INTRODUCTION TO NEURAL NETWORKS

Perceptron

Weights and Bias

Activation Function

Training a neural network

Loss function

Stochastic gradient descent

EMBEDDED MACHINE LEARNING
INTRODUCTION TO NEURAL NETWORKS

An artificial neural network is a algorithmic construct, which takes an input and creates an output. The parameters of the neural network are *trained* using labeled input. This is called supervised learning. The algorithmic idea is based on biological neural networks.

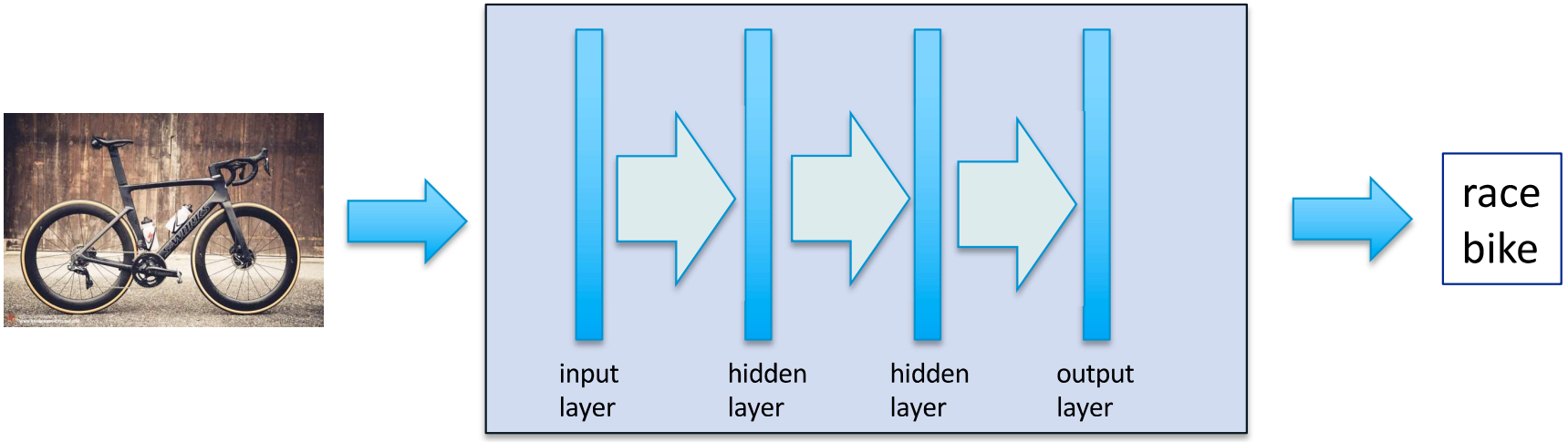
```
if racebike:
        print(”race bike”)
else:
        print(”something else”)
```

race bike

Prof. Dr. Stephan Pareigis

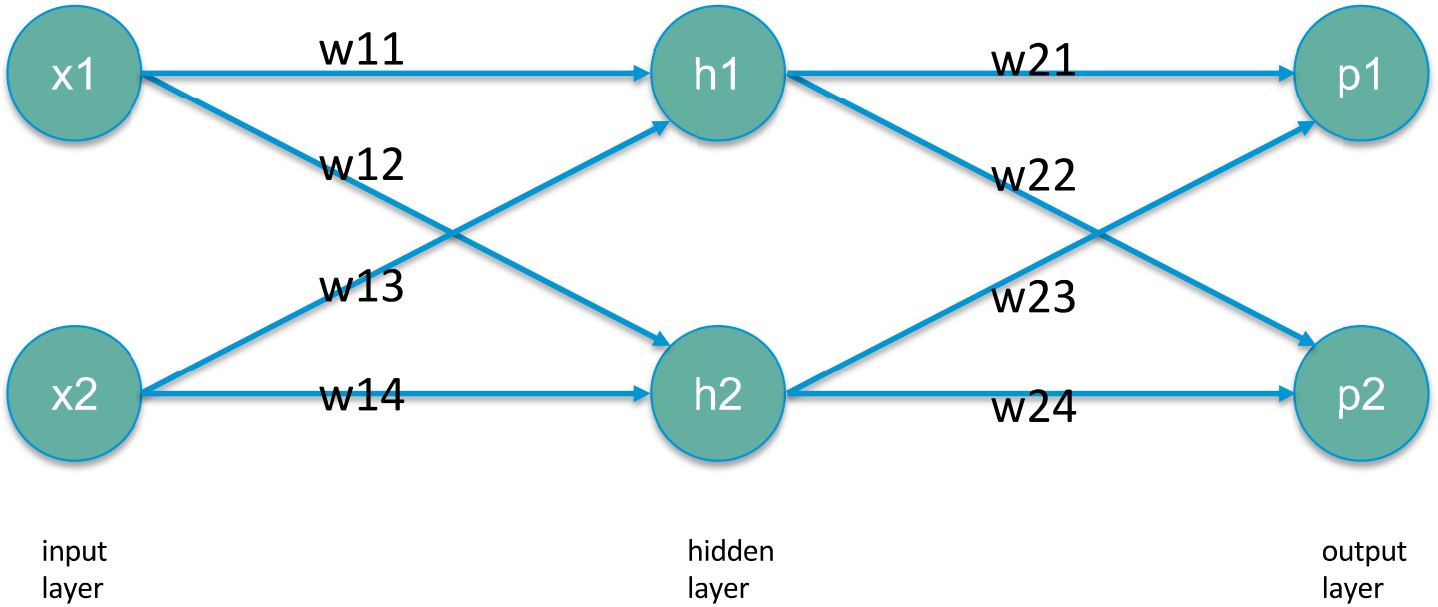
EMBEDDED MACHINE LEARNING
INTRODUCTION TO NEURAL NETWORKS

An artificial neural network consists of an input layer, hidden layers and an output layer. The input layer receives the input image or input signal. Hidden layers store the model in a parameterized way. The output layer produces the output.

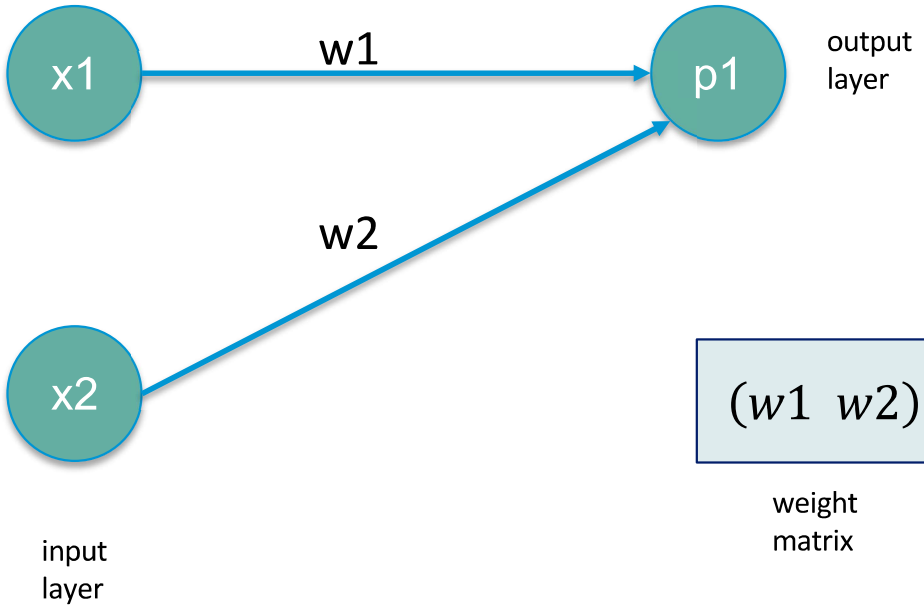


EMBEDDED MACHINE LEARNING
INTRODUCTION TO NEURAL NETWORKS

Input values x1 and x2 are simply numbers. They are multiplied with weights w11, ... w14 to produce values in the hidden layer h1 and h2. This is a matrix multiplication. x1 and x2 are pixel values of the input image. The weights w11,..,w14 and w21,..,w24 have to be learned during a training process. The are initialized randomly. An artificial neural network consists of a series of matrix multiplications.



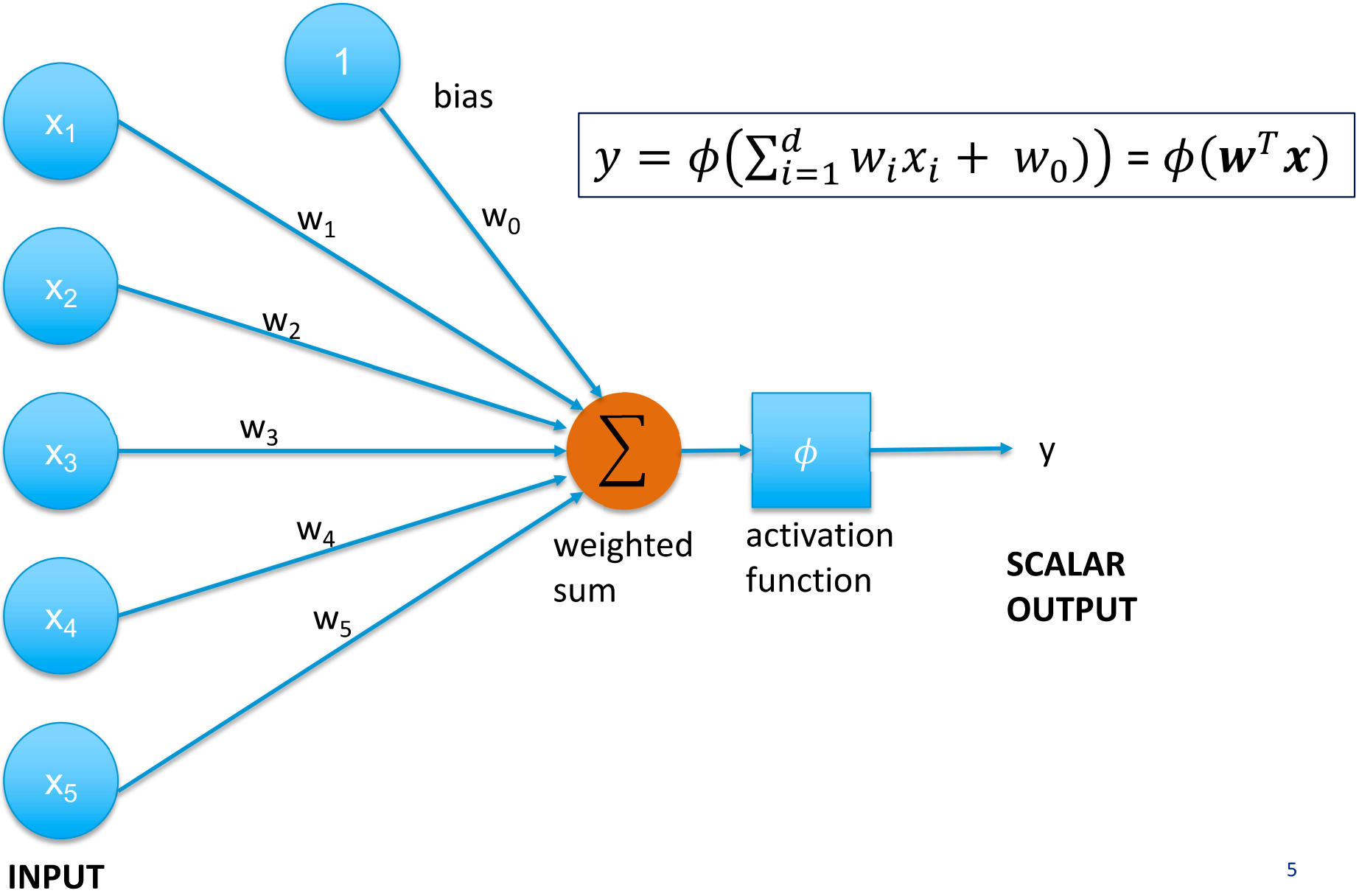
In this very simple example the following can be observed: If the input vector is (0 0), then the output will always be 0. This gives reason to introduce the bias trick.



$$(w1 \;\; w2)\begin{pmatrix} x1 \\ x2 \end{pmatrix} = w1 \cdot x1 + w2 \cdot x2$$

weight matrix | input vector | output value

PERCEPTRON



$$y = \phi\left(\sum_{i=1}^{d} w_i x_i + w_0\right) = \phi(\boldsymbol{w}^T \boldsymbol{x})$$

PERCEPTRON

A perceptron is defined by the following formula

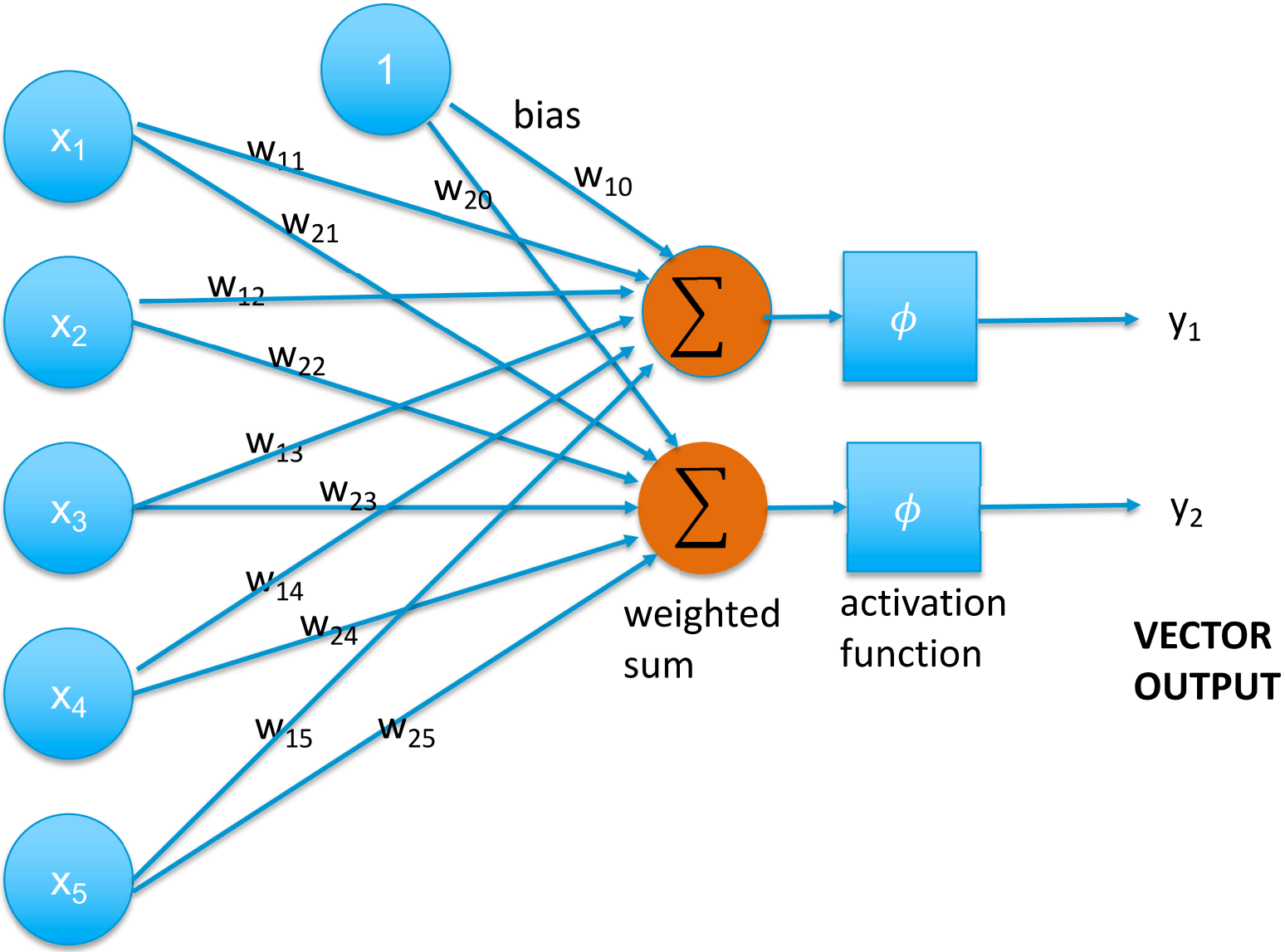
$$y = \phi\left(\sum_{i=1}^{d} w_i x_i + w_0\right) = \phi(\boldsymbol{w}^T \boldsymbol{x})$$

where

$$\boldsymbol{w}^T := (w_0 w_1 w_2 \ldots w_d) \qquad \boldsymbol{x} := \begin{pmatrix} \mathbf{1} \\ x_1 \\ x_2 \\ \ldots \\ x_d \end{pmatrix},$$

and $\boldsymbol{w}^T \boldsymbol{x}$ denotes the scalar product between the weight vector **w** and the input vector **x**, both being vectors in vectorspace $\mathbb{R}^{d+1}$.
w_0 is called the bias, w_i for i=1,...,d are called weights.

PERCEPTRON


1
bias
x_1
x_2
x_3
x_4
x_5
w_{11}
w_{21}
w_{12}
w_{22}
w_{13}
w_{23}
w_{14}
w_{24}
w_{15}
w_{25}
w_{20}
w_{10}
$\sum$
$\sum$
ϕ
ϕ
y_1
y_2
weighted sum
activation function
VECTOR OUTPUT
INPUT

PERCEPTRON

If there are multiple outputs $y_1 \ldots y_k$, then **w** becomes a $d \times k$ weight matrix **W**

$$\boldsymbol{y} = \left(\phi\left(w_{j,0} + \sum_{i=1}^{d} w_{j,i} x_i\right)\right)_{j=1,\ldots,k} = \phi(\boldsymbol{W}\boldsymbol{x})$$

where

$$\boldsymbol{W} := \begin{pmatrix} w_{10} w_{11} \ldots w_{1d} \\ \ldots \\ w_{k0} w_{k1} \ldots w_{kd} \end{pmatrix} \qquad \boldsymbol{x} := \begin{pmatrix} \mathbf{1} \\ x_1 \\ x_2 \\ \ldots \\ x_d \end{pmatrix},$$

and $\mathbf{W}\boldsymbol{x}$ denotes the matrix-vector multiplication between the weight matrix **W** and the input vector **x**

WEIGHTS AND BIAS

$$\boldsymbol{W} := \begin{pmatrix} w_{10} w_{11} \dots w_{1d} \\ \dots \\ w_{k0} w_{k1} \dots w_{kd} \end{pmatrix}$$ is called the **weight matrix**.

The weights w_{10} ... w_{k0} are called **bias**. Sometimes these are also denoted by the letter **b** for *bias:* $b_1,\dots,b_d$.

The purpose of the bias is, to enable the perceptron to map all zero input values x_1 ,..., $x_d = 0$ to non-zero output values. It can be seen in the formula for the output, that in this case $\sum_{i=1}^{d} w_i x_i = 0$ if there was no bias. This way an affine linear approximation is possible.

EXAMPLE 1
LEARNING A LINEAR FUNCTION

Suppose for now the activation function shall be $\phi(y) = y$, the identity function.

If there is only a scalar input $x \in \mathbb{R}$ and a scalar output $y \in \mathbb{R}$, then the perceptron can represent a linear function

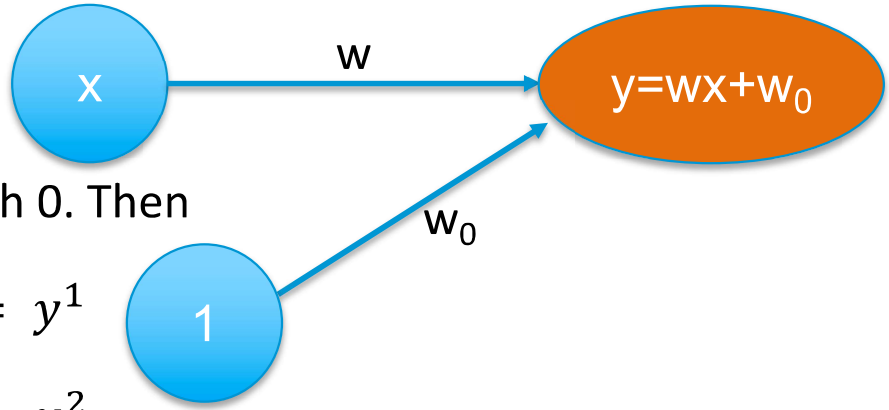
$$y = wx + w_0$$

Given training data $x^1,...,x^n$ with corresponding training output $y^1,...,y^n$, or in short (x^t , y^t), t=1,...,n, then the learning process consists of repeatedly feeding the perceptron with the training data and adjusting the weights w and w_0 , such that a *good* approximation is achived. We will define shortly what is meant by *good.*

EXAMPLE 1
LEARNING A LINEAR FUNCTION

Suppose for now that the training data consists of $(x^1, y^1) = (1,3)$ and $(x^2, y^2) = (2,4)$
We are looking for weights w and w_0 , such that

$$y^t = wx^t + w_0 \quad \text{for } t = 1,2$$



Suppose, w and w_0 have each been initialized with 0. Then

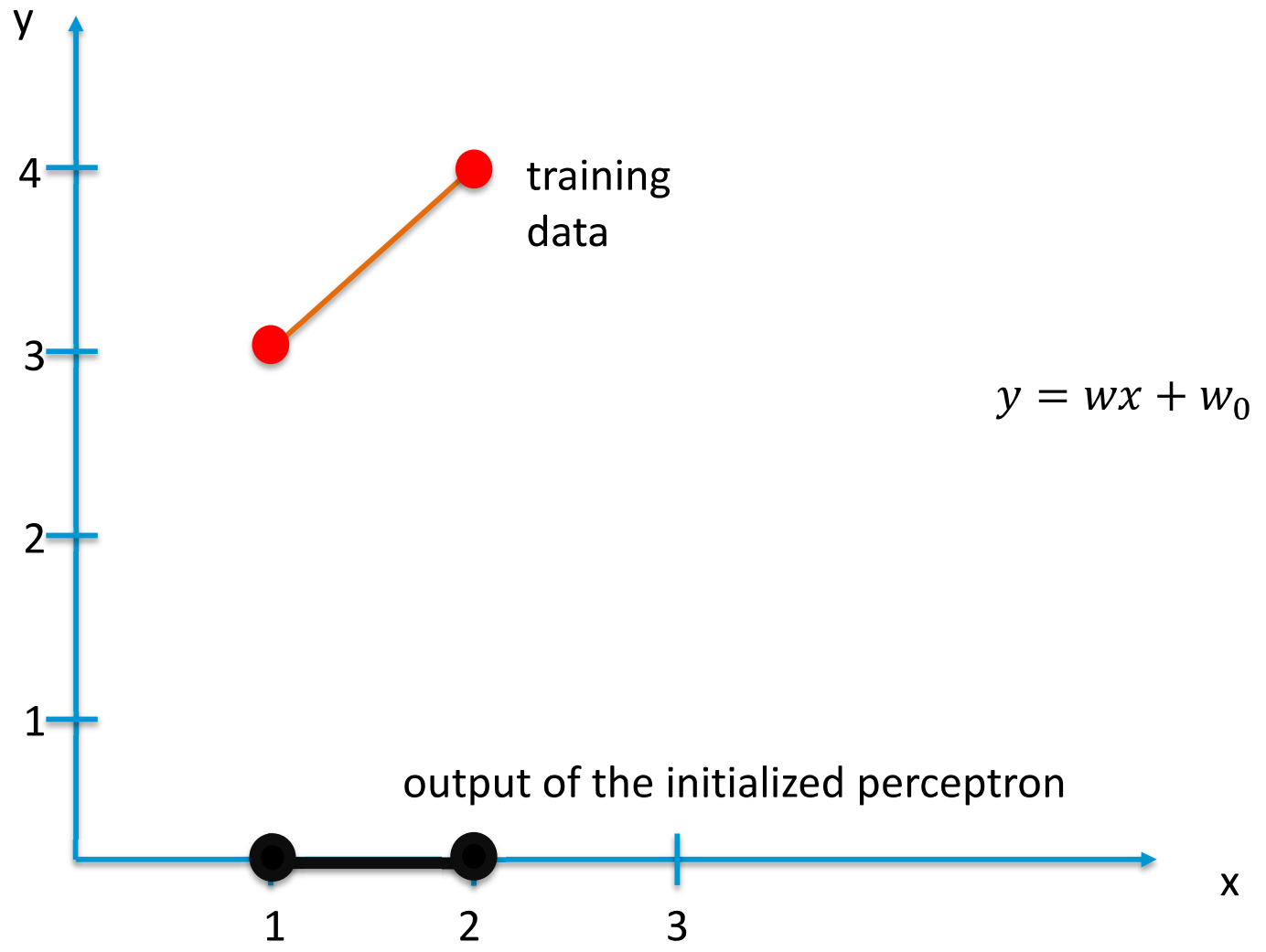
$$wx^1 + w_0 = 0 \cdot 1 + 0 = 0 \neq 3 = y^1$$

and

$$wx^2 + w_0 = 0 \cdot 2 + 0 = 0 \neq 4 = y^2$$

Feeding the x-values of the training data into the percepton leads to y-values 0 and 0, respectively. The perceptron does not approximate the training data very well.

EXAMPLE 1
LEARNING A LINEAR FUNCTION



EXAMPLE 1
LEARNING A LINEAR FUNCTION

We define the following **loss function**

$$\mathcal{L}(\boldsymbol{w}) := \frac{1}{n}\left(\sum_{t=1}^{n}\left(w^T\boldsymbol{x}^t - \boldsymbol{y}^t\right)^2\right)$$

This is called **mean squared error** (MSE). The average is taken over the squares of the differences between the actual output $w^T\boldsymbol{x}^t$ and the expected output $\boldsymbol{y}^t$ over the entire training data.

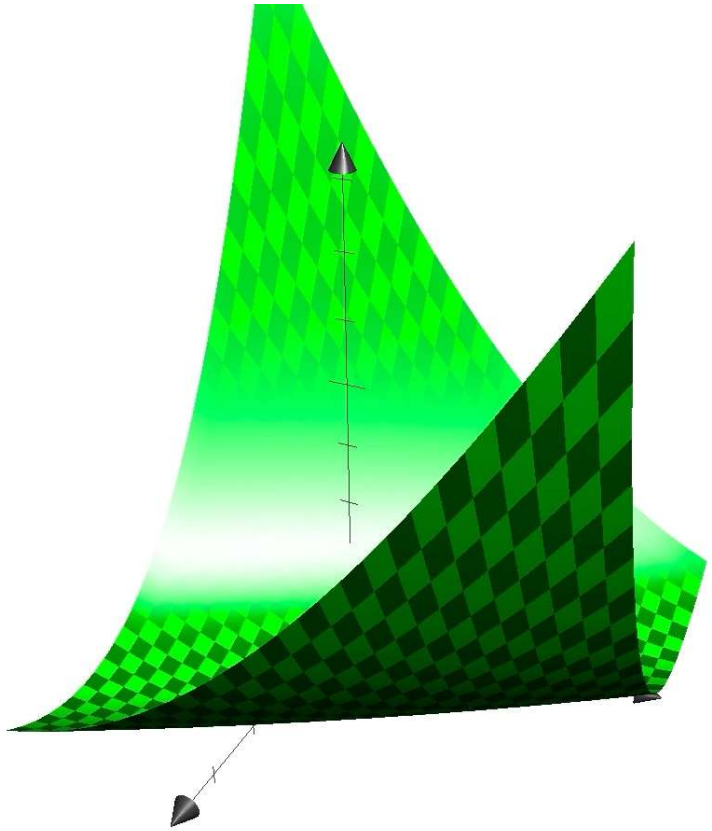
Our example has only 2 training data points, so we get

$$\mathcal{L}(\boldsymbol{w}) := \frac{1}{2}\left(\sum_{t=1}^{2}(wx^t + w_0 - y^t)^2\right) = \frac{1}{2}((0-3)^2 + (0-4)^2) = 12.5$$

EXAMPLE 1
LEARNING A LINEAR FUNCTION

The loss function is a function which depends on the weights, in this case w and w_0, and shows how *wrong* the approximation of the perceptron is. The goal is to find weights, such that the loss function is minimal.

In order to do this, **gradient descent methods** are applied.



EXAMPLE 1
LEARNING A LINEAR FUNCTION

The gradient with respect to the weights w is defined as $\nabla_w \mathcal{L} := \begin{pmatrix} \partial_{w_0} \mathcal{L} \\ \partial_{w_1} \mathcal{L} \\ \partial_{w_2} \mathcal{L} \\ \dots \\ \partial_{w_d} \mathcal{L} \end{pmatrix}$,

the partial derivatives with respect to each weight w_i

In this case we have $\nabla_w \mathcal{L} = \frac{1}{2} \begin{pmatrix} \sum_{t=1}^{2} 2w_0 + 2wx^t - 2y^t \\ \sum_{t=1}^{2} 2w(x^t)^2 - 2x^t y^t + 2w_0 x^t \end{pmatrix}$. Since there are only two weights w and w_0 , this is only a 2-vector. Since the gradient always directs upwards, we have to adjust the weights in the other direction $-\nabla\mathcal{L}$. Taking the values w=0, w_0 = 0 and the given training data, this gradient resolves in

$$\nabla_w \mathcal{L} = \begin{pmatrix} -7 \\ -11 \end{pmatrix}$$

EXAMPLE 1
LEARNING A LINEAR FUNCTION

The weights w and w_0 are now adjusted in the direction - $\nabla_w \mathcal{L}$ using a **learning rate η**. The correction values are denoted by Δw and Δw_0 for the two weights. Suppose we choose a learning rate of $\eta = 0.1$. Then we get

$$\Delta w_0 = -\eta \, \partial_{w_0} \mathcal{L} = -0.1 \cdot (-7) = 0.7.$$

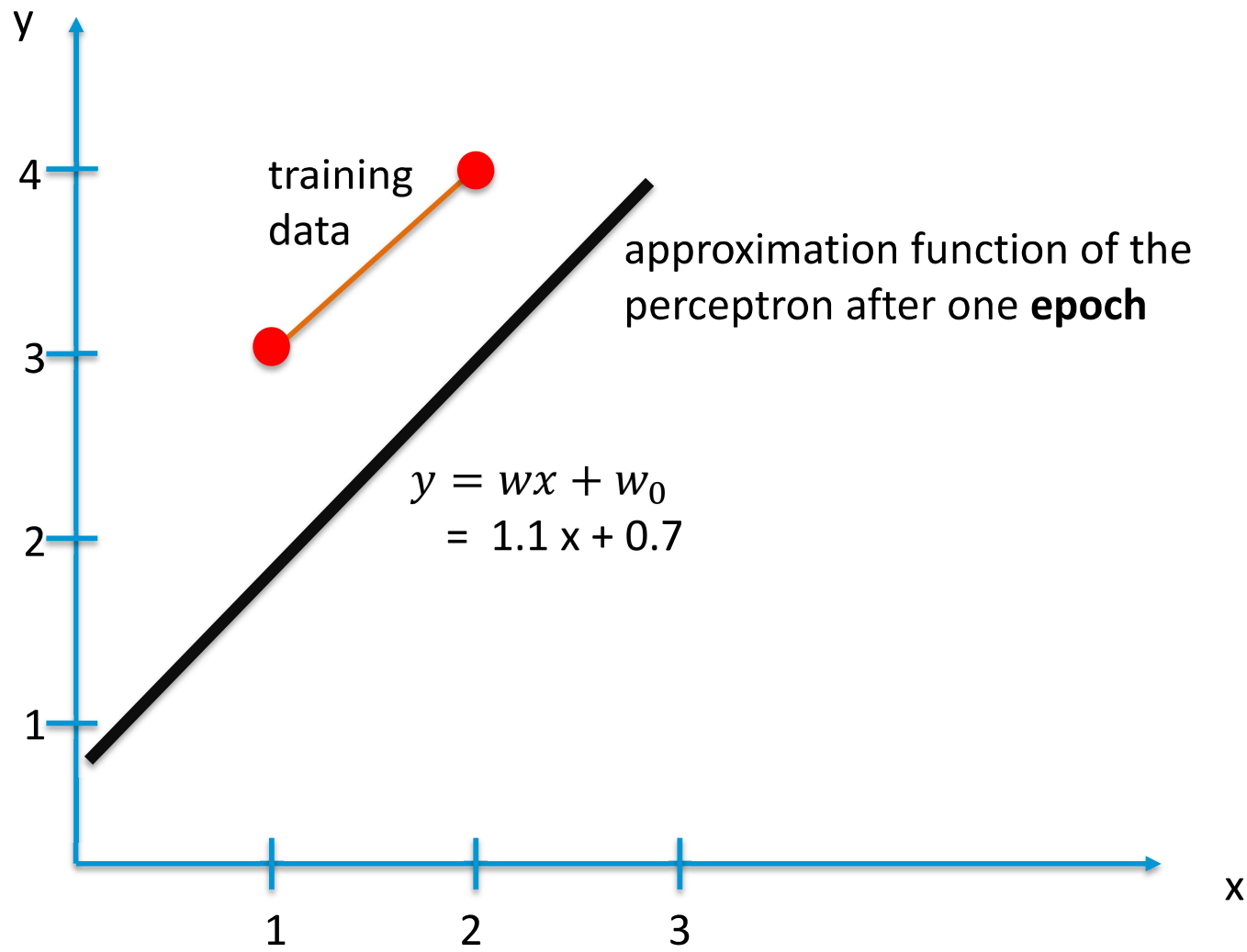
$$\Delta w = -\eta \, \partial_w \mathcal{L} = -0.1 \cdot (-11) = 1.1.$$

We add these correction values to the original weights to receive new weights

$$w_0 \leftarrow w_0 + \Delta w_0 = 0 + 0.7 = 0.7$$

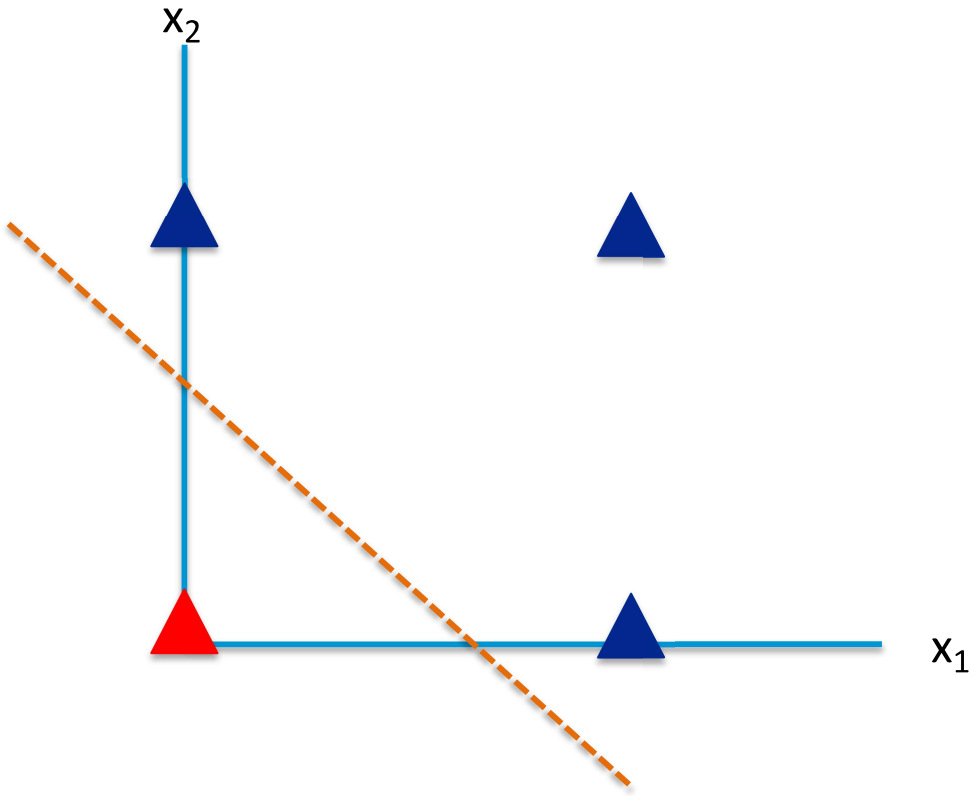
$$w \leftarrow w + \Delta w = 0 + 1.1 = 1.1$$

EXAMPLE 1
LEARNING A LINEAR FUNCTION



EXAMPLE 2
BINARY CLASSIFICATION

We will now let the perceptron solve a **binary classification problem**.
Input will consist of two values, x_1 and x_2. The perceptron shall represent a logical OR:
$y = x_1$ OR x_2

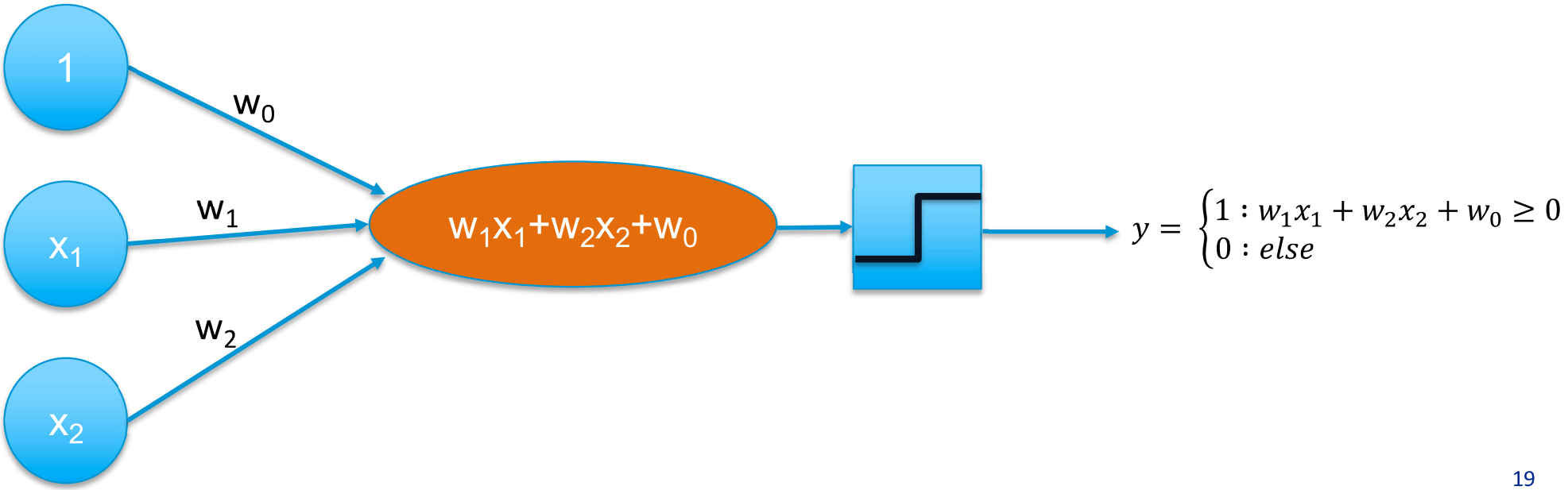


training data (x_1,x_2,y)
(1,0,true)
(1,1,true)
(0,1,true)
(0,0,false)

EXAMPLE 2
BINARY CLASSIFICATION

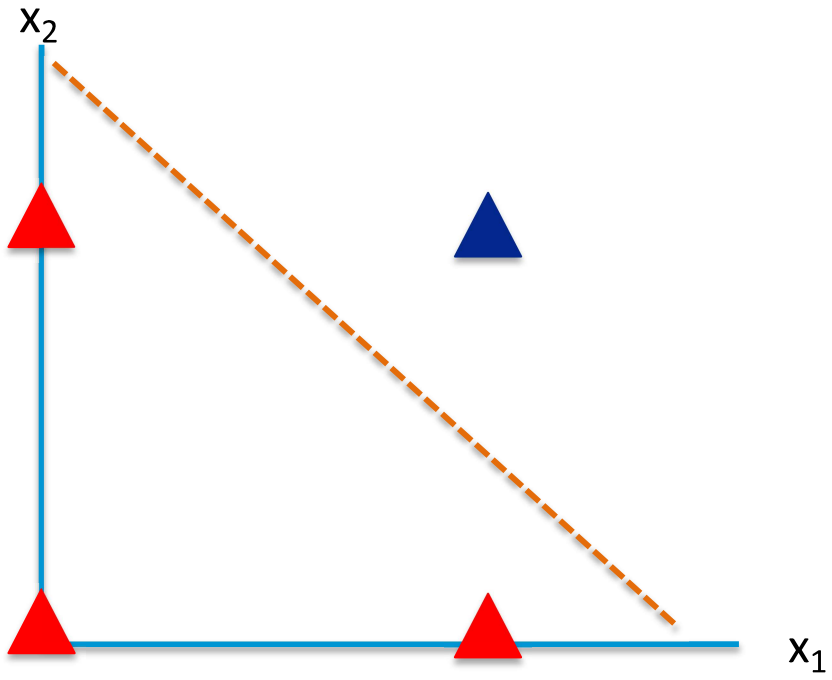
We will construct a perceptron with two input values x_1 and x_2, three weights w_1 and w_2 with bias w_0. As the activation function we choose the **step function**

$$s(a) = \begin{cases} 1: & a \geq 0 \\ 0: & a < 0 \end{cases}$$



EXAMPLE 2
BINARY CLASSIFICATION

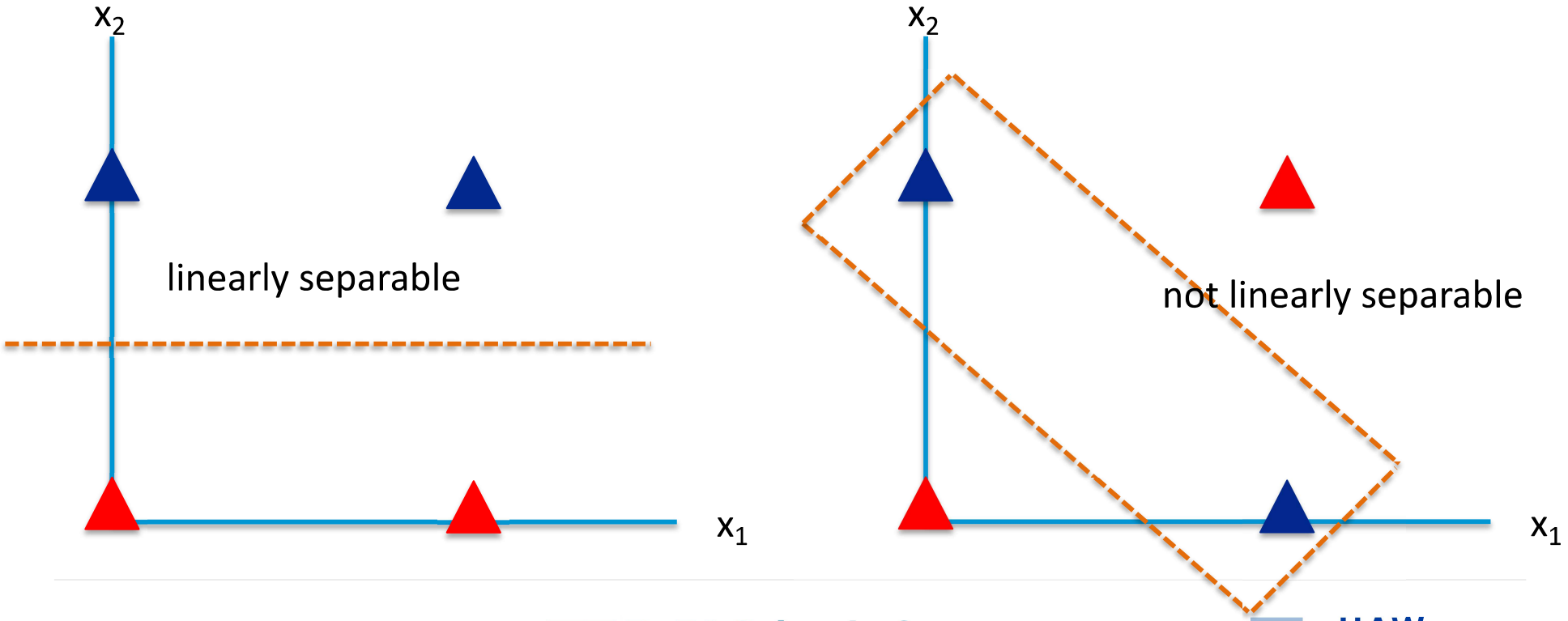
Another possibility would be to let the perceptron learn the logical AND:
$y = x_1$ AND x_2



training data (x_1,x_2,y)
(1,0,false)
(1,1,true)
(0,1,false)
(0,0,false)

EXAMPLE 2
BINARY CLASSIFICATION

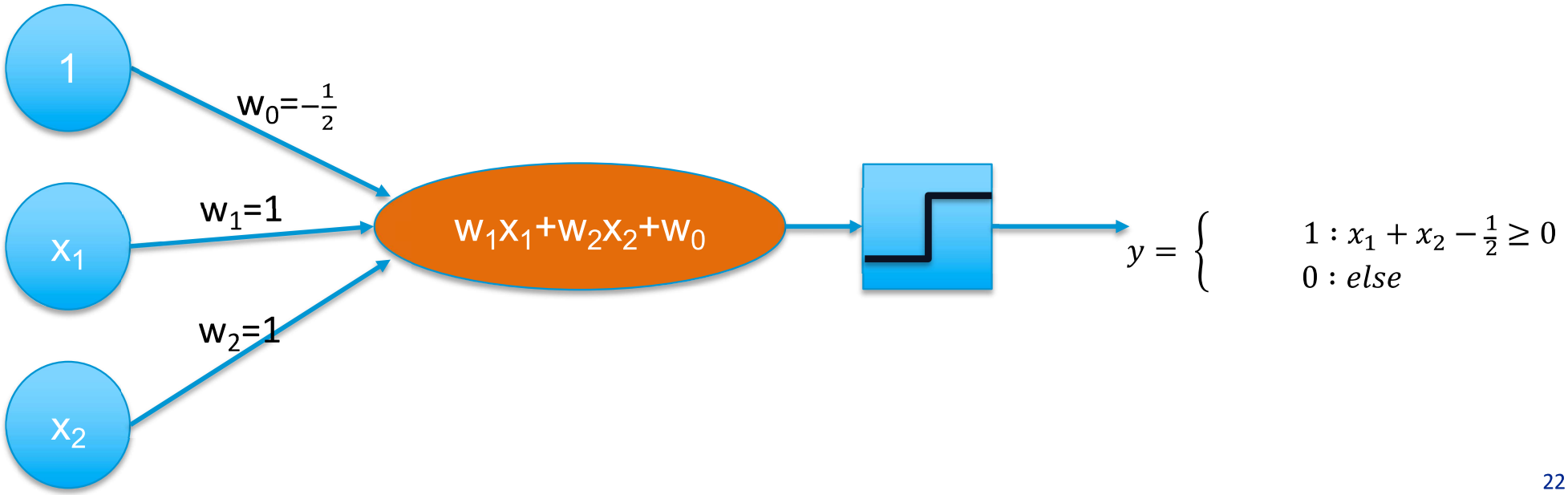
The left example contains data points which are **linearly separable**. The right example XOR is an example for a not linearly separable problem. This is because the **Vapnik-Chervonenkis-Dimension** of a line in two dimensions is 3. This means, a line in two dimensions can at most classify 3 points which have been arbitrarily assiged to two classes. To classify 4 points, rectangles are needed. The VC-dimension of a rectangle is 4. An example of a classificator with VC-dimension infinity is a look-up table.



EXAMPLE 2
BINARY CLASSIFICATION

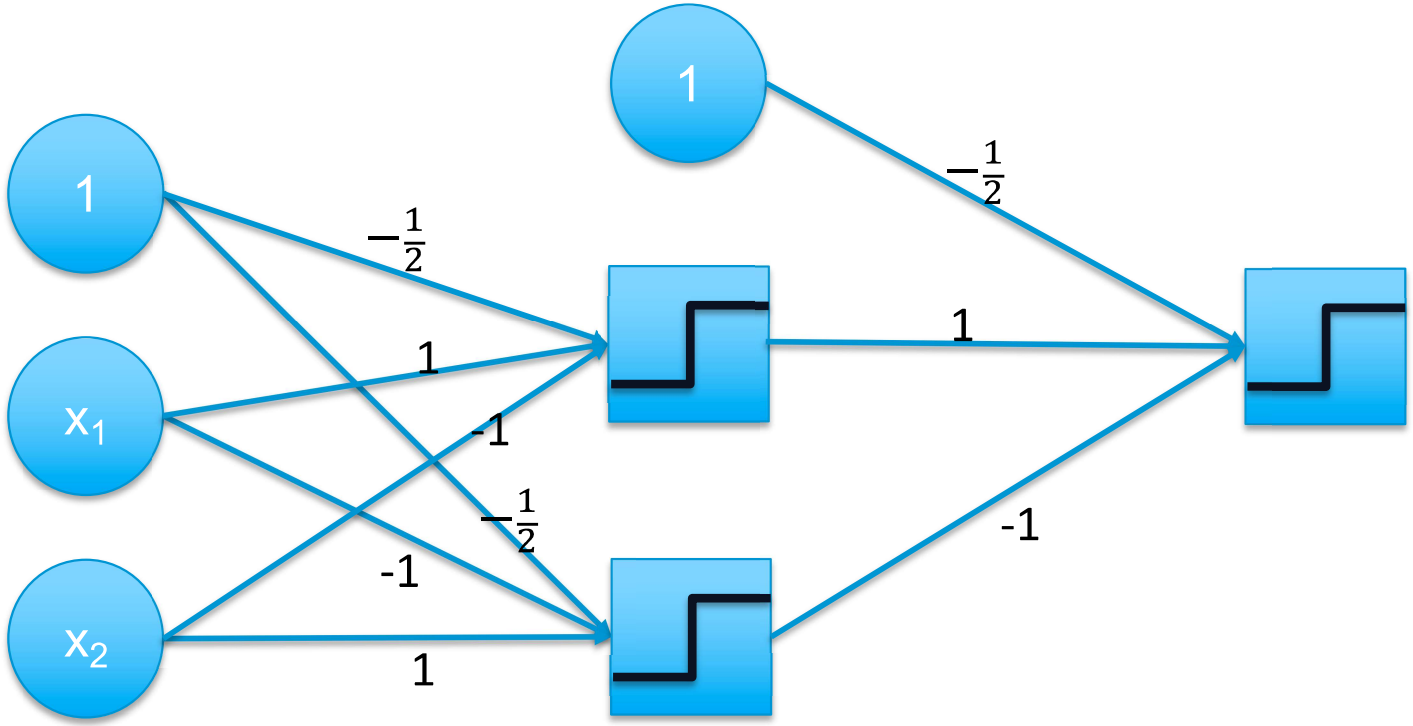
For the original classification problem **OR,** it can be seen easily, that weights $w_1 = w_2 = 1$ and w_0 = -½ solve the problem. This gives us

$$y = s(x_1 + x_2 - \tfrac{1}{2})$$



EXAMPLE 2
BINARY CLASSIFICATION

This **multi-layer perceptron** solves the binary XOR-problem



GLOSSARY

Perceptron An artificial neural net consisting of an input vector (input neurons) and a single output neuron. A multi-layer perceptron has a fully connected layer between input and output

Weights and Bias Factors which are adjusted by the learning algorithm which are multiplied with input values

Activation Function Function which is applied to the output of each neuron

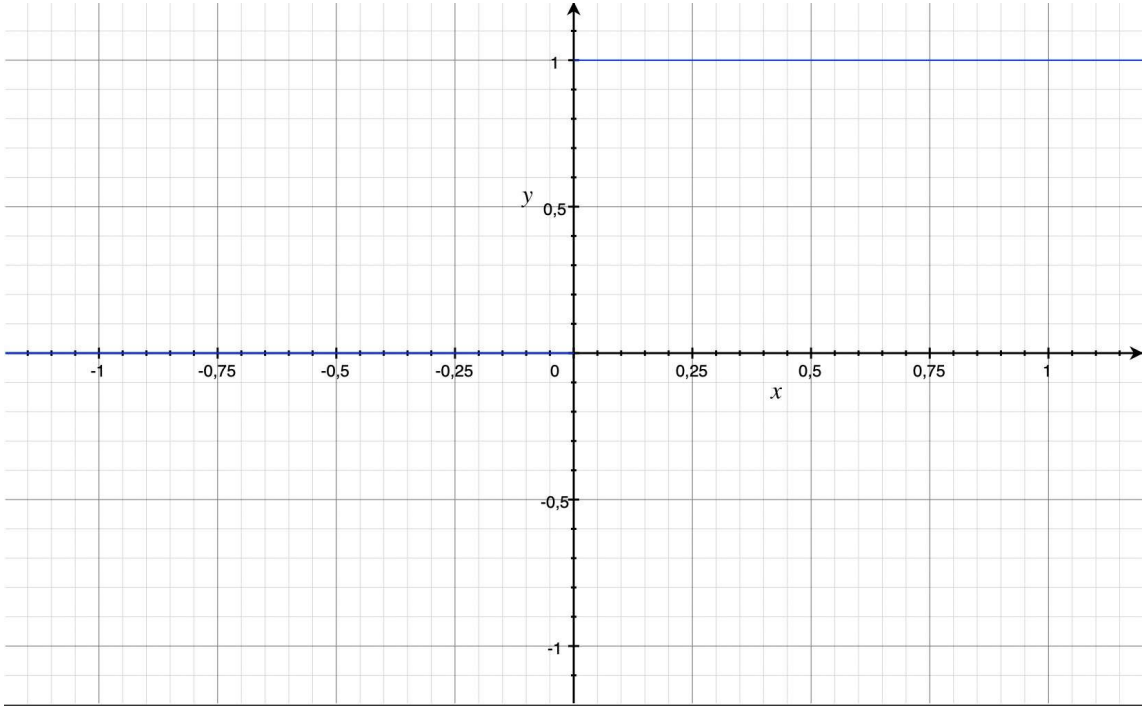
Loss Function Measure for deviation of the output of the neural net to the expected output

Backpropagation Algorithm to adjust the weights. The gradient of the loss function is applied to the weights of the artificial neural net

Learning Rate Factor with which the gradient of the loss function is applied to the weights

ACTIVATION FUNCTIONS

The are several **activation functions** which can be applied to the output of each neuron. We have already met the step function. The step function is not differentiable. It is reasonable to use an activation function which is differentiable because the backpropagation algorithm uses the gradient of the loss function.

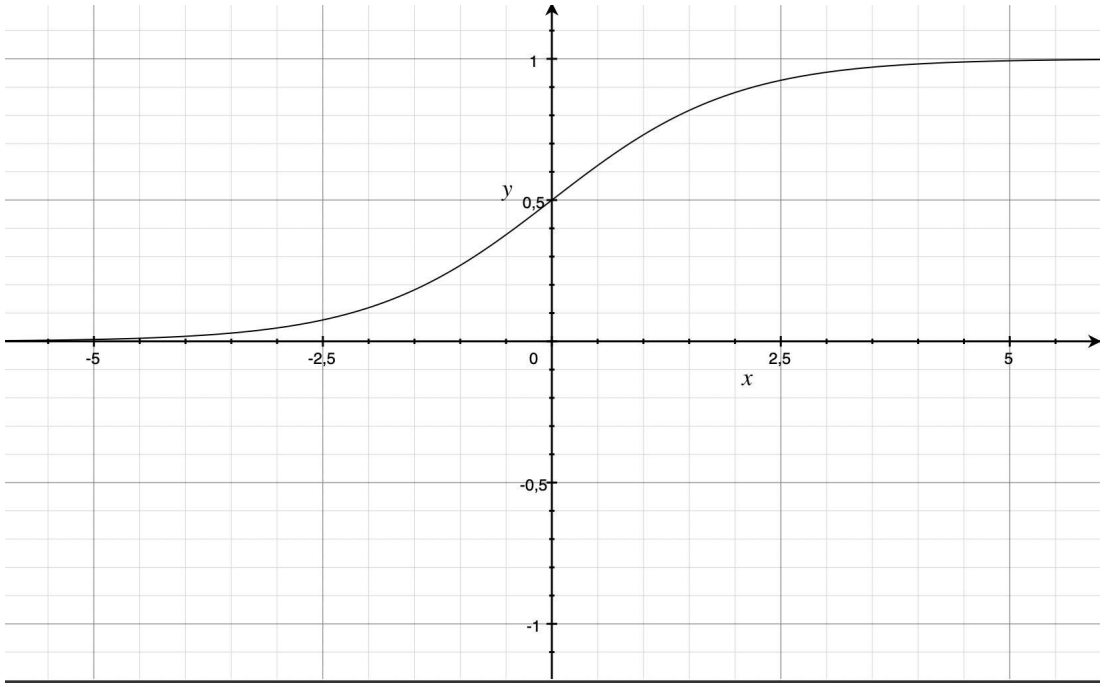


step function

ACTIVATION FUNCTIONS

The **sigmoid** or **logistic function** also returns values between 1 and 0 like the step function and can be used for a classifier. However, it is differentiable and better suited for backpropagation. The derivative is

$$\phi'(z) = \phi(z)(1 - \phi(z))$$



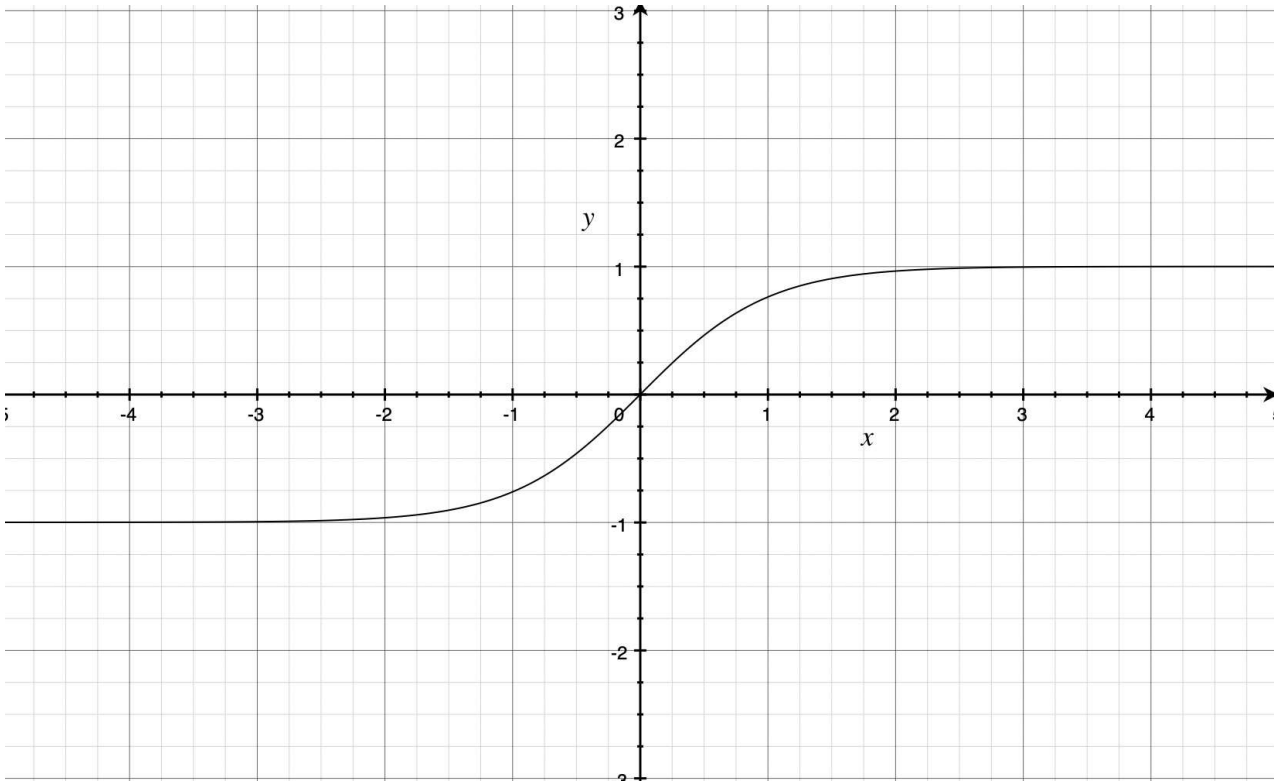
sigmoid or logistic function

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

The sigmoid function gives some problems in practical use: It is not zero-centered and during backpropagation gradient may vanish.

ACTIVATION FUNCTIONS

The **hyperbolic tangent** function saturates to 1 faster than the logistic function. It approaches -1 for large negative values of z, while the logistic function approaches 0.



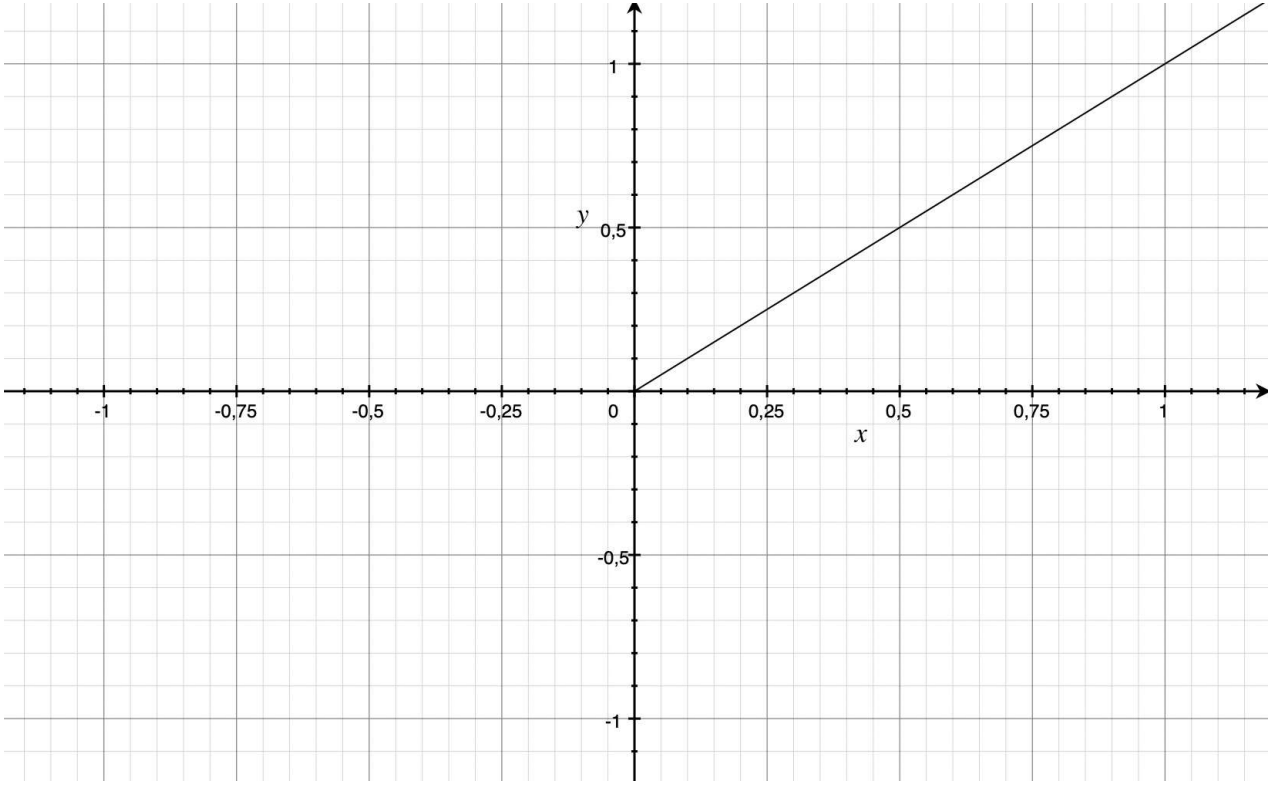
hyperbolic tangent

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

The *tanh* function is zero centered. However, also with this activation function gradients may vanish when the neural net has several layers.

ACTIVATION FUNCTIONS

The **rectified linear unit** or ReLu is a linear function for $z \geq 0$ and it is zero for $z < 0$. This makes it simple to calculate.



rectivied linear unit ReLU

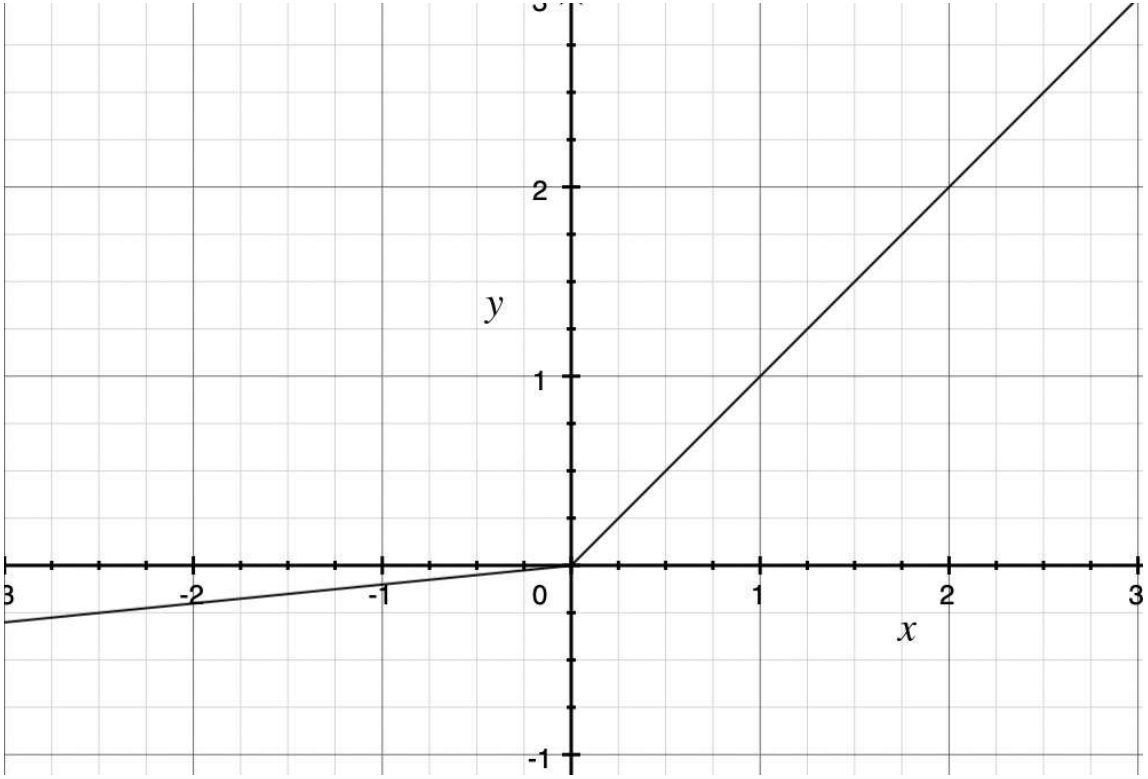
$$\phi(z) = \begin{cases} z : z \geq 0 \\ 0 : z < 0 \end{cases}$$

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

The ReLU is currently the most popular activation function. However, due to being zero for z<0, some neurons may *die* during training.

ACTIVATION FUNCTIONS

The **leaky rectified linear unit** or Leaky ReLu is a linear function for $z \geq 0$ and it is also a linear function for $z < 0$ with a small slope like $0.01 \cdot z$.



Leaky ReLU

$$\phi(z) = \begin{cases} z & : z \geq 0 \\ 0.01 \cdot z & : z < 0 \end{cases}$$

The Leaky ReLU tries to solve the problem of dying ReLU. There are also further varians of leaky ReLU.

ACTIVATION FUNCTIONS

The **softmax** activation function is used for multi-class classification in the last layer. While the logistic or sigmoid activation is good for binary (two-class) classification with a single output neuron, the softmax activation is used when you have an output neuron for each class you wish to classify. Suppose your k output neurons have values $z_1, \ldots, z_k$, then **softmax** is defined as

$$\phi(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

$\phi(z_i)$ gives the probability that class i is chosen. It holds that

$$\sum_{i=1}^{k} \phi(z_i) = 1$$

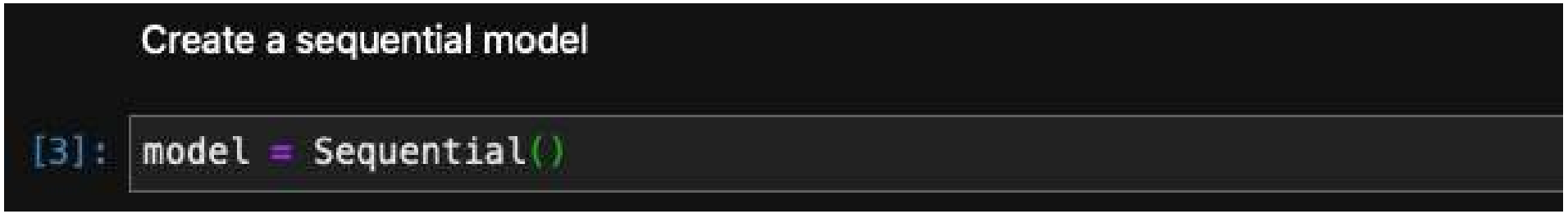
PERCEPTRON IN KERAS

We will now create the two-layer perceptron for the binary classification from the last example using keras. First we import the following modules.

```
[1]: from tensorflow import keras
     from keras.models import Sequential
     from keras.layers import Dense, Activation

     Using TensorFlow backend.
```

PERCEPTRON IN KERAS

Next we create a sequential neural network which we call *model.*



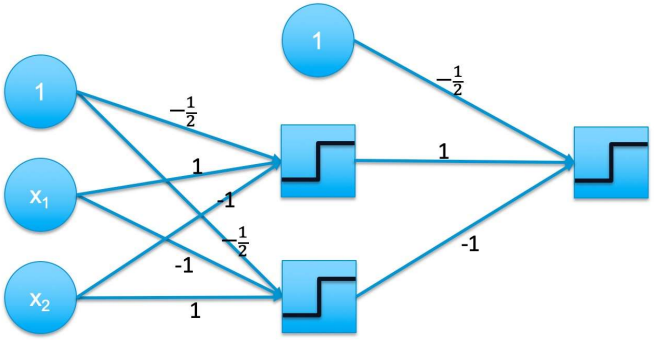
PERCEPTRON IN KERAS

```
[11]: model.add(Dense(units=2, input_shape=(2,)))
      model.add(Activation('sigmoid'))
      model.add(Dense(units=1))
      model.add(Activation('sigmoid'))
      model.compile(loss='mean_squared_error',optimizer='Adadelta')
      model.summary()
```

Layer (type)	Output Shape	Param #
dense_5 (Dense)	(None, 2)	6
activation_5 (Activation)	(None, 2)	0
dense_6 (Dense)	(None, 1)	3
activation_6 (Activation)	(None, 1)	0

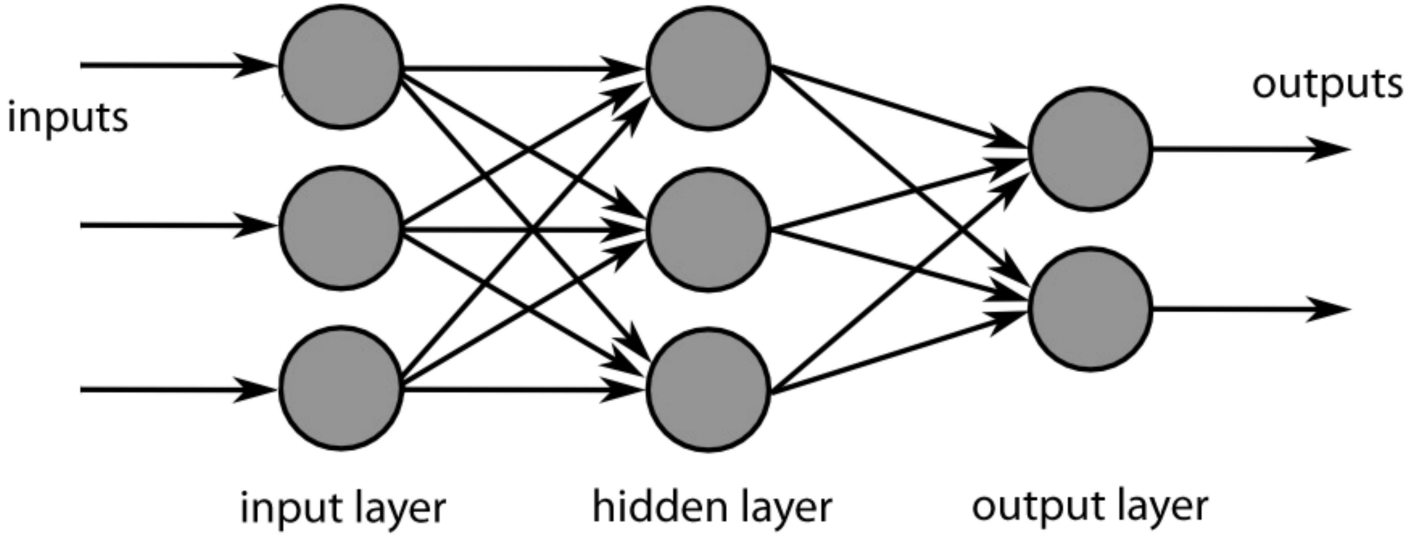
```
Total params: 9
Trainable params: 9
Non-trainable params: 0
```

Then two layers are added. The first layer has two input values and two output values. The activation function is the sigmoid function. Then we add the final layer with a single output neuron and again a sigmoid activation function.
As a loss function the MSE is defined. The optimizer Adadelta is a stochastic gradient descent with an adaptive learning rate for fast and stable convergence.
Then a summary of the model is printed.
There are 9 trainable parameters.



PERCEPTRON IN KERAS

Calculation of number of parameters in a neural net



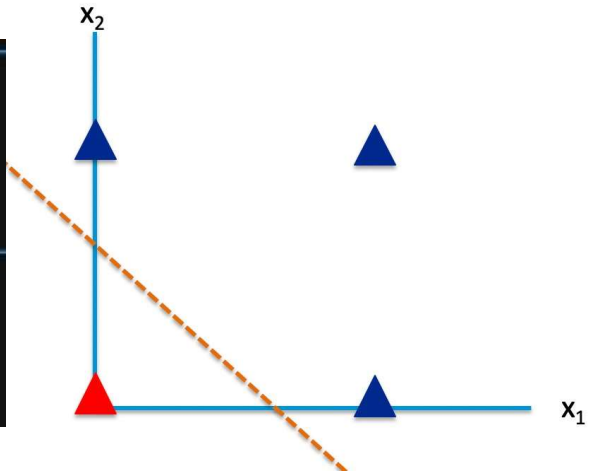
Picture taken from wikipedia

Between two layers with k and n neurons, there are $k \times n$ weights. Additionally there are the n biases of the downstream layer. This makes $k \times n + n$ parameters.
In the example above, there are 3 input neurons and 3 neurons in the intermediate layer. This makes $3 \times 3 + 3$ parameters. The output layer has 2 neurons. So together there are $(3 \times 3 + 3) + (3 \times 2 + 2) = 20$ parameters.

PERCEPTRON IN KERAS

The training data is generated. The numpy library is used to create an input array of the four values.

```
[6]: import numpy as np
     xs=np.array([[0.0,0.0],[0.0,1.0],[1.0,0.0],[1.0,1.0]], dtype=float)
     # ys=np.array([0.0,1.0,1.0,1.0], dtype=float)
     ys=np.array([[0.0],[1.0],[1.0],[1.0]], dtype=float)
     xs

[6]: array([[0., 0.],
            [0., 1.],
            [1., 0.],
            [1., 1.]])
```



The ys-array labels the first data point with 0 (red) and the other three data points with 1 (blue).

The the neural network is given the training data. 2000 epochs means, that 2000 times the full training data set is presented to the network.

```
[8]: model.fit(xs,ys,epochs=2000,verbose=0)
```

PERCEPTRON IN KERAS

After training, the neural network is tested with the training data.

```
[14]: print(model.predict(xs))

[[0.05992442]
 [0.9559951 ]
 [0.9592148 ]
 [0.98817456]]
```

The first value (0,0) has an expected value of 0 (red triangle). The neural network does not quite return 0, since a sigmoid activation function has been used.

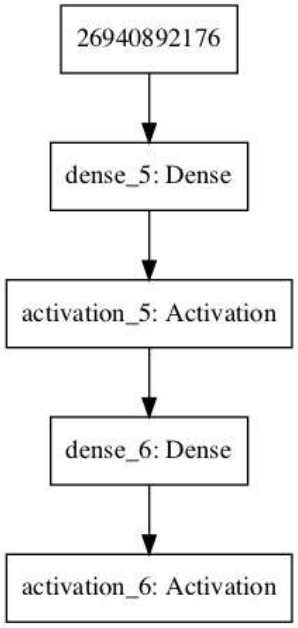
PERCEPTRON IN KERAS
VISUALIZATION AND LEARNING ANALYTICS

A simple way to get an overview over the model is with model.summary().

```
[11]: model.add(Dense(units=2, input_shape=(2,)))
      model.add(Activation('sigmoid'))
      model.add(Dense(units=1))
      model.add(Activation('sigmoid'))
      model.compile(loss='mean_squared_error',optimizer='Adadelta')
      model.summary()
```

Layer (type)	Output Shape	Param #
dense_5 (Dense)	(None, 2)	6
activation_5 (Activation)	(None, 2)	0
dense_6 (Dense)	(None, 1)	3
activation_6 (Activation)	(None, 1)	0

Total params: 9
Trainable params: 9
Non-trainable params: 0

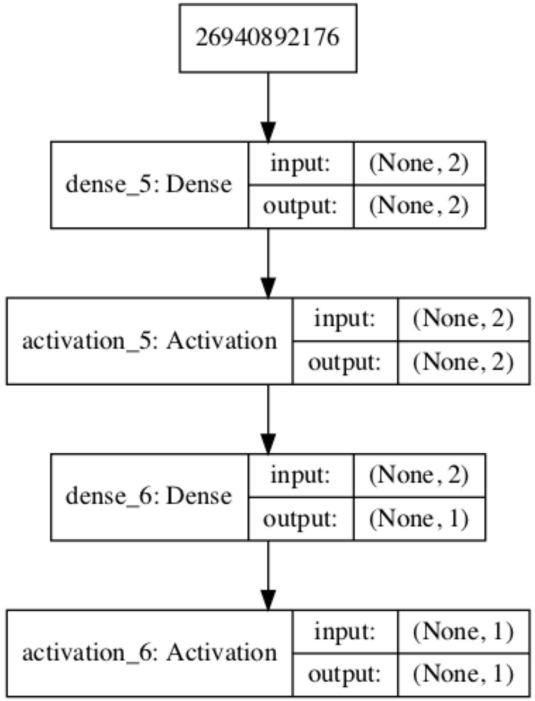


A graphical overview may be obtained by calling plot_model.

```
[17]: from keras.utils import plot_model
      plot_model(model, to_file='model.png')
```

PERCEPTRON IN KERAS
VISUALIZATION AND LEARNING ANALYTICS

A more elaborate picture of the model is obtained by setting show_shapes=True

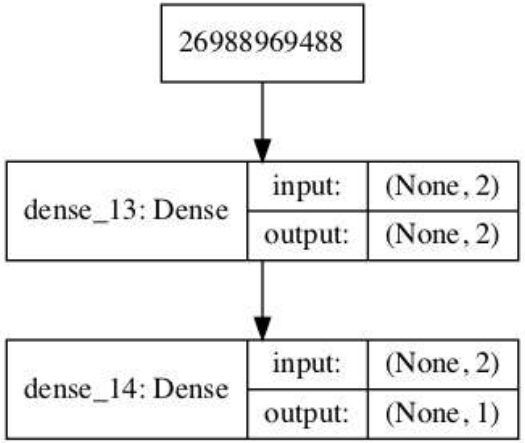



```
[18]: from keras.utils import plot_model
      plot_model(model, to_file='model.png', show_shapes=True)
```

PERCEPTRON IN KERAS
VISUALIZATION AND LEARNING ANALYTICS

If the model is created in a more compact writing by integrating the activation function into the Dense-call, then the activation is not depicted in the model overview. However, the resulting model is identical to the previous one.

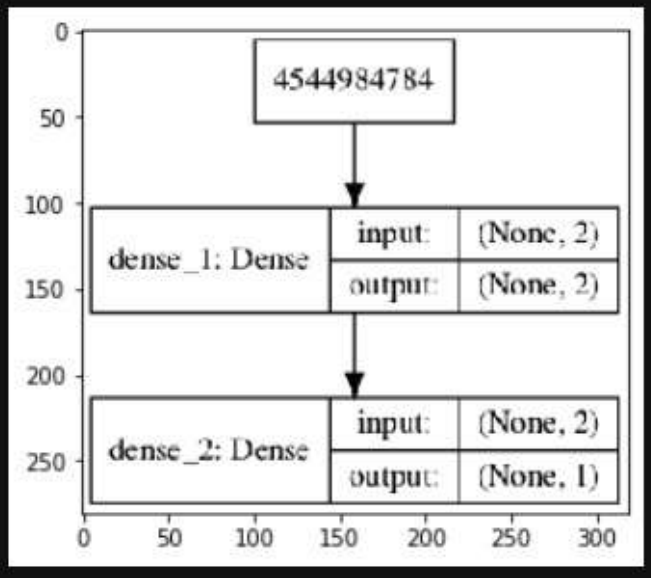
```
[32]: model.add(Dense(units=2, input_shape=(2,), activation='sigmoid'))
      model.add(Dense(units=1,activation='sigmoid' ))
      model.compile(loss='mean_squared_error',optimizer='Adadelta')
      model.summary()

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_13 (Dense)             (None, 2)                 6
_________________________________________________________________
dense_14 (Dense)             (None, 1)                 3
=================================================================
Total params: 9
Trainable params: 9
Non-trainable params: 0
_________________________________________________________________
```



PERCEPTRON IN KERAS
VISUALIZATION AND LEARNING ANALYTICS

If you want to display your model directly in jupyterlab, then you can use matplotlib to display the image.

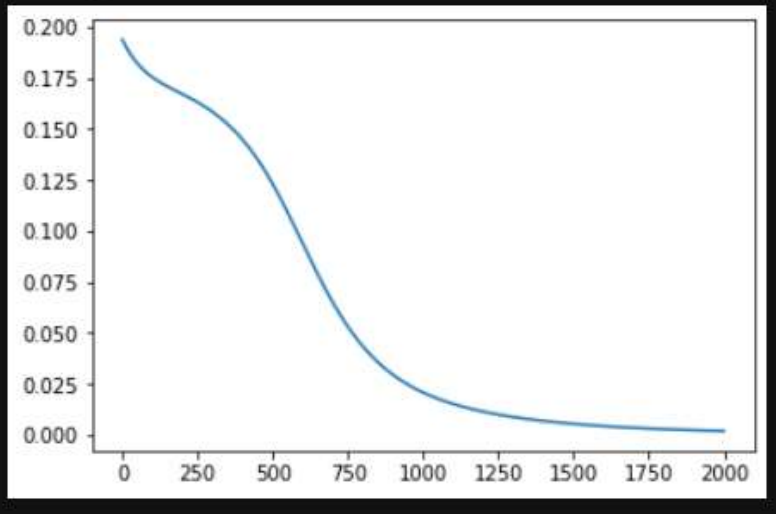
```
[13]: import matplotlib.pyplot as plt
      import matplotlib.image as mpimg
      from keras.utils import plot_model
      plot_model(model, to_file='model.png', show_shapes=True)
      image = mpimg.imread("model.png")
      plt.imshow(image)
      plt.show()
```



PERCEPTRON IN KERAS
VISUALIZATION AND LEARNING ANALYTICS

The fit-call returns a history object. Use print(history.history.keys()) to see which data is available in the history object. Use the matplotlib library to display the values. More advanced analytics can be done by passing a callback function to the fit-call and using **TensorBoard** for visualization.

```
[28]: history=model.fit(xs,ys,epochs=2000,verbose=0)

[36]: import matplotlib.pyplot as plt
      print(history.history.keys())
      plt.plot(history.history['loss'])

      dict_keys(['loss'])
[36]: [<matplotlib.lines.Line2D at 0x649b530d0>]
```



PERCEPTRON IN KERAS
VISUALIZATION AND LEARNING ANALYTICS

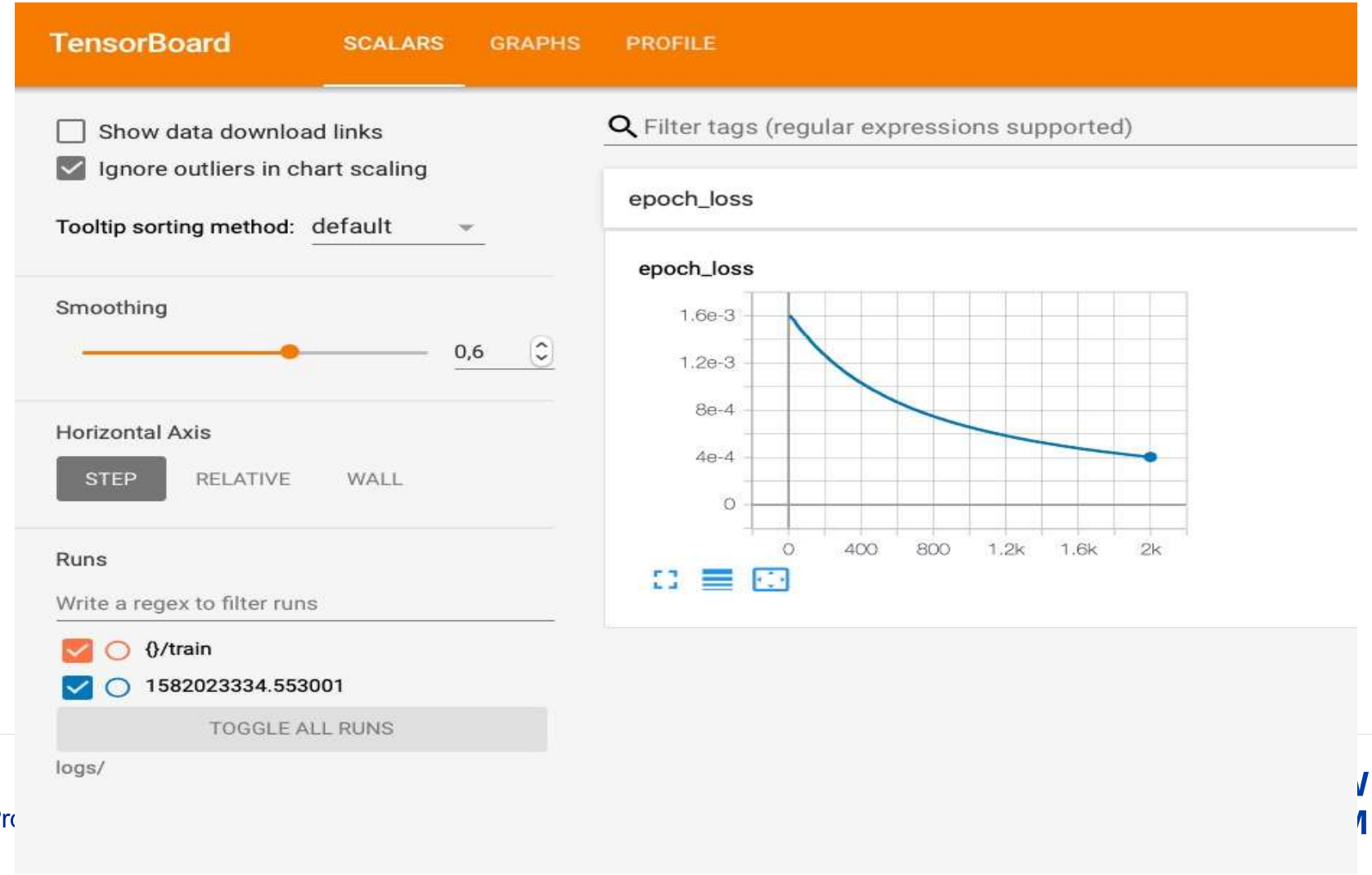
To use TensorBoard, import it from tensorflow.keras.callbacks. Then include a callback function inside of the fit-call as shown.

```
[20]: from tensorflow.keras.callbacks import TensorBoard
      import time
      tensorboard = TensorBoard(log_dir="logs/{}".format(time.time()))
      history=model.fit(xs,ys,epochs=2000,verbose=0, callbacks=[tensorboard])
```

Next, open a terminal inside of jupyterlab (File->New->Terminal). Start tensorboard by typing `tensorboard -- logdir=`logs/`.` This will return a URL to your localhost. Open the link in your browser and you have tensorboard.

PERCEPTRON IN KERAS
VISUALIZATION AND LEARNING ANALYTICS

```
(base) Stephans-iMac:~ stephanpareigis$ tensorboard --logdir='logs/'
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --bind_all
TensorBoard 2.1.0 at http://localhost:6006/ (Press CTRL+C to quit)
```



PERCEPTRON IN KERAS
VISUALIZATION AND LEARNING ANALYTICS