Neural Networks

Natural Language Processing — Lecture 3

Kenneth Enevoldsen | 2024

CENTER FOR HUMANITIES COMPUTING

Agenda

- Recap of what we have learned
- A simple **neuron**
  - The perceptron
- Stacking neurons → A **neural network**
- How do we **train** a neural network
- **Overfitting** and **regularization**
- A couple of linguistic examples

Quiz

- https://www.menti.com/al3iseftv5ce

Recap: Representing Meaning

- Central question: **How do we represent meaning computationally?**
  → analyse, classify and generate text
- One-hot vectors
  → (Sparse) co-occurance with reweighting — PPMI, TF-iDF
  → dense vector approaches — Word2Vec, GloVE

Recap: Static Embedding

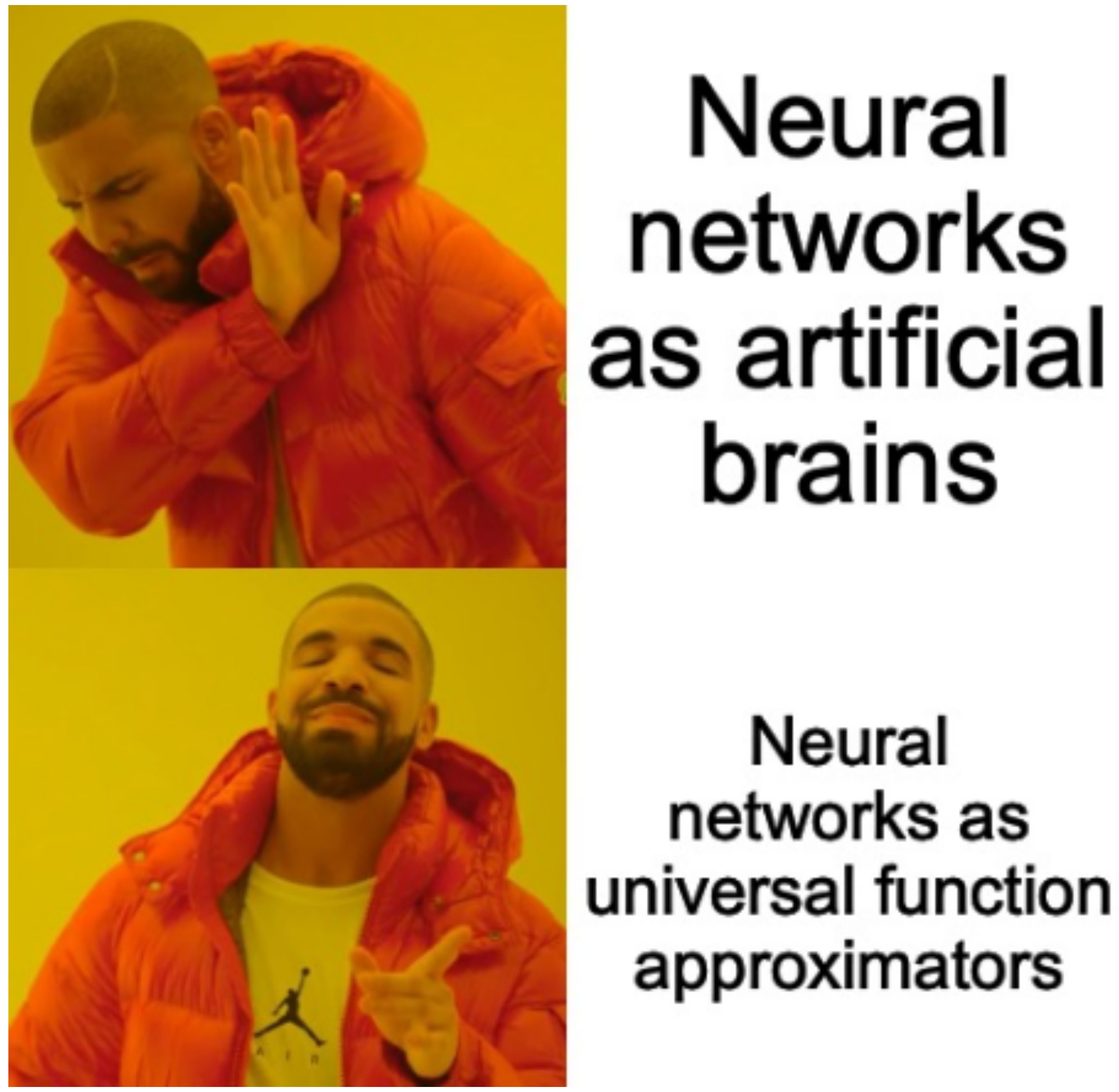
- Central question: **How do we represent meaning computationally?**
  → analyse, classify and generate text
- One-hot vectors
  → (Sparse) co-occurance with reweighting — PPMI, TF-iDF
  → dense vector approaches — Word2Vec, GloVE
- **Resolved**
  Semantic similarity, low dimensional vector representations
- **Remaining Issues**
  Polysemy, compositionality, homonomy

Recap: Incorperating context

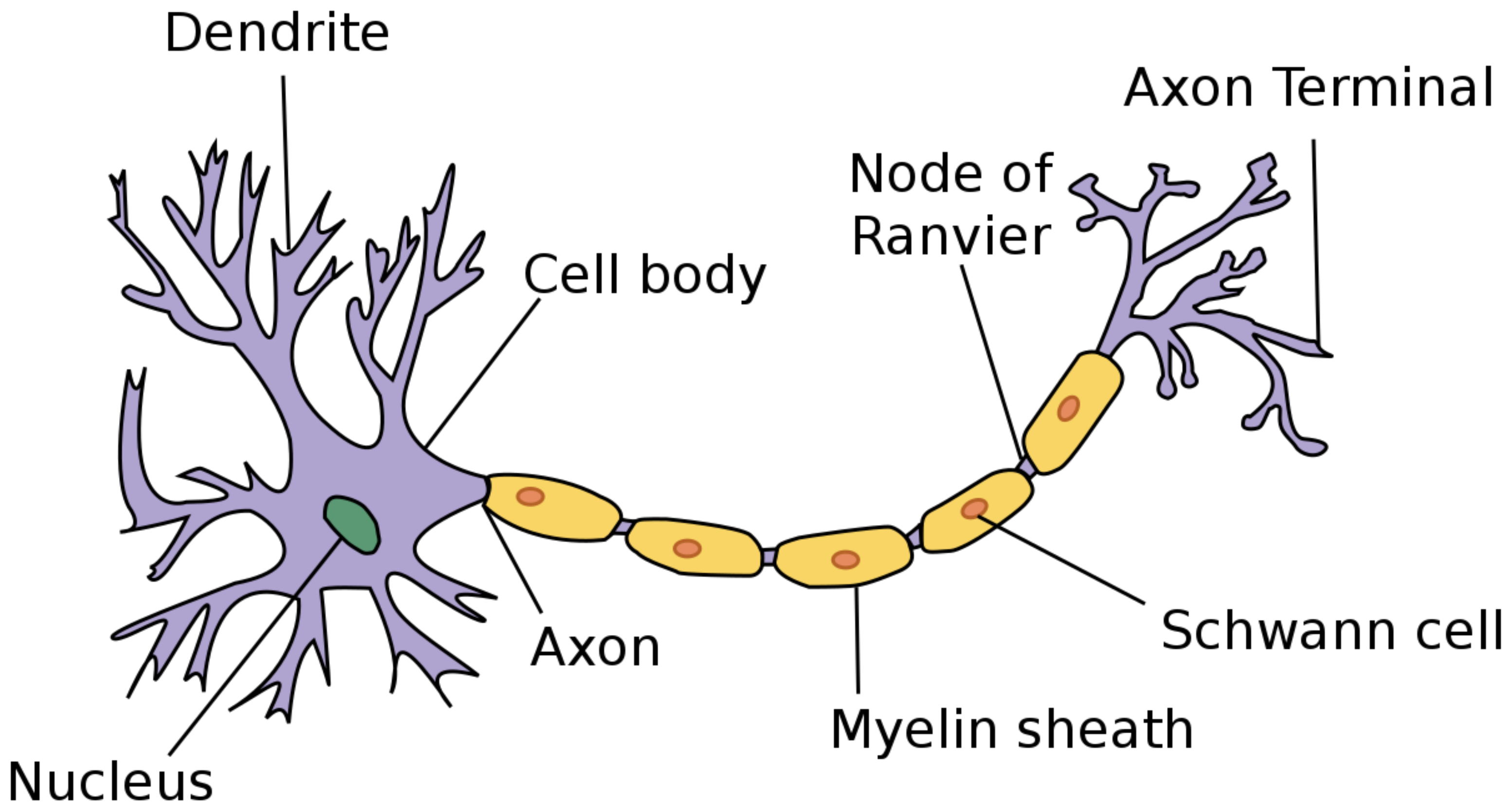
- Two approaches,
  - Recurrent neural networks
  - Attention
- Requires an understanding of Neural Networks

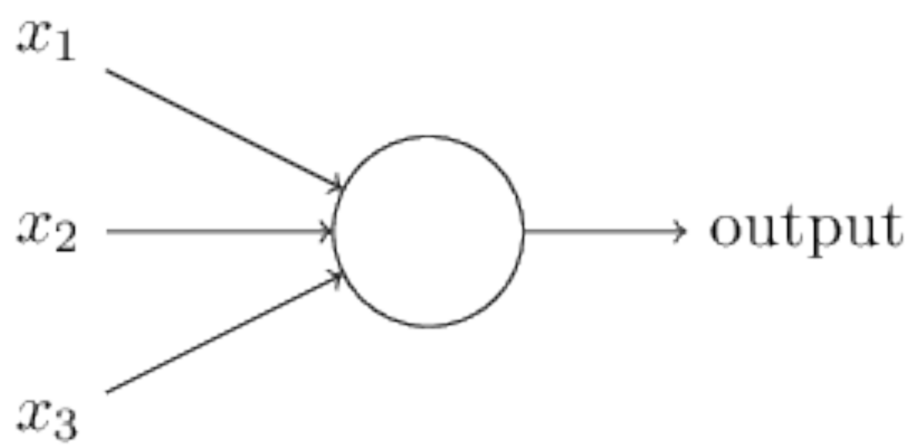
The artificial brain perspective

- Neural networks as *"artificial brains"*
  - Builts poor intuition
- Neural networks as univervals approximators
  - flexible learning systems
  - Want to run fast and efficiently



Biological Neuron vs (Artificial) **Neuron**

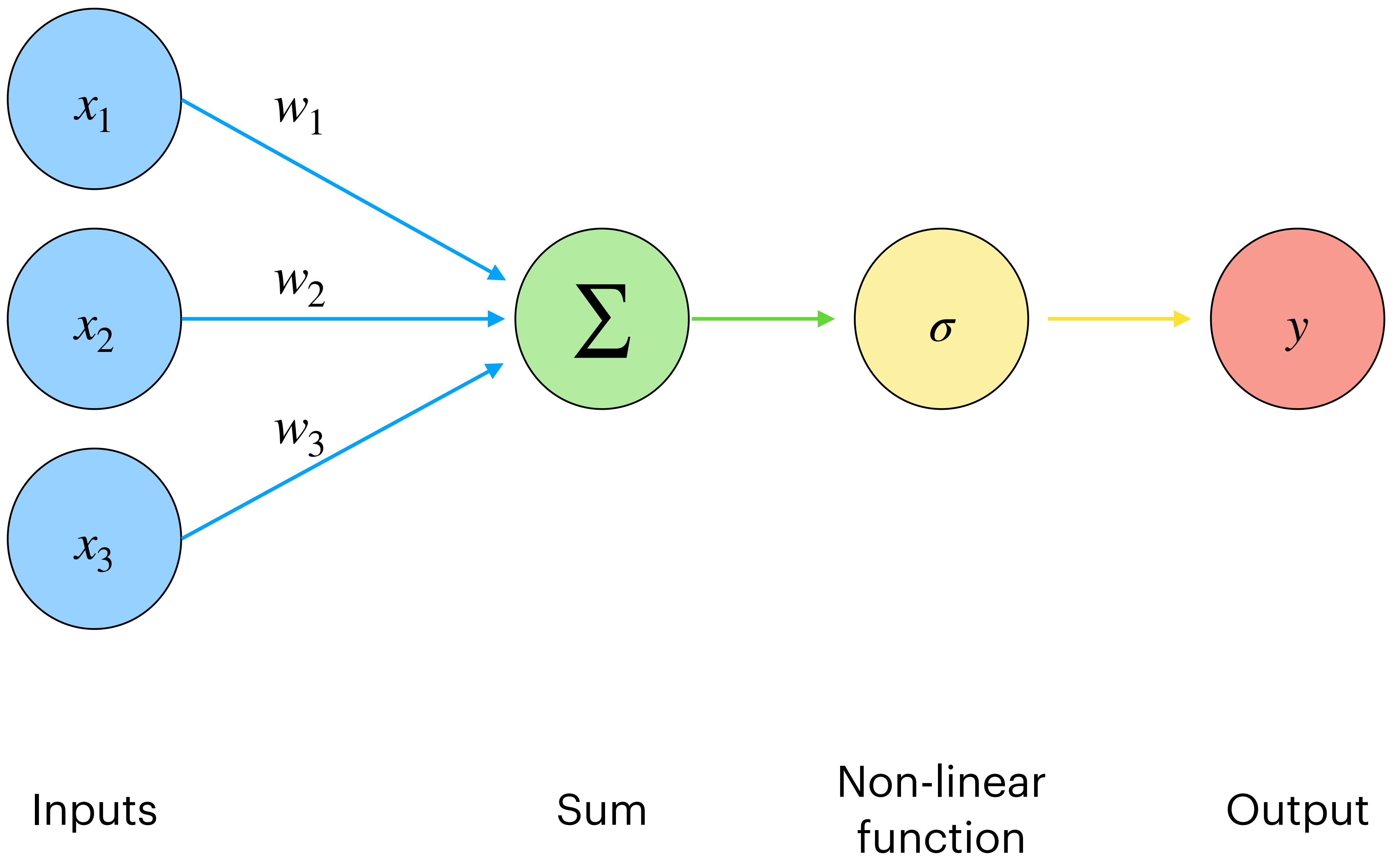




- Recieved inputs from other neurons
- Outputs excitatory or inhibitory signal
- A “computing element”, which transforms a signal

- Recieved inputs from other neurons
- Outputs value
- A “computing element”, which transforms a signal

The Perceptron



Non-linear function

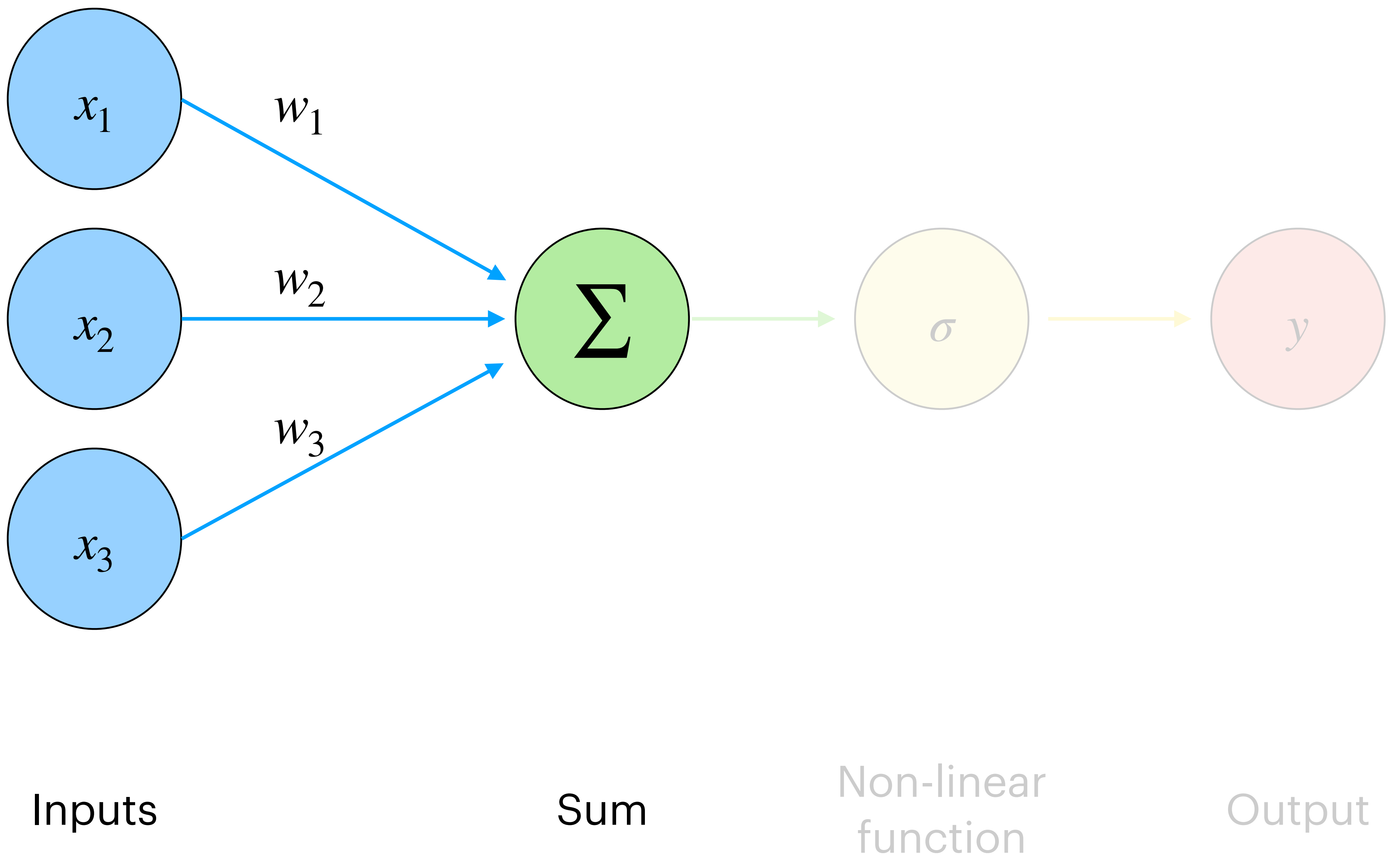
$$y = \sigma\left(\sum_{i=1}^{m} w_i x_i\right)$$

Output

Linear combination

The Perceptron

Question: What could these be?



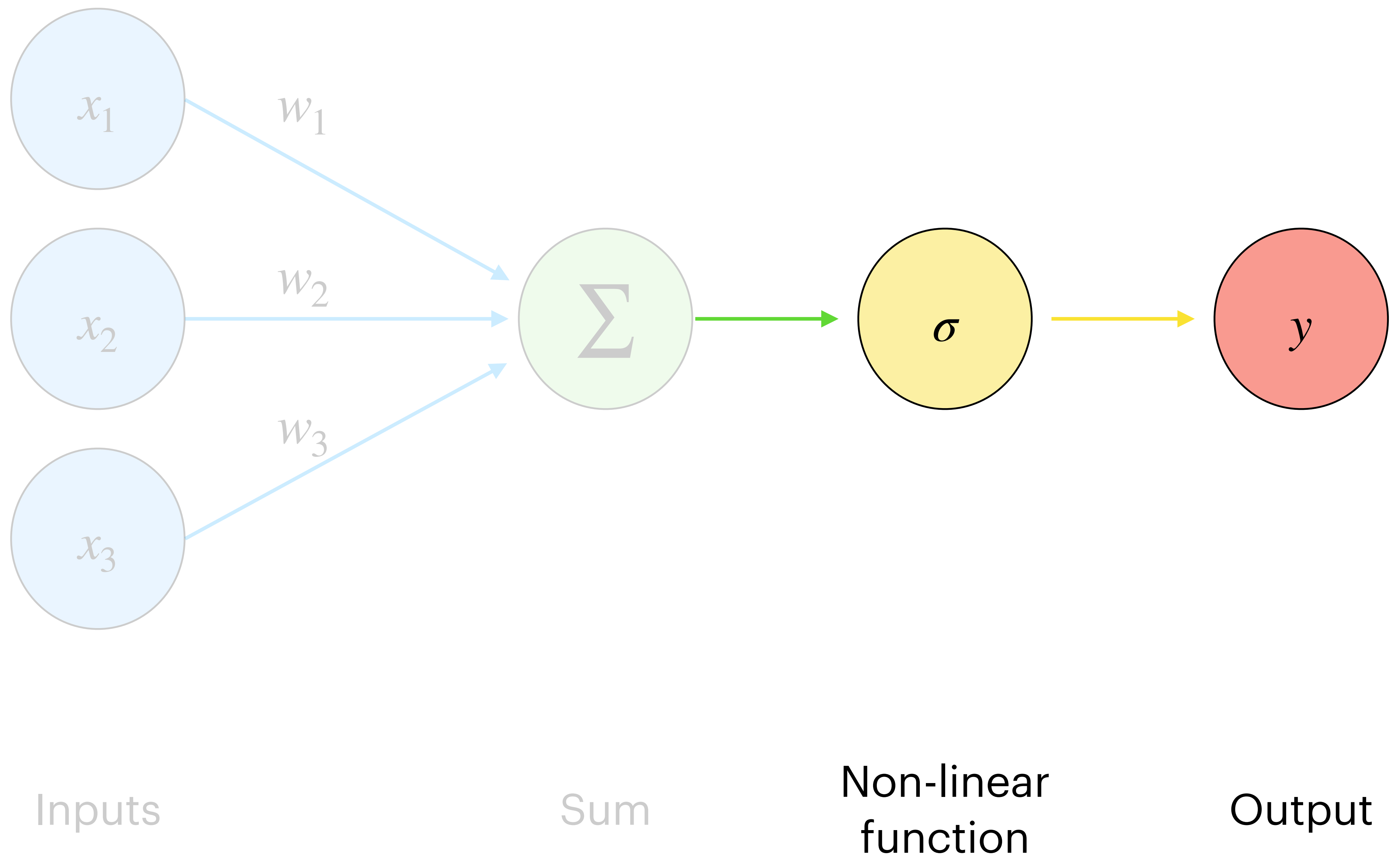
Non-linear function

$$y = \sigma\left(\sum_{i=1}^{m} w_i x_i\right)$$

Output

Linear combination

The Perceptron



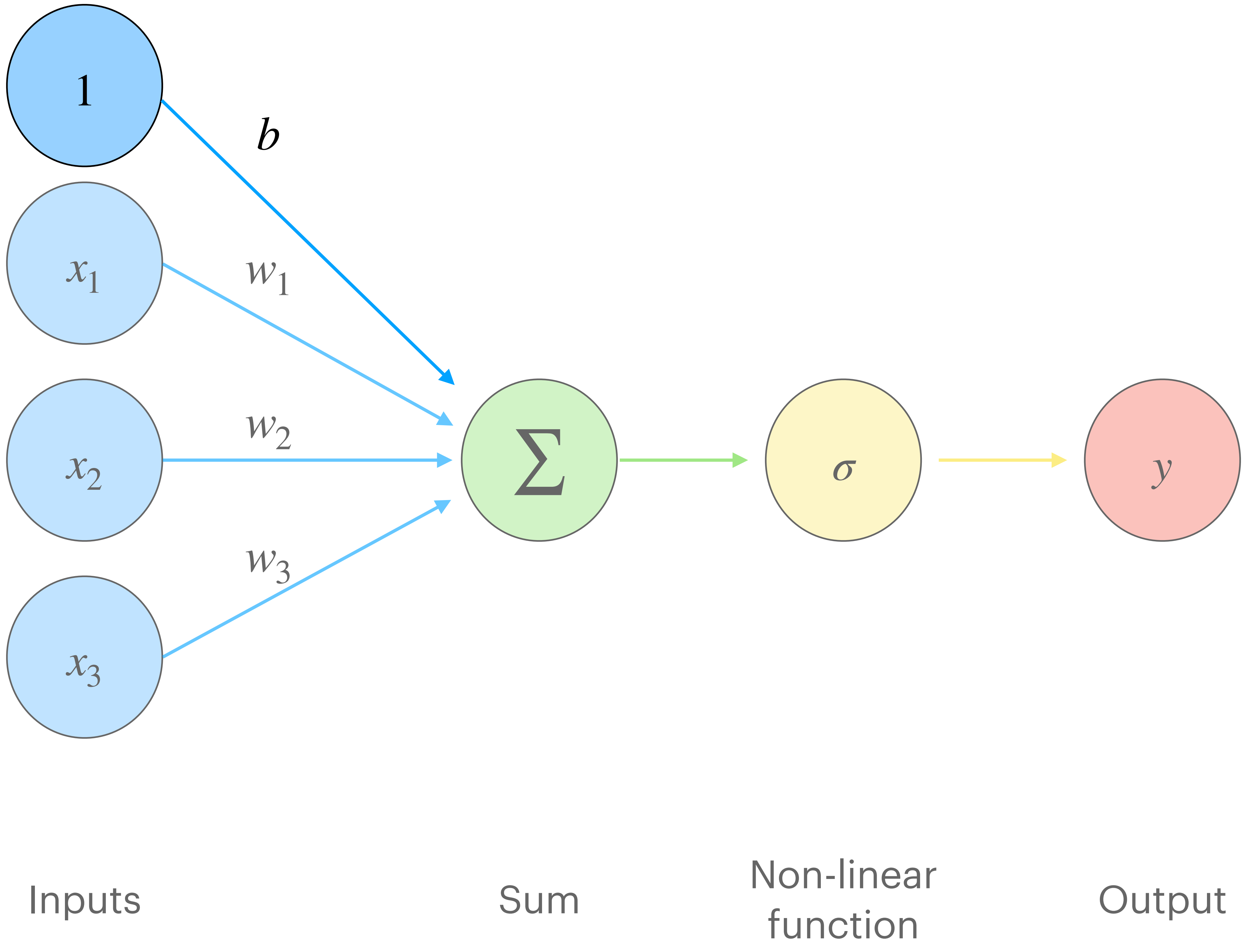
Non-linear function

$$y = \sigma\left(\sum_{i=1}^{m} w_i x_i\right)$$

Output

Linear combination

The Perceptron



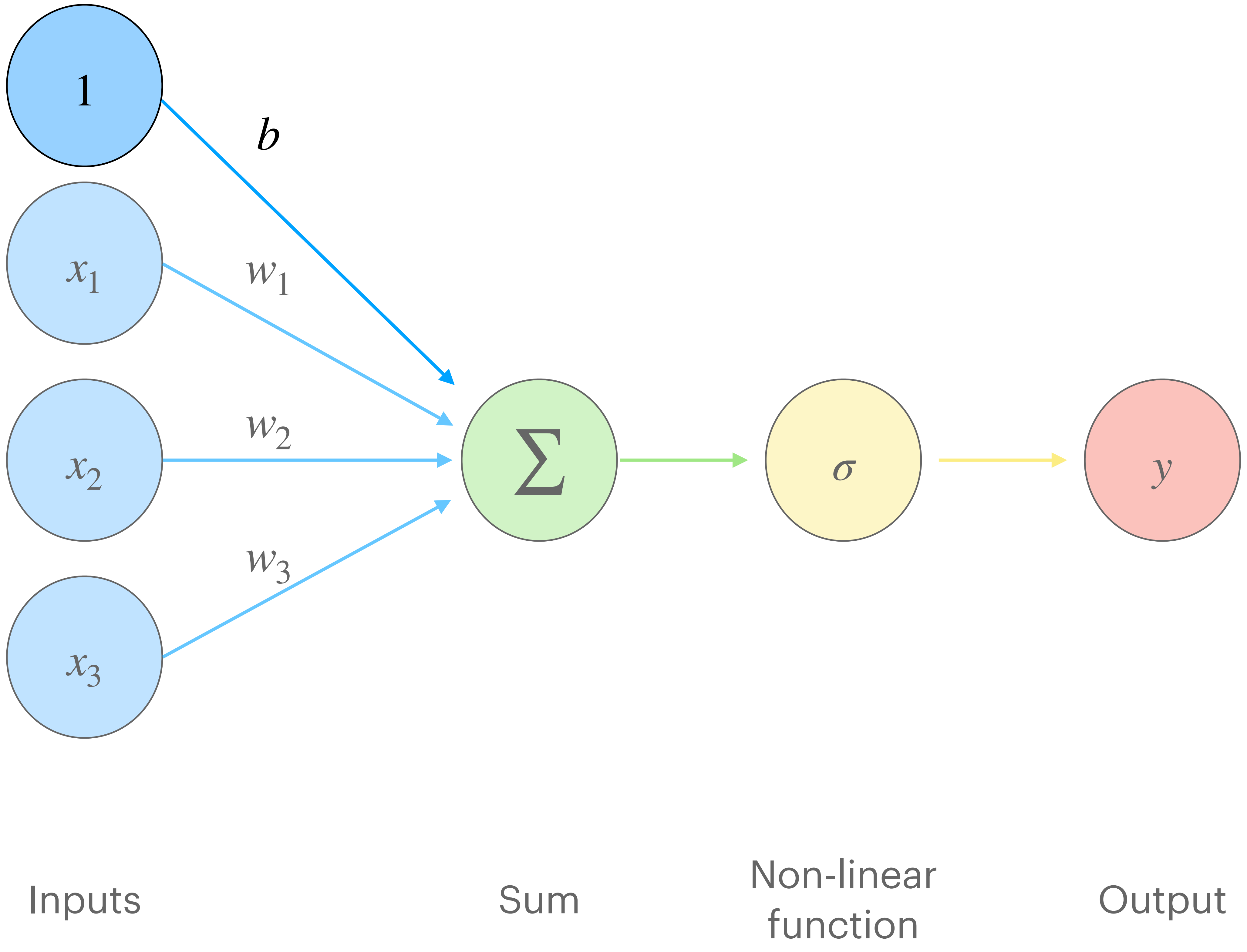
Non-linear function

$$y = \sigma(\sum_{i=1}^{m} w_i x_i + b)$$

Output

Bias Term

The Perceptron



Non-linear function

$$y = \sigma\left(\sum_{i=1}^{m+1} w_i x_i\right)$$

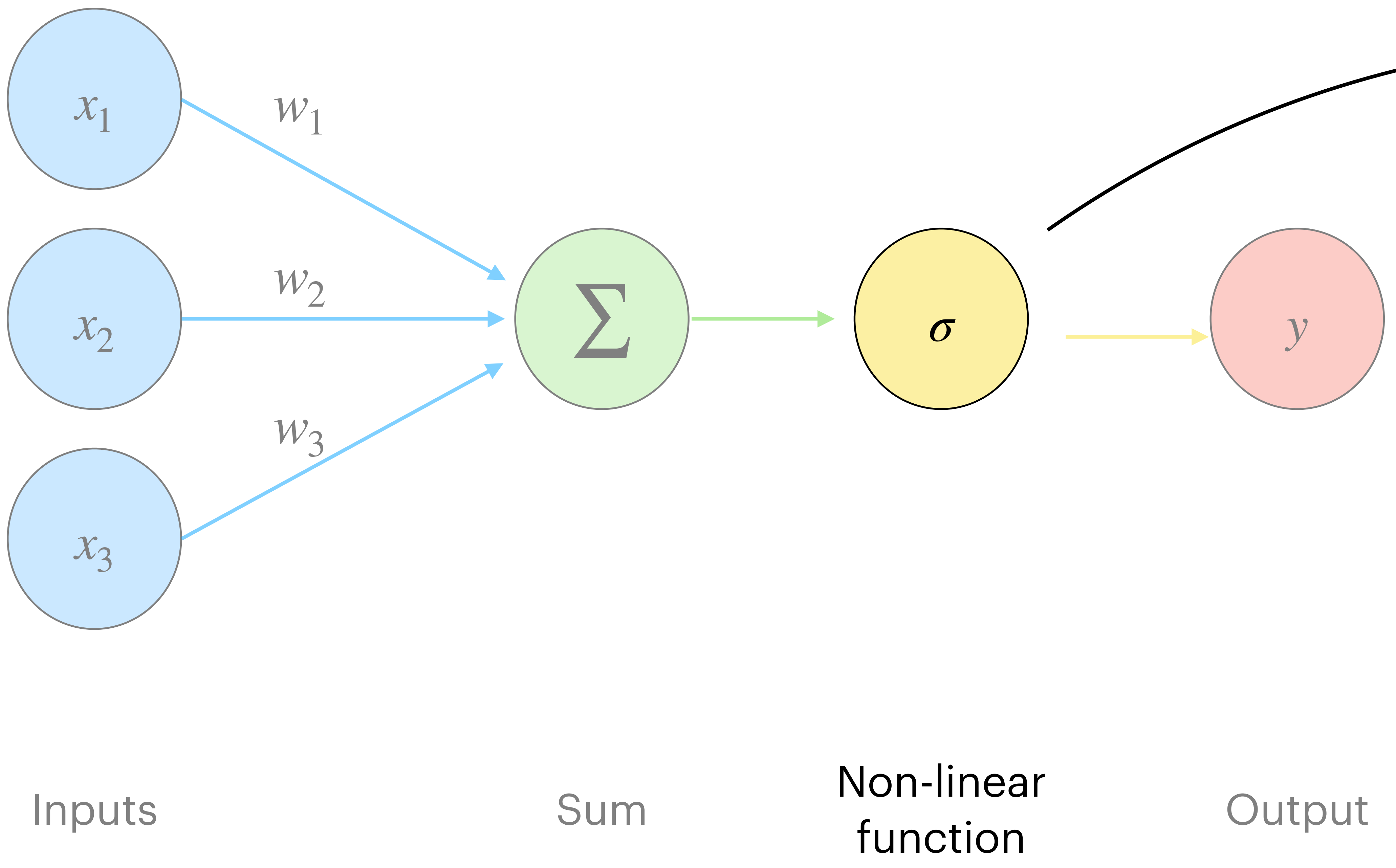
Output

We can include the

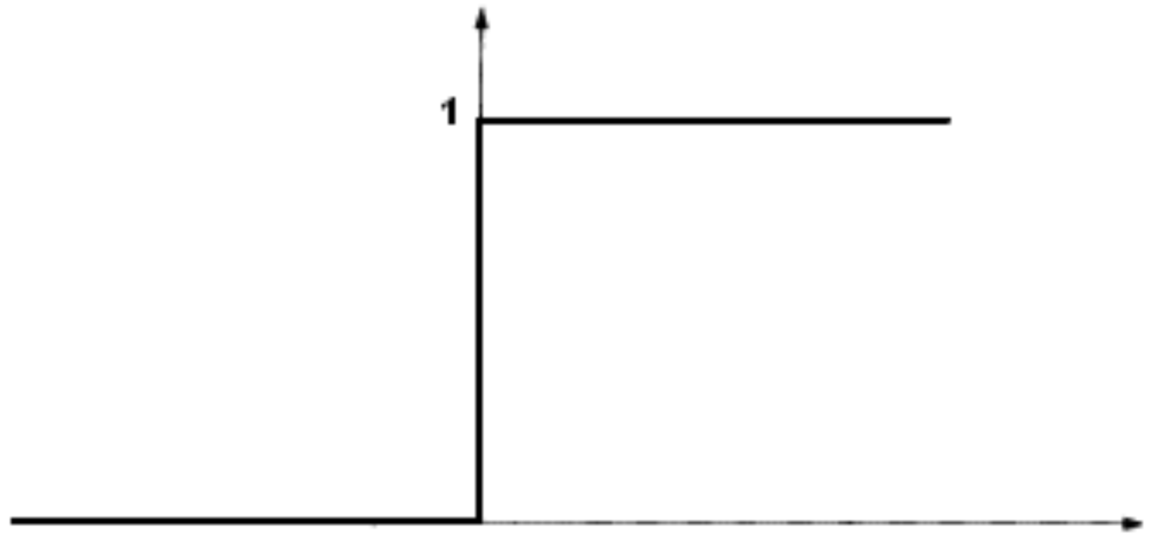
Bias Term

In the weights to make it simpler

The Perceptron

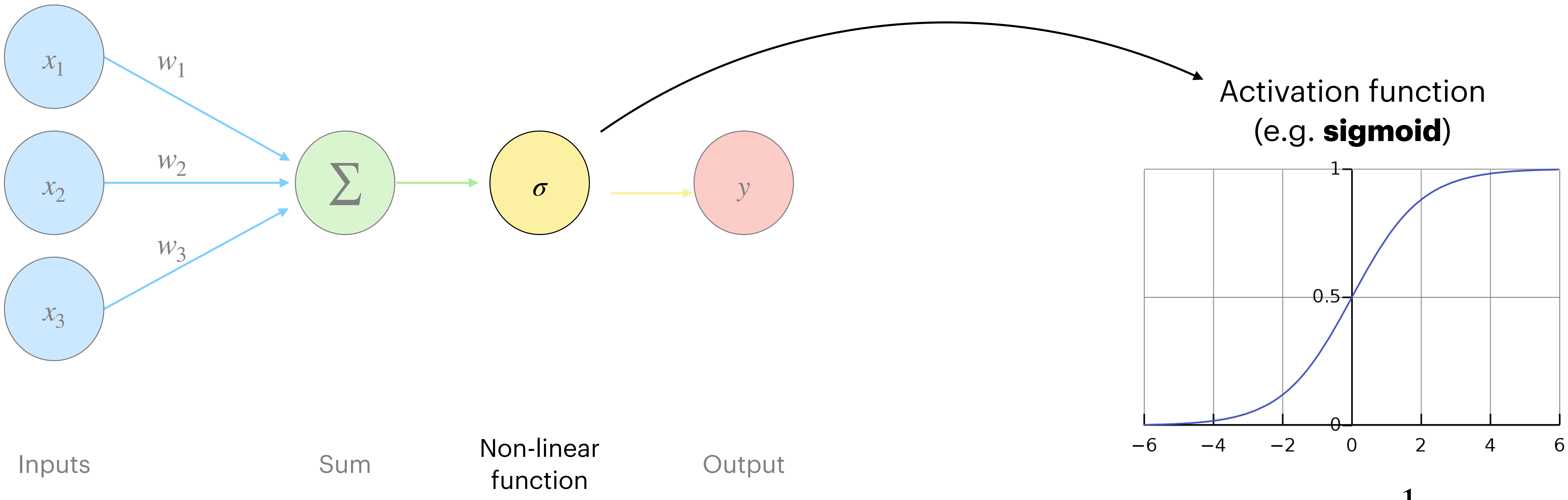


Activation function
(e.g. **threshold**)



$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases}$$

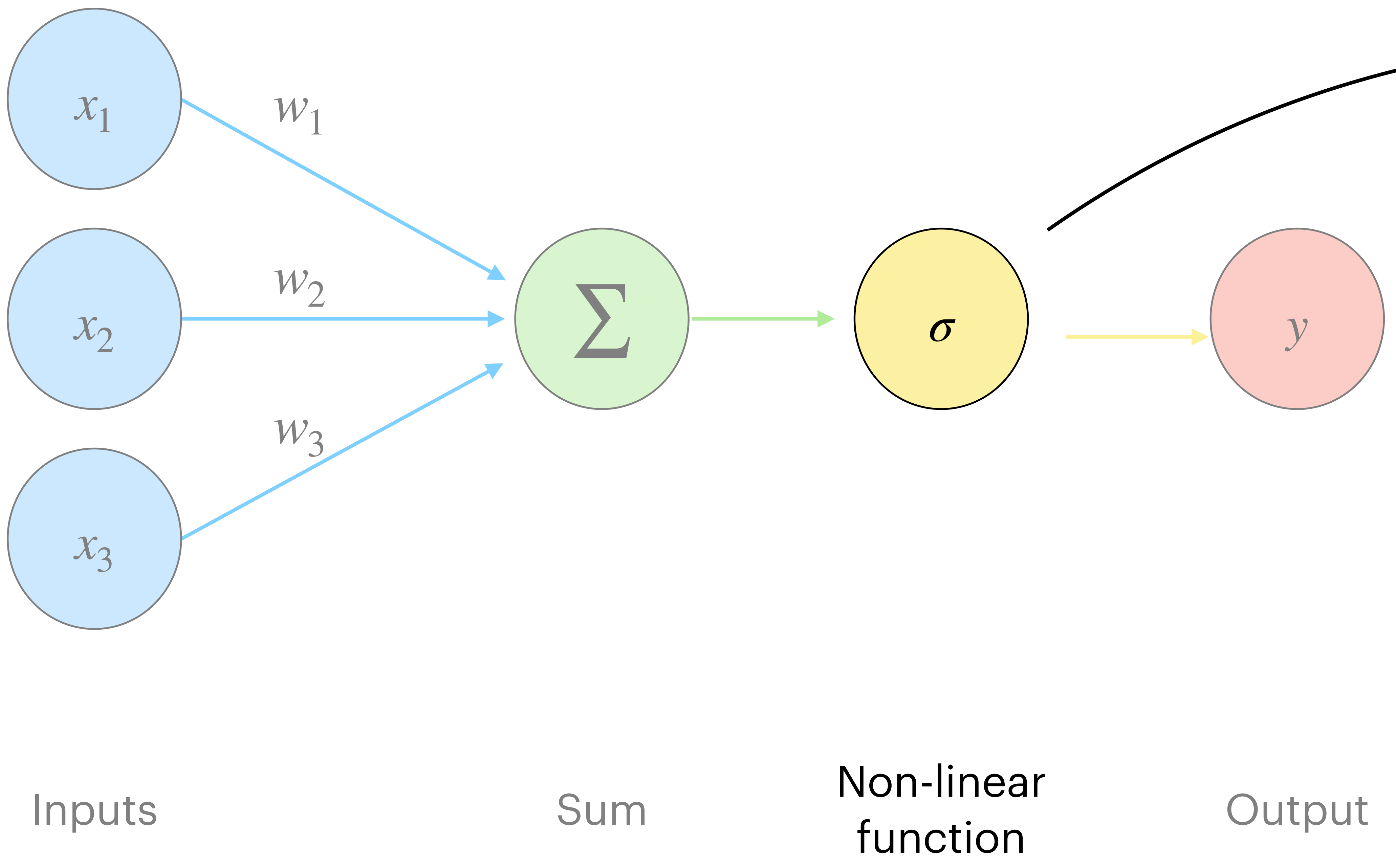
The Perceptron



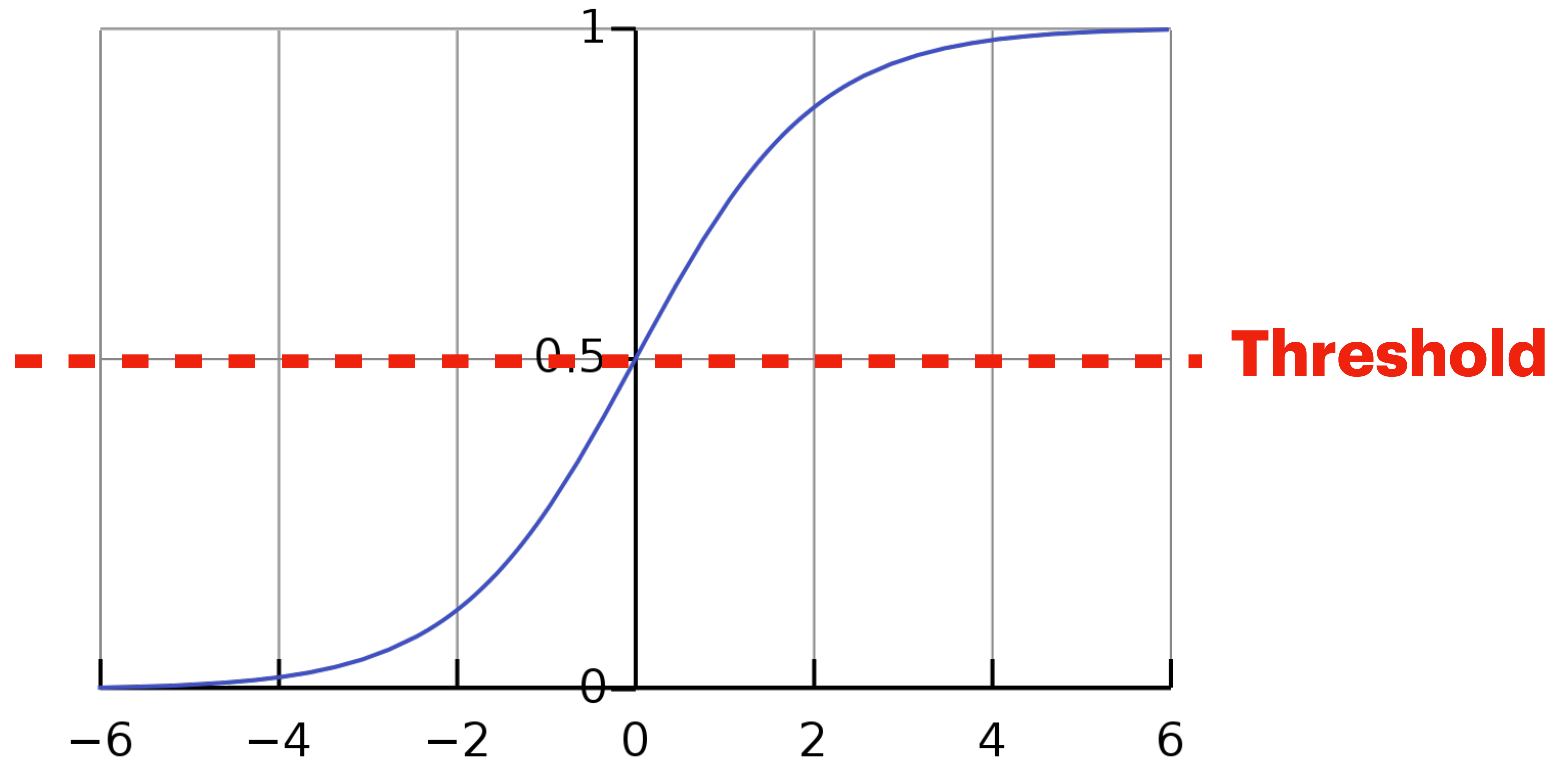
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The Perceptron

Question: Where have you seen this before?



Activation function
(e.g. **sigmoid**)

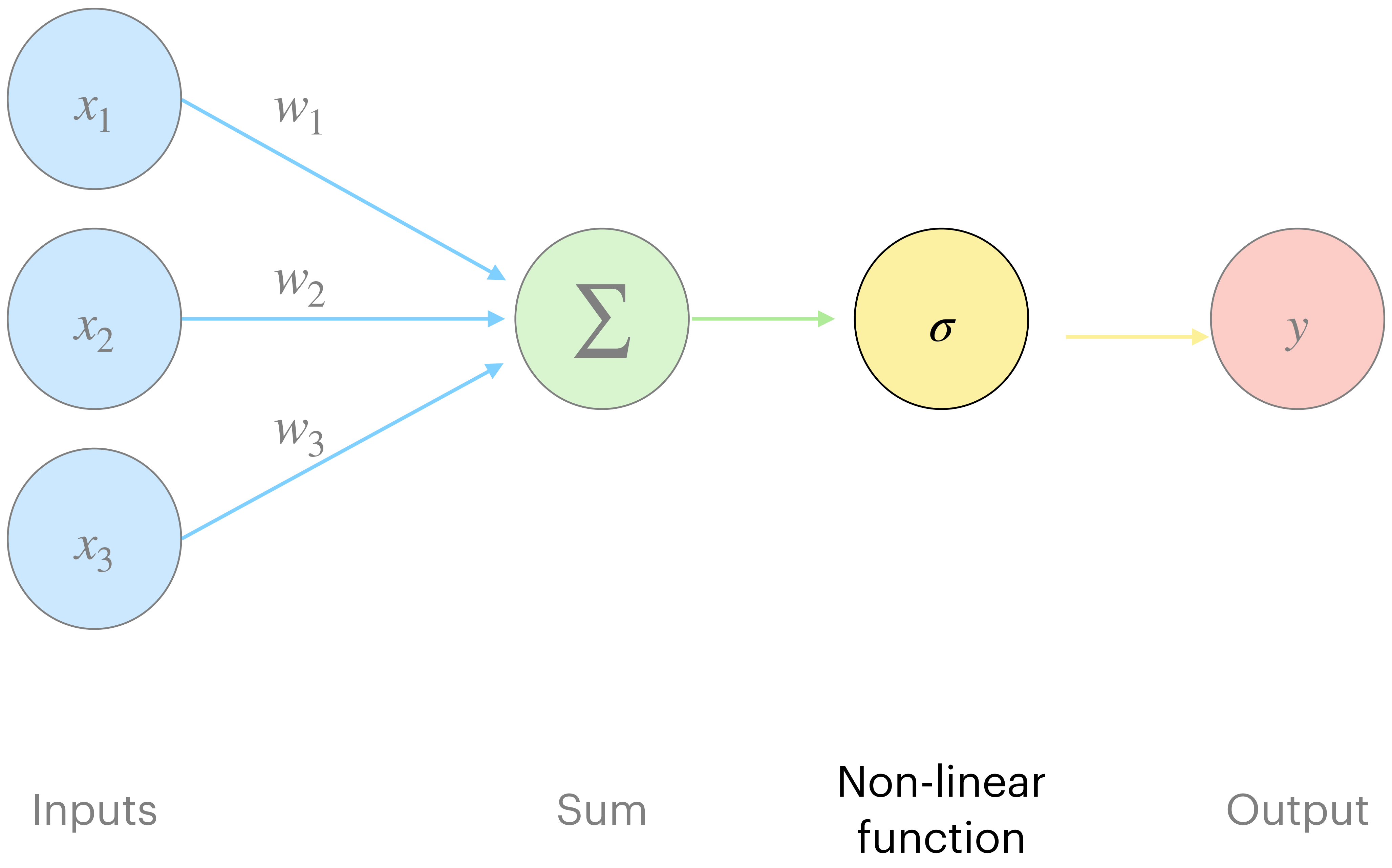


$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

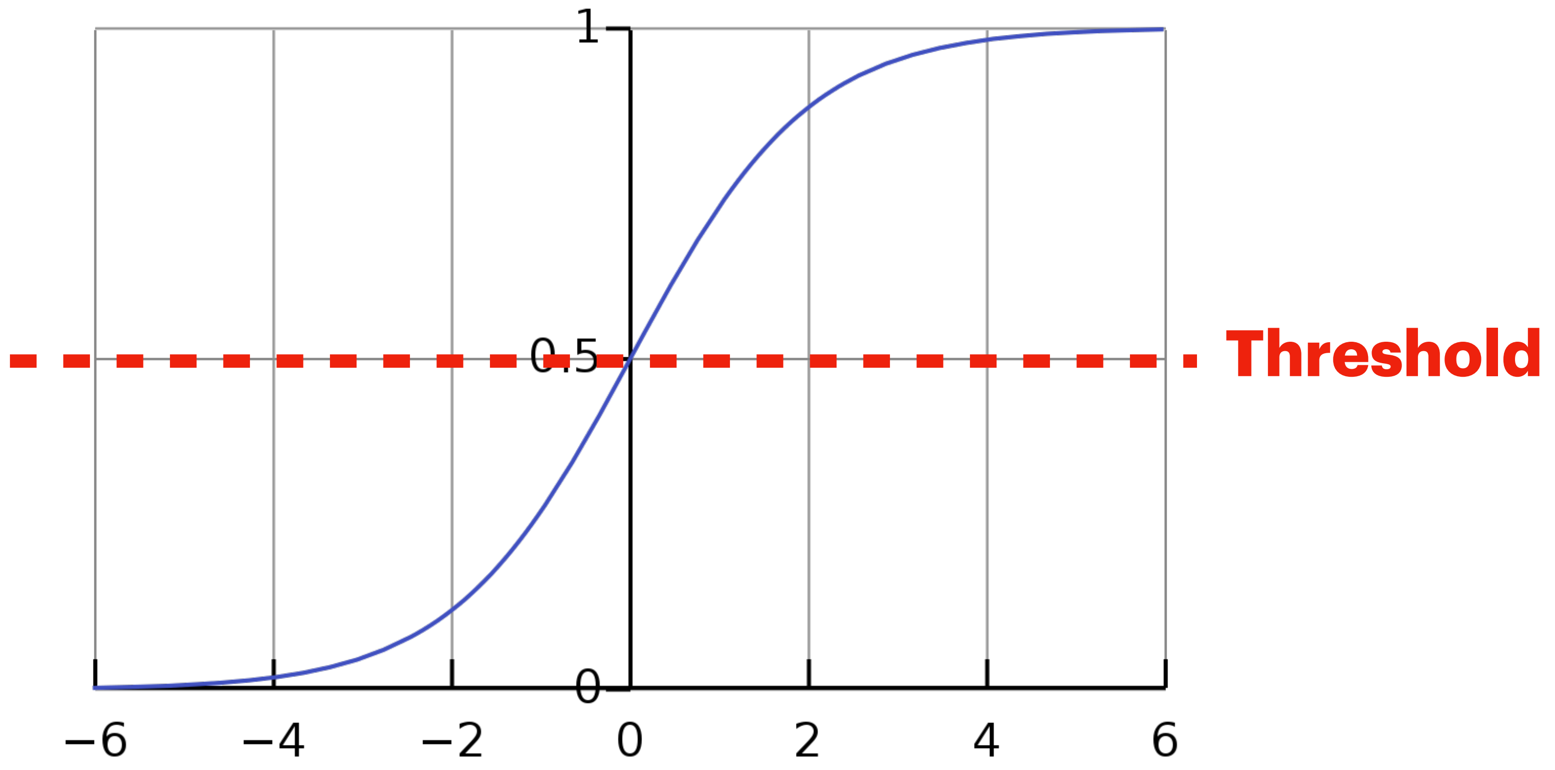
Sources
& Notes

CENTER FOR HUMANITIES COMPUTING

The Perceptron



Activation function
(e.g. **sigmoid**)



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

The Perceptron

Non-linear function

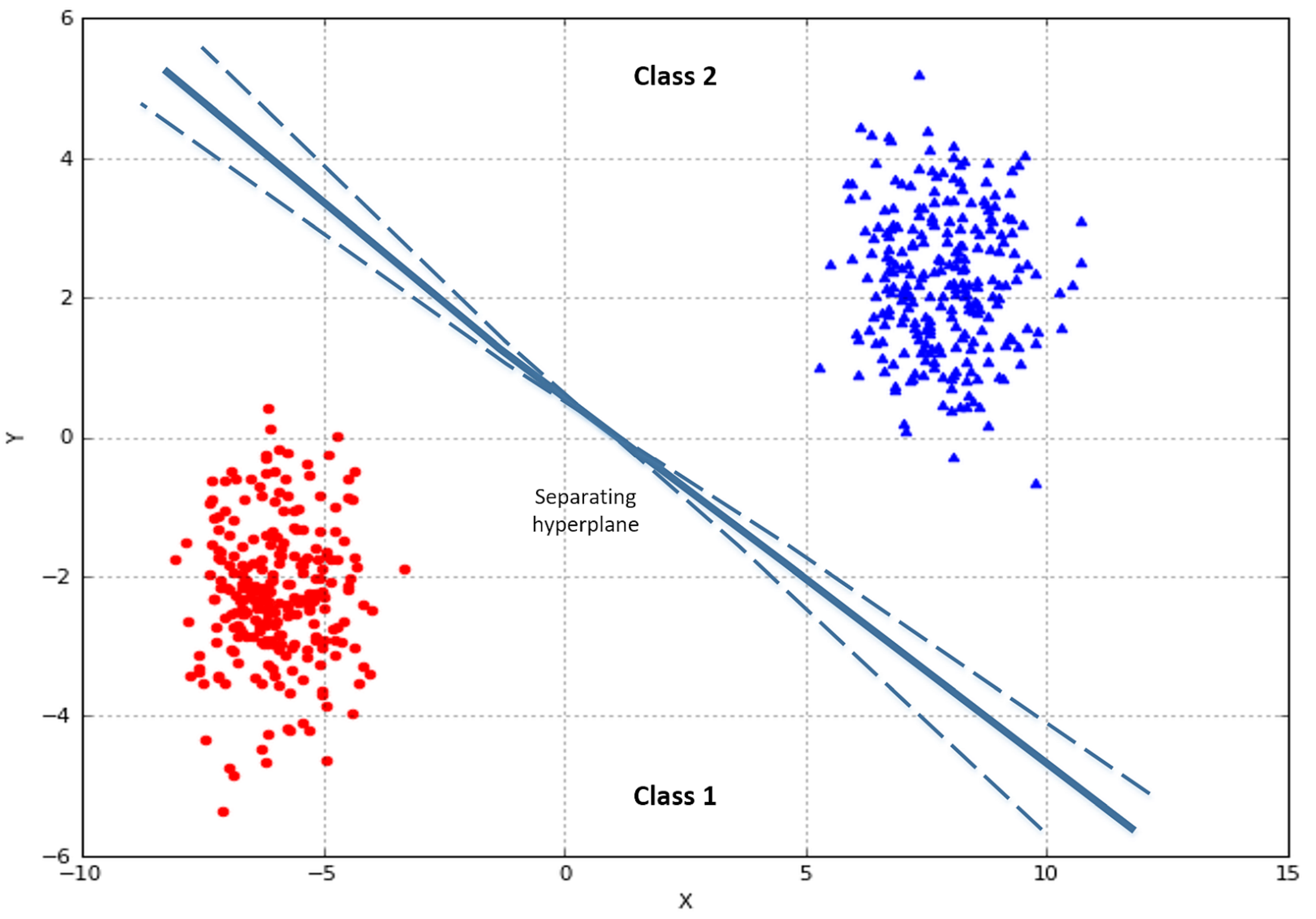
$$y = \sigma(\sum_{i=1}^{m} w_i x_i)$$

Output

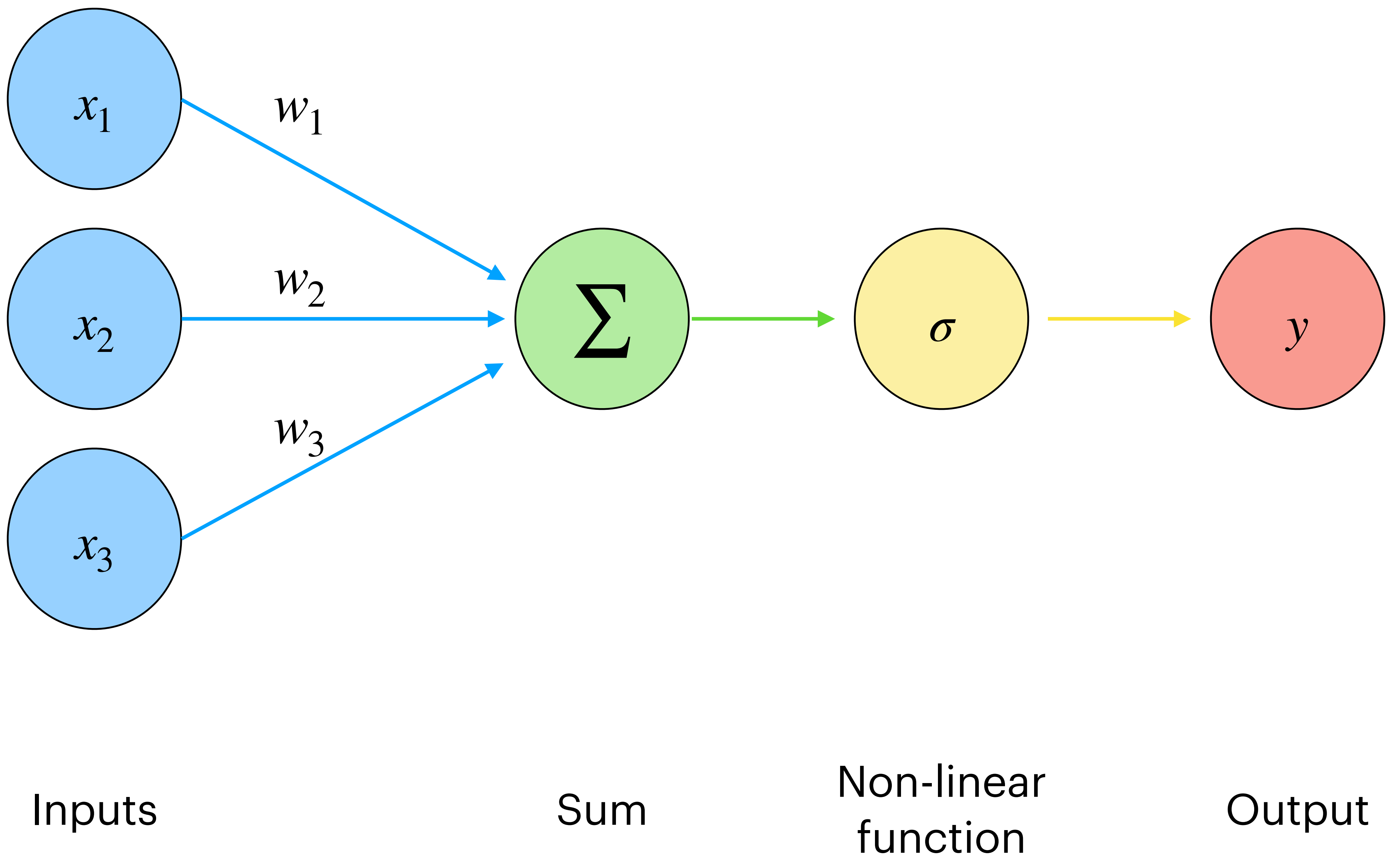
Linear combination

***Linear* classifier**

- Predict binary output
- Defines hyperplane that seperates the classes in feature space



How to we train it? Intuition



Non-linear function

$$y = \sigma\left(\sum_{i=1}^{m} w_i x_i\right)$$

Output

Linear combination

Idea 1: Naïve approach

1. Set weight randomly
2. Measure performance
3. If better: save weights
4. Repeat from step 1

Non-linear function

$$y = \sigma(\sum_{i=1}^{m} w_i x_i)$$

Output

Linear combination

Idea 2: Slightly smarter approach

1. For each weight
   i. Change weight slighly up or down

      Let us call how much we change it the **learning rate**

   ii. If it gives better performance choose the one that it best
2. Repeat from step 1. until satisfied

Non-linear function

$$y = \sigma(\sum_{i=1}^{m} w_i x_i)$$

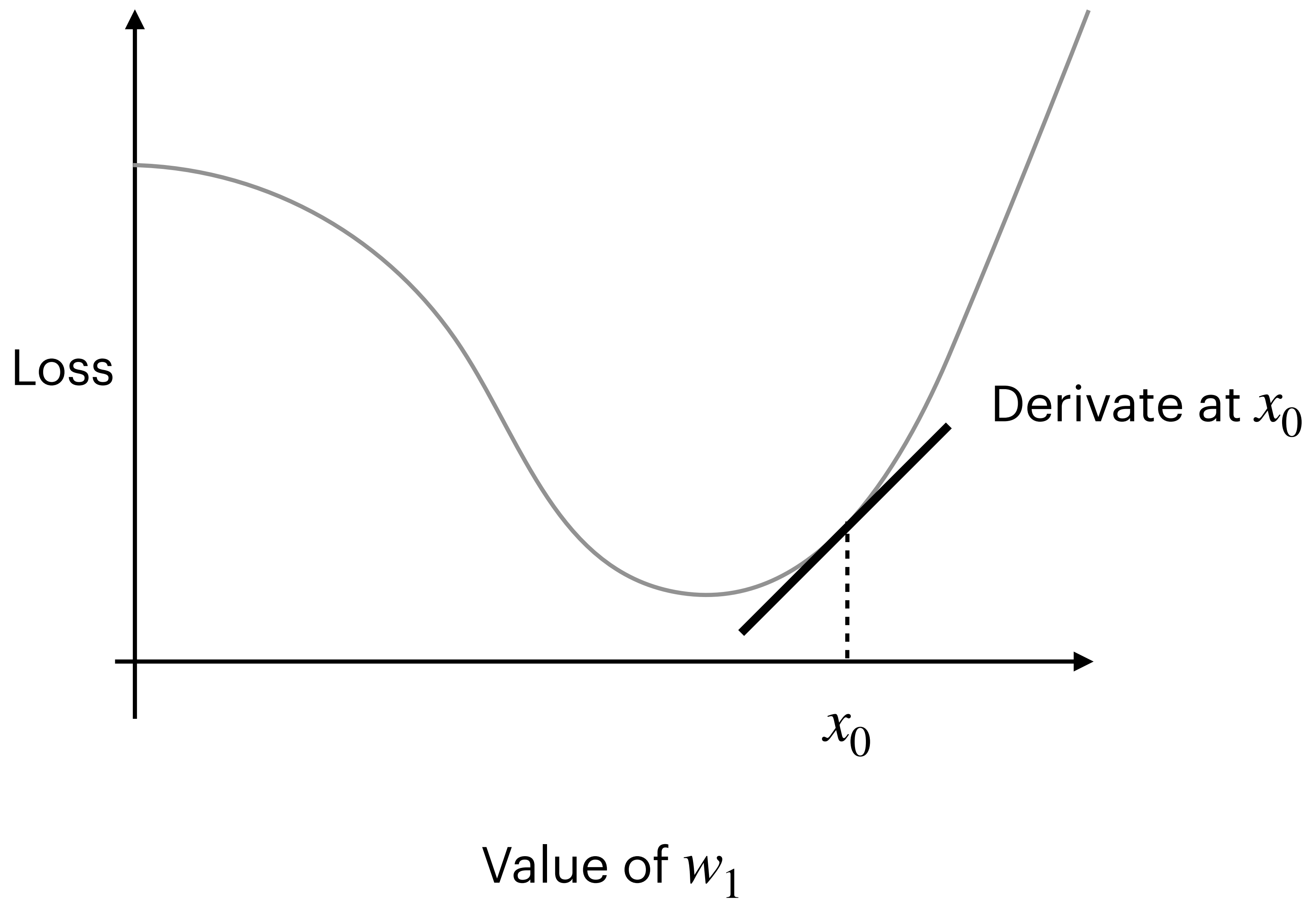
Output

Linear combination

Idea 3: Gradient Descent

We just approximated gradient decent

1. Choose a loss
2. Calculate a gradient
    i. Take a step down the gradient proportional to the gradient
3. Repeat from step 1. until satisfied



Measuring what is good: MSE Loss

- $L(y, \hat{y}) =$ how much does my **prediction** $\hat{y}$ differs from the **true label** y
  $L(y, \hat{y}) = \frac{1}{n} \sum_i \parallel y_i - \hat{y}_i \parallel^2$
- For binary signal is between 0-1
- Good for regression

Measuring what is good: Cross-Entropy Loss

For Classification:

- $L(y, \hat{y}) =$ how much does my **prediction** $\hat{y}$ differs from the **true label** y

$$\downarrow$$

$$L(y, \hat{y}) = y \log \hat{y} + (1 - y) \log(1 - \hat{y})$$

$$\downarrow$$

- **Cross-entropy loss**
  for a single example

Measuring what is good: Cross-Entropy Loss

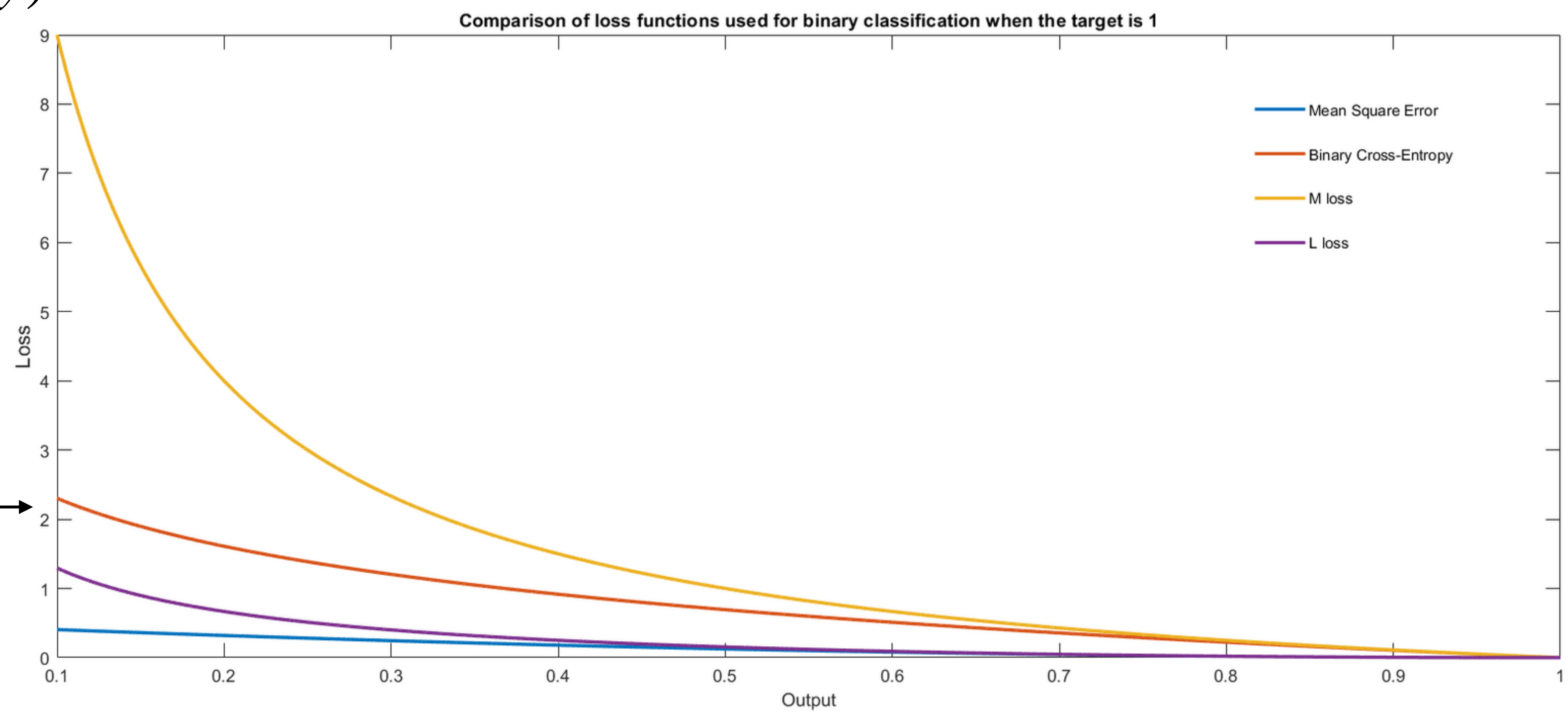
For Classification:

- $L(y, \hat{y}) =$ how much does my **prediction** $\hat{y}$ differs from the **true label** y

$$L(y, \hat{y}) = y \log \hat{y} + (1 - y) \log(1 - \hat{y})$$

- **Cross-entropy loss** for a single example

Let us plot it



Sources & Notes

Source: https://en.wikipedia.org/wiki/Cross-entropy#Cross-entropy_loss_function_and_logistic_regression

Example in PyTorch

- **Forward pass**
- Backward pass

More on this in class

```
import torch

perceptron = torch.nn.Linear(2, 1)

# Weights
print(perceptron.weight)
# Parameter containing:
# tensor([[0.0224, 0.0251]], requires_grad=True)

# Bias
print(perceptron.bias)
# Parameter containing:
# tensor([-0.1012], requires_grad=True)

# Forward pass
x = torch.tensor([0.5, 0.5])
output = perceptron(x)
output = torch.sigmoid(output) # activation function
print(output)
# tensor([0.4806], grad_fn=<SigmoidBackward0>)
```


Note: Activation function is excluded

Example in PyTorch

- Forward pass
- **Backward pass**

More on this in class

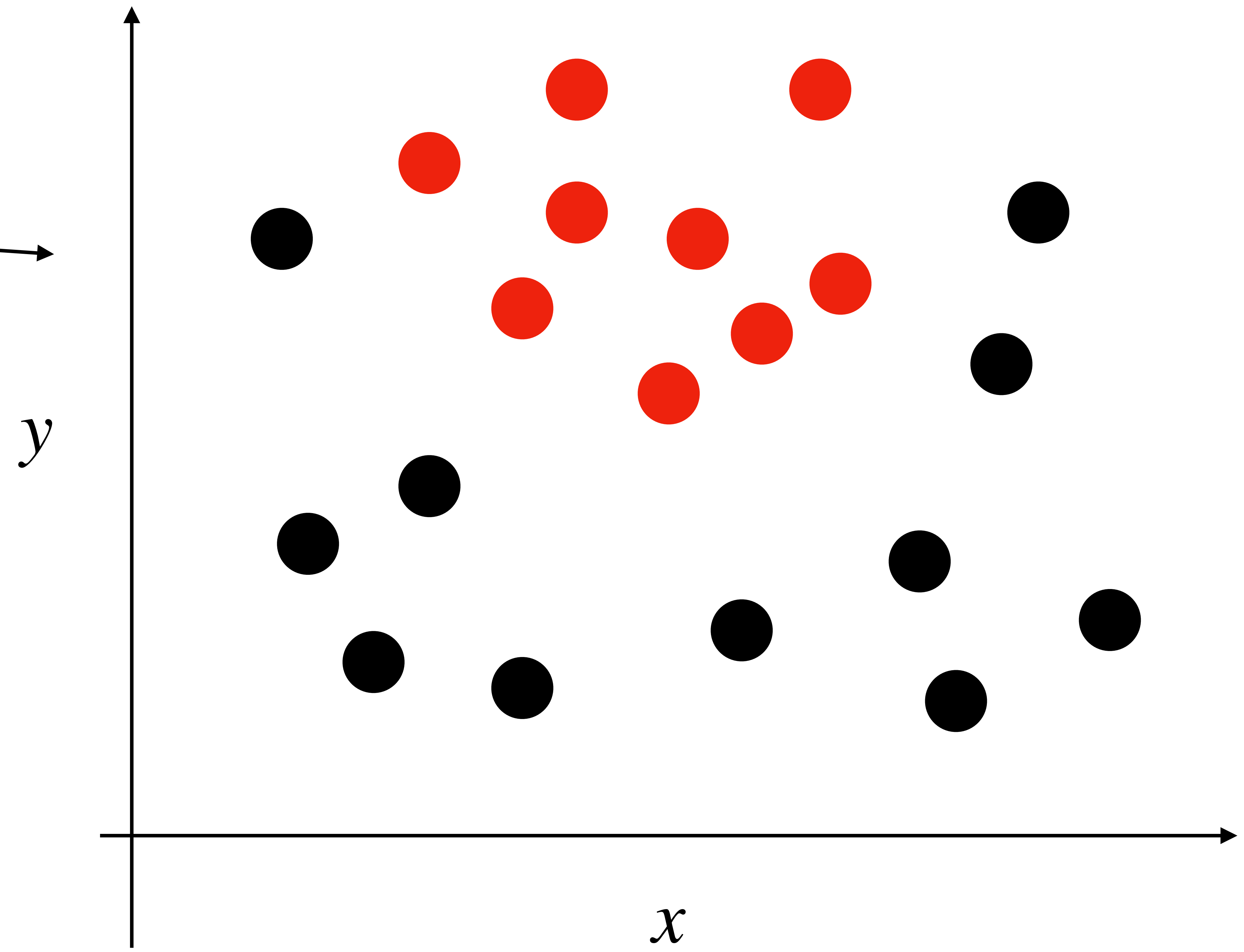
```
# Loss
y = torch.tensor([1.0]) # true value
loss = torch.nn.functional.mse_loss(output, y)
loss.backward() # calculate gradients

# Examining the gradient
print(perceptron.bias.grad)
# tensor([-1.1061])
```

We calculate the gradient here

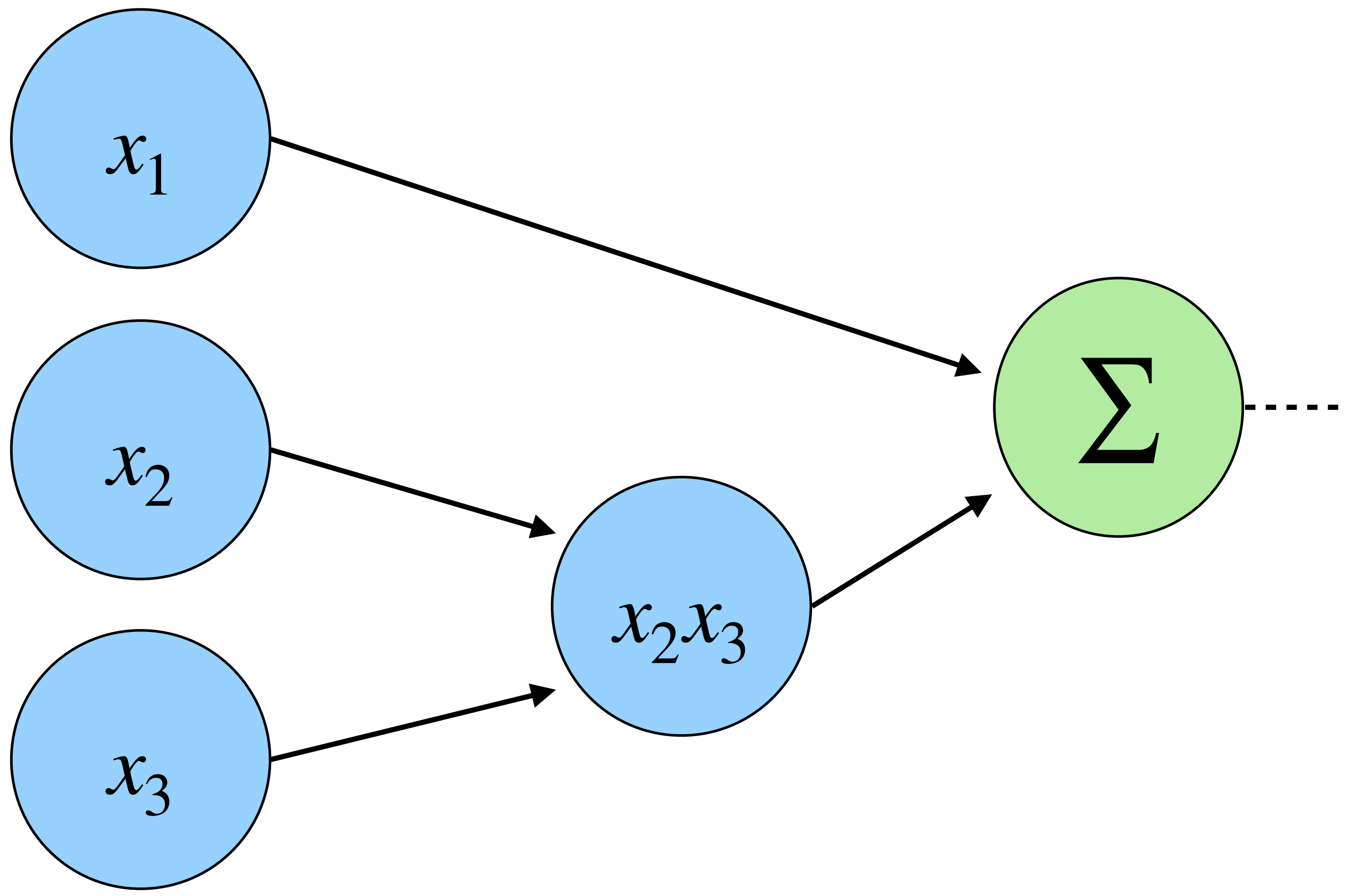
Beyond the Perceptron

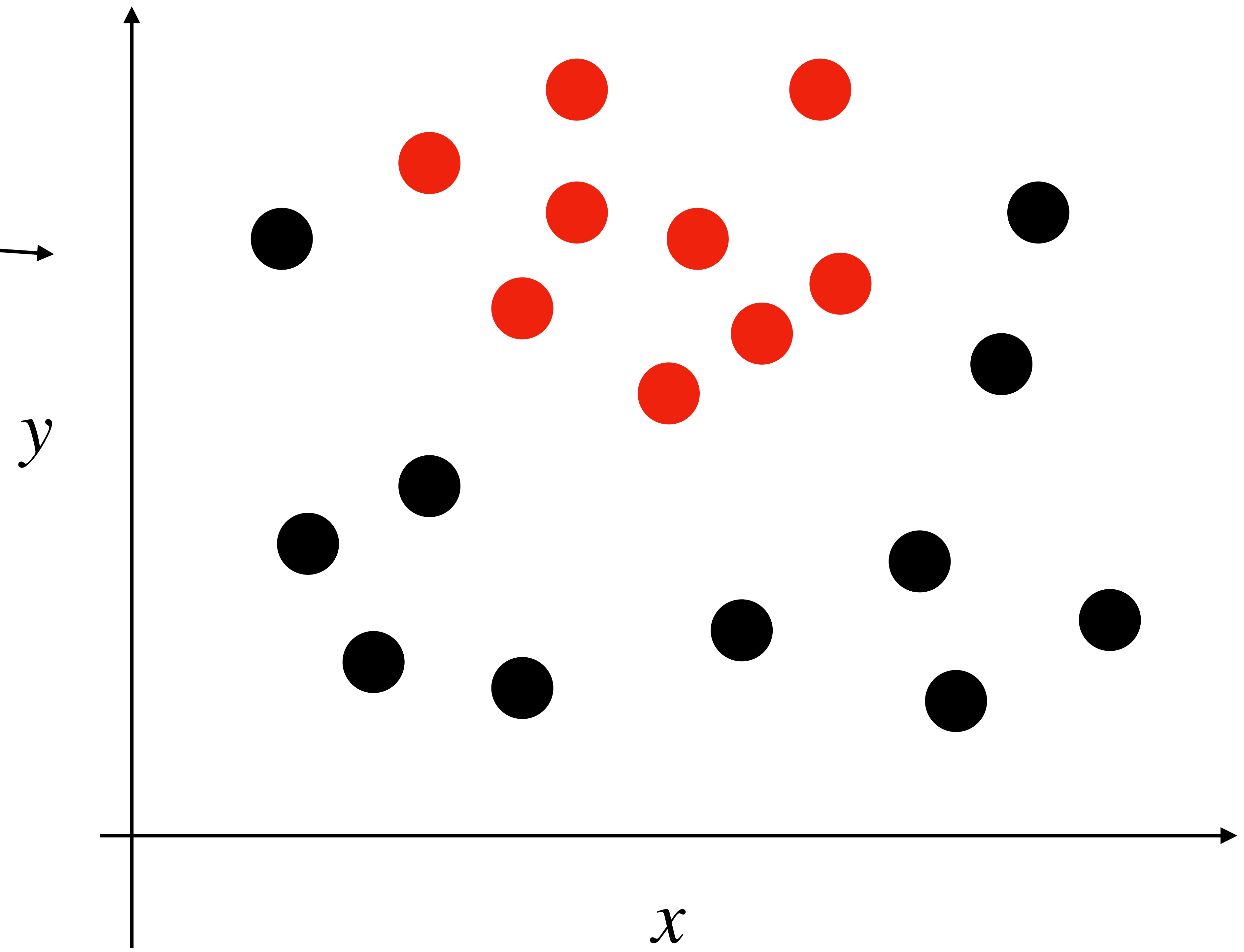
- A lot of real data in **non-linear**
- Question: How would you solve hint: what would you do in a linear model?



Beyond the Perceptron

- A lot of real data in **non-linear**
- Question: How would you solve hint: what would you do in a linear model?
- **Interaction!**

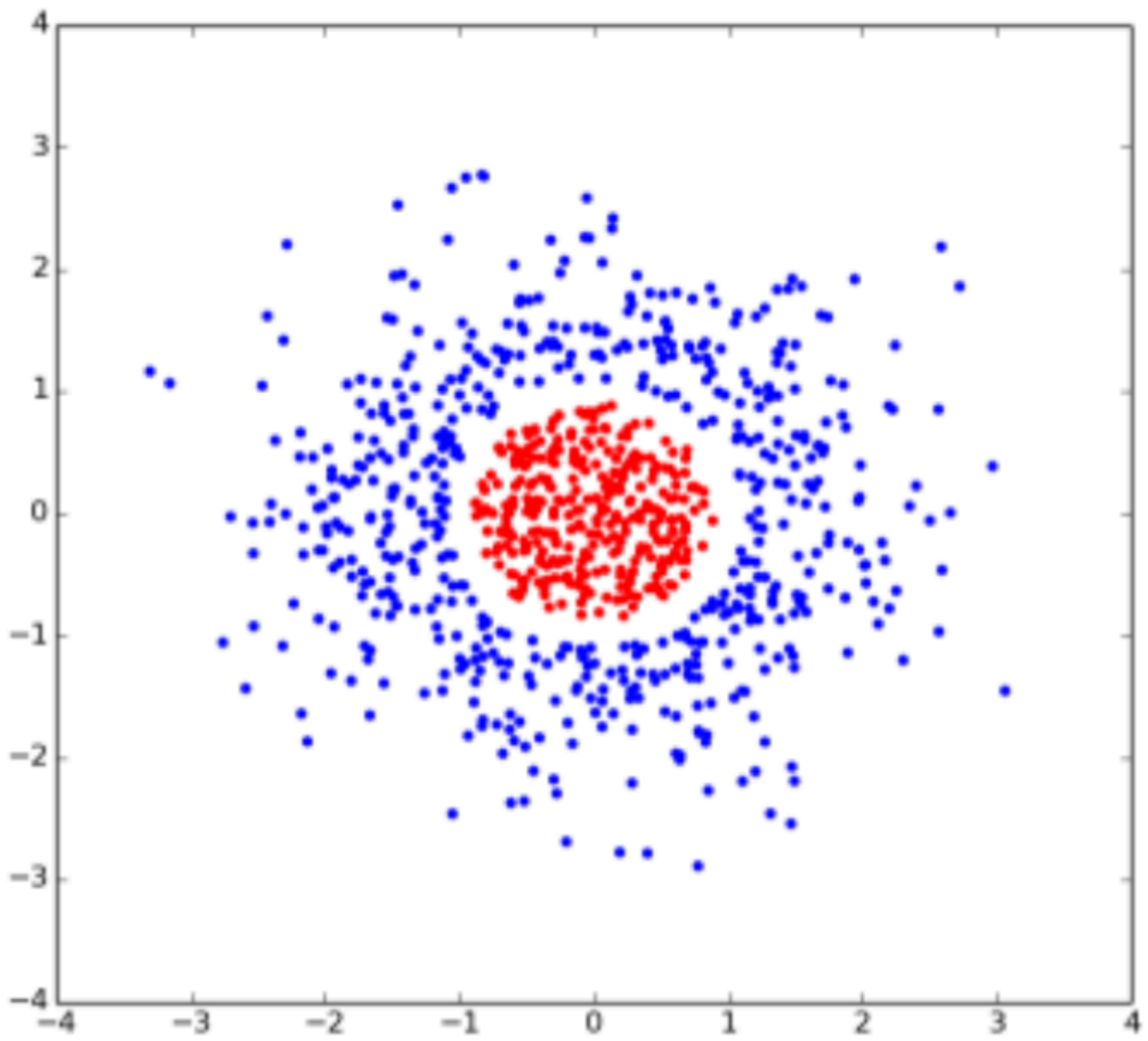




E.g. inputting x^2 would love it in this case

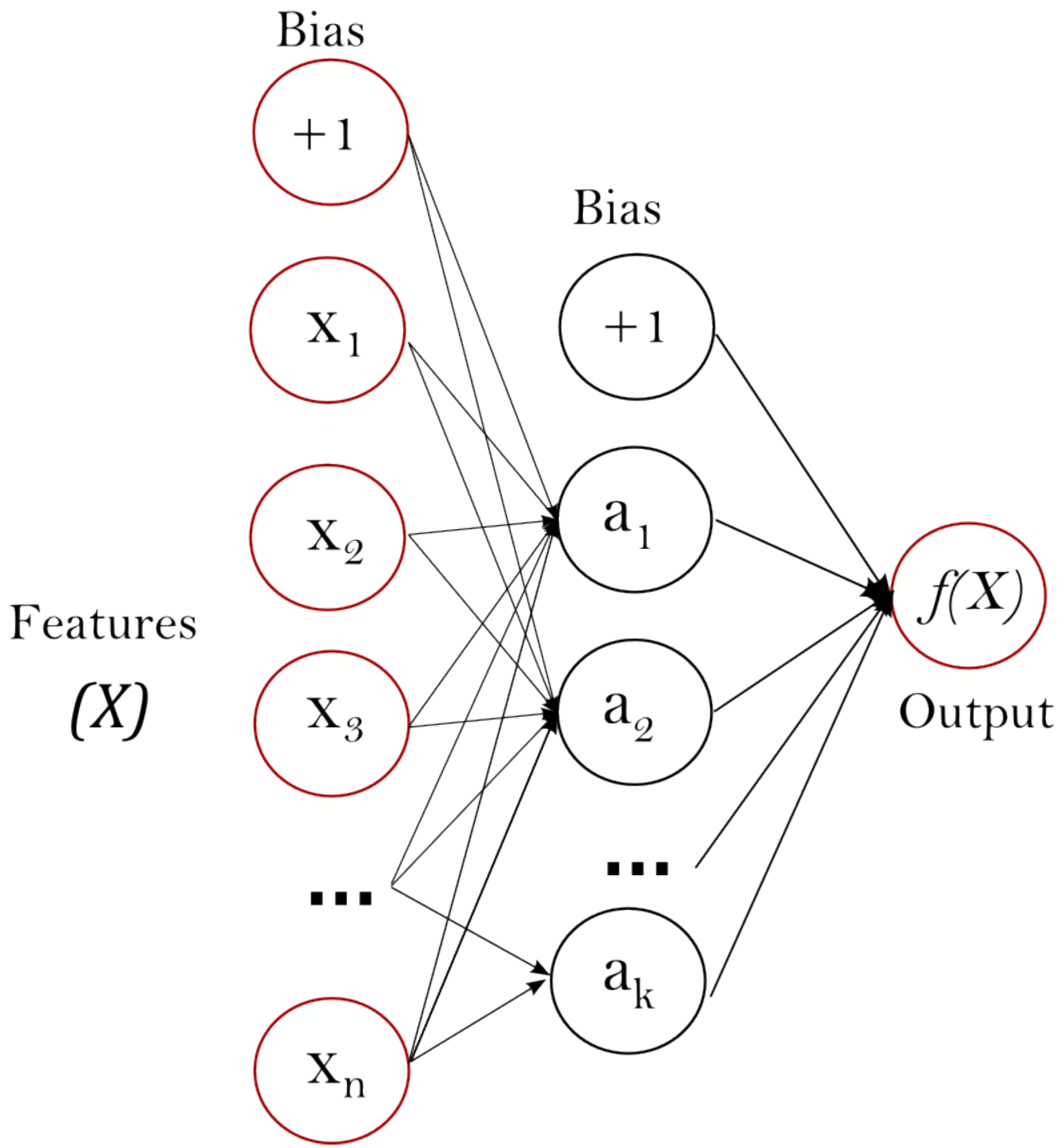
Beyond the Perceptron

- What about this one
- We need something that in **non-linear** and **arbitrarily complex**



Neural Networks

- A **network** of neurons
- Also called
  - multilayer perceptron (MLP)
  - Fully-connected feedforward neural network (FFN, FFNN)
  - Dense neural network
- Stacking neurons with **non-linear** activations allow us to model arbitrarily complex functions*
  - Layer in the middle are called **hidden layers**

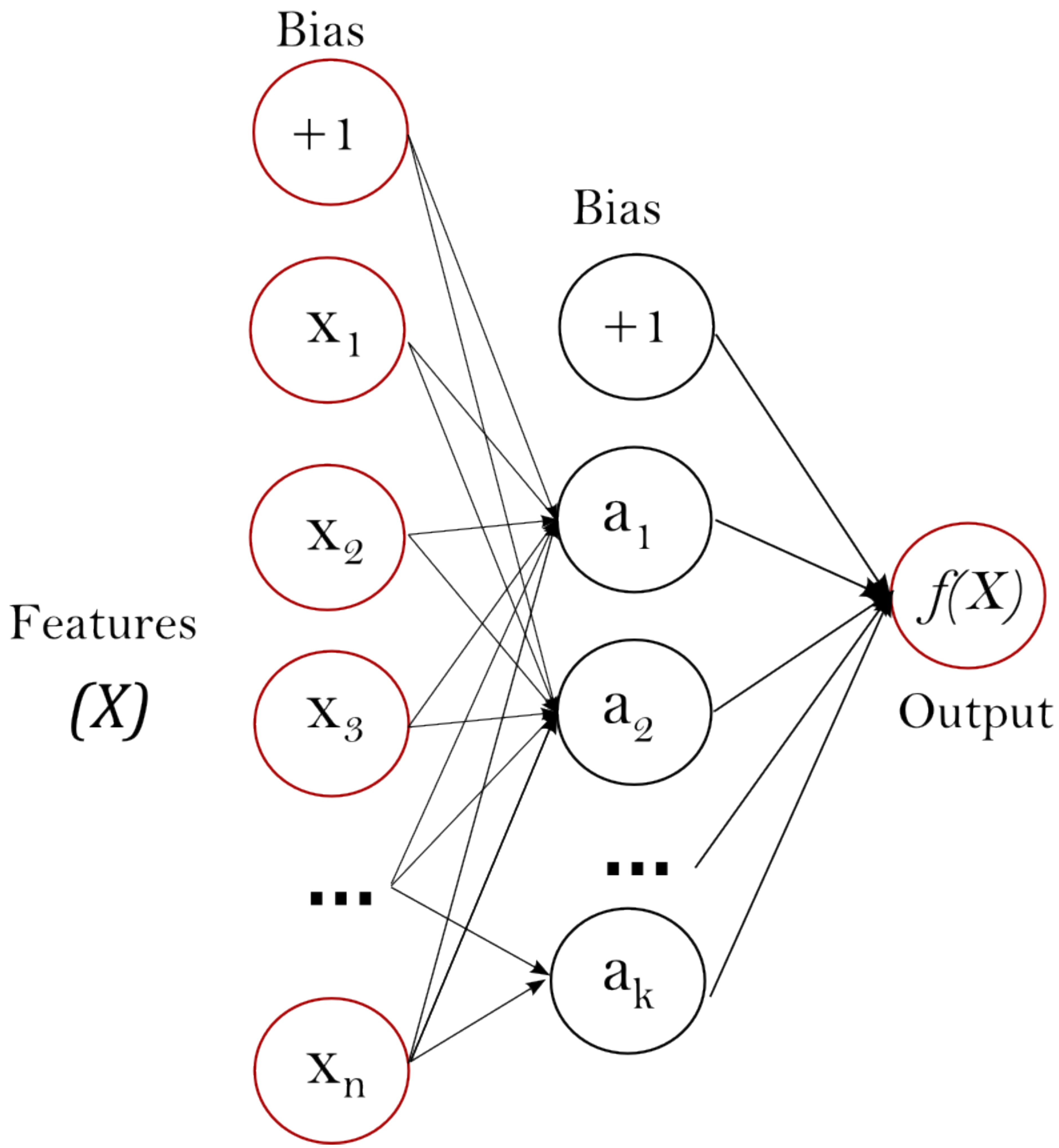


* See also https://en.wikipedia.org/wiki/Universal_approximation_theorem

Neural Networks

- Where:

$$a_k = \sigma(\sum_{i=1}^{m} w_{i,k} x_i)$$



Sources & Notes

* See also https://en.wikipedia.org/wiki/Universal_approximation_theorem

CENTER FOR HUMANITIES COMPUTING

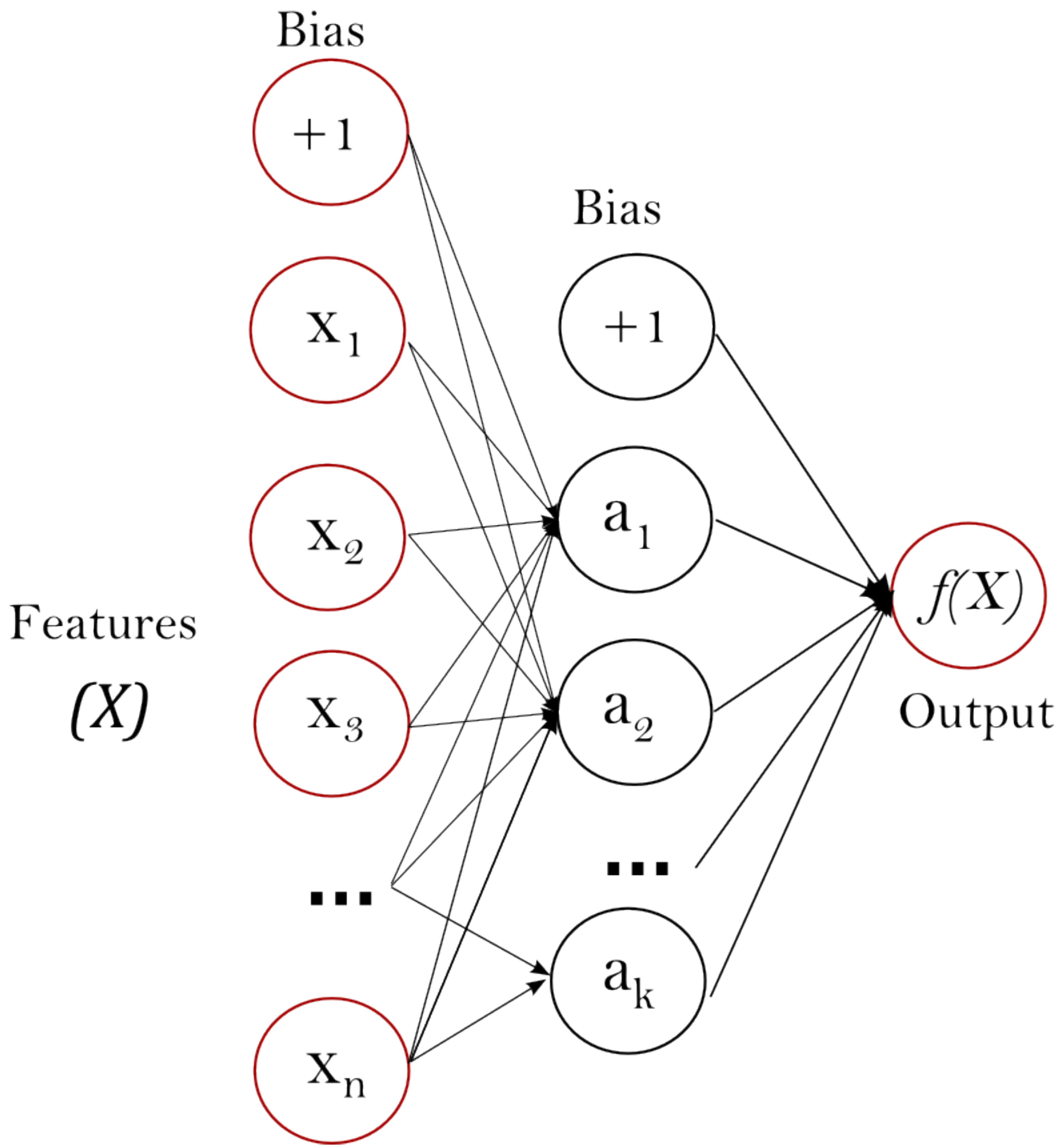
Neural Networks

- Where:

$$a_k = \sigma(\sum_{i=1}^{m} w_{i,k} x_i)$$

$$= \sigma(WX)$$

Matrix notation to simplify



* See also https://en.wikipedia.org/wiki/Universal_approximation_theorem

Multilayer Neural Networks

- Can have any number of layers (**deep** neural network)
- Or any number of nodes (**wide** neural networks)
- Computational complexity grows fast!
  - 300 dimensional input —> 1000 wide hidden layer
  - **300,000** connections
    - A single layer (e.g. linear regression) is just **300**

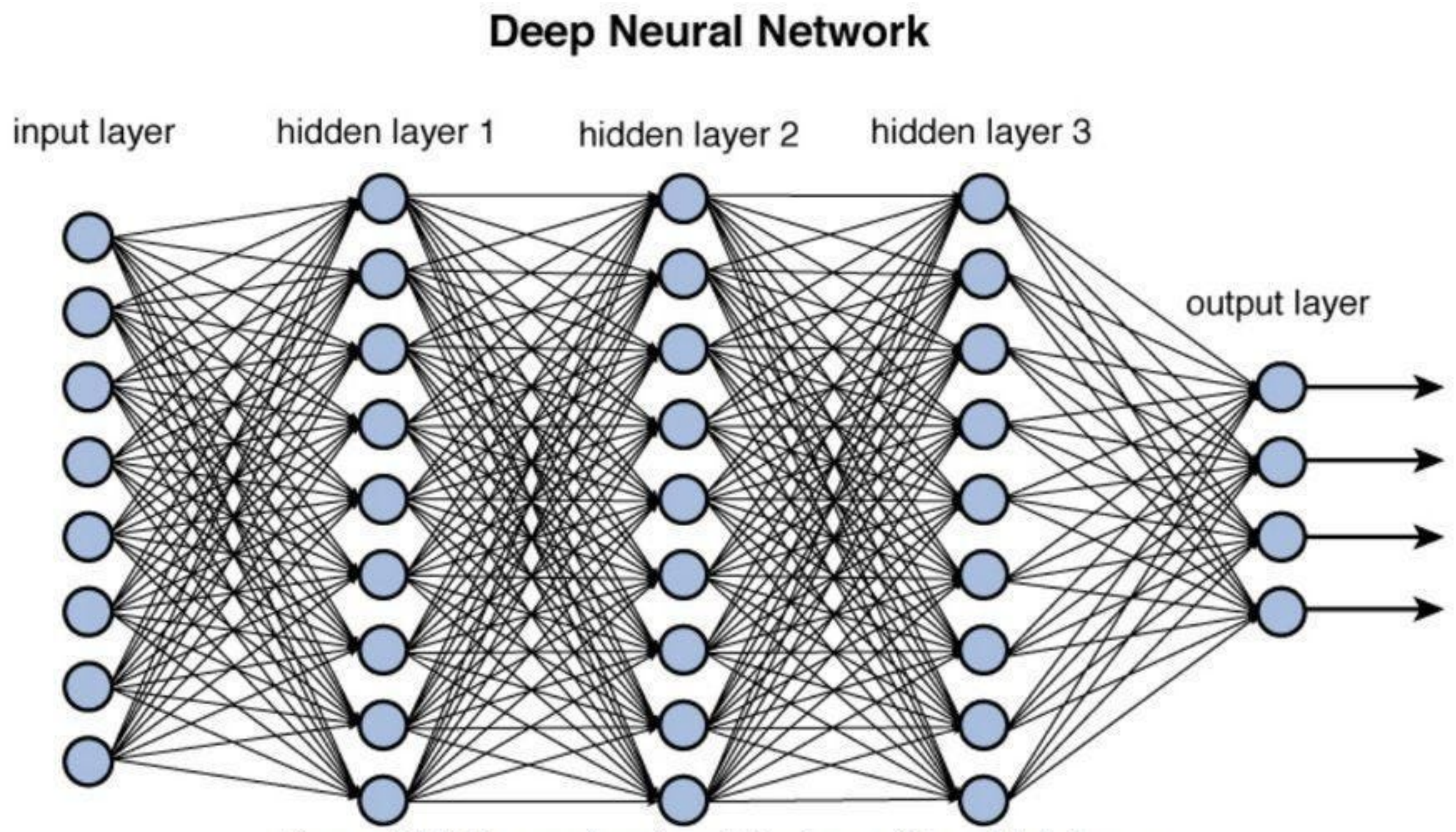


Figure 12.2 Deep network architecture with multiple layers.

Fitting a neural Network

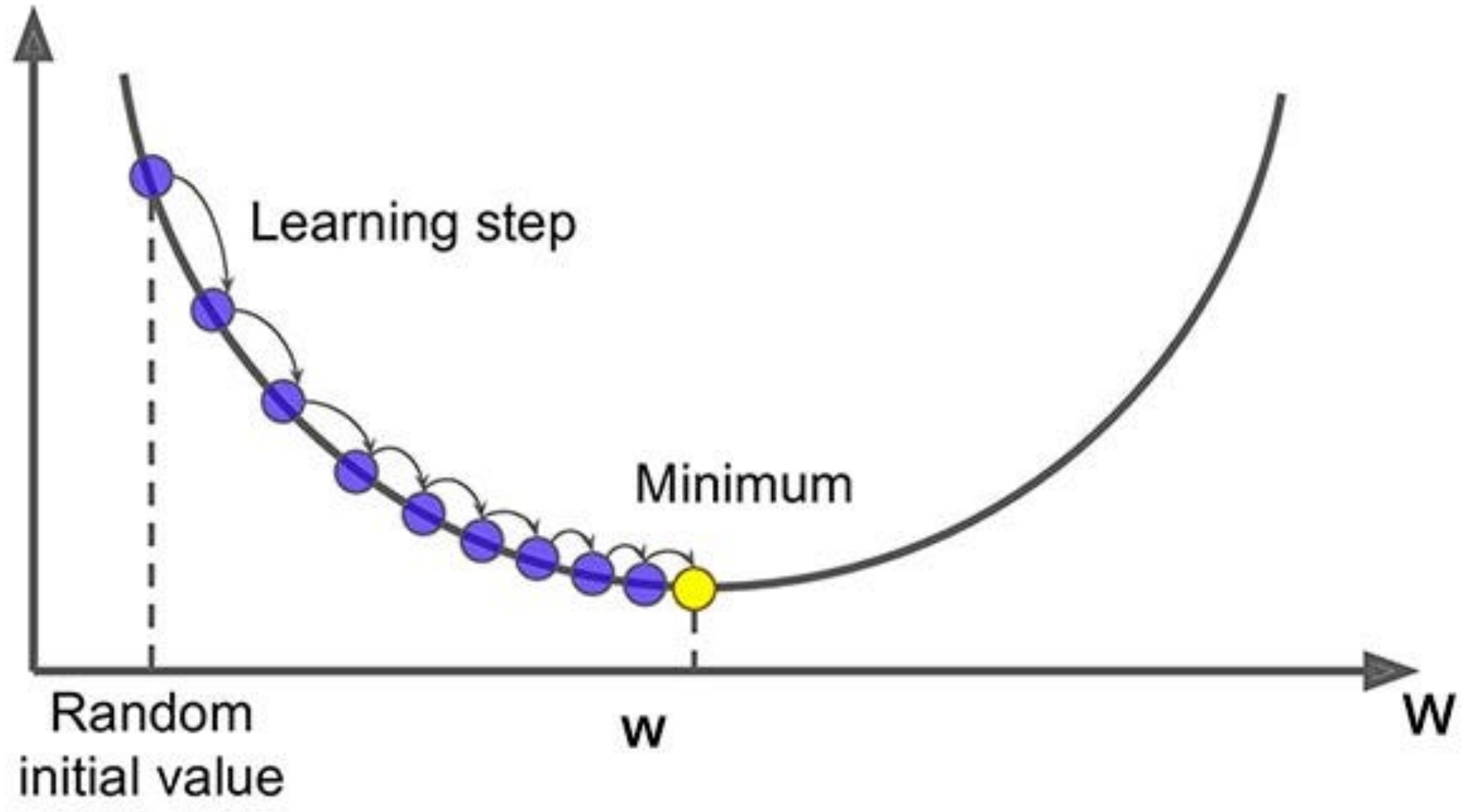
1. **Randomly** initialize weights of the network
2. Calculate the **forward pass** (produce $\hat{y}$)
3. Calculate the **loss** (prediction error)
4. Calculate the gradient (**backward pass**)
5. **Adjust weights** based on gradient
6. Repeat until satisfied

Goal

- Find the best parameters θ that minimized the cross entropy over all examples m:

$$\hat{\theta} = \underset{\theta}{\text{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{CE} f(x_i; \theta), y_i)$$

- We do this using ***derivatives***



Gradient descent with one weight

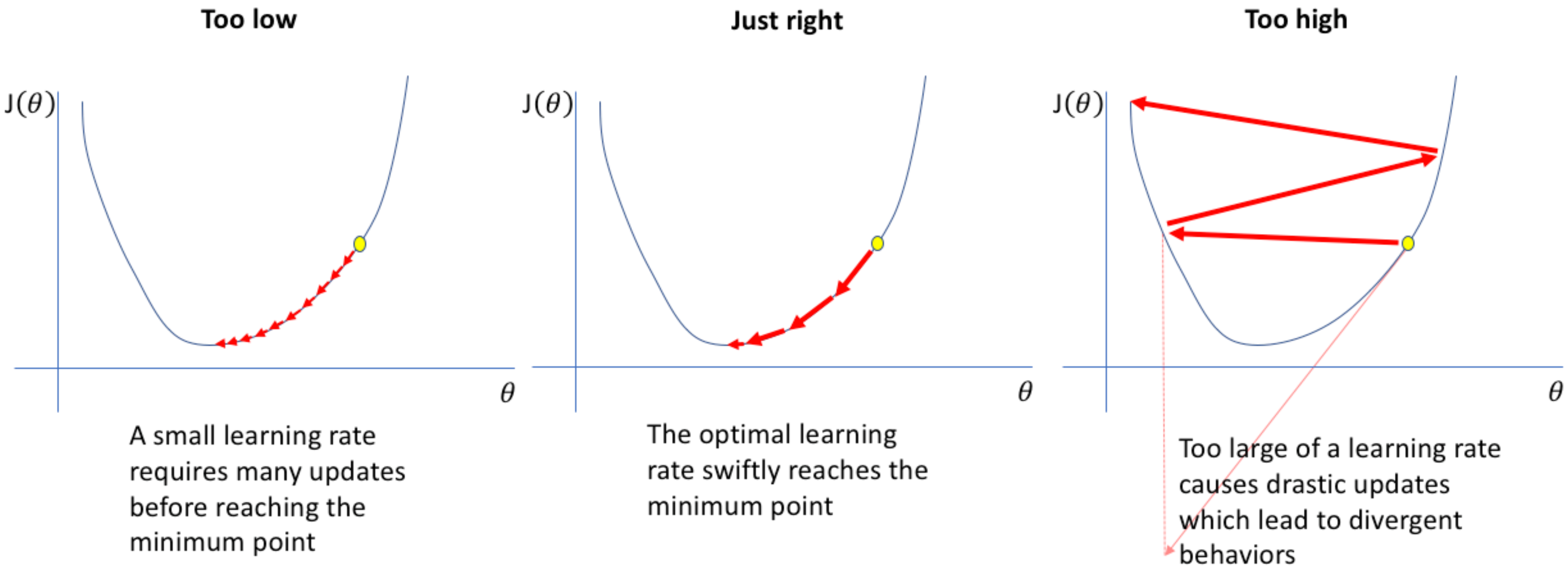
1. **Randomly** initialize weights at w_0
2. Compute the **derivative**
   "How to change the weight to decrease loss"
3. **Update the weight** according to the derivate (weigthed by the learning rate)
4. We repeat until we **converge** on the minimum

Derivative of the loss with respect to w

$$w_{new} = w_{old} - \boxed{\alpha}\boxed{\frac{d}{dw}L(f(x;w),y)}$$

Learning rate

Importance of Learning Rate



Modern optimizers update LR dynamically

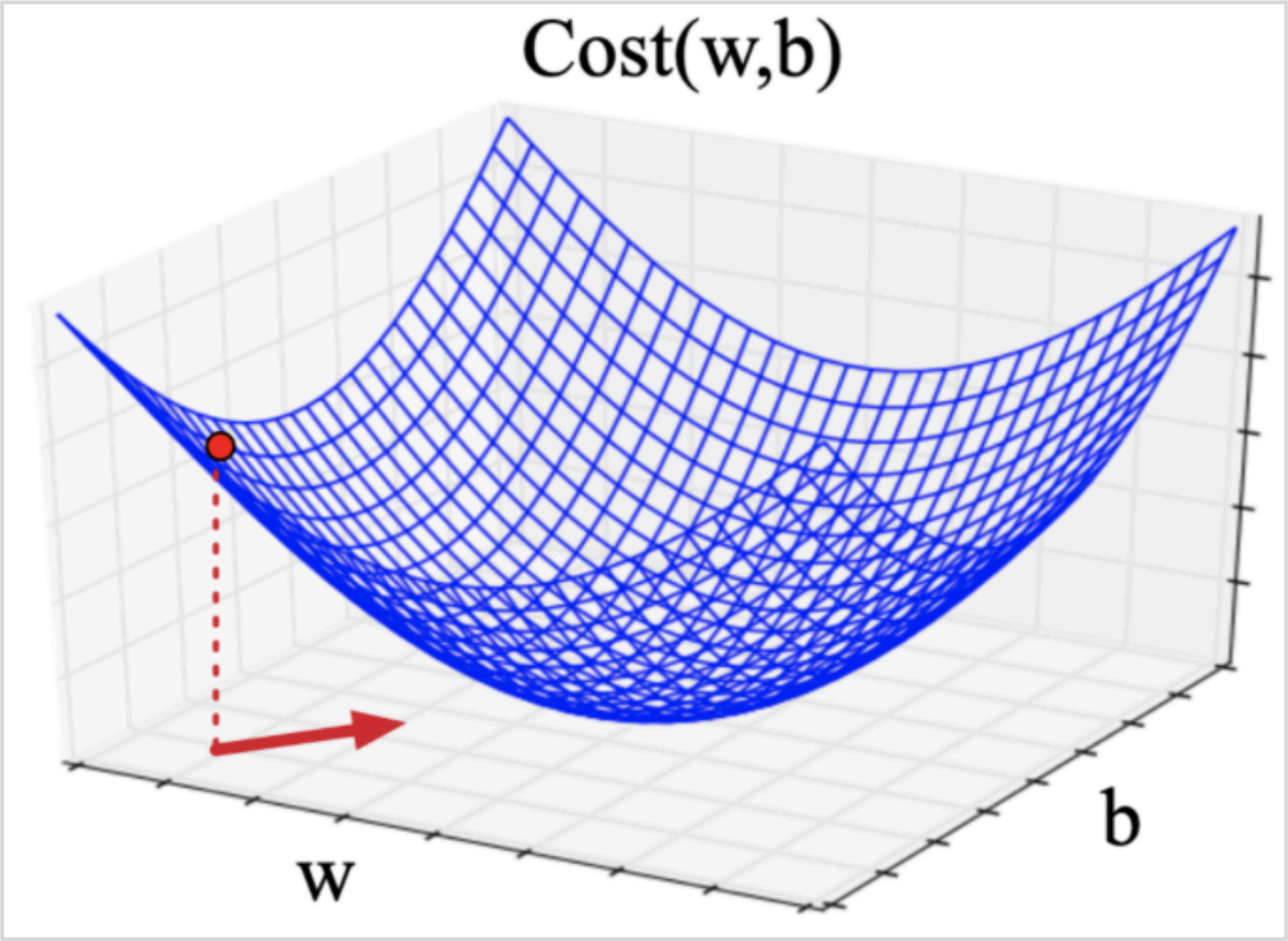
Source. https://www.jeremyjordan.me/nn-learning-rate/

Gradient descent with multiple parameters

- In practice we optimize multiple parameters
- *N*-dimensional space, where *N* is the number of parameters

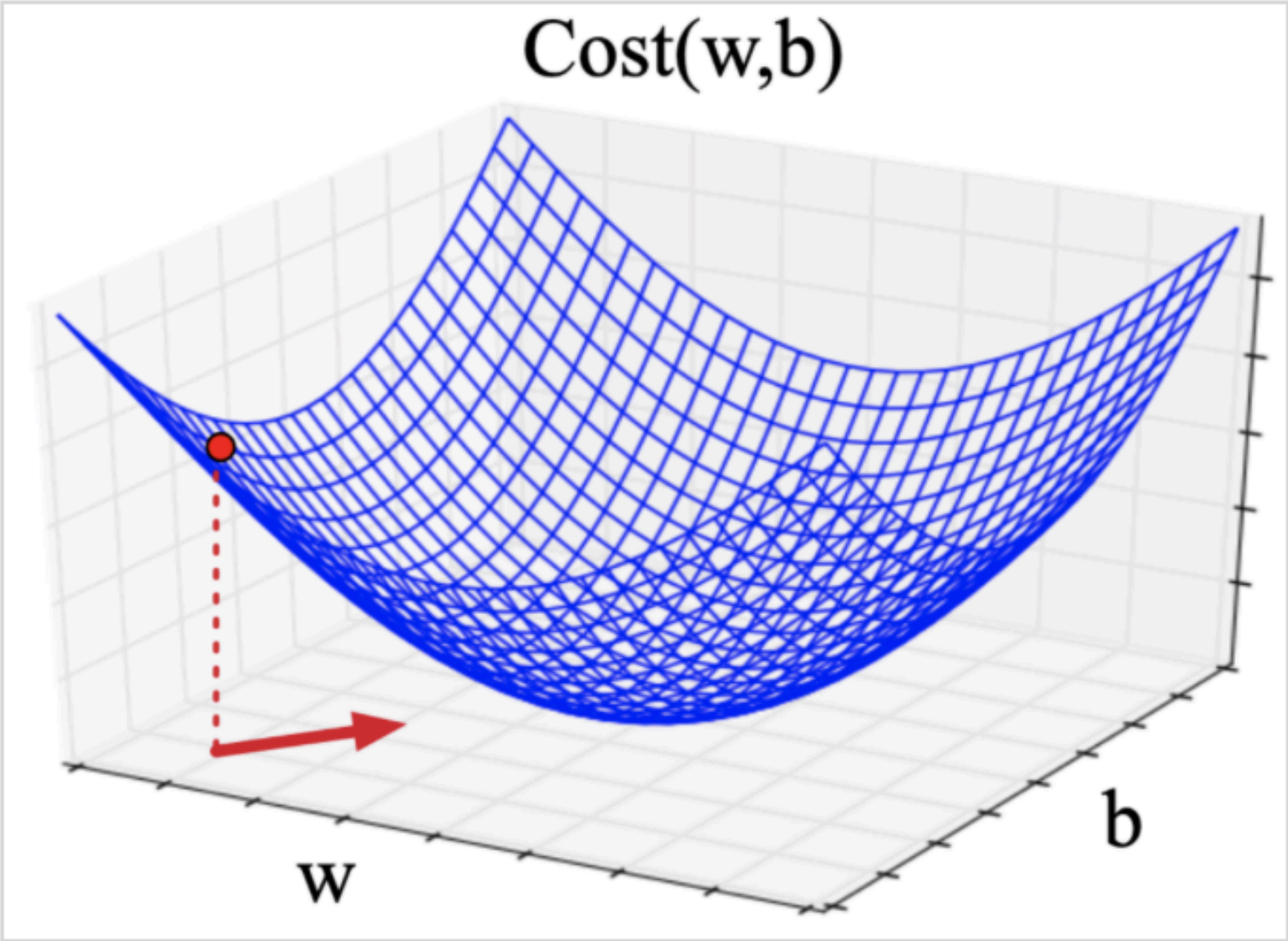
$$w_{new} = w_{old} - \alpha \boxed{\nabla L(f(x; w), y)}$$

Derivative with respect to all w



Stocastic Gradient Descent

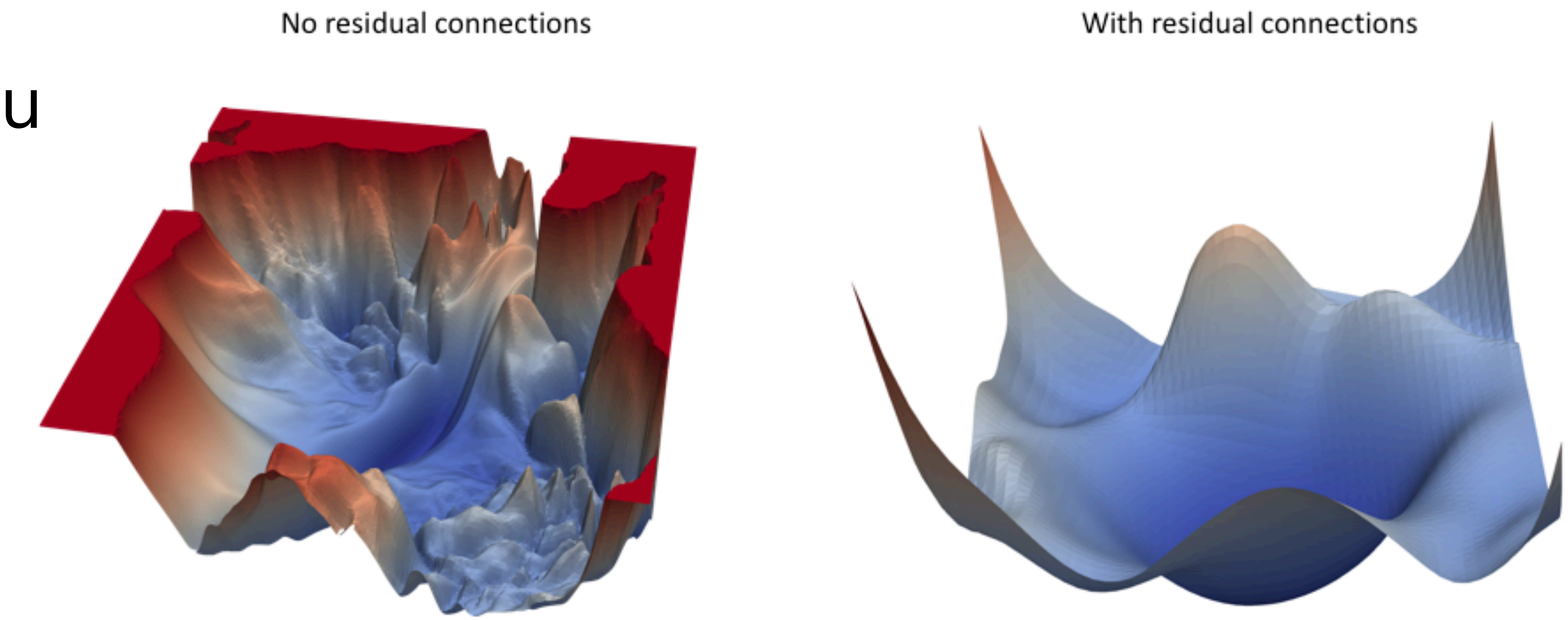
- Optimizing across all samples is expensive
  - Especially with large dataset
  - A solution is to calculate the gradient across *N samples*
  - we call *N* is the **batch size**
- We can compute the forward pass for each sample in **parellel**



Loss Landscape

- Not all functions are convex
- Low learning rates will get you stuck in a local minima

Loss landscape differ drastically based on how you set up your neural network



Same general network architecture

Sources & Notes

Source: https://www.cs.umd.edu/~tomg/projects/landscapes/?ref=jeremyjordan.me
Li et al., 2017, Visualizing the Loss Landscape of Neural Nets

CENTER FOR HUMANITIES COMPUTING

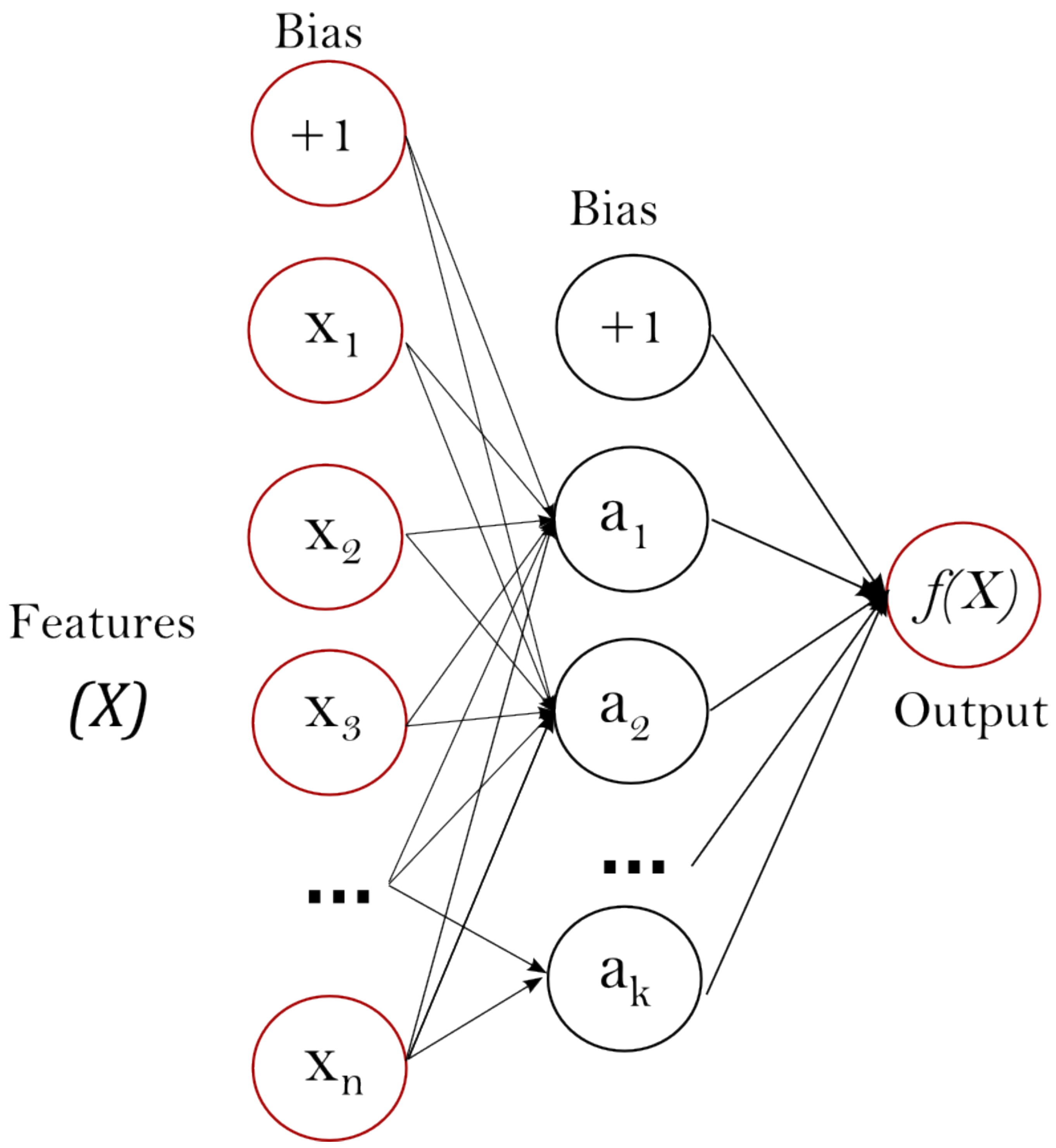
Backpropagation

How do we update across multiple layers?

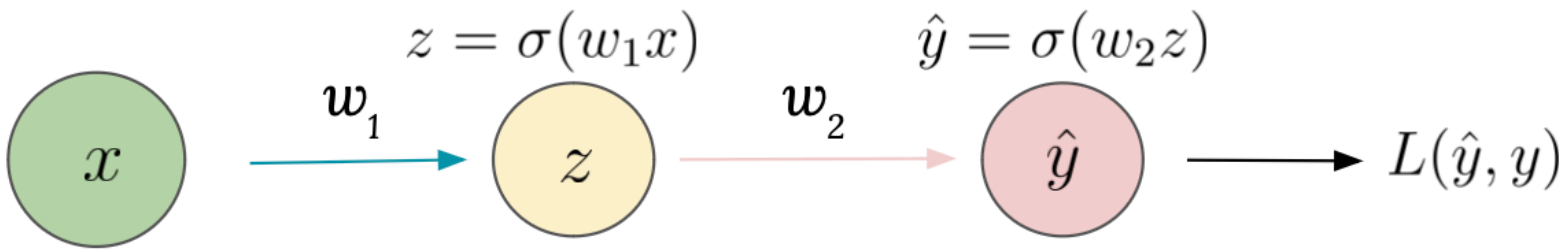
Chain rule!

$$h(x) = f(g(x))$$

$$h'(x) = f'(g(x))g'(x)$$

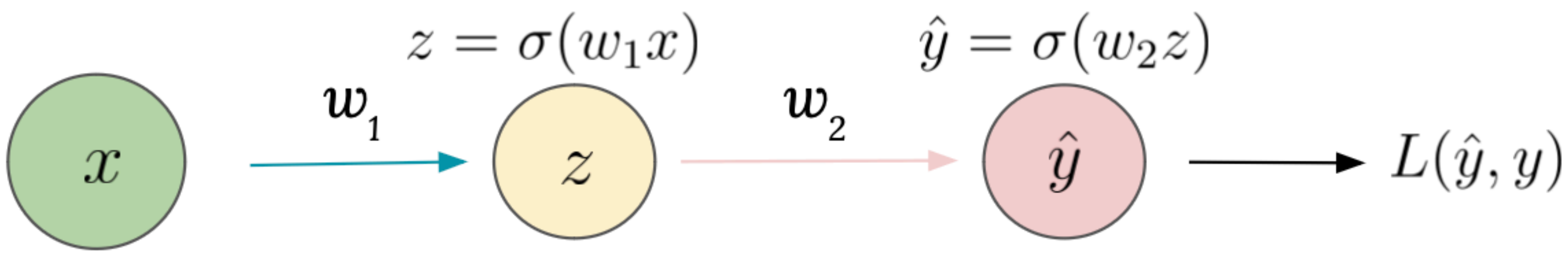


Chain Rule



by how much should update w_1 and w_2 to produce an output that yields a lower loss – i.e., what is the gradient for w_1?

Chain Rule

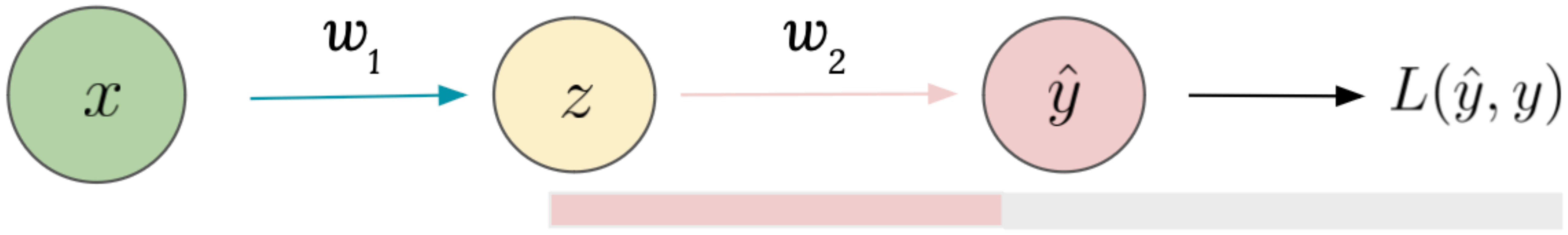


The derivative of the composition of multiple functions is the product of the partial derivatives of each of the functions. That is, if z = f(x), and y = g(z), which is equivalent to y = g(f(x)), then:

$$h'(x) = f'(g(x))g'(x) \quad \longleftrightarrow \quad \frac{\partial y}{\partial x} = \frac{\partial y}{\partial z} * \frac{\partial z}{\partial x}$$

Same thing

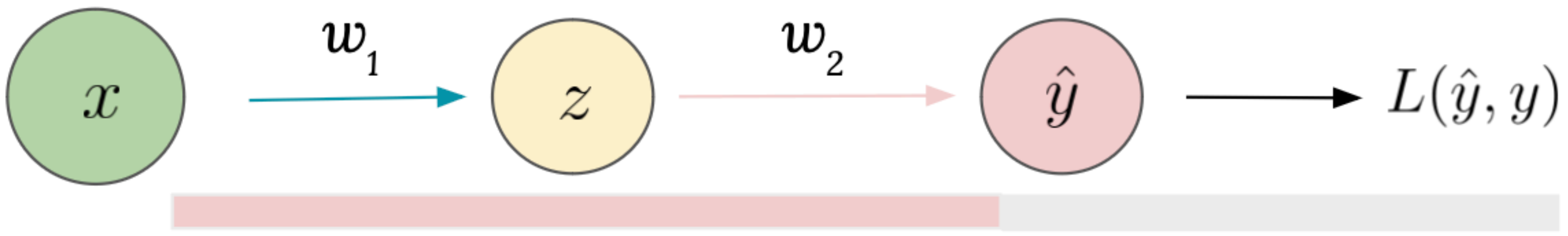
Chain Rule



In our case, this means that the gradient for w_2 is:

$$\frac{\partial L(\hat{y}, y)}{\partial w_2} = \frac{\partial L(\hat{y}, y)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

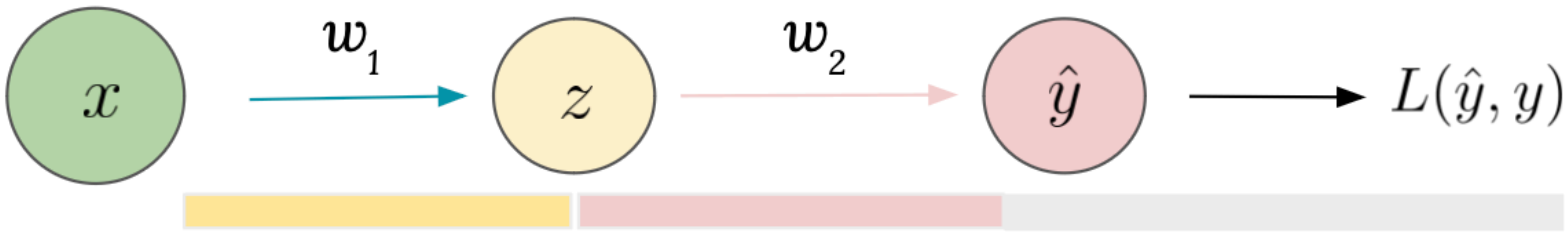
Chain Rule



The gradient for w_1 is, on the other hand:

$$\frac{\partial L(\hat{y}, y)}{\partial w_1} = \frac{\partial L(\hat{y}, y)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Chain Rule

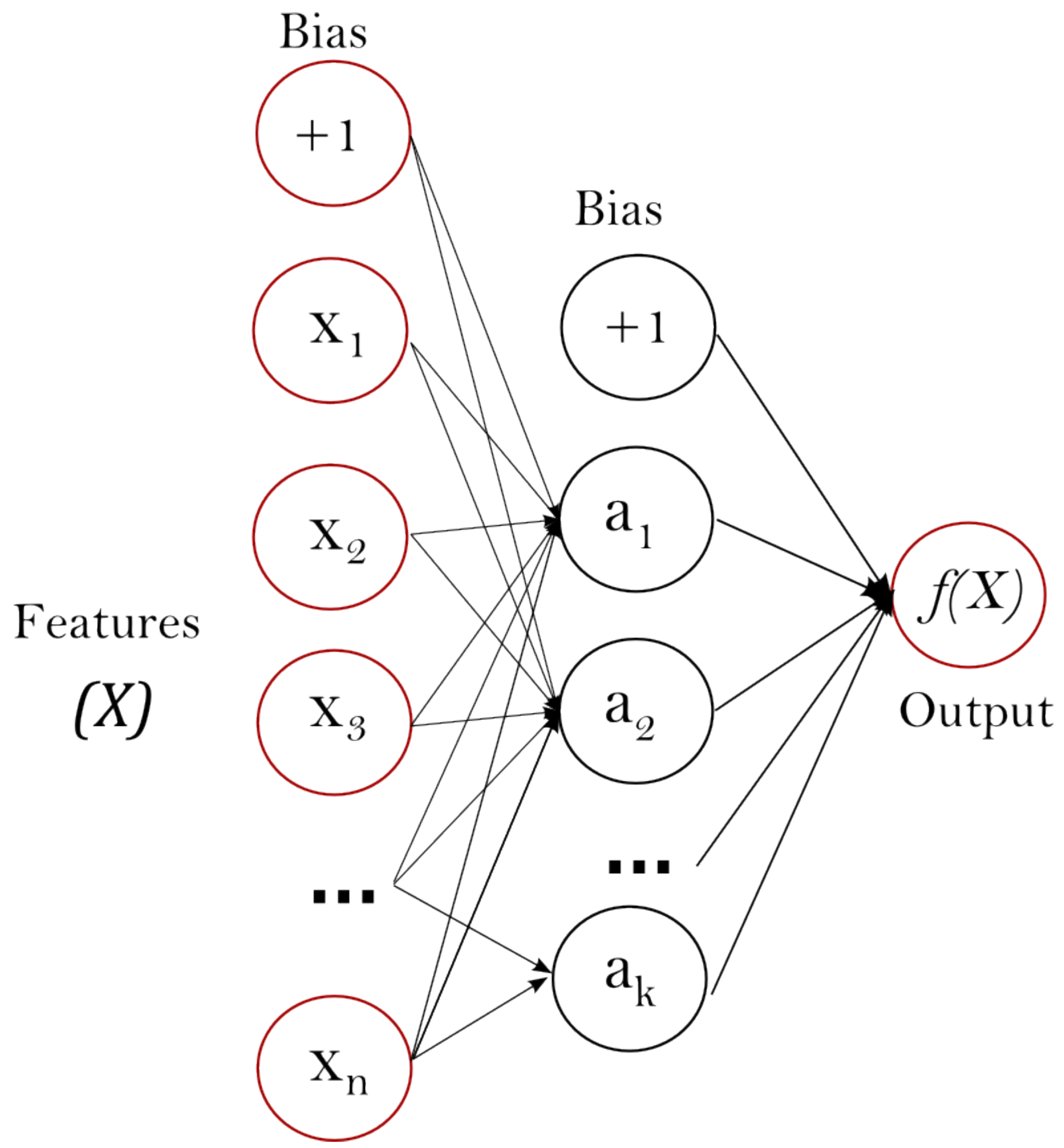


Applying the chain rule:

$$\frac{\partial L(\hat{y}, y)}{\partial w_1} = \frac{\partial L(\hat{y}, y)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial w_1}$$

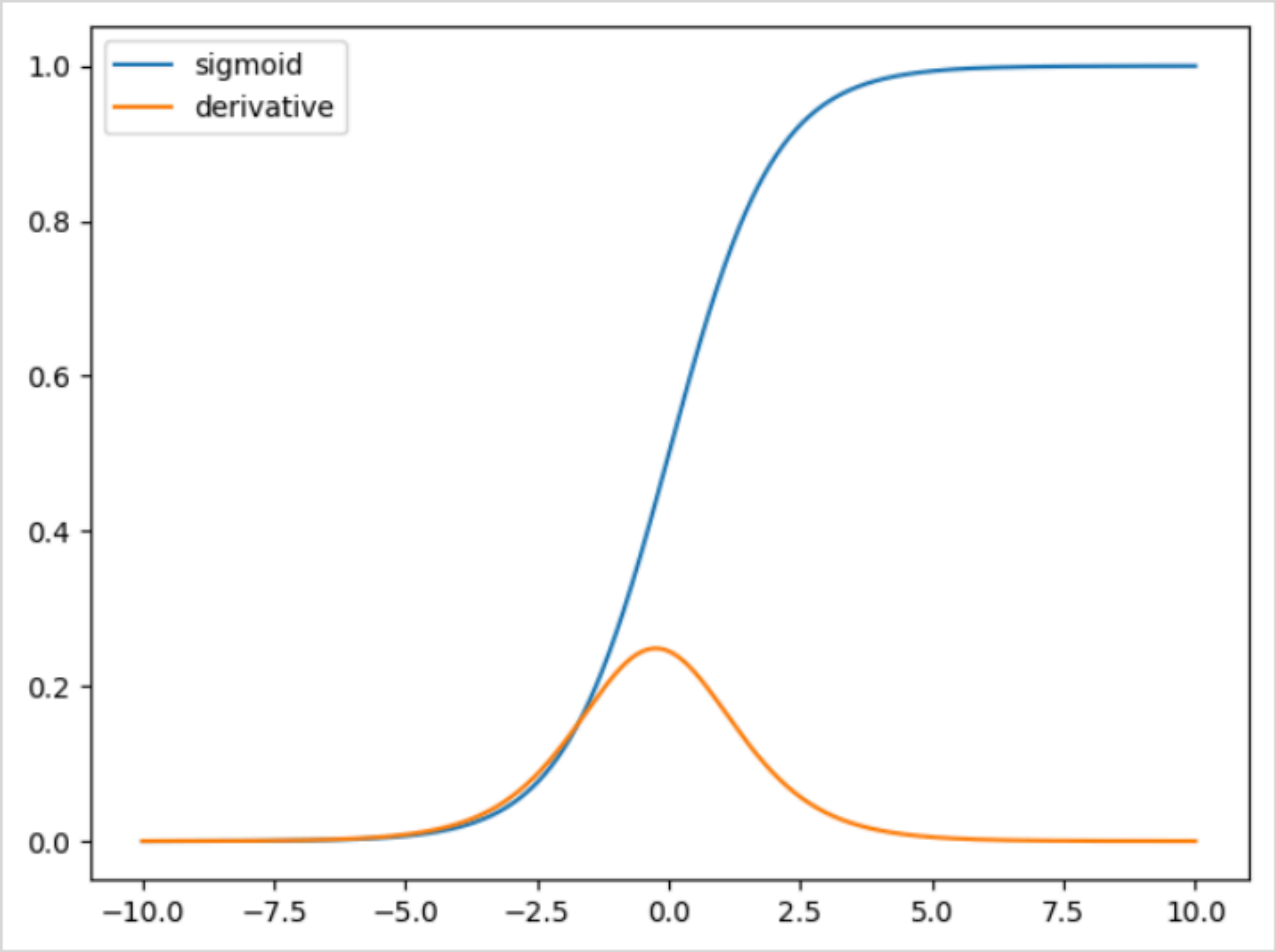
Backpropagation

- The process through which **errors** is being propegated throught the network is called **backpropagation**
- In practice uses Jacobian matrices rather single values
- Most packages implements automatic differentiation (autodiff)
- Overall workflow:
    - Forward pass
    - Compute loss
    - Backward pass (calculate gradient using chain rule)
    - Update weights



Activations Function (Again)

- Derivatives are good for parameters updated
- But derivatives of sigmoid is close to zero at the ends
  - Max is 0.25
- Given what we know about neural networks is this a problem?



Vanishing Gradient

- Gradient gets smaller across layers

Values below 1

$$\frac{\partial L(\hat{y}, y)}{\partial w_1} = \frac{\partial L(\hat{y}, y)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \boxed{\frac{\partial z}{\partial w_1}}$$

- Known as the **vanishing gradient problen**
- Potential solutions
  - Skip-connections
  - Activations functions

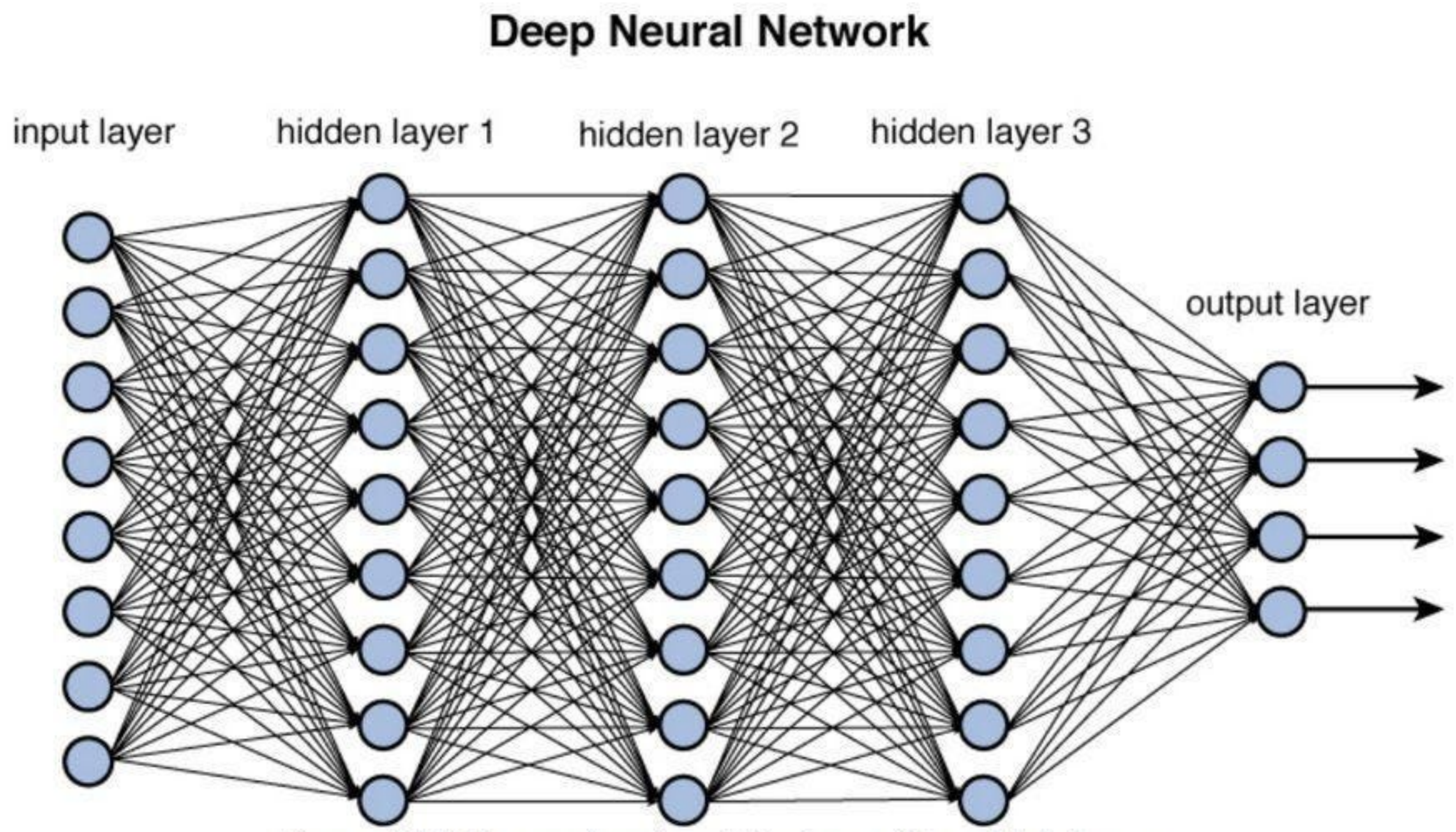
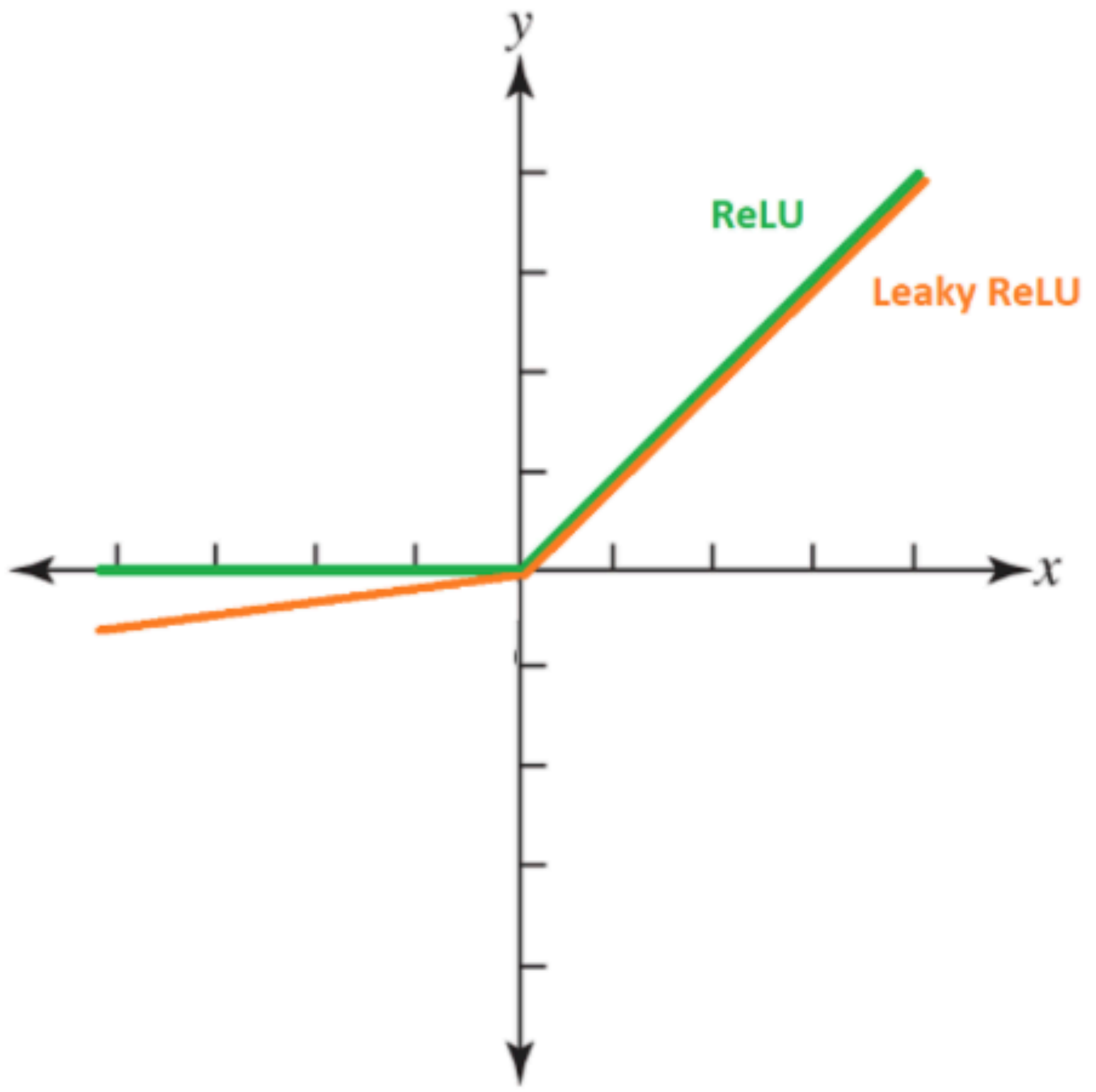


Figure 12.2 Deep network architecture with multiple layers.

Activation Function: **Rectified Linear Unit**

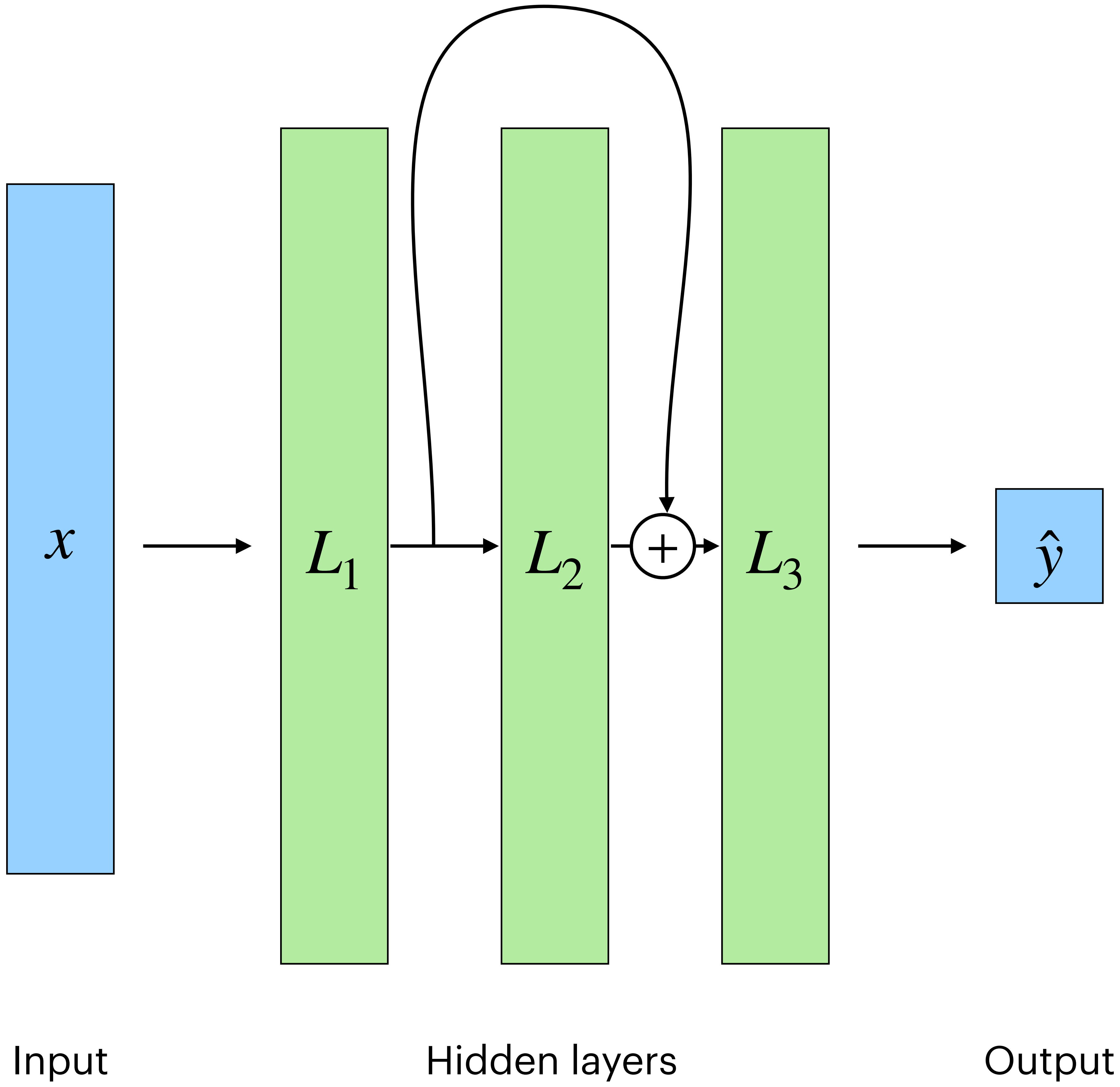
- Benefits
  - **Easy to differentiate**
  - Activation function add **non-linearity**
- Other alternatives such as swiGLU, geGLU, ….

$$ReLU = max(0, x)$$

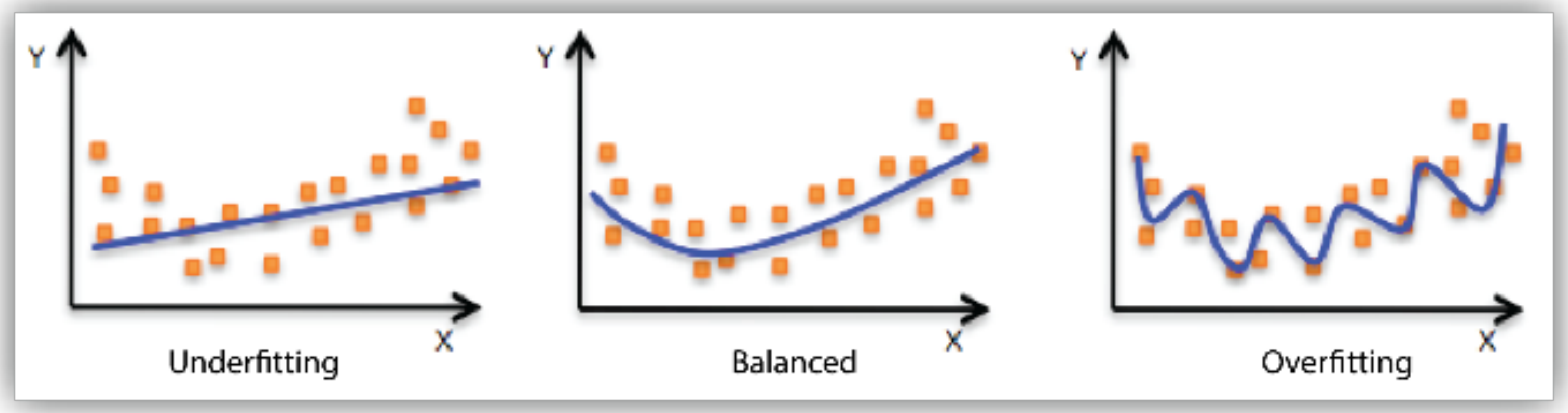


$$Leaky\ ReLU = f(x) = \begin{cases} x, & if\ x > 0 \\ 0.01x, & otherwise \end{cases}$$

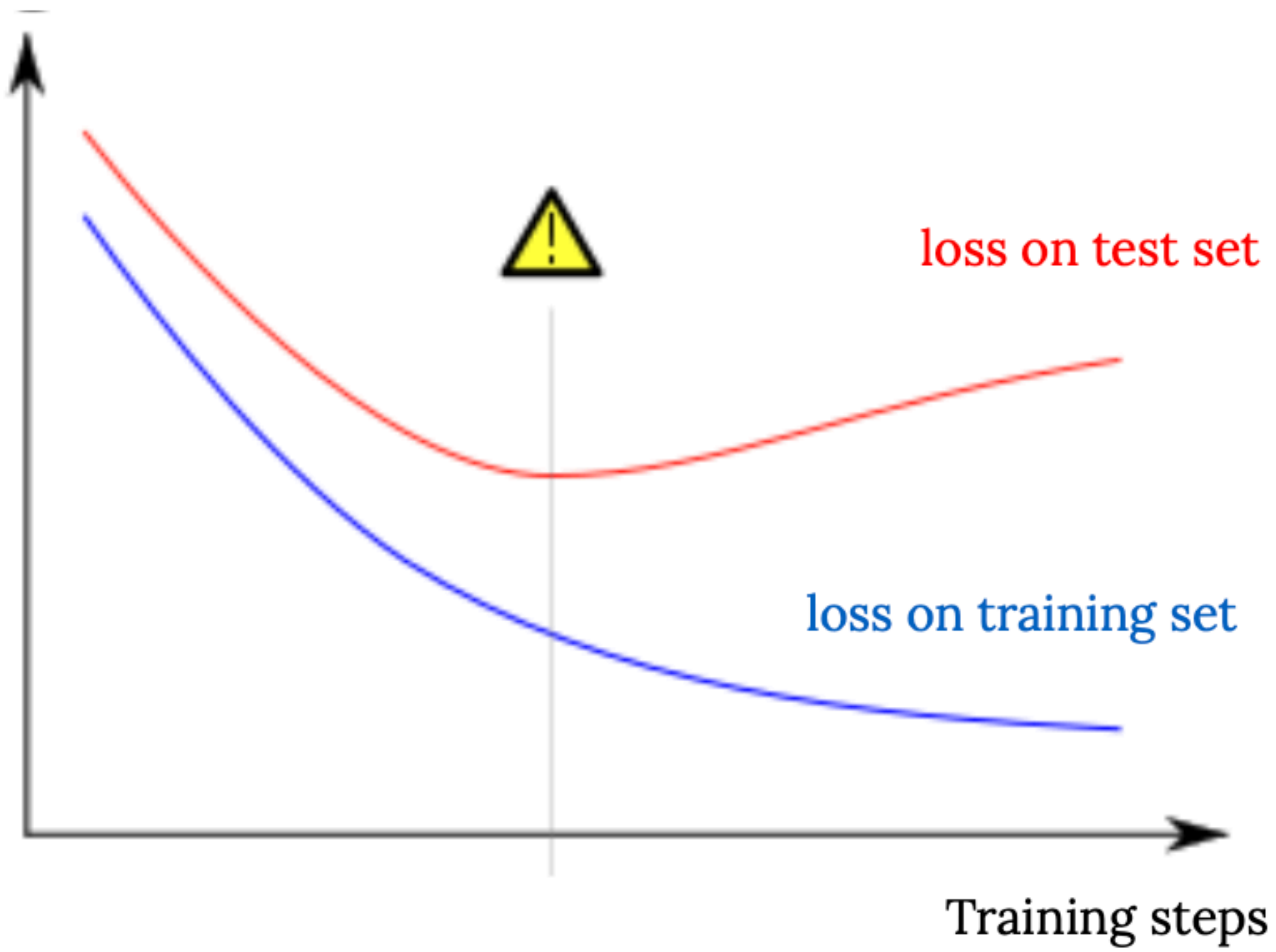
Skip connections



Overfitting



Overfitting

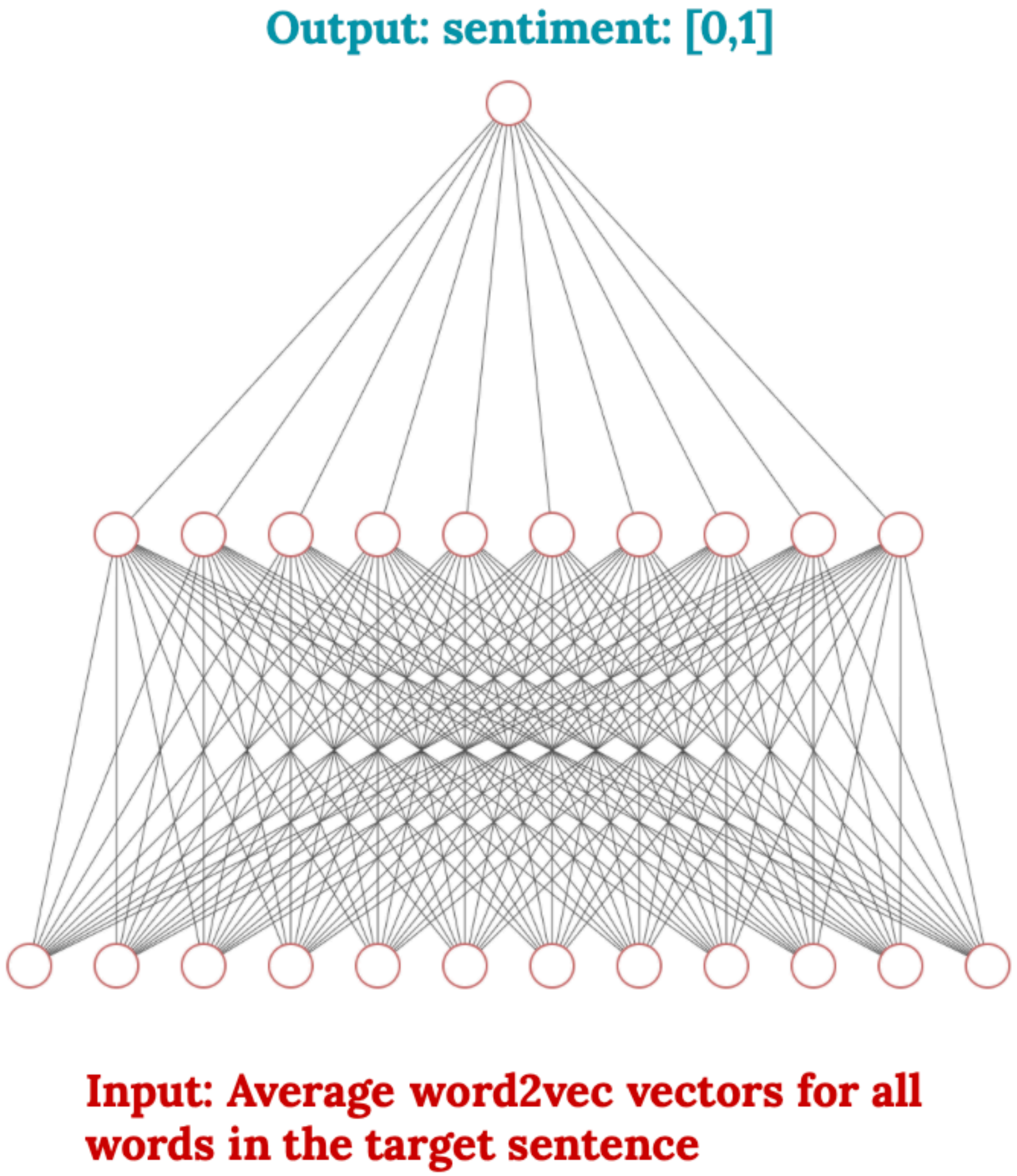


Example Architectures

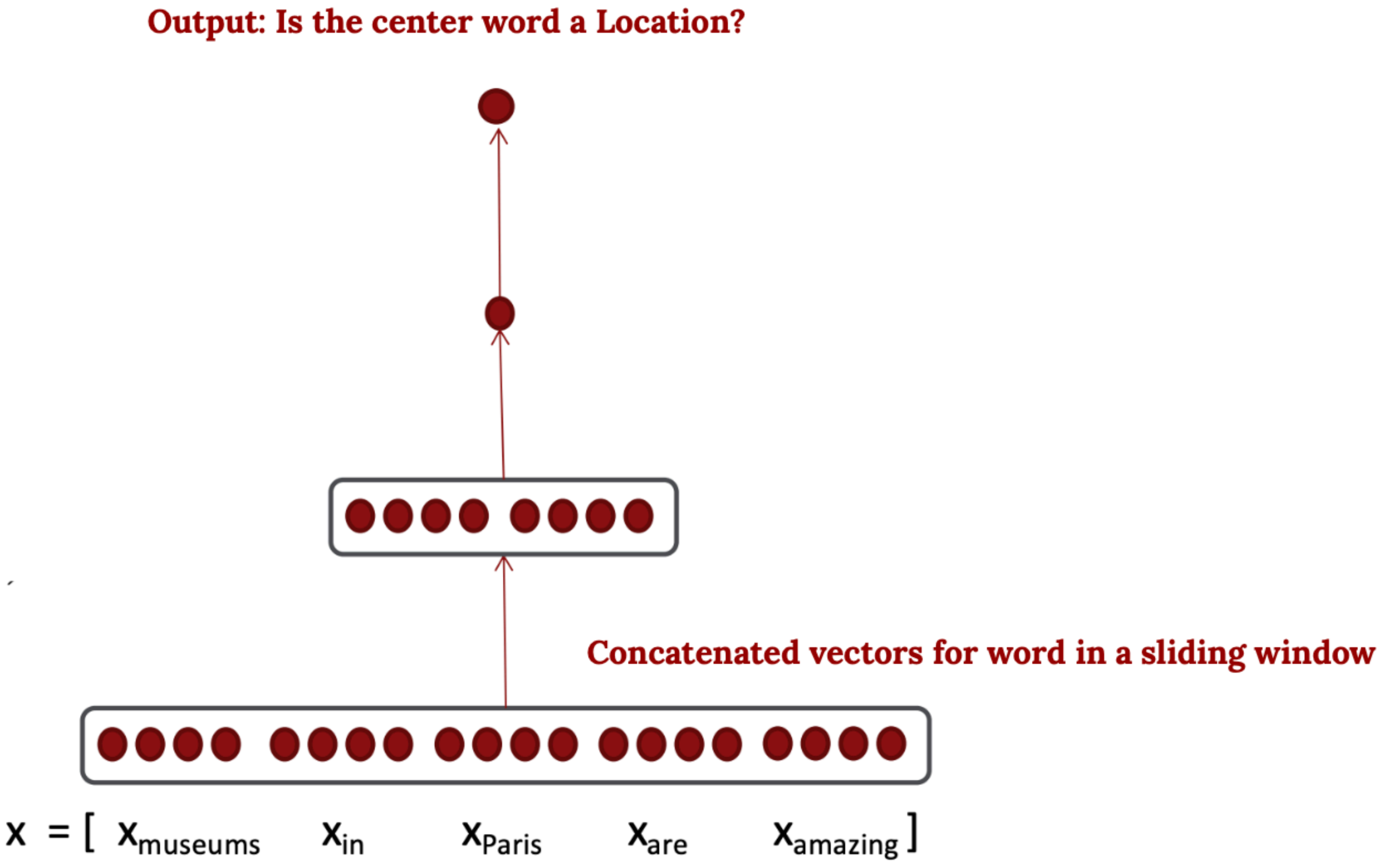
- A few simple ones

Simple Sentiment networks

- Bullet 1
- Bullet 2
- Bullet 3



Named Entities with Context



CBOW Word2Vec with Neural Network