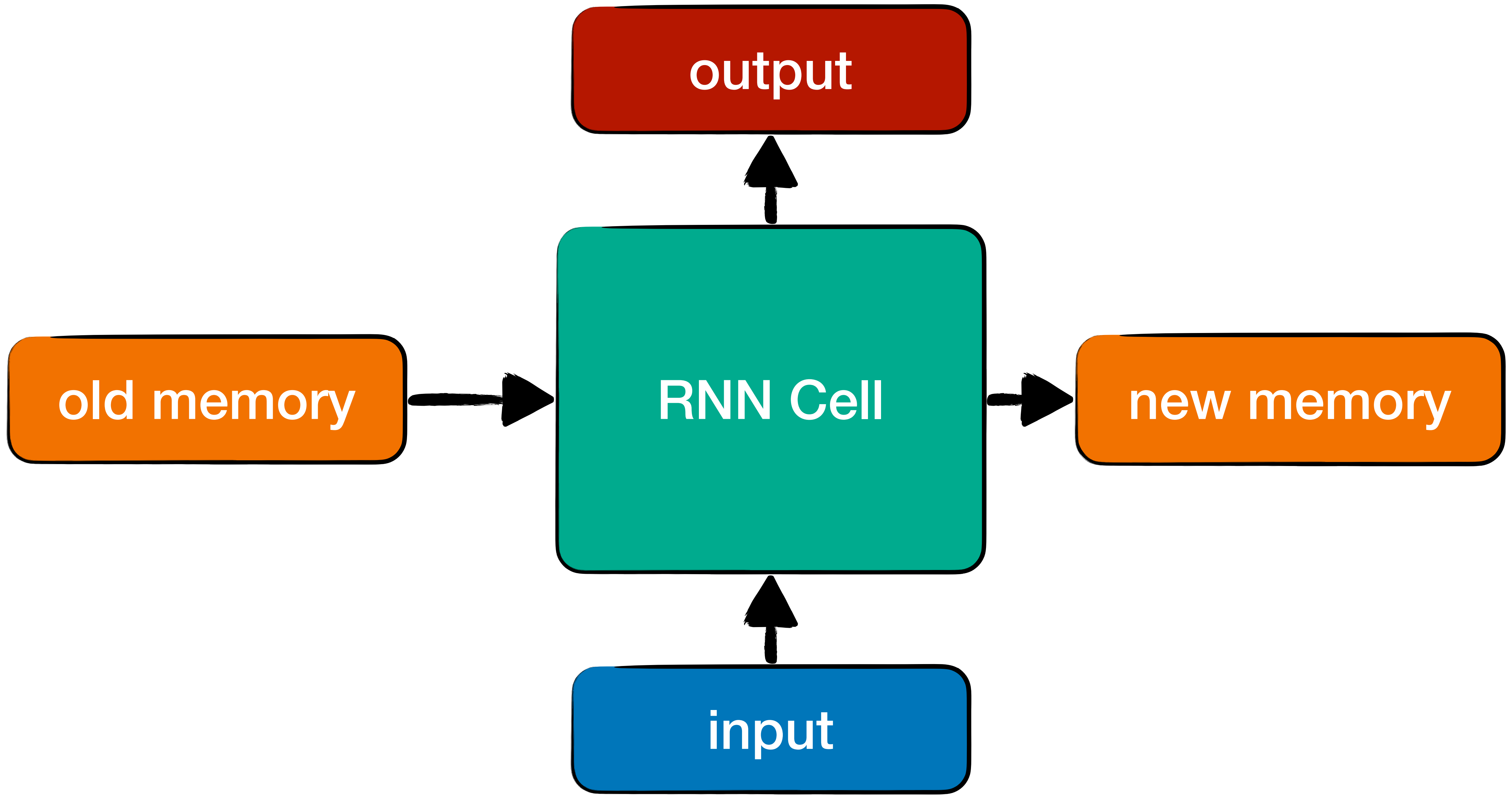
Consume data **step-by-step** and keeping the information within a memory component



von Neuman

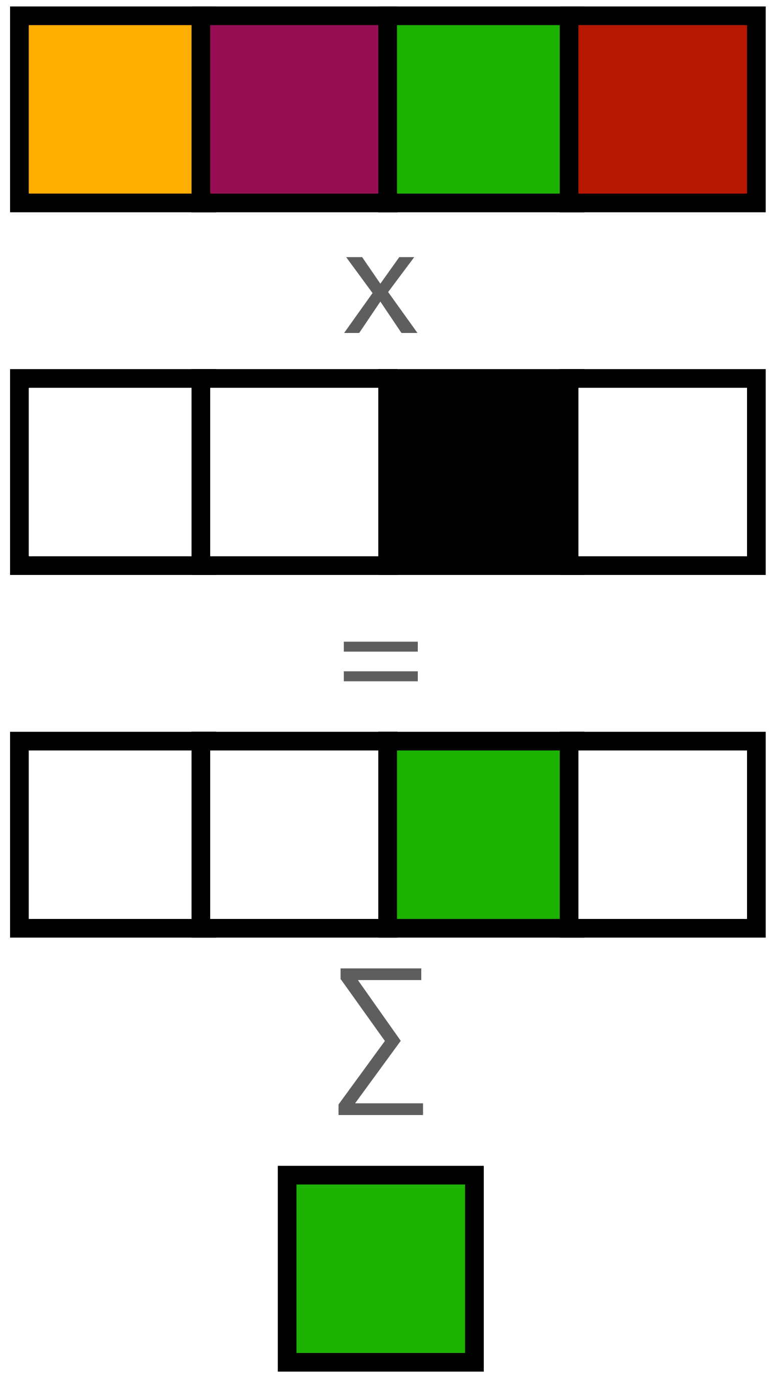
Recurrent Neural Networks

RNNs learn an update function for a memory vector, which can be applied many times until the inputs are exhausted

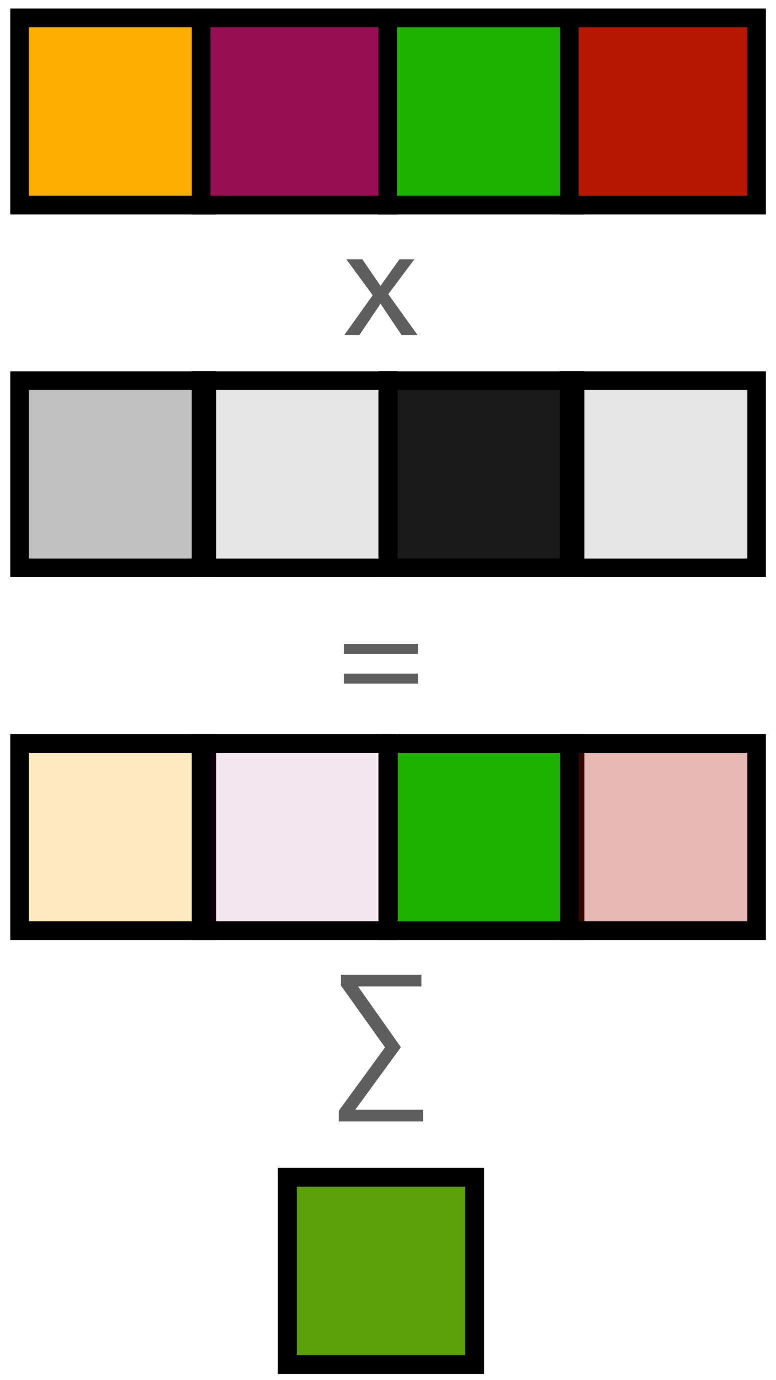


Active Neurons and Gates

RNNs interact with memory like a “soft” computer reading and writing to memory via “gates”, i.e. multiplicative masks



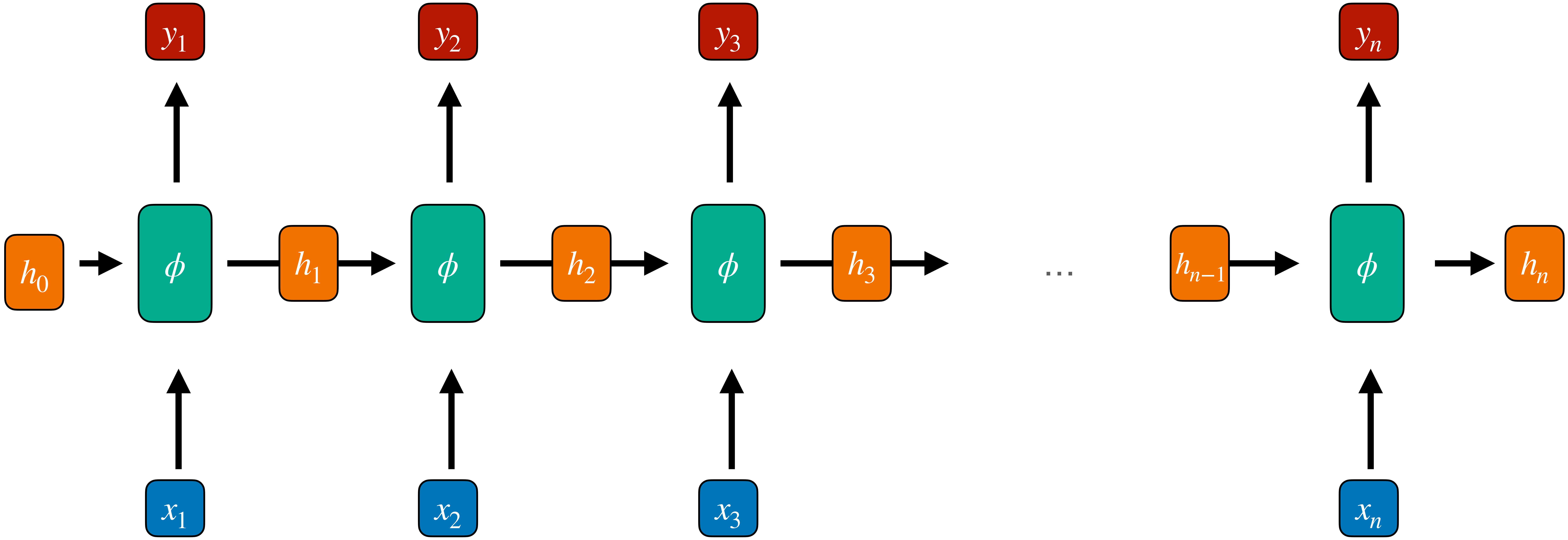
Hard Read



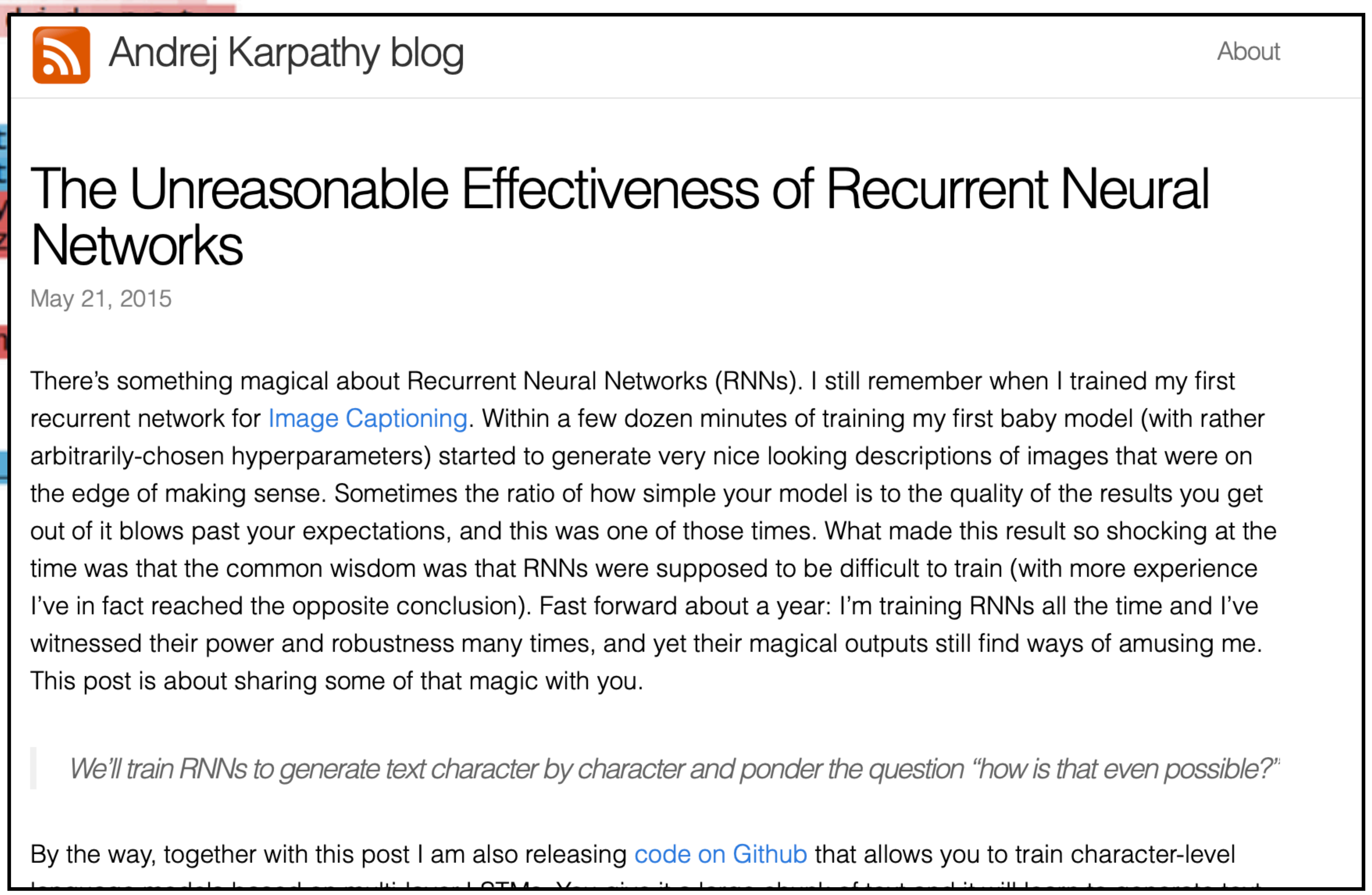
Soft Read

Recurrent Neural Networks

Can be applied to arbitrary length sequences



RNN in a time before ChatGPT



Andrej Karpathy blog — About

The Unreasonable Effectiveness of Recurrent Neural Networks

May 21, 2015

There's something magical about Recurrent Neural Networks (RNNs). I still remember when I trained my first recurrent network for Image Captioning. Within a few dozen minutes of training my first baby model (with rather arbitrarily-chosen hyperparameters) started to generate very nice looking descriptions of images that were on the edge of making sense. Sometimes the ratio of how simple your model is to the quality of the results you get out of it blows past your expectations, and this was one of those times. What made this result so shocking at the time was that the common wisdom was that RNNs were supposed to be difficult to train (with more experience I've in fact reached the opposite conclusion). Fast forward about a year: I'm training RNNs all the time and I've witnessed their power and robustness many times, and yet their magical outputs still find ways of amusing me. This post is about sharing some of that magic with you.

> *We'll train RNNs to generate text character by character and ponder the question "how is that even possible?"*

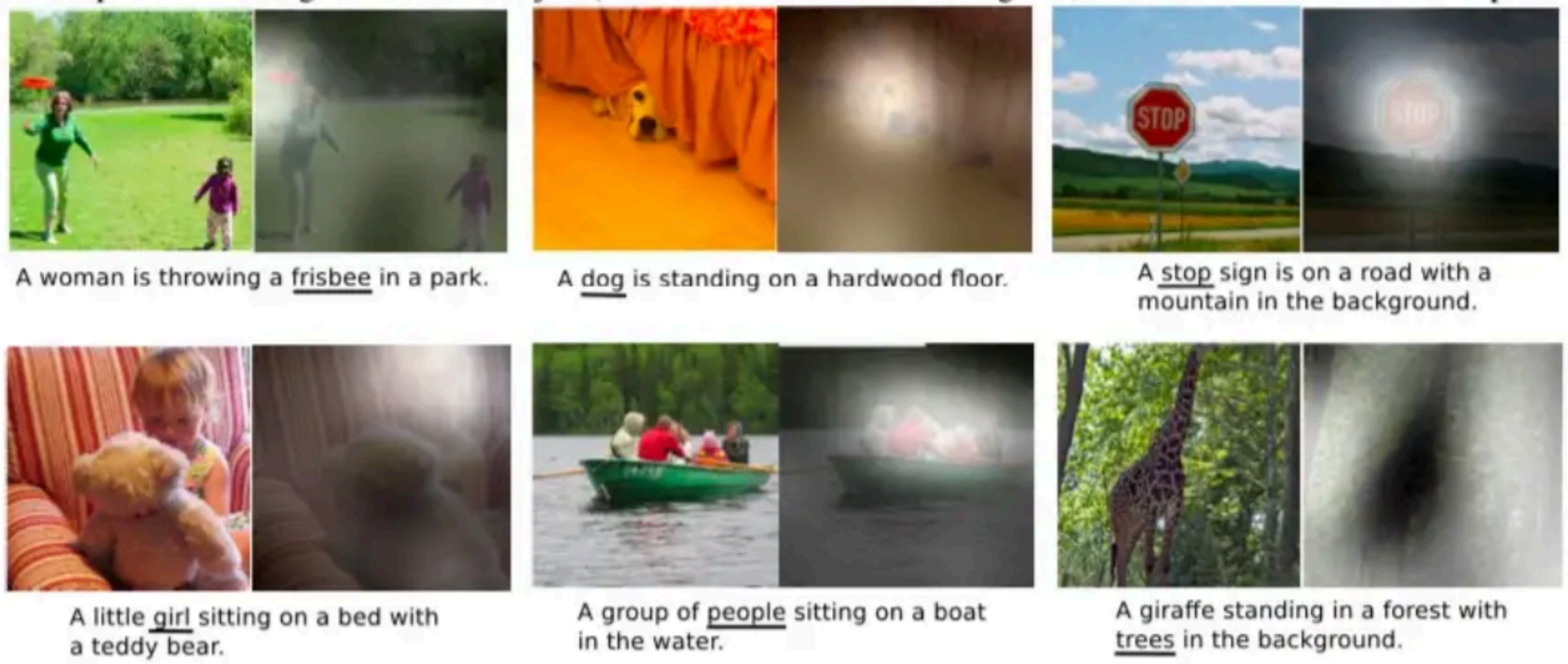
By the way, together with this post I am also releasing code on Github that allows you to train character-level

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Attention Mechanism

The notion of gating / dynamically controlling the flow of information is also key to one of the most impactful ideas in Deep Learning: **Attention**

Figure 3. Examples of attending to the correct object (*white* indicates the attended regions, *underlines* indicated the corresponding word)



Attention Mechanism

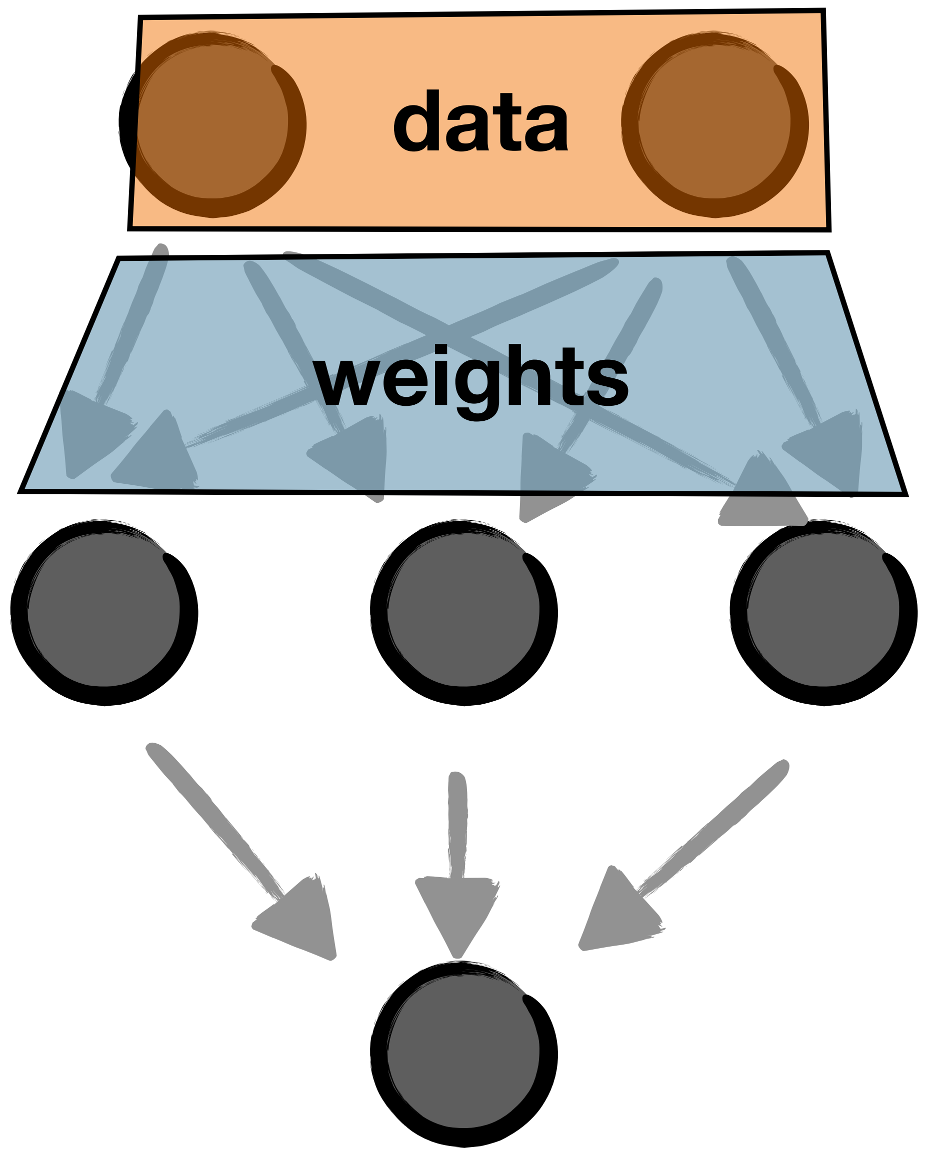
Standard Neural Nets, have globally fixed data processing
Attention Mechanisms add a **data-dependent** processing

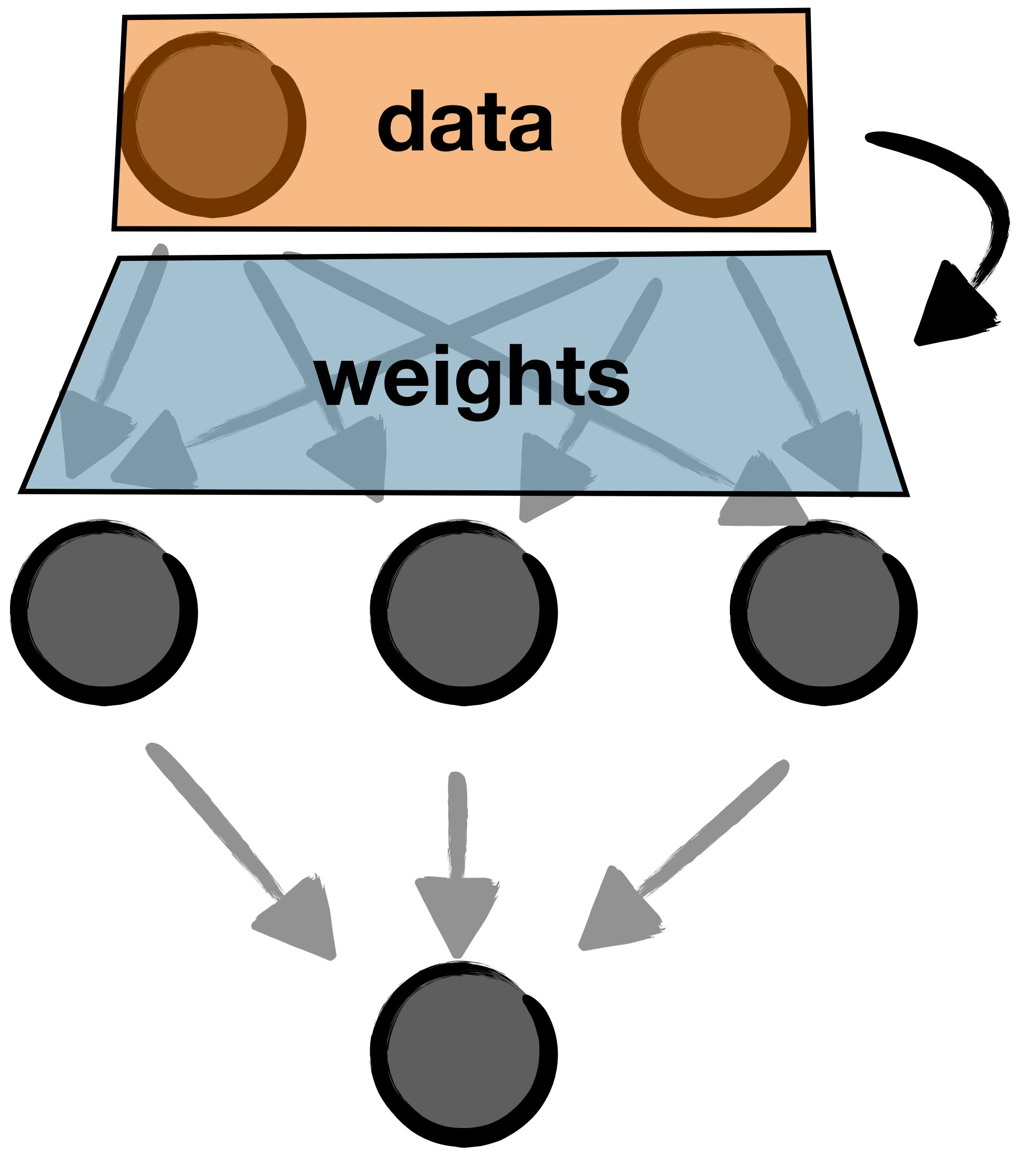
$$y = Wx \longrightarrow y = A(x)\,x$$

Standard Neural Net

Network with attention

weights are fixed
by the training data
input is just passed through

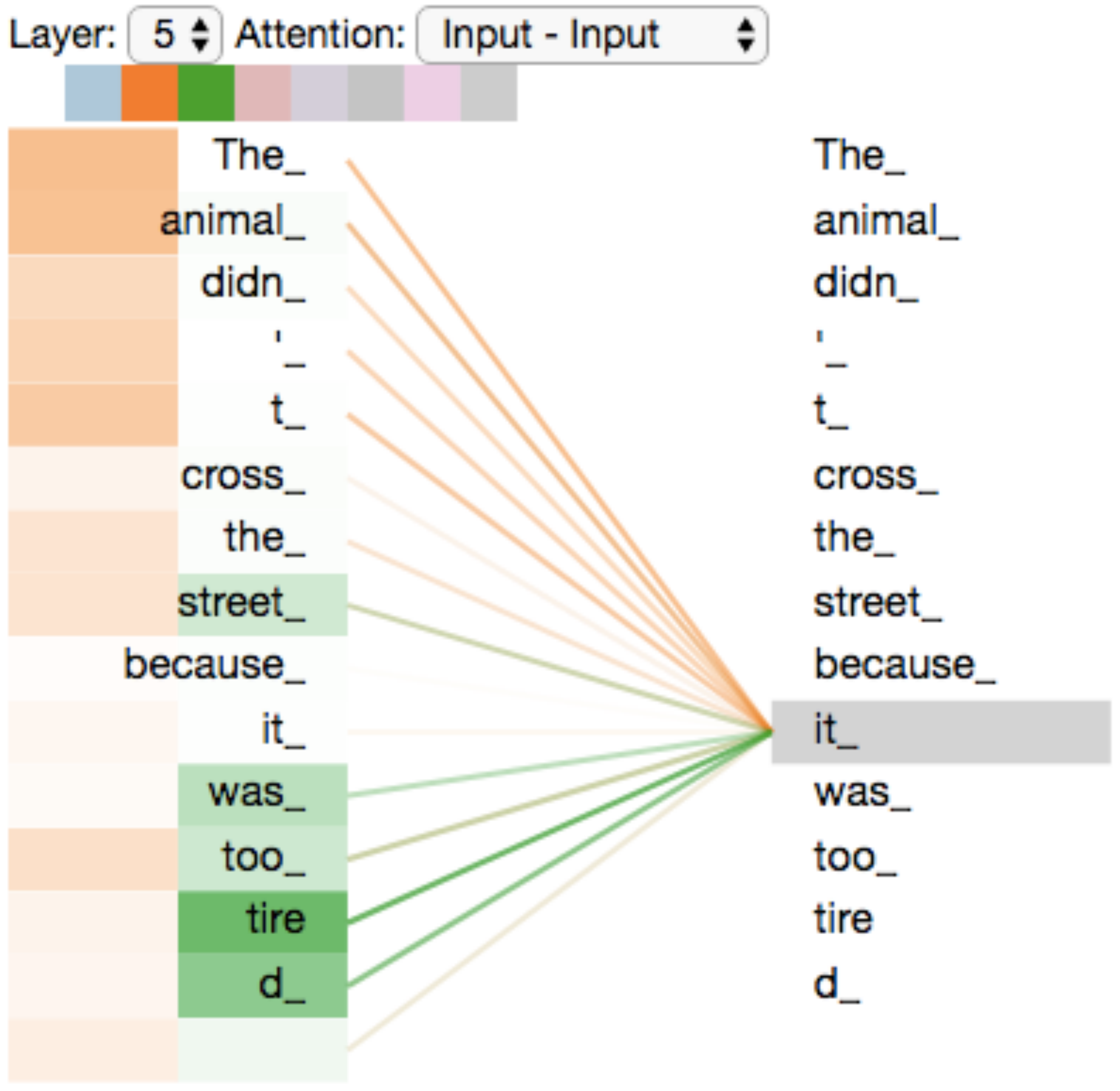


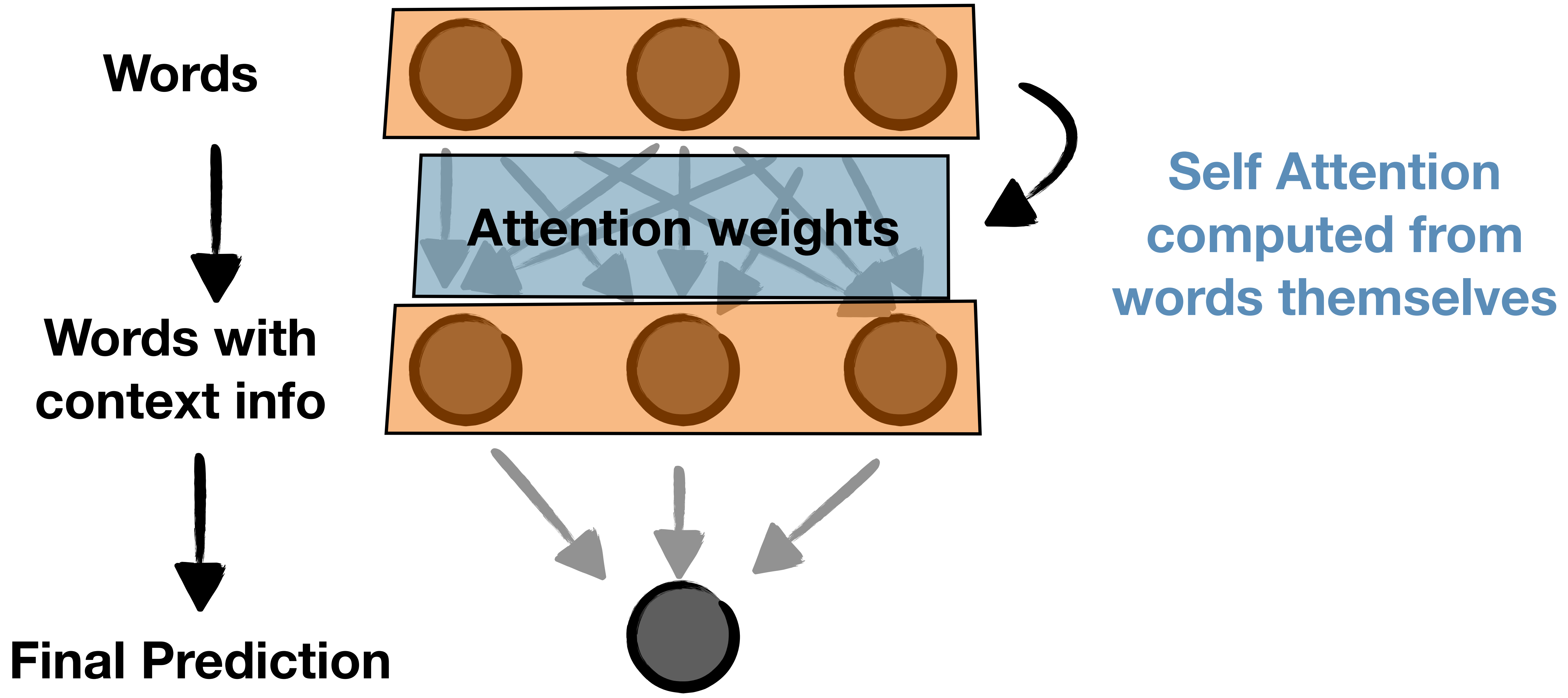


input data influences
the weights at
the time of processing

Attention in Language

Example: when representing words with added context, decide dynamically which other words are relevant





Transformers

Attention is *the* key idea in transformer networks that have e.g. largely replaced RNN-based models for text

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

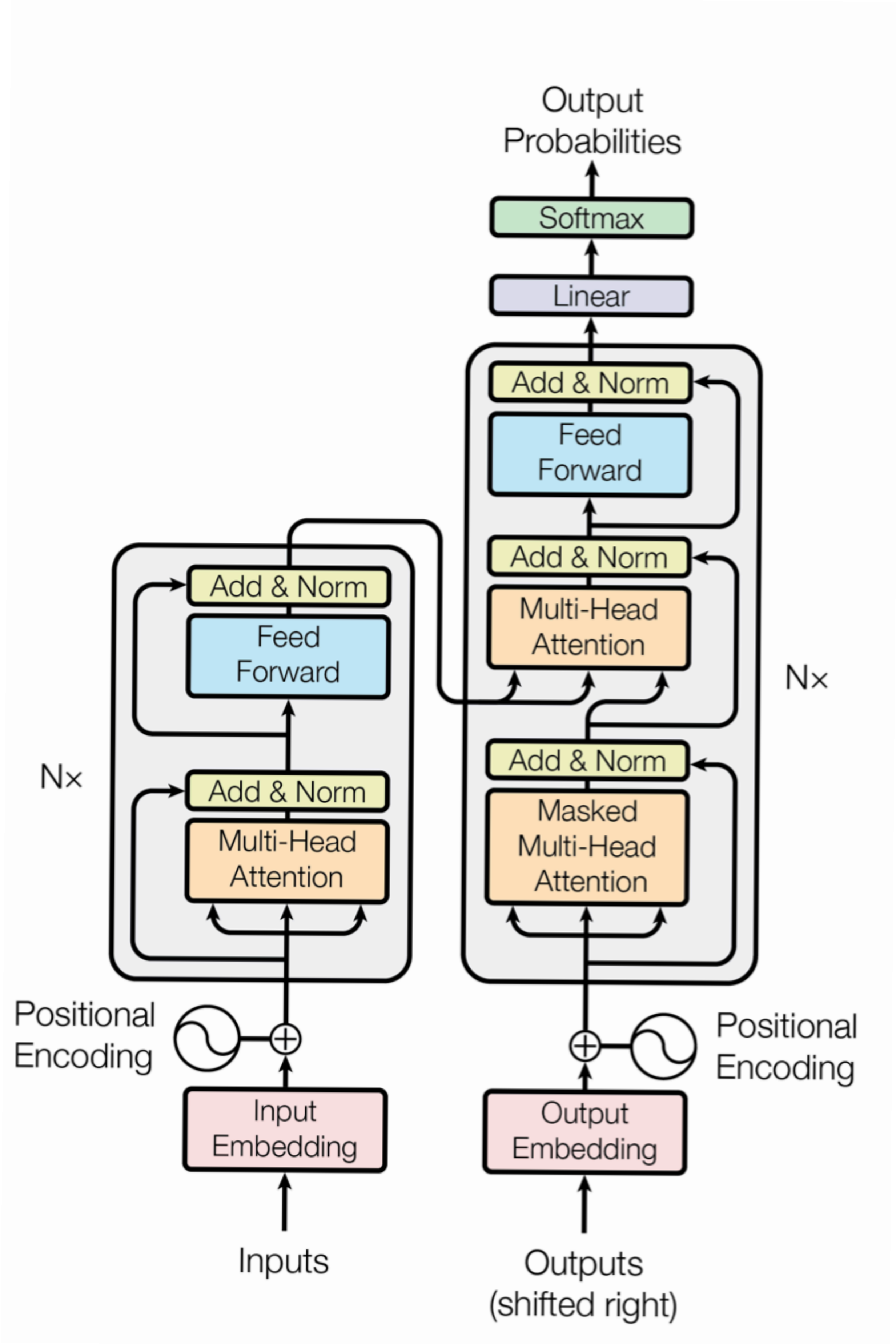
Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com



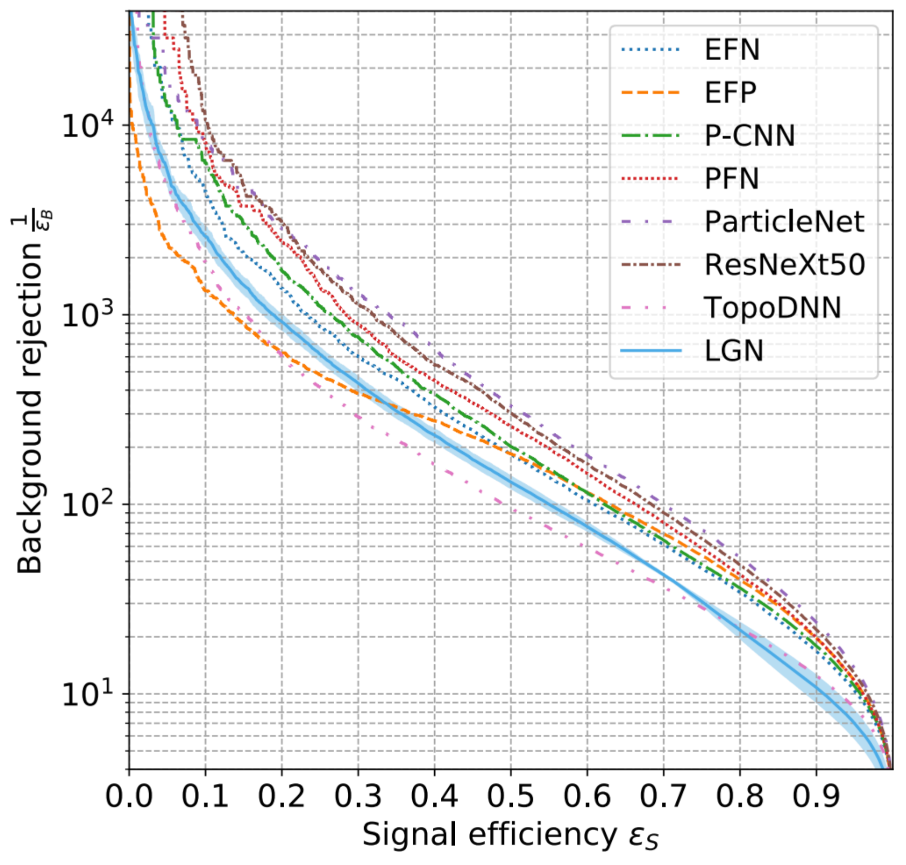
Phyics Inspired Architectures

Physicists were quick to add their own symmetries to inject physics inductive bias to neural networks

Nets with Jet Structure

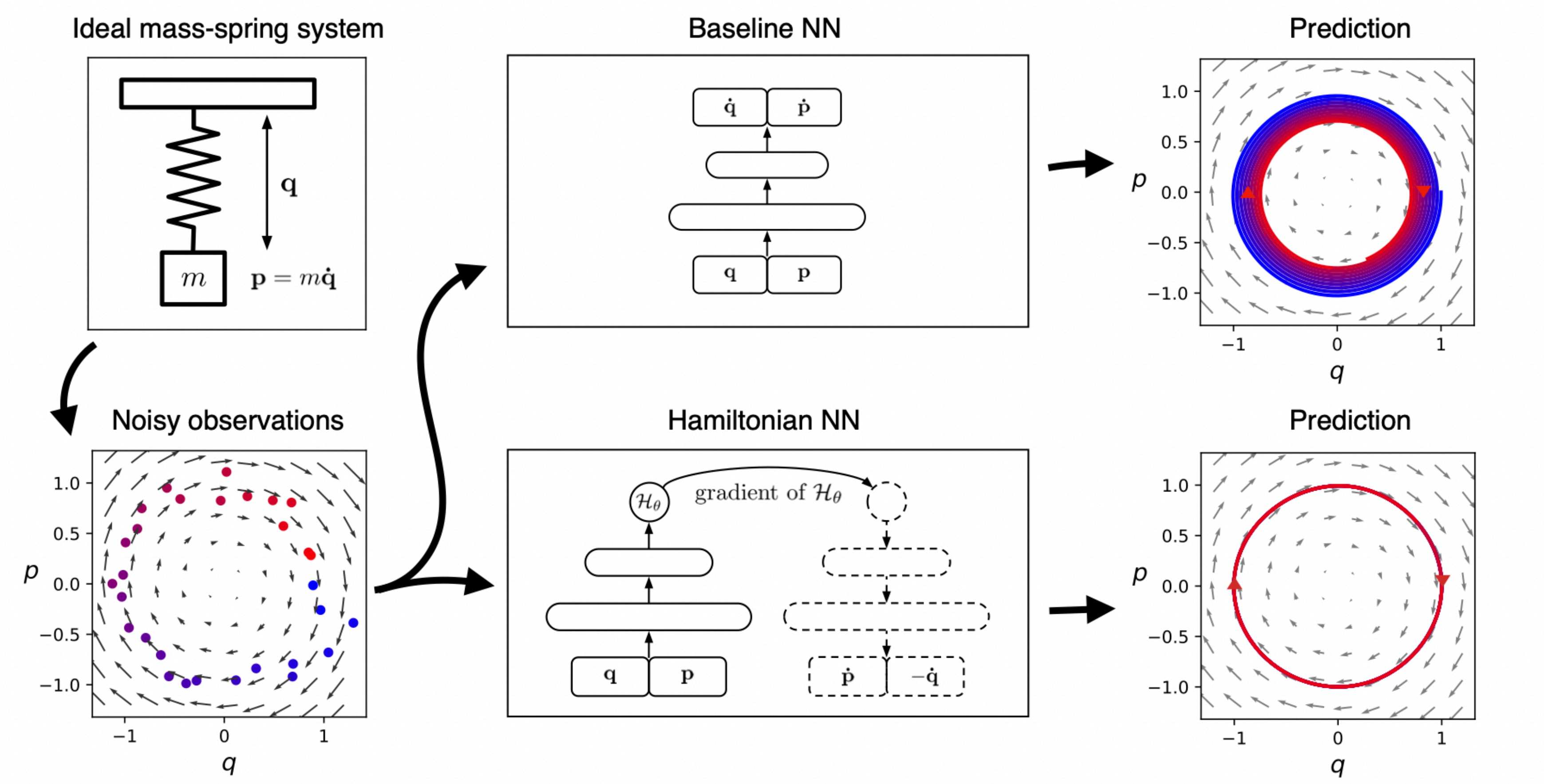
arXiv:1702.00748

Architecture	Accuracy	AUC	$1/\epsilon_B$	#Param
ParticleNet	0.938	0.985	1298 ± 46	498k
P-CNN	0.930	0.980	732 ± 24	348k
ResNeXt	0.936	0.984	1122 ± 47	1.46M
EFP	0.932	0.980	384	1k
EFN	0.927	0.979	633 ± 31	82k
PFN	0.932	0.982	891 ± 18	82k
TopoDNN	0.916	0.972	295 ± 5	59k
LGN	0.929 $\pm$ .001	0.964 $\pm$ 0.018	435 ± 95	4.5k



Lorentz-Invariance

arXiv:2006.04780

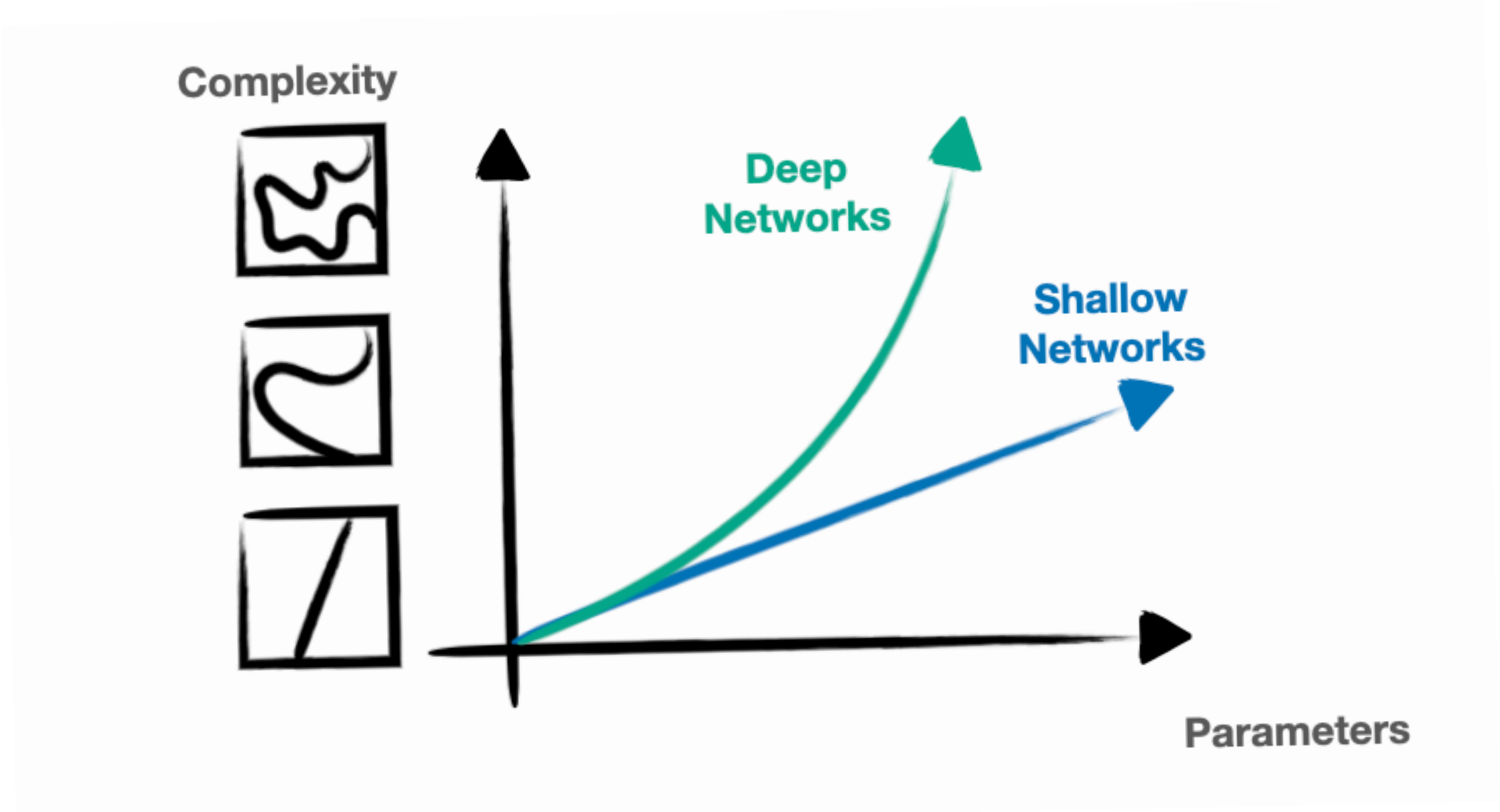


Hamiltonian Neural Nets

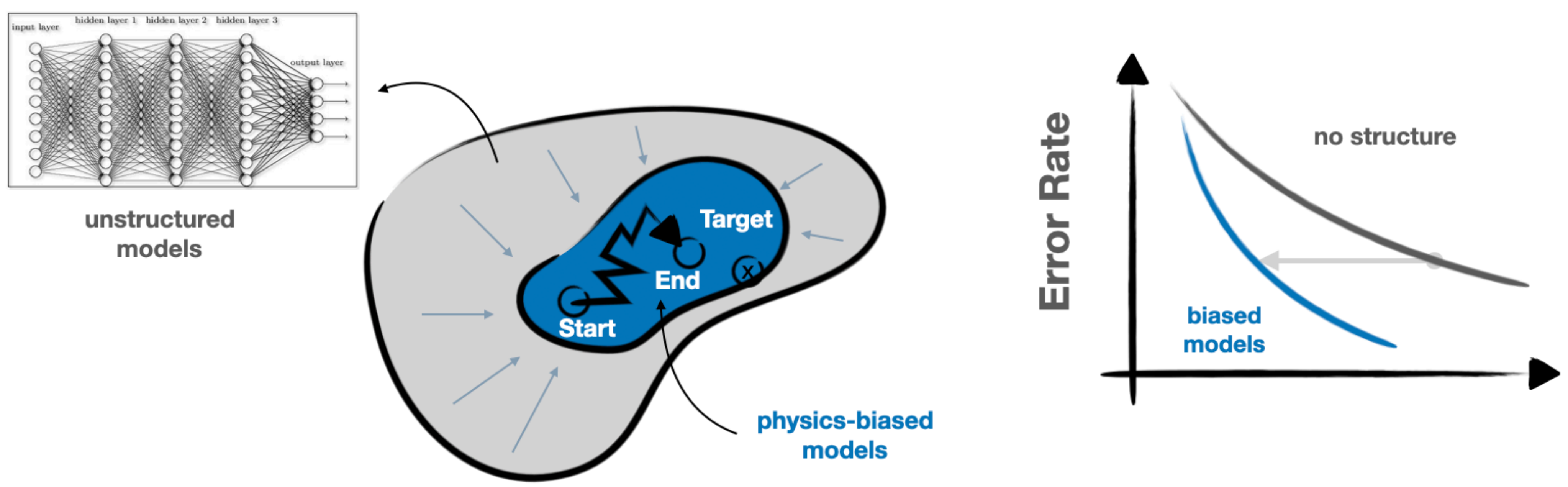
arXiv:1906.01563

Summary

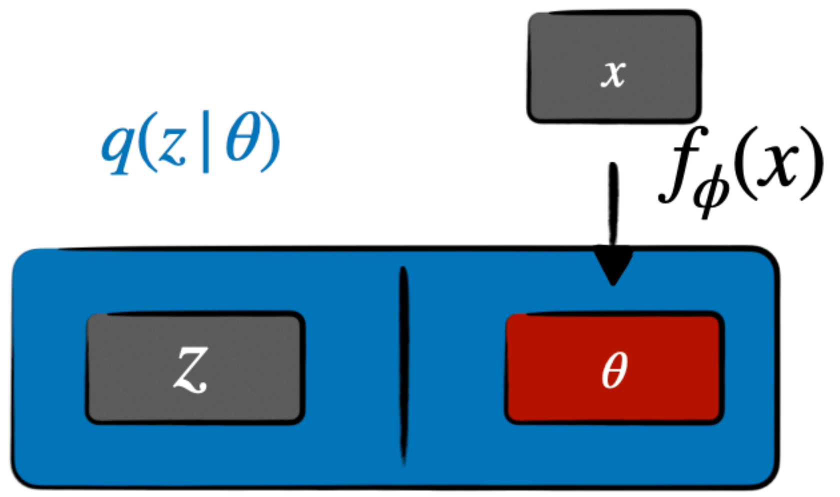
Benefits of Depth



Inductive bias



Cross Entropy Loss

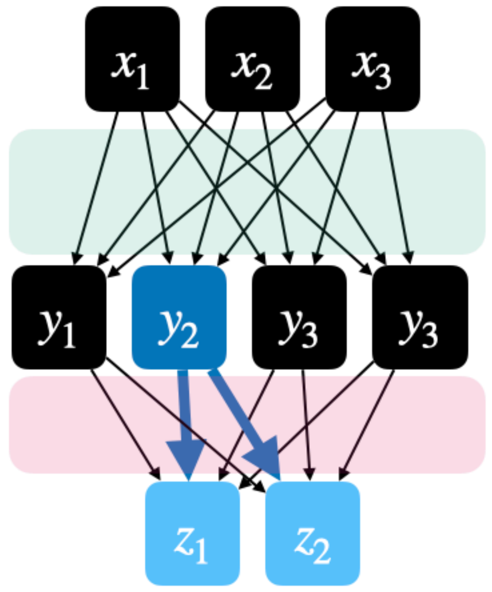


$$L(\phi) = -\mathbb{E}_{p(x,z)} \log q_\phi(z \mid x)$$

Gradients via Backprop

$$J_{x \to y} = \frac{\partial y_i}{\partial x_j}$$

$$J_{y \to z} = \frac{\partial z_i}{\partial y_j}$$



$$g_i^y = (g^z J)_i = \sum_{c \in \mathrm{children}(y_i)} g_c^z \frac{\partial z_c}{\partial z_i}$$