Supervised learning and introduction to deep learning

Gül Varol

IMAGINE team, École des Ponts ParisTech

gul.varol@enpc.fr

http://imagine.enpc.fr/~varolg/

@RecVis, 29.10.2024

With many slides from: J. Sivic, R. Fergus, A. Vedaldi, A. Zisserman, S. Lazebnik, A. Karpathy, L. Fei Fei, M. Aubry, N. Murray, J. Johnson, K. Derpanis, S. Yeung, A. Amini, F. Fleuret, A. Joulin, J. Austin, B. Liu, A. Jain

École des Ponts ParisTech

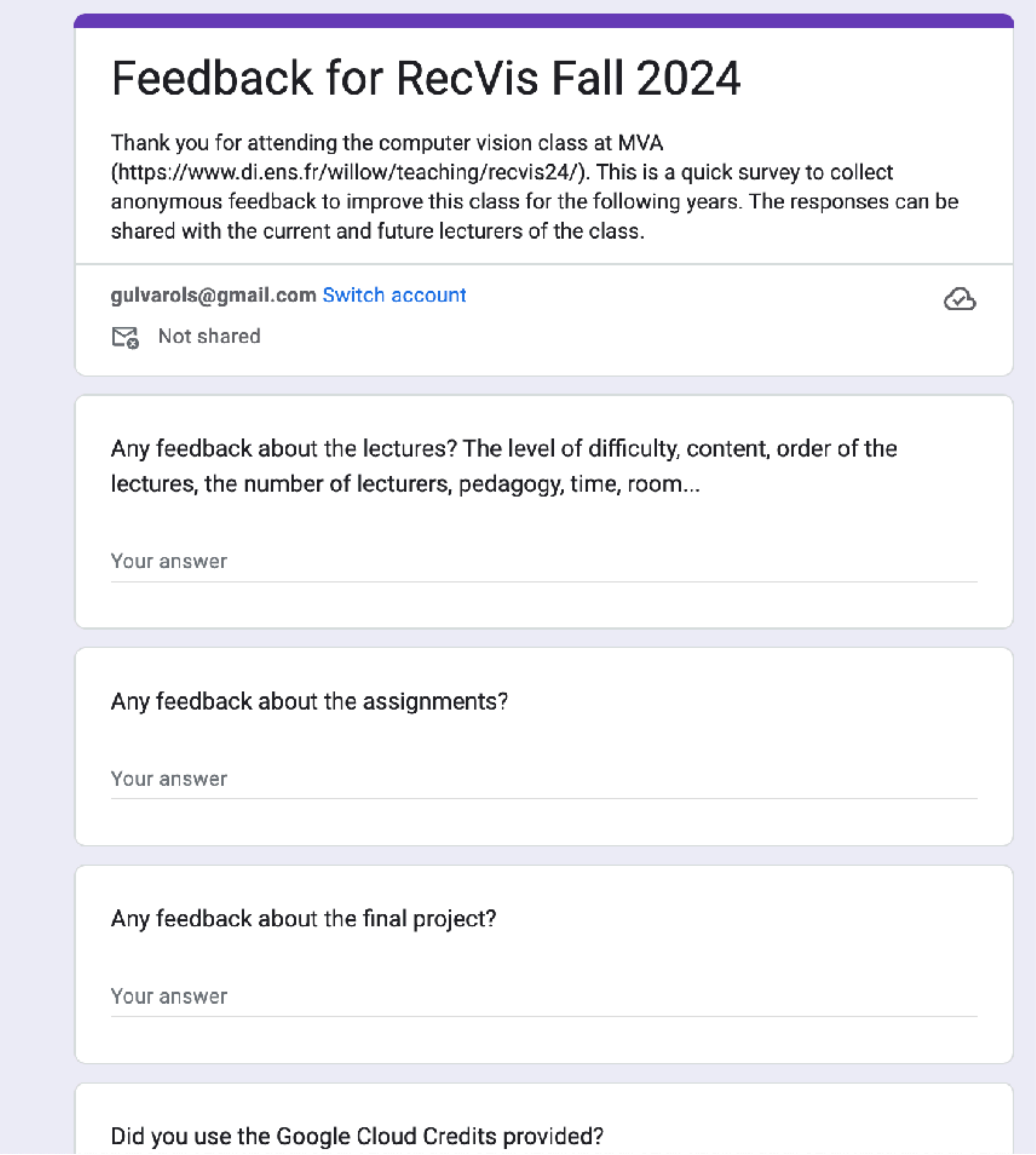
Announcements

- Assignment 2 out
- Assignment 1 due today
- Read papers to pick a final project topic

Announcements

- Assignment 2 out
- Assignment 1 due today
- Read papers to pick a final project topic
- Reminder: Feedback form
  - You do not have to wait the end of the class,
  - You can fill again.

https://forms.gle/x1ktzhnHmURCRHJF7



Feedback for RecVis Fall 2024

Thank you for attending the computer vision class at MVA (https://www.di.ens.fr/willow/teaching/recvis24/). This is a quick survey to collect anonymous feedback to improve this class for the following years. The responses can be shared with the current and future lecturers of the class.

gulvarols@gmail.com Switch account

Not shared

Any feedback about the lectures? The level of difficulty, content, order of the lectures, the number of lecturers, pedagogy, time, room...

Your answer

Any feedback about the assignments?

Your answer

Any feedback about the final project?

Your answer

Did you use the Google Cloud Credits provided?

Today: Introduction to deep learning

[Week 1] Introduction, local features and matching

[Week 2] Camera geometry, image processing (J. Ponce)

[Week 3] Efficient visual search

[TODAY] Introduction to neural networks, training NNs

[Next weeks]

- Neural networks for **visual** recognition: CNNs and **image classification**
- Beyond CNNs: **Transformers**;
- Beyond classification: **other visual tasks**
- ...

Agenda:

- Machine learning concepts

- Basics of supervised learning

- Introduction to neural networks

- Training neural networks
    - Loss
    - Gradient descent and variants
    - Learning rate
    - Backpropagation
    - Regularization

Agenda:

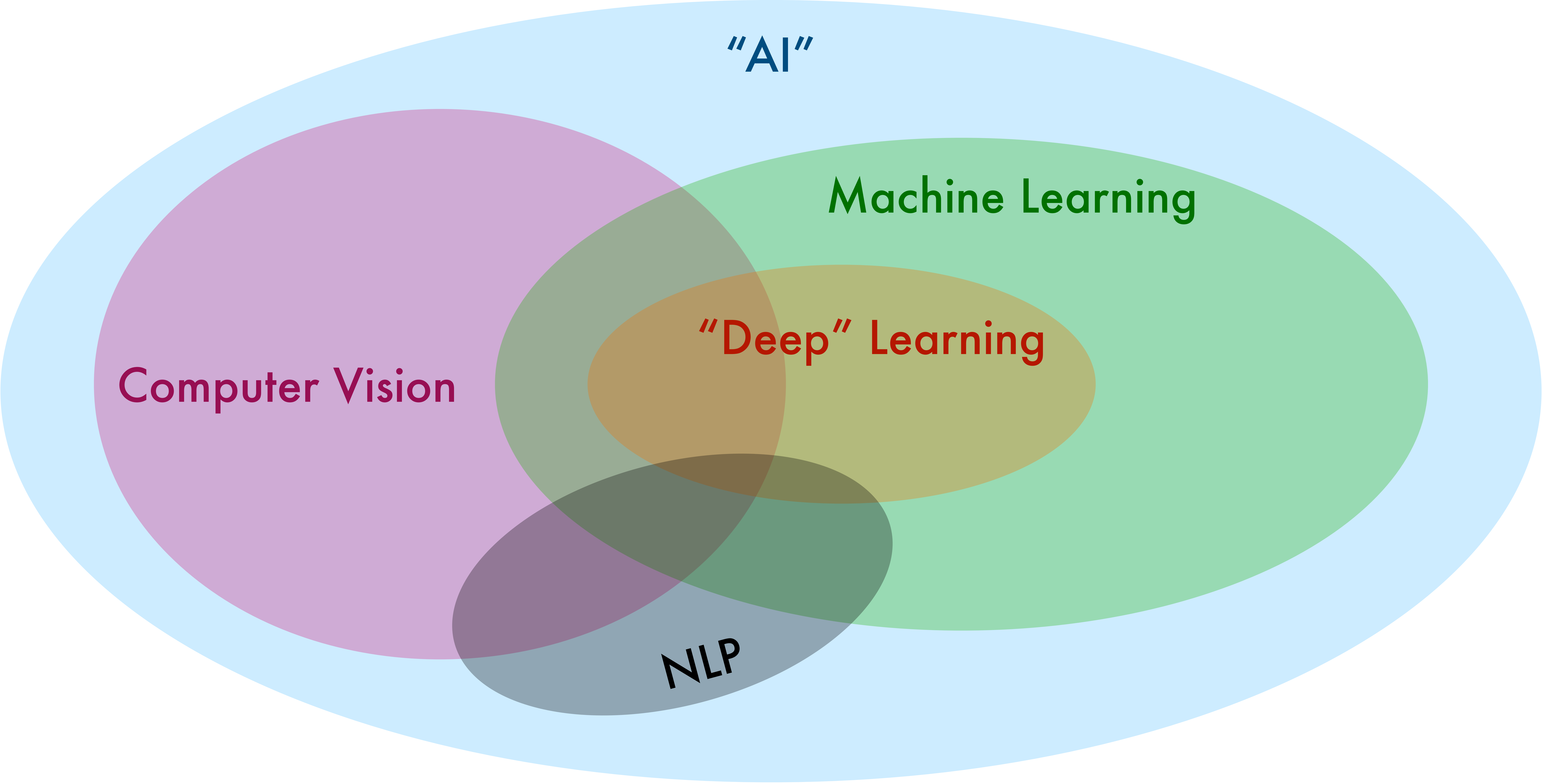
- Machine learning concepts

- Basics of supervised learning

- Introduction to neural networks

- Training neural networks
    - Loss
    - Gradient descent and variants
    - Learning rate
    - Backpropagation
    - Regularization

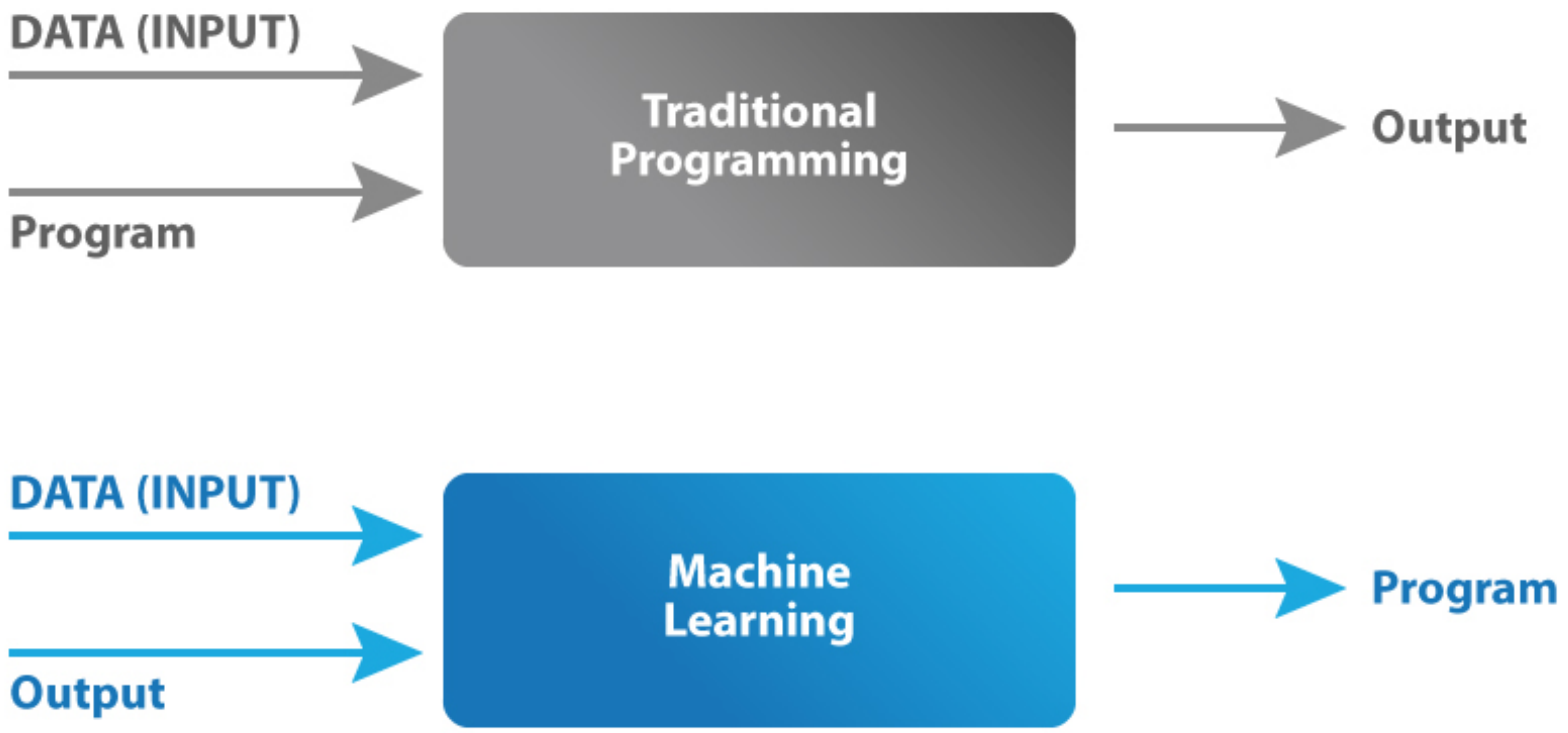
Remember:



Slide credit: Justin Johnson

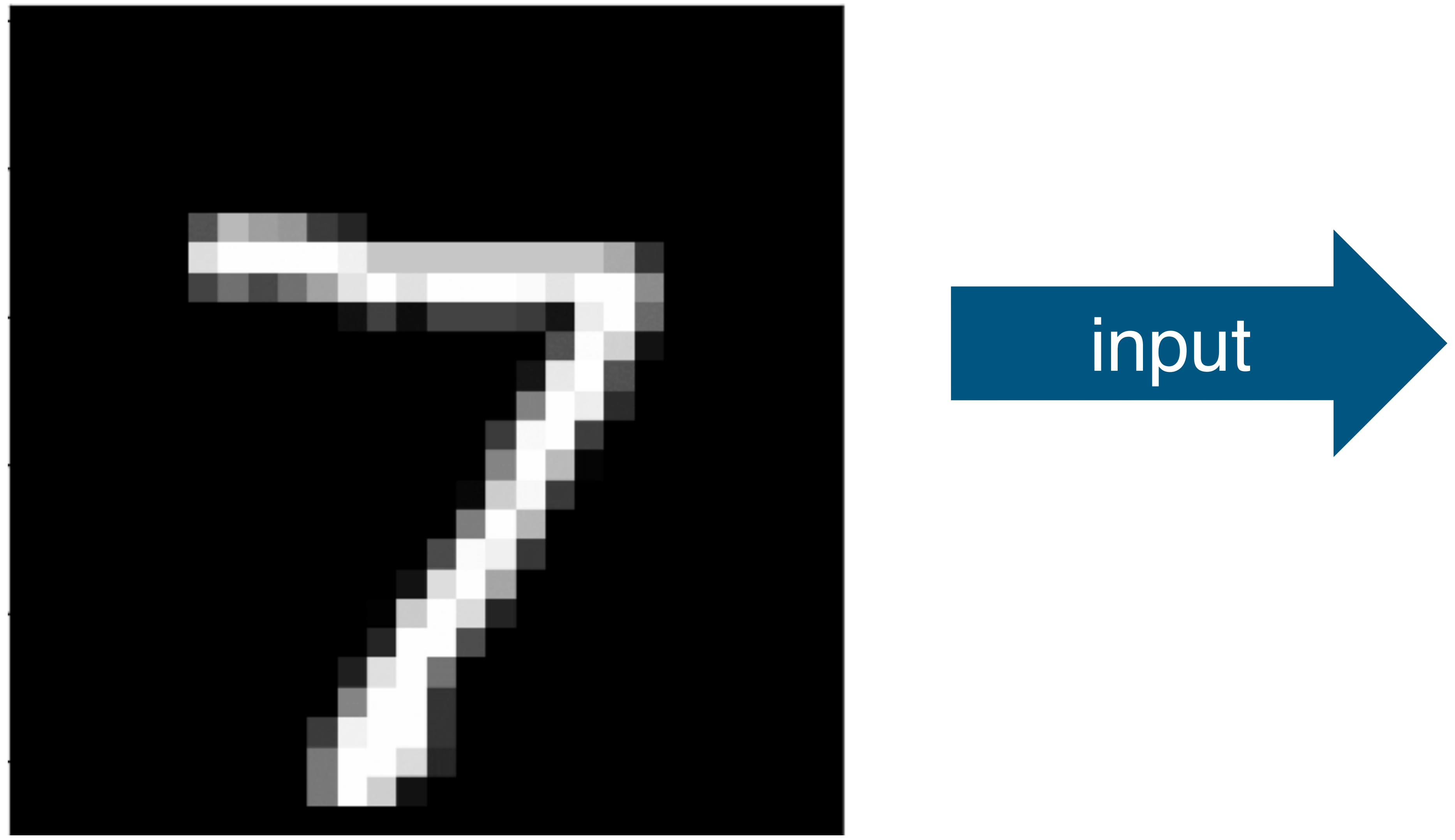
What is Machine Learning (ML)?

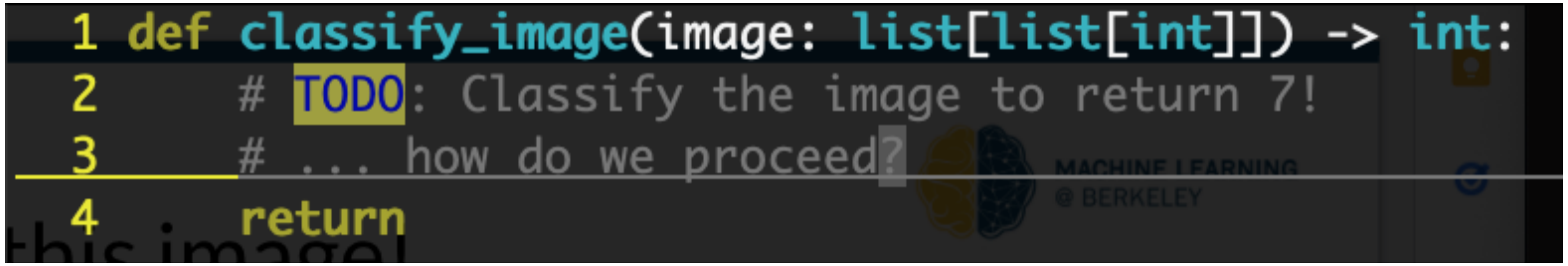
- ML is the paradigm of approximating a function from data
  - A function here is just a set of rules that takes in an input and spits out some output (like a label or a predicted value)



- Why ML instead of programming the functions ourselves?
  - Sometimes we can't possibly understand the patterns in our data, so it is extremely hard to come up with these rules!
  - ML is fundamentally the process of allowing our data to guide a function's creation

Slide credit: J. Austin, B. Liu, A. Jain





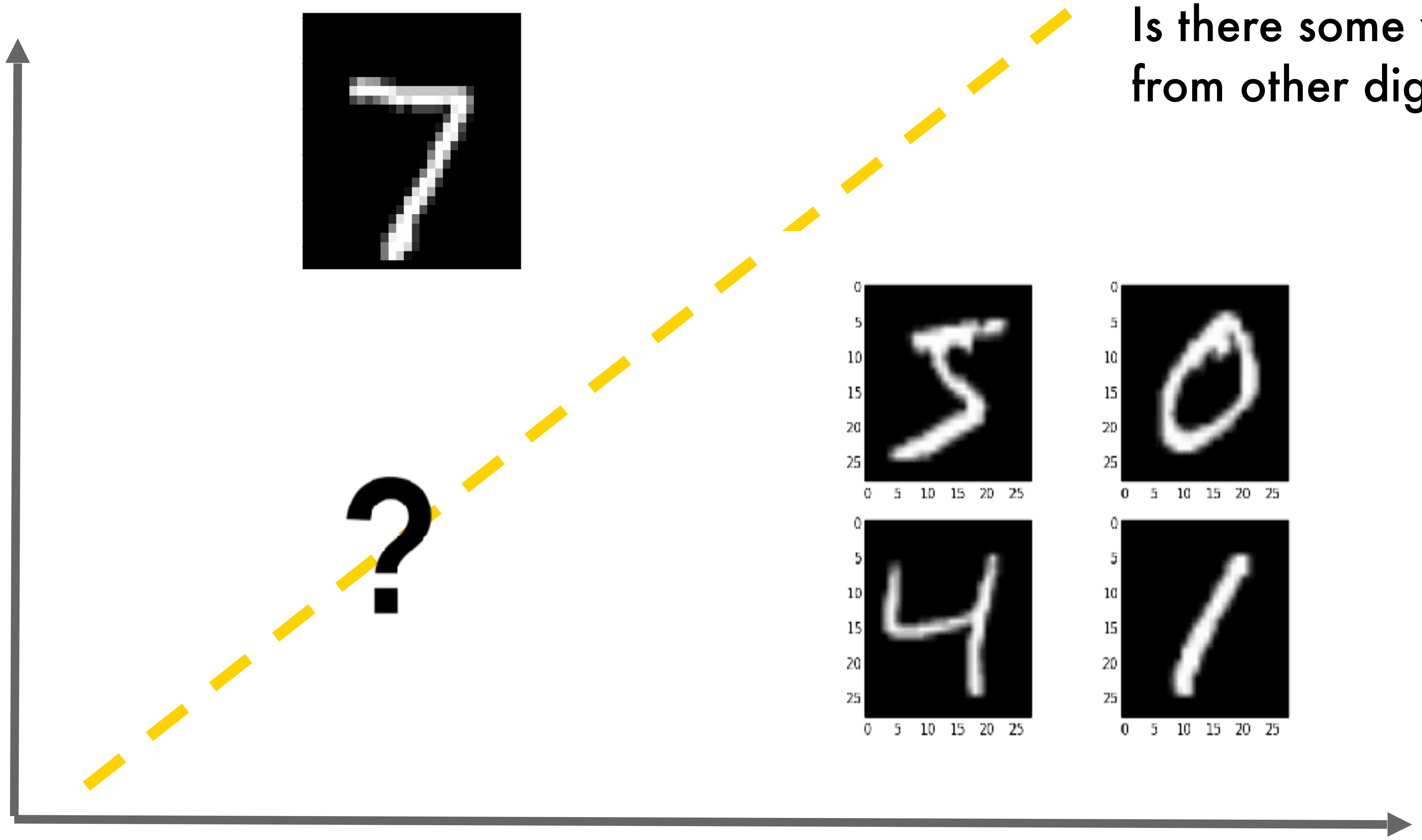
```
def classify_image(image: list[list[int]]) -> int:
    # TODO: Classify the image to return 7!
    # ... how do we proceed?
    return
```

Nope

```
for i in range(10, 30):
        if image[10][i] > 0.5:
                count += 1
if low_thresh < count < high_thresh:
        return 7
```

Challenge: write a function to classify digits?

Slide credit: J. Austin, B. Liu, A. Jain



Is there some way of separating 7′s from other digits?

Challenge: write a function to classify digits?

Slide credit: J. Austin, B. Liu, A. Jain

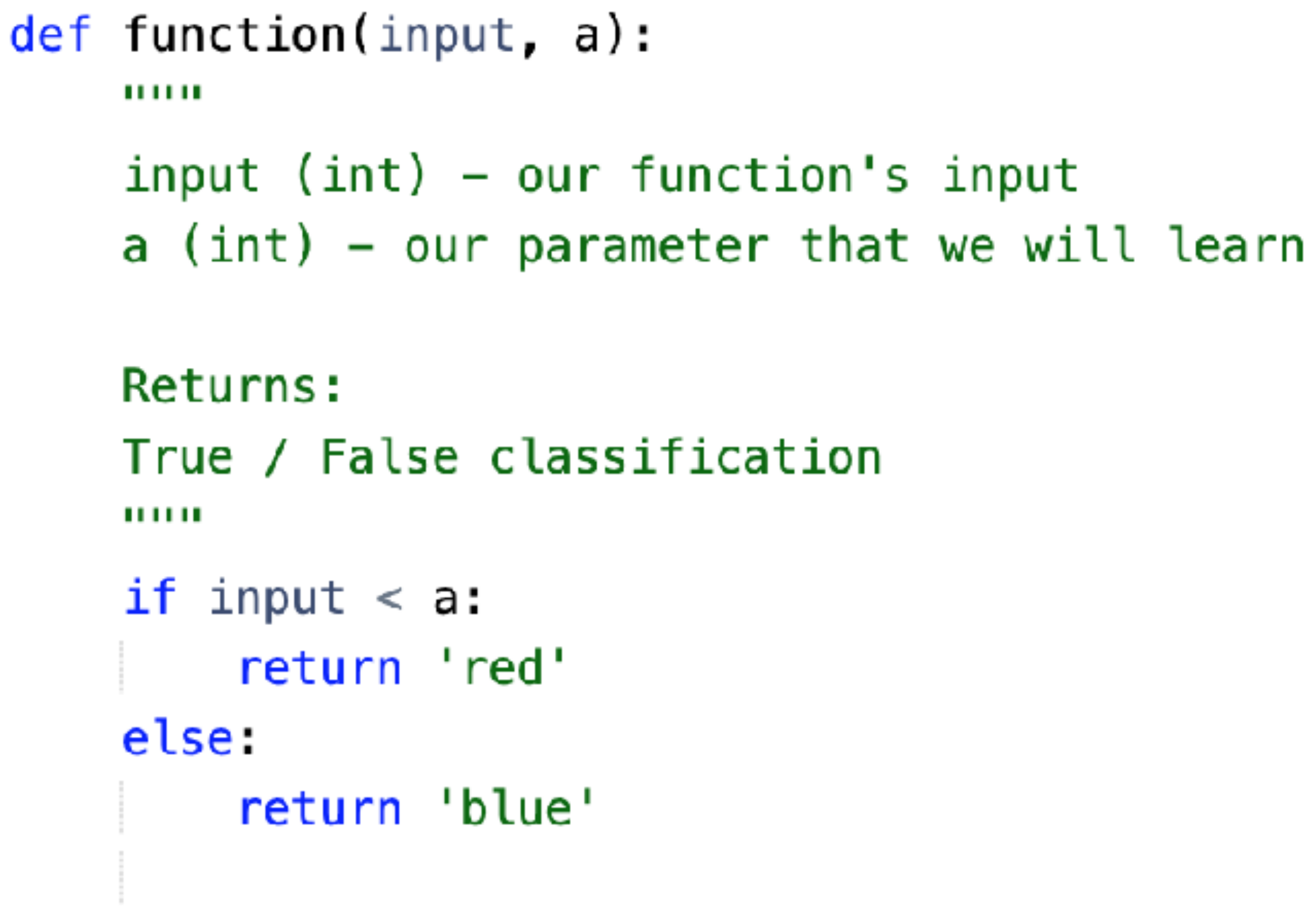
ML

- Think of it as template creation!
  - When we usually define a function by hand, we have to specify EVERYTHING
  - With ML, we are going to **define a function (with math)**, but leave out a few **free parameters** that will be **learned from the data**: these will dictate the exact behavior of the function

- Example:
  - We will define our function to have the form:

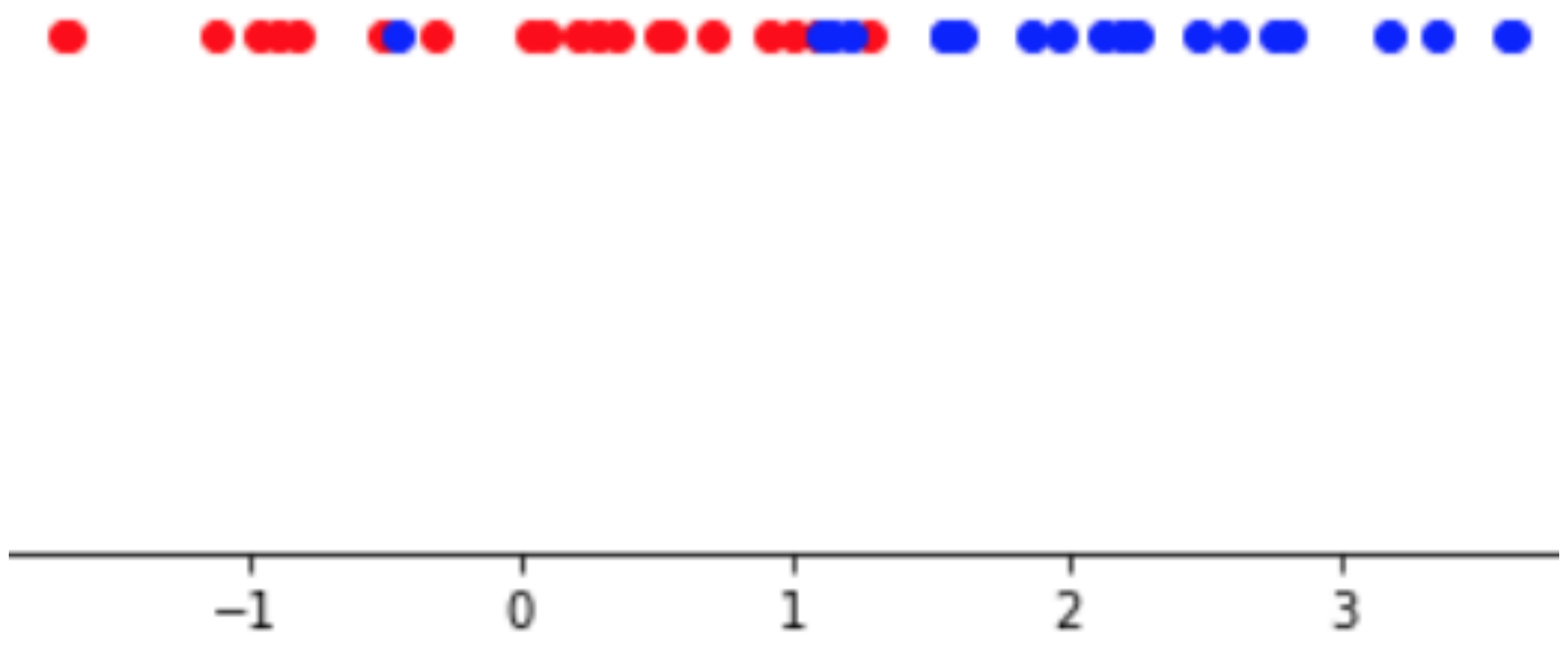
    `if (input < a ) -> output1,  else -> output2,`

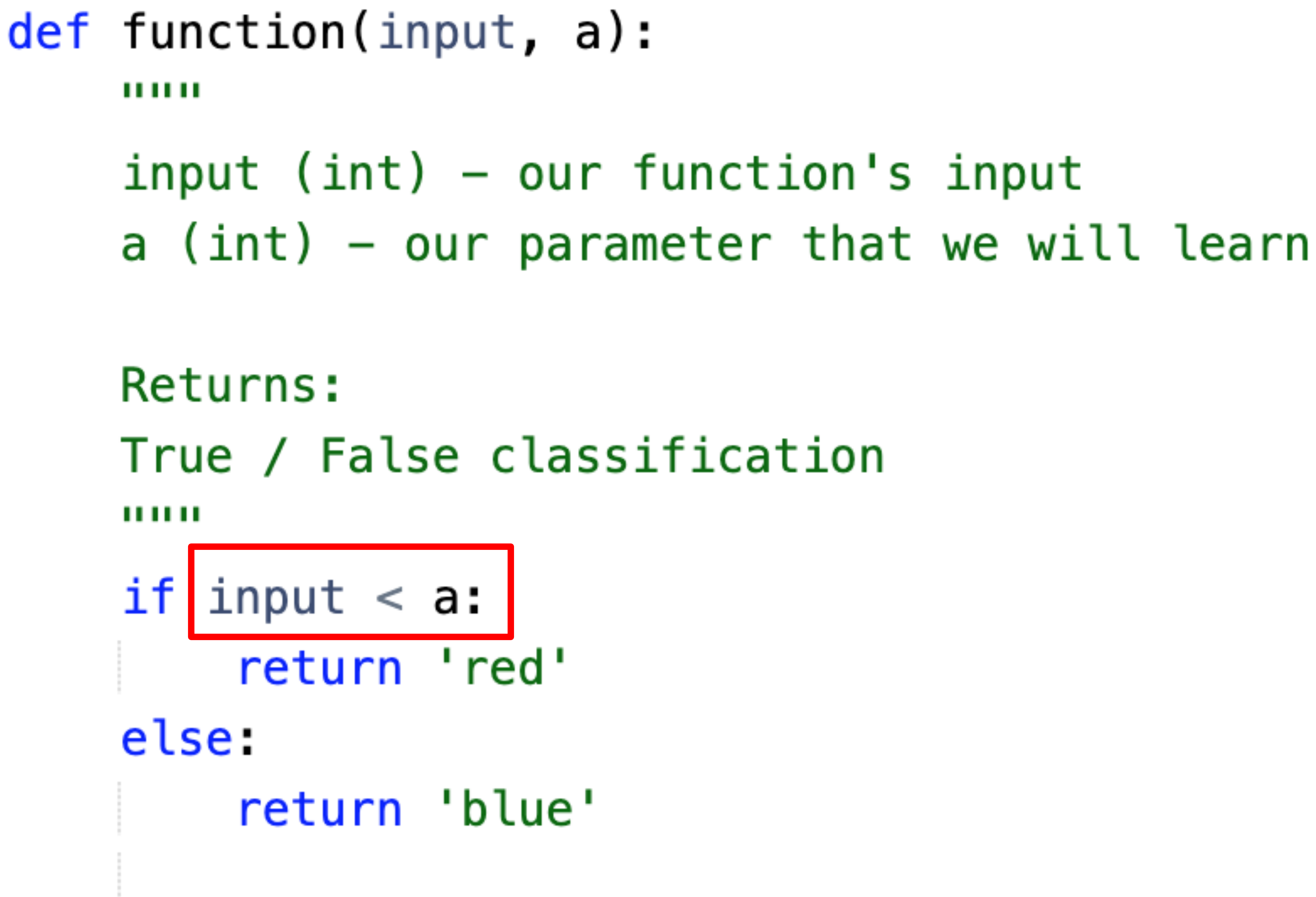
    and learn the best value of **a** from our data
  - Here '**a**' is the free parameter that specifies the exact behavior of our example function



```python
def function(input, a):
    """
    input (int) - our function's input
    a (int) - our parameter that we will learn

    Returns:
    True / False classification
    """
    if input < a:
        return 'red'
    else:
        return 'blue'
```

Slide credit: J. Austin, B. Liu, A. Jain



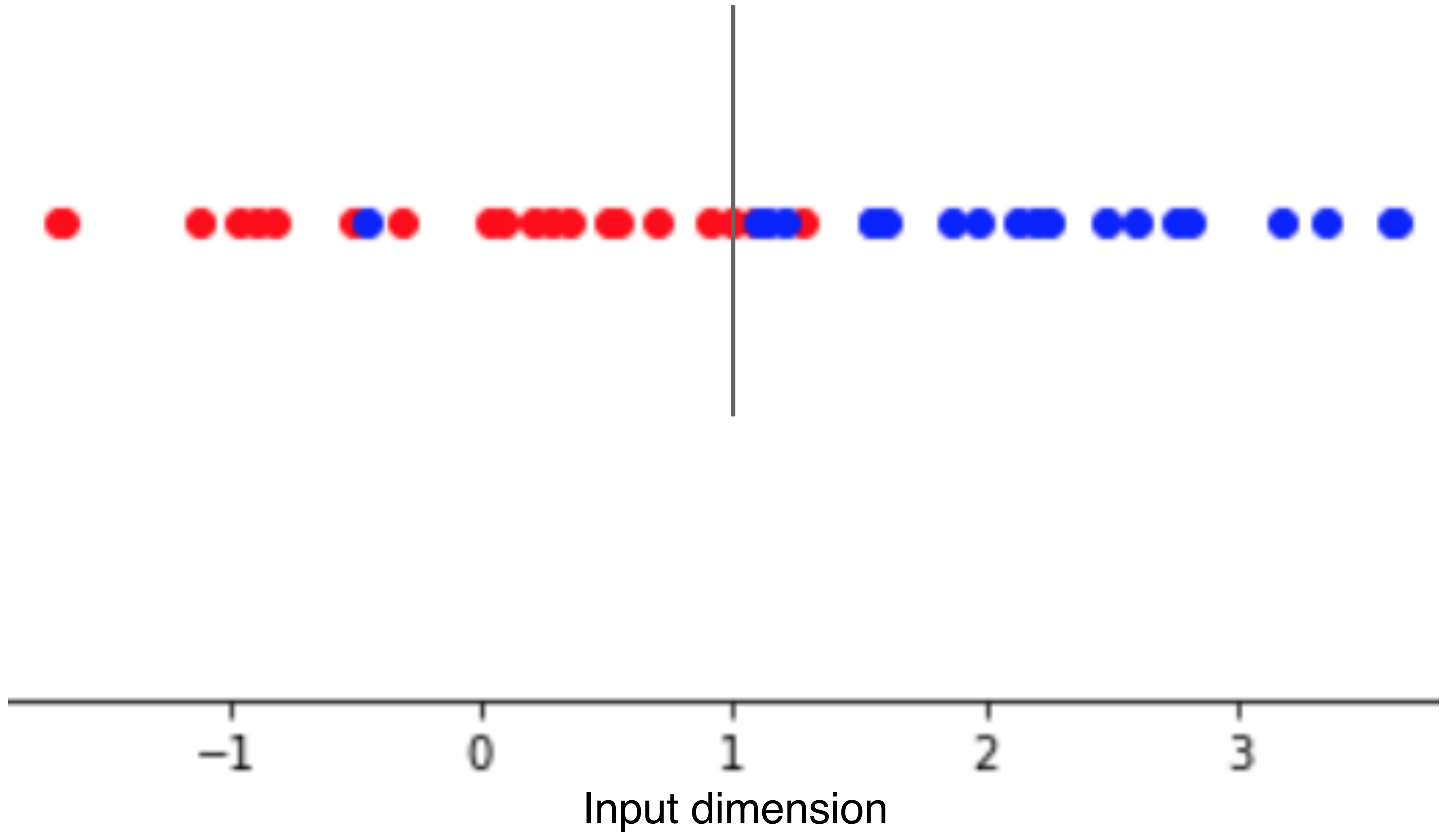


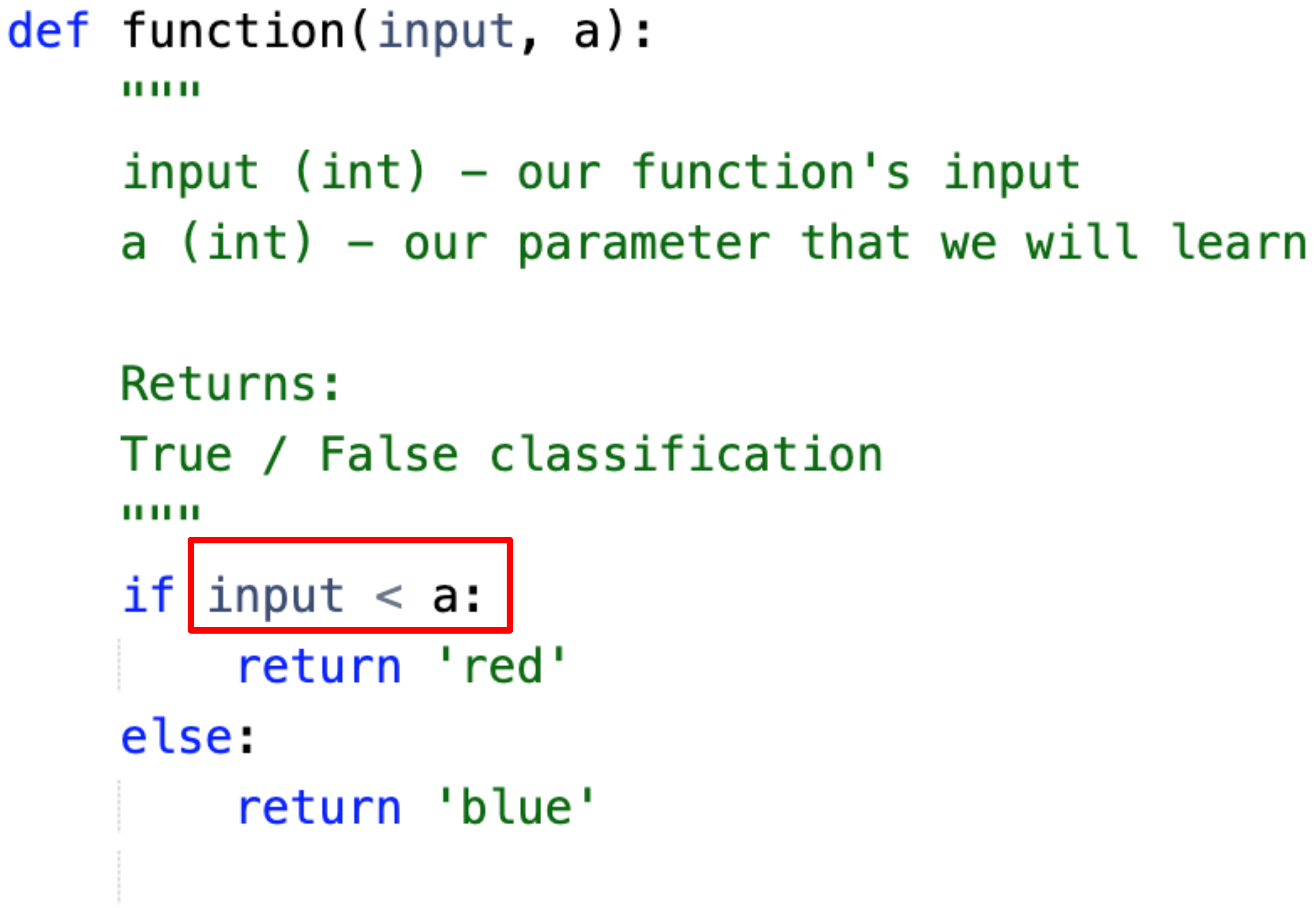
```
def function(input, a):
    """
    input (int) - our function's input
    a (int) - our parameter that we will learn

    Returns:
    True / False classification
    """
    if input < a:
        return 'red'
    else:
        return 'blue'
```

Note: This function is just a *hypothesized* function that we hope will work well based on what the data looks like

What would a good value for ‘a’ be?

Slide credit: J. Austin, B. Liu, A. Jain



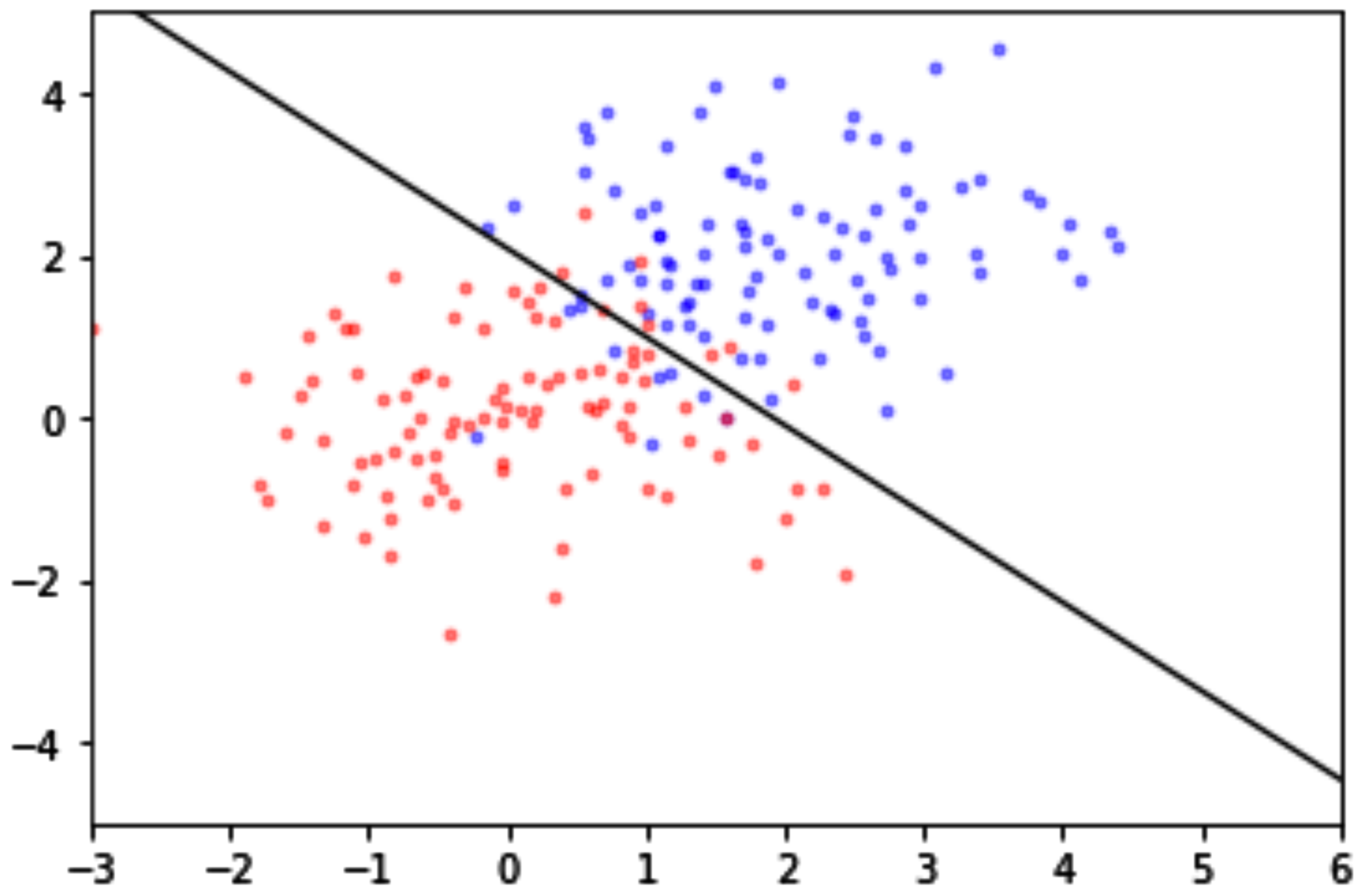


```python
def function(input, a):
    """
    input (int) - our function's input
    a (int) - our parameter that we will learn

    Returns:
    True / False classification
    """
    if input < a:
        return 'red'
    else:
        return 'blue'
```

What would a good value for ‘a’ be? Probably a = 1

Slide credit: J. Austin, B. Liu, A. Jain

2D Example



Previously, we had a single point, above which things were blue, and red otherwise. However, this strategy doesn't really work in 2D...

Now, we might try and hypothesize that a **1D line** separates the data instead, above which all points are blue but red below.

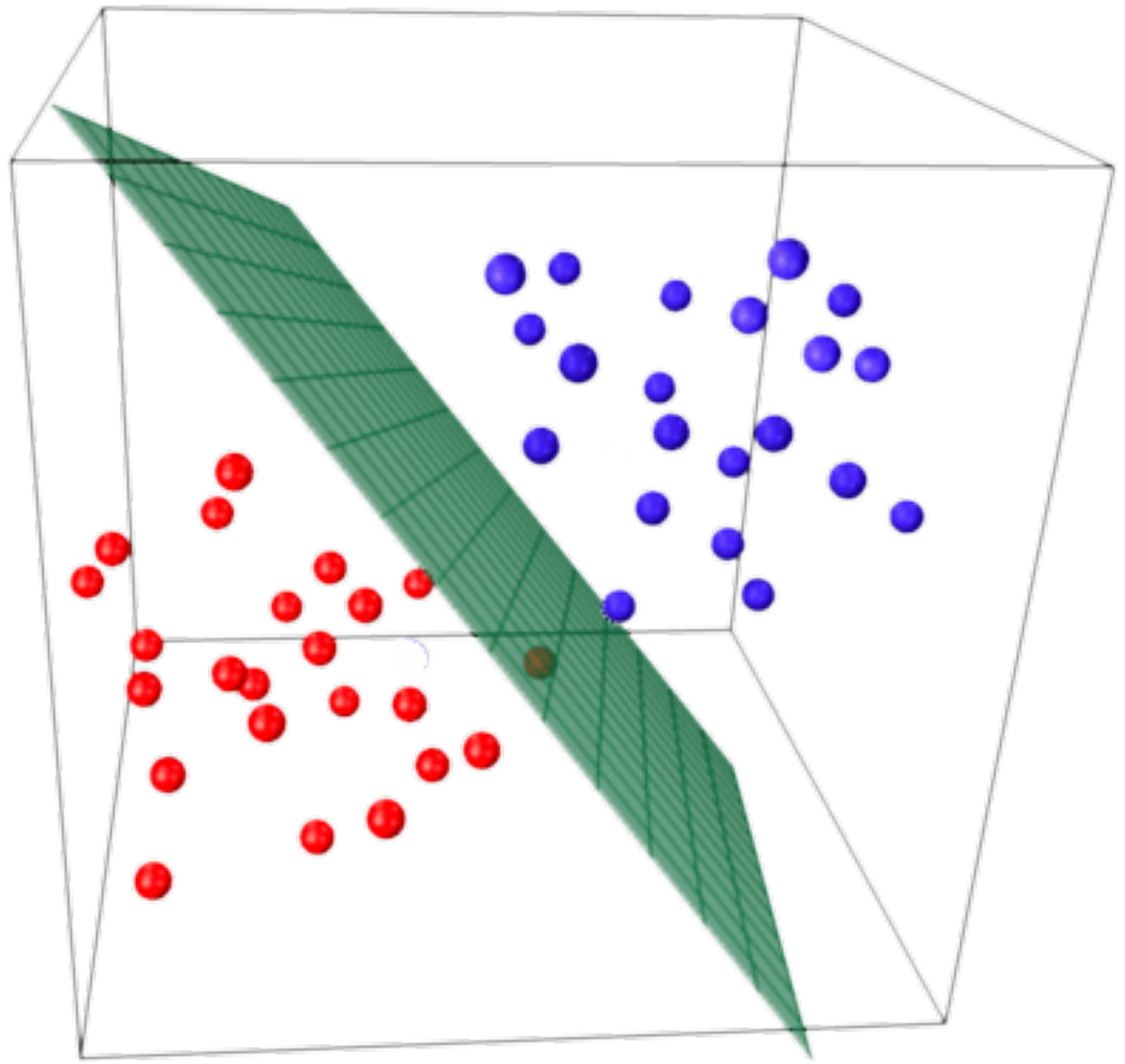
This is our FUNCTION that we are hypothesizing exists... a 1D line in the form

y = mx+b

In this case, our parameters are **m** (the slope) and **b** (the intercept/offset)

Slide credit: J. Austin, B. Liu, A. Jain

3D and so on...



This idea continues on well beyond 2D as well. Here, our data is in 3D and we hypothesize that a 2D plane can separate the data, above which points are marked blue, below which they are marked red... and this again is our function definition.

This can further continue on forever into higher dimensions!

The challenge is that we can't immediately visualize higher dimensional data, so it will be difficult to say if the data will nicely separate along some linear boundary like this or not...

Slide credit: J. Austin, B. Liu, A. Jain

ML

- The art of ML is the following:
  - **Function:** What form our function takes → this can be referred to as a **model class**
  - **Parameters:** What specific parts of this function we are allowed to learn → these are our **parameters**
  - **Optimization:** How we learn these parameters to approximate their “best” possible values

- Every ML algorithm you will ever learn follows this pattern
  - Describe the generic form of a function with free parameters
  - Use the data to decide what free parameters will work best

Slide credit: J. Austin, B. Liu, A. Jain

Vocabulary

- **Function / Model**
  - These terms are used interchangeably
  - These refer to the function template (the "model class") we have chosen for our problem
- **Parameters / Weights (and Biases)**
  - Another way to denote the parameters in ML models that are learned from data
- **Hyperparameters**
  - This is some non-learnable parameter (like model size, model type, details about training procedure, etc) that further specifies our overall learnable function
  - We need to manually choose these ourselves before we start learning the learnable parameters
- **Loss** Function / **alternative terms?**

Slide credit: J. Austin, B. Liu, A. Jain

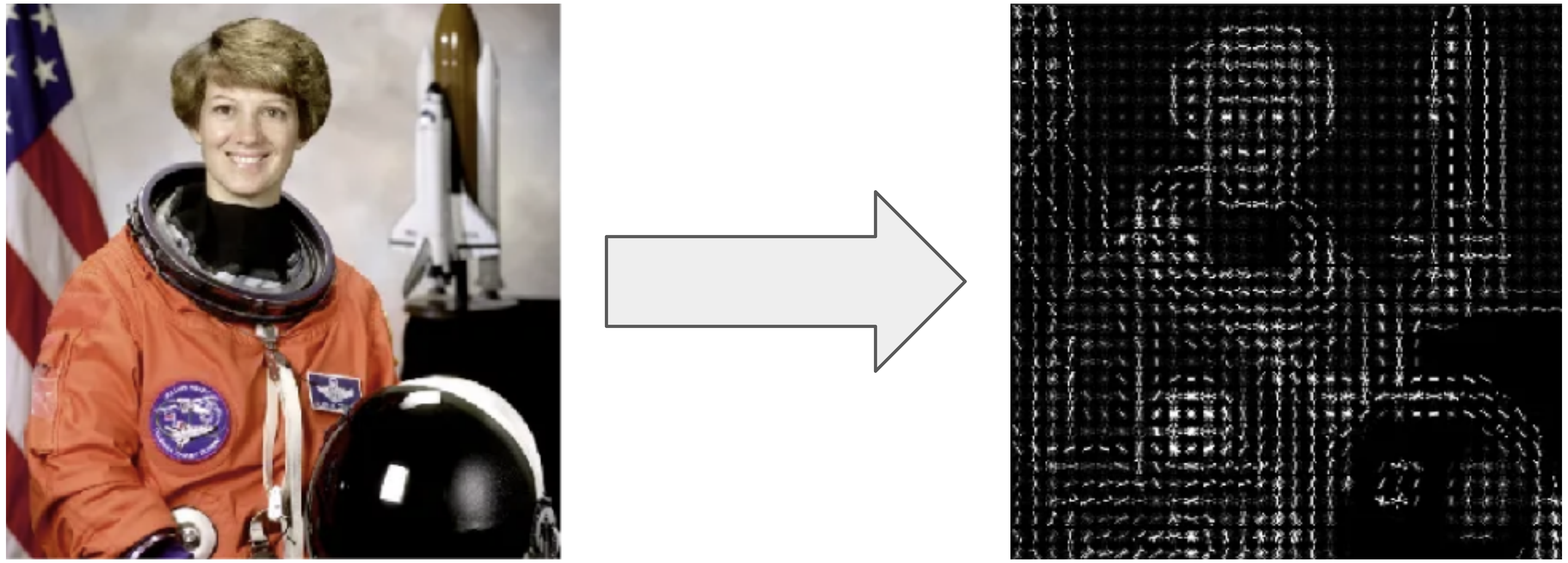
Vocabulary

- **Function / Model**
  - These terms are used interchangeably
  - These refer to the function template (the "model class") we have chosen for our problem
- **Parameters / Weights (and Biases)**
  - Another way to denote the parameters in ML models that are learned from data
- **Hyperparameters**
  - This is some non-learnable parameter (like model size, model type, details about training procedure, etc) that further specifies our overall learnable function
  - We need to manually choose these ourselves before we start learning the learnable parameters
- **Loss** Function / **Cost** Function / **Risk** Function / **Objective / Error…**

Slide credit: J. Austin, B. Liu, A. Jain

Vocabulary: “Feature”

- This can refer to bits of our data (either the inputs themselves or some representation of them) that we feed as input to a model
    - e.g., for a house, you might input quantities like its “number of bedrooms”, “number of floors”, “area in square feet”, “cost of construction” etc. into a model that is trying to predict its price
    - e.g., for an image input, you squish its pixel values into a vector OR extract things like corners, edges, shapes from it – these are both different “features” of the same image that can be fed into a model!



Slide credit: J. Austin, B. Liu, A. Jain

ML Pipeline

1. Define the **problem**
2. Prepare the **data**
3. Define the **model** + **loss** function
4. **Minimize** the loss function (=train the model)
5. DONE(!)

What is the task? How do inputs/outputs look like? What is the success metric? …

DO NOT UNDERESTIMATE DATA PREPARATION !!!

(Data = text, image, labels => numbers)

More on these later.

Data prep**?**

Slide credit: J. Austin, B. Liu, A. Jain

ML Publication Venues

ICML: International Conference on Machine Learning [1980

NeurIPS: Neural Information Processing Systems [1987

...

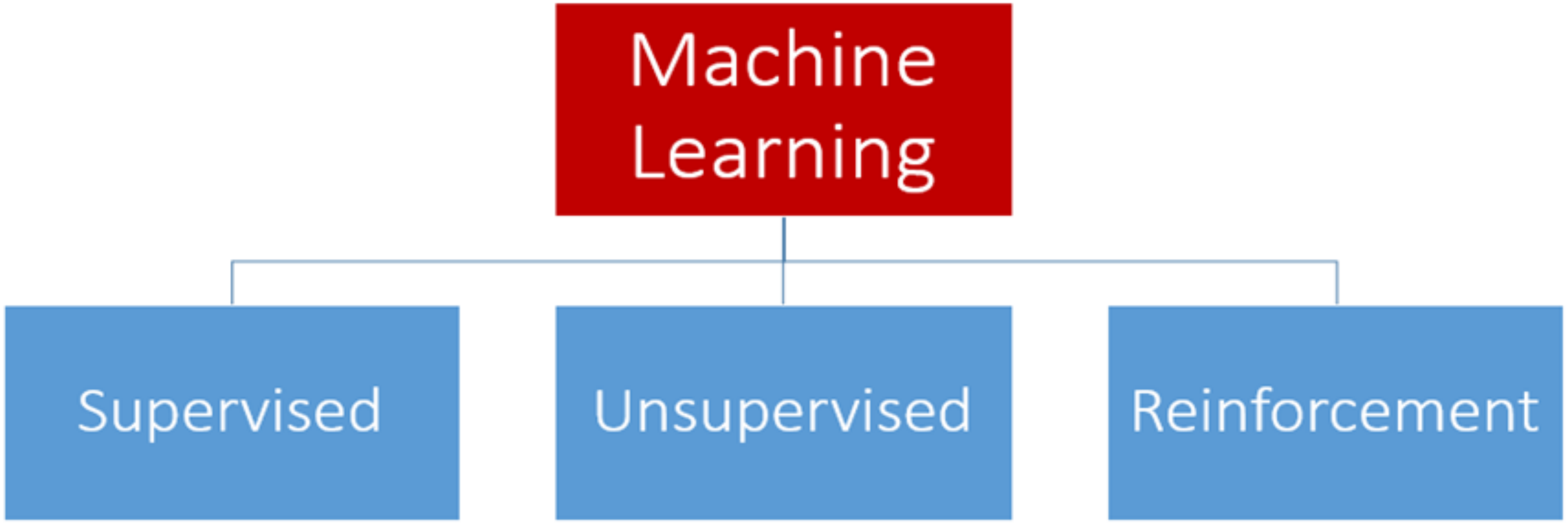
ICLR: International Conference on Learning Representations [2013

AAAI: Conference on Artificial Intelligence [1980,

JMLR: Journal of Machine Learning Research [2000,

TMLR: Transactions on Machine Learning Research [2022 (new)

...



Machine Learning

Supervised

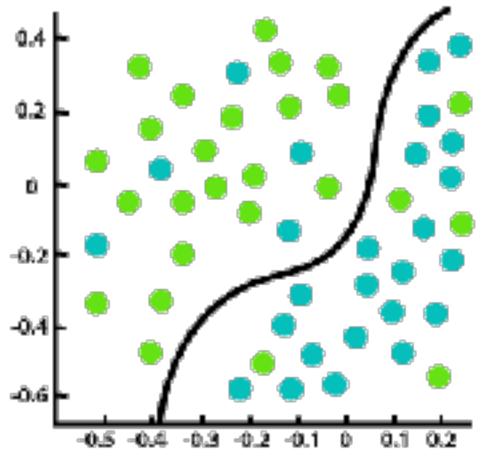
Unsupervised

Reinforcement

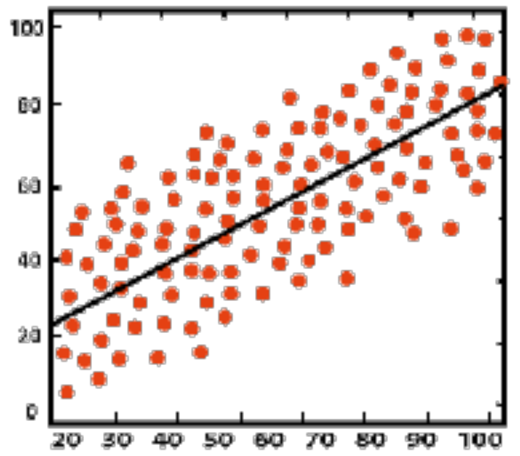
Task driven (Regression / Classification)

Data driven (Clustering)

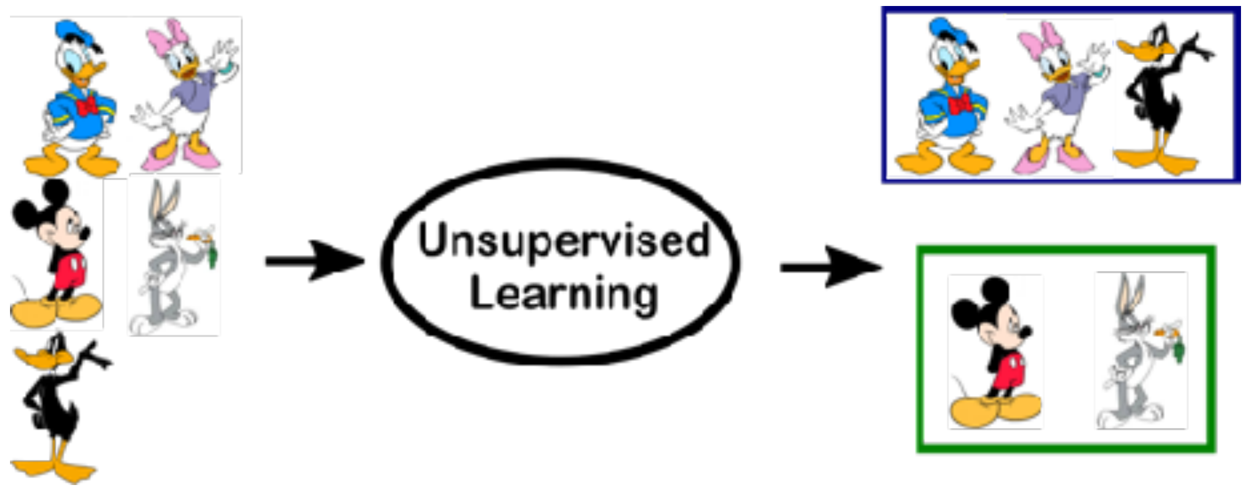
Algorithm learns to react to an environment

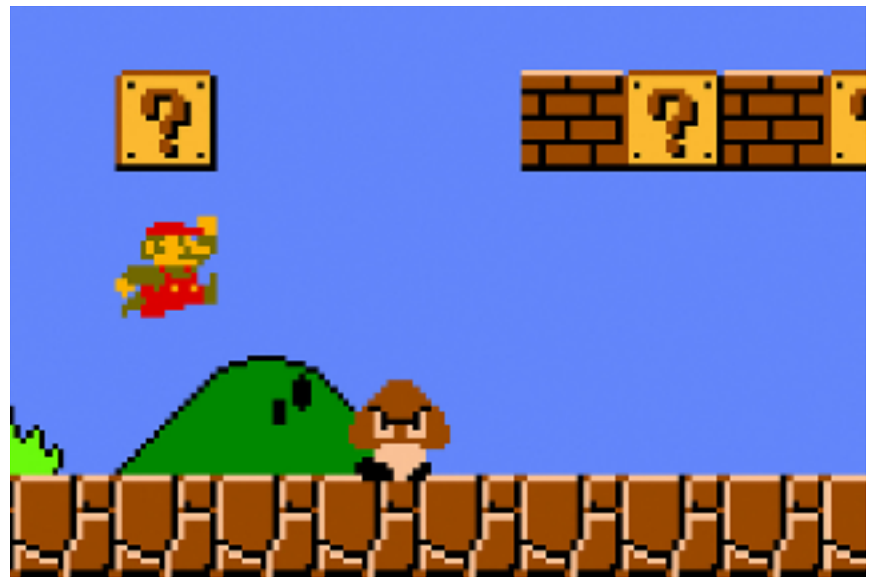


Classification



Regression





With labels.

No labels.

Rewards.

Weakly-supervised? (Labeled for another task)

Semi-supervised? (Dataset partially labeled)

Self-supervised? (Unlabeled, data itself provides supervisory signal)

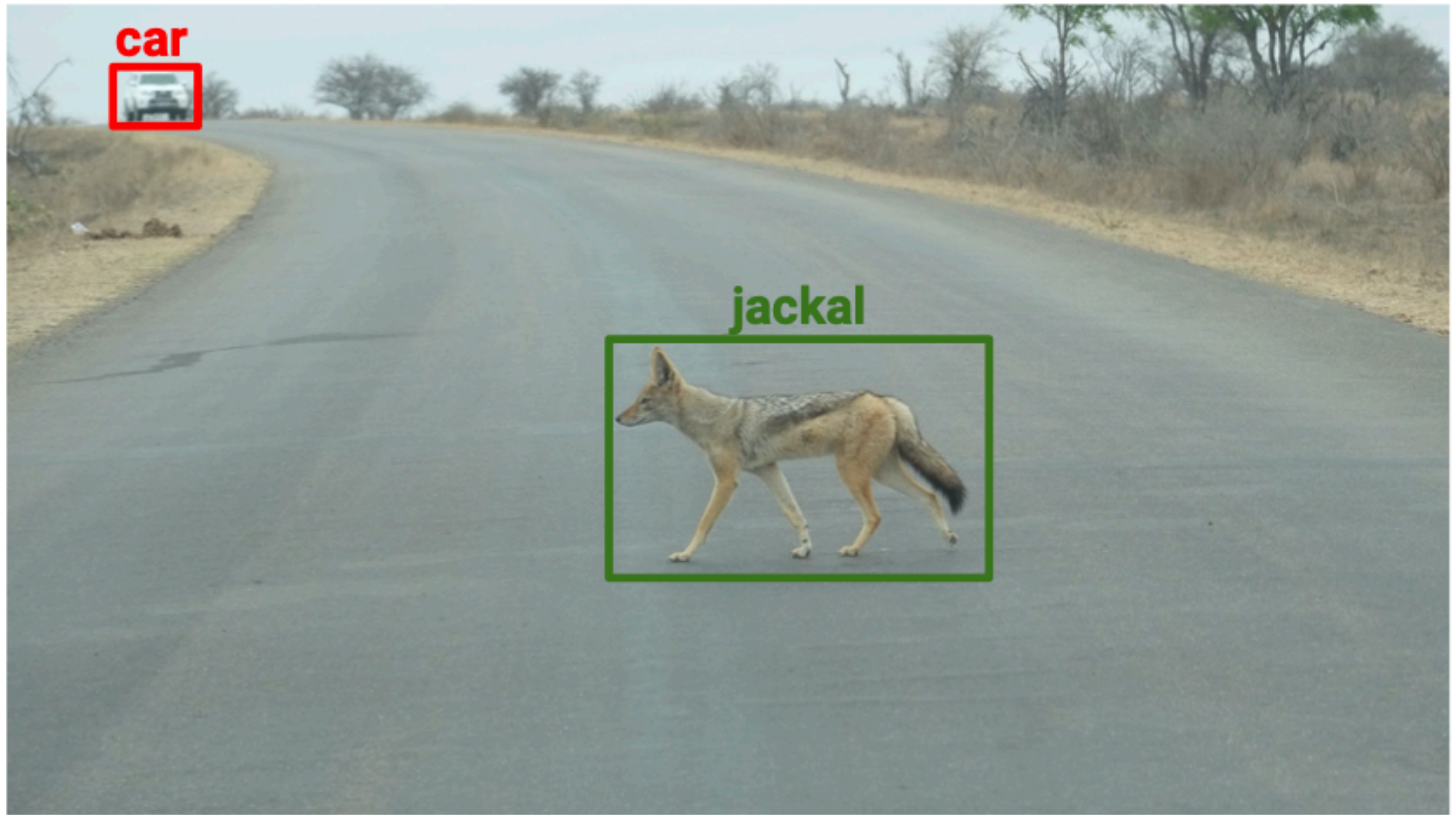
Slide credit: J. Austin, B. Liu, A. Jain

Weakly-supervised? (Labeled for another task)

Example: Object detection (i.e., bounding box and class label prediction)

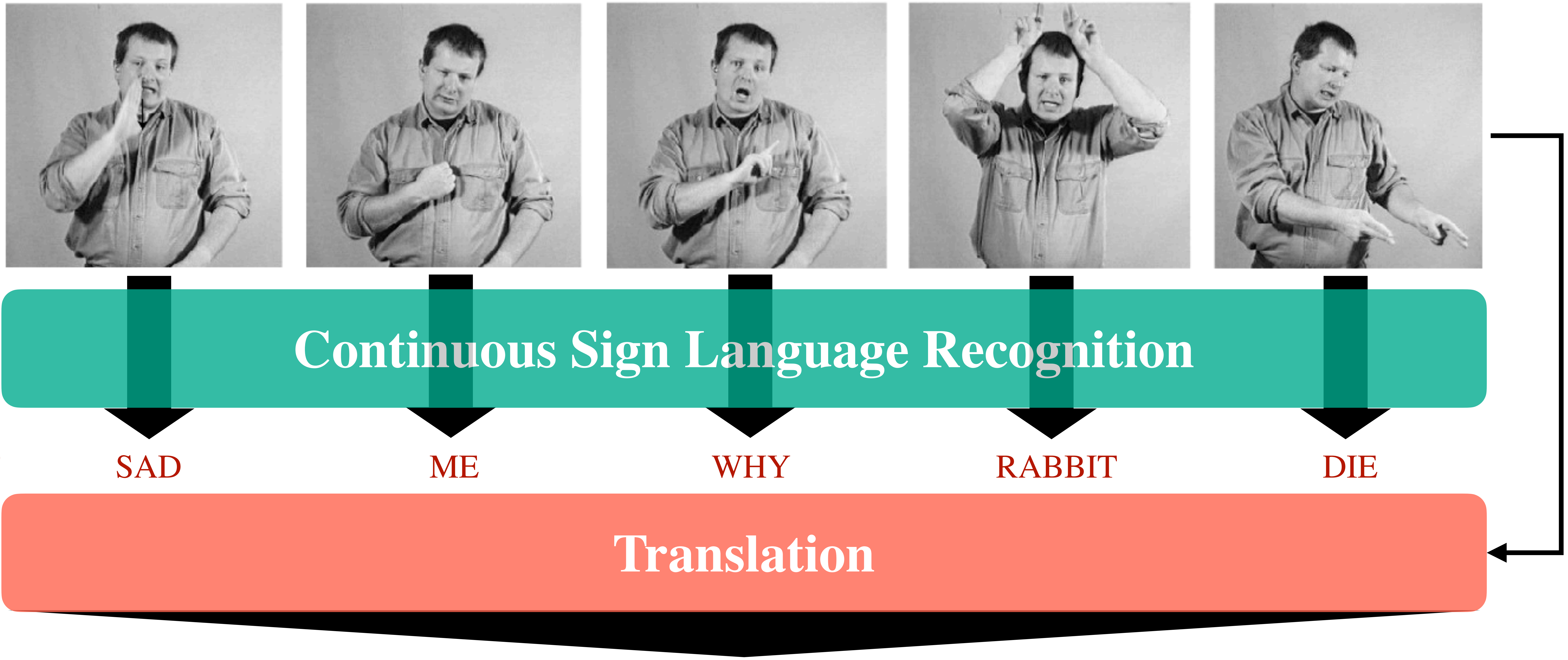
Category labels: car, jackal

No box labels



Weakly-supervised? (Labeled for another task)

Example: Sign language recognition (i.e., video classification)



Continuous Sign Language Recognition

SAD ME WHY RABBIT DIE

Translation

No sign categories

Sign categories (Glosses)

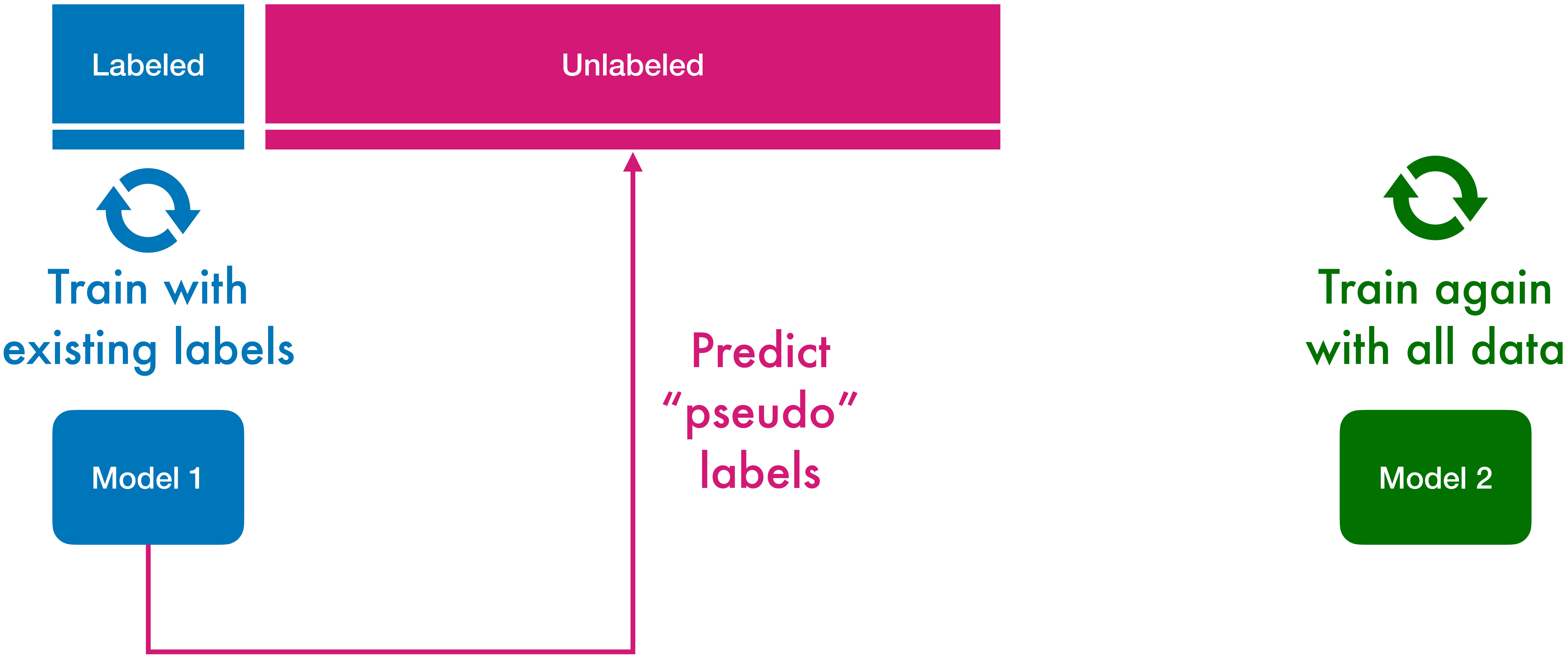
Spoken language sentences

I am sad because the rabbit died.

Available labels: sentence translations

Semi-supervised? (Dataset partially labeled)

Very realistic scenario in today’s research



Noise in supervision

Self-supervised? (Unlabeled, data itself provides supervisory signal)

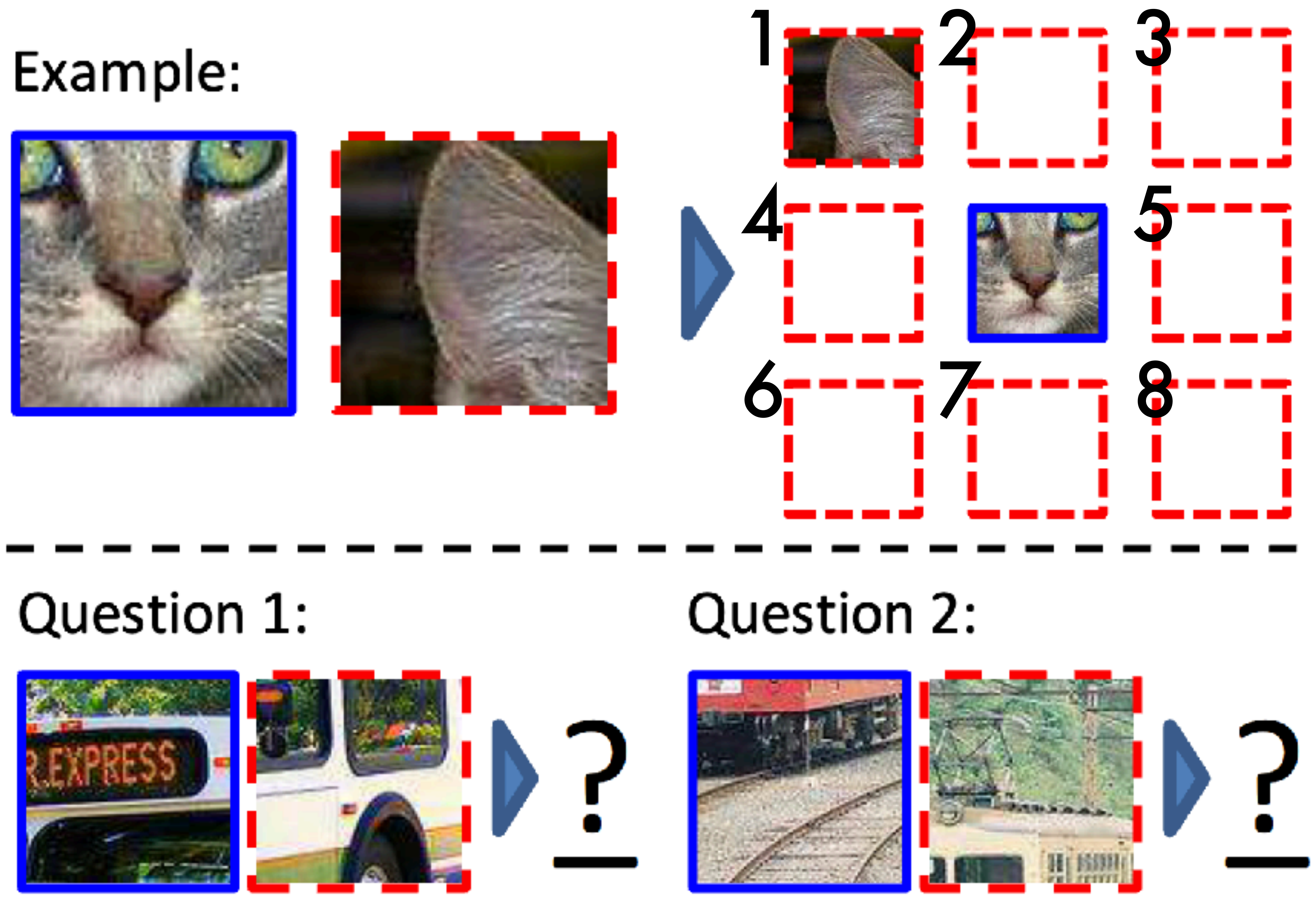


Figure 1. Our task for learning patch representations involves randomly sampling a patch (blue) and then one of eight possible neighbors (red). Can you guess the spatial configuration for the two pairs of patches? Note that the task is much easier once you have recognized the object!

[Doersch, Gupta, Efros, "**Unsupervised** Visual Representation Learning by Context Prediction", ICCV 2015]

What is the problem formulation?

8-way classification given two images

Agenda:

- Machine learning concepts
- Basics of supervised learning
- Introduction to neural networks
- Training neural networks
    - Loss
    - Gradient descent and variants
    - Learning rate
    - Backpropagation
    - Regularization

Basics of supervised learning

- n training data pairs $(x_1, y_1), ..., (x_n, y_n) \in \mathcal{X} \times \mathcal{Y}$
- Learn a predictor/decision function $\hat{f} : \mathcal{X} \to \mathcal{A}$
- By minimizing $\sum_{i=1}^{n} l(f(x_i), y_i)$

Basics of supervised learning

- n training data pairs

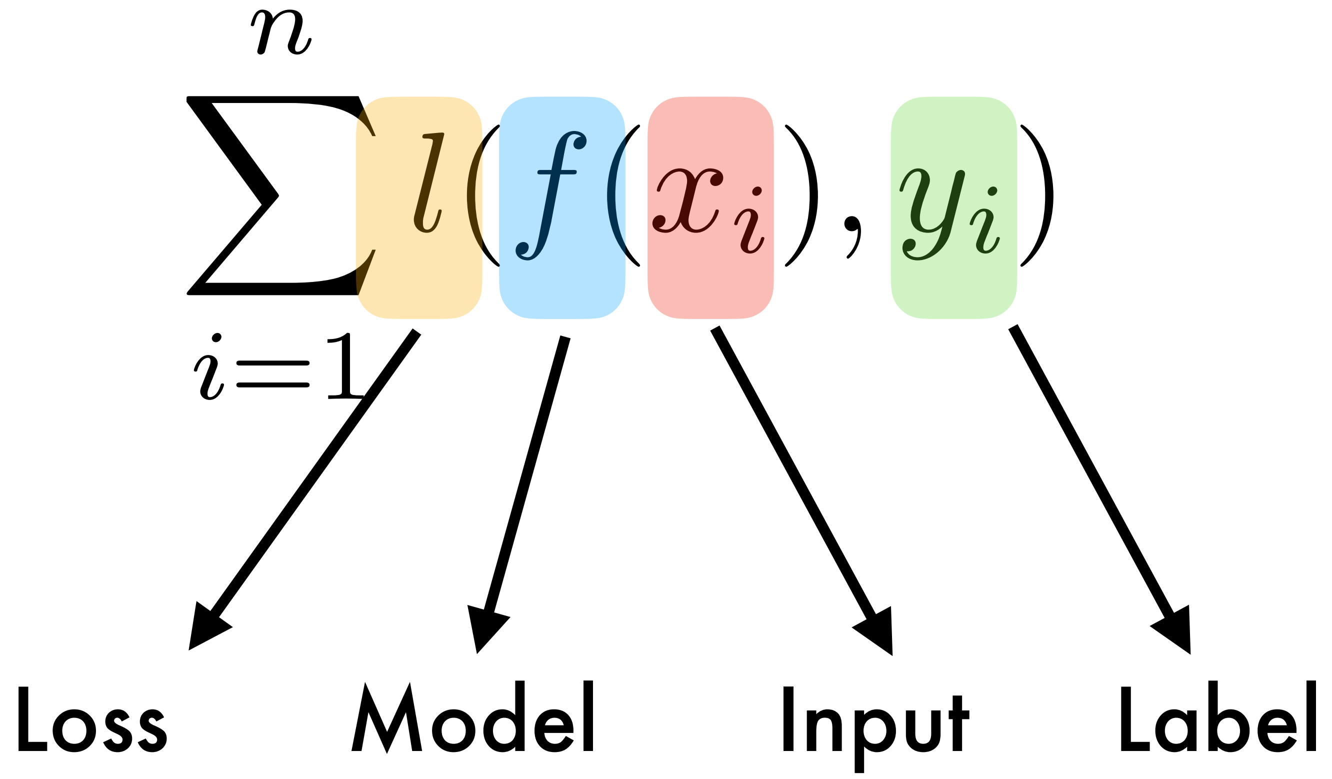
$$(x_1, y_1), \ldots, (x_n, y_n) \in \mathcal{X} \times \mathcal{Y}$$

- Learn a predictor/decision function

$$\hat{f} : \mathcal{X} \to \mathcal{A}$$

- By minimizing

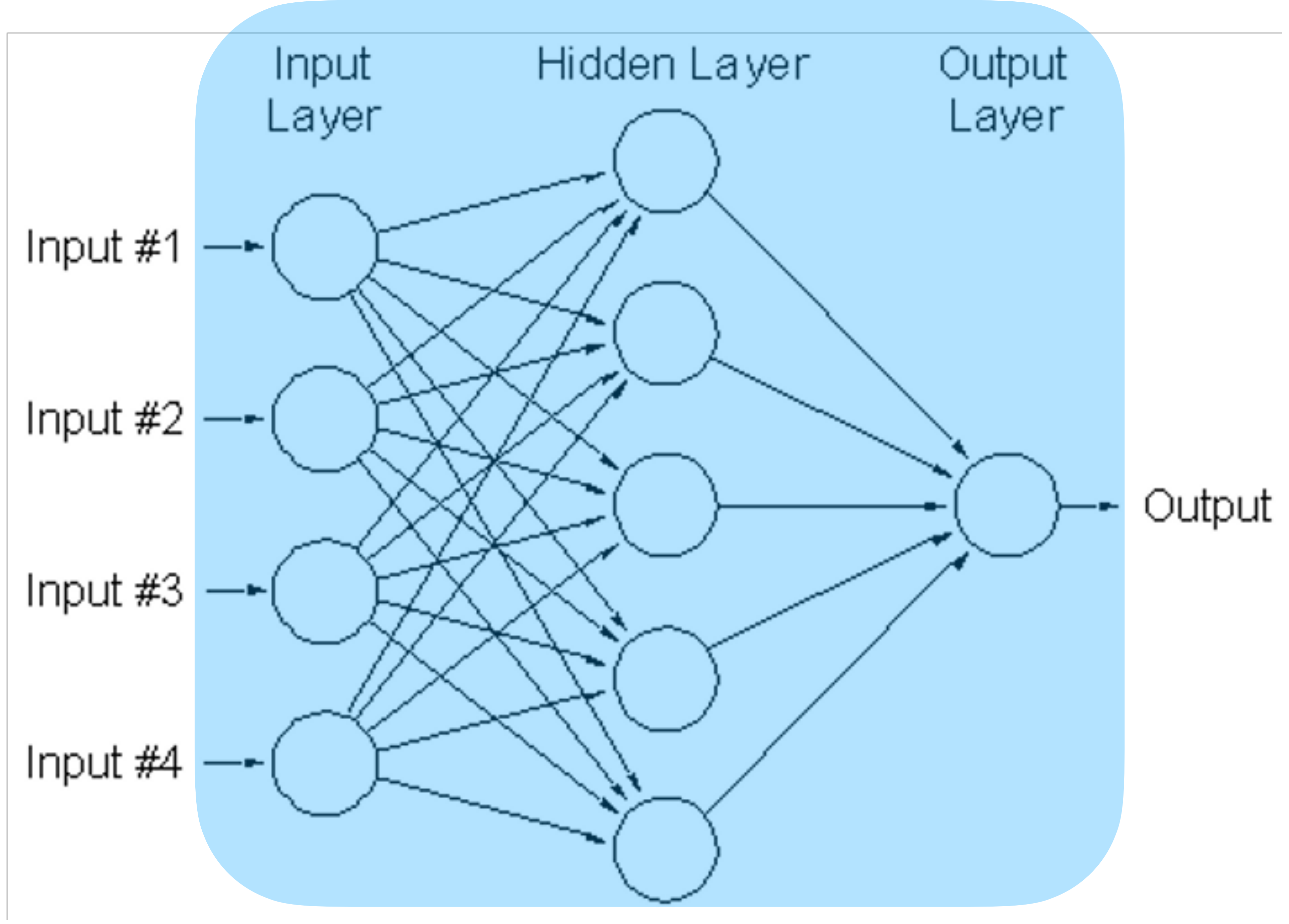
$$\sum_{i=1}^{n} l(f(x_i), y_i)$$



Deep learning

$$\sum_{i=1}^{n} l(f(x_i), y_i)$$

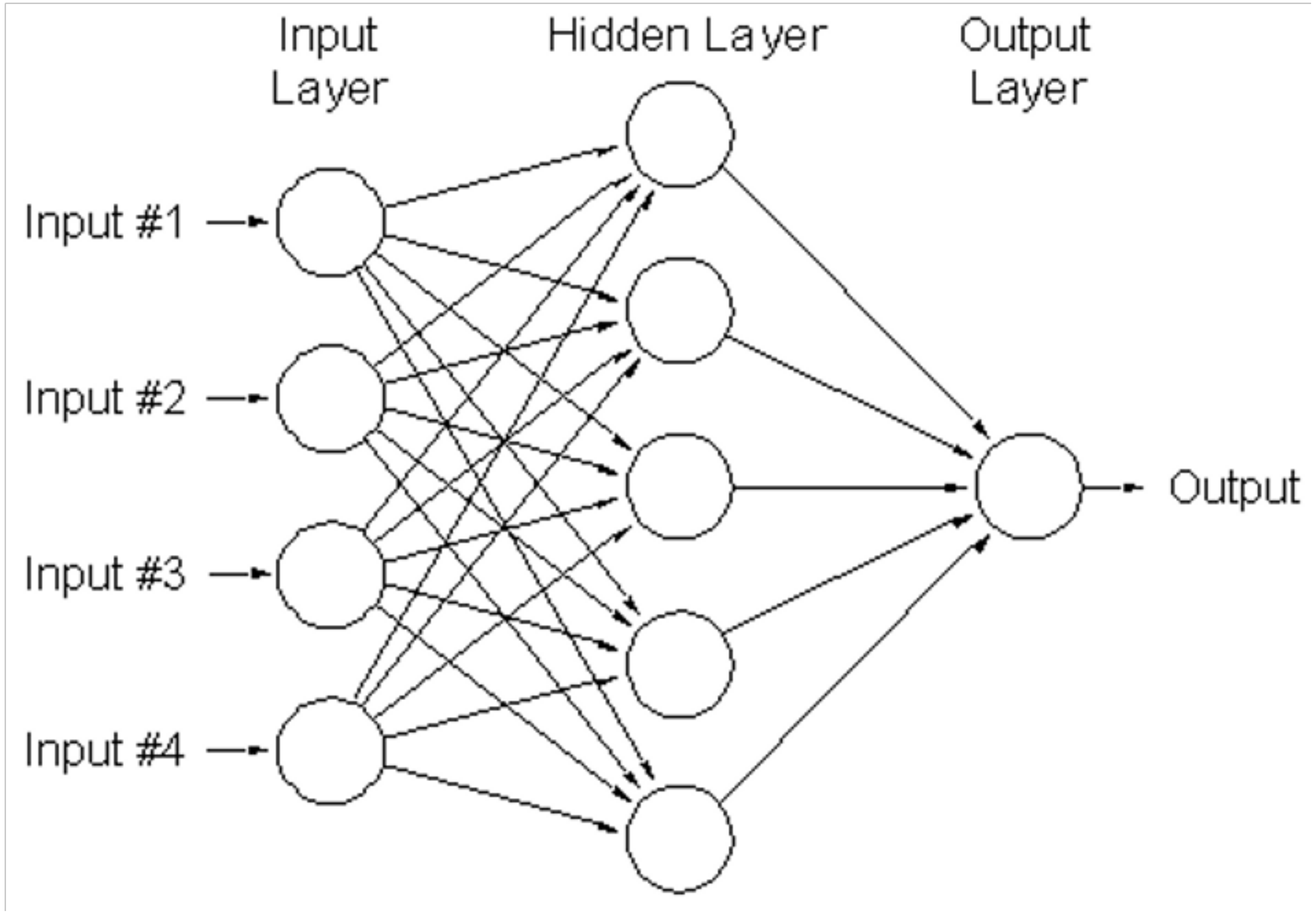
Loss Model Input Label



Deep learning:
Model = neural network

What is a "deep" neural network?

Stacking more than one **layer**



W_1
(Layer 1)

W_2
(Layer 2)

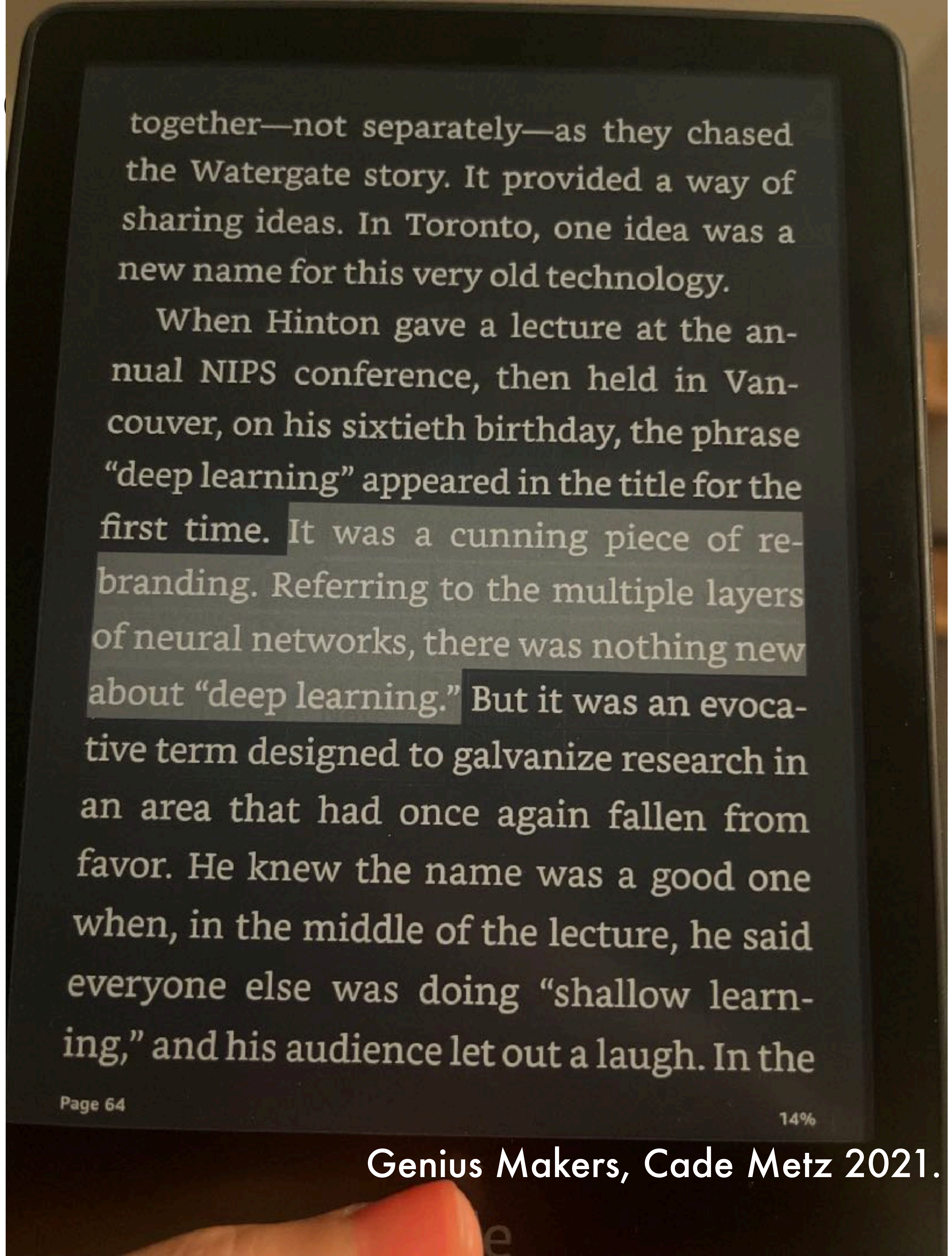
Disclaimer: Terminology

- ~~Deep learning~~
    - Neural networks?
    - Artificial neural networks?
    - Multilayer neural networks?
    - …

Disclaimer: Terminol

- ~~Deep learning~~

https://www.youtube.com/watch?v=H7DgMFqrON0



together—not separately—as they chased the Watergate story. It provided a way of sharing ideas. In Toronto, one idea was a new name for this very old technology.

When Hinton gave a lecture at the annual NIPS conference, then held in Vancouver, on his sixtieth birthday, the phrase "deep learning" appeared in the title for the first time. It was a cunning piece of rebranding. Referring to the multiple layers of neural networks, there was nothing new about "deep learning." But it was an evocative term designed to galvanize research in an area that had once again fallen from favor. He knew the name was a good one when, in the middle of the lecture, he said everyone else was doing "shallow learning," and his audience let out a laugh. In the

Page 64

14%

Genius Makers, Cade Metz 2021.

2007

What is a layer?

Typically matrix multiplication! (But the function can take many forms*)

- **Fully-connected** layer
- **Convolution** layer
- **Pooling** layer (e.g., Max-pooling)
- **Non-linearity** layer (e.g., ReLU)
- **Attention** layer
- …

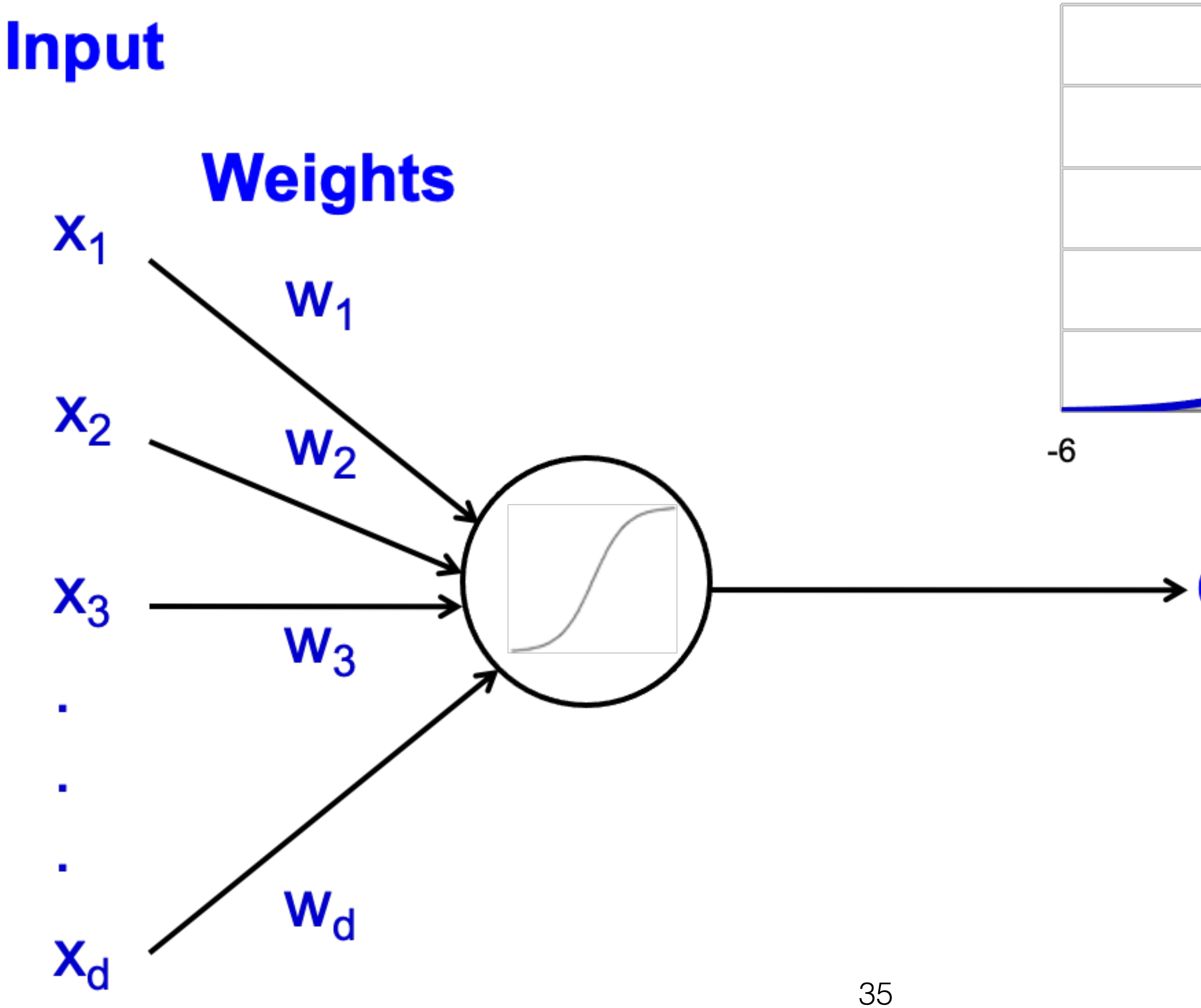
More on different types of layers next week

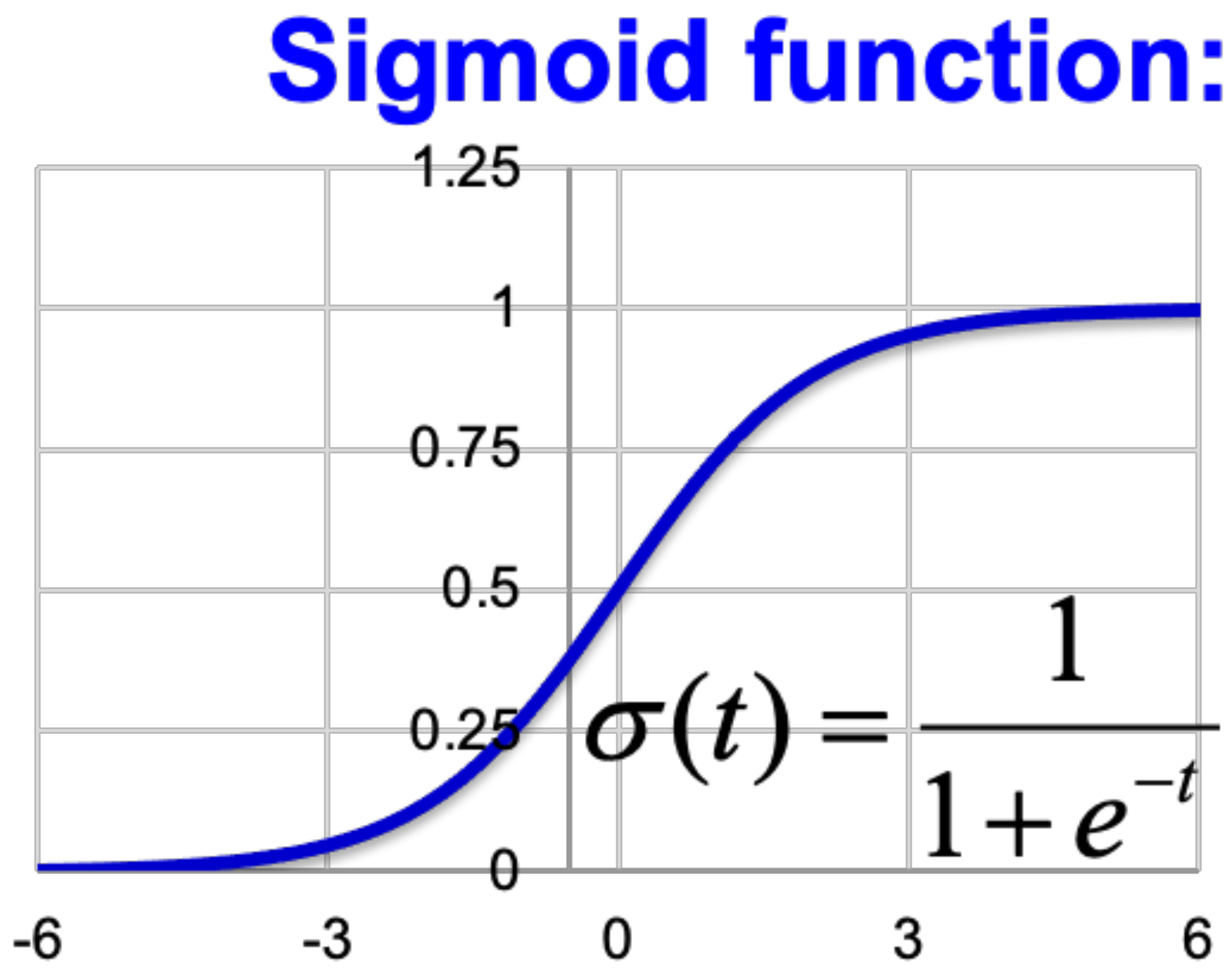
*requirement to be differentiable if optimized with gradient descent algorithm variants

What is a neuron? Perceptrons

[Rosenblatt, 1957]

Most basic form of a neural network





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

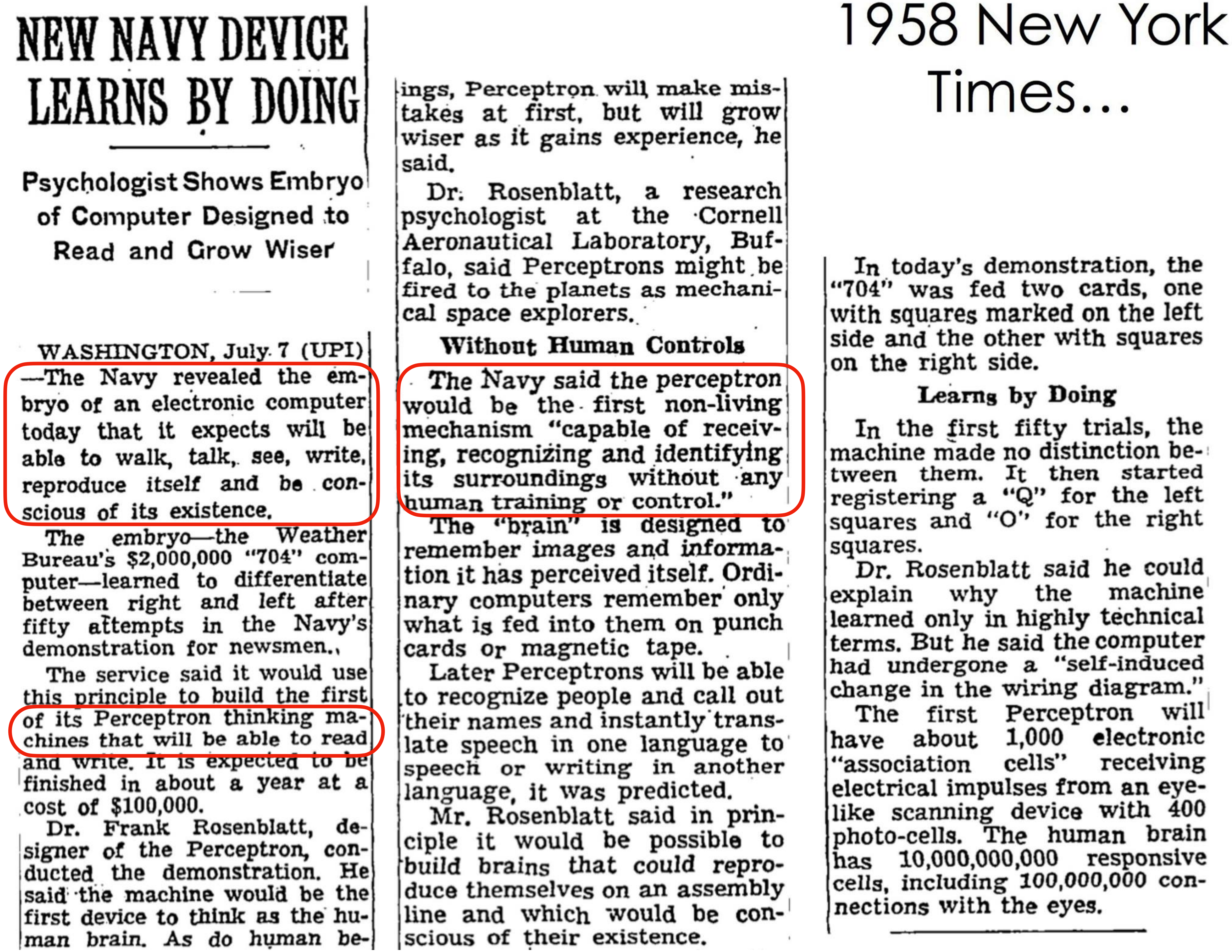
Output: $\sigma(\mathbf{w} \cdot \mathbf{x} + b)$

Non-linearity

Linear combination of inputs

Bias

1958 New York Times...



NEW NAVY DEVICE LEARNS BY DOING

Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

WASHINGTON, July 7 (UPI)—The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

The embryo—the Weather Bureau's $2,000,000 "704" computer—learned to differentiate between right and left after fifty attempts in the Navy's demonstration for newsmen.,

The service said it would use this principle to build the first of its Perceptron thinking machines that will be able to read and write. It is expected to be finished in about a year at a cost of $100,000.

Dr. Frank Rosenblatt, designer of the Perceptron, conducted the demonstration. He said the machine would be the first device to think as the human brain. As do human beings, Perceptron will make mistakes at first, but will grow wiser as it gains experience, he said.

Dr. Rosenblatt, a research psychologist at the Cornell Aeronautical Laboratory, Buffalo, said Perceptrons might be fired to the planets as mechanical space explorers.

Without Human Controls

The Navy said the perceptron would be the first non-living mechanism "capable of receiving, recognizing and identifying its surroundings without any human training or control."

The "brain" is designed to remember images and information it has perceived itself. Ordinary computers remember only what is fed into them on punch cards or magnetic tape.

Later Perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech or writing in another language, it was predicted.

Mr. Rosenblatt said in principle it would be possible to build brains that could reproduce themselves on an assembly line and which would be conscious of their existence.

In today's demonstration, the "704" was fed two cards, one with squares marked on the left side and the other with squares on the right side.

Learns by Doing

In the first fifty trials, the machine made no distinction between them. It then started registering a "Q" for the left squares and "O" for the right squares.

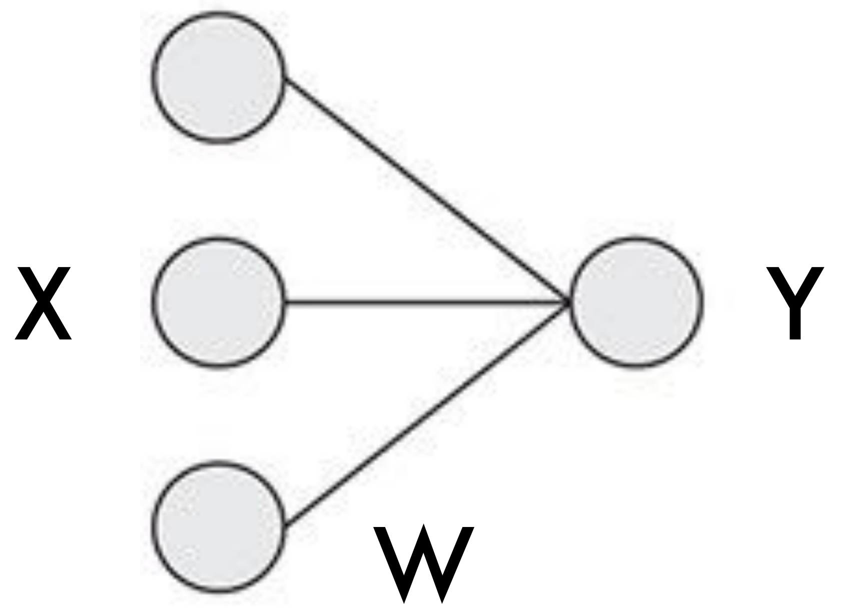
Dr. Rosenblatt said he could explain why the machine learned only in highly technical terms. But he said the computer had undergone a "self-induced change in the wiring diagram."

The first Perceptron will have about 1,000 electronic "association cells" receiving electrical impulses from an eye-like scanning device with 400 photo-cells. The human brain has 10,000,000,000 responsive cells, including 100,000,000 connections with the eyes.

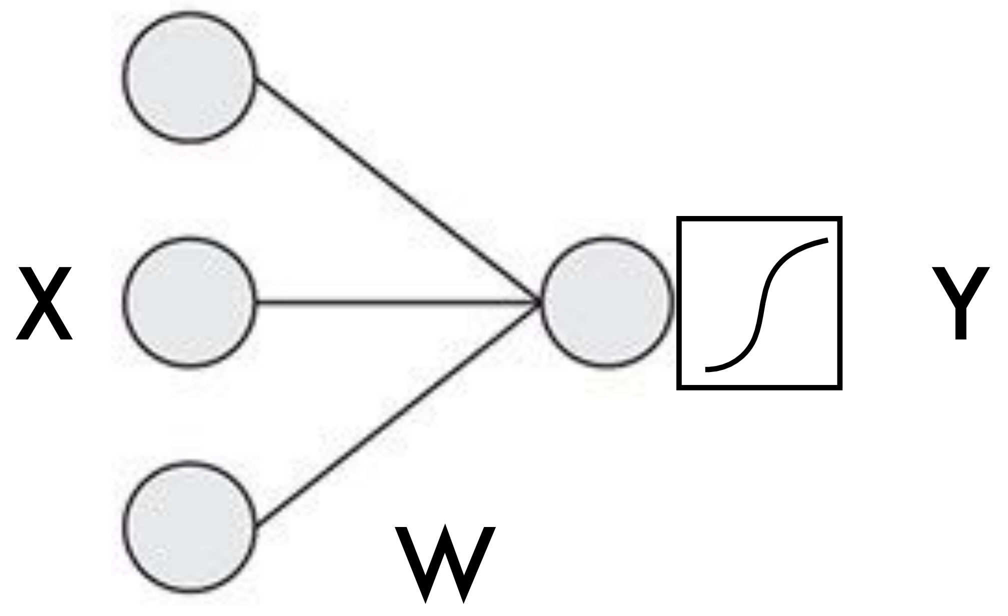
Slide credit: Lana Lazebnik

Multi-Layer Perceptron (MLP)

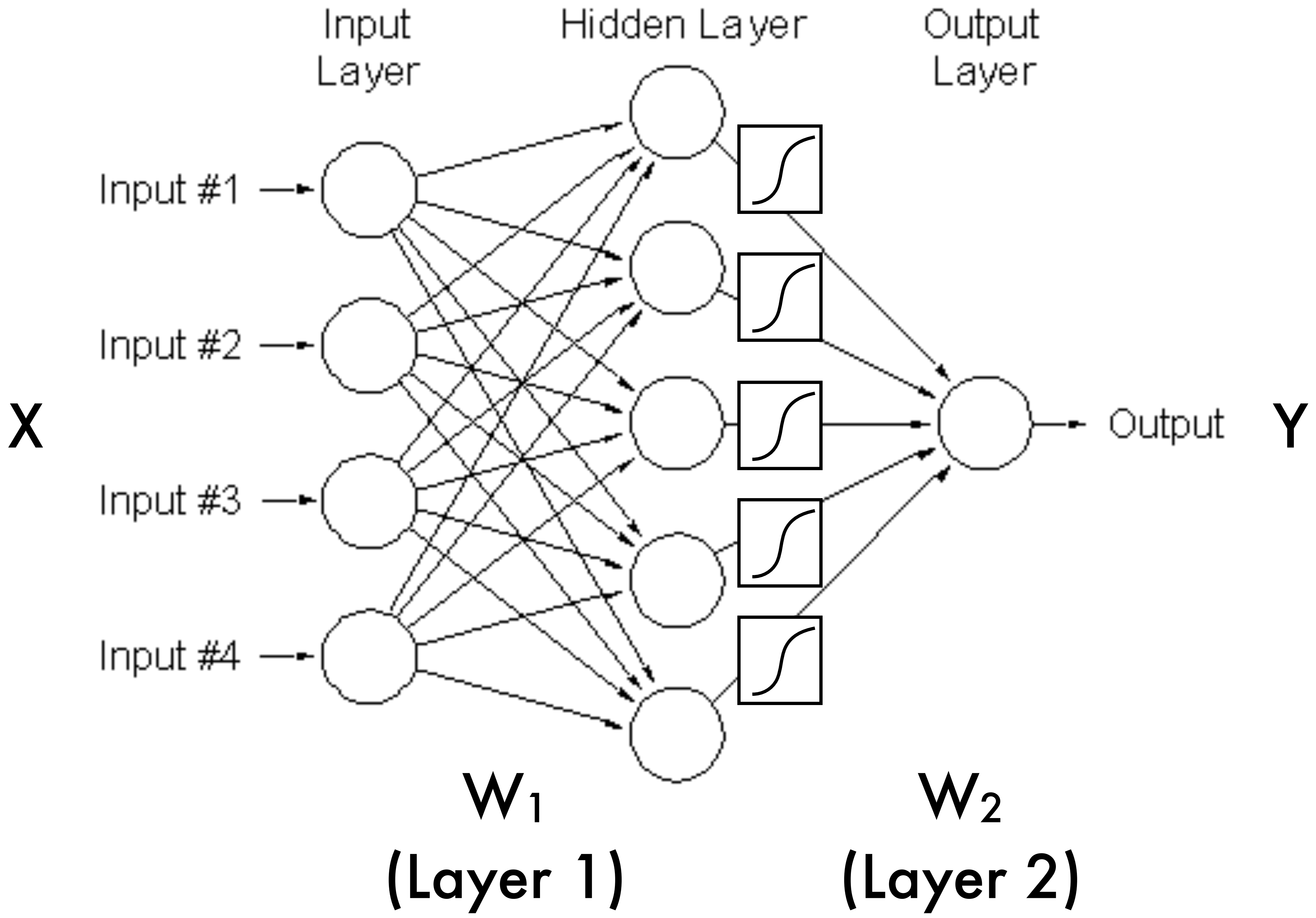
Linear regression:



Perceptron:

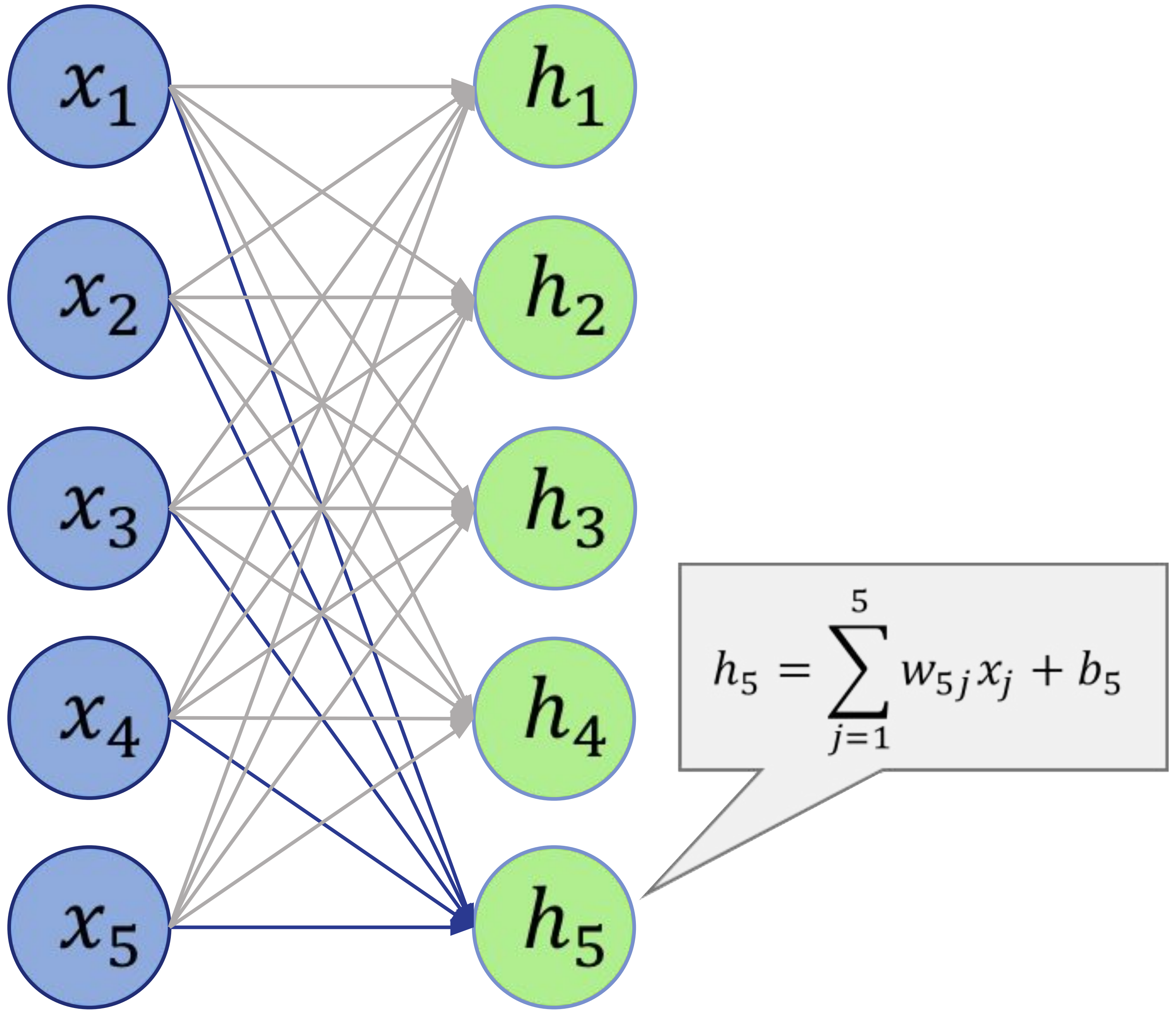


MLP:



Slide: R. Fergus / S. Lazebnik

Multi-Layer Perceptron (MLP)

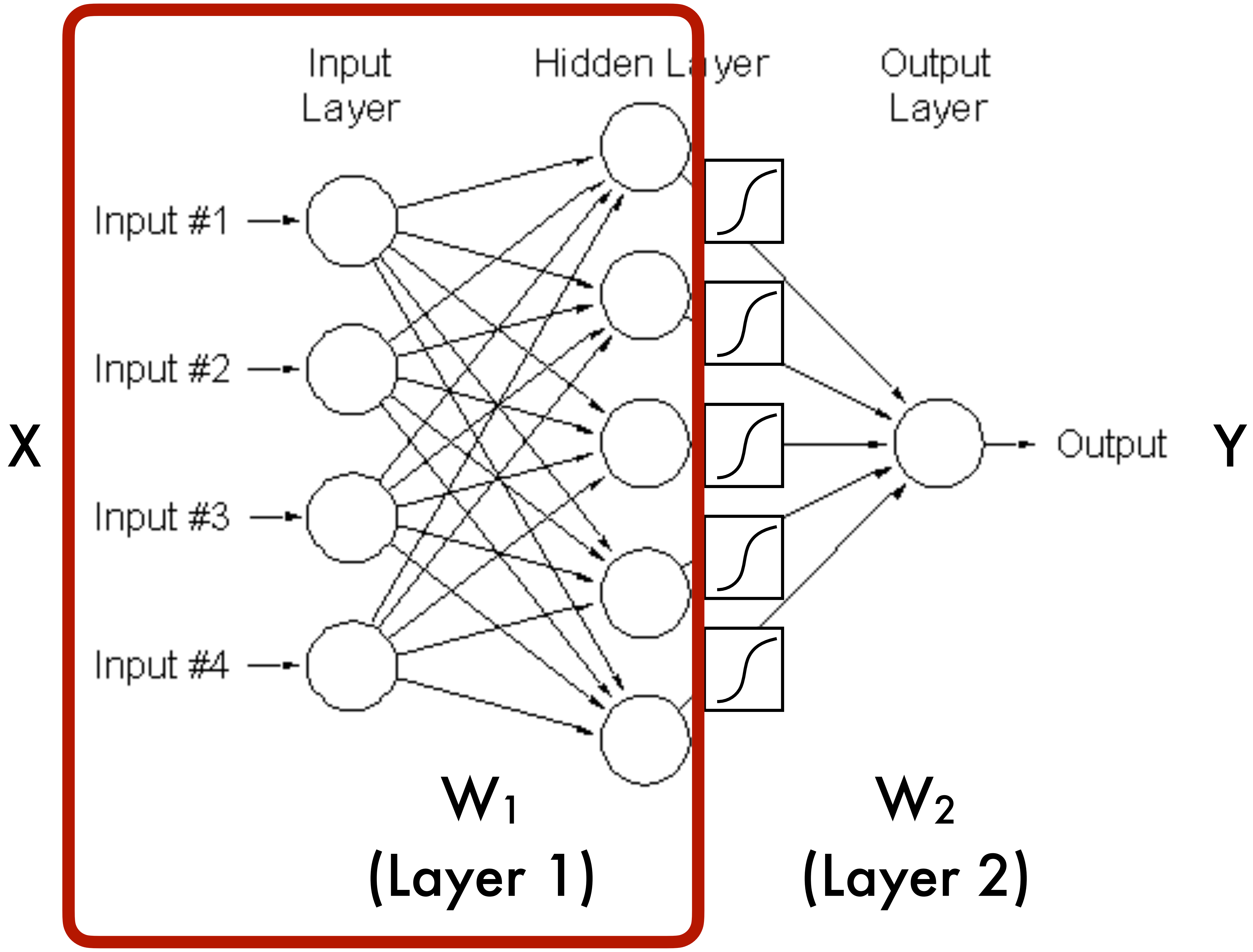


$$h_5 = \sum_{j=1}^{5} w_{5j} x_j + b_5$$

$$\boldsymbol{h} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b};\ h_i = \sum_j w_{ij} x_j + b_i$$

Linear / fully connected layer
= multiplication

MLP:



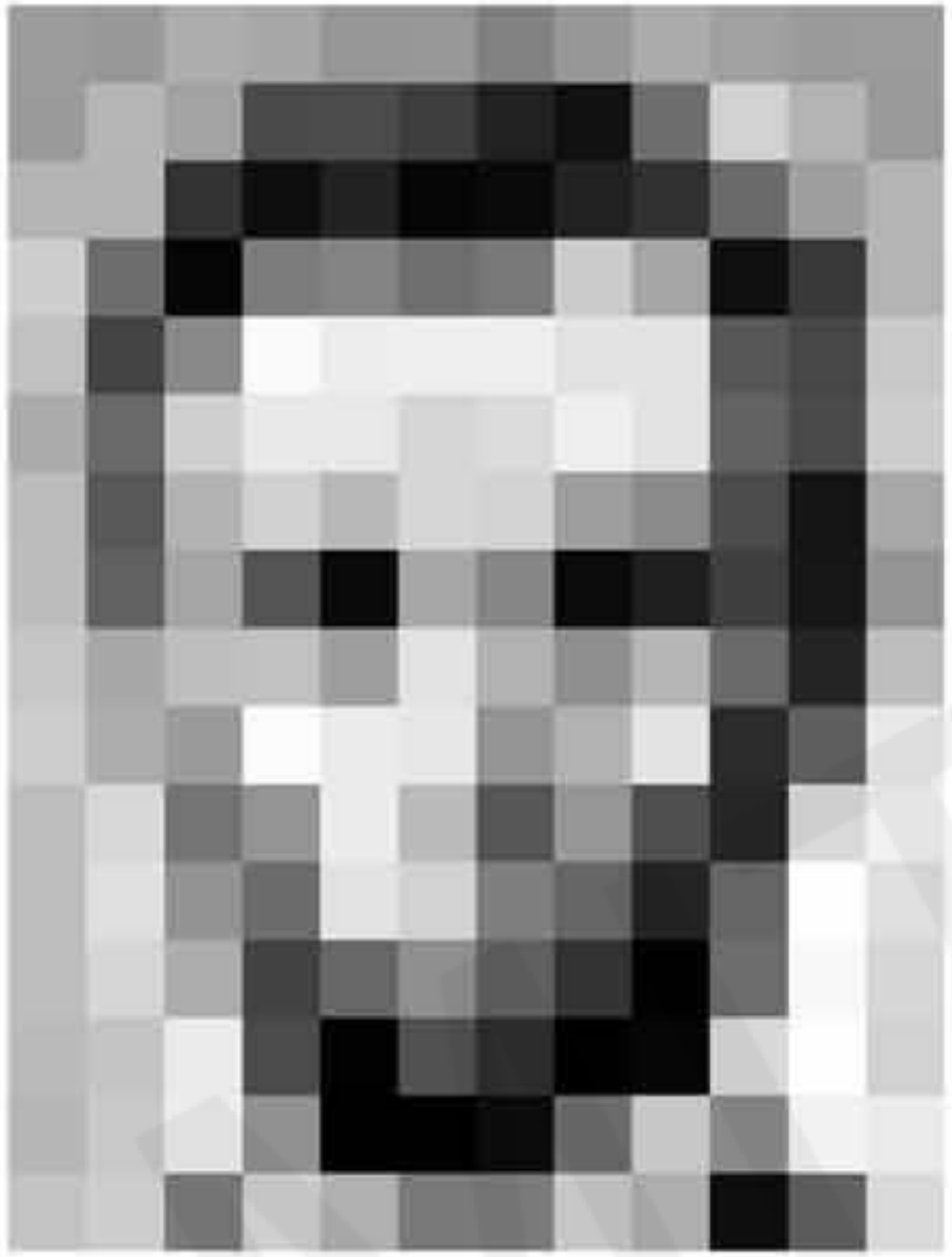
W_1
(Layer 1)

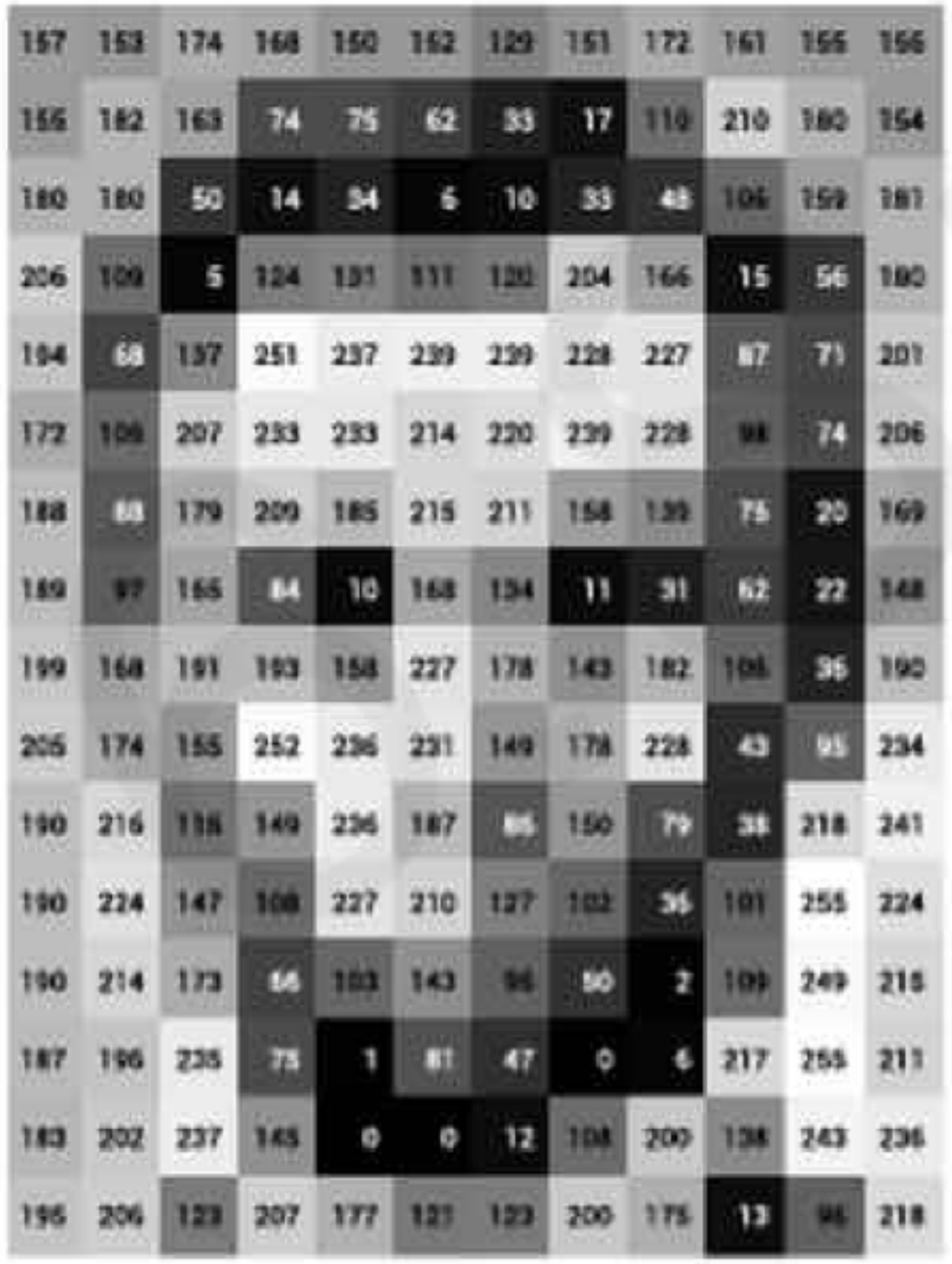
W_2
(Layer 2)

Slide: R. Fergus / S. Lazebnik

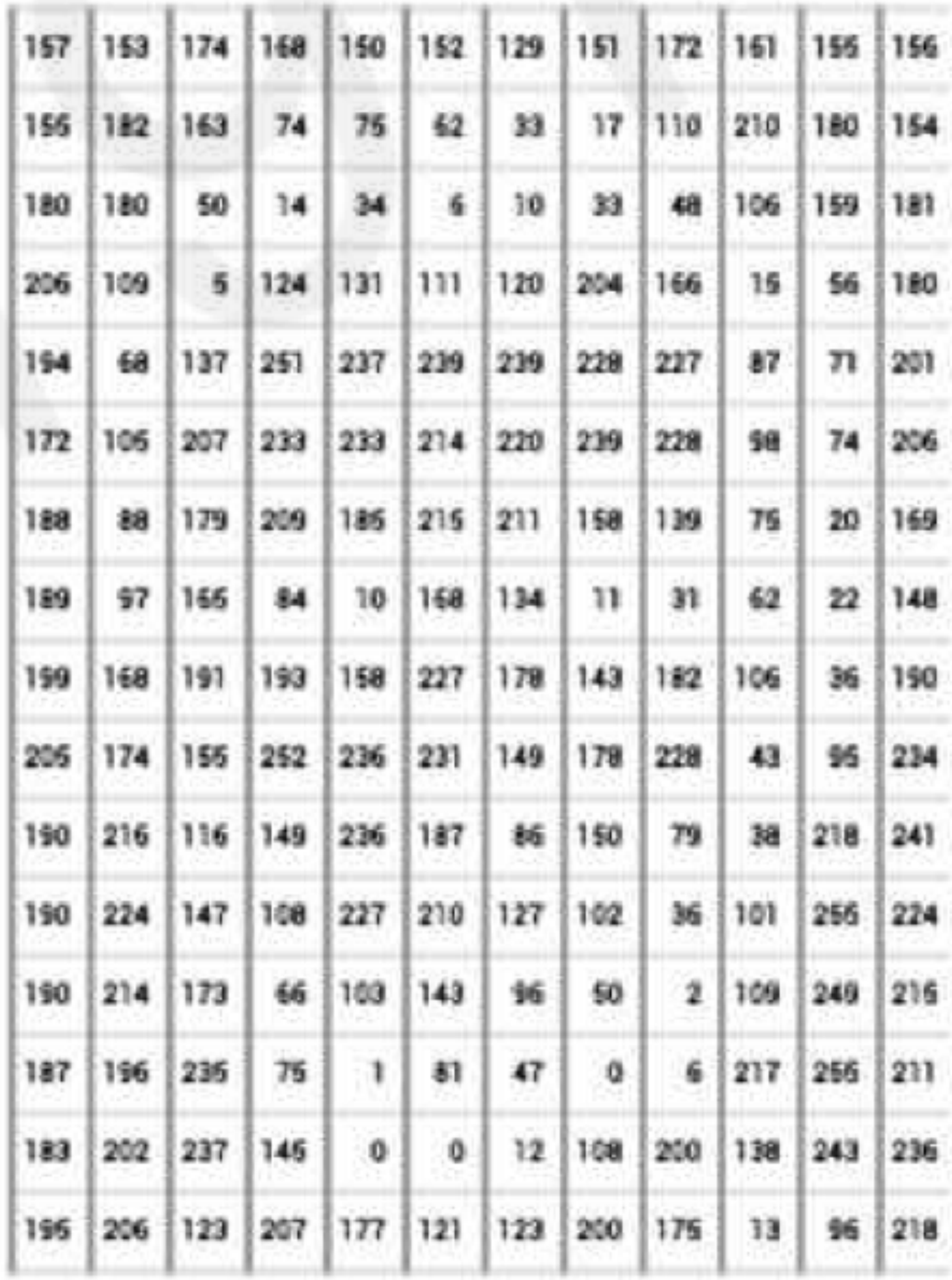
Neural networks for Computer Vision

Images are numbers





What the computer sees

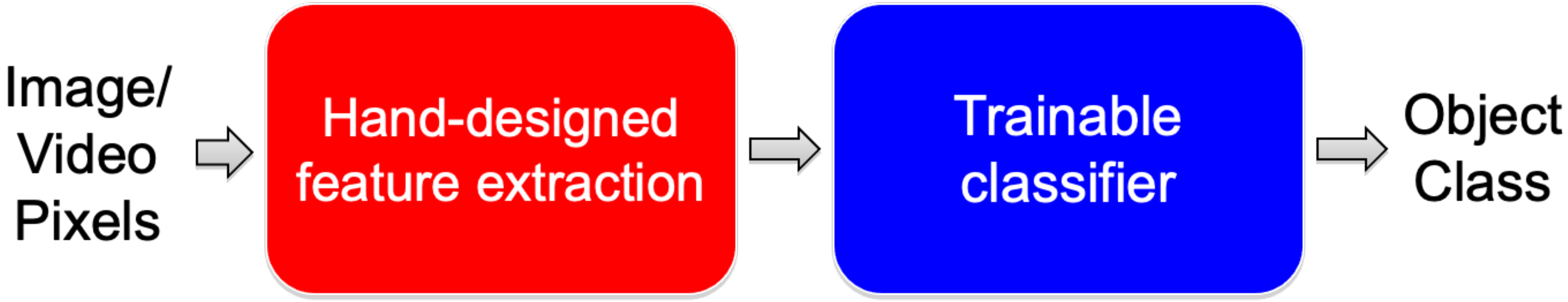


157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

An image is just a matrix of numbers [0,255]!
i.e., 1080x1080x3 for an RGB image

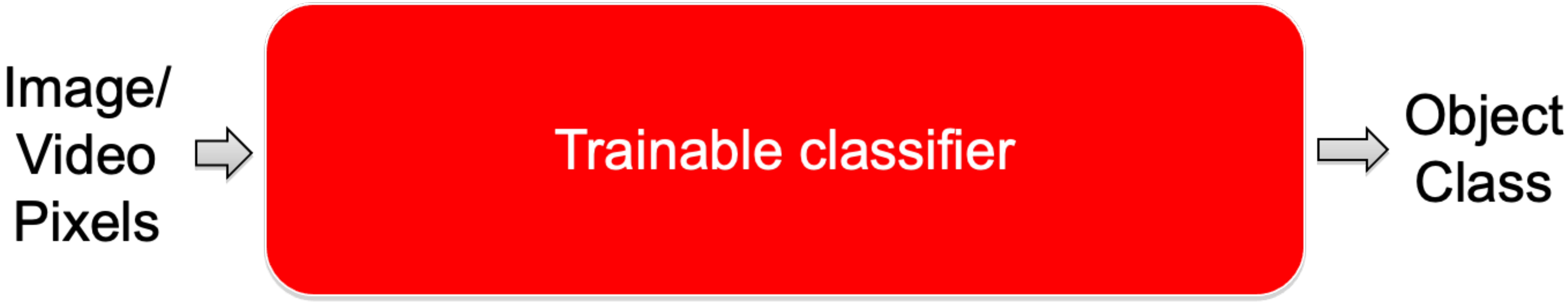
Slide credit: Alexander Amini

Analogy to the traditional visual recognition pipeline



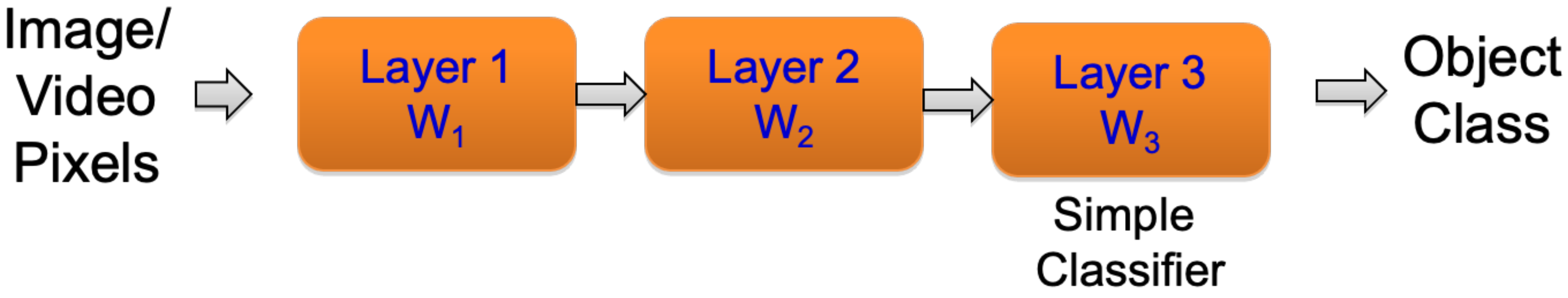
- Features are not learned (e.g., HOG, SIFT, Bag of Features)
- Trainable classifier is often generic (e.g., SVM, Random Forest)

Analogy to the traditional visual recognition pipeline



- Features are learned “end-to-end” (i.e., pixels are input)
- ”Feature hierarchy” all the way from pixels to classifier
- Each layer extracts features from the output of previous layer
- Train all layers jointly

Analogy to the traditional visual recognition pipeline



- Features are learned “end-to-end” (i.e., pixels are input)
- ”Feature hierarchy” all the way from pixels to classifier
- Each layer extracts features from the output of previous layer
- Train all layers jointly

Agenda:

- Machine learning concepts
- Basics of supervised learning
- Introduction to neural networks
- Training neural networks
    - Loss
    - Gradient descent and variants
    - Learning rate
    - Backpropagation
    - Regularization

Training NNs

Recap: Basics of supervised learning

- n training data pairs

$$(x_1, y_1), \ldots, (x_n, y_n) \in \mathcal{X} \times \mathcal{Y}$$

- Learn a predictor/decision function

$$\hat{f} : \mathcal{X} \to \mathcal{A}$$

- By minimizing

$$\sum_{i=1}^{n} \underbrace{l}_{\text{Loss}}(\underbrace{f}_{\text{Model}}(\underbrace{x_i}_{\text{Input}}), \underbrace{y_i}_{\text{Label}})$$

How can we define *f*?

- Using a set of parameters
  - e.g., linear/polynomial regression, **neural networks**
- Directly using the training data, i.e., non-parametric
  - e.g., k-nearest neighbors

Loss Function

- Regression:
  - L1 (absolute error) / L2 (squared error)
- Classification:
  - Cross-entropy loss

Loss Function: Regression

Estimating a continuous value

- L1 (absolute error)

$$L = \left|\ f(X_i,\ \theta) - Y_i \right|$$

- L2 (squared error)

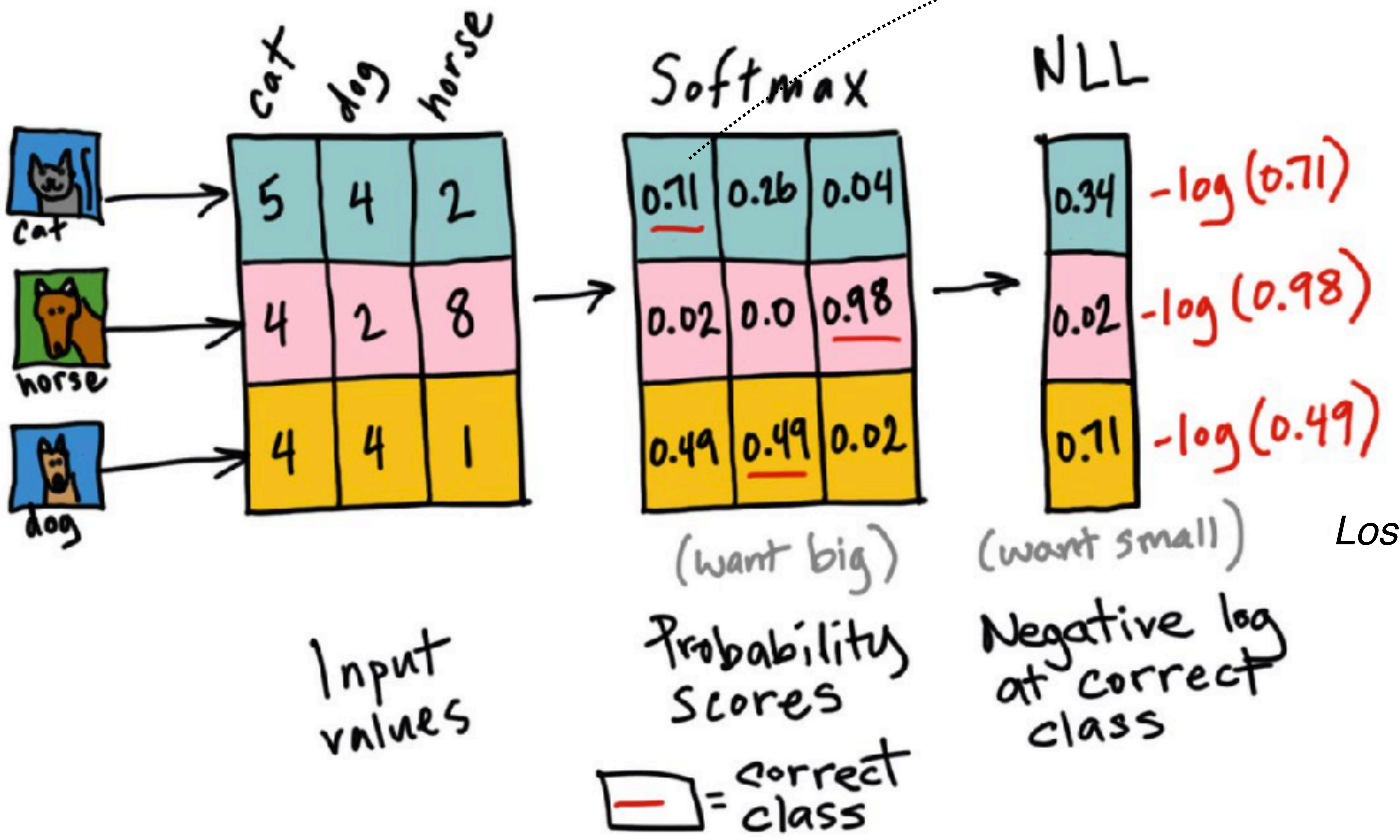
$$L = \left(\underbrace{f(X_i,\ \theta)}_{\substack{\textbf{Prediction:}\\ \text{output of}\\ \text{the network f}\\ \text{with parameters } \theta\\ \text{given input } X_i}} - \underset{\substack{\searrow\\ \textbf{Ground truth:}\\ \text{(label, annotation)}}}{Y_i}\right)^2$$

Loss Function: Classification

- Cross-entropy loss = softmax + negative log-likelihood

$$\text{loss}(\text{x}, \text{class}) = -\log\left(\frac{\exp(\text{x}[\text{class}])}{\sum_j \exp(\text{x}[\text{j}])}\right)$$

$$\frac{e^5}{e^5 + e^4 + e^2}$$



Loss = 0.34+0.02+0.71 = 1.07

Fig: Micheleen Harris

Learning deep networks

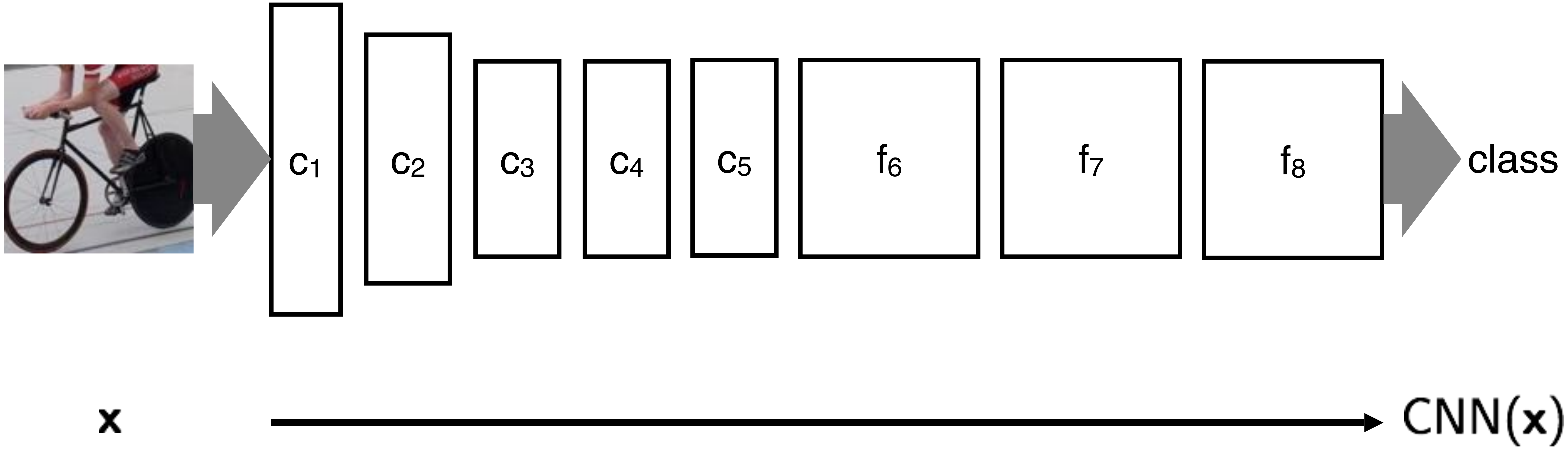
- Non convex!

- Solution: go back to the simplest algorithms, variations around gradient descent, and hope for the best.

- Note: every layer needs to be differentiable almost everywhere.

- It (mainly) works! (but requires a lot of know-how)

Example CNN architecture: AlexNet



From left to right

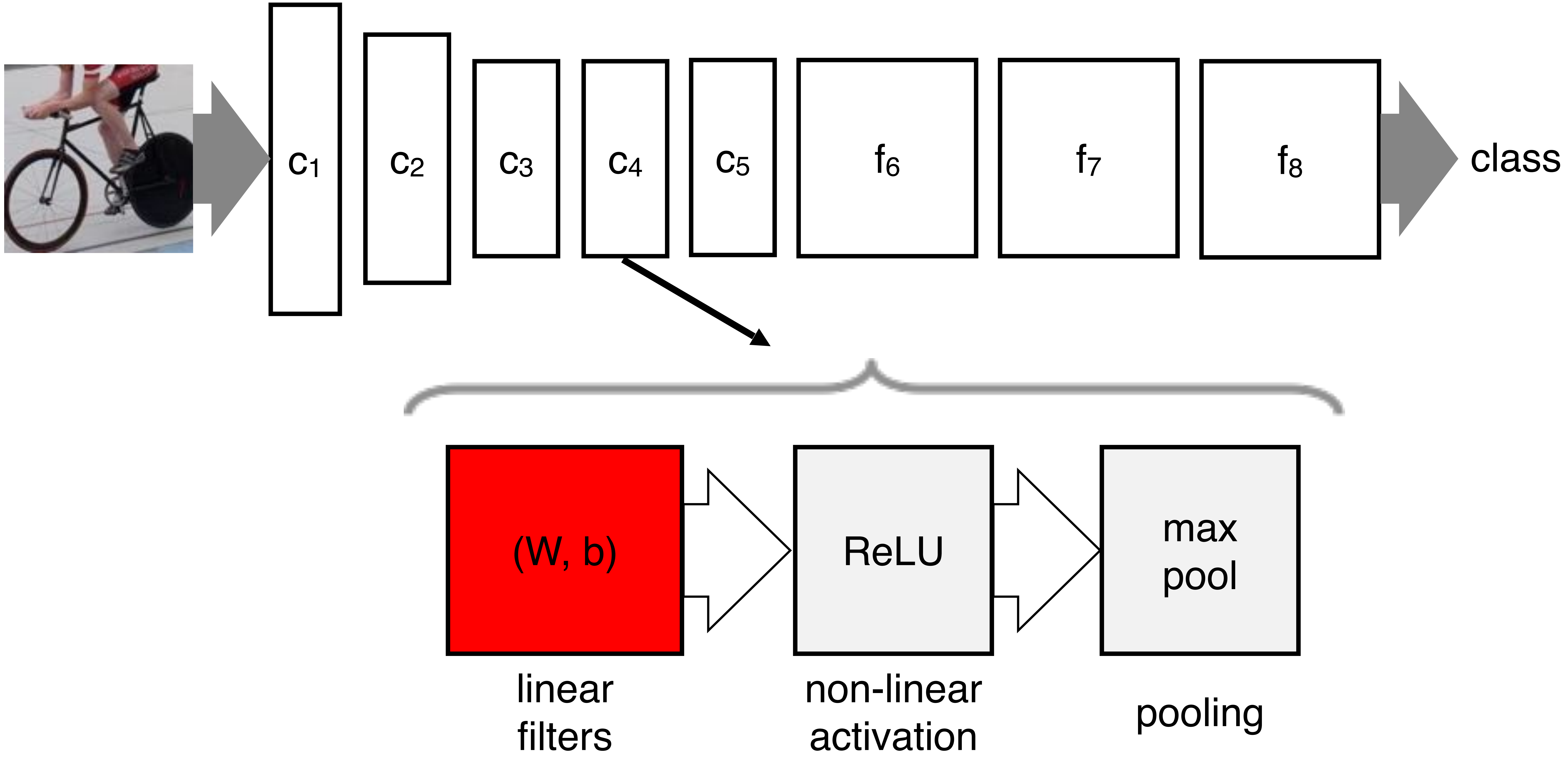
- decreasing spatial resolution
- increasing feature dimensionality

“Fully-connected” layers (f6, f7, f8)

- same as convolutional, but with 1 × 1 spatial resolution
- contain most of the parameters

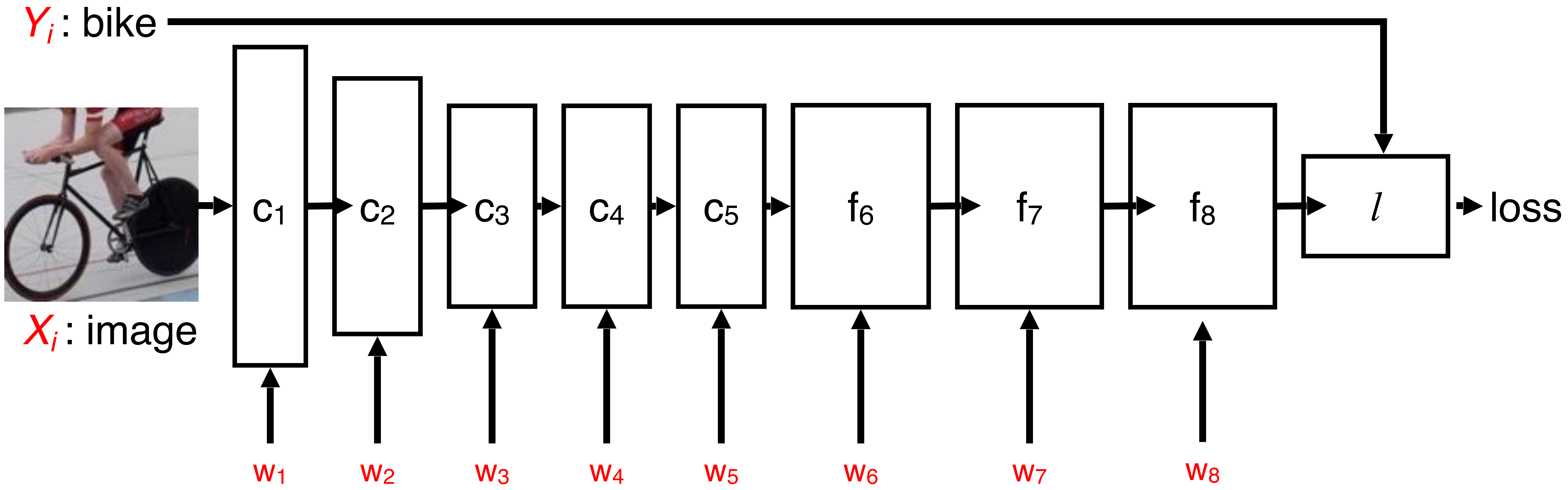
Slide: A. Vedaldi

Convolutional layers



Each block c_1, c_2, …, f_8: convolution + ReLU + pooling.

Training a CNN



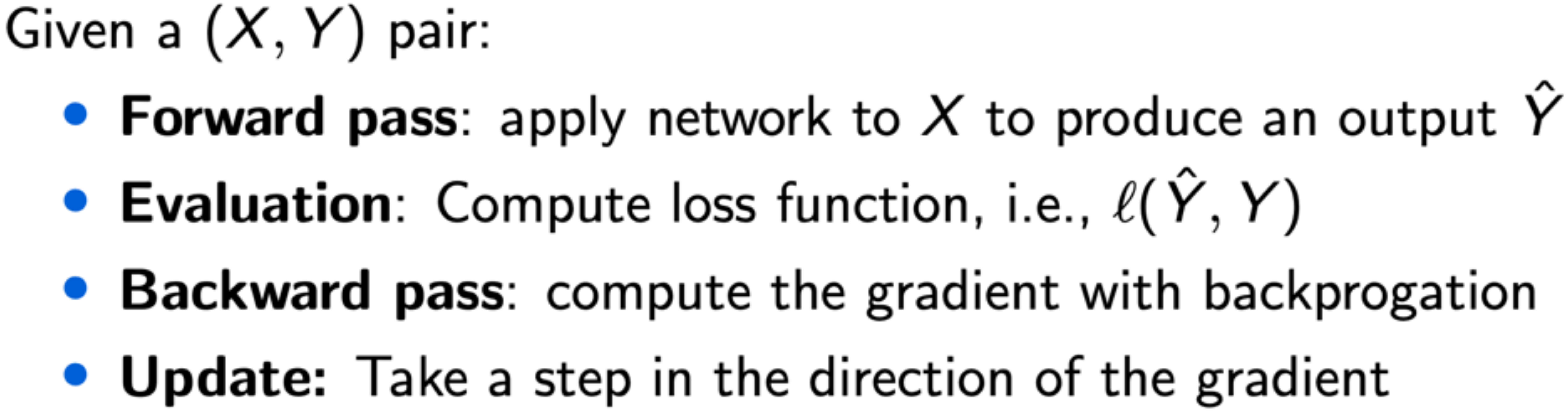
$$\theta = \{w_1,\ w_2, w_3, \ldots,\ w_8\}$$ parameters / filters / weights / kernels…

$$\theta^* = \underset{\theta}{argmin}\ L\big(\theta \,|\, \underbrace{X,\ Y}_{\text{data}}\big), \text{ where } \quad L(\theta) = \frac{1}{N}\sum_i l(\theta,\ X_i,\ Y_i)$$

sum over data points

loss / objective / error

Training a neural network



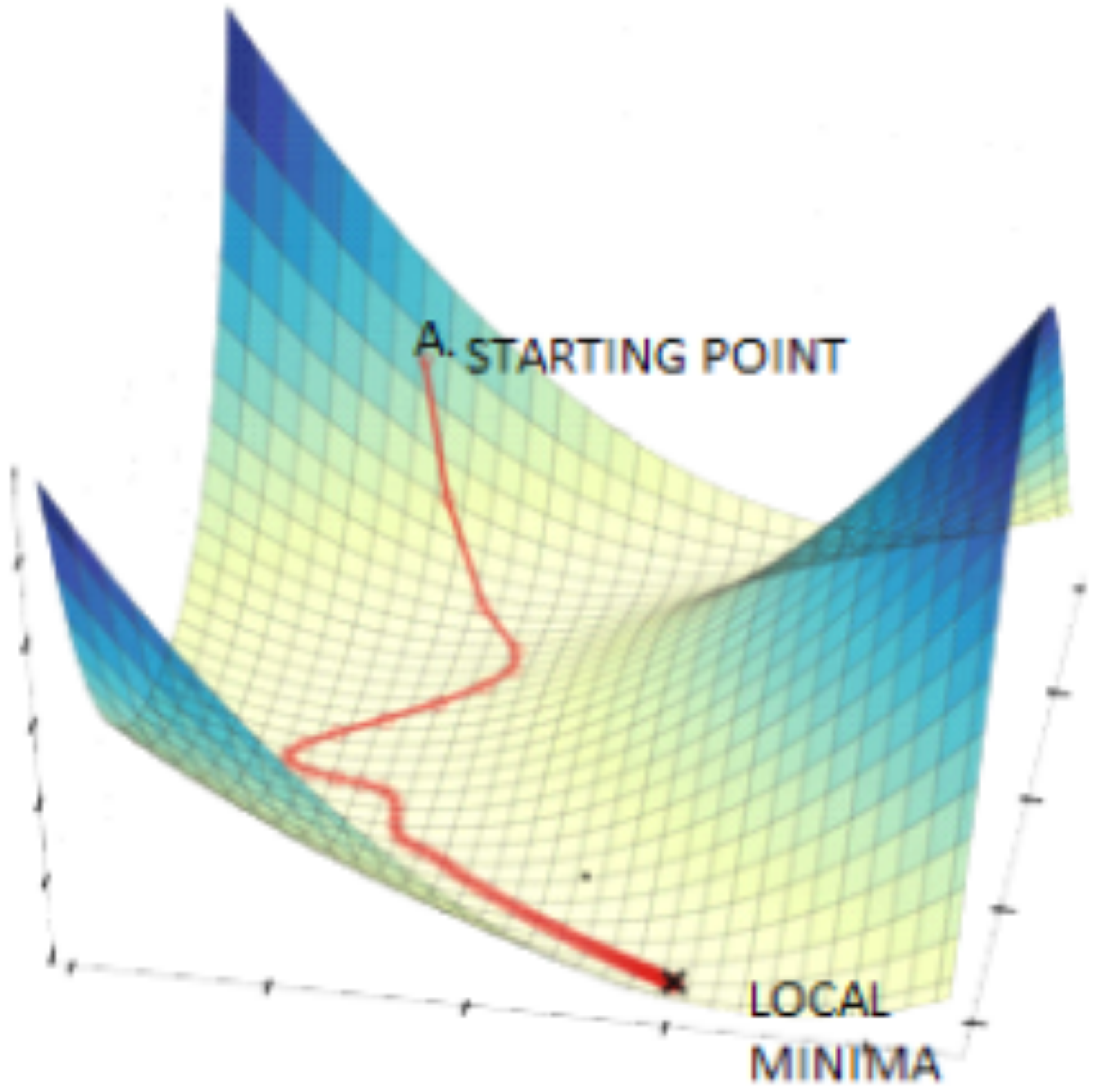
Given a (X, Y) pair:

- **Forward pass**: apply network to X to produce an output $\hat{Y}$
- **Evaluation**: Compute loss function, i.e., $\ell(\hat{Y}, Y)$
- **Backward pass**: compute the gradient with backprogation
- **Update:** Take a step in the direction of the gradient

Slide: A. Joulin

Hill Metaphor

- If you are on a hill and you want to reach the bottom, but can only see a foot around you, what do you do?
  - We should just follow the slope of the hill and hope it gets us down right?
- We will need to take a couple steps in the downward direction, stop and re-evaluate our direction, then take a few more steps and so on

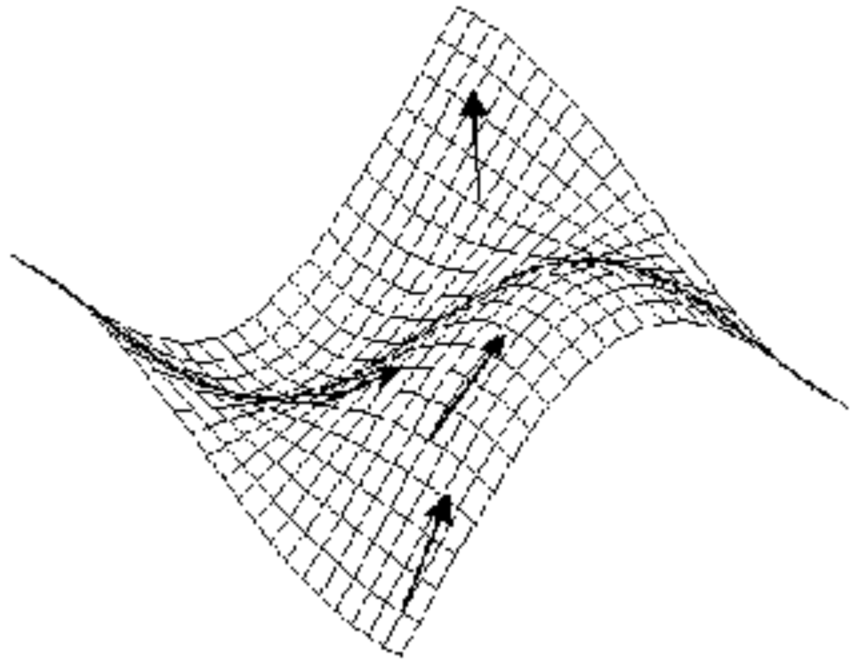


Slide credit: J. Austin, A. Jain

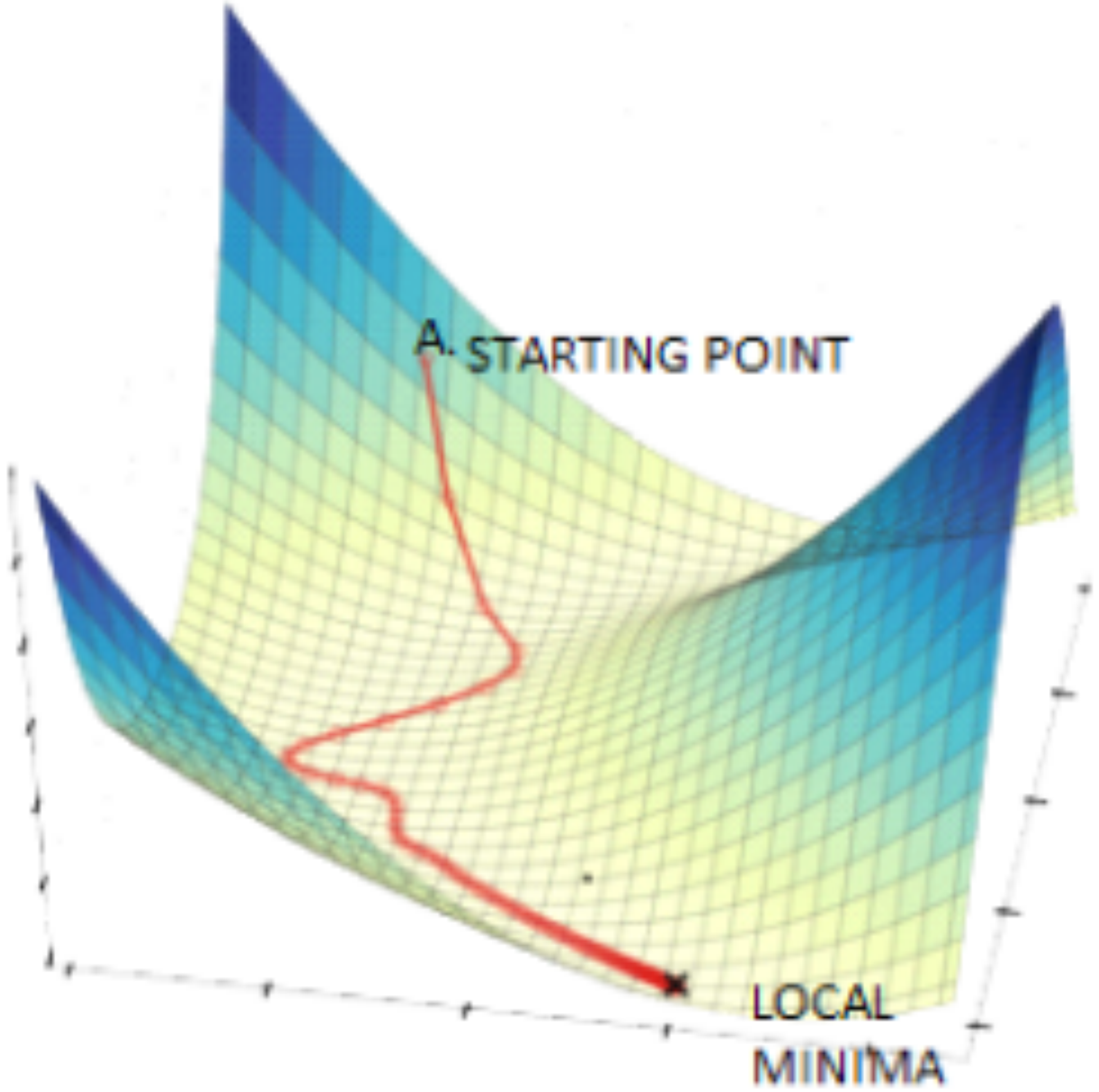
Which direction is the steepest?

- Suppose the hill's elevation is given by a vector function
- It's **steepest descent is along the negative gradient vector**
  - → If we want to go down a hill, follow the **negative gradient <u>evaluated at our current position</u>**

$$\nabla f(w_1, w_2, \ldots) = \begin{bmatrix} \frac{\partial f}{\partial w_1} \\ \frac{\partial f}{\partial w_2} \\ \vdots \end{bmatrix}$$



Gradient Vectors Shown at Several Points on the Surface of cos(x) sin(y)



Slide credit: J. Austin, A. Jain

Gradient descent

- The objective function is an average over all N training data points:

$$L(\theta) = \frac{1}{N}\sum_i l(\theta,\ X_i,\ Y_i)$$

- Performing a gradient descent is iterating.

$$\theta_{t+1} \to \theta_t - \frac{\alpha_t}{N}\sum_i \frac{\partial \ell(\theta, X_i, Y_i)}{\partial \theta}$$

- Need to choose the learning rate policy α_t

- If the function is not convex, get stuck in a local minimum

- Each step can be expensive to compute if the dataset is large

Convex case

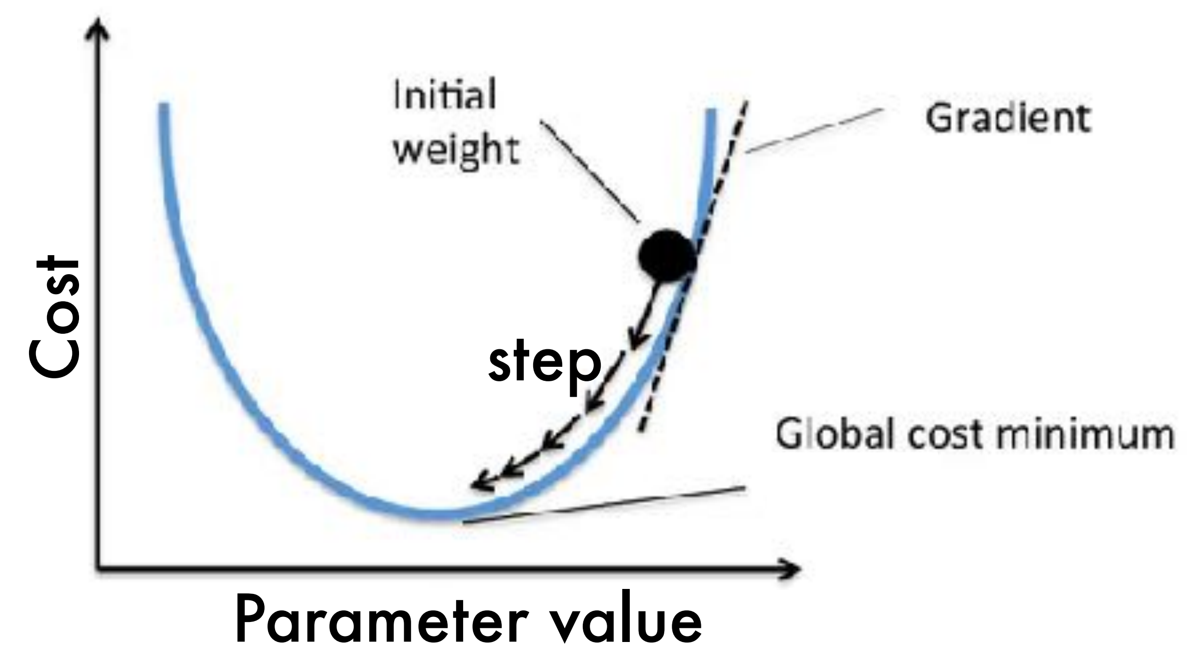


Image source

Non-convex

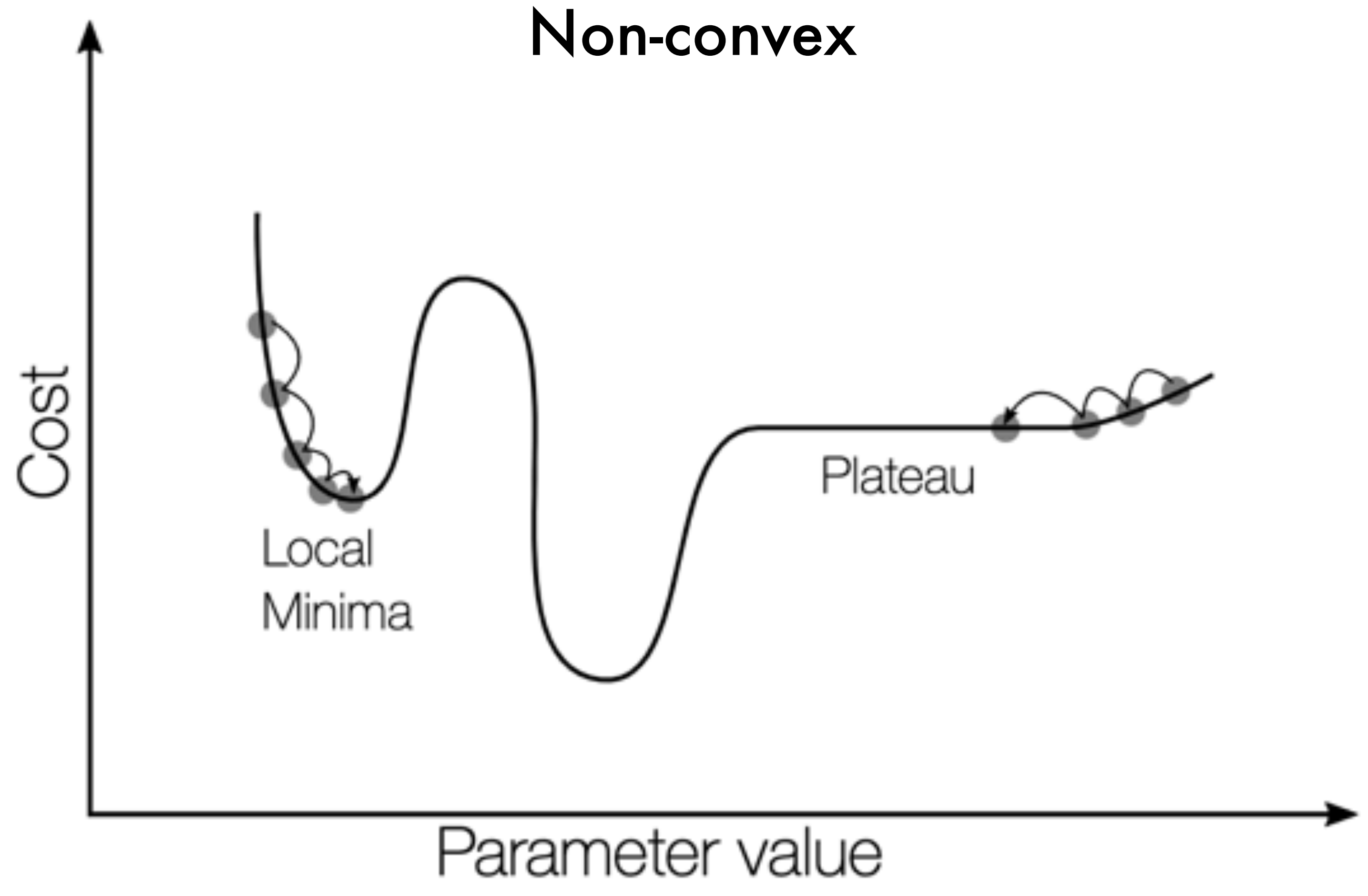


Image source

Stochastic gradient descent

- Instead of computing the gradient, compute an approximation:

$$\theta_{t+1} \rightarrow \theta_t - \frac{\alpha_t}{N} \sum_i \frac{\partial \ell(\theta, X_i, Y_i)}{\partial \theta}$$

$$\Downarrow$$

$$\boxed{\theta_{t+1} \rightarrow \theta_t - \alpha_t \frac{\partial \ell(\theta, X^{(i_t)}, Y^{(i_t)})}{\partial \theta}}$$

- Can take advantage of large datasets, in particular infinite* datasets!

- Introduce stochasticity, which might be good to get out of local minima in the non-convex case

Stochastic gradient descent with minibatch

- Some variance is good, too much can be bad

$$\theta_{t+1} \rightarrow \theta_t - \frac{\alpha_t}{N} \sum_{i} \frac{\partial \ell(\theta, X_i, Y_i)}{\partial \theta}$$

$$\Downarrow$$

$$\boxed{\theta_{t+1} \rightarrow \theta_t - \frac{\alpha_t}{K} \sum_{i=1}^{K} \frac{\partial \ell(\theta, X_i, Y_i)}{\partial \theta}}$$

(with $K << N$)

- It's faster to compute several gradients in parallel Why?

- In practice, using batches as large as possible so that the network fits in the GPU memory (e.g., between 256 images, 10 videos, 1000 features, could be very different depending on the task, network, GPU hardware)

Summary: Stochastic Gradient Descent (SGD)

The objective function is an average over all *N* training data points:

$$\theta_{t+1} \to \theta_t - \frac{\alpha_t}{N} \sum_i \frac{\partial \ell(\theta, X_i, Y_i)}{\partial \theta}$$

(gradient descent)

Key idea: approximate the gradient with 1 random datapoint:

$$\theta_{t+1} \to \theta_t - \alpha_t \frac{\partial \ell(\theta, X^{(i_t)}, Y^{(i_t)})}{\partial \theta}$$

(stochastic gradient descent)

Pick K random points instead of picking 1 (with *K* << *N*):

$$\theta_{t+1} \to \theta_t - \frac{\alpha_t}{K} \sum_{i=1}^{K} \frac{\partial \ell(\theta, X_i, Y_i)}{\partial \theta}$$

(stochastic gradient descent
with mini-batches)

=> commonly used

Slide credit: Andrea Vedaldi

Stochastic gradient descent (SGD)

Details:

- **Epochs**: all points are visited sequentially, but random order within epoch
- **Minibatch size**: set to largest value permitted by the hardware
- **Validation**: evaluate L(θ) on a held-out validation set to diagnose objective decrease
- **Learning rate**: e.g., decreased tenfold once the objective L(θ) stops decreasing, cosine LR scheduler…
- **SGD with momentum**: the gradient estimate is smoothed by using a moving average to encourage directions that are coherent:

$$M_t = \gamma M_{t-1} + \eta g_t$$

$$\theta_{t+1} = \theta_t - M_t$$

Optimization

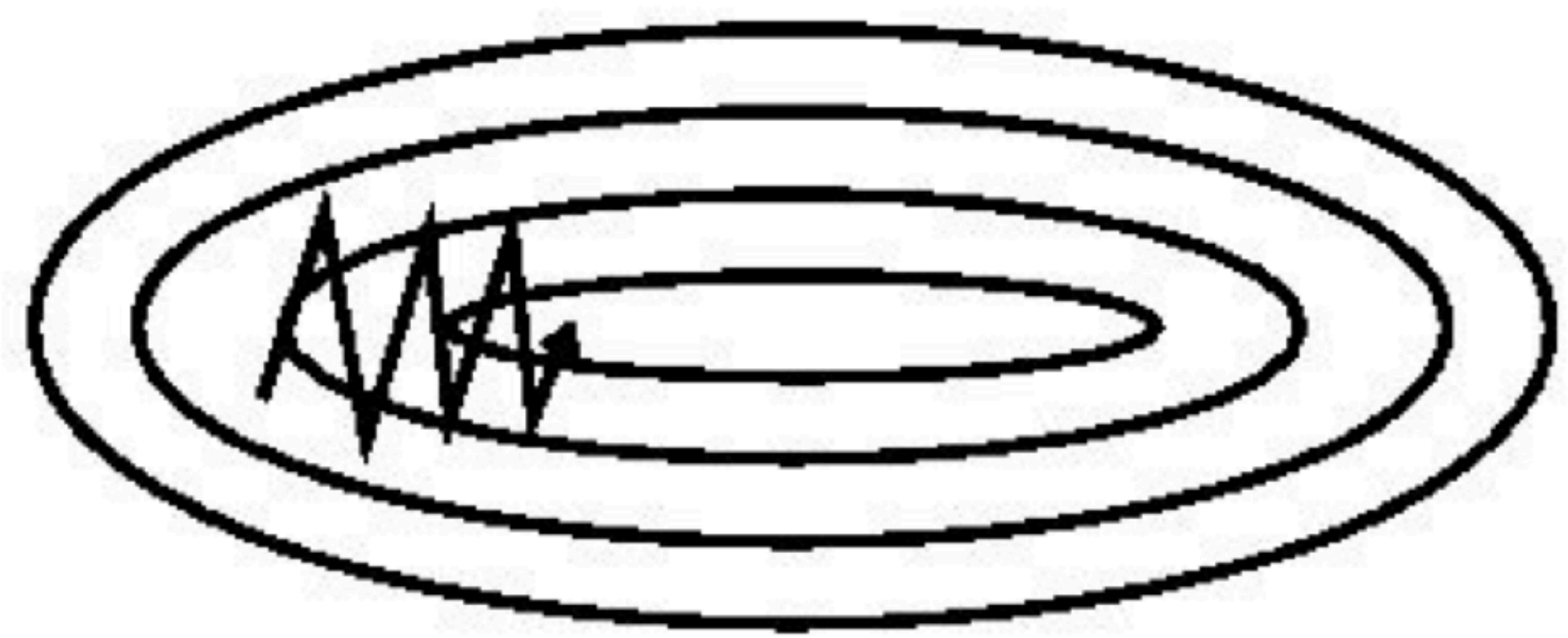
- Can we do better than vanilla gradient descent? Yes

vanilla update rule for reference

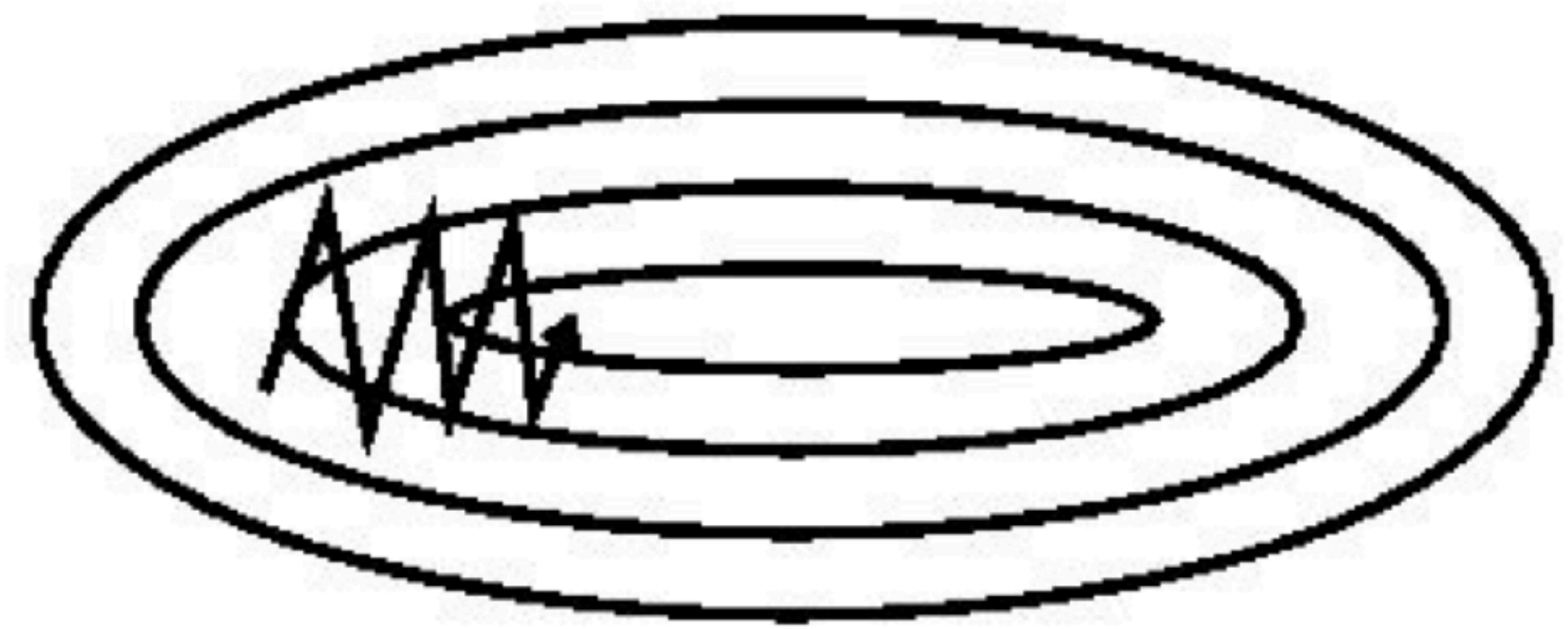
$$\bar{\theta}_{t+1} = \bar{\theta}_t - \lambda \nabla_{\bar{\theta}} \mathrm{L} \Big|_{(\bar{\theta}_t, \,\mathrm{data})}$$

Beyond vanilla SGD

Not all direction are equal



Not all direction are equal



We want to go fast in some directions, slow in others

Vectorized SGD: one step size per dimension

Scalar stepsize:

$$\theta_{t+1,i} = \theta_{t,i} - \alpha_t g_{t,i}$$

Vector stepsize:

$$\theta_{t+1,i} = \theta_{t,i} - \boxed{\alpha_{t,i}} g_{t,i}$$

$\theta_{t,i}$ i^{th} dimension of the parameters at step *t* (scalar)

Vectorized stepsize example: Adagrad

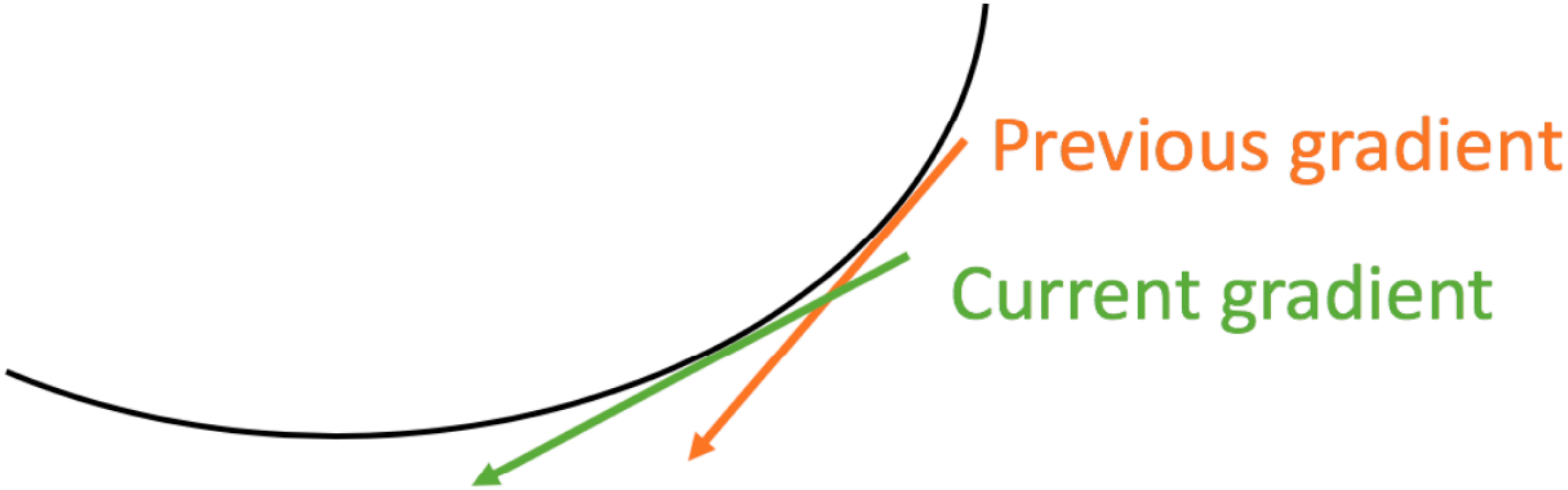
- Some parameters might have more gradient signal than others.
- Adapt the learning rate to how much signal there is for each gradient parameter.

$$G_{t,i} = \sum_{j=1}^{t} g_{j,i}^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,i} + \varepsilon}} g_{t,i}$$

- “No need” to set a learning rate schedule
- $G_{t,i}$ is the accumulation of the squared gradients
- Squared norm avoids exploding or vanishing gradient
- ε avoids numerical issues.

Use previous gradients



Previous gradients are not bad estimates of current curvature

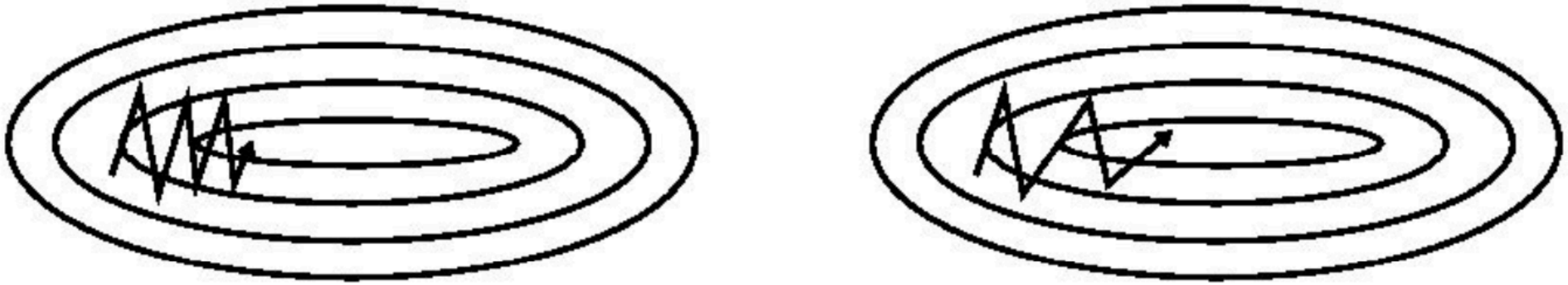
Example: momentum

- Gradient estimate is smoothed by using a moving average to encourage directions that are coherent:

$$M_t = \gamma M_{t-1} + \eta g_t$$
$$\theta_{t+1} = \theta_t - M_t$$

- γ controls the inertia
- M_t is almost a moving average
- Typically, $\gamma = 0.9$ or 0.99

Example: momentum



Vectorized stepsize example: RMSProp

- Instead of keeping a weighted average of gradients, keep a weighted average of **squared gradient components**
  - **Similar idea to AdaGrad, but running avg** instead of sum over all samples for the normalization
  - Similar to ADAM, but no momentum

Vectorized stepsize example: RMSProp

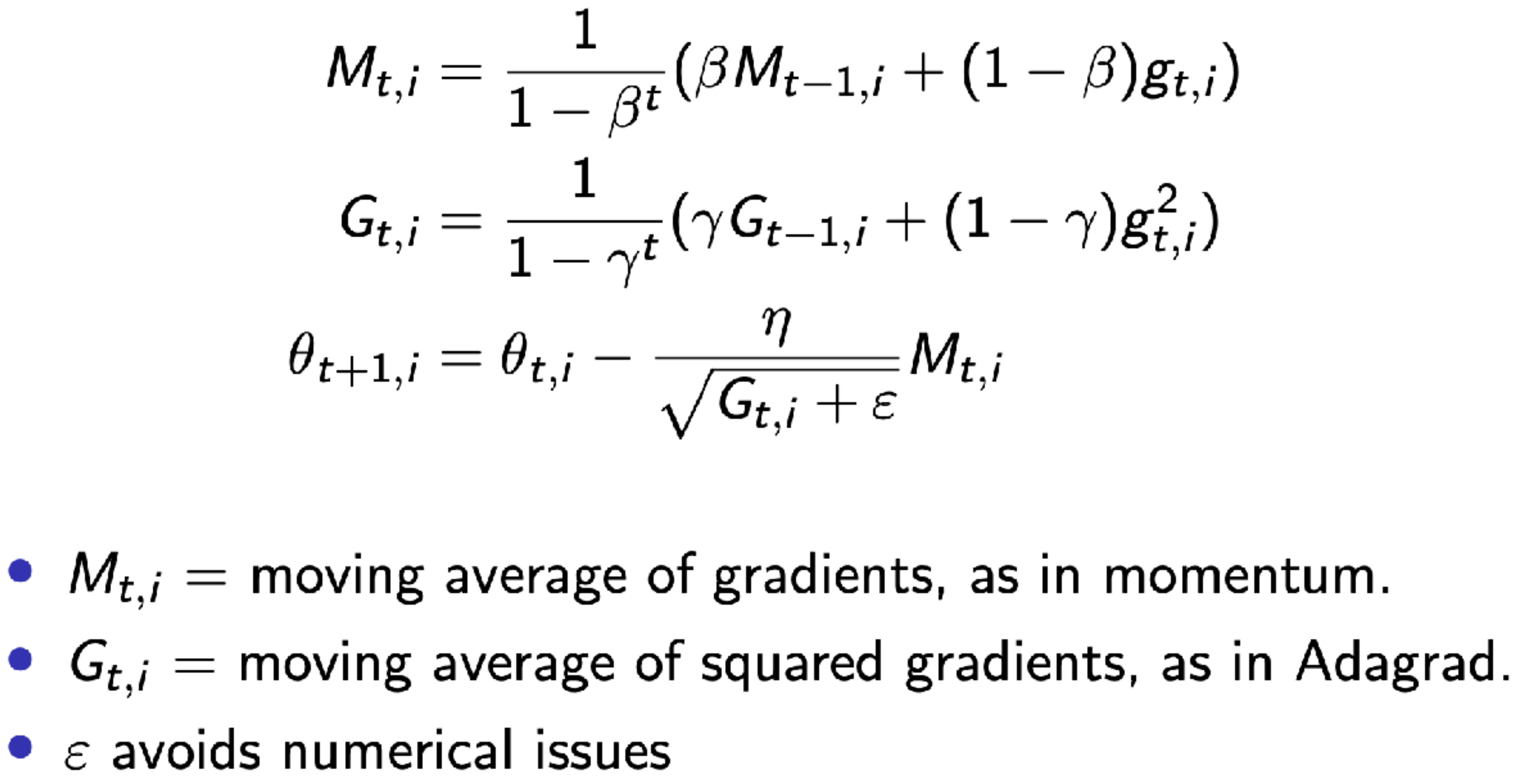
- Case 1: The gradients have been really small in the past
  - Moving average of squared gradients will be even tinier
  - The square root of this moving average will be a really small number, and dividing by it should increase the size of the final gradient update
- Case 2: Our gradients have been really big in the past
  - Moving average of squared gradients will be huge
  - The square root of this moving average will be a really large number, and dividing by it should decrease the size of the final update

Takeaway: this helps combat the issue that gradients can be varying in size, causing us to either get stuck from small gradients or blow past our mark with large gradients. RMSProp makes sure our steps never get too big or too small!

Slide credit: J. Austin, H. Jalan, A. Jain

Momentum + vectorized stepsize = ADAM

- Combination of momentum and RMSProp
- Keep 2 moving averages: 1 for the gradients and 1 for the squared gradients



$$M_{t,i} = \frac{1}{1-\beta^t}(\beta M_{t-1,i} + (1-\beta)g_{t,i})$$

$$G_{t,i} = \frac{1}{1-\gamma^t}(\gamma G_{t-1,i} + (1-\gamma)g_{t,i}^2)$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,i}+\varepsilon}}M_{t,i}$$

- $M_{t,i}$ = moving average of gradients, as in momentum.
- $G_{t,i}$ = moving average of squared gradients, as in Adagrad.
- ε avoids numerical issues

Beyond SGD Summary

- Many other algorithms. Good overview http://ruder.io/optimizing-gradient-descent/

- AdaGrad: Some parameters might have more gradient signal than others. Adapt the learning rate to how much signal there is for each gradient parameter.

$$\theta_{t+1,i} \leftarrow \theta_{t,i} - \frac{\gamma_t}{\sqrt{N_{t,i}}} G_{t,i} \qquad N_{t,i} = \sum_{\tau=1}^{t} G_{t,i}^2$$

- RMSProp: Similar idea to AdaGrad, but use running average instead of sum over all samples for the normalization

$$\theta_{t+1,i} \leftarrow \theta_{t,i} - \frac{\gamma_t}{\sqrt{N_{t,i}}} G_{t,i} \qquad N_{t,i} = mN_{t-1,i} + (1-m)G_{t,i}^2$$

- ADAM: Combination of momentum and RMSProp, currently the most popular optimizer for NNs, along with AdamW (improved version where the weight decay is performed differently)

- Notes:
  - all these optimizers are coded in standard deep learning libraries
  - It’s hard to keep good intuitions with complex optimizer, if things don’t work/you are lost, go back to batch SGD

Slide: M. Aubry

Learning rate

Learning rate policy

- There is no standard policy with NNs.
- The best is usually to pick a LR as high as possible without having the algorithm diverge, keep it constant until convergence, then decrease it (and iterate that until there is no difference). Other more complex approaches are also widely used such as sinusoidal LR schedules, linear warmup etc.
- LR schedules even help with adaptive optimizers (e.g., AdaGrad)
- There are no guarantees with NNs, looking at training curves is crucial!

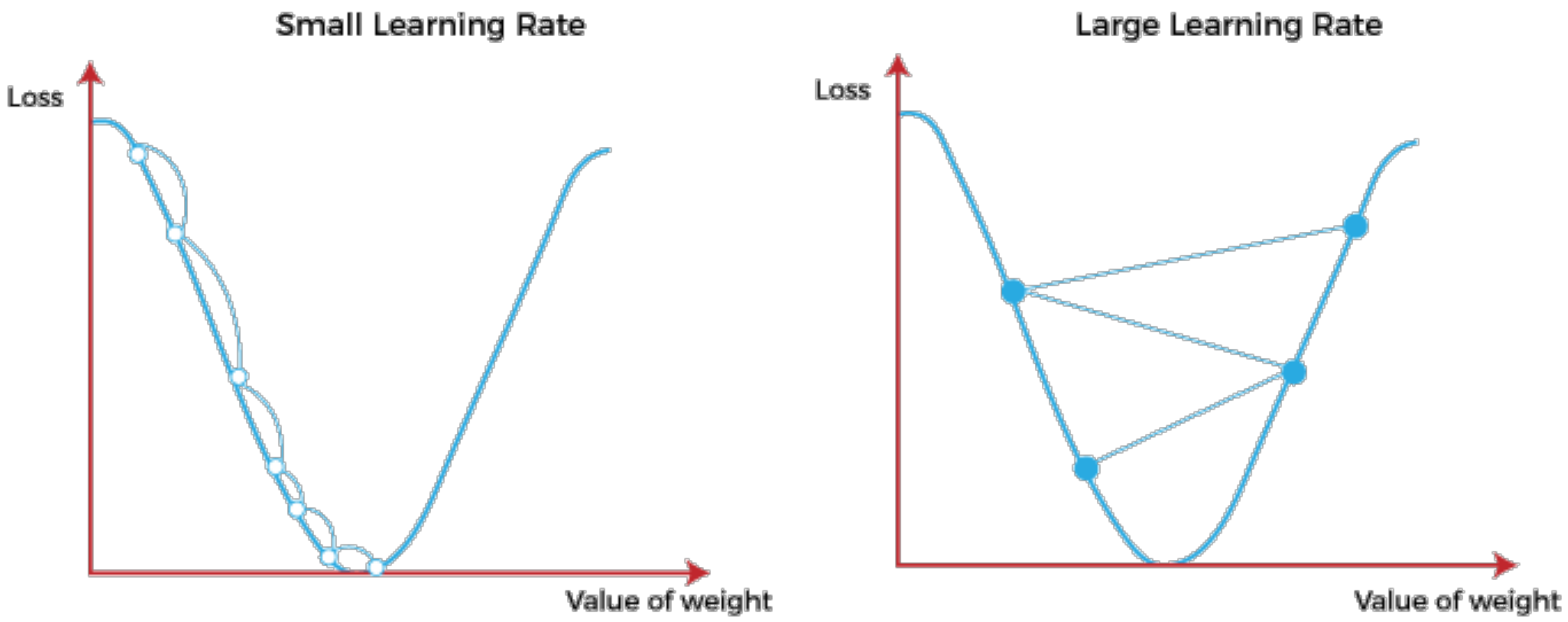
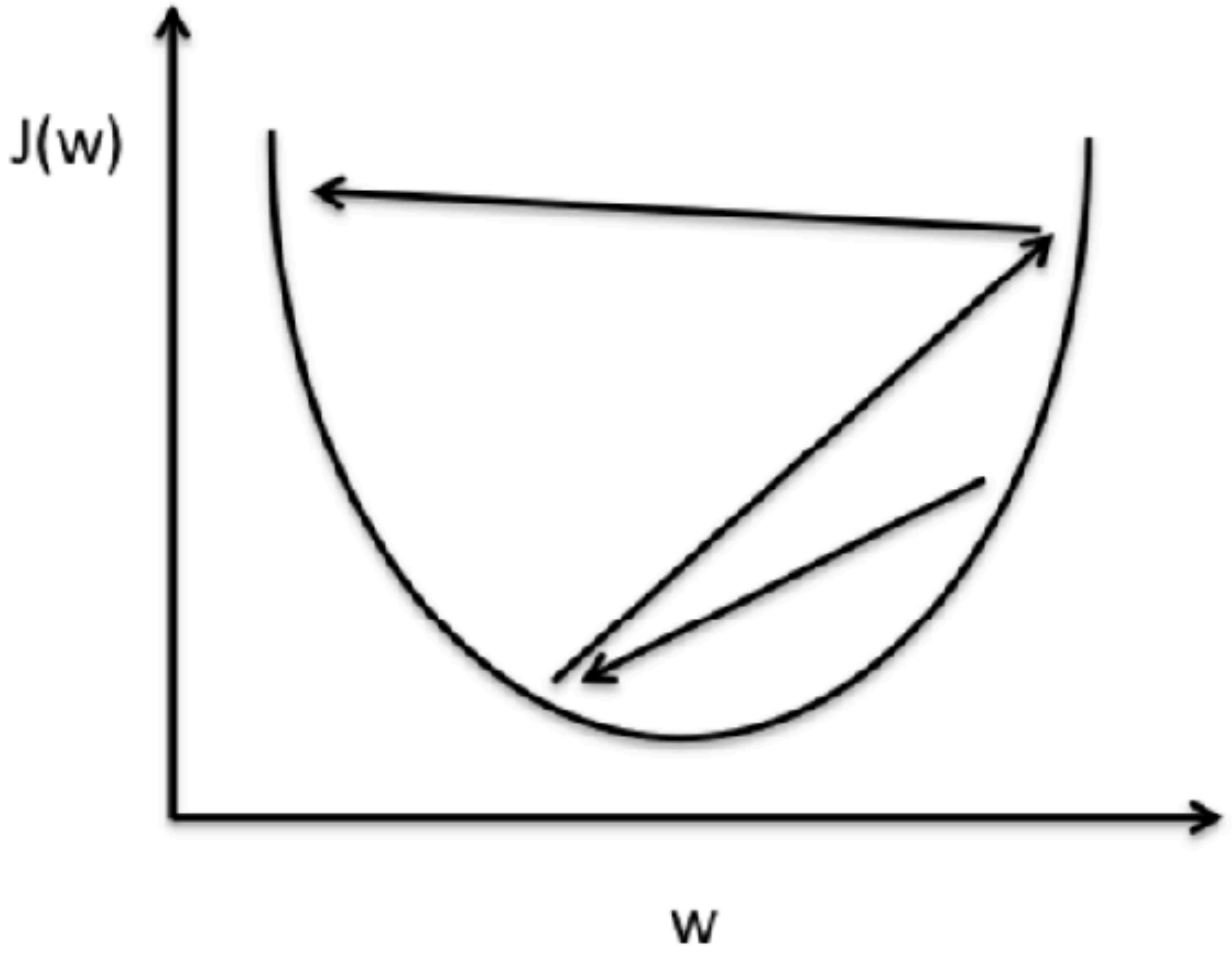
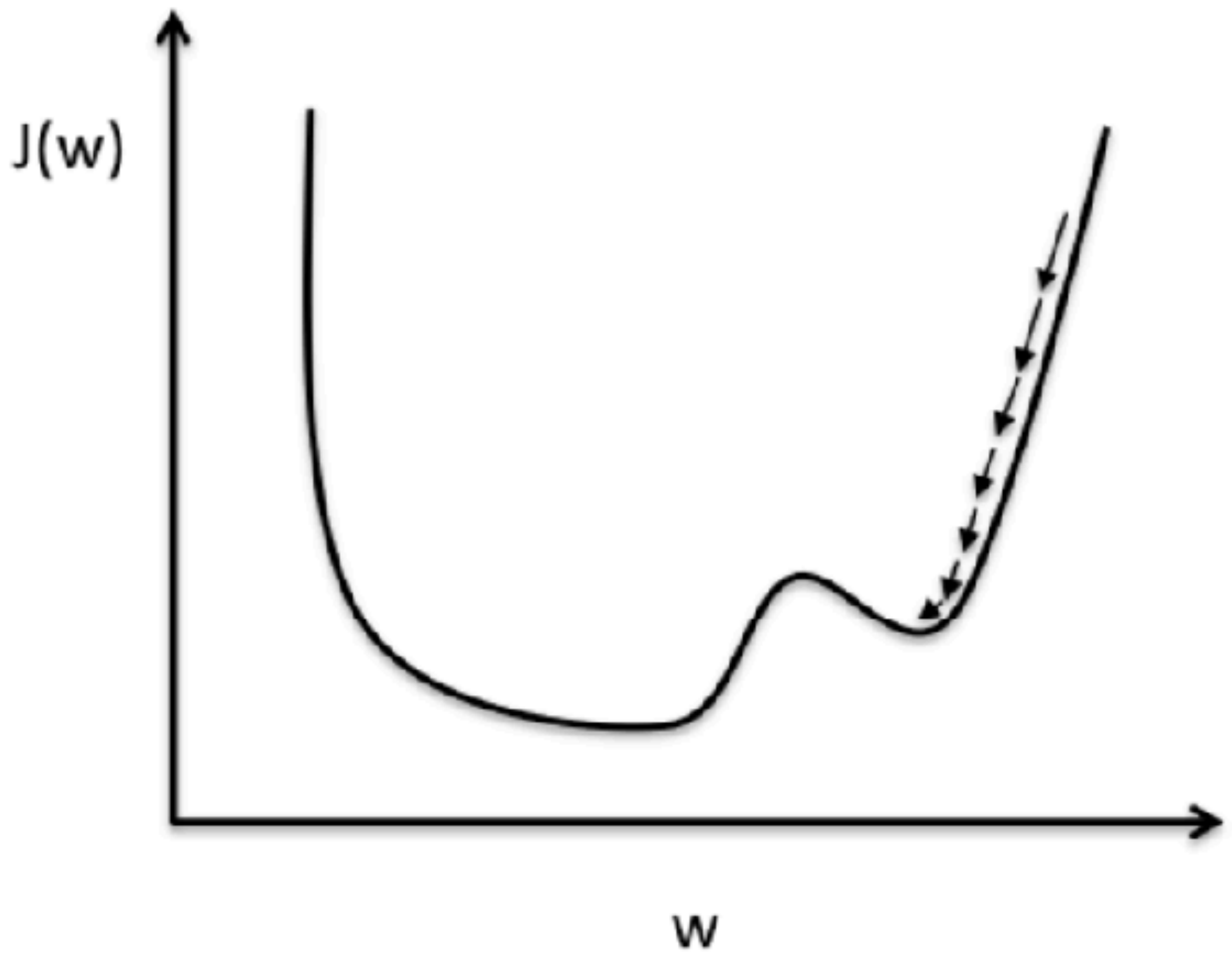


Image source

Slide: M. Aubry

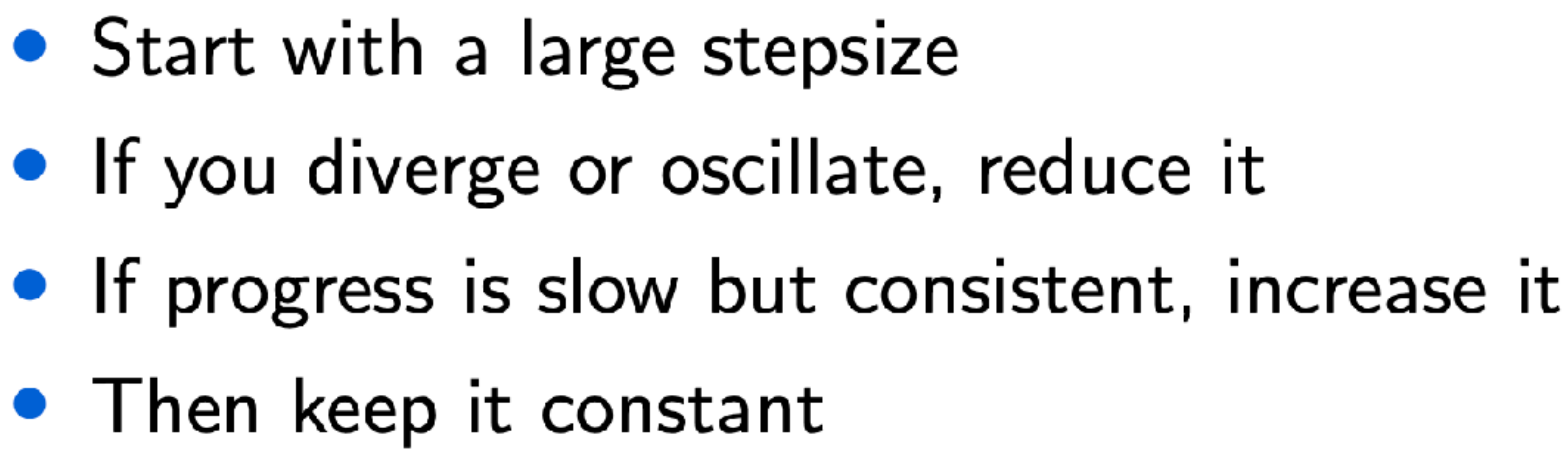


Large learning rate: Overshooting. Divergence.



Small learning rate: Many iterations until convergence and trapping in local minima.

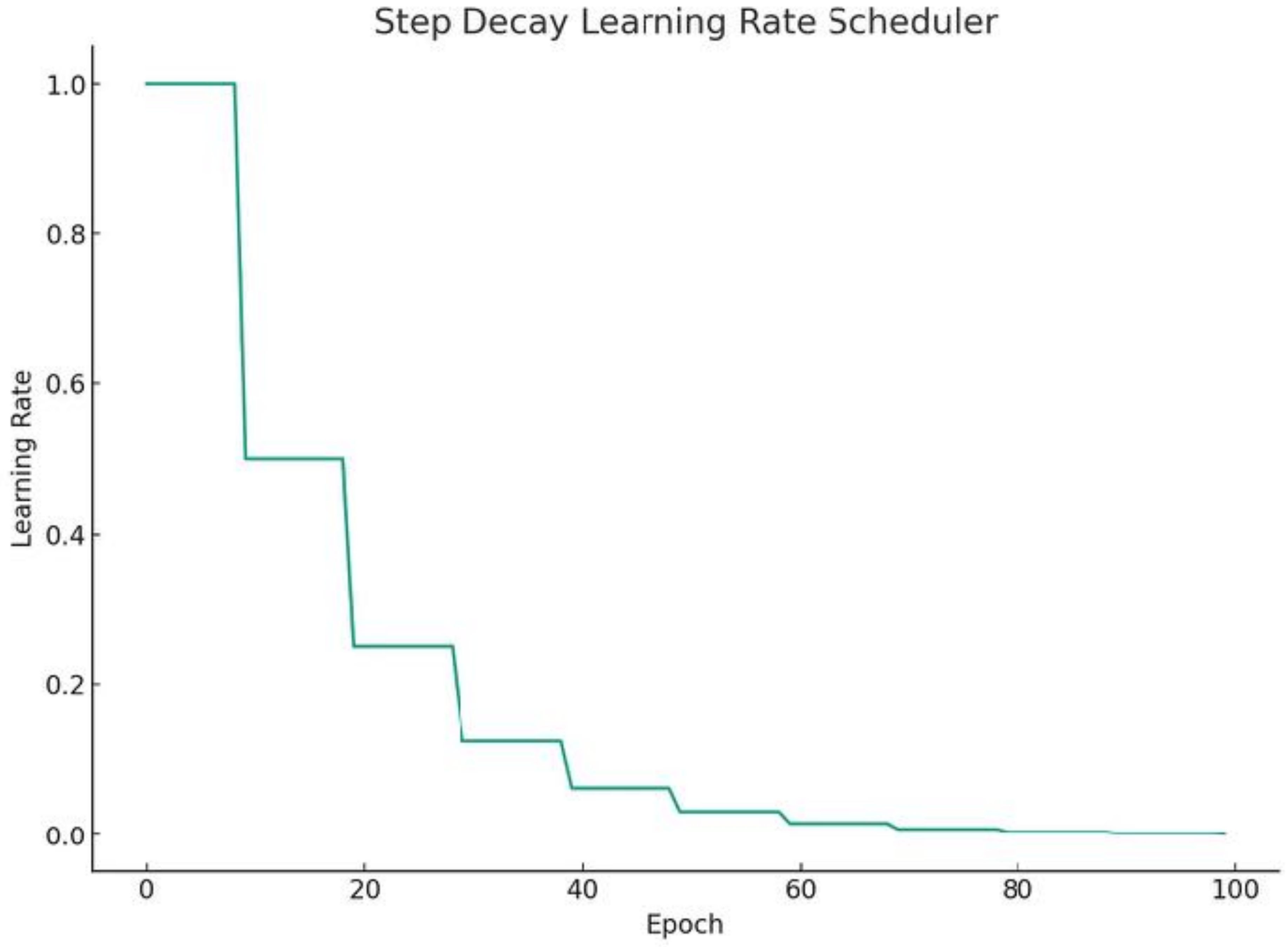
Fixed learning rate



- Start with a large stepsize
- If you diverge or oscillate, reduce it
- If progress is slow but consistent, increase it
- Then keep it constant

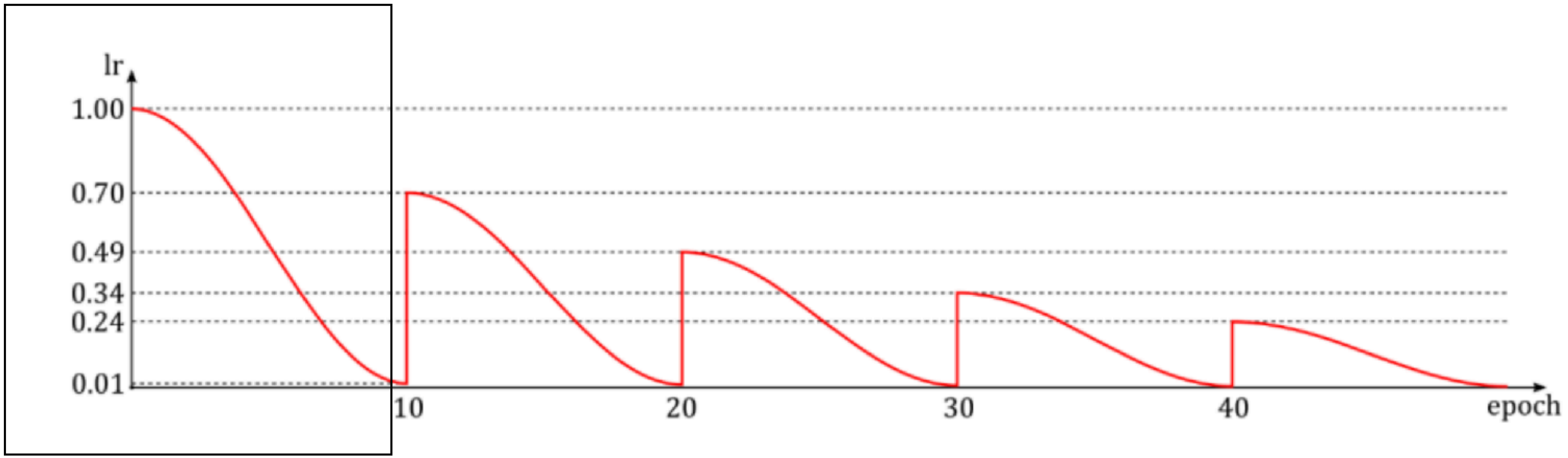
Linear decay, step decay

- Linear decay: $\alpha_t = a/(b+t)$
- Divide learning rate by a factor when loss on validation set does not decrease
- Fix number of iterations T and set learning rate accordingly: $\alpha_t = \alpha_0(T-t)/T$



Cosine scheduler

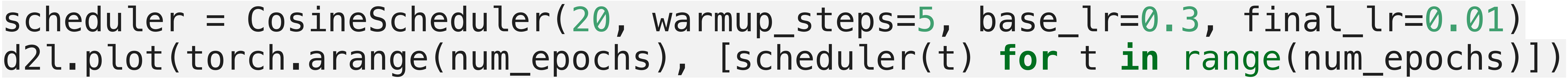
- Decay with a cosine function



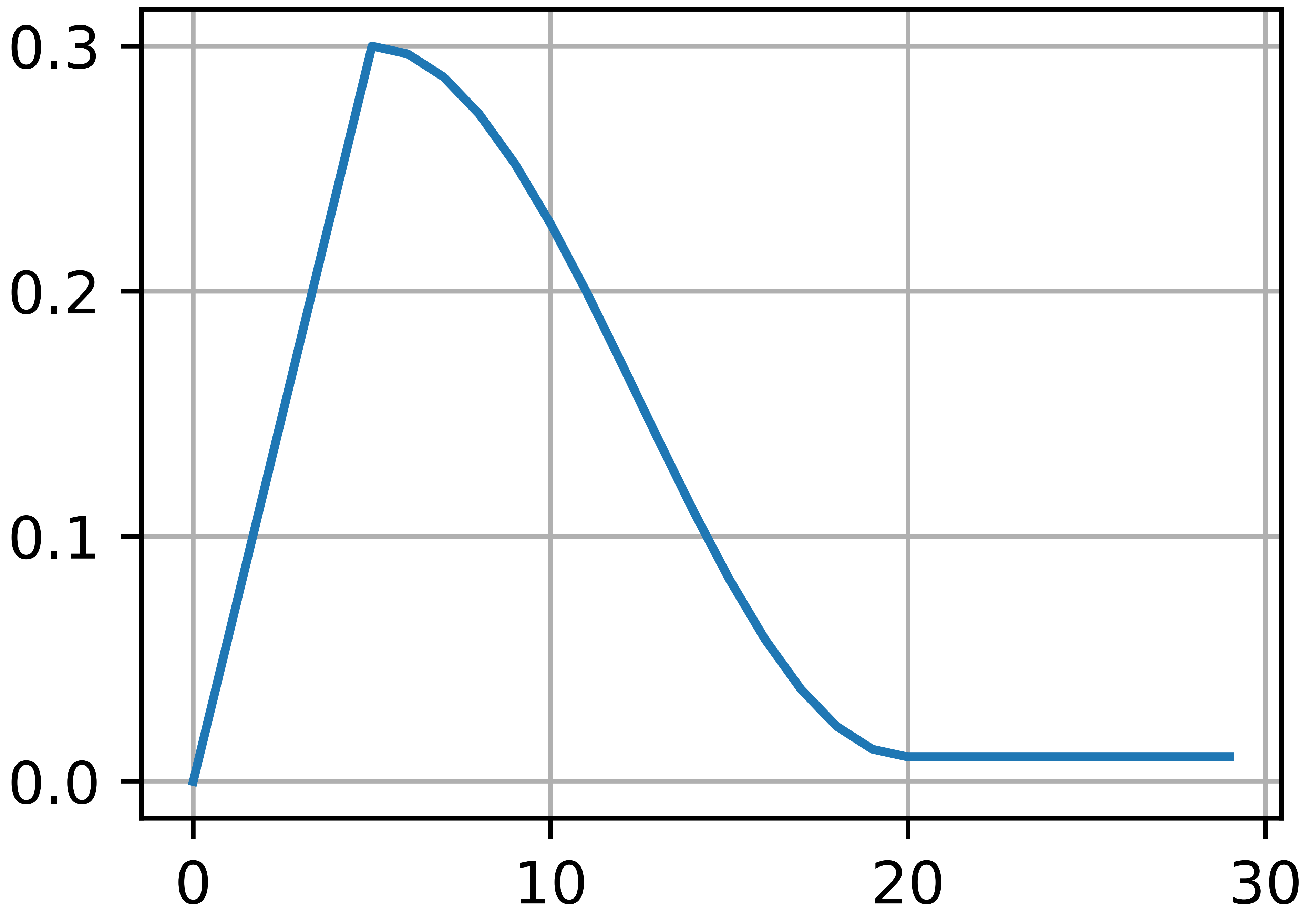
Loshchilov & Hutter, SGDR: Stochastic Gradient Descent with Warm Restarts, ICLR 2017

Linear “warmup”

- Large LR initially => divergence.
- A sufficiently small LR prevents divergence in the beginning. But, this means that progress is slow.
- A rather simple fix for this dilemma is to use a warmup period during which the LR *increases* to its initial maximum.



```
scheduler = CosineScheduler(20, warmup_steps=5, base_lr=0.3, final_lr=0.01)
d2l.plot(torch.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



Loshchilov & Hutter, SGDR: Stochastic Gradient Descent with Warm Restarts, ICLR 2017

Figure from https://d2l.ai/chapter_optimization/lr-scheduler.html

Initial learning rate

- For logistic regression, a good initialization is in the range {0.01, 0.05, 0.1, 0.3, 0.5, 1}

Magic (!) 0.03

- Rule of thumb: pick the LR that is "just below" the one where the network diverges.

* If you start seeing values like 1e-8, there might be something else to change, initialization, normalization etc.
** Multi-tasking (multiple loss terms) is tricky to set a single learning rate for all tasks.

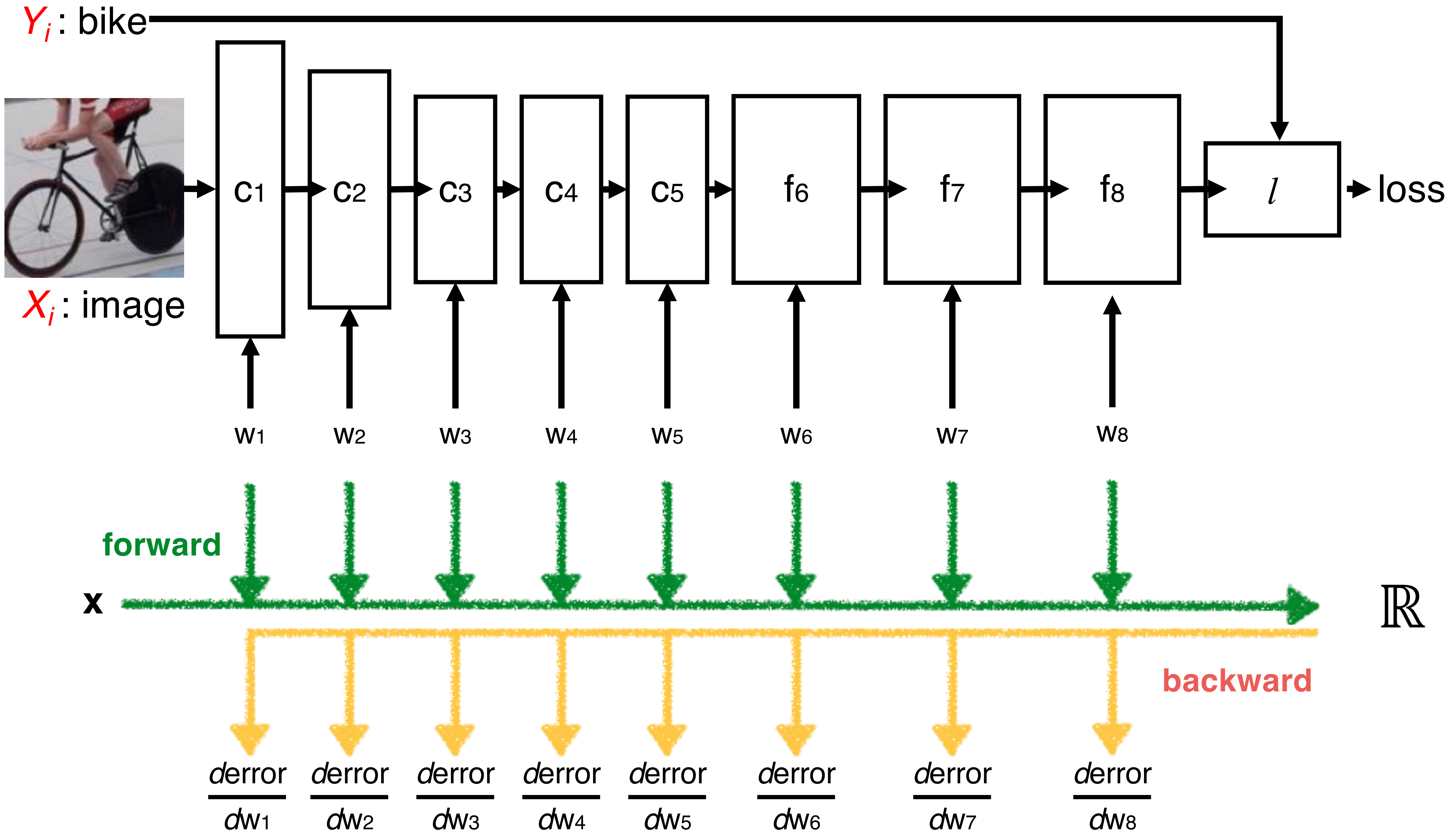
Computing the gradients

- Deep neural networks include a lot of operations, so it’s important to compute the gradient efficiently.
- It helps to leverage GPUs by computing the gradient for several inputs, performing batch SGD
- While in theory, we just have the gradients of composite functions and for that apply **chain rule**, there is an **efficient** way to do it, called **back-propagation**.

Slide: M. Aubry

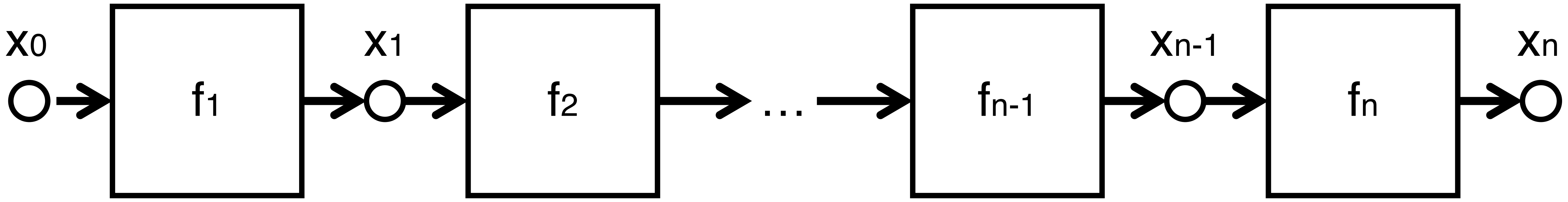
Backpropagation

Computing the gradients: While in theory, we just have the gradients of composite functions and for that apply **chain rule**, there is an **efficient** way to do it, called **backpropagation**.



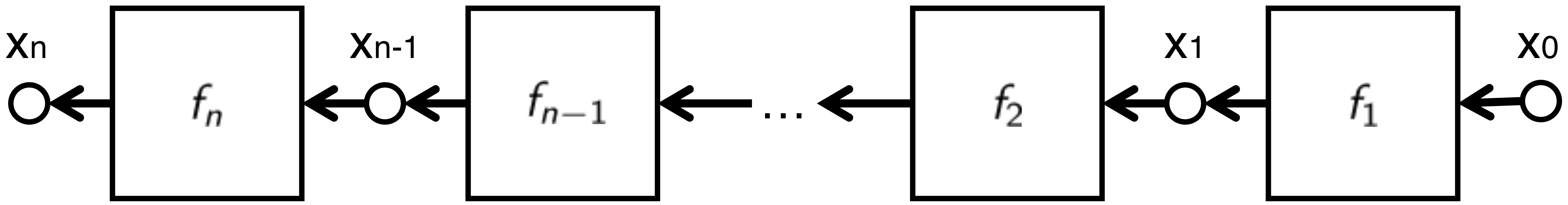
Slide credit: Andrea Vedaldi [Derivatives, Backpropagation, and Vectorization] http://cs231n.stanford.edu/handouts/derivatives.pdf

Chain rule: scalar version



Slide credit: Andrea Vedaldi

Chain rule: scalar version



A composition of *n* functions

$$\mathbf{x}_n = (f_n \circ f_{n-1} \circ \ldots \circ f_2 \circ f_1) (\mathbf{x}_0)$$

$$\frac{dx_n}{dx_0} = \frac{df_n}{dx_{n-1}} \times \frac{df_{n-1}}{dx_{n-2}} \times \ldots \times \frac{df_2}{dx_1} \times \frac{df_1}{dx_0}$$

Derivative obtained using the chain rule

Slide credit: Andrea Vedaldi

Backpropagation

Derivatives:

- Scalar case
- Gradient: Vector in, Scalar out
- Jacobian: Vector in, Vector out
- Generalized Jacobian: Tensor in, Tensor out

Impractical to store in memory

e.g., for a fully connected layer that takes as input a minibatch of *N* vectors, each dimension *D*, and produces a minibatch of *N* vectors, each dimension *M*:

=> *Jacobian matrix (N x M) x (N x D)*

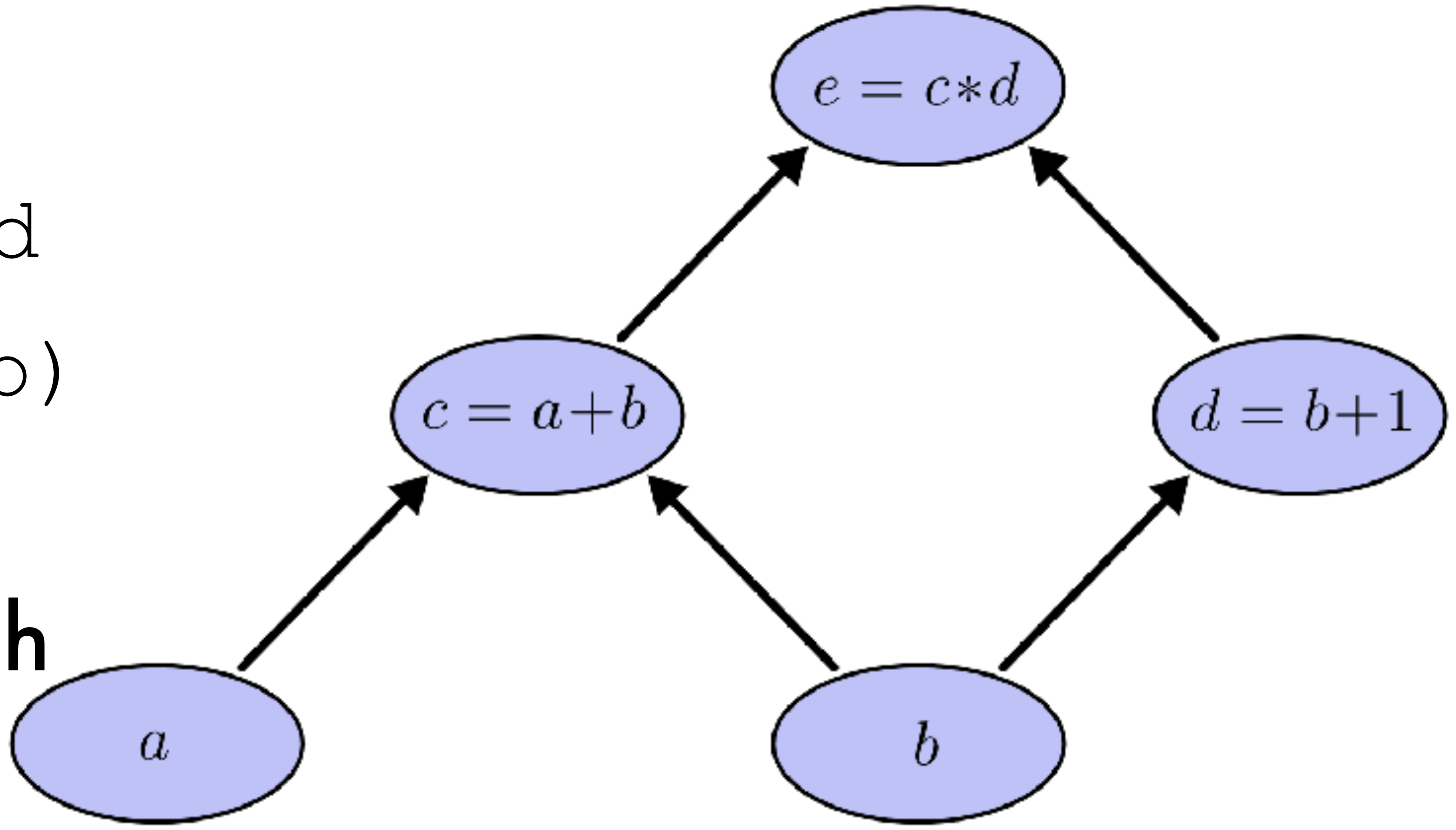
=> *68 billion numbers (256GB) if N=64, M=D=4096*

Chain rule: start from the loss which is a scalar, no explicit forming of the entire Jacobian

[Derivatives, Backpropagation, and Vectorization] http://cs231n.stanford.edu/handouts/derivatives.pdf

Computational Graphs

- Say we have some function `e(c, d)`, but `c` and `d` are functions of other variables. We have `c(a, b)` and `d(b)`
- We can write how these functions depend on each other as a tree
- We call this a **computational graph** because it tells us how to compute the final value `e` from leaf nodes (inputs) `a` and `b`
- <u>**Each node in this tree is a function**</u> of the incoming nodes

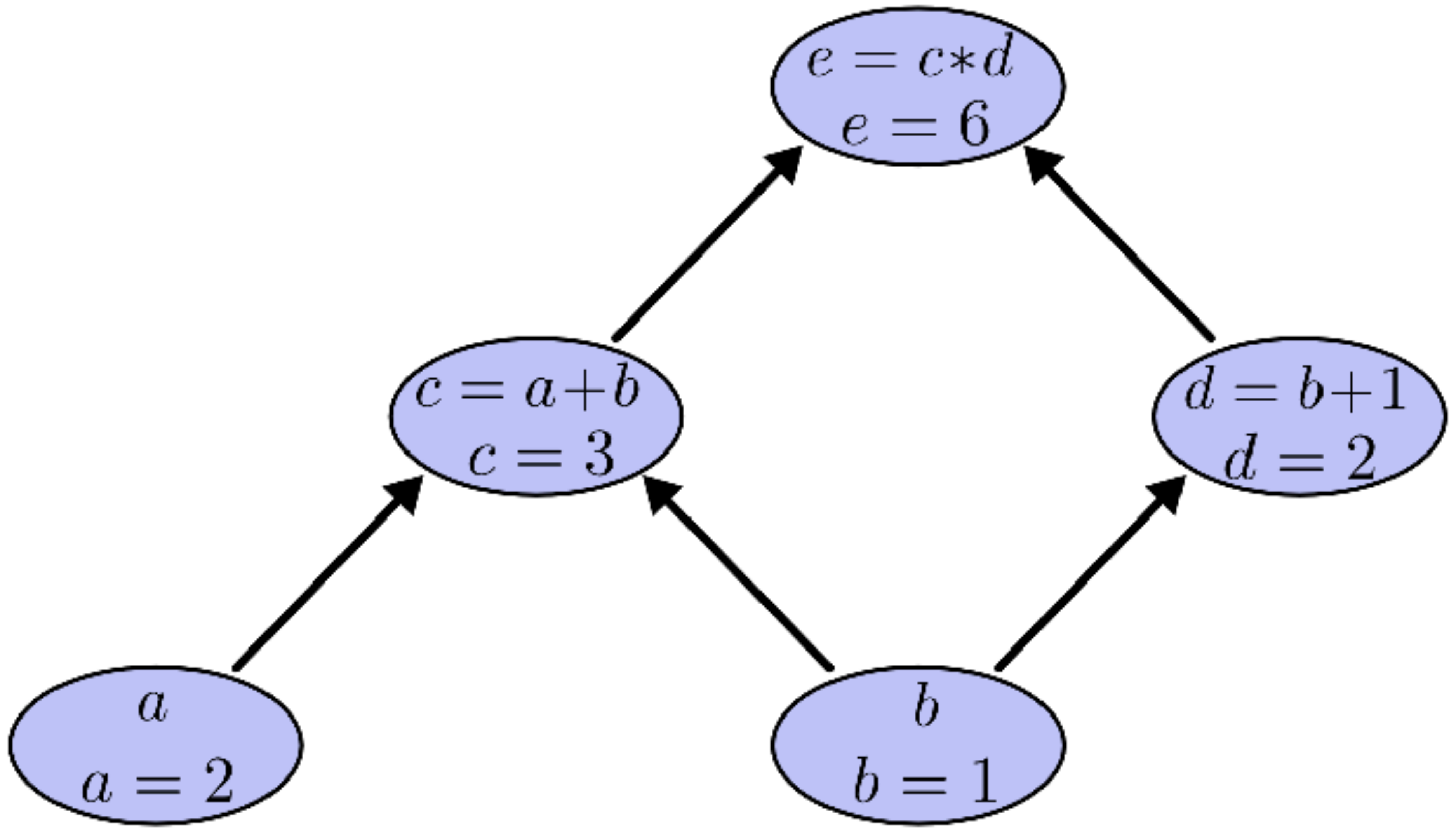


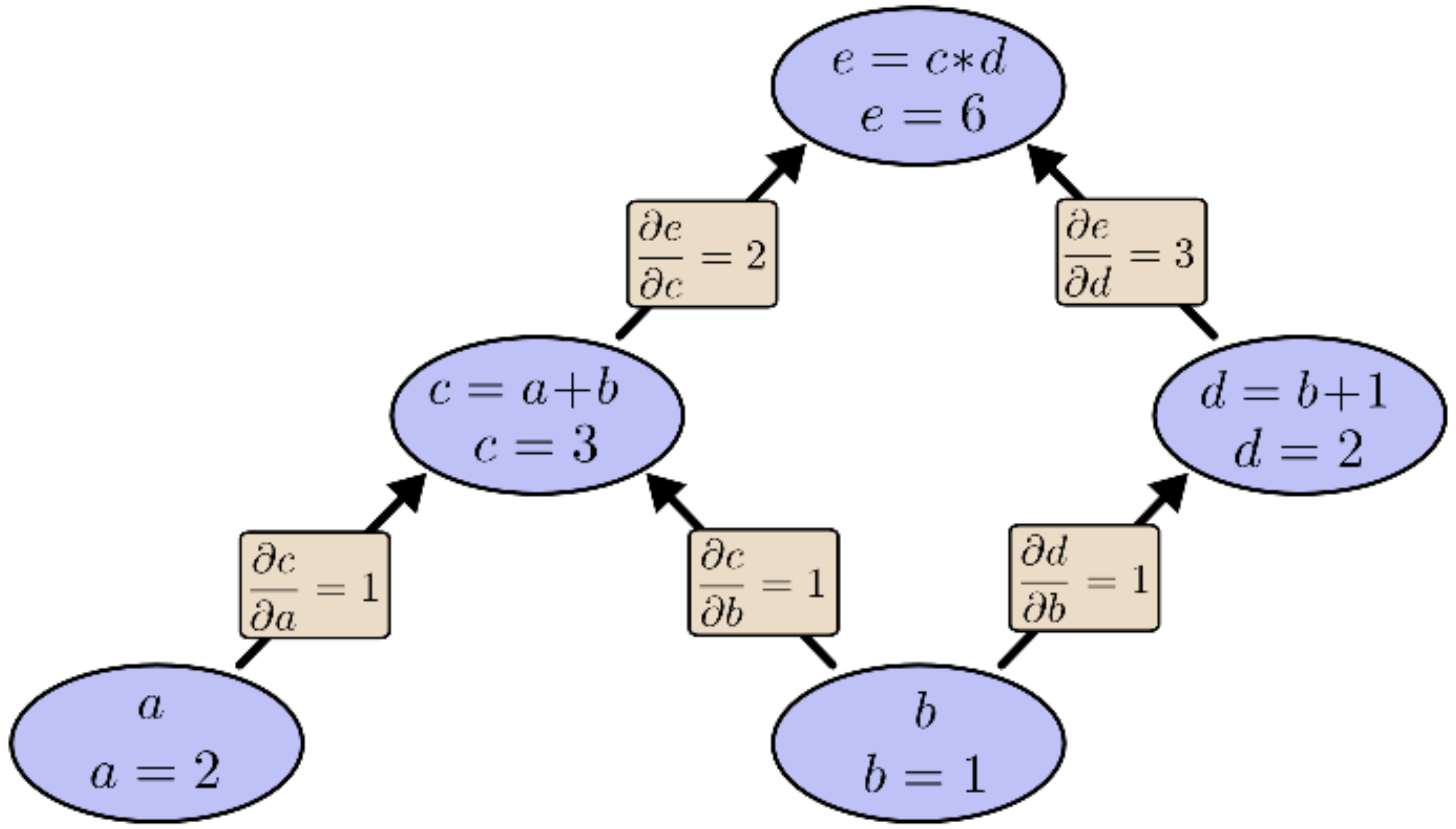
Slide credit: J. Austin, H. Jalan, A. Jain

Computational Graphs and the Chain Rule

- If we want to calculate derivatives of an input with respect to the output, we need to use the multivariable chain rule
  - Sum over all **unique** paths from the input to the output
  - For each path, multiply all partial derivatives of each output node with respect to the corresponding input node

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c}\frac{\partial c}{\partial b} + \frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$





$e = c*d$
$e = 6$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$c = a+b$
$c = 3$
$d = b+1$
$d = 2$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
a
$a = 2$
b
$b = 1$

Slide credit: J. Austin, H. Jalan, A. Jain

Backpropagation

- Here is an example of a computational graph of a toy neural network's MSE loss on a single training example
  - This neural network has **only one neuron per layer, making inputs and outputs <u>scalars</u>**
- Our objective with gradient descent is to **calculate the partial derivative of the output with respect to w1, w2, w3, b1, b2 and b3**... but we don't want to do 6x the computation... how can we do this?

Z1
A1
Z2
A2
Z3
L
X
(W1*X)+B1
ReLU
(W2*A1)+B2
ReLU
(W3*A2)+B3
-
^2
W1
B1
W2
B2
W3
B3
Y

Slide credit: J. Austin, H. Jalan, A. Jain

Backpropagation

- We can write out the chain rule for all these, and see if there is anywhere that we can optimize and save ourselves some compute
- Note: We will be writing out a lot of partial derivatives… **<u>each one is being evaluated for under the current training example and the current parameters</u>**

Z1
A1
Z2
A2
Z3
L
X
(W1*X)+B1
ReLU
(W2*A1)+B2
ReLU
(W3*A2)+B3
-
^2
W1
B1
W2
B2
W3
B3
Y

Slide credit: J. Austin, H. Jalan, A. Jain

- In this example, x, y, b1, b2, b3, w1, w2, w3 are scalars

Side note: on the forward pass, we calculate and save things like the partial of z_3 with respect to a_3, so that we can use it here later

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial z_3}\frac{\partial z_3}{\partial w_3} \quad \frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial z_3}\frac{\partial z_3}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial w_2} \quad \frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z_3}\frac{\partial z_3}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial w_1}$$

Z1
A1
Z2
A2
Z3
L
X
(W1*X)+B1
ReLU
(W2*A1)+B2
ReLU
(W3*A2)+B3
-
^2
W1
B1
W2
B2
W3
B3
Y

Scalar Computation Graph Example

Slide credit: J. Austin, H. Jalan, A. Jain

- In this example, x, y, b1, b2, b3, w1, w2, w3 are scalars

We can see that we’ve calculated these values multiple times

$$\frac{\partial L}{\partial w_3} = \boxed{\frac{\partial L}{\partial z_3}}\frac{\partial z_3}{\partial w_3} \qquad \frac{\partial L}{\partial w_2} = \boxed{\boxed{\frac{\partial L}{\partial z_3}}\frac{\partial z_3}{\partial a_2}\frac{\partial a_2}{\partial z_2}}\frac{\partial z_2}{\partial w_2} \qquad \frac{\partial L}{\partial w_1} = \boxed{\frac{\partial L}{\partial z_3}\frac{\partial z_3}{\partial a_2}\frac{\partial a_2}{\partial z_2}}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial w_1}$$

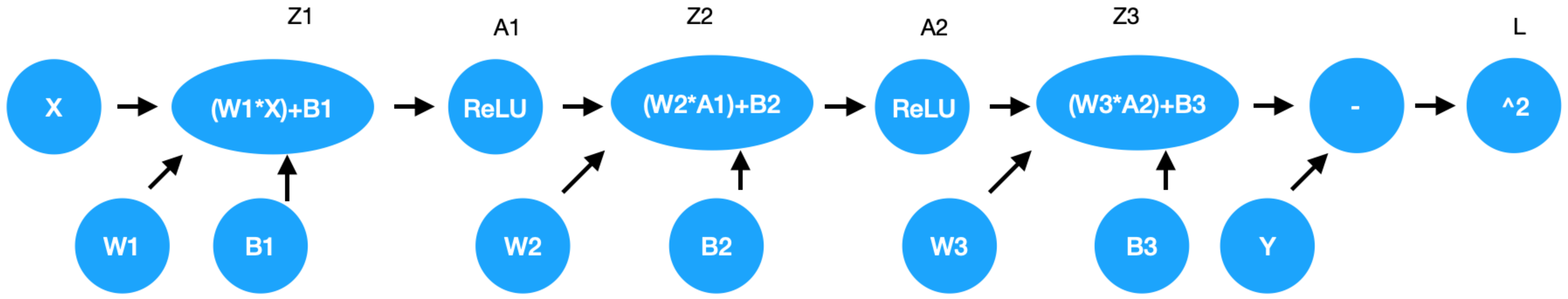
Z1
A1
Z2
A2
Z3
L
X
(W1*X)+B1
ReLU
(W2*A1)+B2
ReLU
(W3*A2)+B3
-
^2
W1
B1
W2
B2
W3
B3
Y

Scalar Computation Graph Example

Slide credit: J. Austin, H. Jalan, A. Jain

Backpropagation

$$\frac{\partial L}{\partial w_3} = \boxed{\frac{\partial L}{\partial z_3}}\frac{\partial z_3}{\partial w_3} \quad \frac{\partial L}{\partial w_2} = \boxed{\boxed{\frac{\partial L}{\partial z_3}}\frac{\partial z_3}{\partial a_2}\frac{\partial a_2}{\partial z_2}}\frac{\partial z_2}{\partial w_2} \quad \frac{\partial L}{\partial w_1} = \boxed{\frac{\partial L}{\partial z_3}\frac{\partial z_3}{\partial a_2}\frac{\partial a_2}{\partial z_2}}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial w_1}$$



- Rather than calculating these values again with repeated multiplication, let's just save and reuse them
  - **<u>This saves a lot of redundant calculations</u>** for deep neural networks
- We will simply work from the **end of the network to the front**, caching values that we need as we go along
- Note: All the partials here are being evaluated for the current data and parameters...

1) First we calculate the update for W3, caching the red
2) Then we use the red to calculate the blue value before calculating the update for W2
3) This pattern of using the last computation to save redundant multiplications on the next update continues

Slide credit: J. Austin, H. Jalan, A. Jain

Backpropagation: Takeaways

- The thing you need to take away from backprop is that **it is a fast method of getting all of the partial derivatives needed for gradient descent, removing redundant (matrix) multiplications**
  - We do this by working from the end of the computational graph to the front, caching any computation used in calculating the previous partial derivatives
  - By working from the end of the graph to the front, we can handle much more complex computational graphs quickly and efficiently
- Modern auto-differentiation software like **pytorch** will keep track of the graph and calculate our gradients with backprop
  - It can handle arbitrarily large computational graphs

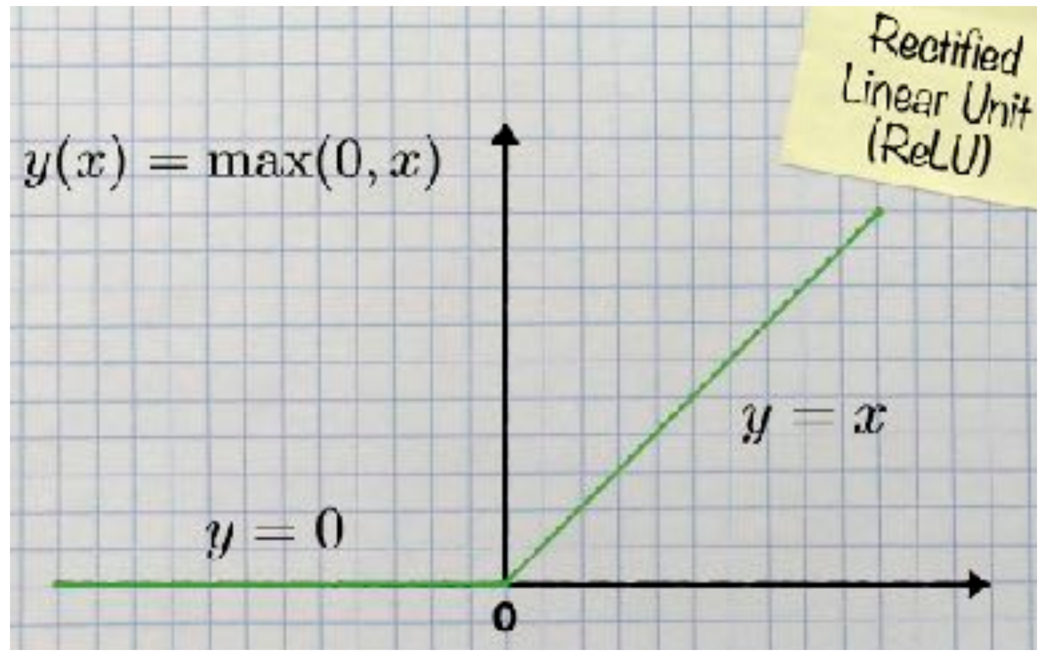
Slide credit: J. Austin, H. Jalan, A. Jain

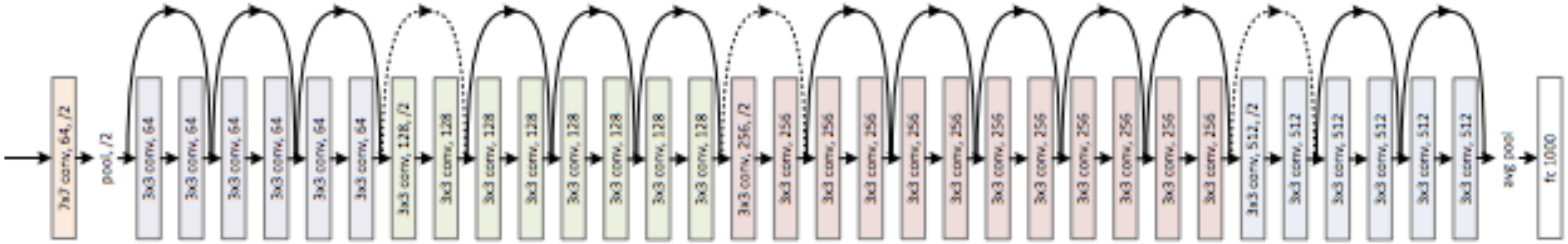
Vanishing / exploding gradients

Multiplying too many small/big values.

Solutions:

- Initialization techniques
- Use ReLU (non saturating)
- Use skip connections in the network





- Use batch normalization
- Use gradient clipping
- Use warmup LR scheduler

Slide: M. Aubry

Batch normalization

- A layer to try avoiding vanishing or exploding signal,

- Idea: normalize the data everywhere in the network using estimates of the mean/variance

$$BN_{\alpha,\beta}(x,\mu,\sigma) = \alpha\frac{x-\mu}{\sigma} + \beta$$

- α, β are learnt, μ, σ are estimated, all have the same dimension as x
- μ, σ are estimated over a mini-batch, or updated using a momentum
- Batch-norms are typically place just before non-linearities

- Careful: often source of bugs!
- Different behavior during training and testing: μ, σ are estimated on one batch during training, stable estimates are estimated with momentum and fixed during test (network in train/test modes)
- Requires large and diverse batches

Slide: M. Aubry

Gradient clipping

Avoid gradient explosion by clipping the value of gradient below some norm:

$$G = \min(\mu, \|G\|)\frac{G}{\|G\|}$$

with $\mu > 0$

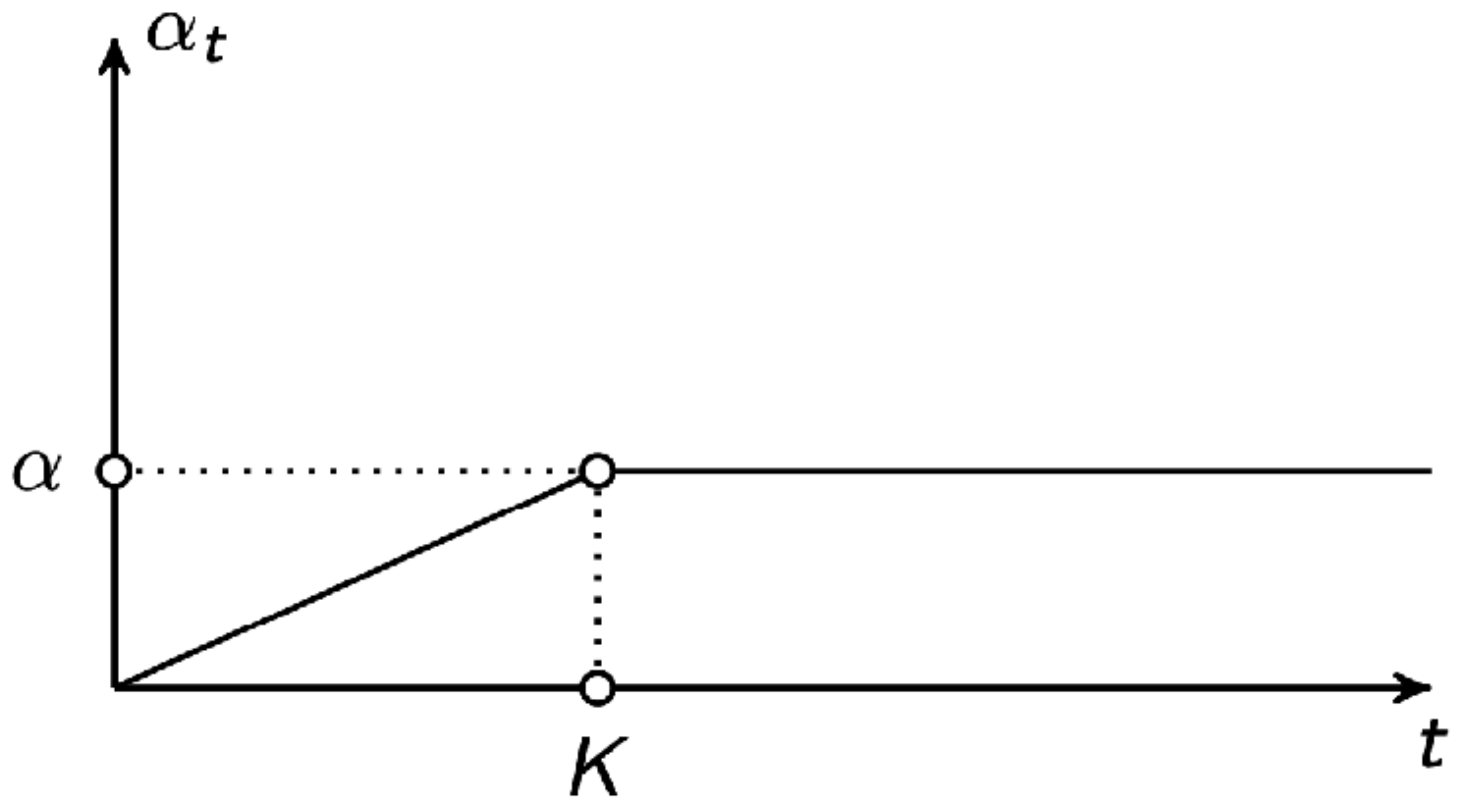
Warm-up

- Most gradient explosion happens at the beginning of training

- Because matrices are poorly set and learning rates are large

- Solution: start with small learning and increase it

Warm-up



Learning rate scheduler $(\alpha_t)_t$

- Set a target learning rate α

$$\alpha_t = \min(1,\ \frac{t}{K})\alpha$$

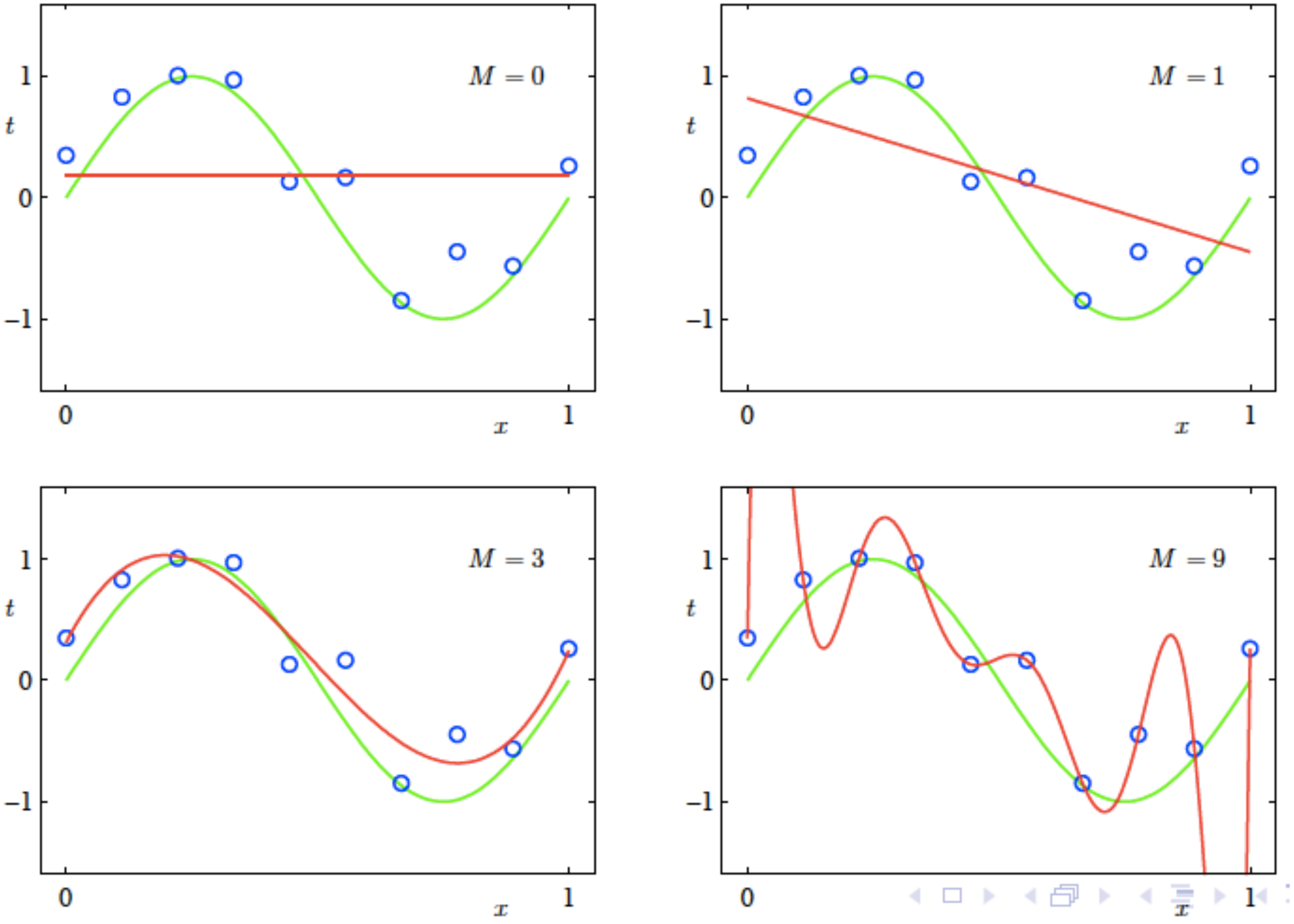
where K is the “warm-up” parameter

“Problems” with training

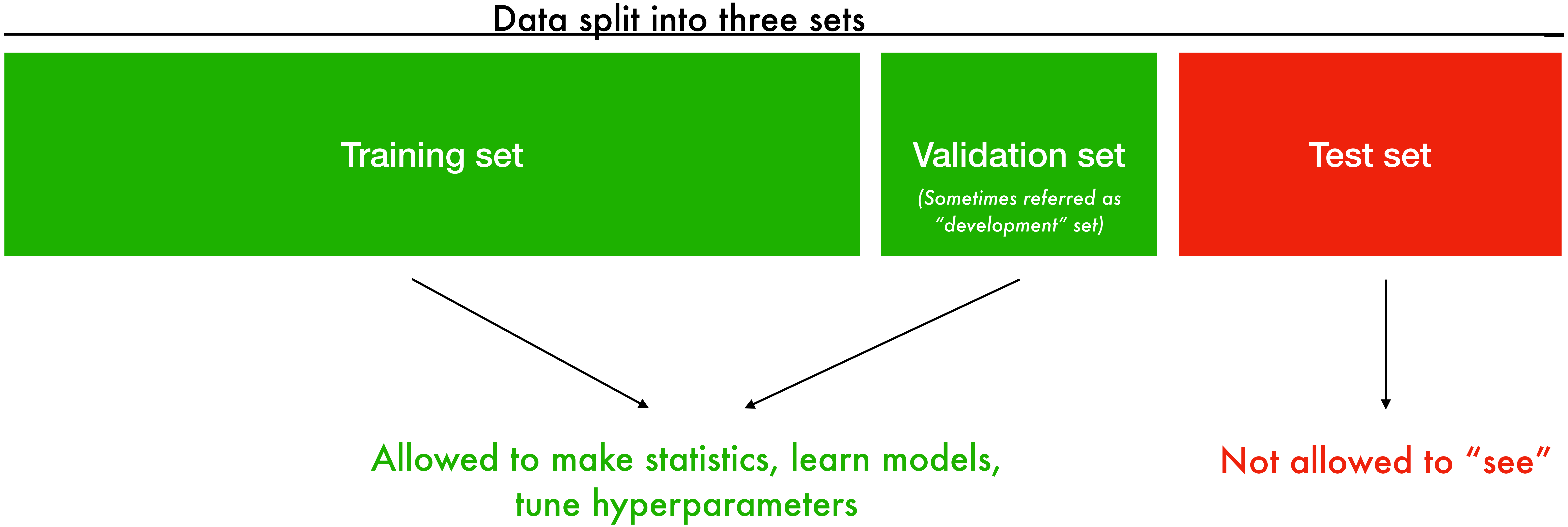
- Underfitting:
  - making poor predictions on the training (and test) data
  - not enough parameters to express complexity in data

- Overfitting:
  - too many parameters match too well complexity in training data
  - not generalizing to unseen data, i.e., high performance on training set, low on test set

Example: polynomial regression of degree M



"Typical" machine learning setup

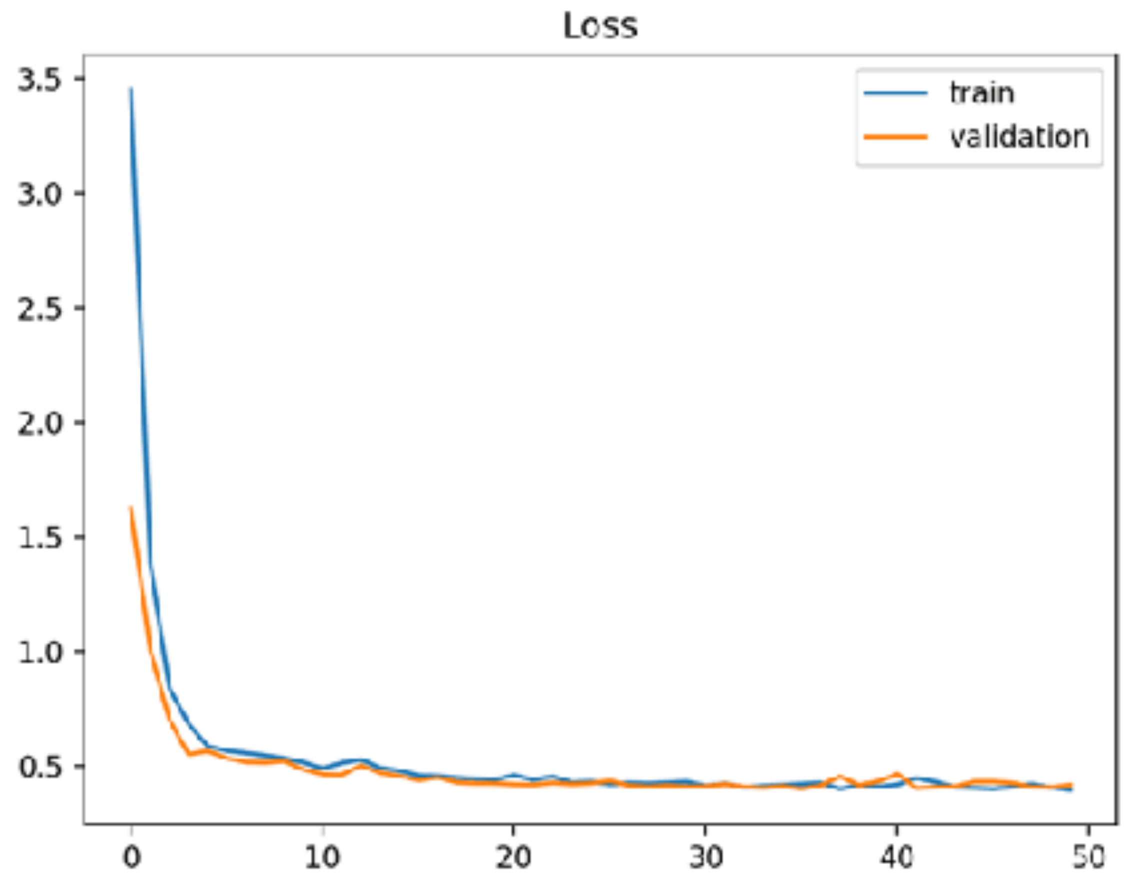

Data split into three sets
Training set
Validation set
(Sometimes referred as "development" set)
Test set
Allowed to make statistics, learn models, tune hyperparameters
Not allowed to "see"

- Learn models on the **training** set
- Evaluate on the **validation** set many times (run experiments to find good hyperparameters, e.g., number of epochs, learning rate, batch size...)
- (Optional: Learn the final model on the combination of **training and validation** sets)
- Evaluate on the **test** set "once"

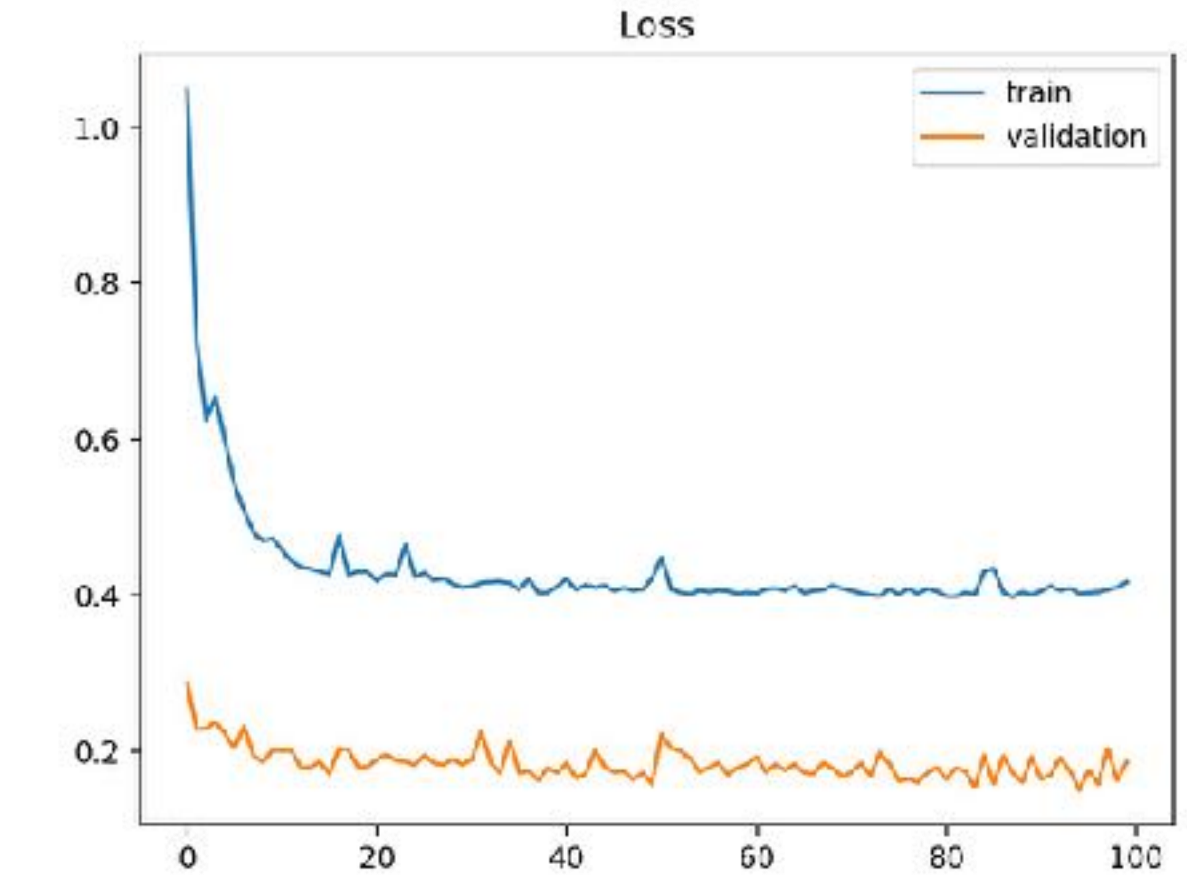
A few possible scenarios for learning curves

train
validation

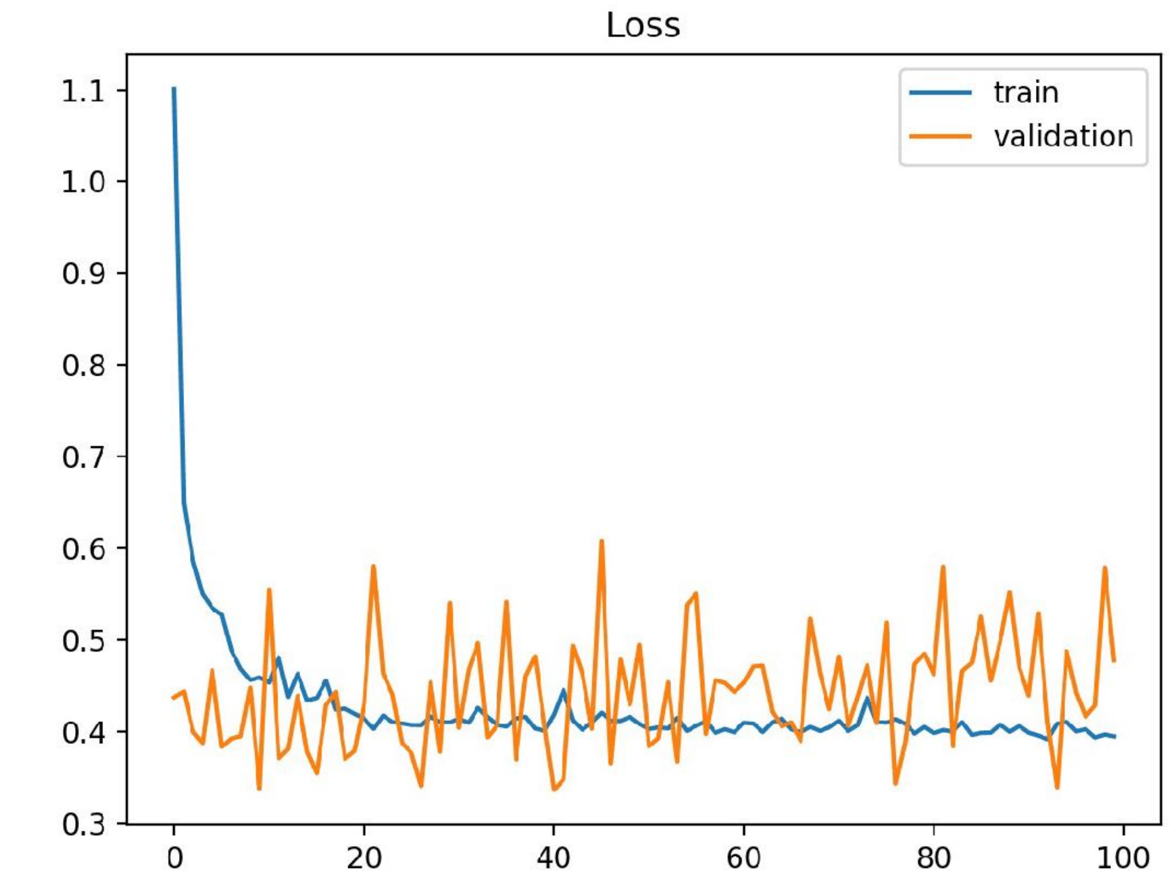
Good fit: both decreasing, converging, minimal gap



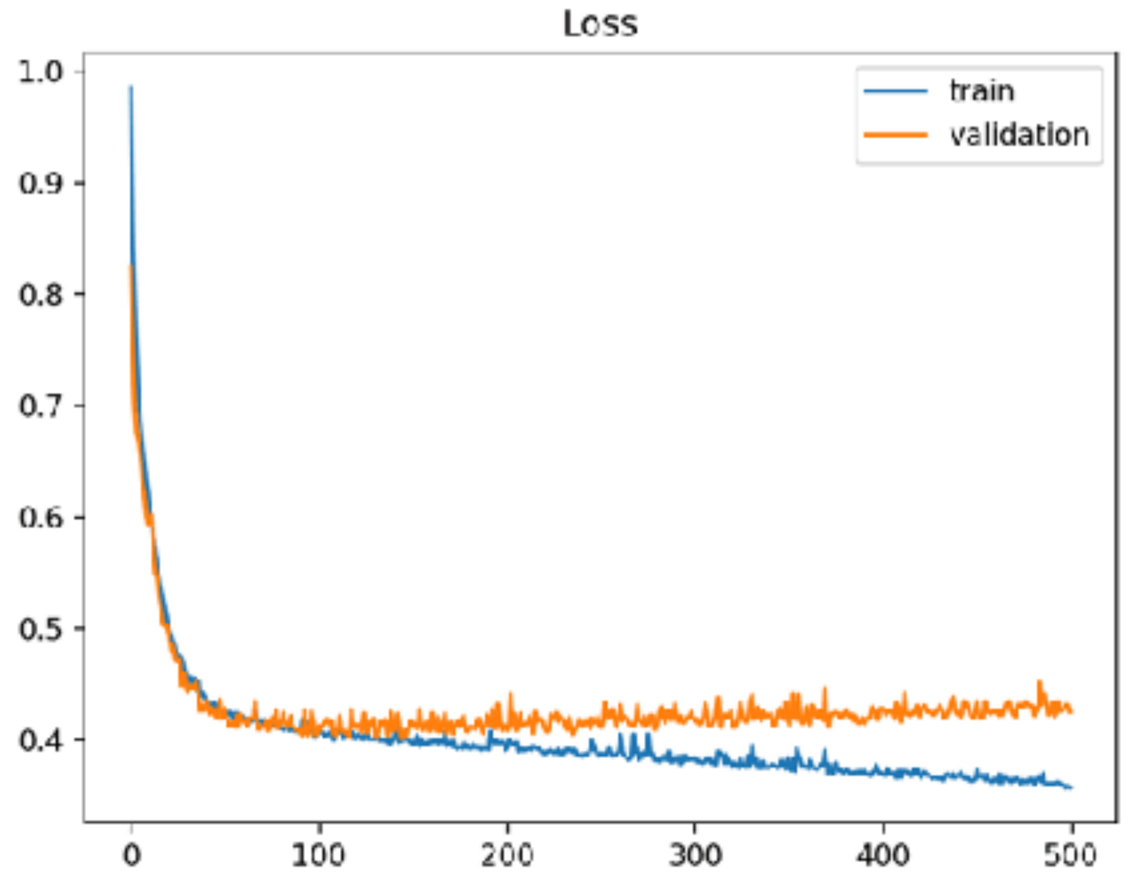
Unrepresentative validation set: easier than training set



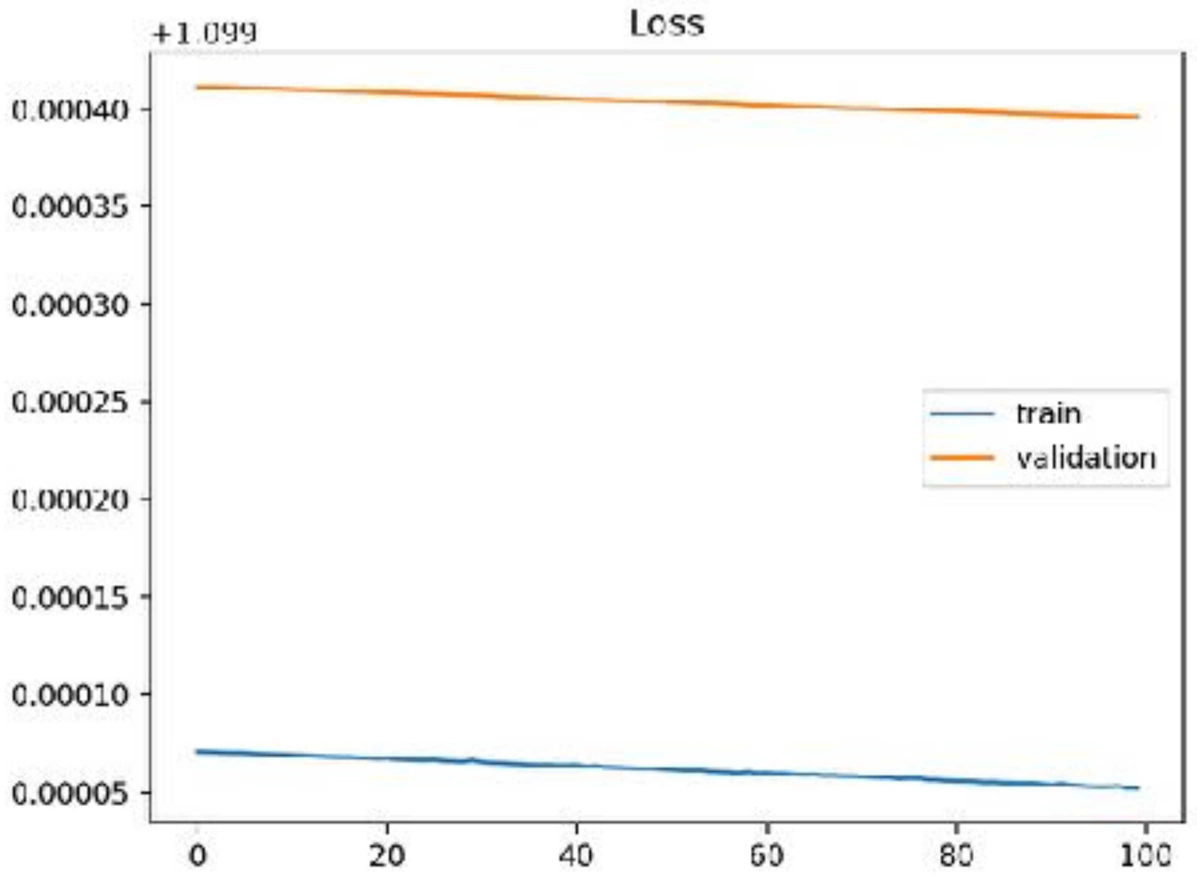
Unrepresentative validation set: too few examples



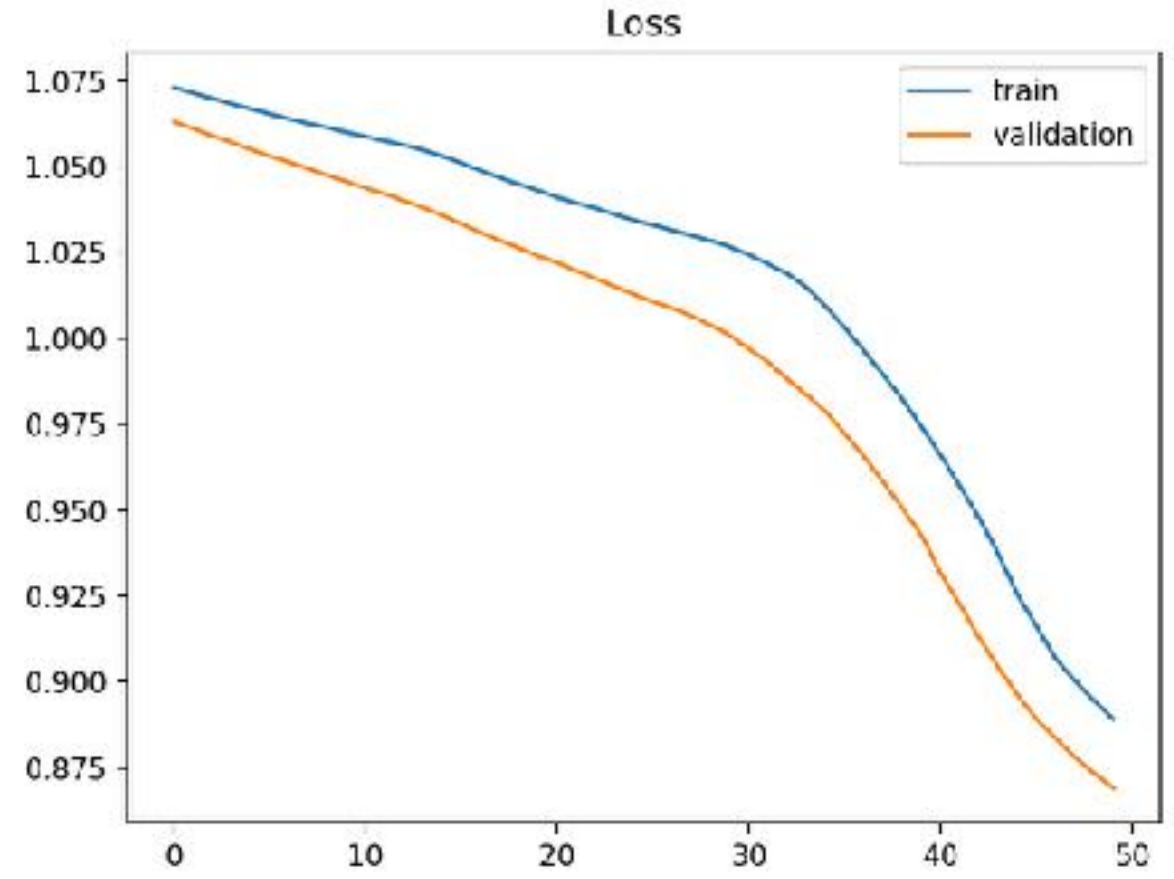
Overfit: validation increasing



Underfit: training loss not decreasing



Underfit: training halted prematurely



loss

number of iterations/epochs

Credit: **Jason Brownlee**

Image sources

How to avoid overfitting?

Deep networks have many parameters.

Some regularization techniques:

- Smaller network, i.e., less parameters
- Data augmentation
- Suboptimize, i.e., “early stopping”
- Force redundancy in hidden units, i.e., “dropout”
- Penalize parameter norms, i.e., “weight decay”

Data augmentation

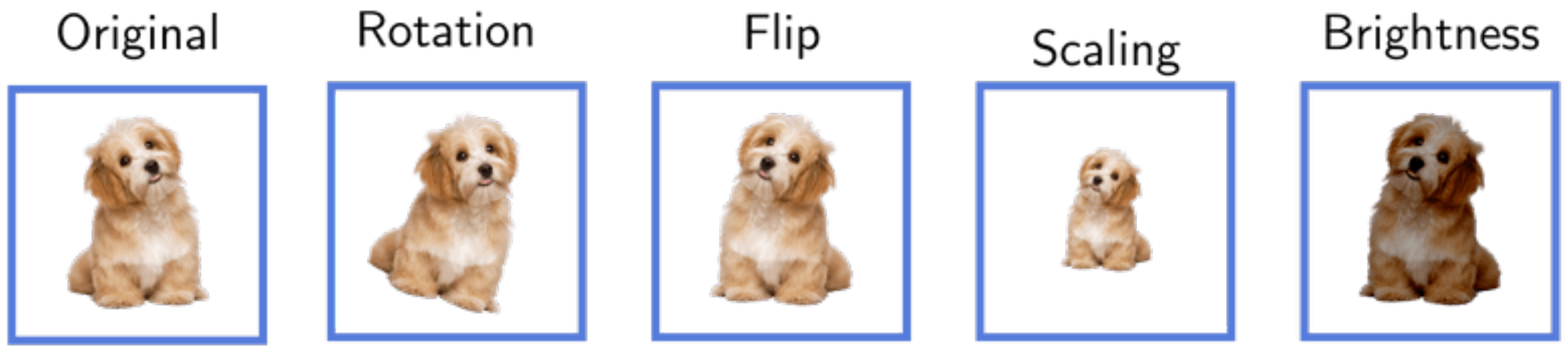
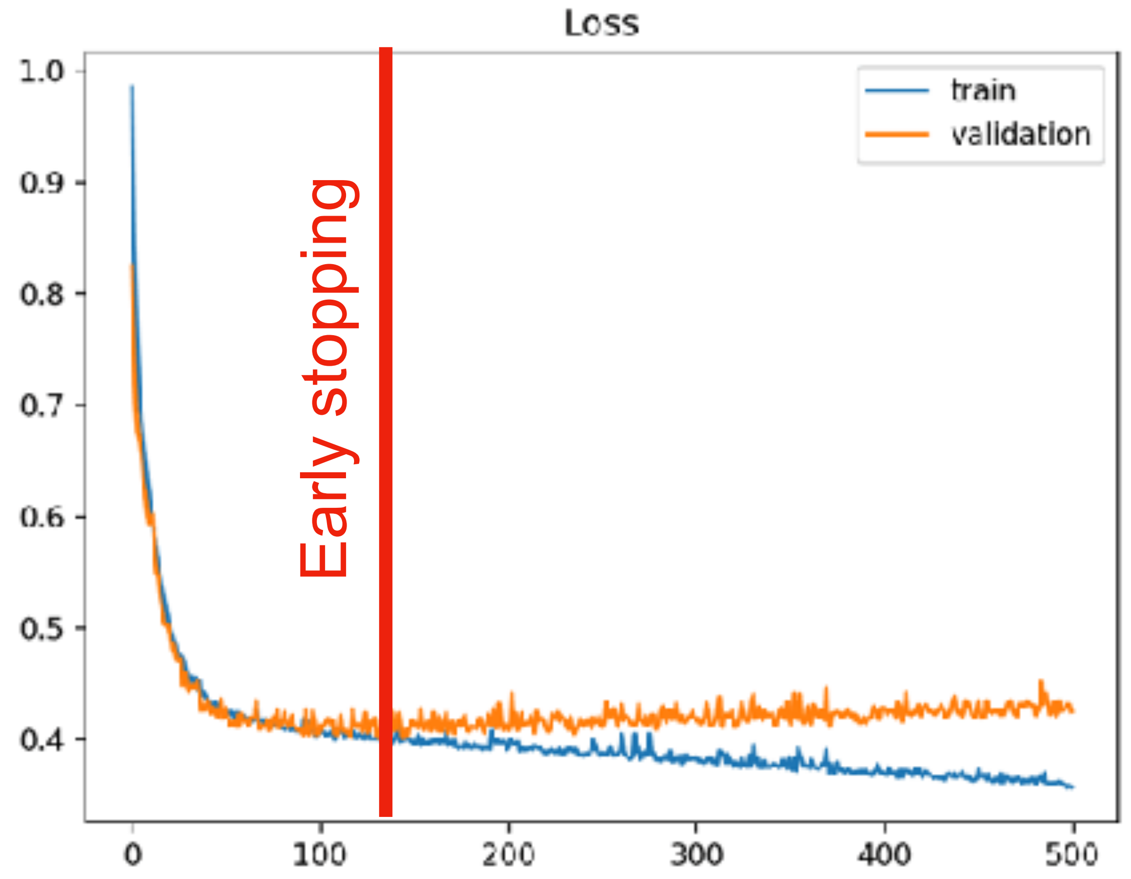


Image source



L2 penalty:
encourages the norm of
the parameters to be low

$$L(\theta) + \underbrace{\frac{\lambda}{2}\,||\theta||_2^2}_{}$$

$$\theta_{t+1} \rightarrow \theta_t - \alpha \frac{\partial L}{\partial \theta_t} - \underbrace{\alpha\,\lambda\theta_t}_{\text{gradient}}$$

Weight regularization

3 complementary approaches:

- Regularizing the weights with weight decay
- Initialization
- Normalization of activations

Why is weight regularization important?

- Many networks produce same results

- Examples: permutations of weights, invariant to multiplication…

- We can reduce space of exploration to smaller set of networks

→ Faster convergence

Weight decay

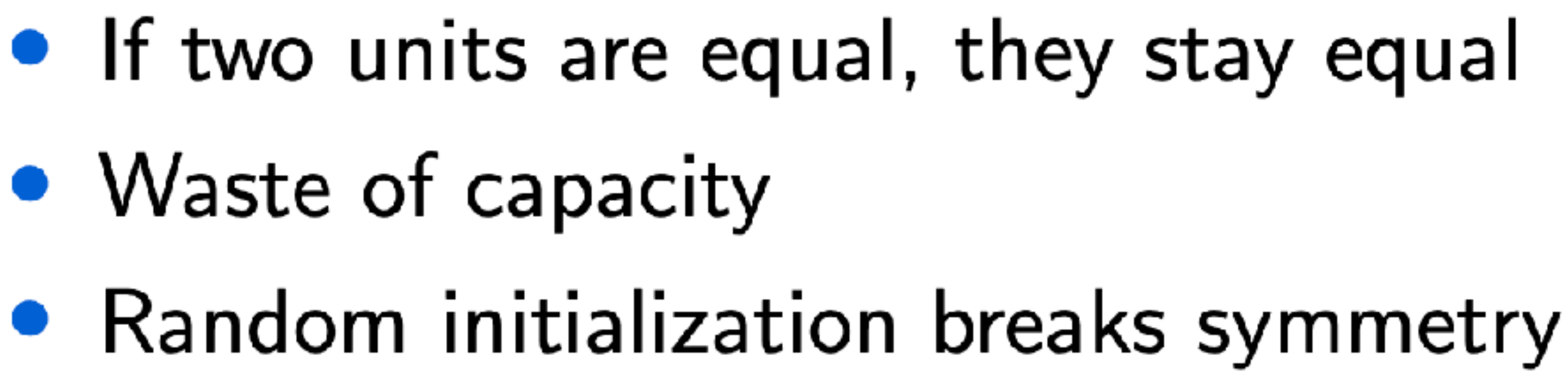
- Apply a L_2 regularization on the parameters

L2 penalty:
encourages the norm of
the parameters to be low

$$L(\theta) + \underbrace{\frac{\lambda}{2}\,||\theta||_2^2} \qquad \theta_{t+1} \rightarrow \theta_t \; - \alpha \frac{\partial L}{\partial \theta_t} \; - \underbrace{\alpha\,\lambda\theta_t}_{\text{gradient}}$$

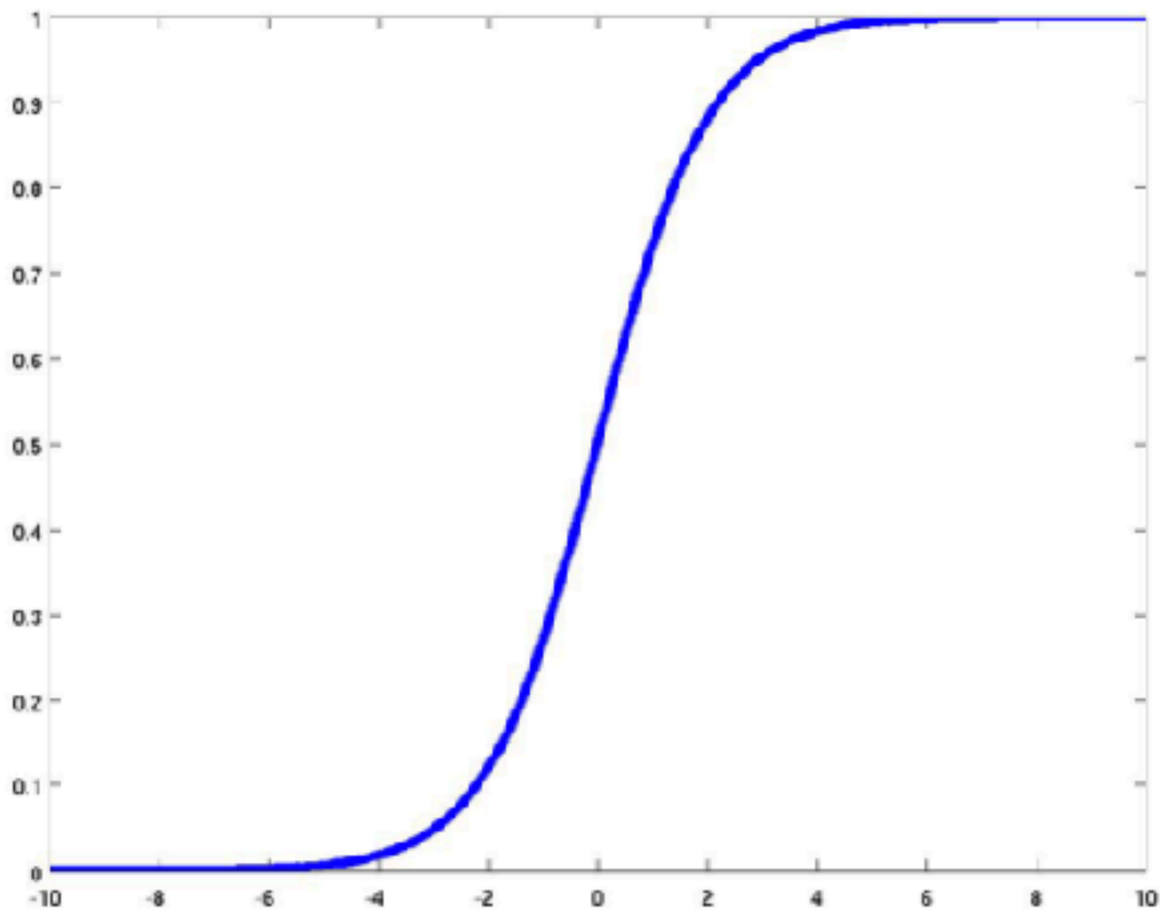
- $\lambda > 0$ decreases during training with the learning rate
- Different from standard regularizartion where $\lambda > 0$ is fixed.

Weight Initialization

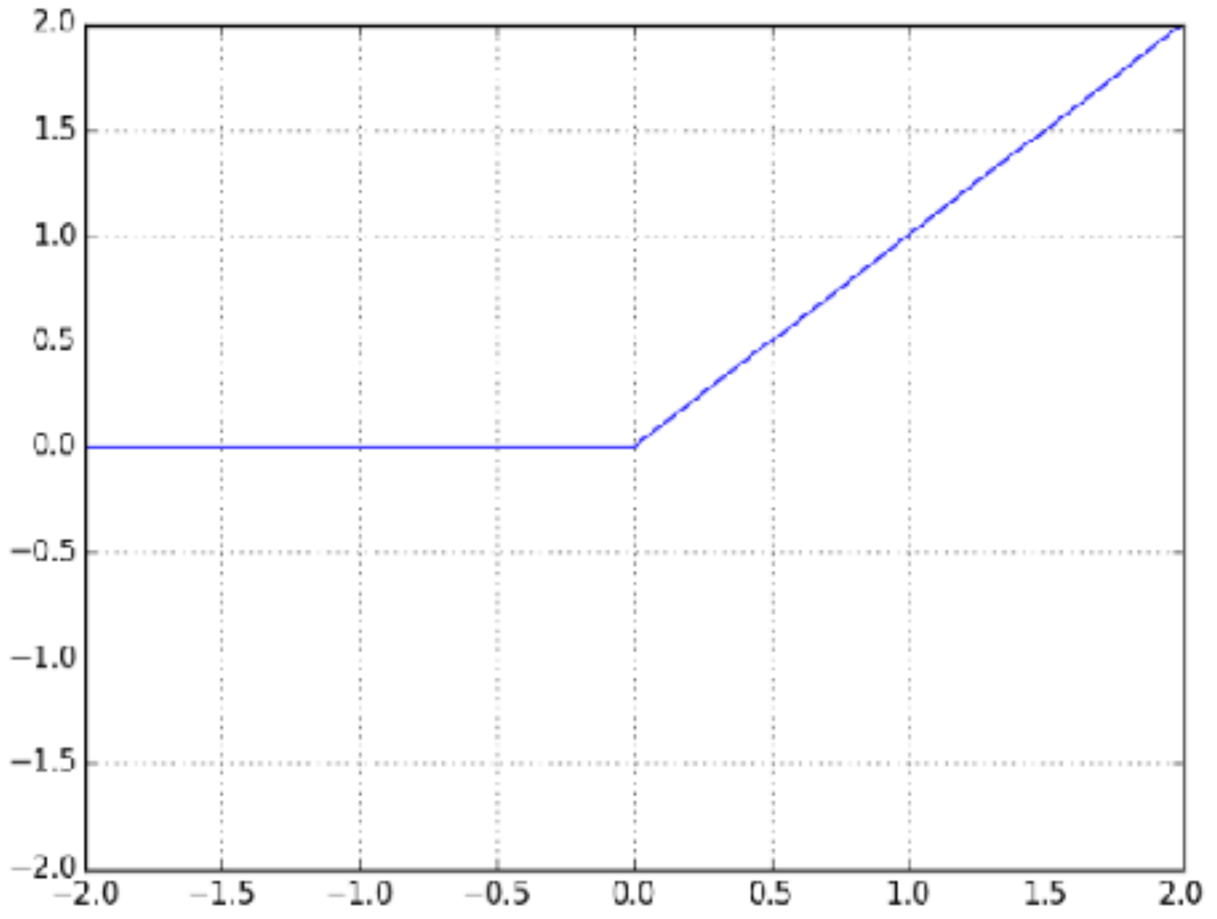


- If two units are equal, they stay equal
- Waste of capacity
- Random initialization breaks symmetry

Weight Initialization



Sigmoid



Rectified linear Unit

- Many nonlinearities have regions with 0 norm gradients
- Initialization must avoid saturated areas
- Alernatively use nonlinearities with no saturation:

$$\text{Leaky ReLU} = \text{ReLU}(x) + \alpha x, \ \text{with } \alpha > 0.$$

Fan-in initialization

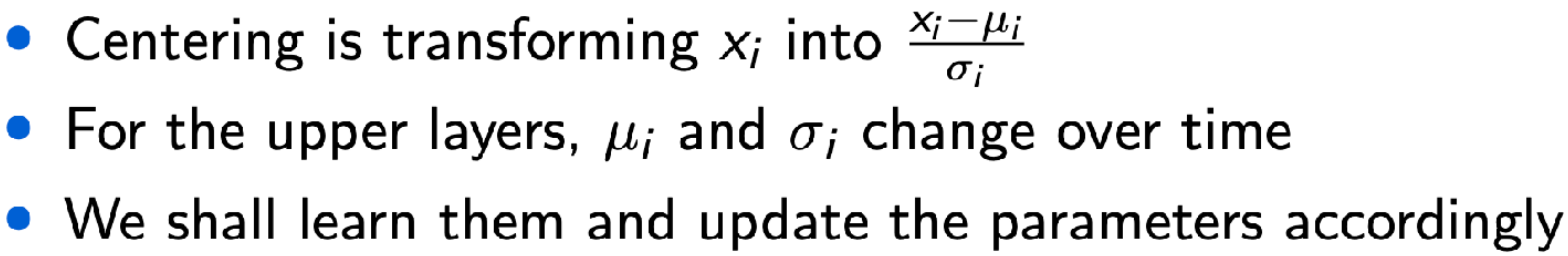
- **Fan-in**: number of inputs used to compute a hidden units
- Large fan-in implies larger changes in hidden variables
- Need smaller initialization
- Typically, weights $\approx 1/\sqrt{\text{fan-in}}$

Data normalization (whitening)

- Update of a layer is proportional to its input
- Example:
  - Assume $X_1 = 100$ and $X_2 = 101$
  - $\nabla \ell_1 = +1$ and $\nabla \ell_2 = -1$
  - Mean of updates is small ($\propto -0.5$) but each update is huge ($\propto 100$)
- Center data is important!
- Centering is transforming x_i into $\frac{x_i - \mu_i}{\sigma_i}$

Intermediate normalization

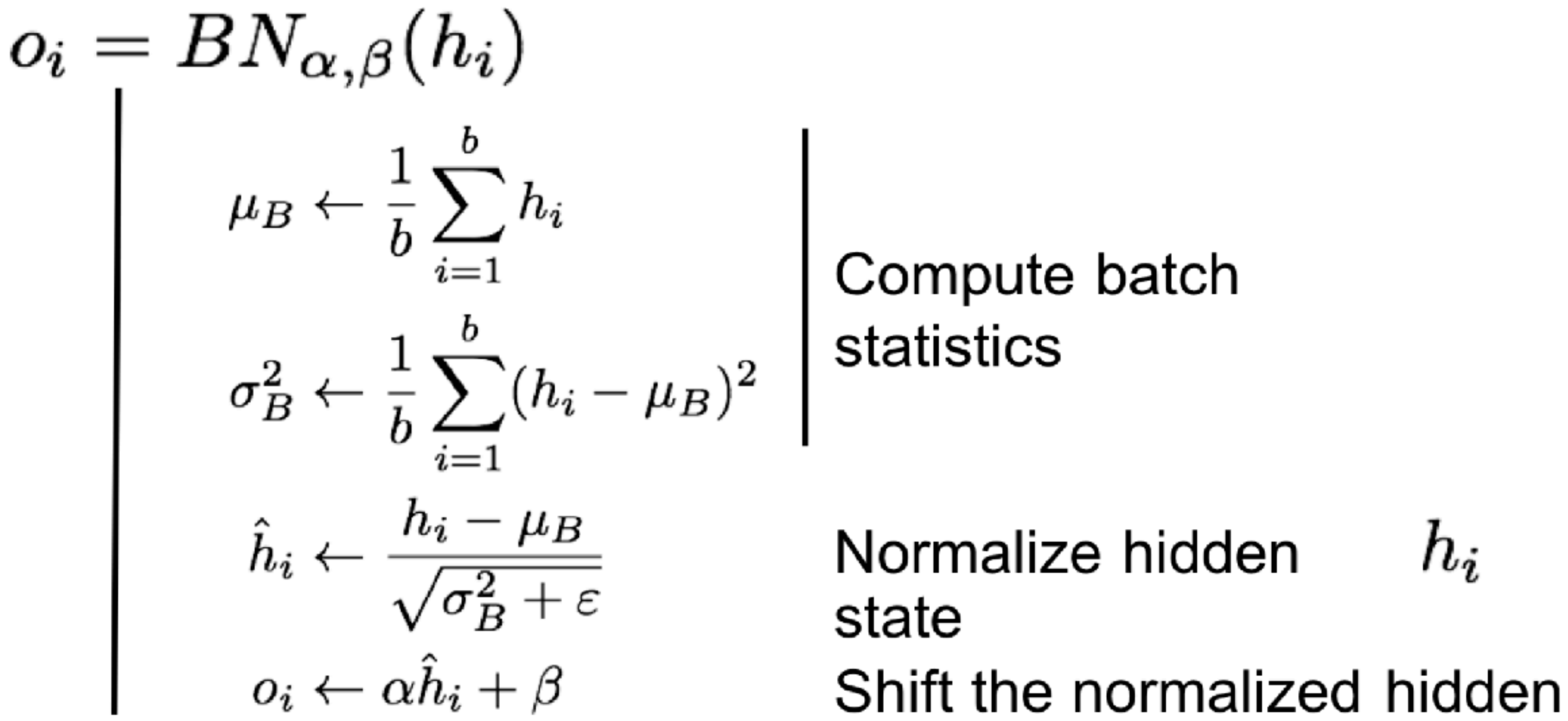
Example: batch normalization (batchnorm)



- Centering is transforming x_i into $\frac{x_i - \mu_i}{\sigma_i}$
- For the upper layers, μ_i and σ_i change over time
- We shall learn them and update the parameters accordingly

Recap: Batch normalization

Why “batch”?
Disadvantages?



$$o_i = BN_{\alpha,\beta}(h_i)$$

$$\mu_B \leftarrow \frac{1}{b}\sum_{i=1}^{b} h_i$$

$$\sigma_B^2 \leftarrow \frac{1}{b}\sum_{i=1}^{b}(h_i - \mu_B)^2$$

Compute batch statistics

$$\hat{h}_i \leftarrow \frac{h_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

Normalize hidden state h_i

$$o_i \leftarrow \alpha \hat{h}_i + \beta$$

Shift the normalized hidden

α and β are learned over time.

Summary of optimization

Standard optimization:

- ADAM (or AdamW)
- clipping
- cosine scheduler
- warm-up
- init based on fan-in
- grid search over initial learning rate and weight decay

Look at your results

- When you train a network, you should try to really understand what is happening:
  - Train/val/test sets are important
  - **Look at loss** and performance on train/val sets during training
  - Choose LR, compare networks, try different initialization (random seeds)
  - Try to overfit on a subset of the training set first

- **Very important: Look** at your data and results (e.g., visualize predictions) on training and testing data.

More tips at Andrej Karpathy blog:
A Recipe for Training Neural Networks https://karpathy.github.io/2019/04/25/recipe/

Slide: M. Aubry

Practical problems

- **Data loading:**
  - Loading "on the fly": needed for big datasets, use efficient database structure, fast disk access, e.g., SSD
  - Loading to RAM: possible for smaller datasets, or pre-computed features
- **Speed:** use GPUs, parallel data loading
- **Network size:** get lots of memory on your GPU or/and use several GPUs

Good news: you don't have to do all of it!

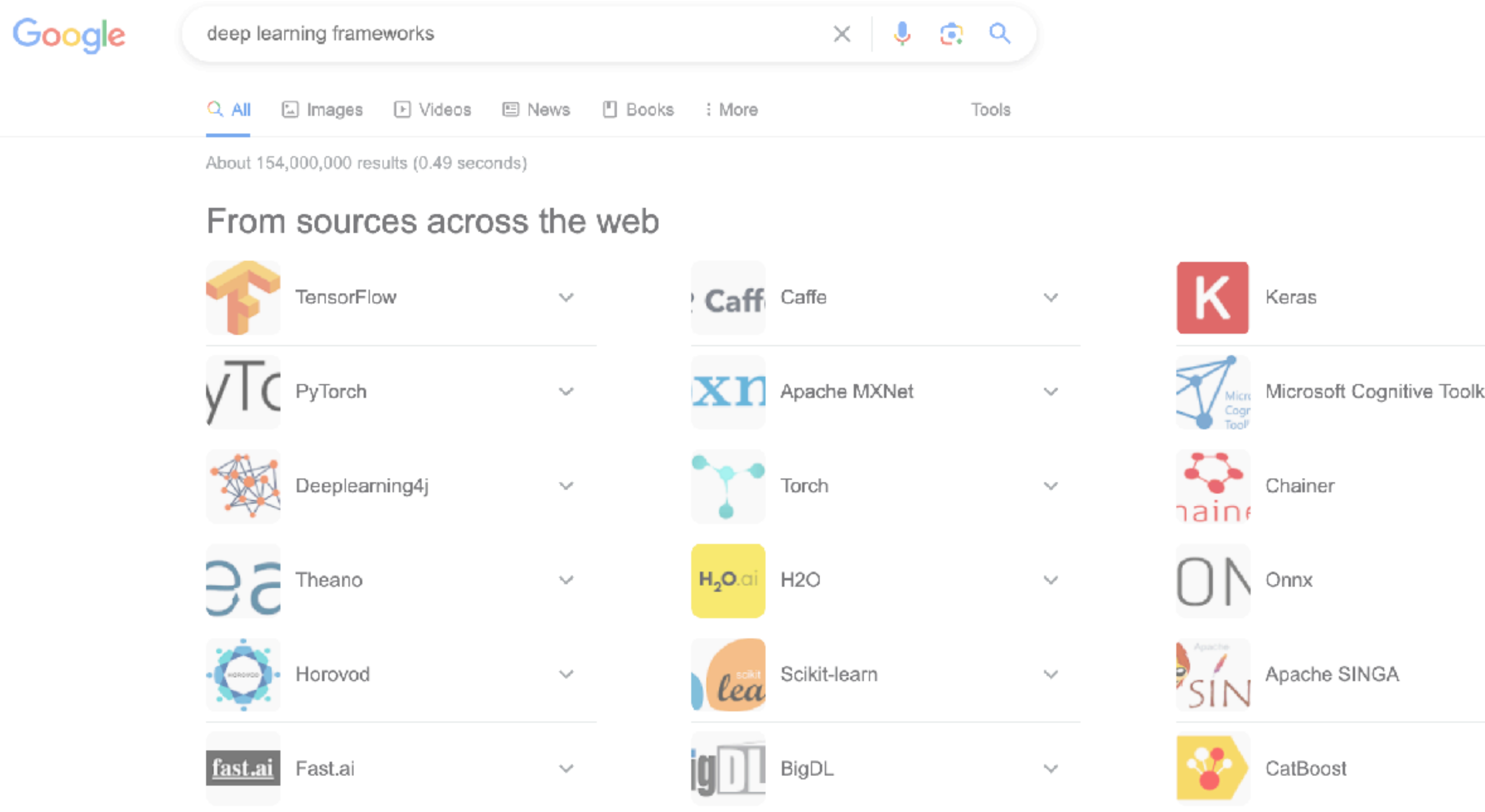
Many ready-to-use and efficient frameworks are available (e.g., Pytorch)

Slide: M. Aubry

NN packages

- PyTorch (Python)
  - http://pytorch.org/
- TensorFlow (Python) - Google
  - https://www.tensorflow.org/
- Lua Torch
  - http://torch.ch/
- Caffe (C++, pycaffe, matcaffe)
  - http://caffe.berkeleyvision.org/
- MatConvNet (Matlab)
  - http://www.vlfeat.org/matconvnet/

●●●



About 154,000,000 results (0.49 seconds)

From sources across the web

TensorFlow
PyTorch
Deeplearning4j
Theano
Horovod
Fast.ai

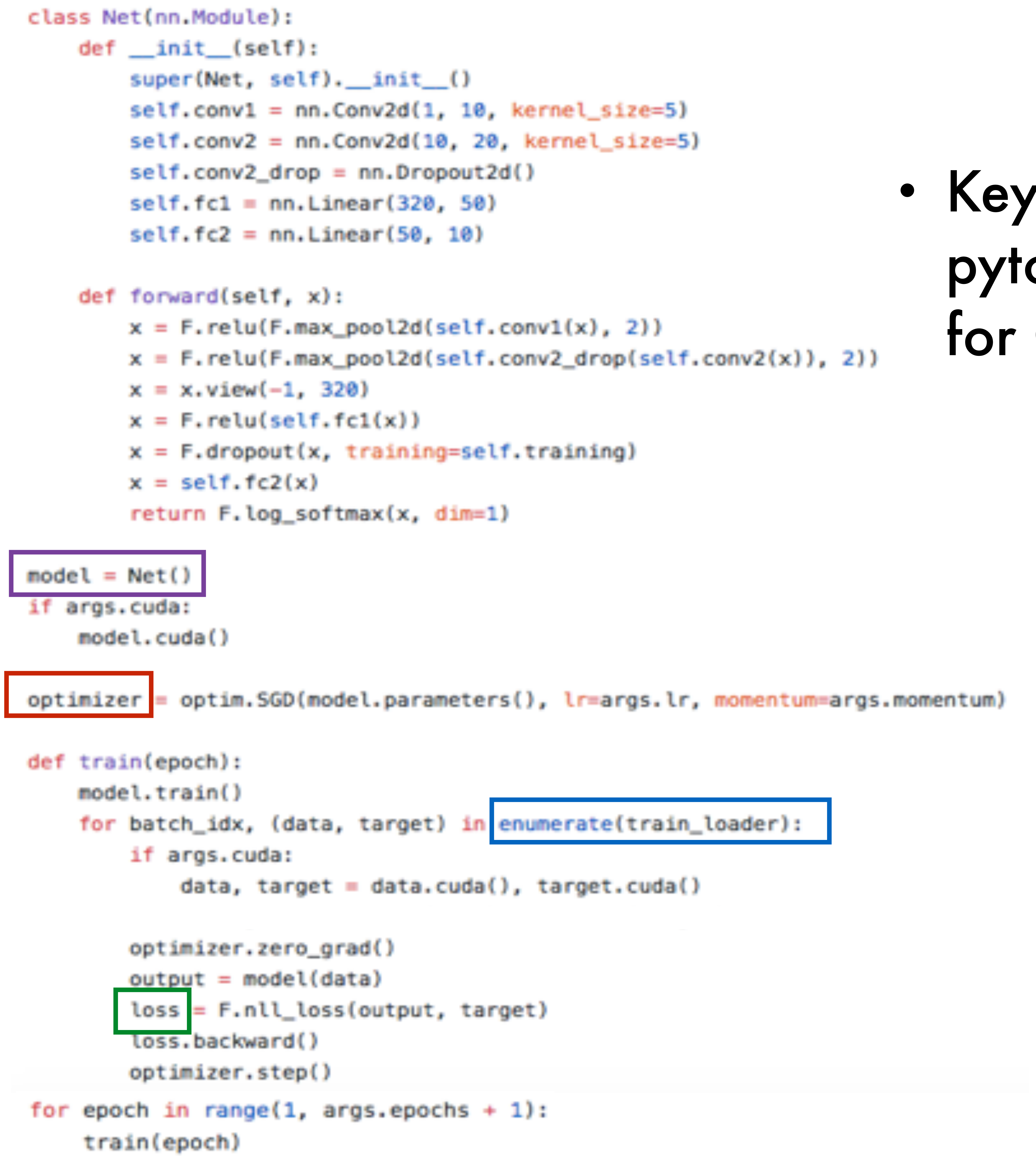
Caffe
Apache MXNet
Torch
H2O
Scikit-learn
BigDL

Keras
Microsoft Cognitive Toolk
Chainer
Onnx
Apache SINGA
CatBoost

Let’s look at some code

(more in Assignment 2)

- The key objects are
  - model,
  - optimizer,
  - dataloader,
  - loss.



```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)

model = Net()
if args.cuda:
    model.cuda()

optimizer = optim.SGD(model.parameters(), lr=args.lr, momentum=args.momentum)

def train(epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        if args.cuda:
            data, target = data.cuda(), target.cuda()

        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()

for epoch in range(1, args.epochs + 1):
    train(epoch)
```

- Key part of pytorch code for CNN learning

Slide: M. Aubry