Recap

Recall: Linear Classifier

Defines a score function:

$$\underbrace{\overbrace{f(\mathbf{x}_i, \mathbf{W}, \mathbf{b}) = \mathbf{W}\mathbf{x}_i}^{\text{image features}}}_{\text{weights}} + \underbrace{\mathbf{b}}_{\text{bias vector}}$$

weights
(parameters)
bias vector

Image Credit: Ioannis (Yannis) Gkioulekas (CMU)

Recall: Linear Classifier

Example with an image with 4 pixels, and 3 classes (cat/dog/ship)

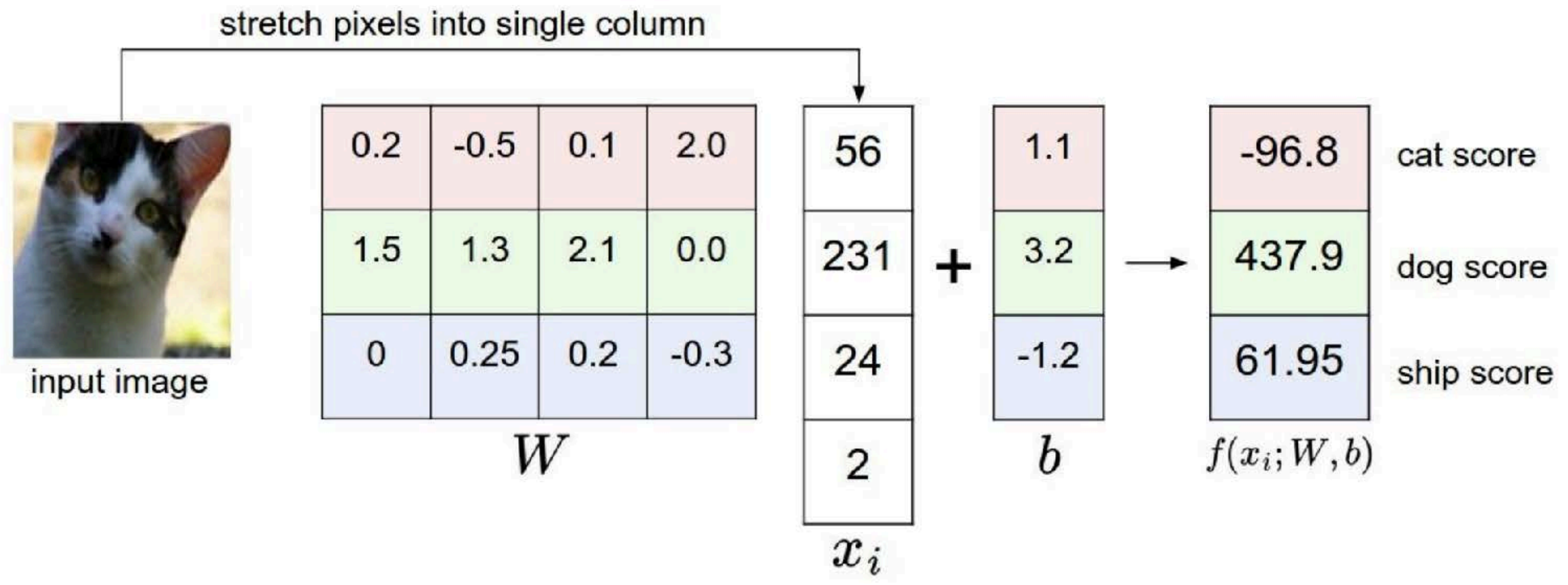


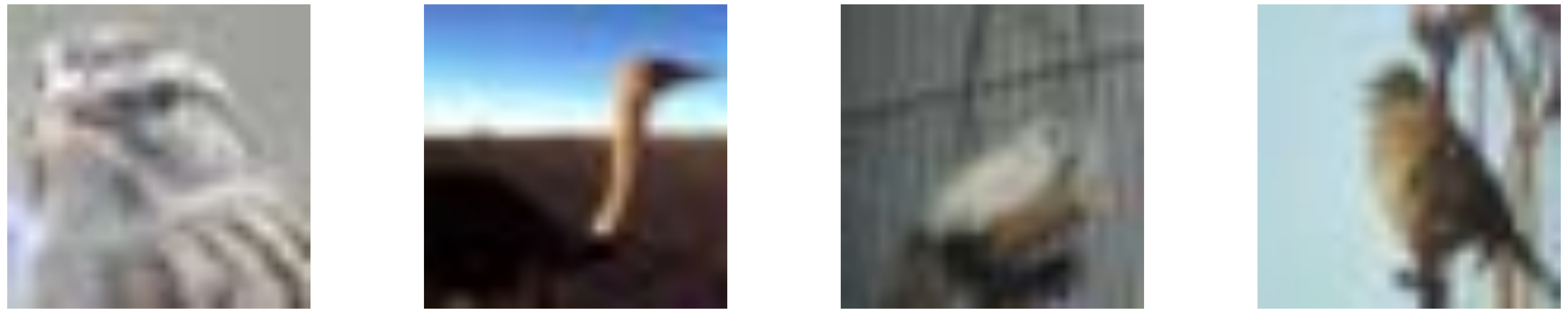
Image Credit: Ioannis (Yannis) Gkioulekas (CMU)

Linear Classification

- Let's start by using 2 classes, e.g., bird and plane
- Apply labels (y) to training set:

y = +1

y = -1

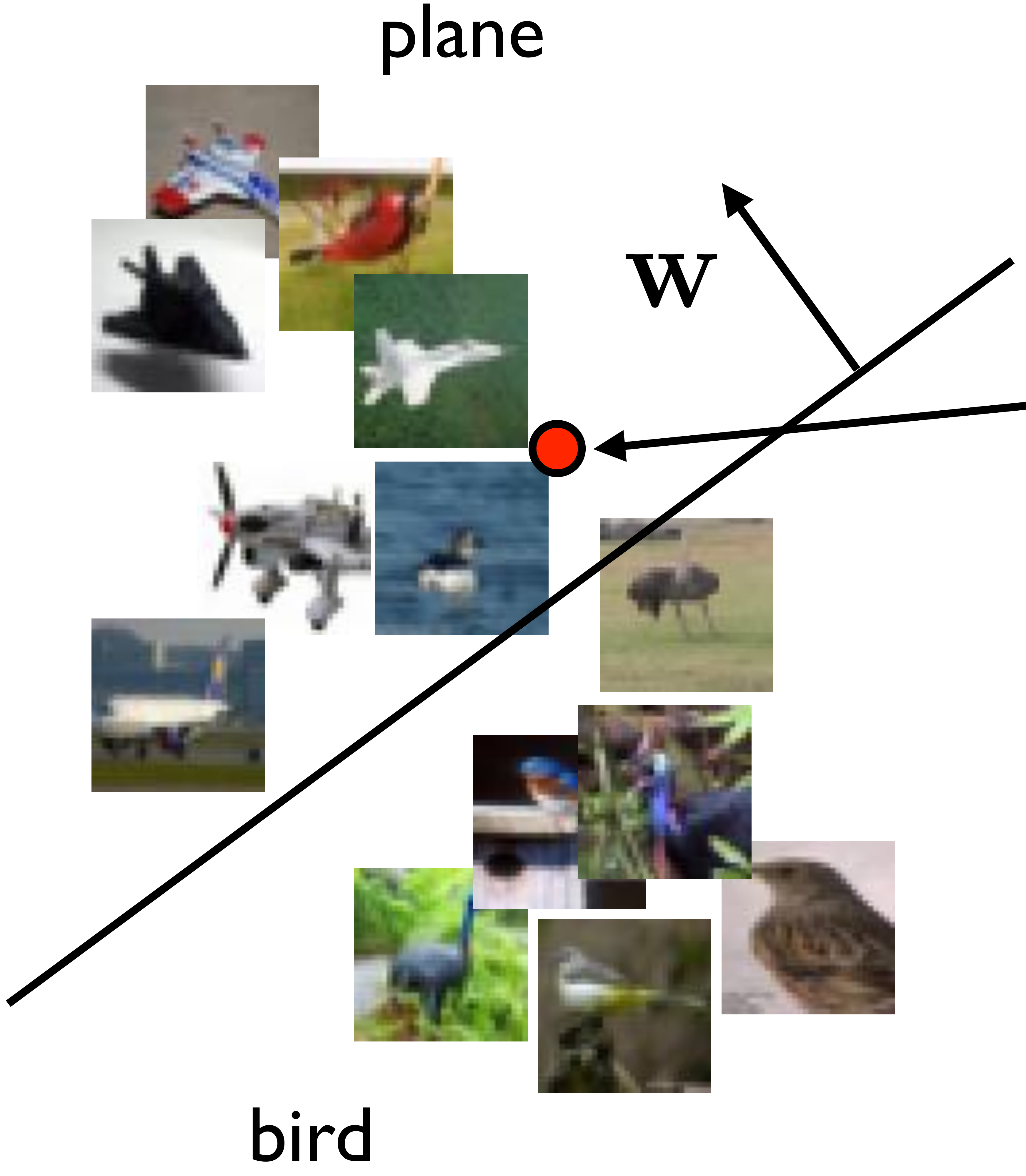


- Use a linear model to regress y from x

$$\hat{y} = \text{sign}\ h = \text{sign}\ \mathbf{w}^T \mathbf{x}_q$$

2-class Linear Classification

- Separating hyperplane, projection to a line defined by **w**



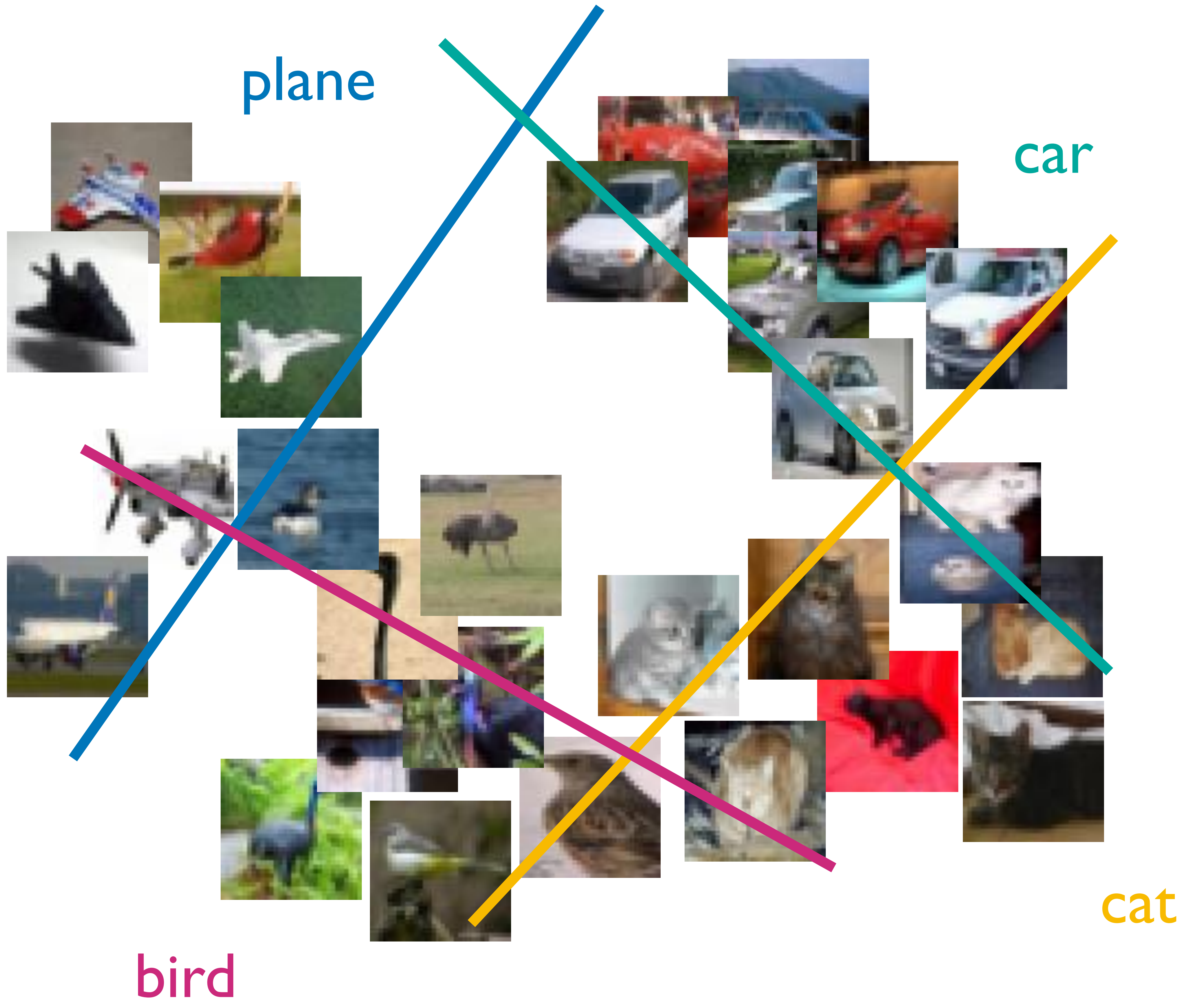
Query:

$\mathbf{x}_q =$ 

$$\hat{y} = \text{sign}\ h = \text{sign}\ \mathbf{w}^T \mathbf{x}_q$$

N-class Linear Classification

- One hot regression = 1 vs all classifiers



One-Hot Regression

- A better solution is to regress to one-hot targets = 1 vs all classifiers

$$\begin{bmatrix} & & \\ & \mathbf{W}^T & \\ & & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \dots \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ \dots \end{bmatrix}$$

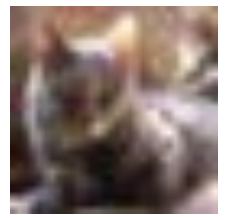
class 2 = ‘automobile’

$$\begin{bmatrix} & & \\ & \mathbf{W}^T & \\ & & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \dots \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ \dots \end{bmatrix}$$

class 4 = ‘cat’

One-Hot Regression

- Transpose (to match Project 3 notebook)

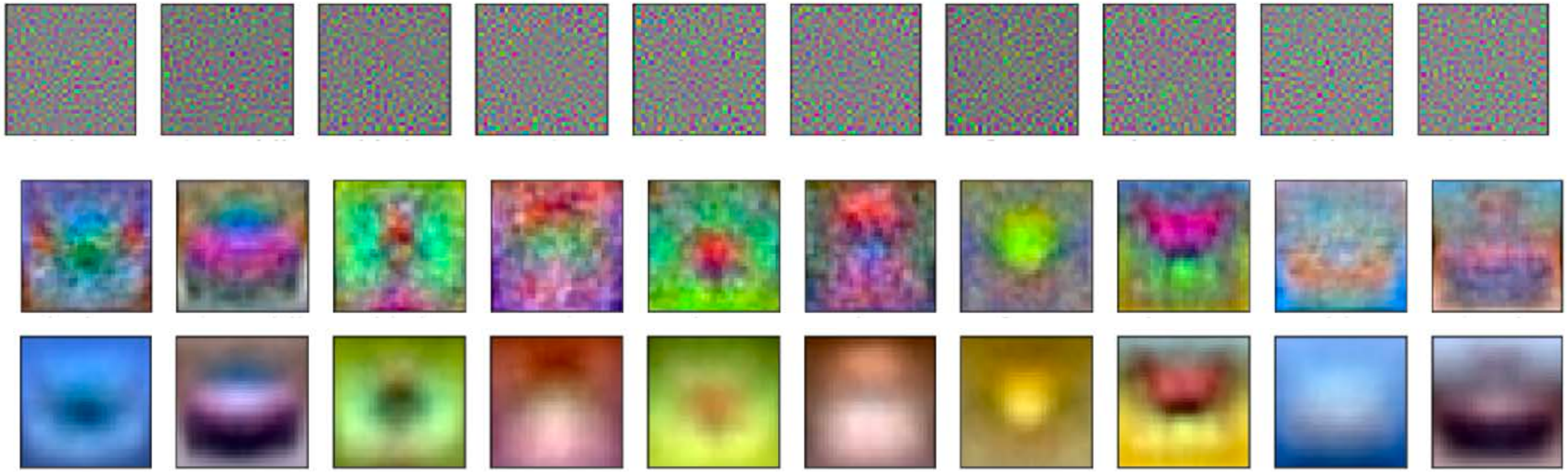
$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots \\ x_{21} & x_{22} & x_{23} & \dots \\ x_{31} & x_{32} & x_{33} & \dots \\ & \dots & & \end{bmatrix} \begin{bmatrix} & & \\ & \mathbf{W} & \\ & & \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & \dots \\ 0 & 0 & 0 & 1 & \dots \\ & .. & .. & & \end{bmatrix} \begin{array}{l} \text{auto} \\ \text{cat} \\ \\ \end{array}$$

$$\mathbf{XW} = \mathbf{T}$$

- Solve regression problem by Least Squares

Regularized Classification

- Add regularization to CIFAR10 linear classifier

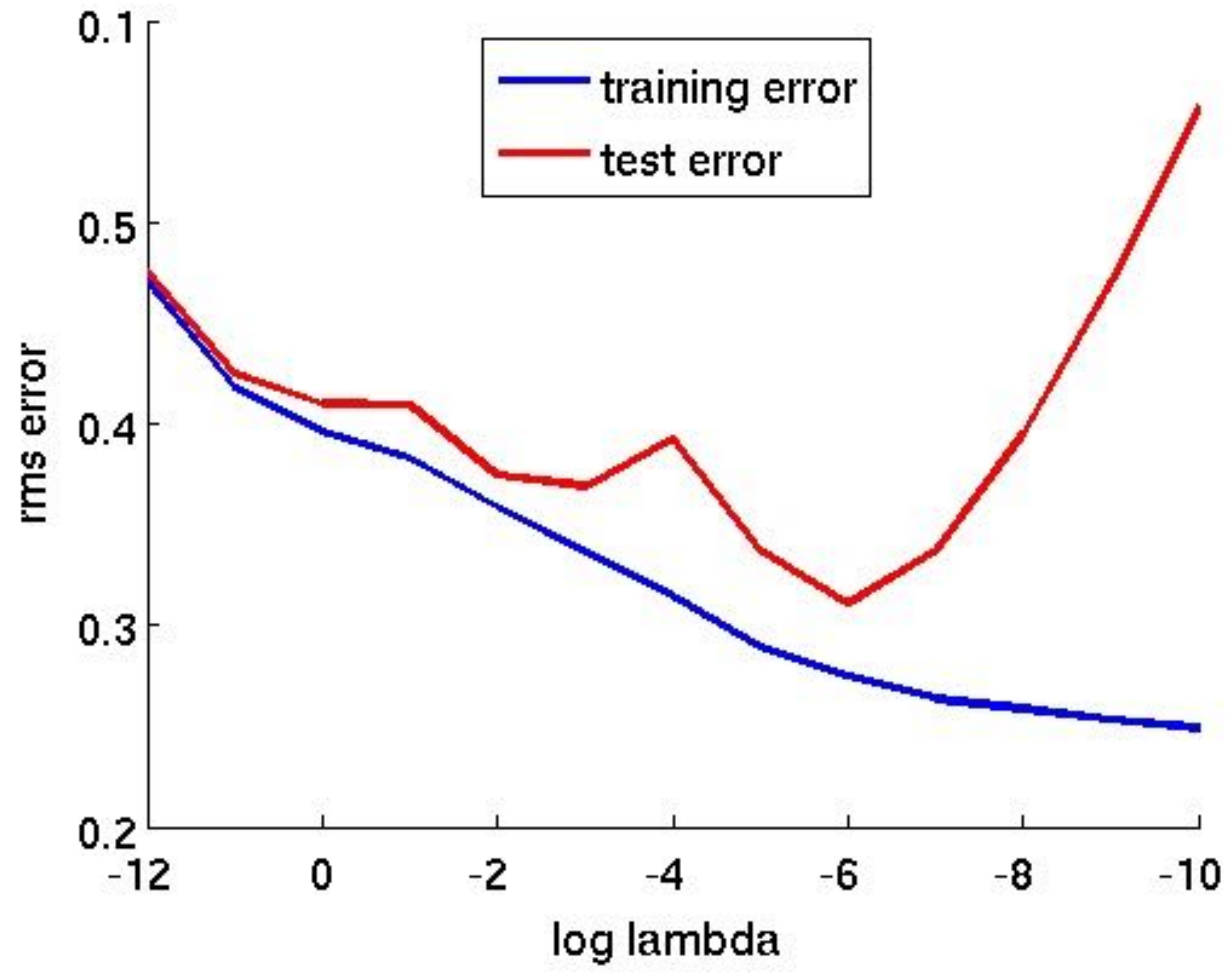


- Row 1 = overfitting, Row 3 = oversmoothing?

$$e = |\mathbf{XW} - \mathbf{T}|^2 + \lambda |\mathbf{W}|^2$$

Under/Overfitting

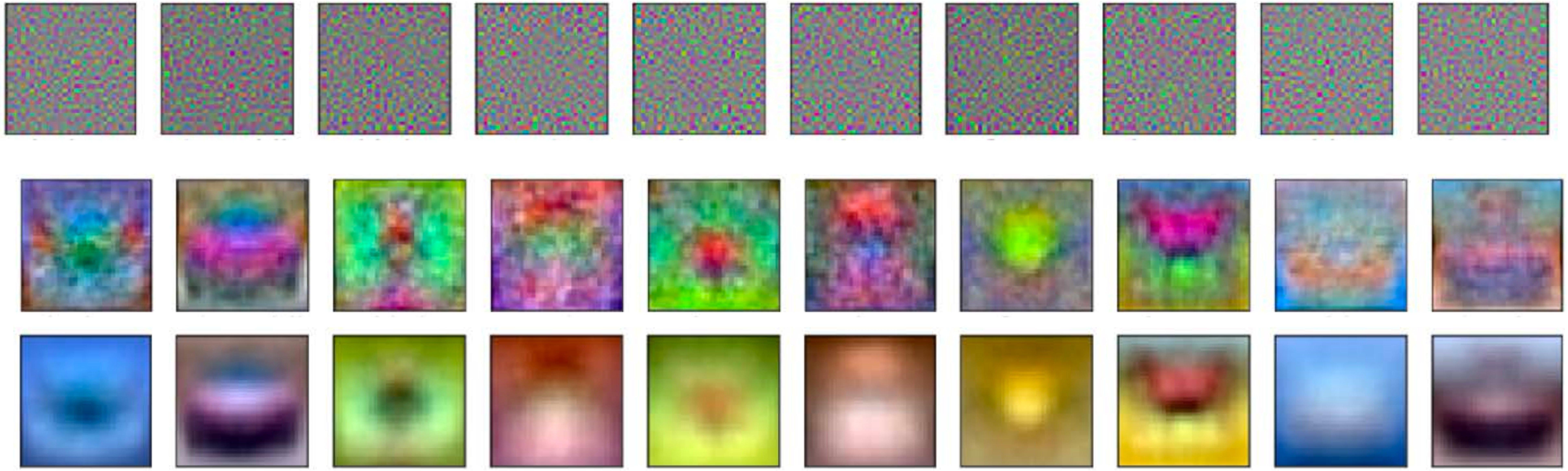
- Test error vs lambda



- Training error always decreases as lambda is reduced
- Test error reaches a minimum, then increases ⇒ overfitting

Regularized Classification

- Add regularization to CIFAR10 linear classifier



- Row 1 = overfitting, Row 3 = oversmoothing?

$$e = |\mathbf{XW} - \mathbf{T}|^2 + \lambda|\mathbf{W}|^2$$

Non-Linear Optimisation

- With a linear predictor and L2 loss, we have a closed form solution for model weights W
- How about this (non-linear) function

$$\mathbf{h} = \mathbf{W}_2 \max(0, \mathbf{W}_1 \mathbf{x})$$

- Previously (e.g., bundle adjustment), we locally linearised the error function and iteratively solved linear problems

$$e = \sum_i |\mathbf{h}_i - \mathbf{t}_i|^2 \approx |\mathbf{J}\Delta\mathbf{W} + \mathbf{r}|^2$$

$$\Delta\mathbf{W} = -(\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T\mathbf{r}$$

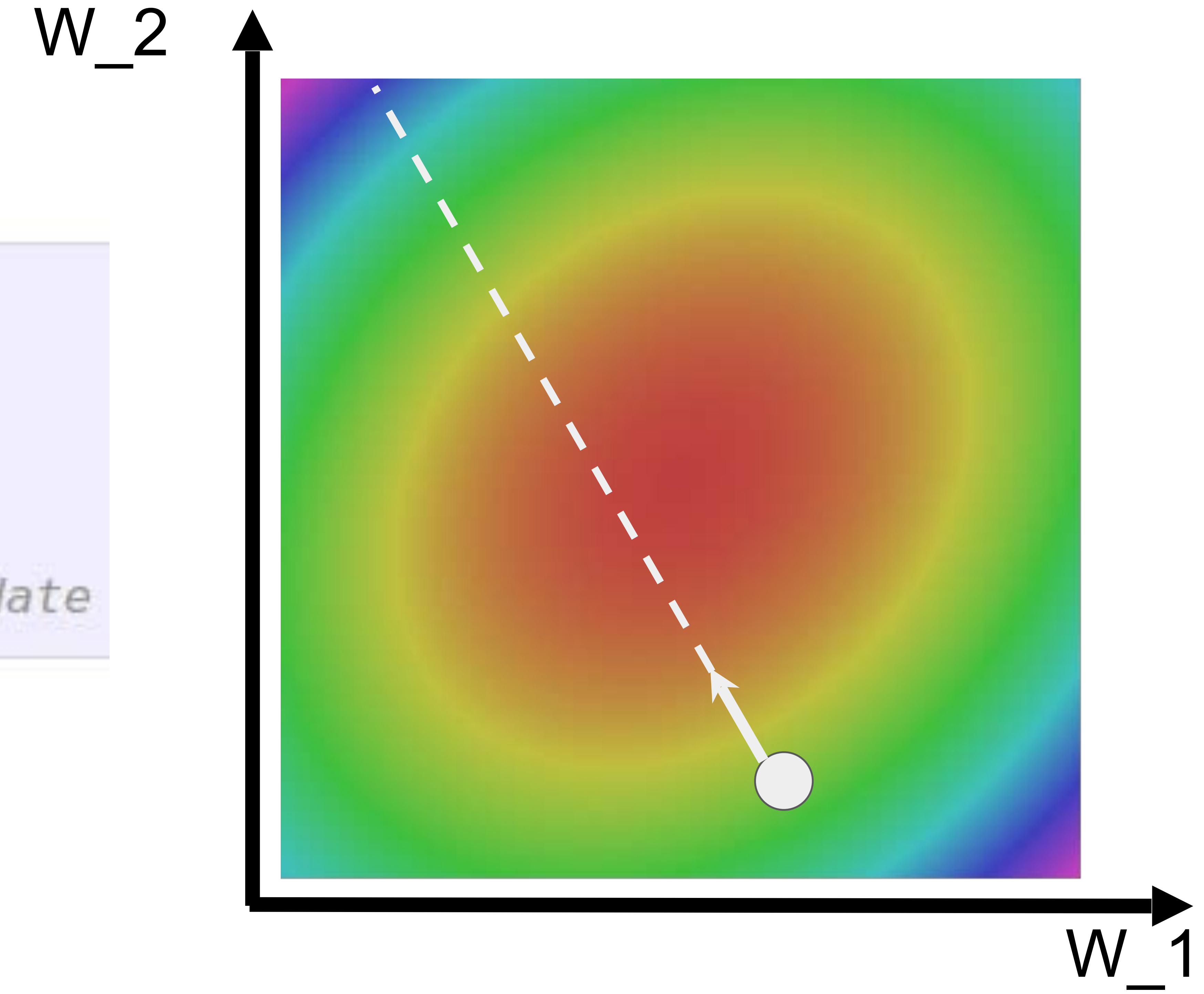
Does this look like a promising approach?

Gradient descent one more time

Vanilla Gradient Descent

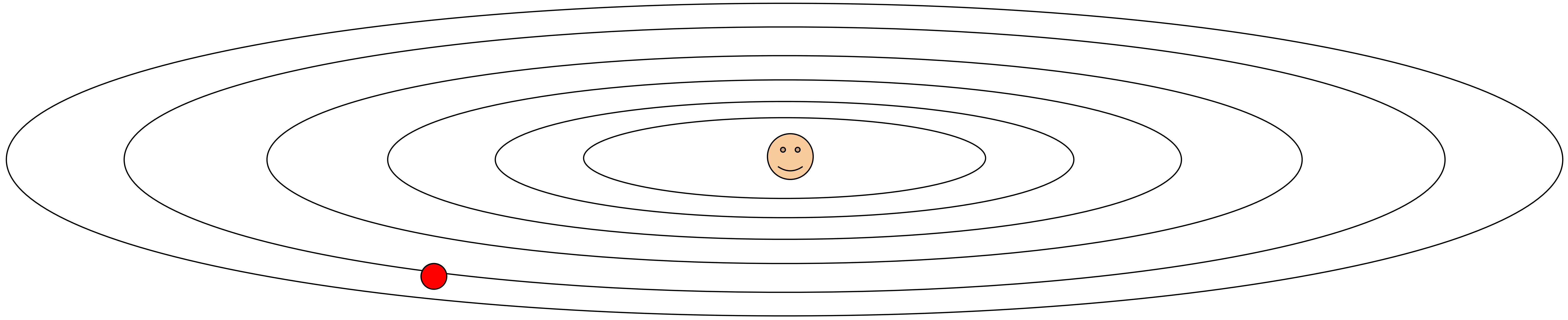
```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```



Problem with vanilla GD

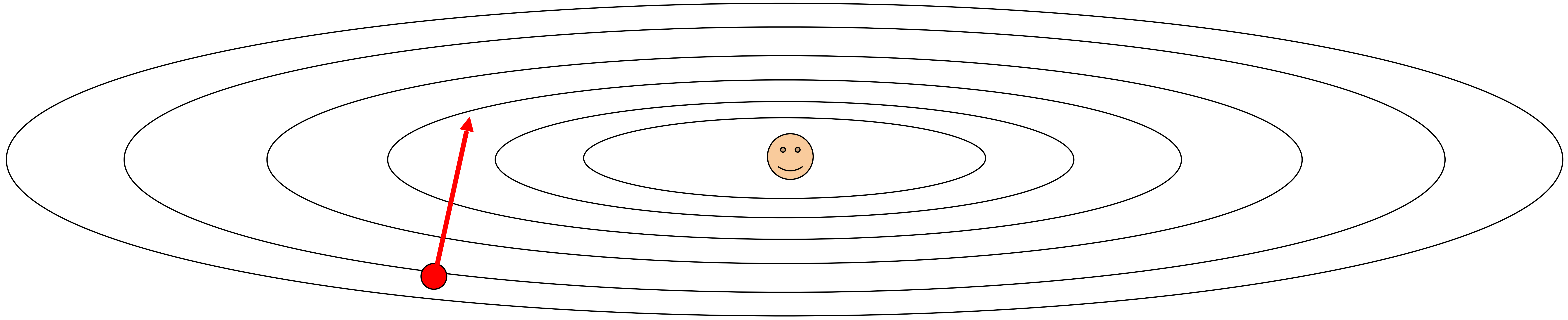
What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction



Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Problem with vanilla GD

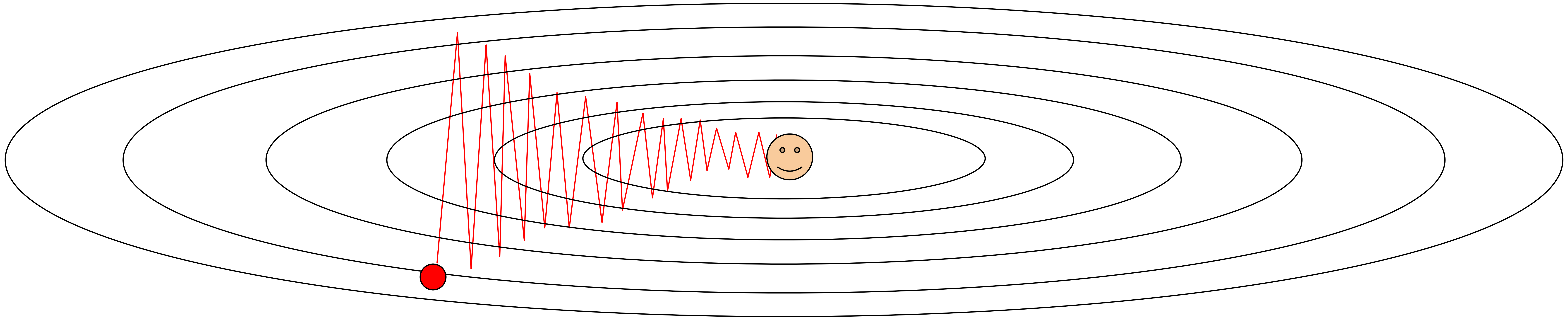
What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction



Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Problem with vanilla GD

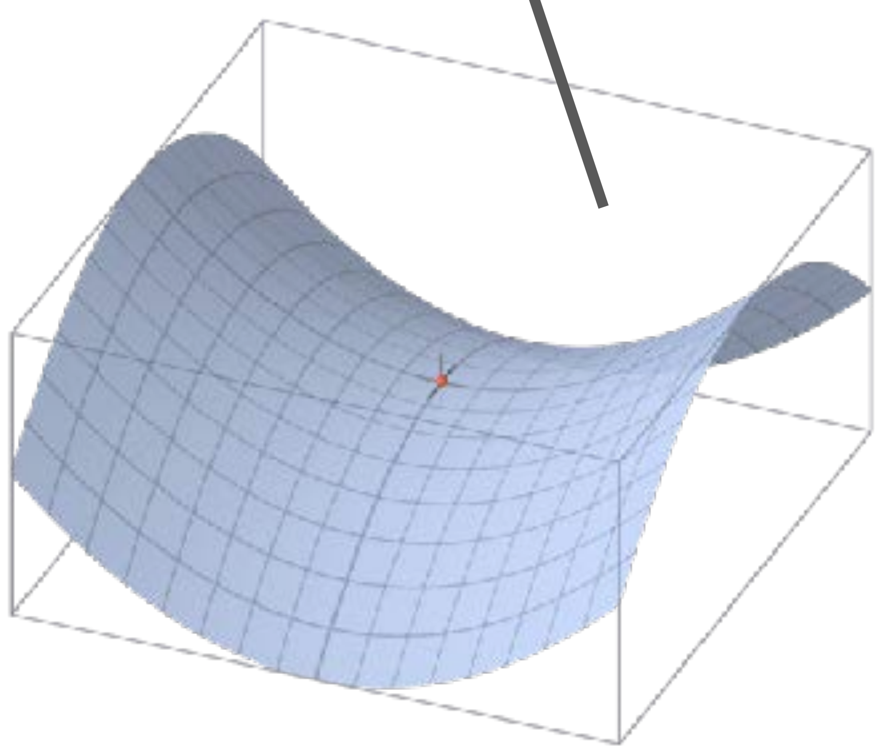
What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction

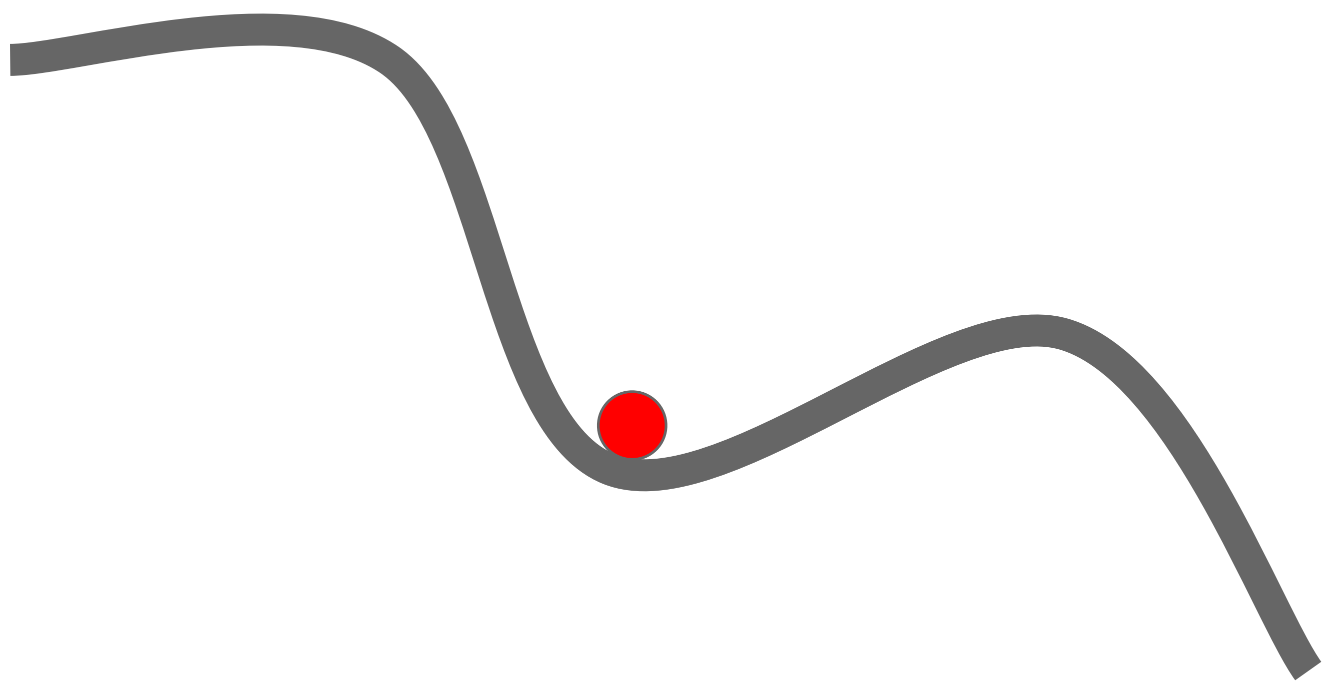


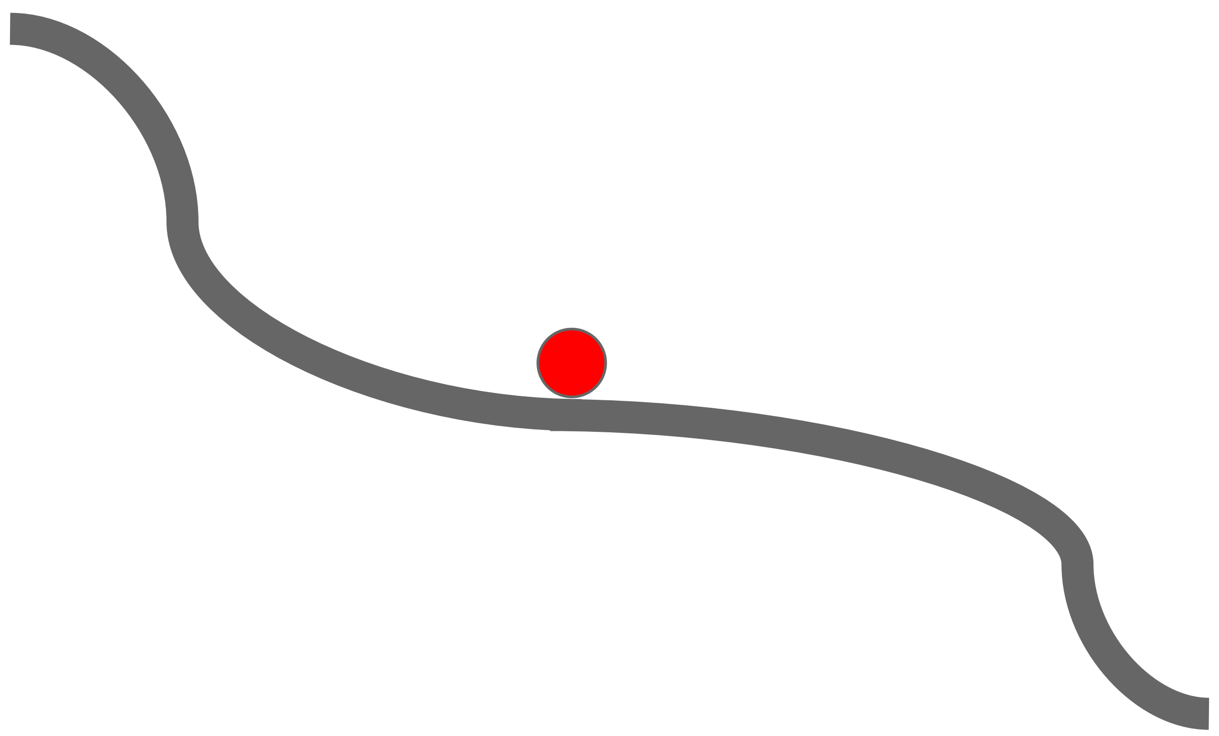
Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Optimization: problem with SGD

What if the loss function has a **local minima** or **saddle point**?







Lee et al, “Gradient Descent Only Converges to Minimizers”, JLMR Workshop and Conference Proceedings, 2016

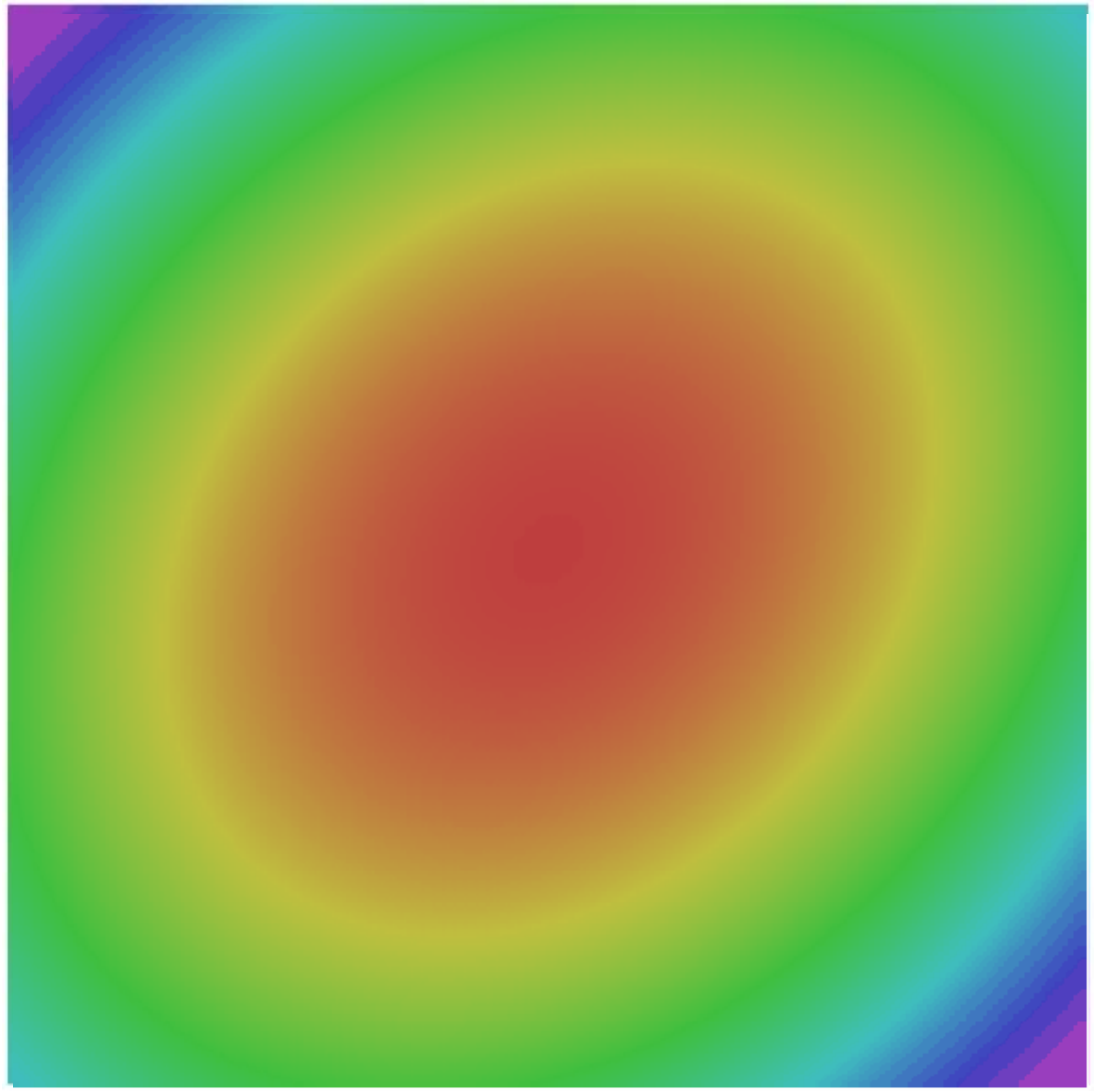
Dauphin et al, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, NIPS 2014

Stochastic gradient descent

Our gradients come from mini-batches so they can be noisy!

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



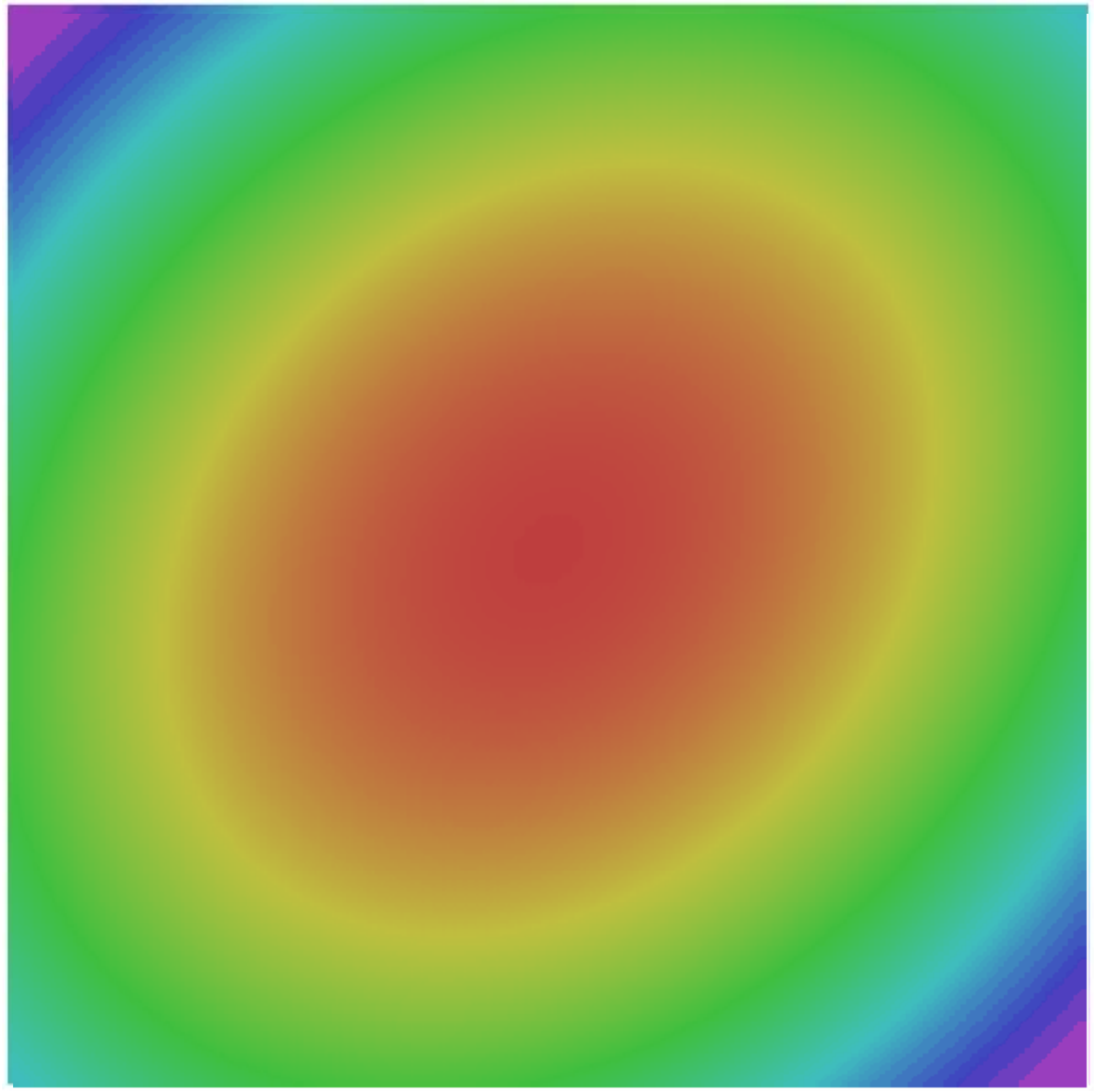
Q: How would you remove the noise?

Stochastic gradient descent

Our gradients come from mini-batches so they can be noisy!

$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



Q: How would you remove the noise?

Based on slides for [Stanford cs231n] by Li, Jonson, and Young. Modified and reused with permission

SGD + Momentum

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

SGD+Momentum

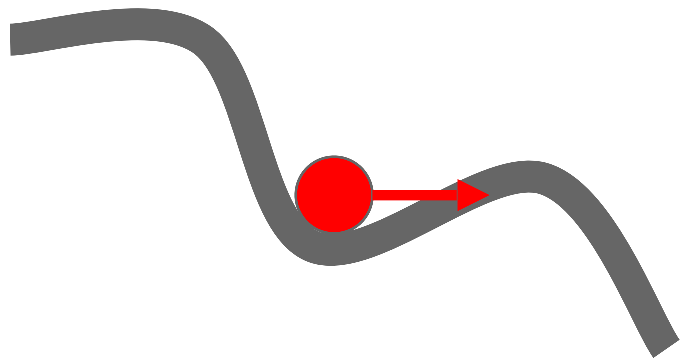
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

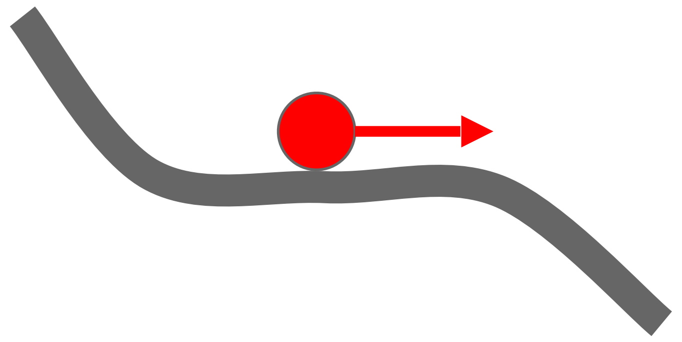
- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

SGD + Momentum

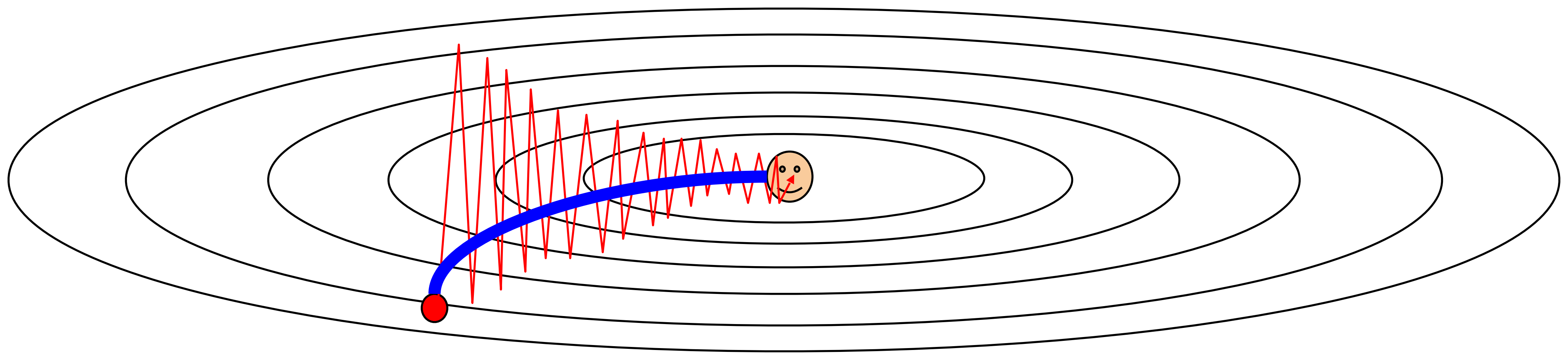
Local Minima



Saddle points



Poor Conditioning

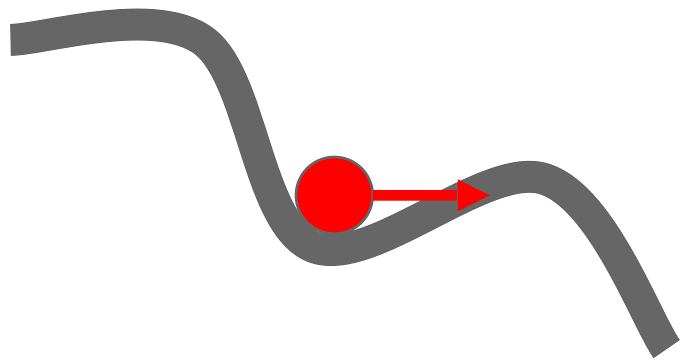


Gradient Noise

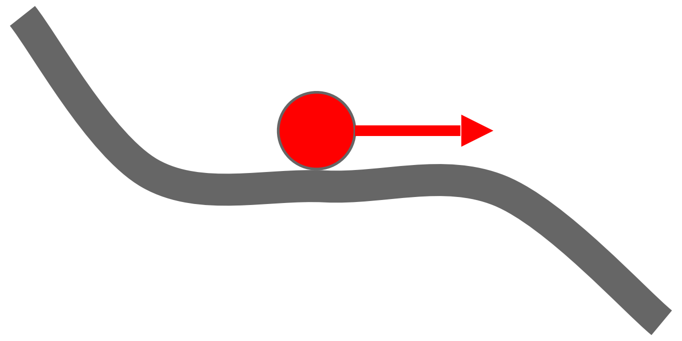
SGD + Momentum

Gradient Noise

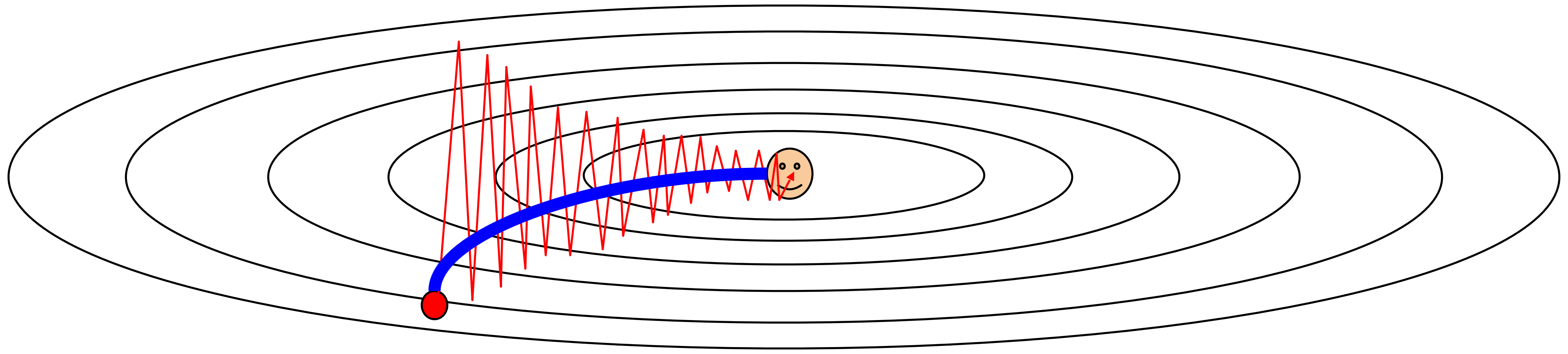
Local Minima



Saddle points



Poor Conditioning



Nesterov Momentum

SGD

SGD+Momentum

Nesterov

Nesterov Momentum

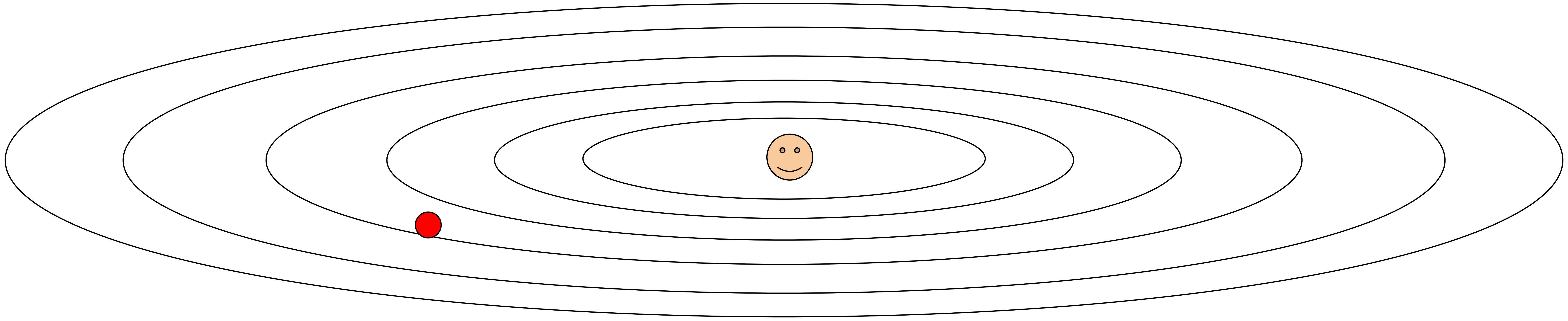
SGD

SGD+Momentum

Nesterov

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q: What happens with RMSProp?

Tieleman and Hinton, 2012

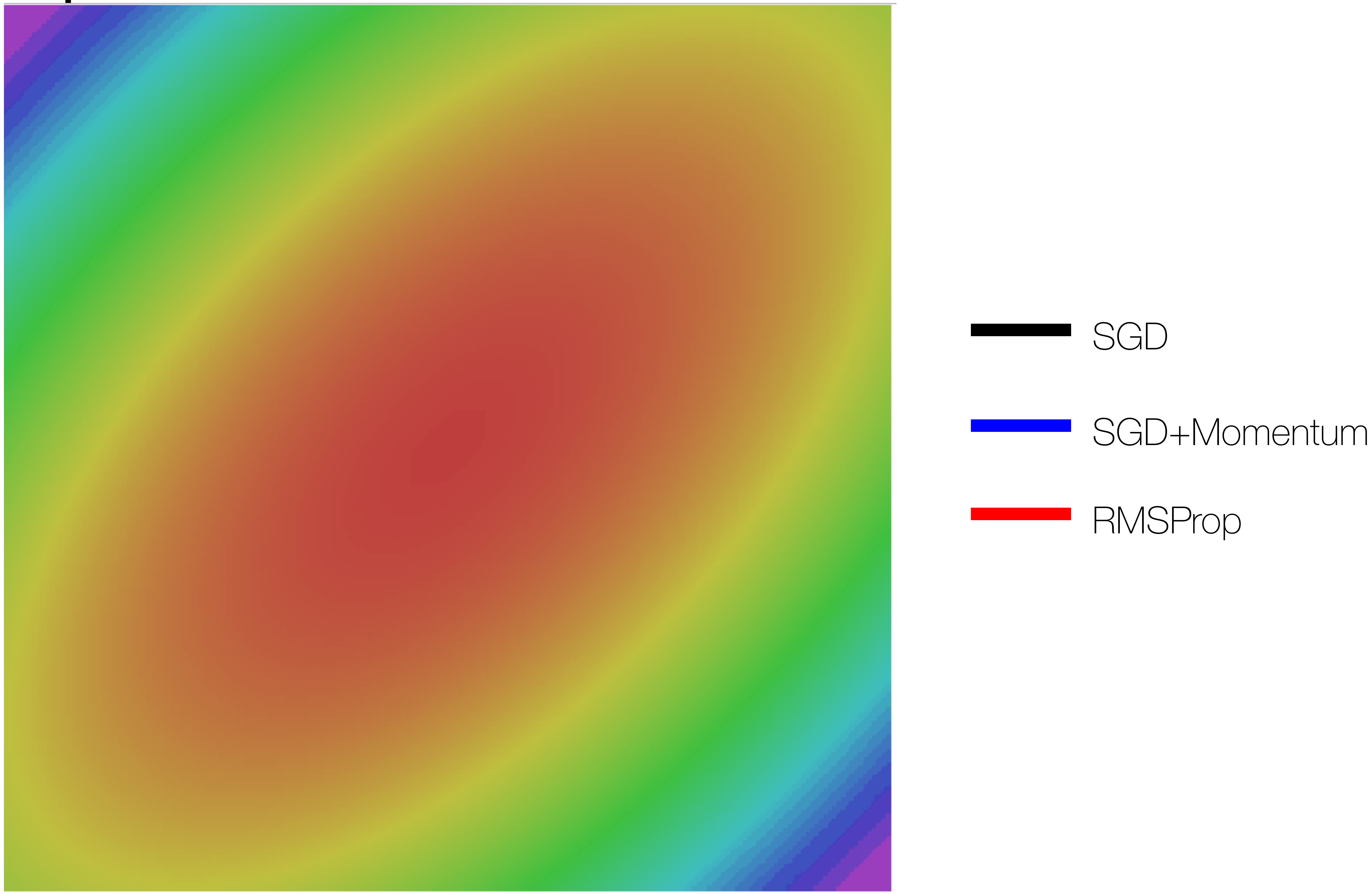
RMSProp

SGD

SGD+Momentum

RMSProp

RMSProp



Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

RMSProp

RMSProp with momentum

Q: What happens at first the timestep?

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-4 is a great starting point for many models!

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Adam

SGD

SGD+Momentum

RMSProp

Adam

Adam

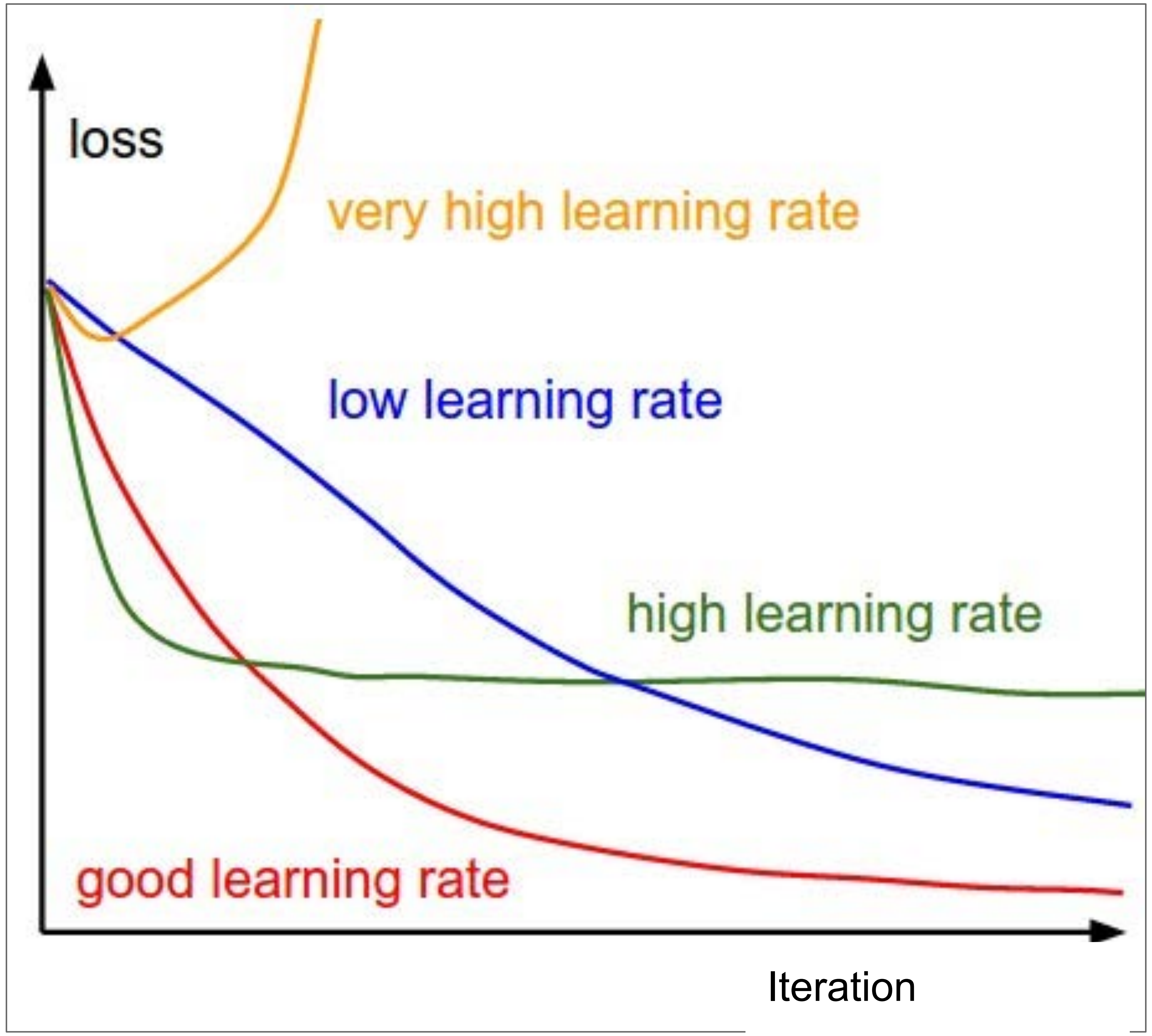
SGD

SGD+Momentum

RMSProp

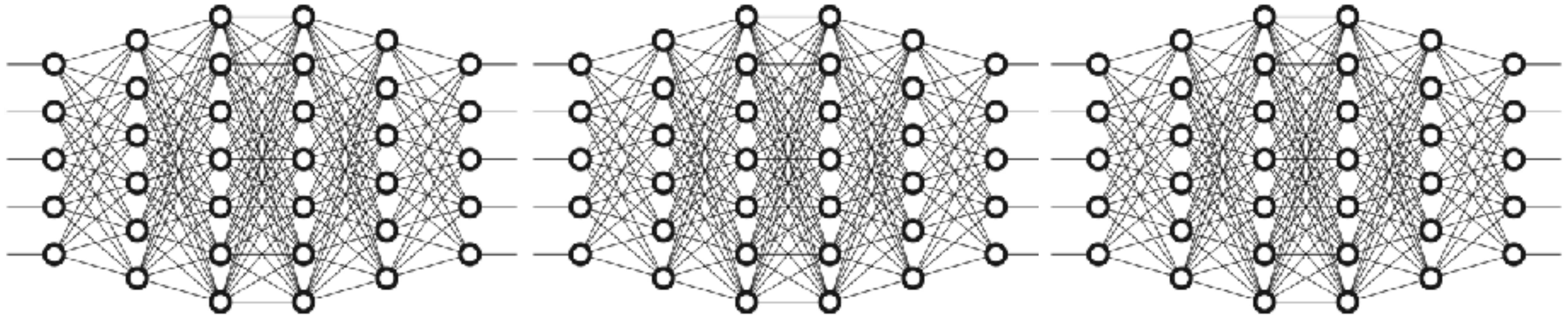
Adam

Learning rate: hyperparameter



Based on slides for [Stanford cs231n] by Li, Jonson, and Young. Modified and reused with permission

CPSC 425: Computer Vision



Lecture 20: Neural Networks 1

Menu for Today

Topics:

- **Neural Networks** introduction
- **Activation functions** softmax, relu
- **2-layer** fully connected net
- **Backprop** intro

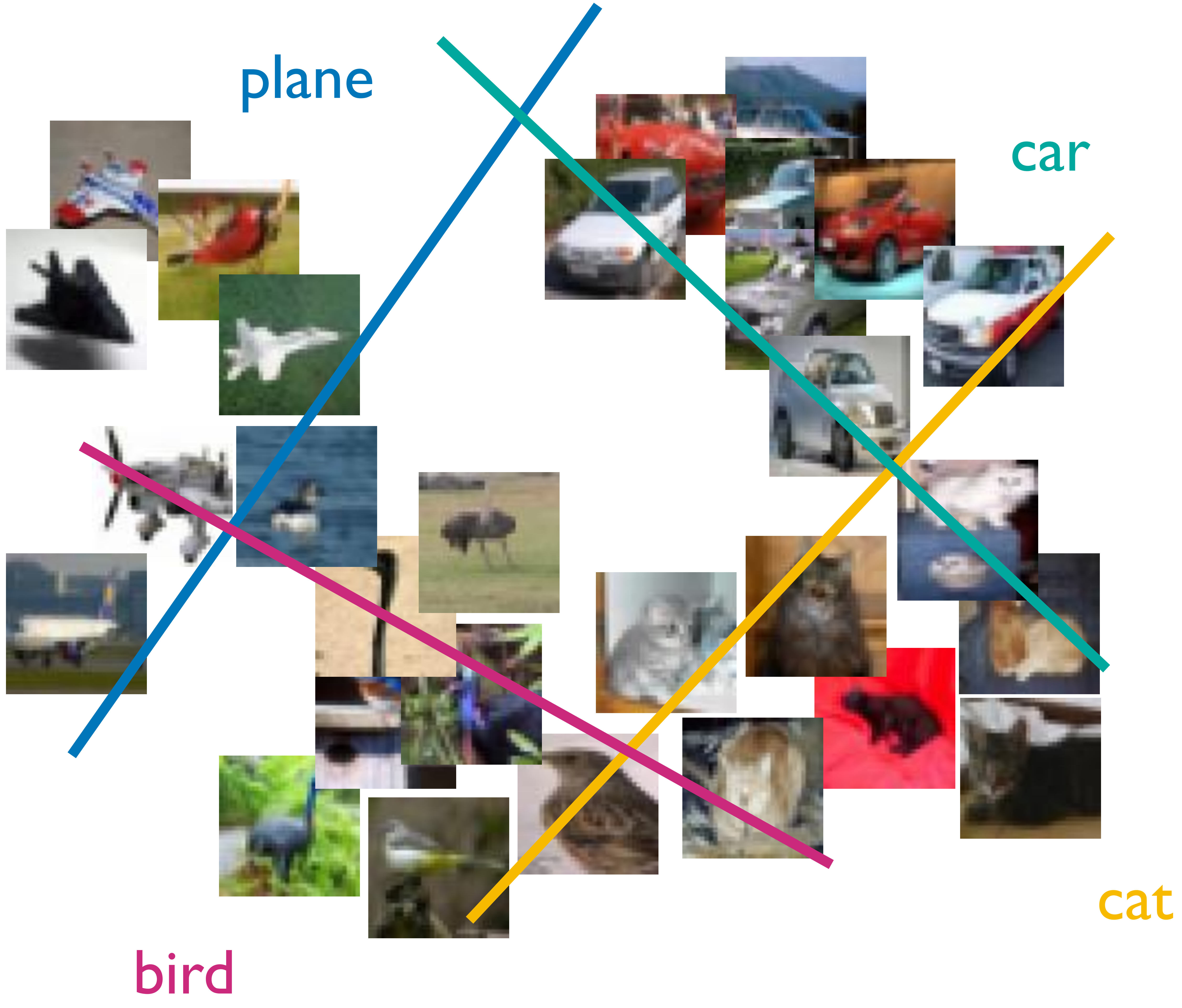
Readings:

- **Today's** Lecture: Szeliski 5.1.3, 5.3-5.4, Justin Johnson Michigan EECS 498/598

Reminders:

- **Assignment 5**: due Apr 3rd
- **Quiz 6** April 7th
- **Assignment 6**: due Apr 10th <— watch out!

Linear Classification



vectorized input image

$$\begin{bmatrix} — & w_1^T & — \\ — & w_2^T & — \\ — & w_3^T & — \\ — & w_4^T & — \\ & \cdots & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \cdots \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ \cdots \end{bmatrix}$$

learned 1 vs all classifiers

one-hot target

Softmax + Logistic Outputs

- Linear regression to one-hot targets is a bit strange..
- Output could be very large, and scores >>1 are penalised even for the correct class, likewise for scores << 1 for incorrect
- How about restricting output scores to 0-1?

19.1

Softmax + Cross Entropy

- What is the gradient of the softmax linear classifier?
- We could use L2 loss, but we'll use cross entropy instead
- This has a sound motivation — it is a measure of the difference between probability distributions
- It also leads to a simple update rule

19.2

Note: $\dfrac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$

Try yourself!

Linear + Softmax Regression

- We found the following gradient descent update rule

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \alpha(\underbrace{\mathbf{h}}_{\text{prediction}} - \underbrace{\mathbf{t}}_{\text{targets}})\underbrace{\mathbf{x}^T}_{\text{data}}$$

- This applies to:

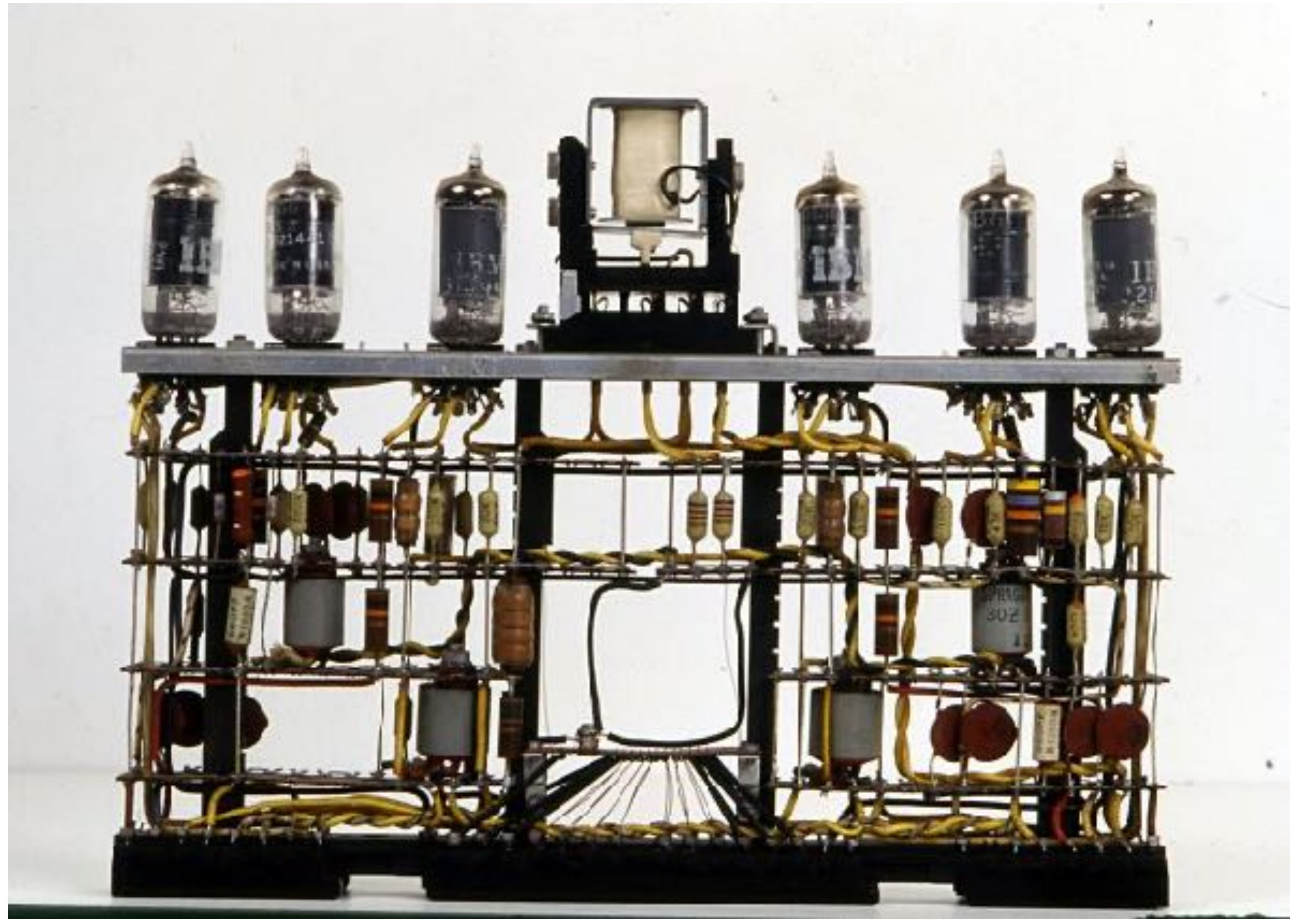
Linear regression	$\mathbf{h} = \mathbf{W}^T\mathbf{x}$	L2 loss
Softmax regression	$\mathbf{h} = \sigma(\mathbf{W}^T\mathbf{x})$	cross-entropy loss

- The same update rule with a binary prediction function

$$\mathbf{h} = \mathbb{1}_{\max}(\mathbf{W}^T\mathbf{x})$$

implements the multiclass Perceptron learning rule

History of the Perceptron



[I.B.M. Italia]

- This machine (IBM 704) was used by Frank Rosenblatt to implement the perceptron in 1958
- Based on his statements, the New York Times reported it as: "the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence."

2-class Perceptron Classifier

- Classification function is

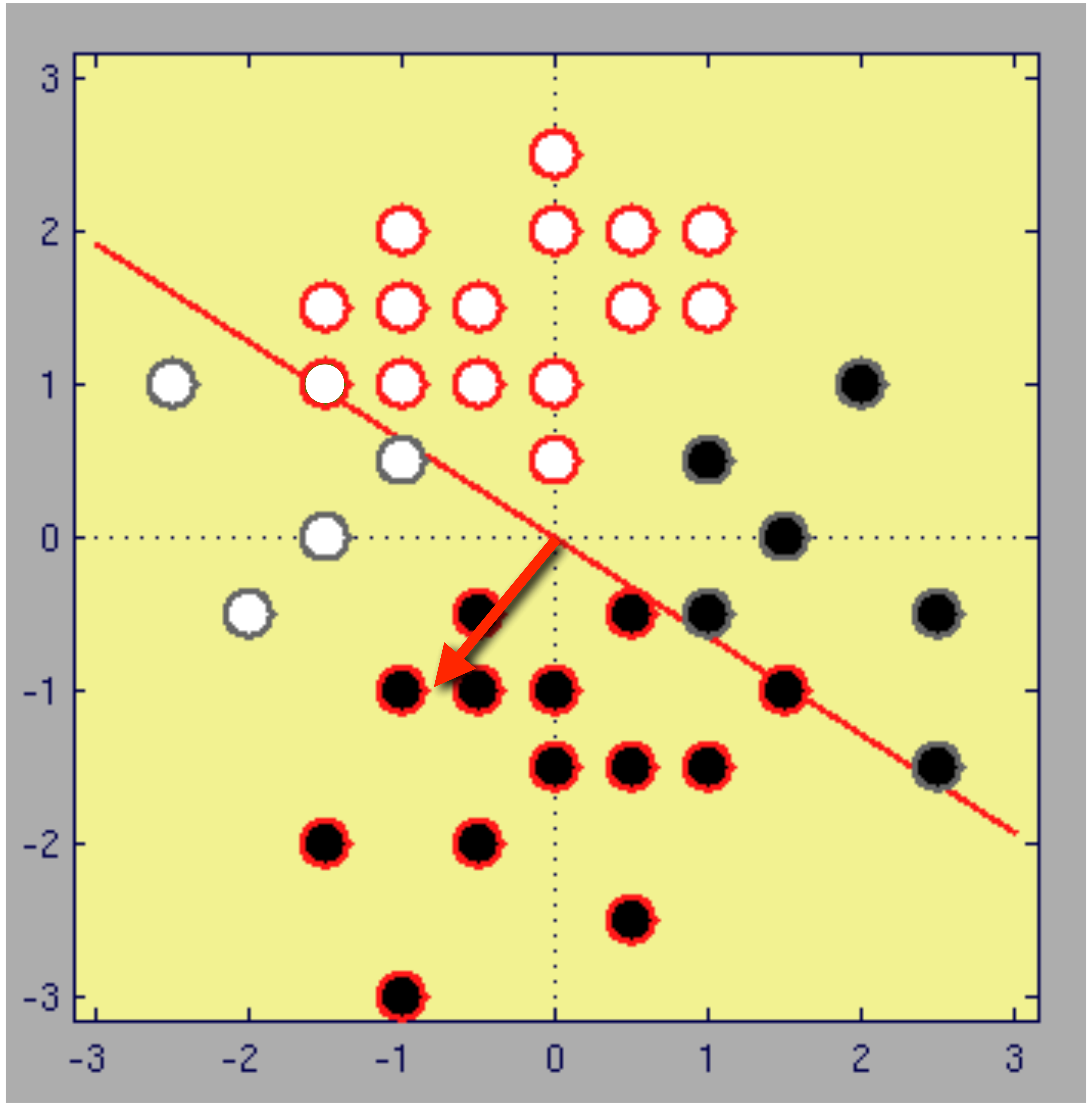
$$\hat{y} = \mathrm{sign}(\mathbf{w}^T \mathbf{x})$$

- Linear function of the data (x) followed by 0/1 activation

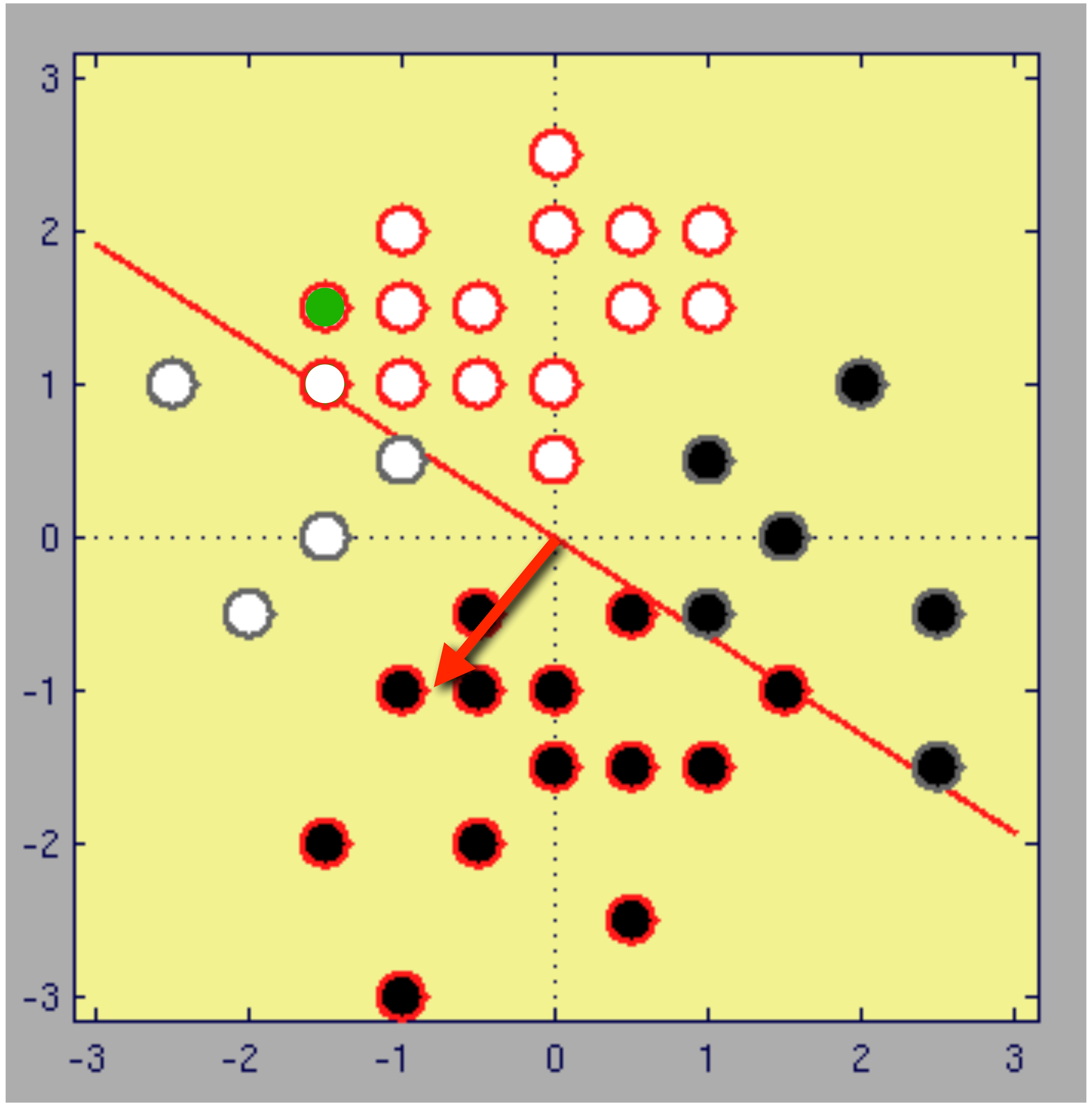
- Update rule: present data x
  - if correctly classified, do nothing
  - if incorrectly classified, update the weight vector

$$\mathbf{w}_{n+1} = \mathbf{w}_n + y_i \mathbf{x}_i$$

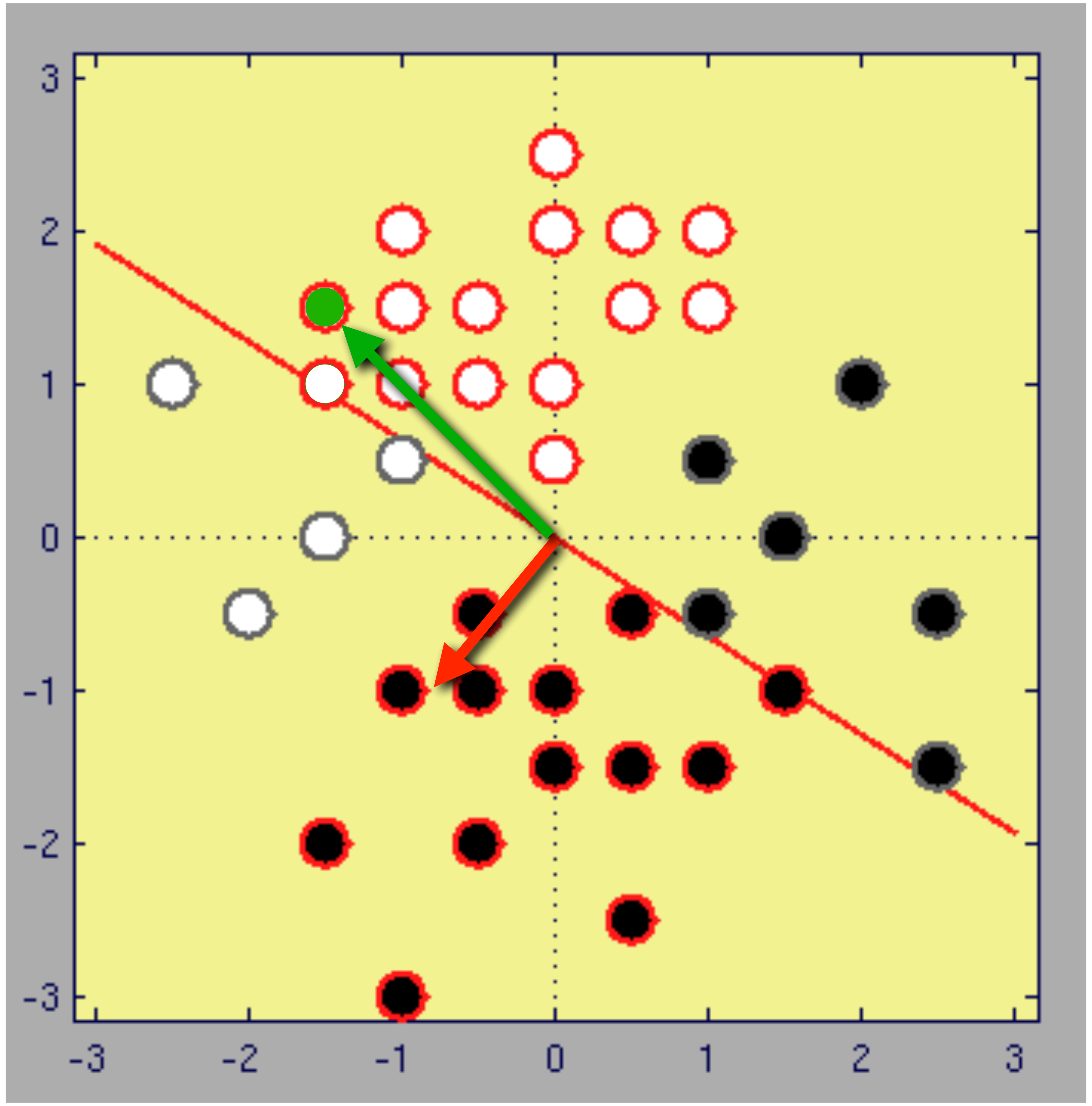
Example of Perceptron Learning



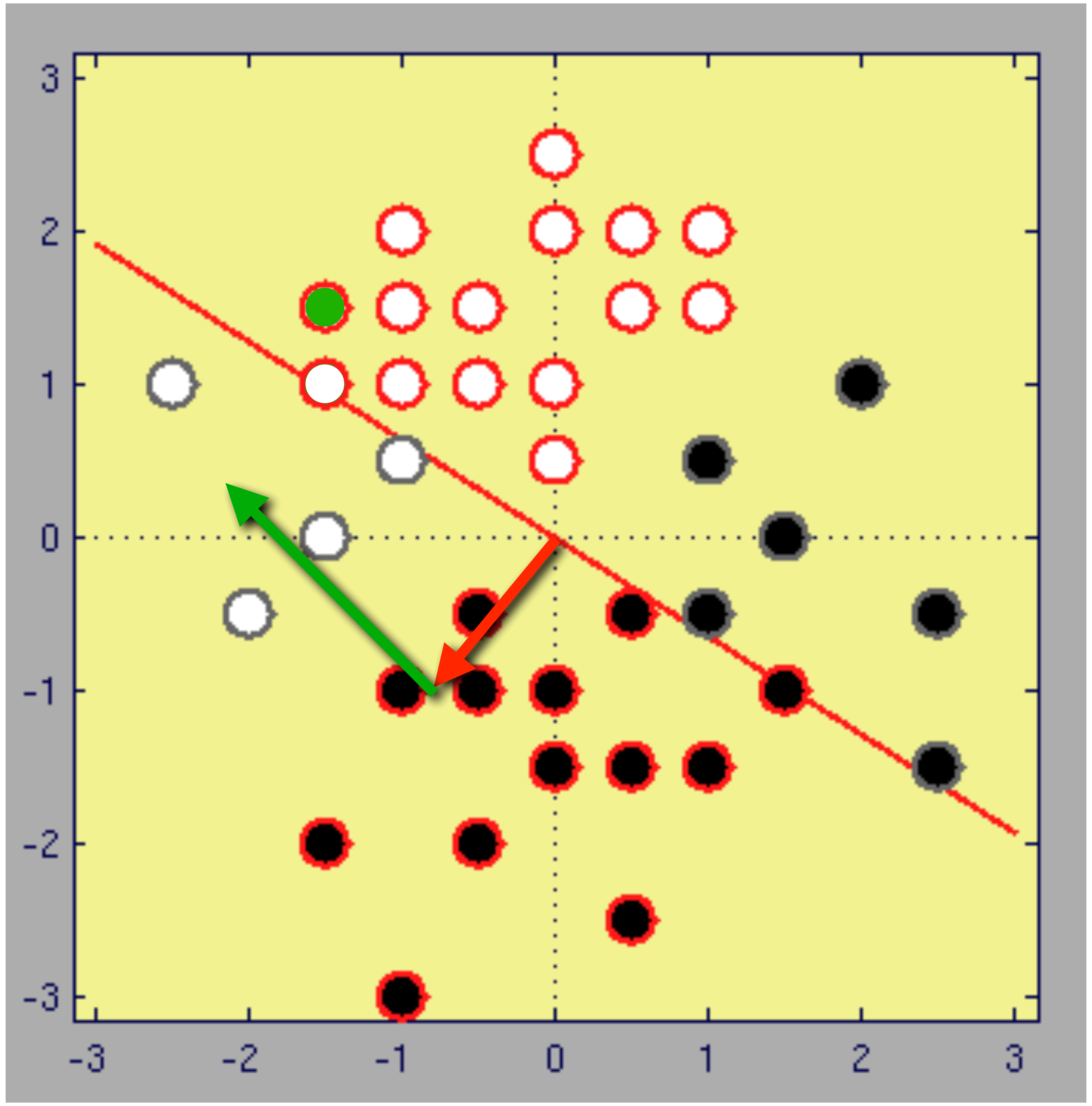
Example of Perceptron Learning



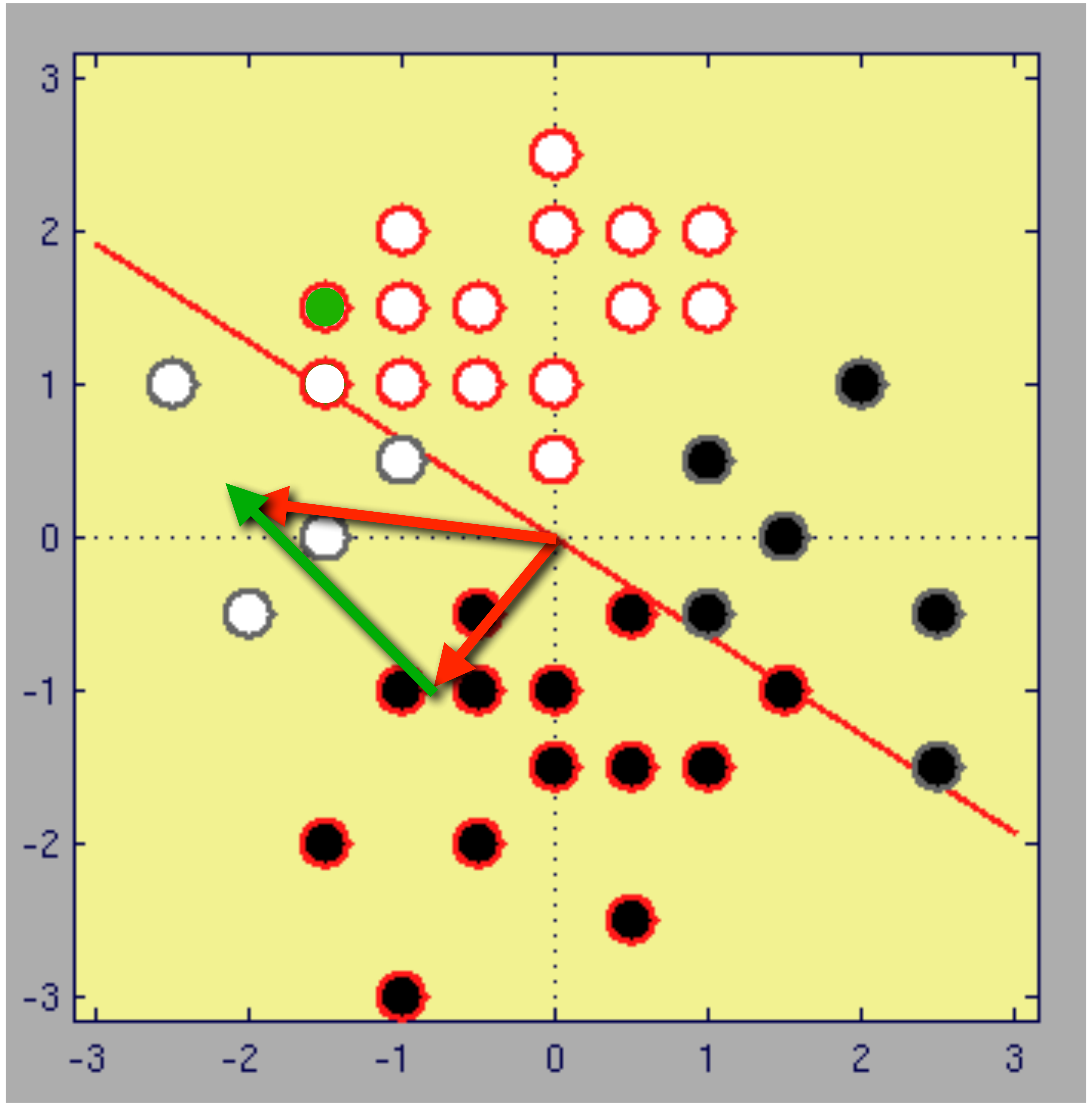
Example of Perceptron Learning



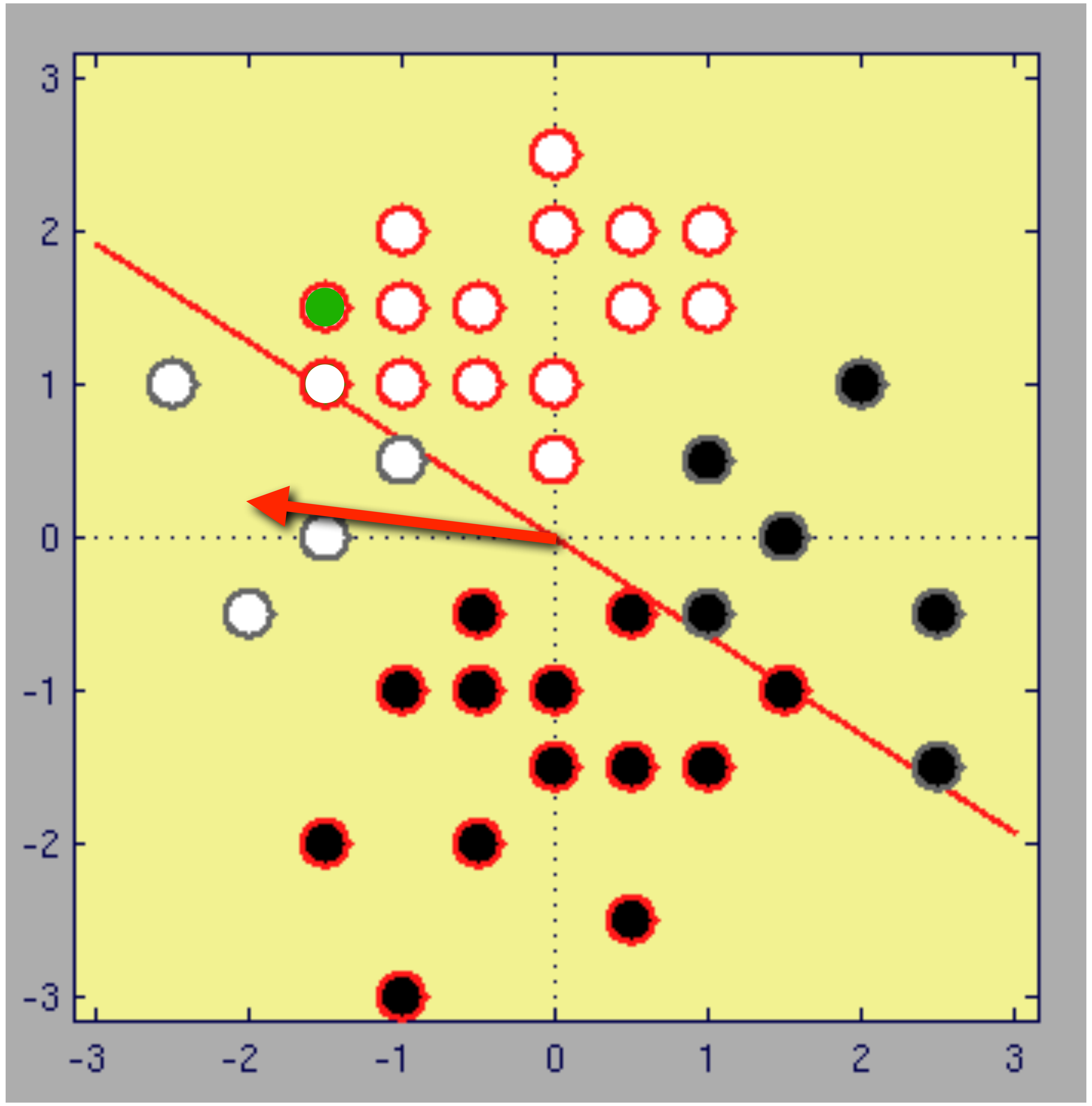
Example of Perceptron Learning



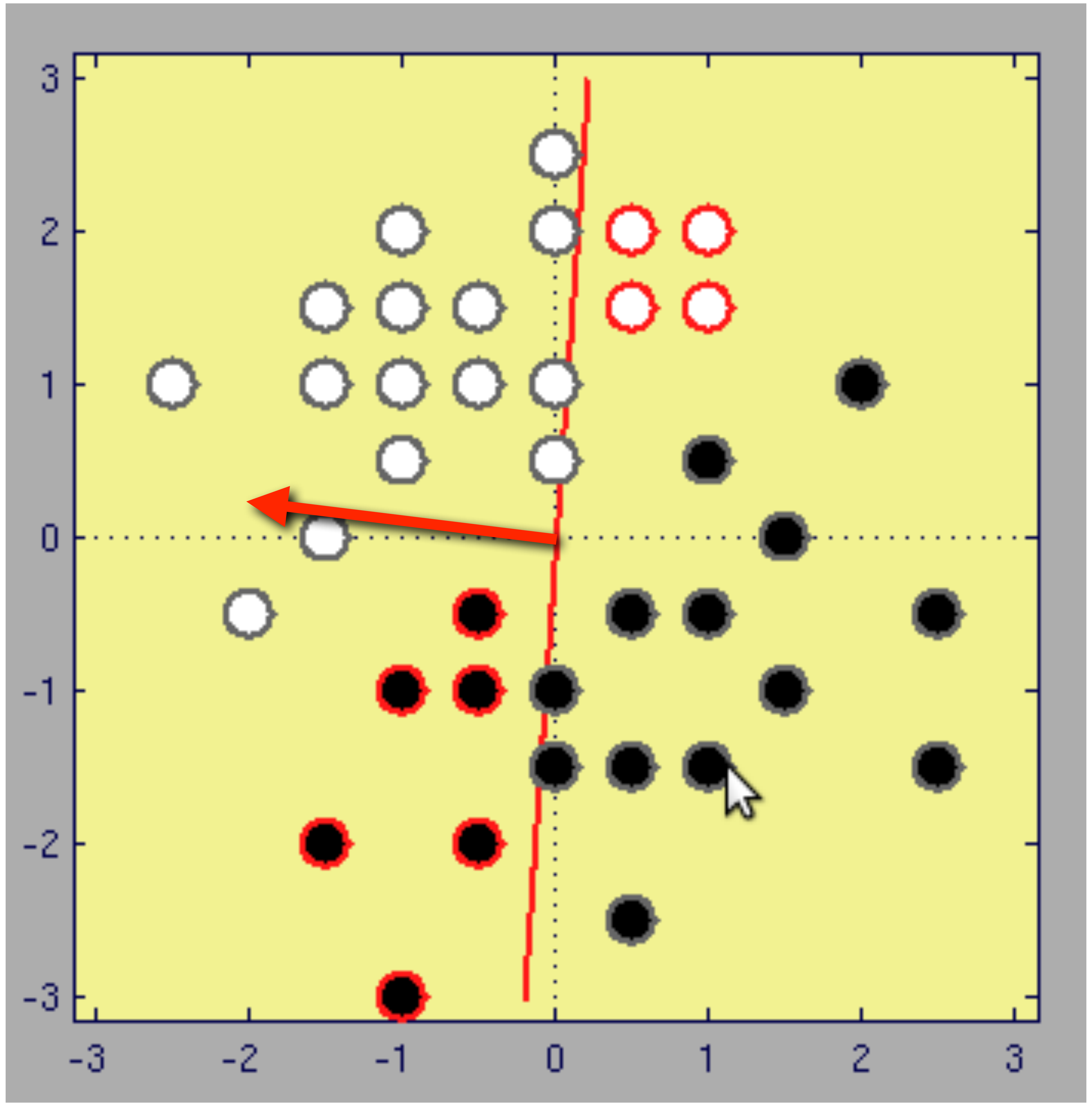
Example of Perceptron Learning



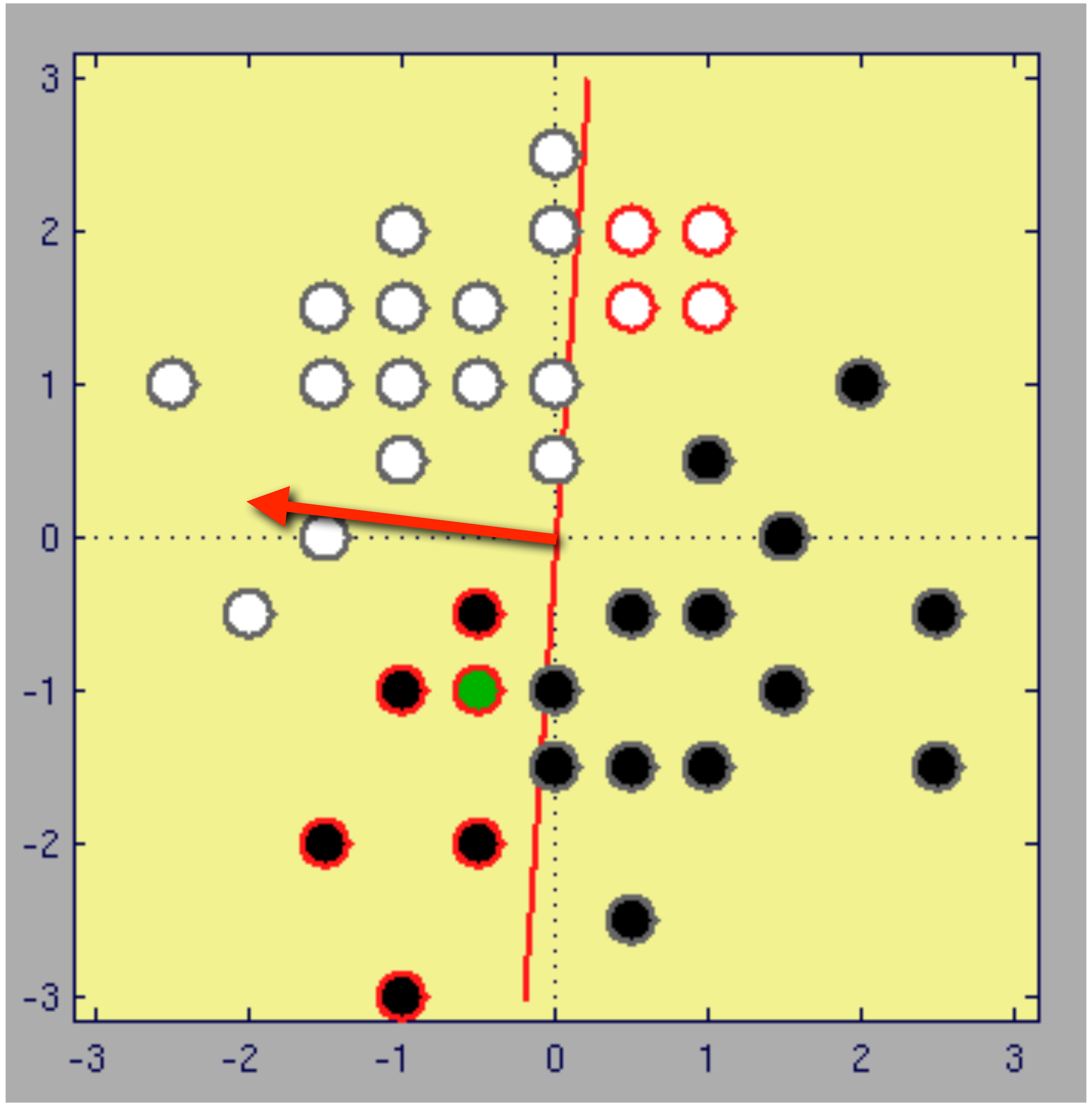
Example of Perceptron Learning



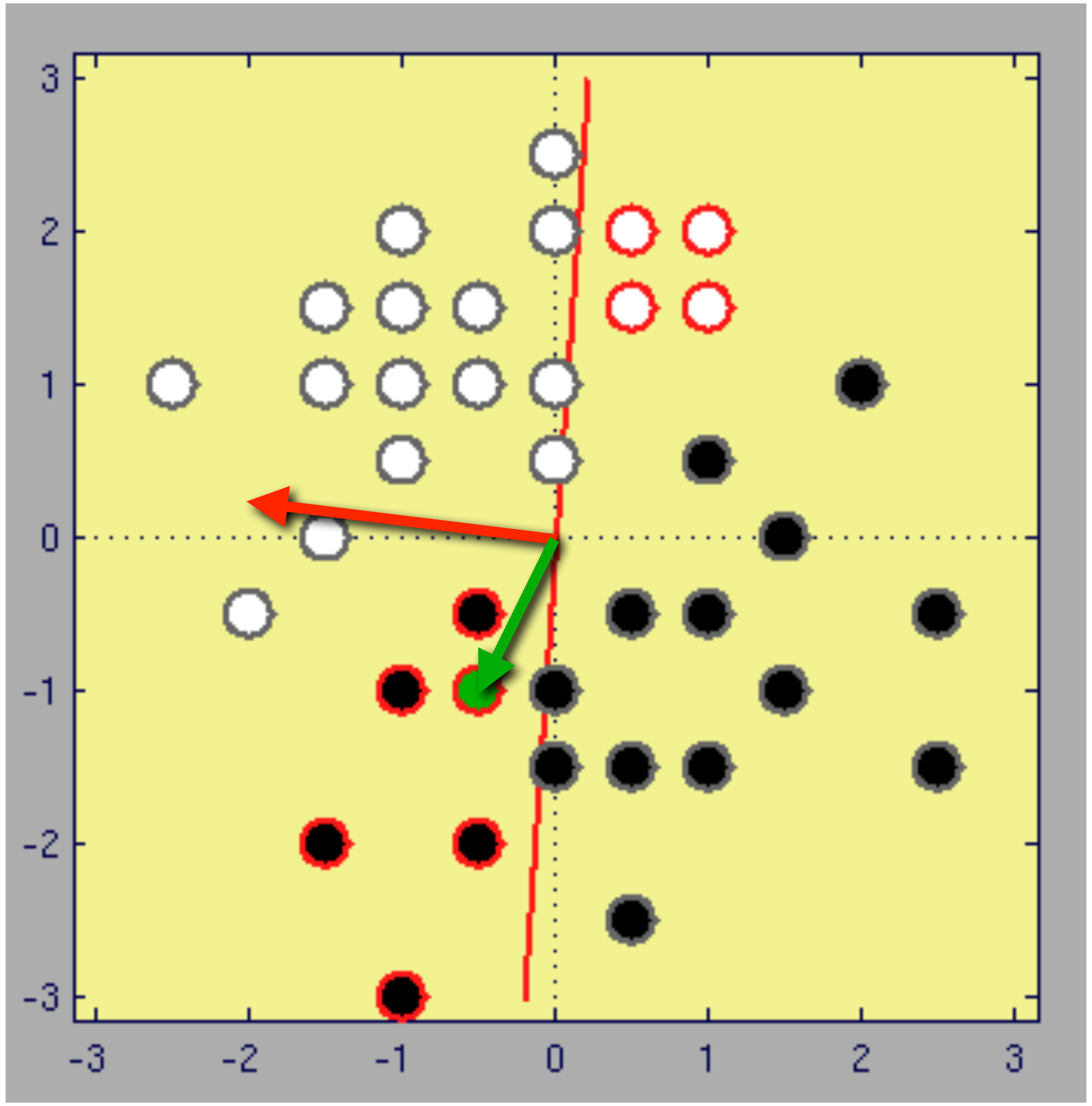
Example of Perceptron Learning



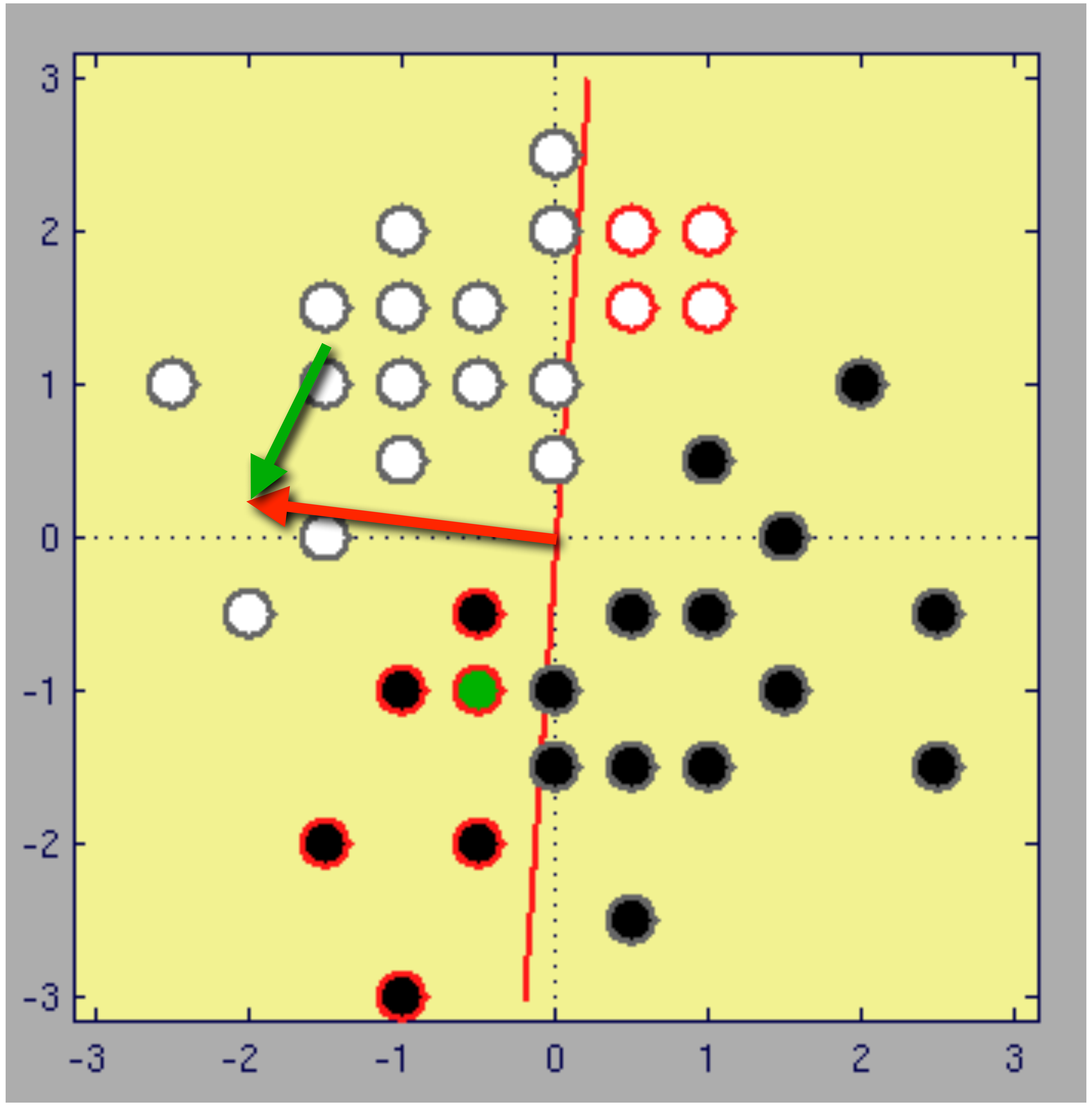
Example of Perceptron Learning



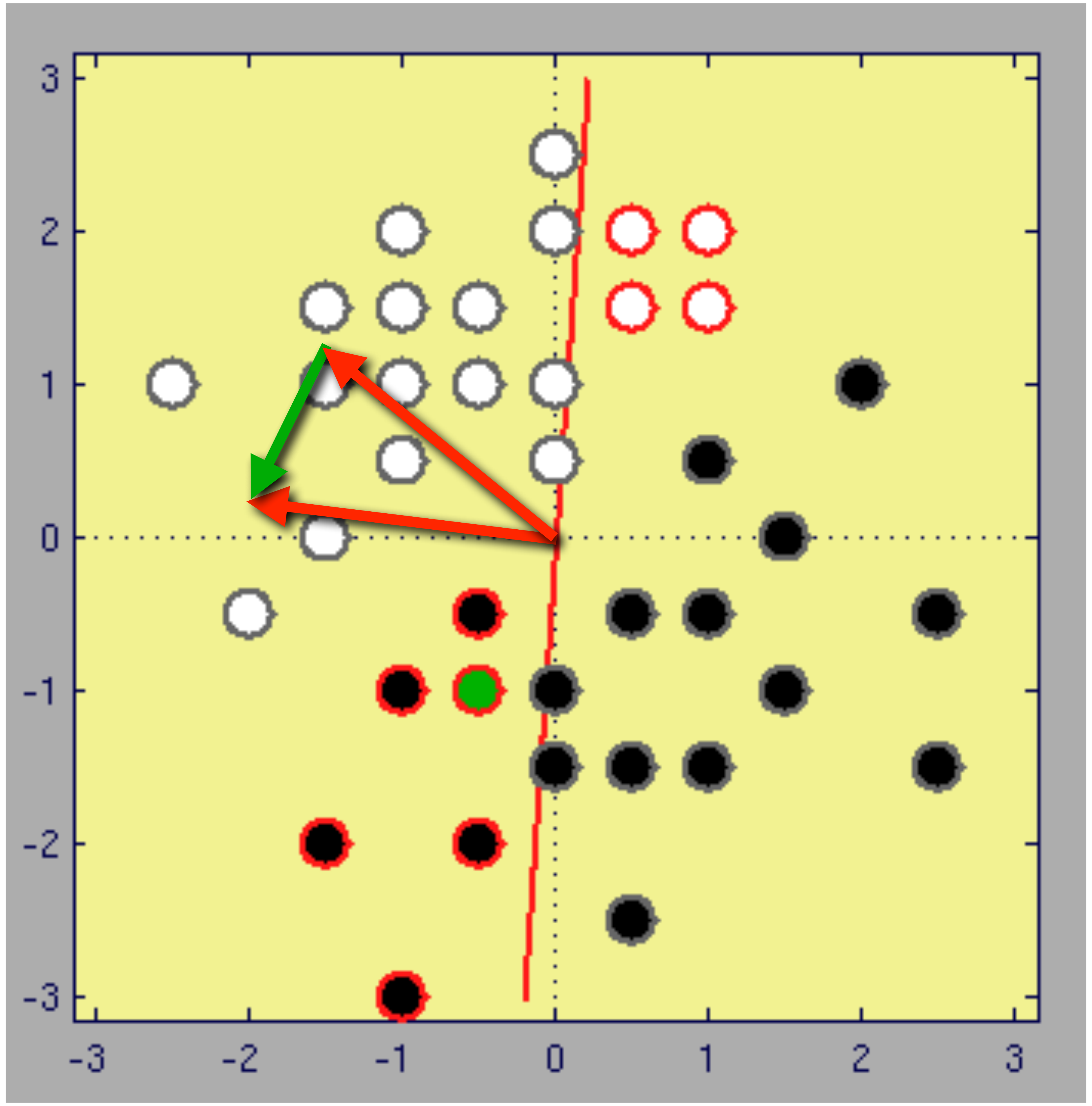
Example of Perceptron Learning



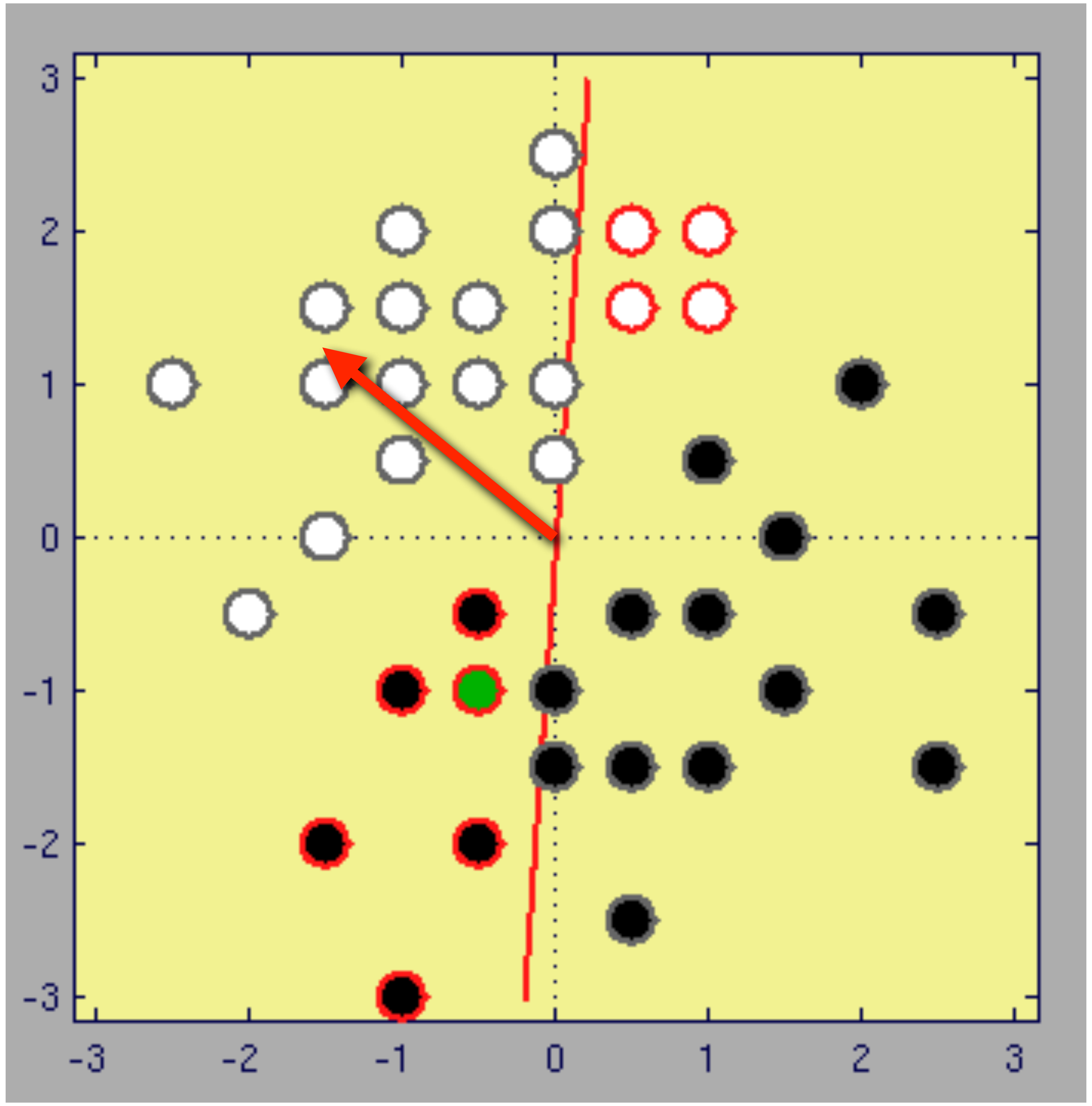
Example of Perceptron Learning



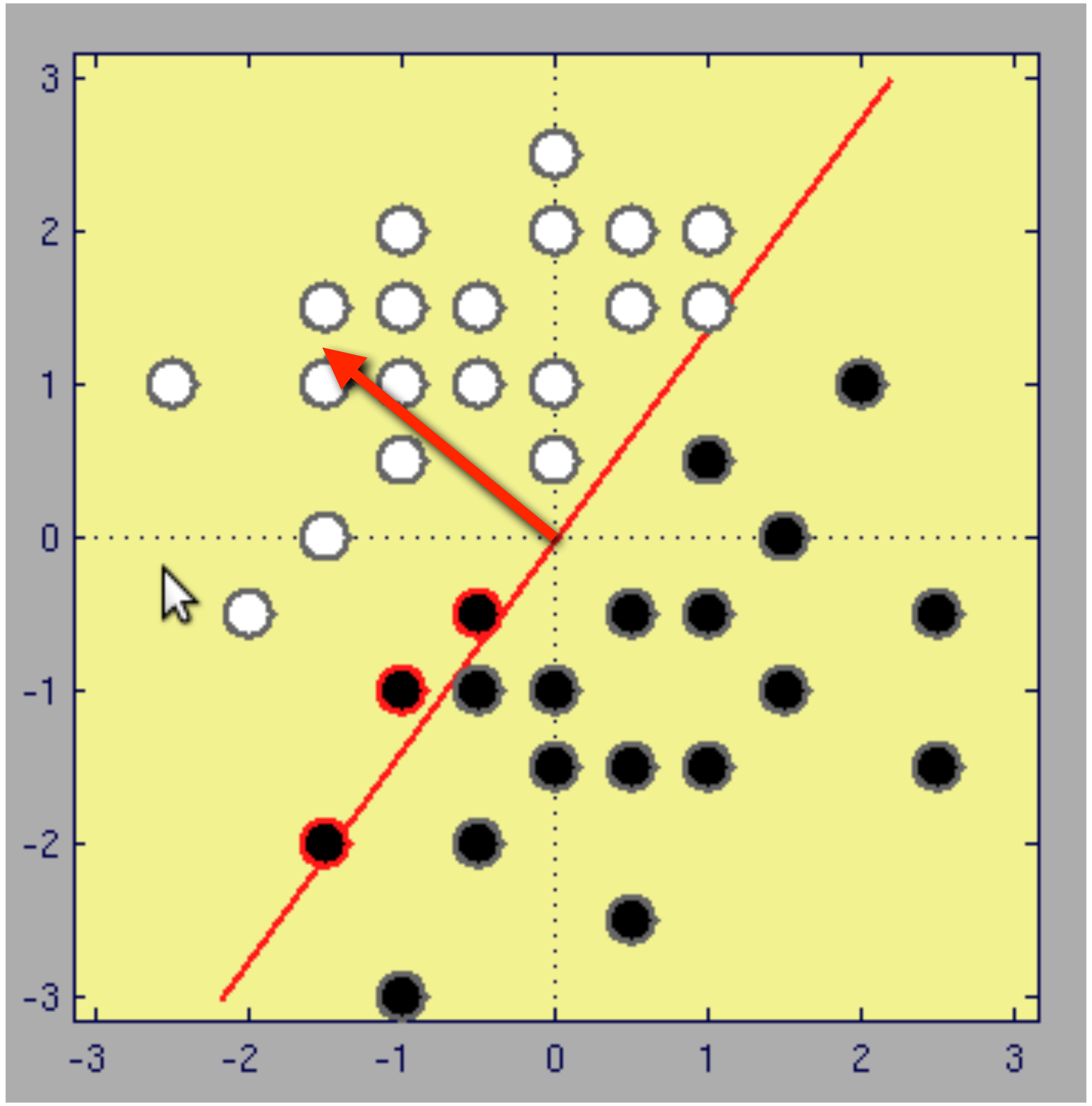
Example of Perceptron Learning



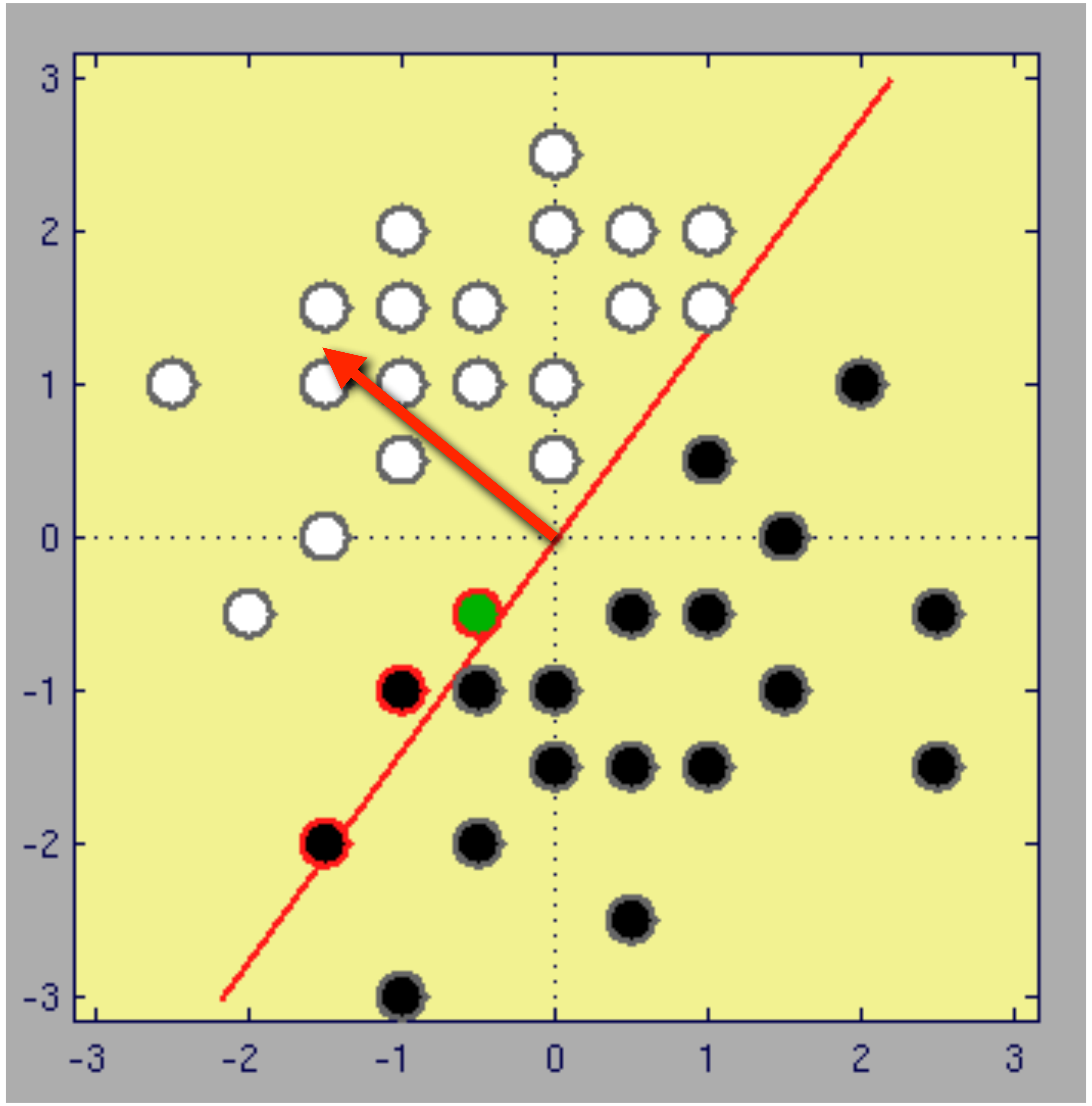
Example of Perceptron Learning



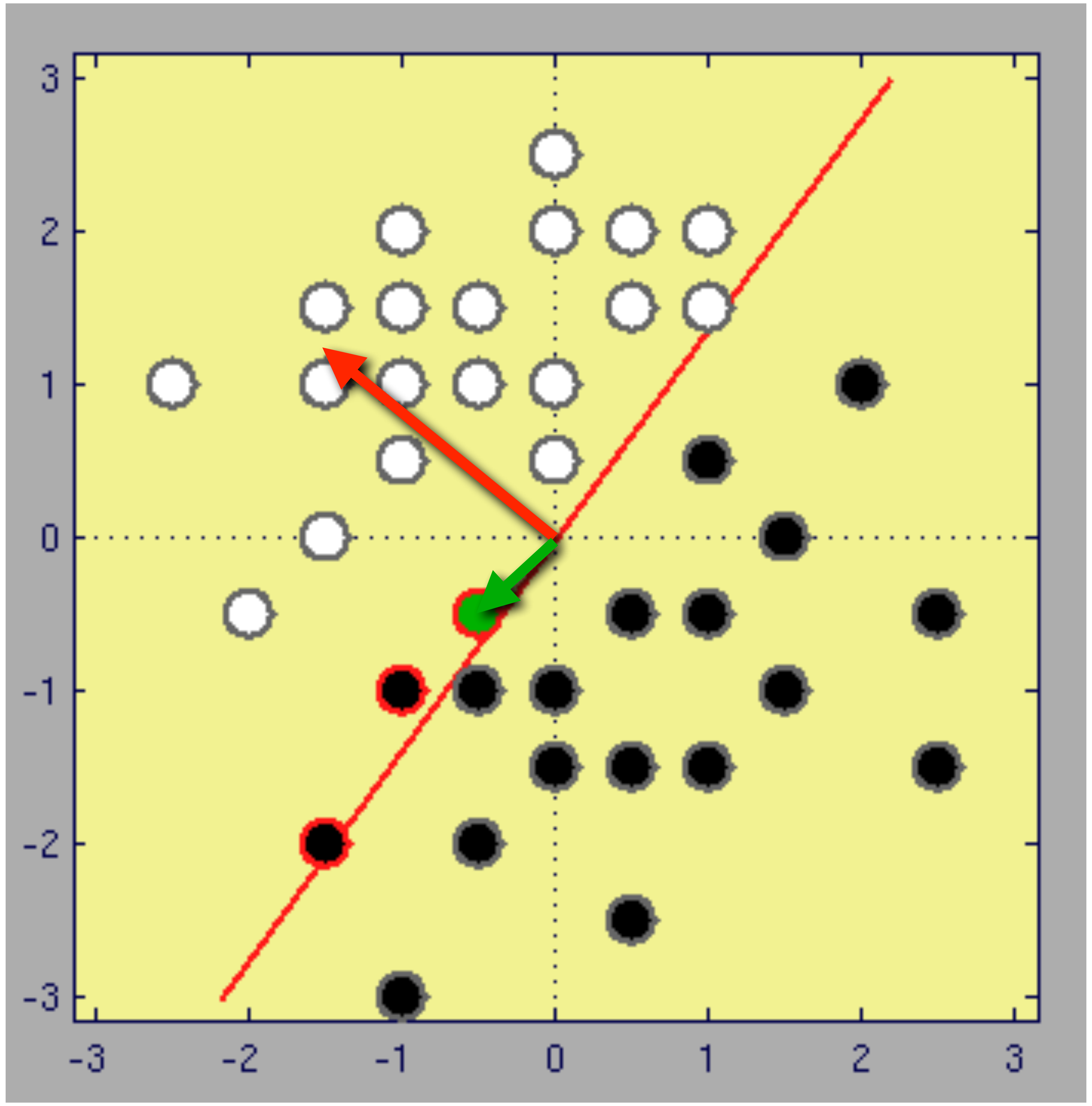
Example of Perceptron Learning



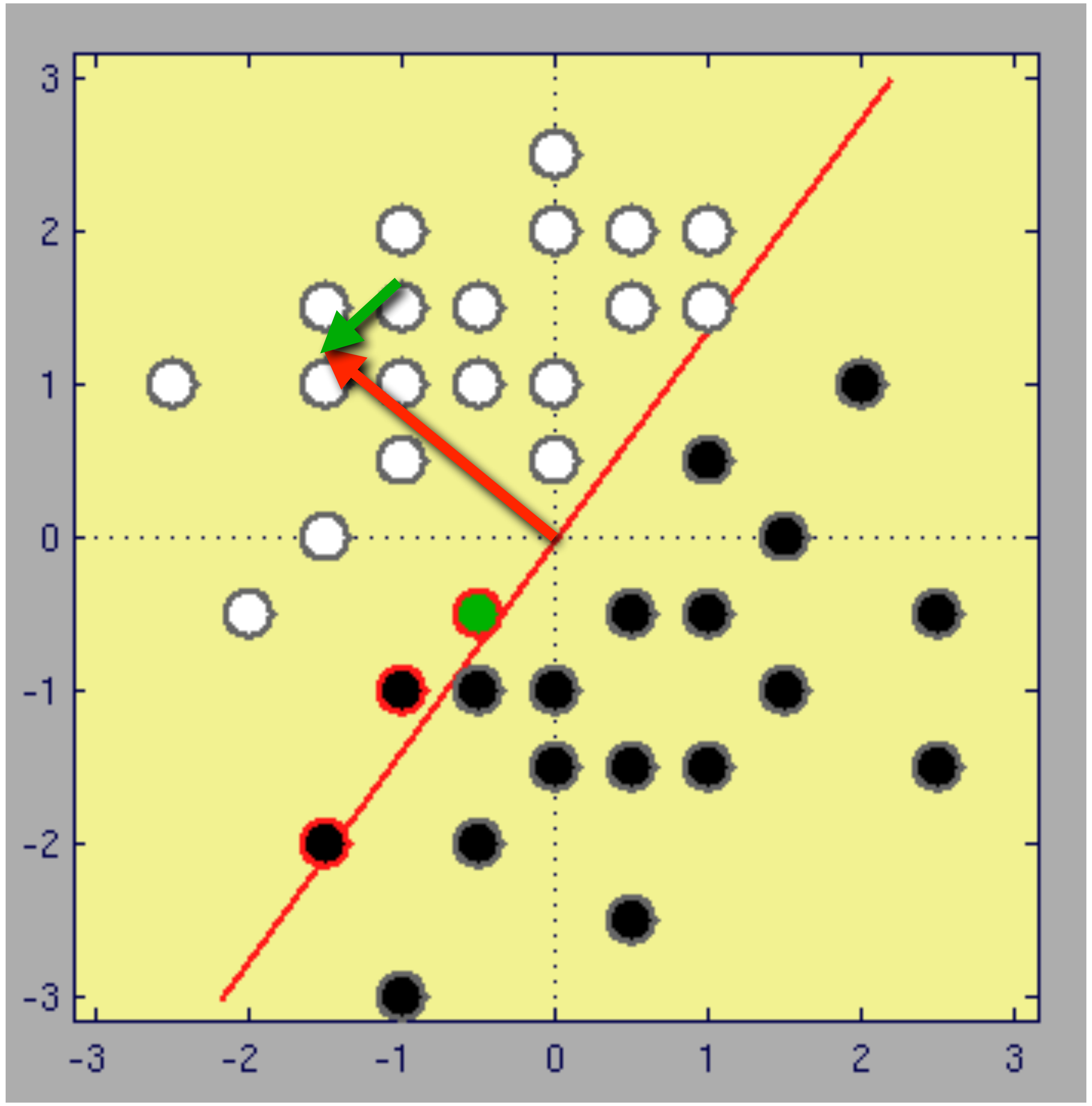
Example of Perceptron Learning



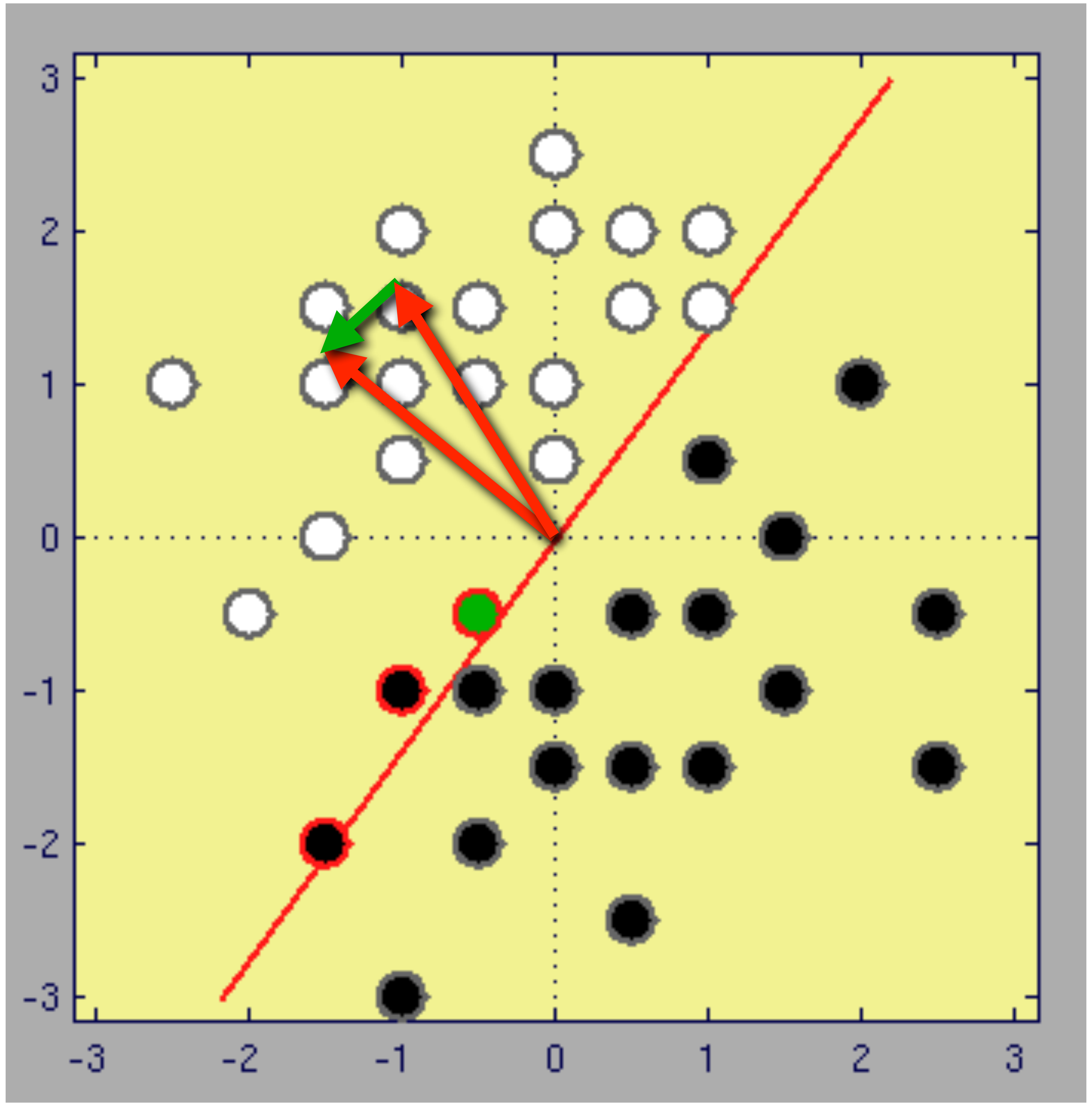
Example of Perceptron Learning



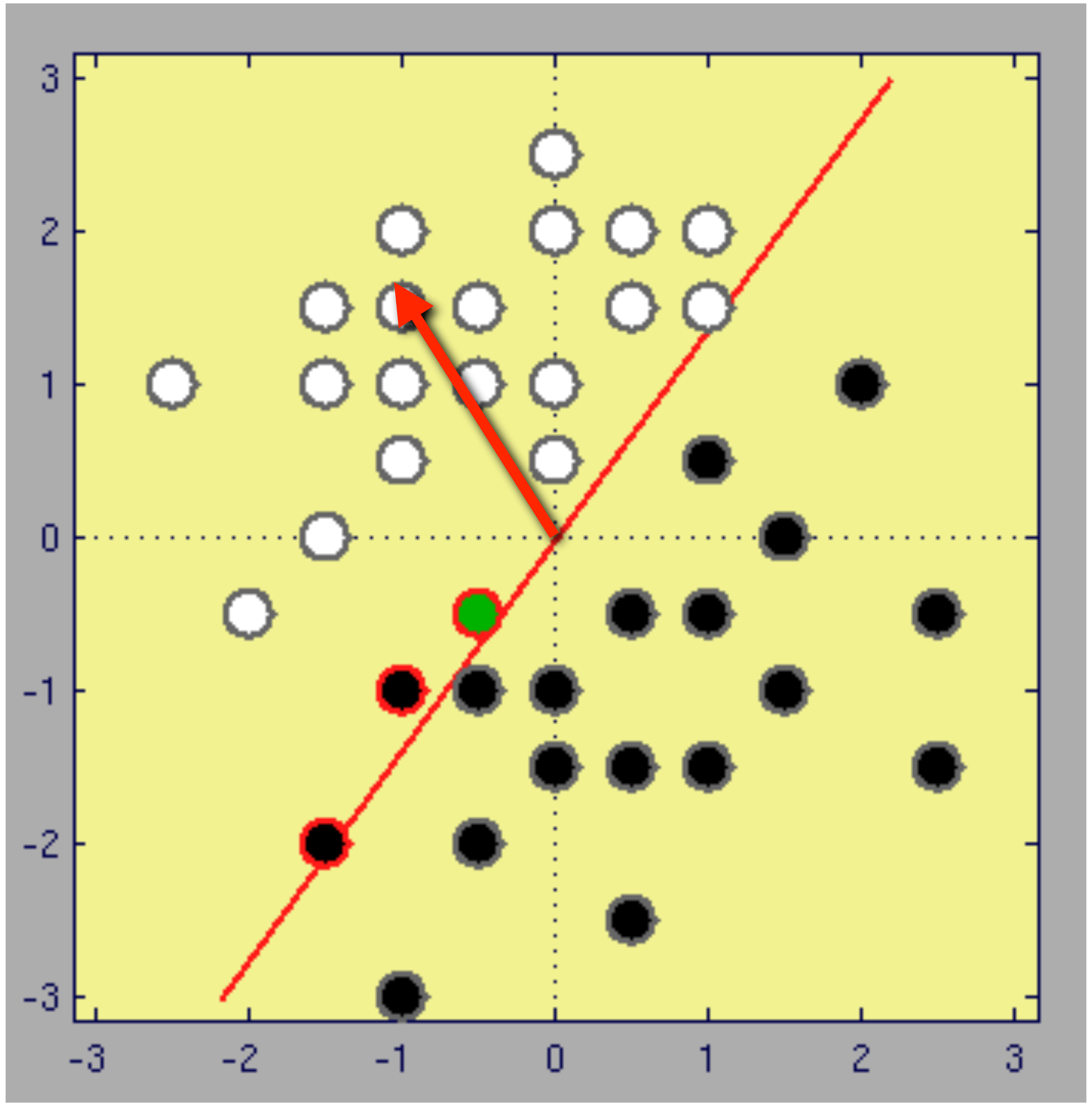
Example of Perceptron Learning



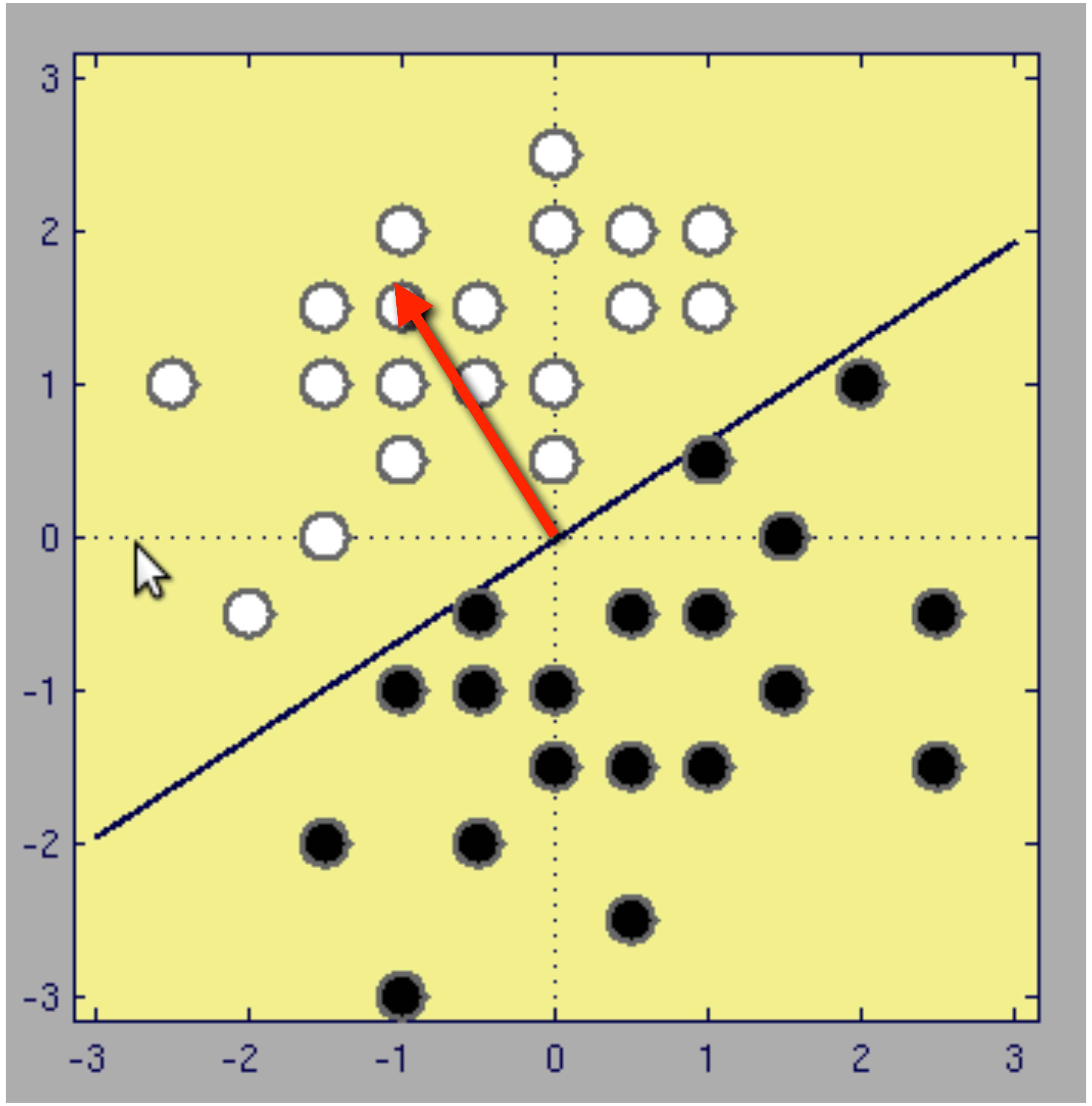
Example of Perceptron Learning



Example of Perceptron Learning

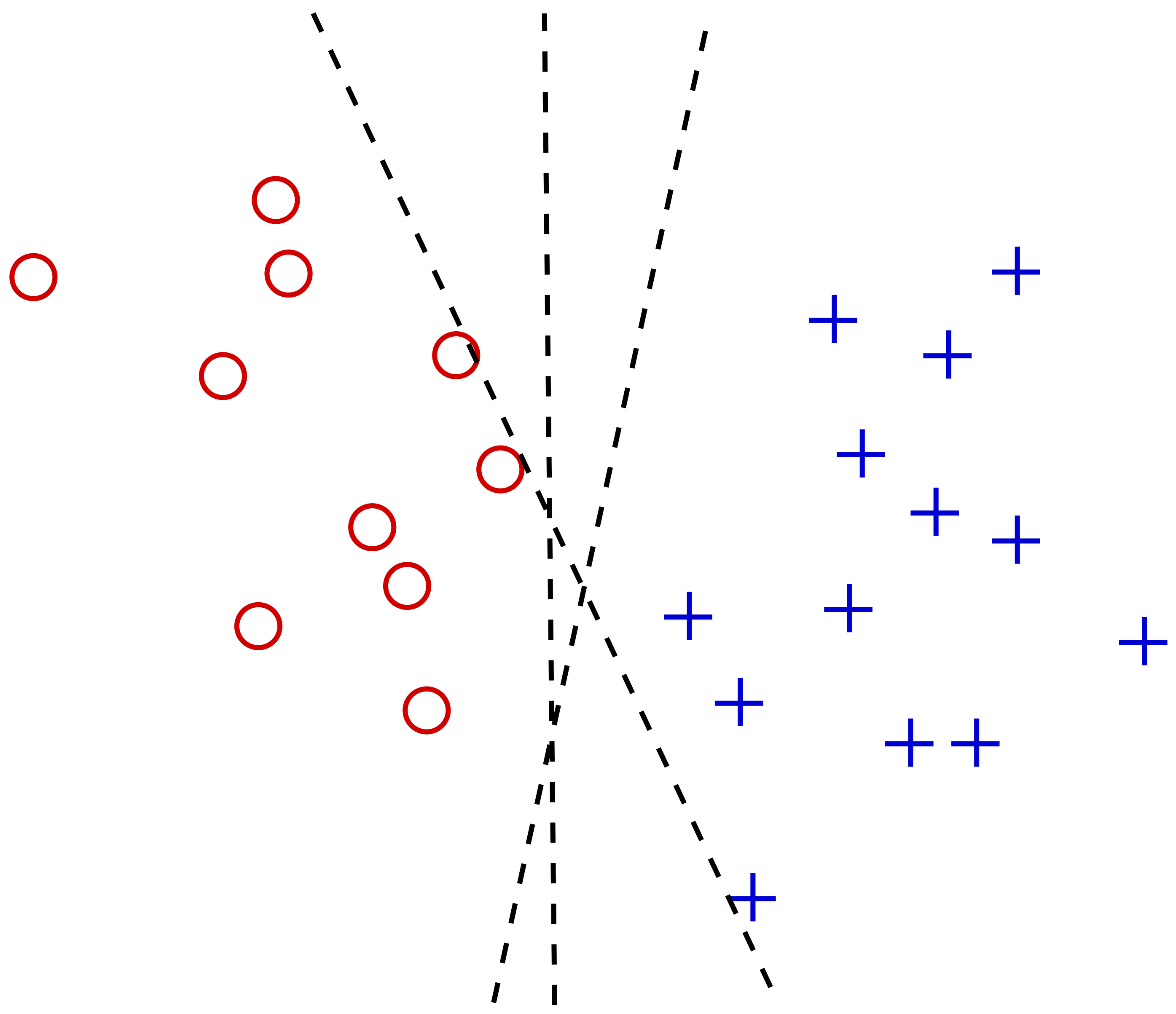


Example of Perceptron Learning

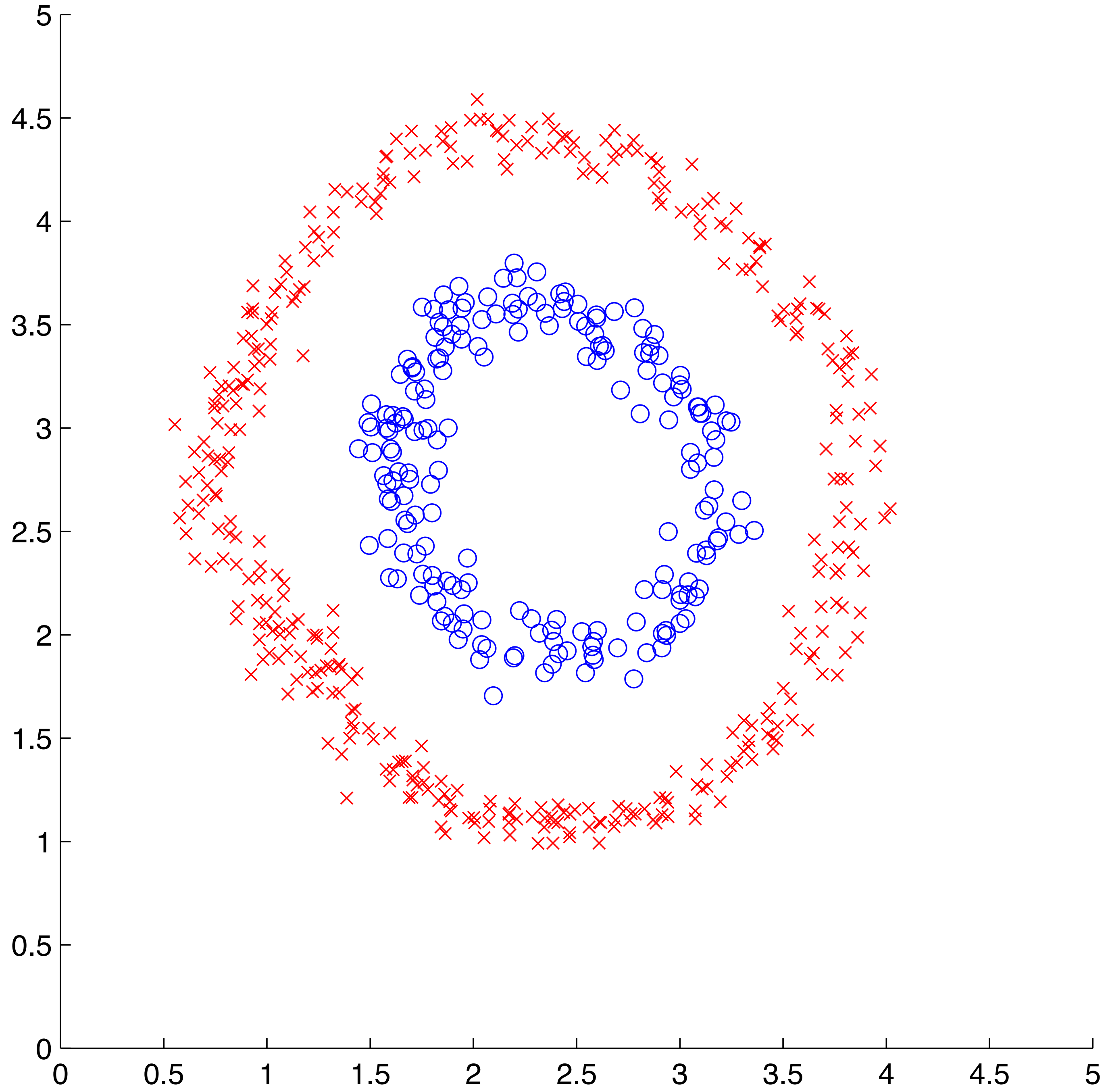


Perceptron Limitations

- Perceptrons + linear + softmax regressors are limited to data that are linearly separable, e.g.,



Linearly separable

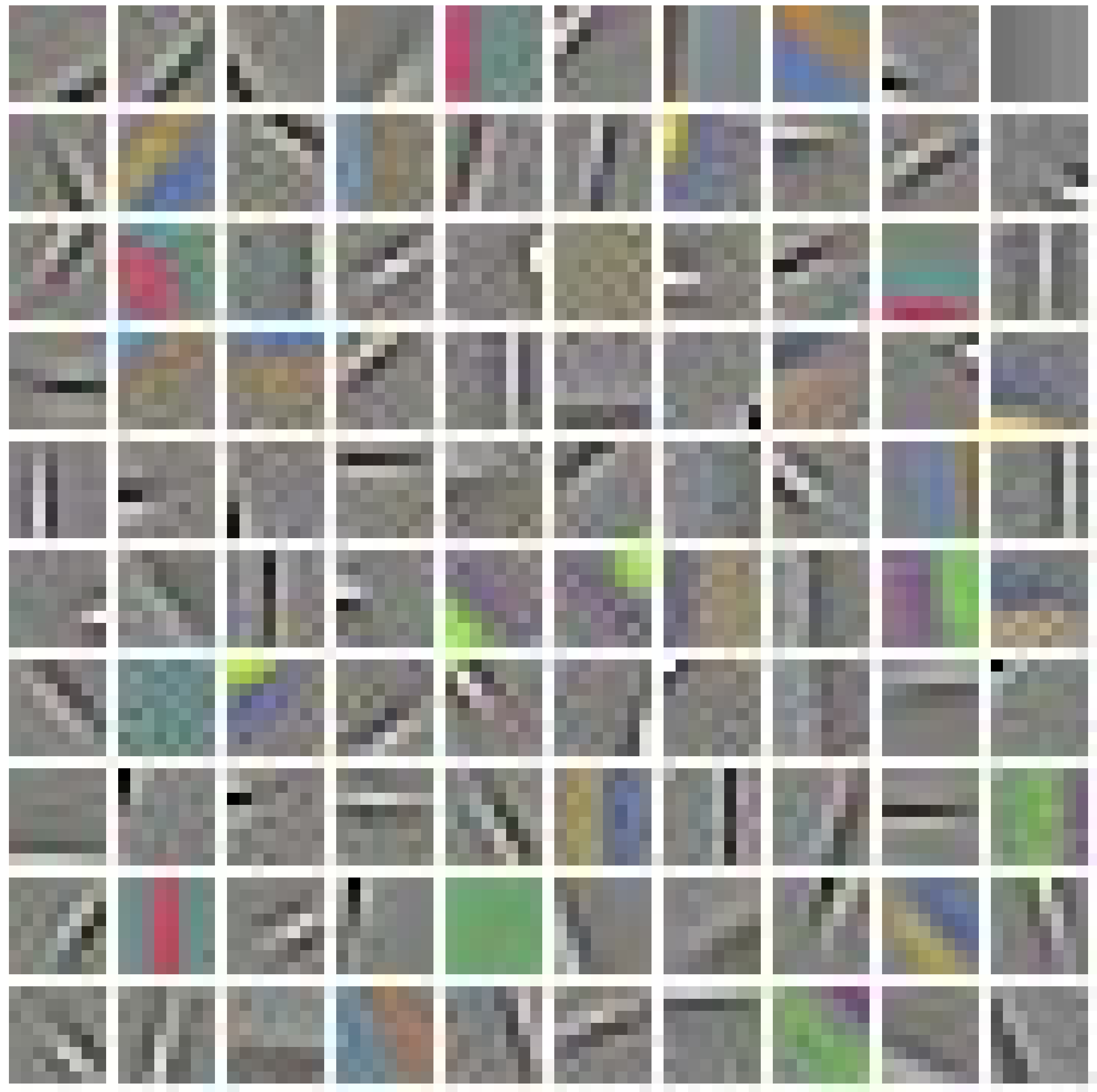


Not linearly separable

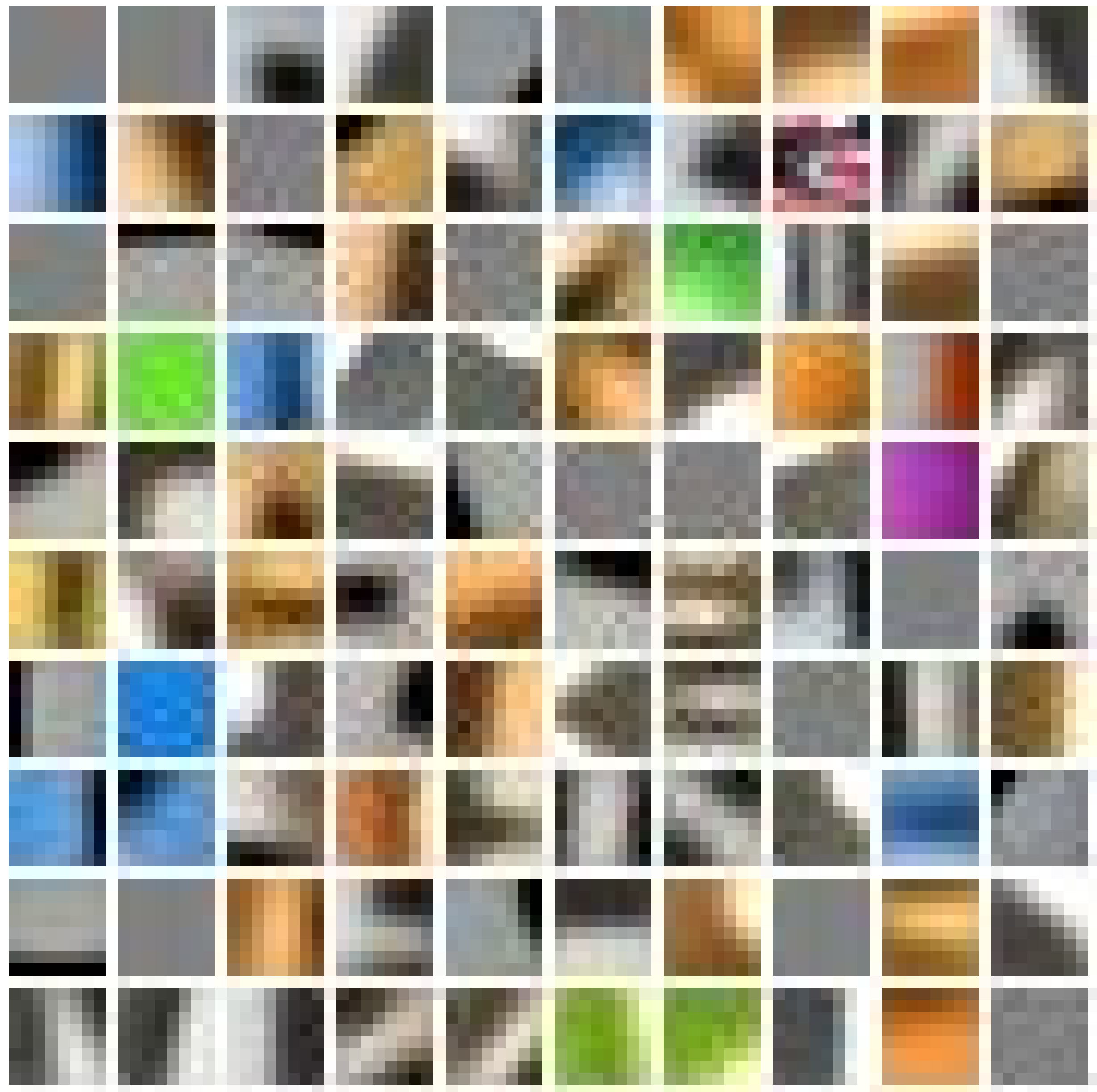
Could we extract features to make the data linearly separable?

CIFAR10 Feature Extraction

- So far, we used RGB pixels as the input to our classifier
- Feature extraction can improve results by a lot
- e.g., Coates et al. achieve 79.6% accuracy on CIFAR10 with a features based on k-means of whitened image patches



k-means, whitened

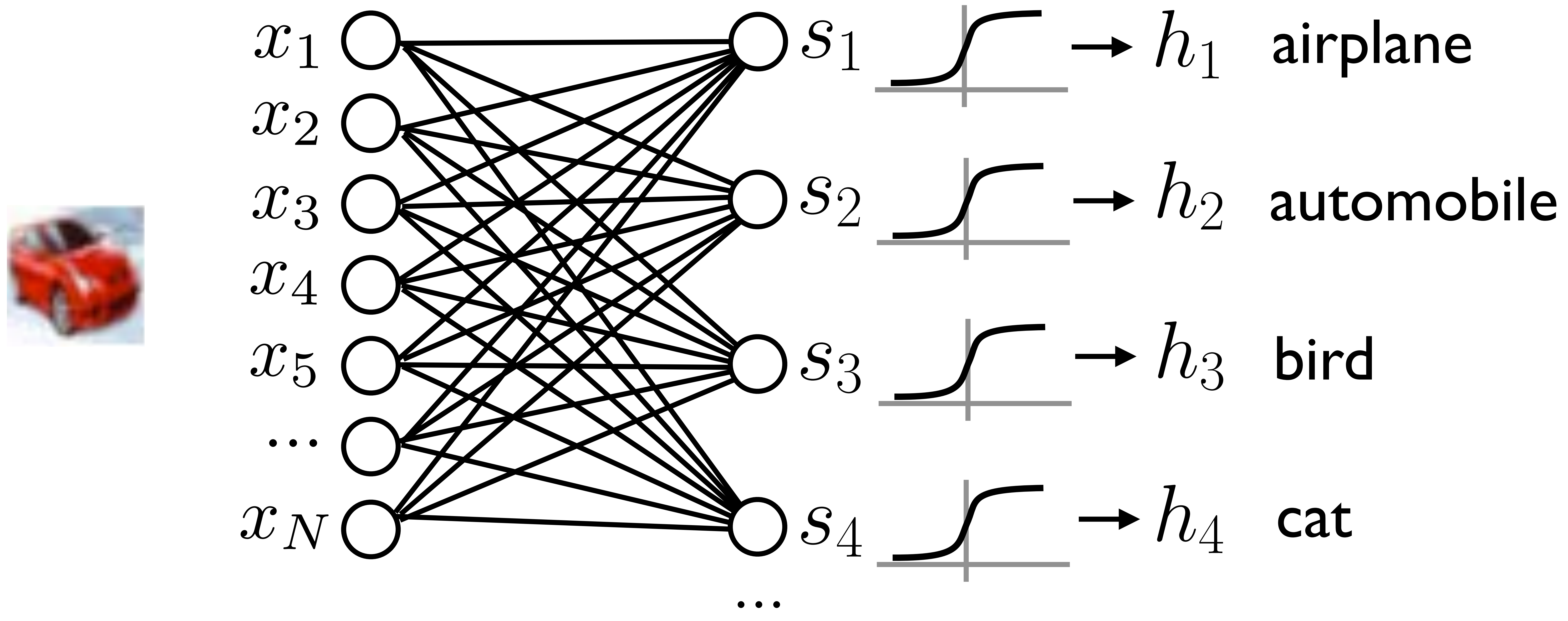


k-means, raw RGB

[Coates et al. 2011]

Linear = Fully Connected Layer

- Note that our linear matrix multiplication classifier is equivalent to a fully connected layer in a neural network

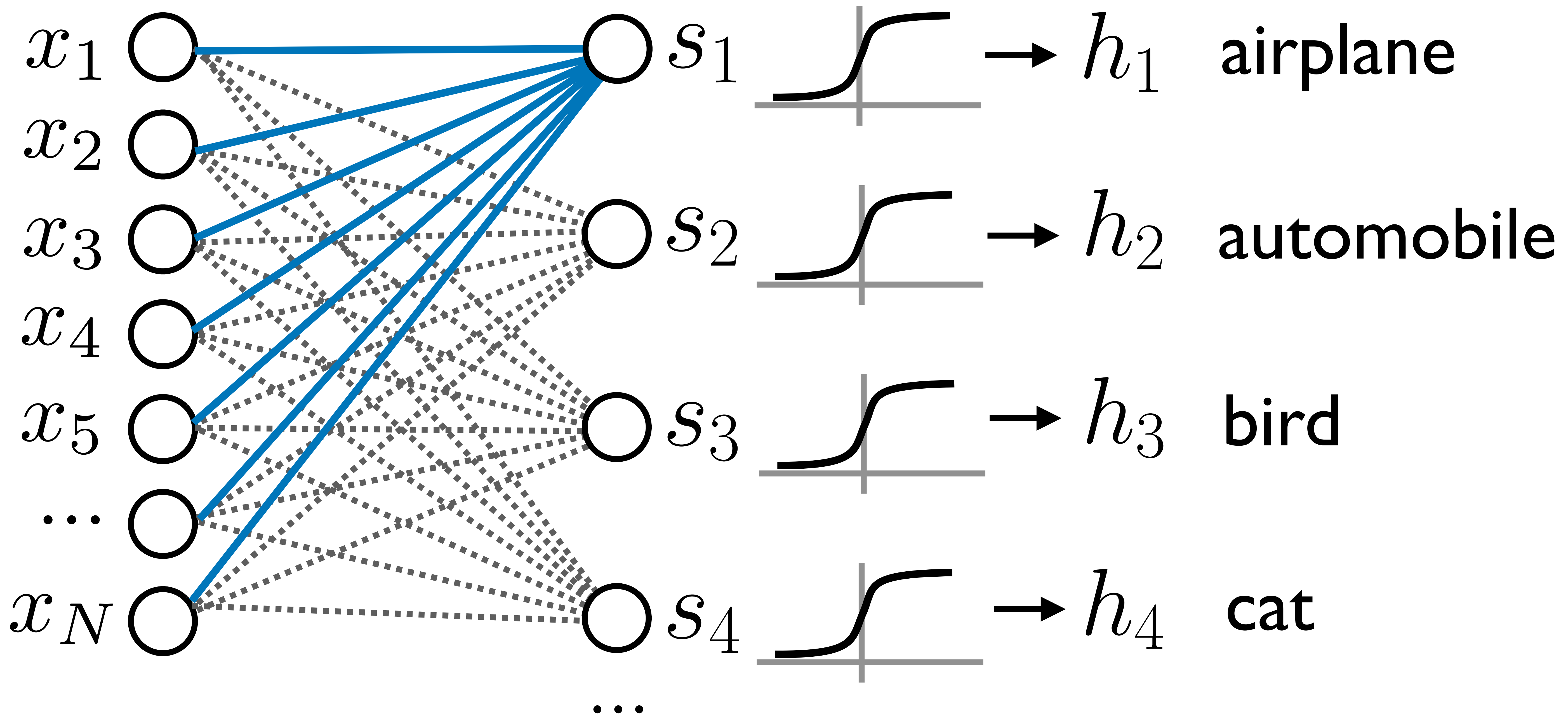


- Typically, we'll also add a bias term b

$$\mathbf{h} = \sigma(\mathbf{W}^T\mathbf{x} + \mathbf{b})$$

Linear = Fully Connected Layer

- Note that our linear matrix multiplication classifier is equivalent to a fully connected layer in a neural network

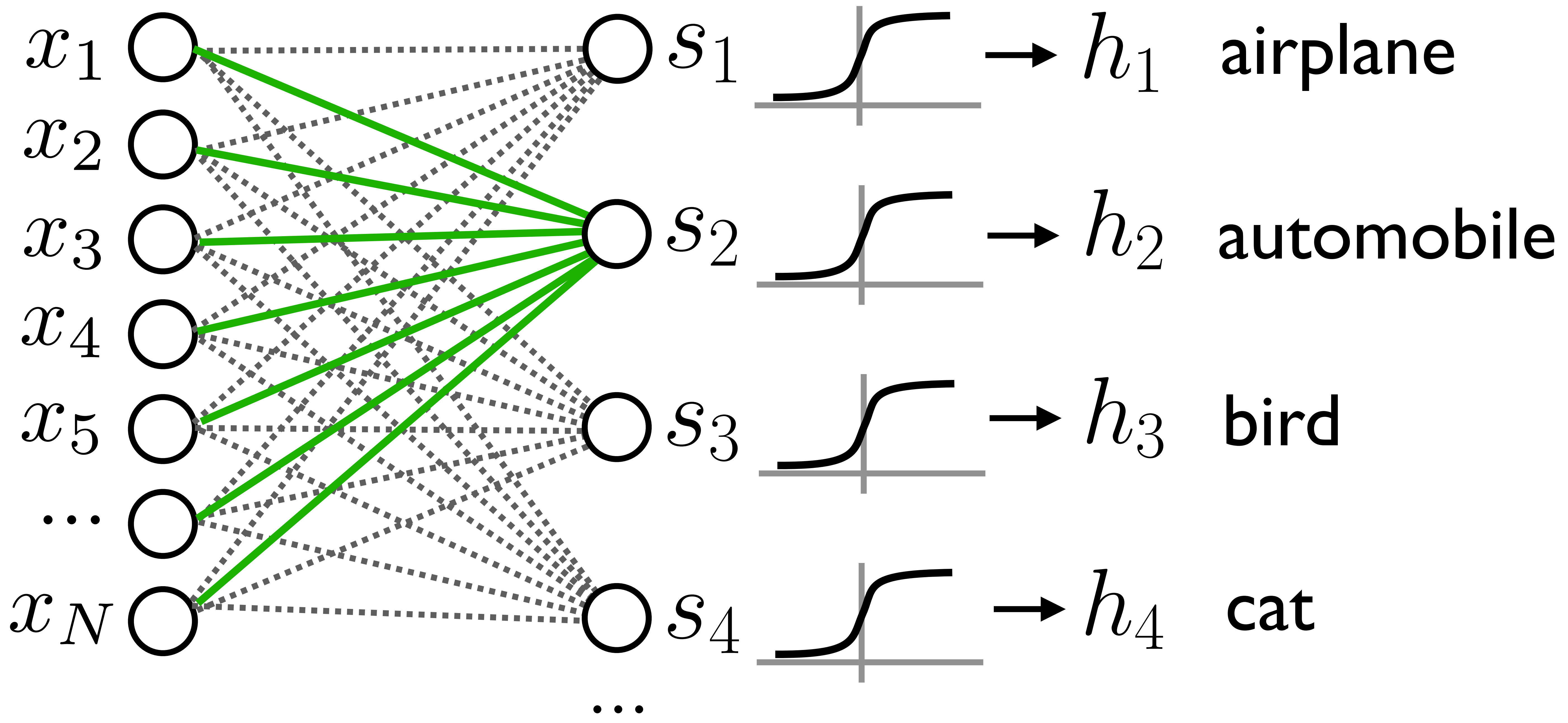


- Typically, we'll also add a bias term b

$$\mathbf{h} = \sigma(\mathbf{W}^T\mathbf{x} + \mathbf{b})$$

Linear = Fully Connected Layer

- Note that our linear matrix multiplication classifier is equivalent to a fully connected layer in a neural network

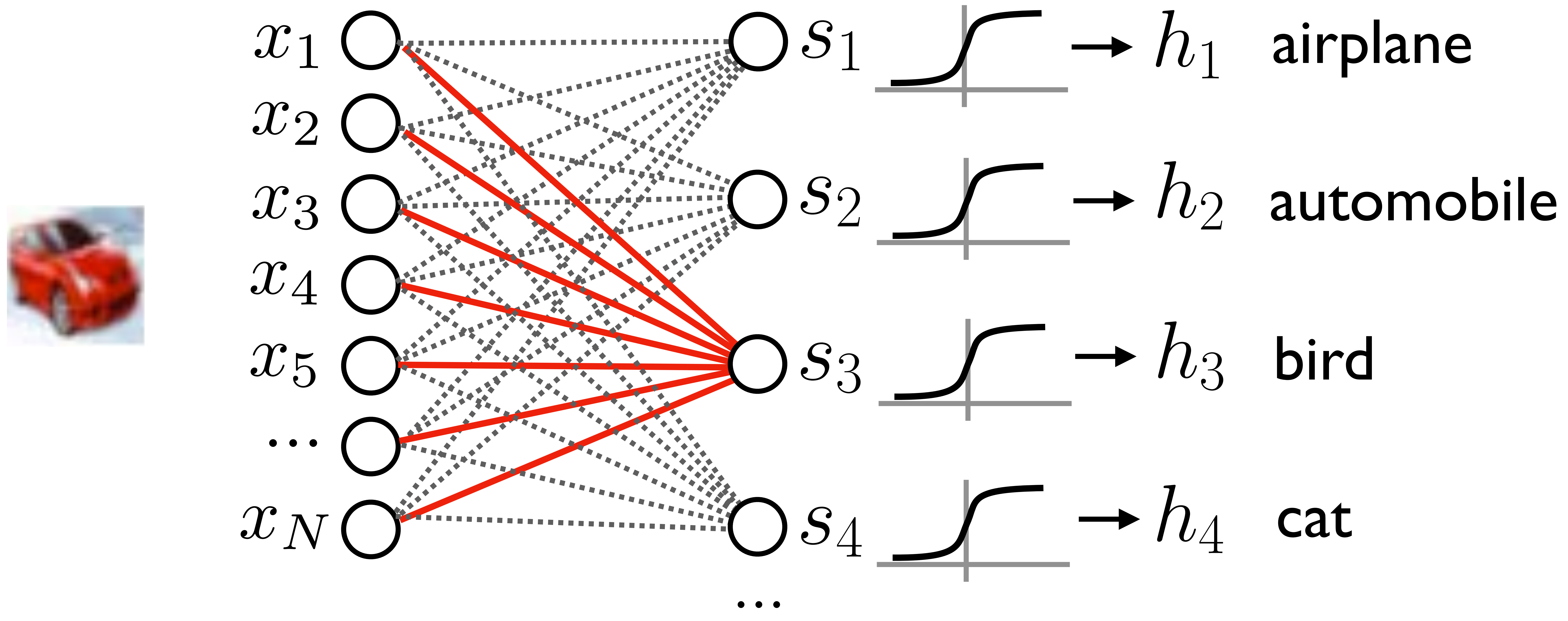


- Typically, we'll also add a bias term b

$$\mathbf{h} = \sigma(\mathbf{W}^T\mathbf{x} + \mathbf{b})$$

Linear = Fully Connected Layer

- Note that our linear matrix multiplication classifier is equivalent to a fully connected layer in a neural network

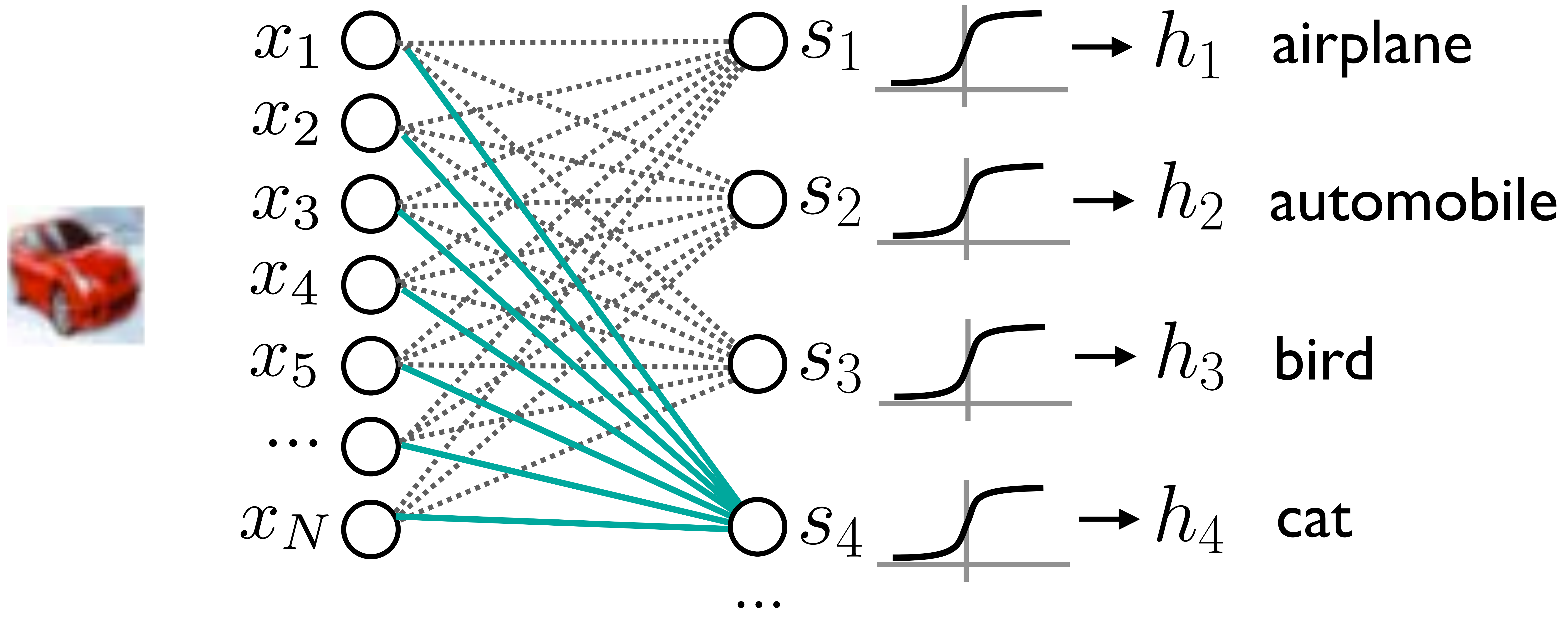


- Typically, we'll also add a bias term b

$$\mathbf{h} = \sigma(\mathbf{W}^T\mathbf{x} + \mathbf{b})$$

Linear = Fully Connected Layer

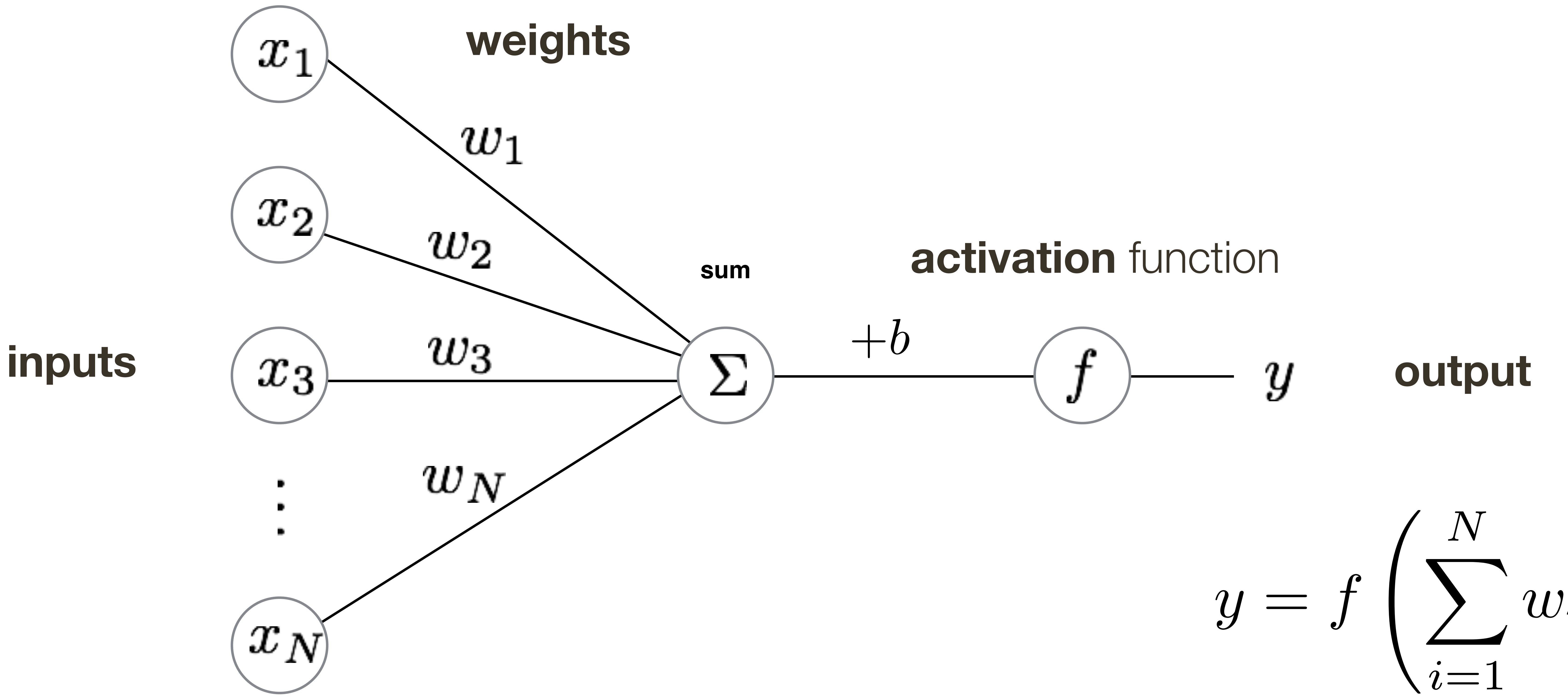
- Note that our linear matrix multiplication classifier is equivalent to a fully connected layer in a neural network



- Typically, we'll also add a bias term b

$$\mathbf{h} = \sigma(\mathbf{W}^T\mathbf{x} + \mathbf{b})$$

A **Neuron**



$$y = f\left(\sum_{i=1}^{N} w_i x_i + b\right)$$

— The basic unit of computation in a neural network is a neuron.

— A neuron accepts some number of input signals, computes their weighted sum, and applies an **activation function** (or **non-linearity**) to the sum.

— Common activation functions include sigmoid and rectified linear unit (ReLU)

Activation Function: **Sigmoid**

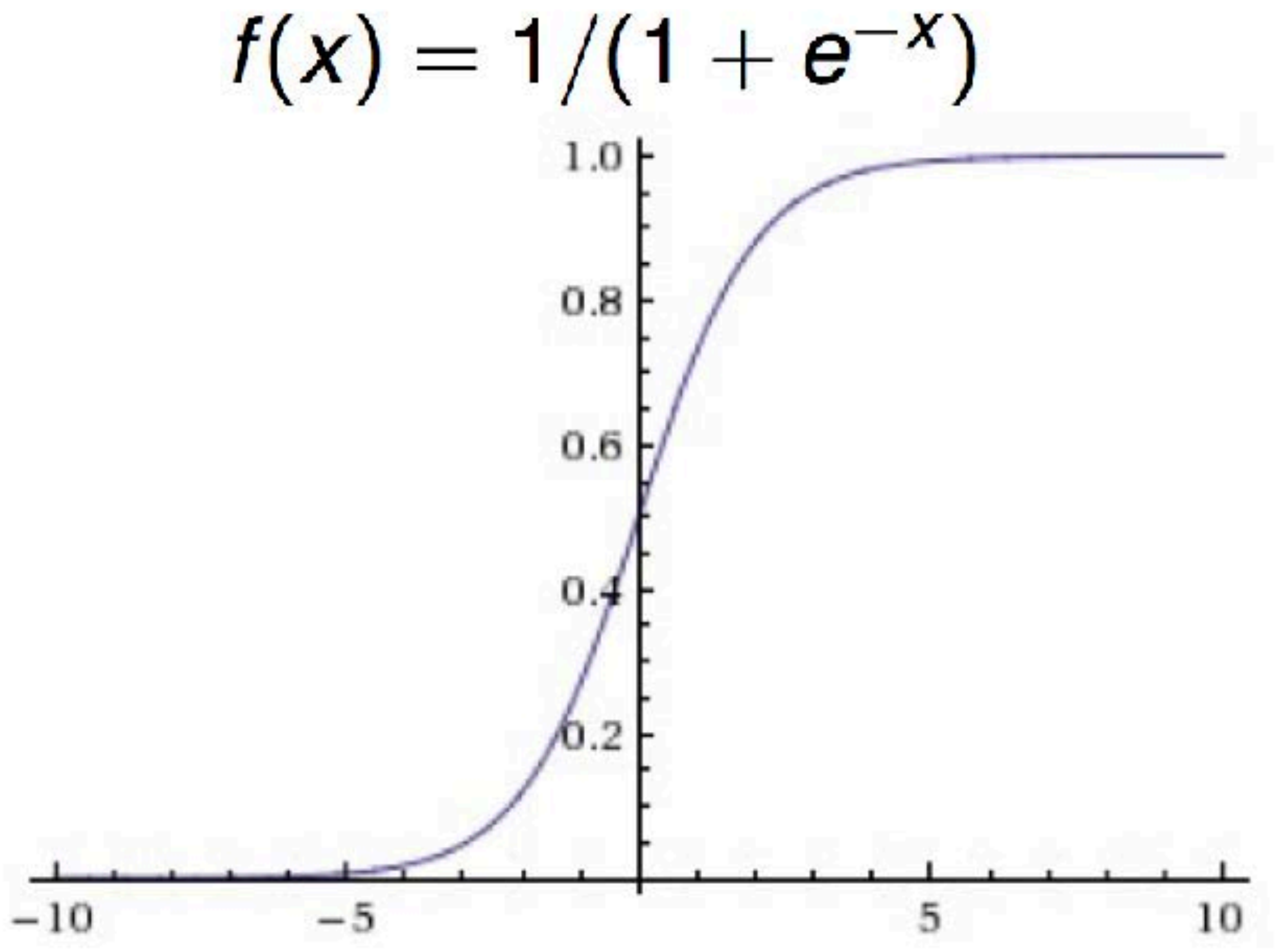


Figure credit: Fei-Fei and Karpathy

Common in many early neural networks

Biological analogy to saturated firing rate of neurons

Maps the input to the range [0,1]

Activation Function: **ReLU** (Rectified Linear Unit)

$$f(x) = \max(0, x)$$

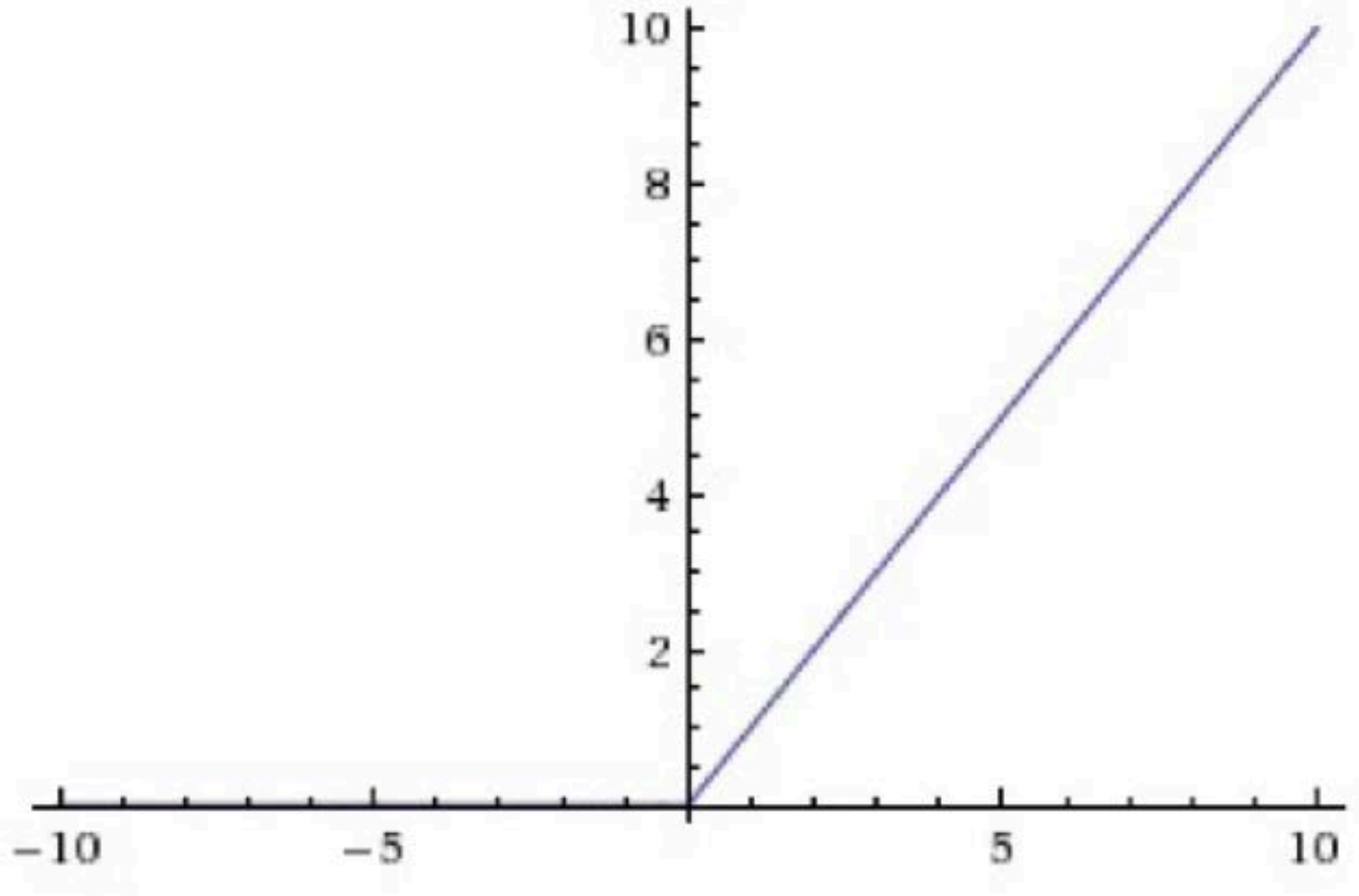


Figure credit: Fei-Fei and Karpathy

Maintains good gradient flow in networks, prevents vanishing gradient problem

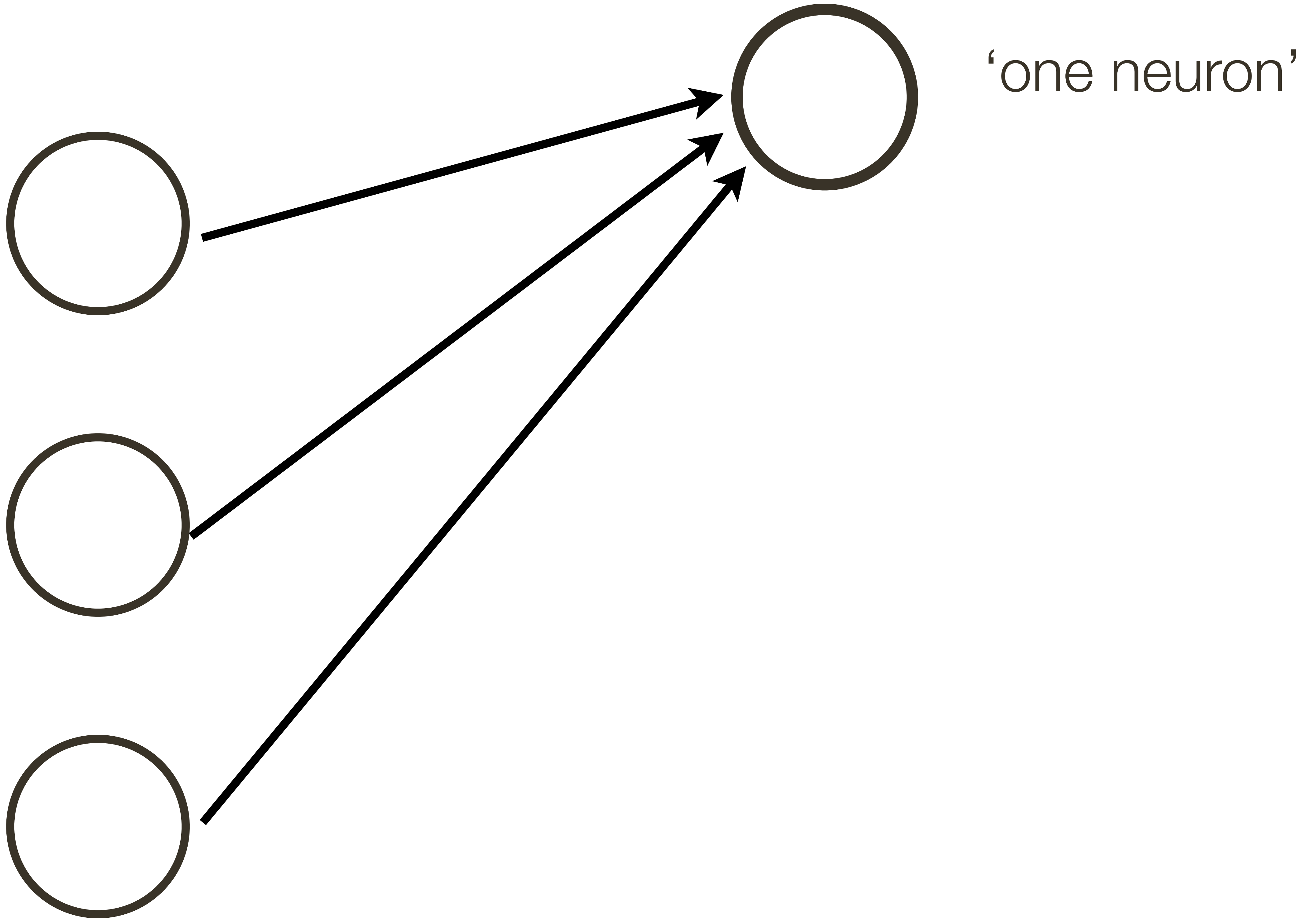
Very commonly used in interior (hidden) layers of neural nets

19.3

Why can't we have **linear** activation functions?

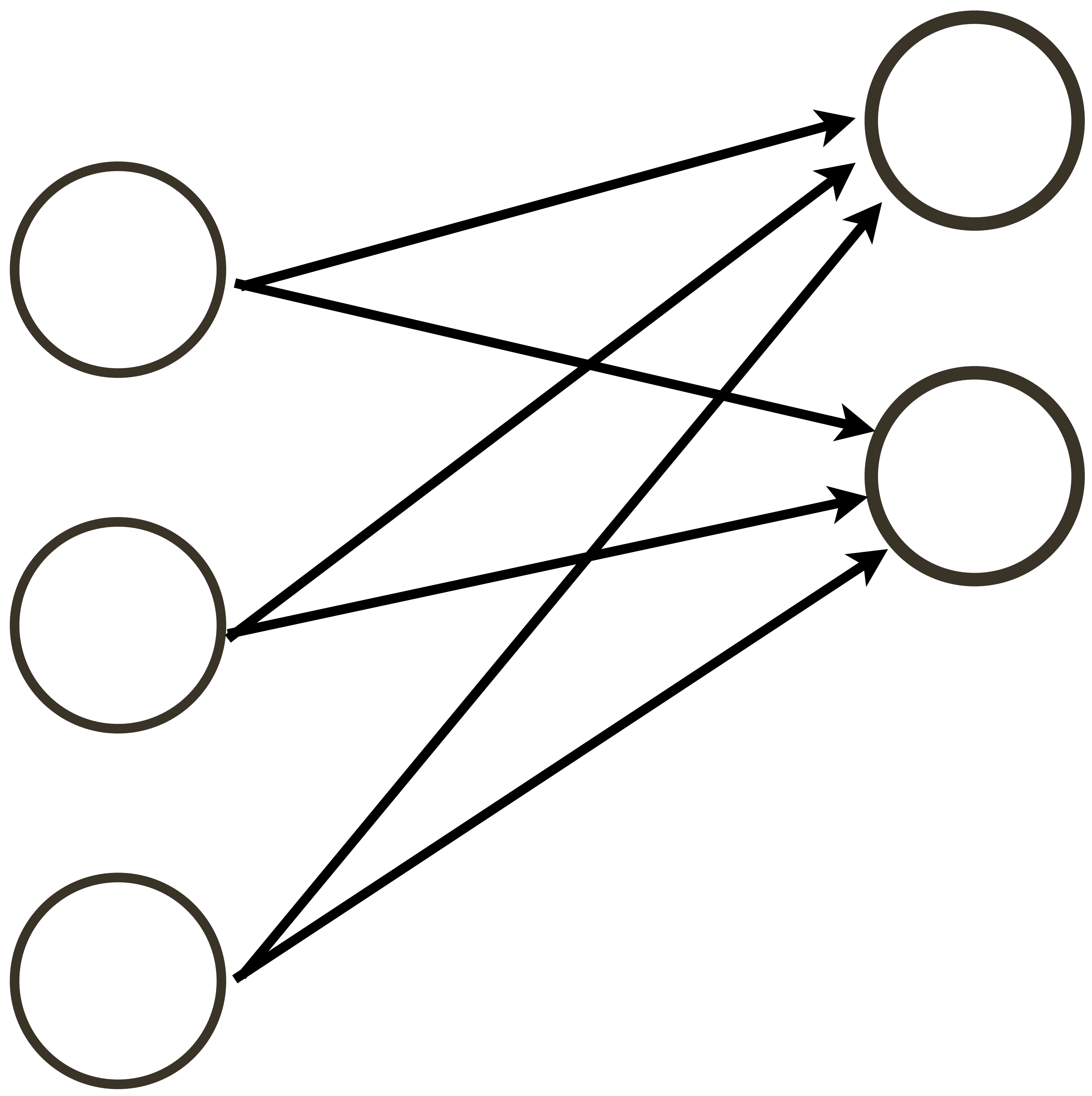
Neural Network

Connect a bunch of neurons together — a collection of connected neurons



Neural Network

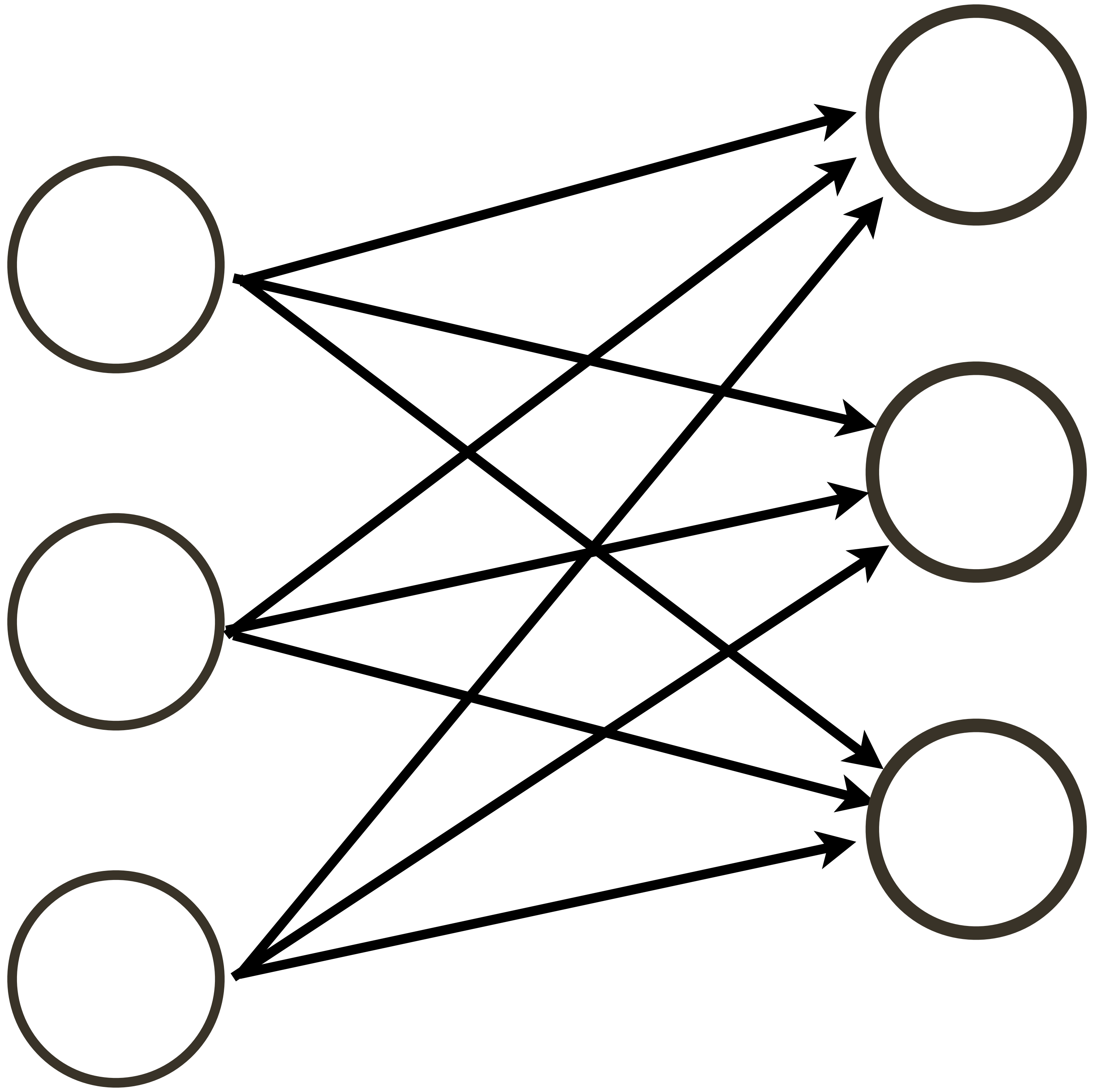
Connect a bunch of neurons together — a collection of connected neurons



‘two neurons’

Neural Network

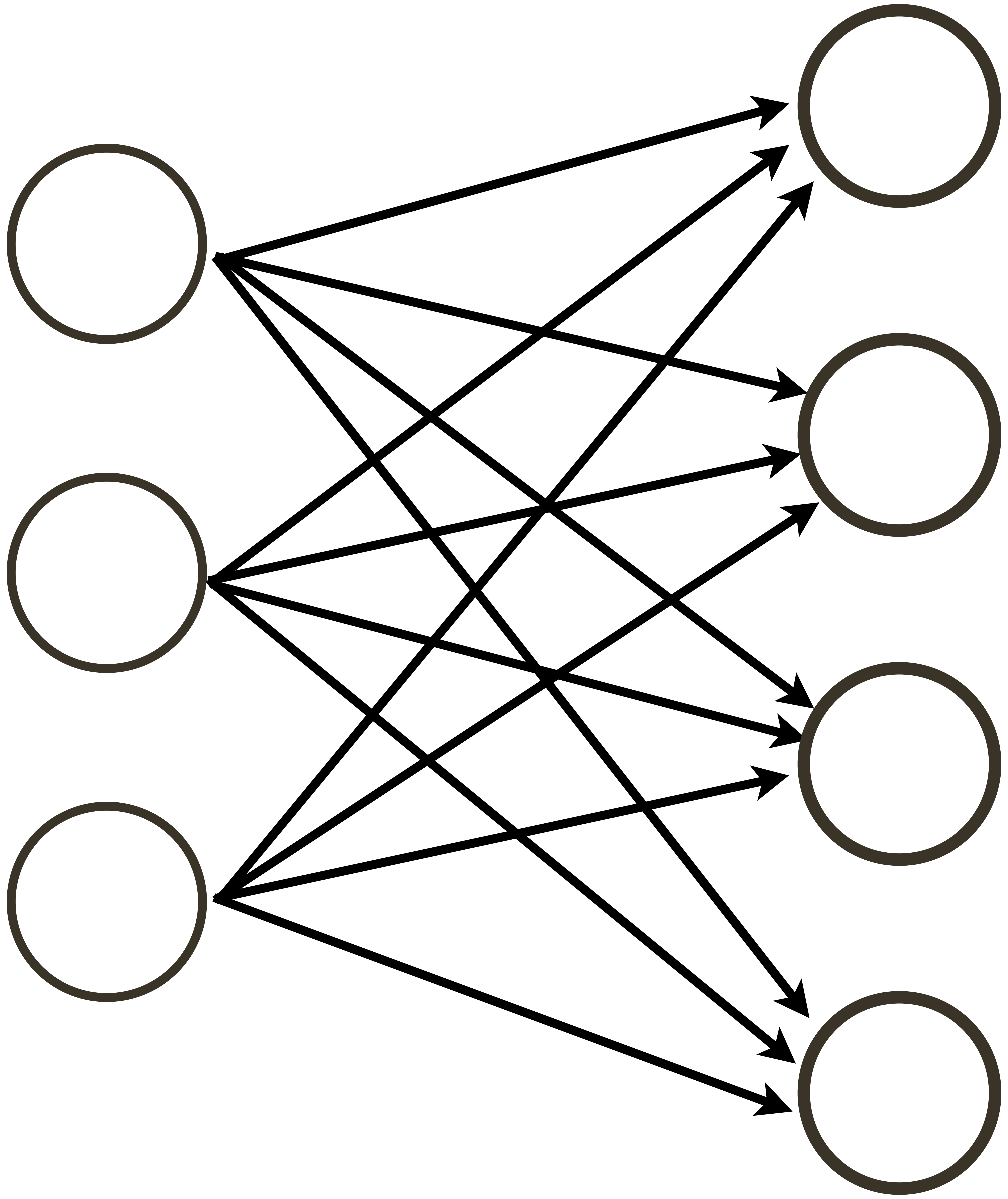
Connect a bunch of neurons together — a collection of connected neurons



‘three neurons’

Neural Network

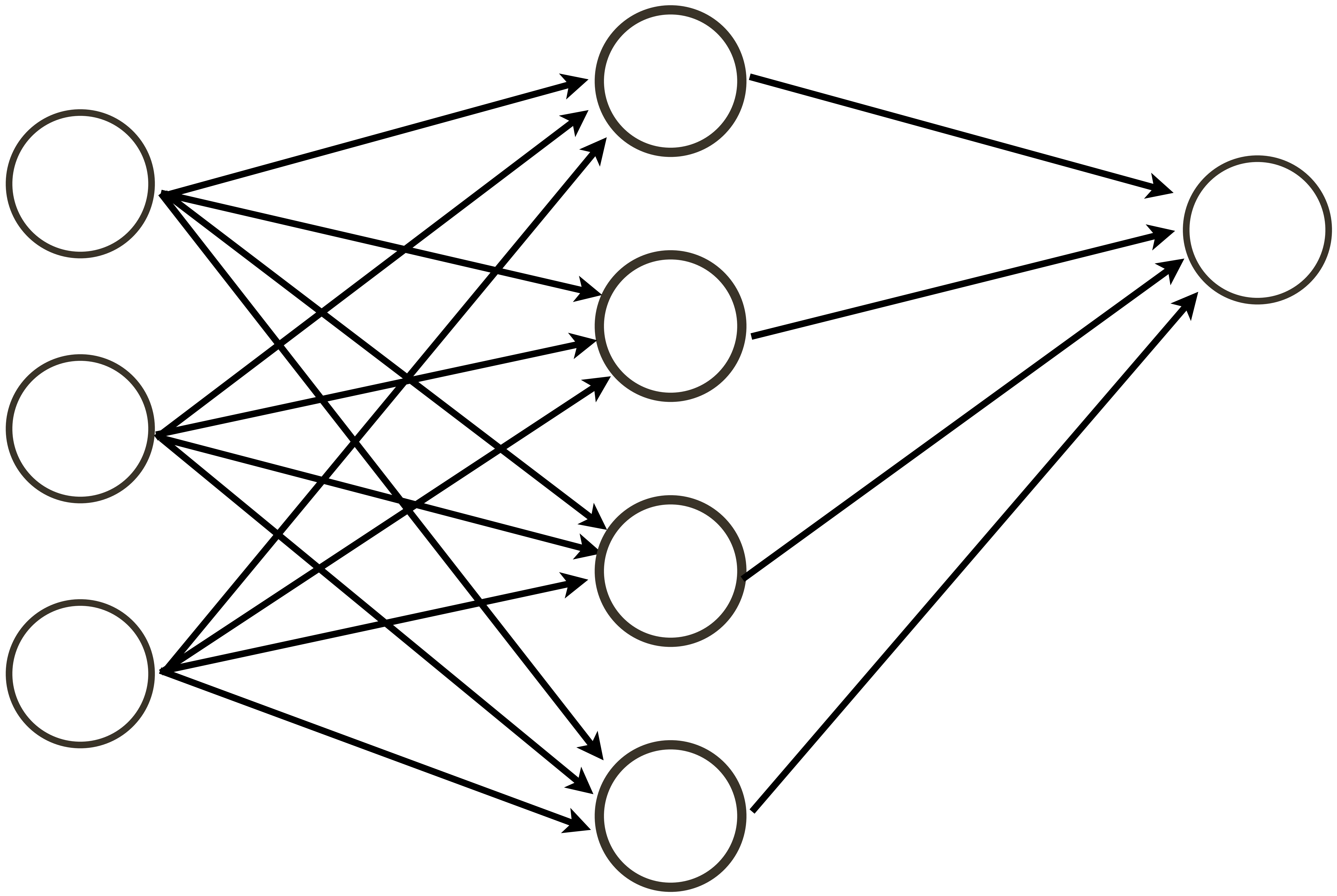
Connect a bunch of neurons together — a collection of connected neurons



‘four neurons’

Neural Network

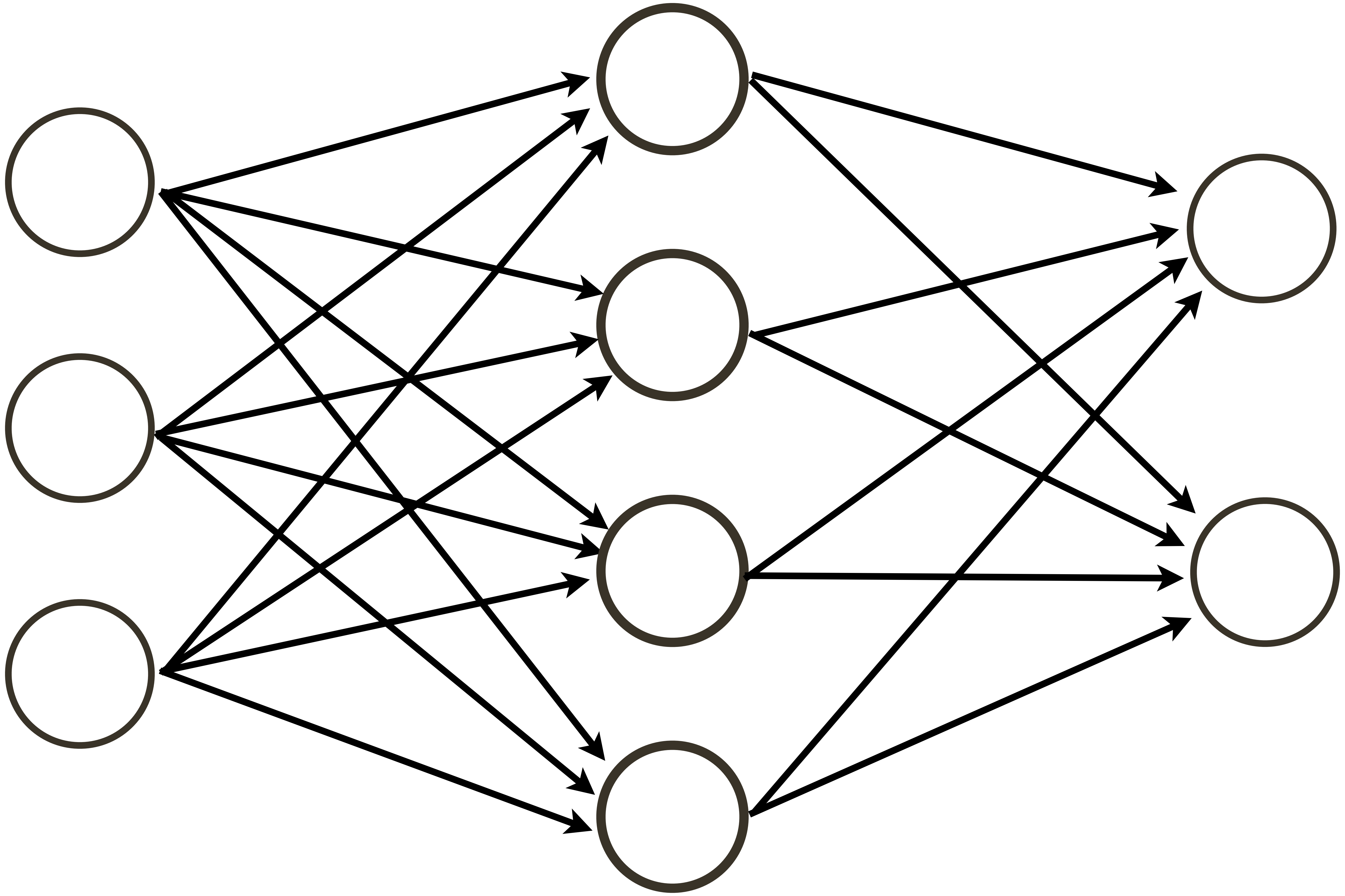
Connect a bunch of neurons together — a collection of connected neurons



‘five neurons’

Neural Network

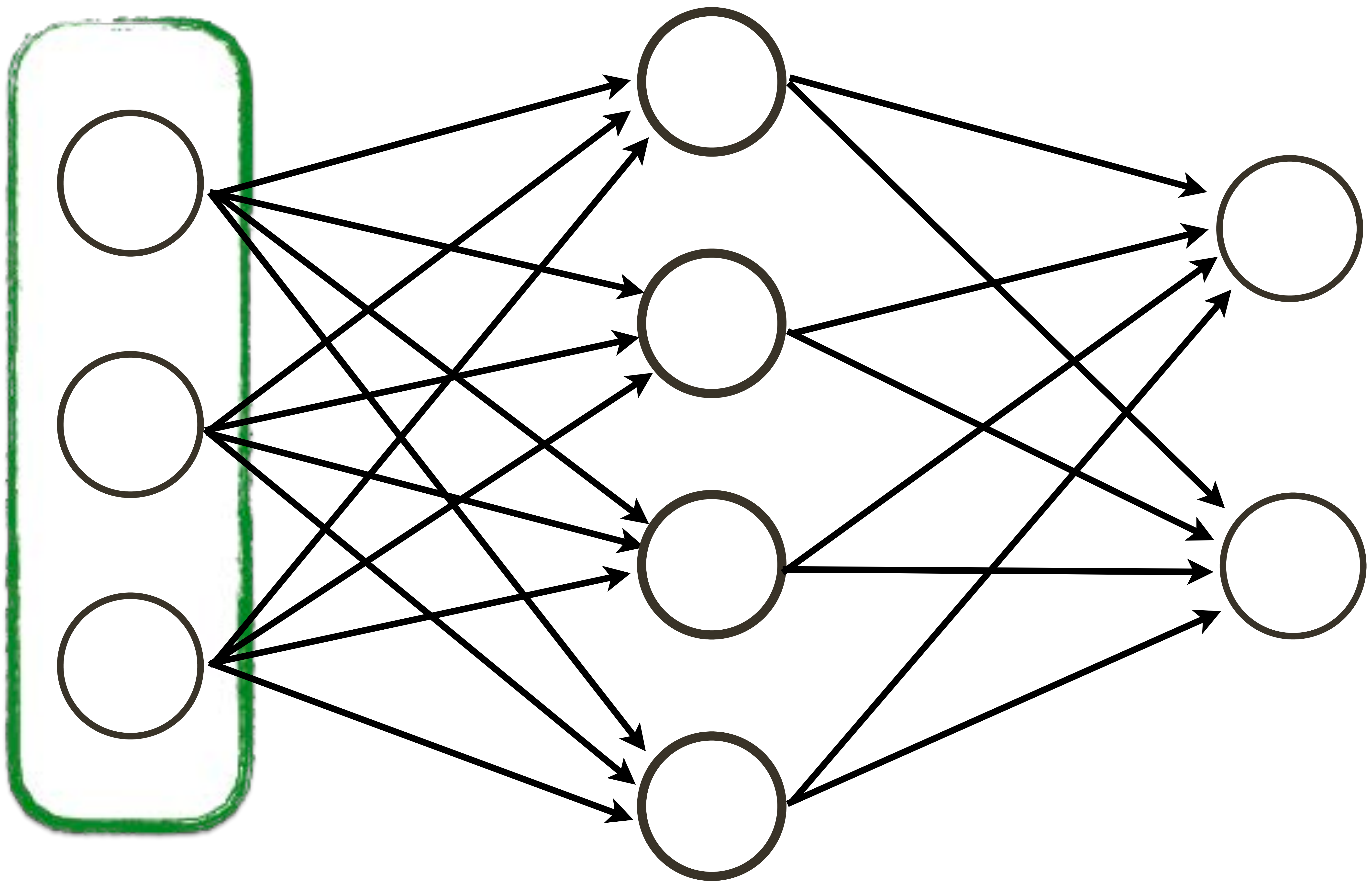
Connect a bunch of neurons together — a collection of connected neurons



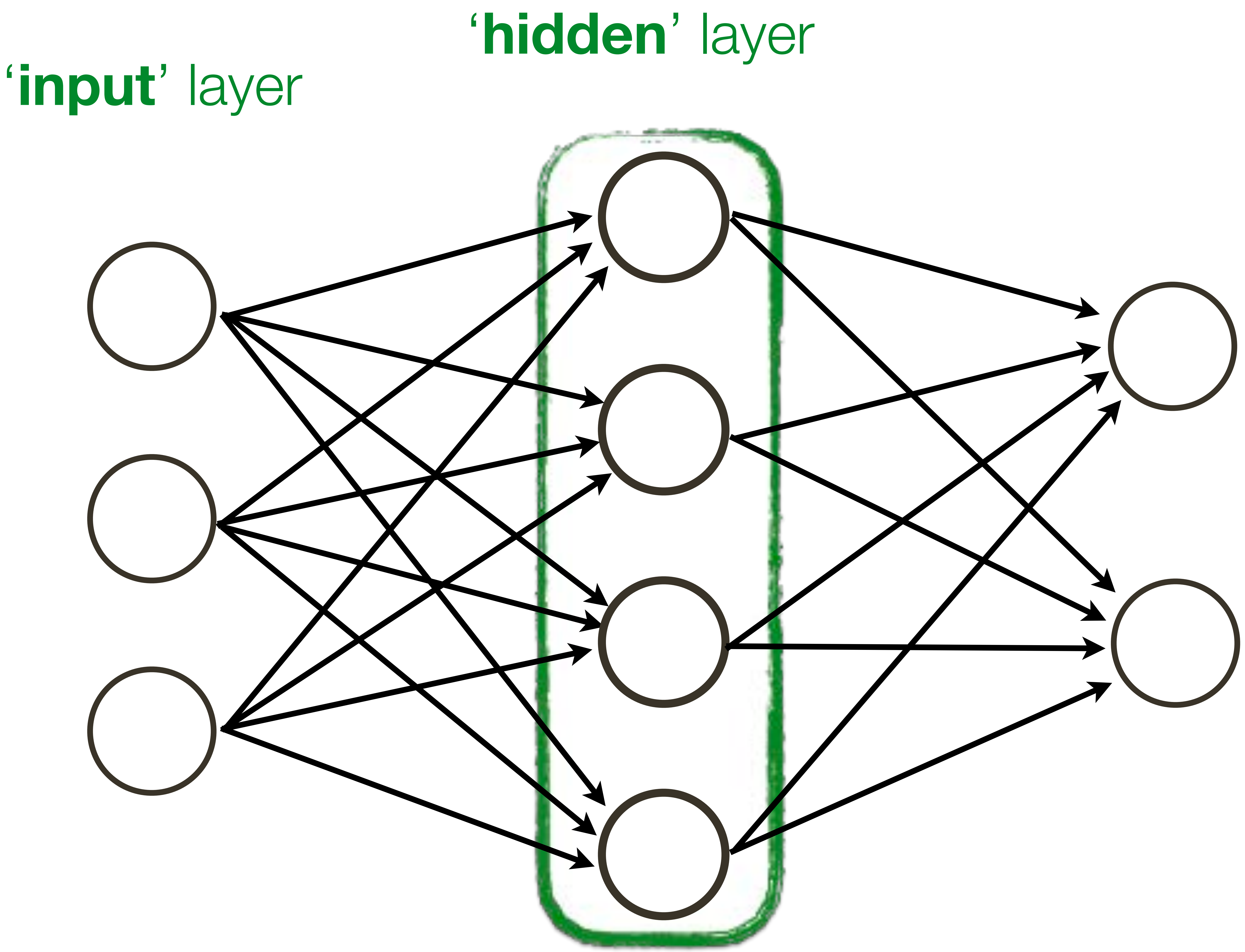
‘six neurons’

Neural Network: **Terminology**

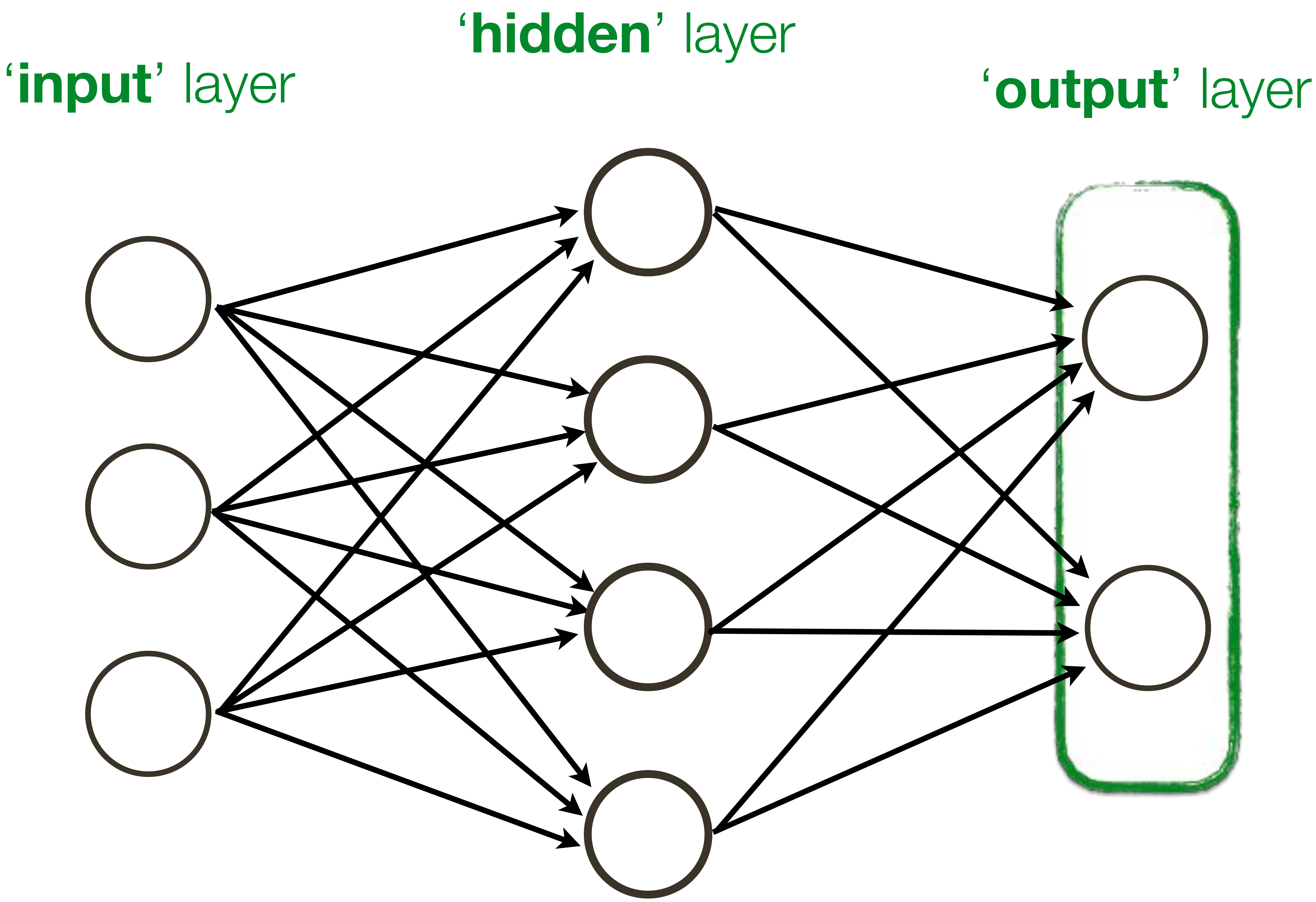
‘**input**’ layer



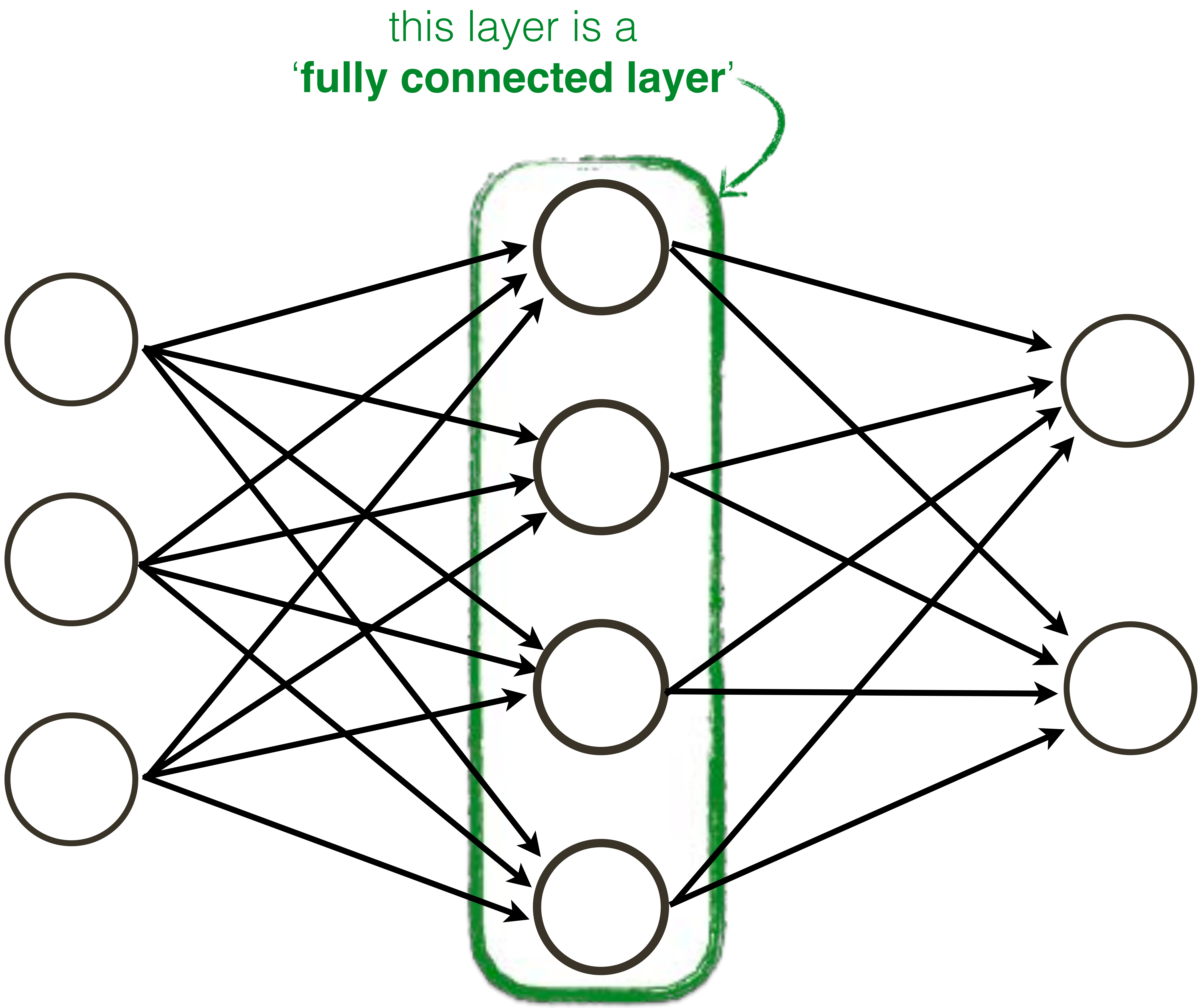
Neural Network: **Terminology**



Neural Network: **Terminology**

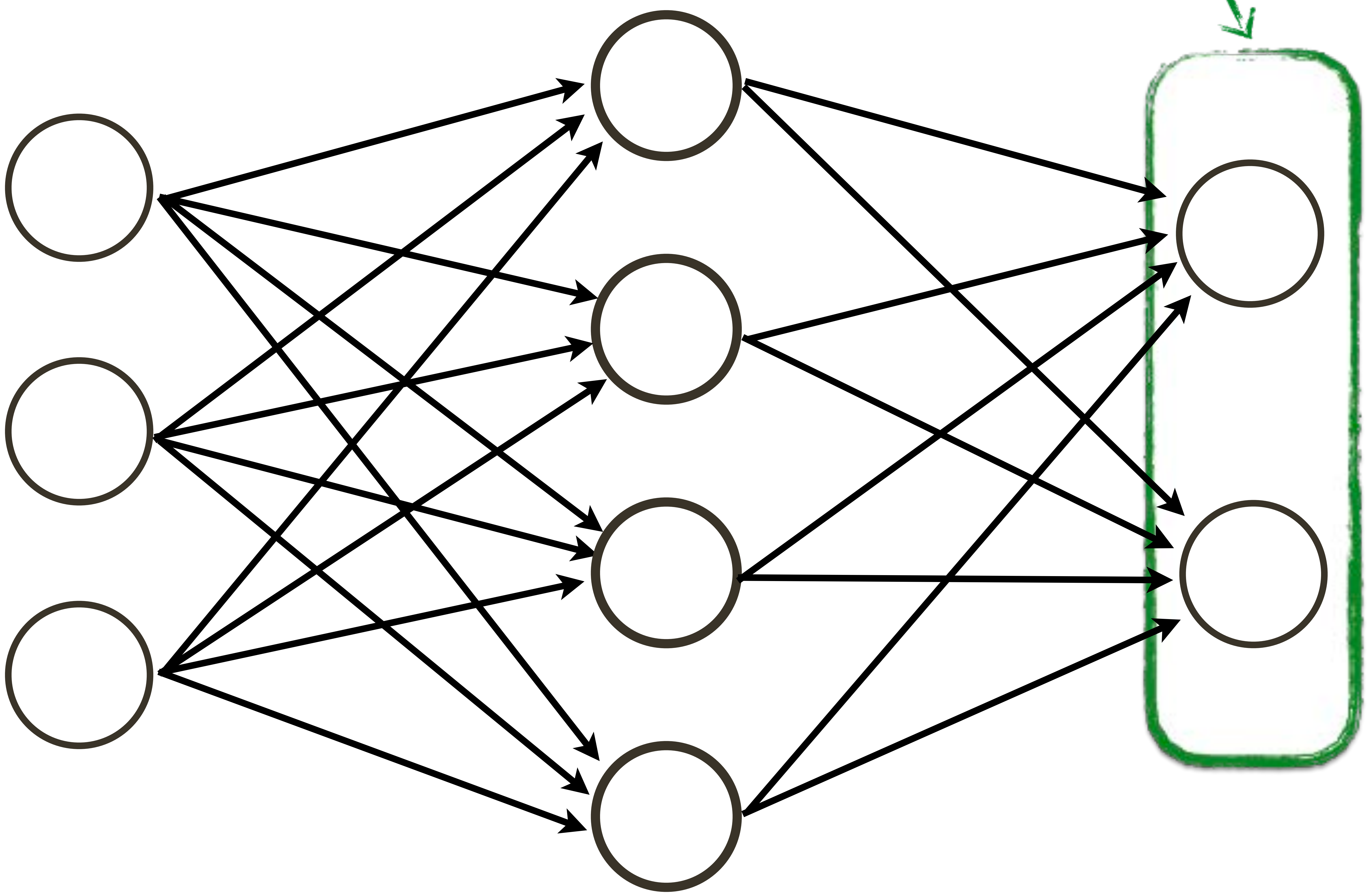


Neural Network: **Terminology**



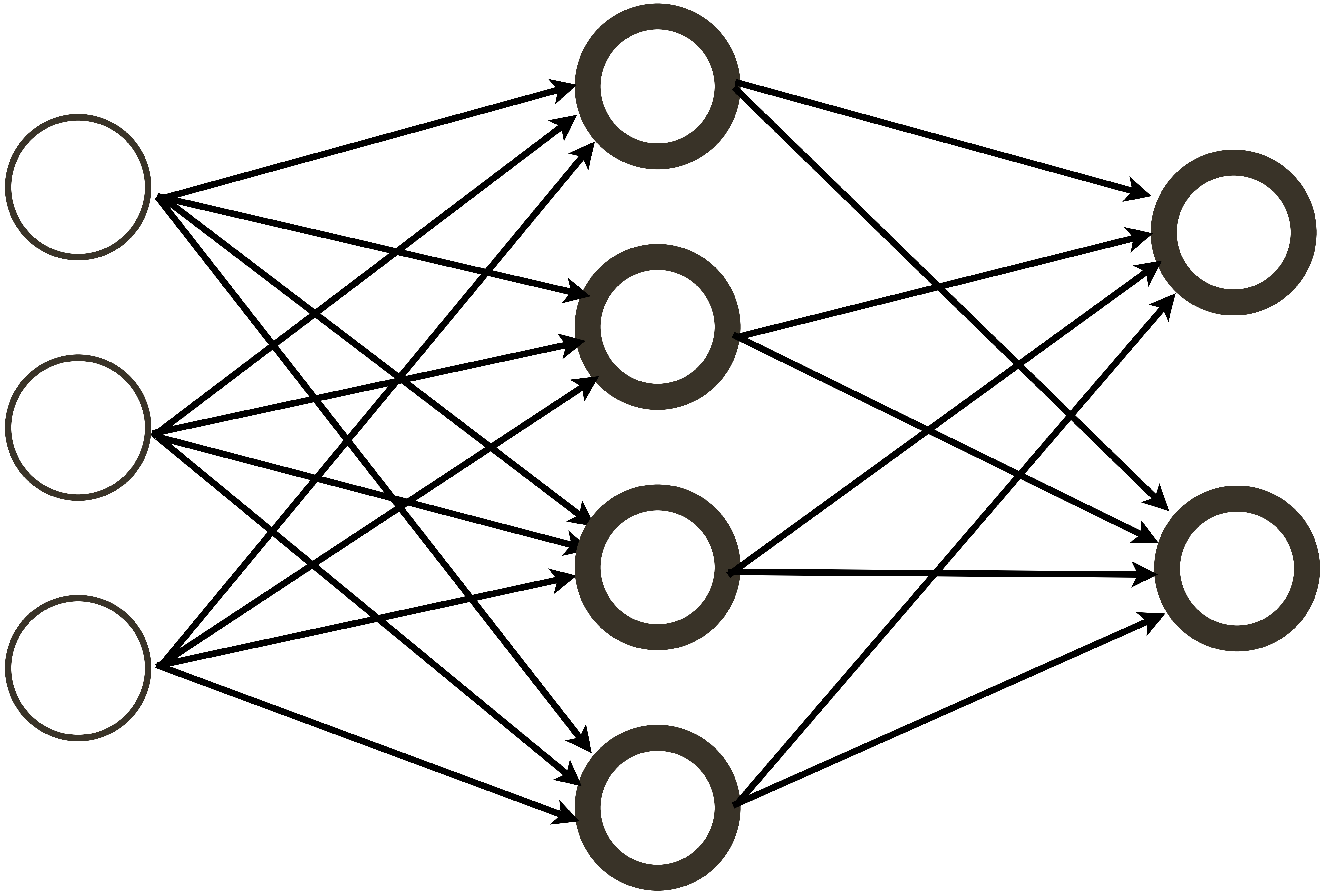
Neural Network: **Terminology**

so is this



Neural Network

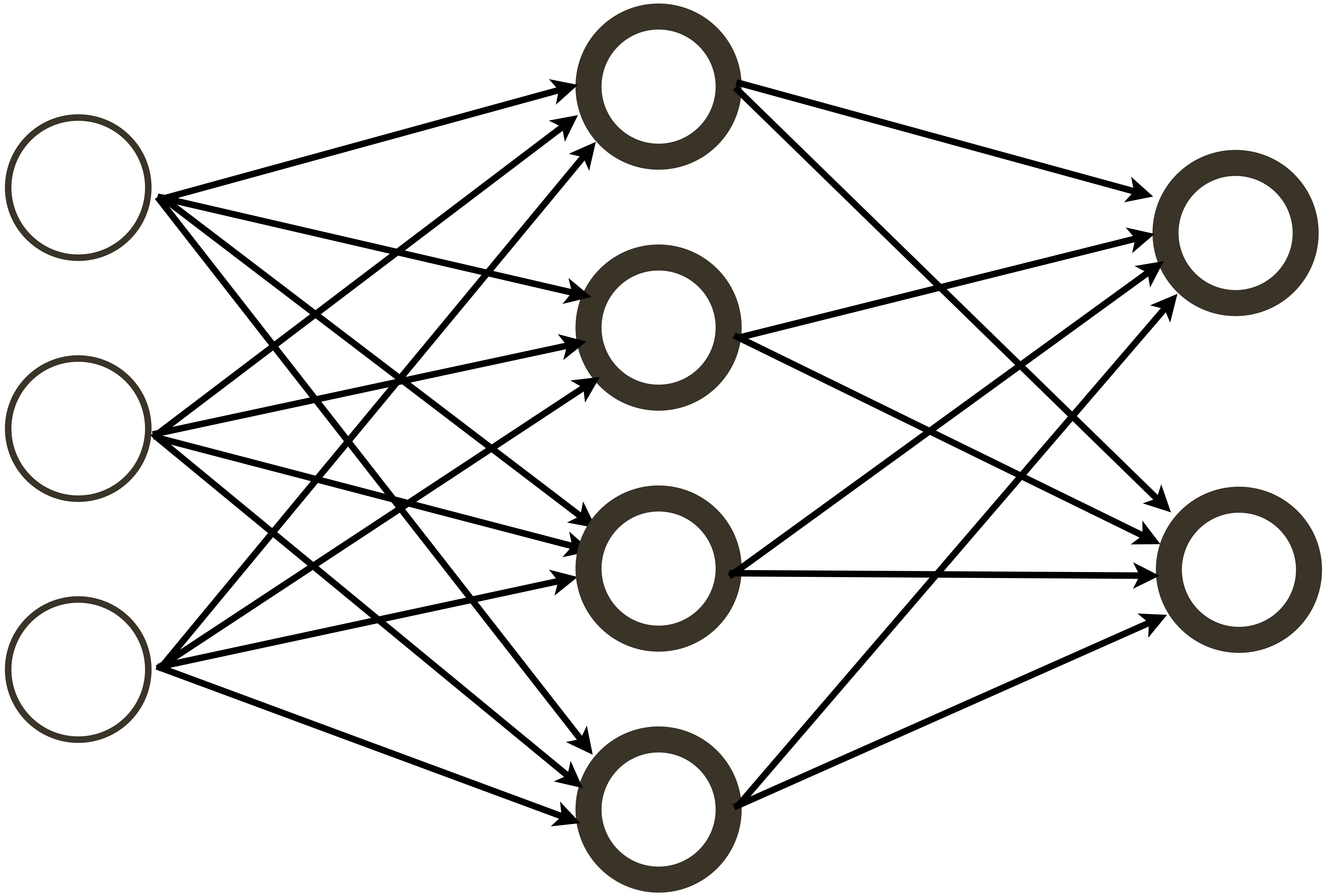
How many neurons?   4+2 = 6



Neural Network

How many neurons? 4+2 = 6

How many weights?

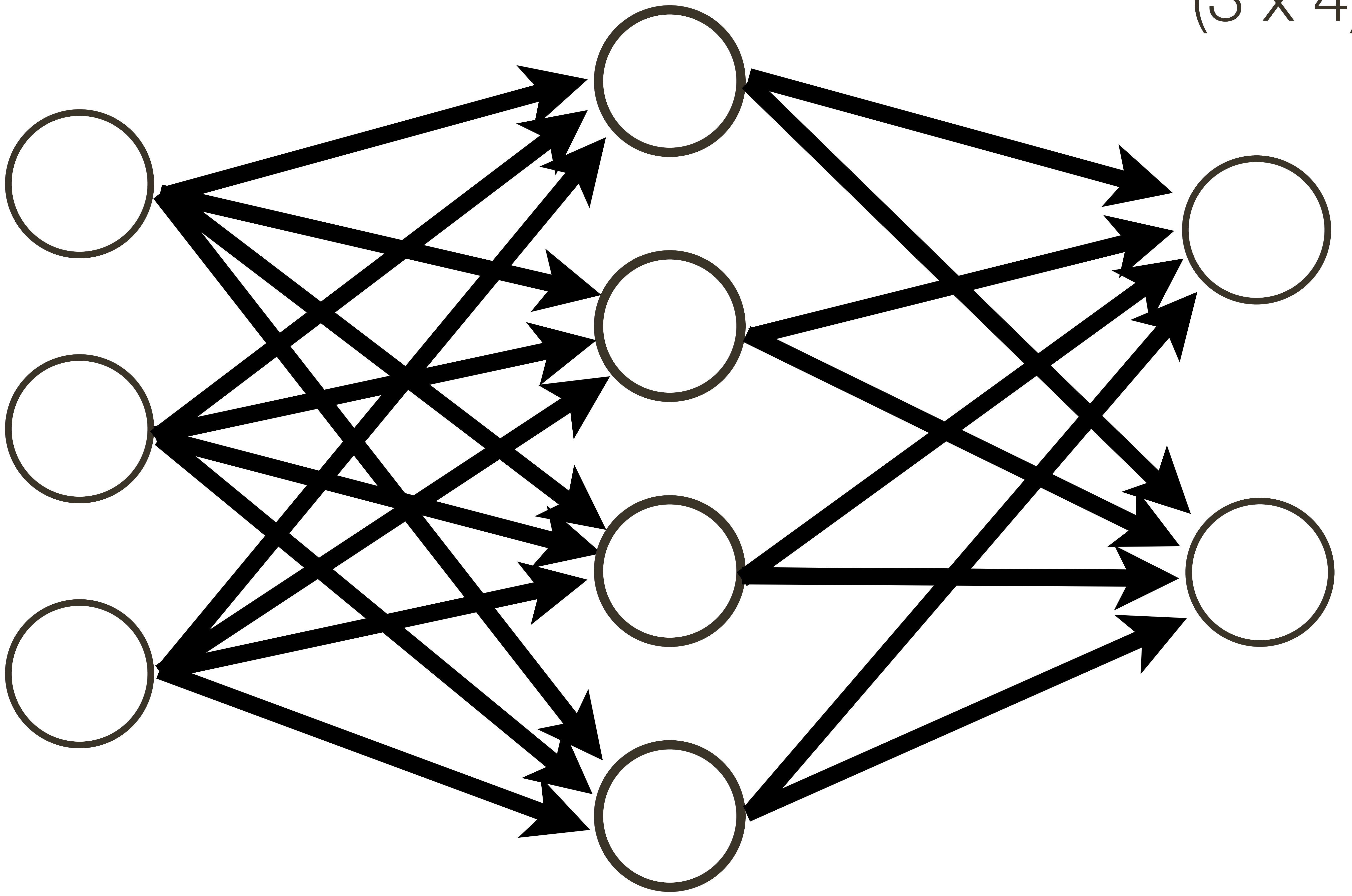


Neural Network

How many neurons? 4+2 = 6

How many weights?

(3 x 4) + (4 x 2) = 20

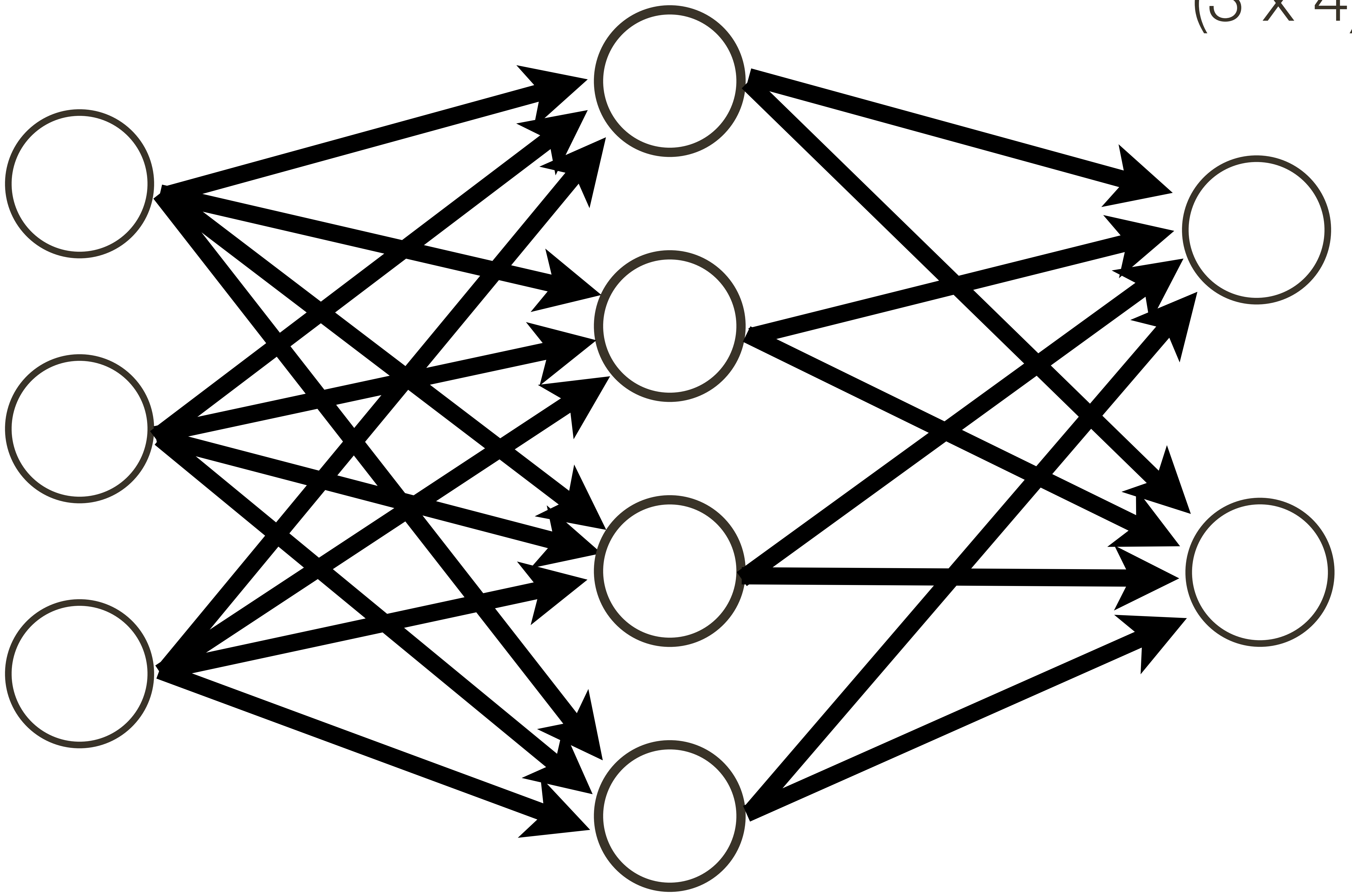


Neural Network

How many neurons? 4+2 = 6

How many weights?

(3 x 4) + (4 x 2) = 20



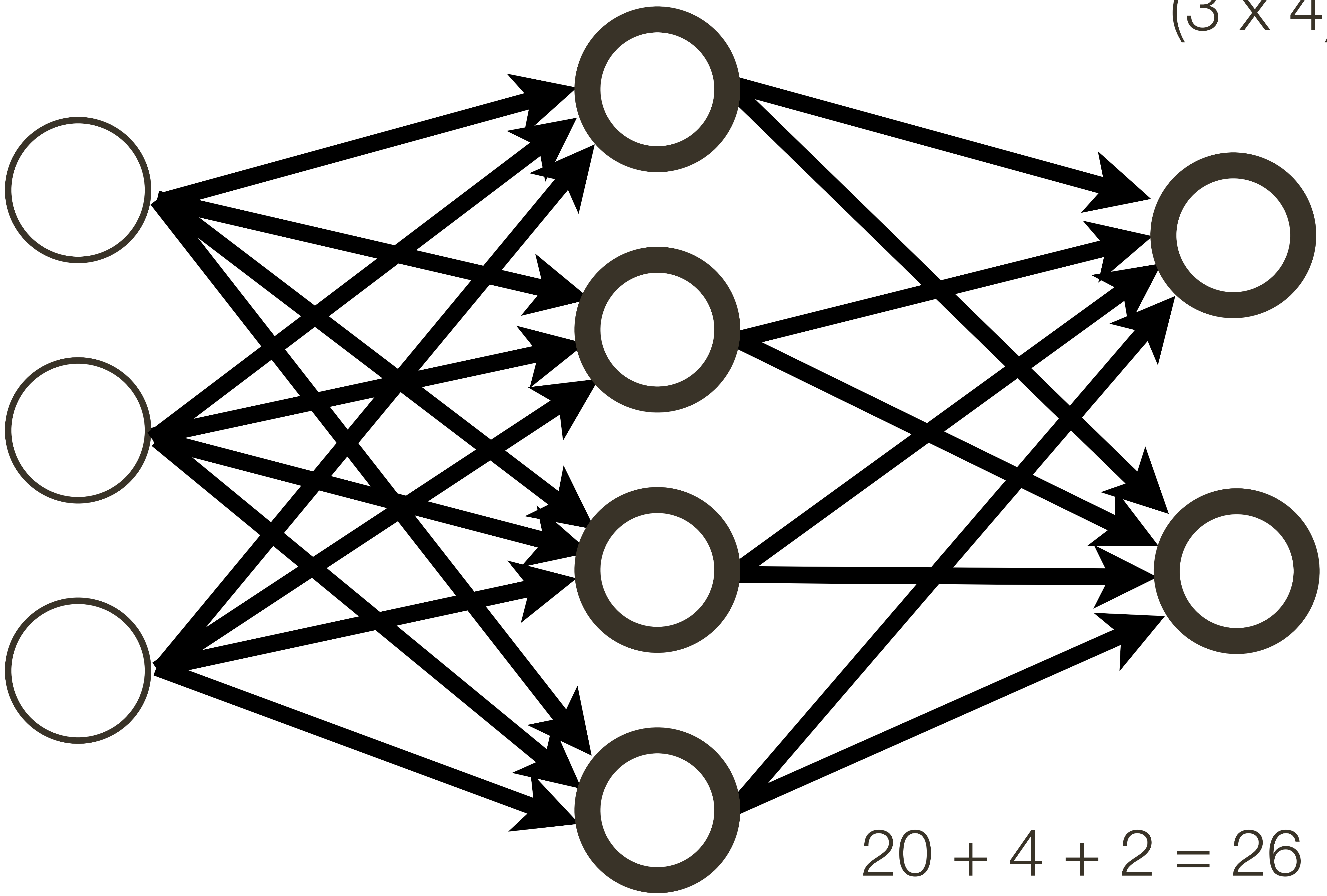
How many learnable parameters?

Neural Network

How many neurons? 4+2 = 6

How many weights?

(3 x 4) + (4 x 2) = 20



How many learnable parameters?

20 + 4 + 2 = 26

bias terms

Neural Network **Intuition**

* slide from Marc'Aurelio Renzato

Neural Network **Intuition**

Question: What is a Neural Network?

Answer: Complex mapping from an input (vector) to an output (vector)

* slide from Marc'Aurelio Renzato

Neural Network **Intuition**

Question: What is a Neural Network?

Answer: Complex mapping from an input (vector) to an output (vector)

Question: What class of functions should be considered for this mapping?

Answer: Compositions of simpler functions (a.k.a. layers)? We will talk more about what specific functions next …

* slide from Marc'Aurelio Renzato

Neural Network **Intuition**

Question: What is a Neural Network?

Answer: Complex mapping from an input (vector) to an output (vector)

Question: What class of functions should be considered for this mapping?

Answer: Compositions of simpler functions (a.k.a. layers)? We will talk more about what specific functions next …

Question: What does a hidden unit do?

Answer: It can be thought of as classifier or a feature.

* slide from Marc'Aurelio Renzato

Neural Network **Intuition**

Question: What is a Neural Network?

Answer: Complex mapping from an input (vector) to an output (vector)

Question: What class of functions should be considered for this mapping?

Answer: Compositions of simpler functions (a.k.a. layers)? We will talk more about what specific functions next …

Question: What does a hidden unit do?

Answer: It can be thought of as classifier or a feature.

Question: Why have many layers?

Answer: 1) More layers = more complex functional mapping

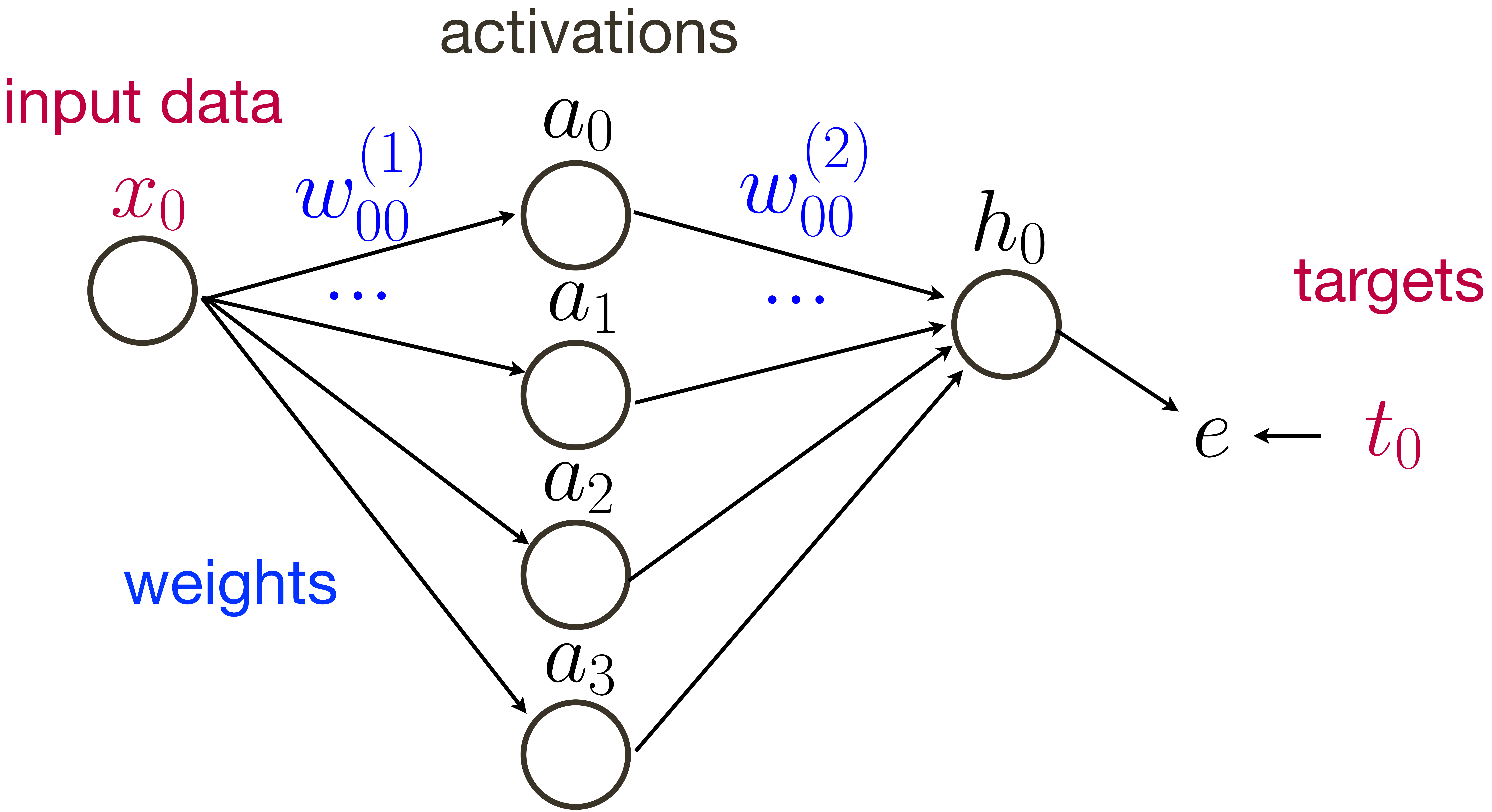
2) More efficient due to distributed representation

* slide from Marc'Aurelio Renzato

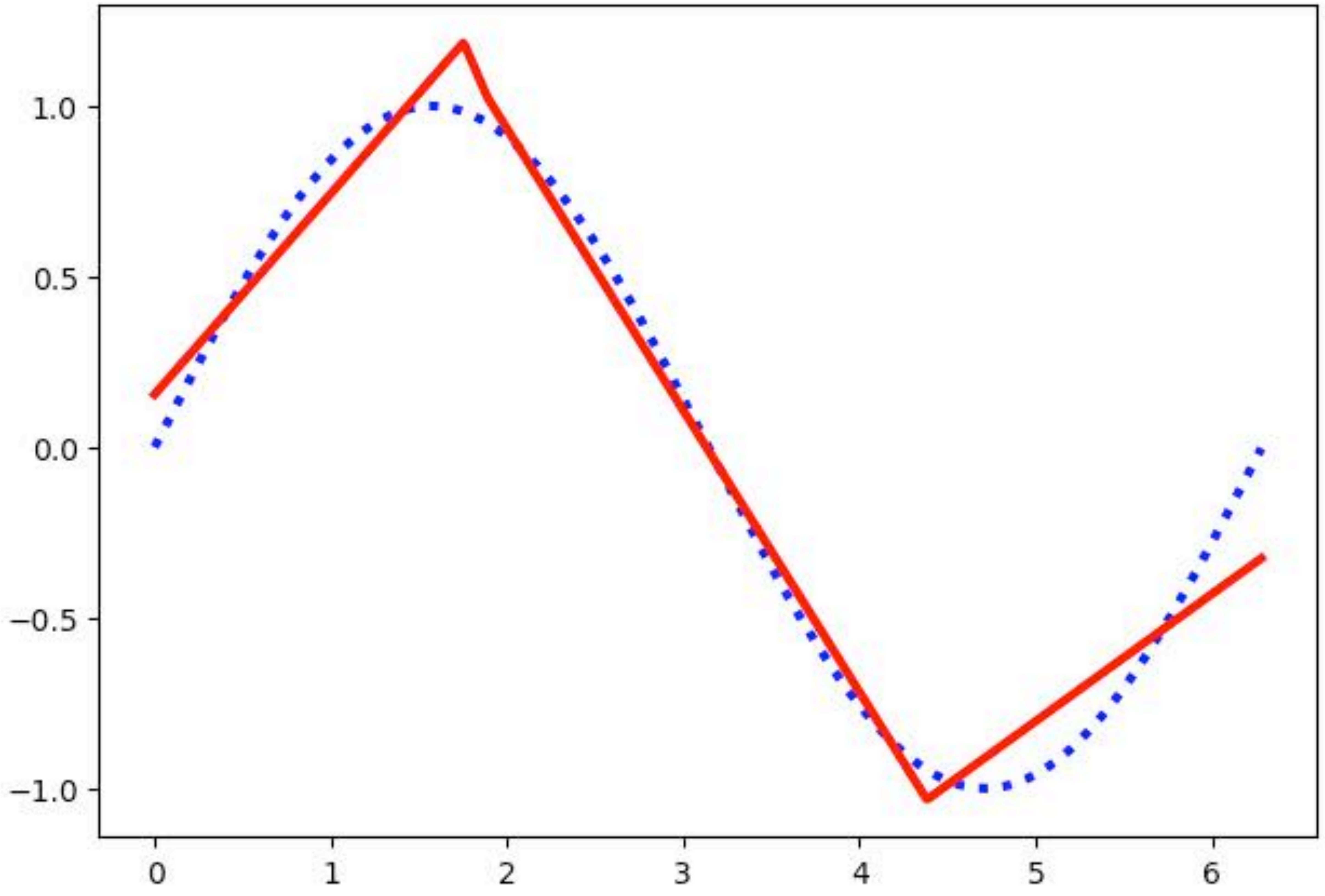
2-Layer **Neural** Network 19.4

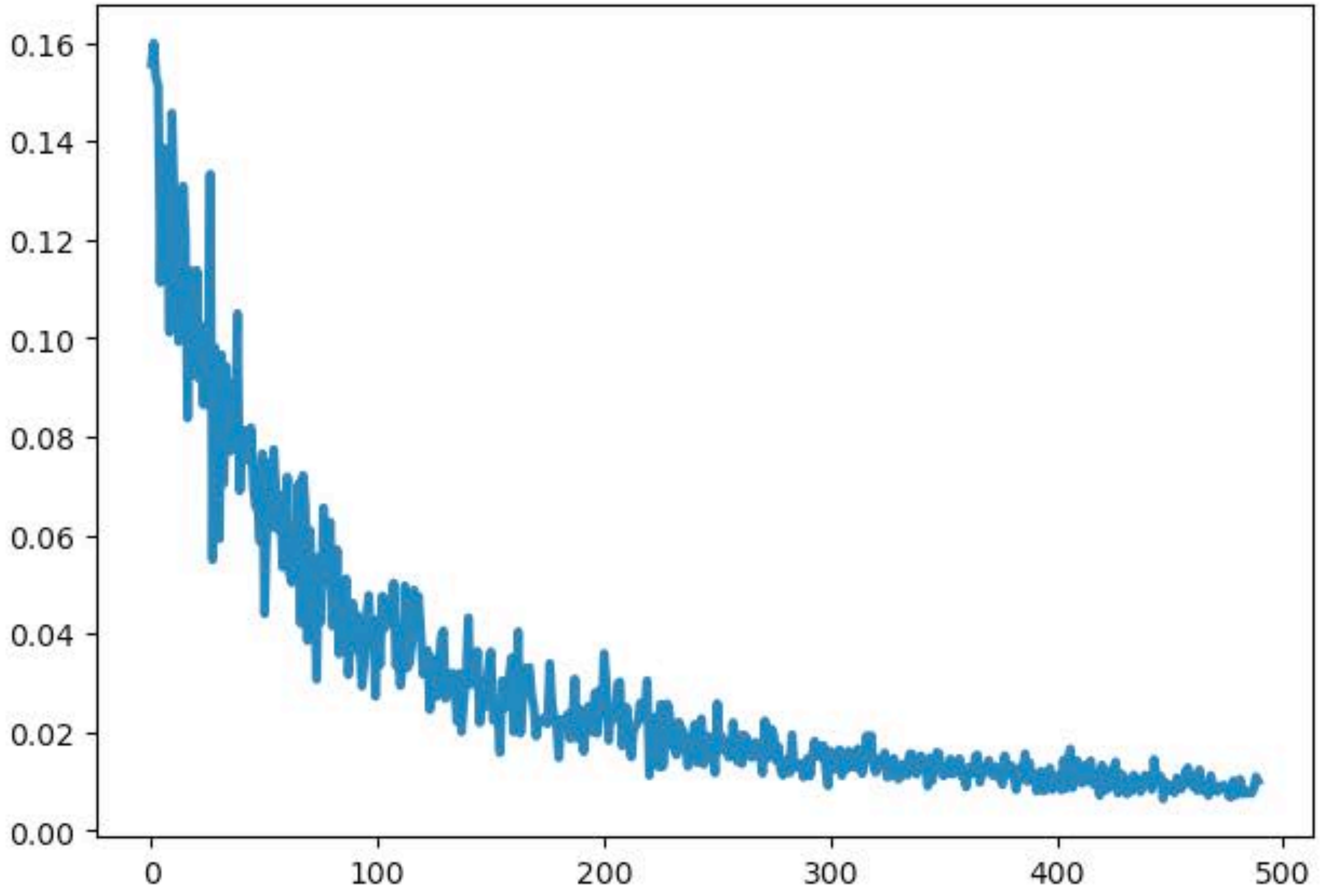
2-Layer **Neural** Network 19.4

2-Layer **Neural** Network — n hidden, 1 input/output



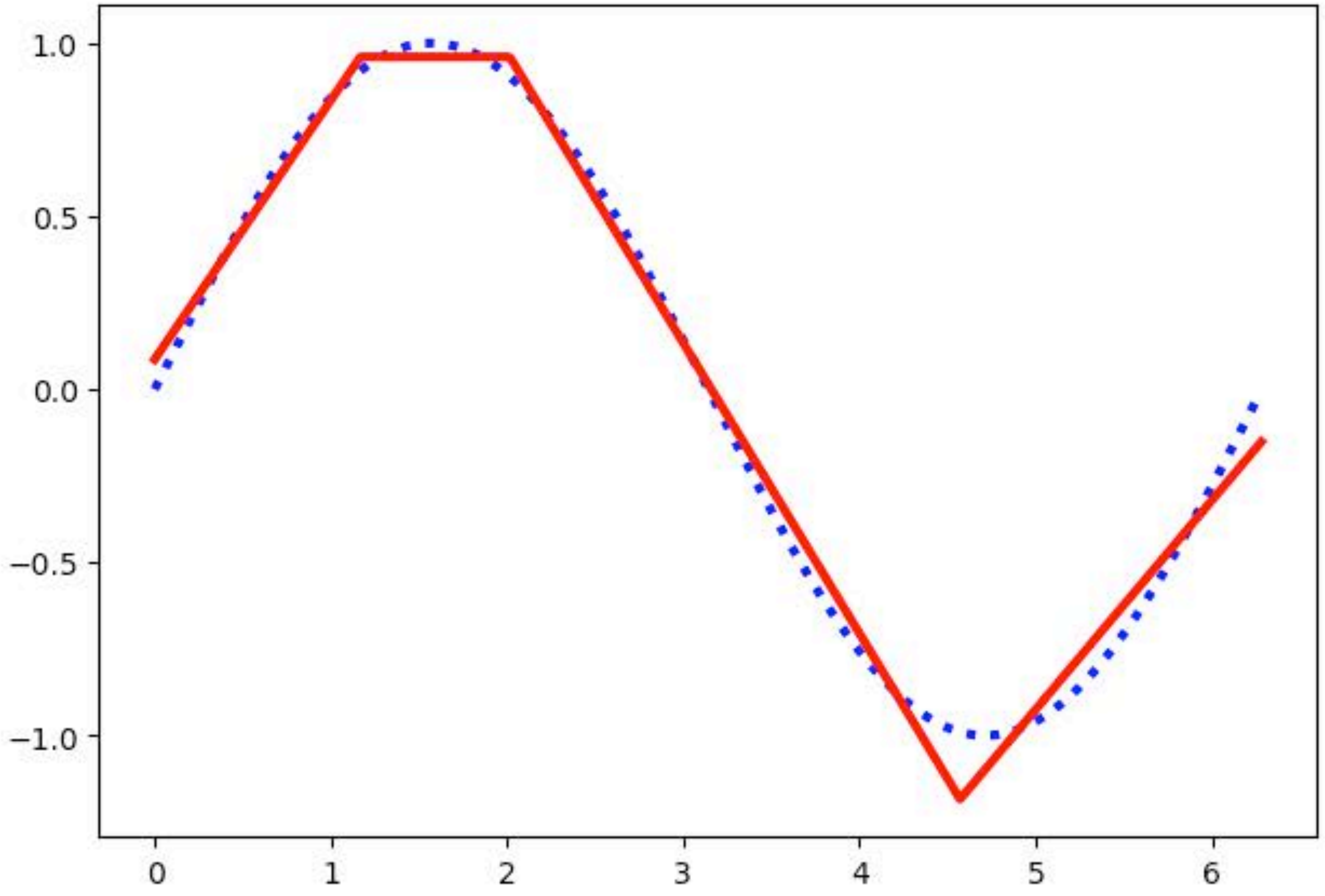
2-Layer **Neural** Network — n hidden, 1 input/output

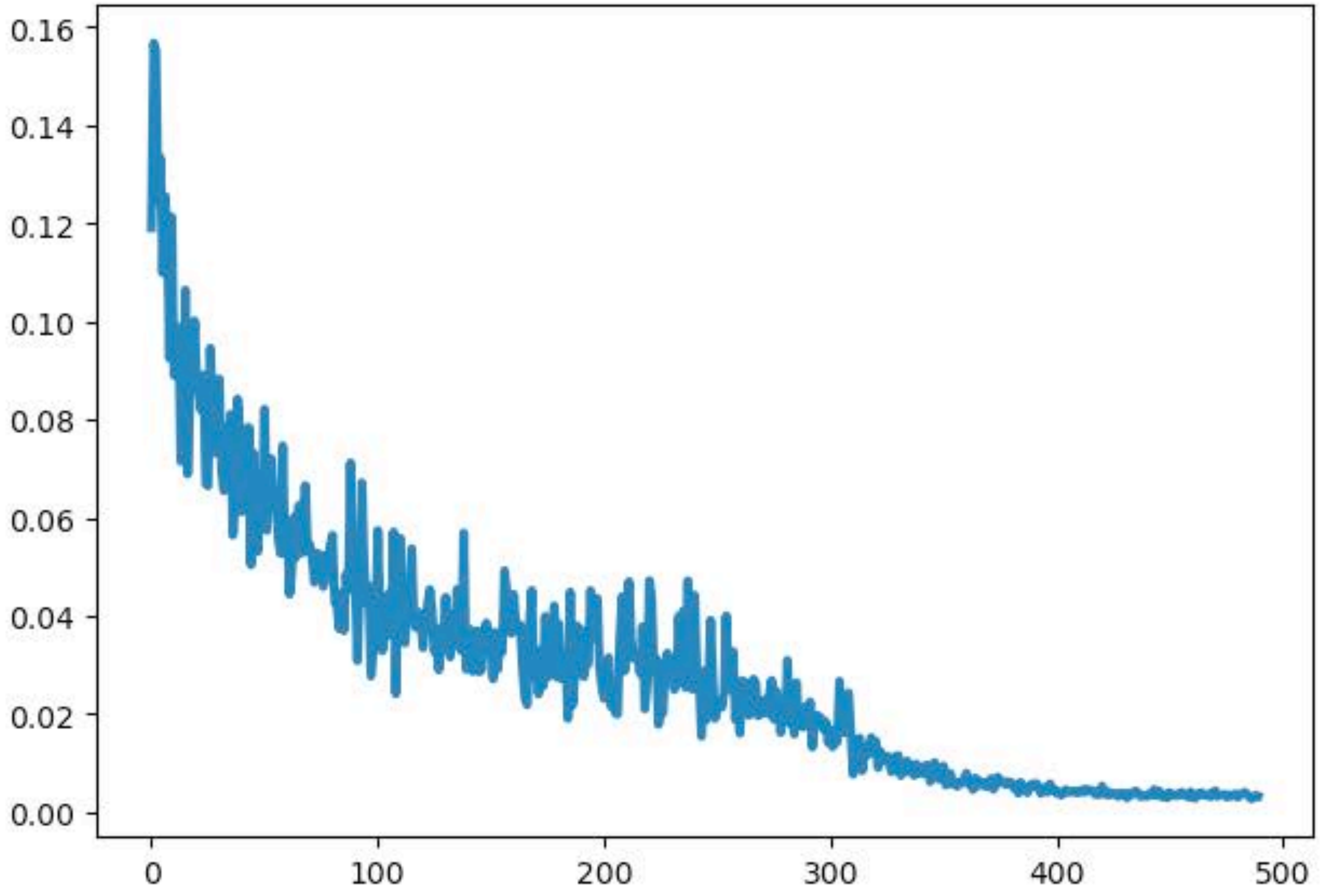




3 hidden units

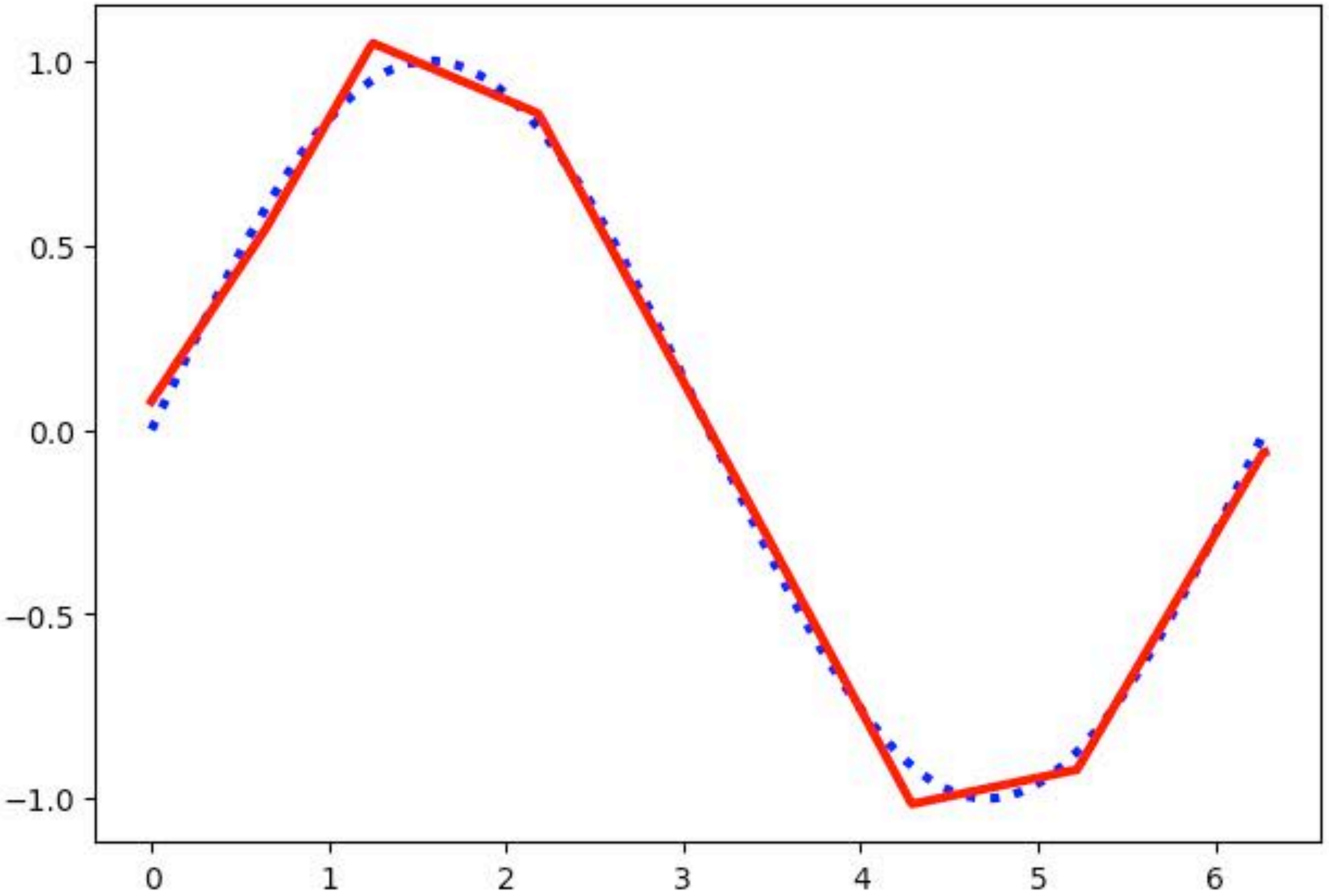
2-Layer **Neural** Network — n hidden, 1 input/output

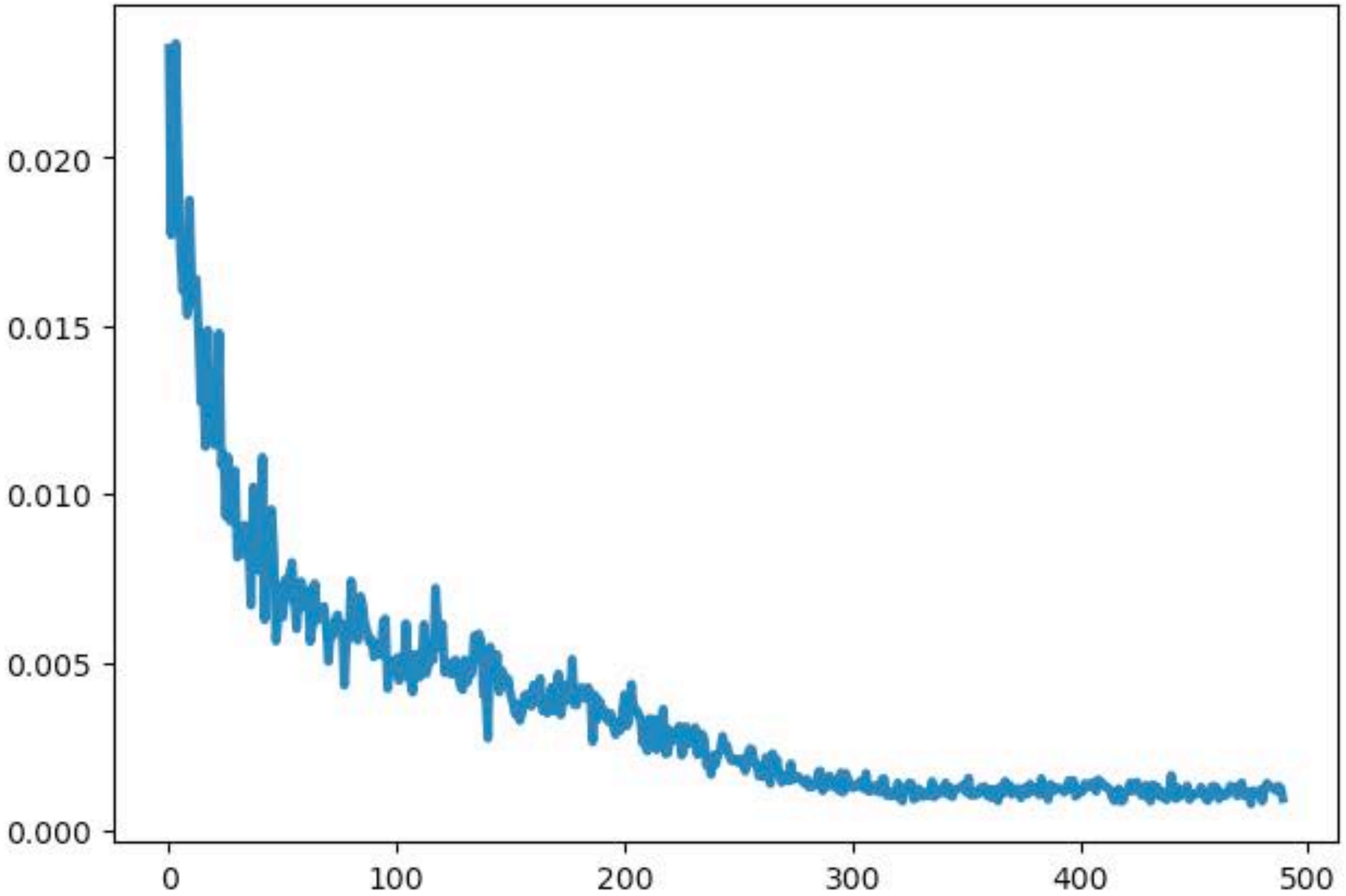




4 hidden units

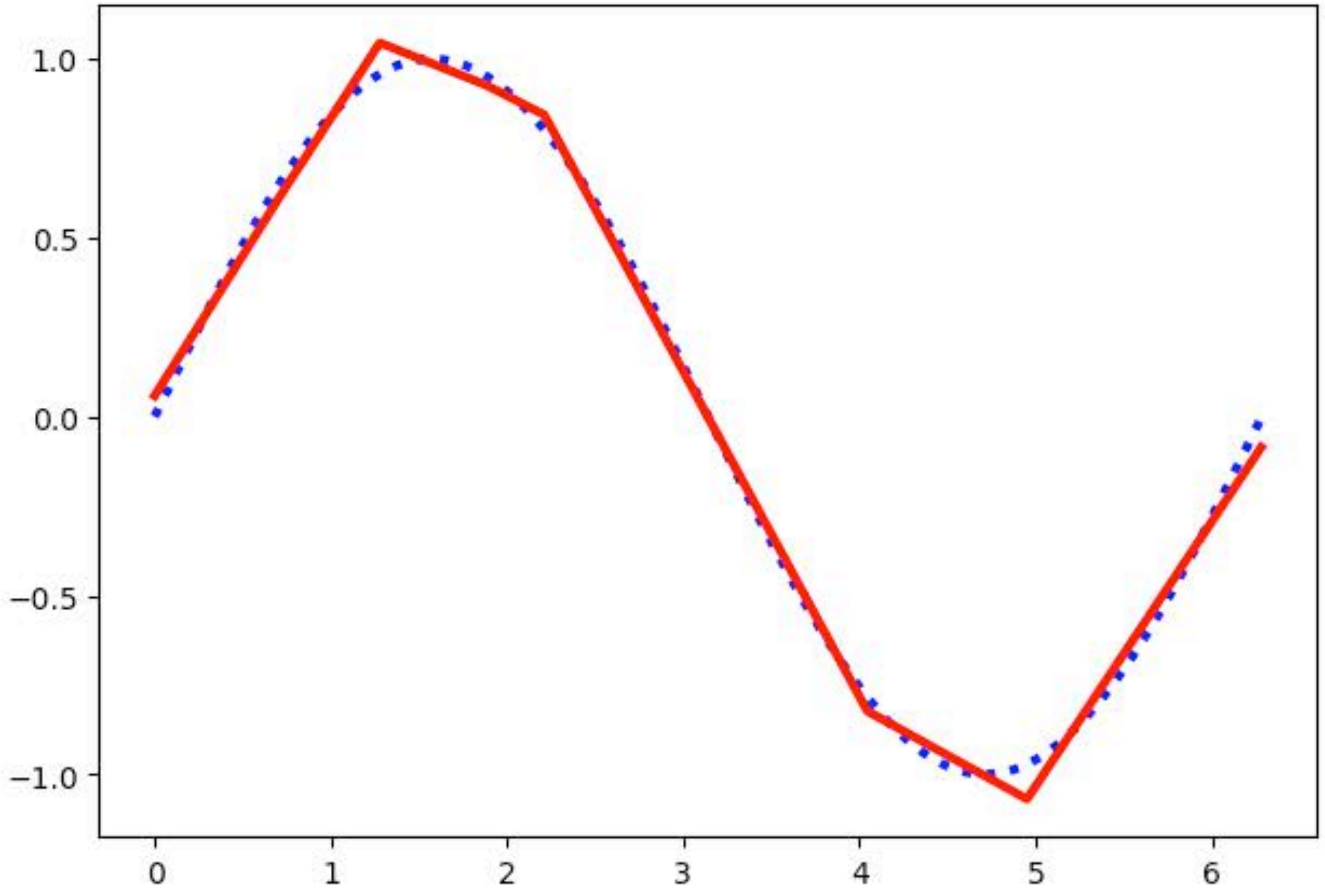
2-Layer **Neural** Network — n hidden, 1 input/output

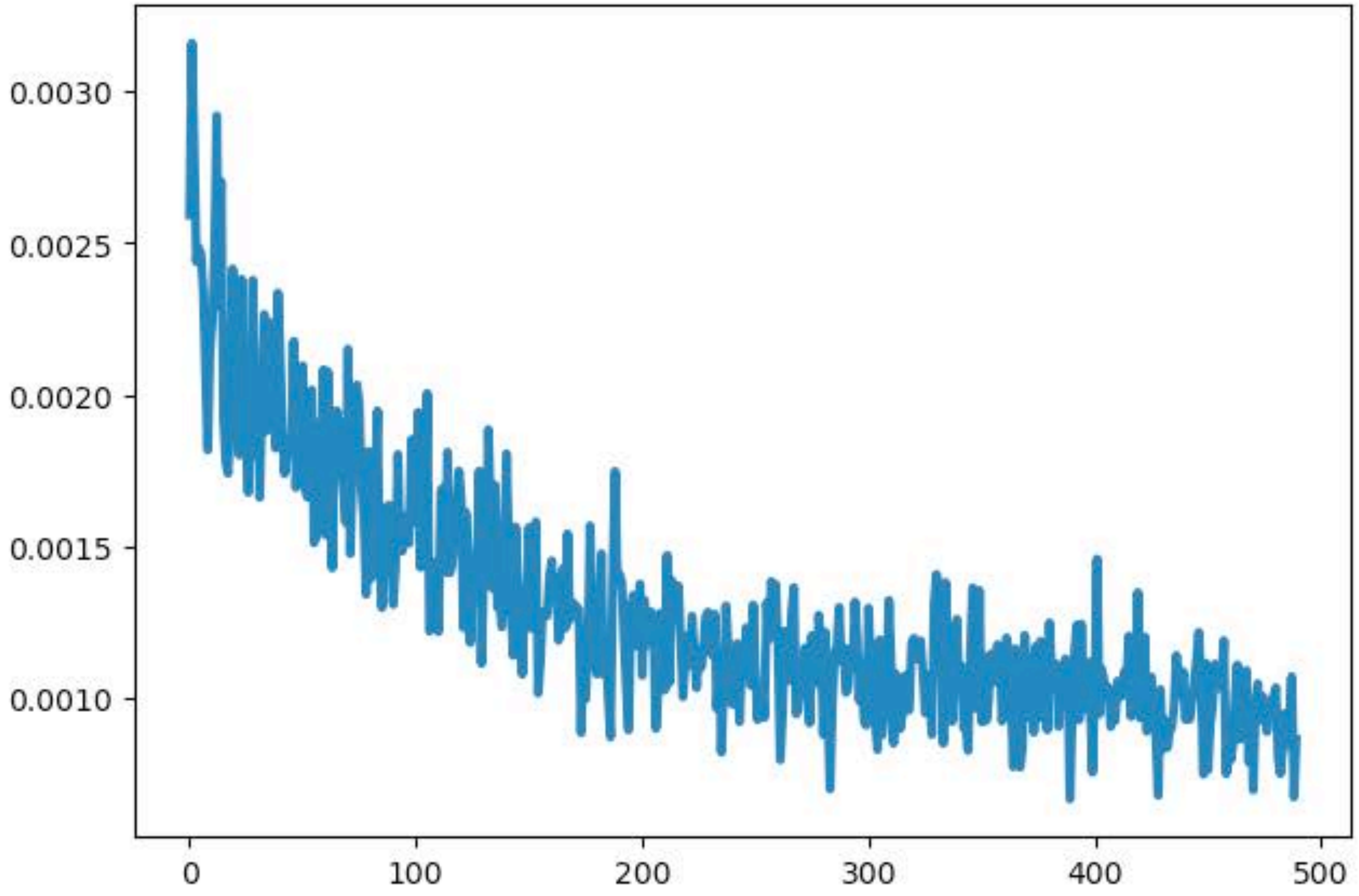




6 hidden units

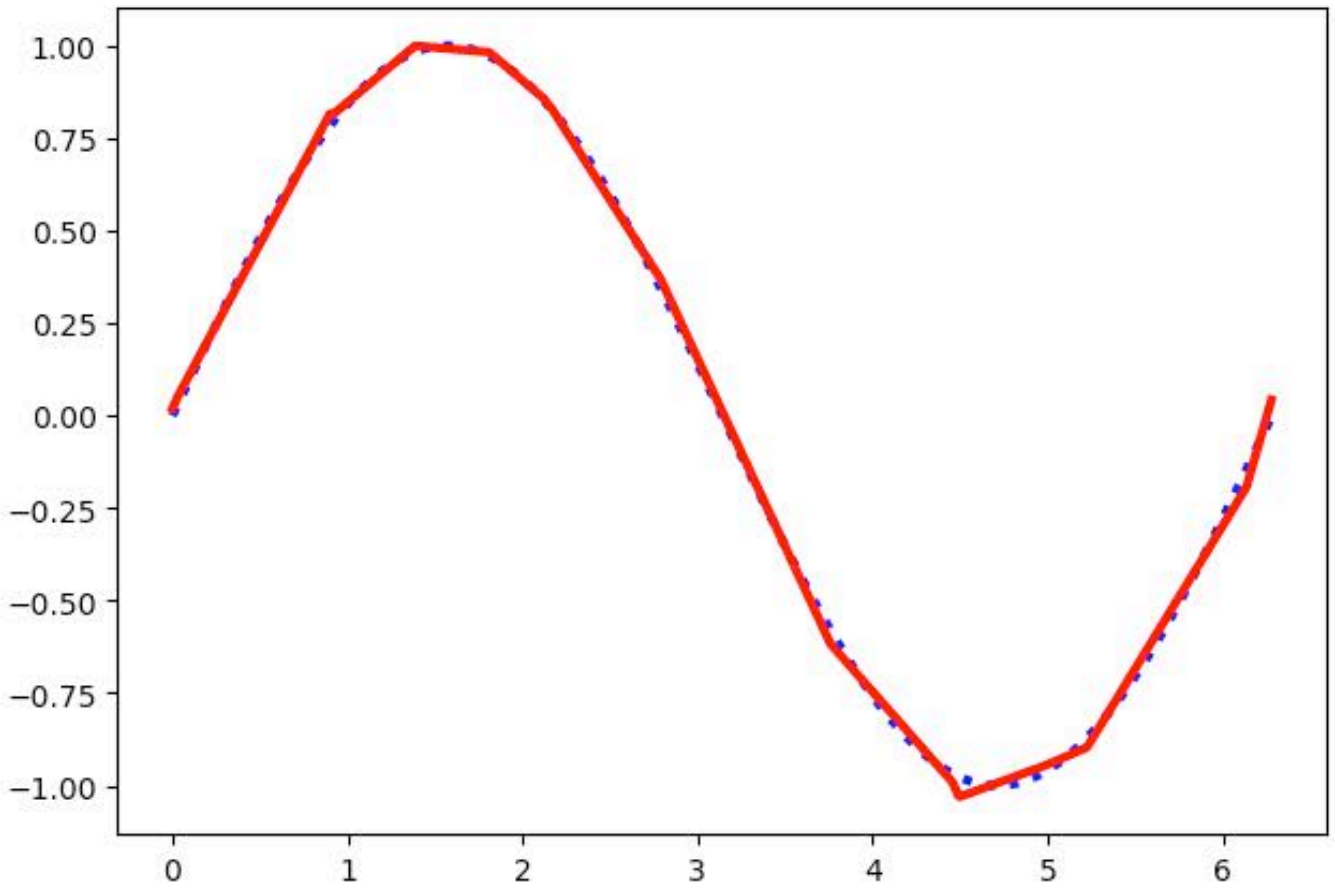
2-Layer **Neural** Network — n hidden, 1 input/output

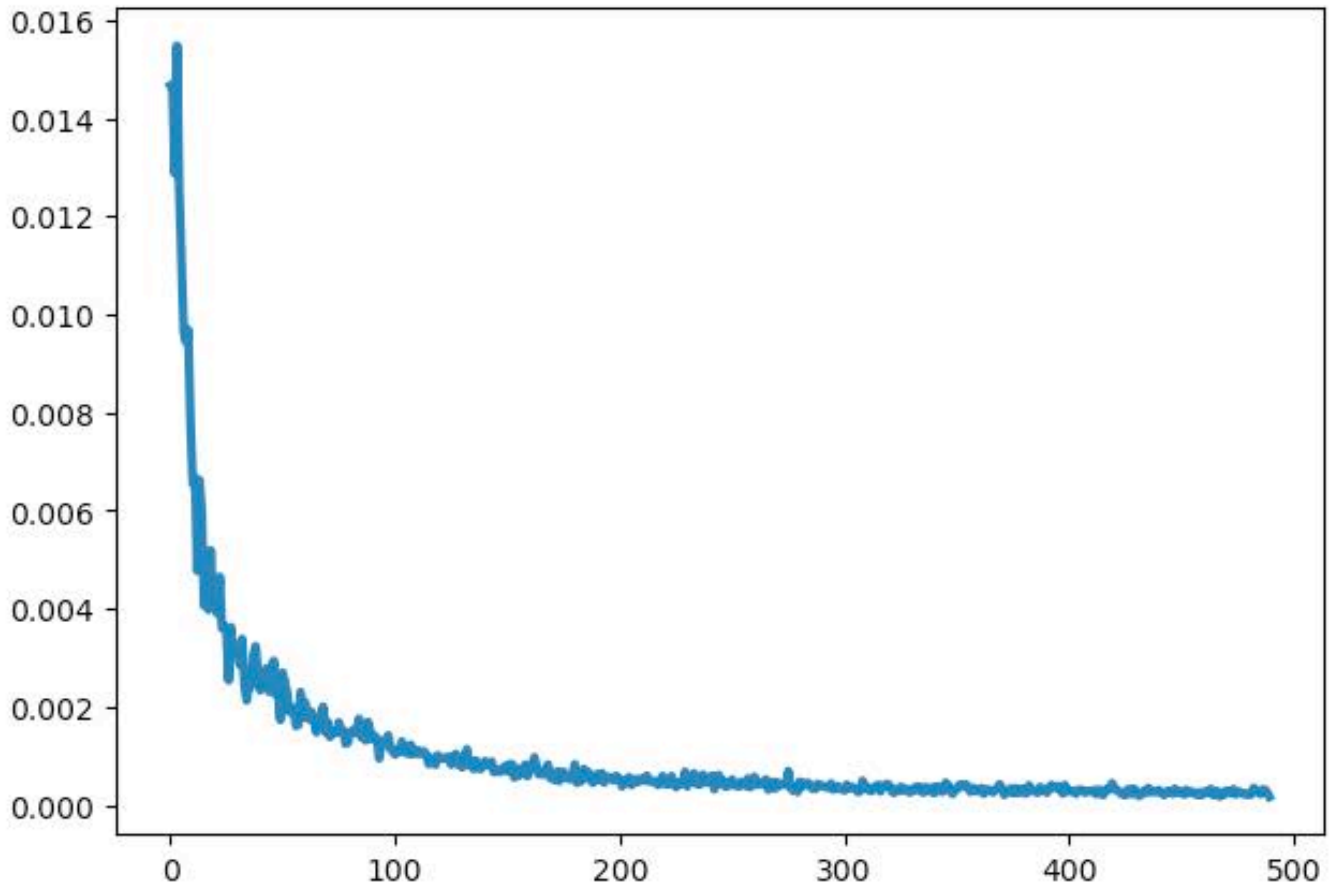




8 hidden units

2-Layer **Neural** Network — n hidden, 1 input/output

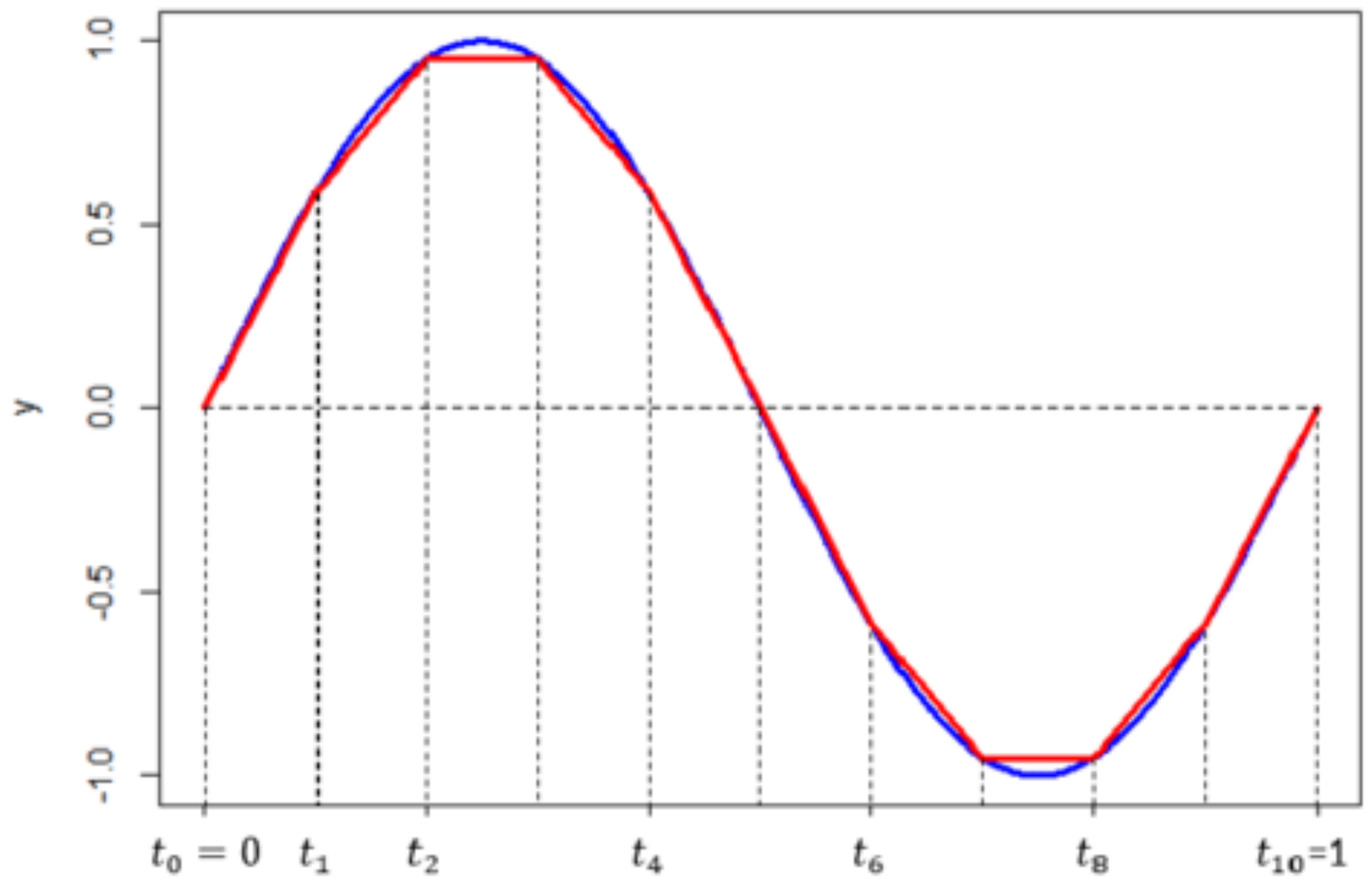




20 hidden units

Neural Network as Universal Approximator

Non-linear activation is required to provably make the Neural Net a **universal function approximator**



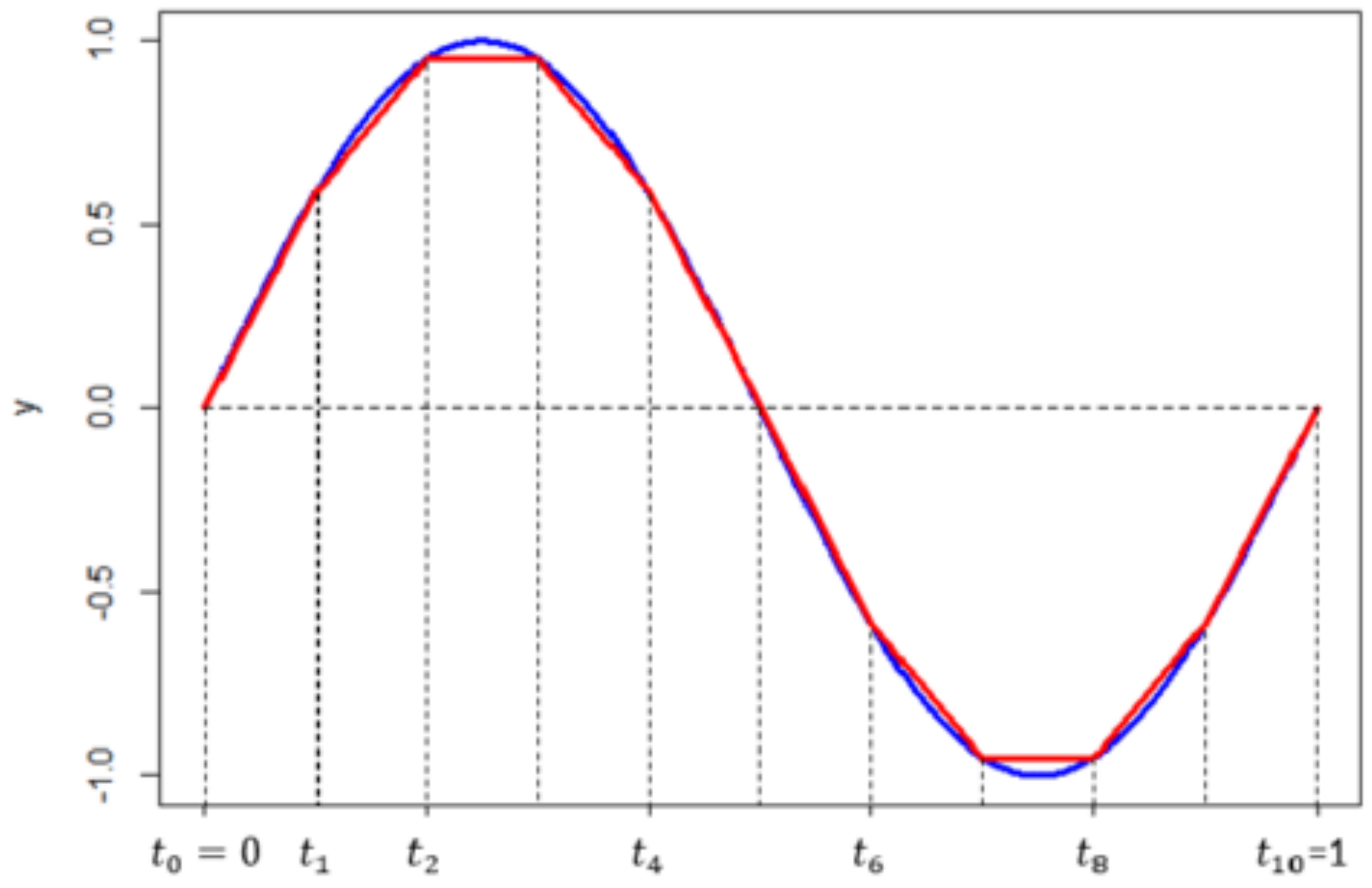
Neural Network as Universal Approximator

Non-linear activation is required to provably make the Neural Net a **universal function approximator**

Intuition: with ReLU activation, we effectively get a linear spline approximation to any function.

Optimization of neural net parameters = finding slops and transitions of linear pieces

The quality of approximation depends on the number of linear segments



Neural Network as Universal Approximator

$d+1$

d

Neural Network as Universal Approximator

Universal Approximation Theorem: Single hidden layer can approximate any continuous function with compact support to arbitrary accuracy, when the width goes to infinity.

[Hornik *et al*., 1989]

$d + 1$ d

Neural Network as Universal Approximator

Universal Approximation Theorem: Single hidden layer can approximate any continuous function with compact support to arbitrary accuracy, when the width goes to infinity.

[Hornik *et al*., 1989]

Universal Approximation Theorem (revised): A network of infinite depth with a hidden layer of size $d+1$ neurons, where d is the dimension of the input space, can approximate any continuous function.

[Lu *et al*., NIPS 2017]

Neural Network as Universal Approximator

Universal Approximation Theorem: Single hidden layer can approximate any continuous function with compact support to arbitrary accuracy, when the width goes to infinity.

[Hornik *et al*., 1989]

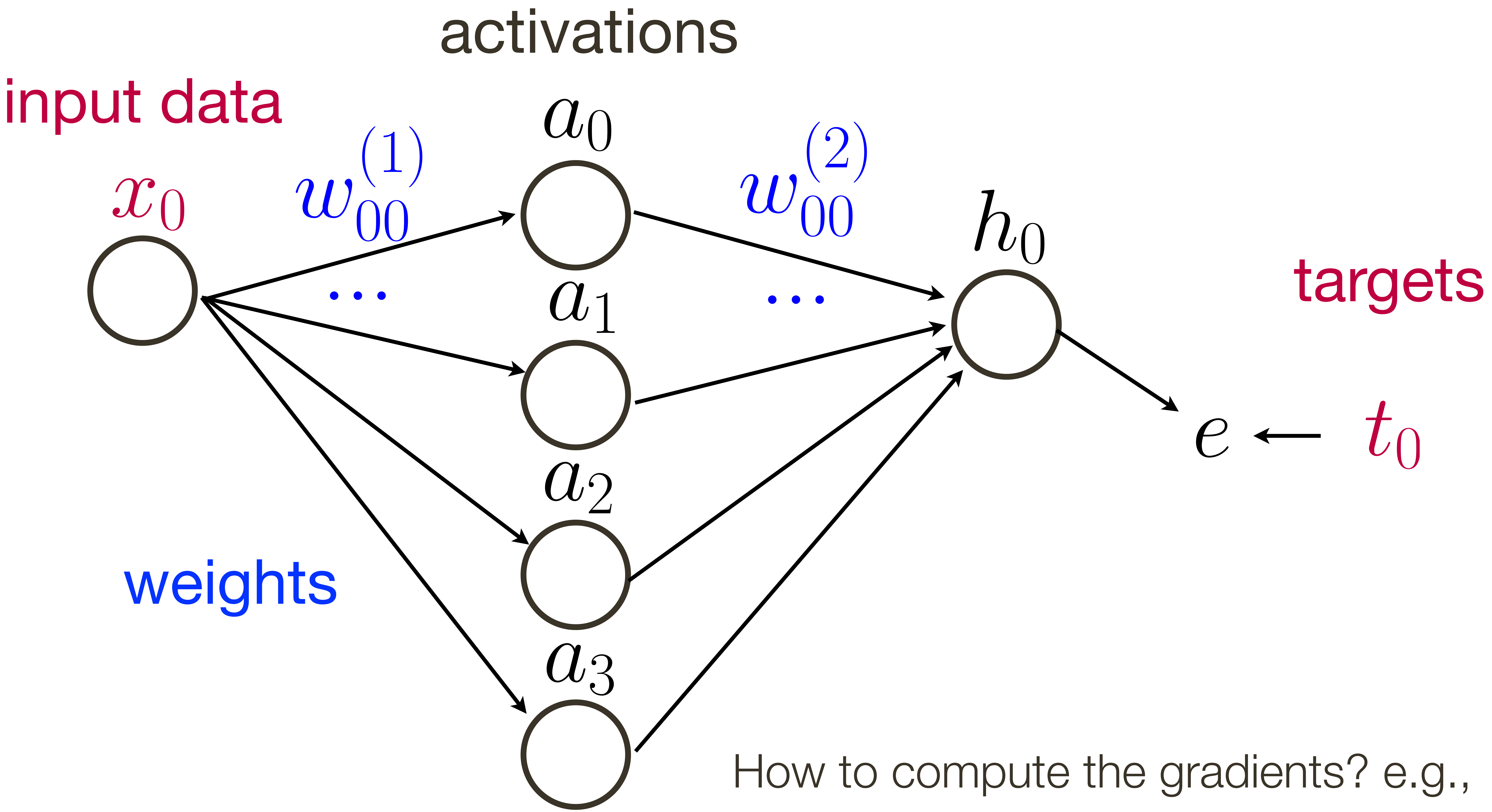
Universal Approximation Theorem (revised): A network of infinite depth with a hidden layer of size $d + 1$ neurons, where d is the dimension of the input space, can approximate any continuous function.

[Lu *et al*., NIPS 2017]

Universal Approximation Theorem (further revised): ResNet with a single hidden unit and infinite depth can approximate any continuous function.

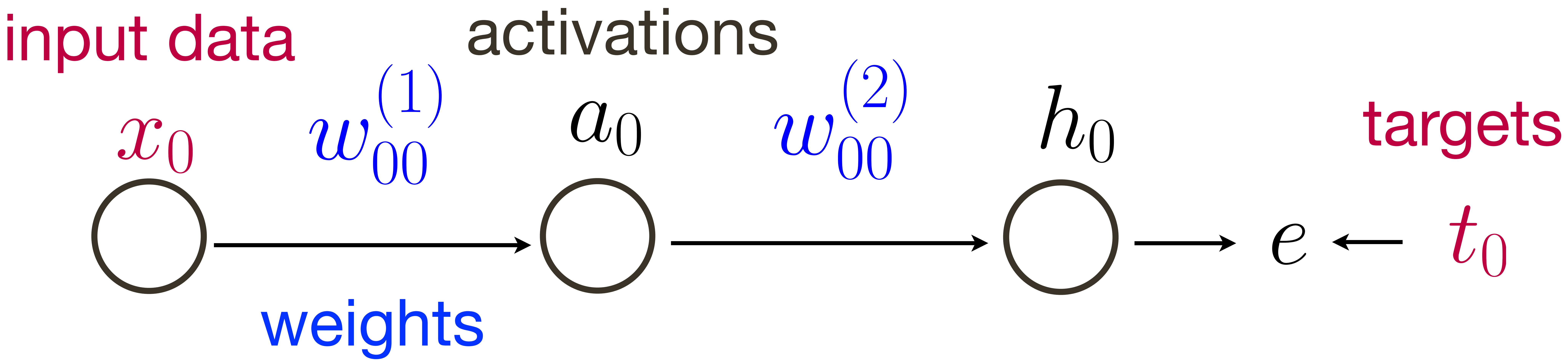
[Lin and Jegelka, NIPS 2018]

2-Layer **Neural** Network — n hidden, 1 input/output



How to compute the gradients? e.g., $\dfrac{\partial e}{\partial w^{(1)}_{00}}$

2-Layer **Neural** Network — 1 hidden, 1 input/output



2-Layer **Neural** Network — 1 hidden, 1 input/output

$$y = w_2(\max(0, w_1 x + b_1)) + b_2 \qquad\qquad L = (y - t)^2$$

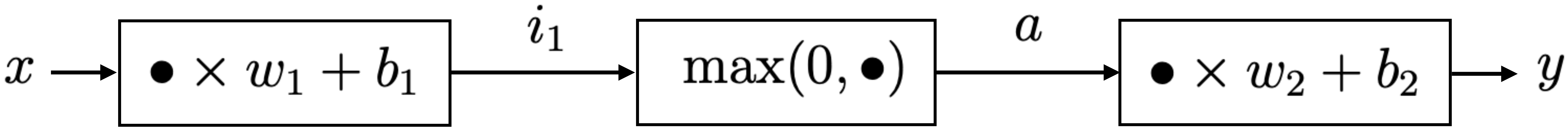
Optimise by **gradient descent**

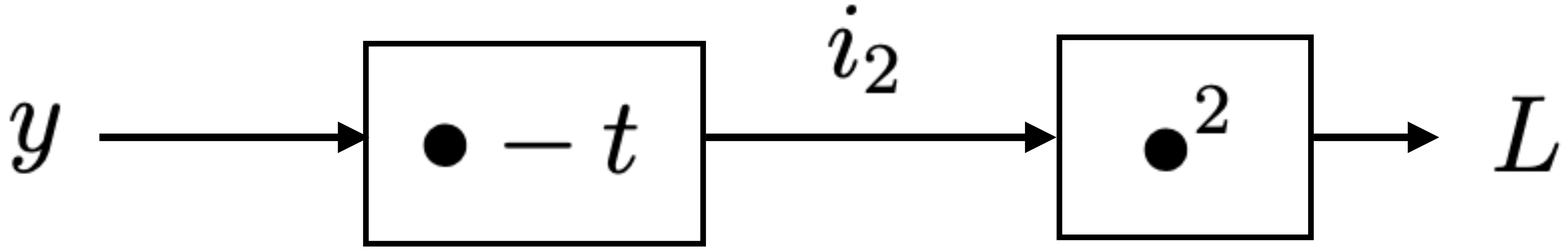
$$\begin{bmatrix} w_1 \\ b_1 \\ w_2 \\ b_2 \end{bmatrix} \to \begin{bmatrix} w_1 \\ b_1 \\ w_2 \\ b_2 \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial b_1} \\ \frac{\partial L}{\partial w_2} \\ \frac{\partial L}{\partial b_2} \end{bmatrix}$$

19.5 How to compute the gradients? e.g., $\dfrac{\partial L}{\partial w_1}$

2-Layer **Neural** Network — 1 hidden, 1 input/output 19.6

$$y = w_2(\max(0, w_1 x + b_1)) + b_2 \qquad L = (y - t)^2$$

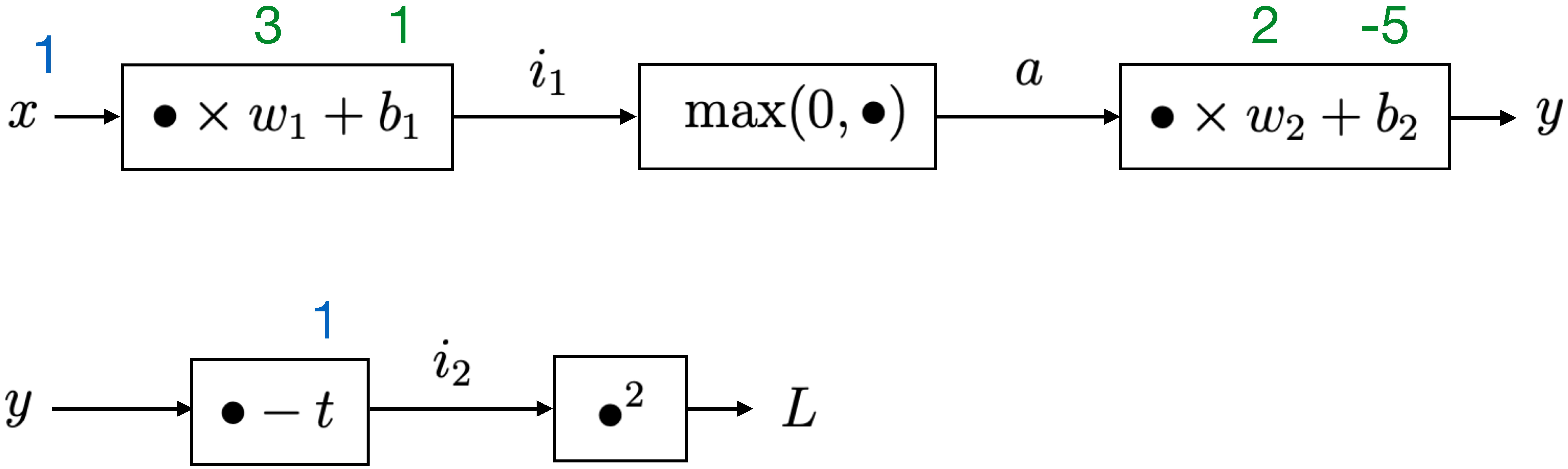




Alternative: build a **computational graph** to apply the **chain rule**

2-Layer **Neural** Network — 1 hidden, 1 input/output 19.6

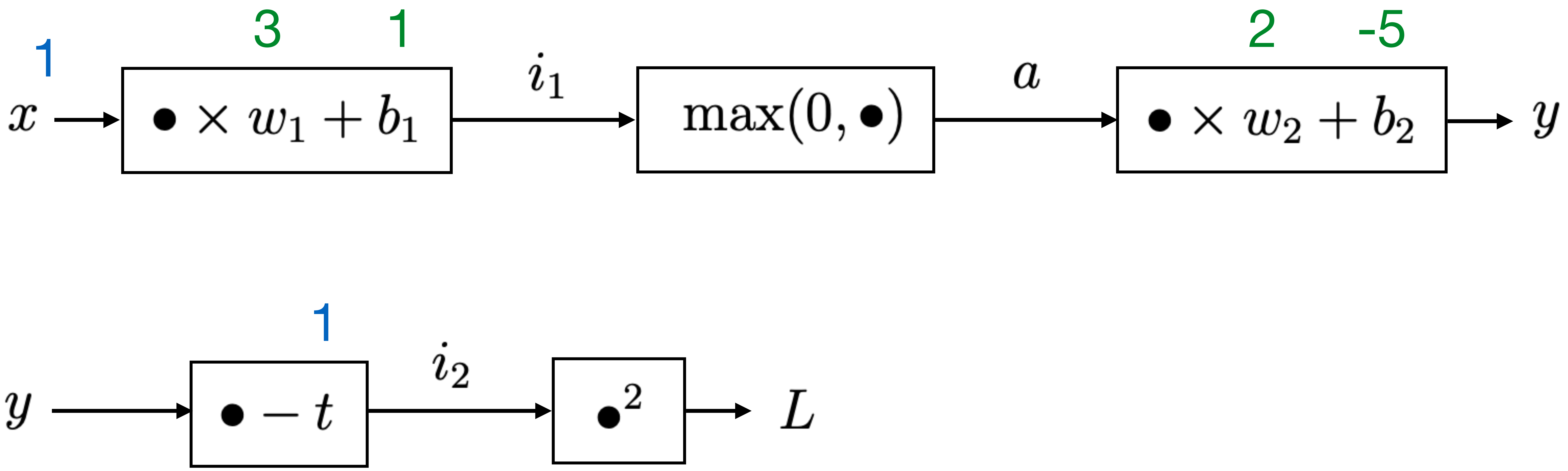
Input + Initial weights
/target



2-Layer **Neural** Network — 1 hidden, 1 input/output

19.6

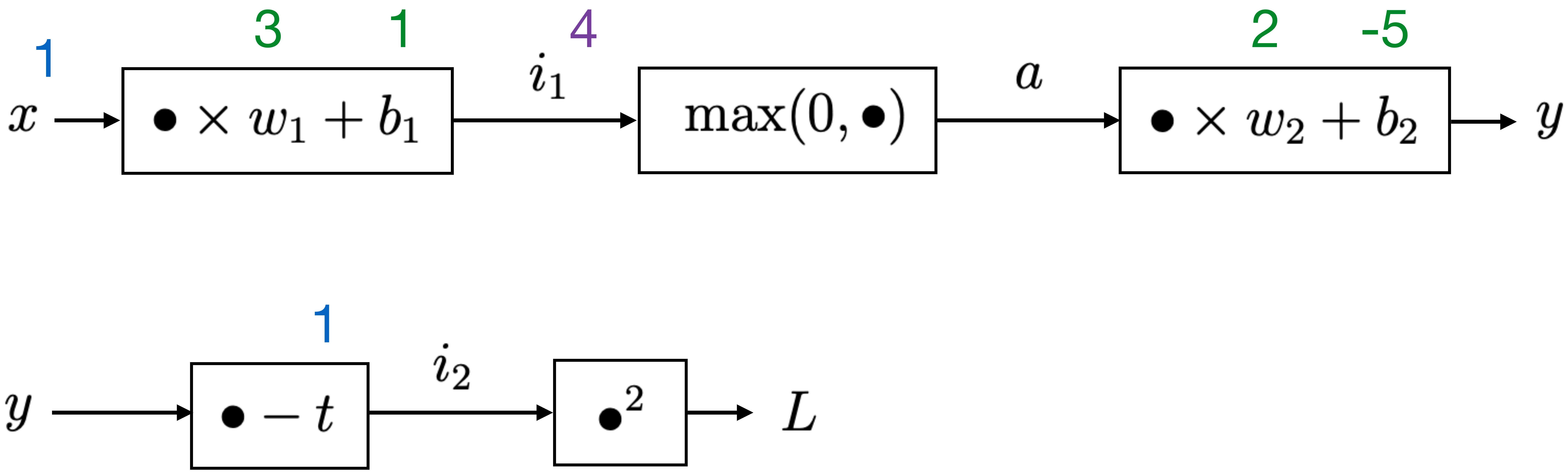
Input + /target   Initial weights   Forward pass



2-Layer **Neural** Network — 1 hidden, 1 input/output

19.6

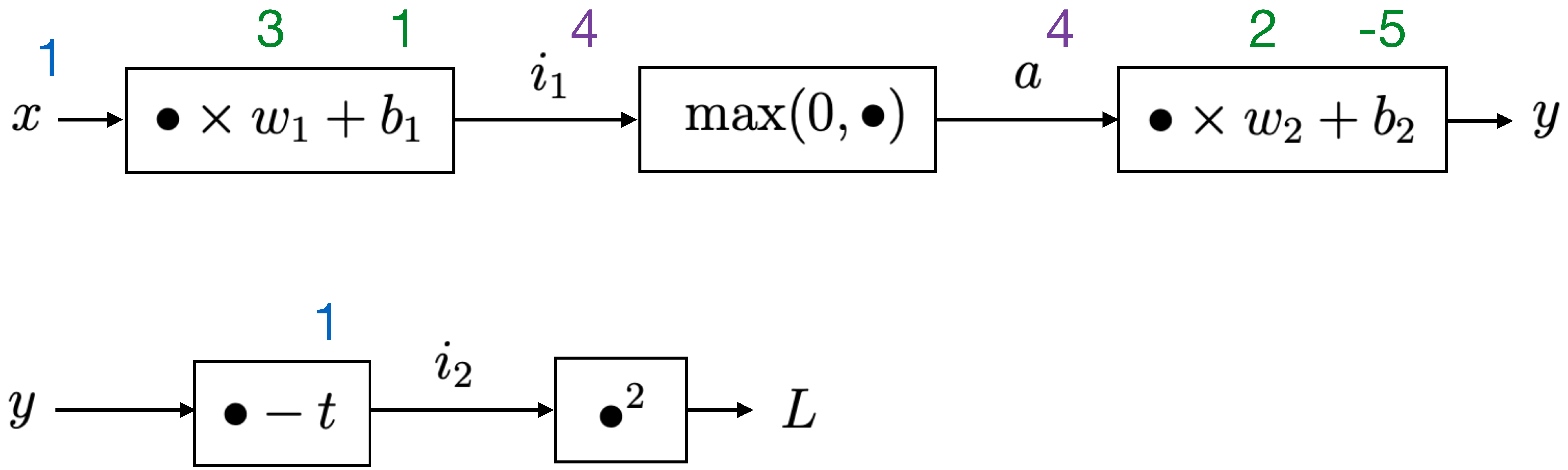
Input + /target   Initial weights   Forward pass



2-Layer **Neural** Network — 1 hidden, 1 input/output

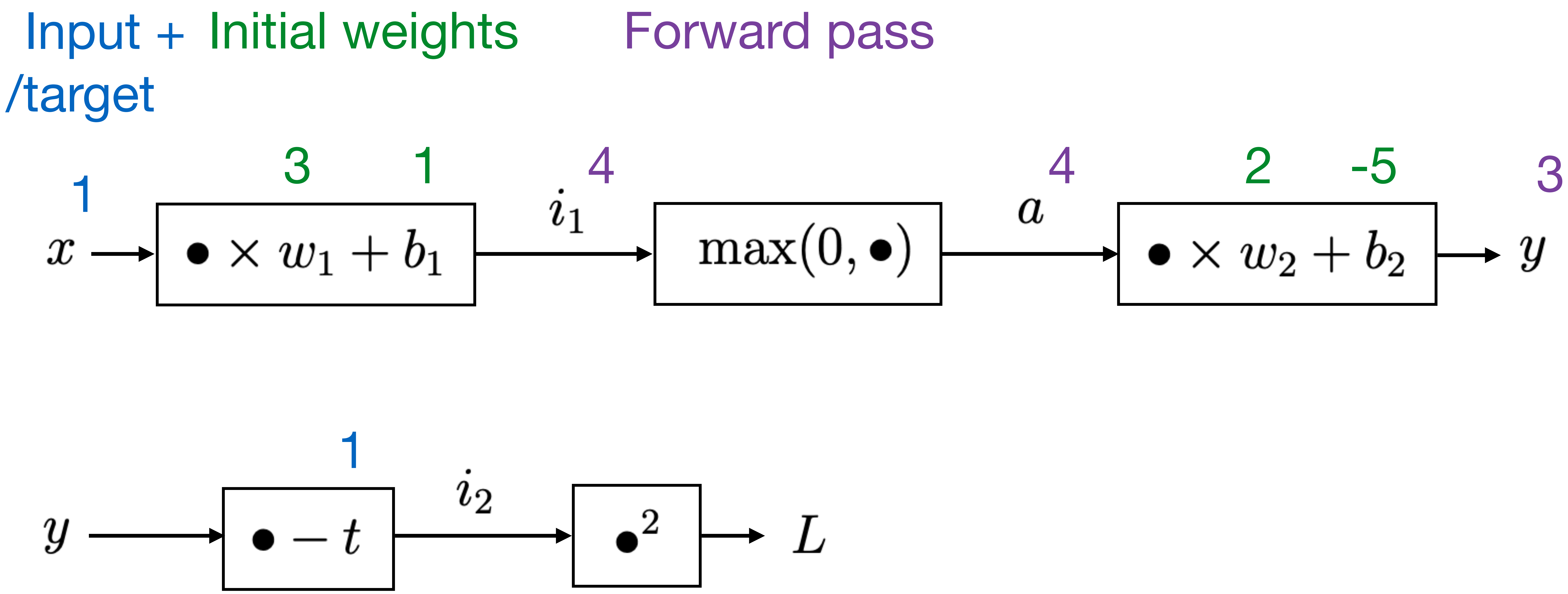
19.6

Input + /target   Initial weights   Forward pass



2-Layer **Neural** Network — 1 hidden, 1 input/output

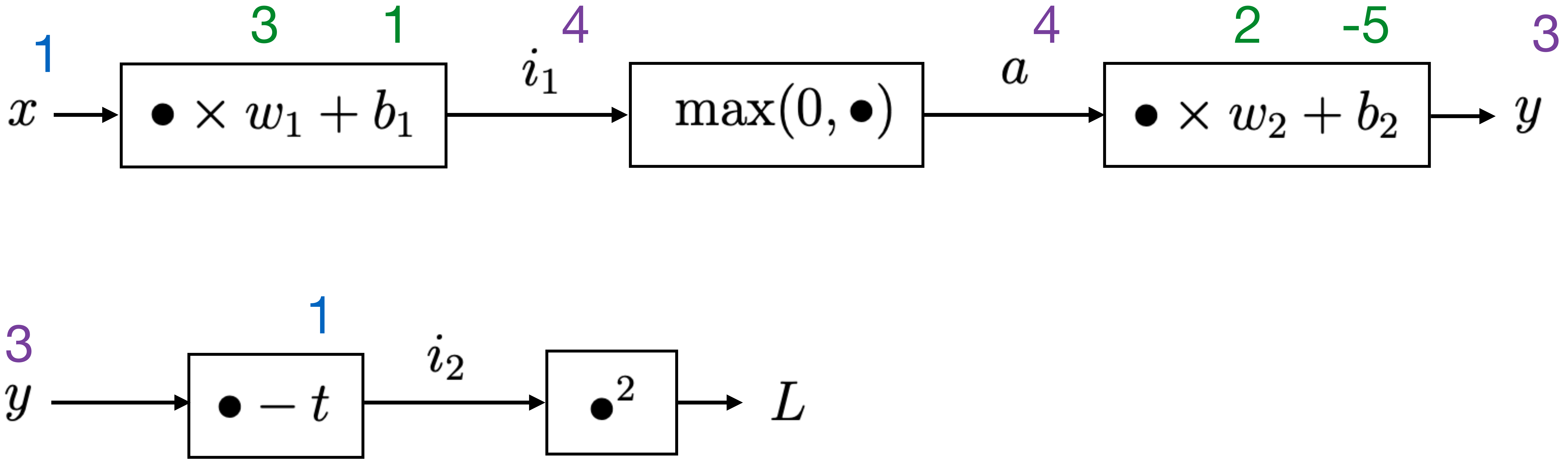
19.6



2-Layer **Neural** Network — 1 hidden, 1 input/output

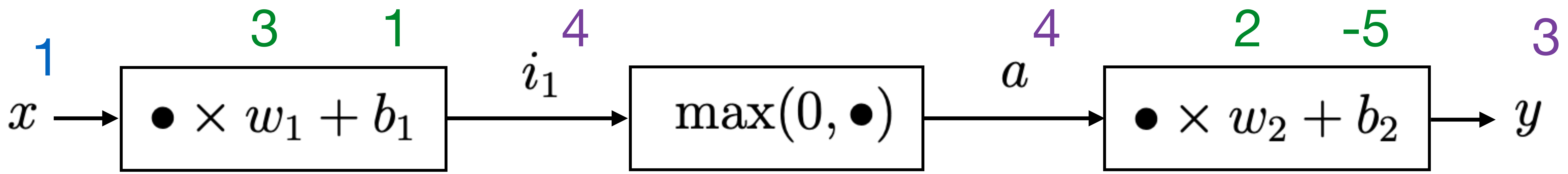
19.6

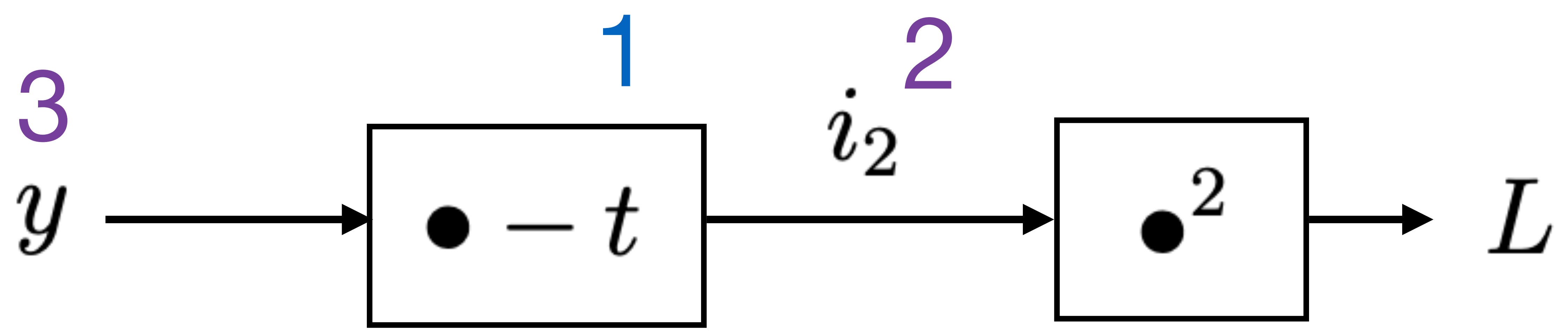
Input + /target Initial weights Forward pass



2-Layer **Neural** Network — 1 hidden, 1 input/output 19.6

Input + /target Initial weights Forward pass

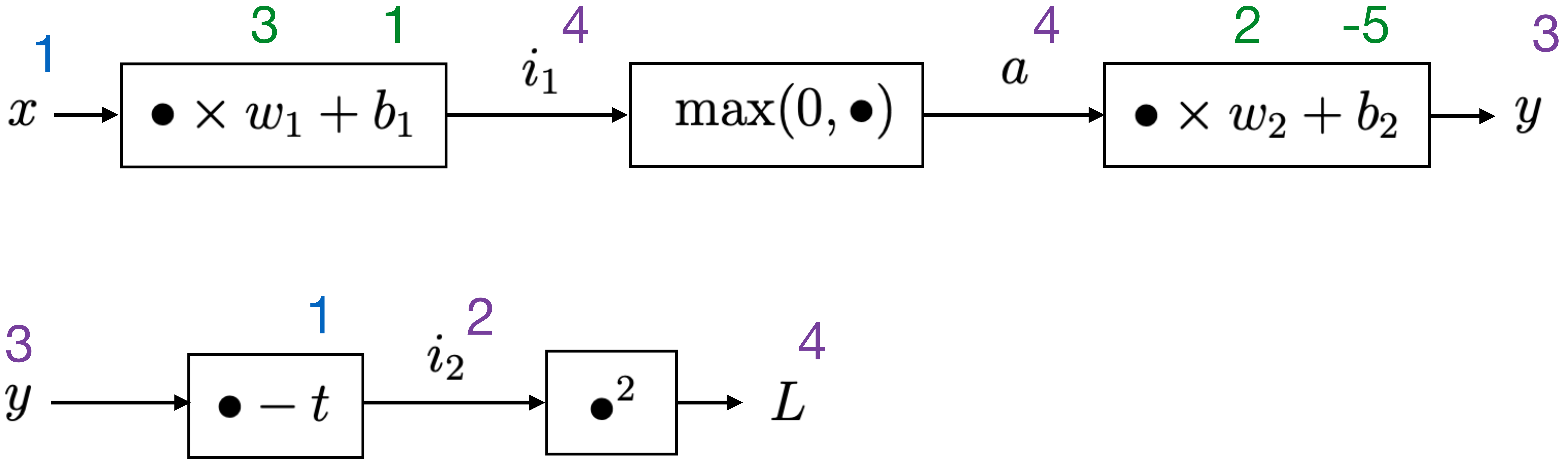




2-Layer **Neural** Network — 1 hidden, 1 input/output

19.6

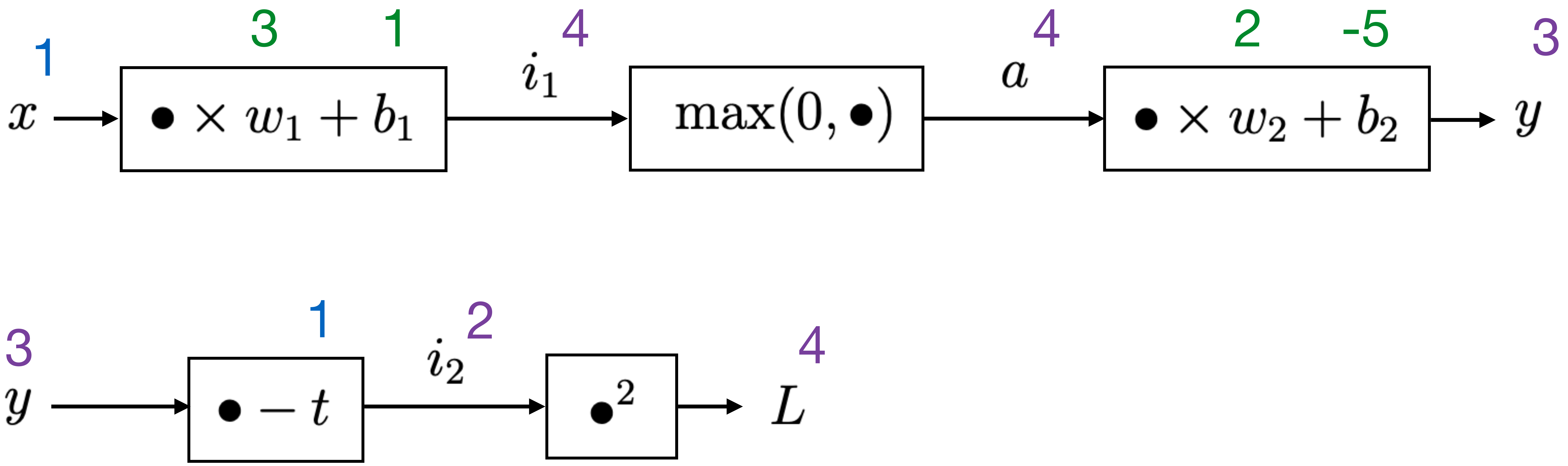
Input + /target   Initial weights   Forward pass



2-Layer **Neural** Network — 1 hidden, 1 input/output

19.6

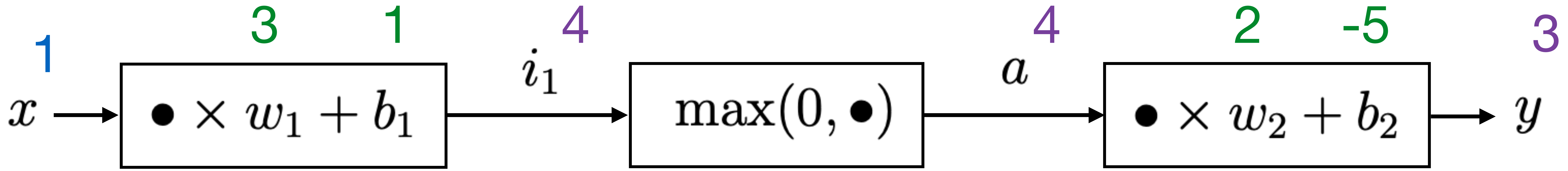
Input + Initial weights /target   Forward pass   Backward pass $= \dfrac{\partial L}{\partial \bullet}$

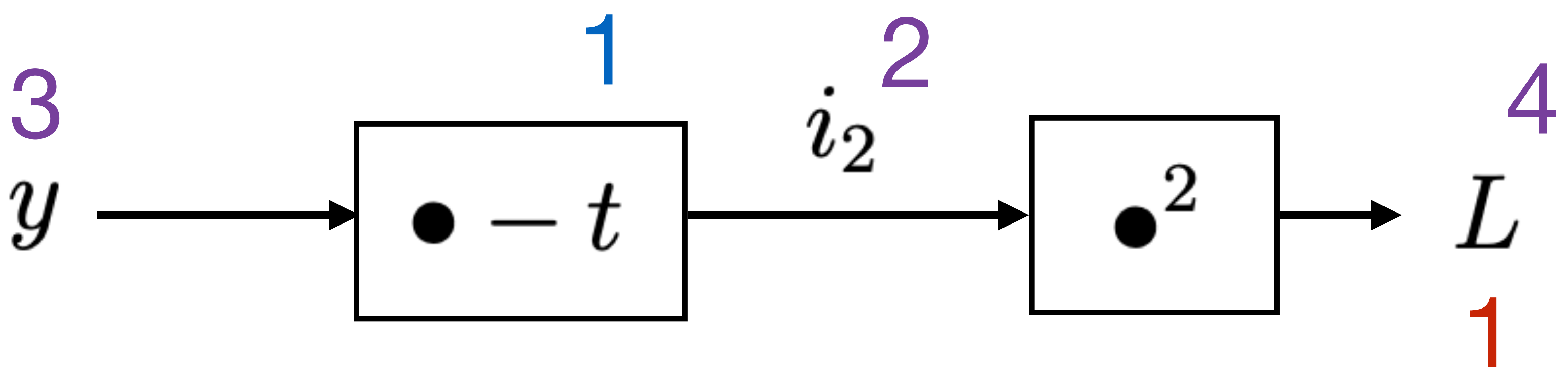


2-Layer **Neural** Network — 1 hidden, 1 input/output

19.6

Input + Initial weights /target   Forward pass   Backward pass $= \frac{\partial L}{\partial \bullet}$

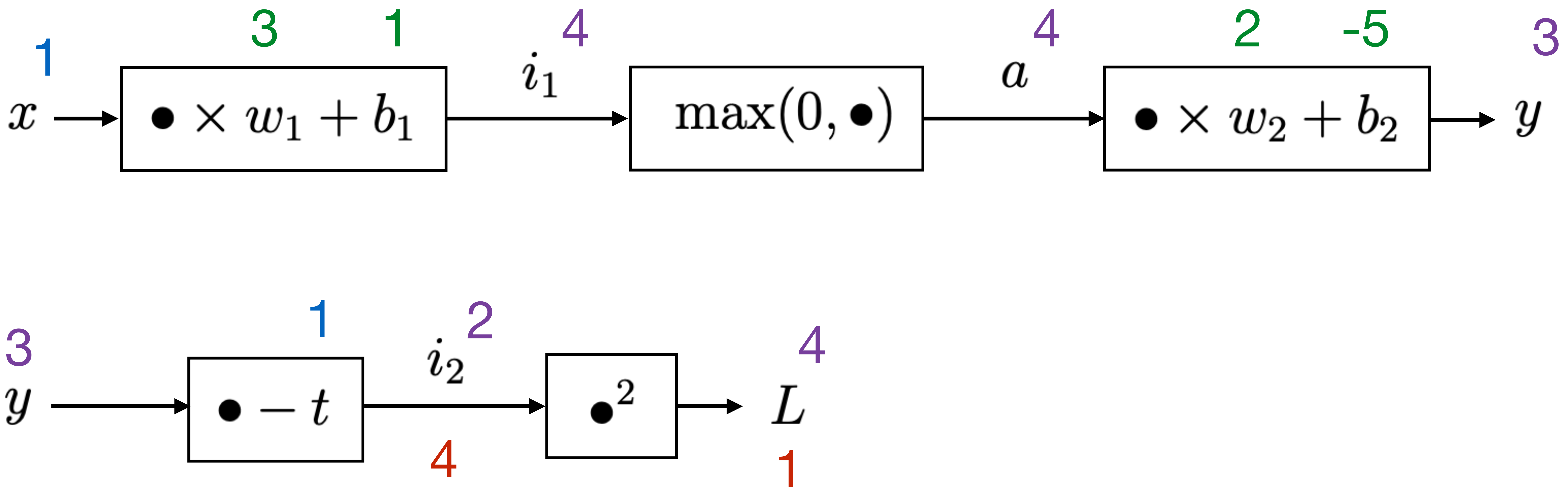




2-Layer **Neural** Network — 1 hidden, 1 input/output

19.6

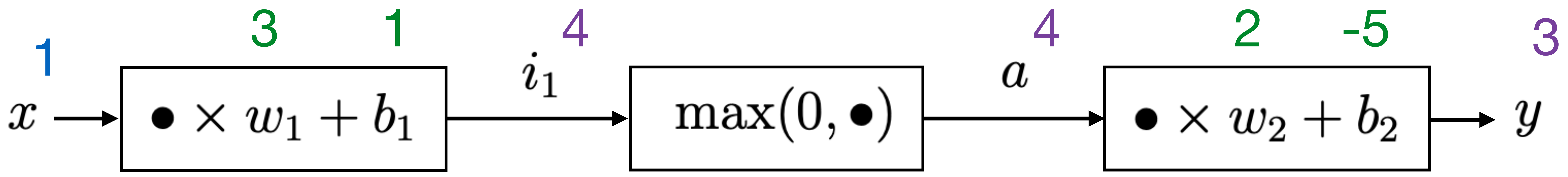
Input + Initial weights /target   Forward pass   Backward pass $= \frac{\partial L}{\partial \bullet}$

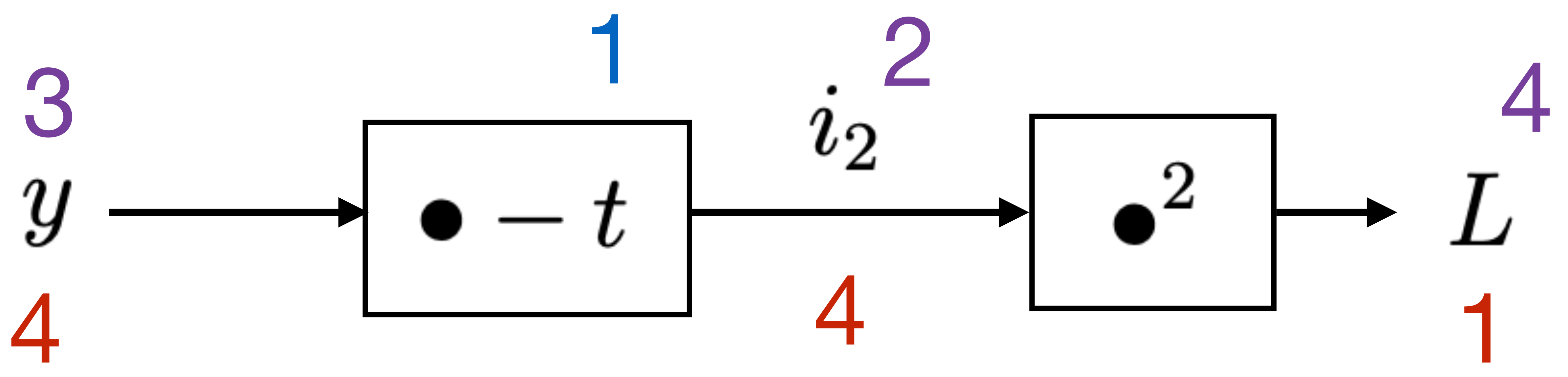


2-Layer **Neural** Network — 1 hidden, 1 input/output

19.6

Input + /target Initial weights Forward pass Backward pass $= \frac{\partial L}{\partial \bullet}$

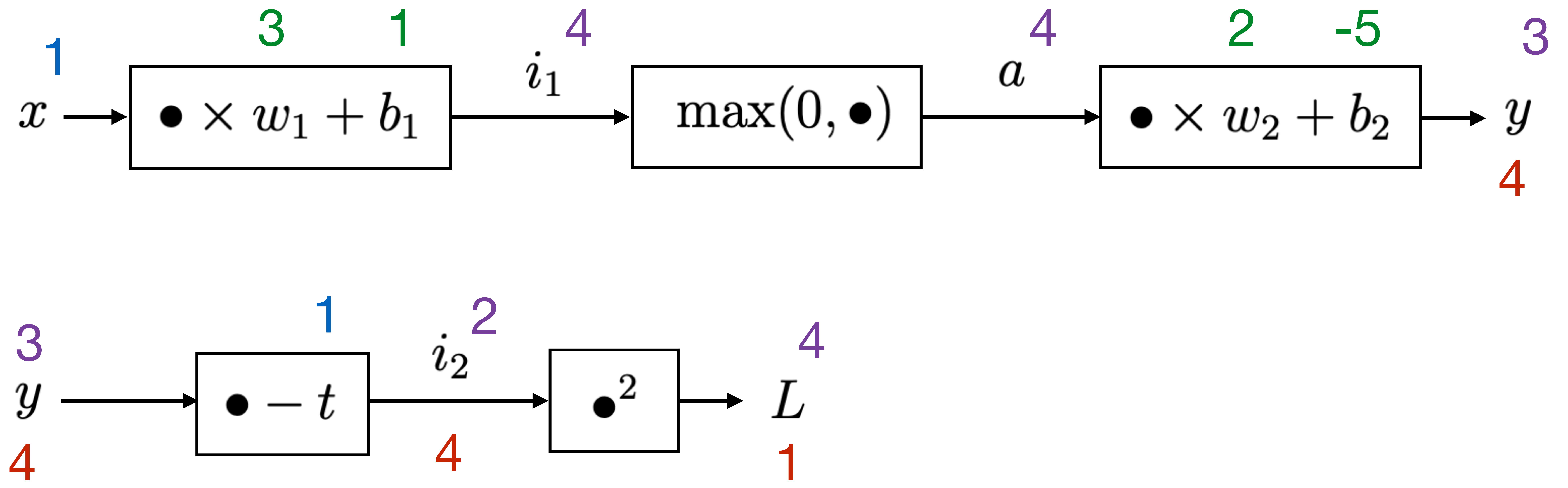




2-Layer **Neural** Network — 1 hidden, 1 input/output

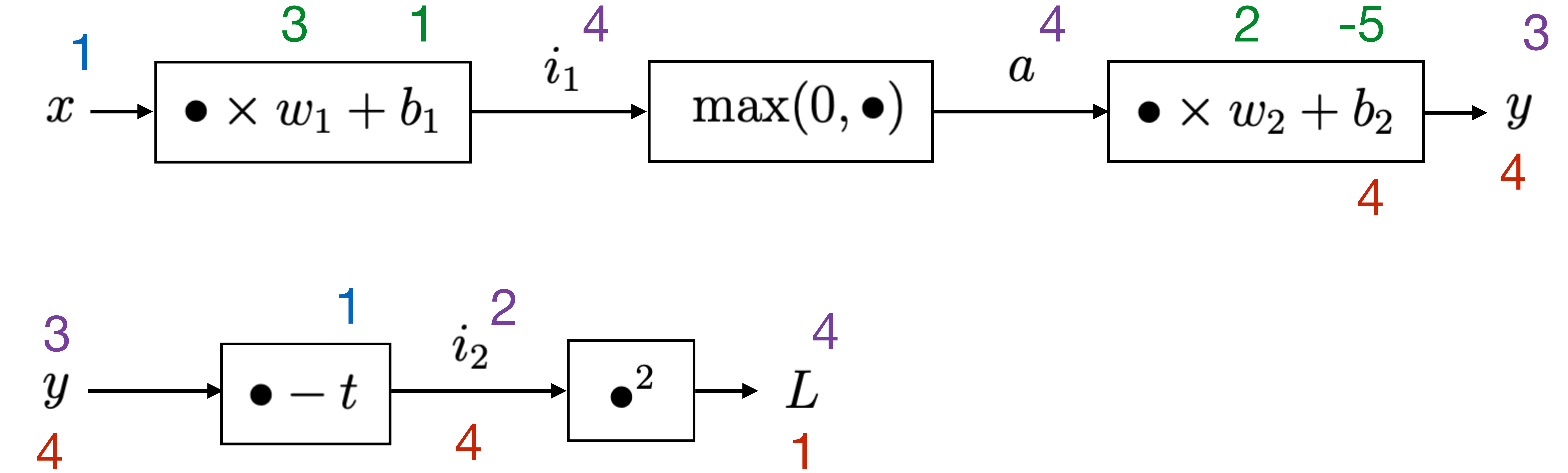
19.6

Input + /target Initial weights Forward pass Backward pass $= \dfrac{\partial L}{\partial \bullet}$



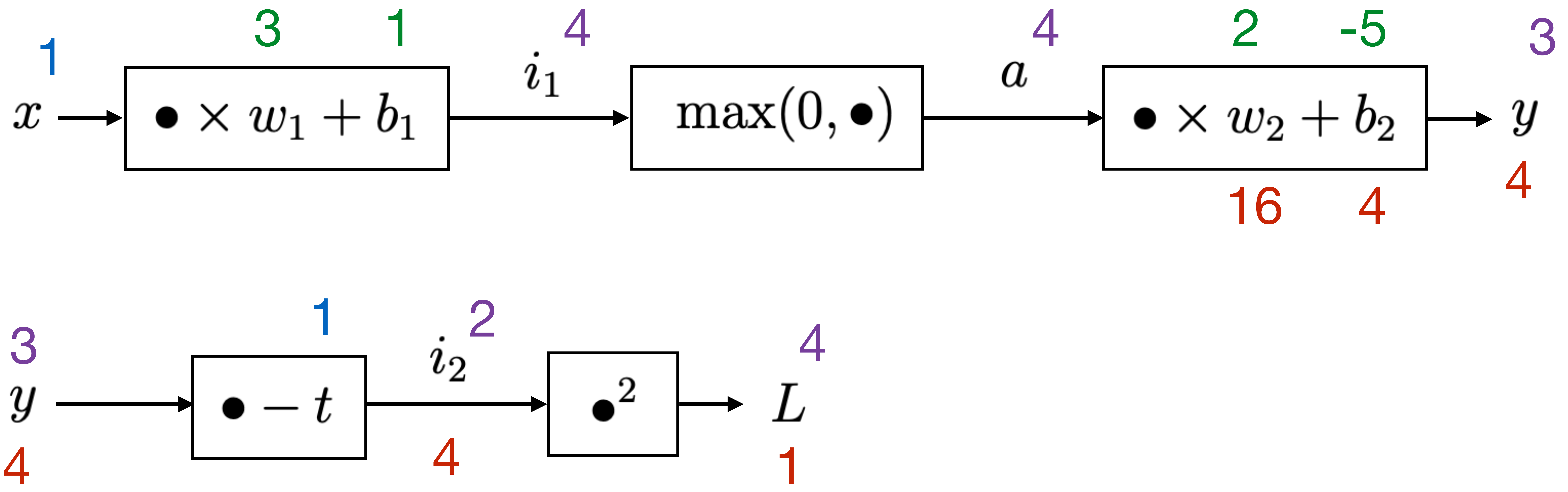
2-Layer **Neural** Network — 1 hidden, 1 input/output 19.6

Input + Initial weights /target   Forward pass   Backward pass $= \dfrac{\partial L}{\partial \bullet}$



2-Layer **Neural** Network — 1 hidden, 1 input/output 19.6

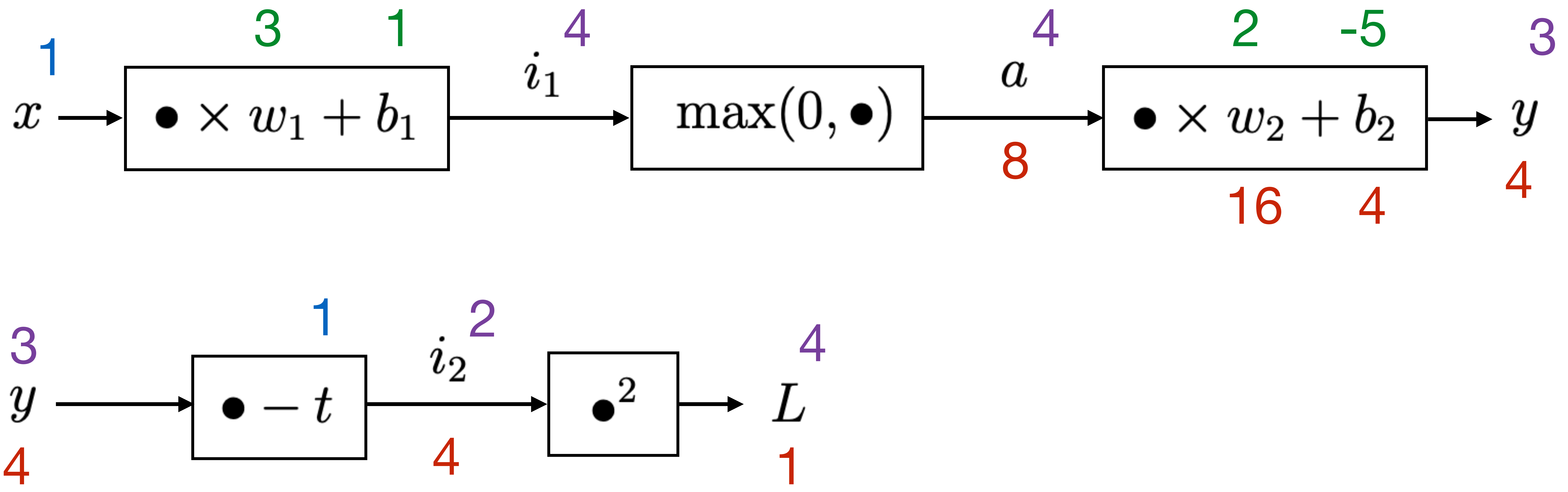
Input + /target Initial weights Forward pass Backward pass $= \dfrac{\partial L}{\partial \bullet}$



2-Layer **Neural** Network — 1 hidden, 1 input/output

19.6

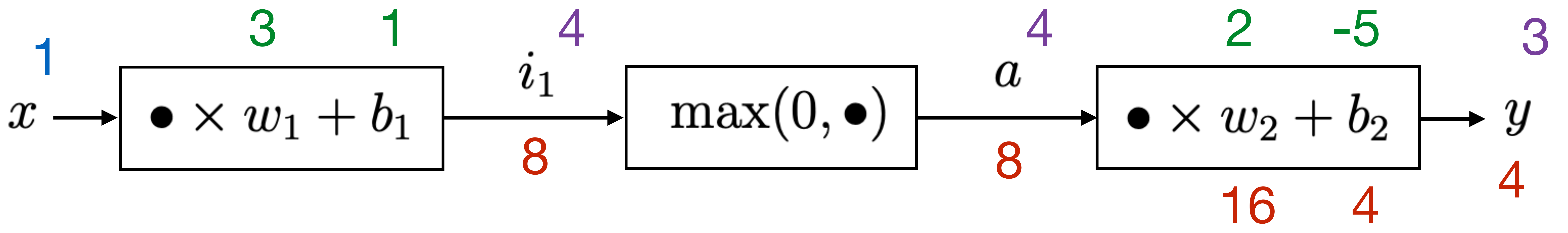
Input + /target Initial weights Forward pass Backward pass $= \frac{\partial L}{\partial \bullet}$

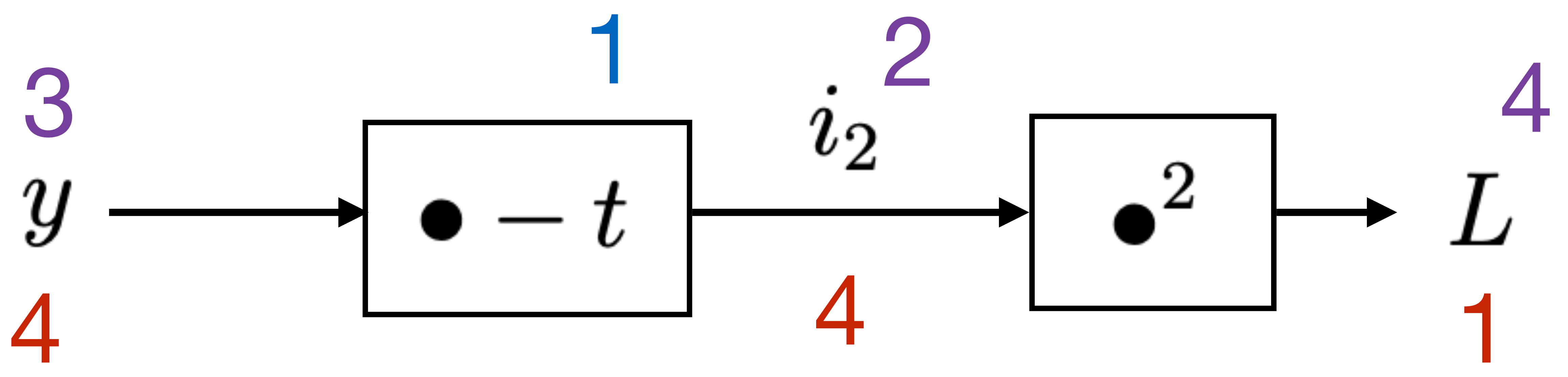


2-Layer **Neural** Network — 1 hidden, 1 input/output

19.6

Input + Initial weights /target   Forward pass   Backward pass $= \dfrac{\partial L}{\partial \bullet}$





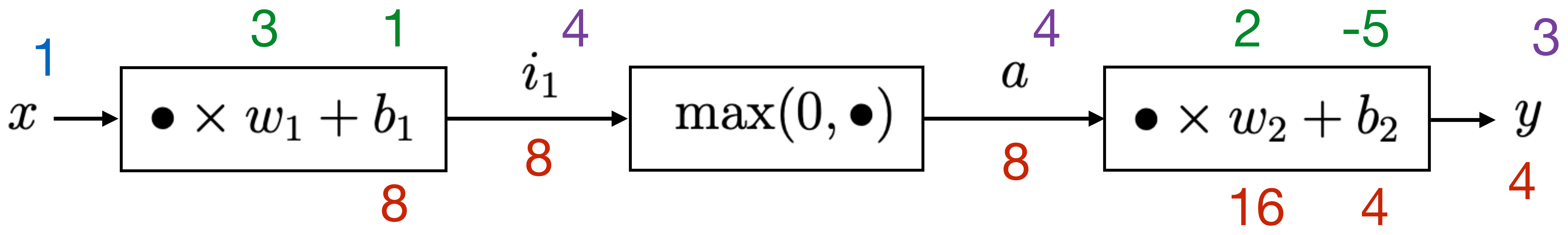
2-Layer **Neural** Network — 1 hidden, 1 input/output

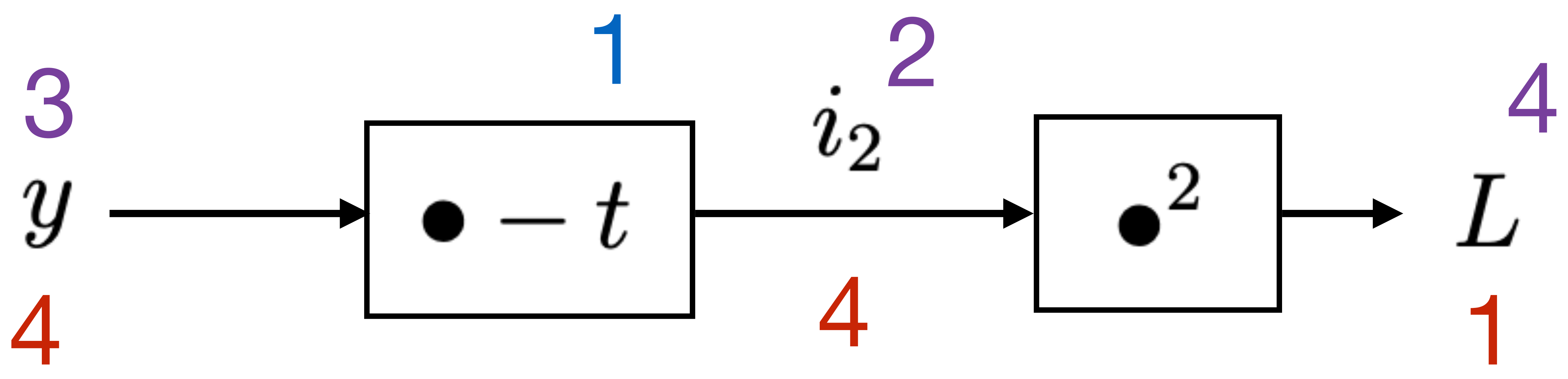
19.6

Input + Initial weights /target

Forward pass

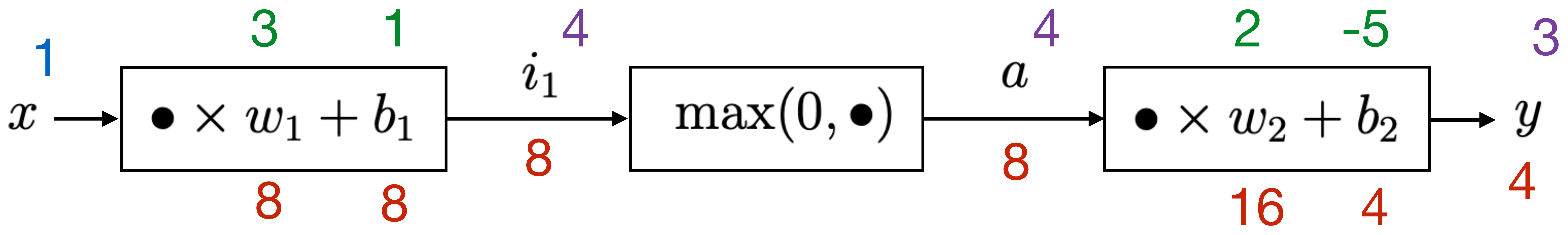
Backward pass $= \dfrac{\partial L}{\partial \bullet}$

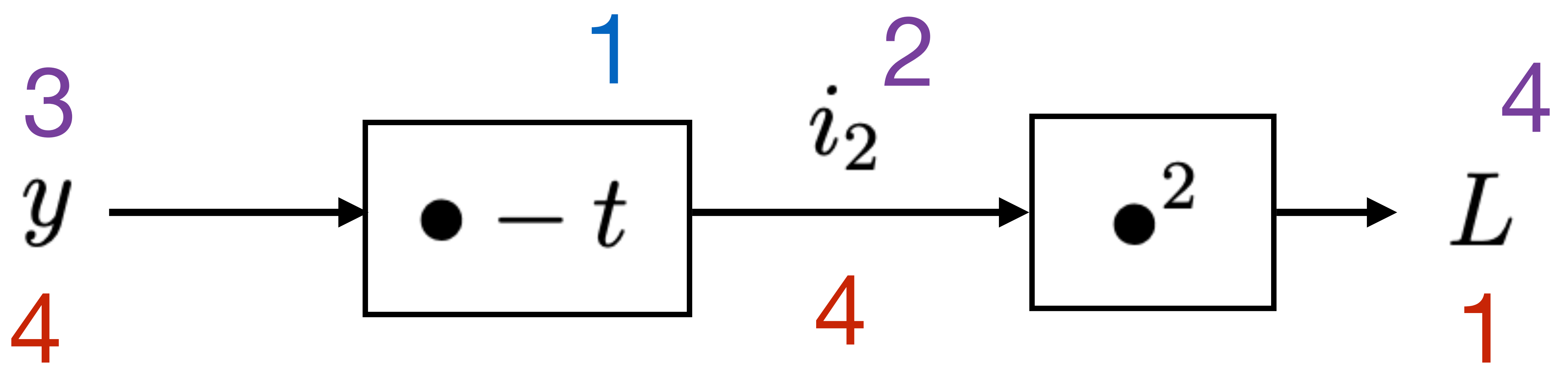




2-Layer **Neural** Network — 1 hidden, 1 input/output 19.6

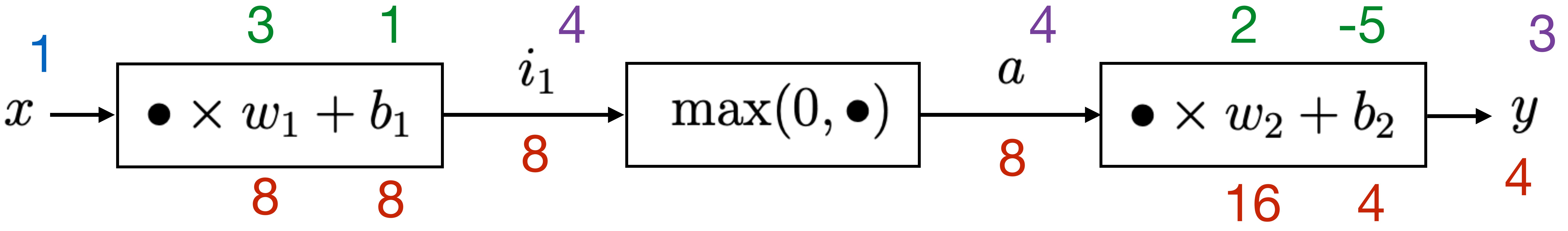
Input + Initial weights /target   Forward pass   Backward pass $= \frac{\partial L}{\partial \bullet}$

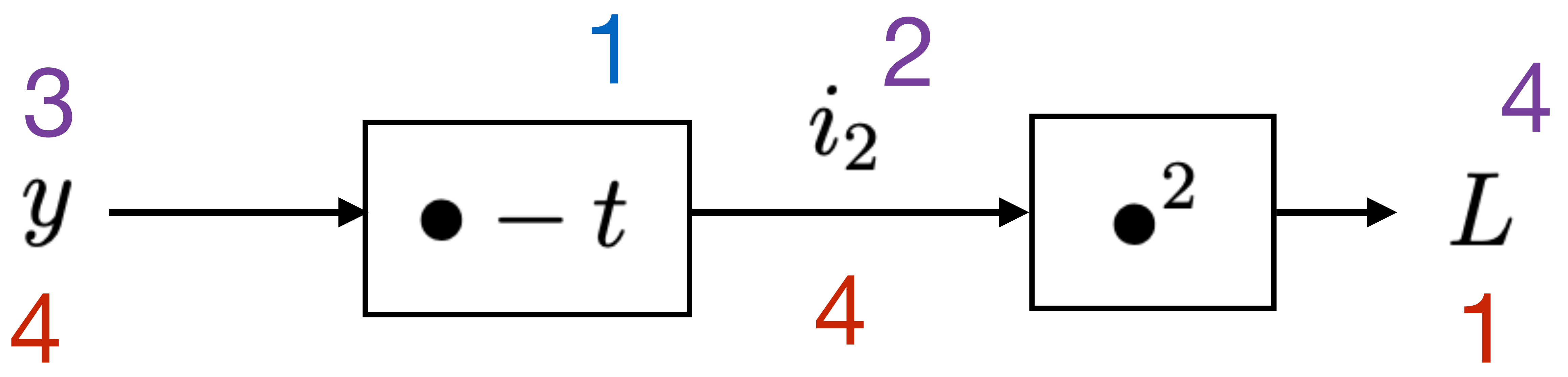




2-Layer **Neural** Network — 1 hidden, 1 input/output 19.6

Input + Initial weights /target   Forward pass   Backward pass $= \dfrac{\partial L}{\partial \bullet}$



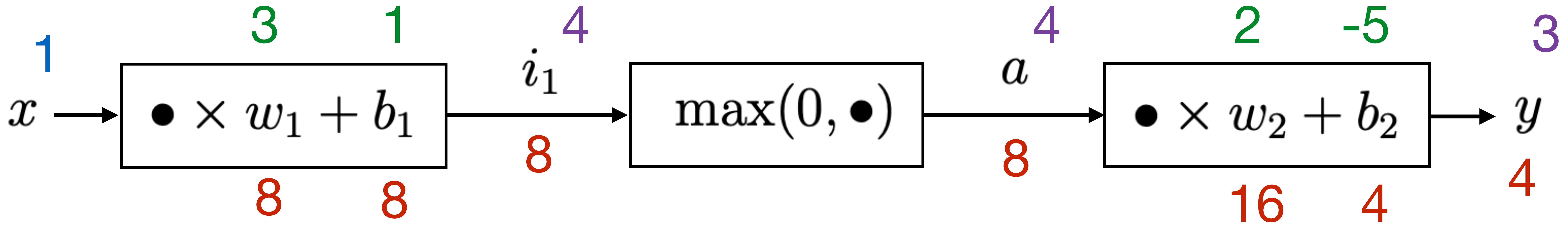


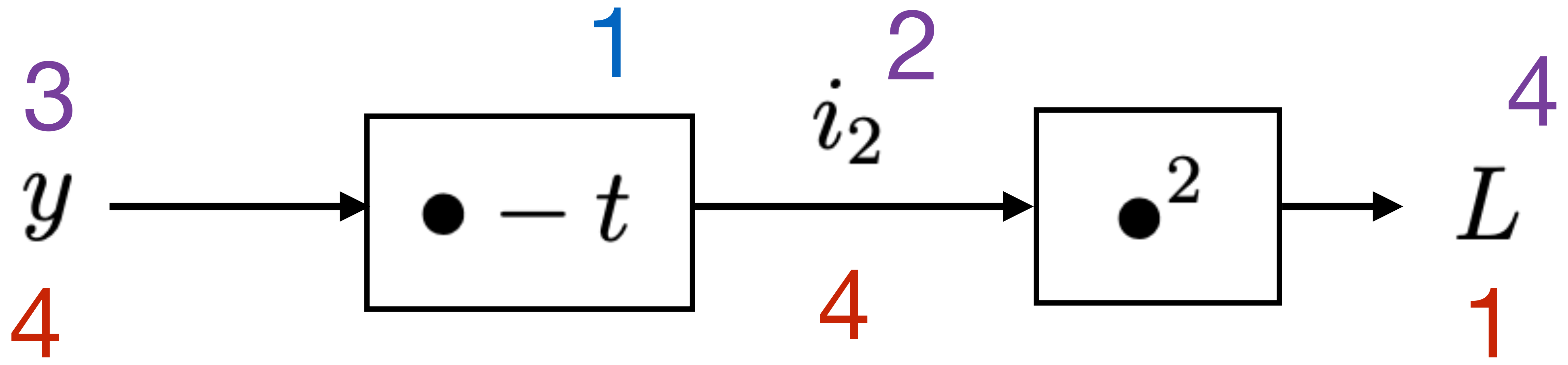
Gradient = $\begin{bmatrix} \dfrac{\partial L}{\partial w_1} \\ \dfrac{\partial L}{\partial b_1} \\ \dfrac{\partial L}{\partial w_2} \\ \dfrac{\partial L}{\partial b_2} \end{bmatrix}$

2-Layer **Neural** Network — 1 hidden, 1 input/output

19.6

Input + Initial weights /target   Forward pass   Backward pass $= \dfrac{\partial L}{\partial \bullet}$



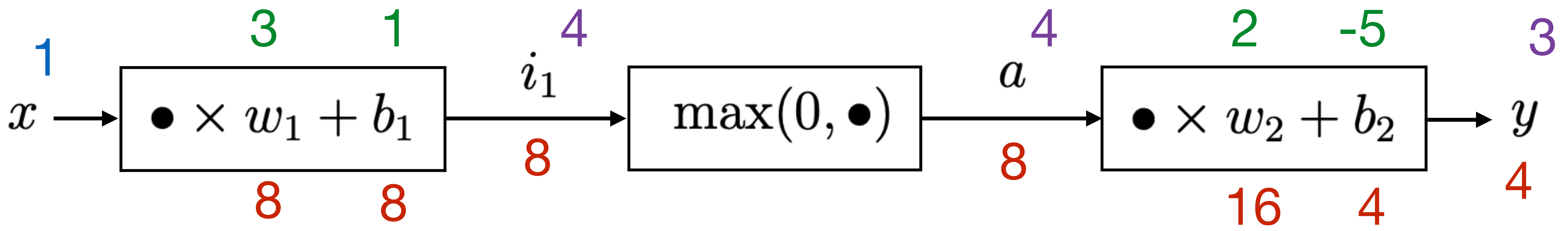


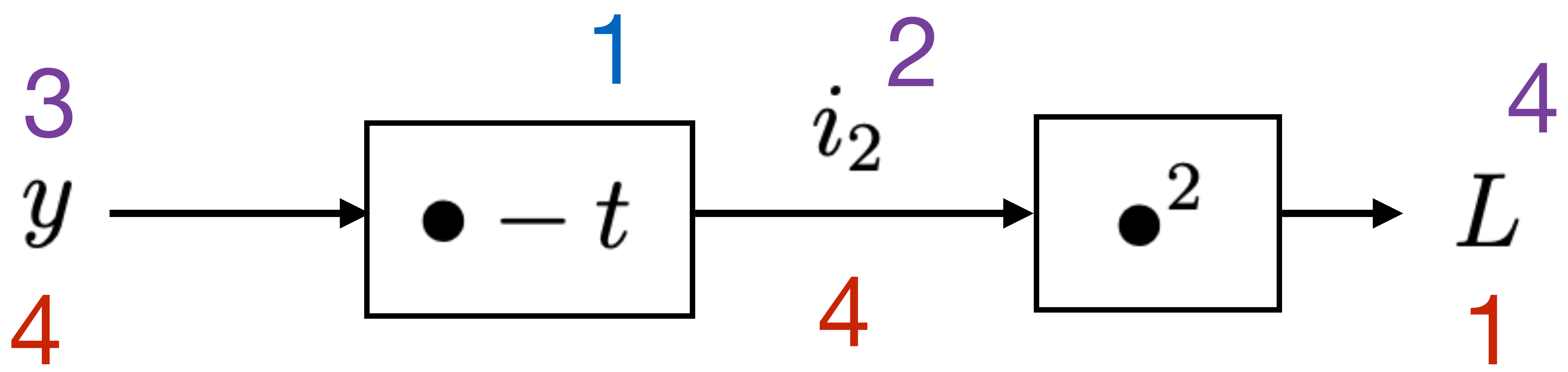
Gradient = $\begin{bmatrix} 8 \\ 8 \\ 16 \\ 4 \end{bmatrix}$

2-Layer **Neural** Network — 1 hidden, 1 input/output

19.6

Input + Initial weights /target   Forward pass   Backward pass $= \dfrac{\partial L}{\partial \bullet}$



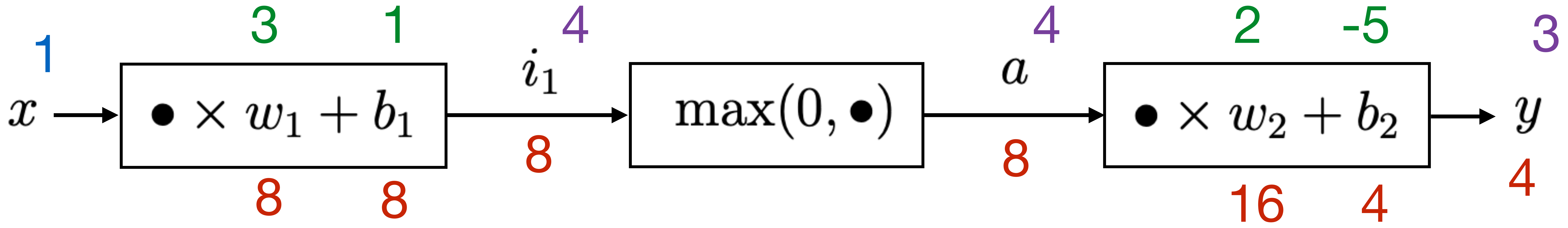


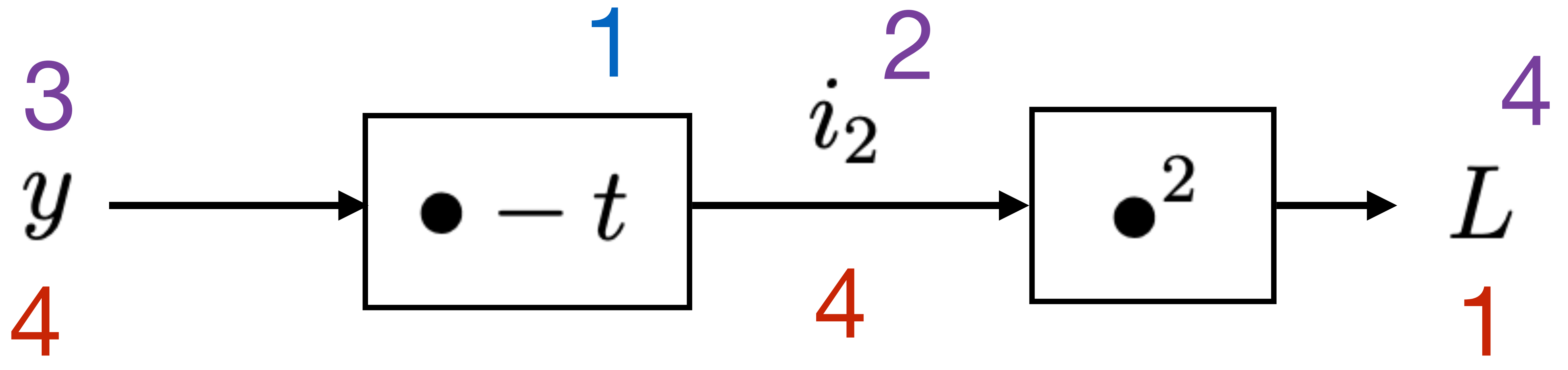
Gradient descent step

$$\begin{bmatrix} 8 \\ 8 \\ 16 \\ 4 \end{bmatrix}$$

2-Layer **Neural** Network — 1 hidden, 1 input/output 19.6

Input + Initial weights /target   Forward pass   Backward pass $= \dfrac{\partial L}{\partial \bullet}$





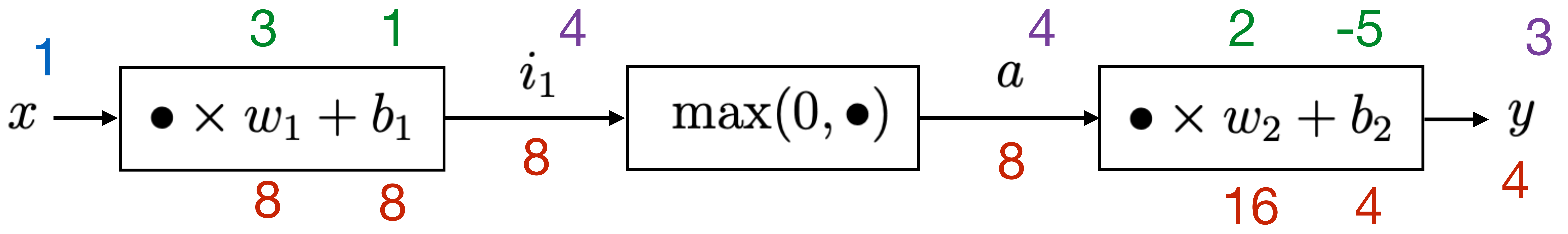
Gradient descent step

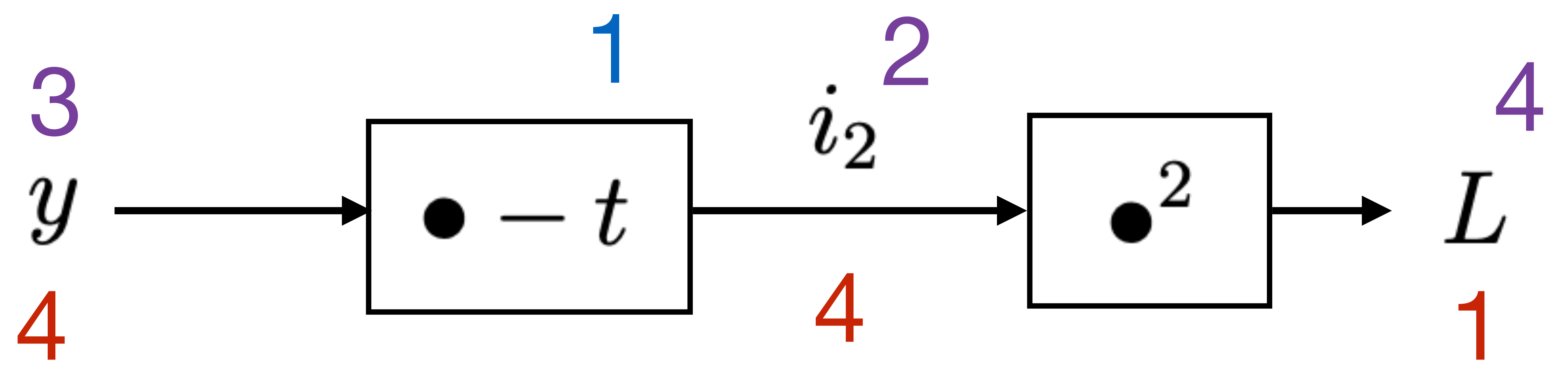
$$\begin{bmatrix} w_1 \\ b_1 \\ w_2 \\ b_2 \end{bmatrix} \rightarrow \begin{bmatrix} 3 \\ 1 \\ 2 \\ -5 \end{bmatrix} - \alpha \begin{bmatrix} 8 \\ 8 \\ 16 \\ 4 \end{bmatrix}$$

2-Layer **Neural** Network — 1 hidden, 1 input/output

19.6

Input + /target Initial weights   Forward pass   Backward pass $= \dfrac{\partial L}{\partial \bullet}$



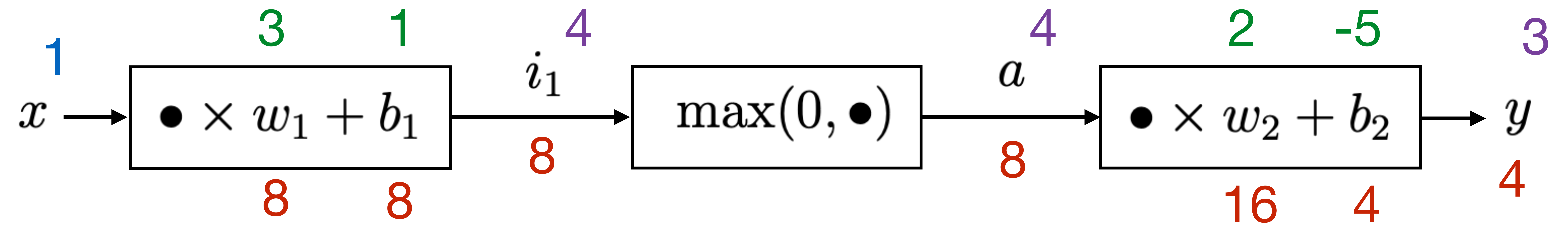


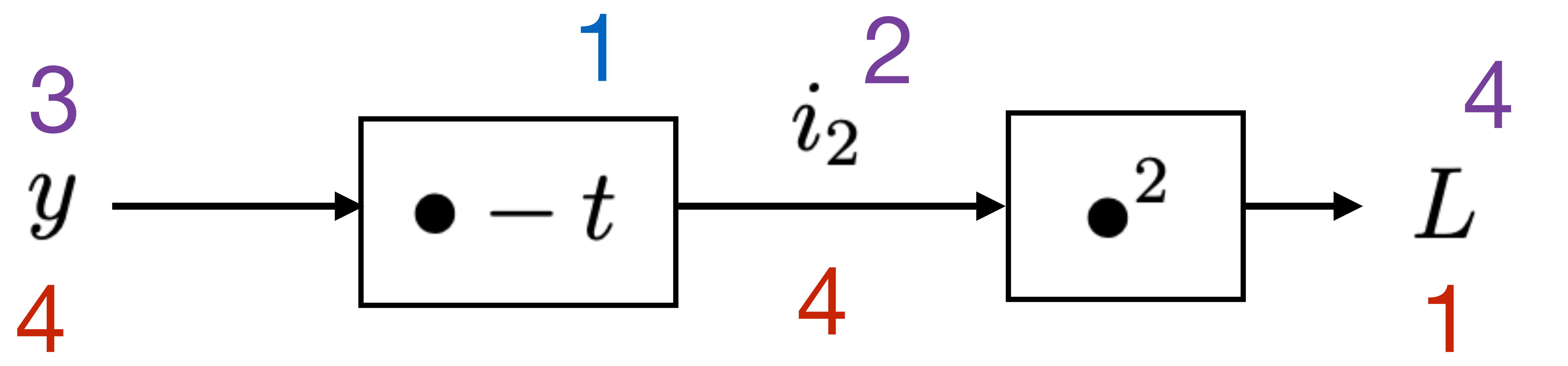
Gradient descent step

$$\begin{bmatrix} w_1 \\ b_1 \\ w_2 \\ b_2 \end{bmatrix} \rightarrow \begin{bmatrix} 3 \\ 1 \\ 2 \\ -5 \end{bmatrix} - \overset{1/4}{\alpha} \begin{bmatrix} 8 \\ 8 \\ 16 \\ 4 \end{bmatrix}$$

2-Layer **Neural** Network — 1 hidden, 1 input/output ✏️ 19.6

Input + Initial weights /target   Forward pass   Backward pass $= \dfrac{\partial L}{\partial \bullet}$



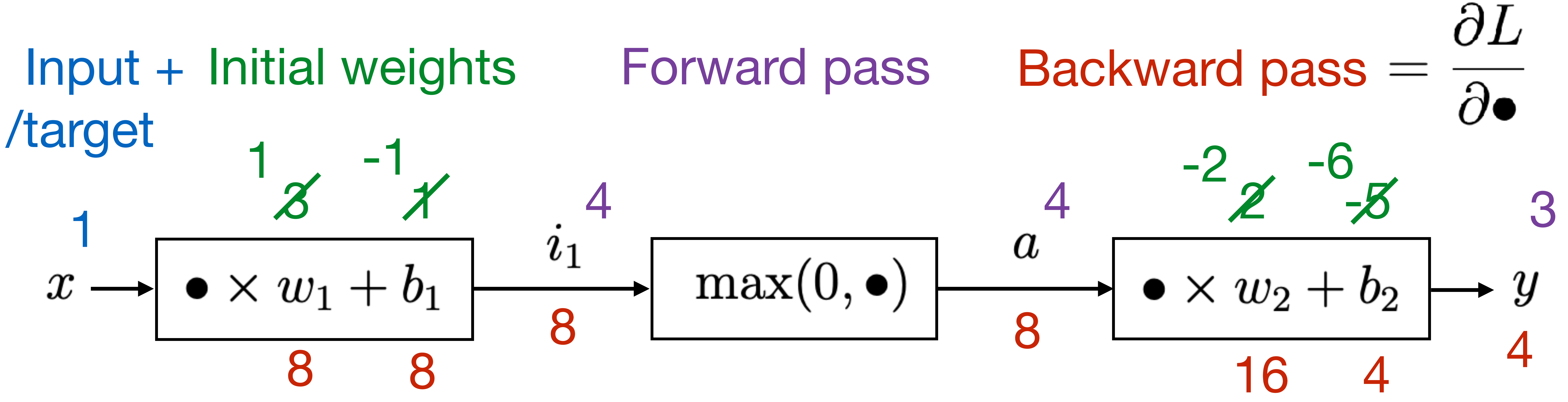


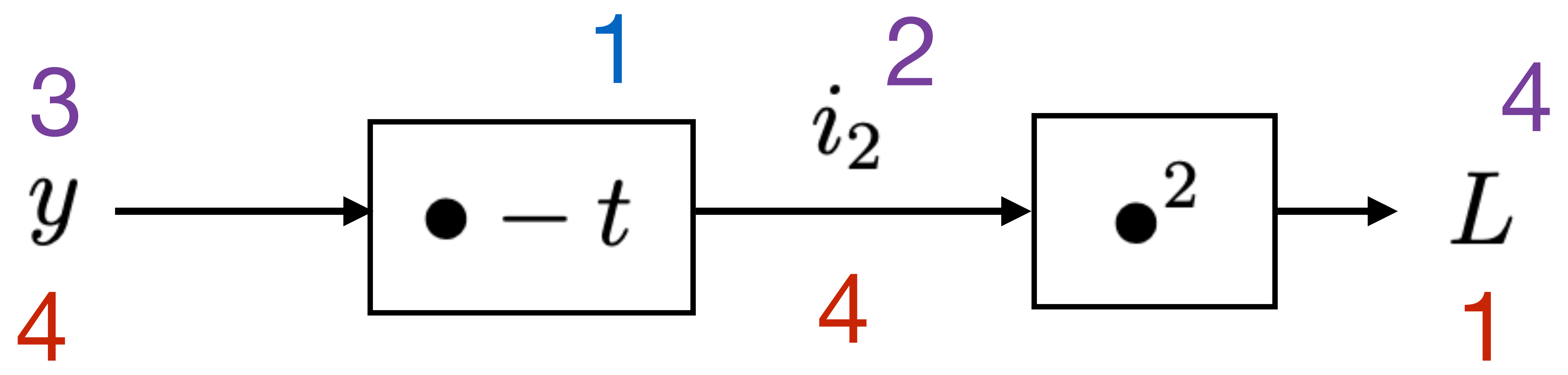
Gradient descent step

$$\begin{bmatrix} w_1 \\ b_1 \\ w_2 \\ b_2 \end{bmatrix} \rightarrow \begin{bmatrix} 3 \\ 1 \\ 2 \\ -5 \end{bmatrix} - \overset{1/4}{\alpha} \begin{bmatrix} 8 \\ 8 \\ 16 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ -2 \\ -6 \end{bmatrix}$$

2-Layer **Neural** Network — 1 hidden, 1 input/output

19.6





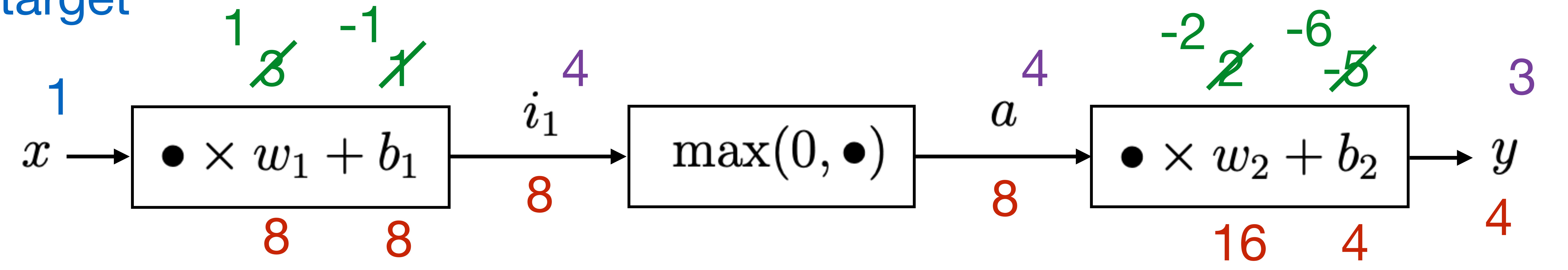
Gradient descent step

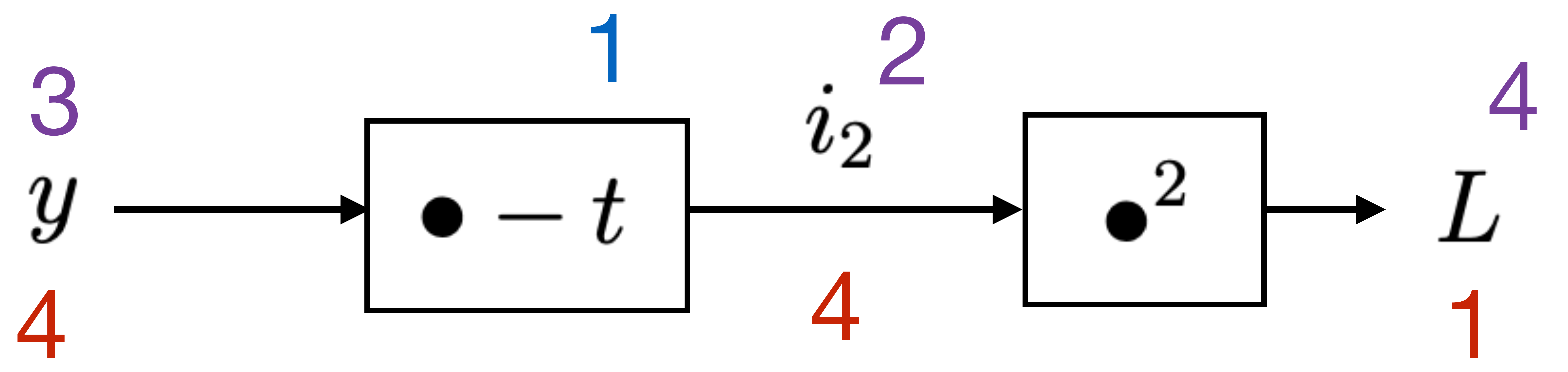
$$\begin{bmatrix} w_1 \\ b_1 \\ w_2 \\ b_2 \end{bmatrix} \rightarrow \begin{bmatrix} 3 \\ 1 \\ 2 \\ -5 \end{bmatrix} - \overset{1/4}{\alpha} \begin{bmatrix} 8 \\ 8 \\ 16 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ -2 \\ -6 \end{bmatrix}$$

+ update weights

2-Layer **Neural** Network — 1 hidden, 1 input/output 19.6

Input + /target   Initial weights   Forward pass   Backward pass $= \dfrac{\partial L}{\partial \bullet}$



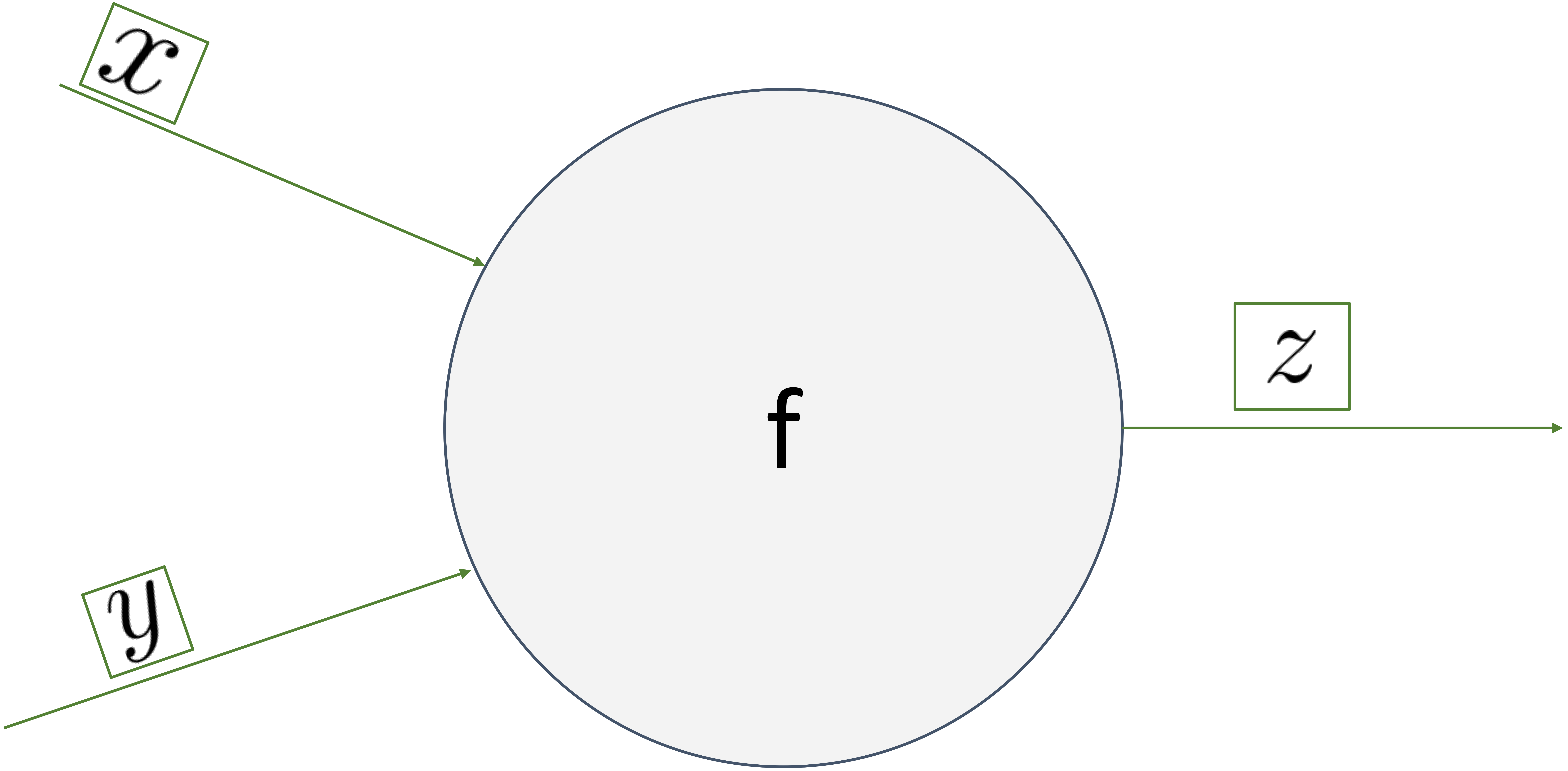


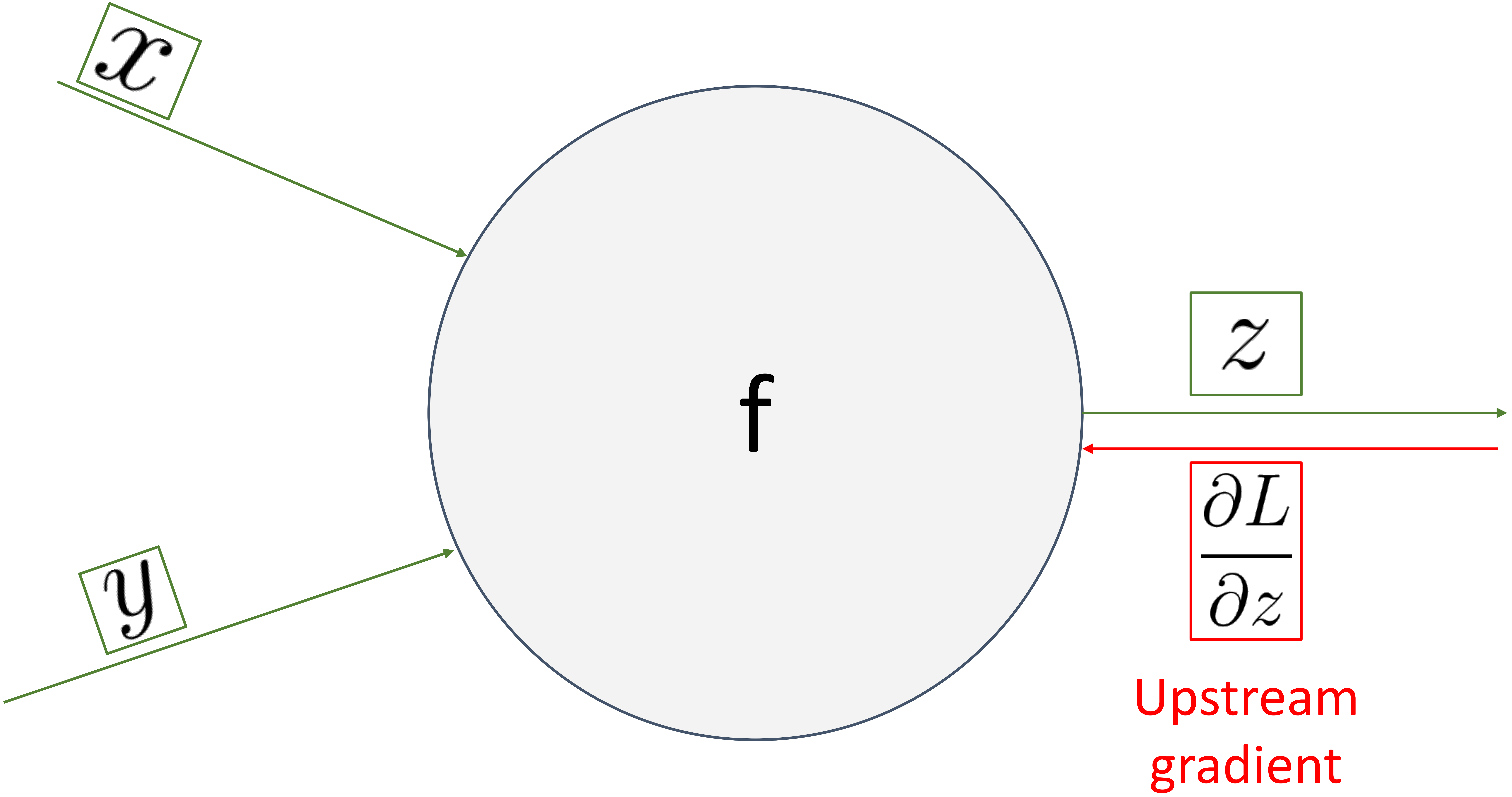
Repeat: +Input/target, Forward, Backward, Update until convergence!

Gradient descent step

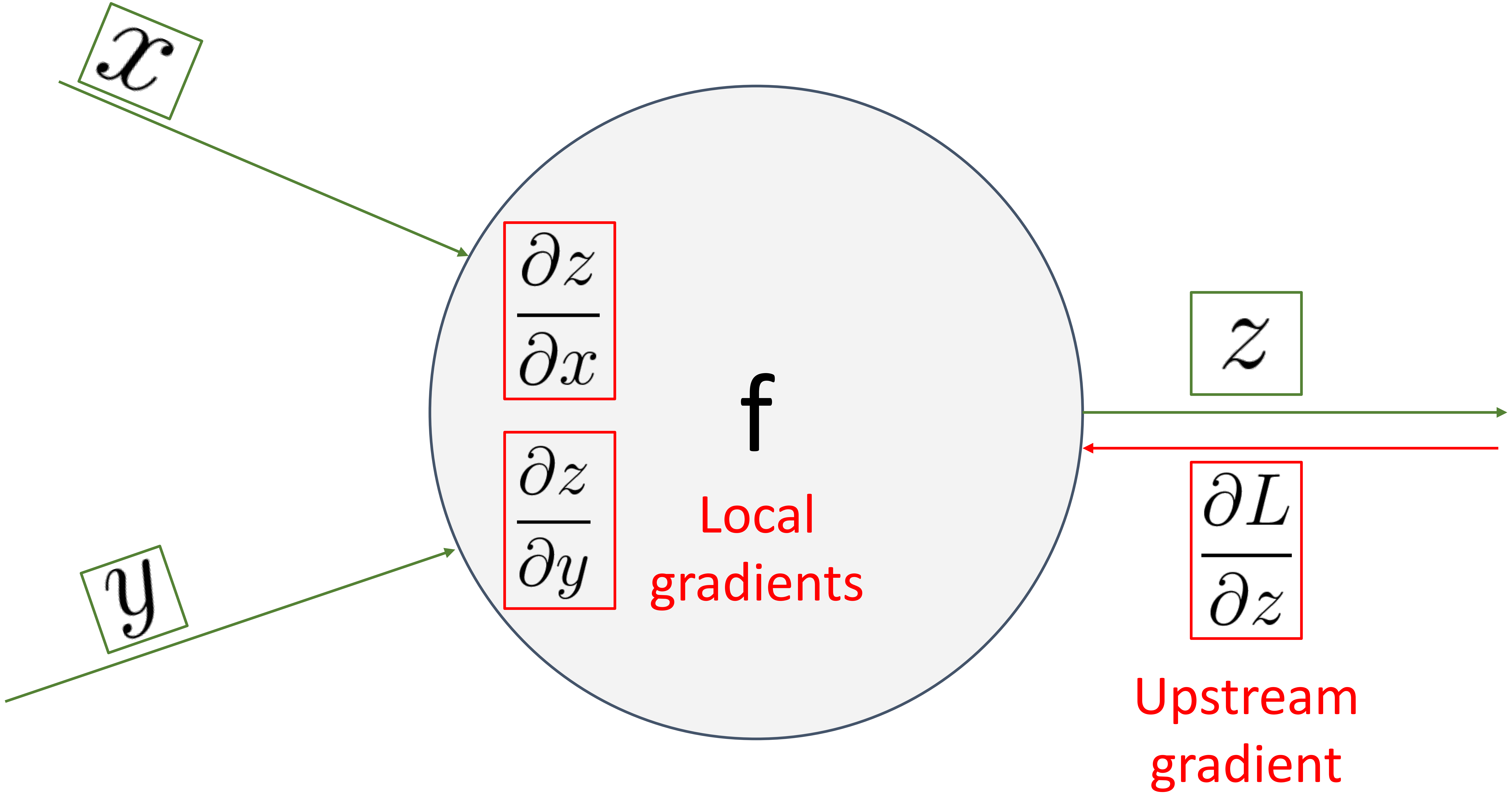
$$\begin{bmatrix} w_1 \\ b_1 \\ w_2 \\ b_2 \end{bmatrix} \rightarrow \begin{bmatrix} 3 \\ 1 \\ 2 \\ -5 \end{bmatrix} - \overset{1/4}{\alpha} \begin{bmatrix} 8 \\ 8 \\ 16 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ -2 \\ -6 \end{bmatrix}$$

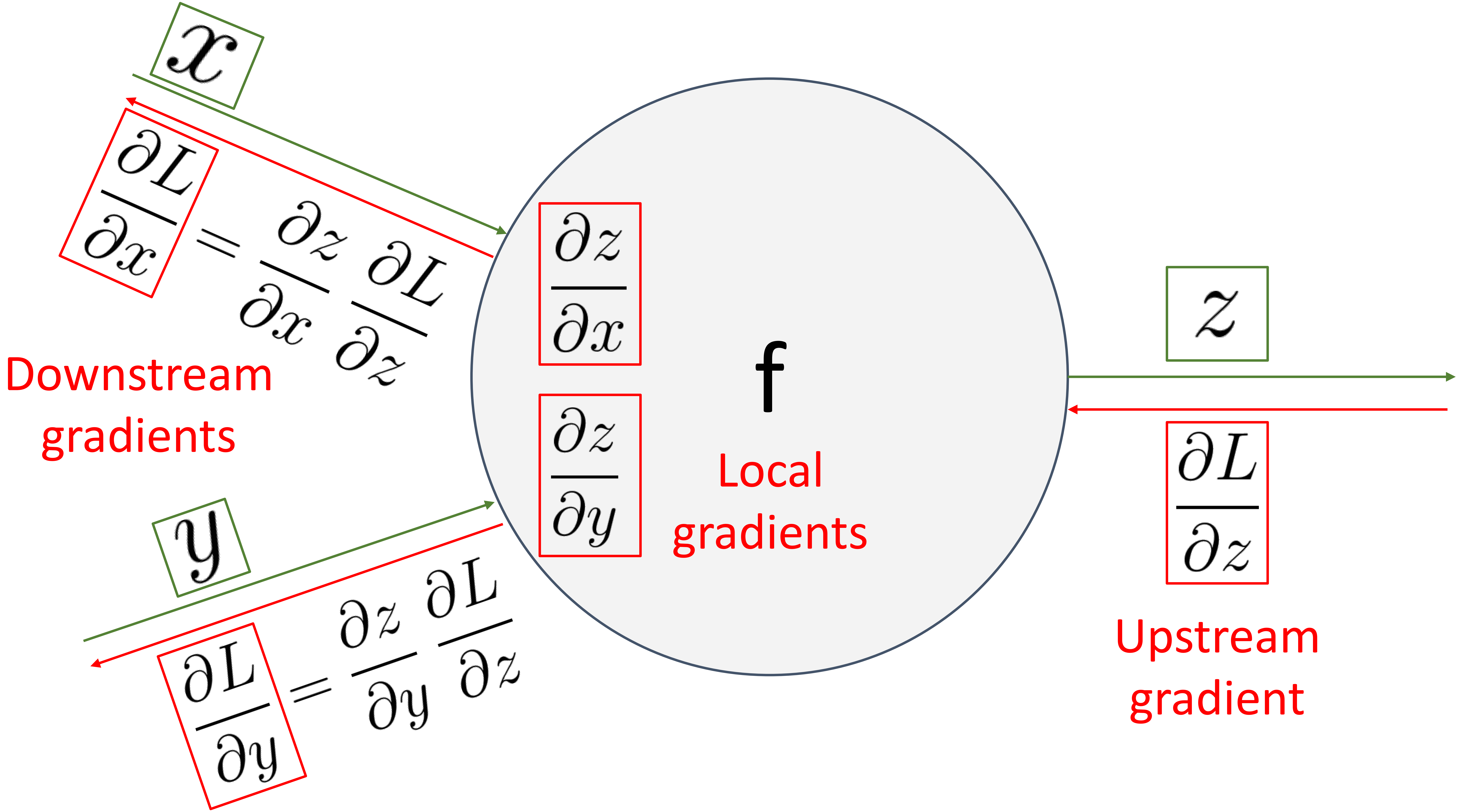
+ update weights


x
y
f
z



x
y
f
z
$\frac{\partial L}{\partial z}$
Upstream gradient

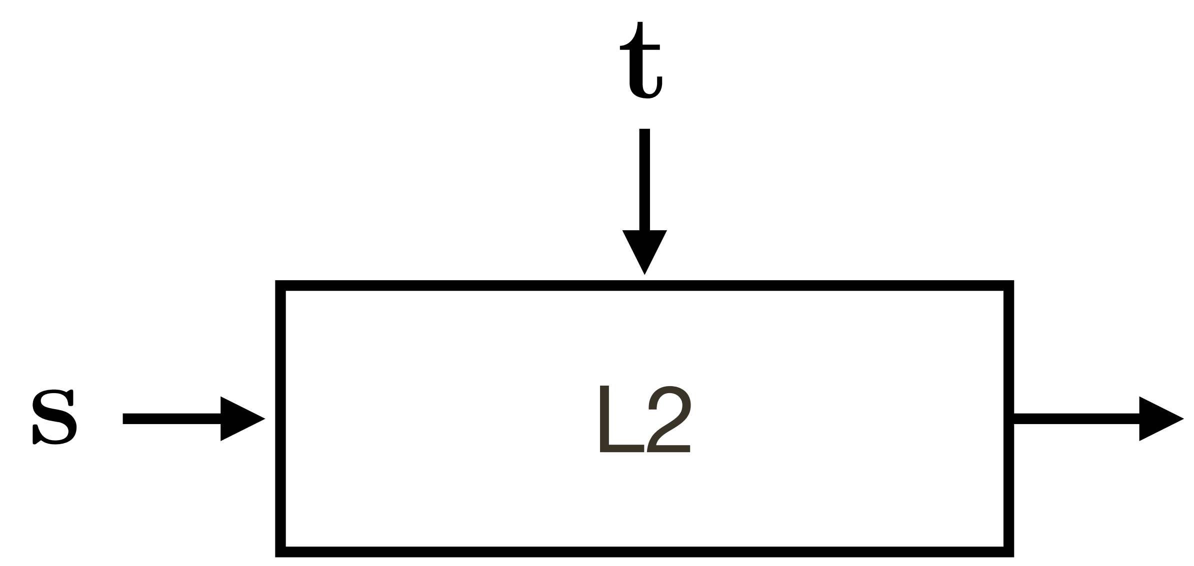

x
y
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
f
Local gradients
z
$\frac{\partial L}{\partial z}$
Upstream gradient



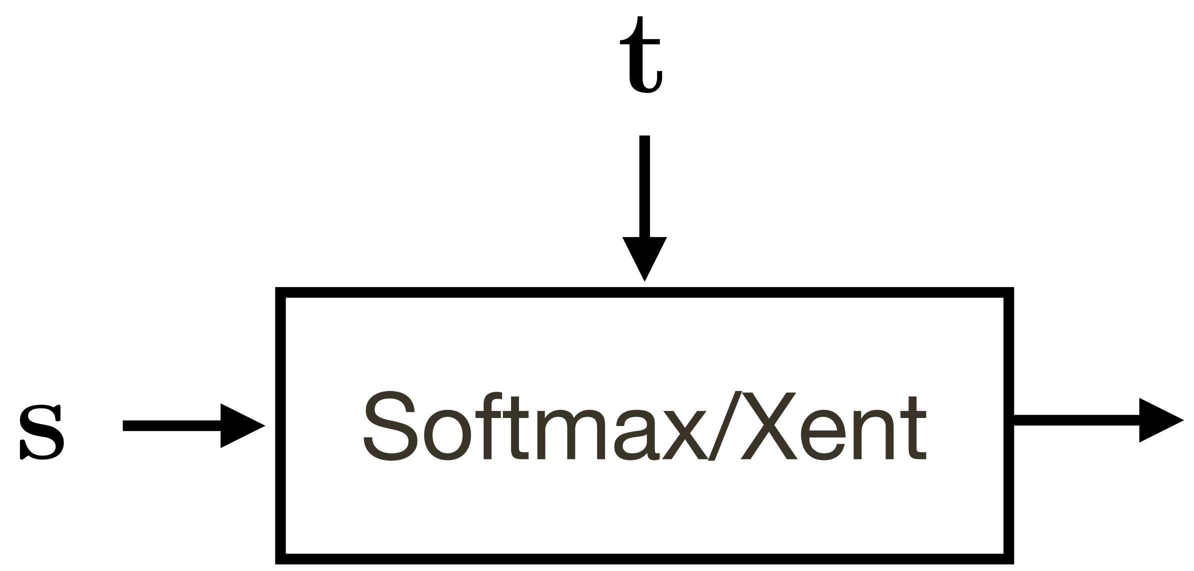
x
$\frac{\partial L}{\partial x} = \frac{\partial z}{\partial x}\frac{\partial L}{\partial z}$
Downstream gradients
y
$\frac{\partial L}{\partial y} = \frac{\partial z}{\partial y}\frac{\partial L}{\partial z}$
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
f
Local gradients
z
$\frac{\partial L}{\partial z}$
Upstream gradient

x
$\frac{\partial L}{\partial x} = \frac{\partial z}{\partial x}\frac{\partial L}{\partial z}$
Downstream gradients
y
$\frac{\partial L}{\partial y} = \frac{\partial z}{\partial y}\frac{\partial L}{\partial z}$
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
f
Local gradients
z
$\frac{\partial L}{\partial z}$
Upstream gradient

Backward Pass for Some **Common Layers**



$$e = \frac{1}{2}|\mathbf{s} - \mathbf{t}|^2$$



$$e = -\log \frac{e^{s_y}}{\sum_i e^{s_i}}$$

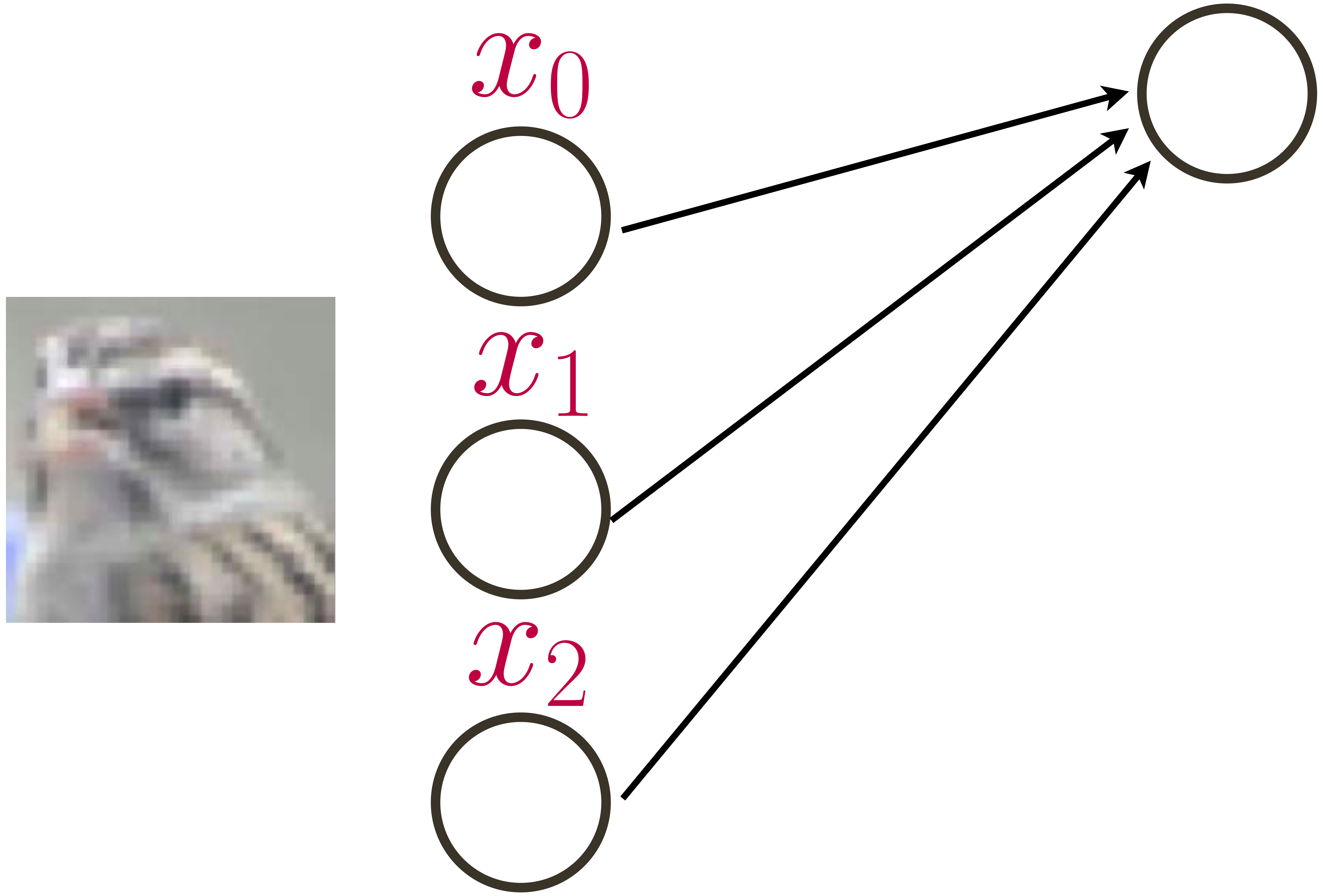
[You will do this for Assignment 6]

$$\frac{\partial e}{\partial \mathbf{s}} = \mathbf{s} - \mathbf{t}$$

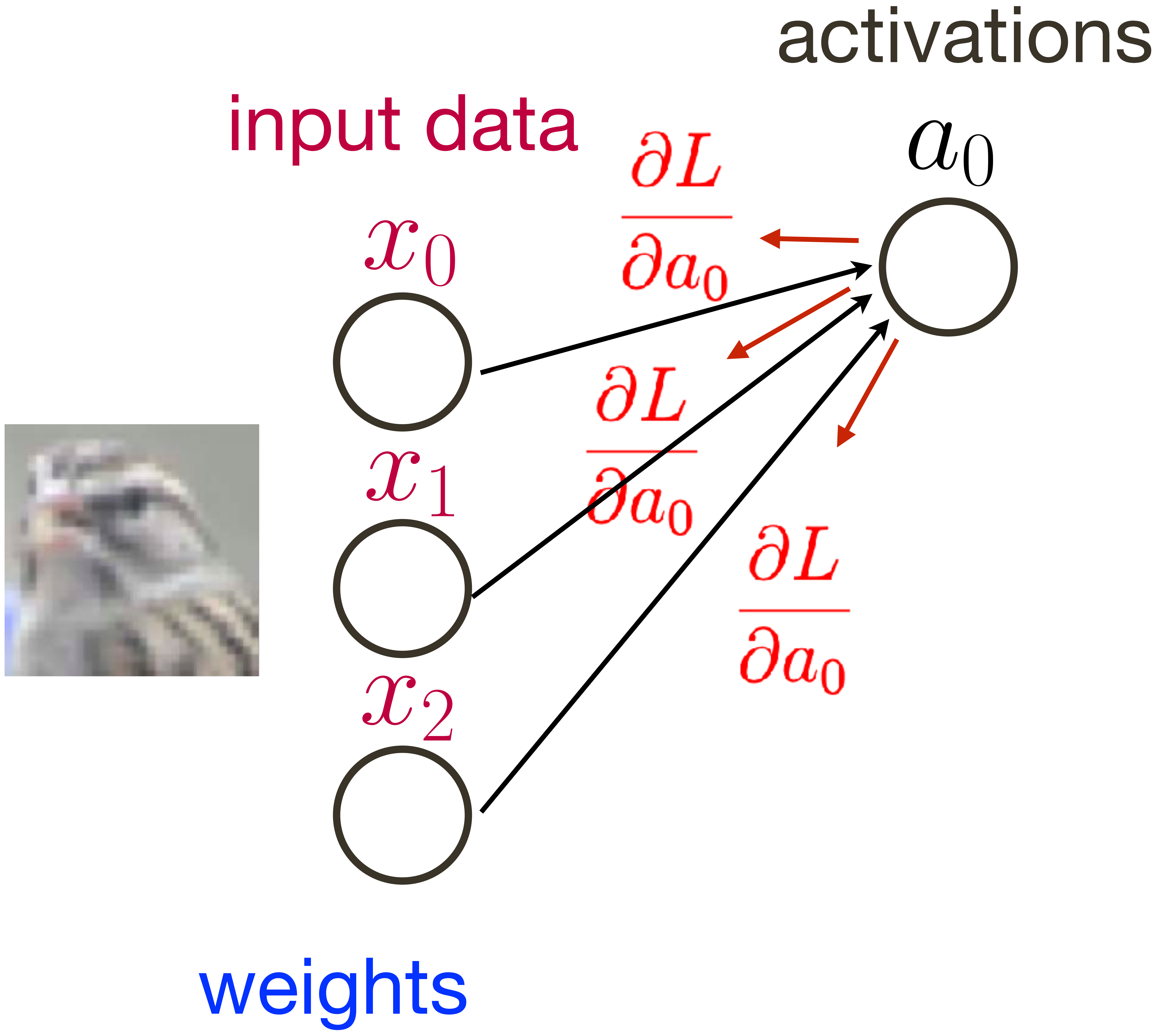
$$\frac{\partial e}{\partial \mathbf{s}} = \sigma(\mathbf{s}) - \mathbf{t}$$

2-Layer **Neural** Network

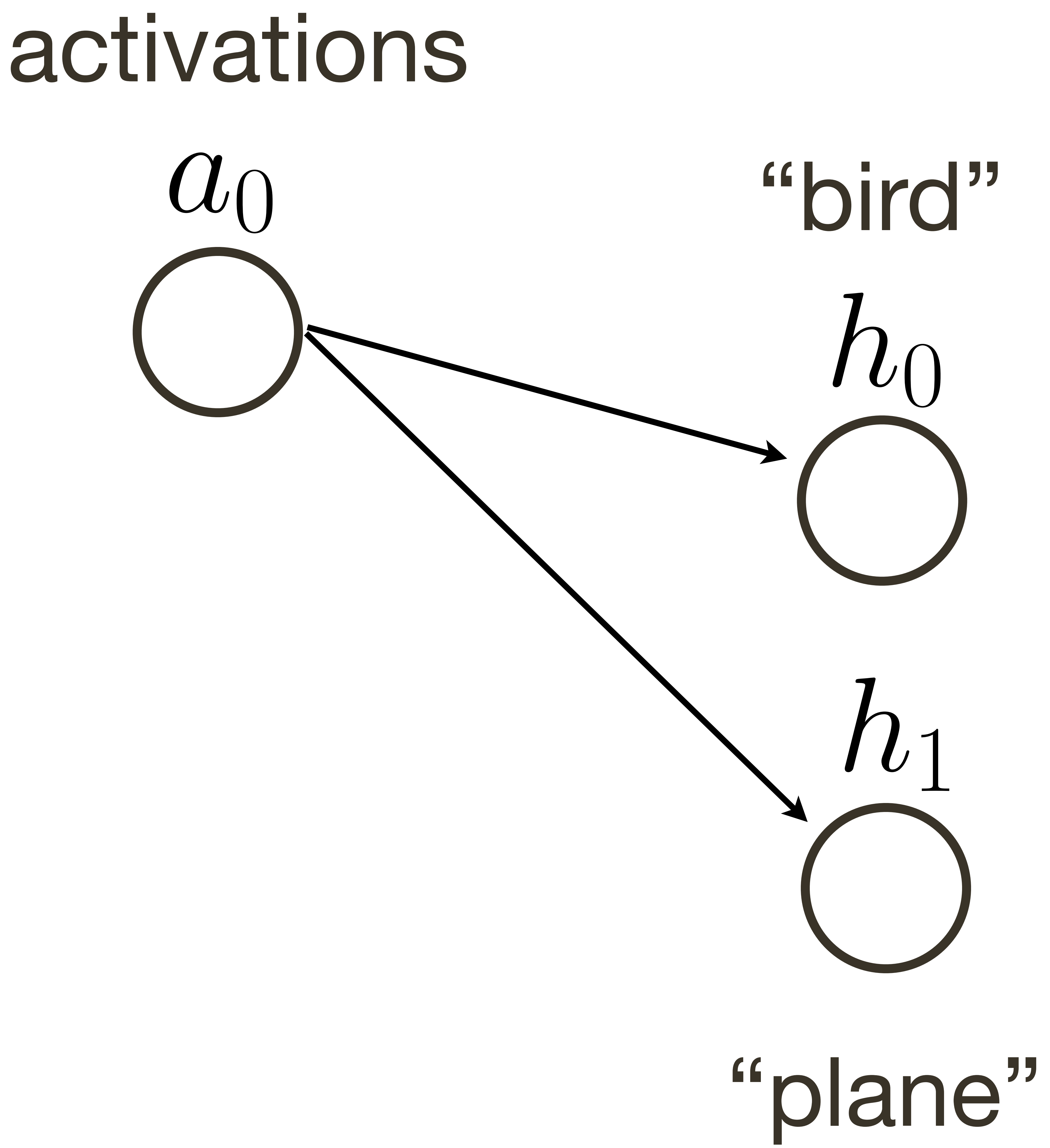
2-Layer **Neural** Network — multiple inputs



2-Layer **Neural** Network — multiple inputs

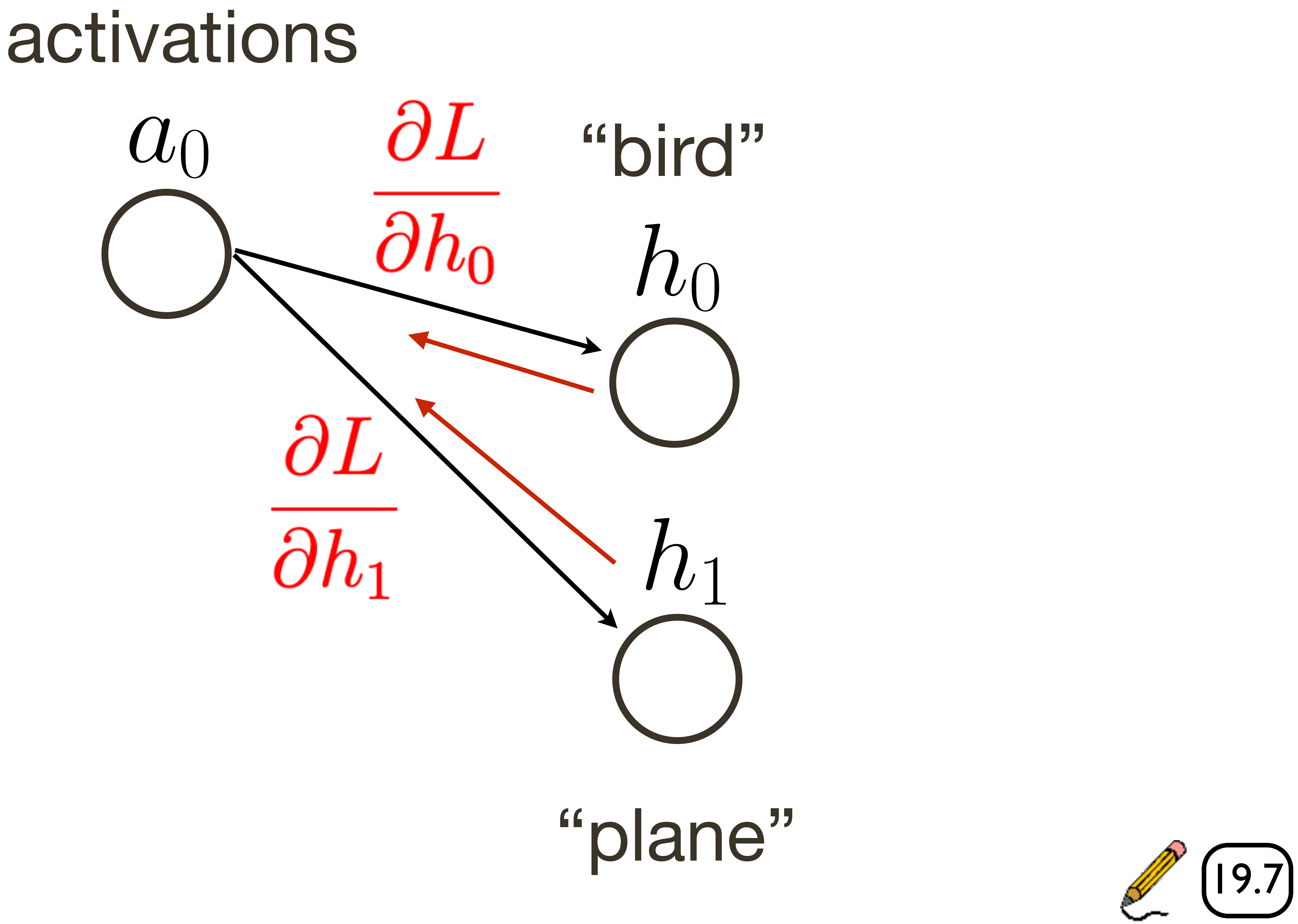


2-Layer **Neural** Network — multiple outputs



19.7

2-Layer **Neural** Network — multiple outputs

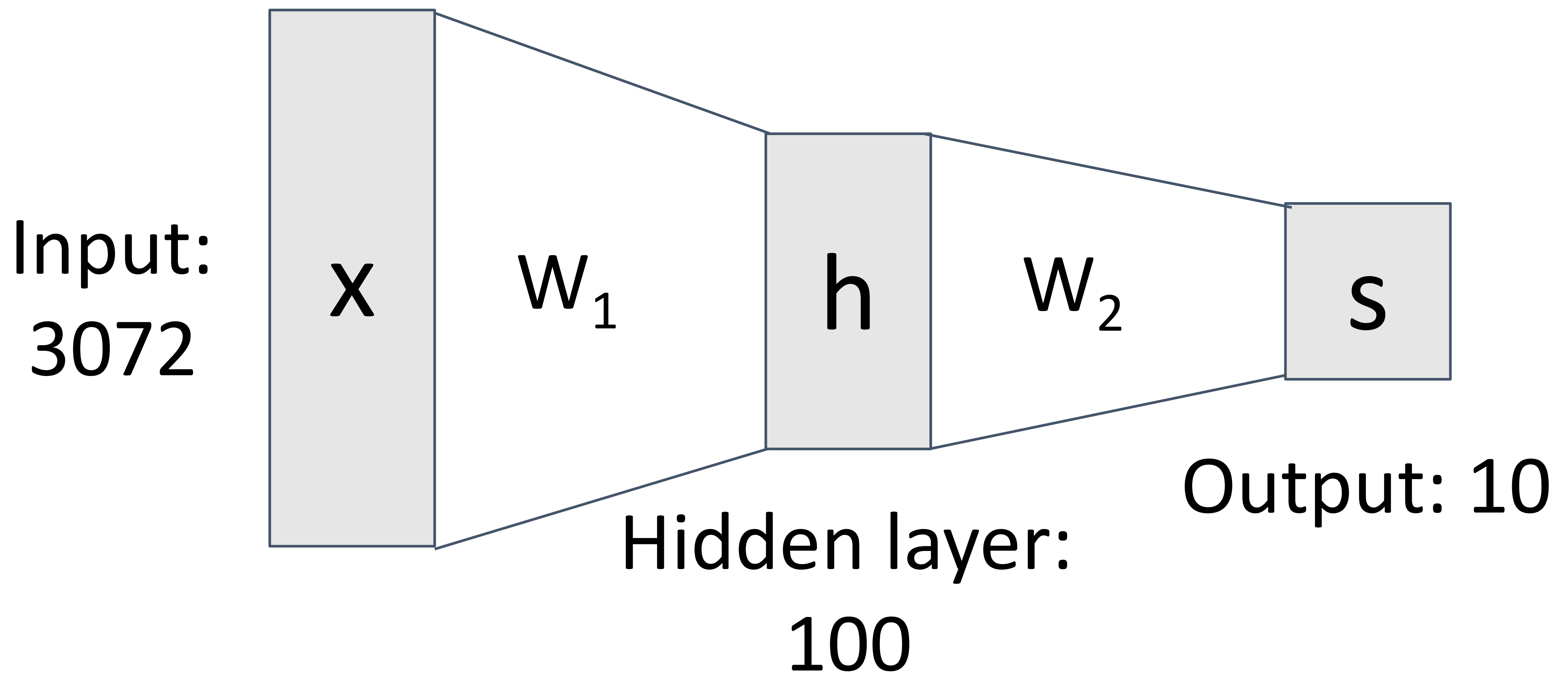


Neural Networks

(**Before**) Linear score function:

(**Now**) 2-layer Neural Network

Linear classifier: One template per class



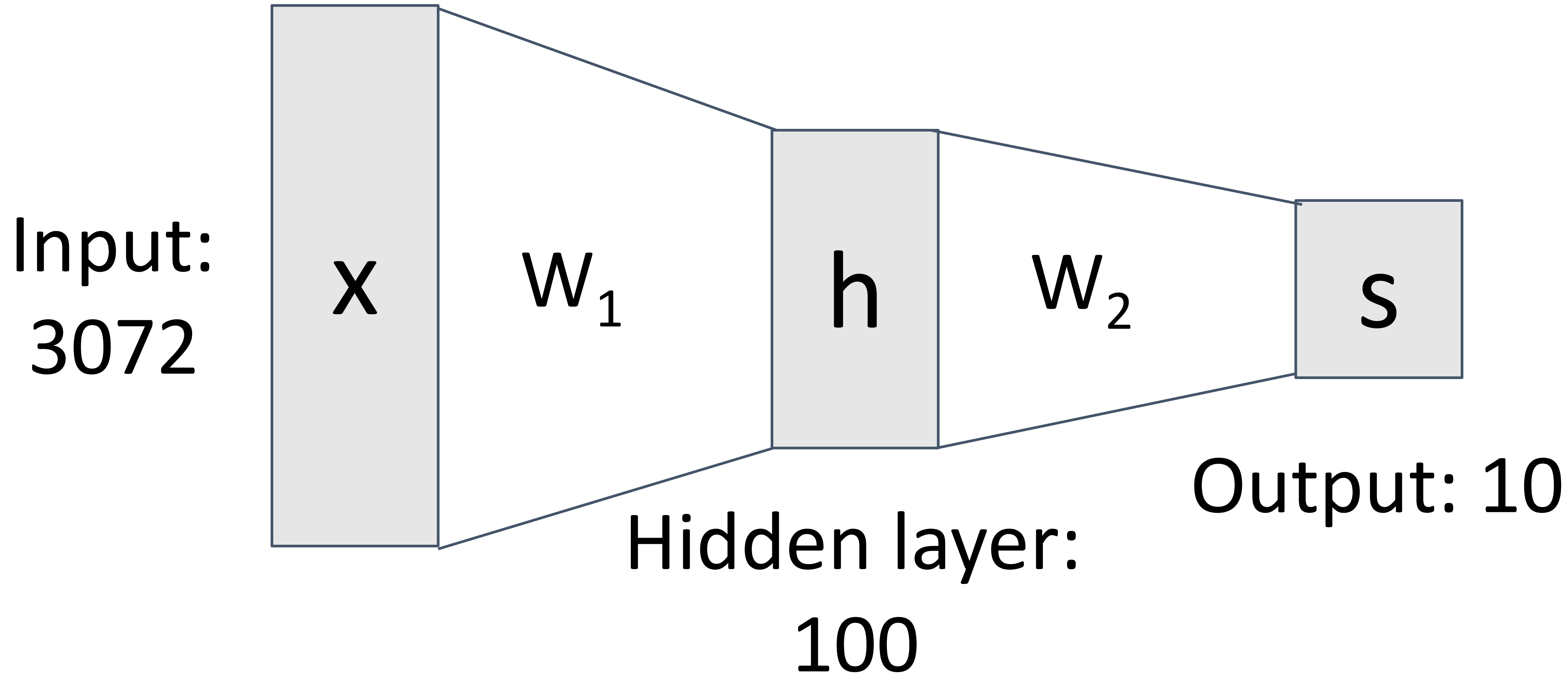
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

Neural net: first layer is bank of templates;
Second layer recombines templates

(**Before**) Linear score function:

(**Now**) 2-layer Neural Network



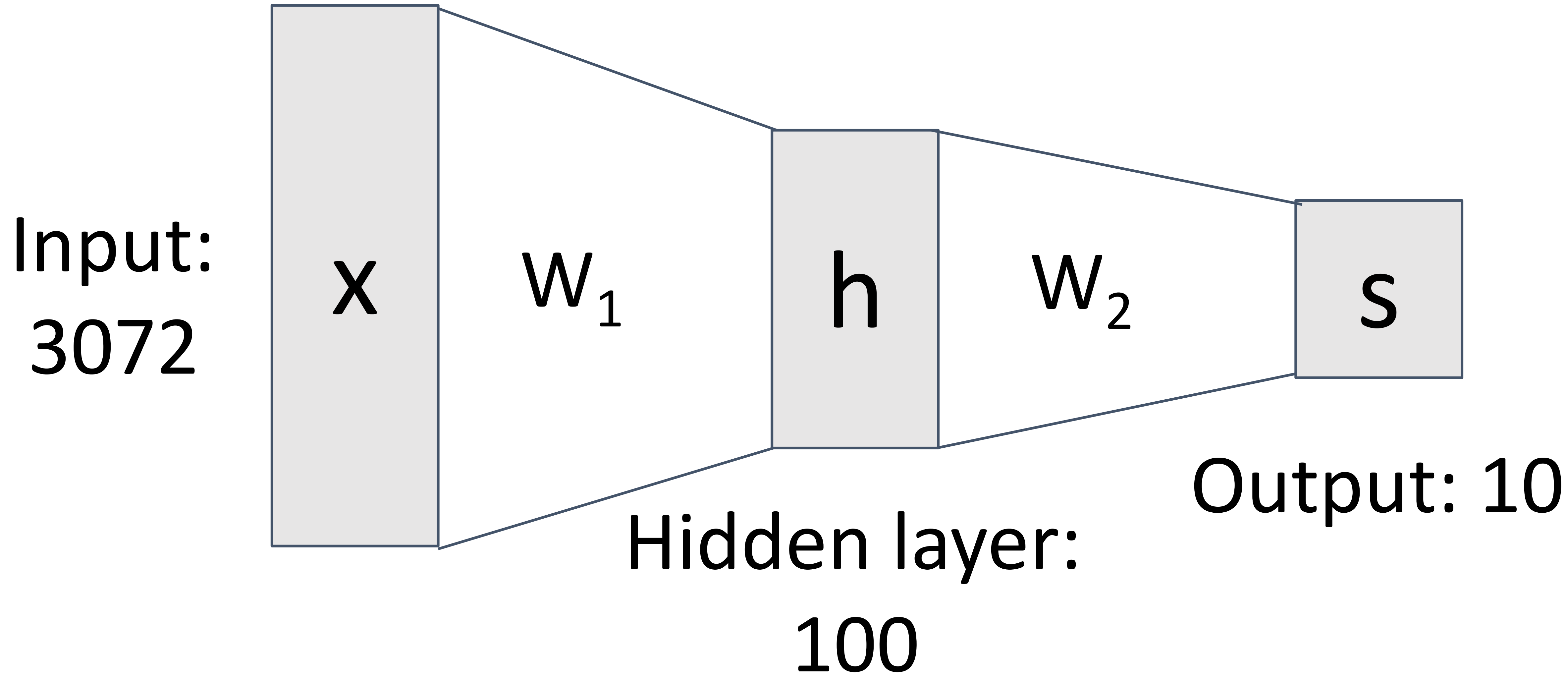
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

Can use different templates to cover multiple modes of a class!

(**Before**) Linear score function:

(**Now**) 2-layer Neural Network



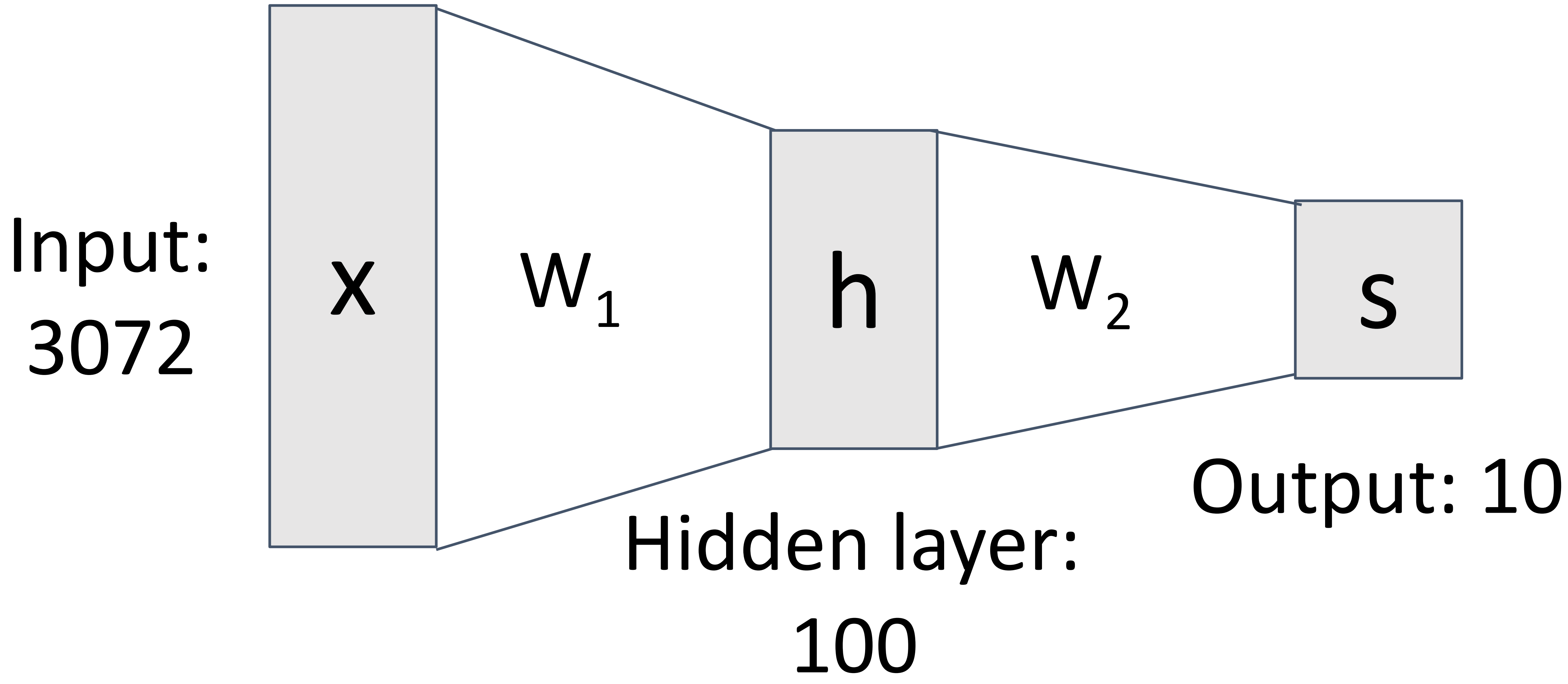
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

"Distributed representation":
Most templates not interpretable!

(**Before**) Linear score function:

(**Now**) 2-layer Neural Network



$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$