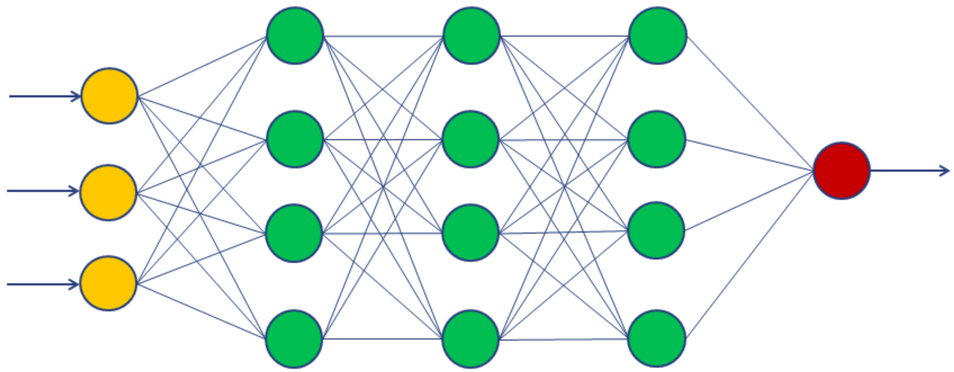
R-course:

Machine Learning using R

Neural Networks



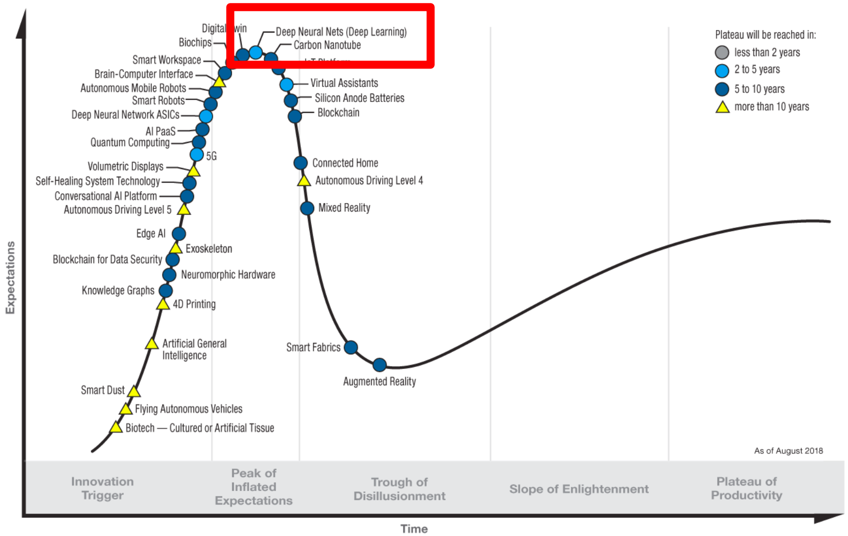
Yannick Rothacher

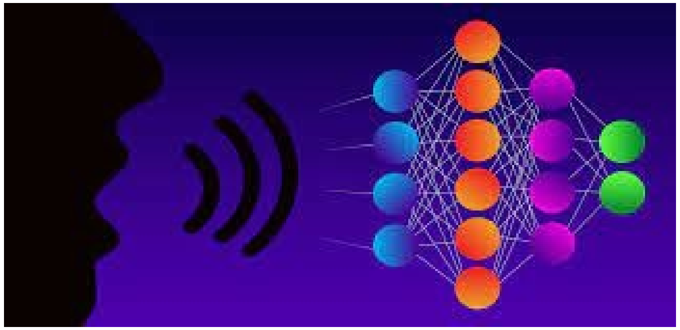
Zürich, 2021

Artificial Neural Networks (today)

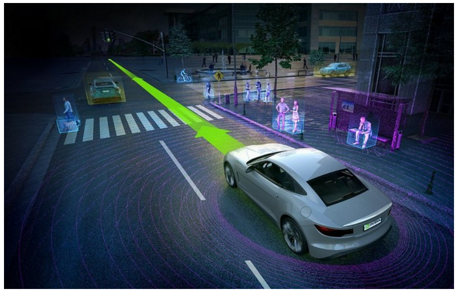
- (Deep) Neural Networks are in trend!
  - Also referred to as "Deep Learning"
- Successful in many ML-competitions
  - Applications in various fields:

Hype Cycle for Emerging Technologies, 2018 (gartner.com)



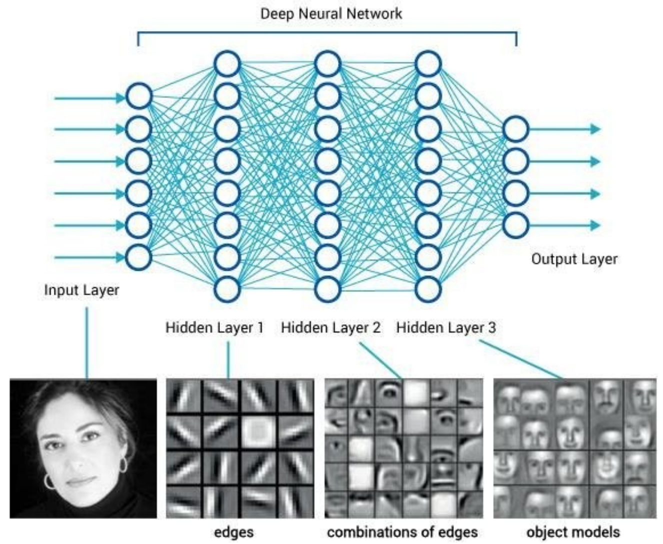


Speech recognition



Autonomous driving

Reinforcement learning (e.g. Alpha Go)

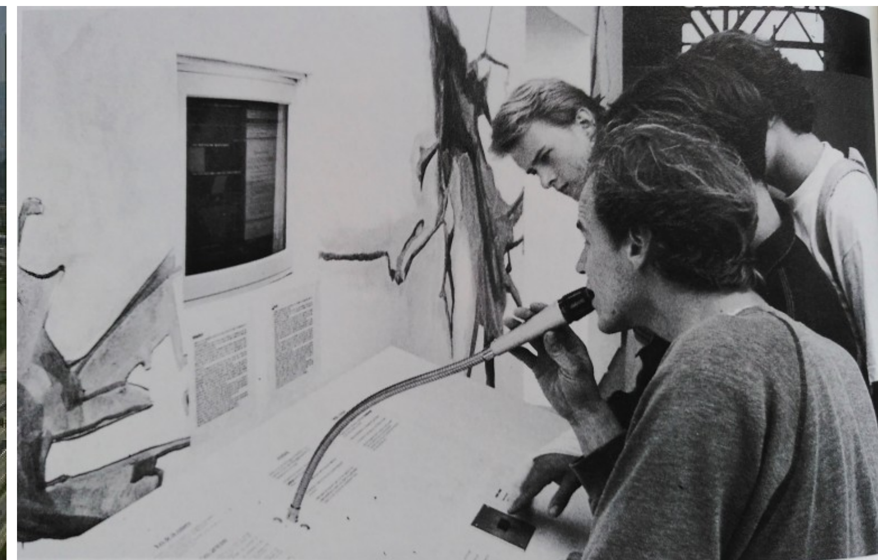


Picture recognition

Artificial Neural Networks (yesterday)

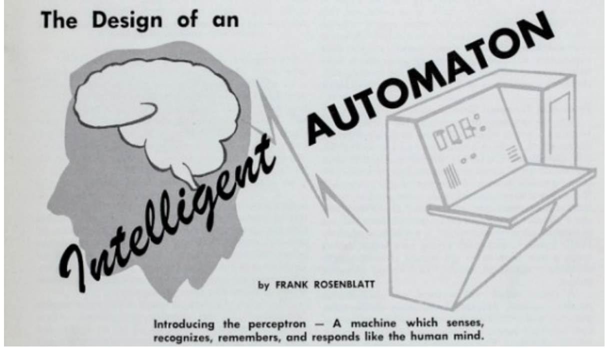
- **Heureka** exhibition in Zürich, Brunau (**1991**)
- Presented the "Forschungsstandort Schweiz"

https://www.e-pics.ethz.ch/index/ethbib.bildarchiv/ETHBIB.Bildarchiv_Com_FC24-8002-0196_24364.html



2
Einzelworterkennung mit neuronalen Netzen (6.3.1)
Neuronale Netze werden seit kurzem in einer Vielzahl verschiedener Gebiete verwendet: Bildverarbeitung, Signalbereinigung, Trendanalyse, Regelungstechnik usw. Daneben werden sie weltweit auf ihre Tauglichkeit zur Erkennung gesprochener Sprache untersucht. Das Problem dabei ist, dass die Erkennung ganzer Wörter sprecherunabhängig sein soll. Als Beispiel suchen wir in einem Computer ein Dokument anstatt mit einer Maus mittels gesprochener Schlüsselwörter.

Artificial Neural Networks (yesterday)



- The idea of artificial neural networks is very old
  - First reference dates back to **1944**
    (Warren S. McCulloch and Walter Pitts)
- The "**perceptron**" (the first "modern" neural network)
  - Invented by psychologist **F. Rosenblatt** (**1958**)

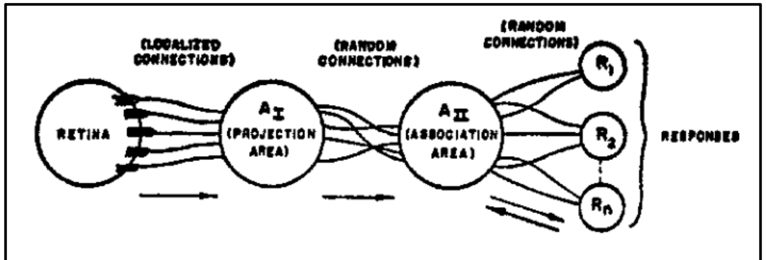
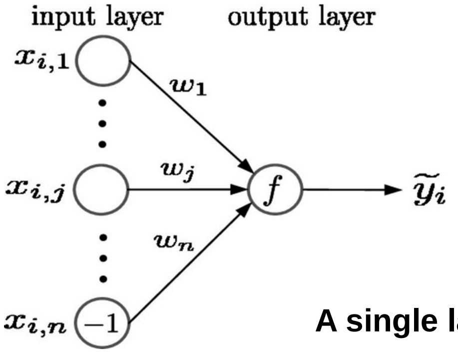
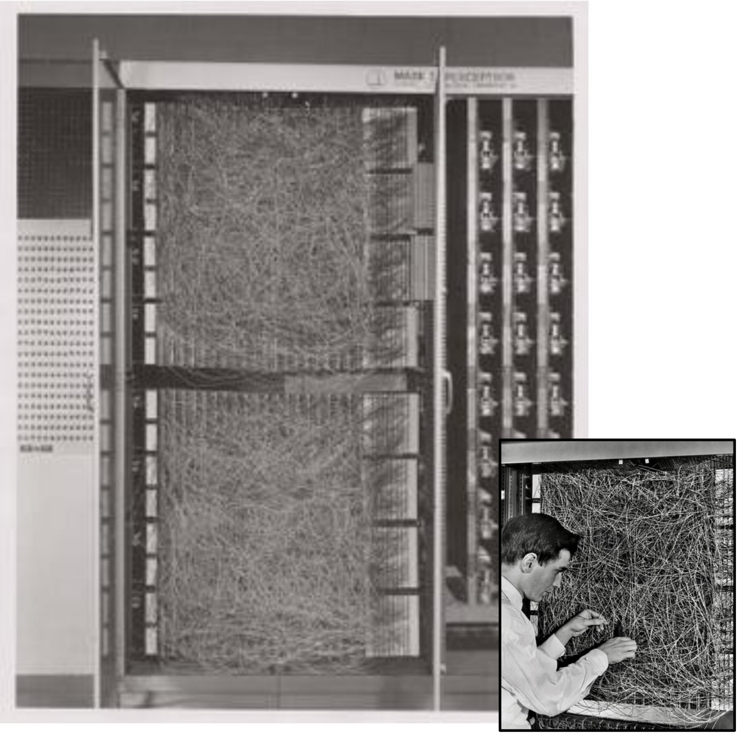


FIG. 1. Organization of a perceptron.

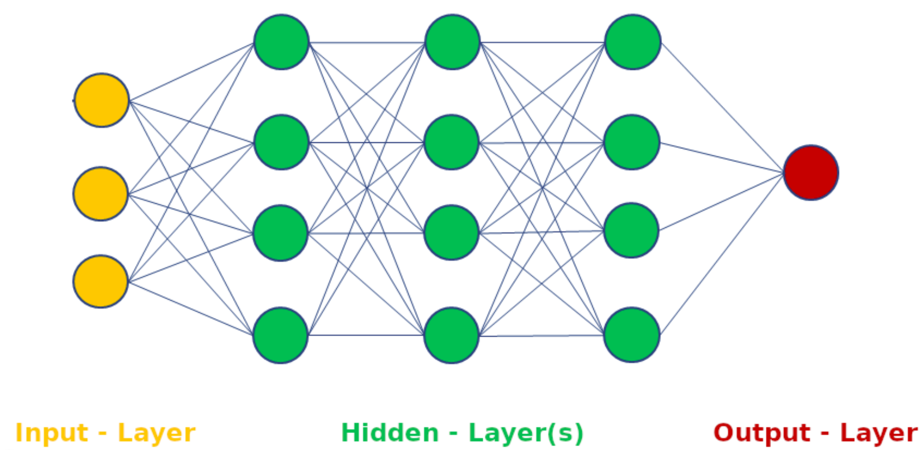


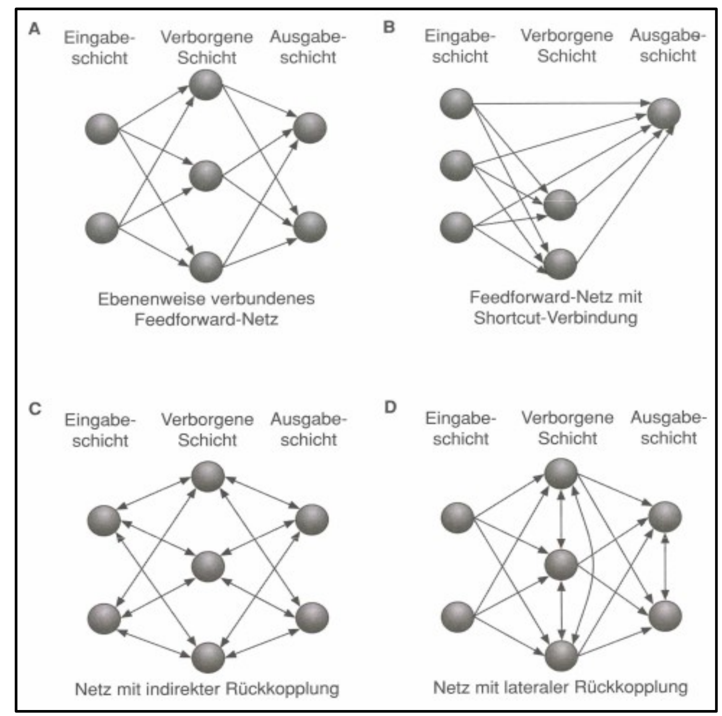
A single layer perceptron


Mark I Perceptron machine

Structure of Neural Networks (NN)

- Usually represented as connected **nodes** (neurons) organized in **layers**
  - Different structures are possible
- We will focus on **feed-forward** networks



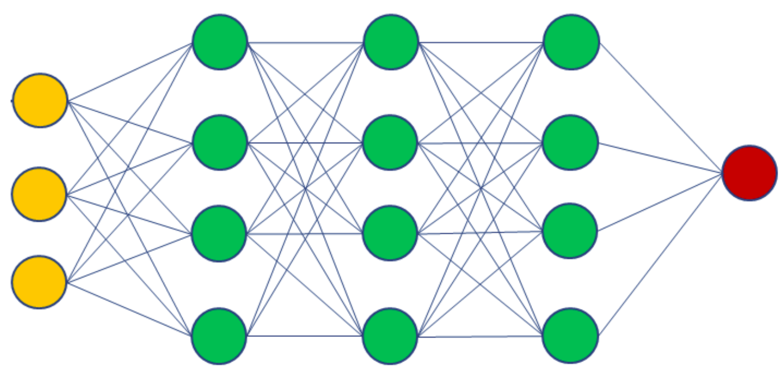


Source: Holling & Schmitz (2010)

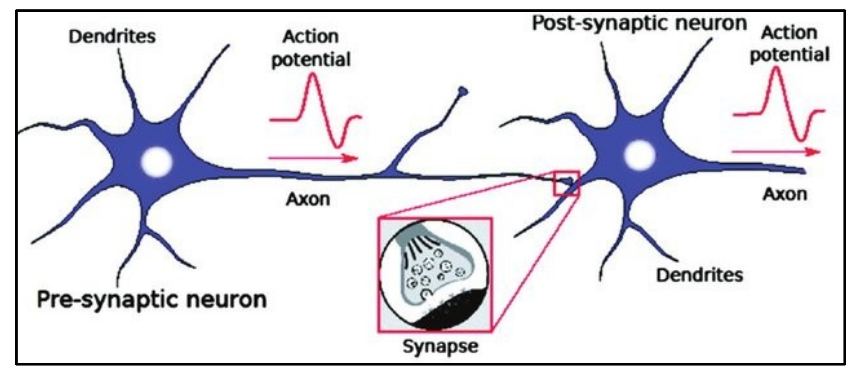
- Structure consists of **input**, **hidden** and **output** layer(s)
- Connections are associated with specific **weights** (strength of connection)
  - Nodes are associated with specific **activation functions** (later)
- The structure is loosely inspired by real-life neurons

Structure of Neural Networks (NN)

- Usually represented as connected **nodes** (neurons) organized in **layers**
  - Different structures are possible
- We will focus on **feed-forward** networks



Input - Layer   Hidden - Layer(s)   Output - Layer



Source: Huang et al. (2018)

- Structure consists of **input**, **hidden** and **output** layer(s)
- Connections are associated with specific **weights** (strength of connection)
  - Nodes are associated with specific **activation functions** (later)
- The structure is loosely inspired by real-life neurons

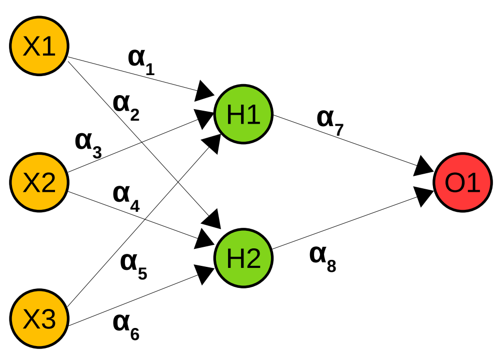
Single hidden layer NN

- Supervised learning with NNs (predict target variable)
  - Exemplary regression data:

Target variable (y)

Training time (X1)	Sleep time (X2)	Body height (X3)	**Performance (0-100)**
2	8	1.8	60
8	9	1.6	100
5	4	1.7	85
8	5	1.8	79
7	7	1.8	62
...	...	...	...

- Applied (single hidden layer) NN:



Input
Size = 3
(3 predictors)

Hidden
Size = 2
(free to choose)

Output
Size = 1
(y is numeric)

$\alpha_1 - \alpha_8$: connection weights

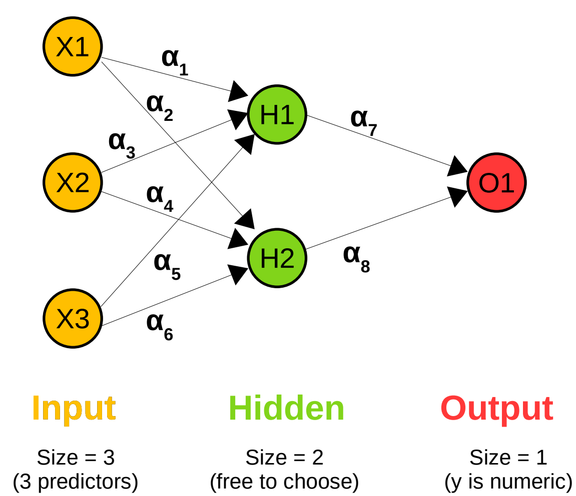
Single hidden layer NN

- Supervised learning with NNs (predict target variable)
  - Exemplary regression data:

Target variable (y)

Training time (X1)	Sleep time (X2)	Body height (X3)	**Performance (0-100)**
2	8	1.8	60
8	9	1.6	100
5	4	1.7	85
8	5	1.8	79
7	7	1.8	62
...	...	...	...

- Applied (single hidden layer) NN:



$\alpha_1 - \alpha_8$ **:** connection weights

- The goal is to find the weights, which allow the prediction of **y**
  - How the prediction works (**slightly simplified**, see later):

X1	X2	X3	**Performance (0-100)**
3	5	1.7	?

Value at H1: H1 = α_1***3** + α_3***5** + α_5***1.7**

Value at H2: H2 = α_2***3** + α_4***5** + α_6***1.7**

Value at O1 (= Prediction): O1 = α_7***H1** + α_8***H2**

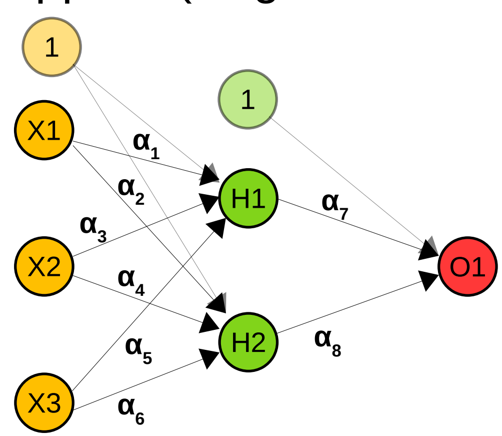
Single hidden layer NN

- Supervised learning with NNs (predict target variable)
  - Exemplary regression data:

Target variable (y)

Training time (X1)	Sleep time (X2)	Body height (X3)	**Performance (0-100)**
2	8	1.8	60
8	9	1.6	100
5	4	1.7	85
8	5	1.8	79
7	7	1.8	62
...	...	...	...

- Applied (single hidden layer) NN:



Input Size = 3 (3 predictors)

Hidden Size = 2 (free to choose)

Output Size = 1 (y is numeric)

$\alpha_1 - \alpha_8$: connection weights

- The goal is to find the weights, which allow the prediction of **y**
  - How the prediction works (**slightly simplified**, see later):

X1	X2	X3	**Performance (0-100)**
3	5	1.7	?

Value at H1: $H1 = \alpha_1 * \mathbf{3} + \alpha_3 * \mathbf{5} + \alpha_5 * \mathbf{1.7}$

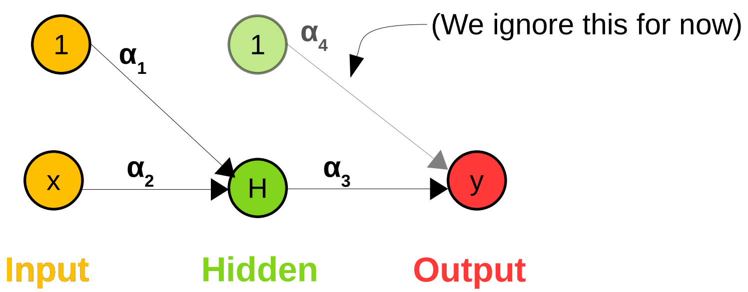
Value at H2: $H2 = \alpha_2 * \mathbf{3} + \alpha_4 * \mathbf{5} + \alpha_6 * \mathbf{1.7}$

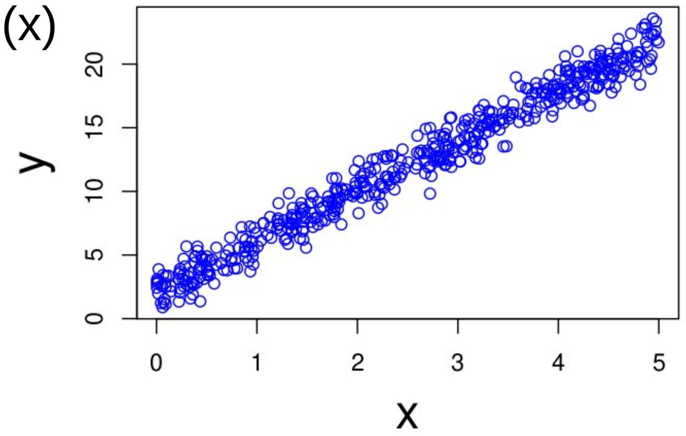
Value at O1 (= Prediction): $O1 = \alpha_7 * \mathbf{H1} + \alpha_8 * \mathbf{H2}$

- "Biases" are often added at each layer

Prediction of NN in more detail

- Univariate regression: Only one predictor (x)
  - $y = b_0 + b_1x + \varepsilon \quad (\varepsilon \sim N(0,\sigma^2))$
- Applied Neural Network:

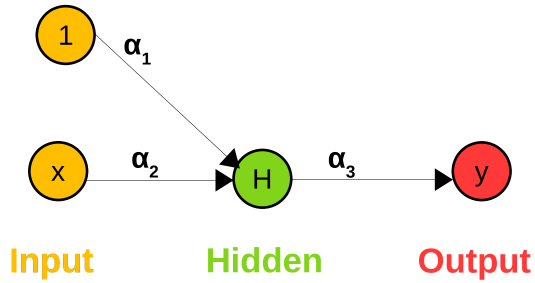


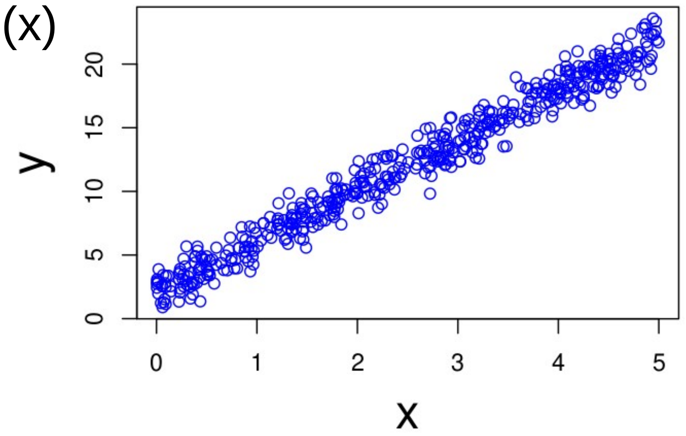


<u>Truth:</u>
$b_0 = 2$
$b_1 = 4$

Prediction of NN in more detail

- Univariate regression: Only one predictor (x)
  - $y = b_0 + b_1x + \varepsilon \quad (\varepsilon \sim N(0,\sigma^2))$
- Applied Neural Network:

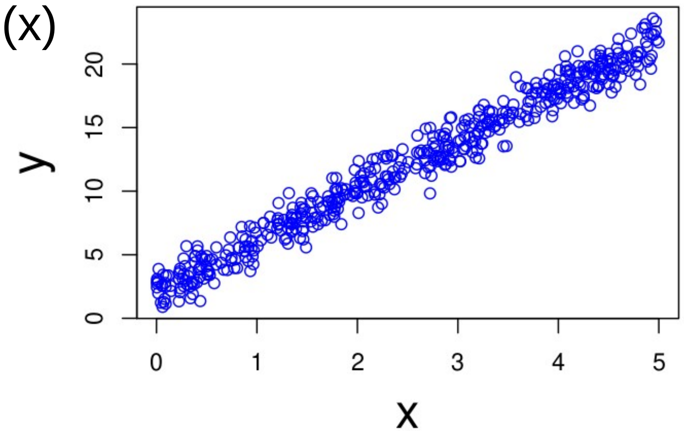




<u>Truth:</u>
$b_0 = 2$
$b_1 = 4$

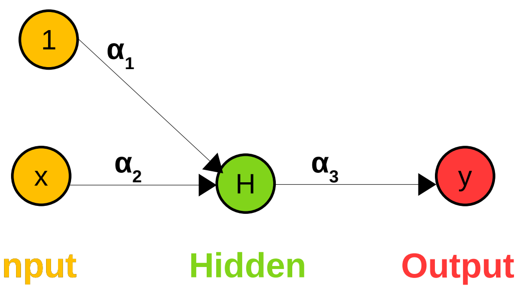
Prediction of NN in more detail

- Univariate regression: Only one predictor (x)
  - $y = b_0 + b_1x + \varepsilon \quad (\varepsilon \sim N(0,\sigma^2))$



<u>Truth:</u>
$b_0 = 2$
$b_1 = 4$

- Applied Neural Network:



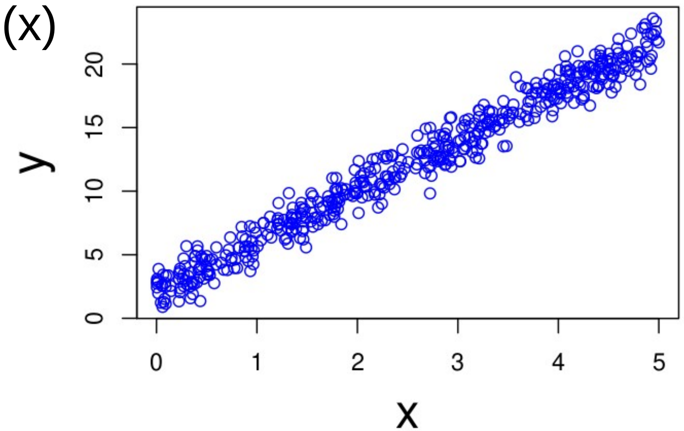
- Prediction of y:

$H = \alpha_1 + \alpha_2 * x$
$y = \alpha_3 * H$

→ Ideal solution:
$\alpha_1 = b_0 = 2$
$\alpha_2 = b_1 = 4$
$\alpha_3 = 1$

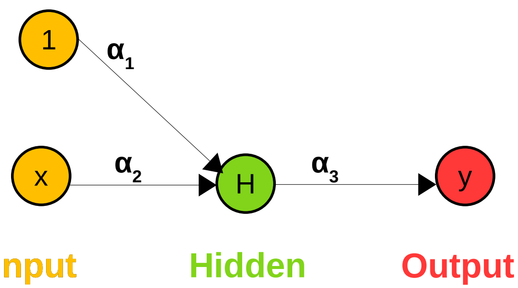
Prediction of NN in more detail

- Univariate regression: Only one predictor (x)
  - $y = b_0 + b_1 x + \varepsilon \quad (\varepsilon \sim N(0,\sigma^2))$



<u>Truth:</u>
$b_0 = 2$
$b_1 = 4$

- Applied Neural Network:



- Prediction of y:

$H = \alpha_1 + \alpha_2 * x$
$y = \alpha_3 * H$

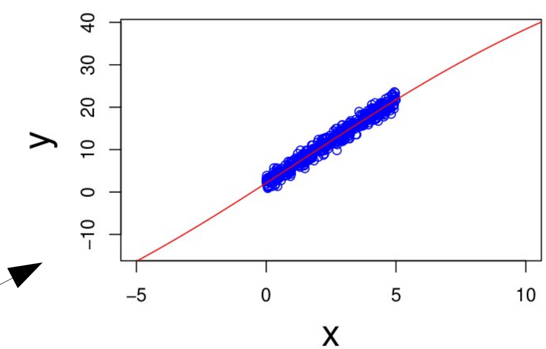
→ Ideal solution:
$\alpha_1 = b_0 = 2$
$\alpha_2 = b_1 = 4$
$\alpha_3 = 1$

Let's compare with solution of NN ...

Fitted with **nnet()** function:
$\alpha_1 = -0.19$
$\alpha_2 = 0.14$
$\alpha_3 = 115.93$

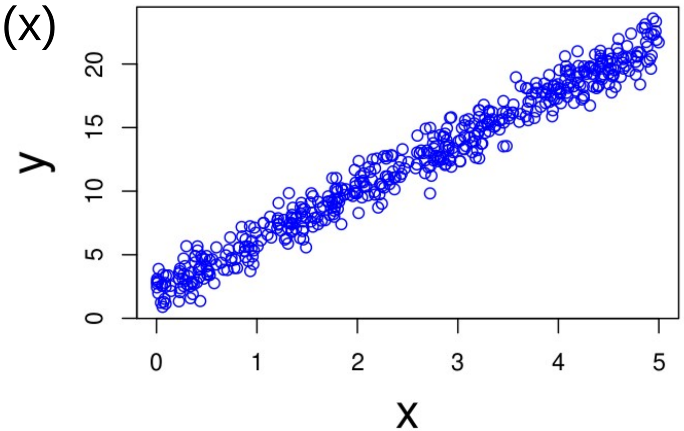
!?

Look at predictions



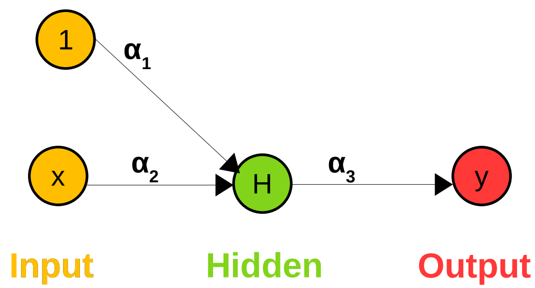
Prediction of NN in more detail

- Univariate regression: Only one predictor (x)
  - $y = b_0 + b_1 x + \varepsilon \quad (\varepsilon \sim N(0,\sigma^2))$



Truth:
$b_0 = 2$
$b_1 = 4$

- Applied Neural Network:



- Prediction of y:

$H = \alpha_1 + \alpha_2 * x$
$y = \alpha_3 * H$

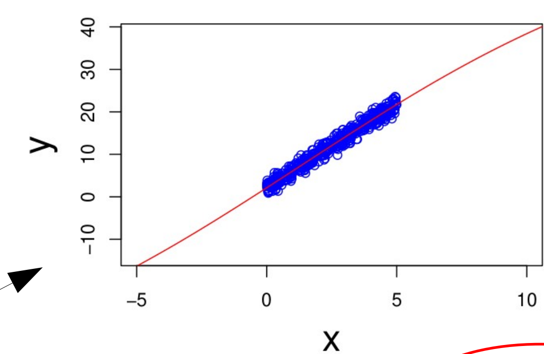
→ Ideal solution:
$\alpha_1 = b_0 = 2$
$\alpha_2 = b_1 = 4$
$\alpha_3 = 1$

Let's compare with solution of NN ...

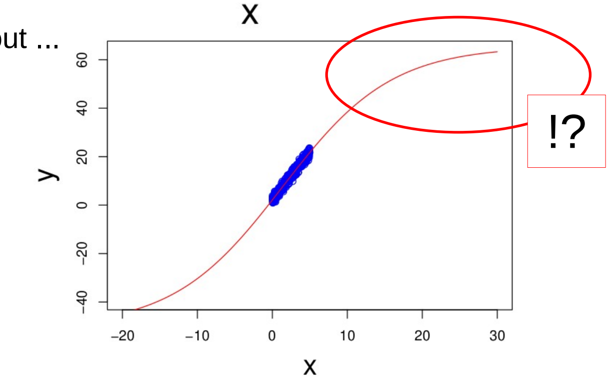
Fitted with **nnet()** function:
$\alpha_1 = -0.19$
$\alpha_2 = 0.14$
$\alpha_3 = 115.93$

!?

Look at predictions



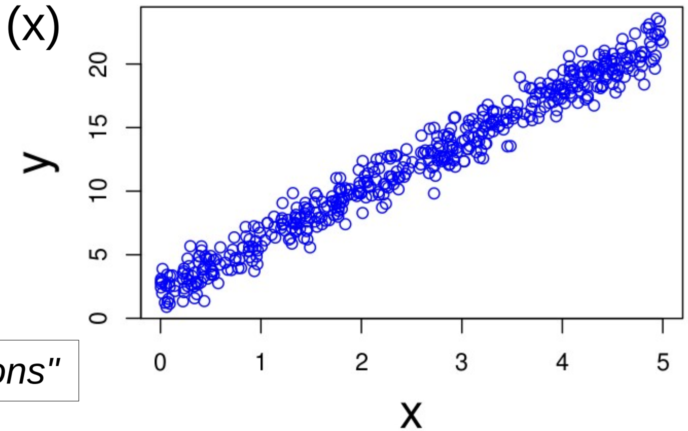
Zoom out ...



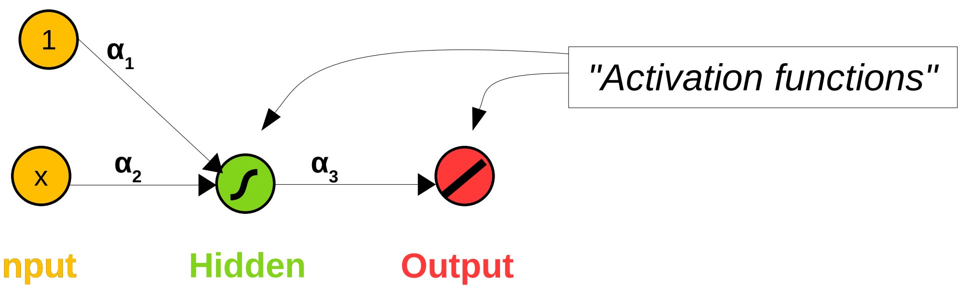
!?

Prediction of NN in more detail

- Univariate regression: Only one predictor (x)
  - $y = b_0 + b_1x + \varepsilon \quad (\varepsilon \sim N(0,\sigma^2))$



- Applied Neural Network:



- Prediction of y:

$H = \alpha_1 + \alpha_2 * x$
$y = \alpha_3 * H$

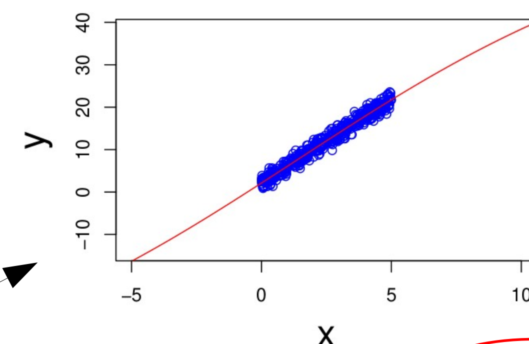
→ Ideal solution:
$\alpha_1 = b_0 = 2$
$\alpha_2 = b_1 = 4$
$\alpha_3 = 1$

Let's compare with solution of NN ...

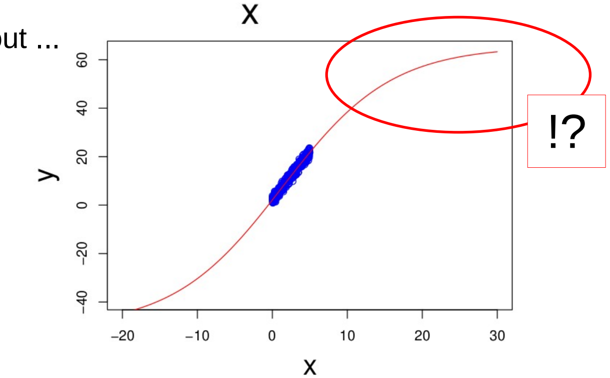
Fitted with **nnet()** function:
$\alpha_1 = -0.19$
$\alpha_2 = 0.14$
$\alpha_3 = 115.93$

!?

Look at predictions



Zoom out ...



Activation functions

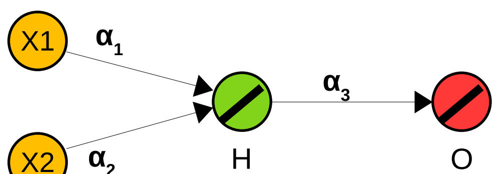
- Activation functions transform the input of a neuron
- Generally, activation functions in the hidden layer are non-linear (e.g. logistic or step function)

Activation function **g()**

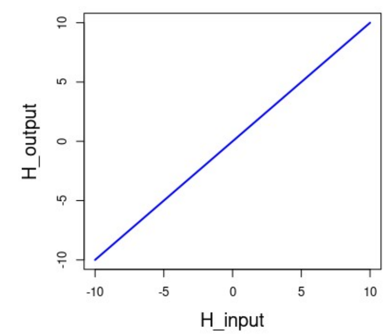
$$H_{Input} = \alpha_1 * X1 + \alpha_2 * X2$$

$$H_{Output} = \mathbf{g(}\ H_{Input}\ \mathbf{)}$$

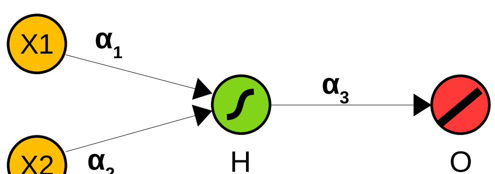
  - **Identity** function:



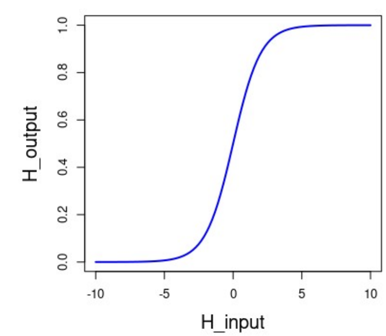
$$H_{output} = g(H_{input}) = H_{input}$$



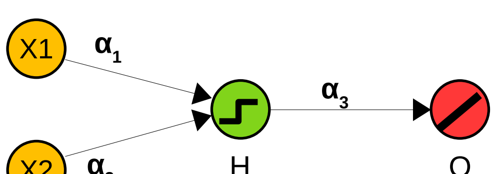
  - **Logistic** function:



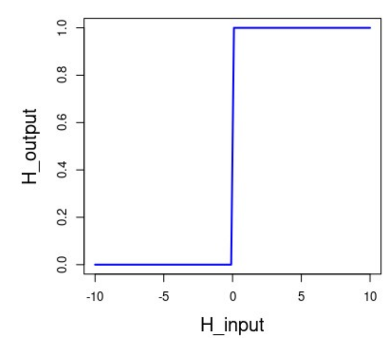
$$H_{output} = g(H_{input}) = \frac{1}{1+e^{-H_{input}}}$$



  - **Step** function:

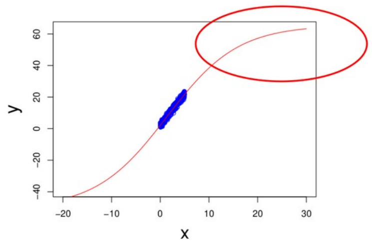


$$H_{output} = g(H_{input}) = \begin{cases} 1 & \text{if } H_{input} > 0 \\ 0 & \text{if } H_{input} \leq 0 \end{cases}$$



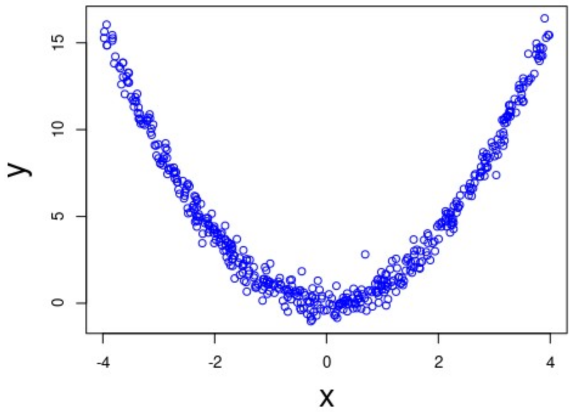
Why non-linear activation functions?

- Only with non-linear activation functions can we model **non-linear patterns**!
- If only the identity function is used the output will **always be a linear function** (no matter how complex/deep the NN is)



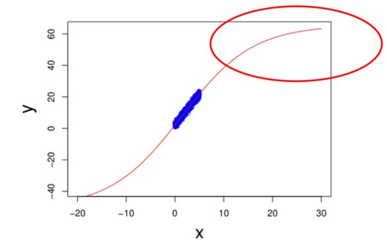
- **Example**: Non-linear univariate regression
  - $y = x^2 + \varepsilon \quad (\varepsilon \sim N(0,\sigma^2))$

Data | **Neural Network** | **Prediction**



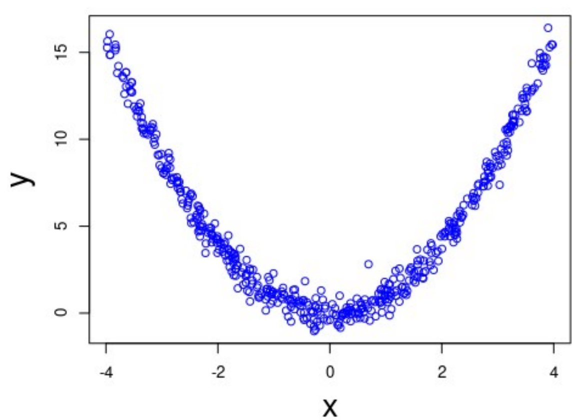
Why non-linear activation functions?

- Only with non-linear activation functions can we model **non-linear patterns**!

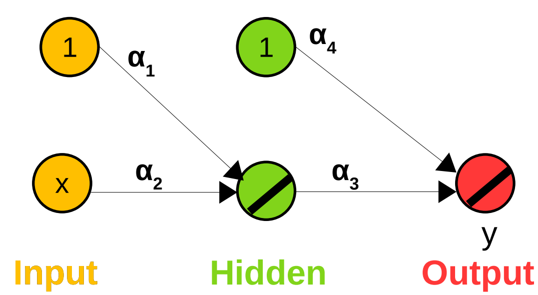


- If only the identity function is used the output will **always be a linear function** (no matter how complex/deep the NN is)
- **Example**: Non-linear univariate regression
  - $y = x^2 + \varepsilon \quad (\varepsilon \sim N(0,\sigma^2))$

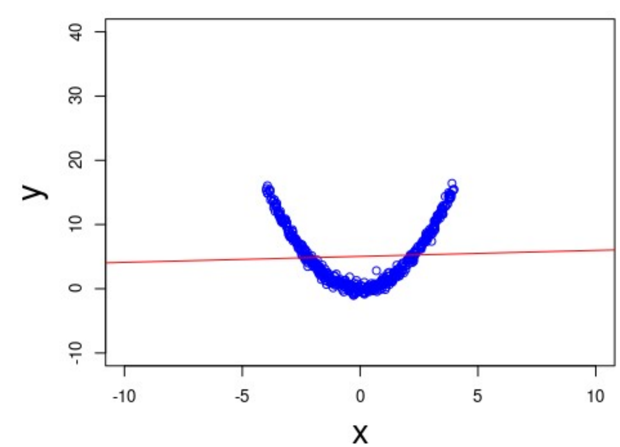
Data



Neural Network

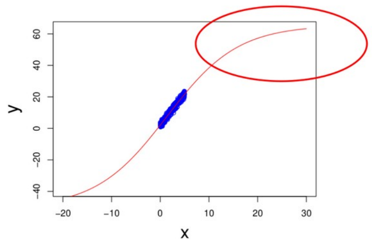


Prediction



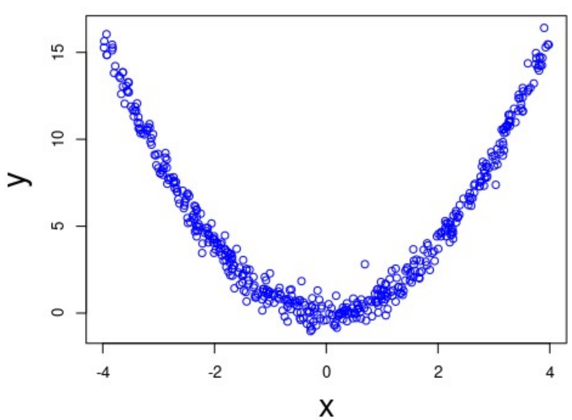
Why non-linear activation functions?

- Only with non-linear activation functions can we model **non-linear patterns**!
- If only the identity function is used the output will **always be a linear function** (no matter how complex/deep the NN is)

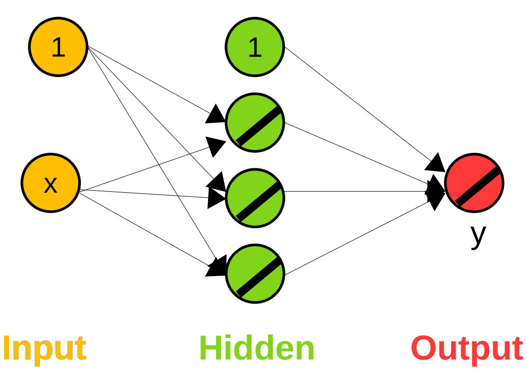


- **Example**: Non-linear univariate regression
  - $y = x^2 + \varepsilon \quad (\varepsilon \sim N(0,\sigma^2))$

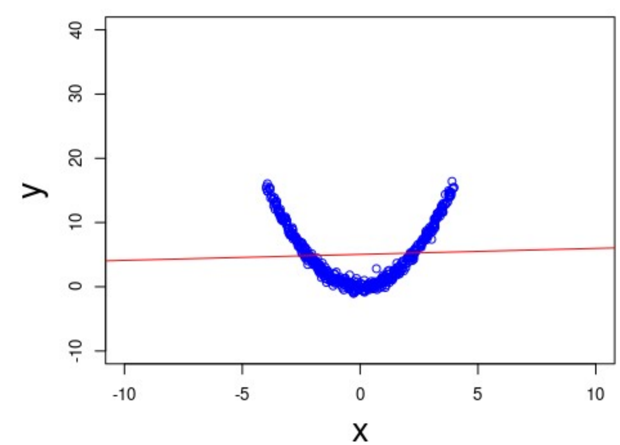
Data



Neural Network



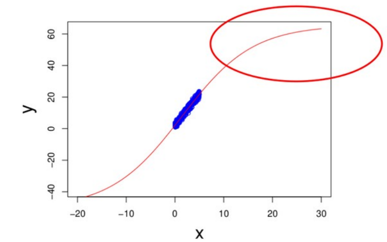
Prediction



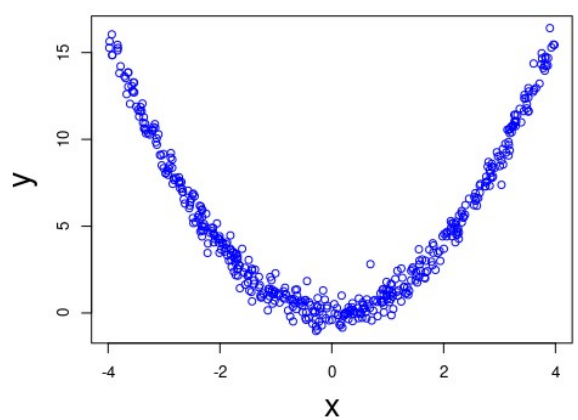
Still just a line!

Why non-linear activation functions?

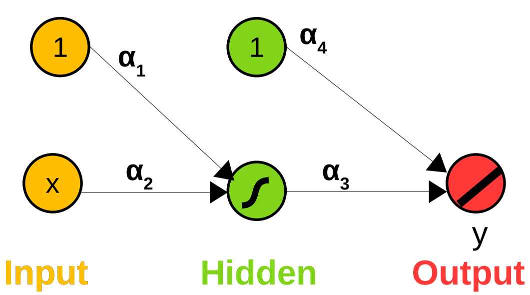
- Only with non-linear activation functions can we model **non-linear patterns**!
- If only the identity function is used the output will **always be a linear function** (no matter how complex/deep the NN is)
- **Example**: Non-linear univariate regression
  - $y = x^2 + \varepsilon \quad (\varepsilon \sim N(0,\sigma^2))$



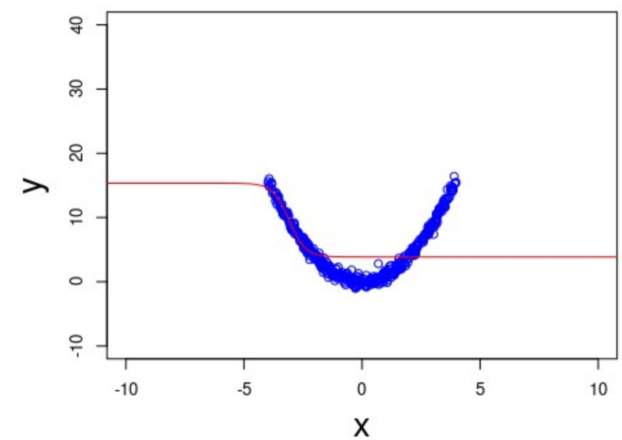
Data



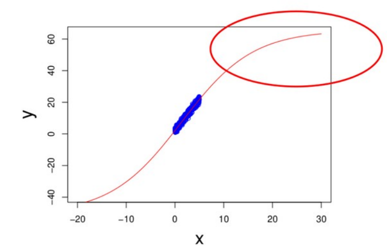
Neural Network



Prediction

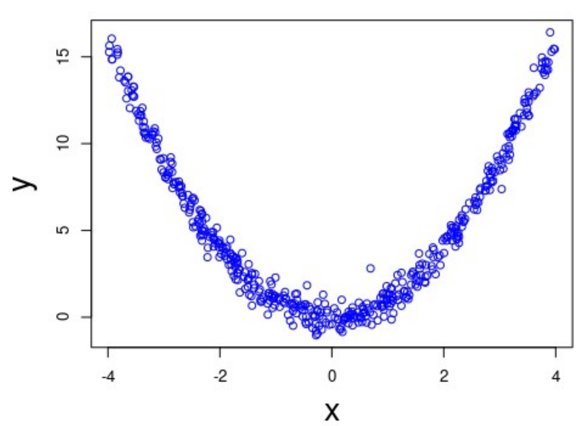


Why non-linear activation functions?

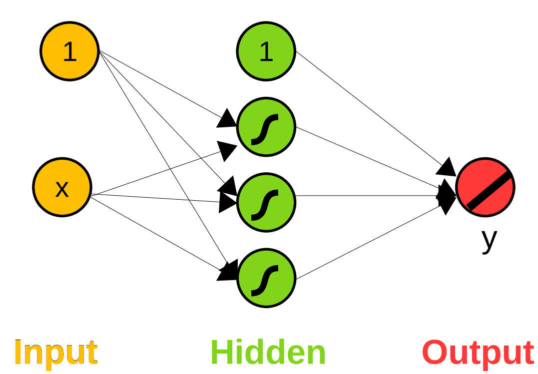


- Only with non-linear activation functions can we model **non-linear patterns**!
- If only the identity function is used the output will **always be a linear function** (no matter how complex/deep the NN is)
- **Example**: Non-linear univariate regression
  - $y = x^2 + \varepsilon \quad (\varepsilon \sim N(0,\sigma^2))$

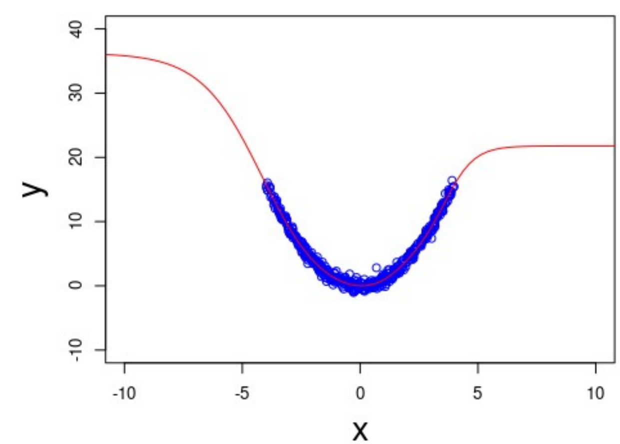
Data



Neural Network

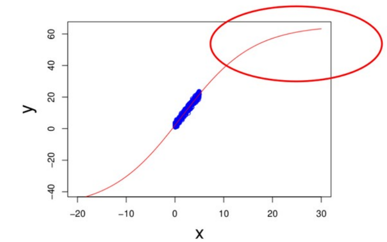


Prediction

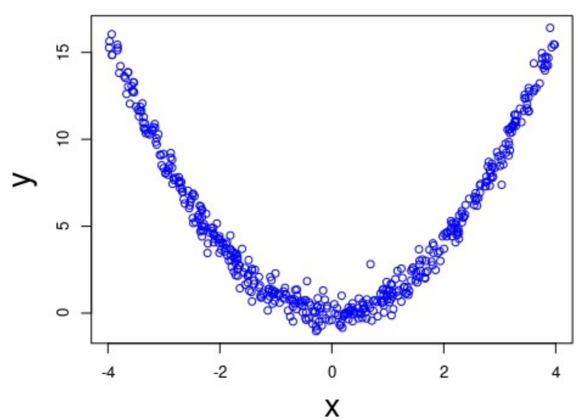


Why non-linear activation functions?

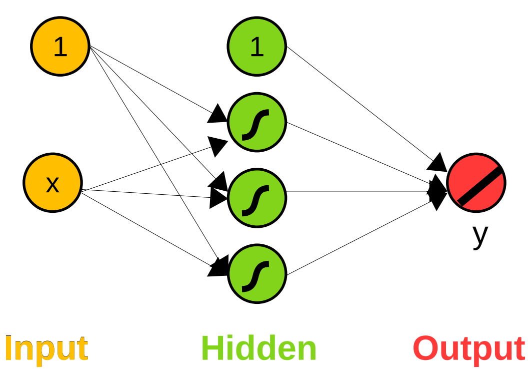
- Only with non-linear activation functions can we model **non-linear patterns**!
- If only the identity function is used the output will **always be a linear function** (no matter how complex/deep the NN is)
- **Example**: Non-linear univariate regression
  - $y = x^2 + \varepsilon \quad (\varepsilon \sim N(0,\sigma^2))$



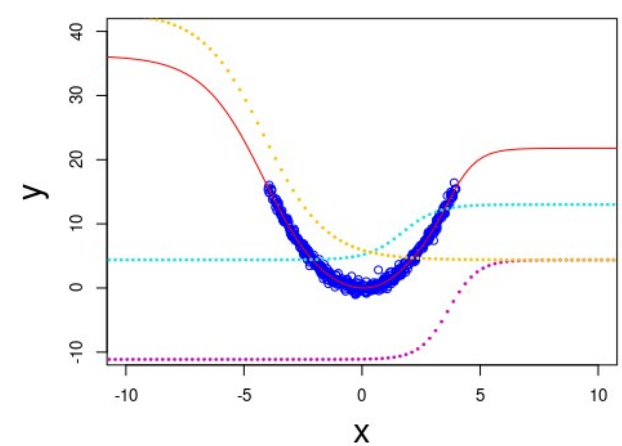
Data



Neural Network



Prediction

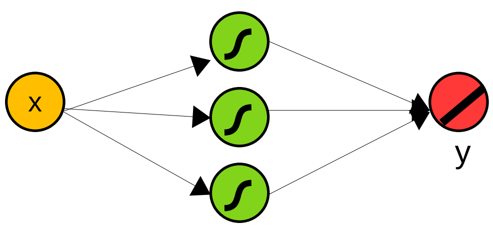


Dashed lines show the individual contributions to the prediction of the three hidden nodes.

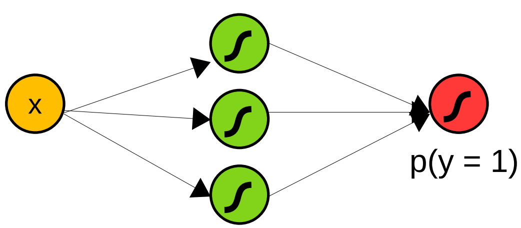
Neural Networks for classification

- The activation function and/or size in the output layer can be adapted to use NN for classification
- **Regression**
  - Can use the identity function in the output layer
- **Binary classification**
  - Can e.g. use the logistic function in the output layer
  - The value of the output node represents the probability of y being equal to class 1
- **Multiclass classification**
  - Could use k output nodes for k possible classes and e.g. a logistic function
  - The k output nodes represent the probabilities that y is equal to a certain class
  - Classically, the probabilities of the k output nodes are forced to sum up to 1 (**softmax function**)

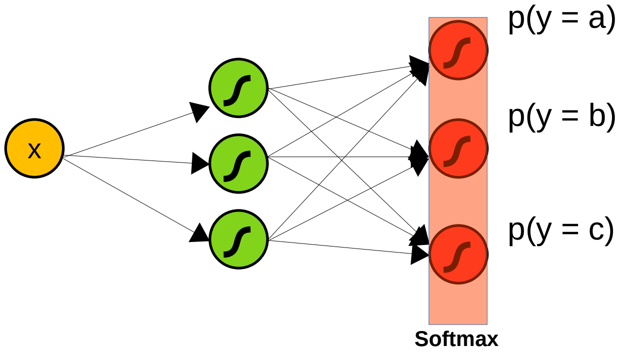
NN for regression:



NN for binary class.:

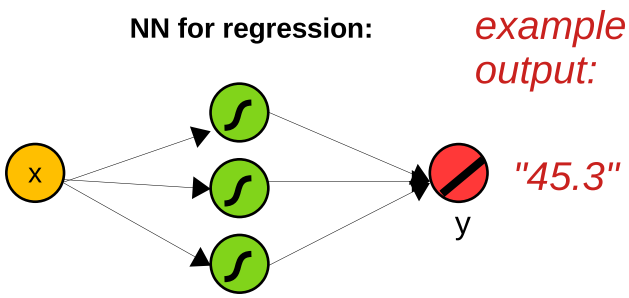


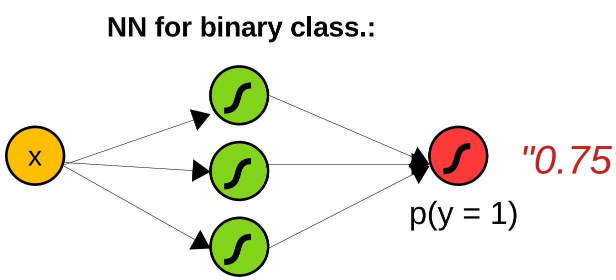
NN for multiclass class.:

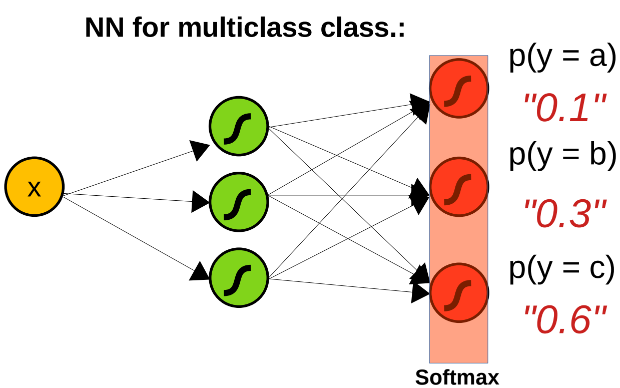


Neural Networks for classification

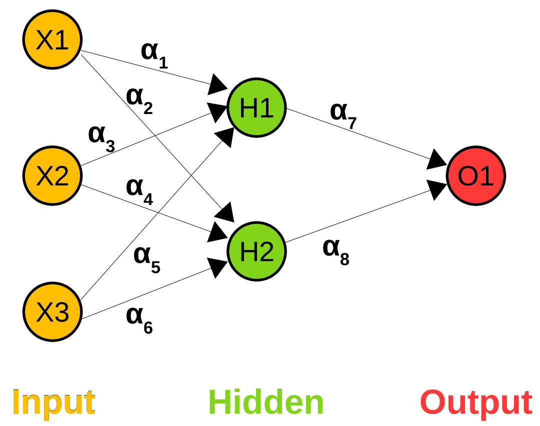
- The activation function and/or size in the output layer can be adapted to use NN for classification
- **Regression**
  - Can use the identity function in the output layer
- **Binary classification**
  - Can e.g. use the logistic function in the output layer
  - The value of the output node represents the probability of y being equal to class 1
- **Multiclass classification**
  - Could use k output nodes for k possible classes and e.g. a logistic function
  - The k output nodes represent the probabilities that y is equal to a certain class
  - Classically, the probabilities of the k output nodes are forced to sum up to 1 (**softmax function**)







How are the weights found?

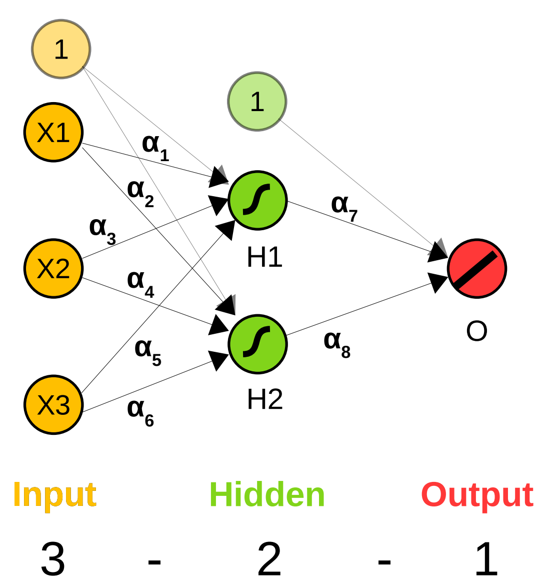


- "Training" of the Network
  - Start with **random weights** (initialization)
  - The predictions will be random as well (bad)
- Compare predictions with true y-values (classically using batches of training data)
  - Calculate how "wrong" the predictions were (**Loss-function**)
- Find out how the weights have to be shifted to improve the predictions (**backpropagation** algorithm)
- Continue to update the weights until the algorithm converges
- Procedure is similar to **gradient descent**

Single hidden layer NN in R

- Single hidden layer NNs can be fitted with **nnet** R-package

```
> library(nnet)
> nn1 <- nnet(Sepal.Length ~ scale(Sepal.Width) + scale(Petal.Length) + scale(Petal.Width),
            data = iris,
            size=2,
            decay=0,
            linout=TRUE,
            maxit=10000)
> summary(nn1)
a 3-2-1 network with 11 weights
options were - linear output units
 b->h1 i1->h1 i2->h1 i3->h1
 60.48  38.70 -20.88 -22.81
 b->h2 i1->h2 i2->h2 i3->h2
 -1.16   0.12   0.13  -0.08
  b->o  h1->o  h2->o
 -2.12  -0.53  21.18

# Make predictions (here for training data):
> predict(nn1, newdata=iris)
```



Single hidden layer NN in R

- Single hidden layer NNs can be fitted with **nnet** R-package

For neural networks it is usually advisable to standardize the predictors!

```
> library(nnet)
> nn1 <- nnet(Sepal.Length ~ scale(Sepal.Width) + scale(Petal.Length) + scale(Petal.Width),
           data = iris,
           size=2,
           decay=0,
           linout=TRUE,
           maxit=10000)
> summary(nn1)
a 3-2-1 network with 11 weights
options were - linear output units
 b->h1 i1->h1 i2->h1 i3->h1
60.48  38.70 -20.88 -22.81
 b->h2 i1->h2 i2->h2 i3->h2
-1.16   0.12   0.13  -0.08
  b->o  h1->o  h2->o
-2.12  -0.53  21.18
```

Number of nodes in hidden layer

"Weight decay" regularization (later)

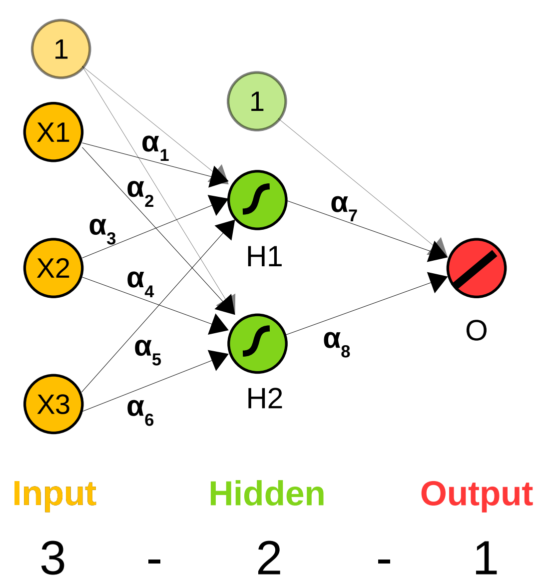
Use identity activation function for output layer (default is logistic)

Number of maximum iterations (how many iterations to adjust weights)

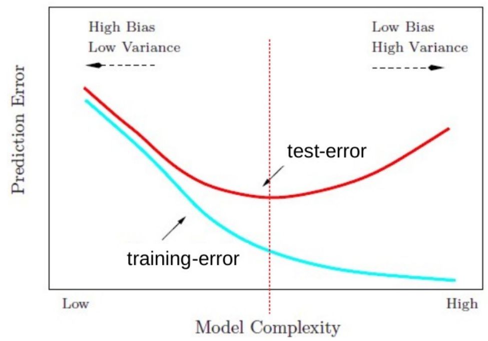
For classification use:

`predict(..., type='class')`

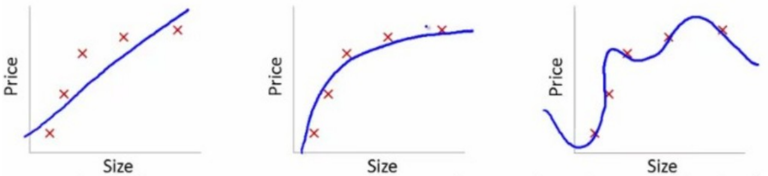
```
# Make predictions (here for training data):
> predict(nn1, newdata=iris)
```



Parameter tuning in NNs



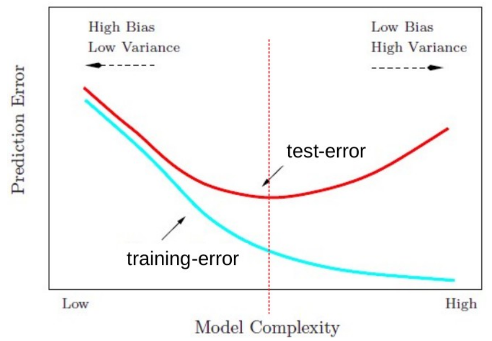
- With the **nnet()** function two main tuning parameter exists
  - "**size**" (number of neurons in hidden layer)
  - "**decay**" (Regularization factor using weight decay)
- The more neurons in the hidden layer the more complex patterns can be modeled
  - Danger of **under/over-fitting**!



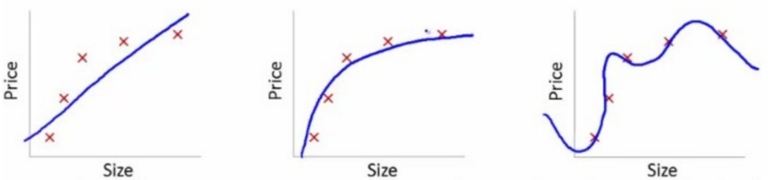
under-fitting **"Just right"** **over-fitting**

- Overfitting on the example of the iris data set:
  - Training error (RMSE) for different NN sizes

Parameter tuning in NNs



- With the **nnet()** function two main tuning parameter exists
  - "**size**" (number of neurons in hidden layer)
  - "**decay**" (Regularization factor using weight decay)
- The more neurons in the hidden layer the more complex patterns can be modeled
  - Danger of **under/over-fitting**!



under-fitting **"Just right"** **over-fitting**

- Overfitting on the example of the iris data set:
  - Training error (RMSE) for different NN sizes

```
> nnet(Sepal.Length ~.,
iris, size=1,
linout=TRUE,
maxit=10000)
```

Training error:
0.681

Under-fitting?

```
> nnet(Sepal.Length ~.,
iris, size=5,
linout=TRUE,
maxit=10000)
```

Training error:
0.071

Good?

```
> nnet(Sepal.Length ~.,
iris, size=20,
linout=TRUE,
maxit=10000)
```

Training error:
0.011

Over-fitting?

Can try to find optimum with **crossvalidation**

Parameter tuning in NNs (caret)

- **Caret** is an R-package to automatically tune various machine learning models
- Allows the tuning of neural networks from **nnet** and **neuralnet** R-packages
- Have to define a **search grid** of parameter values
  - For each setting caret fits a NN and estimates the **test-error** using a resampling-based estimation (similar to crossvalidation)
  - The model with the lowest test-error is selected as the winner

```
> library(caret)
### Create tuning grid:
> t.grid <- expand.grid(size=c(1,5,10),
                        decay=c(0, 0.5))
> t.grid
  size decay
1    1   0.0
2    5   0.0
3   10   0.0
4    1   0.5
5    5   0.5
6   10   0.5

### Tune the model ("train" function):
> tune.caret <- train(Sepal.Length ~ ., data=iris,
      method='nnet', tuneGrid=t.grid, maxit=10000,
      linout=TRUE, preProcess=c("center","scale"))
```

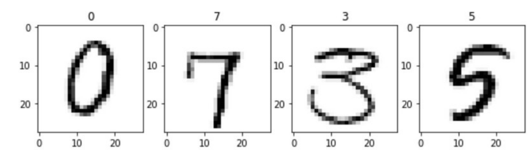
Parameter tuning in NNs (caret)

- **Caret** is an R-package to automatically tune various machine learning models
- Allows the tuning of neural networks from **nnet** and **neuralnet** R-packages
- Have to define a **search grid** of parameter values
  - For each setting caret fits a NN and estimates the **test-error** using a resampling-based estimation (similar to crossvalidation)
  - The model with the lowest test-error is selected as the winner

```
### Tune the model ("train" function):
> tune.caret <- train(Sepal.Length ~ ., data=iris,
      method='nnet', tuneGrid=t.grid, maxit=10000,
      linout=TRUE, preProcess=c("center","scale"))

> tune.caret
Neural Network
  150 samples
  4 predictor

Pre-processing: centered (5), scaled (5)
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 150, 150, 150, 150, 150, 150, ...
Resampling results across tuning parameters:

  size  decay  RMSE       Rsquared   MAE
   1    0.0    0.4912394  0.6274869  0.3949317
   1    0.5    0.3449973  0.8313345  0.2803835
   5    0.0    0.5092789  0.7008530  0.3628299
   5    0.5    0.3321719  0.8435153  0.2730387
  10    0.0    2.3660282  0.3400224  0.8798027
  10    0.5    0.3301436  0.8449822  0.2705284

RMSE was used to select the optimal model using the smallest
value. The final values used for the model were size = 10
and decay = 0.5.
```

Picture recognition with NNs

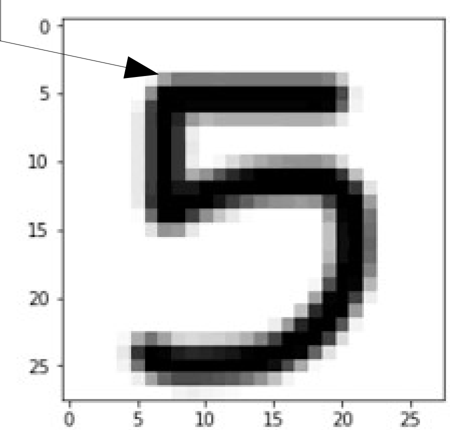
- So far in supervised learning: Data is always a **table** with **predictors** and **target variable**
- What if we want to predict the content shown on a **picture**?
  - A picture is no different! We can translate it into a row of values where each value represents the shade of one **pixel**.
- Example: Predict/recognize **handwritten digits**



…

  - Examplary data table of pictures with 30 x 30 pixels:

The shading of each pixel corresponds to a "**grayscale**" value ranging from 0 (white) to 255 (black).



	Pix1	Pix2	Pix3	…	Pix900	**Digit**
Picture 1	20	24	60	…	44	**0**
Picture 2	10	94	160	…	244	**7**
Picture 3	220	89	143	…	134	**3**
Picture 4	12	123	70	…	230	**5**
…	…	…	…	…	…	…

Grayscale values of all 900 pixels on a picture

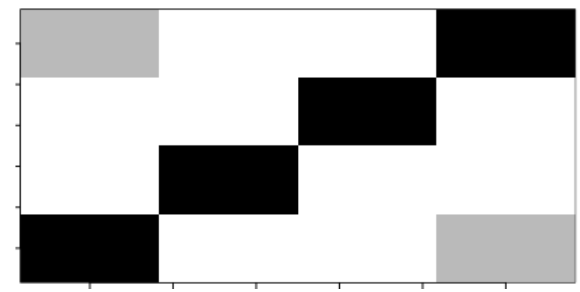
Target variable

Picture recognition with NNs

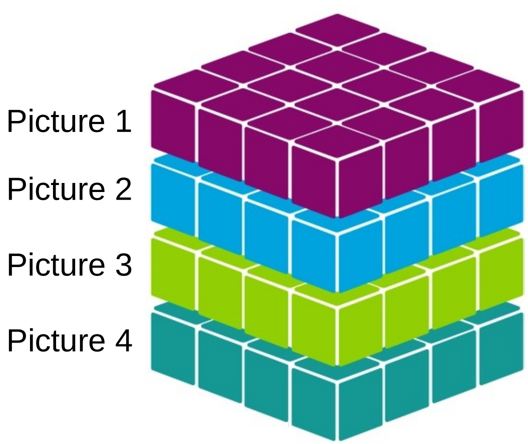
- Grayscale images are normally stored as tables with rows and columns indicating the **pixel positions**
  - Table storing a grayscale picture with 4 x 4 pixels:

100	0	0	255
0	0	255	0
0	255	0	0
255	0	0	100

$\xrightarrow{\text{Is the image}}$



- A collection of multiple images is normally stored as a **3-dimensional array,** which is like a cube with pictures "stacked" as layers
  - Exemplary array storing four 4x4 pixel images:



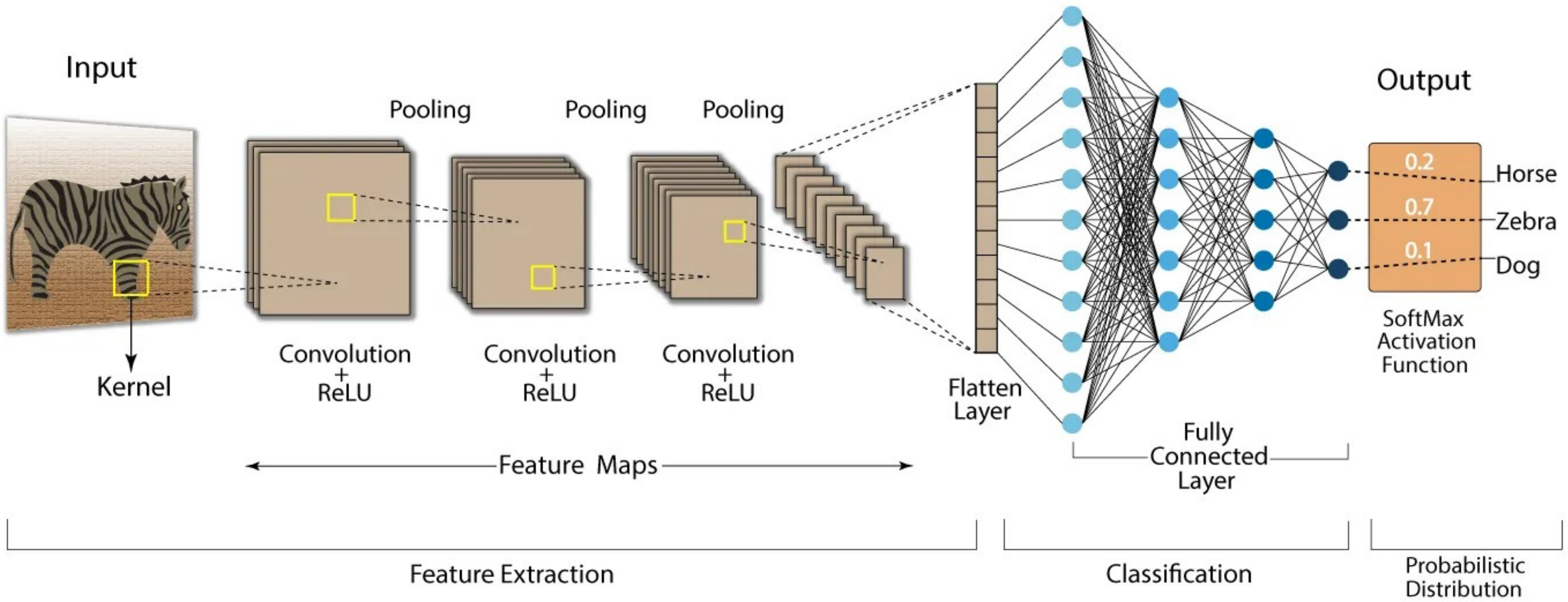
We have to "**flatten**" the images so that we can feed them to a classifier like a NN.

	Pix1	Pix2	Pix3	...	Pix16
Picture 1	20	24	60	...	44
Picture 2	10	94	160	...	244
Picture 3	220	89	143	...	134
Picture 4	12	123	70	...	230

(table-format we need)

Picture recognition with NNs

- Feeding a "flattened" image into a classification NN works for smaller picture sizes
- For large pictures, however, the NN becomes too complex and complicated (millions of parameters)
- **Convolutional NNs** try to more efficiently capture the spatial dependencies in a picture by reducing the image to a set of (hopefully) relevant features (**feature extraction**)
  - The extracted features are subsequently fed into a classification NN



Deep learning with R (book)

- Good introduction to deep neural networks with R (by François Chollet and J.J. Allaire)