Fully connected networks

Problem setup

- We are back to the problem setup of supervised learning
  - n number of features
  - Input data $\boldsymbol{x}^{(i)} = \left[x_1^{(i)}, x_2^{(i)}, \ldots x_n^{(i)}\right]$
    - The features of the i-th training example
  - Output data y (a scalar)

Hypothesis function

- For $\boldsymbol{x} = [x_1, x_2, \dots x_n]$

$$f(\boldsymbol{x}, \boldsymbol{\theta}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 \dots \theta_n x_n$$

- For the convenience of notation, we can say $x_0 = 1$

$$\hat{y} = f(\boldsymbol{x}, \boldsymbol{\theta}) = \boldsymbol{\theta}^T \boldsymbol{x}$$

- But wait... isn't this linear regression? In fact, this is **exactly** linear regression

Multiple outputs

- Let us try something else.
- We will have a vector $\boldsymbol{y}$ of size m as output
- Instead of a **vector** $\boldsymbol{\theta}$, we consider a **matrix** $\boldsymbol{W} = \{w_{ij}\}$, of size $n \times m$

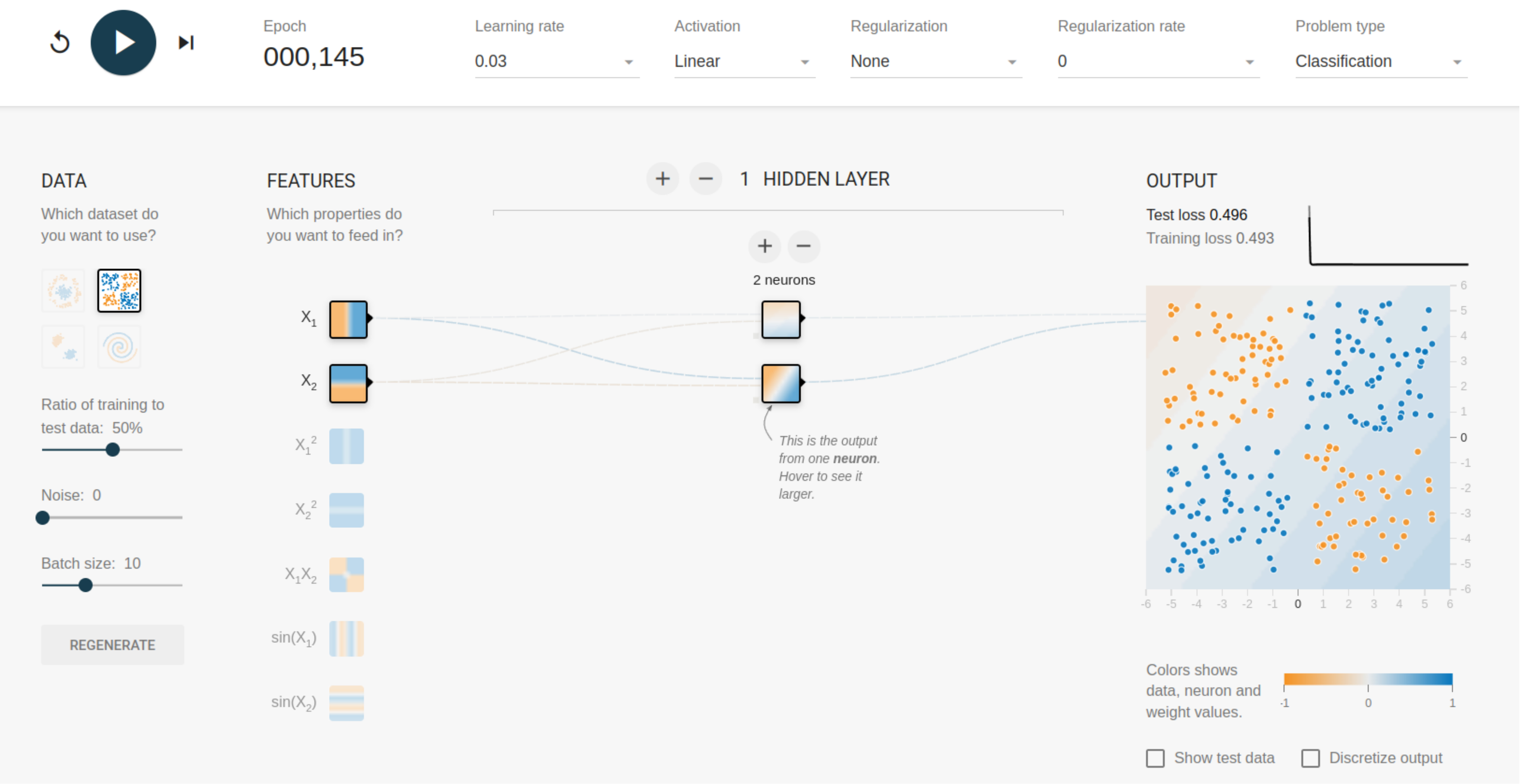
$$\hat{\boldsymbol{y}} = f(\boldsymbol{x}, \boldsymbol{\theta}) = \boldsymbol{W}\boldsymbol{x}$$

or, written out:

$$\hat{y}_j = \sum_i w_{ij} x_i$$

- But, wait: isn't this **still** just linear regression (in fact m linear regressions packaged together)?

One layer network with two outputs



Multiple layers

- What about multiple layers?
- We can have multiple layers with matrices $\boldsymbol{W}^{(1)}$, $\boldsymbol{W}^{(2)}$...$\boldsymbol{W}^{(k)}$
- We have some **hidden layers** $\boldsymbol{z}^c$ with $c = 0 \ldots k$
  - Of some size n^c
  - $\boldsymbol{z}^{(0)}$ is the input $\boldsymbol{x}$
  - $\boldsymbol{z}^{(k)}$ is the output $\boldsymbol{y}$

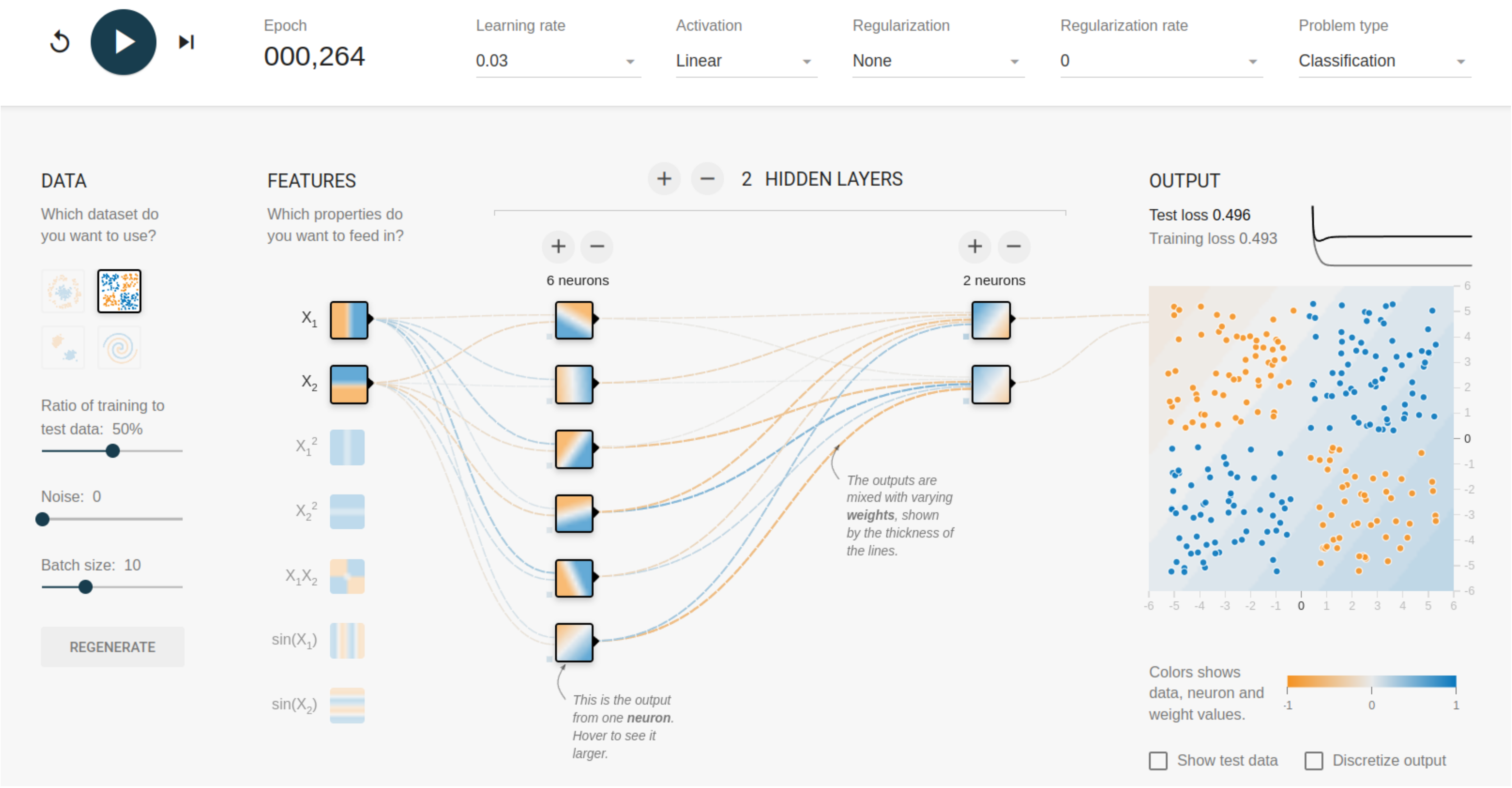
$$\boldsymbol{z}^{(c+1)} = \boldsymbol{W}^{(c)} \boldsymbol{z}^{(c)}$$

so

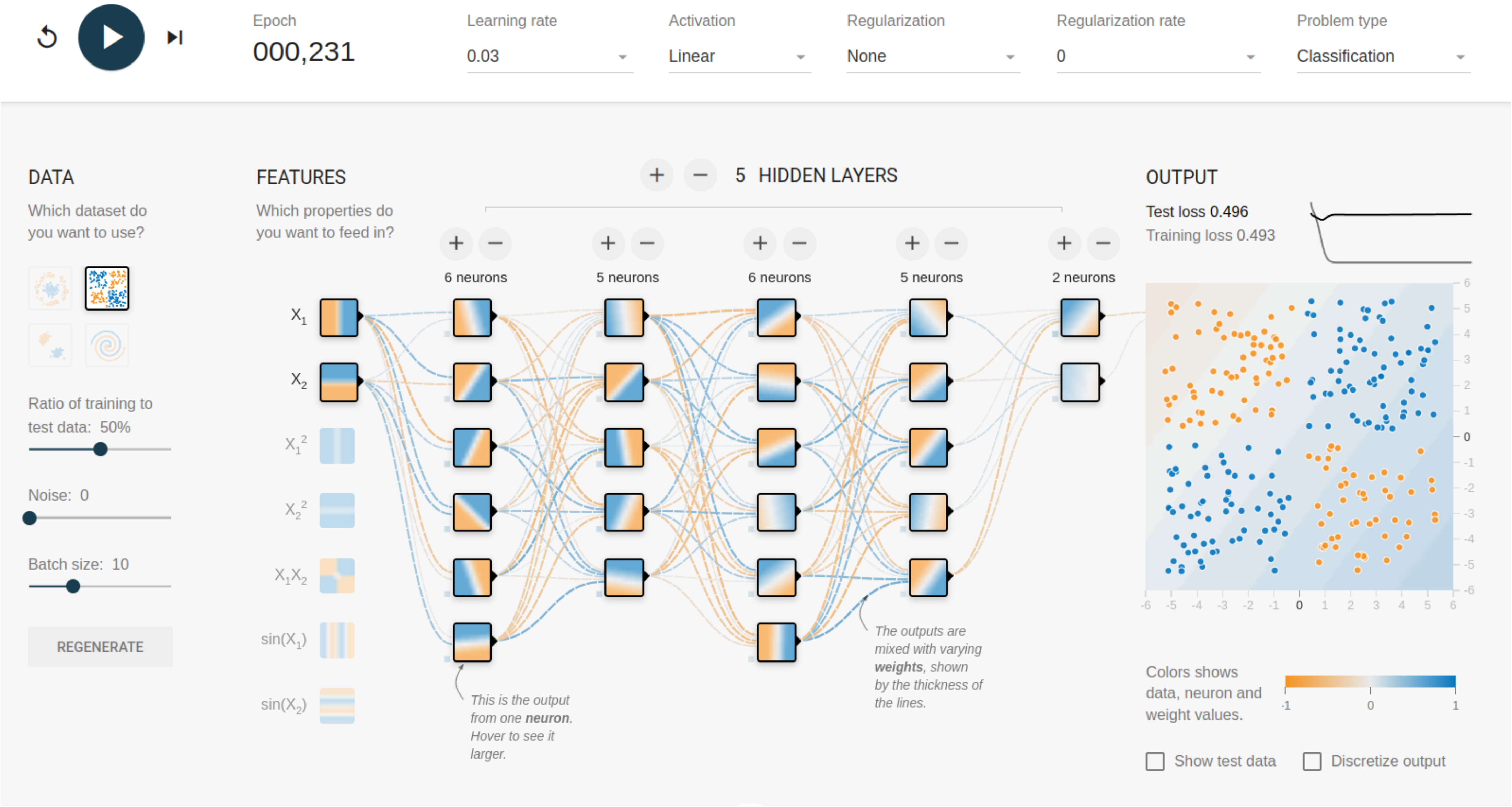
$$\hat{\boldsymbol{y}} = \boldsymbol{W}^{(k)} \boldsymbol{W}^{(k-1)} \ldots \boldsymbol{W}^{(0)} \boldsymbol{x}$$

- We can designate $\boldsymbol{\theta} = \{\boldsymbol{W}^{(1)}$, $\boldsymbol{W}^{(2)}$...$\boldsymbol{W}^{(k)}\}$

One hidden layer linear network



Four hidden layers linear network



But, wait...

- Can't we just multiply together the matrices $\boldsymbol{W}$?

$$\boldsymbol{W} = \boldsymbol{W}^{(k)}\boldsymbol{W}^{(k-1)}\dots\boldsymbol{W}^{(0)}$$

- Then we can just write

$$\hat{\boldsymbol{y}} = f(\boldsymbol{x}, \boldsymbol{\theta}) = \boldsymbol{W}\boldsymbol{x}$$

- So this is **still** just linear regression.
  - We did not gain **anything** in expressivity
  - Cannot solve problems that are not linear, and of course, now we have a large number of totally superflous parameters in $\boldsymbol{\theta}$
- Basically, this is what Minsky based his attack on perceptrons

Nonlinearity

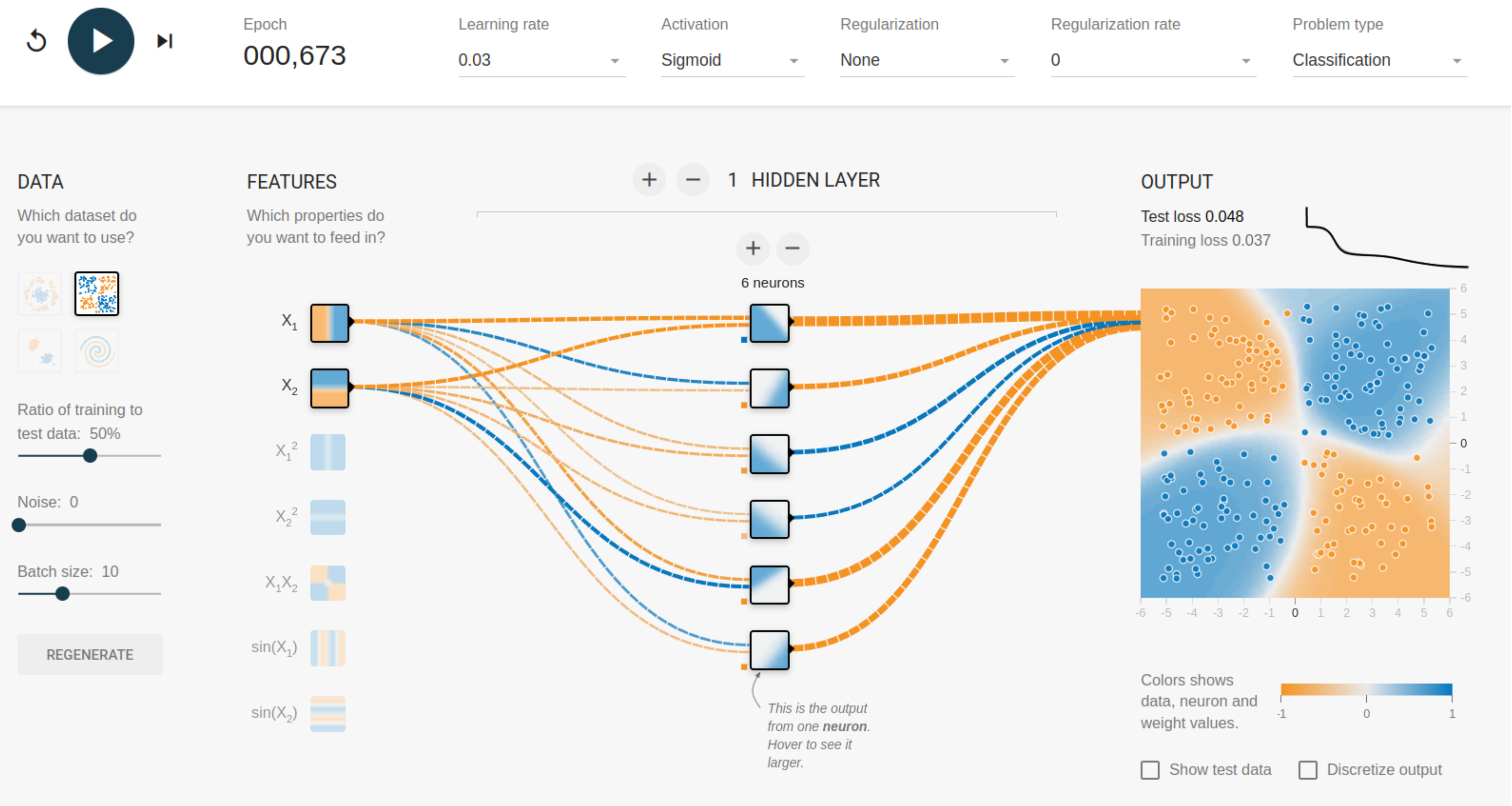
- What about we introduce a **nonlinear** function $g(\cdot)$ and we say:

$$\boldsymbol{z}^{(c+1)} = g\left(\boldsymbol{W}^{(c)}\boldsymbol{z}^{(c)}\right)$$

- This means that we apply g individually to each element of the vector.
- We cannot multiply through any more.

$$\hat{\boldsymbol{y}} = g^{(k)}\left(\boldsymbol{W}^{(k)} \cdot g^{(k-1)}\left(\boldsymbol{W}^{(k-1)} \dots g^{(0)}\left(\boldsymbol{W}^{(0)}\boldsymbol{x}\right)\dots\right)\right)$$

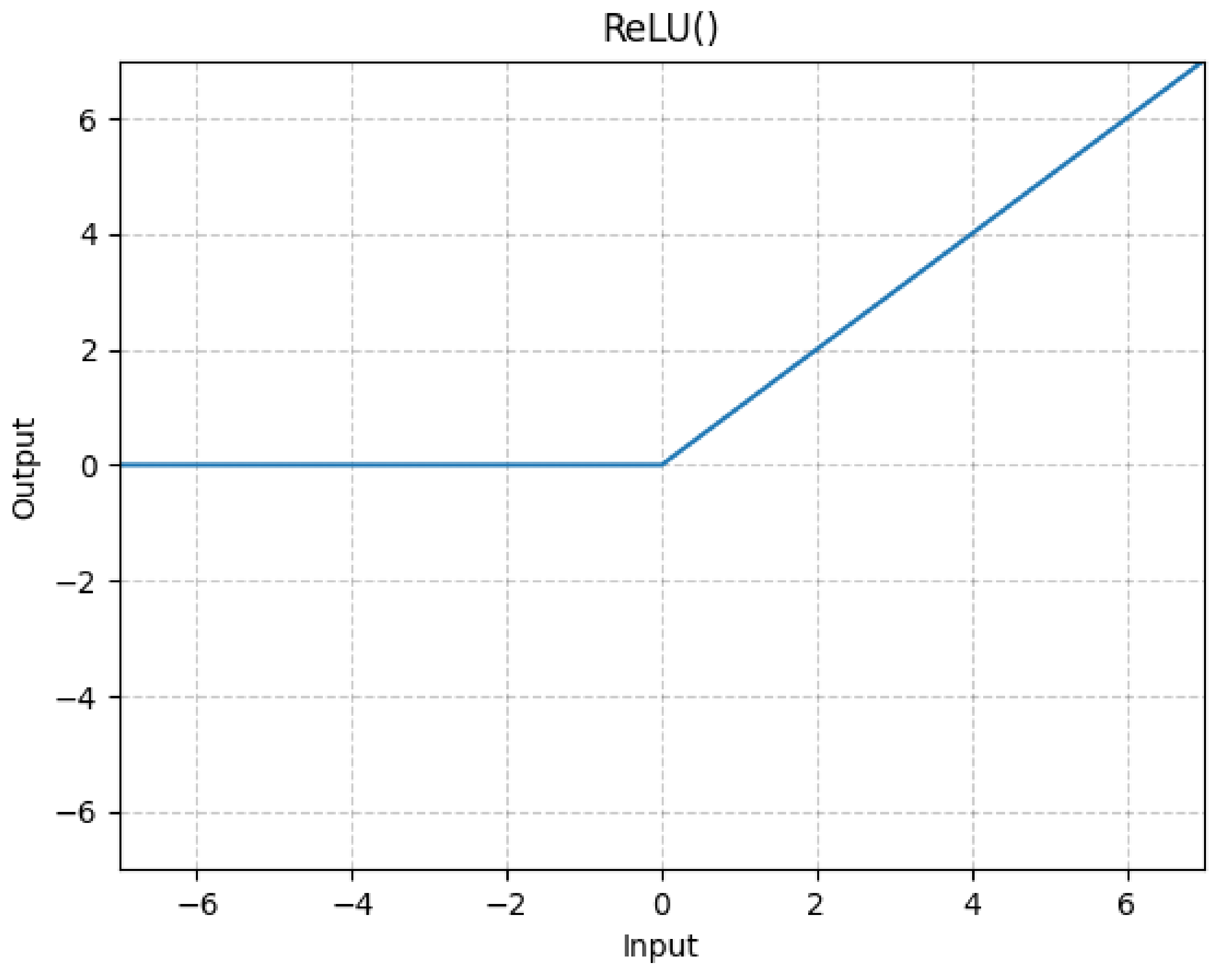
One hidden layer with sigmoid nonlinearity



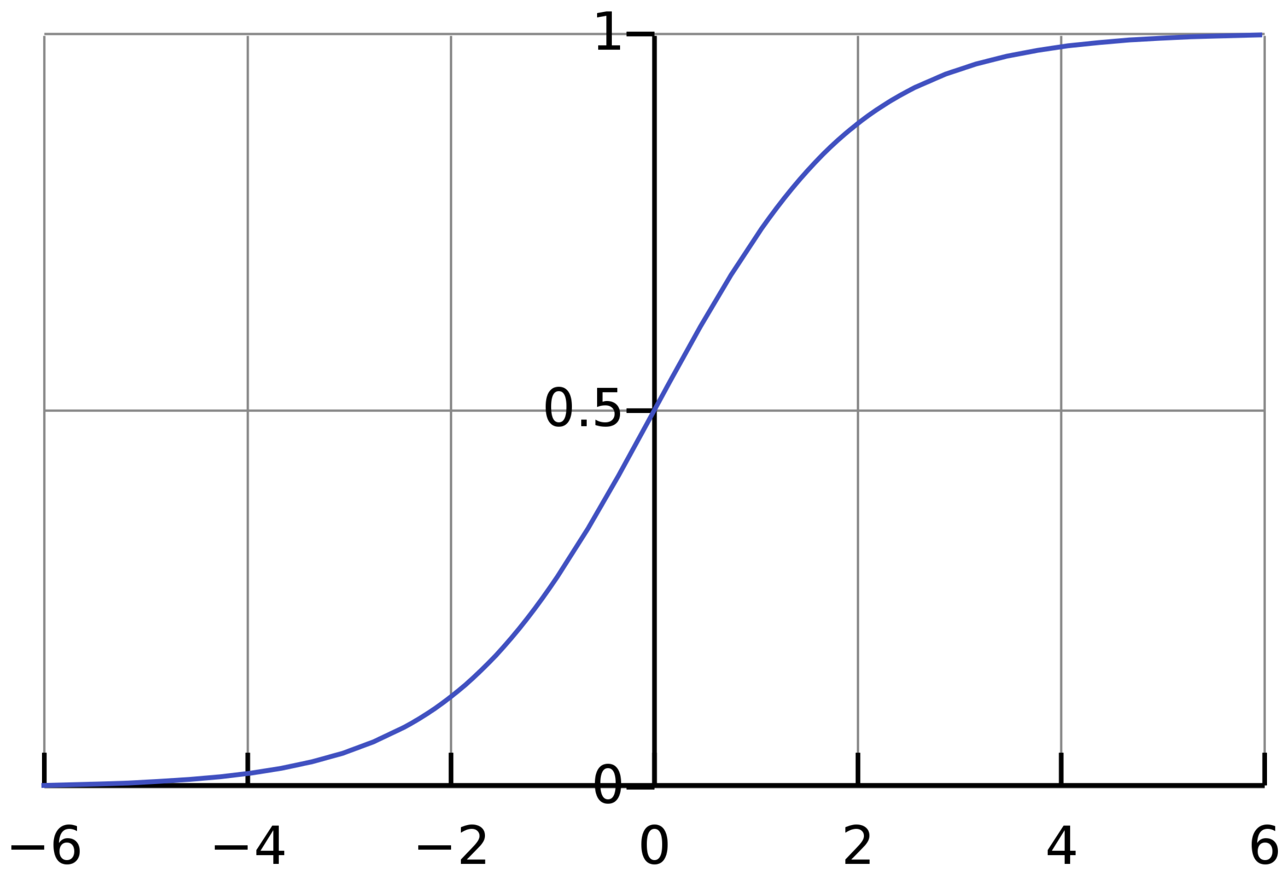
Fully connnected network, multi-layer perceptron

- This is called a **fully connected** neural network (as every node is connected to every node in the next layer and the previous layer, if they exist)
- It is also called a **multi-layer perceptron** or MLP

Nonlinearity: ReLU



Nonlinearity: sigmoid



Did we gain anything in expressivity?

- Yes!!!
- A series of theorems called **universal approximation theorems** show that this model can approximate **arbitrary** functions to **arbitrary** precision with only **one hidden layer** and very mild requirements for the nonlinear function g

  But it needs an infinitely large hidden layer to do that...

- Pretty much all the nonlinearities we discussed before work

How do we train a system like this?

- The most traditional way is to separate the trainable parts into two components:
  - **Architecture:** number of layers, size of each layer, the nonlinearity applied to it
    - This is typically **not** trained, but **engineered**. We choose them based on our experience and/or intuition.
    - We can see these as **hyperparameters**
  - **Parameters**: we designate $\boldsymbol{\theta} = \{\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)} ... \boldsymbol{W}^{(k)}\}$
    - This only make sense once the architecture is fixed.
    - We train this using stochastic gradient descent (or variants) just like we did for linear regression.

How do we train a system like this?

- Remember we have input and output data pairs: $\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}$
- $\hat{\boldsymbol{y}} = f(\boldsymbol{x}, \boldsymbol{\theta})$
- We design a loss function which looks roughly like this:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} dist(\hat{\boldsymbol{y}}, \boldsymbol{y})$$

- Of course, we need to choose the distance function (which might not be quite-quite a distance function)
- Also we might add some extra terms (regularization etc.)
- We find the best $\boldsymbol{\theta}$ that minimizes the loss:

$$\boldsymbol{\theta}^* = argmin\, \mathcal{L}(\boldsymbol{\theta})$$

Number of parameters

- How many parameters we have: $\boldsymbol{\theta} = \{\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)} ... \boldsymbol{W}^{(k)}\}$
  - First matrix: size of input $\times$ size of first hidden layer, +
  - Second matrix: size of first hidden layer $\times$ size of second hidden layer, +
  - $\ldots$ +
  - size of last hidden layer $\times$ size of output
- If the input dimensions are high, and number of layers is high, the size of parameters $\boldsymbol{\theta}$ will be high.

Taking the gradient

- We have to take $\nabla \mathcal{L} = \left[\ldots \frac{\partial \mathcal{L}}{\partial \theta_i} \ldots \right]$
- **Problem** $\mathcal{L}$ might not be differentiable
  - For instance, ReLU is not differentiable at 0.
  - Just wing it. Eg. set it to 0.5.
- **Problem** it doesn't sound fun to differentiate a function with millions of parameters.
  - Solution: automatic differentiation
  - Frameworks such as pytorch, tensorflow etc. implement this for you
- **Problem** intermediate tables in the automatic differentiation can be huge
  - Solution: efficient way of calculating the partial derivatives, going **backward** and using the chain rule ("backpropagation")

Global and unique solution

- For linear regression, with least squares, the loss surface is convex
  - So we have a unique solution, and gradient descent leads us there.
- For a fully connected neural network the loss function is nowhere convex
  - We have many optimal solutions!
  - For instance, if we have a hidden layer with 1000 nodes, we will have $1000! \approx 4 \cdot 10^{2567}$ equivalent minimum loss points!
  - And likely many other local minima as well.
- Gradient descent appears hopeless
  - Yet, it is actually working quite well in practice!
  - No theoretical guarantees of finding an optimum

Exercise

- Fully connected layers
  https://playground.tensorflow.org/
- Exercise 1: Classify linearly separatable data sets with a linear regressor
- Exercise 2: Classify linearly separatable data sets with multi-layer linear regressor
- Exercise 3: Classify non-linearly separatable data sets with multi-layer linear regressor.