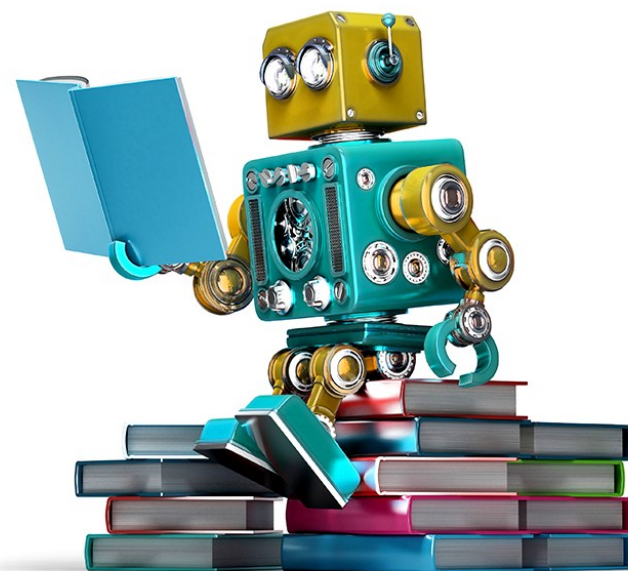


The Mathematics behind AI

(well, only a few first steps)

Wolfgang Waltenberger

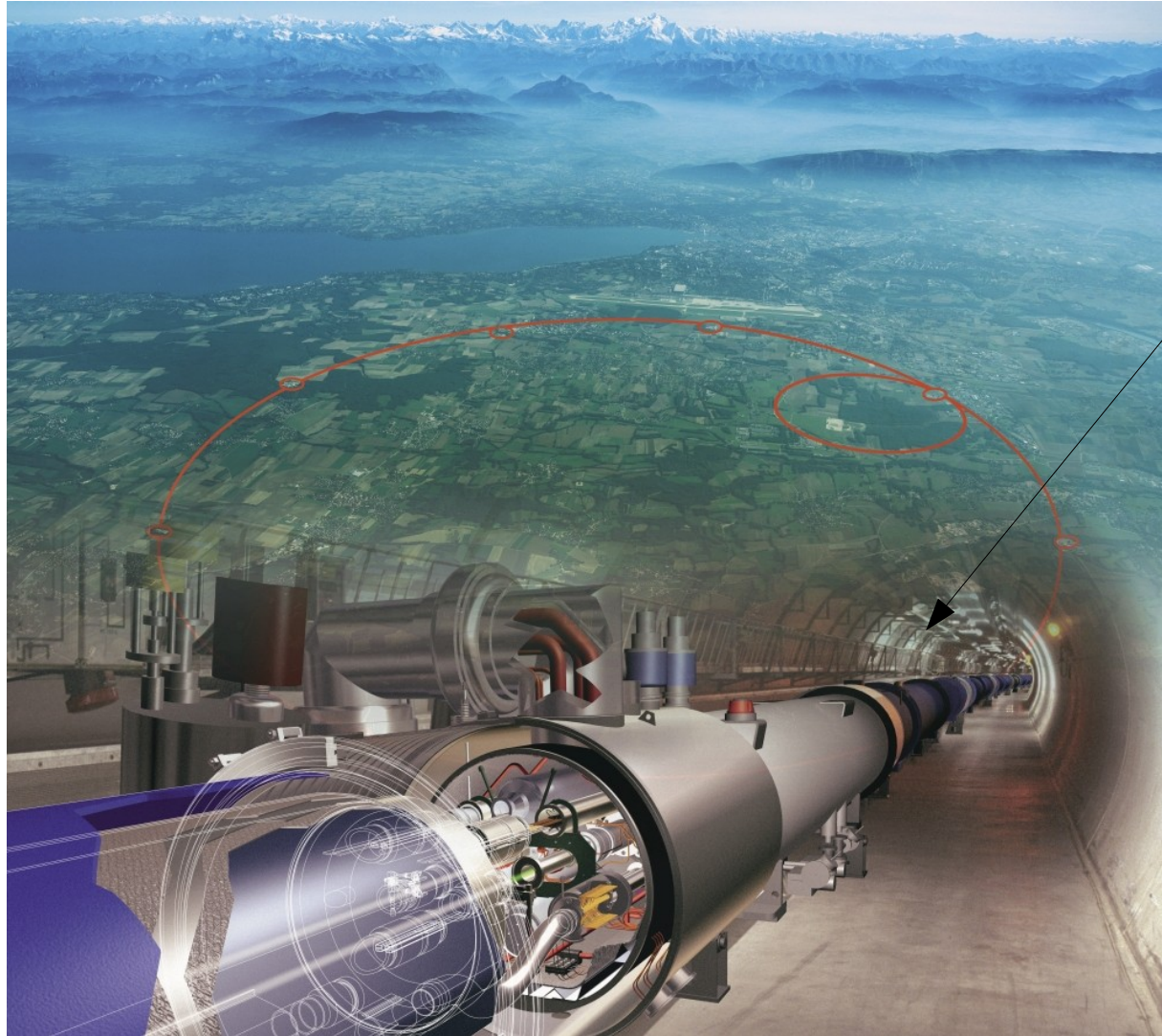
Seminar Studienstiftung Artificial Intelligence,
Feb 12 - 14, 2020





These are pictures of celebrities. Some of them have never been born.
Can you guess which?

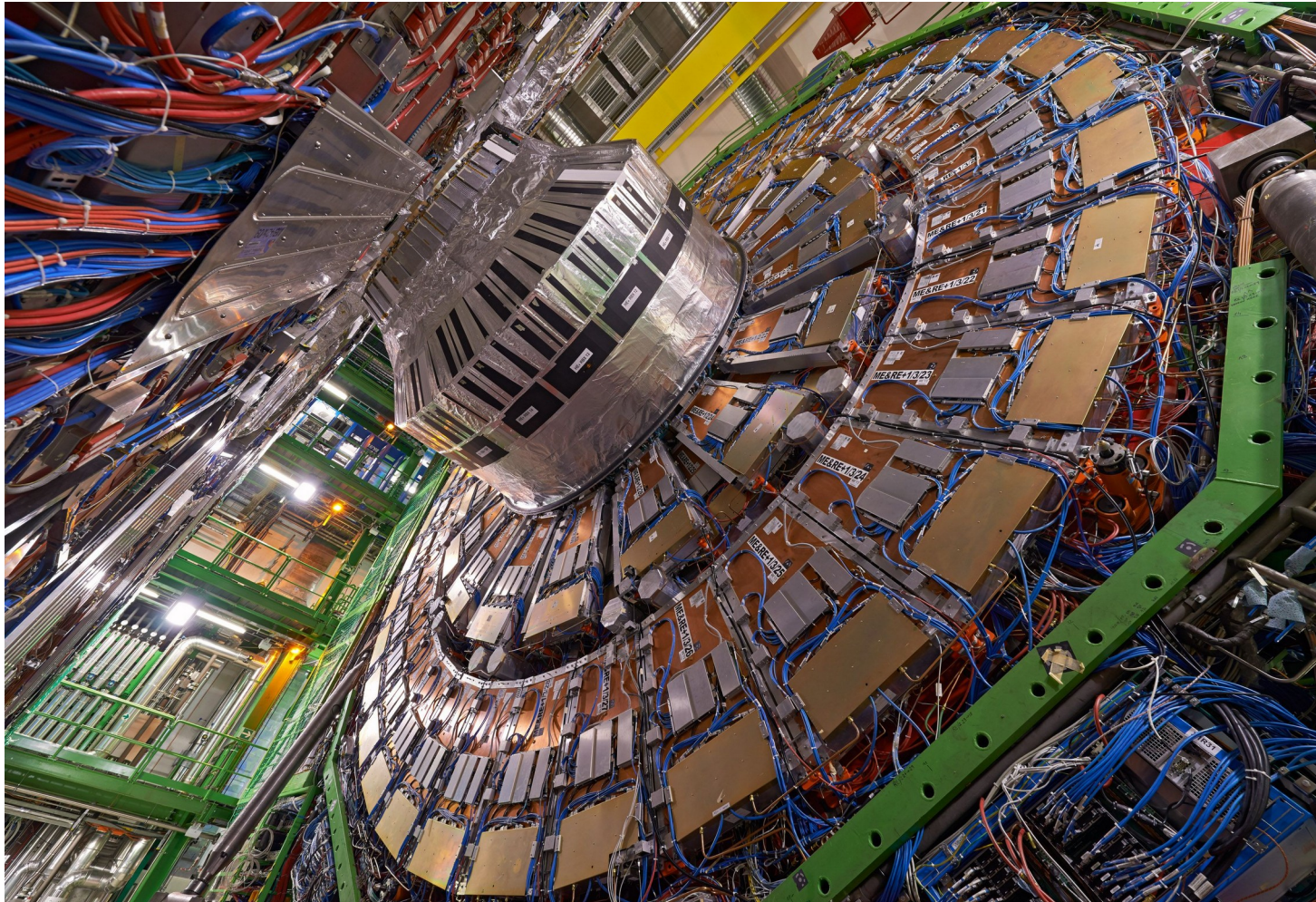
Who am I?



Large Hadron Collider,
30 km particle accelerator,
world's largest experiment

- Particle physicist, PhD 2004 in data science applied to data from the Large Hadron Collider (LHC).

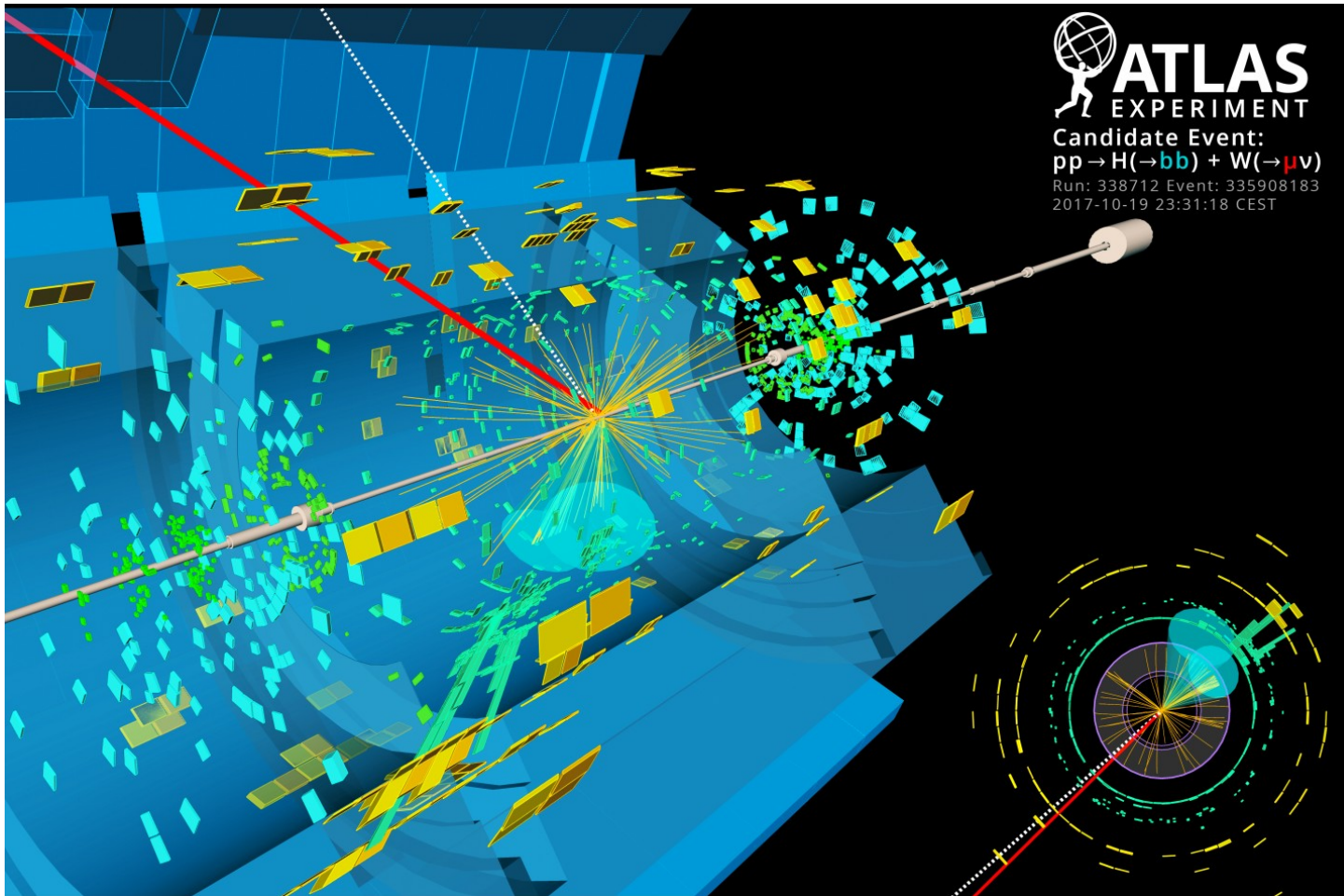
Who am I?



our detectors take data at an enormous rate: counting all data we are currently close to **1 exabyte** (10^{18} bytes, 1 million TB) of data that we process on almost million CPU cores: **we need all the** data science and machine learning **smartness** that **we can get!**

- Particle physicist, PhD 2004 in applied data science for the Large Hadron Collider (LHC).

Who am I?



In 2012 we, together with our 6000 colleagues, found the long sought Higgs boson!

In 2013 the Nobel prize in physics went to the prediction of the Higgs boson (so not us – we discovered it, we did not predict it).

- Particle physicist, PhD 2004 in applied data science for the Large Hadron Collider (LHC).

Who am I?

I teach at the Uni Vienna and TU Vienna – these days data science courses only.

- WS 2017/2018
 - 142.094, 260129, Vorlesung "Astro-particle physics", gemeinsam mit [Claudia Wulz](#)
 - 142.351, Übungen "Statistische Methoden der Datenanalyse", (Vorlesung [Rudi Frühwirth](#))
- Sommer 2018
 - Vorlesung + Übung, "Machine learning in particle physics", [DKPI summer school](#)
- WS 2018/2019
 - 260020 Vorlesung "Advanced topics in particle physics", mit A. Hoang, H. Neufeld, [J. Pradler](#)
 - 260032, 142.351 Übungen "Statistische Methoden der Datenanalyse", (Vorlesung [Rudi Frühwirth](#))
- WS 2019/2020
 - 260032, 142.340, 142.351, Vorlesung + Übungen "Statistische Methoden der Datenanalyse"

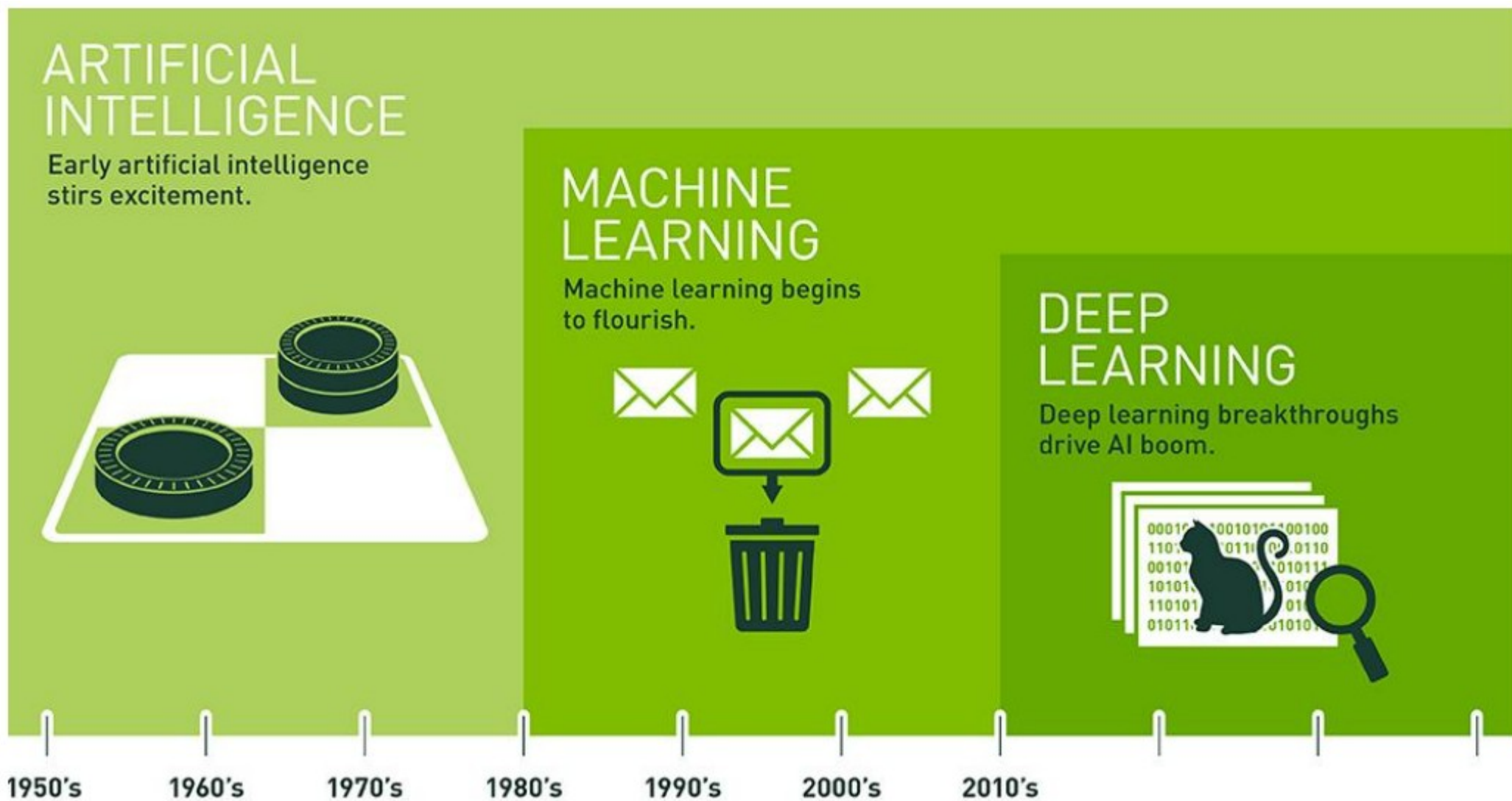
My plans for today

1. **Walk you through the math** behind (deep) neural networks, up to the (currently) standard method for how neural networks are trained:
“**backpropagation**”
2. Discuss (superficially) how to go on from there
3. **Play with neural networks** in the browser, change architecture, see how they perform
4. Optional, for the “pros”: play with some simple python implementation of a neural network.

New concepts require a new language

(so lets introduce a few new words and discuss
their meanings)

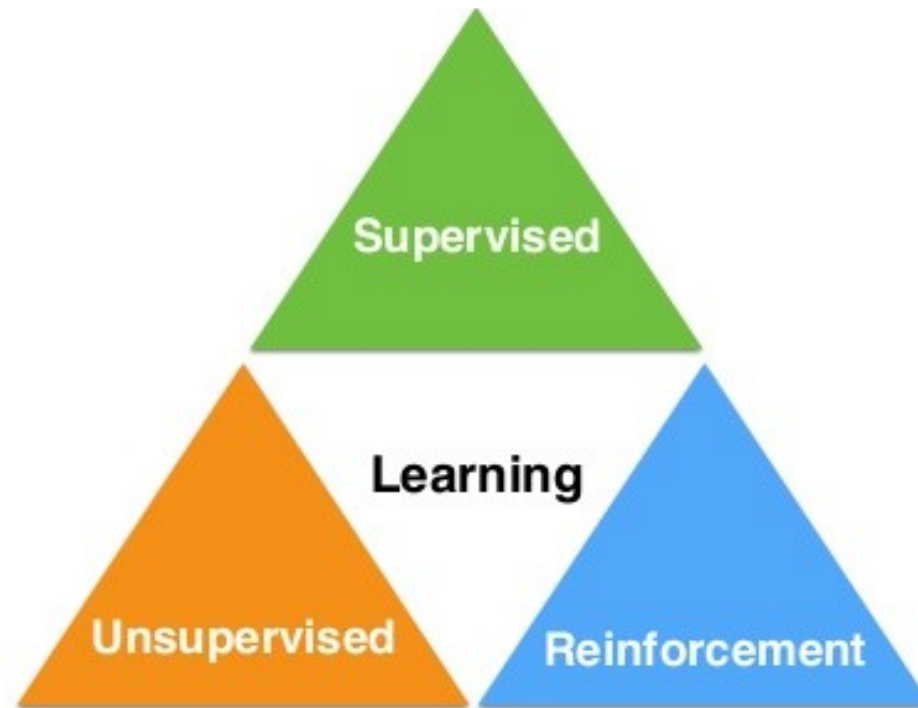
We tend to mix up terms: artificial intelligence, machine learning, (deep) neural networks. The specialists see deep neural networks as a special case of machine learning, and machine learning as a special (though very important) case of artificial intelligence.



So: not all of artificial intelligence is about neural networks, though all the recent “hype” is about deep neural networks, and I will be talking about these.

When we talk about learning, we think of three broad types of learning: supervised, unsupervised, and reinforcement learning.

Supervised: we humans supervise=teach the machine. For example, we show the machine many pictures of cats, and many pictures of dogs, we tell the machine “here is a cat picture”, “here is a dog picture”, and the machine has to learn to understand what is on the picture.

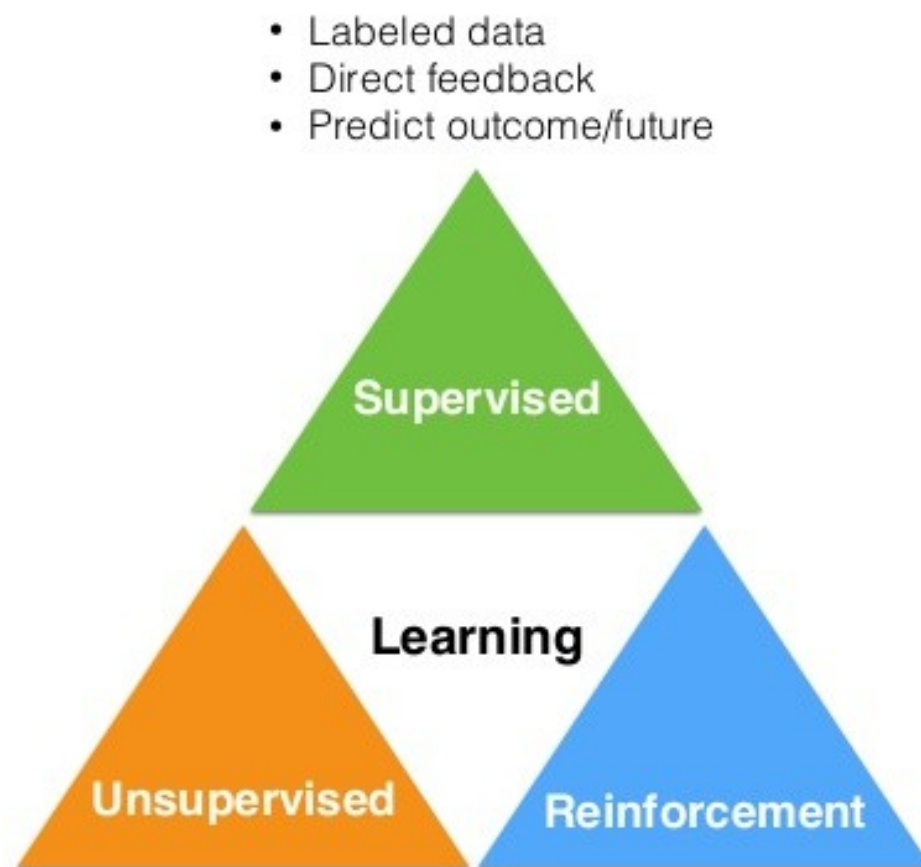


Unsupervised: the machine has to find “patterns” in the data all by itself. Example: the machine gets all the information about how we behave, what we buy, what we look at, on amazon.at. Now it needs to find “stereotypes” of amazon customers.

Reinforcement: the machine has to learn to solve a task. Example: a self-driving racing car. The goal is to drive the car:

-) and not crash
-) and finish first

In this lecture I will try to explain how we use neural networks to solve a **supervised** learning task. Neural networks, however, can also be used to solve unsupervised and reinforcement learning tasks. And also other methods exist to solve supervised learning tasks.



This afternoon, you will hack a bit on a simple neural network, implemented in Facebook's "pytorch" framework, written in python.

Features, Labels, Classification, Regression

In supervised learning, we have “features”, and

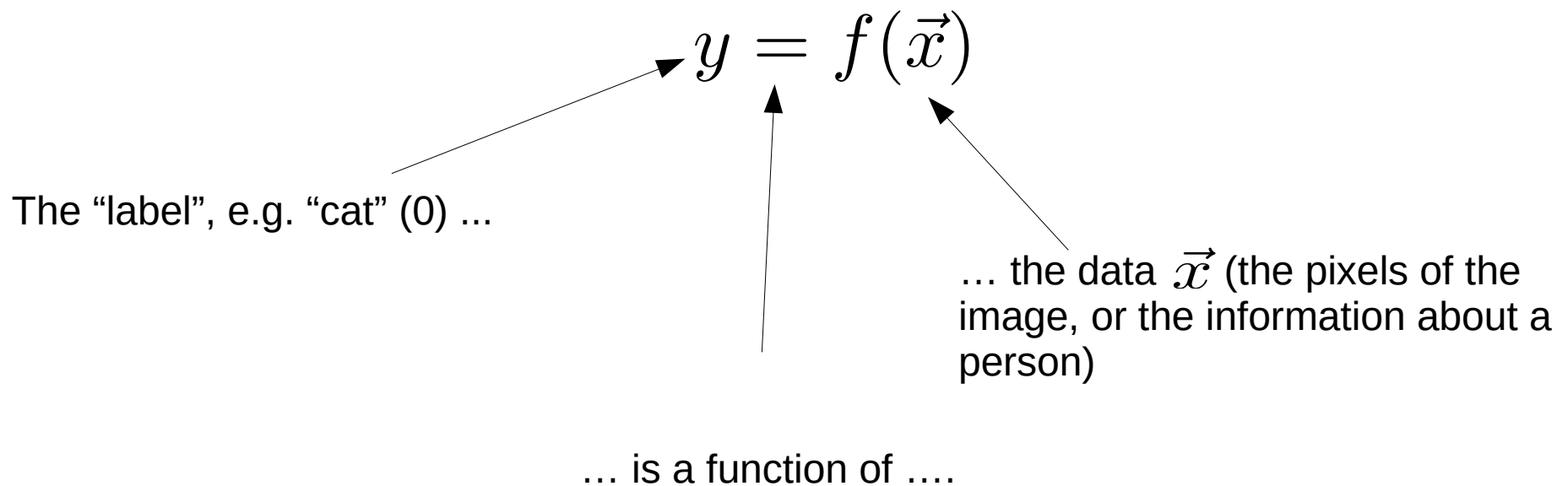
“labels”. “**Features**” are the data we have. We denote them with “**x**”. Examples are all pixels of an image, or data describing a person, like gender, age, height, weight, etc. Usually they are high dimensional, we can interpret them as a very big vector, \vec{x}

The machine will have to learn a “**label**”. We denote the label with “**y**”. If the machine should learn to tell a cat from a dog, then y is an element of the set { “**cat**”, “**dog**” }. We translate the elements to natural numbers though, for example cat=0, dog=1. Problems of this type are called “**classification tasks**”.

If the machine should learn to guess your age from a portrait of you, then y would be an element of all positive real numbers. This type of problem is called “**regression**”.

The function to be learned

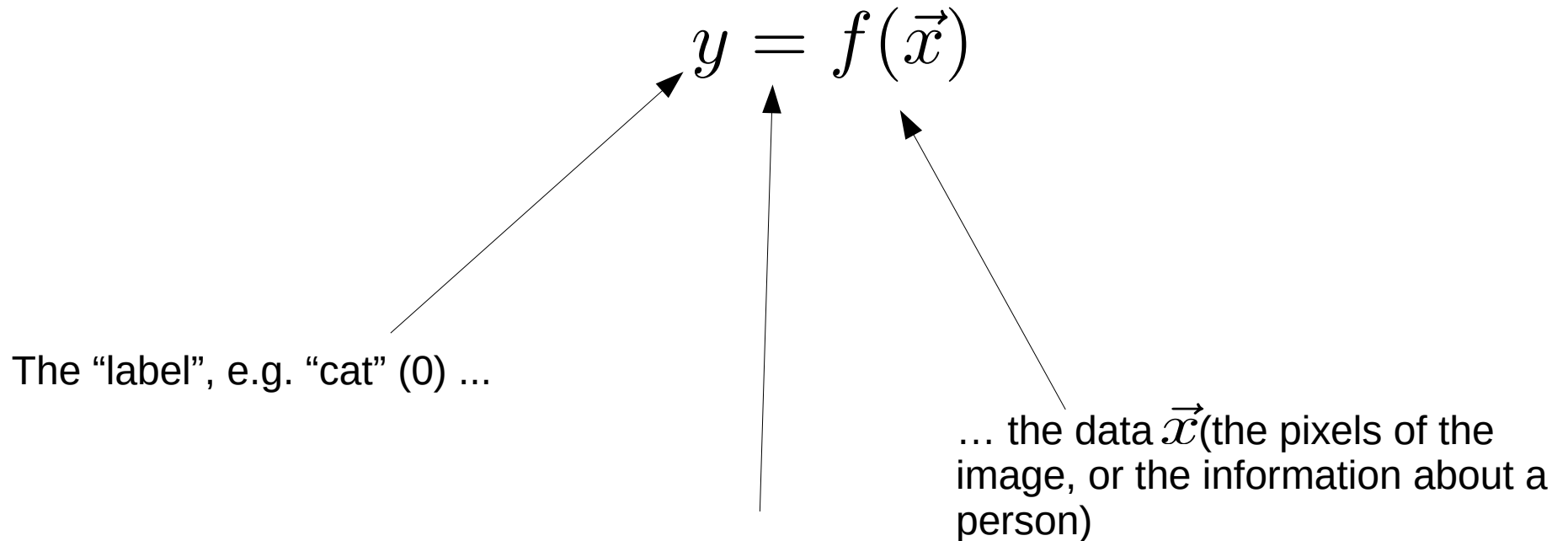
With this vocabulary, we can already phrase our problem mathematically. What our machine needs to find is this function:



The function to be learned

If this function is very simple (like, say, “ $y=3x+2$ ”), or if we already know it explicitly (because we know the “laws” behind the data), then we don't use machine learning. **We use (supervised) machine learning when the function is super complicated, and/or when we only have (many) examples – pairs of x and y values – but not the function itself.**

Capisci?



... is a function of

How can a function be “learned”?

So the question is: how do we “learn” this function:

$$y = f(\vec{x})$$

From examples alone?

How can a function be “learned”?

So the question really is: how do we learn this function:

$$y = f(\vec{x})$$

from examples alone?

Here is the answer:

- we **start with** a completely **random** function (its performance will be lousy).
- we introduce another function – we will call it the “**loss function**” (objective function, energy function) that **tells us how well our** completely random **function is solving the task**.
- we have an **algorithm** that will **change** our completely random function, (hopefully) **for the better**. (The algorithm will be called “backpropagation”).
- we **repeat** until we have a good function.

How can a function be “learned”?

So the question really is: how do we learn this function:

$$y = f(\vec{x})$$

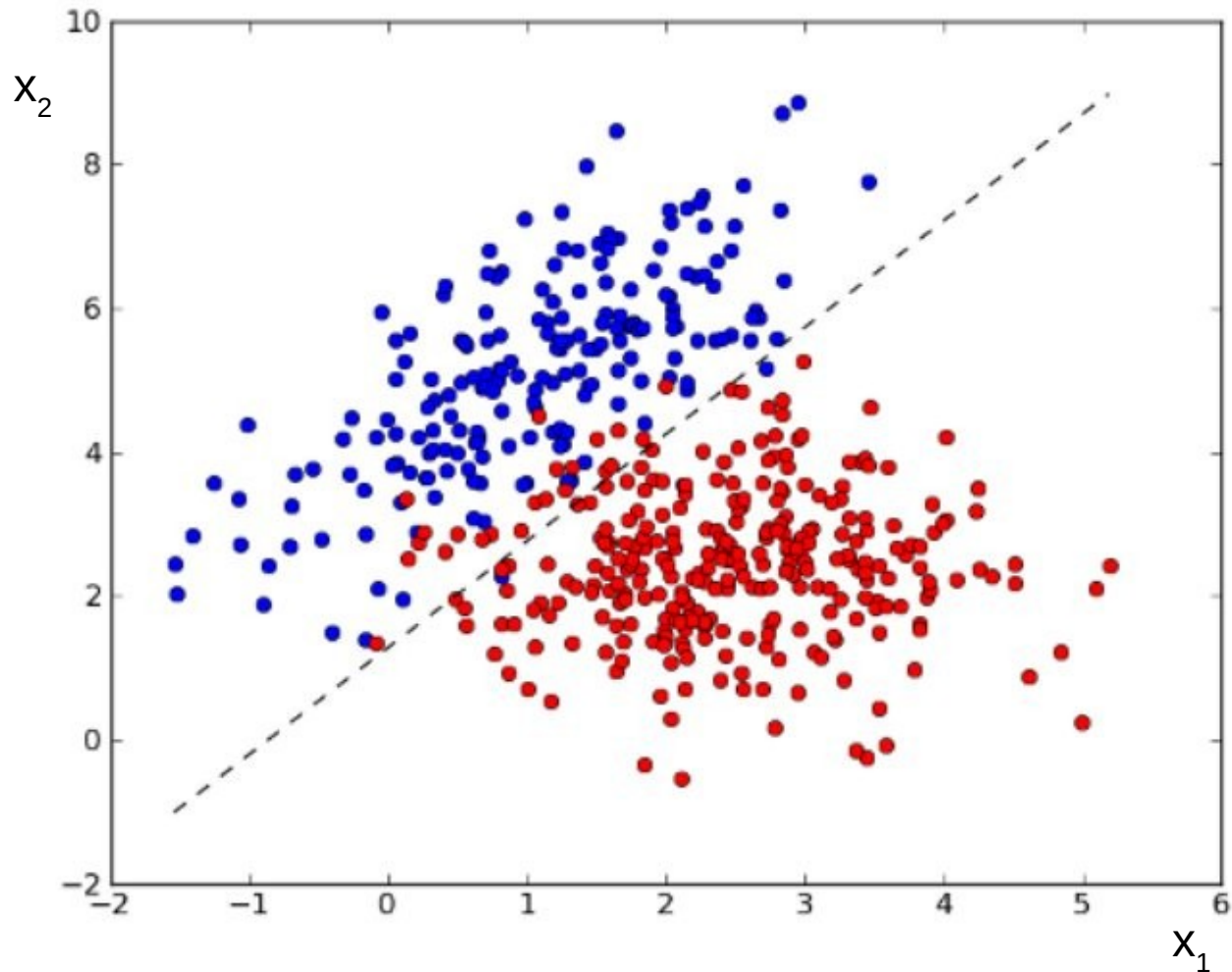
from examples alone?

Here is the answer:

- we start with a completely random function
- we have another function that tells us how well our completely random function is solving the task.
- we have an algorithm that will change our completely random function, hopefully for the better.
- we repeat until we have a good function.

This is very abstract – let’s visualize a simple machine learning problem

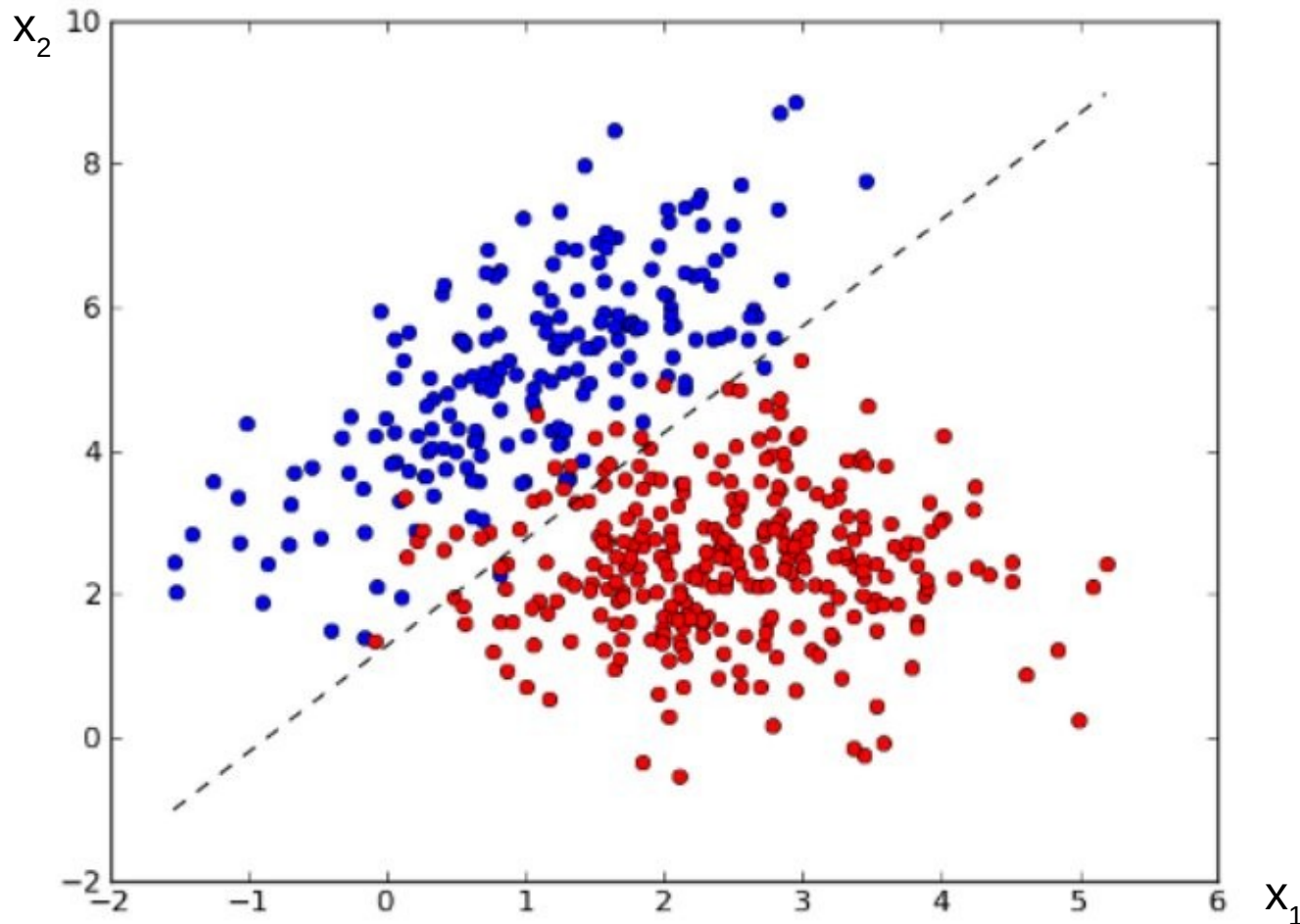
How can a function be “learned”?



The coordinates of the plot are our data. So “x” here is two-dimensional, the y-axis of the plot is the second coordinate of “x”, x_2 (confusing, huh?).

The color is our label. Lets say blue is $y=0$, red is $y=1$. The task for the machine is: given the coordinates of the point, $x=(x_1, x_2)$, tell me the color y of the point.

How can a function be “learned”?



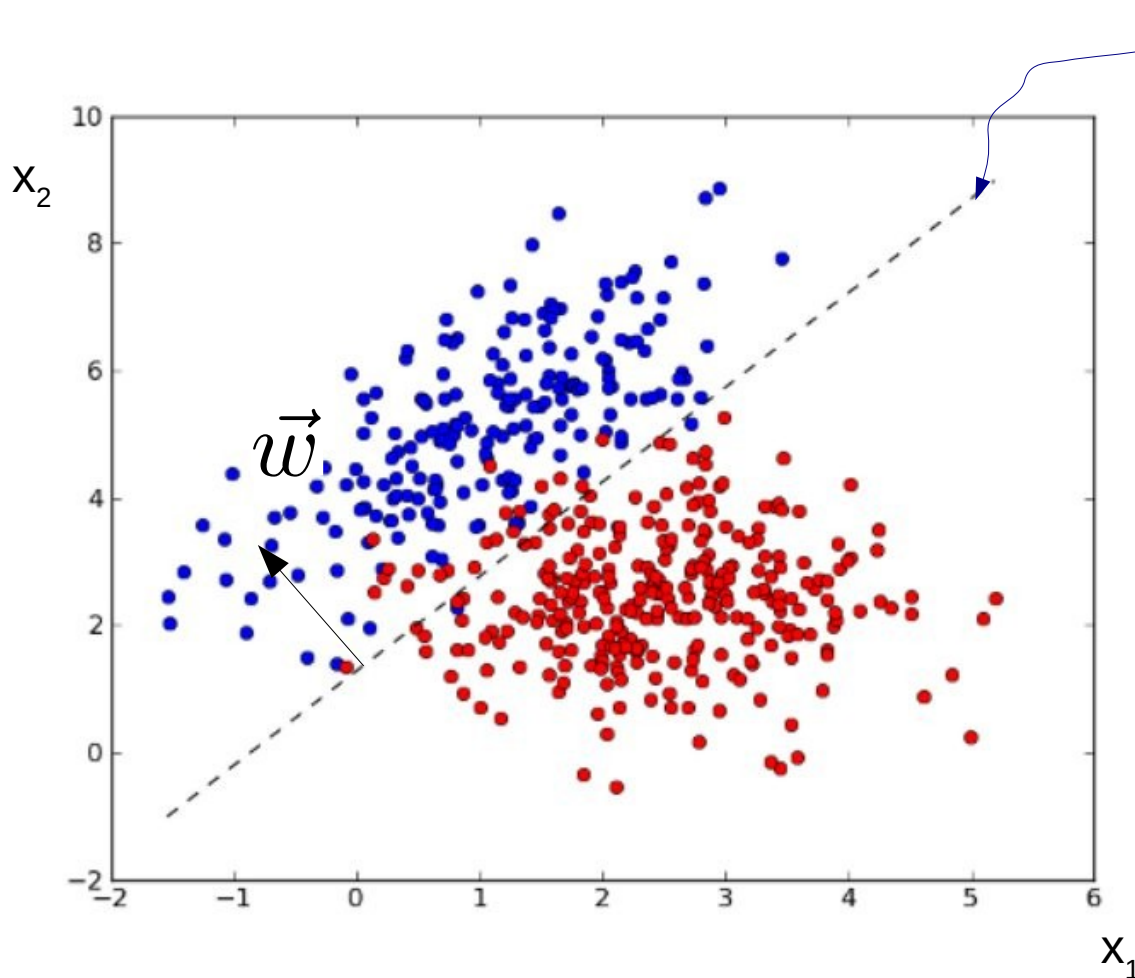
The function that needs to be learned assigns a label (blue or red, zero or one) to every point in this plane, agreed? But let's just visualize the “decision boundary”, that is the line where the decision of the machine changes from red to blue. This line does not have to be straight, it can be very “crooked”.

What is the best *straight* line you can think of?

What is the best line you can think of?

Logistic regression

Let's try to construct the “loss function” for the “straight line boundary”. The loss function should tell us, how good a solution we found. We need two ingredients: **orthogonal vectors**, and **scalar products**!



$$\vec{x} = (x_1, x_2)$$

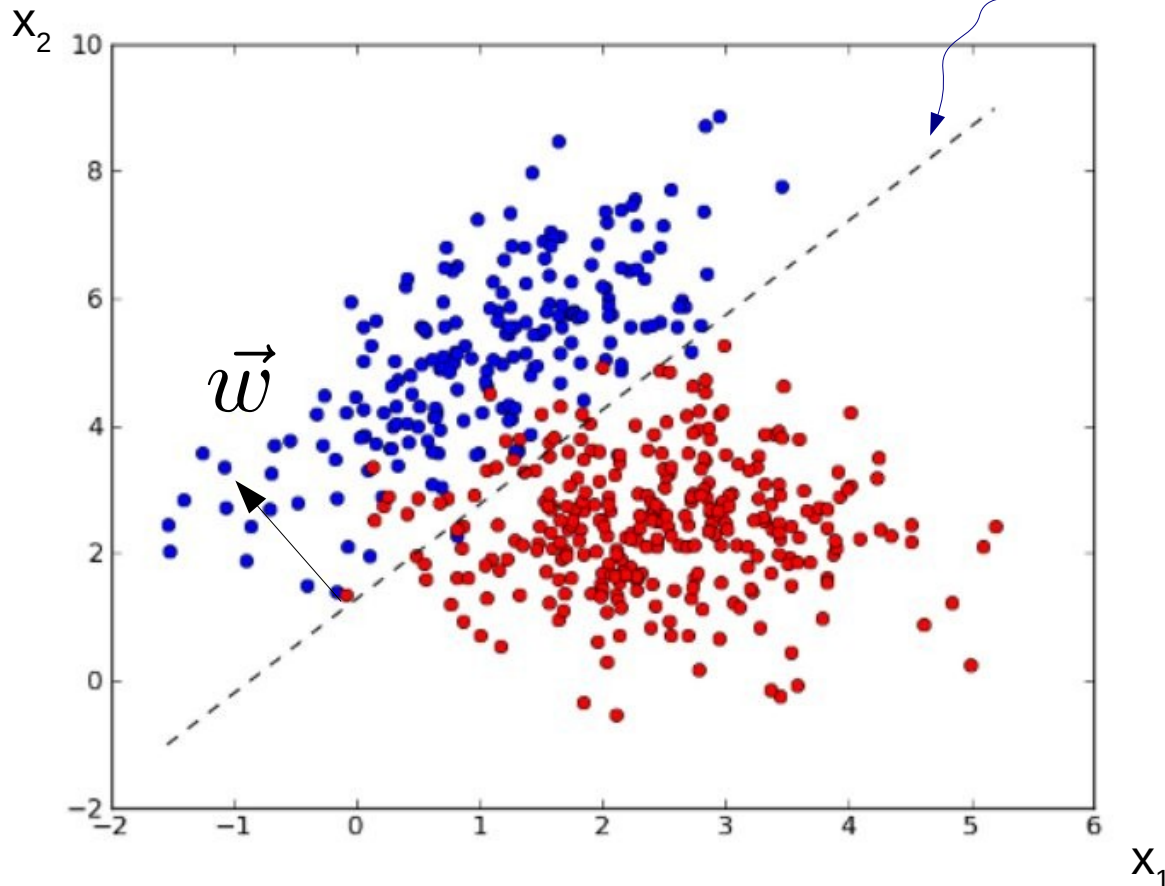
$$x_2 = kx_1 + d$$

orthogonal
vector $\vec{w} := \begin{pmatrix} -k \\ 1 \end{pmatrix}$

$$D(\vec{x}; \vec{w}) := \vec{w} \cdot \vec{x} + d$$

$$\begin{cases} D(\vec{x}; \vec{w}) > 0 \rightarrow \text{blue} \\ D(\vec{x}; \vec{w}) < 0 \rightarrow \text{red} \end{cases}$$

Logistic regression



$$x_2 = kx_1 + d$$

orthogonal
vector

$$\vec{w} := \begin{pmatrix} -k \\ 1 \end{pmatrix}$$

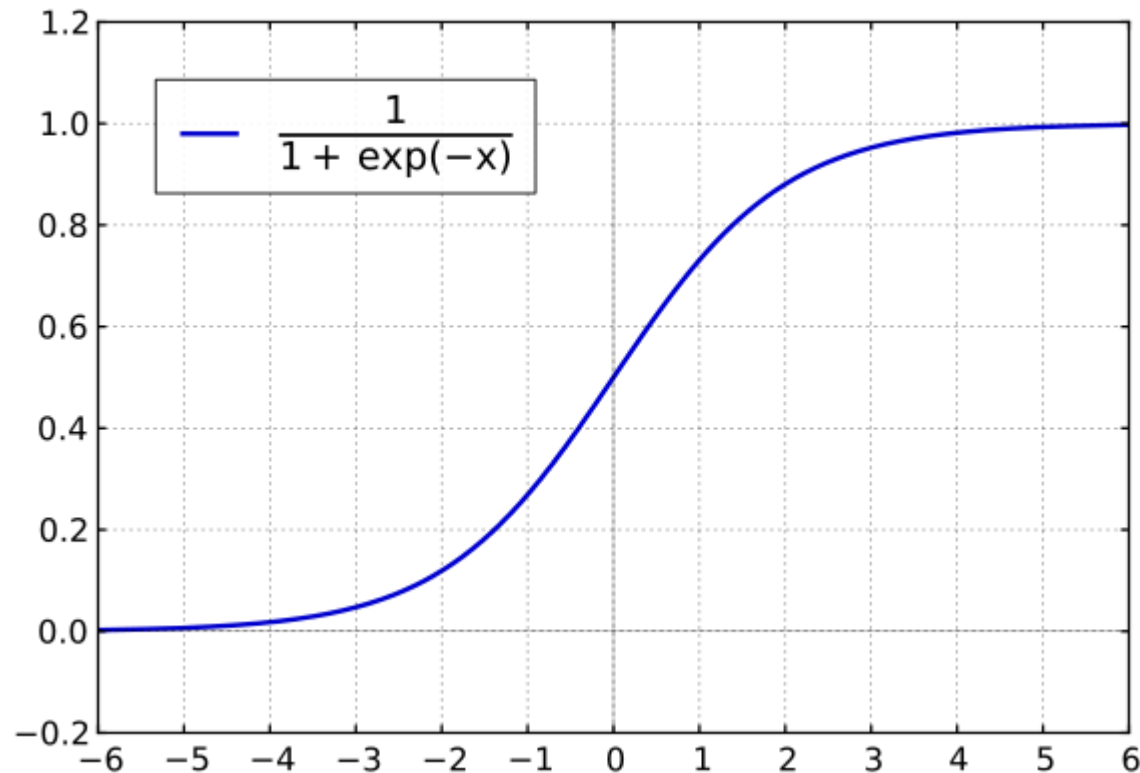
$$D(\vec{x}; \vec{w}) := \vec{w} \cdot \vec{x} + d$$

$$\begin{cases} D(\vec{x}; \vec{w}) > 0 \rightarrow \text{red} \\ D(\vec{x}; \vec{w}) < 0 \rightarrow \text{blue} \end{cases}$$

If you did not fully understand the last slide, remember this: $D(\vec{x}; \vec{w})$ is positive for the points **above** the line, it is negative for the points **below** the line. It is a measure of the distance: the further away, the larger the number. Points **on** the line are $D(\vec{x})=0$. Ok?

Logistic regression

Now let's turn this into something like a “probability” that the point is blue or red. We use this function “ σ ” (pronounce “sigma”, or “activation function”)



If we apply this to our function “D”, $\sigma(D(\vec{x}; \vec{w}))$ then for a “very good line”, we get 1.0 for points that are clearly blue, 0. for points that are clearly red, 0.5 for points that are exactly in between.

Awesome, right? This function gives us a “probability”, that the point is blue!
We use this function to **predict** if the point is blue!

Logistic regression

If we apply this to $D(x)$ for a very good solution, we get 1.0 for points that are clearly blue, 0. for points that are clearly red, 0.5 for points that are exactly in between.

Awesome, right? This function gives us a “probability” that the point is blue!

Now if we subtract the “true” value (1.0 for blue, 0.0 for red) and take the absolute value of the difference:

$$l(\vec{x}) = |y - \sigma(D(\vec{x}; \vec{w}))|$$

then we have something that tells us how well our algorithm predicts the color of a given point, an “error” or “loss” function, right?

The worse the prediction, the higher this number. The function is close to zero, if we predict correctly. It is close to one, if we are “maximally off”.

Logistic regression

Now that we have a “loss function” for one data point,

$$l(\vec{x}) = |y - \sigma(D(\vec{x}; \vec{w}))|$$

we construct one for many data points. How do we do that? We simply sum up the “losses”. The sum tells us how well we are doing on all the data, right?

$$L(\vec{x}_i) = \sum_i |y_i - \sigma(D(\vec{x}_i; \vec{w}))|$$

Here, we introduced the index “i”. The first data point has index $i=1$, the second has $i=2$, and so on. For every i we have a data vector \vec{x}_i and a label y_i .

The orthogonal vector \vec{w} is all about the decision boundary. It has no index. You understand why?

Okay, if you did not understand everything perfectly, no problem. What is important, though, is that you understand that **we now have a function**

$$L(\vec{x}_i) = \sum_i |y_i - \sigma(D(\vec{x}_i; \vec{w}))|$$

... **that tells us, if a certain line is a good decision boundary**. So now we need to find the line for which the function “L” above is as low as possible.

We need to minimize the loss function “L” as a function of \vec{w}

Okay, if you did not understand everything perfectly, no problem. What is important, though, is that you understand that we now have a function

$$L(\vec{x}_i) = \sum_i |y_i - \sigma(D(\vec{x}_i; \vec{w}))|$$

... that tells us, if a certain line is a good decision boundary. So now we need to find the line for which the function “L” above is as low as possible.

We need to minimize the loss function “L” as a function of \vec{w}

So, how does one find the minimum of a function?

Okay, if you did not understand everything perfectly, no problem. What is important, though, is that you understand that we now have a function

$$L(\vec{x}_i) = \sum_i |y_i - \sigma(D(\vec{x}_i; \vec{w}))|$$

... that tells us, if a certain line is a good decision boundary. So now we need to find the line for which the function “L” above is as low as possible.

We need to minimize the loss function “L” as a function of \vec{w}

So, how does one find the minimum of a function?

The “**high school way**”: You compute the derivative of the function, you set the derivative to zero.

Okay, if you did not understand everything perfectly, no problem. What is important, though, is that you understand that we now have a function

$$L(\vec{x}_i) = \sum_i |y_i - \sigma(D(\vec{x}_i; \vec{w}))|$$

... that tells us, if a certain line is a good decision boundary. So now we need to find the line for which the function “L” above is as low as possible.

We need to minimize the loss function “L” as a function of \vec{w}

So, how does one find the minimum of a function?

The “**high school way**”: You compute the derivative of the function, you set the derivative to zero.

Works very well for simple functions. Won't work for complicated functions.

Okay, if you did not understand everything perfectly, no problem. What is important, though, is that you understand that we now have a function

$$L(\vec{x}_i) = \sum_i |y_i - \sigma(D(\vec{x}_i; \vec{w}))|$$

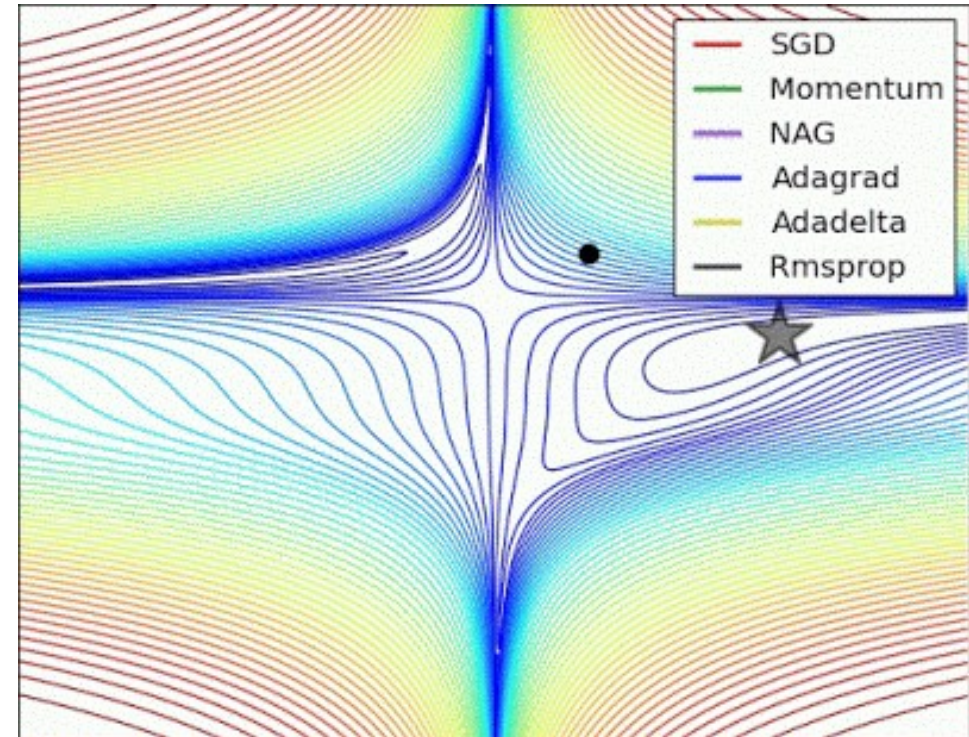
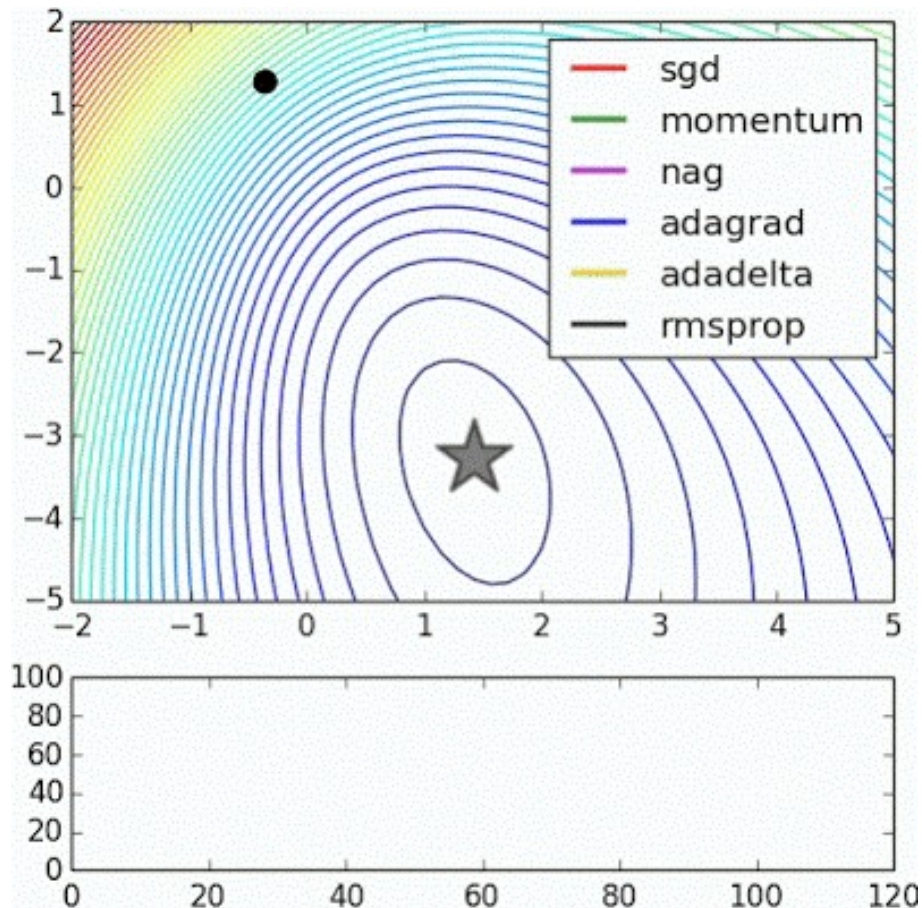
... that tells us, if a certain line is a good decision boundary. So now we need to find the line for which the function “L” above is as low as possible.

We need to minimize the loss function “L” as a function of \vec{w}

So, how does one find the minimum of a function?

The “**physics way**”: You think of “L” as a landscape – like the Alps outside of our windows. You take a ball, you let it roll down the landscape.

Balls rolling down



And this is what “learning” is in neural networks!

Mathematically speaking, balls rolling down in (abstract mathematical) spaces are computed by calculating the “gradient” of the function.

The gradient is the derivative of the function with respect to all parameters “w”.

In our case, we compute the gradient, the derivative of the “loss function” L with respect to all components w_i of \vec{w}

The result is a vector. It tells us how we need to change the components of \vec{w} if we want to have a good chance at arriving at a “lower” loss.

Learning in neural networks is exactly that – computing the gradient of the loss with respect to the parameters “w”. The gradient is a vector that points in the direction where the loss becomes lower. We then take a step in that direction!

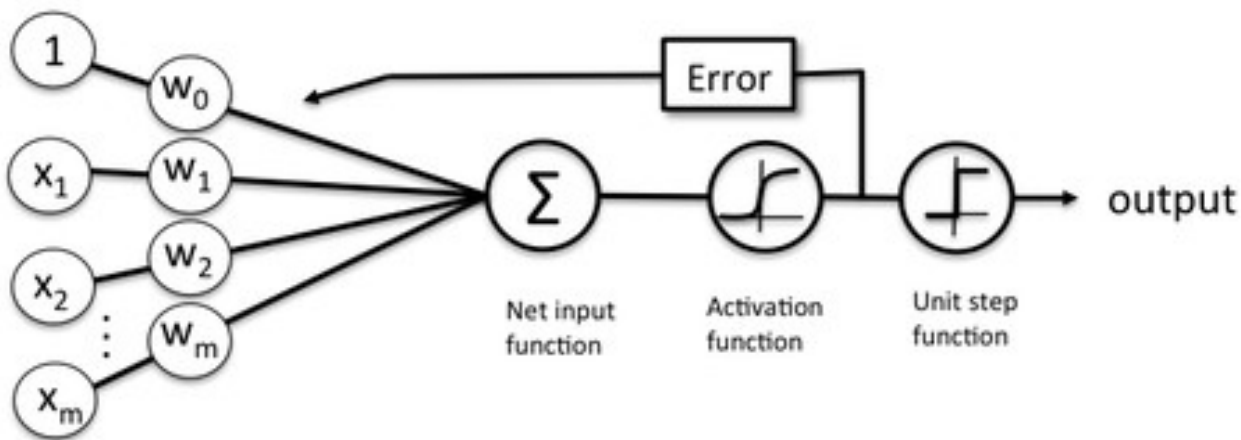
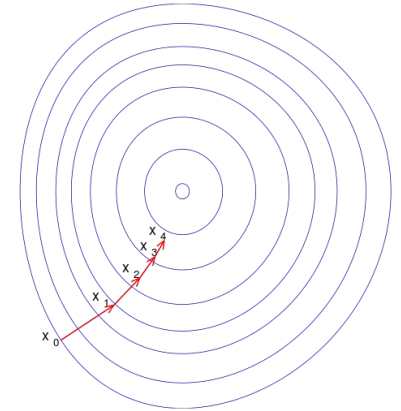
Now we do one more trick: we think of the \vec{w} as weights in a neural network, and the sigmas as the activation functions of the neurons!

Nice metaphor, no?

Learning := gradient descent

Training: find the weights that minimize the loss function → gradient descent!

$$\operatorname{argmin}_{\vec{w}} L(\vec{w})$$

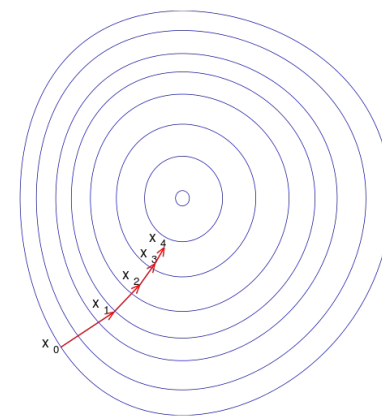


Schematic of a logistic regression classifier.

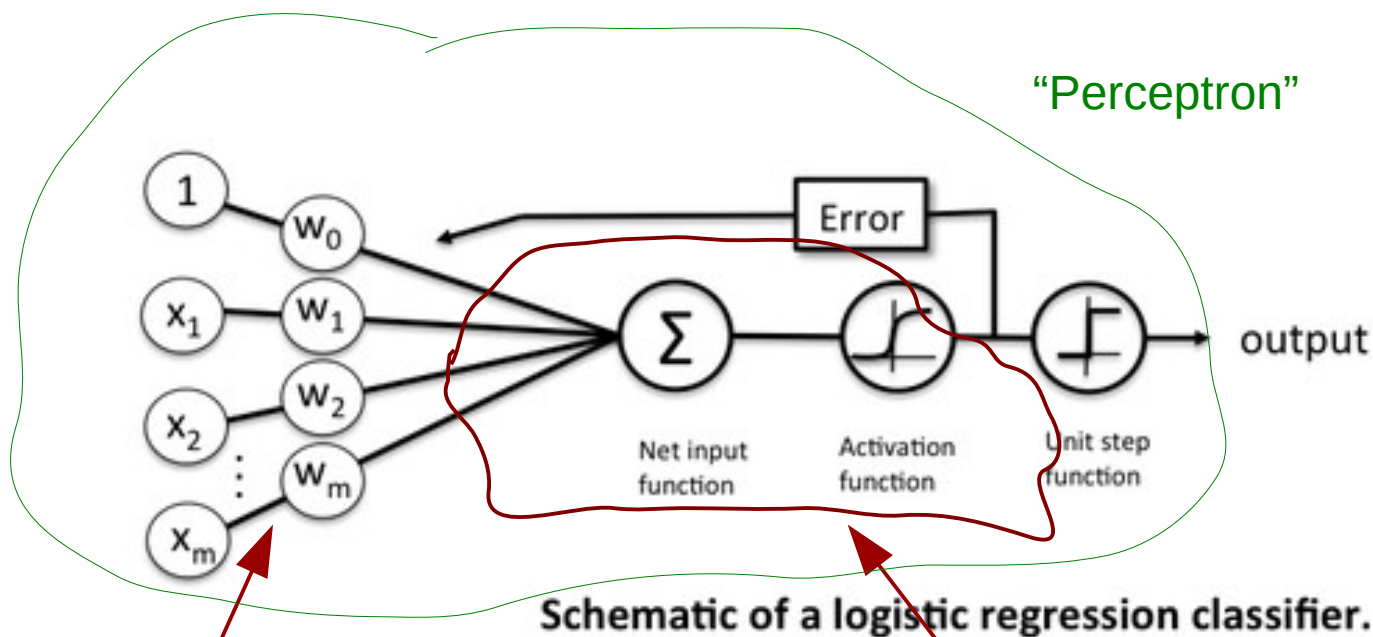
Learning is gradient descent

Training: find the weights that minimize the loss function → gradient descent!

$$\operatorname{argmin}_{\vec{w}} L(\vec{w})$$



“Perceptron”

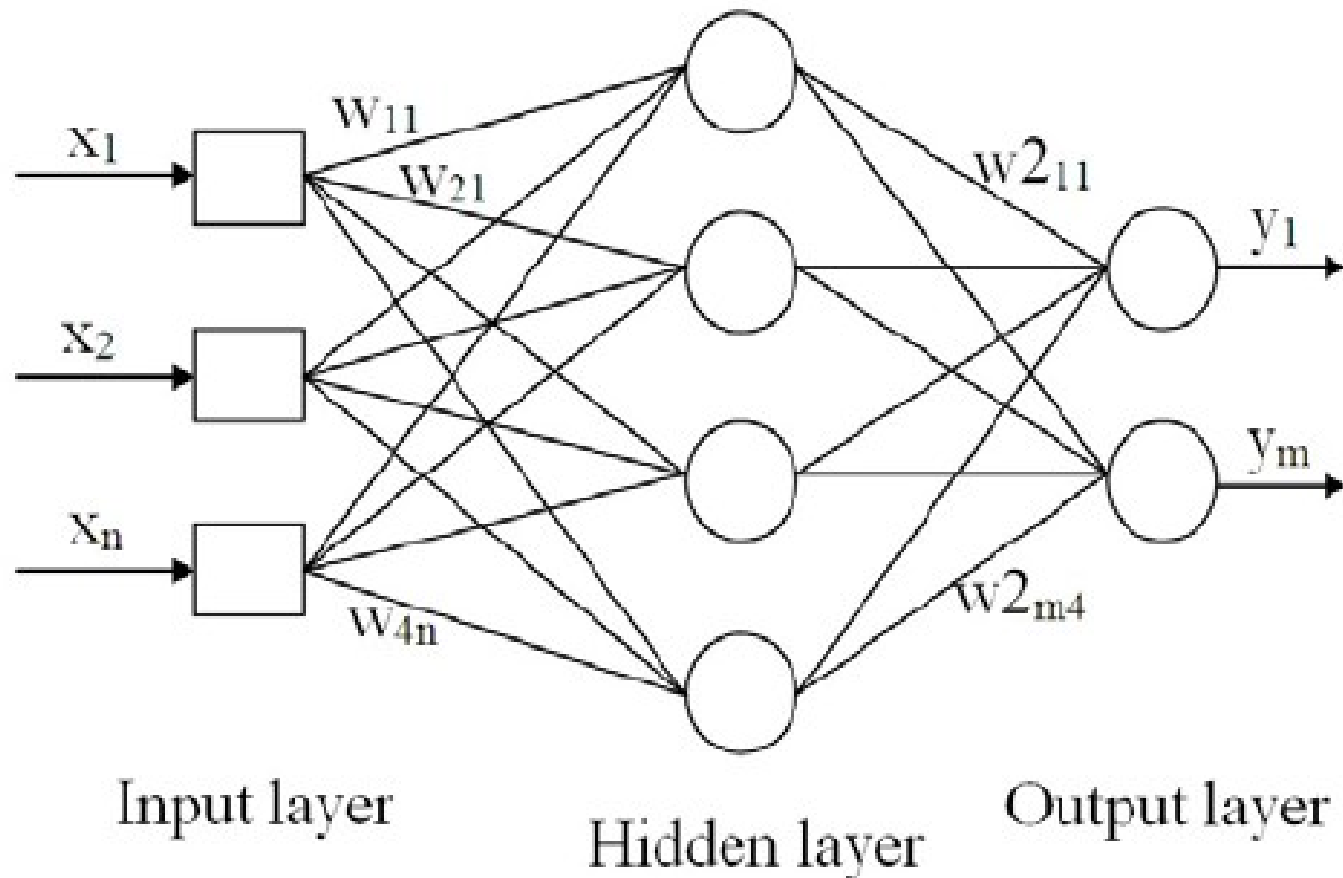


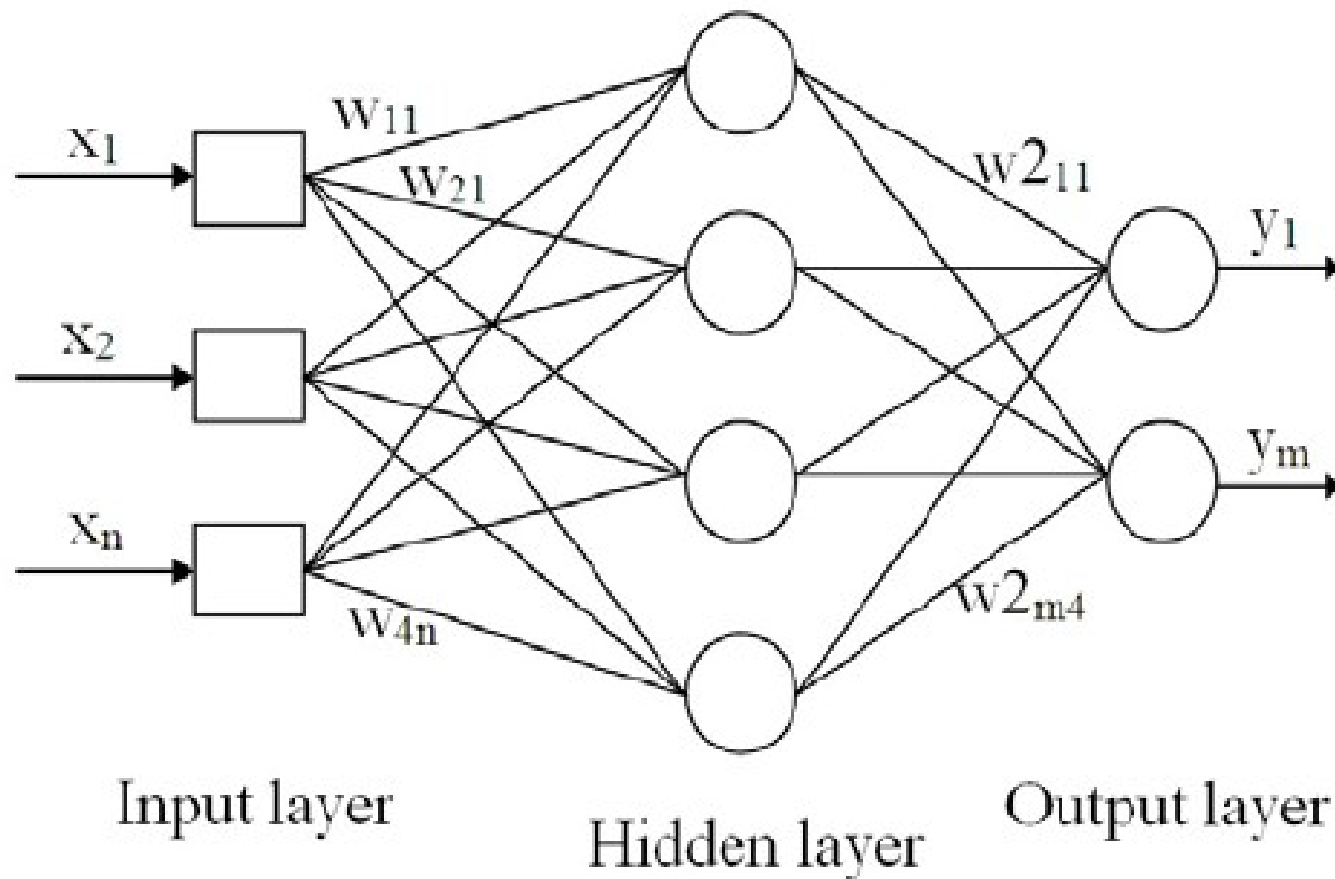
“Synapse”

“Neuron”

Schematic of a logistic regression classifier.

And now we can make such networks more complicated, by adding more (“hidden”) layers:

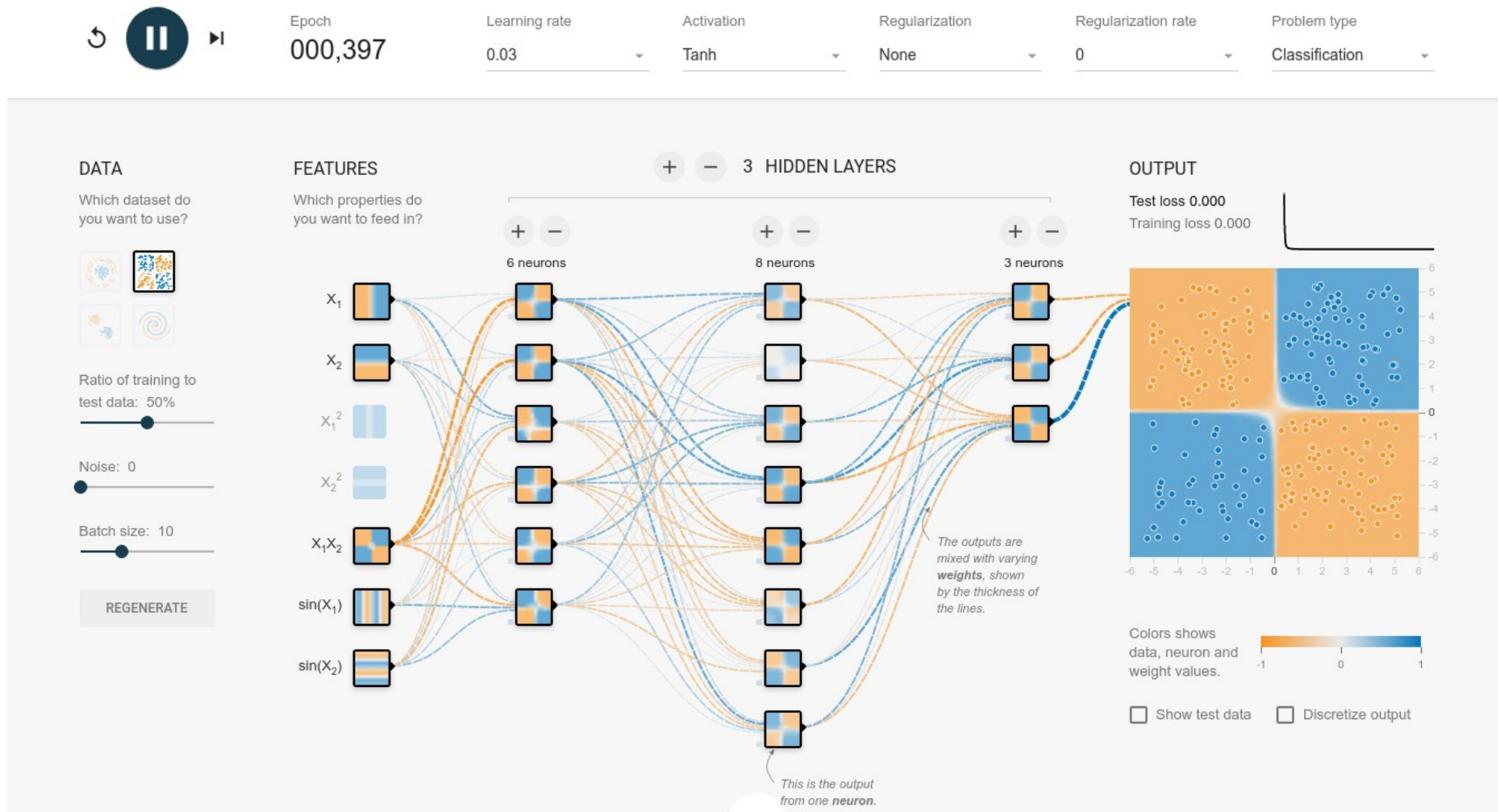




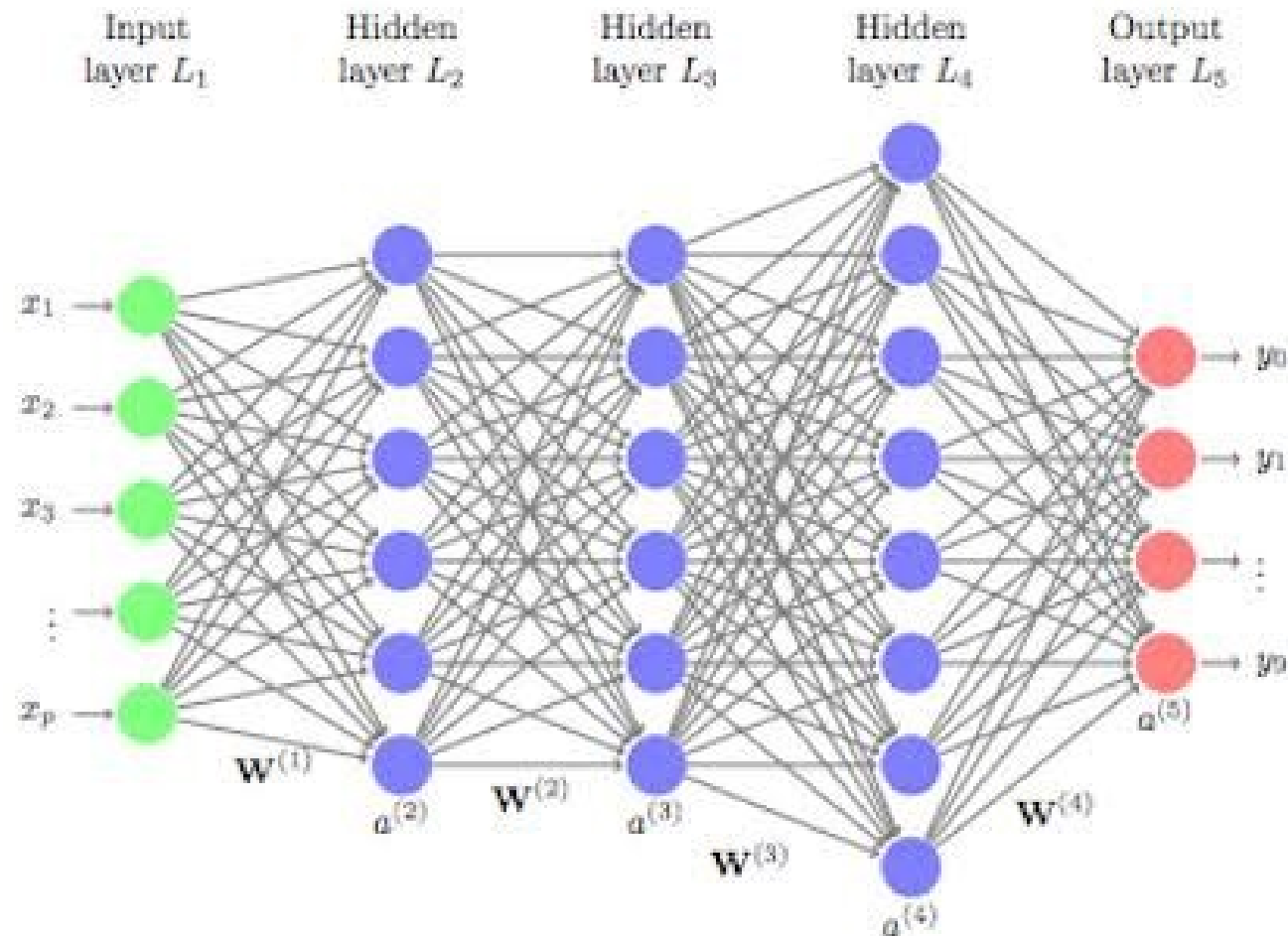
It so happens (and this was proven by an Austrian and some other guy) that **once you add one “hidden layer”, you can “learn” any decision boundary – not just straight lines!**

It so happens (and it was proven by an Austrian and some other guy) that once you add one “hidden layer”, you can “learn” any decision boundary – not just straight lines!

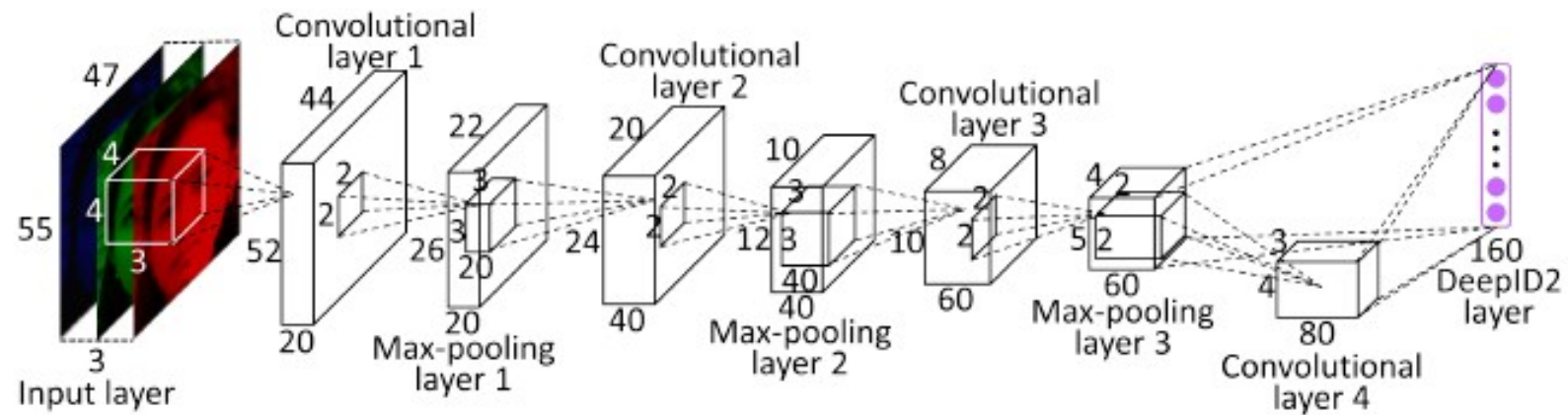
Tinker With a **Neural Network** Right Here in Your Browser.
Don't Worry, You Can't Break It. We Promise.



... and often we add even more (“hidden”) layers ...

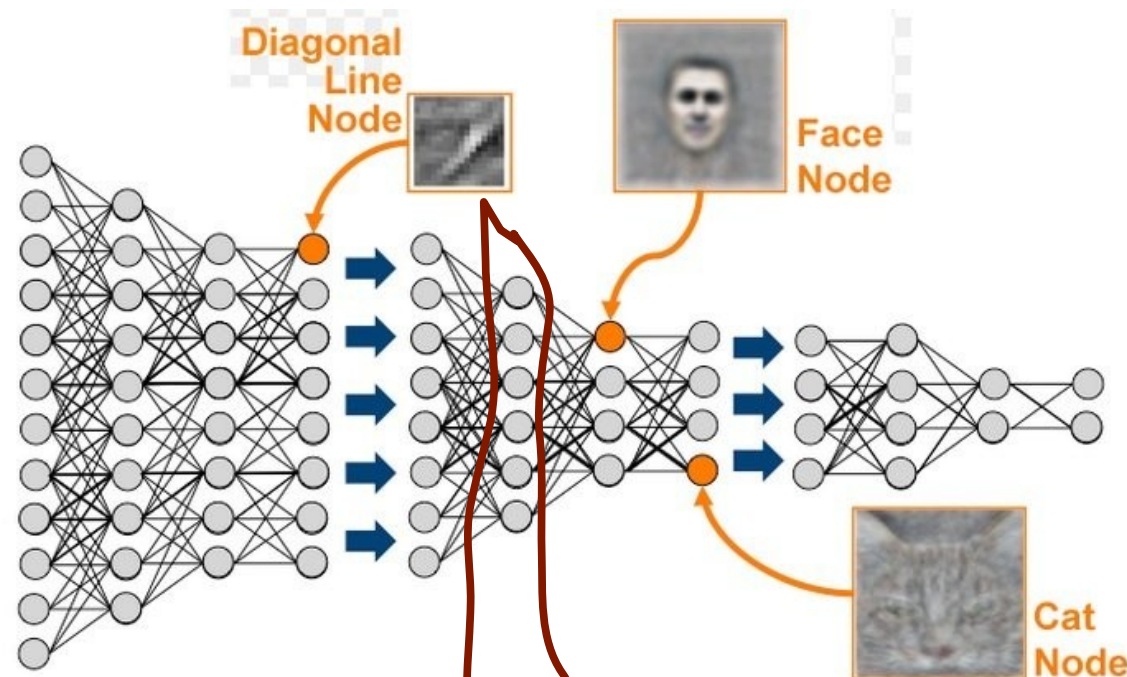


... and more complicated layers ...



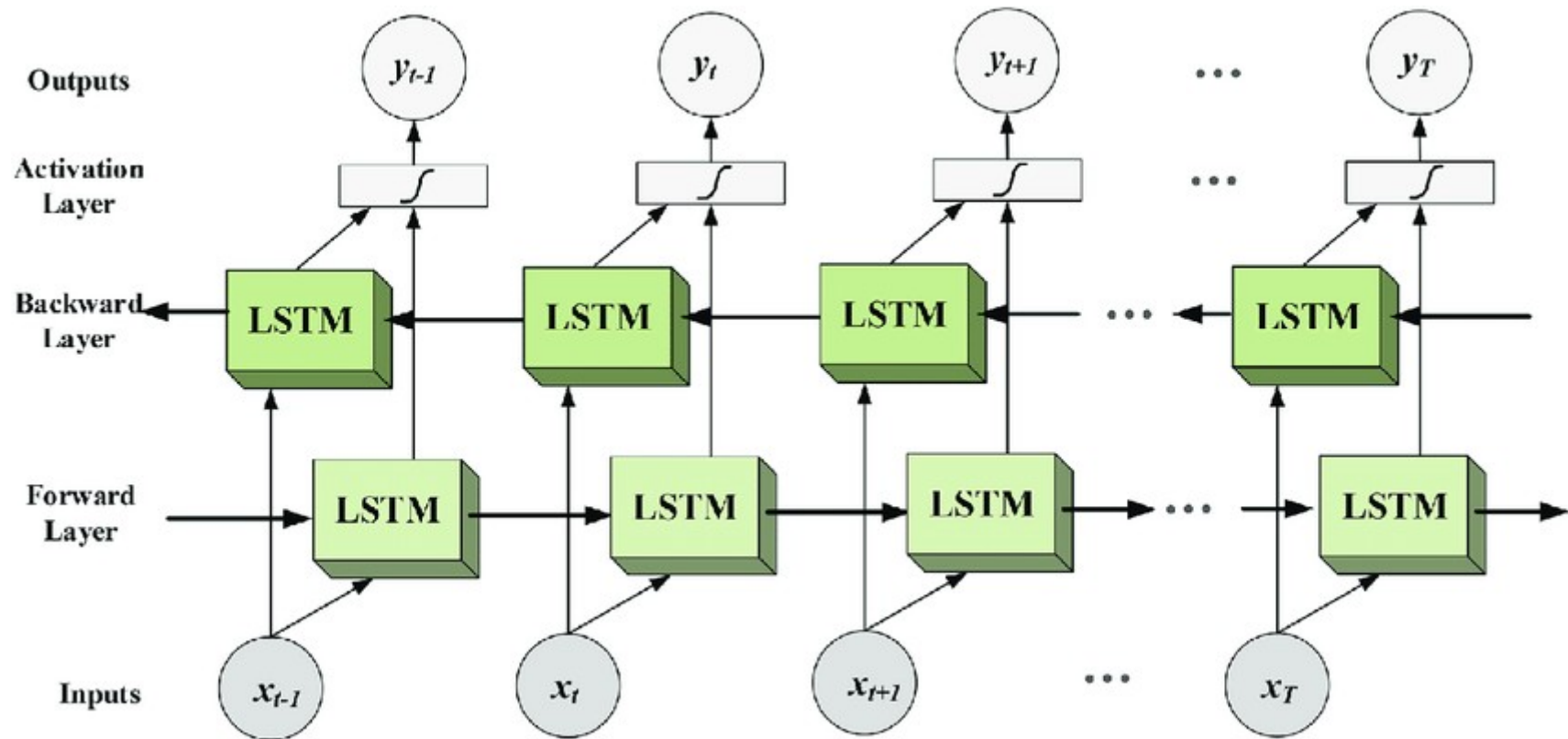
Deep learning = hierarchical representation learning

Why do we add more and more layers? Because when done right, a “deep” network can learn more and more “high level features” of the input data:



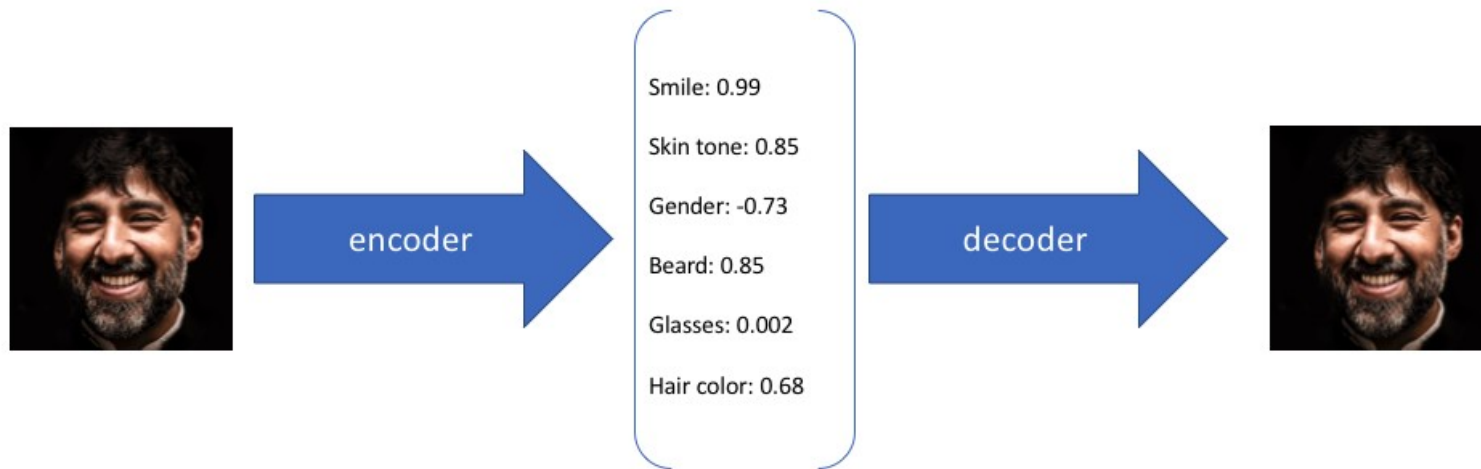
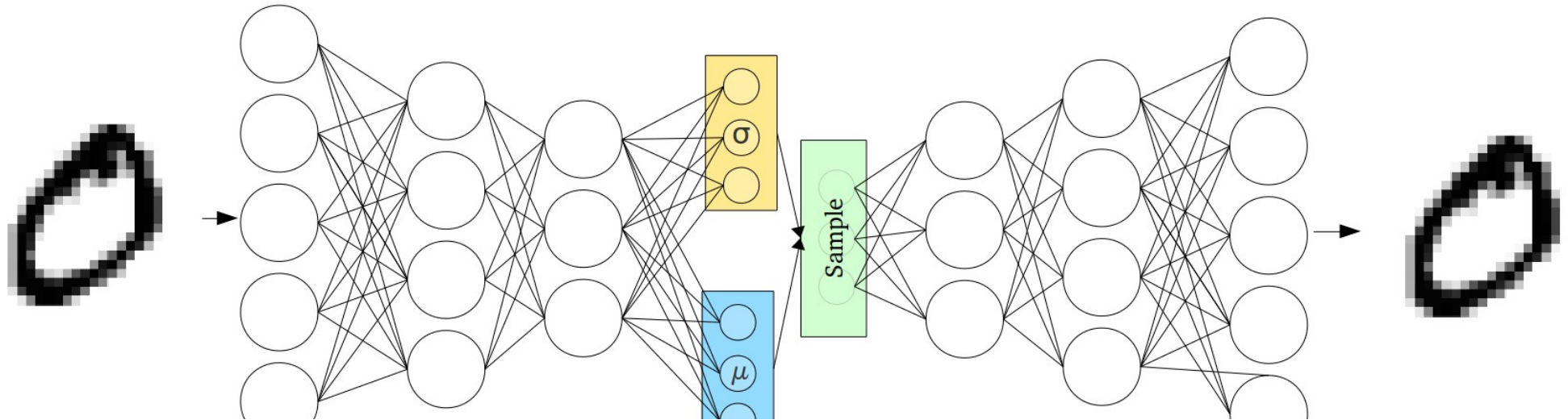
A layer, evaluated for data = a “representation” of the data

... and often we add even more complicated layers ...



(and by the way, the LSTM was invented by a Bavarian who is now a prof in Linz)

.... and construct networks that generate fake data by “inverting the problem” (“create fake data x that will be labelled as y ”)



Latent attributes
“understanding is compression” (these networks find low-dimensional representations of high-dimensional data)

.... and networks that compete against other networks ...

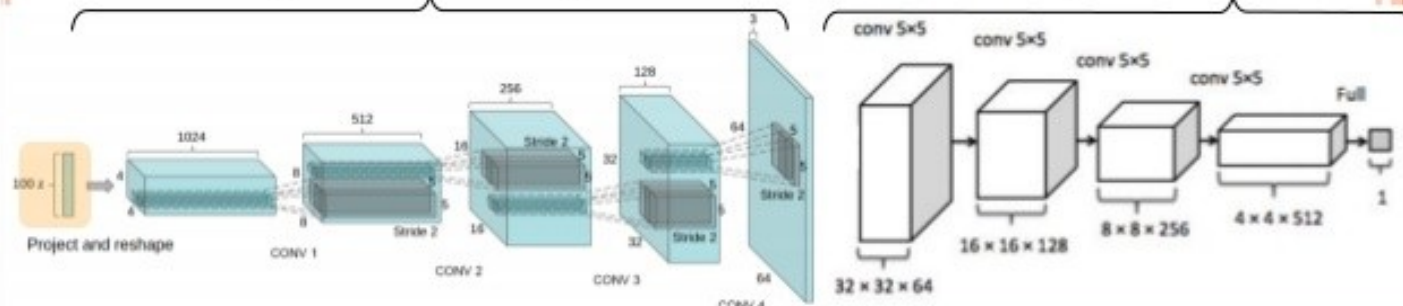
Training data B



Example Architecture:

Generator $G(\cdot)$

Discriminator $D(\cdot)$



As one of you guys said: "if you cannot make it, you do not understand it"

<https://www.slideshare.net/shyamkkhadka/unsupervised-learning-representation-with-dcgan>

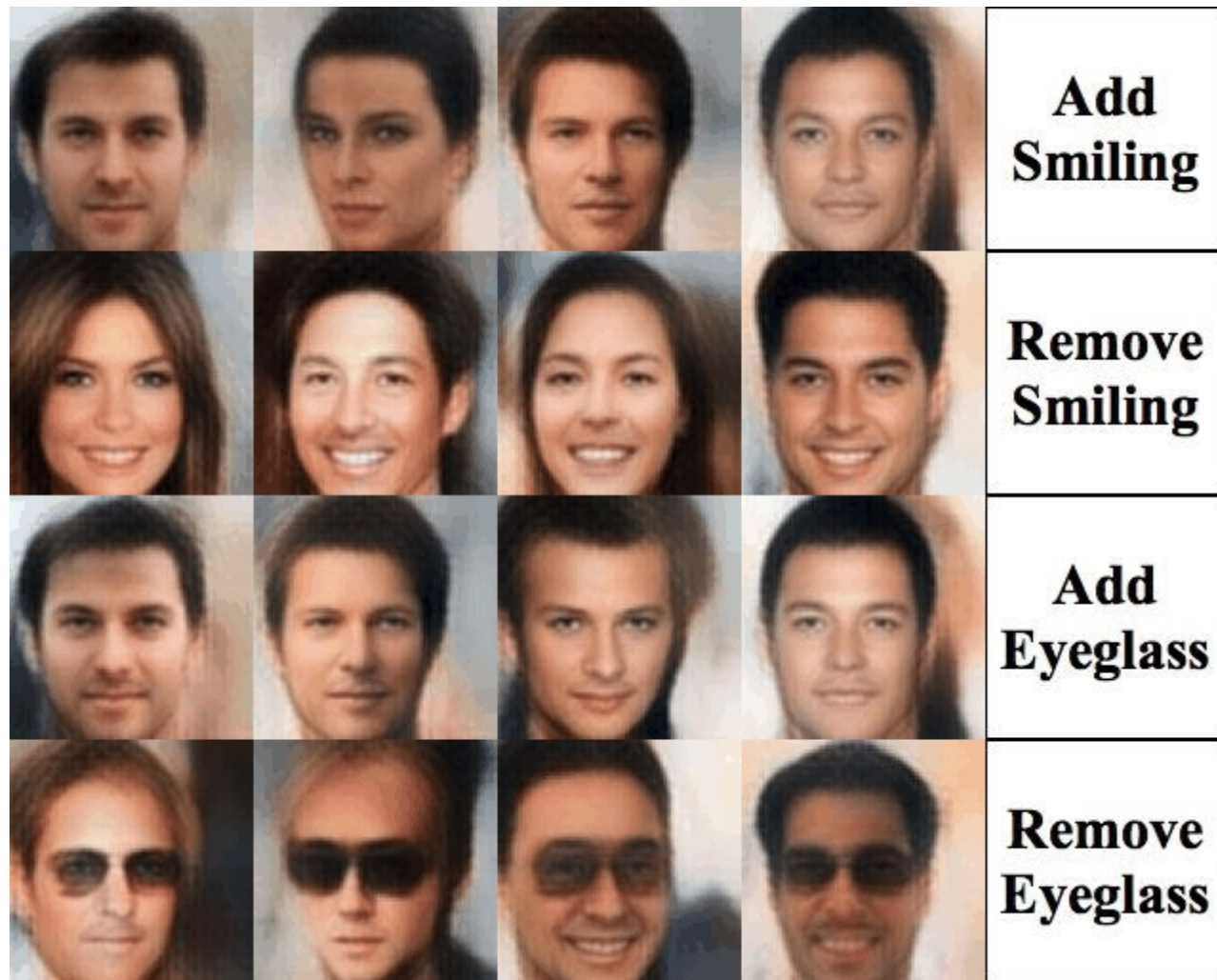
But, to repeat, the **mathematical ingredients** of all these neural networks are:

- data are referred to as **features** and called “**x**”, **labels** we call “**y**”, the problem is to **find the best function $y=f(x;w)$** . w are the parameters of the functions. (sorry for the missing vector symbols)
- we do this by constructing another “**loss**” function **L** that tells us how badly we are doing, based on the true “**y**”.
- we find those values of w , that minimize the function **L**.
- we do this via “**gradient descent**”, that is we take a step in the direction of the “**fastest improvement**”, in this abstract mathematical space.
- we **interpret** the w ’s **as the “weights” of a neural network**, i.e. as a measure for how easily a certain neuron fires.
- we understand “**learning**” **as updating weights** in a neural network. Gradient descent tells us how exactly we update the weights.
- we make our neural networks more and more complicated, and “train” it with more and more data.

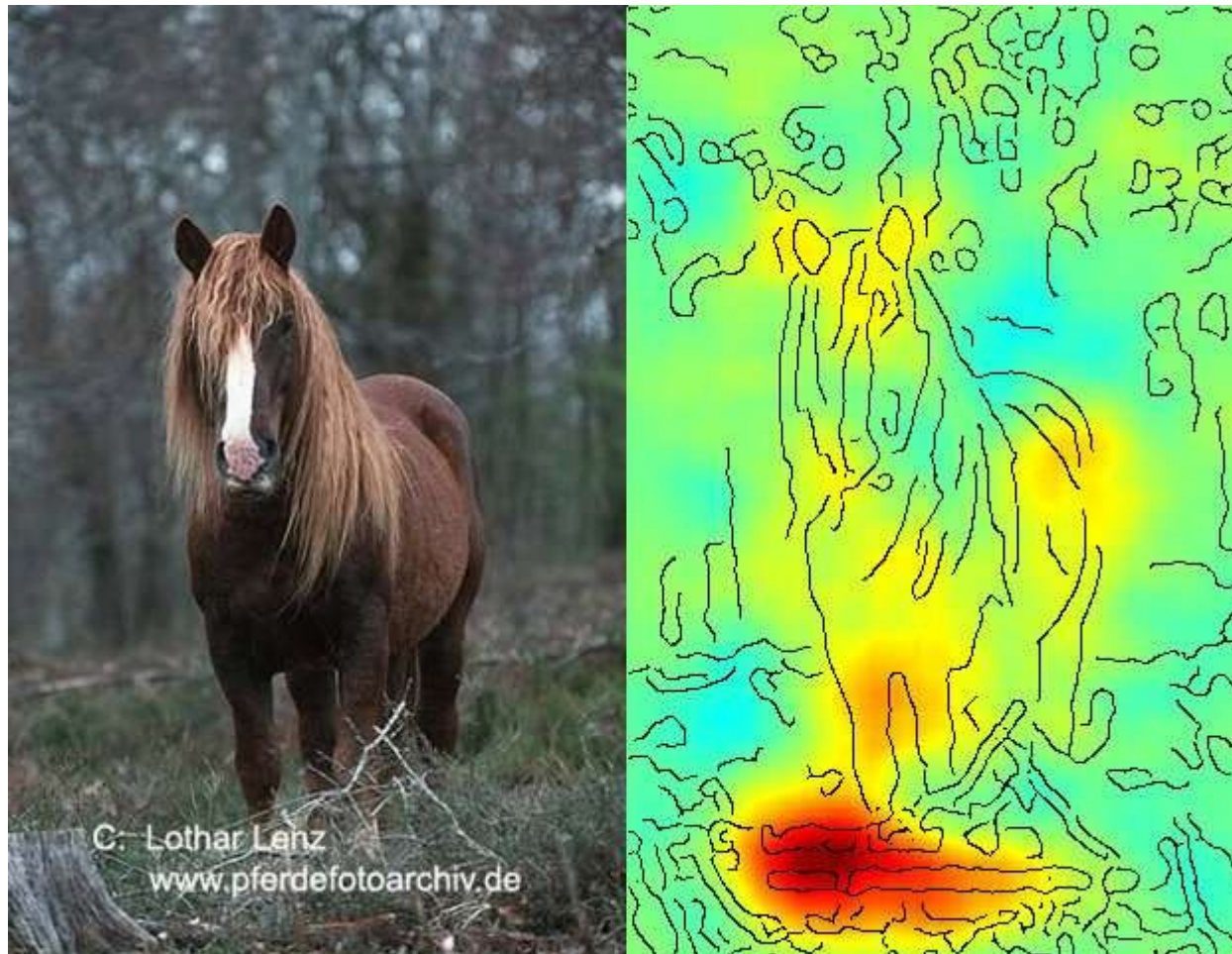
And amazingly enough, such fairly simple math can learn abstract things like “styles”



... or “smiles”



But do keep in mind: all these networks do is exploit correlations in the input data!

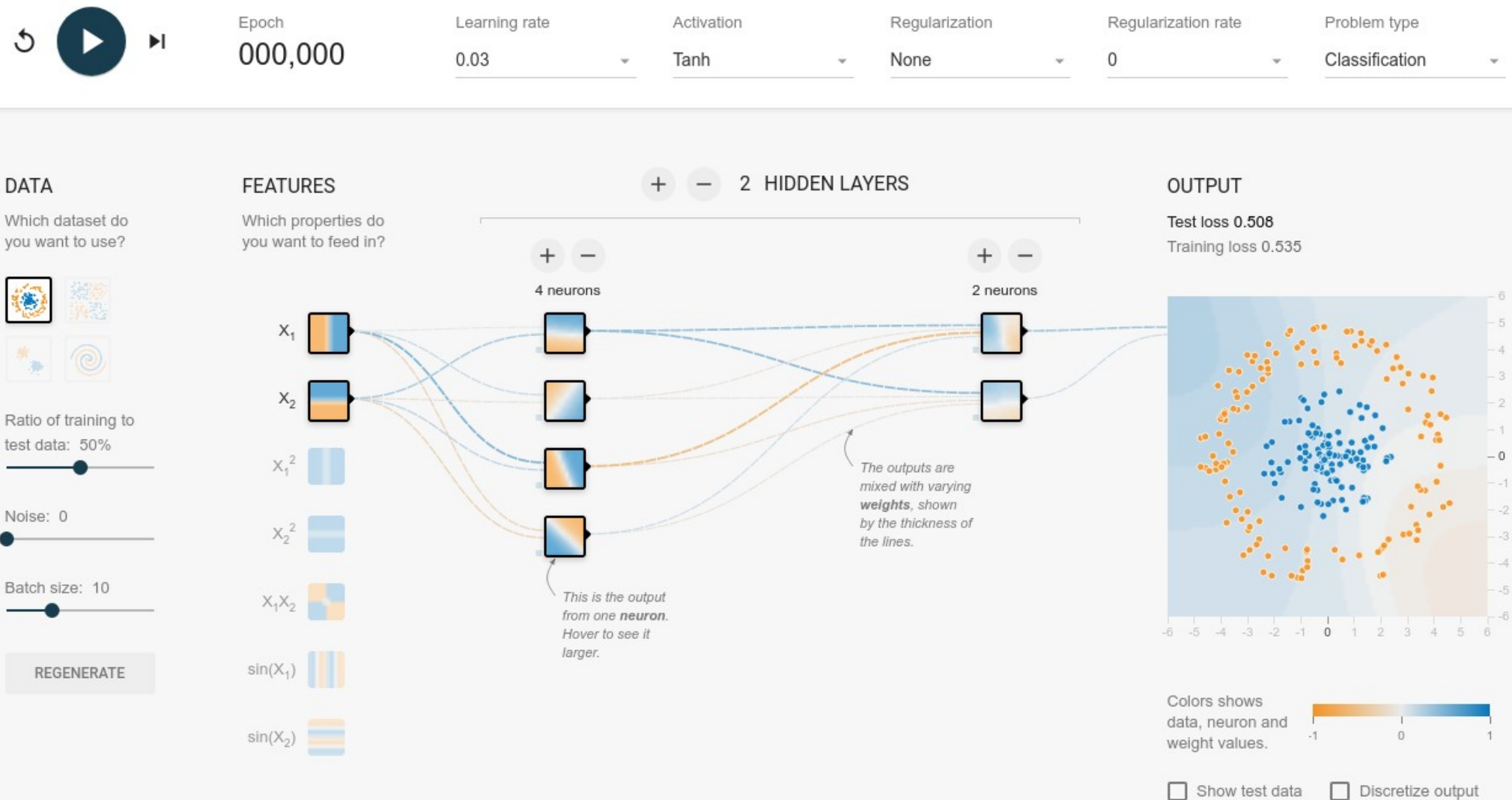


How did it recognize the presence of a horse? By the figure caption, not by the horse!

<http://playground.tensorflow.org>

That was some wild abstract math for you? Luckily, we can simply play with all the math.

Now play, guys! (And I will do some more explaining while we play)



Useful knowledge for tensorflow playground

Loss functions

- The loss function that I introduced before takes the (mean) absolute value of the element-wise difference between prediction $p(x)$ and target y : **L1 Loss**.
- Many other sensible choices for a loss function:

L2 Loss: (mean) squared error between prediction and target.

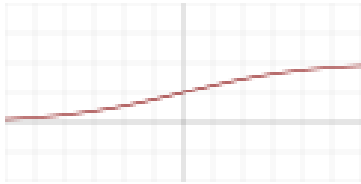
$$L(\vec{w}) = \sum_{i:\text{points}} p_{(\text{red};\text{blue})}^2(\vec{x}_i)$$

(Binary) Cross Entropy: the expected negative log likelihood of misclassification

$$L(\vec{w}) = - \sum_{i:\text{points}} \ln(1 - p_{(\text{red};\text{blue})})(\vec{x}_i)$$

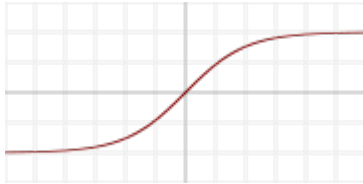
Activation functions

- We have a lot of freedom in the choice of our activation functions

- sigmoid (logistic)  $f(x) = \frac{1}{1 + e^{-x}}$

pro: non-linear, differentiable, confined range [0,1]

con: zero at large values of $|x|$ (vanishing gradient problem)

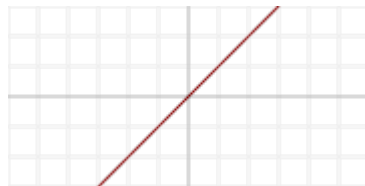
- tanh  $f(x) = \tanh(x)$

scaled version of sigmoid: $\tanh(x) = 2 \sigma(2x) - 1$

Activation functions

- We have a lot of freedom in the choice of our activation functions

- Linear



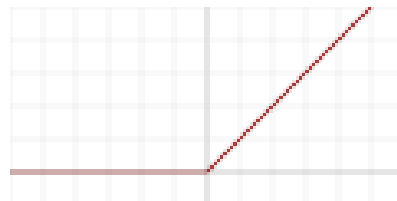
$$f(x) = x$$

Pro: simple, non-zero derivative

Con: constant gradient, cannot introduce non-linearity

- ReLU

rectified linear unit



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

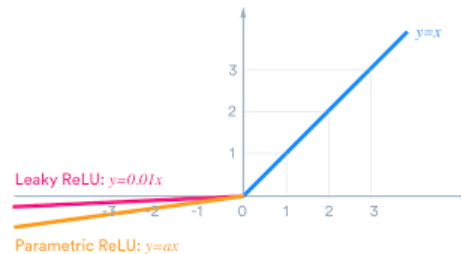
Pro: simple, computationally fast, non-linear, non-zero gradient for any large x

Con: not bound, neurons may “die”, get stuck at zero.

Activation functions

- We have a lot of freedom in the choice of our activation functions

- Leaky ReLU

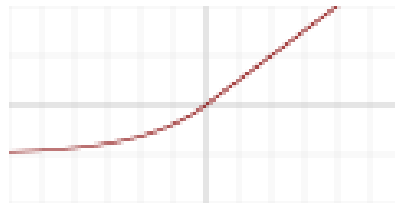


$$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

Advantages of ReLU, plus it doesn't "die" for $x < 0$.

- SeLU

scaled
exponential
linear units



$$f(x) = \lambda \begin{cases} \alpha e^x - \alpha & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

Used in “self-normalizing neural networks”, where the output of the neurons in each layer is distributed with zero mean and unity variance if the input is also distributed with zero mean and unity variance

Google colab

<https://colab.research.google.com/>

If the tensorflow playground is too childish for you, I prepared a very simple pytorch example:

<https://github.com/WolfgangWaltenberger/studienstiftung>