LMP 1210H: Basic Principles of Machine Learning in Biomedical Research

Lecture 4: Neural networks

Sana Tonekaboni

February 1, 2024

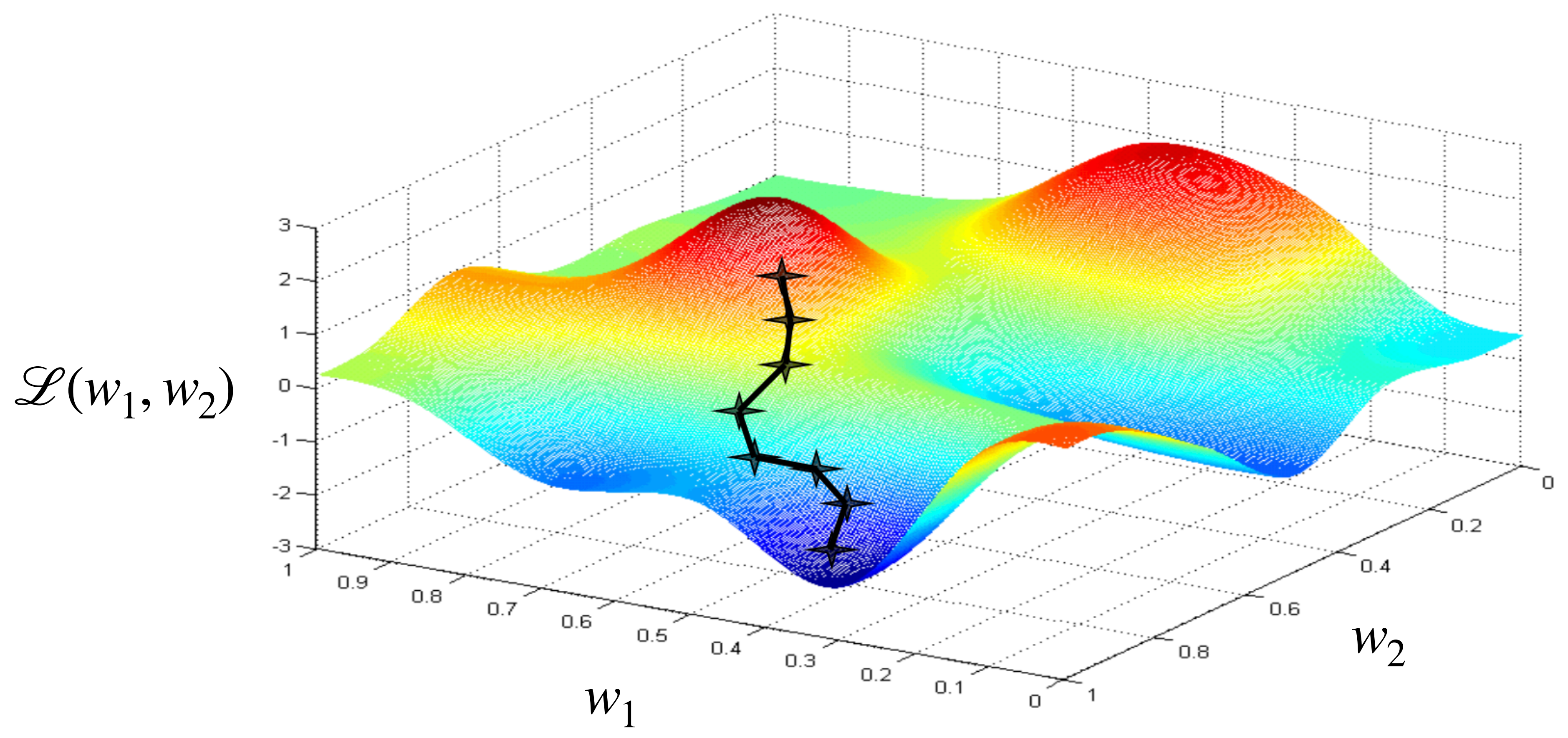
Quick check in…

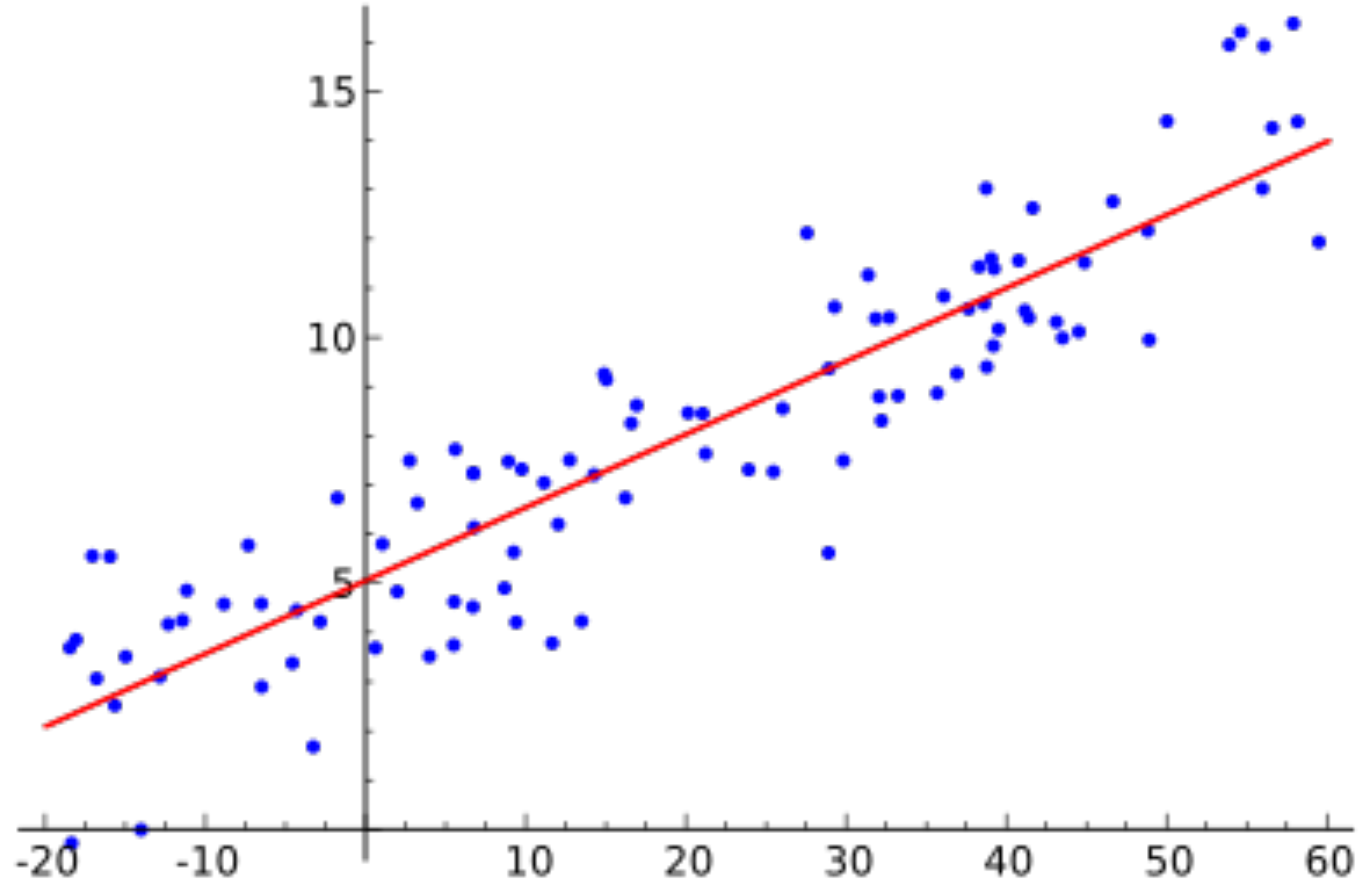
- Assignment 1 is due today.
- Assignment 2 will be released today.
- Project handout is now available on the course webpage.

Recap

- Linear models for regression.
- Loss function to quantify the quality of fit in a model.
- Regularization techniques to avoid overfitting.
- Optimization and gradient descent.

$$f(x) = \sum_j w_j x_j + b$$





Classification

Binary linear classification

- **Classification**: predicting a discrete-valued target.
- **Binary** classification: predicting a binary-valued target $t \in \{0,1\}$
  - Training examples with $t = 1$ are called positive examples, and training examples with $t = 0$ are called negative examples.
  - Examples: predict whether a patient has a disease, given the presence or absence of various symptoms.
- **Linear**: model is a linear function of $\mathbf{x}$, followed by a threshold:

$$z = \mathbf{w}^T\mathbf{x} + b$$

$$y = \begin{cases} 1 & \text{if } z \geq r \\ 0 & \text{if } z < r \end{cases}$$

$$\mathbf{w}^T\mathbf{x} + b \geq r \iff \mathbf{w}^T\mathbf{x} + \underbrace{b - r}_{\triangleq b'} \geq 0$$

$\xrightarrow{\hspace{3em}}$ Simplification

$$z = \mathbf{w}^T\mathbf{x} + b'$$

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Binary classification

Loss function

- Seemingly obvious loss function:

$$\mathcal{L}_{0-1}(y, t) = \begin{cases} 0 & \text{if } y = t \\ 1 & \text{if } y \neq t \end{cases}$$
$$= \mathbb{1}_{y \neq t}.$$

- The cost function will be the average loss over all samples, which is equivalent to error rate in binary cases.

$$\mathcal{J} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}_{y^{(i)} \neq t^{(i)}}$$

- But we can't optimize classification accuracy directly using gradient descent because it is discontinuous.

$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y} \frac{\partial y}{\partial w_j} \qquad y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Classification

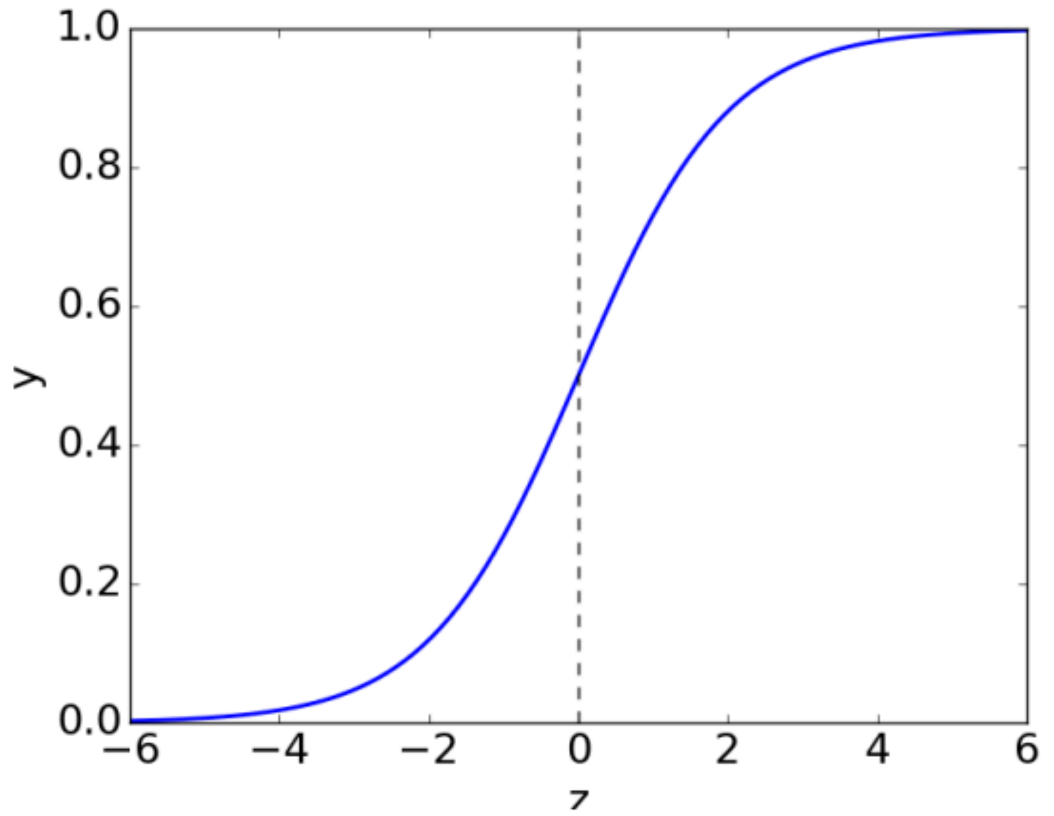
Logistic regression

- We typically define a **continuous surrogate loss function** which is easier to optimize.
- **Logistic regression** is a canonical example of this in a classification setting.
- The model outputs a continuous value $y \in [0,1]$ which you can think of as the probability of a sample being positive.

Logistic regression

Logistic function

- There is no reason to predict values outside $[0,1]$. Let's squash y into this interval then.
- The **logistic function** is a kind of **Sigmoidal** or S-shaped function:



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

- Used in this way, σ is called an **activation function**, and z is called the logit.

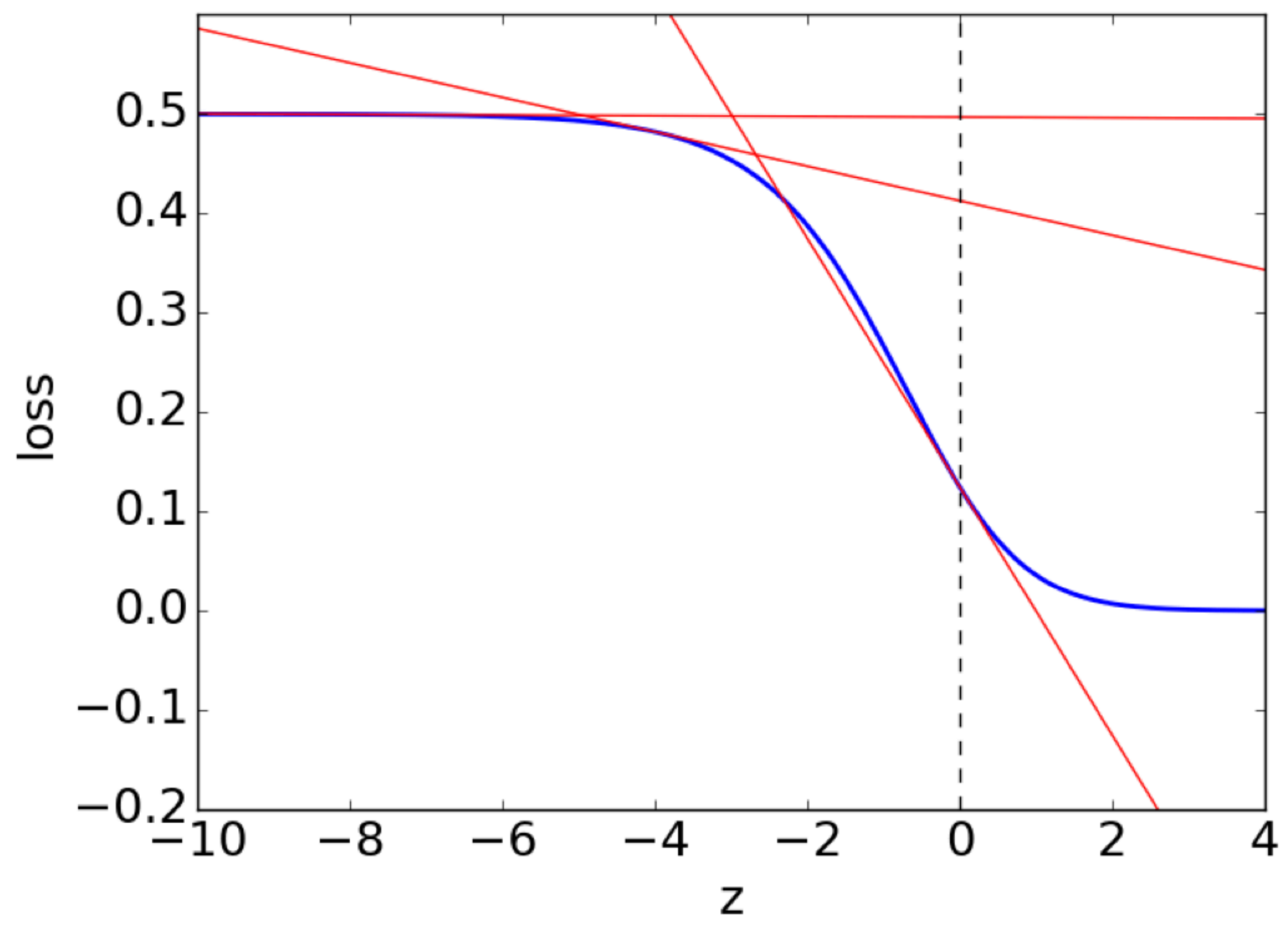
$$z = \mathbf{w}^\top \mathbf{x} + b$$

$$y = \sigma(z)$$

$$\mathcal{L}_{\mathrm{SE}}(y, t) = \frac{1}{2}(y - t)^2$$

Logistic regression
Loss function

- Can we use one of the losses that we knew for regression? What is the problem?



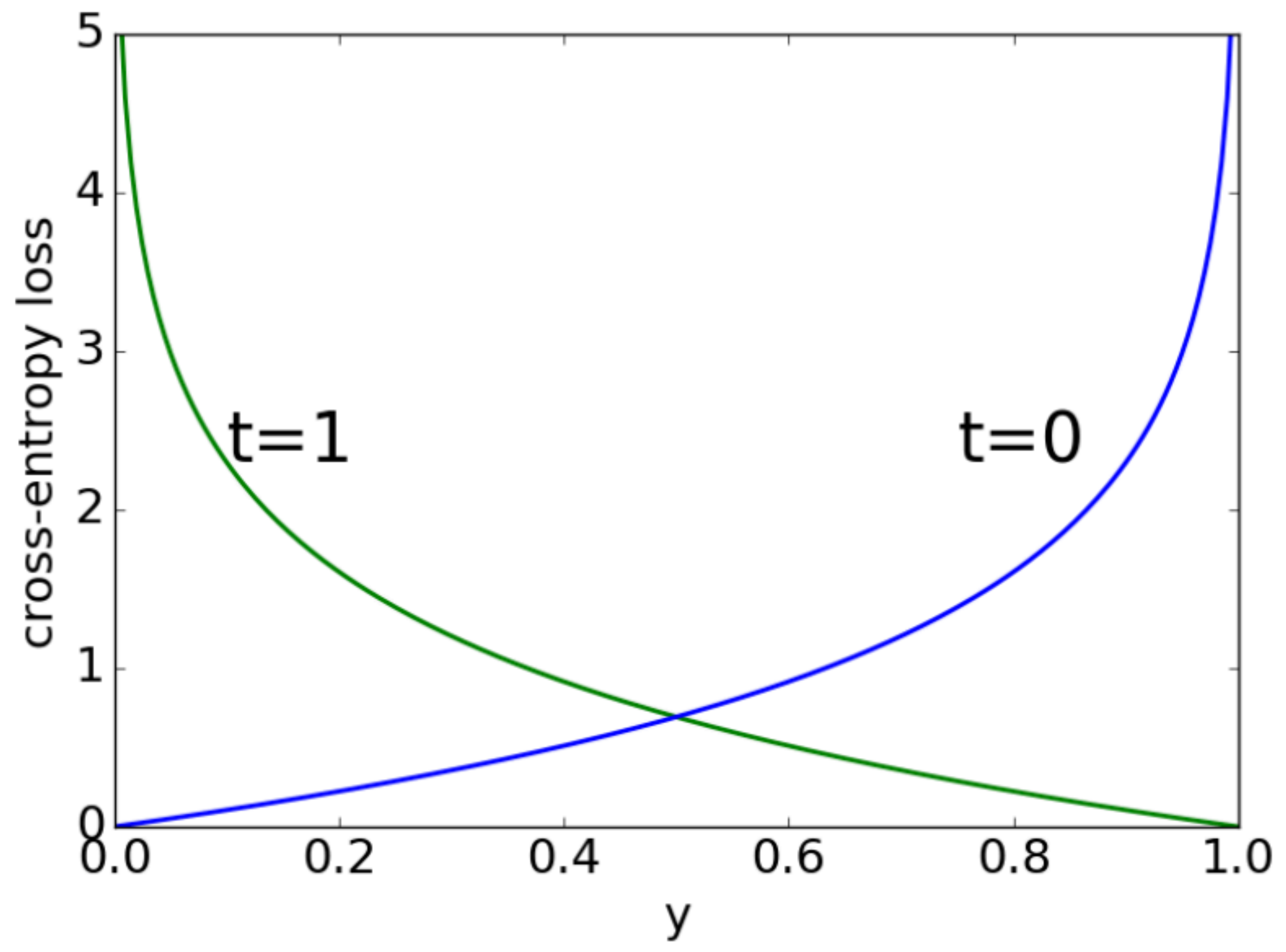
- Loss functions like MSE and MAE saturate at the extremes, meaning small gradient!

Logistic regression
Loss function

- Because y ∈ [0, 1], we can interpret it as the estimated probability that t = 1.
- What loss function do we define? What are some properties that we are interested in?
  - <u>Example:</u> A model that predicts with 99% confidence that a healthy patient is in risk of cancer is much more wrong than the one that predicts 80% confidence.

- Cross-entropy loss captures this intuition:

$$\mathcal{L}_{\mathrm{CE}}(y, t) = \begin{cases} -\log y & \text{if } t = 1 \\ -\log(1 - y) & \text{if } t = 0 \end{cases}$$
$$= -t \log y - (1 - t) \log(1 - y)$$

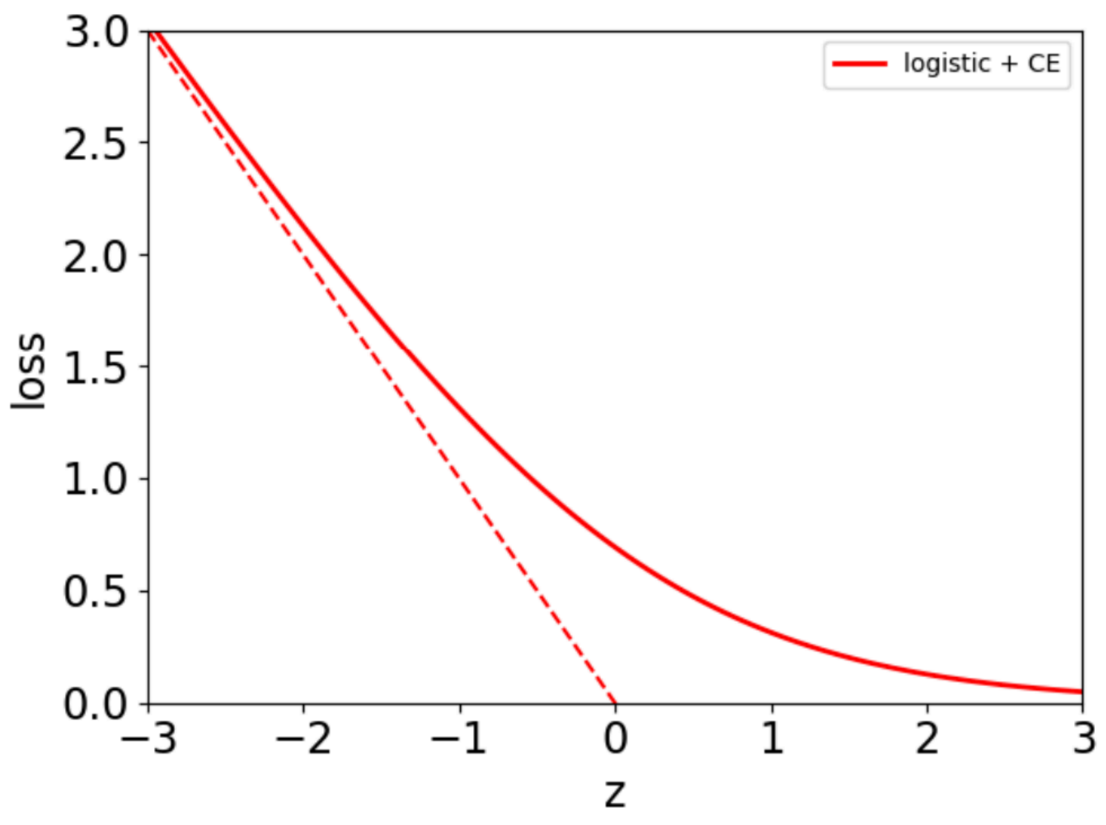


Logistic regression

- Logistic regression combines the **logistic activation function** with a **cross-entropy loss**.

$$
\begin{aligned}
z &= \mathbf{w}^\top \mathbf{x} + b \\
y &= \sigma(z) \\
&= \frac{1}{1 + e^{-z}} \\
\mathcal{L}_{\mathrm{CE}} &= -t \log y - (1 - t) \log(1 - y)
\end{aligned}
$$

- Interestingly, the loss asymptotes to a linear function of the logit z. (full derivation in the readings)



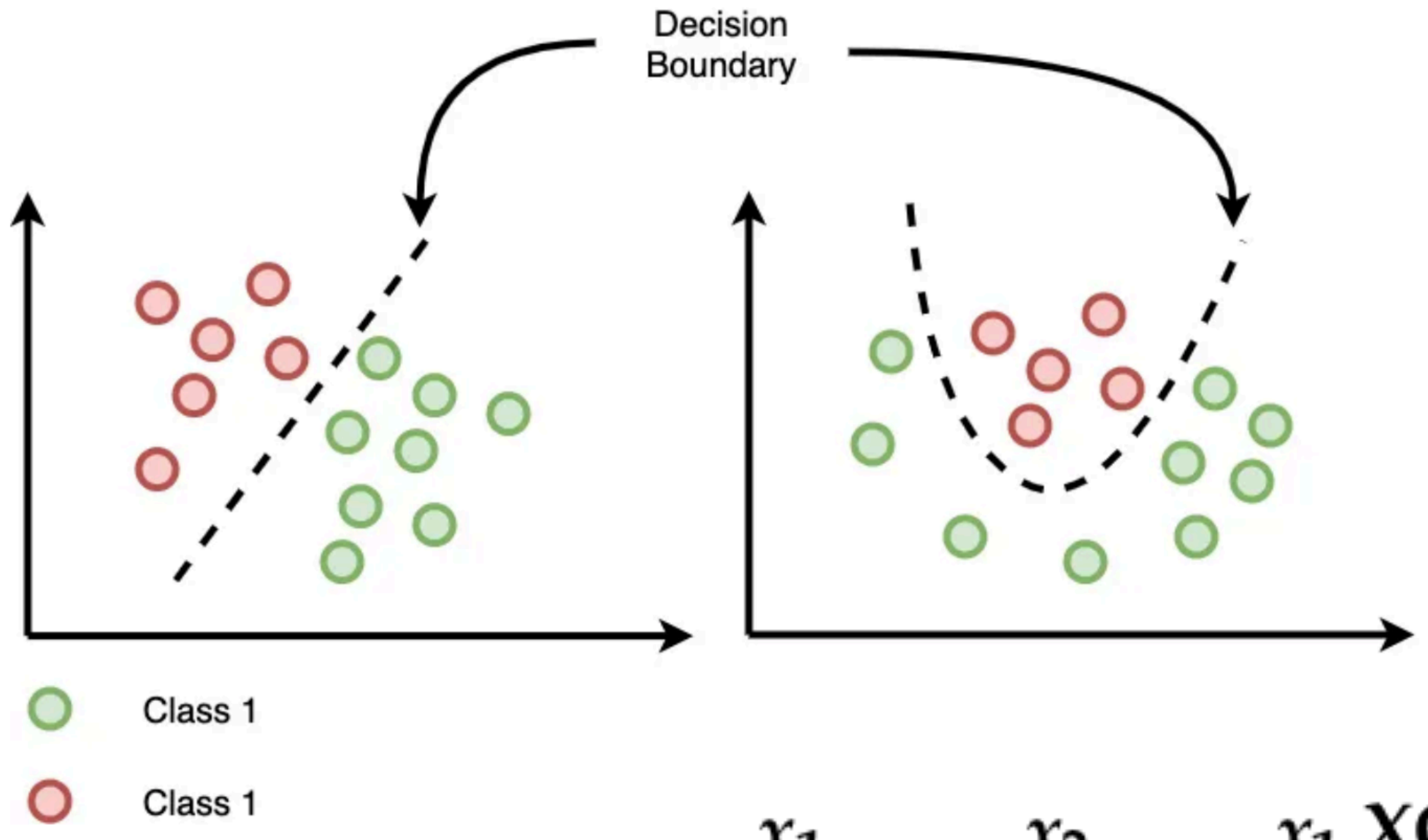
Linear Regression vs Logistic Regression

Linear Regression	Logistic Regression
Mostly used for continuous regression	Mostly used for binary classification
Loss function: Squared error	Loss function: Cross entropy
Optimization: Gradient descend or closed form	Optimization: Gradient descend
Output is linear in inputs	Output is not linear in inputs

Linear classification

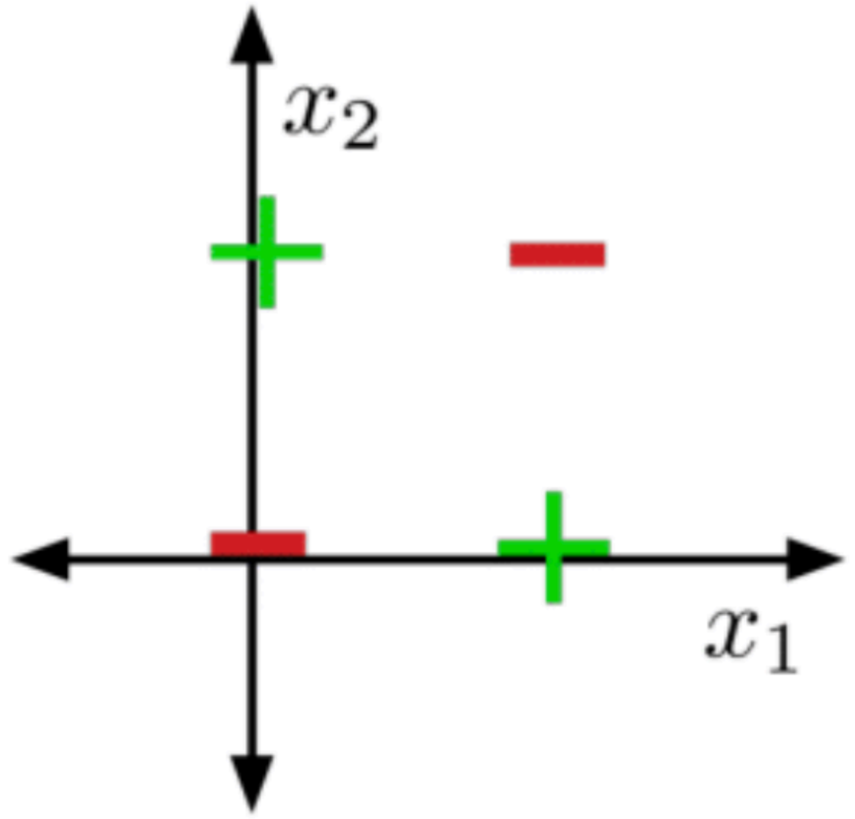
Limitations

- Linear classifiers are very limited in expressive power.



- XOR is a classic example of a simple function that is not linearly separable.

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0



Linear classification

Limitations

- Sometime we can overcome this limitation using **feature maps**.

$$\psi(\mathbf{x}) = \begin{pmatrix} x_1 \\ x_2 \\ x_1 x_2 \end{pmatrix}$$

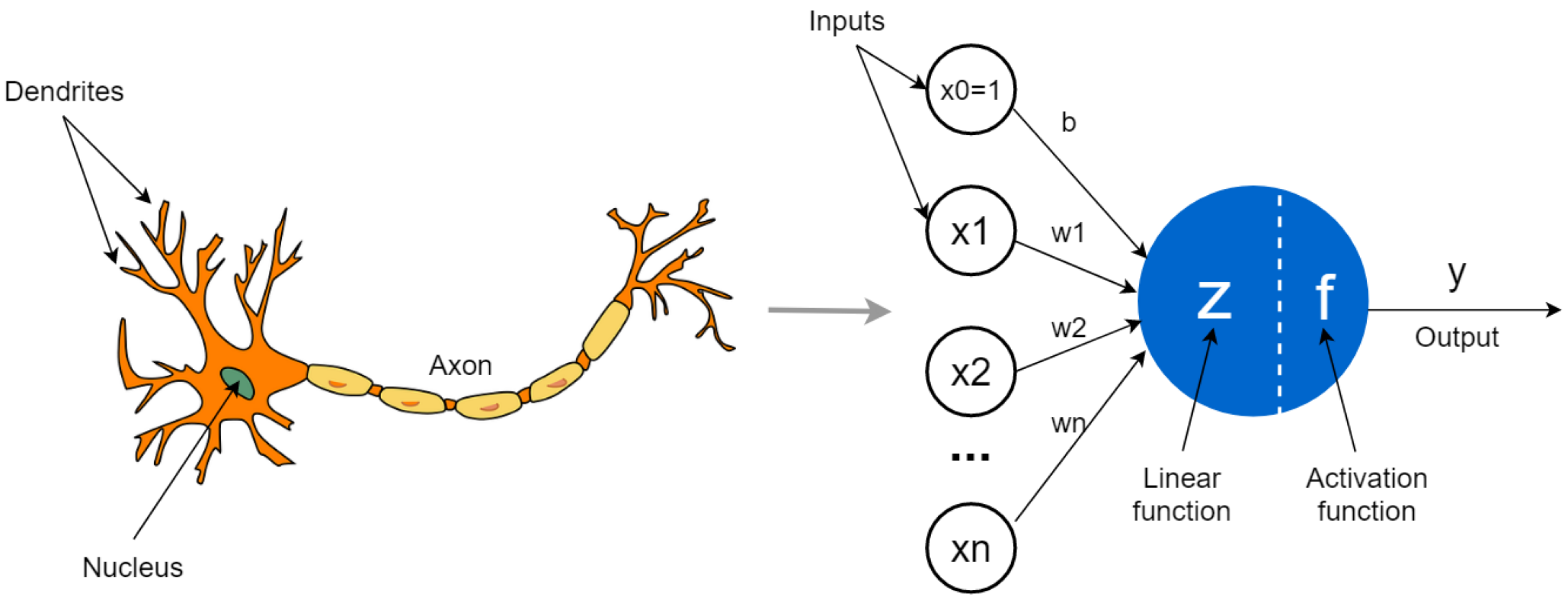
x_1	x_2	$\phi_1(\mathbf{x})$	$\phi_2(\mathbf{x})$	$\phi_3(\mathbf{x})$	t
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	0	0	1
1	1	1	1	1	0

- This is now linearly separable.
- But this cannot be a general solution. Why?
    - Hard to know the right mapping!

Neural networks

How can we model more complex functions?

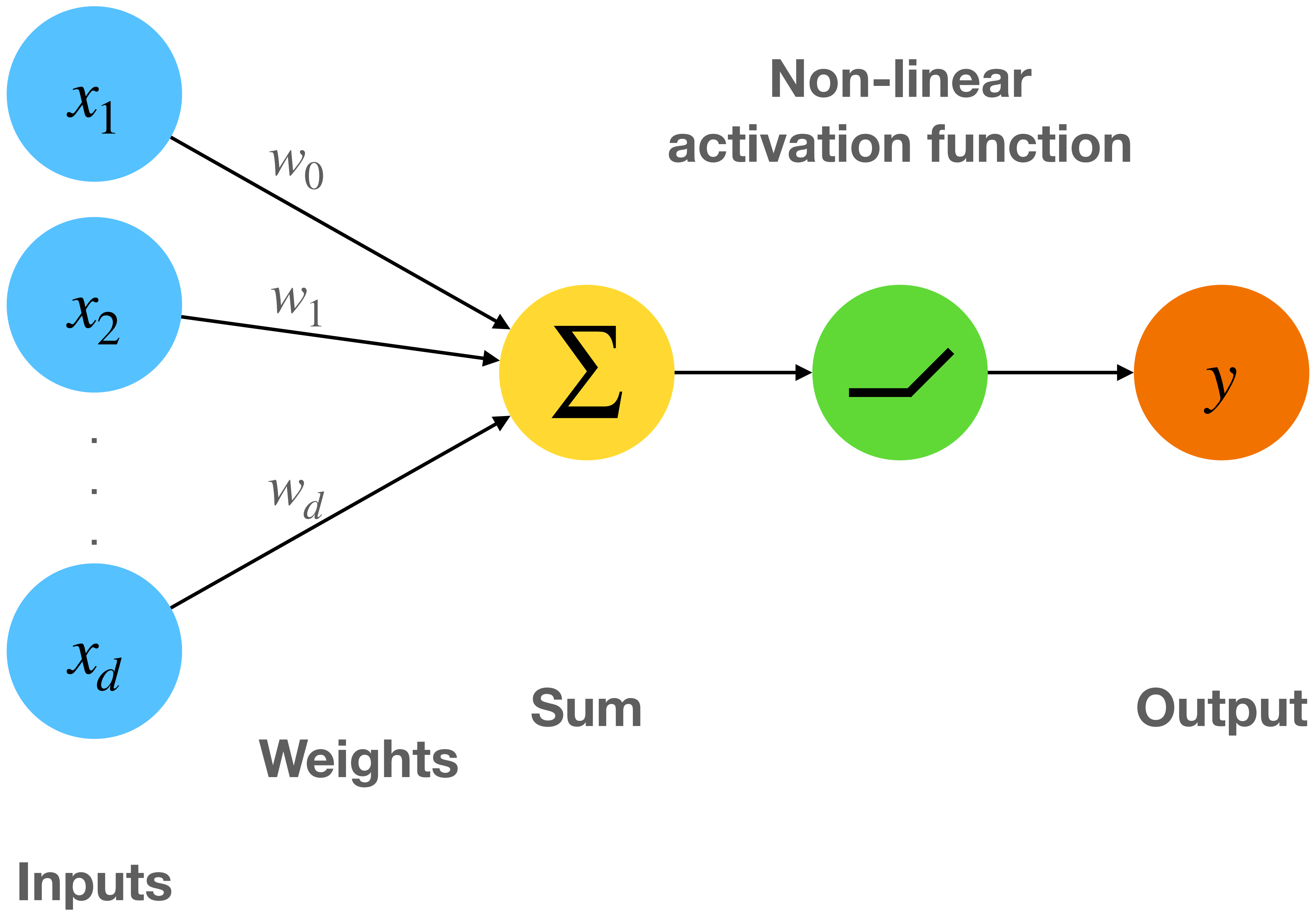
Idea of neural networks inspired from human brain.



https://towardsdatascience.com/the-concept-of-artificial-neurons-perceptrons-in-neural-networks-fab22249cbfc

Neural networks (Multilayer Perceptron)

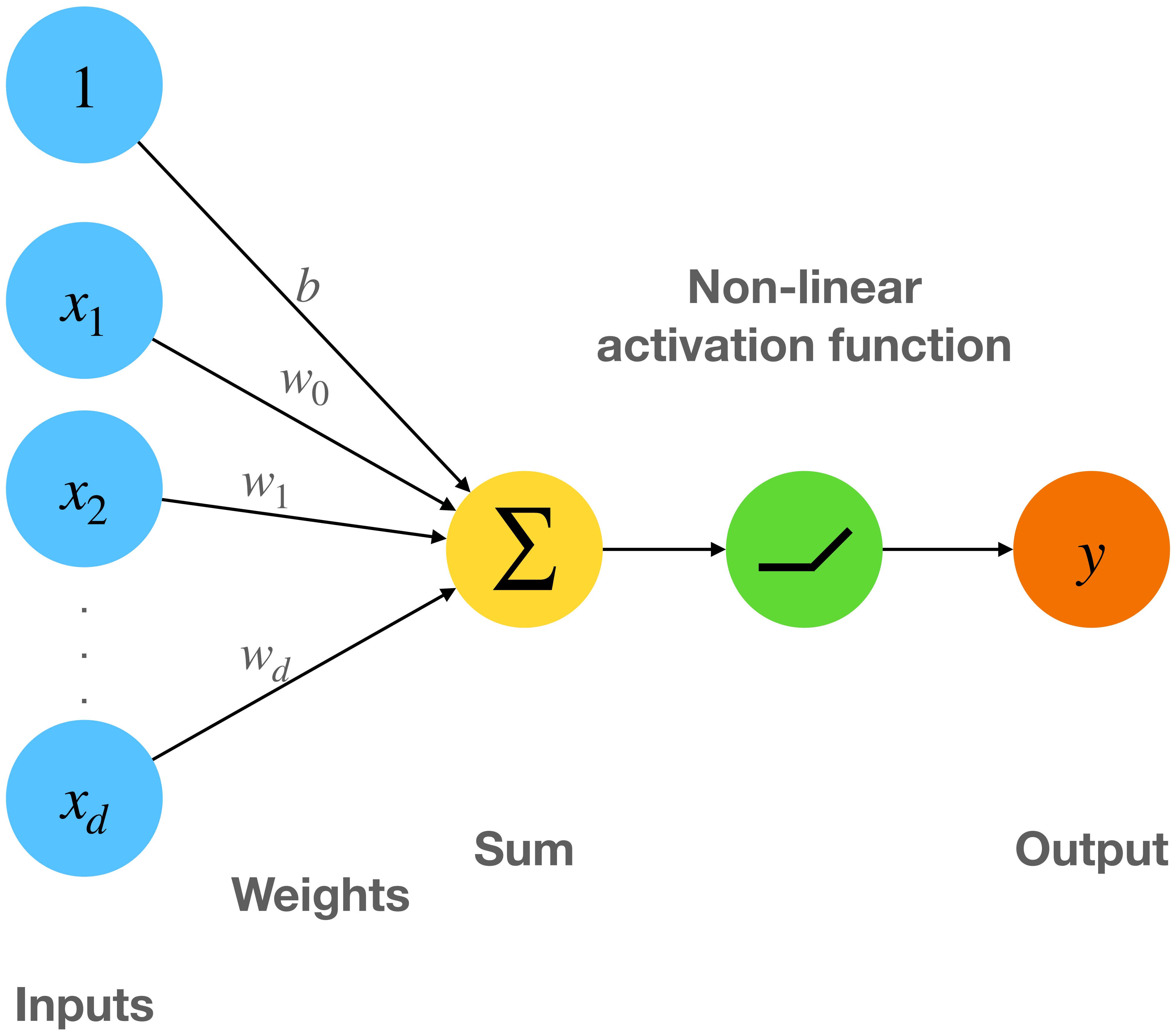
Perceptron



$$y = a(\sum_j^d w_j x_j)$$

Neural networks (Multilayer Perceptron)

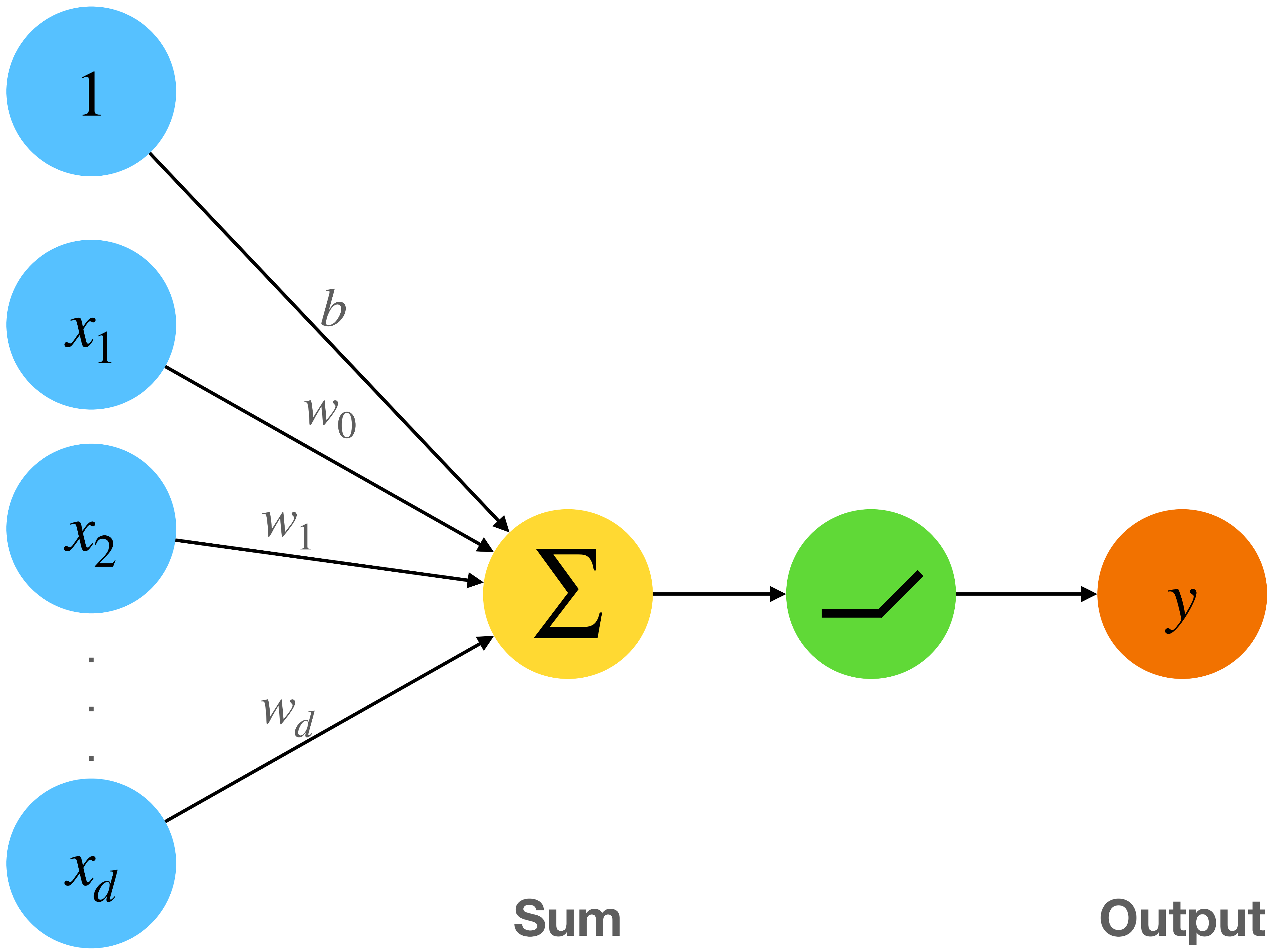
Perceptron

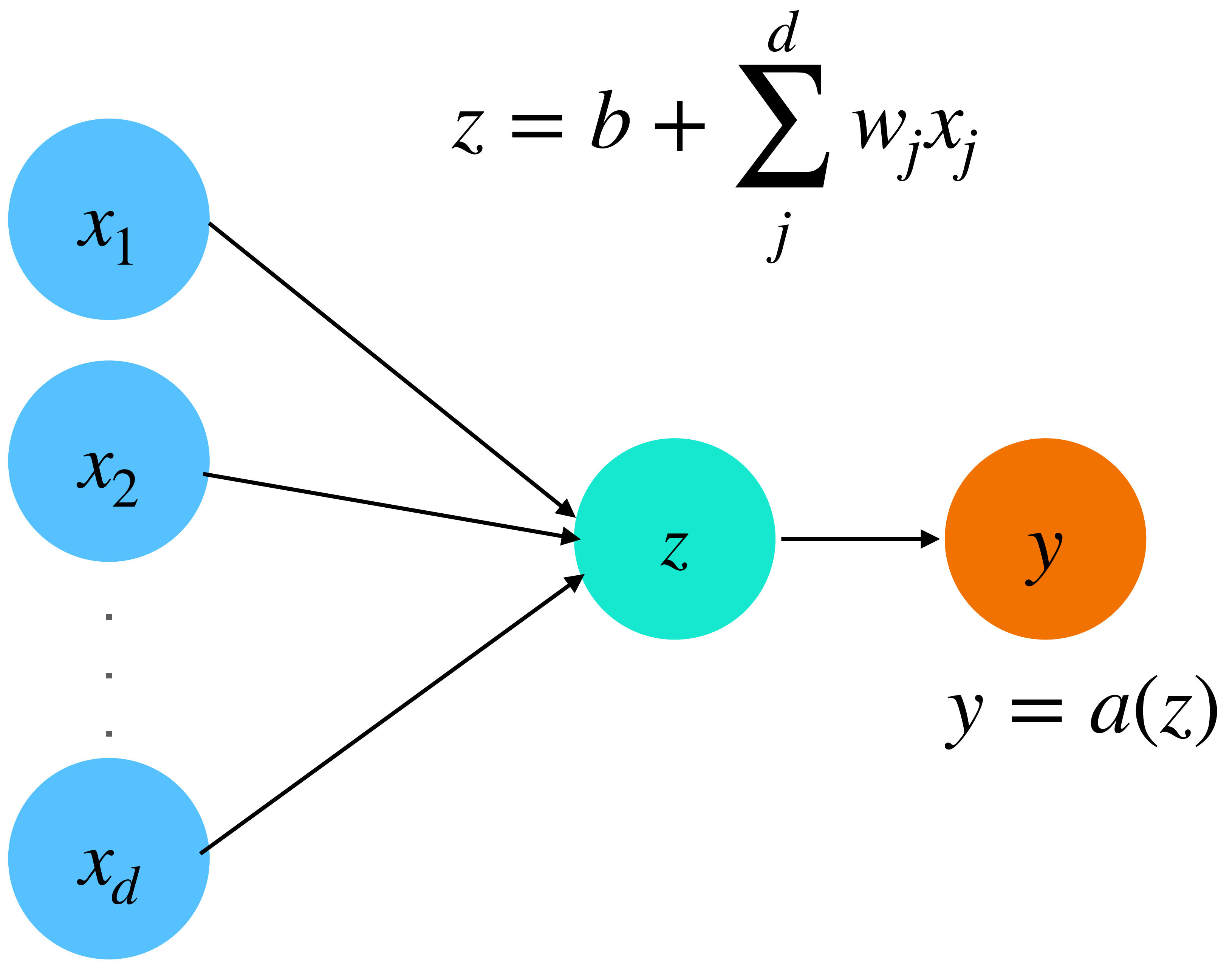


$$y = a\left(\sum_{j}^{d} w_j x_j + b\right)$$

Neural networks (Multilayer Perceptron)

Perceptron

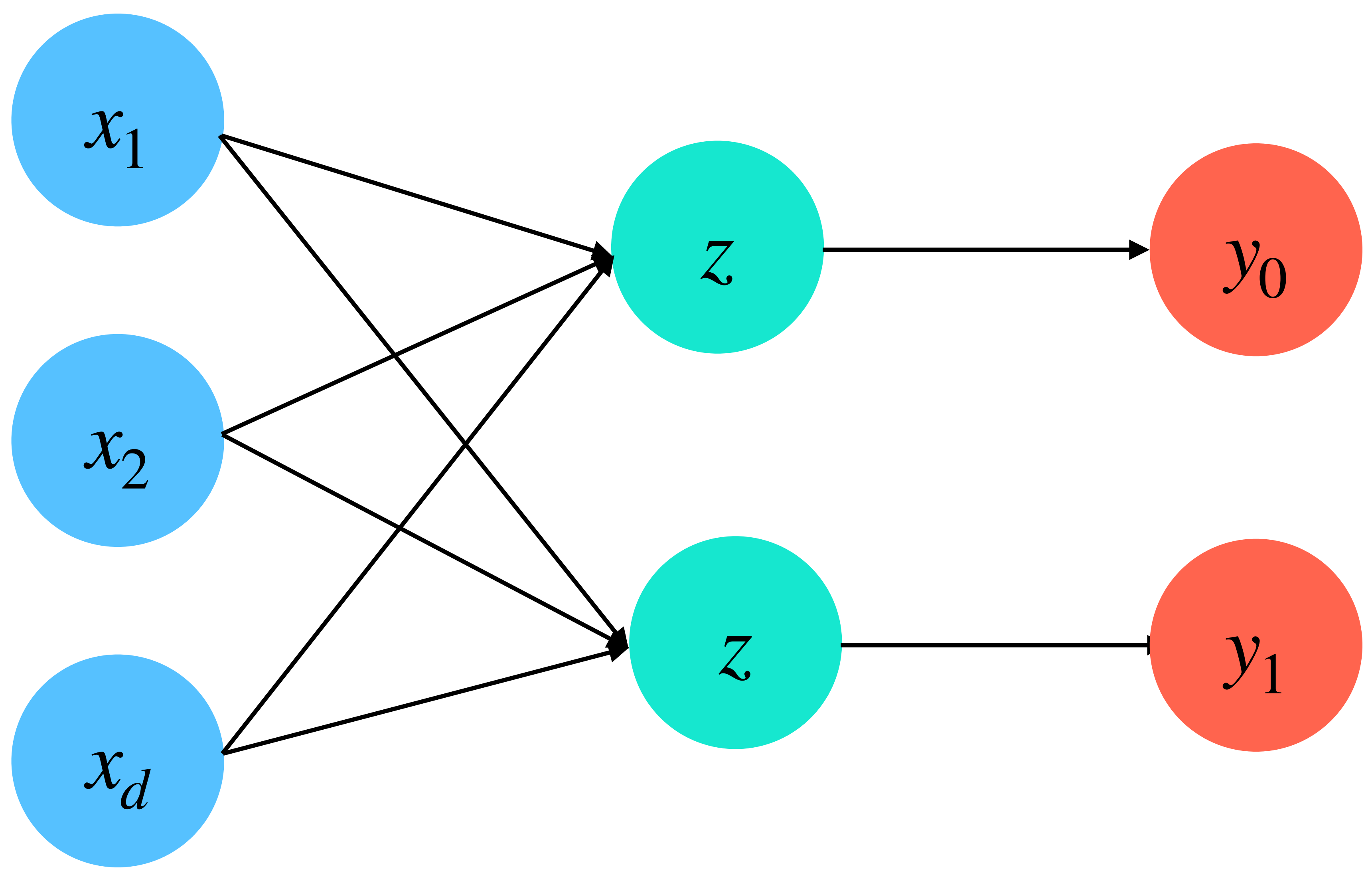




How to build more complicated functions using this perceptron?

Multilayer Perceptron

Multi output perceptron



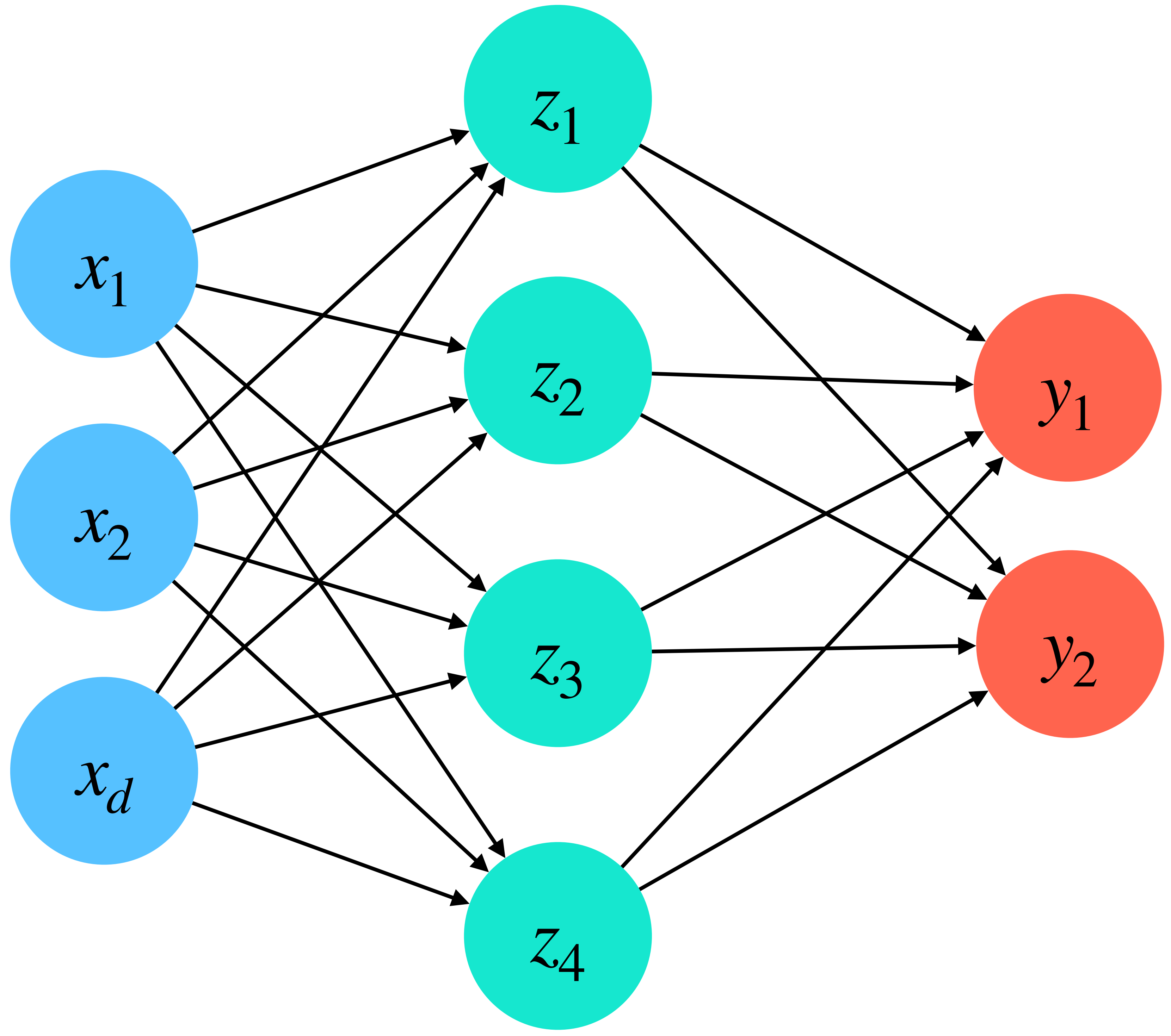
$y_0 = a(z_0)$

$y_1 = a(z_1)$

Because all inputs are connected to all outputs, these layers are called **Dense layers.**

Multilayer Perceptron
Single layer neural network

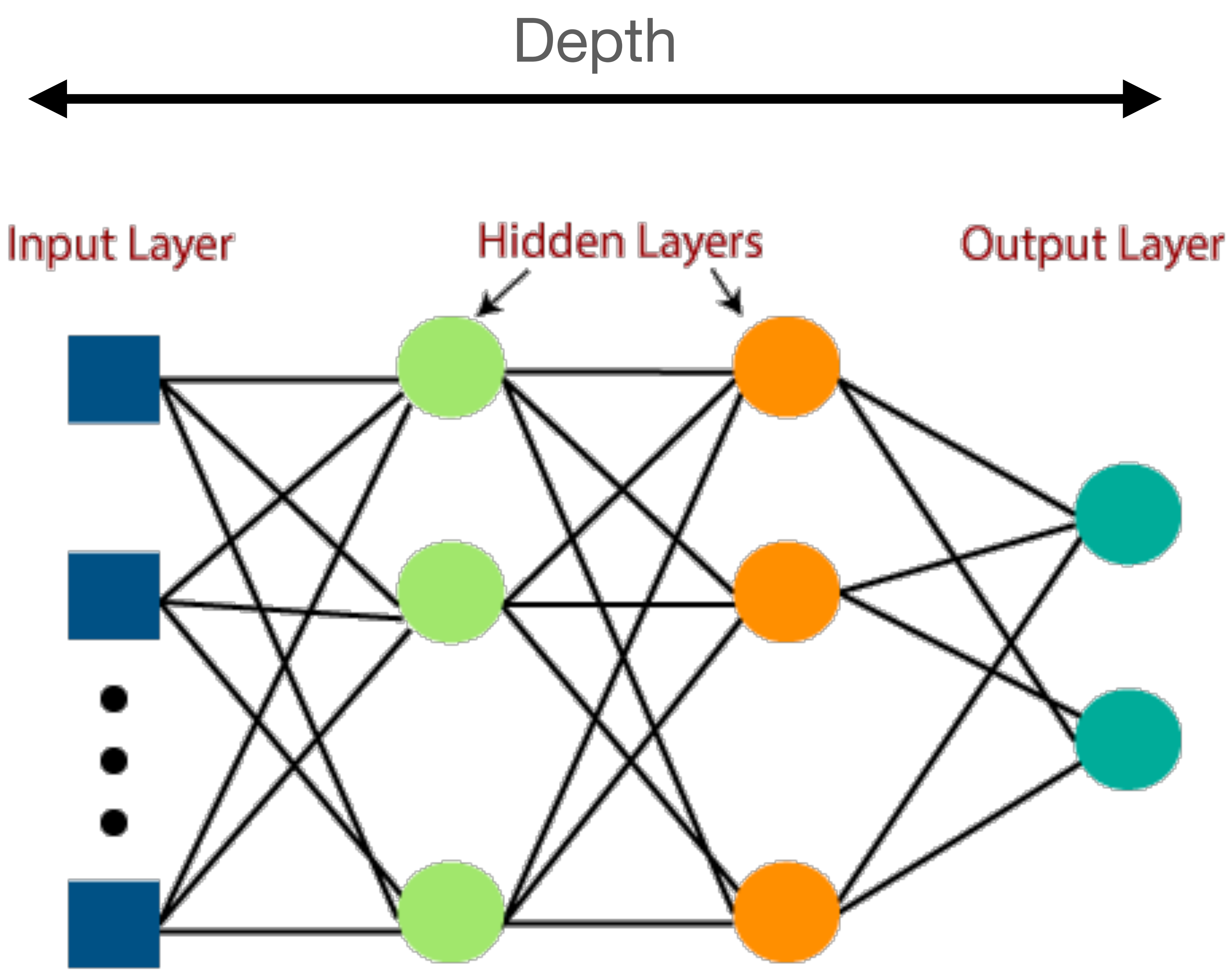
- Typically units are grouped together in layers and we can stack these layers to model more complex functions
- A multilayer networks is called a multilayer perceptron



MLP components

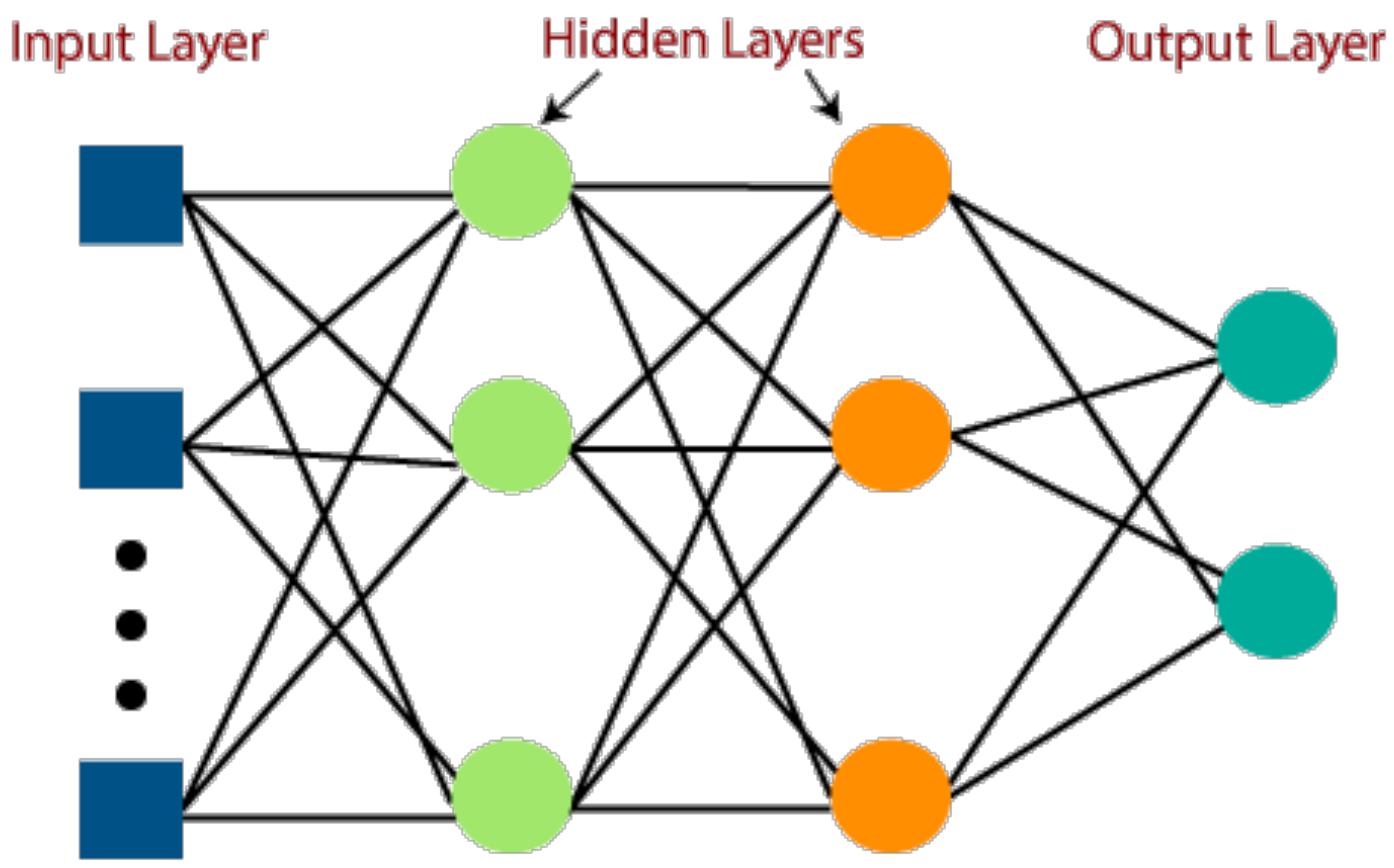
Weights

- Weight: the value of each synaptic connection between neurons.
- Training a network is equivalent to learning the weights.
- Notation:
  - The weight matrix of level k: $\theta^{(k)}$
  - The weight connecting i^{th} neuron of layer k to j^{th} neuron of layer $k+1$: $\theta_{ij}^{(k)}$



Multilayer Perceptron
Layers

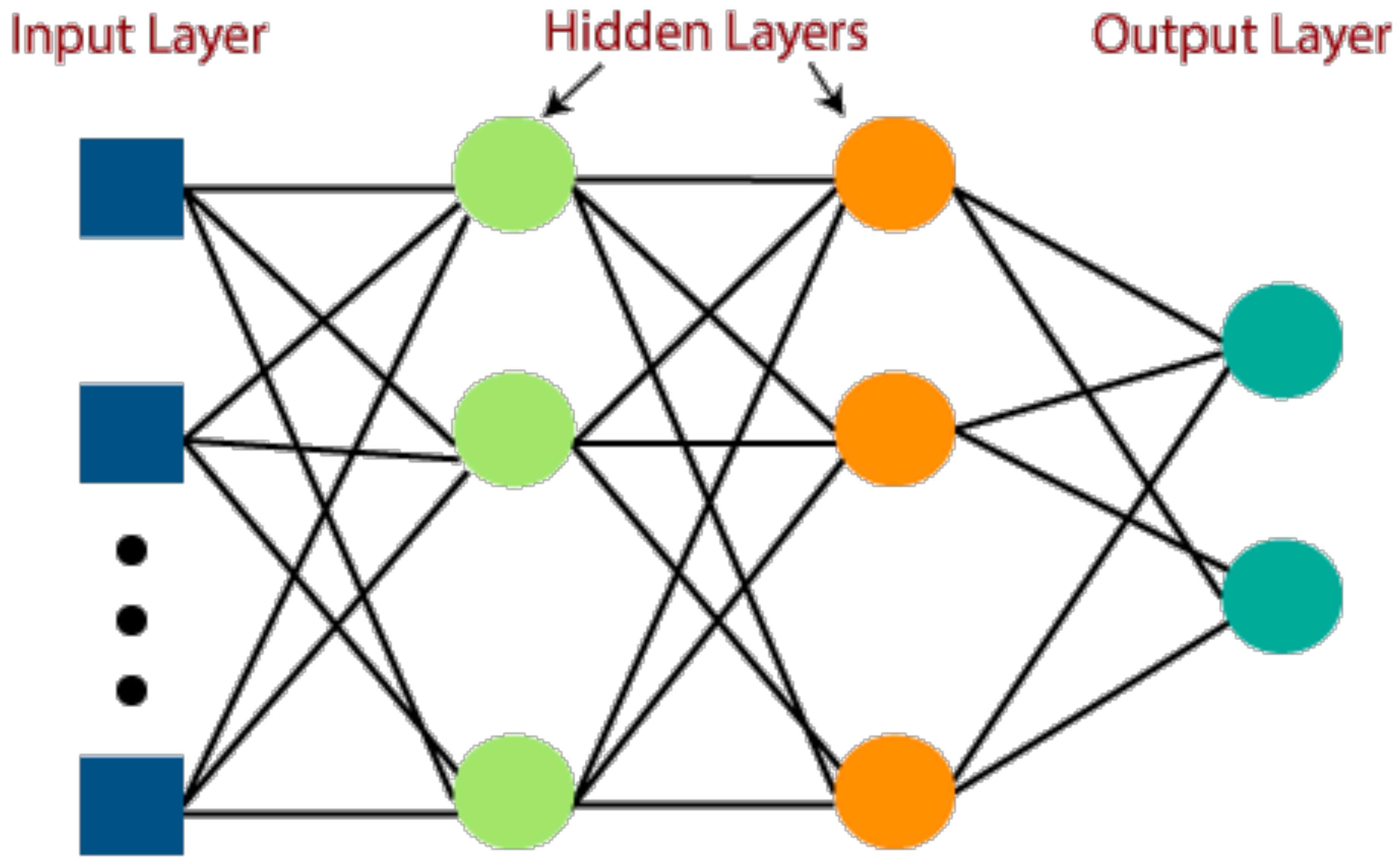
- Each layer connects N inputs to M outputs.
- When all inputs are connected to the outputs, we call it a **fully connected layer.** The weight connecting the layer matrix is $M \times N$.
- Note: Input and output of each layer are distinct from the input/output of the network



MLP components

Forward propagation

- Goal: Given x, estimate y
- propagate x through weights layer by layer (forward pass)

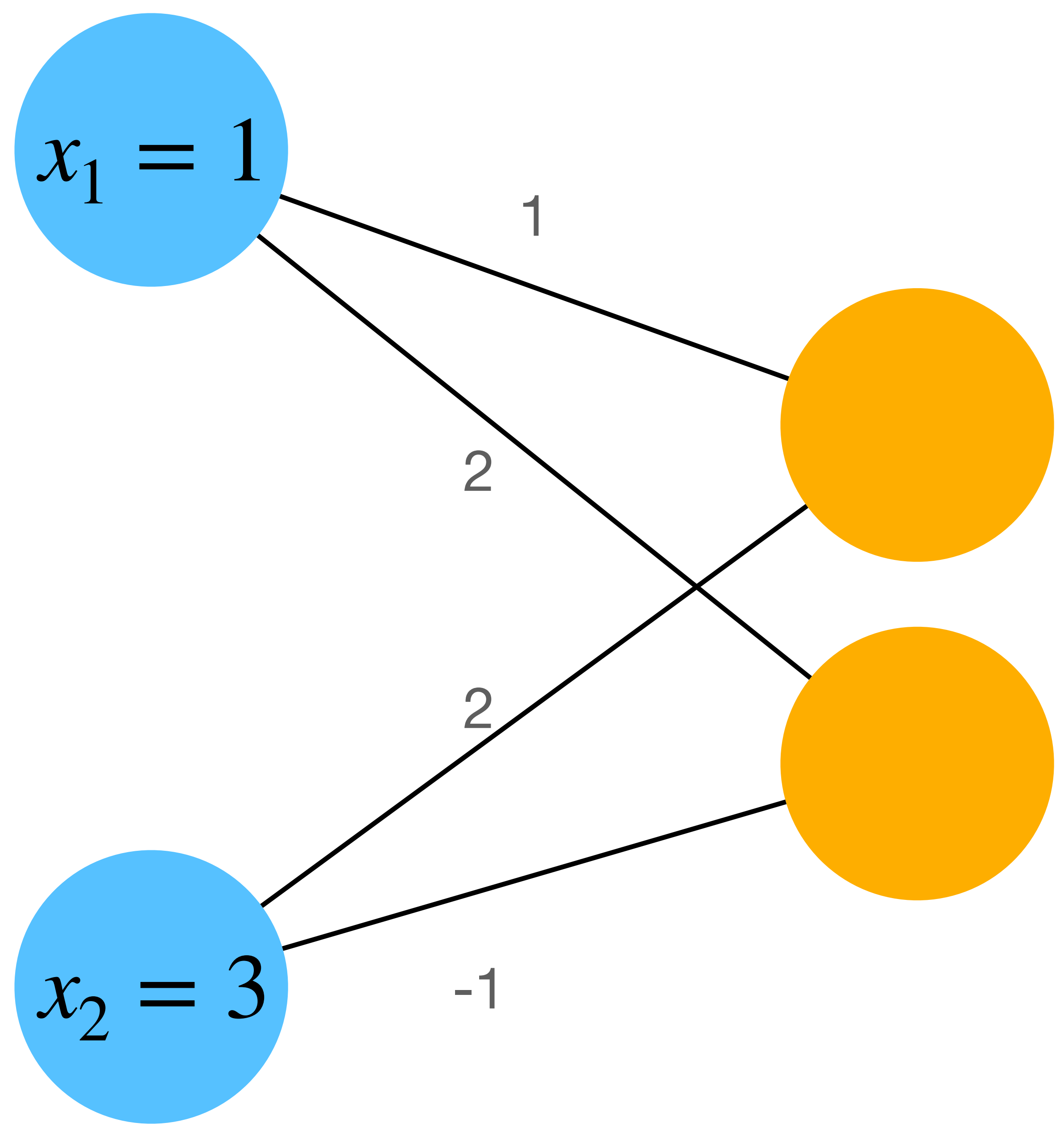


MLP components

Example (Forward pass)

Estimate the value of the neurons

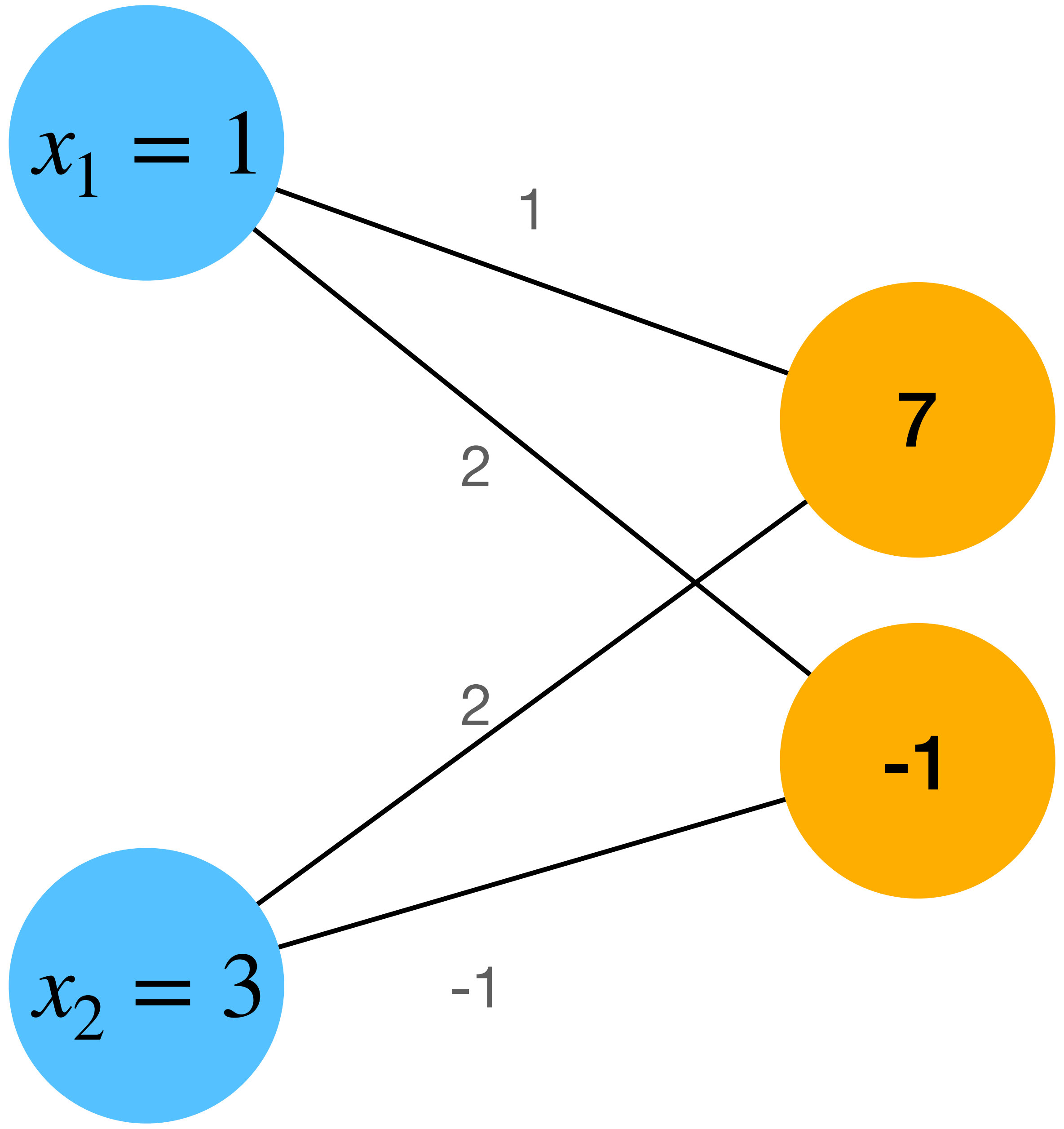
$$z_j = \sum_j \theta_{ij}^{(m-1)} x_i$$



MLP components

Example (Forward pass)

Estimate the value of the neurons



Note: These are the pre activation values!

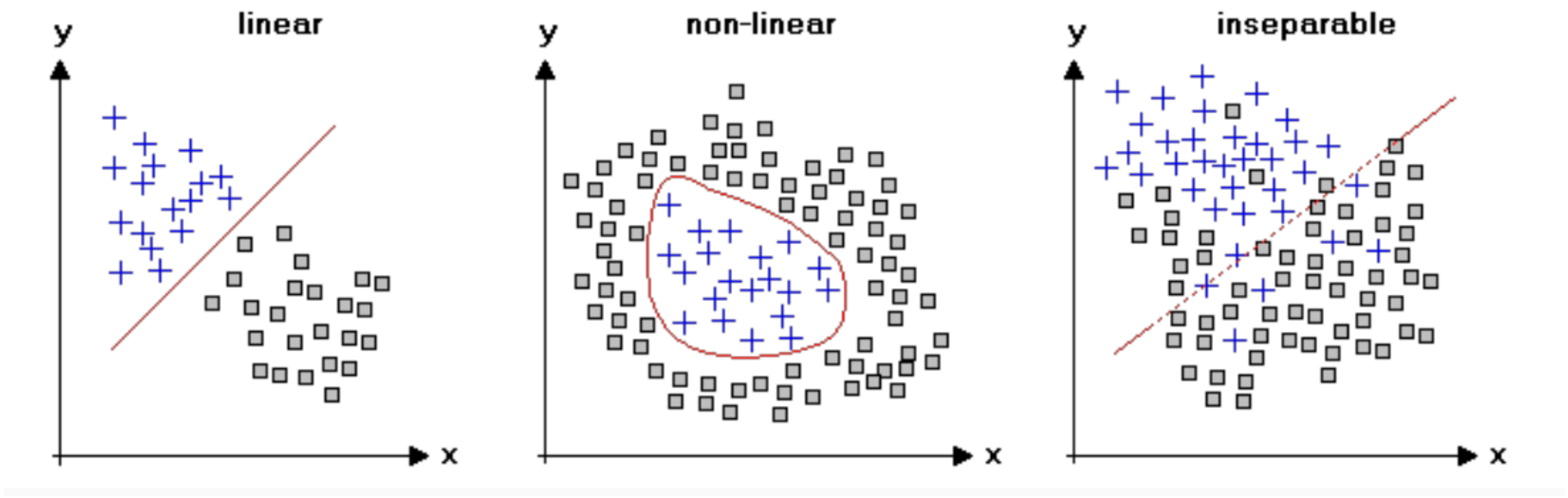
Break

10 minutes

MLP components

Activation functions

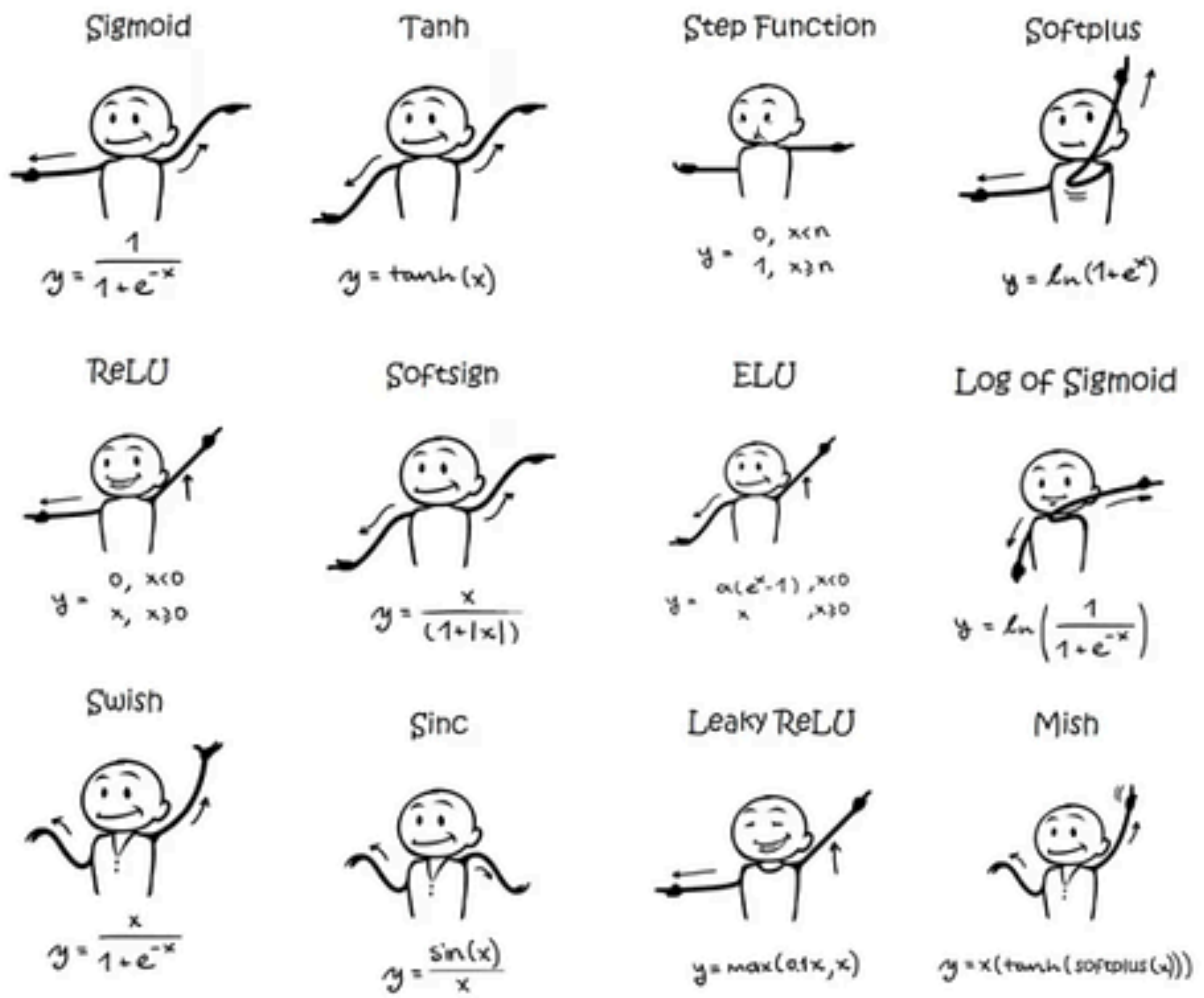
- Successive weight matrices multiplied by the input would just be a **linear transformation**.
- How can we model non-linear decision boundaries?



MLP components

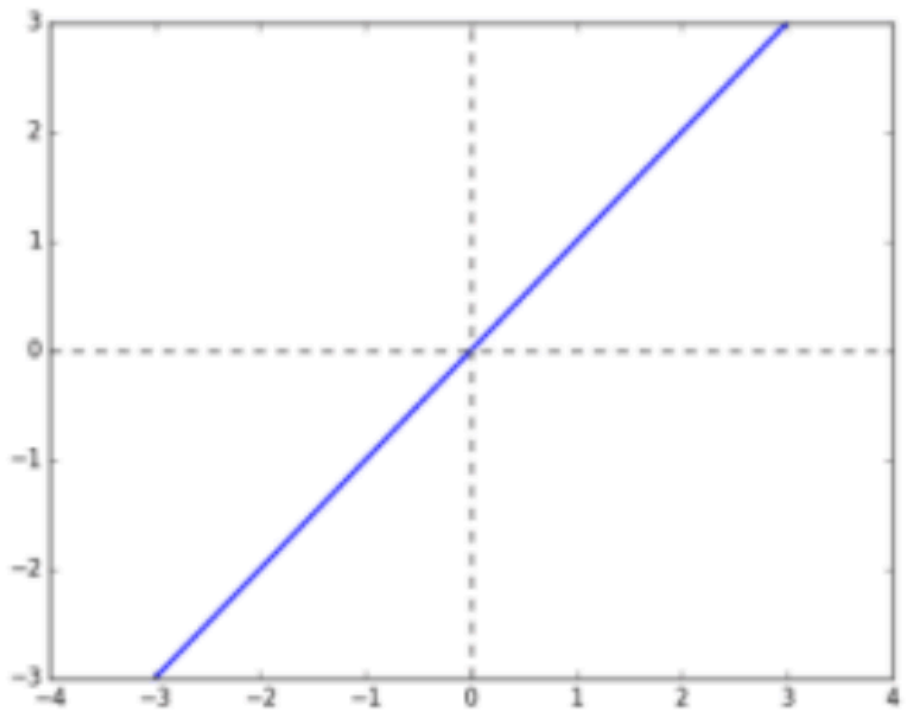
Activation functions

- To learn **non-linearly separable** mappings we need the non-linear **activation functions**.
- Non-linear activation functions introduce non-linearity while allowing us to use gradients for optimizing the weight values.



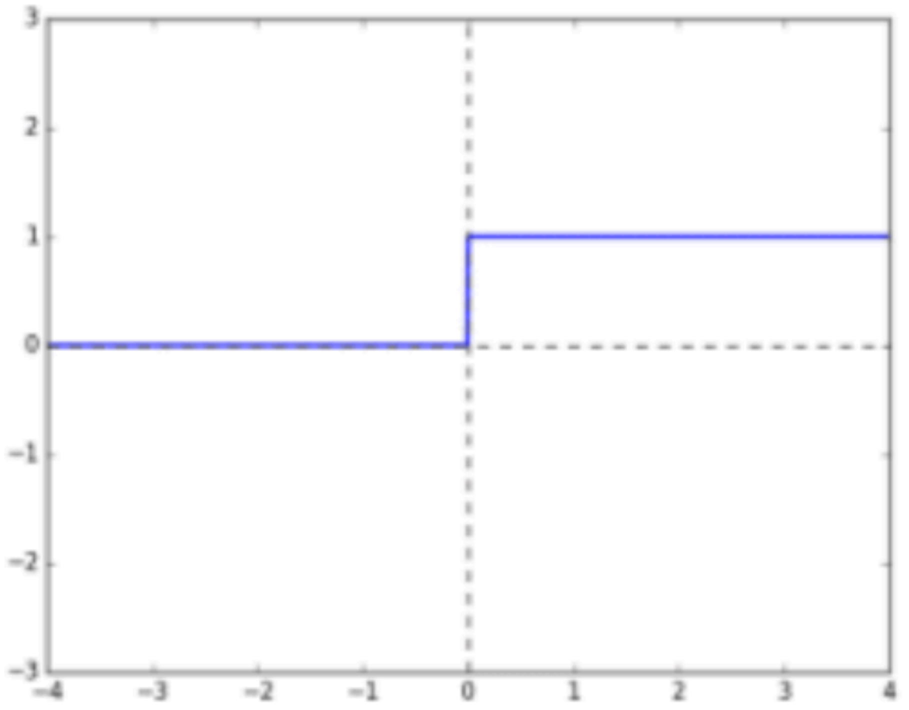
MLP components

Some activation functions



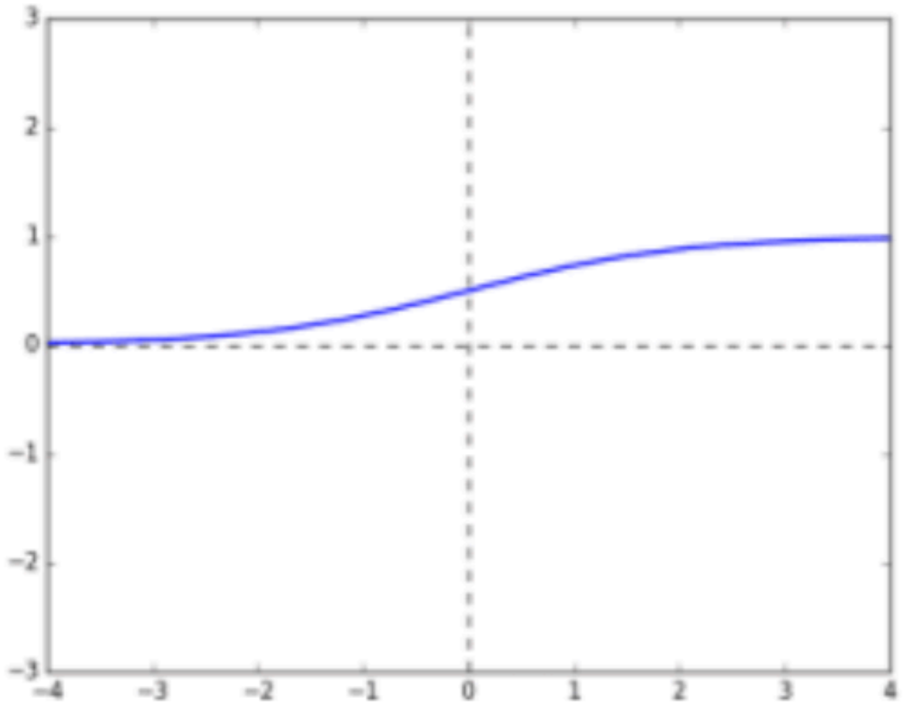
Linear

$$y = z$$



Hard Threshold

$$y = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \le 0 \end{cases}$$

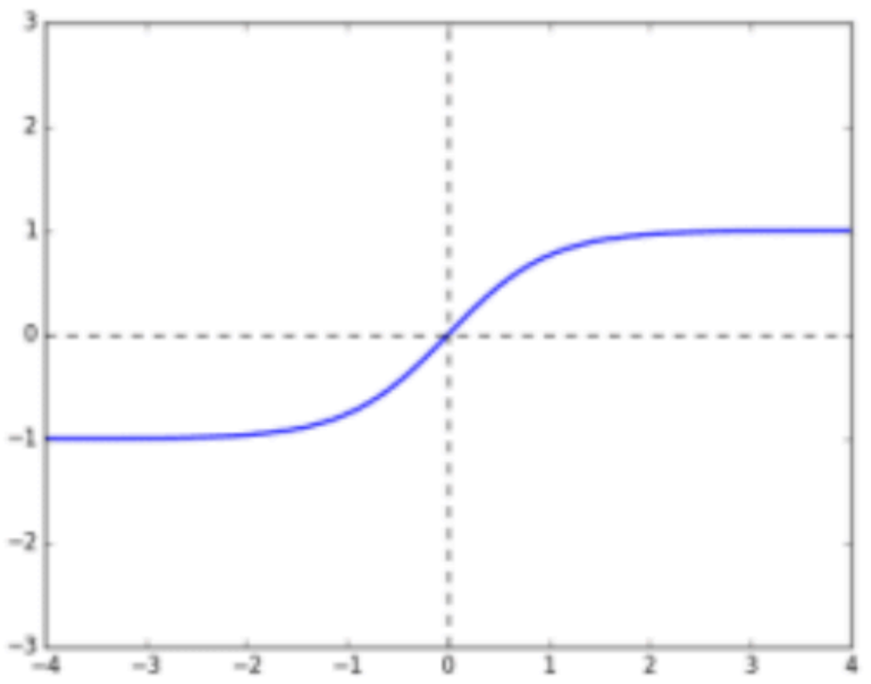


Logistic

$$y = \frac{1}{1 + e^{-z}}$$

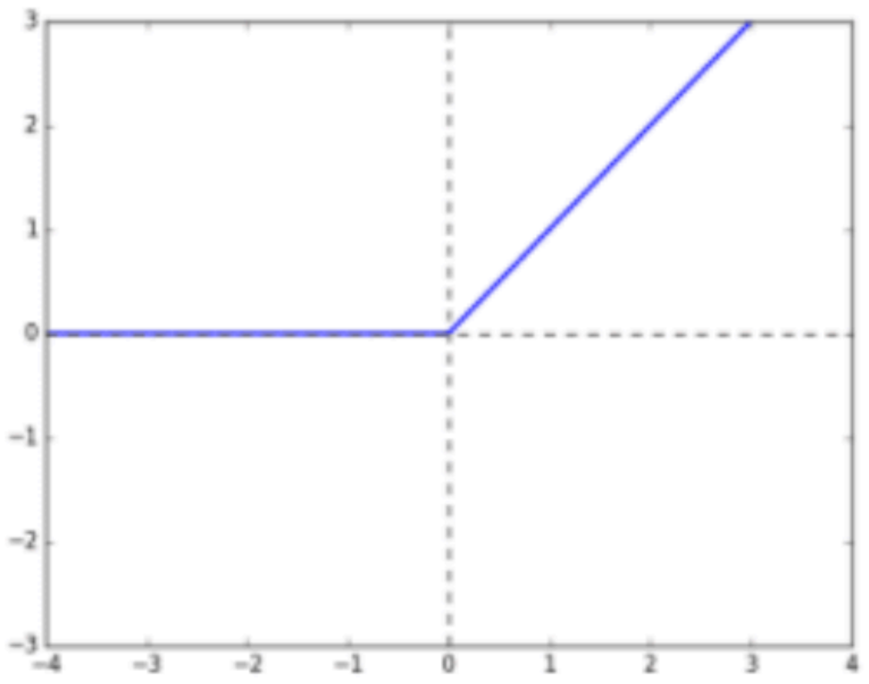
MLP components

Some activation functions



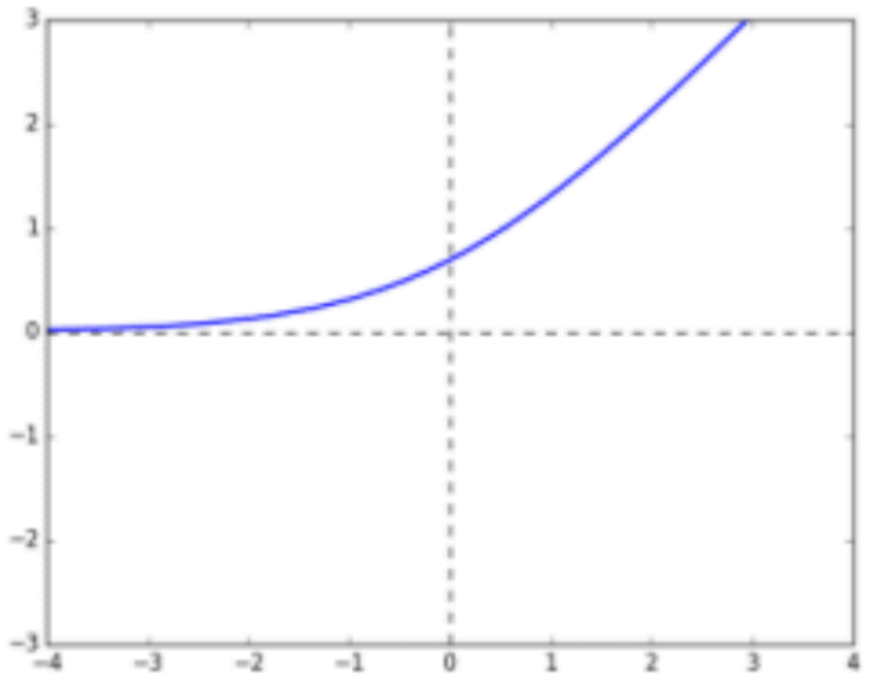
Hyperbolic Tangent (tanh)

$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



Rectified Linear Unit (ReLU)

$$y = \max(0, z)$$



Soft ReLU

$$y = \log 1 + e^z$$

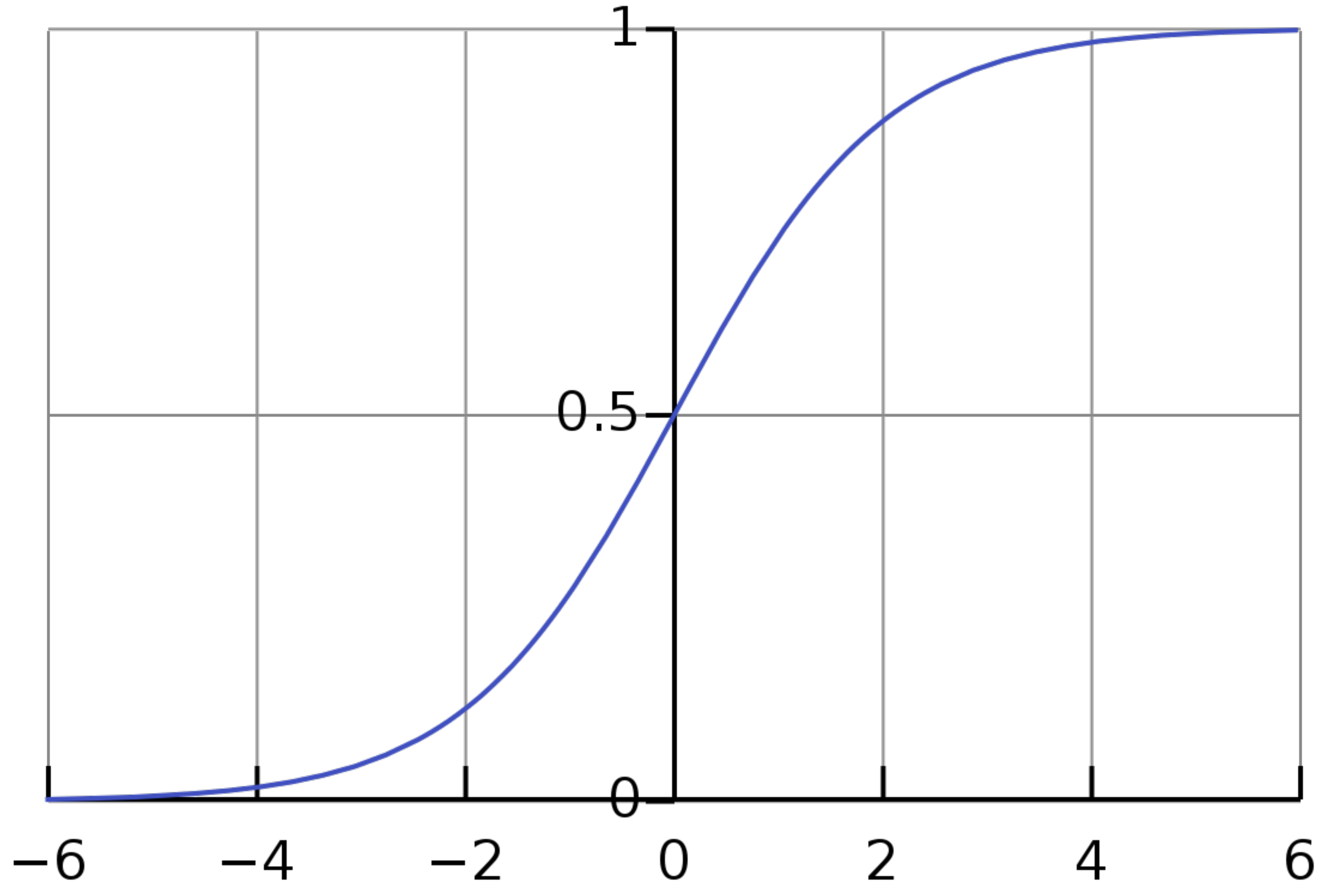
MLP components

Activation functions

Sigmoid $\dfrac{1}{1+e^{-x}}$

Used commonly for cases where the value should be scaled between 0-1.

Used to be, but is no longer the “go to” activation.



Problems:

- Vanishing gradients: gradients are small, and when multiplied for all layers, they become very close to zero.
- Gradient of smaller and larger values approach zero, which means no updates to those weights.

MLP components

Activation functions

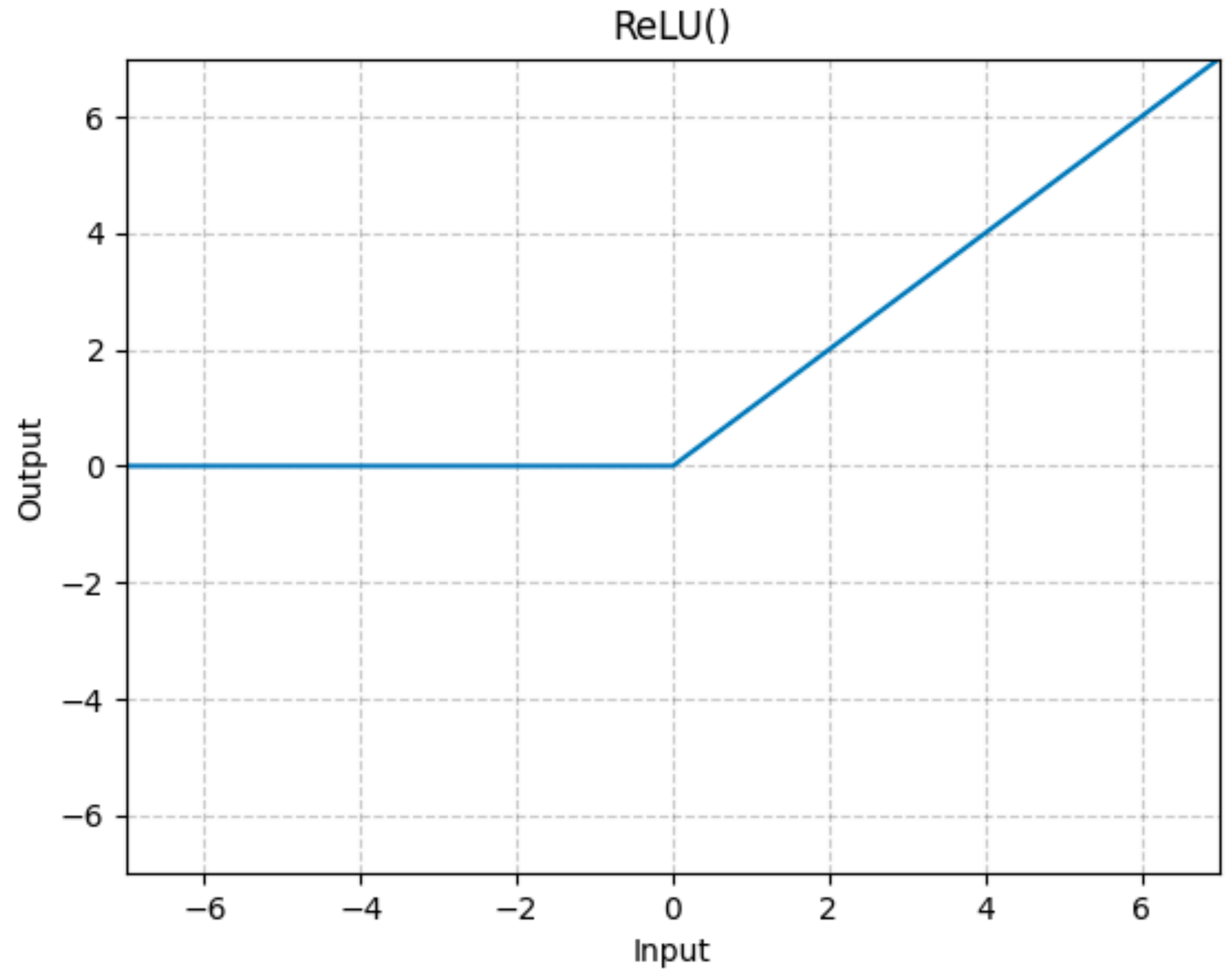
Rectified Linear Unit (ReLU) $\max\{0, x\}$

Works well empirically, so it has become the "go to" activation in many applications

More discriminatory power as values are no longer restricted

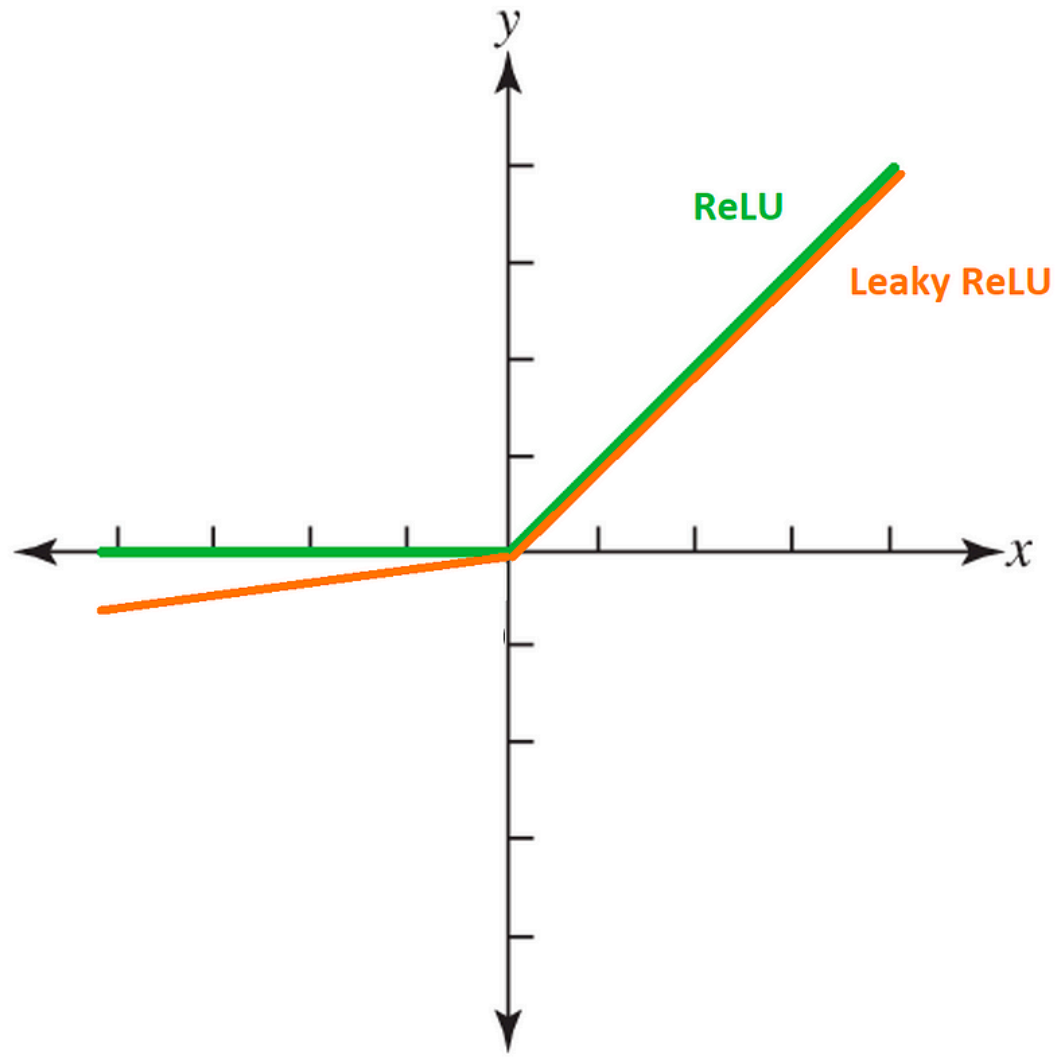
Fixes the vanishing gradient problem of Sigmoids

Problem: Dying ReLU problem —> When the unit always output zeros for all inputs.



Leaky ReLU

Designed to prevent the dying ReLU problem



MLP components

Forward propagation activation function

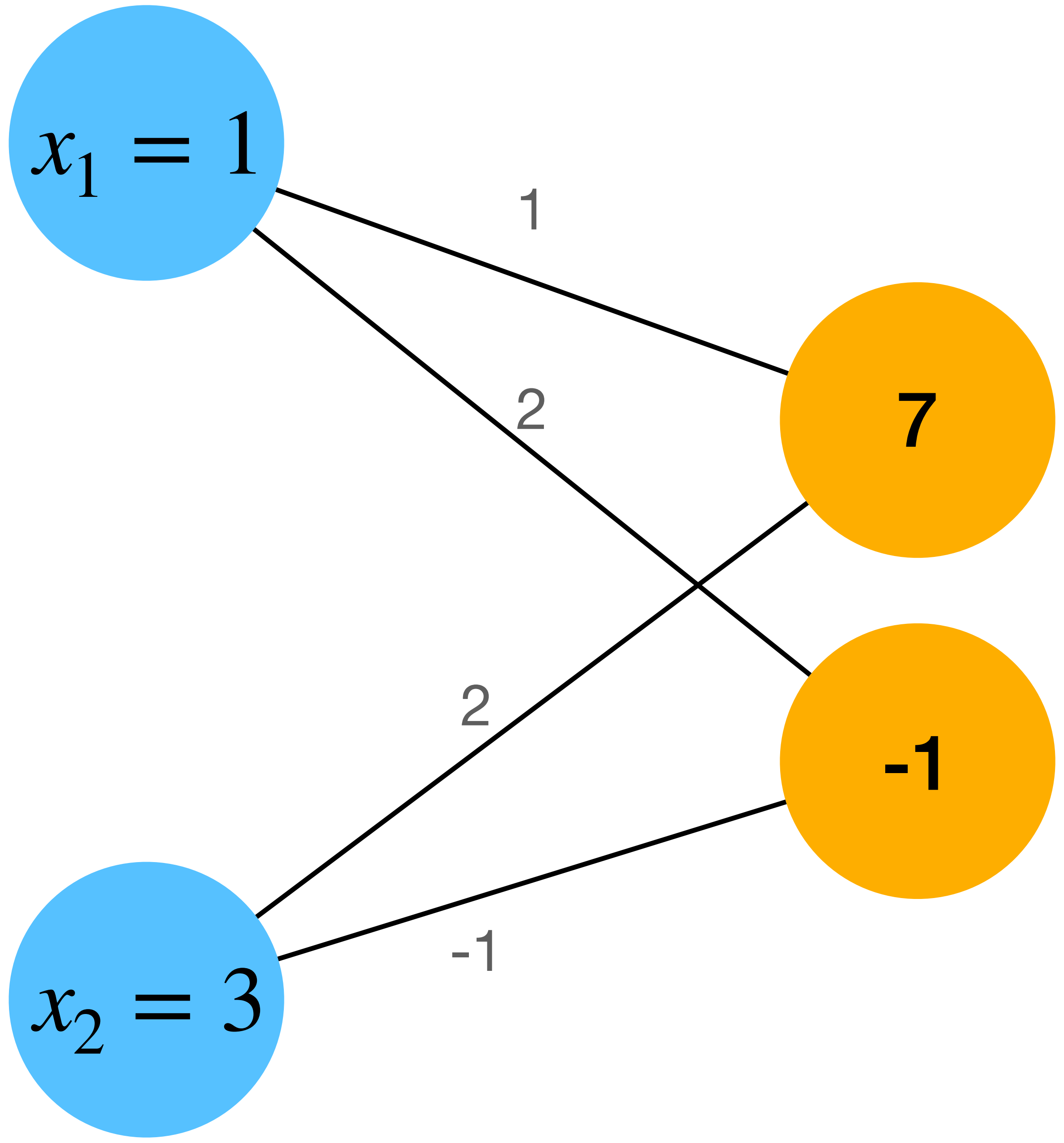
- Apply an activation function $a(x)$ on each output $z_j^{(m)}$
    - Notation: the activation value of the j-th neuron in layer m is $a_j^{(m)}$
- Intuition: Whether a neuron fires or not, and the magnitude of its activation value is useful in piecing together useful information for accomplishing the task. (unhelpful features should be zeroed out by the activation function)

$$z_j = \sum_j \theta_{ij}^{(m-1)} x_i \qquad\qquad a_j^{(m)} = a(z_j^{(m)}) = a(\sum_j \theta_{ij}^{(m-1)} x_i)$$

MLP components

Example

Estimate the value of the neurons post activation (ReLU/sigmoid)



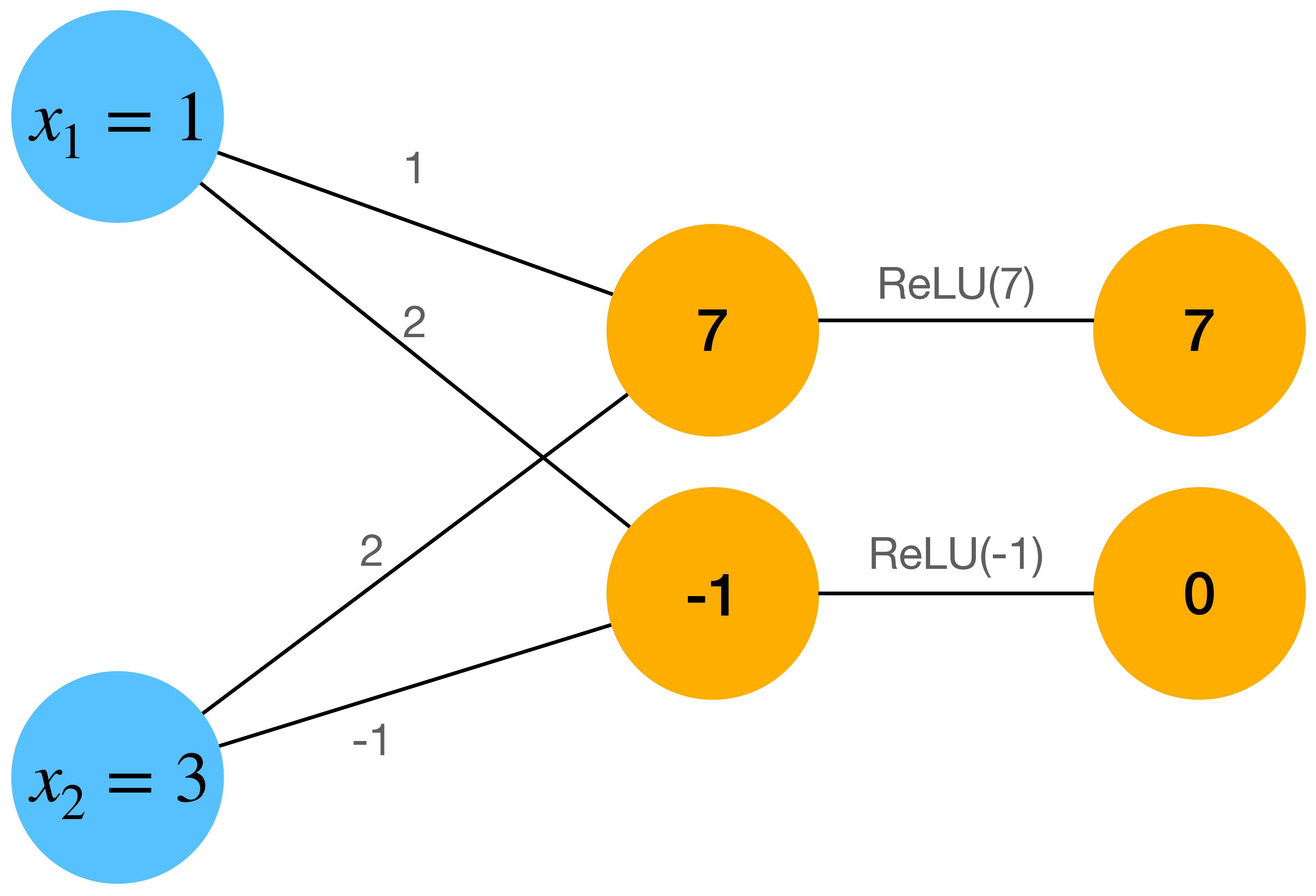
$$z_j = \sum_j \theta_{ij}^{(m-1)} x_i$$

$$a_j^{(m)} = a(z_j^{(m)}) = a(\sum_j \theta_{ij}^{(m-1)} x_i)$$

MLP components

Example

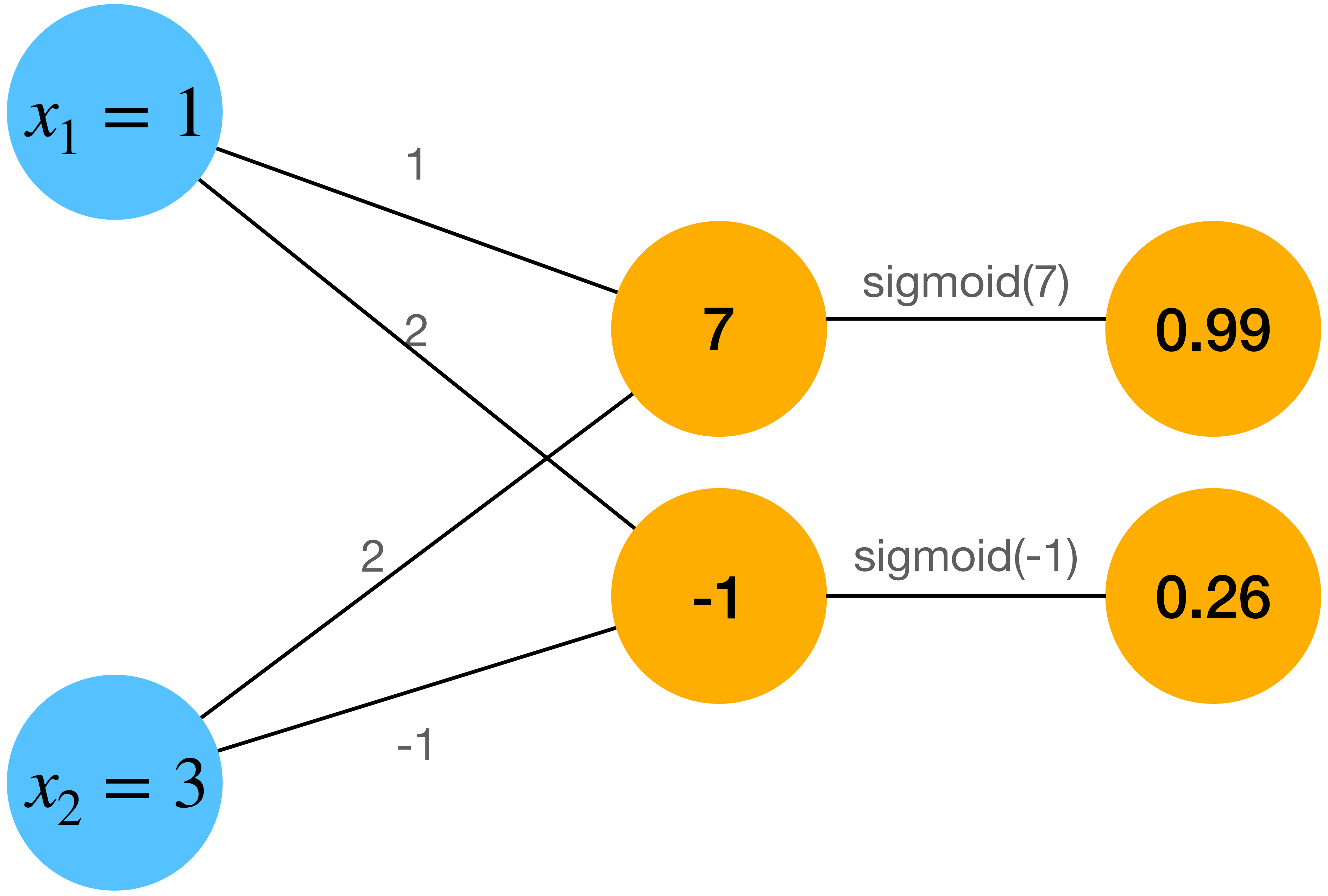
Estimate the value of the neurons post activation: ReLU



MLP components

Example

Estimate the value of the neurons post activation: sigmoid



$$\frac{1}{1 + e^{-7}}$$

$$\frac{1}{1 + e^{1}}$$

MLP components
Loss function

Loss function review:

- Loss function is a measure of how bad the model performed
- Compares model output to ground truth labels (in a supervised setting)
- Goal of training ML models: minimize the loss function

MLP components

Loss function

Mean Square Error Loss:

Measures the variance of model output against target.

Ideal for regression settings

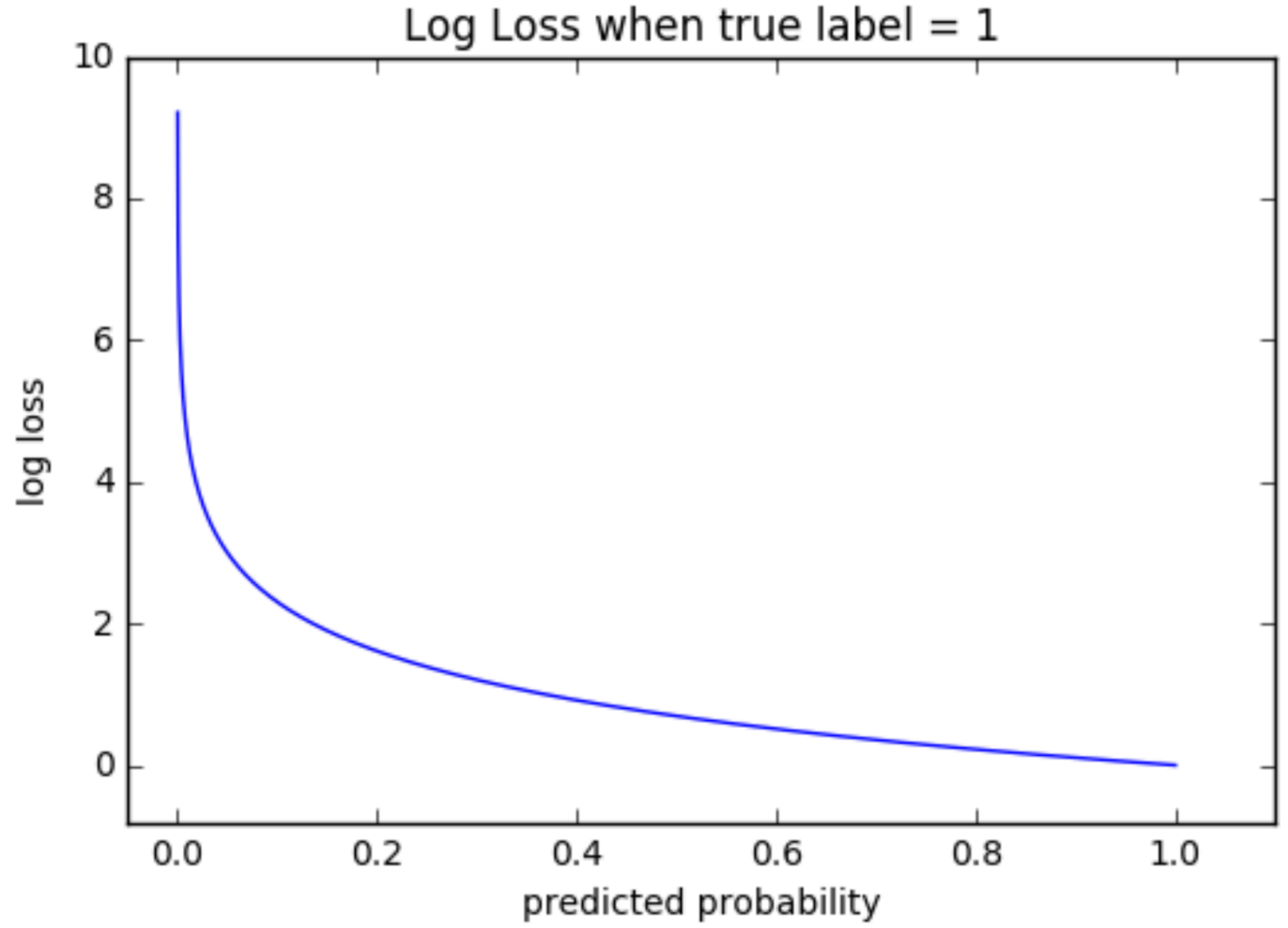
$$\mathcal{J}(\theta) = \frac{1}{2}\sum_{i}^{m}(y^i - t^i)^2$$

Cross Entropy Loss:

Measures the error of a model given the output is between 0-1

Stronger gradients as predicted probability

$$\mathcal{J}(\theta) = -\sum_{i}^{m} t^i \log(y^i) + (1 - t^i)\log(1 - y^i)$$

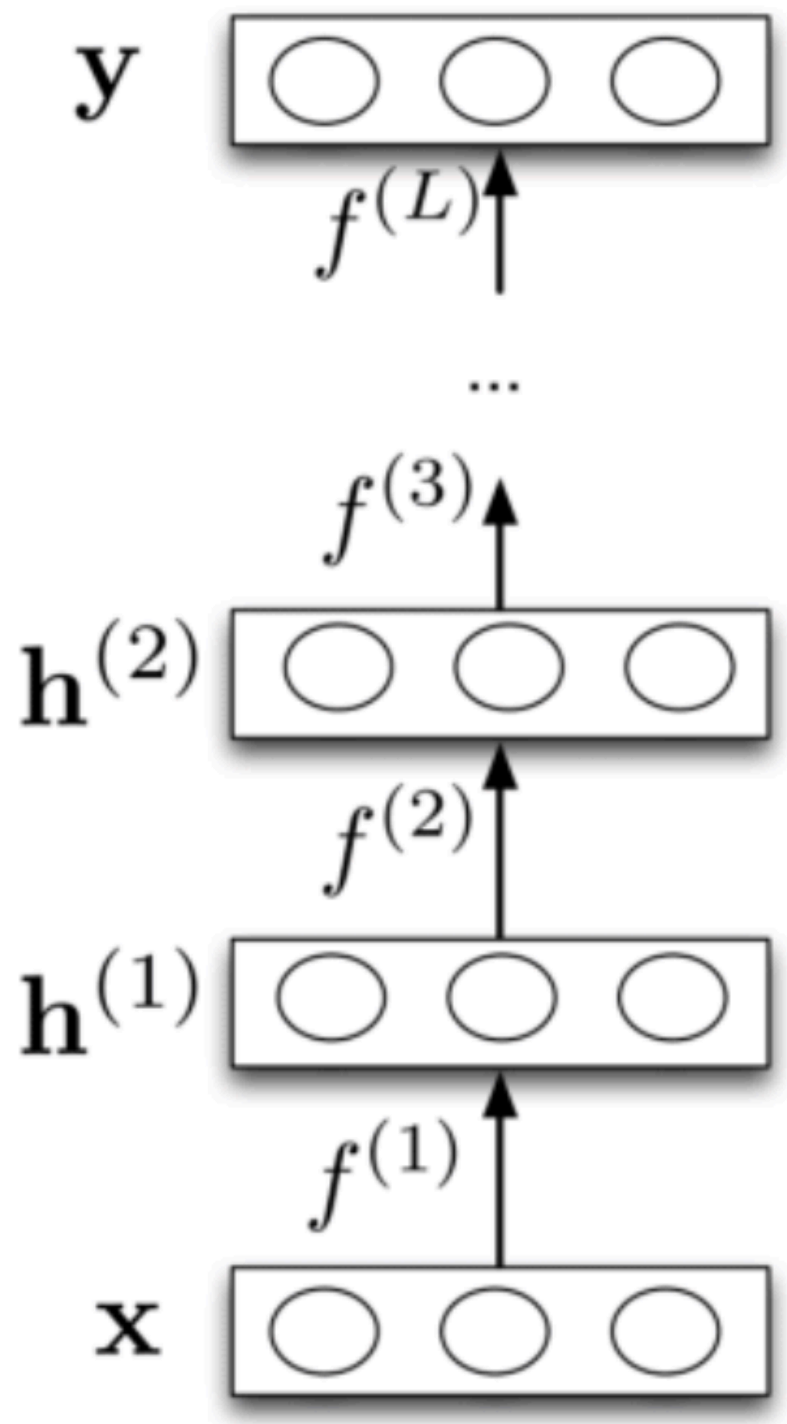


Multilayer Perceptron

Abstraction

- Each layer computes a function, so the network computes a composition of functions. $y = f^{(L)} \ldots f^{(1)}(x)$
- Neural networks provide modularity, we can implement each layer's computations as a black box.

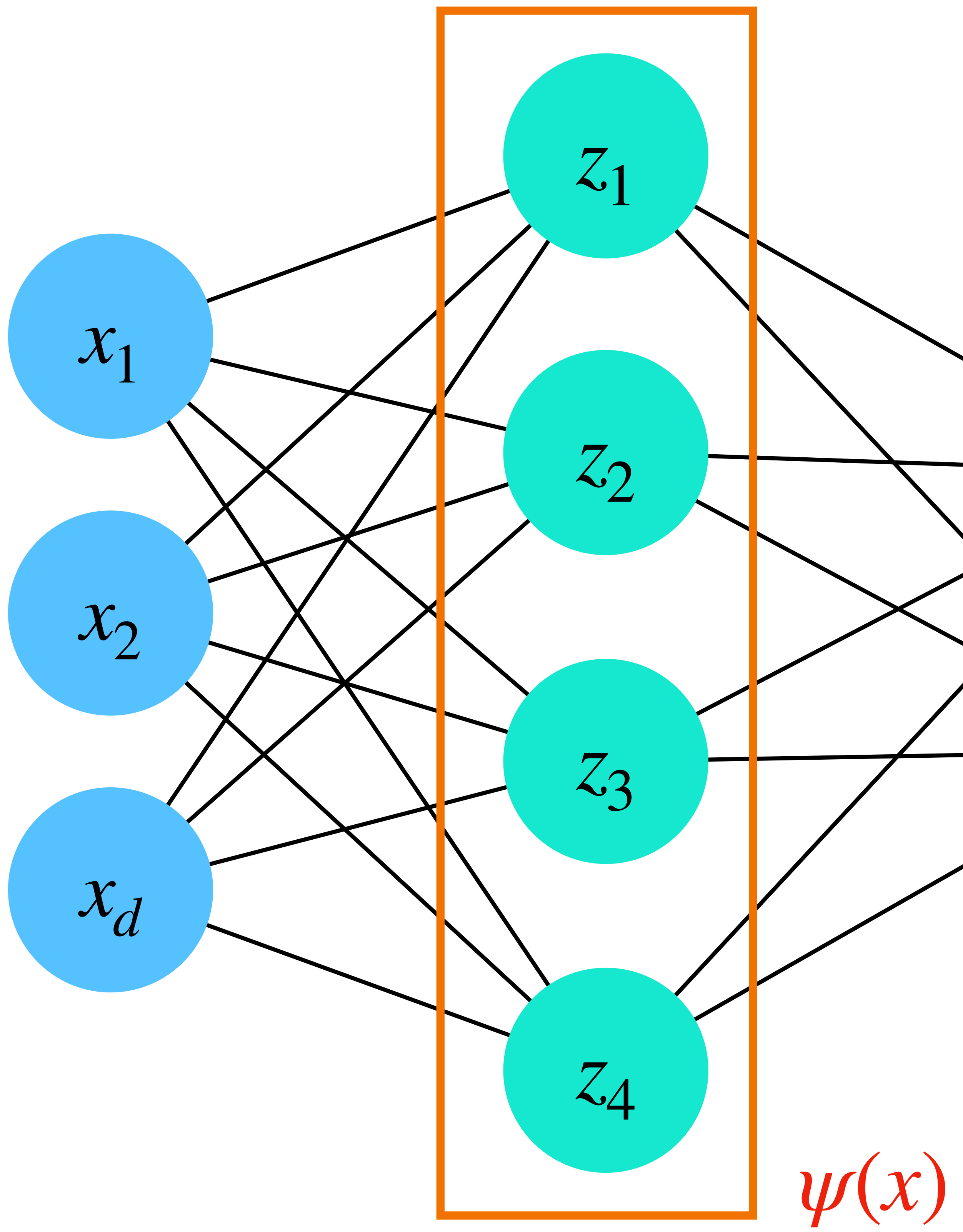
$$
\begin{gathered}
h^{(1)} = f^{(1)}(x) \\
h^{(2)} = f^{(2)}(h^{(1)}) \\
\vdots \\
y = f^{(1)}(h^{(L-1)})
\end{gathered}
$$



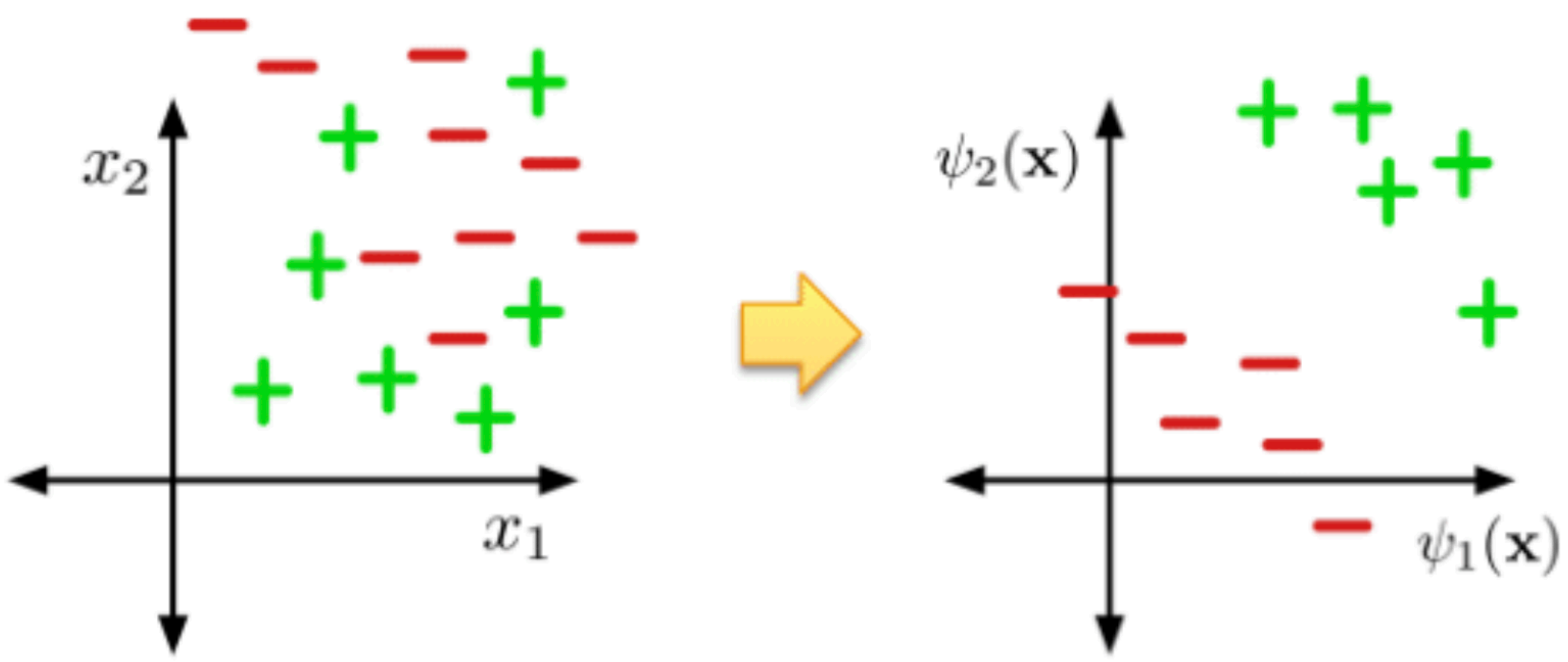
Multilayer Perceptron

Feature learning

Neural networks can be seen as a way of learning features.



The goal:



Expressive power

- We've seen that there are some functions that linear classifiers can't represent. Are deep networks any better?
- A network composed of a sequence of linear layers can be equivalently represented with a single linear layer

$$\mathbf{y} = \underbrace{\mathbf{W}^{(3)}\mathbf{W}^{(2)}\mathbf{W}^{(1)}}_{\triangleq \mathbf{W}'}\mathbf{x}$$

So **deep linear networks** are no more expressive than linear regression!

Expressive power

- Multilayer neural nets with nonlinear activation functions are **universal approximations**: they can approximate any function arbitrarily well.
- This has been shown for various activation functions, even ReLU that is “almost” linear.

<u>Limits of universality:</u>

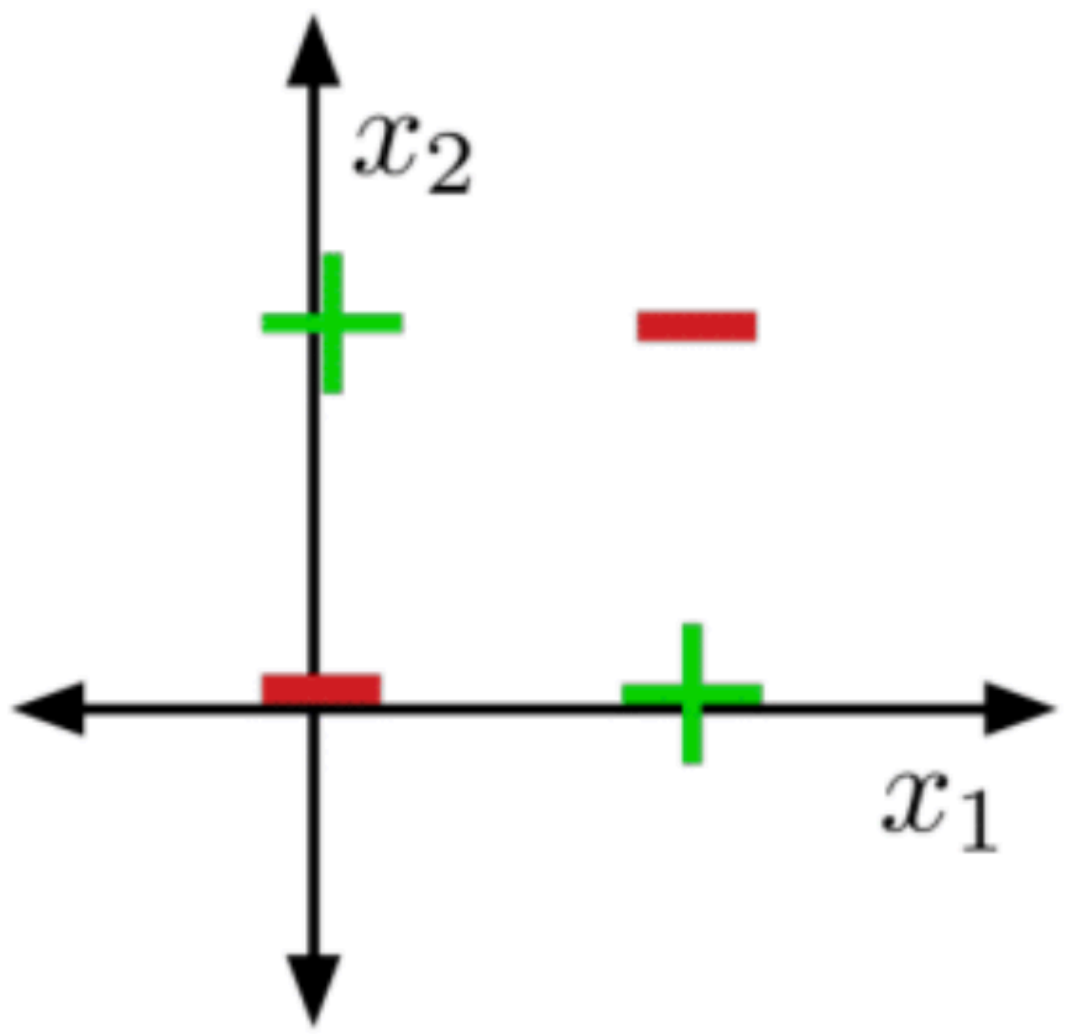
- You may need to represent an exponentially large network
- If you can learn any function, you’ll overfit.

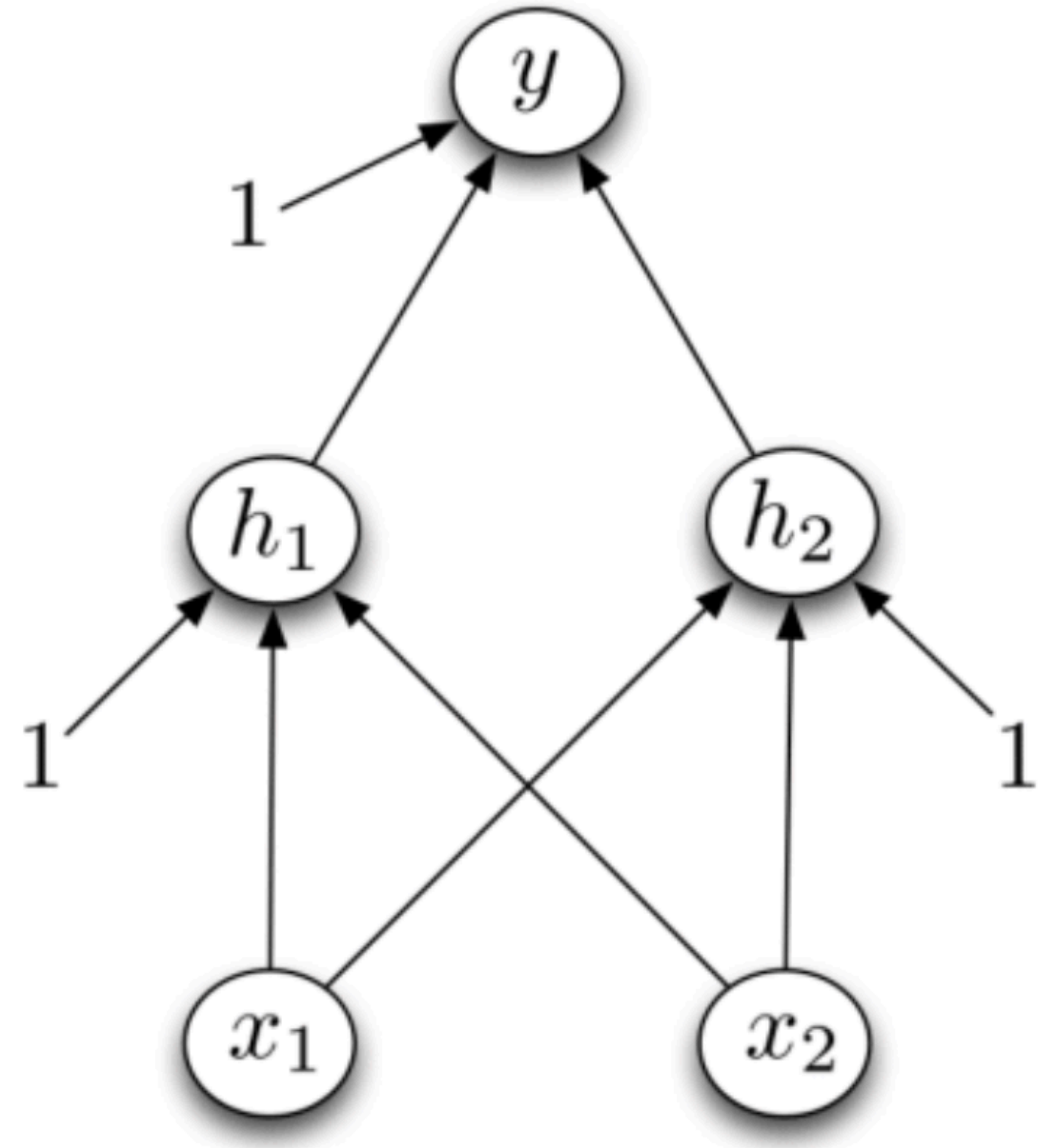
Expressive power

Example

Design a network to compute XOR

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0



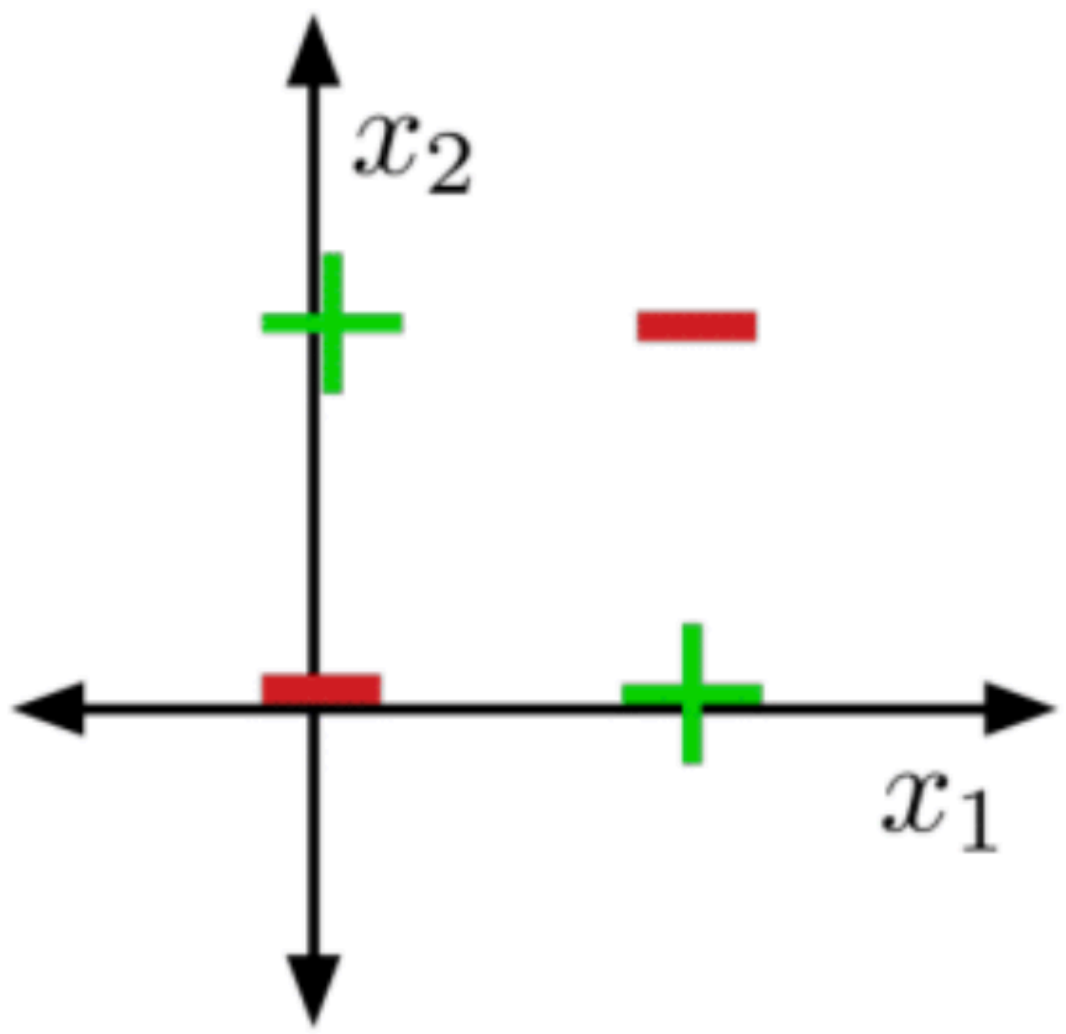


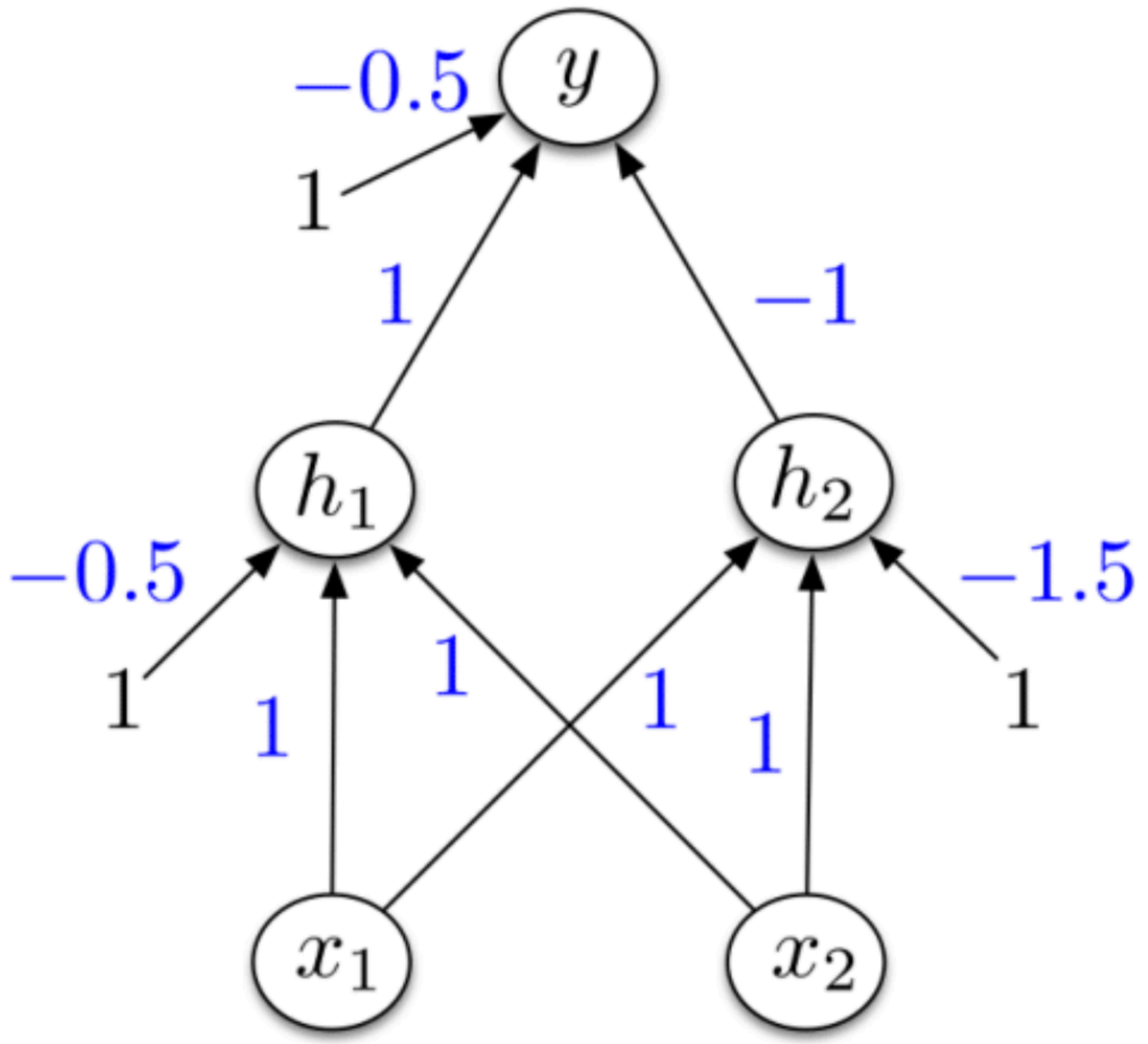
Expressive power

Example

Can you come up with a different set of weights?

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

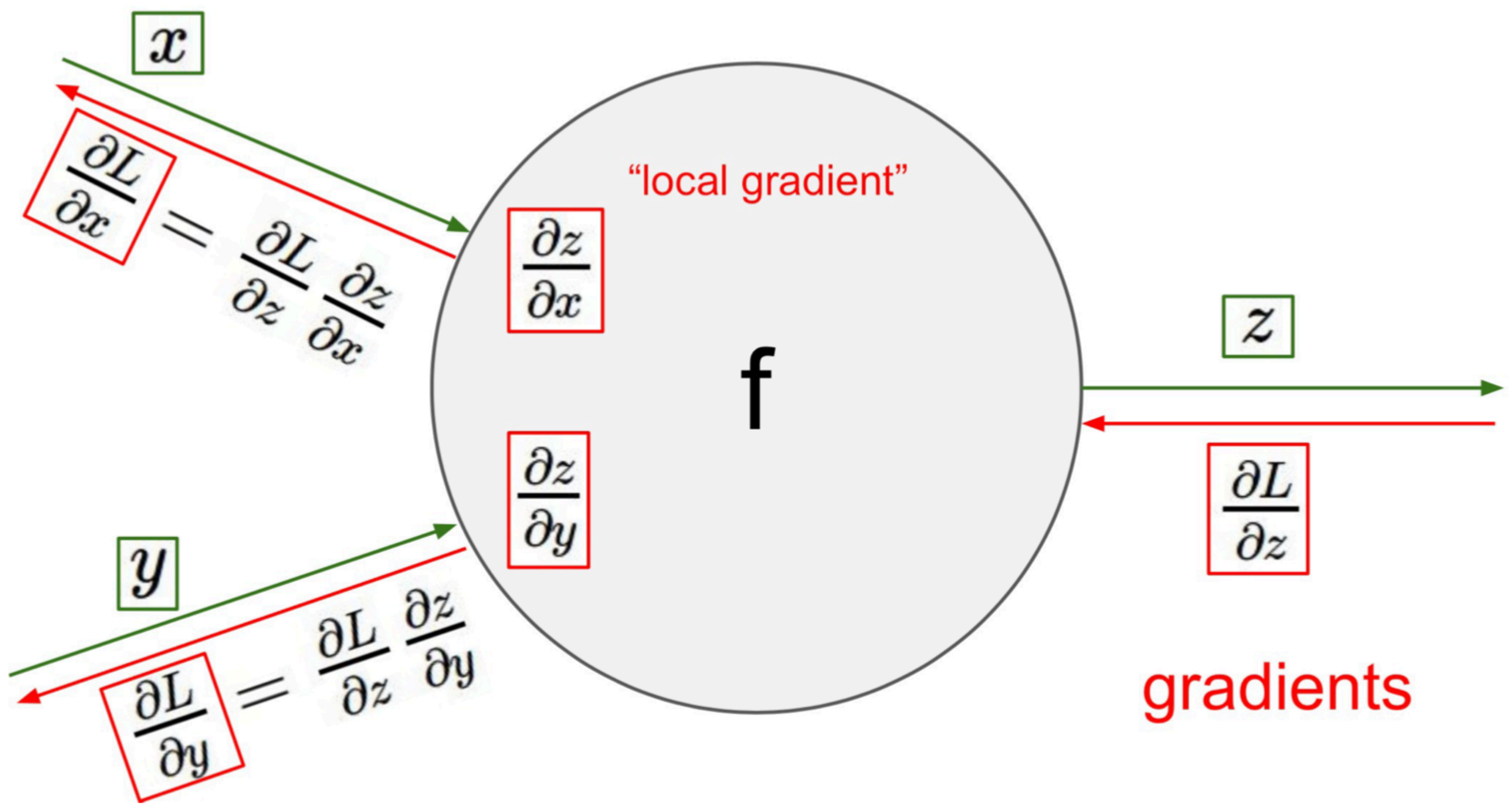




Learning MLPs

Backpropagation

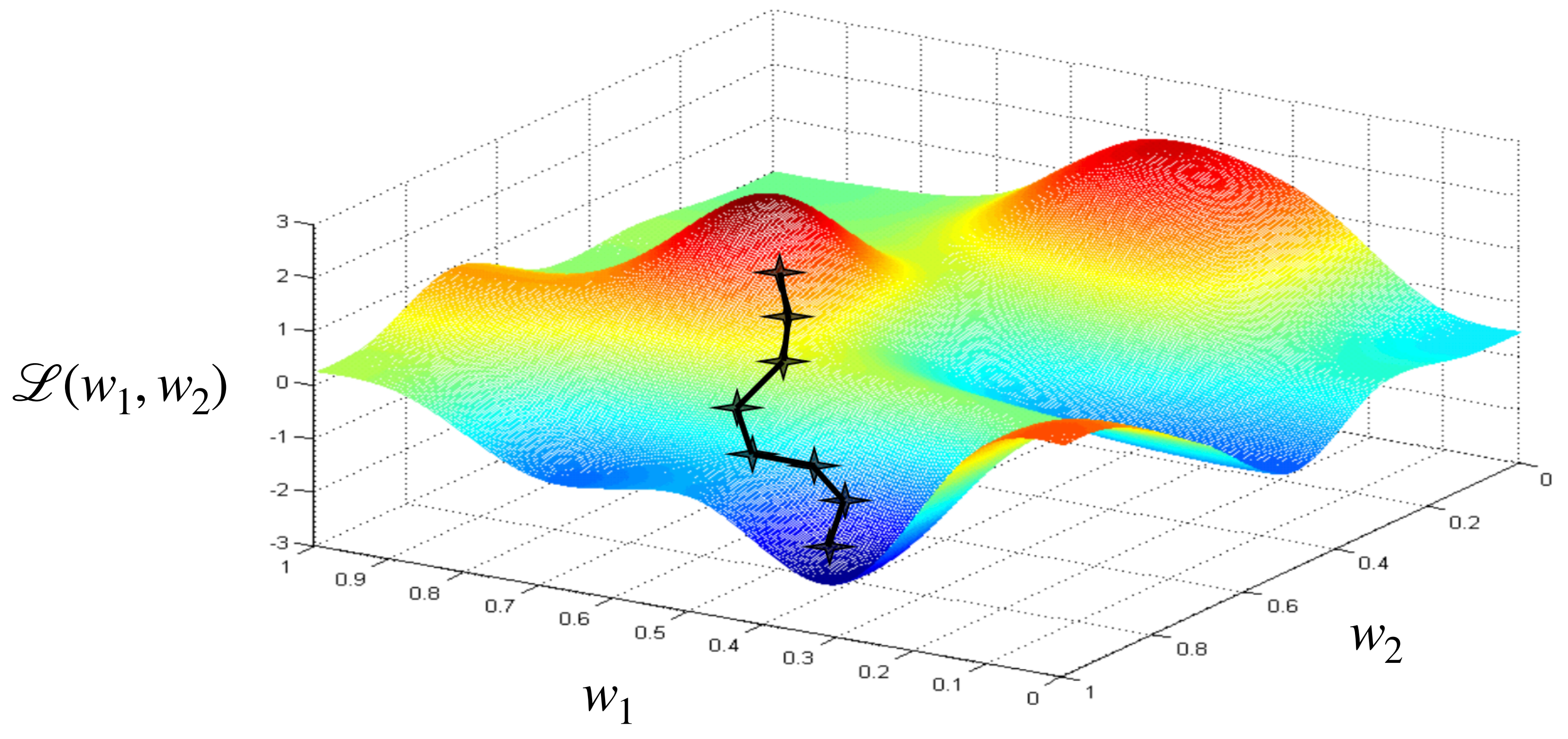
- We have seen that multilayer neural networks are powerful. But how can we learn them?
- **Backpropagation** is the central algorithm that enables that!
  - It is an algorithm for computing gradients.
  - Has a clever and efficient use of the chain rule for derivatives.



Learning MLPs

Back propagation

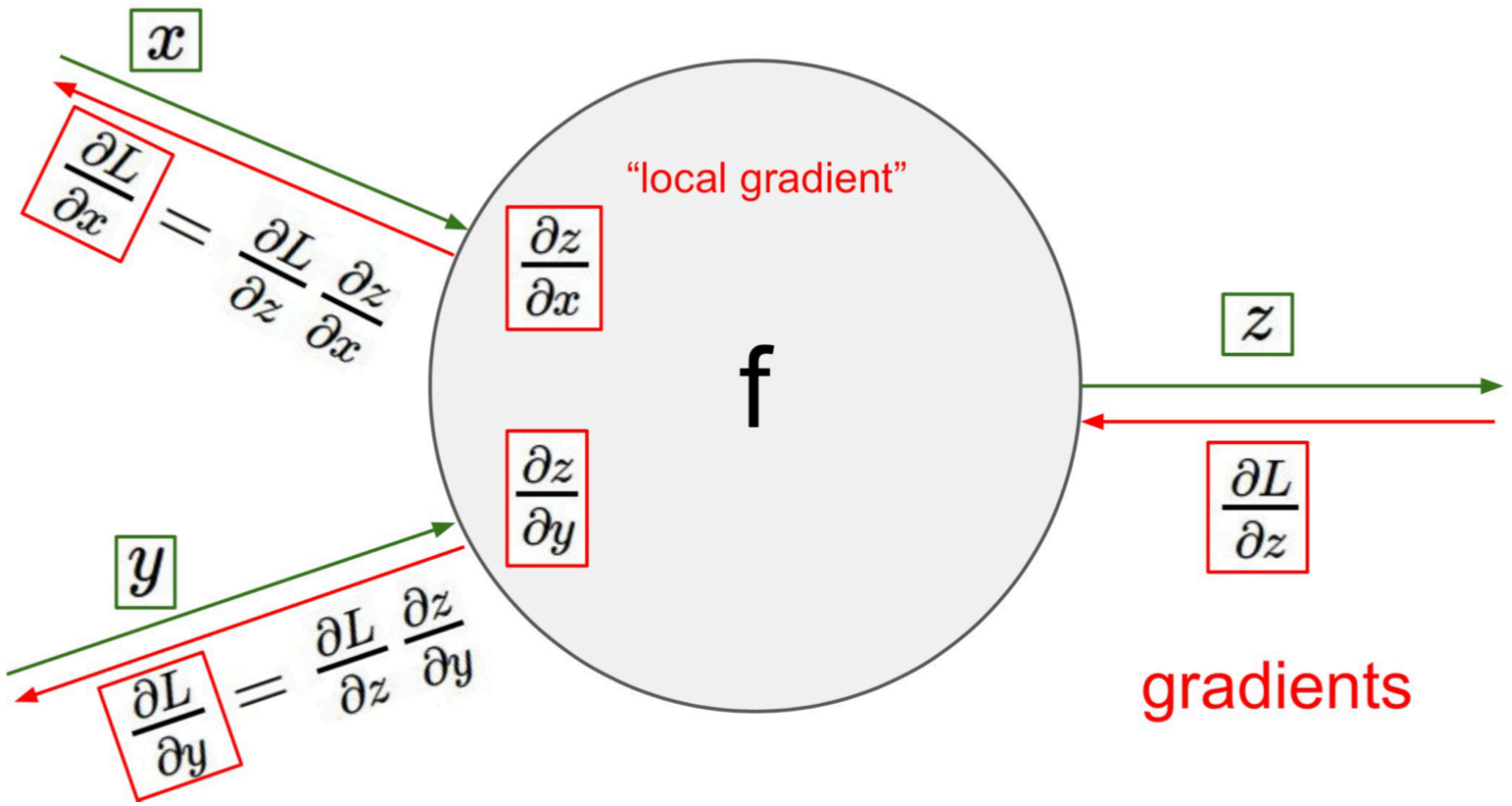
- Gradient descent review: gradient descent updates parameters (weights) in the direction of steepest descent.
- Weight space for an MLP: One coordinate for each weight or bias in the networks in all layers.
- Conceptually, not any different from what we've seen so far, just higher dimensions



Learning MLPs

Back propagation

- We need to compute the partial derivative of the cost function with respect to all weights.
- We will not cover details of how back propagation is done, but we will look into an example.



Learning MLPs

Example: Forward pass

Network:

$$f(x, y, z) = (x + y) \times z$$

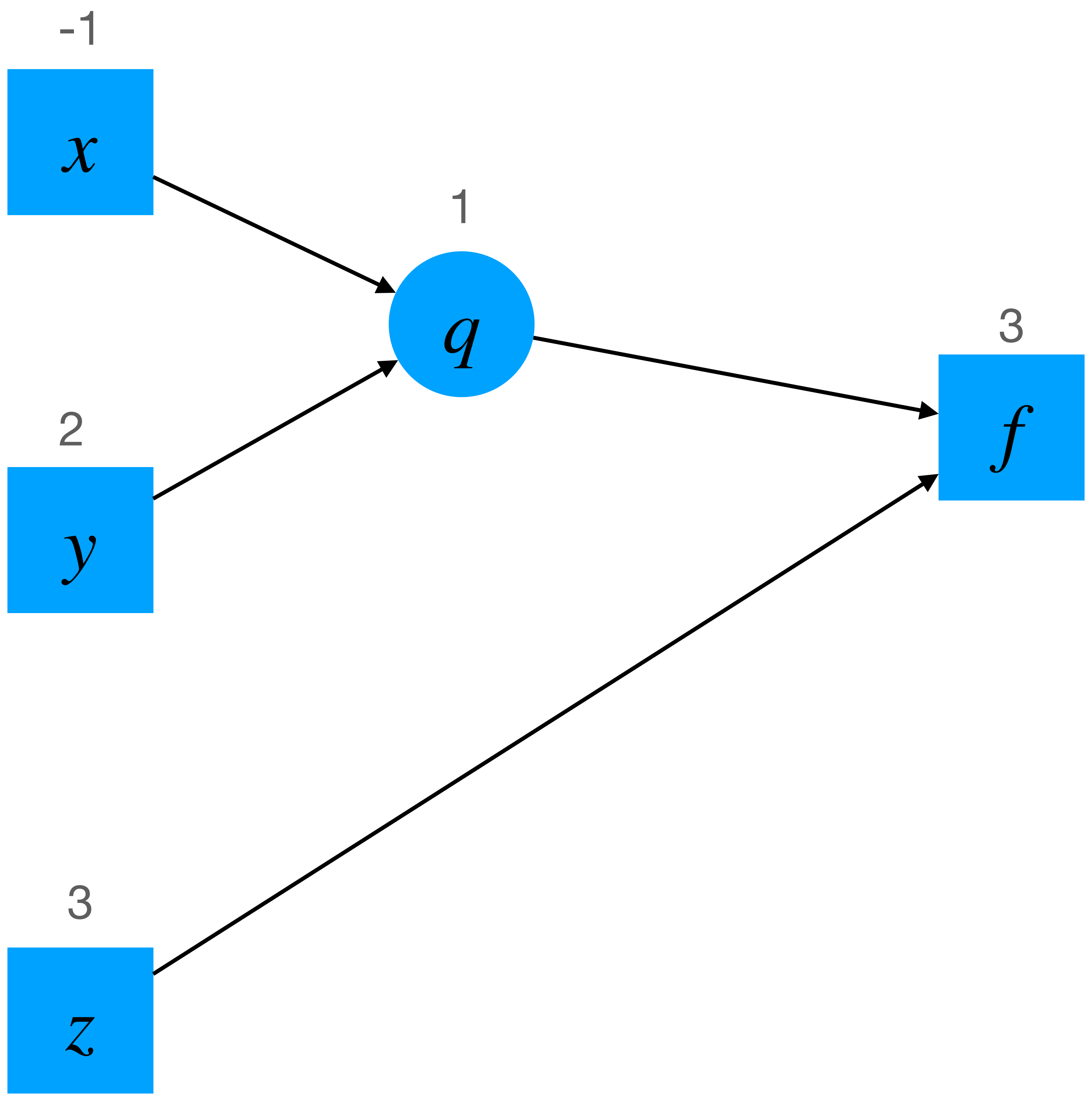
$$q = x + y; f = q \times z$$

Input:

$$x = -1, \quad y = 2 \quad , z = 3$$

Forward propagate: calculate q, f

Backward propagate: calculate $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Learning MLPs

Example: Backward pass

Network:

$$f(x, y, z) = (x + y) \times z$$

$$q = x + y; f = q \times z$$

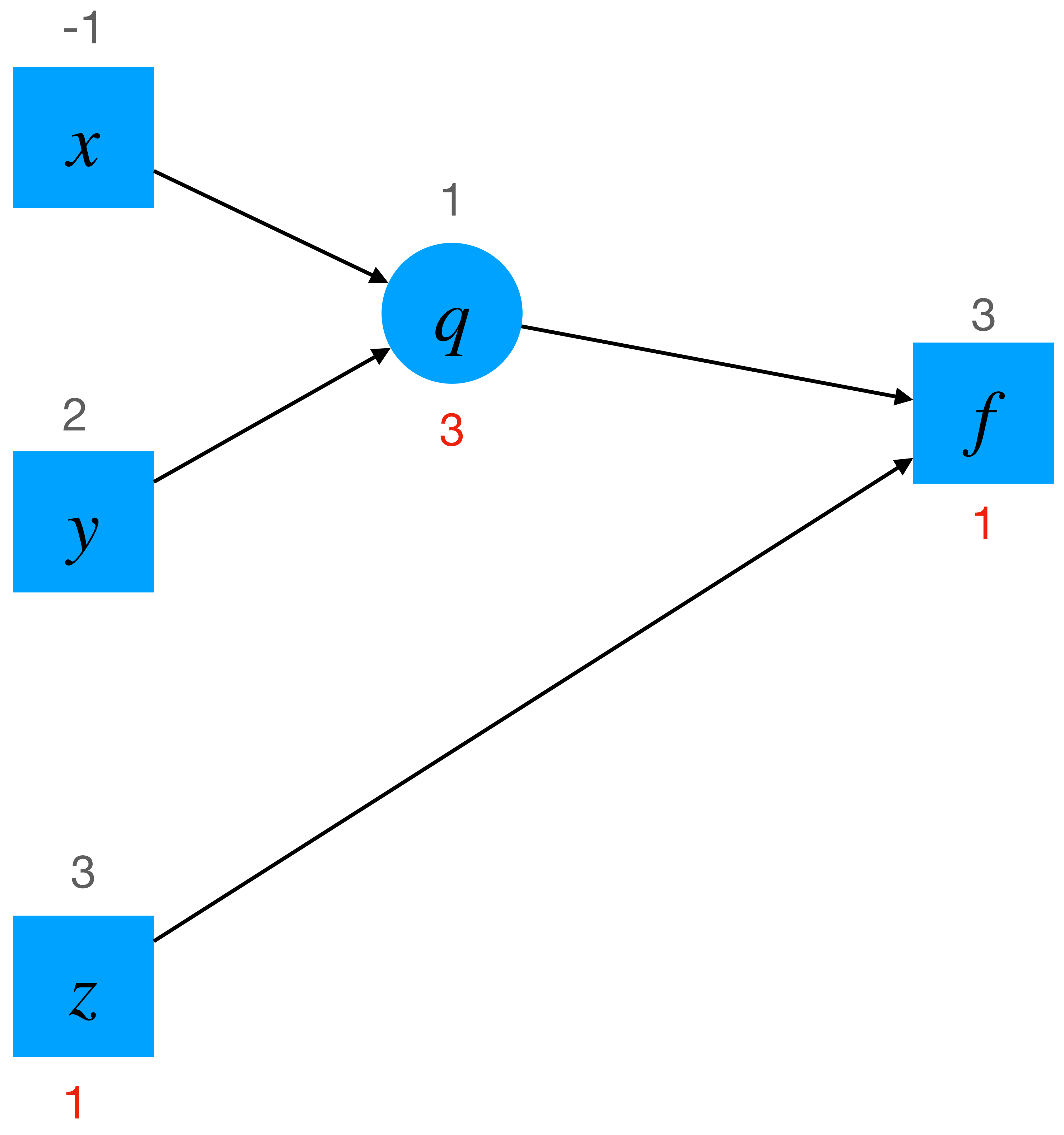
Input:

$$x = -1, \quad y = 2 \quad , z = 3$$

$$\frac{\partial f}{\partial f} = 1$$

$$\frac{\partial f}{\partial q} = \frac{\partial (qz)}{\partial q} = z = 3$$

$$\frac{\partial f}{\partial z} = \frac{\partial (qz)}{\partial z} = q = 1$$



Learning MLPs

Example: Backward pass

Network:

$$f(x, y, z) = (x + y) \times z$$

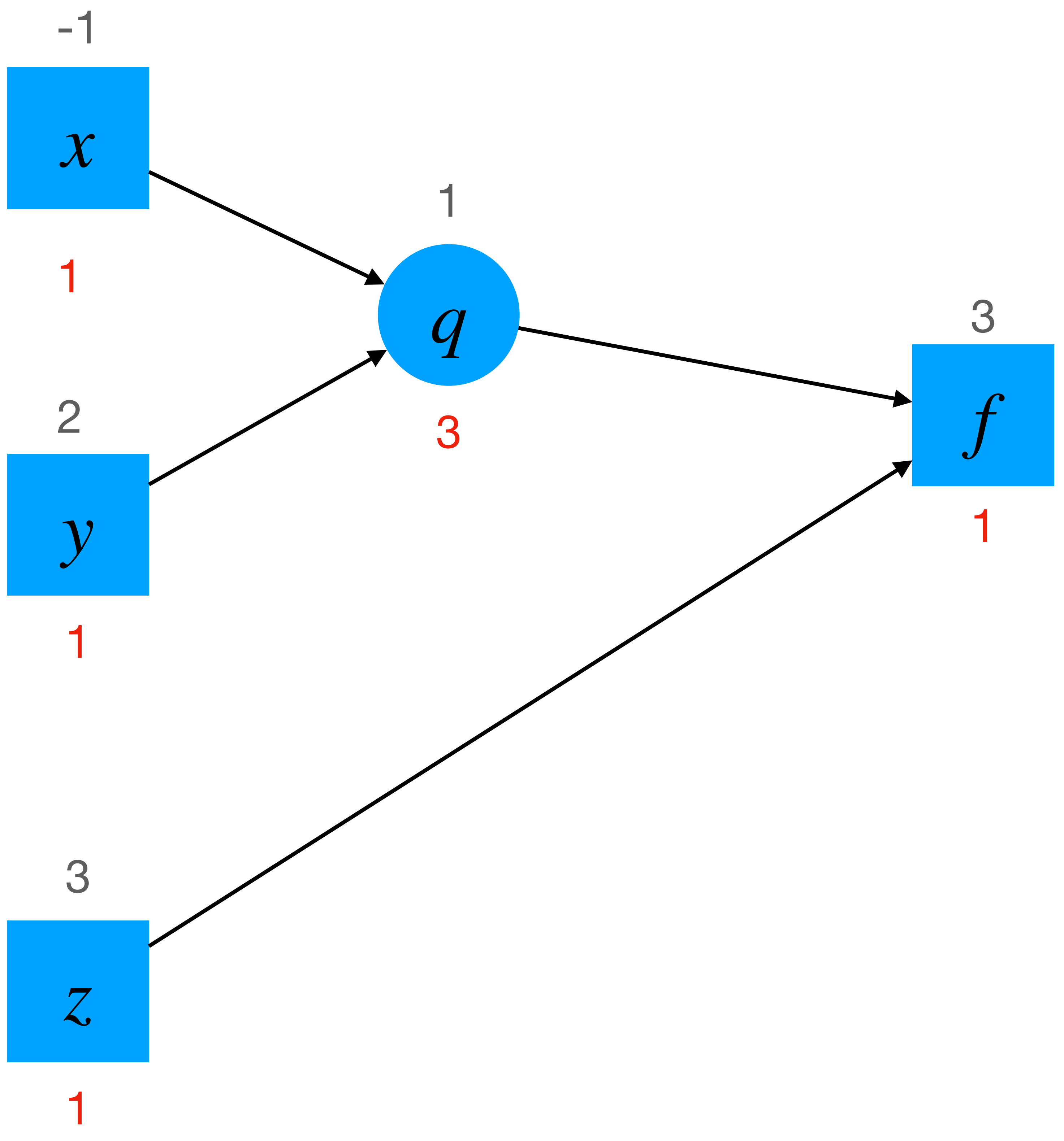
$$q = x + y; f = q \times z$$

Input:

$$x = -1, \quad y = 2 \quad , z = 3$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = z \cdot 1 = 3$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = z \cdot 1 = 3$$



Learning MLPs

Forward/backward pass

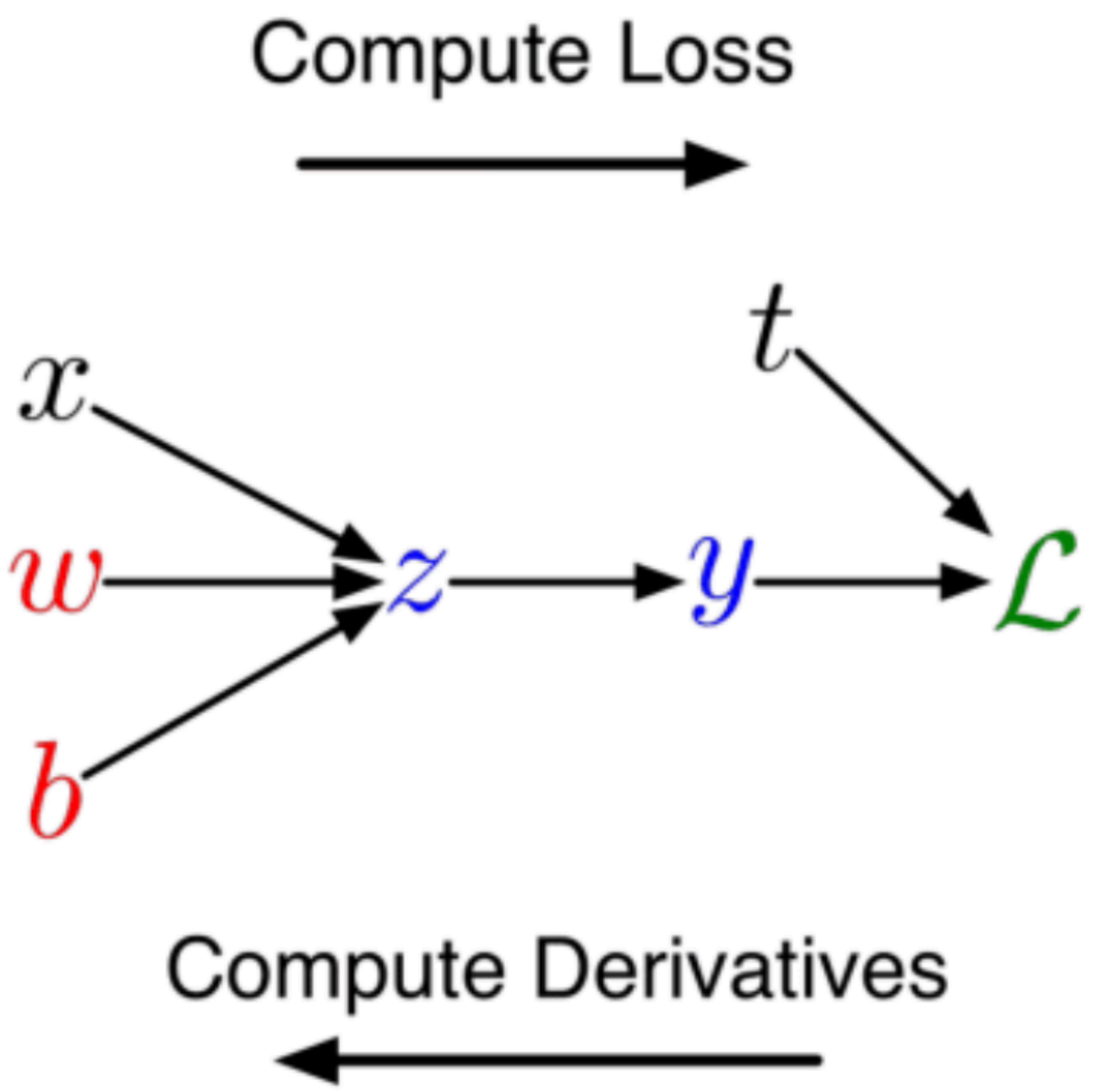
Computing the derivatives:

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y} = y - t$$

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y}\,\sigma'(z)$$

$$\frac{\partial\mathcal{L}}{\partial w} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}\,x$$

$$\frac{\partial\mathcal{L}}{\partial b} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}z}$$



Computing the loss:

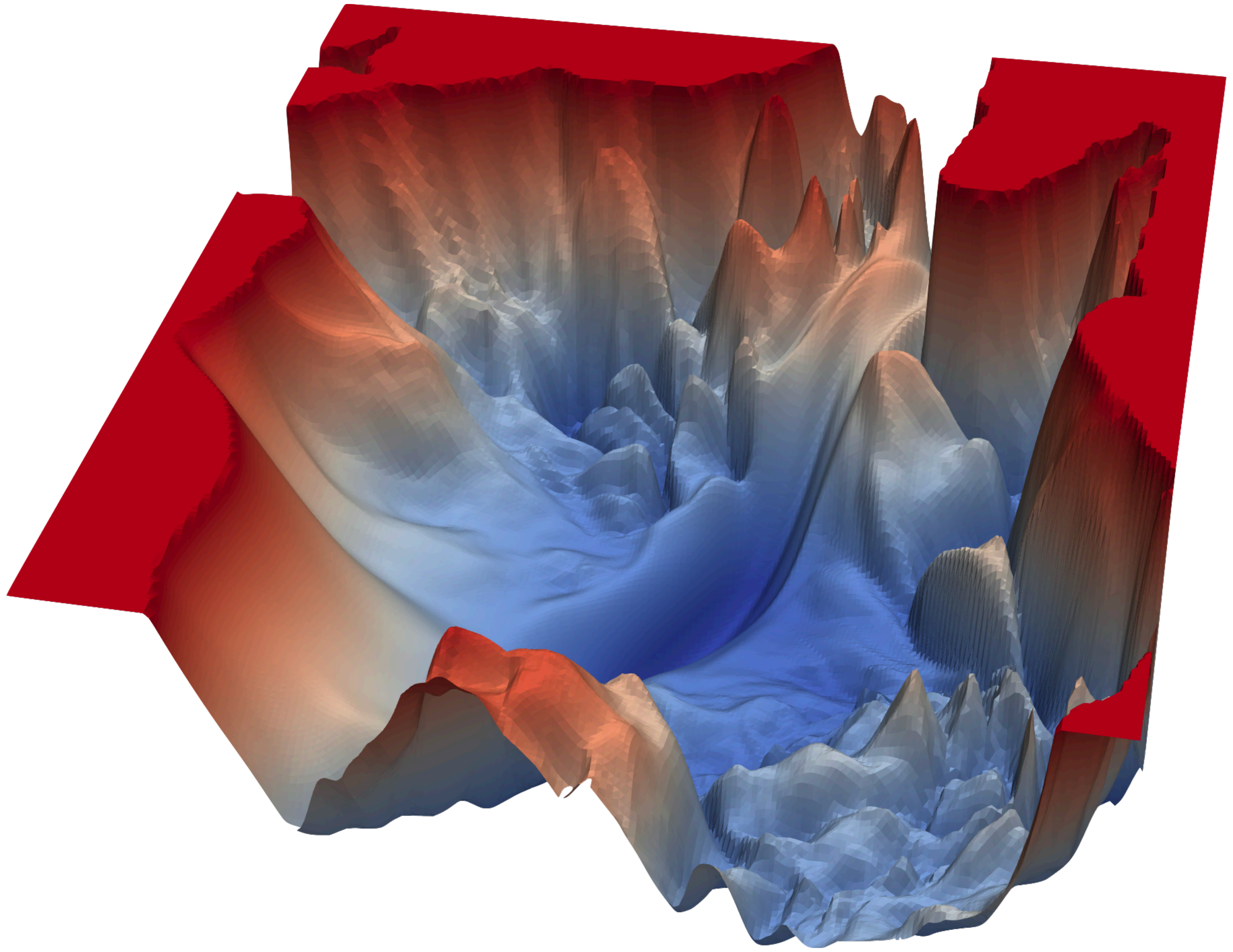
$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Learning MLPs
Practical considerations

- Training neural networks is complicated in practice!
  - The landscape of deep neural network loss function is extremely complex
  - Setting the learning rate is challenging



How to properly set the learning rate?

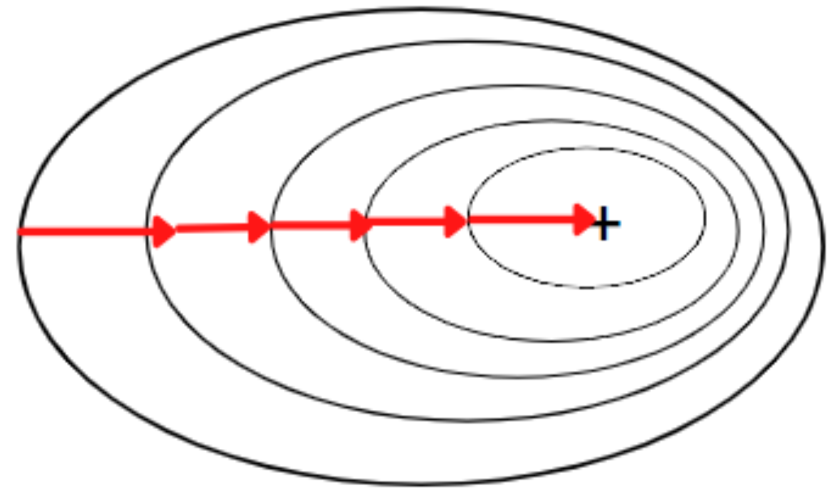
- Try different values and find what works best
- Design an adaptive learning rate that "adapts" to the landscape

Learning MLPs

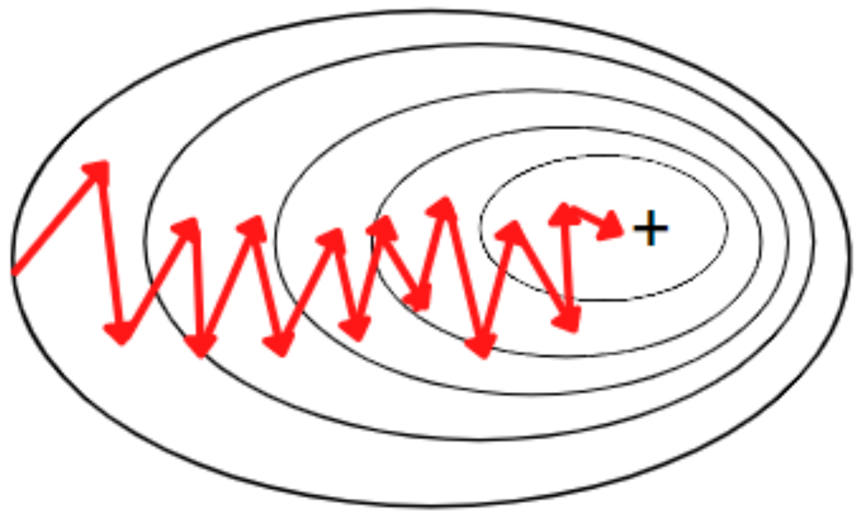
Practical considerations

- **Batch Gradient descent**: Compute the gradient over the entire dataset
  - Computationally expensive, and sometime impossible.
- **Stochastic gradient descent**: compute the gradient over a single sample
  - Fast, but stochastic and noisy
- **Mini-batch Gradient descent:** Computing the gradient over a mini batch of samples)
  - Smoother convergence
  - Allows for larger learning rate
  - Use parallel processing

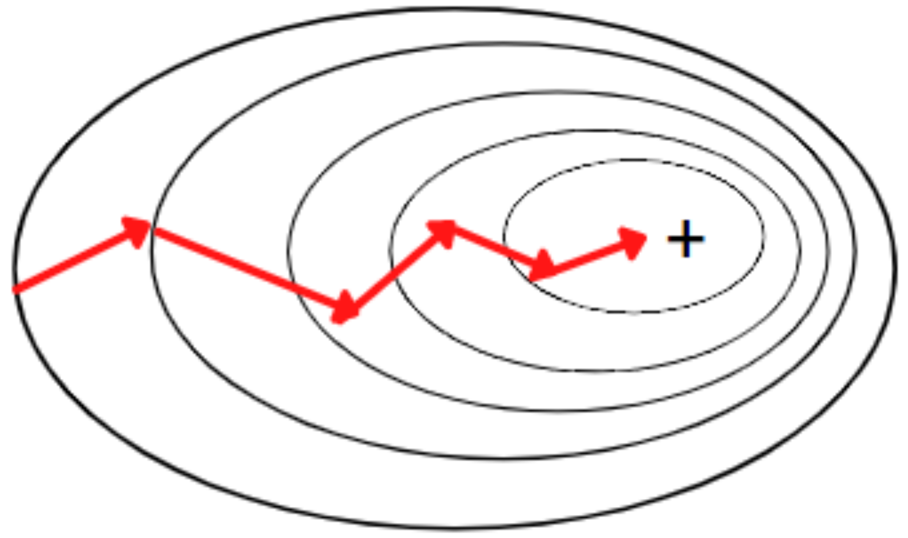
Batch Gradient Descent



Stochastic Gradient Descent



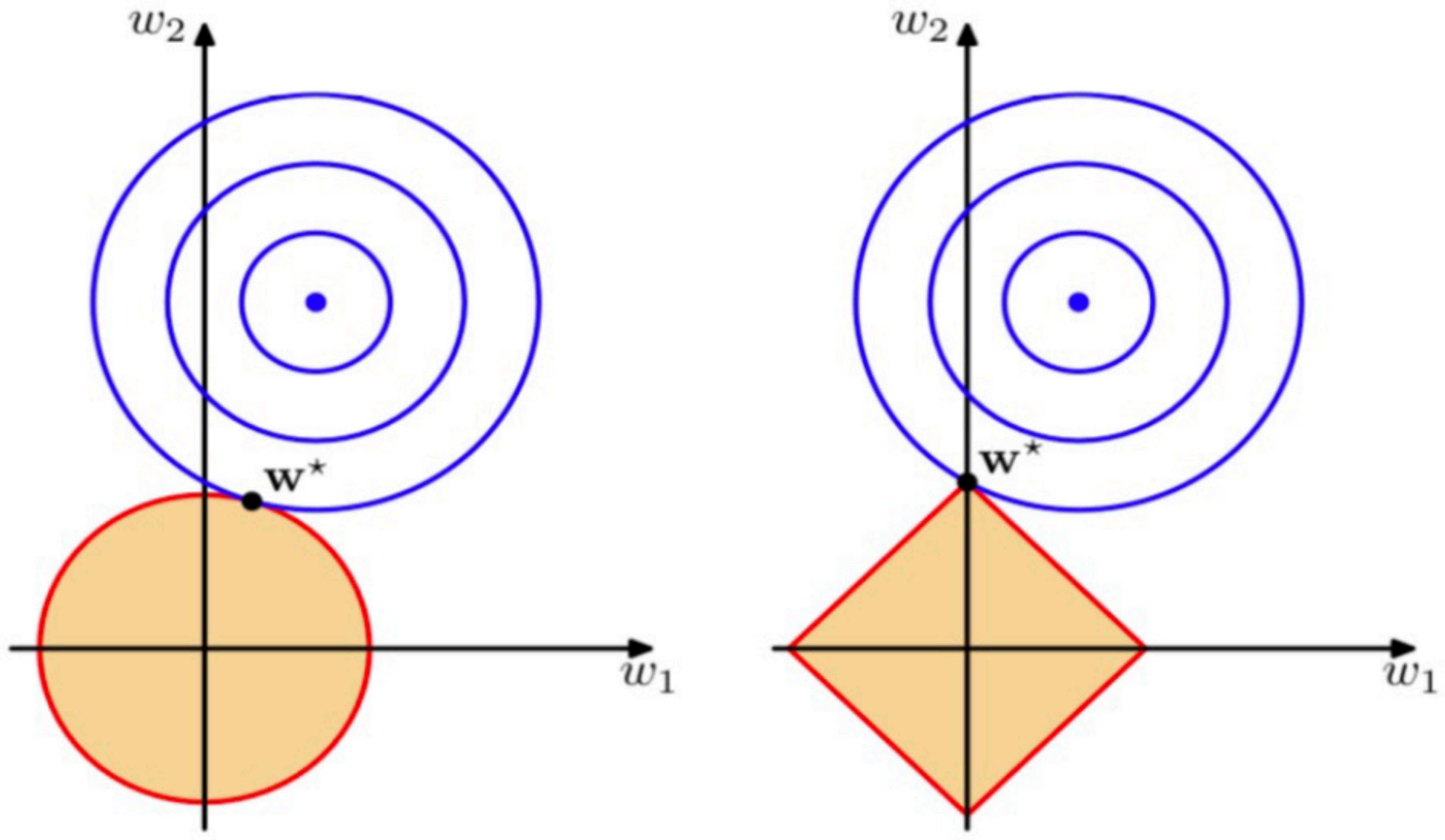
Mini-Batch Gradient Descent



Learning MLPs

Regularization

- L1/L2 regularization



L2 regularization

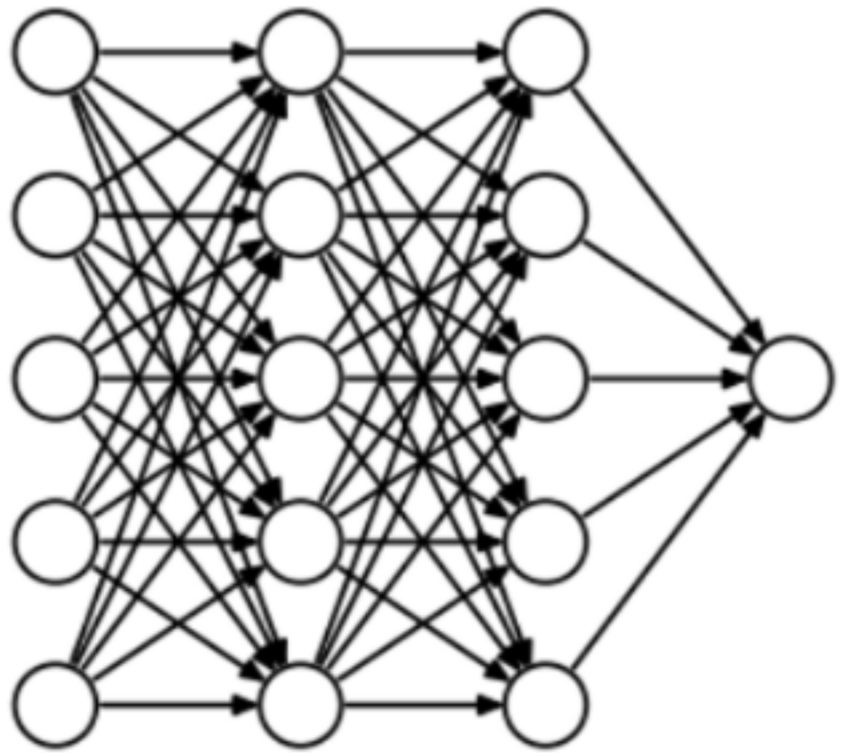
$$\mathcal{R} = \sum_i w_i^2$$

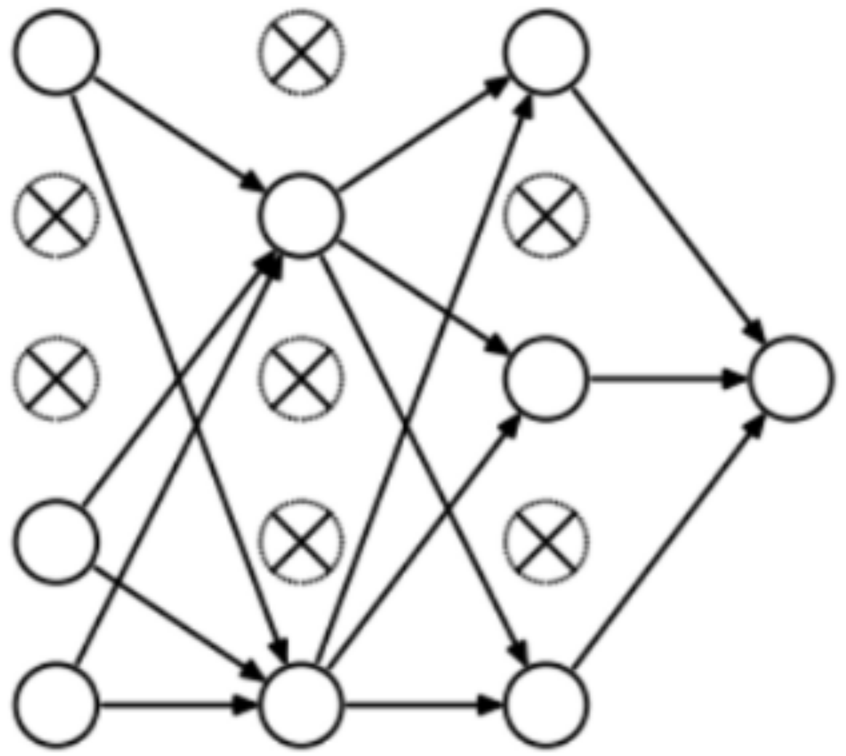
L1 regularization

$$\mathcal{R} = \sum_i |w_i|$$

Learning MLPs
Regularization

- Dropout
  - drop a percentage of activations in layers
  - Forces network not to rely on any node in particular

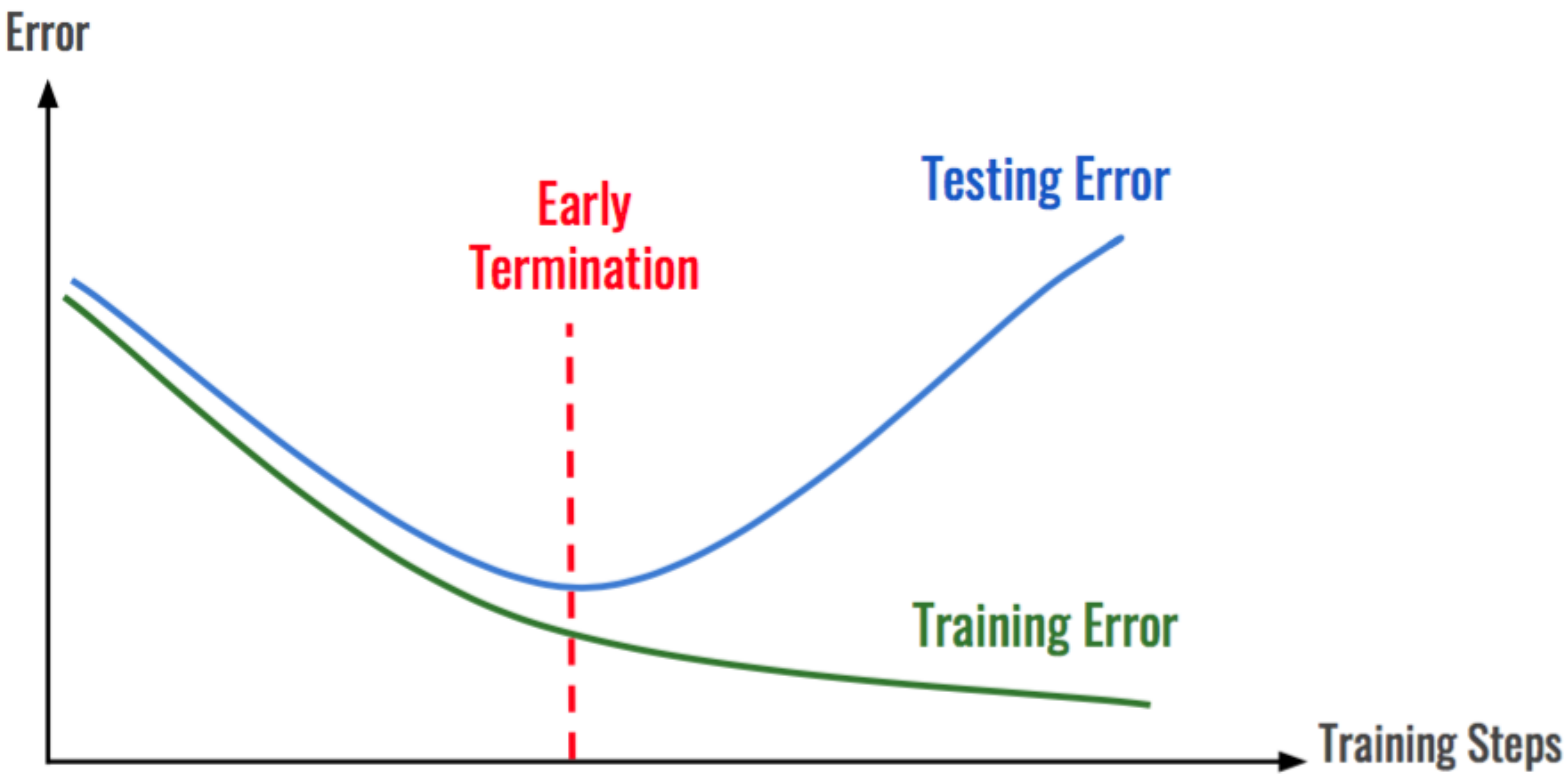




Learning MLPs

Regularization

- Early stopping: Terminate the gradient updates as you start overfitting.



Conclusion

Neural networks (MLP)

- Multilayer neural networks (Multilayer perceptrons) can learn complex relationship between its input and output, using a network of linear functions and non-linear activation functions.
- Non-linear functions are a fundamental part of MLPs that enable modelling non-linear functions.
- We learn the optimal weights for MLPs using the backward propagation procedure.
- Talked about tips for training neural networks in practice
- Next lecture:
    - Ensemble models
    - Python tutorial for supervised learning