

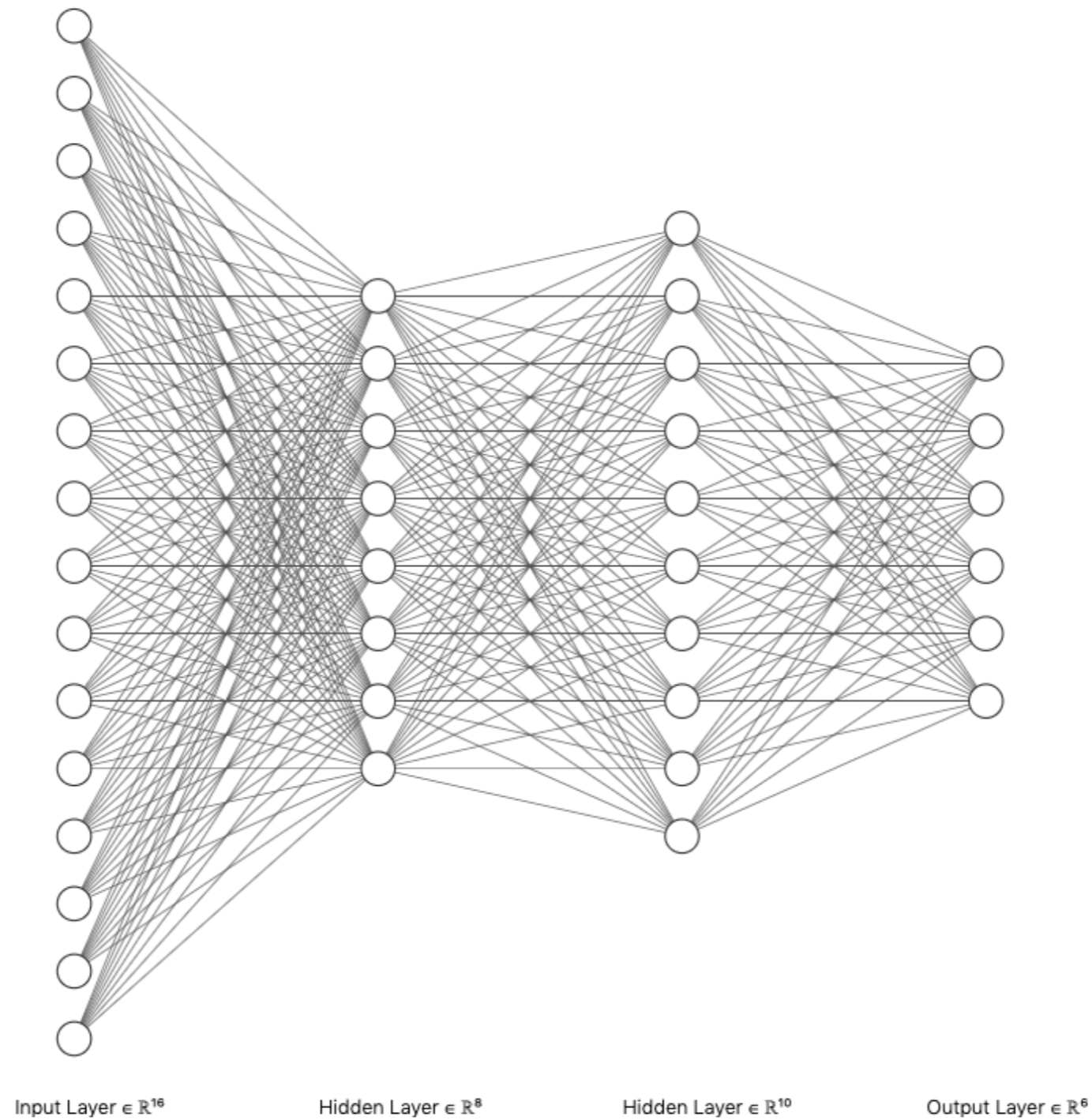
Autonomous and Adaptive Systems

Introduction to Deep Learning and Neural Architectures III

Mirco Musolesi

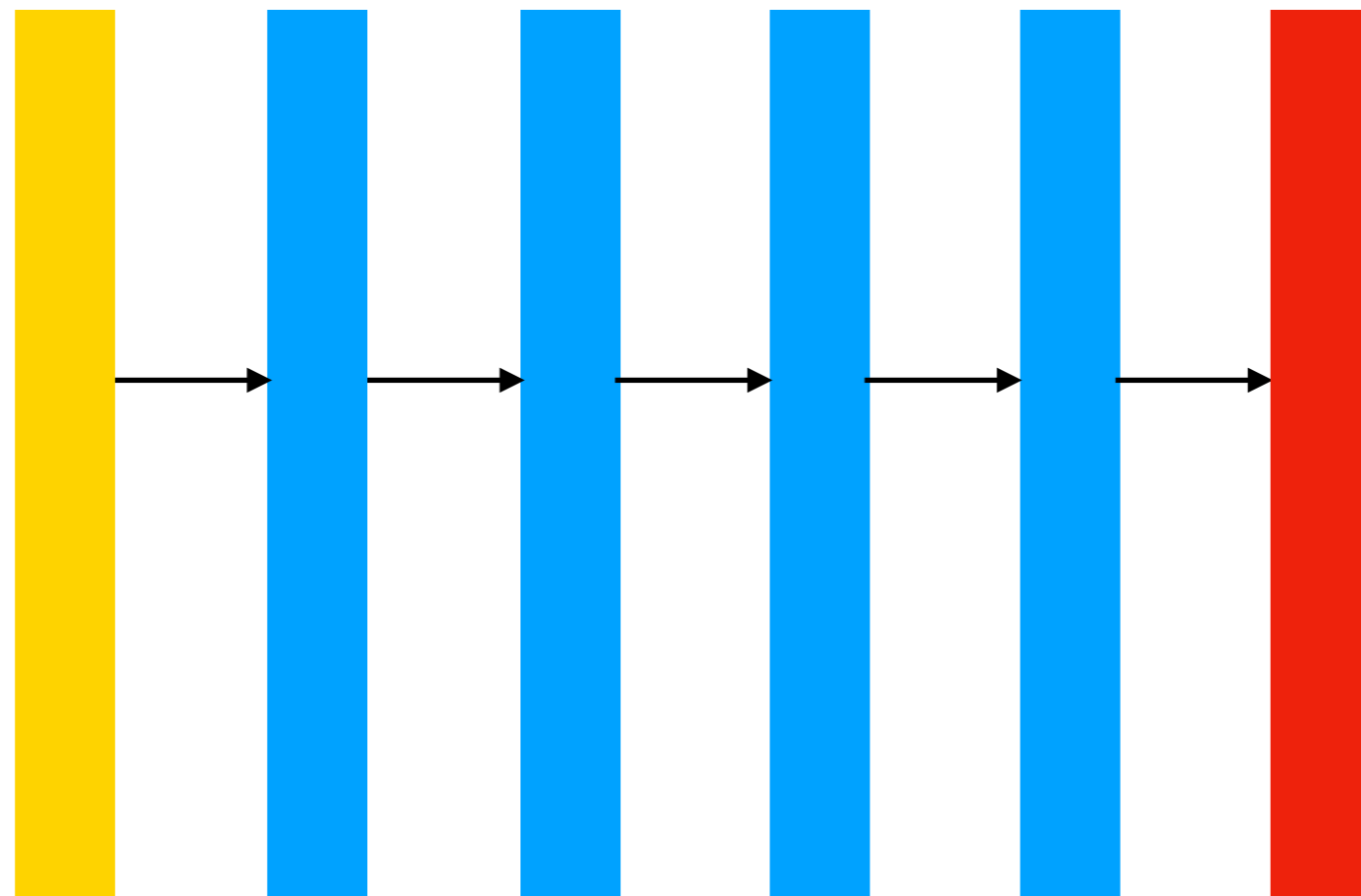
mircomusolesi@acm.org

Deep Neural Networks



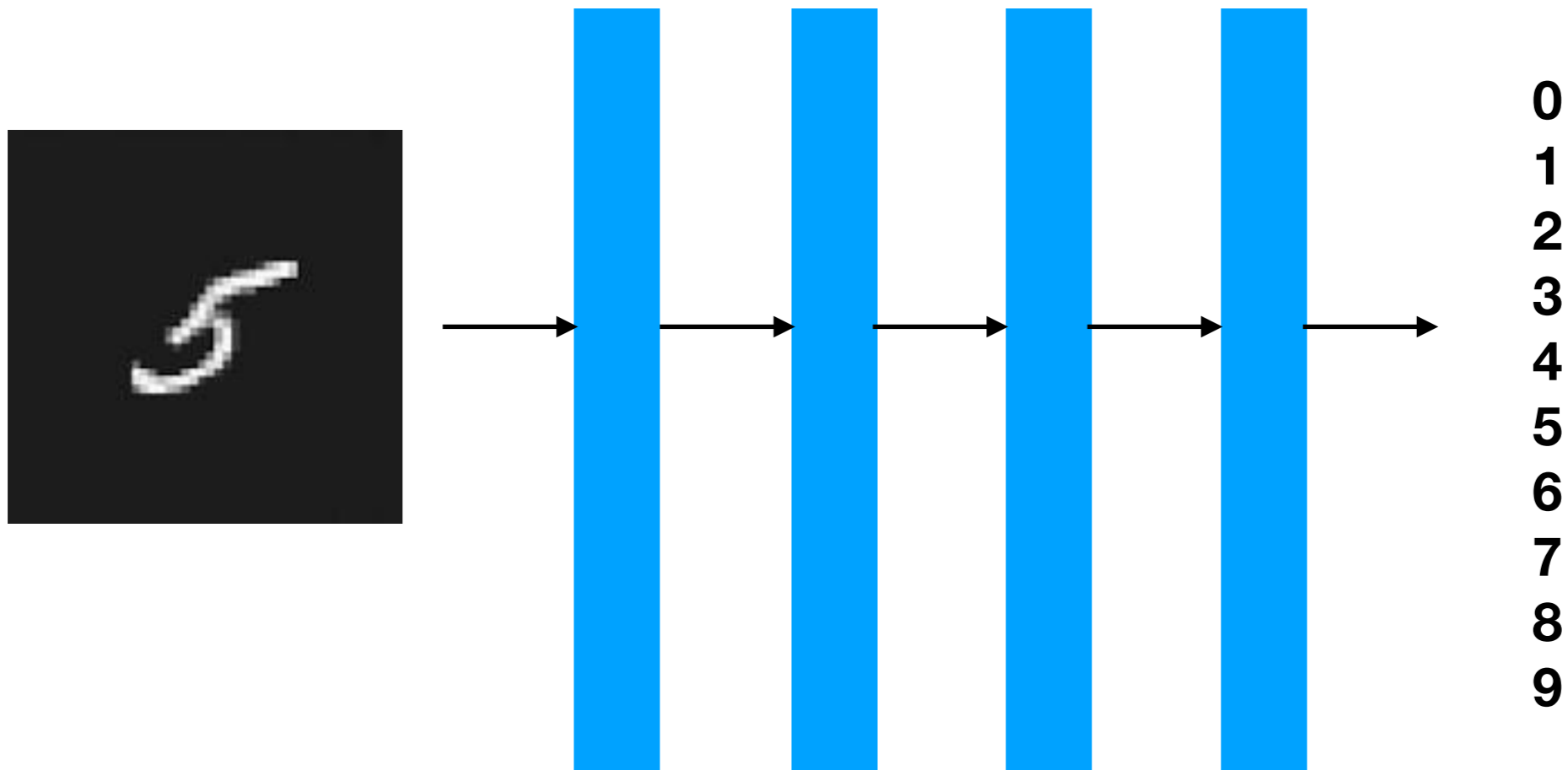
Deep Neural Networks

Inputs Layer 1 Layer 2 Layer 3 Layer 4 Outputs



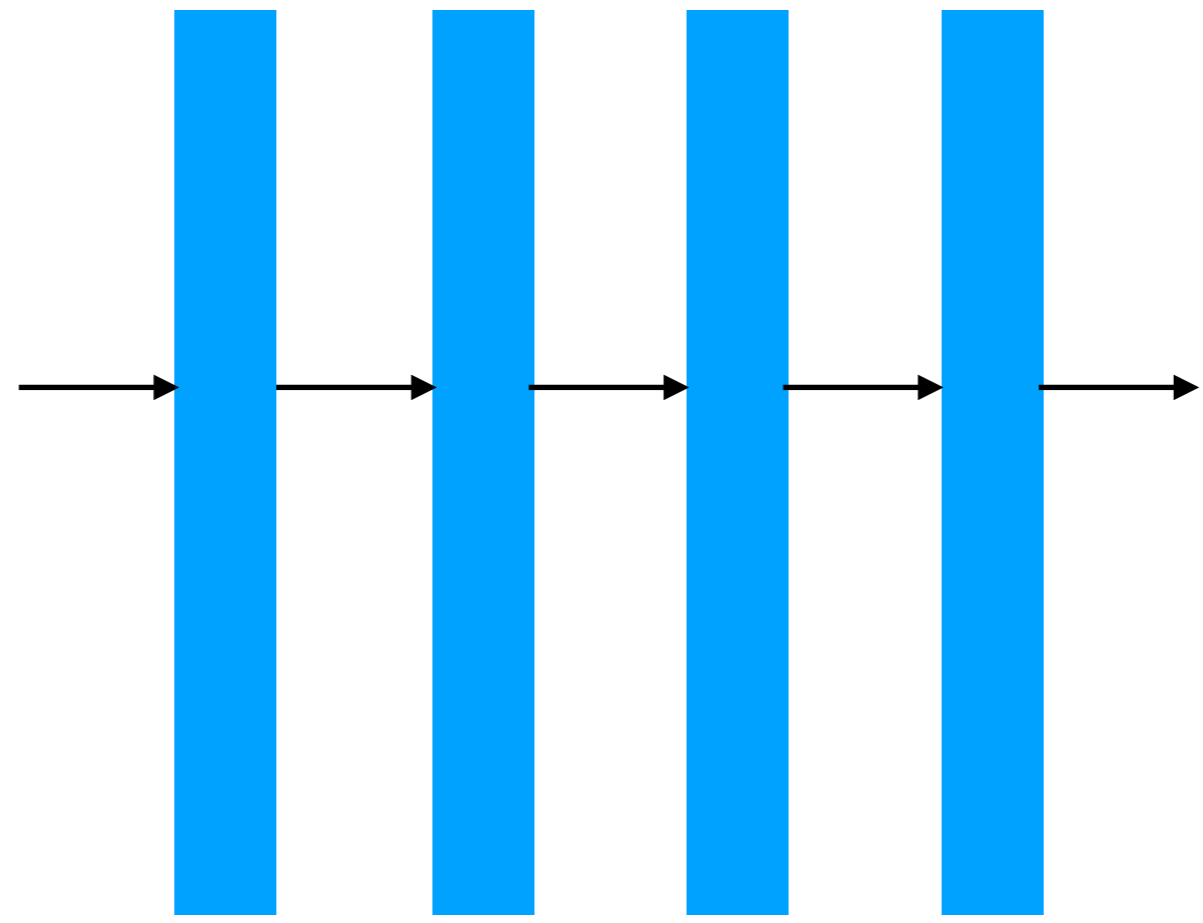
Deep Neural Networks

Inputs Layer 1 Layer 2 Layer 3 Layer 4 Outputs



Deep Neural Networks

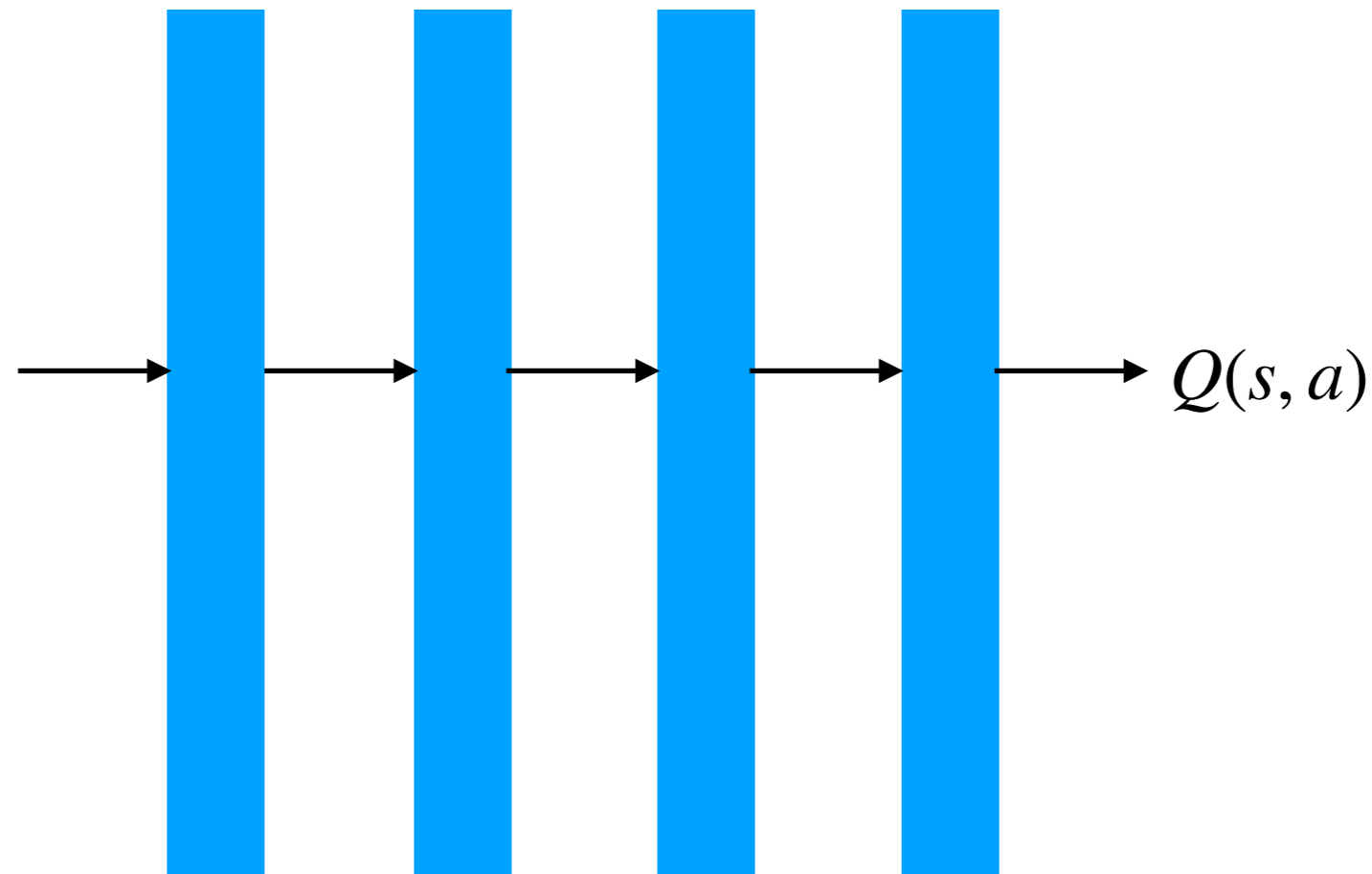
Inputs Layer 1 Layer 2 Layer 3 Layer 4 Outputs



0 0
1 0.05
2 0.1
3 0.05
4 0.1
5 0.5
6 0.05
7 0.05
8 0
9 0

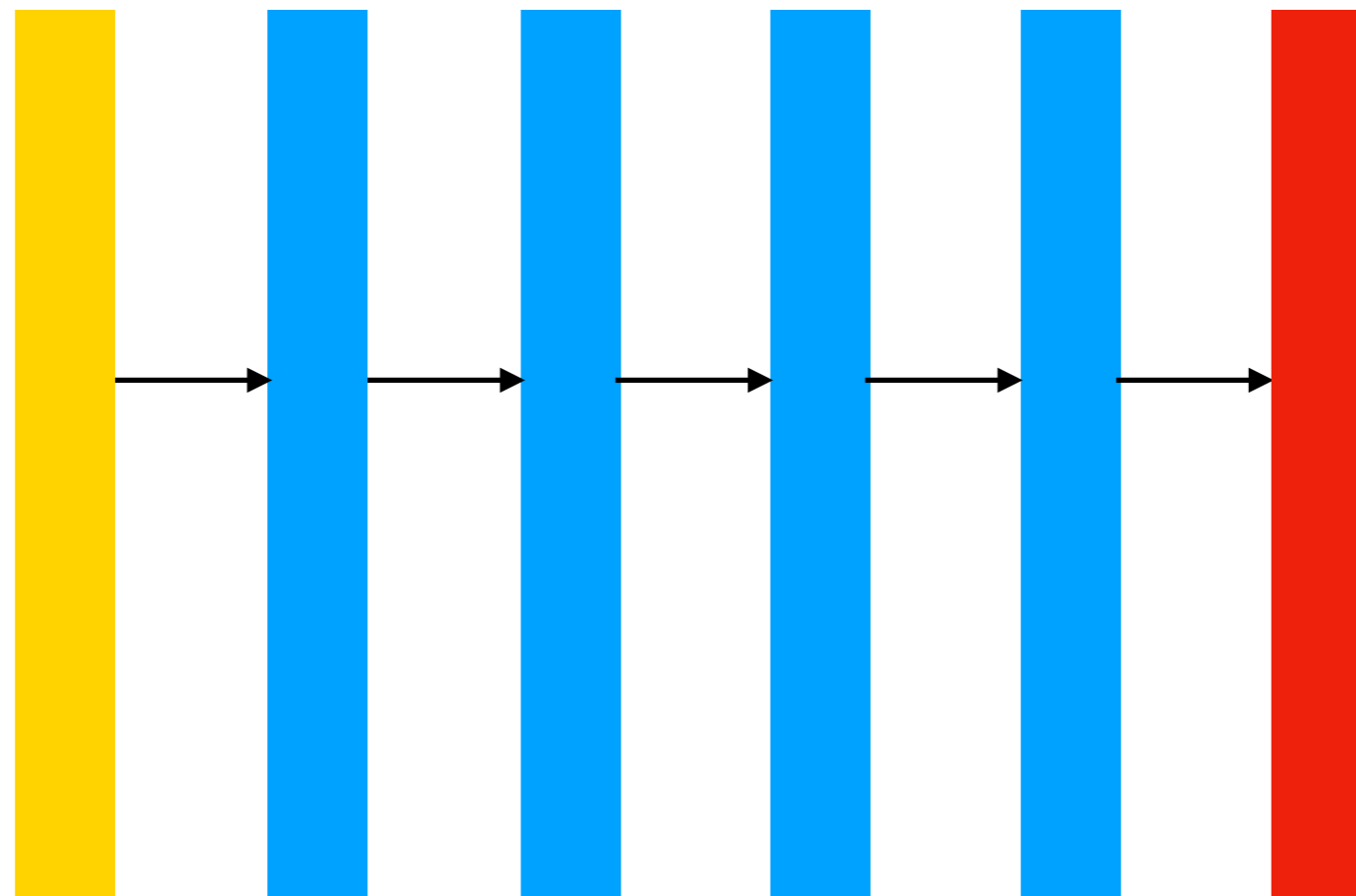
Deep Neural Networks

Inputs Layer 1 Layer 1 Layer 1 Layer 1 Outputs



Deep Neural Networks

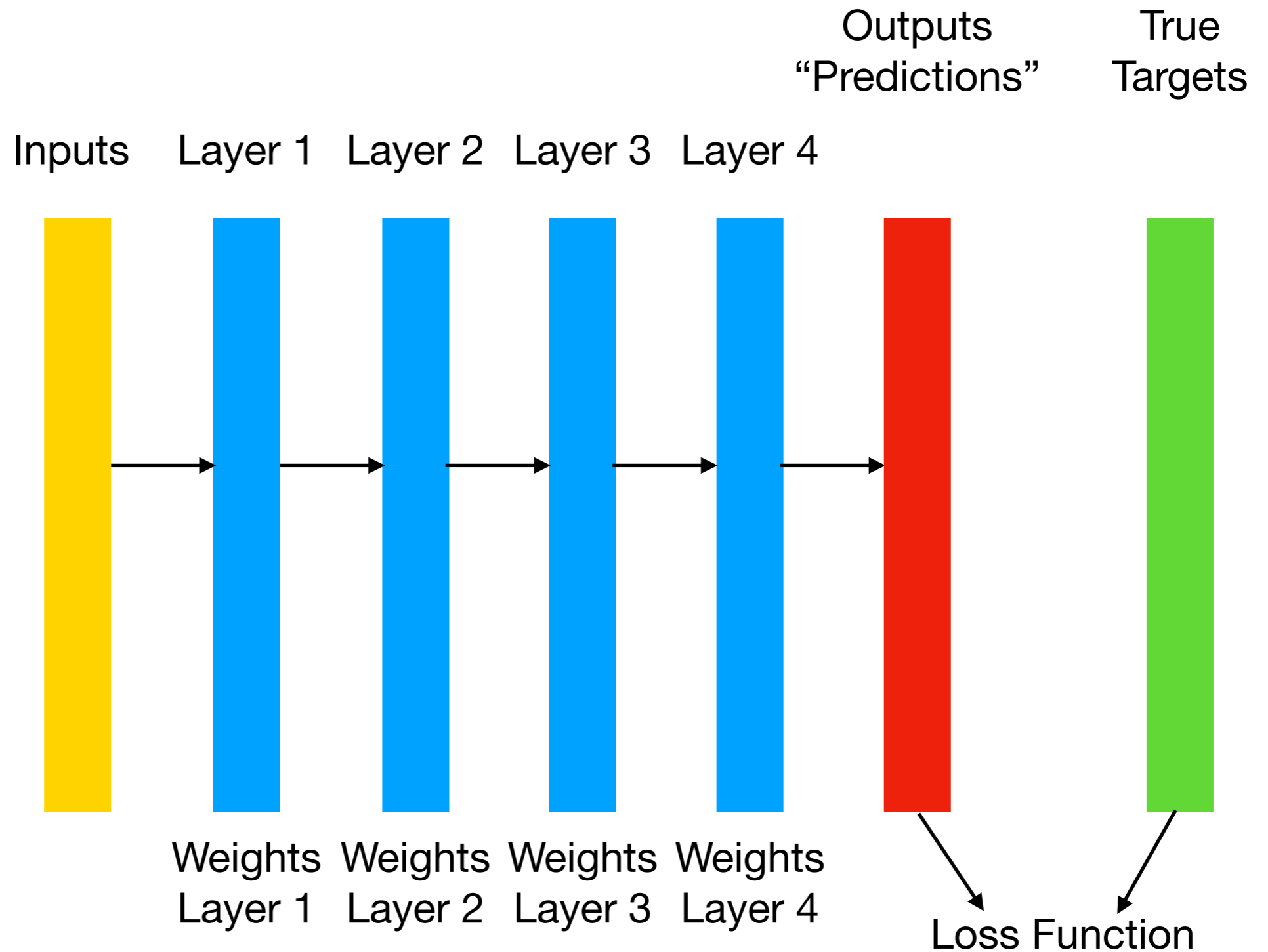
Inputs Layer 1 Layer 1 Layer 1 Layer 1 Outputs



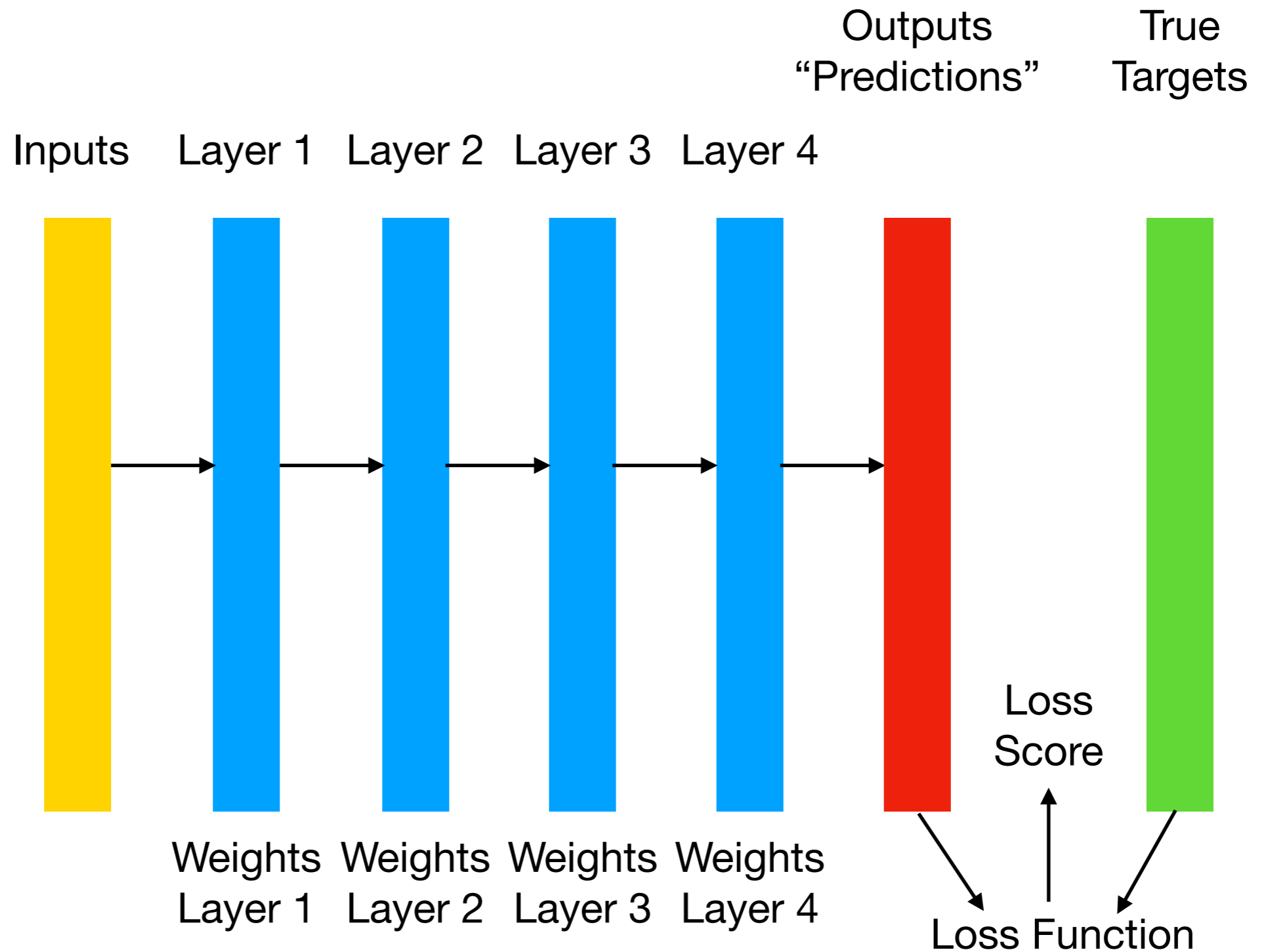
Weights Weights Weights Weights
Layer 1 Layer 2 Layer 3 Layer 4

The goal is to find the right values for these weights.

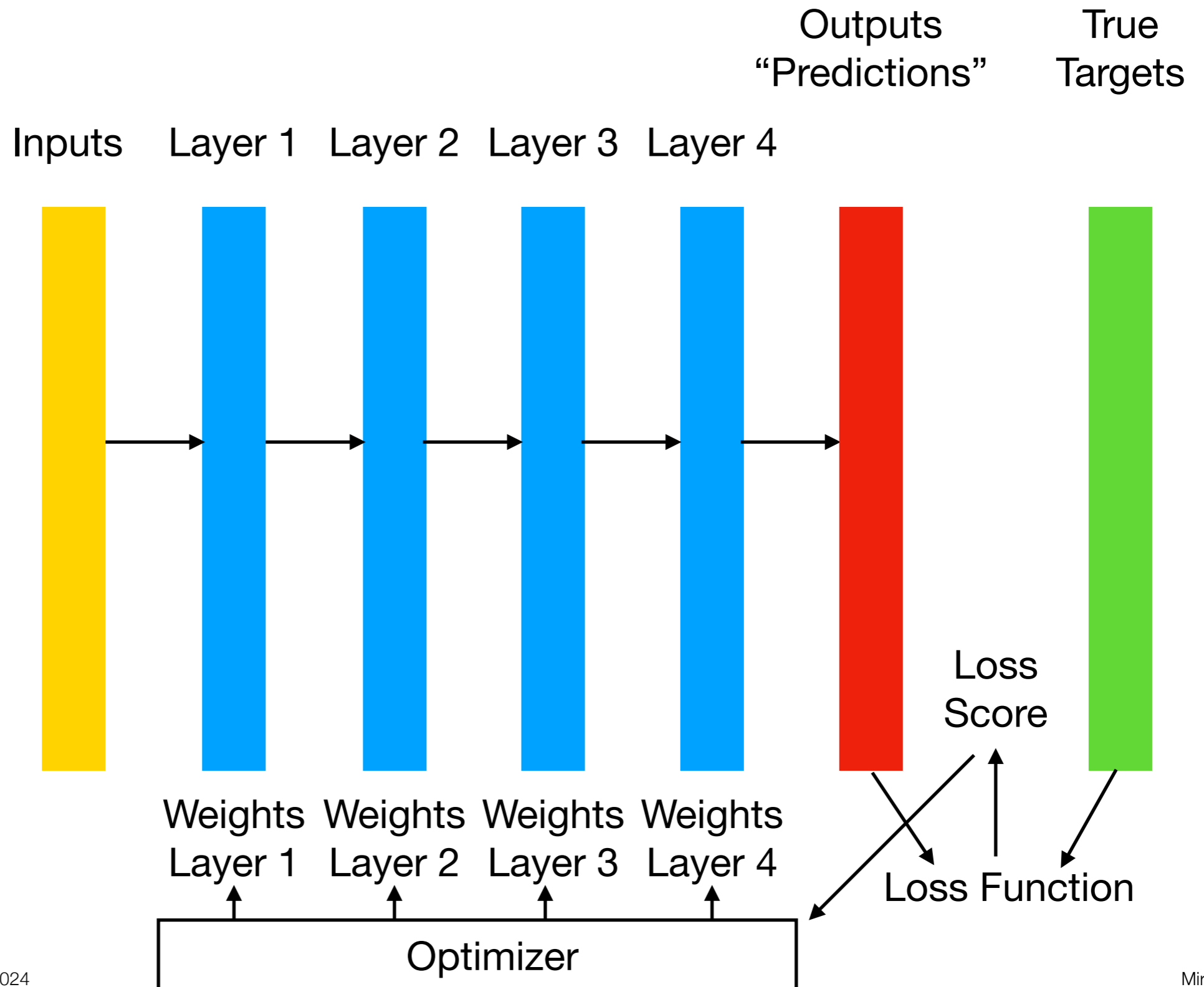
Deep Neural Networks



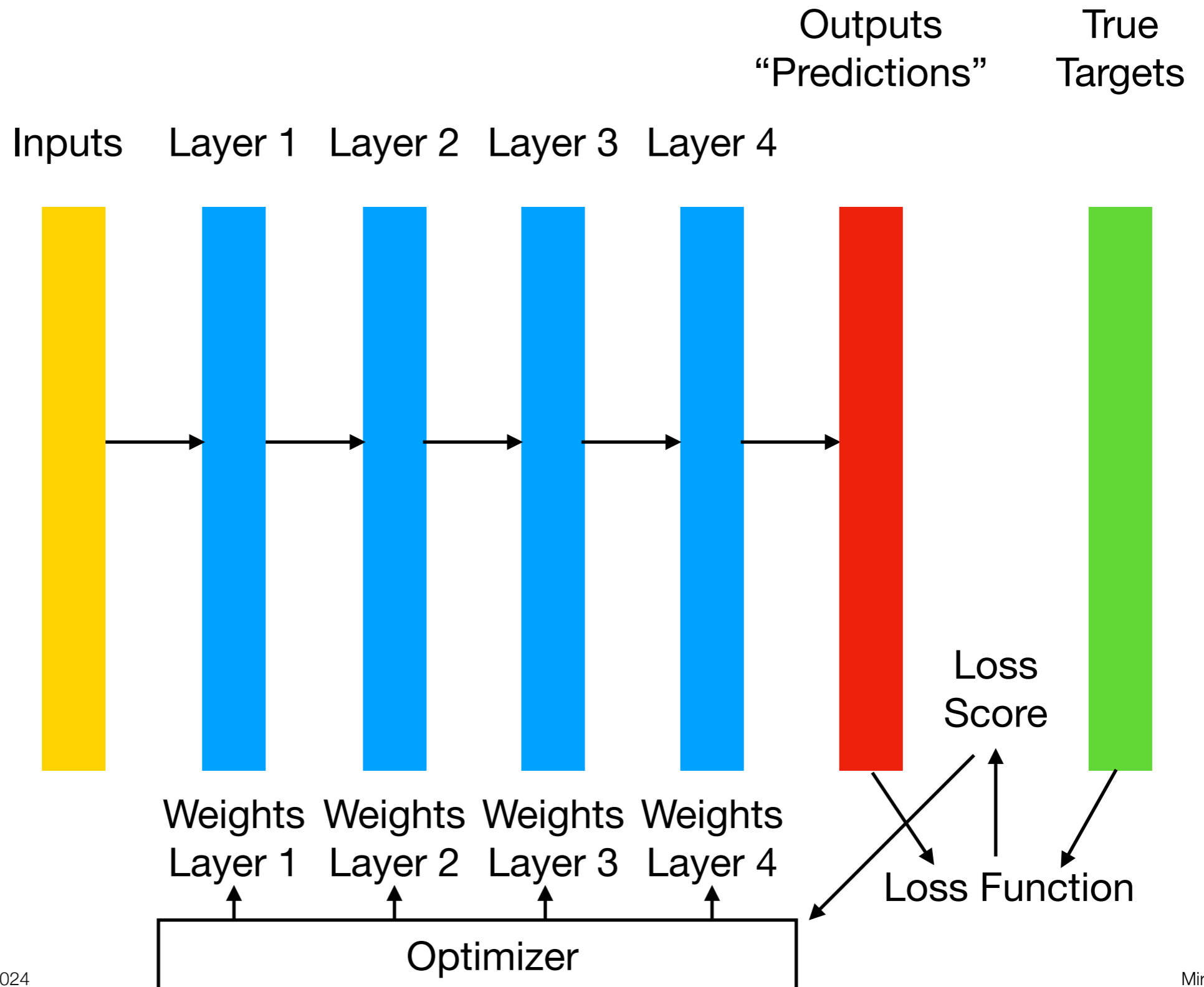
Deep Neural Networks



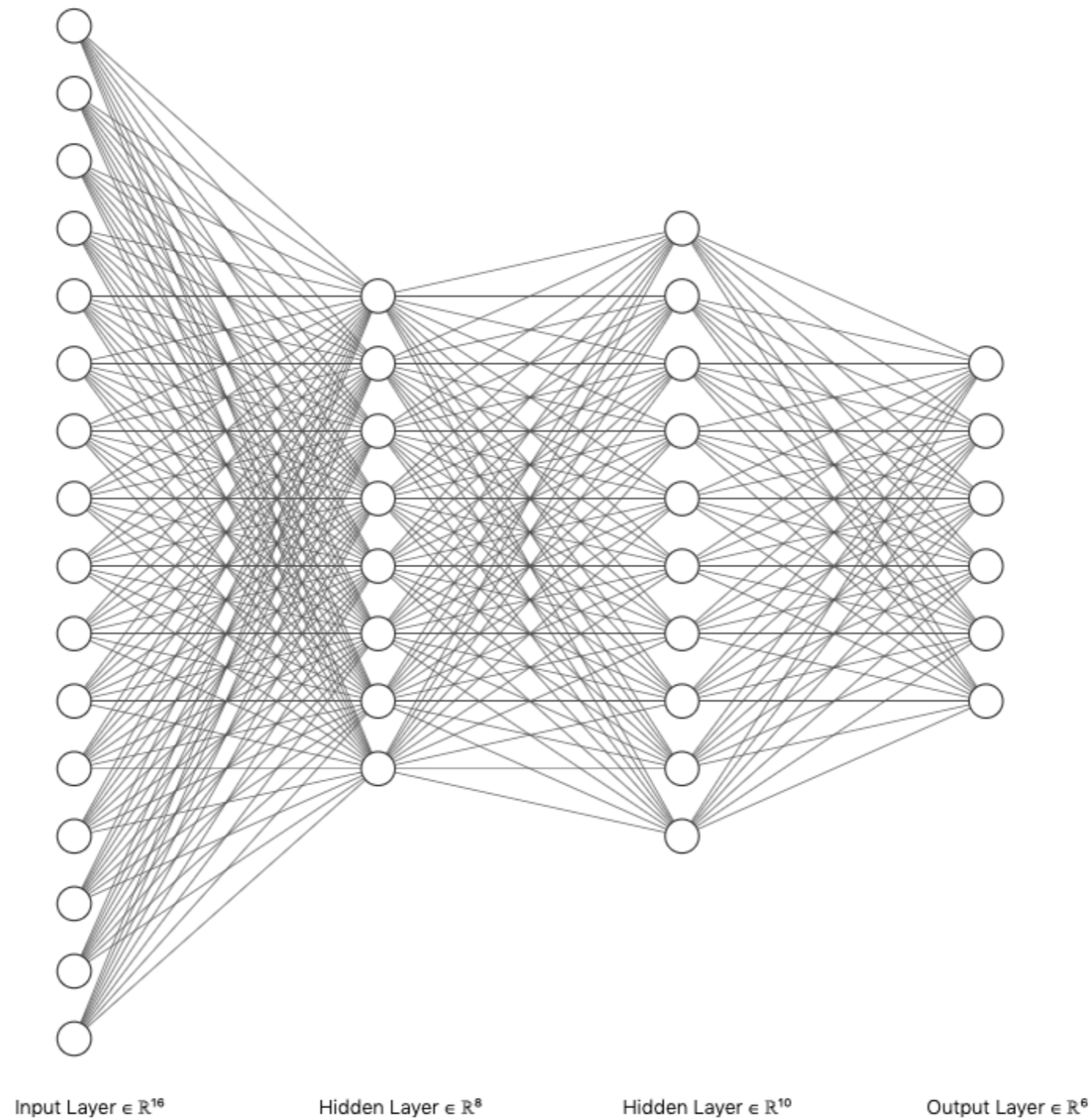
Deep Neural Networks



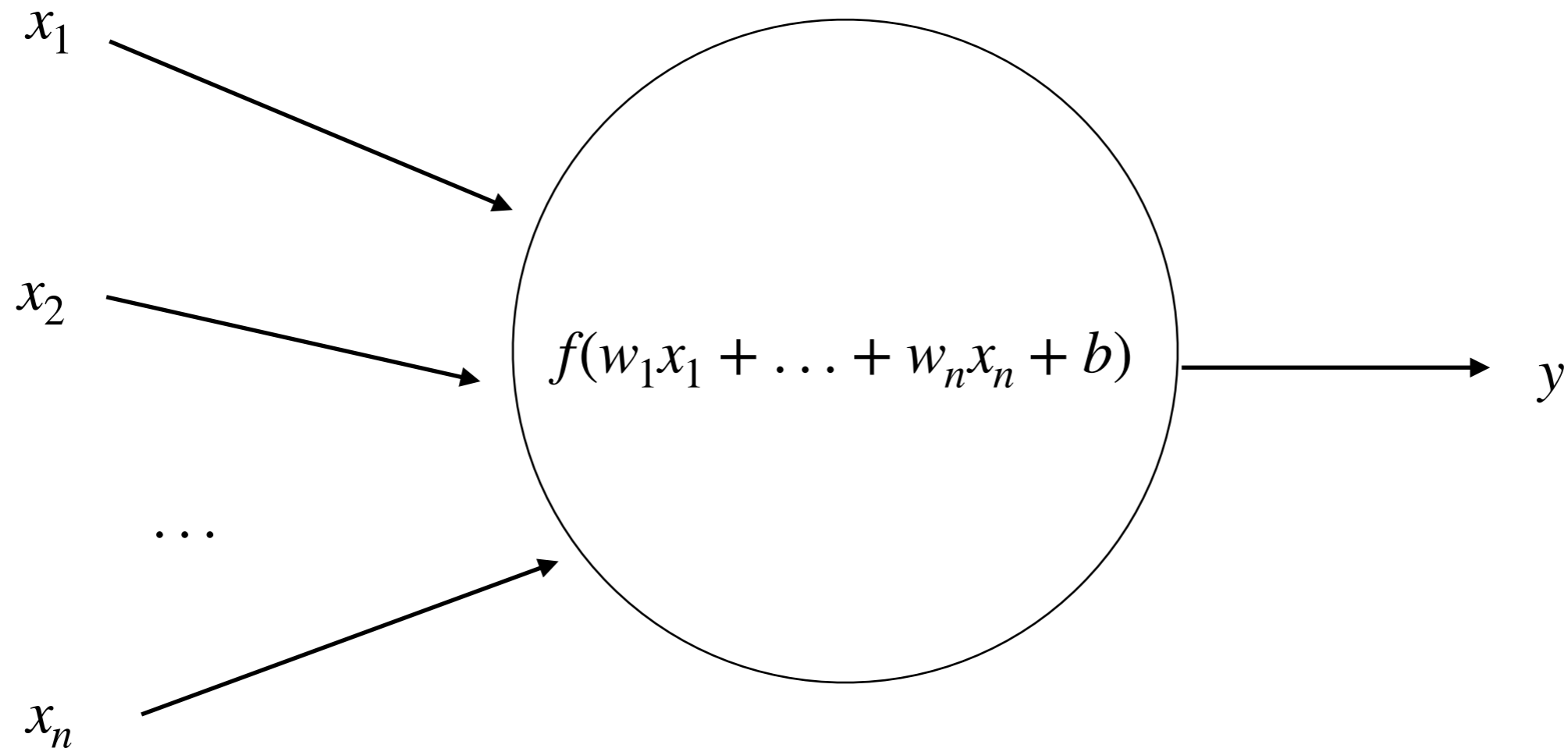
Deep Neural Networks



Deep Neural Networks



Nodes/Units/Neurons



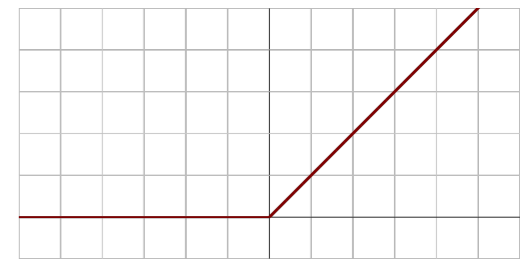
f is called the activation function, b is usually called the bias

Activations Functions

▶ They are generally used to add non-linearity.

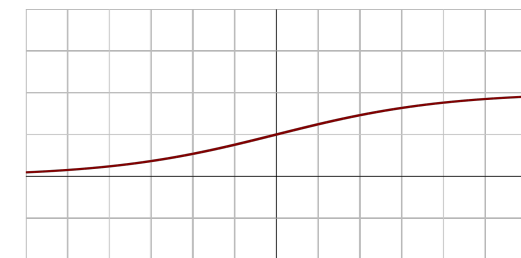
▶ Examples:

▶ *Rectified Linear Unit*: it returns the max between 0 and the value in input. In other words, given the value z in input it returns $\max(0, z)$.

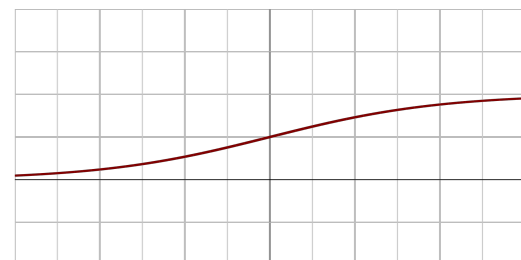


▶ *Logistic sigmoid*: given the value in input z , it returns

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

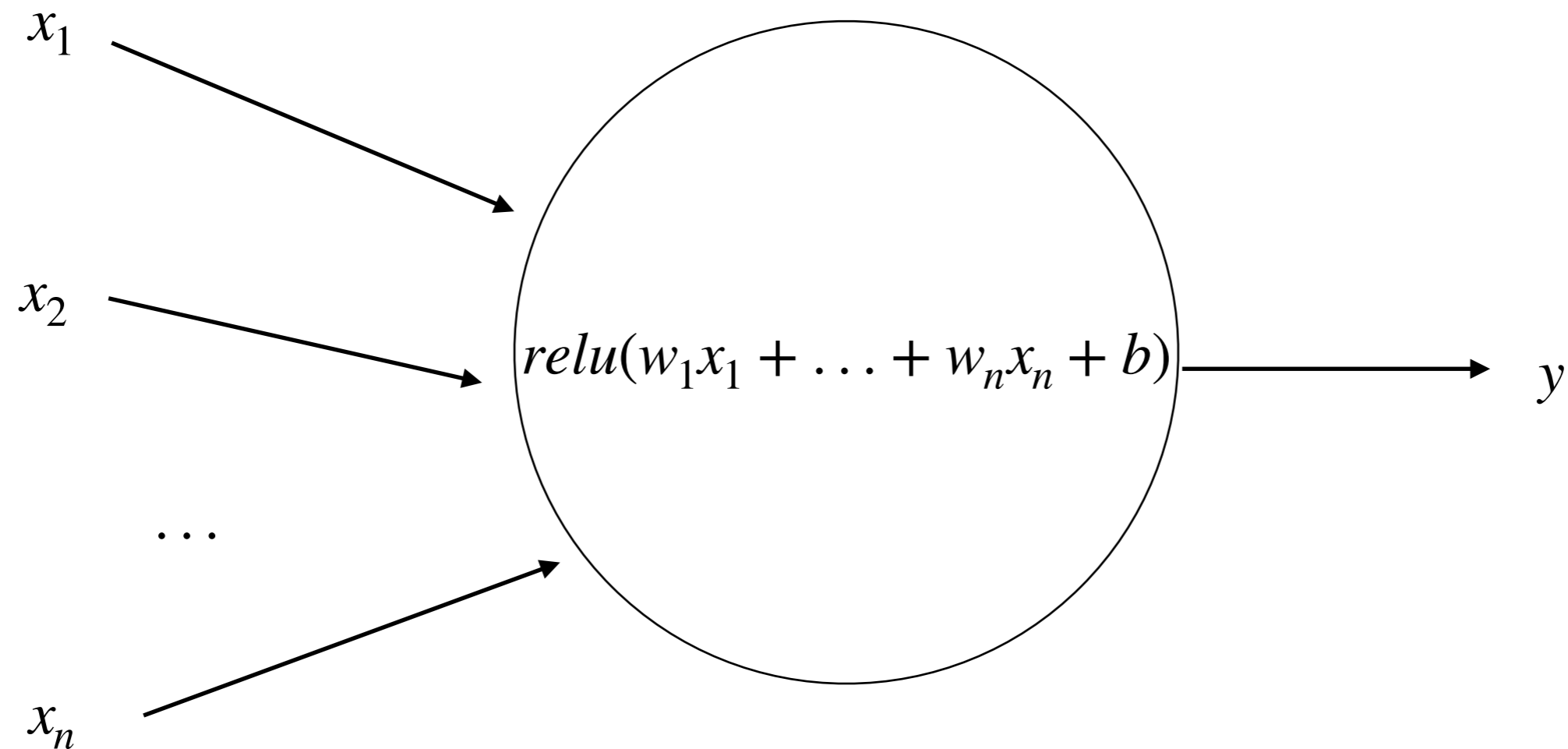


▶ *Arctan*: given the value in input z , it returns $\tan^{-1}(z)$.



Credit: Wikimedia

Nodes/Units/Neurons



Note that here the function in input of relu is 1-dimensional.

Softmax Function

- ▶ Another function that we will use is *softmax*.
- ▶ But please note that softmax is not like the activation functions that we discussed before. The activations functions that we discussed before take in input real numbers and returns a real number.
- ▶ A softmax function receives in input a vector of real numbers of dimension n and returns a vector of real numbers of dimension n .
- ▶ *Softmax*: given a vector of real numbers in input \mathbf{z} of dimension n , it normalises it into a probability distribution consisting of n probabilities proportional to the exponentials of each element z_i of the vector \mathbf{z} . More formally,

$$\mathit{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}} \text{ for } i = 1, \dots, n.$$

Gradient-based Optimization

- ▶ We will now discuss a high-level description of the learning process of the network, usually called *gradient-based optimization*.

- ▶ Each neural layer transforms his input layer as follows:

$$output = f(w_1x_1 + \dots + w_nx_n + b)$$

- ▶ And in the case of a relu function, we will have

$$output = \text{relu}(w_1x_1 + \dots + w_nx_n + b)$$

- ▶ Note that this is a simplified notation for one layer, it should be $w_{1,i}$ for layer i .

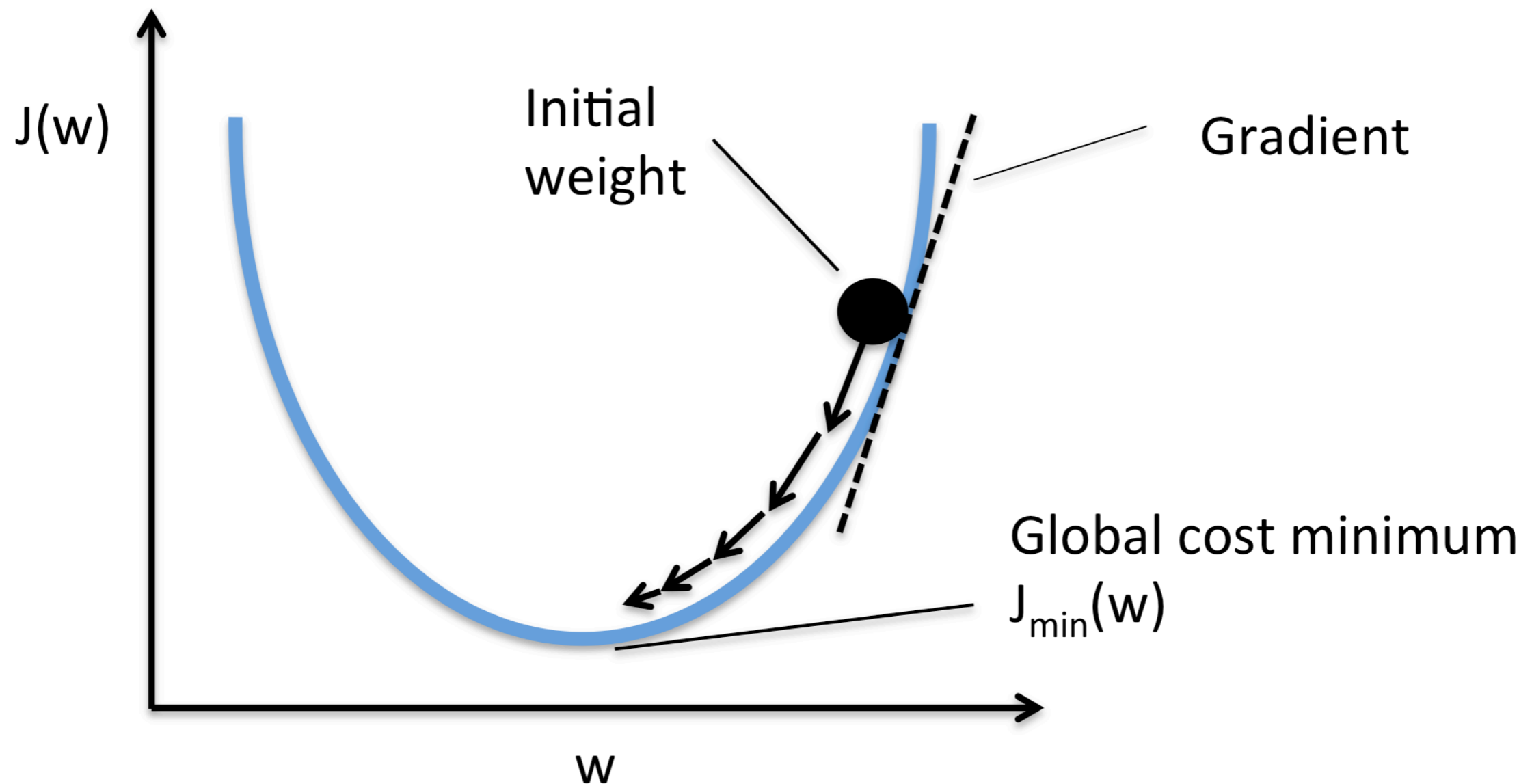
Gradient-based Optimisation

- ▶ The learning is based on the gradual adjustment of the weight based on a feedback signal, i.e., the loss described above.
- ▶ The training is based on the following training loop:
 - ▶ Draw a batch of training examples \mathbf{x} and corresponding targets \mathbf{y}_{target} .
 - ▶ Run the network on \mathbf{x} (forward pass) to obtain predictions \mathbf{y}_{pred} .
 - ▶ Compute the loss of the network on the batch, a measure of the mismatch between \mathbf{y}_{pred} and \mathbf{y}_{target} .
 - ▶ Update all weights of the networks in a way that reduces the loss of this batch.

Stochastic Gradient Descent

- ▶ Given a differentiable function, it's theoretically possible to find its minimum analytically.
- ▶ However, the function is intractable for real networks. The only way is to try to approximate the weights using the procedure described above.
- ▶ More precisely, since it is a *differentiable* function, we can use the gradient, which provides an efficient way to perform the correction mentioned before.

Gradient-based Optimisation



Credit: Sebastian Raschka

Stochastic Gradient Descent

► More formally:

- Draw a batch of training example \mathbf{x} and corresponding targets \mathbf{y}_{target} .
- Run the network on \mathbf{x} (forward pass) to obtain predictions \mathbf{y}_{pred} .
- Compute the loss of the network on the batch, a measure of the mismatch between \mathbf{y}_{pred} and \mathbf{y}_{target} .
- Compute the gradient of the loss with regard to the network's parameters (backward pass).

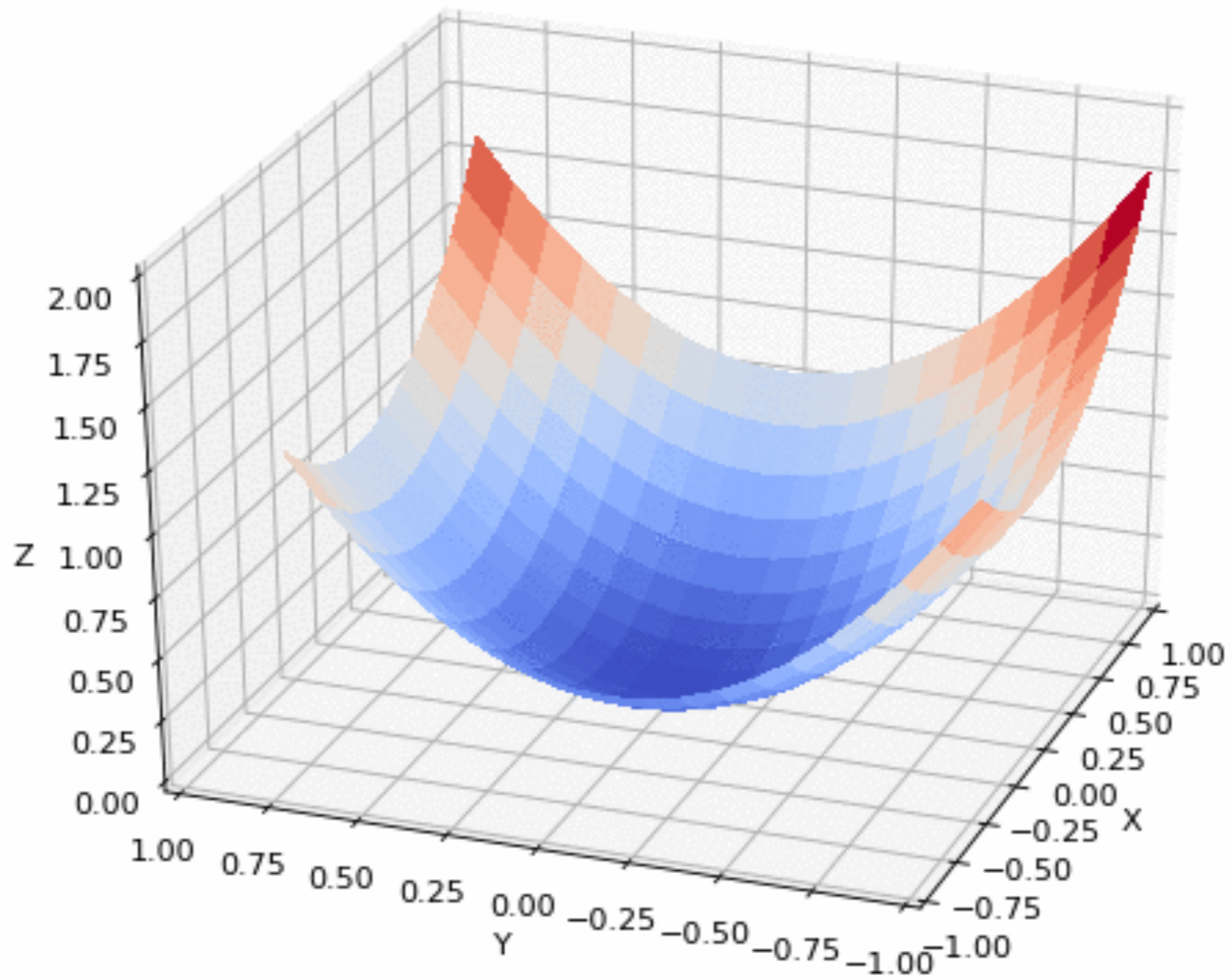
► Move the parameters in the opposite direction from the gradient with: $w_j \leftarrow w_j + \Delta w_j = w_j - \eta \frac{\partial J}{\partial w_j}$
where J is the loss (cost) function.

► If you have a batch of samples of dimension k :

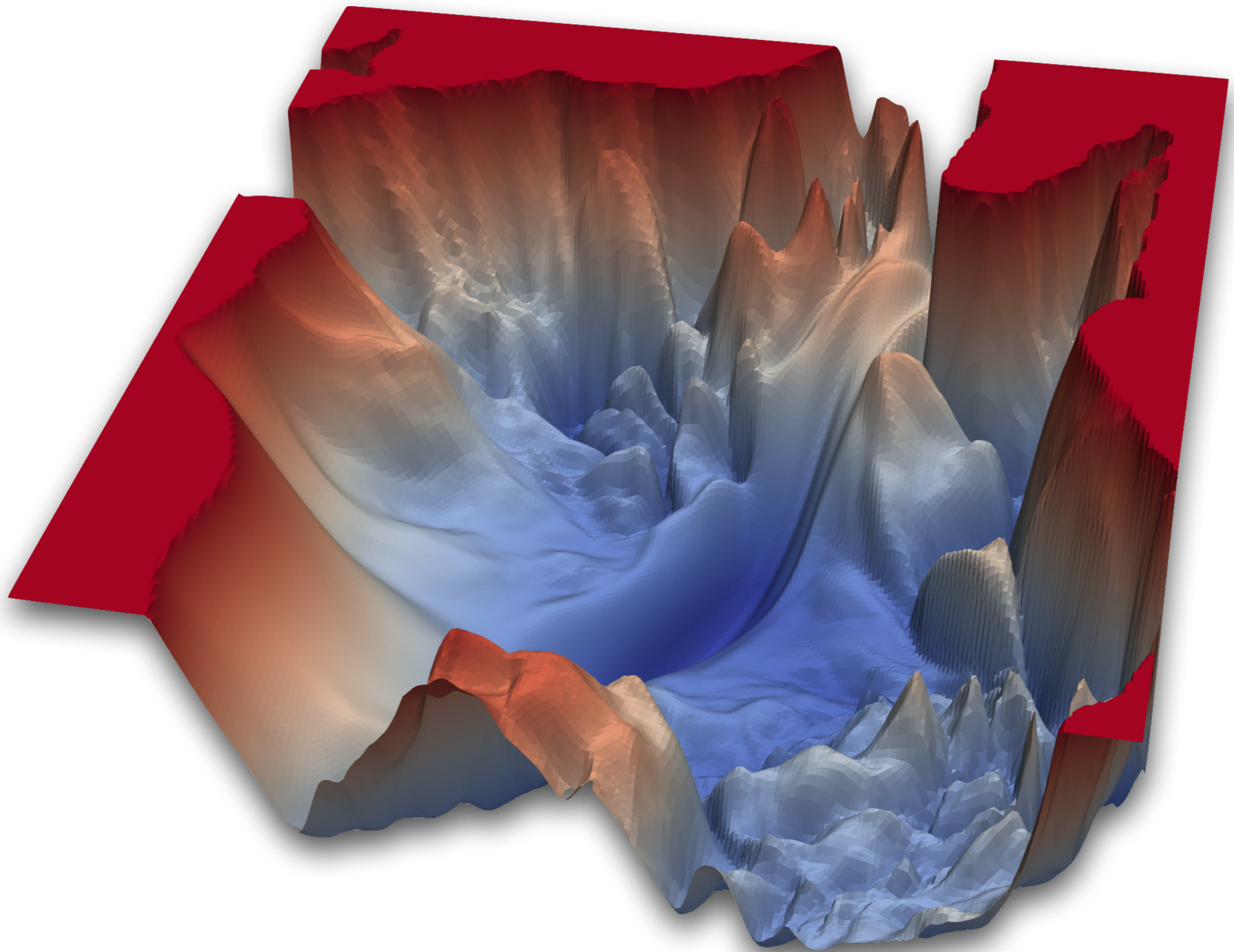
$$w_j \leftarrow w_j + \Delta w_j = w_j - \eta \text{average}\left(\frac{\partial J_k}{\partial w_j}\right) \text{ for all the } k \text{ samples of the batch.}$$

Stochastic Gradient Descent

- ▶ This is called the mini-batch stochastic gradient descent (mini-batch SGD).
- ▶ The loss function J is a function of $f(\mathbf{x})$, which is a function of the weights.
 - ▶ Essentially, you calculate the value $f(\mathbf{x})$, which is a function of the weights of the network.
 - ▶ Therefore, by definition, the derivative of the loss function that you are going to apply will be a function of the weights.
- ▶ The term *stochastic* refers to the fact that each batch of data is drawn randomly.
- ▶ The algorithm described above was based on a simplified model with a single function in a sense.
- ▶ You can think about a network composed of three layers, e.g., three tensor operations on the network itself.



<https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/>



<https://www.cs.umd.edu/~tomg/projects/landscapes/>

Backpropagation Algorithm

- ▶ Suppose that you have three tensor operations/layers f, g, h with weights $\mathbf{W}^1, \mathbf{W}^2$ and \mathbf{W}^3 respectively for the first, second, third layer. You will have the following function:

$$y_{pred} = f(\mathbf{W}^1, \mathbf{W}^2, \mathbf{W}^3, \mathbf{x}) = f(\mathbf{W}^3, g(\mathbf{W}^2, h(\mathbf{W}^1, \mathbf{x})))$$

with $f()$ the *rightmost* function/layer and so on. In other words, the input layer is connected to $h()$, which is connected to $g()$, which is connected to $f()$, which returns the final result.

- ▶ A network is a sort of chain of layers. You can derive the value of the “correction” by applying the chain rule of the derivatives backwards.
 - ▶ Remember the chain rule $(f(g(x)))' = f'(g(x))g'(x)$.

Backpropagation Algorithm

- ▶ The update of the weights starts from the right-most layer *back* to the left-most layer. For this reason, this is called *backpropagation* algorithm.
- ▶ More specifically, backpropagation starts with the calculation of the gradient of final loss value and works backwards from the right-most layers to the left-most layers, applying the chain rule to compute the contribution that each weight had in the loss value.
- ▶ Nowadays, we do not calculate the partial derivatives manually, but we use frameworks like TensorFlow and PyTorch that support symbolic differentiation for the calculation of the gradient.
- ▶ TensorFlow and PyTorch support the automatic updates of the weights described above.
- ▶ More theoretical details can be found in:

Ian Goodfellow, Yoshua Bengio and Aaron Courville. Deep Learning. MIT Press. 2016.

References

- ▶ Chapter 1 of Ian Goodfellow, Yoshua Bengio and Aaron Courville. Deep Learning. MIT Press. 2016.
- ▶ Chapter 2 of Francois Chollet. Deep Learning with Python. Manning 2022.