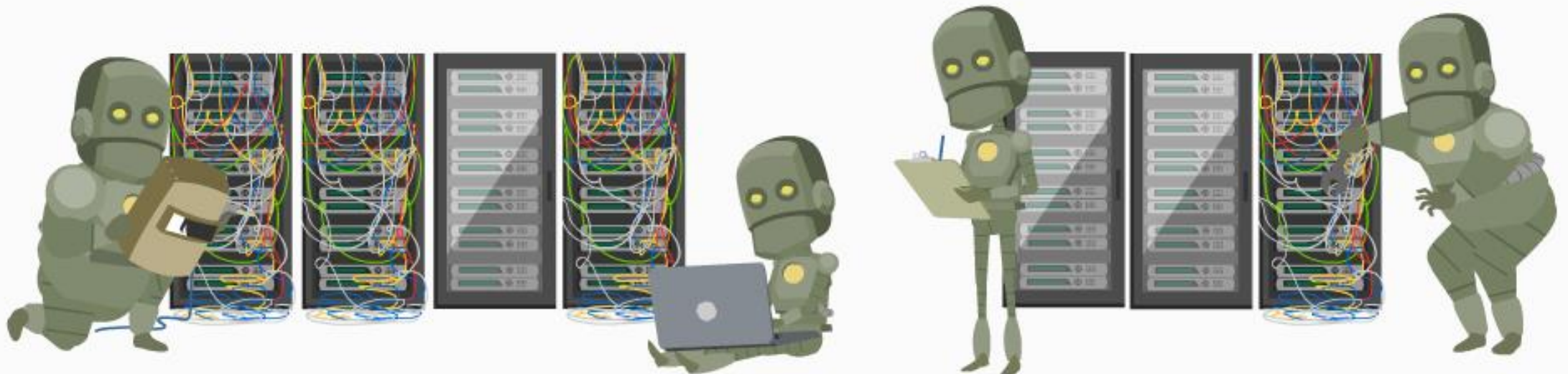


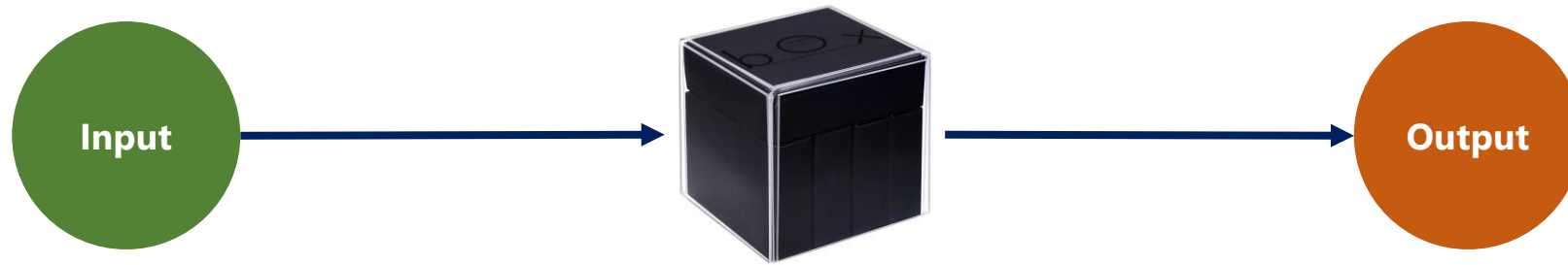
365 DataScience

MACHINE LEARNING

COURSE NOTES – SECTION 2



Introduction



Most generally, a machine learning algorithm can be thought of as a black box. It takes inputs and gives outputs.

The purpose of this course is to show you how to create this 'black box' and tailor it to your needs.

For example, we may create a model that predicts the weather tomorrow, based on meteorological data about the past few days.

The "black box" in fact is a mathematical model. The machine learning algorithm will follow a kind of trial-and-error method to determine the model that estimates the outputs, given inputs.

Once we have a model, we must **train** it. **Training** is the process through which, the model **learns** how to make sense of input data.

Types of machine learning

Supervised

It is called *supervised* as we provide the algorithm not only with the inputs, but also with the targets (desired outputs). This course focuses on supervised machine learning.

Based on that information the algorithm learns how to produce outputs as close to the **targets** as possible.

The objective function in supervised learning is called **loss function** (also cost or error). We are trying to minimize the loss as the lower the loss function, the higher the accuracy of the model.

Common methods:

- Regression
- Classification

Unsupervised

In *unsupervised* machine learning, the researcher feeds the model with inputs, but **not** with targets. Instead she asks it to find some sort of dependence or underlying logic in the data provided.

For example, you may have the financial data for 100 countries. The model manages to divide (cluster) them into 5 groups. You then examine the 5 clusters and reach the conclusion that the groups are: "Developed", "Developing but overachieving", "Developing but underachieving", "Stagnating", and "Worsening".

The algorithm divided them into 5 groups based on **similarities**, but you didn't know what similarities. It could have divided them by location instead.

Common methods:

- Clustering

Reinforcement

In reinforcement ML, the goal of the algorithm is to maximize its reward. It is inspired by human behavior and the way people change their actions according to incentives, such as getting a reward or avoiding punishment.

The objective function is called a **reward function**. We are trying to maximize the reward function.

An example is a computer playing Super Mario. The higher the score it achieves, the better it is performing. The score in this case is the objective function.

Common methods:

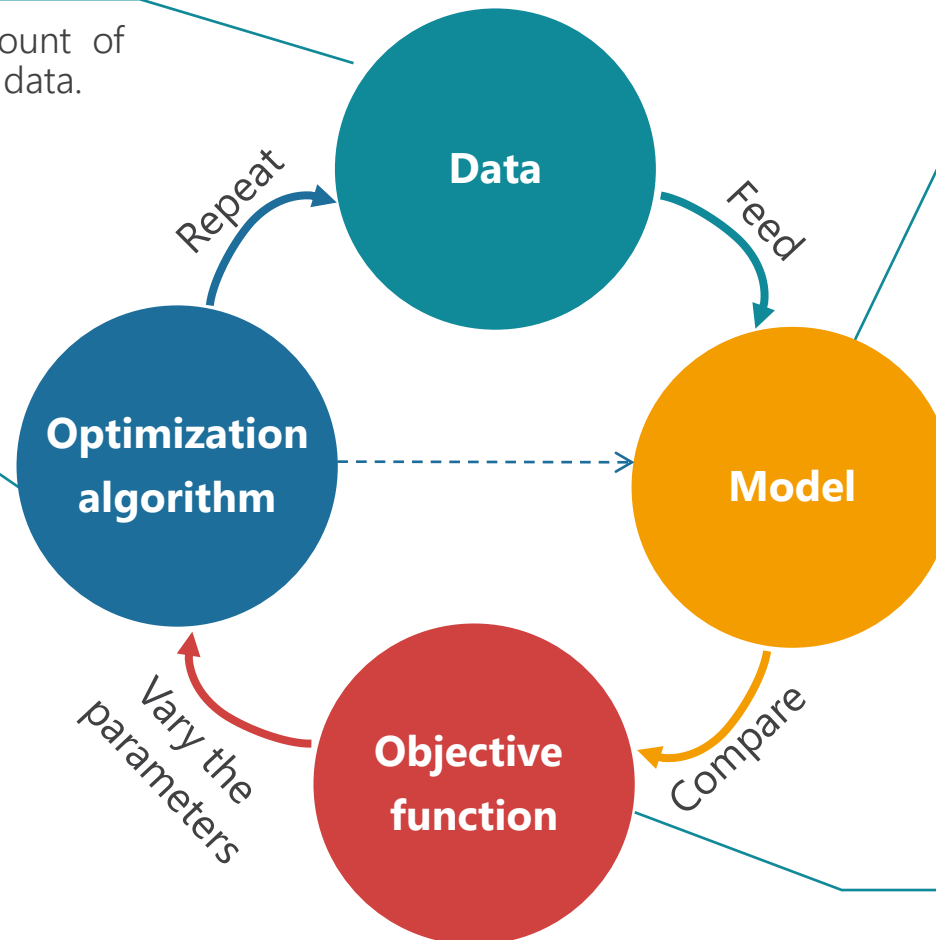
- Decision process
- Reward system

Building blocks of a machine learning algorithm

The basic logic behind training an algorithm involves four ingredients: data, model, objective function, and optimization algorithm. They are **ingredients**, instead of steps, as the process is iterative.

First, we need to prepare a certain amount of **data** to train on. Usually, we take historical data.

We achieve the optimization using an **optimization algorithm**. Using the value of the objective function, the optimization algorithm *varies the parameters* of the model. This operation is repeated until we find the values of the parameters, for which the objective function is optimal.



We choose the type of **model**. Roughly speaking, this is some function, which is defined by the *weights* and the *biases*. We feed the input data into the model. Essentially, the idea of the machine learning algorithm is to find the *parameters* for which the model has the highest predictive power.

The **objective function** measures the predictive power of our model. Mathematically, the machine learning problem boils down to *optimizing* this function. For example, in the case of loss, we are trying to *minimize* it.

Types of supervised learning

Supervised learning could be split into two subtypes – **regression** and **classification**.

Regression

Regression outputs are continuous numbers.

Examples:

Predicting the EUR/USD exchange rate tomorrow. The output is a number, such as 1.02, 1.53, etc.

Predicting the price of a house, e.g. \$234,200 or \$512,414.

One of the main properties of the regression outputs is that they are ordered. As $1.02 < 1.53$, and $243200 < 512414$, we can surely say that one output is bigger than the other.

This distinction proves to be crucial in machine learning as the algorithm somewhat *gets additional information* from the outputs.

Classification

Classification outputs are labels from some sort of class.

Examples:

Classifying photos of animals. The classes are "cats", "dogs", "dolphins", etc.

Predicting conversions in a business context. You can have two classes of customers, e.g. "will buy again" and "won't buy again".

In the case of classification, the labels are not ordered and cannot be compared at all. A dog photo is not "more" or "better" than a cat photo (objectively speaking) in the way a house worth \$512,414 is "more" (more expensive) than a house worth \$234,200.

This distinction proves to be crucial in machine learning, as the different classes are treated on **an equal footing**.

Model

The simplest possible model is **a linear model**. Despite appearing unrealistically simple, in the deep learning context, it is the basis of more complicated models.

The diagram shows the equation $y = xw + b$ in large, bold, black font. Four teal arrows point from labels to the variables in the equation: an arrow from 'output(s)' to 'y', an arrow from 'input(s)' to 'x', an arrow from 'weight(s)' to 'w', and an arrow from 'bias(es)' to 'b'.

There are four elements.

- The input(s), x . That's basically the data that we feed to the model.
- The weight(s), w . Together with the biases, they are called **parameters**. The optimization algorithm will vary the weights and the biases, in order to produce the model that fits the data best.
- The bias(es), b . See weight(s).
- The output(s), y . y is a function of x , determined by w and b .

Each model is determined solely by its parameters (the weights and the biases). That is why we are interested in varying them using the (kind of) trial-and-error method, until we find a model that explains the data sufficiently well.

Model – Continued

$$\mathbf{y} = \mathbf{x}\mathbf{w} + \mathbf{b}$$

Diagram illustrating the dimensions of the variables in the linear model equation $\mathbf{y} = \mathbf{x}\mathbf{w} + \mathbf{b}$:

- \mathbf{y} is $n \times m$
- \mathbf{x} is $n \times k$
- \mathbf{w} is $k \times m$
- \mathbf{b} is $1 \times m$

Where:

- n is the number of samples (observations)
- m is the number of output variables
- k is the number of input variables

A linear model can represent multidimensional relationships. The shapes of y , x , w , and b are given above (notation is arbitrary).

The simplest linear model, where $n = m = k = 1$.

$$\boxed{y} = \boxed{x} \boxed{w} + \boxed{b}$$

The simplest linear model, where $m = k = 1$, $n > 1$

$$\begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1 w + b \\ x_2 w + b \\ \dots \\ x_n w + b \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} \boxed{w} + \boxed{b}$$

Examples:

$$\boxed{27} = \boxed{4} \boxed{6} + \boxed{3}$$

Since the weights and biases alone define a model, this example shows **the same model** as above but for many data points.

$$\begin{bmatrix} 27 \\ 33 \\ \dots \\ -3 \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \\ \dots \\ -1 \end{bmatrix} \boxed{6} + \boxed{3}$$

Note that we add the bias to each row, essentially simulating an $n \times 1$ matrix. That's how computers treat such matrices, that's why in machine learning we define the bias in this way.

Model multiple inputs

$$\mathbf{y} = \mathbf{x}\mathbf{w} + \mathbf{b}$$

Diagram illustrating the dimensions of the variables in the equation $\mathbf{y} = \mathbf{x}\mathbf{w} + \mathbf{b}$:

- \mathbf{y} is $n \times m$ (number of samples n , number of output variables m).
- \mathbf{x} is $n \times k$ (number of samples n , number of input variables k).
- \mathbf{w} is $k \times m$ (number of input variables k , number of output variables m).
- \mathbf{b} is $1 \times m$ (number of output variables m).

Where:

- n is the number of samples (observations)
- m is the number of output variables
- k is the number of input variables

We can extend the model to multiple inputs where $n, k > 1, m = 1$.

$$\begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix} = \begin{bmatrix} x_{11}w_1 + x_{12}w_2 + \dots + x_{1k}w_k + b \\ x_{21}w_1 + x_{22}w_2 + \dots + x_{2k}w_k + b \\ \dots \\ x_{n1}w_1 + x_{n2}w_2 + \dots + x_{nk}w_k + b \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1k} \\ x_{21} & x_{22} & \dots & x_{2k} \\ \dots & \dots & \dots & \dots \\ x_{n1} & x_{n2} & \dots & x_{nk} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_k \end{bmatrix} + \begin{bmatrix} b \end{bmatrix}$$

Dimensions: $n \times 1$, $n \times k$, $k \times 1$, 1×1 .

Note that we add the bias to each row, essentially simulating an $n \times 1$ matrix. That's how computers treat such matrices, that's why in machine learning we define the bias in this way.

Example on the next page.

Model multiple inputs

$$y = xw + b$$

Diagram illustrating the dimensions of the variables in the equation $y = xw + b$:

- y is $n \times m$ (indicated by an arrow from $n \times m$ to y)
- x is $n \times k$ (indicated by an arrow from $n \times k$ to x)
- w is $k \times m$ (indicated by an arrow from $k \times m$ to w)
- b is $1 \times m$ (indicated by an arrow from $1 \times m$ to b)

Where:

- n is the number of samples (observations)
- m is the number of output variables
- k is the number of input variables

We can extend the model to multiple inputs where $n, k > 1$ and $m = 1$. An example.

y_1	=	$9 \times (-2) + 12 \times 5 + \dots + 13 \times (-1) + 4$	=	9	12	...	13	-2	+	4
y_2		$10 \times (-2) + 6 \times 5 + \dots + 2 \times (-1) + 4$		10	6	...	2	5		
...			
...		-1		
y_n		$7 \times (-2) + 7 \times 5 + \dots + 1 \times (-1) + 4$		7	7	...	1			

Dimensions:

- y is $n \times 1$
- x is $n \times k$
- w is $k \times 1$
- b is 1×1

Note that we add the bias to each row, essentially simulating an $n \times 1$ matrix. That's how computers treat such matrices, that's why in machine learning we define the bias in this way.

Model multiple inputs and multiple outputs

$$\mathbf{y} = \mathbf{x}\mathbf{w} + \mathbf{b}$$

Diagram illustrating the dimensions of the variables in the equation $\mathbf{y} = \mathbf{x}\mathbf{w} + \mathbf{b}$:

- \mathbf{y} (output vector) has dimensions $n \times m$.
- \mathbf{x} (input matrix) has dimensions $n \times k$.
- \mathbf{w} (weight matrix) has dimensions $k \times m$.
- \mathbf{b} (bias vector) has dimensions $1 \times m$.

Where:

- n is the number of samples (observations)
- m is the number of output variables
- k is the number of input variables

We can extend the model to multiple inputs where $n, k, m > 1$.

y_{11}	...	y_{1m}
y_{21}	...	y_{2m}
...
...
y_{n1}	...	y_{nm}

$n \times m$

 $=$

x_{11}	x_{12}	...	x_{1k}
x_{21}	x_{22}	...	x_{2k}
...
...
x_{n1}	x_{n2}	...	x_{nk}

$n \times k$

w_{11}	...	w_{1m}
w_{21}	...	w_{2m}
...
w_{k1}	...	w_{km}

$k \times m$

 $+$

b_1	...	b_m
-------	-----	-------

$1 \times m$

Note that we add the bias to each row, essentially simulating an $n \times m$ matrix. That's how computers treat such matrices, that's why in machine learning we define the bias in this way.

Example on the next page.

Model multiple inputs and multiple outputs

$$y = xw + b$$

Diagram illustrating the dimensions of the variables in the equation $y = xw + b$:

- y is $n \times m$ (number of samples n by number of output variables m).
- x is $n \times k$ (number of samples n by number of input variables k).
- w is $k \times m$ (number of input variables k by number of output variables m).
- b is $1 \times m$ (number of output variables m).

Where:

- n is the number of samples (observations)
- m is the number of output variables
- k is the number of input variables

We can extend the model to multiple inputs where $n, k, m > 1$.

$$\begin{array}{|c|c|c|} \hline y_{11} & \dots & y_{1m} \\ \hline y_{21} & \dots & y_{2m} \\ \hline \dots & \dots & \dots \\ \hline \dots & \dots & \dots \\ \hline y_{n1} & \dots & y_{nm} \\ \hline \end{array}
 =
 \begin{array}{|c|c|c|} \hline x_{11}w_{11} + x_{12}w_{21} + \dots + x_{1k}w_{k1} + b_1 & \dots & x_{11}w_{1m} + x_{12}w_{2m} + \dots + x_{1k}w_{km} + b_m \\ \hline x_{21}w_{11} + x_{22}w_{21} + \dots + x_{2k}w_{k1} + b_1 & \dots & x_{21}w_{1m} + x_{22}w_{2m} + \dots + x_{2k}w_{km} + b_m \\ \hline \dots & \dots & \dots \\ \hline \dots & \dots & \dots \\ \hline x_{n1}w_{11} + x_{n2}w_{21} + \dots + x_{nk}w_{k1} + b_1 & \dots & x_{n1}w_{1m} + x_{n2}w_{2m} + \dots + x_{nk}w_{km} + b_m \\ \hline \end{array}
 =
 \begin{array}{|c|c|c|c|} \hline x_{11} & x_{12} & \dots & x_{1k} \\ \hline x_{21} & x_{22} & \dots & x_{2k} \\ \hline \dots & \dots & \dots & \dots \\ \hline \dots & \dots & \dots & \dots \\ \hline x_{n1} & x_{n2} & \dots & x_{nk} \\ \hline \end{array}
 \begin{array}{|c|c|c|} \hline w_{11} & \dots & w_{1m} \\ \hline w_{21} & \dots & w_{2m} \\ \hline \dots & \dots & \dots \\ \hline w_{k1} & \dots & w_{km} \\ \hline \end{array}
 +
 \begin{array}{|c|c|c|} \hline b_1 & \dots & b_m \\ \hline \end{array}$$

Dimensions: $n \times m$ (left), $n \times k$ (middle), $k \times m$ (right), $1 \times m$ (bottom right).

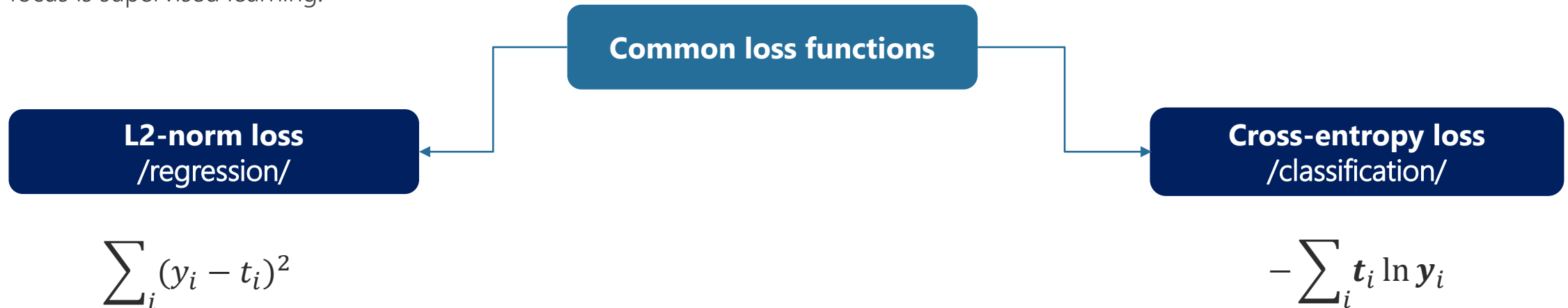
Note that we add the bias to each row, essentially simulating an $n \times m$ matrix. That's how computers treat such matrices, that's why in machine learning we define the bias in this way.

Objective function

The objective function is a measure of how well our model's outputs match the targets.

The **targets** are the "correct values" which we aim at. In the cats and dogs example, the targets were the "labels" we assigned to each photo (either "cat" or "dog").

Objective functions can be split into two types: **loss** (supervised learning) and **reward** (reinforcement learning). Our focus is supervised learning.



The L2-norm of a vector, **a**, (Euclidean length) is given by

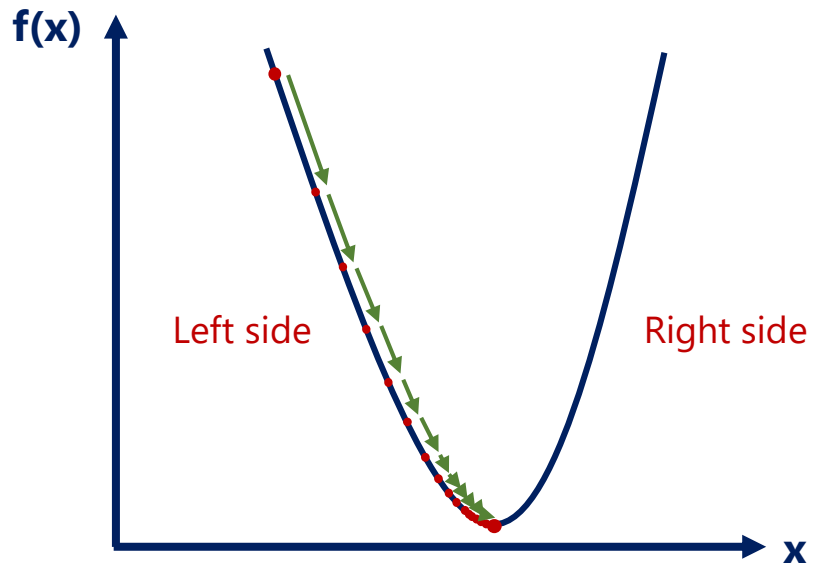
$$\|a\| = \sqrt{a^T \cdot a} = \sqrt{a_1^2 + \dots + a_n^2}$$

The main rationale is that the L2-norm loss is basically the distance from the origin (0). So, the closer to the origin is the difference of the outputs and the targets, the lower the loss, and the better the prediction.

The cross-entropy loss is mainly used for classification. Entropy comes from information theory, and measures how much information one is missing to answer a question. Cross-entropy (used in ML) works with probabilities – one is our opinion, the other – the true probability (the probability of a target to be correct is 1 by definition). If the cross-entropy is 0, then we are **not missing any information** and have a perfect model.

Gradient descent

The last ingredient is the optimization algorithm. The most commonly used one is the gradient descent. The main point is that we can find the minimum of a function by applying the rule: $x_{i+1} = x_i - \eta f'(x_i)$, where η is a small enough positive number. In machine learning, η , is called the learning rate. The rationale is that the first derivative at x_i , $f'(x_i)$ shows the slope of the function at x_i .



If the first derivative of $f(x)$ at x_i , $f'(x_i)$, is negative, then we are on the left side of the parabola (as shown in the figure). Subtracting a negative number from x_i (as η is positive), will result in x_{i+1} , that is bigger than x_i . This would cause our next *trial* to be on the right; thus, closer to the sought minimum.

Alternatively, if the first derivative is positive, then we are on the right side of the parabola. Subtracting a positive number from x_i will result in a lower number, so our next trial will be to the left (again closer to the minimum).

So, either way, using this rule, we are approaching the minimum. When the first derivative is 0, we have reached the minimum. Of course, our update rule won't update anymore ($x_{i+1} = x_i - 0$).

The learning rate η , must be low enough so we don't oscillate (bounce around without reaching the minimum) and big enough, so we reach it in rational time.

In machine learning, $f(x)$ is the **loss function**, which we are trying to minimize.

The variables that we are varying until we find the minimum are the **weights and the biases**. The proper update rules are:

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \eta \nabla_{\mathbf{w}} L(\mathbf{w}_i) = \mathbf{w}_i - \eta \sum_i \mathbf{x}_i \delta_i \quad \text{and} \quad b_{i+1} = b_i - \eta \nabla_b L(b_i) = b_i - \eta \sum_i \delta_i$$

Gradient descent. Multivariate derivation

The multivariate generalization of the gradient descent concept: $x_{i+1} = x_i - \eta f'(x_i)$ is given by: $\mathbf{w}_{i+1} = \mathbf{w}_i - \eta \nabla_{\mathbf{w}} L(\mathbf{w}_i)$ (using this example, as we will need it). In this new equation, \mathbf{w} is a matrix and we are interested in the gradient of \mathbf{w} , w.r.t. the loss function. As promised in the lecture, we will show the derivation of the gradient descent formulas for the L2-norm loss divided by 2.

Model: $y = xw + b$

Loss: $L = \frac{1}{2} \sum_i (y_i - t_i)^2$

Update rule: $\mathbf{w}_{i+1} = \mathbf{w}_i - \eta \nabla_{\mathbf{w}} L(\mathbf{w}_i)$

(opt. algorithm) $b_{i+1} = b_i - \eta \nabla_b L(b_i)$

Analogically, we find the update rule for the biases.

Please note that the division by 2 that we performed does not change the nature of the loss.

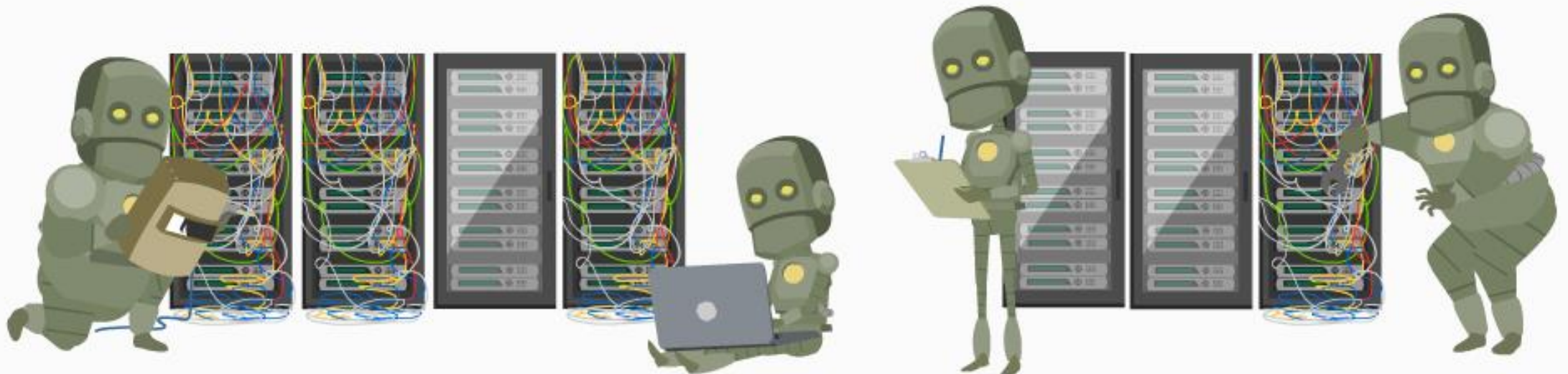
ANY function that holds the basic property of being higher for worse results and lower for better results can be a loss function.

$$\begin{aligned} \nabla_{\mathbf{w}} L &= \nabla_{\mathbf{w}} \frac{1}{2} \sum_i (y_i - t_i)^2 = \\ &= \nabla_{\mathbf{w}} \frac{1}{2} \sum_i ((\mathbf{x}_i \mathbf{w} + b) - t_i)^2 = \\ &= \sum_i \nabla_{\mathbf{w}} \frac{1}{2} (\mathbf{x}_i \mathbf{w} + b - t_i)^2 = \\ &= \sum_i \mathbf{x}_i (\mathbf{x}_i \mathbf{w} + b - t_i) = \\ &= \sum_i \mathbf{x}_i (y_i - t_i) \equiv \\ &\equiv \sum_i \mathbf{x}_i \delta_i \end{aligned}$$

365 DataScience

MACHINE LEARNING

COURSE NOTES – SECTION 6

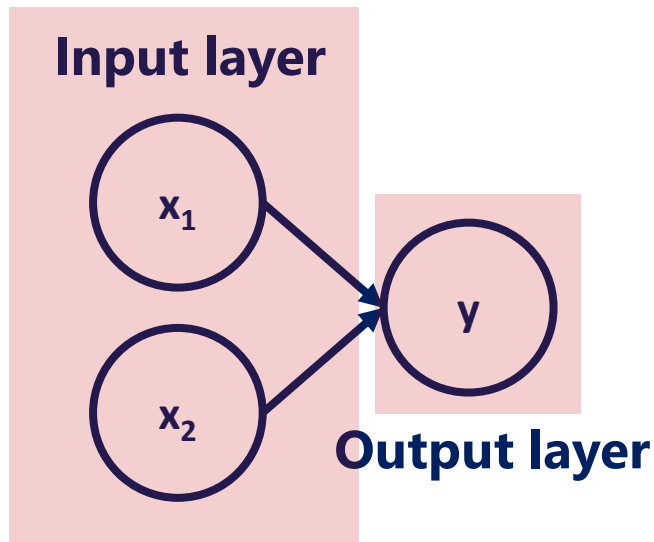


IT'S TIME TO DIG DEEPER

Layers

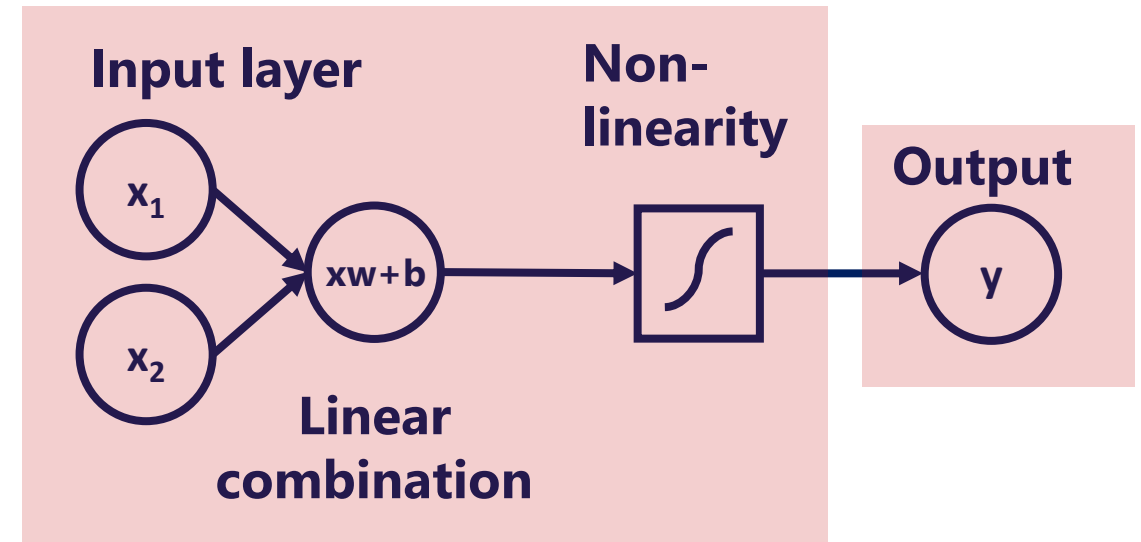
An initial linear combination and the added non-linearity form a **layer**. The layer is the building block of neural networks.

Minimal example (a simple neural network)



In the minimal example we trained a *neural network* which had no depth. There were solely an input layer and an output layer. Moreover, the output was simply a **linear combination** of the input.

Neural networks



Neural networks step on linear combinations, but add a non-linearity to each one of them. Mixing linear combinations and non-linearities allows us to model arbitrary functions.

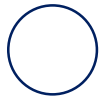
A deep net

This is a deep neural network (deep net) with 5 layers.

How to read this diagram:



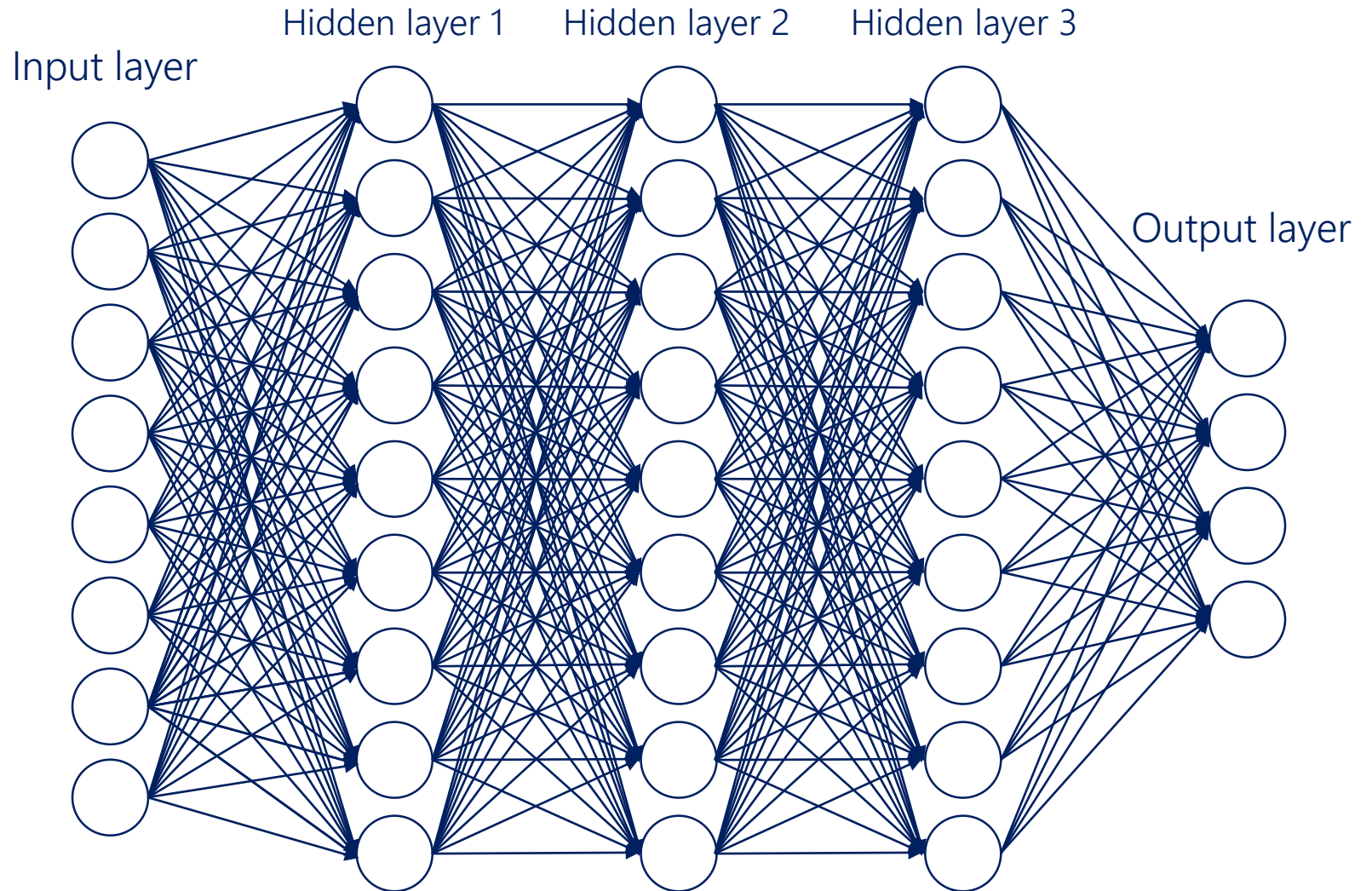
A layer



A unit (a neuron)



Arrows represent
mathematical transformations



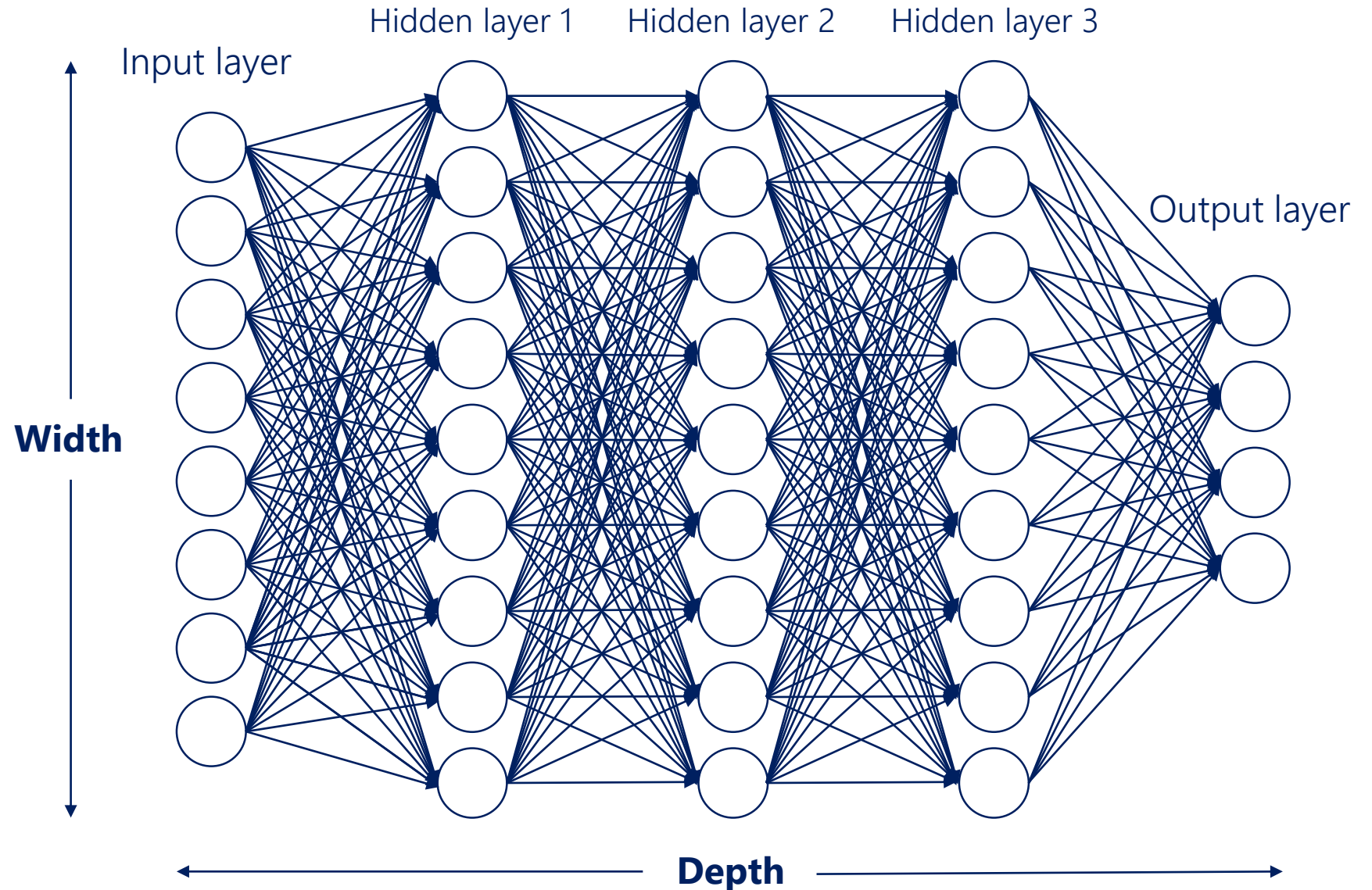
A deep net

The **width** of a layer is the number of units in that layer

The **width** of the net is the number of units of the biggest layer

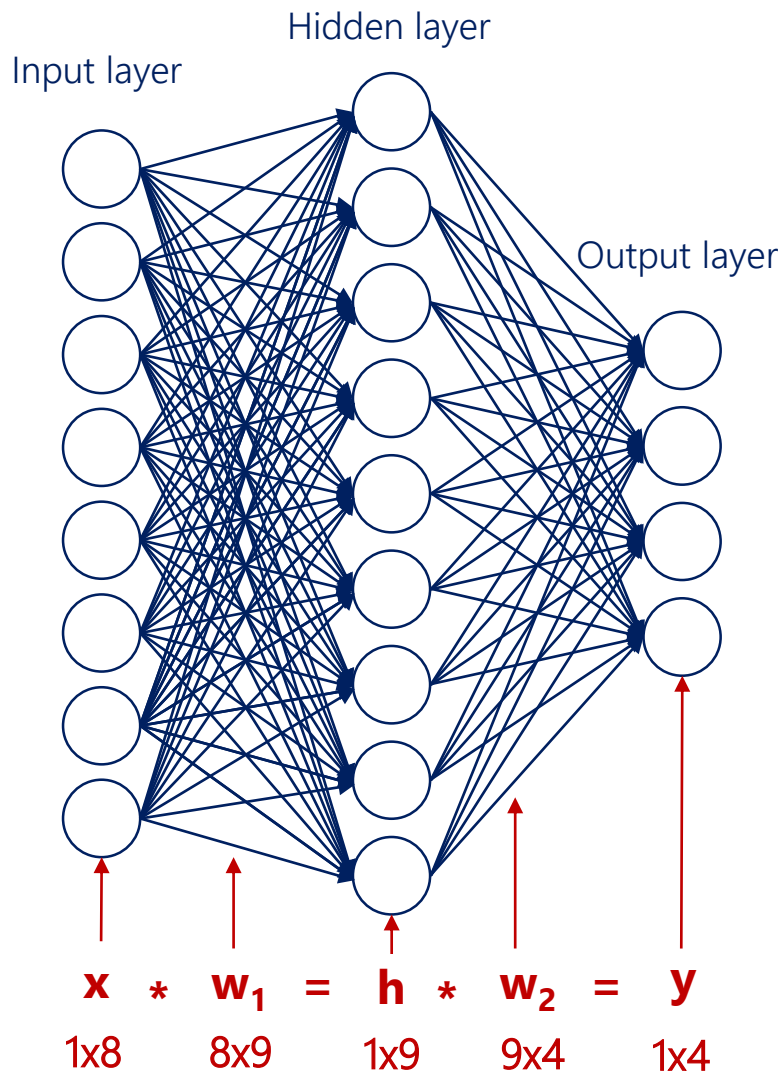
The **depth** of the net is equal to the number of layers or the number of hidden layers. The term has different definitions. More often than not, we are interested in the number of hidden layers (as there are always input and output layers).

The width and the depth of the net are called **hyperparameters**. They are values we manually chose when creating the net.



Why we need non-linearities to stack layers

You can see a net with no non-linearities: just linear combinations.



$$\mathbf{h} = \mathbf{x} * \mathbf{w}_1$$

$$\mathbf{y} = \mathbf{h} * \mathbf{w}_2$$

$$\mathbf{y} = \mathbf{x} * \boxed{\mathbf{w}_1 * \mathbf{w}_2}$$

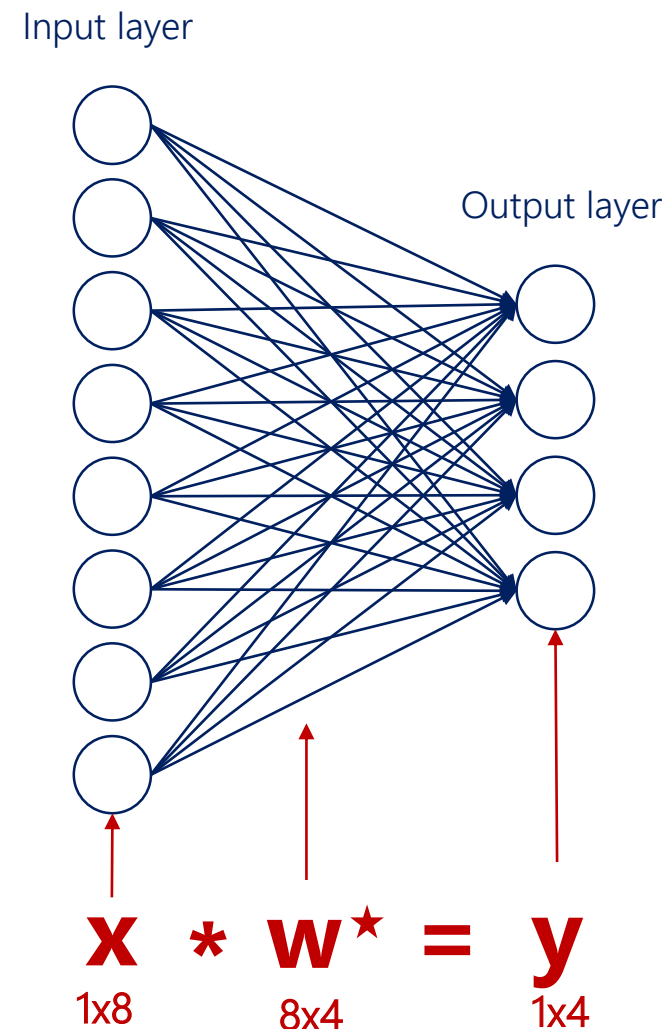
8×9 9×4

$$\mathbf{y} = \mathbf{x} * \mathbf{w}^*$$

8×4

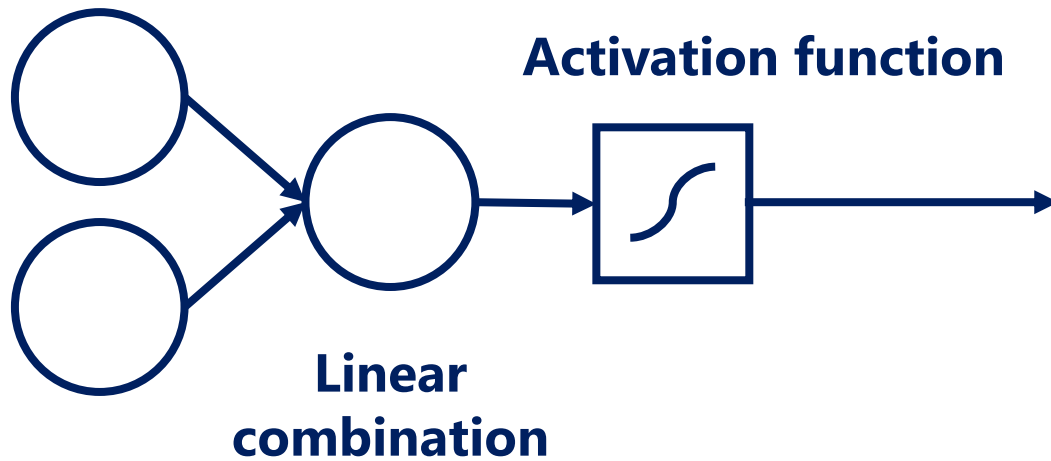
Two consecutive linear transformations are equivalent to a single one.

Two consecutive linear transformations are equivalent to a single one.



Activation functions

Input



Activation functions (non-linearities) are needed so we can break the linearity and represent more complicated relationships.

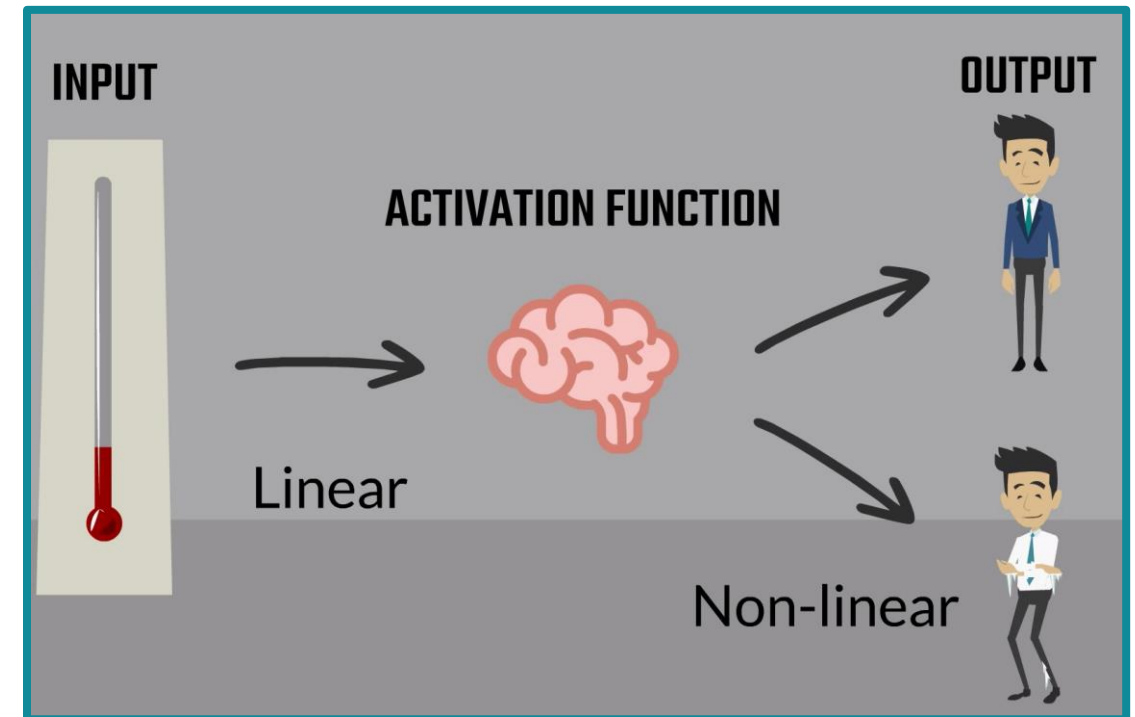
Moreover, activation functions are required in order to **stack layers**.

Activation functions transform inputs into outputs of a different kind.

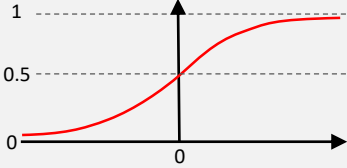
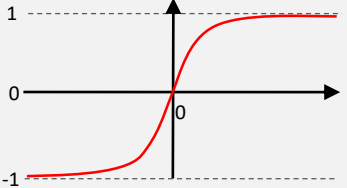
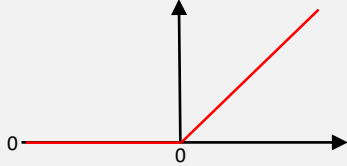
In the respective lesson, we gave an example of temperature change. The temperature starts decreasing (which is a numerical change). Our brain is a kind of an 'activation function'. It tells us whether it is **cold enough** for us to put on a jacket.

Putting on a jacket is a binary action: 0 (no jacket) or 1 (jacket).

This is a very intuitive and visual (yet not so practical) example of how activation functions work.



Common activation functions

Name	Formula	Derivative	Graph	Range
sigmoid (logistic function)	$\sigma(a) = \frac{1}{1+e^{-a}}$	$\frac{\partial \sigma(a)}{\partial a} = \sigma(a)(1 - \sigma(a))$		(0,1)
TanH (hyperbolic tangent)	$\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$	$\frac{\partial \tanh(a)}{\partial a} = \frac{4}{(e^a + e^{-a})^2}$		(-1,1)
ReLu (rectified linear unit)	$\text{relu}(a) = \max(0, a)$	$\frac{\partial \text{relu}(a)}{\partial a} = \begin{cases} 0, & \text{if } a \leq 0 \\ 1, & \text{if } a > 0 \end{cases}$		(0,∞)
softmax	$\sigma_i(a) = \frac{e^{a_i}}{\sum_j e^{a_j}}$	$\frac{\partial \sigma_i(a)}{\partial a_j} = \sigma_i(a) (\delta_{ij} - \sigma_j(a))$ Where δ_{ij} is 1 if $i=j$, 0 otherwise		(0,1)

All common activation functions are: **monotonic**, **continuous**, and **differentiable**. These are important properties needed for the optimization.

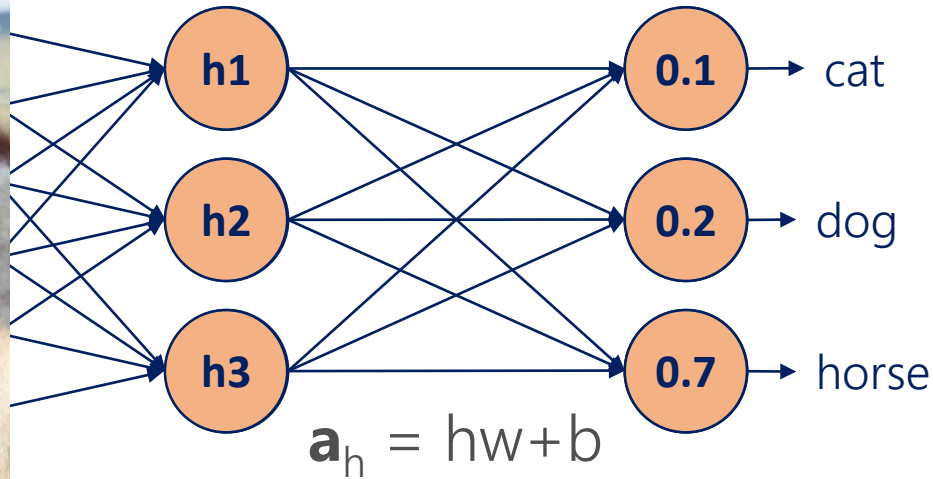
Softmax activation

Input layer



Hidden layer

Output layer



The softmax activation transforms a bunch of arbitrarily large or small numbers into a valid probability distribution.

While other activation functions get an input value and transform it, regardless of the other elements, the softmax considers the information about the **whole set of numbers** we have.

The values that softmax outputs are in the range from 0 to 1 and their sum is exactly 1 (like probabilities).

Example:

$$\mathbf{a} = [-0.21, 0.47, 1.72]$$

$$\text{softmax}(\mathbf{a}) = \frac{e^{a_i}}{\sum_j e^{a_j}}$$

$$\sum_j e^{a_j} = e^{-0.21} + e^{0.47} + e^{1.72} = 8$$

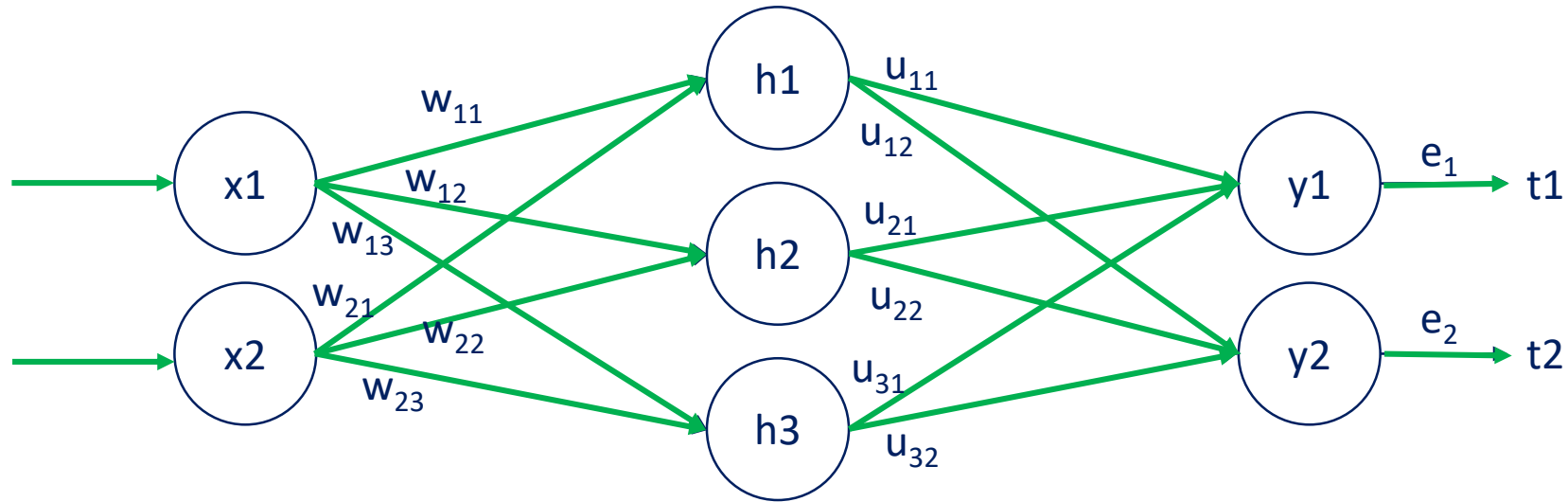
$$\text{softmax}(\mathbf{a}) = \left[\frac{e^{-0.21}}{8}, \frac{e^{0.47}}{8}, \frac{e^{1.72}}{8} \right]$$

$$\mathbf{y} = [0.1, 0.2, 0.7] \rightarrow \text{probability distribution}$$

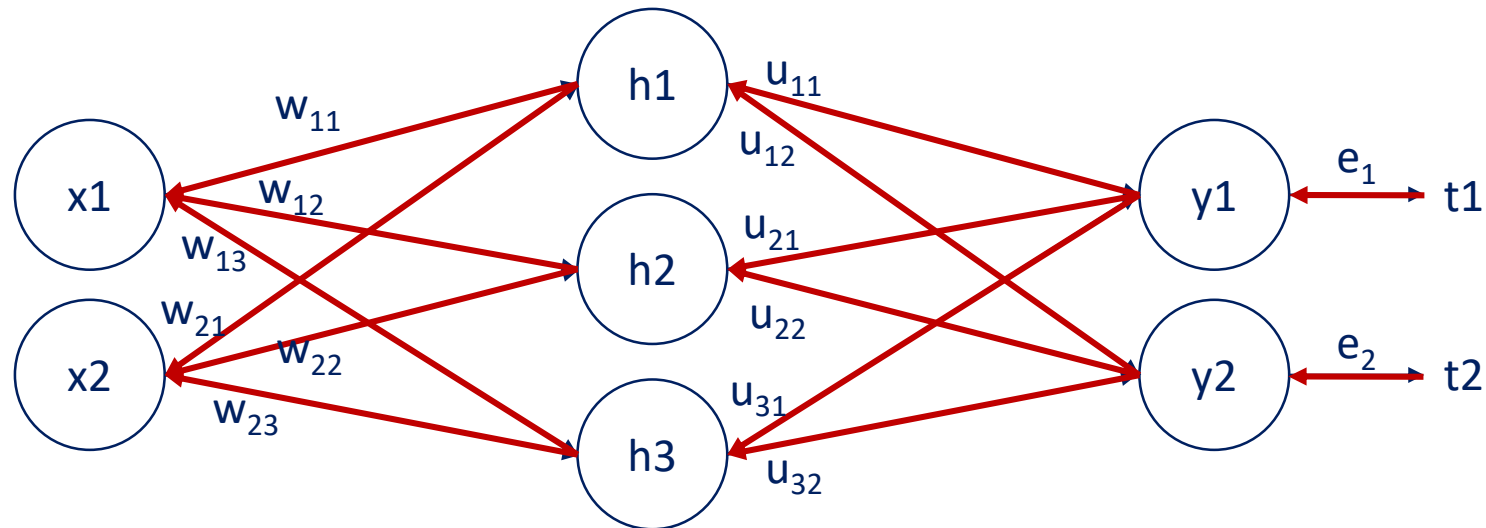
The property of the softmax to output probabilities is so useful and intuitive that it is often used as the activation function for the **final (output) layer**.

However, when the softmax is used prior to that (as the activation of a hidden layer), the results are not as satisfactory. That's because a lot of the information about the variability of the data is lost.

Backpropagation

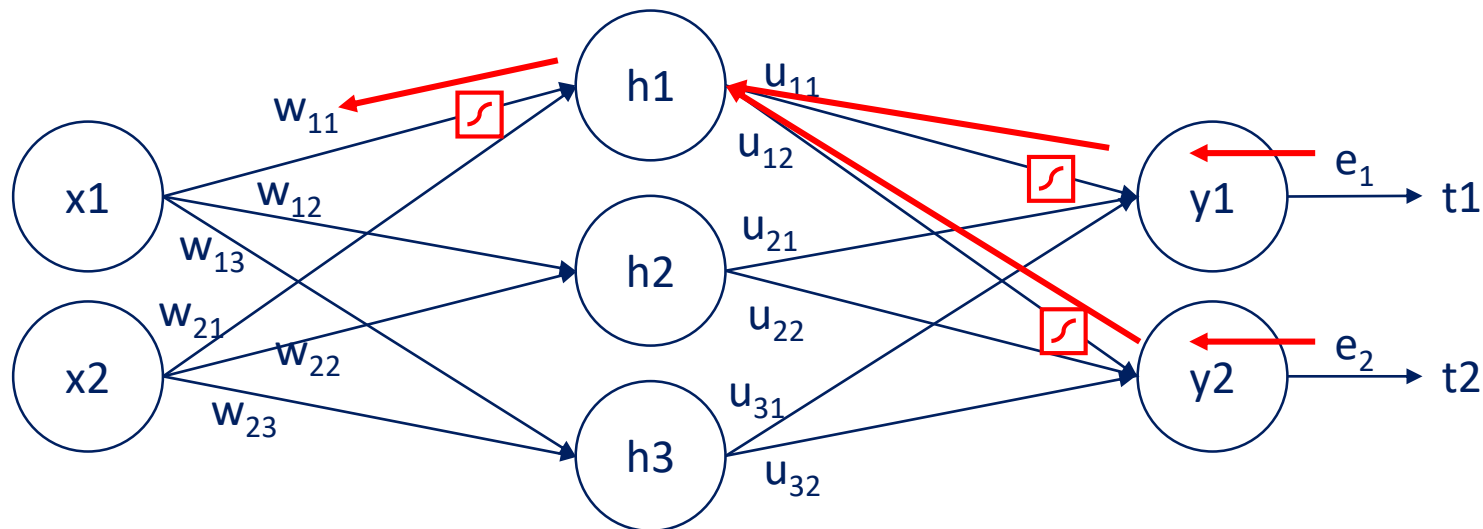


Forward propagation is the process of pushing inputs through the net. At the end of each epoch, the obtained outputs are compared to targets to form the errors.



Backpropagation of errors is an **algorithm** for neural networks using gradient descent. It consists of calculating the contribution of each **parameter** to the errors. We backpropagate the **errors** through the net and **update** the parameters (weights and biases) accordingly.

Backpropagation formula



$$\frac{\partial L}{\partial w_{ij}} = \delta_j x_i, \text{ where } \delta_j = \sum_k \delta_k w_{jk} y_j (1 - y_j)$$

If you want to examine the full derivation, please make use of the PDF we made available in the section: **Backpropagation. A peek into the Mathematics of Optimization.**

Backpropagation. A Peek into the Mathematics of Optimization



1 Motivation

In order to get a truly deep understanding of deep neural networks, one must look at the mathematics of it. As backpropagation is at the core of the optimization process, we wanted to introduce you to it. This is definitely not a necessary part of the course, as in TensorFlow, sk-learn, or any other machine learning package (as opposed to simply NumPy), will have backpropagation methods incorporated.

2 The specific net and notation we will examine

Here's our simple network:

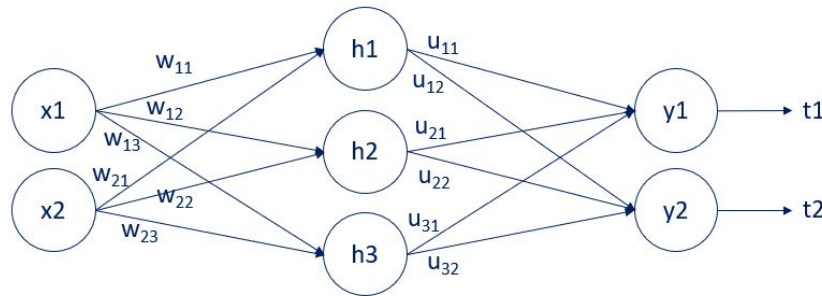


Figure 1: Backpropagation

We have two inputs: x_1 and x_2 . There is a single hidden layer with 3 units (nodes): h_1 , h_2 , and h_3 . Finally, there are two outputs: y_1 and y_2 . The arrows that connect them are the weights. There are two weights matrices: \mathbf{w} , and \mathbf{u} . The \mathbf{w} weights connect the input layer and the hidden layer. The \mathbf{u} weights connect the hidden layer and the output layer. We have employed the letters \mathbf{w} , and \mathbf{u} , so it is easier to follow the computation to follow.

You can also see that we compare the outputs y_1 and y_2 with the targets t_1 and t_2 .

There is one last letter we need to introduce before we can get to the computations. Let a be the linear combination prior to activation. Thus, we have: $\mathbf{a}^{(1)} = \mathbf{x}\mathbf{w} + \mathbf{b}^{(1)}$ and $\mathbf{a}^{(2)} = \mathbf{h}\mathbf{u} + \mathbf{b}^{(2)}$.

Since we cannot exhaust all activation functions and all loss functions, we will focus on two of the most common. A **sigmoid** activation and an **L2-norm loss**.

With this new information and the new notation, the output y is equal to the activated linear combination. Therefore, for the output layer, we have $\mathbf{y} = \sigma(\mathbf{a}^{(2)})$, while for the hidden layer: $\mathbf{h} = \sigma(\mathbf{a}^{(1)})$.

We will examine backpropagation for the output layer and the hidden layer separately, as the methodologies differ.

3 Useful formulas

I would like to remind you that:

$$\text{L2-norm loss: } L = \frac{1}{2} \sum_i (y_i - t_i)^2$$

The sigmoid function is:

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

and its derivative is:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

4 Backpropagation for the output layer

In order to obtain the update rule:

$$\mathbf{u} \leftarrow \mathbf{u} - \eta \nabla_{\mathbf{u}} L(\mathbf{u})$$

we must calculate

$$\nabla_{\mathbf{u}} L(\mathbf{u})$$

Let's take a single weight u_{ij} . The partial derivative of the loss w.r.t. u_{ij} equals:

$$\frac{\partial L}{\partial u_{ij}} = \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial a_j^{(2)}} \frac{\partial a_j^{(2)}}{\partial u_{ij}}$$

where i corresponds to the previous layer (input layer for this transformation) and j corresponds to the next layer (output layer of the transformation). The partial derivatives were computed simply following the chain rule.

$$\frac{\partial L}{\partial y_j} = (y_j - t_j)$$

following the L2-norm loss derivative.

$$\frac{\partial y_j}{\partial a_j^{(2)}} = \sigma(a_j^{(2)})(1 - \sigma(a_j^{(2)})) = y_j(1 - y_j)$$

following the sigmoid derivative.

Finally, the third partial derivative is simply the derivative of $\mathbf{a}^{(2)} = \mathbf{h}\mathbf{u} + \mathbf{b}^{(2)}$. So,

$$\frac{\partial a_j^{(2)}}{\partial u_{ij}} = h_i$$

Replacing the partial derivatives in the expression above, we get:

$$\frac{\partial L}{\partial u_{ij}} = \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial a_j^{(2)}} \frac{\partial a_j^{(2)}}{\partial u_{ij}} = (y_j - t_j) y_j (1 - y_j) h_i = \delta_j h_i$$

Therefore, the update rule for a single weight for the output layer is given by:

$$u_{ij} \leftarrow u_{ij} - \eta \delta_j h_i$$

5 Backpropagation of a hidden layer

Similarly to the backpropagation of the output layer, the update rule for a single weight, w_{ij} would depend on:

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial h_j} \frac{\partial h_j}{\partial a_j^{(1)}} \frac{\partial a_j^{(1)}}{\partial w_{ij}}$$

following the chain rule.

Taking advantage of the results we have so far for transformation using the sigmoid activation and the linear model, we get:

$$\frac{\partial h_j}{\partial a_j^{(1)}} = \sigma(a_j^{(1)})(1 - \sigma(a_j^{(1)})) = h_j(1 - h_j)$$

and

$$\frac{\partial a_j^{(1)}}{\partial w_{ij}} = x_i$$

The actual problem for backpropagation comes from the term $\frac{\partial L}{\partial h_j}$. That's due to the fact that there is no "hidden" target. You can follow the solution for weight w_{11} below. It is advisable to also check Figure 1, while going through the computations.

$$\begin{aligned}\frac{\partial L}{\partial h_1} &= \frac{\partial L}{\partial y_1} \frac{\partial y_1}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial h_1} + \frac{\partial L}{\partial y_2} \frac{\partial y_2}{\partial a_2^{(2)}} \frac{\partial a_2^{(2)}}{\partial h_1} = \\ &= (y_1 - t_1)y_1(1 - y_1)u_{11} + (y_2 - t_2)y_2(1 - y_2)u_{12}\end{aligned}$$

From here, we can calculate $\frac{\partial L}{\partial w_{11}}$, which was what we wanted. The final expression is:

$$\frac{\partial L}{\partial w_{11}} = [(y_1 - t_1)y_1(1 - y_1)u_{11} + (y_2 - t_2)y_2(1 - y_2)u_{12}] h_1(1 - h_1)x_1$$

The generalized form of this equation is:

$$\frac{\partial L}{\partial w_{ij}} = \sum_k (y_k - t_k)y_k(1 - y_k)u_{jk}h_j(1 - h_j)x_i$$

6 Backpropagation generalization

Using the results for backpropagation for the output layer and the hidden layer, we can put them together in one formula, summarizing backpropagation, in the presence of L2-norm loss and sigmoid activations.

$$\frac{\partial L}{\partial w_{ij}} = \delta_j x_i$$

where for a hidden layer

$$\delta_j = \sum_k \delta_k w_{jk} y_j (1 - y_j)$$

Kudos to those of you who got to the end.

Thanks for reading.