UNIVERSITÄT DUISBURG ESSEN

Offen im Denken

Grundlagen des maschinellen Lernens

Neuronale Netze I

V1.0 — 07.12.2020

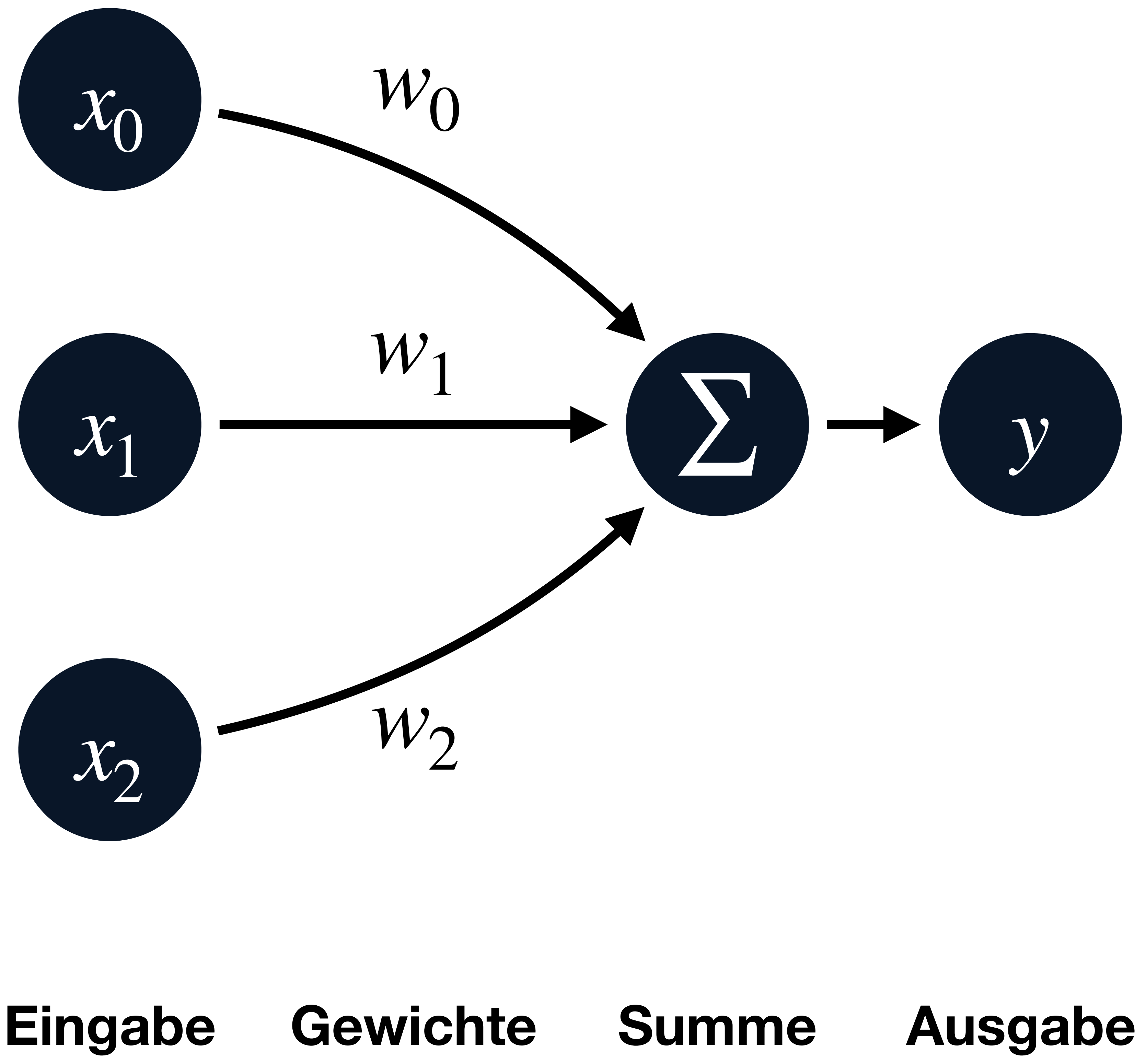
Ole Meyer

Lehrstuhl für Software Engineering

Perceptrons

Der Grundbaustein für Neuronale Netze

Perceptron



$$y = X^T W$$

```
x=torch.FloatTensor([.1,.2,-.3])
w=torch.FloatTensor([.2,.3,.4])

y=(x*w).sum()

-> -0.0400
```

Ausflug: Matrixmultiplikation

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{21} \\ c_{21} & c_{22} \end{bmatrix}$$

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{21} \\ c_{21} & c_{22} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{21} \\ c_{21} & c_{22} \end{bmatrix}$$

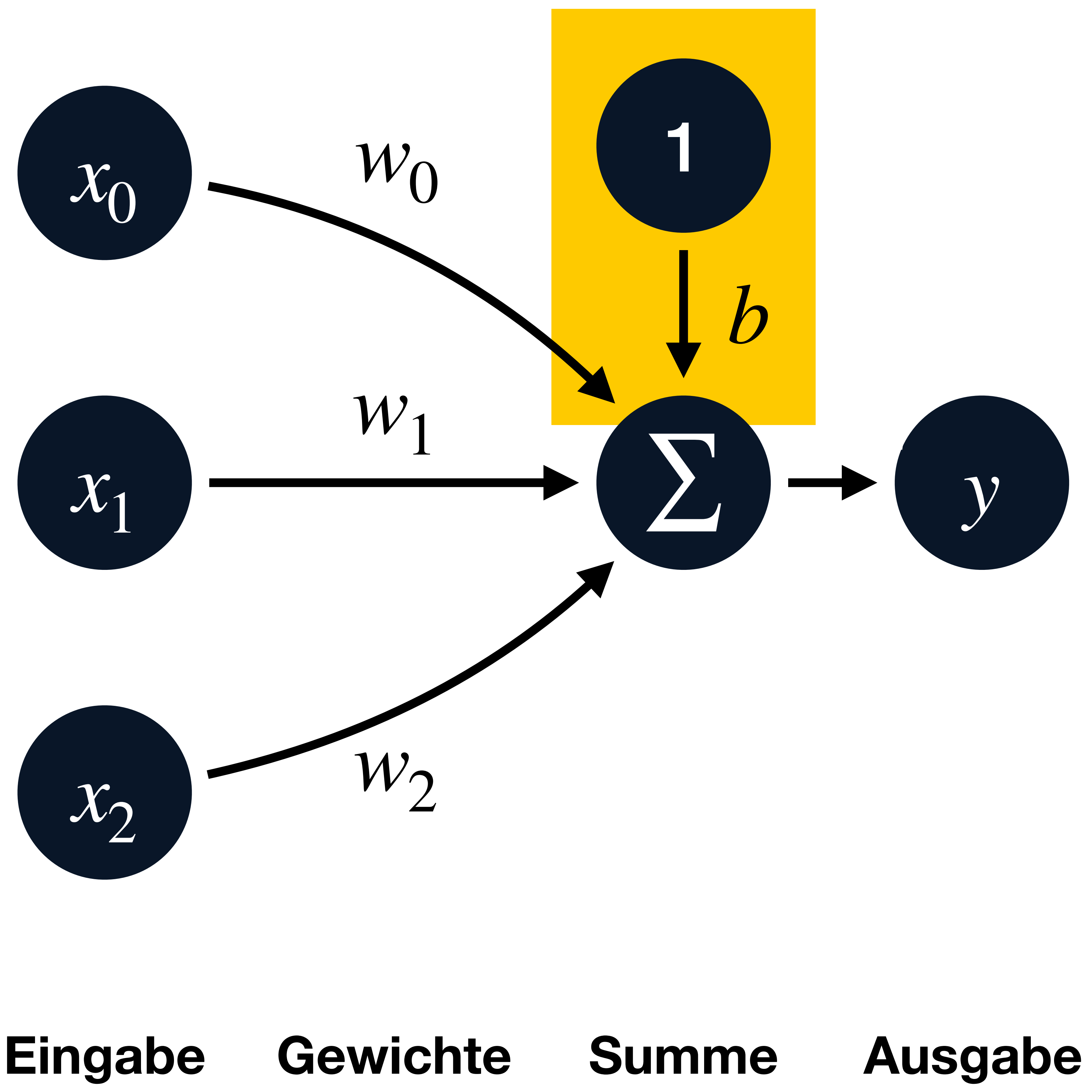
- Es kann vorkommen, dass die mathematische Schreibweise auf den ersten Blick von der Programmierung abweicht.
- Sehr wichtig: Verstehen und Nachvollziehen was genau passiert.
- Die Addition, welche im Programmcode explizit zu sehen ist, kann sich z.B. implizit in der Matrixmultiplikation verstecken.

$$y = X^T W$$

ist gleich

```
x=torch.FloatTensor([.1,.2,-.3])
w=torch.FloatTensor([.2,.3,.4])

y=(x*w).sum()
```

Perceptron: Bias

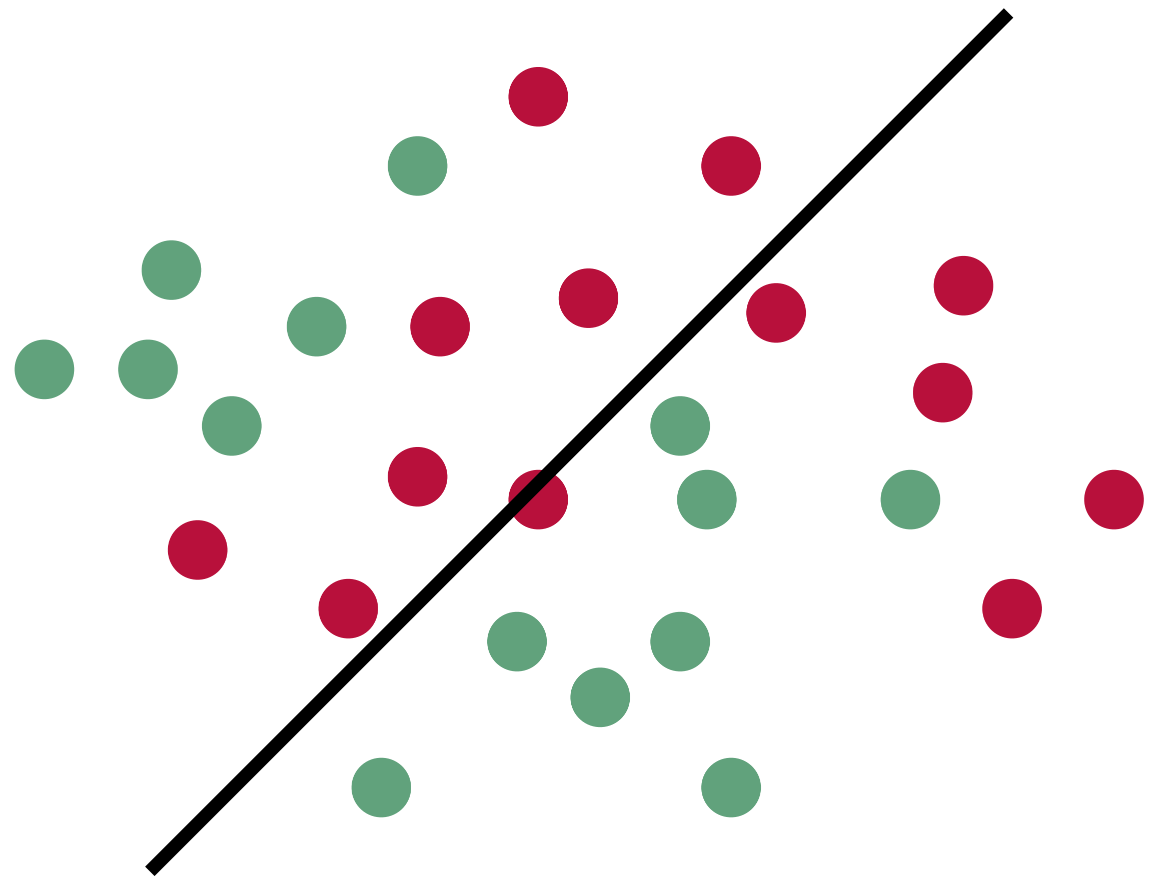


$$y = X^T W + b$$

```
x=torch.FloatTensor([.1,.2,-.3])
w=torch.FloatTensor([.2,.3,.4])
b=torch.FloatTensor([.5])

y=(x*w).sum()+b
-> 0.4600
```

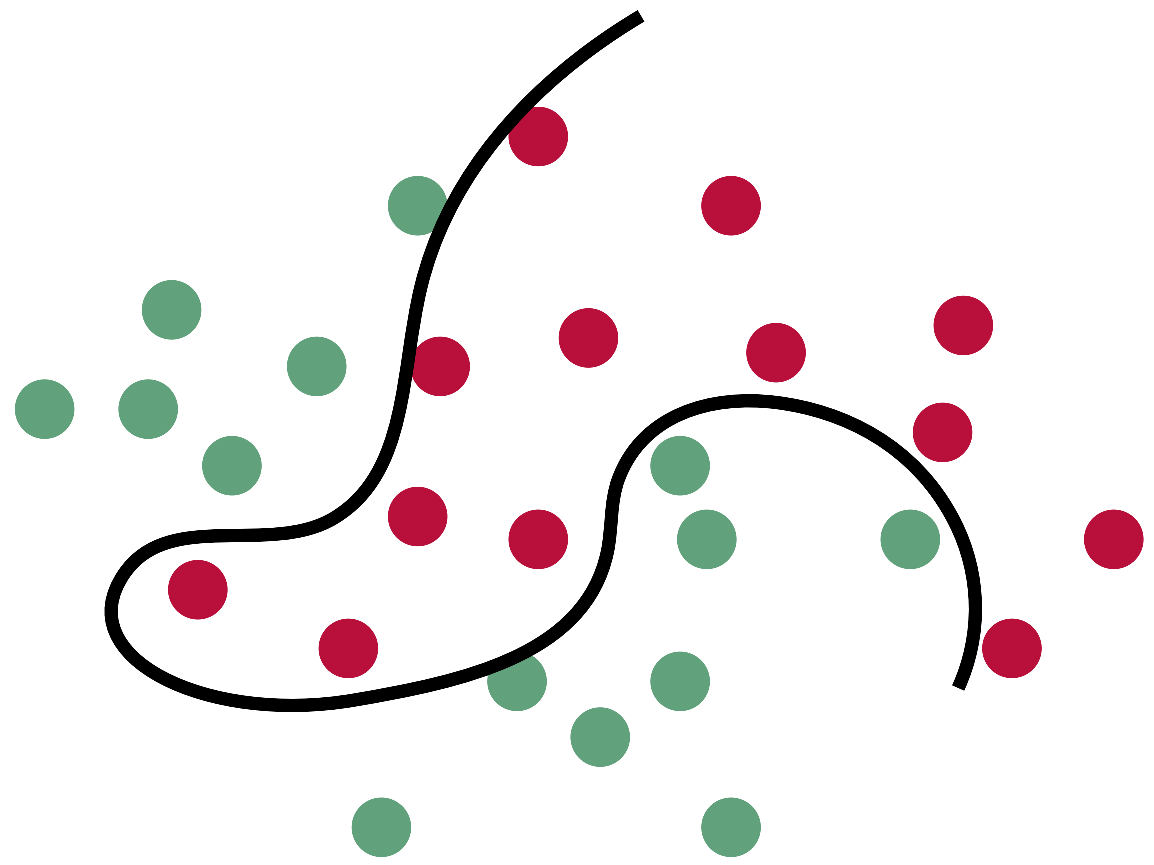
Perceptron: Aktivierungsfunktionen

- Lineare Funktionen erzeugen lineare Entscheidungen
- Für nicht-lineare Entscheidungen werden nicht-lineare Funktionen benötigte
- Lösung: Sogenannte **Aktivierungsfunktionen**

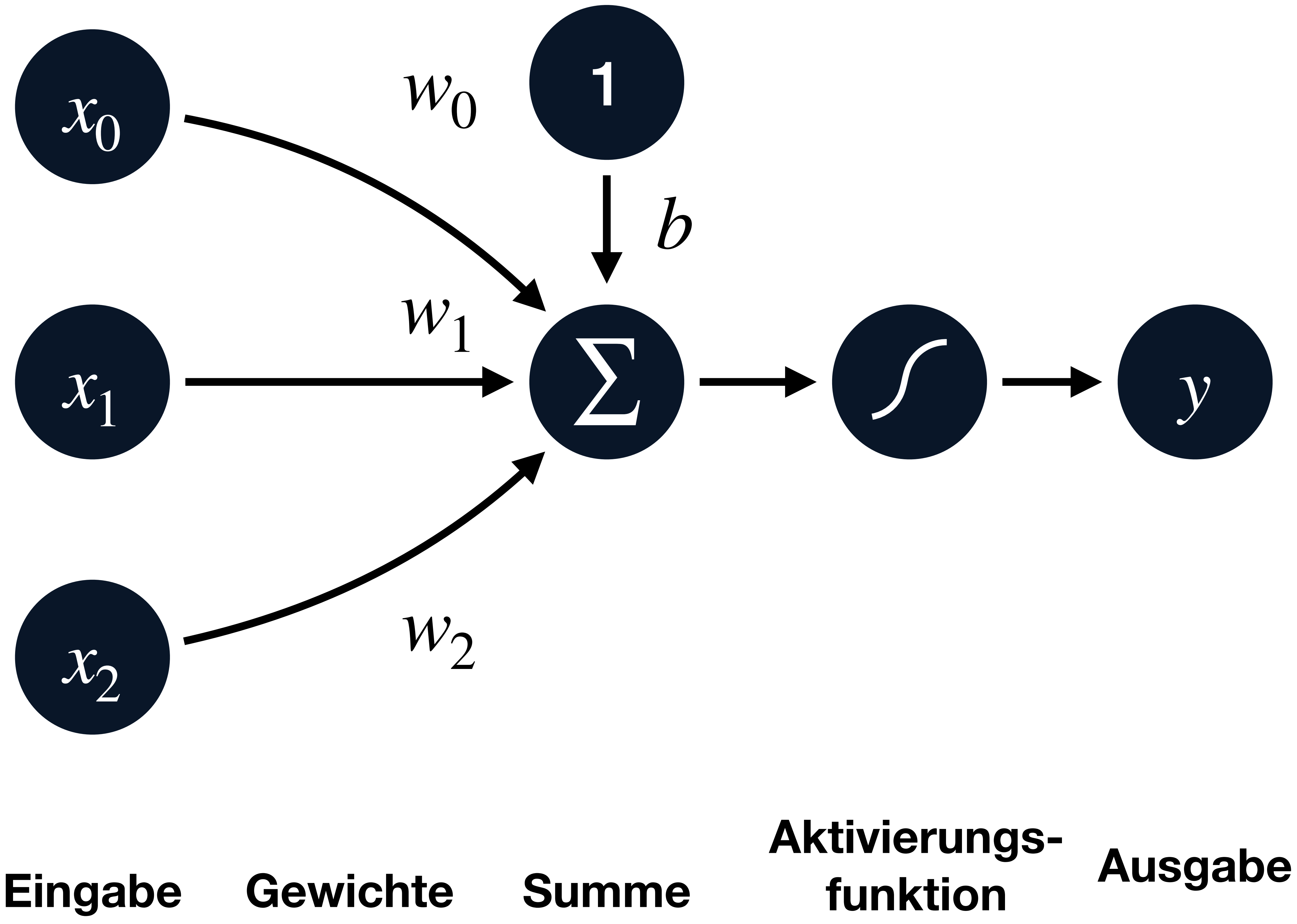


Perceptron: Aktivierungsfunktionen

- **Aktivierungsfunktionen** integrieren „Non-Linearities“
- Dadurch können theoretisch unbegrenzt komplexe Funktionen approximiert werden
  - Voraussetzung: Differenzierbarkeit
- Beispiele für Aktivierungsfunktionen:
  - tanh
  - ReLU
  - ELU
  - Sigmoid



Aktivierungsfunktion



$$y = f(X^T W + b)$$

```
x=torch.FloatTensor([.1,.2,-.3])
w=torch.FloatTensor([.2,.3,.4])
b=torch.FloatTensor([.5])

y=torch.tanh((x*w).sum()+b)


-> 0.4301
```

Übliche Aktivierungsfunktionen

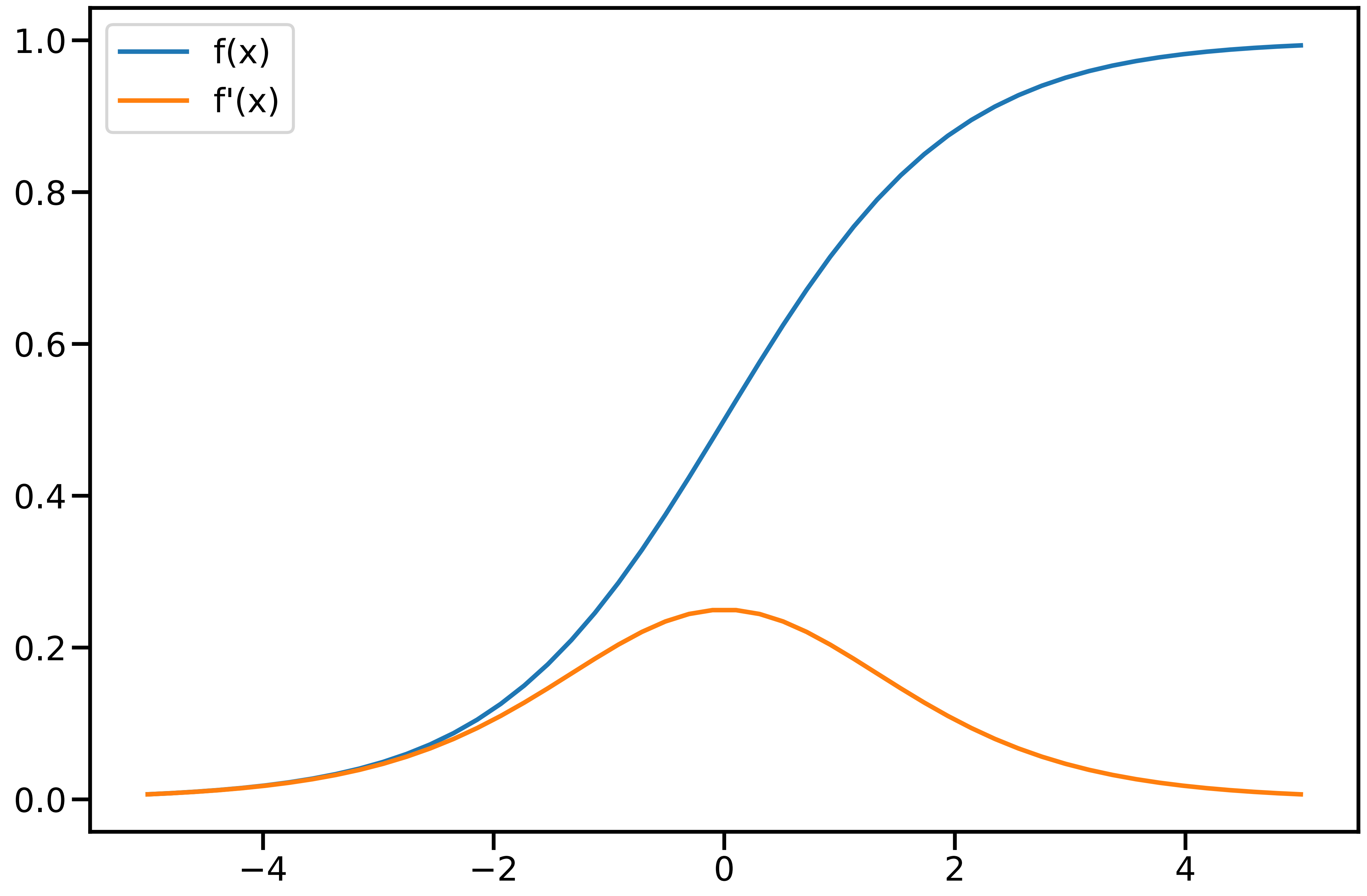
Sigmoid-Funktion

- $f(x) = \dfrac{1}{1 + e^{-x}}$

- $f'(x) = f(x)(1 - f(x))$

- Ausgabe ist immer zwischen 0 und 1

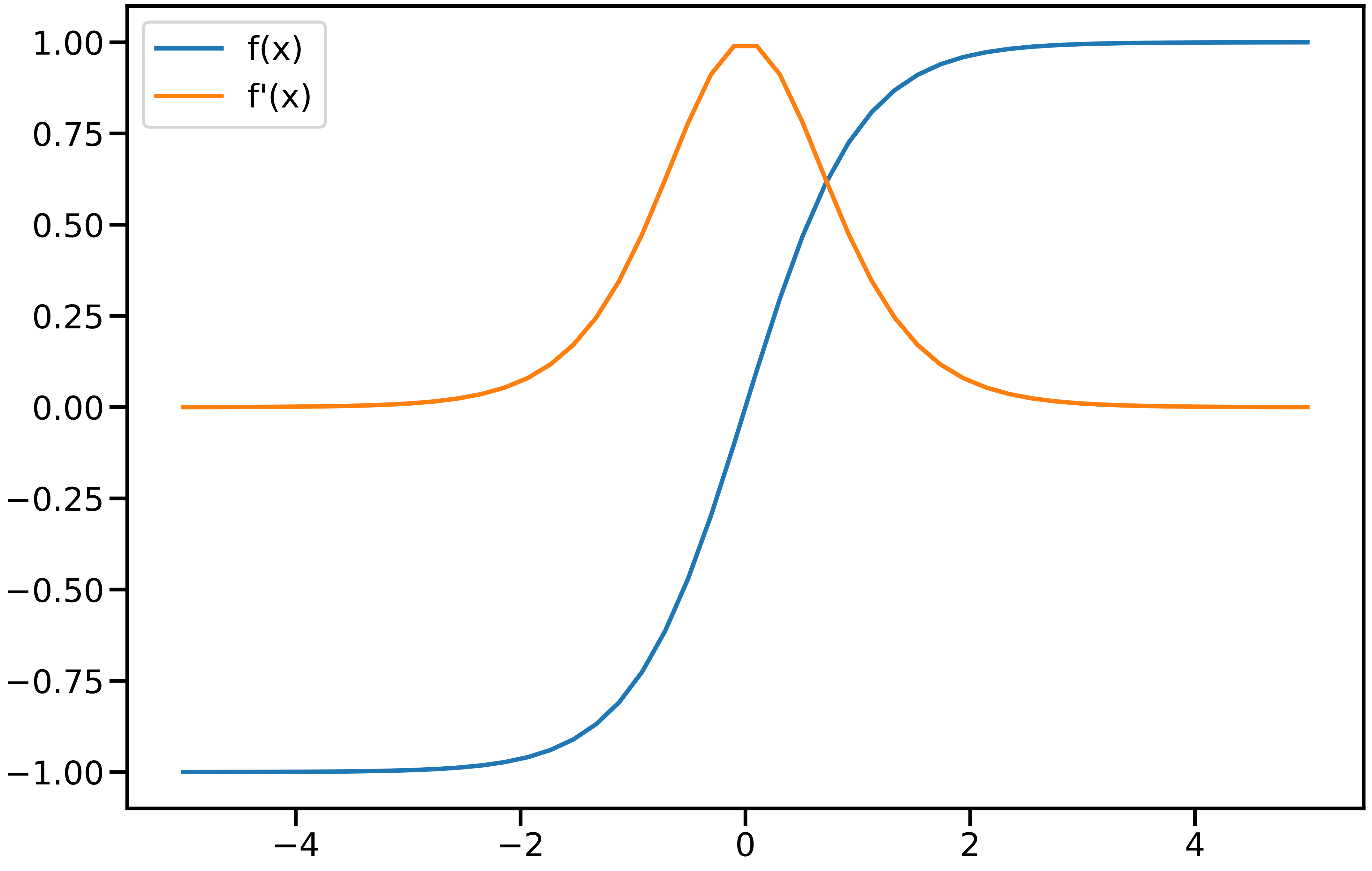
- Nachteil: Die Steigung ist zum Teil sehr klein (hier kommen wir später noch einmal drauf zurück)



Übliche Aktivierungsfunktionen

Tangens Hyberbolicus (TanH)

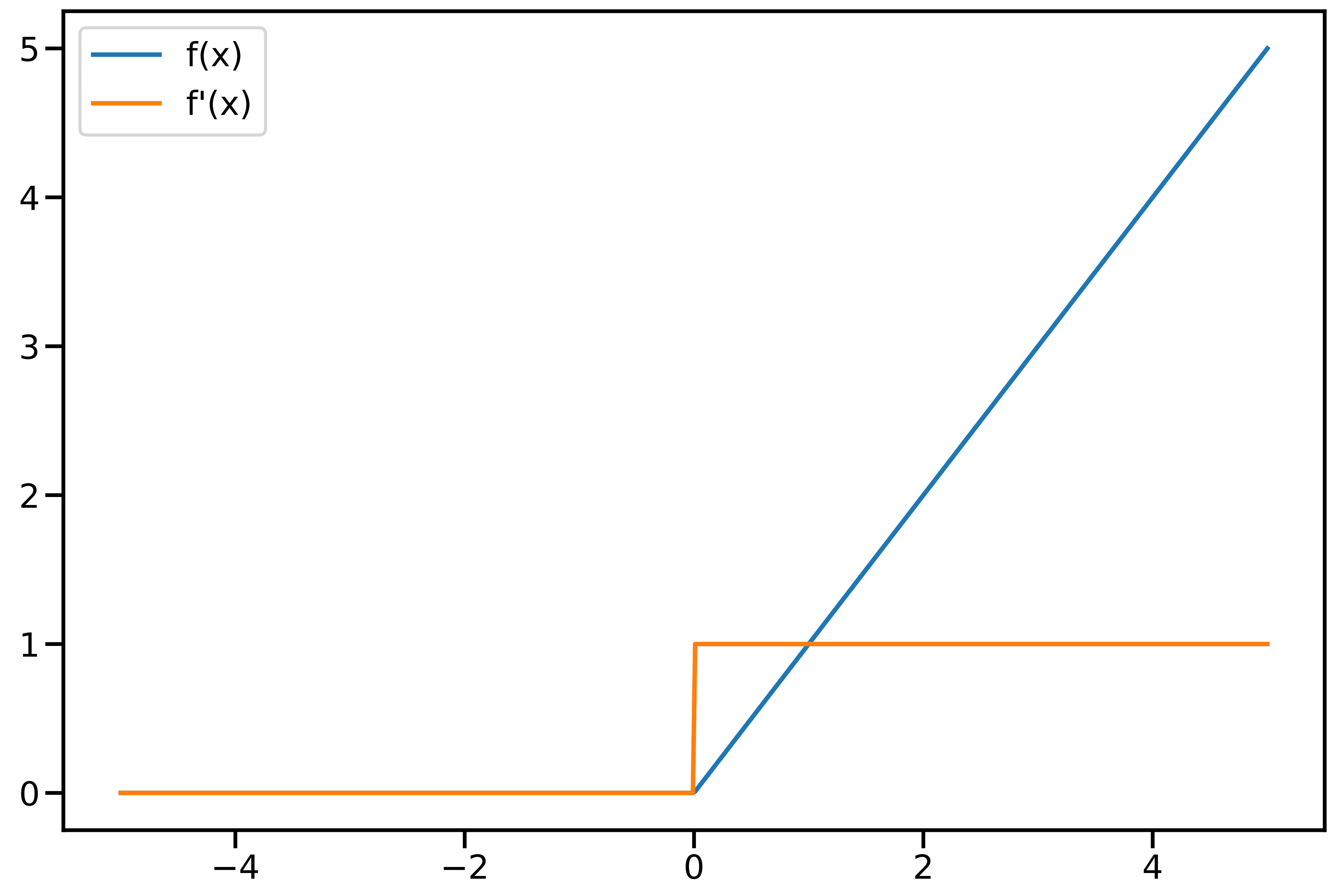
- $f(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$
- $f'(x) = 1 - f(x)^2$
- Ausgabe ist immer zwischen -1 und 1
- Eine skalierte Variante der Sigmoid-Funktion
- Vorteil: Die Steigung der Ableitung fällt z.T. größer aus (hier kommen wir später noch einmal drauf zurück)



Übliche Aktivierungsfunktionen

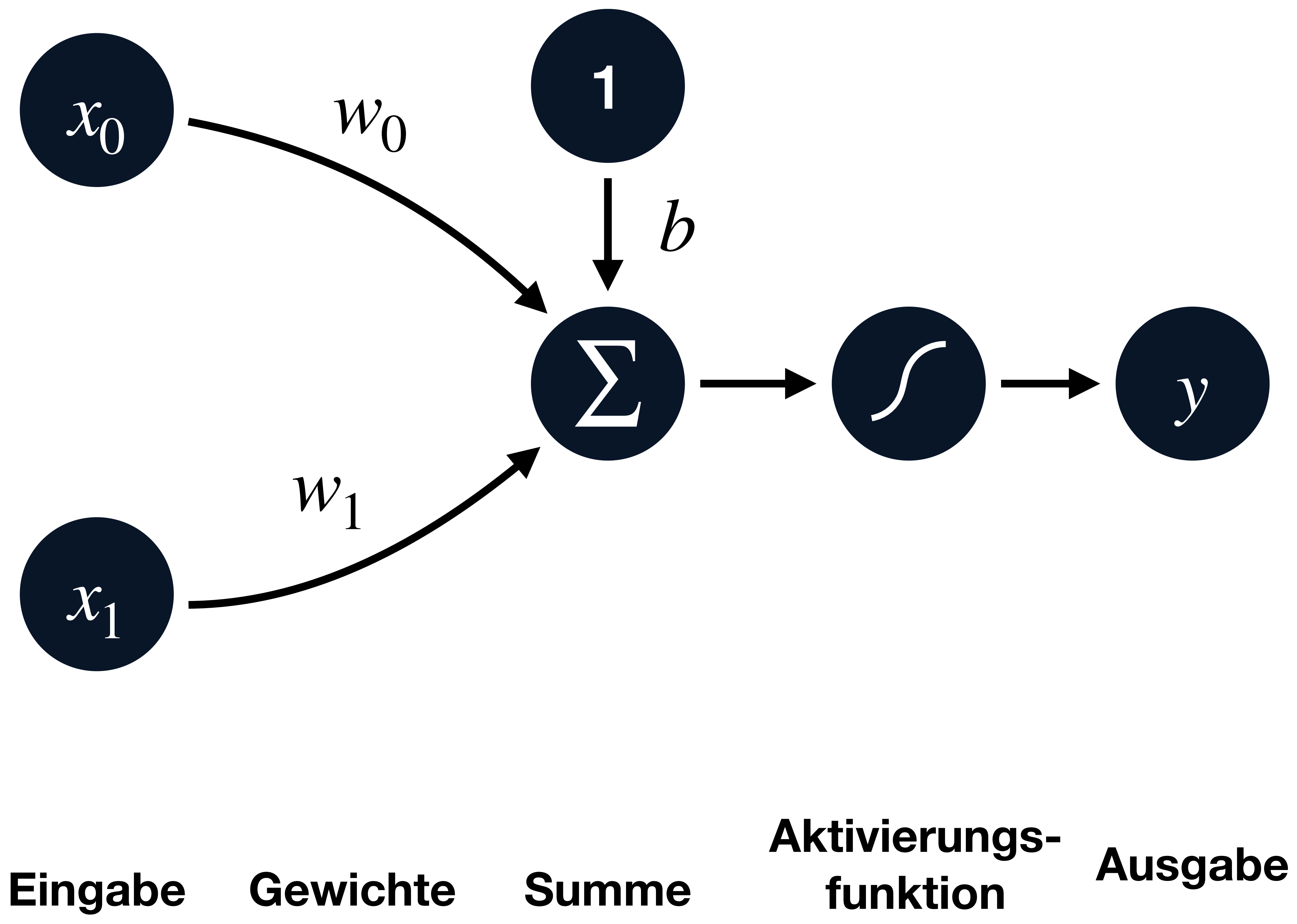
Rectified Linear Unit (ReLU)

- $f(x) = \max(0, x)$
- $f'(x) = \begin{cases} 1 & \text{wenn } x > 0 \\ 0 & \text{sonst} \end{cases}$
- Ausgabe ist immer zwischen 0 und ∞
- Vorteil: Sparsity. Teilbereiche eines Netzes können ausgeschaltet werden.
- Nachteil: ebenfalls Sparsity. Für Werte <0 existiert kein Gradient (wir kommen später darauf zurück).



Perceptron: Durchlauf

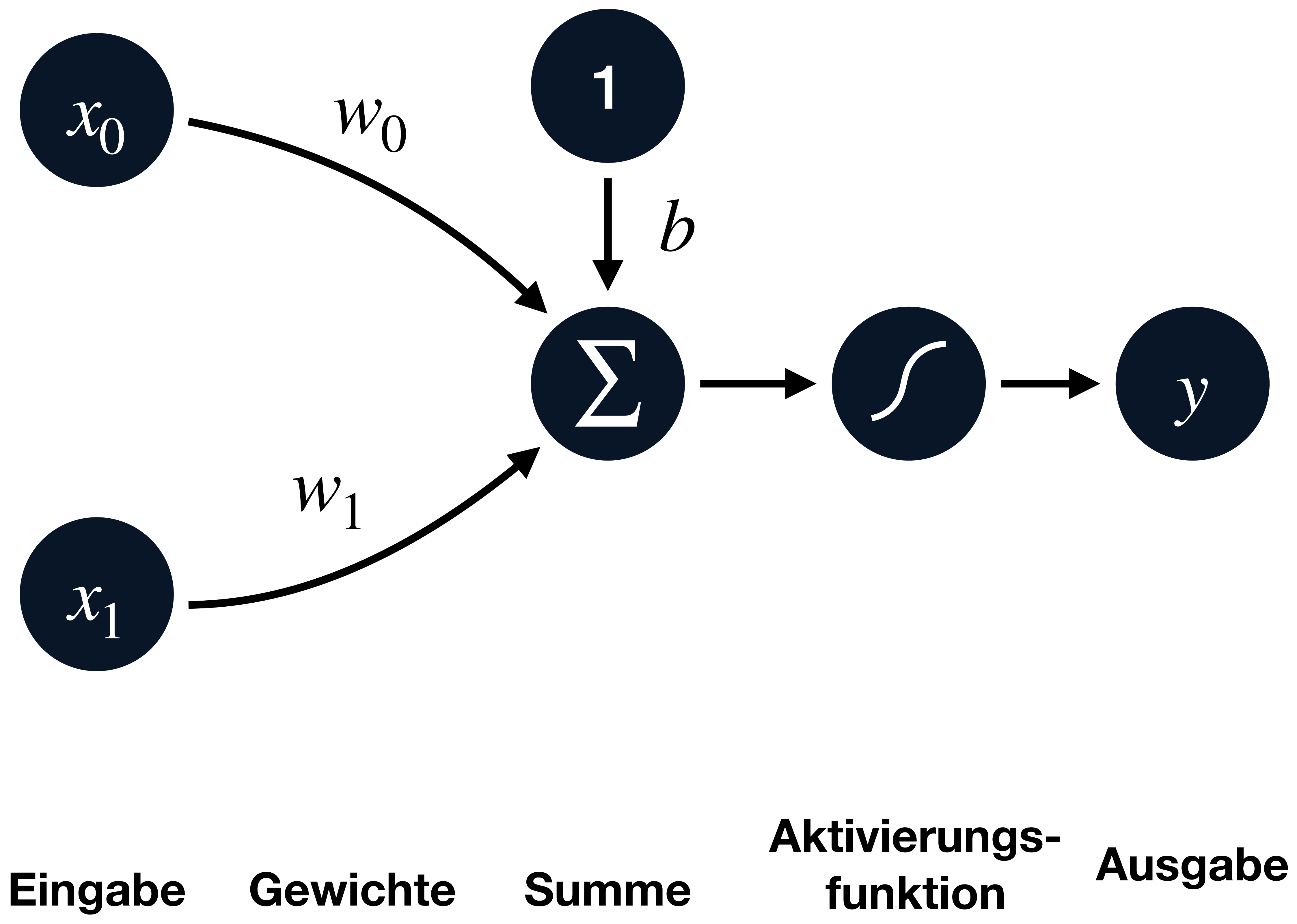
Parameter des Netzwerkes



- Parameter:
  - $W = [3, -2]^T$
  - $b = 1$

Perceptron: Durchlauf

Rechnung (Einsetzen)

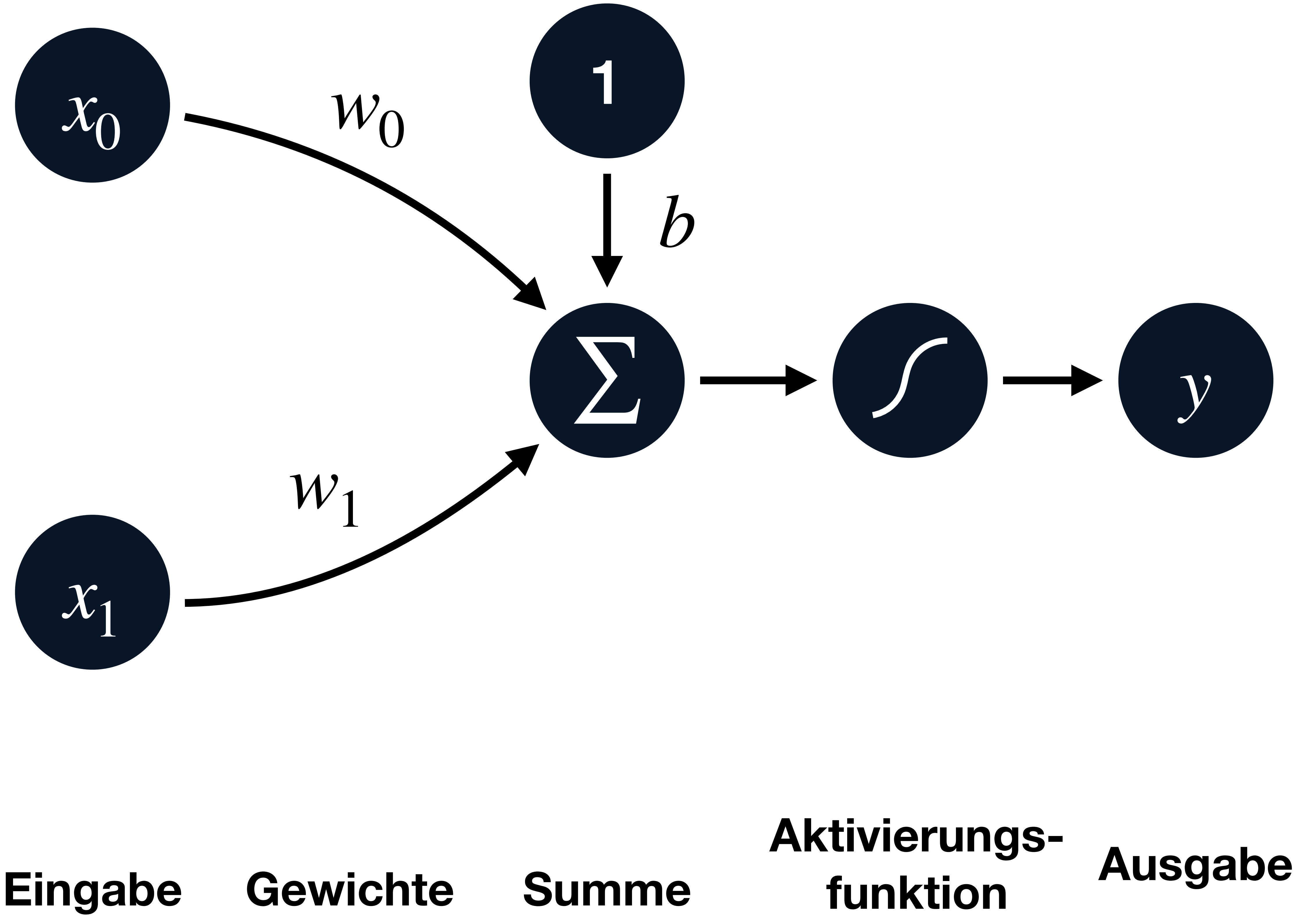


- Parameter:
  - $W = [3, -2]^T$
  - $b = 1$

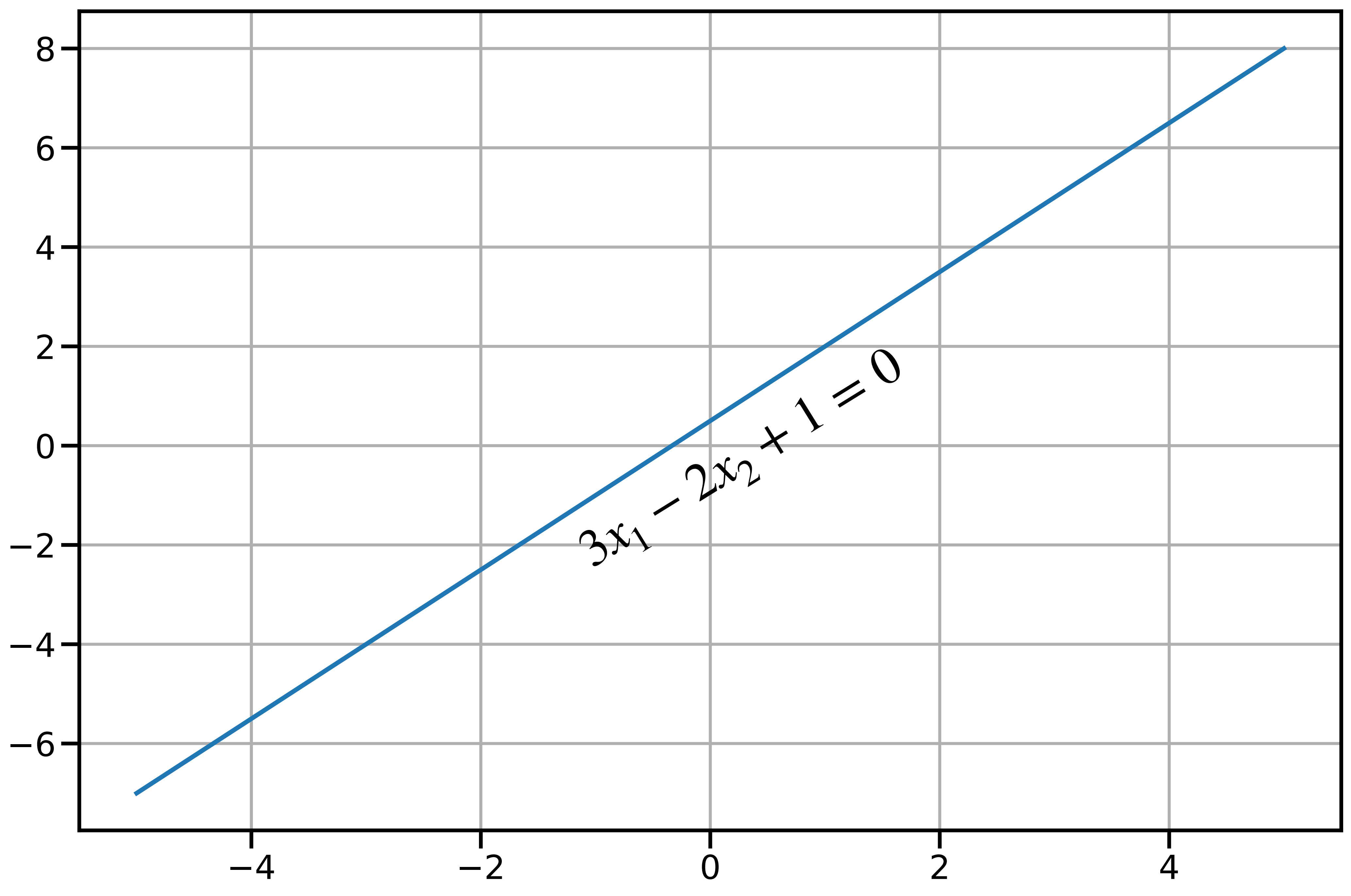
- Rechnung
  - $y = f(X^T W + b)$
  - $y = f([x_1, x_2][3, -2]^T + b)$
  - $y = f(3x_1 - 2x_2 + 1)$

Das ist einfach eine 2D-Linie!

Rechnung (Interpretation)

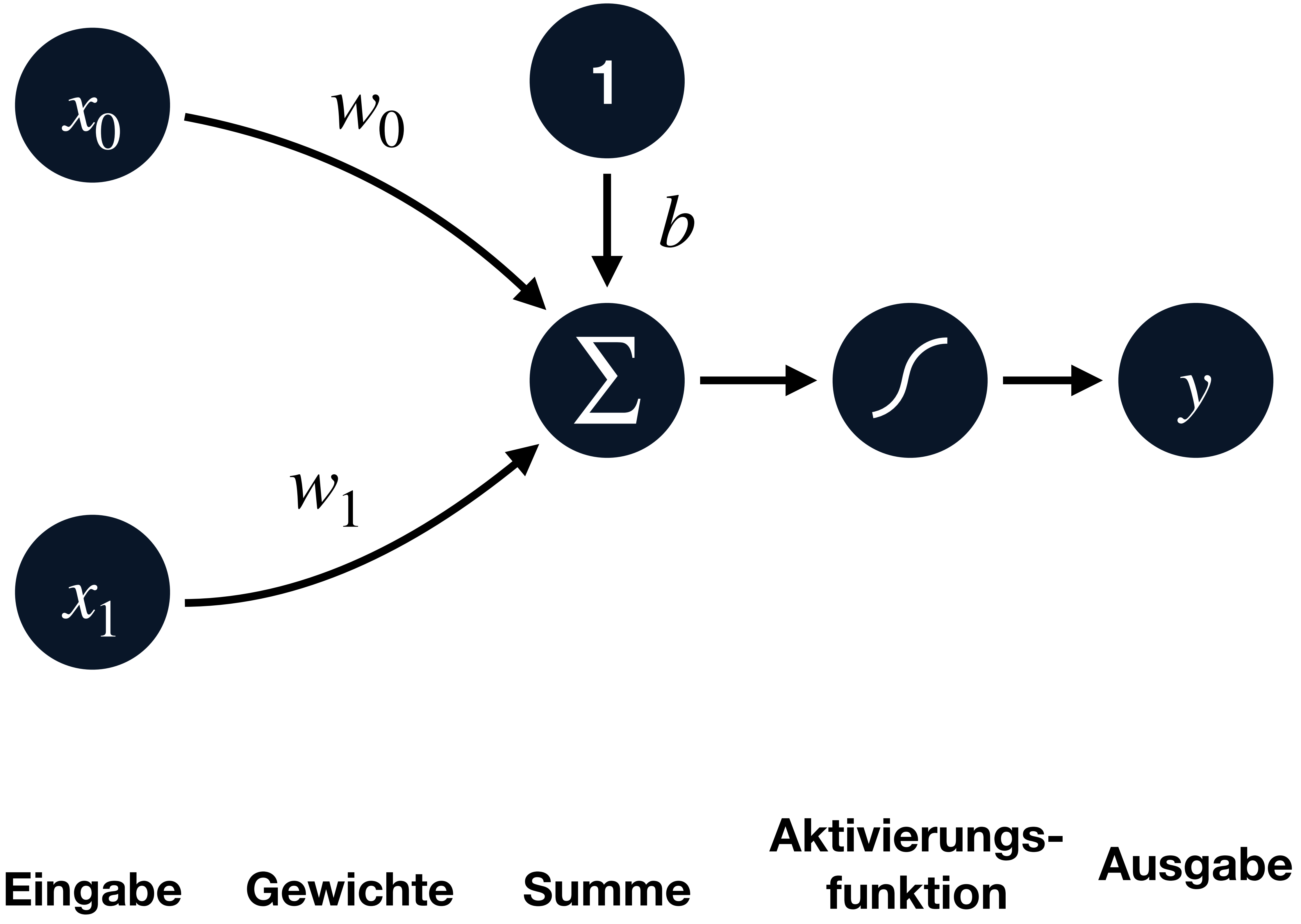


- $y = f(3x_1 - 2x_2 + 1)$

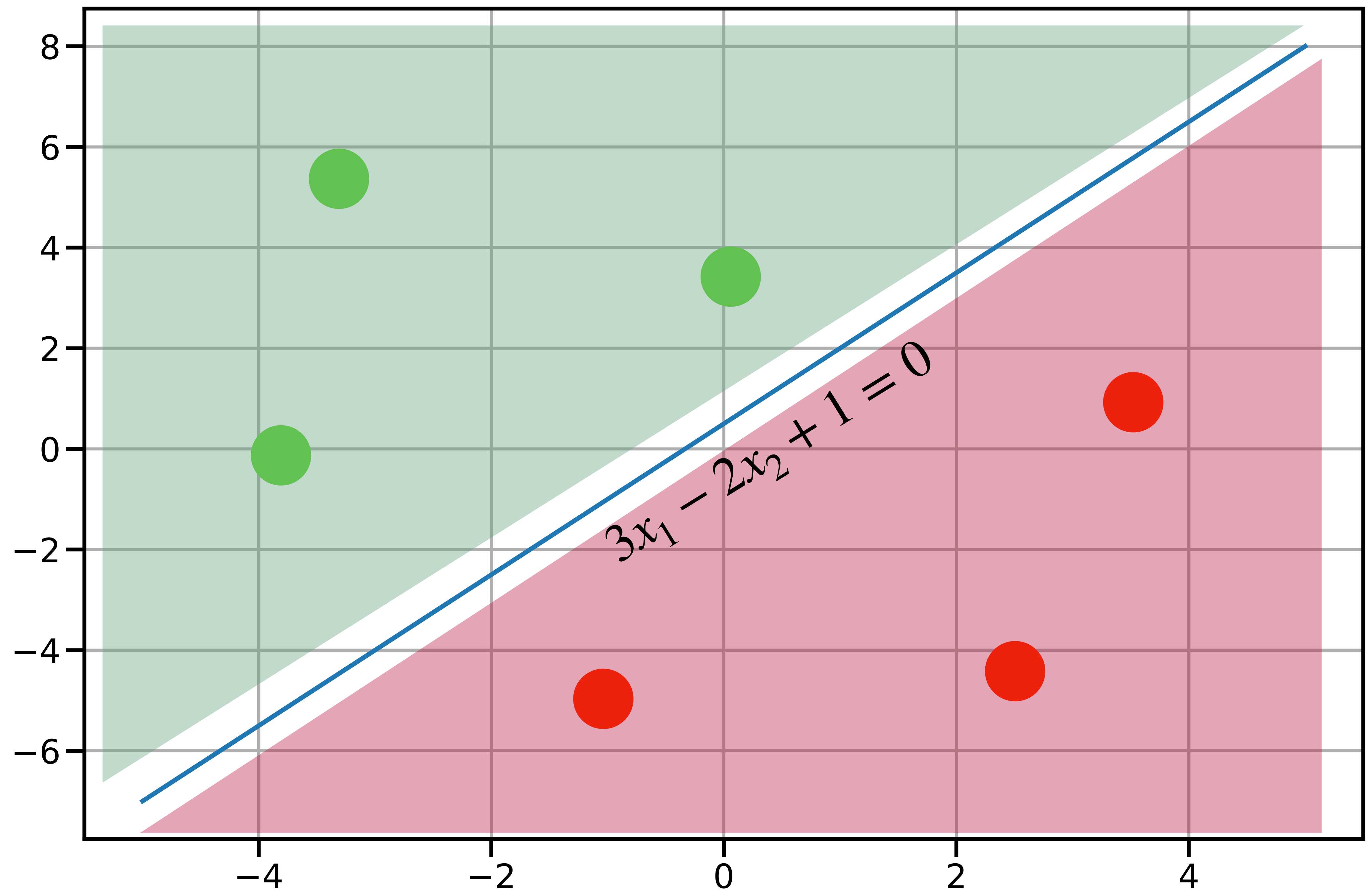


Perceptron: Durchlauf

Rechnung (Interpretation)



- $y = f(3x_1 - 2x_2 + 1)$
- Angenommen wir wählen die Aktivierungsfunktion $f(x) = \begin{cases} 1 & \text{wenn } > 0 \\ 0 & \text{sonst} \end{cases}$



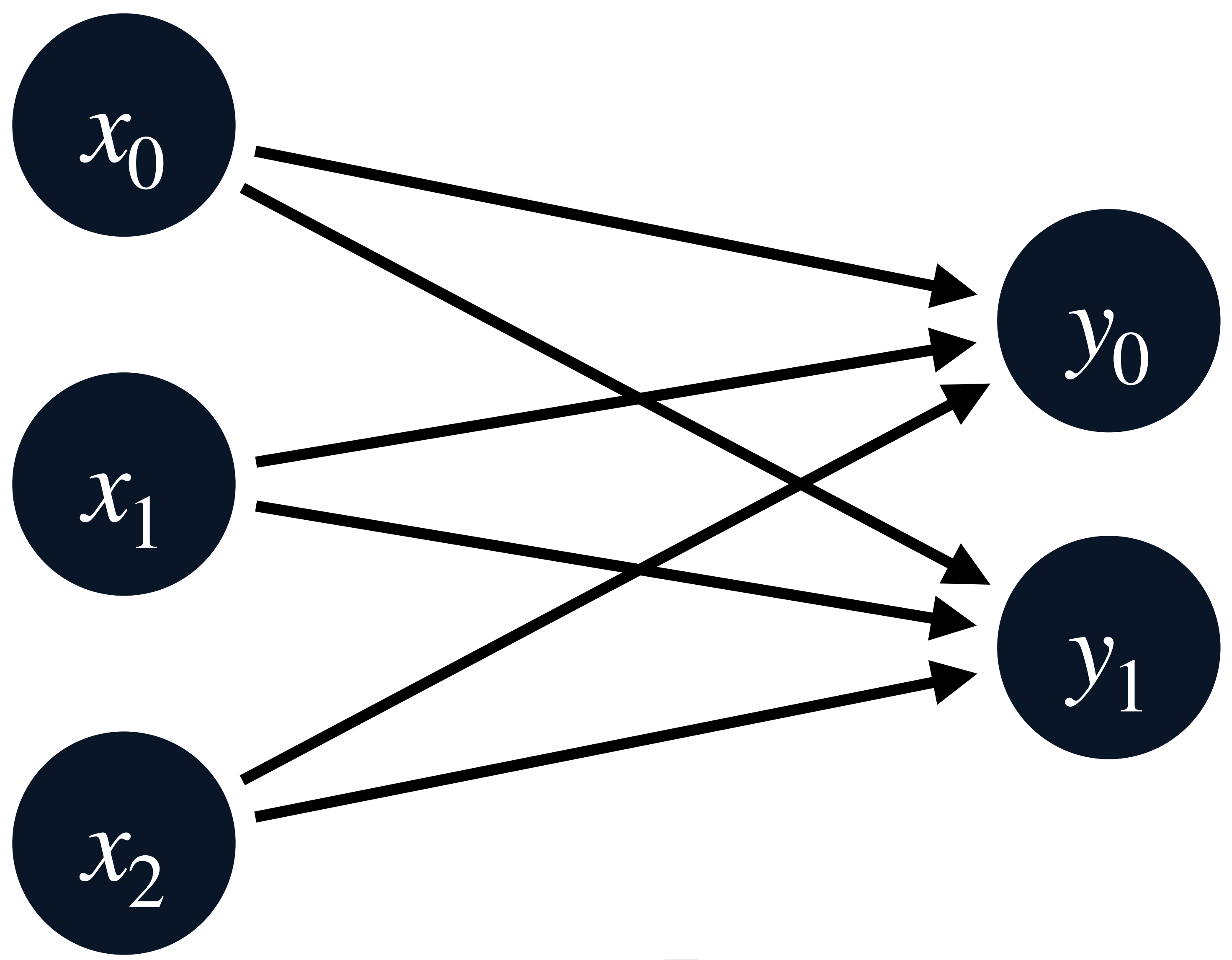
Achtung: Die Nicht-Linearität zeigt sich in der Farbe der Punkte!

Neuronale Netze mit Perceptrons

Multi Output Perceptron

Durch das hinzufügen mehrere Kanten können beliebig viele Output-Knoten erzeugt werden. Da alle Einheiten linear miteinander verbunden sind wird solch eine Kombination auch **Dense Layer** (häufig in der Literatur/ TensorFlow) oder **Linear Layer** (PyTorch).

Achtung: Wir stellen ab sofort das Netzwerk nicht mehr so ausführlich da wie zuvor. Die Funktionsweise ändert sich nicht!



$$Y = f(X^T W + B)$$

oder

$$y_i = b_i + \sum_j x_j w_{j,i}$$

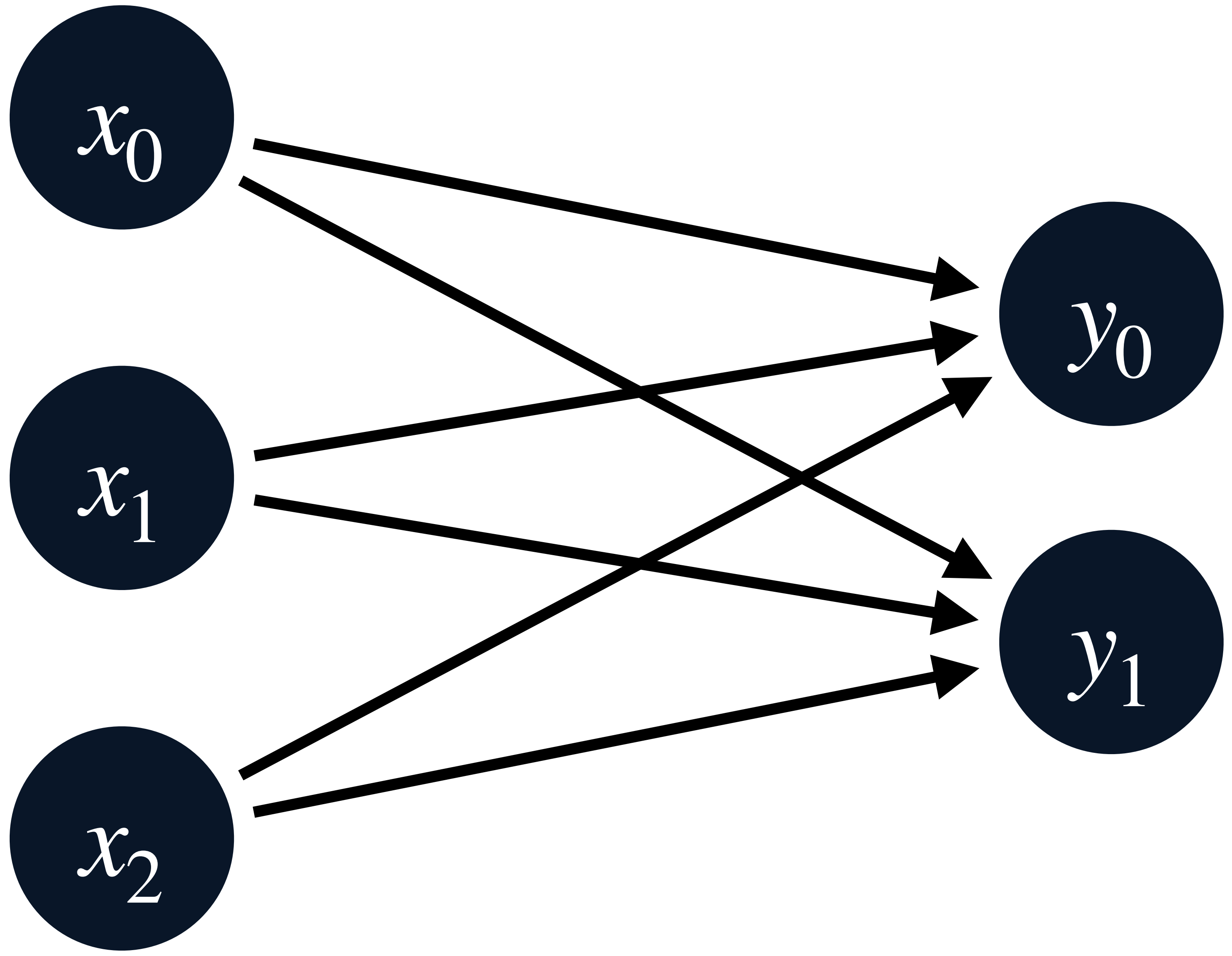
Multi Output Perceptron

Formel

$$Y = f(X^T W + B)$$

Eingabe & Parameter

$$X^T = \begin{bmatrix} x_0 & x_1 & x_2 \end{bmatrix} \qquad W = \begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \\ w_{20} & w_{21} \end{bmatrix} \qquad B = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$$



$$Y = f\left(\begin{bmatrix} x_0 & x_1 & x_2 \end{bmatrix} \begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \\ w_{20} & w_{21} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \right) = \begin{bmatrix} f(b_0 + x_0 w_{00} + x_1 w_{10} + x_2 w_{20}) \\ f(b_1 + x_1 w_{01} + x_1 w_{11} + x_2 w_{21}) \end{bmatrix}$$

Mit der Hilfe einer Matrixmultiplikation kann die Berechnung weiterhin in einem Schritt erfolgen! Das ist unabhängig von der Anzahl der Eingaben oder Anzahl der Ausgaben.

Implementierung in PyTorch

Achtung: Hier ist es einfach die Gewichtsmatrix zu transponieren. Das muss nur einmal beim Erzeugen des Layers passieren.

```
class MyLinearLayer(nn.Module):
  def __init__(self,in_features,out_features):
    super().__init__()
    self.weight = nn.Parameter(torch.randn(in_features, out_features))
    self.bias = nn.Parameter(torch.randn(out_features))

  def forward(self,x):
    return x@self.weight + self.bias

x=torch.rand(1,3)
MyLinearLayer(3,2)(x)
```

```
-> tensor([[ 0.8701, -1.1282]], grad_fn=<AddBackward0>)
```

Implementierung in PyTorch

Unser Layer kann noch mit einer beliebigen Aktivierungsfunktion kombiniert werden.

```
net=nn.Sequential(
    MyLinearLayer(3,2),
    nn.ReLU()
)

x=torch.randn(1,3)
net(x)
```

```
-> tensor([[0.6761, 2.6671]], grad_fn=<ReluBackward0>)
```

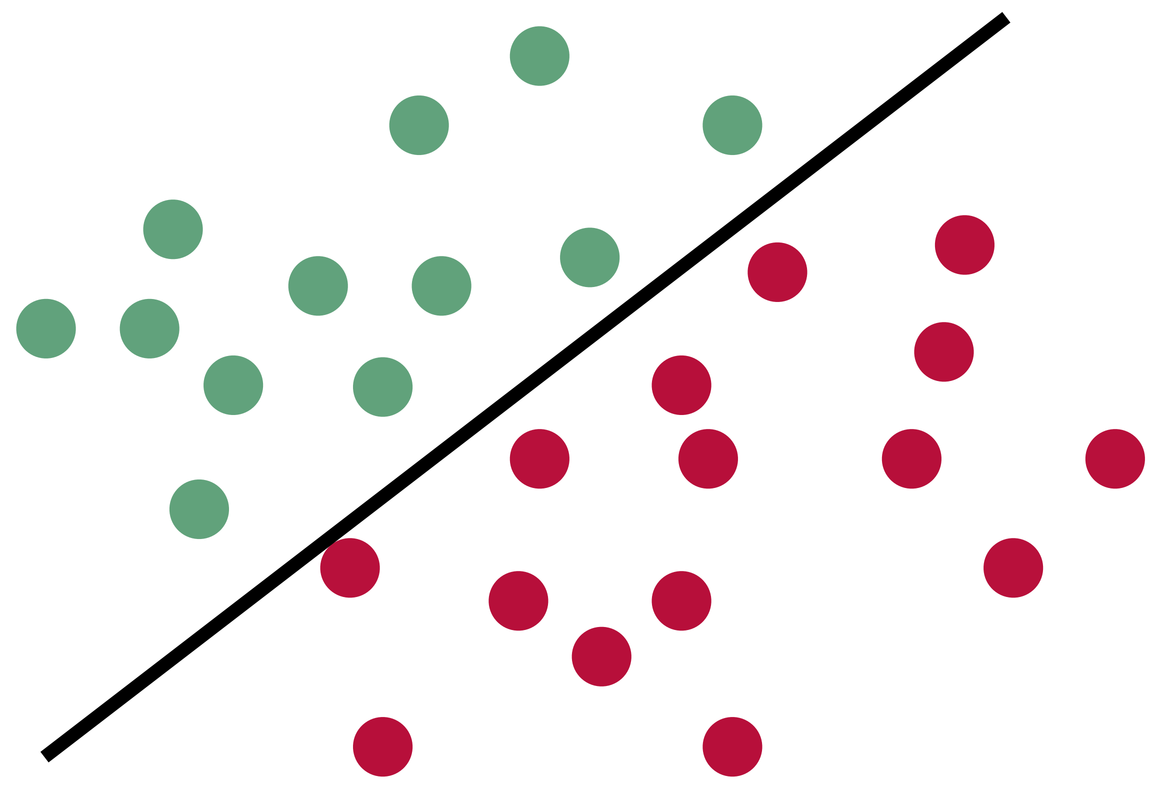
Implementierung in PyTorch

PyTorch bringt bereits eine eigene Implementierung des LinearLayers mit!

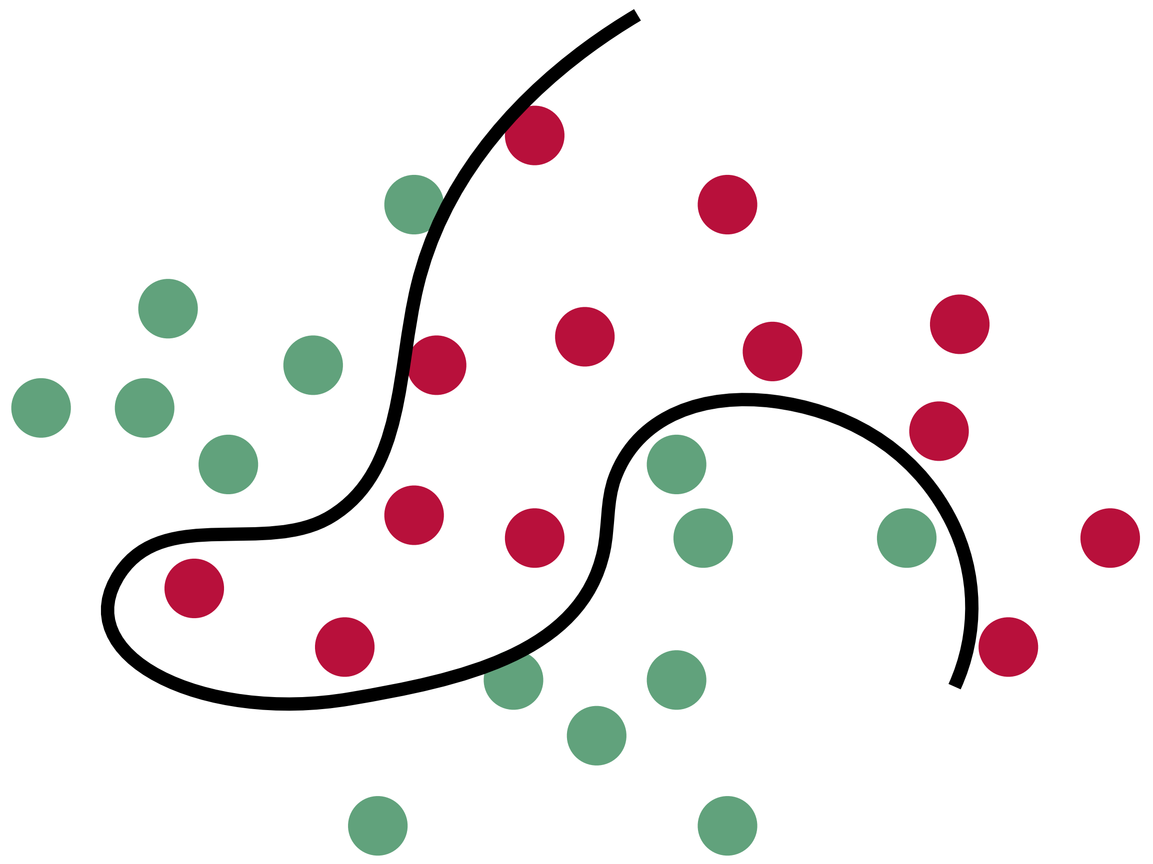
```
net=nn.Sequential(
    nn.LinearLayer(3,2),
    nn.ReLU()
)

x=torch.randn(1,3)
net(x)
```

```
-> tensor([[0.6761, 2.6671]], grad_fn=<ReluBackward0>)
```

Linear separierbare Klassen in $\mathbb{R}^2$



Nicht linear separierbare Klassen in $\mathbb{R}^2$

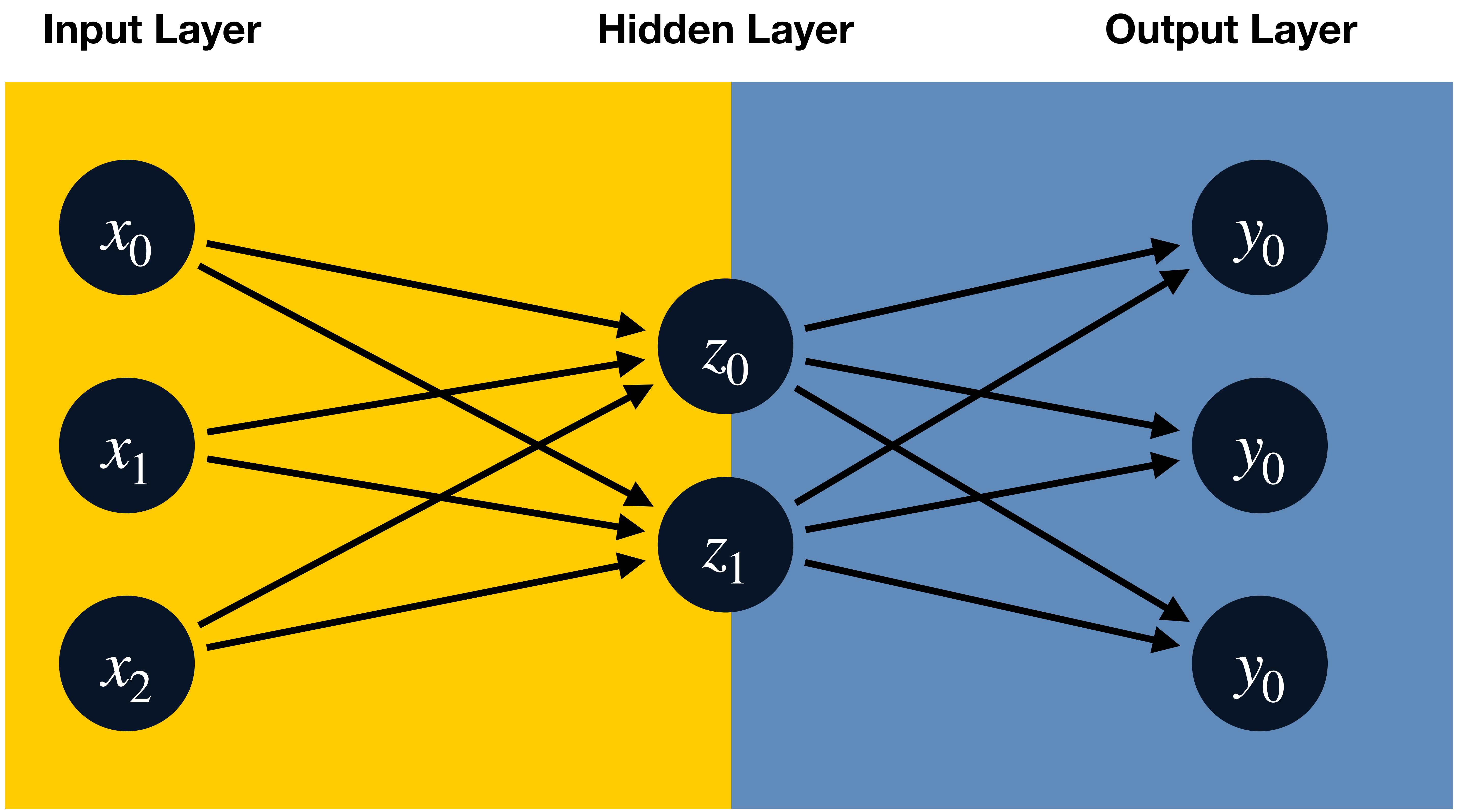


Gab es dafür nicht die Aktivierungsfunktionen?

Ja, allerdings ist die Komplexität der Aktivierungsfunktion in der Praxis begrenzt und auch nicht beliebig. Das Neuronale Netz soll aber möglichst beliebig komplexe Strukturen erlernen können.

Lösung: Multilayer Perceptrons oder auch Neuronale Netze

Multi Layer Network



$$Z = f_0(X^T W_0 + B_0) \qquad Y = f_1(Z^T W_1 + B_1)$$

oder zusammen

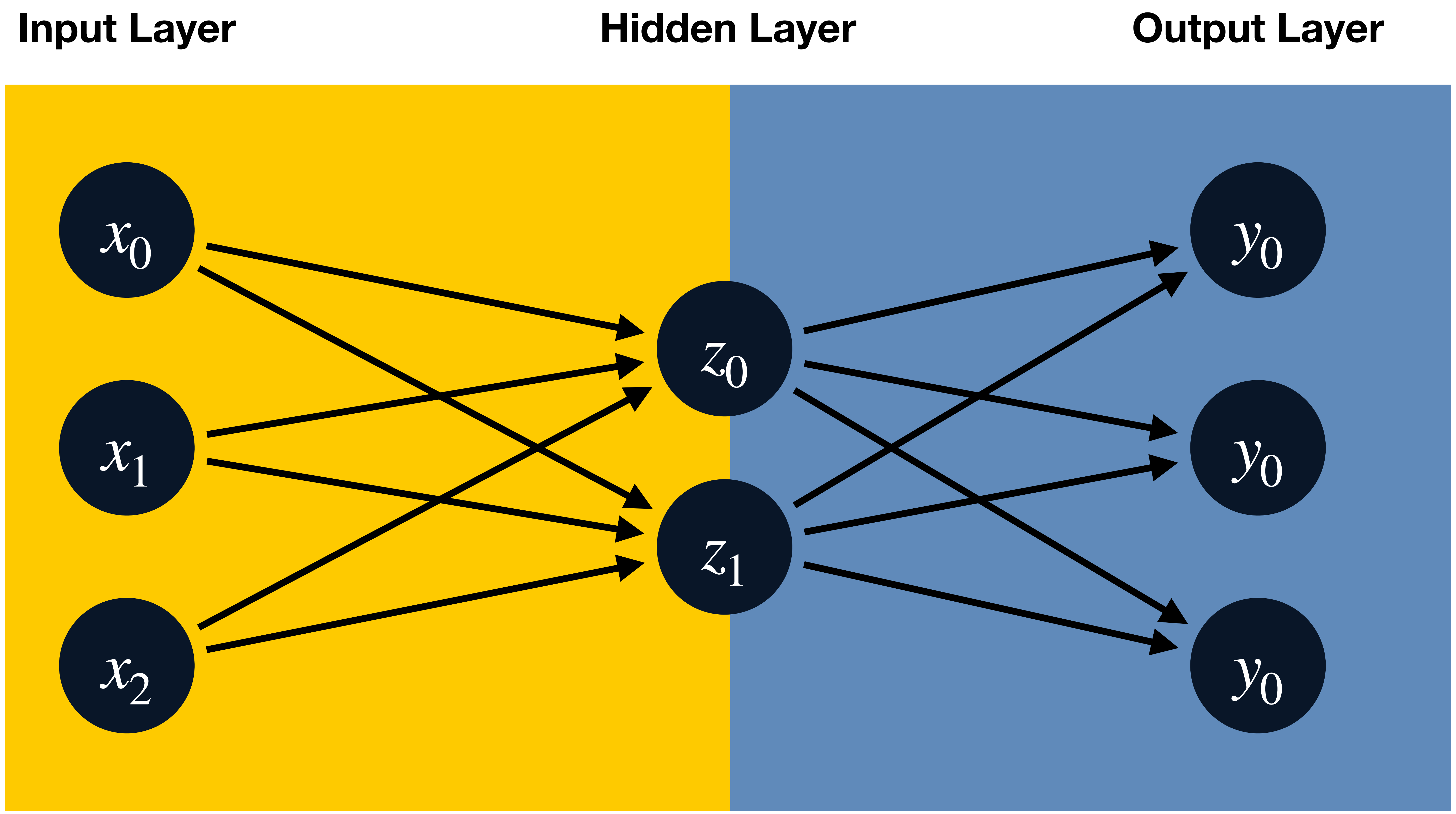
$$Y = f_1\left(\left(f_0(X^T W_0 + B_0)\right)^T W_1 + B_1\right)$$

Implementierung in PyTorch

```
class MyNeuralNetwork(nn.Module):
  def __init__(self):
    super().__init__()
    self.net=nn.Sequential(
        nn.Linear(3,2),
        nn.ReLU(),
        nn.Linear(2,3),
        nn.ReLU()
    )

  def forward(self,x):
    return self.net(x)

x=torch.randn(5,3) # 5 Datenpunkte gleichzeitig, mit 3 Features
net=MyNeuralNetwork()
net(x)


-> tensor([[0.1169, 0.1805, 0.1360],
        [0.4028, 0.3244, 0.6854],
        [0.3915, 0.3559, 0.6023],
        [0.0301, 0.0038, 0.1898],
        [0.0962, 0.0607, 0.2777]], grad_fn=<ReluBackward0>)
```

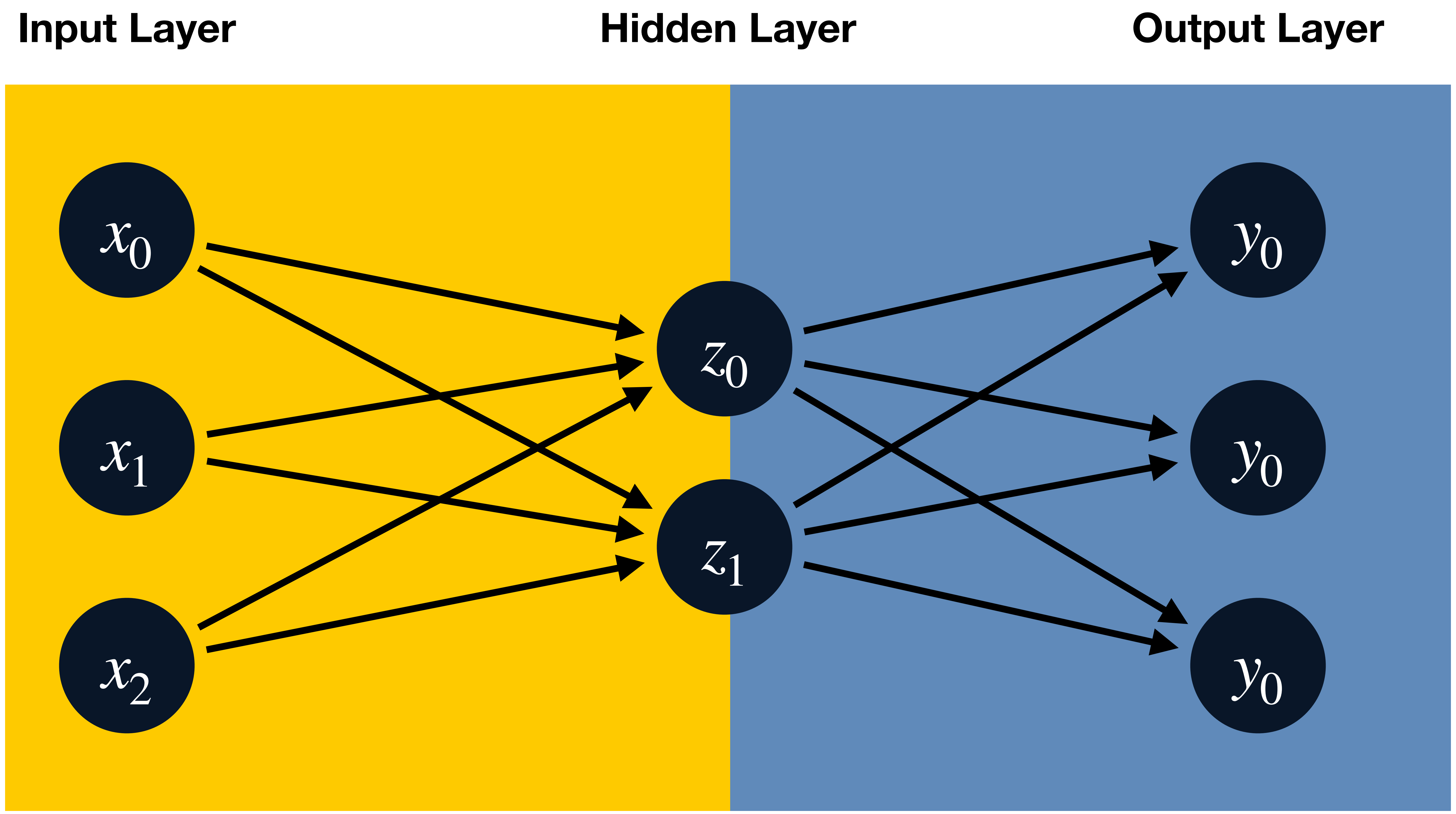
Größere Neuronale Netze werden üblicherweise einfach als eigenes Modul implementiert.
Diese kann später auch in anderen Modulen wiederverwendet werden!

Multi Layer Network



- Wieviele Parameter hat das Netzwerk?

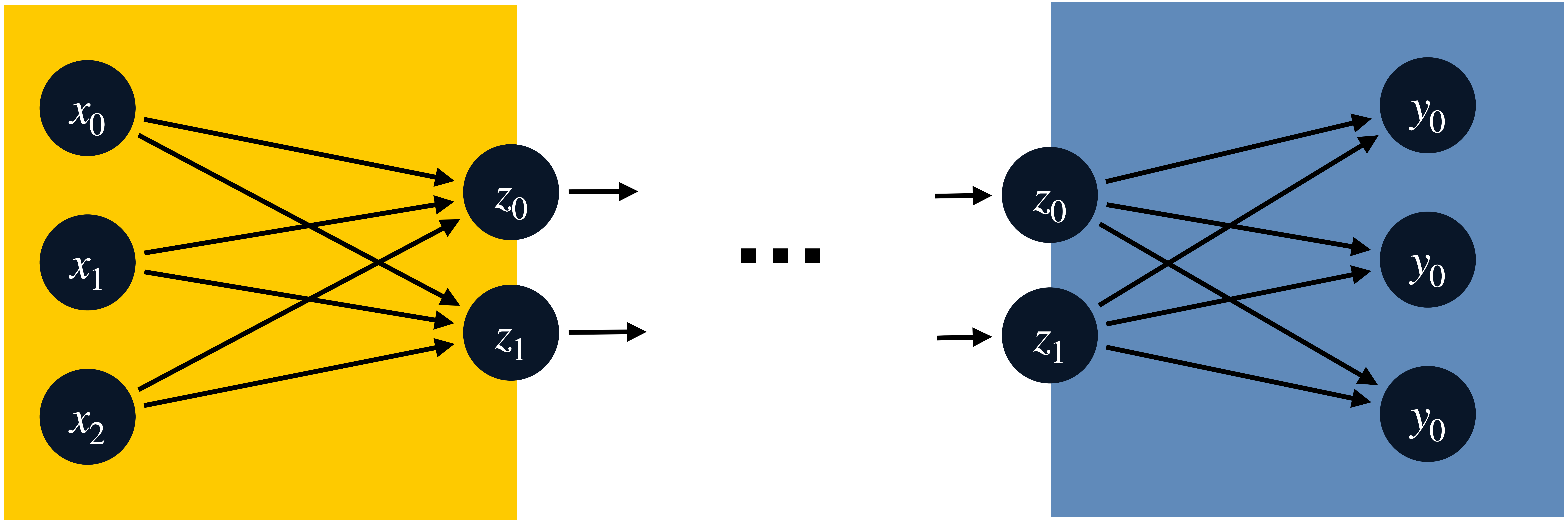
Multi Layer Network



- Wieviele Parameter hat das Netzwerk?
- Antwort: 17
  - 6 Werte für die Gewichtsmatrix des Hidden Layers
  - 2 Werte für den Bias des Hidden Layers
  - 6 Werte für die Gewichtsmatrix des Output Layers
  - 3 Werte für den Bias des Output Layers

Deep Neural Network

Input Layer **Hidden Layers** **Output Layer**



$$Z_1 = f_0(X^T W_0 + B_0)$$

$$Y = f_n(Z_{n-1}^T W_n + B_n)$$

Implementierung in PyTorch

```
class MyDeepNeuralNetwork(nn.Module):
  def __init__(self, layers=[3,2,5,6,7,2,3]):
    super().__init__()
    self.net=nn.Sequential(*[
      nn.Sequential( # Template für ein Layer
          nn.Linear(layers[i-1],layers[i]),
          nn.ReLU()
      )
      for i in range(1,len(layers))
    ])

  def forward(self,x):
    return self.net(x)

x=torch.randn(5,3) # 5 Datenpunkte gleichzeitig, mit 3 Features
net=MyDeepNeuralNetwork()
net(x)
```

Dieses Netzwerk hat 5 Hidden Layers!

```
-> tensor([[0.3204, 0.4431, 0.0000],
        [0.3181, 0.4510, 0.0000],
        [0.3186, 0.4501, 0.0000],
        [0.3204, 0.4430, 0.0000],
        [0.3203, 0.4439, 0.0000]], grad_fn=<ReluBackward0>)
```